

# CPU Pipelining

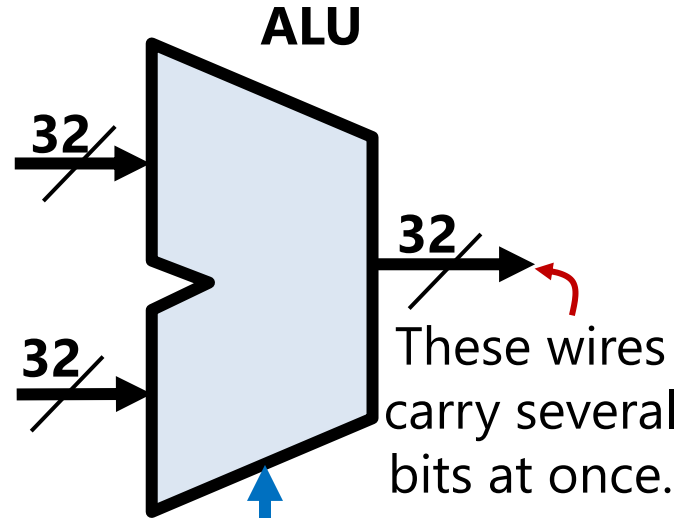
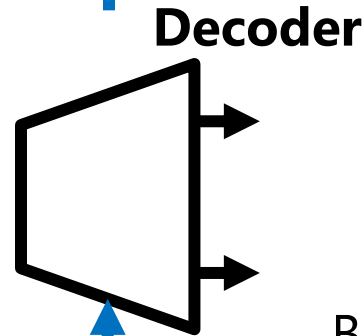
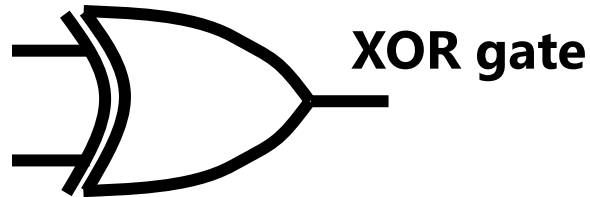
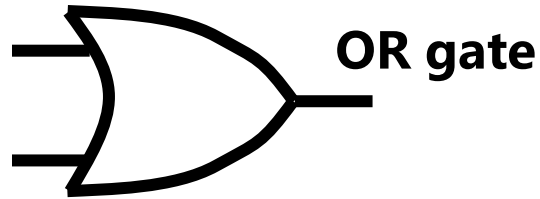
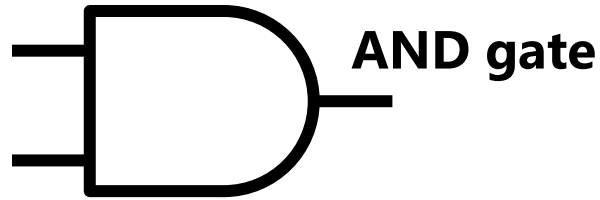
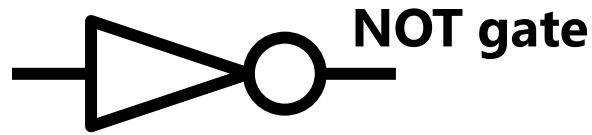
CS/COE 1541 (Fall 2020)  
Wonsun Ahn

# Clocking Review

Stuff you learned in CS 447

# Logic components

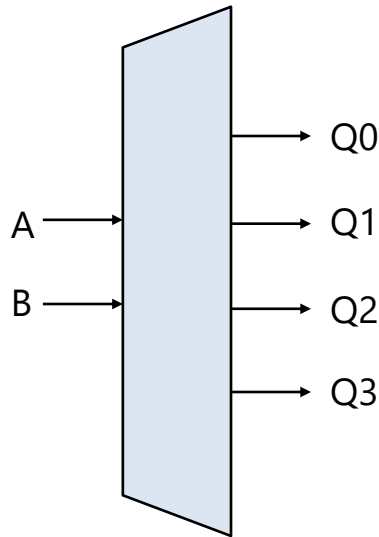
- Do you remember what all these do?



Blue wires are control signals.

# Uses of a Decoder

- Translates a set of input signals to a bunch of output signals.
  - E.g. a binary decoder:



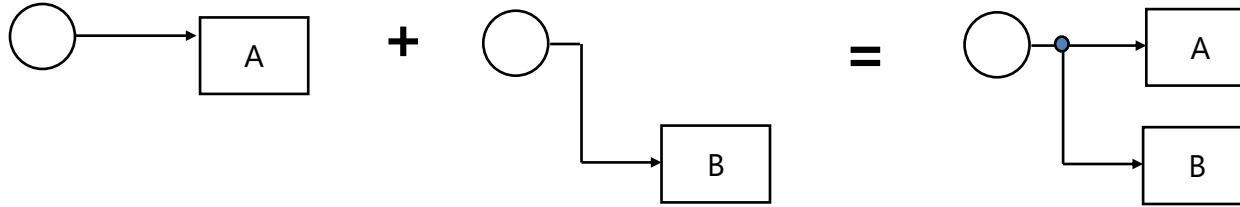
**Truth Table for Decoder**

A	B	Q0	Q1	Q2	Q3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

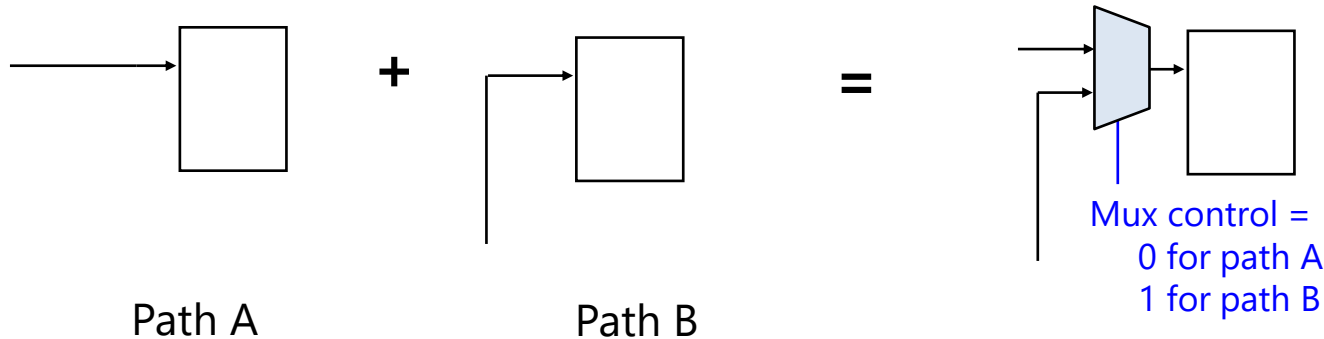
- You can come up with any truth table and make a decoder for it!

# Uses of a Multiplexer

- No problem in fanning out one signal to two points

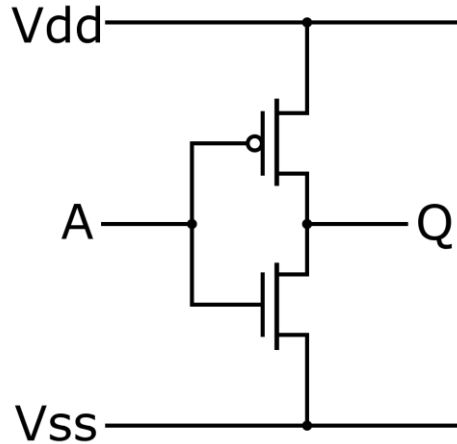


- Cannot connect two signals to one point
  - Must use a multiplexer to *select* between the two

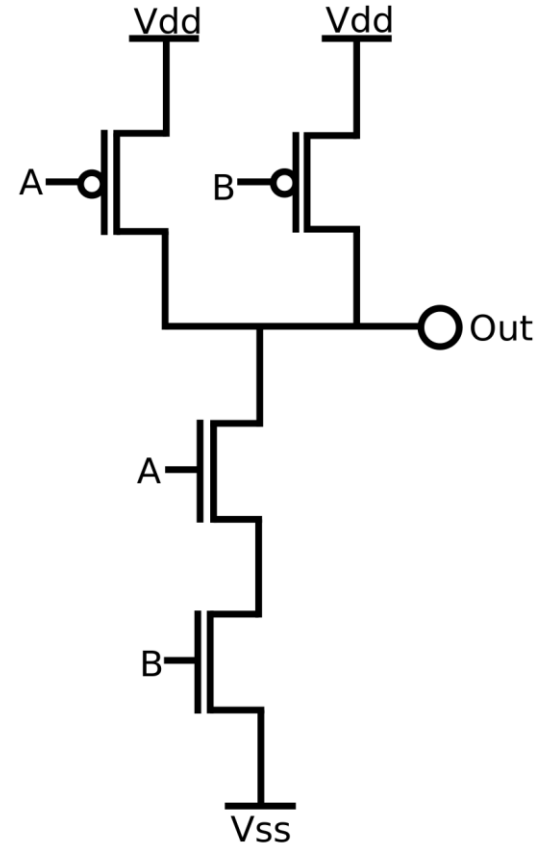


# Gates are made of transistors (of course)

- NOT gate

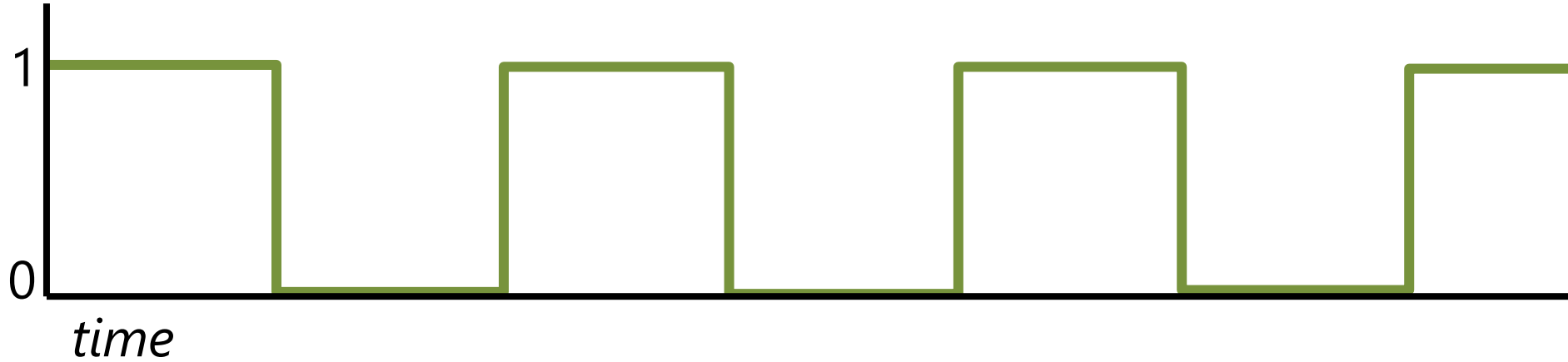


- NAND gate



# The clock signal

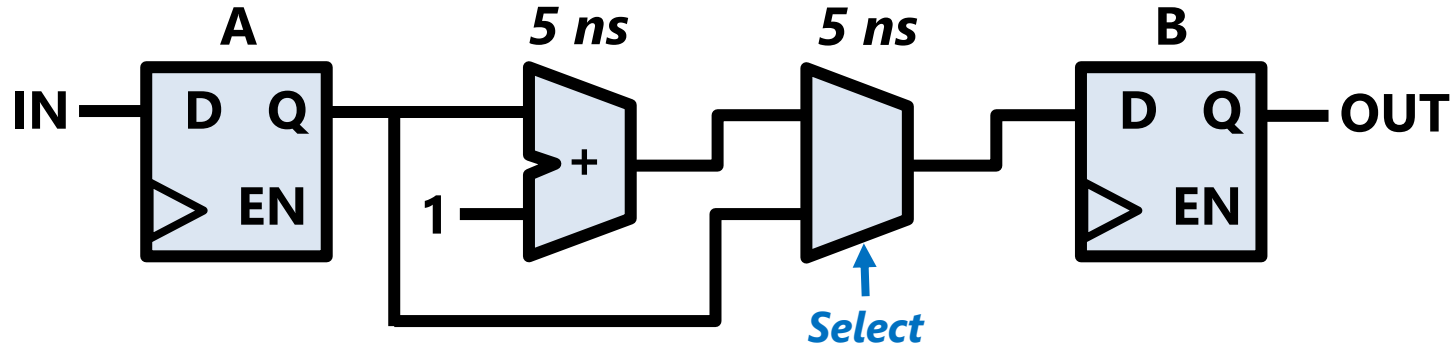
- The clock is a signal that alternates regularly between 0 and 1:



- Bits are latched on to registers and flip-flops on rising edges
- In between rising edges, bits propagate through the logic circuit
  - Composed of ALUs, muxes, decoders, etc.
  - **Propagation delay**: amount of time it takes from input to output

# Critical Path

- **Critical path:** path in a circuit that has longest propagation delay
  - Determines the overall clock speed.



- The ALU and the multiplexer both have a 5 ns delay
- How fast can we clock this circuit?
  - Is it  $1 / 5 \text{ ns}$  ( $5 \times 10^{-9} \text{ s}$ ) = 200 MHz?
  - Or is it  $1 / 10 \text{ ns}$  ( $10 \times 10^{-9} \text{ s}$ ) = 100 MHz? ✓



# MIPS Review

Stuff you learned in CS 447

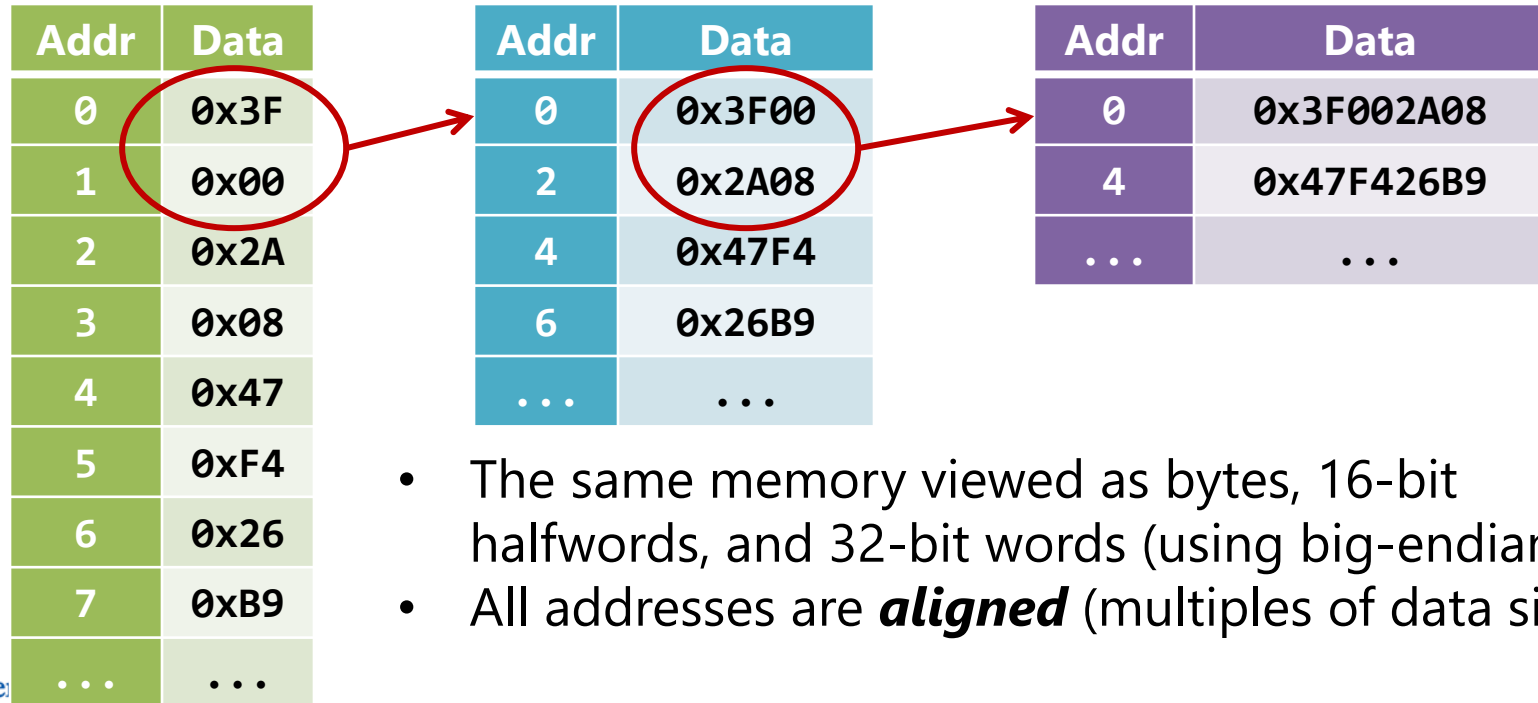
# The MIPS ISA - Registers

- MIPS has 32 32-bit registers, with the following usage conventions
  - But really, all are general purpose registers (nothing special about them)

Name	Register number	Usage
<b>\$zero</b>	<b>0</b>	the constant value 0 (can't be written)
<b>\$at</b>	<b>1</b>	assembler temporary
<b>\$v0-\$v1</b>	<b>2-3</b>	values for results and expression evaluation
<b>\$a0-\$a3</b>	<b>4-7</b>	function arguments
<b>\$t0-\$t7</b>	<b>8-15</b>	unsaved temporaries
<b>\$s0-\$s7</b>	<b>16-23</b>	saved temporaries (like program variables)
<b>\$t8-\$t9</b>	<b>24-25</b>	more unsaved temporaries
<b>\$k0-\$k1</b>	<b>26-27</b>	reserved for OS kernel
<b>\$gp</b>	<b>28</b>	global pointer
<b>\$sp</b>	<b>29</b>	stack pointer
<b>\$fp</b>	<b>30</b>	frame pointer
<b>\$ra</b>	<b>31</b>	return address

# The MIPS ISA - Memory

- MIPS is a **RISC (reduced instruction set computer)** architecture
- It is also a **load-store** architecture
  - **All** memory accesses performed by load and store instructions
- Memory is a giant array of  $2^{32}$  bytes



- The same memory viewed as bytes, 16-bit halfwords, and 32-bit words (using big-endian)
- All addresses are **aligned** (multiples of data size)

# The MIPS ISA - Memory

- Loads move data *from* memory *into* the registers.

**lw** **\$t0**, **8(\$s4)**

This is the address, and it means "the value of \$s4 + 8."

t0	0x0000BEEF
s4	0x00000004

**Registers**

- Stores move data *from* the registers *into* memory.

**sw** **\$t0**, **12(\$s4)**

\$t0 is the SOURCE!

lw

sw

$\$s4 + 8$

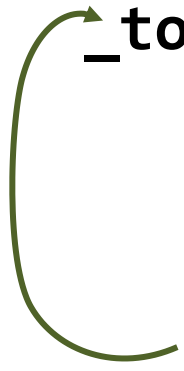
$\$s4 + 12$

0	0x3F002A08
4	0x47F426B9
8	0x00000000
12	0x0000BEEF
16	0x0000BEEF
...	...

**Memory**

# The MIPS ISA – Flow control

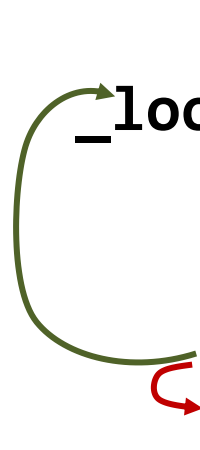
- Jump and branch instructions change the flow of execution.



```
_top:
# ....
# Lots o' code
# ....
j _top
```

**j** : jumps *unconditionally*

- jumps to **\_top**



```
li    $s0, 10
_loop:
# ....
addi  $s0, $s0, -1
bne   $s0, $zero, _loop
jr    $ra
```

**bne** : jumps *conditionally*

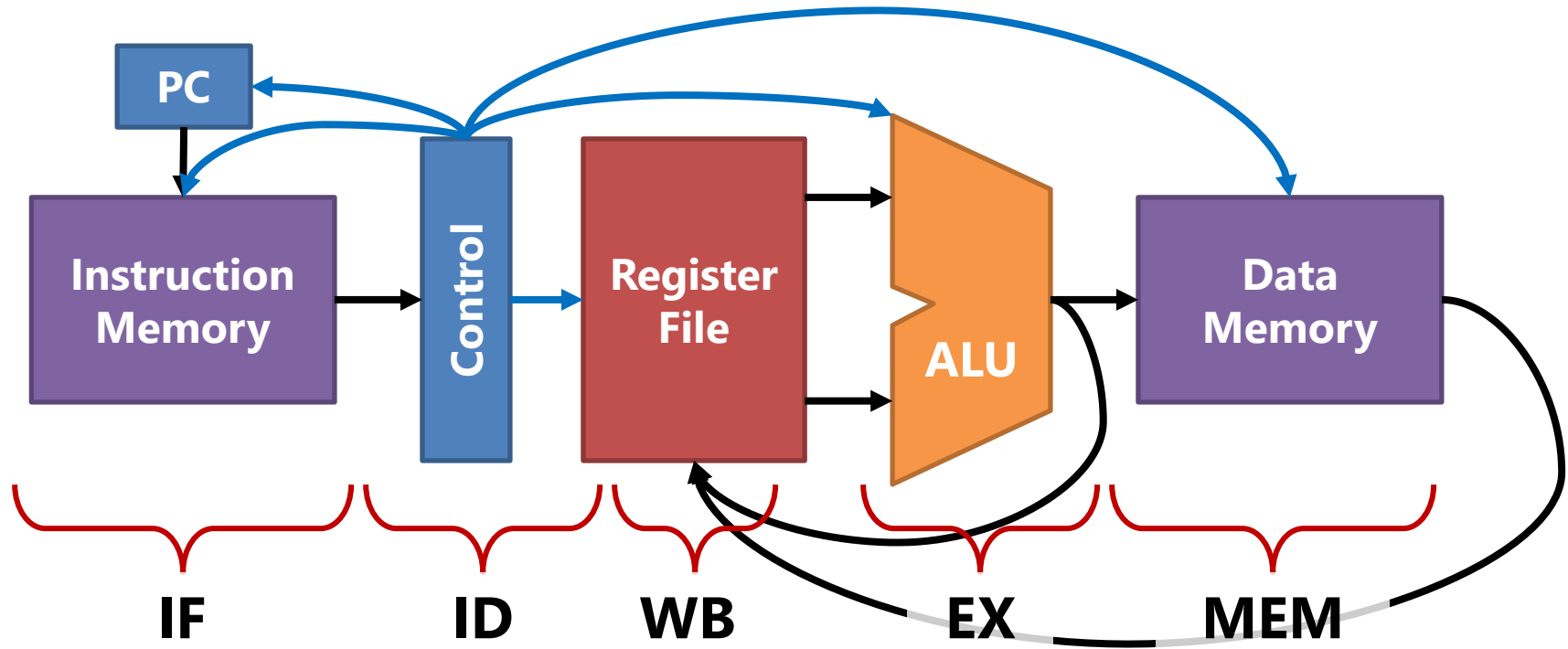
If **\$s0** != **\$zero**, jumps to **\_loop**

If **\$s0** == **\$zero**, continues to **jr \$ra**

# Phases of instruction execution

- In most architectures, there are five phases:
  1. **IF** (Instruction Fetch) – get next instruction from memory
  2. **ID** (Instruction Decode) – figure out what instruction it is
  3. **EX** (Execute – ALU) – do any arithmetic
  4. **MEM** (Memory) – read or write data from/to memory
  5. **WB** (Register Writeback) – write any results to the registers
- Sometimes these phases are chopped into smaller stages

# A simple single-cycle implementation



- An instruction goes through IF/ID/EX/MEM/WB in one cycle

# "Minimal MIPS"

---

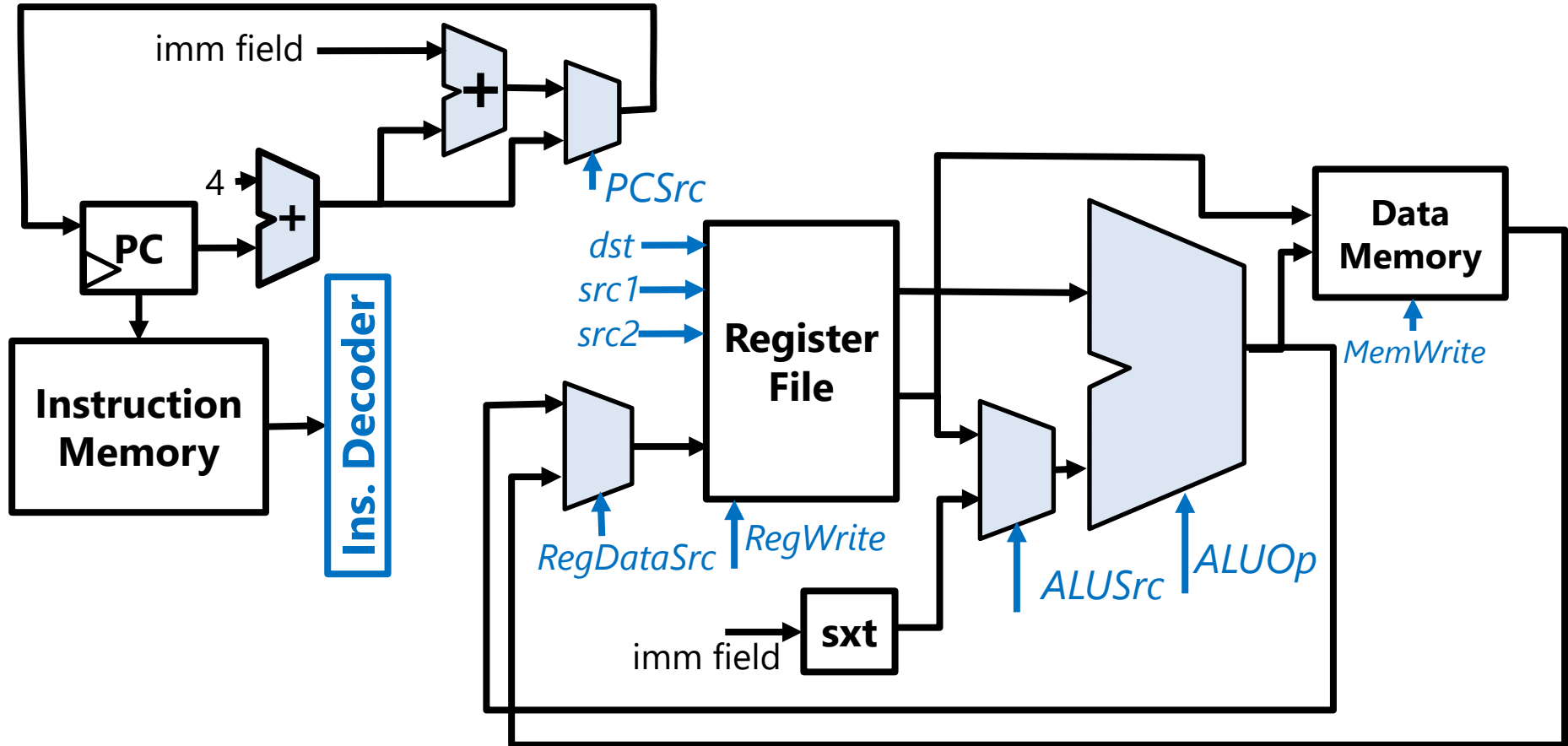


# It's a "subset" of MIPS

- For pedagogical (teaching) purposes
- Contains only a minimal number of instructions:
  - **lw, sw, add, sub, and, or, slt, beq, and j**
  - Other instructions in MIPS are variations on these anyway
- Let's review the Minimal MIPS CPU focusing on the control signals
  - Again, these control signals are decoded from the instruction

# The Minimal MIPS single-cycle CPU

- A more detailed view of the 5-phase implementation

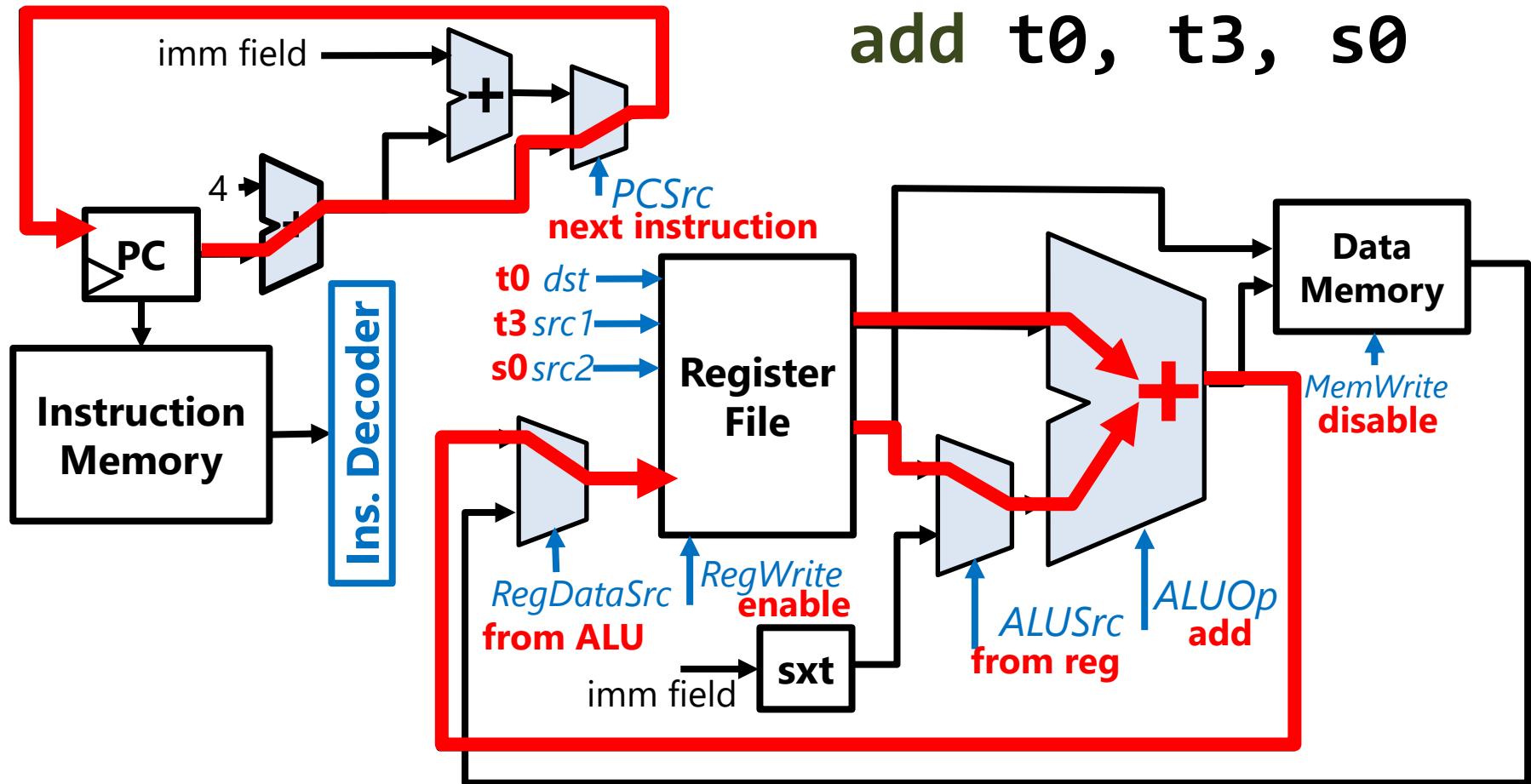


# Control signals

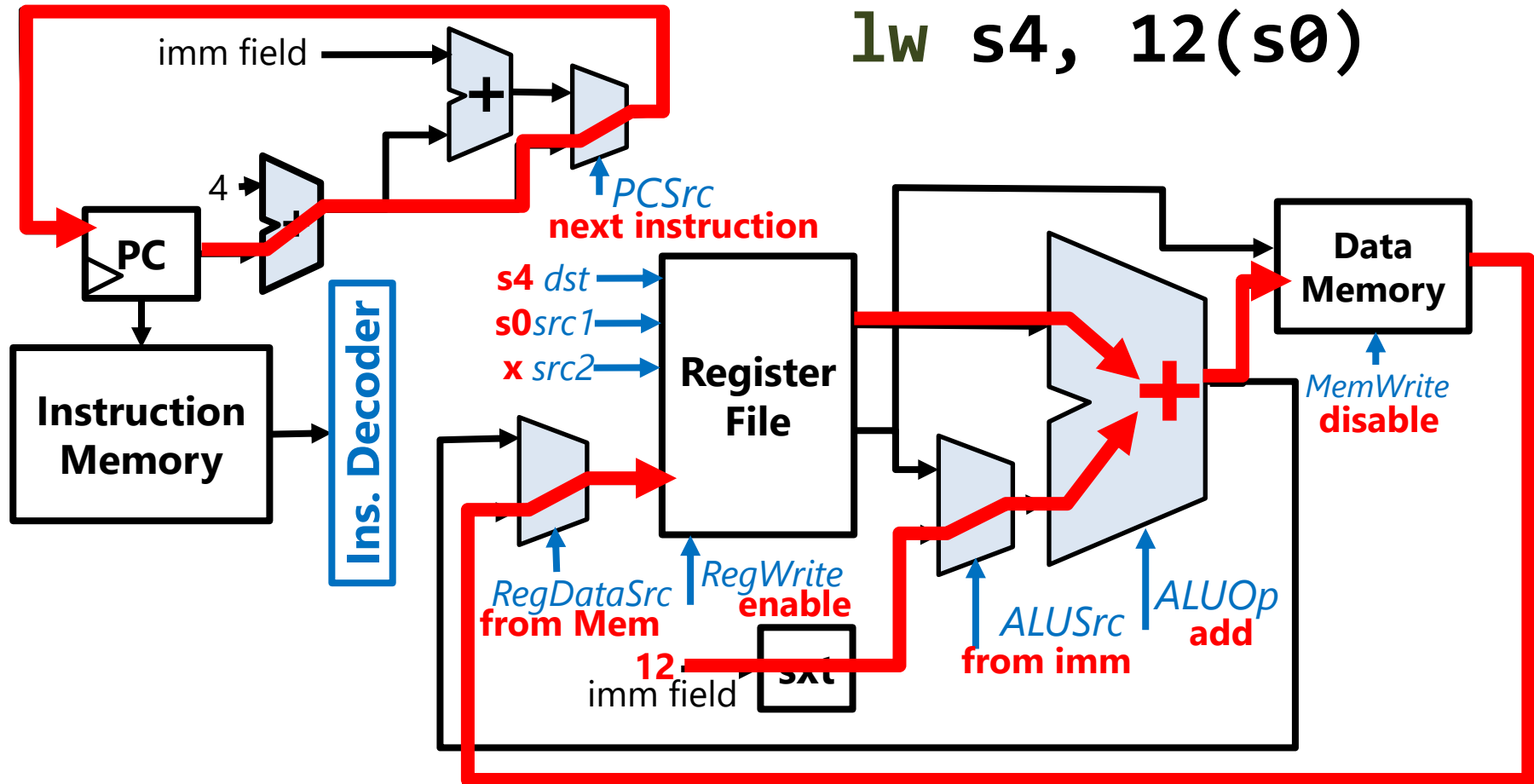
- Registers
  - **RegDataSrc**: controls source of a register write (ALU / memory)
  - **RegWrite**: enables a write to the register file
  - **src1, src2, dst**: the register number for each respective operand
- ALU
  - **ALUSrc**: whether second operand of ALU is a register / immediate
  - **ALUOp**: controls what the ALU will do (add, sub, and, or etc)
- Memory
  - **MemWrite**: enables a write to data memory
- PC
  - **PCSrc**: controls source of next PC ( $PC + 4$  /  $PC + 4 + imm$ )

→ All these signals are decoded from the instruction!

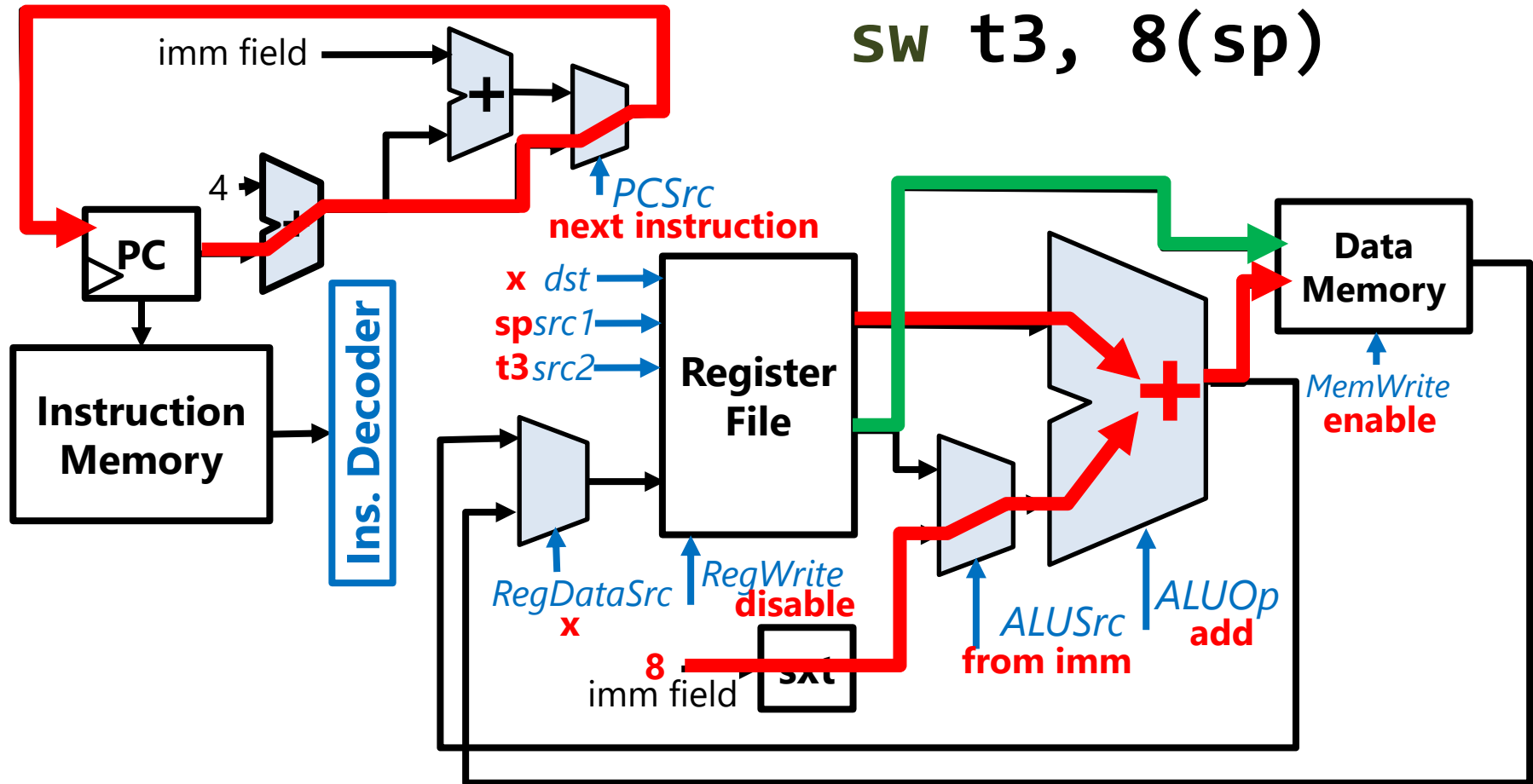
# How an **add/sub/and/or/slt** work



# How an lw works

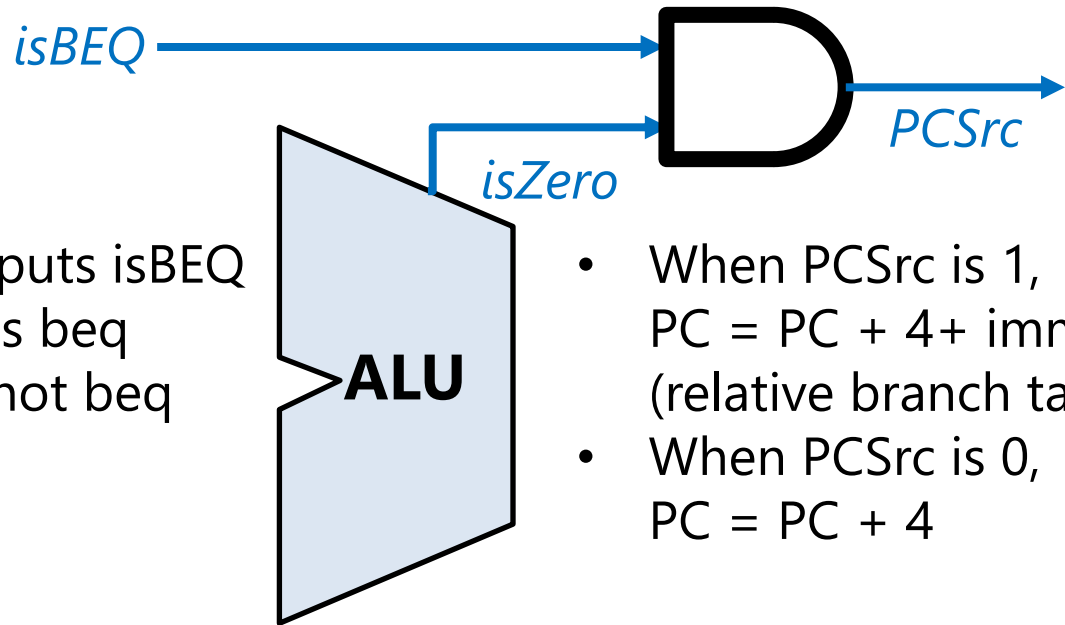


# How an **sw** works



# What about **beq**?

- Compares numbers by subtracting and see if result is 0
  - If result is 0, we set PCSrc to use the branch target.
  - Otherwise, we set PCSrc to  $PC + 4$ .



- Instruction decoder outputs *isBEQ*
  - 1: When instruction is **beq**
  - 0: When instruction not **beq**

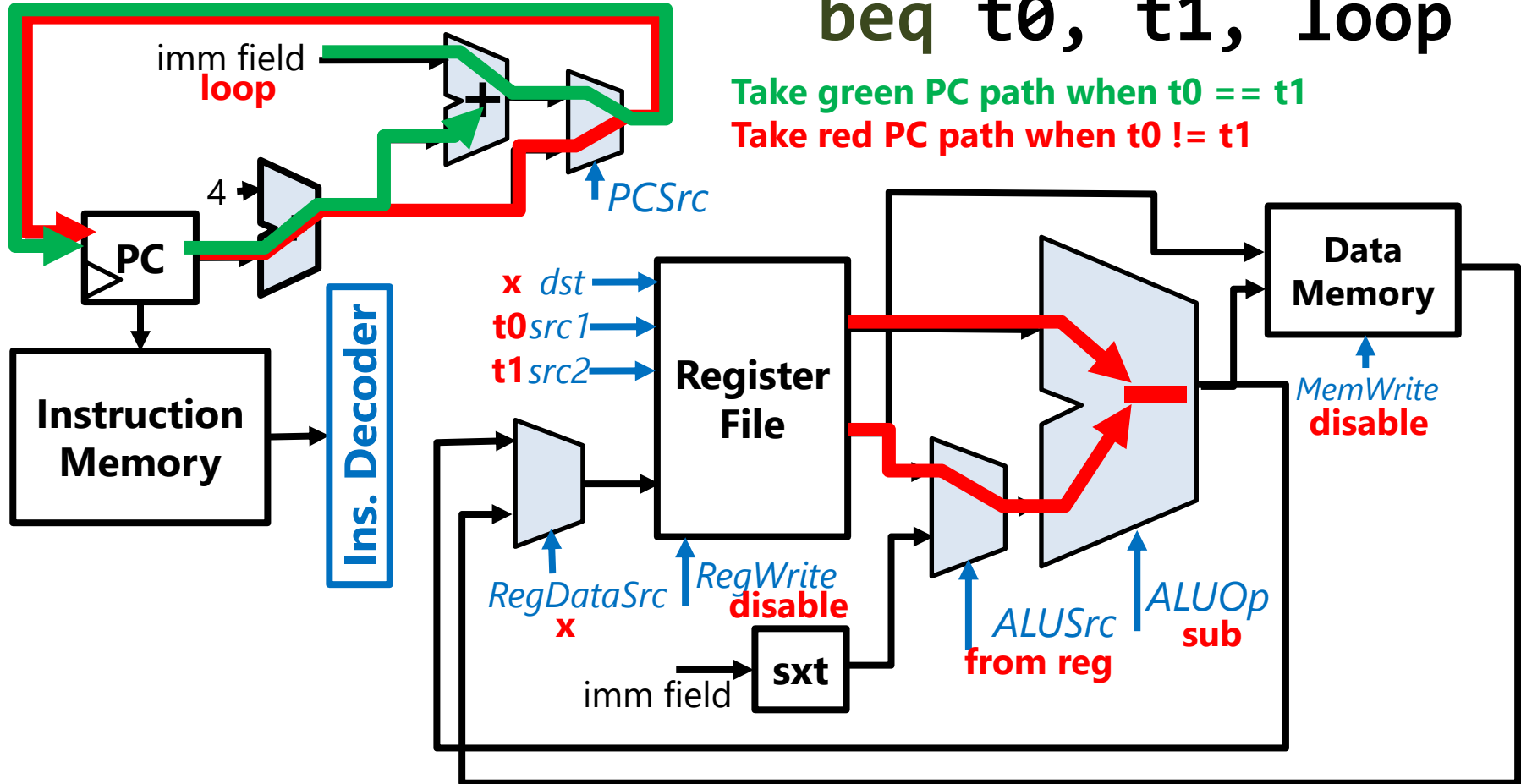
- When *PCSrc* is 1,  
 $PC = PC + 4 + \text{imm}$   
(relative branch target)
- When *PCSrc* is 0,  
 $PC = PC + 4$

# How a **beq** works

**beq t0, t1, loop**

Take green PC path when  $t0 == t1$

Take red PC path when  $t0 \neq t1$

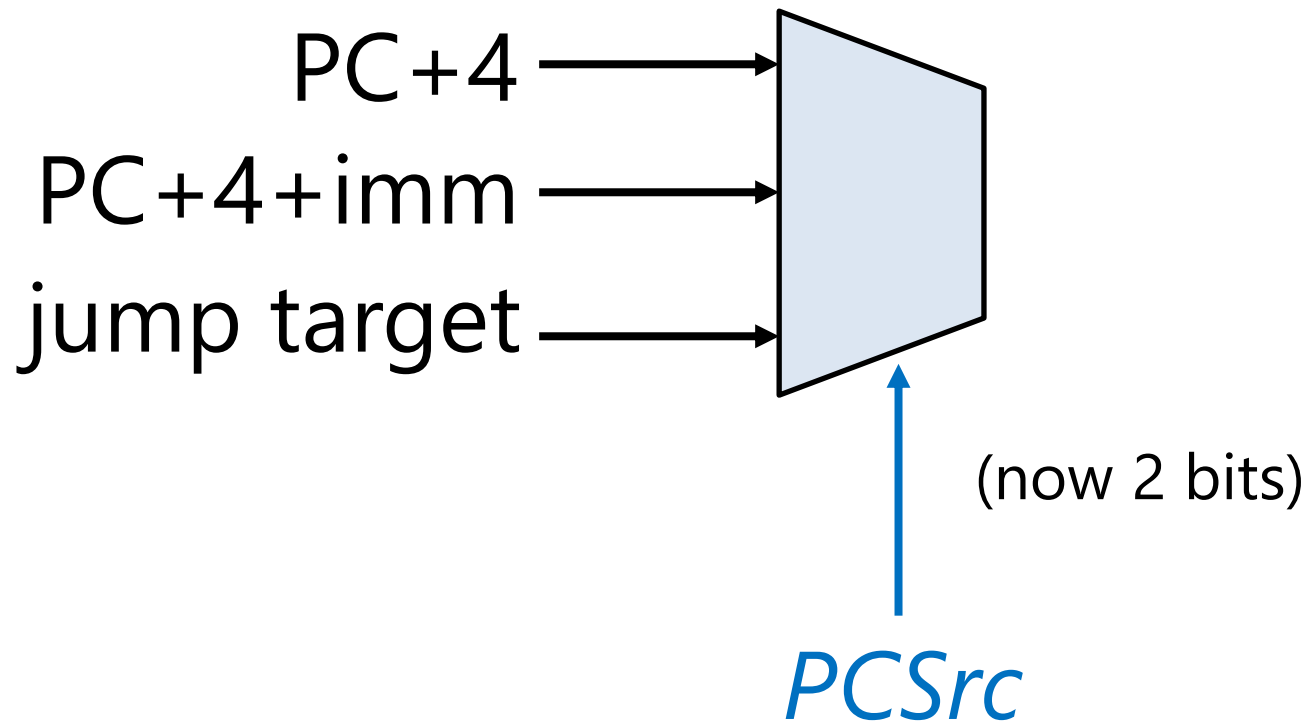




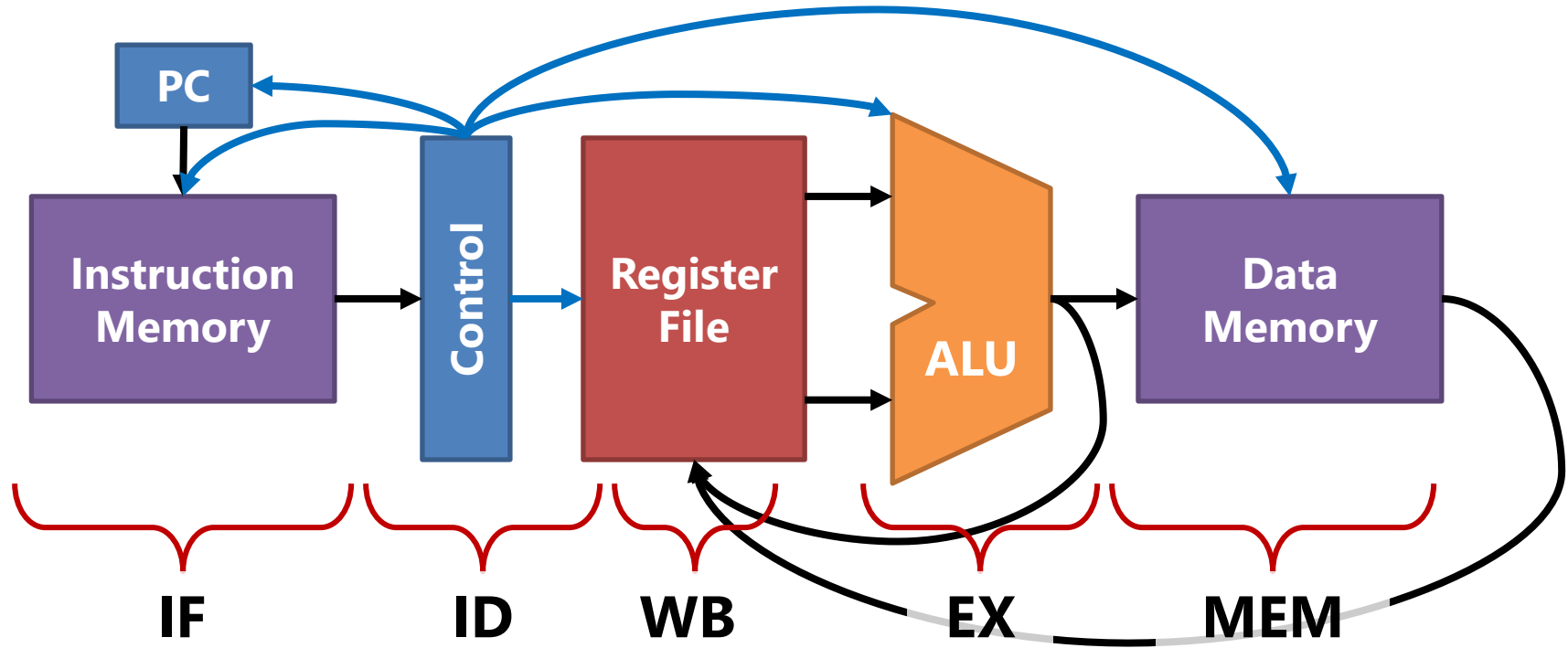
# What about **j**?

- We have to add another input to the PCSrc mux.

**j**    **top**



# A Single-cycle Implementation is not Optimal



- Why? Since the **longest** critical path must be chosen for cycle time
  - And there is a wide variation among different instructions

# A Single-cycle Implementation is not Optimal

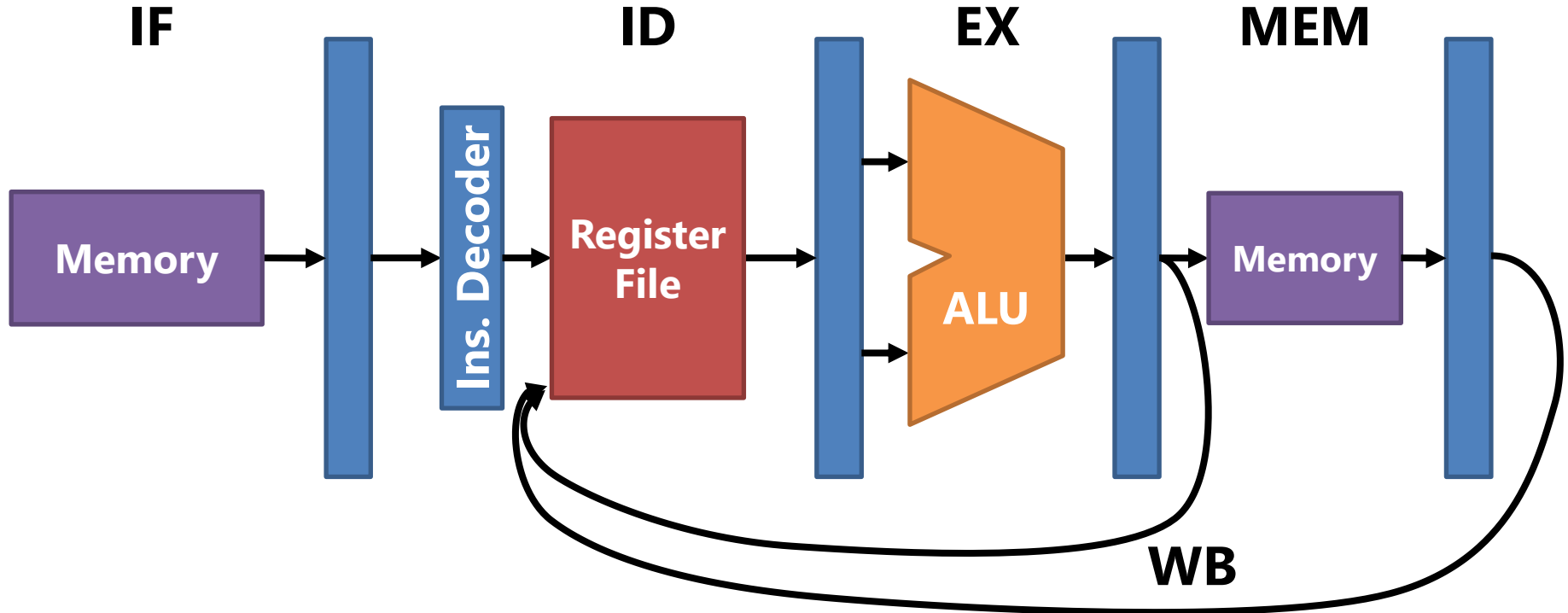
- In our CPU, the **lw** instruction has the longest critical path
  - Must go through all 5 stages: IF/ID/EX/MEM/WB
  - Whereas **add** goes through just 4 stages: IF/ID/EX/WB
- If each phase takes *1 ns* each, cycle time must be *5 ns*:
  - Because it needs to be able to handle **lw**, which takes *5 ns*
  - **add** also takes *5 ns* when it could have been done in *4 ns*

Q) If **lw** is 1% and **add** is 99% of instruction mix,  
what is the average instruction execution time?

A) Still *5 ns*! Even if **lw** is only 1% of instructions!

# A Multi-cycle Implementation

- It takes one cycle for each phase through the use of internal latches



# A Multi-cycle Implementation is Faster!

- Now 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 phase takes  $1\text{ ns}$  as before:
  - **lw** takes  $5\text{ ns}$  and **add** takes  $4\text{ ns}$

Q) If **lw** is 1% and **add** is 99% of instruction mix, what is the average instruction execution time?

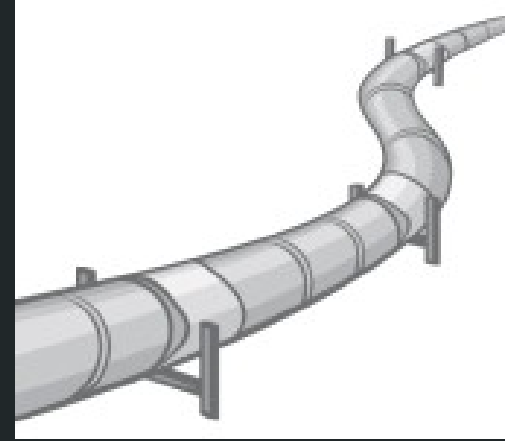
A)  $0.01 * 5\text{ ns} + 0.99 * 4\text{ ns} = 4.01\text{ ns}$  (25% faster than single cycle)

*\* Caveat: delay due to the added latches not shown, but net win*

# And we can do even better!

- Did you notice?
  - When an instruction is on a particular phase (e.g. IF) ...
  - ... other phases (ID/EX/MEM/WB) are not doing any work!
- Our CPU is getting chronically ***underutilized***!
  - If CPU is a factory, 80% (4/5) of the workers are idling!
- Car factories create an assembly line to solve this problem
  - No need to wait until a car is finished before starting on next one
  - Our CPU is going to use a ***pipeline*** (similar concept)

# Pipelining Basics



# Improving Washer / Dryer / Closet Utilization

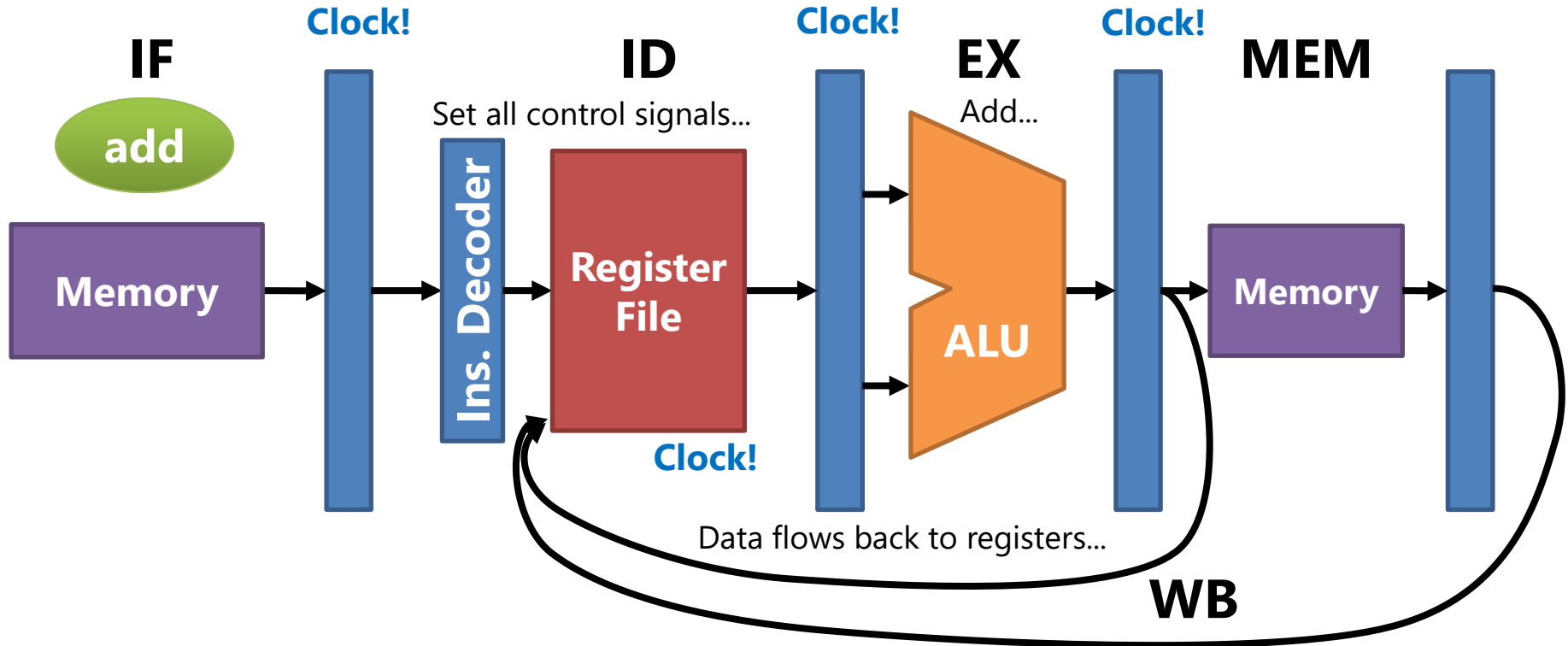
- If you work on loads of laundry one by one, you only get ~33% utilization
- If you form an "assembly line", you achieve ~100% utilization!





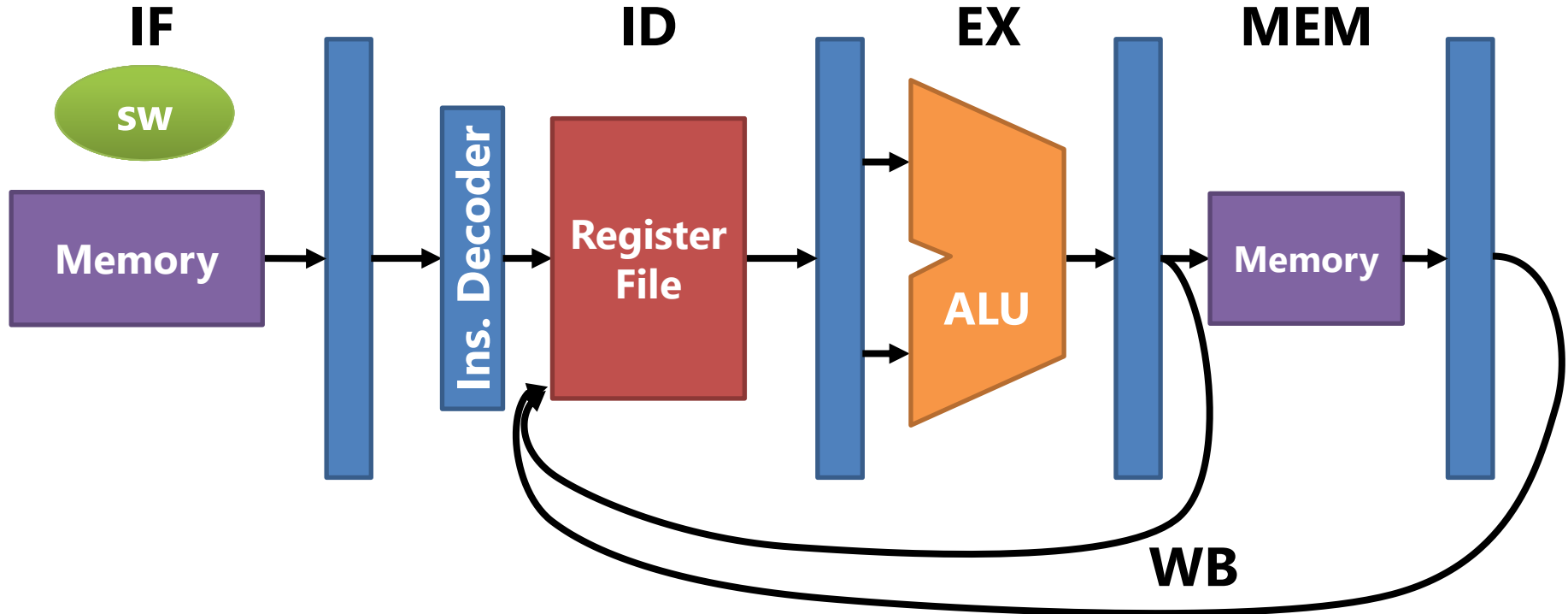
# Multi-cycle instruction execution

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



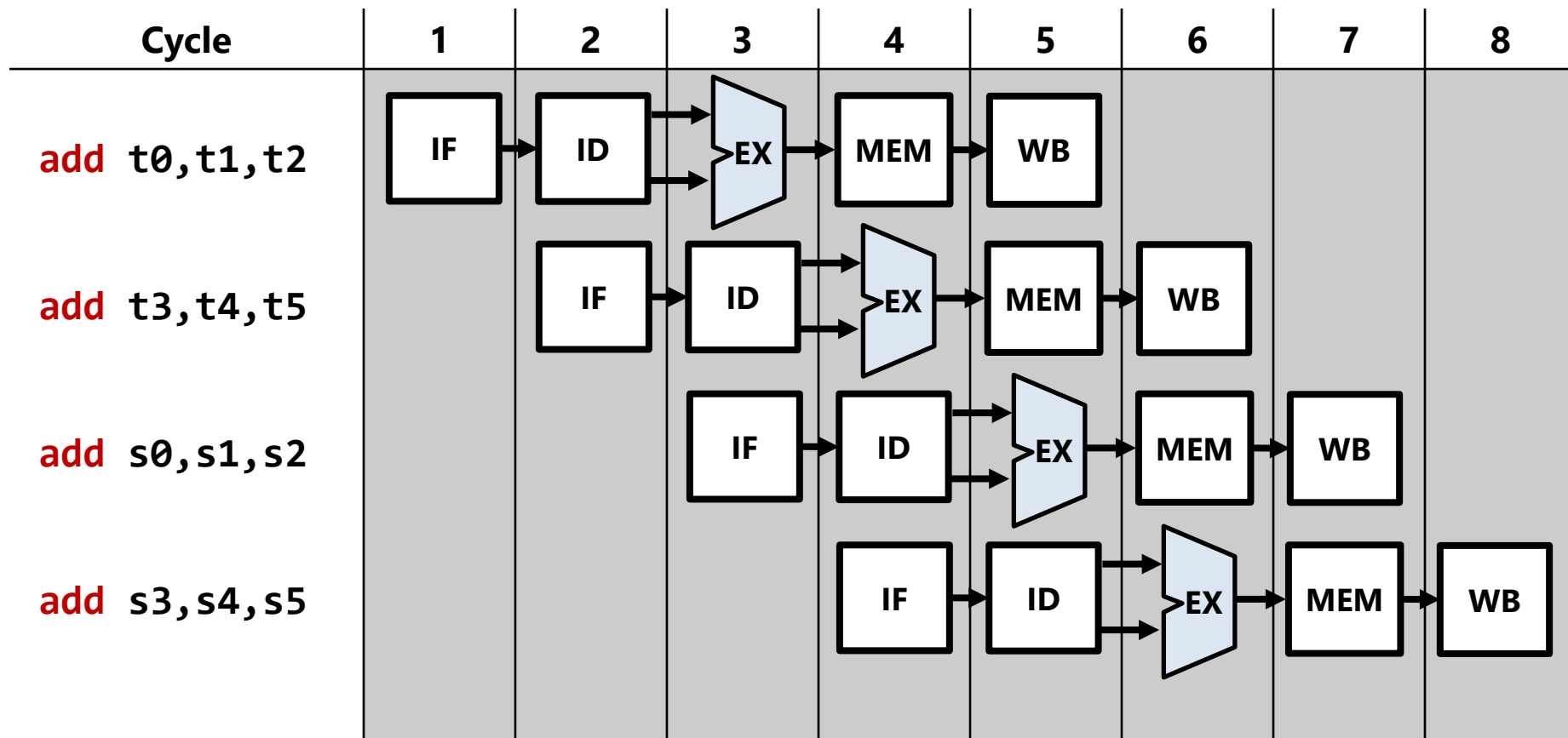
# Pipelined instruction execution

- 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?

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

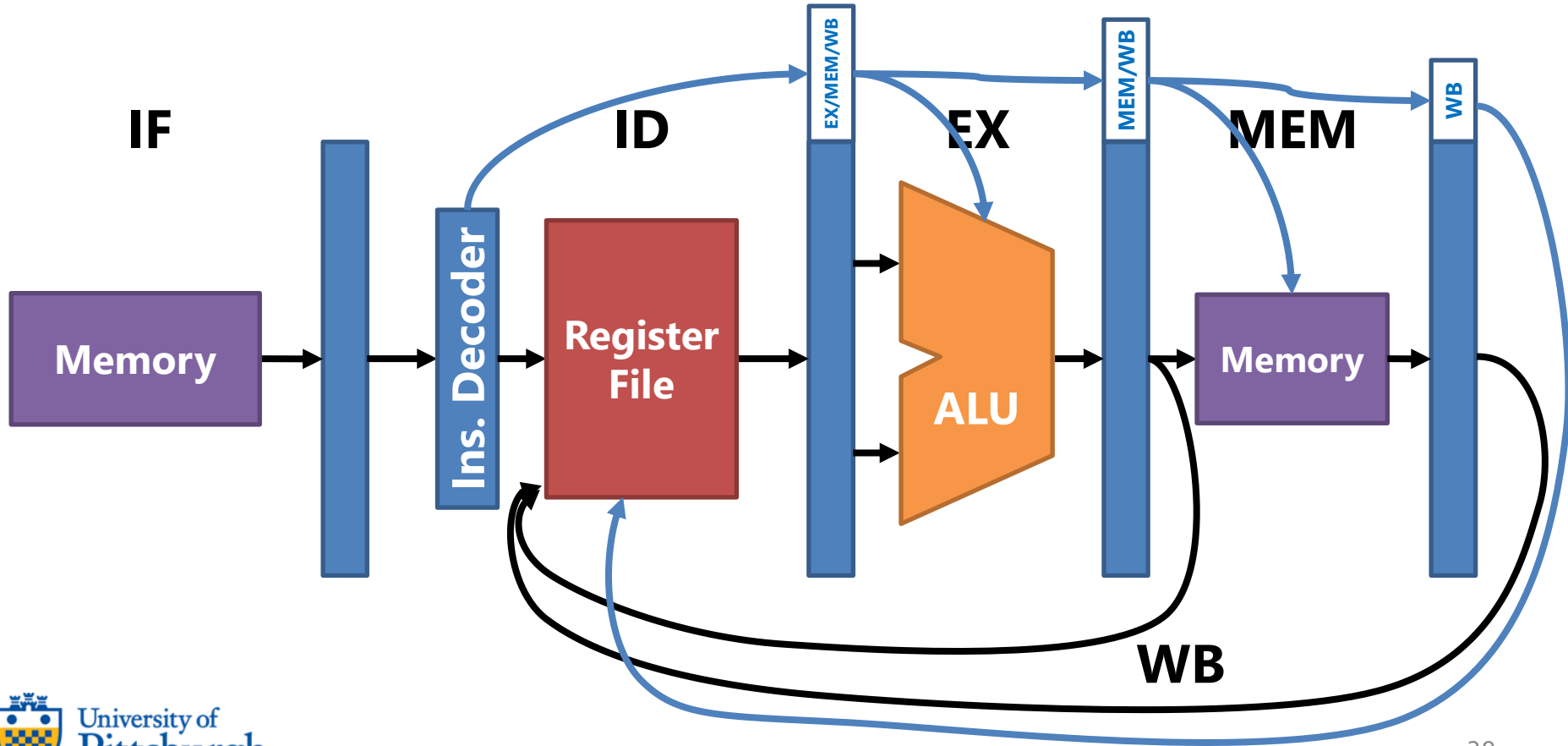
Architecture	Instructions	CPI	Cycle Time (1/F)
Single-cycle	Same	1	5 ns
Multi-cycle	Same	4~5	1 ns
Pipelined	Same	1	1 ns

- 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

---

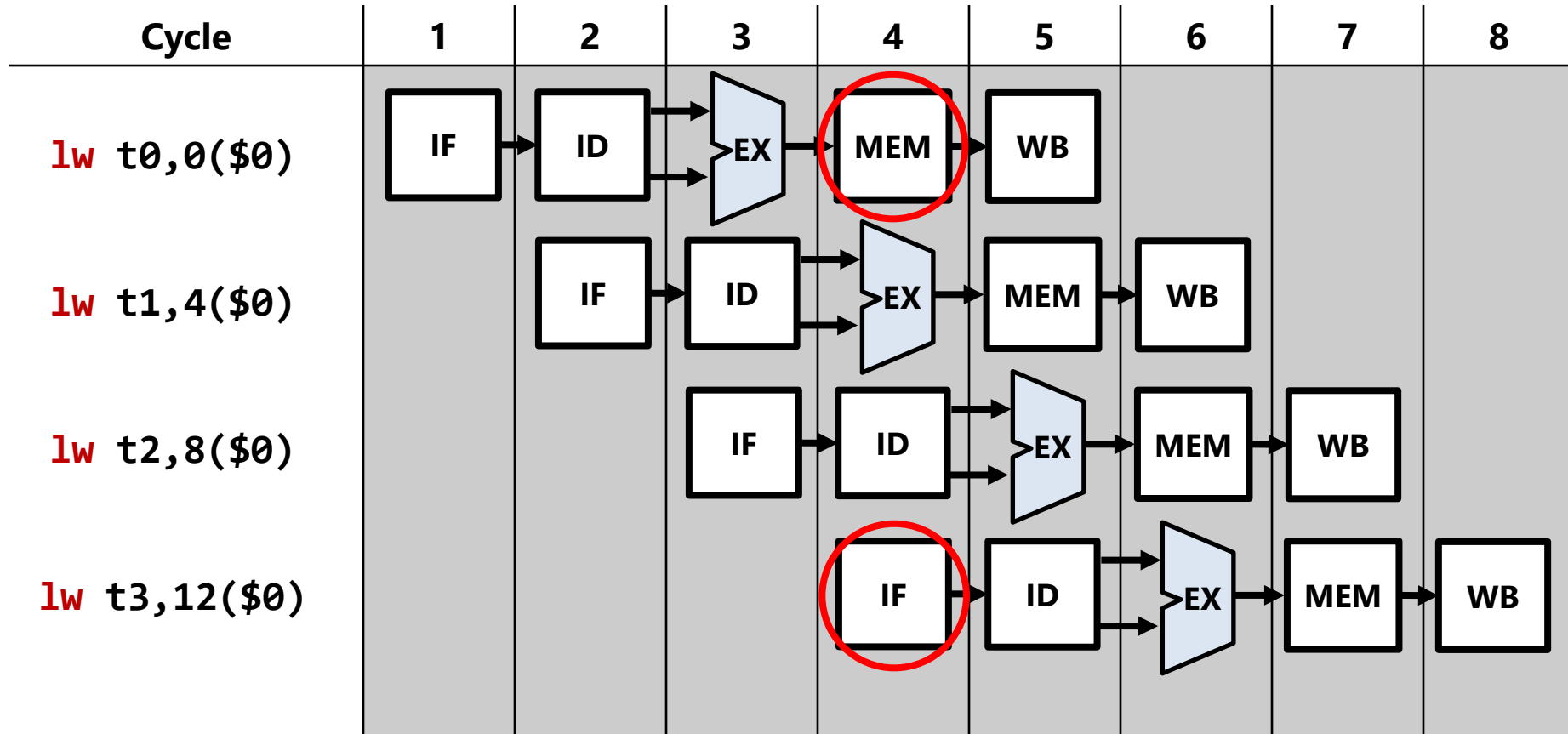
# Pipeline Hazards

- 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*



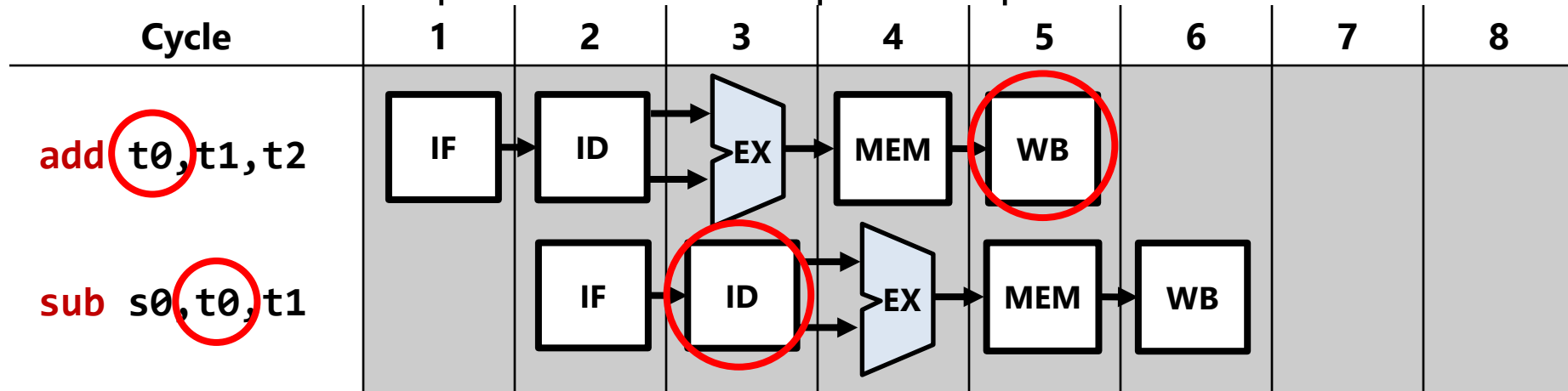
# Structural Hazards

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

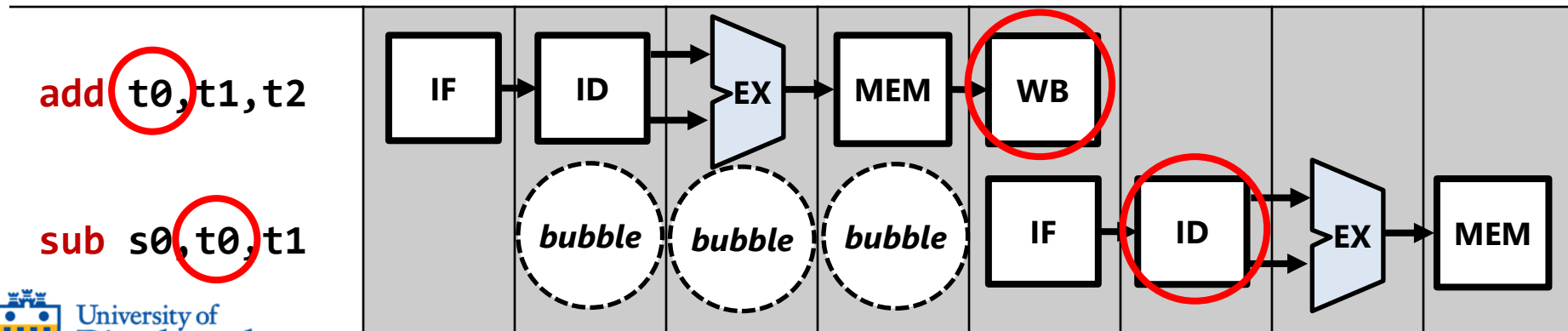


# Data Hazards

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

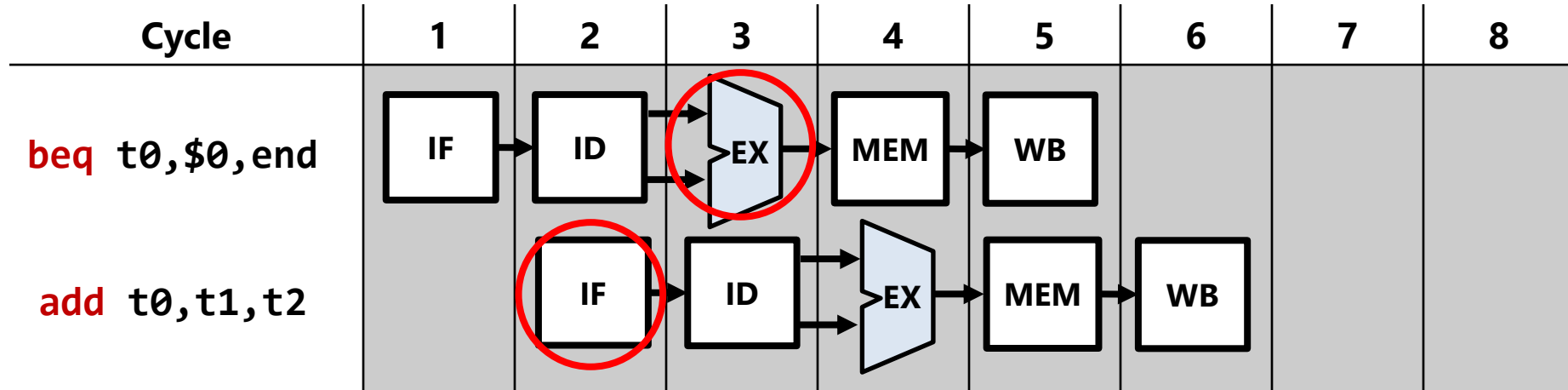


- sub** must wait until **add**'s WB phase is over before doing its ID phase

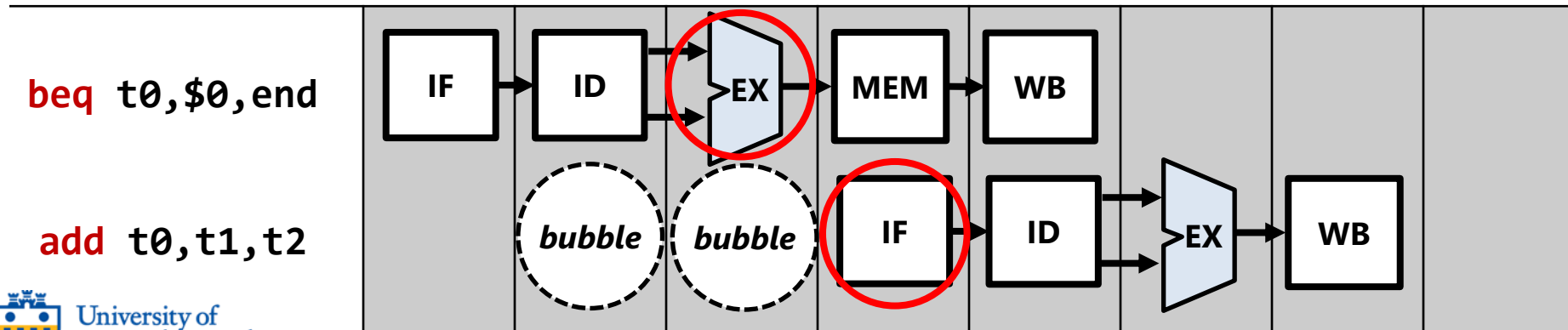


# Control Hazards

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



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

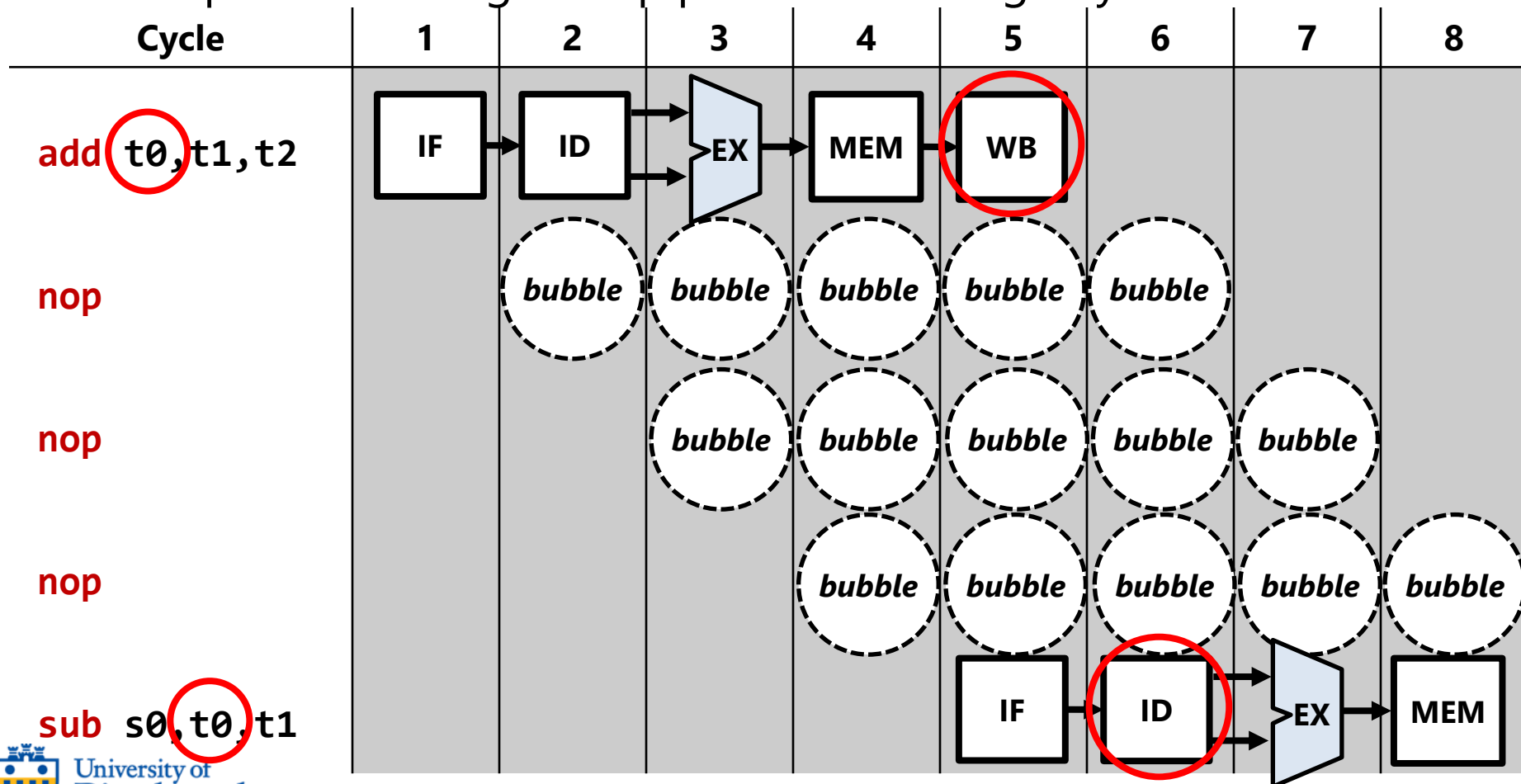


# Dealing with Hazards

- 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

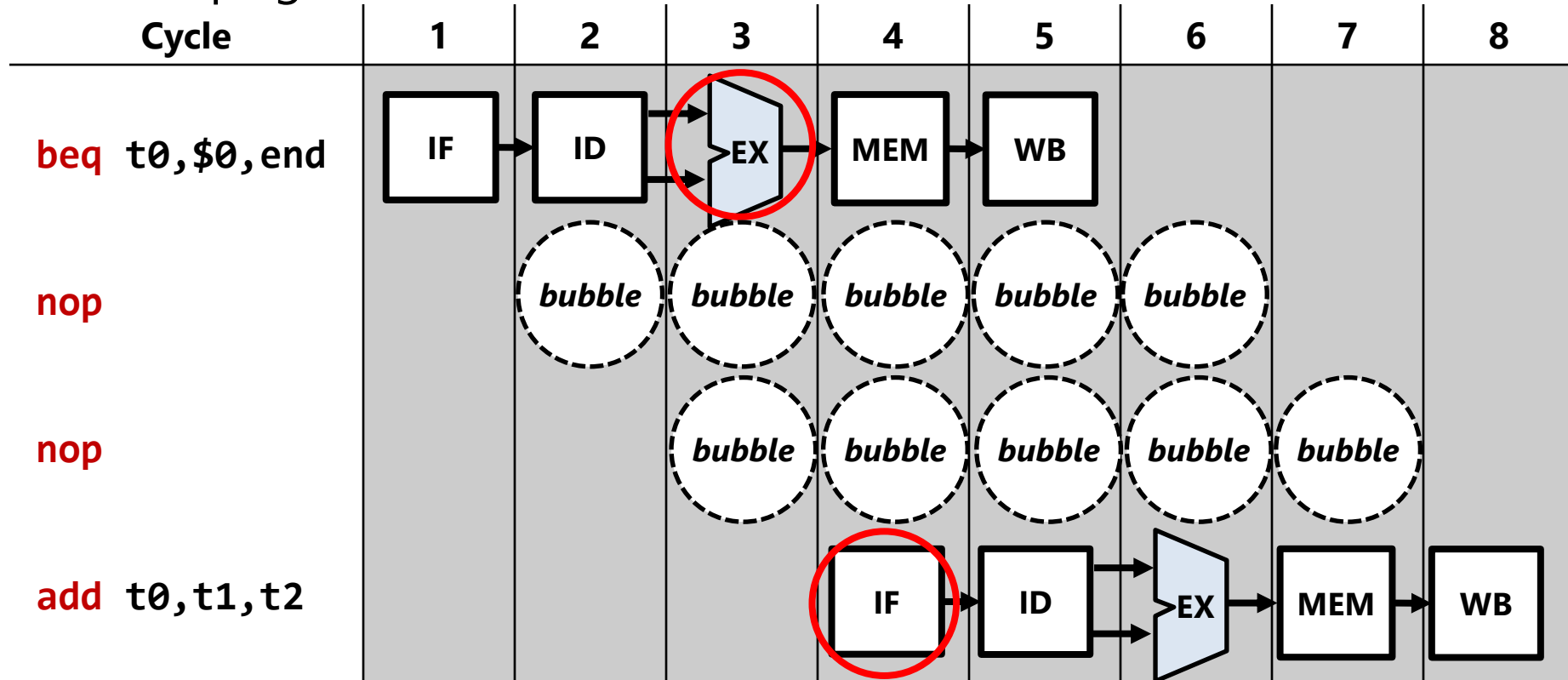
# Compiler avoiding a data hazard

- The nops flow through the pipeline not doing any work

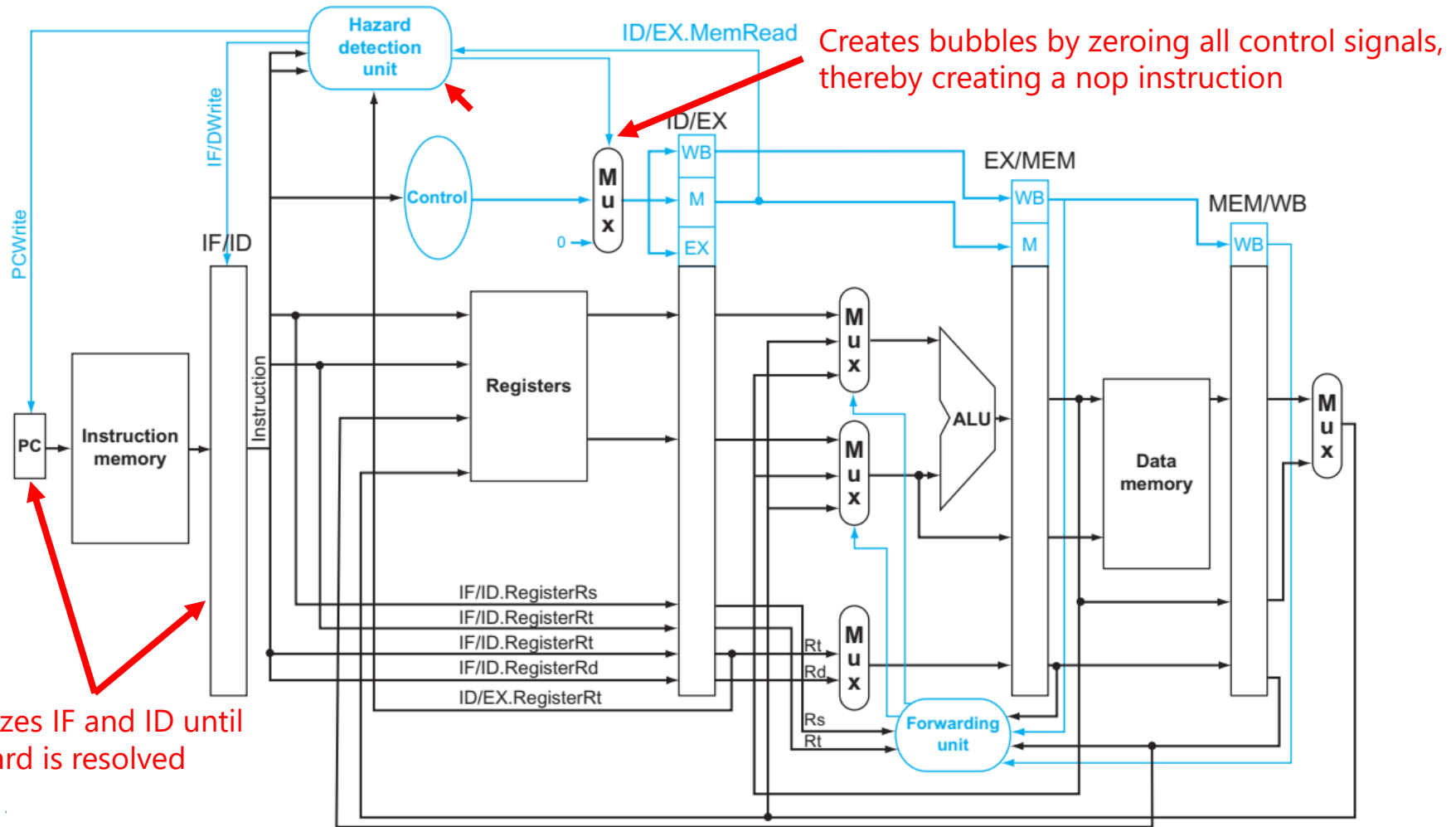


# Compiler avoiding a control hazard

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

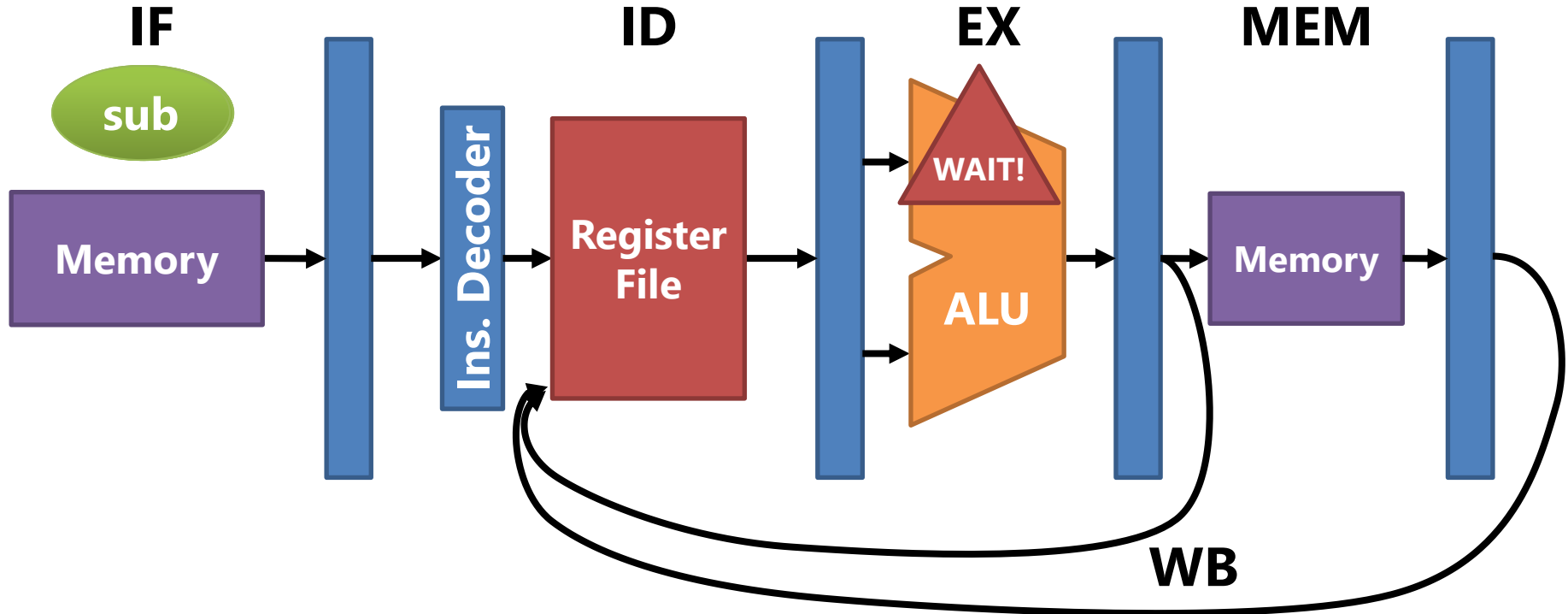


# Hazard Detection Unit



# 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");
```

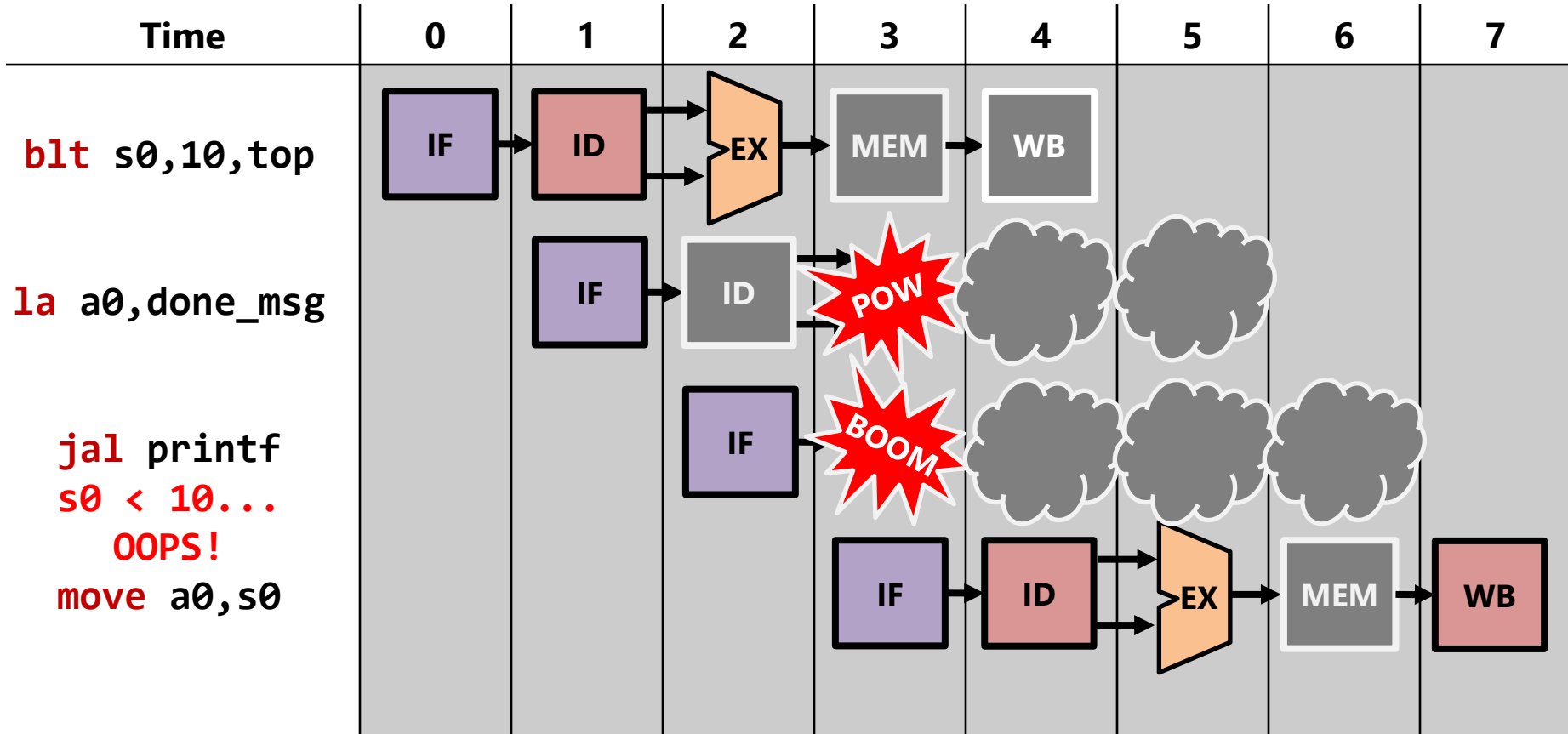
By the time `s0, 10`  
are compared at `blt`  
EX stage, the CPU  
would have already  
fetched `la` and `jal`!

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

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

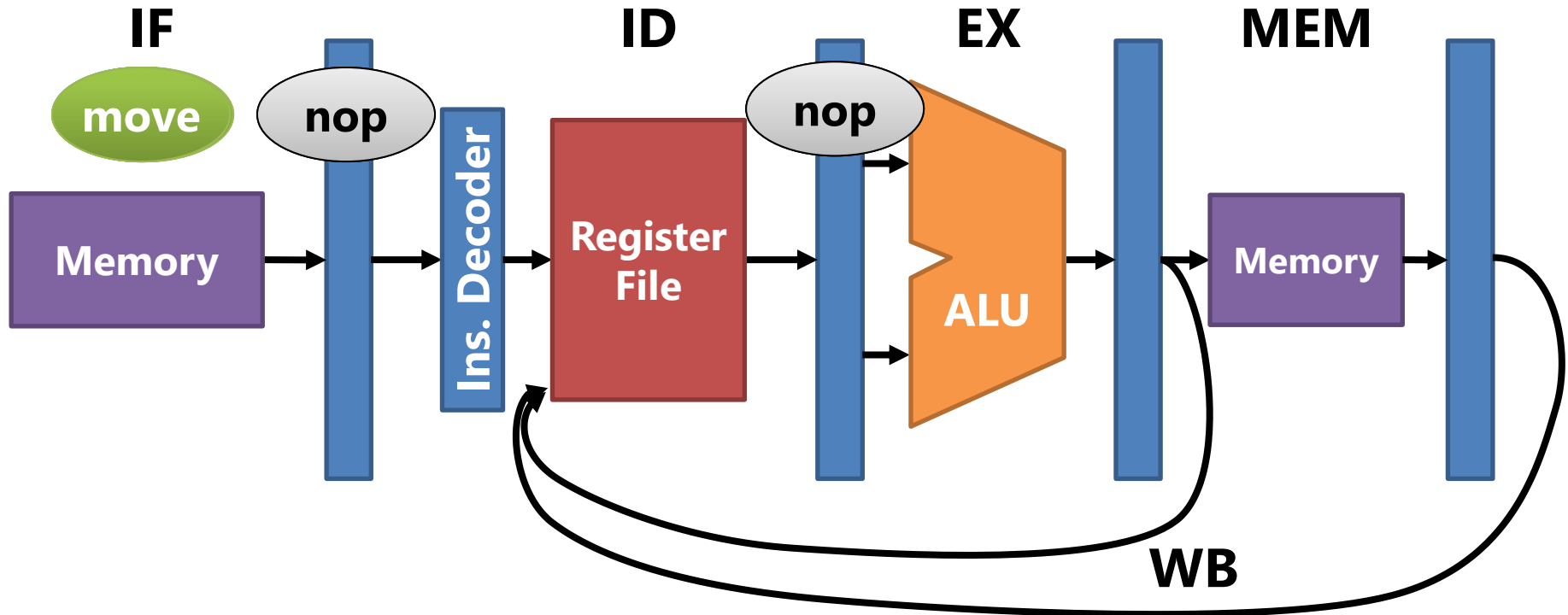
# What's a flush?

- 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) are transformed into **nops**.
  - Any "older" instructions (those that came BEFORE the branch) are left alone to finish executing as normal.

# Performance penalty of pipeline stalls

- Remember the three components of performance:

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

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
    - But processor design is not part of ISA
    - What if processor pipeline design changes in next CPU?
  - 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

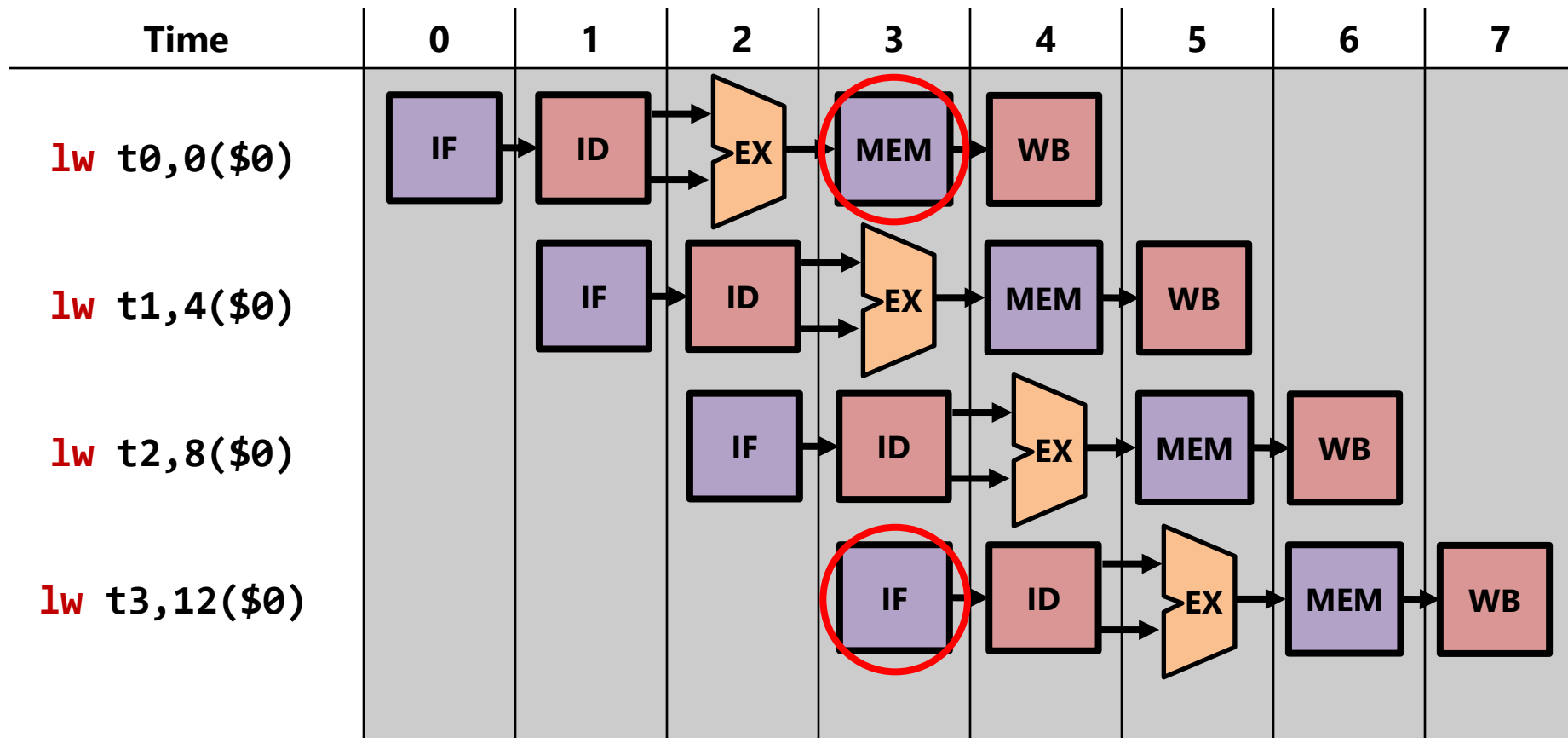
# Solving Structural Hazards

---



# Structural Hazard on Memory

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



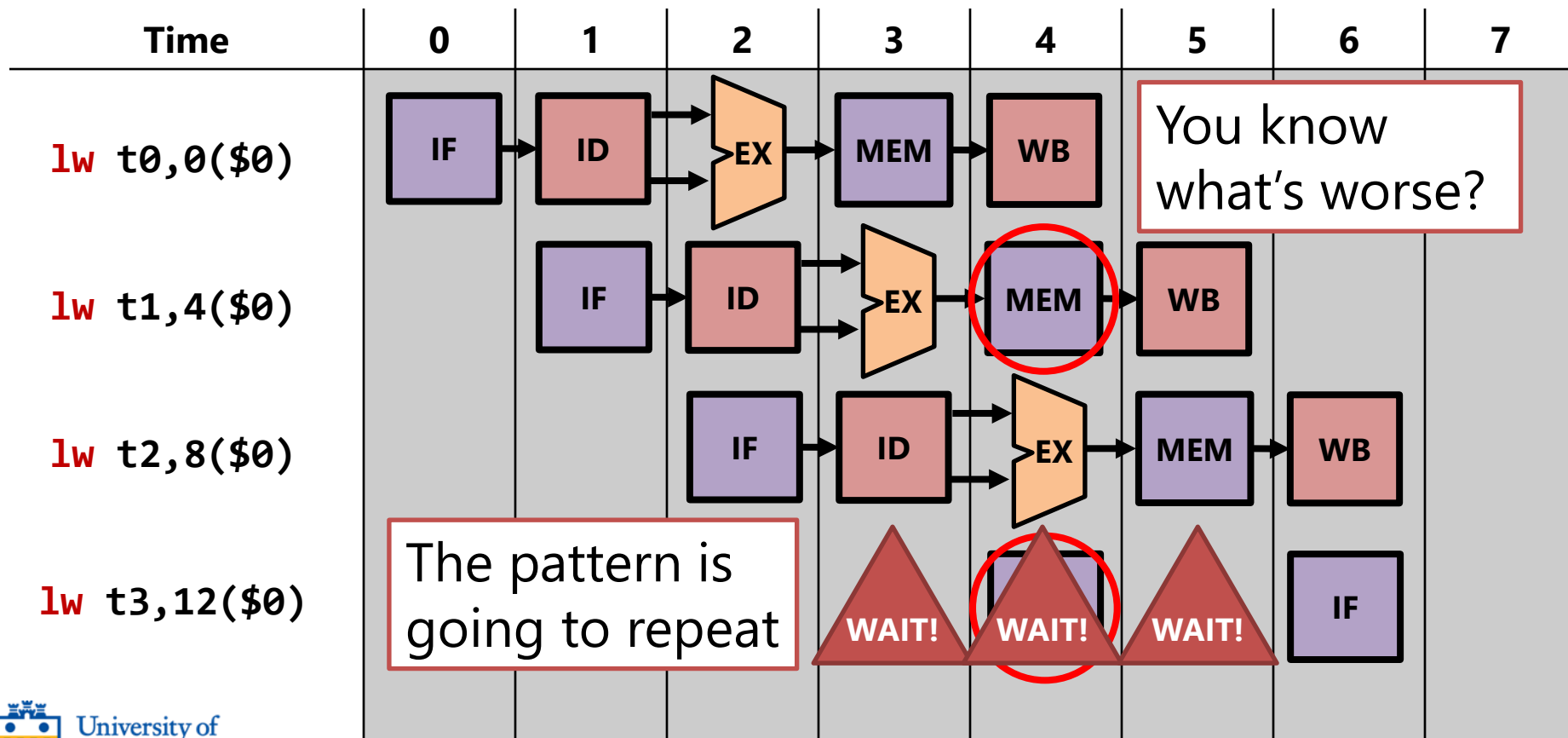
# What could we do??

- Two people need to use **one** sink at the same time
  - Well, in this case, it's memory but same idea



# We can do something similar!

- One option is to **wait** (a.k.a. **stall**).



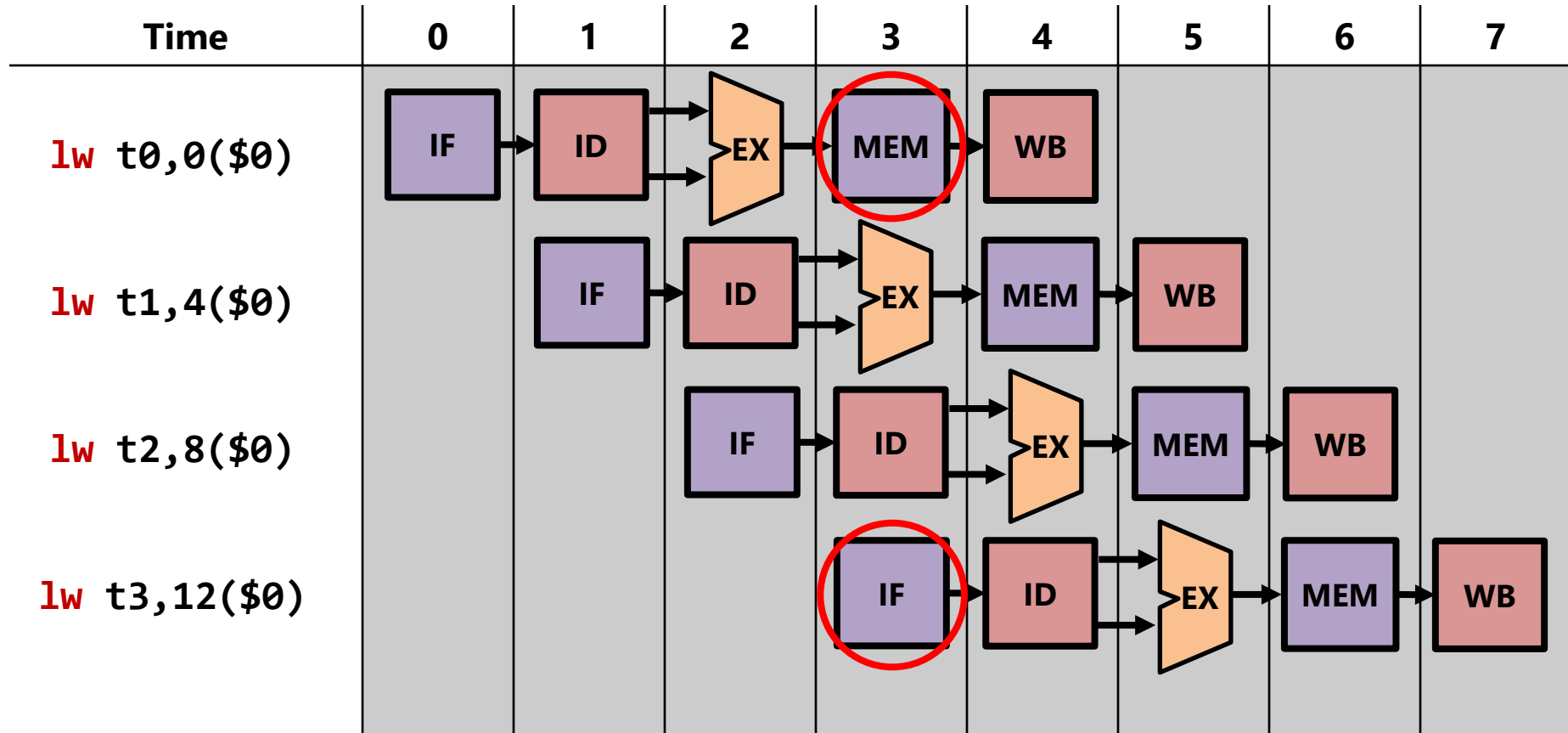
# Or we could throw in more hardware!

- For less commonly used CPU resources, stalling can work fine
- But memory (and some other things) is used **CONSTANTLY**
- How do the bathrooms solve this problem?
  - Throw in lots of sinks!
  - In other words, throw more hardware at the problem!
- Memory's a resource with a lot of **contention**
  - So have two memories, one for instructions, and one for data!
  - Not literally but CPUs have separate **instruction** and **data caches**



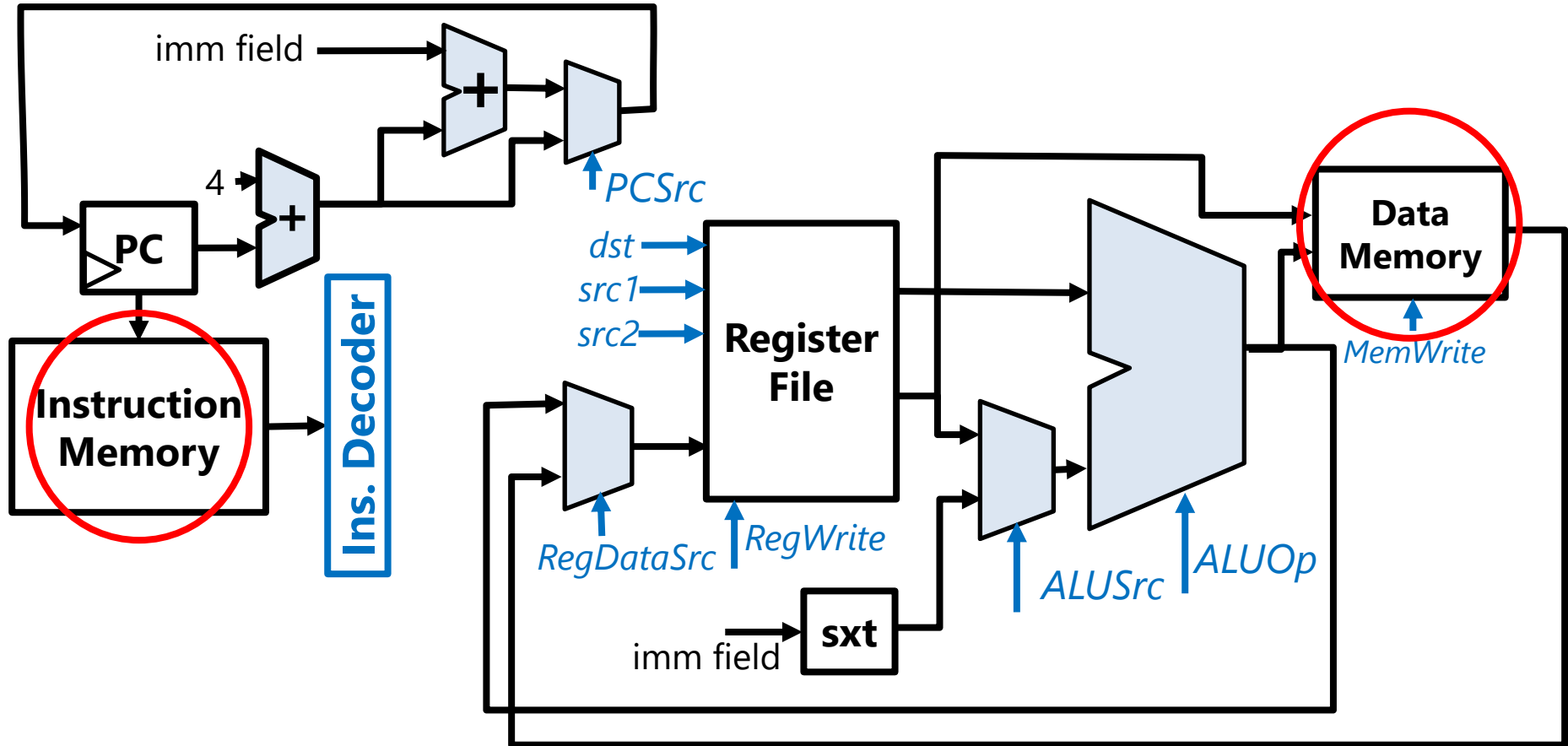
# Structural Hazard removed with two Memories

- With separate i-cache and d-cache, MEM and IF can work in parallel



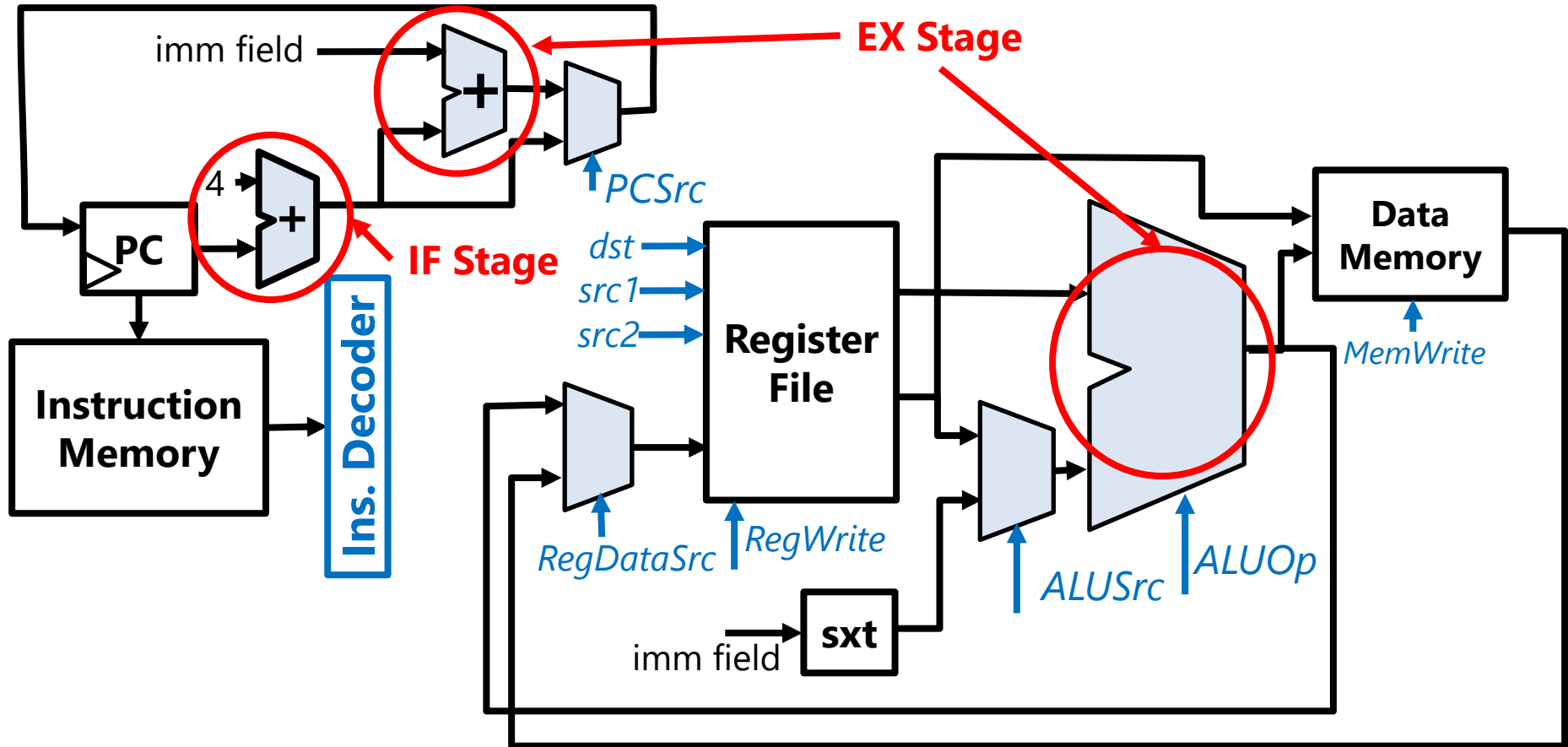
# Structural Hazard removed with two Memories

- But is that the only hardware duplication going on here?



# Structural Hazards removed with Multiple Adders

- Why do we need 3 adders? To avoid stalls due to contention on ALU!



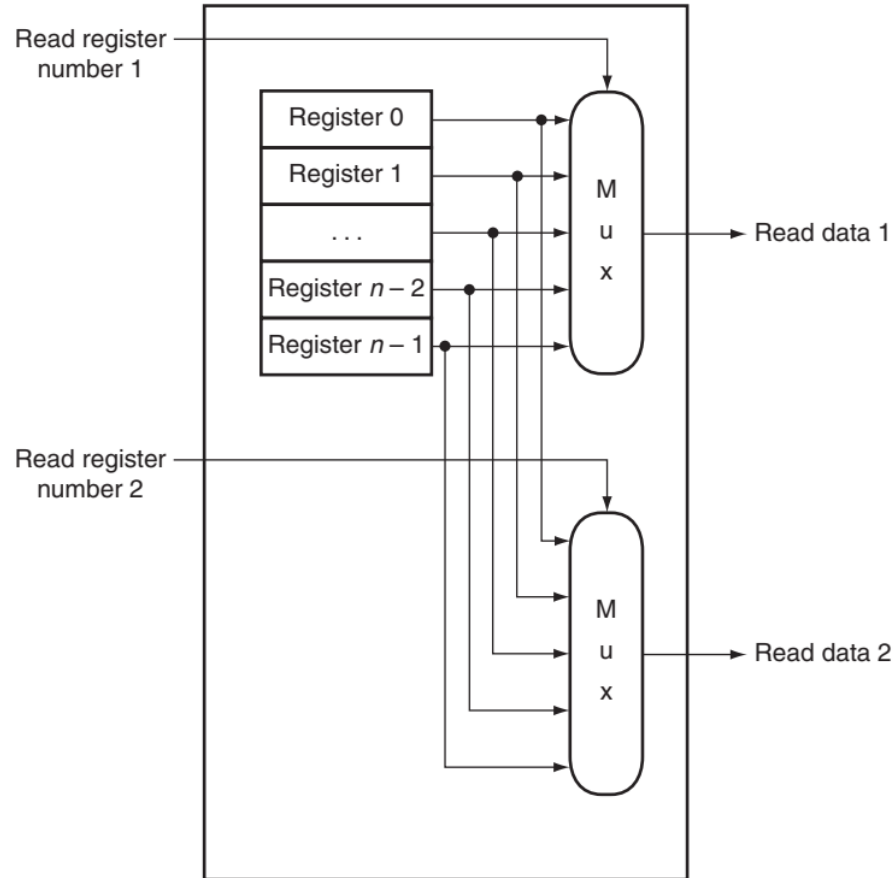
# Solving Structural Hazards

- There are mainly two ways to throw more hardware at the problem
  1. Duplicate contentious resource
    - One memory cannot sustain MEM + IF stage at same cycle  
→ Duplicate into one instruction memory, one data memory
    - One ALU cannot sustain IF + EX stage at same cycle  
→ Duplicate into one ALU and two simple adders
  2. Add ports to a single shared (memory) resource
    - **Port:** Circuitry that allows either read or write access to memory
    - If current number of ports cannot sustain rate of access per cycle  
→ Add more ports to memory structure for simultaneous access



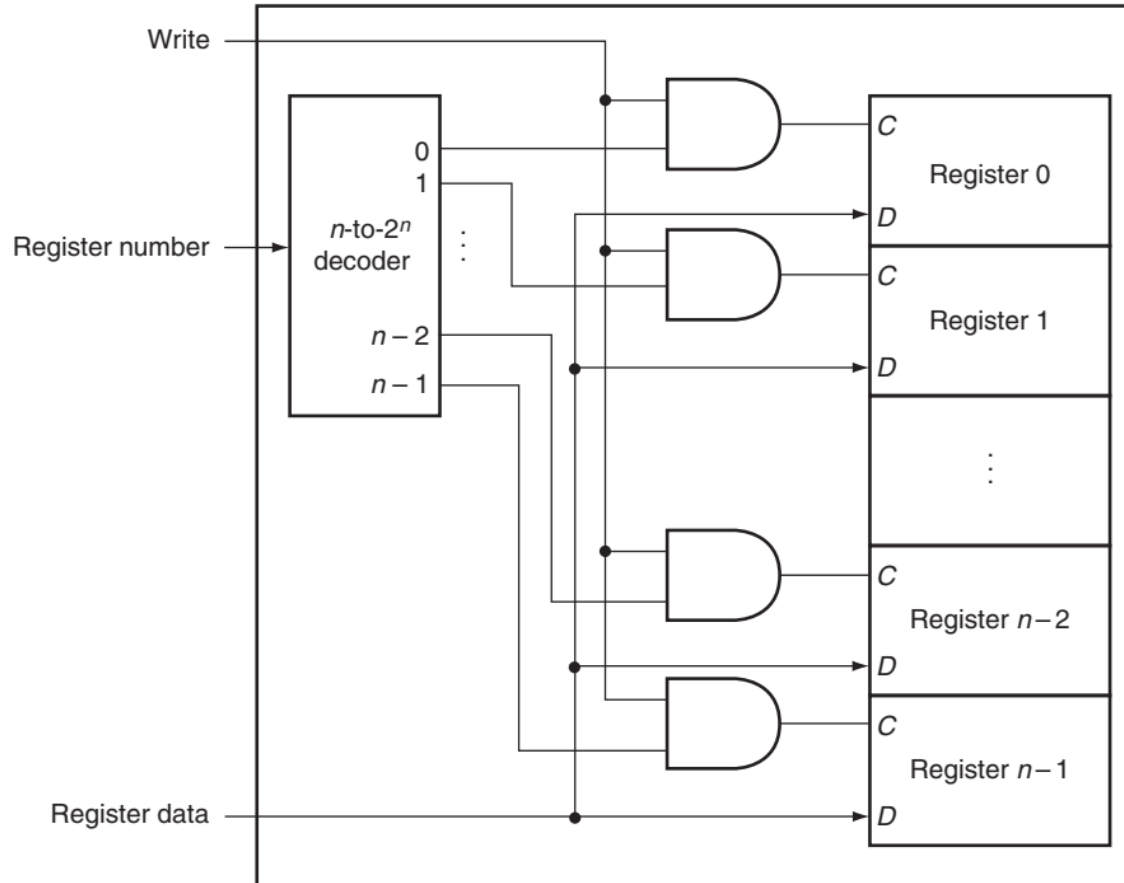
# Two Register Read Ports

- By adding more MUXes, you can add even more read ports



# One Register Write Port

- By adding more decoders, you can add more write ports



# But who would need more register ports?

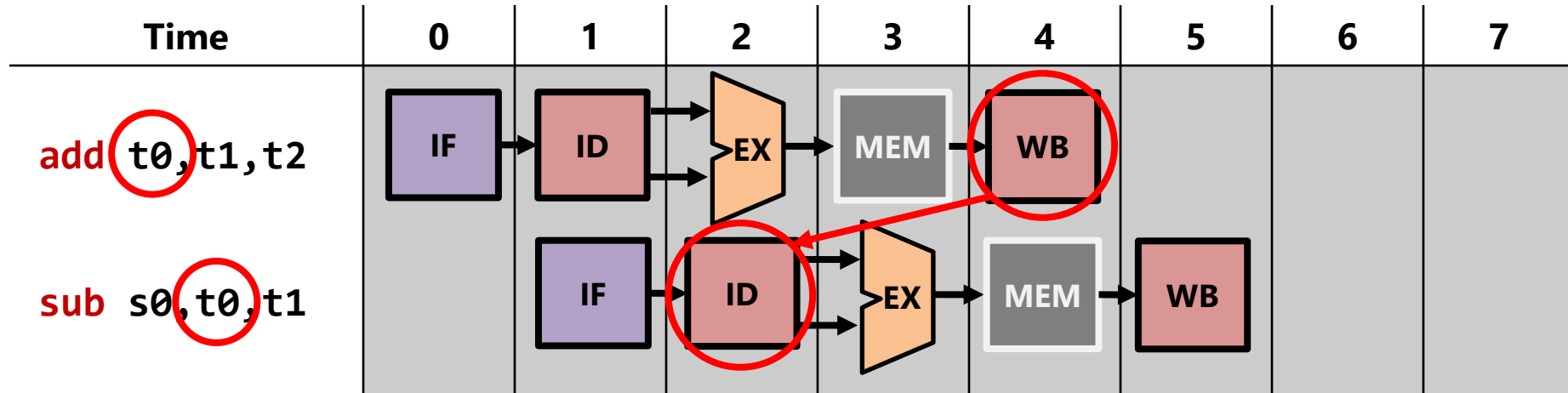
- With two read ports and one write port
  - Enough to sustain one ID and one WB stage per cycle
  - Enough to sustain  $CPI = 1$  (or in other words  $IPC = 1$ )
- But what if we want an  $IPC > 1$ ?
  - More than one instruction per cycle! (a.k.a superscalar processor)
  - Must sustain more than one ID / WB stage per cycle
  - Need more register read ports and write ports!
  - Not only registers, memory would need more ports too!
  - Like everything else, this consumes lots of **power**
- We'll talk more about this when we discuss superscalars

# Solving Data Hazards

---

# Data Hazards

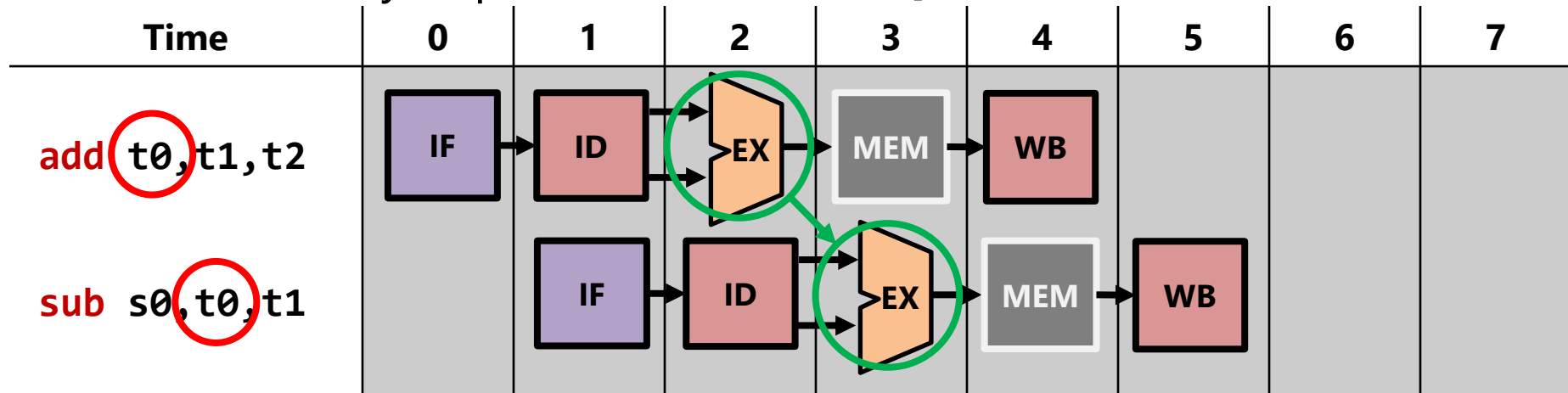
- An instruction depends on the output of a previous one.



- When does **add** finish computing its sum?
- Well then... why not just *use the sum when we need it*?

# Data Forwarding

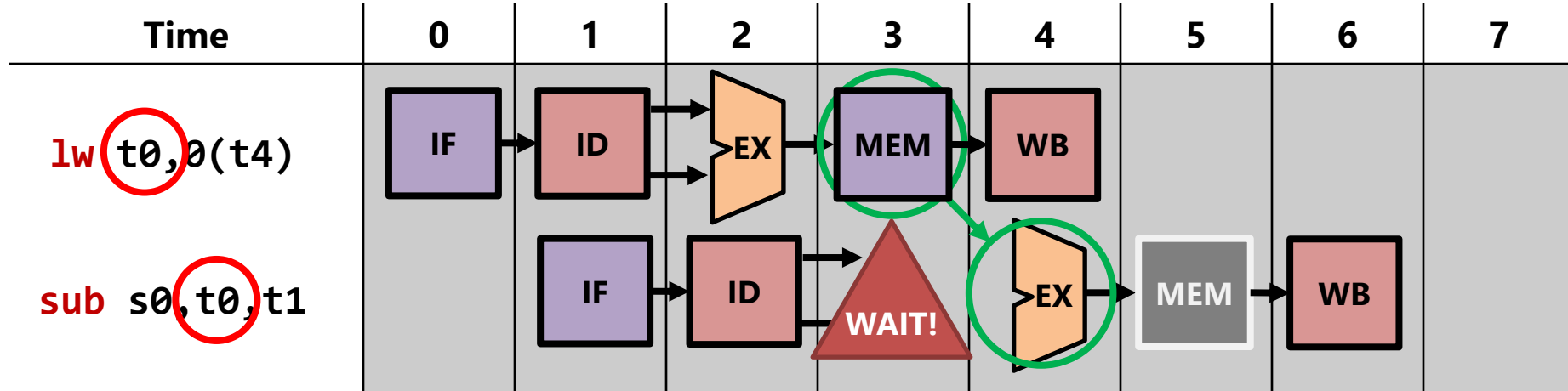
- Since we've pipelined control signals, we can check if instructions in the pipeline depend on each other (see if registers match).
- If we detect any dependencies, we can **forward** the needed data.



- This handles one kind of data forwarding...
- Where else can data come from and be written into registers?
  - Memory!

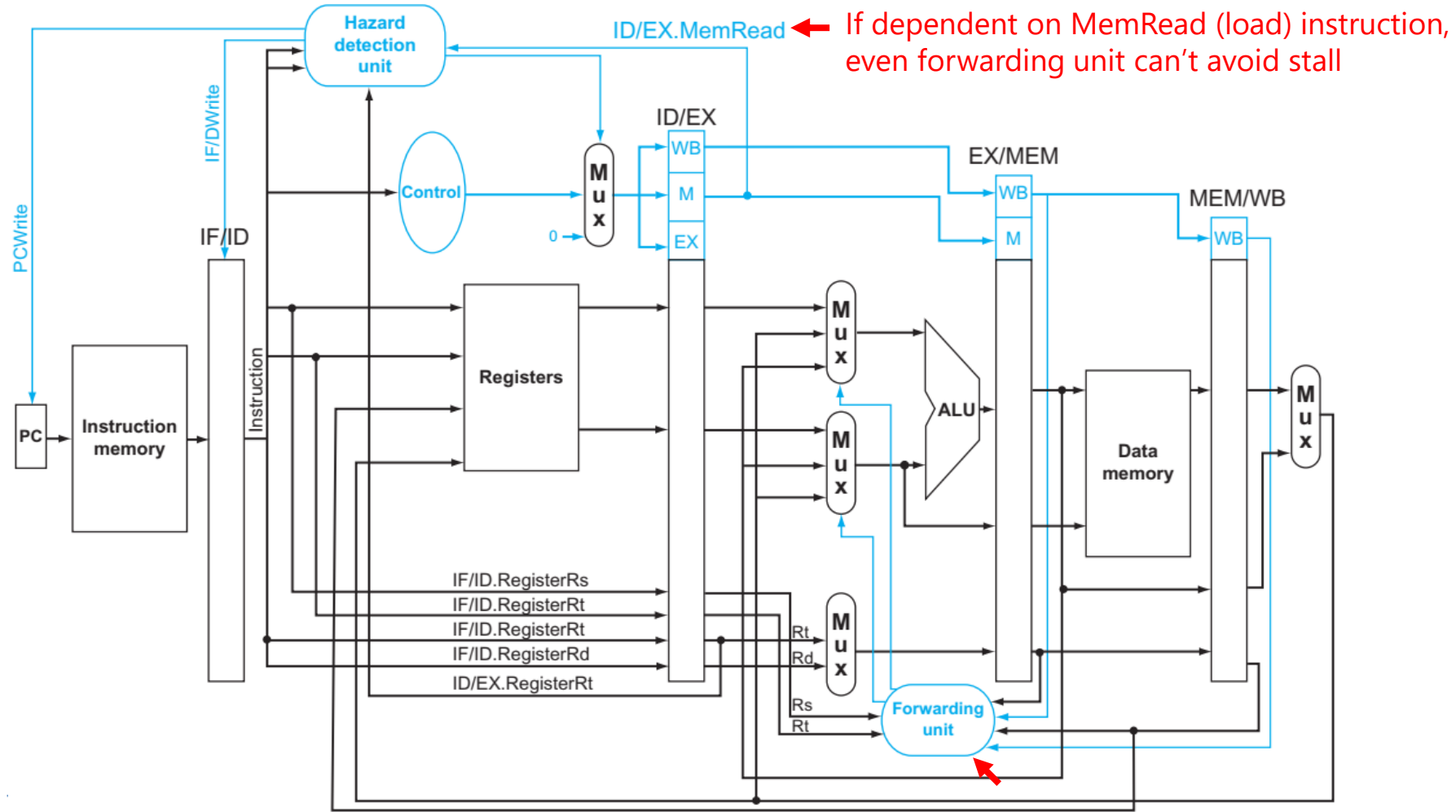
# Data Forwarding from Memory

- Well memory accesses happen a cycle later...
- What are we going to have to do?



- This kind of stall is unavoidable in our current pipeline

# Forwarding Unit and Use-after-load-hazard



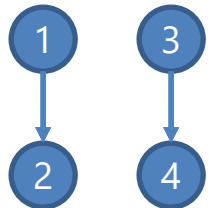


# Forwarding Unit

- Just like the HDU, the Forwarding Unit is **power** hungry
- Number of forwarding wires  $\propto (\text{pipeline stages})^2$ 
  - Why the **quadratic** relationship?
  - Per pipeline stage, N stages after it *from* which data is forwarded
    - In previous picture, see number of inputs to MUX before ALU!
  - And there are N stages *to* which data must be forwarded
    - In previous picture, only one EX stage is shown, but if there are multiple stages, need MUXes in all those stages
- Thus, deep pipelining has diminishing returns on power investment
  - Cycle time improves by a factor of N
  - Power consumption increases by a factor of  $N^2$  (or more)
  - Not the only problem with deep pipelining that we will see

# Avoiding memory stalls

- Let's say the following is your morning routine (*2 hours total*)
  1. Have laundry running in washing machine (*30 minutes*)
  2. Have laundry running in dryer (*30 minutes*)
  3. Have some tea boiling in the pot (*30 minutes*)
  4. Drink tea (*30 minutes*)
- Can you make this shorter? Yes! (*1 hour total*)
  1. Have washing machine running *and* 3. Tea boiling (*30 minutes*)
  2. Have dryer running *and* 4. Drink tea (*30 minutes*)
- How? By simply by **reordering** our actions
  - Steps 1 → 2 and 3 → 4 have data dependencies
  - Other steps can be freely reordered with each other



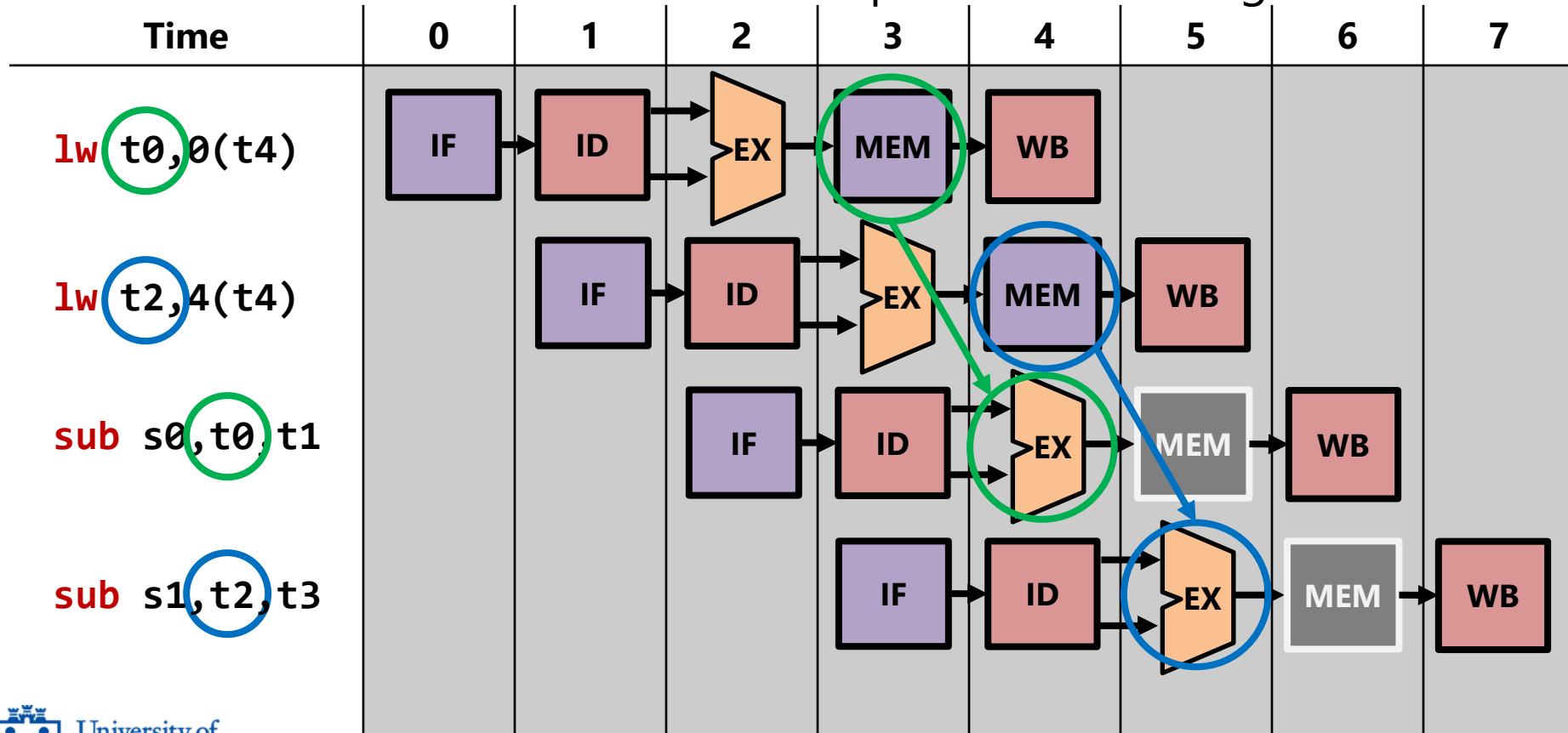
# Data Hazard removed through Compiler Scheduling

- If the **compiler** has knowledge of how the pipeline works, it can **reorder** instructions to let loads complete before using their data.

Time	0	1	2	3	4	5	6	7
<b>lw</b> $t_0, 0(t_4)$								
<b>sub</b> $s_0, t_0, t_1$								
<b>lw</b> $t_2, 4(t_4)$								
<b>sub</b> $s_1, t_2, t_3$								

# Data Hazard removed through Compiler Scheduling

- If the **compiler** has knowledge of how the pipeline works, it can **reorder** instructions to let loads complete before using their data.



# Limits of Static Scheduling

- Scheduling done by the compiler is called ***static scheduling***
- Static scheduling is a powerful tool but is in some ways **limited**
  - Again, compiler must make assumptions about pipeline
  - Length of MEM stage is very hard to predict by the compiler
    - Remember the **Memory Wall**?
  - Data dependencies are hard to figure out by a compiler
    - When data is in registers, trivial to figure out
    - When data is in memory locations, more difficult. Given:  

```
lw t0, 0(t4)
sw s0, 8(t0)
lw t2, 4(t4)
```

We want to reorder to remove the data hazard.  
But what if 8(t0) and 4(t4) are the same addresses?  
This involves *pointer analysis*, a notoriously difficult analysis!

# Dynamic scheduling is another option

- **Dynamic scheduling** is scheduling done by the CPU
- It doesn't have the limitations of static scheduling
  - It doesn't have to predict memory latency
    - It can adapt as things unfold
  - It's easy to figure out data dependencies, even memory ones
    - At runtime, addresses of  $8(t_0)$  and  $4(t_4)$  are easily calculated
- But at runtime it uses lots of power for the data analysis
  - ... which again causes problems with the **Power Wall**
  - But more on this later

# Solving Control Hazards

---

# Loops

- Loops happen *all the time* in programs.

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

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

How often does this **blt** instruction go to **top**? How often does it go to the following **la** instruction?

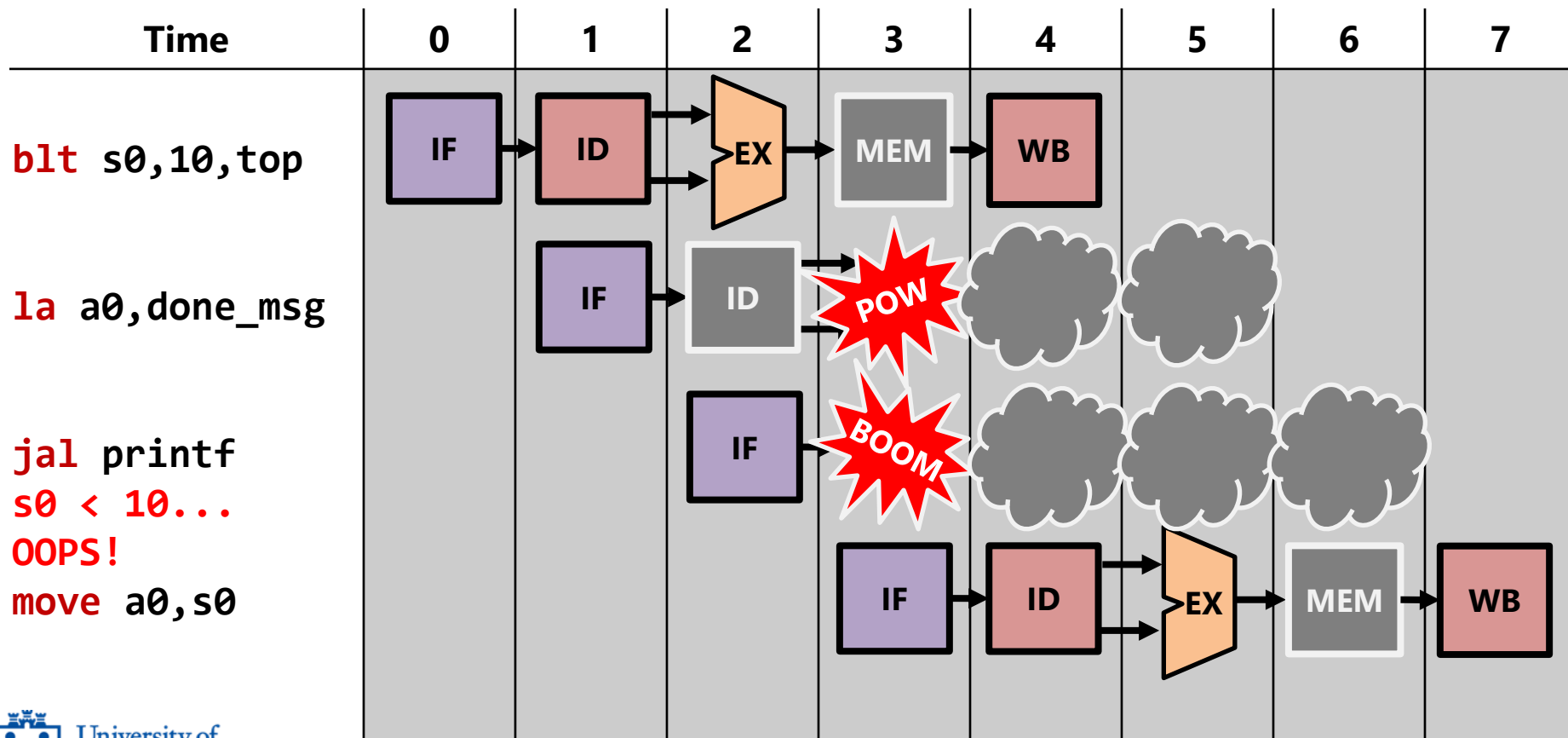
```
li    s0, 0  
top:  
move  a0, s0  
jal   print  
addi  s0, s0, 1  
blt   s0, 10, top
```

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



# Pipeline Flushes at Every Loop Iteration

- The pipeline must be **flushed** every time the code loops back!



# And it just gets worse...

- If the loop is only a few instructions long...
  - That stall is a larger proportion of the time.
- If that loop iterates thousands of times...
  - We'll be spending more time killing instructions than executing!
- The deeper the pipeline gets, the bigger the flush penalty
  - Number of flushed instructions == distance from IF to MEM
  - Current architectures typically have more than a dozen stages!
- How can we solve this?
  - We need to somehow ***predict*** the correct branch outcome
  - But this is complicated! We'll take this up at a later time.