6/1/20

Measurement Techniques for Performance

- Latency → (state transition → done) time
- Throughput → (# operations / second)

} Throughput $\propto \frac{1}{\text{Latency}}$

Ex Car Assembly

chasis → engine → wheel → producing the car

20 stages latency → 4 hrs

Throughput

* 20 stages → 4 hours ⇒ 1 stage → 1/5 hours
* If last stage goes 5 times per hour, so Throughput = 5 cars / hour

$\frac{1}{\text{Throughput}} \neq \text{Latency}$.

* **Comparing Performance:**

"X is n times faster than Y"

Speedup → $n = \frac{\text{Speed}(X)}{\text{Speed}(Y)} \Rightarrow \frac{\text{latency}(Y)}{\text{latency}(X)}$ OR $\frac{\text{Throughput}(X)}{\text{Throughput}(Y)}$

Q Laptop A takes (4 hrs) to compress a video.
New laptop B takes (10 mins) to compress a video. → Execution time.

Speedup → 24

⇒ Performance of a computer system / architecture is measured wrt benchmark programs.

- Programs & inputs which are standardized by stakeholder, academics, industry.

Benchmark Suites

- # of instructions / cycle → Peak performance.

* Engineering workstations → SPEC benchmarks
  Not very I/O intensive → more processor dependent

Consists of → gcc → software development workload compiler
BWAVES, LBM → fluid dynamic applications
PERL → String processing applications
CACTUS ADM → Physics – simulators
BZIP → Compression
XALAN BMK → Parser for XML

	Comp X	Comp Y	Speedup
App A	9s	18s	2
App B	10s	7s	0.7
App C	5s	11s	2.2
Avg time	8s	12s → 1.5	1.63
Cube roots	7.66	11.15	→ 1.45

$\sqrt[3]{2 \times 0.7 \times 2.2}$

$\sqrt[3]{9 \times 10 \times 5}$

Geometric means for the ~~ratio~~ average

Q machine A → machine B

=

App 1 → speedup of 2
App 2 → speedup of 8

Overall avg speed → $\sqrt{2 \times 8} \rightarrow 4$

Iron Law of Performance

CPU Time → # of inst/program × CPI × clock cycle time

$$\frac{\#\text{inst}}{\text{prog}} \times \frac{\text{cycle}}{\text{inst}} \times \frac{\text{time}}{\text{cycle}} \rightarrow \frac{\text{time}}{\text{prog}}$$

Q Program executes 3 Billion (3×10^9) inst, CPI = 2, clock is 3GHz

CPU Time → $3 \times 10^9 \times 2 \times \frac{1}{3 \times 10^9} \rightarrow 2s$

8/1/12 RISC Management - SD → Cont {TODO}

9/1/12

Q 500 instructions:

	CPI
10B	4
15B	2
5B	3
20B	1

Clock → 4GHz
Exec time.

CPU Time → $\left(\sum \left(\frac{\#\text{inst}}{\text{prog}} \times CPI_i\right)\right) \times \frac{1}{\text{clock}}$

AMDAHL's LAW:

$$\text{Speedup} = \frac{1}{(1-f) + \frac{f}{s}}$$

f → fraction enhanced ⇒ part of code to be enhanced → taken w.r.t original exec time

S → speedup enhanced

Q Program with 50B instructions, clock freq is 2GHz.
Improve Branch instr. CPI_{Branch} goes from 6 to 2

Inst type	# inst of overall instructions	CPI
Int	40%	1
Branch	20%	4
Load	30%	2
Store	10%	3

* cannot apply amdahls law w.r.t given %'es as they are in terms of # of instructions, not in terms of original exec time.

# inst	Exec time
20	10
10	20
15	15
5	7.5

$\Rightarrow \frac{20}{52.5} = f$ $\quad \frac{1}{\frac{32.5}{52.5} + \frac{10}{52.5}} \Rightarrow \frac{52.5}{42.5} \Rightarrow \frac{105}{85} \Rightarrow \boxed{\frac{21}{17}} \Rightarrow 1.24$

Ex Enh1 → Speedup of 20 on 10% of time → $\frac{1}{0.9 + \frac{0.1}{20}} \Rightarrow \frac{20}{18.1} \Rightarrow 1.105$

Enh2 → Speedup of 1.6 on 80% of time → $\frac{1.6}{0.32 + 0.8} \Rightarrow \frac{1.6}{1.12} \Rightarrow 1.43$

* Even if we had ∞ speedup on 10% of time then speedup → 1.11

Inst type	% of time	CPI
Int	40	1
Branch	20	4
Load	30	2
Store	10	3

Possible improvements
1. Branch CPI 4→3
2. Clock freq ↑ from 2 to 2.3 GHz
3. Store CPI 3 to 2.

1. Branch 20% → Speedup $= 4/3$ → $\frac{1}{0.8 + \frac{0.2 \times 3}{4}} \Rightarrow \frac{1}{0.8 + 0.15} \Rightarrow \frac{1}{0.95} \Rightarrow 1.05$

2. Here speedup → $\frac{2.3}{2} \Rightarrow 1.15$

3. Store 10% → Speedup $= \frac{3}{2}$ → $\frac{1}{0.9 + \frac{0.1 \times 2}{3}} \Rightarrow \frac{3}{2.7 + 0.2} \Rightarrow \frac{3}{2.9} \Rightarrow 1.035$

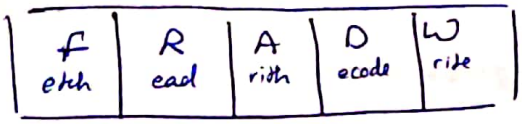
Improvement of 90% of time. Speedup → 2
Slowing down the rest by 10×.

$$\text{Speedup} \Rightarrow \frac{1}{\frac{0.1}{0.1} + \frac{0.9}{2}} \Rightarrow \frac{1}{1 + 0.45} \Rightarrow \frac{1}{1.45} \Rightarrow 0.7$$

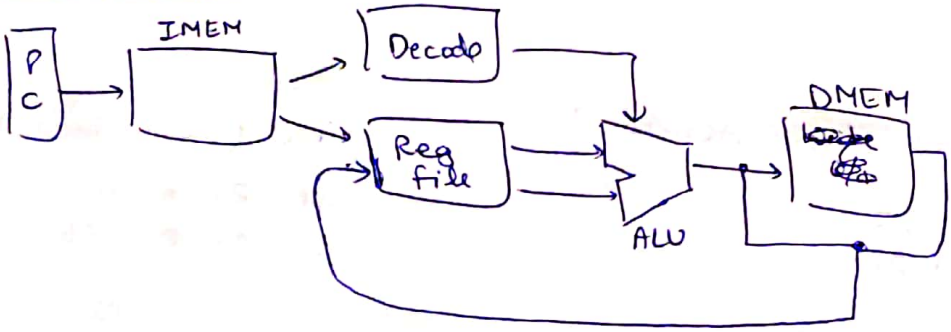
* If Speedup of 90% of time is ∞ then too speedup → 1

AMDAHL's LAW → Do not mess the uncommon case too badly.

Classic 5 stage pipeline



Reg file → fastest Memory



13/1/20

f	D	A	M	W
	ADD R2,R1,1	LW R1,w		
		X	LW. R1,w	
			X	LW R1,w

If command ADD R2,R1,1 was moved to next step A, then it would be working on wrong data as LW R1,w haven't written value.

Assumption:
→ We can write & read in the same clock cycle 

So we get stalls (x). Earlier it would take n cycles for n instructions (1 bubbles)
but now it takes n+Sn cycles where Sn → # of stalls.

F	D	A	M	W
Jump				
~~Sub1~~	Jump			
~~Sub2~~	~~Sub1~~	Jump		

↳ flush the pipeline
Control dependency.

Q 5 stage processor, branch gets resolved in the 3rd stage.

* 20% of instructions are branch instructions.
* Slightly more than 50% of these branches are taken

$$CPI = CPU_{stall} + \underbrace{0.2 \times \frac{1}{2} \times 2}_{\text{depth of pipeline}} \rightarrow 1.2$$

$\rightarrow$ # of flushed instructions

Q 25% of all instr are branch / Jump.

10-stage pipeline

Correct target for branch / jump gets computed in the 6th step

$$CPI_{avg} \rightarrow 1 + \frac{1}{4} \times 5 = \underline{2.25}$$ # all are taken

RAW (True / flow dependencies)

```
Add  R1, R2, R3
Sub  R7, R1, R8
```

WAR
WAW (name / false dependies)

```
Add  R1, R2, R3
MUL  R1, R5, R6
Sub  R7, R1, R8
MUL  R1, R5, R6
```

Q

```
I1  MUL  R1, R2, R3
I2  ADD  R4, R4, R1
I3  MUL  R1, R5, R6
I4  Sub  R4, R4, R1
```

I1 → I2 RAW I2 → I3 WAW I1 → I4 RAW (Broken by I3)

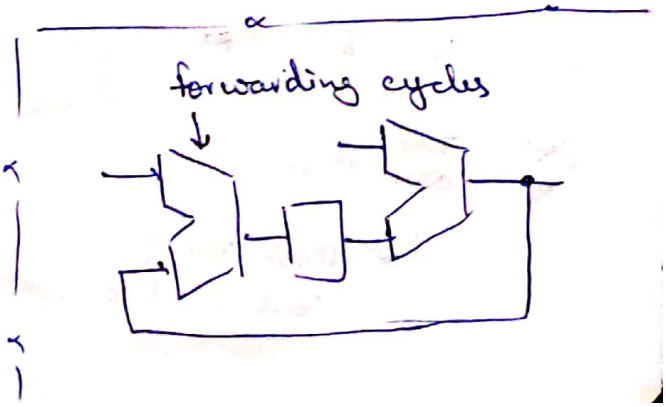
I2 → I3 WAR

15/1/20

Dependencies And Hazard

↓ Properties of the program and oblivious of the processor details

↳ properties of both the program and also depends on the pipeline



I_1	ADD	R1, R2, R3
I_2	MUL	R7, R4, R5
I_3	SUB	R4, R6, R7
I_4	DIV	R10, R4, R8
I_5	XOR	R11, R8, R7

Hazards?

→ true dependency (RAW)

--→ false dependency

* for 5 stage pipeline ⇒ IF → ID → E → MEM → WB
* Among the two dependencies ⇒ $I_1 \to I_5$ & $I_2 \to I_5$ only one of them is a hazard ($I_2 \to I_5$) as for the other one we have 3 instructions between them (This is okay due to the 5-stage pipeline)

2. I_1 ADD R1, R2, R3
I_2 SUB R5, R1, R4
I_3 DIV R6, R1, R7
I_4 MUL R7, R1, R8

3-stage pipeline:

Fetch & Decode	Read Reg & ALU	Mem Read & Write

⇒ 3 dependencies: $(I_1 \to I_2)$ $(I_1 \to I_3)$ $(I_1 \to I_4)$

Dependence	Hazard
$I_1 \to I_2$	Yes (RAW) → Not a hazard if same cycle read & write is allowed.
$I_1 \to I_3$	No
$I_1 \to I_4$	No

I_1	I_1	I_1
	I_2	I_2 I_3
Rough	I_3	I_3 I_3
	-	- -

Handling Hazards:

* Control ~~hazards~~ dependency (Branches): flush the pipeline
* Data dependency → stall (introducing bubbles in the pipeline)
  ↓
  forwarding

⇒ Example (Control dependency)

```
BEQ   R2, R2, label
ADD   - -
SUB   - -

label: {  - -
          - -
          - -
       }
```

⇒

SUB	ADD	BEQ

In pipeline

↓ If Branch is taken

~~SUB~~	~~ADD~~	BEQ

→ flush the pipeline

→ Data dependencies:

```
ADD  R2, R2, R3
SUB  R4, R2, R5
```

Read	ALU
	ADD
SUB	

↳ SUB instr reads old value.

* Stall cycles can be avoided with suitable forwarding techniques (valid for certain cases)

Example:

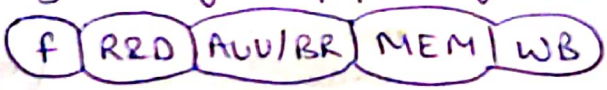
			Flush	Stall	forward

Control dependency	1)	ADD R4, R5, R6
		BNE R1, R4, label
		SUB R5, R4, R3
			✓	x	x

	2)	MUL R1, R2, R3 LW R2, 0(R1)	x	x	✓
	3)	ADD R2, R2, R1	x	✓	✓

* Stall & forward have no place in control dependency.

→ The organization of the pipeline follows:



Keywords:
- out of order execution
- Instruction level parallelism (defined w.r.t to ideal computer)

Sol: Main dependencies are indicated by:

2) 3)

F	RED	ALU/BR	MEM	WB
		MUL		
	LW	↓ LW		

(forwarding takes place in this stage

* forwarding is always a feedback loop in the processor data path.

ILP: ~~Intro~~ Instruction level parallelism

• Briefly → Tells exactly how many instructions are executed in one cycle

→ $R_1 = R2 + R3$
↳ $R4 = R1 - R5$
$R6 = R7 \oplus R8$
$R5 = R8 \times R9$
$R4 = R8 + R9$

Cycle 1	Cycle 2
✓	
	✓
✓	
✓	
✓	

} Because of the dependency these two instructions cannot execute in the same clock cycle

Ideal world
$CPI = \frac{S}{\infty} \to 0$
↙
# of instructions

In real life → there are dependencies which leads to an increase in CPI

for a classic pipeline ⇒ $\lim_{n\to\infty} \frac{d+n-1}{n} \to 1$

Q 5-stage pipeline. Allows forwarding. If no dependencies, 10 instructions in each stage.

		Ex	WB
MUL	R2, R2, R2	2	4
ADD	R2, R2, R2	3	5
MUL	R3, R3, R3	2	4
ADD	R2, R2, R3	4	6
MUL	R4, R4, R4	2	4
ADD	R2, R2, R4	5	7

* Physical register ⇒ P0, P1 ... (actual location of the register value)

Q
```
ADD  R1, R2, R3
SUB  R4, R1, R5
XOR  R6, R7, R8
MUL  R5, R8, R9
ADD  R4, R8, R9
```
⇓
```
ADD  P17, P2, P3
SUB  P18, P17, P5
XOR  P19, P7, P8
MUL  P20, P8, P9
ADD  P21, P8, P9
```

$R_0 \to P_0$
$R_1 \to P_1 \to P_{17}$
$R_2 \to P_2$
$R_3 \to P_3$
$R_4 \to P_4 \to P_{18} \to P_{21}$
$R_5 \to P_5 \to P_{20}$
$R_6 \to P_6 \to P_{19}$
$R_7 \to P_7$
$R_8 \to P_8$

* Inside RAT table, we write the physical address of the corresponding architectural register.

* On write, rename the register

⇒ Even after renaming, the RAW dependency remains (true dependency)
WAR & WAW dependencies are removed
(also called false dependencies or name dependencies)

16/1/20

```
MUL  R2, R2, R2
ADD  R1, R1, R2
MUL  R2, R4, R4
ADD  R3, R3, R2
MUL  R2, R6, R6
ADD  R5, R5, R2
```

→ true dependencies (RAW)
⇢ false dependencies (WAW, WAR)

RAT

R1	P1
R2	P2
R3	P3
R4	P4
R5	P5
R6	P6

```
MUL  P7, P2, P2
ADD  P8, P1, P7
MUL  P9, P4, P4
ADD  P10, P3, P9
MUL  P11, P6, P6
ADD  P12, P5, P11
```

false dependencies are gone.
Register renaming can only remove false dependencies
These can be done parallelly in 2 cycles. & so CPI = $\frac{2}{6} \Rightarrow \frac{1}{3}$

Instruction level Parallelism (ILP)

→ **On Ideal Processor** : Processor can execute all instructions in one cycle.
Processor can execute any number of instructions in 1 cycle.
Processor has to obey ~~two~~ true dependencies.

* ILP holds for only ideal processor. IPC → Instructions per cycle.
(Property of the program & not the processor)

```
ADD  P10, P2, P3
XOR  P6, P7, P8
MUL  P5, P8, P9
ADD  P4, P8, P9
SUB  P11, P10, P5
```

* Ignore false dependencies
* 1 true dependency.
* ILP ⇒ $\frac{5}{2}$ ⇒ 2.5

```
ADD  R1, R1, R1
ADD  R2, R2, R1
ADD  R3, R2, R1
ADD  R6, R7, R8
ADD  R8, R3, R7
ADD  R1, R1, R1
ADD  R1, R7, R7
```

```
ADD  P9, P1, P1
ADD  P10, P2, P9
ADD  P11, P10, P9
ADD  P12, P7, P8
ADD  P13, P11, P7
ADD  P14, P9, P9
ADD  P15, P7, P7
```

ILP ⇒ $\frac{7}{4}$

ILP for control & structural dependencies

- No. structural dependencies (no limit to hardware)
- Assume there is an ideal branch predictor (correct prediction everytime)

```
Add  R1, R2, R3        Label: MUL R5, R7, R8
Mul  R1, R1, R9
BNE  R5, R1, label
ADD  R5, R1, R2
```

* We are assuming perfect BPU - which means even before branch is executed in the fetch cycle itself we know the outcome of the branch
→ Branch itself maybe delayed (due to true dependencies) but is not causing any delay in the following instructions.

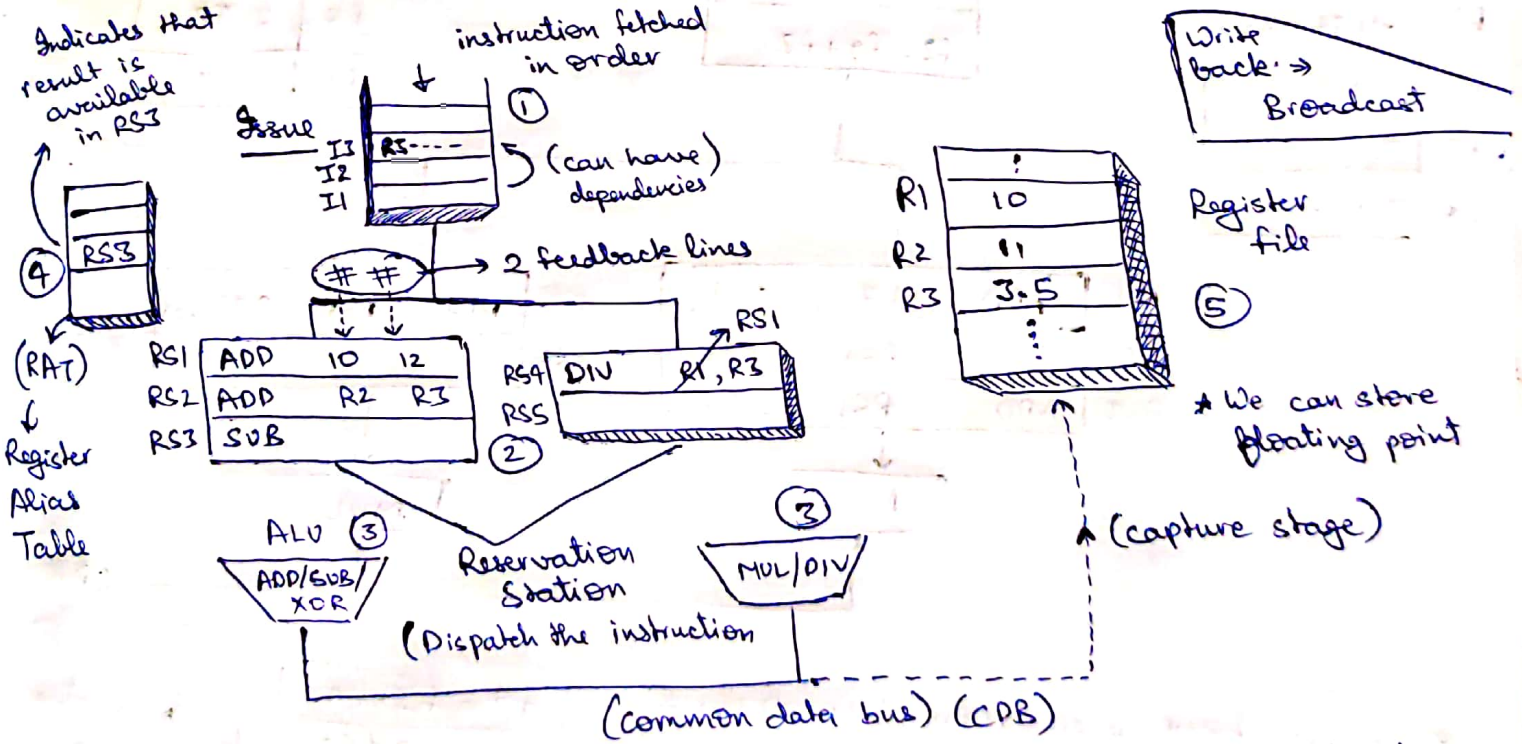
```
ADD R1, R2, R3      P10, P2, P3
SUB R4, R1, R5      P11, P10, P5
XOR R6, R7, R8      P12, P7, P8
MUL R5, R8, R9      P13, P8, P9
ADD R4, R8, R9      P14, P8, P9
```

ILP → 2.5
IPC → 2.5
(1 MUL, 2 ADD/SUB/XOR)

If (1 MUL, 1 ADD/SUB/XOR)
IPC = 1.25

Jan 20

Tomasulo's Algorithm



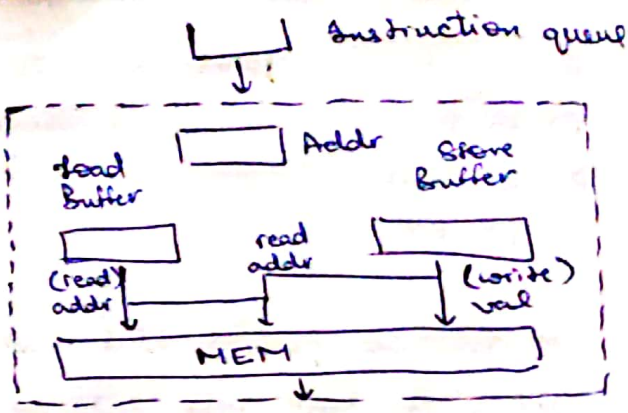
* for the instructions we have two operands, hence two feedback lines to update both.
* for a value in ~~reg~~ reservation station we check if the register is ~~this~~ under some operation (stored in RAT). If yes then take that alias otherwise take value from register file.

- **Load / store unit:**

★ In the classical Tomasulo algorithm, the load/store is done in order

★ Consider only one piece of memory.

★ load/store instructions dependency is a bit tricky.



Issue:

i) Take next instruction in program order from instruction queue.

ii) Determine where the inputs will come from.

iii) Get free reservation stations and of correct kind. [ADD for ADD/SUB]

iv) Put instruction in reservation station.

v) Tag destination to register ⇒ RS4 | DIV R3, R1, R3 | R3 → R1/R3

R3 | RS4 | (RAT)

Ex

(RAT)

f1	→ RS4 → RS3
f2	→ RS1
f3	→
f4	→ RS2

Instruction Queue

f1 = f2 + f3
f4 = f1 - f2
f1 = f2 / f3
f2 = f4 + f2

Register file

f1	3.14
f2	-1.00
f3	2.72
f4	0.71

RS1	ADD	0.71 (f4)	3.14 (f1)
RS2	SUB	RS4	RS1
RS3	ADD	RS1	2.72 (f3)

RS4	DIV	RS1	2.72 (f3)
RS5			

ADD

MUL

CDB

★ If we have two more instructions ⇒

f4 = f3 × f4
f4 = f2 / f2

↓ Instruction Queue

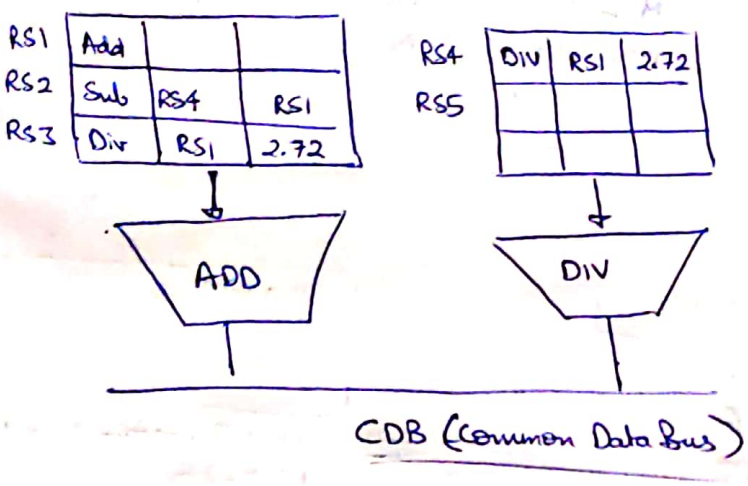
→ MUL/DIV Reservation station has space for just two instructions & already had one instruction.

→ As there is no space for last MUL instruction → it has to wait for any instruction to complete in MUL/DIV Reservation Station.

Jan 22

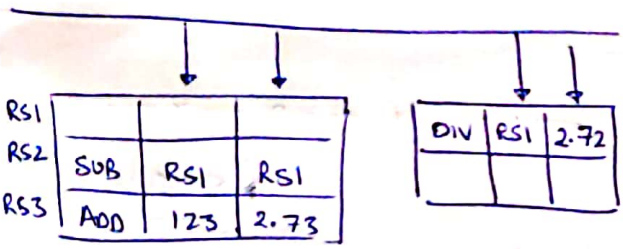
DISPATCH → Put the instruction into the execution unit from Register Station (RS)

CDB (RS1) - 0.29



* At the beginning of the cycle we latch/update the broadcasted result. At the end of the cycle we decide which instructions are ready to go (be dispatched)

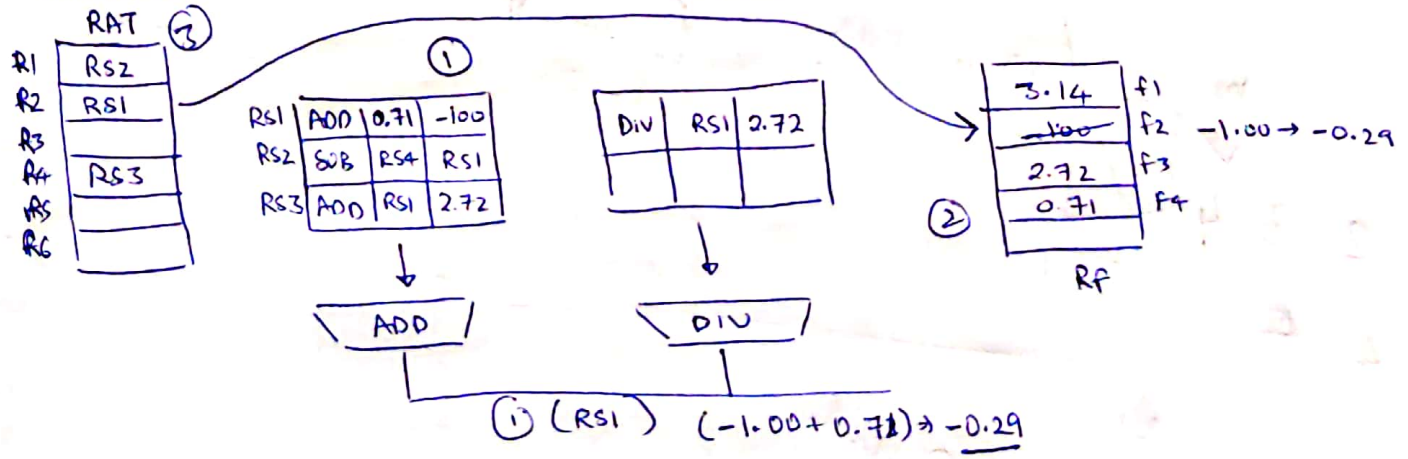
(RS1) - 0.29



Q Why has RS3 not dispatched before this cycle?

1. ~~It was issued in the previous cycle~~
2. ~~Another instruction was dispatched to add unit~~
3. RS2 is older than RS3, so RS3 cannot dispatch until RS2 does.

Write Result - Broadcast onto CDB, RAT, RF



1. Put TAG R result on CDB
2. Write to RF
3. Update RAT (empty the element in RAT do point to the Rf)
4. free RS (invalidate a valid bit)

update the RS4 parameter in the RS. If the entries in RAT do not match the tag RS4, dont update the RAT & RF

Load → 2 cycle
Add → 2 cycle
Mult → 10 cycle
Divide → 40 cycle

R2 → 100
R3 → 200
F4 → 25

	Busy	Op	V_j	V_k	Q_j	Q_k	Disp	Tomasulo (optional)
~~LD1~~	~~✓ (1)~~	~~Load~~		~~134~~			~~✓ (2)~~	(4)
~~LD2~~	~~✓ (2)~~	~~Load~~		245			✓ (4)	(6)
~~AD1~~	~~✓ (4)~~	~~SUB~~	~~-2.5~~	~~7.1~~	~~LD2~~	~~LD1~~	✓ (7)	(9)
~~AD2~~	~~✓ (6)~~	~~ADD~~	~~-9.6~~	~~2.5~~	~~AD1~~	~~LD2~~	✓ (10)	(12)
ML1	✓ (3)	MUL	-2.5	2.5	LD2		✓ (7)	(17)
ML2	✓ (5)	DIV	-6.25	7.1	ML1		✓ (18)	(58)

values for 1st & 2nd operands

1st & 2nd operands for which the instr is waiting.

f0	f2	f4	f6	f8	f10
~~ML1~~	~~LD2~~		~~LD1~~ ~~AD2~~	~~AD1~~	ML2

RAT

	Issue	Execute	Write
LD f6, 34(R2)	1	2-3	4
LD f2, 45(R3)	2	4-5	6
MUL f0, f2, f4	3	7-16	17
SUB f8, f2, f6	4	7-8	9
DIV f10, f0, f6	5	18-57	58
ADD f6, f8, f2	6	10-11	12

Capture → dispatch not in same cycle

Issue: Dispatch not in same clock cycle.

Write back happened after execution cycle ends

Load does not dispatch in same cycle

R2 → 100
R3 → 200
f4 → 25

later
f6 → 7.1 → AD2 → -12.1
f2 → -2.5
f8 → -9.6
f0 → -6.25

Rough

23 Jan Tutorial

gem5 simulator tutorial

3 Feb

Branch Prediction Unit (BPU)

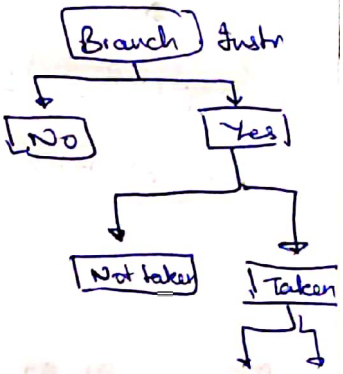
BEQ R1, R2, label (Branch if equal)

* Refuse to predict vs predict → choose predict
* when we need to predict → done at fetch stage.

* **Branch Prediction works on the next value of PC**

— 011011100011 — — —

- Is it a branch?
- Is it taken?
- If taken then what is target PC.



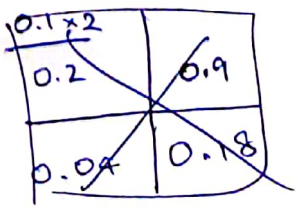
Effect of misprediction on CPI

$$CPI = CPI_{ideal} + \underbrace{\frac{\text{Mispredictions}}{\text{instruction}}}_{\text{Accuracy of prediction}} \times \underbrace{\frac{\text{Penalty}}{\text{Mispredictions}}}_{\text{Pipeline stage when branch is resolved.}}$$

cycles on average per instruction that we add because of mispredictions.

	Resolve branch in 3rd stage	Resolve branch in 10th stage
50% accuracy for branch, 100% accuracy for all other instr	1+0.5×0.2×2 = 1.2	1+0.5×0.2×9 = 1.9
90% accuracy for branch, 100% accuracy for all other instr	1+0.1×0.2×2 = 1.04	1+0.1×0.2×9 = 1.18
Speedup	$\frac{1.2}{1.04} \Rightarrow 1.15$	$\frac{1.9}{1.18} \Rightarrow 1.6$

20% of all instructions are branch.



0.2×0.1

0.2×0.1×9

Q3

5 stage pipeline

Branch is resolved at 3rd stage

fetch nothing until sure of what to fetch

: refuse to predict

Execute many iterations of:

LOOP	ADDI	R1, R1, -1	2 clock cycle
	ADD	R2, R2, R2	2 clock cycle
	BNEZ	R1, LOOP	3 clock cycle (without prediction)

⇒ 7 clock cycle for one LOOP without prediction.

•

Simple Prediction Logic

→ not taken prediction

Refuse to predict	not taken pred.
on branch instr penalty is 3	on branch instr penalty is 1 or 3
for other instr 2 cycle are needed	for other instr 1 cycle needed to fetch

```
BNE   R1, R2, label A
BNE   R1, R3, label B
  A:
  B:
  C:
label A:
  X:
  Y:
label B:
  Z
```

using not-taken prediction compute no of cycles which are wasted on misprediction until we get to Y?

- feb 5

Q 20% instr → branches 60% branches are taken

$CPI_{ideal} = 1$ so CPI now?

⇒ Not-taken Prediction → penalty → 2

$CPI \Rightarrow 1 + 0.2 \times 0.6 \times 2 \Rightarrow 1.24$

Q

	Not taken prediction	Better prediction 99%	Speedup
5 stage (3rd stage)	1 + 0.12×2 ⇒ 1.24	1 + 0.01×2 ⇒ 1.02	1.22
14 stage (11th stage)	1 + 0.12×10 ⇒ 2.2	1 + 0.01×10 ⇒ 1.1	2
14 stage (11th stage) (4 instr/cycle)	0.25 + 0.12×10 ⇒ 1.4	0.25 + 0.01×10 ⇒ 0.35	4.14

~~% Branch instr 20%~~

Not taken prediction

88% (accuracy)

↓

{20% instr * 60% taken}

↓

12% mispredict

↓

88% accuracy

Q Program has 20% of br inst

10% of branches are mispredicted

$CPI_1 = 0.5$ Penalty → 30 cycles

& 2% of branches are mispredicted

⇒ $CPI_1 = CPI_I + 0.2 \times 0.01 \times 30 \Rightarrow 0.5 = CPI_I + 0.06$

$CPI_2 = CPI_I + 0.2 \times 0.02 \times 30$ $CPI_I = 0.5 - 0.06$

$= 0.5 - 0.06 + 0.12$

⇒ 0.56

Reasons for B Prediction accuracy:

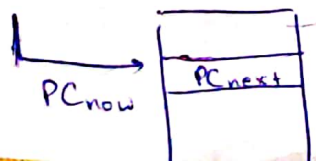
① Improve CPI

② Reduce wasted instr/cycle

$PC_{next} = f(PC_{now})$

→ Assumption: Branches tend to behave the same over & over

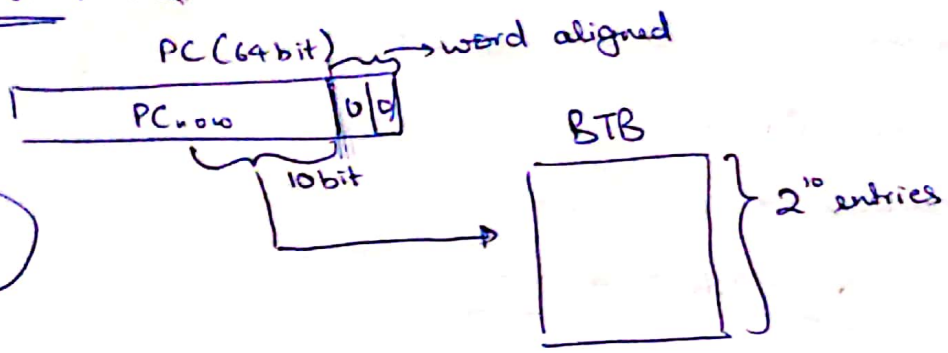
→ History [PC_{now}] can help



(BTB) Branch Target Buffer

→ we can make BTB very big but that won't help as then the access time will be large. Also we want no aliasing.

Solution ?



Q BTB has 2^{10} entries & size 4 byte instructions (word aligned)
- Instructions would begin at addresses divisible by 4
- 32 bit addresses
- which BTB entry is used for PC = 0x0000ABOC

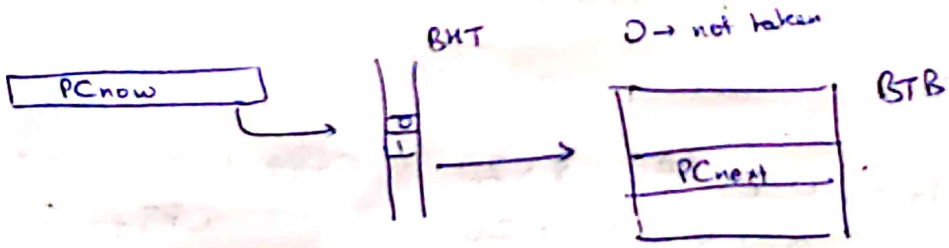
⇒ PC = 0x0000ABOC

→ 0x 0000 0000 0000 0000 1010 1011 0000 1100

→ 1011000011 → 707

Direction Predictors

new table → BHT → Branch history table



Q

```
0xC000   MOV   R2,100
0xC004   MOV   R1, 0
loop: 0xC008   BEQ   R1, R2, Done
0xC00C   ADD   R4, R3, R1
0xC010   LW    R4, 0(R4)
0xC014   ADD   R5, R5, R4
0xC018   ADD   R1, R1, 1
0xC01C   B     loop
```

Assume perfect predicter

BHT → 16 entries

BTB → 4 entries

	# BTB accesses	BTB row
0xC008	1	2
0xC01C	100	3
All rest	0	—

# BHT accesses	BHT row
1	1
1	2
101	3
100	4
100	5
100	6
100	7
100	8

* If we had non-perfect predictor then we would have BHT entries for the instructions after the loop too.

Lets say we have BHT with 1-bit prediction.
i.e. predict with the outcome in the previous access

→ # Misprediction

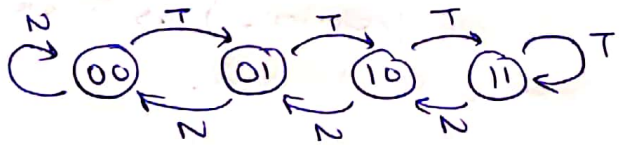
```
{
0x C008   LOOP   BEQ   R1,R2, Done
0x C01C          B     LOOP
}
```

1 (Initially 0. Misprediction at end)
1 ↘ (Initially mispredict)

→ this might have just 2 misprediction in total for a loop
→ But there can be some cases where we have short loops like NNNTTNNN --- . There we will have many mispredictions.

Solution ? we 2-bit predictor (Prediction with history)

(00) strong not taken
(01) weak not taken
(10) weak taken
(11) strong taken



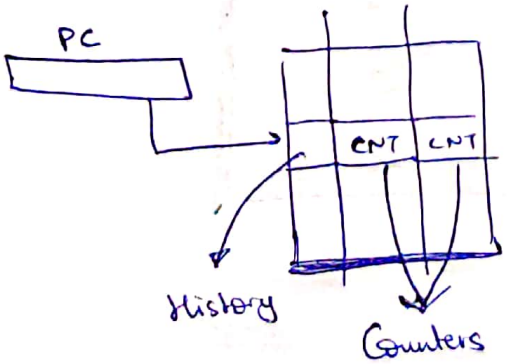
Transitions

* Saturation Prediction (2BC → 2 Bit Counter) 2 bit predictors.

—feb6

1-bit predictors → 2-bit predictors (SPEC'89 benchmark ⇒ 88% to 92% accuracy.

→ 1 bit history with 2BC per history



STATE	PREDICT	OUTCOME	CORRECT?
(0, SN, SN)	N	T	x
(1, WN, SN)	N	N	✓
(0, WN, SN)	N	T	x
(1, WT, SN)	N	N	✓
(0, WT, SN)	T	T	✓
(1, ST, SN)	N	N	✓
(0, ST, SN)	T	T	✓

use steps as: → update
* shift
* predict (USP)

SN → Strong N WN → Weak N
ST → strong T WT → Weak T

→ STATE → (Index, Predict$_0$, Predict$_1$)

→ UPDATE → on current index updak as based on state transition:

Q for sequence → (NNT)*. We operate it. for N repetitions of sequence what are the no. of mispredictions.

→ Total no. of instructions is N×3

Each cycle faces 1 misprediction on T so:

Total mispredictions → N

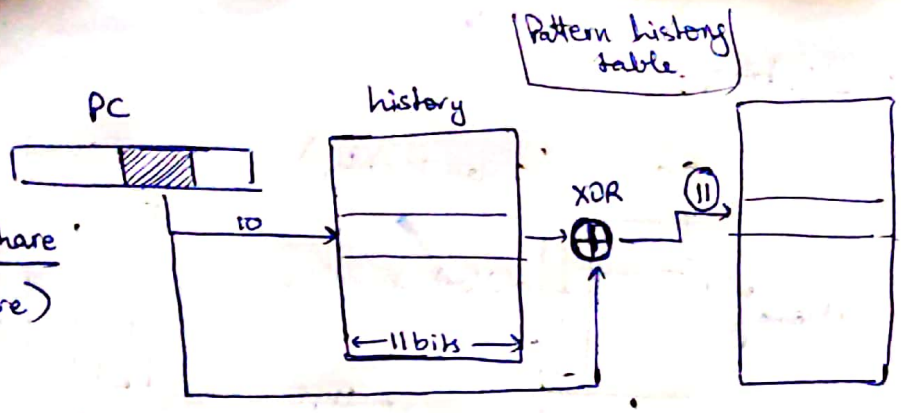
* In general, N-bit history predictors can learn all patterns of length ≤ N+1. BUT we might not use some counters ever as for (NNT)* we never use (counter$_3$) as TT state never occurs. So the resources are not well utilized.

N-bit history | 2BC per history

Need 1024 entries for each branch to have an entry

N =	cost (bits)	(NNNT)* learned?	# of 2BCs used for (NT)*
1	5×1024	x	Both (no wastage) 2
4	(4+2^2×2)×2^{10}	✓	2
8	(8+2^8×2)×2^{10}	✓	2
16	(16+2^{16}×2)×2^{10}	✓	2

10 feb



Size here ⇒ $(2^{10} \times 11 + 2^{11} \times 2) \Rightarrow 31$ Kbits

→ for same size of history in previous method, it would take some mbits memory.

* update the PHT (Pattern history table) in a non-speculative ~~master~~ manner.

* A[i] $\in_R$ [0, 99]

```
if (A[i] < 50)
        do task1()
else if (A[i] > 50)
        do task2()
```

(correlated branches)

second else if condition is benefitted by knowing the history of first if. So here global share helps (G-share)

→ In g-share we cannot use non-speculative way as it will introduce a lot of delay in which the commands might be executed.

→ So in g-share we update in speculative manner.

```
for (i = 1000; i != 0; i--)
        if (i % 2) n = n + i;
```

→ for good accuracy on all branches history should be:
  → PSHARE → 1 bit
  → G-SHARE → 3 bits → order is (011001---)
  ↳ instances to predict

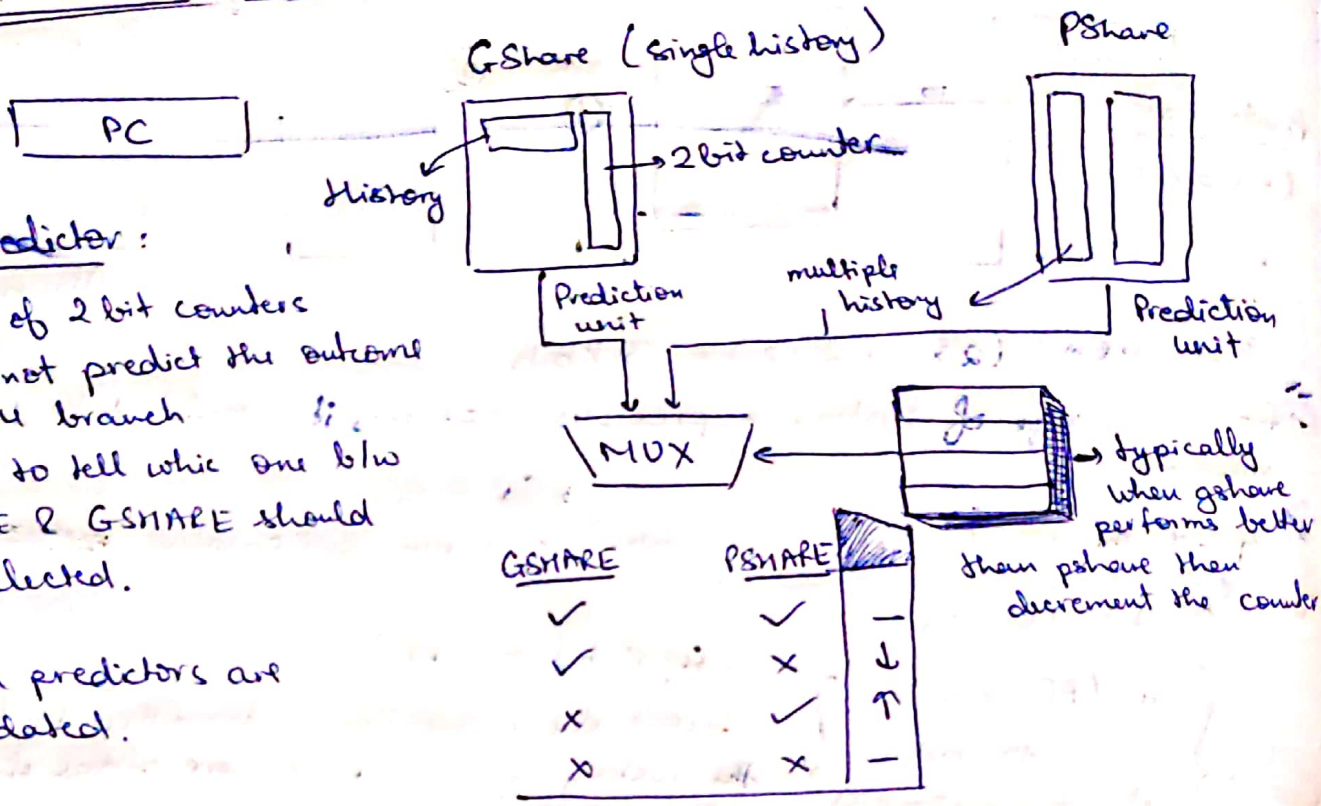
```
Loop
        BEQ     R1, Zero, Exit
        AND     R2, R1, 1
        BEQ     R2, Zero, Even
        ADD     R3, R3, R1
Even
        ADD     R1, R1, -1
        B       Loop.
```

B1 → (1000 ---) T
without no history too it remains same.

B2 → (NT)*
can be predicted with just 1 bit history.

B3 → normal loop
no history needed

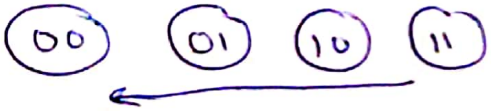
Tournament Predictors:



Meta Predicter:

- array of 2 bit counters
- does not predict the outcome of the branch
- tries to tell whic one b/w PSHARE & GSHARE should be selected.

* Both predictors are updated.



as we go on decrementing it gets to 0. Therefore automatically GSHARE gets selected

Hierarchical Predictors

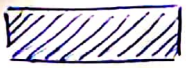
* Present in most modern day processors
→ one good predictor (Good Predictor)
→ one not so good/simple predictor (2 Bit Counter) (OK predicter)

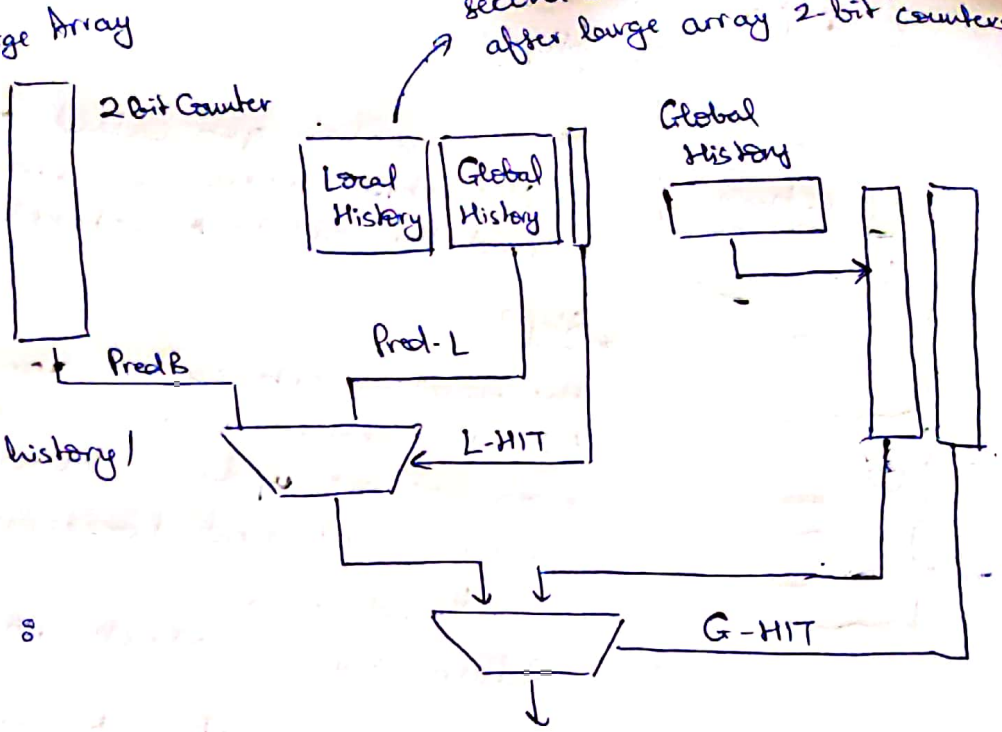
* The update gets reflected on the GSHARE or PSHARE predictor depending upon the one which is being used.

// Update OK predicter on each decision.
Update Good predicter only if the OK predicter fails.

Ex Consider an OK predicter which is a 2-bit counter
- GSHARE (Global history)
- PSHARE (Private history)

→ consider GSHARE the best in this example. Hence it is used at the end if both the remaining predictors fail.





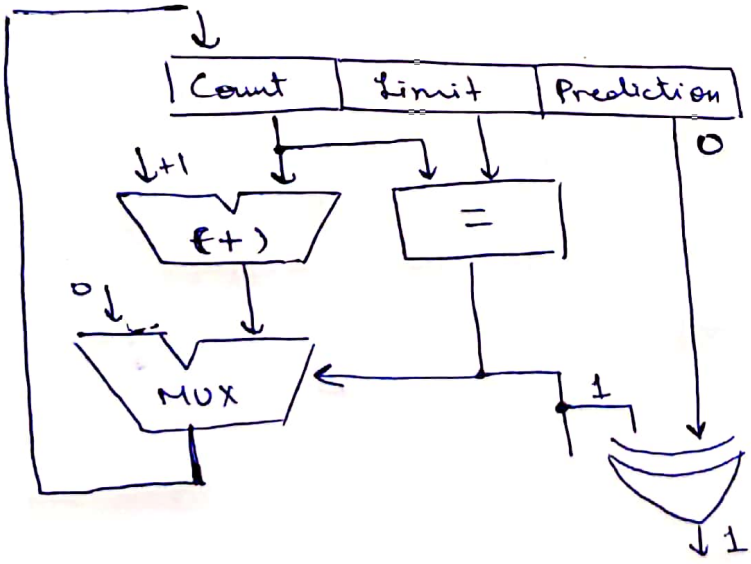
* Based on something similar to TAG in the cache memory there will be a hit/miss of the global history/local history.

Loop Detection

Since there is no restriction on the value of N (no. of iterations) N in a loop can be very high >> size of PHT.

⇒ Although we have the predictors available, it is better to have a dedicated block to handle such loop structures.

→ Say (NNN ---- NT)* is a long sequence.

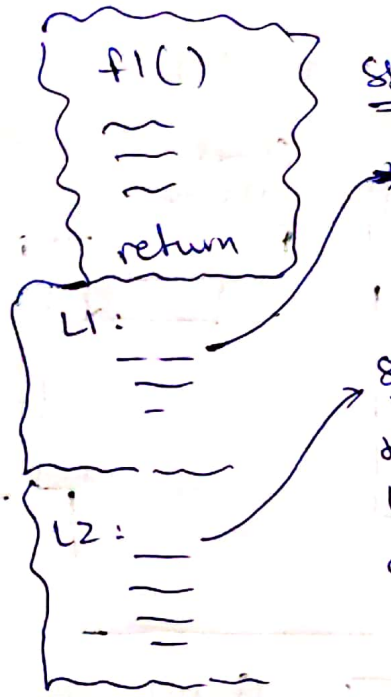


* Direction of the branch
* Target address.

* Consider in a program there is a function f1, which gets called from two locations L1 & L2 in the program

Hardware Dedicated unit called:

- RAS (Return address stack (LIFO)
- Predecoding.



Step1:

→ the function gets called from here, the return address get stored in the BTB and the functions returns correctly.

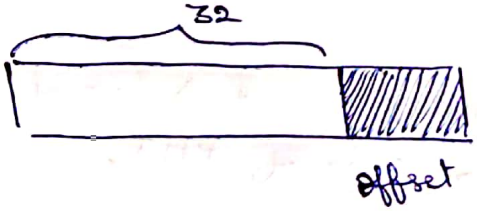
Step2:

the function gets called again from the location L2: the return address stored in the BTB does not get updated, hence it returns to the previous L1 next location.

↓ Solved by

Return Address Stack (RAS) [LIFO]

Q How do you know if it is RET (Return)?



→ predecoding (decides whether PNT)

- Instruction comes from memory to L2-cache to I-cache, maintain an extra bit to precode whether it is RET.