

The hazard unit handles 3 types of hazards:

RAW (Read After Write):

Detection:

- Compare source registers at EX stage with destination reg at MA and WB stages. If they are the same we have a hit
- Check if previous instructions will write to registers. In case they decide to overwrite them.
- Check if dest register is not x0

Handling:

E_forward_alu_op1 - Selects ALU operand 1 source logic:

- 00: Register file value (E_rf_rd1)
- 01: WB stage result - when E_rs1 == W_rd
- 10: MA stage result - when E_rs1 == M_rd AND not LW

- E_forward_alu_op2 - Selects ALU operand 2 source logic:

- 00: Register file value (E_rf_rd2)
- 01: WB stage result - when E_rs2 == W_rd
- 10: MA stage result - when E_rs2 == M_rd AND not LW

LW Hazard:

Detection:

- Check if instruction at EX stage is LW
- Check if ID stage needs the result
- Ensure destination reg is not x0

Handling:

- Stall: Sets PC_en = 0 and IF_ID_en = 0 to stall PC and IF/ID register
- Flush: Sets ID_EX_clr = 1 which flushes ID/EX register by inserting NOP
- The whole pipeline stalls for one cycle until LW completes at WB stage

BEQ Hazard: // Not Implemented !!!!!

- Flush: If branch taken flush 2 pipeline registers IF-ID and ID-EX. This removed incorrectly loaded instructions

In the program example for lw hazard (figure 3b), you may observe that no matter whether flush or not flush the PLR2, x2/x3/x4 always end up with the correct value. Please explain this concept and write a small program to demonstrate when flushing can make a difference.

The reason why is addi x2, x1, 1 it is in ID stage when LW is in EX stage so the stall prevents it from advancing. When LW completes, x1 is available from WB stage and x2 gets the correct value.

addi x3, x1, 2 is stalled in IF stage so by the time it reaches EX, x1 is already available from register file so it gets the correct value regardless of whether we flush or not. addi x4, x1, 3 is the same, it gets the correct value after stall.

Example:

lw x1, 0(x2)

addi x3, x0, 10 // doesn't use x1 but is in pipeline

addi x4, x1, 1 // depends on the result of lw

Without a flush x4 might execute with the wrong control signals.

Write your understanding of data forward, flush and stall, considering the circuit level behavior and the effect.

Make sure the sentences are clear and concise

Data forward: We bypass the register file by directly routing the data from later pipeline stages to earlier ones. We use MUX to do that. This reduces CPI by avoiding stalls. It's mainly used inside RAW hazards.

Flush: Clears the pipeline registers by resetting them to NOP. Prevents incorrect execution but also causes performance penalties since we are wasting cycles. It is necessary for data hazards mainly branches and LW

Stall: prevents new instructions from entering the pipeline and keeps the current pipeline state unchanged. Disables enable signals on pc and pipeline registers. Causes performance penalties but it's necessary when the data required for an operation is not yet ready also in LW hazards.

1	Hazard Type	Name	Conditions	Encoding
2	RAW	E_forward_alu_op1	(E_rs1 == M_rd) AND M_we_rf AND (M_rd != 0) AND (M_sel_result != 01)	10
3	RAW	E_forward_alu_op1	(E_rs1 == W_rd) AND W_we_rf AND (W_rd != 0)	01
4	RAW	E_forward_alu_op1	Otherwise	00
5	RAW	E_forward_alu_op2	(E_rs2 == M_rd) AND M_we_rf AND (M_rd != 0) AND (M_sel_result != 01)	10
6	RAW	E_forward_alu_op2	(E_rs2 == W_rd) AND W_we_rf AND (W_rd != 0)	01
7	RAW	E_forward_alu_op2	Otherwise	00
8	LW	PC_en	(E_sel_result == 01) AND E_we_rf AND (E_rd != 0) AND ((D_rs1 == E_rd) OR (D_rs2 == E_rd))	0
9	LW	IF_ID_en	(E_sel_result == 01) AND E_we_rf AND (E_rd != 0) AND ((D_rs1 == E_rd) OR (D_rs2 == E_rd))	0
10	LW	ID_EX_clr	(E_sel_result == 01) AND E_we_rf AND (E_rd != 0) AND ((D_rs1 == E_rd) OR (D_rs2 == E_rd))	1
11	LW	PC_en	Otherwise	1
12	LW	IF_ID_en	Otherwise	1
13	LW	ID_EX_clr	Otherwise	0
14	Control	IF_ID_clr	E_branch AND E_zero	1
15	Control	ID_EX_clr	E_branch AND E_zero	1
16	Control	IF_ID_clr	Otherwise	0
17	Control	ID_EX_clr	Otherwise (unless LW hazard)	0
18	Control	IF_ID_clr	D_jump	1
19	Control	ID_EX_clr	D_jump	0

3.2.2

Single-Cycle processor:

- 1. Register clk-Q: 40ps
- 2. Instruction Memory Read: 200ps
- 3. Register File Read: 100ps
- 4. ALU: 120ps
- 5. Data Memory Read: 200ps
- 6. Register File Write: 100ps

Total Critical Path: 40 + 200 + 100 + 120 + 200 + 100 = 760 ps

Max freq: 1/760ps =~ 1.316 Ghz

Multi-Cycle:

- Fetch: Register clk-Q + Mem Read = 40ps + 200ps = 240ps
- Decode: Register Read = 100ps
- Execute: Register Read + ALU = 100ps + 120ps = 220ps
- Memory: Mem Read = 200ps
- Writeback: Register Write = 100ps

Longest Stage: Fetch = 240 ps

Max Frequency: 1 / 240ps = 4.167 GHz

Pipelined Processor:

- IF Stage:
 - Register clk-Q + Instruction Memory Read = 240ps
- ID Stage:
 - Register File Read + Decoder/Ctrl Logic + Sign Extender = 155ps
- EX Stage:
 - Forwarding MUX + ALU= 150ps
- MA Stage:
 - Forwarding MUX + Data Memory Read = 230ps
- WB Stage:
 - Result MUX = 30ps

Critical Path = longest stage = IF stage = 240 ps

Max Frequency: 1 / 240ps = 4.167 GHz

Pipelined is only around 3x faster not 5x. This is because we have a big bottleneck on memory access and the speed is determined by the longest stage. In reality we also are going to have a lot of overhead because of the additional mux and units like hazard detection. Additionally in reality there will also be hazard penalties which would require us to stall or even flush the pipeline.