CPU Pipelining

CS/COE 1541 (Fall 2020)
Wonsun Ahn

Clocking Review

Stuff you learned in CS 447

● Do you remember what all these do?

NOT gate
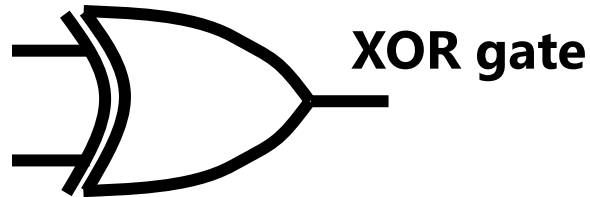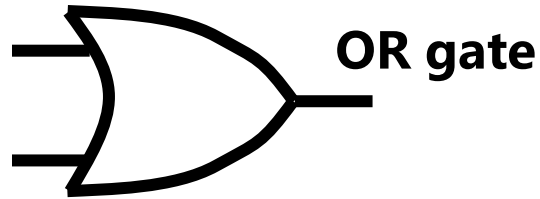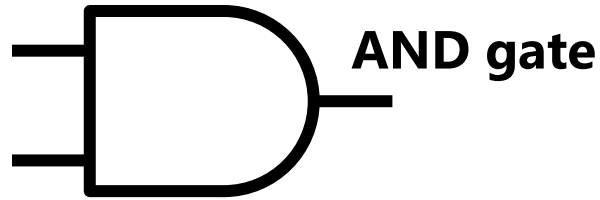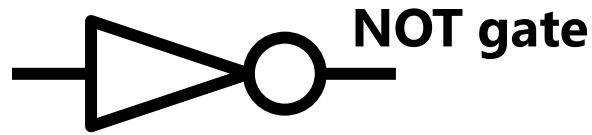
AND gate

OR gate

XOR gate

Multiplexer (MUX)

Decoder

ALU

32

32

32

32

These wires carry several bits at once.
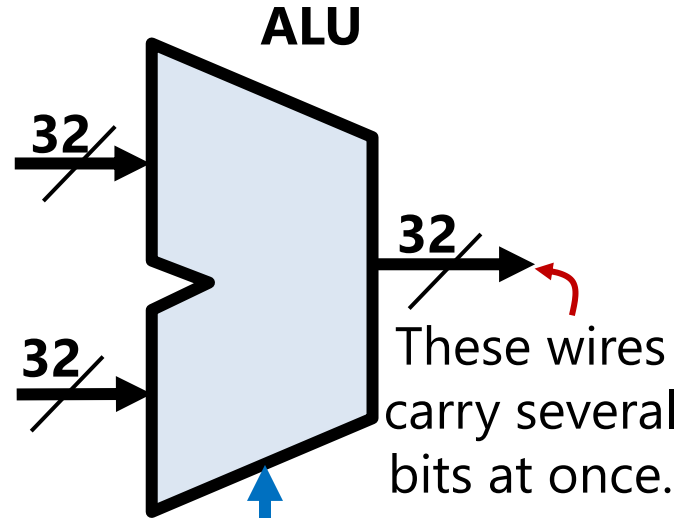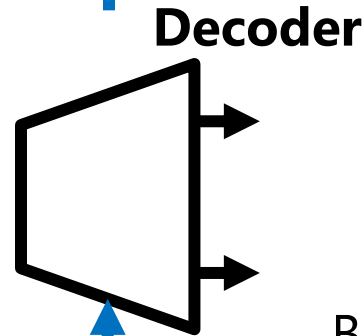
Blue wires are control signals.

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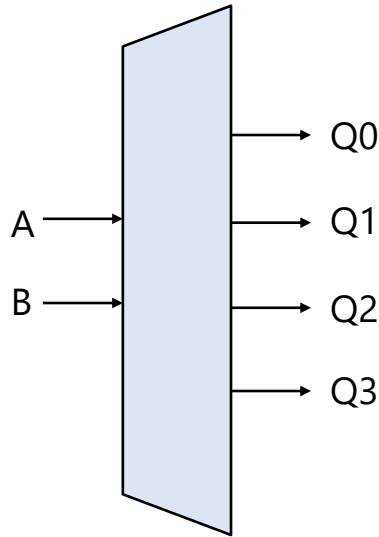
Truth Table for Decoder

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

Q0
A
Q1
B
Q2
Q3

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

University of Pittsburgh

- No problem in fanning out one signal to two points
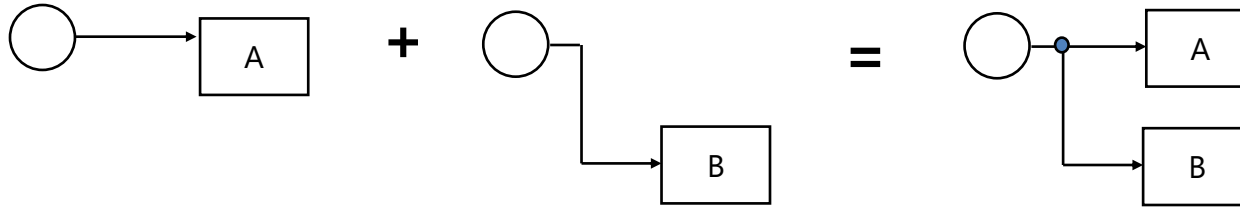
- Cannot connect two signals to one point
  - Must use a multiplexer to *select* between the two

Path A   Path B

Mux control =
  0 for path A
  1 for path B
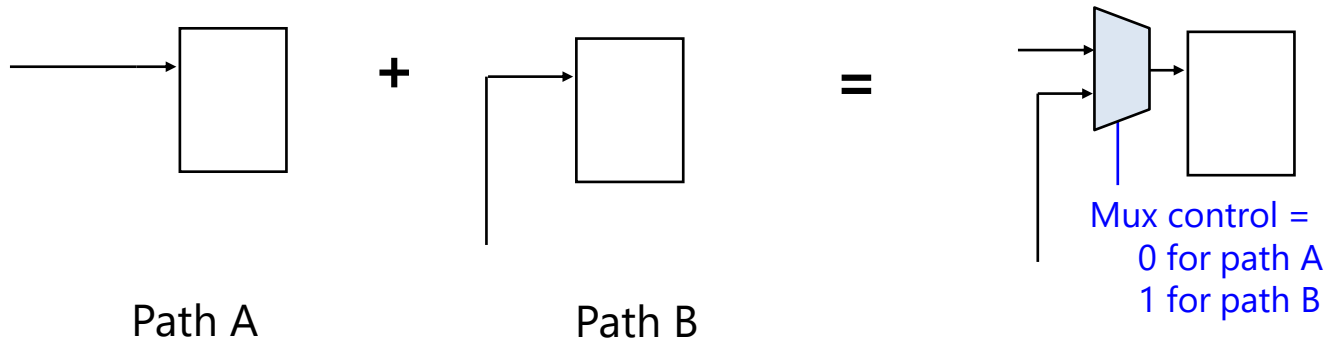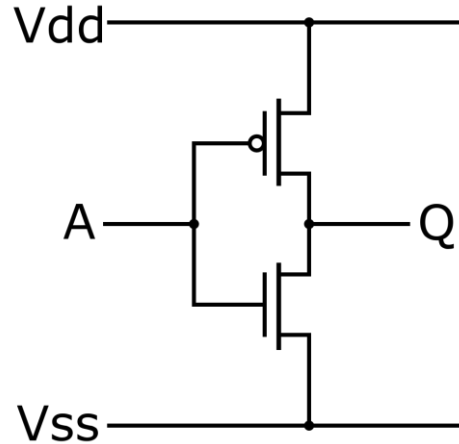
- NOT gate
- NAND gate
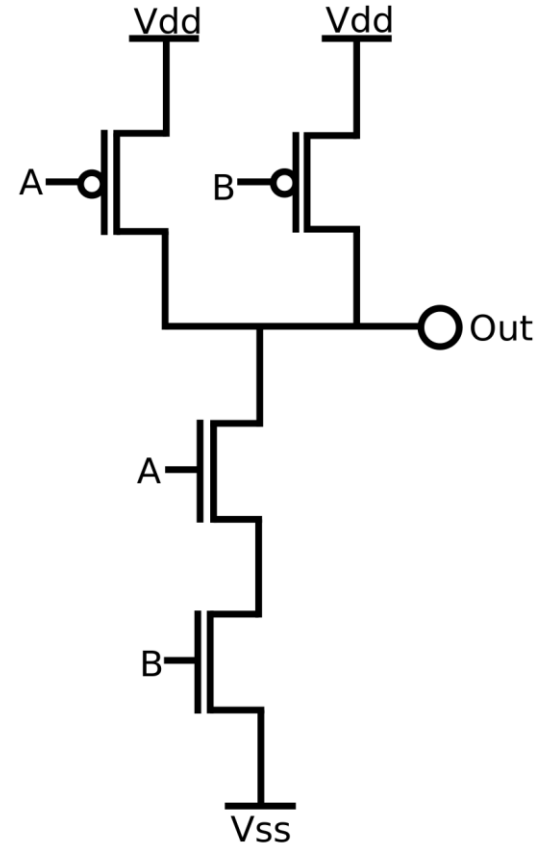
● The clock is a signal that alternates regularly between 0 and 1:
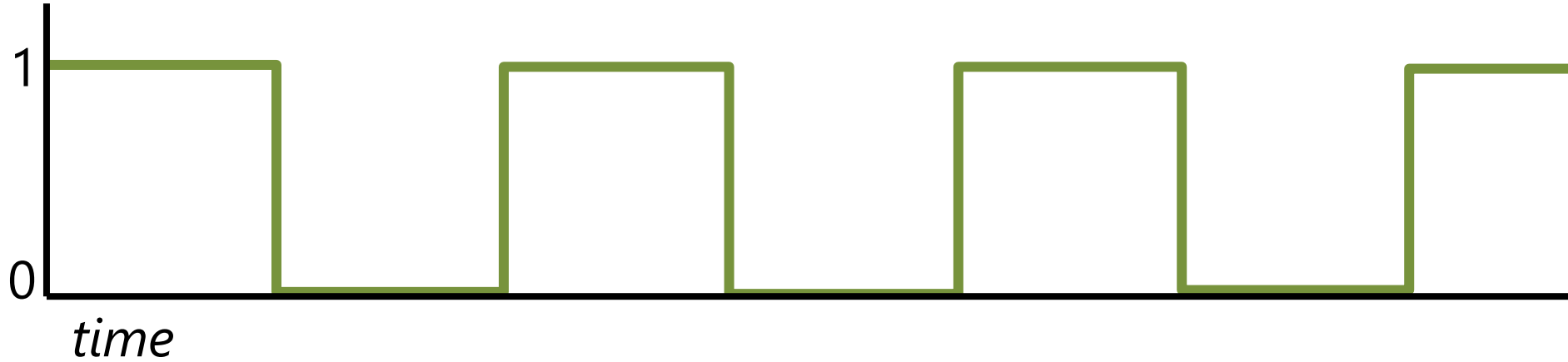
time

● Bits are latched on to registers and flip-flops on rising edges

● In between rising edges, bits propagate through the logic circuit
  o Composed of ALUs, muxes, decoders, etc.
  o ***Propagation delay***: amount of time it takes from input to output

- **_Critical path_**: path in a circuit that has longest propagation delay
  o Determines the overall clock speed.
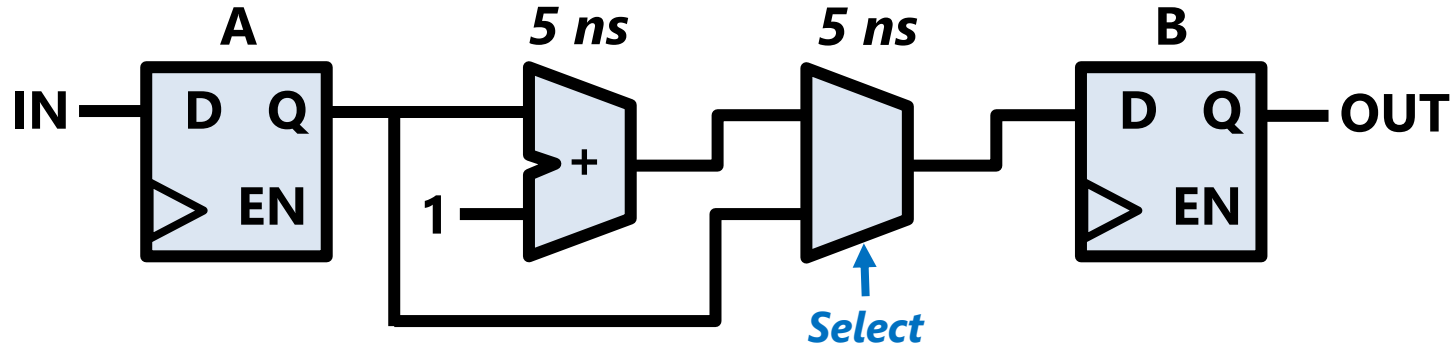
  o The ALU and the multiplexer both have a 5 ns delay
- How fast can we clock this circuit?
  o Is it 1 / 5 ns (5 × 10^{-9}s) = 200 MHz?
  o Or is it 1 / 10 ns (10 × 10^{-9}s) = 100 MHz? ✓

University of Pittsburgh

8

MIPS Review

Stuff you learned in CS 447

University of Pittsburgh

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

University of Pittsburgh

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

Addr	Data
0	0x3F
1	0x00
2	0x2A
3	0x08
4	0x47
5	0xF4
6	0x26
7	0xB9
...	...

Addr	Data
0	0x3F00
2	0x2A08
4	0x47F4
6	0x26B9
...	...

Addr	Data
0	0x3F002A08
4	0x47F426B9
...	...
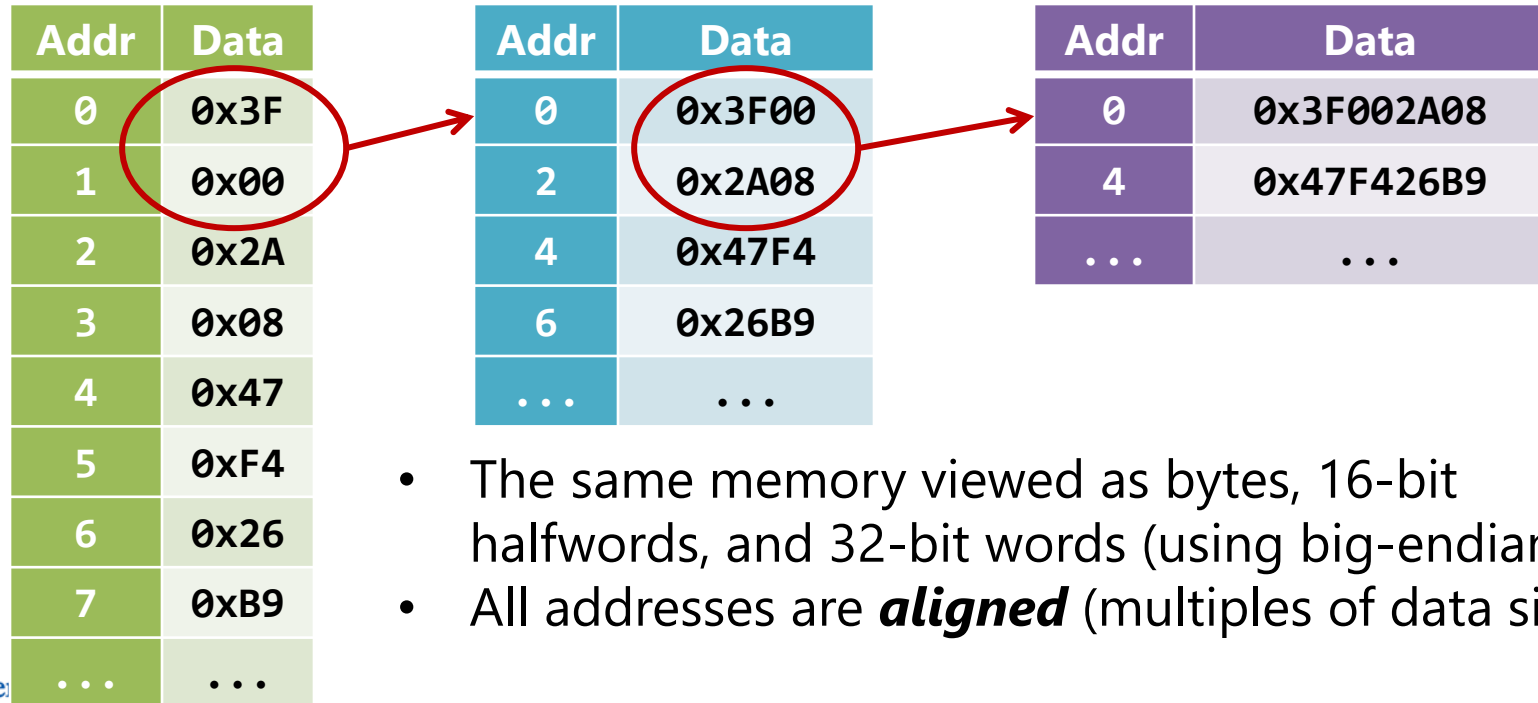
- The same memory viewed as bytes, 16-bit halfwords, and 32-bit words (using big-endian)
- All addresses are ***aligned*** (multiples of data size)

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

lw $t0, 8($s4)

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

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

sw $t0, 12($s4)

$t0 is the SOURCE!
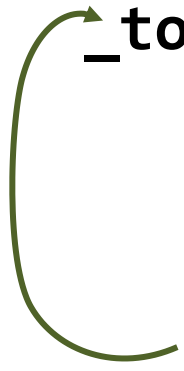
0	0x3F002A08
4	0x47F426B9
8	0x00000000
12	0x0000BEEF
16	0x0000BEEF
...	...

Memory

t0	0x0000BEEF
s4	0x00000004

Registers

lw

sw

$s4 + 8

$s4 + 12
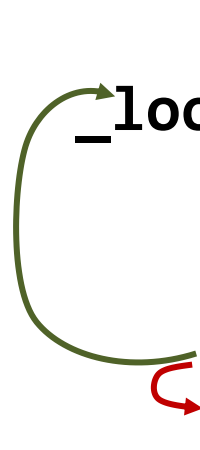
● Jump and branch instructions change the flow of execution.

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

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

j : jumps *unconditionally*
• jumps to **_top**

bne : jumps *conditionally*
If **$s0** != **$zero**, jumps to **_loop**
If **$s0** == **$zero**, continues to **jr $ra**

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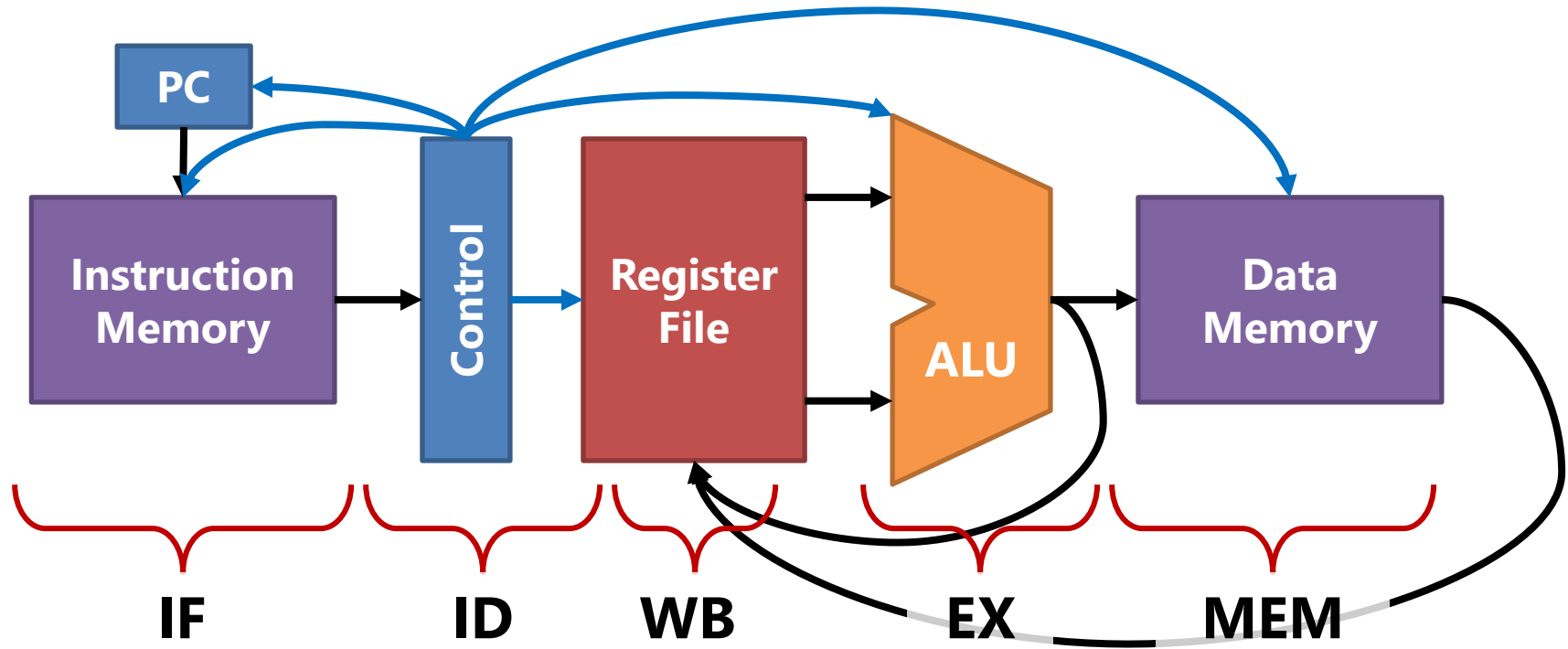
- An instruction goes through IF/ID/EX/MEM/WB in one cycle

"Minimal MIPS"

It's a "subset" of MIPS
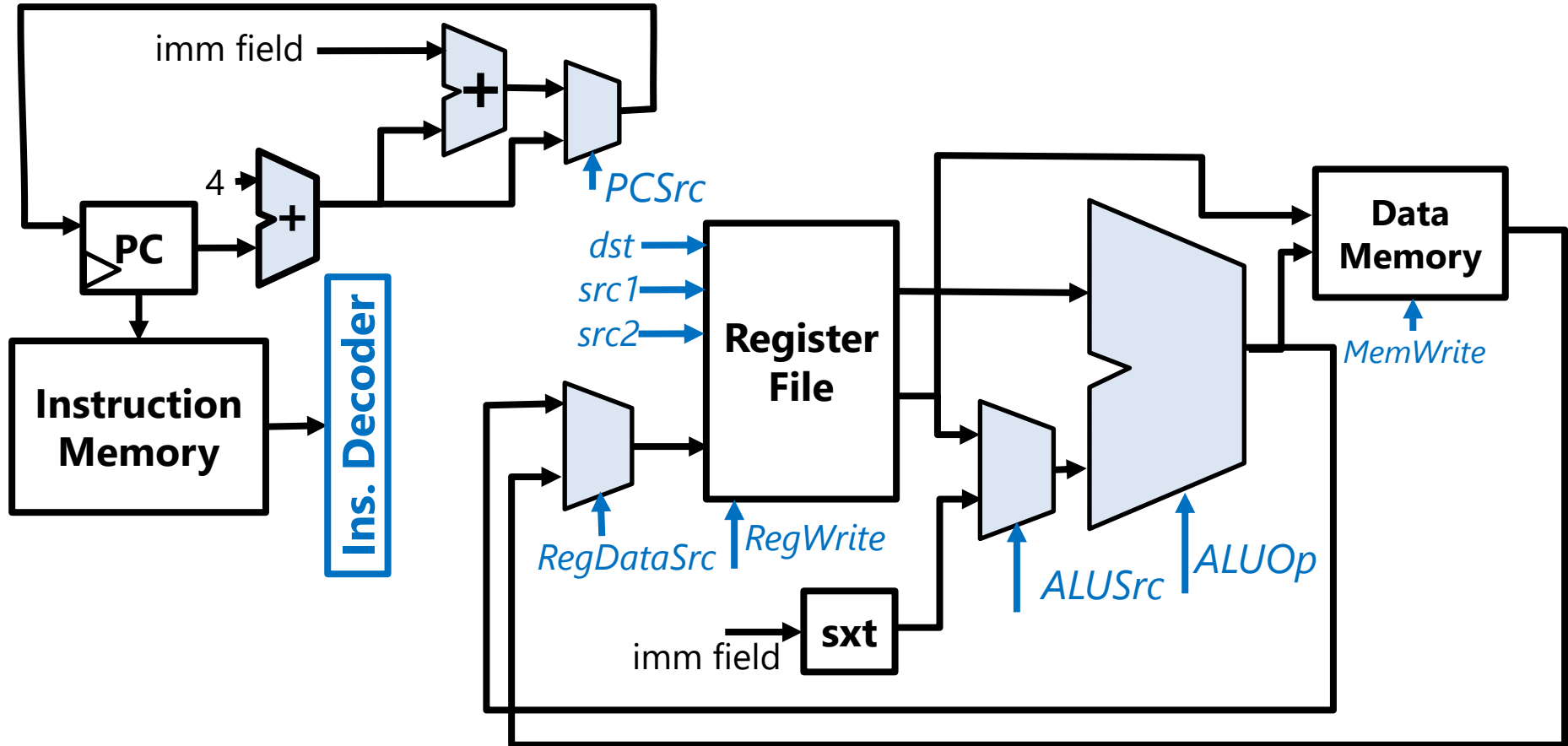
- For pedagogical (teaching) purposes

- 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

- 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!

add t0, t3, s0

lw s4, 12(s0)

sw t3, 8(sp)
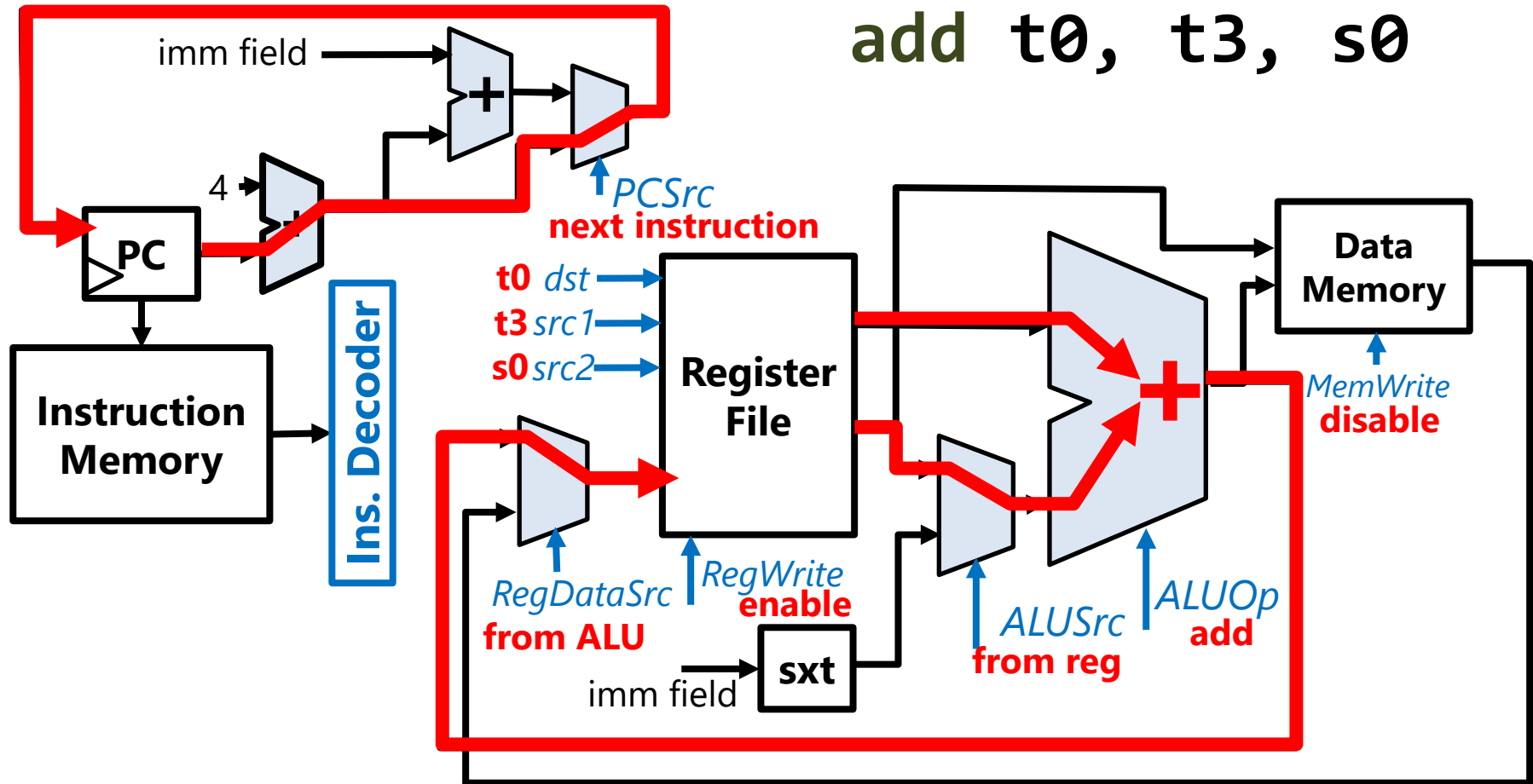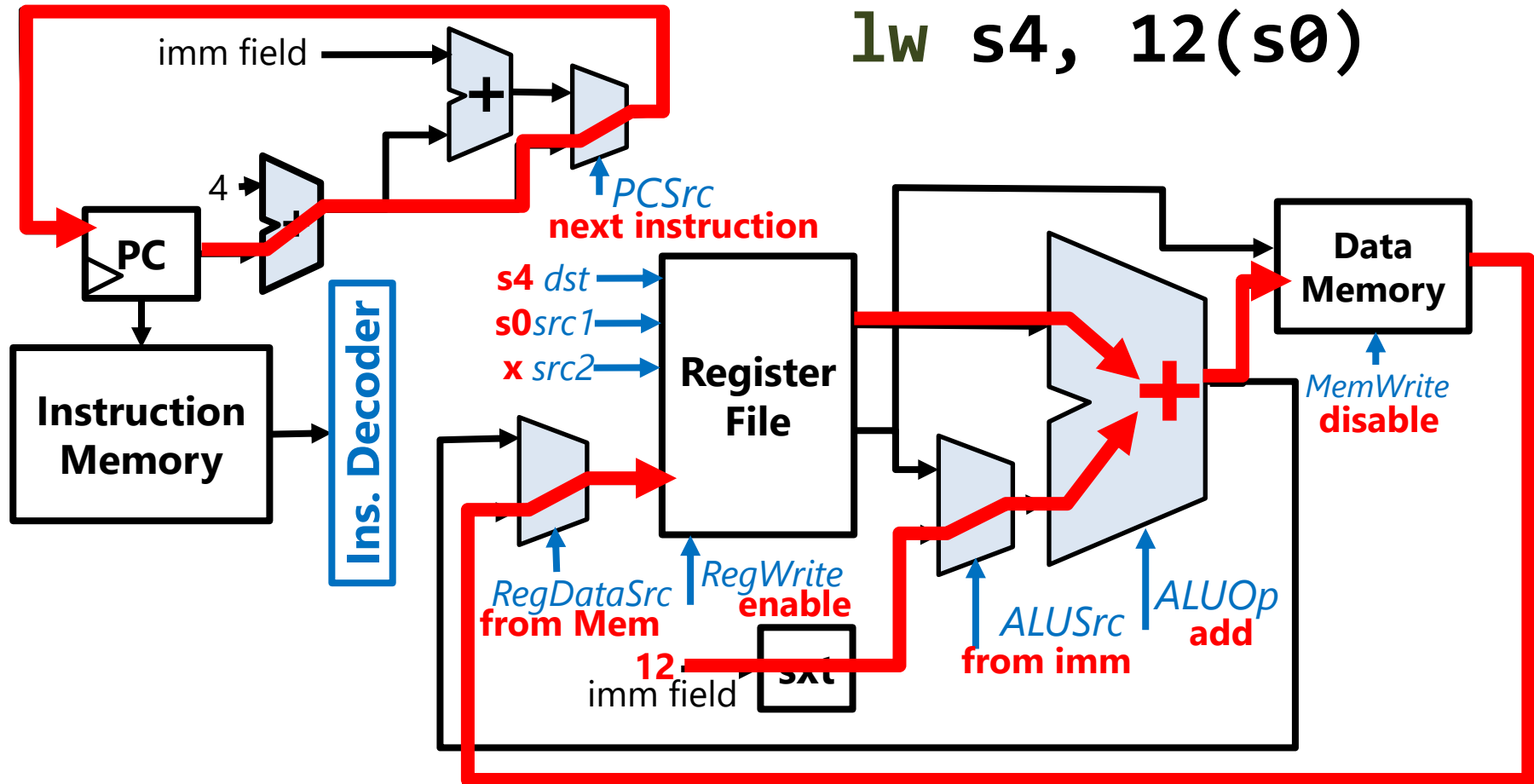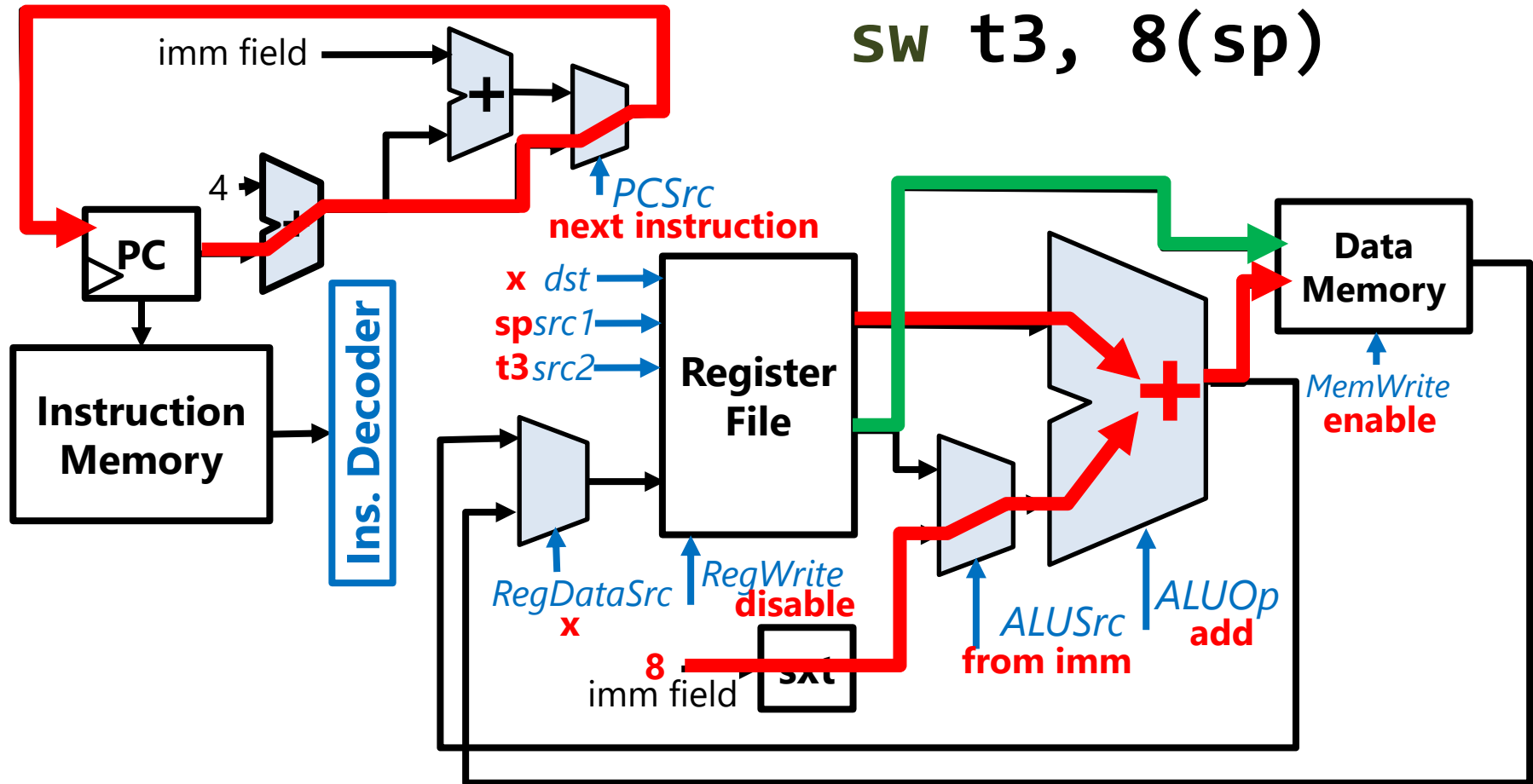
- Compares numbers by subtracting and see if result is 0
  - If result is 0, we set PCSrc to use the branch target.
  - Otherwise, we set PCSrc to PC + 4.

isBEQ

PCSrc

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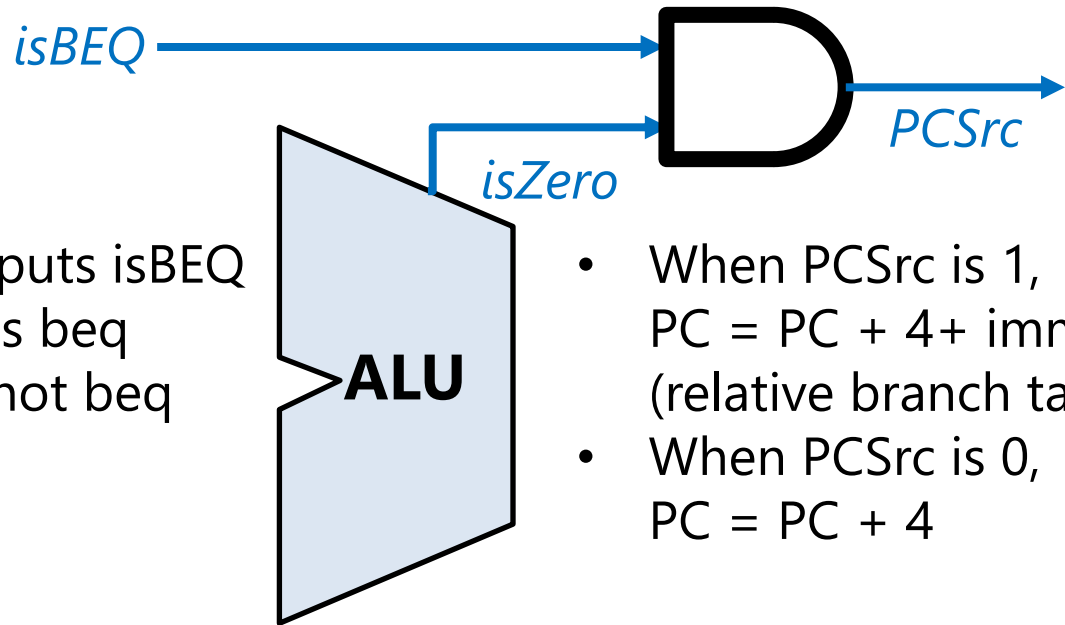
- Instruction decoder outputs isBEQ
  - 1: When instruction is beq
  - 0: When instruction not beq

ALU

- When PCSrc is 1, PC = PC + 4+ imm (relative branch target)
- When PCSrc is 0, PC = PC + 4

University of
Pittsburgh

23

beq t0, t1, loop

Take green PC path when t0 == t1
Take red PC path when t0 != t1
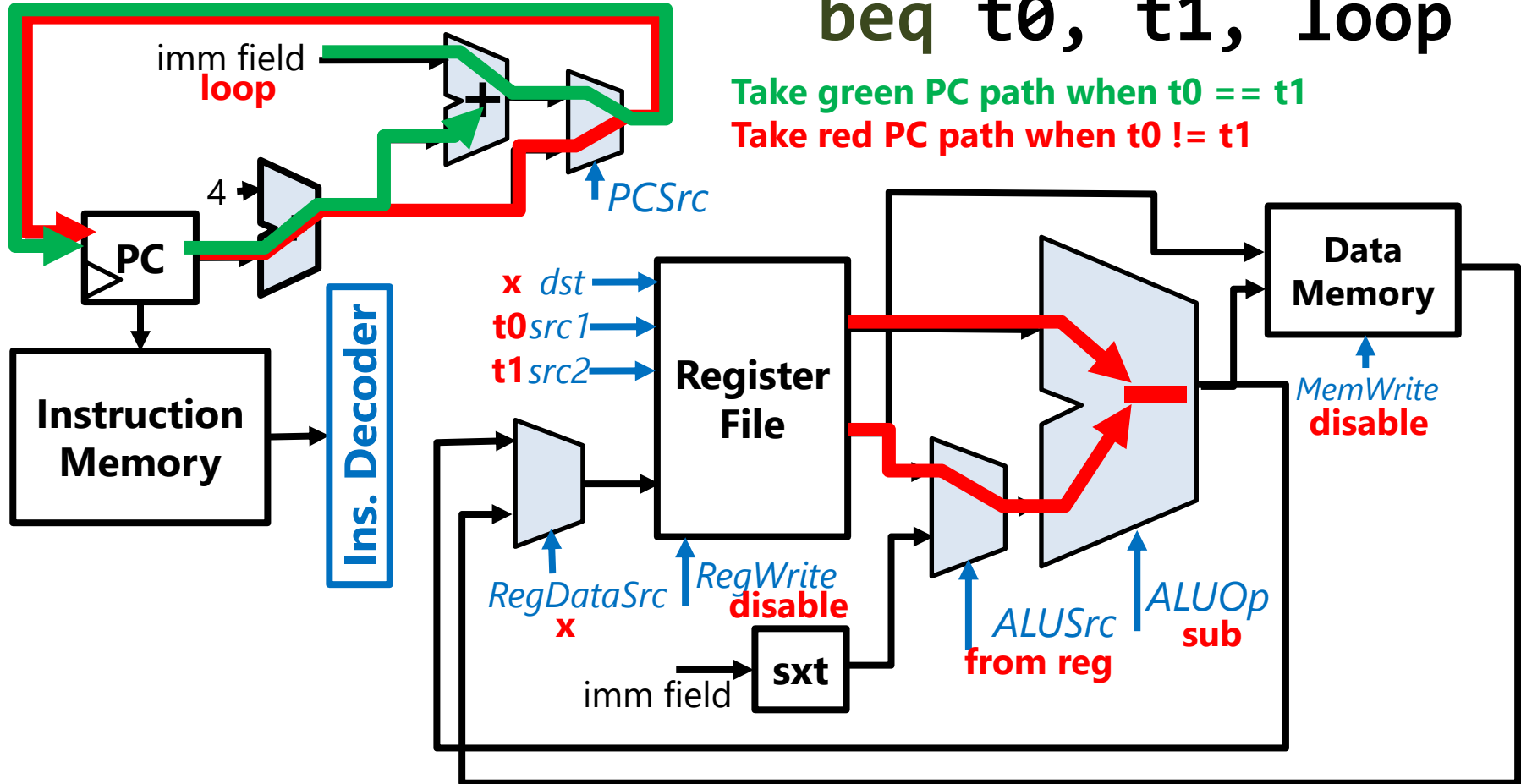
- We have to add another input to the PCSrc mux.

`j   top`

PC+4

PC+4+imm

jump target

(now 2 bits)

PCSrc

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

- 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 add is 99% of instructions!

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

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

Memory → | → | → Ins. Decoder → Register File → | → ALU → | → Memory → |

WB
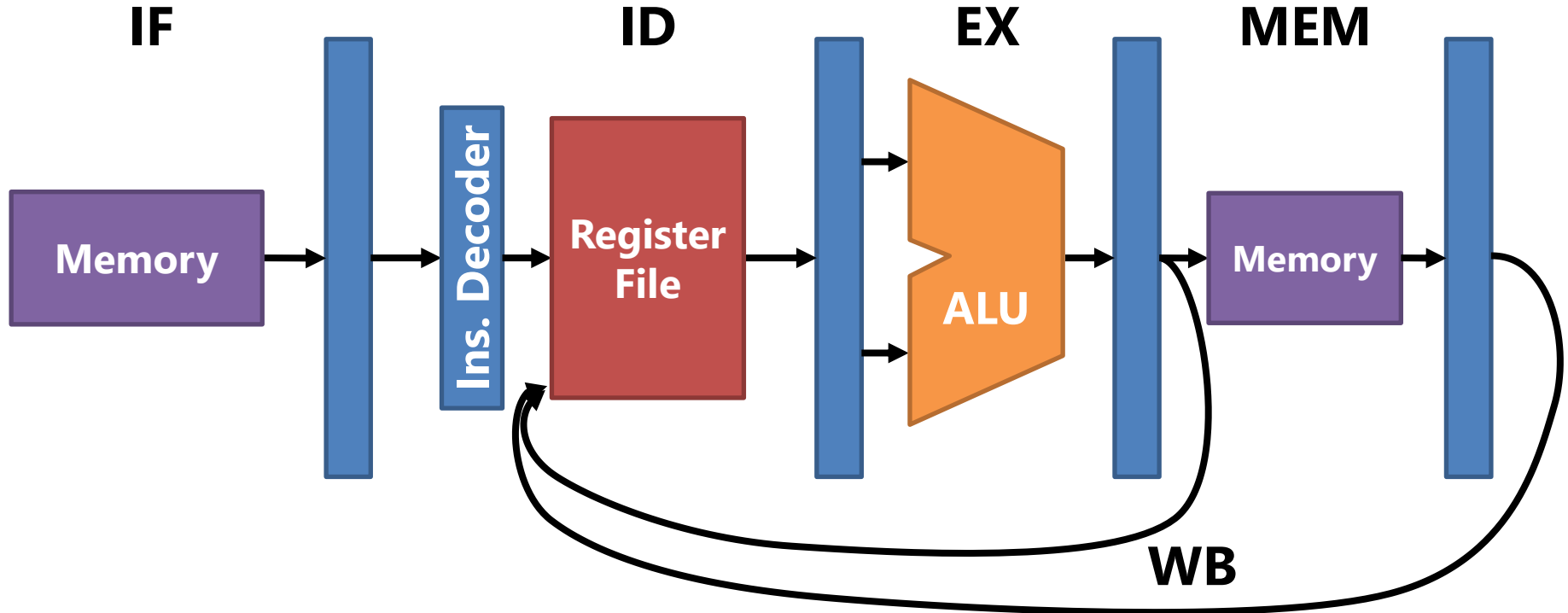
A Multi-cycle Implementation is Faster!

- 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 ns* each as before:
  - **lw** takes *5 ns* and **add** takes *4 ns*

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

A) 0.01 * *5 ns* + 0.99 * *4 ns = 4.01 ns* (25% faster than single cycle)

Caveat: there is some delay due to the added latches but net win

- 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)

University of Pittsburgh

Pipelining Basics

- If washer works on next load immediately after passing on to dryer
  → Washer gets higher utilization → Improves overall throughput!

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

IF   **Clock!**   **ID**   **Clock!**   **EX**   **Clock!**   **MEM**

add

Set all control signals...

Add...

Memory

Ins. Decoder

Register File

ALU

Memory

Clock!

Data flows back to registers...

WB

University of Pittsburgh

33
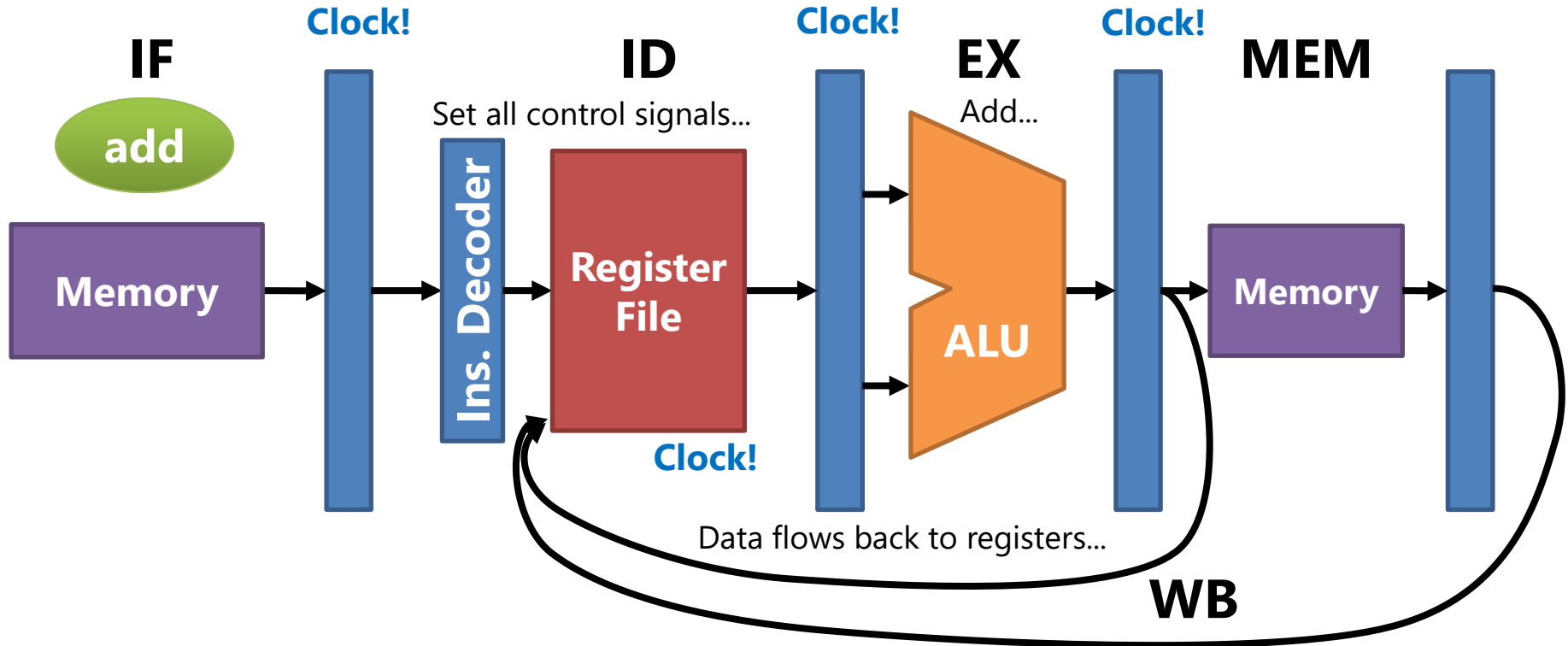
- Pipelining is just an extension of that idea!

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

SW

Memory

Ins. Decoder

Register File

ALU

Memory

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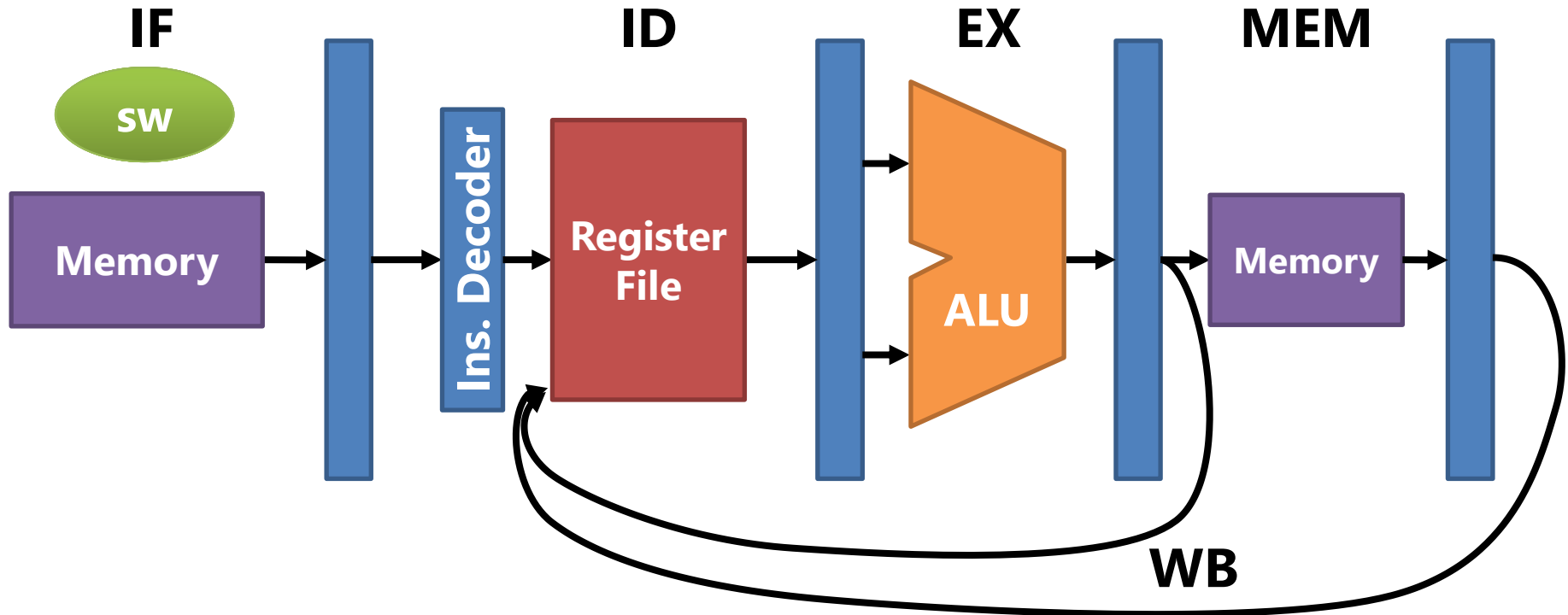
- This type of parallelism is called *pipelined parallelism.*

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

- Again each instruction takes different number of cycles to complete
  - o **lw** takes 5 cycles: IF/ID/EX/MEM/WB
  - o **add** takes 4 cycles: IF/ID/EX/WB

- If each stage takes *1 ns* each:
  - o **lw** takes *5 ns* and **add** takes *4 ns*

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

A) (*99 ns + 5 ns*) / 100 = *1.04 ns*
  - o Assuming last instruction is a **lw** (a 5-cycle instruction)
  - o A ~**5X** speed up from single cycle!

- What happened to the three components of performance?

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

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
  - o Or in other words CPU **clock frequency**
  - o The deeper the pipeline, the higher the frequency will be

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

* Caveat: pipeline hazards and memory delay can increase CPI

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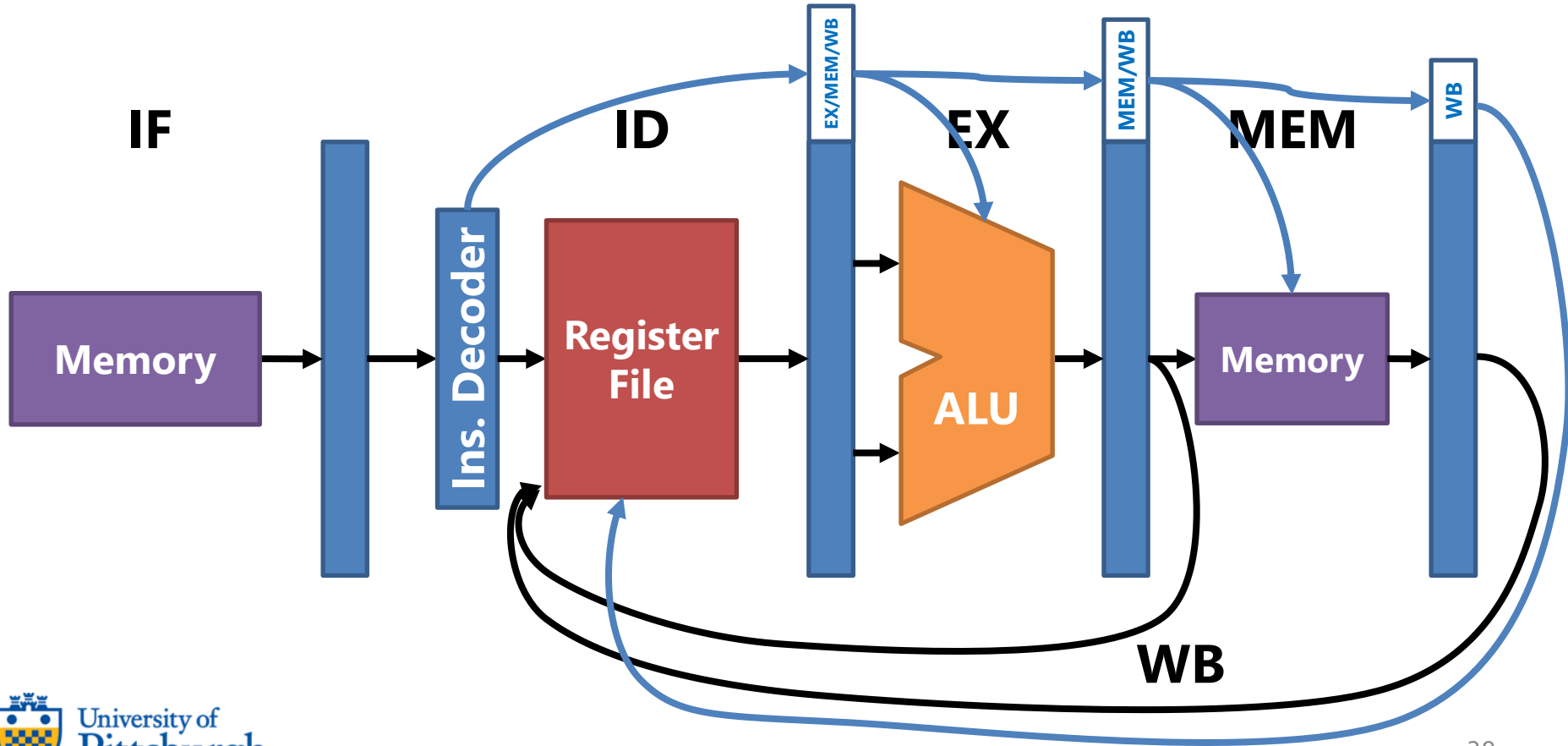
Pipeline Hazards

- 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** or **stalls** 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*

University of Pittsburgh

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($0)	IF	ID	EX	MEM	WB			
lw t1,4($0)		IF	ID	EX	MEM	WB		
lw t2,8($0)			IF	ID	EX	MEM	WB	
lw t3,12($0)				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		
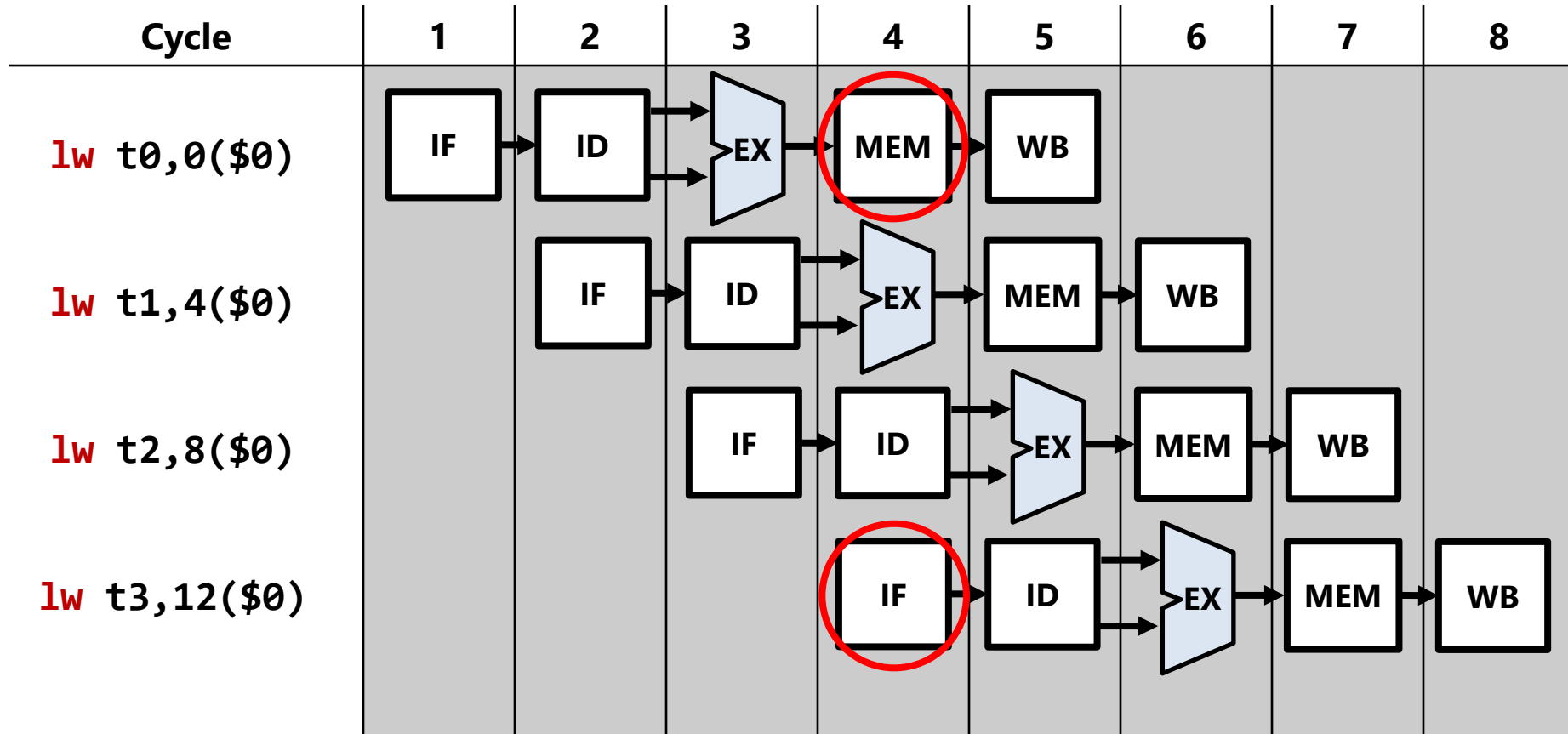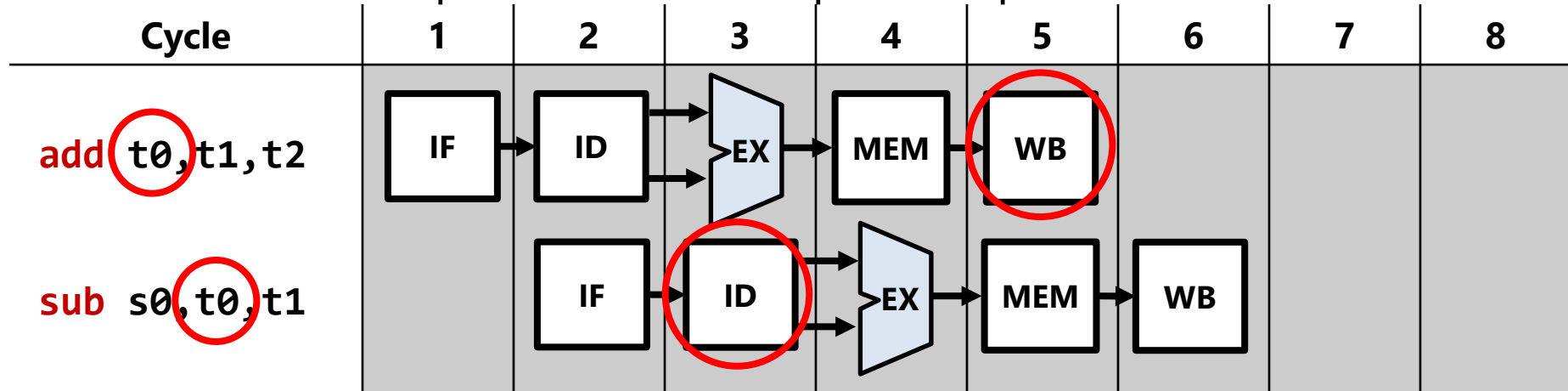
- **sub** must wait until **add**'s WB phase is over before doing its ID phase

add t0,t1,t2	IF	ID	EX	MEM	WB			
sub s0,t0,t1		bubble	bubble	bubble	IF	ID	EX	MEM
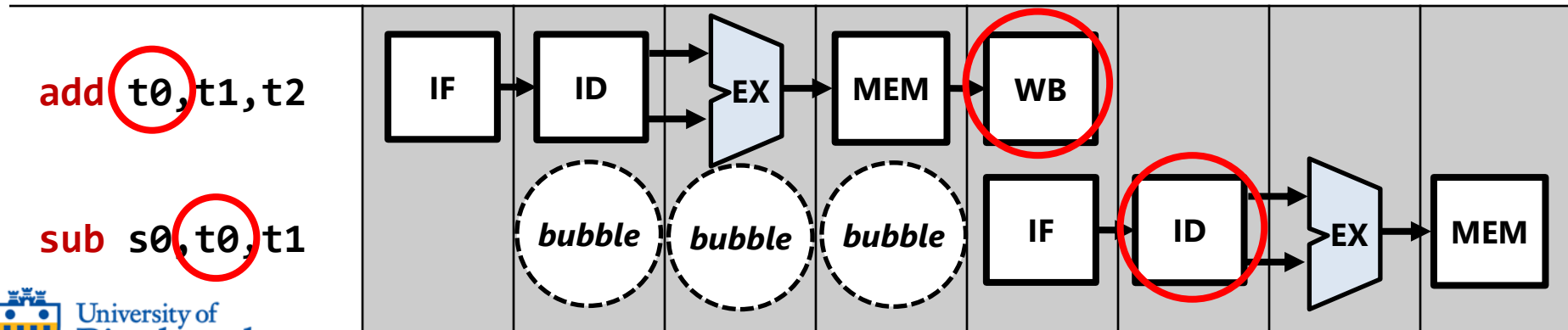
● You don't know the outcome of a conditional branch.

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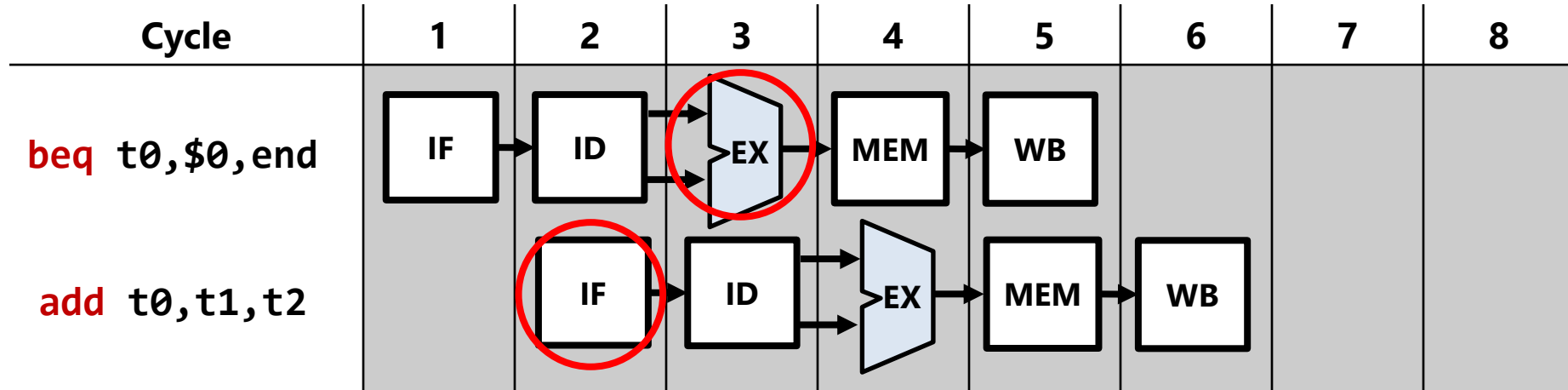
● **add** must wait until **beq**'s EX phase is over before its IF phase

beq t0,$0,end — IF, ID, EX, MEM, WB

add t0,t1,t2 — bubble, bubble, IF, ID, EX, WB

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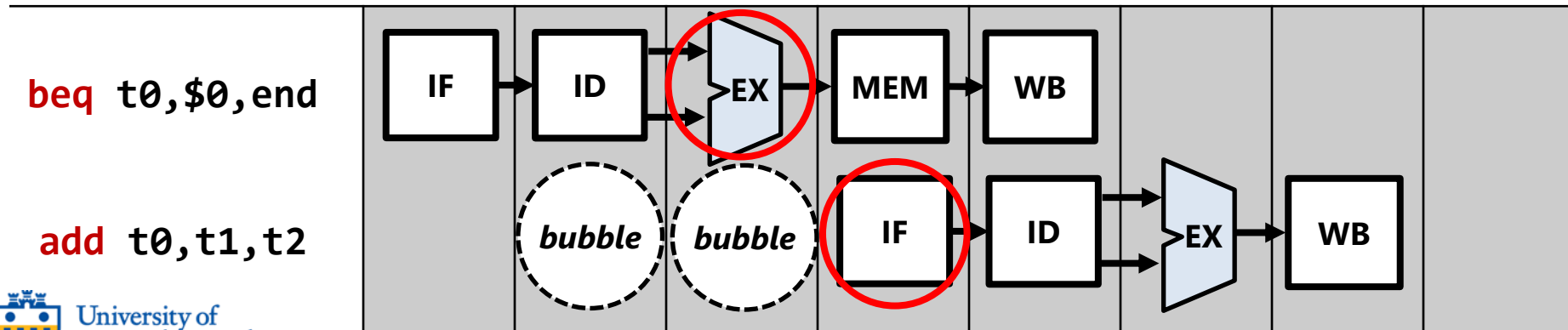
- Pipeline must be controlled so that hazards don't cause malfunction

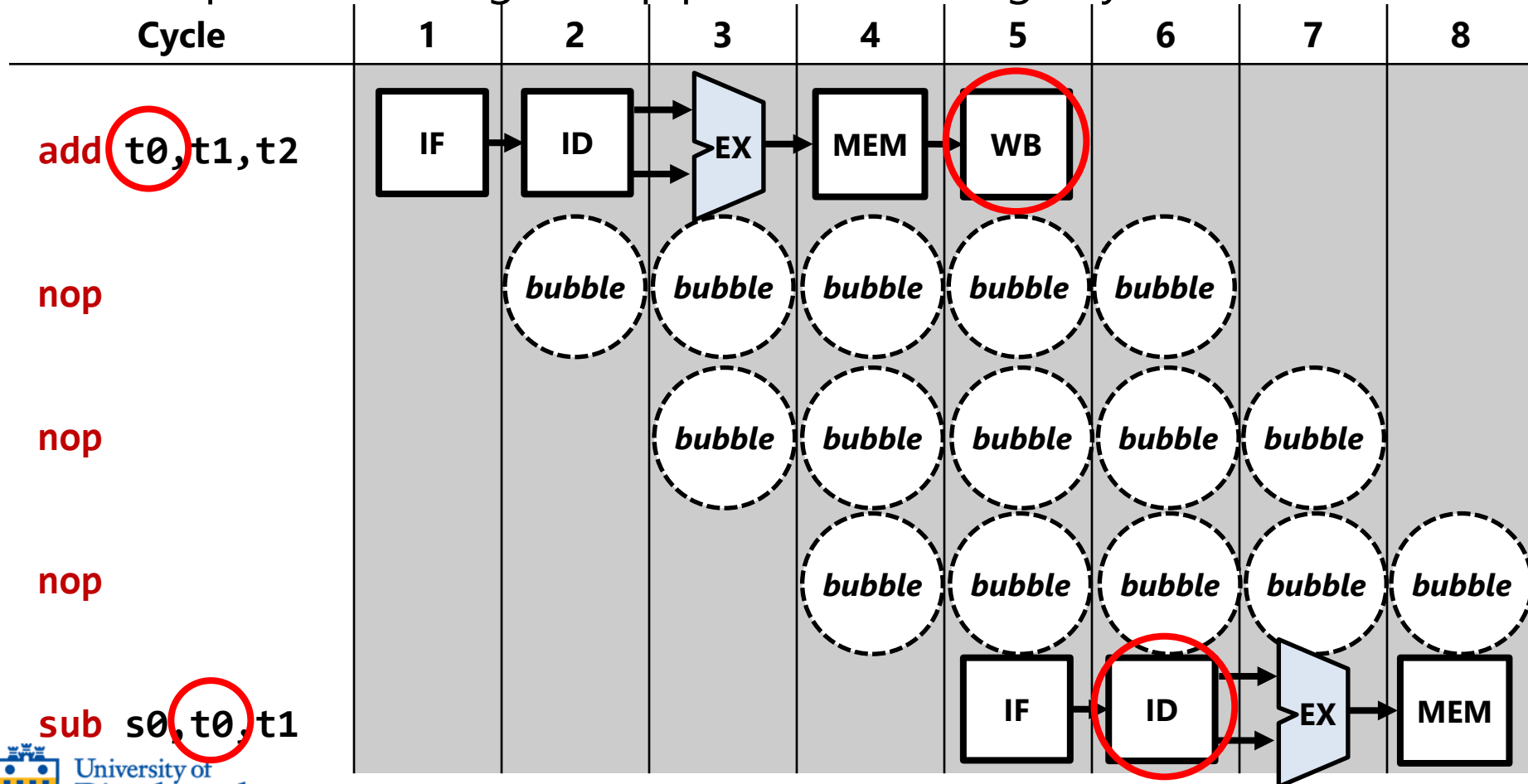- Who is in charge of that? You have a choice.

  1. Compiler can avoid hazards by inserting nops
     - Insert 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
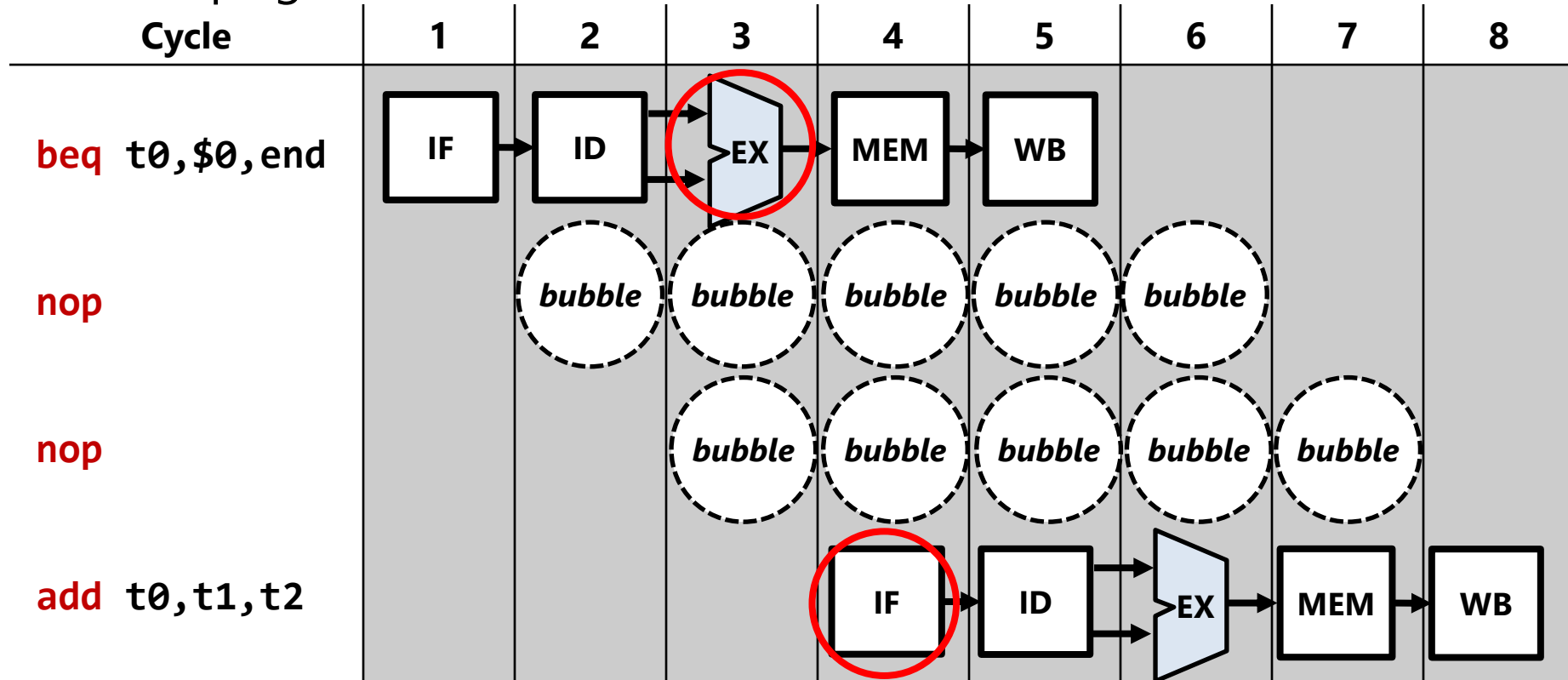
Compiler avoiding a data hazard

- The nops flow through the pipeline not doing any work

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
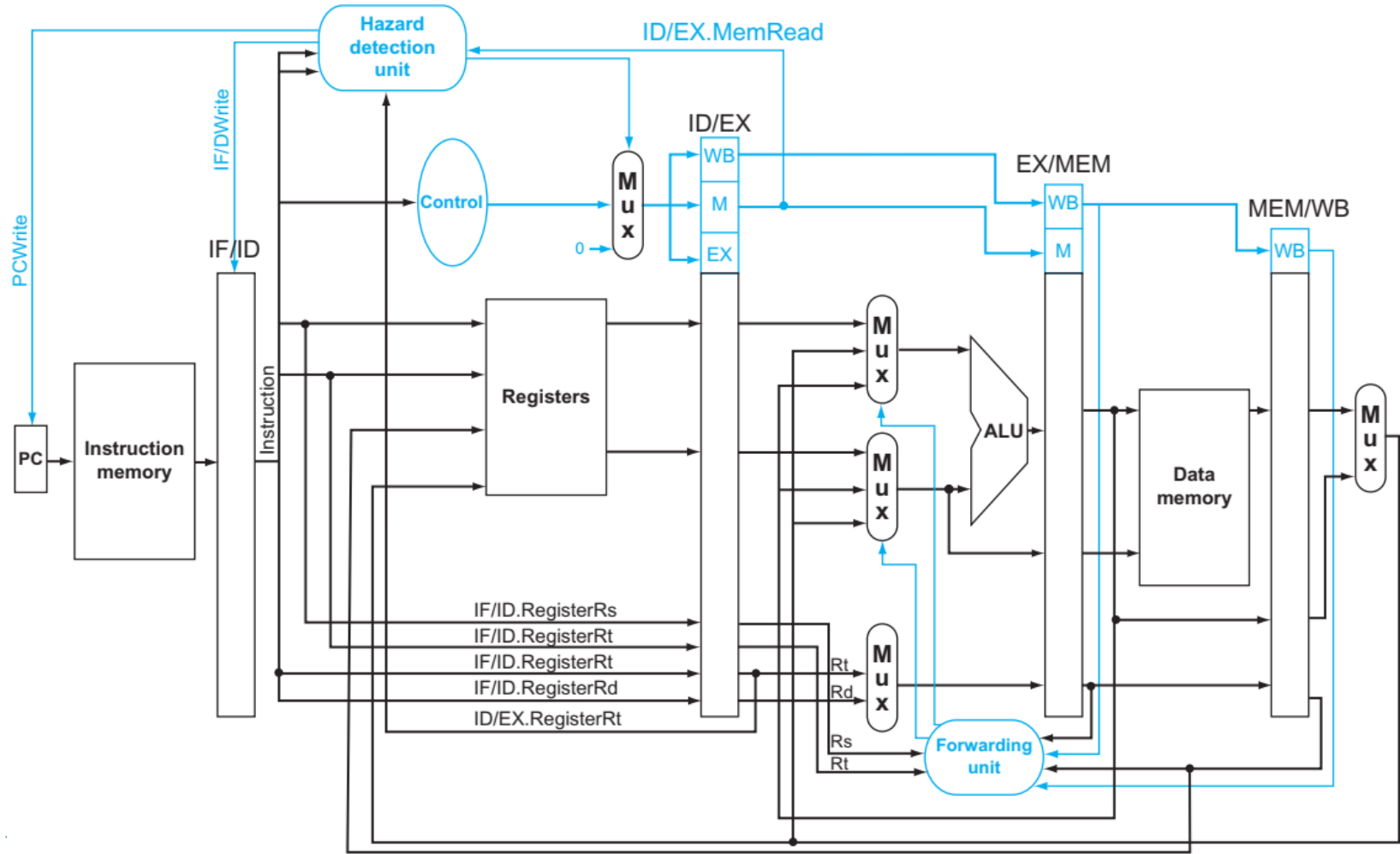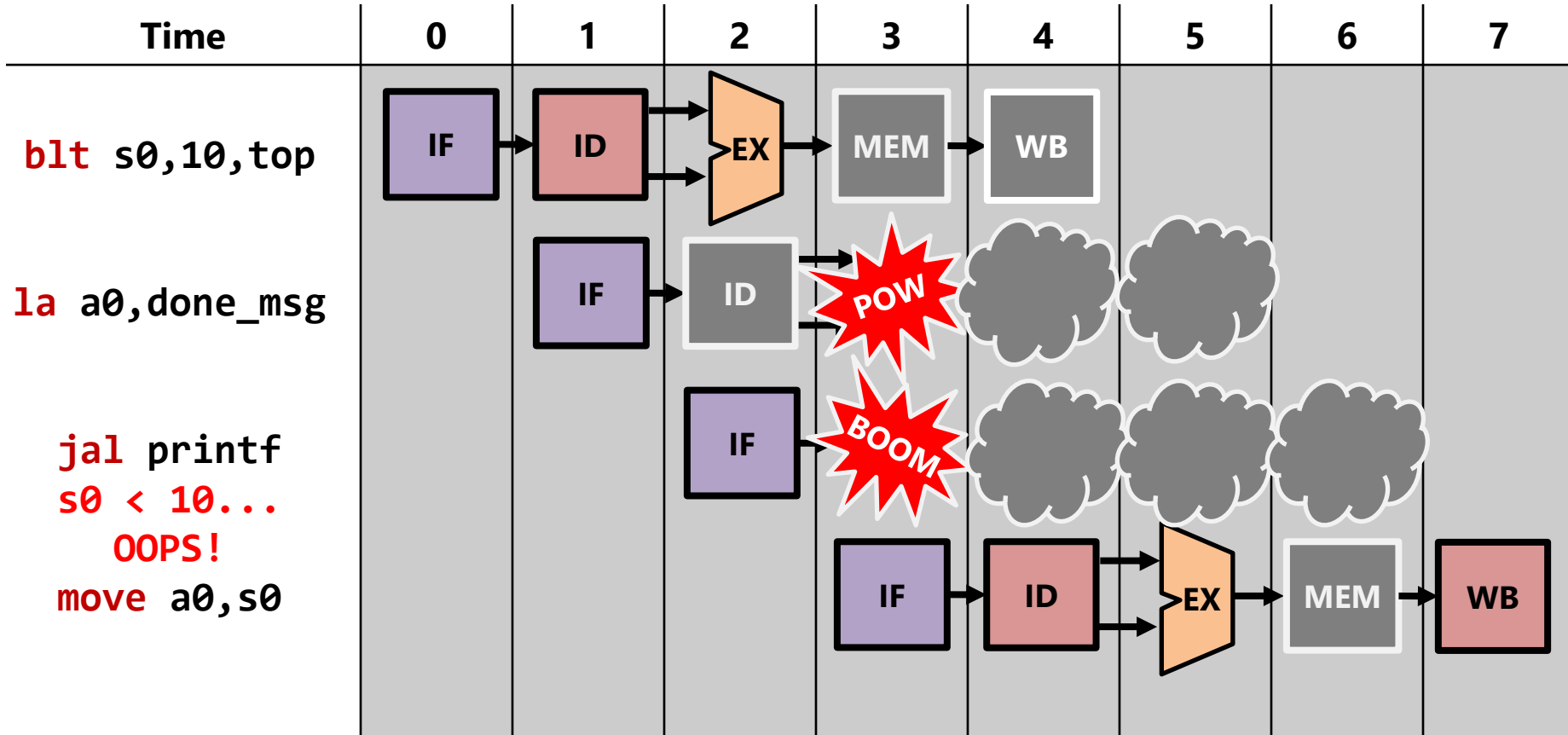
Hazard Detection Unit

- Suppose we have an **add** that depends on an **lw**.
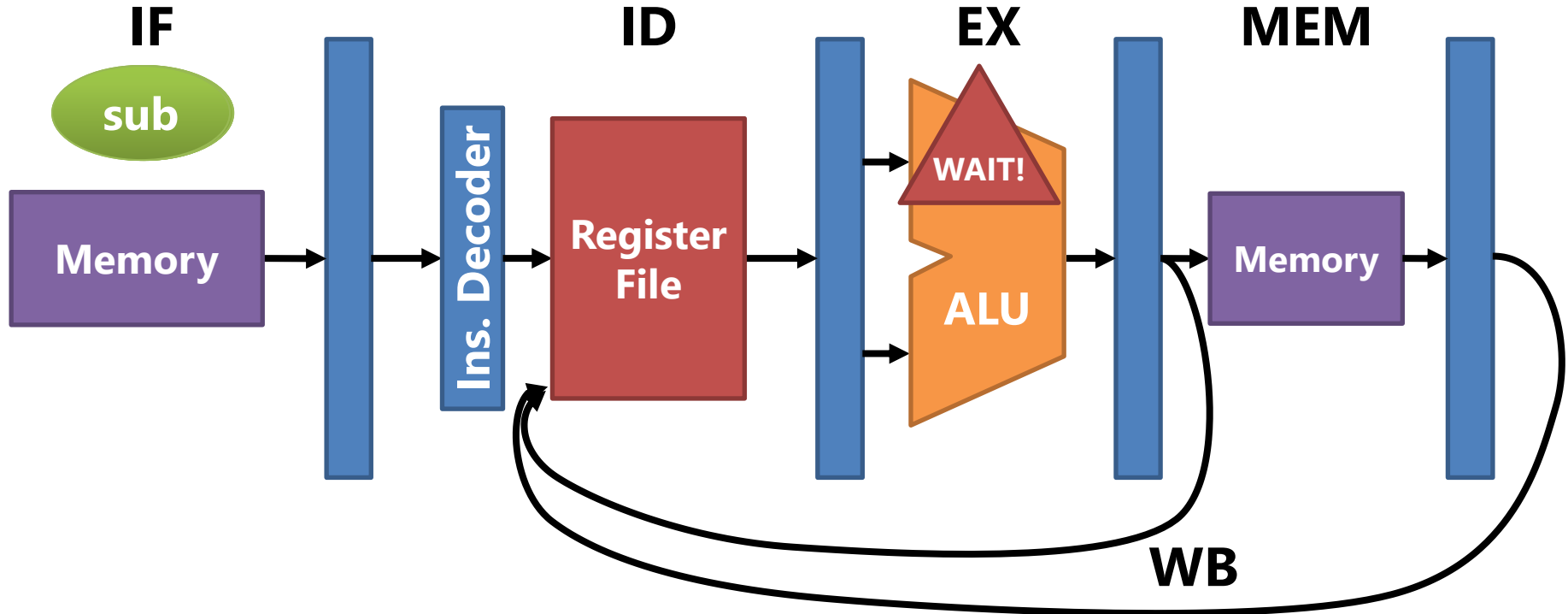
- If HDU detects a structural or data hazard, it does the following:
  - It **stops fetching instructions** (doesn't update the PC).
  - It **stops clocking the pipeline registers for the stalled stages.**
  - The stages after the stalled instructions **are filled with nops.**
    - Change control signals 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!
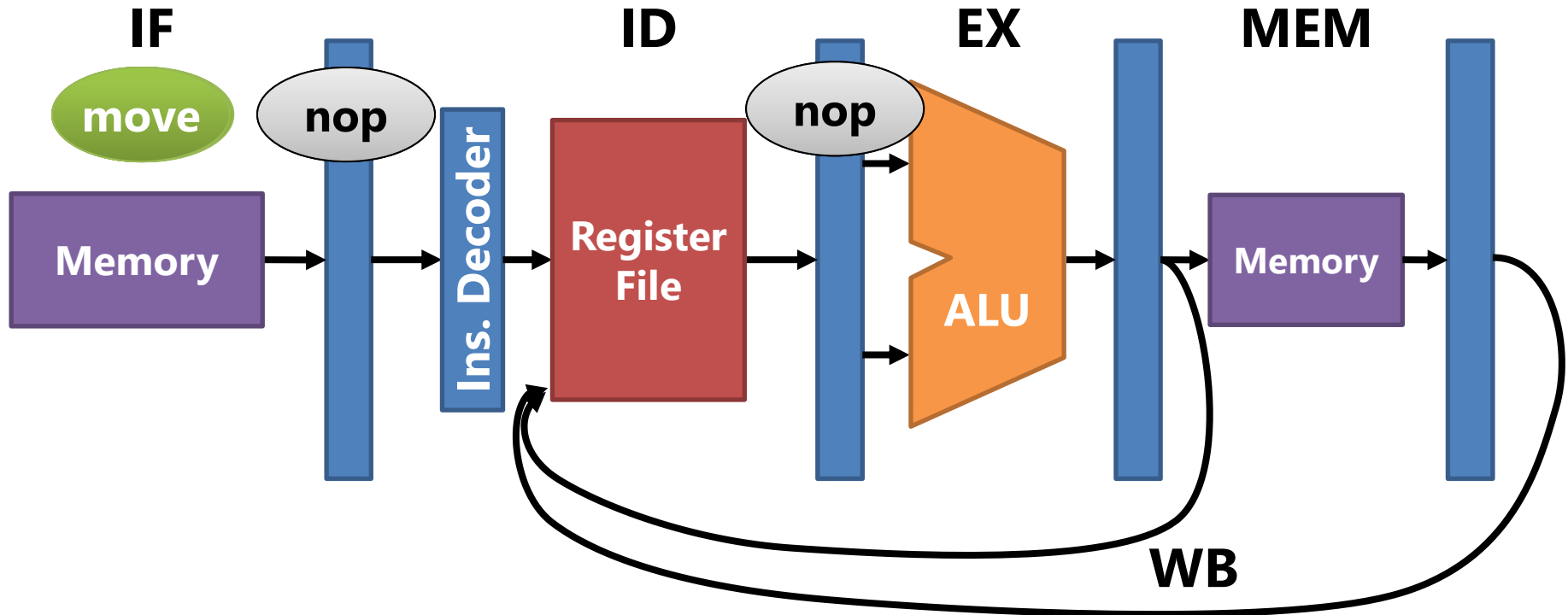 - Need to *flush* these instructions

- A pipeline flush removes all wrong path instructions from pipeline

- Let's watch the previous example.

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

move nop nop

Ins. Decoder

Memory

Register File

ALU

Memory

WB

- 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.

- Remember the three components of performance:

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

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!

University of Pittsburgh

53

- 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
    - Did you see all those long wires going to all parts of the CPU?
    - Can cause problems with the **Power Wall**
  - Compiler scheduling via nops removes need of HDU

University of Pittsburgh

Solving Structural Hazards

Structural Hazard on Memory

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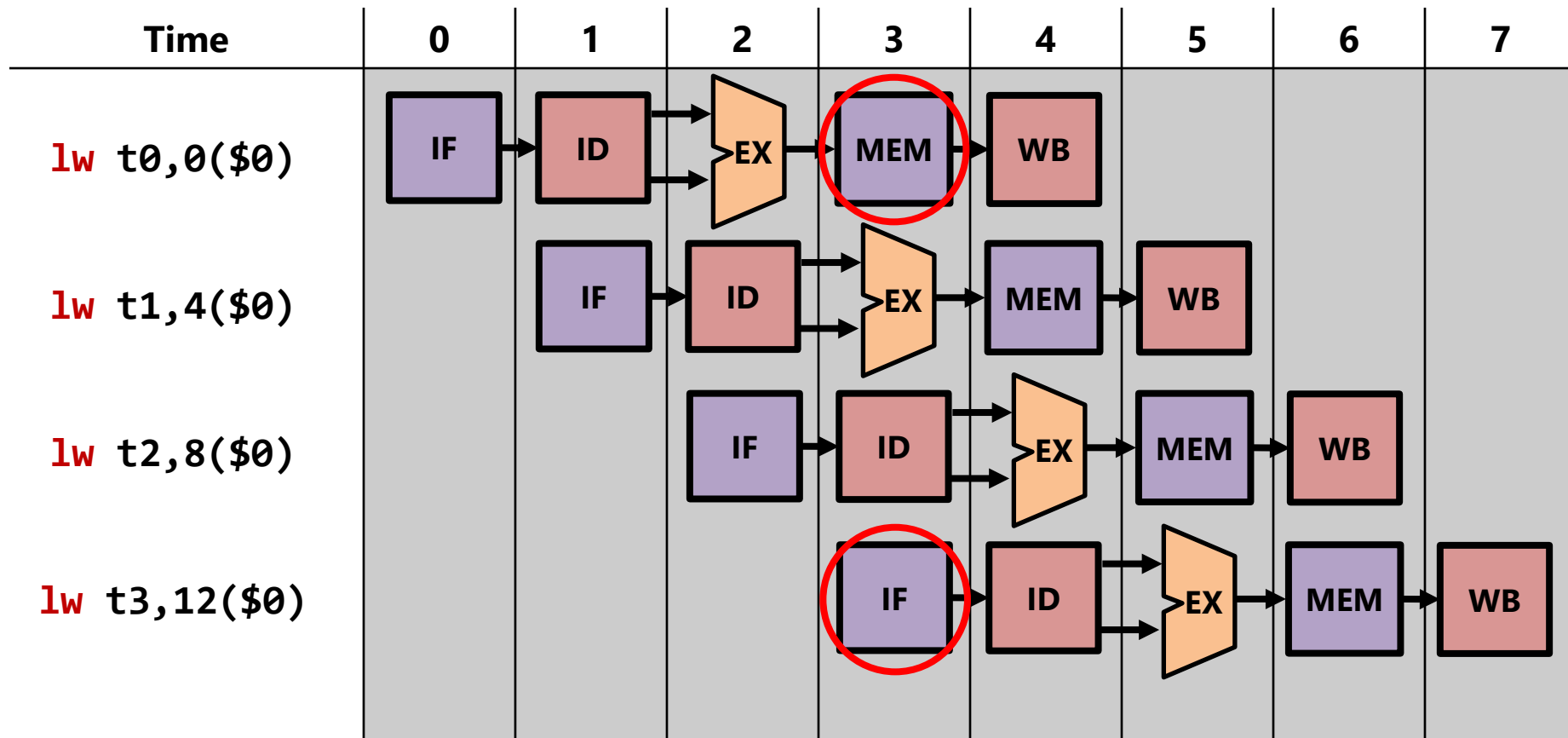
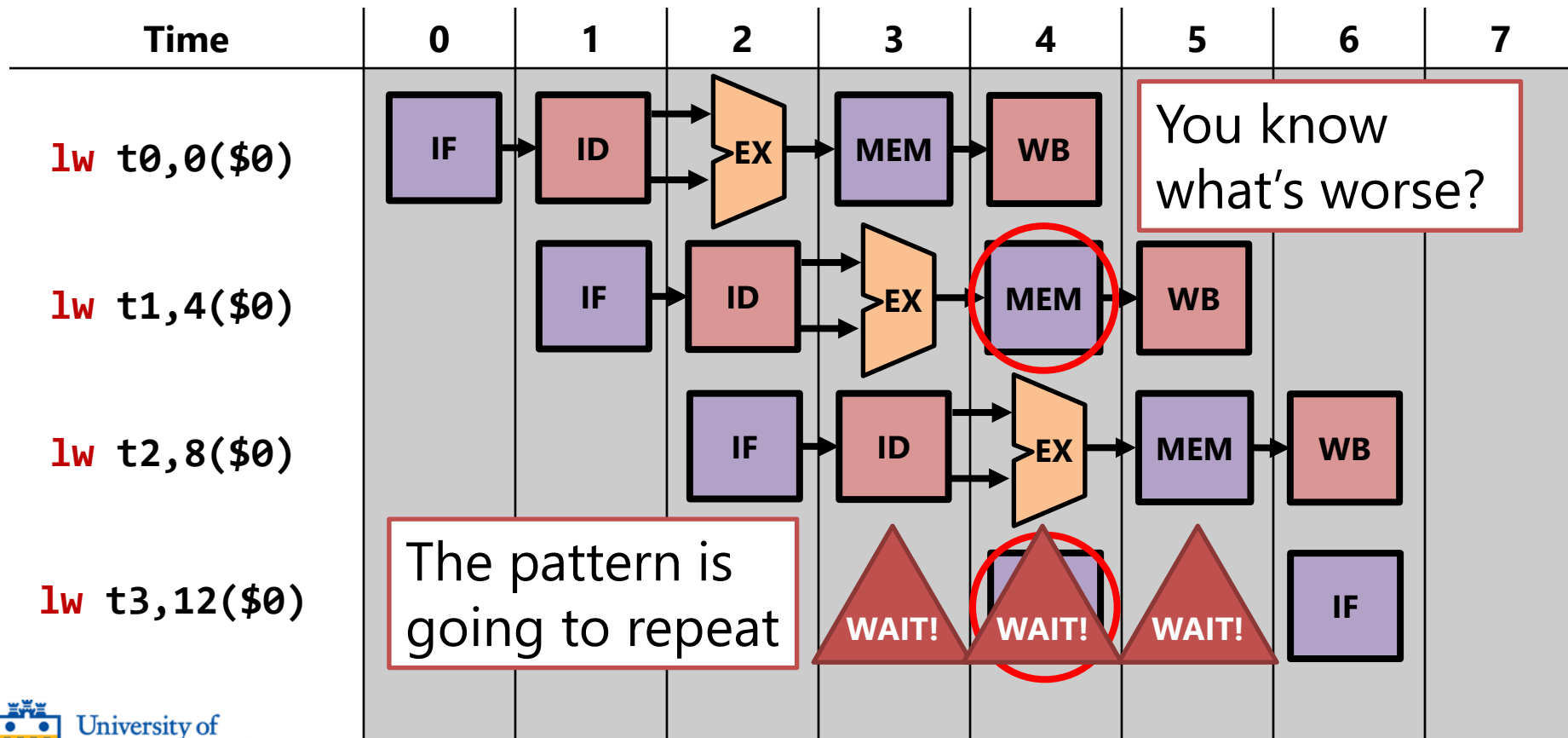- Two people need to use **one** sink at the same time
  - Well, in this case, it's memory but same idea

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

Time	0	1	2	3	4	5	6	7

lw t0,0($0)

IF → ID → EX → MEM → WB

You know what's worse?

lw t1,4($0)

IF → ID → EX → MEM → WB

lw t2,8($0)

IF → ID → EX → MEM → WB

lw t3,12($0)

The pattern is going to repeat

WAIT! WAIT! WAIT! IF

- 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!
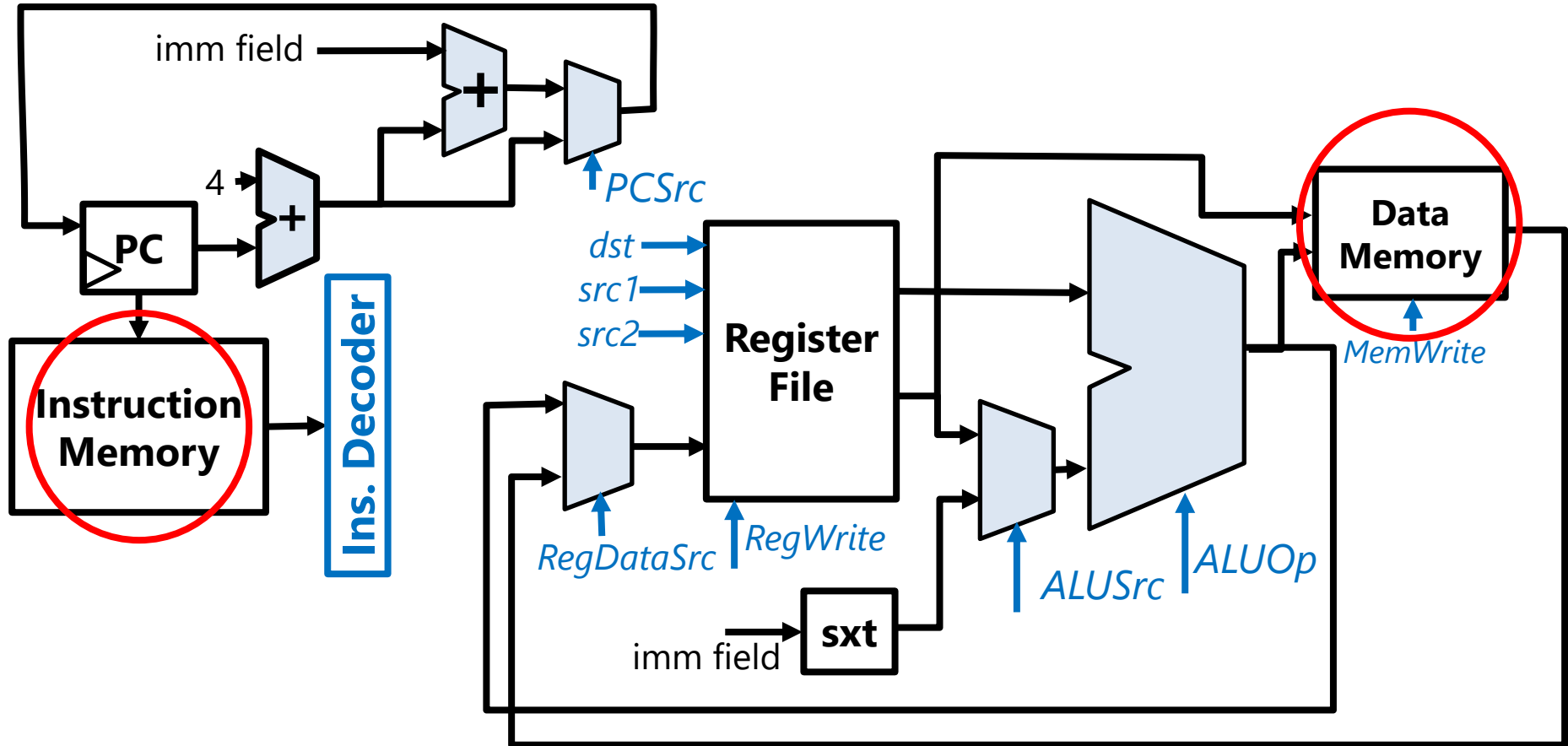 - Not literally but CPUs have separate *instruction* and *data caches*

- With separate i-cache and d-cache, MEM and IF can work in parallel
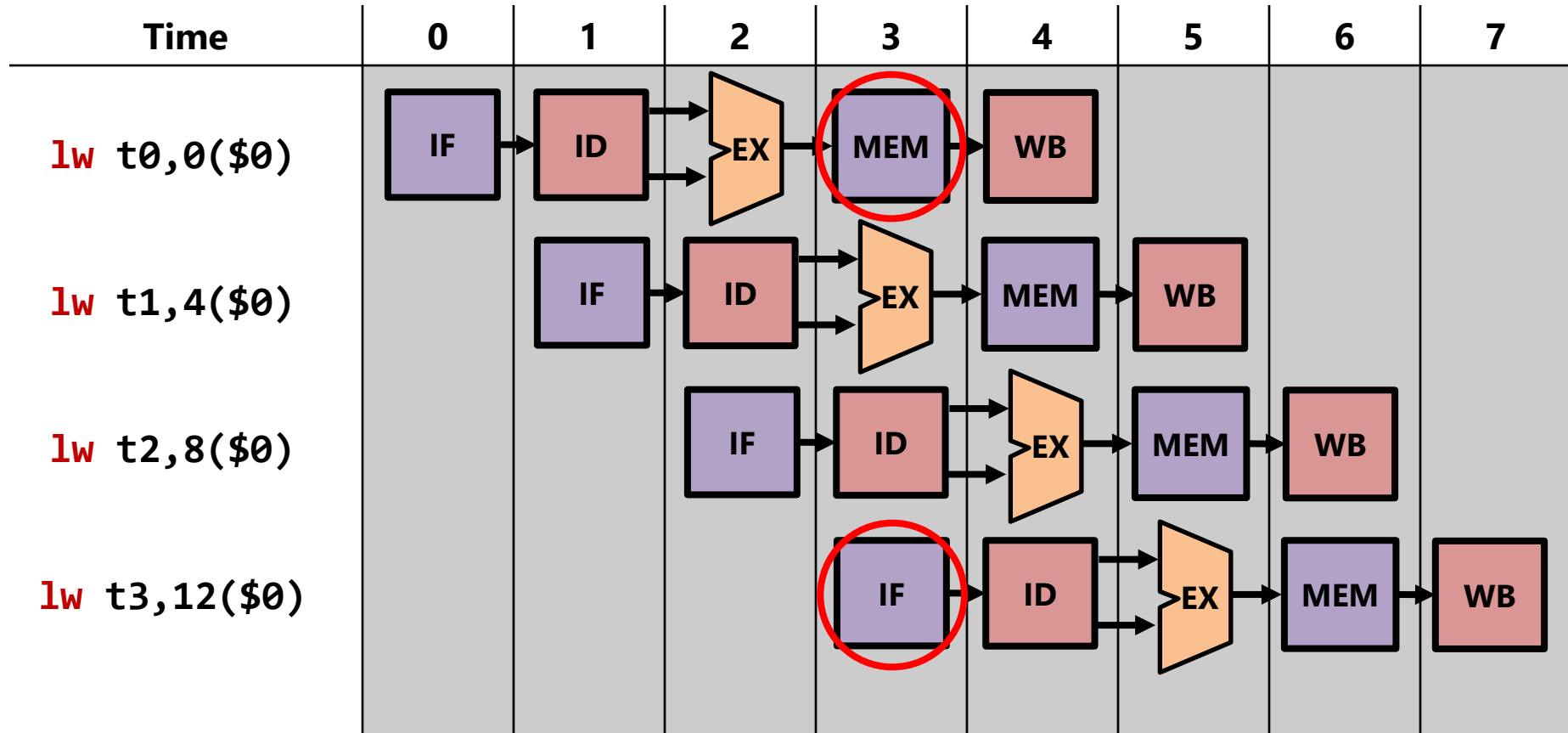
- But is that the only hardware duplication going on here?

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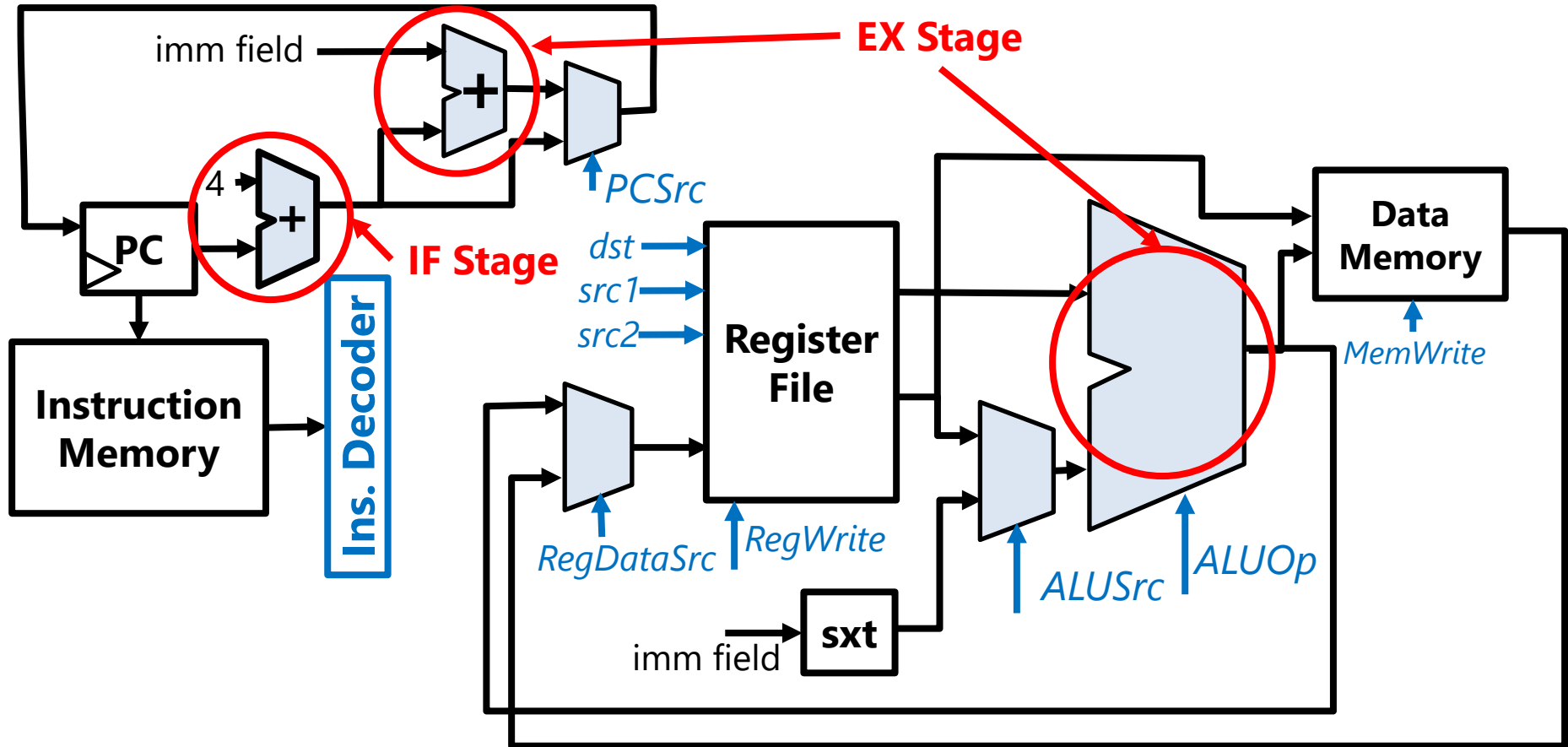
Solving Structural Hazards

● There are mainly two ways to throw more hardware at the problem

1.   Duplicate contentious resource
    o One memory cannot sustain one MEM / one IF stage per cycle
        → Duplicate into one instruction memory, one data memory
    o One ALU cannot sustain one IF / one EX stage per cycle
        → Duplicate into one ALU and two simple adders

2.   Add ports to a single shared (memory) resource
    o **Port**: Circuitry that allows either read or write access to memory
    o If current number of ports cannot sustain rate of access per cycle
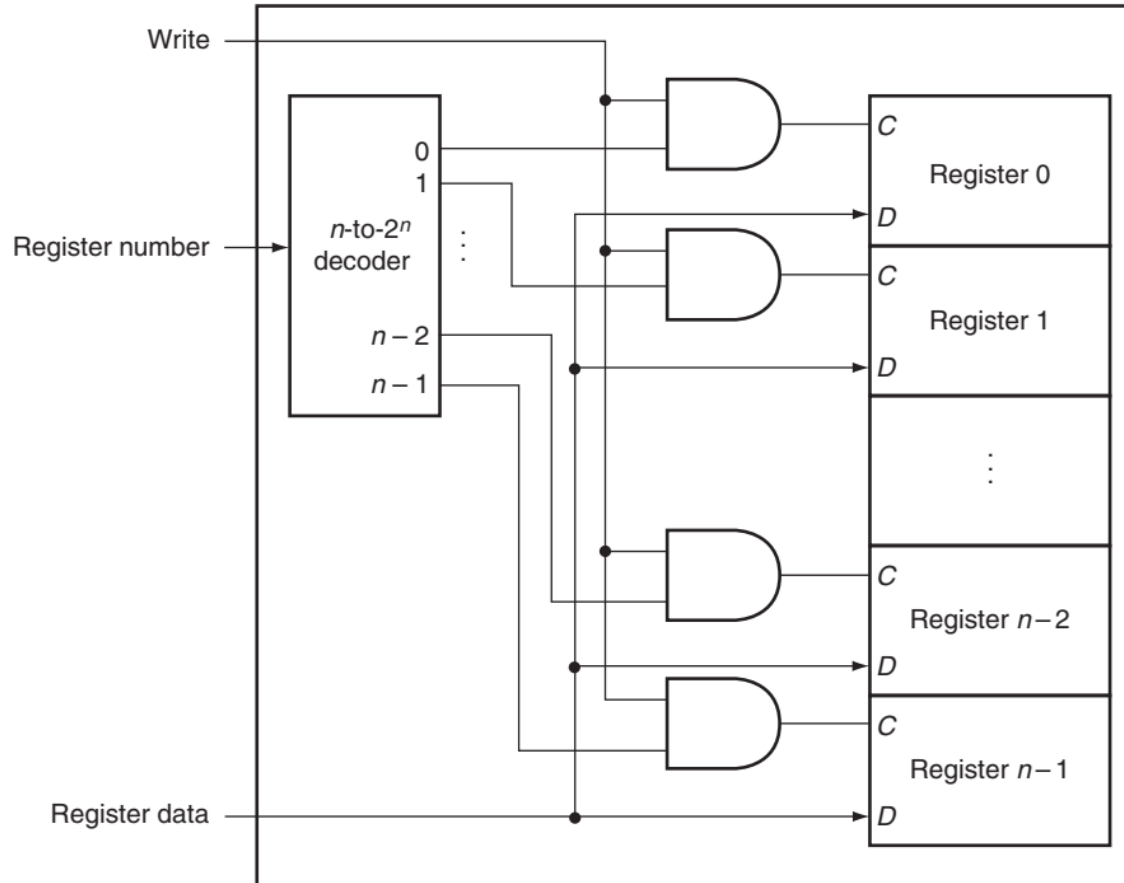   → Add more ports to memory structure for simultaneous access

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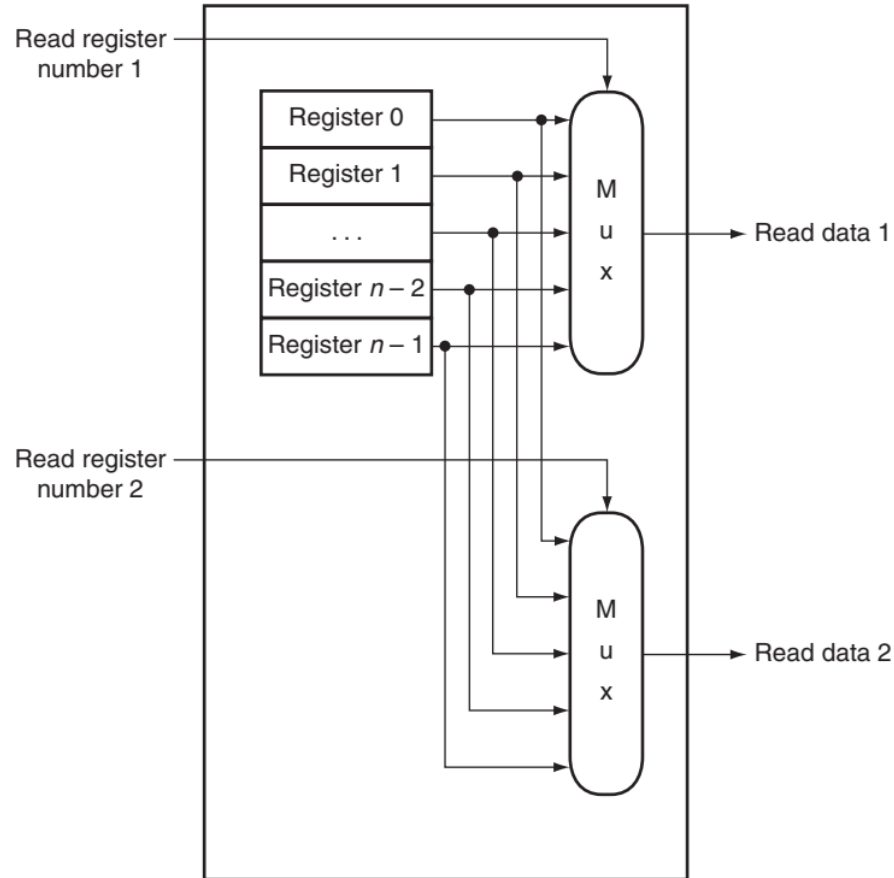
- By adding more decoders, you can add more write ports

- With two read ports and one write port
  - Enough to sustain one ID / 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!

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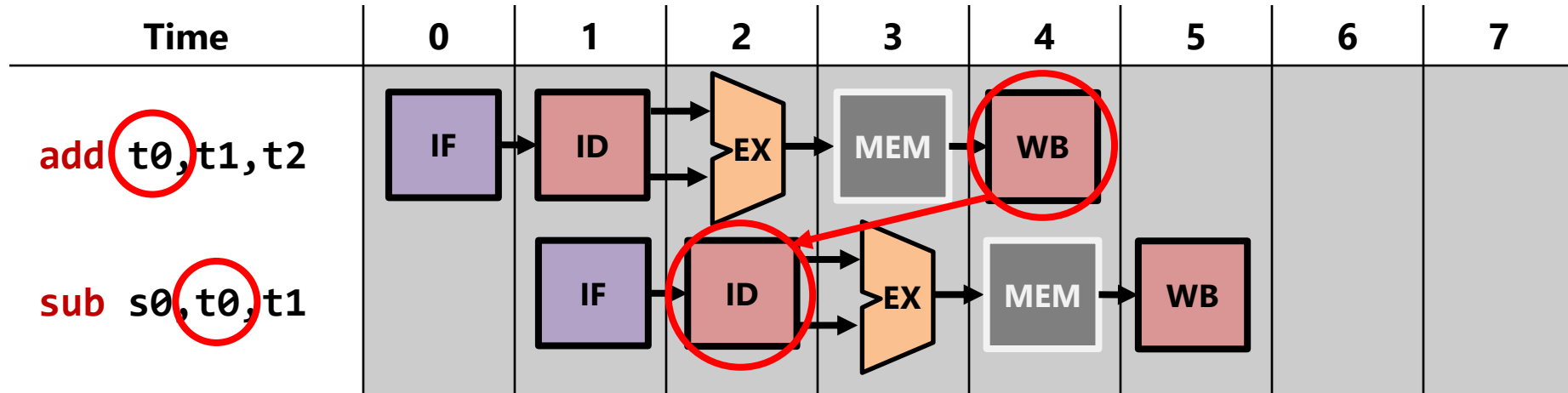
● When does **add** finish computing its sum?
● Well then... why not just *use the sum when we need it?*

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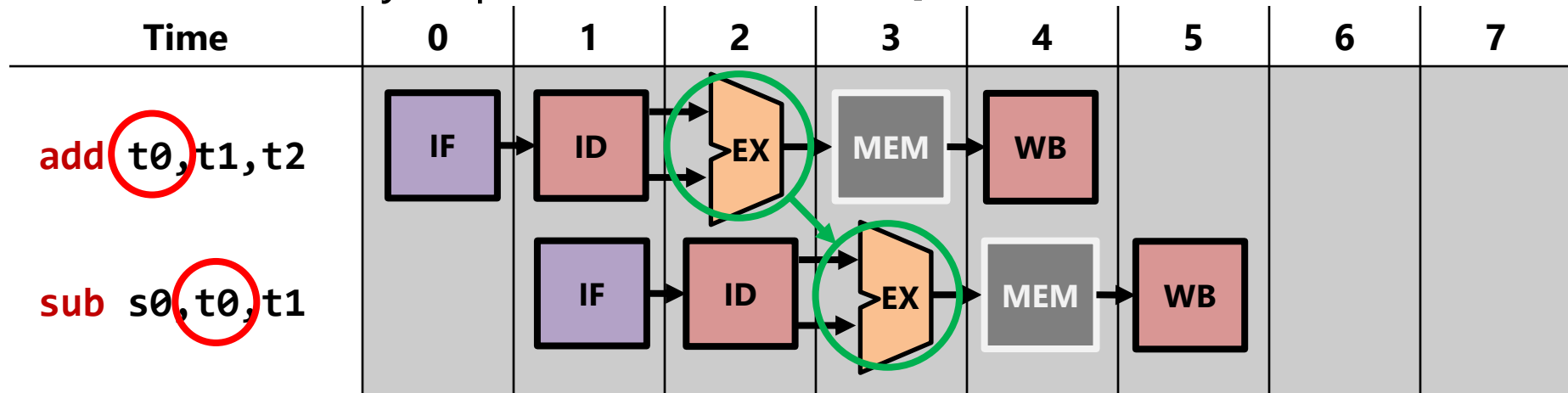
- This handles one kind of data forwarding...
- Where else can data come from and be written into registers?
  - Memory!

- Well memory accesses happen a cycle later...
- What are we going to have to do?
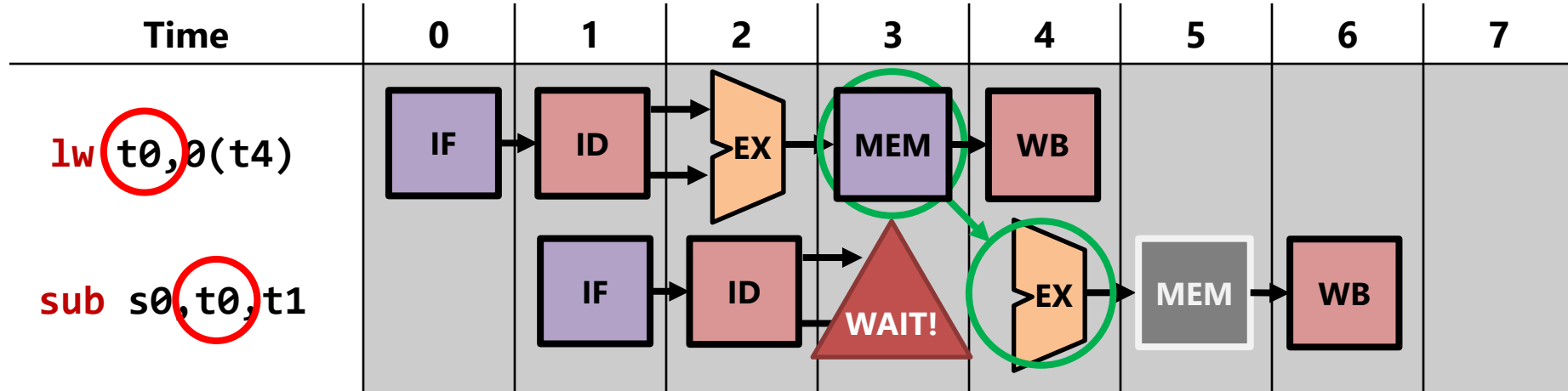
- This kind of stall is unavoidable...
  - *or is it?*

- Let's say the following is your morning routine *(2 hours total)*
  1. Have laundry running in washing machine *(30 minutes)*
  2. Put laundry into drawers *(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* the tea boiling *(30 minutes)*
  2. Put laundry into drawers while drinking tea *(30 minutes)*

- How? By simply by reordering our actions
  - Steps 1 → 2 and 3 → 4 have data dependencies
  - Steps 1, 3 and 2, 4 respectively can be **overlapped**

- 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
`lw t0,0(t4)`								
`sub s0,t0,t1`								
`lw t2,4(t4)`								
`sub s1,t2,t3`								

- If the **compiler** has knowledge of how the pipeline works, it can **reorder** instructions to let loads complete before using their data.
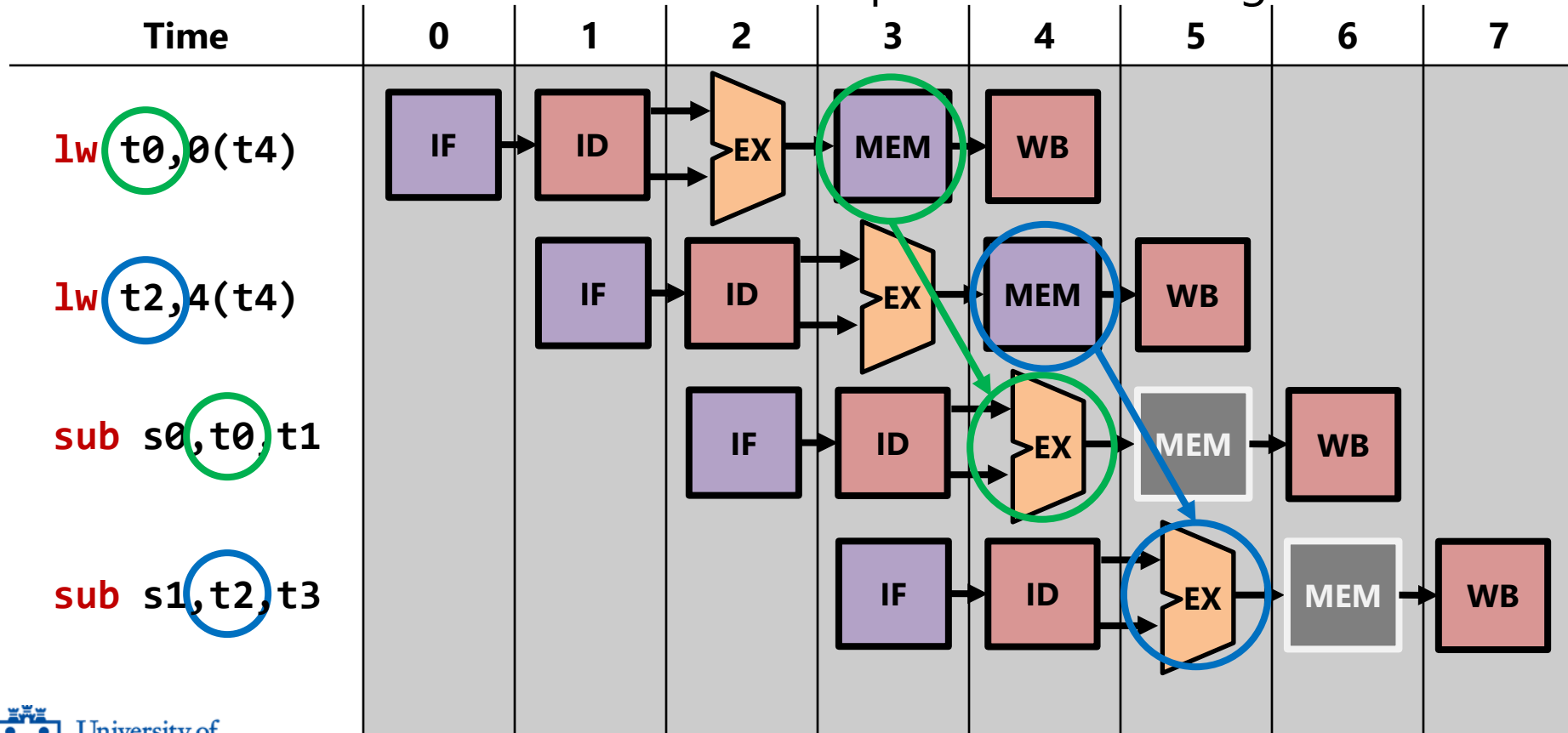
- Scheduling done by the compiler is called ***static scheduling***

- 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, much more difficult
      (e.g. Are `array[i]` and `array[j]` data dependent? Hard to tell.)

- ***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, indices `i`, `j` for `array[i]`, `array[j]` are known

- 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 happen *all the time* in programs.

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

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

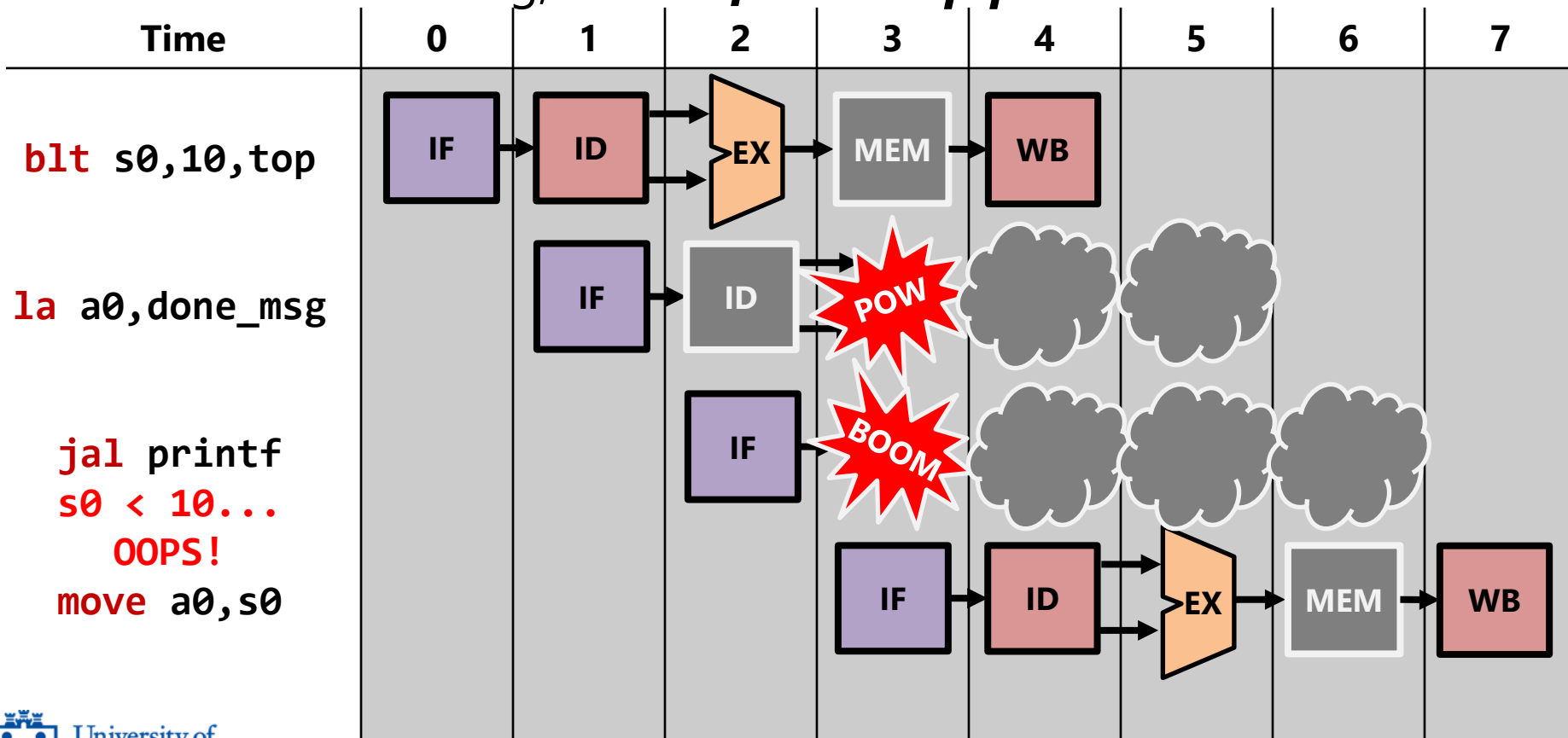How often does this **blt** instruction go to **top**? How often does it go to the following **la** instruction?

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

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

University of
Pittsburgh

77

- Let's assume our pipeline *always* fetches the next instruction
  o When CPU is wrong, it must **flush** the **pipeline**

And it just gets worse…

- If the loop is only a few instructions long…
  o That stall is a larger proportion of the time.

- If that loop iterates thousands of times…
  o We'll be spending more time killing instructions than executing!

- The deeper the pipeline gets, the bigger the flush penalty
  o Number of flushed instructions == distance from IF to MEM
  o Current architectures typically have more than a dozen stages!

- How can we solve this?
  o We need to somehow **predict** the correct branch outcome
  o But this is complicated! We'll take this up at a later time.

University of Pittsburgh