High Performance Computing

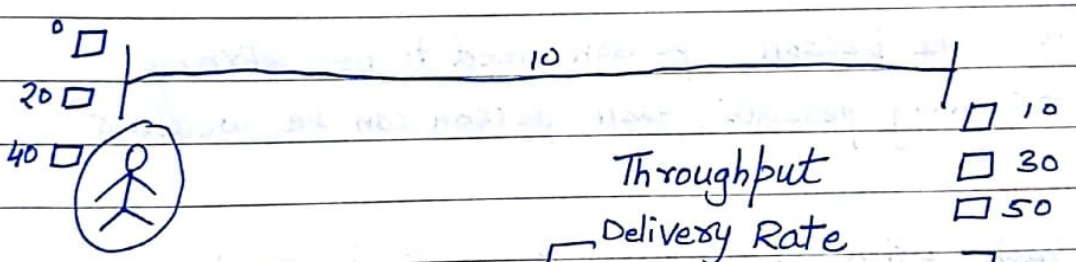
CAO → Sequential Computer → Instr" goes through instr" cycle & then only next instr" is fetched.

Disadv: Lot of h/w units in idle state while others working

Performance Enhancements → Pipelined Execution.
→ Parallel Execution

Sequential Processor → Software (Parallel Software)
→ Hardware growth towards a Parallel system design
↓ Enhanced
Pipelined Processor
↓ Parallelism
Parallel processor System

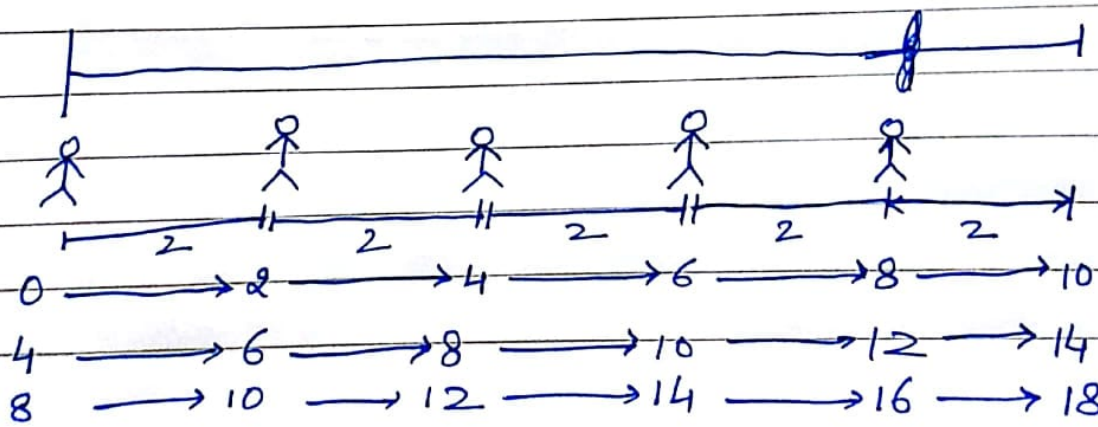
- had downsides also
∴ not much continued & 11 system came.



Throughput
Delivery Rate

Brick Delivery Rate = [1 brick every 20 units of Time]

Improvement → Employ multiple resources (people here)



There are sufficient people so that when a brick reaches, the person is free (Consider if 3rd person is not present! At t=6, person 2 not free)

→ Pipelined execution.

[Delivery Rate
1 brick every
4 unit of time]

This is analogous to pipelined processor.

We divide same H/w into independent stages & one ~~person~~ stage passes the result to other stage on completion of work.

If no work for stage 1, idle.

Initially, some stages are idle, but later, all stages work.

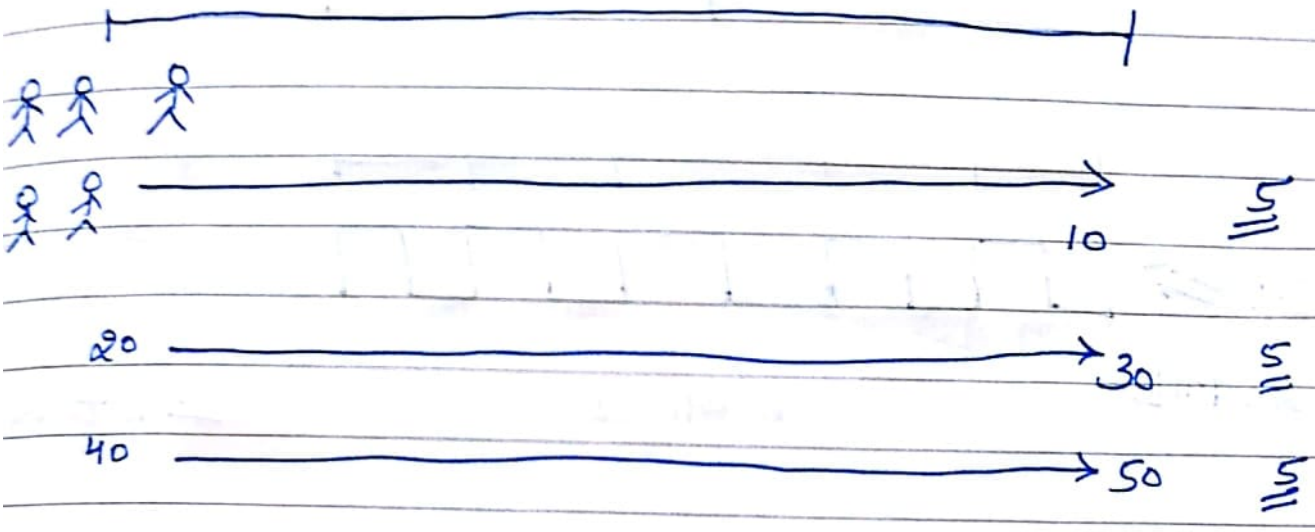
for one person, person need to be strong
for many persons, each person can be weaker.

Each brick still takes 10 units to reach other side.

So work time for each i/p is same but i/p rate is↑

Latency → Amount of work done for one i/p
↳ remains same.

Parallel rss available at starting point.



Still Throughput is 4 bricks / 20 units i.e 1bricks/4units time.

But rss need to be stronger.

If 1 Sequential costs 100

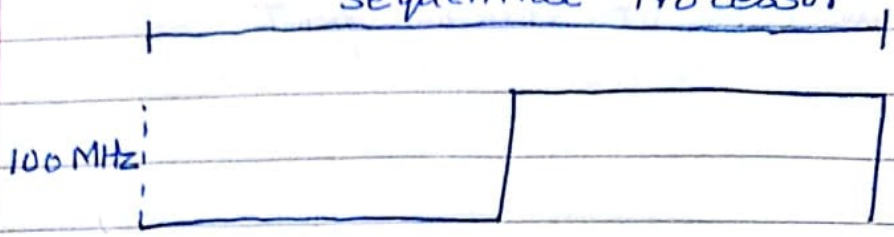
Pipelining doesnt increase cost much 120
(only latches to separate stages)

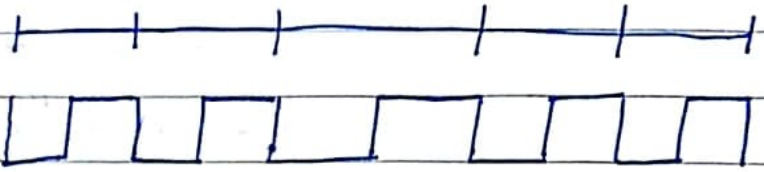
Parallelism → costlier.

Pipelined → gives equivalent results but has a demerit

20 unit

Sequential Processor





Clock freq ⇒ 500 MHz

~ 5 times

10 stages

≃ 10 times.

Clock freq > 1000 MHz

Made of CMOS transistor

↳ No power when in stable state only during transition.

More transitions → more switching → Power Diss ↑

↓

Ultimately Heat will burn the CMOS transistor

Dicadv. → Clock freq ↑ → heat ↑

~~Bottlenecks~~. → Pipelining overheads → Latch Delay decides min. delay.

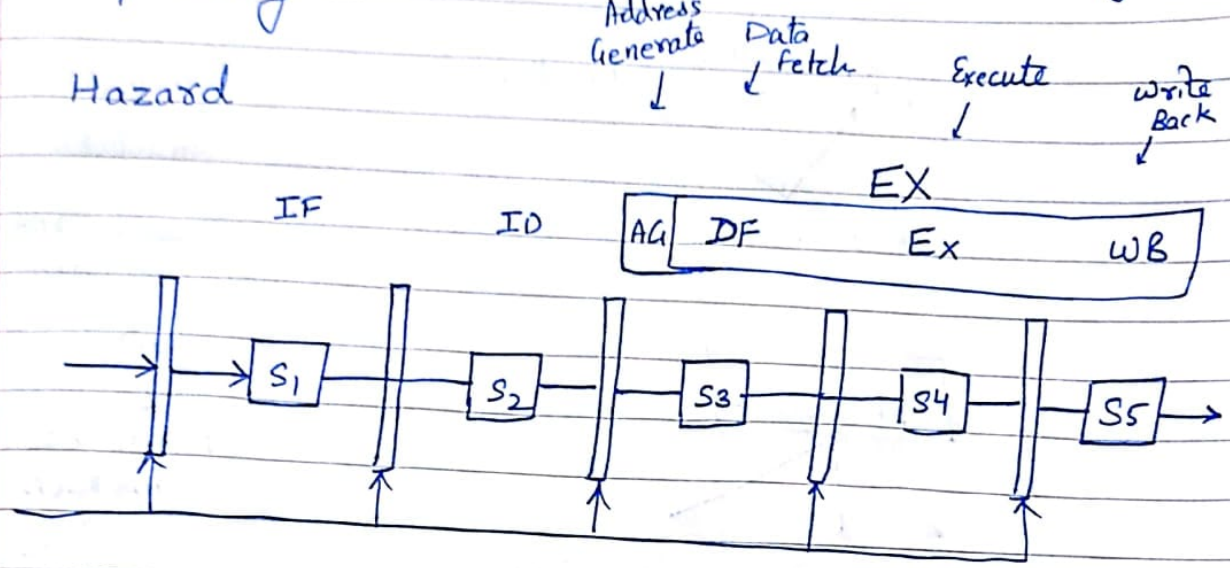
At a pt. Latch Delay becomes significant to stage Delay. After that no more enhancements.

Battle necks

S/W aspect → Dependency

H/W aspect → Hazard

Fetch → for Memory Not registers



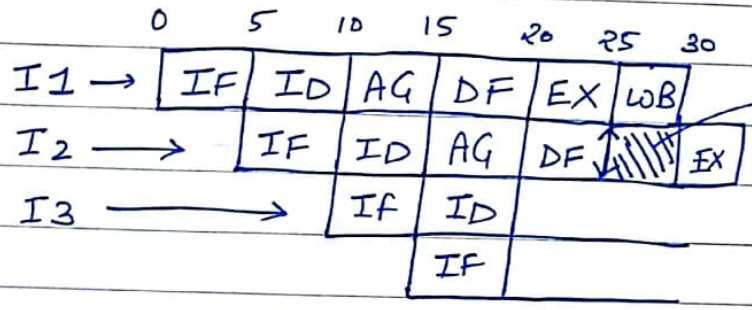
$(r1) \leftarrow r2 + M[10 + r3]$

↳ r1 is written to only at stage 5 (write Back)

$r4 \leftarrow r1 - r5$

add r1, r2, 10(r3)

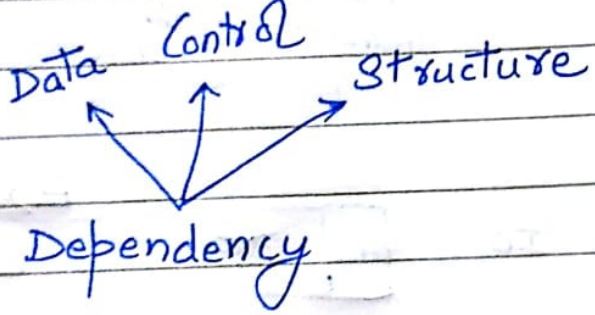
sub r4, r1, r5

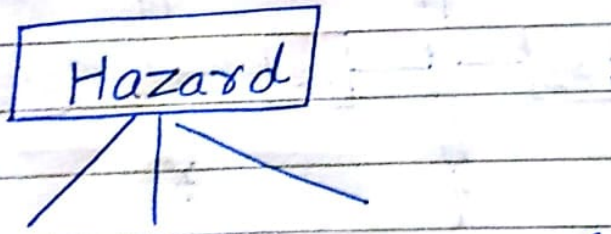


Execution of I2 requires r1's value at 25. Logically correct value only available at 30 ∴ idle from 25 to 30.

Pipeline Stall / Stall Cycle / Hazard

eason of Hazard → Dependency (r1 common)
if r7 instead of r1, no stall





structural
↓
same resource required.

Control
↓
Due to branch instructions
↙ ↘
Conditional: Action depends on cond^n (flag)
unconditional: Jump, Call, return, Interrupt.

Hazard → Dependencies { Data → True / False; Control; Structural }

Max. Adv. of pipelining is restricted by dependencies

False { Anti Dependency; Output Dependency } — There is no data flow from one instruction to another; only a name is involved.

True → Data flows from one instruction to another

I_1 — add Y_1, Y_2, Y_3

I_2 — sub Y_4, Y_5, Y_1

If value reqd is not available → Pipeline Hazard / Pipeline Stall

If dependencies resulting in stall → Hazard.

WAW (write after write)

add Y_1, Y_2, Y_3

sub Y_1, Y_4, Y_5

Dependency is there but no data flow

↳ Eg of output dependency.

Anti Dependency

WAR (Write after Read)

add Y_1, Y_2, Y_3

sub Y_3, Y_4, Y_5

RAW

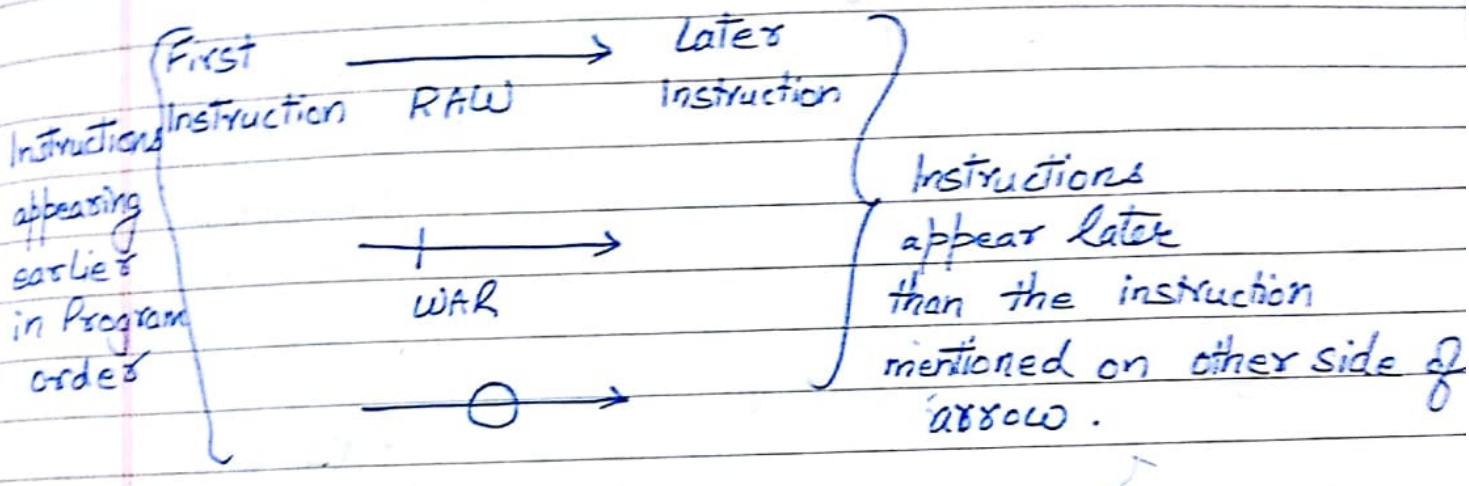
Read after ~~Read~~ Write

I_{10} —— add R_1, R_2, R_3

I_{15} —— sub R_4, R_5, R_1

Dependencies shown pictorially in a dep. Graph.

Dependency Graph



Name dependency come in picture when Out of Order Execution is allowed

↓ To increase efficiency.

Technique to handle these:

Register Renaming

Normal Pipeline processor → No false dep. → But not max. efficiency.

↓

Super Scalar Processor (Multiple Pipelined Processor)

4 Pipelines can fetch 4 instr^n's at a time.
↳ in H/W, ~~Pipelining~~ Instr^n fetch H/W fetches 4 bytes at a time

WAW add R1, R2, R3
sub R11, R4, R5 ← renamed.

All occ. after this of R1 replaced by R11
→ No dependency.

Can be implemented by H/W or S/W → Compiler. (Superscalar)
→ VLIW
(Very Large Instr^n Word

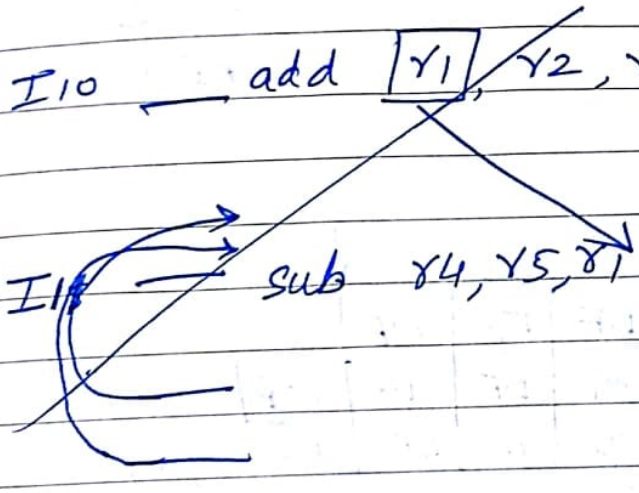
Pentium → 8S Processors
486 → Deep Pipelined "

Superscalar ← Hardware
Register Renaming
VLIW → Software

↳ can be used for true dependencies as there is actual data flow.

In True Dep ⟶ Instr^n Reordering.

Place independent instr^{n}s. b/w the dependent instr^{n}s.



RAW

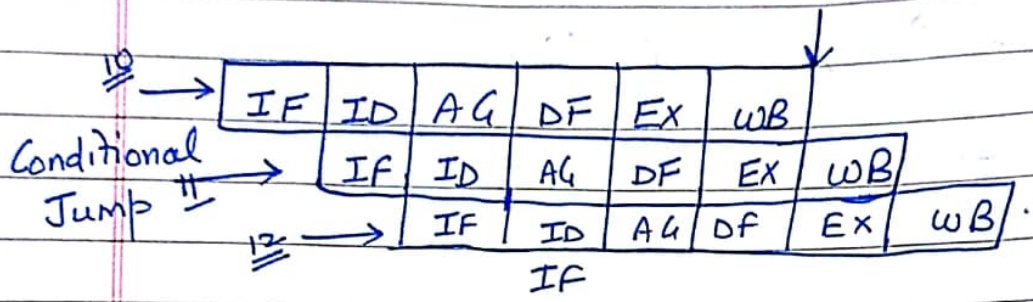
I10 add R1, R2, R3

I11 sub R4, R5, R1

If data is generated by ALU & reqd by ALU, directly provide a bypass path instead of involving Register file.

Control

→ Jump → Conditional — JNZ, JZ
→ Jump → Unconditional → JMP, CALL, RET



AGAIN: —
5 —
—
—
—

Flag → Same
Condition Code

10 → DCR C
11 → JNZ AGAIN
12 → —
—
—
—
—

If we know that we have to go to target (5) after EX, All work done will be dumped & pipeline will be started from beginning.

will study to minimize this. ← cycles in which we did not get data → Control Hazard cycles.

Unconditional → Less Penalty (after ID we know)
Conditional → More Penalty (we know after EX of arith. instr^n.)

→ Branch Elimination — By replacing Branch instr^{n}s by Cond^{n}al Arithmetic logical Instr^n.

→ Branch speed up → Early CC Set
→ Delayed Branch

→ Branch Prediction

Done by compiler → Insert indep. instr^{n}s b/w Arith. & Jump instr^n.

DB → Insert Indep. instr^{n}s after Jump which will be run irrespective of Jump / Not → Better for Unconditional Jump.

? Try to predict outcome of branch & instr^{n}s in predicted direction will run without penalty

Branch Prediction

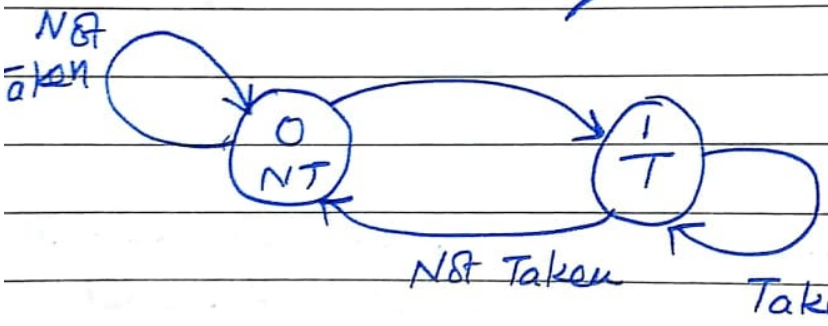
Branch History Bits
B.H.T — Branch History Table

Date
Page Branch will

Single bit History → 0 → Branch not Taken (not go to target)

1 → " Taken (will go to target)

→ Taken.



name.

HPC

lib) m(.so shared object → depends on external libraries

lib)m(. a collection of object files → full fledged executable
archive files.

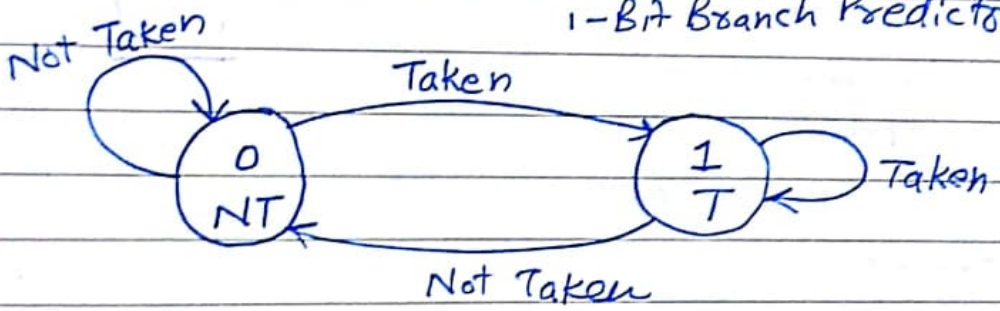
Branch Prediction

ARM

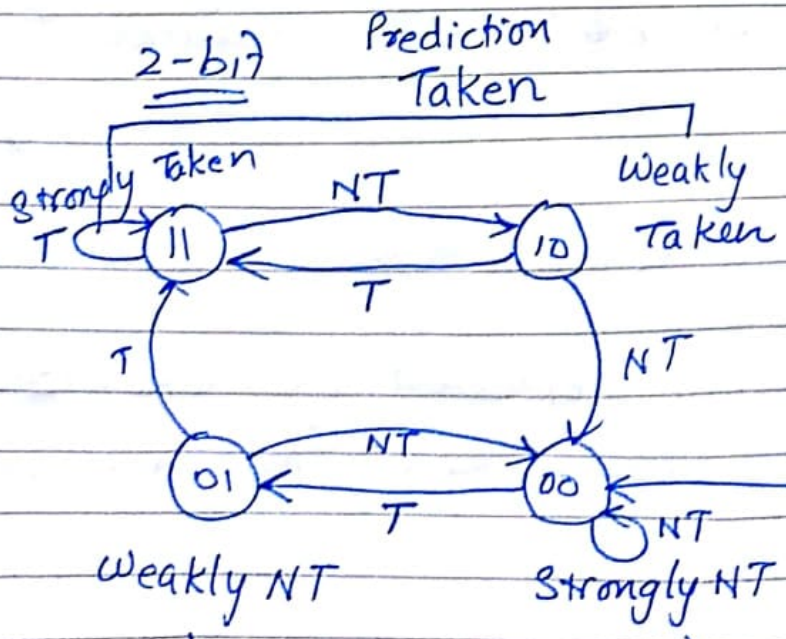
MIPS

SPIM → Allows. to chose b/w Delayed Branch, etc.
(Simulator)

1-Bit Branch Predictor.



2-bit Prediction



In some cases
↓
3 Taken to go to Strongly Taken

C=10

AGAIN:

DCR C

JNZ AGAIN

Prediction Action

NT	T
NT	T
T	T
:	
T	NT

3 Wrong Predictions

If 1 bit → 2 Wrong Predictions

Each Branch has separate Predictor.

1-bit

NT-T
T-T
:
T-NT

C=10

AGAIN1:

AGAIN2:

DCR C
JNZ AGAIN2

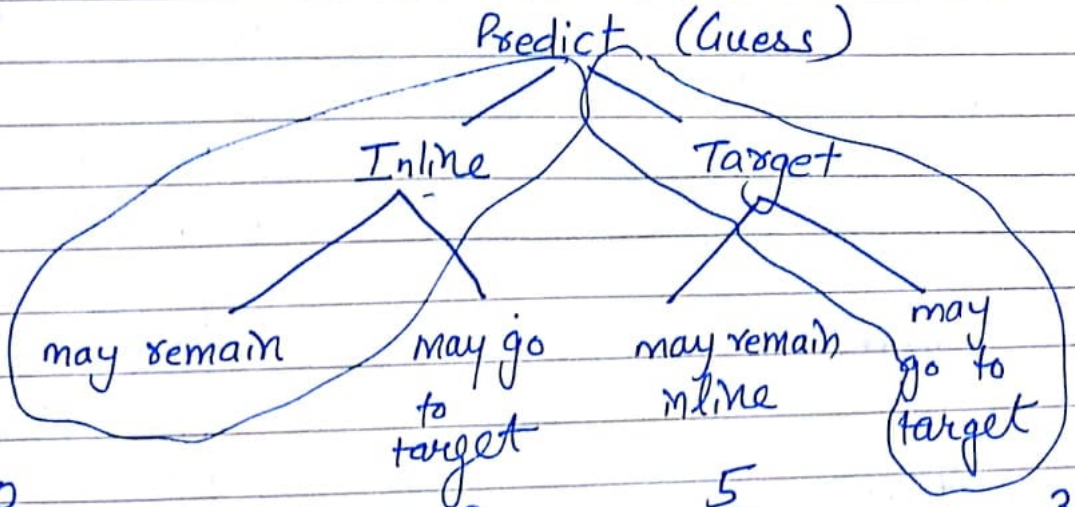
DCR B
JNZ AGAIN1

Similarly for outer

If outer 10 times & inner 10

Inner Mispredictions → 20

10 → 8 times.



0

3

5

3

Cycle Penalty

non-zero penalty, say 3

∴ always better to predict inline

∵ at start → inline

AIM → improve this penalty → by improving predictions

BTAC → Branch Target Address Cache

BTIC → B

B·H·T

Branch Address	History Bits

or if in 1-bit predictor, we can remove history bits & detect it by presence of address.

Branch Address	History Bit	Target Instruction Address

ADD R1, 10 (R2)

R1 ← R1 + M [10 + R2]

In case of Branch, DF will be Target Instrⁿ Fetch
Target Address is not available before D·F ∴ It has to wait till that ∴ some penalty even if goes to target & predicted target.

∴ Target address stored in B·HT ⇒ 0 cycle Branch

wireshark
Ethereal
Packet Analyser
Capture

pcap library → PCAP file

MAC Address ↕ Ethernet Address.

ATM Cell Switching

HPC

Branch Prediction

B.H.T

Branch Instr^n Address	Branch History Bits	Branch Target Address

Correlation Branch Predictors

⇓

Captures the branches that are data Dependent

(If one branch taken, other is also taken → this type of predictor)

or in dedicated appl^n specific instr^n processor like Router

Multiprocessor Systems

→ Shared Memory

→ Distributed Memory

Clouds

Laptop/Mobile

Napster (sharing MP3 files)

Supercomputers

→ PARAM 10000

→ Blue Gene

Uses Message Passing.

Shared Memory Multiprocessor

Cache Coherence → Maintaining consistency in diff. copies of a memory in caches of diff. processors

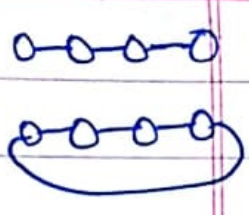
Problems →

→ Process Migration

Policies → Write Update → All others will also update

Write invalidate → All others will invalidate copy

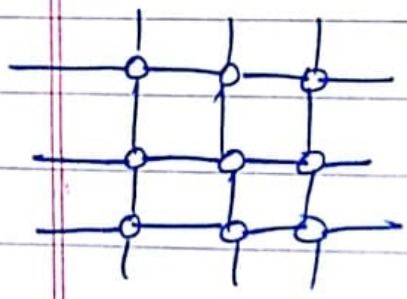
Static N/W → Dynamic N/Ws



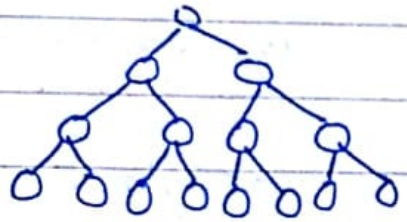
Linear array

Ring

Mesh



Tree

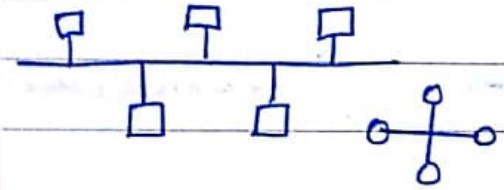


Congestion Problem

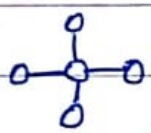
FAT Tree → As we Go towards root, bandwidth of links increases.

BUS

Used in HUB



Star used in switch



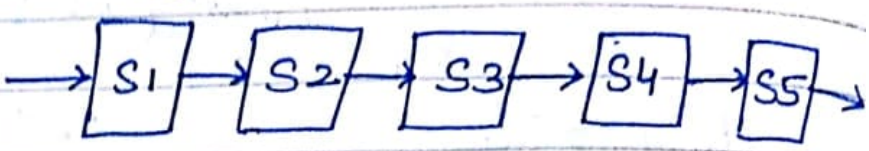
Collision Domain.

High Performance Computing

Pipeline Models

we have stages which have conn's going to next stage only

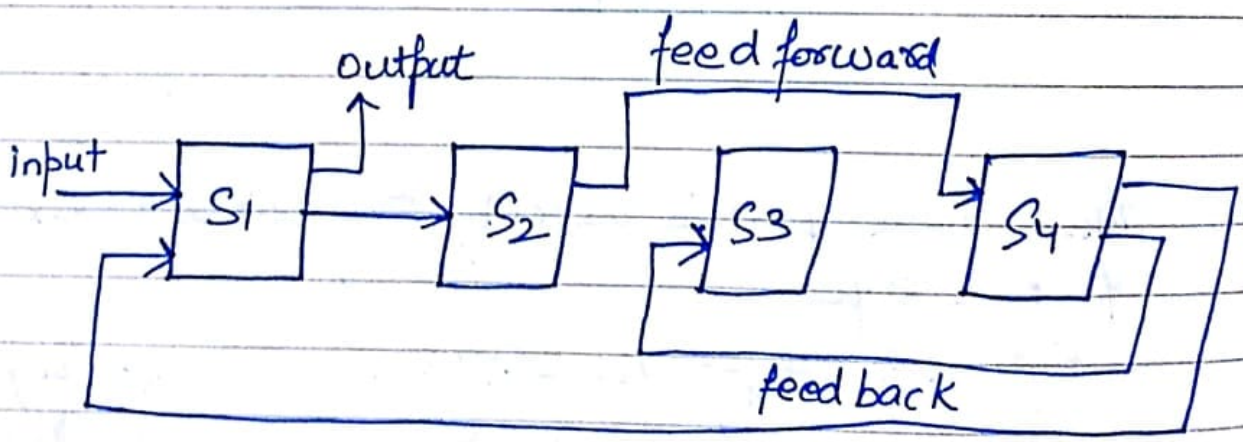
Linear Pipelines



Non-linear Pipelines

Feedback connection

feed



Row for each stage of pipelined system shows stage usage pattern

x → stage inactive state

Reservation Table

Cycles →

Stages ↓	1	2	3	4	5
S1	X				
S2		X			
S3			X		
S4				X	
S5					X

Linear
↳
only along diagonal.

	1	2	3	4		
S1	X					X
S2		X				
S3				X		
S4			X		X	

Non-linear
can have multiple crosses in a row/col.
~~Multiple in col~~
↳ multiple stages in 1 clock cycle.
Multiple in 1 row
↓
single stage in multiple cycles.

Why need for Non-Linear?

(Any NL can be converted to Linear)

Von Neumann → Instr^n & Data in same memory unit
Harvard Architecture

		1	2				
(Fetch)	IF / DF / WB	X			X		X
(Decode)	ID		X				
(ALU)	ALU (AG / EX)			X		X	

using 1 H/w unit is utilised & another in ID. AG in ALU. DF again in fetching unit EX in ALU & WB again ~~Same~~ fetching unit

∴ Whole H/W divided into 3 parts.

For converting to L, for each op^n, diff. H/W Unit stage.

By using more less.

In Harvard Arch → Introduced 2 Memory Blocks instead of 1

Linear

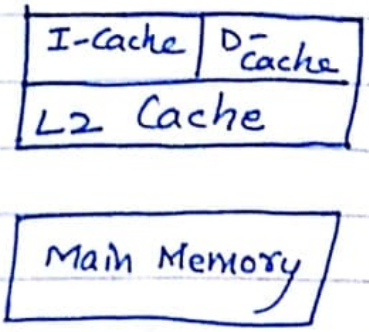
Demerits → More Rss.

Merits → Throughput ↑

Many tasks

each cycle can be employed to 1 task

Now - Both.



1	2	3	4	5	6	7
T_1	T_2		T_1	T_2	T_1	T_2
	T_1	T_2				
		T_1	T_2	T_1	T_2	

→ No conflict for 2 tasks but in 3 tasks

1	2	3	4	5	6	7
T_1	T_2	T_3	T_1	T_2	T_1	T_2
	T_1	T_2	T_3			
		T_1	T_2	T_1	T_2	

If we start 2 tasks at 1 & 2, we can start 3rd only at 8th cycle ∴ no ops in many cycles

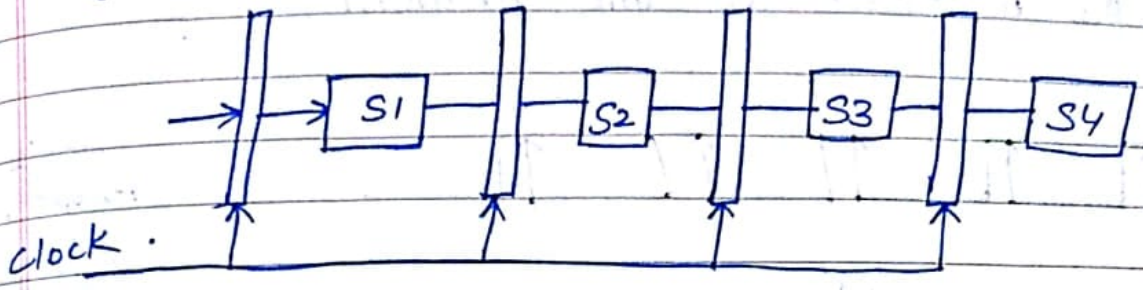
Avg. No of o/ps / unit time ↑

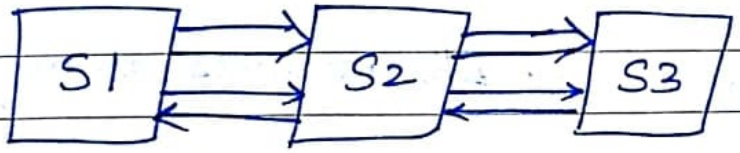
∴ Linear Pipelining reqd.

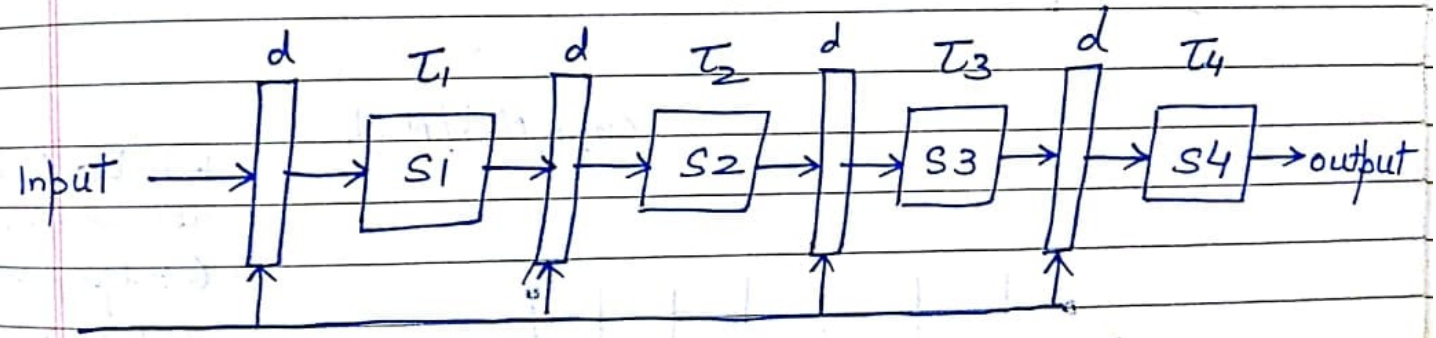
Pipeline Modes

- Asynchronous
- Synchronous

Synchronous → All stages separated by latch given same clock.







Neglecting latch Delay, $\left(\max_{i=1}^{N} \tau_i\right)$ → Max. Stage Delay (delay of slowest stage)

considering d,

$$\text{Clock Cycle Time } \tau = \left(\max_{i=1}^{N} \tau_i\right) + d$$

$$= \tau_m + d$$

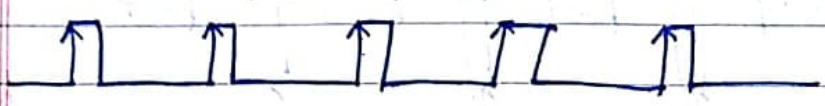
Pipeline Frequency → $\frac{1}{\tau}$

clock

clock doesn't arrive at diff places at same time

Clock Skewing

Active Clock Edges do not arrive at various places in the hardware at same moment.



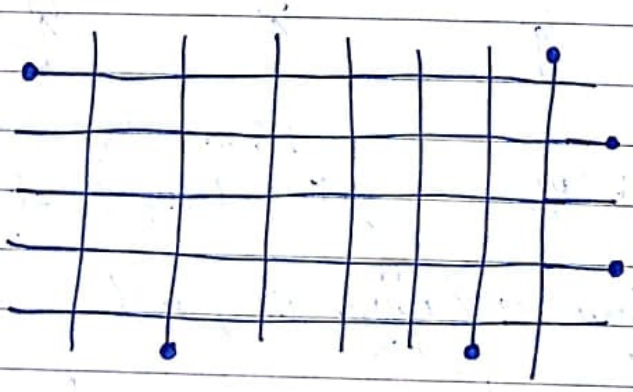
Inertial Delay
Transport Delay.
↓
due to e⁻s which take time to go from one place to another.

Transmission delay

Parasitic Capacitance → ~~gets~~ increases as we go far away.

Clock Net / Mesh

clock provided at many places (tapping provided at multiple places) & then taken from this net.

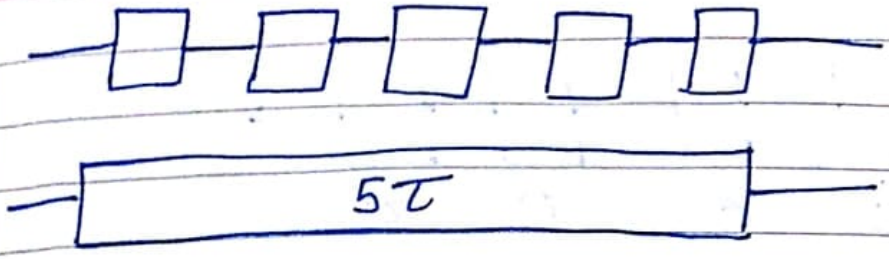


Speedup =

N tasks

K Stage

Uniform Stage delay τ



5τ	$5\tau + 0\tau$
$2 \times 5\tau$	$5\tau + 1\tau$
$3 \times 5\tau$	$5\tau + 2\tau$
$N \times 5\tau$	$5\tau + (N-1)\tau$

$$\text{Speedup} = \frac{\text{Time Required in a Sequential System}}{\text{Time Required in a Pipelined System}} = \frac{N \times 5\tau}{5\tau + (N-1)\tau}$$

$$\frac{N \times k\tau}{k\tau + (N-1)\tau} = \frac{Nk}{k + (N-1)}$$

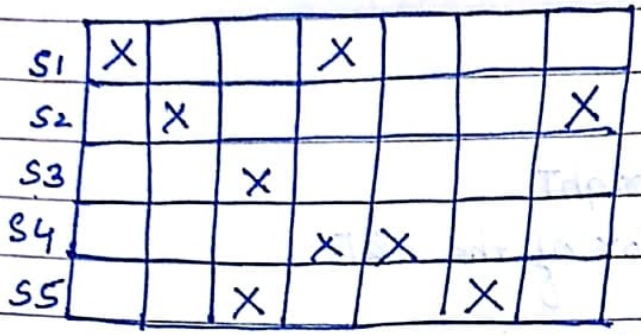
$$\lim_{N \to \infty} S_k = \lim_{N \to \infty} \frac{NK}{K + (N-1)} = \lim_{N \to \infty} \frac{N(K)}{N\left[\frac{K}{N} + \frac{N}{N} - \frac{1}{N}\right]}$$

$$= \frac{K}{1}$$

→ Uniform stage Delay is not possible

Dt. _____ B+
Pg. _____

Latency Analysis, Collision Free Scheduling



collision
↓
if at same time, two tasks require same stage.

Latency

the difference in two successive task initiations.

Forbidden
↓
may lead to Collision

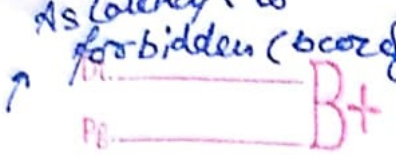
Permissible
↓
will not suffer from **Collisions**

CV → to represent Latencies
MSB of CV will always be 1
↓
shows max. forbidden Latency
Forb → count dist b/w all set of crosses in each row of Reservation table.

Binary Vector named Collision Vector
Forbidden latencies are shown by 1
and permissible by 0
Most significant Bit of a Collision Vector will always be 1

Forbidden → S1 → 3
S2 → 5
S3 → –
S4 → 1
S5 → 3
} 1, 3, 5

As latency 1 is forbidden (b coz of S4)



Collision Vector 1 0 1 0 1
5 4 3 2 1

(10101)

If we start T_2 at T_2, 2nd cycle will be 1st cycle for T_2

To get state now, right shift (logical)

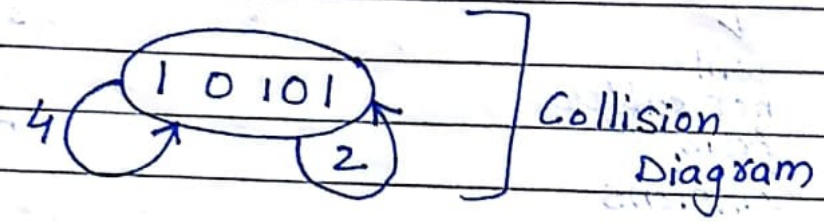
Logical shift right the collision vector of the state

Bitwise OR with original collision vector

If shift given, we cannot initiate task, otherwise we can

00101 (01
10101
10101

00001 (0101
10101
10101



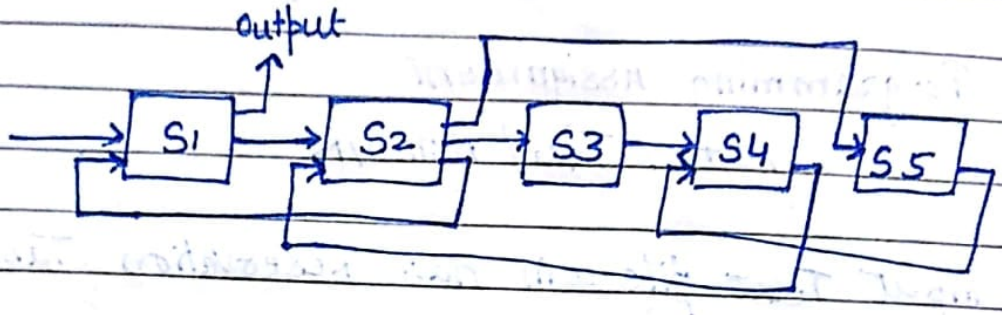
Simple Cycle → ~~Start with a node, visit others & come at start without visiting a node multiple times~~

Simple Cycle → 2
→ 4

Greedy Cycle → in which all ~~cycles~~ involved latencies are minimal permissible latencies for all nodes in the cycle.

	1	2	3	4	5	6
5 ← S1	X					X
3 ← S2		X			X	
– ← S3			X			
– ← S4				X		
– ← S5			X			

- We can make pipeline structure from table but not table from pipeline structure



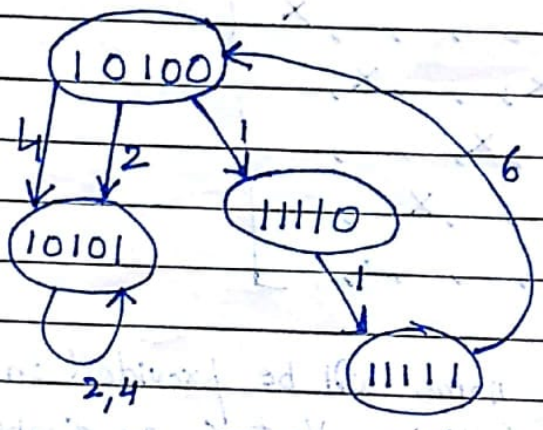
01010(0
10100
11110

01111(0
10100
11111

Cnot take at 1

0000(1 1 1 1 1
5 4 3 2 1

1 ∴ can't take.



00101(00
10100
10101

00101(01
10100
10101

00001(0101
10100
10101

Bitwise OR with original vector

★ Edge 6 will be from all nodes to original

write only if no cycle (edge?) from a node

Simple Cycle

1, 1, 6
2
4

Avg. Latency → 1,1,6 : $\frac{1+1+6}{3} = \frac{8}{3} = 2.67$
2 : 2
4 : 4

Greedy cycle : [1, 1, 6
2

Greedy
↓
all nodes of cycle are from minimal permissible edge.

Dt.
Pg.

M.A.L

Minimal Avg. Latency = 2

min. of avg. latency of all Greedy cycles.

Bounds on MAL

Cannot be lower than max. no of crosses in any row of table

⇒ ≥ 2

We got 2

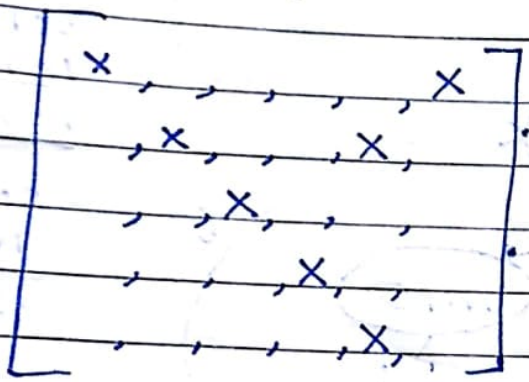
∴ We got Optimal MAL

Optimal MAL

Programming Assignment

C / C++ / Java / Python / R

Input Text file will have Reservation Table



Text file name will be provided on command line

Compute Collision Vector, all simple cycles, Greedy cycle and MAL

Find whether the MAL is optimal or not.

Forbidden → 1, 2, 4
Permissible → 3
Collision Vector → 1011

	1	2	3	4	5
S1	X				X
S2		X		X	
S3			X	X	

0001011
1011
1011

Simple cycle = 3 MAL = 3
Greedy Cycle = 3

Optimal MAL = max. number of cross marks in any row of the reservation table.

Bounds of MAL

→ MAL is upper bounded by total number of 1's in the initial collision vector plus 1

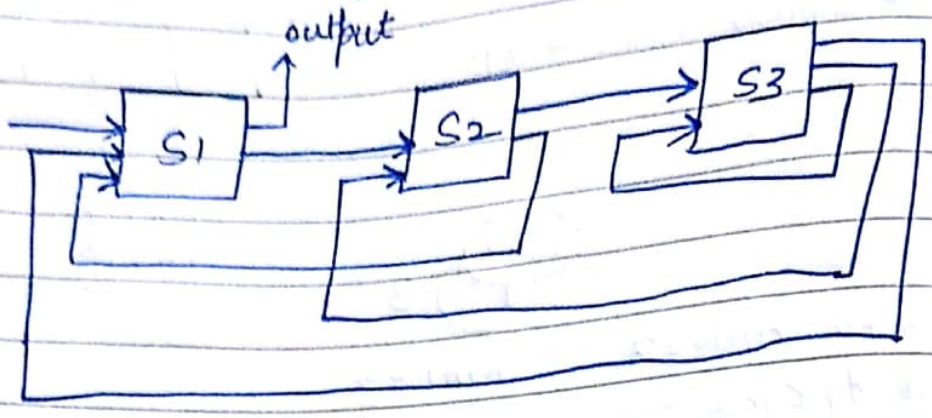
→ MAL is lower bounded by the max. no. of cross marks in any row of the reservation table.

→ MAL is lower than or equal to the average latency of any greedy cycle in the Collision state Diagram.

Average Latency of any greedy cycle is upper bounded by the no. of 1's in initial Collision Vector plus 1.

For above eg → Upper bound → 4
Lower Bound → 2.
∴ This ~~table~~ MAL is not optimal

if we dont get optimal MAL, we may insert non-compute delay to approach towards optimal value.

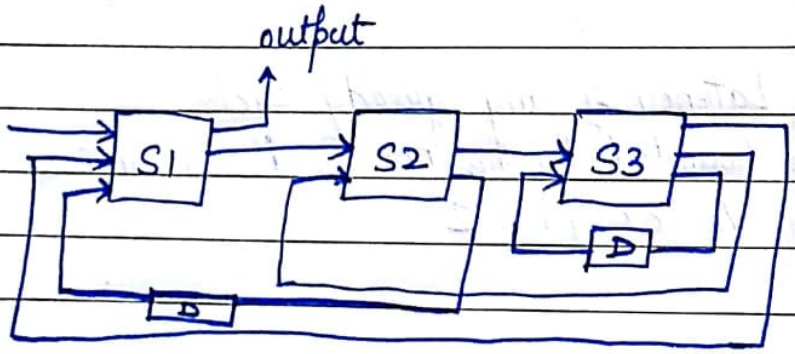


	1	2	3	4	5
S1	X				X
S2		X		X	
S3			X	X	
D1					

work done in 4 to be transferred next stage
∴ S2→S1 Delay & S3→S1 delay.

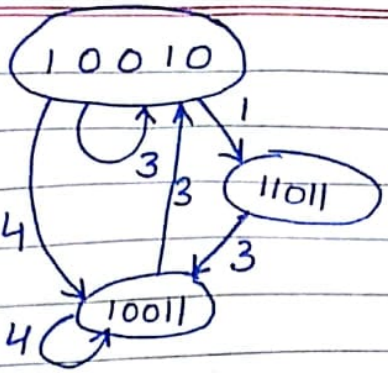
	1	2	3	4	5	6
S1	X				X	X
S2		X		X		
S3			X	X	X	
D1				X		

increase To ~~reach~~ Latency, separate S3 with 1 Latency.



Forbidden – 2, 5
Permissible → 1, 3, 4

5 4 3 2 1
1 0 0 1 0



Simple Cycles →

1, 3, 3

3

4, 3

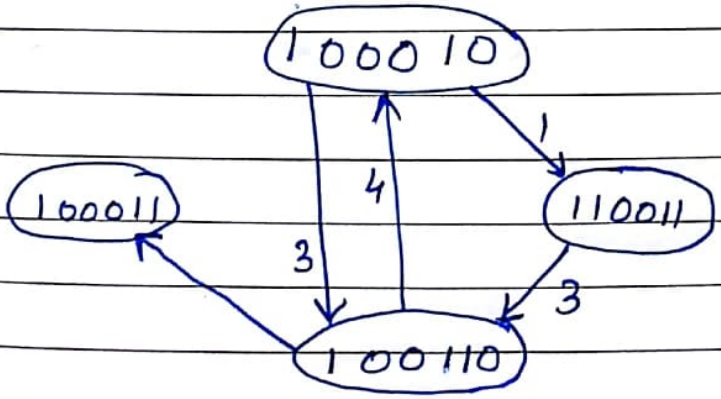
4

Greedy → 1, 3, 3

MAL → $\frac{1+3+3}{3} = \frac{7}{3} = 2.33$

Although task is residing for longer duration but throughput is increased.

	1	2	3	4	5	6	7
S1	X				(X)	(X) → X	X
S2		X					
S3			X	(X) →	X		
D1				X			
D2						X	



Instruction Pipeline Design

Instruction Execution → Fetch
→ Decode
→ Execute operation

CISC → | IF | ID | AG | DF | EX | → Write Back may be separate stage or may be considered as part of EX stage

x86

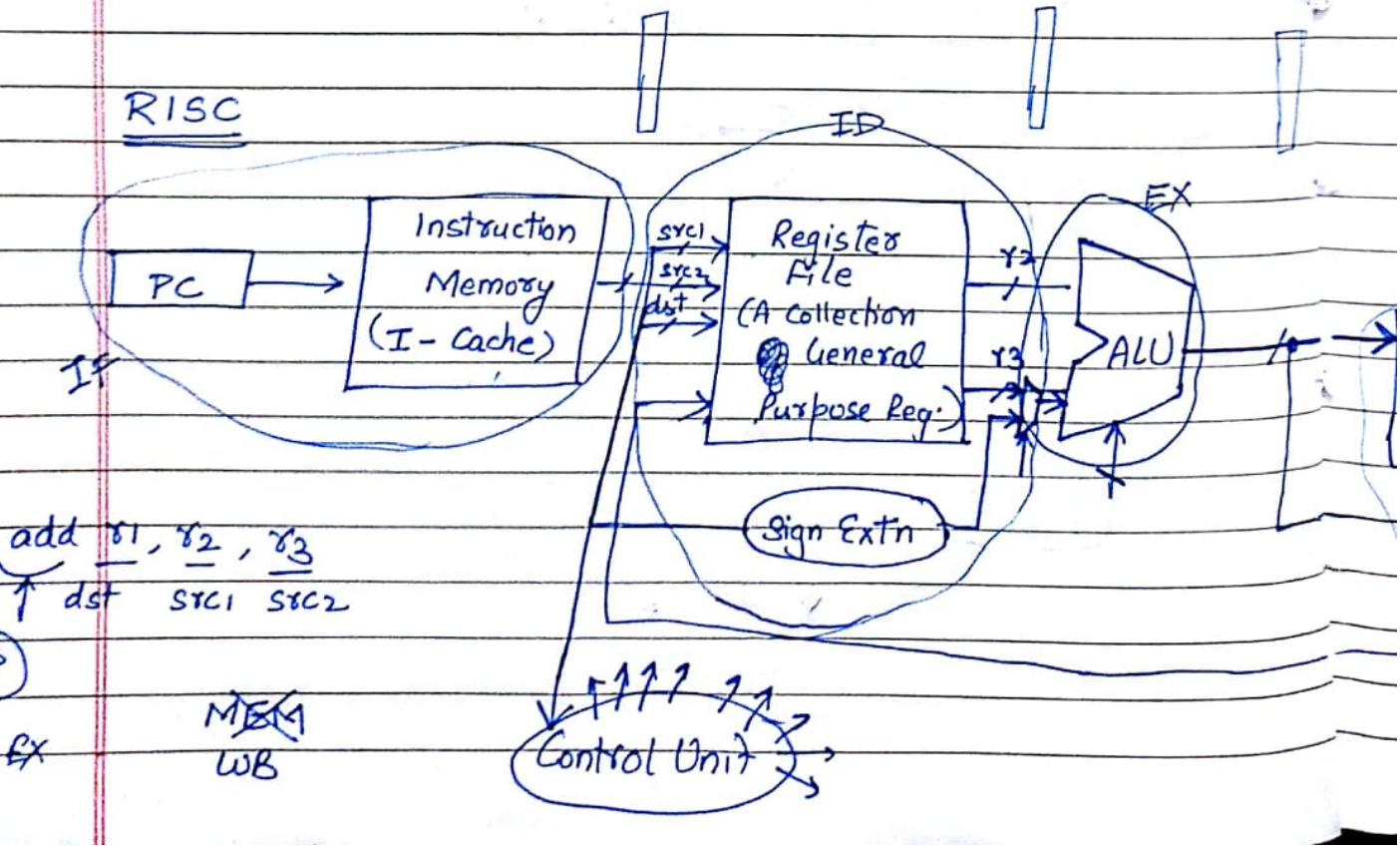
| IF1 | IF2 | ID | AG | DF1 | DF2 | EX1 | EX2 |

RISC → | IF | ID | EX | MEM | WB |

ARM

MIPS

Hennery & Patterson



Instr^n?

Register file → 2 Instruction Port

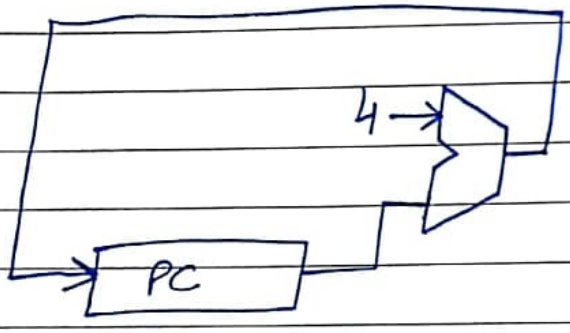
You can do 2 read op^{n}s at the same time (same reg or diff reg.)

ld r1, 10(r2)

r2
src1

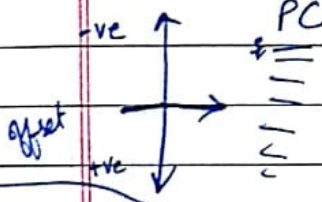
10
src2

dst
r1



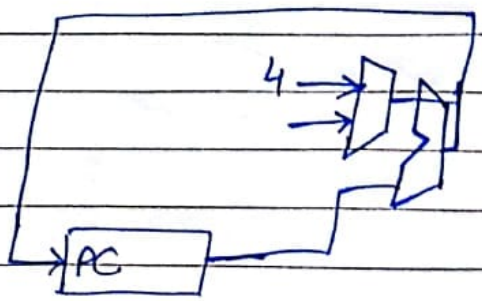
JMP

PC Relative



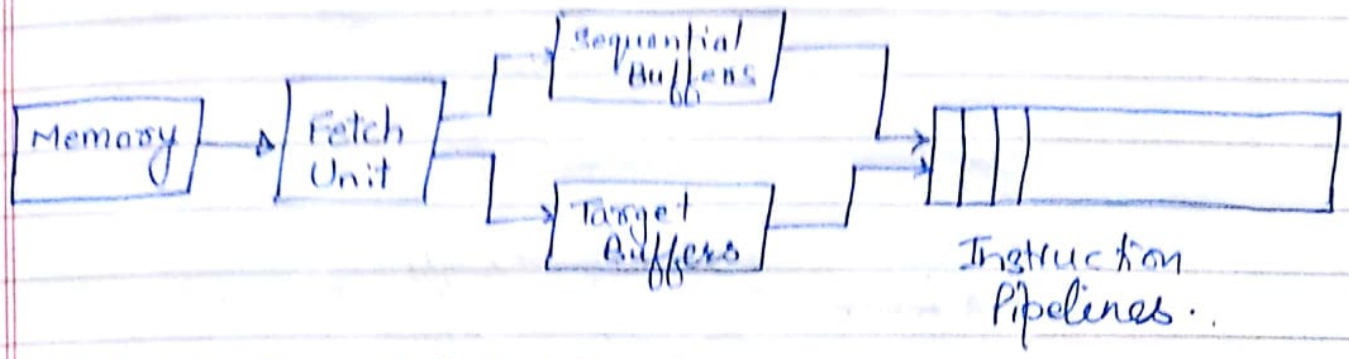
All Instructions are of uniform size equal to 32-bits

Default increment → 4. Sometimes (in PC Relative) increment by other value (specified in instr^n)



HPC

Prefetch Buffers



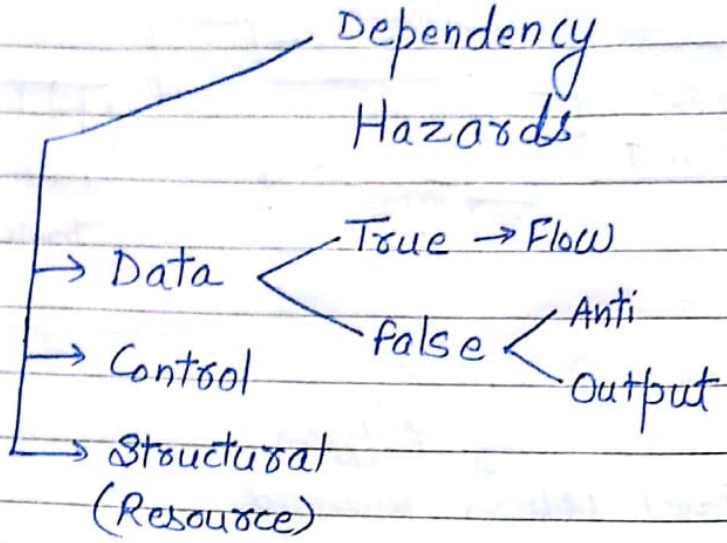
fetched by Mem. Unit & Instrs
instrs are directed to Seq. Buffers
Whenever Target address generated
for Branch instr then that instrn corresp to that address is
stored in Target Buffer. When exact direction is known,
if Branch gone to target, instn in Target Buffer will be
provided to Pipeline & then they will swap their
functionality.

Branch Instruction

→ When Branch instruction is executed and Branch goes to Target → Target Buffers start working as Sequential Buffers and vice-versa.

multiple Functional Units → Latest

Internal Data Forwarding



→ How dep b/w ALU type instr^n s.

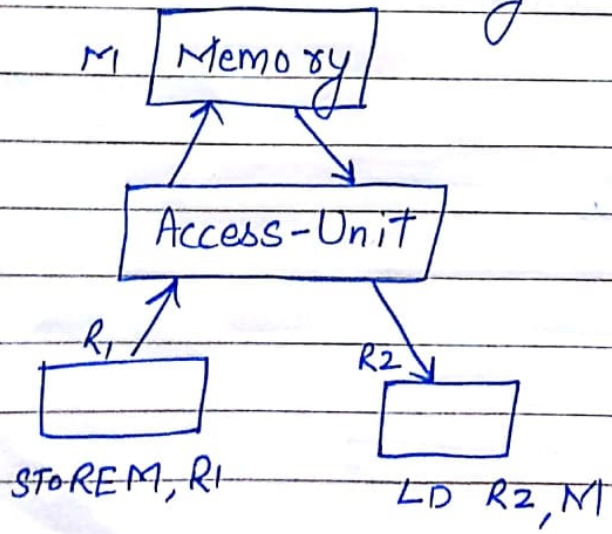
```
add  R1, R2, R3
sub  R4, R5, R1
```

~~~

→ b/w load & ALU

```
load  R1, 10(R2)
add   R4, R5, R1
```

~~~

→ b/w ALU & store.

```
add   R4, R5, R1
store R4, 10(R7)
```

Store - Load Forwarding



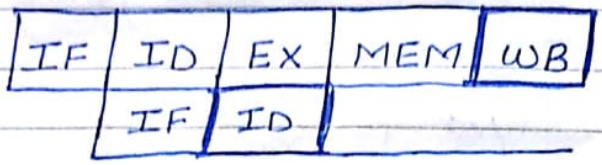
Load Load Forwarding

Store Store Forwarding

CISC

MOV R2, R1

RISC



add (r1), r2, r3

sub r4, r5, (r1)

data not available ∴ need to wait for 3 cycles
∴ 3 cycles waste → no o/p.

value is generated in Ex of 1st & reqd. in Ex of 2nd ∴ instead of writing & then reading, provide a direct path from o/p of ALU to the i/p of ALU & activate the bypass path if o/p of one instr^n is same as source of ~~the~~ next instr^n.

2 instr^{n}s that are ALU type & dest^n of 1st = Source of 2nd.
∴ logically wrong value (old) is not used at all.

If ALU followed by ALU, we can fully remove stall cycles
But if →

Load r1, 10(r2)
add r4, r5, r1

IF ID EX MEM WB

Here, reqd 1 cycle before being generated ∴ scheduled after 1 cycle

ALU o/p → Tapping → to MUX joined to i/p of ALU

Data forwarding → Purely H/W sol^n.

Compiler → Instr^n Reordering

Hazard

Bernstein Conditions

Instr^n Process Statement Task.

Instr^n followed by Instr^n Process

If
- Input of $I_1 \cap$ Output of $I_2 = \phi$ — False-Anti
- Output of $I_1 \cap$ ~~Output~~ Input of $I_2 = \phi$ — TRUE
- Output of $I_1 \cap$ Output of $I_2 = \phi$ — False-output

I_1 { P_1 { S_1 } T_1 }
I_2 { P_2 { S_2 } T_2 }

then I_1 is said to be parallel to I_2

(if both i/p = no dependency)

False + Anti — WAR

TRUE — RAW

False + output — WAW

→ I1 : add r1, r2, r3
I2
I3
→ I4 : load r3, 10(r5)
I5

If normal pipeline, no effect of false dependency (Anti Above)
If out of order Exec. ~~Instr Reordering~~ → Problem
↳ To take mak adv. of parallelism.

If 1 instr^n waiting of some other thing to happen, then rest of independent instr^n's can run.

To increase utilized Parallelism.

Dynamic Instruction scheduling → Scoreboard
→ Tomasulo's Algorithm

Scoreboard	Tomasulo's
Computer CDC-6600	IBM 360/91

~~Scoreboard appeared earlier in market & Tomasulo came 3 yrs later~~

Scoreboard

To achieve the IPC = 1

IPC Instruction per cycle
CPI Cycles per instruction

$\left(CPI = \frac{1}{IPC}\right)$

we dont get IPC = 1 because of Dependencies causing Hazards / Stall cycles

Dependencies

↓

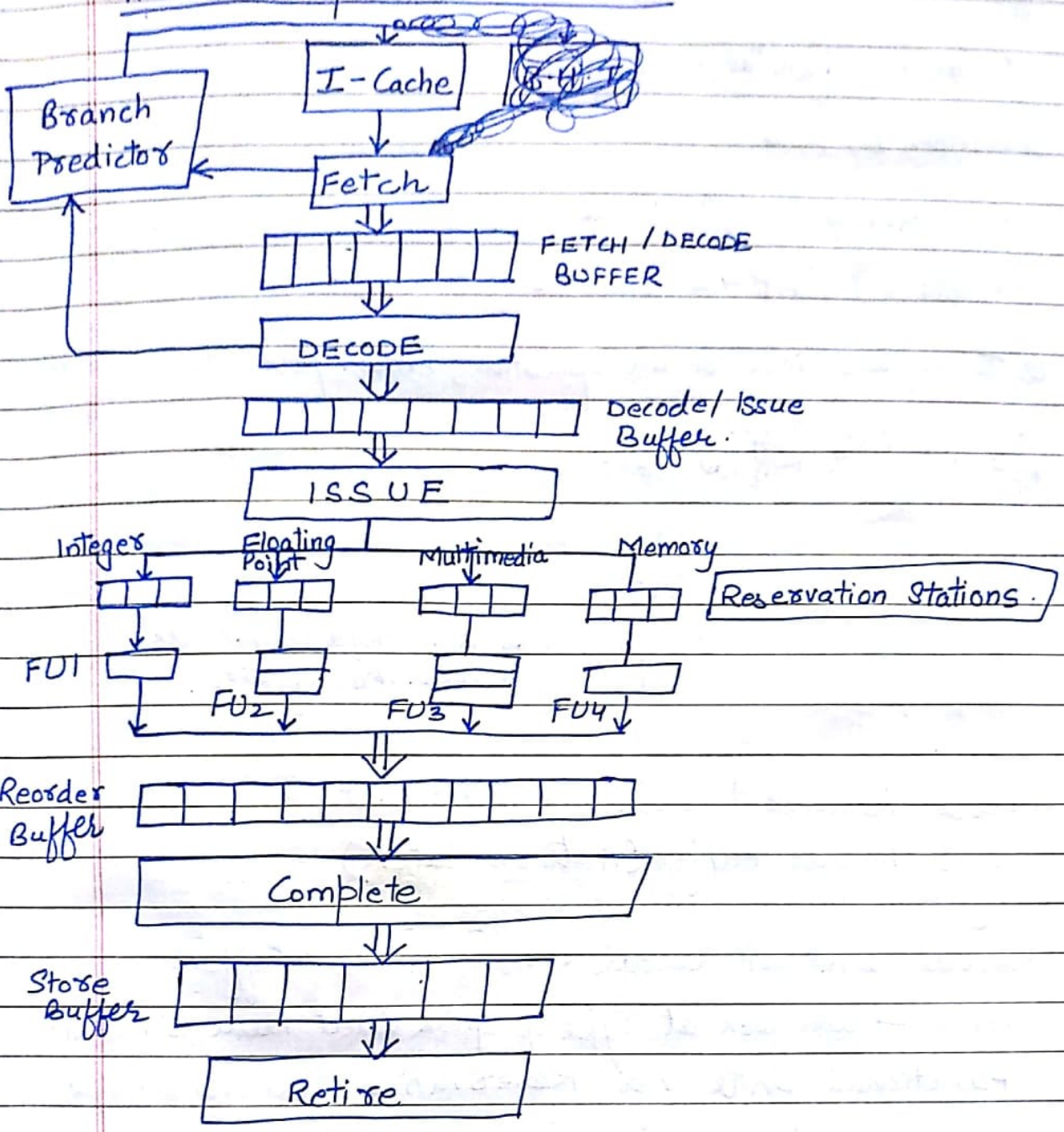
Hazards

Ideal case Avg. CPI = 1

CPI ≠ 1

Avg. CPI = 1 + Avg Stall cycles

A Modern Superscalar Processor



Dynamic Instⁿ Scheduling

- Scoreboard — 1963 — CDC 6600
- Tomasulo — 1966 — IBM 360/91

Scoreboard → To achieve IPC = 1
However, they also used stalling to handle dependencies

Tomasulo → Implemented Instⁿ reordering, register renaming, loop unrolling

In normal Pipeline → Only 1 EX unit to handle all type of operations.
Heterogeneous func"al units.

Load/Store by FU 4.

As complexity of f" ↑, we try to have more pipeline units (stages)
∴ Integer → 1, FP → 2 MM → 3

To improve time of an operation, either parallelism (multiple seq. units) or divide into stages.
Ilism → Disadv Cost.

Pipelining Adv + Cost
Simpler (No HW reqd to monitor allocation of rss).

Data Mem → I-Cache → Fetch Unit → Stores in Fetch Decode buffer (Acts as latch)
↓
Decode Unit will decode & store in DI Buffer.
Issue → will look at type of op" & direct them to relevant Functional Units via Reservation Stations (buffers)
Can go directly to FUs if no Instr" waiting in Reservation station (without buffer → implementation dependent.
↓
Reorder Buffer → Complete.
These FUs can take diff. amounts of time

We can allow out of Order Execution but not out of order completion
∴ Out of Order Exec but in order completion

Complete ⇒ Restores the program exec^{n} order.

Last stage → Mainly to handle memory instr^{n} cycle.
If written into buffer → assumed work is over
other H/W will keep writing to Memory.
(Can't read & write to mem at same time)

Dynamic Instruction Scheduling with Scoreboard

→ Scoreboard is a centralized hardware that tries to maintain an execution rate of one instruction per cycle by executing an instruction as soon as its operands are available in registers and no hazard conditions prevent it.
→ We have IF, ID1, ID2, EX, WB stages
→ Every instruction goes through Scoreboard where a record of data dependencies is constructed.
→ A system with Scoreboard is assumed to have several Functional Units (FUs) with their status information reported to the Scoreboard.
→ If the scoreboard determines that an instruction can not execute immediately, it executes another waiting instruction and keeps monitoring FU's status to decide when the instruction can proceed to execute

Issue (ID1) → An instruction can be issued if
- Required FU is available (structural Hazard)
- No WAW Hazard (will not allow if dest^{n}s. same)

ID2 (Read operand) :- a FU can read the source operand when all of them are available
(If some earlier op^n has marked the reg to be written, then wait.) / if 3 byte instr^n, if r_1 - av. but not r_2, wait not continue

A register is not available if some earlier active instruction has marked it for writing.
(No RAW hazard)

EX (Execute) → When all source operands are available, FU starts execution

Write Result (WB) - FU can write the result to a reg. on execution completion if there is no WAR hazard.

(Later instr^{n}s are not allowed to write if WAR hazard. as old instr^n may require old value of a reg.)

~~If integer type source xy~~

Hennessy
Patterson

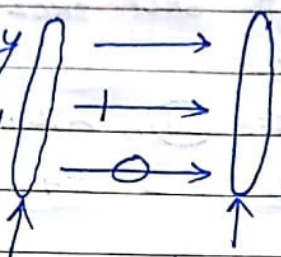
I1	LD	F6,	34	(R2)
I2	LD	F2,	45	(R3)
I3	MUL	F0,	F2,	F4
I4	SUB	F8,	F6,	F2
I5	DIV	F10,	F0,	F6
I6	ADD	F6,	f8,	F2

Dependency Graph

RAW — Flow Dependency

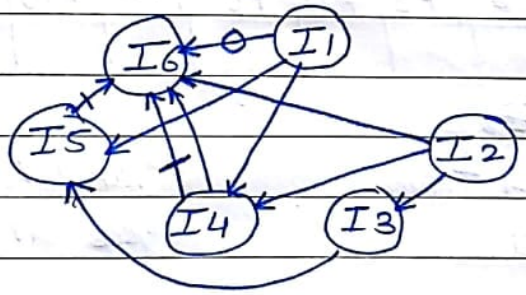
WAR — Anti Dependency

WAW — Output Dependency



Instructions earlier in Program order.

Instruction Later in Program order



Scoreboard can be constructed with this graph.

		No. of FU	EX. Cycles
	Inter	1	1
Floating Point	Mul	2	10
FP	ADD	1	2
FP	DIV	1	40

Instruction Status

	J	K	Issue	Read operand	Ex Complete	Write Result
LD F6	34	R2	1	2	3	4
LD F2	45	R3	5	6	7	8
MUL F0	F2	F4	6	9	19	20
SUB F8	F6	F2	7	9	11	12
DIV F10	F0	F6	8	21	61	62
ADD F6	F8	F2	13	14	16	22

FU Status

	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer (Load)	Yes	LD	F6	F2	~~34~~ 45	~~R2~~ R3	—	Yes	Yes
MUL1	Yes	MUL	F0	F2	F4	Integer	—	~~No~~ Yes	Yes
MUL2									
ADD/SUB	Yes	SUB	F8	F6	F2	—	Integer	Yes	~~No~~ Yes
DIV	Yes	DIV	F10	F0	F6	MUL1	—	No	Yes

Register Status

	F0	F2	F4	F6	F8	F10	f12
FU →				Integer			

~~Integer~~ MUL1 Integer ADD/SUB DIV

MUL1 ADD/SUB DIV

Right now only 1 instr^{n} ∴ no WAW Hazard ∴ can Issue.

Issue is always done in Program Order.
Rest of stages may be out of order

We cannot issue any other instr^{n} I3+ even if resource available as Program order need to be same.

4→ Integer Load marked Free

In Tomasulo's Algo → Enhancement.

We will Read from FU (Reservation station) instead of Registers

Reservation station

Work as Rename Registers

Data Forwarding through Common Data Bus

~~through~~ will all FUs will write their result to Register.

~~Data Bus → Data &~~ ~~Source of Data~~ (who is writing) is present.

*Each Unit will store which FU has to write. When Data bus has source as that FU, ll to writing in Register File, Data is read ~~by FU~~ from Bus. ∴ 1 cycle is saved,

Tomasulo's Algorithm

→ Control and Buffers distributed with FUs vs. centralized in Scoreboard

* FU buffers are called Reservation Stations. These Reservation Stations are also referred to as renaming registers which are different from architectural registers specified by ISA

– ISA registers in instructions are replaced by either values (if available) or pointers (renamed) to reservation stations that will supply value later on. This process is called Register Renaming

→ It allows hardware based loop unrolling

→ Instruction results go (forwarded) from RSs to RSs, not through registers, over common data bus which broadcasts result to all waiting RSs

→ Load and Store are treated as FU with RSs as well