- Performance :- Arithmetic/Logical :- 50%
  Load :- 20%
  Store :- 10%
  ~~Branch~~ BEq :- 15
  Jump :- 5%

For multicycle, Performance = N × ~~4.15 × 3ns~~ 4.15 × 3ns
= 12.45

single cycle = 12.5

- Depending on fractions of instruction types, either of single cycle or multicycle implementation.
- In spite of fewer registers, single-cycle is more expensive overall in terms of hardware.
- Single cycle :- Lowest possible CPI (1)
  Multi cycle cycle time
- Possible to strike a balance :- eg- 2 cycle implementation.

→ Pipelining

- Begin next instruction before previous is over, using the idle resources.
- In RISC, subtasks in every instruction are very similar - fetch instruction, fetch operands, alu operation, etc.

- If is possible to get least possible cycle time (like multicycle)
  - If we assume all instructions to be independent, then we can do pipelining and then we will get cycle time equal to max{sub-task time}

Mealy :- Output Logic & Next-state logic depend on both input and current state

Moore :- NSL is same as before
OL depends only on current state.

→ 5 subtasks
IF ID/OR
1 R-type :- Instruction fetch, Instruction Decode / Operand Read,
EX WB
Execute, Write Back

Mem
2 LOAD :- IF, ID/OR, EX, Memory Read, WB

3 Store :- IF, ID/OR, EX, Mem

4 BEQ :- IF, ID/OR, EX

5 JUMP :- IF, ID, EX

exhaustive set of sub-tasks

- First 4 cycles, no instruction will be completed.
After that, one instruction will be completed every cycle.
(for independent instructions and if instructions are consecutive from top to bottom (no control flow changes in between))

→ Performance = N × 1 × 3ns
CPI Cycle time.
CPI ≈ 1 Cycle time = max(subtask time)

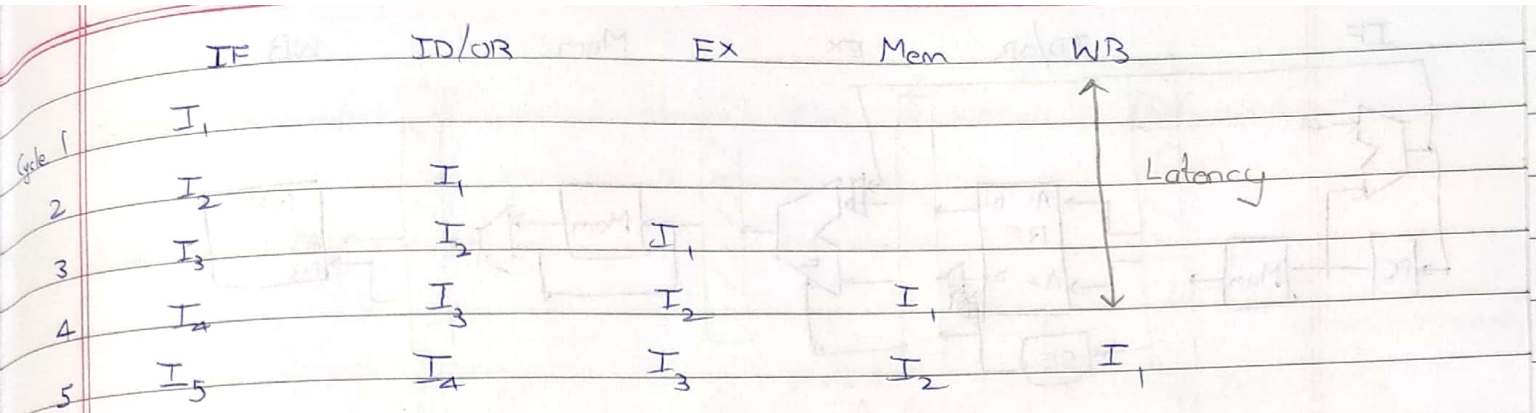
- Every instruction will go through all 5 stages
IF → ID/OR → EX → Mem → WB
eg - JUMP will waste time in Mem & WB stages

- Wasting time :- Do the job in each state, but don't write to any register (using control signals on registers)

- Making 5 stages common for all instructions converts each stage from a ~~Mealy~~ Moore machine (which depended on instruction) to a ~~Moore~~ Mealy machine (which does not)
Moore :- Control signals generated depended only on current state.
Mealy and op-code

- To use pipelining, all instructions must use similar type of stages, and in same order too.

Cycle 1
2
3
4
5



→ How do you design a pipeline (determine the required subtasks)?

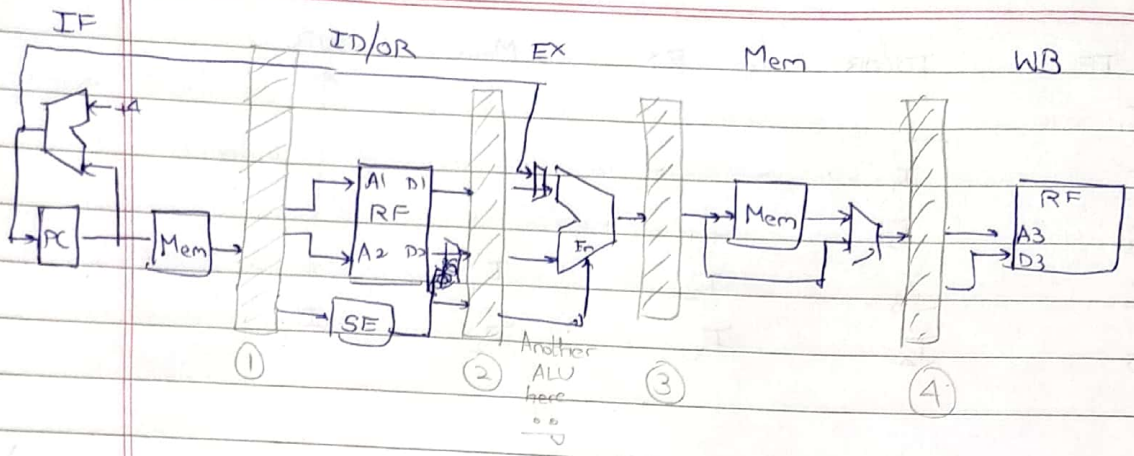
∴ - Start from multicycle :- We will have minimal resources used.
Could be an issue if resource is shared by subtasks

eg- In multicycle, same memory is used by IF & Mem.

- ∴ Start with single cycle to get maximal resources.

→ We generate control signals through combinational decoder of single cycle implementation (like always), and "remember" them in a register
The control signals for each instruction must be propagated consecutively to each task along with the instruction itself.
The respective subtask uses whichever control signals it needs.
- So every subtask will have a register (to store control signals) of the size equal to number of control signals flowing from previous state to that state
~~Register in subtask 1 will have size equal to total number of control signals.~~
- ~~PC is also propagated through the stages, as it will be required for returning control flow changes~~

Instructions move in lock step manner (= FIFO, no jumping)



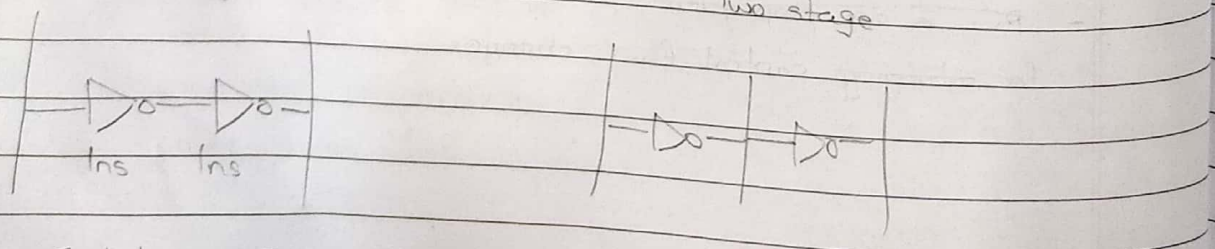
- Signals crossing interface after IF :
  Instruction (32 bit), PC+4 (32 bit), etc

- Interface 2 :-
  PC+4 (32-bit), Operand 1 (32-bit), Operand 2 (32), Sign extended Imm (32), ALU control signals, destination register ~~addr~~ address, etc

- Interface 3 :-
  ALU output (32), A3 address etc

- Interface 4 :-
  Destination register address, data to be written, etc

→ How many stages to divide the instruction into?
- More no. of (smaller) stages will give lower cycle time ☺
  - Becomes costlier, as you need more registers for interfaces ☹
  - Performance ↑, CPI still ≈ 1



Cycle time = 2ns

Cycle time = 1ns

22/10

- Choose the value of n (no. of stages) that maximize Performance/Cost

Cost = $G + (n-1)k$ (k: avg cost of interface registers)

Performance = $\frac{1}{\frac{I}{n} + q}$ ← Some small delay (inherent)

($\frac{I}{n}$: time without pipelining)

($G + (n-1)k$: cost without pipelining)

* $\frac{\partial}{\partial n} \frac{1}{\left(\frac{I}{n} + q\right)\left(G + (n-1)k\right)} = 0$

22/10

→ Our assumptions :- (Hazards)

eg Dependent instructions :- I1 : $r_1 = r_2 + r_3$

I2 : $r_4 = r_1 + r_5$

In ID/OB stage, I2 will read incorrect r_1

- Detecting violation of independence of consecutive instructions :-

Source registers in ID/OB stage instruction = Destination register of Ex stage instruction.

i.e → $(RS1^{ID/OR} == RD^{Ex}) \;||\; (RS2^{ID/OR} == RD^{Ex})$

- If this violation occurs, 'freeze' I2 in the ID/OB stage for 3 cycles (until I1 completes). Then resume.
  - On every violation, we are losing 3 cycles
  - eg - If 10% instructions have this 'immediate' dependency.

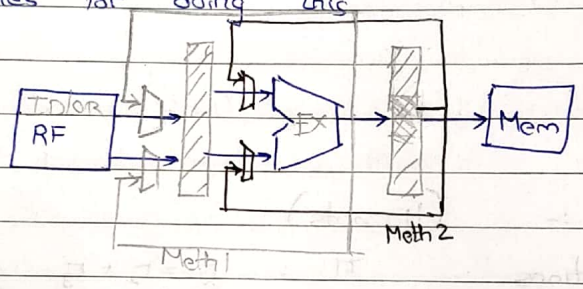
New CPI = $1 + 3 \times 10\% = 1.3$

I1	—	—	—	—	
I2	I1	—	—	—	freeze
I3	I2	I1	—	—	
I3	I2	—	I1	—	
I3	I2	—	—	I1	
I3	I2	—	—	—	resume

- The result of I_1 is getting written back in stage 5. (WB). But it is being produced in stage 3 (EX) itself. This result will be stored in the pipelining interface register at the boundary of EX and Mem. Let I2 fetch the result from there.

I_1 — — — —
I_2 I_1 — — —
I_3 I_2 I_1 — —
I_4 I_3 I_2 I_1 —

Two possibilities for doing this



→ Non-consecutive dependency.

I_1 :- $r_1 = r_2 + r_3$
I_2 :- $r_4 = r_5 + r_6$
I_3 :- $r_7 = r_1 + r_8$

23/10

1 $(RS1 \text{ or } RS2)^{ID} == RD^{Mem}$:- Where to fetch operand from?

2 $(RS1 \text{ or } RS2)^{ID} == RD^{WB}$:- Obtain result from rf-d3 of WB stage.

→ Multiple ~~write~~ conflicts.

$r_1 = r_2 + r_3$
$r_1 = r_4 + r_5$
$r_6 = r_1 + r_7$

- Priority order :- For conflicting dependencies, fetching result :-

  EX > Mem > WB
  (EX: newest instruction; WB: oldest instruction)

→ Load instruction produces result in Mem stage.

$I1 :- r_1 \leftarrow (R2 + 50)$

$I2 :- r_3 = r_1 + r_2$

When I2 is in ID/OR, the required result has not been produced yet. Forwarding will not work. We have to wait (freeze) for one cycle at least in such cases.

1	–	–	–	–	
2	1	–	–	–	
3	2	1	–	–	~~freeze~~
4	3	2	1	–	freeze .... 'Bubble' or 'Stall'
4	3	2	–	1	resume
5	4	3	2	–	

23/10 – We are losing one cycle for every instruction of this type.

eg – α% AL instructions, β% Load (of which λ% have such immediate hazard), γ% Store, δ% Branch.

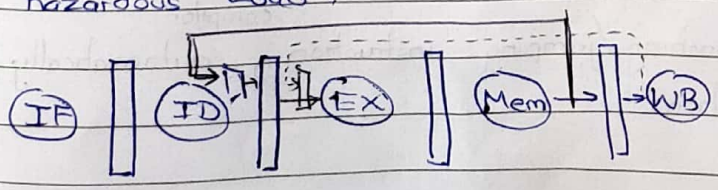
New CPI = $1 + \beta\lambda(1)$

- Bubble or Stall :-

All interface registers should have an enable signal, which is made zero to stop updation

Make enable of PC zero, too.

- For hazardous Load.



→ Minimizing the damage of Load hazard :-

eg $A = B + C + D$ → load r_1, B

$T1 = B + C$

$A = T1 + D$

A, B, C, D, T1 are in memory

load r_2, C

add r_3, r_2, r_1

store r_3, t_1

load r_4, D

add r_5, r_4, r_3

store r_5, A

Smart compiler →

load r_1, B

load r_2, C

load r_4, D

add r_3, r_2, r_1

store r_3, t_1

add r_4, D

store r_5, A

'Compiler optimization'

→ Change in Control Flow

eg I_1 contains JUMP.

∴ I_2 is not supposed to be executed.

	IF	ID/OR			
a)	1	—	—	—	—
b)	2	1	—	—	—

- Every interface register should have an extra bit, which will decide if an instruction is to be disabled.

If I_1 is JUMP, then the disabled I_2 will go through all stages, but will not write to any registers, flags

Meth 2 Delayed Branch Execution

✗ After every branching/jumping instruction, compiler automatically adds a NOP instruction.

P.T.O.

If I2 is JUMP,

- If I2 does not depend on I1 (unconditional jump), move I1 to just after I2.
- If I2 depends on I1 (conditional jump), add a NOP instruction to just after I2.

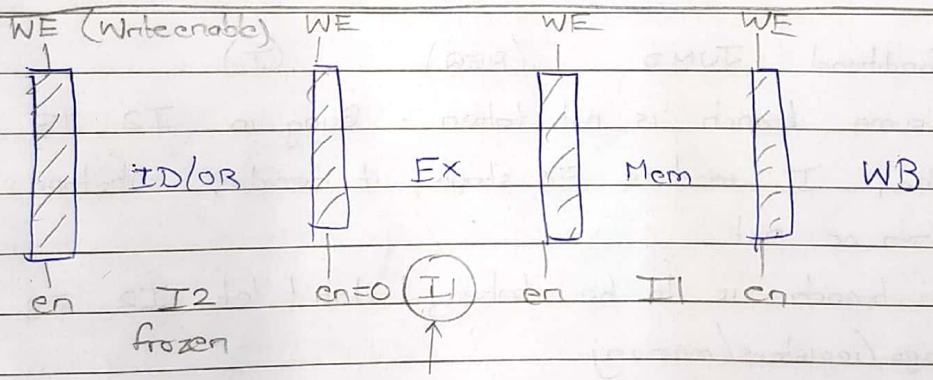
→ Conditional branching

eg I1 is BEQ

1 _ _ _ _

2 1 _ _ _

3 2 1 _ _



Comes here because interface register 2 was not updated in previous cycle. This extra instruction needs to be killed.

- Instead of using $(R_{S1}^{ID} == R_D^{EX}) \,||\, (R_{S1}^{ID} == R_D^{Mem}) \,||\, (R_{S1}^{ID} == R_D^{WB})$,

we now need more checks because I1 is duplicated

Use :- $(R_{S1}^{ID} == R_D^{EX}$ && $R_{WE}^{EX} == 1)) \,||\, (R_{S1}^{ID} == R_D^{Mem}$ && $R_{WE}^{Mem})) \,||\, (\ldots\ldots)\ldots$

&& it should be LOAD instruction

&& MEM_RD == 0

Only LOAD instruction reads from data memory.

'Stall' Requirement detection

If this condition is met, $IF_{EN} = ID/OR_{EN} = PC_{En} = RF_{WR} = 0 = Flag_{WB}$

$Mem\_WR = 1$

→ Unconditional JUMP

Let all interface register have a bit 'valid'
If I1 is found to be jump, make 'valid' of I2 (in I1) '0'.
If valid = 0, don't let that instruction write to flags, registers, memory.

☆ Assume that for JUMP instruction, we can update PC as soon as ID stage (using separate ALU)

→ Conditional JUMP (BEQ) (I_1)

- Assume branch is not taken. Bring in I2, I3 serially. When I_1 reaches EX stage, it decides whether branch is taken or not.
- If branch is to be taken, don't let I2, I3 update flags/registers/memory
  - Penalty of 2 cycles ∴
  * Most of the times, branches are taken
- Assume that if branch is to be taken, PC is updated at the end of EX stage.

• Our initial assumption cannot be to take the branch, as we don't know yet where to branch to.

• $\alpha \rightarrow$ AL, $\beta \rightarrow$ Load, $\gamma \rightarrow$ Store, $\delta \rightarrow$ Branch (Conditional), $\epsilon \rightarrow$ Jump
β% ↓ independency; δ ↓ 70% branches are taken

$$CPI = 1 + \beta\lambda(1) + \epsilon(1) + \delta(0.7)(2)$$
* Why