CPU Pipelining

CS/COE 1541 (Fall 2020)
Wonsun Ahn

University of Pittsburgh

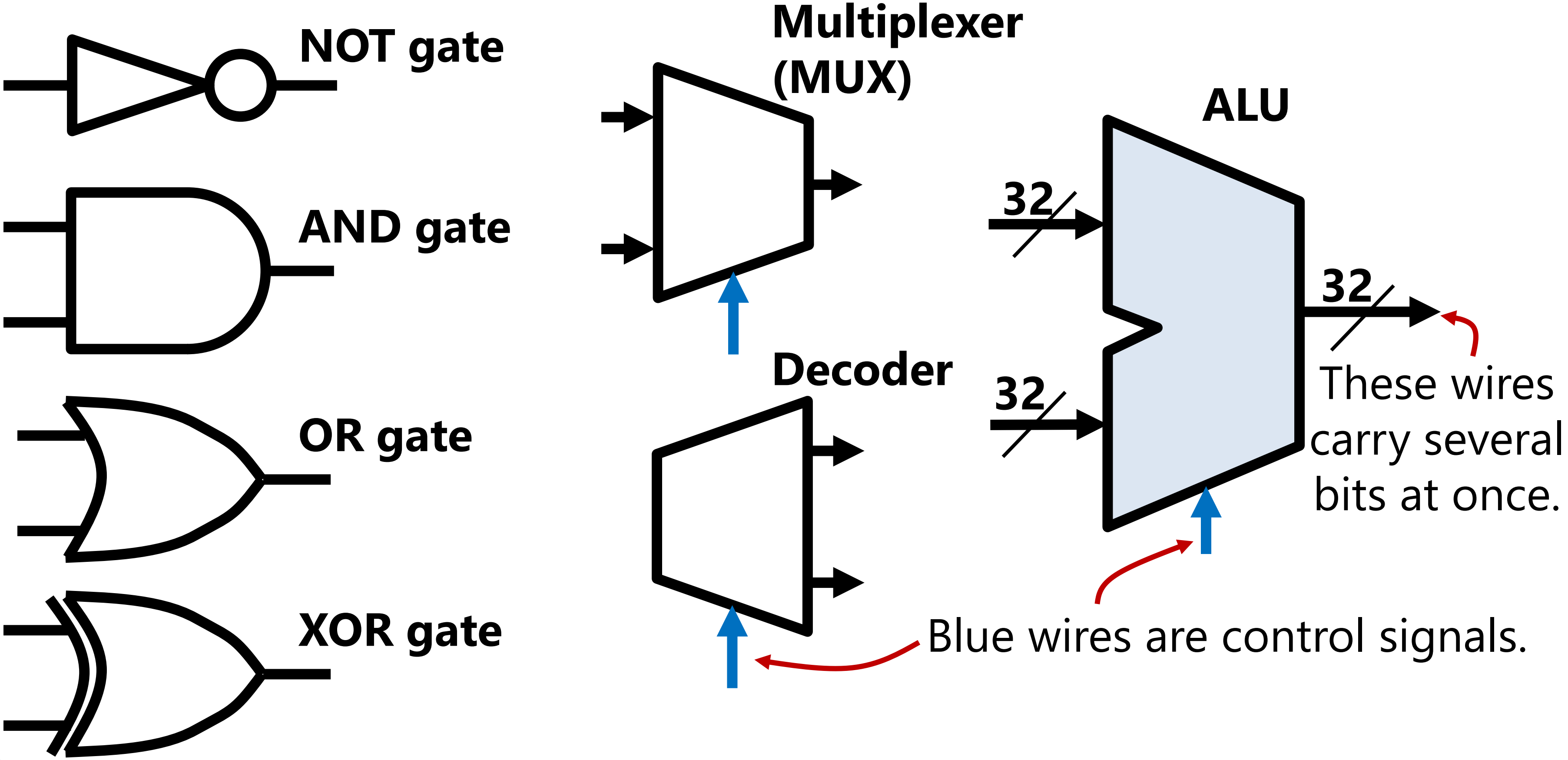
Clocking Review

Stuff you learned in CS 447

University of Pittsburgh

Logic components

- Do you remember what all these do?

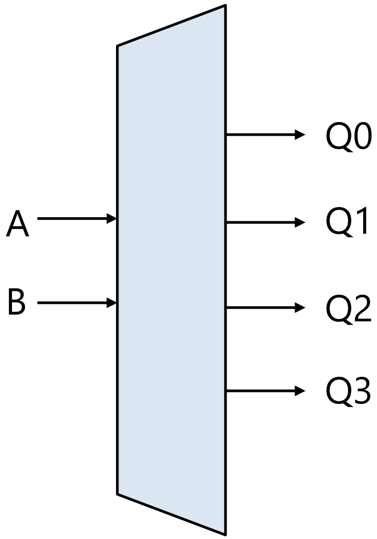


These wires carry several bits at once.

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:



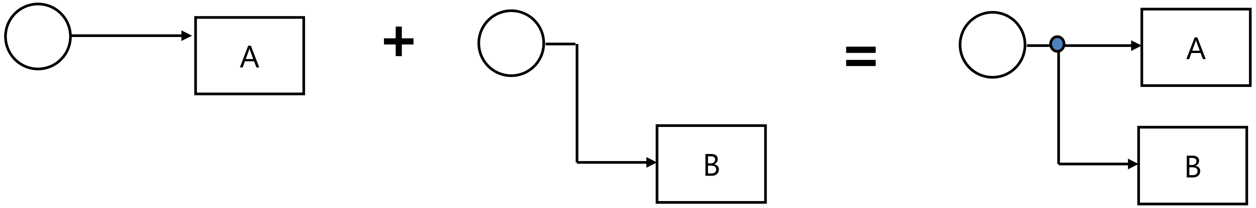
Truth Table for Decoder

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

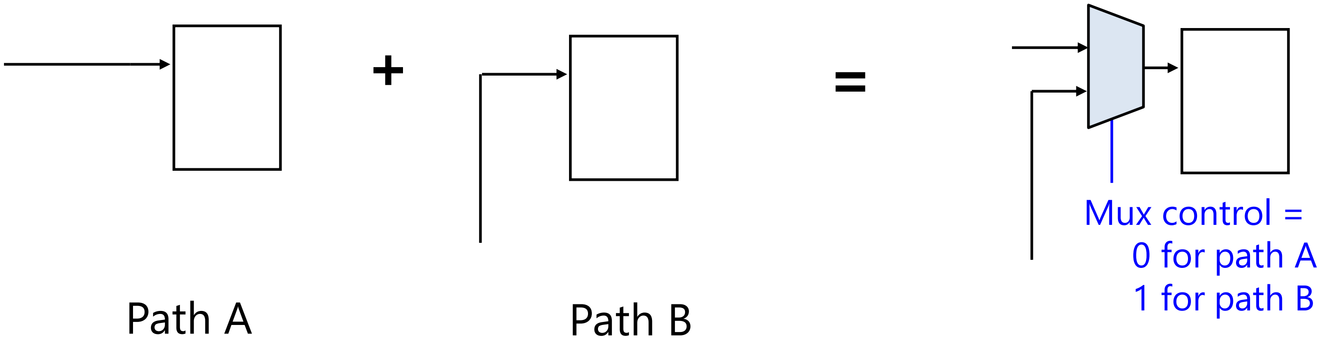
  - You can come up with any truth table and make a decoder for it!

Uses of a Multiplexer

- No problem in fanning out one signal to two points

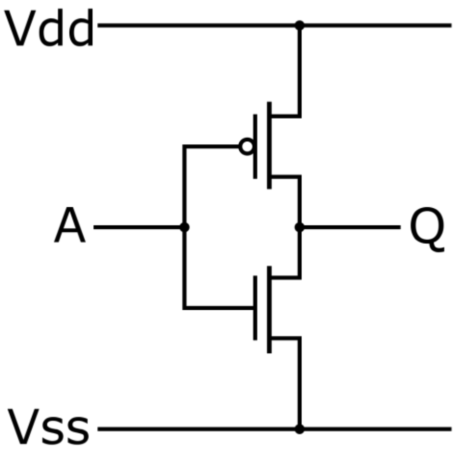


- Cannot connect two signals to one point
  - Must use a multiplexer to *select* between the two

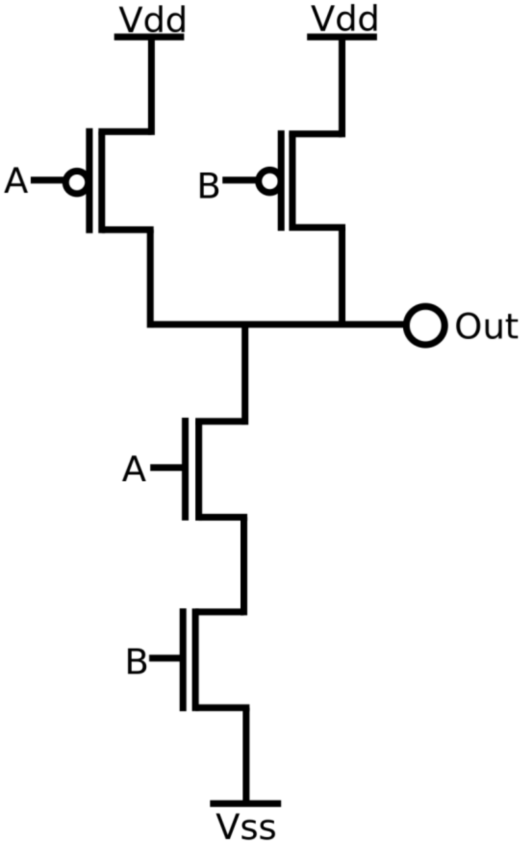


Gates are made of transistors (of course)

- NOT gate

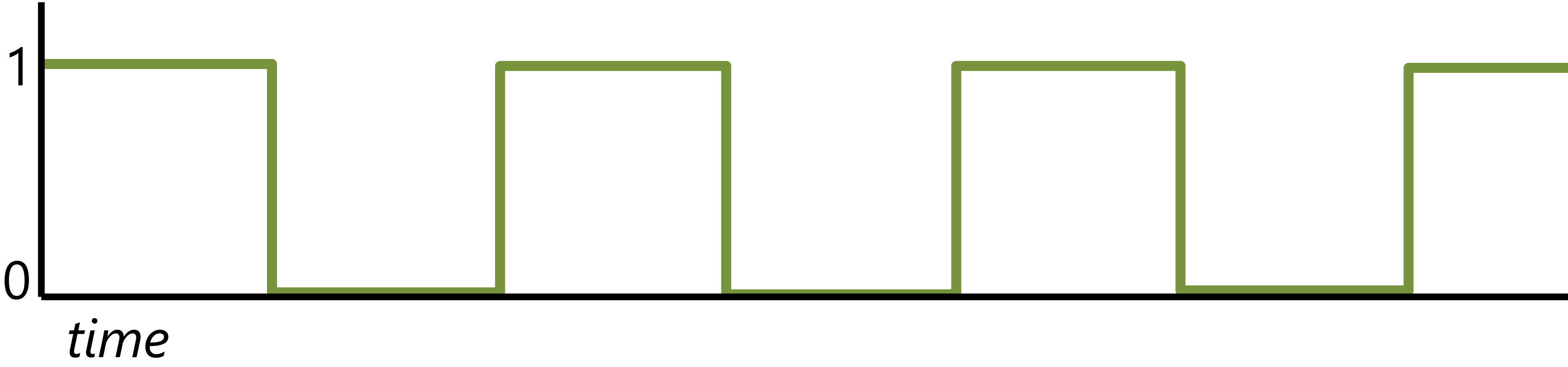


- NAND gate



The clock signal

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

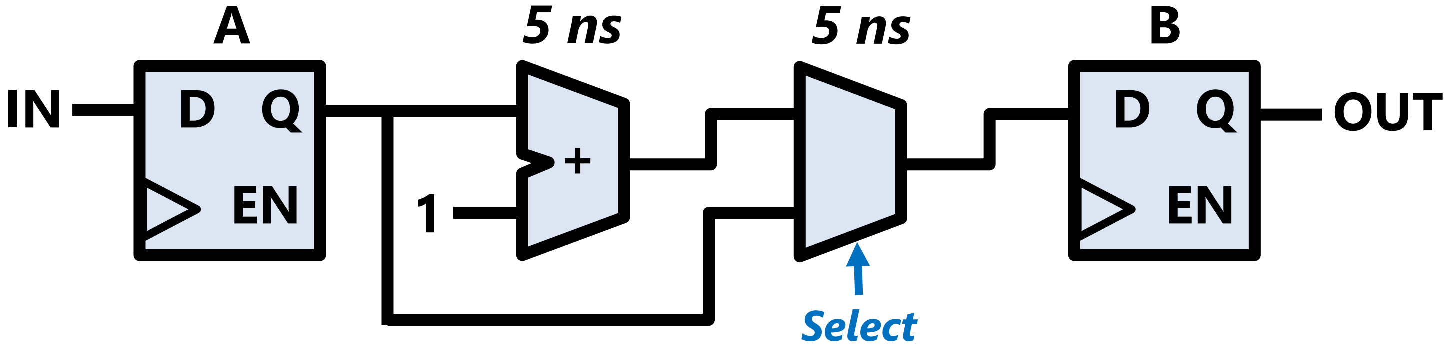


- Bits are latched on to registers and flip-flops on rising edges

- 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 ns (5 × 10^{-9}s) = 200 MHz?
  - Or is it 1 / 10 ns (10 × 10^{-9}s) = 100 MHz? ✓

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

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

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)

The MIPS ISA - Memory

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

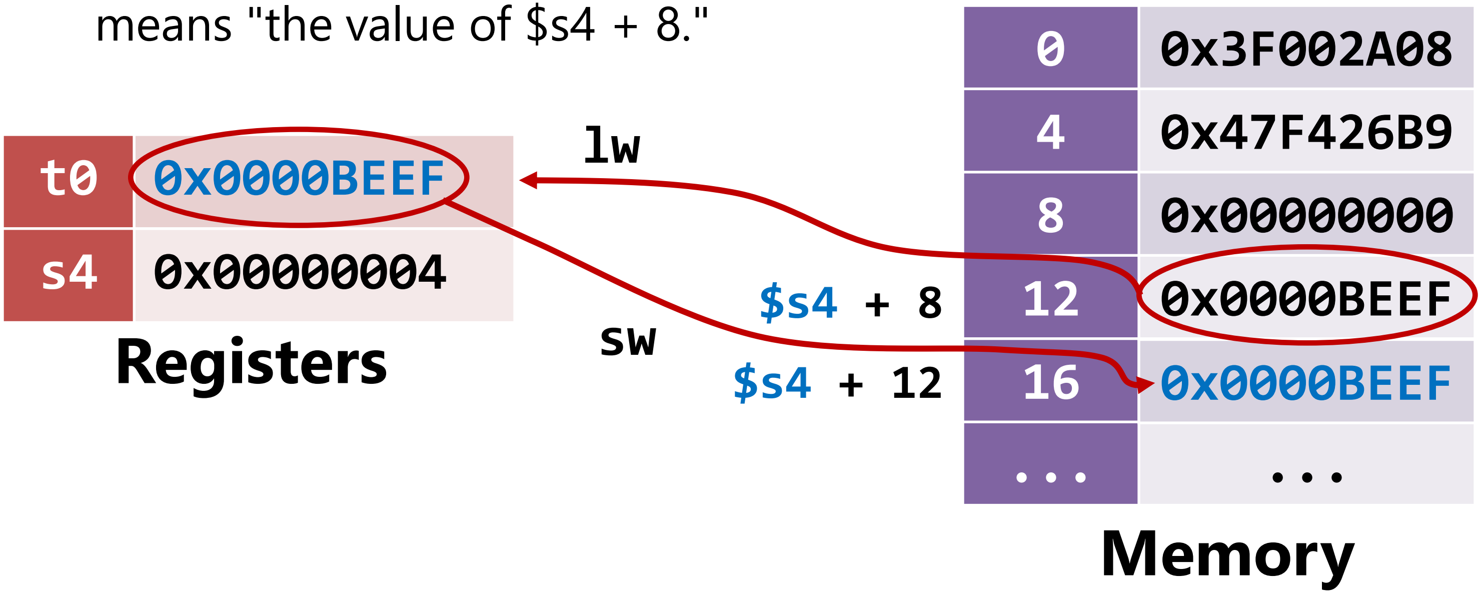
```
lw $t0, 8($s4)
```

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

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

```
sw $t0, 12($s4)
```

$t0 is the SOURCE!



t0	0x0000BEEF
s4	0x00000004

Registers

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
```

```
    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`**

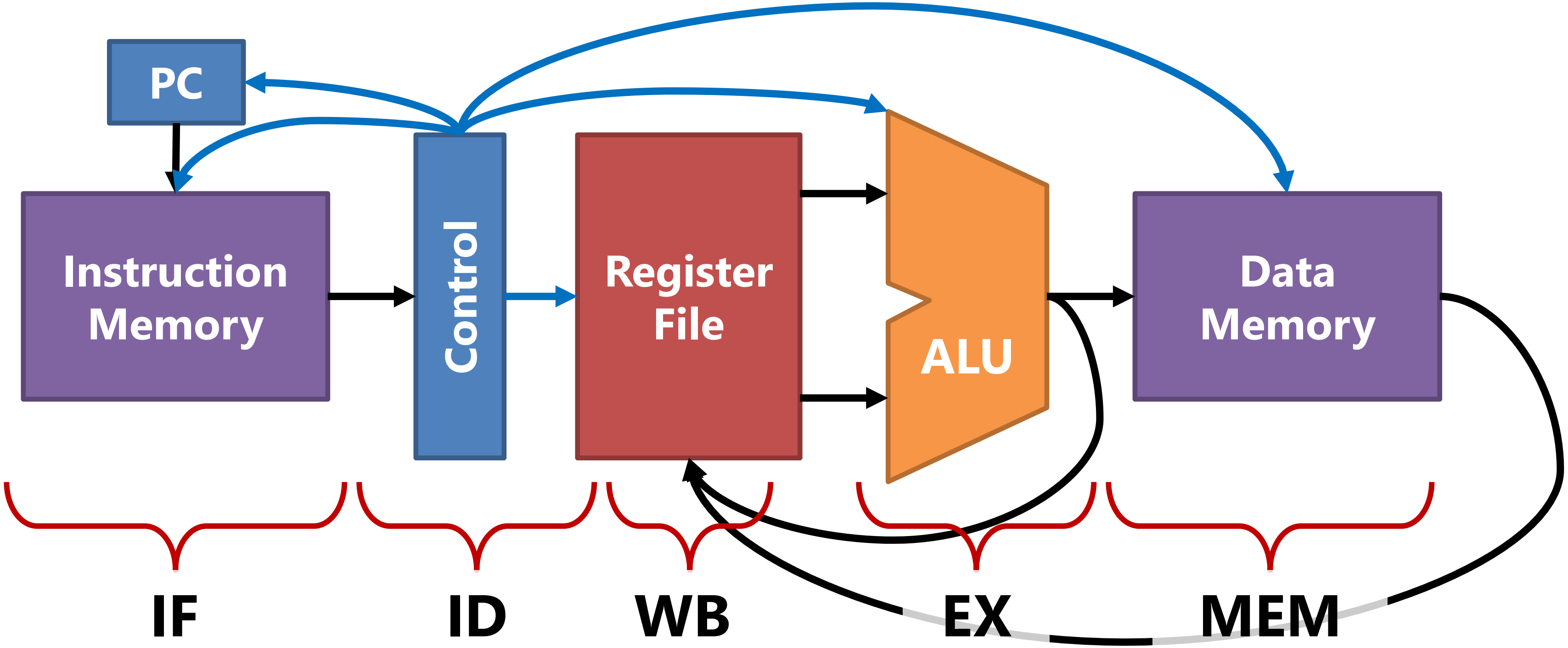
If `$s0` == `$zero`, continues to `jr   $ra`

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

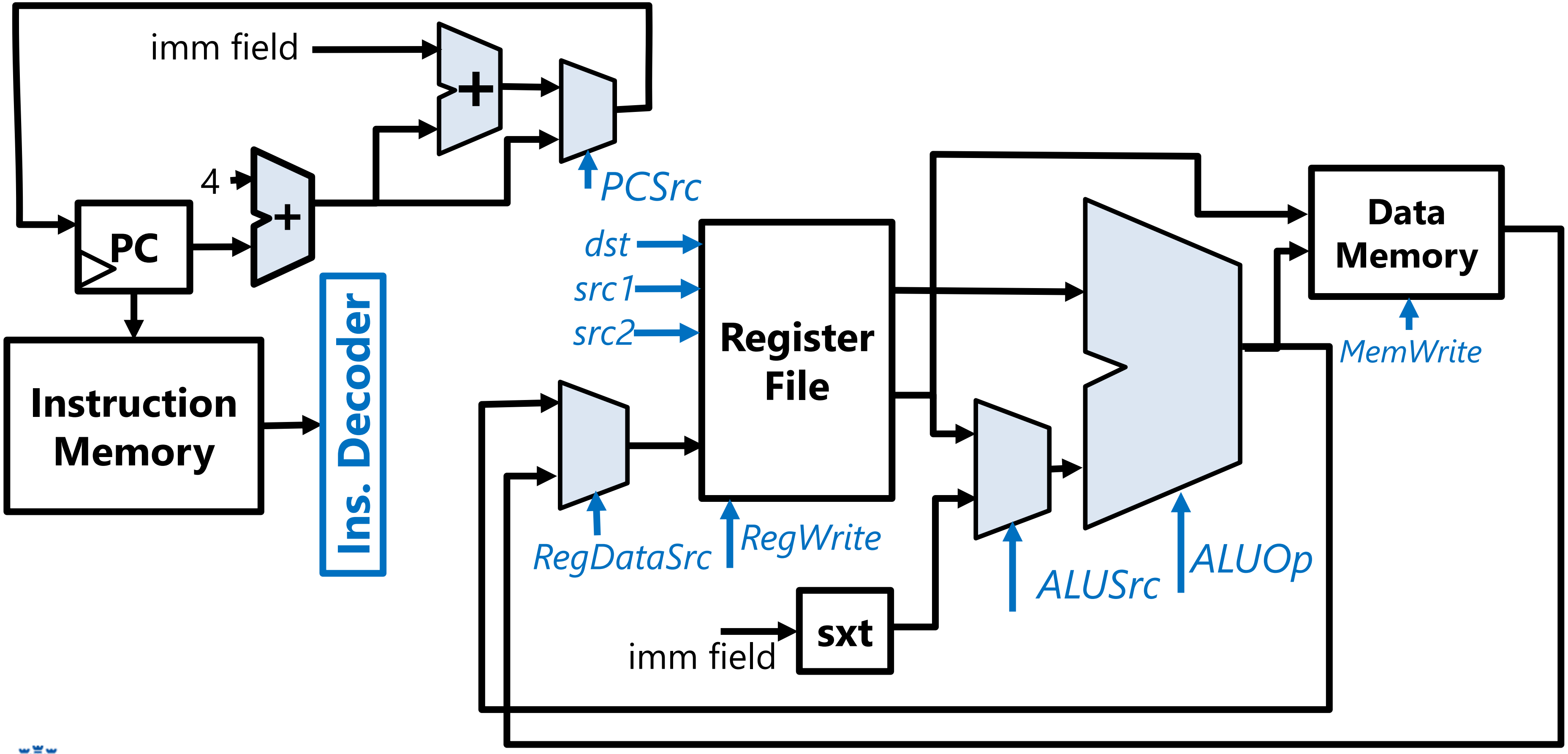
- For pedagogical (teaching) purposes

- 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

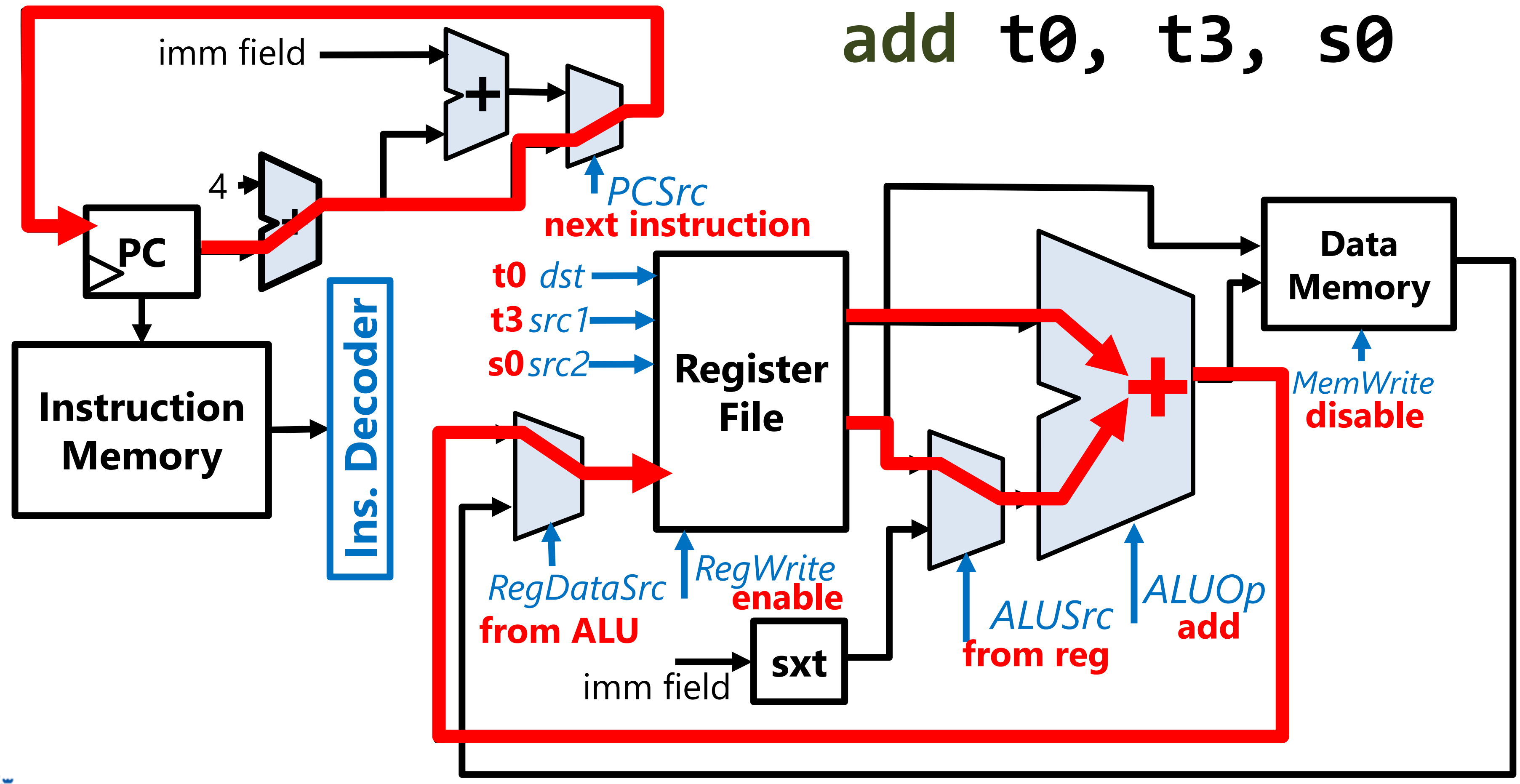


Control signals

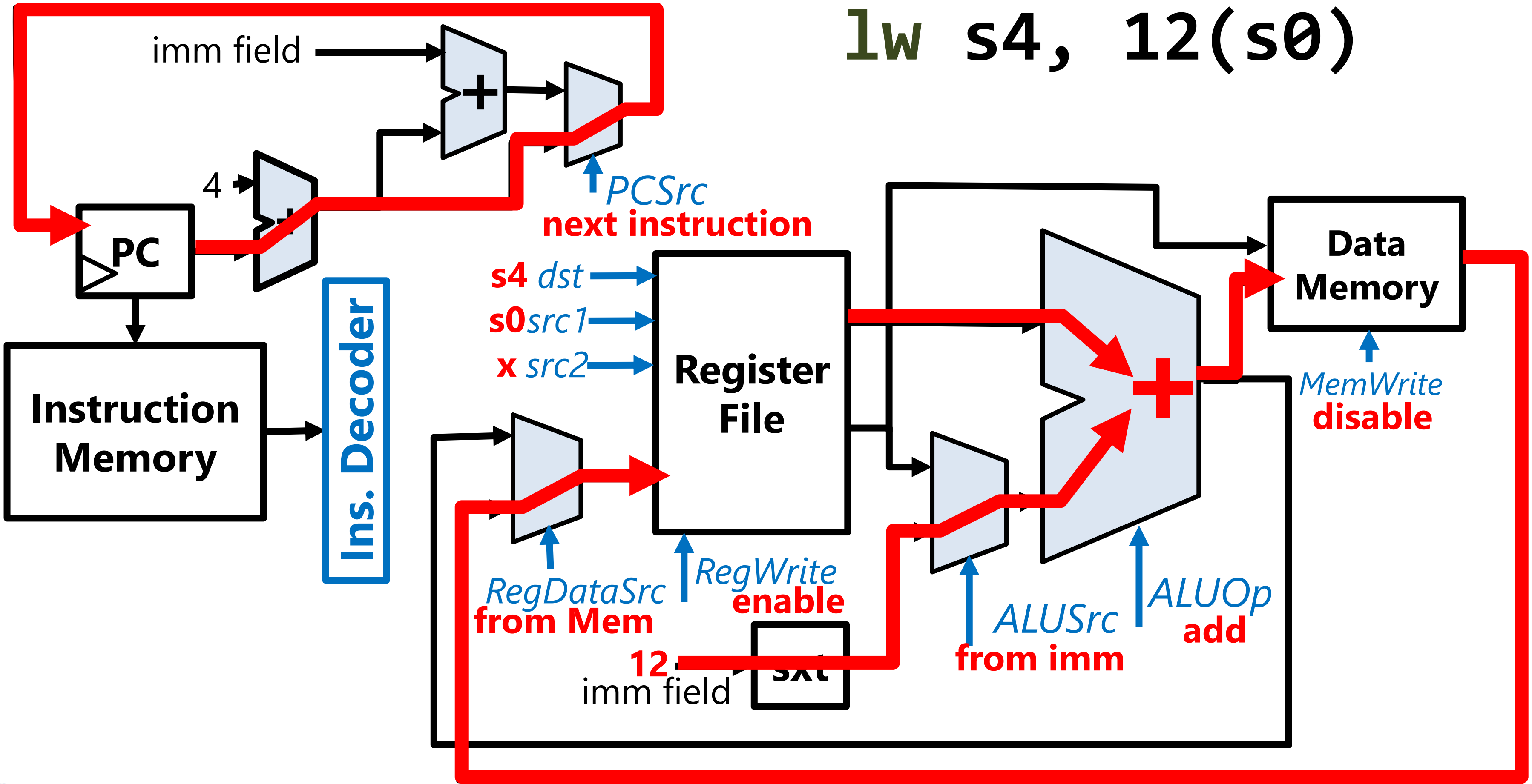
- Registers
  - **RegDataSrc**: controls source of a register write (ALU / memory)
  - **RegWrite**: enables a write to the register file
  - **src1, src2, dst**: the register number for each respective operand
- ALU
  - **ALUSrc**: whether second operand of ALU is a register / immediate
  - **ALUOp**: 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

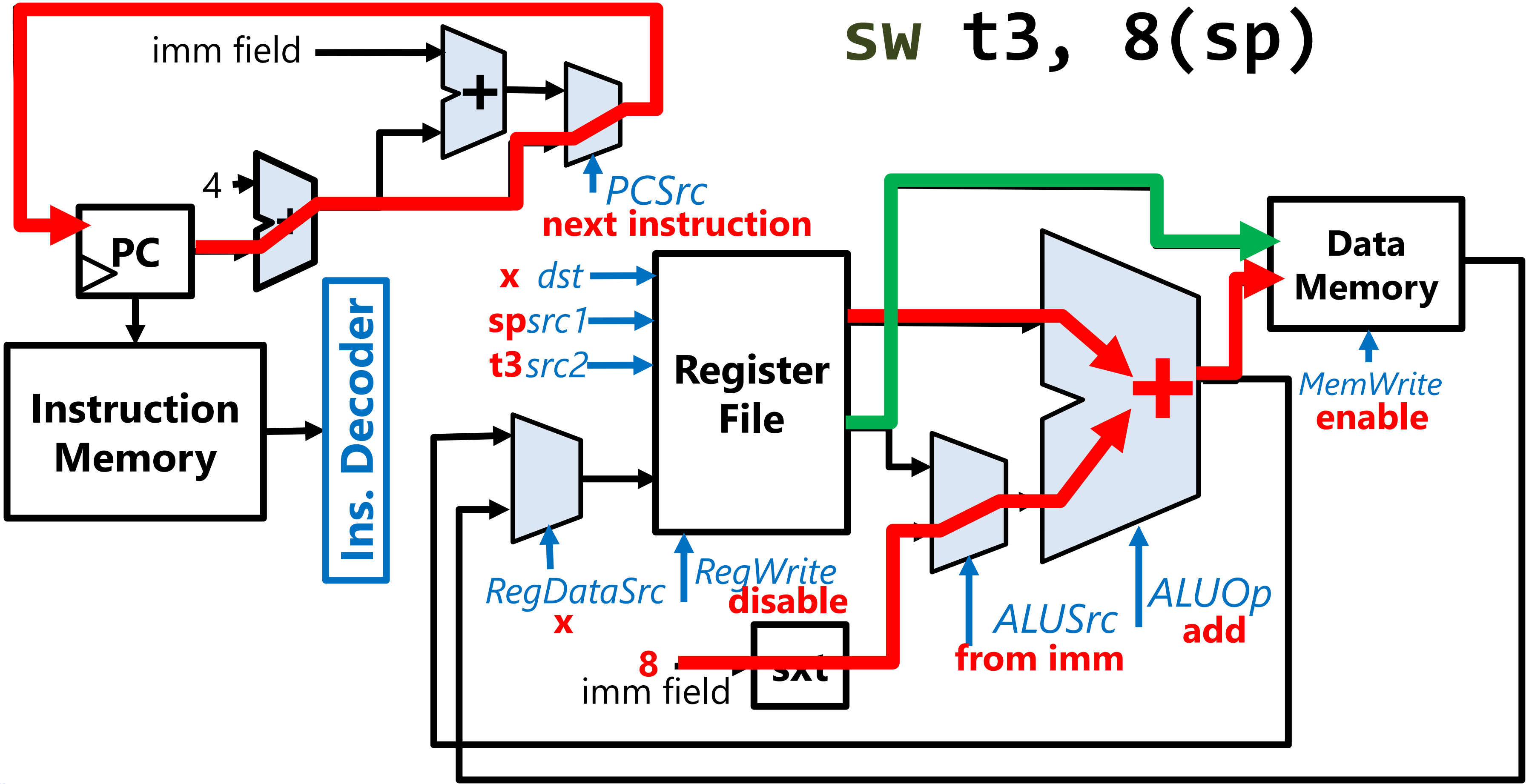


How an **lw** works



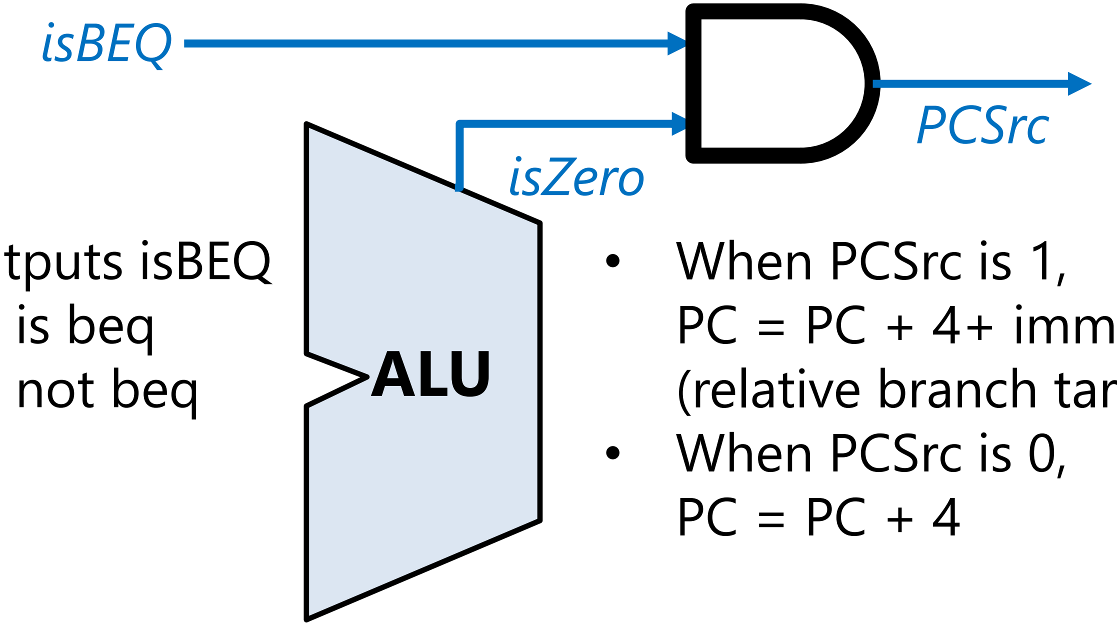
How an **sw** works

`sw t3, 8(sp)`



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

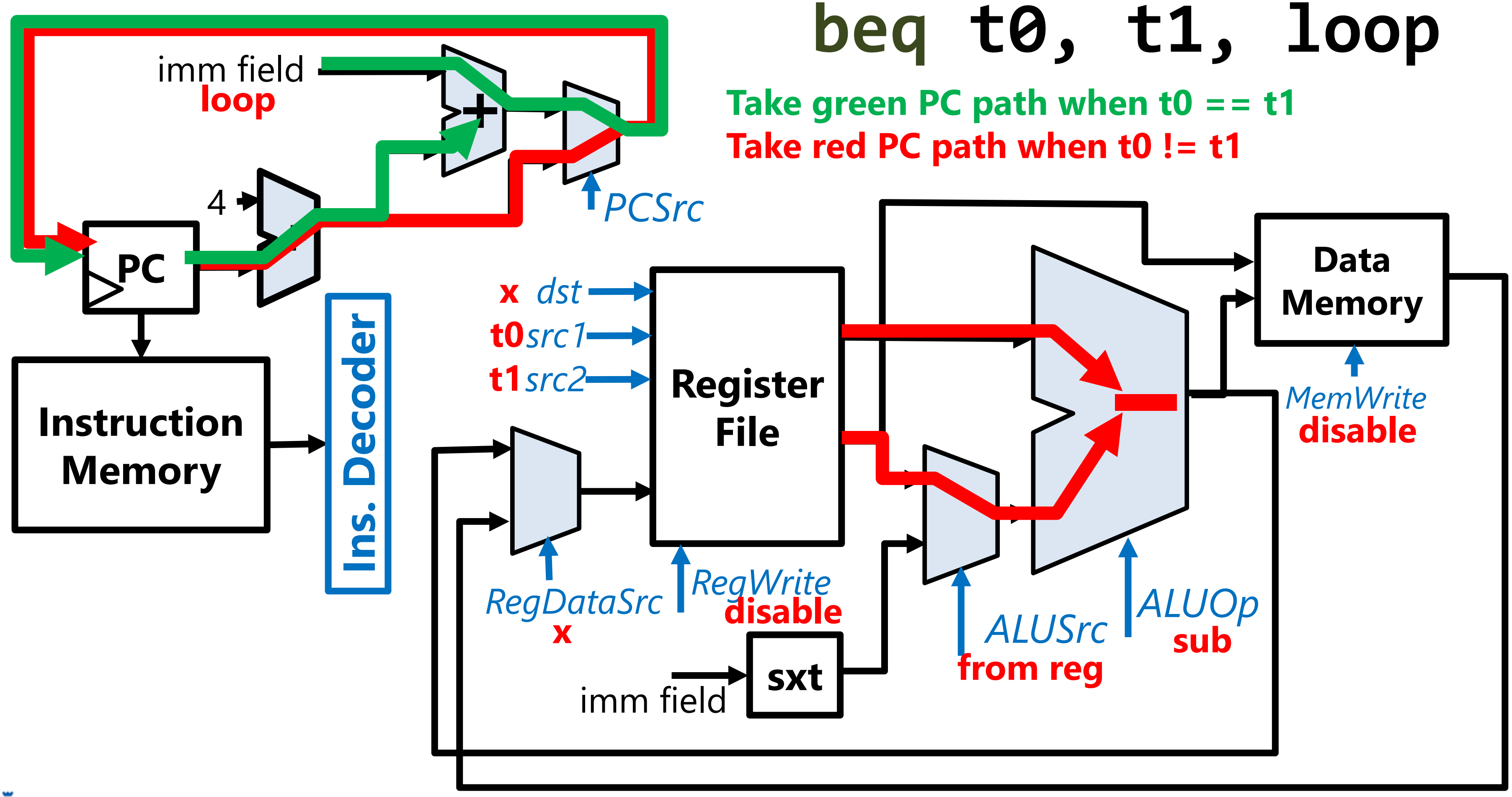
- When PCSrc is 1, PC = PC + 4+ imm (relative branch target)
- When PCSrc is 0, PC = PC + 4

How a **beq** works

```
beq t0, t1, loop
```

Take green PC path when t0 == t1

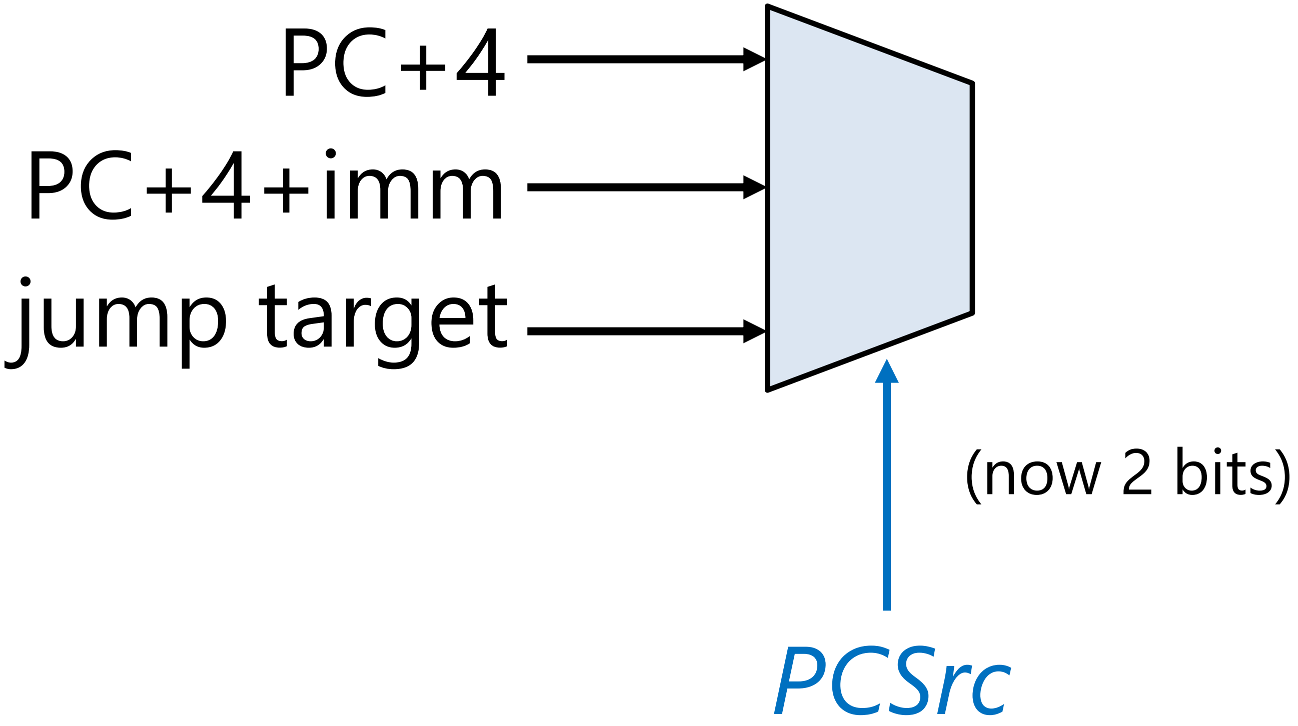
Take red PC path when t0 != t1



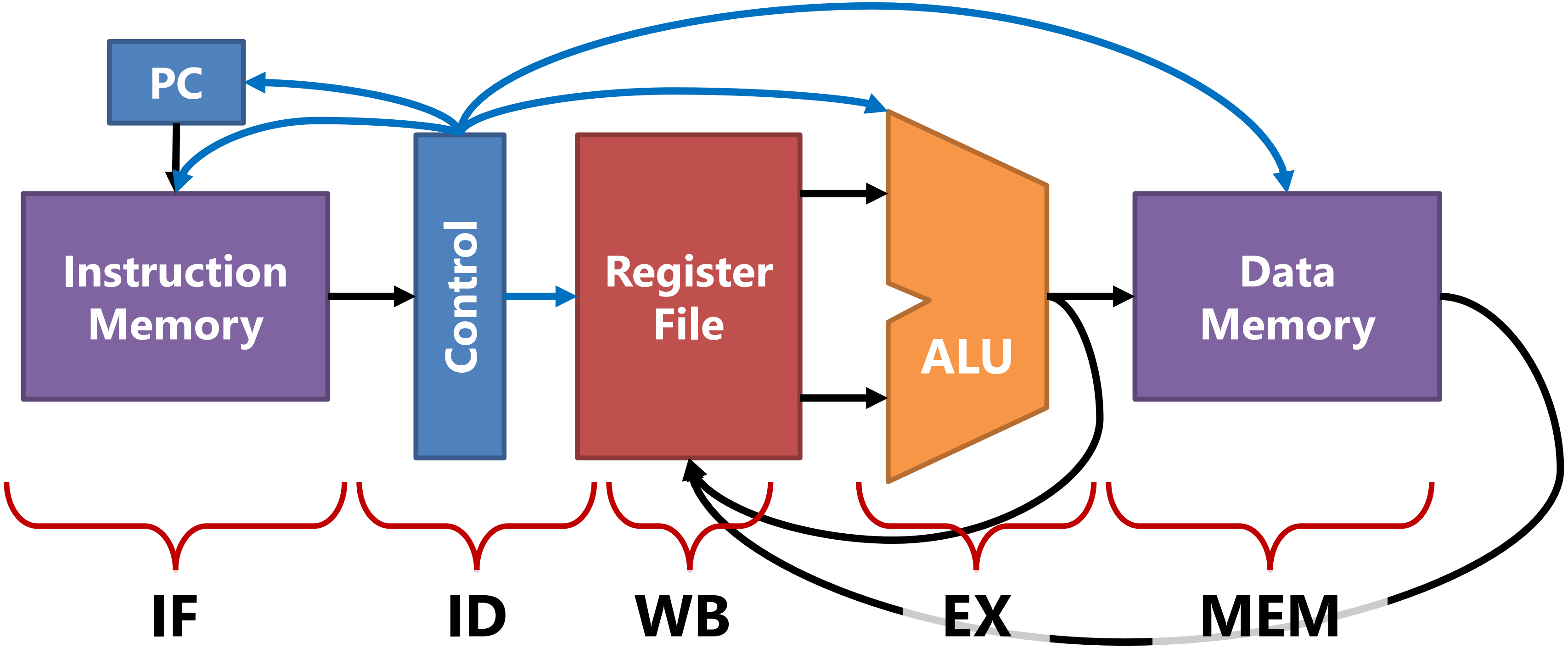
What about j?

- 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

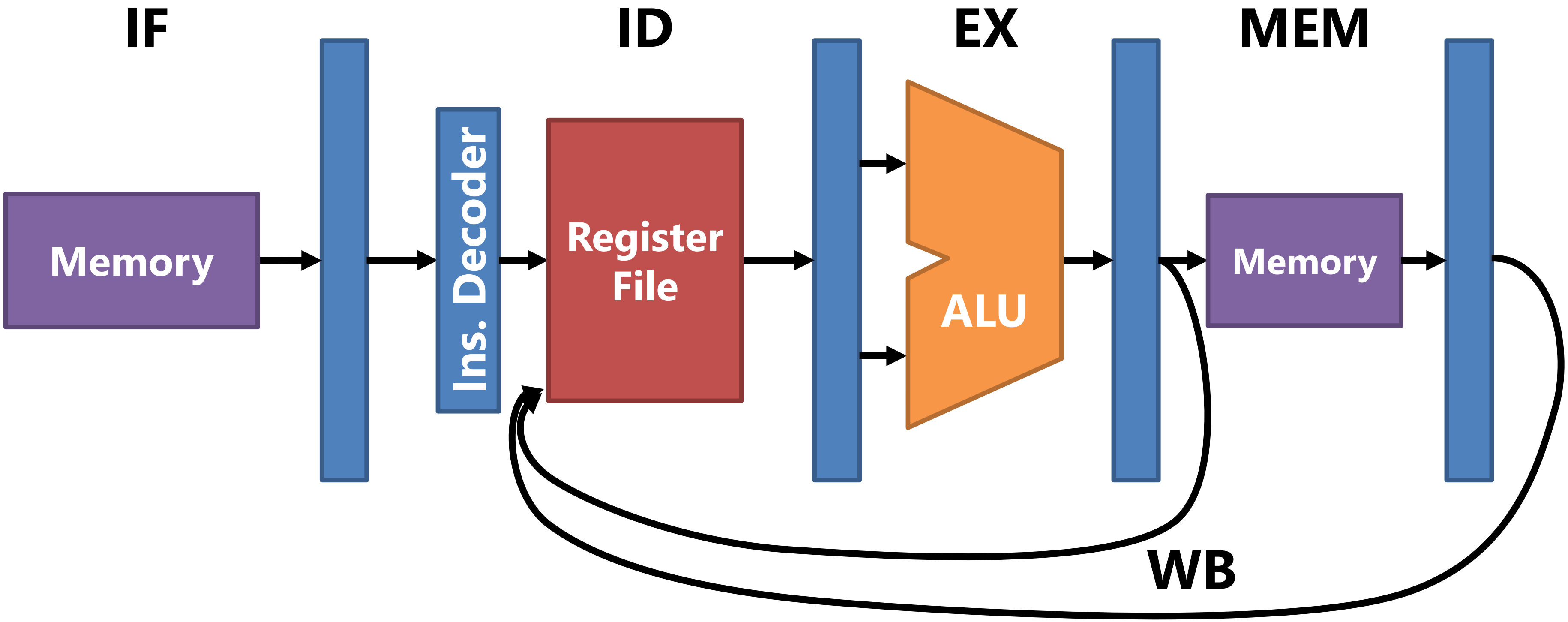
- If each phase takes *1 ns* each, cycle time must be *5 ns*:
  - Because it needs to be able to handle **lw**, which takes *5 ns*
  - **add** also takes *5 ns* when it could have been done in *4 ns*

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

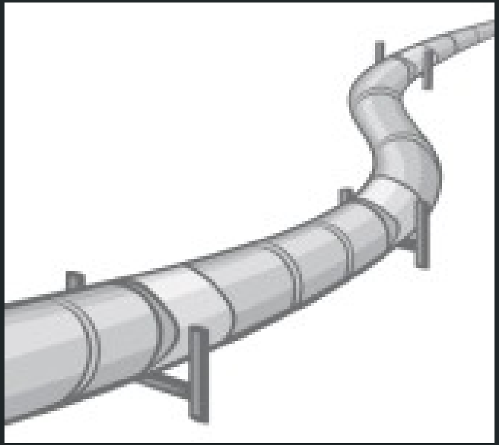
And we can do even better!

- 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!

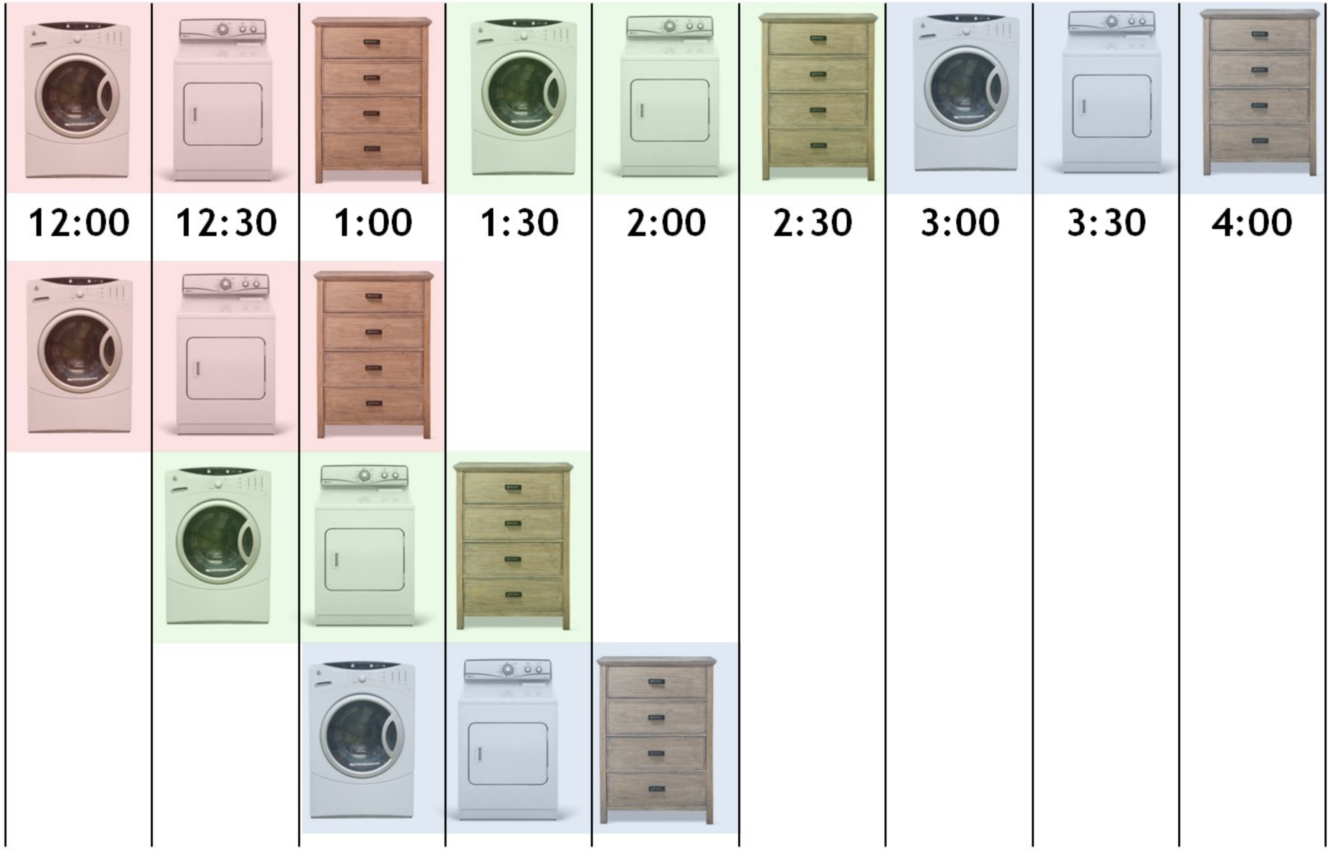
- Car factories create an assembly line to solve this problem
  - No need to wait until a car is finished before starting on next one
  - Our CPU is going to use a ***pipeline*** (similar concept)

Pipelining Basics



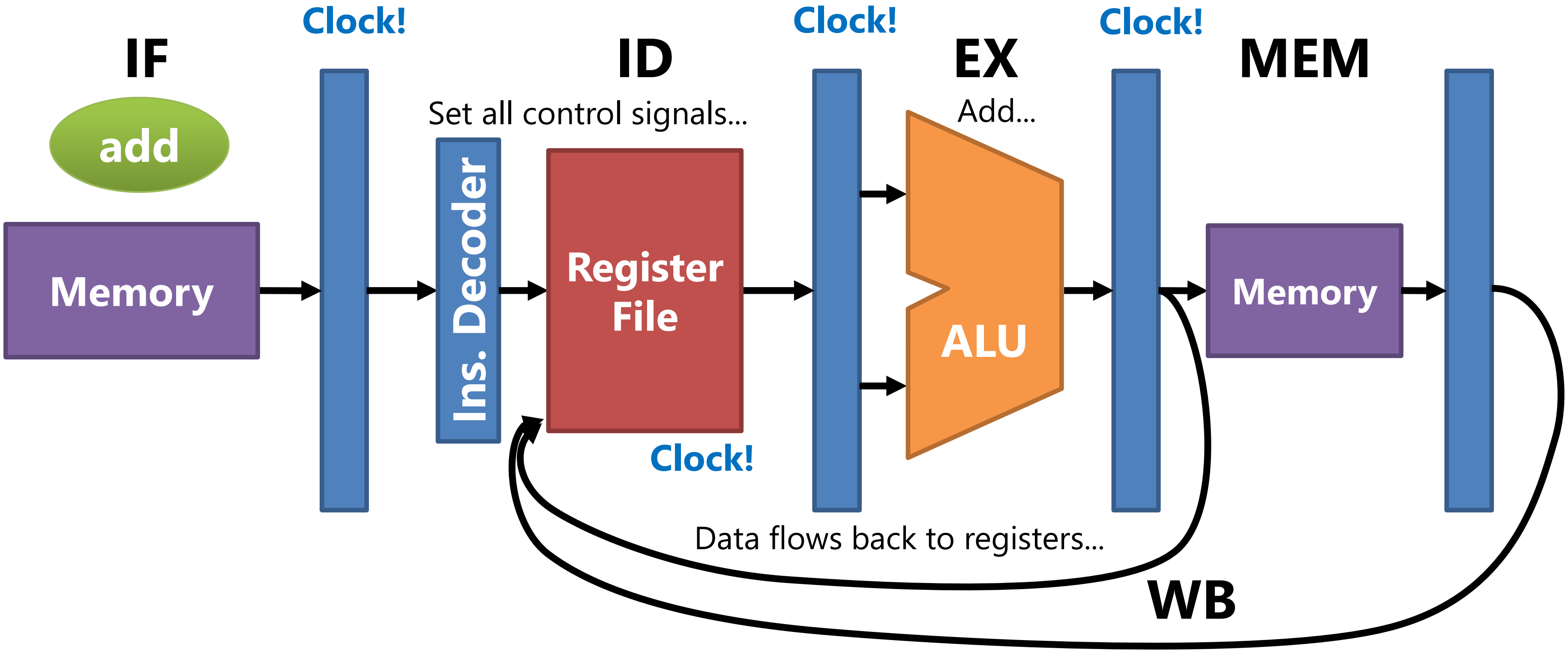
Improving Washer / Dryer / Closet Utilization

- 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!



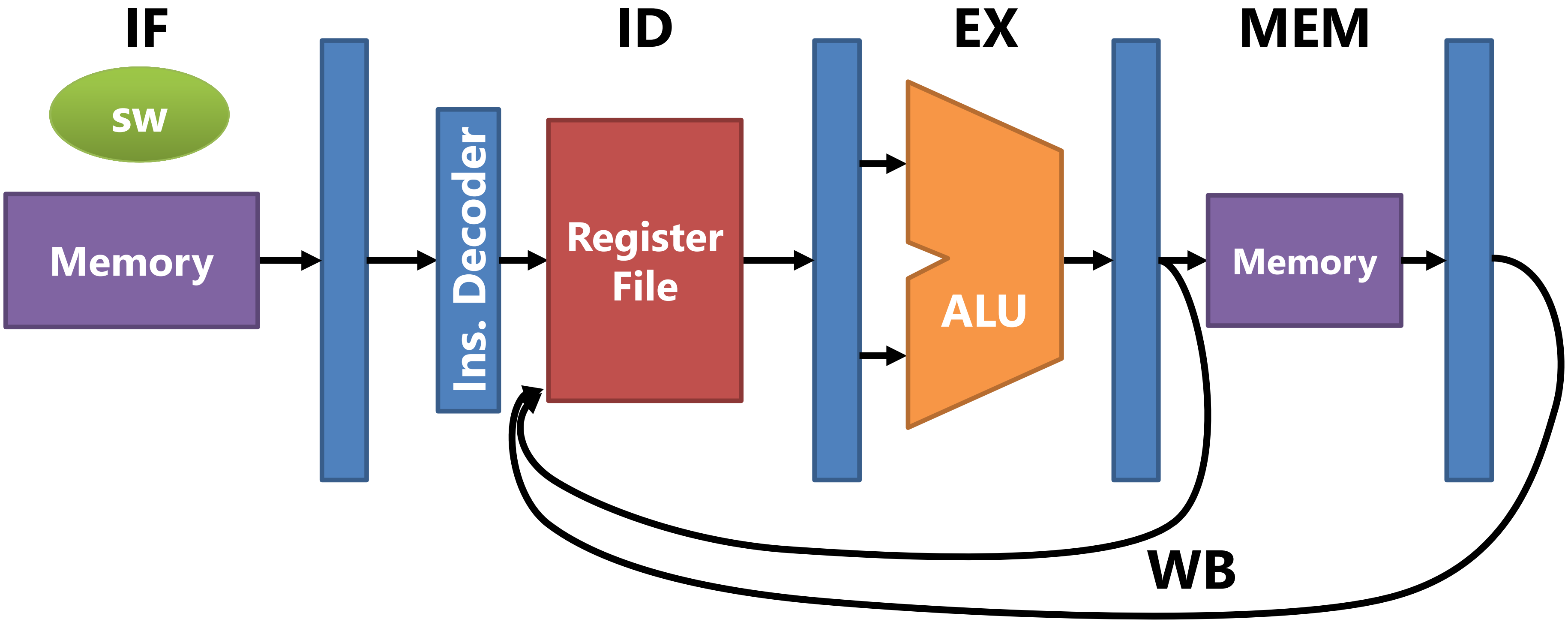
Multi-cycle instruction execution

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



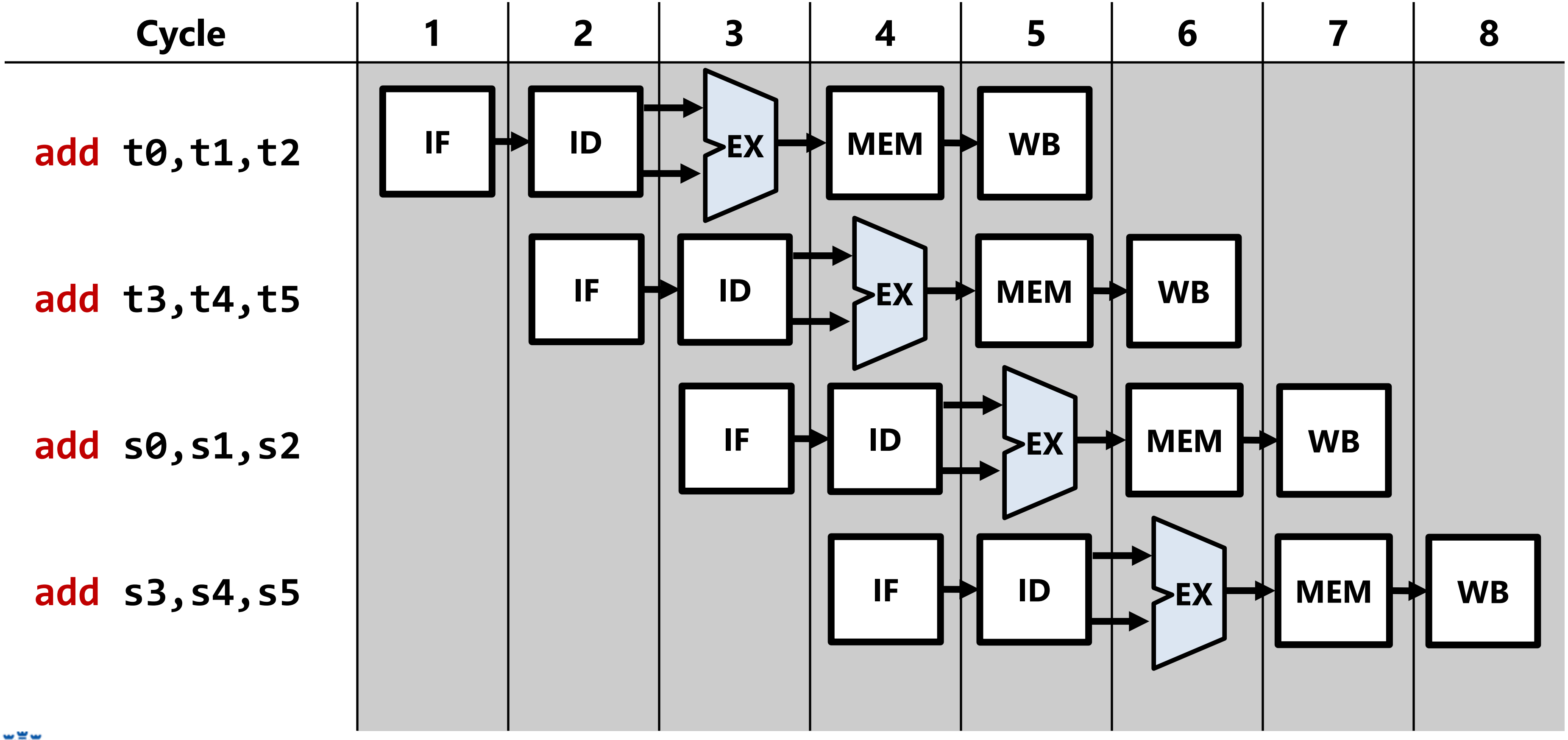
Pipelined instruction execution

- Pipelining allows one instruction to be fetched each cycle!



Pipelining Timeline

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



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

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 ns* + *5 ns*) / 100 = *1.04 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?

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

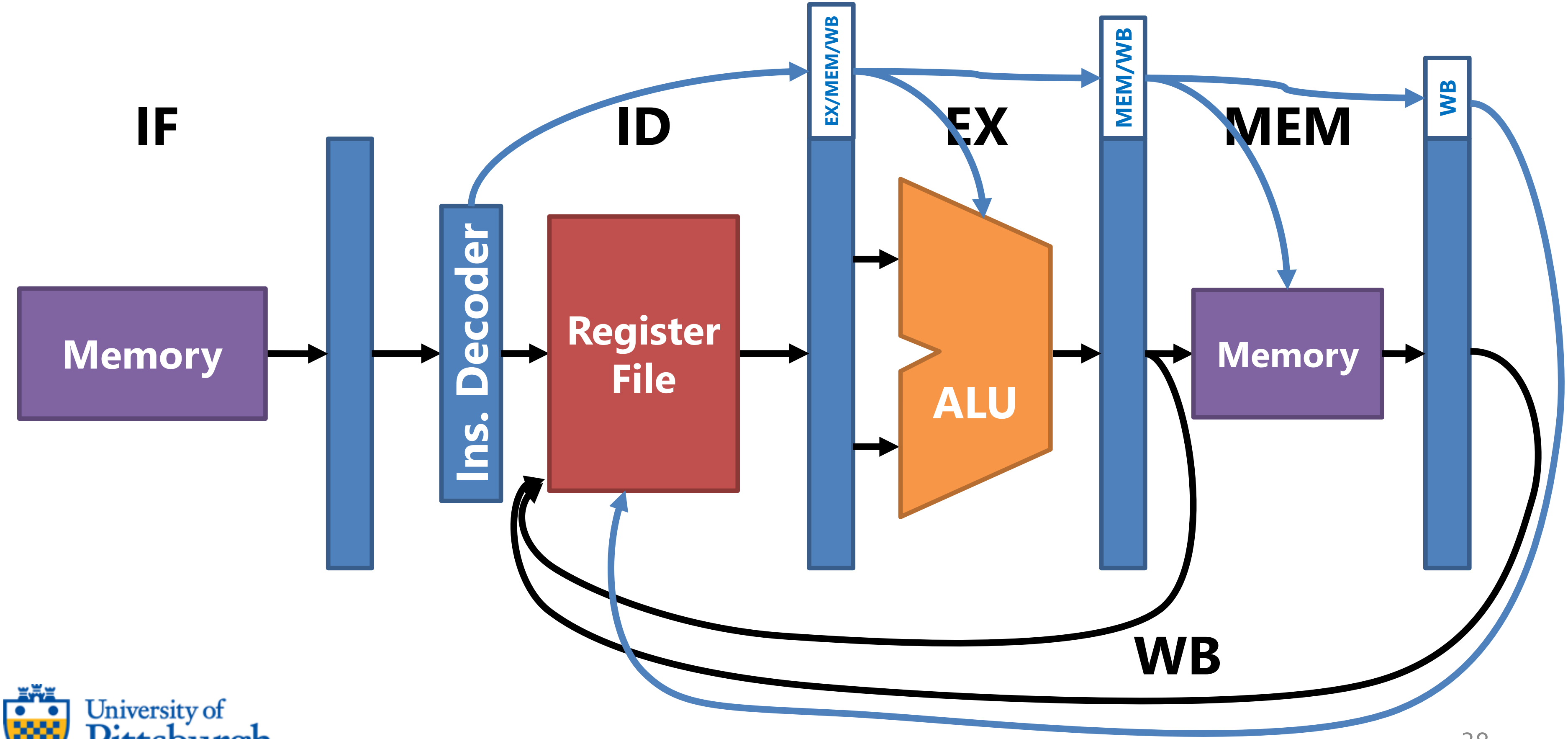
Architecture	Instructions	CPI	Cycle Time (1/F)
Single-cycle	Same	1	5 ns
Multi-cycle	Same	4~5	1 ns
Pipelined	Same	1	1 ns

- 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

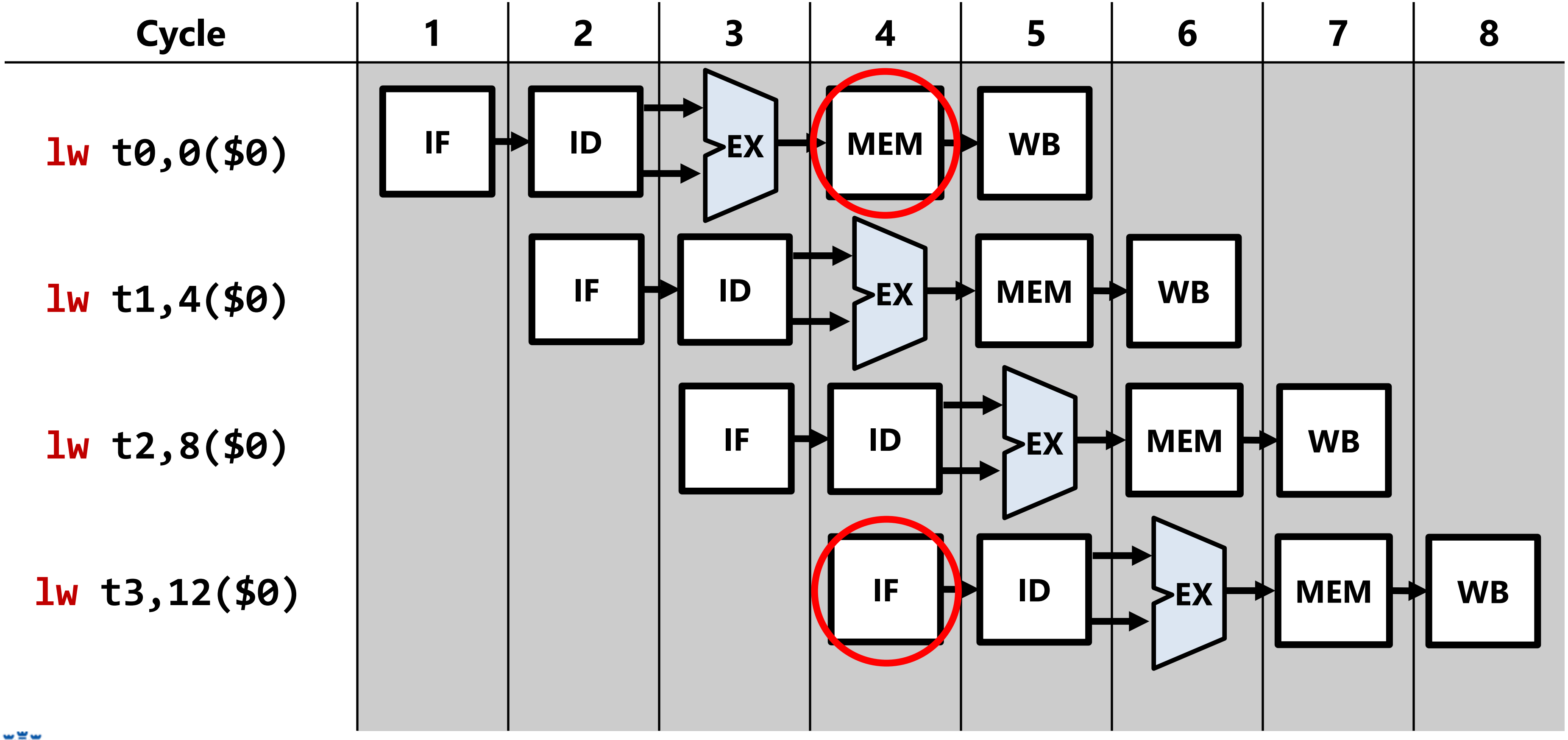
- ***Pipeline Hazard***
  - When the next instruction cannot execute in the following cycle
  - Hazards introduce ***bubbles*** (delays) into the pipeline timeline

- 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

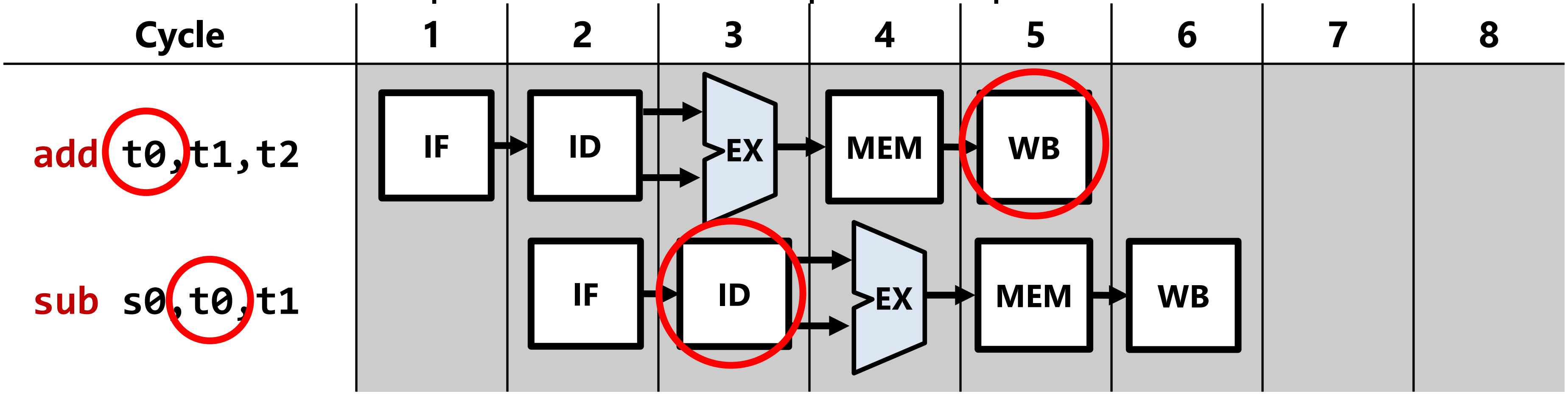
- Two instructions need to use the same hardware at the same time.



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

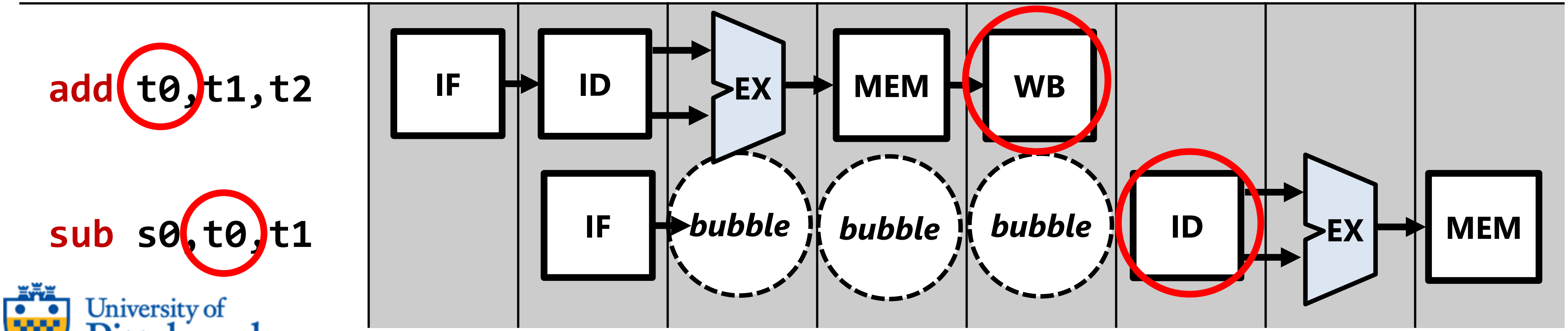
Data Hazards

- 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		

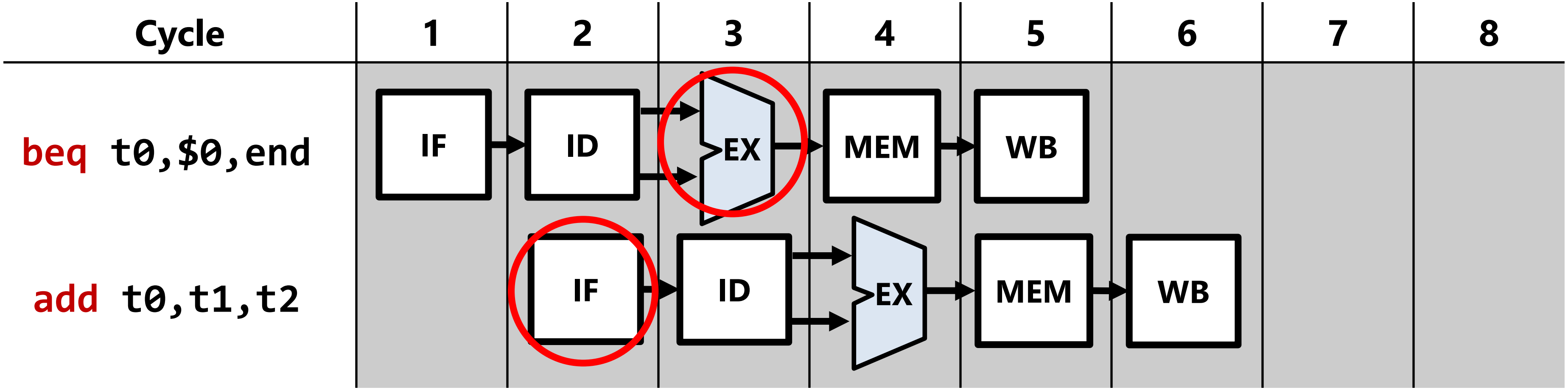
- **sub** must wait until **add**'s WB phase is over before doing its ID phase



add t0,t1,t2	IF	ID	EX	MEM	WB			
sub s0,t0,t1		IF	*bubble*	*bubble*	*bubble*	ID	EX	MEM

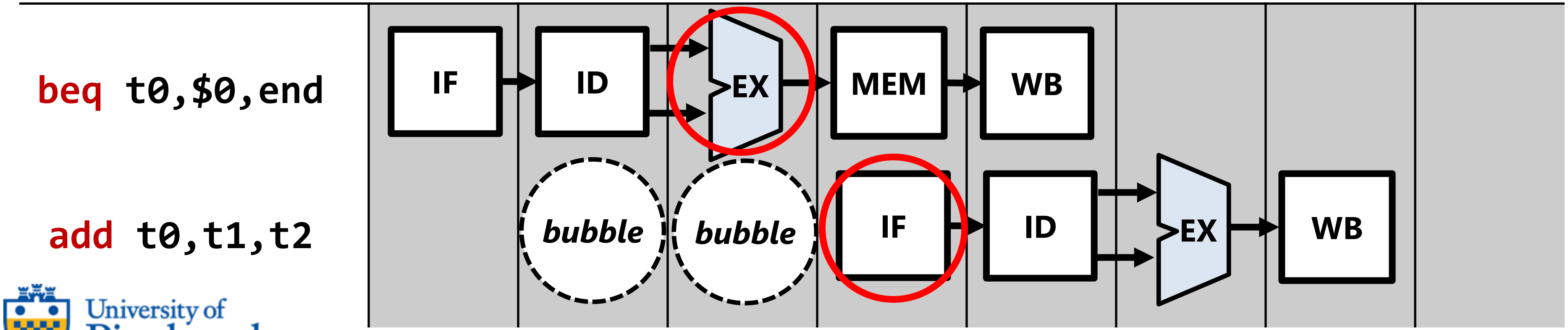
Control Hazards

- 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	

Dealing with Hazards

- Pipeline must be controlled so that hazards don’t cause malfunction

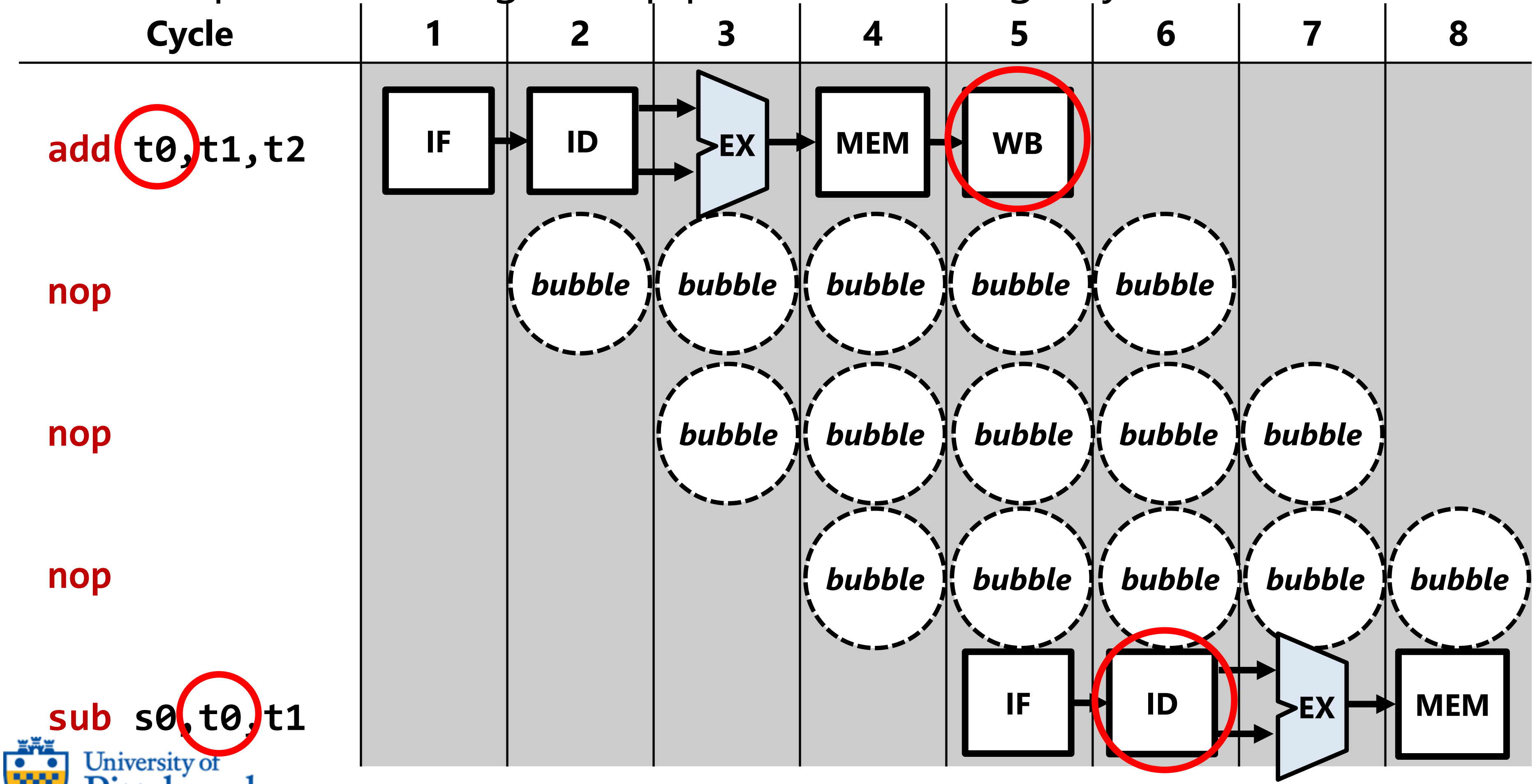
- Who is in charge of that? You have a choice.

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

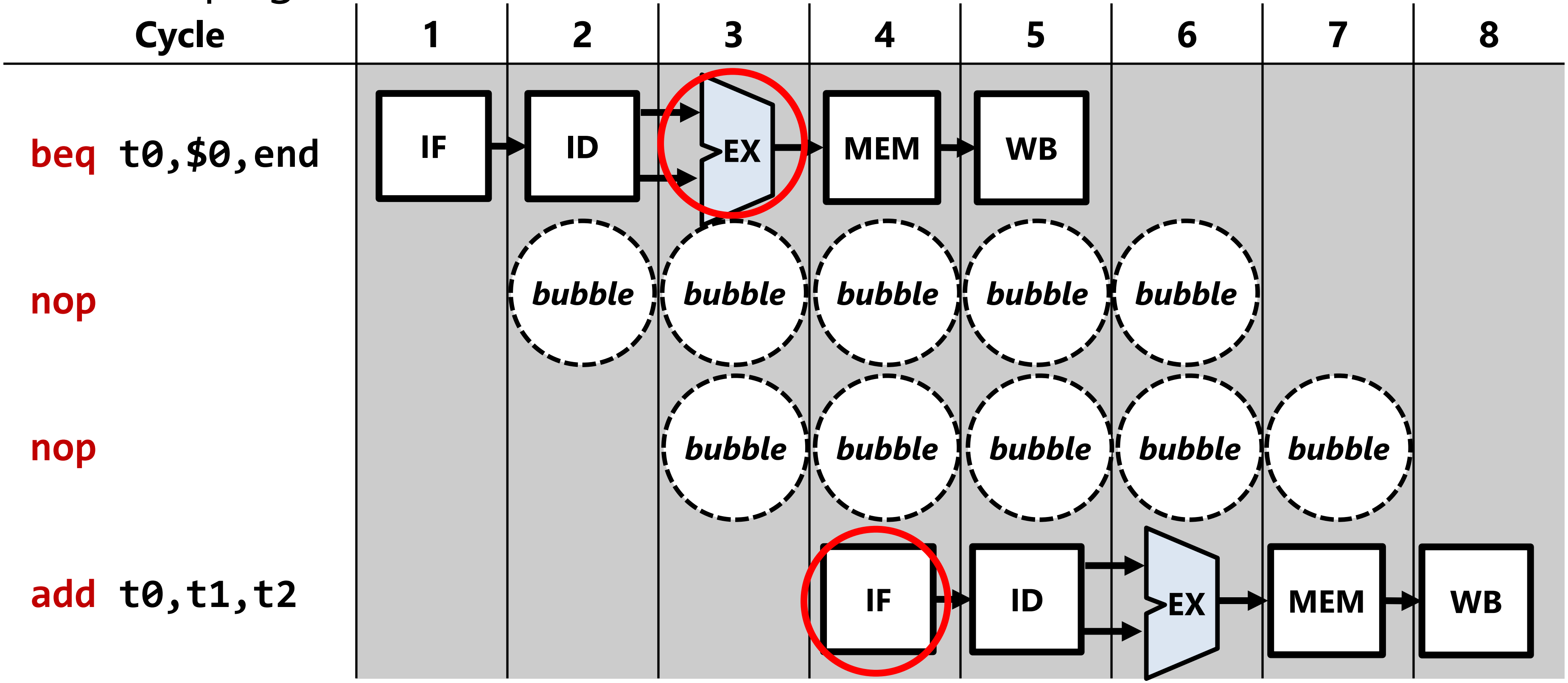
- The nops flow through the pipeline not doing any work



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

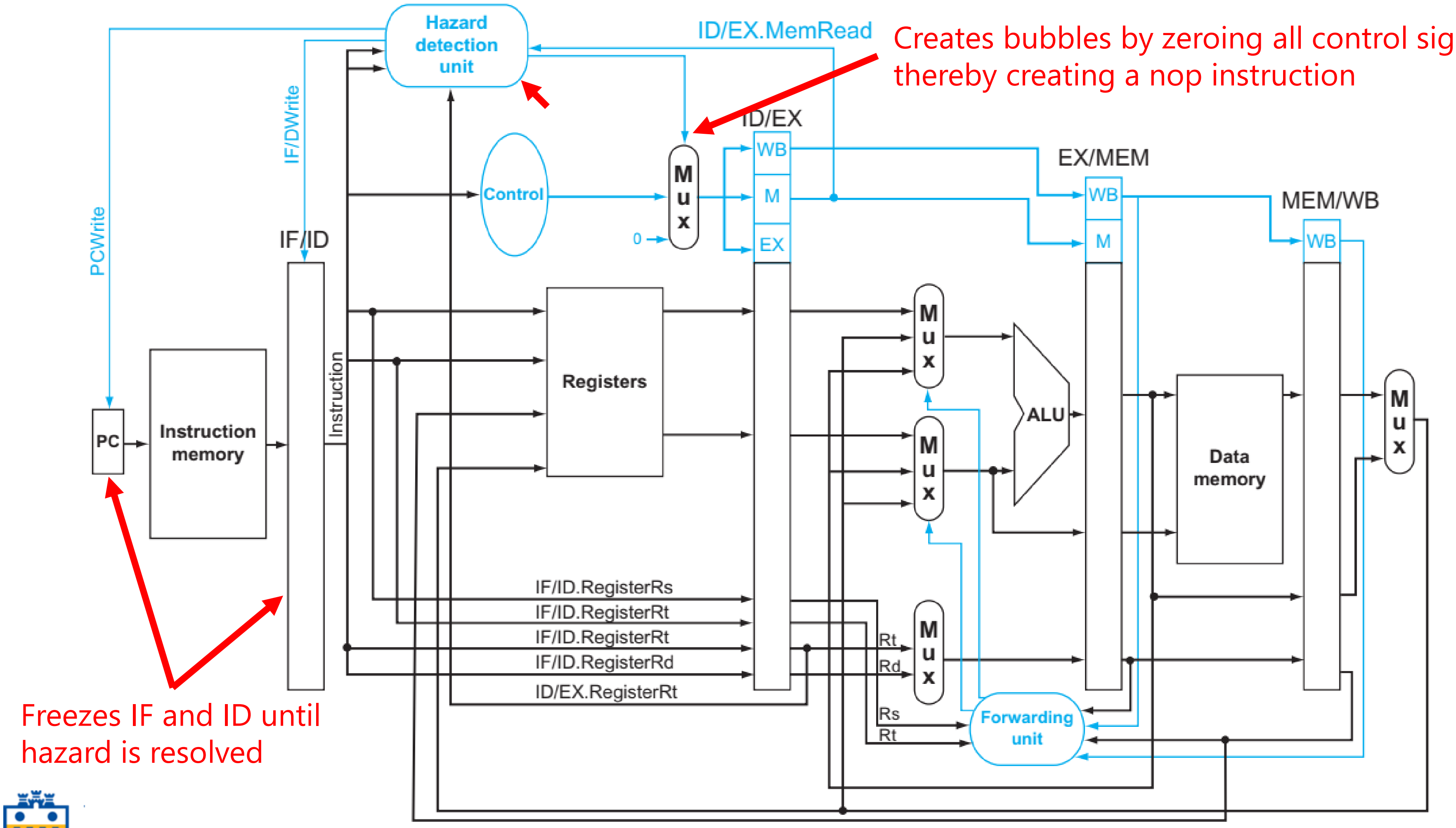
Compiler avoiding a control hazard

- 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

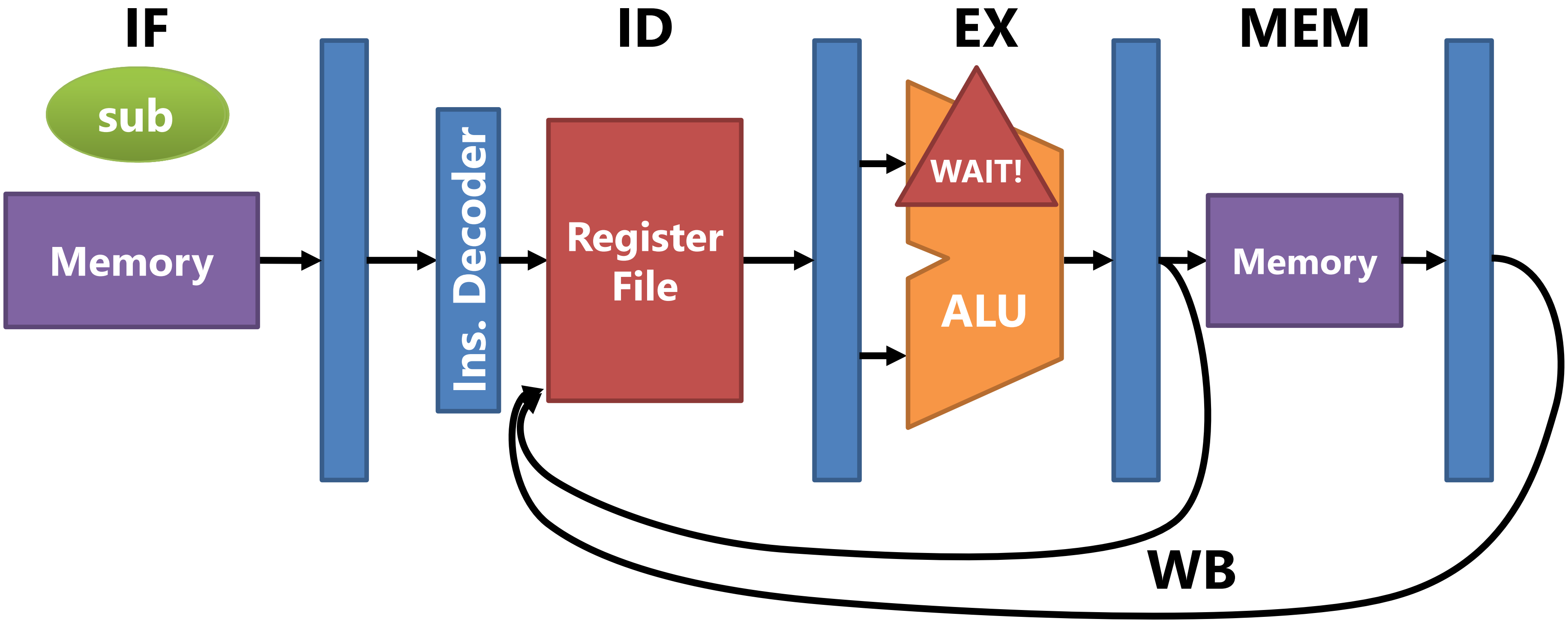


Creates bubbles by zeroing all control signals, thereby creating a nop instruction

Freezes IF and ID until hazard is resolved

Hazard Detection Unit avoiding a data hazard

- Suppose we have an **add** that depends on an **lw**.



Structural / Data Hazards cause stalls

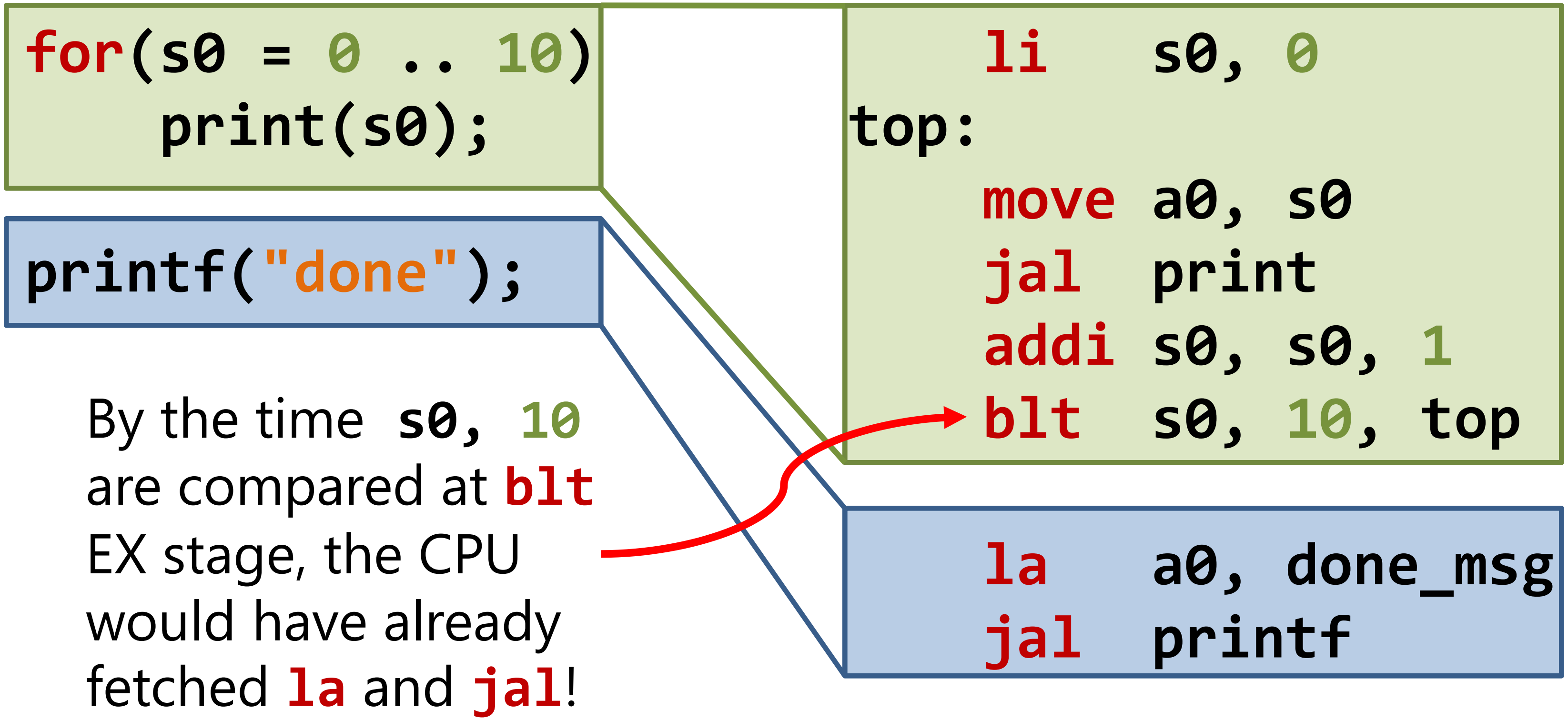
- If HDU detects a structural or data hazard, it does the following:
  - It **stops fetching instructions** (doesn't update the PC).
  - It **stops clocking the pipeline registers for the stalled stages.**
  - The stages after the stalled instructions **are filled with nops.**
    - Change control signals 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");
```

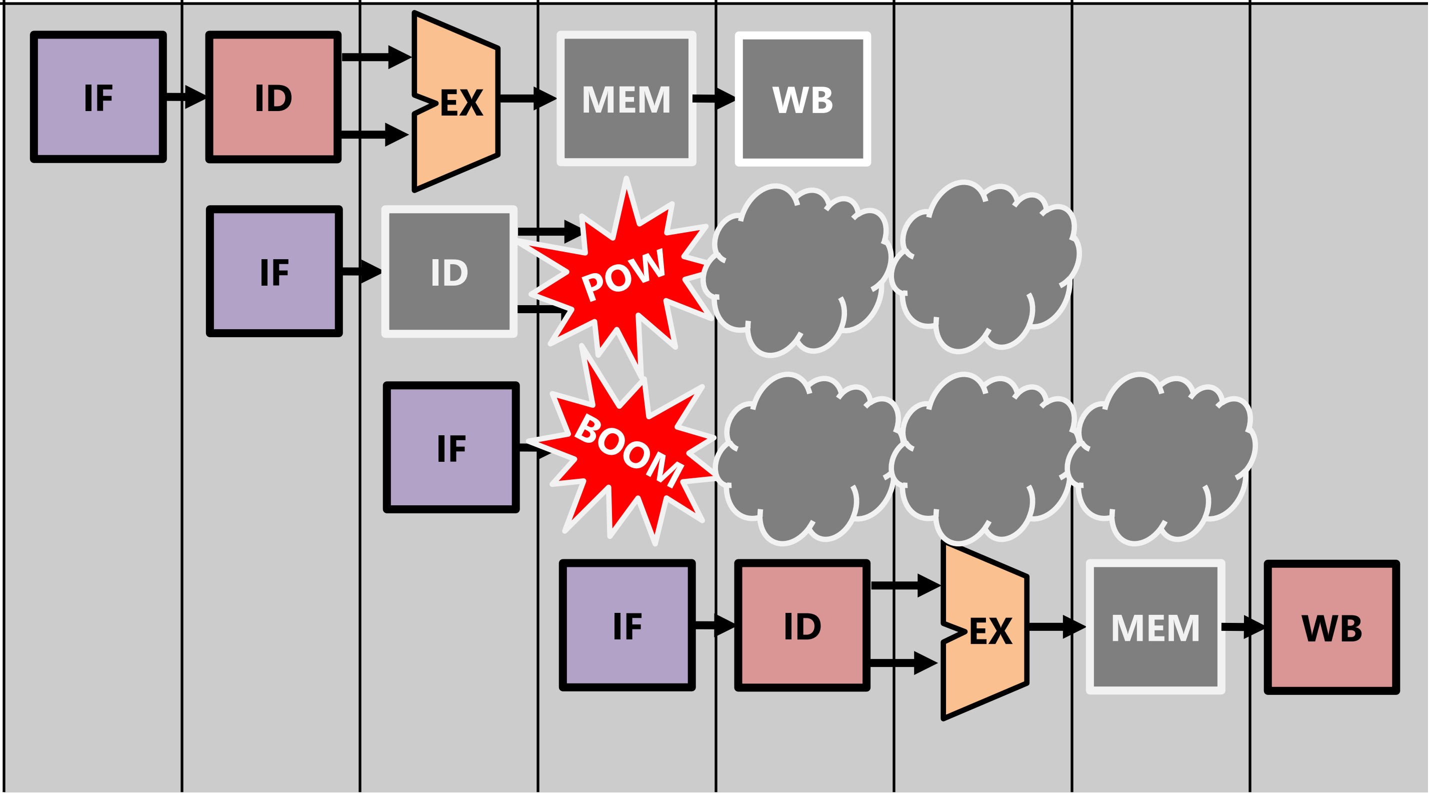
```
      li    s0, 0
top:
      move  a0, s0
      jal   print
      addi  s0, s0, 1
      blt   s0, 10, top
```

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

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

What's a flush?

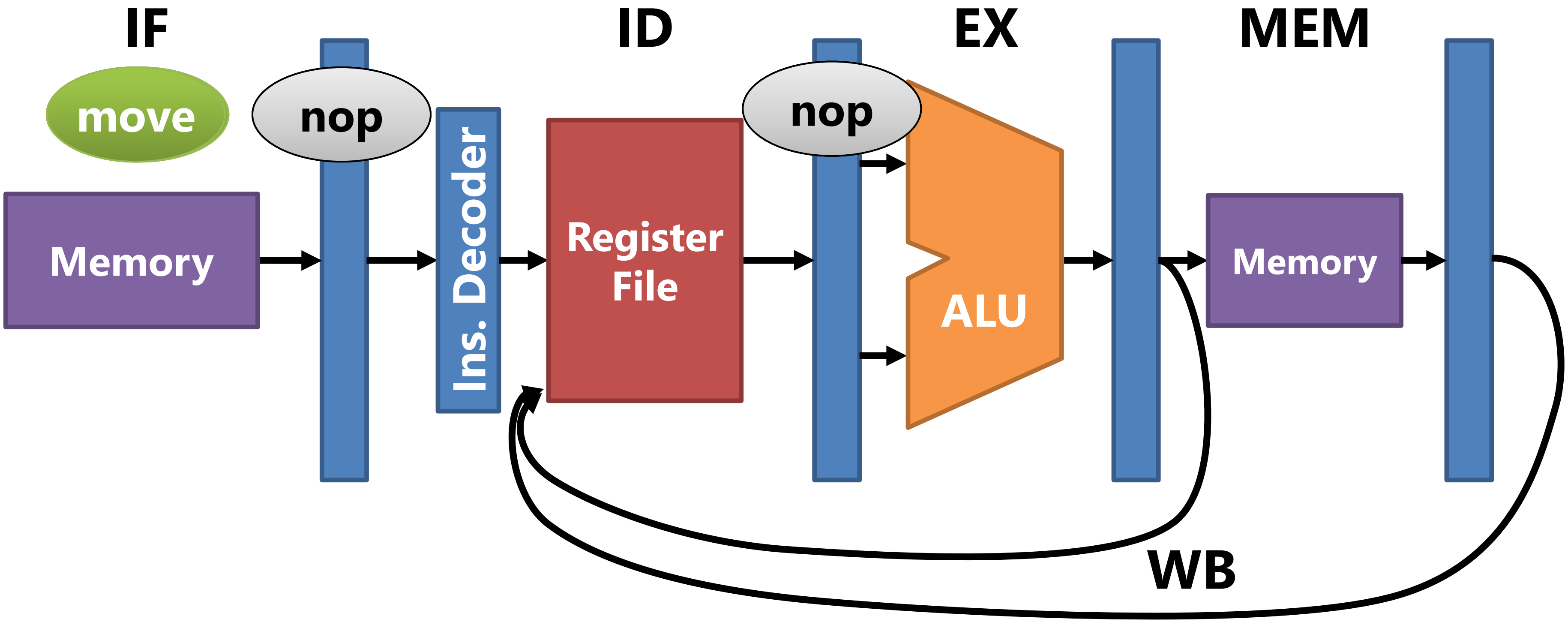
- A pipeline flush removes all wrong path instructions from pipeline



Time	0	1	2	3	4	5	6	7
blt s0,10,top	IF	ID	EX	MEM	WB			
la a0,done_msg		IF	ID	POW				
jal printf			IF	BOOM				
s0 < 10... OOPS! move a0,s0				IF	ID	EX	MEM	WB

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!

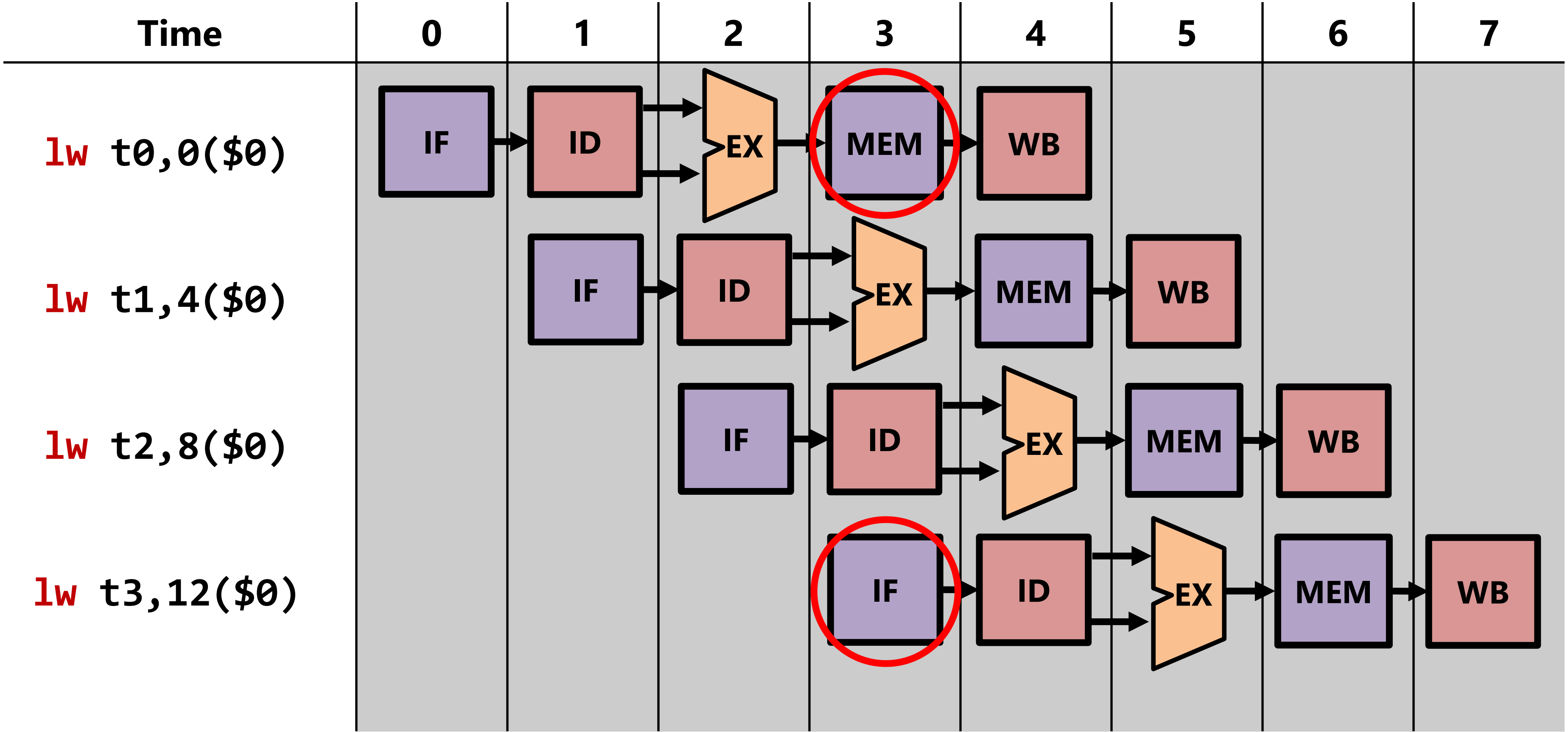
Compiler nops vs. CPU Hazard Detection Unit

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

Solving Structural Hazards

Structural Hazard on Memory

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



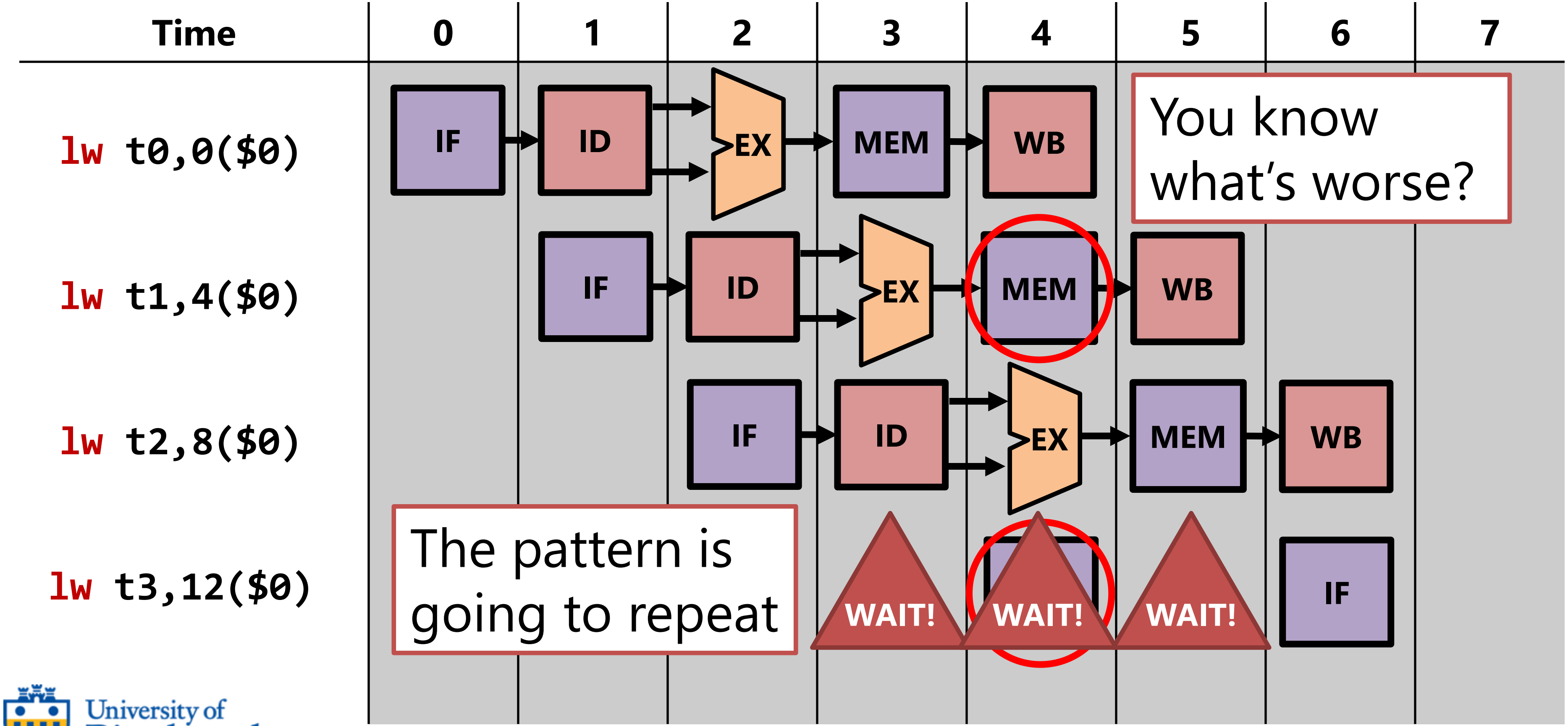
Time	0	1	2	3	4	5	6	7
`lw t0,0($0)`	IF	ID	EX	MEM	WB			
`lw t1,4($0)`		IF	ID	EX	MEM	WB		
`lw t2,8($0)`			IF	ID	EX	MEM	WB	
`lw t3,12($0)`				IF	ID	EX	MEM	WB

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**).



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?

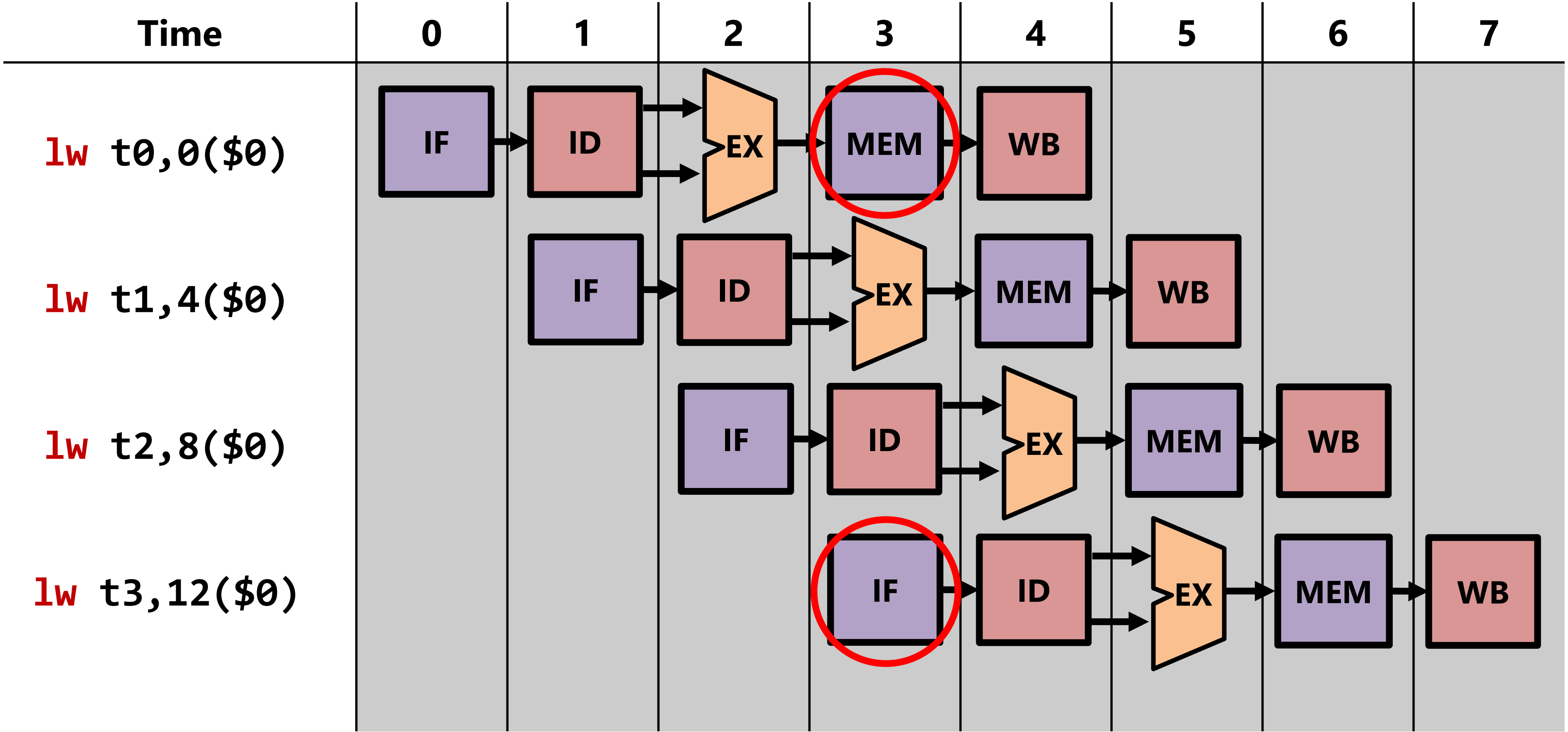
The pattern is going to repeat

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

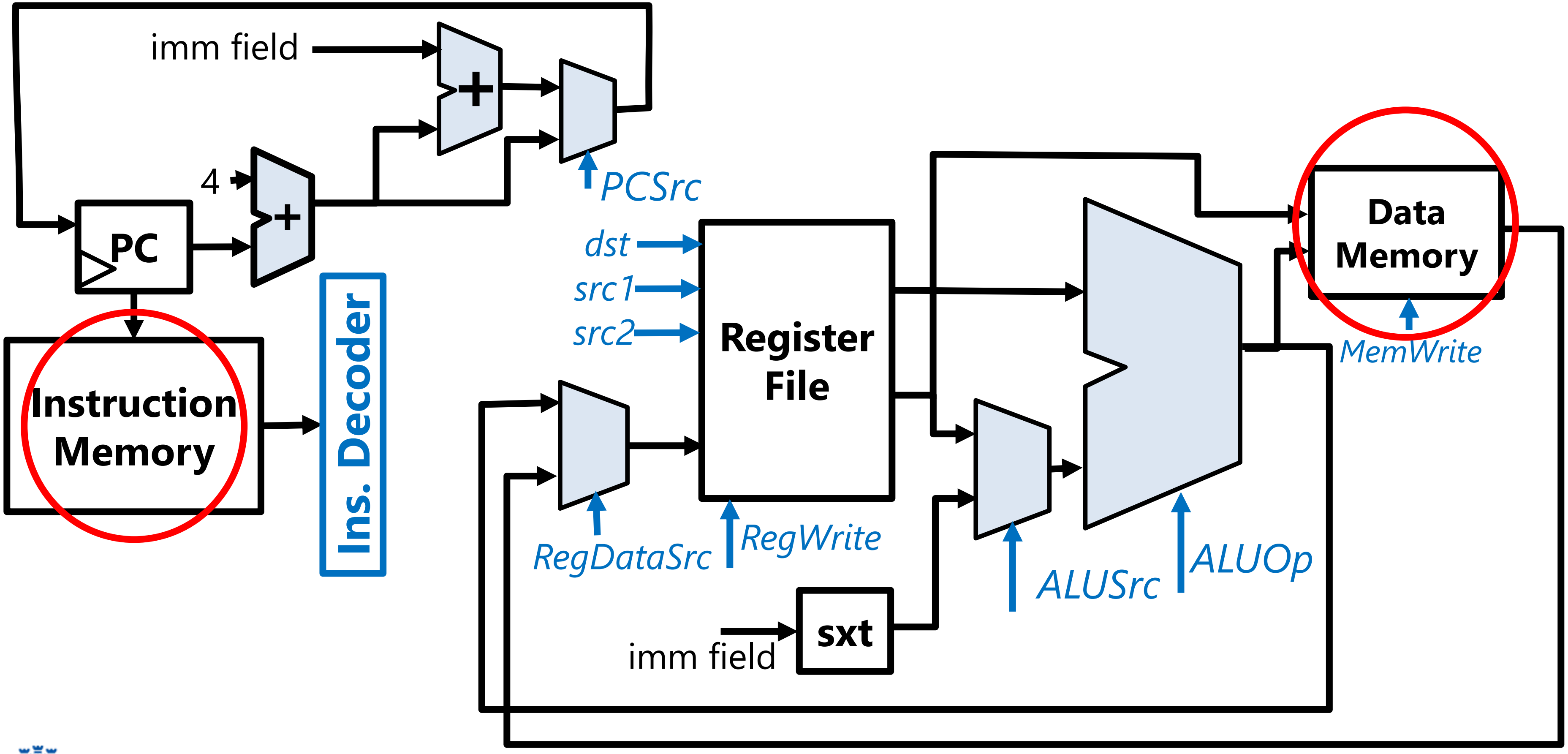
- With separate i-cache and d-cache, MEM and IF can work in parallel



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

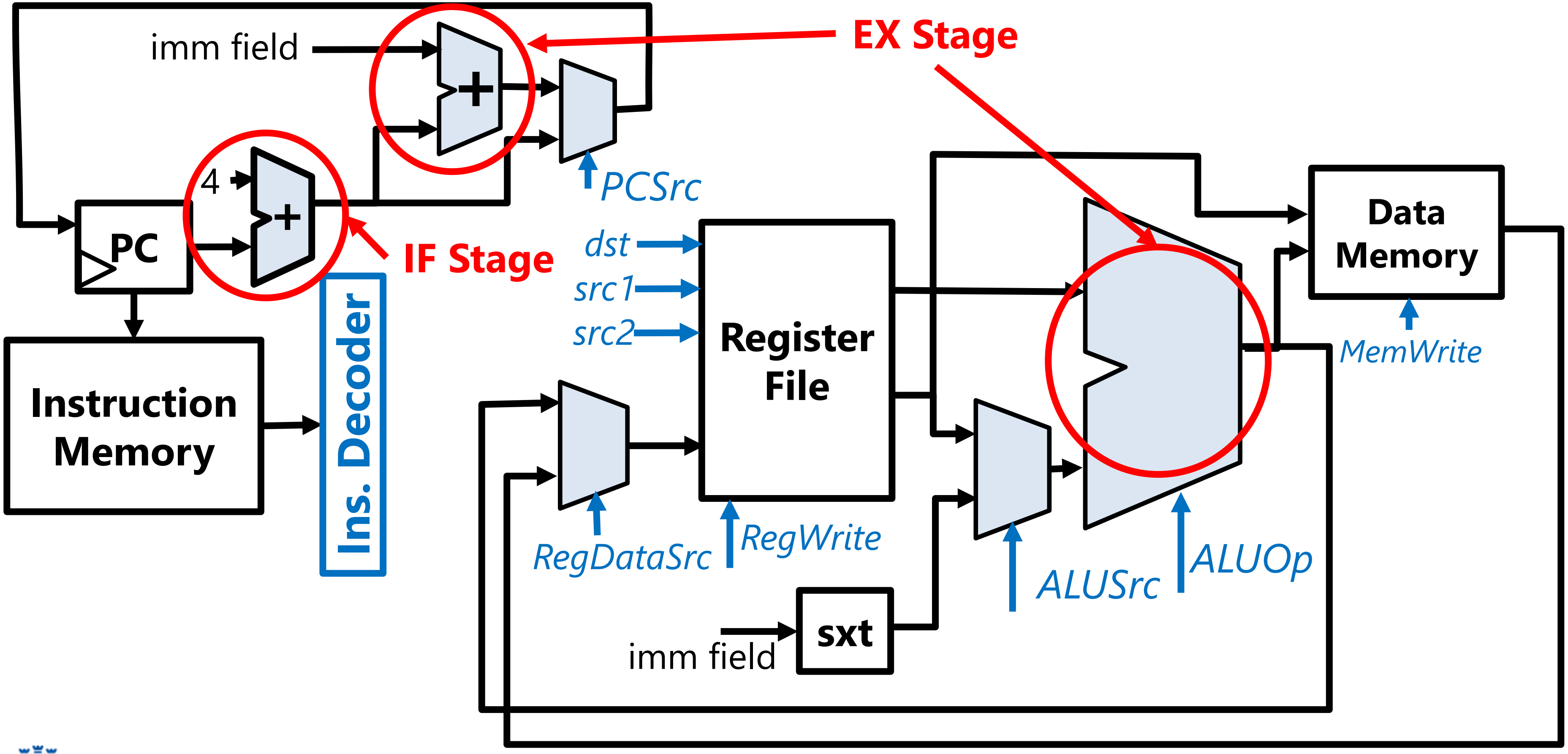
Structural Hazard removed with two Memories

- 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

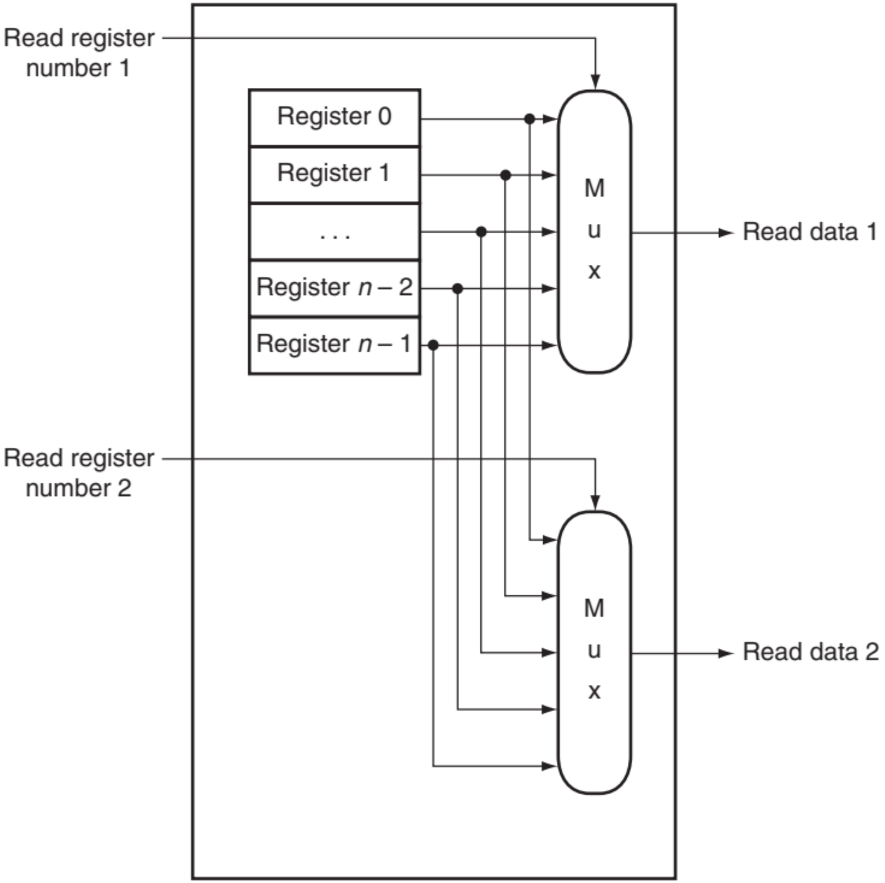
- There are mainly two ways to throw more hardware at the problem

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

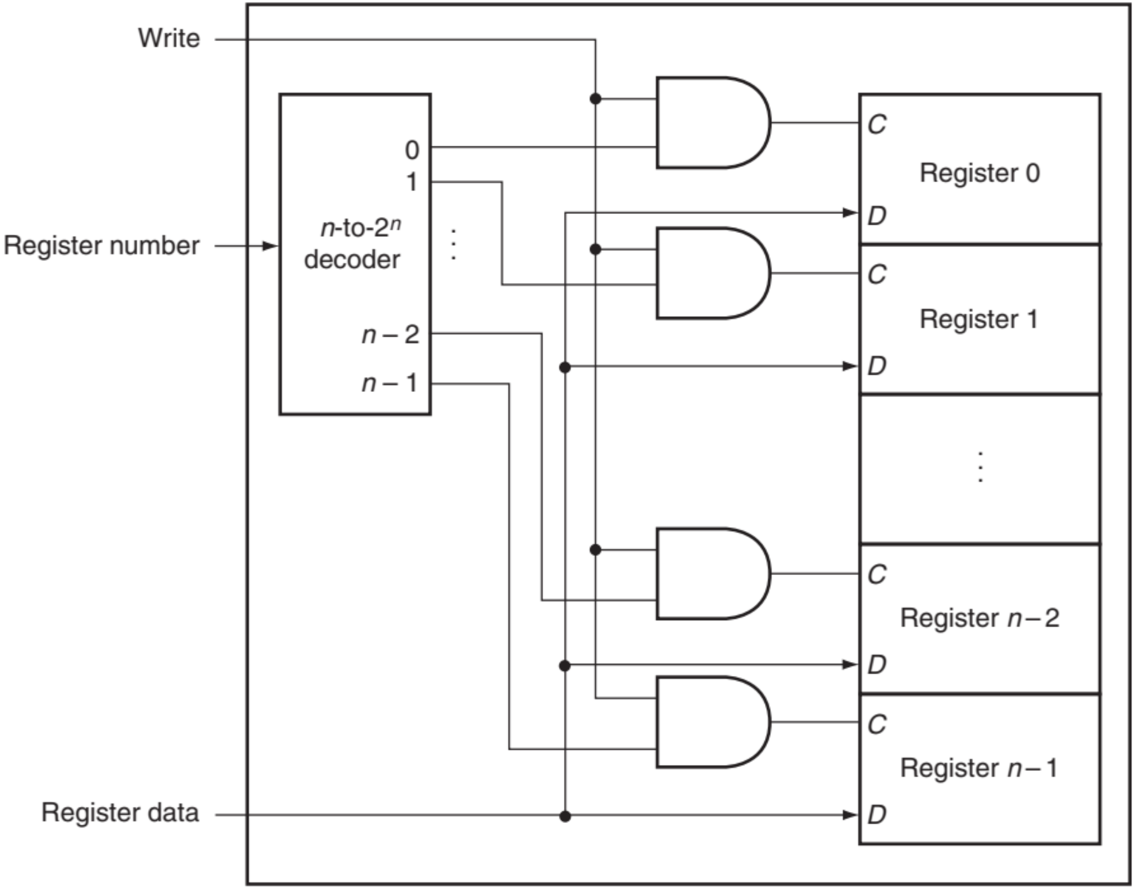
Two Register Read Ports

- 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)

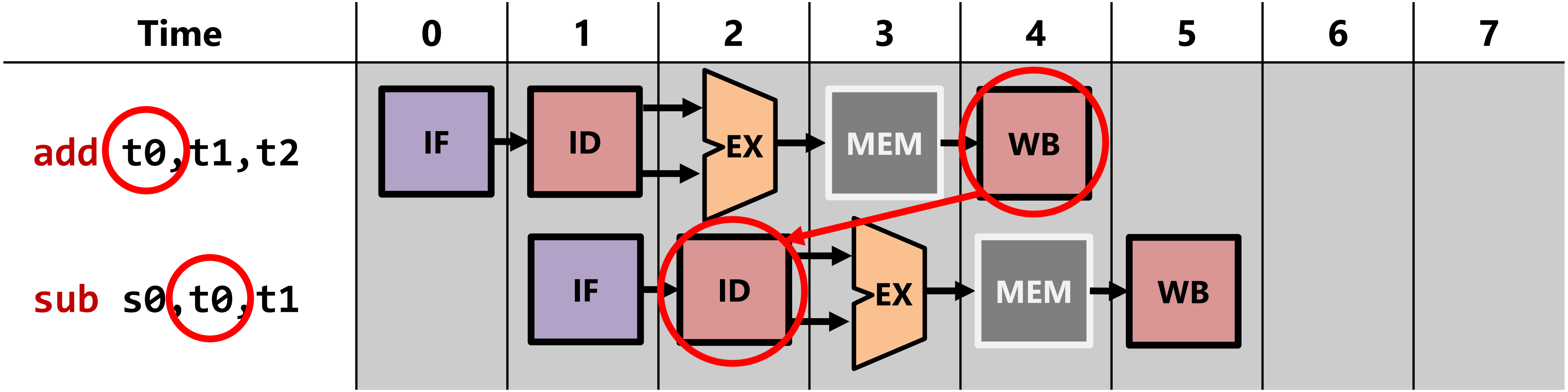
- But what if we want an IPC > 1?
  - More than one instruction per cycle! (a.k.a superscalar processor)
  - Must sustain more than one ID / WB stage per cycle
  - Need more register read ports and write ports!
  - Not only registers, memory would need more ports too!
  - Like everything else, this consumes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

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

Solving Data Hazards

Data Hazards

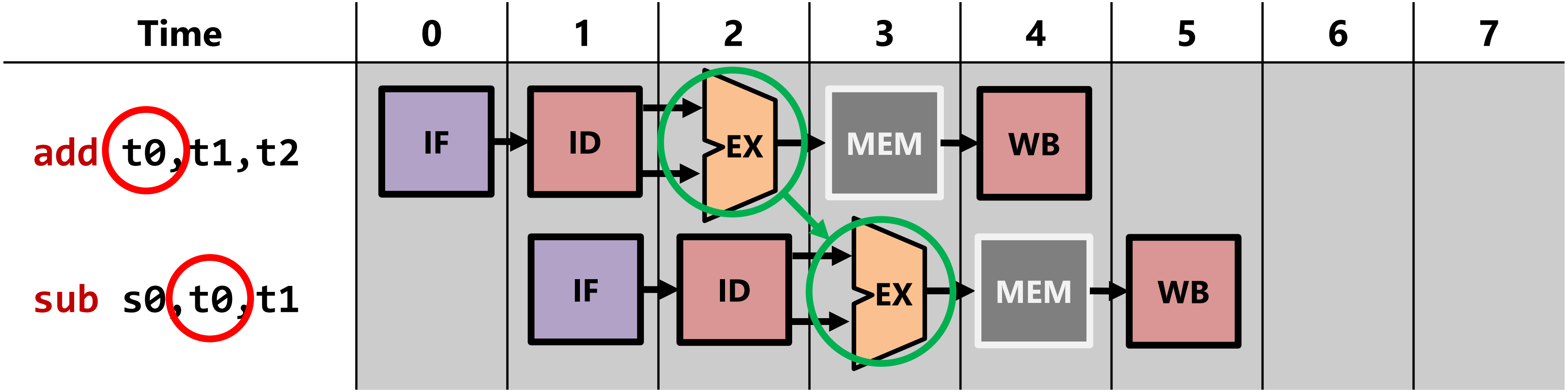
- An instruction depends on the output of a previous one.



- 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.

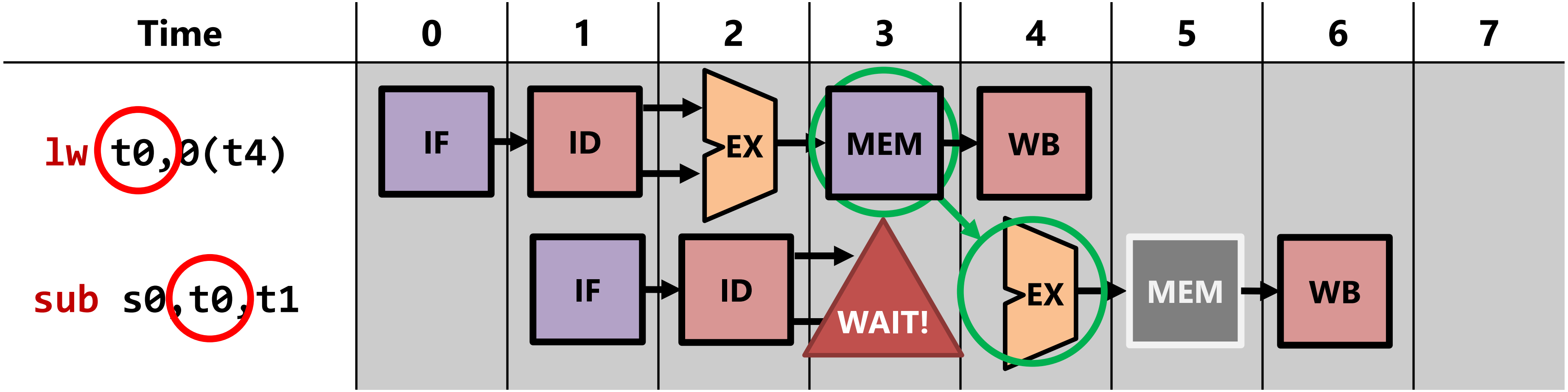


Time	0	1	2	3	4	5	6	7
add t0,t1,t2	IF	ID	EX	MEM	WB			
sub s0,t0,t1		IF	ID	EX	MEM	WB		

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

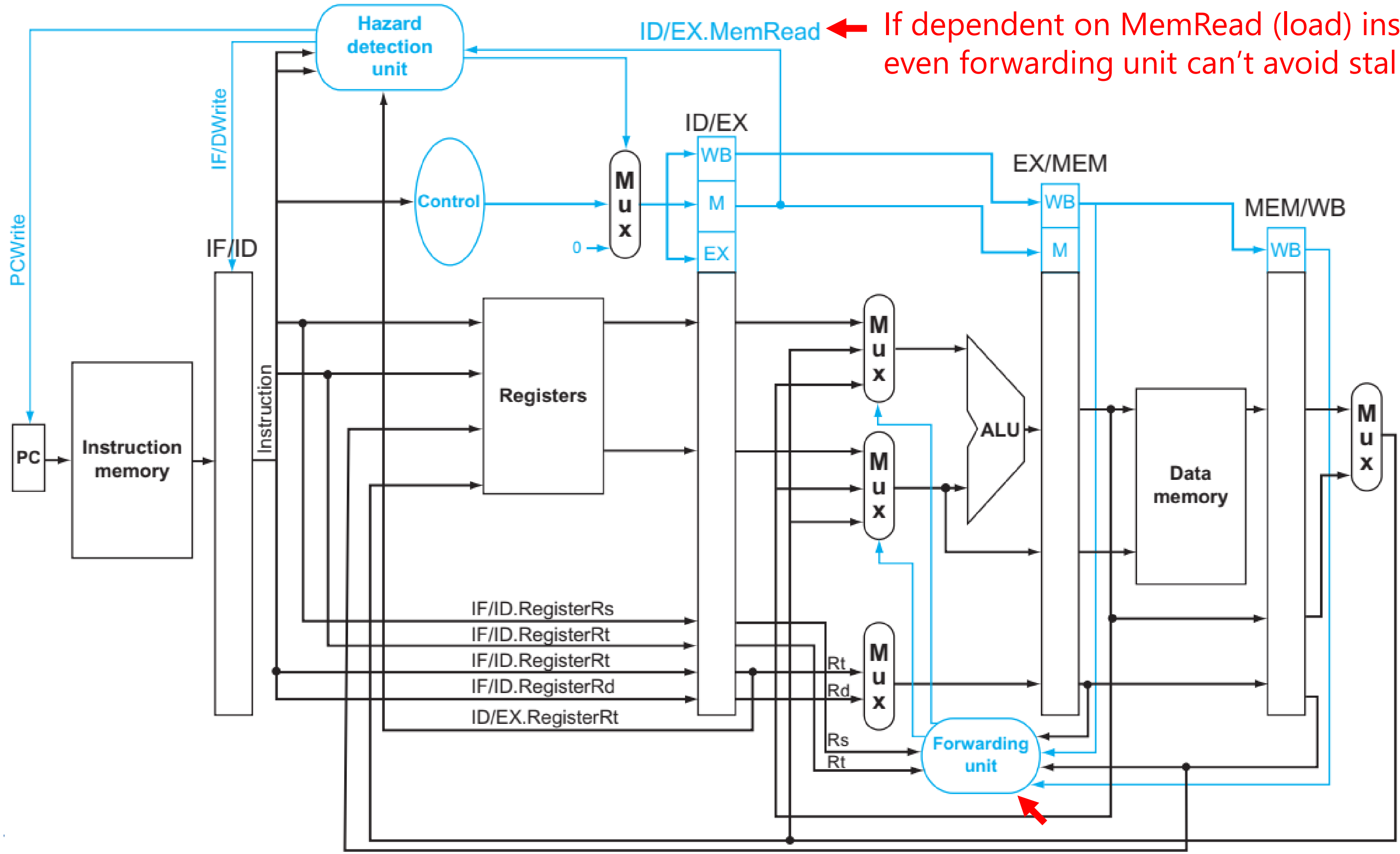
Data Forwarding from Memory

- 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



If dependent on MemRead (load) instruction, even forwarding unit can't avoid stall

Forwarding Unit

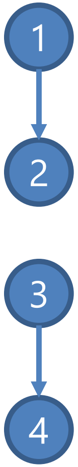
- Just like the HDU, the Forwarding Unit is **power** hungry

- Number of forwarding wires ∝ (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

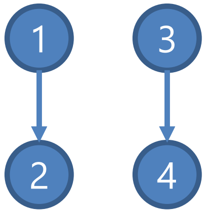
- **Deep pipelining** has **diminishing returns on power investment**
  - Cycle time improves by a factor of N
  - Power consumption increases by a factor of N^2 (or more)
  - Not the only problem with deep pipelining that we will see

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

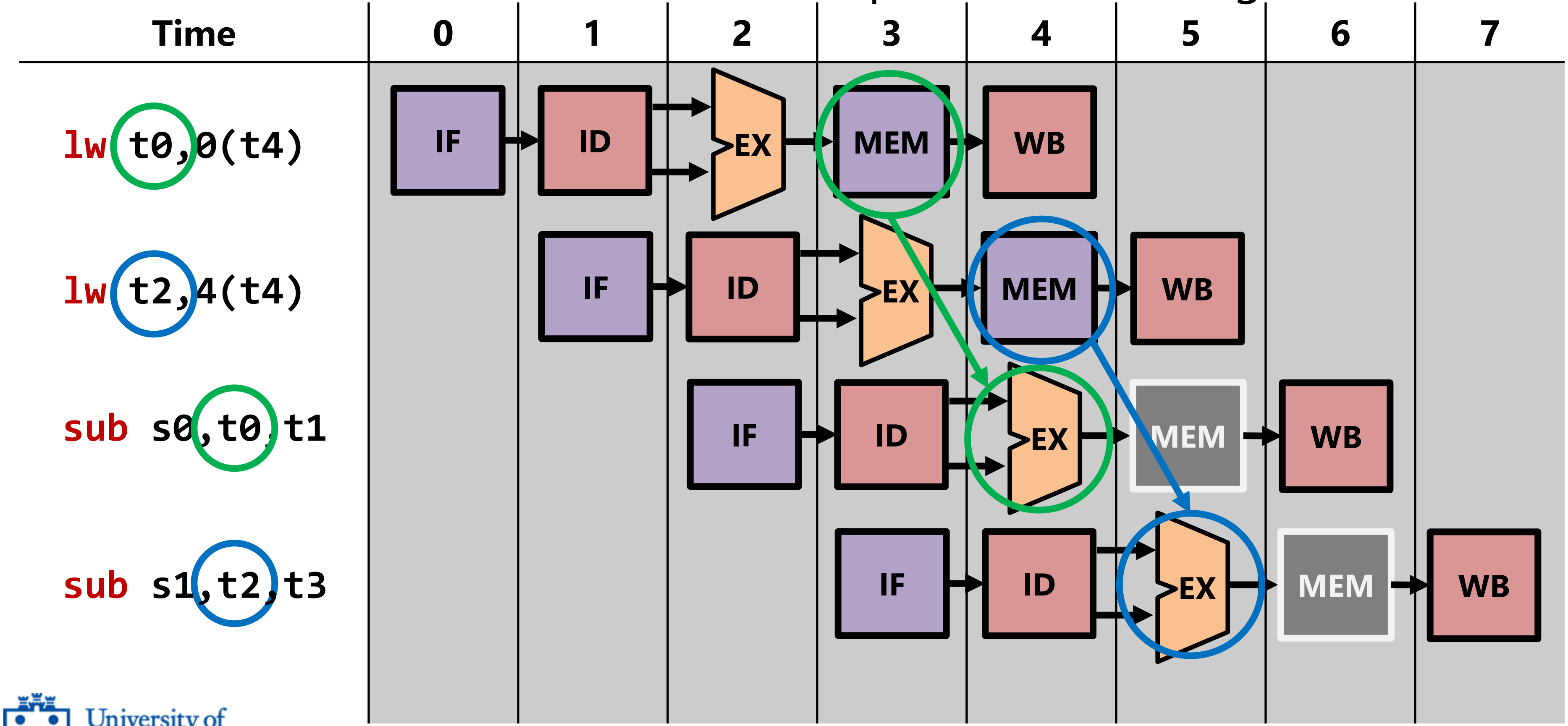
Data Hazard removed through Compiler Reordering

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

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`								

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)	IF	ID	EX	MEM	WB			
lw t2,4(t4)		IF	ID	EX	MEM	WB		
sub s0,t0,t1			IF	ID	EX	MEM	WB	
sub s1,t2,t3				IF	ID	EX	MEM	WB

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

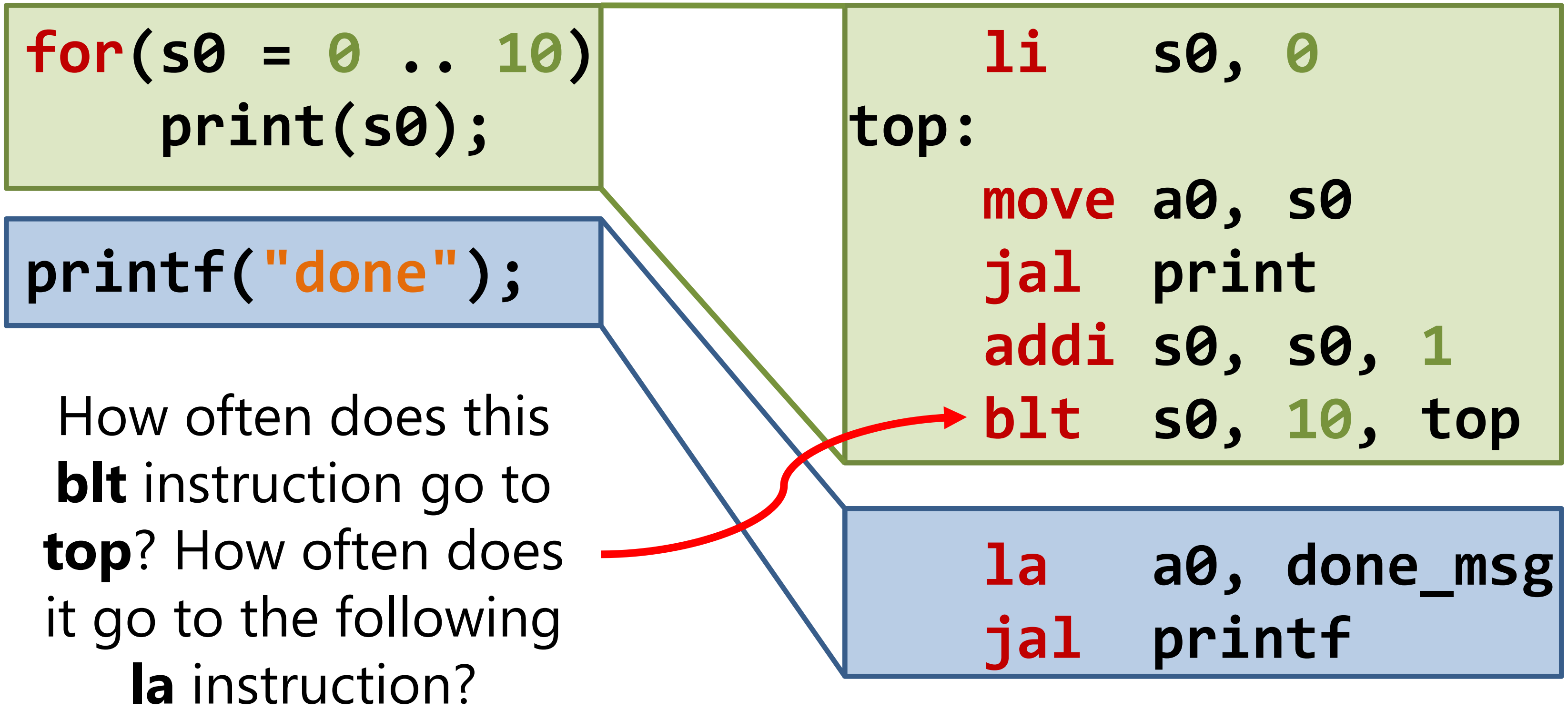
- It doesn't have the limitations of static scheduling
  - It doesn't have to predict memory latency
    - It can adapt as things unfold
  - It's easy to figure out data dependencies, even memory ones
    - At runtime, addresses of 8(t0) and 4(t4) are easily calculated

- 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");
```

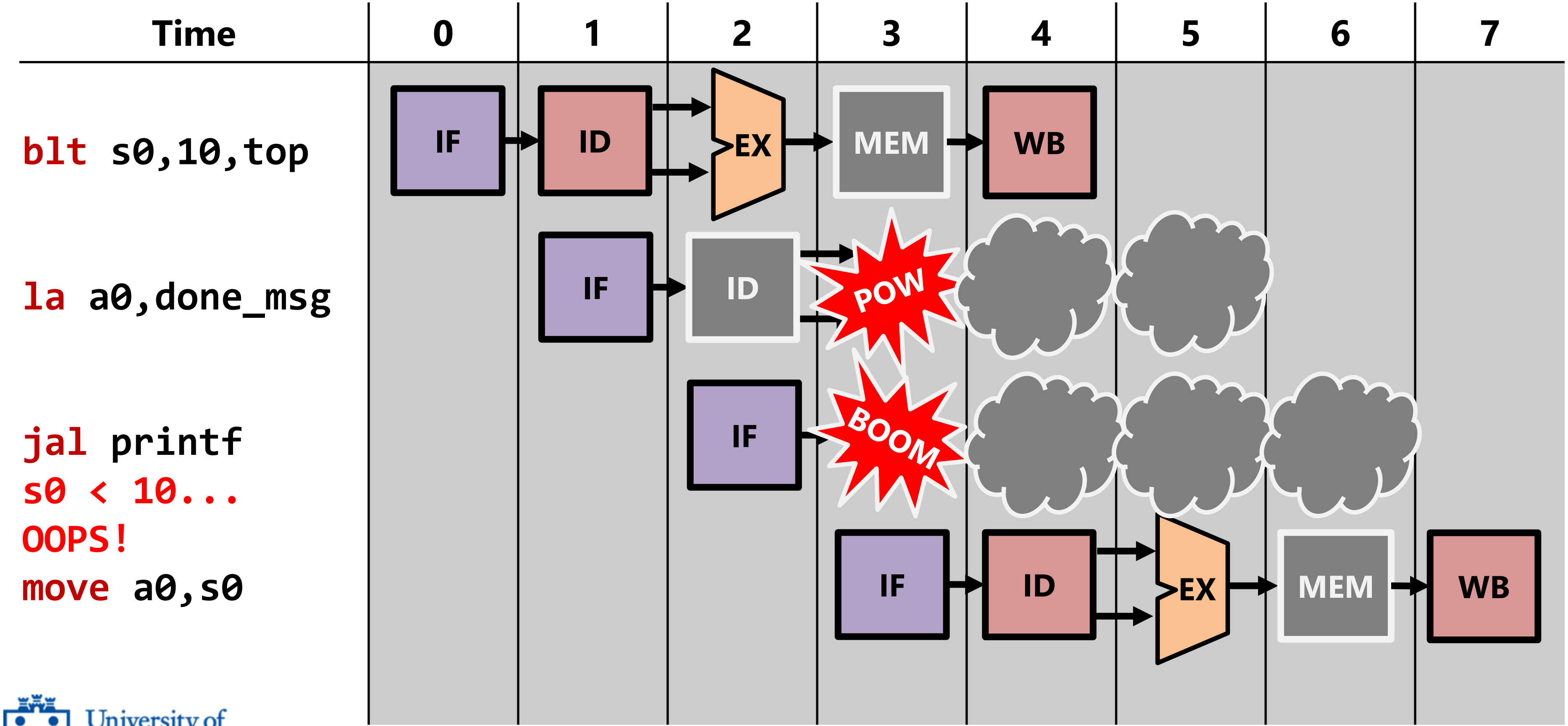
```
    li   s0, 0
top:
    move a0, s0
    jal  print
    addi s0, s0, 1
    blt  s0, 10, top
```

```
    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!

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

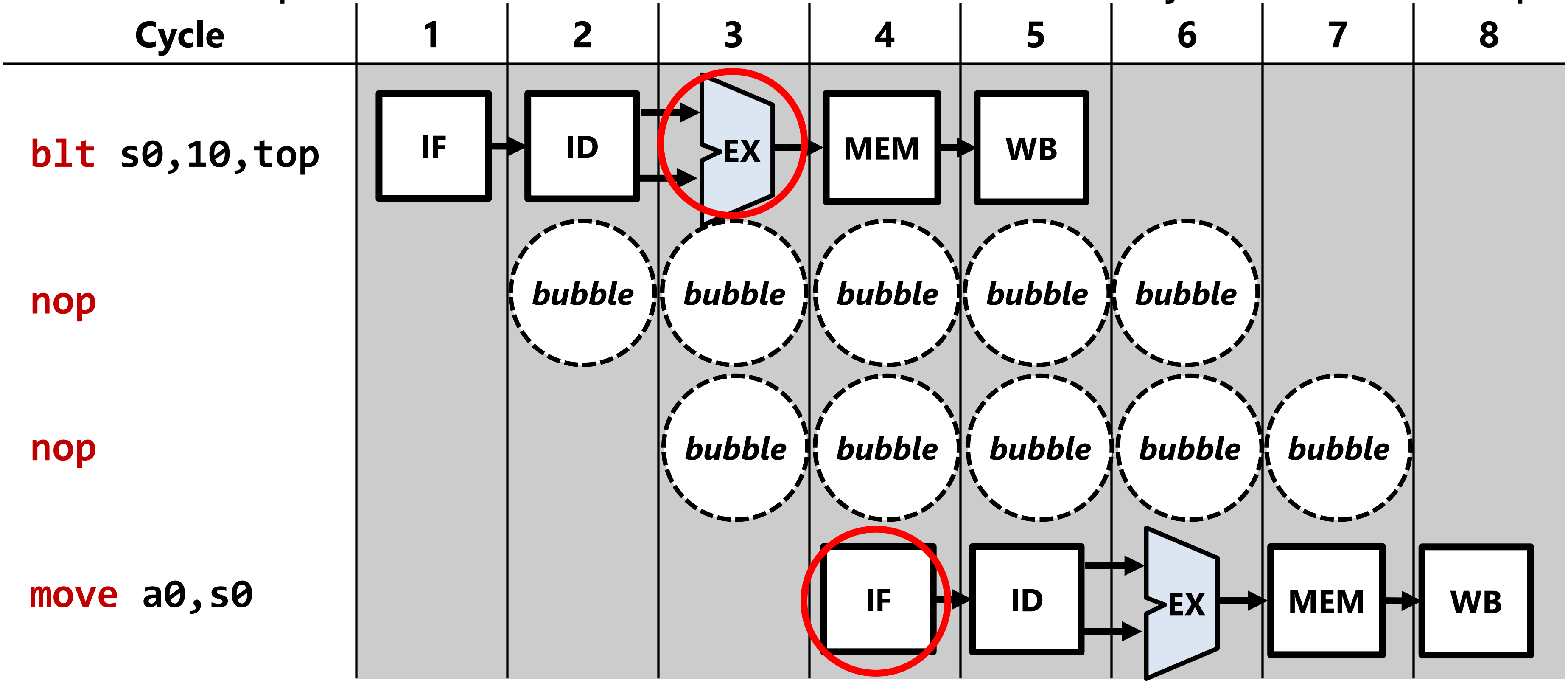
Example: If 20% of instructions are branches and the probability that a branch is taken is 50%, and pipeline flush penalty 7 cycles, then:

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

- What if we had a compiler insert no-ops, with no HDU?

Compiler avoiding the control hazard without HDU

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



Cycle	1	2	3	4	5	6	7	8
`blt s0,10,top`	IF	ID	EX	MEM	WB			
`nop`		bubble	bubble	bubble	bubble	bubble		
`nop`			bubble	bubble	bubble	bubble	bubble	
`move a0,s0`				IF	ID	EX	MEM	WB

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:

$$CPI = CPI_{nch} + 0.2 * 7 = CPI_{nch} + 1.4 \text{ cycles per instruction}$$

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

- Is there a way to minimize the performance impact?

Solution 1: Delay Slots

- This solution is specific to compiler no-ops (not dynamic flushes)

- 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
  - Delay slots are **executed when branch is taken**
  - Delay slots are **executed when branch is not taken**
  - 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

- The two **addi** instructions are moved into delay slots
  - They are not control dependent on the outcome of the branch
  - They are not data dependent on register **t0**

```
   blt  s0, 10, target
   nop  # Delay slot 1
   nop  # Delay slot 2
   addi t0, t0, 1
 target:
   add  t1, t1, t0
   add  t2, t2, t0
   addi t3, t3, 1
   addi t4, t4, 1
```

```
   blt  s0, 10, target
   addi t3, t3, 1 # Slot 1
   addi t4, t4, 1 # Slot 2
   addi t0, t0, 1
 target:
   add  t1, t1, t0
   add  t2, t2, t0
```

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

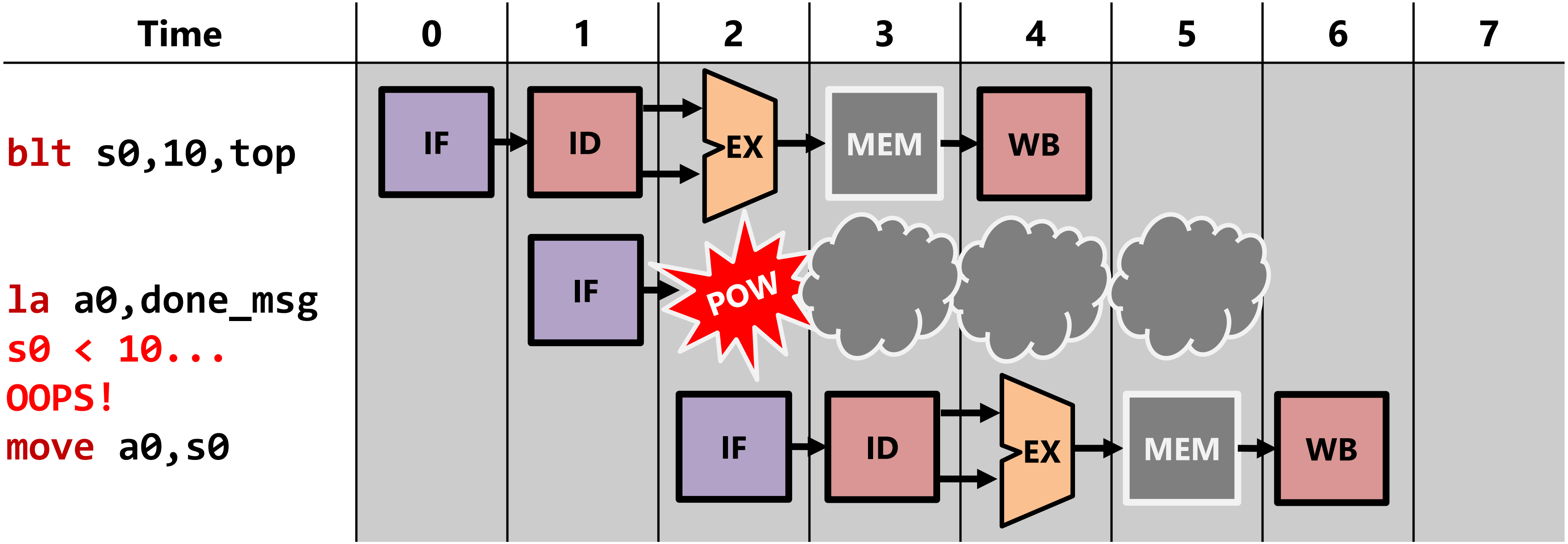
2. Delay slots baked into the ISA were not future proof
    - Number of delay slots did not match new generation of CPUs
    - New generation of CPUs had fancier ways to avoid bubbles
    - Delays slots ended up being a hindrance

- Next idea please!

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

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

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

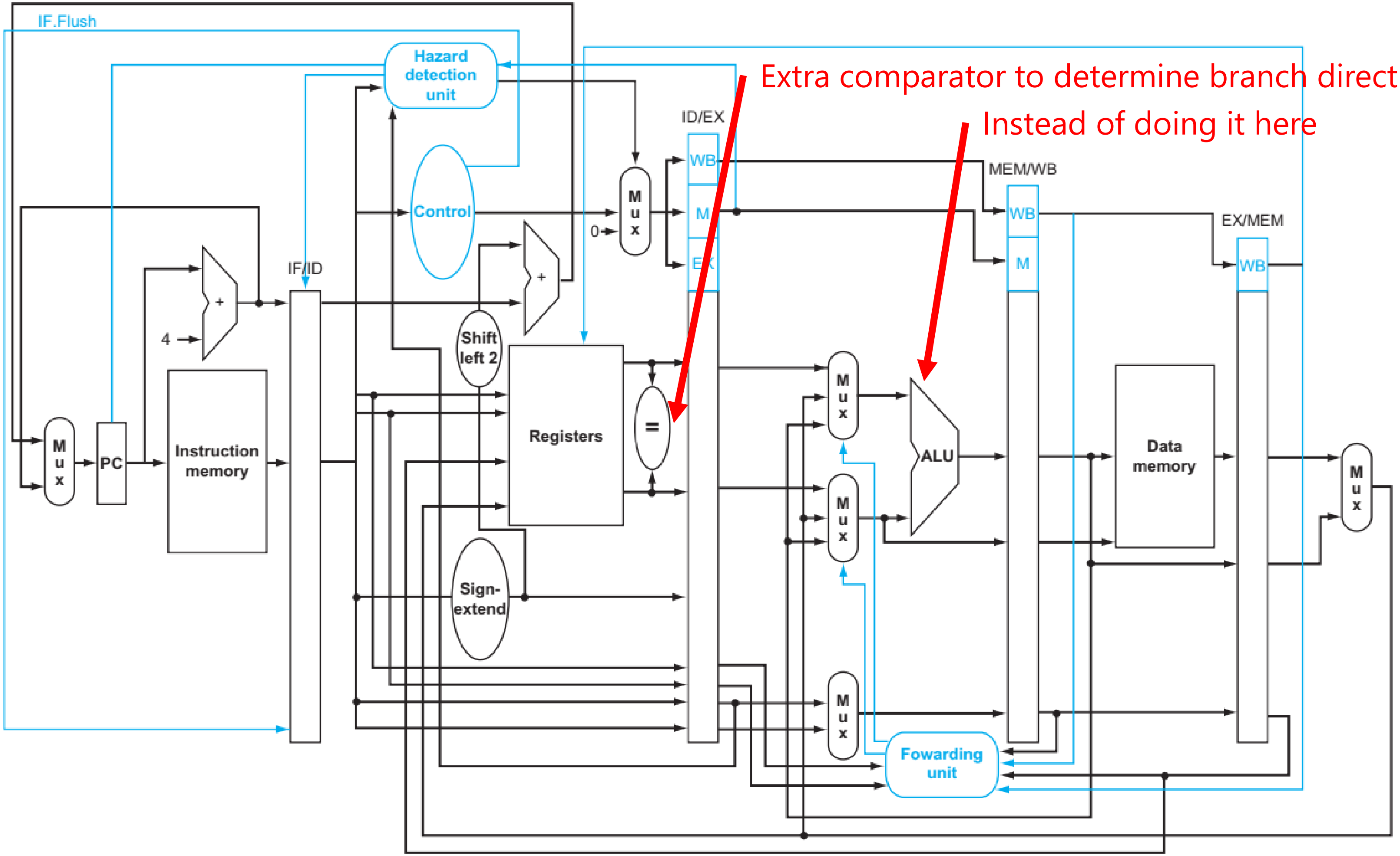


```
blt s0,10,top

la a0,done_msg
s0 < 10...
OOPS!
move a0,s0
```

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

Solution 1: MORE SINKS! (a.k.a. hardware)

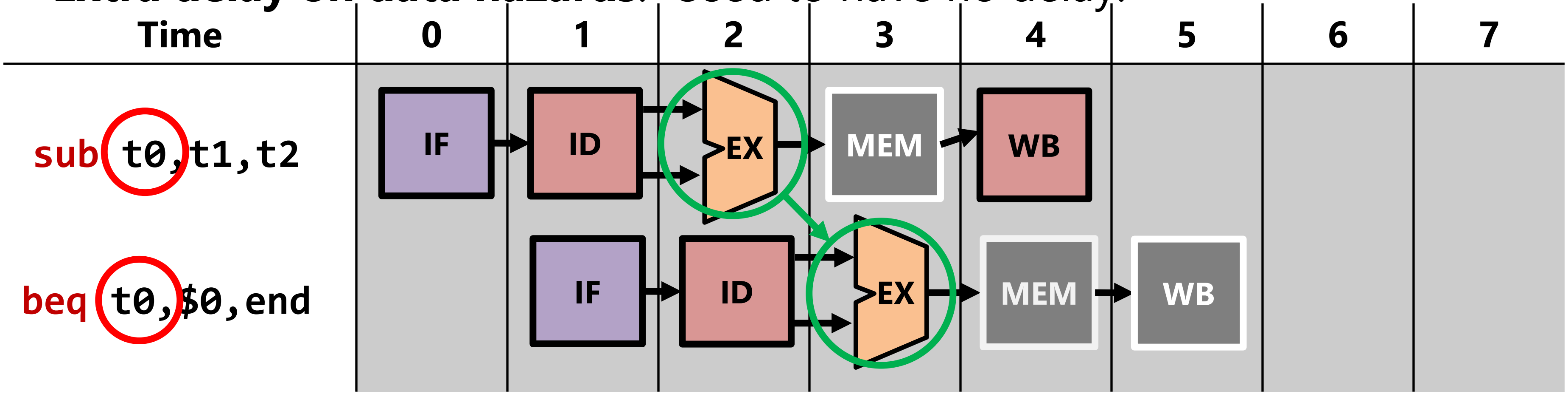


Extra comparator to determine branch direction

Instead of doing it here

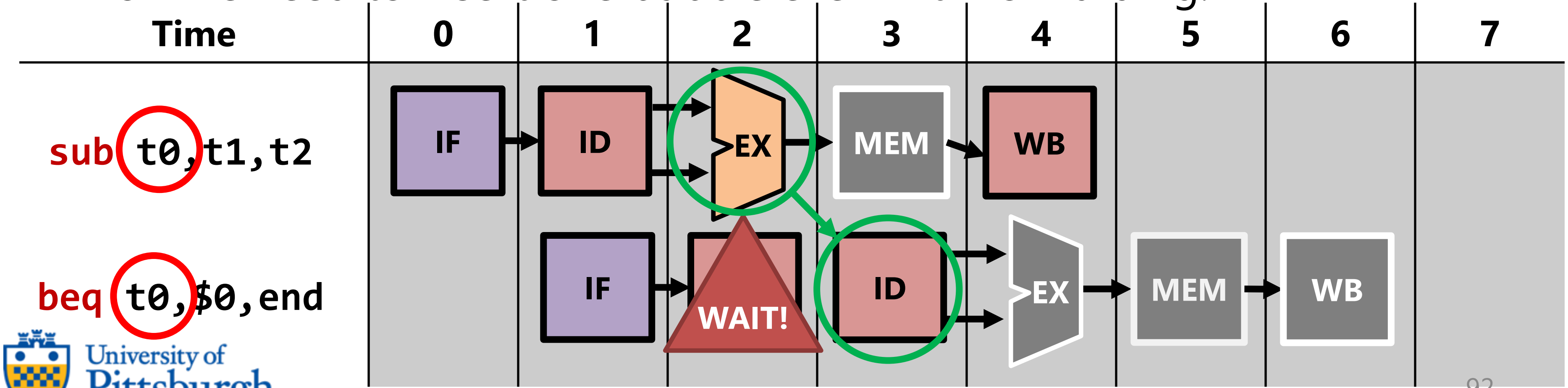
Not all sunshine and rainbows

- **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		

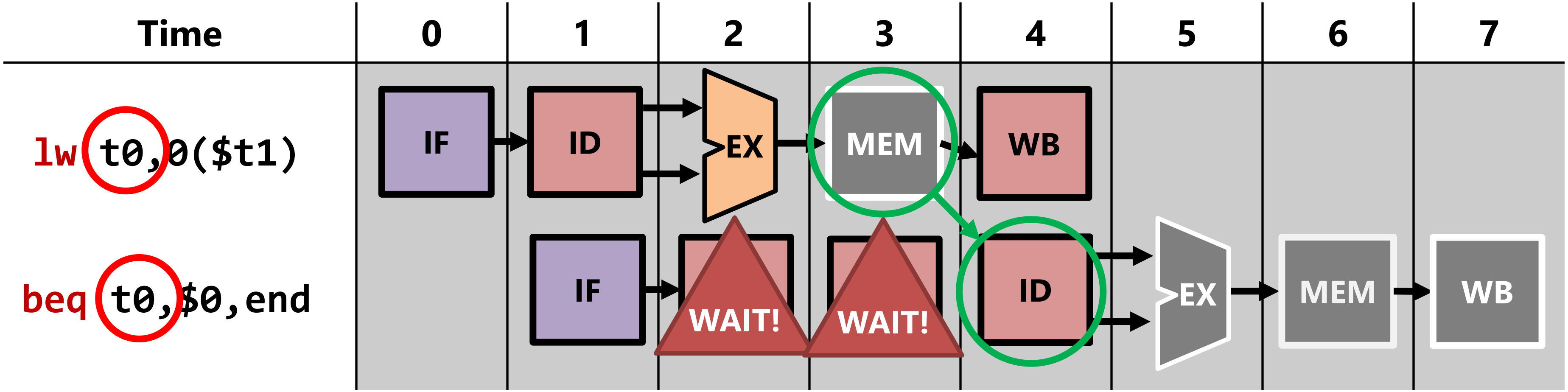
- Now we need to insert one bubble even with forwarding:



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	

Not all sunshine and rainbows

- 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:
  - From **EX** → **ID**
  - From **MEM** → **ID**
- **Doing things in more stages** means **more forwarding paths**.

Branch Prediction

Solution 3: Branch Prediction

- Comparator at ID stage is not satisfactory

- What if …
  - We were able to **predict** the branch outcome?
  - But were able to do that **without reading registers**?

- What would that get us?
  1. No extra data hazard bubble due to **EX** → **ID** forwarding
     - We are not reading register values, remember?
  2. No extra data forwarding wires
  3. While still removing bubbles from control hazards!

Types of Branch Prediction

- ***Static Branch Prediction***
  - Predicting branch behavior based on code analysis
  - **Compiler** gives hints about what to fetch next through ISA
  - Not used nowadays due to inaccuracy of compiler predictions

- ***Dynamic Branch Prediction***
  - Predicting branch behavior during program execution
  - Typically using **hardware** that tracks history information
  - *Premise*: **history repeats itself**

Dynamic Branch Prediction

- We have been doing a form of branch prediction all along!
  - We assumed that all branches will be ***not taken***

- Two simple policies:
  - Predict ***not taken***: continue fetching PC + 4, flush if taken
    Pros: Can start fetching the next instruction immediately
  - Predict ***taken***: fetch branch target as soon as ID, flush if not taken
    Pros: 67% of branches are taken, on average (due to loops)

- What if we use past history as a guide?
  - Branches not taken in the past are likely not taken in the future
    (e.g. branches to error handling code)
  - Branches taken in the past are likely taken in the future
    (e.g. branch back to the top of the loop)

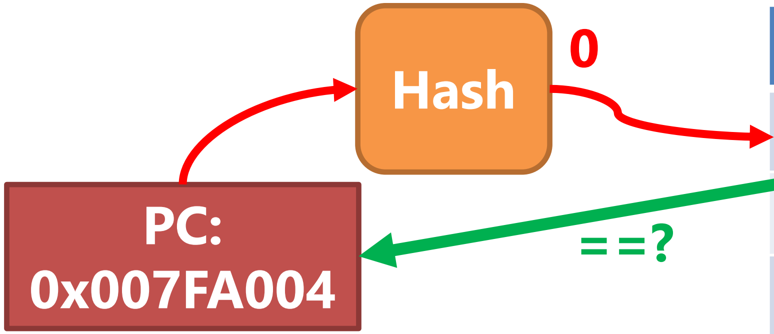
The Branch History Table (BHT)

- BHT stores Taken (**T**) or Not Take (**NT**) history info for each branch
  - If branch was taken most recently, **T** is recorded
  - If branch was not taken most recently, **NT** is recorded

- BHT is indexed using PC (Program Counter)
  - Each branch has a unique PC, so a unique entry per branch

- BHT, being hardware, is limited in capacity
  - Cannot have a huge table with all PCs possible in a program
  - Besides, not every PC address contains a branch
  - Best to use **hash table** to map branch PCs to (limited) entries

The Branch History Table (BHT)



#	Branch PC	Pred.
0	0x007FA004	T
1	0x007FC60C	NT
2	0x007FA058	T
3	…	NT
4	0x007FC380	T
5	…	T
6	…	NT
7	…	NT

T?

```
entry = Hash(PC)
if(entry.PC == PC
   && entry.pred == T)
   NextPC = inst.target
else
   NextPC = PC+4
```

To filter out conflicts in hash function

Limitations of Branch History Table (BHT)

- Ideally, we would like a prediction at the IF stage
  - So that correct instruction is immediately fetched in next cycle
  - But this is hard to do with only a BHT

- It's possible to have the BHT at the IF stage
  - All the information needed is the PC (which is available at IF)

- But must still wait until ID to decode branch target, if branch taken
  - Even if BHT gives us a T or NT prediction at the IF stage
  - If NT: no need to wait (branch target is irrelevant)
    But if T: need to wait until target decode to jump to target

- That introduces a **bubble** for **taken branches**

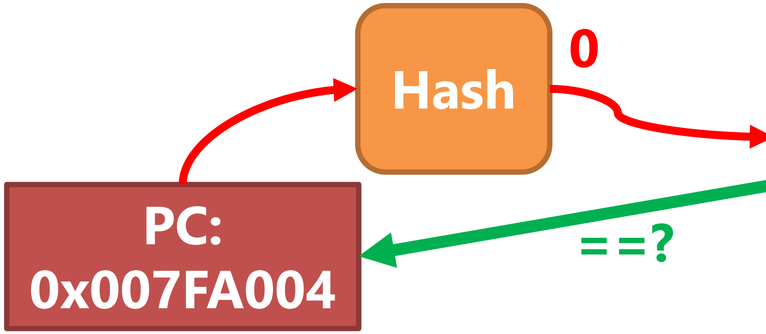
The Branch Target Buffer (BTB)

- BTB stores branch target for each branch

- BTB is also indexed using PC of branch using a hash table

- BTB allows full prediction to happen on the IF stage
  - **No need to wait until ID stage** for branch target to be decoded

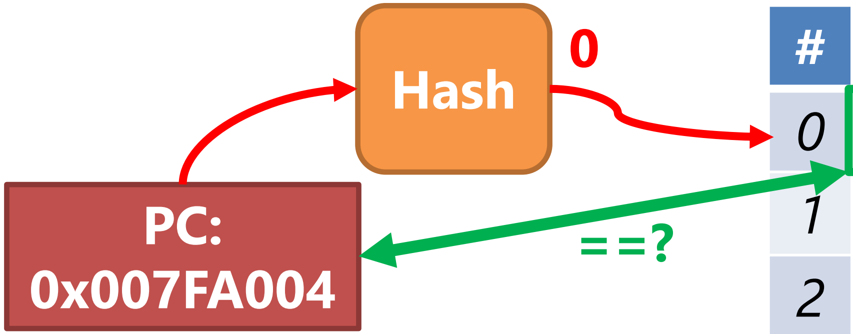
The Branch Target Buffer (BTB)



```
entry = Hash(PC)
if(entry.PC == PC
   && entry.pred == T)
   NextPC = entry.target
else
   NextPC = PC+4
```

#	Branch PC	Branch Target
0	0x007FA004	0x007FA03C
1	0x007FC60C	0x007FC704
2	0x007FA058	0x007FA040
3	…	…
4	0x007FC380	0x007FC398
5	…	…
6	…	…
7	…	…

Combined Branch Predictor

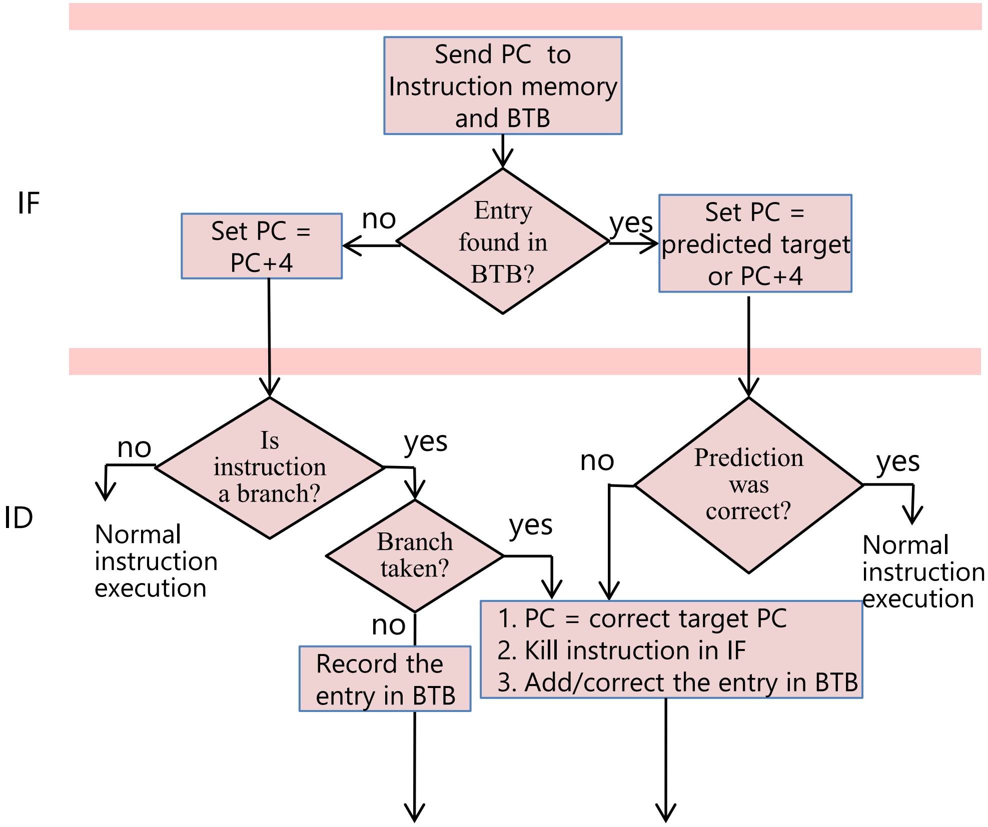


```
entry = Hash(PC)
if(entry.PC == PC
  && entry.pred == T)
  NextPC = entry.target
else
  NextPC = PC+4
```

#	Branch PC	Pred.	Branch Target
0	0x007FA004	NT	0x007FA03C
1	0x007FC60C	NT	0x007FC704
2	0x007FA058	T	0x007FA040
3	…	NT	…
4	0x007FC380	T	0x007FC398
5	…	T	…
6	…	NT	…
7	…	NT	…

Branch Prediction Decision Tree

Assuming that branch condition and target are resolved in ID stage



IF

Send PC to Instruction memory and BTB

Entry found in BTB?

no

Set PC = PC+4

yes

Set PC = predicted target or PC+4

ID

Is instruction a branch?

no

Normal instruction execution

yes

Branch taken?

yes

no

Record the entry in BTB

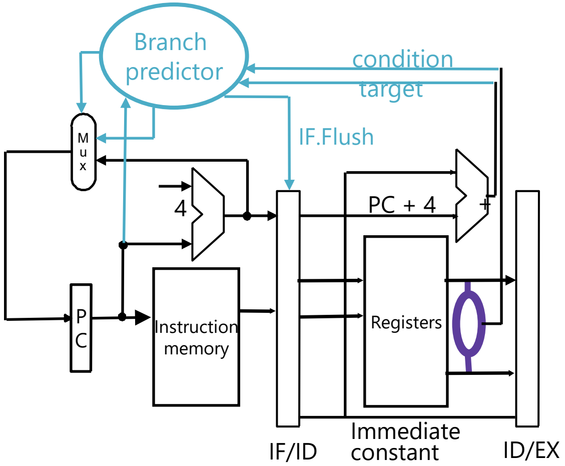
Prediction was correct?

no

1. PC = correct target PC
2. Kill instruction in IF
3. Add/correct the entry in BTB

yes

Normal instruction execution



Limitations of 1-bit BHT Predictor

- Is 1-bit (T / NT) enough history to make a good decision?

- Take a look at this example:

```
for (j=0; j<100; j++) {
  for (i=0; i< 5; i++) {
    A[i] = B[i] * C[i];
    D[i] = E[i] / F[i];
  }
}
```

Predicted	-	T	T	T	**T**	**NT**	T	T	T	**T**	**NT**	T	T
Actual	T	T	T	T	NT	T	T	T	T	NT	T	T	T

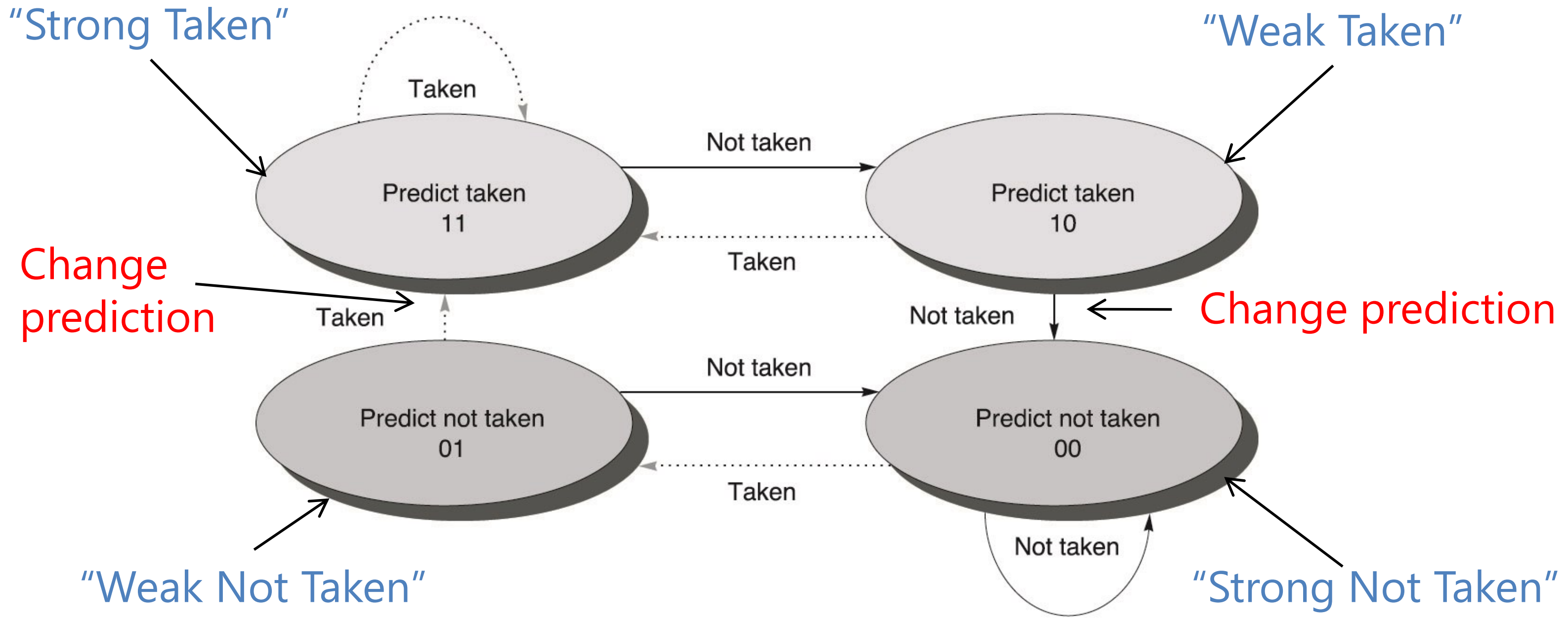
this branch is predicted wrong twice every inner loop invocation (every 5 branches)

- It would have been better to stay with T than flip back and forth!

- Idea behind the 2-bit predictor: create some hysteresis
  - So that predictions don't flip immediately

2-bit BHT Predictor

- State transition diagram of 2-bit predictor:



- Requires two consecutive mis-predictions to flip direction!

2-bit BHT Predictor

- How well does the 2-bit predictor do with our previous example?

- Our previous example:

```
for (j=0; j<100; j++) {
  for (i=0; i< 5; i++) {
    A[i] = B[i] * C[i];
    D[i] = E[i] / F[i];
  }
}
```

Predicted	-	T	T	T	**T**	T	T	T	T	**T**	T	T	T
Actual	T	T	T	T	NT	T	T	T	T	NT	T	T	T

this branch is predicted wrong **only once** every inner loop invocation (every 5 branches)

- Does it help beyond 2 bits? (e.g. 3-bit predictor, or 4-bit predictor)
  - Empirically, no. 2 bits already cover loop which is most common.
  - 2 bits + large BHT gets you **~93% accuracy**

- We need other tricks to improve accuracy!

Correlating Predictors

- Sometimes you need to know more than the PC of your branch

```
for (j=0; j<100; j++) {
  if (j % 2) {
  }
}
```

You get the prediction **wrong every single time**!

Predicted	-	**NT**	**T**	**NT**	**T**	**NT**	**T**	**NT**	**T**	**NT**	**T**	**NT**	**T**
Actual	NT	T	NT	T	NT	T	NT	T	NT	T	NT	T	NT

  - For a 1-bit predictor, but a 2-bit predictor doesn't do well either
  - Should base the prediction also on the history of that branch!
  - This is called ***local branch history*** *(since it's the same branch)*
- Knowing the result of other branches in your history also helps

```
If (j == 0) {
}
...
If (j != 0) {
}
```

Knowing result of a **previous different branch** in your history helps in predicting this branch!

  - This is called ***global branch history*** *(since it's a different branch)*

University of Pittsburgh

Correlating Predictors

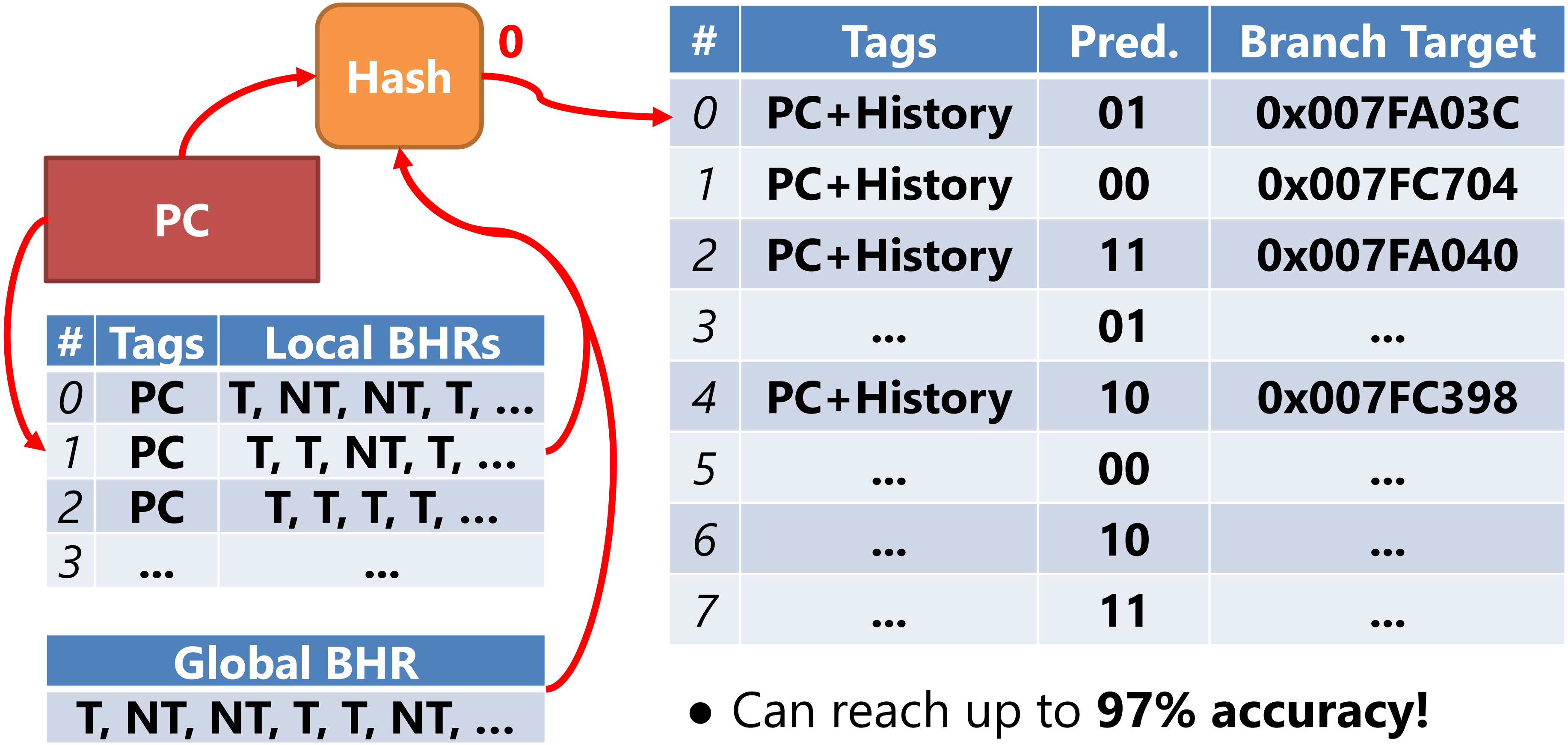
- Idea: have multiple entries per branch depending on history
  - Local branch history + Global branch history
  - An entry with matching history gives more precise prediction!

- Now, instead of indexing into BHT by branch PC only
  - Use hash of PC + Local branch history + Global branch history

- History is stored in register called Branch History Shift Register (BHR)
  - T/NT bit is shifted on to BHR whenever branch is encountered

  1. One Global BHR (there is just one global history)
  2. Multiple Local BHRs (local histories for each branch)

Correlating Predictors



#	Tags	Local BHRs
0	PC	T, NT, NT, T, …
1	PC	T, T, NT, T, …
2	PC	T, T, T, T, …
3	…	…

Global BHR
T, NT, NT, T, T, NT, …

#	Tags	Pred.	Branch Target
0	PC+History	01	0x007FA03C
1	PC+History	00	0x007FC704
2	PC+History	11	0x007FA040
3	…	01	…
4	PC+History	10	0x007FC398
5	…	00	…
6	…	10	…
7	…	11	…

- Can reach up to **97% accuracy!**

How about jr $ra?

- **jr $ra**: Jump return to address stored in **$ra**
  - When a function is called, the caller stores return address to **$ra** (**jal funcAddr** stores PC of next instruction to **$ra**)
  - When a function returns, **jr $ra** jumps to return address in **$ra**

- Why is this a problem?
  - Unlike other branches, branch target is not an immediate value! (Jumping to a variable target is called an ***indirect branch***)
  - Target can change for same **jr** depending on who caller is
  - Makes life difficult for BTB which relies on target being constant

- Target of **jr** is predicted using the ***Return Stack Buffer***
  - Not the Branch Target Buffer (BTB)

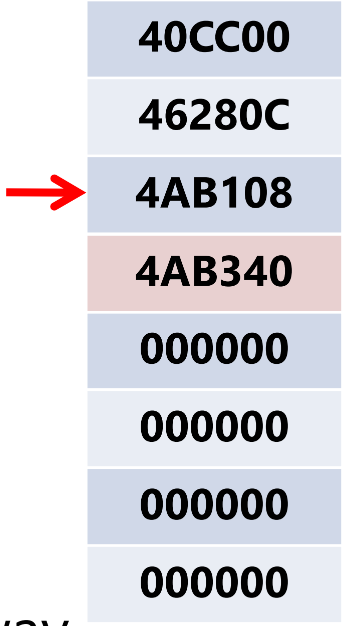
The Return Stack Buffer

- Since functions return to where they were called every time, it makes sense to cache the return addresses (in a stack)

```
4AB33C jal someFunc
4AB340 beq v0, $0, blah
       ...
       someFunc:
           ...
           jr $ra
```

When we encounter the jal, push the return address.

When we encounter the jr $ra, pop the return address. Easy!



40CC00

46280C

4AB108

4AB340

000000

000000

000000

000000

- On misprediction or stack overflow, empty stack
  - Not a problem since this is for prediction anyway