HIGH PERFORMANCE COMPUTER ARCH:

Microprocessor: A single chip containing the entire CPU.

RISC-V (32-g.p. and 32-f.p.)

* Classic RISC design
* Load-Store design
* Plenty of General Purpose Registers
* Extensible for advanced features.

CISC	RISC
$[A_1] \leftarrow [A_2] + [R]$	$[R_2] \xleftarrow{load} [A_2]$ $[R_3] \leftarrow [R_2] + [R]$ $[A_1] \xleftarrow{store} [R_3]$

How to achieve high performance?

Parallelism

* System-level (e.g. scale-up servers)
* Individual processor level (e.g. pipelining)
* Design level optimizations (e.g. faster ckt, larger cache, e.t.c.)

Rule of Thumb: An executable spends ~90% of its execution time in about 10% of its code.

"Make the common case fast"

Principle of Locality:

Spatial locality: Instructions (or data) whose addresses in memory are close to each other are likely to executed (or accessed) close in time.

(1 out of 7 instructions are branches / function calls / jumps, e.t.c). ~ 15%

Temporal locality: Recently accessed data (or executed instruction) are likely to accessed (or executed) again.

(about 6 instructions are in one basic block, i.e. without jumps / function-calls / branches. Processors use branch prediction for better performance)

Amdahl's Law [IBM – 1967]

* "Law of Diminishing Return"
* "Inherently pessimistic law" --- (Wonk Wonk :/)

Definitions

- Speedup after enhancement, (S_e)
- Fraction of program which is parallelizable (f_e)

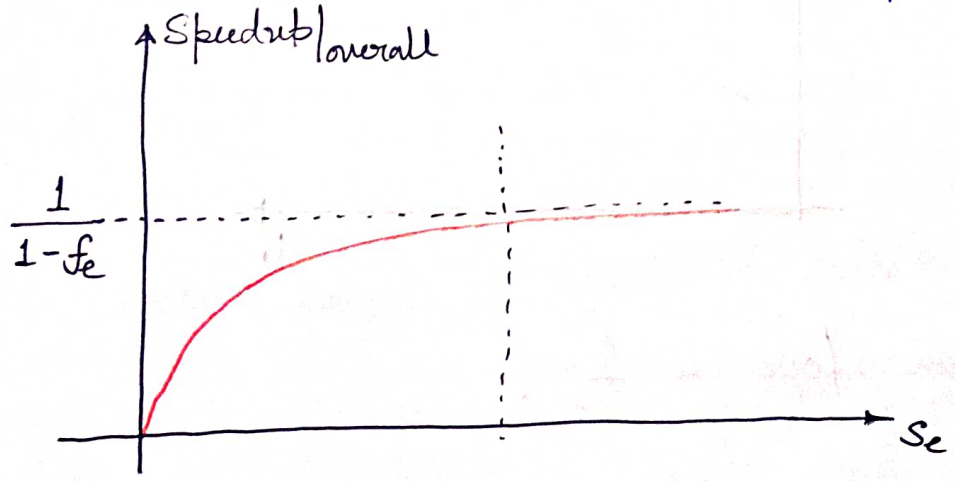
$$\text{Speedup}|_{\text{overall}} = \frac{\text{Execution time of entire task without enhancement}}{\text{Execution time of entire task with enhancement}}$$

$$= \frac{\text{Ex.-time}|_{\text{old}}}{\text{Ex.-time}|_{\text{new}}}$$

$$= \frac{\text{Ex.-time}|_{\text{old}}}{\text{Ex-time}|_{\text{old}} \left[\underbrace{(1-f_e)}_{\text{serial}} + \underbrace{\frac{f_e}{S_e}}_{\text{parallel}} \right]}$$

$$= \frac{1}{(1-f_e) + \frac{f_e}{S_e}}$$

$$\therefore \lim_{S_e \to \infty} \left(\text{Speedup}|_{\text{overall}} \right) = \frac{1}{1-f_e}$$



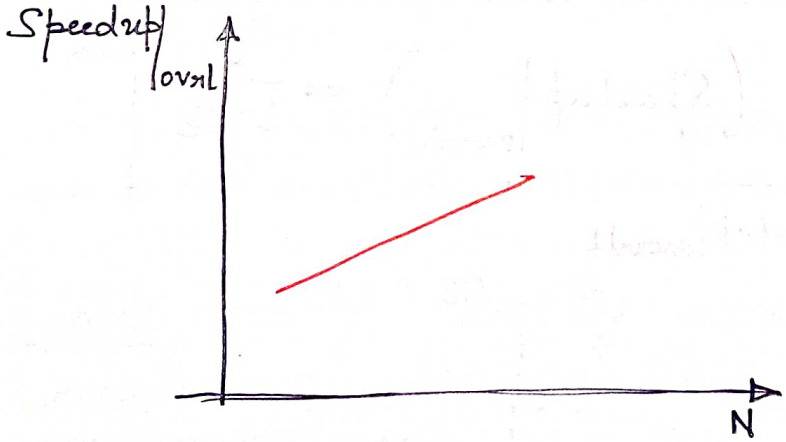
Gustafson's Law [1988]

* "Logic is the following: We scale up the problem size as well, as we are capable of scaling up the system; Problem-size $\propto$ # processors"
* "Extremely ~~Pessimistic~~ Optimistic."
* "The serial part of the problem stays the same even when we scale up."

$$\frac{\text{Ex-time}|_{old} \longleftrightarrow \#\text{processors} = 1}{\text{Ex-time}|_{new} \longleftrightarrow \#\text{processors} = N}$$

$$\therefore \text{Ex-time}|_{old} = \text{Ex-time}|_{new} \left[(1-f_e) + N * f_e\right]$$

$$\therefore \boxed{\text{Speedup}|_{overall} = \frac{\text{Ex-time}|_{old}}{\text{Ex-time}|_{new}} = (1-f_e) + N * f_e}$$



(Max slope is 1).

Comparing Results:

Example:

Sequential part, $1 - f_e = 0.2 \Rightarrow f_e = 0.8$

M1 — single core

M2 — 80 core $\Longrightarrow N = 80 = S_e$

$\therefore$ [Amdahl] $\text{Speedup}|_{AMDL} = \frac{1}{(1-f_e) + f_e/S_e} = \frac{1}{0.2 + (0.8)/80} = 4.76$

$\therefore$ [Gustafson] $\text{Speedup}|_{GSTFSN} = (1-f_e) + N \cdot f_e = 0.2 + 80 * 0.8$

$= 64.2.$

"In early 2000's, Gustafson's Law was completely trashed because of extremely optimistic results"

(Moye Moye)

Processor Performance Equations:

0. CPU time = (# CPU clk cycles) * (clk cycle time)

= (# CPU clk cycles) * t_c = (# CPU clk cycles) $\cdot \frac{1}{f_c}$

1. $T_c = N_c \cdot t_c$
2. $N_c = T_c \cdot f_c$.

We would analysis 2 cases of scenarios.

Case-I: (Each instruction takes the same number of clock cycle).

A bit unrealistic case (delulu)

$\therefore$ Clock cycles per instruction (CPI) $= \frac{1}{IPC}$

$$= \frac{\#\text{ clk cycles for a program}}{\#\text{ instructions in program}}$$

$$\Rightarrow \boxed{T_\phi = \text{CPI} \cdot n_\phi \cdot t_\phi} \quad \ldots\ldots\ldots (A)$$

n_ϕ and t_ϕ are hard to decrease; so the natural choice is reducing CPI. A computer architect's primary goal here should be to try reducing T_ϕ by reducing CPI.

Case-II: (There are several classes of instructions, each with a different CPI)

n_i instructions get CPI_i

$$\boxed{T_\phi = \left[\sum_i (n_i \cdot \text{CPI}_i)\right] \cdot t_\phi} \quad \ldots\ldots (B)$$

Equivalently, if ~~CPI$_i$~~ $f_i = \frac{n_i}{\sum_i n_i}$, then

$$\boxed{\text{CPI}\Big|_{\text{effective}} = \sum_i f_i \cdot \text{CPI}_i}$$

There are further generalizations (due to page fault / disk access)

A further generalization of (B) designed to handle penalty would be something like:

$$T_{\ell} = \left[\sum_i (n_i \cdot CPI_i) + n_{penalty} \right] \cdot t_{\ell}$$

and, $$CPI\Big|_{effective} = \sum_i f_i \cdot CPI_i + \frac{n_{penalty}\, n_{penalty}}{n_{penalty} + \left(\sum n_i \cdot \cancel{CPI_i}\right)}$$

where $n_{penalty}$ = # clk cycles wasted for penalty.

Memory Hierarchy - I

/* slides : C */

"n-way associative" ⟶ #entries in one set is n.

$$\#\text{sets} = 2^{\#\text{index\_bits}} = \frac{\#\text{blocks}}{\text{associativity}}$$

Corollary: For fully associative ⟶ #index_bits = 0.

"4-way set associative is the most common."

Write through: Always write to main memory whenever "store" operation happens.

Write Back: Store dirty bit & write back to main memory only when needed.

Write Buffer: (in CPU) Holds data to be written back to memory; CPU continues (can be used together with Write Back).

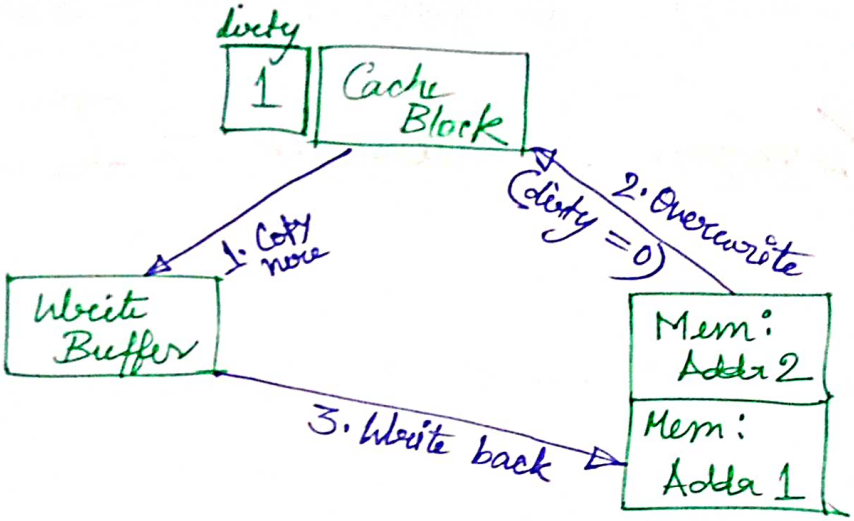


Fig: Replace data of Addr 1 in Cache (dirty) with data of Addr 2

Reads get more preference over writes

Write Allocate:

When write miss happens, swap the ~~too~~ thing on memory that is to be written (handle dirty-ness as needed)

No-write allocate:

When write miss happens, don't even update cache, write directly to memory.

"In a modern day single-core CPU, more than 100 instructions are on flight at a given moment"

Similar hierarchy is used in Hard disk – main memory boundary.

Translation Lookaside buffers are used (TLB):

* 16-512 PTE's are brought
* 0.5 – 1 cycle for hit
* 10 – 100 cycle for miss
* 0.01% – 1% miss rate
* Fully associative.

TLB Misses:

- Case 1: In PTE but not in TLB
- Case 2: Not in main-memory.

"Tag comparison and getting out data happen in parallel. Whether the CPU will pay attention to this data will depend on the tag comparison."

Estimation of size of Multi-Level Page Tables

[Worst case analysis] Entire 48-bit VM space is utilized.

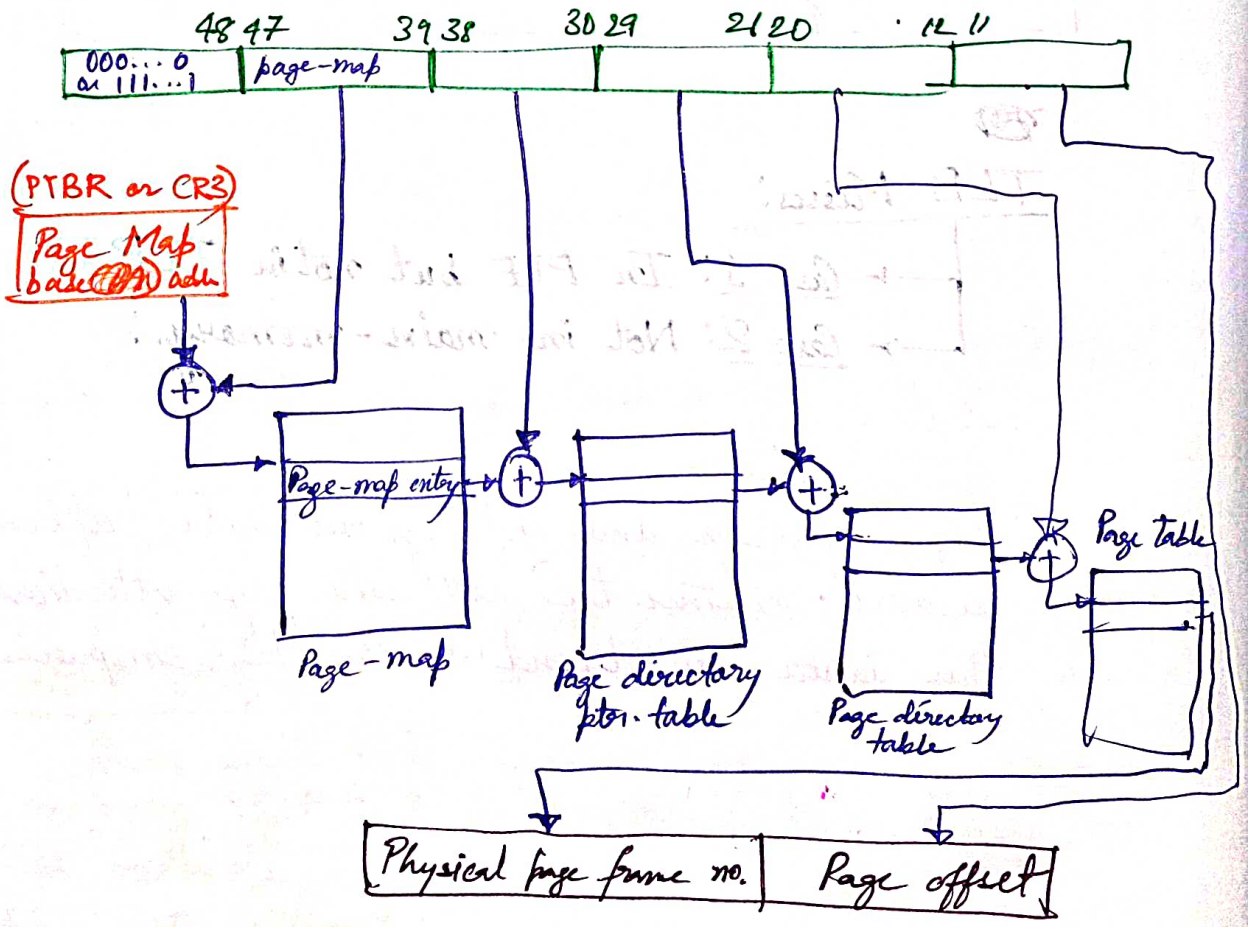
Page size — 4kB $= 2^{12}$ B, Page table entry size — 8B $= 2^3$ bytes

Option 1: Single Level (Flat) P.T.

$$\#\text{pages} = \frac{2^{48}}{2^{12}} = 2^{36}; \text{ Size of PT} = 2^{39}.$$

Option 2: 4-level P.T.

Outer level P.T. = Page Map (= 1 page)



$\therefore$ #pages for P.T.'s $= \underbrace{1}_{\text{L4 P.T.}} + \underbrace{2^9}_{\text{L3 P.T.}} + \underbrace{2^{18}}_{\text{L2 P.T.}} + \underbrace{2^{27}}_{\text{L1 P.T.}}$

$\therefore$ Memory used for P.T. $= 2^{12}\left(1 + 2^9 + 2^{18} + 2^{27}\right)$ bytes.

--- হালকা হালকা বুঝলাম; খুব ভালো ভাবায় ঢোকেনি।

Look into VIPT Cache (L1 & L2) diagram!

Aliasing:

"Aliasing is basically the mapping of multiple virtual addresses being mapped into the same physical address"

- We will study aliasing in the context of Cache design.

This can happen if cache index collects some bits from V.A. which is not part of P.A.

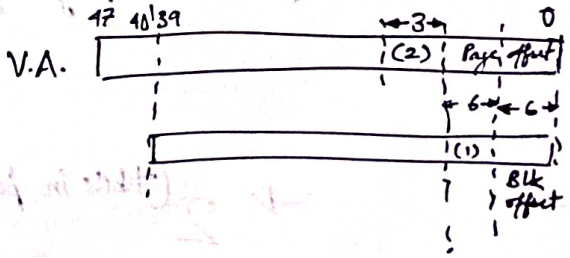
Example:

*[Opteron I Cache]:

- 64 kB, 4 kB pages
- 12-bit page offset
- 2-way set associative
- 48 bit V.A.
- 40 bit P.A.
- 64B blocks

$2^{(\#\text{index-bits})} = \frac{64\,kB}{(64B)\times \underbrace{2}_{\text{assoc.}}} = 2^9$

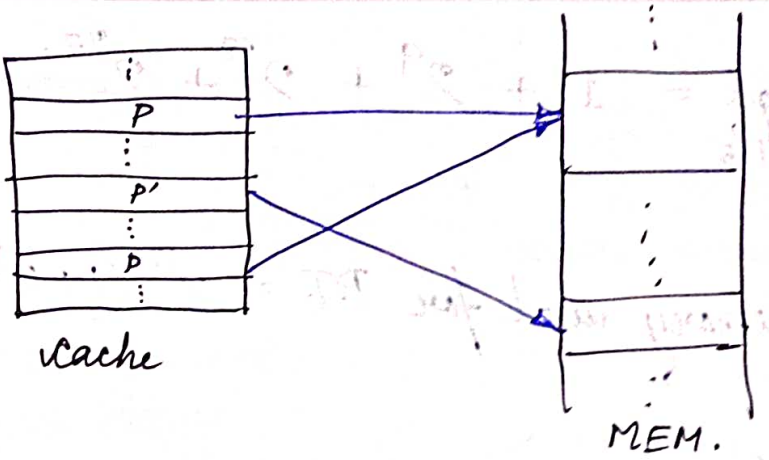
$\Rightarrow$ # index bits $= 9$



(1) 6-bit portion of cache index $\Rightarrow$ both in P.A. & V.A.

(2) 3-bit portion of cache index $\Rightarrow$ only from V.A.!

can be used to introduce ambiguity; ie. 2^3 cache lines can point to the same physical addr.



This is dead because one virtual memory might be overwritten whereas the other thinks that it is not written since the last time.

* One solution is the mark every other cache entry which can be mapped to ~~every~~ the same physical address, everytime something is written.

What we want then is the following:

$$\underbrace{(\#\text{bits in page-offset})}_{\text{virtual address}} \geq \underbrace{(\#\text{bits in block offset}) + (\#\text{bits in set index})}_{\text{physical address}}$$

$$\Rightarrow 2^{(\#\text{bits in page-offset})} \geq 2^{(\#\text{bits in blk offset})} \cdot 2^{(\#\text{bits in set index})}$$

$$\Rightarrow \text{Page-Size} \geq \text{Cache Size}.$$

$$\boxed{\text{Page Size} < \text{Cache Size} \Rightarrow \text{Aliasing}}$$

Average Memory Access Time (AMAT):

$$AMAT = (\text{Hit Time}) + (\text{Miss rate}) * (\text{Miss penalty})$$

Overall goal: decrease AMAT

* The aim is to decrease any of the three
* Things become difficult to analyse; maybe an attempt to decrease 'Hit Time' might actually increase Miss rate.

Example: Analysing 2-level caches

$$AMAT\Big|_{2\text{-level}} = \underbrace{(\text{Hit Time})\Big|_{L_1}}_{HT} + \underbrace{\text{Miss rate}\Big|_{L_1}}_{MR} * \underbrace{\text{Miss penalty}\Big|_{L_1}}_{MP}$$

$$\Rightarrow AMAT\Big|_{2\text{-lvl}} = HT\Big|_{L_1} + MR\Big|_{L_1} \cdot \left[HT\Big|_{L_2} + \underbrace{MR\Big|_{L_2}}_{\text{quite low}} \cdot \underbrace{MP\Big|_{L_2}}_{\text{very large}} \right]$$

$MR|_{L_2}$ is low as:

- Larger size of L_2
- Set associativity, e.t.c.

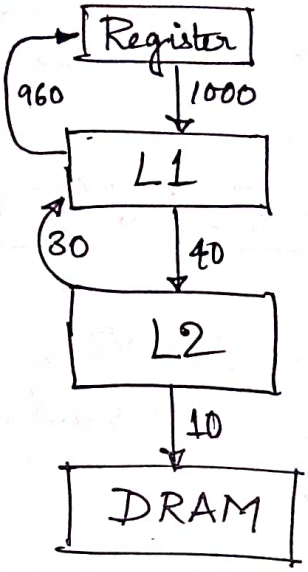
OK sorry $MR|_{L_2}$ is actually high. 😐

Typically,

$$HT|_{L_1} < HT|_{L_2}$$

$$MR|_{L_2} \not> MR|_{L_1}$$

$$MP|_{L_2} \approx 100 \text{ cycles}.$$



The local miss rate of L2 is large because most of the queries are satisfied by L1 itself.

∴ We will call this the Local Miss Rate.

Here $MR_{L_1} = \frac{10}{40}$

∴ Global Miss Rate $|_{L_2} = MR|_{L_1} * MR|_{L_1}$

"Don't disturb the memory system unnecessarily, there is a huge penalty for the same."

Example (1)

20% → store
30% → load

⇒ 100 instructions → 150 memory accesses.

Eg.2

Among 1000 memory accesses
40 misses L_1
20 misses L_2

$M.P.|_{L_1} = 10$, $HT|_{L_1} = 1$

$M.P.|_{L_2} = 200$, $HT|_{L_2} = 10$

A.M.A.T. = ?

$$MR.|_{L_1} = \frac{40}{1000} = 0.04$$

$$MR|_{L_2} = \frac{20}{40} = 0.5$$

$$\therefore AMAT = HT|_{L_1} + MR|_{L_1}\left[HT|_{L_2} + MR|_{L_2} \cdot MP|_{L_2}\right]$$

$$= 1 + 0.04\left[10 + 0.5 \times 200\right]$$

$$= 1 + 0.04 \times 110$$

$$= 5.4 \text{ cycles}.$$

Now, 1.5 memory accesses on average per instruction.

$$\Rightarrow \#\text{ instr} = \frac{1000}{1.5}$$

∴ Average memory stall per instruction

=

Memory stall can happen if L1 misses, L2 hits
or L1 misses & L2 misses.

Advanced Cache Optimizations

1. Way Predictions:

- Enables faster S.A. cache (comparable to D.M. cache)

(Previously the last layer OR-gate creates delay)

Exact কি করা হয় বললেন না ; রিসার্চাররাও নাকি জানেনা । AMD/Intel-এর লোকজনই জানে নাকি ।

* Some kind of prediction hardware is used.

2. Pipelined Caches & Multibanked Cache

Superscalar processors: Run multiple instructions.
(Will study in more detail)

* Cache can have a different operating frequency than the microprocessor operating frequency. Therefore it is nice if the CPU can continue executing unrelated instructions rather than waiting for cache to return. [Pipelined cache]

* Break a cache into banks

Bank 1	Bank 2	Bank 3	Bank 4
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

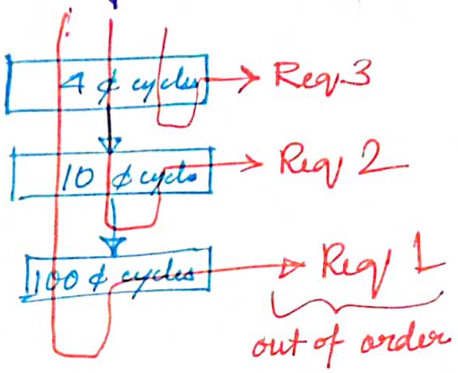
This allows "parallel" access in some sense.
If two consecutive cache accesses are from different banks, the second access can be done even before first one is incomplete

3. Non blocking Cache (lockup-free cache)

When one memory request has missed the cache, the cache moves on to the next memory request (while remembering that one request exists)

* Earlier caches could keep at most one request miss pending (hit-under-single-miss). Others are known as hit-under-multiple-miss.

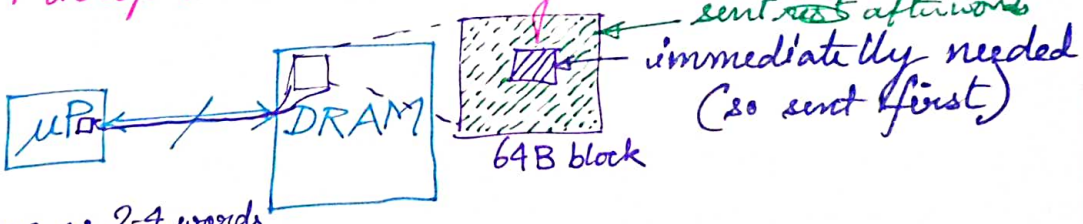
However, ordering is problematic.



Superscalar processors take care of it.

4: Early Restart / Critical Word First

Observation: Microprocessor needs only 1 word in a block.



Bus size ~ 2-4 words
Block size ~ 8/64 word

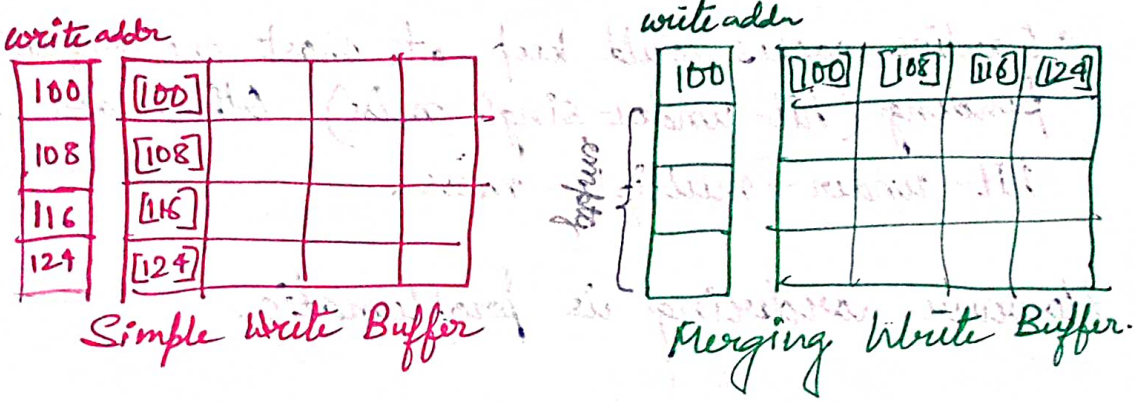
Fig.: Critical Word first

(Immediately needed word sent first, CPU works on it while other parts are sent after that)

Early Restart: DRAM sends sequentially, when CPU finds its immediately needed block, it starts executing while the rest of the block is still being brought in.

5. Write Merging

* If 'close-by' write operations queue up, then put them 'together' at write buffer & write them back at one write operation.



Compiler-based scheduling → static scheduling
Hardware-based scheduling → dynamic scheduling (runtime)

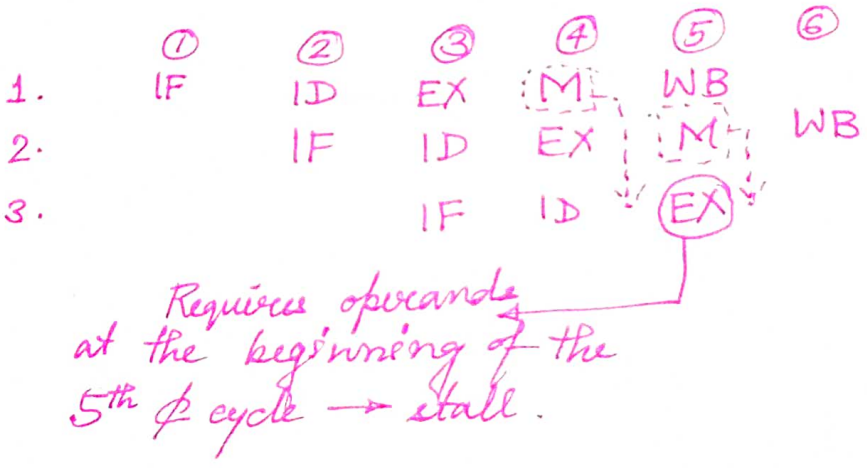
What we are going to study today is different from RISC-V.

Motivation:

```
for (i = 0; i < N; ++i)
    z[i] = x[i] + y[i];
```

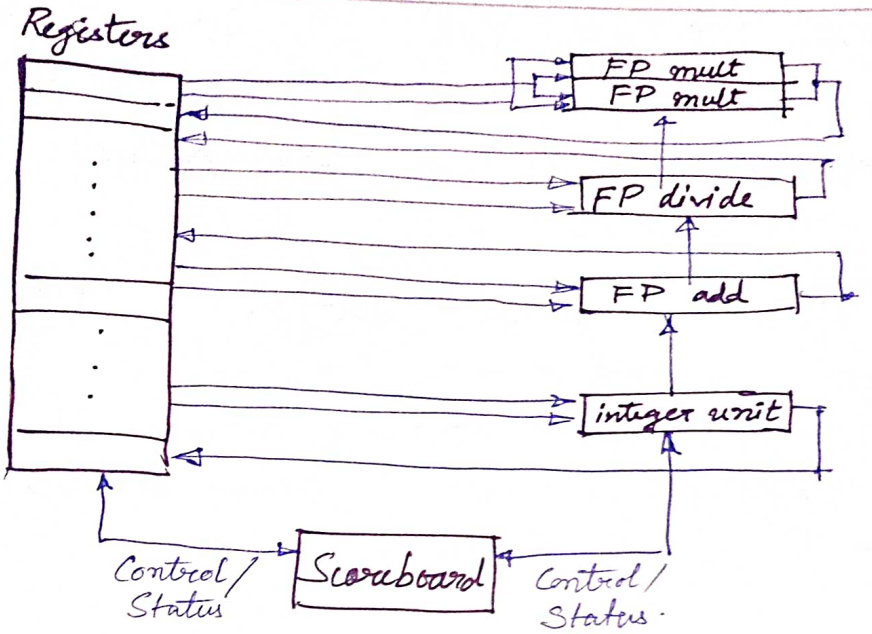
Each iteration gives us:

1. R1 ← load x[i]
2. R2 ← load y[i]
3. R3 ← R1 + R2 } "Use after load" 1 ¢ cycle stall!
4. R3 → store z[i]



Goal:

1: Start execution of 1 instruction (if possible) every ¢ cycle
2: Finish execution of as many instructions as possible every ¢ cycle.



example:

```
fmul   f1, f2, f3   ..... 10¢
fdiv   f4, f5, f6   ..... 40¢
fadd   f2, f7, f8
```

anti dependence

Decode —— (?) ——

Issue (sending an instruction to the execution unit ~~Dispatch~~, after fetching & decoding) happens in-order..... apparantly for each unit only

```
fmul   f1, f2, f3   ..... Issued to FP Mul
fadd   f1, f4, f5   ..... Not even issued to FP Add
```

↓ STALL

- For WAR → The write is issued but the permission for the later instruction to write is only given after the earlier instruction reads & finishes
- For Structural & WAW → The issue itself stalls!

গত এক ঘন্টা ধরে কিছু বুঝলাম না। অন্য ছাত্ররাও বুঝেছে বলে বিশ্বাস হচ্ছে না। এমন কি ছাত্ররাই একমাত্র না হলে থাকা অমঙ্গল বলে মনে হয় না।

কিছু একটা হচ্ছিল, হচ্ছে, হয়ে চলেছে। যা পড়ার একাই পড়তে হবে। চতুর্থ বছরে আর্কিটেকচার পড়তে হচ্ছে গত-জন্মের কোন পাপের প্রায়শ্চিত্ত করতে, তা হেনেসি বা প্যাটারসন ও জানে না!

Correction: It stores to (?!?!!?) issue in order.

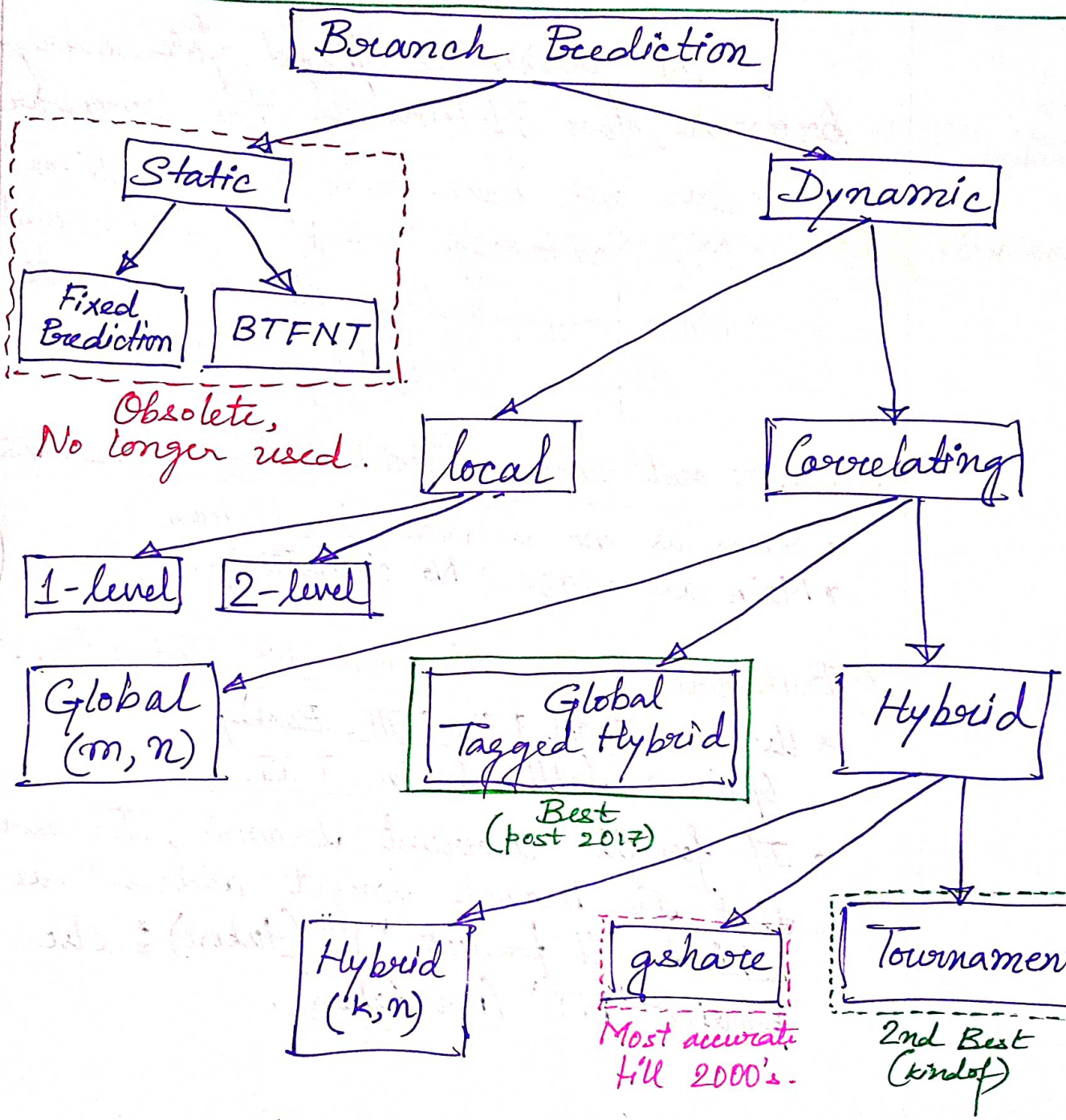
"আগে আমায় ছুটি দিতে বল!"
— অমিত্র

More correction: Issue does stop. Issue happens in order!

RAW ⟶ instruction starts, but operands are not allowed to be read.

* Once issued, a scoreboard can only restrict read-from/write-to Register file
* Scoreboard is a pure controller
* As a consequence, instructions mapped to the same execution unit are executed in order.

Branch Prediction : Hardware



Mainly two types:

- Static branch predictors (used earlier)
  - Fixed prediction
  - BTFNT (Backward Taken Forward Not Taken).
- Dynamic branch predictors:
  - Is an FSM (state machine); changes state on misprediction.

Static Branch Prediction:

The policy is fixed for every branch, for lifetime of the processor.

- Does not learn from mispredictions
- Main advantage: Simple to implement.
- Main disadvantage: Not so accurate (often control hazard causes stalls).

1. Predict not taken (Intel i486) — ALWAYS!

* Same as no branch prediction
* Main advantage: No circuitry.

2. Backward Taken, Forward Not Taken (BTFNT)

* Used in Intel PII, PIII, ~~Backup~~ Optional fallback in PIV.
* If for the current branch, I observe that the branch target address is behind, I predict "T" (taken); else predict "NT" (Not taken).

Dynamic Branch Prediction

Observe branch outcome, predict based on observed outcome, learn from mistakes.

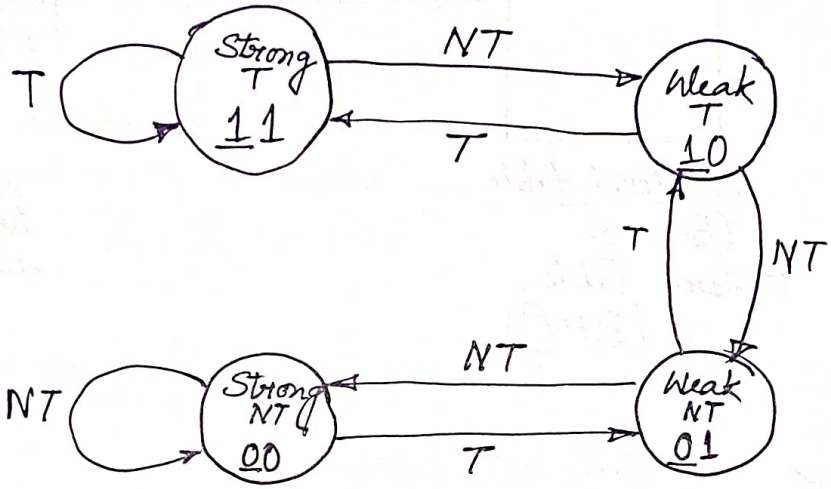
- 1 bit: predict last outcome
- 2 bit: change prediction only on 2 consecutive mispredictions.

Dynamic Branch Predictors:

1) Local-1-level:

Branch prediction FSM. e.g: 2-bit FSM / 3-bit FSM or higher.

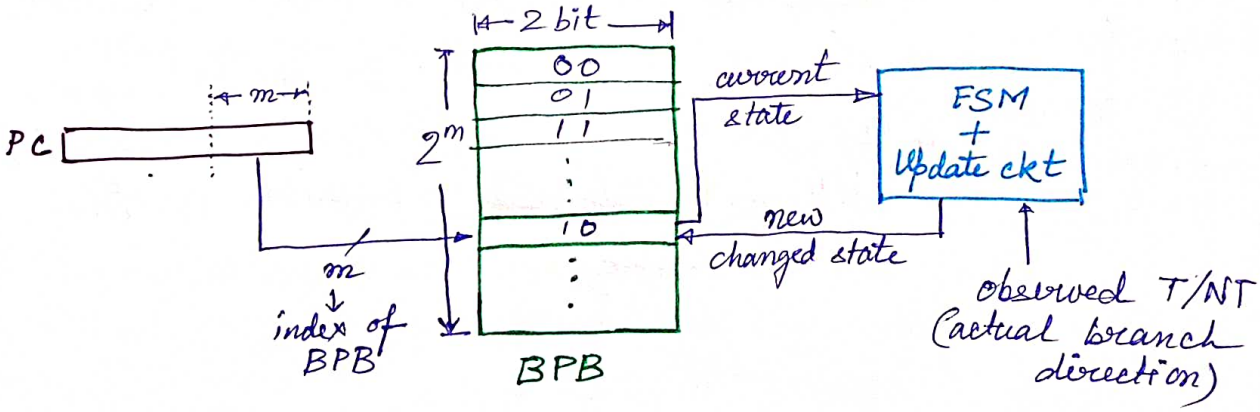
e.g: 2-bit FSM: (MSB is the prediction), Moore Machine



Branch Prediction Buffer (BPB):

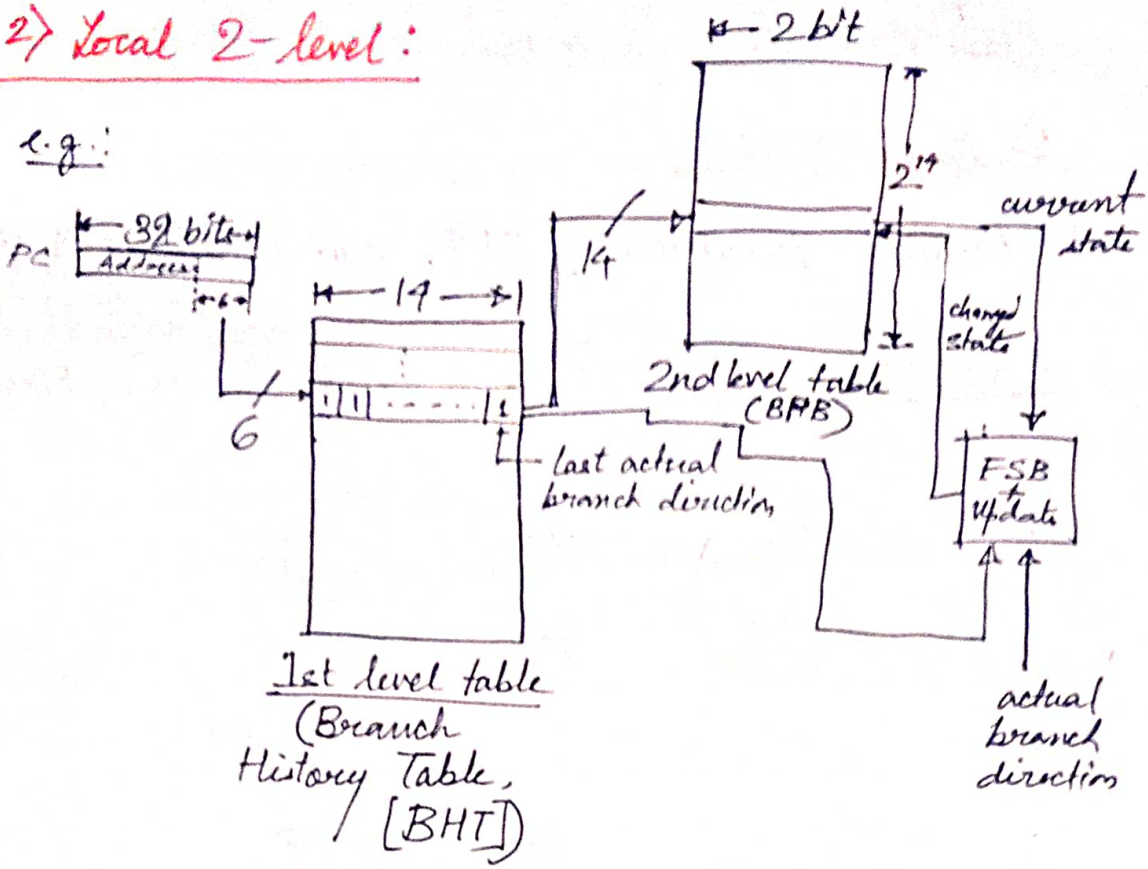
(Confusingly it is also called Pattern History Table (PHT) although this is not exactly pattern history.)

* Contains FSM states:



* DM Cache
* There is no tag — allows aliasing.
  (The index bit might cause aliasing among branch addresses B1, B2 where B1[m-1:0] = B2[m-2:0]

2) Local 2-level:



* Aliasing is still not gone
* (History can have a mixture if two branches B1 and B2 have the same last m bits).
* Update on BPB
  - depends on actual branch direction
  - depends on history as well.
  - If an entry of BPB is read it will get updated (possibly to the same value).

Example:

```
B1: if (aa == 2) aa = 0;
B2: if (bb == 2) bb = 0;
B3: if (aa != bb) {---}
```

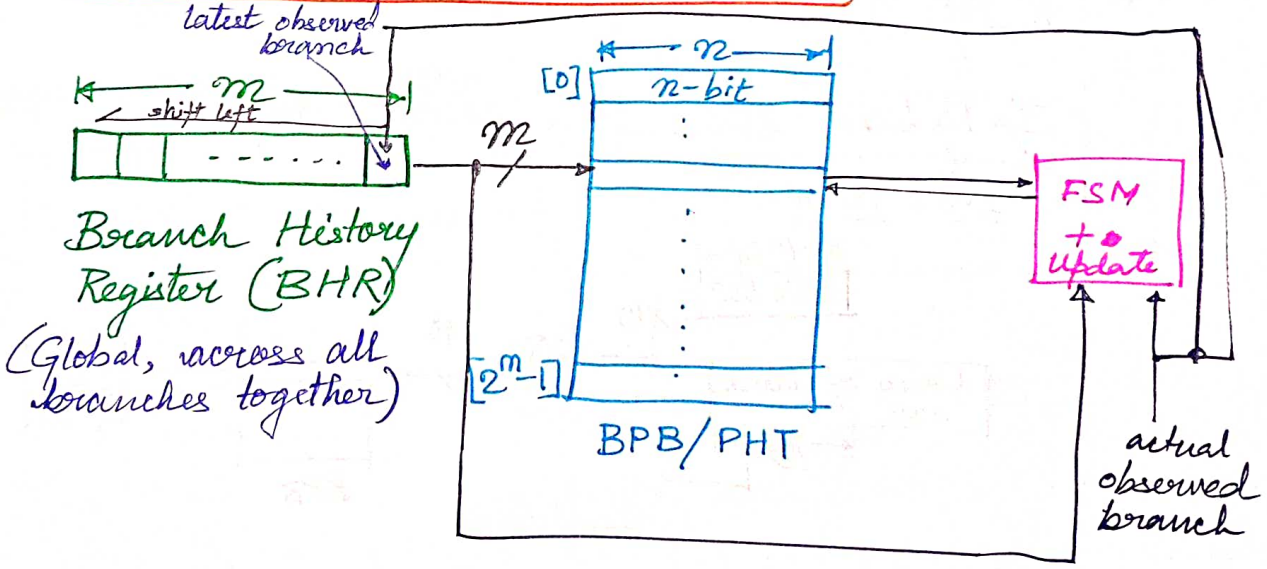
Observation: If B1 & B2 are taken ⇒ B3 is not taken

⇒ Behaviour of B3 is "correlated" to the behaviours of B1 and B2.

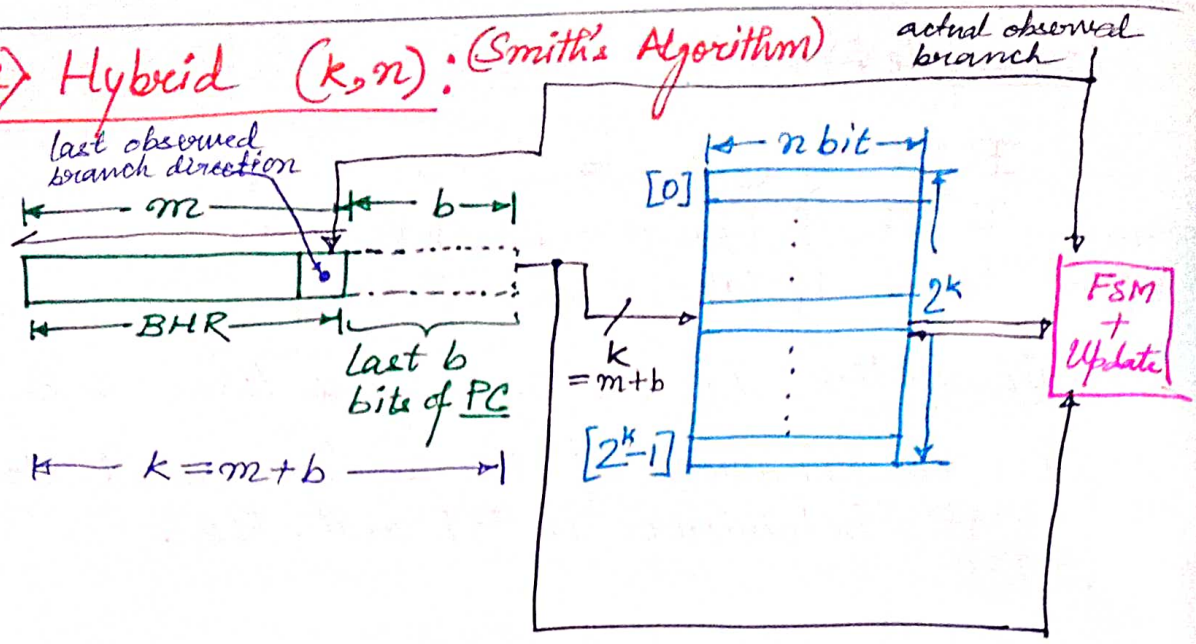
These correlations give us the idea of correlating branch predictors.

Correlating Branch Predictors:

1) Global (m, n) branch predictor:



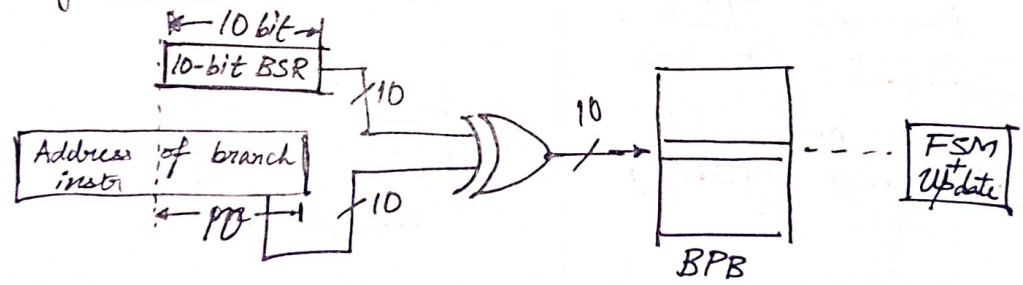
2) Hybrid (k, n): (Smith's Algorithm)



* Unconditional jumps (jsr) are always taken, then any prediction of NT would be wrong. This can be avoided by seperately noting down jsr instructions.
* If k is fixed then $m \uparrow \Rightarrow b \downarrow$.

3) Hybrid Correlating: ~~gselect~~ gshare [Macfarling, 1993]:

e.g: 10 bit:



* Insight: LSB's of branch instruction address & BHR is uncorrelated.
* Best for ~20 years despite its adhoc nature

Tournament Predictor

Idea:

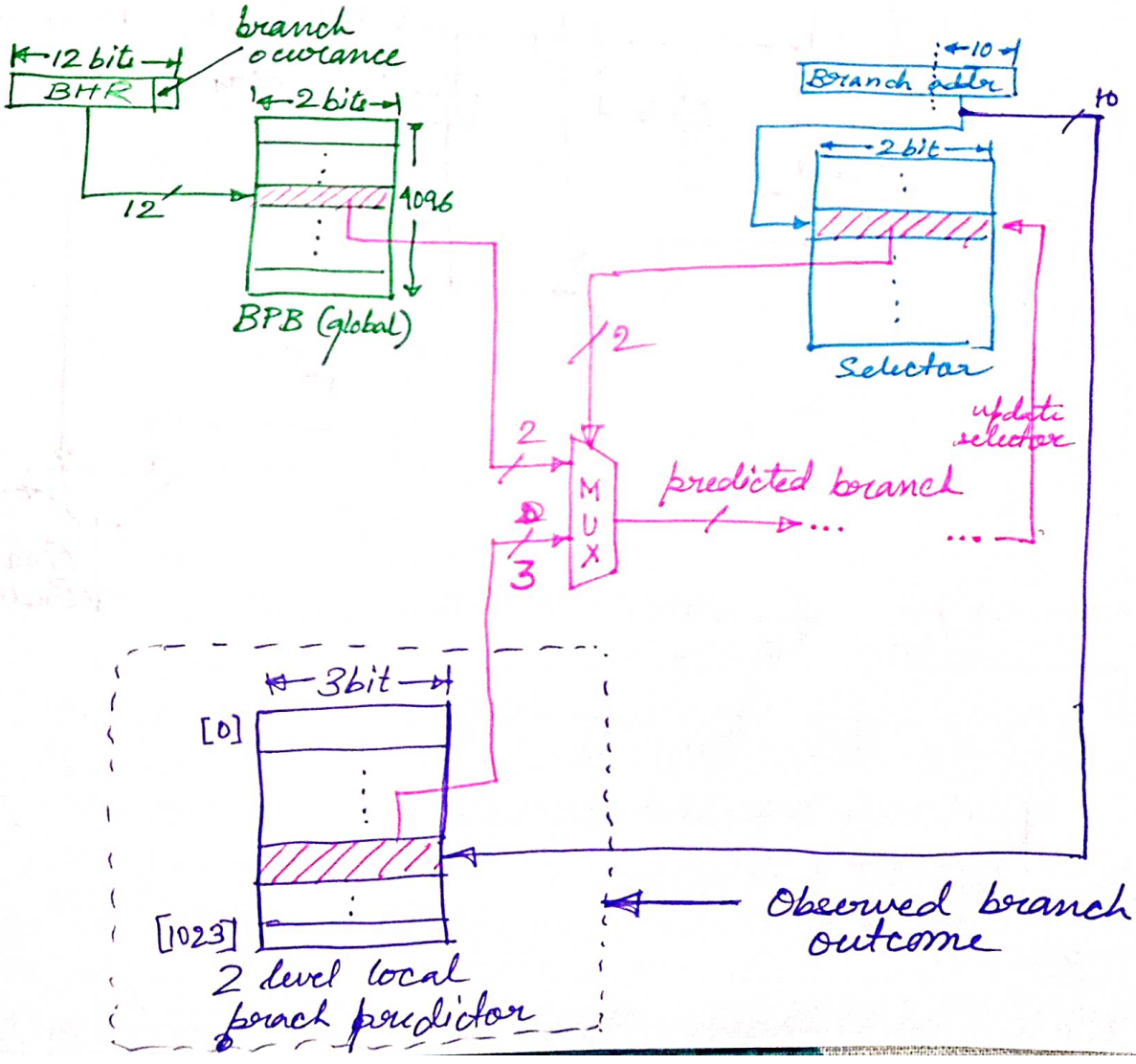
(i) Have both local & global predictor

(ii) Local predictor: 2-level

(iii) Dynamically (during runtime), by observing the recent performance (accuracy of prediction) select either global/local predictor.

[Change predictor type after 2 mispredictions]

Example: BHR is 12-bits,
Global predictor BPB is 4096 entries,
each two bit.

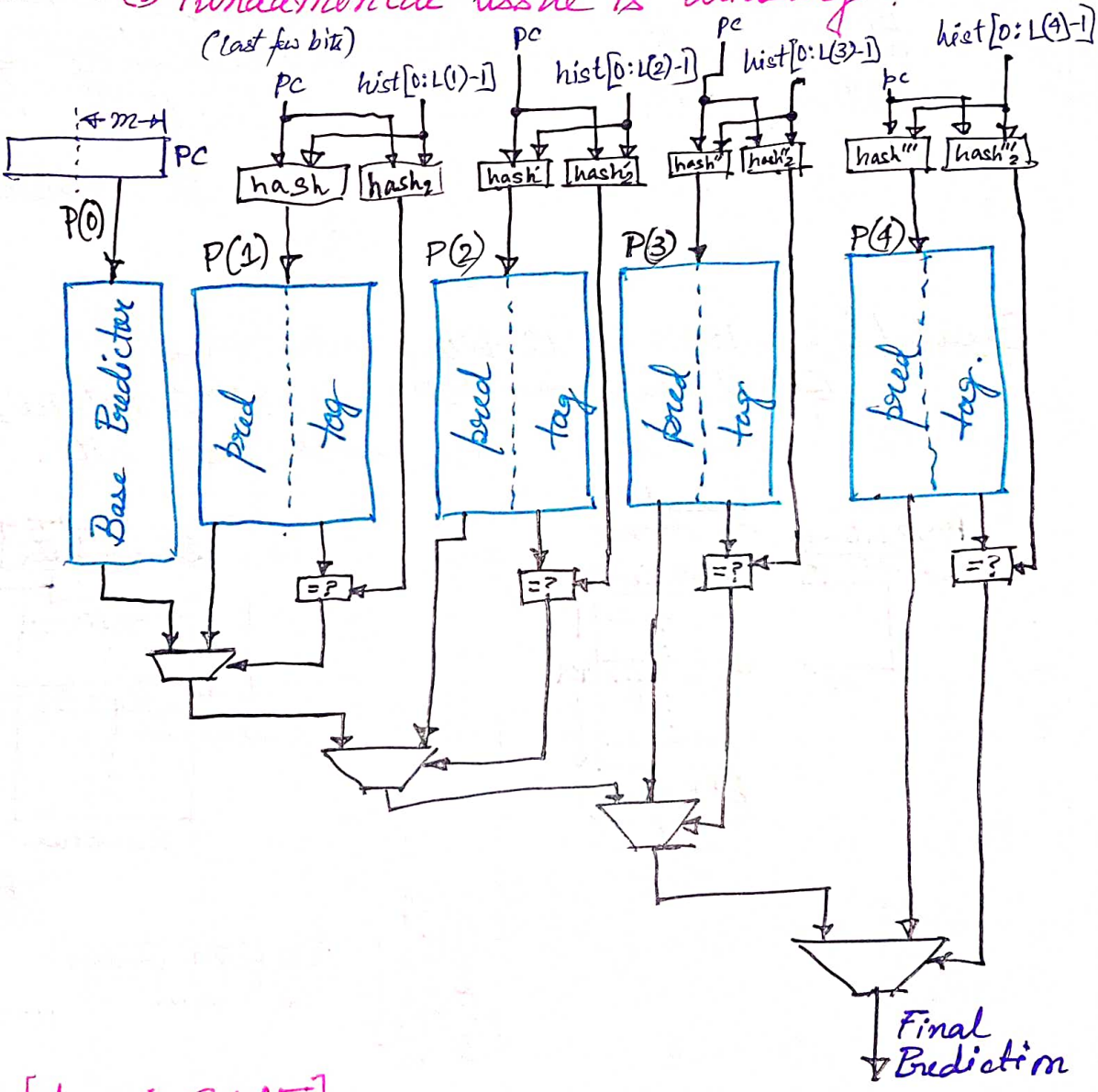


Tag Hybrid Predictor:

* Widely believed that modern processors do this.

* Motivation:

(i) Fundamental issue is aliasing.



[tag ≤ 8 bits]

As compared to Tournament, with equivalent hardware overhead, achieves better prediction. ≈ 100%.

Instruction Level Parallelism : Hardware Techniques

As of now, issue is done once per clk cycle
⇒ CPI ≥ 1, IPC ≤ 1.

Tomasulo, IBM, 1967 : Tomasulo's Algo.

* Differences from Scoreboard : (Better Phrased in slides)

- Communication between units are direct (can also be through Reg. File)
- No central controller (execution units can ~~be~~ have some amount of hazard detection)
- Every unit has local buffer (once read, don't need to read again).

* Similarities

- Instruction issue in order.
- Issue stalls for structural hazard.
- Read Operand & write to Registerfile can be out of order.

Example : [WAR]

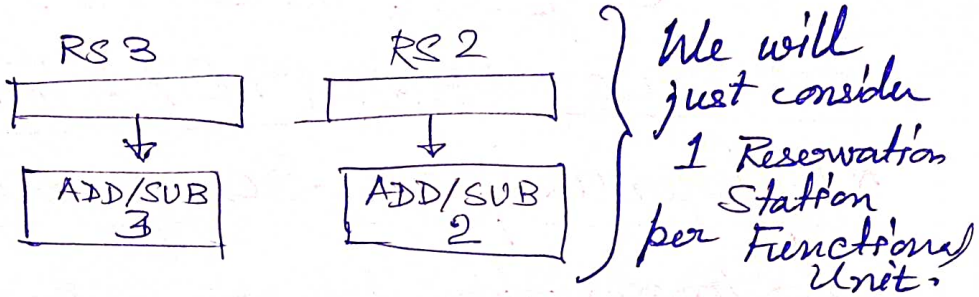
```
add  R1, R2, R3
sub  R2, R4, R5
```

~~R2 co~~
add can read R2 into local buffer (at the very beginning)
& sub can write to R2 ~~bef~~ anyway, anytime! (yey!).

২৫।১১
২(ল)

Diagram & beyond.
↳ Slides

Reservation Stations cannot communicate with each other except CDB (central data bus).



* Steps of Tomasulo: (check slides)

1. Issue (or Dispatch):

- Perform renaming to avoid WAW/WAR
- When things go to reservation stations, register names have gone.

2. Execute:

- When operand becomes available (check by monitoring CDB), store those in reservation station
- Execute when all operands are available. (this delay in execution until operands are available, ~~same from RAW~~ avoids RAW hazards).

3. Write Result

⋮

In Tomasulo, issue stops only on structural hazard (not even when WAW hazard is possible).

⋮

More slides.

⋮

"বেনিয়ে গেছো তুমি আমার মাথার ওপরে
আমার চুলগুলি গায়ে সতন
বুঝতে পারিনি তোমারে।"

⋮

Branch Target Buffer (BTB)
on slides.

Branch addr.	Branch target addr.	Instr @ target
A1	A2	add $1, $2, $3

আজিকে বুঝিবার চেষ্টা করিব। জনসংখ্যা এত কম কেন কে জানে!) গুমাজগুলোর ভয় লুকিয়া থাকা অনুভব আজ আকার অবয়ে জাগিয়ে উঠিবে!

- In classic 5 stage pipeline, write back is in order.
- commit of Tomasulo with speculation is also in order.
- lw/sw latencies are in MEM stage.

stall: issue stops (or in classic 5 stage pipeline, IF stops).

USE AFTER LOAD

We consider the following instruction.

				1	2	3	4	5	6	7	8	9
lw	x1	0(x2)		IF	ID	EX	MEM	WB				
add	x4	x3	x1		IF	ID	stall	EX	WB			
sub	x7	x5	x6			IF	stall	ID	EX			
mul	x10	x11	x12					IF	ID			

BRANCH HAZARD (RISC-V), (Branch cond. eval : EX)

Instr # \ clock	1	2	3	4	5	6	7	8	9	10
i beq x1 x2 tar	IF	ID	EX	MEM	WB					
pred: NT i+1 add . . .		IF	ID	nop	nop	nop		•		
i+2 sub . . .			IF	nop	nop	nop	nop			
tar: i+300 add . . .				IF	ID	EX	MEM	WB		

mis-prediction

Finishing in cycle 8 instead of 6

(Slides again)

Multiprocessor Computers and Thread Level Parallelism

Instruction level parallelism

[Adv] throughput increase, transparent to programmer.

[Disadv]

* (in slides)

We must look at other sources of parallelism.

Parallelism exploited by Software

Data-level parallelism: Operating on several data items at the same time, in the same task.

Task-level parallelism: create & operate multiple tasks (independtly or cooperatively)

Parallelism exploited by Hardware

Instruction level parallelism (ILP): DLP in modest level

Vector processors / GPU's / multimedia instruction set: DLP

Thread level parallelism: DLP/TLP

Request level parallelism: DLP/TLP

Flynn's Taxonomy: (Slides)

SISD / SIMD / MISD / MIMD

Single instr. stream
Single data stream

(Slides)

(More slides T_T)

(more slides)

If a local processor wants to update contents of a local memory, it sends a signal to others and others invalidate in their own cache. The ~~original~~ initial local processor becomes the owner of the block.

If a processor finds a block to be invalid, it sends a signal, asking for that block. If someone (the owner) sends the block —— great! Otherwise it takes from LLC (last level cache)

(MSI in slides)

[Ek Slides,
Paani me gyi
Chapaak!]

"Syllabus is the entire syllabus, stress will be given on stuff taught after midsem

Hardware Support for Synchronous Primitives

"Suppose Processor P1 does this in an infinite loop & P2 does that in an infinite loop"

```
P1:  while (1) { count++; }
P2:  while (1) { count--; }
```

variable count is shared between P1 and P2. Both can R/W. So basically, the following happens (&A be address of count)

P1		P2	
lw	X1, A	lw	X2, A
addi	X1, 1	addi	X2, -1
sw	X1, A	sw	X2, A

Ideal case:

At time $t = t_0$, count == 5
At time $t_1 > t_0$, P1 executes, count ← 6
At time $t_2 > t_1$, P2 executes, count ← 5

this is what is expected; but here we may the (incorrect) assumption, that P1 executes entirely, then P2 executes.

Let us see what can happen in reality.

Possible Actual Execution Sequence:

```
P1:  lw   X1, A    // X1 == 5
P1:  addi X1, 1    // X1 == 6
P2:  lw   X2, A    // X2 == 5
P2:  addi X2, -1   // X2 == 4
P1:  sw   X1, A    // count = 6
P2:  sw   X2, A    // count = 5   !!! aaaaa!!!
```

This is called a data race, where data depends on the actual sequence these set of instructions are executed.

"To avoid data races, you must incorporate synchronization in your processes."

The framework of a process:

Process P:

```
do {  ...
        entry section
        critical section;   // Process potentially performs
                            // Read/Write operations on
                            // one or more shared
                            // variables in memory
        exit section
      ... (remainder section)
} while (1);
```

"ও মন, কখন শুরু কখন যে শেষ, কে জানে
এ যে বাজিকরের খেলা রে মন, বাজিকরের খেলা রে মন
যার খেলা হয় সে জানে!"

Simple Hardware Based Solution

This requires atomic operation (basically assume that the instruction set implements the same thing without interruption).

"This assumption is really aggressive & difficult to acheive in modern processors."

The Process framework becomes:

```
do {
        ⋮
        [ acquire lock ]   //atomic, 'lock' is shared
        critical section
        [ release lock ]   //also atomic
        remainder section
} while(1);
```

There are two methods to acheive this

1) TestAndSet Atomic Operation:

```
bool TestAndSet(bool *target){
        bool rv = *target;
        *target = true; //causing it to spin
        return rv;
}
```

"rv probably means 'resistor value' and hence not $\frac{2}{3}^{rd}$ of RVS's initials."

We can use it as follows:

```
bool lock = false ; // global variable shared
                    // across process
```

The 'lock' shared variable is initially false. For each process, we have the following fragment of code:

```
do {
        while (TestAndSet(&lock)){
                ; // do nothing & spin
        }.
        <execute critical section>
        lock = false ;
        <execute remainder section>
} while (1);
```

"জীবন যাত্রার প্রতি পাতায়, যতই লেখা হিসাব নিকাশ,
কিছুই বাদ না ..."

TestAndSet returns false the first time, & hence the first process gets to execute critical section. (say P1). Others who want to enter in their own critical sections, waits by spinning (until P1 releases the lock by [lock = false] instruction).

This TestAndSet was available in many old ISA's as a single instruction. It is no longer popular due to the fact that two memory accesses are required.

2) Atomic Swap/Atomic Exchange: (EXCH)

```
void swap (bool *a, bool *b){
    bool temp=*a; //load, a in mem
    *a = *b; //store, b is in reg.
    *b =temp;
}
```

We use this in the following way:

```
bool lock=false;
```

Each process executes the following code segment.

```
do{
    bool key=true; // local variable
                   // mapped to register
    while(key){
        swap(&lock, &key);
    }
    <execute critical section>
    lock=false;
    <remainder section>.
} while(1);
```

Here basically, the key is (of at most one function) captures the locks; others wait, till the key release the lock.

Again, this is too complicated a design to implement as a single instruction for modern day processor.

We will look into an alternative which kind of takes care of the atomicity assumption. We will look into a special load and a special store. The design is such that if these two instructions are not executed in a ~~non-interrupting the~~ non-interruptive way, there will be a mark. We can understand that something went wrong and we will consider this to be 'lock not acquired'.

"This is a defensive measure & there is nothing better that we can do."

We will have a look how such a pair of instructions can simulate EXCH.

Special Set of Instructions

1) Load Linked (LL) / Load Reserved (LR):

lw.d or lw.w.

lw.{d/w} <Reg>, <Address of memory contained in a register>

example: lw.w R1, Ø(R2)
or, lw.w X2, (X1)

There is a special (non-architectural) register "Link Register"/"Reserved Register" (R_{lr})

∴ for lw.w X2, (X1), the following are done

$X2 \leftarrow MEM[X1]$ and $R_{lr} \leftarrow X1$.

In other words, "lr has created a reservation on MEM[X1]". This is because of the following reason.

> If an interrupt (including context switch) happens after lr, for the cache block to which MEM[X1] is invalidated, R_{sr} is set to 0 forcibly.
> $R_{sr} \leftarrow 0$, happens by force!

2) Store Conditional (SC)

```
sc.{d/w}   <Reg>, <Address in Reg>
```

Example: sc.w X3, (X1)

This is very complex .... T_T

~~[Step 1] SC checks value of Rsr.~~

```
if (Rsr ≠ <Address_argument>) // arg 2 of sc
        X3 ← Non-zero value  // sc failed
else { MEM[X1] ← X3;
       X3 ← 0; } // sc passed (yeyyy)
```

for sc.w X3, (X1)

```
if (Rsr ≠ <Address_argument>) { //2nd arg of sc
        X3 ← Non-zero value; //sc failed
} else{  //sc passed
        MEM[X1] ← X3;
        X3 ← 0;
}
```

[Example] Atomic EXCH using LL/SC

EXCH: MEM[X4] ⟷ X4. atomically

LL/SC:

```
try:  mv    X3, X4     // X3 ← X4
      lr    X2, (X1)   // X2 ← MEM[X1], R ← X1
      sc    X3, (X1)   // If sc succeeds,
                       //    MEM[X1] ← X3
                       //    X3 ← 0
      bnez  X3, try    // try again if sc failed.
      mv    X4, X2     // X4 ← X2 (=MEM[X1])
```

If there was an interrupt in between lr & sc, I would try again.

So in five instructions, we can 'effectively' implement an atomic exchange

"সে এক অন্যকথাতেই হোক, কখনো দেখা হয়না তবু শেষ!"

We will now look into the atomic fetch & increment operation (done using LL/SC).

[Example] Atomic Fetch & Increment using LL/SC

We want to acheive the following atomically

$$MEM[X1] \leftarrow MEM[X1] + 1.$$

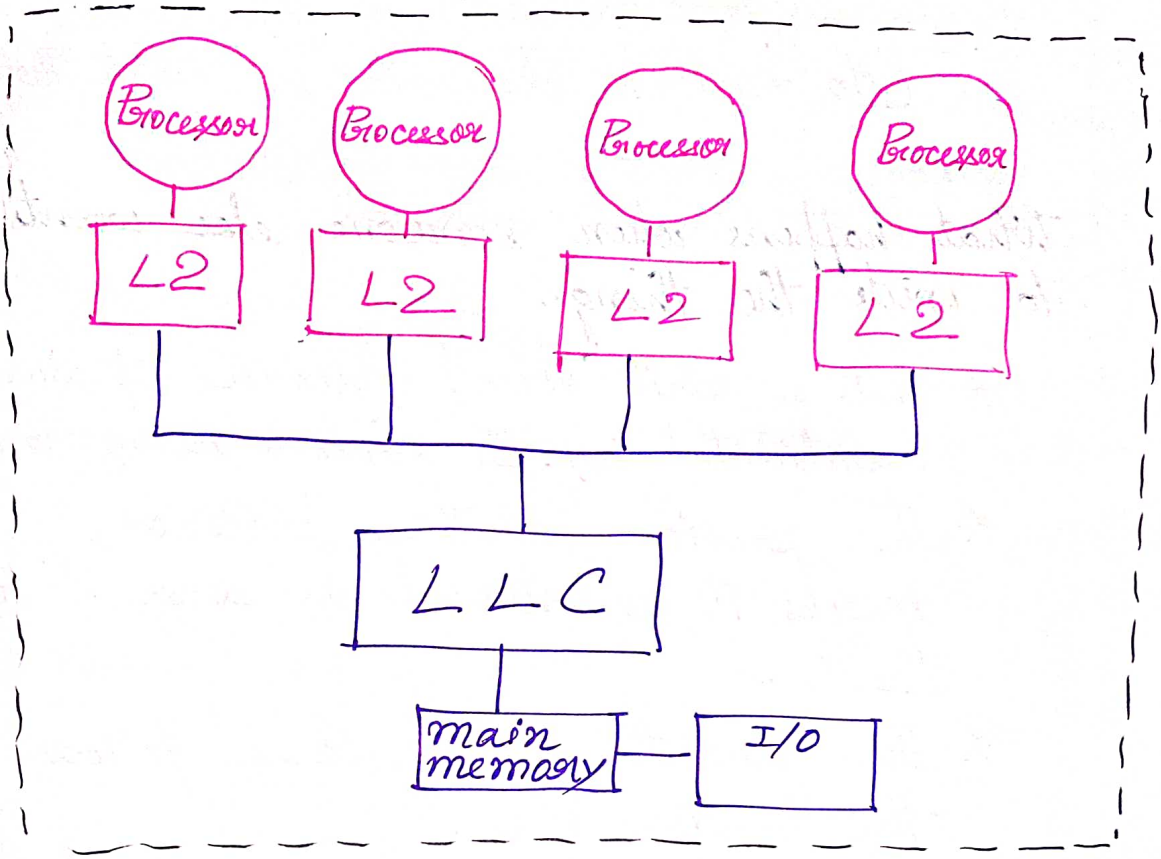
```
try:  lr   X2, (X1)
      addi X3, X2, 1
      sc   X3, (X1)
      bnez X3, try
```

Basic Rule of Thumb:

* Completely avoid memory access between lr & sc.
* Try to add only simplest register operations between lr & sc. Stuff like exceptions (divide by 0) is problematic.
* Best case is to put nothing in between lr and sc.

IN THE EXAM: You may be asked to write other atomic operations with these lr/sc command given their intended behaviors

(Back to MSI slides [T_T])



When a processor wants to write something:

* send invalidate-message to everyone (broadcast)
* other processors invalidate the block immediately.
* Hardware based mechanisms are there to arbitrate and break ties
* After ~~processor writing~~, this, the writing processor marks this cache block as "modified" and claims ownership.

* Writing processor writes to their L2 (without updating LLC or other ~~L1/L2~~ L2 caches).
(if it immediately modifies to LLC, it is a write through scheme).
(No modern ~~CPU~~ caches are write through)

What happens when someone else wants to write the thing.

* Gets a write miss! (because it was invalidated it). It sends a message to everyone
* The previous writing processor, marks it as shared as sends it to LLC.
* The present writing cache follows the previous steps.

Optimization (MESI):

E: (Exclusive): only the owner has this block and is not dirty.

[LLC maintains directory ~~who~~ of which all L2 caches have the block]

"SHARED" state is Read Only!

"In the MSI state diagram (for one block), the 'exclusive' is actually 'modified'."

এ জীবনে যত কিছু চেয়েছি, মন বলে তারও বেশী পেয়েছি, পেয়েছি

[Slides with 'blue' & 'orange' table]

"LLC writing takes a lot of time."

"On a 'modified' block request, the owner writes to LLC, but also directly provides it to the seeker."

আমি যত বলি তবু
এমনি রোজ যেতে হবে
দুয়ারে দাঁড়াতে বলে
না, না, না
যে যে যাবে [illegible]

"Caches are well behaved"

"All actions on cache blocks are done by local cores on response to some messages."

"Processors keep on listening."

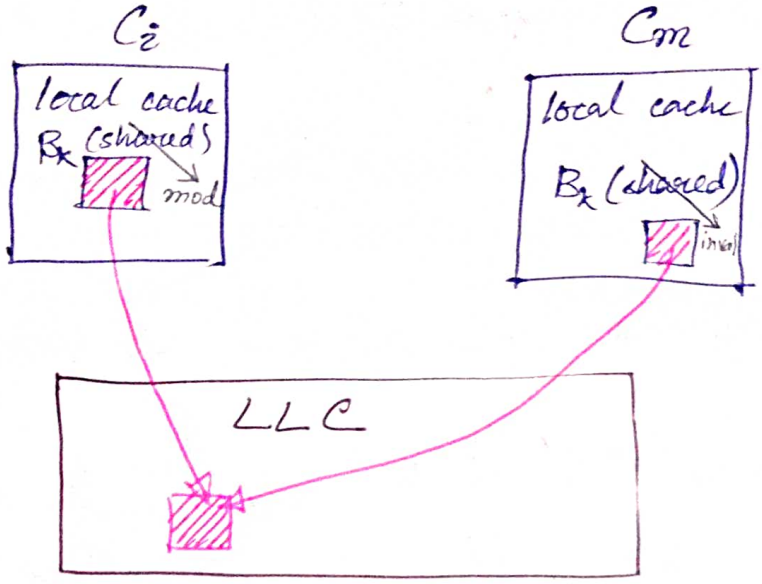
আমি আকাশে পাতিয়া কান
শুনেছি শুনেছি তোমারই গান।

"This micro architecture is confusing."

কোথাও আমার হারিয়ে যাওয়ার নেই মানা
মনে মনে

Coherence Miss:

কবিরও কবিতা ভাবই, তুলি দিয়ে আঁকা ছবি



* C_i wants to write $W_t \in B_k$
* C_i invalidates (request to invalidate) B_k in C_m
* If C_m wants $W_t \in B_k$
  - ↳ Read miss [True Sharing Miss]
* If C_m wants $W_t \in B_k$, $W_t \neq W_t$
  - ↳ Read miss still happens.
    [False Read Miss]

"Now that you've grown up
you understand there is no Santa"

Directory Protocol

Based → Coherent

[slides]

For each ~~directory~~ LLC block, store in directory:

- A bit vector with (#cores) bits.
- Denoting if the core has a private copy in the L2 cache.

"Go through details of this"

Back to critical section

Solution to Critical Section Problem:

Most common solution:

```
do {
    [acquire lock]   // atomically (try to)
                     // acquire lock, if ~~no~~
                     // cannot acquire lock,
                     // loop ("spin").

    <critical section>   // only one process
                         // can execute its
                         // critical section at
                         // a time

    [release lock]

    <remainder section>
} while (1)
```

We will assume "effectively atomic" instruction for EXCH

We will also ~~talk~~ base our discussion considering the shared multi-core "write-invalidate" protocol!

Solution 1. (Classic Solution)

Assume that the lock variable is located at 0(X1), in memory. Assume X0 = $0, contains the value 0 always.

```
// entry section (can enter if lock == 0)

        addi   X2, X0, 1
lockit: EXCH   X2, 0(X1)   // try (pseudo) atomic EXCH
        bnez   X2, lockit  // if lock not acquired, spin
```

Now, say, P1 successfully performs EXCH.

* P1 "holds" lock in its local cache, ownership.
* P2 "wants" lock.
  - It will exchange X2 to 0(X2).
  - lock will not be ~~aq~~ acquired, (as both X2 and 0(X1) hold value of 1)
  - However, this writes 1 to 0(X1) anyway
    - Other caches (of P3, P4, e.t.c. and P1), are invalidated

This causes unnecessary crowding in the common bus.

Solution: ~~Change over~~ Only perform write when lock value is 1.

So the issue was: Every process/thread, which performs a spin to acquire the lock, must send an invalidate message on the bus covering its EXCH. However such invalidates are useless to the originating process/thread, if the lock value in the memory is already 1.; This happens ~~value~~ on every spin.

Solution 2.

Do not send invalidate messages unnecessarily. First check if lock variable is zero. Only if it is zero, perform EXCH

```
lockit : ld.   X2, 0(X1)   //lock at 0(X1)
         bnez  X2, lockit  // lock not available
         addi  X2, X0, 1   // X2 ← 1
         ~~bnez~~
         EXCH  X2, 0(X1)   // same as before
         bnez  X2, lockit
```

"Invalidate messages are not diminated, the volume just decreases by a lot."

Notice that EXCH performs load (of the lock variable) anyway. So why not use that, instead of keeping another ld.?!

Solution-3. (Optimized Solution)

We would use the lr in the EXCH pseudo atomic operation to read the value. Another way of looking into it is that, we slightly modify EXCH.

```
lockit :  lr    X2, 0(X1)
          bnez  X2, lockit
          addi  X2, X0, 1
          sc    X2, 0(X1)
          bnez  X2, lockit
```

← could had been dangerous if this gets "memory access violation". (Won't happen here, cause lockit is right here, i.e. Trustable)

We now hand-simulate this on 3 processor (example in text-book).

Hand-Simulation :

step #	P0	P1	P2

তার যা ছিল বুকে গান
শোনবার শুনে নাও
মনে রেখো, ভুলে যেয়ো
তার এই শেষ গান
গান দুনন ভরিয়ে দেব
ভেবেছিল একটি পাখি

[P.T.O]

Step #	P0	P1	P2	State of cache block to which lock is mapped	Bus Activity
1.	has lock=1 in L2 of P0 (in critical section)	spins to check lock==0, gets read miss	same as P1	shared	P0 provides P1/2 in arbitrary order; P0 writes back to LLC
2.	lock ← 0	invalidate	invalidate	ex/mod, inval, inval,	P0 sends invalidate
3.	<don't care>	tries to perform LR; read miss ~~gets hold of lock from P0~~ wait for bus	same as P1, but gets the bus first to send read miss message	inval in P2. shared in P0, P1	P2 gets hold of bus, sends read miss. P0 gives lock value (=0) to P2, P0 writes back to LLC
4.	<don't care>	~~lock==0, test succeeds~~ wait for bus	~~wait for bus~~ test succeeds	~~shared, shared, inval~~ shared, inval, shared	meh...
5.	<don't care>	~~execute swap~~ get hold of bus send read miss, receive from P2	execute swap receive read-miss from P1. send to P1		

:(Windows ran into an error

4.	<don't care>	wait for bus	test succeeds (lock == 0)	shared, inval, shared	meh...
5.	<don't care>	~~tries to execute~~ swap gets hold of bus, gets lock val, test lock==0	tries to execute swap, gets cache miss (as shared is read-only)	all shared.	P1 got the lock value.
6.	<don't care>	Execute swap, but got ~~read~~ cache miss as P2 has invalidated.	Performs swap after changing local cache to MODIFIED. set lock ← 1	P2: Modified P0, P1: Invalid	Invalidate from P2.
7.	<don't care>	swap completes, gets lock==1, set lock=1 (after sending invalidate on bus)	enters crit. sec.	P1: modified P0, P2: inval.	P1 sends invalidate over bus.
8.	<don't care>	spins on local copy	continues crit sec	P1: modified P0, P2: inval	meh...

——— END OF COURSE ———

বহে নিরন্তর অনন্ত ধারা