

Processor Pipelining

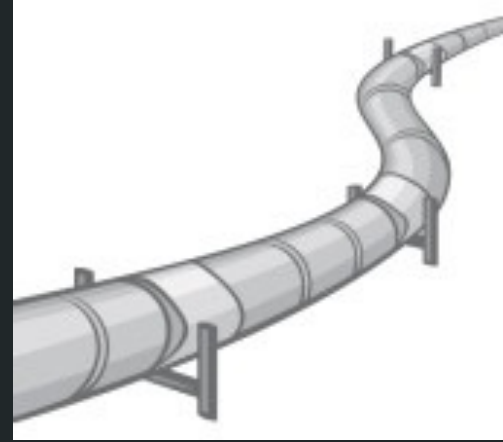
CS 1541

Wonsun Ahn



University of
Pittsburgh

Pipelining Basics



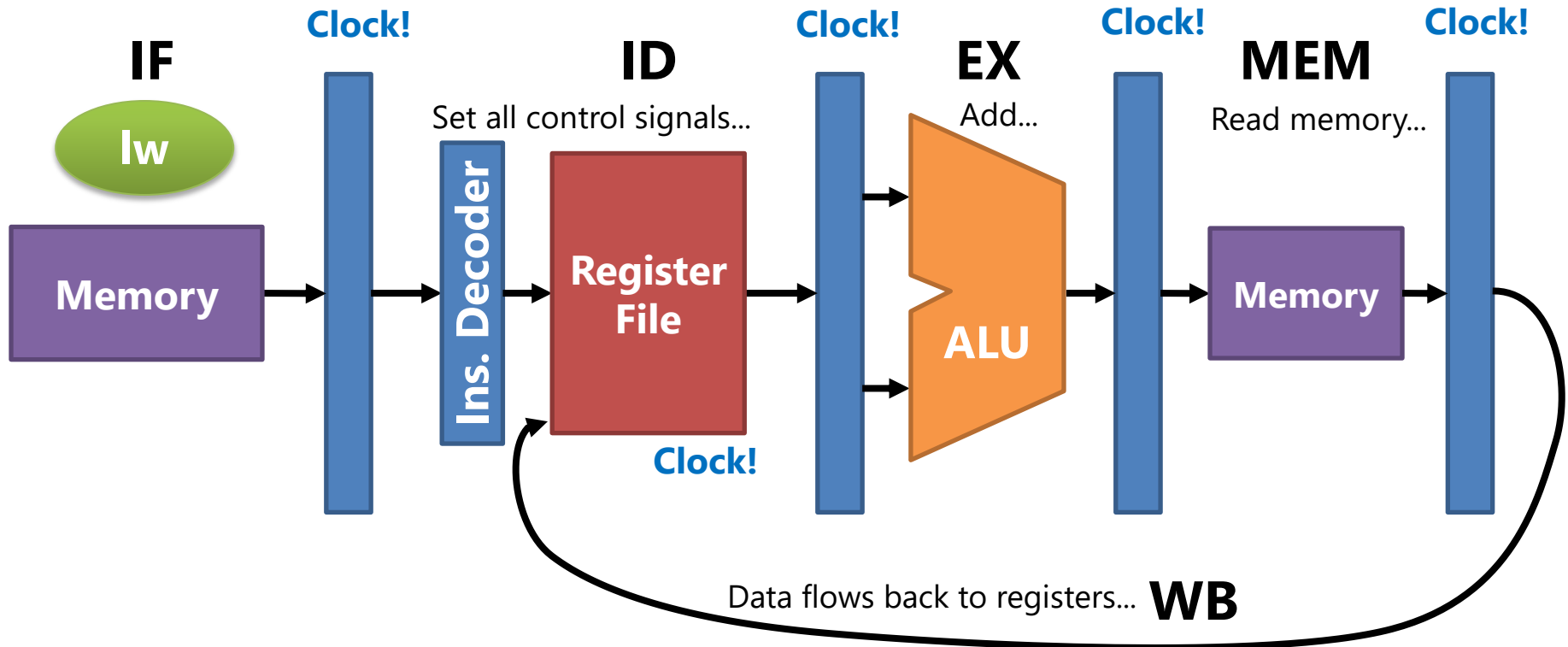
Improving Washer / Dryer / Closet Utilization

- If you work on loads of laundry one by one, you only get ~33% utilization
- If you form an "assembly line", you achieve ~100% utilization!



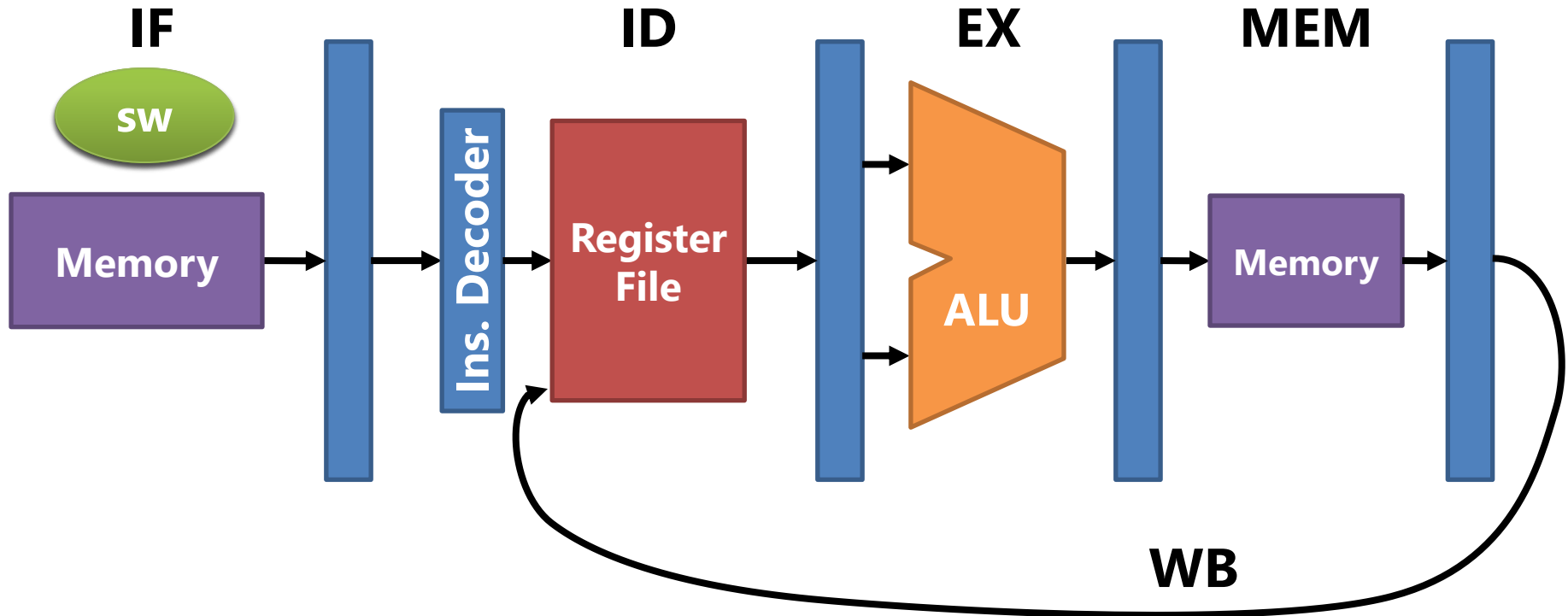
Multi-cycle instruction execution

- Let's watch how an instruction flows through the datapath.



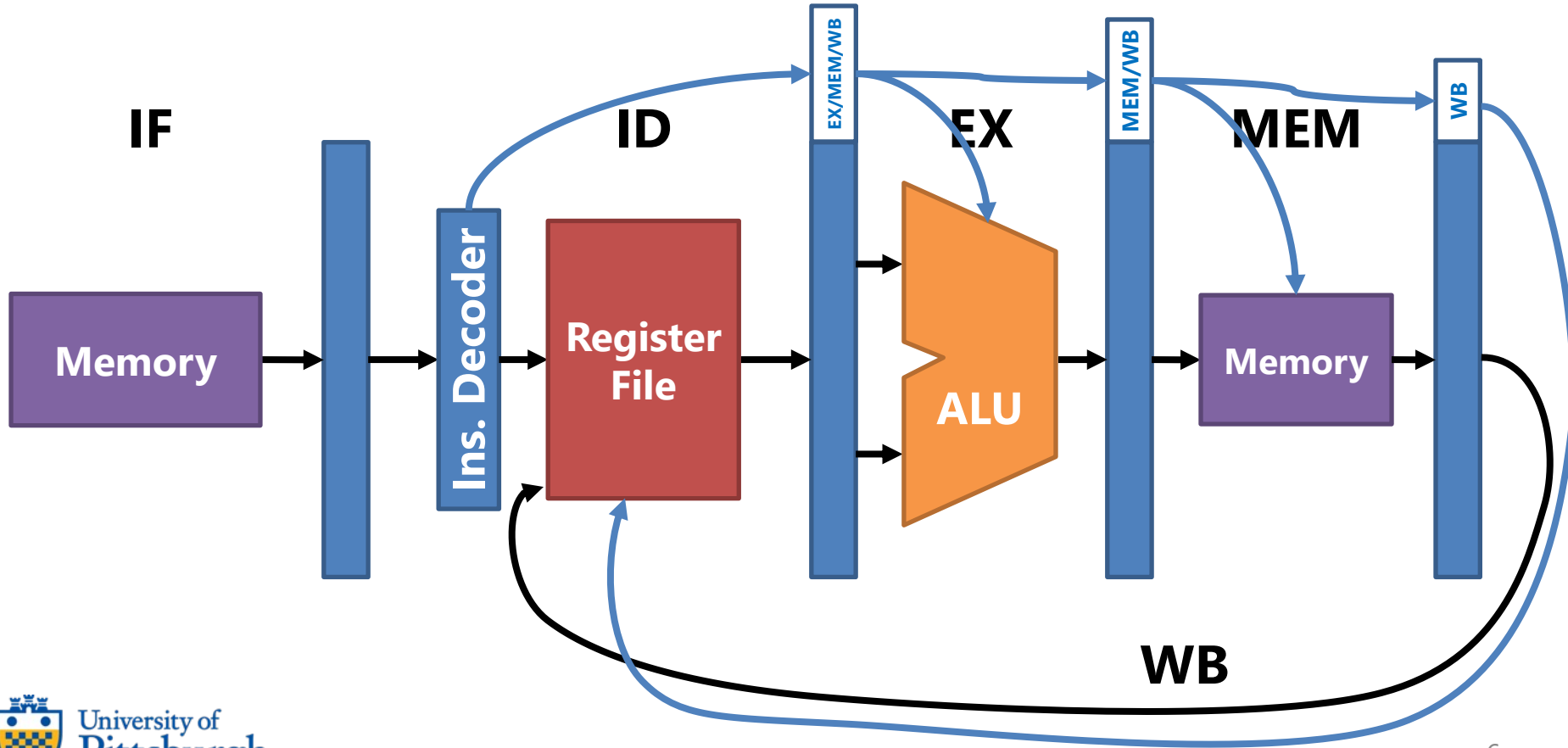
Pipelined instruction execution

- Pipelining allows one instruction to be fetched each cycle!



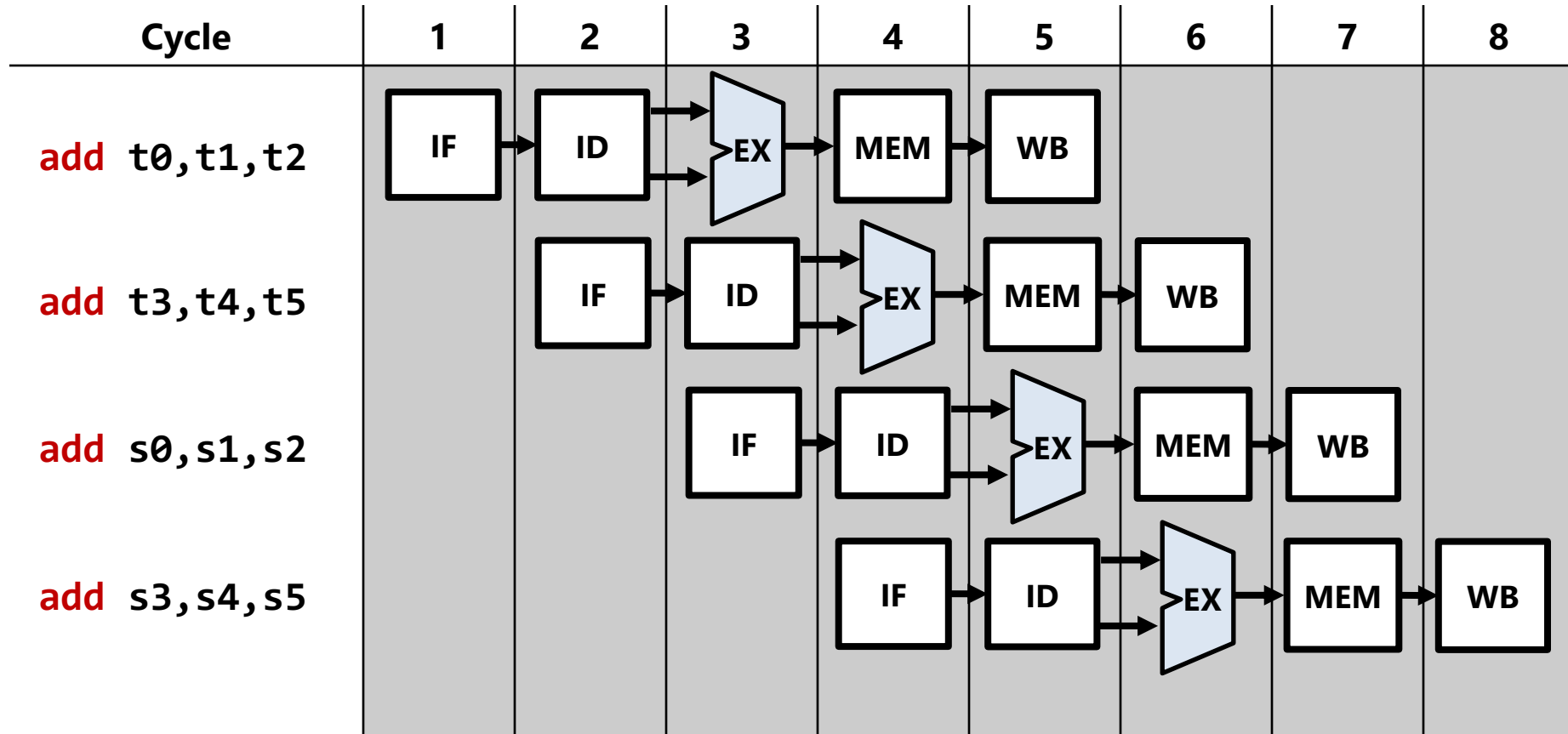
How about the control signals?

- A new instruction is decoded at every cycle!
- Control signals must be passed along with the data at each stage



Pipelining Timeline

- This type of parallelism is called *pipelined parallelism*.



A Pipelined Implementation is even Faster!

- Now every instruction takes the same number of cycles
 - **lw** takes 5 cycles: IF/ID/EX/MEM/WB
 - **add** takes 5 cycles: IF/ID/EX/---/WB
 - **sw** takes 5 cycles: IF/ID/EX/MEM/---
- If each stage takes 1 ns each:

Q) Given 100 instructions, the average instruction execution time is?

- A) $(5\text{ ns} + 99\text{ ns}) / 100 = 1.04\text{ ns}$
- A ~**5X** speed up from single cycle!

Pipelined vs. Multi-cycle vs. Single-cycle

- What happened to the three components of performance?

$$\frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}$$

Architecture	Instructions	CPI	Cycle Time (1/F)
Single-cycle	Same	1	5 ns
Multi-cycle	Same	4~5	1 ns
Pipelined	Same	1	1 ns

- Compared to single-cycle, pipelining improves clock cycle time
 - Or in other words CPU **clock frequency**
 - The deeper the pipeline, the higher the frequency will be

** Caveat: latch delay and unbalanced stages can increase cycle time*

Pipeline Hazards

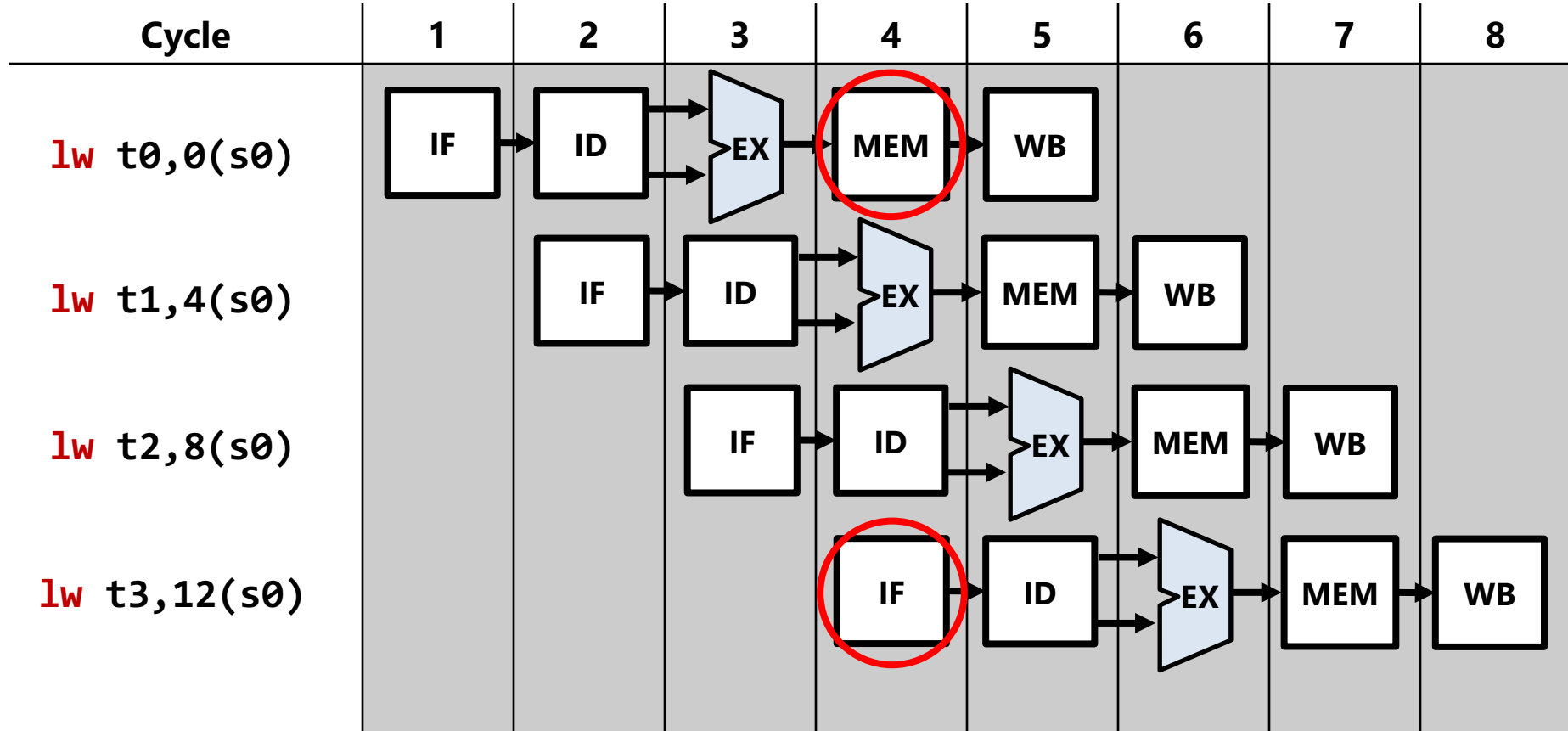


Pipeline Hazards

- For pipelined CPUs, we said CPI is practically 1
 - But that depends entirely on having the pipeline filled
 - In real life, there are **hazards** that prevent 100% utilization
- **Pipeline Hazard**
 - When the next instruction cannot execute in the following cycle
 - Hazards introduce **bubbles** (delays) into the pipeline timeline
- Architects have some tricks up their sleeves to avoid hazards
- But first let's briefly talk about the three types of hazards:
Structural hazard, Data hazard, Control Hazard

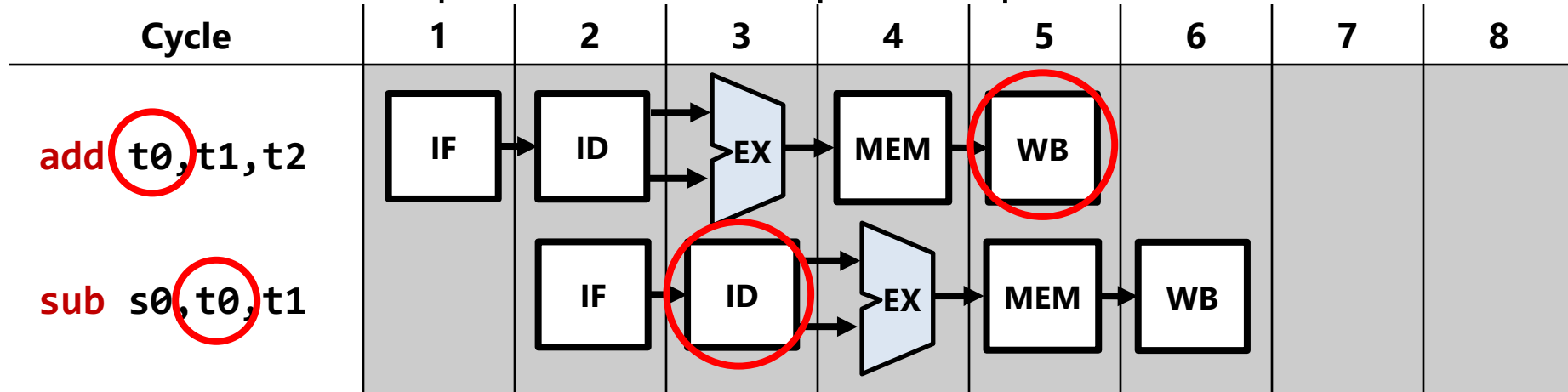
Structural Hazards

- Two instructions need to use the same hardware at the same time.



Data Hazards

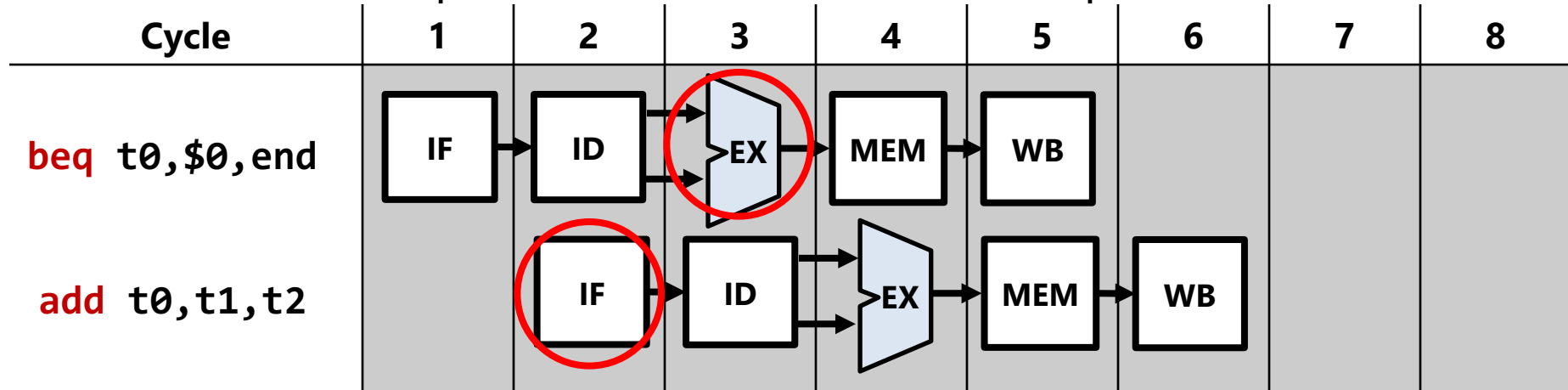
- An instruction depends on the output of a previous one.



- sub** reads in **t0** before **add** has had a chance to write it back!

Control Hazards

- An instruction depends on branch outcome of previous instruction.



- add** (PC+4) is fetched before **beq** branch outcome is known!

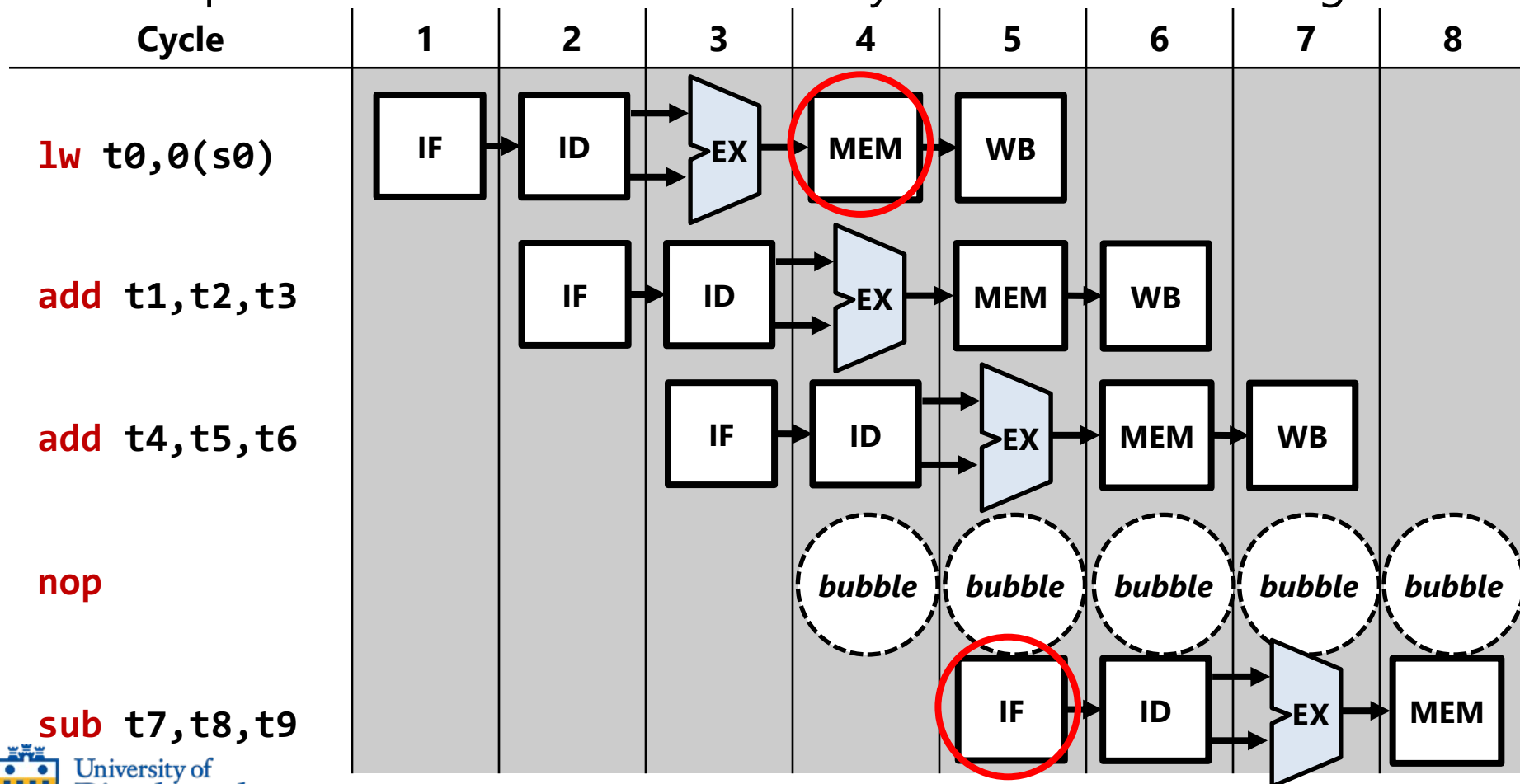
Dealing with Hazards

- Pipeline must be controlled so that hazards don't cause malfunction
- Who is in charge of that? You have a choice.
 1. Compiler can avoid hazards by inserting nops
 - Insert nops where compiler thinks a hazard would happen
 - Nops flow through the pipeline not doing any work
 2. CPU can internally avoid hazards using a ***hazard detection unit***
 - If structural/data hazard, pipeline ***stalled*** until resolved
 - If control hazard, pipeline ***flushed*** of wrong path instructions

Dealing with Hazards Using Compiler Scheduling

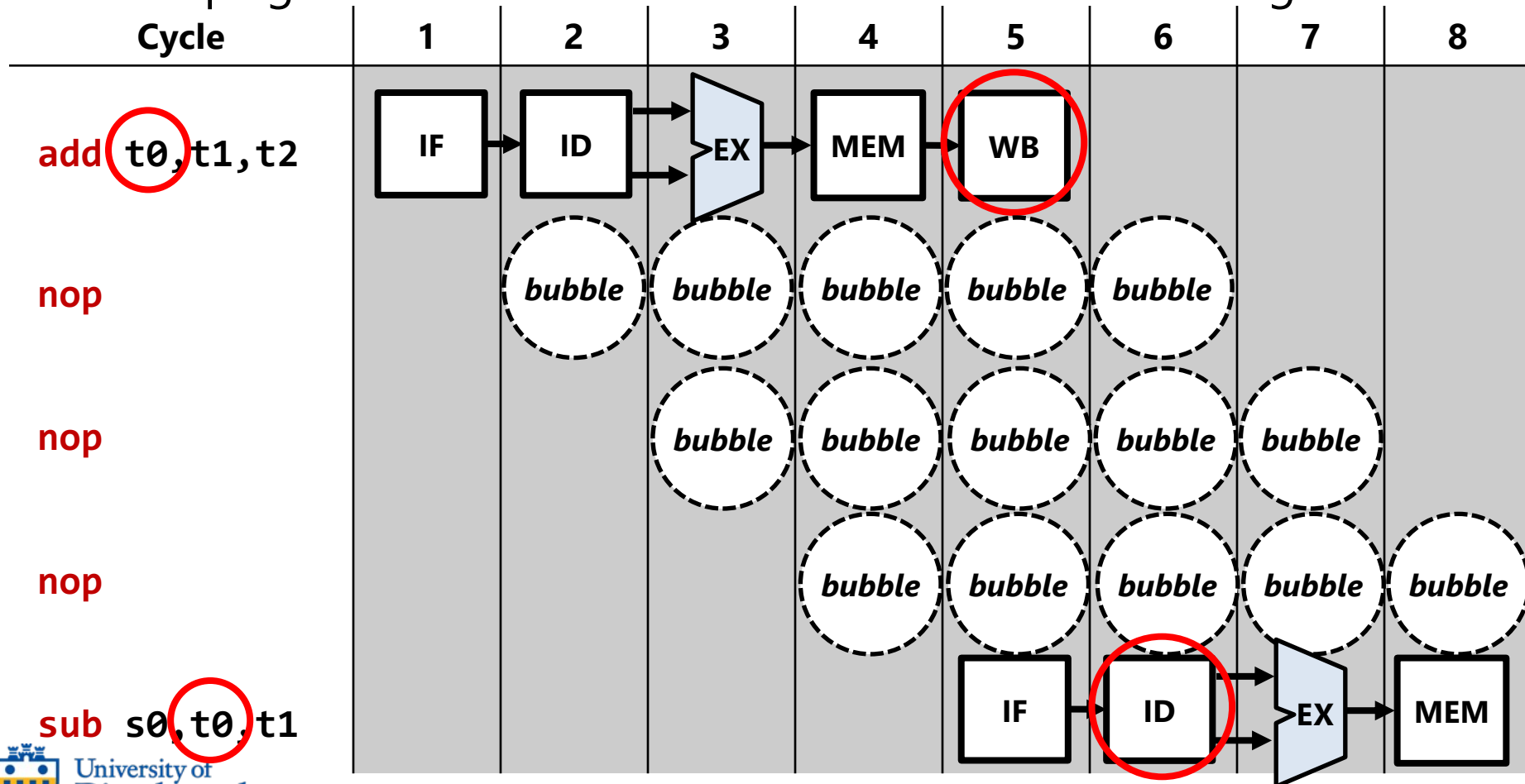
Compiler avoiding a structural hazard

- The nop avoids simultaneous access by the MEM and IF stages



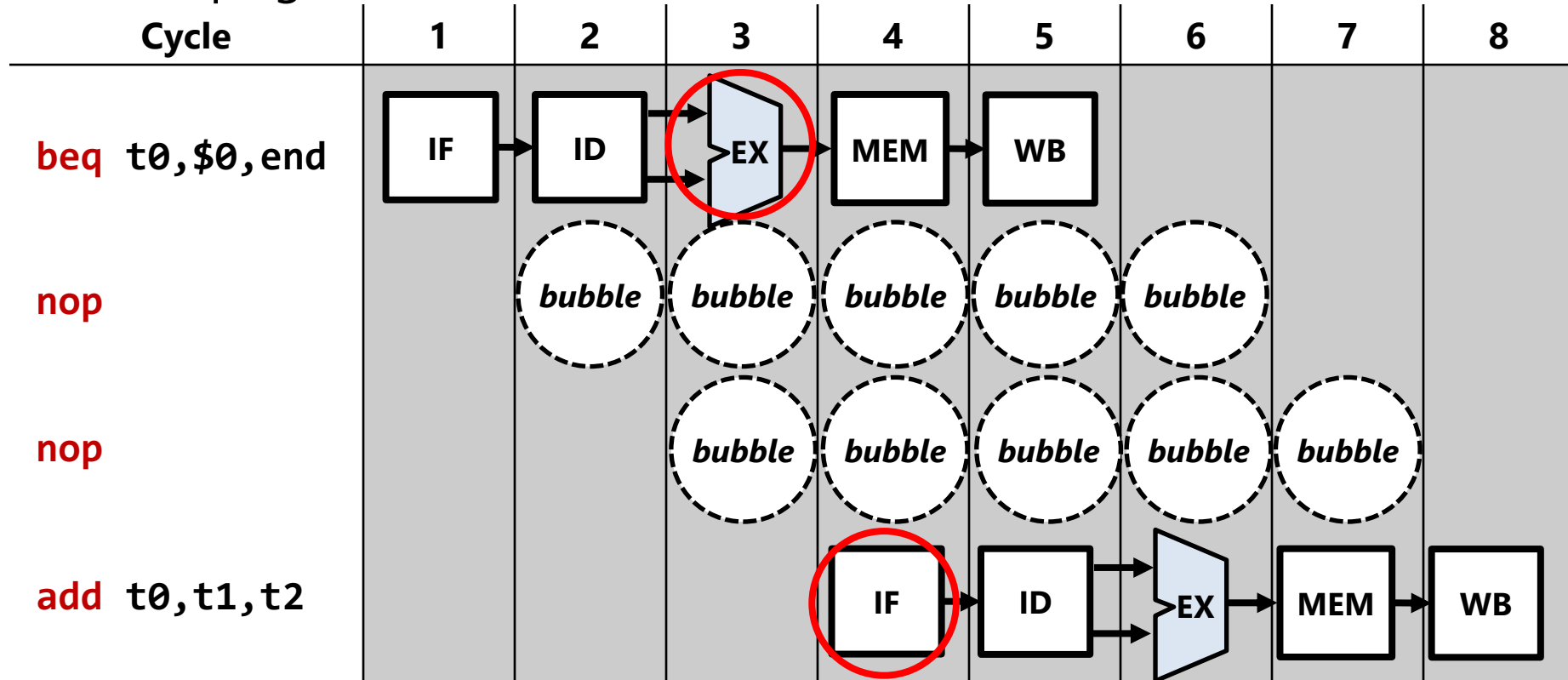
Compiler avoiding a data hazard

- The nops give time for **t0** to be written back before being read



Compiler avoiding a control hazard

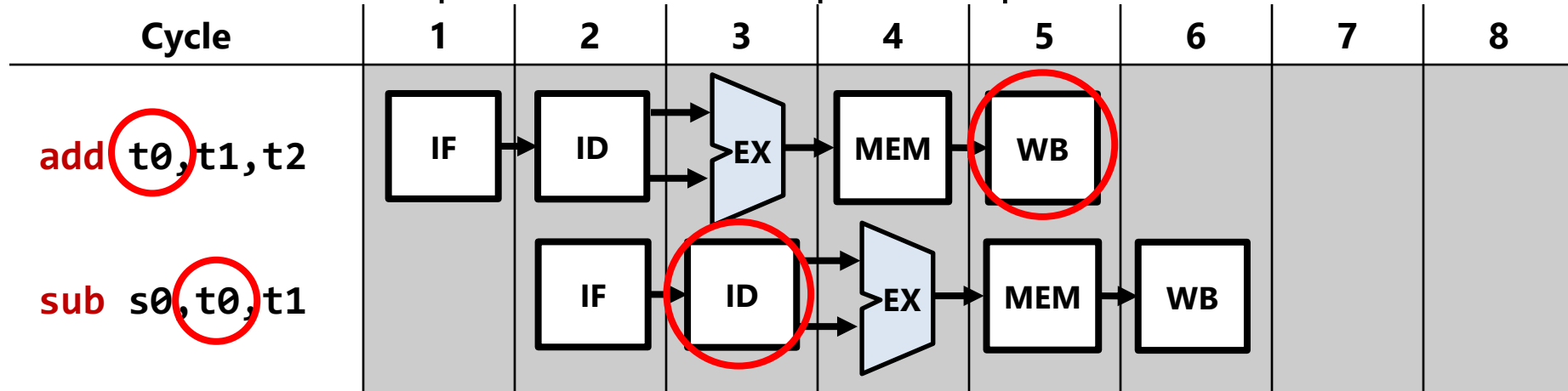
- The nops give time for condition to resolve before instruction fetch



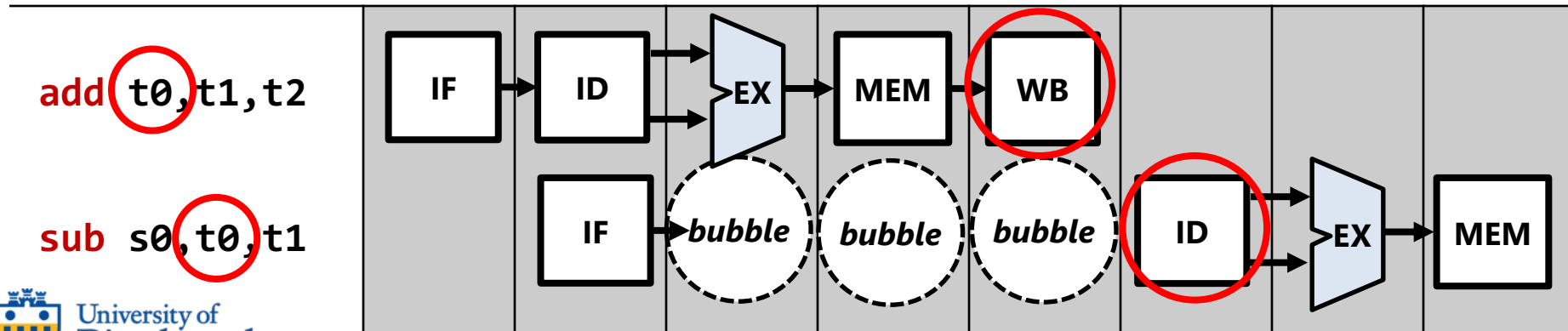
Dealing with Hazards Using Hardware Scheduling

Hardware avoiding a data hazard

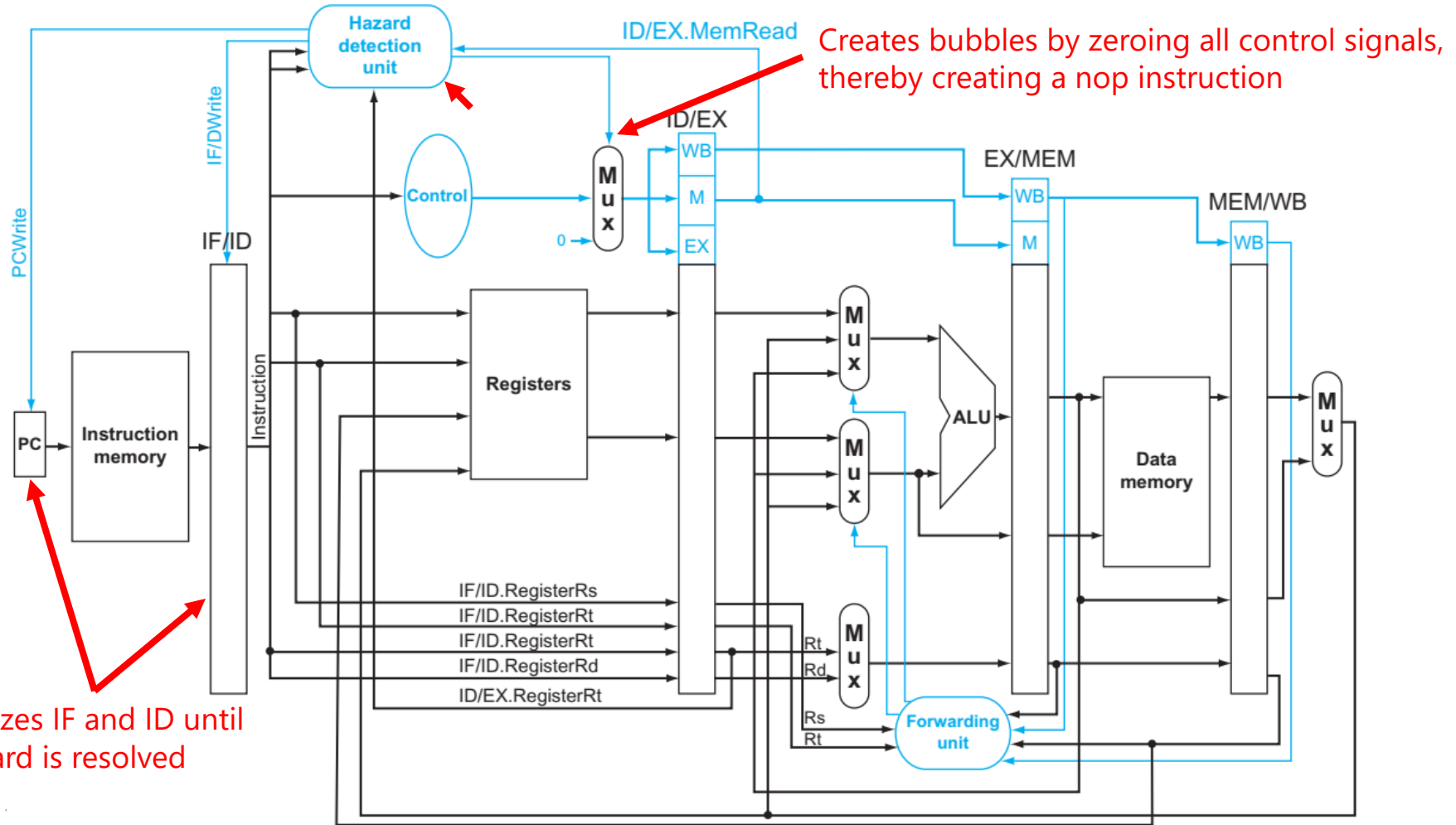
- An instruction depends on the output of a previous one.



- sub** waits until **add**'s WB phase is over before doing its ID phase

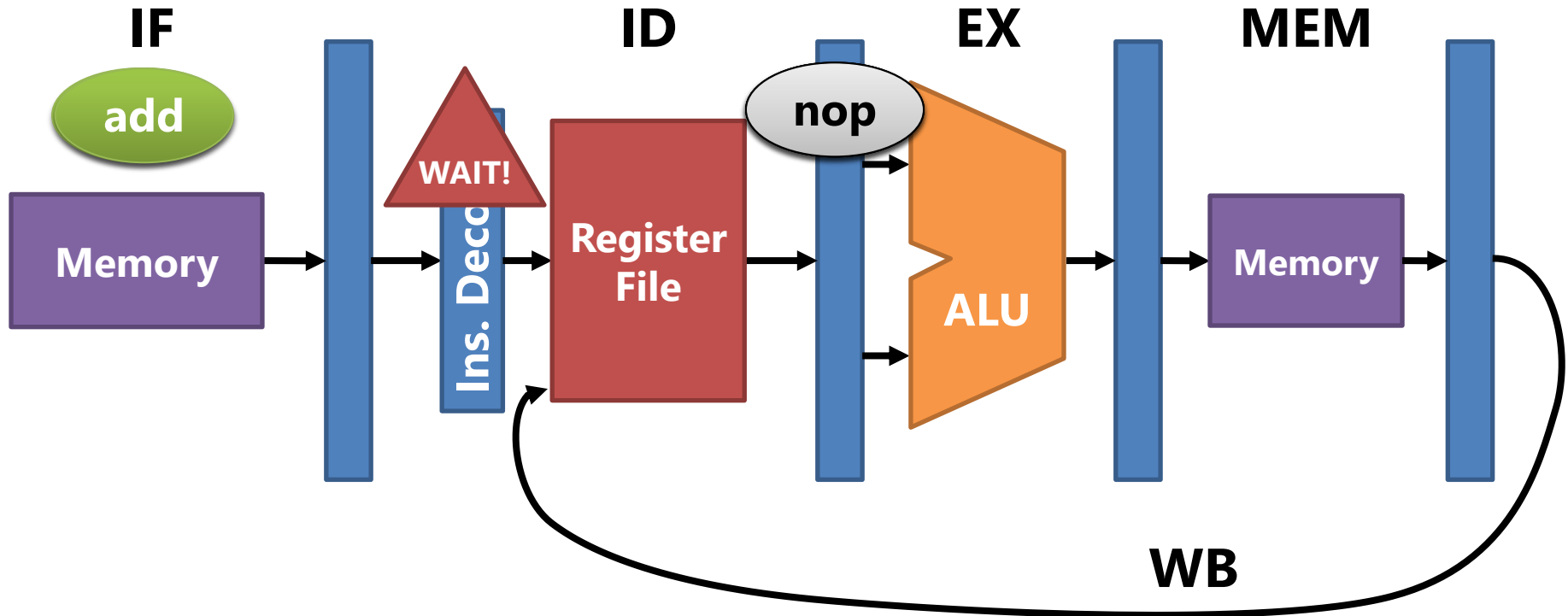


Hazard Detection Unit (HDU)



Hazard Detection Unit avoiding a data hazard

- Suppose we have an **add** that depends on an **lw**.

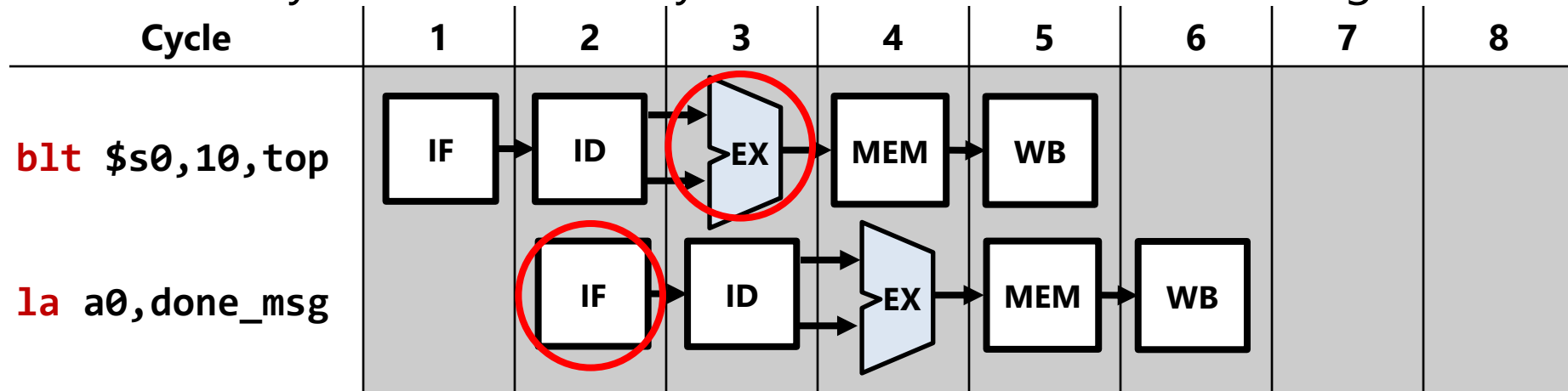


Structural / Data Hazards cause stalls

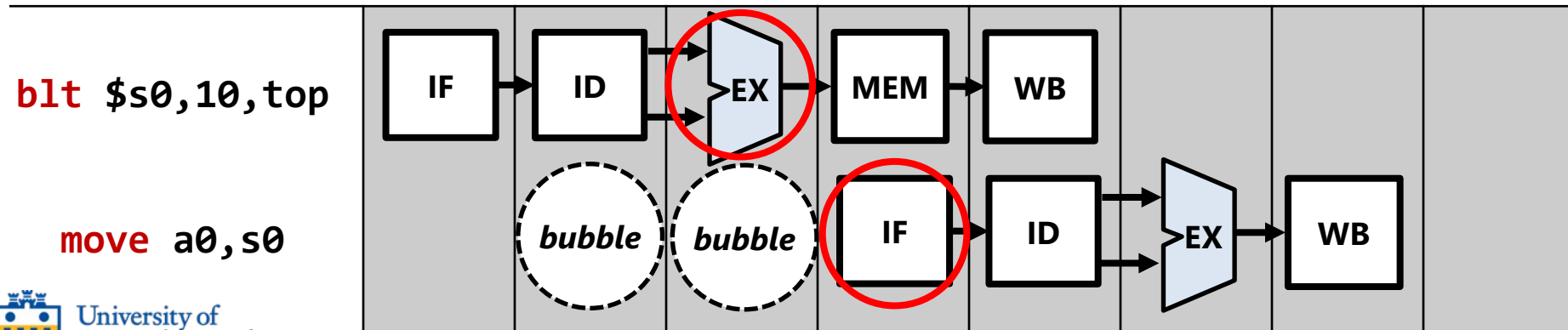
- If HDU detects a structural or data hazard, it does the following:
 - It **stops fetching instructions** (doesn't update the PC).
 - It **stops clocking the pipeline registers for the stalled stages.**
 - The stages after the stalled instructions **are filled with nops.**
 - Change control signals to 0 using the mux!
 - In this way, all following instructions will be stalled
- When structural or data hazard is resolved
 - HDU resumes instruction fetching and clocking of stalled stages
- But what about control hazards?
 - Instructions in wrong path are already in pipeline!
 - Need to **flush** these instructions

Hardware avoiding a control hazard

- You already fetched **1a** but you later discover it's the wrong branch.



- HDU flushes instructions fetched while resolving branch.



Control Hazard Example

- Supposed we had this for loop followed by printf("done"):

```
for(s0 = 0 .. 10)  
    print(s0);
```

```
printf("done");
```

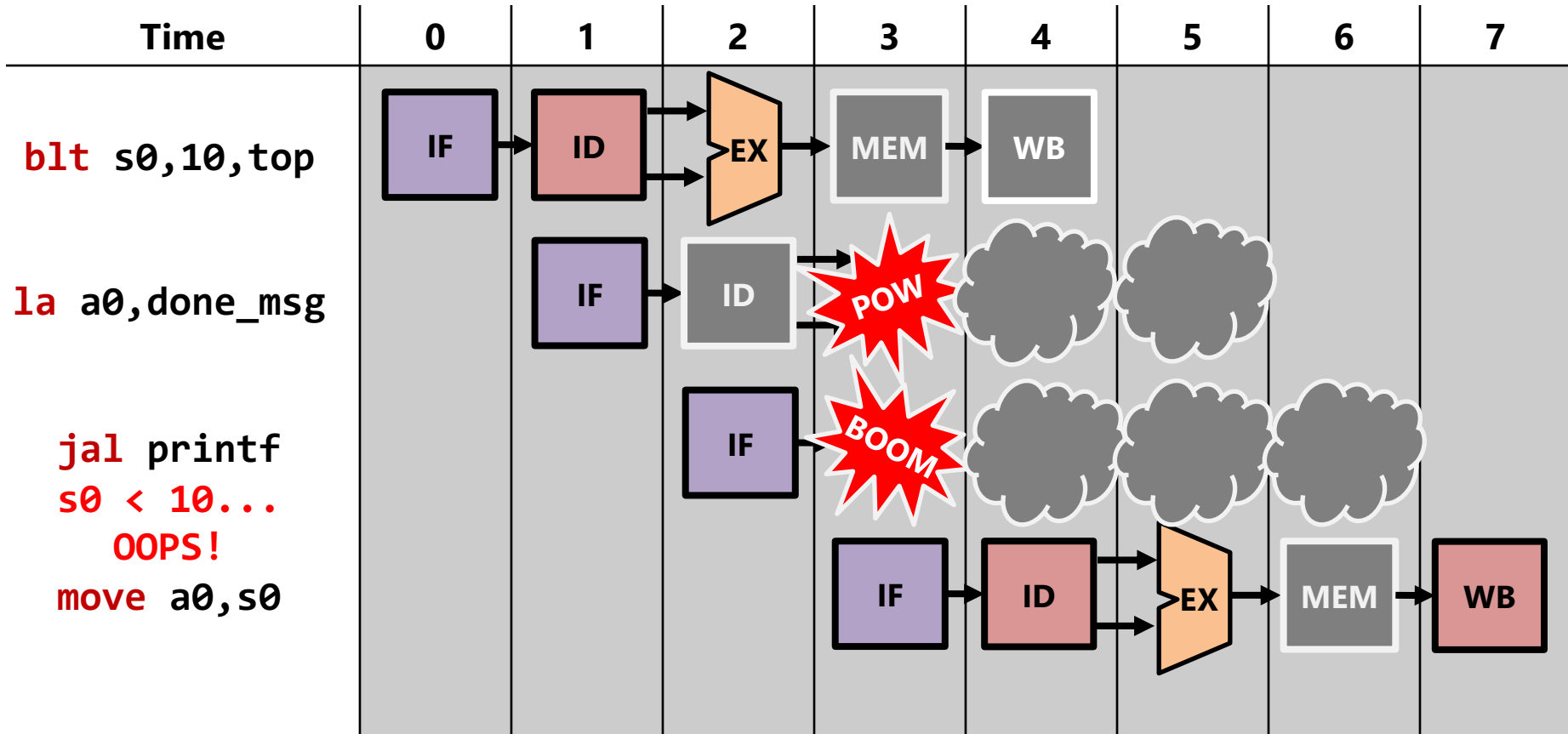
By the time `s0, 10`
are compared at `blt`
EX stage, the CPU
would have already
fetched `la` and `jal`!

```
li    s0, 0  
top:  
move  a0, s0  
jal   print  
addi  s0, s0, 1  
blt   s0, 10, top
```

```
la    a0, done_msg  
jal   printf
```

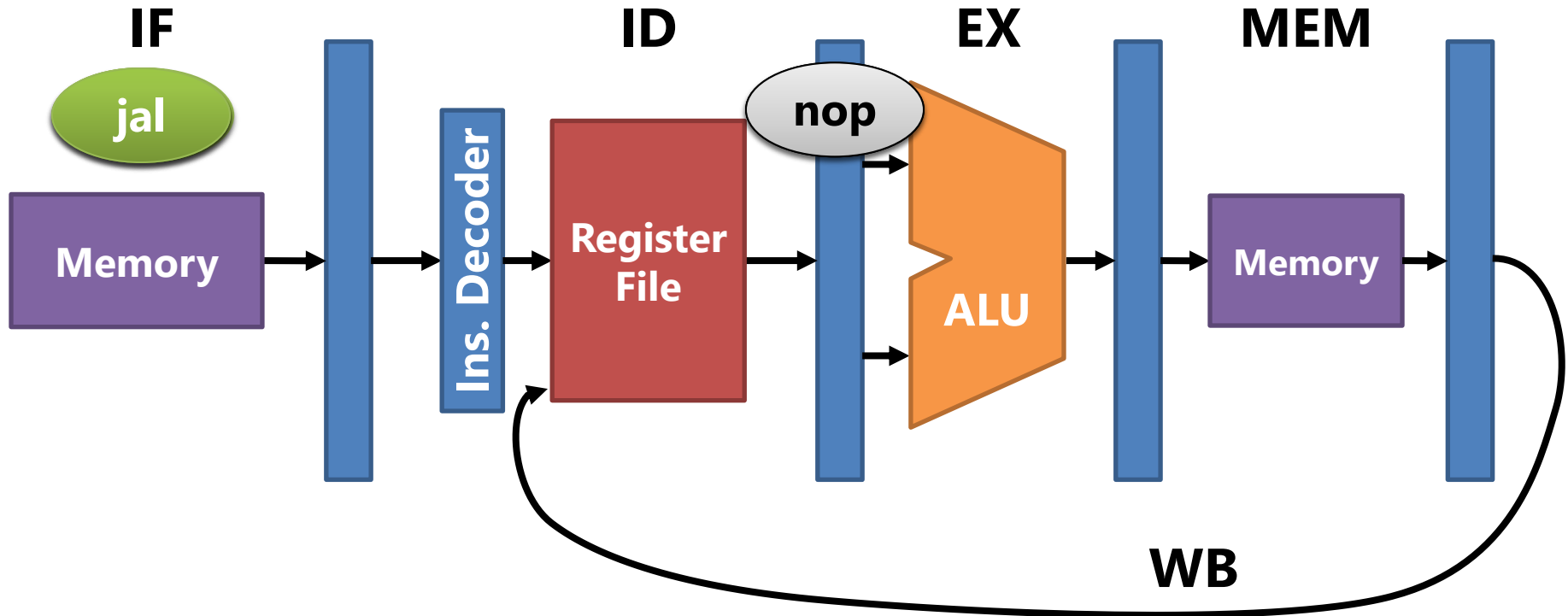
What's a flush?

- A pipeline flush removes all wrong path instructions from pipeline



Hazard Detection Unit avoiding a control hazard

- Let's watch the previous example.



Control Hazards cause flushes

- If a control hazard is detected due to a branch instruction:
 - Any "newer" instructions (those already in the pipeline)
 - transformed into **nops**.
 - Any "older" instructions (those that came BEFORE the branch)
 - left alone to finish executing as normal.

Performance penalty of pipeline stalls

- Remember the three components of performance:







$$\frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}$$

Architecture	Instructions	CPI	Cycle Time (1/F)
Single-cycle	Same	1	5 ns
Ideal 5-stage pipeline	Same	1	1 ns
Pipeline w/ stalls	Same	> 1	1 ns

- Pipelining increases **clock frequency** proportionate to depth
- Stalls and flushes increase **CPI** (cycles per instruction)
- These stalls and flushes can be sometimes avoided.
 - Wait for it... it is coming up in the next chapter.

Compiler scheduling vs. Hardware scheduling

Compiler scheduling vs. Hardware scheduling

	Compiler-scheduled	Hardware-scheduled
Energy Efficiency	 Scheduling happens off-line.	 Hazard detection by HDU and scheduling done at runtime.
Binary Portability	 <ul style="list-style-type: none">• Compiled binary contains assumptions about processor pipeline design• These assumptions must become part of ISA.	 ISA strictly about instruction functionality with no assumptions on hazards or delays due to instruction.
Quality of Scheduling	 <ul style="list-style-type: none">• Pro: No limit to scheduling scope (full view of code)• Con: No runtime info. Hard to predict hazards. Hard to predict delay of memory instructions (due to caching).	 <ul style="list-style-type: none">• Pro: Can know exactly presence of hazards and instructions delays at runtime.• Con: Scheduling scope limited by size of hardware instruction scheduling queue