L8-addrsing modes:-

five common addressing modes
register names or memory locations for operands.

1) Immediate:

```
addi  $t0, $t1, 5
```

2) Register:-

```
add  $t0, $t1, $t2
```

3) Base address:-

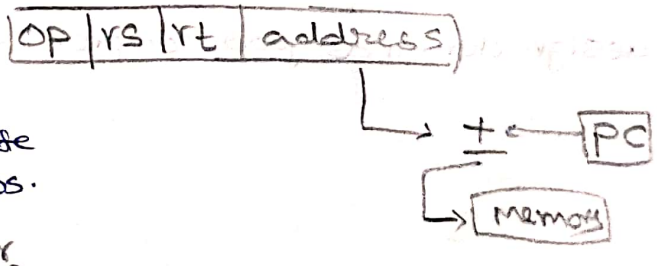
```
lw  $t1, 4($t2)
```

address in t_2 + 4 is our required!

```
lw  $t1, ($t2)
```

4) PC-relative:-

```
beqz  $t0, goEnd
```

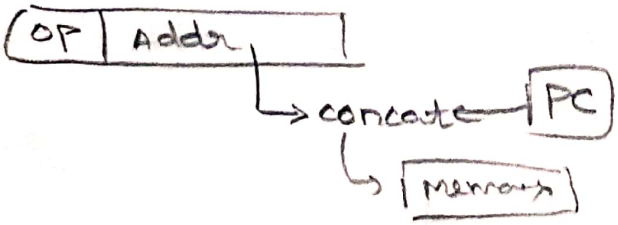


isn't appended?
Nah!

assembler will calculate the rel. pos.

PC += Addr$_{18}$ ✓

5) Pseudo-direct:-




```
j  forLOOP
```

assembler will calculate this label's addr.

J, JAL, JR } will not give relative to PC.
will give addr$_{28}$

PC = PC$_{31:28}$ :: Addr$_{28}$

BEQZ, BEQ, BNE, BGTZ } will give relative to PC

PC += OFF$_{18}$

L9- ISA and microarchitecture:-

* Microarchitecture is implementation of ISA operations.

Like for "add (.)(.)(.)"; should we ripple (or) carry-lookahead?

→ company chips differ in their microarchitecture; but respect the ISA (x86, ARM, MIPS, RISC-V)...

* ISA provides:-

Instructions
- Opcodes
- Addressing modes
- Types & format
- Registers
- Access control: user/OS

Memory
- Address space
- Addressability
- Alignment

ISA must satisfy requirements of software-
- assembler
- complier
- OS.

rest of CS305 after ISA will be caches, memory controllers, Branch predictors, Prefetchers.

woah!

*
registers ⟶ ISA
Cycles to register access ⟶ Microarchitecture
register width ⟶ ISA
Instruct^n to access memory ⟶ ISA
cycles to read memory → Architecture.

* Where to place our ISA specifications?

- close to high level language? → small semantic gap; complex instructions (an Opcode for quicksort)

  CISC → complex instruction set computer.
  like x86. (x86 is CISC with RISC mysteries)

- closer to hardware? → large semantic gap.
  simple instructions.

  RISC → reduced instruction set computer.
  like MIPS.

L10:- World of ISAs

x86: Intel, AMD : PCs, Servers

ARM: Qualcomm, Apple, Samsung : Mobiles
↳ Advanced RISC

RISC-V: opensource ISA.] meaning, no patent filed.

* subtle differences in x86 ISA:-

1) x86 arithmatic/logic instructions:
   one operand should act as source and destination.

   add $t0, $t1

   t0 ← t0+t1

2) one operand can be in memory! not allowed for MIPS
   Programmer: Awesome!
   Instruction: NOO! Size is too large
   add $t0, ($t1)

3) NO fixed length for instructions. (Booo!)

   ↳ NOW code is smaller, but decoding is tough!.

<trivia in slides>

Fallacies:

- CISC instructions provide higher performance.
- Assembly code provides higher performance.

things are not that straightforward

* RISC was motivated over CISC by memory stalls.
  - (when fetching memory, no other arithmetic/logic work can be done in complex instruction)
  - But RISC enables the compiler to optimize the code by fine-grained parallelism to reduce stalls.

* ARM has some thumb instructions to reduce code density.

Normal inst. (32bit) → Thumb (16 bit).

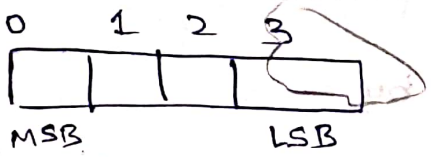
less code written wrt original situ.

L11- Endianess and Alignment:-

Endianess : Byte ordering within a word.

1) **Bigendian** : word address = MSB of word

MIPS



Big end

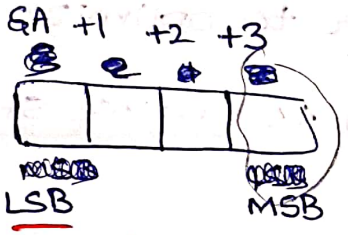
"The usual"

int A = 0xABCD1234

&A: | AB | CD | 12 | 34 |

2) **Little Endian**: word address = LSB of word

x86



Little endian

&A: | 34 | 12 | CD | AB |

Eg:

```
unsigned int i = 0x12345678;
char *c = (char*) &i;
```

1 byte length — word address.

```
printf ("%d", *c);
```

Littleendian : 78 ✓

Bigendian : 12

→ **Alignment:-**

MIPS doesnot allow unaligned access.

lw cant use. but lh, lb okay!

lw → load word

lh → load half

lb → load byte

x86 allows.

> Alignment for fast data fetching!
> Otherwise 2-times memory requests

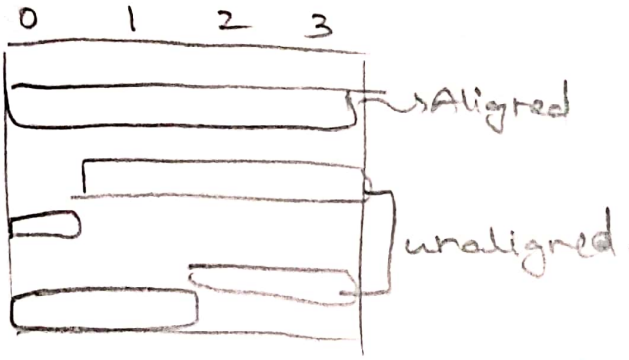
* compiler generates aligned/unaligned access....

if "lw A" then A%4==0

if "lh A" then A%2==0

if "lb A" then A%1==0 (always true)

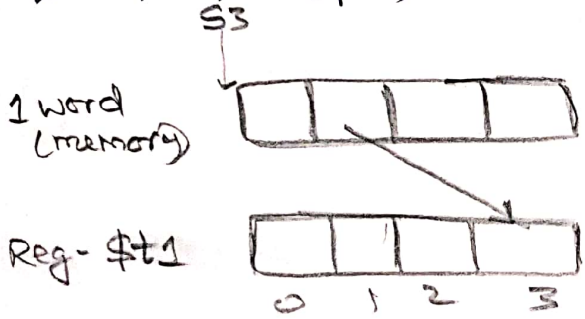
↳ Load byte



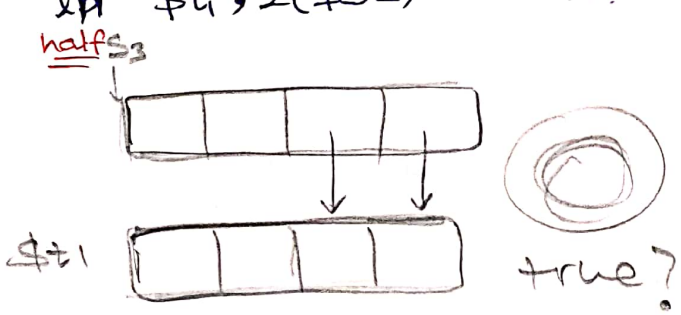
→ Memory instructions & Alignment network :-

LOADS & STORES need a Alignment network that makes sure loaded (written) data is aligned.
(word/half/byte)

lb $t1, 1($s3)

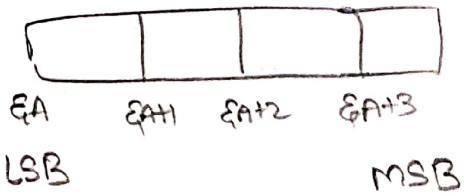


lh $t1, 2($s3) ...

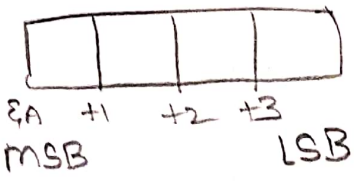


* Quiz question based on endianess :-
cool now!

in Little endian :-



Big endian :-



L12: Single cycle CPU (processor 101)
&
L13

* All operations in single cycle each. → so, instructions need to be simple enough.

* Hence, clock cycle will be dependent on the longest instruction's delay. (critical path)

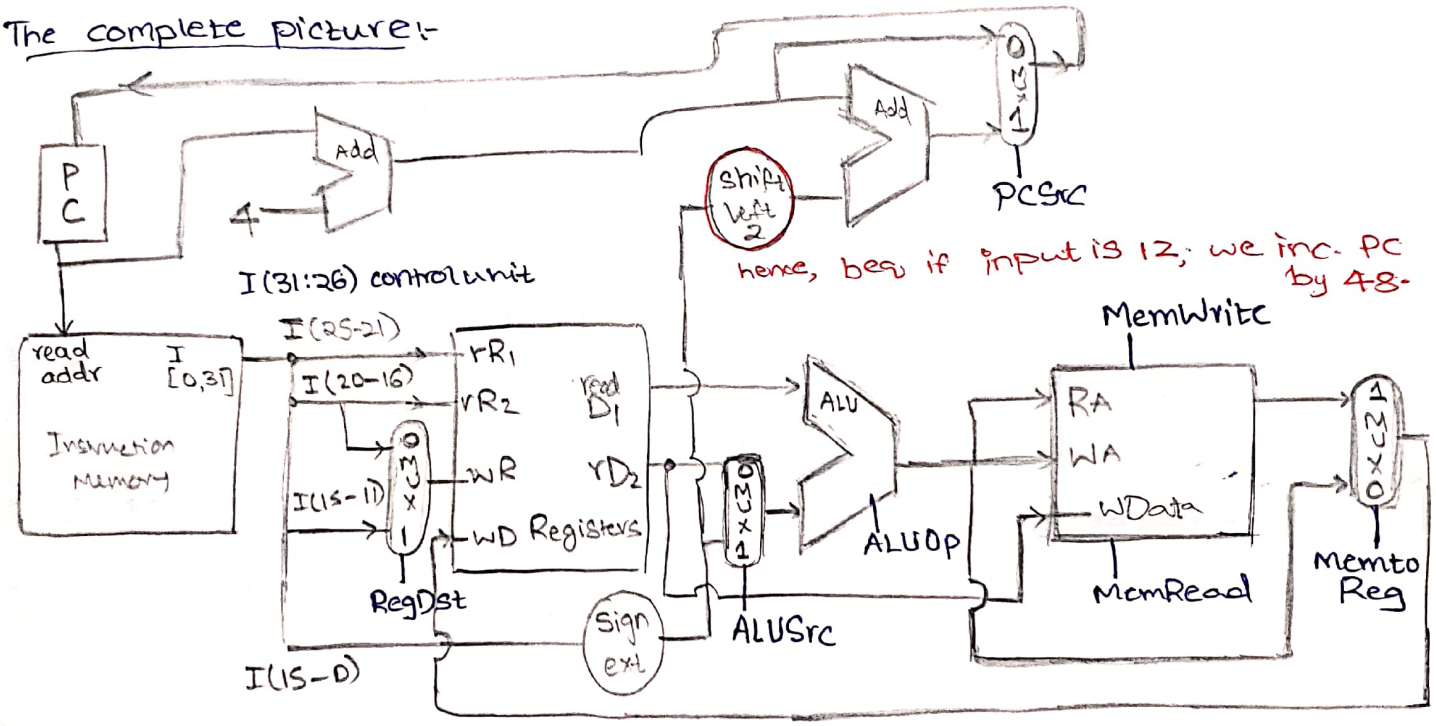
rate-determining path in a instruction execution.

• Datapath & Control unit.

"Anything that stores data or operates on data". "manipulates data"

"Guides" data flow.

* The complete picture:-



here, beq if input is 12, we inc. PC by 48.

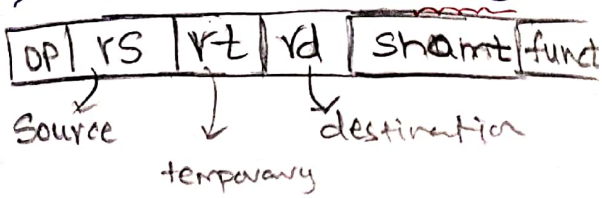
* ALUSrc: asserted: lower 16 bits of instructions
non asserted: output from 2nd reg.

* MemtoReg: asserted: data mem to reg.
deasserted: ALU to reg.

* PCSrc: asserted: branch
deasserted: PC+4

* RegDst: asserted: rd
deasserted: rt

this order different from MIPS.



MIPS:- add $dest, $r1, $r2.

* Shamt- Shift amount

in MIPS, all R instructions have opcode 0 (P&H book ref-)
but different funct.

sll is R type!!!
(shift left logical) so, ALUSrc not used.

sll $t0, $t0, 1 } ALUSrc is 0.
Shamt. quiz error

L-14: Iron law:-

$$\text{Time/program} = \frac{\text{Instructions}}{\text{program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \text{Time/cycle}$$

- Source code
- compiler
- ISA

- ISA
- microArch.

- microArchitecture
- Technology.

L-15:16:17 : Empirical evaluation:-

* we see performance as,
  - Latency (execution or response time) → additive
  - Bandwidth (not additive)
    no. of tasks/unit time

$$\text{performance} = \frac{1}{\text{latency}}$$

* Latency lags bandwidth. Bandwidth hurts latency.
  (but not decreases!)

* better latency helps bandwidth.
  B.W usually hurts latency!!

→ Amdahl's law:- (common case; make it fast)

$$\text{Speedup}_{\text{overall}} = \frac{\text{Time Execution}_{\text{old}}}{\text{Time Execution}_{\text{new}}} = \frac{1}{(1 - f_{\text{enhanced}}) + \frac{f_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

- Small speedup on large fraction > large speedup on small fraction

→ Evaluation:-

* Gonna compare proc. A & B by running programs.
  which programs?
  > The ones that you care & ought to occurs in processors usage.

Benchmarks:

- SPEC CPU 2017
- Cloudsuite
- PARSEC
- Mobilebench
- Geekbench
- ML perf
- Graph perf.

Pitfalls of benchmarks:-

- not representative of all features.
  if workload is I/O bound, CPU spec is useless.
- Too old benchmarks.

* Okay! lets create a chip & run these benches.

Pfft! lol!
use simulators.

- Functional Simulator:-
  - used to verify correct.
  - worthless to check performance.

- Performance Simulators:-

  i) Trace-driven:

  ii) Execution-driven:

* So, pick a relavant suite of benchmark

  2) measure IPC of each program

  3) Summarize performance using

~~AM~~ (GM) ~~HM~~ }

AM if 100
0.01
is present;
gives unfair
results

→ Energy and power:-

* Power efficency = performance/watt

  Energy efficency = performance/Joule

* Dynamic power ; Static power

  during running instructions

  even when transistor is off

L18- Multi Cycle CPU:-

* in single cycle. why every process should wait for longest proc time!

So bad design!

Multicycle processor!

Now, even though longest = T
others ≤ T.
cool.

now CPI increases.
even though cycle time decreases drastically.

1 CPI, C = 8ns → 4 CPI, C = 2ns.

* can we have both

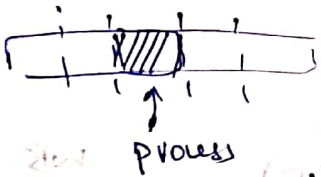
CPI = 1, C = 2ns?

Pipelining! yea.

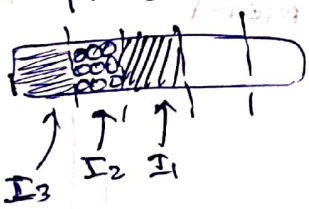
hm, we are leaving some part of processor empty, when running an instruction.

fill those too.

hm!



pipelined:



Eg: normal covid schedule

S1 : 15 min
S2 : 10 min
S3 : 5 min
S4 : 20 min
S5 : 10 min

(one after another completes)

1 person per 1 hr

pipelined!

S1 : (20 min)
S2 : (20 min)
S3 : (20 min)
S4 : 20 min
S5 : (20 min)

(in a line proceed)

3 per hour.

even though Single person latency = 100 min.

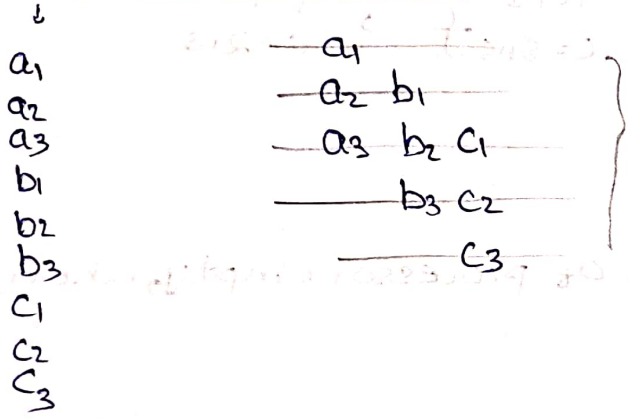
L19,20,21 - Pipelining:-

* single cycle CPI: 1 cycle time: long

Multicycle CPI > 1 Short cycles

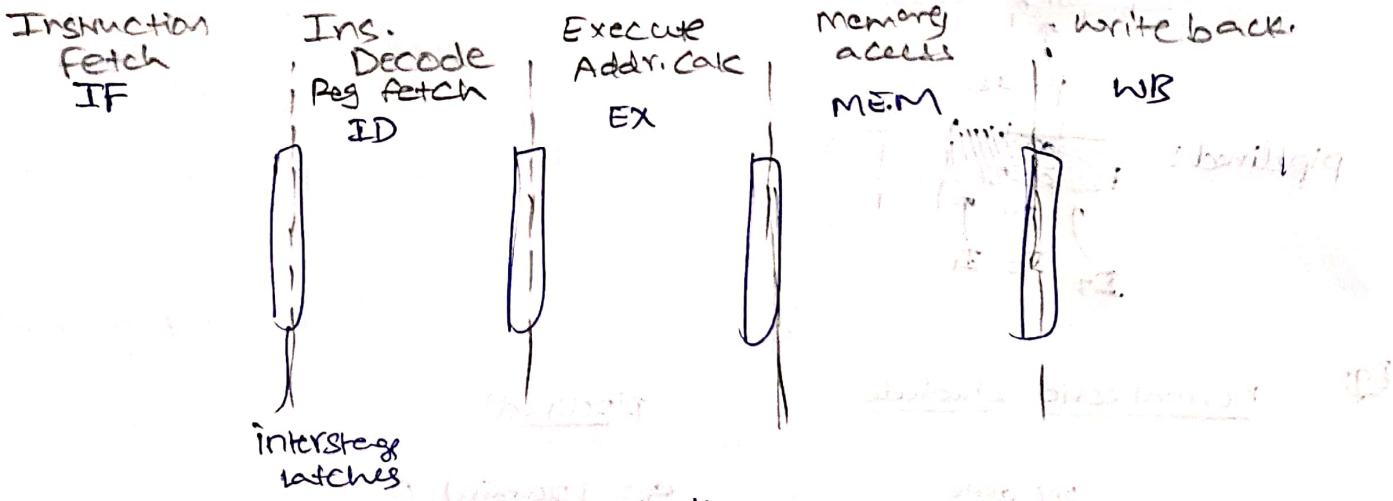
Pipelined CPI: 1 Short cycles (but throughput increases latency worsens)

* multicycled vs. pipelined:-



ila resource utilization continuous; instead of staying idle.

→ vanilla 5-staged pipeline:-



5 stage pipeline

* For a K-staged pipeline, with N instructions

1st instruction: K cycles

next instructions: 1 cycle each.

net = K + N - 1

∴ avg CPI = $\frac{K+N-1}{N} = 1 + \frac{K-1}{N}$.

* single cycle design vs pipelined.

	ifetch	Decode	Execute	Memory	Writeback	Total time
Load	200	100	200	200	100	800
Store	200	100	200	200	-	700
Add	200	100	200	-	100	600
branch	200	100	200	-	-	500

in Single cycle:-
clock = 800 ns

∴ latency for the 4 net = 3200ns

for pipelined:-
clock = 200ns

net time taken = 1000 + 3(200)
= 1600ns

⇒ 2x speed up.

* if 10^9 instructions: $\frac{1000 + 10^9(200)}{800 \cdot 10^9} \sim$ 4x speed up.

pipeline latches may add overhead 200 + 10ns.
≈ 3.8X speed up

→ Dividing datapath into pipelines:-

$t_{IM} = 10$
$t_{DM} = 10$
$t_{ALU} = 5$
$t_{RF} = 1$
$t_{RW} = 1$
} combine stages. NO loss in perf.

Eg:

	unpipeline	pipeline	speedup
10, 10, 5, 1, 1 4-Staged pipeline	27	10	2.7
5, 5, 5, 5, 5 4-stated pipeline	25	5+5	2.5
5-Stage PP	25	5	5// for k-stage; k is max speedup

L22: Instruction Pipeline Hazards:-

- Structural Hazard → Anything that prevents an instruction to move forward in a pipeline.
- Data hazards
- Control hazards

1. Structural: two instructions want to access the same physical structure at once.

Eg: if we have a unified memory; rather than instruction mem & data mem.

> what about the registers?

can we read and write in same cycle?

Yes! Since the edge triggering property. whatever !

* Structural Hzds are highly infrequent.

2. Data hazards:-

* Hazards due to data dependencies.
* Current instruction depends on the result(data) of previous instructions.

a) Read after Write:- RAW

add R1, R2, R3
Sub R2, R4, R1

True dependency

b) Write after Read:- WAR

Add R1 R2 R3
Sub R2 R4 R6

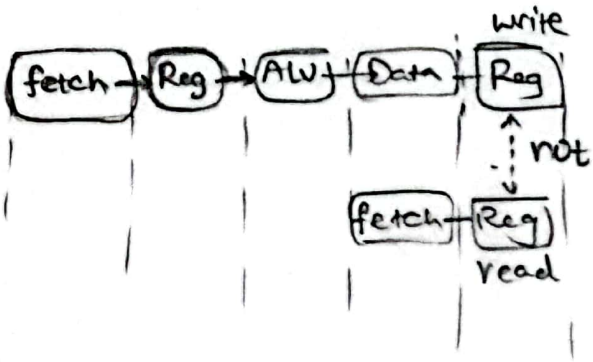
Anti Dependancy

c) Write after write:- WAW

add R1 R2 R3
or R1 R4 R5

Output dependancy → not possible in vanilla 5-stage pipeline.

Slide 10:



* महत्वपूर्ण point!
Such a nuisance!

the writes happens at clock rise. This is reflected in read.

* This last potential hazard can be resolved by the design of register file hardware:

we assume that write is in the first half of clock and read in the second half, so the read delivers what is written.

3. Control hazards:-

* hazards that arise from jump/branch instructions (those which change PC).

Eg:

10: Beq r1, r3, 36 goto 50

14: And r2, r3, r5 ✗

18: or --- ✗

22: and --- ✗

50: ✓

Wat to do to these 3 instructions?

- Instructions cannot move forward, & must wait for hazard to get resolved.

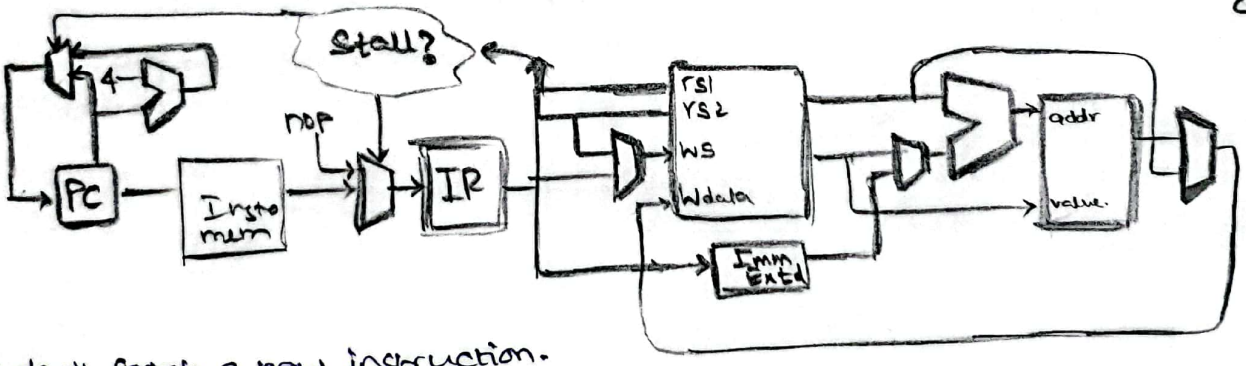
- Pipeline stalls.

!

Air Bubbles in pipeline.

* In a stall; PC is not updated to PC+4.
IR is not updated. (the same as prev. instruction)
deassert the control signals. or nop instruction!

* Implementing a Stall:- in a stall; PC, IF/ID are frozen & ID/EX is set to all 0 controls



> dont fetch a new instruction.
> don't change PC.
> insert nop into IR.] compiler way
no operations.

Hazard detector : ID stage
forwarding unit : EX stage

* Two ways of inserting stalls:-

1) compiler can insert nops

2) Microarchitecture has a hazard detector & gives a stall signal.

* Eg for NOP:

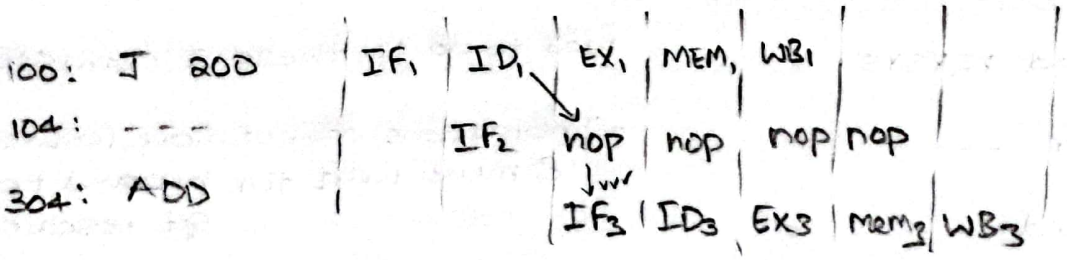
sll $0 $0 (in MIPS)

NOPs for control hazard.

How for data hazard?

* Jump instruction → PC value fetched in ID Stage.

only one NOP instruction.



* for branch instruction, 2 NOPs needed....
in EX Stage we get next PC.

* Hence

$$\text{speedup} = \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} = \frac{\text{CPI unpipelined}}{\text{ideal CPI} + \text{stalls/Instruction}}$$

① for balanced pipeline.
(meaning; all stages are equal time).

stalls affect speedup

L23: Hazards detection & mitigation:-

→ data hazard detectors:- conditions.

- Execute to decode:

  EX/MEM. Reg Rd = ID/EX. Reg Rs
  ALU
  or
  ID/EX. Reg Rt

- memory to decode:

  MEM/WB. Register Rd = ID/EX. Reg Rs
  Load
  or
  ID/EX. Reg Rt

write dst.

op	Rs	Rt	Rd	Shamt	funct

read1 read2

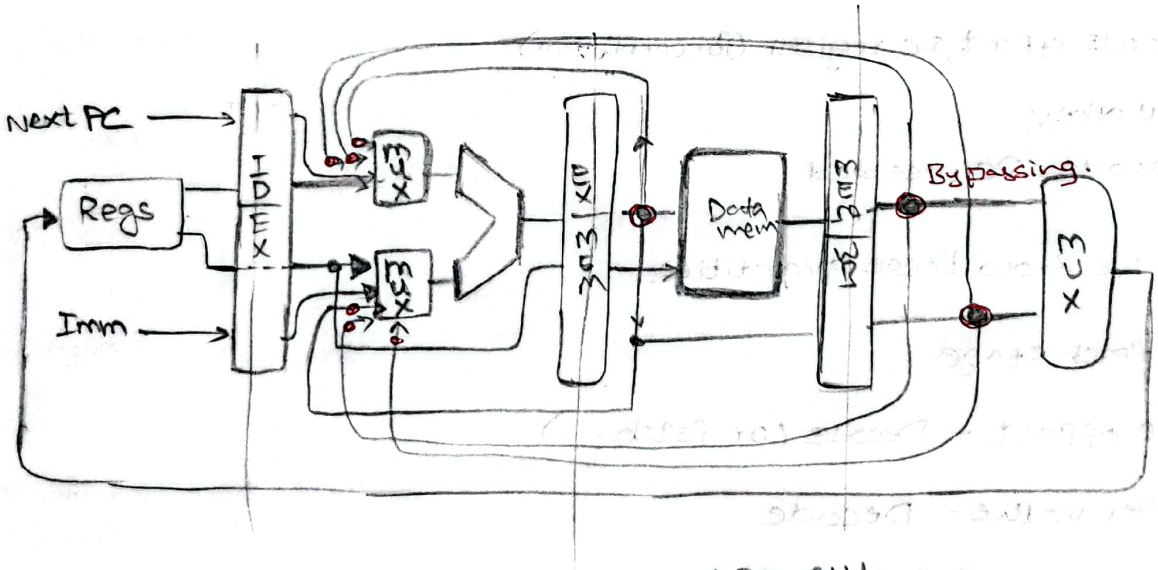
} cases of hazard.

what about instructions that dont write into registers?

→ Bypassing:

PPA-2: logic of hazard detector in 5-stage pipeline.

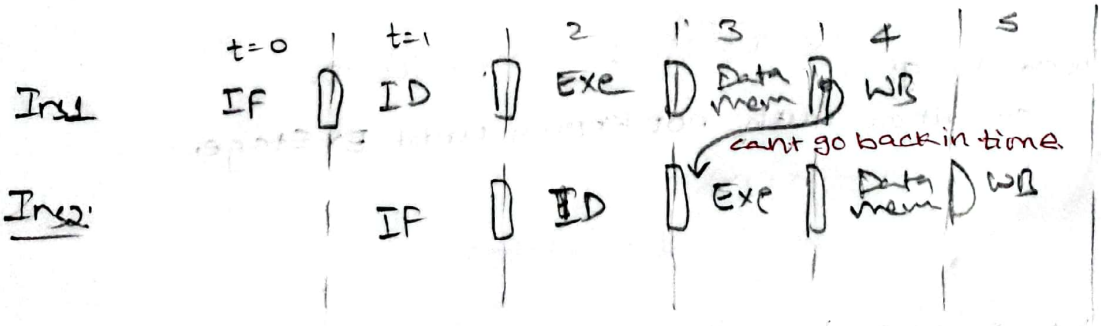
* Route data as soon as possible after its calculated, to an earlier pipeline stage.



<L23 slide 56>

Imp note: lw
* When we load something & immediately try to read; data hazard! even with bypass!!

Since; when Ins2 is in Exec, Ins1 is still in DATA MEM Stage. didnt fetch it.



* Bypass doesnt mitigate control hazards at all! only data hazards.

→ Control hazards:- ocurs during branch or jump.

* When do we calculate next PC: what we need

  - Jumps
    - opcode, offset, PC
  - Jump Register
    - opcode, registervalue
  - Conditional Branches
    - opcode, offset, PC, register (for condition)
  - For all others
    - opcode, PC needed

* When do we know these quantities:

  - PC — fetch stage
  - opcode, offset — Decode (or fetch...)
  - Register value — Decode
  - Branch condition ((rs)==0) — Execute (or decode?)

* Speculate PC=PC+4. might work ~50%
  - IF we mis-speculate, then kill, or insert nops.

- conditional branching: → 2 nops 1 nop for jump.
  condition truth not known until Ex Stage.

<L23, S11>
pipline visualization.

- instructions b/w branch, target are called wrong-path.

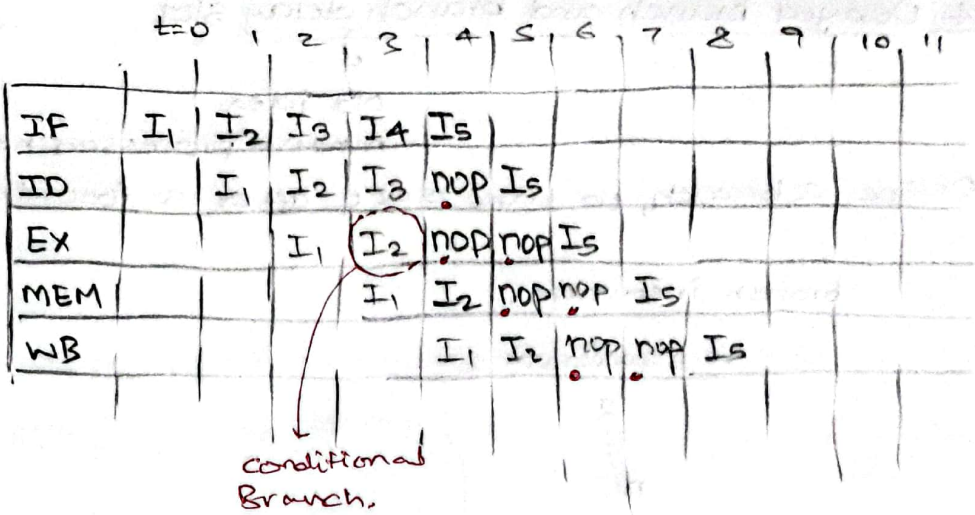
I_1: 96: ADD

I_2: 100: BEQZ 200

I_3: 104: Add

I_4: 108: Sll

I_5: 304: Add



conditional Branch. EX stage completed!

if we jump; give 2 nops.

*nops do no change in datamem or registers.
Sll $0, $0.

* Branch instruction; when will we know if taken or not. When will we know target address.

Inst.	Taken known?	Target known?
J	After ID	After ID
BEQZ	After EX	After EX

→ if we add an ALU in decode stage,

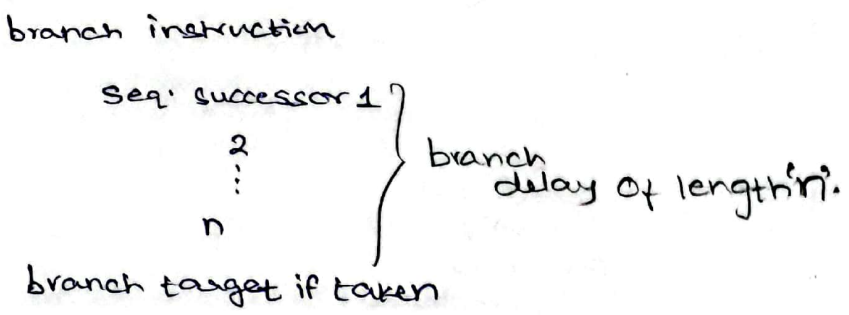
BEQZ	AfterID	After EX

really?
hmmm...
vanilla 5-stage pipeline do in the.
- EX — we calculate.
MEM Stage; we write back....

L24: Delayed branch and branch delay slot

old idea.
modern processors have branch predictors.

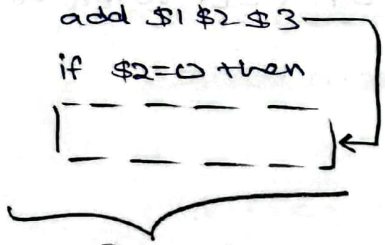
* Define a branch, to take place after a following instruction.



* Don't put a branch in a branch delay slot.

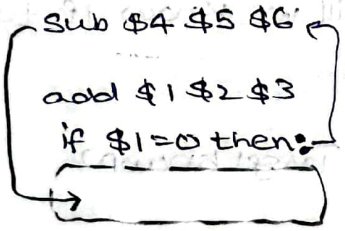
→ Scheduling branch delay slots:-

a. from before branch:

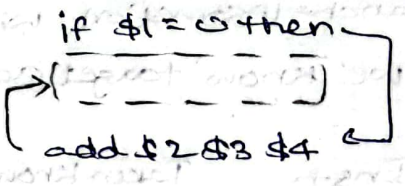


Best choice

b. From branch target:-



c. From fallthrough:-



might have to kill; in case not correct.

→ Stalls and performance:-

* N instructions & S Stalls;

$$CPI = \frac{N+S}{N}$$

* $$\text{Pipeline Speedup}_{new} = \frac{\text{Pipeline depth}}{1 + \text{Stalls due to branches}}$$

where Stalls (branches) = Branch frequency x penalty.

Data hazards
- Bypassing
- Stalls (NOPs) if bypassing won't work

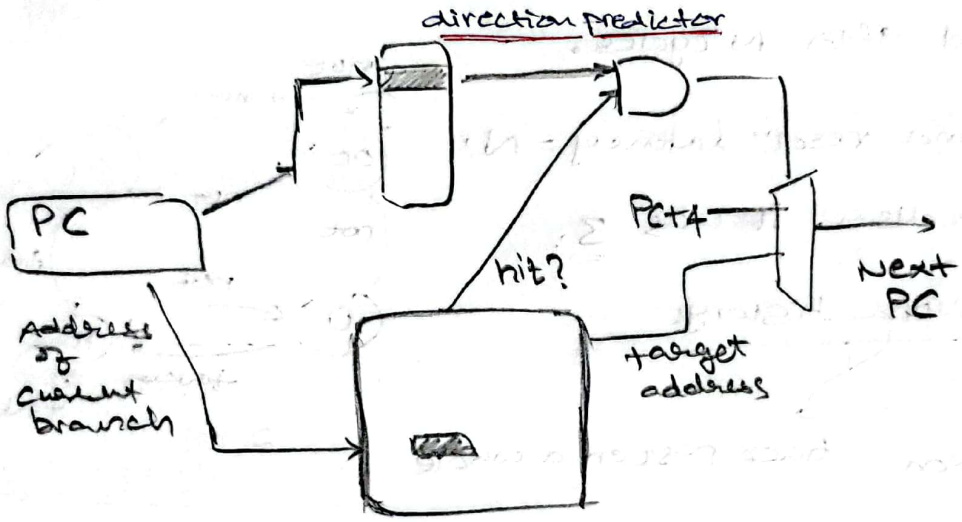
Control hazards:-
- Speculate PC=PC+4; Kill if wrong
- Delayed branch & branch slots
- Branch predictors

L25:- Branch Predictions:-

L26

* One way to mitigate control stalls. If we mispredict; kill all!
* Done in the IF stage itself.

* 3 tasks:-
  1. Is the instruction a branch instruction? In ID we'll know
  2. If yes, can be predict taken or not? → A hit in BTB means that Instruct^{n} is a branch instruct^{n}!
  3. If taken, predict the target address.



Repo of target addresses.

BTB: branch target buffer

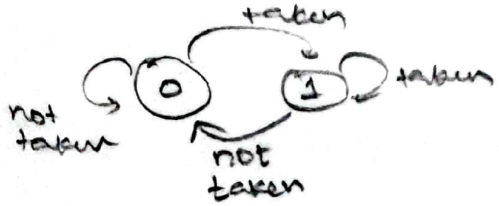
→ Static direction prediction techniques:-

1) Always not-taken:
   - no need BTB
   30~40% accuracy

2) Always taken:
   - need BTB
   60~70% accuracy
   Since loops are taken most of the time.

→ Dynamic:-
micro-architectural way! not software....

i) Last time predictor



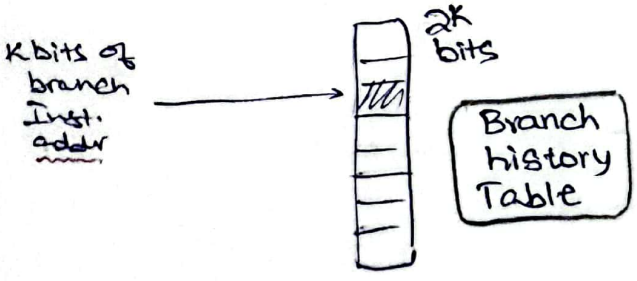
TTTTNNNN — 1 out of 4 times wrong....

* for loop with N-iterations

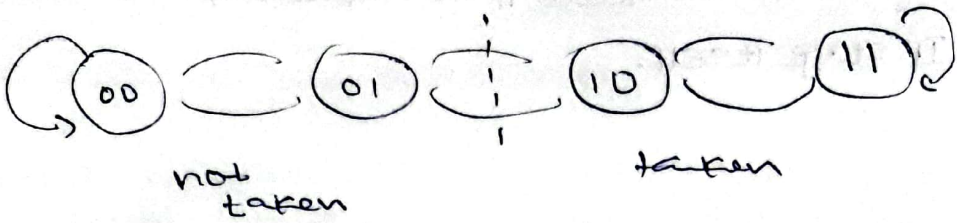
$$\text{Accuracy} = \left(\frac{N-2}{N}\right)$$

Mindyou! Instruction each branch address has its own predictor.

Hence a BHT.



2) 2-bit bimodal predictor :-

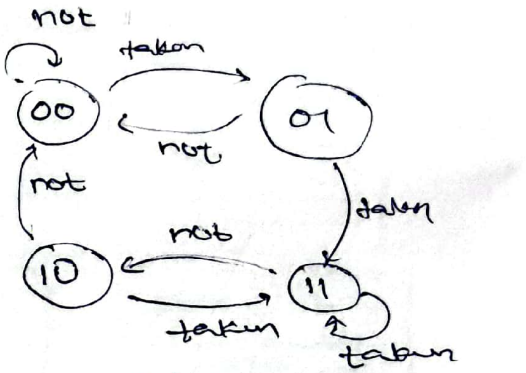


* Branch resolution latency :-

next fetch address after a control flow instruction is not determined after N cycles.

Branch resolⁿ latency = N.

for our dlx it was 3.

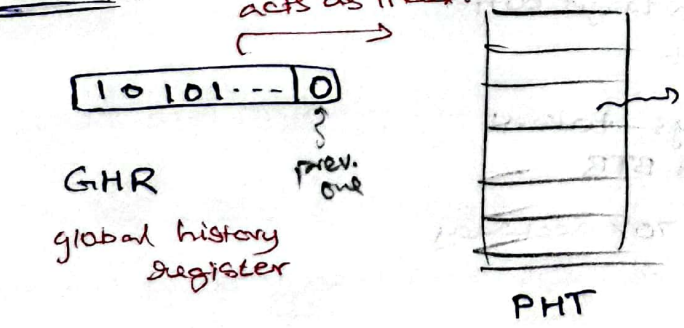


2-bit Hysterisis counter

→ Local history & Global history :

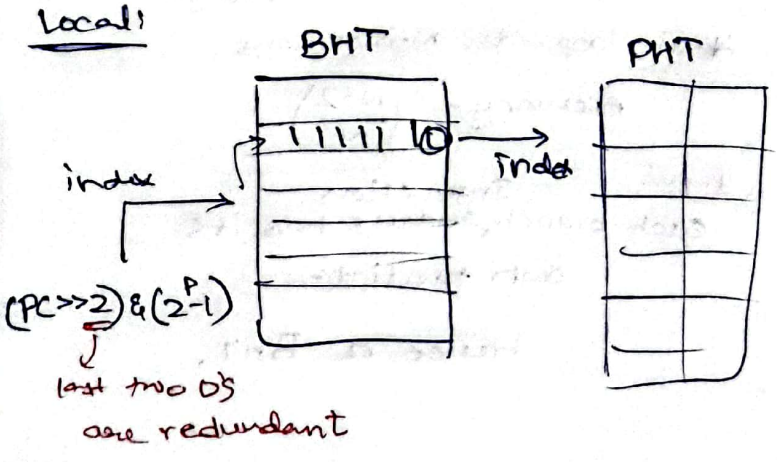
- for each branch instruction track its past
- track past on a whole

Global :

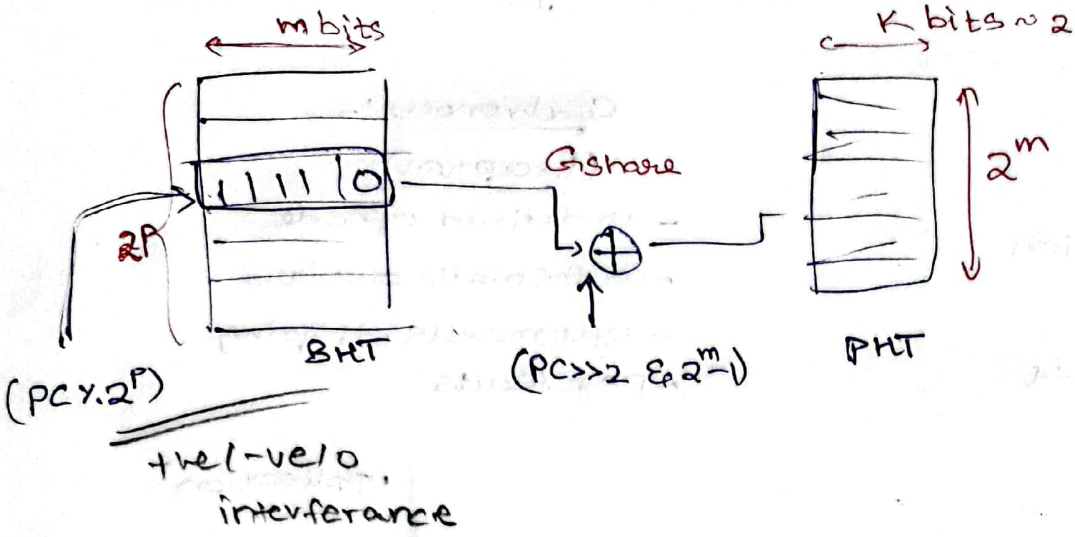


We mostly use a 2-bit rather than 3/4/---

Local :



* we also use set of branches

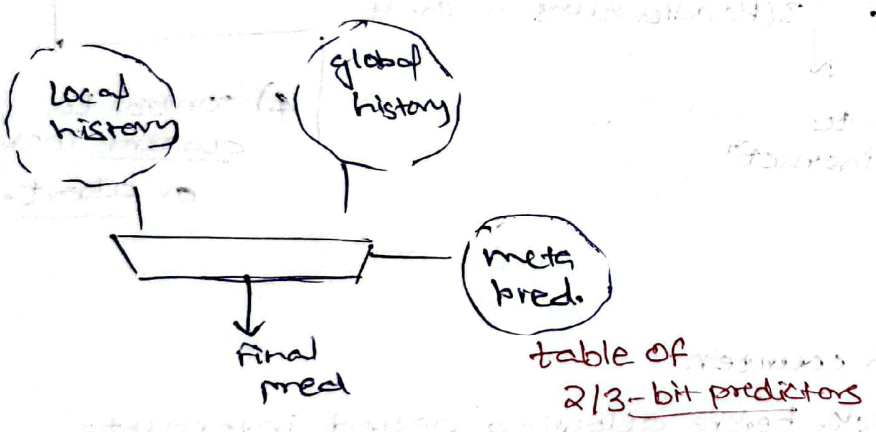


* Branch prediction in IF stage
* BHT, PHT updated after EX stage.

issues? branch after branch.....

Mahh!

→ Tournament predictor:-

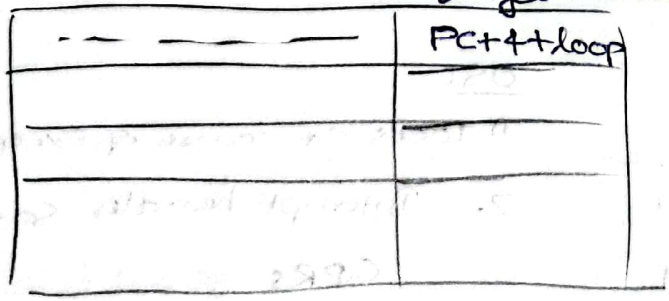


→ BTB:

last 2 bits redundant

30 bit address tag

target address



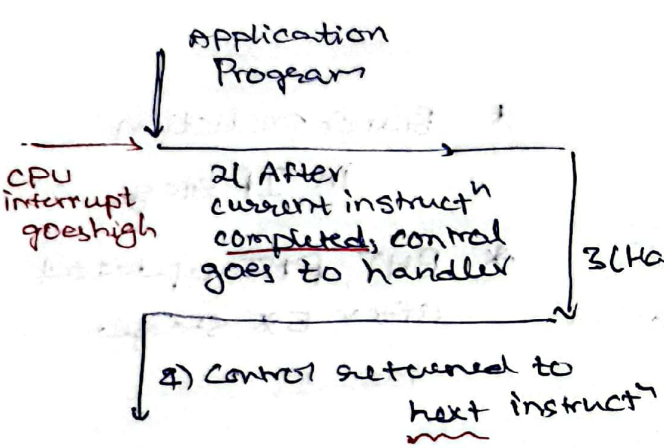
* A hit in BTB indicates PC is branch instruction.

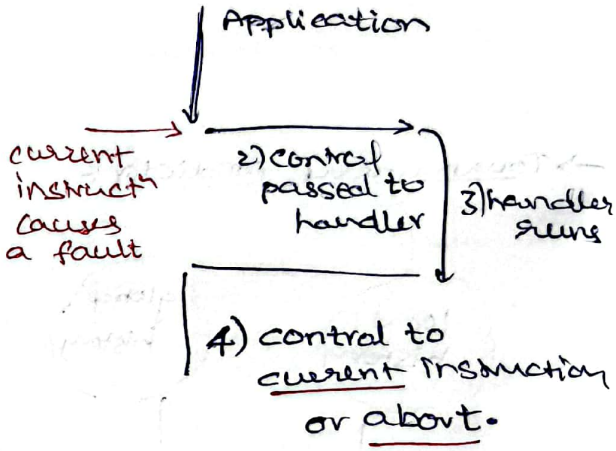
L27:- Exceptions in Pipeline:-

* An unscheduled event, that disrupts program in action

Asynchronous:-
(Interrupt)
- timer expiration
- device I/O
- hardware failure

Synchronous:-
(Exception)
- undefined opcode
- arithematic overflow
- systemcalls - traping
- page faults





→ **Interrupt Handler:-**

* 1) EPC: exception program counter.
2) Save epc onto stack; before allowing nested interrupts.
3) Also save the cause register.
otherwise use vectored interrupts.

→ **Processor – OS handshake:-**

Processor:-
1. Stops the offending ins.
2. completes the prior instructions
3. flushes the future instructions
4. Sets cause register
5. Saves EPC
6. Disables further interrupts.
7. Jumps to predecided address.

OS:
1) Looks at cause of exception
2. Interrupt handler saves the GPRs general purpose regs.
3. Handles the exceptions
4. Calls RFE
→ indirect jump instruction: return from excpt^n
- enables interrupts
- goes from OS mode to user mode.

Except handling & pipline.

Watch lec again.

L28:- Superscalar & O3 processors:-
out-of-order

- Scalar pipeline: CPI ≥ 1.0
  - can never run more than 1 ins/cycle.

Superscalar can have CPI < 1.
- execute multiple instructions parallelly.

→ Instruction level parallelism ILP:-

1. Scalar:
   say D-stage pipeline.

- Instruction overlap parallelism = D
- peak IPC = 1.0 (in all stages loaded.)

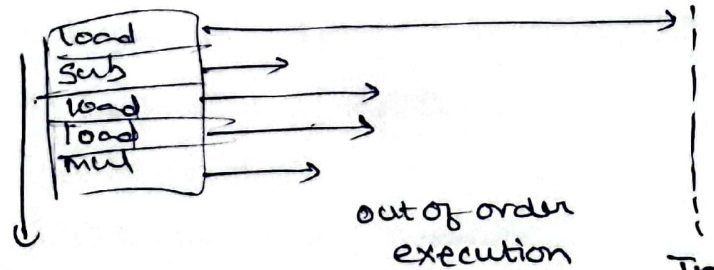
2. Superscalar:
   say N instructions fetched at once.

- Instruction overlap parallelism = D × N
- peak IPC = N.

* Hope is that we're able to fetch N independent instructions.
  - complicates datapath
  - complicates exceptions

→ O3 processors:-

* orthogonal development to superscaling/pipelining.

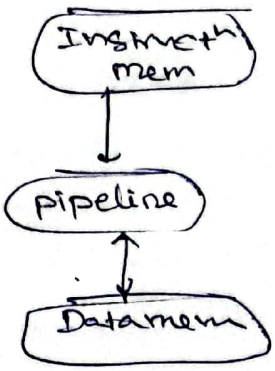


O3 + superscalar

So inorder is must

L29: Memory Hierarchy:-

* Ideally



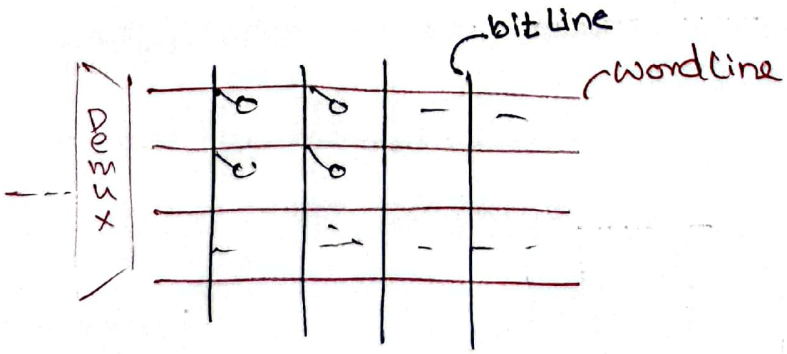
- zero-cycle latency
- Infinite capacity
- Perfect control flow! Oh!

Branches ~ 20% of code print.

- zero cycle latency
- Infinite capacity
- Infinite bandwidth.

SRAM:	DRAM:
fast	
costly	
no refreshing	- refreshing needed, as capacitor leaks
- 2 inverters	- capacitor charge
- 6 transistors per bit	- 1 transistor + 1 capacitor per bit

* Dram organisation

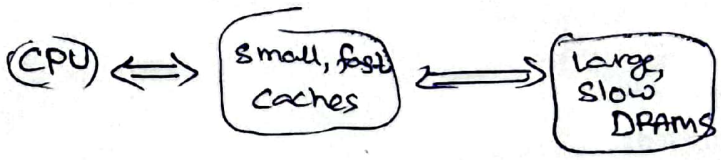


* Bigger is slower

Faster is costly.

* But we want fast and big!

Hence
- memory hierarchy,
- caches...



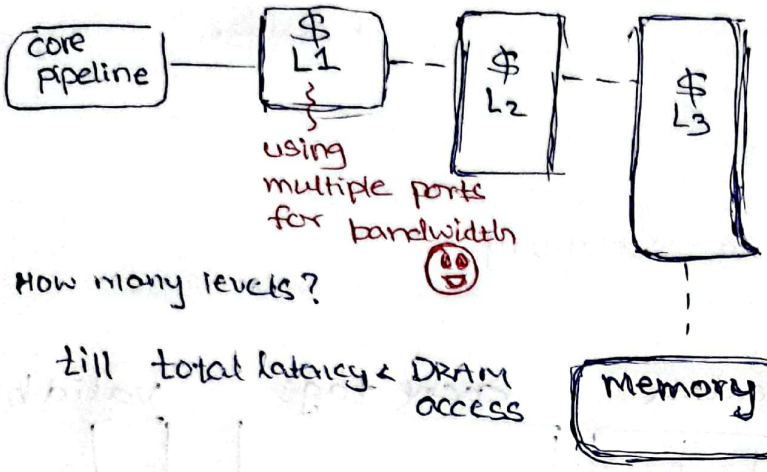
L-30: Caches $ // coz, costly jobs.

* DRAM access takes like 200-300 clockcycles.

Need to improve this.

→ Introduce caches.

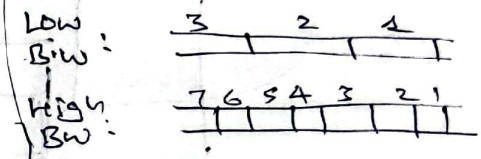
need to have less latency, more bandwidth



Personal doubt?

Is it bandwidth = 1/latency?

Nah! Bcoz, even if speed of transmission is same; length of message matters



or add parallel channels!

* Our utmost priority is getting better latency.

L1: latency + bandwidth

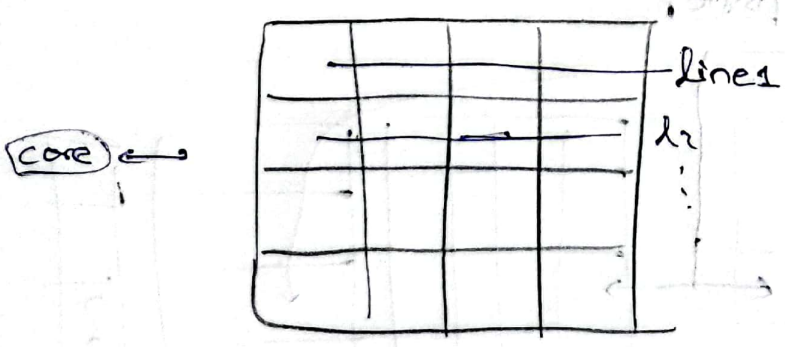
L2: latency

L3: Capacity... used by multicores.

L31:

* We always see cache $ in terms of lines:-

whenever we write into cache; we write a line.

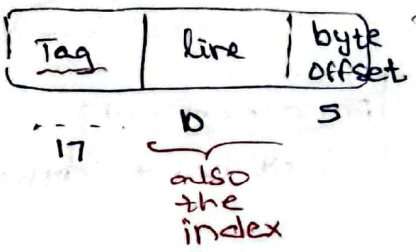


Typical line 64 to 128 bytes

Eg: 1024 lines with 32 bytes each

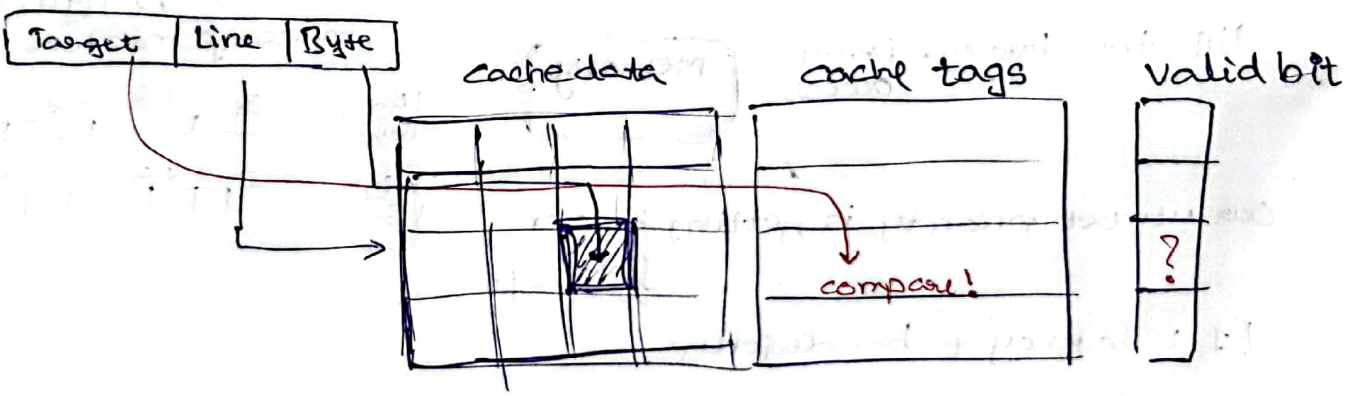
2^{10} 2^{5}

for Address of 32-bit



→ Hence, we need to compare tags finally.

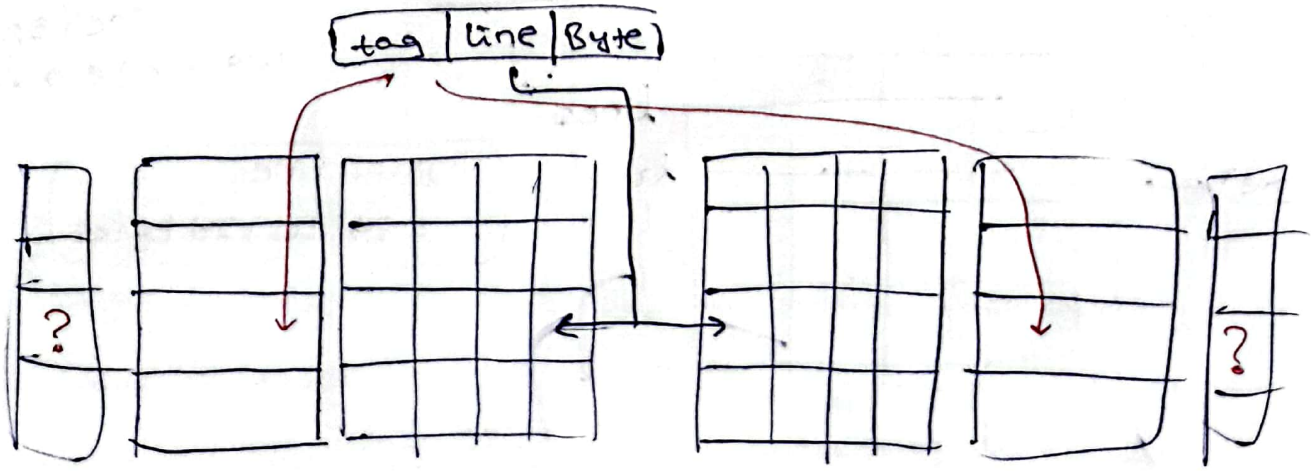
1) Directly Mapped Cache:- 1-way associativity.



2) 2-way associative caches:-

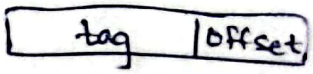
- for a single line index, we get multiple data+tag blocks

Set ≡ bunch of lines, with a index.



Similarly, 4 way associative, n-way associative...

* Fully associative Caches:-



single line.

* Knobs: size of offset
Line Size, associativity, cache size.

Tradeoffs: latency, complexity, energy

L32:-

→ cache misses cuz they are costly:-

line ≡ block — but line has many blocks in n-way associativity.

cold miss:- misses that would occur, even with ∞ cache
- first reference to a line.

** capacity miss:- misses that would occur, even with perfect (Belady's) replacement policy. (know the future.)
- cache size insufficient.

** conflict miss:- misses that wouldn't occur, with ideal full associativity.
(can be eliminated.)
- many mapped to same line

→ on miss; replace a block:-

* Each block has priorities (dynamic ...).

Ideal policy:- Belady's OPT policy:

* Replace the one, which comes in the farthest future.

impractical!

* 3 decisions — Insertion :-
Promotion :-
Eviction :-

1) Simple LRU:-

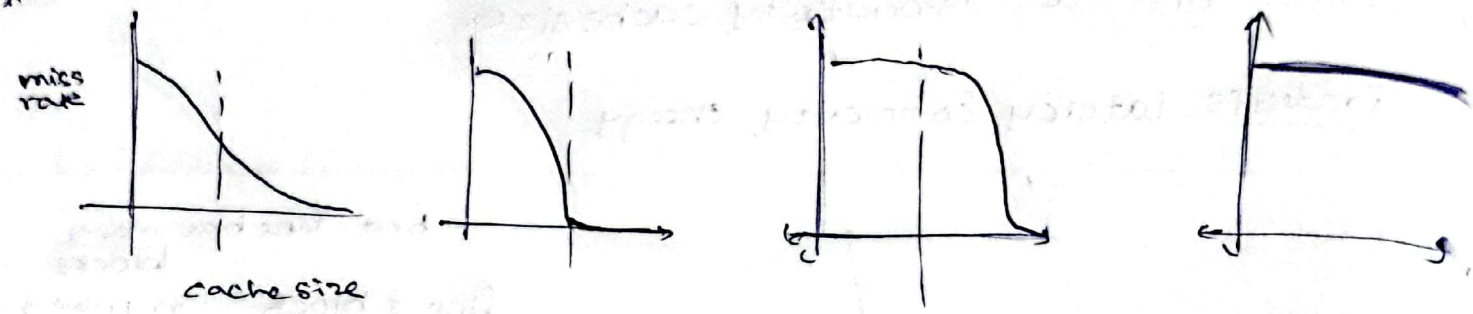
Insertion: at MRU
eviction: at LRU
Promotion: bring to MRU

* we need priority bits per block in a line.

∴ 16-way cache;
each block → 4 bits
expensive!

* LRU causes thrashing when working set > cache size.
at some duration of time...

*



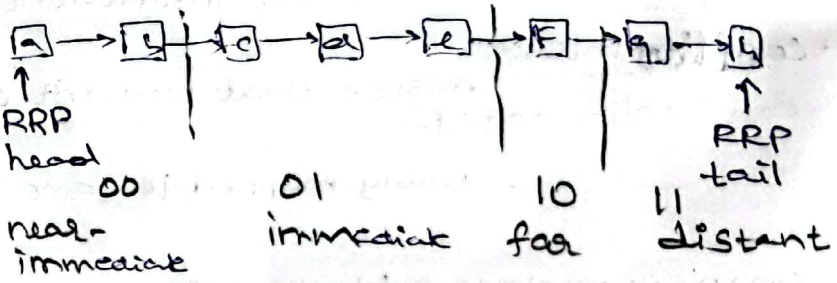
a) cache friendly workload b) Cache fitting workload c) Cache Thrashing workload d) Streaming workload.

* Also; LRU not effective for shared caches
L3 cache.

*

RRIP:

re-reference interval prediction policy



* RRPV
- pred. values

We'll group into 4 groups.
∴ need only 2 bits / Block
In 16-way caches

Evict:- value of block = 11

Promotion:- make block value = 0

Belief:
New cache block will not be re-referenced soon.

Insertion:- insert with RRPV = 2
not 0 like LRU

→ Knobs & misses:-

1) Larger cache size
- reduces capacity misses, conflict misses.
- Hit time increases
latency

2) Higher associativity
- reduces conflict misses
- May increase hit time
(more compasators)

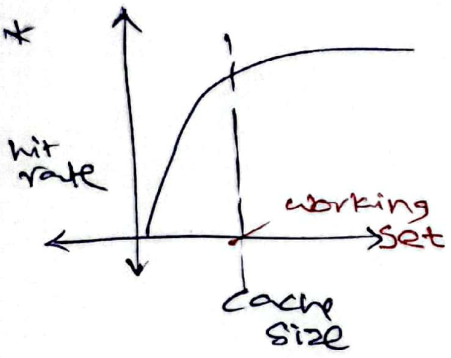
3) Larger line size
- reduces cold misses
since we load, 1 line at a time.
- increases miss penalty
(more to be loaded)

not sure
- increases conflict misses.

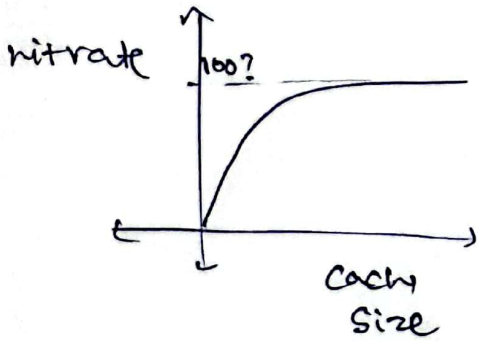
* too few block size: spatial locality not exploited

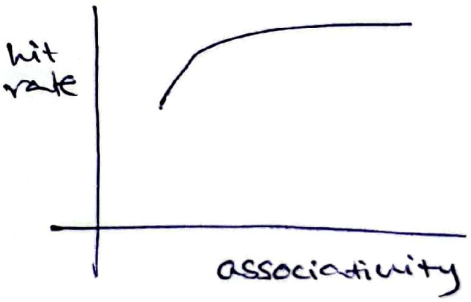
too large block size: waste info copied. more energy.

*



: size of data, referenced in a interval.





But hittime also increases.

L1: low associativity

L3: high associativity

* Missrate per kilo instructions

(MPKI)

per 1000 instructions is a normalizing factor.

→ Average Memory Access Time:

AMAT = Hit time + MissRate (Miss Penalty)

even misses need this time to check... cool!

Also

$= (\text{Hit time})_1 + \text{missrate}_1 \; (\text{Miss Penalty}_1)$

$\|$

$(\text{Hit})_2 + (\text{miss Rate})_2 \; (\text{miss penalty}_2)$

3

L33:

* L1 data cache:-

  * High bandwidth
    low associativity to fast hit time

  Victim Cache:

  Store some evicted lines of L1 cache
  for better hit rate.

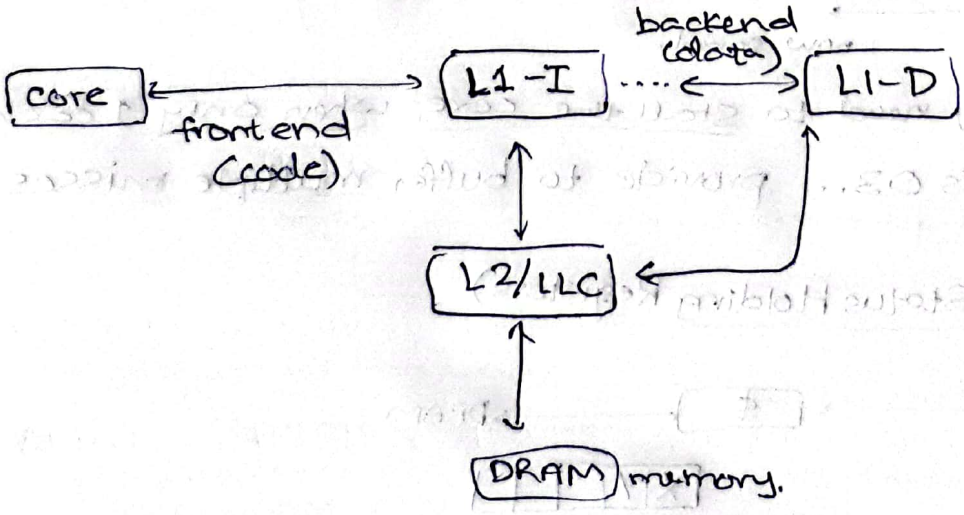
* L1 instruction cache

  Hmmm...

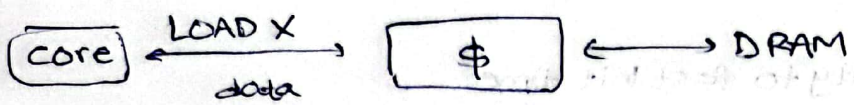
  - special cache that packs multiple non-contiguous blocks into one contiguous trace cache line

  - single fetch brings in multiple basic blocks
  - Indexed by start address & next n branch predictions.

*



→ Core, cache, DRAM interaction:-

* 

Hit

A few cycles ⏳.

* if we miss;

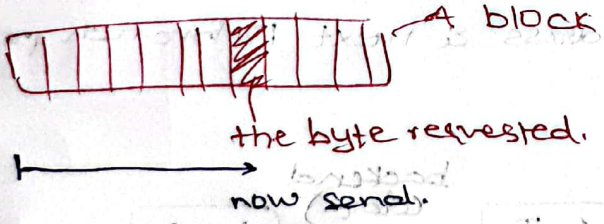
a) Critical word first:-

* respond with the word/byte requested to the core; & then continue fetching rest of the block.

since we load a cache block wise!

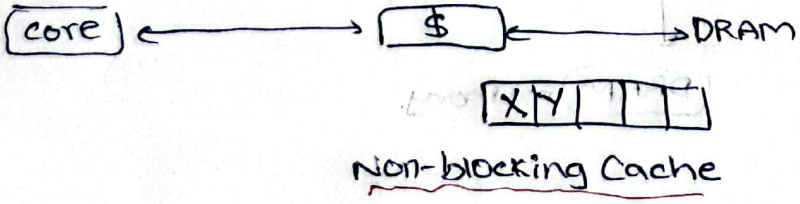
b) Early restart:-

* fetch the words/bytes in normal order, but as soon as requested word arrives, send to core.



* So; do we really need to stall the core; when only 1 cache miss? the core is OO... provide to buffer multiple misses.

→ MSHRs (Miss Status Holding Registers)



Non-blocking Cache

K-entry MSHRs allow K-outstanding Misses:

Memory Level parallelism

MLP

Are all fetched parallely?

→ What about write?

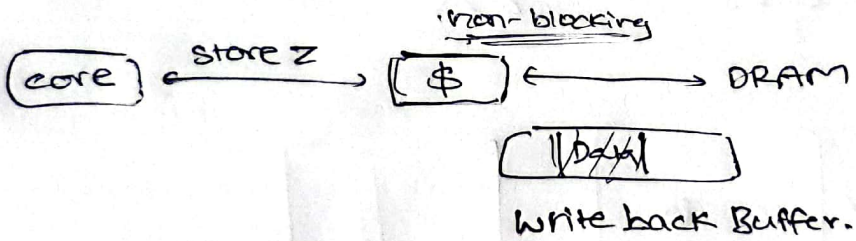
* On hit, update cache and set dirty bit to 1.
  other bits are valid bit.
  replacement priority bits.

a) Write-through cache :-

On a hit, write into a cache & also into next-level in hierarchy.
(immediate)

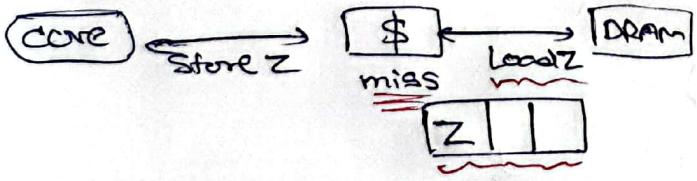
b) Write-back cache :-

On hit, write into cache only. During eviction; write into next level.
later, during replacement

* 

In general, stores are not critical for performance...

→ Write-miss :-

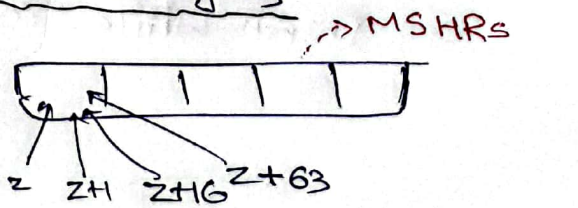


* store gets converted into load;
  & data is allocated into cache

write-allocate policy

used, along with write-back policy.

Write-merging



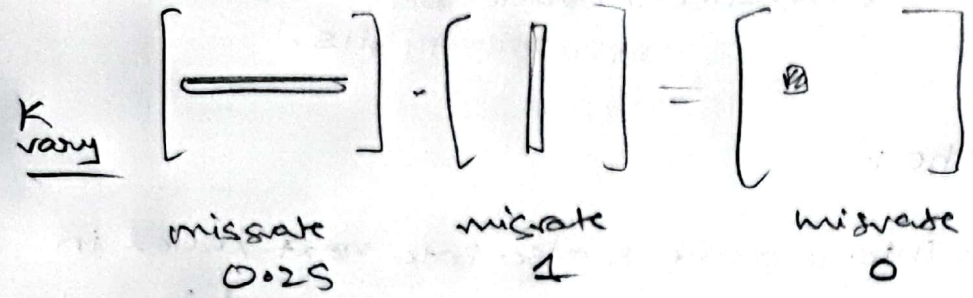
All these store requests are merged into one.

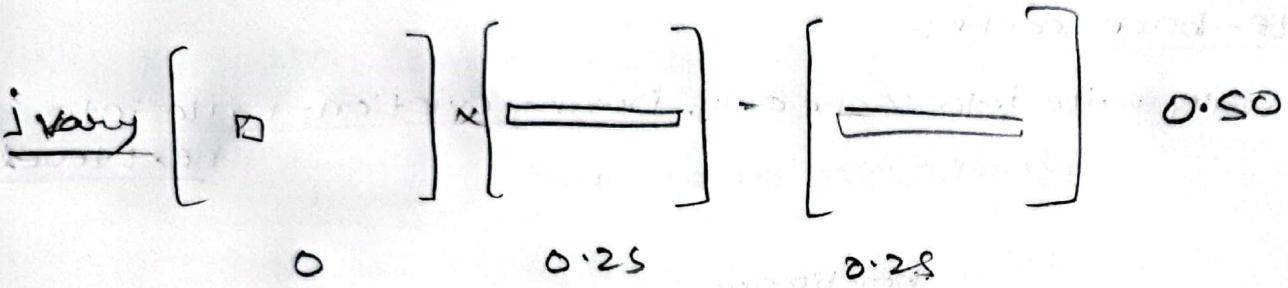
Meh! even load merge then.

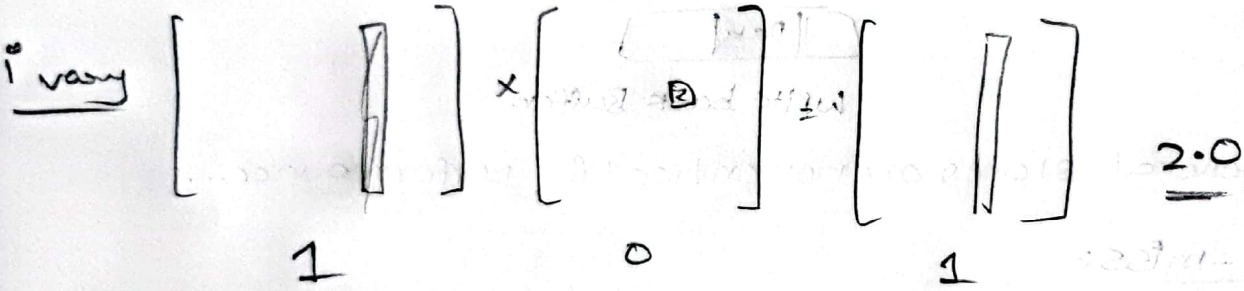
* CPU time = CPU execution cycles + Memory-stalls
  = read stalls + write stalls
  = # Read/writes × Read/write miss rate × miss penalty.

L34)

matrix mult:







Think while coding!