Optimizing Pipeline Hazards

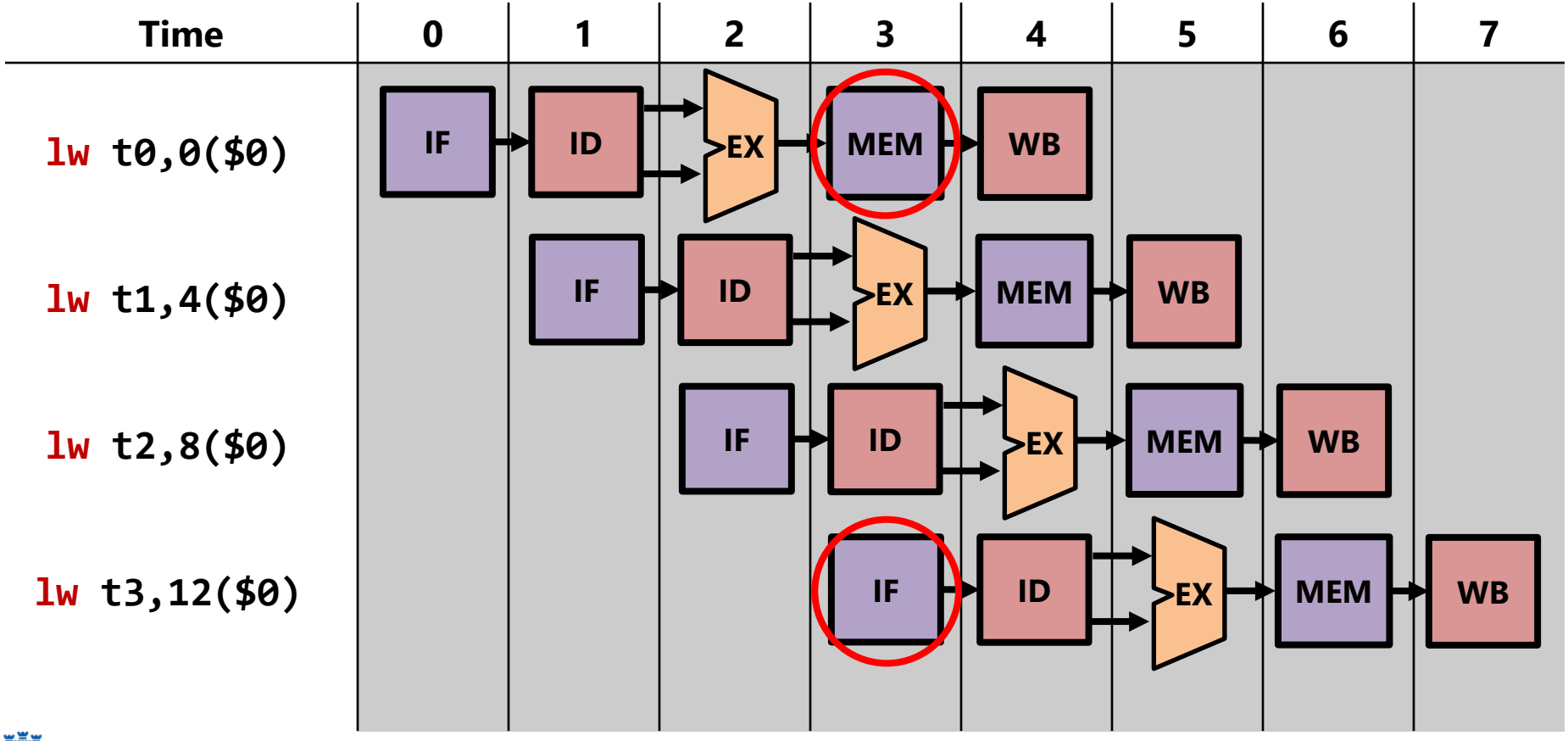
CS/COE 1541 (Fall 2020)

Wonsun Ahn

Solving Structural Hazards

Structural Hazard on Memory

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

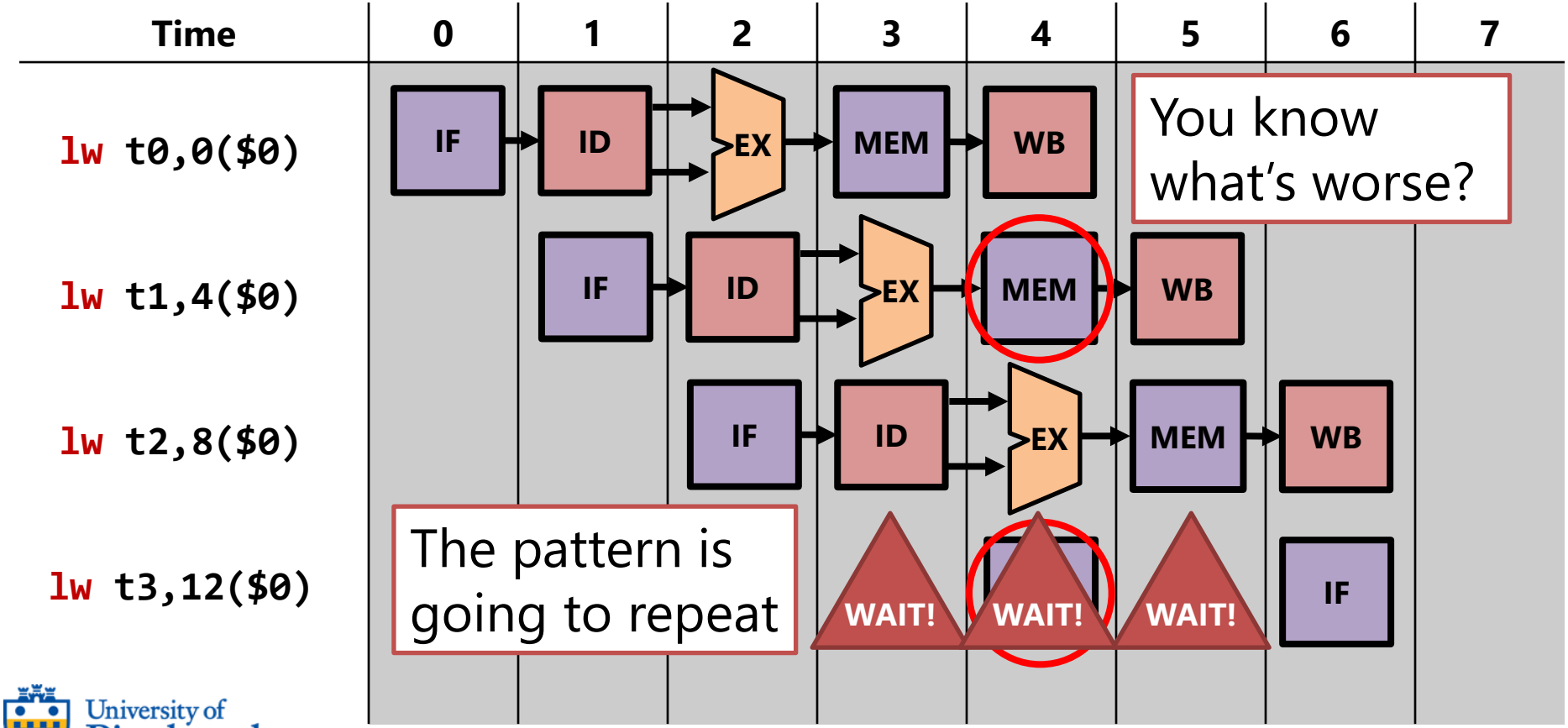


What could we do??

- 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**).

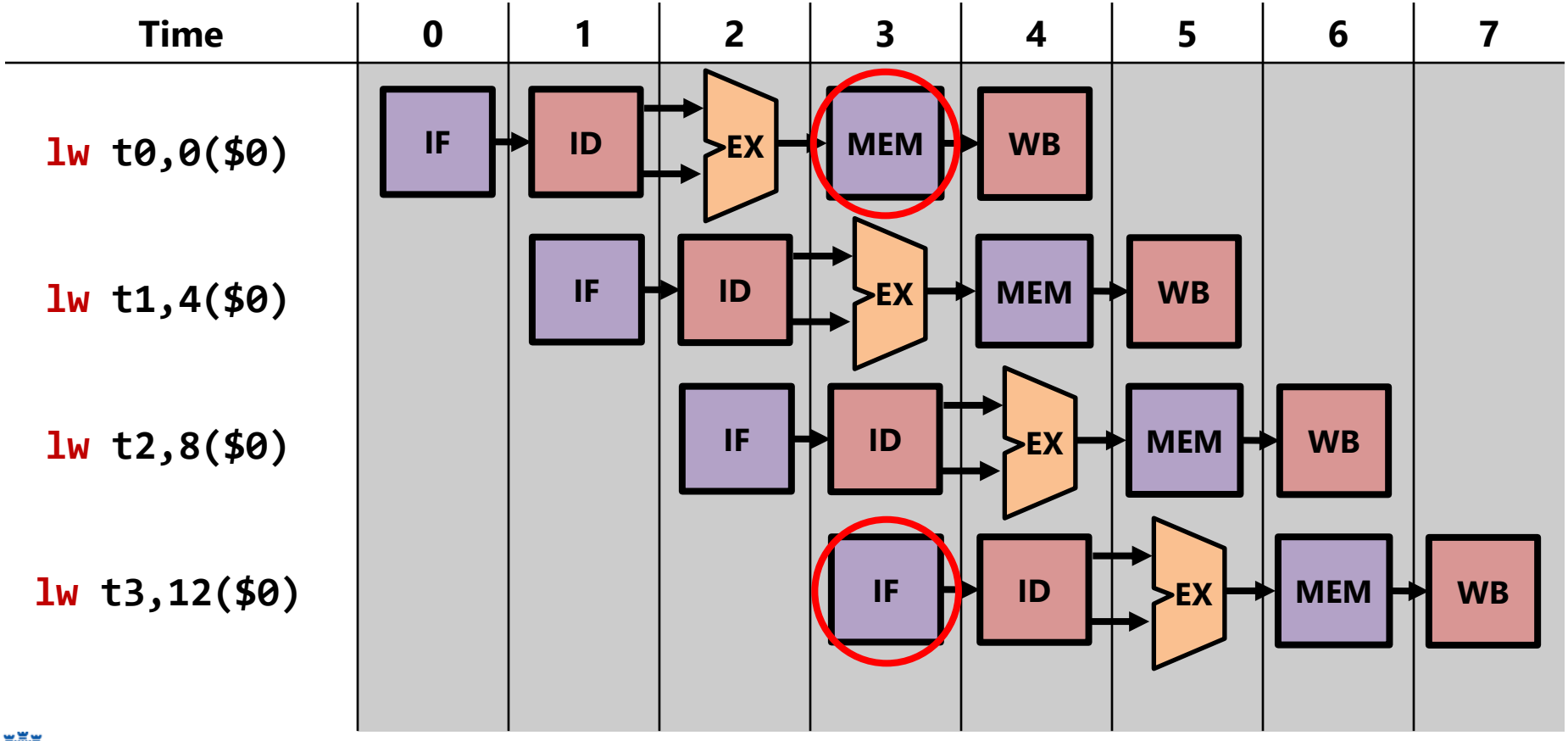


Or we could throw in more hardware!

- 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***

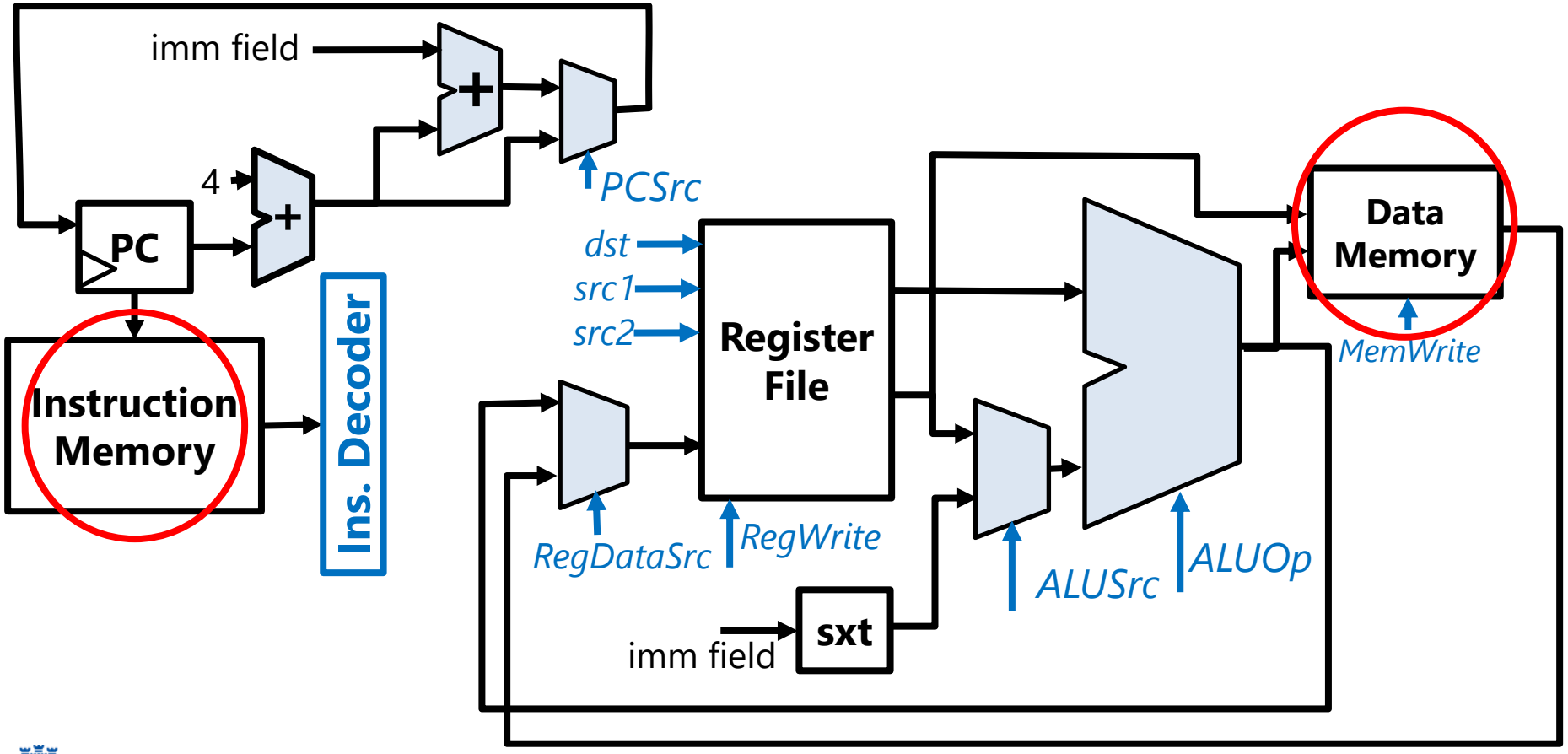
Structural Hazard removed with two Memories

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



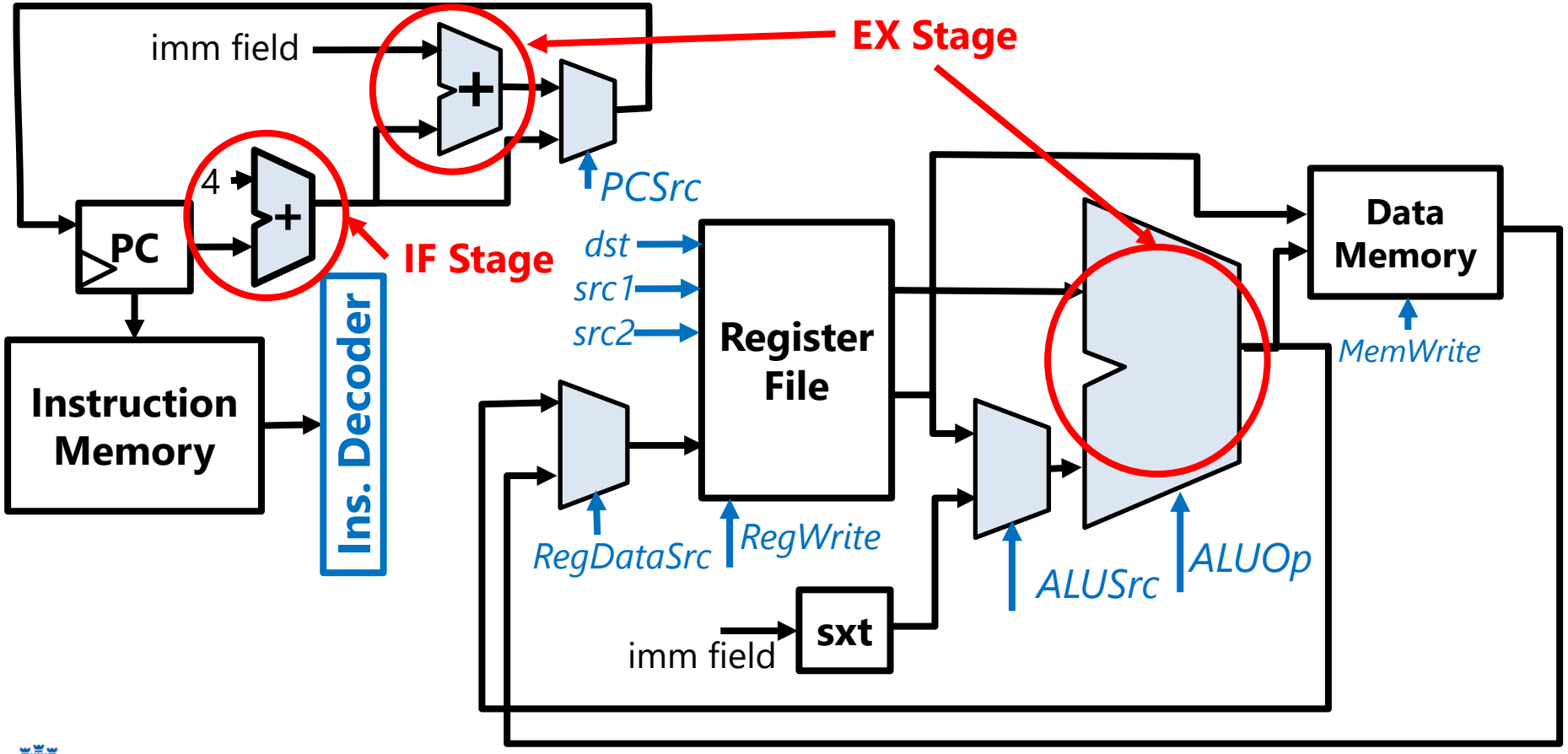
Structural Hazard removed with two Memories

- 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

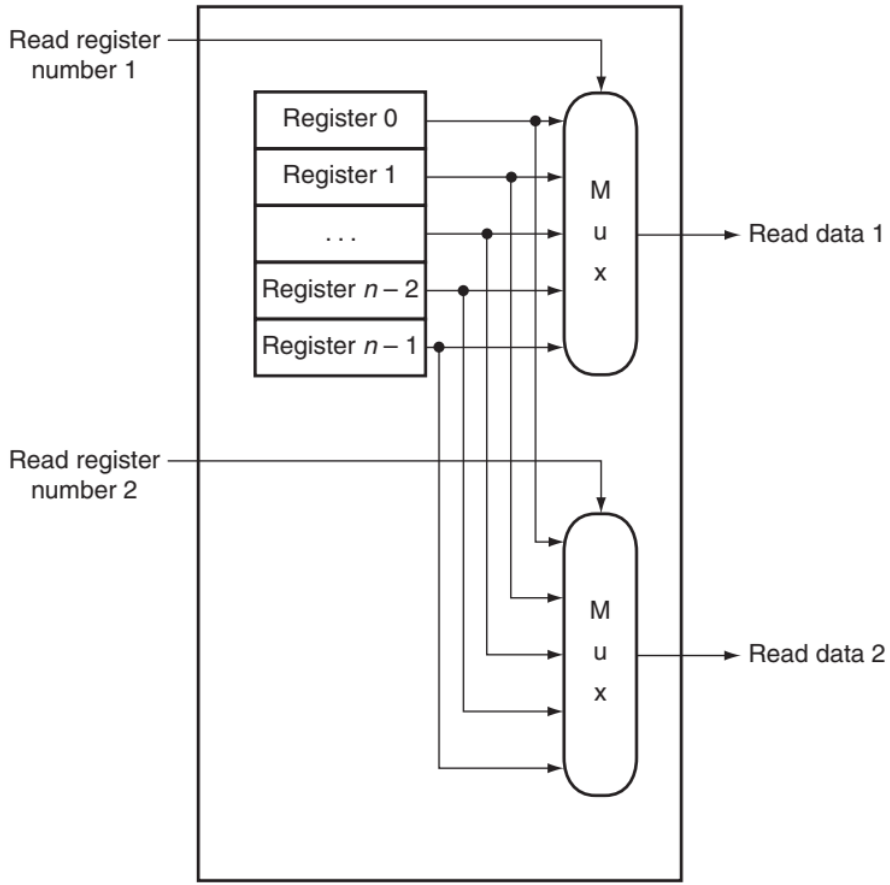
- There are mainly two ways to throw more hardware at the problem

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

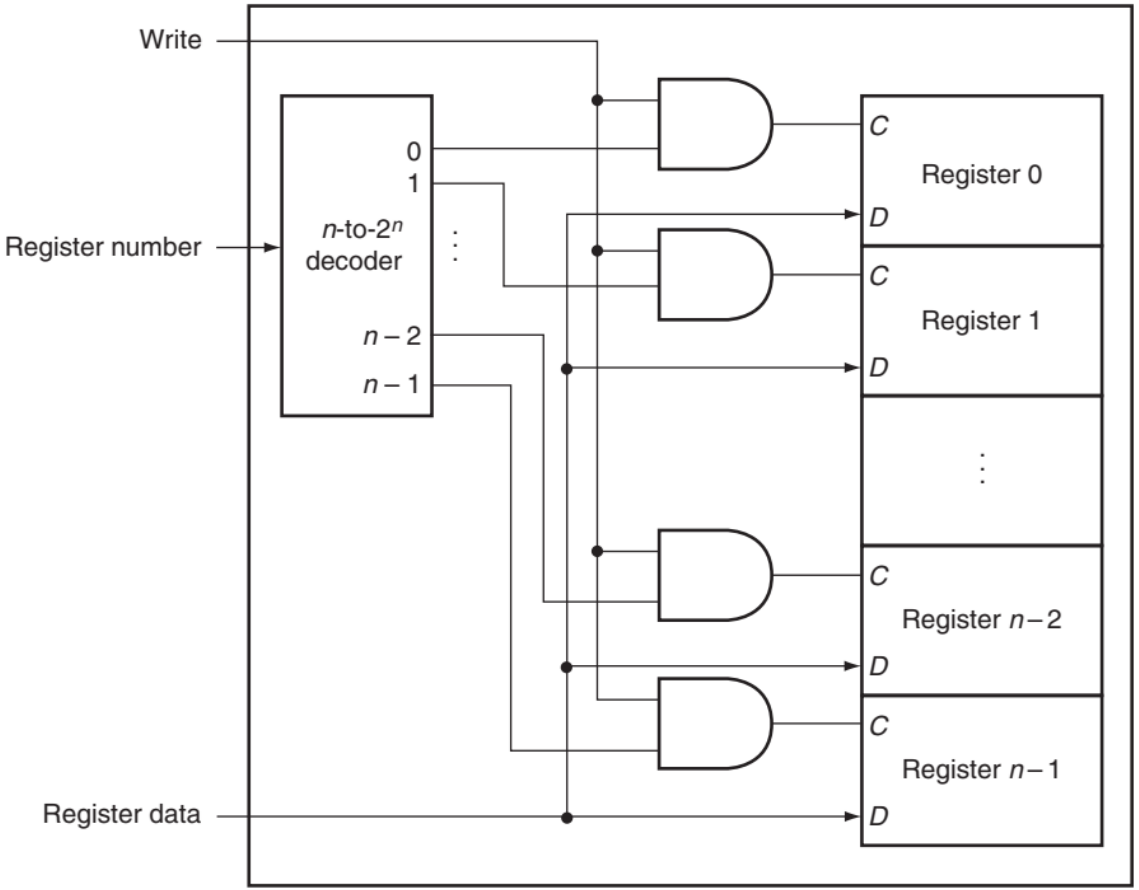
Two Register Read Ports

- 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)

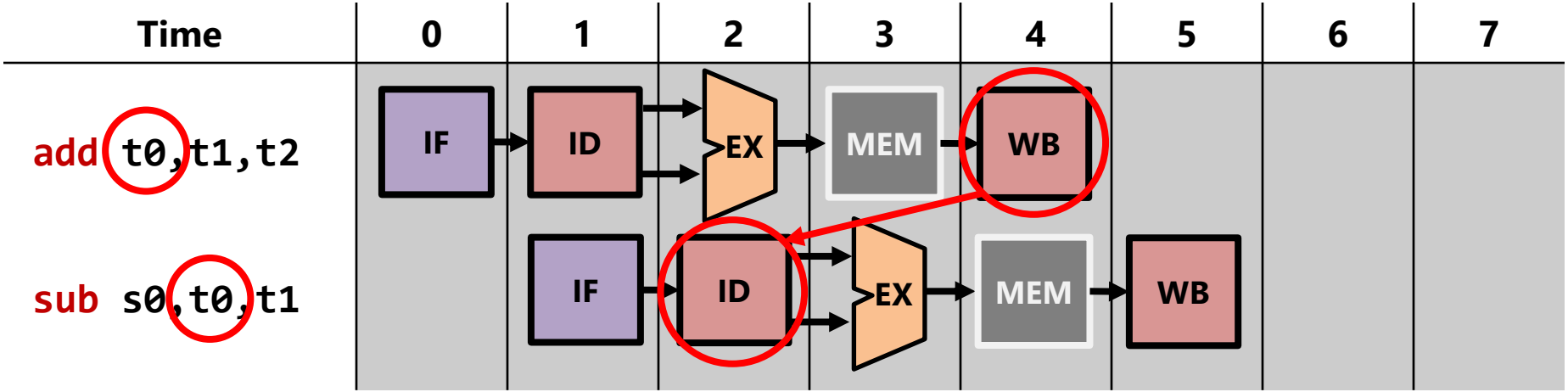
- But what if we want an IPC > 1?
  - More than one instruction per cycle! (a.k.a superscalar processor)
  - Must sustain more than one ID / WB stage per cycle
  - Need more register read ports and write ports!
  - Not only registers, memory would need more ports too!
  - Like everything else, this consumes lots of **power**

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

Solving Data Hazards

Data Hazards

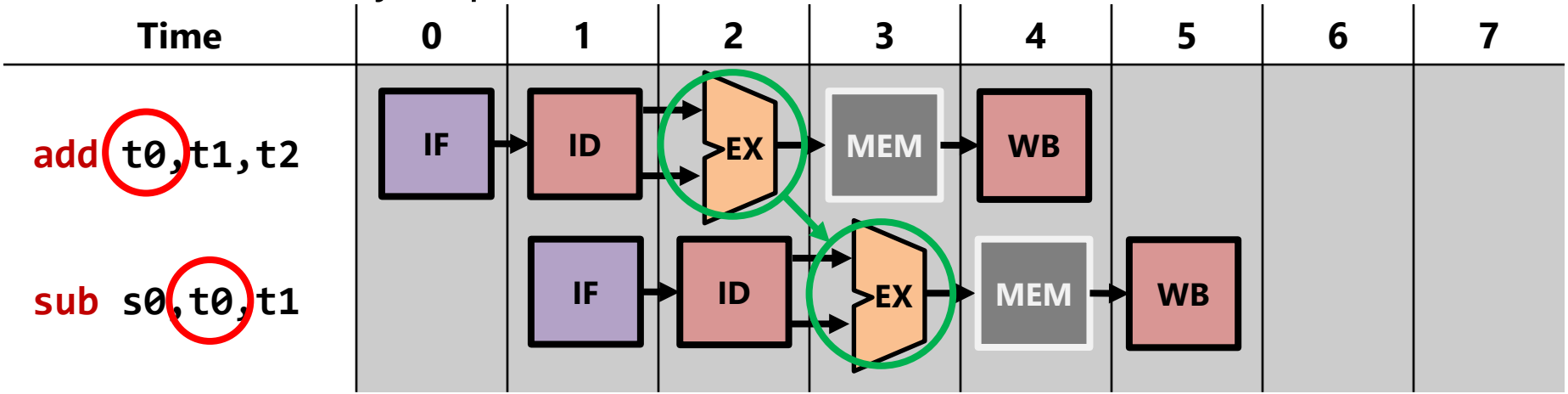
- An instruction depends on the output of a previous one.



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

Solution 1: Data Forwarding

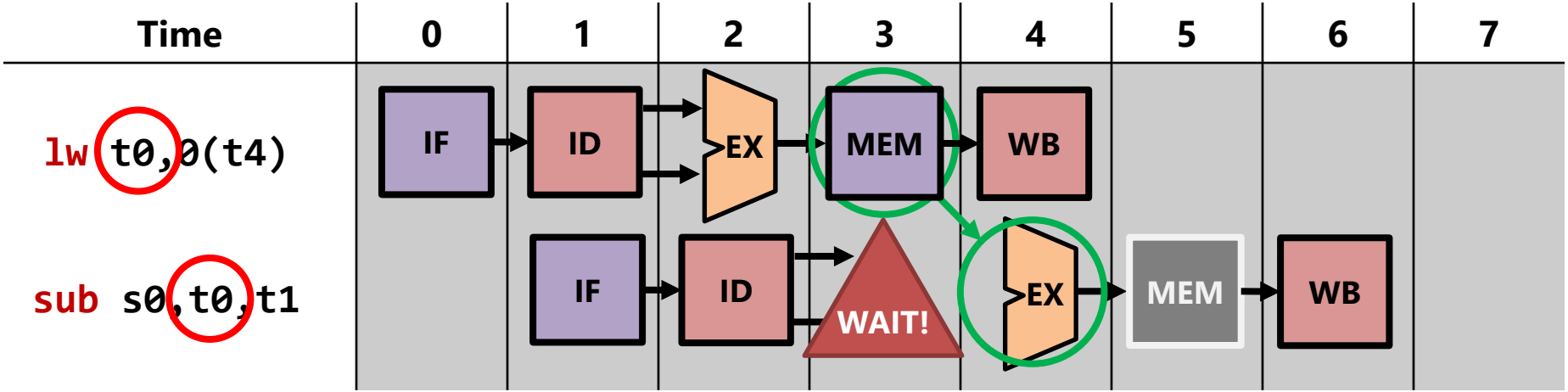
- Since we've pipelined control signals, we can check if instructions in the pipeline depend on each other (see if registers match).
- If we detect any dependencies, we can ***forward*** the needed data.



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

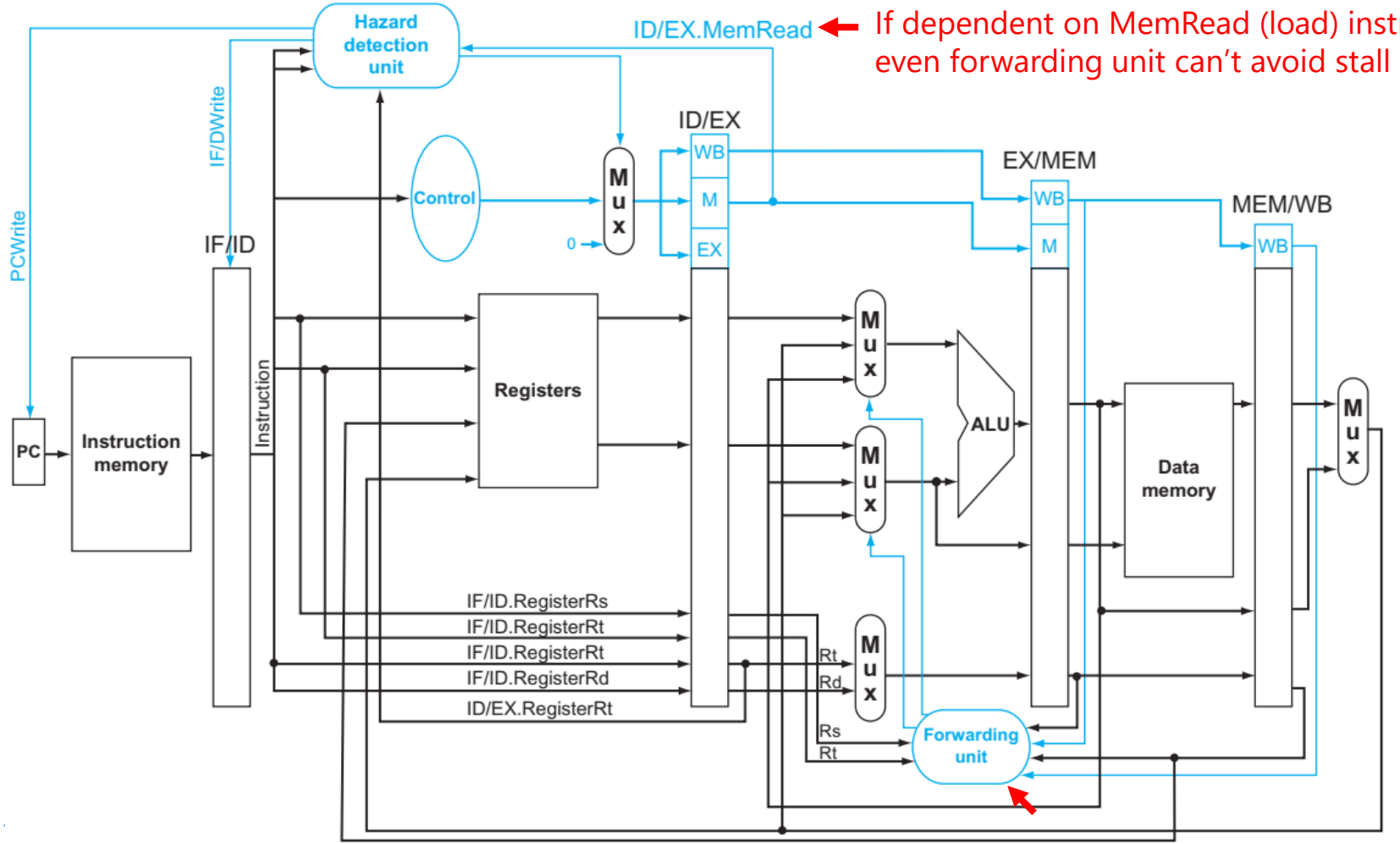
Data Forwarding from Memory

- 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

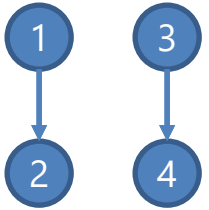
- Number of forwarding wires $\propto$ (pipeline stages)^2
  - Why the **quadratic** relationship?
  - Per pipeline stage, N stages after it *from* which data is forwarded
    - In previous picture, see number of inputs to MUX before ALU!
  - And there are N stages *to* which data must be forwarded
    - 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

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)*

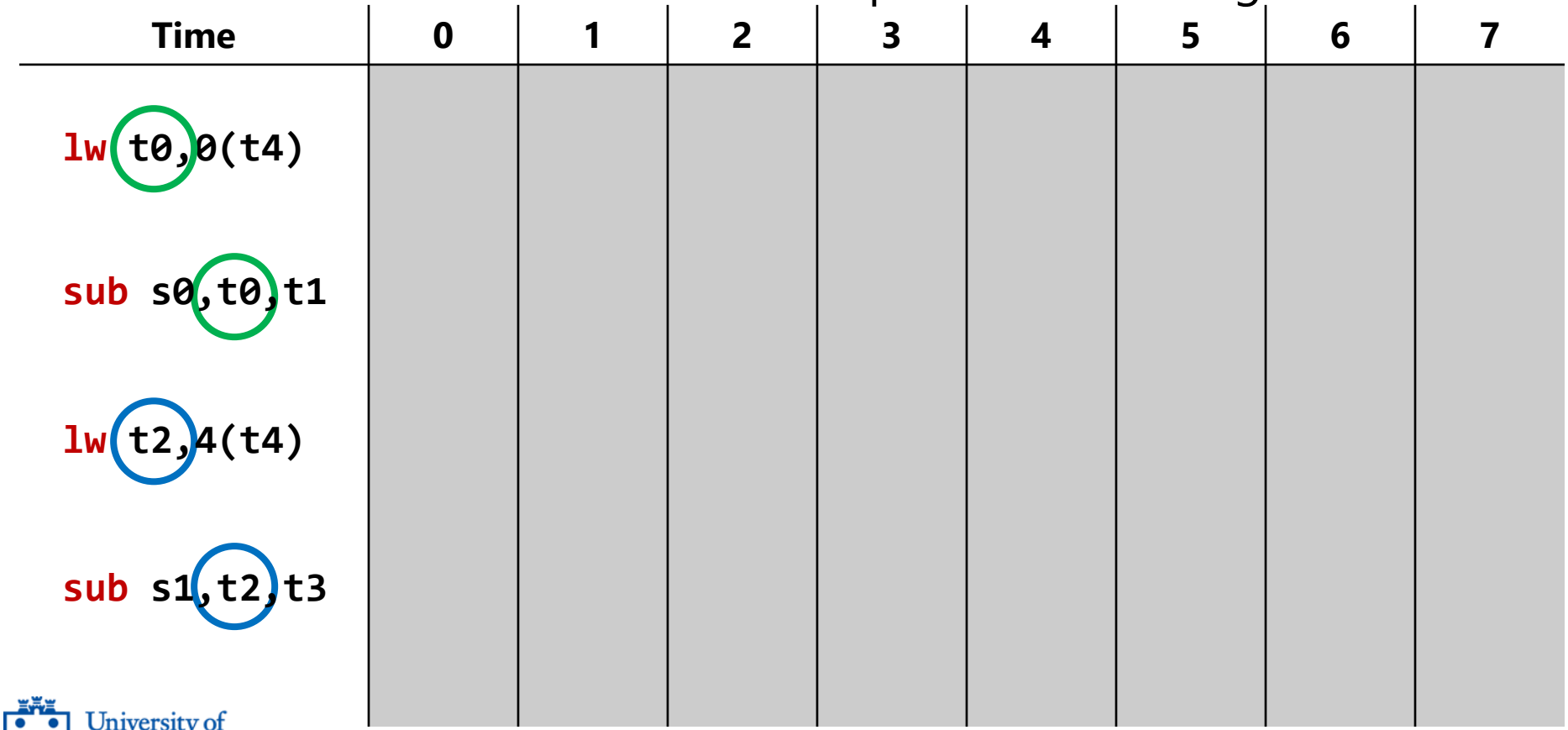
- Can you make this shorter? Yes! *(1 hour total)*
  1. Have washing machine running *and* 3. Tea boiling *(30 minutes)*
  2. Have dryer running *and* 4. Drink tea *(30 minutes)*



- 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

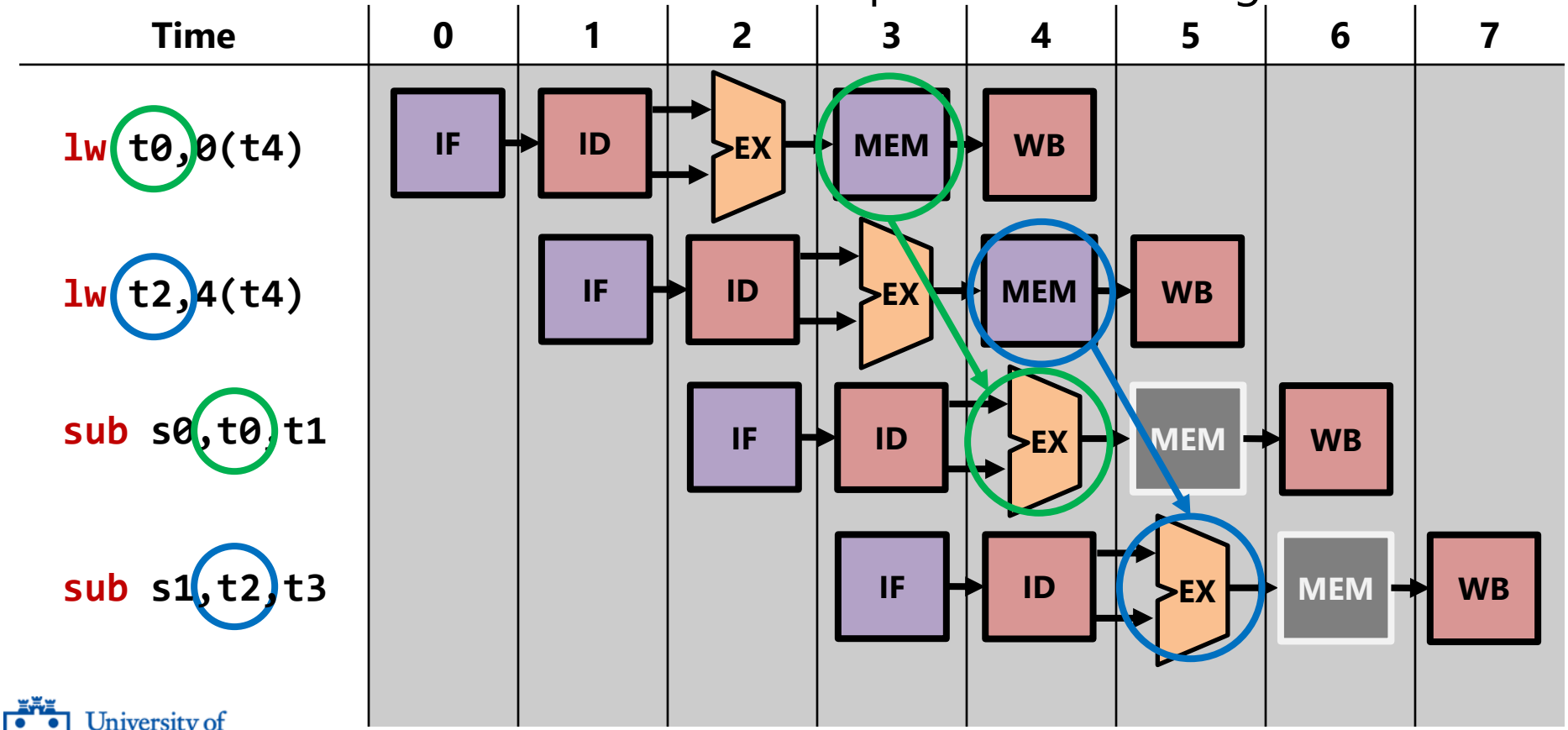
Data Hazard removed through Compiler Reordering

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



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.



Limits of Static Scheduling

- 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 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(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

- Loops happen *all the time* in programs.

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

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

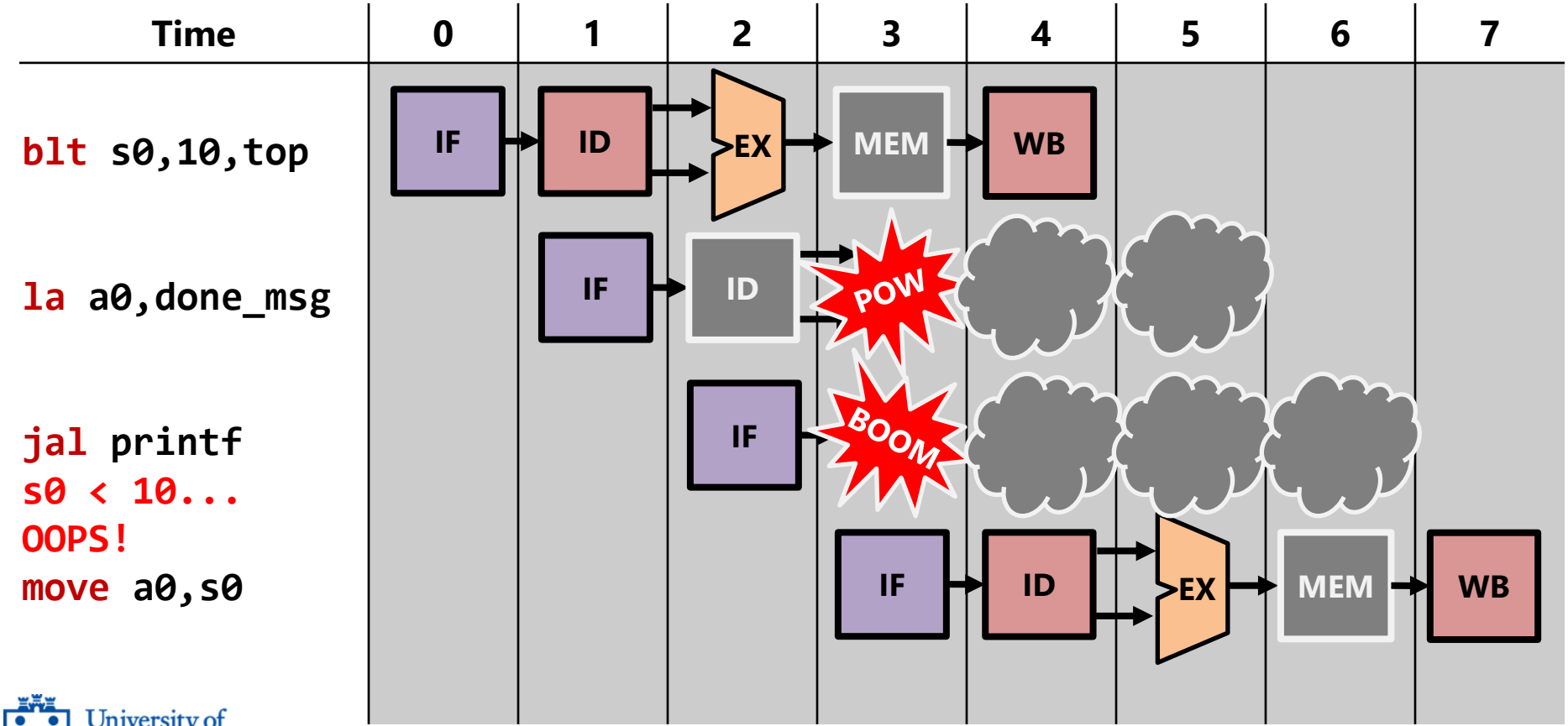
```
    li   s0, 0
top:
    move a0, s0
    jal  print
    addi s0, s0, 1
    blt  s0, 10, top
```

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

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

Pipeline Flushes at Every Loop Iteration

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



Performance Impact from Control Hazards

- **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 MEM
  - What if there are **4** ID stages and **3** EX stages? Penalty == 7!

- 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
  - α : fraction of branch instructions in the instruction mix
  - π : probability a branch is actually taken
  - K : penalty per pipeline flush

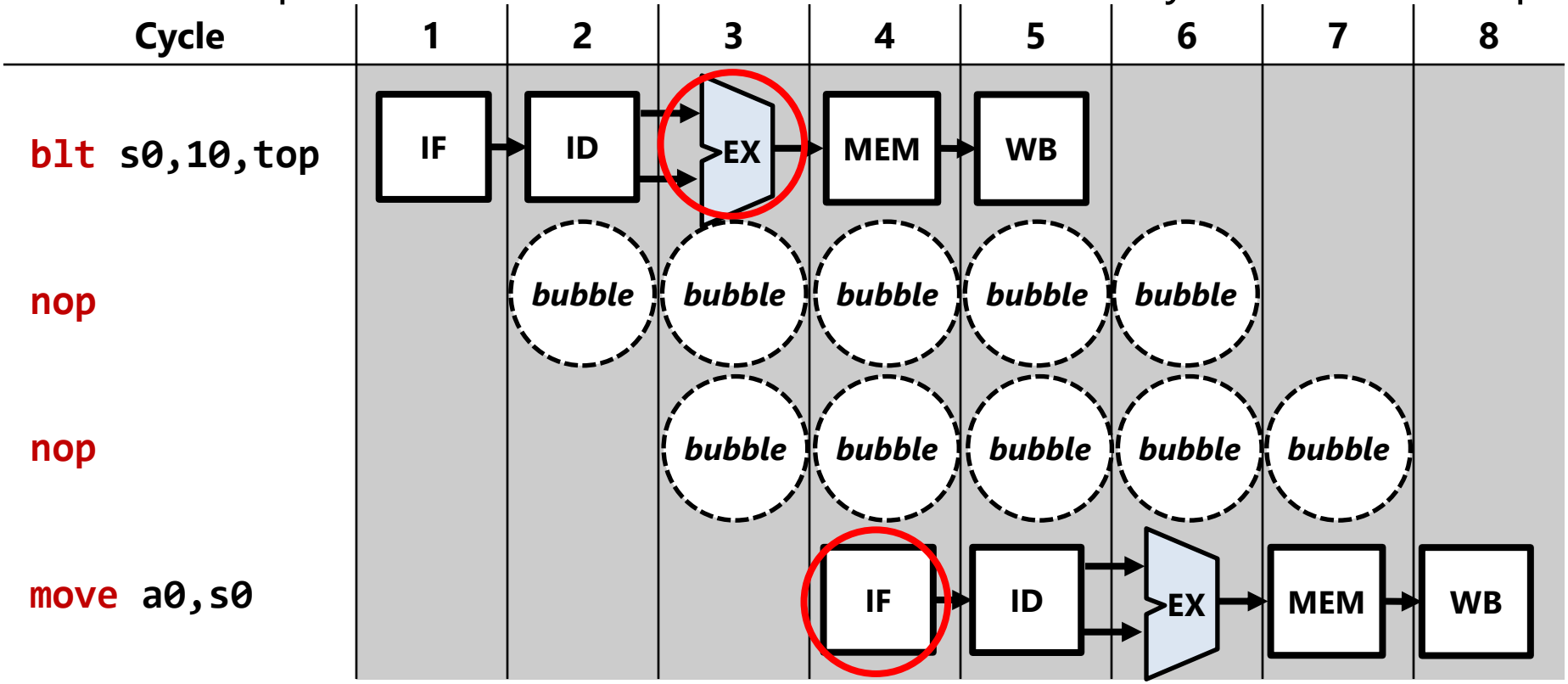
Example: If 20% of instructions are branches and the probability that a branch is taken is 50%, and pipeline flush penalty 7 cycles, then:

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

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

Compiler avoiding the control hazard without HDU

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



Performance Impact without Hazard Detection Unit

- $CPI = CPI_{nch} + \alpha * K$
  - CPI_{nch} : CPI with no control hazard
  - α : 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?

Solution 1: Delay Slots

- 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

Compiler static scheduling using delay slots

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

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

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

Delay slots are losing popularity

- 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
   - Often data and control independent instructions don't exist

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

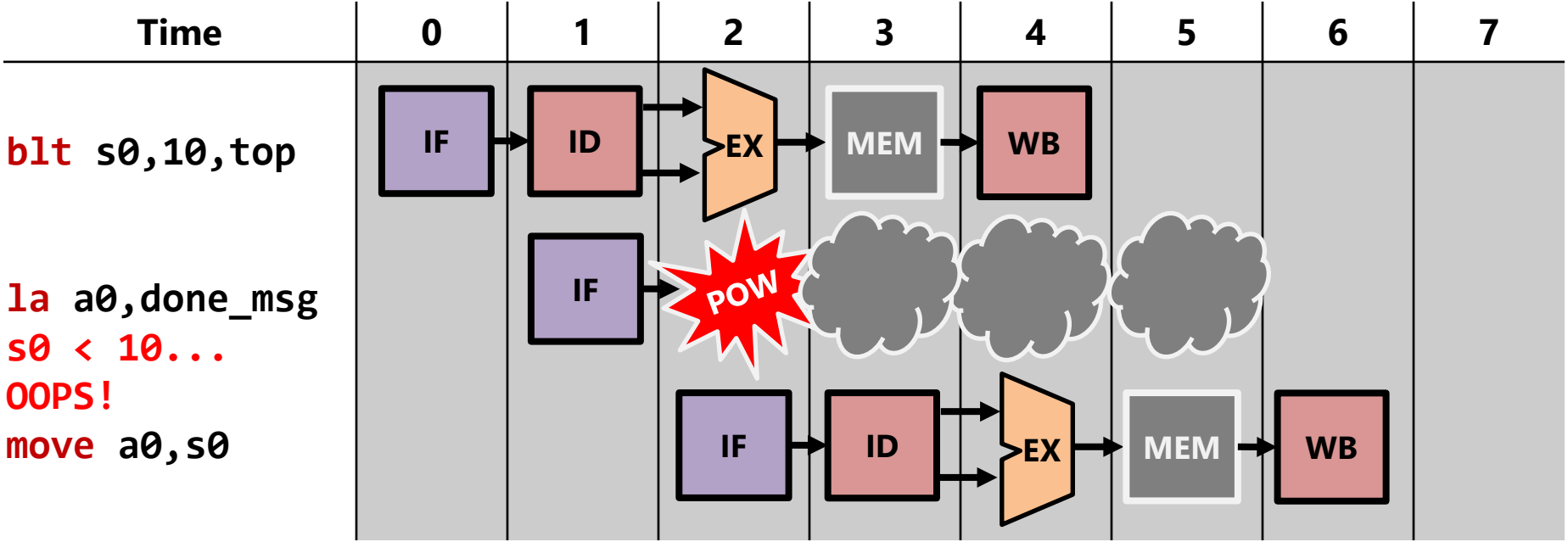
- Next idea please!

Solution 2: MORE SINKS! (a.k.a. hardware)

University of Pittsburgh

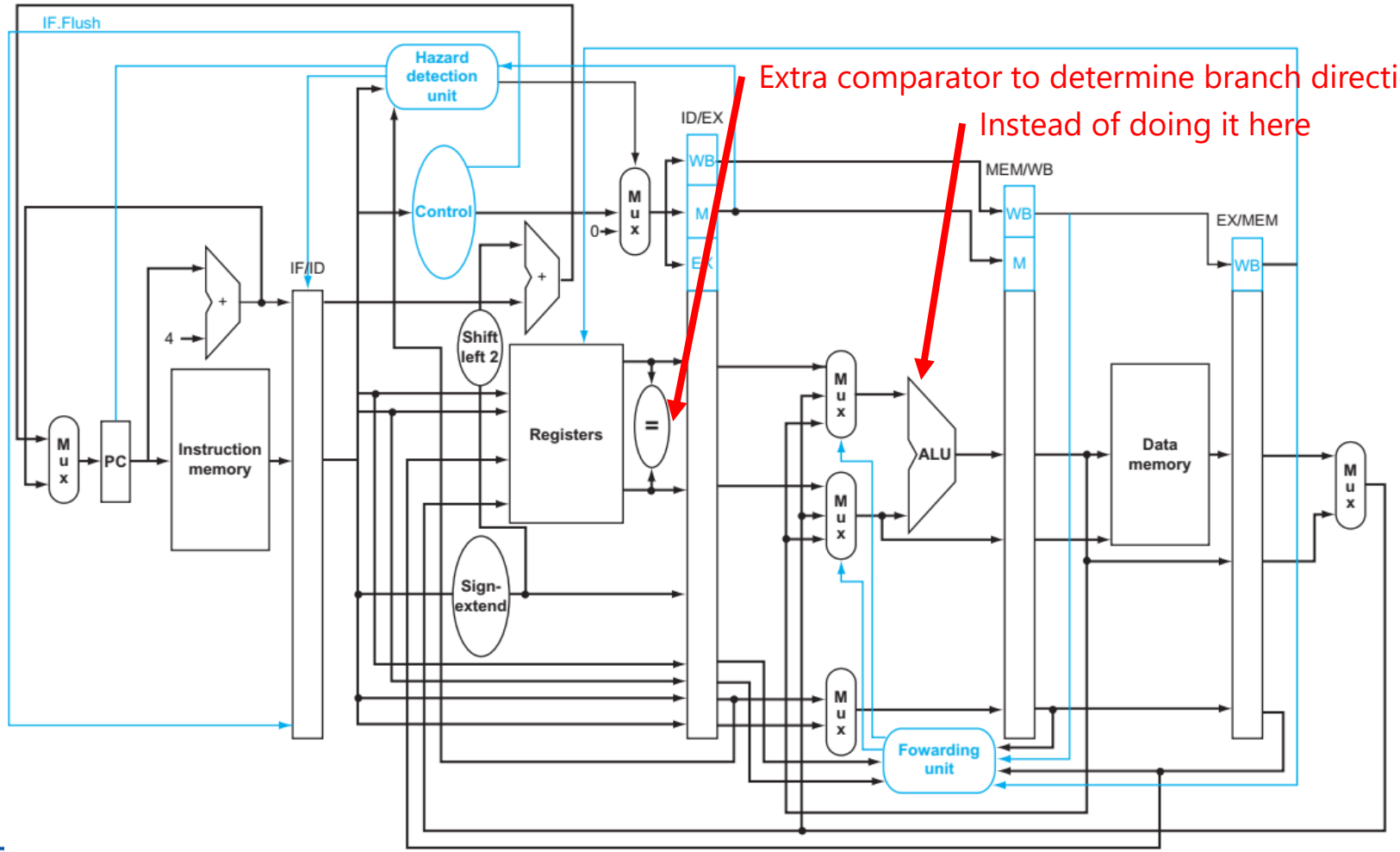
Do we *reeeally* need to compare at EX stage?

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



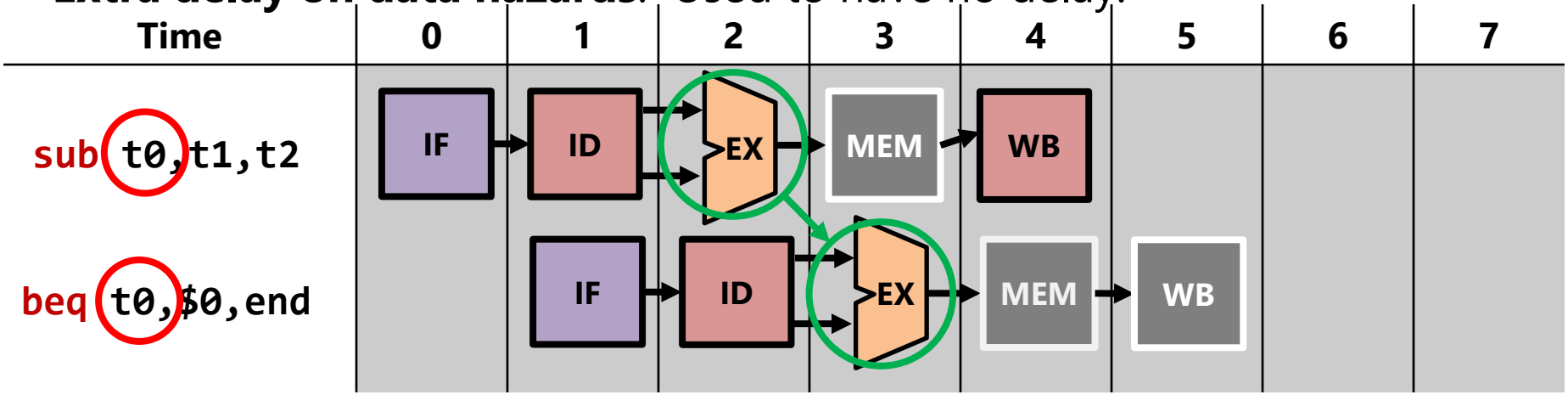
- Reduced penalty from 2 cycles → 1 cycle!
- But of course that means we need a comparator at the ID stage

Solution 2: MORE SINKS! (a.k.a. hardware)

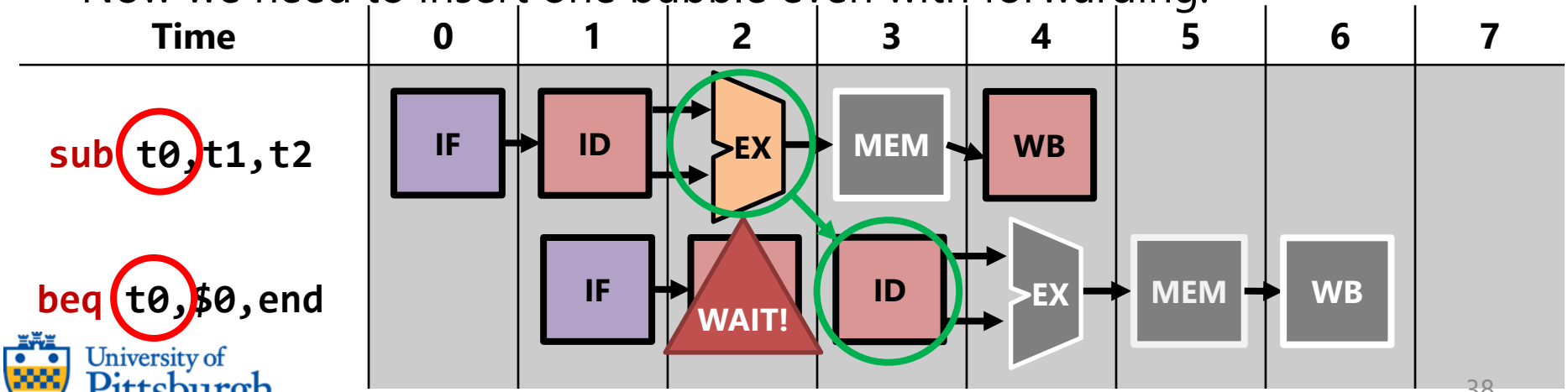


Not all sunshine and rainbows

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



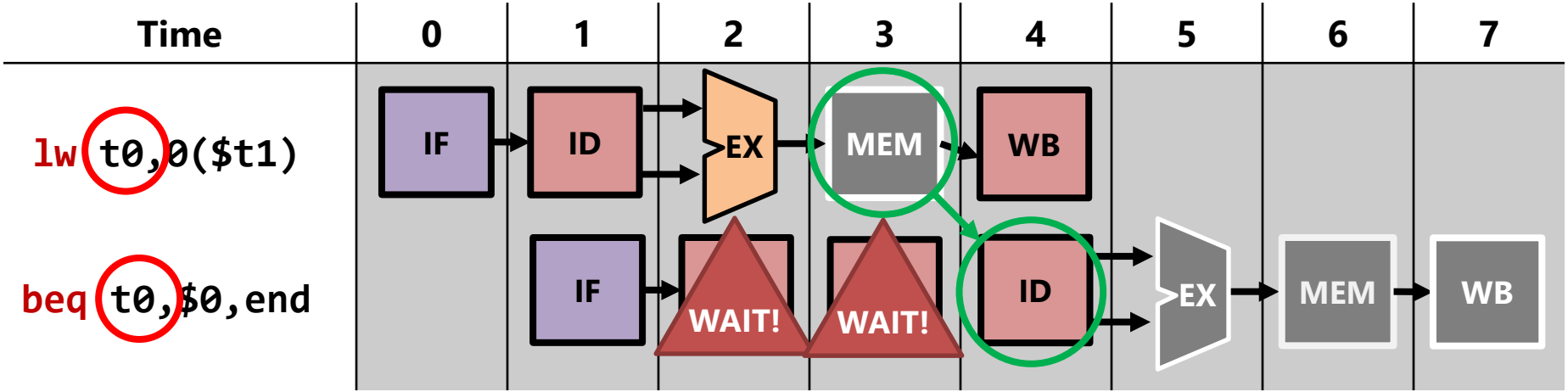
- Now we need to insert one bubble even with forwarding:



University of Pittsburgh

Not all sunshine and rainbows

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



- 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

Textbook figure correction