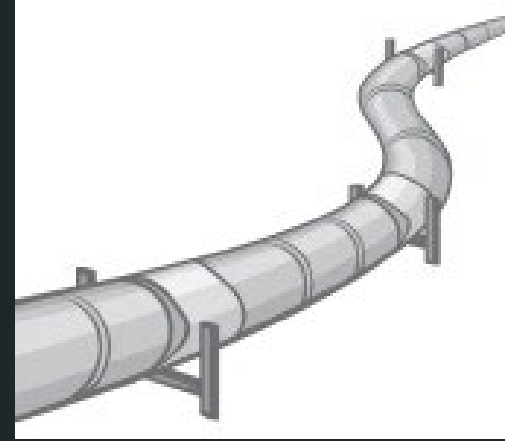# Processor Pipelining

CS 1541
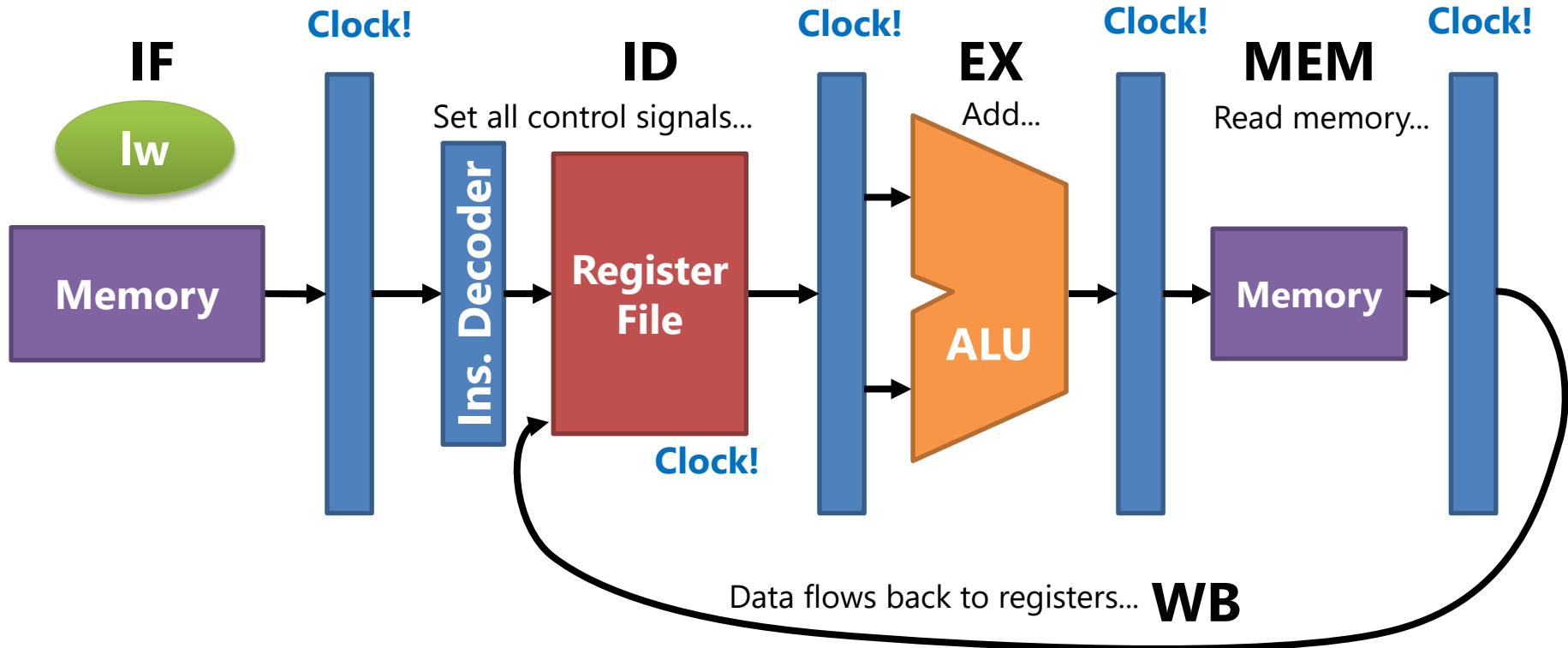Wonsun Ahn

Pipelining Basics

Improving Washer / Dryer / Closet Utilization

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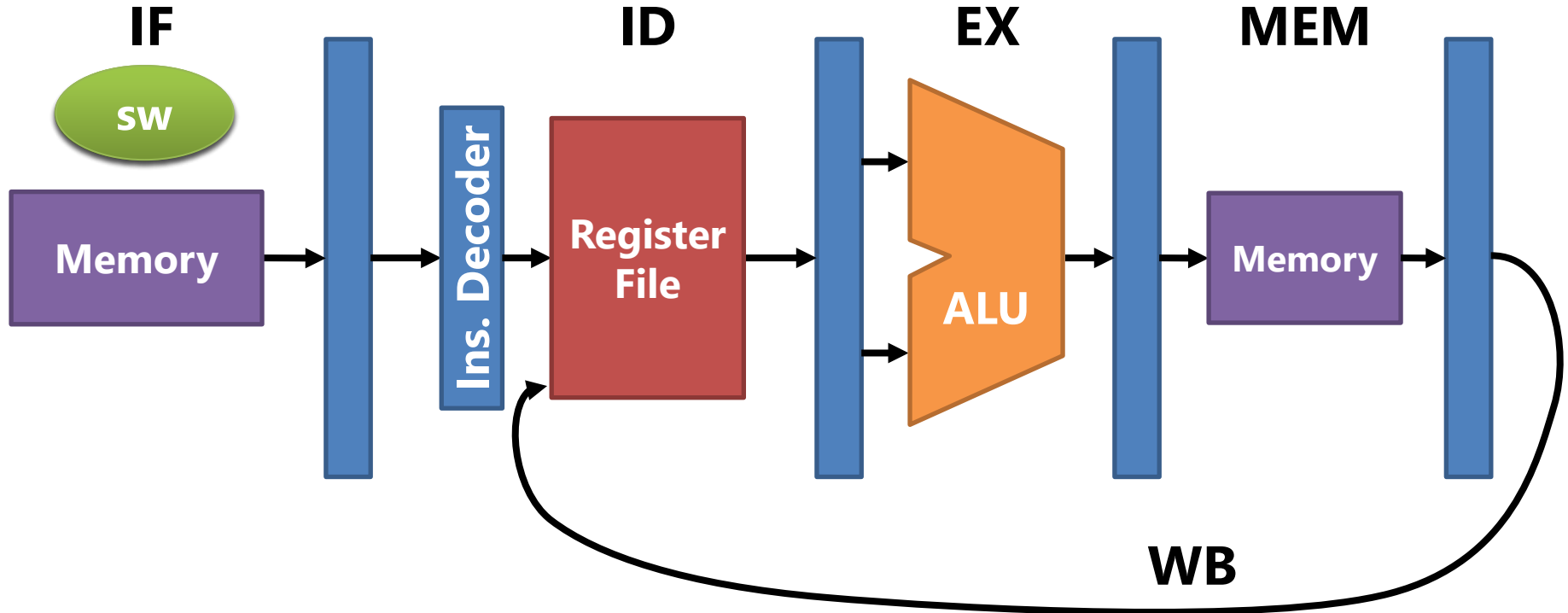- If you form an "assembly line", you achieve ~100% utilization!

Multi-cycle instruction execution

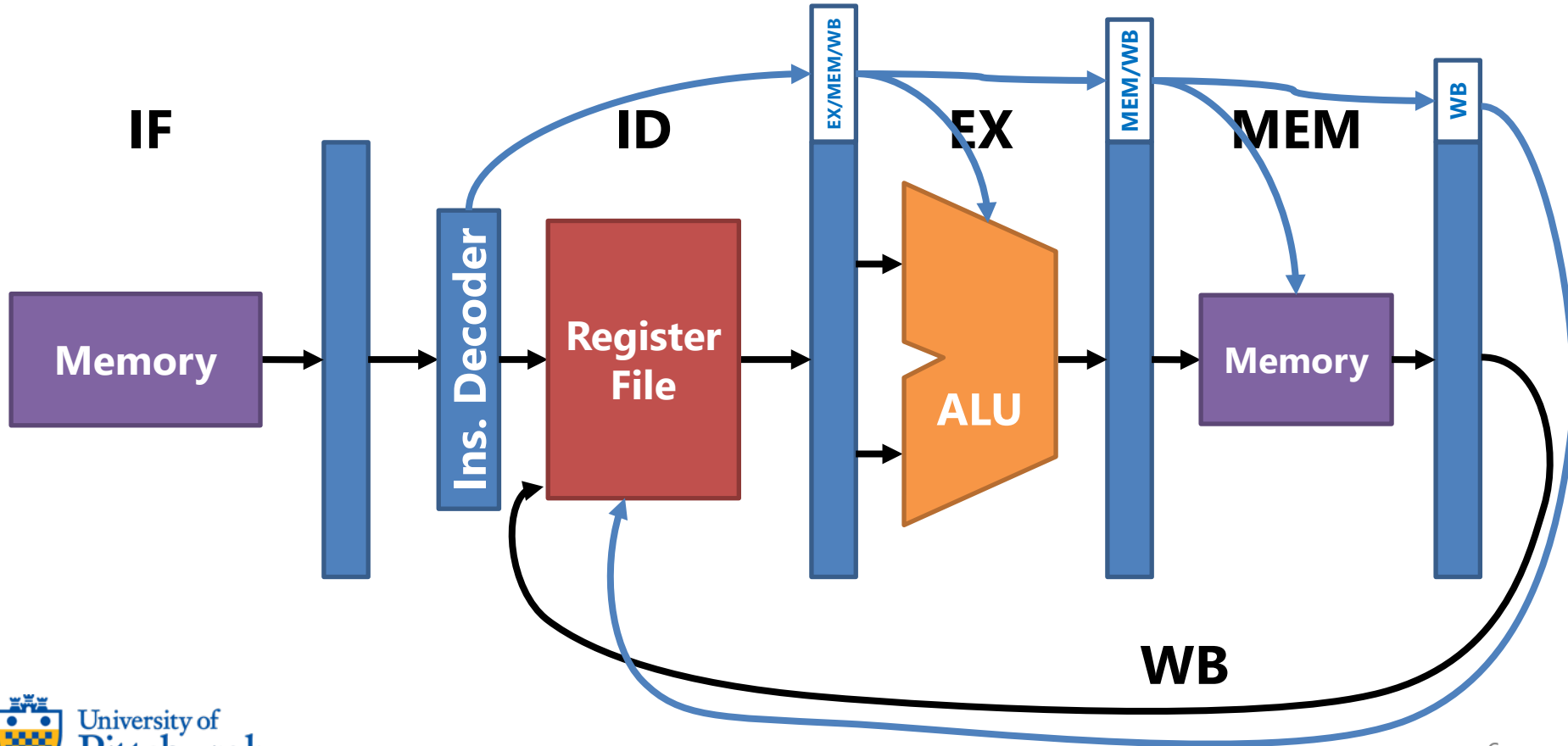● Let's watch how an instruction flows through the datapath.

Pipelined instruction execution

- Pipelining allows one instruction to be fetched each cycle!

- 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.*

Cycle	1	2	3	4	5	6	7	8
add t0,t1,t2	IF	ID	EX	MEM	WB			
add t3,t4,t5		IF	ID	EX	MEM	WB		
add s0,s1,s2			IF	ID	EX	MEM	WB	
add s3,s4,s5				IF	ID	EX	MEM	WB
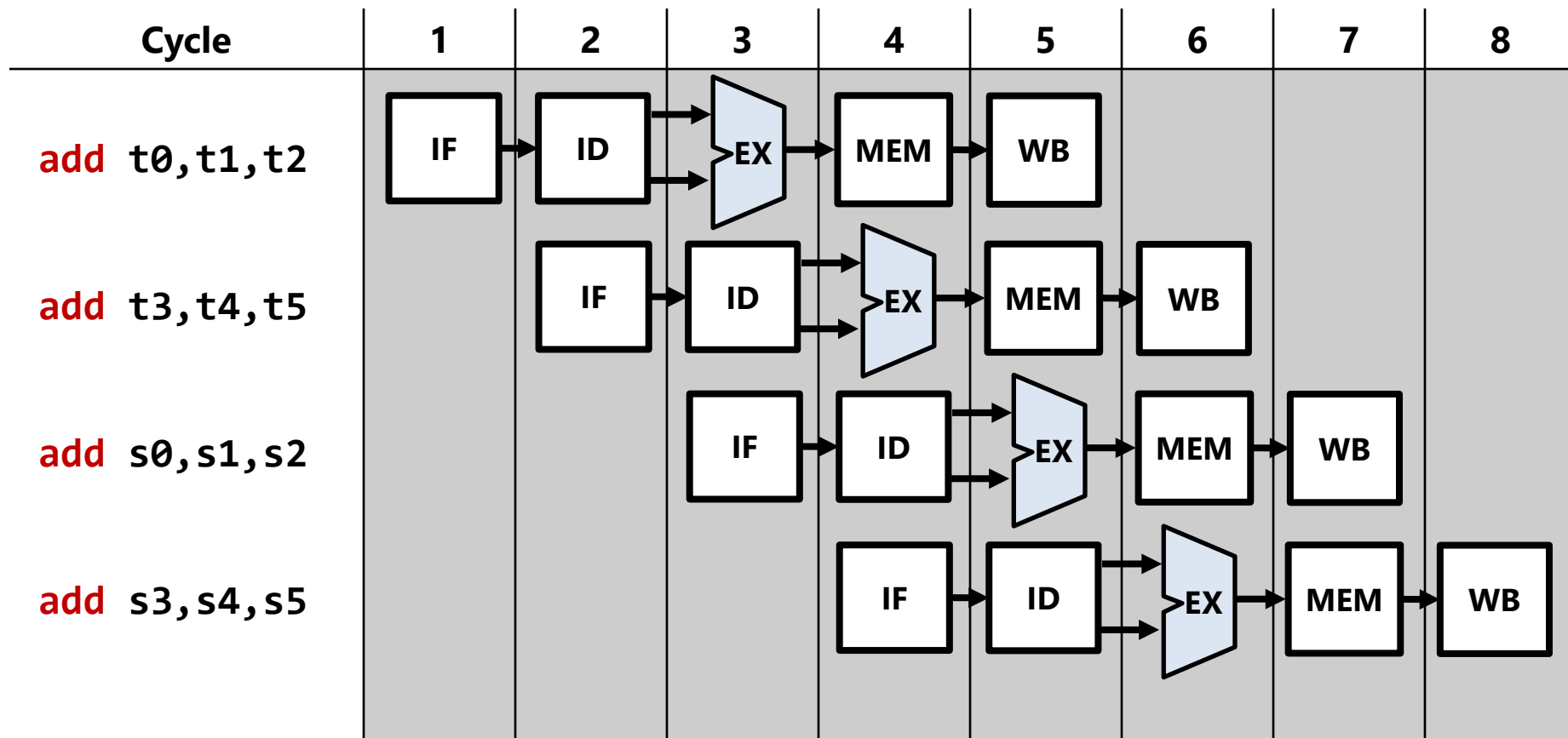
- Now every instruction takes the same number of cycles
  - **lw** takes 5 cycles: IF/ID/EX/MEM/WB
  - **add** takes 5 cycles: IF/ID/EX/---/WB
  - **sw** takes 5 cycles: IF/ID/EX/MEM/---

- If each stage takes *1 ns* each:

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

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

- What happened to the three components of performance?

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

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
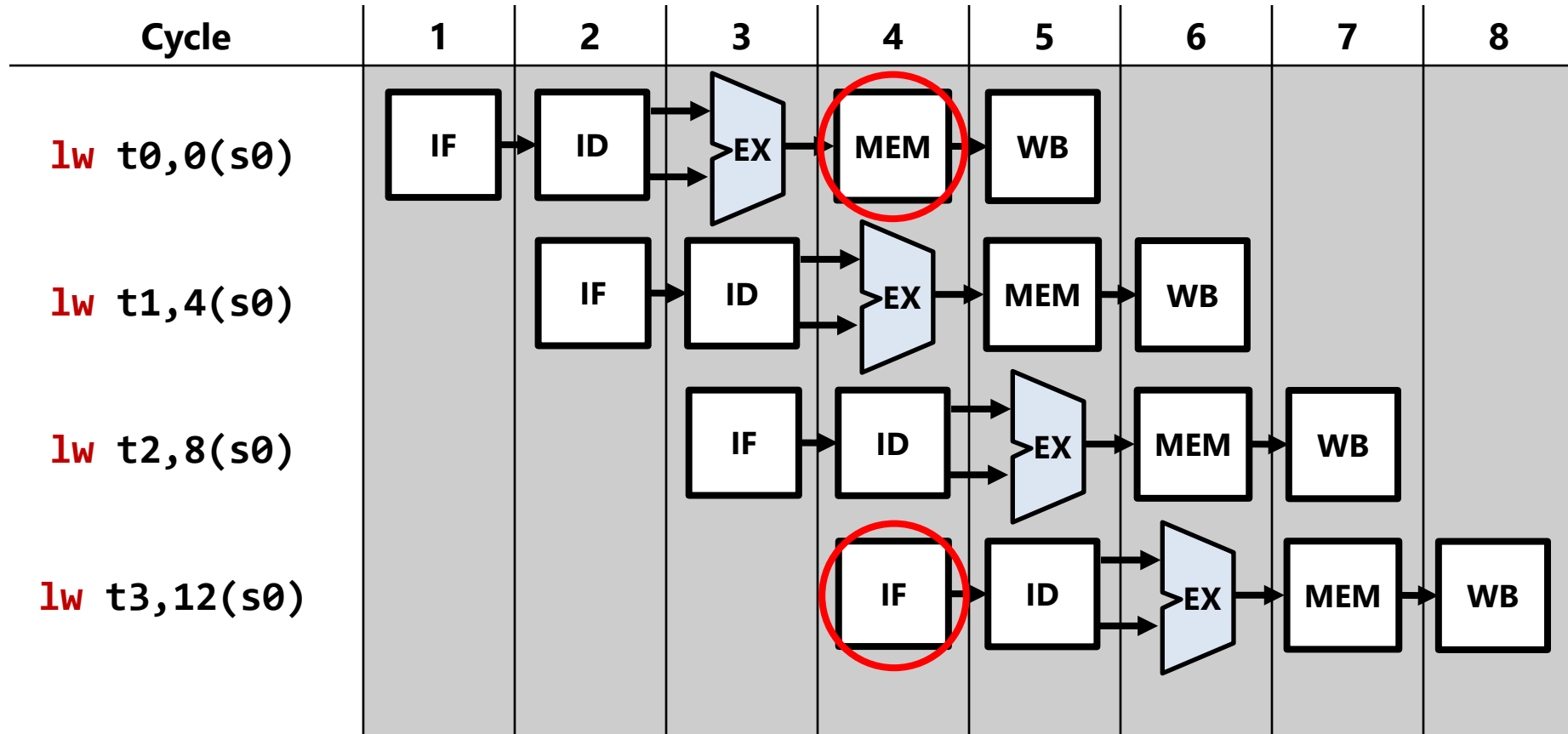
University of Pittsburgh

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.

Cycle	1	2	3	4	5	6	7	8
lw t0,0(s0)	IF	ID	EX	MEM	WB			
lw t1,4(s0)		IF	ID	EX	MEM	WB		
lw t2,8(s0)			IF	ID	EX	MEM	WB	
lw t3,12(s0)				IF	ID	EX	MEM	WB

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

Cycle	1	2	3	4	5	6	7	8

add t0,t1,t2

IF → ID → EX → MEM → WB

sub s0,t0,t1

IF → ID → EX → MEM → WB
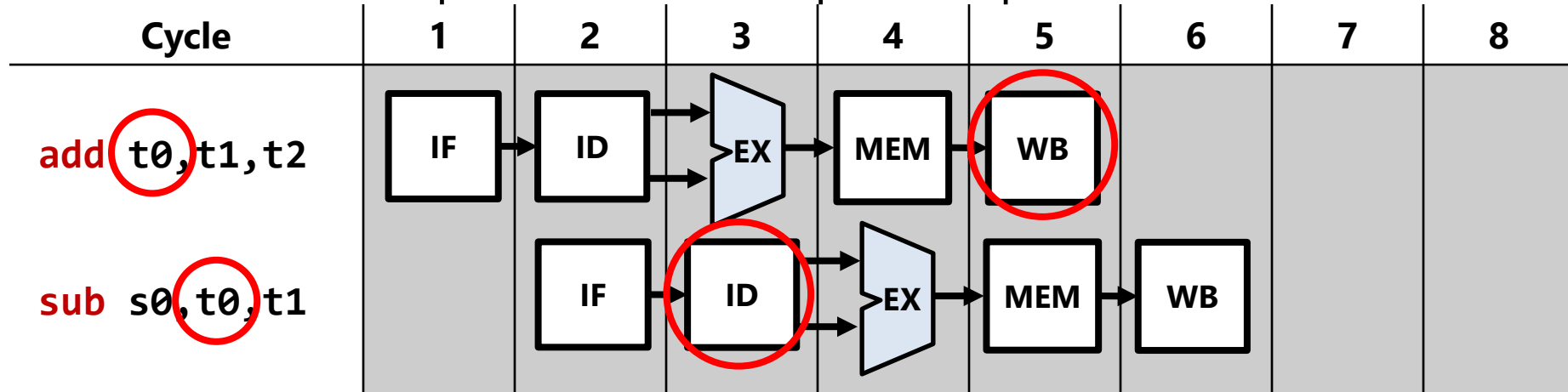
● **sub** reads in **t0** before **add** has had a chance to write it back!

University of Pittsburgh

- An instruction depends on branch outcome of previous instruction.

Cycle	1	2	3	4	5	6	7	8
`beq t0,$0,end`	IF	ID	EX	MEM	WB			
`add t0,t1,t2`		IF	ID	EX	MEM	WB		
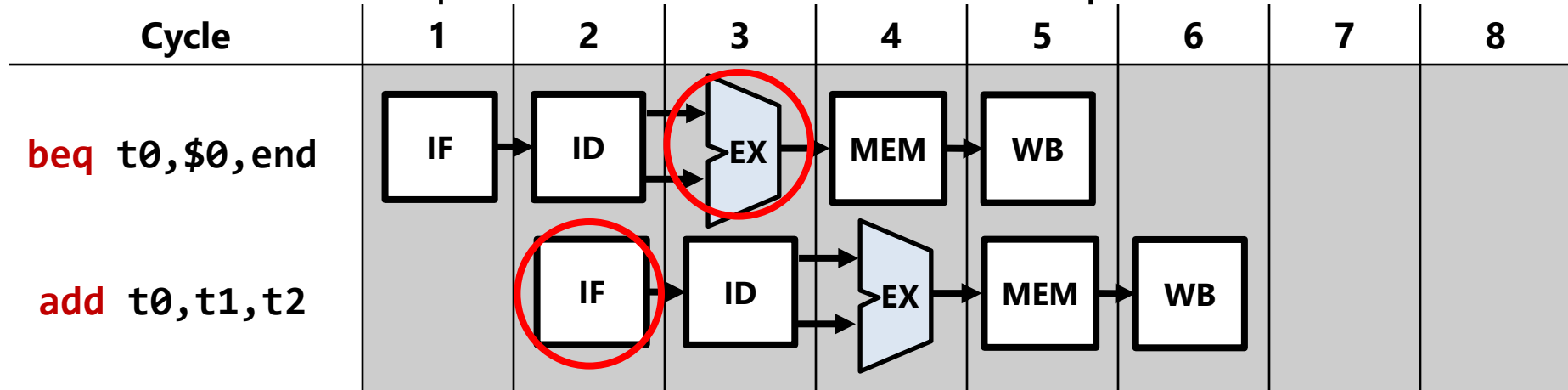
- **add** (PC+4) is fetched before **beq** branch outcome is known!

University of Pittsburgh
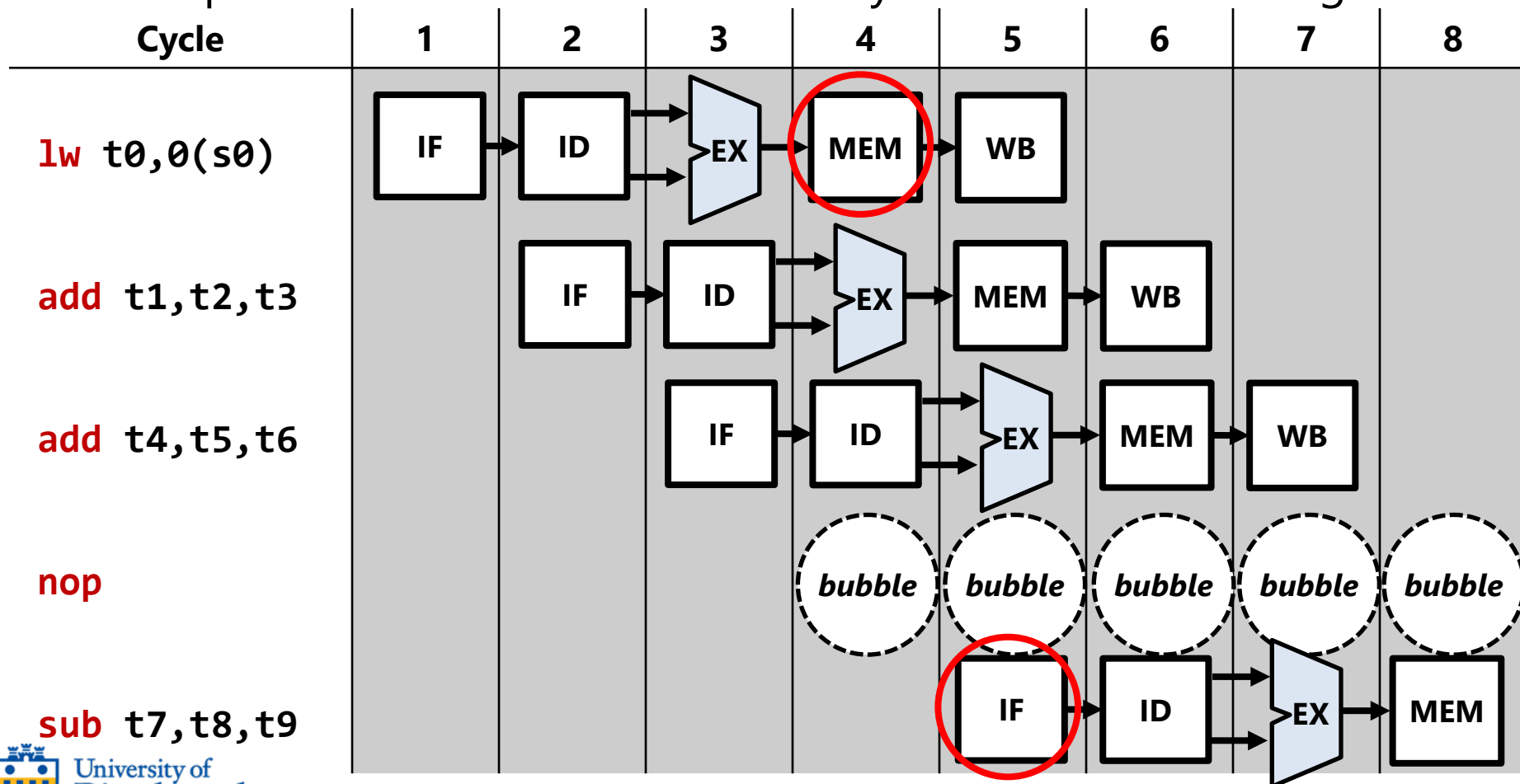
- Pipeline must be controlled so that hazards don't cause malfunction

- 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
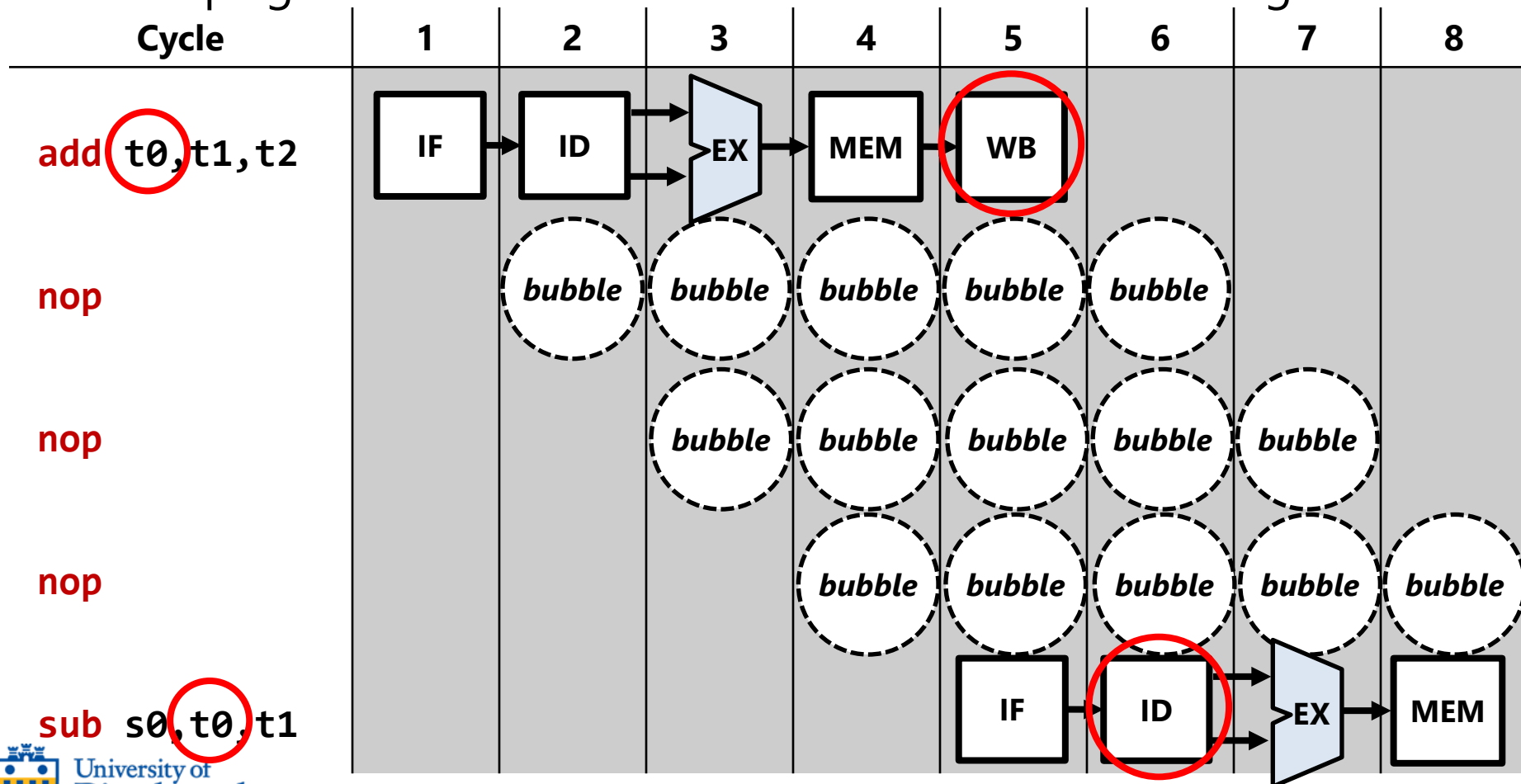
Dealing with Hazards Using Compiler Scheduling

University of Pittsburgh

Compiler avoiding a structural hazard

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

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

Cycle	1	2	3	4	5	6	7	8
add t0,t1,t2	IF	ID	EX	MEM	WB			
nop		bubble	bubble	bubble	bubble	bubble		
nop			bubble	bubble	bubble	bubble	bubble	
nop				bubble	bubble	bubble	bubble	bubble
sub s0,t0,t1					IF	ID	EX	MEM

Compiler avoiding a control hazard

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

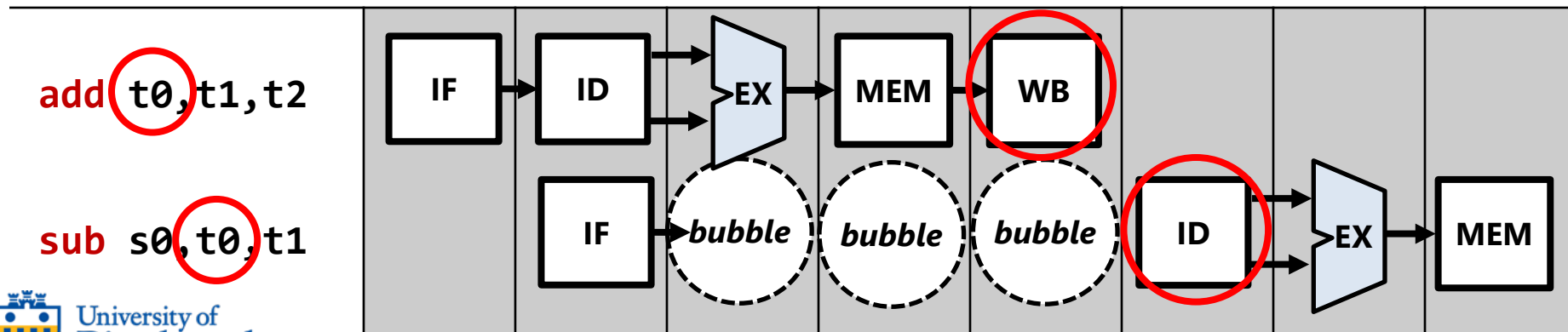| Cycle	1	2	3	4	5	6	7	8
beq t0,$0,end	IF	ID	EX	MEM	WB			
nop		bubble	bubble	bubble	bubble	bubble		
nop			bubble	bubble	bubble	bubble	bubble	
add t0,t1,t2				IF	ID	EX	MEM	WB
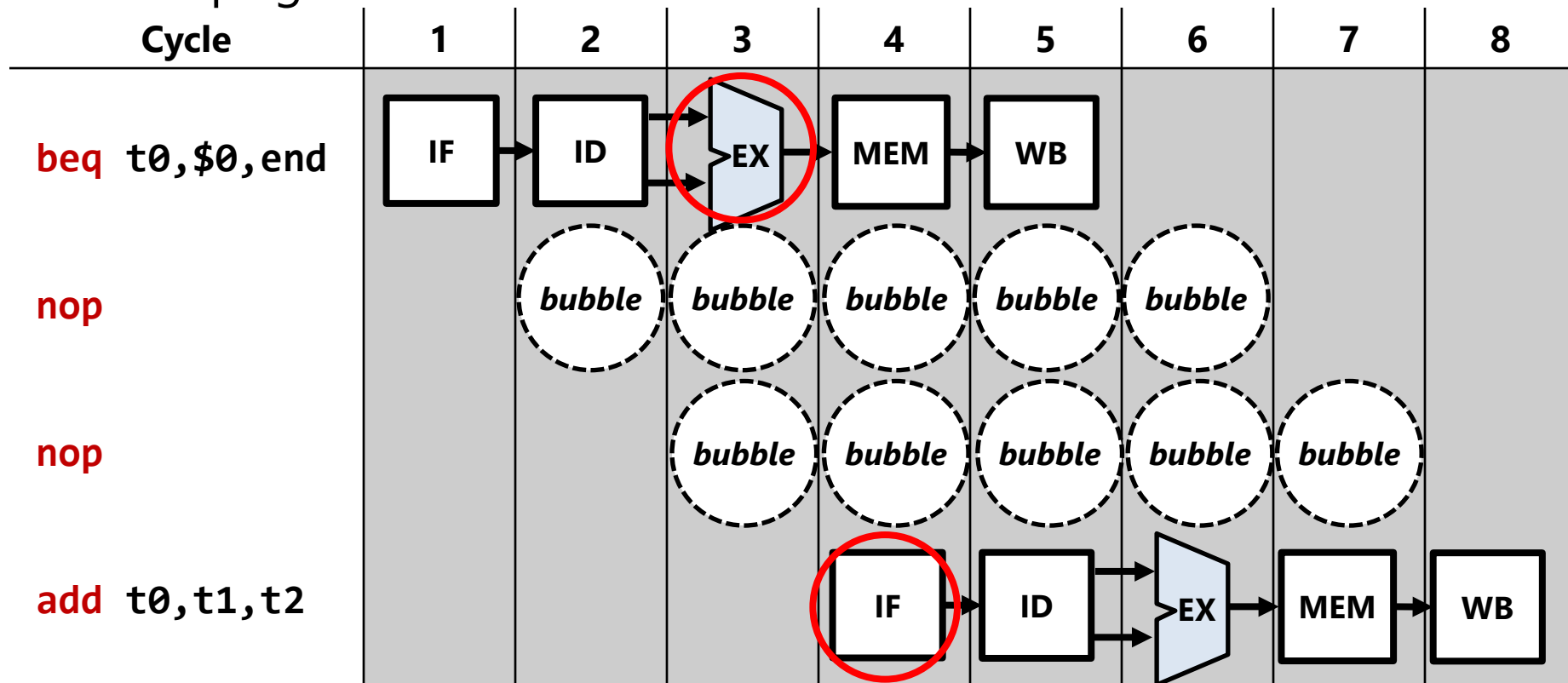
Dealing with Hazards Using Hardware Scheduling

Hardware avoiding a data hazard

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

Cycle	1	2	3	4	5	6	7	8
add t0,t1,t2	IF	ID	EX	MEM	WB			
sub s0,t0,t1		IF	ID	EX	MEM	WB		
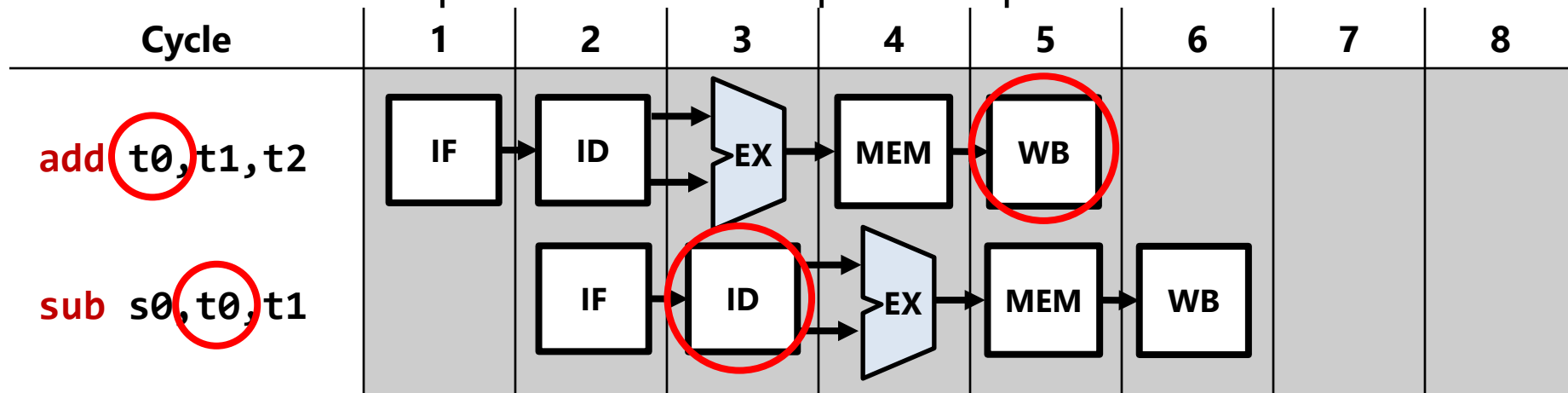
- **sub** waits until **add**'s WB phase is over before doing its ID phase

add t0,t1,t2	IF	ID	EX	MEM	WB			
sub s0,t0,t1		IF	*bubble*	*bubble*	*bubble*	ID	EX	MEM

Creates bubbles by zeroing all control signals, thereby creating a nop instruction
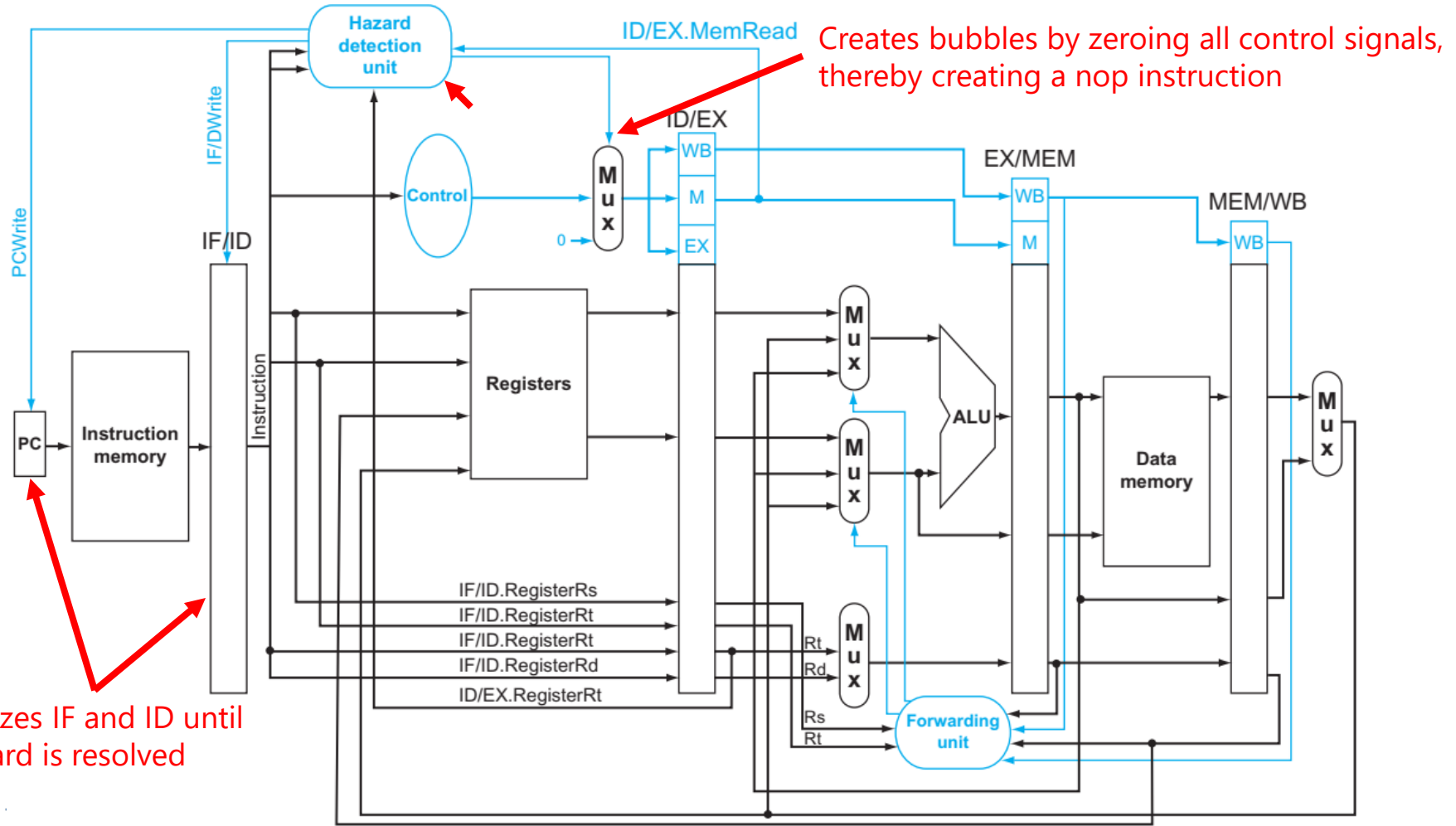
Freezes IF and ID until hazard is resolved

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

IF   **ID**   **EX**   **MEM**

add

WAIT!

nop

Ins. Decode

Memory

Register File

ALU

Memory
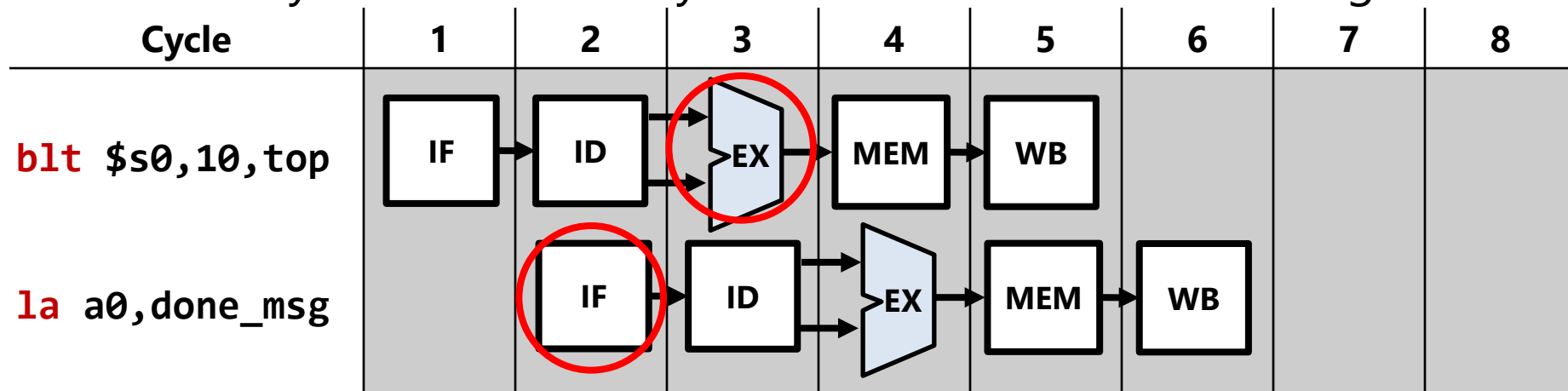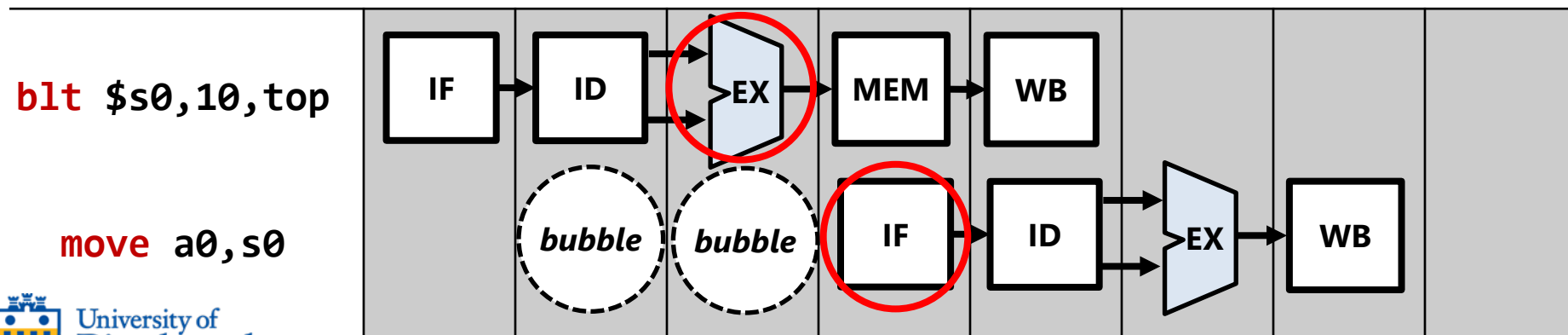
WB
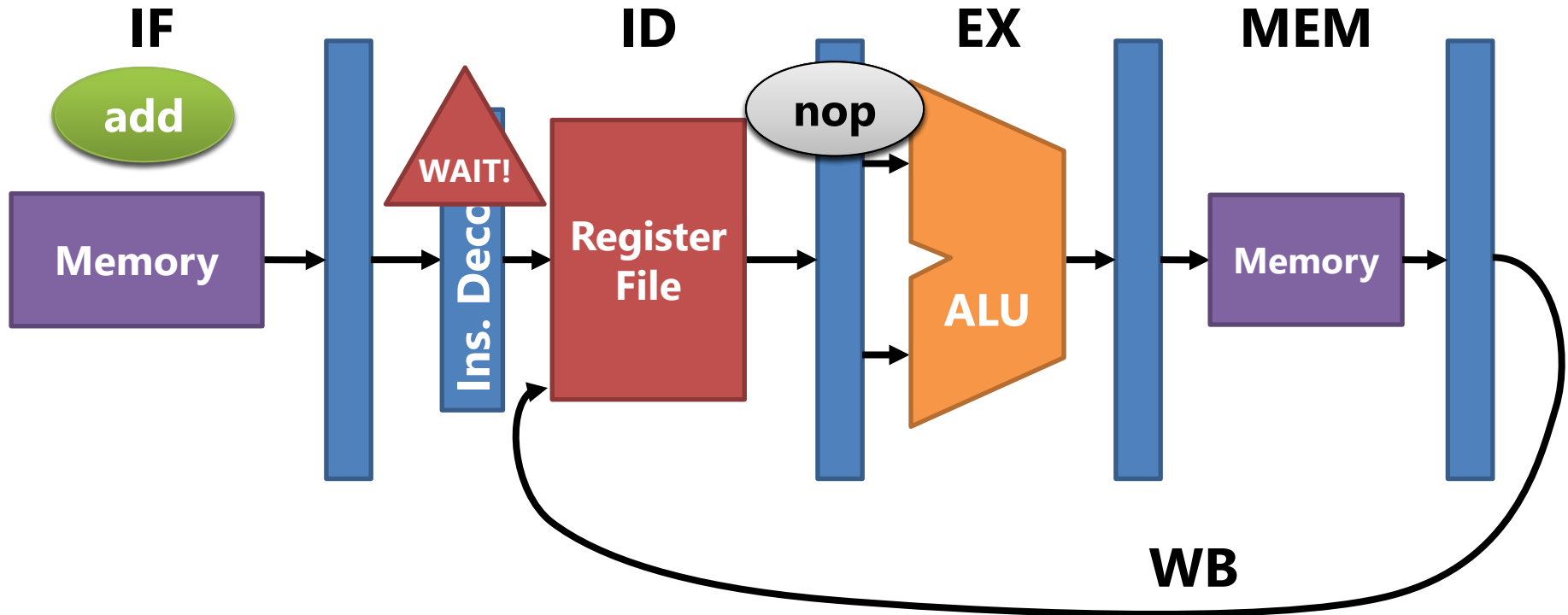
- If HDU detects a structural or data hazard, it does the following:
  - It **stops fetching instructions** (doesn't update the PC).
  - It **stops clocking the pipeline registers for the stalled stages.**
  - The stages after the stalled instructions **are filled with nops.**
    - Change control signals 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

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

Cycle	1	2	3	4	5	6	7	8
blt $s0,10,top	IF	ID	EX	MEM	WB			
la a0,done_msg		IF	ID	EX	MEM	WB		

- HDU flushes instructions fetched while resolving branch.

blt $s0,10,top	IF	ID	EX	MEM	WB			
move a0,s0		*bubble*	*bubble*	IF	ID	EX	WB	

25

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

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

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

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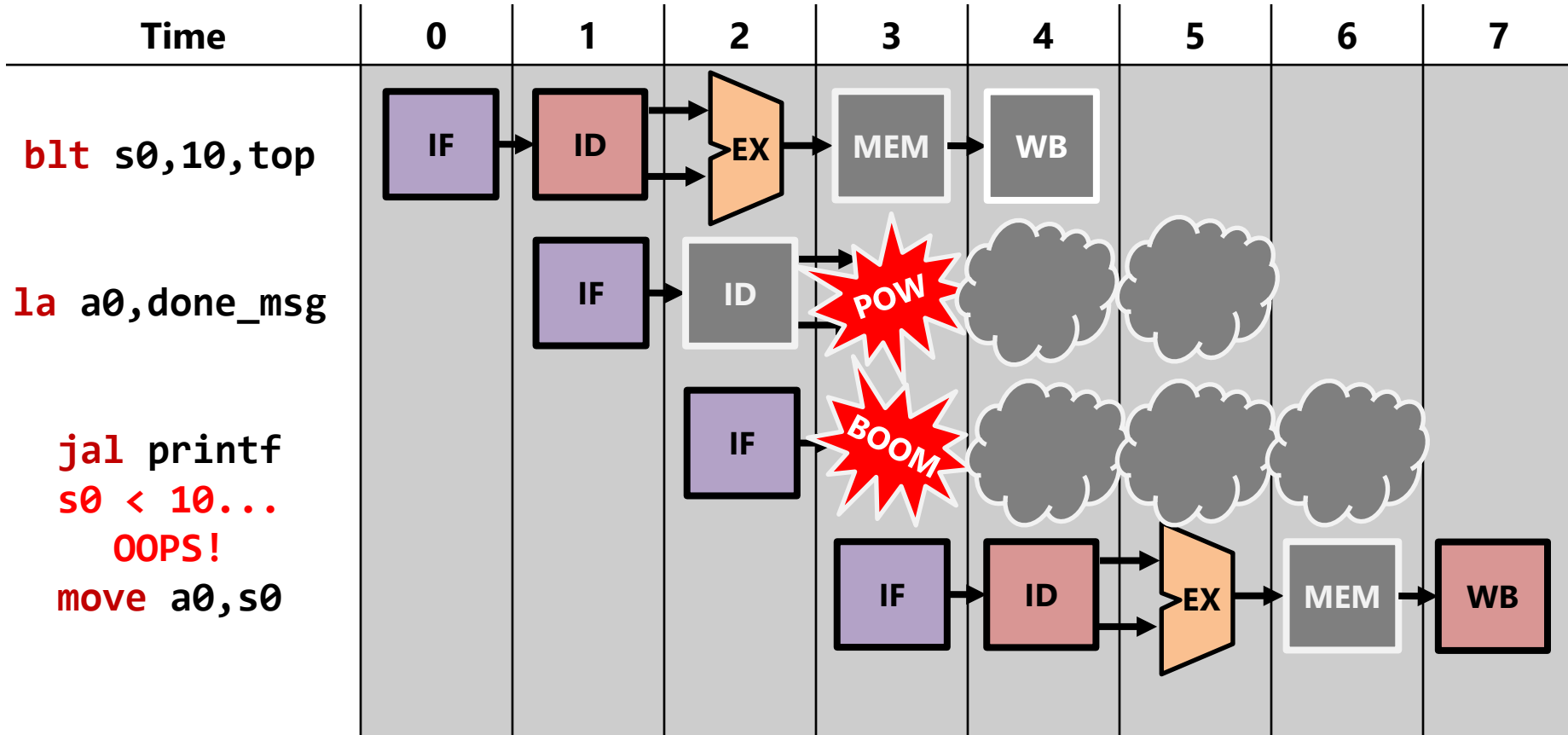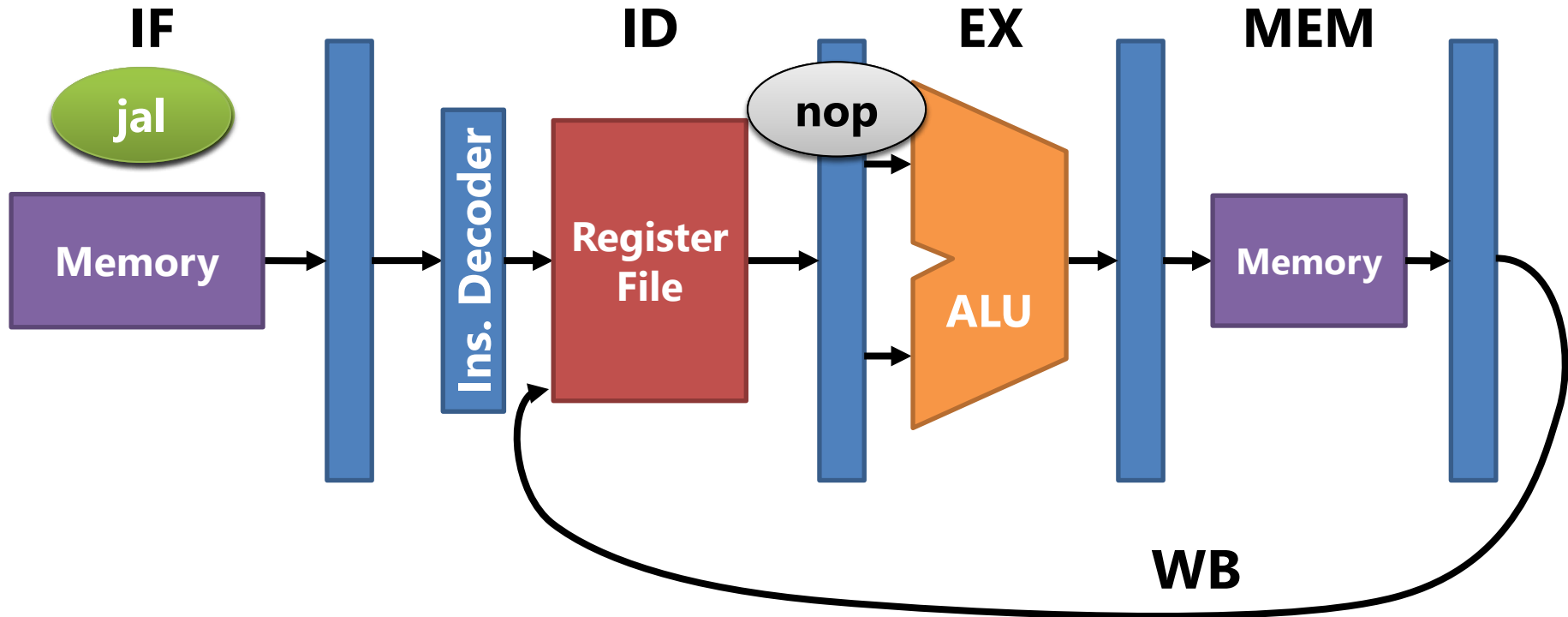
```
        la    a0, done_msg
        jal   printf
```

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

● A pipeline flush removes all wrong path instructions from pipeline

● Let's watch the previous example.

IF

ID

EX

MEM

jal

nop

Memory

Ins. Decoder

Register File

ALU

Memory

WB

- 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.

Dealing with Hazards Summary

- Compiler scheduling **pro**: Energy-efficiency
  - Hazard Detection Unit can be very power hungry
    - A lot of long wires controlling remote parts of the CPU
    - Adds to the **Power Wall** problem

- Compiler scheduling **con**: Must make assumptions about pipeline
  - That means pipeline design must become part of ISA
    - Pipeline design tends to change drastically across generations
  - Length of MEM stage is hard to predict by the compiler
    - Until now we assumed MEM takes a uniform one cycle
    - But remember what we said about the **Memory Wall**?
    - Depending on whether access hits in cache: 1 ~ 100s of cycles

Performance penalty of pipeline stalls

- Remember the three components of performance:

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

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
- But stalls increase **CPI** (cycles per instruction)
  - If CPI of 2 → Only 2.5X speed up (instead of 5X)

- We'd like to avoid this penalty if possible!

University of Pittsburgh