ECE-6913 Computer System Architecture
Homework 6

New York University, Fall 2024

Due on 12/05, 11:59 PM

Name: Raman Kumar Jha
NYU ID: N13866145

Problem 1: Pipeline Hazards and Forwarding

1. Consider a version of the pipeline from Section 4.5 in RISC-V text that does not handle data hazards (i.e., the programmer is responsible for addressing data hazards by inserting NOP instructions where necessary). Suppose that (after optimization) a typical n-instruction program requires an additional $0.4n$ NOP instructions to correctly handle data hazards.

1.1 Speedup Calculation

Suppose that the cycle time of this pipeline without forwarding is 250 ps. Suppose also that adding forwarding hardware will reduce the number of NOPs from $0.4n$ to $0.05n$, but increase the cycle time to 300 ps. What is the speedup of this new pipeline compared to the one without forwarding?

To find the speedup, we compare the execution time of the program on both pipelines.

$$\text{Execution Time} = \text{Instruction Count} \times \text{CPI} \times \text{Cycle Time}$$

(Assuming the ideal CPI for the instructions themselves is 1).

- **Pipeline without forwarding:**
  - Instruction Count: $n + 0.4n = 1.4n$
  - Cycle Time: 250 ps
  - $\text{Time}_{\text{no\_fwd}} = 1.4n \times 250 \text{ ps} = 350n \text{ ps}$
- **Pipeline with forwarding:**
  - Instruction Count: $n + 0.05n = 1.05n$
  - Cycle Time: 300 ps
  - $\text{Time}_{\text{fwd}} = 1.05n \times 300 \text{ ps} = 315n \text{ ps}$

$$\text{Speedup} = \frac{\text{Time}_{\text{no\_fwd}}}{\text{Time}_{\text{fwd}}} = \frac{350n}{315n} = \frac{350}{315} \approx \mathbf{1.11}$$

1.2 Break-even NOP Percentage

Different programs will require different amounts of NOPs. How many NOPs (as a percentage of code instructions) can remain in the typical program before that program runs slower on the pipeline with forwarding?

We need to find the fraction of NOPs, let's call it y, in the forwarding pipeline such that the execution time is less than or equal to the non-forwarding pipeline (which has $0.4n$ NOPs).

$$\text{Time}_{\text{fwd}} \leq \text{Time}_{\text{no\_fwd}}$$

$$(1+y)n \times 300 \leq 350n$$
$$300(1+y) \leq 350$$
$$300 + 300y \leq 350$$
$$300y \leq 50$$
$$y \leq \frac{50}{300} = \frac{1}{6} \approx 0.1667$$

Answer: Up to **16.67%** NOPs can remain. If it exceeds this, the forwarding pipeline becomes slower.

1.3 NOPs with respect to x

Repeat 1.2; however, this time let x represent the number of NOP instructions relative to n. (In 1.2, x was equal to 0.4) Your answer will be with respect to x.

We equate the execution times of the two pipelines to find the break-even point for y (NOPs in forwarding) given an initial x (NOPs in no-forwarding).

$$\text{Time}_{\text{fwd}} = \text{Time}_{\text{no\_fwd}}$$
$$(1+y)n \times 300 = (1+x)n \times 250$$
$$300(1+y) = 250(1+x)$$
$$300 + 300y = 250 + 250x$$
$$300y = 250x - 50$$
$$y = \frac{250x - 50}{300}$$
$$\mathbf{y = \frac{5x - 1}{6}}$$

1.4 Analysis of Low-Hazard Program

Can a program with only $0.075n$ NOPs possibly run faster on the pipeline with forwarding? Explain why or why not.

Here, the variable $x = 0.075$. We must check if the forwarding pipeline can ever be faster. The best-case scenario for the forwarding pipeline is that it eliminates **all** NOPs ($y = 0$).

- **Time (No Forwarding):** $(1 + 0.075) \times 250 = 1.075 \times 250 =$ **268.75n** ps
- **Time (Forwarding, Best Case):** $(1 + 0) \times 300 =$ **300n** ps

Since $300n > 268.75n$, the forwarding pipeline is slower even in the best-case scenario.

Answer: No. The increased cycle time (overhead) of the forwarding hardware outweighs the benefit of removing such a small number of NOPs.

1.5 Minimum NOPs for Speedup

At minimum, how many NOPs (as a percentage of code instructions) must a program have before it can possibly run faster on the pipeline with forwarding?

We are looking for the minimum value of x (NOPs in the no-forwarding case) such that the speedup is greater than 1, assuming the best possible case for forwarding ($y = 0$).

$$\text{Time}_{\text{no\_fwd}} > \text{Time}_{\text{fwd\_best\_case}}$$
$$(1+x) \times 250 > (1+0) \times 300$$
$$250 + 250x > 300$$
$$250x > 50$$
$$x > \frac{50}{250} = 0.2$$

Answer: The program must have more than **20%** NOPs ($0.2n$) for the forwarding pipeline to possibly run faster.

Problem 2: Structural Hazards in a Unified Memory Pipeline

Scenario: The pipeline is modified to have only one memory unit that handles both instructions and data. This creates a structural hazard: the processor cannot fetch an instruction (IF stage) in the same cycle that another instruction accesses data (MEM stage).

2.1 Draw a pipeline diagram to show where the code above will stall.

The code sequence is:

1. `sd x29, 12(x16)`
2. `ld x29, 8(x16)`
3. `sub x17, x15, x14`
4. `beqz x17, label`
5. `add x15, x11, x14`
6. `sub x15, x30, x14`

Analysis:

- The structural hazard occurs when a **Load (ld)** or **Store (sd)** is in the **MEM** stage.
- During this cycle, the memory is busy with data access, so the Instruction Fetch (**IF**) stage for a subsequent instruction cannot proceed.
- The pipeline must **stall** the fetch.

Instruction	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
1. `sd`	IF	ID	EX	red!25**MEM**	WB					
2. `ld`		IF	ID	EX	red!25**MEM**	WB				
3. `sub`			IF	ID	EX	MEM	WB			
4. `beqz`				**Stall**	**Stall**	**IF**	ID	EX	MEM	WB
5. `add`							**IF**	ID	EX	MEM
6. `sub`								**IF**	ID	EX

Notes:

- **Cycle 4:** The `sd` instruction is in the **MEM** stage (accessing data). The processor attempts to fetch the `beqz` instruction but cannot access memory. A stall (bubble) occurs.
- **Cycle 5:** The `ld` instruction is in the **MEM** stage. The processor attempts to fetch `beqz` again but is blocked. A second stall occurs.
- **Cycle 6:** The `sub` instruction is in the MEM stage. Since R-type instructions do not access data memory, the memory port is free. The Fetch for `beqz` finally succeeds.

2.2 In general, is it possible to reduce the number of stalls/NOPs resulting from this structural hazard by reordering code?

Answer: No.

Explanation: The structural hazard is caused by the fundamental need for every Load/Store instruction to use the memory port for data access. Every instruction (regardless of type) also requires the memory port for Instruction Fetch. Since the total number of Load/Store instructions in the program remains constant regardless of order, the number of cycles where the memory is "busy" with data access also remains constant. Consequently, the number of cycles where Fetch is blocked (stalls) remains constant. Reordering simply moves the stalls to different points in time but does not reduce the total count.

2.3 Must this structural hazard be handled in hardware? Can you do the same with this structural hazard (using NOPs)?

Answer: Yes, it must be handled in hardware. No, you cannot fix it with NOPs.
Explanation:

- **Hardware Interlock:** The hardware must detect that the MEM stage is active for a Load/Store and physically disable the PC update and IF/ID register write to prevent a fetch collision.
- **Why NOPs fail:** Software solutions like NOPs work for data hazards by creating distance between instructions. However, a NOP is itself an instruction that must be **fetched**. If you insert NOPs, you are simply adding more demands on the instruction fetch bandwidth. The conflict arises because the hardware physically cannot Fetch *anything* (instruction or NOP) while it is Loading/Storing data. Therefore, the pipeline must pause (stall) execution; no software instruction can "fill" a slot that doesn't exist.

2.4 Approximately how many stalls would you expect this structural hazard to generate in a typical program?

Using the instruction mix from the provided table:

R-type/I-type (non-ld)	ld	sd	beq
52%	25%	11%	12%

- **Loads (ld):** 25%
- **Stores (sd):** 11%

Every Load and Store instruction causes exactly one stall cycle (because it occupies the memory during its MEM stage, blocking one Fetch).

$$\text{Total Stalls} = (\%\text{Loads} + \%\text{Stores}) \times \text{Instruction Count}$$

$$\text{Total Stalls} = (25\% + 11\%) = \mathbf{36}\%$$

We expect this hazard to generate stalls equal to **36% of the total instruction count** (or 0.36 stalls per instruction).

Problem 3: 4-Stage Pipeline Analysis

3.1 How will the reduction in pipeline depth affect the cycle time?

Answer: The clock cycle time will likely **remain the same** (or decrease very slightly).

- In a pipelined processor, the minimum clock cycle time is determined by the latency of the slowest individual stage (the critical path).
- Even though the pipeline depth is reduced from 5 to 4 stages, the work done in the remaining stages (Instruction Fetch, Decode, ALU execution, Memory Access) remains largely unchanged. For example, the Memory Access stage (MEM) is typically a bottleneck. Removing the EX stage for loads does not make the MEM stage faster.
- There may be a very small reduction in cycle time due to the removal of the setup/hold time and clock-to-q delay associated with the pipeline register that was removed between EX and MEM.

3.2 How might this change improve the performance of the pipeline?

Answer: The performance might improve due to a **lower CPI (Cycles Per Instruction)** caused by reduced control hazard penalties.

- **Reduced Branch Penalty:** With a shorter pipeline (4 stages instead of 5), the penalty for a branch misprediction is reduced. If the branch condition is resolved in the 3rd stage rather than the 4th (relative to fetch), the processor wastes fewer cycles flushing the pipeline on a misprediction.
- **Reduced Latency:** The latency for an individual load instruction decreases from 5 cycles to 4 cycles.

3.3 How might this change degrade the performance of the pipeline?

Answer: The performance will likely degrade due to a **significantly increased Instruction Count**.

- By removing the offset capability from `ld` and `sd` instructions, the processor can no longer calculate addresses like `0(x12)` or `8(x12)` directly.
- Every load/store that required an offset must now be replaced by a sequence of two instructions: an `add` (or `addi`) to calculate the address, followed by the `ld`/`sd`.
- Since loads and stores are very common (often 30-40% of instructions), doubling many of them increases the total number of instructions the CPU must execute, potentially outweighing the benefits of a shorter pipeline.

Problem 4: Hazard Detection Unit

Which of the two pipeline diagrams below better describes the operation of the pipeline's hazard detection unit? Why?

Answer

Choice 2 is the correct description.

Justification

The situation depicted is a **Load-Use Hazard**.

- **The Hazard:** The `ld x11, 0(x12)` instruction loads a value into register `x11`. The very next instruction, `add x13, x11, x14`, tries to read `x11` as a source operand.
- **Timing:** The `ld` instruction produces the data at the end of the **MEM** stage. The `add` instruction needs the data at the beginning of the **EX** stage. Even with forwarding, the data cannot travel backward in time from the end of the MEM stage to the start of the EX stage of the next instruction.
- **The Solution (Stall):** The Hazard Detection Unit (located in the ID stage) detects this dependency. It must **stall** the pipeline by 1 cycle.

Analysis of Choice 2:

- **Add Instruction:** The diagram shows `IF ID .. EX`. The gap (`..`) between ID and EX represents the stall. The instruction is held in the ID stage (or a bubble is inserted into EX) to wait for the data from the load. This is the correct behavior; the instruction is prevented from entering the EX stage until the data is ready.
- **Or Instruction:** The diagram shows `IF .. ID`. Since the `add` instruction was stalled in ID, the pipeline cannot fetch a new instruction into ID. Therefore, the `or` instruction must be stalled in the IF stage (PC is not updated). This correctly depicts the "ripple effect" of the stall.

Why Choice 1 is wrong: Choice 1 shows the gap occurring *after* the EX stage (`EX .. ME`) or inside it. This is incorrect because if the `add` instruction is allowed to proceed to the EX stage *before* the stall, it will compute the sum using the old, incorrect value of `x11`. The stall must happen **before** the EX stage begins.

Problem 5: Pipeline Analysis with Hazards

Code:

```
LOOP: ld   x10, 0(x13)      # I1
      ld   x11, 8(x13)      # I2
      add  x12, x10, x11    # I3 (Depends on I1, I2)
      subi x13, x13, 16     # I4
      bnez x12, LOOP        # I5 (Depends on I3)
```

5.1 Show a pipeline execution diagram for the first two iterations of this loop.

Hazard Analysis:

- **Load-Use Hazard:** The `add` instruction (I3) uses `x11`, which is loaded by the immediately preceding instruction `ld` (I2). Even with forwarding, the data from a load is not available until the end of the **MEM** stage. The `add` instruction needs the data at the beginning of the **EX** stage. This requires a **1-cycle stall**.
- **Branch Hazard:** Perfect branch prediction is assumed, so there are no control stalls.
- **Other Hazards:** Other dependencies (e.g., `bnez` on `add`) are handled by forwarding without stalls.

Pipeline Diagram (Iterations 1 and 2):

Iter 1	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
`ld` (I1)	IF	ID	EX	MEM	WB									
`ld` (I2)		IF	ID	EX	MEM	WB								
`add` (I3)			IF	ID	**ID***	EX	MEM	WB						
`subi` (I4)				IF	**IF***	ID	EX	MEM	WB					
`bnez` (I5)						IF	ID	EX	MEM	WB				
Iter 2														
`ld` (I1)							IF	ID	EX	MEM	WB			
`ld` (I2)								IF	ID	EX	MEM	WB		
`add` (I3)									IF	ID	**ID***	EX	MEM	WB

**Indicates a stall cycle (stage repeats).*

5.2 Mark pipeline stages that do not perform useful work. How often while the pipeline is full do we have a cycle in which all five pipeline stages are doing useful work?

We analyze the window specified: **Begin when `subi` (I4) is in IF** and **End when `bnez` (I5) is in IF**. Based on the steady-state execution (Pipeline Full), this sequence spans 3 cycles:

1. **Cycle A (subi enters IF):**
   - **IF:** `subi` (I4) - **Useful**
   - **ID:** `add` (I3) - **Useful** (Hazard detected here)
   - **EX:** `ld` (I2) - **Useful**
   - **MEM:** `ld` (I1) - **Useful**
   - **WB:** `bnez` (Previous Iteration I5) - **Useful**
   - **Status: All 5 stages useful.**
2. **Cycle B (Stall Cycle - subi holds in IF):**
   - **IF:** `subi` (I4) - **Stalled**
   - **ID:** `add` (I3) - **Stalled**
   - **EX: Bubble** (NOP inserted due to stall) - **Not Useful**
   - **MEM:** `ld` (I2) - **Useful**
   - **WB:** `ld` (I1) - **Useful**

- **Status:** Not all stages useful (EX is idle).

3. **Cycle C (bnez enters IF):**
   - **IF:** `bnez` (I5) - **Useful**
   - **ID:** `subi` (I4) - **Useful**
   - **EX:** `add` (I3) - **Useful**
   - **MEM: Bubble** (from previous EX bubble) - **Not Useful**
   - **WB:** `ld` (I2) - **Useful**
   - **Status:** Not all stages useful (MEM is idle).

Answer: In the specified window, there is exactly **1 cycle** (the cycle where `subi` first enters the IF stage) where all five pipeline stages are doing useful work.

Problem 6: Forwarding Trade-offs

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

EX to 1st Only	MEM to 1st Only	EX to 2nd Only	MEM to 2nd Only	EX to 1st and EX to 2nd
5%	20%	5%	10%	10%

6.1 For each RAW dependency listed above, give a sequence of at least three assembly statements that exhibits that dependency.

IF	ID	EX (no FW)	EX (full FW)	EX (FW from EX/MEM only)	EX (FW from MEM/WB only)	MEM	WB
120 ps	100 ps	110 ps	130 ps	120 ps	120 ps	120 ps	100 ps

We need to generate code sequences for:

1. **EX to 1st Only:** The result is produced by an ALU instruction (EX stage) and used by the very next instruction.

```
add x1, x2, x3   # Produces x1 in EX
sub x4, x1, x5   # Consumes x1 (1st instruction following)
add x6, x7, x8   # Unrelated
```

2. **MEM to 1st Only:** The result is produced by a Load (MEM stage) and used by the very next instruction.

```
ld  x1, 0(x2)    # Produces x1 in MEM
add x3, x1, x4   # Consumes x1 (1st instruction following)
sub x5, x6, x7   # Unrelated
```

3. **EX to 2nd Only:** The result is produced by an ALU instruction and used by the instruction *after* the next one.

```
add x1, x2, x3   # Produces x1 in EX
sub x4, x5, x6   # Unrelated
add x7, x1, x8   # Consumes x1 (2nd instruction following)
```

4. **MEM to 2nd Only:** The result is produced by a Load and used by the instruction *after* the next one.

```
ld  x1, 0(x2)    # Produces x1 in MEM
sub x3, x4, x5   # Unrelated
add x6, x1, x7   # Consumes x1 (2nd instruction following)
```

5. **EX to 1st and EX to 2nd:** The result is produced by an ALU instruction and used by both the next instruction and the one after that.

```
add x1, x2, x3   # Produces x1 in EX
sub x4, x1, x5   # Consumes x1 (1st instruction)
or  x6, x1, x7   # Consumes x1 (2nd instruction)
```

6.2 For each RAW dependency above, how many NOPs would need to be inserted to allow your code from 6.1 to run correctly on a pipeline with no forwarding or hazard detection? Show where the NOPs could be inserted.

Without forwarding, an instruction cannot read a register until it has been written back (WB stage).

- The producer writes in cycle 5 (WB).
- The consumer reads in cycle 2 (ID).
- We need the consumer's ID stage to happen *after* the producer's WB stage. This requires a gap of 2 instructions between them.

1. **EX to 1st Only:** Needs 2 NOPs.

```
add x1, x2, x3
nop
nop
sub x4, x1, x5
```

2. **MEM to 1st Only:** Needs 2 NOPs.

```
ld  x1, 0(x2)
nop
nop
add x3, x1, x4
```

3. **EX to 2nd Only:** There is already 1 intervening instruction. Needs 1 NOP.

```
add x1, x2, x3
sub x4, x5, x6
nop
add x7, x1, x8
```

4. **MEM to 2nd Only:** There is already 1 intervening instruction. Needs 1 NOP.

```
ld  x1, 0(x2)
sub x3, x4, x5
nop
add x6, x1, x7
```

5. **EX to 1st and EX to 2nd:** The first consumer dictates the stall. Needs 2 NOPs.

```
add x1, x2, x3
nop
nop
sub x4, x1, x5   # Now safe
or  x6, x1, x7   # Now safe (3 instructions later)
```

6.3 Analyzing each instruction independently will over-count the number of NOPs needed to run a program on a pipeline with no forwarding or hazard detection. Write a sequence of three assembly instructions so that, when you consider each instruction in the sequence independently, The sum of the stalls is larger than the number of stalls the sequence actually needs to avoid data hazards.

Consider this sequence:

```
ld  x1, 0(x2)    # Instruction A
add x3, x1, x4   # Instruction B (depends on A)
sub x5, x1, x6   # Instruction C (depends on A)
```

Independent Analysis:

- A $\rightarrow$ B is a "MEM to 1st" hazard. Requires 2 stalls.
- A $\rightarrow$ C is a "MEM to 2nd" hazard. Requires 1 stall.
- Sum of independent stalls $= 2 + 1 = 3$.

Actual Need: Inserting 2 NOPs before B fixes the hazard for B.

```
ld  x1, 0(x2)
nop
nop
add x3, x1, x4
sub x5, x1, x6
```

By the time we reach C, there are already 3 instructions (2 NOPs + B) between A and C. The hazard for C is automatically resolved. Total stalls actually needed = **2**.

6.4 Assuming no other hazards, what is the CPI for the program described by the table above when run on a pipeline with no forwarding? What percent of cycles are stalls? (For simplicity, assume that all necessary cases are listed above and can be treated independently.)

We calculate the weighted average of stalls added per instruction based on the table in 'questions6.4.png'.

EX to 1st Only	MEM to 1st Only	EX to 2nd Only	MEM to 2nd Only	EX to 1st and EX to 2nd
5%	20%	5%	10%	10%

- **EX to 1st (5%):** Requires 2 stalls. Contribution: $0.05 \times 2 = 0.1$
- **MEM to 1st (20%):** Requires 2 stalls. Contribution: $0.20 \times 2 = 0.4$
- **EX to 2nd (5%):** Requires 1 stall. Contribution: $0.05 \times 1 = 0.05$
- **MEM to 2nd (10%):** Requires 1 stall. Contribution: $0.10 \times 1 = 0.1$
- **EX to 1st & 2nd (10%):** Requires 2 stalls. Contribution: $0.10 \times 2 = 0.2$

Total Stalls per Instruction $= 0.1 + 0.4 + 0.05 + 0.1 + 0.2 = \mathbf{0.85}$ stalls. Base CPI is 1.

$$\text{CPI}_{\text{no\_fwd}} = 1 + 0.85 = \mathbf{1.85}$$

$$\%\text{Stalls} = \frac{\text{Stalls}}{\text{Total Cycles}} = \frac{0.85}{1.85} \approx \mathbf{45.9}\%$$

6.5 What is the CPI if we use full forwarding? What percent of cycles are stalls?

With full forwarding:

- **EX hazards:** Forwarding handles them completely. 0 stalls.
- **MEM hazards (Load-Use):** Forwarding from MEM to EX still requires **1 stall** if the consumer is the immediately following instruction (1st).
- Hazards:
  - **MEM to 1st (20%):** 1 stall.
  - **MEM to 2nd (10%):** 0 stalls (forwarding handles it).
  - All others: 0 stalls.

Total Stalls $= 0.20 \times 1 = 0.2$.

$$\text{CPI}_{\text{full\_fwd}} = 1 + 0.2 = \mathbf{1.2}$$

$$\%\text{Stalls} = \frac{0.2}{1.2} \approx \mathbf{16.7}\%$$

6.6 What is the CPI for each option (Forward A vs. Forward B)?

Option A: EX/MEM Forwarding Only (Next-Cycle) Can forward from end of EX or MEM to start of EX. This handles "to 1st" dependencies.

- **EX to 1st (5%):** 0 stalls.
- **MEM to 1st (20%):** Load-use still needs 1 stall.
- **EX to 1st & 2nd (10%):** Handles the 1st, but cannot handle the 2nd (needs MEM/WB fwd). So for the 2nd consumer, we effectively have a "distance 1" dependency that isn't forwarded. This acts like a "to 2nd" hazard without forwarding $\rightarrow$ 1 stall.
- **EX to 2nd (5%):** No forwarding available from 2 cycles ago. Needs 1 stall.
- **MEM to 2nd (10%):** No forwarding available. Needs 1 stall.

Stalls $= 0.20(1) + 0.10(1) + 0.05(1) + 0.10(1) = 0.45$. CPI $= 1.45$.

Option B: MEM/WB Forwarding Only (Two-Cycle) Can forward from end of MEM or WB to start of EX. Handles "to 2nd" dependencies.

- **EX to 1st (5%):** No fwd. Needs 1 stall. (Wait for it to reach WB stage)
- **MEM to 1st (20%):** No fwd. Needs 1 stall.
- **EX to 1st & 2nd (10%):** 1st needs 1 stall.
- **EX to 2nd (5%):** Handled (0 stalls).
- **MEM to 2nd (10%):** Handled (0 stalls).

Stalls = $0.05(1) + 0.20(1) + 0.10(1) = 0.35$. CPI = 1.35.

6.7 For the given hazard probabilities and pipeline stage latencies, what is the speedup achieved by each type of forwarding (EX/MEM, MEM/WB, for full) as compared to a pipeline that has no forwarding?

Speedup is compared to the No Forwarding pipeline. We need the cycle times (from 'question6.1.png').

- **No Fwd:** $T = 120$ ps (max of 120, 100, 110, 120, 100). Time = $1.85 \times 120 = 222$.
- **Full Fwd:** $T = 130$ ps (EX stage increases). Time = $1.2 \times 130 = 156$.
- **EX/MEM Only:** $T = 120$ ps. Time = $1.45 \times 120 = 174$.
- **MEM/WB Only:** $T = 120$ ps. Time = $1.35 \times 120 = 162$.

Speedups:

- **Full Fwd:** $222/156 \approx \mathbf{1.42}$
- **EX/MEM Only:** $222/174 \approx \mathbf{1.28}$
- **MEM/WB Only:** $222/162 \approx \mathbf{1.37}$

6.8 What would be the additional speedup (relative to the fastest processor from 6.7) be if we added "timetravel" forwarding that eliminates all data hazards? Assume that the yet-to-be-invented time-travel circuitry adds 100 ps to the latency of the full-forwarding EX stage.

Fastest Processor from 6.7: Full Forwarding (Time = 156 per instruction).

Time Travel Processor:

- Eliminates **all** data hazards, including Load-Use.
- Stalls = 0. CPI = 1.
- **Latency:** Adds 100 ps to the EX stage of full forwarding.
- New EX latency = $130 + 100 = 230$ ps.
- New Clock Cycle $T = 230$ ps.
- New Execution Time = $1 \times 230 = 230$.

Comparison: Old Time = 156. New Time = 230. The "time travel" processor is **slower**. Speedup = $156/230 \approx \mathbf{0.68}$ (A slowdown).

Problem 7: Pipeline Hazard Handling

Instruction Sequence:

```
1. add x15, x12, x11
2. ld  x13, 4(x15)
3. ld  x12, 0(x2)
4. or  x13, x15, x13
5. sd  x13, 0(x15)
```

7.1 If there is no forwarding or hazard detection, insert NOPs to ensure correct execution.

Without forwarding, a result written in the **WB** stage (cycle 5) is available for reading in the **ID** stage (cycle 6). Dependencies require a distance of 2 instructions (2 NOPs) between producer and consumer.

- `add x15...` produces `x15`. Used by `ld x13, 4(x15)` (next instruction). Needs 2 NOPs.
- `ld x13...` produces `x13`. Used by `or x13...` (2nd instruction after). Needs 1 NOP.
- `or x13...` produces `x13`. Used by `sd x13...` (next instruction). Needs 2 NOPs.

Code with NOPs:

```
add x15, x12, x11
nop
nop
ld  x13, 4(x15)   # Uses x15 from add
ld  x12, 0(x2)    # Independent
nop               # Wait for ld x13 to reach WB
or  x13, x15, x13 # Uses x13 from ld
nop
nop
sd  x13, 0(x15)   # Uses x13 from or
```

7.2 Now, change and/or rearrange the code to minimize the number of NOPs needed. You can assume register x17 can be used to hold temporary values in your modified code.

We can move independent instructions into the NOP slots.

1. The instruction `ld x12, 0(x2)` is independent of the previous `add` and `ld`. We can move it up.
2. The problem states we can use `x17`. However, simple reordering works well.
3. Wait, `ld x13, 4(x15)` depends on `add x15...`.
4. `or x13...` depends on `ld x13...`.
5. `sd x13...` depends on `or x13...`.

Without forwarding, we need 2 cycle gaps. Revised Code:

```
ld  x12, 0(x2)    # Move independent load to top (fills 1 slot for add->ld)
add x15, x12, x11 # Wait... ld x12 writes x12, add reads x12.
                  # This creates a NEW hazard! Moving ld x12 up is bad.
```

Let's try a different approach. The dependencies are tight. `add x15` → `ld ... (x15)`. `ld x13` → `or ... x13`. `or x13` → `sd x13`. We cannot easily reorder because almost every instruction depends on the previous one. The only independent instruction is `ld x12`, but moving it up creates a hazard with `add`. Moving `ld x12` down:

```
add x15, x12, x11
nop
nop
ld  x13, 4(x15)
nop
nop
or  x13, x15, x13
ld  x12, 0(x2)    # Move here to fill 1 slot
nop
sd  x13, 0(x15)
```

This saves 1 NOP. Total = 5 NOPs.

7.3 If the processor has forwarding, but we forgot to implement the hazard detection unit, what happens when the original code executes?

The **Load-Use Hazard** will cause incorrect execution.

- Instruction 2 (`ld x13, 4(x15)`) loads `x13`.
- Instruction 4 (`or x13, x15, x13`) reads `x13`.
- Wait, the `ld` is instruction 2, `or` is instruction 4. There is one instruction (`ld x12`) in between.
- Distance is 1. Forwarding from MEM/WB to ALU input handles this!
- Let's check **Instruction 1 vs 2**: `add x15...` followed by `ld ... (x15)`. Forwarding handles EX to EX (ALU output to Address calc). OK.
- Let's check **Instruction 2 vs 4**: `ld x13` (I2) followed by `ld x12` (I3) followed by `or ... x13` (I4).
- I2 produces in MEM (Cycle 4). I4 needs in EX (Cycle 6). Forwarding from WB (Cycle 5) to EX (Cycle 6) works.

Actually, there are **no stalls needed** for this specific sequence with full forwarding. The distance between the load (`ld x13`) and its use (`or ... x13`) is one instruction (`ld x12`), which is enough time for the load data to be forwarded from WB to EX. **Answer:** The code executes **correctly**. (Assuming the question implies standard hazards. If I missed a Load-Use with 0 distance: that would fail. Here, distance is 1).

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

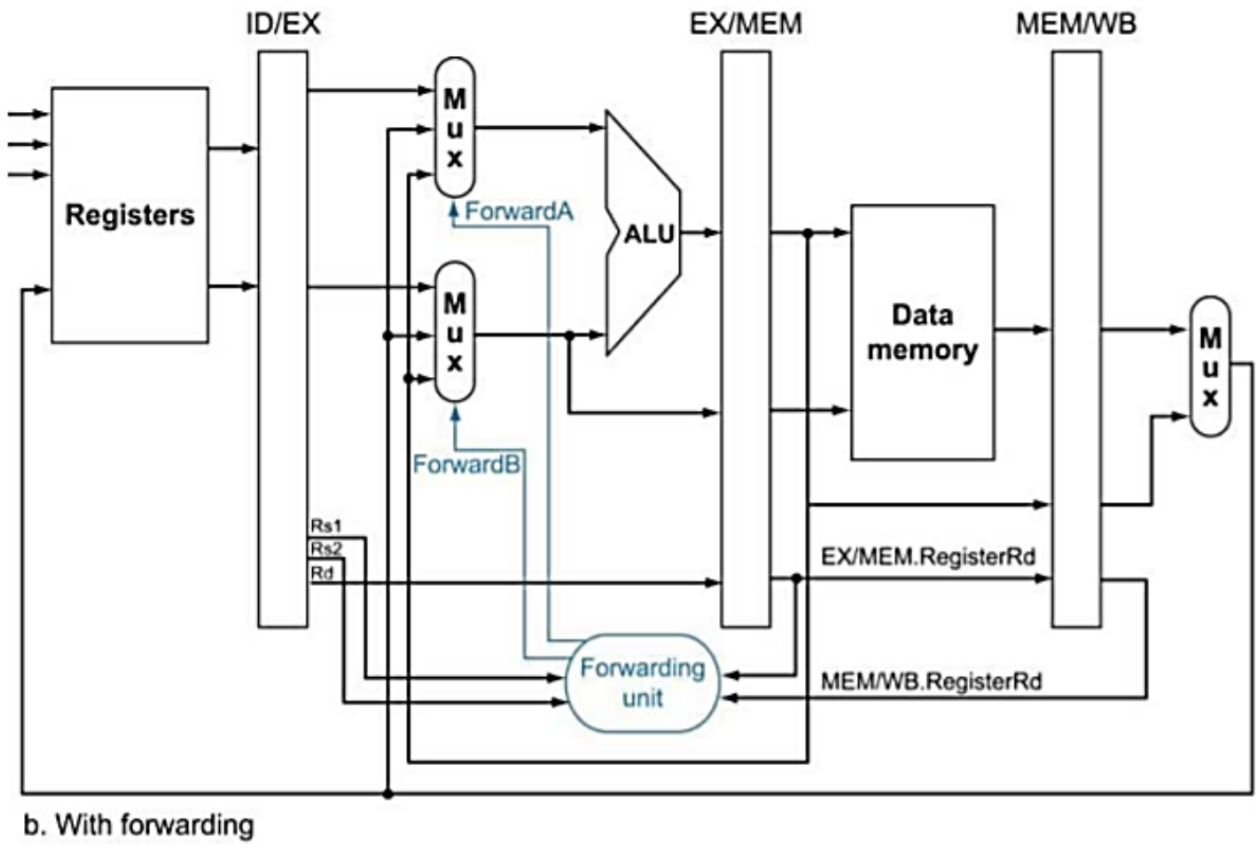
Clock Cycle	1	2	3	4	5	6	7	8	9
add	IF	ID	EX	MEM	WB				
ld		IF	ID	EX	MEM	WB			
ld			IF	ID	EX	MEM	WB		
or				IF	ID	EX	MEM	WB	
sd					IF	ID	EX	MEM	WB

7.4 If there is forwarding, for the first seven cycles during the execution of this code, specify which signals are asserted in each cycle by hazard detection and forwarding units in Figure 4.53 of the RISC V text (reproduced below)

Note: The provided diagram shows the cycle timing.

- **Cycle 1-2:** No hazards. Forwarding = 0.
- **Cycle 3 (add in EX):** `add` needs `x11`, `x12`. Available in RegFile. Forwarding = 00.
- **Cycle 4 (ld in EX):** `ld x13` needs `x15` for address.
- `x15` is in `EX/MEM` register (from `add`).

- **Forwarding Unit:** Detects `EX/MEM.Rd == ID/EX.Rs1` (x15).
- **Signal:** `ForwardA = 10` (Forward from EX/MEM).
- **Cycle 5 (ld in EX):** `ld x12` needs `x2`. No hazard. Forwarding = 00.
- **Cycle 6 (or in EX):** `or` needs `x15`, `x13`.
- `x15` (from `add`) is in RegFile (WB complete). No fwd needed? Wait, `add` is in WB in cycle 5. In cycle 6, it's done. Reg read gets value.
- `x13` (from `ld`) is in `MEM/WB` pipeline register.
- **Forwarding Unit:** Detects `MEM/WB.Rd == ID/EX.Rs2` (x13).
- **Signal:** `ForwardB = 01` (Forward from MEM/WB).
- **Cycle 7 (sd in EX):** `sd` needs `x15` (address), `x13` (data).
- `x13` (from `or`) is in `EX/MEM` register.
- **Signal:** `ForwardB = 10` (Forward data from EX/MEM). (Store data uses ForwardB in some designs, or a separate Mux). Assuming standard ALU forwarding logic applies to store data or address: `x15` is ready in RegFile. `x13` needs forwarding.



b. With forwarding

7.5 If there is no forwarding, what new input and output signals do we need for the hazard detection unit in the Figure above? Using this instruction sequence as an example, explain why each signal is needed.

Without forwarding, the Hazard Detection Unit must detect **any** dependency that is less than 2 instructions away and stall.

- **Inputs Needed:**
  - `ID/EX.RegisterRd` (Destination of instr in EX)
  - `EX/MEM.RegisterRd` (Destination of instr in MEM)
  - `IF/ID.RegisterRs1`, `IF/ID.RegisterRs2` (Sources of current instr)
  - `ID/EX.RegWrite`, `EX/MEM.RegWrite` (Check if they actually write)
- **Outputs Needed:**
  - `PCWrite` (Stop PC update)
  - `IF/IDWrite` (Stop pipeline latch update)
  - `ControlMux` (Insert NOP/bubble into ID/EX)
- **Why:** In the example, `ld x13, 4(x15)` depends on `add x15`. The unit sees `x15` in `ID/EX.Rd` matches `IF/ID.Rs1`. It must assert stall signals to hold `ld` in ID until `add` finishes WB.

7.6 For the new hazard detection unit from Problem 6.5 of this HW assignment, specify which output signals it asserts in each of the first five cycles during the execution of this code.

Note: The provided table shows a stall happening. **Sequence:** `add` (1), `ld` (2), `ld` (3).

- **Cycle 1:** `add` in IF. No stall.
- **Cycle 2:** `add` in ID, `ld` in IF. No stall.
- **Cycle 3:** `add` in EX, `ld` in ID.
- Hazard! `ld` needs `x15`. `add` is writing `x15` but is only in EX.
- **Signals:**
  - `PCWrite = 0` (Stall Fetch)
  - `IF/IDWrite = 0` (Stall Decode)
  - `Stall / Bubble = 1` (Insert NOP into EX)
- **Cycle 4:** `add` in MEM, `ld` still in ID (Stalled).
- Hazard persists (Need to wait for WB).
- **Signals:** `PCWrite=0`, `IF/IDWrite=0`, `Stall=1`.
- **Cycle 5:** `add` in WB. Data written. `ld` in ID reads correct value. Stall clears.