18-344 Recitation 2

09/12/2025

Outline

1. Logistics
2. Review
3. Labs
    a. Lab 0
    b. Lab 1

Logistics

- Lab 0 due September 15 (next Monday) (in 3 days)
  - Review last week's recitation if needed
- Homework 1 due September 14 (Sunday) (in 2 days)
  - Covers materials from lectures 1-4
- ***NEW*** Lab 1 will be released on September 15 (next Monday) (in 3 days)
  - From lab 1 onwards, all labs will be ***partnered (group of 2 unless otherwise approved)***. Start looking for a lab partner now!
    - Use the Piazza post and/or Slack to find your lab partner.

Review

Speedup

$$S = \frac{t_{\text{base}}}{t_{\text{improved}}}$$

Speedup w/ Example

When we say: A is X times faster than B, we mean:

$$X = \frac{T_B}{T_A}$$

Speedup w/ Example

“You friend proposes an optimization which would speed up ***all*** operations by 30%”

$$X = 1.3 = \frac{T_{\text{original}}}{T_{\text{optimized}}}$$

ISA Design

ISA is the contract between hardware and software. It's what the hardware offers and what the software would expect to be available.

If something is not necessarily required to be known by ***both*** hardware and software, it shouldn't be in the ISA[1].

[1] Exceptions[2] apply.

[2] These exceptions are out of the scope of this class.

So, what's usually in an ISA?

- ***Architectural*** registers
  - e.g. `x0` … `x31` in RISC-V's RV32/RV64 ISAs
- Instruction ***definitions***
  - e.g. `ADD`, `XOR`, `LOAD`, `STORE`, etc.
  - Definitions include:
    - What the operation and operands are
    - What the side-effects are
    - How the instruction should be encoded
- Memory

  More specifically, the idea of ***the existence of memory***.
  - The ISA defines how large the memory address space is, and how to use the memory.
    - e.g. RV32I has a byte-addressable memory address space of 2^32 bytes.

Aside: RISC-V instruction syntax

```
opcode rd, rs1, rs2
⇒ rd = op(rs1, rs2)

opcode rd, rs1, imm
⇒ rd = op(rs1, imm)
```

... for more, check out
https://github.com/riscv/riscv-isa-manual/

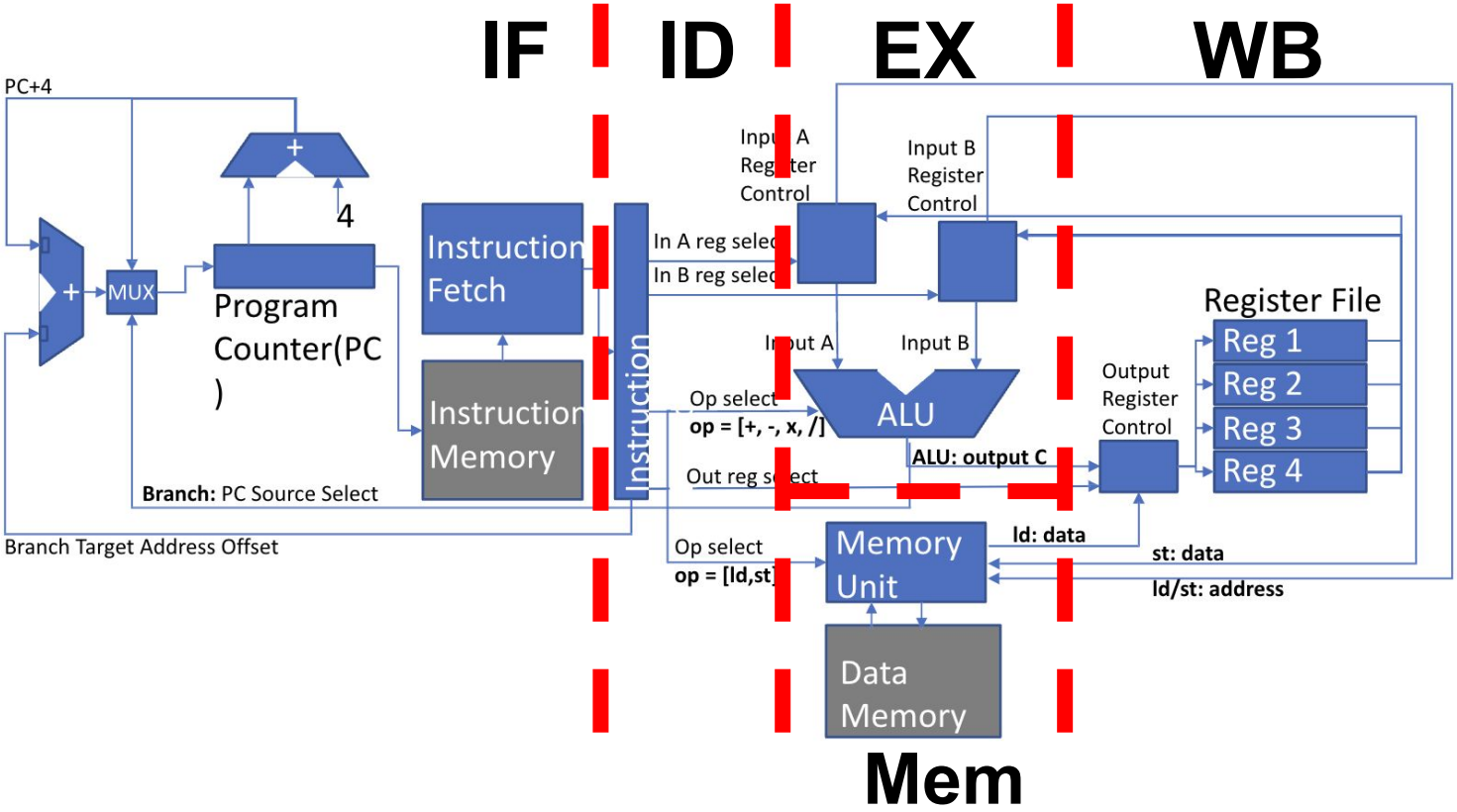
So, what's usually ***not*** in an ISA?

- ***Microarchitectural*** registers
  - i.e. Physical register files used to implement architectural registers are not in the ISA.
- Instruction ***implementation***
  - e.g. It is 100% valid for the implementation of an ISA to choose to use a single adder to implement `MUL`, as long as the result of the `MUL` adheres to what the ISA defines.
  - e.g. It is also 100% valid for the implementation of an ISA to choose to use a slow adder for some `ADD`s and a faster adder for other `ADD`s under different circumstances[1].
- What memory is made of
  - e.g. ISA doesn't tell you how much physical memory is available.
  - e.g. ISA doesn't specify how much time it takes to write to/read from memory.

[1] This is true as long as all the `ADD`s are using the same instruction and the ISA does not specify instruction latencies[2] (whether cycles or wall-plug time).

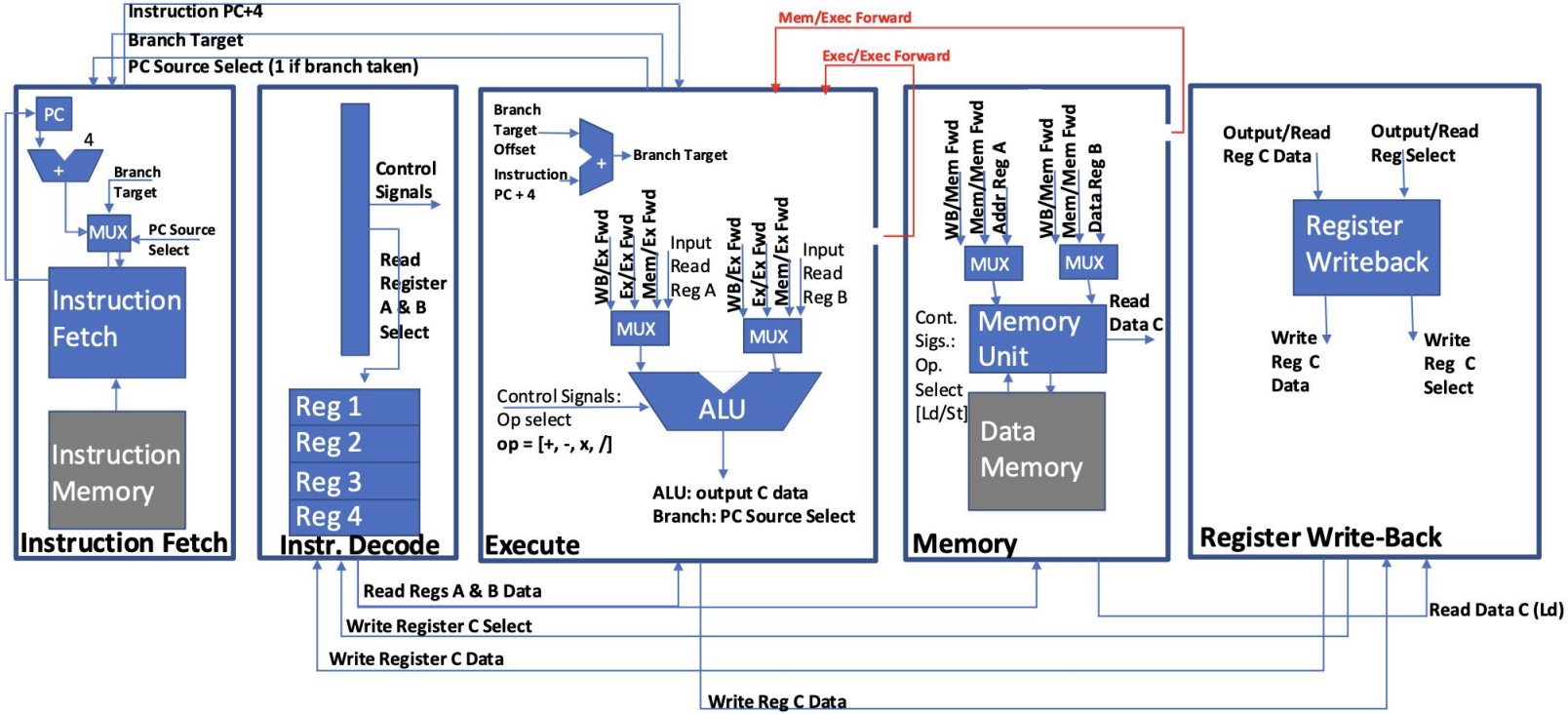
[2] For this course, we assume that ISAs don't specify instruction latencies.

Basic RISC-V Datapath



5-Stage Pipelined Processor Datapath
a.k.a. The 5-Stage Pipeline

with Branch Prediction and EX->EX & Mem->EX Forwarding



5-Stage Pipelined Execution

<u>WITHOUT</u> EX->EX & Mem->EX Forwarding

Cycle	IF	ID	EX	Mem	WB
1	`lw x1,0×10(x0)`				

5-Stage Pipelined Execution

<u>WITHOUT</u> EX->EX & Mem->EX Forwarding

Cycle	IF	ID	EX	Mem	WB
1	`lw x1,0×10(x0)`				
2	`addi x2,x1,0×1`	`lw x1,0×10(x0)`			

5-Stage Pipelined Execution

<u>WITHOUT</u> EX->EX & Mem->EX Forwarding

Cycle	IF	ID	EX	Mem	WB
1	`lw x1,0×10(x0)`				
2	`addi x2,x1,0×1`	`lw x1,0×10(x0)`			
3	`add x3,x2,x1`	`addi x2,x1,0×1`	`lw x1,0×10(x0)`		

5-Stage Pipelined Execution

***WITHOUT* EX->EX & Mem->EX Forwarding**

Cycle	IF	ID	EX	Mem	WB
1	`lw x1,0×10(x0)`				
2	`addi x2,x1,0×1`	`lw x1,0×10(x0)`			
3	`add x3,x2,x1`	`addi x2,x1,0×1`	`lw x1,0×10(x0)`		
4			~~`addi x2,x1,0×1`~~	`lw x1,0×10(x0)`	

`addi` references a value in `x1`, but the previous `lw` has not yet put a new value in `x1` yet!

5-Stage Pipelined Execution

WITHOUT EX->EX & Mem->EX Forwarding

Cycle	IF	ID	EX	Mem	WB
1	`lw x1,0×10(x0)`				
2	`addi x2,x1,0×1`	`lw x1,0×10(x0)`			
3	`add x3,x2,x1`	`addi x2,x1,0×1`	`lw x1,0×10(x0)`		
4	`add x3,x2,x1`	`addi x2,x1,0×1`	1	`lw x1,0×10(x0)`	

Solution: insert a pipeline bubble in the EX stage to force the `addi` to wait until `x1` has the new value. In practice people use the `nop` instruction or clear the control signals to create the pipeline bubble. We also call this "stalling the EX stage".

5-Stage Pipelined Execution

***WITHOUT* EX->EX & Mem->EX Forwarding**

Cycle	IF	ID	EX	Mem	WB
1	`lw x1,0×10(x0)`				
2	`addi x2,x1,0×1`	`lw x1,0×10(x0)`			
3	`add x3,x2,x1`	`addi x2,x1,0×1`	`lw x1,0×10(x0)`		
4	`add x3,x2,x1`	`addi x2,x1,0×1`	1	`lw x1,0×10(x0)`	
5	`add x3,x2,x1`	`addi x2,x1,0×1`	2	1	`lw x1,0×10(x0)`

Another pipeline bubble in the Mem stage is also needed because x1's content won't be updated until the end of the WB stage.

5-Stage Pipelined Execution

***WITHOUT* EX->EX & Mem->EX Forwarding**

Cycle	IF	ID	EX	Mem	WB
1	`lw x1,0×10(x0)`				
2	`addi x2,x1,0×1`	`lw x1,0×10(x0)`			
3	`add x3,x2,x1`	`addi x2,x1,0×1`	`lw x1,0×10(x0)`		
4	`add x3,x2,x1`	`addi x2,x1,0×1`	1	`lw x1,0×10(x0)`	
5	`add x3,x2,x1`	`addi x2,x1,0×1`	2	1	`lw x1,0×10(x0)`
6	`sw x3,0×14(x0)`	`add x3,x2,x1`	`addi x2,x1,0×1`	2	1

The `addi` is cleared to proceed once `x1` has been updated (`lw` passes the WB stage). Note that pipeline bubbles also proceed.

5-Stage Pipelined Execution

***WITHOUT* EX->EX & Mem->EX Forwarding**

Cycle	IF	ID	EX	Mem	WB
1	`lw x1,0×10(x0)`				
2	`addi x2,x1,0×1`				
3	`add x3,x2,x1`				
4	`add x3,x2,x1`				
5	`add x3,x2,x1`	`addi x2,x1,0×1`	2	1	`lw x1,0×10(x0)`
6	`sw x3,0×14(x0)`	`add x3,x2,x1`	`addi x2,x1,0×1`	2	1
7			~~`add x3,x2,x1`~~	`addi x2,x1,0×1`	2

Wait... `add` references a value in `x2`, but the previous `addi` has not yet put a new value in `x2` yet! Also, note that `add`'s reference to `x1` won't be an issue here.

5-Stage Pipelined Execution

***WITHOUT* EX->EX & Mem->EX Forwarding**

Cycle	IF	ID	EX	Mem	WB
1	lw x1,0×10(x0)				
2	addi x2,x1,0×1	lw x1,0×10(x0)			
3	add x3,x2,x1	addi x2,x1,0×			
4	add x3,x2,x1	addi x2,x1,0×			
5	add x3,x2,x1	addi x2,x1,0×1	2	1	lw x1,0×10(x0)
6	sw x3,0×14(x0)	add x3,x2,x1	addi x2,x1,0×1	2	1
7	sw x3,0×14(x0)	add x3,x2,x1	3	addi x2,x1,0×1	2

Solution: more pipeline bubbles...

5-Stage Pipelined Execution

***WITHOUT* EX->EX & Mem->EX Forwarding**

Cycle	IF	ID	EX	Mem	WB
1	lw x1,0×10(x0)				
2	addi x2,x1,0×1	lw x1,0×10(x0)			
3	add x3,x2,x1	addi x2,x1,0×			
4	add x3,x2,x1	addi x2,x1,0×			
5	add x3,x2,x1	addi x2,x1,0×1	2	1	lw x1,0×10(x0)
6	sw x3,0×14(x0)	add x3,x2,x1	addi x2,x1,0×1	2	1
7	sw x3,0×14(x0)	add x3,x2,x1	3	addi x2,x1,0×1	2
8	sw x3,0×14(x0)	add x3,x2,x1	4	3	addi x2,x1,0×1

Solution: more pipeline bubbles...

5-Stage Pipelined Execution

***WITHOUT* EX->EX & Mem->EX Forwarding**

Cycle	IF	ID	EX	Mem	WB
1	lw x1,0×10(x0)				
2	addi x2,x1,0×1	lw x1,0×10(x0)			
3	add x3,x2,x1	addi x2,x1,0			
4	add x3,x2,x1	addi x2,x1,0			
5	add x3,x2,x1	addi x2,x1,0×1	2	1	lw x1,0×10(x0)
6	sw x3,0×14(x0)	add x3,x2,x1	addi x2,x1,0×1	2	1
7	sw x3,0×14(x0)	add x3,x2,x1	3	addi x2,x1,0×1	2
8	sw x3,0×14(x0)	add x3,x2,x1	4	3	addi x2,x1,0×1
9	add x0,x0,x0	sw x3,0×14(x0)	add x3,x2,x1	4	3

Same deal as before, add is allowed to proceed once x2's value has been updated.

5-Stage Pipelined Execution

WITHOUT EX->EX & Mem->EX Forwarding

Cycle	IF	ID	EX	Mem	WB
1	lw x1,0×10(x0)				
2	addi x2,x1,0×1	lw x1,0×10(x0)			
3	add x3,x2,x1	addi			
4	add x3,x2,x1	addi			
5	add x3,x2,x1	addi x2,x1,0×1	2	1	lw x1,0×10(x0)
6	sw x3,0×14(x0)	add x3,x2,x1	addi x2,x1,0×1	2	1
7	sw x3,0×14(x0)	add x3,x2,x1	3	addi x2,x1,0×1	2
8	sw x3,0×14(x0)	add x3,x2,x1	4	3	addi x2,x1,0×1
9	add x0,x0,x0	sw x3,0×14(x0)	add x3,x2,x1	4	3

There isn't enough space to show this – after cycle 11, this add instruction will exit the pipeline.

5-Stage Pipelined Execution

<u>WITHOUT</u> EX->EX & Mem->EX Forwarding

What's the CPI of the pipeline for the first three instructions?

Cycles = 11, Instructions = 3
=> CPI = 11/3 ~= 3.67

$$\mathrm{CPI} = \frac{\mathrm{Cycles}}{\mathrm{Instructions}}$$

What's the IPC of the pipeline for the first three instructions?

Instructions = 3, Cycles = 11
=> IPC = 3/11 ~= 0.273

$$\mathrm{IPC} = \frac{\mathrm{Instructions}}{\mathrm{Cycles}}$$

Ideally, you'd want IPC to get as close to 1 as possible.

5-Stage Pipelined Execution

<u>WITH</u> EX->EX & Mem->EX Forwarding

Cycle	IF	ID	EX	Mem	WB
1	`lw x1,0×10(x0)`				

5-Stage Pipelined Execution

<u>WITH</u> EX->EX & Mem->EX Forwarding

Cycle	IF	ID	EX	Mem	WB
1	`lw x1,0×10(x0)`				
2	`addi x2,x1,0×1`	`lw x1,0×10(x0)`			

5-Stage Pipelined Execution

<u>WITH</u> EX->EX & Mem->EX Forwarding

Cycle	IF	ID	EX	Mem	WB
1	`lw x1,0×10(x0)`				
2	`addi x2,x1,0×1`	`lw x1,0×10(x0)`			
3	`add x3,x2,x1`	`addi x2,x1,0×1`	`lw x1,0×10(x0)`		

5-Stage Pipelined Execution

<u>WITH</u> EX->EX & Mem->EX Forwarding

Cycle	IF	ID	EX	Mem	WB
1	`lw x1,0×10(x0)`				
2	`addi x2,x1,0×1`	`lw x1,0×10(x0)`			
3	`add x3,x2,x1`	`addi x2,x1,0×1`	`lw x1,0×10(x0)`		
4			~~`addi x2,x1,0×1`~~	`lw x1,0×10(x0)`	

`addi` references a value in `x1`, but the previous `lw` has not yet put a new value in `x1` yet!

5-Stage Pipelined Execution

<u>WITH</u> EX->EX & Mem->EX Forwarding

Cycle	IF	ID	EX	Mem	WB
1	`lw x1,0×10(x0)`				
2	`addi x2,x1,0×1`	`lw x1,0×10(x0)`			
3	`add x3,x2,x1`	`addi x2,x1,0×1`	`lw x1,0×10(x0)`		
4	`add x3,x2,x1`	`addi x2,x1,0×1`	1	`lw x1,0×10(x0)`	

Solution: (same as last time) insert a pipeline bubble in the EX stage to force the `addi` to wait until `x1` has the new value.

5-Stage Pipelined Execution

<u>WITH</u> EX->EX & Mem->EX Forwarding

Cycle	IF	ID	EX	Mem	WB
1	`lw x1,0×10(x0)`				
2	`addi x2,x1,0×1`	`lw x1,0×10(x0)`			
3	`add x3,x2,x1`	`addi x2,x1,0×1`	`lw x1,0×10(x0)`		
4	`add x3,x2,x1`	`addi x2,x1,0×1`	1	`lw x1,0×10(x0)`	
5	`sw x3,0×14(x0)`	`add x3,x2,x1`	`addi x2,x1,0×1`	1	`lw x1,0×10(x0)`

Mem->EX forwarding!

At the trigger clock edge between cycle 4 and 5, the result of `lw` becomes available. The pipeline identifies that the following instruction – `addi` – depends on the result of `lw` via register `x1`, and forwards this value to `addi` as it enters the EX stage.

5-Stage Pipelined Execution

***WITH* EX->EX & Mem->EX Forwarding**

Cycle	IF	ID	EX	Mem	WB
4	add x3,x2,x1	addi x2,x1,0×1	1	lw x1,0×10(x0)	
5	sw x3,0×14(x0)	add x3,x2,x1	addi x2,x1,0×1	1	lw x1,0×10(x0)
6	add x0,x0,x0	sw x3,0×14(x0)	add x3,x2,x1	addi x2,x1,0×1	1

At the trigger clock edge between cycle 5 and 6, the result of `addi` becomes available. The pipeline identifies that the following instruction – `add` – depends on the result of `addi` via register `x2`, and forwards this value to `add` as it enters the EX stage.

EX->EX forwarding!

5-Stage Pipelined Execution

***WITH* EX->EX & Mem->EX Forwarding**

Cycle	IF	ID	EX	Mem	WB
6	`add x0,x0,x0`	`sw x3,0×14(x0)`	`add x3,x2,x1`	`addi x2,x1,0×1`	1
7	`add x0,x0,x0`	`add x0,x0,x0`	`sw x3,0×14(x0)`	`add x3,x2,x1`	`addi x2,x1,0×1`
8					

At the trigger clock edge between cycle 6 and 7, the result of `add` becomes available. The pipeline identifies that the following instruction – `sw` – depends on the result of `add` via register `x3`, and forwards this value to `sw` as it enters the EX stage.
Note that this forwarding still happens for `sw` – a memory instruction that does not execute the EX stage – the forwarded value will follow the control signals into Mem.

EX->EX forwarding!

5-Stage Pipelined Execution

<u>WITH</u> EX->EX & Mem->EX Forwarding

Cycle	IF	ID	EX	Mem	WB
1	`lw x1,0×10(x0)`				
2	`addi x2,x1,0×1`	`lw x1,0×10(x0)`			
3	`add x3,x2,x1`	`addi x2,x1,0×1`	`lw x1,0×10(x0)`		
4	`add x3,x2,x1`	`addi x2,x1,0×1`	1	`lw x1,0×10(x0)`	
5	`sw x3,0×14(x0)`	`add x3,x2,x1`	`addi x2,x1,0×1`	1	`lw x1,0×10(x0)`
6	`add x0,x0,x0`	`sw x3,0×14(x0)`	`add x3,x2,x1`	`addi x2,x1,0×1`	1
7	`add x0,x0,x0`	`add x0,x0,x0`	`sw x3,0×14(x0)`	`add x3,x2,x1`	`addi x2,x1,0×1`
8	`add x0,x0,x0`	`add x0,x0,x0`	`add x0,x0,x0`	`sw x3,0×14(x0)`	`add x3,x2,x1`

5-Stage Pipelined Execution

<u>WITH</u> EX->EX & Mem->EX Forwarding

Cycle	IF	ID	EX	Mem	WB
1	lw x1,0×10(x0)				
2	addi x2,x1,0×1	lw x1,0×10(x0)			
3	add x3,x2,x1	addi x2,x1,0×1	lw [illegible]		
4	add x3,x2,x1	addi x2,x1,0×1			
5	sw x3,0×14(x0)	add x3,x2,x1	add [illegible]	[illegible]	[illegible]×10(x0)
6	add x0,x0,x0	sw x3,0×14(x0)	add x3,x2,x1	addi x2,x1,0×1	1
7	add x0,x0,x0	add x0,x0,x0	sw x3,0×14(x0)	add x3,x2,x1	addi x2,x1,0×1
8	add x0,x0,x0	add x0,x0,x0	add x0,x0,x0	sw x3,0×14(x0)	add x3,x2,x1
9	add x0,x0,x0	add x0,x0,x0	add x0,x0,x0	add x0,x0,x0	add x0,x0,x0

With forwarding, the third instruction exits the pipeline after the 8th cycle.

5-Stage Pipelined Execution

<u>WITH</u> EX->EX & Mem->EX Forwarding

What's the CPI of the pipeline for the first three instructions?

Cycles = 8, Instructions = 3
=> CPI = 8/3 ~= 2.667

$$\mathrm{CPI} = \frac{\mathrm{Cycles}}{\mathrm{Instructions}}$$

What's the IPC of the pipeline for the first three instructions?

Instructions = 3, Cycles = 8
=> IPC = 3/8 = 0.375

$$\mathrm{IPC} = \frac{\mathrm{Instructions}}{\mathrm{Cycles}}$$

IPC is no way near 1 yet, but it's better than 0.273.

Iron Law of Computer Performance

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

From without forwarding to with forwarding, we've improved CPI:

$$\text{CPI Improvement} = \frac{8/3}{11/3} = \frac{8}{11}$$

Speedup again?

From without forwarding to with forwarding, we've improved CPI:

$$\text{CPI Improvement} = \frac{8/3}{11/3} = \frac{8}{11}$$

That is, the new CPU Time is 8/11 of the original CPU Time.
How much speedup did we just get?

$$S = \frac{t_{\text{base}}}{t_{\text{improved}}} = \frac{t_{\text{base}}}{\frac{8}{11} \times t_{\text{base}}} = \frac{11}{8} = 1.375$$

Pipeline hazards

We've already seen this:

Cycle	IF	ID	EX	Mem	WB
1	`lw x1,0×10(x0)`				
2	`addi x2,x1,0×1`	`lw x1,0×10(x0)`			
3	`add x3,x2,x1`	`addi x2,x1,0×1`	`lw x1,0×10(x0)`		
4			`addi x2,x1,0×1`	`lw x1,0×10(x0)`	

`addi` reads an incorrect value from `x1` in EX, then, `lw` writes to `x1`.

This is the Read-After-Write (RAW) hazard, the only possible hazard type in our simple 5-stage pipeline.

Pipeline hazards

: the order in which the accesses to register should happen

```
sub x6,  x5, x4
lw  x16, 0×abc
add x12, x6, x14
```

Read-After-Write (RAW)

Can happen in our pipeline.

```
sub x8, x16, x4
add x16, x6, x14
lw  x16, 0×abc
```

Write-After-Read (WAR)

Can happen in an (incorrectly implemented) out-of-order pipeline.

```
lw  x6, 0×abc
sub x6, x5, x4
add x12 x6, x14
```

Write-After-Write (WAR)

Can happen in an (incorrectly implemented) out-of-order pipeline.

Labs

Lab 0

Recitation 0 and 1 should have everything you need for finishing Lab 0.

https://course.ece.cmu.edu/~ece344/course_documents/18344-f25-recitation0.pdf

https://course.ece.cmu.edu/~ece344/course_documents/18344-f25-recitation1.pdf

Lab 1

- Starting from Lab 1, labs will be ***partnered*** (in most cases, group of 2)
- We ***recommend*** using proper version control tools
  - e.g. `git`; ***MAKE SURE YOUR REPO IS PRIVATE IF USING GITHUB OR ALTERNATIVES***
  - Not likely that we can be of much help if you nuked your work on the number machines
- Schedule regular meetings with your lab partner
  - This is more important than you'd think
- Watch out for an announcement on the "code freeze" policy
  - Expect a code deadline a few days before the report deadline