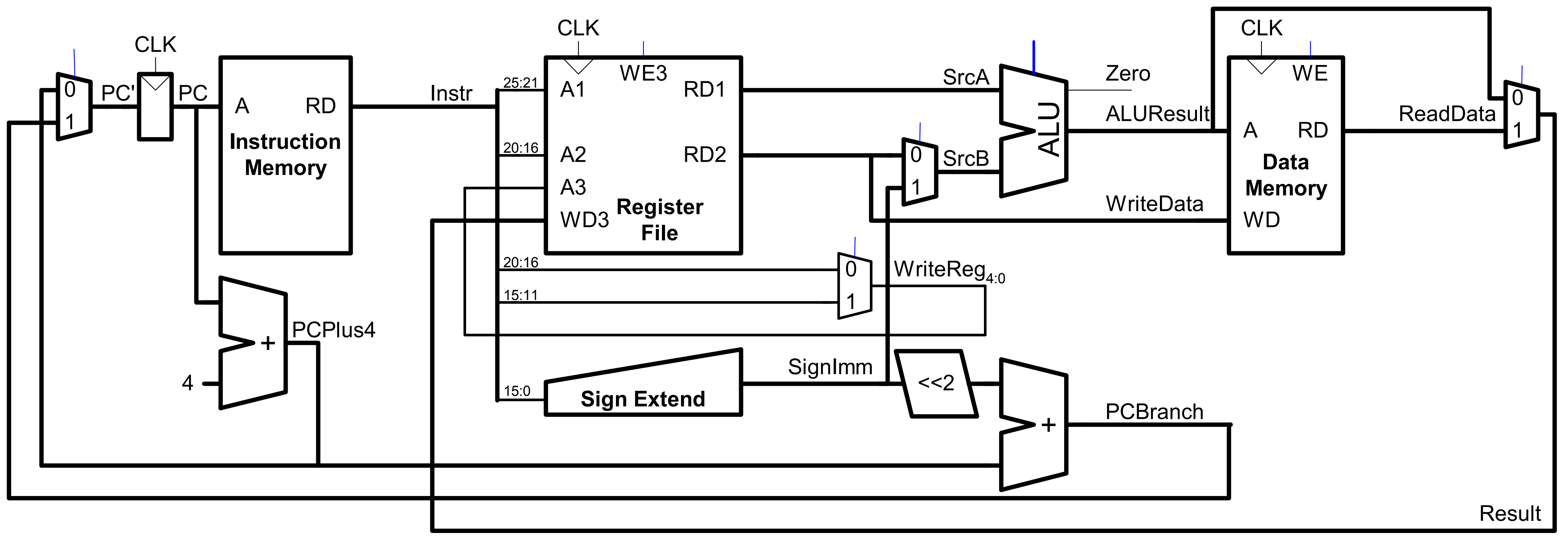
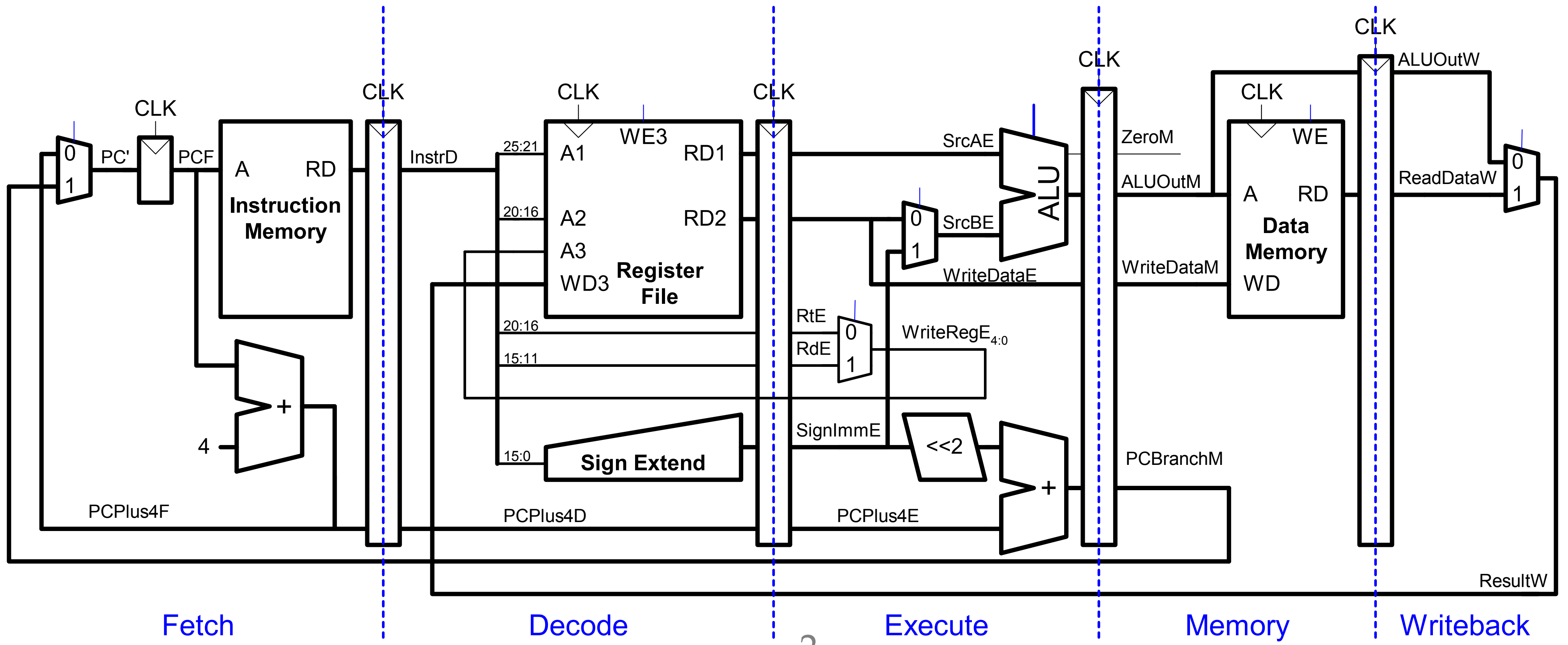
EECE 2322: Fundamentals of Digital Design and Computer Organization

Lecture 14_2: Microarchitecture

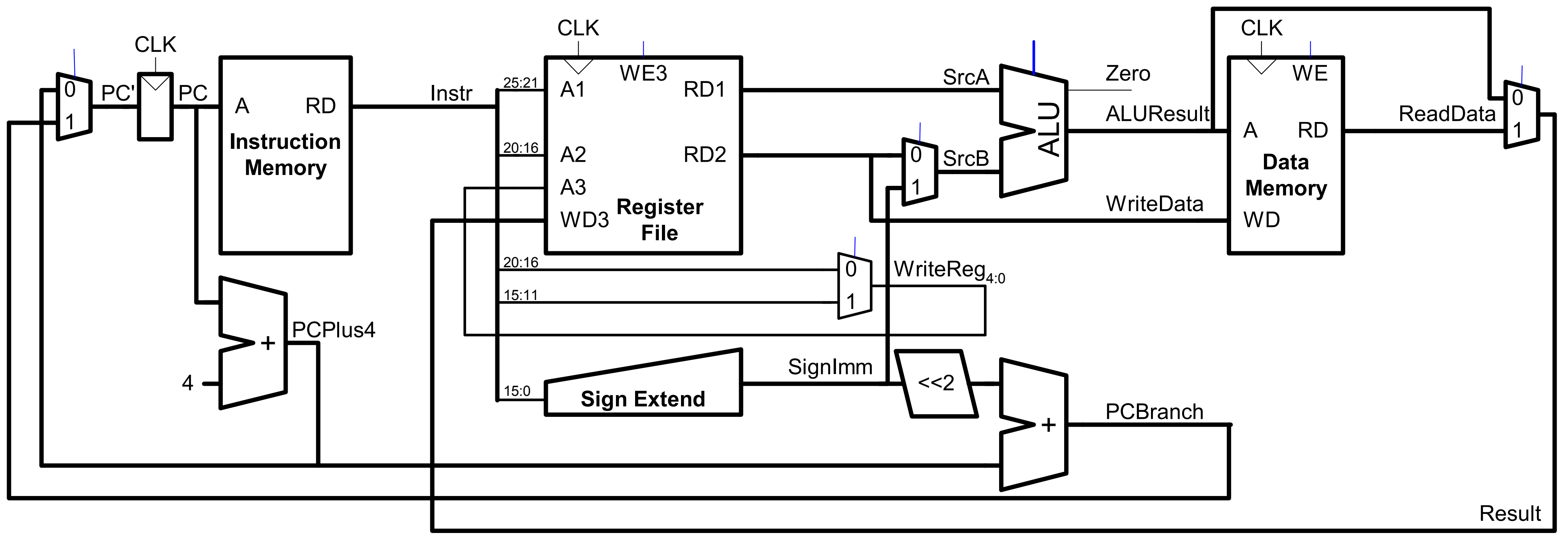
Xiaolin Xu
Department of ECE
Northeastern University

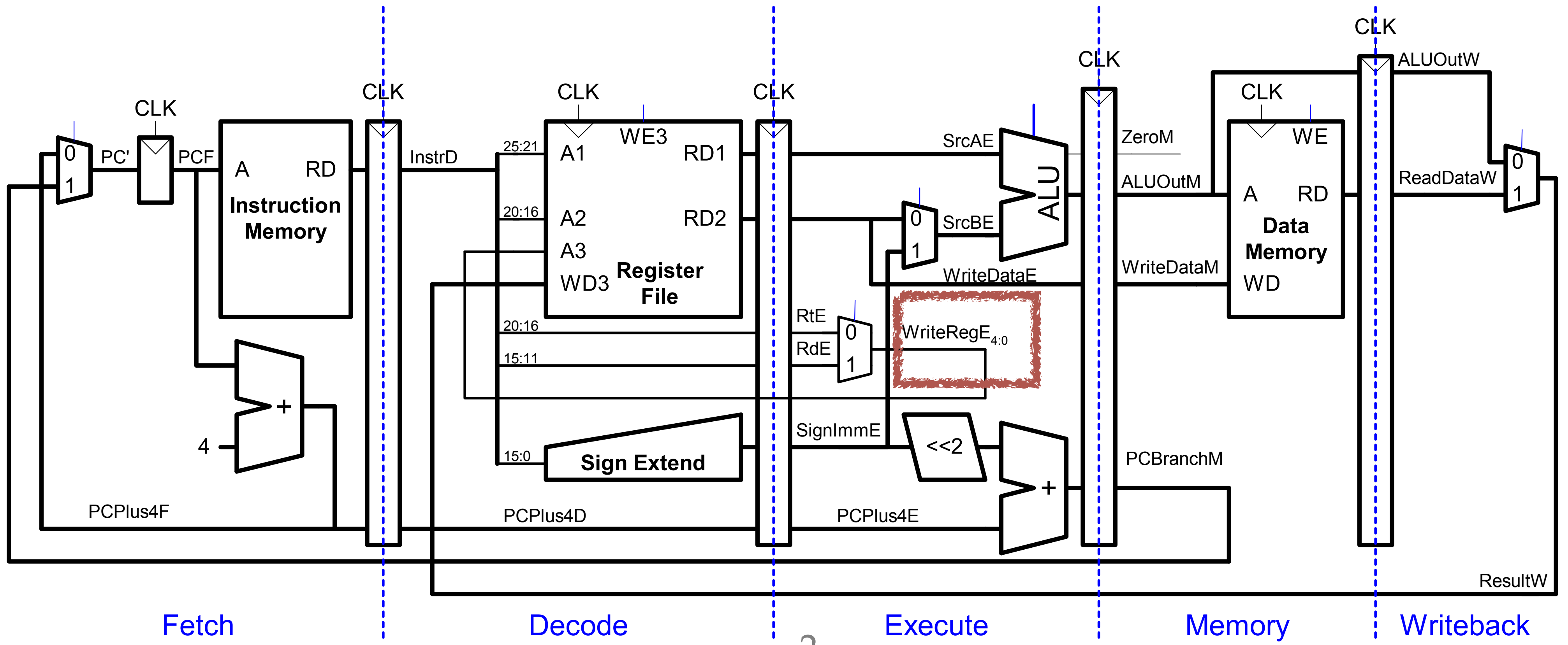
Single-Cycle & Pipelined Datapath



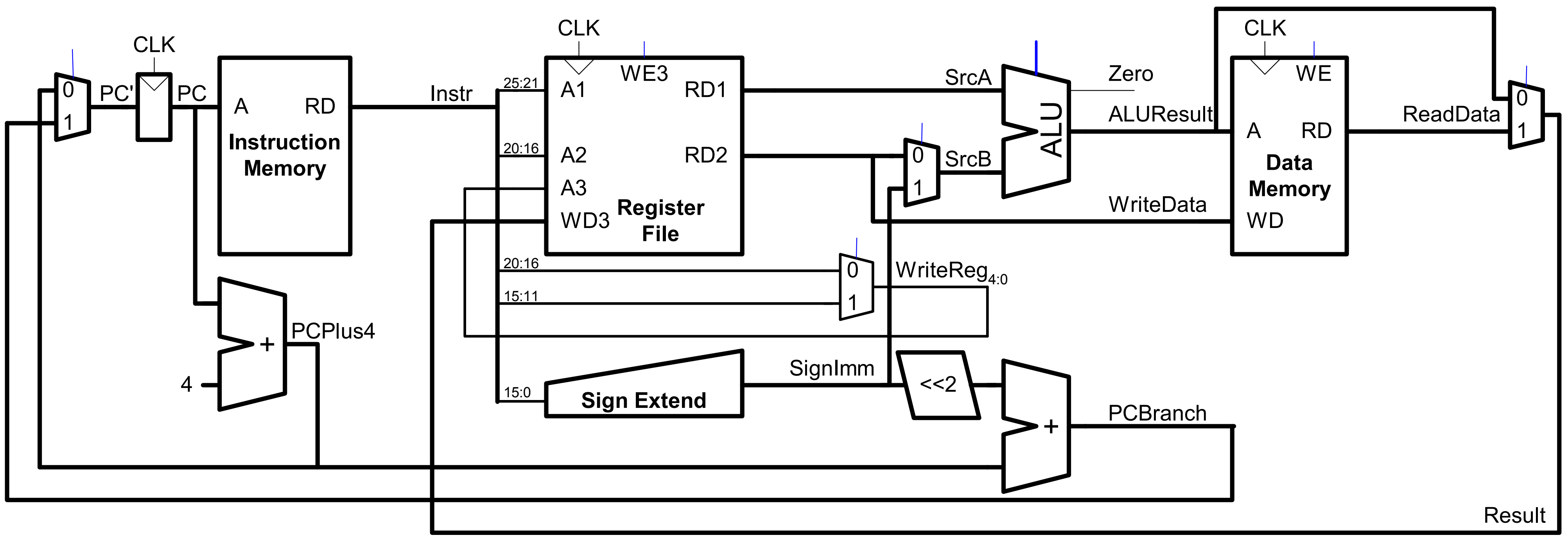


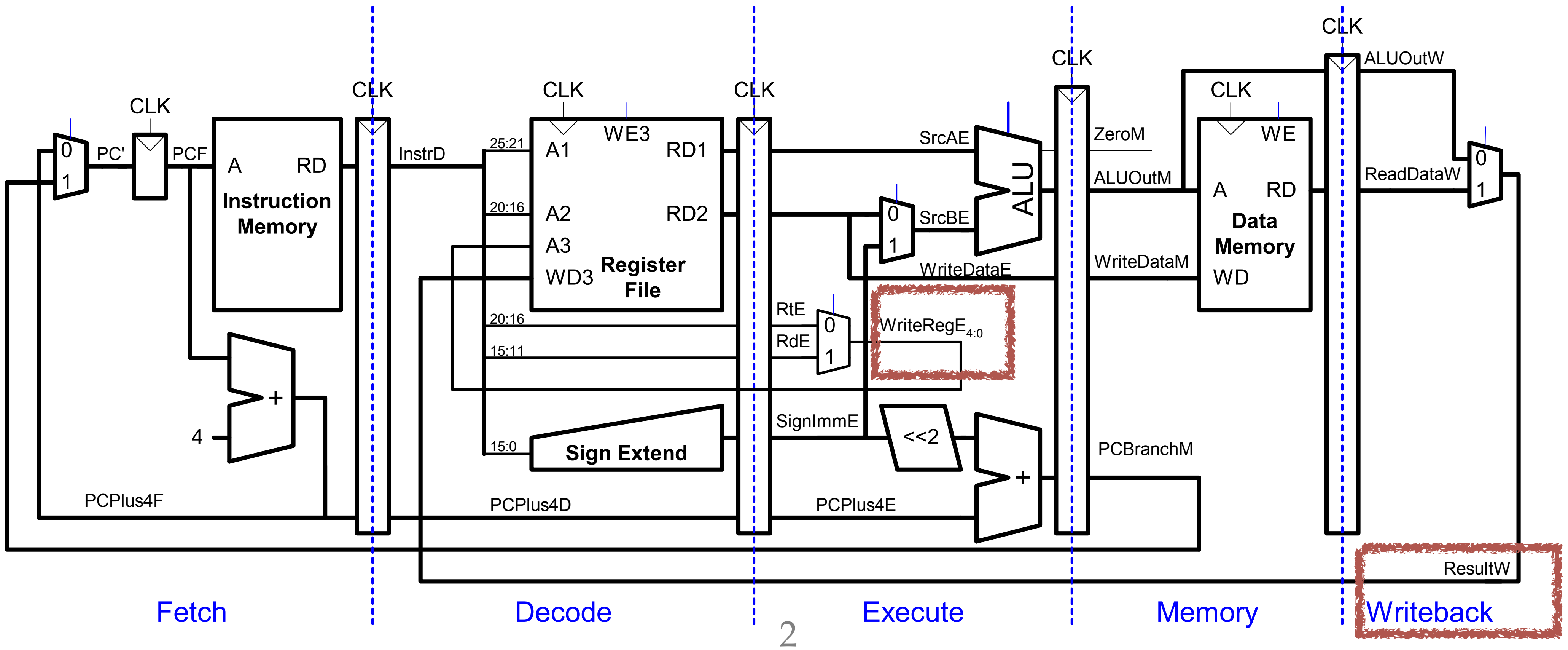
Single-Cycle & Pipelined Datapath



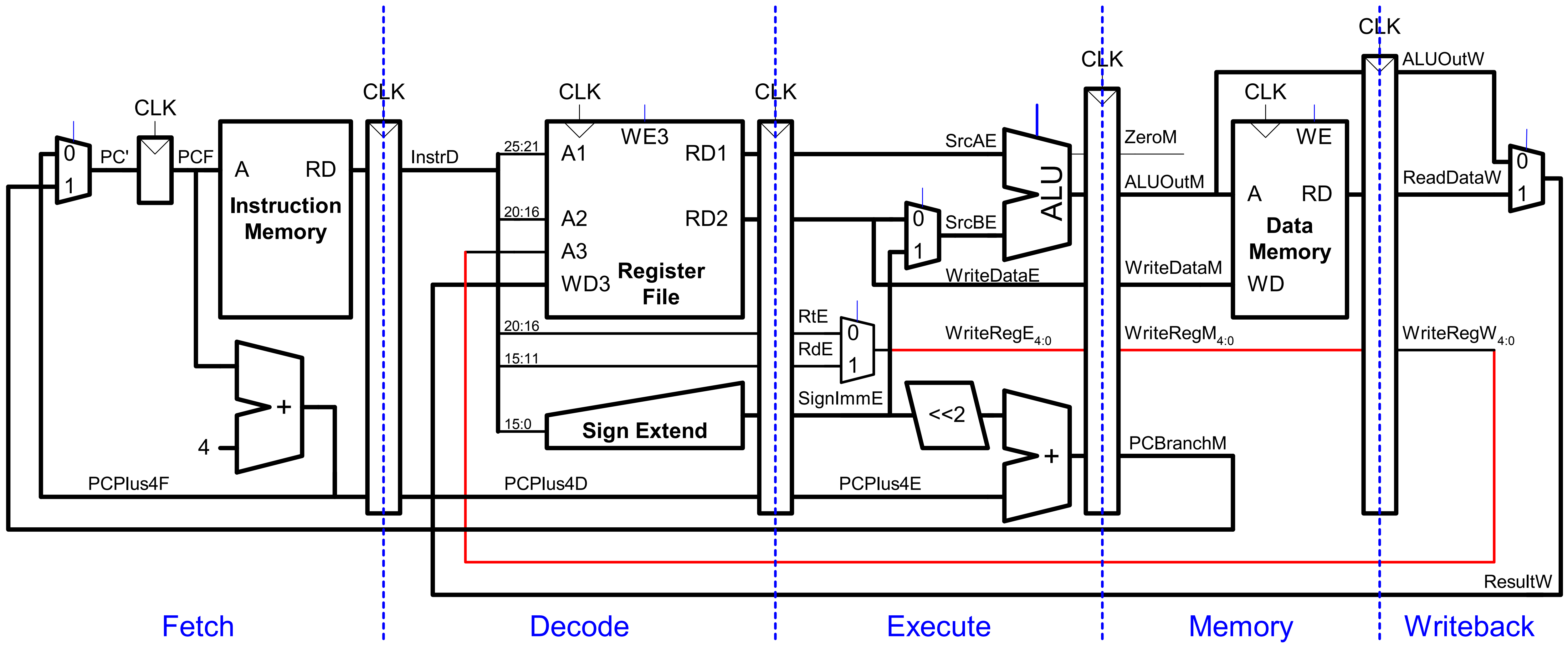


Single-Cycle & Pipelined Datapath



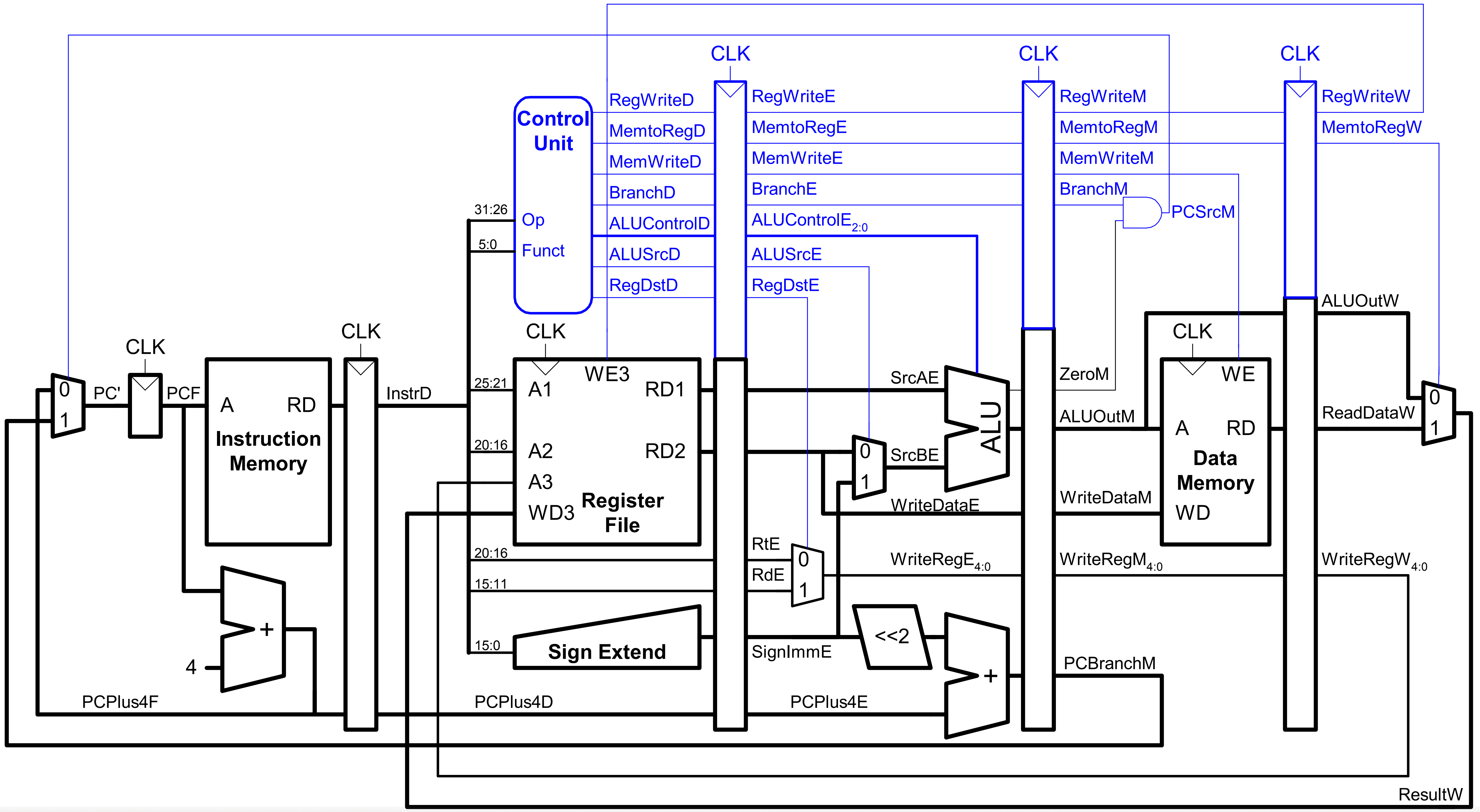


Corrected Pipelined Datapath



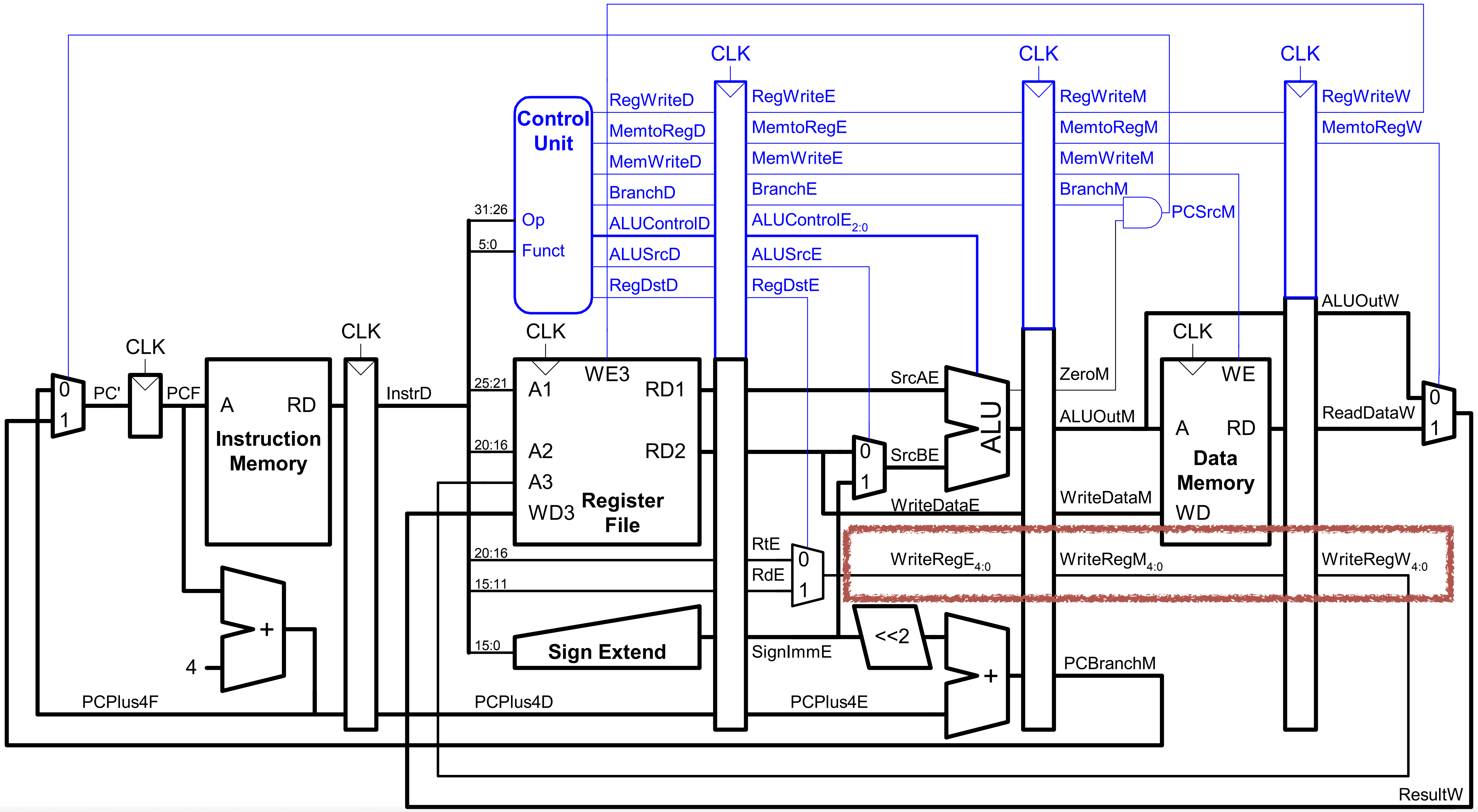
WriteRegE is also pipelined, synced with the *ResultW*

Pipelined Processor Control



Same control unit as single-cycle processor; Control signals must be pipelined along with the data, so as to remain synchronized

Pipelined Processor Control

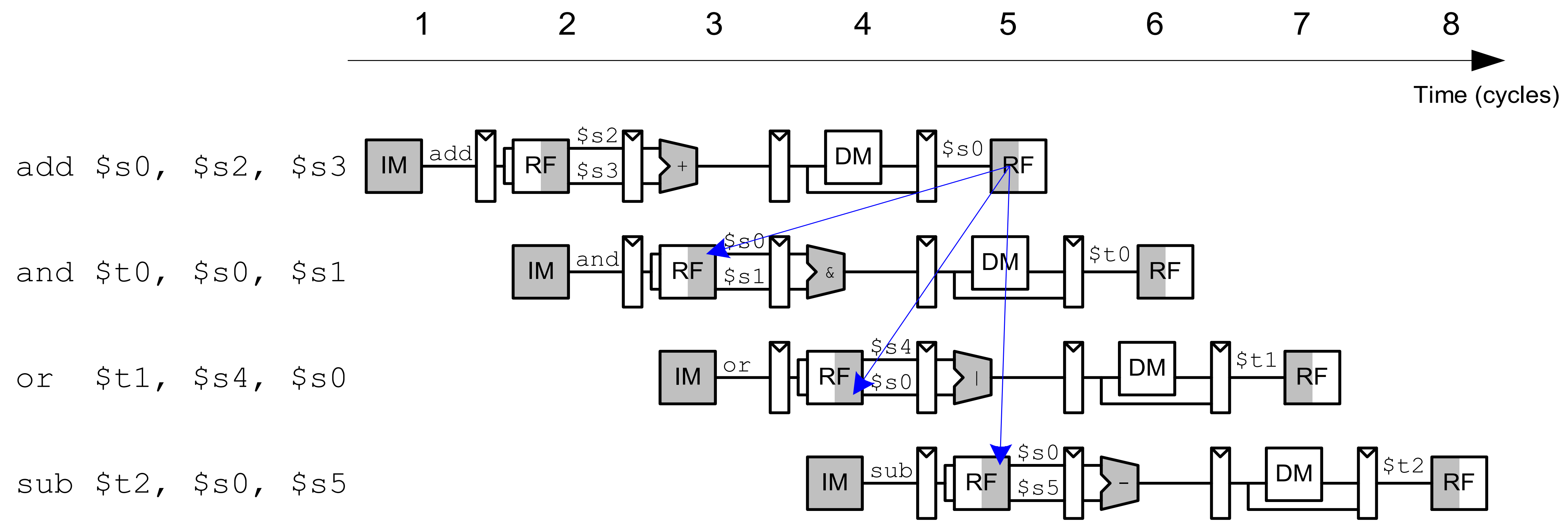


Same control unit as single-cycle processor; Control signals must be pipelined along with the data, so as to remain synchronized

Pipeline Hazards

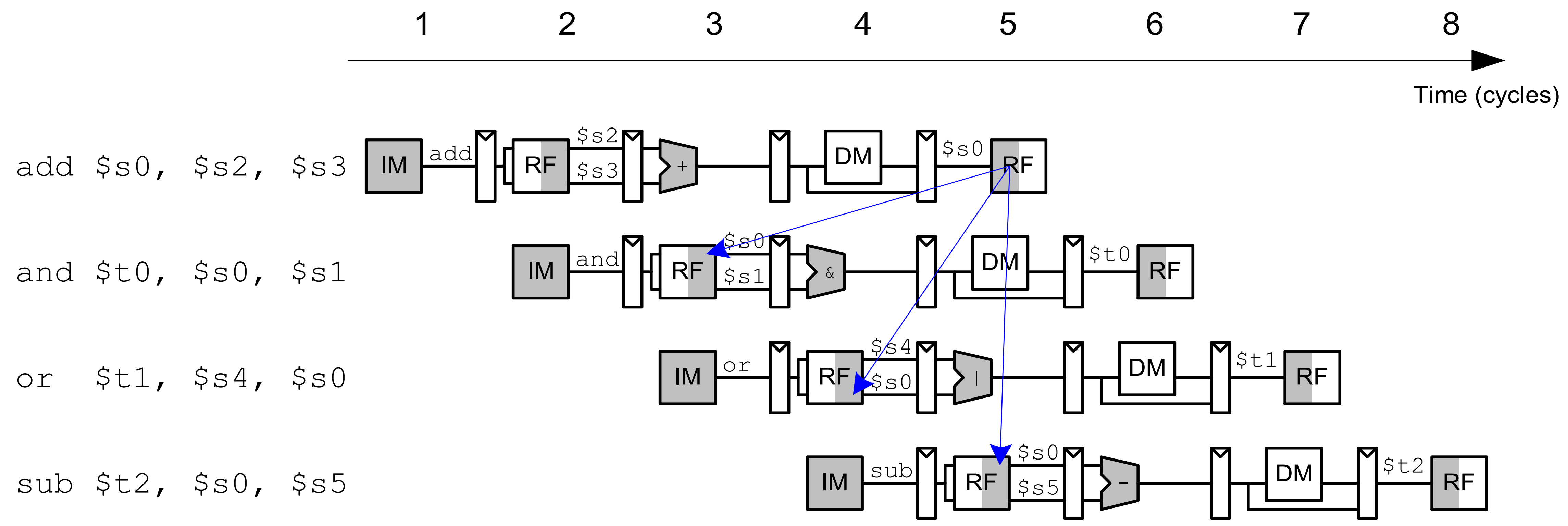
- An instruction depends on result from instruction that **hasn’t completed**
- Types:
  - **Data hazard:** register value not yet written back to register file
  - **Control hazard:** next instruction not decided yet (caused by branches)

Data Hazard



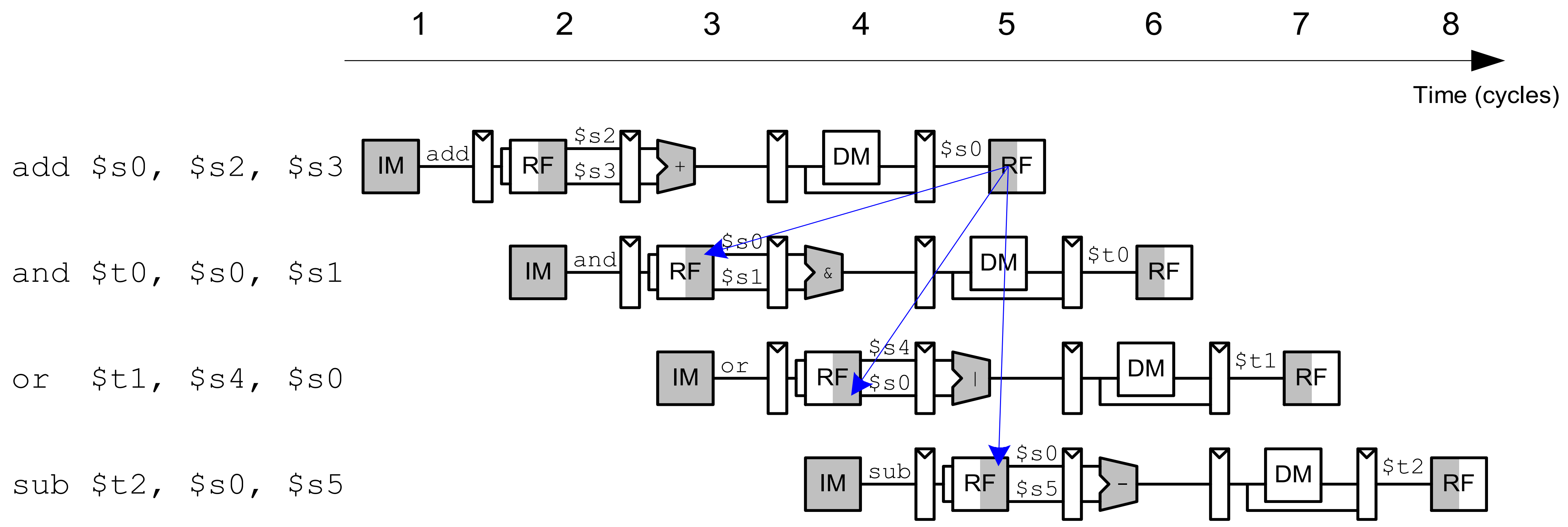
1
2
3
4
5
6
7
8
Time (cycles)
add $s0, $s2, $s3
and $t0, $s0, $s1
or $t1, $s4, $s0
sub $t2, $s0, $s5
IM
add
RF
$s2
$s3
+
DM
$s0
RF
and
$s0
$s1
&
$t0
or
$s4
|
$t1
sub
$s5
-
$t2

Data Hazard



RF is designed for high-speed reading/writing. The writing can be accomplished in the 1st half of a clock cycle, while the reading done in the 2nd half.

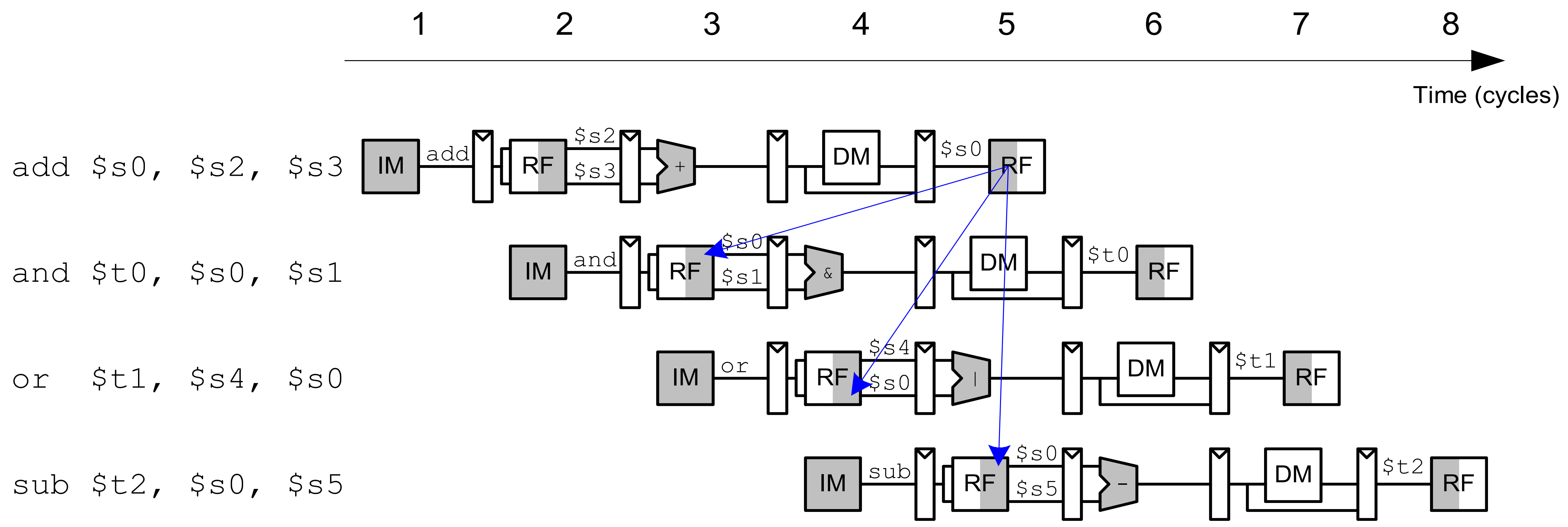
Data Hazard



1
2
3
4
5
6
7
8
Time (cycles)
add $s0, $s2, $s3
and $t0, $s0, $s1
or $t1, $s4, $s0
sub $t2, $s0, $s5
IM
add
RF
$s2
$s3
+
DM
$s0
RF
IM
and
RF
$s0
$s1
&
DM
$t0
RF
IM
or
RF
$s4
$s0
|
DM
$t1
RF
IM
sub
RF
$s0
$s5
-
DM
$t2
RF

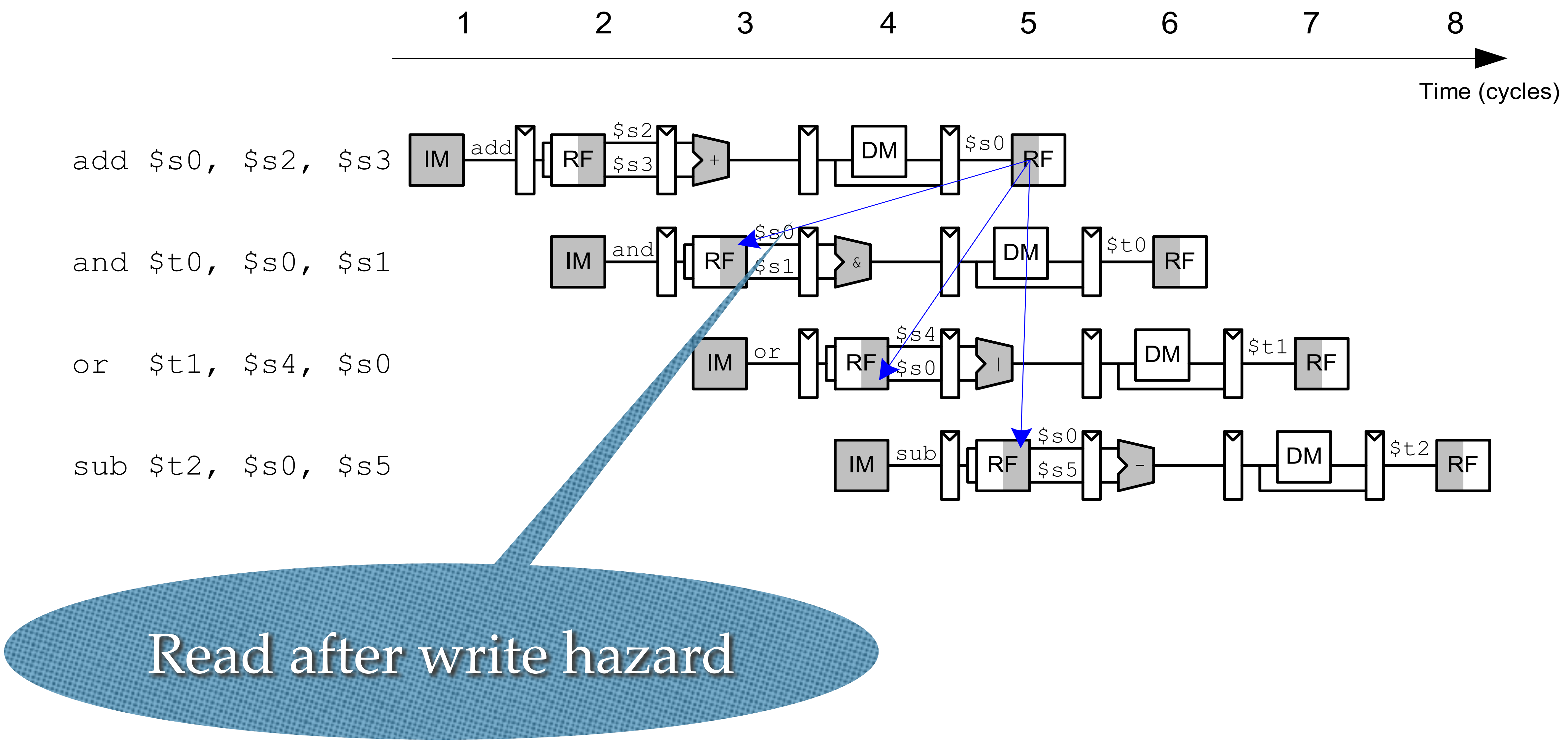
Data Hazard

How many data hazards? How many wrong results?



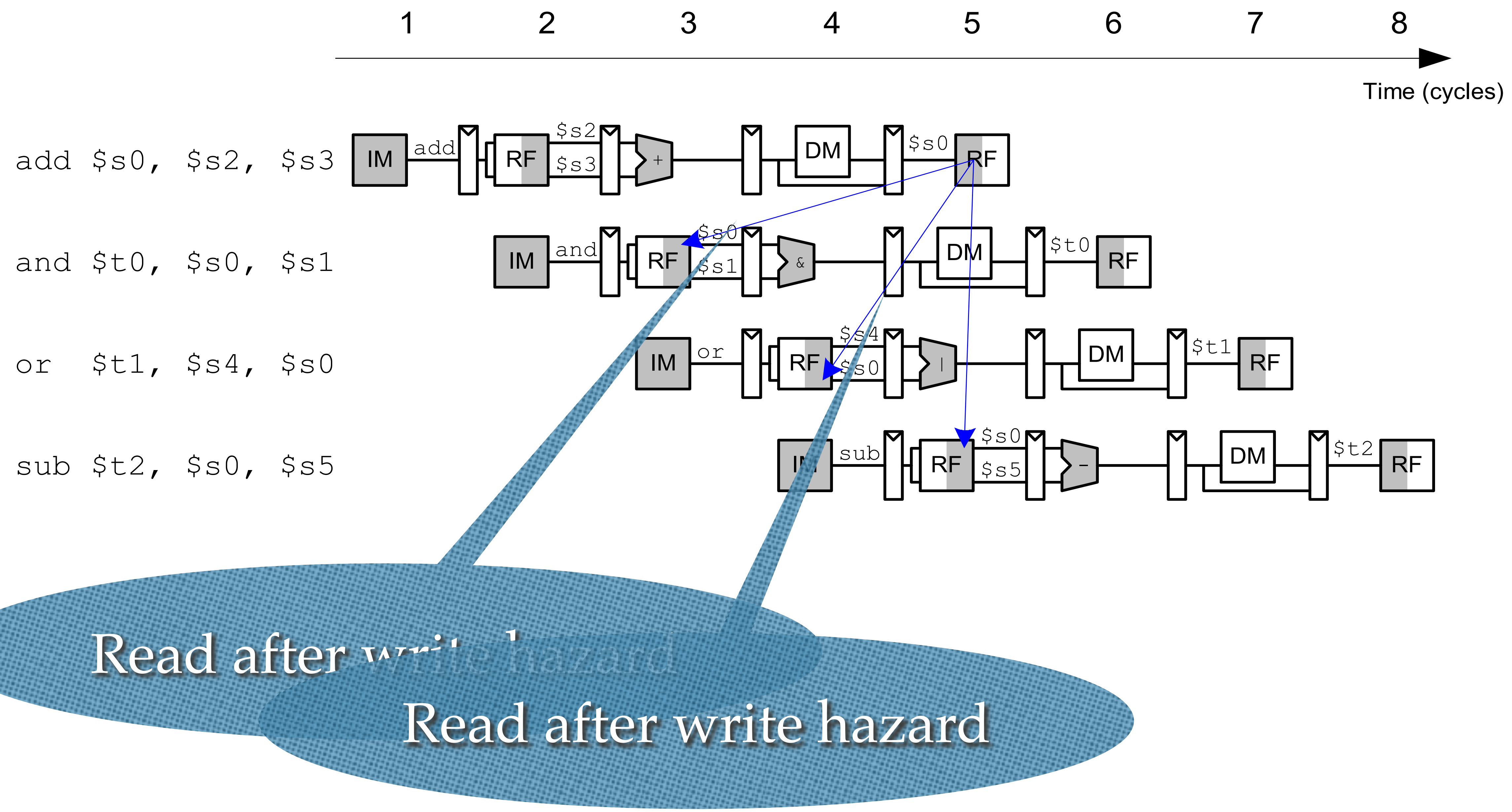
Data Hazard

How many data hazards? How many wrong results?



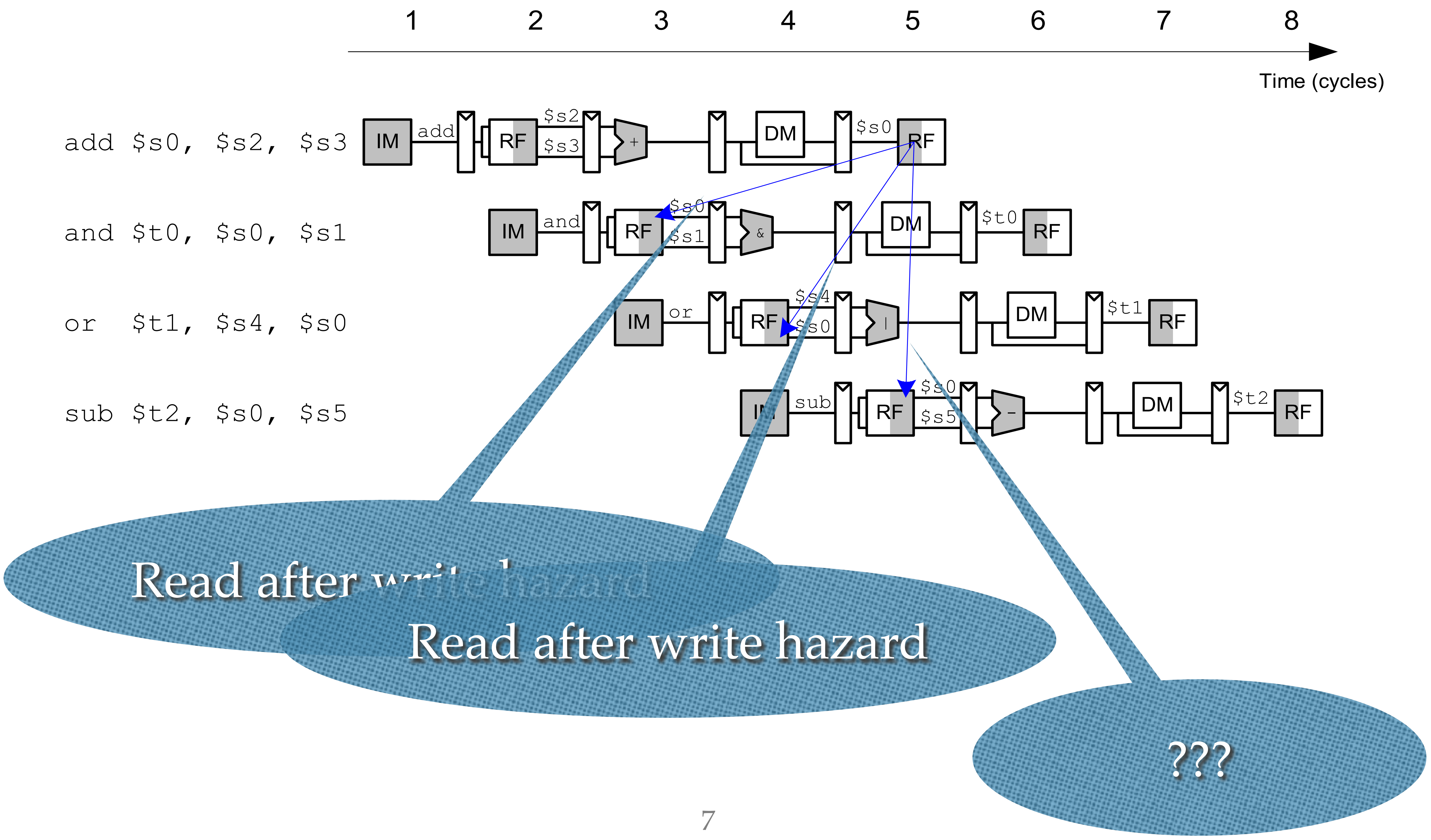
Data Hazard

How many data hazards? How many wrong results?



Data Hazard

How many data hazards? How many wrong results?

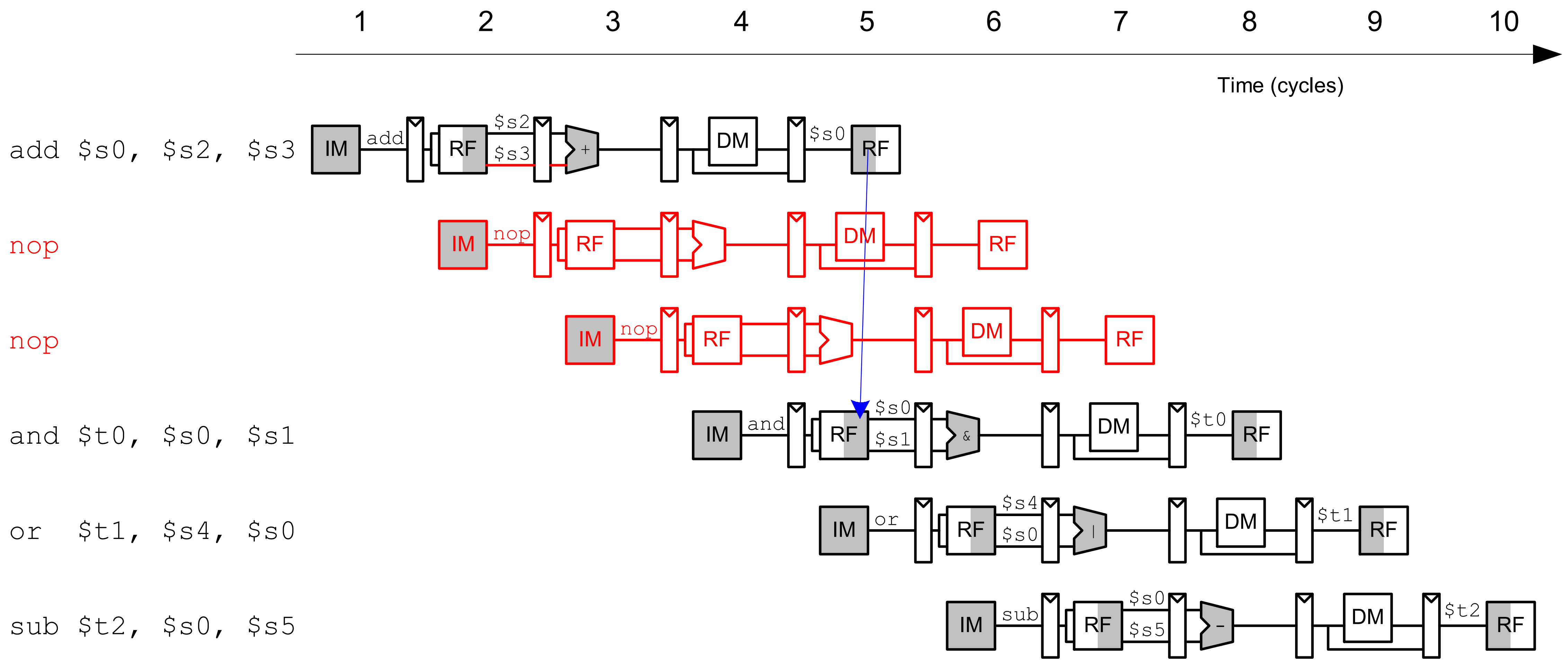


Handling Data Hazards

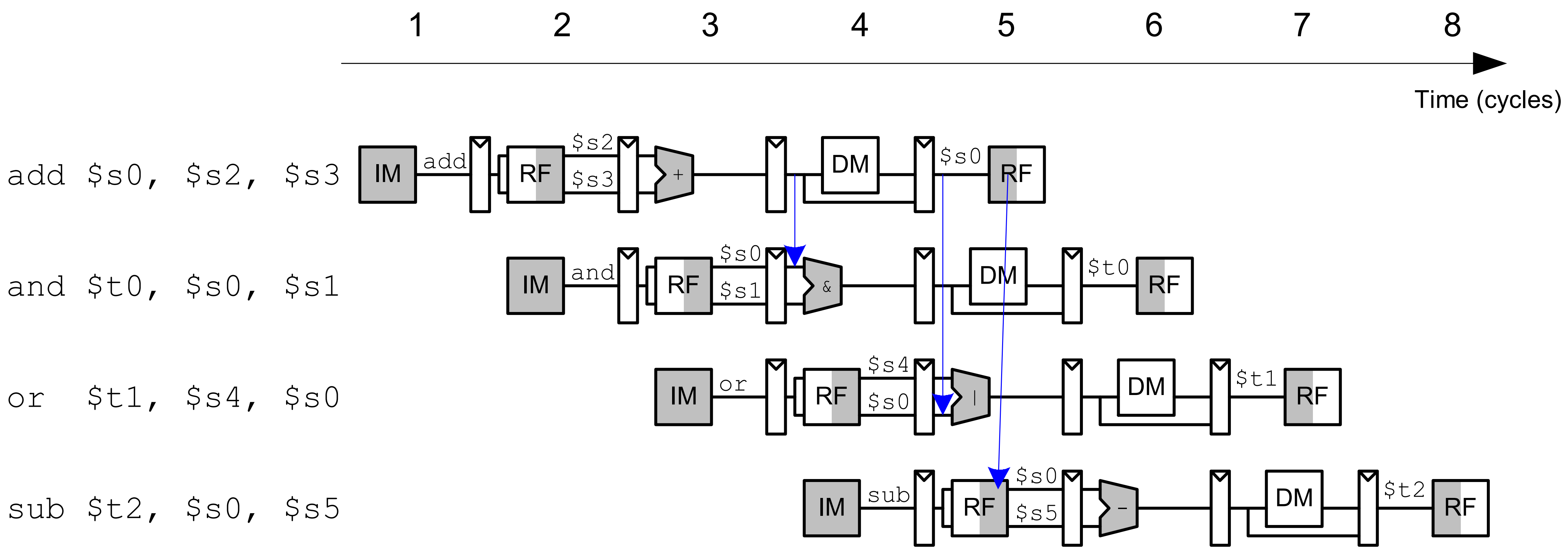
- Insert `nops` in code at compile time
- Rearrange code at compile time
- Forward data at run time
- Stall the processor at run time

Compile-Time Hazard Elimination

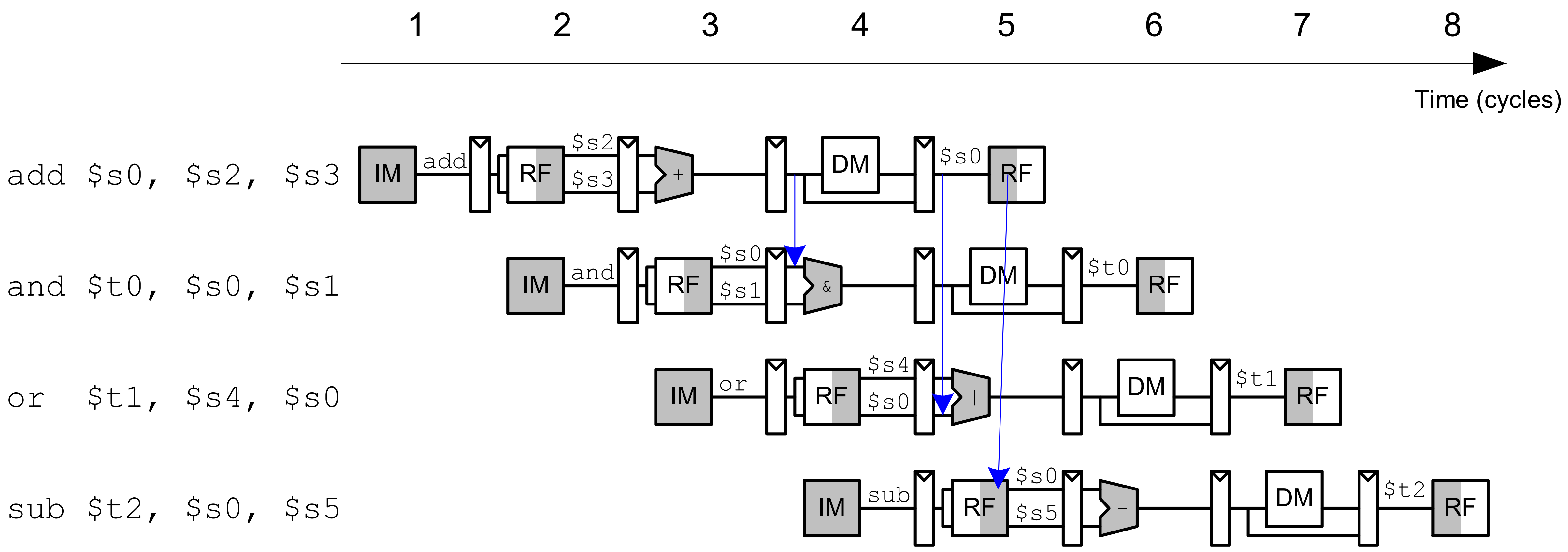
- Insert enough `nops` for result to be ready
- Or move independent useful instructions forward



Data Forwarding

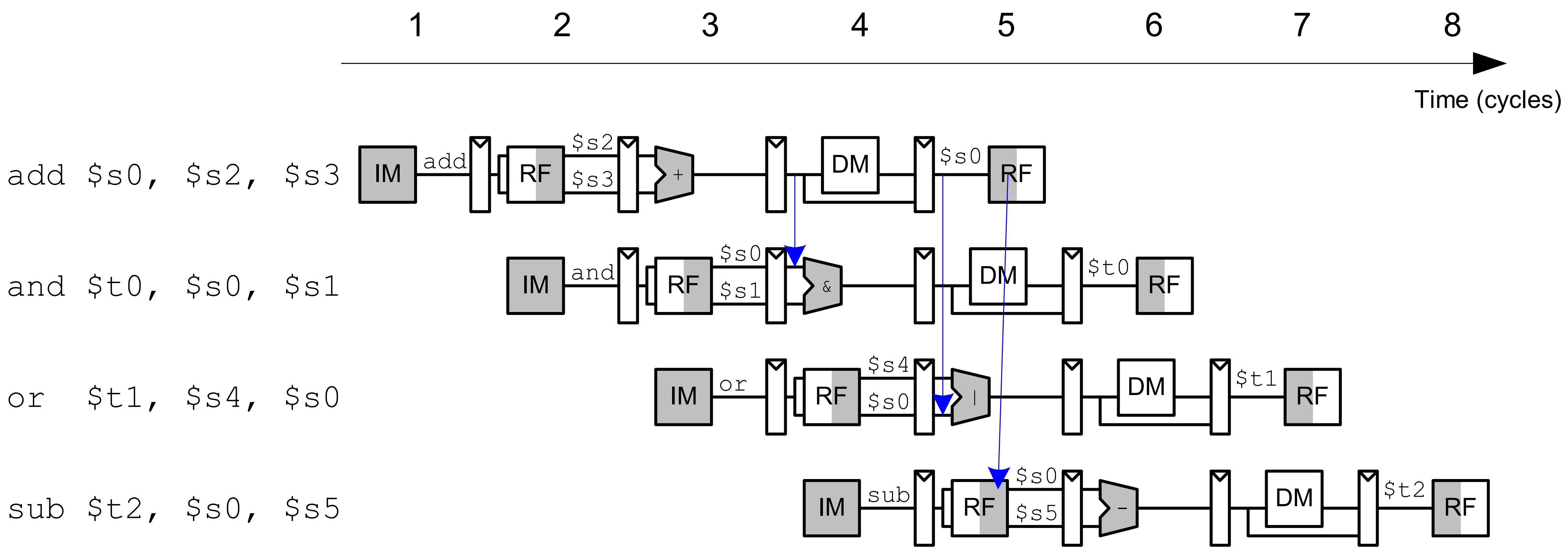


Data Forwarding



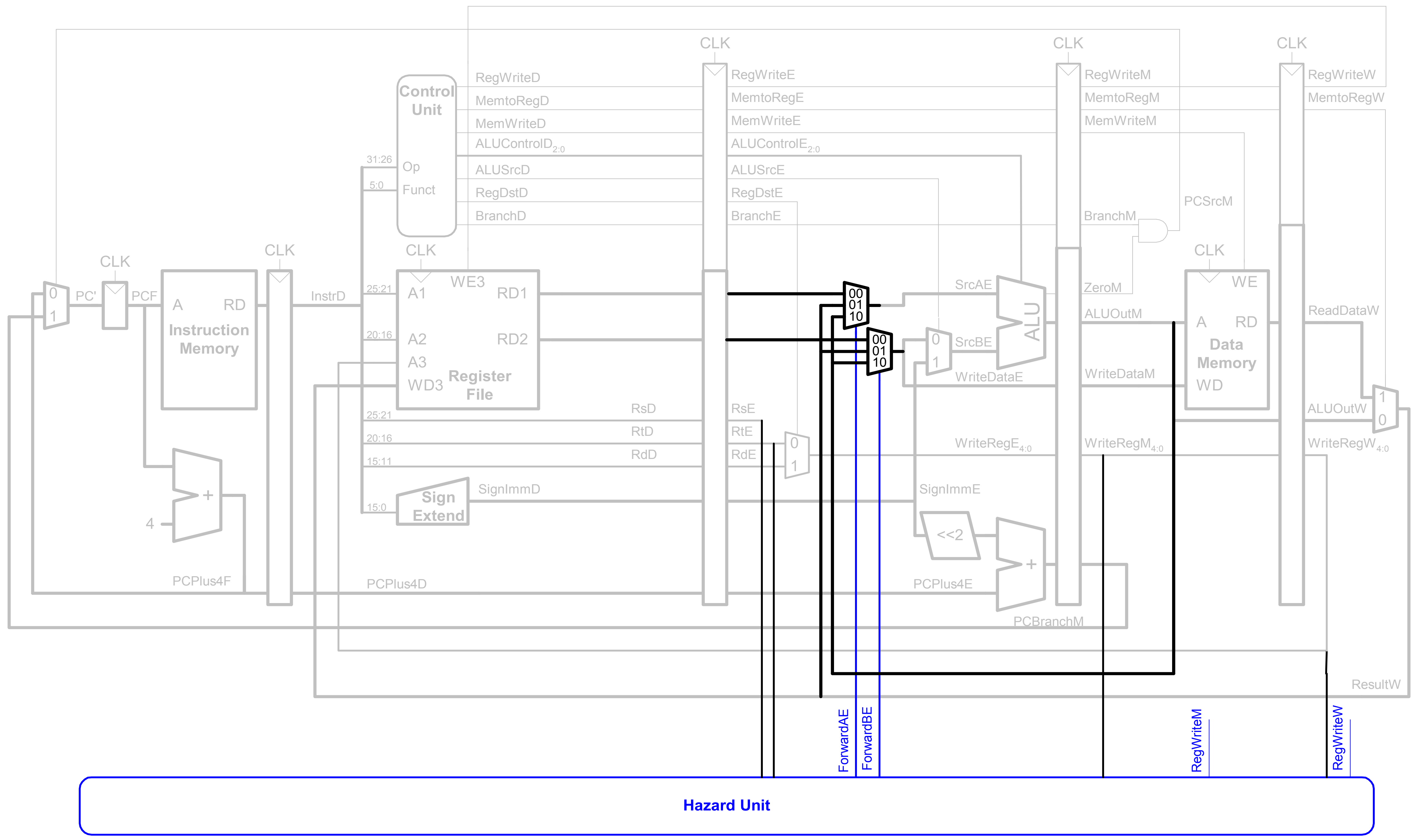
- Forwarding results from Mem or Writeback stage, **directly to the execute stage of a dependent instruction**

Data Forwarding

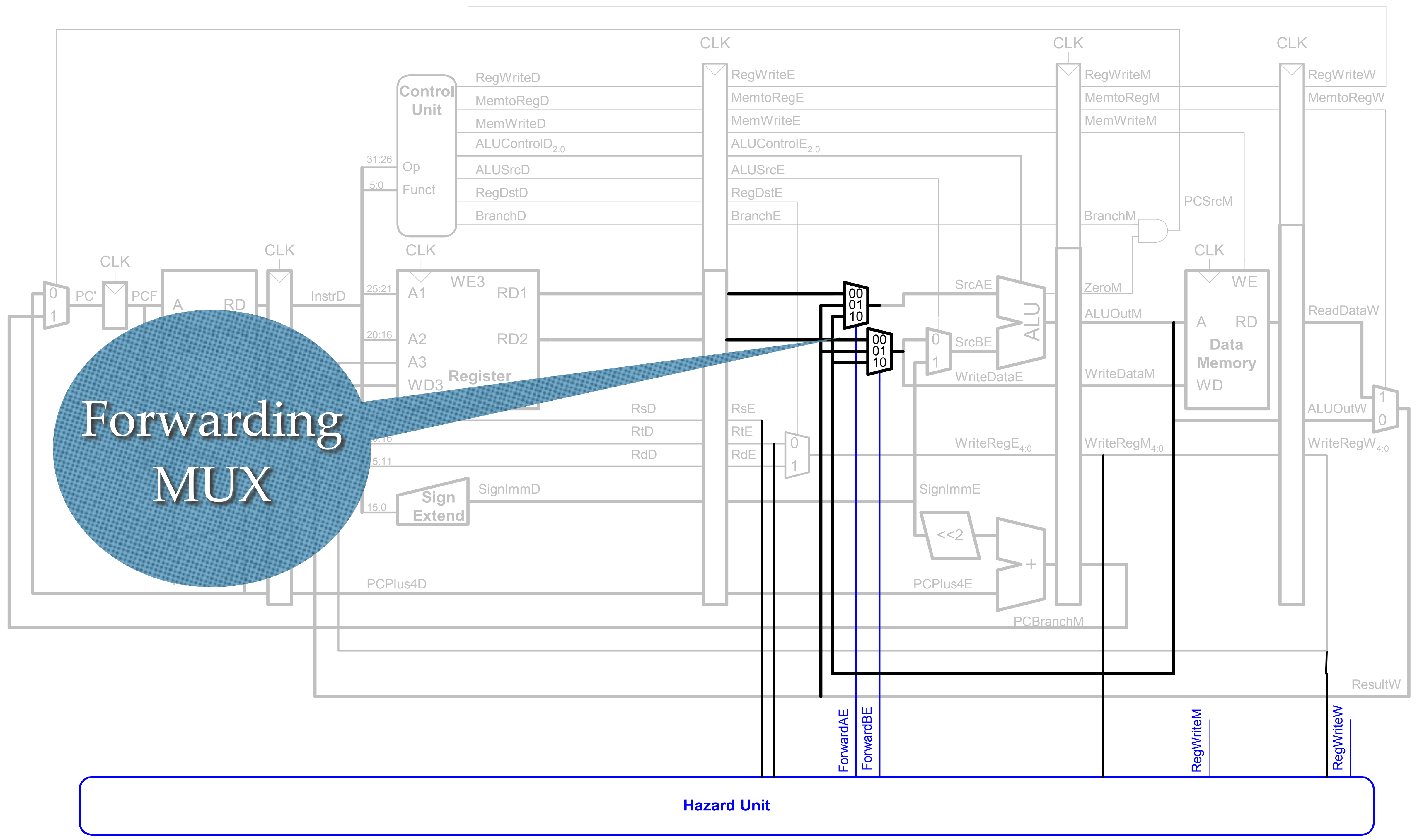


- Forwarding results from Mem or Writeback stage, **directly to the execute stage of a dependent instruction**
- What will be needed?

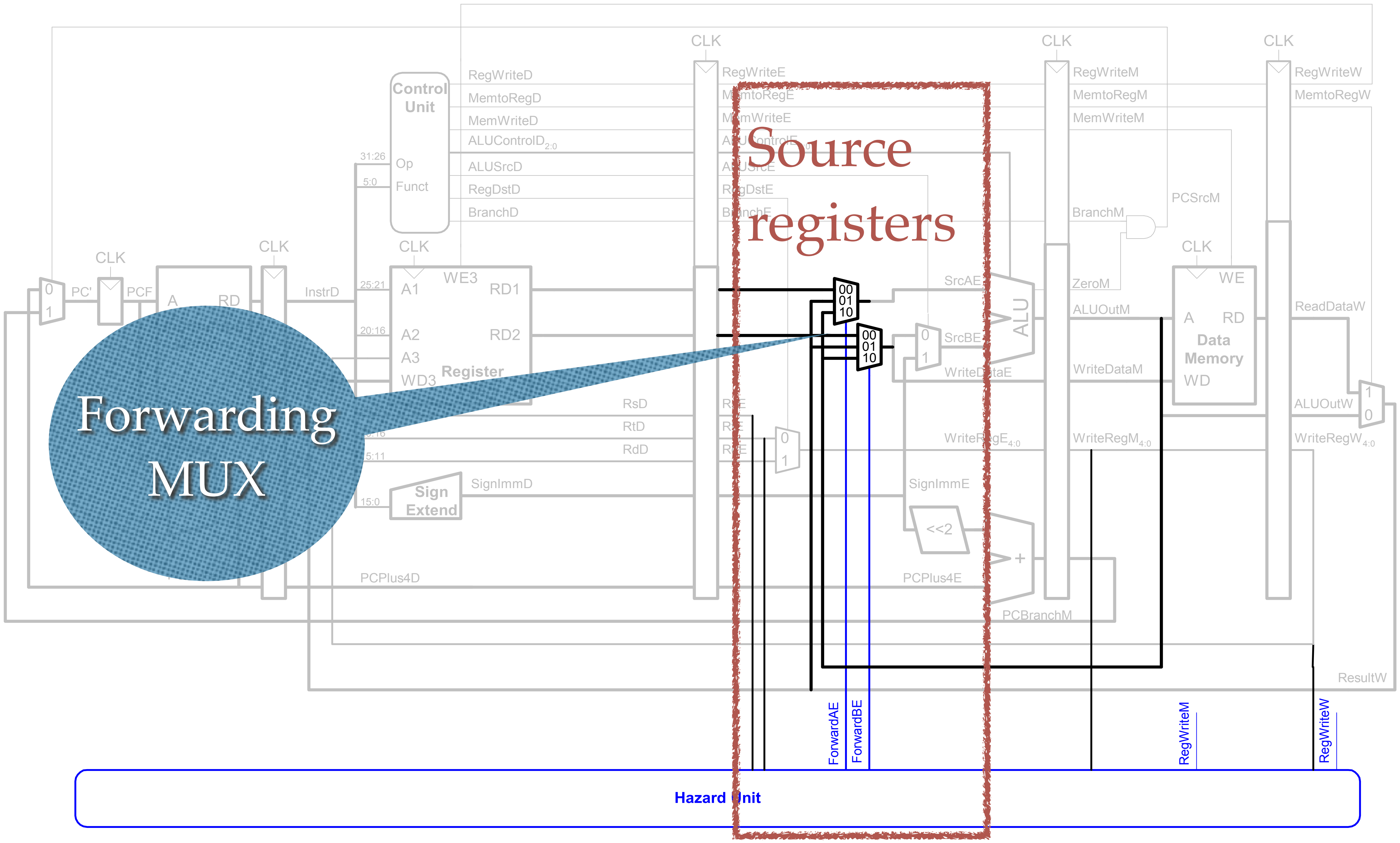
Data Forwarding



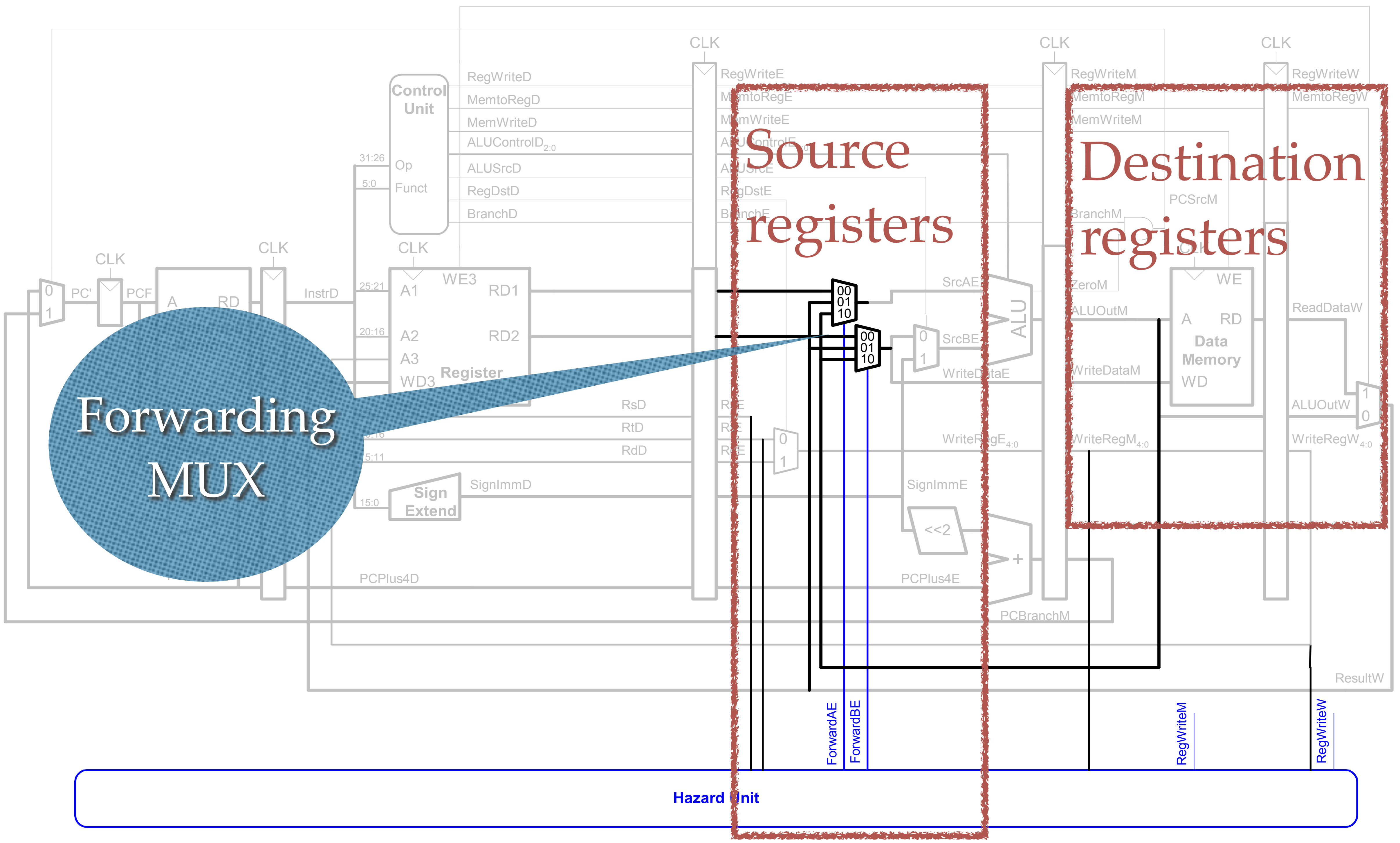
Data Forwarding



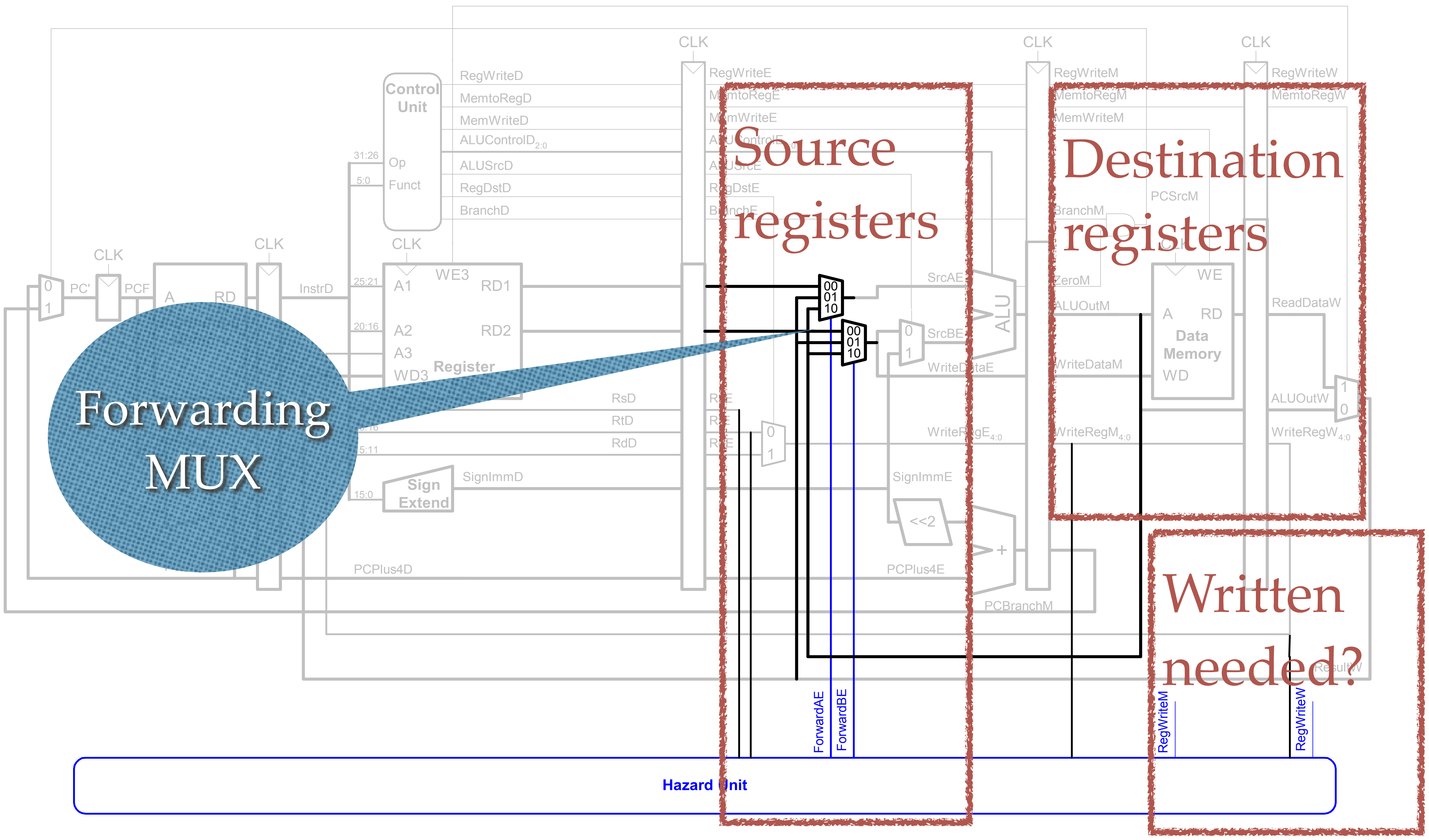
Data Forwarding



Data Forwarding



Data Forwarding



Hazard Detection Unit

- Computes control signals, to serve the forwarding MUXes

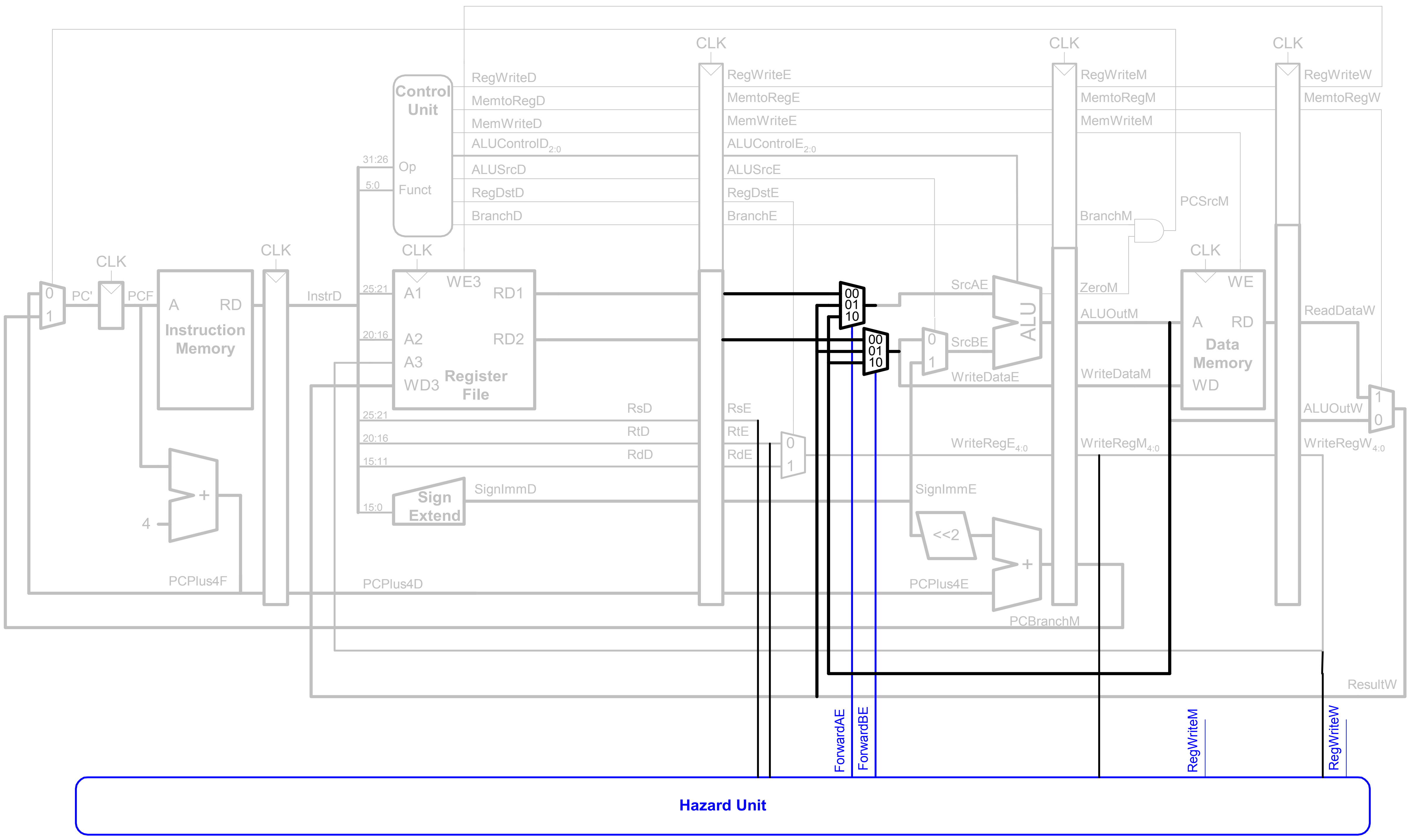
Hazard Detection Unit

- Computes control signals, to serve the forwarding MUXes
  - A forward **should happen**, if a specific stage will write data to a destination reg., which is used as the source reg. by a dependent instruction

Hazard Detection Unit

- Computes control signals, to serve the forwarding MUXes
    - A forward **should happen**, if a specific stage will write data to a destination reg., which is used as the source reg. by a dependent instruction
- What if both Mem. and Writeback have the same matching destination reg.?

Hazard Detection Unit



Data Forwarding

- Forward to Execute stage from either:
  - Memory stage or
  - Writeback stage
- Forwarding logic for *ForwardAE*:

```
if        ((rsE != 0) AND (rsE == WriteRegM) AND RegWriteM)
 then   ForwardAE = 10
else if ((rsE != 0) AND (rsE == WriteRegW) AND RegWriteW)
 then   ForwardAE = 01
else              ForwardAE = 00
```

Forwarding logic for *ForwardBE* same, but replace *rsE* with *rtE*

What Left in Data Hazard

- All instructions that have “read after write” (RAW) data hazard can be secured
  - Forwarding
  - Hazard detection unit

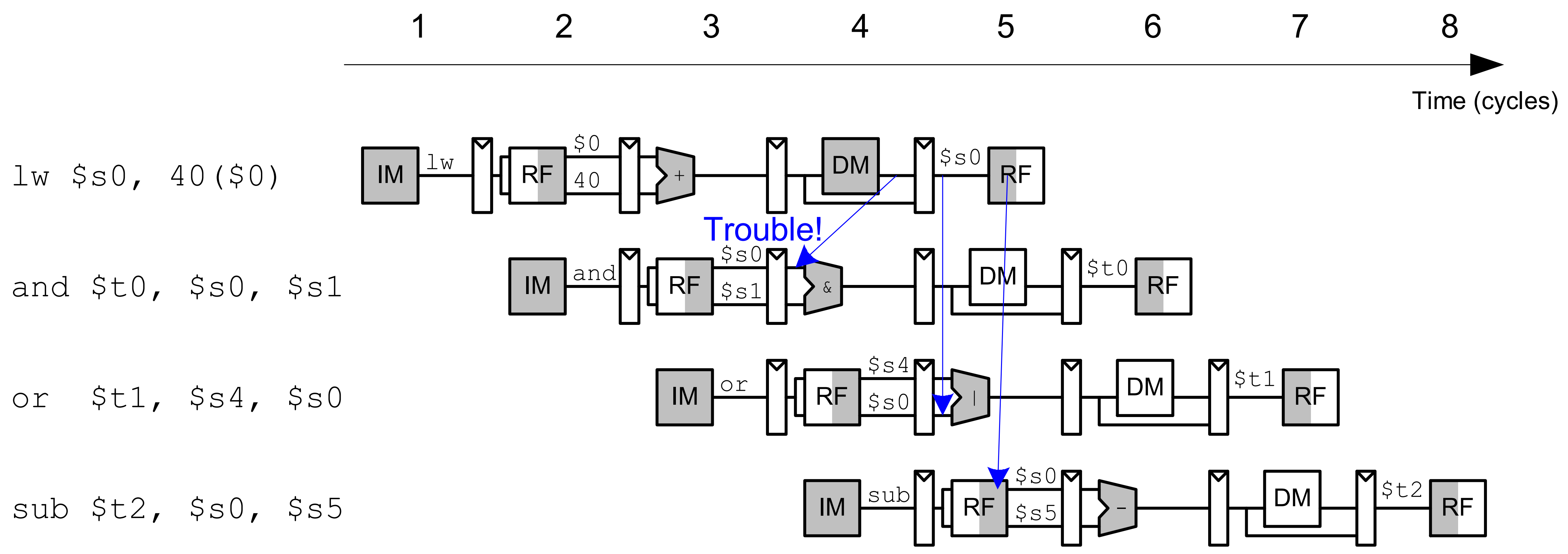
What Left in Data Hazard

- All instructions that have “read after write” (RAW) data hazard can be secured
  - Forwarding
  - Hazard detection unit
- What else?

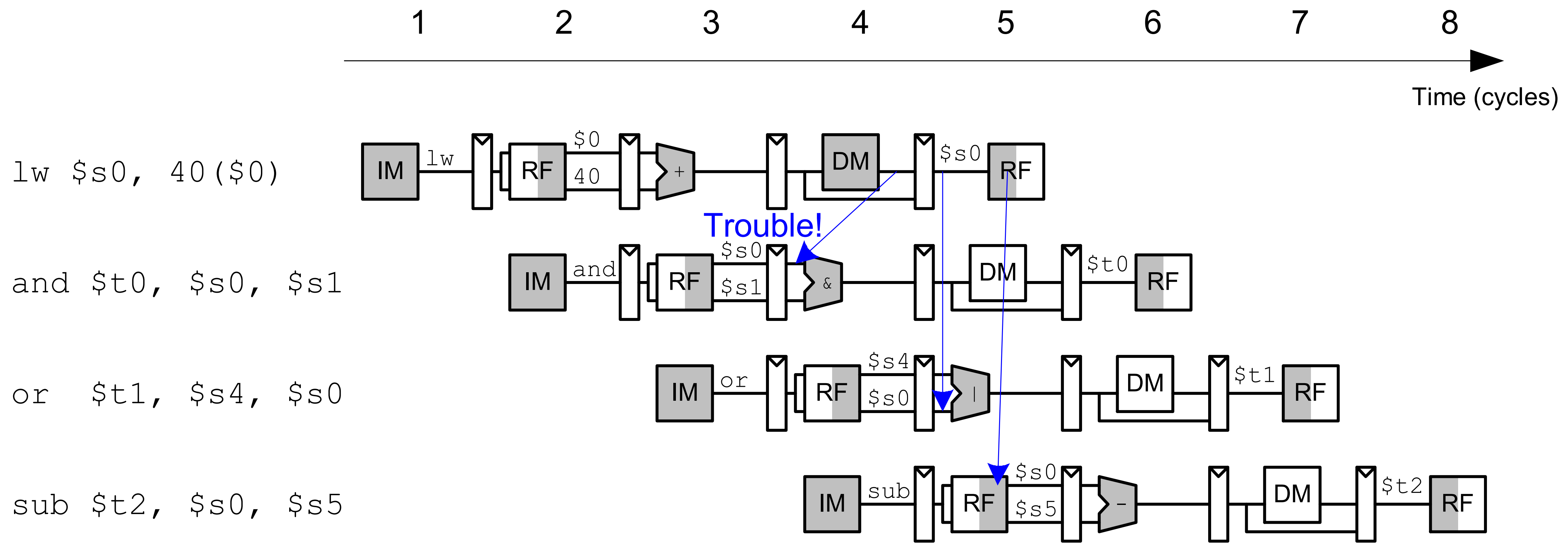
What Left in Data Hazard

- All instructions that have “read after write” (RAW) data hazard can be secured
    - Forwarding
    - Hazard detection unit
- What else?
- The “slowest” MIPS instruction

Stalling

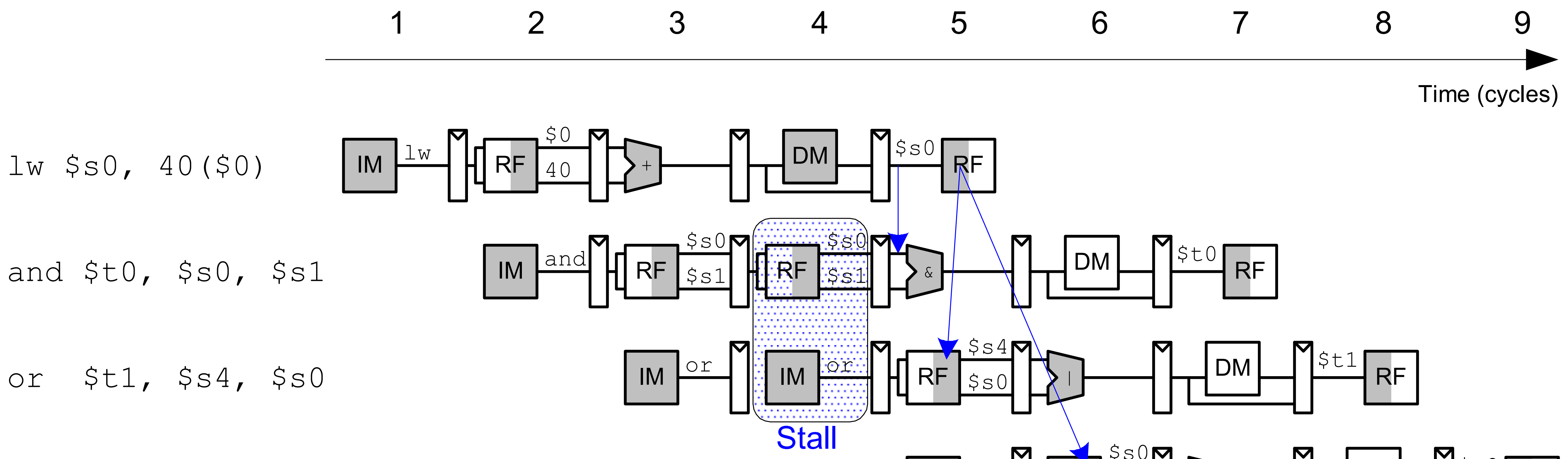

1
2
3
4
5
6
7
8
Time (cycles)
lw $s0, 40($0)
and $t0, $s0, $s1
or $t1, $s4, $s0
sub $t2, $s0, $s5
IM
lw
RF
$0
40
+
DM
$s0
RF
Trouble!
IM
and
RF
$s0
$s1
&
DM
$t0
RF
IM
or
RF
$s4
$s0
|
DM
$t1
RF
IM
sub
RF
$s0
$s5
-
DM
$t2
RF

Stalling



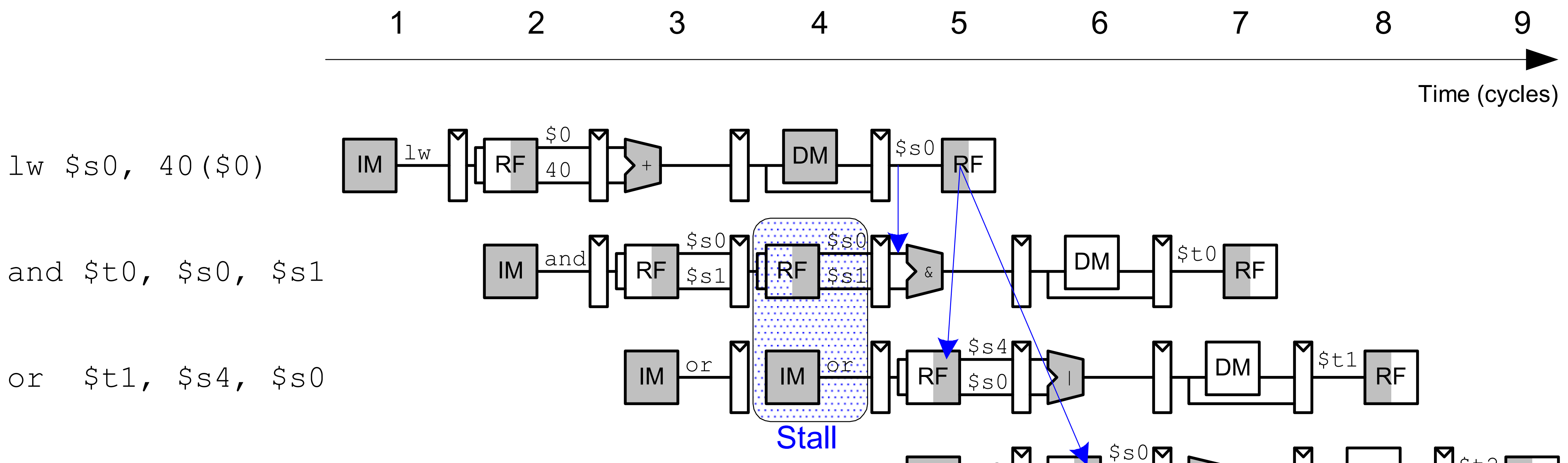
- *lw* receives data ($s0) **at the end of cycle 4**
- *and* needs this data **at the beginning of cycle 4**

Stalling



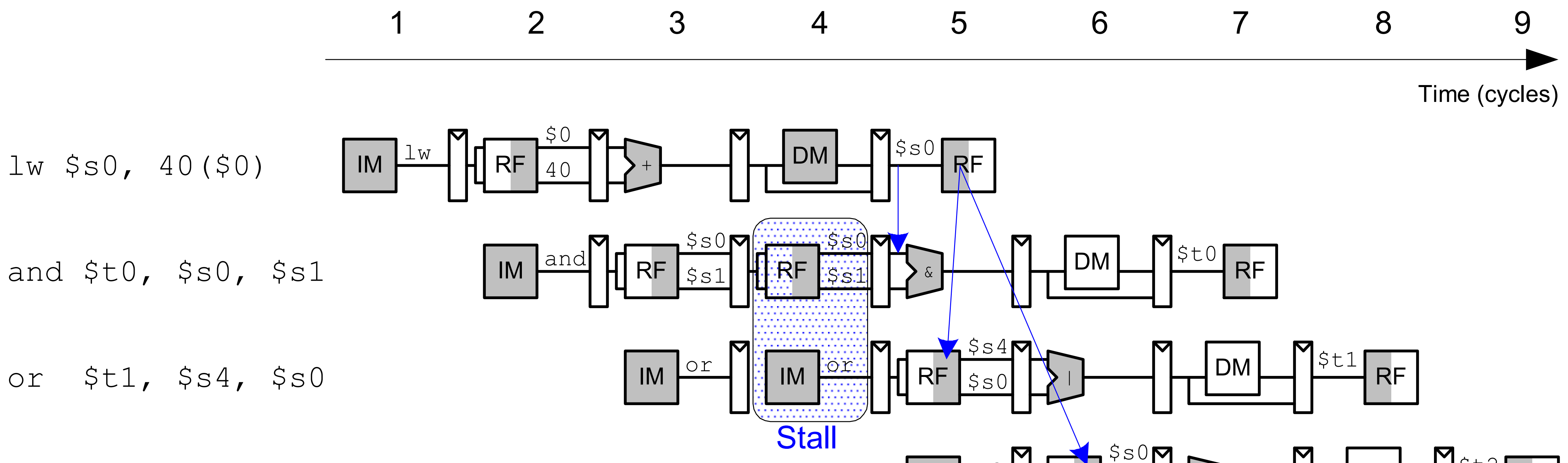
- Holding up the dependent —"and" operation, until the data becomes available!
  - *and* enters Decode in cycle 3 and stays there in cycle 4
  - *or* should remain in the instruction Fetch stage as well — during cycle 3 and 4

Stalling



- In Cycle 4 —> Execution unused
- In Cycle 5 —> Data Mem. unused
- In Cycle 6 —> Writeback unused

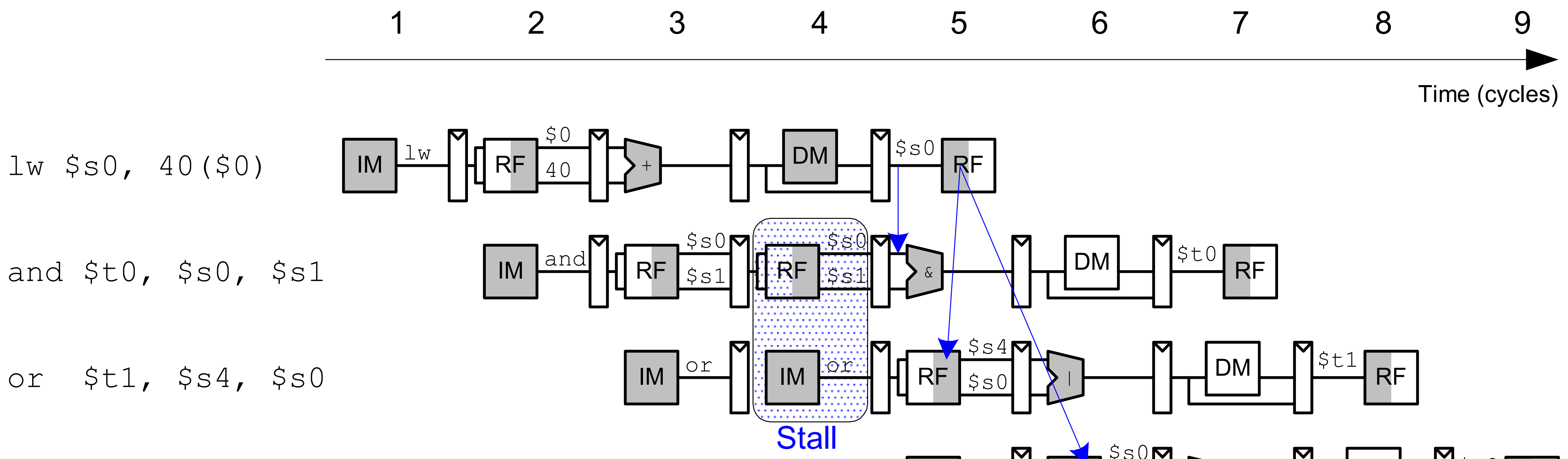
Stalling



- In Cycle 4 —> Execution unused
- In Cycle 5 —> Data Mem. unused
- In Cycle 6 —> Writeback unused

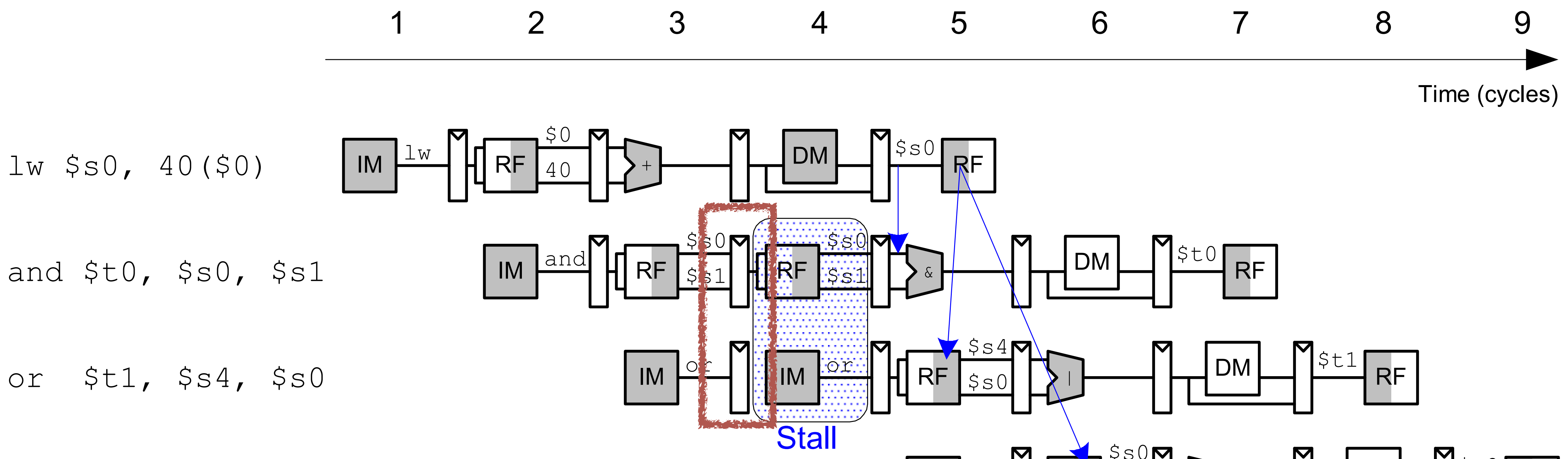
Bubble

Stalling



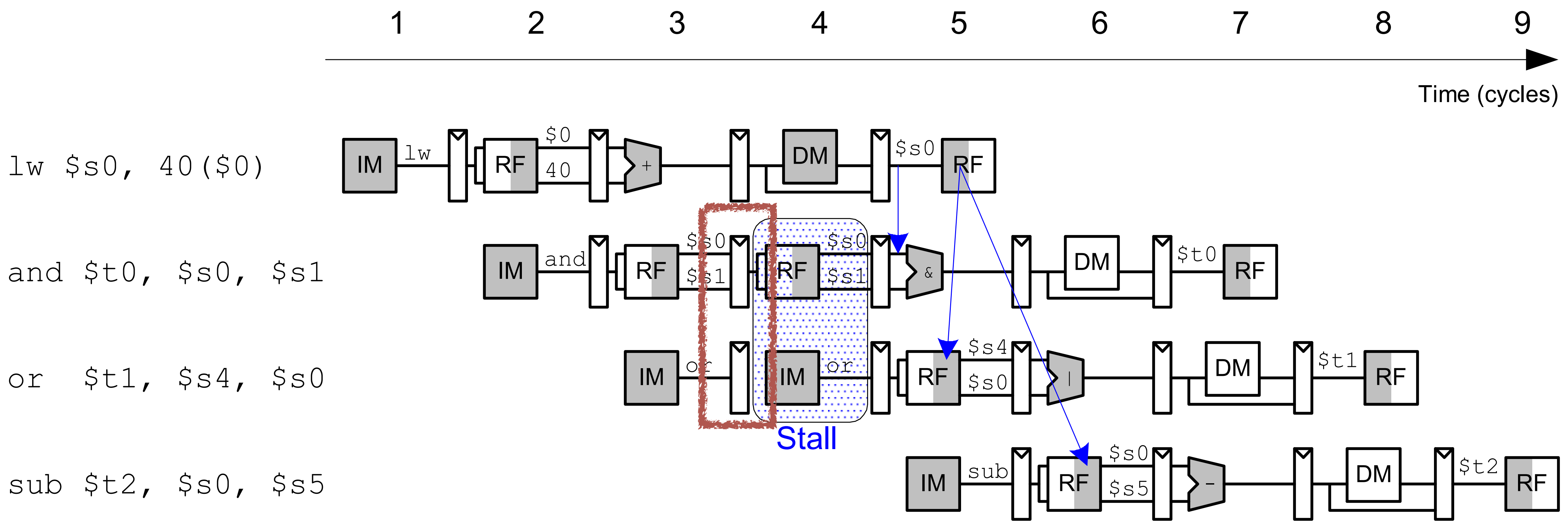
❖ Hardware realization?

Stalling



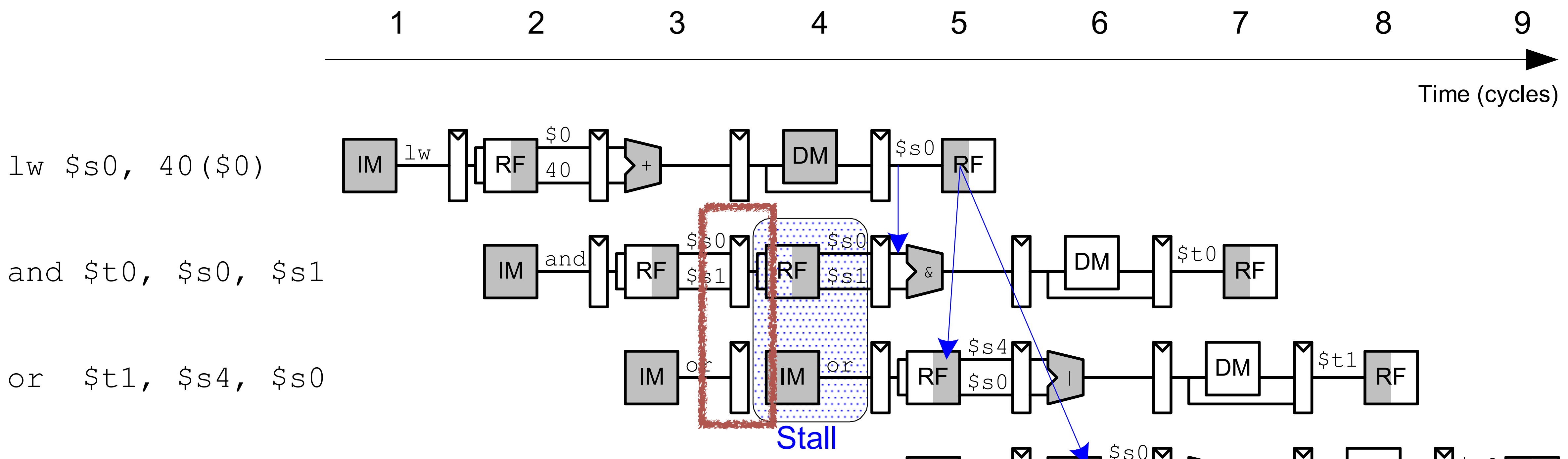
- Hardware realization?

Stalling



- Hardware realization?
  - Stage register —> disable it!

Stalling

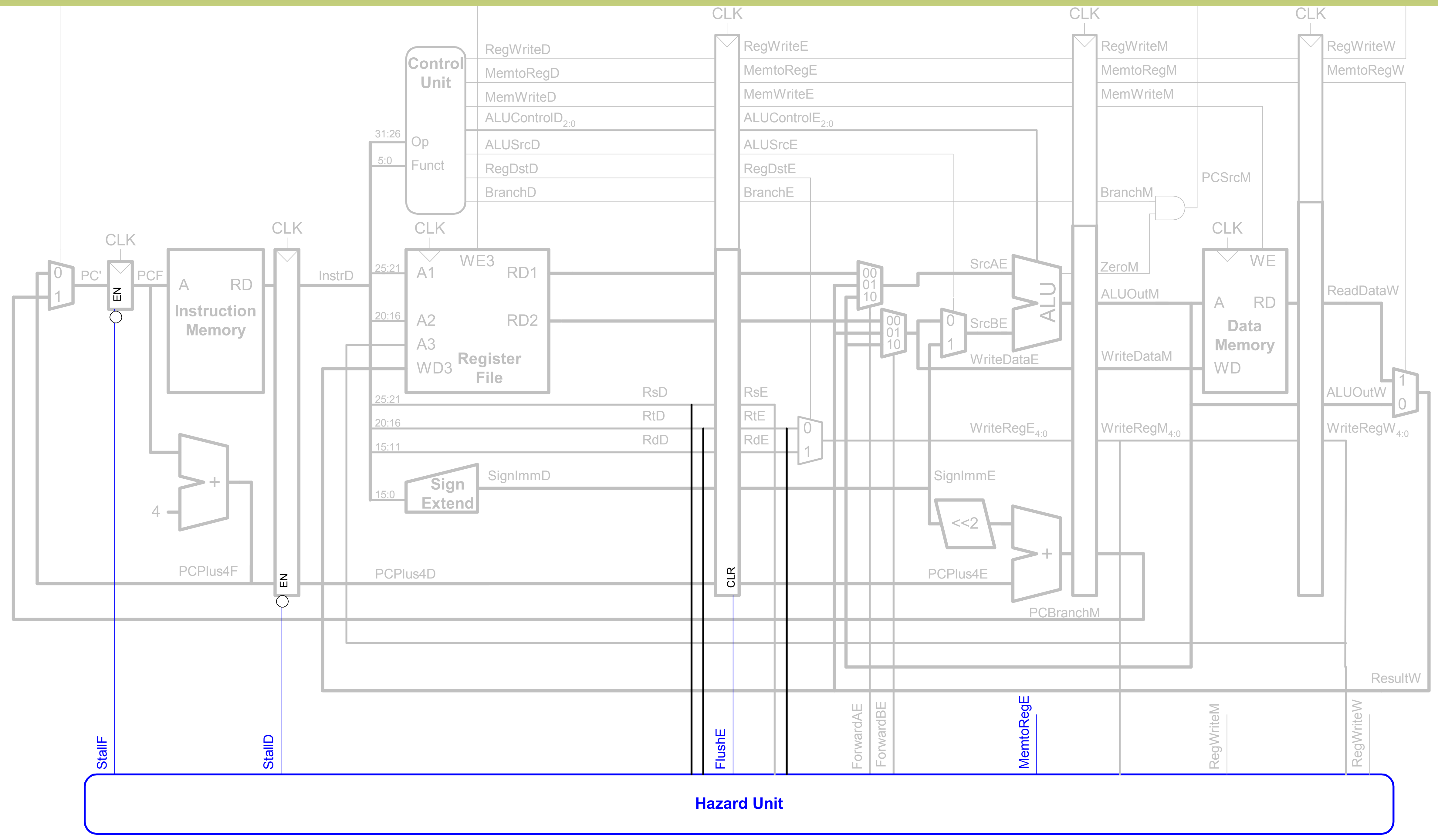


- Hardware realization?
    - Stage register —> disable it!
    - A stage is stalled, all previous stages should also be stalled!

Stalling Hardware

Step1: The hazard unit will detect: *lw* **in the Exe. stage**

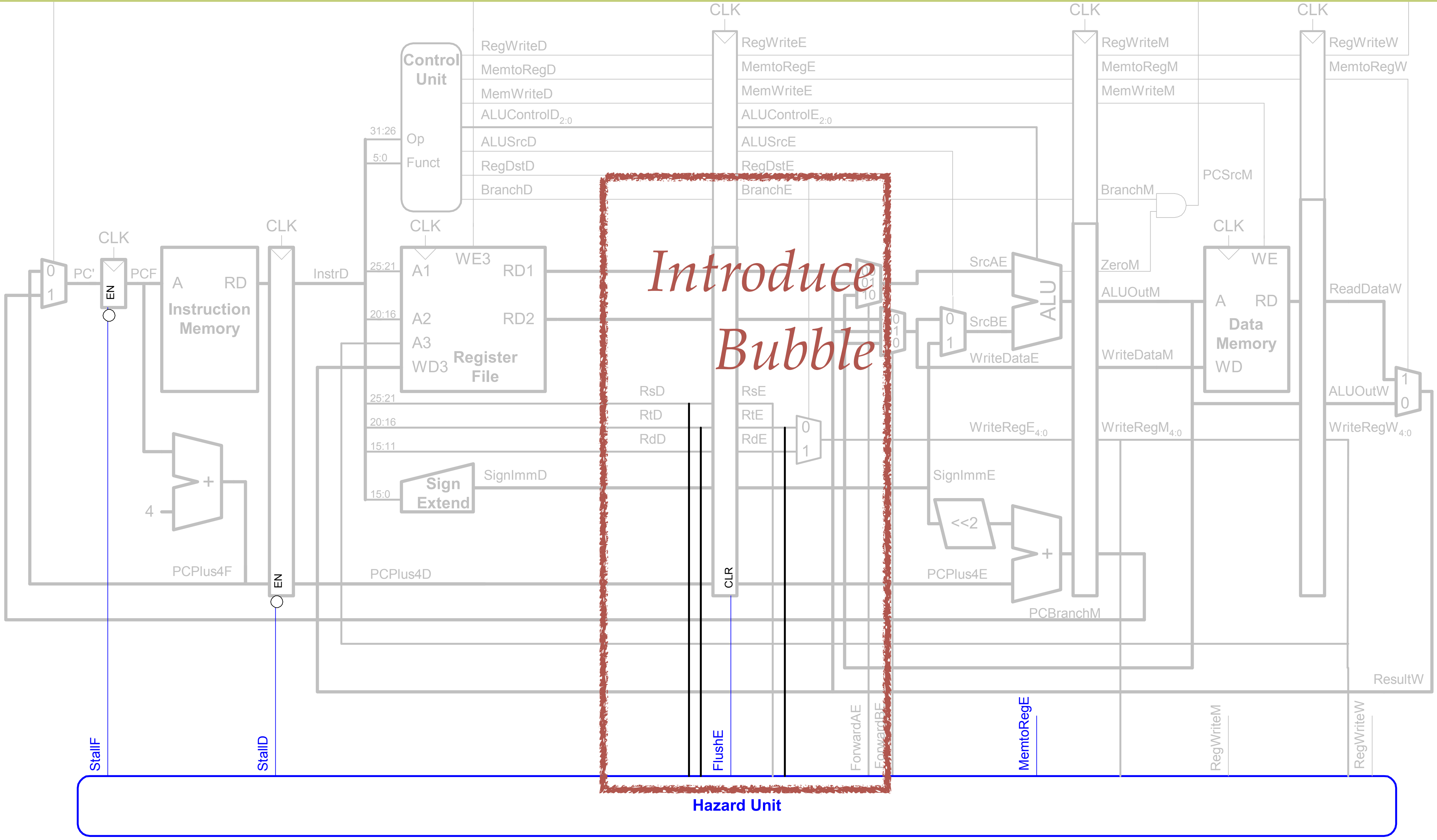
Step2: If yes, and destination reg. (rtE) == source operand (rsD/rtD)



Stalling Hardware

Step1: The hazard unit will detect: *lw* **in the Exe. stage**

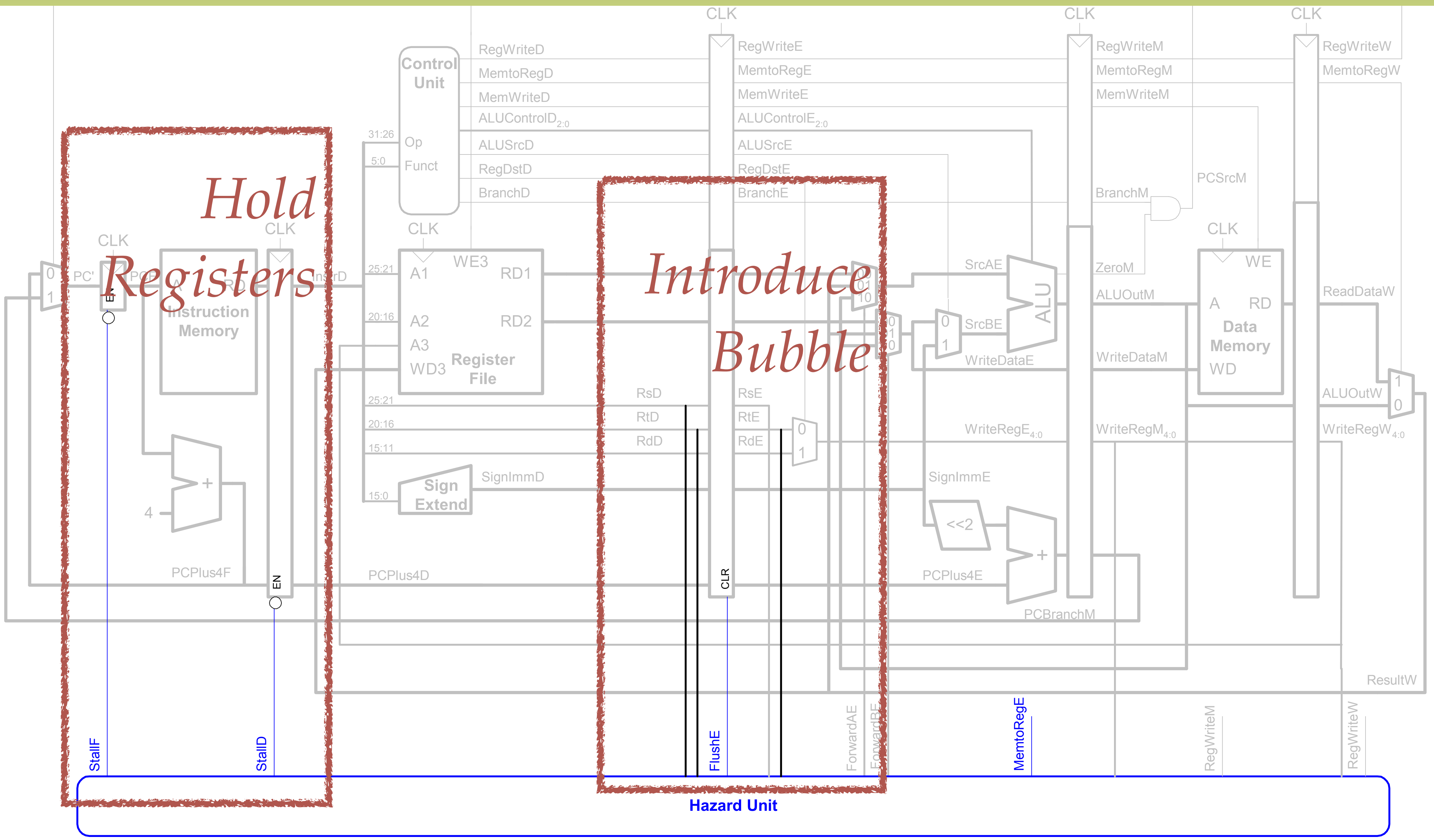
Step2: If yes, and destination reg. (rtE) == source operand (rsD / rtD)



Stalling Hardware

Step1: The hazard unit will detect: *lw* **in the Exe. stage**

Step2: If yes, and destination reg. (rtE) == source operand (rsD/rtD)



Stalling Logic

```
lwstall =
   ((rsD==rtE) OR (rtD==rtE)) AND MemtoRegE

StallF = StallD = FlushE = lwstall
```