ECE 552: Introduction to Computer Architecture

14. Pipelining, Hazards & Exceptions #2

Lecture notes based in part on slides created by Joshua San Miguel, Mikko Lipasti, Mark Hill, David Wood, Guri Sohi, John Shen, and Jim Smith

October 21, 2025
George Tzimpragos

Announcements

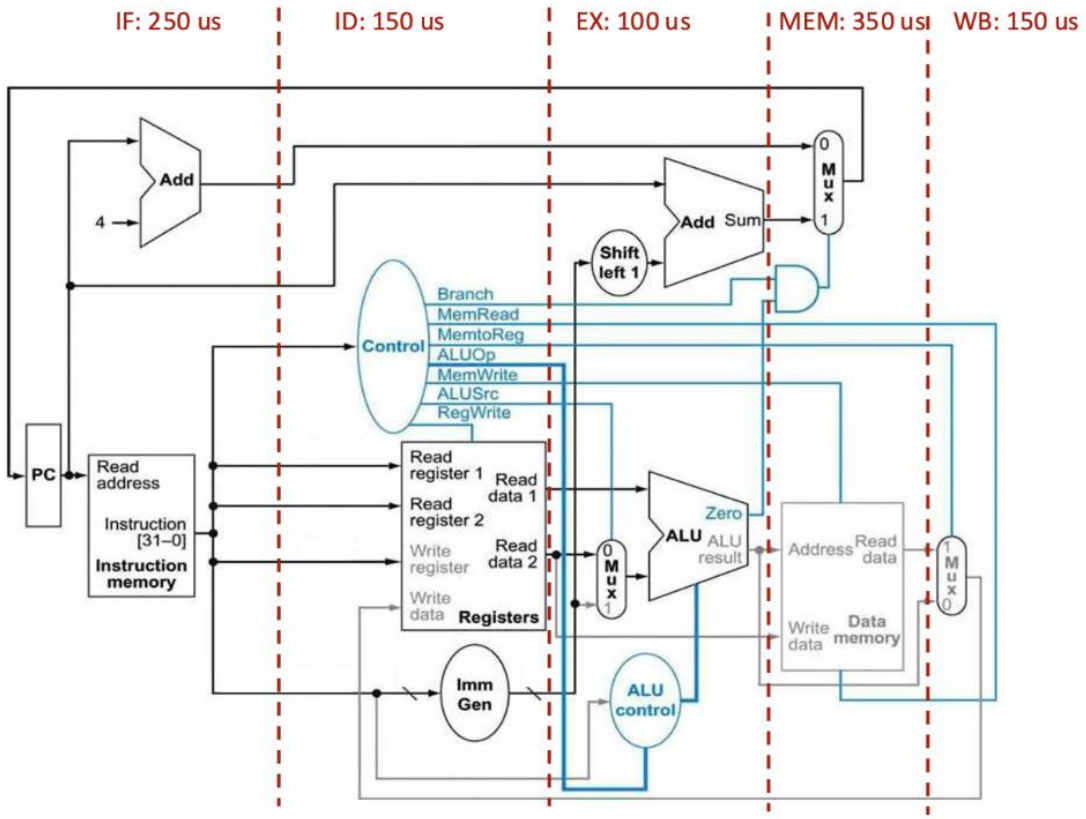
- **New grader:** Dharaneedaran Kalyanam Sendhil; kalyanamsend@wisc.edu
- **Updated office hours:**

Day	Time	Location	Person
Monday	15:00-16:00	Zoom	Dharaneedaran
Tuesday	15:00-16:00	1422 EHall	Dharaneedaran
Wednesday	15:30-16:30	1325 Comp Sci	Zhewen
Thursday	14:30-15:30	3421 EHall	George
Friday	11:00-12:00	Zoom	Rotation

- **Midterm #1:** Tuesday, Oct 28—In class

If you have an accommodation on file and do not hear from us by tomorrow, please email the instructor and TA by Thursday, Oct 23.

Review: Performance comparisons



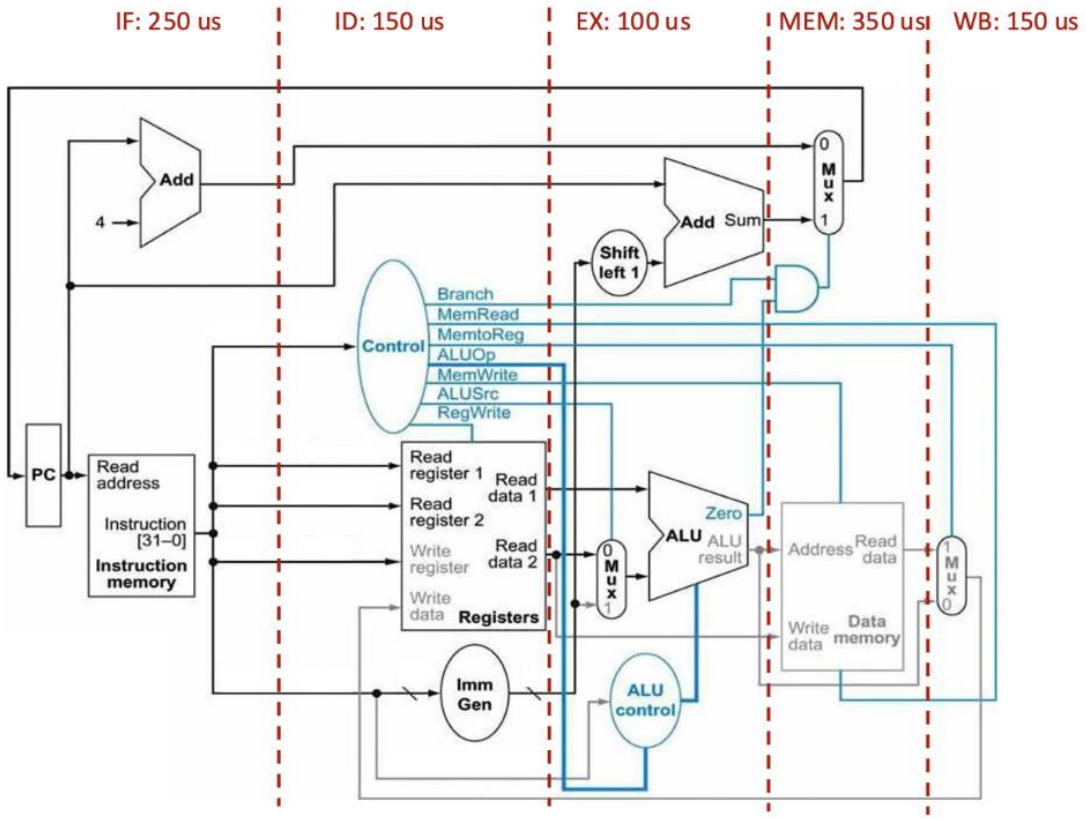
Assume a single-cycle implementation

- ❑ **Q1**: what is the latency? 1,000 us
- ❑ **Q2**: what is the throughput? 1,000 instr/s (instructions per second)

Assume a five-cycle implementation (no stalls and no register delay)

- ❑ **Q3**: what is the latency? 5*350=1,750 us
- ❑ **Q4**: what is the throughput? 2,857 instr/s (instructions per second)

Review: Performance comparisons



(A) IF: 10% improvement.

(B) EX: 35% improvement.

(C) WB: 18% improvement.

(D) ID: 20% improvement.

(E) MEM: 7% improvement.

❑ **Q5:** Which choice yields the greatest benefit for latency on the **single-cycle processor**? B

Throughput = 1/(250us+150us+0.65*100us+350us+150us)

❑ **Q6:** Which choice yields the greatest benefit for latency on the **5-cycle processor**? E

Throughput = 1/(0.93*350us)

Review: RAW data hazard

- Assume a 5-stage pipelined RV32I. Draw instruction flow in a cycle-accurate manner for the following program:

```
add x1, x2, x3 # write to x1
add x4, x1, x5 # read from x1
```

Data hazard: Read-after-write (RAW) dependency

Cycle	1	2	3	4	5	6	7	8	9	10
Instr. 1	F	D	X	M	W					
Instr. 2		F	D	X	M	W				

Review: RAW data hazard

- Assume a 5-stage pipelined RV32I. Draw instruction flow in a cycle-accurate manner for the following program:

```
add x1, x2, x3 # write to x1
add x4, x1, x5 # read from x1
```

Data hazard: Read-after-write (RAW) dependency

Cycle	1	2	3	4	5	6	7	8	9	10
Instr. 1	F	D	X	M	W					
Instr. 2		F	D*	D*	D*	D	X	M	W	

Stalls

Review: RAW data hazard

- Assume a 5-stage pipelined RV32I. Draw instruction flow in a cycle-accurate manner for the following program:

```
add x1, x2, x3 # write to x1
add x4, x1, x5 # read from x1
```

Data hazard: Read-after-write (RAW) dependency

Cycle	1	2	3	4	5	6	7	8	9	10
Instr. 1	F	D	X	M	W					
Instr. 2		F	D*	D*	D	X	M	W		

* The standard assumption in pipelined processors is that **writes happen early in the cycle** and **reads happen later in the cycle** This allows a write and read to occur in the same cycle.

Review: WAR data hazard

- Assume a 5-stage pipelined RV32I. Draw instruction flow in a cycle-accurate manner for the following program:

```
add x1, x2, x3 # reads x2 and x3, writes x1
add x2, x4, x5 # reads x4 and x5, writes x2
```

Data hazard*: Write-after-read (WAR) dependency

* Out-of-order execution

Cycle	1	2	3	4	5	6	7	8	9	10
Instr. 1	F	D	X	M	W					
Instr. 2		F	D	X	M	W				

Review: WAW data hazard

- Assume a 5-stage pipelined RV32I. Draw instruction flow in a cycle-accurate manner for the following program:

```
add x1, x2, x3 # reads x2 and x3, writes x1
add x1, x4, x5 # reads x4 and x5, writes x1
```

Data hazard*: Write-after-write (WAW) dependency

* multi-cycle functional units, register file structural hazard, etc.

Cycle	1	2	3	4	5	6	7	8	9	10
Instr. 1	F	D	X	M	W					
Instr. 2		F	D	X	M	W				

Pipeline hazards

- What are pipeline hazards?
  - Potential violations of program dependences
  - Must ensure program dependences are not violated

- How to resolve?
  - Static: compiler/programmer guarantees correctness
  - Dynamic: processor hardware checks at runtime

Pipeline hazards

- What are pipeline hazards?
  - Potential violations of program dependences
  - Must ensure program dependences are not violated
- How to resolve?
  - Static: compiler/programmer guarantees correctness
  - Dynamic: processor hardware checks at runtime

Participation question: In a 5-stage in-order scalar pipeline, which types of data dependencies—RAW, WAR, or WAW—can be pipeline hazards?

Pipeline data hazards

- RAW data dependence:
  - **Hazard:** when sub is reading [$1] in ID stage, add is still in EX stage and has not written the new value of [$1] to the register file yet

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

insn\cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
add	F	D	X	M	W								
sub		F	D	X	M	W							

Pipeline data hazards

- WAR data dependence:
  - **Not a hazard:** every instruction performs WB after ID, so a younger instruction will never write a register (WB) before an older instruction's read (ID)

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

insn\cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
add	F	D	X	M	W								
sub		F	D	X	M	W							

Pipeline data hazards

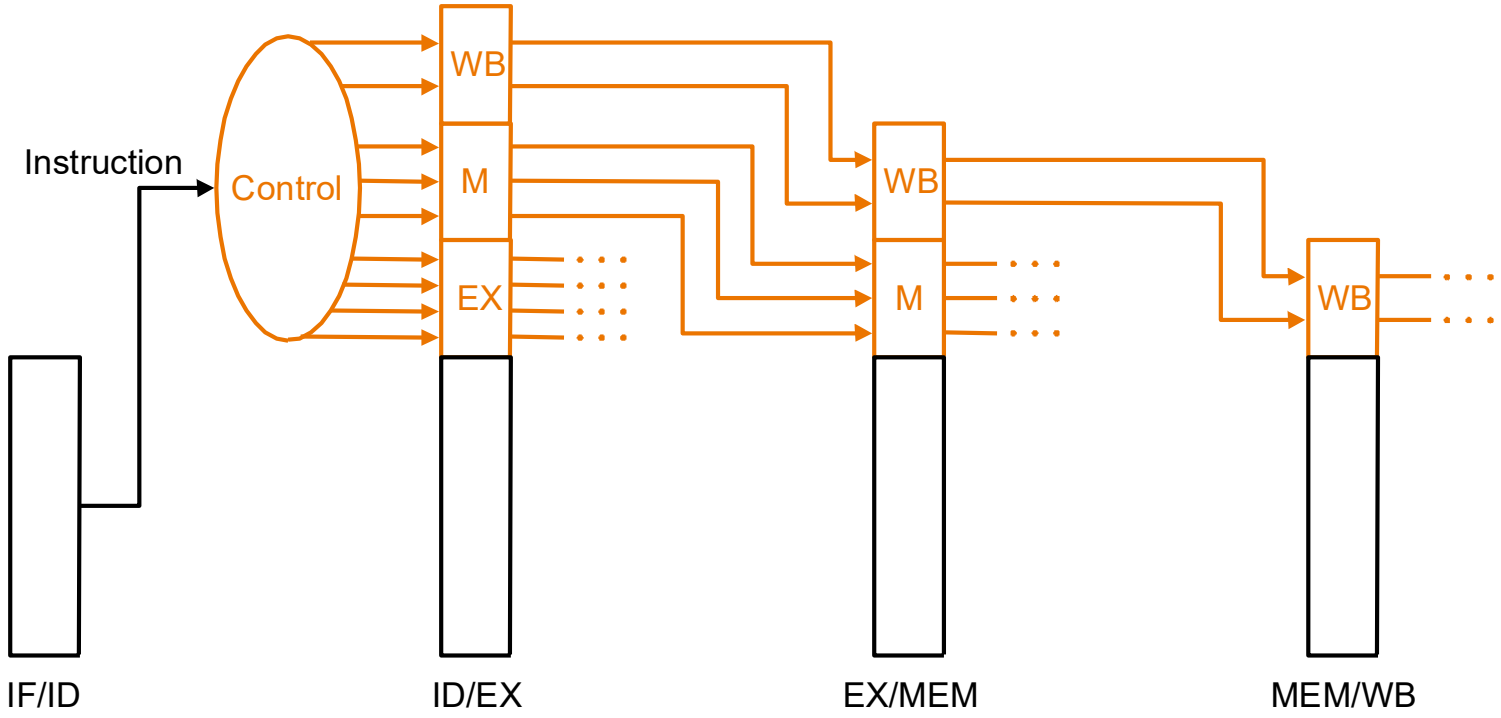
- WAW data dependence:
  - **Not a hazard:** instructions perform WB in order, so a younger instruction will never write a register (WB) before an older instruction's write (WB)

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

insn\cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
add	F	D	X	M	W								
sub		F	D	X	M	W							

Resolving RAW hazards

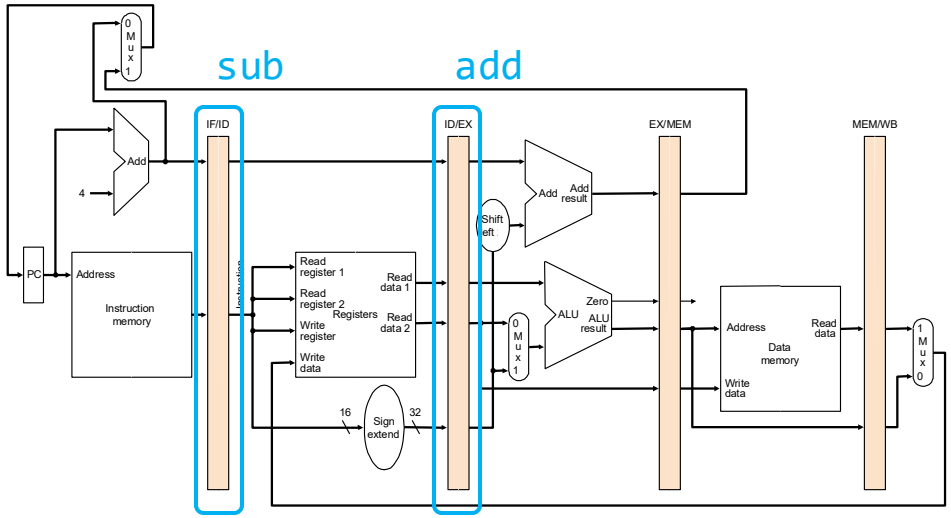
- **Step 1:** Detect hazard at ID stage
  - Compare control signals of other stages



Resolving RAW hazards

- **Step 1:** Detect hazard at ID stage
  - Compare control signals of other stages
  - Hazard if:
    - ID/EX.WriteRegister == IF/ID.RegisterRs1, or
    - ID/EX.WriteRegister == IF/ID.RegisterRs2

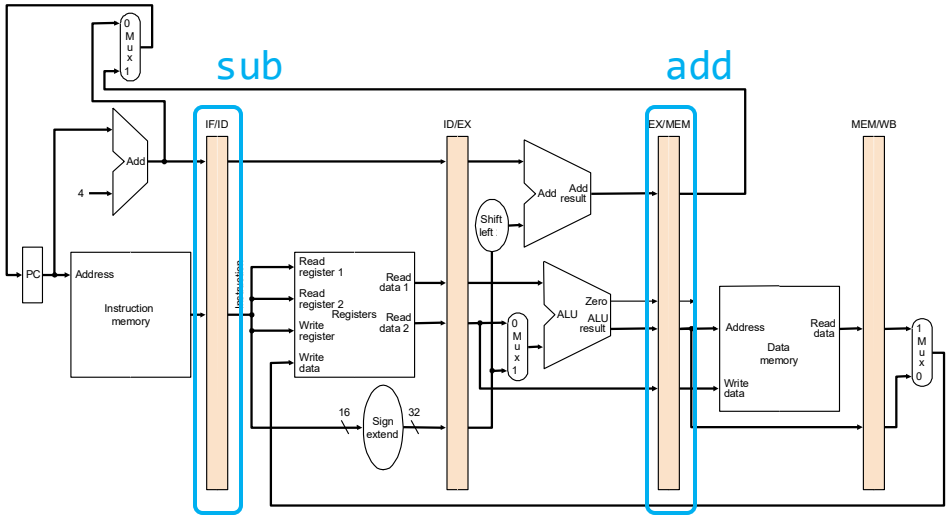
*R-format




```
add $1, $2, $3
sub $4, $5, $1
```

insn\cycle	1	2	3	4	5	6	7	8	9
add	F	D	X	M	W				
sub		F	D	X	M	W			

Resolving RAW hazards



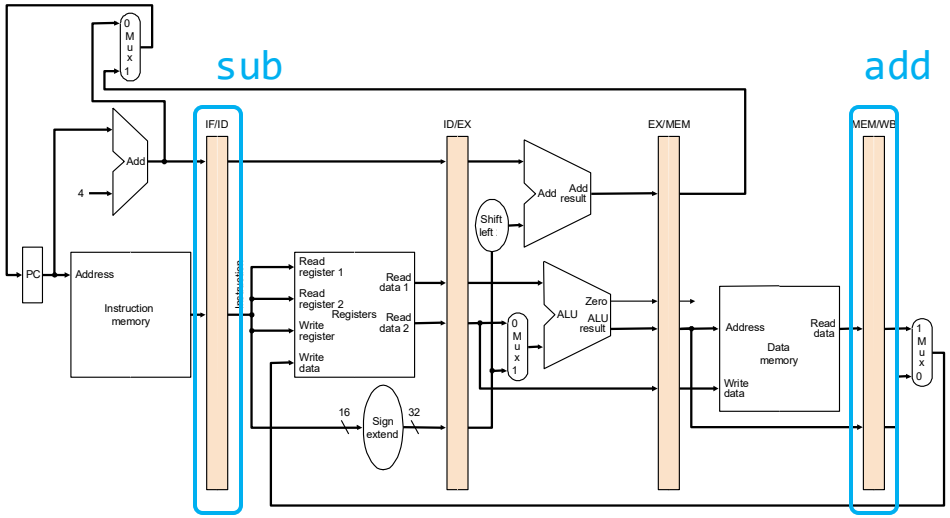
- **Step 1:** Detect hazard at ID stage
  - Compare control signals of other stages
  - Hazard if:
    - EX/MEM.WriteRegister == IF/ID.RegisterRs1, or
    - EX/MEM.WriteRegister == IF/ID.RegisterRs2

*R-format

```
add $1, $2, $3
…
sub $4, $5, $1
```

insn\cycle	1	2	3	4	5	6	7	8	9
add	F	D	X	M	W				
…									
sub			F	D	X	M	W		

Resolving RAW hazards



- **Step 1:** Detect hazard at ID stage
  - Compare control signals of other stages
  - Hazard if:
    - MEM/WB.WriteRegister == IF/ID.RegisterRs1, or
    - MEM/WB.WriteRegister == IF/ID.RegisterRs2

*R-format

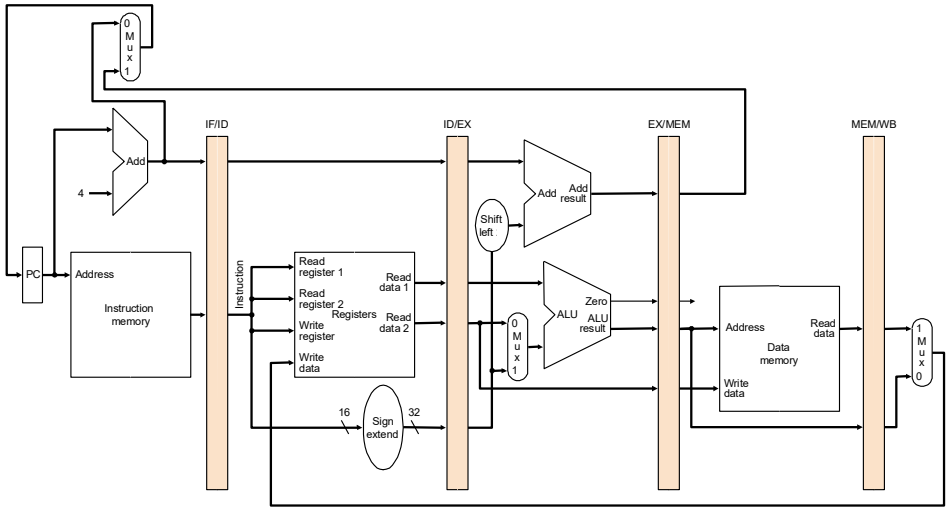
```
add $1, $2, $3
…
…
sub $4, $5, $1
```

insn\cycle	1	2	3	4	5	6	7	8	9
add	F	D	X	M	W				
…									
…									
sub				F	D	X	M	W	

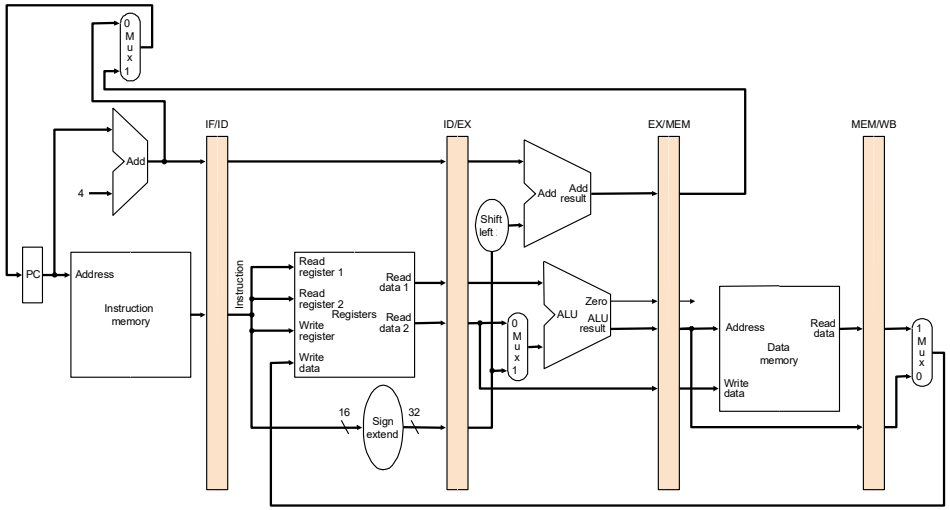
Question

- Does the following program lead to a RAW hazard?

```
sw x1, 0(x2)
sub x4, x5, x1
```



Question



- Does the following program lead to a RAW hazard?

```
sw x1, 0(x2)
sub x4, x5, x1
```

  - `sw x1,0(x2)` **does not write** to `x1` (it *reads* `x1` to store its value into memory)
  - `sub x4,x5, x1` reads `x1`

* We need to check for more conditions:
  - WriteRegister not necessarily used (e.g., sw)
  - RegisterRs1/RegisterRs2 not necessarily read (e.g., addi, jump)
  - Etc.

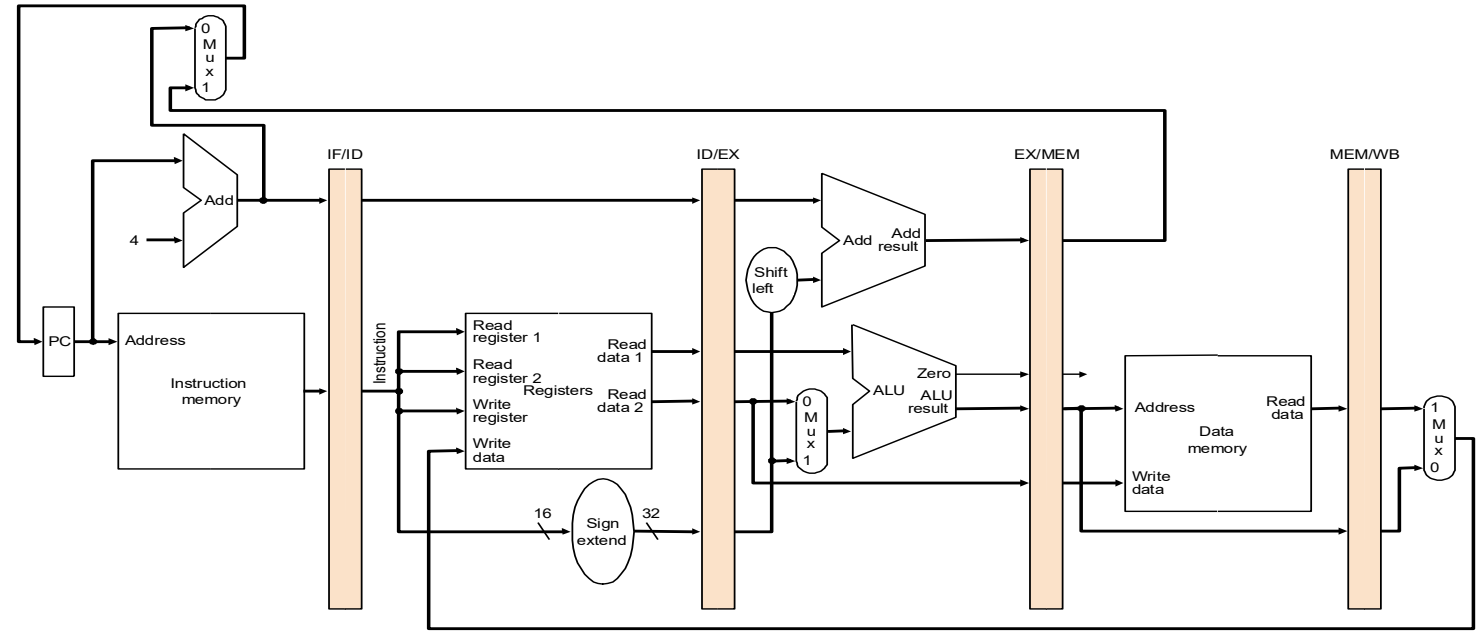
Resolving RAW hazards

- **Step 2a:** *Stall* the reader instruction at ID stage
  - Instructions in IF and ID stay there for next cycle
  - Send *nop* (no operation, or *bubble*) ahead of reader until hazard is resolved

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

insn\cycle	1	2	3	4	5	6	7	8	9	10
add	F	D	X	M	W					
sub		F	D*	D*	D*	D	X	M	W	
lw			F*	F*	F*	F	D	X	M	W

Resolving RAW hazards



cycle 3:

add x1, x2, x3
sub x4, x5, x1
lw x6, 0(x7)

insn\cycle	1	2	3	4	5	6	7	8	9	10
add	F	D	X	M	W					
sub		F	D*	D*	D*	D	X	M	W	
lw			F*	F*	F*	F	D	X	M	W

Resolving RAW hazards

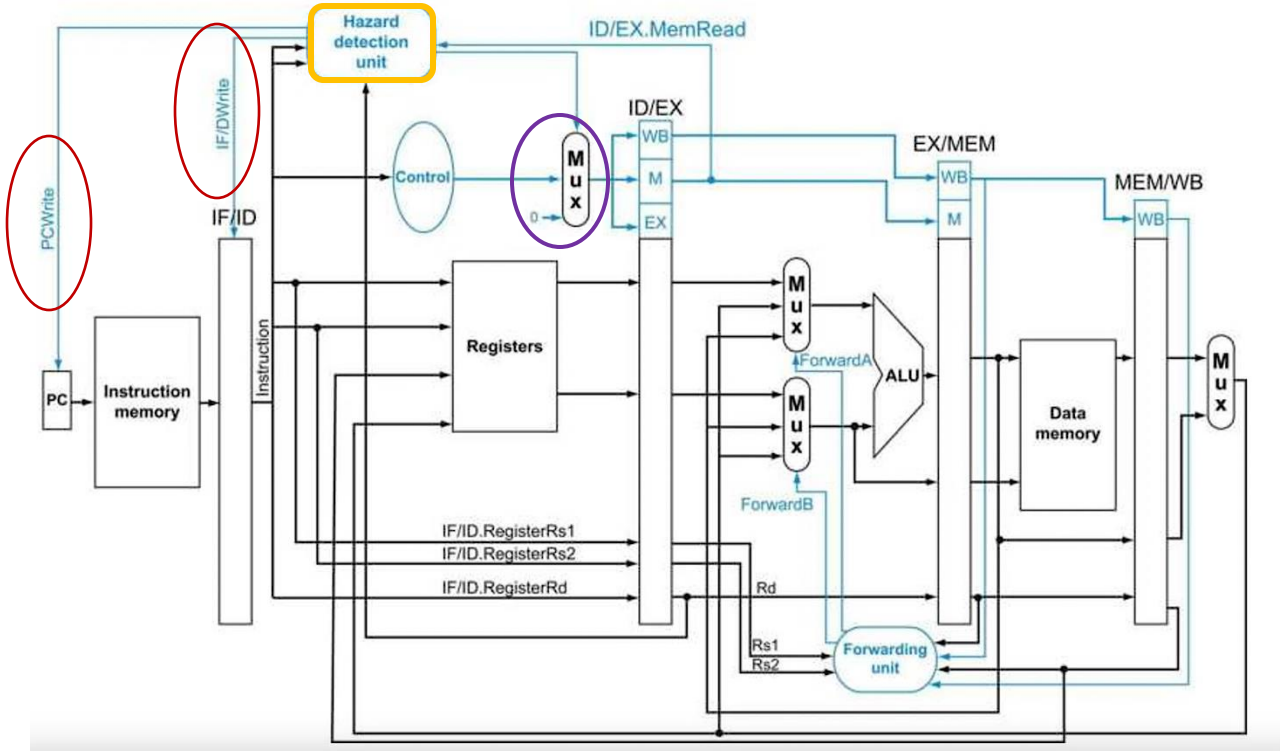
```
add x1, x2, x3
nop
nop
nop
sub x4, x5, x1
lw x6, 0(x7)
```

insn\cycle	1	2	3	4	5	6	7	8	9	10
add	F	D	X	M	W					
sub		F	D*	D*	D*	D	X	M	W	
lw			F*	F*	F*	F	D	X	M	W

Resolving RAW hazards

2. Ensure that the machine state does not get affected
   a. Set control fields to 0 if the hazard test is true.

1. Stop progression
   a. Disable the write-enable signal for the PC register
   b. Disable the write-enable signal for the IF/ID pipeline register

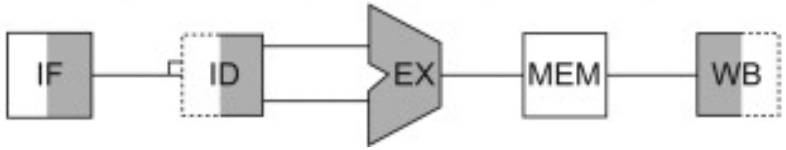



```
add x1, x2, x3
nop
nop
nop
sub x4, x5, x1
lw x6, 0(x7)
```

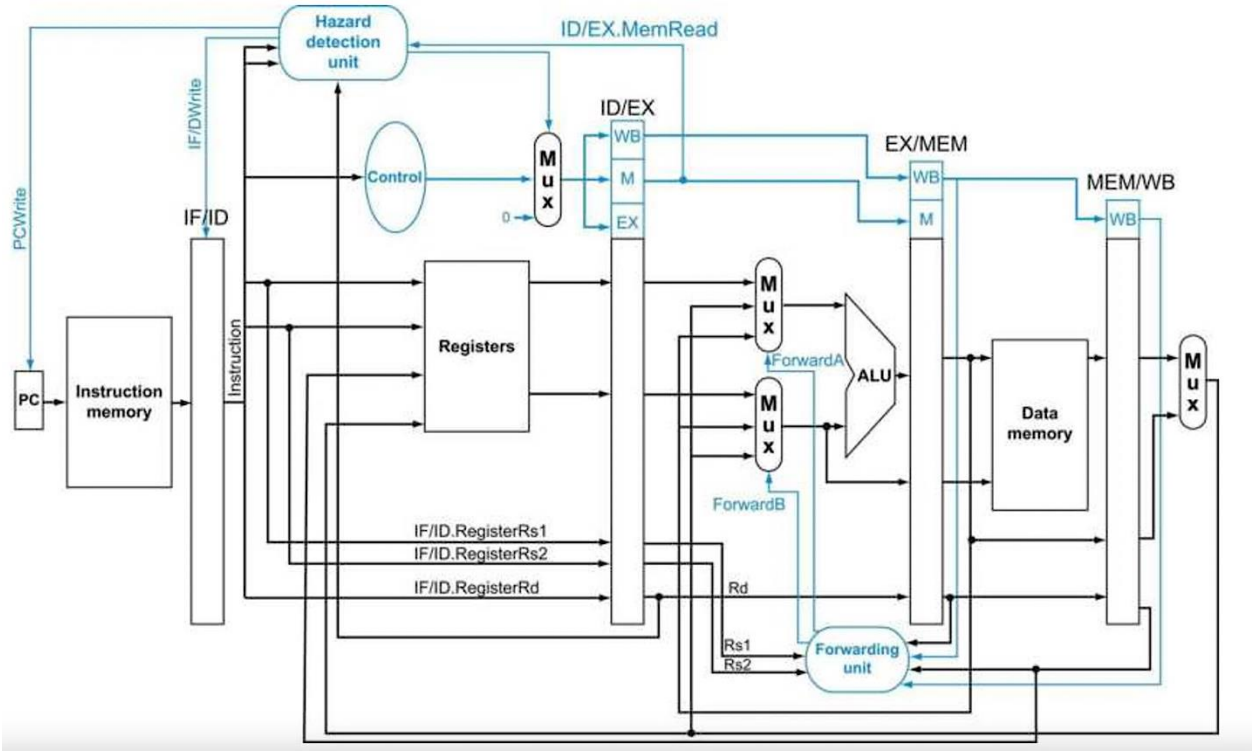
insn\cycle	1	2	3	4	5	6	7	8	9	10
add	F	D	X	M	W					
sub		F	D*	D*	D*	D	X	M	W	
lw			F*	F*	F*	F	D	X	M	W

Resolving RAW hazards

- **Step 2b:** Stall with *register file bypassing*
  - Implement a write-before-read RF where the instruction at WB writes its result into the RF in the first half of the cycle and then the instruction at ID reads its operands.



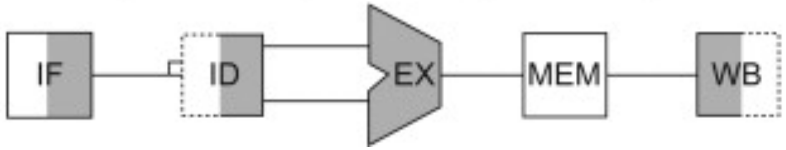

```
add x1, x2, x3
nop
nop
sub x4, x5, x1
lw x6, 0(x7)
```



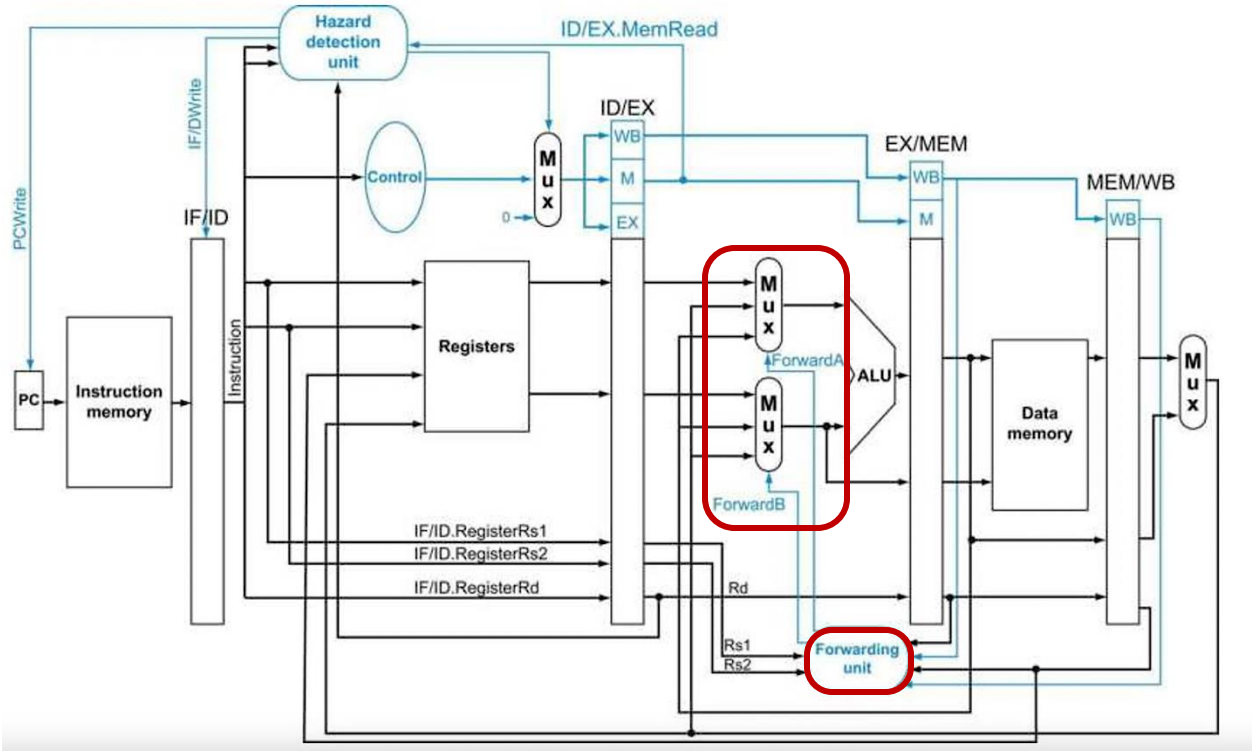
insn\cycle	1	2	3	4	5	6	7	8	9	10
add	F	D	X	M	W					
sub		F	D*	D*	D	X	M	W		
lw			F*	F*	F	D	X	M	W	

Resolving RAW hazards

- **Step 2b:** Stall with *register file bypassing*
  - Implement a write-before-read RF where the instruction at WB writes its result into the RF in the first half of the cycle and then the instruction at ID reads its operands.



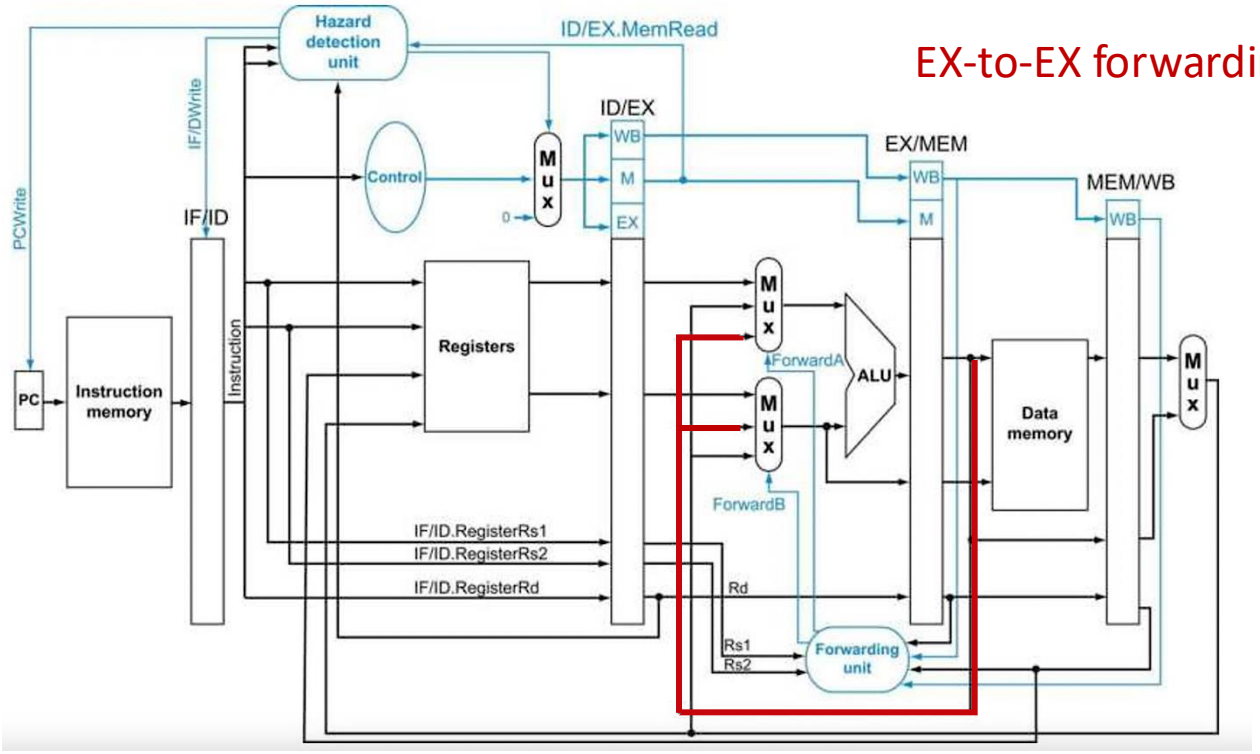

```
add x1, x2, x3
nop
nop
sub x4, x5, x1
lw x6, 0(x7)
```



insn\cycle	1	2	3	4	5	6	7	8	9	10
add	F	D	X	M	W					
sub		F	D*	D*	D	X	M	W		
lw			F*	F*	F	D	X	M	W	

Resolving RAW hazards

- **Step 2c:** RF bypassing and *data forwarding*
  - For some instructions, their results are ready before WB
    - ✓ Ready after EX for arithmetic and logic
  - For all instructions, they do not need their operands until EX
  - Add forwarding muxes to inputs of ALU

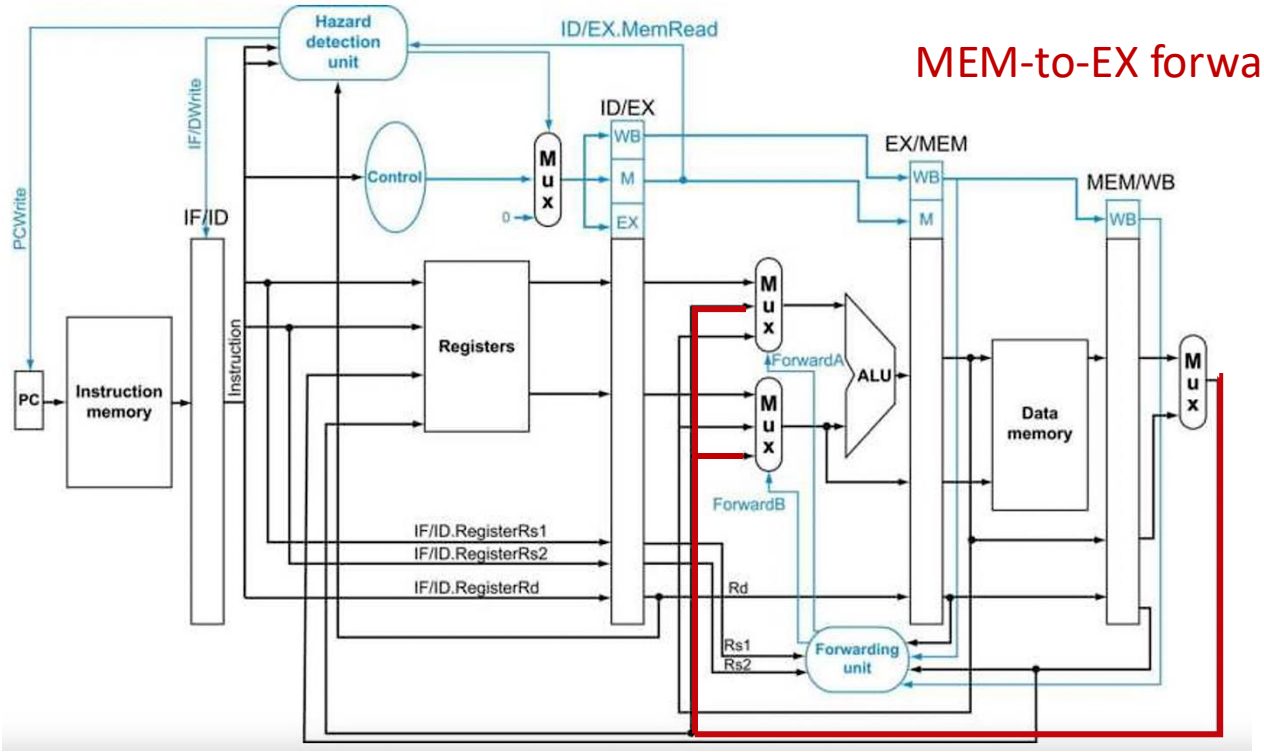



```
add x1, x2, x3
sub x4, x5, x1
lw x6, 0(x7)
```

insn\cycle	1	2	3	4	5	6	7	8	9	10
add	F	D	X	M	W					
sub		F	D	X	M	W				
lw			F	D	X	M	W			

Resolving RAW hazards

- **Step 2c:** RF bypassing and *data forwarding*
  - For some instructions, their results are ready before WB
    - ✓ Ready after EX for arithmetic and logic
  - For all instructions, they do not need their operands until EX
  - Add forwarding muxes to inputs of ALU




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

insn\cycle	1	2	3	4	5	6	7	8	9	10
add	F	D	X	M	W					
…										
sub			F	D	X	M	W			

Resolving RAW hazards

- **Step 2c:** RF bypassing and *data forwarding*
  - For some instructions, their results are ready before WB
    - ✓ Ready after EX for arithmetic and logic
  - For all instructions, they do not need their operands until EX
  - Add forwarding muxes to inputs of ALU

- Not all operands are needed at EX stage
  - ✓ Store instructions do not need RegisterRs until MEM stage

```
lw x1, 0(x2) # Load from memory into x1
sw x1, 4(x3) # Store x1 to memory
```

- The `lw` doesn't have the data until the MEM stage
- The `sw` needs x1 in the MEM stage

MEM-to-MEM forwarding

Exercise

```
lw x1, 0(x2)
lw x3, 4(x1)
add x5, x4, x3
```

(a) Assume **no RF bypassing** and **no forwarding**
(b) Assume **RF bypassing** but **no forwarding**
(c) Assume **RF bypassing** and **forwarding**

insn\cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
lw x1	F	D	X	M	W								

Questions?