classmate
Date
Page

Introduction :-

software
ISA
H/W

ISA - CISC
RISC

→ Classes of Computers :-
1. Personal mobile
2. ~~Laptops~~ Desktop
3. Servers
4. Cluster of servers

→ Flynn's Classification - 4 types of computers
SISD :- Single Instruction Single Datastream
SIMD :- Single Instruction Multiple Datastream (Data level parallelism)
MISD :- Multiple Instruction Single Datastream → No such machine
MIMD :- Multiple Instruction Multiple Datastream

1) SISD - I_n ..... I_3 I_2 I_1 → CPU

{I_i = Instruction}

One instruction is performed at a time

I ≡ x = (a+b) stored

2) for (i=0; i<10; i++) ← eg Vector processor or GPU or SIMD
z[i] = x[i] + y[i]
z[0] = x[0] + y[0]
⋮
z[10] = x[10] + y[10]
} Executed in single attempt by SIMD but SISD will execute this in 10 times

4) I_n I_{n-1} ..... I_3 I_n I_1
CPU CPU CPU

Increasing performance decreases execution

Improve execution time] — Increasing performance & decrease execution.
or Improve performance time

→ Measuring, Reporting & Summarizing

one task (response time)
multiple task (throughput)

Assume program P is executed on Machine A & B

$$\therefore \frac{\text{Ex. Time}_A}{\text{Ex. Time}_B} = n = \frac{\frac{1}{\text{Performance}_A}}{\frac{1}{\text{Performance}_B}}$$

$$\therefore \text{performance}_B = n \cdot \text{performance}_A$$

Type of program
- Benchmark program - P → used to test machines
- Toy prgms - can't be used to test
- Kernels -
- Artificial - Biased programs

Set of benchmark programs = Benchmark suits.

(SPEC CPU 2006)
- 12 Integers
- 17 floating

exclusively test performance of CPU

Test
SPEC WEB :- Web Server
TPC-A | TPC-B | TPC-D (Transaction Processing council)

Eg →

	P_1	P_2	P_3	Sum of execution time (S.E.T)	(Sum of weighted ex. time) SWET
A	$10 * 1/10$	$8 \times 1/8$	$25 * 1/25$	= 43	3
B	12	9	20	= 41	3.125
C	$8 \times 1/10$	$8 \times 1/8$	$30 \times 1/25$	= 46	3

Reference machine :- We compare B with A & C with A & then compare B with C.

According to SET, $(Perf)_B > (Per)_A > (Per)_C$

But, with SWET

used in SPEC benchmarks

$$\frac{\frac{\text{Exe Time A (reference)}}{\text{Exe Time B}}}{\frac{\text{Exe Time A (reference)}}{\text{Exe Time C}}} = \frac{\text{Ex Time C}}{\text{Ex Time B}} = \frac{3}{3.125} = \text{Eq}$$

Geometric mean Comparison

A	$(10 \times 8 \times 25)^{1/3}$	$= 12.6$
B	$(12 \times 9 \times 20)^{1/3}$	$= 12.93$
C	$(8 \times 8 \times 30)^{1/3}$	$= 12.43$

Prog. 3 is taking much time as compare to 2 & 1. ^ may be due to memory or CPU time & we multiply it by a no.

	A	B	C
Prog 3	25×0.6	20×0.6	30×0.6

The multiplication of 0.6 with prog 3 will have no effect on geometric mean, either it is multiplied by prog 1 or 2
& Remove the const. value from geometric mean

According to geometric mean, $(Per)_C > (Per)_A > (Perf)_B$

SE.T	S.W.E.T	G.M
x	✓	✓
	Require reference m/c	No reference m/c

There is no optimal method to measure performance of machines.

Eg-2

	A	B	C
P_1	1	10	20
P_2	1000	100	20
GM	$(1 \times 1000)^{1/2}$	$(1000)^{1/2}$	$(400)^{1/2}$

$(M/c\ C)_{Perf} > (M/c)_A$

Now we want to execute 100 times P_1 prgm for 1 P_2 prgm

	A	B	C
total time	1100	1100	2020

↑ Failure of GM approach

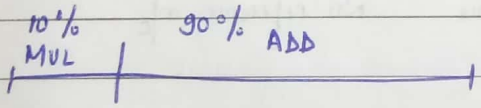
→ Quantative Principles of Computer design :-

Principle 1 - Take advantages of parallelism
- Instructions
- Instruction Level Parallelism
- Hardware

2. Locality of Reference (program tends to use data & instr. they have recently accessed items are likely to be accessed in near future)
- Temporal LOR ← used recently.
- Spatial LOR → whose address are near one another tends to be referenced close together in time

3. Focus on common Case

=



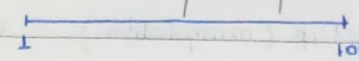
		total time
M/c A	10 multiplier + 1 adder	10+90 = 91
M/c B	1 multiplier + 10 adder	10+9 = 19

$$\frac{(Ex)_A}{(Ex)_B} = \frac{91}{19} = 4.$$

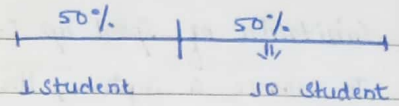
Speedup = $\frac{\text{Perf. of entire task using the enhancement}}{\text{Perf. of entire task without using enhancement}}$

→ **Amdahl's Law** :- performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used.

Case 1 : 100 units



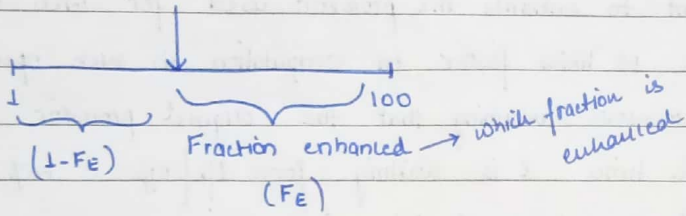
Case 2 : 55 units time



50 + 5 = 55

$\frac{\text{Ex Time}_{old}}{\text{Ex Time}_{New}} = \frac{100}{55} = \frac{(Perf)_{new}}{(Perf)_{old}} = 1.818$

Performance enhancement = 1.818



(S_E) Speedup Enhanced :- Using boosters to speedup. (How we are enhancing)

$(Ex\ Time)_{NEW} = (Ex.\ Time)_{old} \times \left[(1-F_E) + \frac{F_E}{S_E}\right]$

eg $(Ex.\ Time)_{New} = 100\left[(1-0.5) + \frac{0.5}{10}\right] = 55$

→ Speed up ~~(S_E)~~ = $\frac{1}{(1-F_E) + \frac{F_E}{S_E}}$

Speedup enhanced : time of original mode / time of enhanced mode

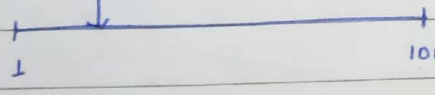
$= \frac{1}{1-0.5+\frac{0.5}{10}} = 55$

Fraction of the computation time in the original machine that can be converted to take advantage of enhancement.

Fraction enhanced ≤ 1

Eg 4

Case 1 :-



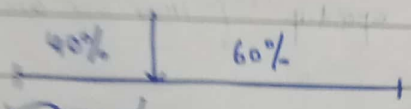
⇒ 90% Additions & 10% Multiplication

10 multipliers & 1 adder = 91 units of time

Case 2 :- 1 multiplier & 10 adder = 19 units of time

(Using principle of focus on common case)

Suppose that we want to enhance the processor used for web operations. The new processor is 10 times faster on computation in web operations than the original processor. Assuming that the original processor is busy with computation for the time & is waiting for I/O for 60% of time what is the overall speedup gained by incorporating enhancement.



$F_E = 0.4$

$S_E = 10$

$$\text{Speedup} = \frac{1}{(1-0.4) + \frac{0.4}{10}} = \frac{1}{1-0.4+0.04} = \frac{100}{64}$$

$$= 1.562$$

Suppose you want to achieve a speedup of 90 with 200 processors what fraction of the original computation can be sequential.

$$\text{Speedup} = \frac{1}{1 - F_E + \frac{F_E}{S_E}}$$

$$90 = \frac{1}{1 - F_E + \frac{F_E}{200}}$$

$$1 - F_E + \frac{F_E}{200} = \frac{1}{90} \Rightarrow F_E = 0.9888$$

$$F_E = 99.38\%$$

Speedup (addition) = $\frac{1}{1-0.9+\frac{0.9}{10}}$

Speed up (multiplication) = $\frac{1}{1-0.9+\frac{0.1}{10}}$

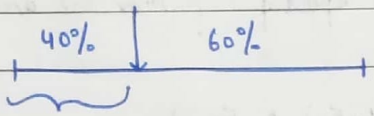
→ Limitation of Speed up :-

To enhance a system, the entire system has to be enhanced.

~~Max (speedup)~~ Ex. Time cannot be less ~~that~~ than $(1-F_E)$

Ques Suppose that we want to enhance the processor used for web operation The new processor is 10 times faster on computation in web operations then the original processor. assuming that the original processor is busy with computation ~~for~~ 40% of the time & is waiting for I/p/o/p 60% of time what is the overall speedup gained by incorporating enhancement.

Soln



$F_E = 0.4$

$S_E = 10$

$$\text{Speedup} = \frac{1}{(1-0.4)+\frac{0.4}{10}} = \frac{1}{1-0.4+0.04} = \frac{100}{64}$$

$$= 1.562$$

Ques Suppose you want to achieve a speedup of 90 with 200 processors what fraction of the original computation can be sequential.

Soln

$$\text{Speedup} = \frac{1}{1-F_E+\frac{F_E}{S_E}}$$

$$90 = \frac{1}{1-F_E+\frac{F_E}{200}}$$

$$1-F_E+\frac{F_E}{200} = \frac{1}{90} \quad \Rightarrow F_E = 0.9938$$

$$F_E = 99.38\%$$

Ques 100 instructions

1. Additions - 25% → 10 (boosters)
2. Multiplication - 30% → 20
3. division - 15% — 5
4. Others

$$S_A = \frac{1}{(1-0.25)+\left(\frac{0.25}{10}\right)} = \frac{1}{0.75+0.025} = \frac{1}{0.775} = 1.290$$

$$S_B = \frac{1}{(1-0.3)+\frac{0.3}{20}} = 1.398$$

$$S_C = \frac{1}{(1-0.15)+\frac{0.15}{5}} = \frac{1}{0.85+0.03} = 1.136$$

total speed up = 2.702

M.2 $$\frac{1}{0.3+\frac{0.25}{10}+\frac{0.30}{20}+\frac{0.15}{5}} = 2.702 = \text{speedup.}$$

Ques If 85% of operations in a parallel prog. must be perform & sequentially. what is the max^m speedup achievable.

$F_E = 0.75$

Sol^n $$\text{Speedup} = \frac{1}{1-0.75+\frac{0.75}{S_E}} = 4$$

$S_E \longrightarrow \infty$

Sequentially - which is not evaluated.

→ Speedup is unitless (Speedup v/s percentage)

	Old M/c	New M/c
progA	100	70

$$\text{Speed up} = \frac{\text{Ex Time Old}}{\text{Ex. Time New}} = \frac{100}{70} = 1.428$$

$$\frac{Perf_{New} - perf_{old}}{Perf_{old}} = \text{percentage change}$$

$$= \frac{1/Ex.\,Time_{New} - 1/Ex.\,Time_{old}}{1/Ex.\,Time_{old}} = \frac{1/70 - 1/100}{1/100} = 0.428$$

$$\%age = 42.8\%$$

→ Processor Performance Equation

$$T \propto \frac{1}{F} \Rightarrow \text{clock cycle time} = \frac{1}{\text{Clock cycle speed}}$$

CPU time :- time require to execute a program

= clock cycles for a program × clock cycle time

↓ IC × CPI

eg CPU time = 500 cycle × 1ns = 100ns

prog (A)

↓

n instructions ⟶ IC (Instruction Count)

(Instruction path length)

CPI = Clock cycles per instruction = $\frac{\text{Clock cycles for a program}}{IC}$

∴ **CPU time = IC × CPI × clock cycle time**

Unit = CPU time

Clock cycle time - Hardware technology & organisation

CPI - Organisation & Instruction Set Architecture

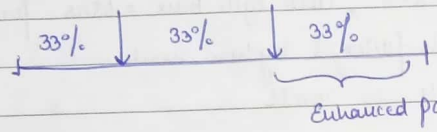
IC - Compiler technology & ISA -

	M/C A	M/C B
Cycle	500	104
Instruction	100	100

∴ $CPI = \frac{500}{100} = 5$ $\quad \frac{104}{100} \simeq 1$

IC : Depend on the user, less no. of instruction leads to good program (CPU time ↓)

CPI : Same Hardware but difference in architecture



Max^m speedup achievable $= \frac{1}{(1-0.33) + \frac{0.33}{\infty}} = \frac{100}{66}$

If all part is enhanced then

Max^m speedup achievable $= \frac{1}{\frac{33}{\infty} + \frac{33}{\infty} + \frac{33}{\infty}} = \infty$

To improve the CPU time, you have to enhance all the factors.

→ Let for any Program (A)

CPU clock cycle $= \sum_{i=1}^{n} IC_i \times CPI_i$

- → type 1 instruction
- → type 2 "
- ⋮
- type n "

CPU time $= \sum_{i=1}^{n} IC_i \times CPI_i \times$ Clock cycle time

Overall CPI $= \frac{\sum_{i=1}^{n} IC_i \times CPI_i}{IC} = \sum_{i=1}^{n} \left[\frac{IC_i}{IC}\right] \times CPI_i$

$\left[\frac{IC_i}{IC}\right]$ gives %age of i nstruction

→ If we are comparing 2 machines by their CPU time, then

$$\text{speedup} = \frac{\text{CPU time old}}{\text{CPU time New}}$$

Ques Suppose you have 2 implementations of the same IS architecture. M/c A has a clock cycle time of ~~15~~ 50ns & CPI of 4.0 for some program X. M/c B → CPI = 2.5, clock cycle time = 65ns for the same program. Which machine is faster & by how much.

Soln

$$\text{CPU time}_A = IC \times 50 \times 4 = 200IC$$

$$\text{CPU time}_B = IC \times 2.5 \times 65 = 162.5\,IC$$

M/c B is faster than M/c A

$$\frac{\text{CPU time}_A}{\text{CPU time}_B} = \frac{200}{162.5} = 1.230$$

~~CPU~~ M/c A is 1.23 time ~~e~~ slower than ~~CPU~~ M/c B

Ques Consider a M/c A for which following performance measures were recorded when executing a set of benchmark programs. Assuming the execution of benchmark programs what is the overall CPI of M/c A.

Instruction type	% of occurence	CPI
ALU	35	1.0
L&S	20	3.0
Branch	40	4.0
Other	5	5.0
	100	

$$CPI = \frac{35}{100} \times 1 + \frac{20}{100} \times 3 + \frac{40}{100} \times 4 + \frac{5}{100} \times 5$$

$$= \frac{35 + 60 + 160 + 25}{100} = \frac{280}{100} = 2.8$$

→ CPU Time :- IC × CPI × CCT

IPC = Instruction per clock

$$\text{Performance} = \text{clock speed} \times \frac{IPC}{IC}$$

$$CCT \propto \frac{1}{\text{Clock speed}_{(GHz)}}$$

Ques My New laptop has an IPC ie 20% worse than my old Laptop. It has a clock speed ie 30% higher than the old one. I am running the same binaries on both machine. What is the speedup of new laptop

Soln

$$\text{Speedup} = \frac{Ex\ Time\ old}{Ex\ Time\ New} = \frac{(Perf)_{New}}{(Perf)_{old}}$$

$$= \frac{1.3 \times \text{clock speed}_{old} \times 0.8 \times IPC_{old} \,/\, IC}{\text{clock speed}_{old} \times IPC_{old} \,/\, IC} = 1.04$$

=

→

IPC	CPI
Equal No. of clock cycles ie all program finishes in equal no. of clock cycles.	~~When all~~ Equal no. of instruction is considered.

Arithmetic Mean of IPCs = $\frac{1}{\text{HM of CPIs}}$

Am of CPIs = $\frac{1}{\text{HM of ICs}}$

Gm of IPCs = $\frac{1}{\text{Gm of CPIs}}$

→ My New laptop has a clock speed = 30% higher than old one. I saw running the same binaries on both PC. their IPCs are listed below.

	P_1	P_2	P_3
Old IPC	1.2	1.6	2.0
New IPC	1.6	1.6	1.6

find speedup of New laptop.

Soln $(Perf)_{New} = \frac{1.3 \times \text{clock speed}_{old} \times IPC_{New}/IC}{\text{clockspeed}_{old} \times IPC_{old}/IC}$

Using Am :-

Am of IPC_{New} = 1.6

Am of IPC_{old} = 1.6

∴ Speedup = 1.3

Using Gm.

Gm of IPC_{New} = 1.67

IPC_{old} = 1.57

Speedup = 1.32

Ques

Soln

Power v/s Energy

power = Dynamic Power (D) + Leakage Power (L)

Dynamic Power $\propto$ Activity $\times$ capacitance $\times V^2 \times f$ → freq.
↓ Voltage

$DP \propto V^2 f$

Activity = No. of transistors switches
when machine is in ideal state, Activity = 0 & DP = 0

Leakage Power $\propto f(TC, I, V) = TX I_c X V$
↓ leakage current

when system is doing nothing but consuming power = leakage power

Energy = power x time

Ques Processor A consumes 1.2 times the power of processor B. B at finishes the task in 30% less time. which processor is better.

Sol^n

	A	B
Power	1.2W	W
time	0.7t	t

$Energy_A = (1.2 \times 0.7)Wt = 0.84 Wt = 0.84 Energy_B$

processor A less energy then processor B

1 GHz = 1 ns

↳ expectation

so we decrease the freq

→ power = DP + LP

DP $\propto V^2 f$

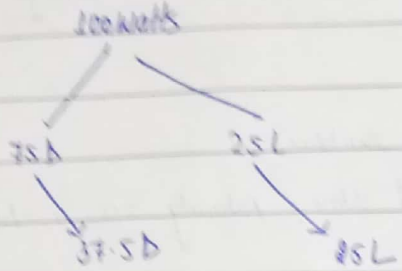
LP $\propto$ I_c WTC (85%)

↓ No of transistor

DFS (Dynamic Frequency Scale)

↳ frequency can change

Process A →



$D \propto V^2 F$

$F \rightarrow F/2$

P_{new} = D + L = 62.5

as freq = half time = double

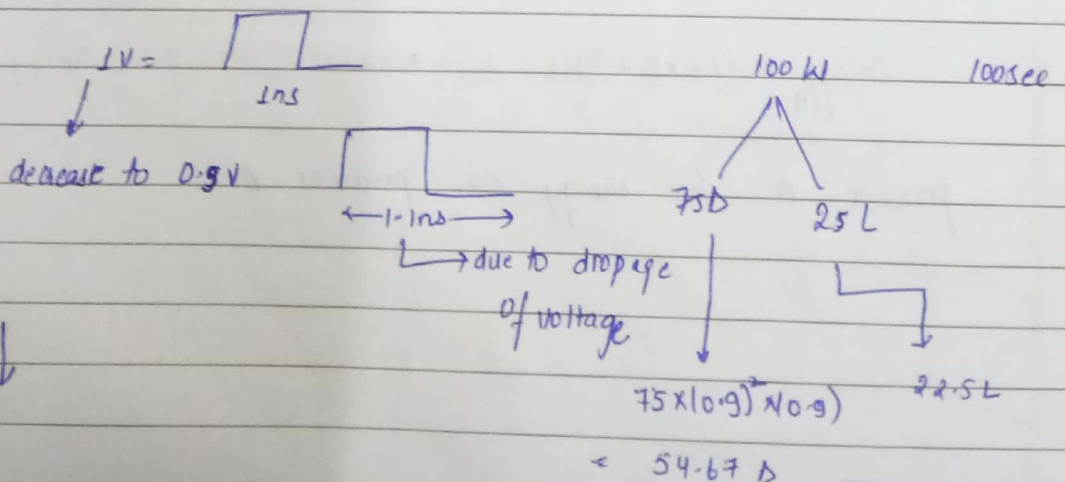
∴ New Energy = 62.5 X 200 = 12500

Old Energy = 100 X 100 = 10000

E_{new} = @ 1.25 E_{old}

Here power is reduced but time increases so Overall Energy ↑

DVFS :- Dynamic Voltage Frequency Scale



Time ↑ ⇒ freq ↓

Scanned by CamScanner

New power = 77.5 New time = 110 ns.

New Energy = 77.5 X 110 = = 8525

old = 10000

$Energy_{New} = 0.85\ Energy_{old}$

Ques For a processor running at 100% utilization at 100 watts. 20% of power is attributed to leakage. What is the total power dissipation when the processor is running at 50% utilization.

Soln $\Delta P \propto$ Activity $\times$ capacitance $\times V^2 \times f$

A ⟶ 100% utilization B ⟶ 50% utilization

100 watts

20% leakage

(Activity = utilization)

$P = \Delta + L$

$= 80 + 20$

↳ 80 watts for 100% utilization

for B, $P = \Delta + L$

$= \frac{80 \times 50}{100} + 20$

$= 60$ Watts

Ques If processor A consumes 1.4 times the power of processor B but finishes the task in 20% less time. which processor would you pick

(a) If you were constraint by power requirements — processor B

(b) If you wue trying to minimize energy — processor B

(c) If you were tying to minimize response time — processor A

Soln

	A	B
Power	1.4W	W
Time	0.8t	t
Energy	1.4 X 0.8 Wt = 1.12 Wt	Wt

Ques Processor A at 1 GHz consumes 80 Watts of dynamic power & 20 Watts of leakage Power. It completes a program in 20 sec.
(2100) a) what is the energy consumption if I scale freq. down by 20%
(1428) b) what is the energy " " " freq. & voltage down by 20%

Soln Initial freq = 1 GHz
Δ Power = 80 Watts
I Power = 20 Watts

a) freq ↓ by 20% freq = 0.8 GHz
I Power = 20 Watts
Δ Power = 0.8 × 80

Power = Δ + I
= 64 + 20 = 84

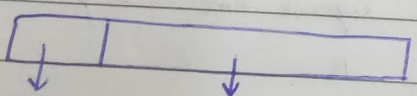
Energy = P × t
= 84 × 20/0.8 = 2100

b) freq & voltage ↓ by 20%
Δ Power =

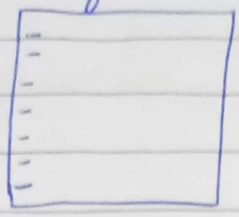
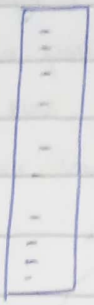
→ Instruction set Architecture

CISC :- Complex Instruction Set Computer

Machine Instruction



RISC : Reduced Instruction Set Computer
- Smaller, Faster

CISC	RISC
1) Small No. of instructions but wider	
 Only 40 instructions required	 100 instructions to execute. - Instruction size - small
2) Length of instruction - variable	- Const length
3) There is no separate load, store instructions.	- Easily fit in pipeline structure.

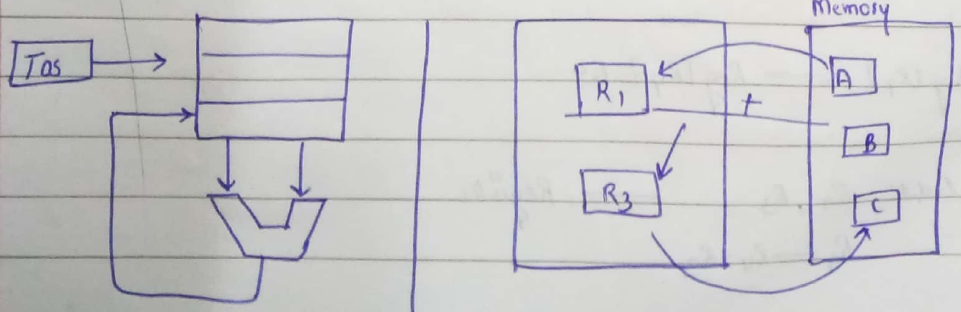
1. How data is stored in a processor
   ↓
   Internal storage (Registers)

Based on internal storage, 4 type of ISA

i) stack
ii) An Accumulator - ALU
iii) Register - memory architecture
iv) Register - Register Architecture

Eg: $C = A + B$, (A, B, C are stored in memory)

Stack	An accumulator	R-M	R-R
Push A	Load A	Load R_1, A	Load R_1, A
Push B	ADD B	Add R_3, R_1, B	Load R_2, B
ADD	Store C	Store R_3, C	Add R_3 R_1 R_2
Pop C			Store R_3 C



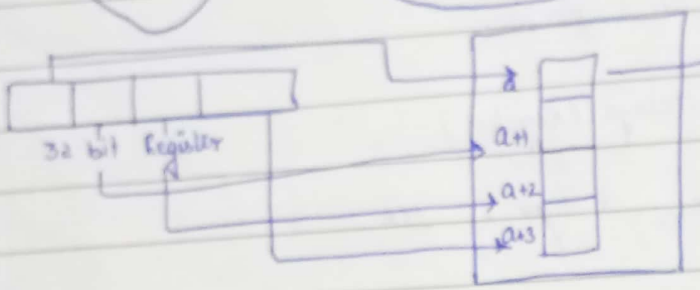
Memory Addressing

1 Byte = 8 bit

Half word = 16 bits

word = 32 bits

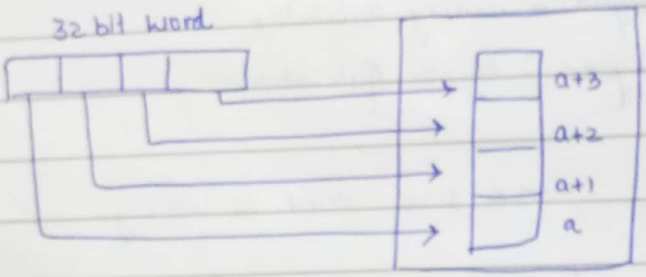
double word = 64 bits



Convert 32 bit word in Hexadecimal & use 8 bit to store in every location

Big Endian

Little Endian



a
a+1 — offset
a+2
a+3

Addressing Modes

Eg Add R_4, #3 → Immediate

$Reg[R_4] \leftarrow Reg[R_4] + 3$

Add R_4, R_3 — Register

$R_4 \leftarrow R_4 + R_3$

Add $R_4, (R_1)$ — Register Indirect

Add $R_4, (1000)$ — Direct

Add $R_4, 100(R_1)$ displacement

($R_4 \leftarrow R_4 + Mem[R_1 + 100]$)

PC- Relative addressing mode :-

Types of Instruction :-

1. Arithmatic & logical Instruction
   - Add, or, AND, SUB
2. Data Transfer
   Load, store
3. System
4. Floating point operations
5. Decimal Operation
6. Strings
7. Graphics
8. Branch

1. Load - 22%
2. Conditional Branch - 20%
3. Compare - 16%
4. Add
5. AND
6. SUB
7. MOVE
8. CALL
9. Return

} 96%

$I_1 \rightarrow PC \rightarrow CPU$ 1word = 32bits = 4byte

I_2 PC = PC + 4

I_3 PC = PC + 4

if (Cond.) then - $I_1 \rightarrow PC$

Statement 1 - I_2 PC = PC + 4

X

can't execute any statement until CPU specify which inst. will execute.

else

Statement 2 - I_3

I_n

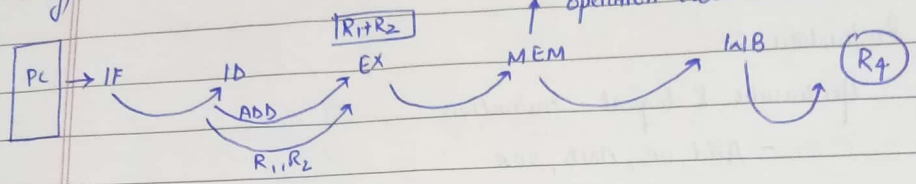
double → ADD = 64 bit operation
ADD = 32 bit operation

Pipelining :- Implementation technique in which there exist overlapping of instructions

1. Instruction Fetch (IF)
2. Instruction Decode (ID)
3. Execution (EX)
4. Memory (Mem)
5. Write Back (WB)

1. Instruction Fetch (IF)

eg) ADD R_4 R_1 R_2 = 5 clock cycle

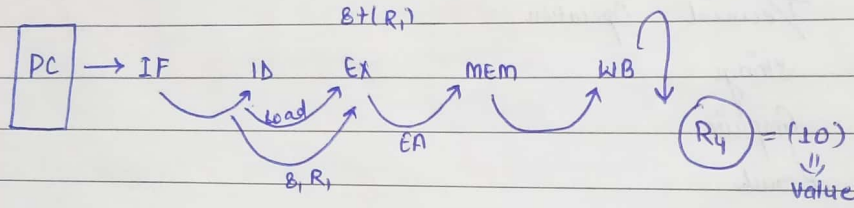


eg) LOAD R_4, B(R_1) = 5 clock cycle

{ $R_4 \leftarrow$ Mem [B + Reg(R_1)] }

B = offset

EA = effective address

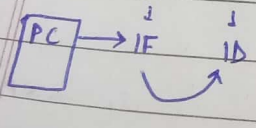


if (B+R_1)^{mem} = value

eg) store B(R_1), R_4 = Only 4 clock cycle

(WB not required, store fn work at Mem)

eg) BNZ R_3, L
(Branch Not zero)

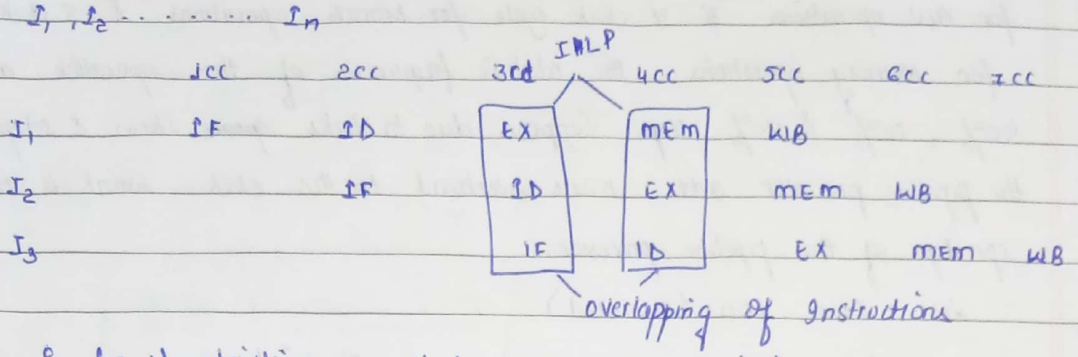


2 clock cycle to detect either it is branch or not

I_1 if (x==0) then
I_2 x = x+10
else
I_3 x = x-10

Let there are N instructions to be executed by pipeline

$I_1, I_2 \ldots\ldots I_n$



∴ for N instructions - N+4 clock cycles required.

Instruction level parallelism - ILP

$$CPI_{UN} = \frac{5N}{N}, \quad CPI_P = \frac{N+4}{N}$$

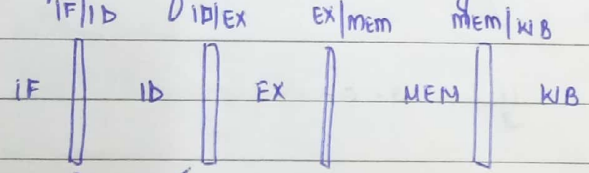
[CPI can never be less < 1]

$(N \to \infty)$ $CPI_{UN} = 5$ $CPI_P = 1$

$CPI_{Ideal} = 1$ (Whatever be instruction)

The objective of ILP is to make CPI = 1 ie for every clock, I have to execute one instruction

Irrespective of architecture, every instruction has to pass 5 clock cycle.



Due to pipeline registers, instruction may take some extra clock cycle

Ques

$CPI_I = 1$ but due to overhead $CPI_I = 1+?$

Ques An unpipeline processor with one clock cycle time = 1ns & it uses 4 clock cycles for ALU operations & 4 clock cycle for branch operations & 5 clock cycle for memory operations. The relative frequencies of this operation are 40%, 20% & 40% resp. Suppose due to clocks ~~queue~~ skew & setup the pipeline processor adds 0.2ns overhead to the clock. What is the speedup of the pipeline processor.

Sol^n Let IC = 100 (assumed)

Unpipelined

CCT = 1ns

ALU = 40%

Branch = 20%

Memory = 40%

pipelined

CCT = 1.2ns

Not given for pipeline

∴ we assume $CPI_p = 1$

$$\frac{CPU_{old}}{CPU_{New}} = \frac{IC \times CPI_{UP} \times CCT_{UP}}{IC \times CPI_{PP} \times CCT_P}$$

$$\text{Overall CPI} \Big/ (CPI)_{UP} = \sum_{i=1}^{n} \frac{IR_i}{IR} \times CPI_i$$

$$= \frac{40}{100} \times 4 + \frac{20}{100} \times 4 + \frac{40}{100} \times 5$$

$$\Rightarrow \left(\frac{160 + 80 + 200}{100}\right) = 4.4$$

$$\text{Speedup} = \frac{4.4 \times 1}{1.2 \times 1} = \frac{11}{3} = 3.67$$

≠ speedup from pipelining = $\frac{IC_{UP} \times CPI_{UP} \times CCT_{UP}}{IC_P \times CPI_P \times CCT_P}$

Ideal pipeline

Same prgm

IC

Unpipeline

IC

CCT = 5

∴ Speedup = $\frac{CPI_{UP}}{CPI_P}$

for N instructions, $CPI_{UP} = \frac{5N}{N} = 5$, CPI_p (ideal) = 1

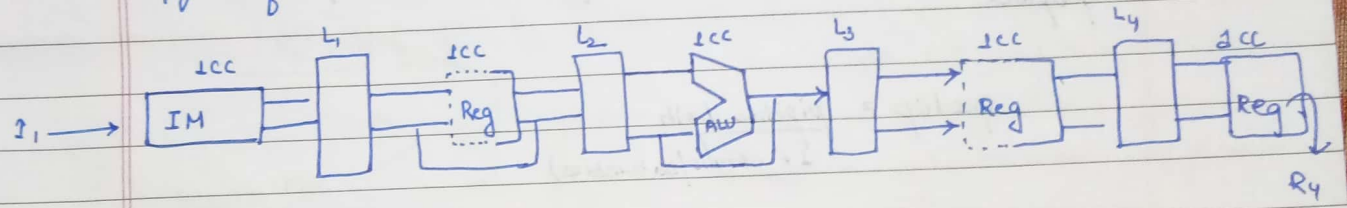
$\therefore$ speedup $= \frac{5}{1} = 5$

If pipeline depth = 6, then speedup = 6

$$\text{Speedup} = \frac{\text{pipeline depth}}{1}$$

If $CPI_p \neq CPI_{ideal}$

→ Hazards of Overhead :-



Types of Hazard -

1. Structural Hazard
2. Data Hazard
3. Control / Branch Hazards

I_1	IF	ID	EX	MEM	WB	
I_2		IF	ID	EX	MEM	WB

Due to some probms / Hazards

I_1	IF	ID	EX	MEM	WB	
I_2		0	0	0	0	0 → Bubbles (No o/p at 6cc)
		IF	ID	EX	MEM	WB

Note • There is only one register which can hold data / cache.

1. Structural Hazard

	1cc	2cc	3cc	4cc	5cc	6cc	7cc	8cc
I_1	IF	ID	EX	MEM	WB			
I_2		IF	ID	EX	MEM	WB		
I_3			IF	ID	EX	MEM	WB	
I_4				IF	ID	EX	MEM	WB

The register which fetches the inst., store the data so same register is used in IF & MEM.

So at time of 4 instruction, the same register can hold data as well as store data of I_1 inst. So I_4 instruction start at 7th cc.

This is structural Hazard.

Structual Hazard arises due to use of same device for more than 1 purpose.

$$\text{speedup} = \frac{\text{Pipeline depth}}{1 + (\text{stalls/bubbles})}$$

Solution of Structual Hazards :- Increment in resources.

2. Data Hazards :-

I_1 ADD R_4, R_1, R_2

I_2 ADD $R_5\ R_4\ R_6$

	1cc	2cc	3cc	4cc	5cc	6cc	7cc	8cc
I_1	IF	ID	EX	MEM	WB			
	Add R_1, R_2		R_1+R_2					
I_2	IF	ID	B	B	ID	EX	MEM	WB

So reading R_4, R_6 Not allowed ∴ bubble

Reading R_4, R_6 (old value of R_4) Not updated

decoder will detect dependency.

at 5cc, data is available

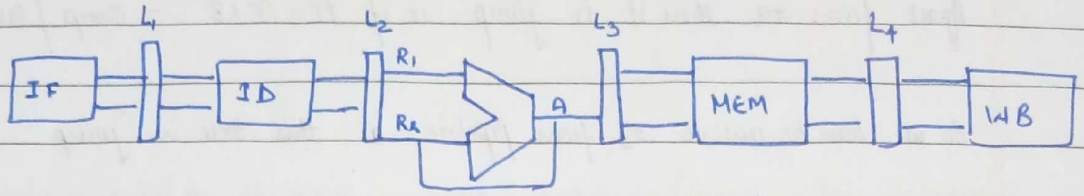
Due to data Hazards, I_2 finishes at 8cc

if there is bubble in pipeline, all will stop, no forward action.

If I_3, I_4 are independent just then at 9 & 10cc, these instruction finishes

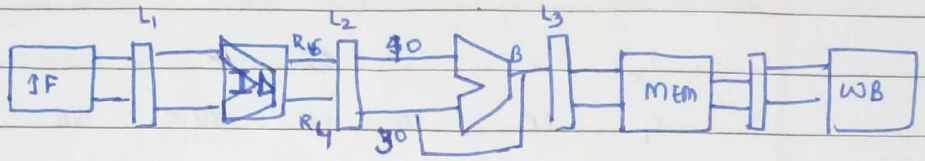
Solution :- By passing or data forwarding

ADD R_4, R_1, R_2

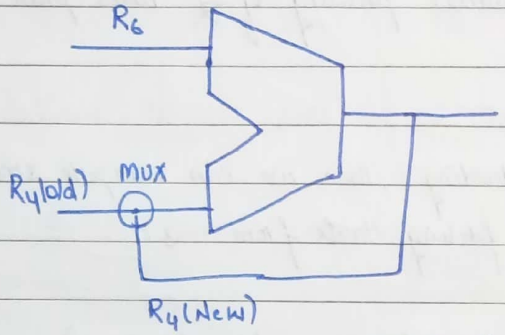


$R_1 = 10$
$R_2 = 10$
$R_4(old) = 30$
$R_6 = 40$

ADD R_5 R_4 R_6



point A & point B are same. so.



↳ as a result of 1st Instruction

So old value is replaced by New value

∴ No bubble in Pipeline

→ Data forwarding

2. Delayed Instruction :- but not very useful

Data forwarding is not applicable for ~~load~~ loading operations.

3. Control Hazards :-

I_1 : if (Condition)

then / else

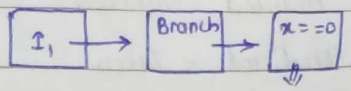
I_2 : $R_4 = R_4 + 7$

I_3 : $R_3 = R_4 + 8$

I_3 → IF

I_2 → IF ID

I_1 → IF ID EX MEM WB



eg. condition : x==0 false

Fig - C.26

I_1 = Branched Inst.

I_2 =

I_3 = Branched Target Inst.

No jump / Untaken :- PC = PC+4

PC+4

+4

!

Apart from +4, when it is jump ie if PC = PC+8 - Jump / Taken

So we have to remove I_2 from pipeline as ~~this~~ then is jump.

I_3		~~IF~~	IF	-	-	- - - 8cc
I_1	IF	ID	EX	MEM	WB	

No o/p at 6 & 7 cc. Outcome of I_1 come at 3cc & fetching of I_3 takes place at 4cc while fetching of ~~I_2~~ takes place at 2cc only.

→ If we add a small ALU at decoding, then we can compute small operations at decode stage & fetching starts from 3 cc.

3 ALU in pipeline :- ~~1.~~ 1. At fetching (for PC = PC+4)
2. At decoding (for small computations)
3. Execution (for Computations)

Q How pipelining is implemented ?

→ Techniques available for control Hazards -

1. Flushing
2. Predicted - Untaken
3. Predicted - Taken
4. Delayed branches

1. Flushing :-

Br: I_1 → IF ID

I_2 IF → flushed down from pipeline

I_3 IF ------ Fcc

2. Predicted - Untaken :- We assume that there is no jump. All instructions are executed ~~to~~ like wise

Jump

Br: I_1 IF ID EX MEM WB

I_2 IF No-op → (No-operation)

IF

both methods waste Icc

3. Predicted - taken :- We assume that there will always be a jump.

Br. I_1 → IF ID EX MEM

I_3 IF ~~th~~ → switching to I_2

I_2 IF

If there is no jump, switch to I_2 & Icc is wasted.

4. Delayed branches :-

Choice 1

Br: I_1 → IF ID

I_x IF ID EX MEM WB

depending upon branching or Not → I_2 | I_3 IF ID EX MEM WB

I_x = those instructions which are independent of branch instruction

I_x

Branch (I_1)

80% → I_2

20% → I_3

Pipeline Speedup = $\frac{\text{pipeline depth}}{1 + \text{overhead / bubbles / stall due to Br.}}$

↓

Branch Frequency

Ques For a deeper pipeline (more than 5 stages), it takes atleast 3cc or 3 stages before the target address is known. & an additional 1cc for evaluating the branch condition. find the effective addition to CPI ideal arising from branches assuming the following frequency

I 1. Unconditional branch - 4%

II 2. Conditional branch Untaken - 6%

III 3. " " taken - 10%

Schemes -	I	II	III	
1. flushing	2	3	3	Penalities
2. predictive taken	2	3	2	
3. Predicted Untaken	2	0	3	

Soln CPI ideal = 1

effective addition = 1 + ?

for flushing, effective addition - $\frac{2\times4 + 3\times6 + 3\times10}{100} = 0.56$

for PT, effective addition = $\frac{2\times4 + 3\times6 + 2\times10}{100} = 0.46$

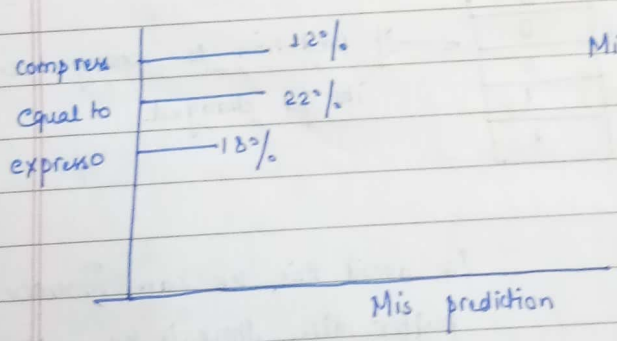
for PNT, effective addition = $\frac{2\times4 + 3\times10}{100} = 0.38$

Prediction Techniques :-

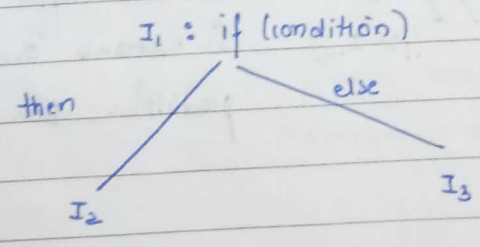
→ Static Branch Prediction technique
- → 1 bit predictor
- → 2 bit predictor
- → Correlating prediction
- → Branch Target Buffer (BTB)
- → Return Address
- → MISC
  - ↳ Value prediction (x = x + 10).

→ Prog.

Before Running program you are scheduling instruction that are to be executed.
(Suppose prog. is analyzed for 10 minutes & I_3, I_5 & I_7) are executed then only these inst will be executed again & again.



Mis prediction rates
= 3-25%



1 bit predictor :-

decided I_1 is taken or not taken

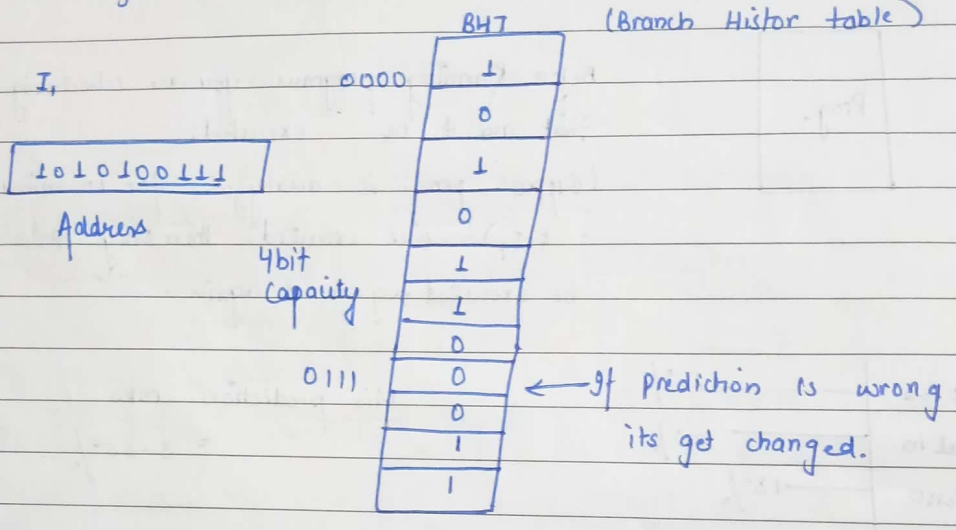
	IF	ID	EX	MEM	WB
I_1	IF	ID	EX	MEM	WB
I_2		IF			

We have to find before ID that I_1 will be taken or not

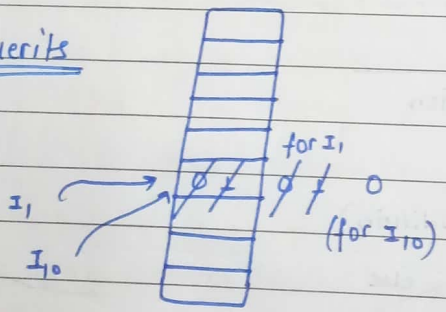
BHT

	IF	ID	EX	MEM	WB
I_1	IF	ID	EX	MEM	WB
I_2		IF			

If correct then it will execute else value in BHT for that address is changed & then next time & I_3 will be executed.



Demerits



To avoid this, we can increase the buffer size. there is no information on what buffer size should be.

To overcome this 1-bit branch prediction tech. is used.

```
while (1)
{
    for (i=0; i<10; i++)
    {
        true
    }
    false
    for (j=0; j<20; j++)
    {
    }
```

Predict Untaken :-

NT – 0
T – 1

i =	0	1	2	3	4	5	6	7	8	9	10
Actual outcome	1	1	1	1	1	1	1	1	1	1	0
Predict – NT	0	0	0	0	0	0	0	0	0	0	0

1/11 is correct in this prediction technique

```
while (1)
{
    for (i=0; i<10; i++)        BR1
    {
        (BR1)
    }
    for (j=0; j<20; j++)        (BR1)
    {
    } (BR1)
}
```

(if true) 1/T

(false) 0/NT

i =	0	1	2	3	4	5	6	7	8	9	10
actual outcome	1	1	1	1	1	1	1	1	1	1	0
P.NT –	0	0	0	0	0	0	0	0	0	0	0

1/11, 7/22 → 2/32

1-bit –

0 1 1 1 1 1 1 1 1 1 1 1

– 2/11, 2/21 = 28/32

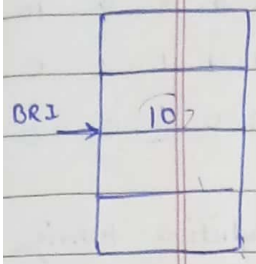
→ 4 bit addressing :- 16 address in buffer

0000 | 00
: | 01
: | 10
 | 11
 | 00
 | 01
 | 10
 | 11

BR1 [| 0111]
10 bit

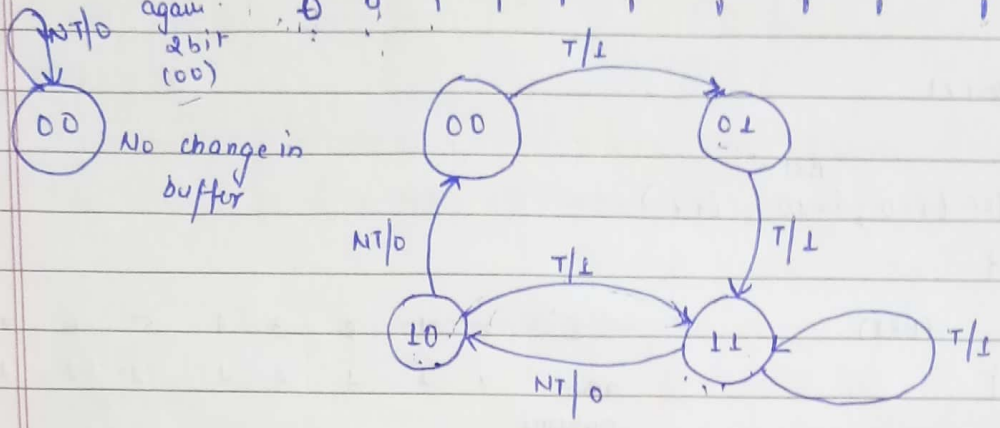
00 = Strongly not taken
01 = Weakly not taken
10 = Weakly taken
11 = Strongly taken

→ 2 bit prediction :-



i	0	1	2	3	4	5	6	7	8	9	10
Actual o/come	1	1	1	1	1	1	1	1	1	1	0
2 bit (start with 10)	1	1	1	1	1	1	1	1	1	1	1
again 2 bit (00)	0	0	1	1	1	1	1	1	1	1	1

10/11 ↑ correct



~~2b~~ buffer 00 → 11 → 11 · - - - - - 11 → 10

again 10 → 11

↑ 1st loop · ↑ 2nd loop

in 1 bit, 9/11 → 19/11

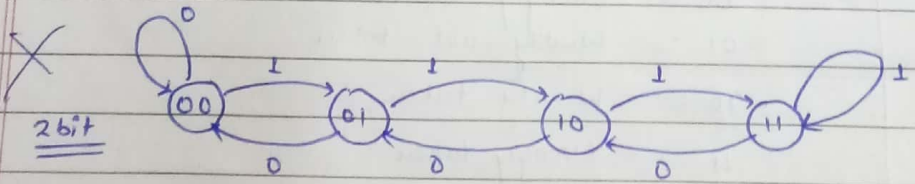
in 2bit, 10/11 → 20/11

∴ 1st loop – 28/32 = 87%

2nd loop – 30/32 = 93%

starting with 00, 00 → 01 → 11 → - - - - 10

Starting pattern can be anything but ending pattern is always 10.

→ 

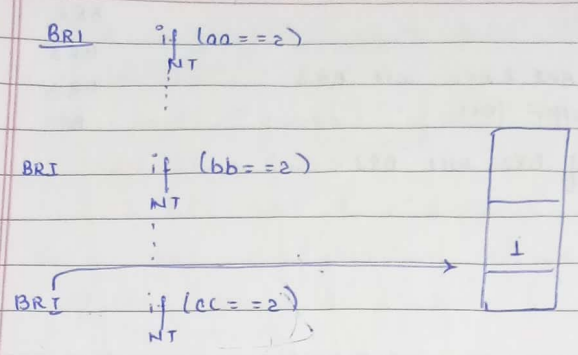
saturating counter

n bits buffer  7 bits

MSB

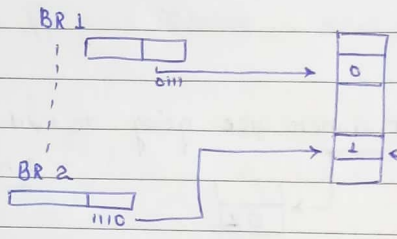
MSB = 1 → taken

MSB = 0 → Not taken

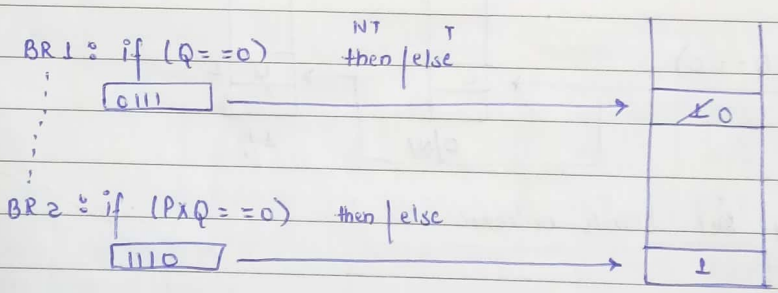


1 bit & 2 bit predictions fails in this example.

→ Correlating predictor (m,n)



If real outcome in BR 1 is 1 then prediction is wrong ∴ flips 0 → 1



→ (m,n)

m = { last outcome of m branches
2^m Branch Histor Table

n = { every slot is occupied by n bit binary value

BR0
BR1
BR2
BR3

eg (2,1)

2 → { outcome of BR1 & BR2 wrt BR3
$2^2 = 4$ buffer | BHT

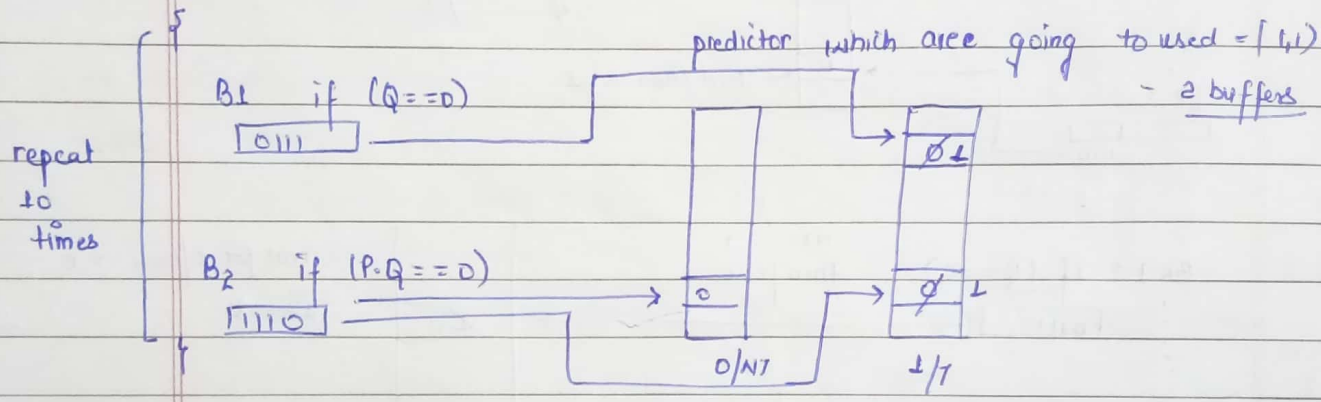
if m=1, outcome of BR2 wrt BR3.

1 → {0,1}

eg (2,2)

2 = { outcome of BR1 & BR2 wrt BR3
$2^2 = 4$ BHT

BR_x 2 → {00, 01, 10, 11}



Consider only last branch outcome

i	1	2	3	4	5	6	7	8	9	10
Real → BR1	NT	NT	NT	NT	NT	T	T	T	T	T
BR2	NT	NT	NT	NT	NT	T	T	T	T	T

1st buffer — 2nd buffer

this may be wrong.

1/100 = wrong.

(Not taken = then past)

when BR1 = Not taken, then BR2 must be not taken according to correlating predictor.

Which buffer I have to index ?

→ BR1 consider outcome of BR_x

Let BR_x = taken

∴ BR1 index to 1 buffer.

Let us take 0 is prediction for BR1 (Also taken)

If BR1 = taken (real outcome) & flip 0 → 1

for BR2, index to 1 buffer

prediction for BR2 = NT but real outcome = taken ∴ flip 0 → 1

For next iteration, BR2 is last branch for BR1

BR_x = NT

→ eg

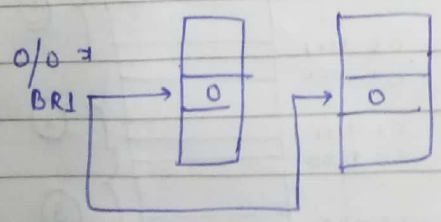
5 times

B_1 : T, NT, T, NT, T

B_2 : T, T, T, NT, NT

Predictor (1,1)

Iteration	BR1 prediction (buffer)	BR1 prediction	Actual outcome	Values in buffer	BR2 Prediction	BR2 Actual outcome
1	0/0	NT	T	0/0	NT	T
2	1/0	NT	NT	0/1	NT	T
3	1/0	NT	T	1/1	T	T
4	1/1	T	NT	1/1	T	T
5	1/0	NT	T	1/1	T	NT
exit	1/1			1/0		



Working of Corelating predictor (1,1)

Representation of 1 bit predictor

= (0,1) correlating predictor

$BR_X = NT$, $BR_Y = NT$

predictor (2,1)

iter	predictor (buffer) BR1	prediction	Actual
1	0/0/0/0	NT	T
2	1/0/0/0	NT	NT
3	1/0/0/0	NT	T
4	1/1/0/0	NT	NT
5	1/1/0/0	T	T
exit	1/1/0/0		

BR2

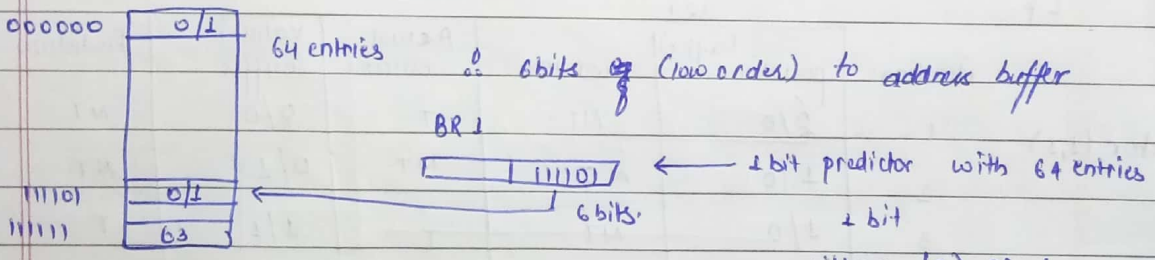
Predictor (buffer	prediction	Actual
0/0/0/0	NT	T
0/1/0/0	NT	T
0/1/1/0	NT	T
0/1/1/1	T	T
0/1/1/1	T	NT
0/1/1/0		

Q-1 How many bits are in the (0,2) branch predictor with 4K entries.?

Q-2 How many entries are in a (2,2) predictor with the same no. of bits. (as in ques 1)

→ Entries = slots

4K entries = 4×2^{10} slots (entries) = 2^{12} entries

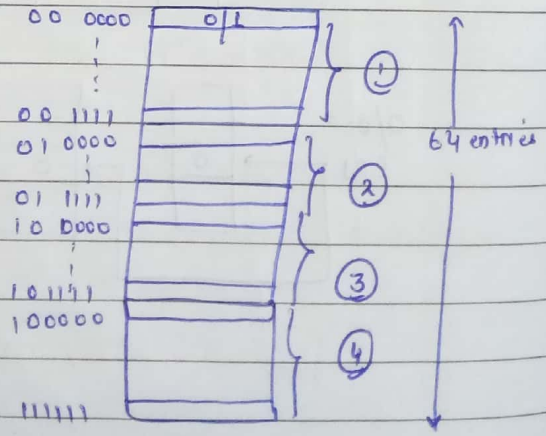


(0,2) = 2 bit predictor

(2,1) = - Consult last 2 branches

- 4 buffers
- each slot of buffer is 0/1

Physically there is only one buffer



Soln (0,2) :- 1 buffer

- 4k entries = 2^{12} entries = 12 bits
- each slot = 2 bits {00, 01, 10, 11}

∴ 12 bits - $2^{12} \cdot 2^1 = 8K$

2^m =

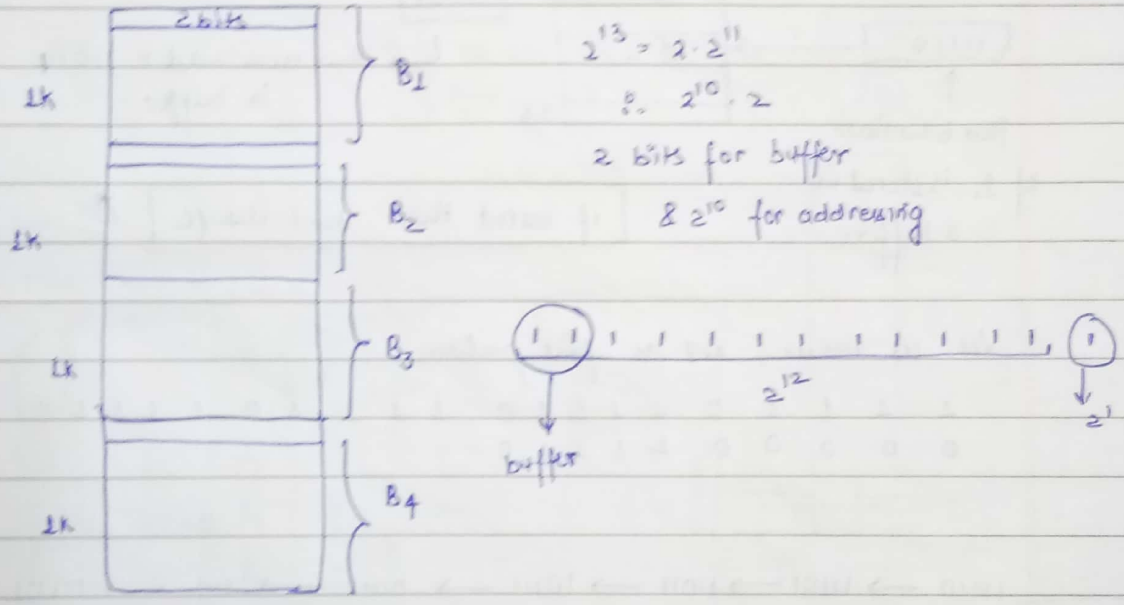
total 18 bits

$2^m \times n \times 4B$

$2^0 \times 2^1 \times 4$

(8 K)

for (2,2) predictor



$2^{13} = 2 \cdot 2^{12}$

∴ $2^{10} \cdot 2$

2 bits for buffer

& 2^{10} for addressing

Q. →

The no. of bits in an (m,n) predictor = $2^m \cdot n$ • (Number of predictor entries selected by the branch address)

eg for (4,2) predictor

→ 12 bit size buffer = 4K entries

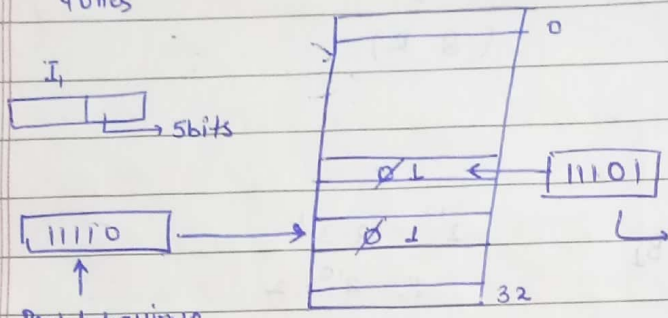
→ Local predictor :-

5 times → for() → T T T T NT
1 1 1 1 0

3 ones, 0, 1 one

1110 | 11110 | 11110 | 11110 | 11110 → No pattern of with 5 ones

4 Ones



Now add. changes to 11101

↳ Now new add = 11010 in buffer

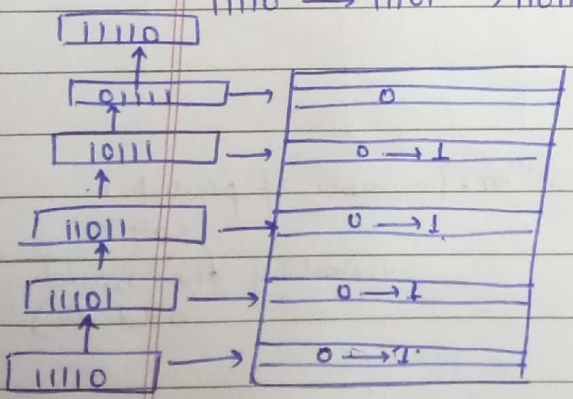
Past behaviour of I1 is stored in a buffer.

if correct then don't change *

Let us assume, NT as first prediction

1 1 1 1 0 1 1 1 1 0 1 1 1 1 0 1 1 1 1 0 1 1 1 1 0
0 0 0 0 0 1 1 1 1 0

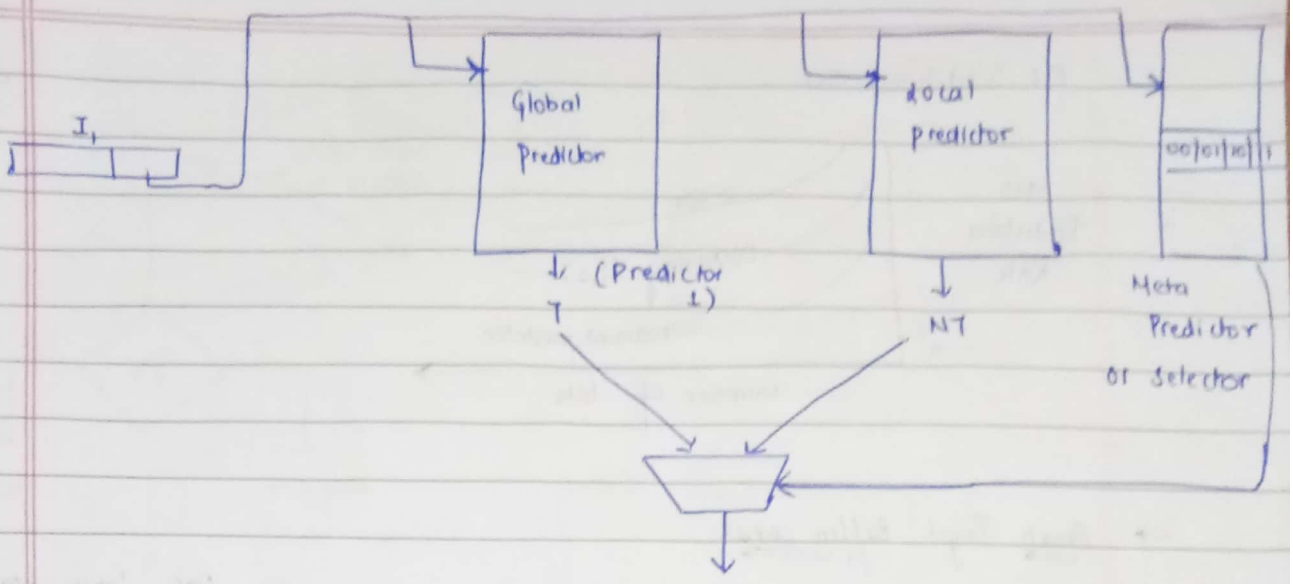
11110 → 11101 → 11011 → 10111 → 01111 → 11110 → 11101 → 11011



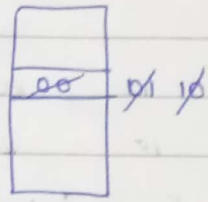
for n iteration, 95% - 100% predictions are correct.

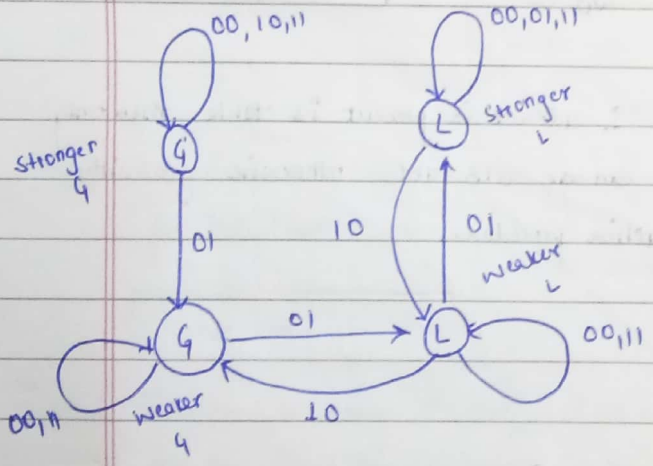
→ If I1 is influenced by itself → go for local predictor & if it is influenced by other branches, go for global

Tournament Predictor



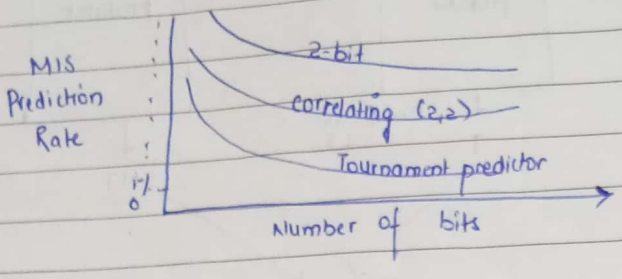
	predictor 1	predictor 2	Last Selection = G (assume)
both prediction wrong	0 X	: 0X	Meta predictor - 00
	0X	1✓ (Meta → 01) (G=NT, L=T)	It means, last time selected G but both predictions are wrong (G=T, L=T)
again selecting G		(Meta = 10)	





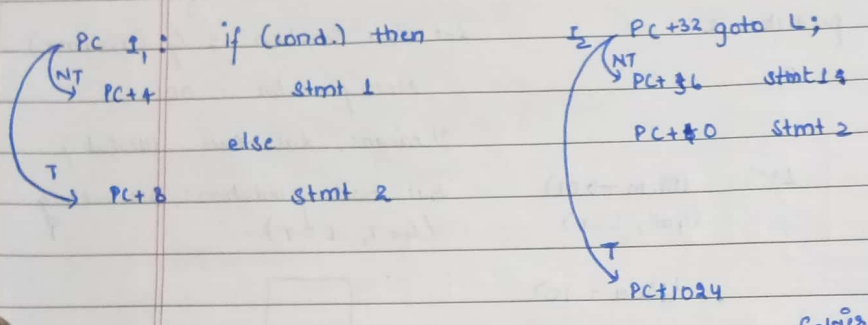
eg	predictor 1	predictor 2	G | L
	0	0	G^S
	0	1	G^W
	0	0	G^W
	1	1	G^W
	0	1	L^W
	0	1	L^S_W
	01	0	L

For Conditional inst.



→ Branch Target Buffer (BTB)

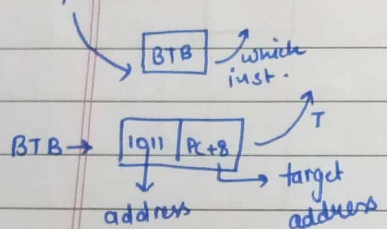
Only Taken branches are in buffer.



	PC	Predicted PC
Entries	PC	PC+8
	PC+32	PC+1024

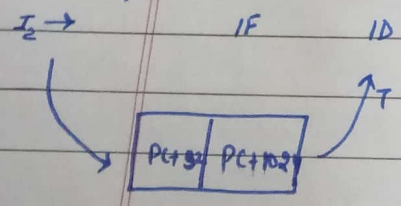
Ex 1-

	1cc	2cc	3cc	4cc	5cc
I_1 →	IF	ID	EX	MEM	WB

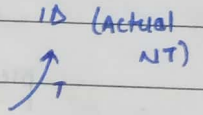


If I_1 address is present in table then only we can use BTB table otherwise go with earlier procedure.

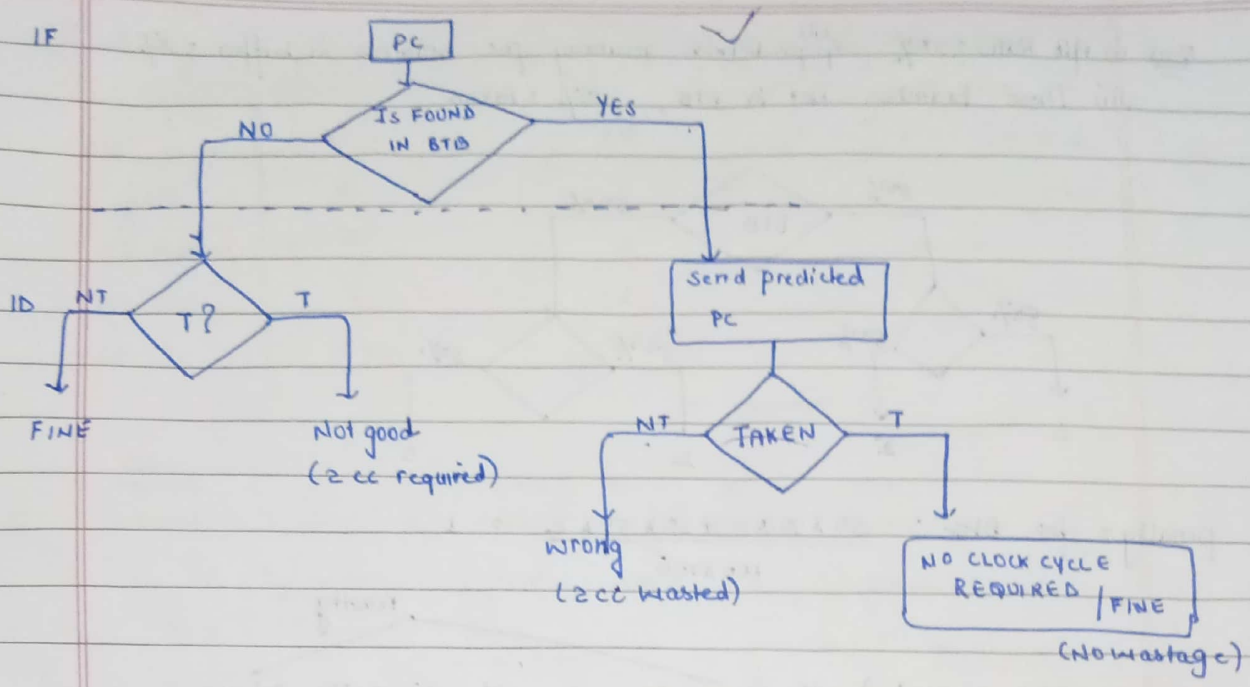
eg:-



∴ it saves a lot of time



find wrong & right one & update the buffer values.



Inst. in buffer (BTB)	Prediction	Actual Outcome	Penalty
YES	T	T	0
YES	T	NT	2
NO	NT	NT	0
NO	NT	T	2

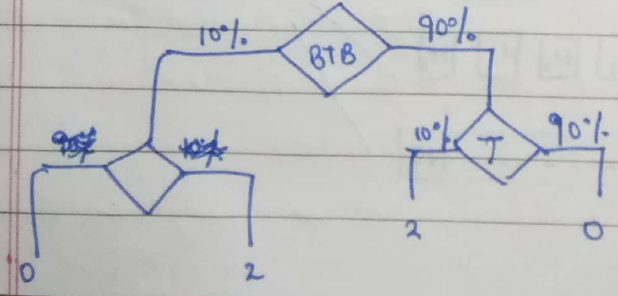
Ques Determine the total branch penalty for BTB assuming the penalty cycles for individual mis prediction. Make the following assumptions about the prediction accuracy & hit rate. (from above table)

(i) Prediction accuracy is 90% (for inst. in the buffer)

(ii) Hit Rate in buffer is 90%, (for branches predicted Taken)

↓ (all branches are taken)

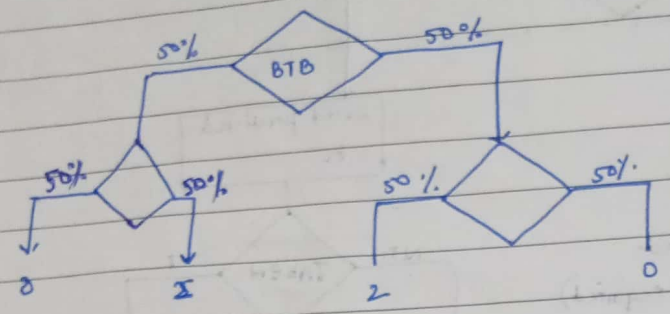
Soln Hit Rate = 90 instructions are present in buffer



$\frac{90 \times 10 \times 2}{10000} + \frac{10 \times 10 \times 2}{100}$

$\Rightarrow$ 0.38

Ques (i) Hit Rate = 50% (ii) & prediction accuracy for branches in buffer = 50%.
(iii) Those branches Not in BTB, 50% = taken



penalty = ~~for BTB~~ $\frac{50 \times 50 \times 2 + 50 \times 50 \times 2}{100 \times 100} = 1$

Penalty

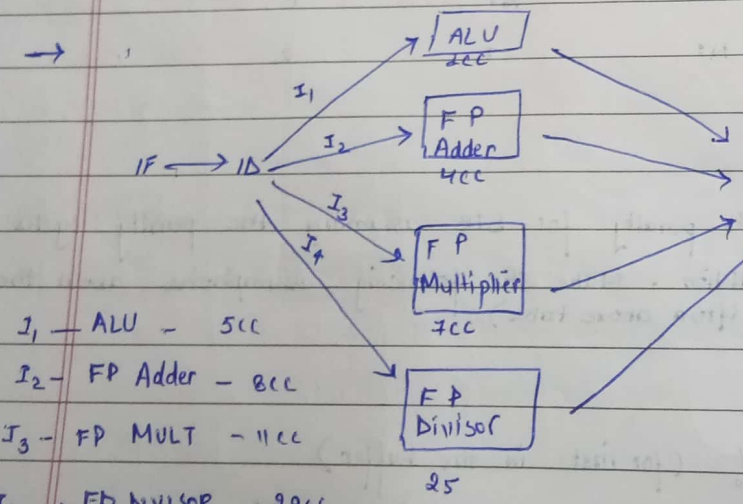
P(in buffer, T) = 1/2 x 2 = (0.25 x 2)

P(Not in buffer, T) x 2 = (0.25 x 2)

P(in buffer, NT) x 0 = (0.25 x 0)

P(NOT in buffer, NT) x 0 = (0.25 x 0)

→

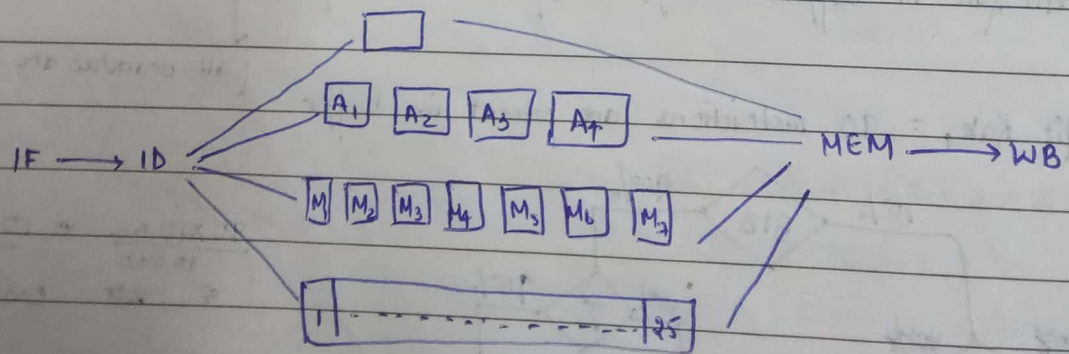


I_1 — ALU - 5cc

I_2 - FP Adder - 8cc

I_3 - FP MULT - 11cc

I_4 — FP DIVISOR - 29cc



1) Latency 2) Initiation Interval

	L	II
ALU	0	1
FP Adder	3	1
FP Mul	6	1
FP D	24	25 *

Latency

	1cc	2cc	3cc	4cc	5cc
x = a+b I_1 →	IF	ID	EX	MEM	WB
I_2 →	IF	ID	EX	MEM	WB

	I_1 →	IF	ID	A_1	A_2	A_3	A_4	MEM	WB
(Not dependent)	I_2 →	IF	ID	A_1	A_2	A_3	A_4	MEM	WB

dependent on o/p of I_1

(Dependent) ∵	I_2 →	IF	ID	B	B	B	A_1	A_2	A_3	A_4 MEM WB

3 gap

Initiation Interval :-

In case of division, I_2 can't enter in (1-25) when I_1 is present ∵ I_2 is dependent on I_1

I_1	IF	ID
I_2		IF

MUL D F_0 F_2* F_4 →

Double ↑

	IF	ID	M_1	M_2	M_3	M_4	M_5	M_6	M_7	MEM	WB
	IF	ID	A_1	A_2	A_3	A_4	MEM	WB			

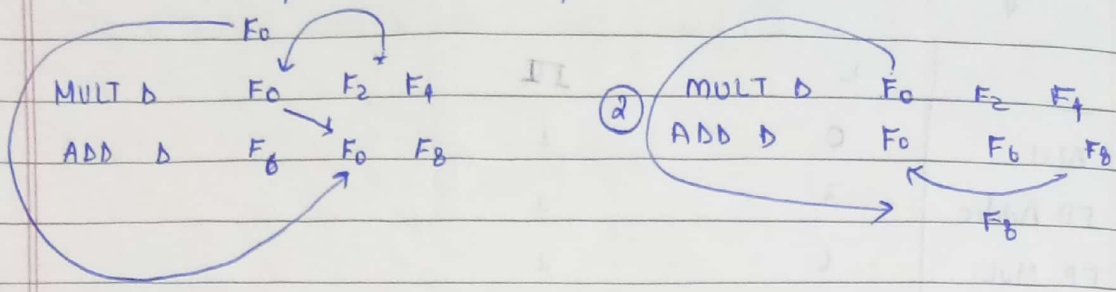
ADD D	F_6	F_8	F_{10}					
L D	F_{12}	F_{14}	F_{16}	IF	ID	EX	MEM	WB
S D	F_{18}	F_{20}	F_{22}	IF	ID	EX	MEM	WB

out-of-order

0 0 0

IF ID EX MEM WB

* same register is used by 2 instructions.



→ Data Dependence & Hazards

Data Dependence

1) Data dependence or true data dependence
2) Anti dependancy } Name dependance
3) Output dependance

1) True Data Dependence :-

RAW → (Read After Write)

I: MULT D F0 F2 F4
J: ADD D F6 F0 F8

(z = x+y
a = z+b)

Can't proceed J out of order ie can't be executed before I

2) Anti Dependence :-

WAR-Hazard → (write after Read)

I: MULT D F0 F2 F4
J: ADD D F2 F6 F8

(z = x+y
x = a+b)

No data is transferred from I → J but there is dependency ie if J is executed before I, value of F_2 will is updated which makes F_0 incorrect.

So first I is executed & then J

3) Output Dependence :-

WAW → (write After write)

I: MULT D F0 F2 F4
J: ADD D F0 F6 F8
⋮
SUB D F10 F0 F14

If J is executed before I then value of F_0 will be updated & in last execution, value of F_{10} is not correct.

RAR
(This is not a Hazard)

MULT D F0 F4 F6
ADD D F8 F4 F10

How to Overcome this Hazards?

1] RAW :- reading of ~~1st~~ 2nd inst. before writing of 1st inst.

MULT D F0 F2 F4
ADD D F6 F0 F8

- Soln :- fill this by delayed inst.

x = a+b
y = x+z ⟶ delayed inst.

2] WAR Hazard :- This method is called Register Renaming

WAR [MULT D F0 F2 F4
WAW [ADD D F2 F6 F8
SUB D F2 F4 F10] WAR
RAW [ADD D F8 F12 F4
SUB D F6 F14 F16] RWAR

Soln :- 1) Replace $F_2 \rightarrow F_x$ in eqn 2
So replace all $F_2 \rightarrow F_x$ in all eqn except 1

2) Replace $F_x \rightarrow F_y$ in eqn (3) (Replace all F_x)

3] Replace $F_8 \rightarrow F_z$ in eqn (4) (Replace all F_8)

4] Replace $F_6 \rightarrow F_A$ in eqn (5)

MULT D F0 F2 F4
ADD D Fx F6 F8
SUB D Fy F4 F10
RAW [ADD D Fz F12 F4
SUB D FA F14 F16

Advance pipelining

eg

ADDD F_0 F_2 F_4] RAW & WAR] RAW & WAR] WAW

WAW [ADDD F_2 F_0 F_4

WAR [ADDD F_2 F_0 F_6

WAR [ADDD F_0 F_2 F_6] WAR] WAR

[ADDD F_8 F_6 F_6

$F_2 \rightarrow F_A$

F_0 F_2 F_4

F_A F_0 F_4

F_8 F_0 F_6

F_C F_B F_6

F_D F_C F_6

→ Advance pipelining

I → S/w Techniques | static scheduling
compiler Techniques

II → H/w Techniques | Dynamic scheduling

I :- S/w Techniques

- → loop unrolling
- → Software pipelining
- → VLIW
- → predicate Instruction

II : Hardware Techniques

- → Score boarding
- → Tomasula Alg
- → Tomasula Alg + loops
- → Tomasula Alg + ROB
- → Tomasula Alg + ROB + VLIW

→ Compiler Techniques :-

- loop unrolling

```
for(i=1; i<=1000 ; i++)   } — 1000 times checking
  x[i] = x[i] + 10
```

↳ rolled version

```
for (i=1 ; i<=1000; i+=4)
  x[i]   = x[i] + 10;
  x[i+1] = x[i+1] + 10;
  x[i+2] = x[i+2] + 10;
  x[i+3] = x[i+3] + 10;
```

250 times checking

Unrolled 4 times

if 8 times then → 125

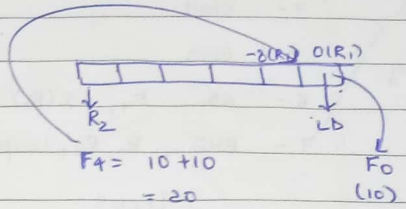
$\frac{1000}{125}$ = 8 times complexity is reduced.

```
for (i=999; i>=0; i--)
    x[i] = x[i] + s;
```

Assembly code

```
Loop : LD   F0, 0(R1)
       ADD  F4, F0, F2      → const. value
       SD   F4, 0(R1)
       DADDUI  R1, R1, #-8
       BNE  R1, R2, Loop
```

(branch Not equal)



	IPR	IUR (Inst. Using Result)	Latency
(1)	FP op	FP op	3
(2)	FP op	Store	2
(3)	Load	FP op	1
(4)	Load	Store	0

LD F0, 0(R1) → IF ID EX MEM WB → 1 gap

Add F4, F0 F2 → ~~FF~~ IF ----------

x[i] = x[i] + 10

```
LD   F0  0(R1)      IF (1)        ↑ cc
Stall               (2)
Add  F4, F0  F2     IF (3)
Stall               (4)
Stall               (5)
SD   F4, 0(R1)      IF (6)
DADDUI  R1, R1, #-8 (7)   → i = i-1
Stall               (8)
BNE  R1, R2, Loop   (9)   → i >= 0
```

eg =>

1- LD F0 , 0(R1)
2- DADDUI R1 , R1 , #-8
3- ADD D F4 F0 F2
4- Stall
5- Stall
6- SD F4, 8(R1)
7- BNE R1, R2, loop

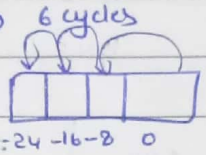
avoided in unrolling

Loop:

Unrolling — x[i]

LD F0 , 0(R1) } 1 gap
ADDD F4, F0 F2 } 2 gap
SD F4, 0(R1)

⇒ 6 cycles



Replace F4 → FA x[i+1]
F0 → Fx

LD F0, -8(R1)
ADD F4, F0, F2
SD F4, -8(R1)
= 6 CC

Replace F4 → FB x[i+2]
F0 → Fy

LD F0, -16(R1)
ADD F4, F0, F2
SD F4, -16(R1)
6 CC

Replace F4 → Fc x[i+3]
F0 → Fz

LD F0, -24(R1)
ADD F4, F0, F2
SD F4, -24(R1)
6 CC

DADDUI R1 R1 #-32 } 1 CC
BNE R1 R2 loop

∴ 27 CC for complete process.

Gap utilized

LD F0 , 0(R1)
LD FX , -8(R1)
LD FY , -16(R1)
LD FZ , -32(R1)

ADD F4 , F0, F2
ADD FA , FX , F2
ADD FB, FY , F2
ADD Fc, FZ, F2

SD F4 , 0(R1)
SD FA , -8(R1)
SD FB , +16(R1) (to adjust)
SD Fc , +8 (R1)
DADDUI R1, R1 #-32
BNE R1 R2 , loop

∴ only = 14 CC

Gap b/w load & ADD = 1, ~~not filled~~ consumed by 3 cc

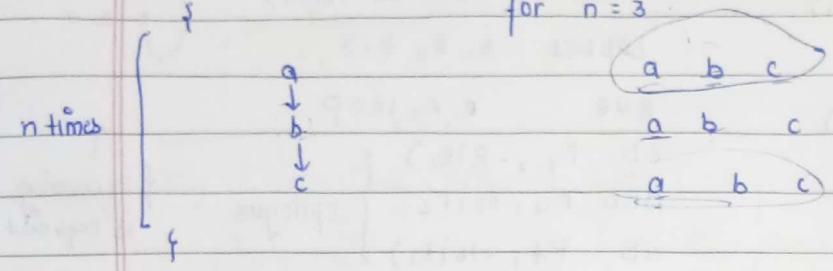
Add & store = 2 " by 3 cc

DADDUI & BNE = 1 , consumed by 2 cc

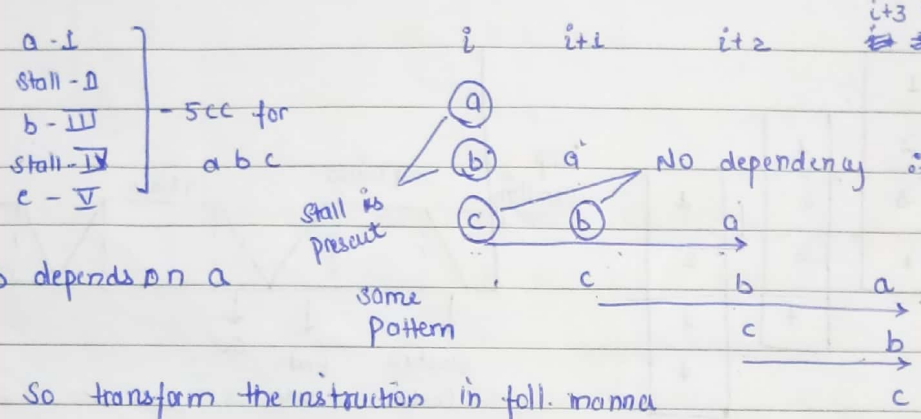
Unrolled 4 times-

2 Software pipelining :-

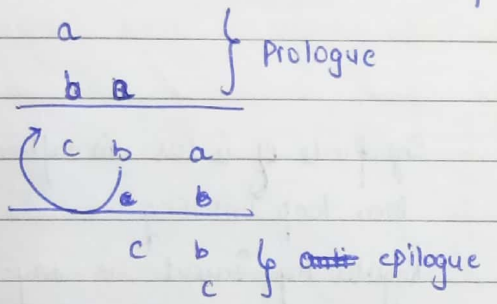
for n = 3



So we execute the inst- in following way.



So transform the instruction in foll. manner



eg Loop LD F_0, 0(R_1) Unrolling 3 times

ADD F_4, F_0, F_2

SD F_4, 0(R_1)

DADDUI R_1, R_1, #-8

BNE R_1, R_2, Loop

i : LD F0, 0(R1)
ADD F4, F0, F2
SD F4, 0(R1)

∴ pattern - 3rd inst of i_1, 2nd of i_2 & 1st of i_3

Prologue {
LD F0, 0(R1)
ADD F4, F0, F2
LD F0, -8(R1)

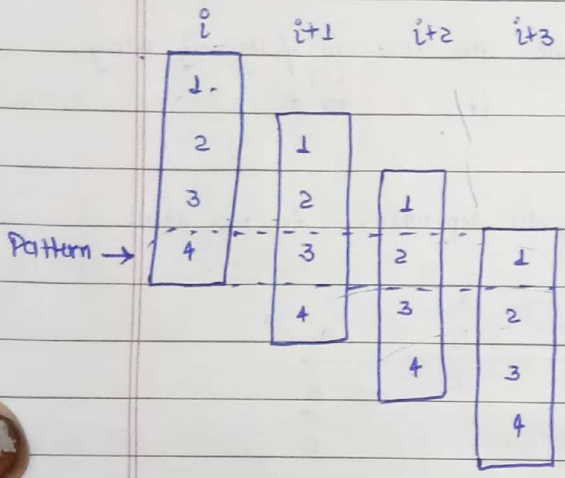
i+1 : LD F0, -8(R1)
ADD F4, F0, F2
SD F4, -8(R1)

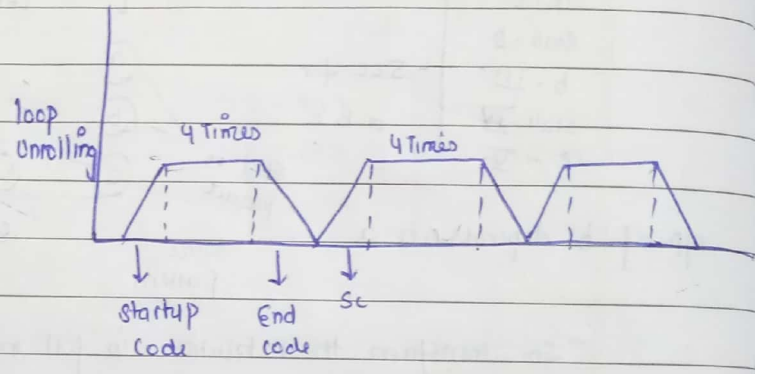
i+2 : LD F0, -16(R1)
ADD F4, F0, F2
SD F4, -16(R1)

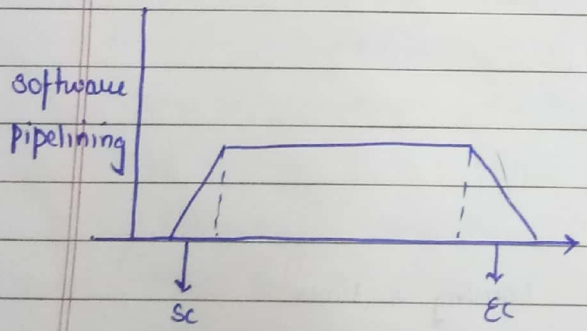
Loop: SD F4, 0(R1)
ADD F4, F0, F2
LD F0, -16(R1)
DADDUI R1, R2, #-8
BNE R1, R2, loop

epilogue {
SD F4, -8(R1)
ADD F4, F0, F2
SD F4, -16(R1)

Renaming is required



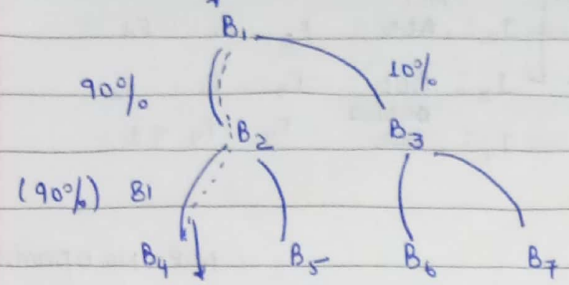


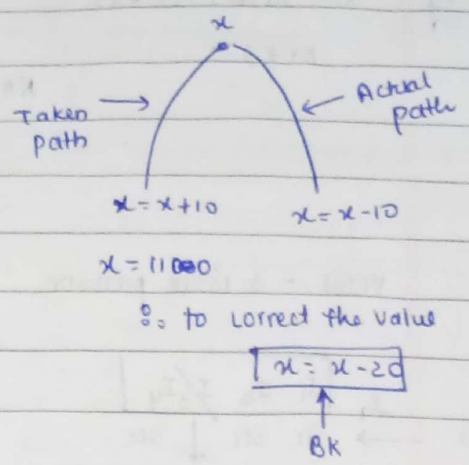


- Size of code is less in software pipelining then loop unrolling
- Register requirement is very less

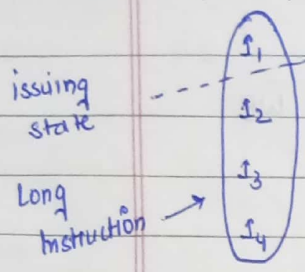
To overcome the problems of nested loops, loop unrolling & software pipeline fails so we use trace scheduling.

Trace Scheduling :-





→ VLIW (Very Long Instruction Word)



	IF	ID	EX (ALU1)	MEM	WB
I1	IF	ID	EX (ALU1)		
I2		IF	ID		
I3			IF	ID	

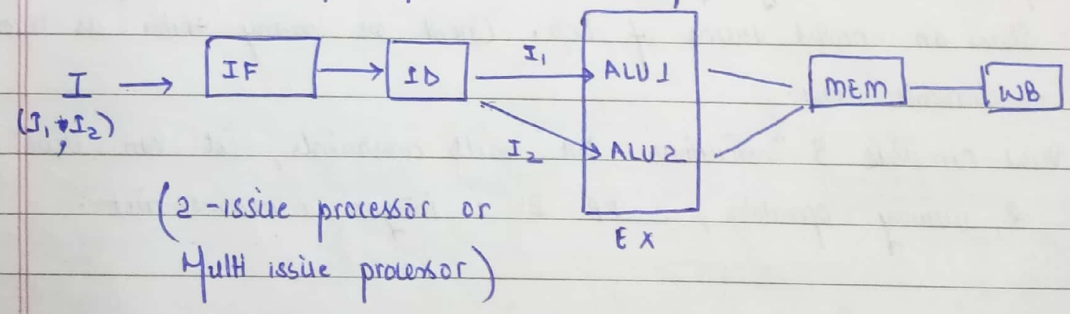
Let there are 2 ALU

{ ALU 1
ALU 2 }

I_1, I_2 are Independent

At 3cc, ALU 1 is busy & ALU 2 is ideal
At 4cc, ALU 2 is busy & ALU 1 is ideal
} inefficient use of h/w

In VLIW, pairing of inst- is done . both I_1 & I_2 is issued at the same time with a fetching state & decoding state.



(2-issue processor or Multi issue processor)

Multi issue processor
- → Statically Scheduled Superscaler processor
- → VLIW
- → Dynamically Scheduled Superscaler processor

Assembly code

eg: $x = (a+b) + (c*d)$

b++;

RAW

I_1 : ADD F_1 F_a F_b

MUL

I_2 : ~~ADD~~ F_2 F_c F_d

RAW

I_3 : ADD F_3 F_1 F_2

I_4 : ~~ADD~~ ADDUI F_4 F_4 #1

VLIW - 3 issue processor

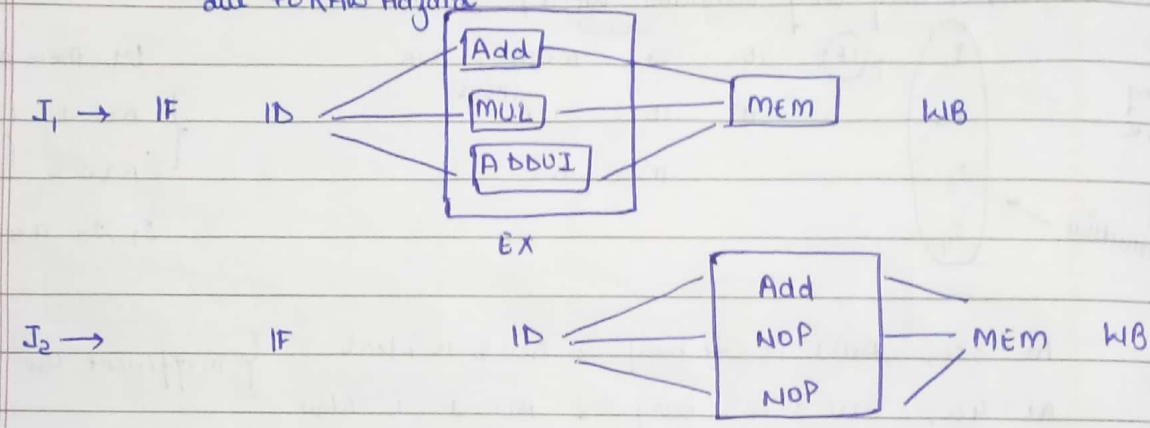
NOP = No operation

I_1 | I_1 I_2 ~~I_3~~ I_4 | I_2 | I_3 | NOP | NOP |

Completes in → 5cc 5cc 5cc

I_2 → 6cc

Not allowed due to RAW Hazard



Q Suppose we have VLIW that could issue 2 memory references, ~~1~~ 2 FP operations & 1 integer operation or branch in every clock cycle. Show an unrolled version of loop. Unroll as many times as necessary to eliminate stalls.

Soln VLIW can issue 5 instructions but with constraints, it can issue only 2 memory operation, 2 FP & 1 integer or branch inst.

Loop:	LD_1 F_0, 0(R_1)	ADD D_1 F_4, F_0, F_2	SD_1 F_4, 0(R_1)
	LD_2 F_6, -8(R_1)	ADD D_2 F_8, F_6, F_2	SD_2 F_8, -8(R_1)
	LD_3 F_{10}, -16(R_1)	ADD D_3 F_{12}, F_{10}, F_2	I_1 → DADDUI R_1, R_1, #-32
	LD_4 F_{14}, -24(R_1)	ADD D_4 F_{16}, F_{14}, F_2	SD_3 F_{12}, +16(R_1)
	LD_5 F_{18}, -32(R_1)	ADD D_5 F_{20}, F_{18}, F_2	SD_4 F_{16}, ~~-24~~ -8(R_1)
	LD_6 F_{22}, -40(R_1)	ADD D_6 F_{24}, F_{22}, F_2	B_1 → BNE R_1, R_2 loop
	LD_7 F_{26}, -48(R_1)	ADD D_7 F_{28}, F_{26}, F_2	SD_5
			SD_6
			SD_7

	M_1	M_2	FP_1	FP_2	I/B	cc
	L_1	L_2	ADD1 ↓ dependent on 4	ADD2 ↓ dependent on L_2	I_1 ↓ Not required (dependent on all inst)	
1st issue ∴	L_1	L_2	–	–	–	1cc
	L_3	L_4	ADD_1	ADD_2	I_1	
			can't keep due to latency (one gap is b/w L_1 & ADD_1 & L_2 & ADD_2)			
2nd issue	L_3	L_4	–	–	–	2cc
3rd issue	L_5	L_6	ADD1	ADD_2	~~I_1~~ –	3cc
4th issue	L_7	⊖	ADD3	ADD_4	–	4cc
		Can't put store because latency b/w ADD1 & S1 is 2cc				
5th issue	~~S1~~ –	–	ADD_5	ADD_6	–	
	latency b/w ADD1 & SD1 is 2cc					
6th issue	SD_1	SD_2	ADD_7	–	–	
7th issue	SD_3	SD_4	–	–	I_1	
8th issue	SD_5	SD_6	–	–	B_1	
9th issue	SD_7	–	–	–	–	

latency b/w I & B is 1 cc

∴ total 9cc VLIW taken to issue all instructions.
But in case of unrolling, 23 cc is required. So most of hardware is ideal but in case of VLIW, all hardware is used efficiently.

TRACE SCHEDULING →

```
for (i=0; i<3; i++)
    x[i] = x[i] + 10
```

⇒ unrolling

```
x[0] = x[0] + 10
x[1] = x[1] + 10
x[2] = x[2] + 10
```

— Basic Block

Single Basic Blocks - Execution of statement without any jump except at last statement sequentially

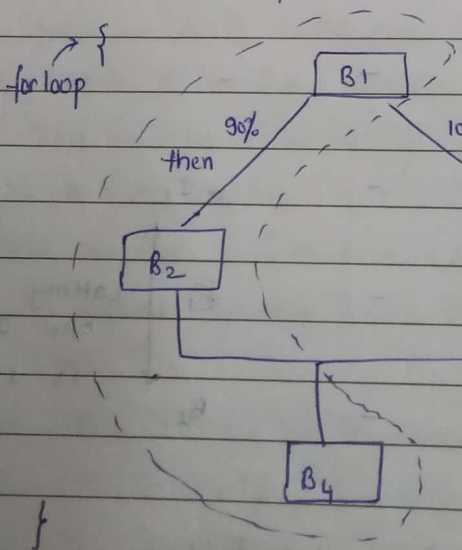
```
w = 0
x = x+y
if (x > y)
{
    z++
    x = x-y
}
else {
    y++
    y = x+y
}
a = b+c
```

BB1 BB2 BB3 BB4

```
test = y[i];
if (test > 0) then
    sum = sum + 10;
else
    sum = sum + 20;
c[x] = c[x] + c[y];
```

B1 B2 B3 B4

? How to Convert Multiple Basic Block into Single Basic Block ?



→ After tracing

```
test = y[i]
if (test > 0) then
    sum = sum + 10
    c[x] = c[x] + c[y]
```

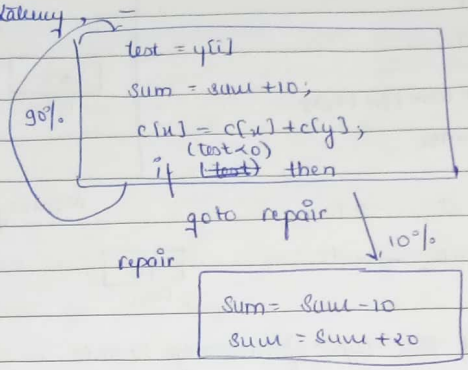
latency

⇓

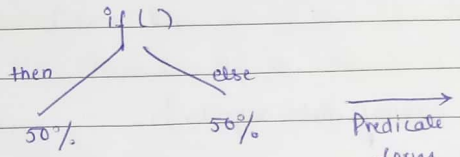
Not Single Basic Block.

After executing 100 times, note down the most frequent path of .

∴ To fill latency, —



→ Predicate Instruction :-



$R_7 = \; !R_1$

$R_2 = R_3 + R_4$ (Predicate on R_7)

$R_4 = R_5 + R_6$ (Predicate on R_1)

if (R_1 = =0) then

$R_2 = R_3 + R_4$;

else

$R_4 = R_5 + R_6$;

→ Hardware Techniques :- (Dynamic Scheduling)

Data Hazards - RAW → MULT D F_0 F_2 F_4

ADD D F_1 F_0 F_3

MULT D F_0 F_2 F_4 ← WAW

F_0 F_1 F_6 WAR → MULTD F_0 F_2 F_4

ADDD F_2 F_6 F_8

All these Hazards require Registers ∴ also known as Register Data Hazards

Memory Data Hazard :-

LD$_1$
ADD$_1$
SD$_1$
LD$_2$
ADD$_2$
SD$_2$
DADDUI
BNE

⇒

LD$_1$
LD$_2$
ADD$_1$
ADD$_2$
SD$_1$
SD$_2$
DADDUI
BNE

We can't Bypass LD operation over Store operation.

wrong because of bypassing

50 → ABC

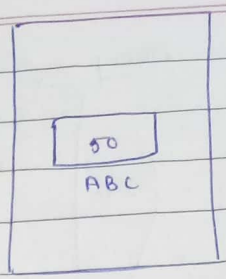
EA = Mem[0+(R1)]

LD1 F0, 0(R1)

SD1 F2, 8(R2)

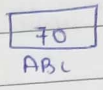
70

EA = Mem[8+(R2)] → ABC

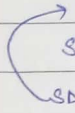


Actual answer: SD1 → 70

LD2 ← 70

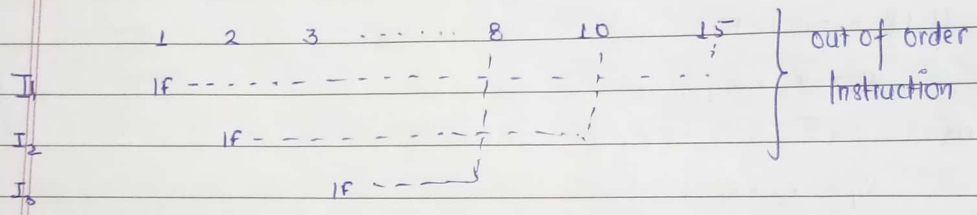


Compiler can't find this type of memory hazards.



WAW Hazard

Q How hardware techniques remove memory data hazards?

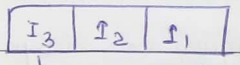


dependent DIV D F0 F2 F4

ADD D F6 F0 F8

independent → SUB D F10 F12 F4

∴ bypass



→ Bypass & execute

If I1 stucks in pipeline, then I2 can't execute but I3 can bypass & get executed.

- Register Data Hazards – Register renaming, Instruction Scheduling
- Memory Data Hazards
- Control Hazards – both s/w & h/w techniques are available

Compiler can't solve this problem

only h/w can solve

Hardware techniques to solve above problems are

- Scoreboarding
- Tomasulo's Algo
  - loop
  - ROB
  - loop + ROB + VLIW

IS - Issue Stage

RD - Read Operands

→ Scoreboarding :-

IF → IS (ID_1) → RD (ID_2) → Ex → WB

Fetches Instruction

1. Issue Stage :-
   - Check for structural Hazard. (functional unit is available or not)
   - Check for WAW Hazard
   - Issue the Instruction in pipeline

2. Read Operand :-

   ADD D F_0 F_2 F_4

   If both operands F_2 & F_4 are available then only execution takes place else RD has to wait (RAW Hazard)

→ Instruction Status Table :-

	IS	R	EX	WB
LD F_6, 34(R_2)	1	2	3	4
LD F_2, 45(R_3)	5	6	7	8
MULTD F_0, F_2, F_4	6	9	~~10~~ 19	20
SUBD F_8, F_6, F_2	7	9	11	12
DIV D F_{10}, F_0, F_6	8	21	61	62
ADDD F_6, F_8, F_{20} (WAR Hazard)	13	~~14~~	15	22 ~~15~~ X

execute

(what type of operation) (dest. register) Source operands (who produce F_j & F_k (operands are available or not))

	Busy	OP	dest	F_j	F_k	Q_j	Q_k	R_j	R_k
(1) Integer (to perform LD & simple Arithmetic OP.)	No	LD	F_6	-	R_2	-	-	-	Yes
(10) MULT 1	~~No~~ Yes	MULT	F_0	F_0	F_2	LD_2	-	~~No~~ Yes	Yes
MULT 2	No								
(2) ADD 1	~~No~~ Yes	SUBD	F_8	F_6	F_2	LD_1	LD_2	Yes	~~No~~ Yes
(40) DIV D 1	~~No~~ Yes	DIVD	F_{10}	F_0	F_6	MULTD	LD_1	No	Yes

Functional Unit Status Table

(clock cycles to execute)

Registers

F_6	F_2	F_0	F_4	F_8	F_{10}
LD	LD	MULT		SUB D	DIV D
ADD D					

- When job is over, remove the operation so at last all registers are available
- Initially LD operation check whether functional unit is available or not
- All Instructions follow the inorder
- At First clock cycle, LD 1 is issued
- At Second clock cycle, Integer function unit is not present for 2nd LD op.
  ∴ reading of operands of LD, takes place
- After 4th cc, Integer functional unit status → Yes → No
- All instructions are executed out of order.

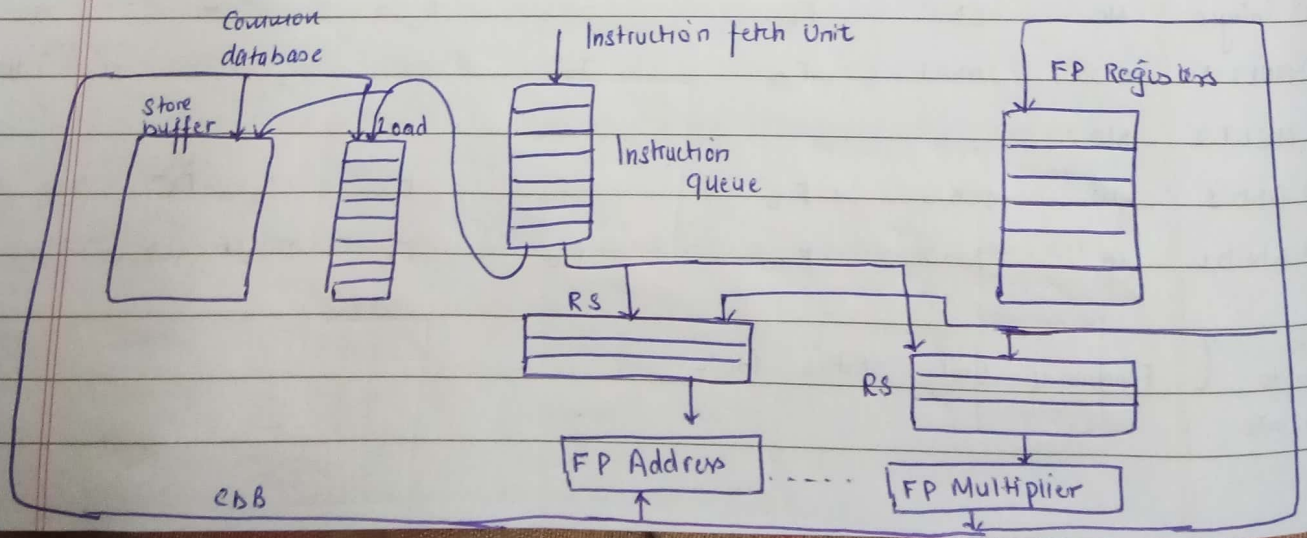
LIMITATION
- Scoreboarding can't solve WAW & WAR Hazard by register renaming
- Forwarding is Not present in Scoreboarding.
- Able to solve only single Basic Problem.

To solve the limitations of scoreboarding, we have Tomasulo's algorithm

Tomasulo's Algorithm :-

1) Reservation station (RS)
   - using this we can apply register renaming

RS, CDB → using these 2, we apply forwarding



In Scoreboarding, instructions passes 4 stages
In Tomasulo's algorithm, instruction passes 3 stages.

Scoreboarding → IS RD EX WB
Tomasulo's algo → IS EX WB
~~Issue Stage~~

2 types of functional units → FP Address, FP Multiplier

In Tomasulo's Algorithm, Issue(IS)
1) If RS is available, then I can issue ie copy the content of operand registers into the reservation station.

Execution - 1) If operands available then start execution
WB - 1) Write result in CDB
2) Write result in registers

Instruction status.

LD F_6, 34(R_2)
LD F_2 45(R_3)
MULTD F_0 F_2 F_4
SUBD F_8 F_6 F_2
DIVD F_{10} F_0 F_6
ADDD F_6 F_8 F_2

	IS	EX	WB
1	1	3	4
2	2	4	5
3	3	(5+10) 15	16
4	4	7	8
5	5	56	57
6	6	10	11

↓
diff than one in DIVD.

In DIVD, F_6 is stored in M1(A_1)
So ADDD will look in M1(A_1)
not in F_6 anywhere so WAR Hazard is completely eliminated.

Reservation Station

		Busy	OP	V_j	V_k	Q_j	Q_k
(2)	Add1	Yes	SUB	M(A1)	M(A2)	Load2	
(2)	Add2	Yes	ADD		M(A2)	SUB	
(2)	Add3						
(10)	Mul1	No	Yes	MUL	M(A1)	R(F4)	
(40)	DIV1	No	Yes	DIV		M(A1)	MUL

Once the Algo is over, the RS is completely free.

	Busy	A
Load1	Yes	34+R2 → New location M(A)
Load2	Yes	45+R3 M(A1) passed using CDB to all another units
Load3		

F6	F2	F4	F8	F10
M(A1)	M(A2)			

In 1st cc, check whether inst can be issued or not.

Above example enables forwarding prevents WAR & WAW Hazards Only

Limitation :- Common Data Bus → All units try to write on the CDB at the same time

↳ Soln → have multiple CDB but the complexity of architecture increases.

Ex-1 2cc I1 : $F_1 = F_3 + F_4$ WAW Hazard

4cc I2 : $F_1 = F_4 / F_2$

2cc I3 : $F_4 = F_1 - F_2$

Ex-2 $F_1 = F_3 + F_4$ WAR Hazard

$F_1 = F_4 / F_2$

$F_2 = F_4 - F_3$

Using Tomasulo's Algo →

1) $F_1 = F_3 + F_4$

$F_1 = F_4 / F_2$

$F_2 = F_4 - F_2$

	IS	EX	WB
1	4	5	
2	43	44	
3	44+2	47	

2 for reading (3, 4 for execution)

Original Table

	2.0
F_1	~~1.0~~
F_2	2.0
F_3	3.0
F_4	4.0

Register Renaming Table

	RS1	RS2
F_1	~~RS1~~	
F_2		
F_3		
F_4		

RS = Reservation Station

	OP
RS1	+
RS2	/
RS3	−

Ex-2

free 5cc

free 4cc

free 7cc

Q

RS = Reservation Station

	OP	OP1	OP2
RS1	+	3.0	4.0
RS2	/	4.0	2.0
RS3	-	~~RS1~~ 2.0	2.0

= ADD → 4cc → 7.0

= DIV → 43cc → (2.0)

= ADD → 46cc → (0.0)

When writing is done, Register renaming table & RS is made free.
All the things stored earlier are removed.

Ex-2

$F_1 = F_3 + F_4$

$F_1 = F_4 / F_2$

$F_2 = F_4 - F_3$

	IS	EX	WB
1	4	5
2	43	44
3	6	7

	RRT	free
F_0	44cc → 2.0	RS2 ~~44~~
F_1	1.0 → 15cc → 7.0	~~RS1~~ 5cc free
F_2	3.0 → 7cc (1.0)	~~RS3~~ 7cc free
F_3	3.0	
F_4	4.0	

free			OP	~~OP1~~	~~OP2~~	
free ~~5cc~~	RS1	+	3.0	4.0	= A → 4cc
free ← 44cc	RS2	/	4.0	2.0	A → 43cc
free ← 7cc	RS3	-	~~4.0~~	3.0	A → 6cc

Q

$F_2 = F_4 + F_1$

$F_1 = F_2 / F_3$

$F_4 = F_1 - F_2$

$F_1 = F_2 + F_3$

	IS	EX	WB
1	4	5
2	45	46
3	48	49
4	7	8

		7cc → 45cc 1.67
F_1	1.0 → 10.0	
F_2	2.0 → 4cc 5.0	
F_3	3.0	
F_4	4.0 → 7cc 7.0	

			OP	OP1	OP2	
F_1	~~RS3~~ ~~RS4~~ → free	RS1	+	4.0	1.0	A 5.0 — 4cc
F_2	~~RS1~~ 5cc free	RS2	-	~~RS3~~ → 1.66	~~RS1~~ 5.0	A 48cc — 3.34
F_3		RS3	/	~~RS1~~ 5.0	3.0	D 45cc 1.67
F_4	~~RS2~~ 49cc free	RS4	+	~~RS1~~ 5.0	3.0	A 8.0 — 7cc

F_1: ~~1.0~~ → 8cc 8.0

F_2: ~~2.0~~ → 5cc 5.0

F_3: 3.0

F_4: ~~4.0~~ → 49cc 3.34

→ Drawback of Tomasulo's -

Precise Exception - Division with zero detected
40th cc

I_1	DIVD	F_{10}	F_0	F_6	→ Imprecise Exception ~~Soln~~
I_2	LD	F_2	45(R_3)		Soln — ① I_1 · seq-exe
I_3	MULTD	F_0	F_2	F_4	I_2
I_4	SUBD	F_8	F_4	F_6	}

If $F_6 = 0$ → leads to exception (division by 0)

exceptional handler increases 0 → 0.00---1

② Store value only after previous instr. is solved.

→ To overcome exception and 'get out of order execution'

↳ Tomasulo's + ROB

					IS	EX	WB
2	I_1	SUBD	F_1	F_3 F_2	1	4	5
10	I_2	MULTD	F_4	F_2 F_3	2	13	14
40	I_3	DIVD	F_0	F_3 F_1	3		
2	I_4	ADDD	F_3	F_4 F_2	4	16	17

RRT

F_0	0.0		F_0	RS3
F_1	~~1.0~~	0.0	F_1	~~RS1~~ → SCC free
F_2	2.0		F_2	
F_3	~~2.0~~	6.0	F_3	~~RS4~~
F_4	~~4.0~~	4.0	F_4	RS2

RS1	SUBD	2.0	2.0	A — 4cc (0.0)
RS2	MULTD	2.0	~~2.0~~ 14cc	M — 13cc (4.0)
RS3	DIVD	2.0	~~RS1~~ (0.0)	D
RS4	ADDD	~~RS2~~ 4.0	2.0	A — 16cc (4.0)

System call to the exception handler which increases the value of $f_1 \rightarrow 0.$

$\longrightarrow 0.001$

Hence the system call return to the inst.

I_3 DIVD F_0 F_3 F_1

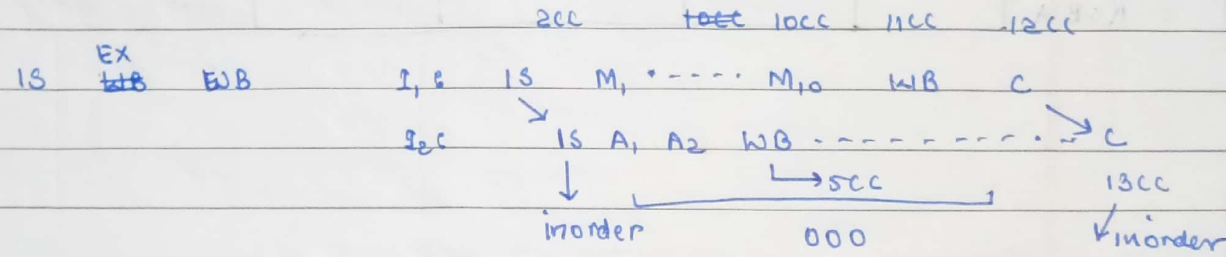
RS_3 DIVD 6.0 0.001

Wrong value of F_3 used which gives wrong result.

Re-Order Buffer:-

C: Comment

Tomasulo's Algorithm -



Hence if exception occur in I_1, I_2 restarted from the start

SUBD	F_1	F_3	F_2
MULTD	F_4	F_2	F_3
DIV D	F_0	F_3	F_1
ADDD	F_3	F_4	F_2

	IS	EX		W.	C
1	✓	4		5	6
2	✓	13		14	15
3	W	✓			
4	W	✓	16	17	W

all inst. flushed out of table & restarted

ROB entry

A	RS1	–	2.0	2.0	ROB0	4 → (0.0) + ROB0 (tag)
M	RS2	X	2.0	2.0	ROB1	13 → (4.0) + ROB1
D	RS3	D	2.0	~~ROB0~~ 0.0	ROB2	(start at 4CC)
A	RS4	A	~~ROB1~~ 4.0	2.0	ROB3	16 → (6.0) + ROB3

F_0	0.0
F_1	1.0
F_2	2.0
F_3	3.0
F_4	4.0

F_0	ROB 2
F_1	ROB 0 → 0.0 (in 6cc)
F_2	
F_3	ROB 3
F_4	ROB 1 → 4.0 (in 15cc)

	Type	dest.	Value	Status	
ROB 0	S	F_1	- / 0.0	0	H
ROB 1	M	F_4	- / 4.0 (in 4cc)	0	H
ROB 2	D	F_0	-	0	H
ROB 3	A	F_3	- / 6.0 (in 7cc)	0	
ROB 4					

Head H (circular queue)

Speculating - Not confirmed

→ Tomasulo's Algo for Branches (Hardware Based Speculation)

Eg:

B_1 - MULT F_2 F_0 F_1
BNEQ F_2, 0, label

B_2 - ADD D F_4 F_3 F_1
SUB D F_5 F_3 F_1

B_3 - label: ADDD F_3 F_4 F_2
SUBD F_2 F_4 F_2

	0/1

when Branch is Not taken then otherwise.

using branch prediction techniques 3rd cycle is issued.

	IS	EX	W	C
1	✓12	13	14	
2	13		15	
3	✓6	7		
4	7	8		
15				
16				

	I	D	V	S	Header
ROB0	M	F_2	$\neq_{0-0}$	0	H
ROB1	B	F_2	-	0	
ROB2	A	F_4	$\neq_{4-0}$	0	
ROB3	S	F_5	$\neq_{2-0}$	0	

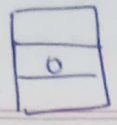
	OP	OP1	OP2	ROBE
RS1	MULTD	0-0	1-0	ROB0
RS2	ADDD	3-0	1-0	ROB2
RS3	SUBD	3-0	1-0	ROB3

F_0	0-0	
F_1	1-0	
F_2	2-0	ROB0
F_3	3-0	
F_4	4-0	ROB2
F_5	5-0	ROB3

The real instructions will be known after execution of branch inst. so ADD & SUB inst are called speculating inst because these inst are issued before knowing the outcome

The real outcome of branch is known at 13th clock cycle but real outcome shows taken. Even though the ADD & SUB inst. are executed but their values are not updated so flush down these inst & change value in buffer from 0→1

Suppose that the prediction is NT & real is NT so the values are modified



Prediction = NT so execute inst 1

Tomasulo's Algo. for Branch & Exceptions

Eg:
MULT D
BNEQ
ADDD
SUBD
DIVD → This inst is speculative and also generates exception (divide by 0)

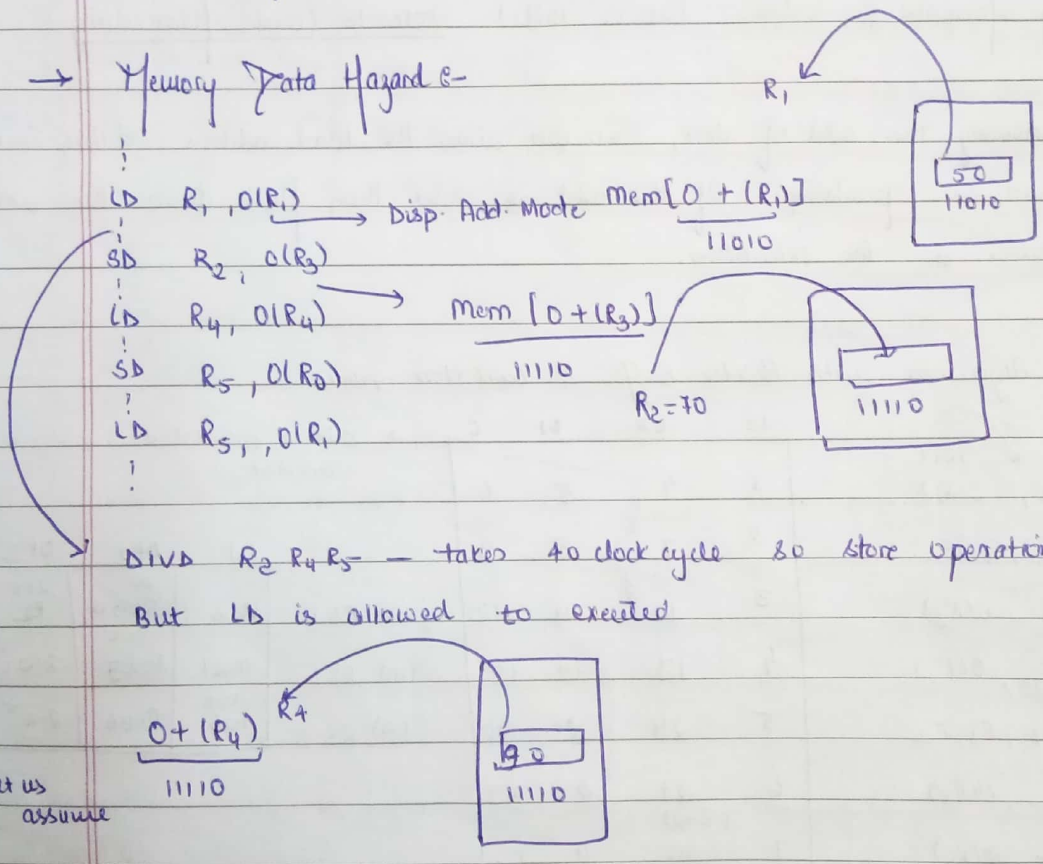
I, (NT)

Label: ADD D
SUB D

If branch outcome says NT then only the exception is handled otherwise the instructions are flushed down when the real outcomes comes.

→ Inst. speculative → Not Handled
Not Speculative → Exception Handled.

→ Memory Data Hazard :-



RAW Hazard b/w memory locations so DIVD operation give wrong answer.

→ LSQ (Load Store Queue)

L/S	EA	V	C
L	1024	~~15~~	
S	—	70	
L	2048	—	
S	-	90	
L	2048		

Speculative load

① LD R_1, $0(R_1)$
② SD R_2, $0(R_3)$
③ LD R_4, $0(R_4)$
④ SD R_5, $0(R_6)$
⑤ LD R_5, $0(R_1)$

15
1024

Non speculative load technique

If effective add. of recent SD & LD is same so no need to go for memory cal. simply forward the value. This is called Store to Load forwarding.

Before knowing the add. of store, we can allow the load address & this inst. is known as speculative. If the add. is same then flush down the values, & otherwise do the calculation.

→ Tomasulo's Algorithm with Reorder Buffer & Load Store Queue

Eg =1

LD	F_0,	$0(R_1)$
ADDD	F_1,	F_0, F_2
SD	F_1,	$0(R_2)$
LD	F_3,	$0(R_2)$
MULTD	F_4,	F_3, F_2
SD	F_4,	$0(R_3)$
LD	F_5,	$0(R_1)$
SUBD	F_6,	F_5, F_2

10 → 1024 (LSQ)

	IS	EX	WR	C
1		4	5	6
2		7 (waiting)	8	9
3		10 (waiting)	11	12
4		13	14	15
5		24	25	26
6		27 (8+2)	28	29
7		10	11	12
8				

→ add. is calculated & value is also obtained.

	OP	OP1	OP2	ROB#
(A) RS1	ADD	10 ~~ROB0~~	2.0 ~~F2~~	ROB1
(M) RS2	MULT	50.0 ~~ROB5~~	2.0	ROB4
(A) RS3	ADD ~~SUB~~	ROB6	2.0	ROB7

ROB

T	~~ROB~~ D
ROB0	L F0
ROB1	A F1
ROB2	S F1
ROB3	L F3
ROB4	M F4
ROB5	S F4
ROB6	L F5
ROB7	S F6

ROB

	T	D	V	S	
ROB0	L	F_0	t_{10}	0	H
ROB1	A	F_1	-	0	
ROB2	S	F_1	-	0	
ROB3	L	F_3	-	0	
ROB4	M	F_4	-	0	
ROB5	S	F_4	-	0	
ROB6	L	F_5	-	0	
ROB7	S	F_6	-	0	

LSQ

L/S	EA	V	C	
L	t_{1024}	t_{10}	0	H
S	t_{2048}	t_{50}	0	> Non spec.
L	t_{2048}	t_{50}	0	
S	t_{4096}	-	0	> Spec.
L	t_{4096}	-	0	

F_0	0.0	ROB0
F_1	~~1.0~~ 12.0	ROB1
F_2	2.0	
F_3	~~3.0~~ 50.0	ROB3
F_4	~~4.0~~ 100.0	ROB4
F_5	5.0	ROB6
F_6	6.0	ROB7

Let us take that the store 1 is not yet calculated the effective add. & load 2 is done with effective add. Assume that the effective add. of LD 2 & SD 1 is same.

Non Speculation technique - LD wait until SD completes its calculation of EA

Speculative technique - Before knowing the add. of store, we can fetch the value in load operation. After calculation of store, if the value of EA is ~~not~~ same that of load then flush down the value of in LD & all inst. following LD inst is flushed down.

→ Tomasulo's Algorithm + ROB + Multi-issive processor

Eg 1 2 issue processor.

				IS	EX	W	C	
SUBD	F_0	F_1	F_2	1	4	5	6	
MULTD	F_3	F_0	F_4	1	15	16	17	
ADDD	F_5	F_1	F_4	2	5	6	~~7~~	18
DIVD	F_2	F_1	F_2	2	43	44	~~45~~	45
ADDD	F_6	F_5	F_2	3	46	47	48	
MULTD	F_3	F_5	F_1	3	16	17	18	

Check for functional Units

2 issue

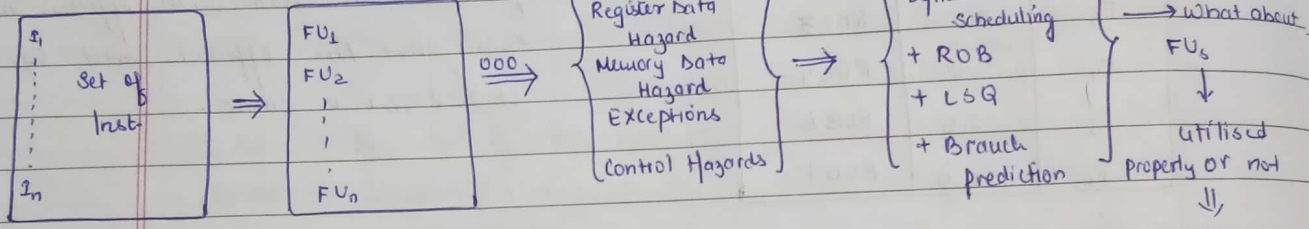
	IS	EX	W	C
1'	4	5	6	
1	15	16	17	
2	43	44	45	
2	45	-	46	
3	6	7	47	
3	17	18	48	
4	9	10	49	

Eg
SUBD F_0 F_1 F_2
MULT F_3 F_0 F_4
DIVD F_2 F_1 F_2
BNE F_2, 0, label
ADDD F_5, F_1, F_4 } speculative so can't complete
MULTD F_3, F_5, F_1
ADDD F_6, F_5, F_1
label :

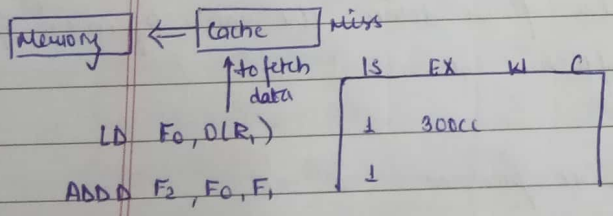
BNE → NT

If taken then remove the values of ADDI & MULTD

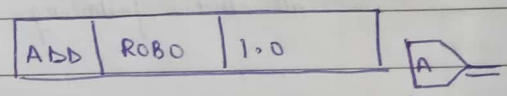
→ Simultaneous Multithreading



(Is) Issue means occupying the reservation stations



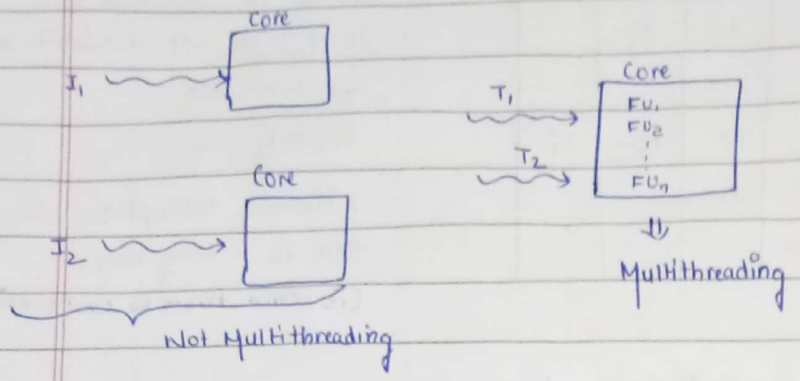
Single thread



wait for 300 cc to execute
so ALU remain ideal for 300cc

Multithreading :-

i7-8550 - 6 cores
12 threads



It allows multiple threads share functional units of a single processor.

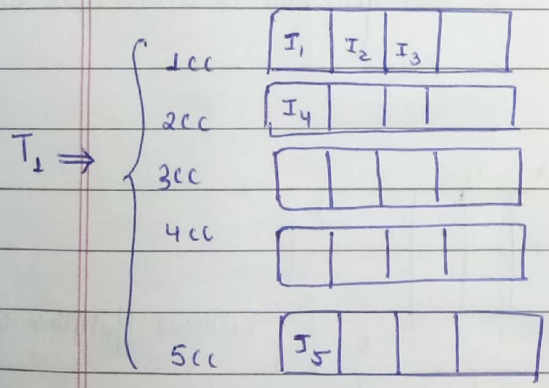
Multithreading

- Superscaler processor without multithreading
- Coarse grain multithreading
- Fine-grain multithreading
- Simultaneous multithreading (SMT)

} All are Superscaler - issuing multiple inst in single processor

Single-Core :- Max^m 4 issue (ie only 4 inst. in single clock cycle)

1) Superscaler processor :-



- I_4 is dependent on I_1
- I_5 having cache miss (so takes 2 extra clock cycles)
- only 5 slots out of 20 are used remaining are ideal

T_1
I_1
I_2
I_3
I_4
I_5

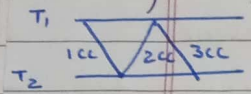
- All the inst. belonging of thread 1 is executed first. then new thread starts.

2) Coarse Grain multithreading :-

	1cc	I_1	I_2	I_3	
T_1	2cc	I_4			
T_2'	3cc	I_1'	I_2'	I_3'	I_4'
	4cc	I_5'	I_6'		
	5cc	I_5			

- I_5 of T_1 is Not available due to Cache Miss.
Similarly I_7'
- Switching takes place when there is a long time (i.e when there is cache miss)

3) Fine grain Multithreading :- Switching takes place at every clock cycle.



	1cc	I_1	I_2	I_3	
	2cc	I_1'	I_2'	I_3'	I_4'
T_1	3cc	I_1	I_2	I_3	I_4
T_2	4cc	I_5	I_6	I_7	
T_3	5cc	I_5'	I_6'	I_7'	

- Irrespective of dependency, threads are issued.
- Horizontal slots are completely used but not vertical slots.

4) Simultaneous Multithreading :-

1cc	I_1	I_2	I_3	I_1'
2cc	I_4	I_2'	I_3'	I_4'
3cc	I_5'	I_6'	I_5	
4cc				
5cc				

- Single clock cycles contain Inst. from diff. threads.
- Both Horizontal & vertical slots are used.

Eg

T_1

LD	$F_0, 0(R_1)$
ADDD	F_2, F_0, F_1

T_2

ADDD	F_4	F_5	F_6

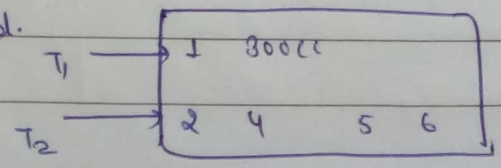
IS	EX	WC
1	300cc	
1		

RS

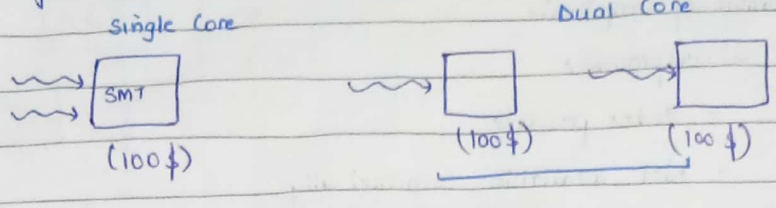
ADD	R0B0	1-0

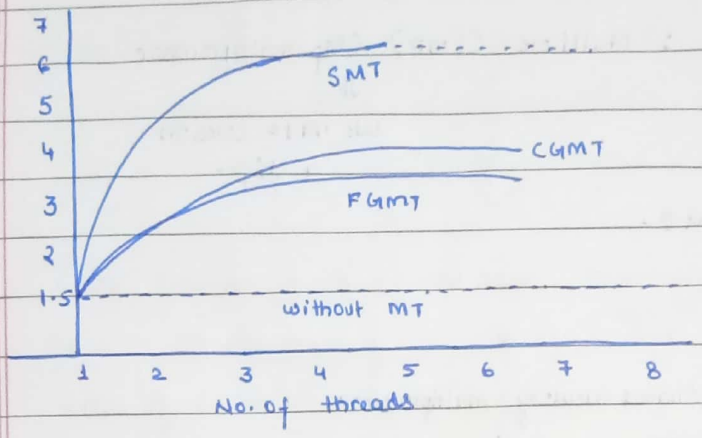
} Without Multithreading

In Simultaneous multithreading, it flushes the inst. present in RS sitting ideal & uses inst. from next thread.



Single core SMT has almost same performance as multiple cores without multithreading & ~~although the~~ SMT is less expensive





IPC = 6 (SMT)

IPC = 1.5 (without MT)

After a certain no. of IPC, even though the no. of threads increases ⇒ SMT remains constant as only a certain IPC can be handled at a time after using ~~Multithreading~~

Thread level Parallelism :- TLP → ILP

2 inst. can't be executed at the same time using one core. One inst. executes & then the next one. To achieve TLP, more than one core should be ~~then~~ there. More than one core will help in executing more instructions at a time.

Single core v/s Multicore

1cc 1cc 1cc 1cc 1cc

IF ID EX M WB = 5cc

perform all these 5 functions in single clock cycle

For single core -

↳ - If frequency ↑ → voltage ↑ → $P \uparrow^3$ $[P \propto V^2 f]$

cannot increase its capacity

as a high power system → break down

In multicore, more than one thread will execute at same time

Flynn classification of computers:-

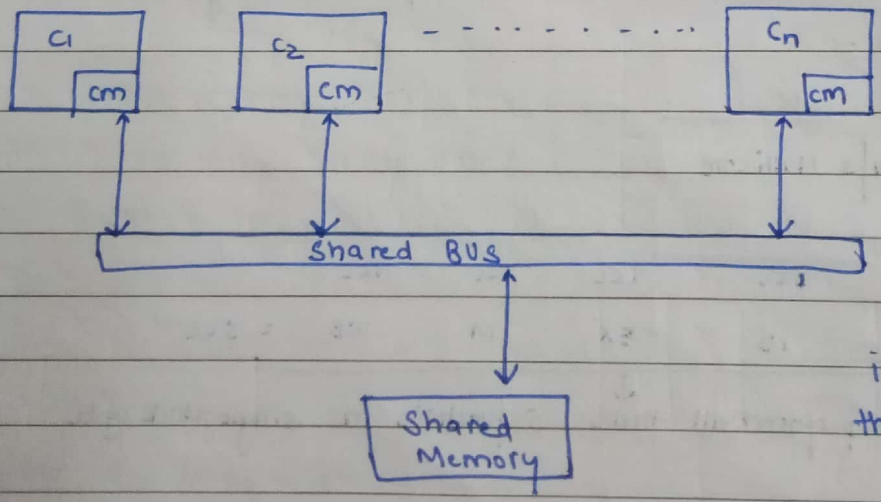
- SISD - uniprocessor
- SIMD - Vector processor
- MISD - Not available commercially
- MIMD - Multiprocessor
  - Eg: Multicore (CMP) - Chip multiprocessor
    - ⇓
    - We have cores on a chip.

Thread level parallelism :-

MIMD

- → Centralised shared memory multiprocessor
- → distributed memory multiprocessor.

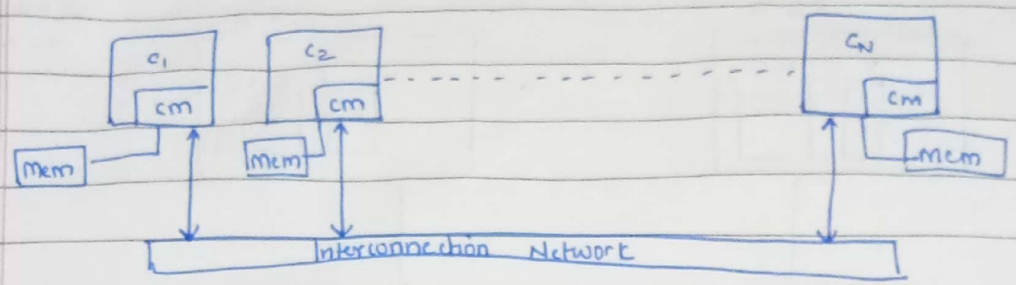
We are switching in Multicore as in single core even upon using SMT, performance became stagnant at certain IPC even though no. of threads is increased.



Objective:-
CPI < 1
⇓
ie to execute more than 1 inst per clock cycle.

Here if each core executes 1 inst then CPI = 1/n

→ Distributed Memory Multiprocessor



All these above form can form a cluster

- Direction Based problem

→ Cache Coherence problem :- Let Shared memory has X=1 . Let C_1CM_1 & C_2CM_2 reads X from memory & places it in ~~its~~ their own cache. When C_2 write the value of Cache from X=1 → 2 , value only changes in its cache, all other will use the previous value of X. This problem is known as Cache Coherence problem.

Incoherent Support :- When C_2 modifies the value of X but Shared memory keeps on sending X=1 to other cores.

Cache

→ Write through Cache

When a core makes a change to X, it is propagated to all the cores & Shared memory.

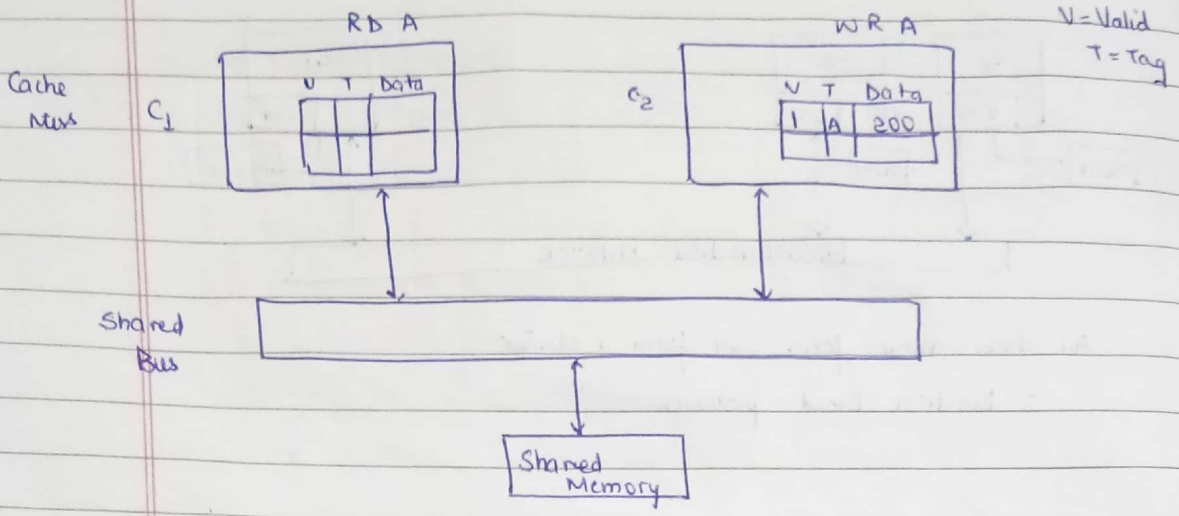
→ Write Back Cache

When a core makes a change to X, it is propagated only when X is removed from the core.

To solve Incoherent Support

- Write - Invalidate with Snoop Bus (DBP)
- Write Update with Snoop Bus

Cache Coherence Problem :-



When the block is not available in cache = Cache Miss

Let C_1 = Cache Miss

$\therefore$ V=1, T=A, Data=100

Let C_2 want to write A $\therefore$ write miss (A not available in cache)

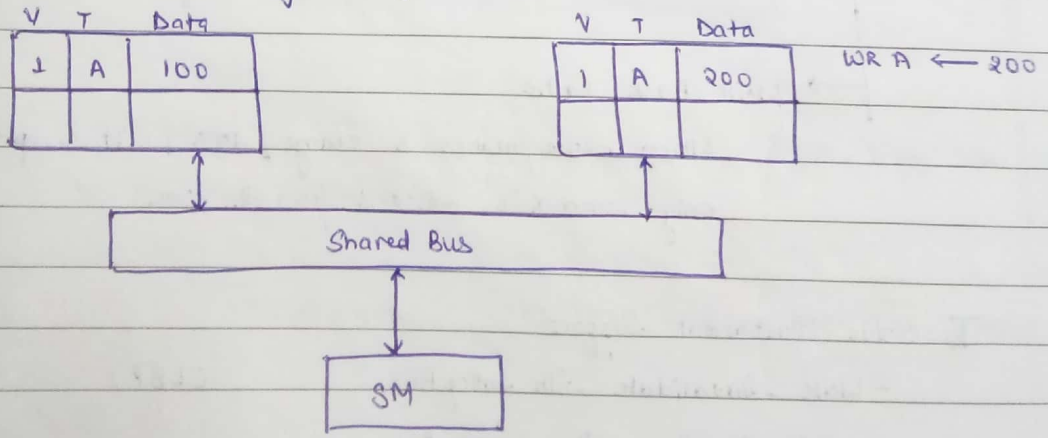
So read A & change A $\rightarrow$ 200

So the value of A in both the caches are different - this problem is called cache coherence problem.

(1) Write update using Snoopy Bus
(2) Write Invalidate using Snoopy Bus
(3) Write Update using Directory Based
(4) Write Invalidate using Directory Based

} Soln of Cache-Coherence problem.

1) RD-A Write Update Using Snoopy Bus



When Core 2 writes A, it sends a message regarding the change in value. While message passing, Core 1 will check the change in the Bus - Called Snoopy Bus & Changes its Value.
If the Core 1 does not have A in Cache then none of the block will take the value. Only SM will change the value.

Problem - i) Memory bus handling.
ii) If A is modified 10^5 times then SM is also modified 10^5 times
iii)

At one time, only one bus is present so if 2 cache simultaneously changes the value of A then which one is allowed?
The cache which first access the bus, will can change the value of block.

→ How to minimize the Use of Bus & Shared Memory ?

RDA

V	T	D	Data
1	A	0	100
1	B	1	200

V	T	D	Data
1	A	1	100
1	B	0	200

D = Dirty bit

WR A ← 100

wants to update B → 300

∴ D = 1 & update the value in Cache.

Shared Bus

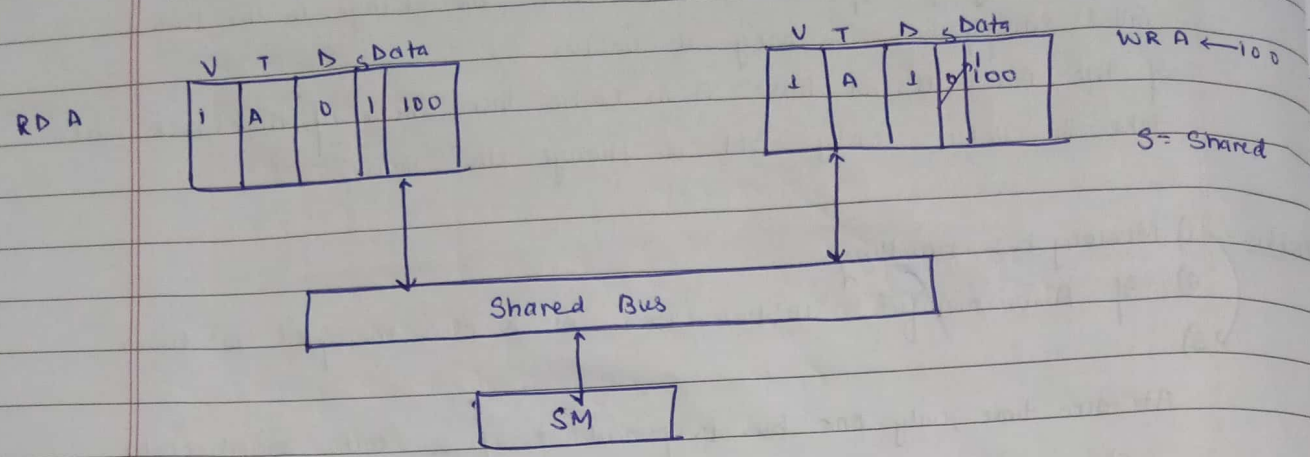
SM

Write back Cache

Core Cache will keep the latest value of block in in its Cache only. It will not propagate the value in Bus.

If D = 1, then SM will not allow the change in SM.
If Core 1 request to read A, Core 2 will supply as Cache speed is greater than memory speed. So A = 100 in Core 1. & D = 0 for Core 1.
When A block is removed from Cache 2, ~~it is updated with the value in SM~~ Value of A is updates in SM.

→ How to minimize the access of Shared Bus ?



How the cache will know that the block is shared or not -

→ when the cache is writing, it will check whether the block is is shared or Not.

If Not Shared ; S = 0

If Cache 1 wants to read A ∴ Cache 2 will supply the value of A

∴ S = 1 in both Cache

When Cache writes a shared block, it broadcast the value on the bus.

If the block is removed from Cache, S=0 again & no broadcasting takes place.

Eg =1

CORE 0 CORE 1

WR A

1000 times

RD A

1) How many Bus Access
2) How many Memory writes

] No optimization / D / D & S.

Soln

1) No optimization

first cycle

Memory writes = 1 + 999 = 1000

Bus Access = 1 + 1 + 999 = 1001

for reading A from SM

2) Using D bits

If A is replaced

Bus Access = 1 + 1 + 999 + 1 = 1002

Memory writes = 1

↳ when A block is removed

first cycle, P sets D = 1

3) Using D & S bits

Bus Access = 1 + 999 + 1 = 1001

Memory Writes = ~~1~~ 1

Eg

CORE 0

RDA } 500 times

WR A

CORE 1

RDA } 500 → Memory supply data in No optimization method

WR A

	No opti	D	D & S
Bus Access	1+1+1+1+1+499 = 1002	1+500+1+500 = 1002	1+1+500 = 502 ✓
Memory writes	1+499+499+1 = 1000	~~1~~ 0	0

A is NOT removed from core

2) Write Invalidate Using Snoopy Bus.

RD-A

V	T	D	S	Data
1	A	0	0	100

V	T	D	S	Data

WR A → 300

Shared Bus

INV(A)

SM

When Core 2 want to write A, it will send a invalidate message to Bus. Since D=1 so no change in SM but in Core 1 ~~V=0~~ V becomes 0 which means U can't access A because someone is ~~is~~ writing A

Core 0 — V=0, T=A, D=0, S=1

Core 1 — V=1, T=A, D=1, S=1, Data = 300

If Core 1 wants to update 1000 times, in case of sharing for ~~it~~ it can write 1000 times broadcasting the write message.

After updation, it sends message & changes V=0 ⟶ V=1

Eg-1 ⟶

	Write Invalidate (D&S)	Write Update D&S
Bus Access	1+1 + 999 + 999 = 2000+1	1002
Memory writes	0 1	0 1 ⟶ replaced

Sends a invalid message for reading

Eg-2

	Invalidate (D&S)	Update D&S
Bus Access	1+1+1+1 = 4	502
Memory writes	0	0

Reading for 2nd core

Invalidate message for first cycle

Since No one is accessing ~~the~~ V.A so

Eg

Core 0 — WRA

Core 1 — RPA } 1000 times

Write Update Snoopy Bus.

	Replace A No optimization	Replace A D	D & S
Bus Access	1+1+999 = 1001	1+1+999+1 = 1002	1+1+999+1 = 1002
Memory writes	1 + 999 = 1000	1	1

- Replace A in No optimization not require the bus as it is updating the value time to time
- Reading not require the Bus access in No optimization & D because updation is done ~~as~~ time to time.
- When the writing process takes place, it doesn't know that the ~~Bus~~ Block is shared or not so it broadcasts the message ~~everytime~~ first time.
As reading process happen, ~~it~~ ~~knows the~~ shared bit = 1

	Write Update (D&S)	Write Invalidate (D&S)
Bus Access	1002	1 + 1 + 999 + 999 + 1
Memory Write	1	1

to get right data from core 0

Eg-2 Core 0 Core 1 (Without Replacing A)

RD A
WR A } 500

RD A
WR A } 500

Write Update →	No opti	D	D&S
Bus Access	1+500 + 1+500	1+500+1+500	1+1+500
Memory writes	500 + 1+500	0	0

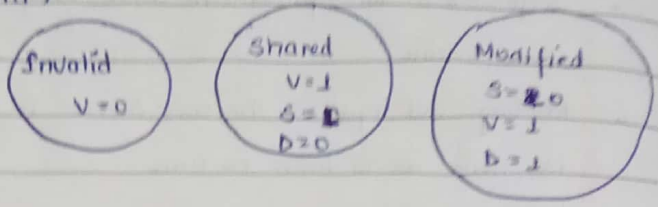
With the help of read operation, Core 1 knows that there is no sharing but the 2nd read ~~not~~ gives the information of sharing of block.

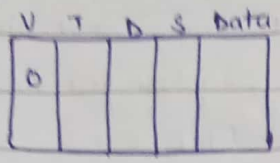
	WU	* WI
Bus Access	502	1+1+1 = 3
Memory writes	0	0

1 = Reading Core 0
1 = ~~to~~ Reading Core 1
1 = ~~to~~ Sending Invalid message to core 0.

→ MSI (Modified Shared Invalid)

- MOSI
- MESI
- MOESI

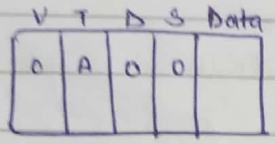




Invalid (V=0)

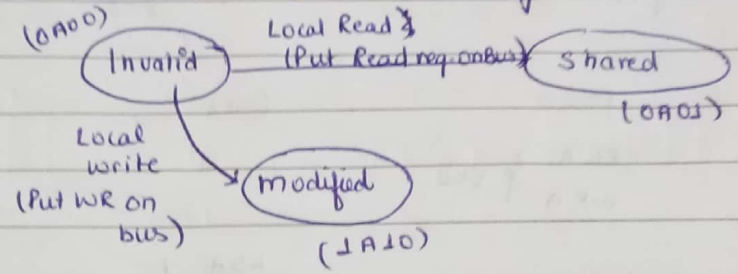
- block is present but can't access (someone is writing)
- Block not present

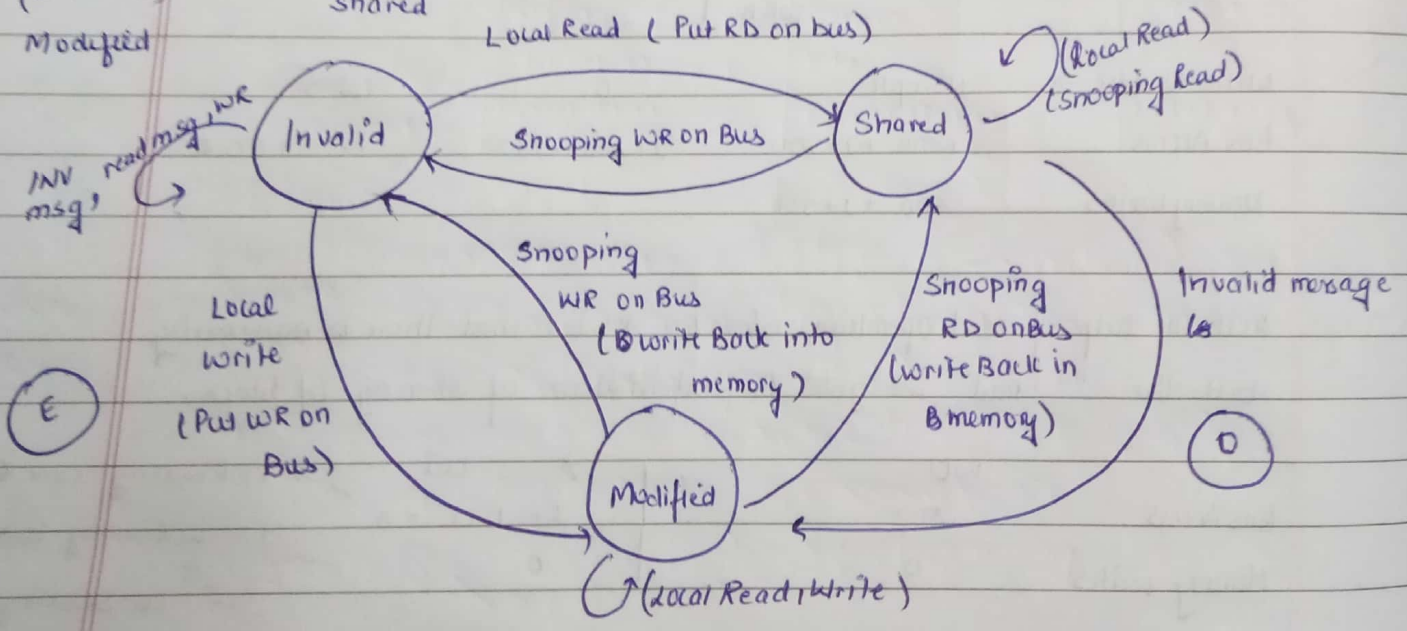
Modified - Only one valid block is present remaining all invalid



Wants to WR A←100
So send msg to Bus
(1 A 1 0)
Modified

RD A
(0 A 0 1)
Shared





Modified → Invalid (first update the value in memory)

Shared → Modified (Send Invalid msg to invalid all the shared blocks)

Eg Block X in main memory

C_1	C_2	State of X in C_1	State of X in C_2
RDX	–	shared	Invalid
–	RDX	shared	shared
WRX		modified	Invalid

Eg

C_1	C_2	State of X in C_1	State of X in C_2
RDX	–	shared	Invalid
–	WRX	Invalid	modified
WRX	–	modified	Invalid

→ How to minimise the Write Back operations ?

(MOSI)

C_1: Block A in M

C_2: Read (A) ⇒ C_1: S | C_2: S (O)

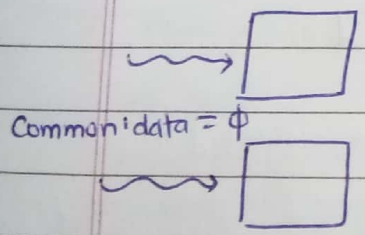
C_3: Read (A) ⇒ C_1: S | C_2: S | C_3: S (Who provide data to C_3)

memory will provide the data when more than 1 cache have same value of A

To avoid problem & memory access, C_1 change in its state from M → O (O = owner) and C_1 will provide the block to C_2

when the block is replaced from block, it will replace the value in memory.

→ MESI :- E = Exclusive



When the block read by core from the memory, No copy of block is ~~not~~ present in any of the core So no need to send message to any core ∴ Exclusive block.

→ MOESI :- Sharing a block, one block become owner & in case of exclusive, one core is exclusive

MOESI | MESI | MOSI | MSI

Eg → MOESI - 3 cores 1 Block X is in memory

C_0	C_1	C_2
RDX		
Exclusive	Invalid	Invalid
	RDX	
Shared	Shared	Invalid
		RDX
Shared	Shared	Shared
	WRX	
Invalid	Modified	Invalid
		RDX
Invalid	Owner	Shared
RDX		
Shared	Owner	Shared

Eg

	MESI (Memory will give data to C_2)	MOSI	MOESI
How many Memory Reads Bus Request	1 + 1 + 1 + 1 = 4	1 +	1
	1 + 1 + 1 + 1 + 1 = 5	1 + 1 + 1 + 1 + 1 + 1 = 6	1 + 1 + 1 + 1 + 1 = 5

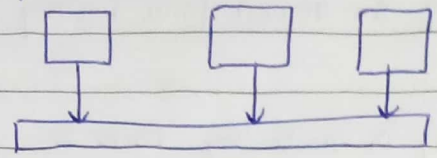
C_1 : RDA
C_1 : WRA
C_2 : RDA
C_2 : WRA
C_3 : RDA
C_4 : RDA
C_2 : RDA

1. Acc. to MESI & MOESI, C_1 → Exclusive but acc to MOSI, C_1 → Shared so send a message to bus
2. C_1 : Modified
3. C_2 : S, C_1 (Acc to 2nd & 3rd) = 0, $C_1^{(1)}$ = S
4. C_2 : Modified, C_1 : ~~Shared~~ Invalid
5. C_3 : S, C_2 : S (1st protocol) → data from memory / 0 (2nd & 3rd)
6. C_1 : S, C_3 : S, C_2 : S / 0
7. C_2 = Owner ∴ No Bus Access.

Reply
forward

→ Directory Based Cache Coherence Techniques

In case of snoopy bus →



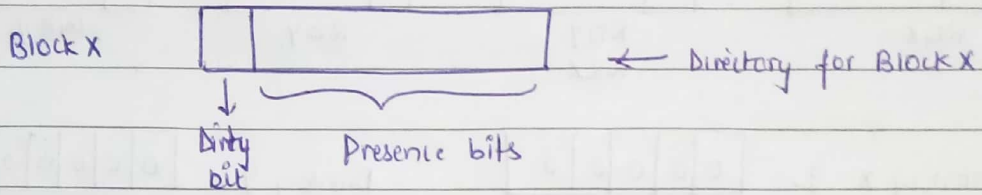
> 8-16 Cores

- Not scalable, can't go beyond 4-5 cores

Every read & write message pass through the bus. And there is no relevant means of broadcasting in case of large no. of cores.

But in case of directory-based protocol

For each block X, one directory is created

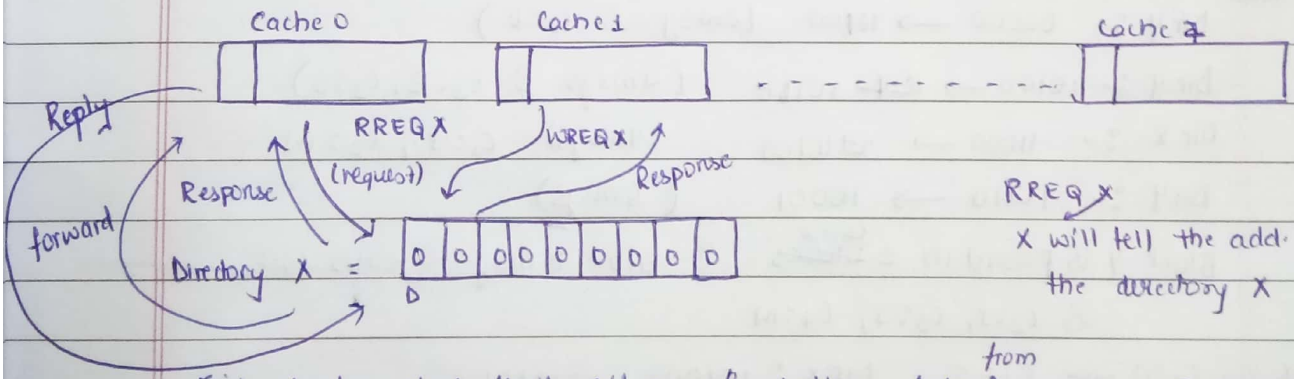


If we have n caches then size of presence bit : 0 ⟶ n-1

block present in Core 1 only 100···

0000··· = Not present in any cache

Let us assume that there are 8 Cache



RREQ X
X will tell the add. of the directory X

First directory check that all 0 ∴ directory fetch from memory & send data to cache 1 ∴ 0000··· → 110000000

& C_1 : Invalid ⟶ Exclusive

so Dirty bit = 1 that means cache 1 can write without any msg

After some time, C_2 wants to WR X ∴ send a message to directory to write X

Directory check that the block is available in cache 1 So it will forward the req. to cache 1. So Cache 1 change E ⟶ I & reply (msg) &

Send response msg to Cache 2

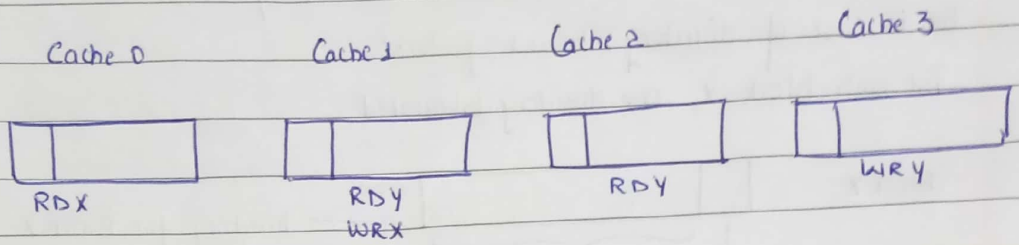
10000000 ⟶ 10100000

Now ~~the~~ Cache 7 wants Block Y so request from Directory Y.

If Instead of writing, reading req. is made by Cache 2

∴ C_1 : E ⟶ S & Directory → 11000000 ⟶ 11100000

Eg ⇒



Directory X :- 0 0 0 0 0 Directory Y :- 0 0 0 0 0

Directory Z :- 0 0 0 0 0

Sol^n

C_1 : RDX ⟶ Dir X :- 00000 ⟶ 11000 (2msg - request & response) for Cache 1) C_1 : E

C_2 : RDY Dir Y :- 00000 ⟶ 10000 (2msg ∴ C_2 : E)

C_3 : RDY Dir Y :- 10100 ⟶ ~~10100~~ 10110 (4msgs & C_2 : S, C_3 : S)

C_2 : WRX Dir X :- 11000 ⟶ ~~11~~ 10100 (4msgs) C_1 : I, C_2 : M

C_3 : WRY Dir Y :- 10110 ⟶ 10001 (6 msgs)

Block Y is present in 2 ~~blocks~~ Cache ∴ send 2 msg to each Cache

∴ C_2 : I, C_3 : I, C_4 : M

C_3 : RDX C_4 : O ⟶ C_3 : S DIR X : 10100 ⟶ 10101

Eg ⇒

C_0 : RDA
C_1 : RDA
C_2 : RDA
C_3 : RDA
C_0 : WRA

1) How many req. to directory
2) How many forward msg.
3) How many reply msg.
4) How many response msg.

Request	Forward	reply	Response
1+1+1+1	1 ~~+0+0~~ +3	1 ~~+0+0~~ +3	1+1+1+1
+1	= ~~4~~	= ~~4~~	= 45
= 5			

Soln

00000 ⟶ ~~10~~ 11000

11000 ⟶ 11100

11100 ⟶ 11110

11110 ⟶ ~~11111~~ 11111

11111 ⟶ 10000

when cache want to ~~write~~ read a shared data ⇒ No forwarding