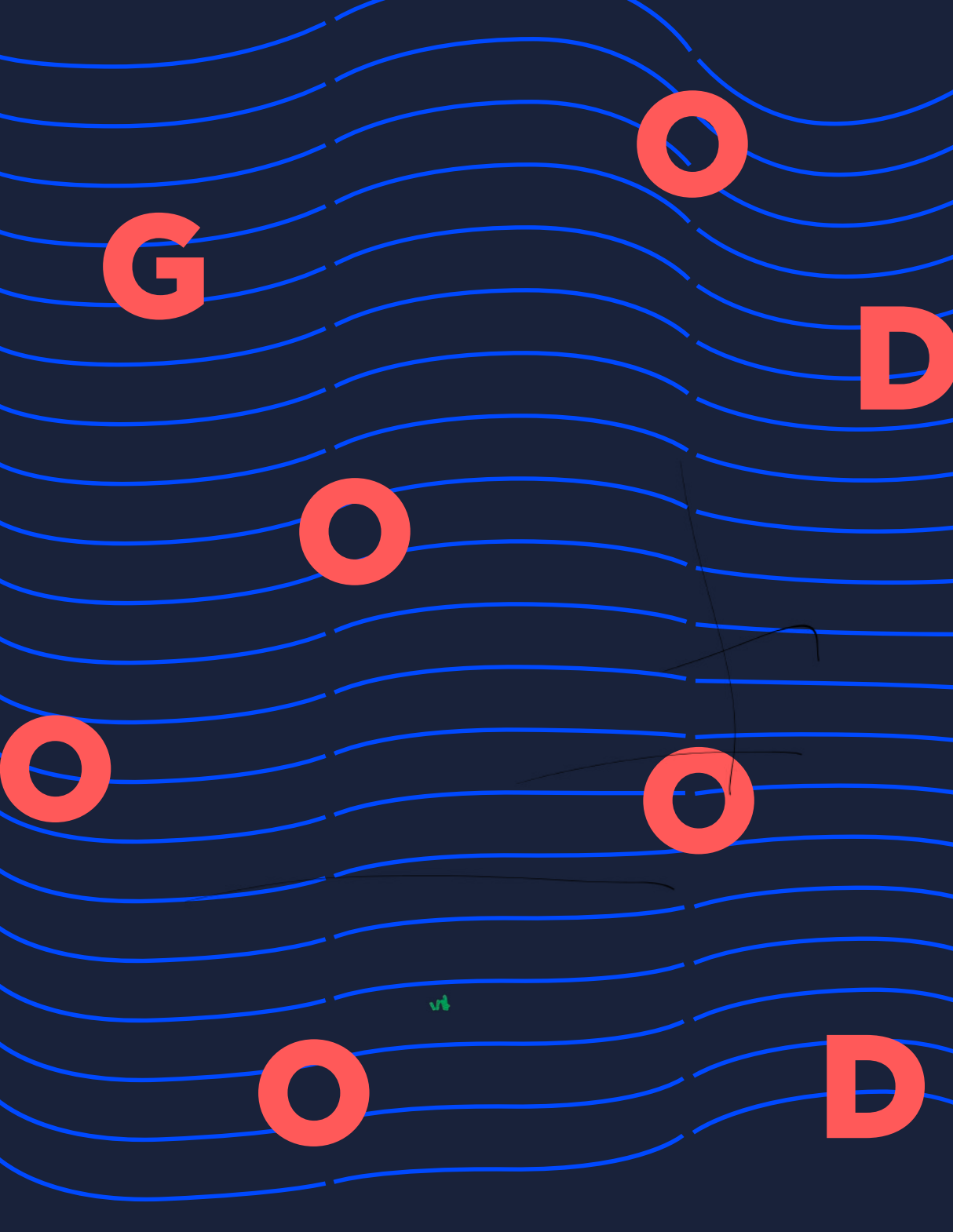
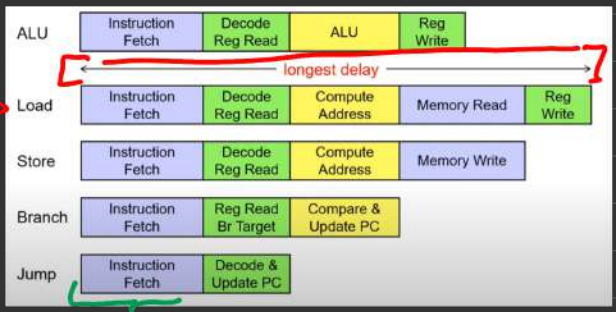

G
O
D
O
O
O
O
D

11.1 Single cycle (SC) VS multi cycle (MC) Processor



cycle time SC

- Our design currently functions to execute 1 instruction per cycle
- Thus the cycle time must accomodate for the Longest operation redundantly

Solution

- Reduce clock cycle
- Every instruction -> diff no. of cycle
- Imagine every phase -> one cycle

Multicycle Implementation

- Break instruction execution into five steps
  - Instruction fetch
  - Instruction decode, register read, target address for jump/branch
  - Execution, memory address calculation, or branch outcome → ALU stage
  - Memory access or ALU instruction completion → Load/Store or WB ALU output
  - Load instruction completion
- One clock cycle per step (clock cycle is reduced)
  - First 2 steps are the same for all instructions

Instruction	# cycles	Instruction	# cycles
ALU & Store	4	Branch	3
Load	5	Jump	2

Performance Example

- Assume the following operation times for components:
  - Access time for Instruction and data memories: 200 ps
  - Delay in ALU and adders: 180 ps
  - Delay in Decode and Register file access (read or write): 150 ps
  - Ignore the other delays in PC, mux, extender, and wires
- Which of the following would be faster and by how much?
  - Single-cycle implementation for all instructions
  - Multicycle implementation optimized for every class of instructions
- Assume the following instruction mix:
  - 40% ALU, 20% Loads, 10% stores, 20% branches, & 10% jumps

• CPI SC: 1

For MC:

• We need a $ to:
Store instruction } making it Multicycle
Store ALU Reg
Store ALU Vars

Instruction Class	Instruction Memory	Register Read	ALU Operation	Data Memory	Register Write	Total
ALU	200	150	180		150	680 ps
Load	200	150	180	200	150	880 ps
Store	200	150	180	200		730 ps
Branch	200	150	180 ← Compare and update PC			530 ps
Jump	200	150 ← Decode and update PC				350 ps

ALU:
Instruction Fetch = 200 ps
Decode/Read File = 150 ps
ALU Execution = 180 ps
Writing back = 150 ps
} 680 ps

Decode overlaps with ctrl parallel, Reg File takes longer
So ctrl is ignored

Instruction Class	Instruction Memory	Register Read	ALU Operation	Data Memory	Register Write	Total
ALU	200	150	180		150	680 ps
Load	200	150	180	200	150	880 ps
Store	200	150	180	200		730 ps
Branch	200	150	180 ← Compare and update PC			530 ps
Jump	200	150 ← Decode and update PC				350 ps

- For fixed single-cycle implementation:
  - Clock cycle = 880 ps determined by longest delay (load instruction)
- For multi-cycle implementation:
  - Clock cycle = max (200, 150, 180) = 200 ps (maximum delay at any step)

$$CPI = Freq_{ALU} \cdot CPI_{ALU} + \dots$$

$$CPI = 0.4 \cdot \left\lceil \frac{680}{200} \right\rceil + 0.2 \left\lceil \frac{880}{200} \right\rceil + 0.1 \left\lceil \frac{730}{200} \right\rceil + 0.2 \cdot \lceil 530/200 \rceil + 0.1 \lceil 350/200 \rceil$$

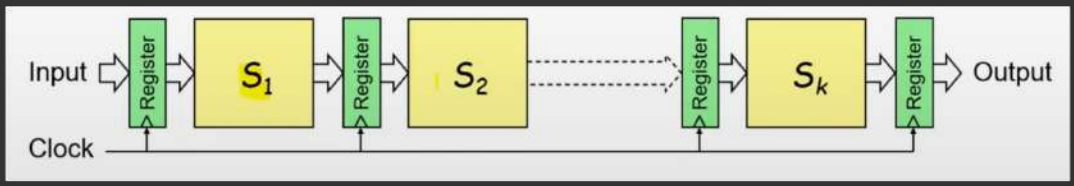
Analysis:

- steps and time per step, where max step time = clock cycle
- Time per instruction
- $CPI_i = \left\lceil \frac{\text{time instruction}_i}{\text{cycle time}} \right\rceil$
- CPI weighted avg

- Average CPI = 0.4×4 + 0.2×5 + 0.1×4+ 0.2×3 + 0.1×2 = 3.8

11.2 Serial Vs Pipline

In between every stage/step a register



- Let τ_i = time delay in stage S_i
- Clock cycle $\tau = \max(\tau_i)$ is the maximum stage delay
- Clock frequency $f = 1/\tau = 1/\max(\tau_i)$
- A pipeline can process n tasks in $k + n - 1$ cycles
  - k cycles are needed to complete the first task
  - $n - 1$ cycles are needed to complete the remaining $n - 1$ tasks
- Ideal speedup of a k-stage pipeline over serial execution

$$S_k = \frac{\text{Serial execution in cycles}}{\text{Pipelined execution in cycles}} = \frac{nk}{k + n - 1} \qquad S_k \to k \text{ for large } n$$

- same cycle time
- larger S_k ↑ serial or ↓ pipline

MIPS Processor Pipeline

5 steps → 5 stages:

1. IF: Instruction Fetch
2. ID: Decode, Read, Instruction Addressing
3. EX: Execution, Mem Addressing, CMP Branch 0
4. Mem: Mem access for load/store
5. WB: WB to $ or PC

- Consider a 5-stage instruction execution in which …
  - Instruction fetch = ALU operation = Data memory access = 200 ps
  - Register read = register write = 150 ps
- What is the clock cycle of the single-cycle processor? 1
- What is the clock cycle of the pipelined processor? 2
- What is the speedup factor of pipelined execution? 3

3. CPI = 1 after Filling Pipeline

1. The clock cycle of the SC CPU is equal to the instruction that takes the most time: LW = Fetch + Decode + Execute + MemRead + WB
   = 150 + 200 + 200 + 200
   = 950 ps or 1.2 GHz

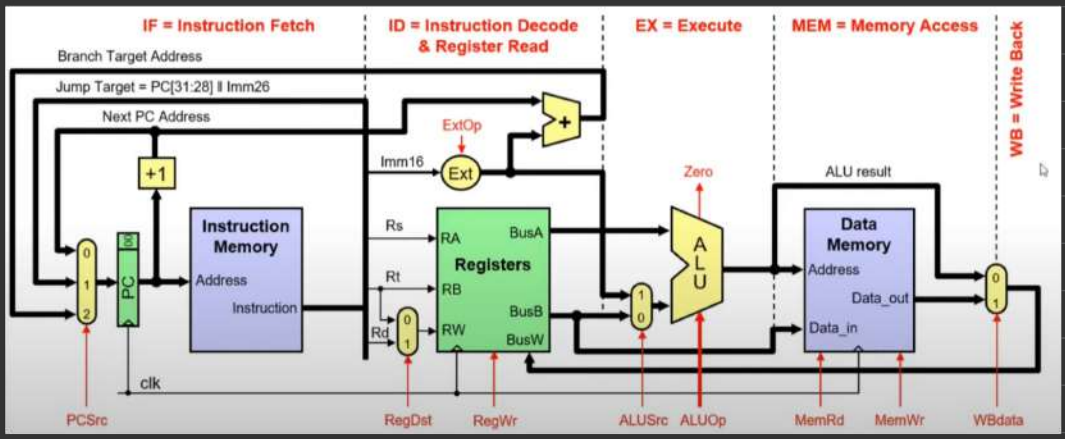
2. The clock cycle of Pipeline is equal to the step that takes longest: 200 ps

$$\text{Speed-up} = \frac{\text{Serial clock cycle}}{\text{Pipeline clock cycle}} = \frac{400\ ps}{200\ ps}$$

$$= 4.5$$

12.1 Pipeline Data Path

Distinguish different stages

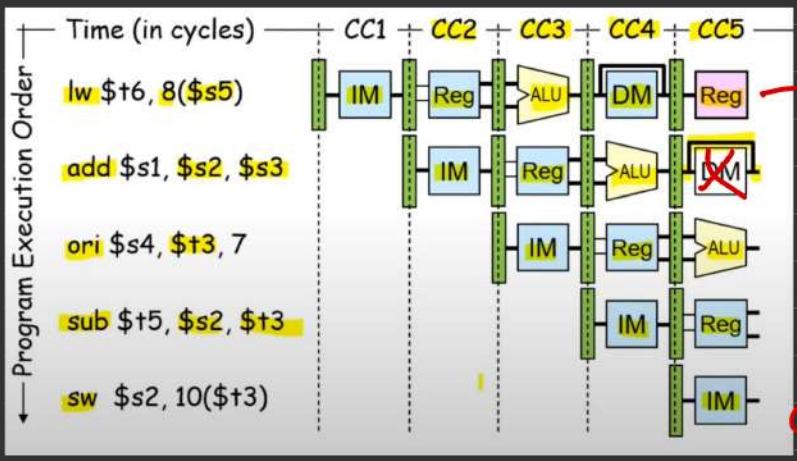


How to make it pipelined: add $ between different stages

↳ carry as long as needed only!!

Problem.
The Rt is constantly changing, so by the time in during WB, the Rt would be diff
So we save Rt through Propagation

Graphical Representation



Maximum of 5 instructions simultaneously since its a 5 stage Pipeline

Some instructions do nothing at at some stages, but don't jump at to keep synchronization.

CTRL

Same ctrl signals, but Propagated

J & Addressing

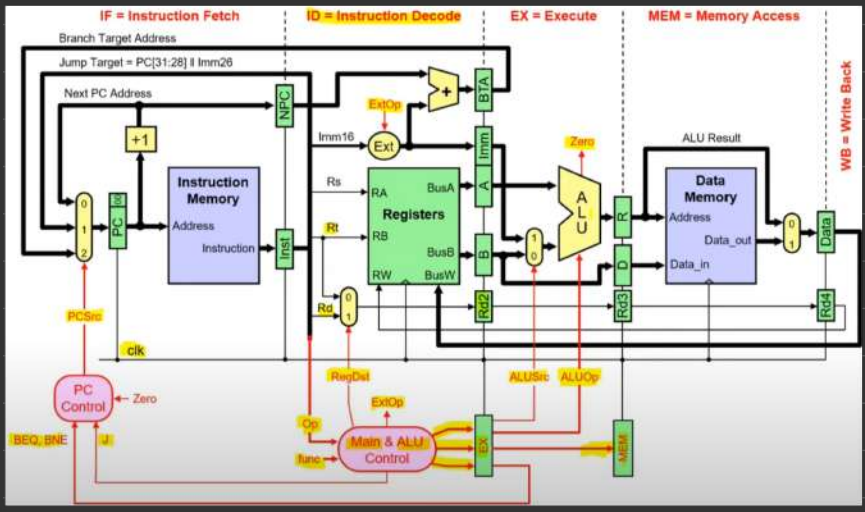
Do we need to store the destinations of PC:

no delay no $ since ID

J, no, because jump decision and address are decided at ID, the jump is directly linkes to MUX

one delay one $ since ID

B, Since decision is made in EX, we must delay the address being at MUX by 1, so one Branch destination $.



Ctrl Signaling

Ctrl For Each Stage: ■ outputs generates but not Pipelined ■ outputs generates and Pipelined ■ Ctrl Signals needed

1. IF: Nothing

2. ID:

jumpDist directly connected to PC Mux
branchDist # Must be saved For branch decision in Ex
PC+1 # needed in MEM at WB Mux 2
LOI immediate # needed in MEM at WB Mux 3
Data1
Data2
EXOP
ALU Src

3. EX:

Zero not in $, directly connect to PC ctrl
Result # connected to Mux 03 in case of Jr
ALUOP

4. MEM:

MemRd
MemWr
WB data
Data

Structural Hazards

When instructions attempt to write to the register file simultaneously

↳ This is why we don't jump stages; all writings at WB stage ✓

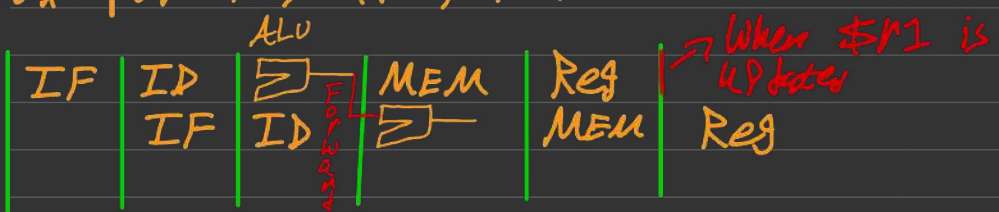
Data Hazards

1. Read after Write (RAW)

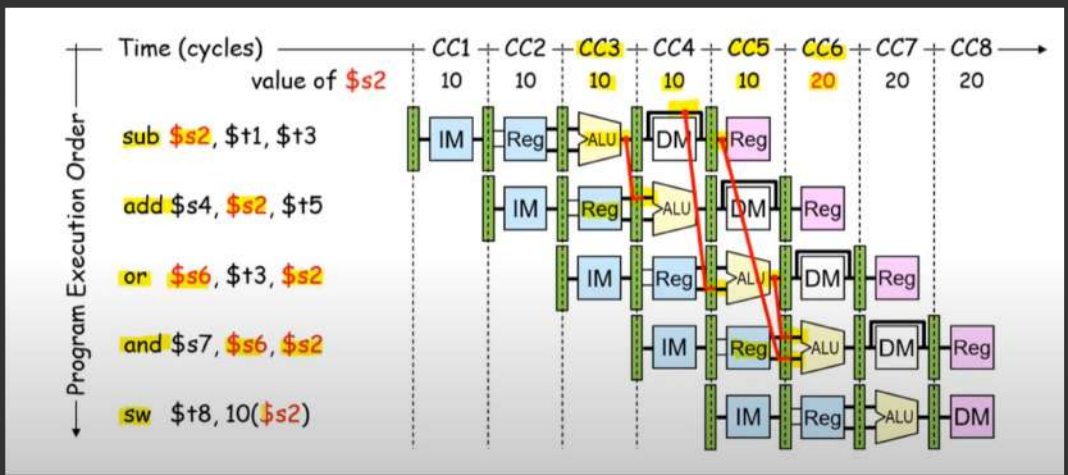
Ex.

```
0x0 | add $r1, $r2, $r3
0x1 | ori $r5, $r1, $r4
```

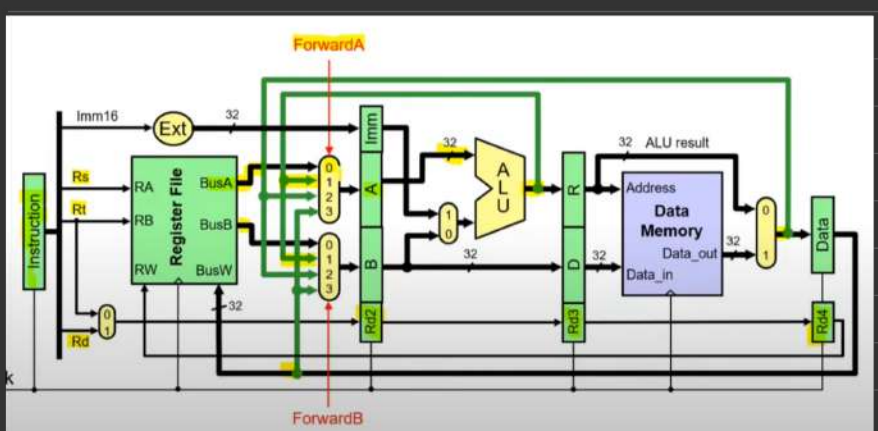
data dependency



Cont'd... Ex.



- Detect it
- Adjust data1 & data2

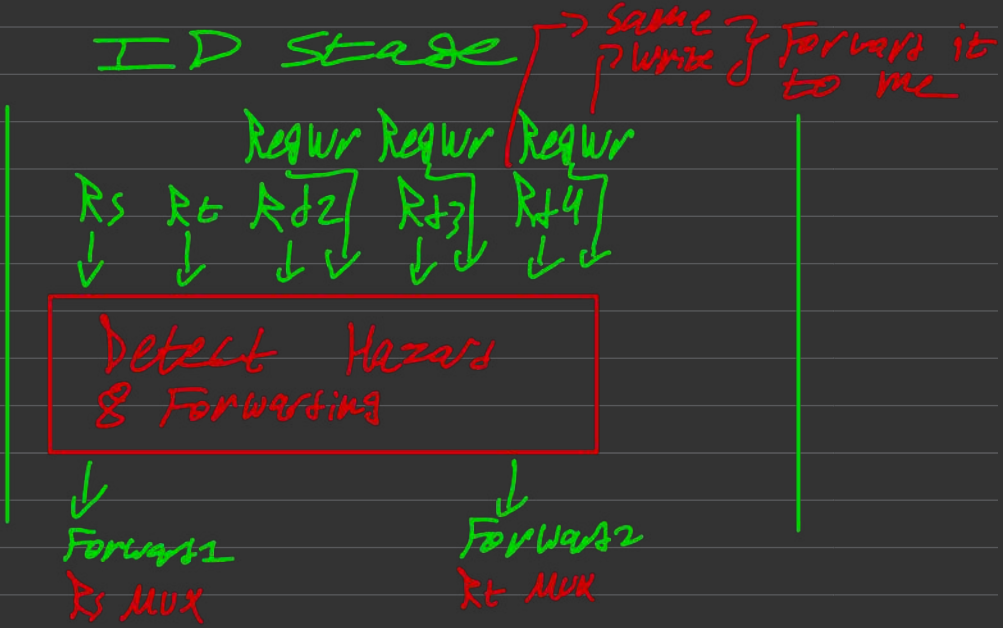


Detection:

IF (Rs == Rd2) { Forward = 1;
elif (Rs == Rd3) { Forward = 2;
elif (Rs == Rd4) { Forward = 3;
else { Forward = 0;

happens in decoding phase

RAW Detection & Forwarding



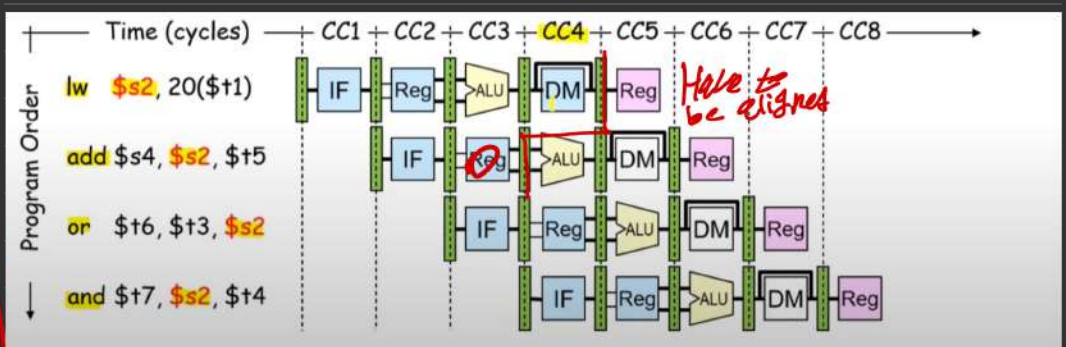
Other Delays

becomes 1

- Load can't be solved by forwarding but the RAW ctrl can still detect this hazard since the lw instruction:

1. The Rd of lw is Rs/Rt of prior instruction
2. RegWrite == 1
3. MEMRd == 1

But MEMRd == 1 → 1 & 2. So MEMRd EX is enough



So how can we align this?

Stall one stage

Recycling signals

→ IF (MEMRd >> 8 (ForwardA^B == 1))

Stall

Notes.

MEMRs

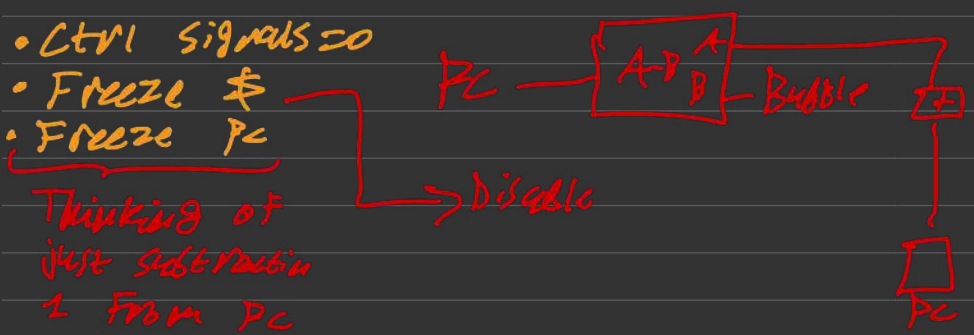
- Stall only needed if LW and immediate next instruction is dependent

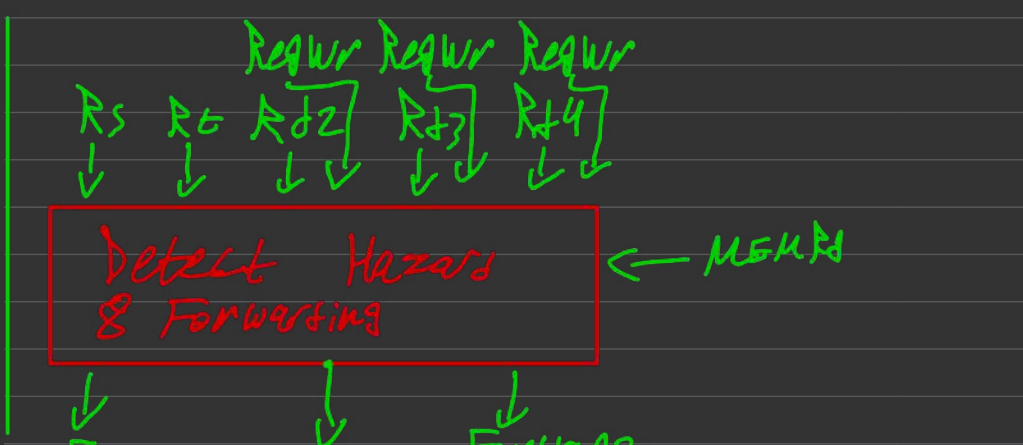
forwards = 1

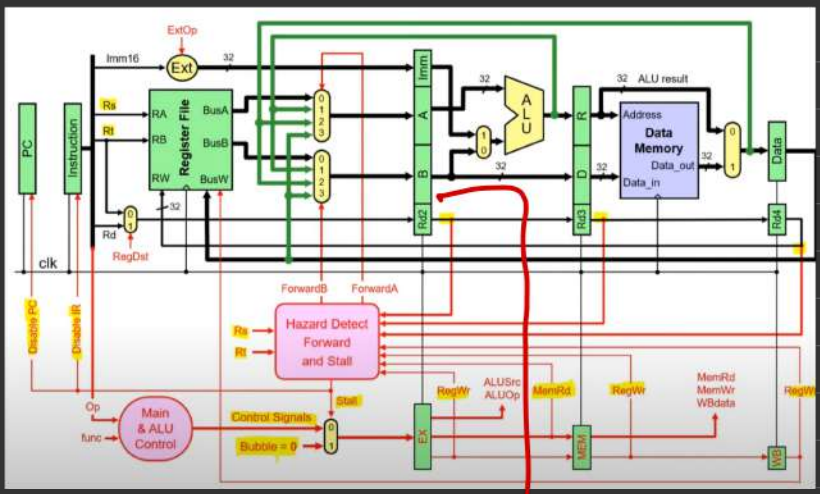
- Dependency occurs on if, WLOG, $Rs == $Rs AND Rs write at Ex, Mem, or WB
- Think LW to ALU
- LW result doesn't mature until Mem so stall

Stall

- Insert a bubble into the EX stage after a load instruction
  - Bubble is a no-op that wastes one clock cycle
  - Delays the dependent instruction after load by one cycle
    - Because of RAW hazard







Why not disabled? After 1 clock pulse it will be forwarded the address of MEM along with flushed signals, but in 2nd pulse it will be forwarded MEM Result

We don't have Structure hazards: "out put dependence" Since all writes take place in last stage

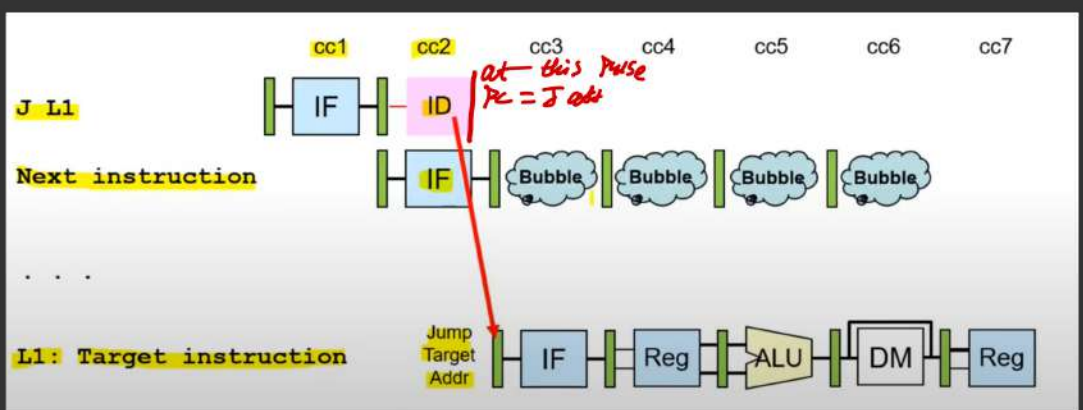
Ctrl Hazards

J-Type

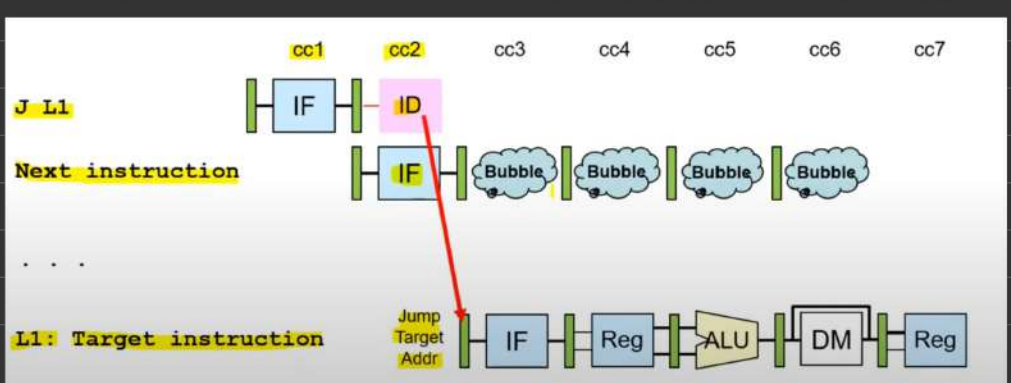
Since jump occurs at ID, we have 1 extra cycle # one redundent instruction in pipeline

B-Type

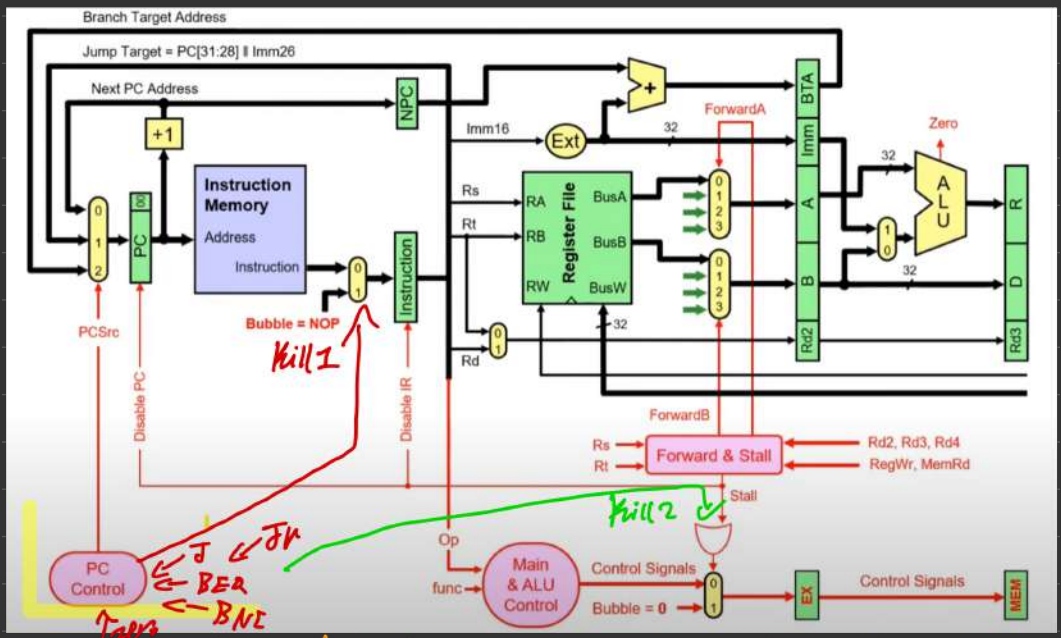
Since b occurs at EX, we have 2 extra cycles # two redundent instruction in pipeline



To avoid this, we simply override all ctrl to zero without disabling registers like in Stall



Bubble Conversion



Once we detect a Jump at ID, Kill2 will bubble the next instruction

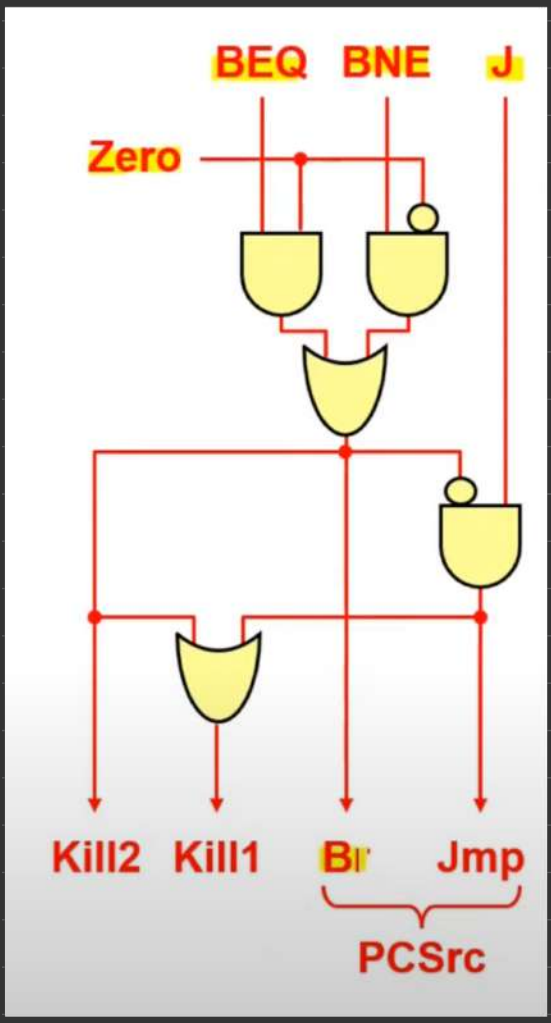
Kill2 Doesn't Freeze registers, so it kills the instruction instead of stall the Pipeline

Process Cheat Sheet

IF (BEQ && zero) || (BNE && !zero) { br=1, jmp=0, Kill1=1, Kill2=1 }

Else if (Jump) { br=0, jmp=1, Kill1=1, Kill2=1 }

Else all 0


BEQ
BNE
J
Zero
Kill2
Kill1
Br
Jmp
PCSrc

CPI Impact

- Base CPI = 1 without counting jump and branch
- Unconditional Jump = 5%, Conditional branch = 20%
- 90% of conditional branches are taken
- Jump kills next instruction, Taken Branch kills next two
- What is the effect of jump and branch on the CPI?

Solution:

- Jump adds 1 wasted cycle for 5% of instructions = 1 × 0.05
- Branch adds 2 wasted cycles for 20% × 90% of instructions
  = 2 × 0.2 × 0.9 = 0.36
- New CPI = 1 + 0.05 + 0.36 = 1.41 (due to wasted cycles)

Imagine you have 100 instructions:

75 instructions. 75 cycles
5 instructions: 10 cycles
20 × 0.9 = 18 instr: 54 cycles
2 instructions: 2

141 cycles / 100 instructions

1.41 CPI

Pipelining CPU

Since JR can only be detected in EX Phase, we gotta kill two instructions like in branch: kill1 kill2

→ no, do it in ID

So PC ctrl needs 3 entry modes.

- is Branch: BEQ BNE → ctrl signals from EX stage
- is Jump: jump jal → ctrl signals from ID
- is JR: JR → ctrl signals from ID

For the new mainX ALU ctrl just combine and add 4 new Flags:

1. BEQ
2. BNE
3. is Jump { Jump, Jal }
4. is JR — JR # this is special bcs we need both the Funct & opcode

Ctrl signals per stage

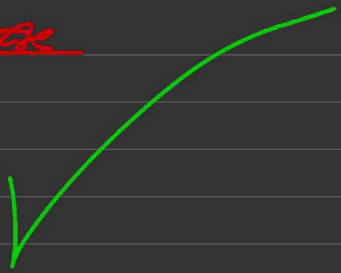
- PC MUX

- ID:
1. RegDst
2. ExtOp
3. RtR7R2
4. is Jump
5. is Jr

- Ex:
1. ALU Src
2. ALU op
3. is BEQ
4. is BNE

- MEM
1. MemWr
2. WB

- WB
1. RegWrite



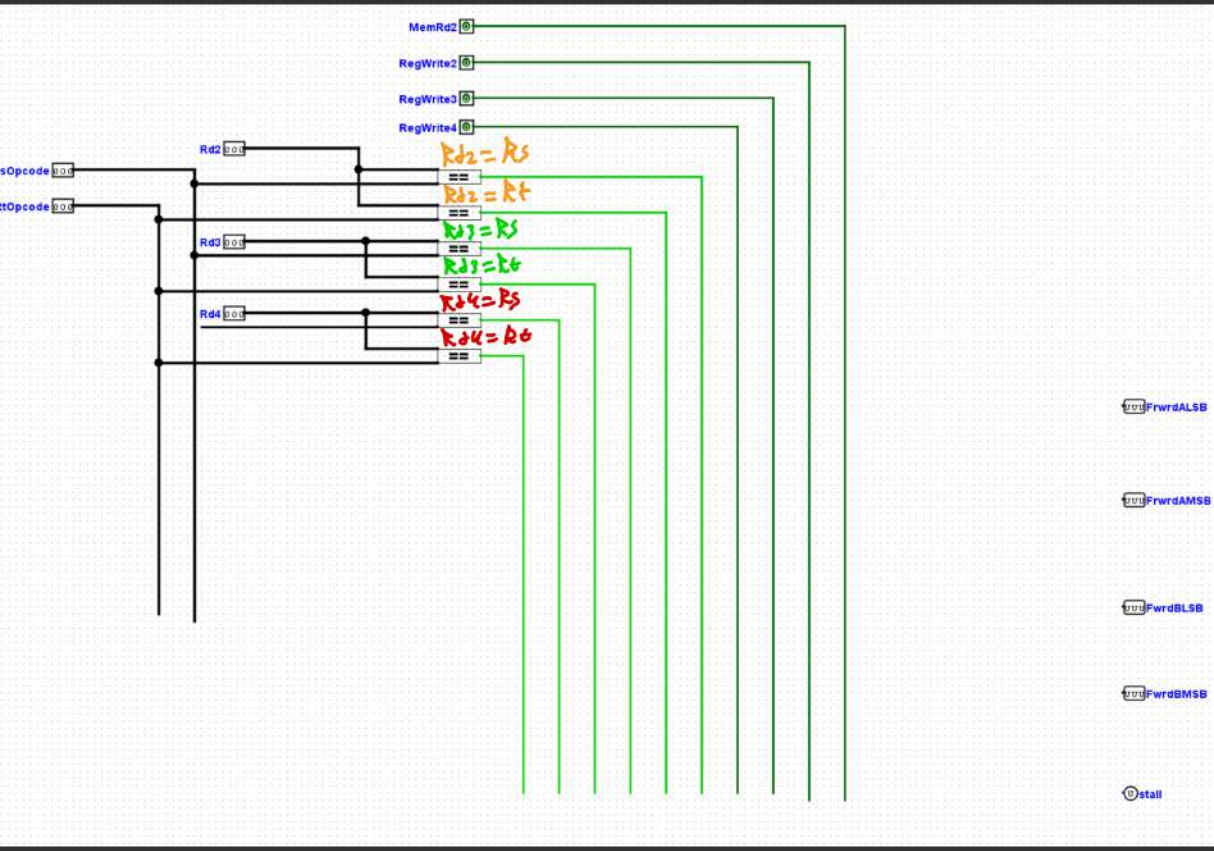
Detection:

IF (Rs == Rd2) { Forward = 13

elif (Rs == Rd3) { Forward = 23

elif (Rs == Rd4) { Forward = 37

else { Forward = 07



RsRd2	Wr2	RsRd3	Wr3	RsRd4	Wr4	A
1	1	X	X	X	X	0 1
0	0	1	1	X	X	1 0
0	0	0	0	1	1	1 1

lw $r3, 0($r2)
add $r2, $r2, $r1

ここ