Ch4_review2

Yuanqing Miao

https://github.com/myuanqing/CMPEN331_Spring2025

Data Hazards

Data hazards occur when an instruction depends on the result of a previous instruction that has not yet completed execution.

Read After Write (RAW) – True Dependency

- Occurs when an instruction needs a register value that is yet to be written by an earlier instruction.

Data Hazards

ALU-to-ALU Hazard and **Load-Use Hazard**

1. Forwarding is Effective for ALU Hazards

- **ALU-to-ALU dependencies** (e.g., `ADD` → `SUB`) can be resolved using **forwarding** because the result is available in the EX/MEM or MEM/WB pipeline registers.
- **Forwarding minimizes stalls** by directly passing values from later pipeline stages back to the EX stage.
- **No stalls are needed** for ALU dependencies if forwarding is implemented.

Data Hazards - Load-Use Hazards

2. Load-Use Hazards Require Stalls

- **Forwarding cannot solve load-use hazards** because the loaded data is only available in the **WB stage**.
- If an instruction immediately uses a loaded value, a **1-cycle stall (pipeline bubble) is necessary** to wait for the data.

Control Hazards

- instructions are fetched in stage 1 (IF)
- branch and jump decisions occur in stage 3 (EX)

  i.e. next PC is not known until ***2 cycles*** *after* branch/jump
- Can optimize and move branch and jump decision to stage 2 (ID)

  i.e. next PC is not known until ***1 cycles*** *after* branch/jump

Q1

I0: ADD R4 = R1 + R0;
I1: SUB R9 = R3 - R4;
I2: ADD R4 = R5 + R6;
I3: LDW R2 = MEM[R3 + 100];
I4: LDW R2 = MEM[R2 + 0];
I5: STW MEM[R4 + 100] = R2;
I6: AND R2 = R2 & R1;
I7: BEQ R9 == R1, Target;
I8: AND R9 = R9 & R1;

For the above code, complete the pipeline diagram

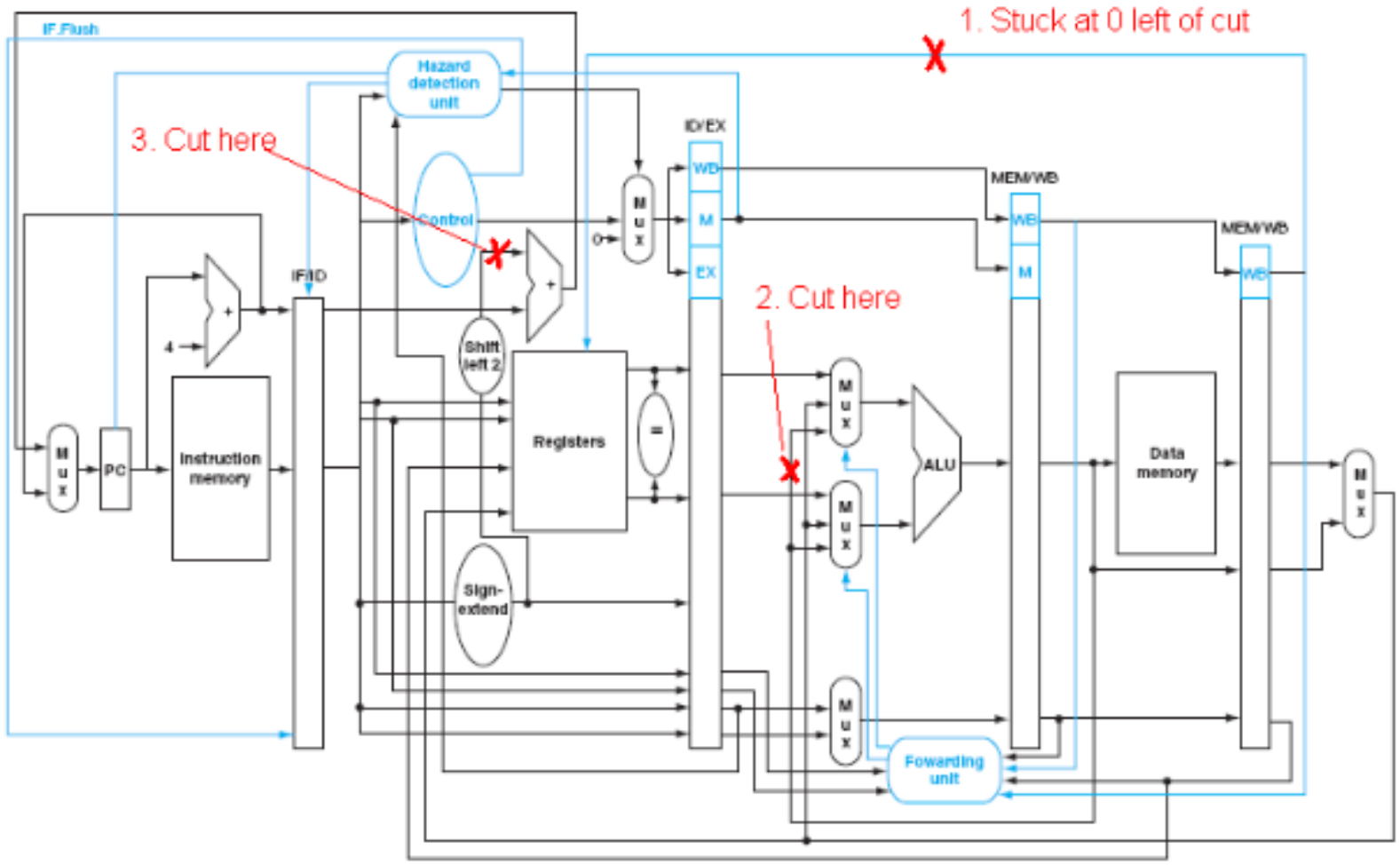
Instr.	Instruction	Dependency	Hazard Type	Stall Needed?	Why?
I0	`ADD R4, R1, R0`	—	None	❌ No	No prior dependency
I1	`SUB R9, R3, R4`	R4 ← I0	**RAW (ALU-ALU)**	❌ No	Resolved via forwarding (EX/MEM → EX)
I2	`ADD R4, R5, R6`	—	None	❌ No	Independent instruction
I3	`LDW R2, 100(R3)`	—	None	❌ No	No dependency
I4	`LDW R2, 0(R2)`	R2 ← I3	**RAW (Load-Use)**	✅ Yes	R2 needed too soon, stall inserted
I5	`STW 100(R4), R2`	R2 ← I4	**Load-Store**	❌ No	R2 needed in MEM stage, ready from WB via forwarding
I6	`AND R2, R2, R1`	R2 ← I4	**RAW (Load-Use)**	✅ Yes	Needs R2 in EX, value ready in WB → stall inserted
I7	`BEQ R9, R1, Target`	R9 ← I1	**Control + RAW**	❌ No	R9 forwarded, and branch uses delay slot
I8	`AND R9, R9, R1`	R9 ← I7	**RAW (ALU-ALU)**	❌ No	Executes after I7 writes R9, no stall needed

```
I0: ADD R4 = R1 + R0;
I1: SUB R9 = R3 - R4;
I2: ADD R4 = R5 + R6;
I3: LDW R2 = MEM[R3 + 100];
I4: LDW R2 = MEM[R2 + 0];
I5: STW MEM[R4 + 100] = R2;
I6: AND R2 = R2 & R1;
I7: BEQ R9 == R1, Target;
I8: AND R9 = R9 & R1;
```

Cycle	I0 (ADD R4)	I1 (SUB R9)	I2 (ADD R4)	I3 (LDW R2)	I4 (LDW R2)	I5 (STW R2)	I6 (AND R2)	I7 (BEQ R9)	I8 (AND R9)
T0	IF								
T1	ID	IF							
T2	EX	ID	IF						
T3	MEM	EX	ID	IF					
T4	WB	MEM	EX	ID	IF				
T5		WB	MEM	EX	ID	IF			
T6			WB	MEM	EX	ID	IF		
T7				WB	STALL	STALL	STALL	STALL	
T8					MEM	EX	ID	IF	
T9					WB	MEM	EX	ID	IF
T10						WB	MEM	EX	ID
T11							WB	MEM	EX
T12								WB	MEM
T13									WB

Q2

For the MIPS datapath shown below, several lines are marked with "X". For each one:



Describe in words the negative consequence of cutting line 1 relative to the working, unmodified processor. Give an example for a code that would fail and another one for a code that will not fail.

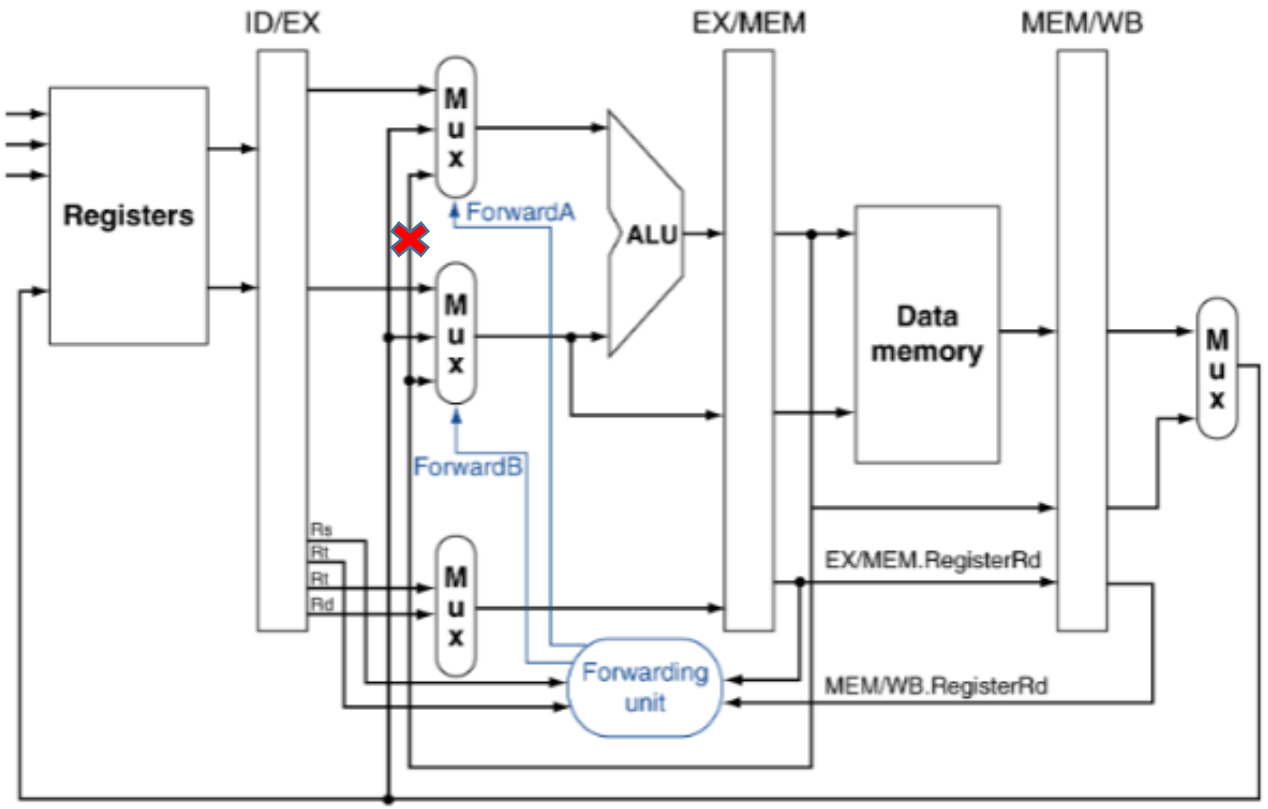
Cannot write to register file. This means that R-type and any instruction with write back to register file will fail. An example of code snippet that would fail is:

```
add $s1, $s2, $s3
```

An example of a code snippet that will not fail is:

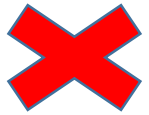
```
sw $s1, 0($s2)
```

Describe in words the negative consequence of cutting line 2. Provide a snippet of code that will fail and a code that will still work.



b. With forwarding

add $s1, $t0, $t1
add $s1, $s1, $s1



add $s1, $t0, $t1
add $s1, $t2, $s1

Describe in words the negative consequence of cutting line 3. Provide a snippet of code that will fail and a code that will still work.

Jumping to a branch target does not work. Example of code that fails:

```
addi $s1, $zero, 2
addi $s2, $zero, 2
beq $s1, $s2, exit
```

Code that will still work:

```
addi $s1, $zero, 10
addi $s2, $zero, 20
beq $s1, $s2, exit
```

Q3

Consider the 5-stage single-issued pipelined MIPS datapath consisting of Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory (MEM), and Write-Back (WB). **(10 points)**

You are given the following MIPS instruction sequence:

```
# $s0 to $s3 = 56, 30, 30, 7

# $t0 to $t4 = 7, 7, 7, 7, 7

      add $t0, $s0, $0

      and $t1, $t0, $s1

      or $t2, $t0, $s2

      sub $t3, $t0, $s3

      srl $t4, $t0, 2
```

What are the values of $t0 to $t4 at the end of cycle 7?

What are the values of $t0 to $t4 at the end of cycle 8?

What are the values of $t0 to $t4 at the end of cycle 9?

Start numbering the cycles with 1 when the add instruction enters the IF stage.

For part i. to iii. assume that the datapath is broken and there is no forwarding and no stalling.

Cycle	`add $t0, $s0, $0`	`and $t1, $t0, $s1`	`or $t2, $t0, $s2`	`sub $t3, $t0, $s3`	`srl $t4, $t0, 2`
1	IF				
2	ID	IF			
3	EX	ID	IF		
4	MEM	EX	ID	IF	
5	WB (`$t0` updated)	MEM (`$t1` uses old `$t0`)	EX	ID	IF
6		WB (`$t1` updated)	MEM (`$t2` uses old `$t0`)	EX	ID
7			WB (`$t2` updated)	MEM (`$t3` uses old `$t0`)	EX
8				WB (`$t3` updated)	MEM (`$t4` uses old `$t0`)
9					WB (`$t4` updated)

```
# $s0 to $s3 = 56, 30, 30, 7

# $t0 to $t4 = 7, 7, 7, 7, 7
```

End of Cycle 7:

- **$t0 = 56** → The result of `add $t0, $s0, $s0`, but due to no forwarding, use the old value.
- **$t1 = 6** → Computed using **stale $t0** (before `add` writes back).
- **$t2 = 31** → Computed using **stale $t0**.
- **$t3 = 7** → Computed using **stale $t0**.
- **$t4 = 7** → Computed using **stale $t0**.

Cycle	`add $t0, $s0, $0`	`and $t1, $t0, $s1`	`or $t2, $t0, $s2`	`sub $t3, $t0, $s3`	`srl $t4, $t0, 2`
1	IF				
2	ID	IF			
3	EX	ID	IF		
4	MEM	EX	ID	IF	
5	WB (`$t0` updated)	MEM (`$t1` uses old `$t0`)	EX	ID	IF
6		WB (`$t1` updated)	MEM (`$t2` uses old `$t0`)	EX	ID
7			WB (`$t2` updated)	MEM (`$t3` uses old `$t0`)	EX
8				WB (`$t3` updated)	MEM (`$t4` uses old `$t0`)
9					WB (`$t4` updated)

```
# $s0 to $s3 = 56, 30, 30, 7

# $t0 to $t4 = 7, 7, 7, 7, 7
```

At the end of **Cycle 8**, we have:

- `$t0` = **56** (Correct value from `add`)
- `$t1` = **6** (Computed using stale `$t0`)
- `$t2` = **31** (Computed using stale `$t0`)
- `$t3` = **49** (Updated in `WB` this cycle)
- `$t4` = **7** (Still using stale `$t0`, will update in next cycle)

Cycle	`add $t0, $s0, $0`	`and $t1, $t0, $s1`	`or $t2, $t0, $s2`	`sub $t3, $t0, $s3`	`srl $t4, $t0, 2`
1	IF				
2	ID	IF			
3	EX	ID	IF		
4	MEM	EX	ID	IF	
5	WB (`$t0` updated)	MEM (`$t1` uses old `$t0`)	EX	ID	IF
6		WB (`$t1` updated)	MEM (`$t2` uses old `$t0`)	EX	ID
7			WB (`$t2` updated)	MEM (`$t3` uses old `$t0`)	EX
8				WB (`$t3` updated)	MEM (`$t4` uses old `$t0`)
9					WB (`$t4` updated)

```
# $s0 to $s3 = 56, 30, 30, 7

# $t0 to $t4 = 7, 7, 7, 7, 7
```

At the end of **Cycle 9**, we have:

- `$t0` = **56** (Correct value from `add`)
- `$t1` = **6** (Stale `$t0`)
- `$t2` = **31** (Stale `$t0`)
- `$t3` = **49** (Correct from `sub`)
- `$t4` = **14** (Now updated)