Announcements:

- Dan's office hours on Fri, Oct 18 cancelled.
- Don't discuss midterm until grades are released.
- No midterm grade ETA right now, but reminder that we have a clobber policy.

Lectures 21–22

# Pipelining

**CS 61C, Fall 2024 @ UC Berkeley**

Slides credit: Dan Garcia, Bora Nikolic, Peyrin Kao

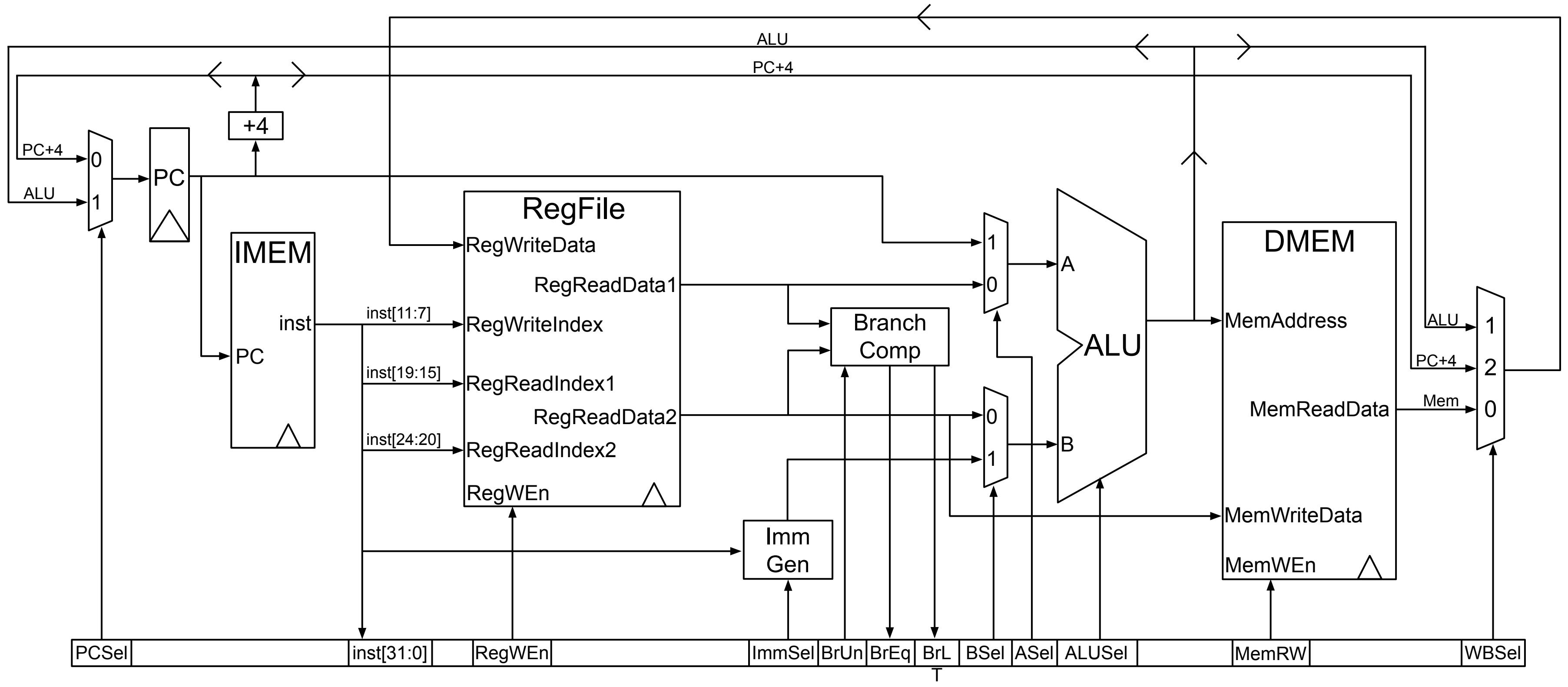
Slides template credit: Josh Hug, Lisa Yan

# Measuring Performance

Lectures 21–22, CS 61C, Fall 2024

## The Single-Cycle RISC-V Datapath

Behold: Our circuit for running RISC-V instructions. We built this in the last 4 lectures.



## Instruction Timing

What is the clock period of our datapath circuit?

- Determined by longest path (lw).
- Clock period: 800ps.
- Frequency: 1/800ps = 1.25 GHz.
- 1.25 billion instructions per second.

Can we improve our datapath performance?

- What does it mean to improve performance?
- Quicker response time = one job finishes faster?
- More jobs per unit time?
- Longer battery life?

| instr | Time taken per stage (in ps): | | | | | Total time taken: |
|---|---|---|---|---|---|---|
| | IF | ID | EX | MEM | WB | |
| add | 200 | 100 | 200 | | 100 | 600ps |
| beq | 200 | 100 | 200 | | | 500ps |
| jal | 200 | 100 | 200 | | 100 | 600ps |
| **lw** | **200** | **100** | **200** | **200** | **100** | **800ps** |
| sw | 200 | 100 | 200 | 200 | | 700ps |

This table shows some example timings.

Some cells are empty if the instruction doesn't use that stage.

## Analogy: Transportation

Consider two different vehicles:

| | Sports Car | Bus |
|---|---|---|
| **Passenger Capacity:** | 2 | 50 |
| **Travel Speed:** | 200 mph | 50 mph |
| **Gas Mileage:** | 5 mpg | 2 mpg |

For a 50 mile trip (assuming instant return trips)...

| | Sports Car | Bus |
|---|---|---|
| **Travel Time:** | 15 min | 60 min |
| **Time for 100 passengers:** | 750 min (50 trips) | 120 min (2 trips) |
| **Gallons per passenger:** | 5 gallons | 0.5 gallons |

## Measuring Performance in Computers

We'll look at 3 different measures of computer performance:

| **Transportation Analogy** | **In a Computer** |
|---|---|
| Trip Time | **Program execution time**<br>(Example: Time to update display) |
| Time for 100 passengers | **Throughput**<br>(Example: Number of server requests handled per hour) |
| Gallons per passenger | **Energy per task**<br>(Example: How many movies you can watch per battery charge) |

## Optimizing Time Per Program

The "Iron Law" of processor performance:

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \quad \frac{\text{Cycles}}{\text{Instruction}} \quad \frac{\text{Time}}{\text{Cycle}}$$

There are 3 components for optimizing the time it takes to execute a program.

## Optimizing Time Per Program (1/3)

The "Iron Law" of processor performance:

$$\frac{\text{Time}}{\text{Program}} = \boxed{\frac{\text{Instructions}}{\text{Program}}} \quad \frac{\text{Cycles}}{\text{Instruction}} \quad \frac{\text{Time}}{\text{Cycle}}$$

Instructions per program determined by:

- The task we're trying to do.
- The algorithm we implement, e.g. $\theta(N^2)$ or $\theta(N)$.
- The programming language we use.
- The compiler we use.
- The instruction set architecture (ISA) we use, e.g. x86 or RISC-V.

## Optimizing Time Per Program (2/3)

The "Iron Law" of processor performance:

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \boxed{\frac{\text{Cycles}}{\text{Instruction}}} \frac{\text{Time}}{\text{Cycle}}$$

Average clock cycles per instruction (CPI) determined by:

- The instruction set architecture (ISA).
- Processor implementation and microarchitecture, e.g. our datapath design.

Examples:

- For the datapath we made, CPI = 1.
- For complex instructions (e.g. `strcpy`), CPI > 1.
- We'll see CPI < 1 processors later.

## Optimizing Time Per Program (3/3)

The "Iron Law" of processor performance:

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \quad \frac{\text{Cycles}}{\text{Instruction}} \quad \boxed{\frac{\text{Time}}{\text{Cycle}}}$$

Time per cycle (1/frequency) determined by:

- Processor microarchitecture: Critical path through logic gates.
- Technology: How many transistors fit in a given space.
- Power budget: Lower voltages reduce transistor speed.

## Optimizing Time Per Program

The "Iron Law" of processor performance:

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \quad \frac{\text{Cycles}}{\text{Instruction}} \quad \frac{\text{Time}}{\text{Cycle}}$$

| | **Processor A** | **Processor B** |
|---|---|---|
| **# Instructions** | 1 million | 1.5 million |
| **Average CPI** | 2.5 | 1 |
| **Clock rate *f*** | 2.5 GHz | 2 GHz |
| **Execution time** | 1 ms | 0.75 ms |

Processor B is faster for this task, despite executing more instructions and having a slower clock rate!

## Optimizing Energy Per Program (1/2)

There are 2 components for optimizing the energy it takes to execute a program.

$$\frac{\text{Energy}}{\text{Program}} = \boxed{\frac{\text{Instructions}}{\text{Program}}} \frac{\text{Energy}}{\text{Instruction}}$$

We already saw instructions per program earlier – it appears in this equation, too.

## Optimizing Energy Per Program (2/2)

There are 2 components for optimizing the energy it takes to execute a program.

$$\frac{\text{Energy}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \boxed{\frac{\text{Energy}}{\text{Instruction}}}$$

Energy per instruction = $CV^2$

- $C$ = capacitance. Depends on hardware technology and processor features.
- $V$ = supply voltage.

To improve performance, we can reduce capacitance and voltage.

- Newer processors improve efficiency from Moore’s law and lower supply voltage.
- Reducing supply voltage too far causes “leakage power” - transistor switches don’t fully turn off.
- In recent years, it's gotten harder to reduce supply voltage.

## Optimizing Tasks Per Second

The energy "Iron Law":

$$\underbrace{\frac{\text{Tasks}}{\text{Second}}}_{\text{Performance}} = \underbrace{\frac{\text{Joules}}{\text{Second}}}_{\text{Power}} \quad \underbrace{\frac{\text{Tasks}}{\text{Joule}}}_{\text{Energy Efficiency}}$$

Energy efficiency is a key metric in all computing devices.

- If system is constrained by power, we need better energy efficiency to get more performance at the same power.
- For energy-constrained systems, we need better energy efficiency to prolong battery life.

# Pipelining Analogy: Laundry

Lectures 21–22, CS 61C, Fall 2024

## Pipelining Analogy: Laundry

Andrew, Bora, Caroline, Dan are doing laundry.

Each person needs to Wash, Dry, Fold, and Stash.

- **Wash** using the washer takes 30 mins.
- **Dry** using the dryer takes 30 mins.
- **Fold** using the table takes 30 mins.
- **Stash** using the drawers takes 30 mins.

## Laundry Timing Diagram

Example: From 2:00 to 2:30, Andrew is folding.

Time of day.

Order of people.

| | 1:00 | 1:30 | 2:00 | 2:30 | 3:00 | 3:30 |
|---|---|---|---|---|---|---|
| **Andrew** | Wash | Dry | Fold | Stash | | |
| **Bora** | | | | | | |
| **Caroline** | | | | | | |
| **Dan** | | | | | | |

## Sequential Laundry

Naive approach: Andrew does everything, then Bora does everything, and so on.

- How long for everybody to finish laundry? 8 hours.
- How long for one person to do their laundry? 2 hours.

| | **1:00** | **1:30** | **2:00** | **2:30** | **3:00** | **3:30** | **4:00** | **4:30** | **5:00** | **5:30** | **6:00** | **6:30** | **7:00** | **7:30** | **8:00** | **8:30** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Andrew** | Wash | Dry | Fold | Stash | | | | | | | | | | | | |
| **Bora** | | | | | Wash | Dry | Fold | Stash | | | | | | | | |
| **Caroline** | | | | | | | | | Wash | Dry | Fold | Stash | | | | |
| **Dan** | | | | | | | | | | | | | Wash | Dry | Fold | Stash |

## Sequential Laundry

Can we do better than this?

Idea: When Andrew's drying, nobody is using the washer.

| | **1:00** | **1:30** | **2:00** | **2:30** | **3:00** | **3:30** | **4:00** | **4:30** | **5:00** | **5:30** | **6:00** | **6:30** | **7:00** | **7:30** | **8:00** | **8:30** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Andrew** | Wash | Dry | Fold | Stash | | | | | | | | | | | | |
| **Bora** | | | | | Wash | Dry | Fold | Stash | | | | | | | | |
| **Caroline** | | | | | | | | | Wash | Dry | Fold | Stash | | | | |
| **Dan** | | | | | | | | | | | | | Wash | Dry | Fold | Stash |

## Pipelined Laundry

Pipelined approach: While Andrew uses the Dryer, Bora can use the Washer.

- How long for everybody to finish laundry? 3.5 hours.
- How long for one person to do their laundry? 2 hours. (Same as before!)

| | **1:00** | **1:30** | **2:00** | **2:30** | **3:00** | **3:30** | **4:00** |
|---|---|---|---|---|---|---|---|
| **Andrew** | Wash | Dry | Fold | Stash | | | |
| **Bora** | | Wash | Dry | Fold | Stash | | |
| **Caroline** | | | Wash | Dry | Fold | Stash | |
| **Dan** | | | | Wash | Dry | Fold | Stash |

## Pipelined Laundry

At any given moment in time, multiple people are in the system.

- Different people are using different resources at the same time.
- Example: At 2:30, Dan is using the Washer, Caroline is using the Dryer, and so on.

| | **1:00** | **1:30** | **2:00** | **2:30** | **3:00** | **3:30** | **4:00** |
|---|---|---|---|---|---|---|---|
| **Andrew** | Wash | Dry | Fold | Stash | | | |
| **Bora** | | Wash | Dry | Fold | Stash | | |
| **Caroline** | | | Wash | Dry | Fold | Stash | |
| **Dan** | | | | Wash | Dry | Fold | Stash |

## Pipelined Laundry

Over time, one person uses each of the resources, in order.

| | **1:00** | **1:30** | **2:00** | **2:30** | **3:00** | **3:30** | **4:00** |
|---|---|---|---|---|---|---|---|
| **Andrew** | Wash | Dry | Fold | Stash | | | |
| **Bora** | | Wash | Dry | Fold | Stash | | |
| **Caroline** | | | Wash | Dry | Fold | Stash | |
| **Dan** | | | | Wash | Dry | Fold | Stash |

## Filling and Draining the Pipeline

The ideal speedup for everyone to finish is 4x, since 4 things are happening at once.

Limited by:

- Startup costs.
  "Filling" the pipeline.
- Shutdown costs.
  "Draining" the pipeline.
- At the start and end, some resources are unused.

| | 1:00 | 1:30 | 2:00 | 2:30 | 3:00 | 3:30 | 4:00 | 4:30 | 5:00 | 5:30 | 6:00 | 6:30 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Andrew | W | D | F | S | | | | | | | | |
| Bora | | W | D | F | S | | | | | | | |
| Caroline | | | W | D | F | S | | | | | | |
| Dan | | | | W | D | F | S | | | | | |
| Eddy | | | | | W | D | F | S | | | | |
| Jero | | | | | | W | D | F | S | | | |
| Myrah | | | | | | | W | D | F | S | | |
| Rosalie | | | | | | | | W | D | F | S | |
| Peyrin | | | | | | | | | W | D | F | S |

## Pipelines Are Limited By Slowest Stage

What if Dry only took 20 mins, instead of 30?

- Bora finishes Dry at 2:20.
- But Andrew finishes Fold at 2:30. Bora still has to wait.

Pipeline rate is limited by the slowest pipeline stage.

| | **1:00** | **1:30** | **2:00** | **2:30** | **3:00** | **3:30** | **4:00** |
|---|---|---|---|---|---|---|---|
| **Andrew** | Wash | Dry | Fold | Stash | | | |
| **Bora** | | Wash | Dry | Fold | Stash | | |
| **Caroline** | | | Wash | Dry | Fold | Stash | |
| **Dan** | | | | Wash | Dry | Fold | Stash |

## Pipelining Laundry: Observations

**Latency** (time it takes to finish a single task) is unchanged.

- It still took 2 hours for each person to do laundry.

**Throughput** (number of jobs finished per hour) increased.

- Sequential: 4 people took 8 hours = 0.5 people per hour.
- Pipelined: 4 people took 3.5 hours = 1.14 people per hour.

Maximum throughput speedup = number of stages.

- If 4 people were doing laundry at any given time, we'd have 4x speedup.
- Limited by cost of filling and draining pipeline at the start and end.

Pipeline rate is limited by the slowest pipeline stage.

- Must wait for the slowest stage to finish before moving on to the next stage.
- Balancing the length of each pipeline stage is good for speedup.

# Pipelining RISC-V Instructions

Lectures 21–22, CS 61C, Fall 2024

## Pipeline Timing Diagrams

Example: In the third time step, the `add` instruction is in the Execute (EX) stage of the pipeline.

Numbers represent time steps.

Sequence of instructions.

| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
|---|---|---|---|---|---|---|---|---|---|---|
| `add s0 t0 t1` | IF | ID | EX | M | WB | | | | | |
| `sub t2 s0 t0` | | | | | | IF | ID | EX | M | WB |

## Sequential Instructions

Sequential approach: One instruction finishes, then the next one starts.

- How long to run 2 instructions? 10 time steps.
- How long to run each instruction? 5 time steps.

Our clock period would need to be 5 time steps.

| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
|---|---|---|---|---|---|---|---|---|---|---|
| `add s0 t0 t1` | IF | ID | EX | M | WB | | | | | |
| `sub t2 s0 t0` | | | | | | IF | ID | EX | M | WB |

## Pipelined Instructions

Pipelined approach: Multiple instructions in the circuit at the same time.

- How long to run 2 instructions? 6 time steps.
- How long to run each instruction? 5 time steps. (Same as before!)

Our clock period is now 1 time step!

| | **1** | **2** | **3** | **4** | **5** | **6** |
|---|---|---|---|---|---|---|
| `add s0 t0 t1` | IF | ID | EX | M | WB | |
| `sub t2 s0 t0` | | IF | ID | EX | M | WB |

## Pipelined Instructions

At any given moment in time, multiple instructions are in the circuit.

- Different instructions are using different resources at the same time.
- Example: At time step 3, the `add` instruction is using the Execute (EX) stage, while the `sub` instruction is using the Decode (ID) stage.

| | **1** | **2** | **3** | **4** | **5** | **6** |
|---|---|---|---|---|---|---|
| `add s0 t0 t1` | IF | ID | EX | M | WB | |
| `sub t2 s0 t0` | | IF | ID | EX | M | WB |

## Pipelined Instructions

Over time, each instruction passes through all 5 pipeline stages.

- 1 stage per clock cycle.

Note: The pipelined CPU uses one clock for all stages.

- Clock cycle time is limited by the slowest stage (just like in the analogy).

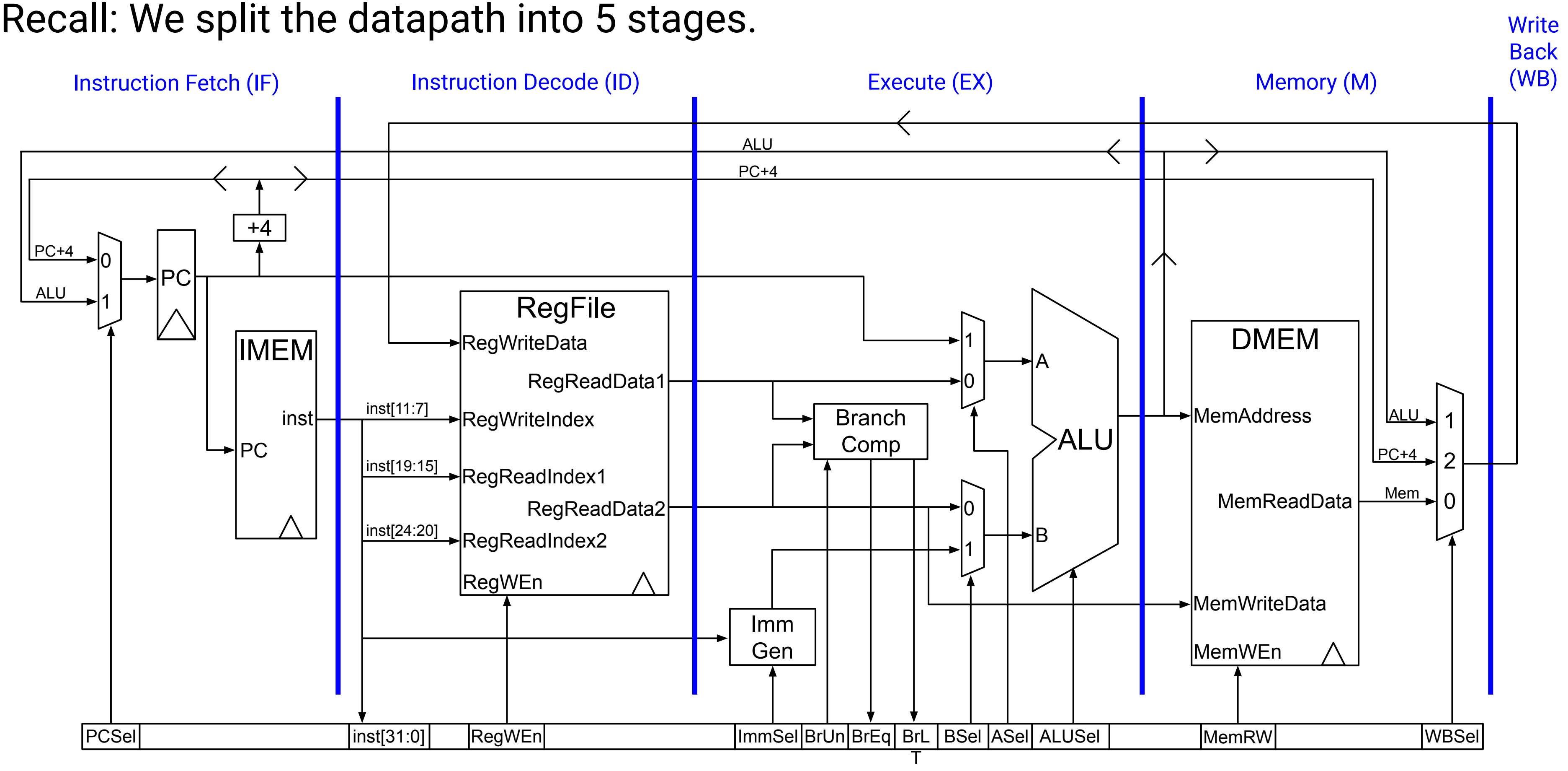
| | **1** | **2** | **3** | **4** | **5** | **6** |
|---|---|---|---|---|---|---|
| `add s0 t0 t1` | IF | ID | EX | M | WB | |
| `sub t2 s0 t0` | | IF | ID | EX | M | WB |

# Pipelined Datapath

Lectures 21–22, CS 61C, Fall 2024

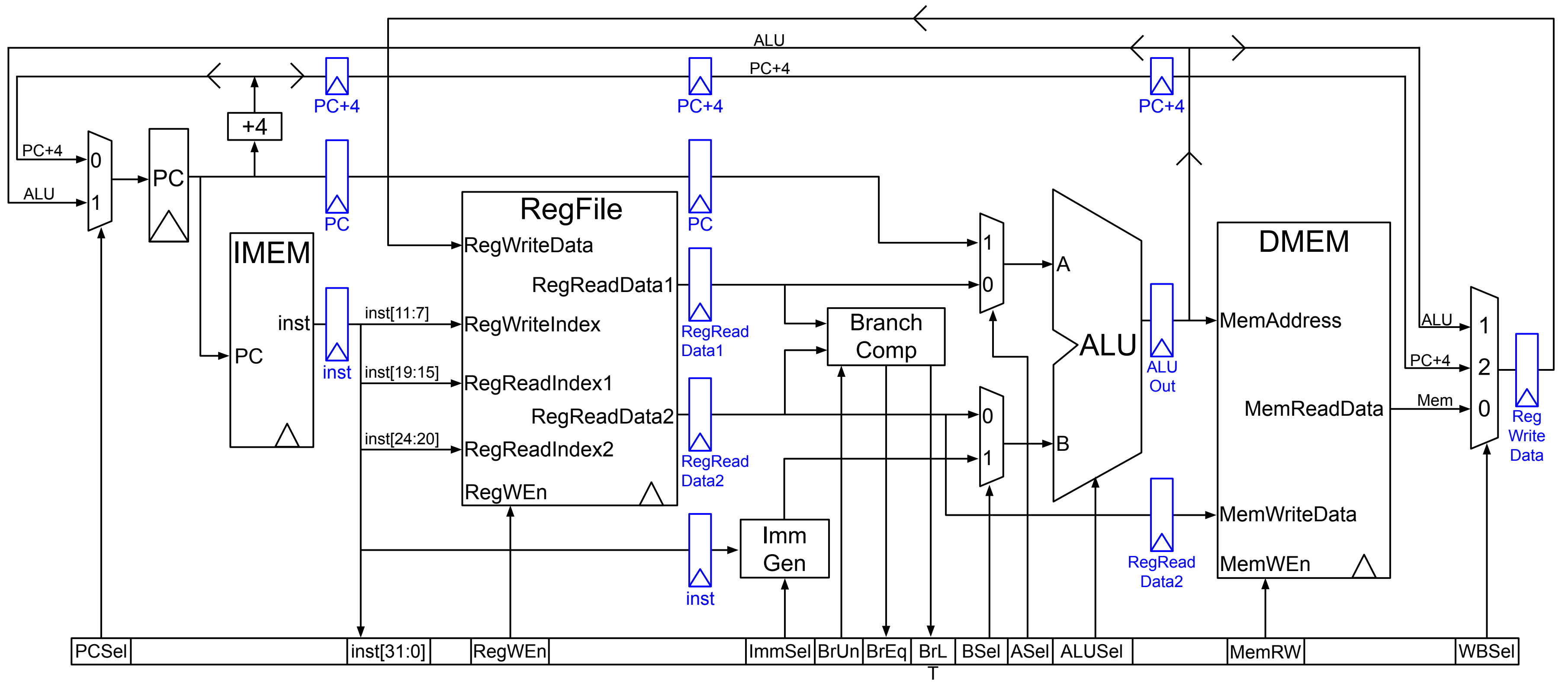
## Pipelined Datapath

Recall: We split the datapath into 5 stages.



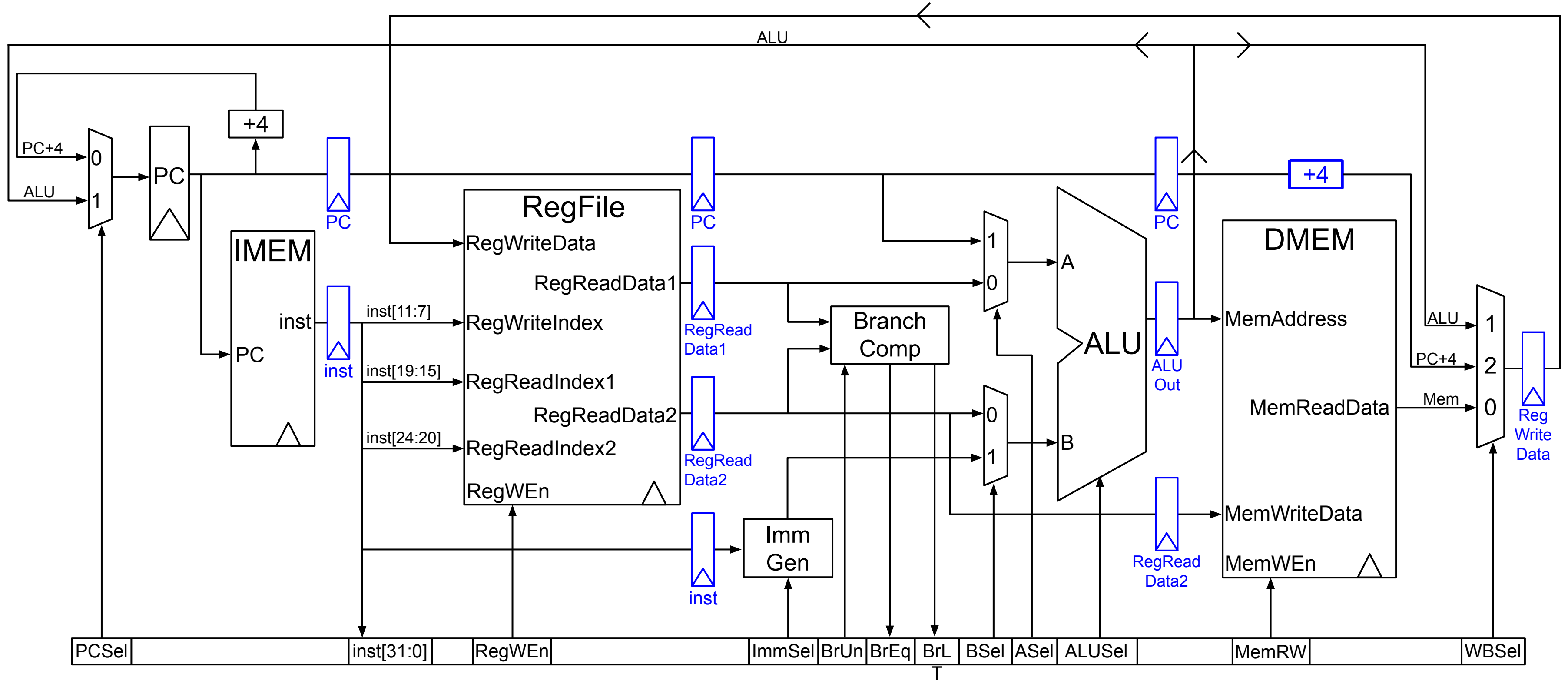
## Pipelined Datapath

Add registers between each stage to "hold" a signal until the next clock cycle.



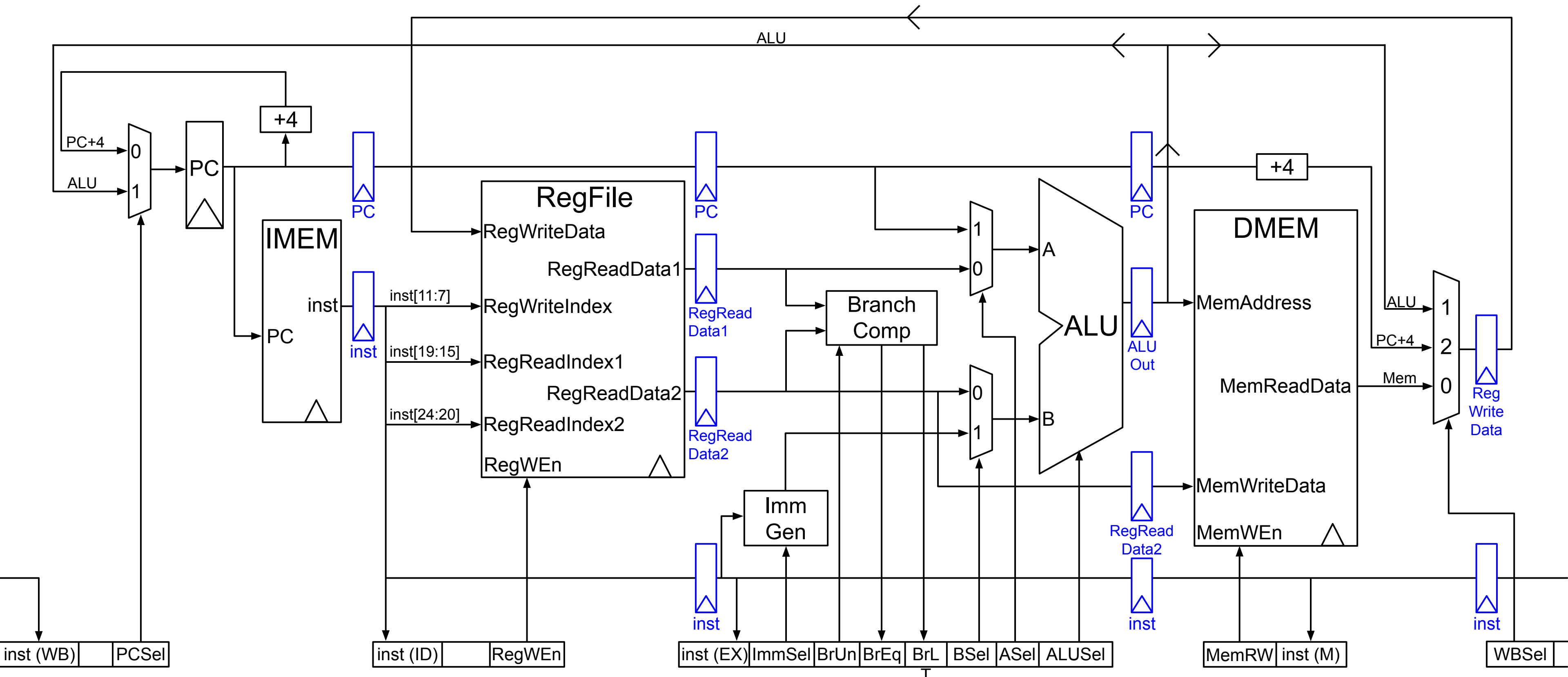
## Pipelined Datapath

Optimization: Recalculate PC+4 from PC in Memory stage to avoid storing/sending both PC and PC+4 down the pipeline.



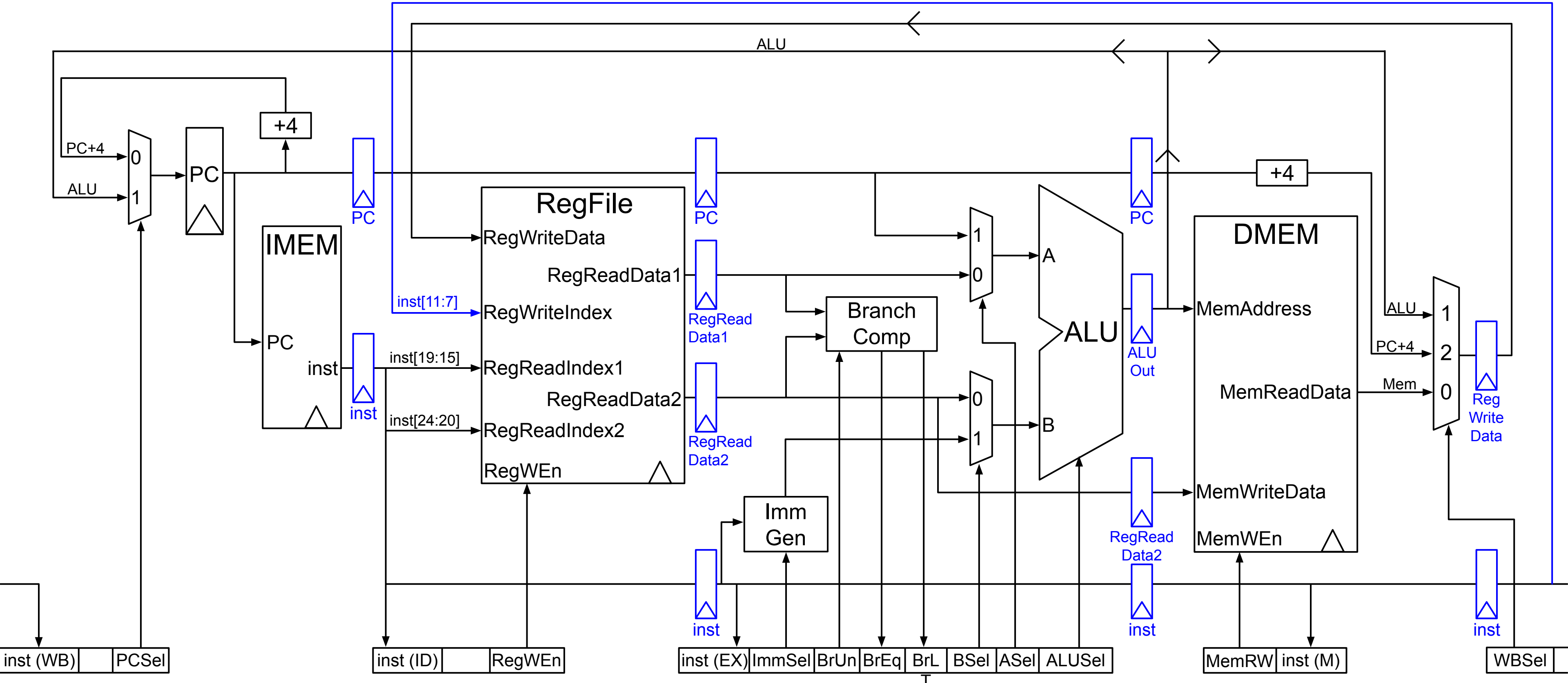
## Pipelined Datapath

Send instructions down the pipeline so control logic operates correctly in each stage.



## Pipelined Datapath

Change RegWriteIndex to use the rd field from the WB stage instruction, not the ID stage instruction.



## Pipelining Control Logic

Approach #1: One control logic subcircuit per stage.

- Store and send the instruction through the pipeline stages.
- Calculate relevant signals in each stage.
- Seen on the previous slides.

Approach #2: One control logic subcircuit.

- Calculate all the control logic signals in the decode (ID) stage.
- Send the control logic signals through the pipeline stages.

## Structural Hazards

Lectures 21–22, CS 61C, Fall 2024

## Structural Hazards

Problem: Two or more instructions in the pipeline both need to access a single physical resource. In other words, a physical resource is needed in multiple stages.

- Analogy example: What if the "Fold" and "Stash" tasks both needed the table?

Solution 1: Instructions take turns to use the resource.

- Takes additional cycles per instruction.
- One instruction must *stall* (wait) while the other uses the resource.

Solution 2: Add more hardware to machine.

- Structural hazards can always be solved by adding more hardware.

Solution 3: Design the instruction set architecture (ISA) to avoid structural hazards.

- RISC-V is designed like this.
- Example: RegFile hardware can read two values per cycle. No RISC-V instruction needs to read 3 or more registers at once.

## Structural Hazard Example: Regfile

What are the physical limits of the Regfile?

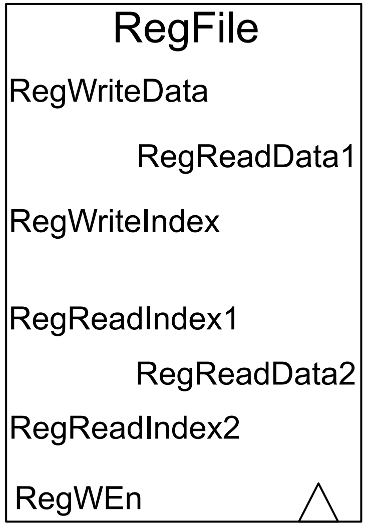
- It can read from 2 registers in a single cycle.
- It can write to 1 register in a single cycle.

We avoid structural hazards by having separate "ports."

- 2 independent read ports, and 1 independent write port.
- Allows 3 simultaneous accesses per cycle.

Hypothetical structural hazards:

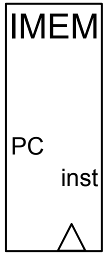
- What if the Regfile only had 1 read port?
- What if a RISC-V instruction needed to write to 2 registers?
- Good design helped us avoid structural hazards!



## Structural Hazard Example: Memory Access

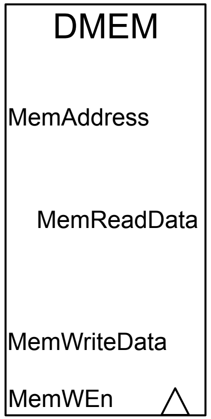
The datapath has two memory blocks:

- IMEM: Instruction Memory.
- DMEM: Data Memory.
- But they're really the same memory.



In a pipelined datapath, IMEM and DMEM are used at the same time.
This is a structural hazard!

- Solution 1: IF stage *stalls* a cycle while MEM uses memory.
- Solution 2: Use two separate memories.
- Solution 3: Use caches to make IMEM and DMEM appear like separate memories.
    - This is what RISC-V does.
    - More on caches later!



## Data Hazards

Lectures 21–22, CS 61C, Fall 2024

## Data Hazards

When does add **write** to register s0? t = 5.

When does sub **read** the value in s0? t = 3.

Is this a problem?

Reminder: Regfile lives in the ID stage, so that's why the read happens in the ID stage.

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| `add s0 t0 t1` | IF | ID | EX | M | **WB** | |
| `sub t2 s0 t0` | | IF | **ID** | EX | M | WB |

## Data Hazards

Problem: An instruction depends on the result from the previous instruction.

- The previous instruction hasn't finished executing yet...
- ...so the current instruction doesn't know the result to use!

Example: When sub reads s0, add has not updated s0 yet.

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| `add s0 t0 t1` | IF | ID | EX | M | **WB** | |
| `sub t2 s0 t0` | | IF | **ID** | EX | M | WB |

## Data Hazard Example (1/2)

Does this code have a data hazard?

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| `add t0 t1 t2` | IF | ID | EX | M | WB | | |
| `or t3 t4 t5` | | IF | ID | EX | M | WB | |
| `slt t6 t0 t3` | | | IF | ID | EX | M | WB |

## Data Hazard Example (1/2)

To identify data hazards: Check if we **write** to a register and then **read** from it later.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| `add t0 t1 t2` | IF | ID | EX | M | WB | | |
| `or t3 t4 t5` | | IF | ID | EX | M | WB | |
| `slt t6 t0 t3` | | | IF | ID | EX | M | WB |

## Data Hazard Example (1/2)

To identify data hazards: Check if we write to a register and then read from it later.

- add **writes** to t0 at t = 5.
- slt **reads** from t0 at t = 4. This is too early to get the updated value!

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| `add t0 t1 t2` | IF | ID | EX | M | **WB** | | |
| `or t3 t4 t5` | | IF | ID | EX | M | WB | |
| `slt t6 t0 t3` | | | IF | **ID** | EX | M | WB |

## Data Hazard Example (2/2)

Does this code have a data hazard?

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| `add t0 t1 t2` | IF | ID | EX | M | WB | | | | |
| `or t3 t4 t5` | | IF | ID | EX | M | WB | | | |
| `slt t6 t2 t3` | | | IF | ID | EX | M | WB | | |
| `lw t3 4(t0)` | | | | IF | ID | EX | M | WB | |
| `add t0 t1 t2` | | | | | IF | ID | EX | M | WB |

## Data Hazard Example (2/2)

To identify data hazards: Check if we **write** to a register and then **read** from it later.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| `add t0 t1 t2` | IF | ID | EX | M | WB | | | | |
| `or t3 t4 t5` | | IF | ID | EX | M | WB | | | |
| `slt t6 t2 t3` | | | IF | ID | EX | M | WB | | |
| `lw t3 4(t0)` | | | | IF | ID | EX | M | WB | |
| `add t0 t1 t2` | | | | | IF | ID | EX | M | WB |

## Data Hazard Example (2/2)

To identify data hazards: Check if we write to a register and then read from it later.

- add **writes** to t0 at t = 5.
- lw **reads** from t0 at t = 5. Is that too early?

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| `add t0 t1 t2` | IF | ID | EX | M | **WB** | | | | |
| `or t3 t4 t5` | | IF | ID | EX | M | WB | | | |
| `slt t6 t2 t3` | | | IF | ID | EX | M | WB | | |
| `lw t3 4(t0)` | | | | IF | **ID** | EX | M | WB | |
| `add t0 t1 t2` | | | | | IF | ID | EX | M | WB |

## Data Hazard Example (2/2)

Some regfiles support writing a new value to a register, and then reading the new value, in the same cycle.

Not all regfiles support this. Check assumptions in any question.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| `add t0 t1 t2` | IF | ID | EX | M | **WB** | | | | |
| `or t3 t4 t5` | | IF | ID | EX | M | WB | | | |
| `slt t6 t2 t3` | | | IF | ID | EX | M | WB | | |
| `lw t3 4(t0)` | | | | IF | **ID** | EX | M | WB | |
| `add t0 t1 t2` | | | | | IF | ID | EX | M | WB |

# Fixing Data Hazards: Stalling

Lectures 21–22, CS 61C, Fall 2024

## Fixing Data Hazards

Solution 1: **Stalling**.

- Wait for the first instruction to write its result before the second instruction reads the value.

Solution 2: **Forwarding**.

- Add hardware to directly send the result back to the earlier stage, before the result is even written.

Solution 3: **Code Scheduling**.

- Rearrange instructions to avoid data hazards.

## Stalling

Without stalling, we have a data hazard:

- add **writes** to t0 at t = 5.
- slt **reads** from t0 at t = 4 (too early to get the updated value).

We need slt to wait 2 time steps before reading t0.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| `add t0 t1 t2` | IF | ID | EX | M | **WB** | | |
| `or t3 t4 t5` | | IF | ID | EX | M | WB | |
| `slt t6 t0 t3` | | | IF | **ID** | EX | M | WB |

## Stalling

Now, the read happens after the write. No more hazard!

- add **writes** to t0 at t = 5.
- slt **reads** from t0 at t = 6. The correct updated value is read!

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| `add t0 t1 t2` | IF | ID | EX | M | **WB** | | | | |
| `nop` | | IF | ID | EX | M | WB | | | |
| `nop` | | | IF | ID | EX | M | WB | | |
| `or t3 t4 t5` | | | | IF | ID | EX | M | WB | |
| `slt t6 t0 t3` | | | | | IF | **ID** | EX | M | WB |

## Stalling

To stall, we add no-ops into the code to delay an instruction.

- Recall: A no-op is an instruction that does nothing.
    - Example: `addi x0 x0 0`.
    - nop is a pseudo-instruction you can type in RISC-V.
- Compiler can insert no-ops to avoid data hazards.
    - Requires the compiler to know about the pipeline structure.

Stalls reduce performance, but they're required to get correct results!

# Fixing Data Hazards: Forwarding

Lectures 21–22, CS 61C, Fall 2024

## Forwarding Example (1/2)

Problem: add writes t0 at t = 5, which is too late.

Idea: When do we first learn the result of the add computation?

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| `add t0 t1 t2` | IF | ID | EX | M | **WB** | |
| `slt t6 t0 t3` | | IF | **ID** | EX | M | WB |

## Forwarding Example (1/2)

Problem: add writes t0 at t = 5, which is too late.

Idea: When do we first learn the result of the add computation?

- At t = 3, we already know the value that will be written to t0.

We can add hardware to *forward* this value so that slt can use it directly.

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| `add t0 t1 t2` | IF | ID | EX | M | WB | |
| `slt t6 t0 t3` | | IF | ID | EX | M | WB |

## Forwarding

Idea: Use the result as soon as it's computed.

- Read the value directly from a wire in the datapath.
- Don't wait for the result to be written to a register (and then read back).

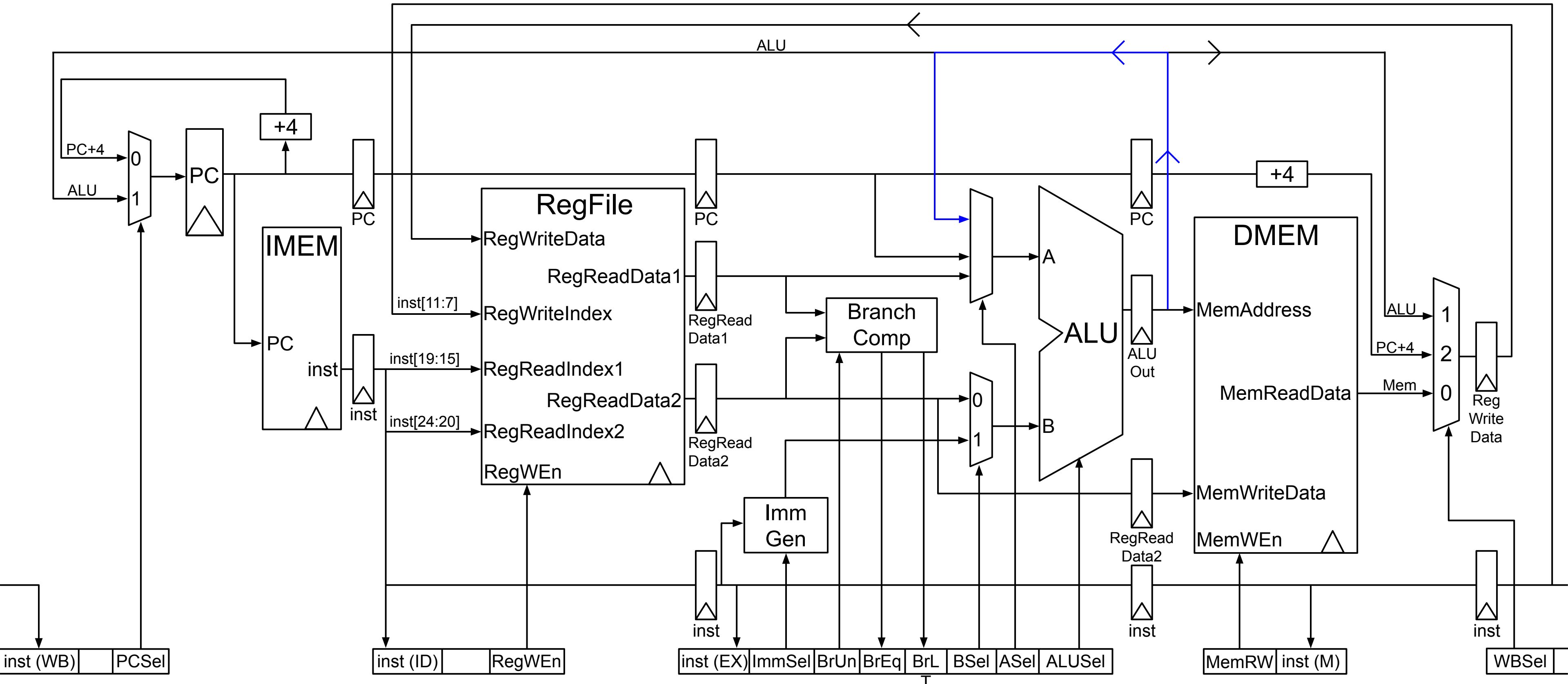
Forwarding requires extra connections in the datapath.

- We need extra wires to forward values to earlier stages.
- We need extra logic to check if forwarding is needed.
  - Where is the current instruction writing to? (i.e. what is rd?)
  - Where is the next instruction reading from? (i.e. what is rs1/rs2?)
  - If they're the same register*, we need to forward.

*Be careful to ignore writes to x0. If the current instruction is writing to x0,
there's no data hazard, because writes to x0 do nothing.

## Adding Forwarding to Datapath (1/2)

This wire forwards the ALU result, so the next instruction can use result as ALU input.



## Forwarding Example (2/2)

Problem: lw writes to s2 at t = 5, which is too late.

Idea: When do we first learn the result of the load?

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| `lw s2 20(s1)` | IF | ID | EX | M | **WB** | |
| `add s4 s2 s5` | | IF | **ID** | EX | M | WB |

## Forwarding Example (2/2)

Problem: lw writes to s2 at t = 5, which is too late.

Idea: When do we first learn the result of the load?

- The value from memory shows up at t = 4.
- That's too late to send the value into the ALU!

We have to stall for one cycle.

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| `lw s2 20(s1)` | IF | ID | EX | **M** | WB | |
| `add s4 s2 s5` | | IF | ID | **EX** | M | WB |

## Forwarding Example (2/2)

Problem: lw writes to s2 at t = 5, which is too late.

Idea: When do we first learn the result of the load?

- The value from memory shows up at t = 4.
- That's too late to send the value into the ALU!

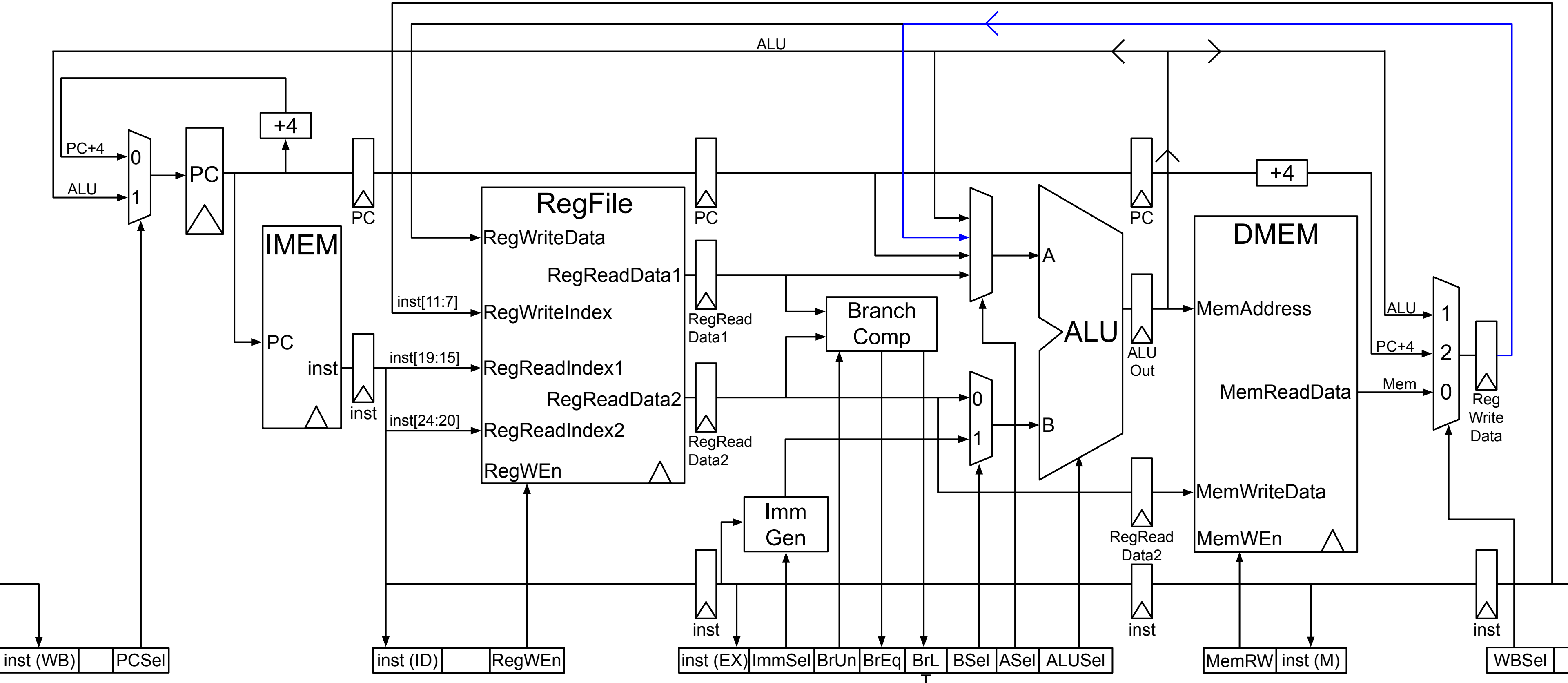
We have to stall for one cycle.

- Now, we can add hardware to forward the value so add can use it directly.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| `lw s2 20(s1)` | IF | ID | EX | **M** | WB | | |
| `nop` | | IF | ID | EX | M | WB | |
| `add s4 s2 s5` | | | IF | ID | **EX** | M | WB |

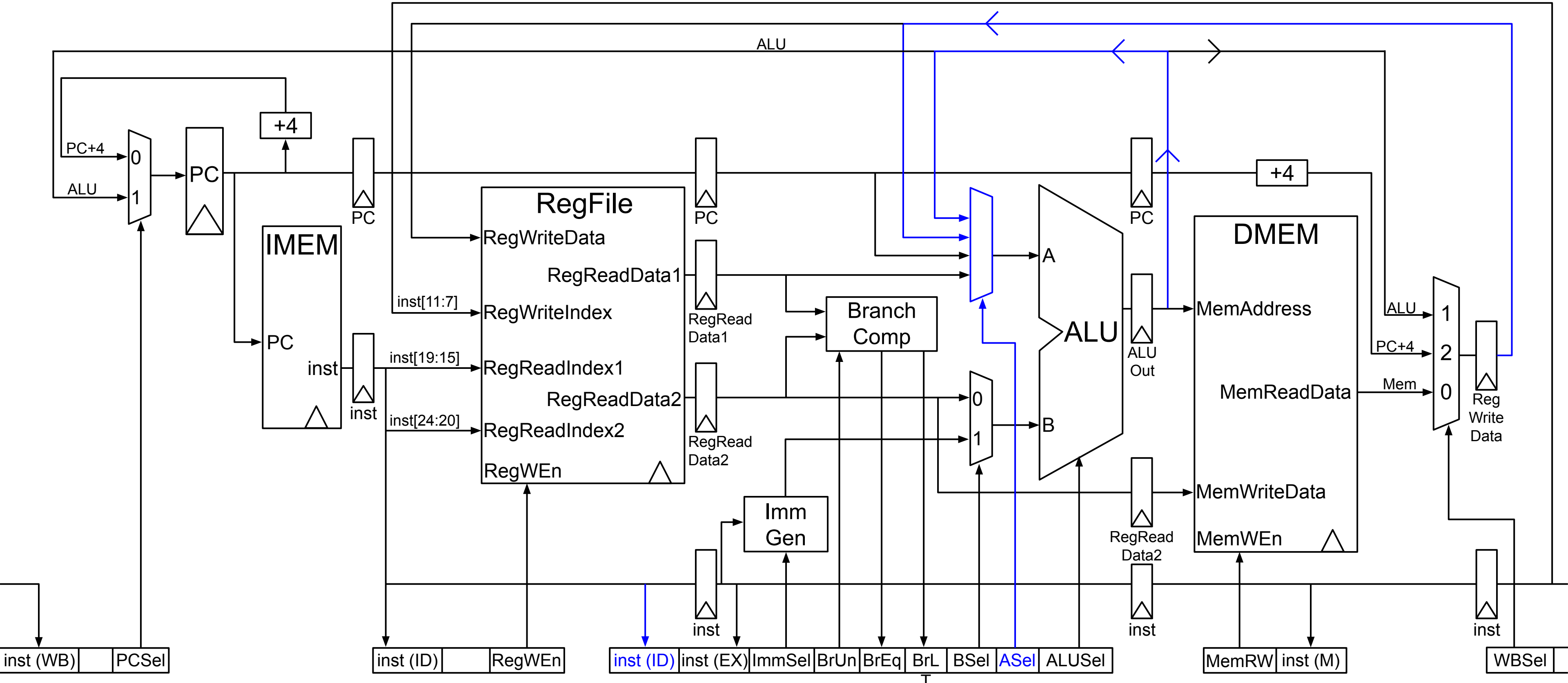
## Adding Forwarding to Datapath (2/2)

This wire forwards the value read from memory, so the next instruction can use the value as ALU input.



## Adding Forwarding Control Logic to Datapath

We need extra control logic to check whether we need to apply forwarding.



# Fixing Data Hazards: Code Scheduling

Lectures 21–22, CS 61C, Fall 2024

## Code Scheduling

Stalling inserts no-ops, which causes performance loss.

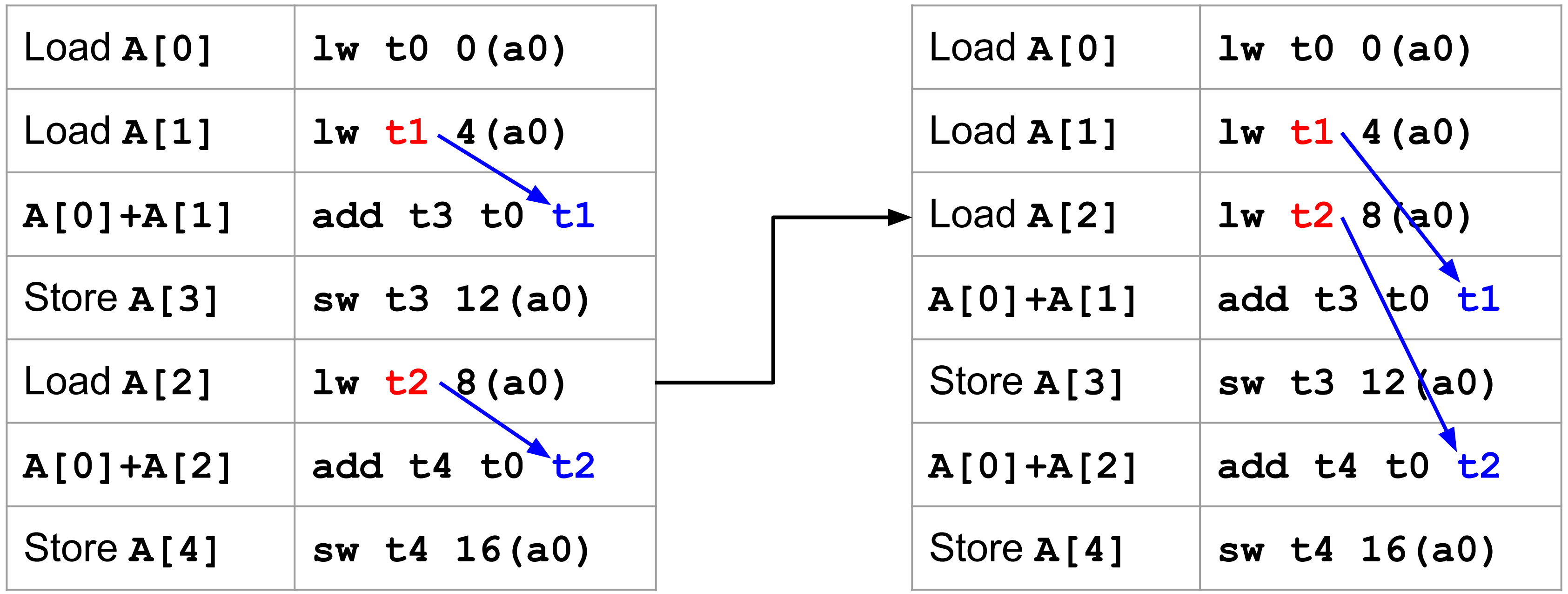
Idea: The compiler could use that no-op slot to execute an unrelated instruction.

- In other words, we *reorder* code to avoid data hazards.
- Requires the compiler to know about the pipeline structure.

## Code Scheduling Example

Consider this C code:

```
A[3] = A[0] + A[1];
A[4] = A[0] + A[2];
```



| | |
|---|---|
| Load `A[0]` | `lw t0 0(a0)` |
| Load `A[1]` | `lw t1 4(a0)` |
| `A[0]+A[1]` | `add t3 t0 t1` |
| Store `A[3]` | `sw t3 12(a0)` |
| Load `A[2]` | `lw t2 8(a0)` |
| `A[0]+A[2]` | `add t4 t0 t2` |
| Store `A[4]` | `sw t4 16(a0)` |

Original instruction order: 2 data hazards.

| | |
|---|---|
| Load `A[0]` | `lw t0 0(a0)` |
| Load `A[1]` | `lw t1 4(a0)` |
| Load `A[2]` | `lw t2 8(a0)` |
| `A[0]+A[1]` | `add t3 t0 t1` |
| Store `A[3]` | `sw t3 12(a0)` |
| `A[0]+A[2]` | `add t4 t0 t2` |
| Store `A[4]` | `sw t4 16(a0)` |

After reordering the "Load A[2]" instruction: 0 data hazards.

# Control Hazards

Lectures 21–22, CS 61C, Fall 2024

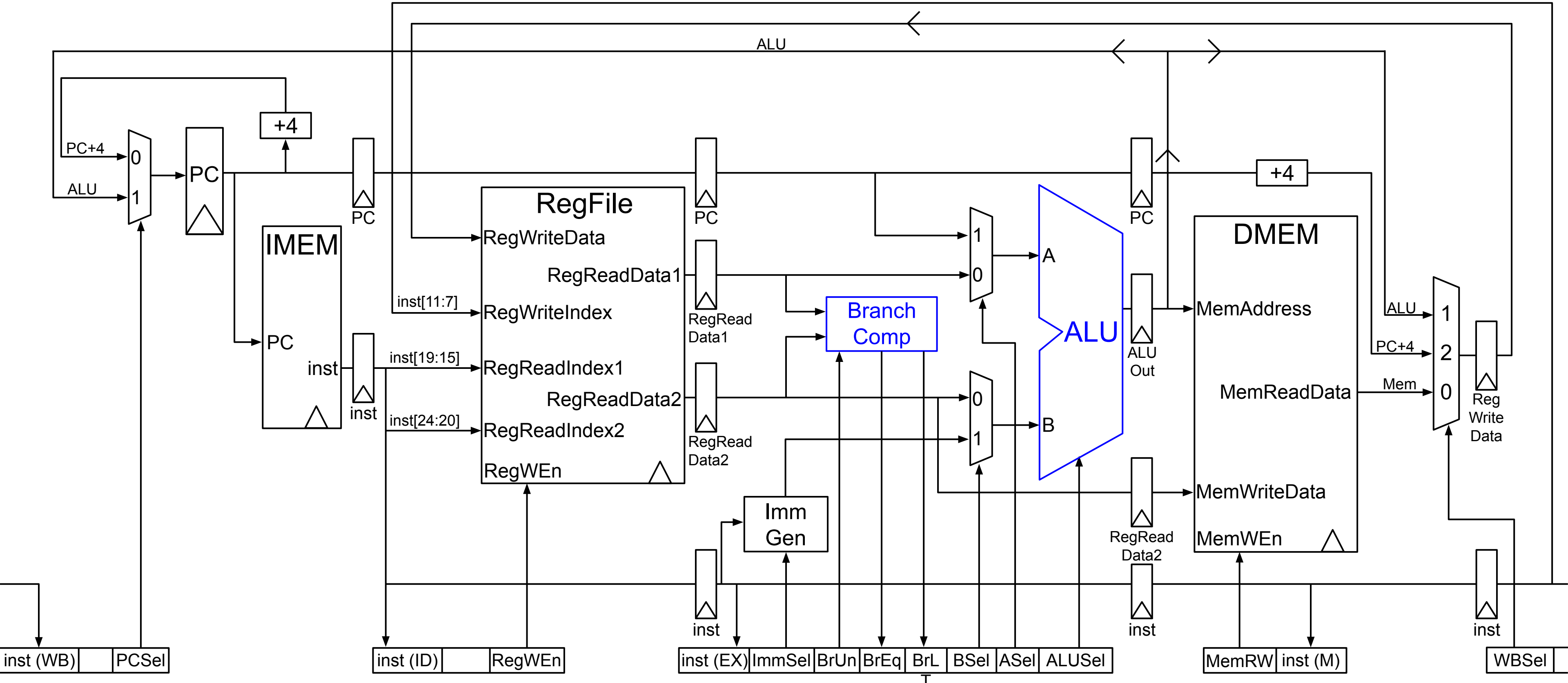
## Control Hazards

If the branch is taken (t0 == t1), when do we figure out where to branch to?

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| `beq t0 t1 Label` | IF | ID | EX | M | WB | | | | |
| `sub t2 s0 t0` | | IF | ID | EX | M | WB | | | |
| `add t6 s0 t3` | | | IF | ID | EX | M | WB | | |
| `xor t5 t1 s0` | | | | IF | ID | EX | M | WB | |
| `sw s0 8(t3)` | | | | | IF | ID | EX | M | WB |

## Control Hazards

Recall: The EX stage is where we compute where to branch to.



## Control Hazards

If the branch is taken (t0 == t1), when do we figure out where to branch to?

- At t = 3, the branch instruction runs the EX stage, which tells us where to go.
- But at t = 3, two more unwanted instructions have entered the pipeline!

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| `beq t0 t1 Label` | IF | ID | **EX** | M | WB | | | | |
| `sub t2 s0 t0` | | IF | **ID** | EX | M | WB | | | |
| `add t6 s0 t3` | | | **IF** | ID | EX | M | WB | | |
| `xor t5 t1 s0` | | | | IF | ID | EX | M | WB | |
| `sw s0 8(t3)` | | | | | IF | ID | EX | M | WB |

## Control Hazards

If the branch is taken, we're going somewhere else.

- We do not want to execute sub and add.
- We need to convert the next two instructions after the branch to no-ops.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| `beq t0 t1 Label` | IF | ID | **EX** | M | WB | | | | |
| ~~`sub t2 s0 t0`~~ | | ~~IF~~ | ~~ID~~ | ~~EX~~ | ~~M~~ | ~~WB~~ | | | |
| ~~`add t6 s0 t3`~~ | | | ~~IF~~ | ~~ID~~ | ~~EX~~ | ~~M~~ | ~~WB~~ | | |
| `xor t5 t1 s0` | | | | IF | ID | EX | M | WB | |
| `sw s0 8(t3)` | | | | | IF | ID | EX | M | WB |

## Control Hazards

If a branch is not taken:

- The instructions fetched after the branch are correct.
- There's no control hazard.

If the branch is taken, or if there's a jump:

- The next two instructions in the pipeline are incorrect.
- There's a control hazard!
- To fix, we need to convert the incorrect instructions in the pipeline to no-ops.
- This fix is called *flushing* the pipeline.

## Branch Prediction

Problem: Every time we take a branch or jump, we lose 3 cycles to no-ops.

Solution: Branch prediction.

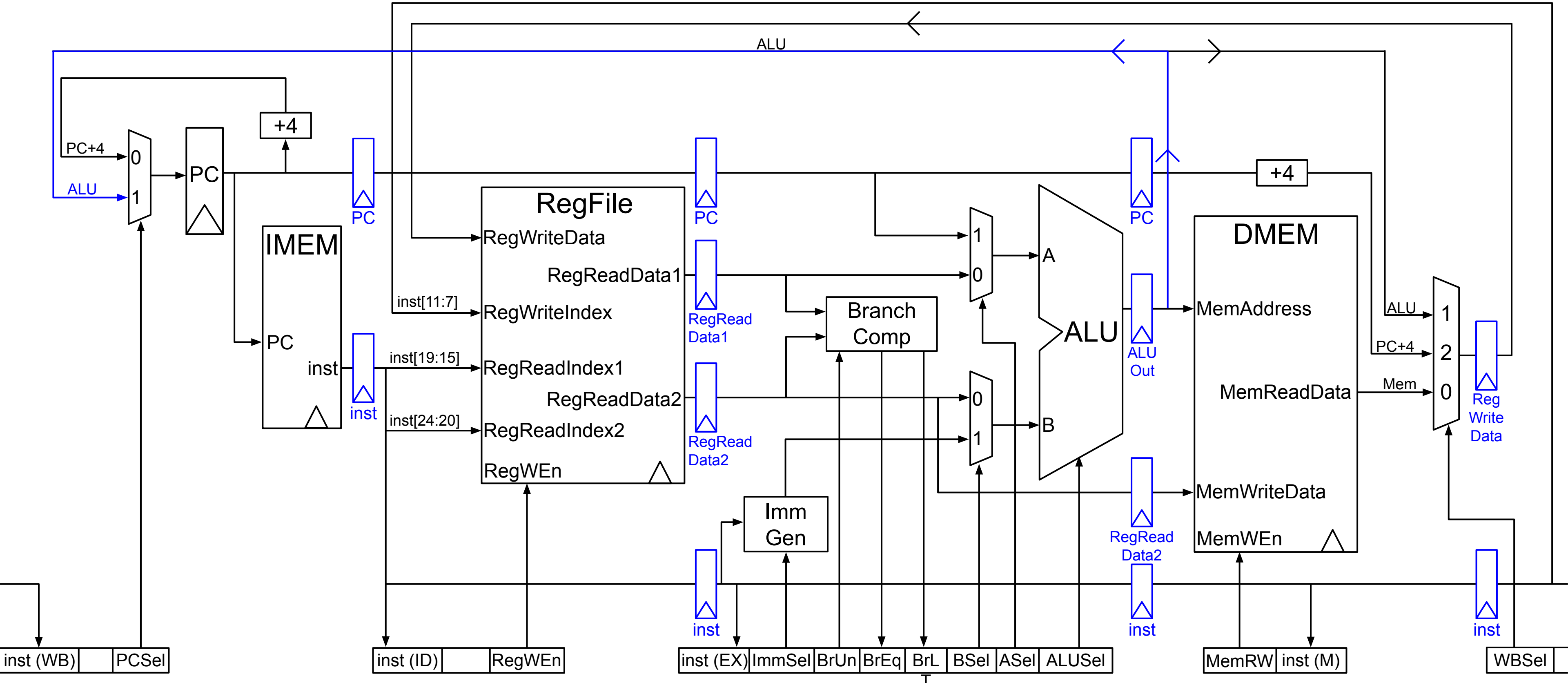
- Try and guess whether the branch will be taken.
- Only flush the pipeline if we guess wrong.
- Branch prediction improves performance *on average*, but there's still a penalty for wrong guesses.

What does our datapath do?

- Naive prediction: Our datapath always guesses "not taken" and fetches the next instructions.

## Pipelined Datapath

Note: We lose 3 cycles on a branch/jump, because updated PC is written back in MEM stage. While this happens, 3 instructions are in pipeline and need to be flushed.



## Superscalar Processors

Lectures 21–22, CS 61C, Fall 2024

## Pipelining and ISA Design

The RISC-V ISA is designed for pipelining.

- All instructions are 32 bits wide.
  - Easy to fetch in one clock cycle, and decode in one clock cycle.
  - Contrast with CISC (R = Reduced, C = Complex) ISAs like x86: Instructions can be 1–15 bytes wide!
- There's a small set of standard instruction formats.
  - Can decode instruction and read from registers in one stage.
- Load/store addressing is conceptually simple.
  - In Stage 3 (EX), use the ALU to compute the address.
  - In Stage 4 (MEM), access memory.
- Memory operands are all aligned.
  - Memory access only takes one cycle.

The 5-stage pipeline is commonplace in many devices: Cars, appliances, etc.

## Further Increasing Processor Performance?

1. Increase clock rate.

- Limited by technology and power dissipation.

2. Increase pipeline depth.

- "Overlap" instruction execution through deeper pipeline, e.g. 10 or 15 stages.
- Less work per stage means shorter clock cycles and lower power.
- But more potential for all 3 types of hazards!
- More hazards means more stalling. Cycles per instruction > 1.

3. Design a *superscalar* processor.

- Desktops, laptops, cell phones, etc. often have a few of these, combined with simpler 5-stage pipeline processors.

## Superscalar Processors

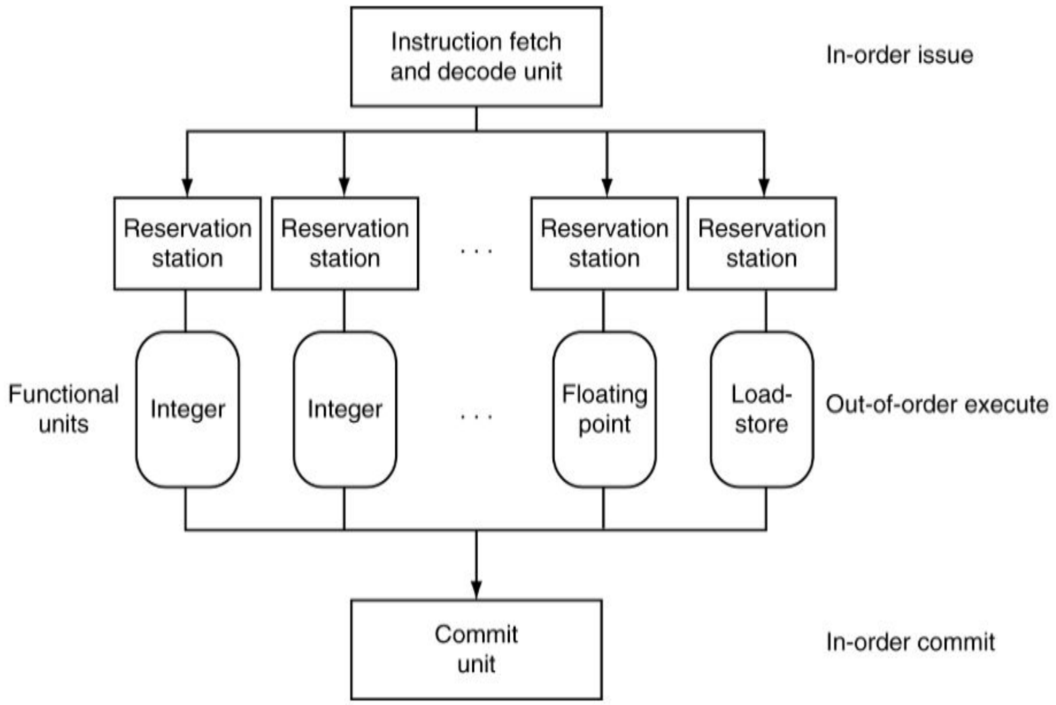
*Multiple-issue*: Start multiple instructions per clock cycle.

- Multiple execution units execute instructions in parallel.
- Each execution has its own pipeline.
- CPI < 1: Multiple instructions completed per clock cycle.

*Dynamic* "out-of-order" execution:

- Reorder instructions dynamically in hardware to reduce impact of hazards.

Take CS 152 to learn more!



## Computing Cycles Per Instruction (CPI)

How do we measure CPI?

- In practice, we measure CPI on various benchmarks.
- There are benchmark programs from a variety of application domains:
  Data compression, code compilation, video decoding, network simulation, etc.

Recall the "Iron Law" of processor performance:

$$\underbrace{\frac{\text{Time}}{\text{Program}}}_{\text{Can measure.}} = \underbrace{\frac{\text{Instructions}}{\text{Program}}}_{\text{Can count.}} \boxed{\frac{\text{Cycles}}{\text{Instruction}}} \underbrace{\frac{\text{Time}}{\text{Cycle}}}_{\text{1/clock rate.}}$$

Rearrange terms to solve for CPI:

$$\boxed{\frac{\text{Cycles}}{\text{Instruction}}} = \frac{\text{Time}}{\text{Program}} \div \left( \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Time}}{\text{Cycle}} \right)$$

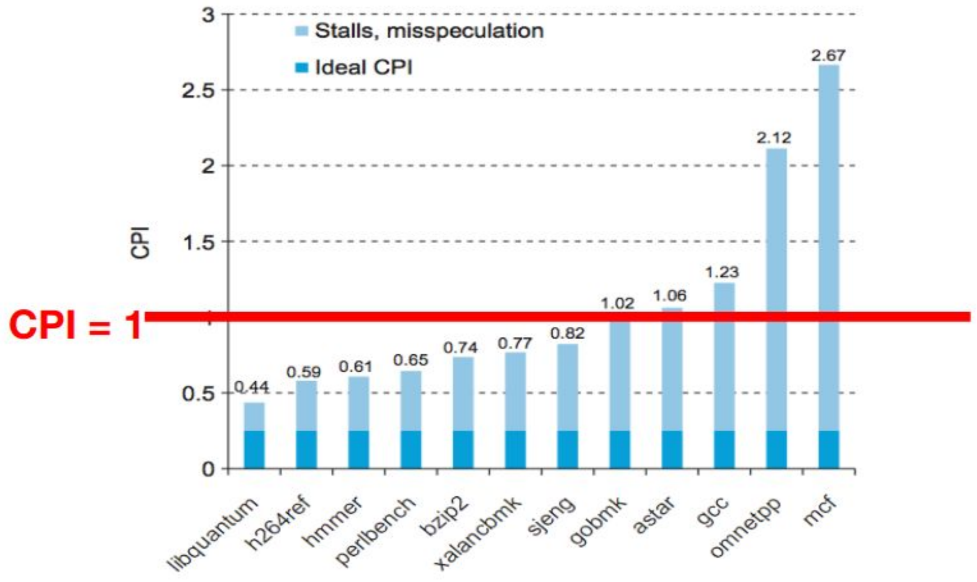
## ARM A53 Benchmark

ARM Cortex-A53 Core: 2 GHz, dual-issue processor:

- 4 BIPS (billion instructions per second).
- Peak CPI = 0.5.
- However, instruction/pipeline dependencies reduce performance.

If CPI < 1, we can also measure in terms of IPC (Instructions Per Cycle).

More about parallelism and performance later in CS 61C!



CPI of Intel Core i7 920 running SPEC2006 integer benchmarks.