Types of Instructions

① Data transfer Instructions
→ Use to move the data from
- •) Register to m/m
- •) m/m to register
- •) register to register

② Data manipulation Instructions
- •) Arithmatic opⁿ
- •) Logical opⁿ
- •) Shift opⁿ

③ Program Control Instructions
- •) Use to make decisions
- •) execution transfer statements

Data transfer instructions

We have 3 type of components
•) memory •) register •) I/O device
Generally I/O devices also consider as a registers.

① **Load**: Load instruction from m/m to register.
→ Generally use in loading accumulator.

② **Store**: register to m/m store
→ Use in case of Accumulator

③ **exchange**: register to register or memory to register.
→ Just like swap.

④ **Input/output**: Put input data in register and from register take out as Output.

⑤ **PUSH/POP**: for stack implementation.

Each instruction can generate many instruction on the basis of mode and operands.

Ex Load can be modified as—
LDI → Load Immediate
LD R, X → Direct addressing mode

Arithmatic Instruction

Basic operation → Addition, Subtraction
We can implement any operation using above two.

Increment ←
Decrement ← → Doesn't require CPU

Multiply, Divide, Add with carry
Subtract with borrow, Negate.

Each instruction can have various execution on the basis of mode and addressing operands.
Negate gives 2's complement.

Logical Instruction
Shift Instructions
Program Control Instruction

~~Learned from book~~

Logical Instructions

→ Operations work on bits called logical operations.

Logical 'AND' or 'OR' (∧/∨)
→ 'X-OR' operations.
→ 'Clear' done at register level. It means change all bit to '0'.
→ 'complement' means 1's complement.

XOR of any no with all 1's gives 1's complement.

→ Carry is a flag (1 bit data)

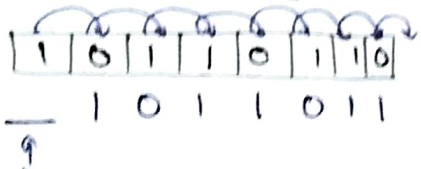
→ Clear carry / set carry are operations works on flag bit.

→ Complement carry instruction is useful in implementing subtraction.

→ Enable / Disable interrupts opns. have interrupt flag seperately.

Shift Instructions:

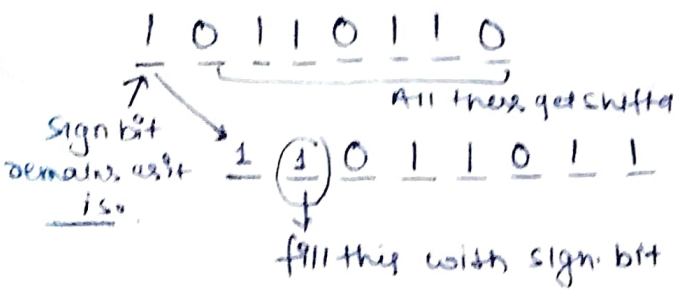
① Logical shift Right / Left



→ fill it with zero, and this zero is not due to LSB. It's just to fill the vacant bits

Similarly works "Logical shift left."

② Arithmetic Shift Right / Left



③ Ex.

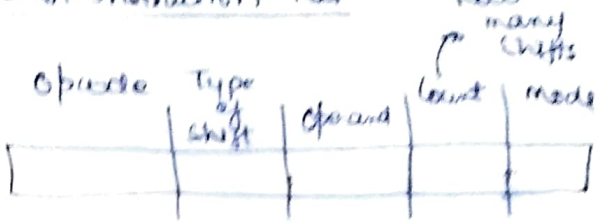
0 1 0 0 1 1 0 1

After >> shift

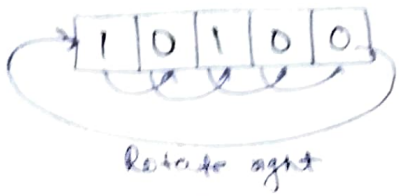
0 0 1 0 0 1 1 0

Arithmetic and Logical Left Shift both are same.

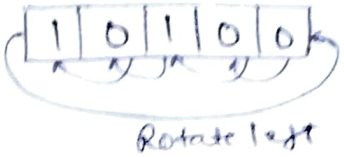
Shift instruction has



Rotate right / left

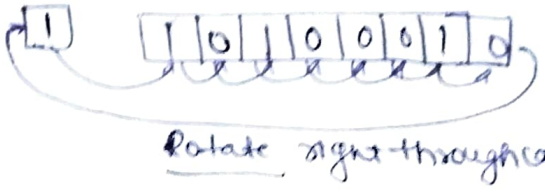


0 1 0 1 0 0/p



0 1 0 0 1 → 0/p

Rotate right / left through carry



→ We have an extra carry bit.

As right shift we can do left shift also.

Program Control Instructions

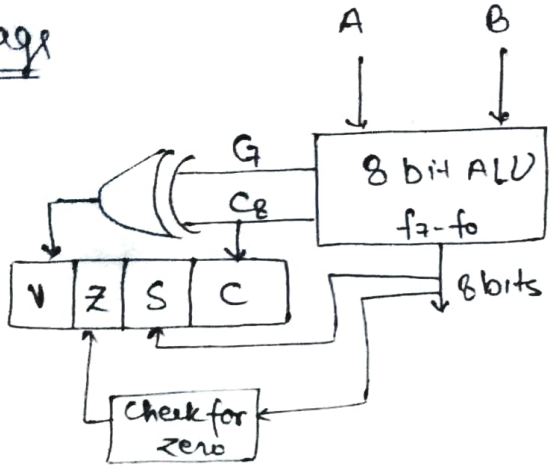
Branch, Jump, Skip,
(BR) (JZ, JNZ)

Call, return, compare, test.

→ Skip means to ignore the next instruction.

→ Compare and test are similar kind of operations.

flags



V → overflow flag

Z → Zero flag checks result is zero or not.

$C_7 \oplus C_8 = 1$ means overflow

→ We can also verify the value of particular bit using zero flag and test instruction.

S → Sign flag.

C → carry flag.

On A - B

Signed	Unsigned
A = B, Z = 1	A = B, Z = 1
A > B, S = 0, Z = 0, V = 0	A > B, C = 1, Z = 0

otherwise A < B

flags can be used in comparing two numbers.

Examples

→ Generally we performed subtraction using adder.

A = 1100

B = 0010

A - B = ?

2's comp. of B = 1101 + 0001

= 1110

A - B = A + 2's comp of B

= 1 1010

Carry → MSB

Program status word

Z	V	S	C
0	0	1	1

$V = C_{in} \oplus C_{out} = 1 \oplus 1 = 0$

So, now we can compare A & B using all the flags values.

A = 1100	(12)	(-4)
B = 0010	(2)	(2)
	↓ Unsigned (A>B)	↓ Signed (A<B)

On signed values if,

① V = Z = 0, S = 1 (A < B)

On Unsigned numbers,

→ We don't care about overflow

⇒ Z = 0, C = 1 (A > B)

Ex → A = 0010 | 2 | 2

B = 1100 | 12 | -4

A - B = 0010 + 0100

= 0110

PSW

Z	V	S	C
0	0	0	0

Signed : V = Z = 0, S = 0 (A > B)

Unsigned : Z = 0, C = 0 (A < B)

→ Carry is complement of borrow
It means if C=0 then B=1

Symmary

Signed Number

V	Z	S	Remark
1	x	x	Invalid
0	1	x	(A = B)
0	0	0	(A > B)
0	0	1	(A < B

Unsigned Number

Z	C	Remark
1	x	A = B
0	0	(A < B) if C=0 ⇒ B=1 in 2's complement substraction
0	1	(A > B) (No borrow)

Conditional Branch Instruction

BZ : Branch if zero (Z=1)
BNZ : Branch if no zero (Z=0)
BC : Branch if carry (C=1)
BNC : Branch if no carry (C=0)
BP : Branch if positive (S=0)
BM : Branch if negative (S=1)
BV : Branch if overflow (V=1)
BNV : Branch if no overflow (V=0)

Unsigned

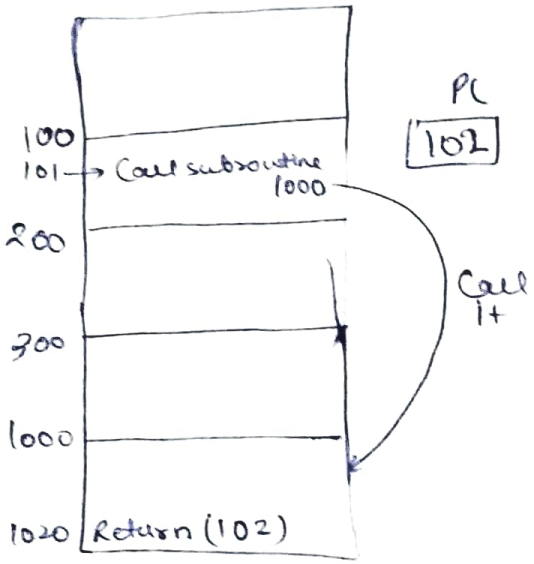
BHI → Branch if higher (A>B)
BHE → —— higher or equal (A≥B)
BLO → Branch if lower (A<B)
BLOE → Branch if lower or equal (A≤B)
E → Branch if equal (A=B)
→ (A!=B)

Signed

BGT → Branch if greater (A>B)
BGE → —— greater or equal
BLT → —— lower (A<B)
BLE → —— lower or equal
BE → A = B
BNE → (A != B)

Call and return Instructions

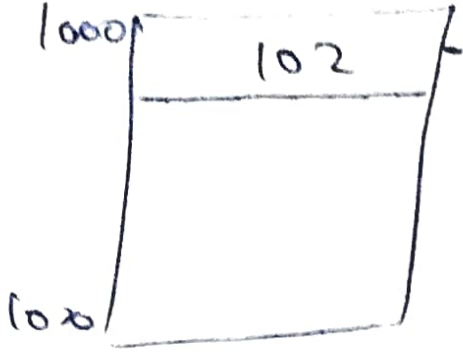
→ Both are program control instructions.
→ They manages the flow of program



After completion we get return to 102 but if during execution of subroutine calling function get shifted to 200 then (102) add. Creates inconsistency for 102 we need to return on address 202. for this we have soln like

① Store the value of 102 to temp. register and when return back load the value of temp to PC again but this was using long back and it was not much appropriate soln.

(2) Store in subroutine like



→ Maintain some space for value of PC and load it back to PC on return.

(3) Store in any fixed memory location.

(4) store in m/m stack and it is the best method known till now

Note → Read this topic from book too or from any other YT Video, if we get issue while solving problem.

Pipelining

→ We can execute more than one instruction at a time using one CPU.

Operations performs on instruction -

① Instruction fetch ② Instruction Decode ③ Execute ④ Write Back.

→ Each instruction takes some time in terms of clock cycles Suppose an instruction performs all four opn in 4 units then for 3 instructions we require $3 \times 4 = 12$ units bcoz we have only single CPU but we can do it in less time using pipelining.

Phase time diagram

→ We are trying to manage CPU in such a way it get divided into 4 parts and each will perform single operation so that we can achieve parallelism.

Clock	1	2	3	4	5	6	7	8	9	10	11	12
IF	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9			
ID	-	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9		
EX	-	-	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9	
BW	-	-	-	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9

Here, we complete even more than 4 instn in 12 clocks.

CPI → Clock per Instruction

Main aim of Pipelining is to keep, CPI ≅ 1 which is ideal.

1 Instruction = 1 Clock, 1 Instr. = $\frac{1}{f}$ sec, ⇒ 1 sec = f Instr.

Performance is measured using utilization or efficiency and speedup.

Load = No. of Instruction that can be perform in given time. In above example Load = 9I

$$\text{efficiency} = \frac{36}{12 \times 4} = \frac{3}{4} \quad \frac{(\text{No Vacant box})}{(\text{Total box})}$$

$$\text{Speedup (load is given)} = \frac{T(\text{without pipeline})}{T(\text{with pipeline})} = \frac{9 \times 4}{12} = 3$$

$m \times efficiency = speed\ up$ $\quad m \times e = s$

where, 'm' is no of operations required to perform an operation.

→ In practice we will never get 100% efficiency, because some blocks are empty in table. ($S < m$)

↗ but in theory ($S \cong m$) nearly equal.

Suppose we execute 'n' instructions then time taken in

pipelining = $1 \times m + (n-1) * 1$

$= m + (n-1)$

without pipeline = $m \times n$

Speed up = $\frac{n * m}{m + n - 1}$

$= \frac{m}{1 + (m-1)/n}$

↘ Suppose, $n \to \infty$ very large

Speed up $\cong \frac{m}{1}$ In theory

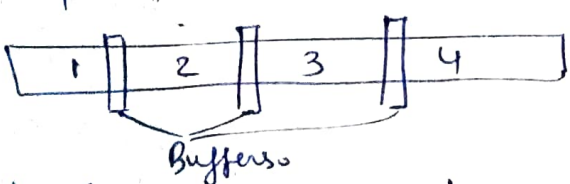
So, T_{wp} (without pipeline) $< m * T_p$ → Just remember only.

Speed up → how fast we perform with pipeling?

So, second aim of pipelining is to keep $S \cong m$ ideal.

In practical value of 'm' can lie between 3 → 24 but various costs and delay add up in between. So generally we go for 5 (experimentally).

In non-pipeline mode, CPU acts as a single unit, but in pipeline mode we divide CPU to various stages to perform various operations as Suppose each instruction is performing 4 operations then,



we store the result of previous opn in buffers.

→ It is not always possible that all the opn takes equal time so it might provide some delays in loading to buffer or loading from buffers. So we can say that

$T_p \geq T_{wp}$ → for a single instruction not for various instructions.

Gatee 99

	F	D	E	W
1	1	2	1	1
2	1	2	2	1
3	2	1	3	2
4	1	3	2	1
5	1	2	1	2

Soln ① There are four operations and 5 instructions

② Each instruction doesn't takes equal no. of clock cycles

③ Don't start two instruction operation of same type at the same clock cycle

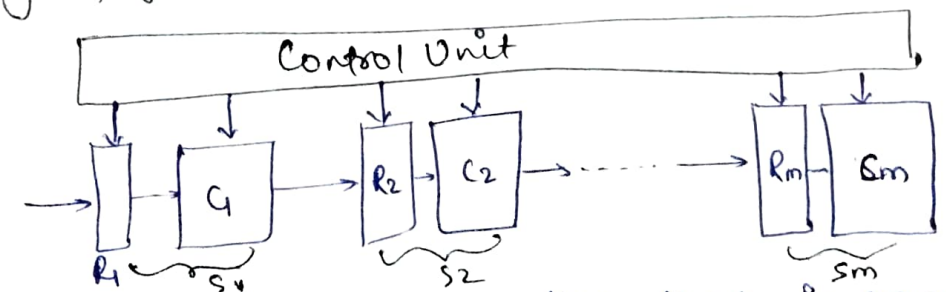
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
I_1	F	D	D	E	W										
I_2		F	X	D	D	E	E	W							
I_3			F	F	X	D	X	E	E	E	W	W			
I_4					F	X	D	D	D	X	E	E	W		
I_5						F	X	X	X	D	D	X	E	W	W

complete I_1 ↑ Complete I_2 ↑ Complete I_3 ↑ Complete I_4 ↑ Complete I_5

→ Instead of 15 we can take more than it but it will waste of space.

→ Choose max. value of cycle (15 here) as the max. value given in option.

Designing of pipeline (Hardware)



There are 'm' stages or operation that is managed by control unit with the help of clock signals.

R_i → Registers used as buffer, C_i → Combinational ckt performing its tasks.

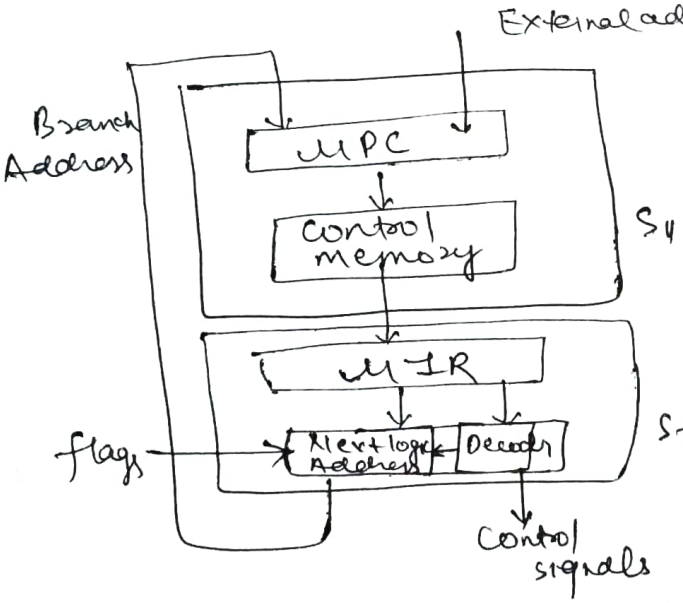
We designed it, to achieve CPI ≈ 1 (Ideally)

→ If clock is double then data will not move from one Ci → another Ci.

→ If the stages in pipelining are having uneven or non-uniform delays then we will set the clock cycle according to largest delay, and in this case we will not achieve (CPI = 1).

To achieve CPI = 1 we have to ensure that delay should be uniform and each opn should take same time but this is not possible.

Here is internal str. of 2 stage pipelining



μPC → micro Program Control [Tells about next micro instruction]

μIR → micro Instruction Register

Q A 4-stage pipeline has the stage delays 150, 120, 165, 140 ns (Gate 2004)

Soln Since we are using constant clocking rate (given), So we will go for clock with max. time = 165 ns

for 1000 data items / instructions in 4 stages

$$1 \times 4 \times 165 + 999 \times 165$$

$$= 165.495 \text{ us}$$

$$= 165.5 \text{ us}$$

Gate 2011 : Consider an Instⁿ pipeline with 4-stages (S_1, S_2, S_3, S_4). What is the approximate speed up of the pipeline.

Sol^n speedup = $\frac{T_{wp}}{T_p}$ = $\frac{\text{Sum of time of all stages without buffer}}{\text{Max stage time with buffer}}$

$= \frac{30}{12} = 2.5$.

Doubt : You might get confused the the question is talking about Ideal condition so, $S \cong m$ (4) here. But it is possible if delays are same but here at each stage delays are uneven.

But if they talk about max. possible speedup with giving any delay then speedup = 4 (here).

Nok → High freq. means → more no. of instruction executed per clock or more no. of clocks generated.

Our aim is to get CPI ≅ 1 means clock per instruction = 1 but it is not possible due to hazards or dependencies.

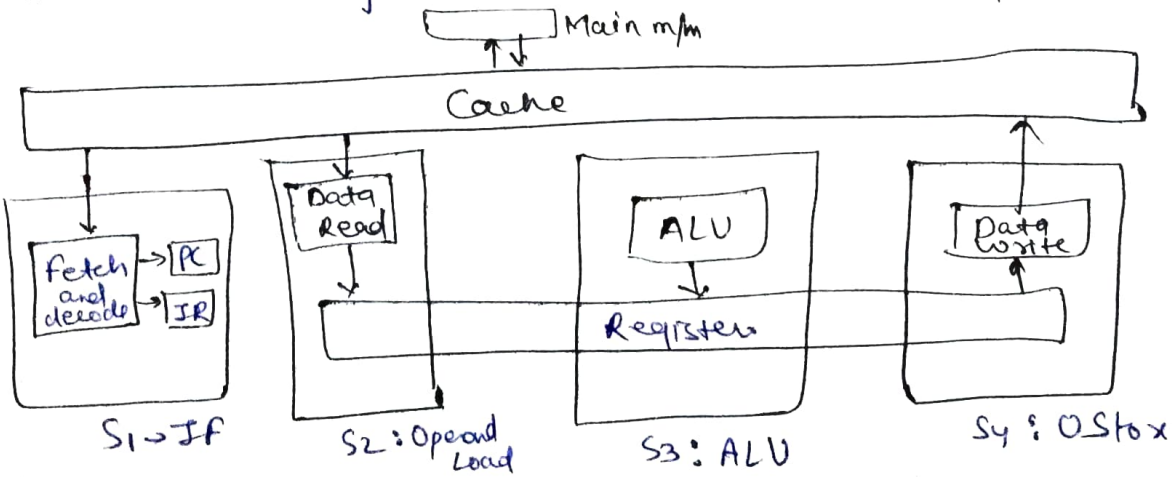
Hazards / dependencies :

1) Structural Hazards
2) Control Hazards
3) Data Hazards.

→ Structural dependencies occurs when two operations tries to access the same resource but only one can get it called as resource restriction.

we can avoid it using resource duplication or replication.

Ex



Pipeline

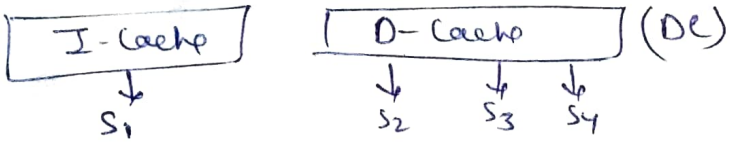
IF	OL	ALU	OS
	IF	OL	ALU	OS

At the same time two operations are trying to access the same memory.

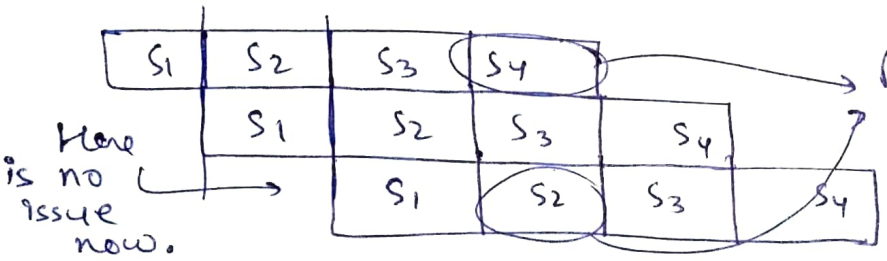
→ In resource duplication we will create multiple copy of Cache.

We devide it into 2 parts ① Instruction Cache (I-C)

② Data Cache (D-Cache)



But here, is again an issue



→ This way we need to devide cache into more parts, but we can divide it upto some extent only. So for this we have a solution that we should rearrange the stages in such a way that similar operations can work with synchronizations.

Not in gate but read it to

→ Now, we are discussing how (μP) handled this hazard.

We are considering here (μP) MIPS Rx000 or specifically MIPS R2000/R3000 as example, and these μP have the various pipelining instⁿ given below.

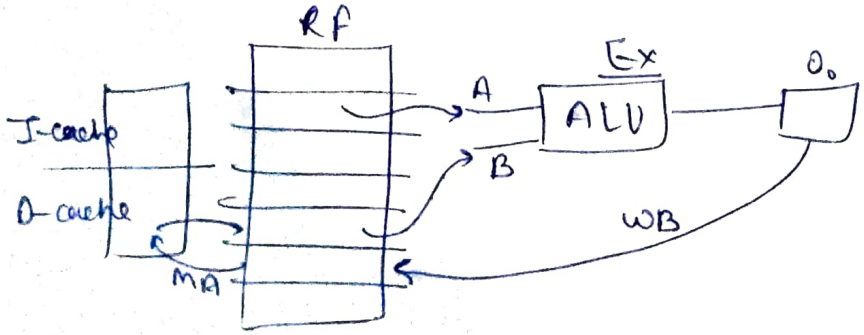
IF → Instruction fetching using I-cache.

RD → Operand loading (reading) from the register file (RF) while decoding the fetched instruction.

Ex → Data processing using the ALU and RF as needed.

MA → Operand accessing (load or store) using the D-cache.

WB → Operand storing (writing back) to RF.



Here we execute Data Read and Data write as single instruction MA.

Control dependency → When flow of control become unpredictable.

1. conditional branches.
2. Unconditional ——— .
3. function calls/return.
4. Program Control Instructions that may effect PC.

Throughput → Instruction per sec.

Throughput from CPI: Given CPI = 1

⇒ 1 clock → 1 Instruction

Say 1 clock = 1 μsec

⇒ 10^6 = 1 Instn

⇒ 1 sec = 10^6 Instruction = Throughput

Cost of time → flushing of pipeline and branch penalty. Due to branch penalty we can't make CPI = 1.

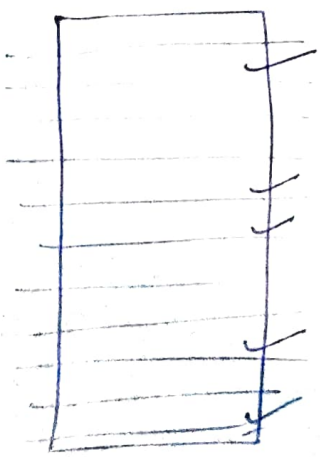
Cost of Pipeline → Stalling the pipeline (As you identify that the instruction is branching then hault the pipeline till last instruction get executed).

Cost of Pipeline → NOP (No operation instruction). → Our compiler will be smart enough.

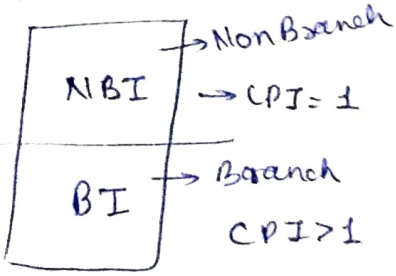
Best method → Reschedule or Rearrange → Our compiler should be smart to fill NOP operations with the valid operations so that it doesn't effect the sequence of operation of other instruction.

Analysis → Effect of branching on CPI.

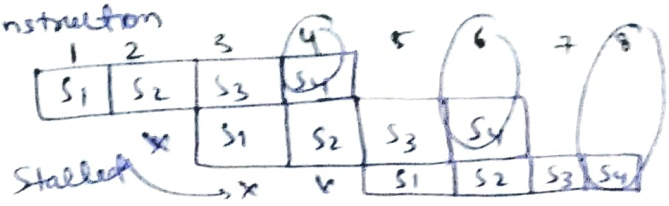
Problem: A program contains 1000 Instruction with 200 branching instruction. If result of branching Ins^n known after the second stage. What will be the effective CPI.



Assume branch instructions are distributed non-uniformly, but for our easyness we distribute in like



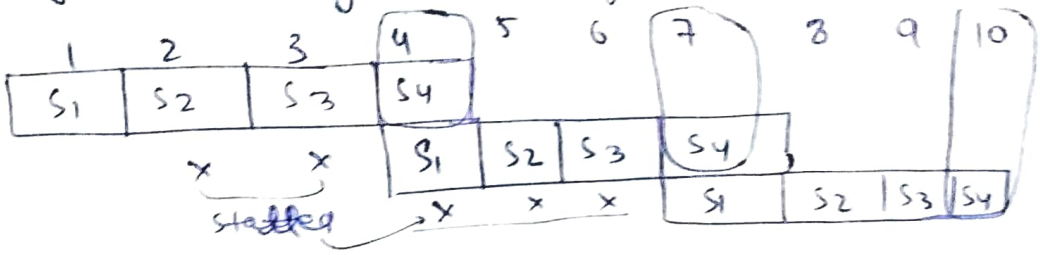
So, in branch instruction



→ CPI = 2, because every instruction came out after 2 clock cycles, after exit of 1st instruction.

So, avg CPI = $\frac{80\times1 + 20\times2}{100} = \underline{1.2}$

Suppose → on changing the question that result of branching get to know after 3rd stage then,



CPI = 3

So, avg CPI = $\frac{80\times1 + 20\times3}{100} = \underline{1.4}$

→ Similarly we can extend it up.

Generic formula for CPI

Branch
- Conditional
- Unconditional

→ If branching is unconditional then we will definitely move to another instruction Ex. goto 4. [100% chance]

→ On conditional branching its not certain bcoz if condition fails we will not move anywhere [if-else] [50% chance]

→ Unconditional branching have less penalty because we can directly access the next instruction just after decoding the branch instruction.

→ But in conditional you have to execute the instruction after fetching it.

Ex
JNZ 200
as you decode you will get the next address 200.

If (a>b)
you have to execute (a>b) after decoding it to verify the condition

So, conditional branching provide more penalty than unconditional branching.

So, CPI = 1 for Unconditional and conditional with true statements

If conditional is false then no penalty CPI = 1.

Case: Suppose there is 'n' instruction and 'p' is probability that it is conditional statement and 'q' is probability that condition is true. Pipeline have 'm' stages and we get to know about target statement after 'm' stages.

Soln Find avg. or effective CPI.

No. of conditional branch instructions = np

Total instructions which branch = npq

Here, CPI = m

Total instruction which doesn't branch = $(n - npq)$
$= n(1-pq)$

So, effective CPI = $\frac{n(1-pq) \times 1 + npq \cdot m}{n}$

$= \frac{n - npq + mpqn}{n}$

$= \frac{1 - pq + pqm}{}$

$= \underline{1 + pq(m-1)}$

'm' → after which we get to know about branch o/p.

Gate 2006: $f = 1\,GHz \Rightarrow T = \frac{1}{10^9} = 1\,ns$

$n = 10^9 \quad p = 0.2 \quad q = 0.8 \quad m = 3$

Time = $\underbrace{\left(10^9 \times \frac{20}{100}\right)\left(3 \times 10^{-9}\right)}_{\text{for conditional}} + \underbrace{\left(10^9 \times \frac{80}{100}\right) \times \left(1 \times \frac{1}{10^9}\right)}_{\text{for remaining statements}}$

$= 1.4\,sec$

v.9mp

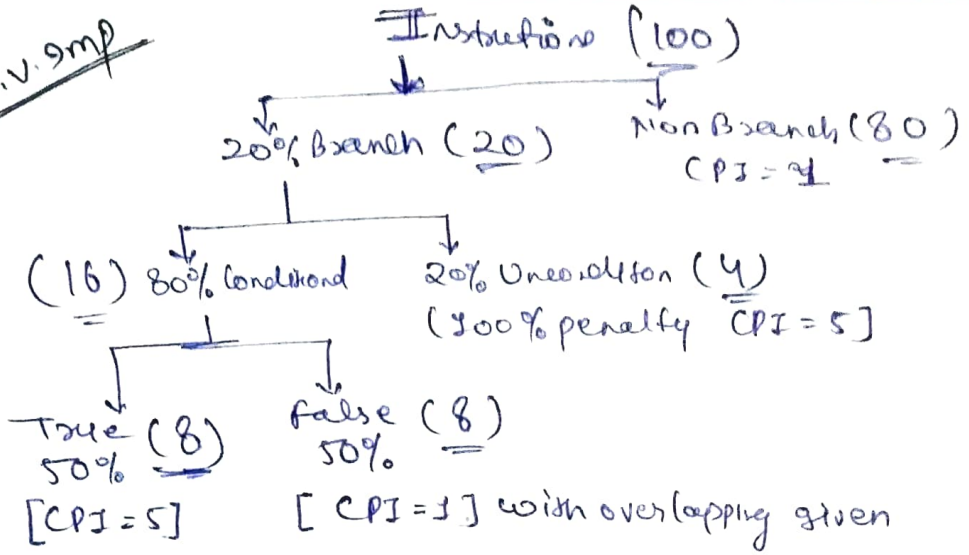
Gate 2000: (a) $T_{avg} = \frac{n \times 0.2 \times 5 \times 2 \times 10^{-9} + n \times 0.8 \times 1 \times 10^{-9} \times 2}{n}$

$= 2 \times 10^{-9} + 1.6 \times 10^{-9}$

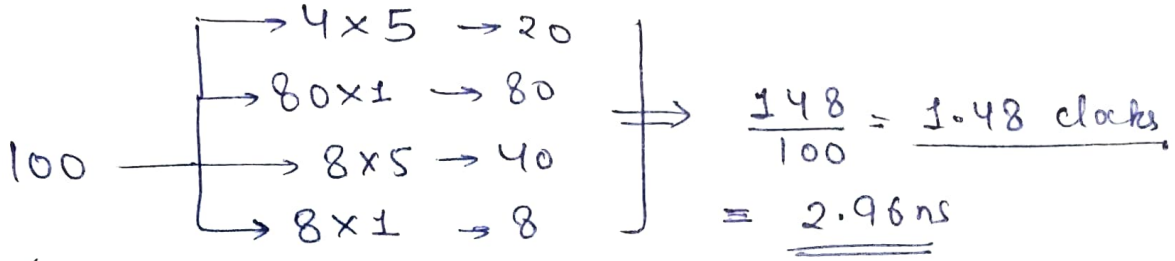
$= 3.6\,ns$

Penalty = $3.6 - 2 = 1.6\,ns$

(b) V.V.Imp



Says 100% → 100 Instructions.



Sometimes, time for Unconditional branches given separately.

→ Balanced stages means, all stages are taking same time.

Ways to avoid stall cycles

→ These ways works 80% of time not 100%.

① Branch Prediction & speculative execution.

→ flushing is involved

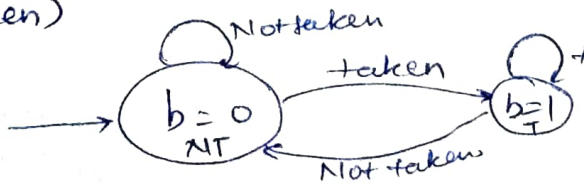
Static → Two strategies : Branch always taken, Branch never taken

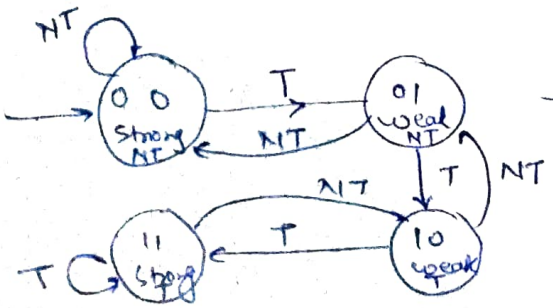
→ Dynamic Prediction

→ Bit implementation

→ Branch prediction Buffer Implementation.

In bit implementation if b=0 (Branch Not Taken), b=1 (branch is taken)





Delayed branching

	101	102	103	104	105	106	107	108	109
S_1	I_1			I_2					
S_2		I_1			I_2				
S_3			I_1			I_2			
S_4				I_1			I_2		

Suppose we can get the branched addr. after S_3. So next Instruction I_2 can't start before I_1 completes the S_3 stage. This waste lots of space in pipeling but we can use it using smart compilers. Smart compilers can perform Load and store kind of operations in vacant slots. So that whenever new Instn come, it will get execute smoothly.

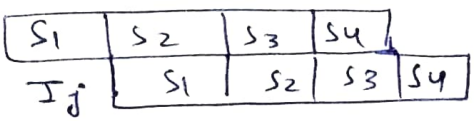
Since I_2 is delayed by 2 clock cyles that is why it is called as delayed branching.

→ Program meaning shouldn't Changes
→ A smart compiler needed.

Data Hazards / Dependency : Due to read/write data.

① RAW : Read after Write (True Dependency)
② WAR : Write — Read (Anti Dependency)
③ WAW : — — Write (Write/output dependency)

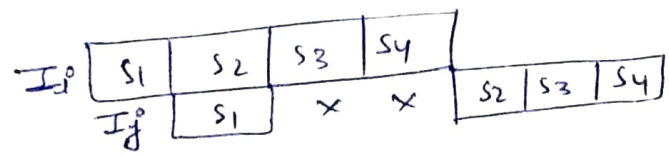
Ex-① I_i : $R_1 \leftarrow R_2 + R_3$
I_j : $R_4 \leftarrow R_1 + R_5$



Here I_j is reading R_1 which is written by I_i. So in pipeline we need to wait for I_j until I_i gets executed because I_i will write back R_1 in S_4. So final pipeline will be look like.

$S_1 \rightarrow$ IF
$S_2 \rightarrow$ Operand fetch
$S_3 \rightarrow$ ALU
$S_4 \rightarrow$ WB

Read after I_i write



Ex-2 I_i : $R_1 \leftarrow R_2 + R_3$
I_j : Load m, R_2

Here, value of m is need to be loaded in R_2. Means I_j need to execute before I_i.

③ WAW: $I_i : R_1 \leftarrow R_2 * R_3$ Say order of execution is $I_i \rightarrow I_j$
$I_j : R_1 \leftarrow R_5 + R_6$

But, pipeline get designed in such a way that order of execution becomes $I_j \rightarrow I_i$. so it will cause an error because we are writing to same register.

We can identify the dependency as —

RAW: Ex: $I_i : R_1 \leftarrow R_2 + R_3$ such that I_i occurs before I_j
$I_j : R_4 \leftarrow R_1 + R_5$

$D(I_i) \rightarrow R_2, R_3 \quad R(I_i) \rightarrow R_1$
$D(I_j) \rightarrow R_1 R_5 \quad R(I_j) \rightarrow R_4$

If $R(I_i) \cap D(I_j) \neq \phi$ then there is a dependency.

WAR: $I_i : R_1 \leftarrow R_2 + R_3$
$I_j :$ Load m, R_2 (or) $R_2 \leftarrow R_5 + R_6$

Next instruction is try to update the value, so read the update value of first then perform I_i.

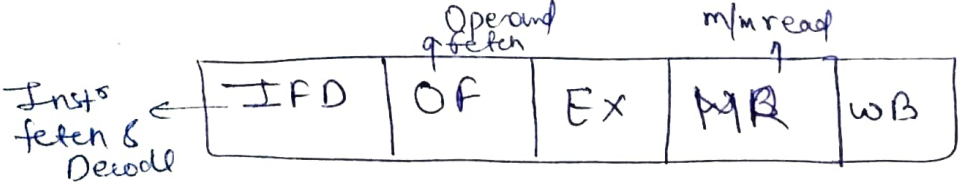
$D(I_i) \cap R(I_j) \neq \phi$

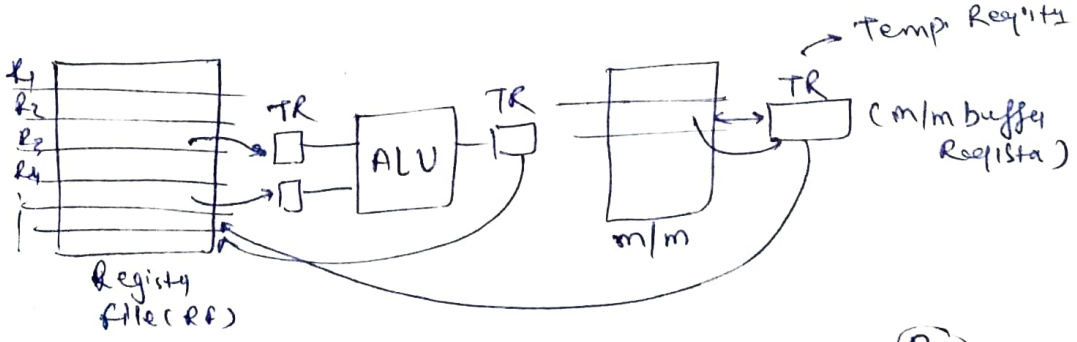
$I_i \rightarrow I_j$ Ordy

WAW:

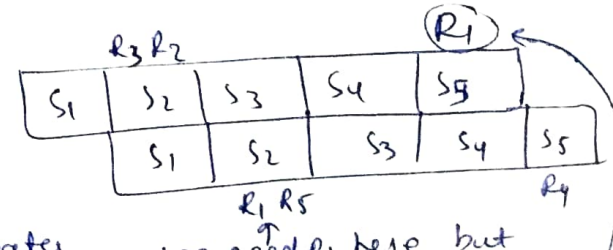
$R(I_i) \cap R(I_j) \neq \phi$

Operand forwarding: (Short-circuiting)





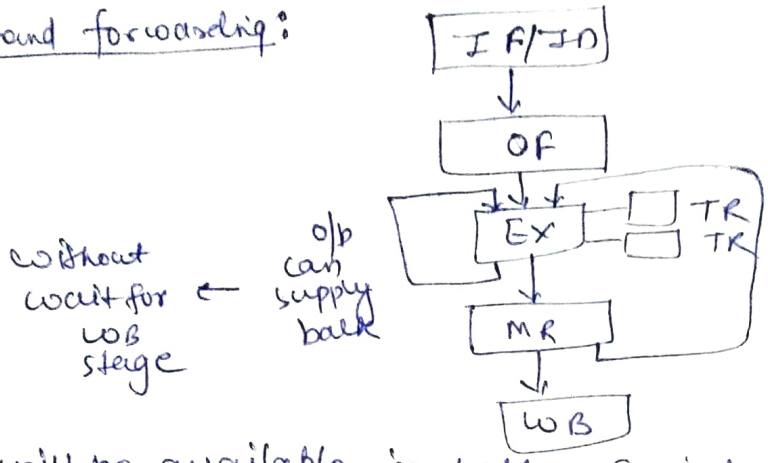
Say $I_1 : R_1 \leftarrow R_2 + R_3$
$I_2 : R_4 \leftarrow R_1 + R_5$



so it creates a delay because $S_2 S_3 S_4$ of I_2 gets haults

we need R_1 here but we will get R_1 in S_5

So in operand forwarding:



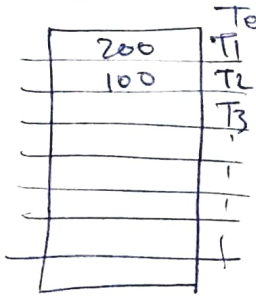
So, R_1 will be available just after S_3 instead of S_5 in previous example.

So, we can reduce stalls upto 100% using this technique.

Gate-2010: Solution given in videos.

Operand forwarding used to resolve RAW hazards,
To resolve WAW, WAR hazards we have concept of register renaming.

I_1: $R_1 \leftarrow R_2 * R_3$
I_2: $R_4 \leftarrow R_4 + R_5$



say I_1 set value $R_1 \rightarrow 200$
I_2 set value $R_2 \rightarrow 100$
Time taken (I_1) > Time (I_2)

But I_2 needs to write the final value and in this case I_1 will write final value. So resolve this we have temporary register. We will rename them as $T_1 \leftrightarrow R_1$, $T_2 \leftrightarrow R_2$ and keep the values in temp. reg. instead of keeping them in R_1 and when both value will write inside temp. register then we will upload the desired result in R_1.