

Data Hazards

Today

- Quiz 4
- Hazards
- Watch for announcement about signing up for a project interview.
- Remember! There is a midterm on the 3rd. It covers everything through next Tuesday.
- Review session: Next Thursday. Bring questions.

Hazards: Key Points

- Hazards cause imperfect pipelining
 - They prevent us from achieving $CPI = 1$
 - They are generally caused by “counter flow” data dependences in the pipeline
- Three kinds
 - Structural -- contention for hardware resources
 - Data -- a data value is not available when/where it is needed.
 - Control -- the next instruction to execute is not known.
- Two ways to deal with hazards
 - Removal -- add hardware and/or complexity to work around the hazard so it does not exist
 - Bypassing/forwarding
 - Speculation
 - Stall -- Sacrifice performance to prevent the hazard from occurring
 - Bubbles

Data Dependences

- A data dependence occurs whenever one instruction needs a value produced by another.
 - Register values (for now)
 - Also memory accesses (more on this later)

```
add $s0, $t0, $t1
```

```
sub $t2, $s0, $t3
```

```
add $t3, $s0, $t4
```

```
and $t3, $t2, $t4
```

```
sw  $t1, 0($t2)
```

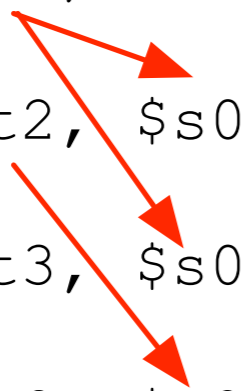
```
ld  $t3, 0($t2)
```

```
ld  $t4, 16($s4)
```

Data Dependences

- A data dependence occurs whenever one instruction needs a value produced by another.
 - Register values (for now)
 - Also memory accesses (more on this later)

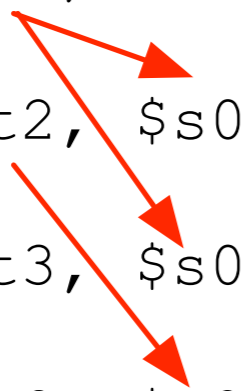
<code>add \$s0, \$t0, \$t1</code>	<code>sw \$t1, 0(\$t2)</code>
<code>sub \$t2, \$s0, \$t3</code>	<code>ld \$t3, 0(\$t2)</code>
<code>add \$t3, \$s0, \$t4</code>	<code>ld \$t4, 16(\$s4)</code>
<code>and \$t3, \$t2, \$t4</code>	



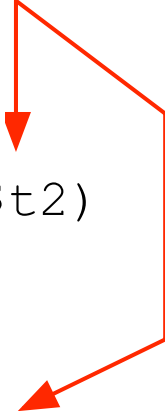
Data Dependences

- A data dependence occurs whenever one instruction needs a value produced by another.
 - Register values (for now)
 - Also memory accesses (more on this later)

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
add $t3, $s0, $t4
and $t3, $t2, $t4
```

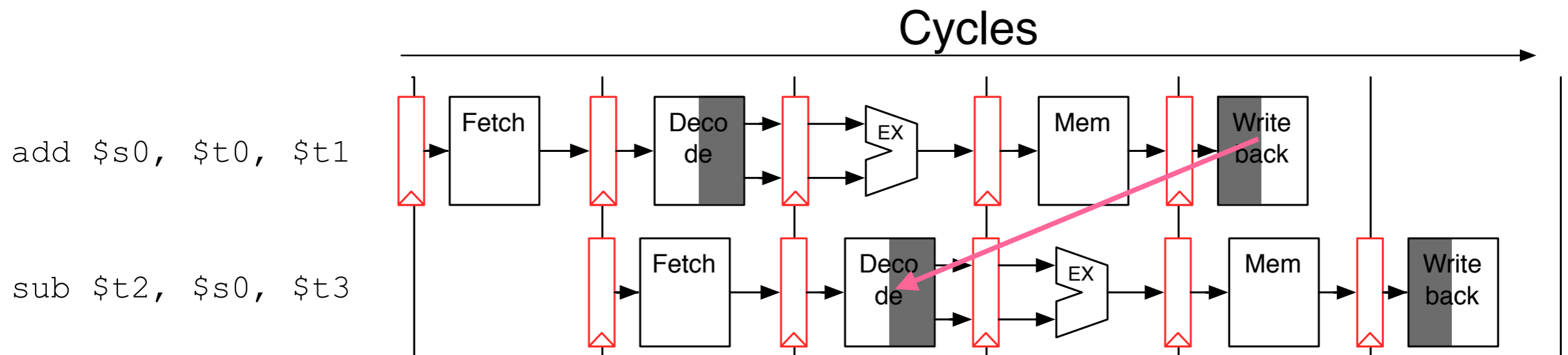


```
sw $t1, 0($t2)
ld $t3, 0($t2)
ld $t4, 16($s4)
```



Dependences in the pipeline

- In our simple pipeline, these instructions cause a hazard

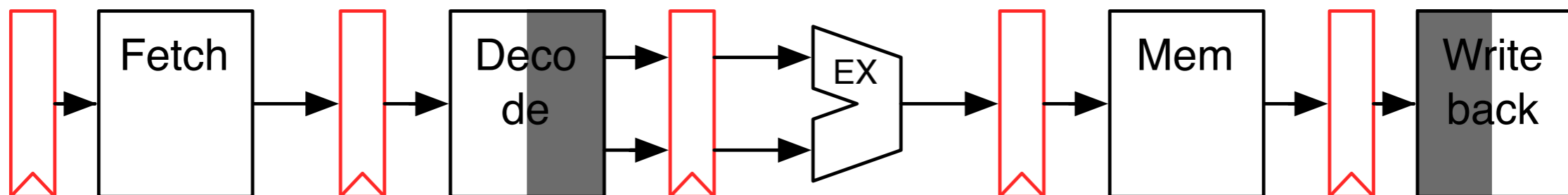


How can we fix it?

- Ideas?

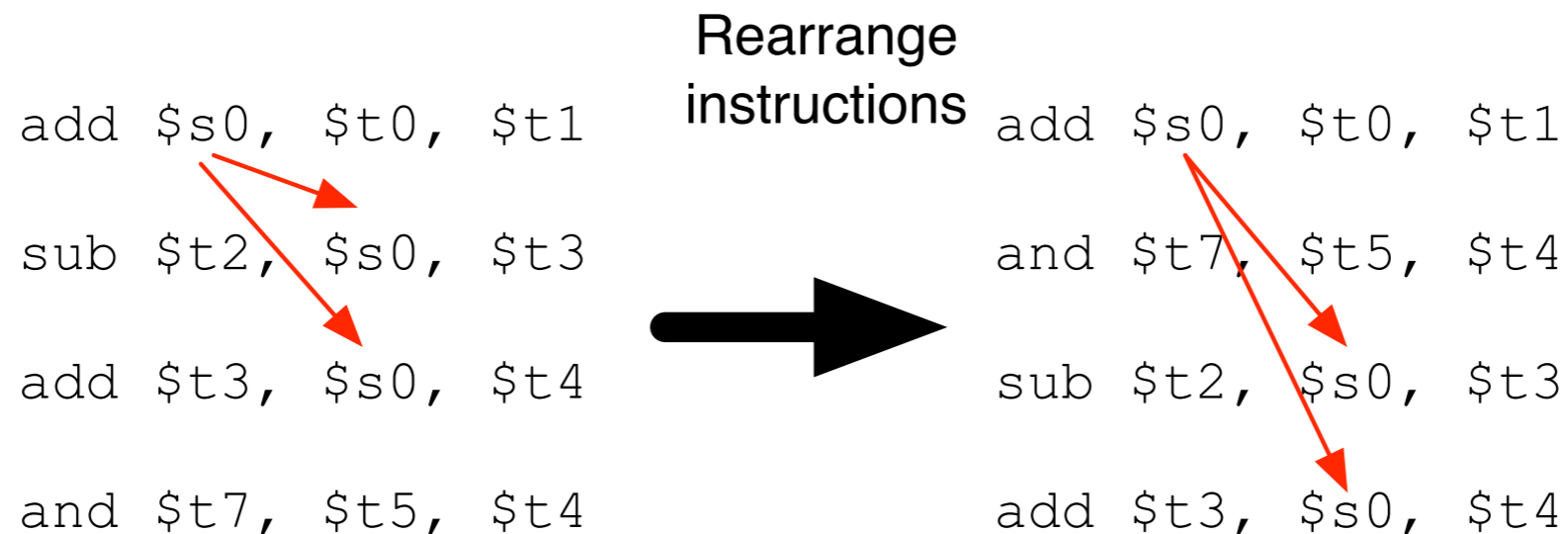
Solution 1: Make the compiler deal with it.

- Expose hazards to the big A architecture
 - A result is available N instructions after the instruction that generates it.
 - In the meantime, the register file has the old value.
 - “delay slots”
- What is N ?
- Can it change?
- What can the compiler do?



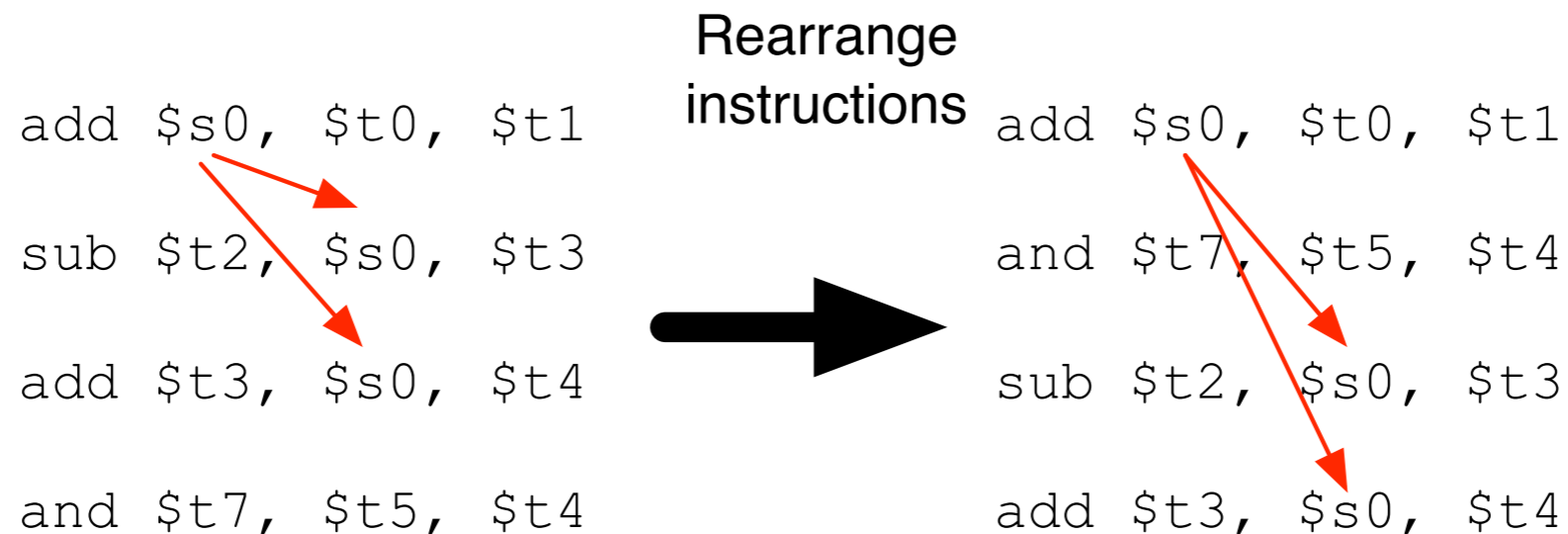
Compiling for delay slots

- The compiler must fill the delay slots with other instructions
- What if it can't?



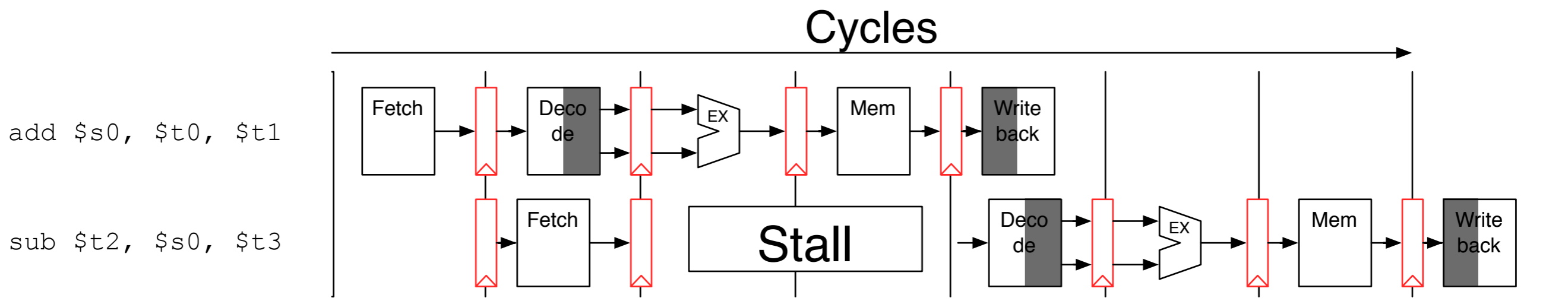
Compiling for delay slots

- The compiler must fill the delay slots with other instructions
- What if it can't? **No-ops**



Solution 2: Stall

- When you need a value that is not ready, “stall”
 - Suspend the execution of the executing instruction
 - and those that follow.
 - This introduces a pipeline “bubble”



Stalling the pipeline

- All pipeline stages preceding the stage where the hazard occurs freeze
 - Disable the PC update
 - Disable the pipeline registers
- This essentially equivalent to always inserting a nop when a hazard exists
 - Insert nop control bits at stalled stage (decode in our example)
 - How is this solution still potentially “better” than relying on the compiler?

Stalling the pipeline

- All pipeline stages preceding the stage where the hazard occurs freeze
 - Disable the PC update
 - Disable the pipeline registers
- This is essentially equivalent to always inserting a nop when a hazard exists
 - Insert nop control bits at stalled stage (decode in our example)
 - How is this solution still potentially “better” than relying on the compiler?

The compiler can still act like there are delay slots to avoid stalls.
Implementation details are not exposed in the ISA

The Impact of Stalling On Performance

- $ET = I * CPI * CT$
- I and CT are constant
- What is the impact of stalling on CPI ?

- What do we need to know to figure it out?

The Impact of Stalling On Performance

- $ET = I * CPI * CT$
- I and CT are constant
- What is the impact of stalling on CPI ?

- Fraction of instructions that stall: 30%
- Baseline $CPI = 1$
- Stall $CPI = 1 + 2 = 3$

- New $CPI =$

The Impact of Stalling On Performance

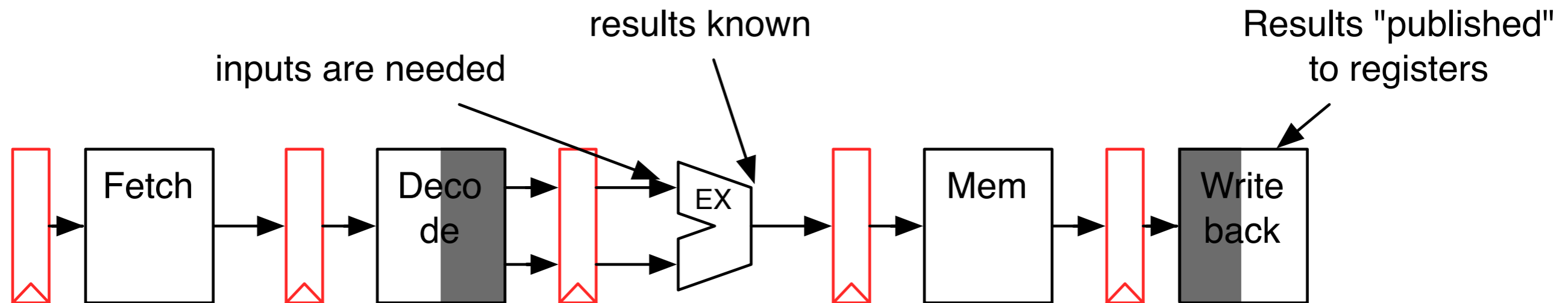
- $ET = I * CPI * CT$
- I and CT are constant
- What is the impact of stalling on CPI ?

- Fraction of instructions that stall: 30%
- Baseline $CPI = 1$
- Stall $CPI = 1 + 2 = 3$

- New $CPI = 0.3*3 + 0.7*1 = 1.6$

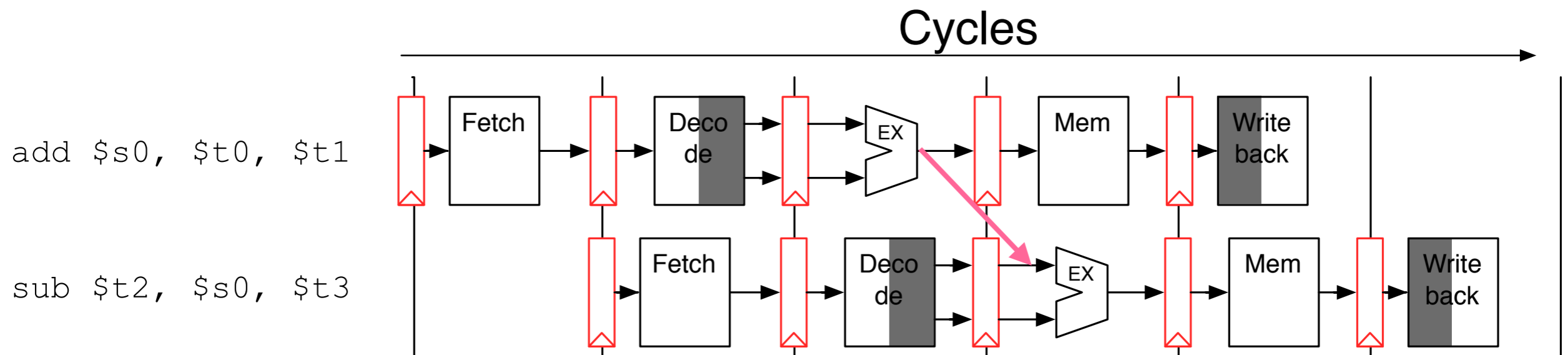
Solution 3: Bypassing/Forwarding

- Data values are computed in Ex and Mem but “publicized in write back”
- The data exists! We should use it.

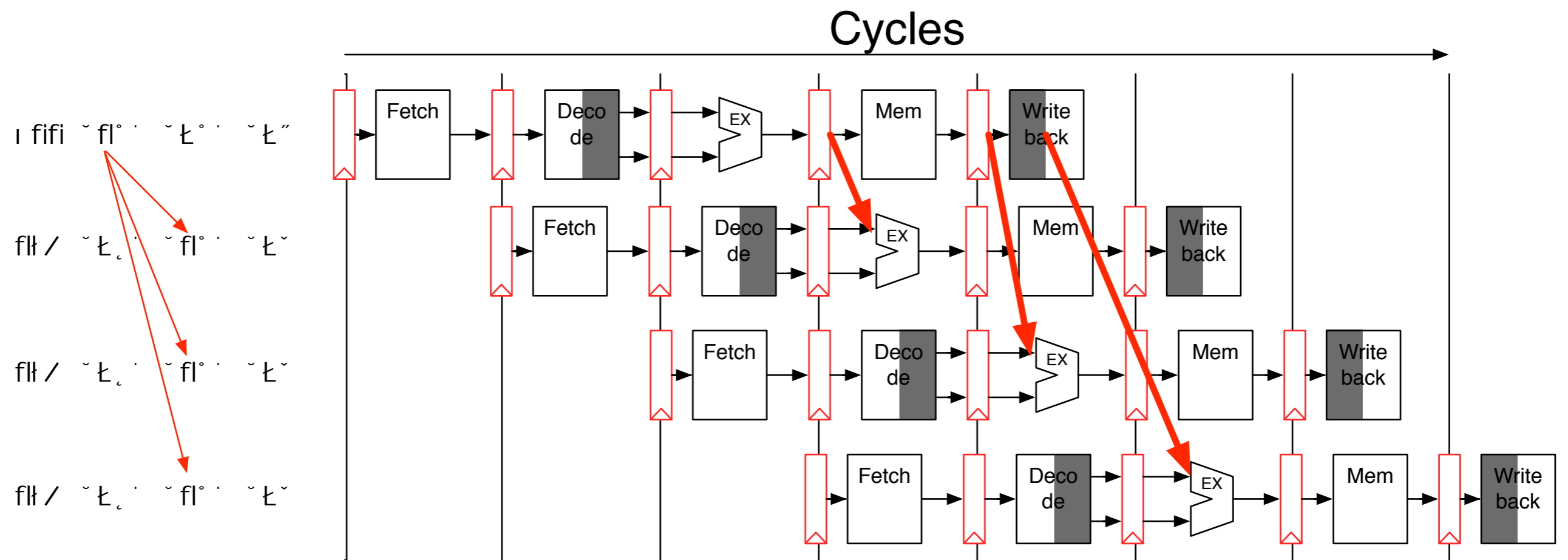


Bypassing or Forwarding

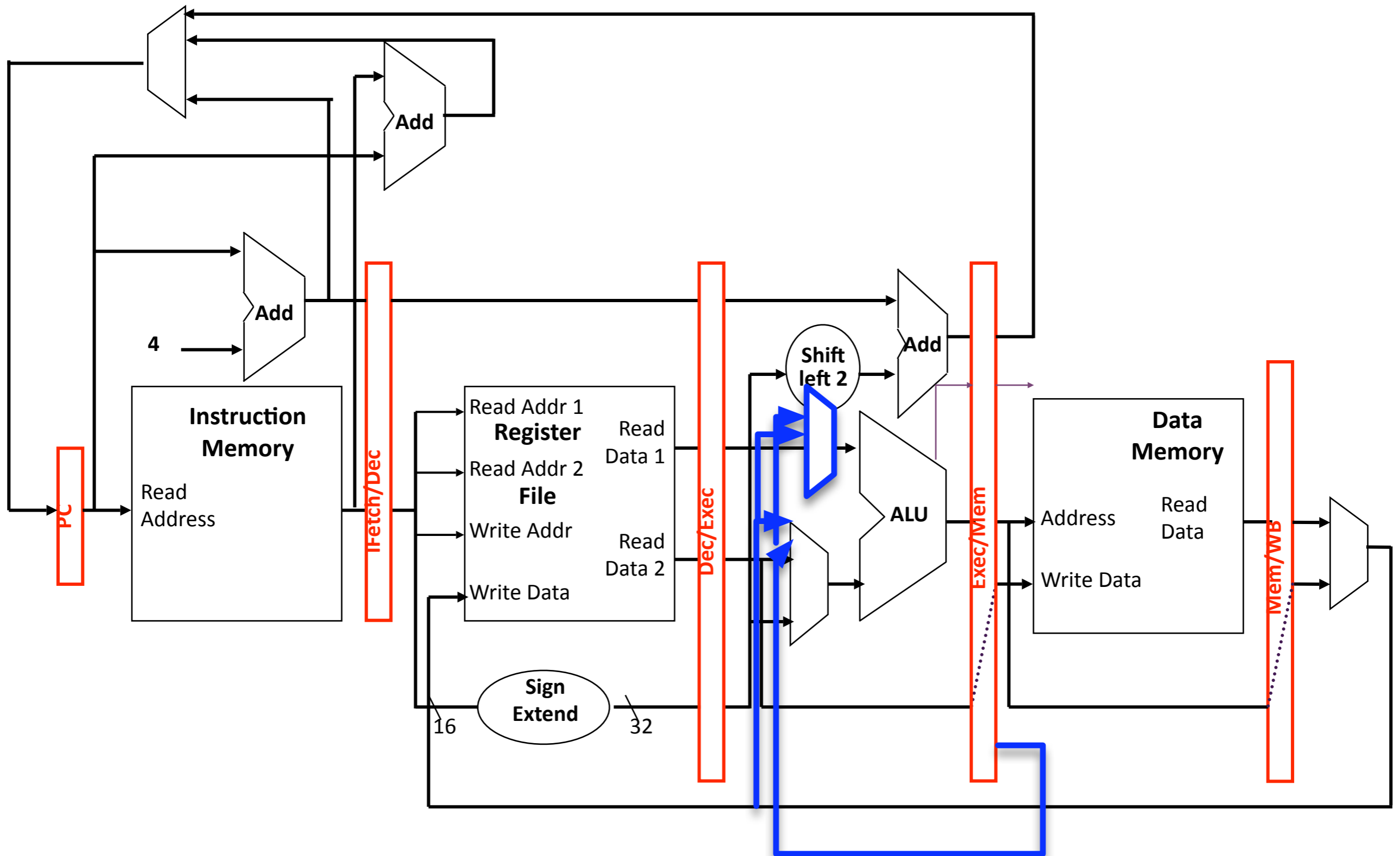
- Take the values, where ever they are



Forwarding Paths



Forwarding in Hardware

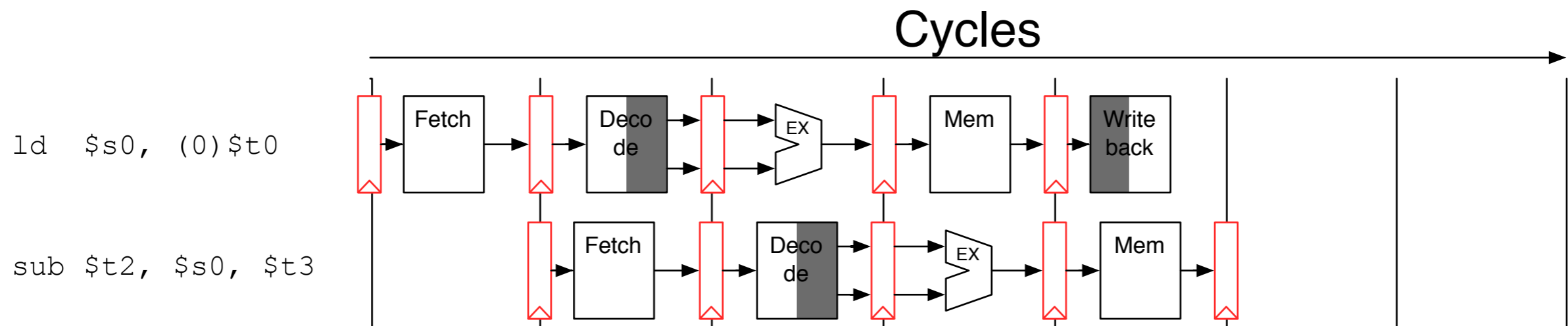


Forwarding Control

- The forwarding unit detects instances when the destination and source registers of executing instructions match
 - Set the control lines on the ALU input muxes accordingly
 - Stall if, for some reason, forwarding is not possible.

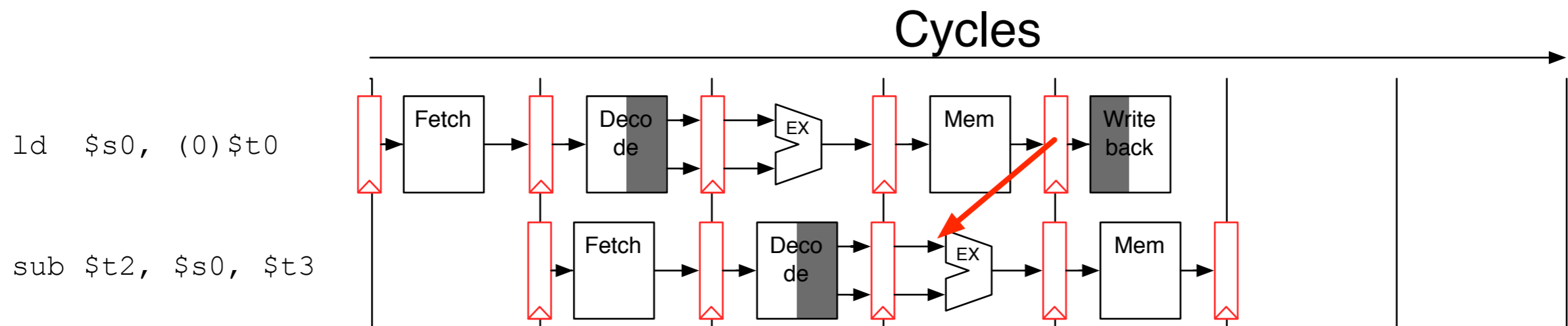
Forwarding for Loads

- Load values come from the Mem stage



Forwarding for Loads

- Load values come from the Mem stage



Time travel presents significant implementation challenges

What can we do?

- Punt to the compiler
 - Easy enough.
 - Will work.
 - Same dangers apply as before.
- Always stall.
- Forward when possible, stall otherwise
 - Here the compiler still has leverage
 - Code will be faster if the compiler generates code as if there is a delay slot.
 - If the compiler can't fix it, the hardware will stall

Performance cost of stalling

- $ET = I * CPI * CT$
- $CPI = \%Stall * StallTime$
- % Stall is determined by how aggressive our bypassing is and the quality of our compiler.
- Stall time is related to pipeline depth. In our case, it is 1 or 2, because our pipeline is shallow.
- In deeper pipelines, it can be larger.

Hardware Cost of Forwarding

- In our pipeline, adding forwarding required relatively little hardware.
- For deeper pipelines it gets much more expensive
 - Roughly: $ALU * \text{pipeline stages}$ you need to forward over
 - Some modern processor have multiple ALUs (4-5)
 - And deeper pipelines (4-5 stages of to forward across)
- Not all forwarding paths need to be supported.
 - If a path does not exist, the processor will need to stall.