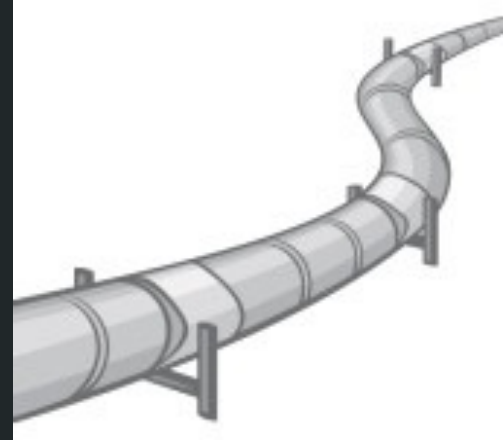


# Processor Pipelining

CS 1541

Wonsun Ahn

# Pipelining Basics



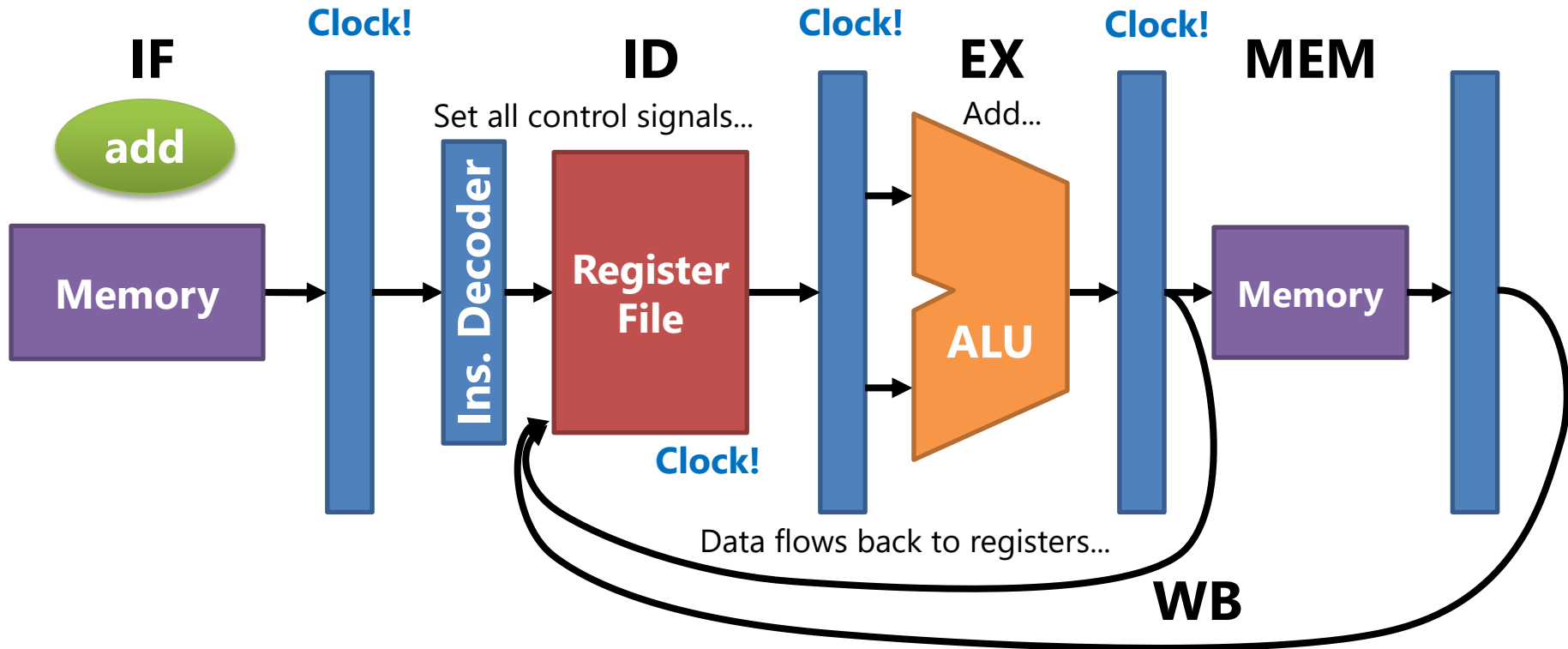
# Improving Washer / Dryer / Closet Utilization

- 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!



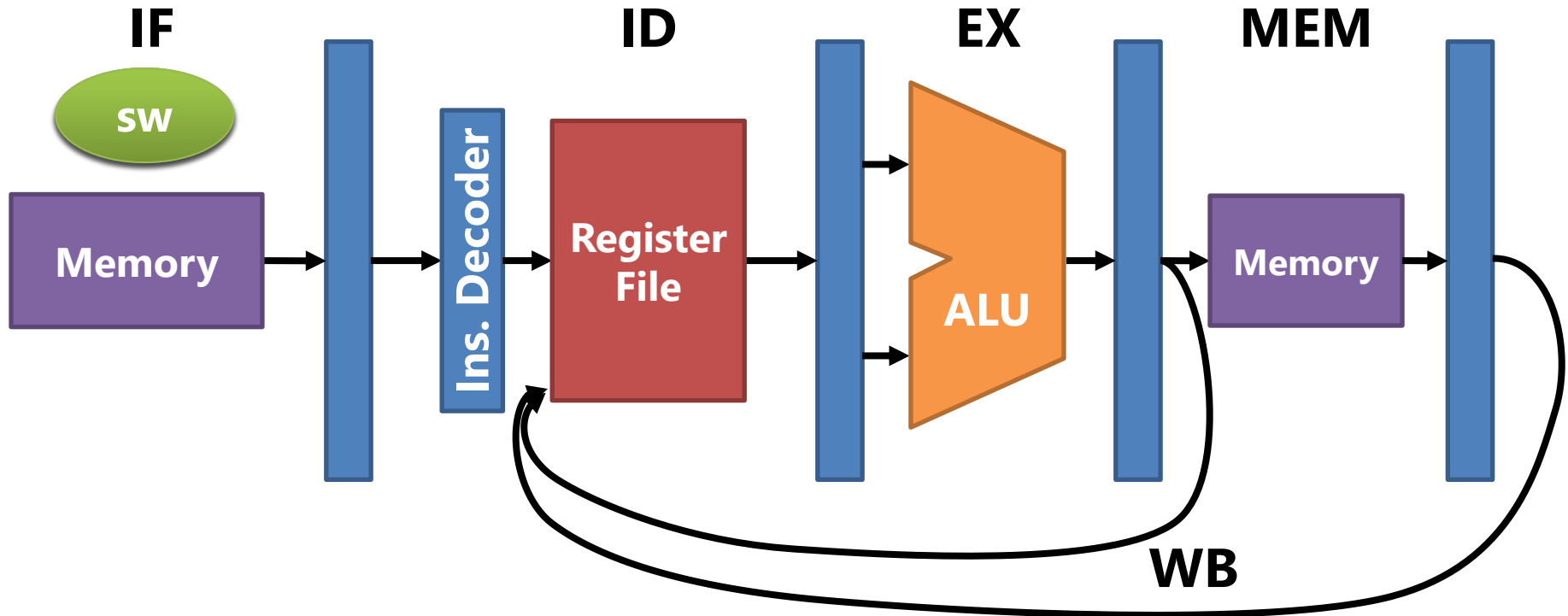
# Multi-cycle instruction execution

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



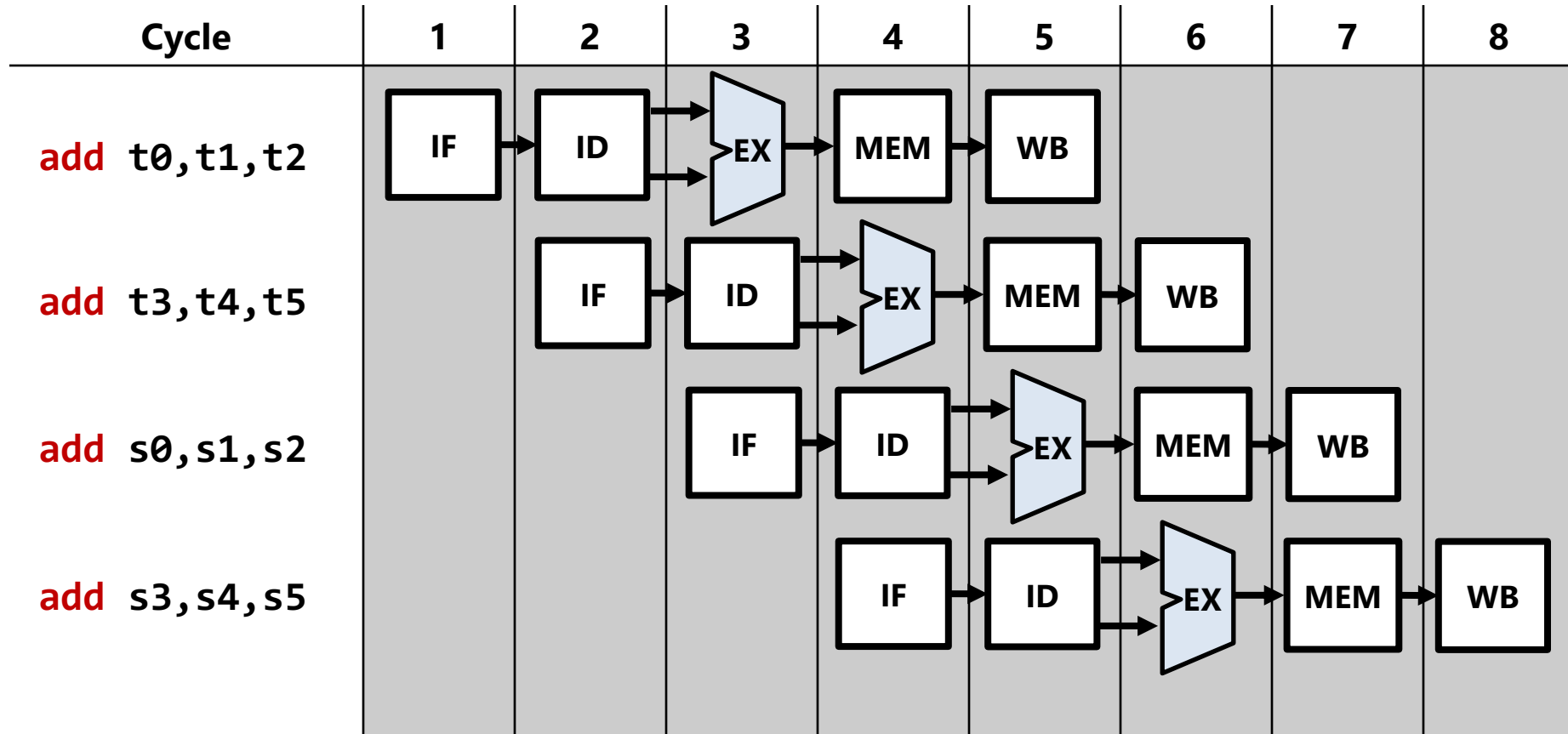
# Pipelined instruction execution

- Pipelining allows one instruction to be fetched each cycle!



# Pipelining Timeline

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



# A Pipelined Implementation is even Faster!

- Again each instruction takes different number of cycles to complete
  - **lw** takes 5 cycles: IF/ID/EX/MEM/WB
  - **add** takes 4 cycles: IF/ID/EX/WB
- If each stage takes *1 ns* each:
  - **lw** takes *5 ns* and **add** takes *4 ns*

Q) The average instruction execution time (given 100 instructions)?

A)  $(99 \text{ ns} + 5 \text{ ns}) / 100 = 1.04 \text{ ns}$

- Assuming last instruction is a **lw** (a 5-cycle instruction)
- 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

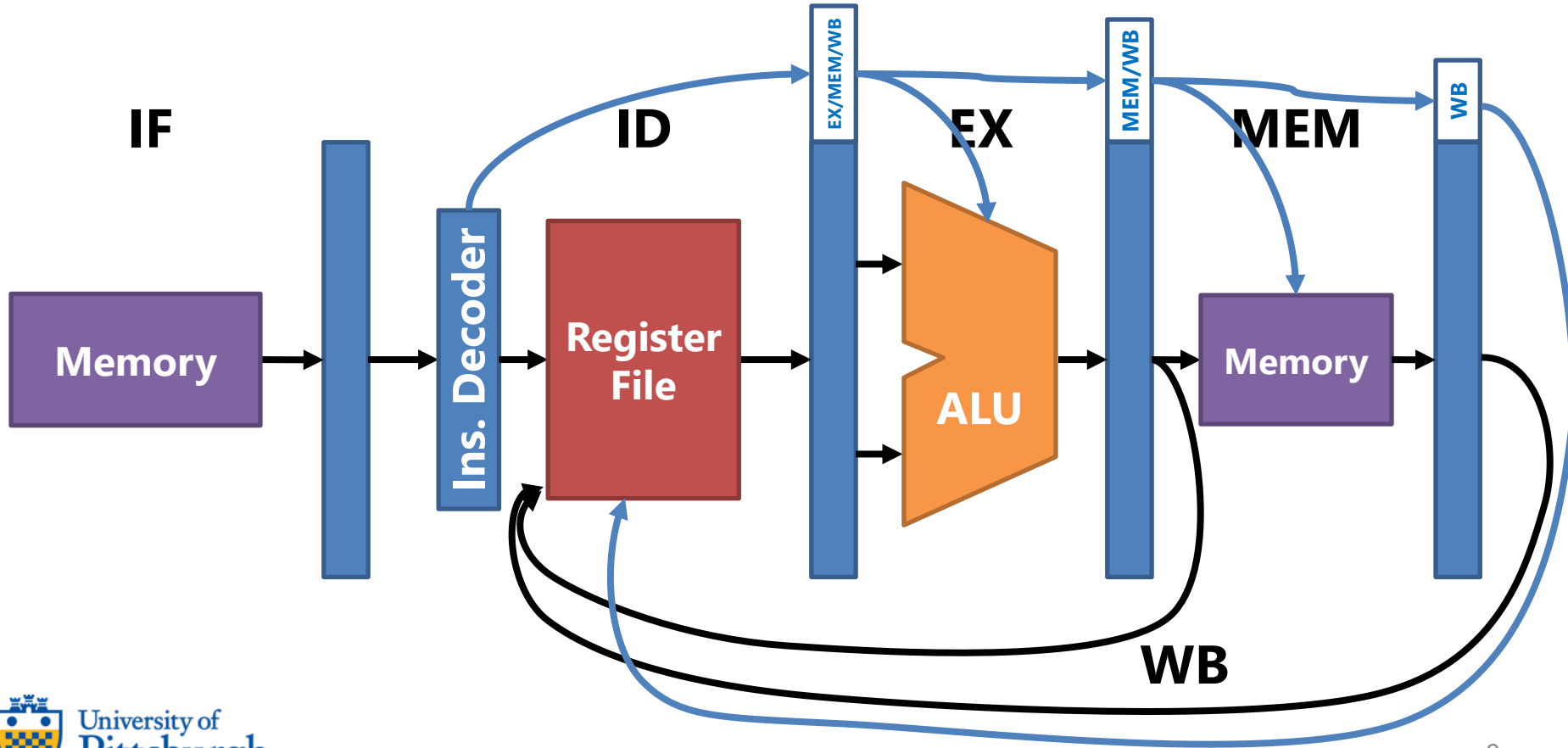
- Compared to single-cycle, pipelining improves clock cycle time
  - Or in other words CPU **clock frequency**
  - The deeper the pipeline, the higher the frequency will be

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



# 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



# Pipeline Hazards

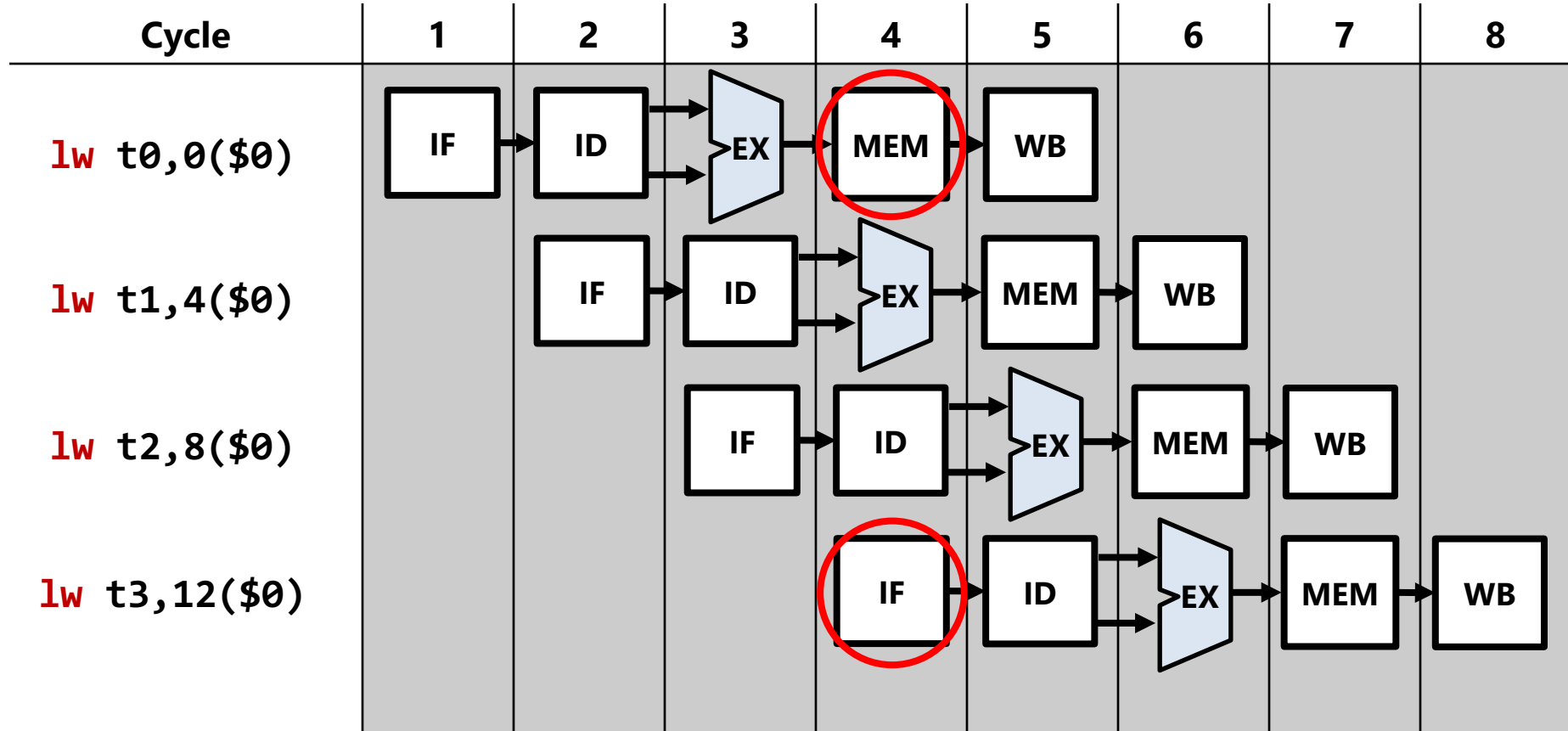


# Pipeline Hazards

- 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*

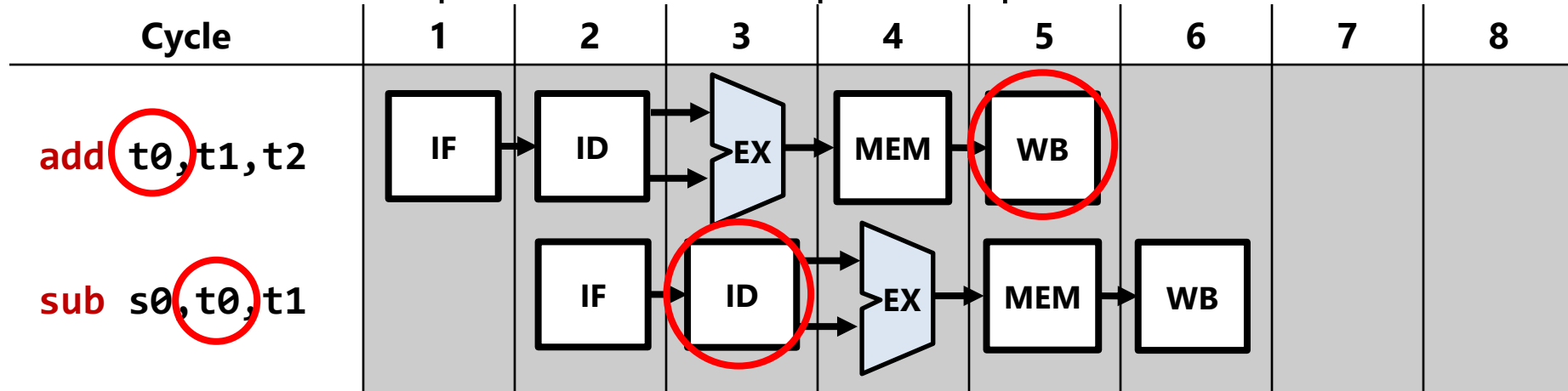
# Structural Hazards

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

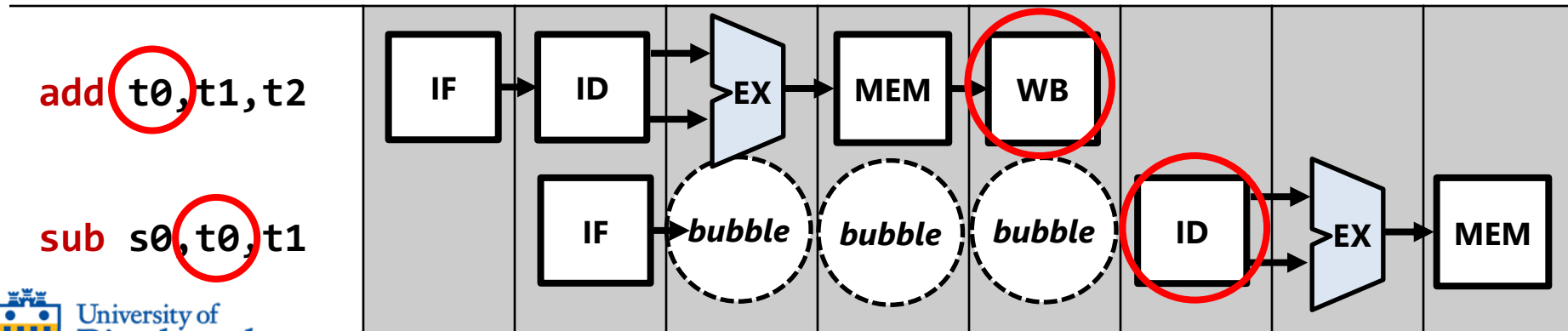


# Data Hazards

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

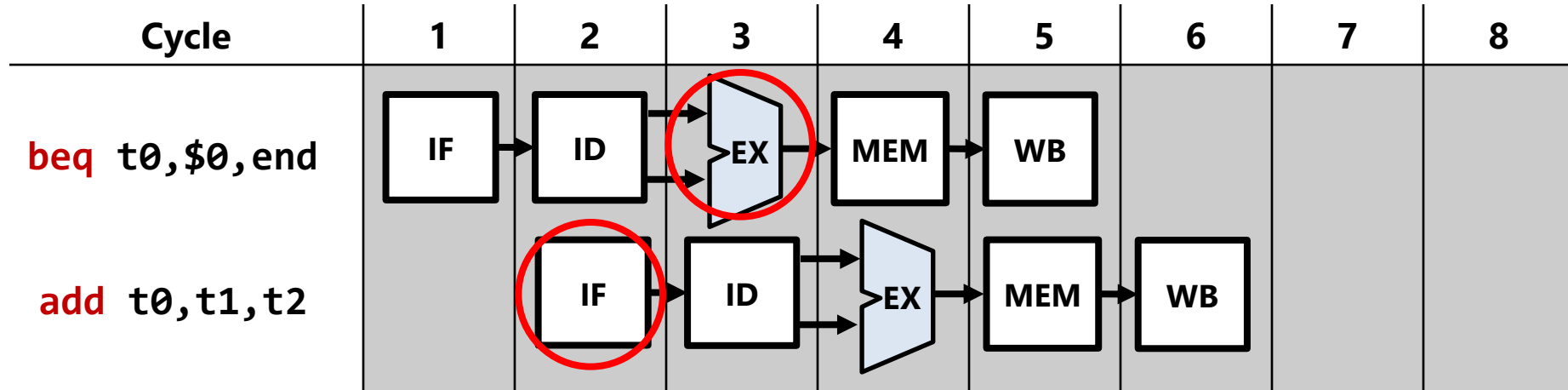


- sub** must wait until **add**'s WB phase is over before doing its ID phase

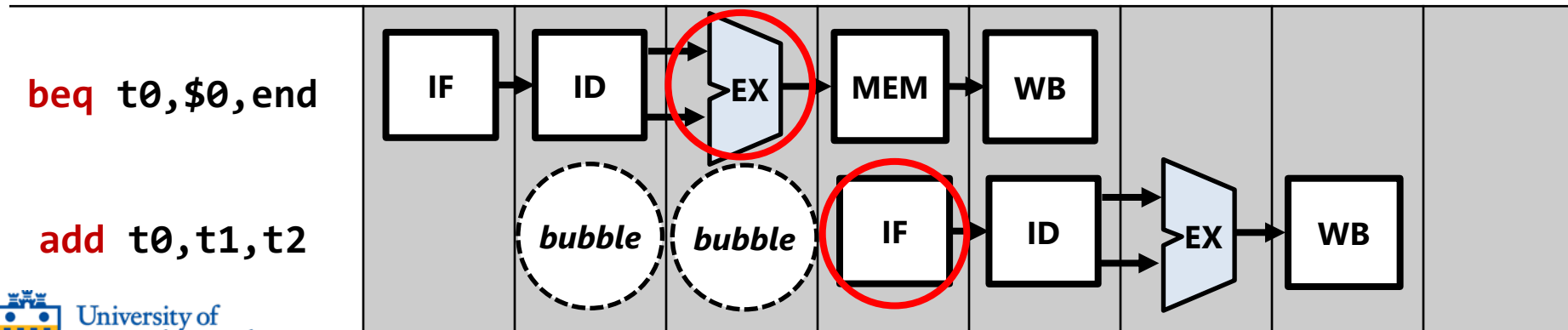


# Control Hazards

- You don't know the outcome of a conditional branch.



- add** must wait until **beq**'s EX phase is over before its IF phase

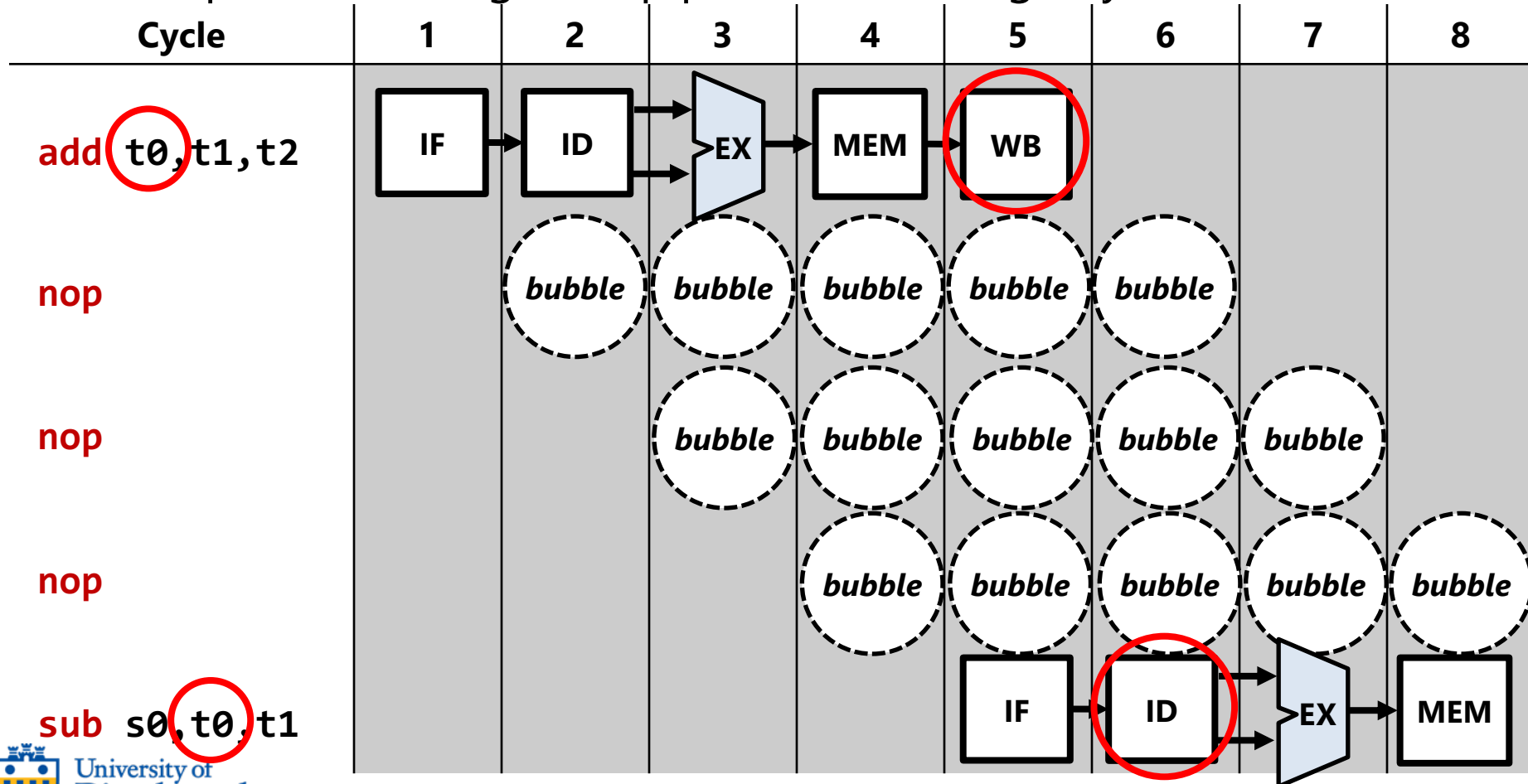


# Dealing with Hazards

- 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 a nop where compiler thinks a hazard would happen
  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

# Compiler avoiding a data hazard

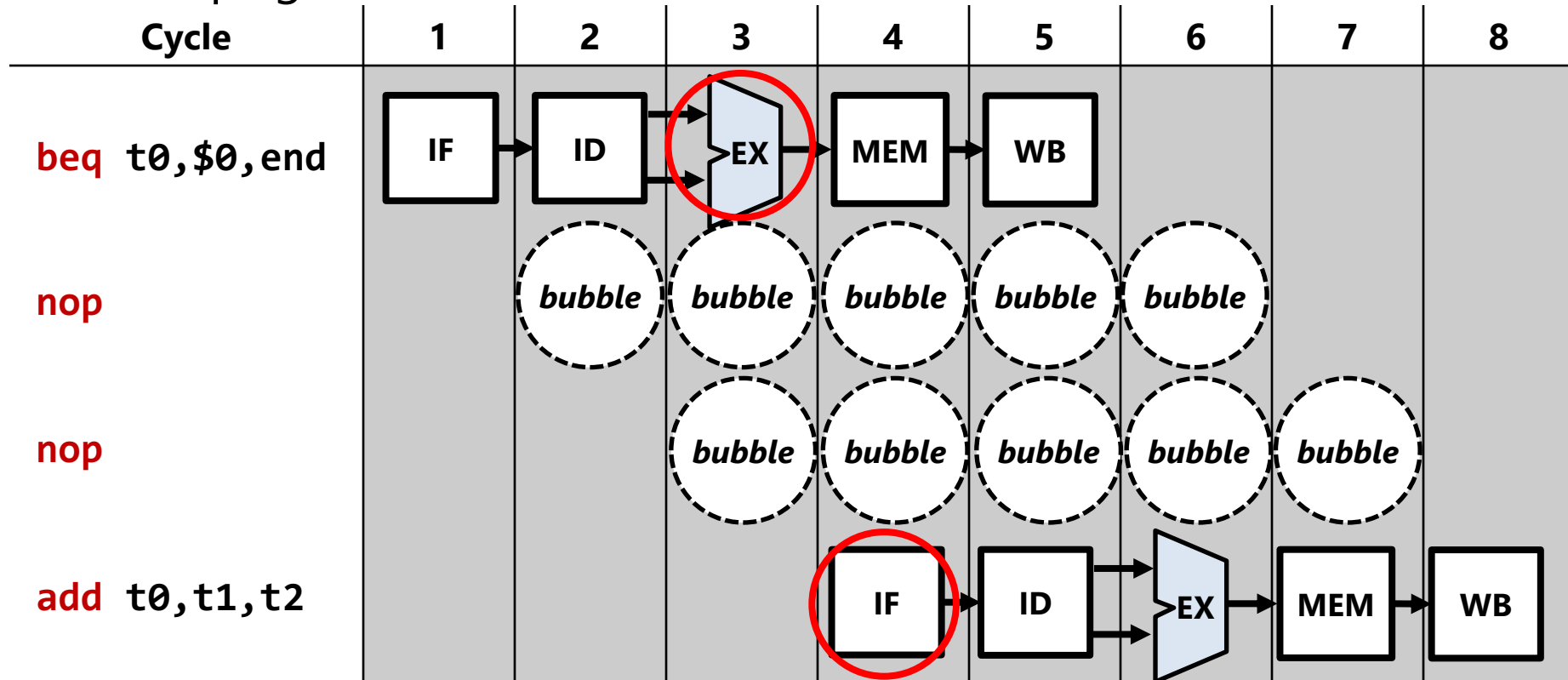
- The nops flow through the pipeline not doing any work



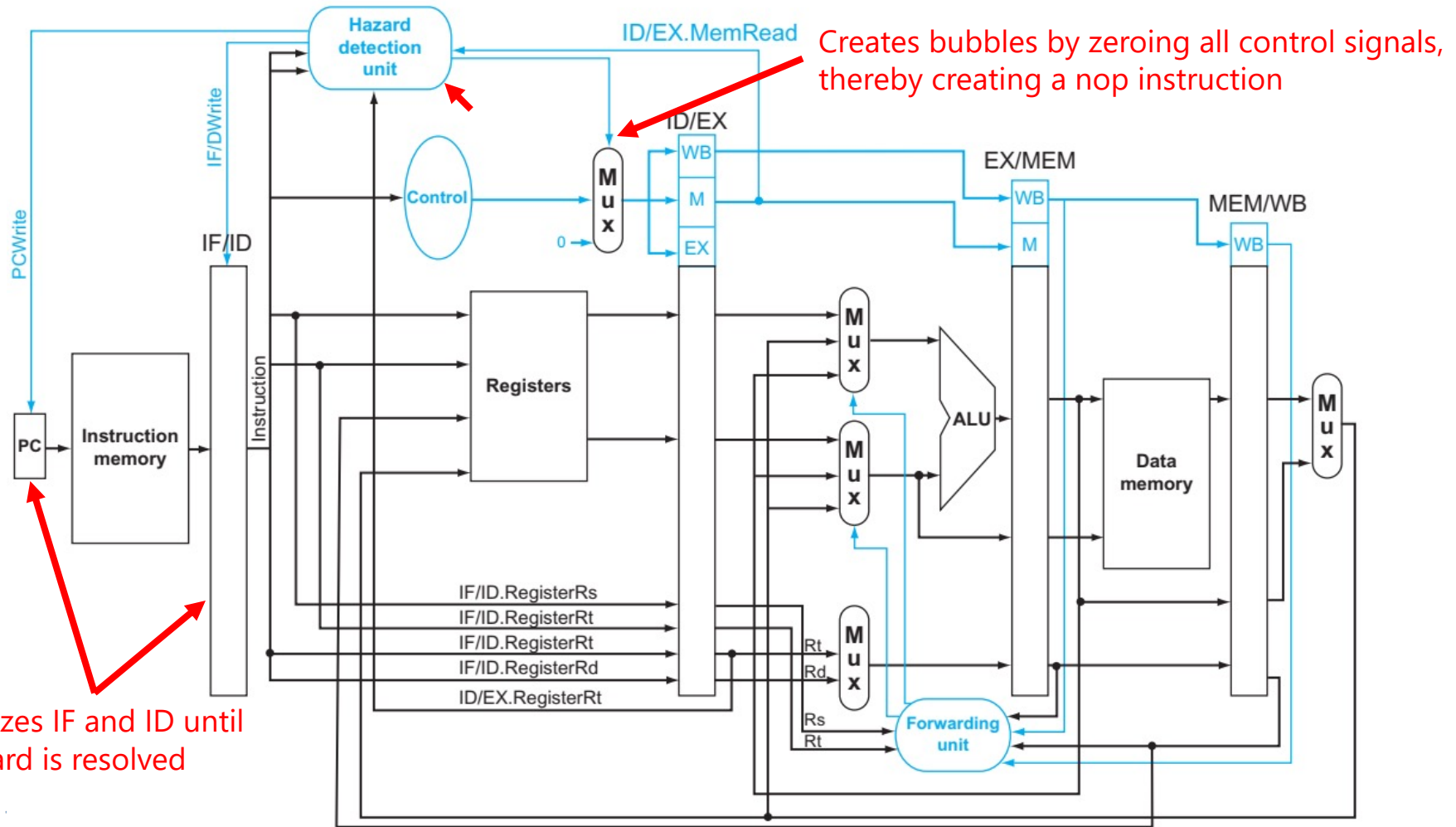


# Compiler avoiding a control hazard

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

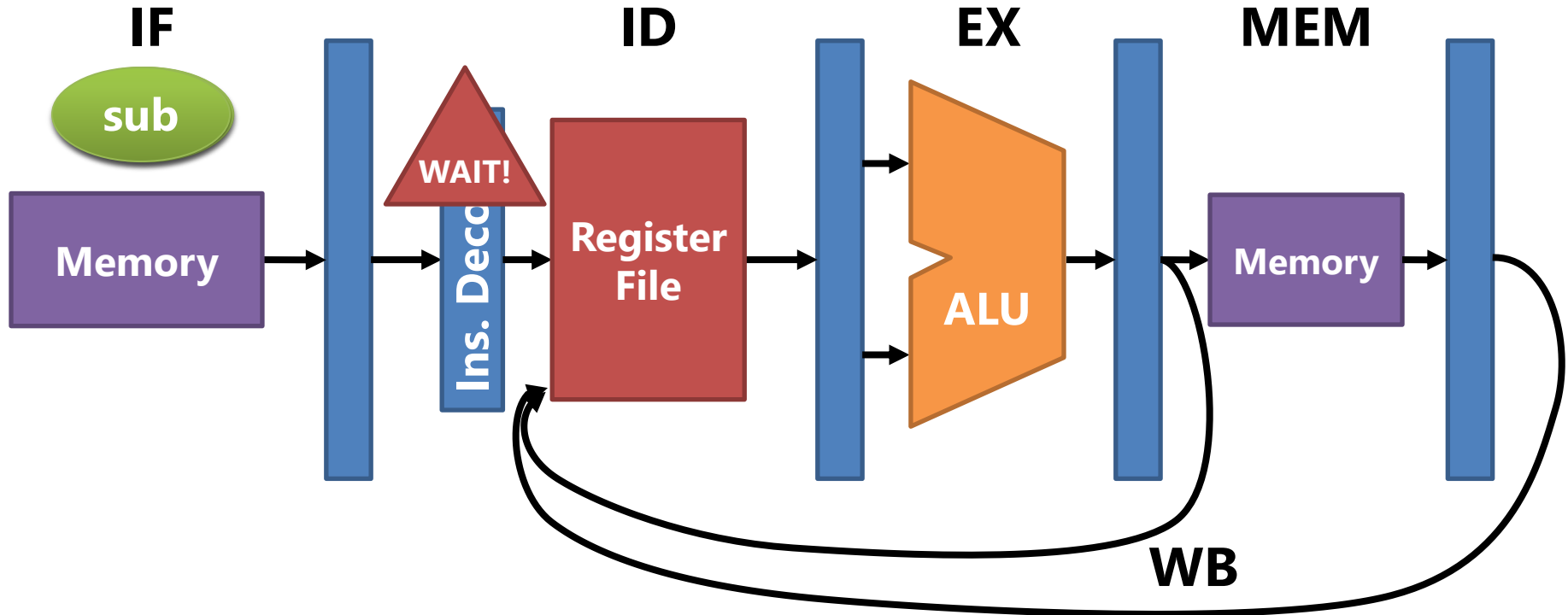


# Hazard Detection Unit



# 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

# Control Hazard Example

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

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

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

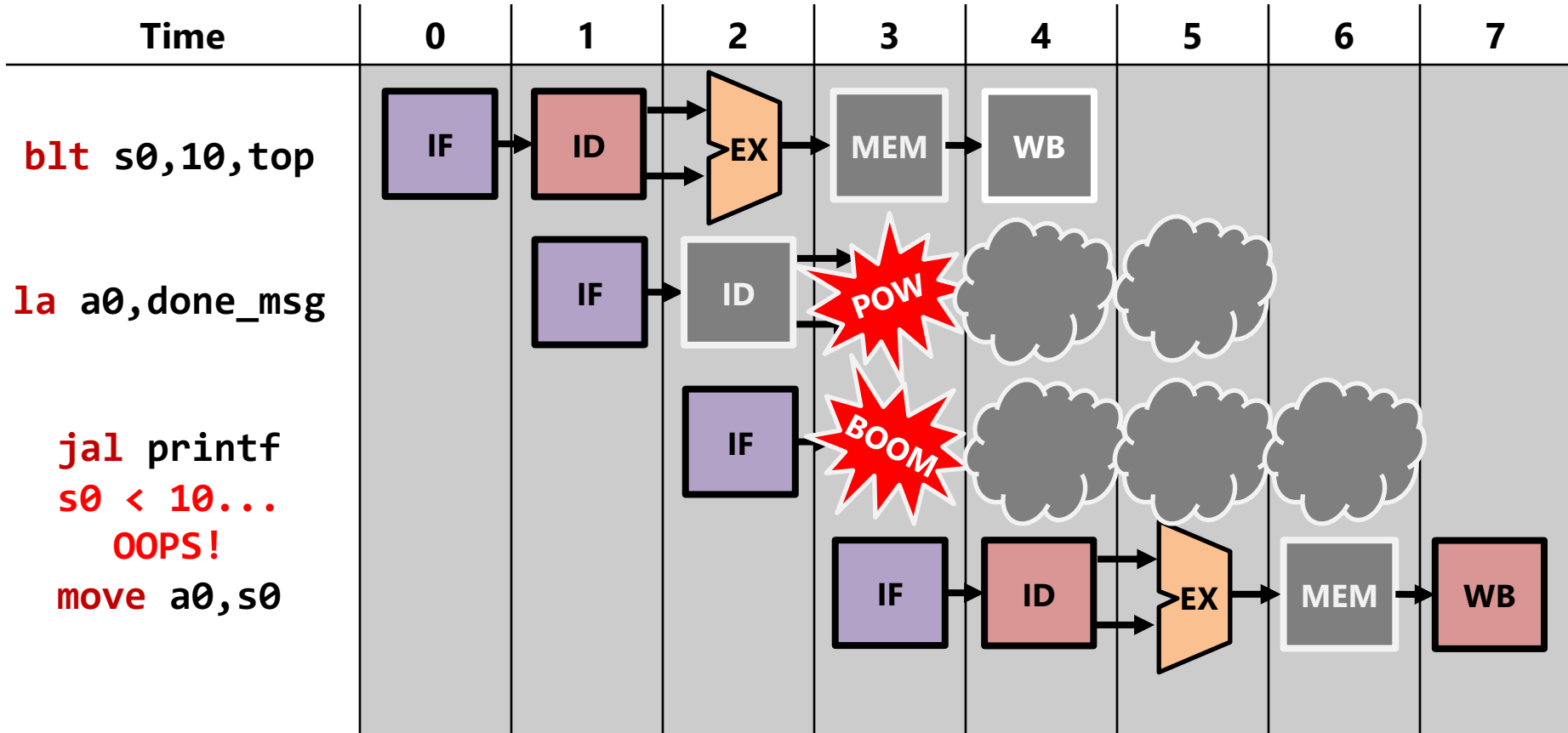
By the time `s0, 10`  
are compared at `blt`  
EX stage, the CPU  
would have already  
fetched `la` and `jal`!

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

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

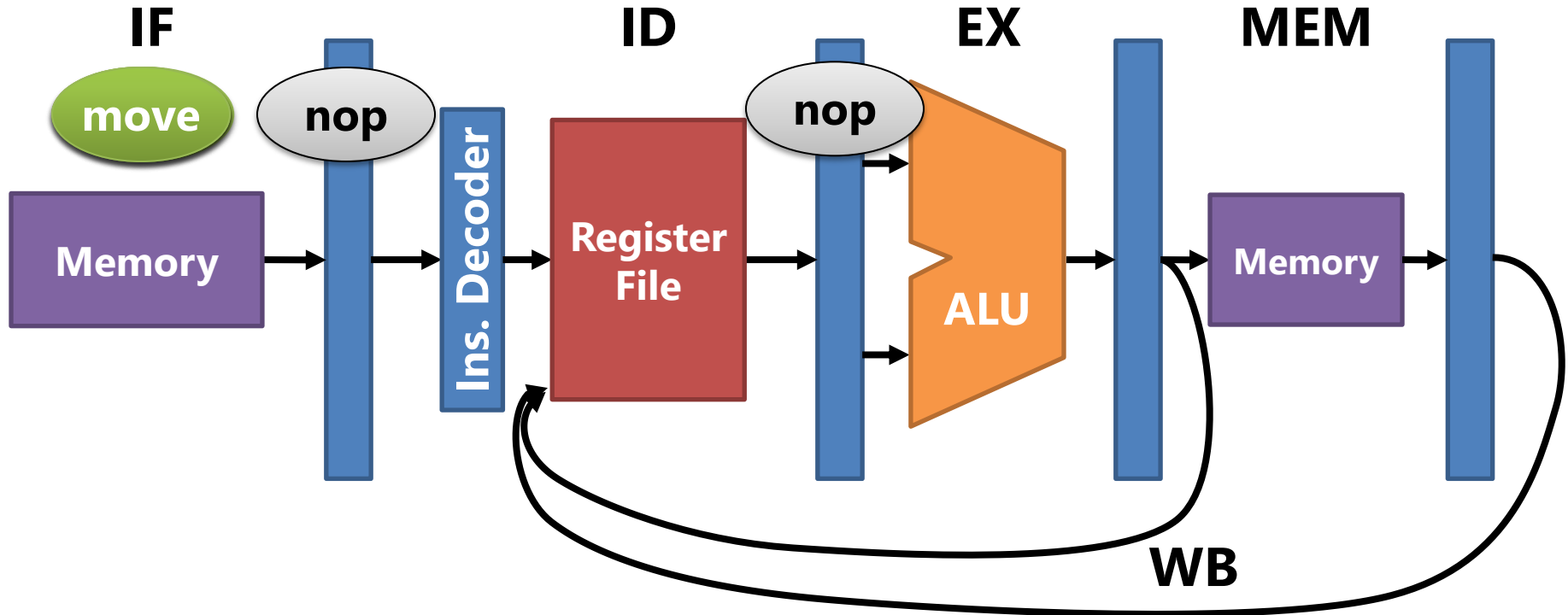
# What's a flush?

- 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	2	1 ns

- Pipelining increases **clock frequency** proportionate to depth
- But stalls increase **CPI** (cycles per instruction)
  - If stalls prevent new instructions from being fetched half the time, the CPU will have a CPI of 2 → Only 2.5X speed up (instead of 5X)
- We'd like to avoid this penalty if possible!

# Compiler nops vs. CPU Hazard Detection Unit

- Limitations of compiler nops
  - Compiler must make assumptions about processor design
    - That means processor design must become part of ISA
    - What if that design is no longer ideal in future generations?
  - Length of MEM stage is very 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**?
    - MEM isn't uniform really and sometimes hundreds of cycles
- But compiler nops is very energy-efficient
  - Hazard Detection Unit can be power hungry
    - A lot of long wires controlling remote parts of the CPU
    - Adds to the **Power Wall** problem
  - Compiler scheduling via nops removes need for HDU