CSEE 3827: Fundamentals of Computer Systems, Spring 2011

9. Pipelined MIPS Processor

Prof. Martha Kim (martha@cs.columbia.edu)
Web: http://www.cs.columbia.edu/~martha/courses/3827/sp11/

Outline (H&H 7.5)

- Pipelined MIPS processor
- Pipelined Performance

Single-Cycle CPU Performance Issues

- Longest delay determines clock period
  - Critical path: load instruction
  - instruction memory → register file → ALU → data memory → register file
- Not feasible to vary clock period for different instructions
  - A multicycle implementation would solve this (See H&H 7.4)
- We will improve performance by pipelining

Pipelining Laundry Analogy

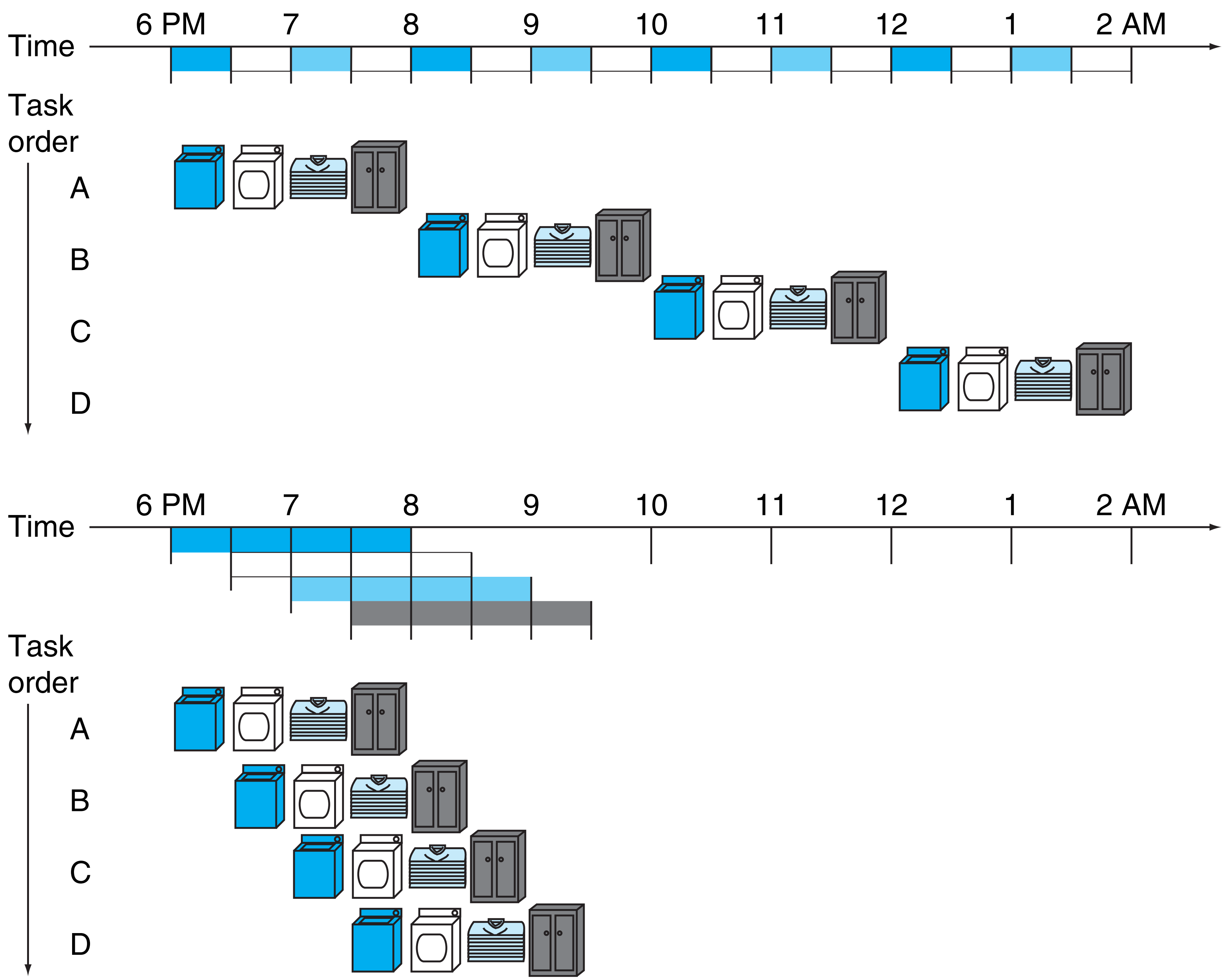
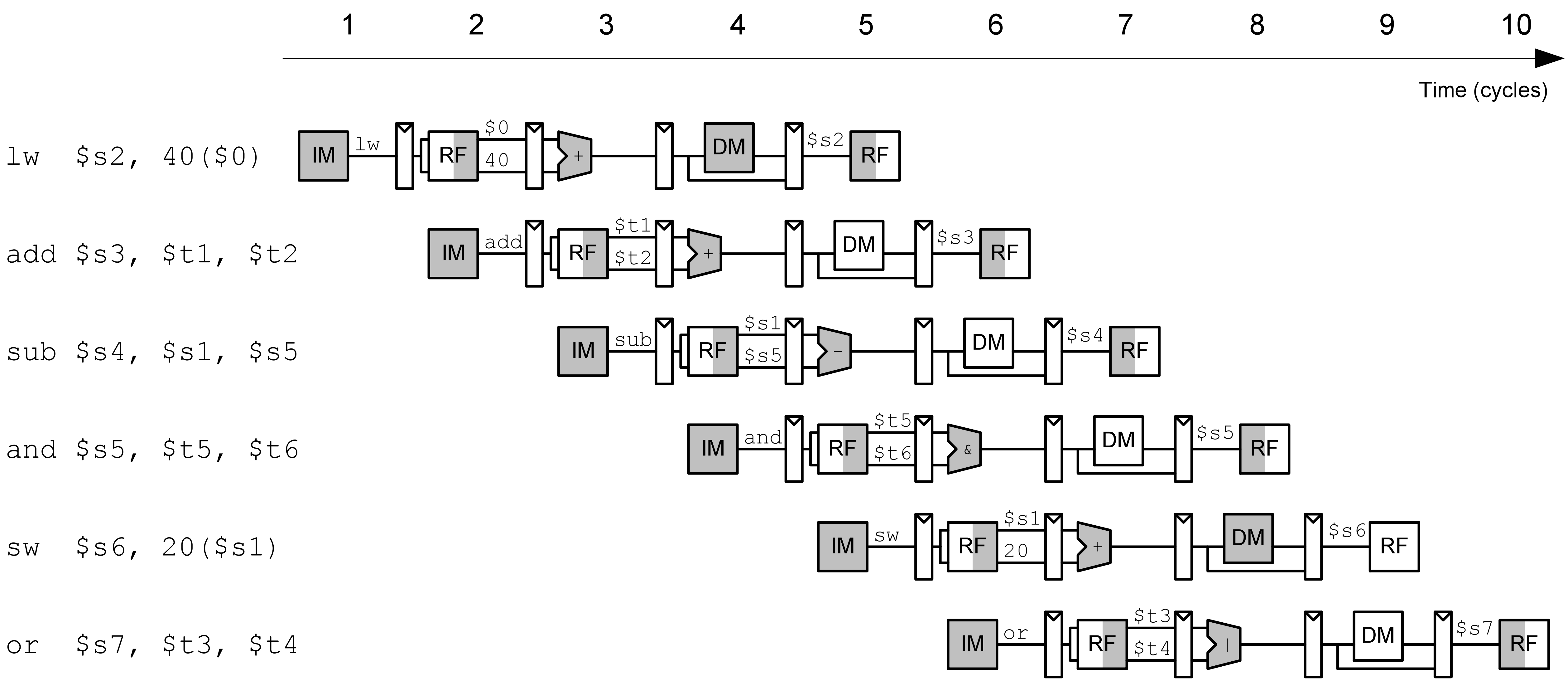


FIGURE 4.25 The laundry analogy for pipelining. Ann, Brian, Cathy, and Don each have dirty clothes to be washed, dried, folded, and put away. The washer, dryer, "folder," and "storer" each take 30 minutes for their task. Sequential laundry takes 8 hours for 4 loads of wash, while pipelined laundry takes just 3.5 hours. We show the pipeline stage of different loads over time by showing copies of the four resources on this two-dimensional time line, but we really have just one of each resource. Copyright © 2009 Elsevier, Inc. All rights reserved.

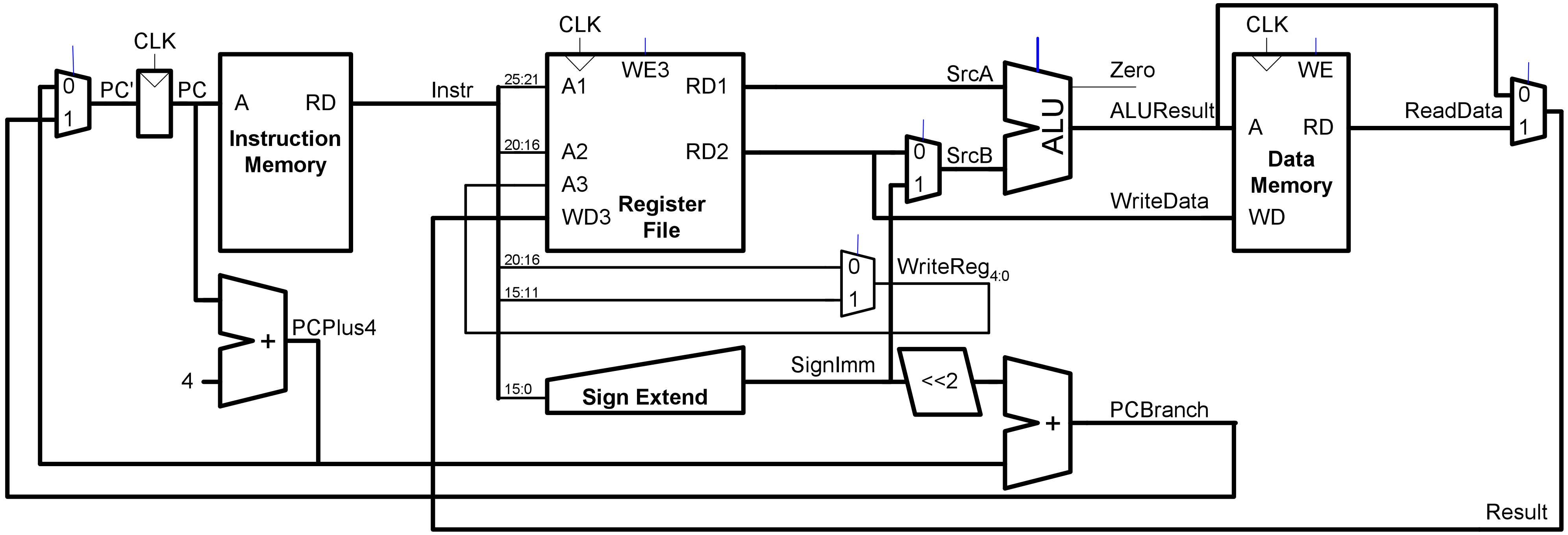
Pipelining Abstraction

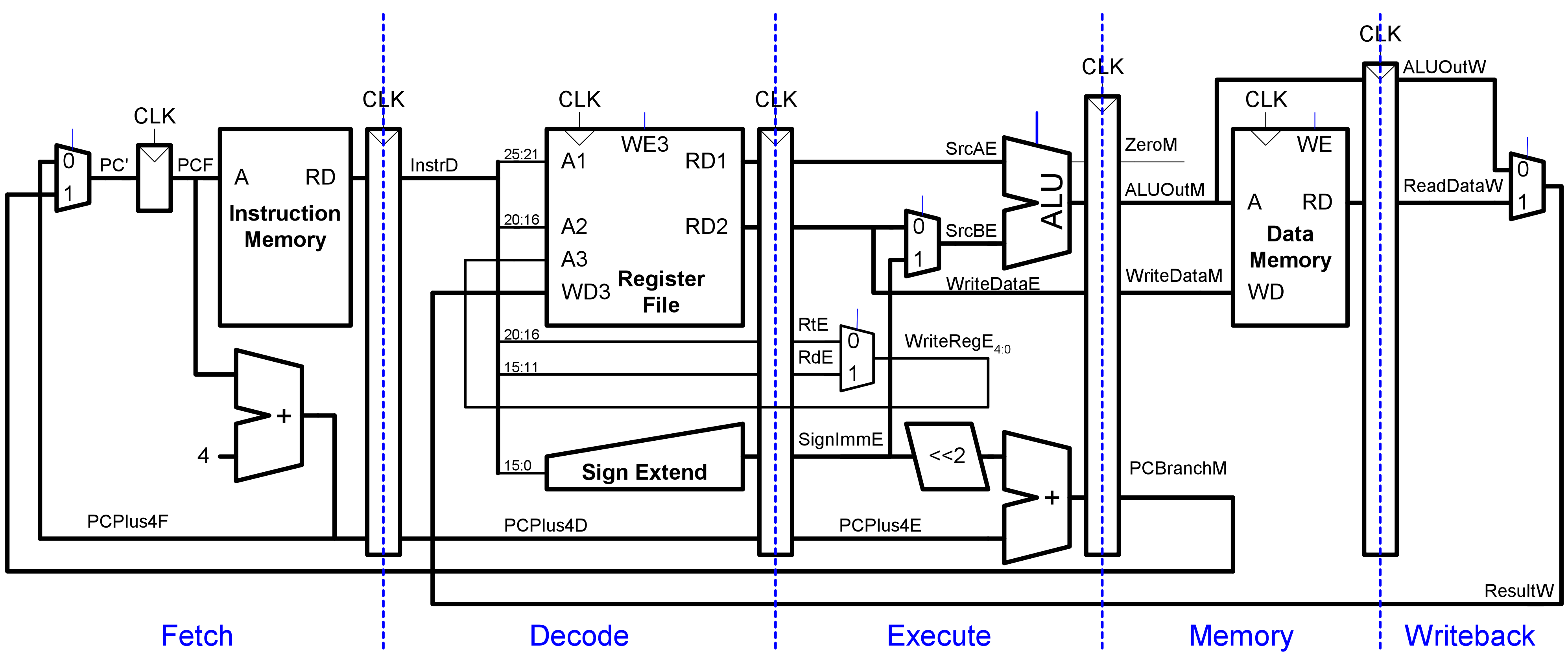


MIPS Pipeline

- Five stages, one step per stage, one stage per cycle
  - **IF**: Instruction fetch from (instruction) memory
  - **ID**: Instruction decode and register read (register file read)
  - **EX**: Execute operation or calculate address (ALU) or branch condition + calculate branch address
  - **MEM**: Access memory operand (memory) / adjust PC counter
  - **WB**: Write result back to register (reg file again)
- Note: Every instruction has every stage, though not every instruction needs every stage

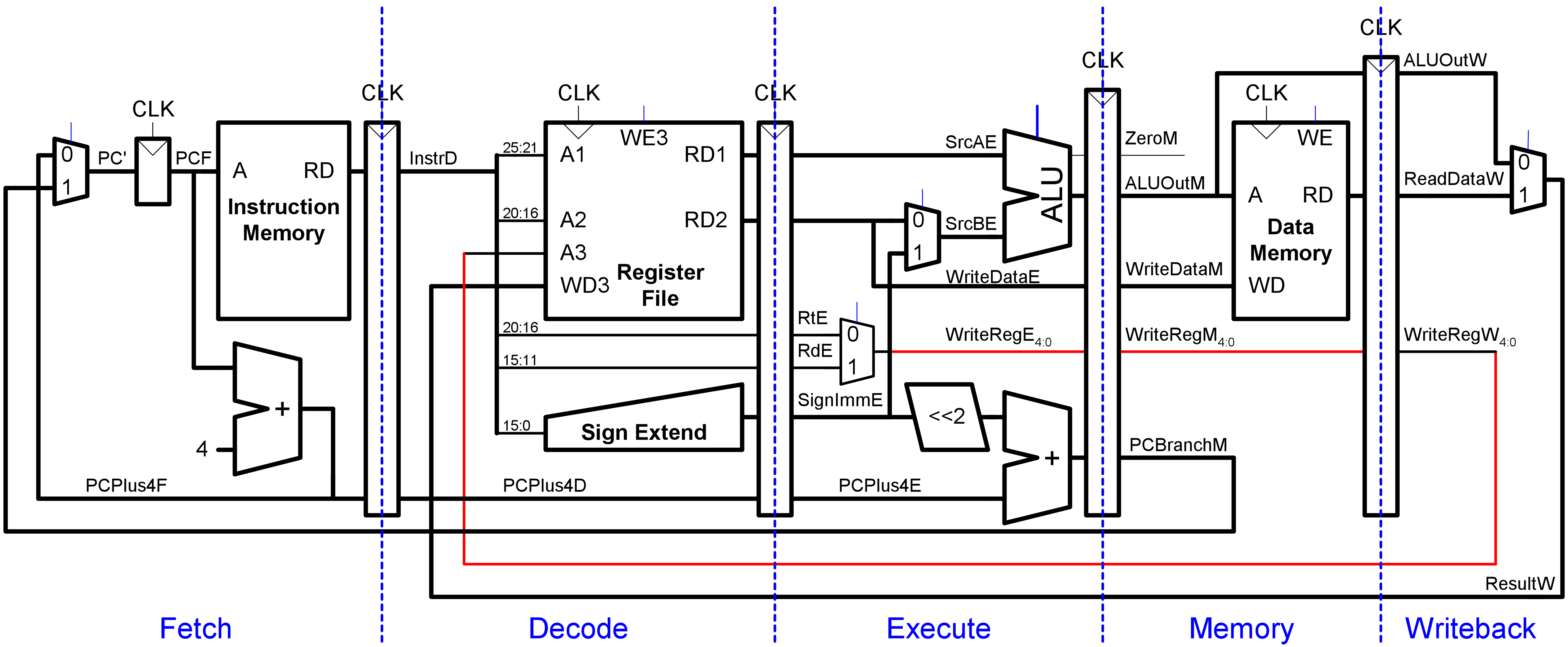
Single-Cycle and Pipelined Datapath



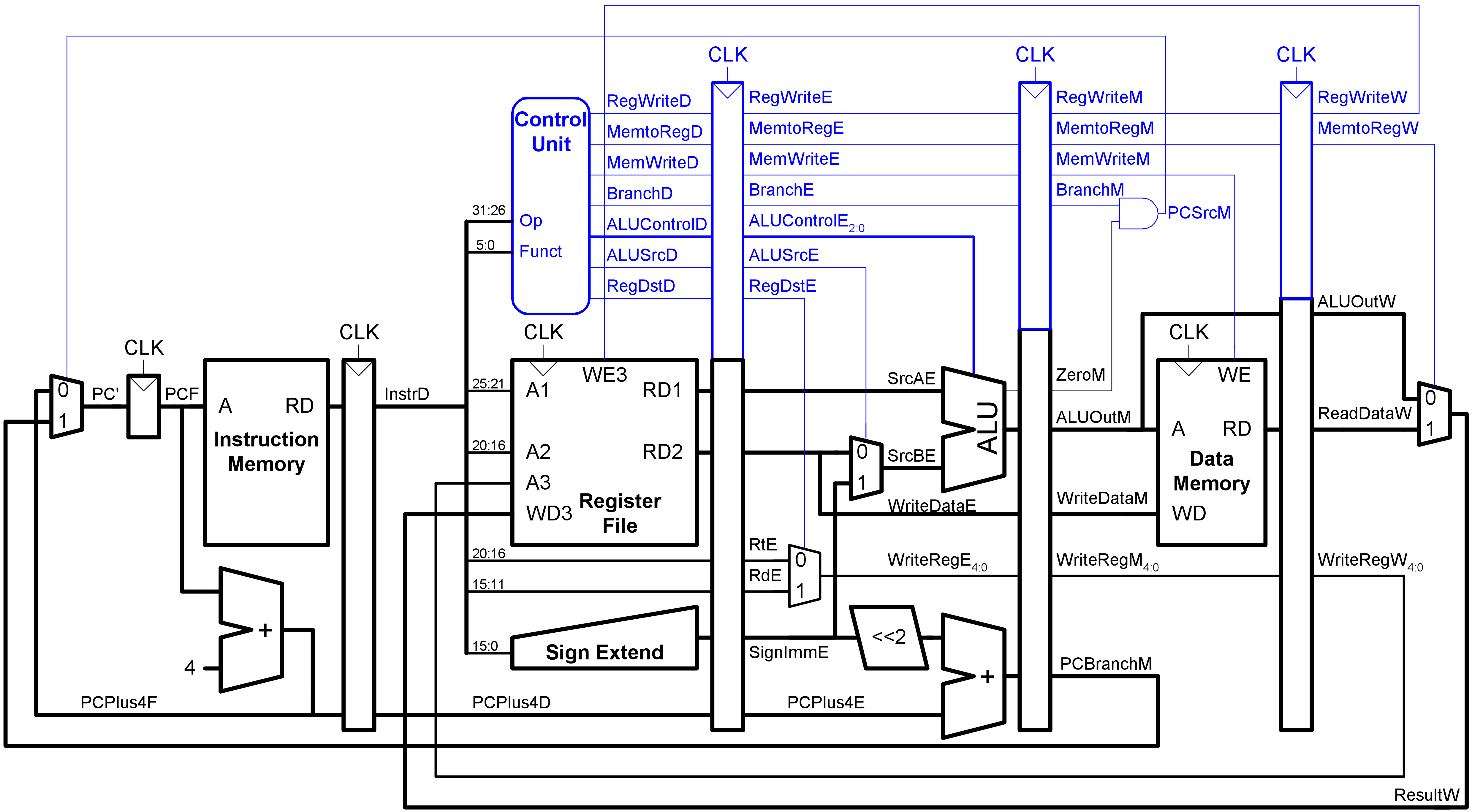


Corrected Pipelined Datapath

- WriteReg must arrive at the same time as Result



Pipelined Control



Same control unit as single-cycle processor
Control delayed to proper pipeline stage

Pipeline Hazard

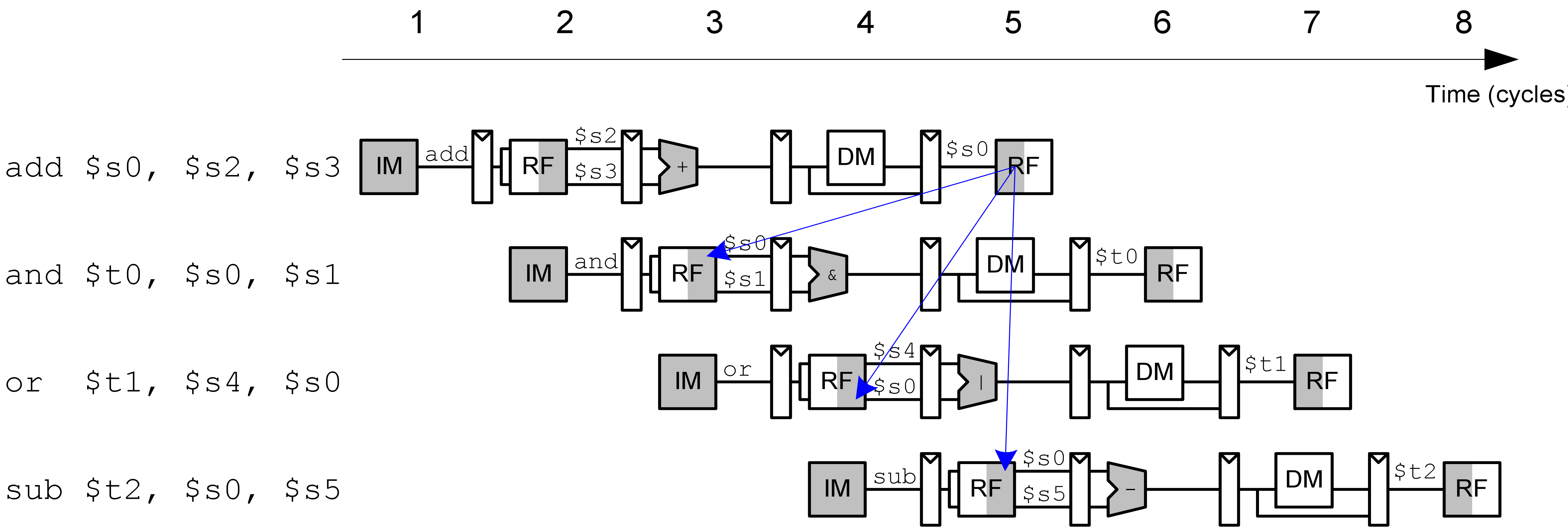
- Occurs when an instruction depends on results from previous instruction that hasn't completed.

- Types of hazards:

  - **Data hazard**: register value not written back to register file yet

  - **Control hazard**: next instruction not decided yet (caused by branches)

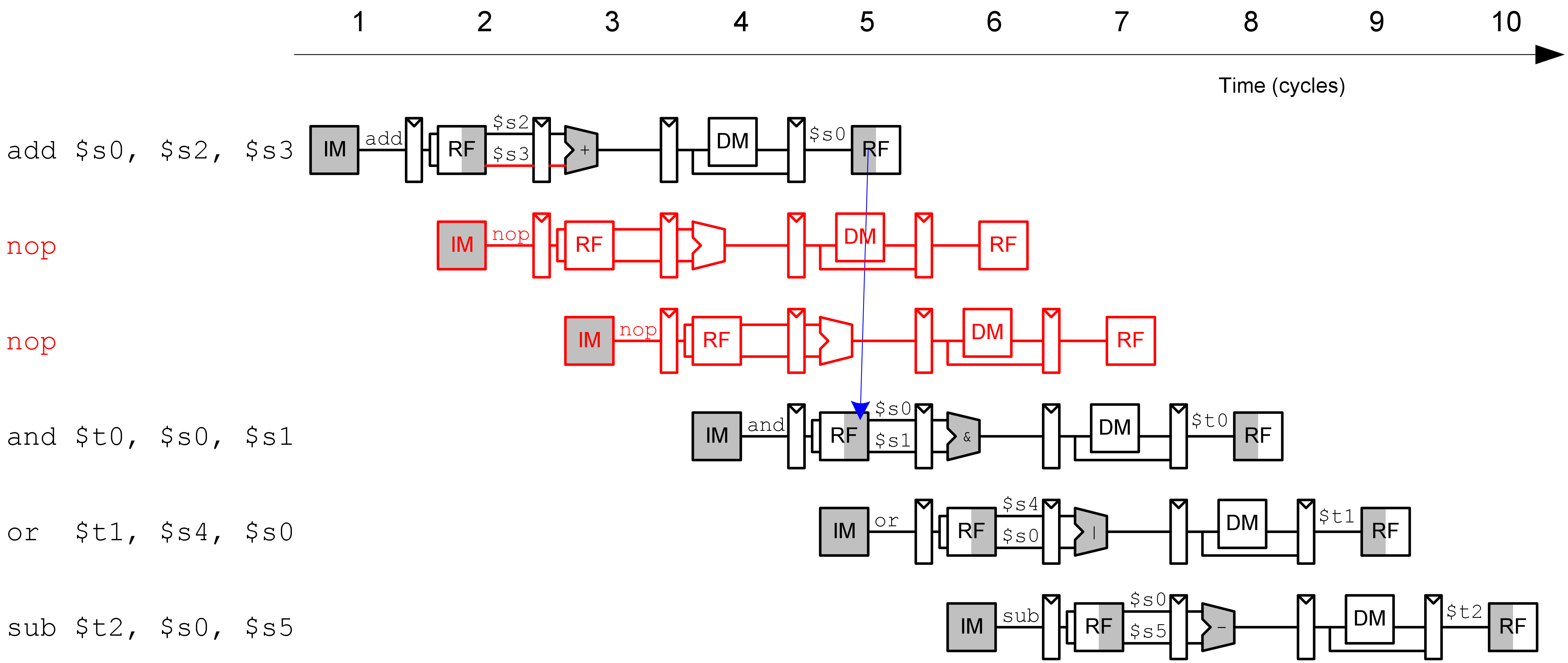
Data Hazard



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

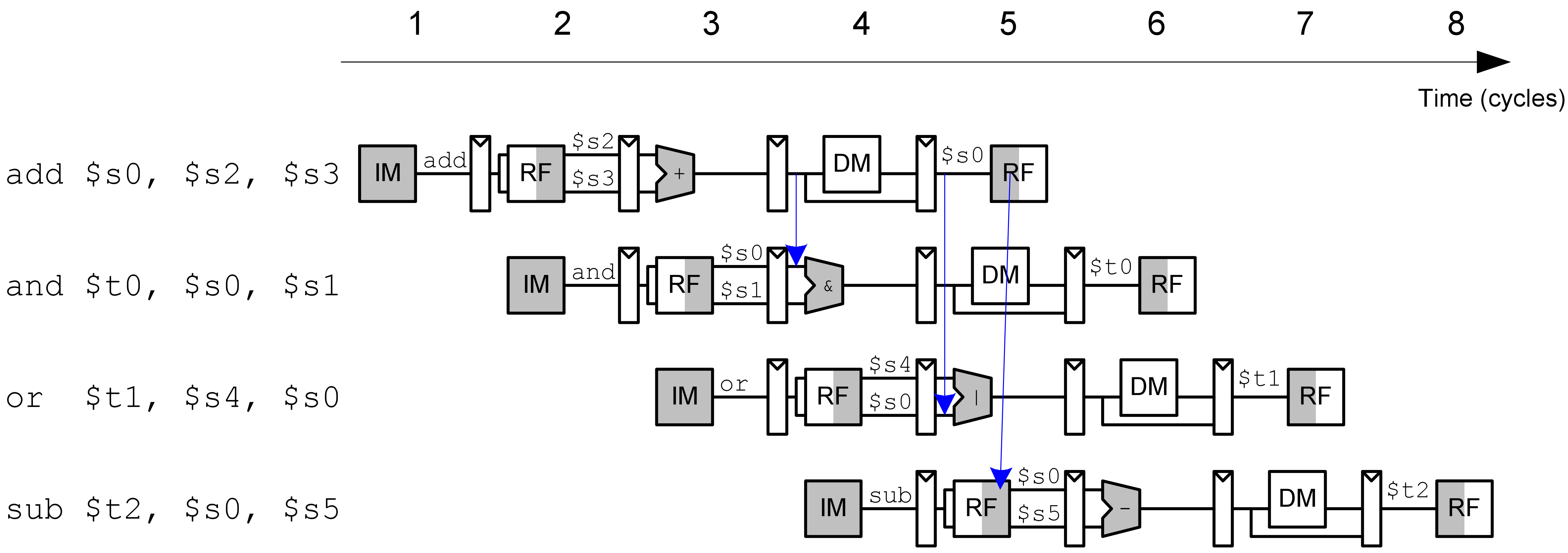
Compile-Time Hazard Elimination

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

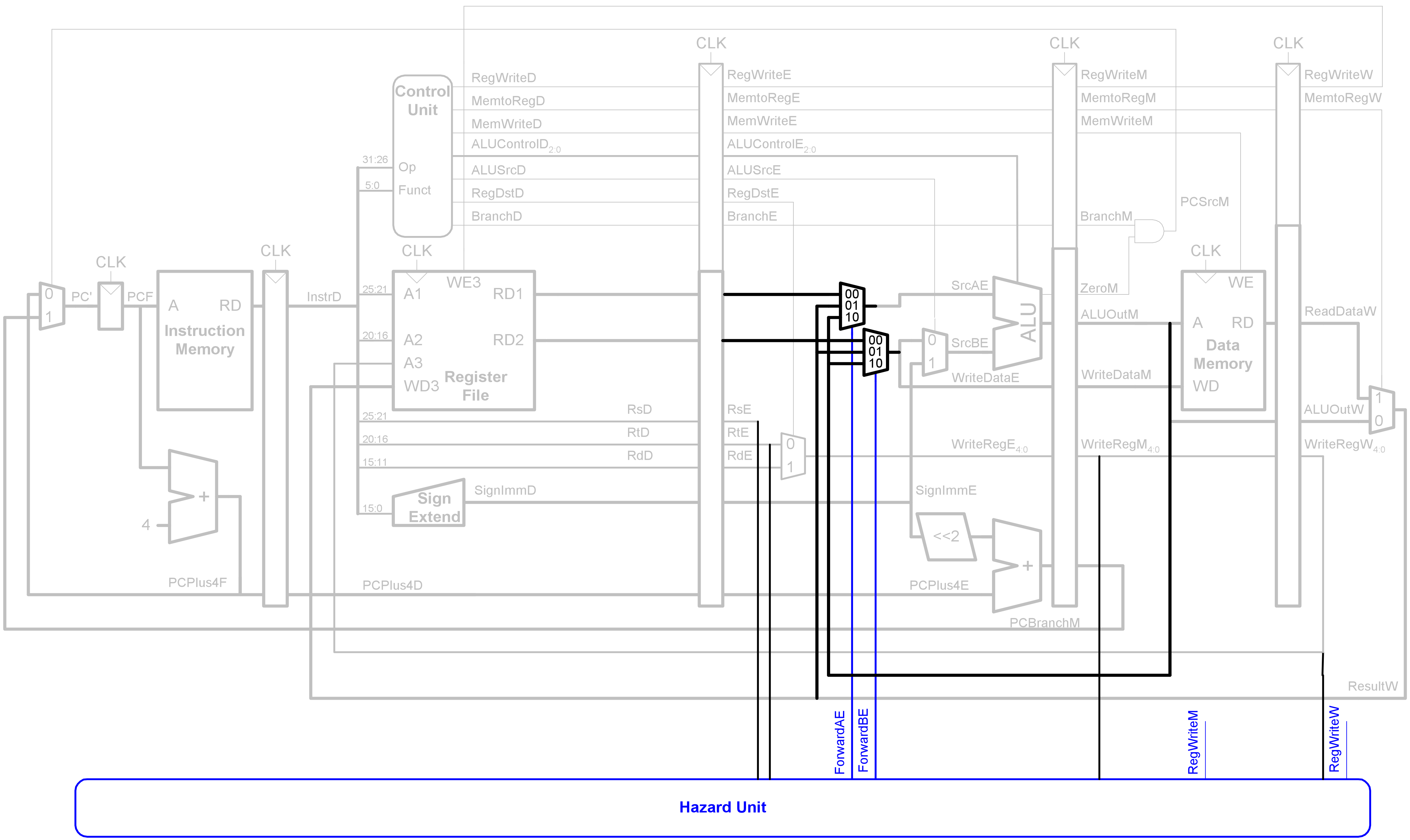


Data Forwarding (Concept)

- Don't wait for data to be written to register file, send it directly to where needed.



Data Forwarding (Circuitry)



Data Forwarding

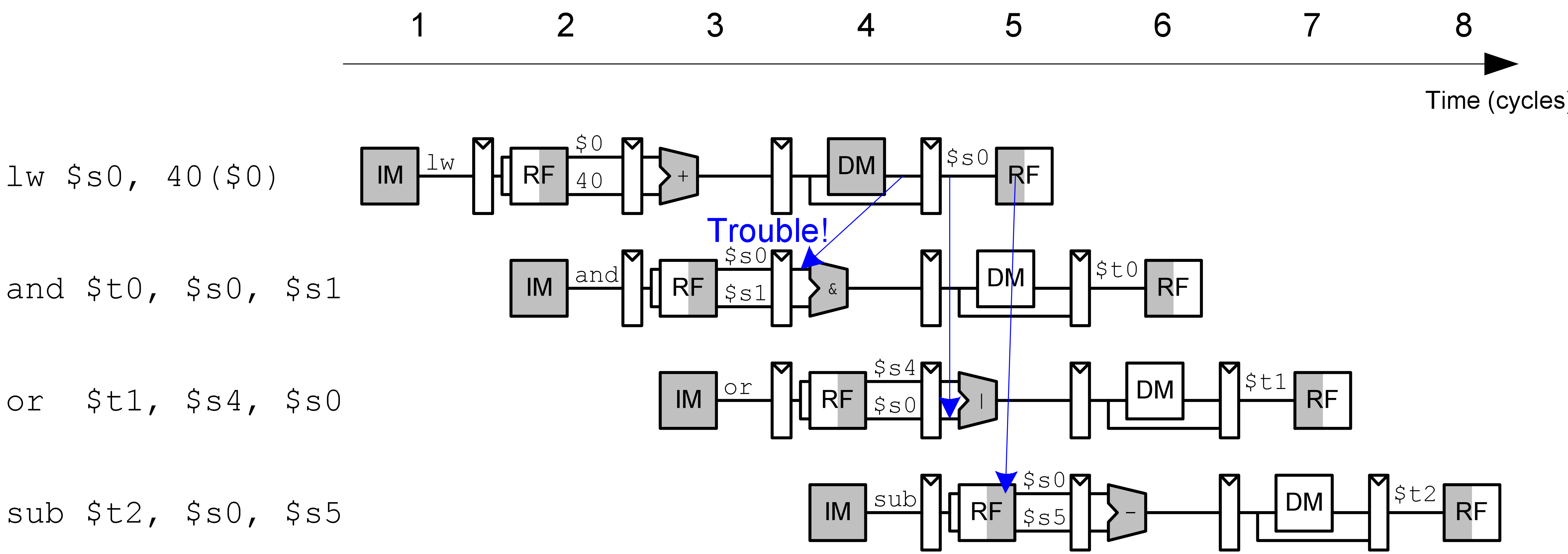
- Forward to X stage from either M or WB

- Forwarding logic for *ForwardAE*:

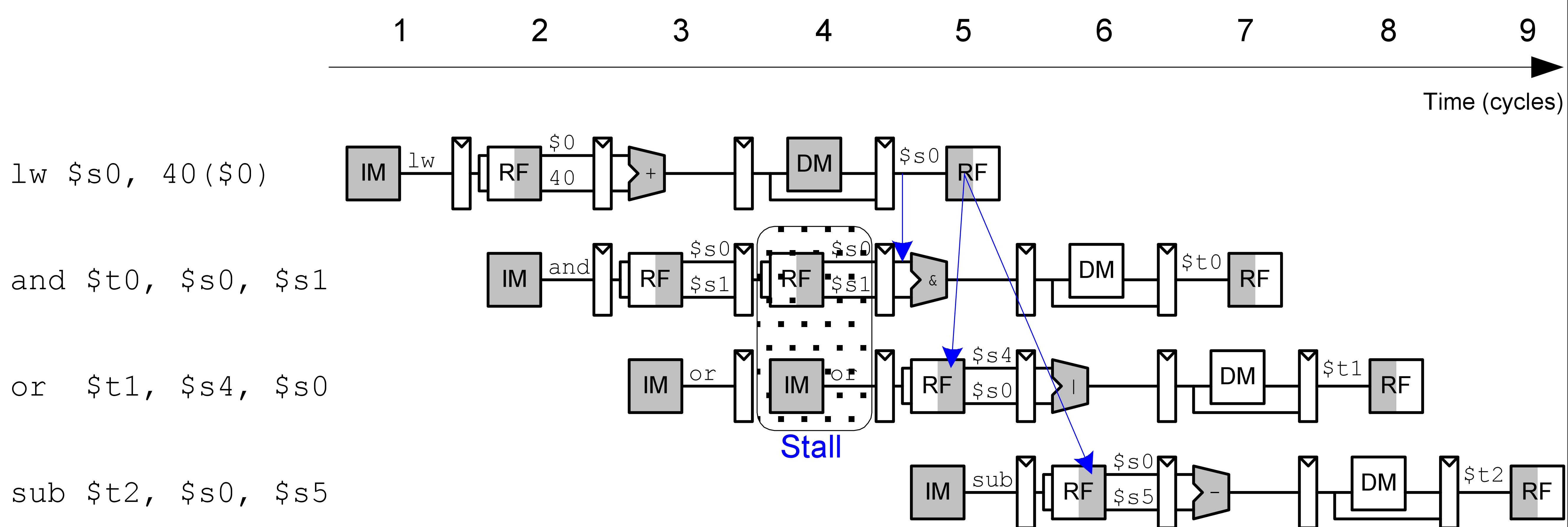
```
if        (rsE != 0 AND rsE == WriteRegM AND RegWriteM)
then      ForwardAE = 10
else if   (rsE != 0 AND rsE == WriteRegW AND RegWriteW)
then      ForwardAE = 01
else      ForwardAE = 00
```

- Forwarding logic for `ForwardBE` same, but replace `rsE` with `rtE`

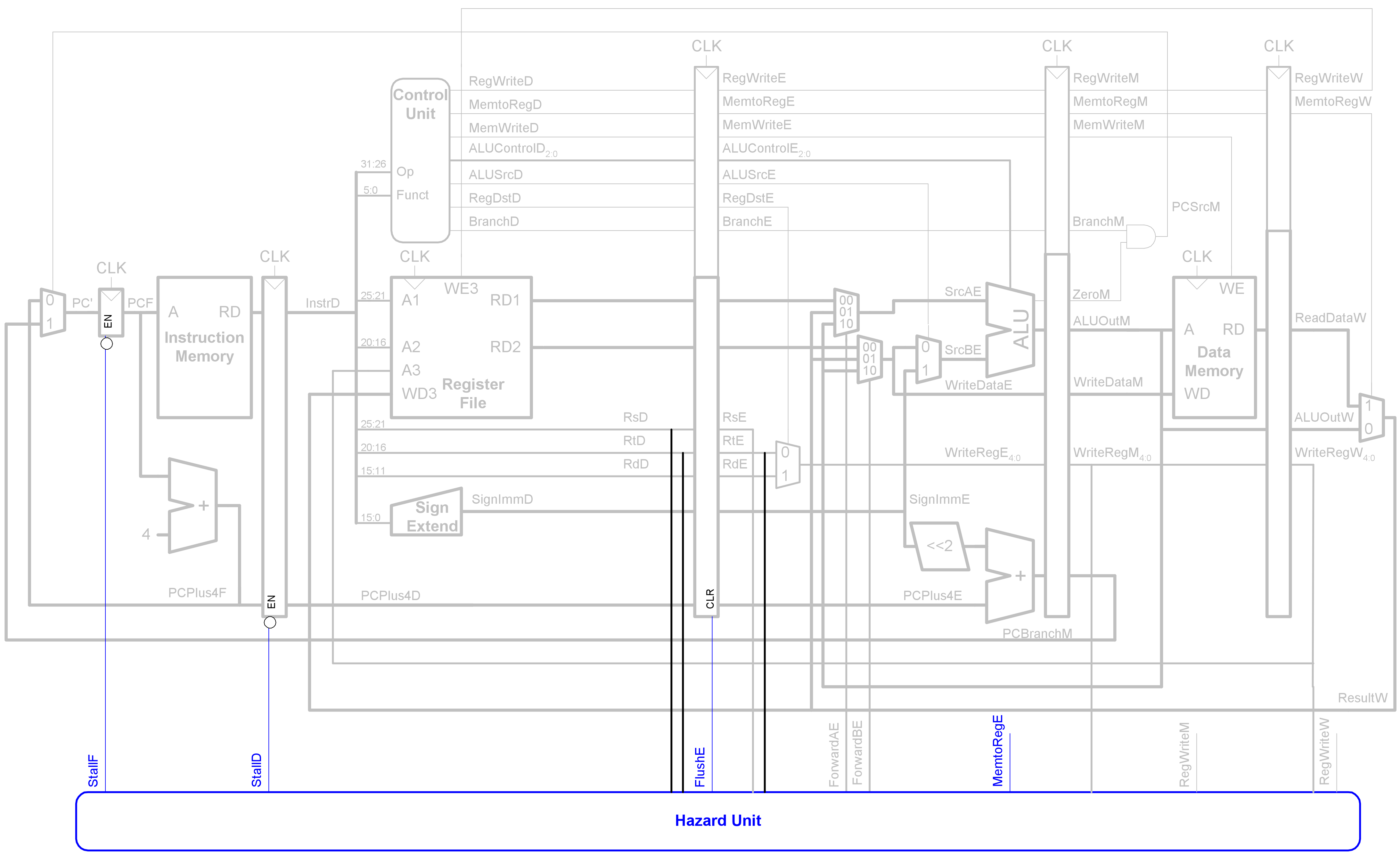
Stalling (Stall Needed)



Stalling (Instructions Stalled)



Stalling Hardware



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

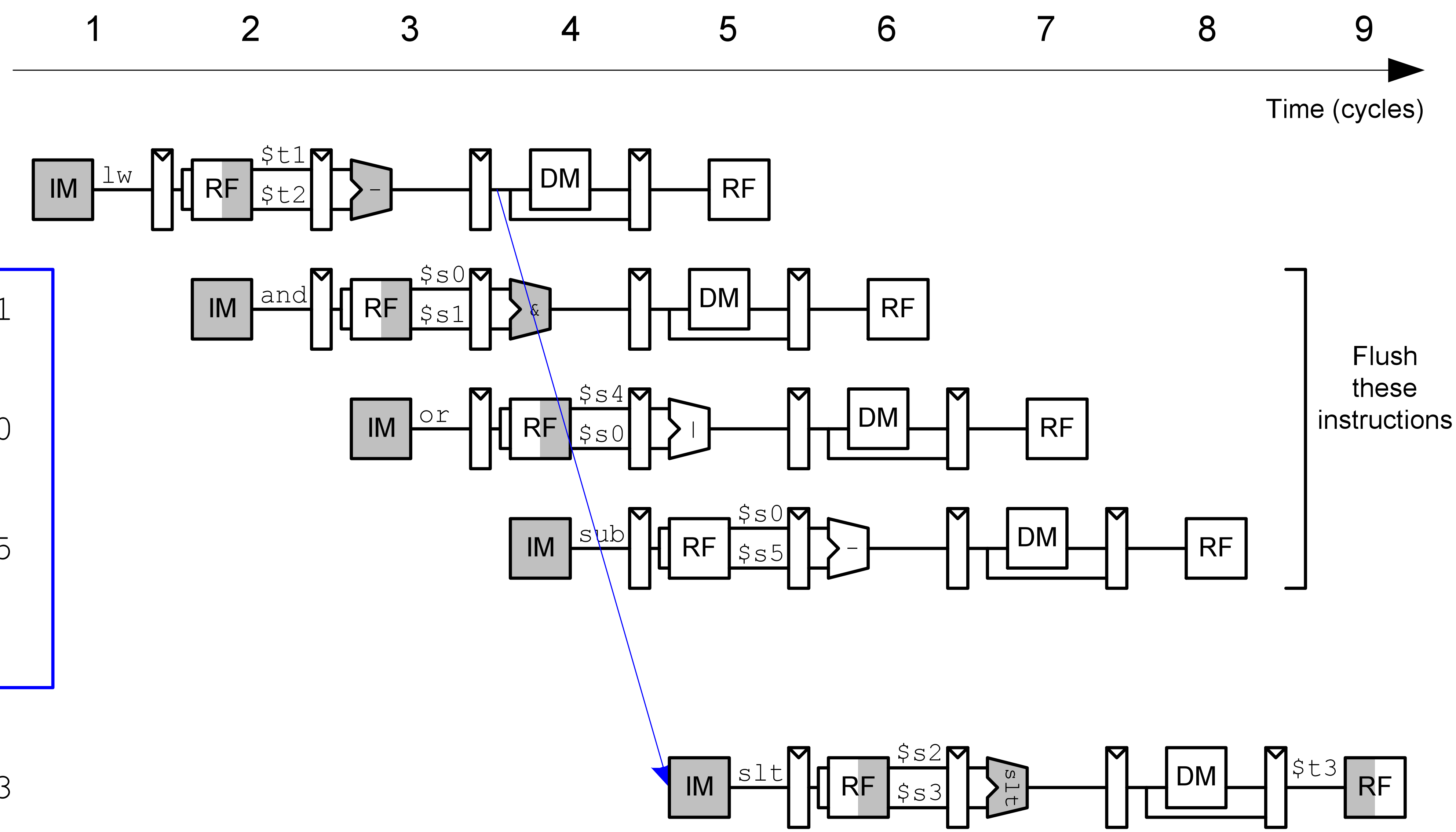
StallF = StallD = FlushE = lwstall

Control Hazards

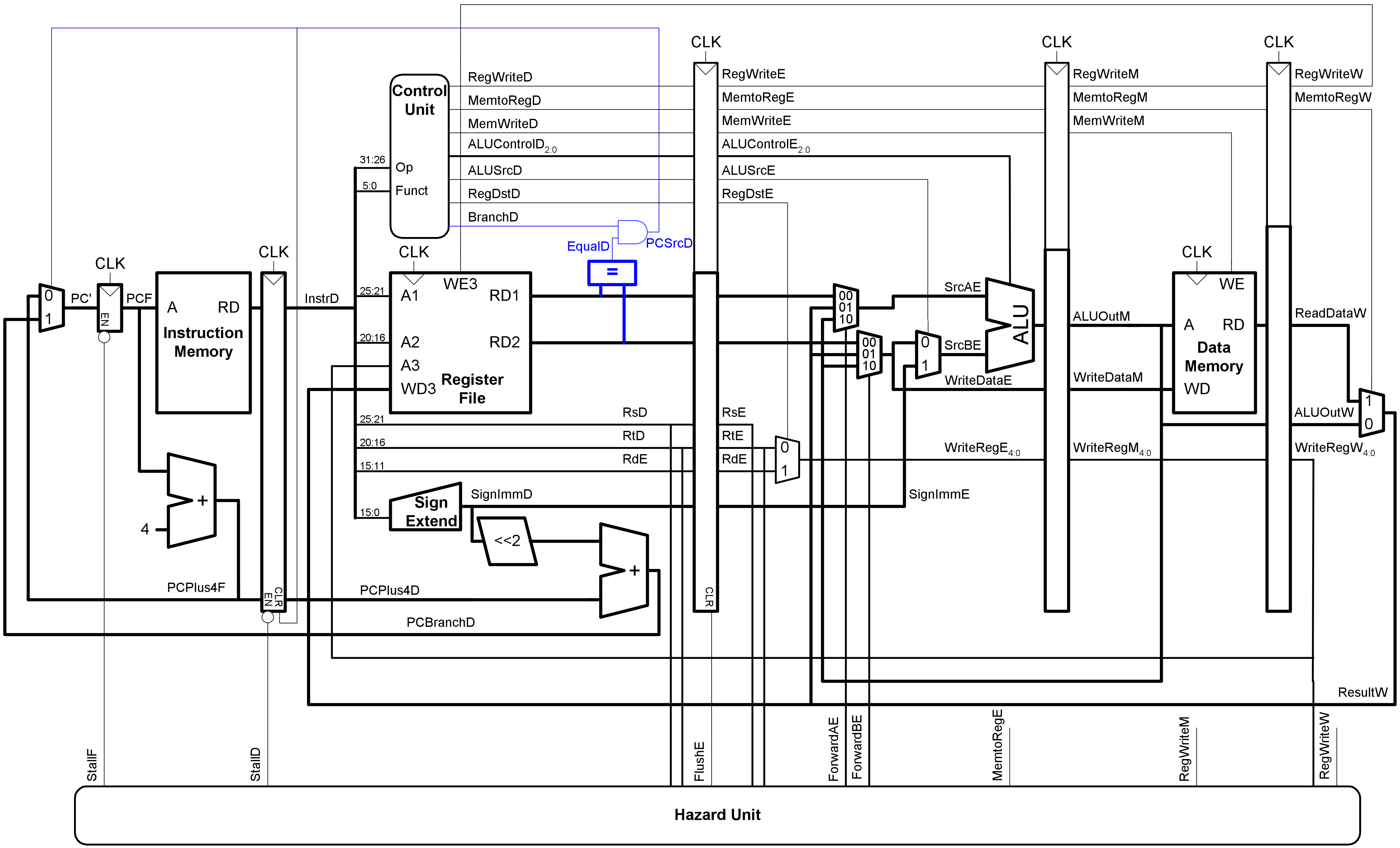
- `beq`:
  - Branch is not determined until the fourth stage of the pipeline
  - Instructions after the branch are fetched before branch occurs
  - These instructions must be flushed if the branch happens
- Branch misprediction penalty
  - Number of instruction flushed when branch is taken
  - May be reduced by determining branch earlier

Control Hazards

```
20   beq $t1, $t2, 40
24   and $t0, $s0, $s1
28   or  $t1, $s4, $s0
2C   sub $t2, $s0, $s5
30   ...
...
64   slt $t3, $s2, $s3
```



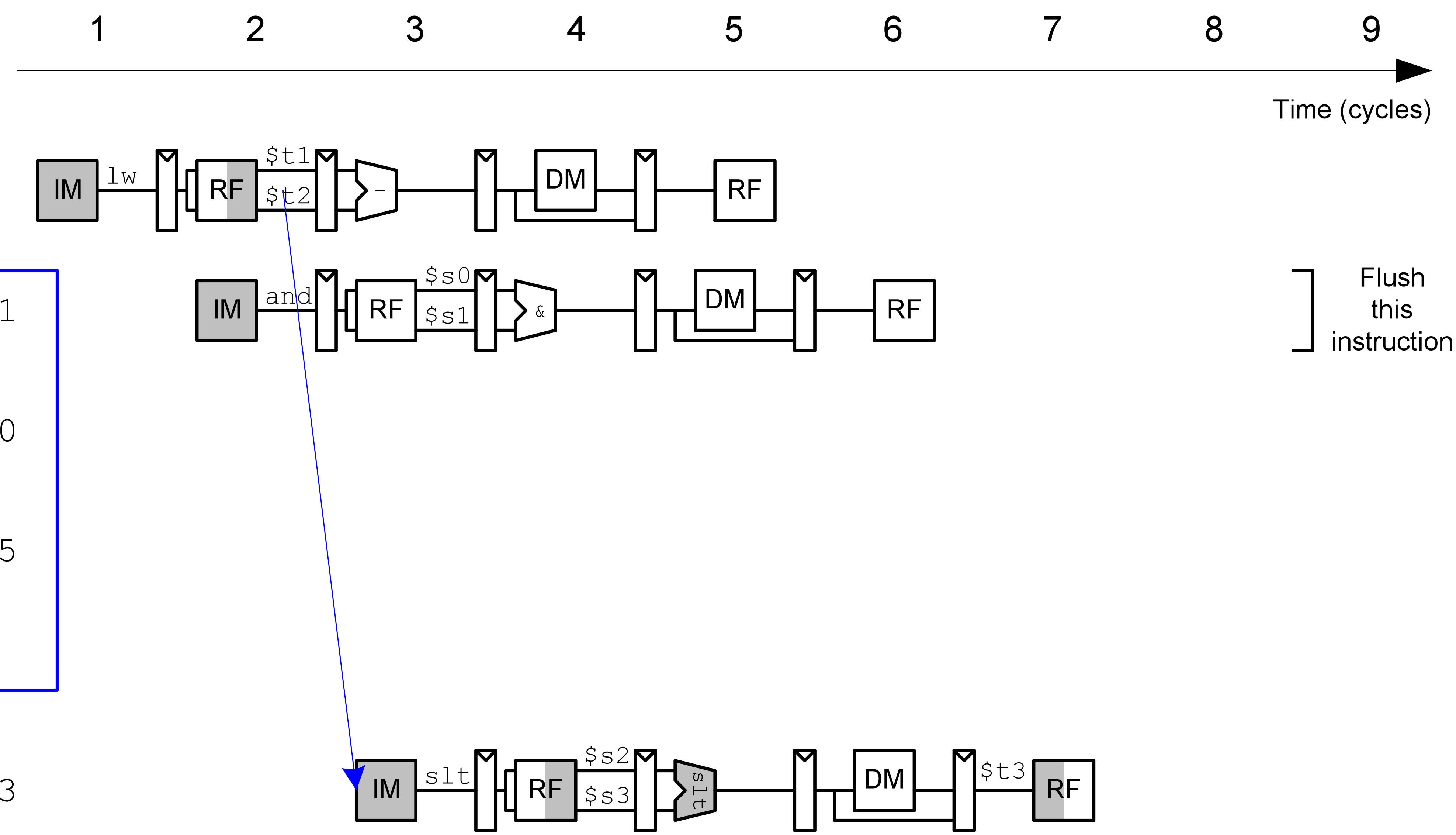
Control Hazards: Early Branch Resolution



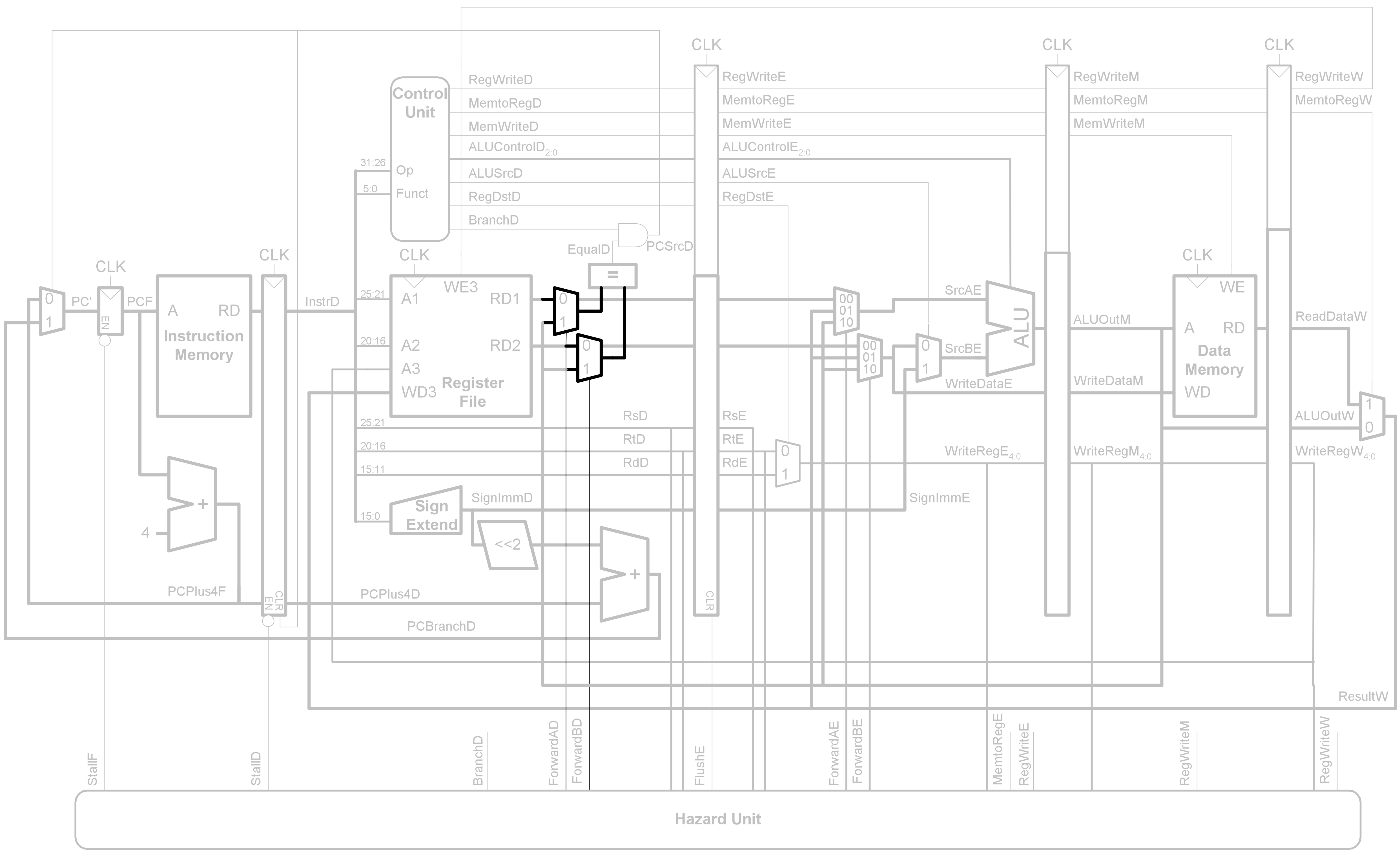
Introduced another data hazard in Decode stage

Control Hazards with Early Branch Resolution

```
20   beq $t1, $t2, 40
24   and $t0, $s0, $s1
28   or  $t1, $s4, $s0
2C   sub $t2, $s0, $s5
30   ...
...
64   slt $t3, $s2, $s3
```



Handling Data and Control Hazards



Control Forwarding and Stalling Hardware

- Forwarding logic:

```
ForwardAD = (rsD !=0) AND (rsD == WriteRegM) AND RegWriteM
ForwardBD = (rtD !=0) AND (rtD == WriteRegM) AND RegWriteM
```

- Stalling logic:

```
branchstall = (BranchD    AND
               RegWriteE  AND
               (WriteRegE == rsD OR WriteRegE == rtD))
               OR
               (BranchD    AND
                MemtoRegM  AND
                (WriteRegM == rsD OR WriteRegM == rtD))

StallF = StallD = FlushE = lwstall OR branchstall
```

Branch Prediction

- Guess whether branch will be taken
  - Backward branches are usually taken (loops)
  - Perhaps consider history of whether branch was previously taken to improve the guess
- Good prediction reduces the fraction of branches requiring a flush

Pipelined Performance Example

- Ideally CPI = 1
- But need to handle stalling (caused by loads and branches)

- SPECINT2000 benchmark:
  - 25% loads
  - 10% stores
  - 11% branches
  - 2% jumps
  - 52% R-type

- Suppose:
  - 40% of loads used by next instruction
  - 25% of branches mispredicted
- **What is the average CPI?**

Pipelined Performance Example (SOLN)

- Ideally CPI = 1
- But need to handle stalling (caused by loads and branches)

- SPECINT2000 benchmark:
  - 25% loads
  - 10% stores
  - 11% branches
  - 2% jumps
  - 52% R-type

- Suppose:
  - 40% of loads used by next instruction
  - 25% of branches mispredicted
- **What is the average CPI?**

Load/Branch CPI
= 1 when no stalling
= 2 when stalling

Thus,
$CPI_{lw} = 1(0.6) + 2(0.4) = 1.4$
$CPI_{beq} = 1(0.75) + 2(0.25) = 1.25$

Thus,
Average CPI
$= (0.25)(1.4) + (0.1)(1) + (0.11)(1.25) + (0.02)(2) + (0.52)(1)$
$= 1.15$

Pipelined Processor Critical Path

$$T_c = \max\{$$
$$t_{pcq} + t_{\text{mem}} + t_{\text{setup}}$$
$$2(t_{RFread} + t_{\text{mux}} + t_{\text{eq}} + t_{\text{AND}} + t_{\text{mux}} + t_{\text{setup}})$$
$$t_{pcq} + t_{\text{mux}} + t_{\text{mux}} + t_{\text{ALU}} + t_{\text{setup}}$$
$$t_{pcq} + t_{\text{memwrite}} + t_{\text{setup}}$$
$$2(t_{pcq} + t_{\text{mux}} + t_{\text{RFwrite}})\}$$

Pipelined Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq} PC	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	$t_{RF\text{read}}$	150
Register file setup	$t_{RF\text{setup}}$	20
Equality comparator	t_{eq}	40
AND gate	t_{AND}	15
Memory write	T_{memwrite}	220
Register file write	$t_{RF\text{write}}$	100

$$T_c = \mathbf{2(t_{RFread} + t_{mux} + t_{eq} + t_{AND} + t_{mux} + t_{setup})}$$
$$= 2[150 + 25 + 40 + 15 + 25 + 20] \text{ ps } \mathbf{= 550 \text{ ps}}$$

Pipelined Performance Example (2)

For a program with 100 billion instructions executing on a pipelined MIPS processor,

CPI = 1.15

T_c = 550 ps

Execution Time = (# instructions) × CPI × T_c

$= (100 \times 10^9)(1.15)(550 \times 10^{-12})$

= 63 seconds

Processor	Execution Time (s)	Speedup (single cycle baseline)
Single-cycle	95	1
Pipelined	63	1.51