# Optimizing Pipeline Hazards
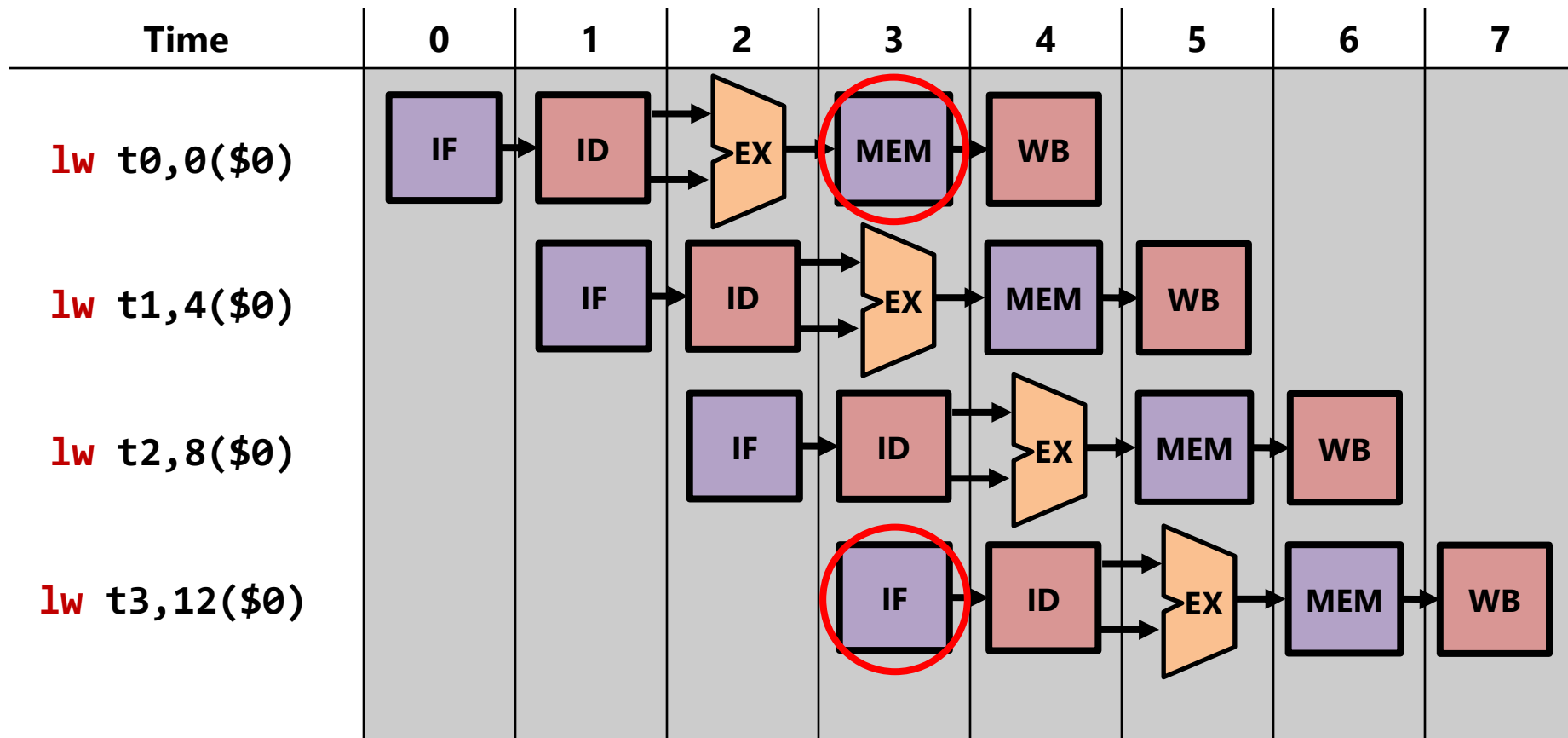
CS 1541
Wonsun Ahn

University of Pittsburgh

# Solving Structural Hazards

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

| Time | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|------|---|---|---|---|---|---|---|---|
| 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 |

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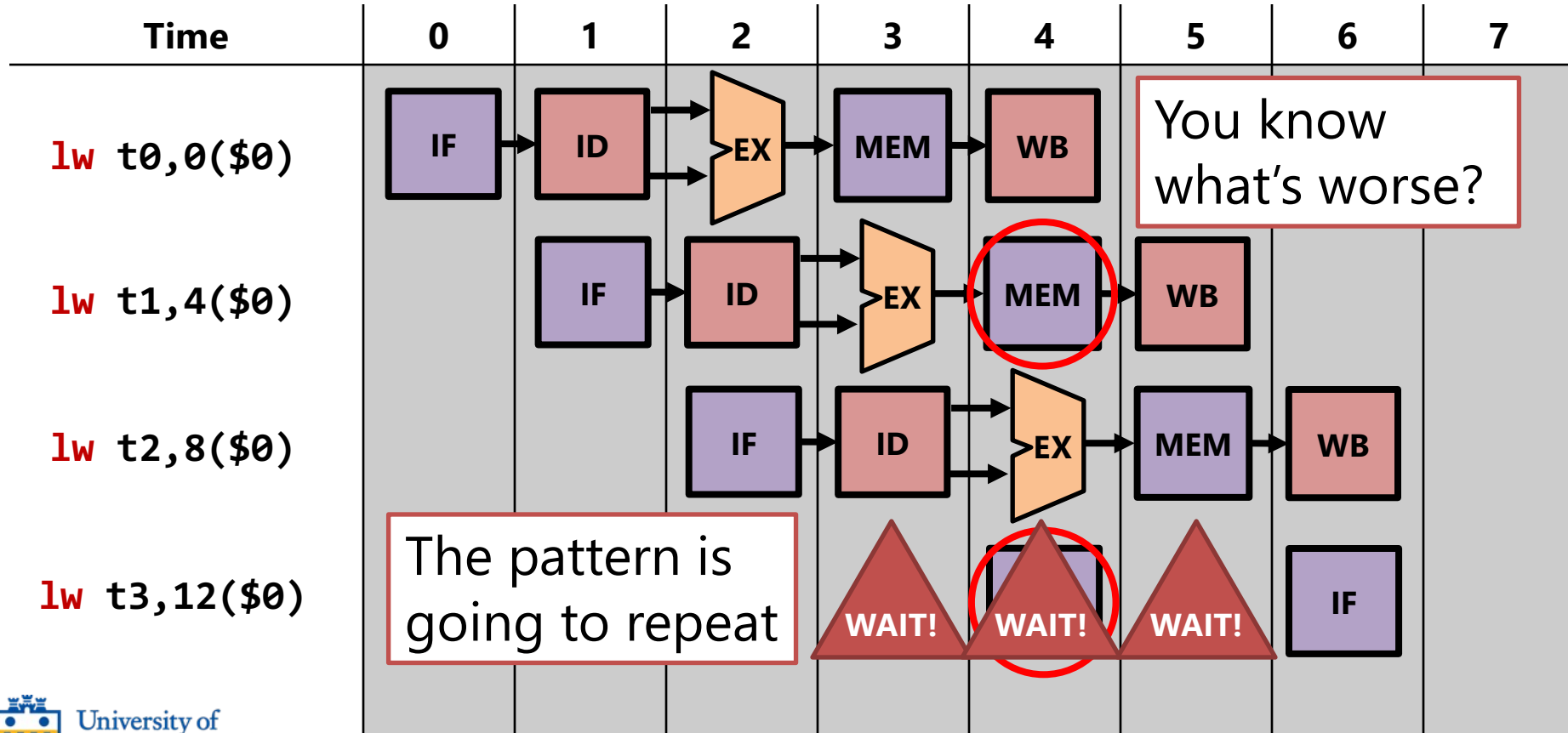
- Two people need to use **one** sink at the same time
  - Well, in this case, it's memory but same idea

# We can do something similar!
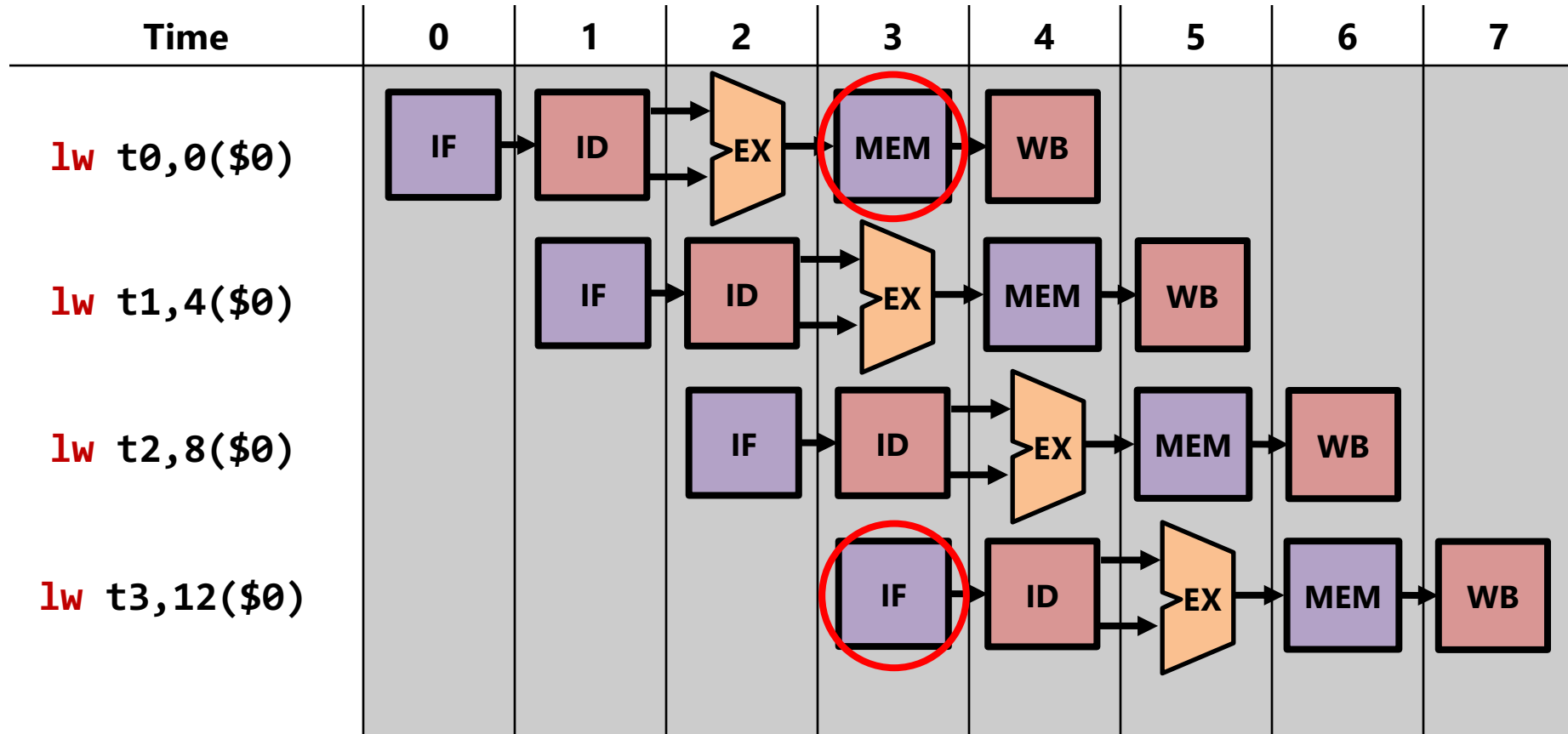
- One option is to **wait** (a.k.a. **stall**).



| Time | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|------|---|---|---|---|---|---|---|---|
| `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)` | | | | WAIT! | WAIT! | WAIT! | IF | |

You know what's worse?

The pattern is going to repeat

University of Pittsburgh

5

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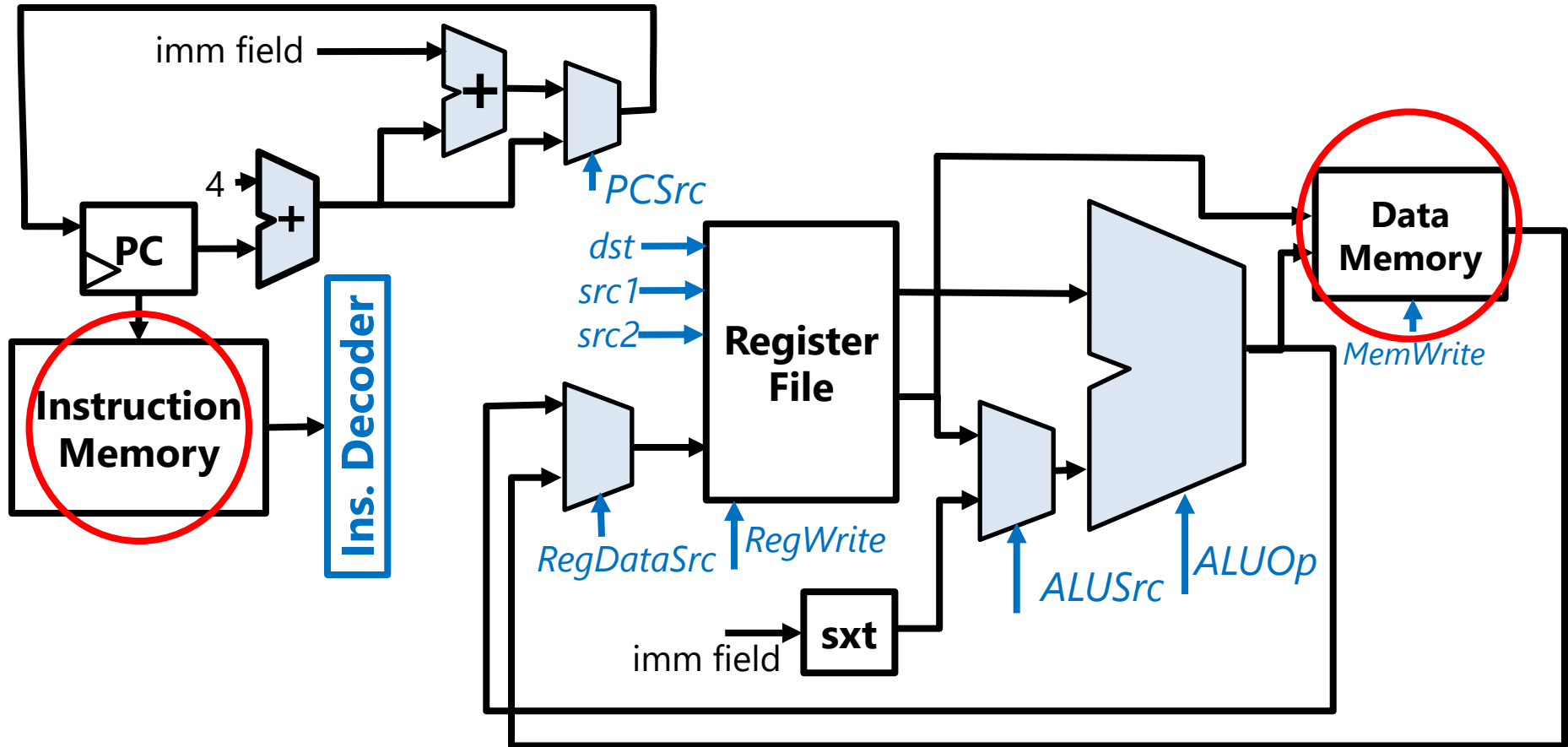  - Not literally but CPUs have separate *instruction* and *data caches*

University of
**Pittsburgh**

# Structural Hazard removed with two Memories

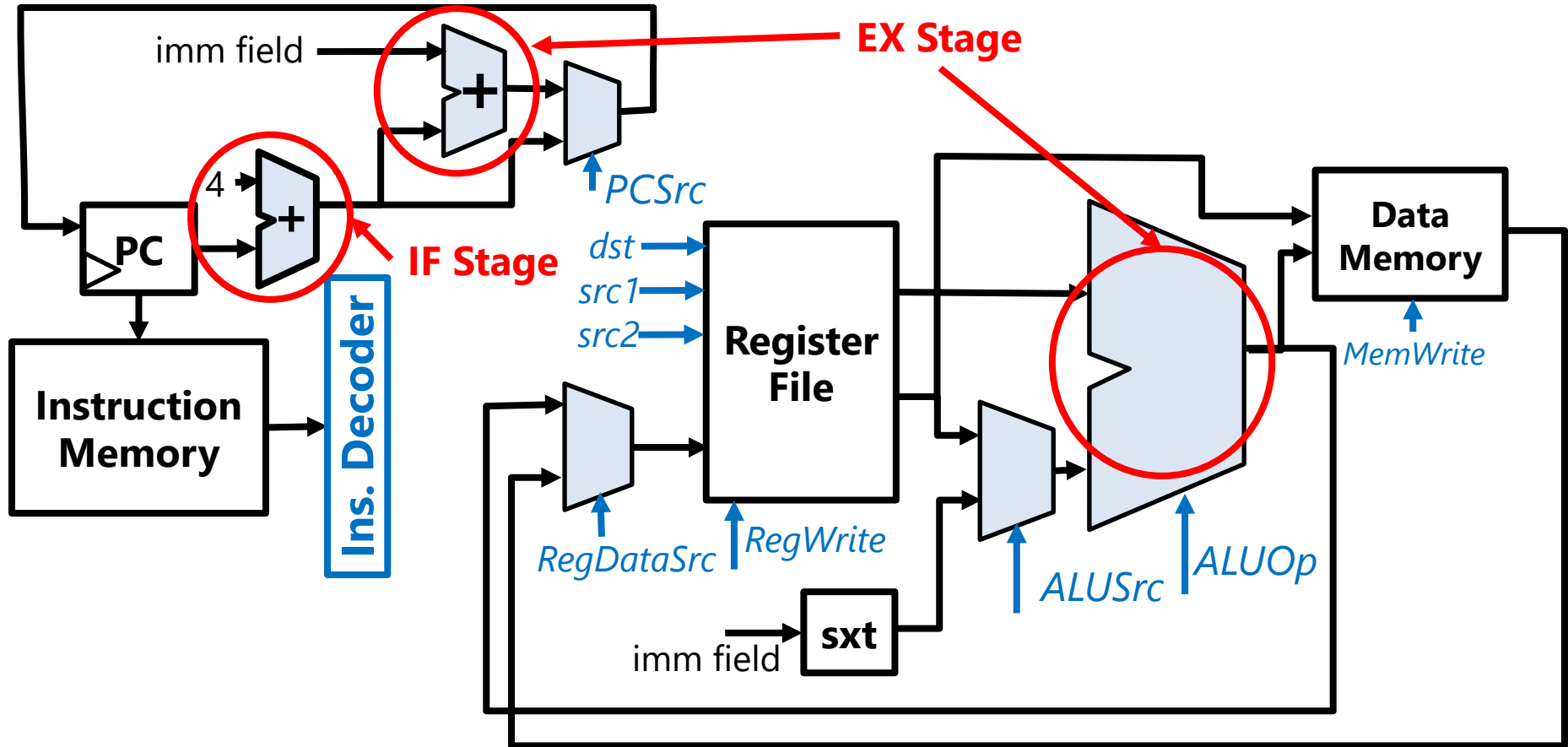- With separate i-cache and d-cache, MEM and IF can work in parallel

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

- Why do we need 3 adders?  To avoid stalls due to contention on ALU!
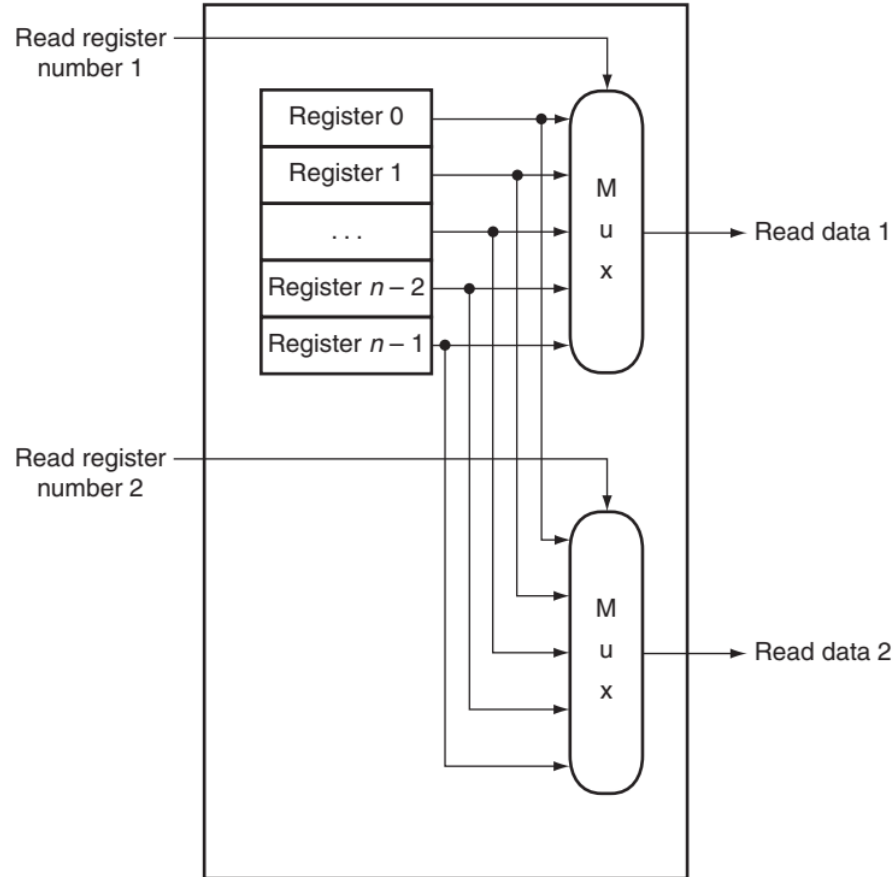
● There are mainly two ways to throw more hardware at the problem

1. Duplicate contentious resource
   o One memory cannot sustain MEM + IF stage at same cycle
      → Duplicate into one instruction memory, one data memory
   o 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
   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
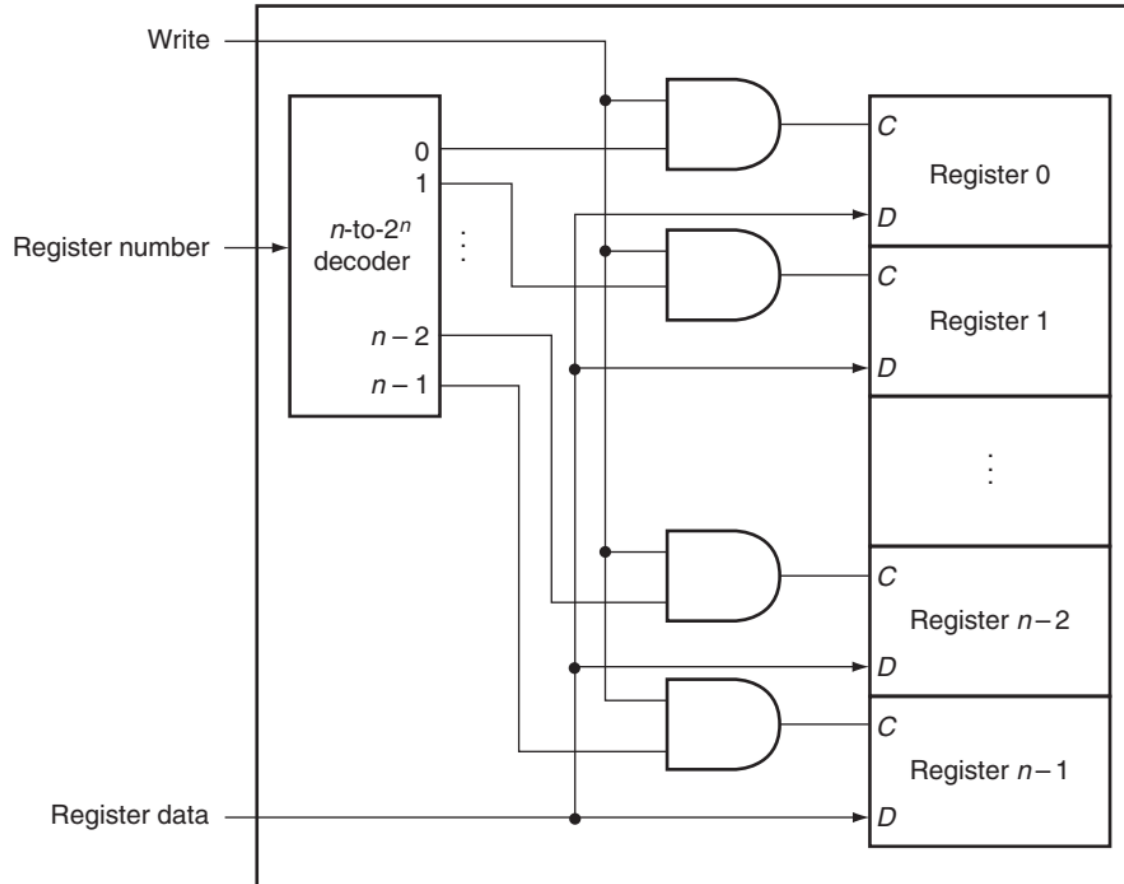      → Add more ports to memory structure for simultaneous access

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

● By adding more decoders, you can add more write ports
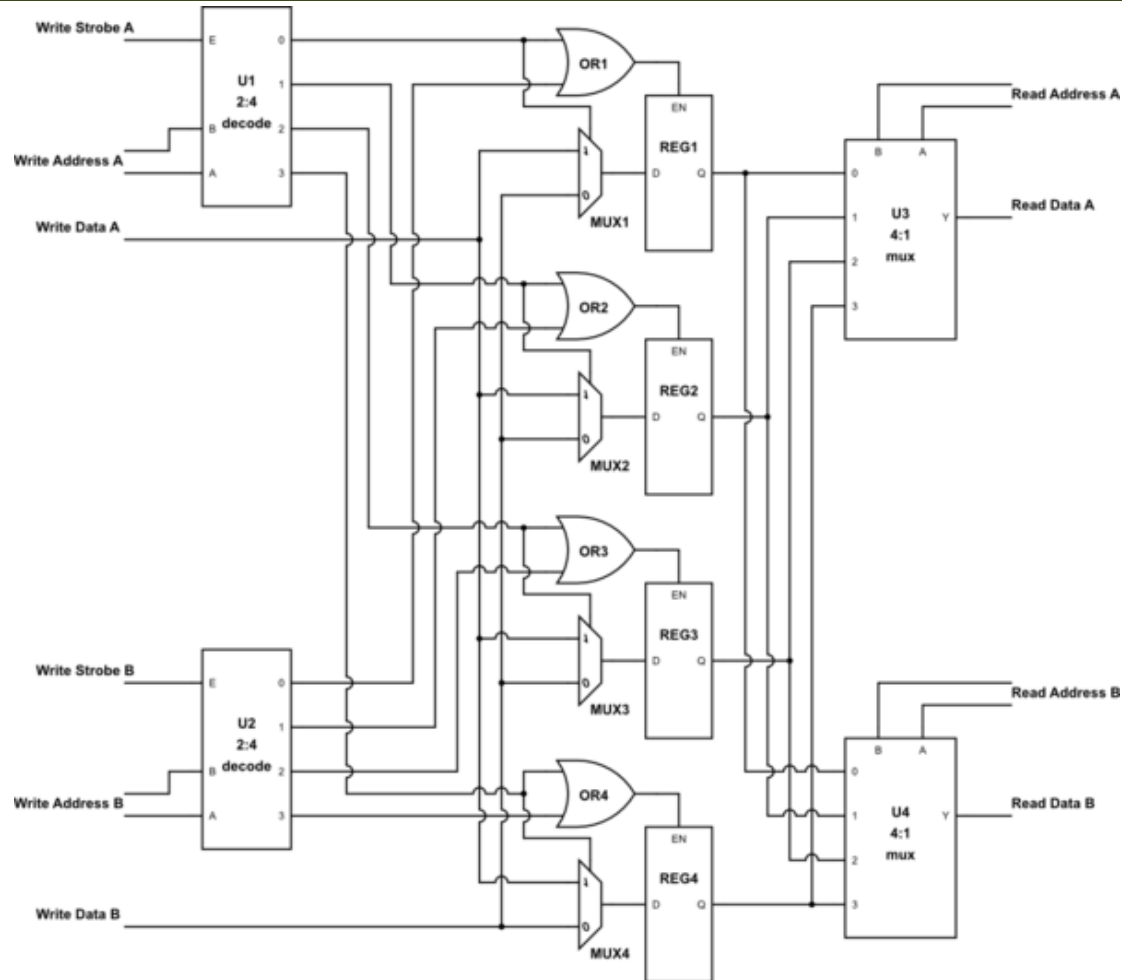
Dave Tweed (https://electronics.stackexchange.com/users/11683/dave-tweed),
Build A Two Port Write and Two Port Read Register File with 4 Registers, URL (version: 2017-02-21):
https://electronics.stackexchange.com/q/273002

13

- 2 read ports for 2 source registers, 1 write port for dest register
  - 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? (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, (cache) memory would need more ports too!
  - → Muxes, decoders increase critical path (lowers **frequency**)
  - → Extra circuitry consumes more **power**

- We'll talk more about this when we discuss superscalars

University of Pittsburgh

# Solving Data Hazards

# Data Hazards

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

| Time | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|------|---|---|---|---|---|---|---|---|

add t0,t1,t2 — IF → ID → EX → MEM → WB

sub s0,t0,t1 — IF → ID → EX → MEM → WB
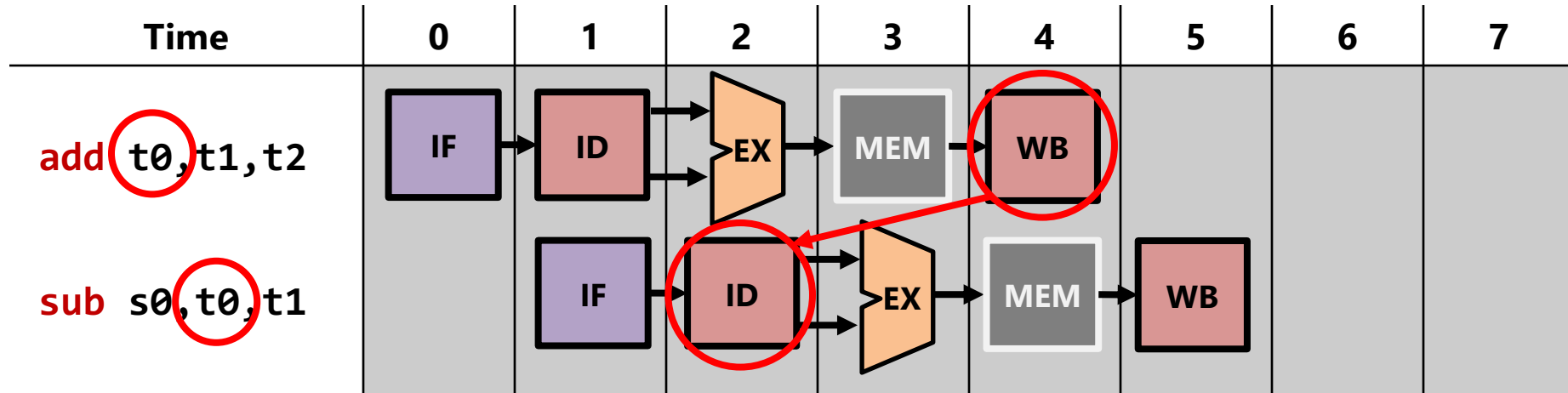
- When does **add** finish computing its sum?
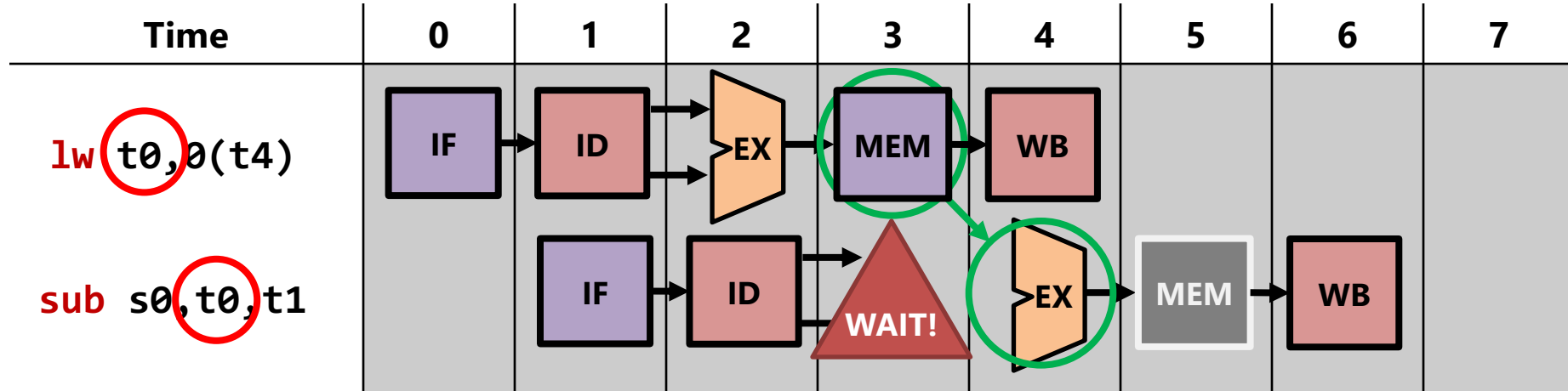- Well then... why not just *use the sum when we need it?*

- 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.

| Time | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|------|---|---|---|---|---|---|---|---|



add t0,t1,t2

sub s0,t0,t1
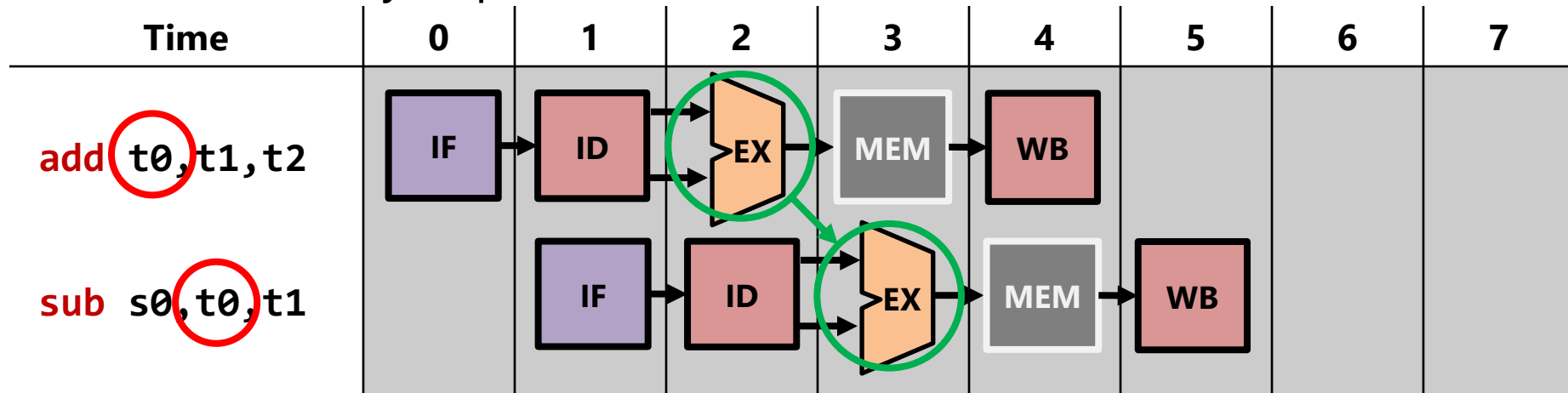
- This handles one kind of data forwarding…
- Where else can data come from and be written into registers?
  - Memory!

University of Pittsburgh

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



| Time | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|------|---|---|---|---|---|---|---|---|
| lw t0,0(t4) | IF | ID | EX | MEM | WB | | | |
| sub s0,t0,t1 | | IF | ID | WAIT! | EX | MEM | WB | |

- This kind of stall is unavoidable in our current pipeline

University of Pittsburgh

18

If dependent on MemRead (load) instruction, even forwarding unit can't avoid stall
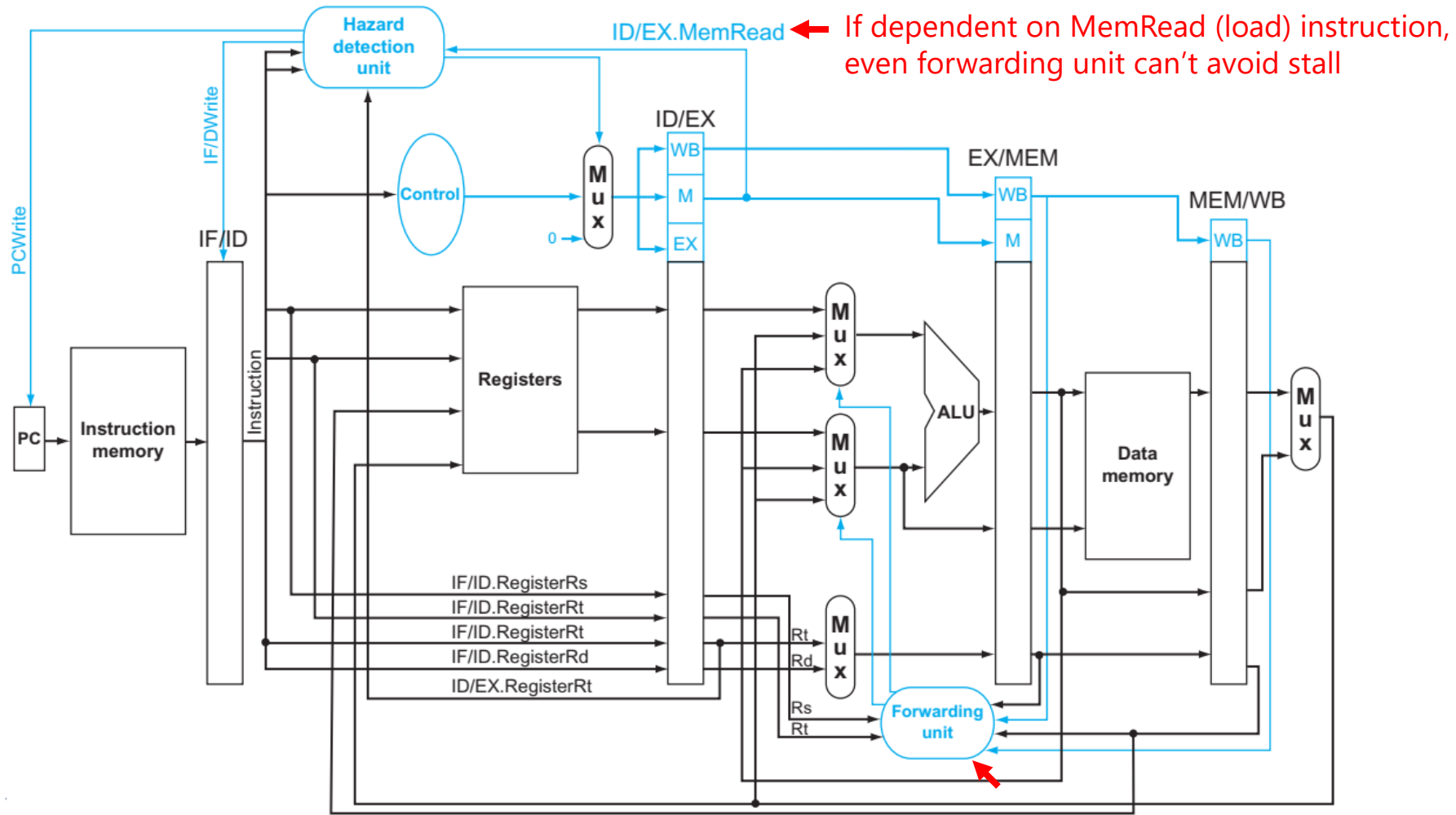
- Just like the HDU, the Forwarding Unit is **power** hungry

- Number of forwarding wires $\propto$ (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

- **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

University of Pittsburgh

# Solution 2: Avoid stalls by reordering

- 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

University of Pittsburgh
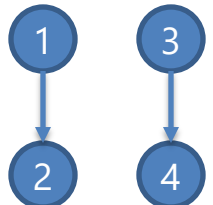
# Data Hazard removed through Compiler Reordering

- 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.
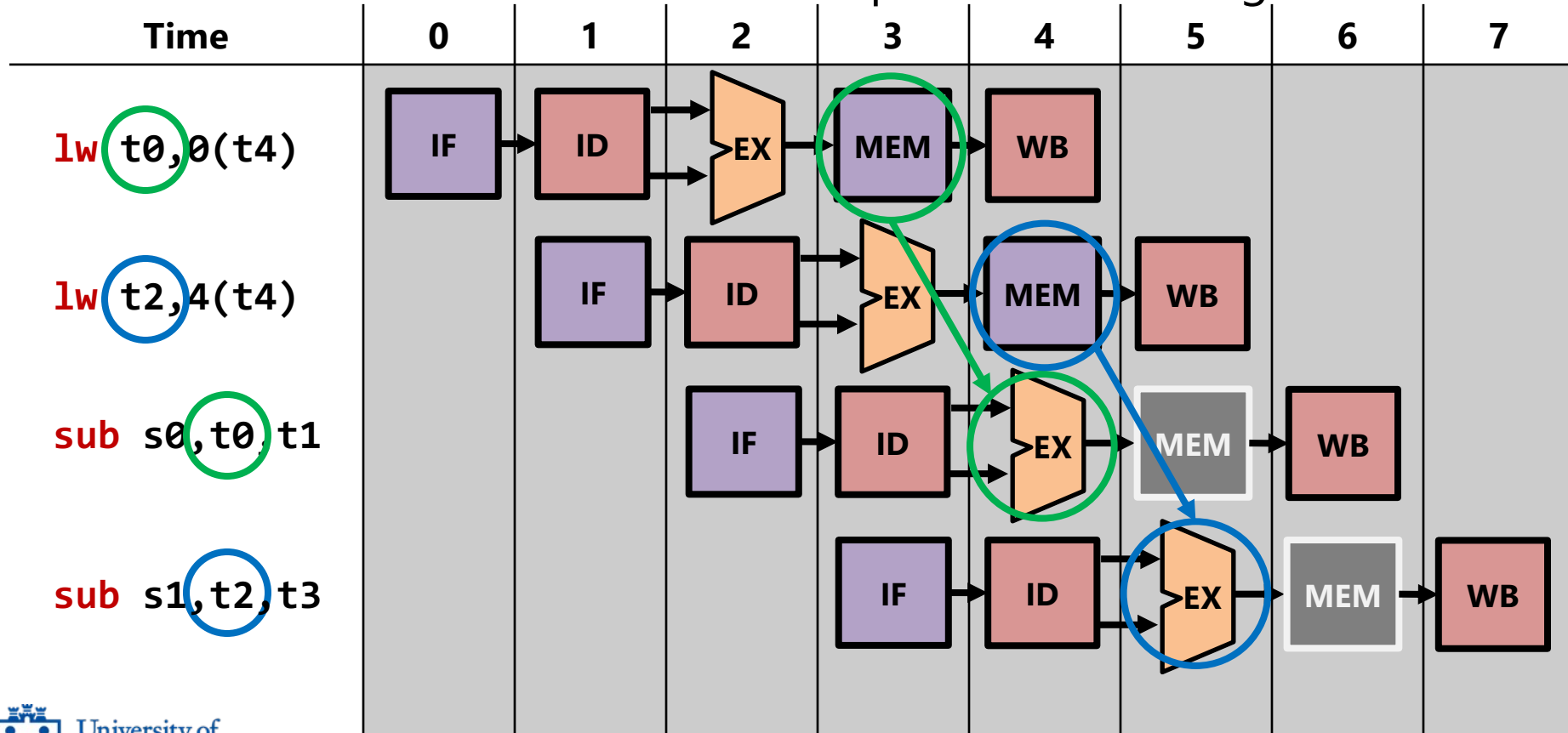
- Reordering done by the compiler is called ***static scheduling***

- 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 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(t0) and 4(t4) 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 happen *all the time* in programs.

```
for(s0 = 0 .. 10)
    print(s0);
```
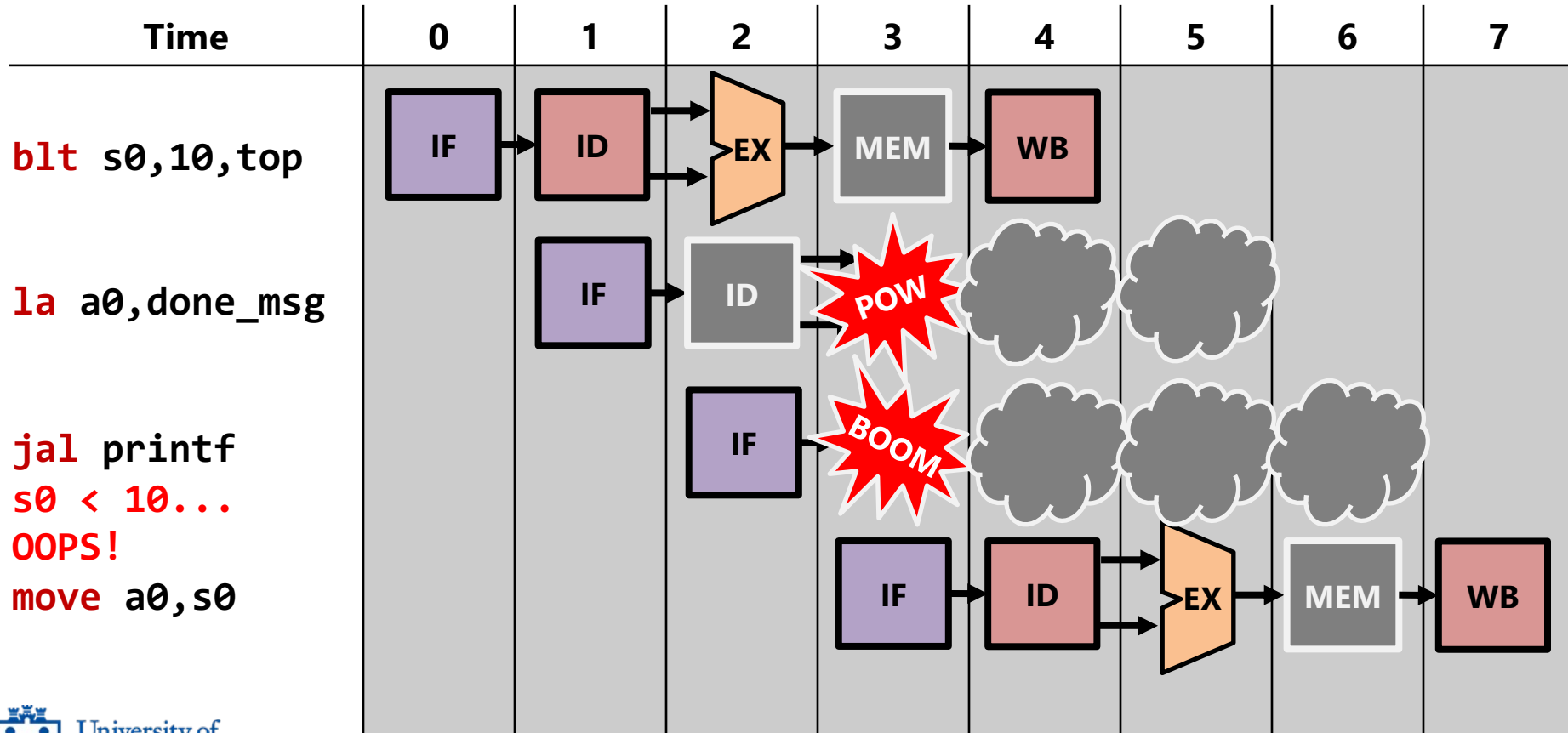
```
printf("done");
```

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
```

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

| Time | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|------|---|---|---|---|---|---|---|---|

`blt s0,10,top`

`la a0,done_msg`

`jal printf`
`s0 < 10...`
`OOPS!`
`move a0,s0`



28

- **Frequency** of flushes ∝ frequency of branches
  - If we have a tight loop, branches happen every few instructions
  - Typically, branches account for 15~20% of all instructions

- **Penalty** from one flush ∝ depth of pipeline
  - Number of flushed instructions == distance from IF to EX
  - What if **3** IF stages, **4** ID stages, and **3** EX stages? Penalty == 10!

- Current architectures can have more than 20 stages!
  - May spend more time just flushing instructions than doing work!
  - Another reason why deep pipelines are problematic

# Performance Impact from Control Hazards

- CPI = CPI$_{nch}$ + $\alpha$ * $\pi$ * K
  - CPI$_{nch}$ : CPI with no control hazard
  - $\alpha$ : fraction of branch instructions in the instruction mix
  - $\pi$ : probability a branch is actually taken
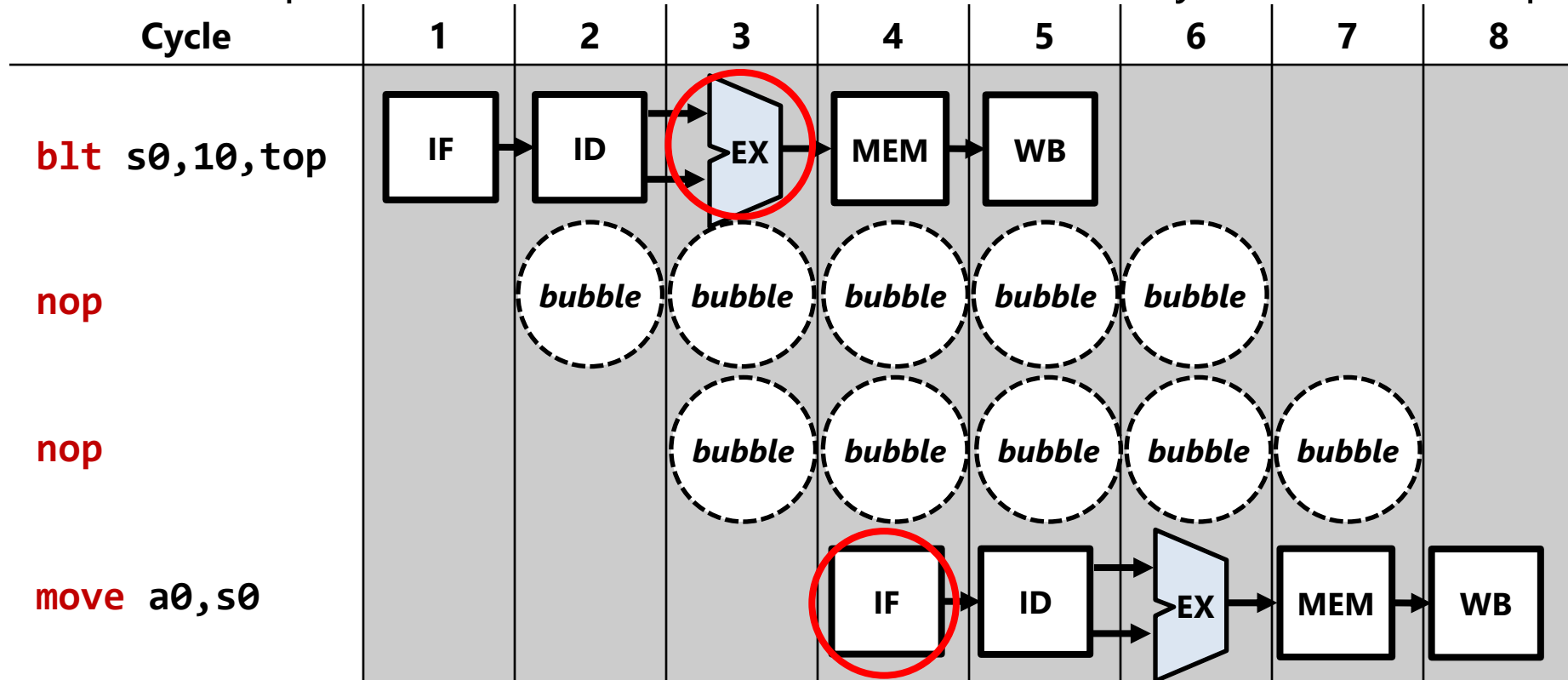  - K : penalty per pipeline flush

**Example:** If 20% of instructions are branches and the probability that a branch is taken is 50%, and pipeline flush penalty 7 cycles, then:

CPI = CPI$_{nch}$ + 0.2 * 0.5 * 7 = CPI$_{nch}$ + 0.7 cycles per instruction

- What if we had a compiler insert no-ops, with no HDU?
  - It's even worse, as we will soon see.

University of Pittsburgh

- Since compiler does not know direction, must always insert two nops

- CPI = $CPI_{nch}$ + $\alpha$ * K
  - $CPI_{nch}$ : CPI with no control hazard
  - $\alpha$ : fraction of branch instructions in the instruction mix
  - K : no-ops inserted after each branch

  **Example:** If 20% of instructions are branches and the probability that a branch is taken is 50%, and branch resolution delay of 7 no-ops, then:
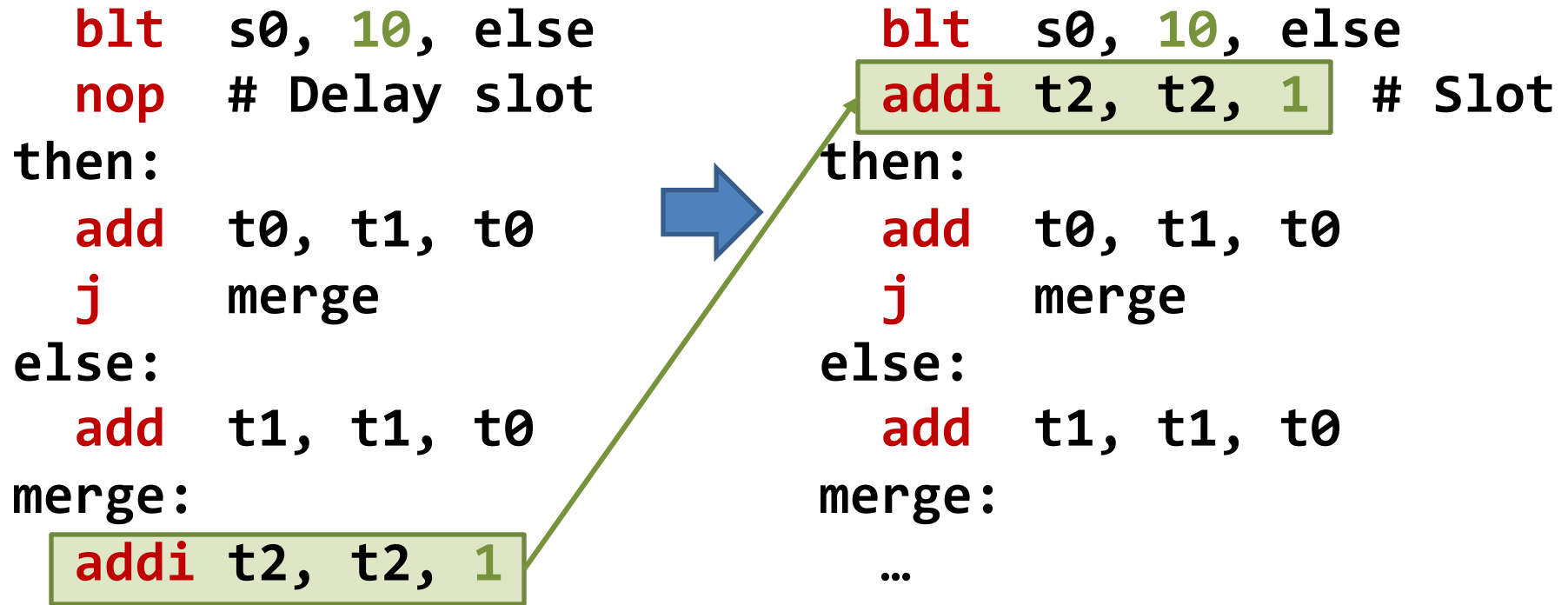  CPI = $CPI_{nch}$ + 0.2 * 7 = $CPI_{nch}$ + 1.4 cycles per instruction

  - Branch-taken rate is irrelevant - compiler always inserts two nops

- Is there a way to minimize the performance impact?

- Idea: Use compiler **static scheduling** to fill no-ops with useful work
  - Remember? We did the same for no-ops due to data hazards.

- ***Delay slot***: One or more instructions immediately following a branch instruction that **executes regardless of branch direction**
  - Processor never needs to flush these instructions!
  - ISA must be modified to support this branch semantic
  - It's compiler's job to fill delay slots as best as it can, with instructions not control dependent on the branch

University of Pittsburgh

```
    blt   s0, 10, else                blt   s0, 10, else
    nop   # Delay slot                addi t2, t2, 1  # Slot
then:                              then:
    add   t0, t1, t0                  add   t0, t1, t0
    j     merge                       j     merge
else:                              else:
    add   t1, t1, t0                  add   t1, t1, t0
merge:                             merge:
    addi t2, t2, 1                    …
```

- The **addi** instruction is moved into delay slot
  o It is not control dependent on the branch outcome of **blt**
  o It is not data dependent on registers **t0** or **t1**

University of
Pittsburgh

34

● Sounded like a good idea on paper but didn't work well in practice

1. Turns out filling delay slots with the compiler is not always easy
   o Often data and control independent instructions don't exist

2. Delay slots baked into the ISA were not future proof
   o Number of delay slots did not match new generation of CPUs
   o New generation of CPUs had fancier ways to avoid bubbles
   o Delays slots ended up being a hindrance

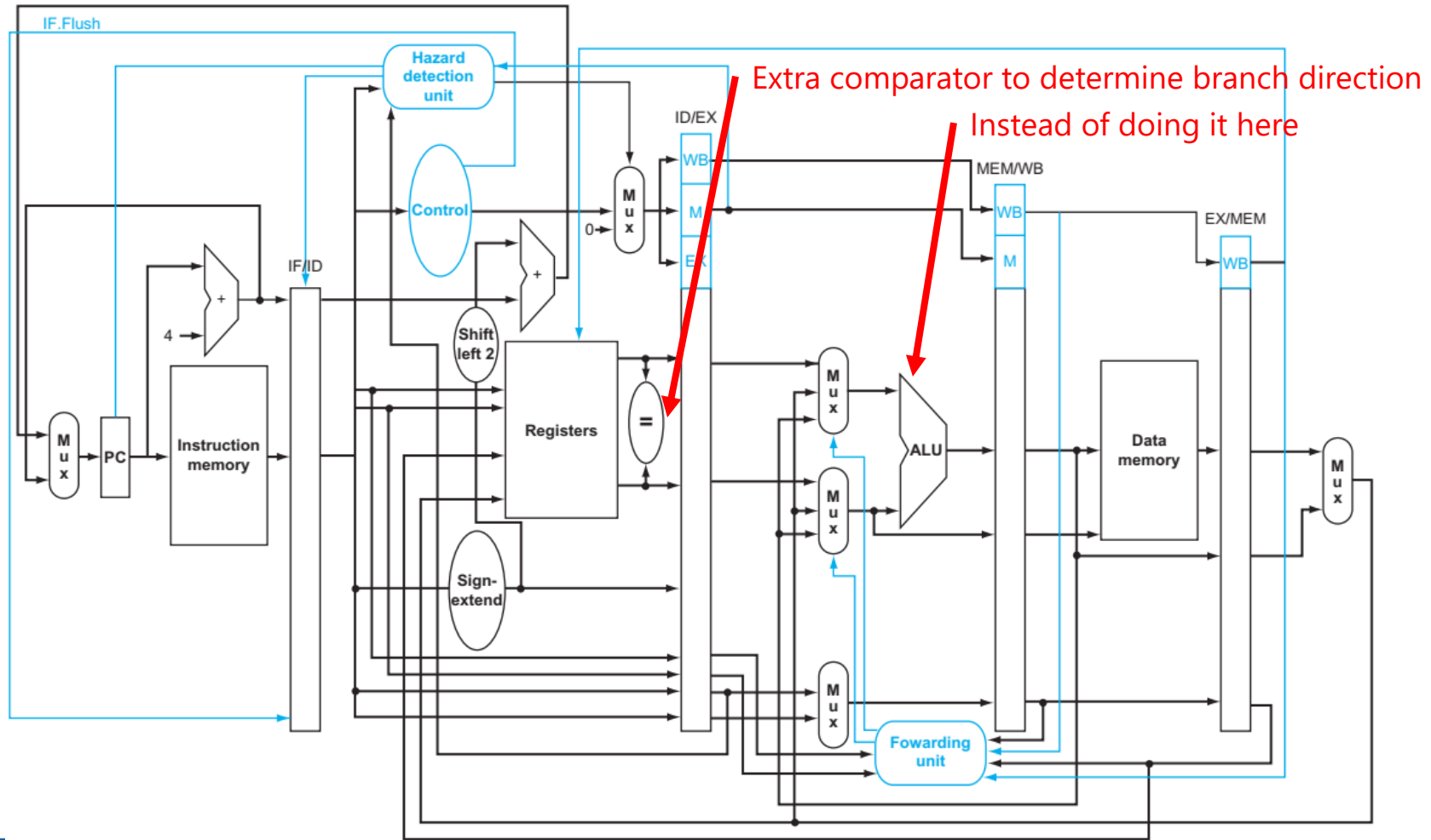● Next idea please!

- What if branch comparison was done at the ID stage, not EX stage?
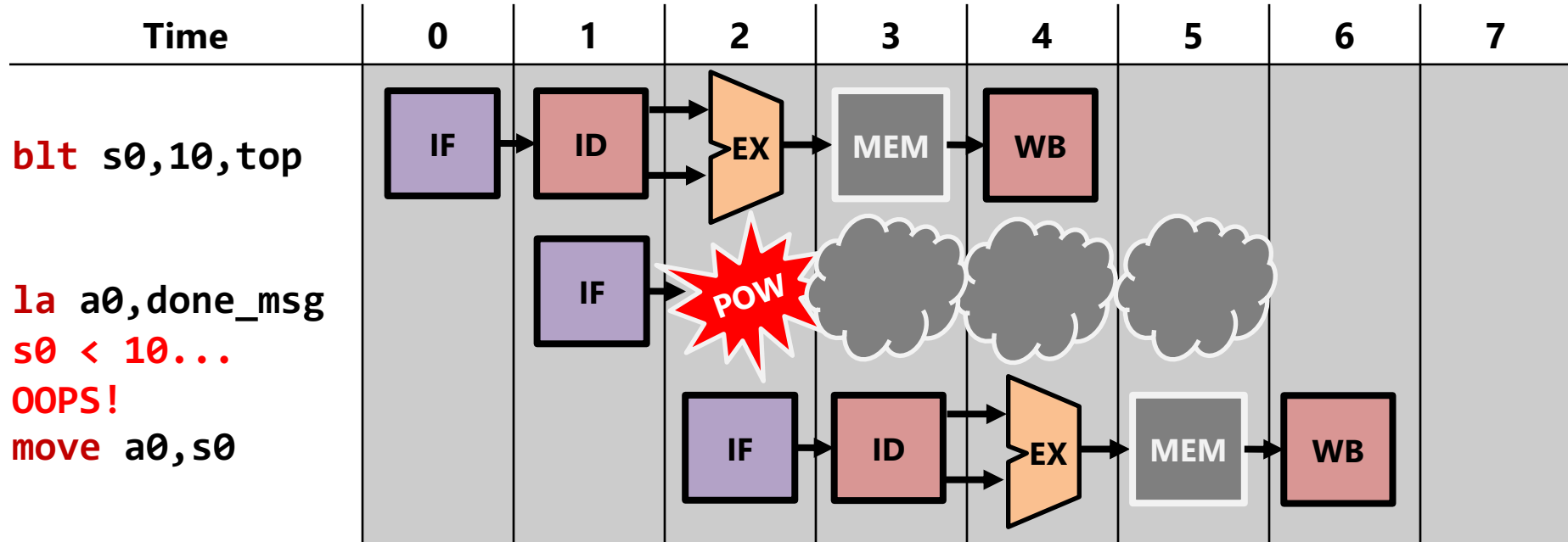


- Reduced penalty from 2 cycles → 1 cycle!
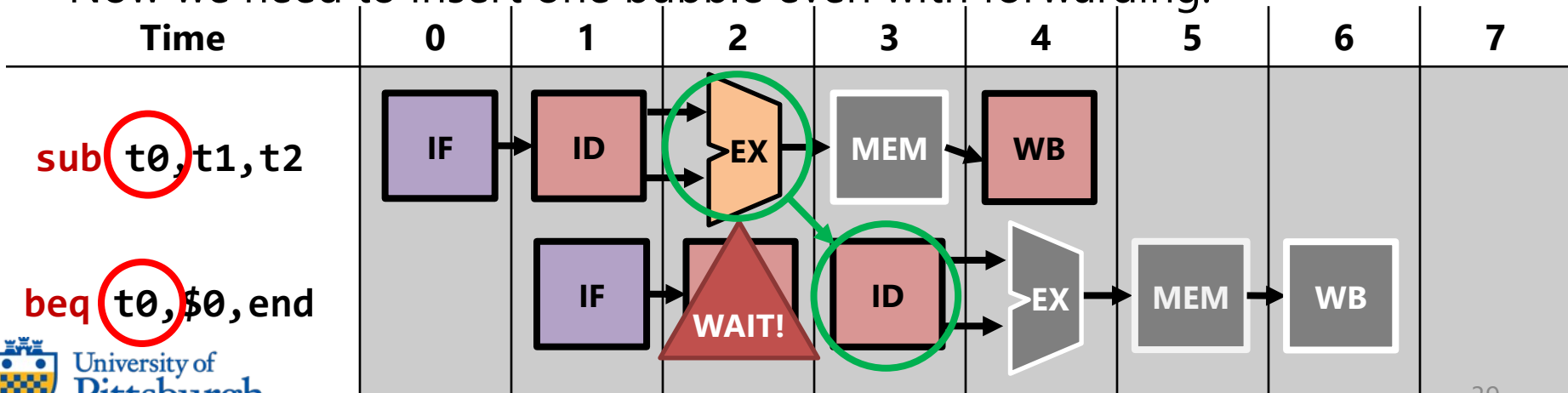- But of course that means we need a comparator at the ID stage

Extra comparator to determine branch direction

Instead of doing it here

- **Extra delay on data hazards**. Used to have no delay:

| Time | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|

sub t0,t1,t2 — IF → ID → EX → MEM → WB

beq t0,$0,end — IF → ID → EX → MEM → WB
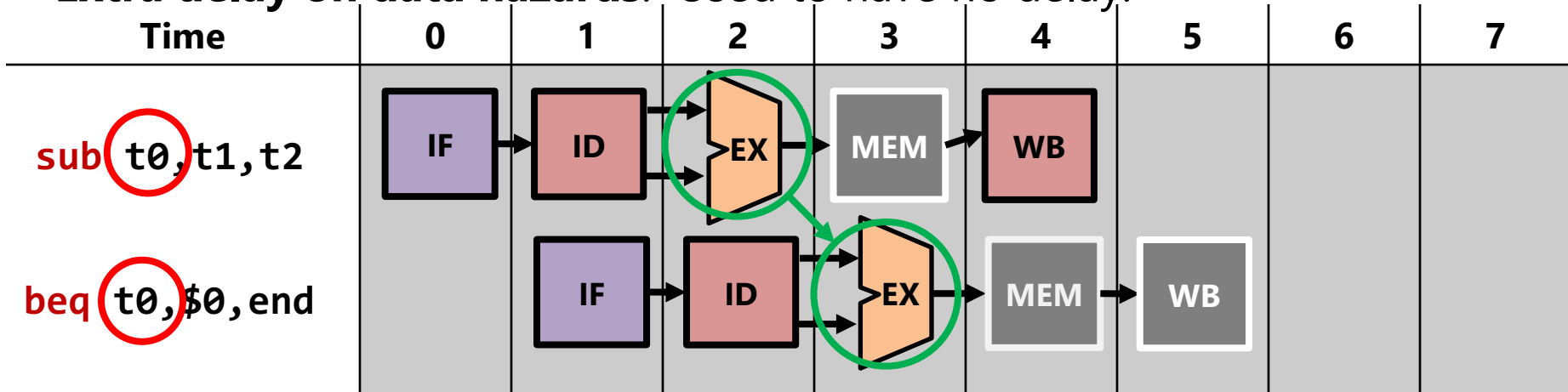
- Now we need to insert one bubble even with forwarding:

| Time | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|

sub t0,t1,t2 — IF → ID → EX → MEM → WB

beq t0,$0,end — IF → WAIT! → ID → EX → MEM → WB

39

- Extra delay on data forwarded from `lw` also:

| Time | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|------|---|---|---|---|---|---|---|---|
| `lw t0,0($t1)` | IF | ID | EX | MEM | WB | | | |
| `beq t0,$0,end` | | IF | WAIT! | WAIT! | ID | EX | MEM | WB |
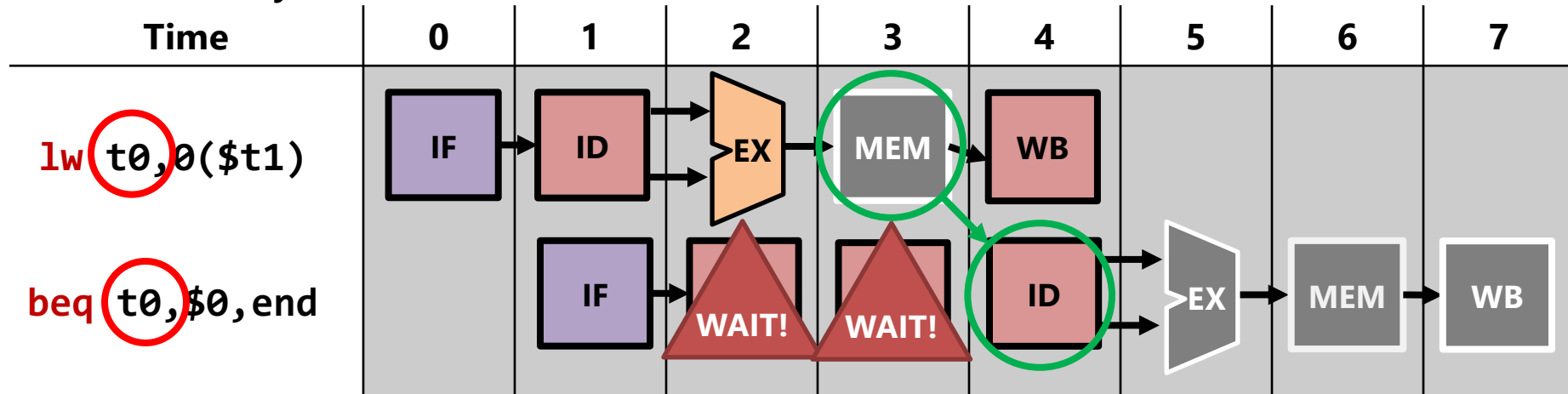
- Now we must insert two bubbles instead of one!
- Not to mention we must now add **more forwarding paths**:
  - **EX → ID**, **MEM → ID**
- We also need to add MUXes before our new comparator

The figure in textbook is incomplete.
Needs MUXes and forwarding lines just like the ALU.