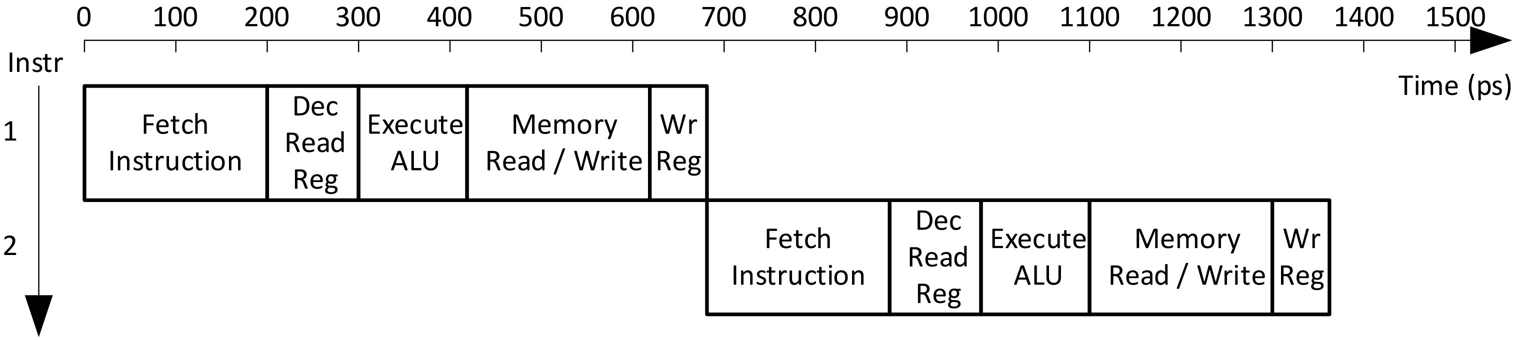
Pipelined RISC-V Processor

Pipelined RISC-V Processor

- **Temporal parallelism**
- Divide single-cycle processor into **5 stages**:
  - Fetch
  - Decode
  - Execute
  - Memory
  - Writeback
- Add **pipeline registers** between stages

Single-Cycle vs. Pipelined Processor

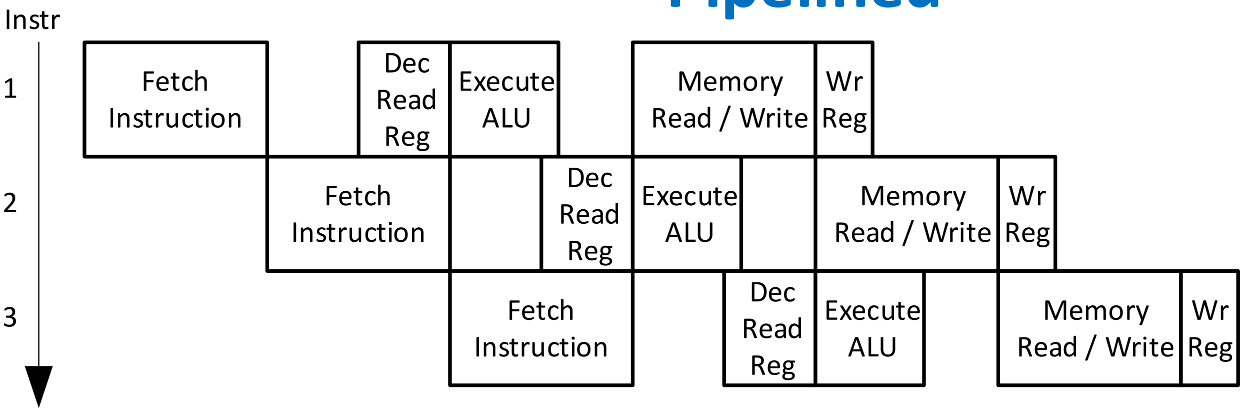
Single-Cycle



1: Fetch Instruction | Dec Read Reg | Execute ALU | Memory Read / Write | Wr Reg

2: Fetch Instruction | Dec Read Reg | Execute ALU | Memory Read / Write | Wr Reg

Pipelined



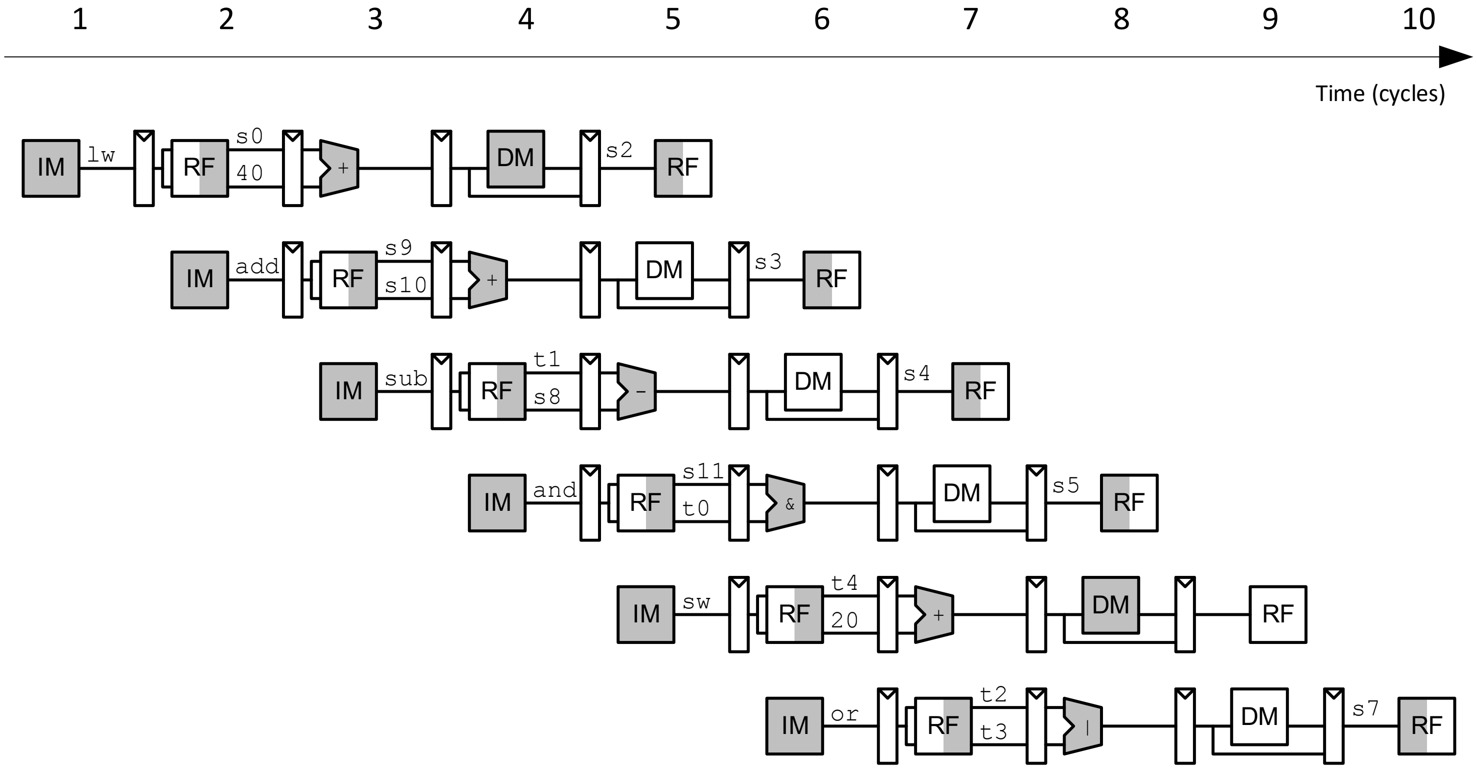
1: Fetch Instruction | Dec Read Reg | Execute ALU | Memory Read / Write | Wr Reg

2: Fetch Instruction | Dec Read Reg | Execute ALU | Memory Read / Write | Wr Reg

3: Fetch Instruction | Dec Read Reg | Execute ALU | Memory Read / Write | Wr Reg

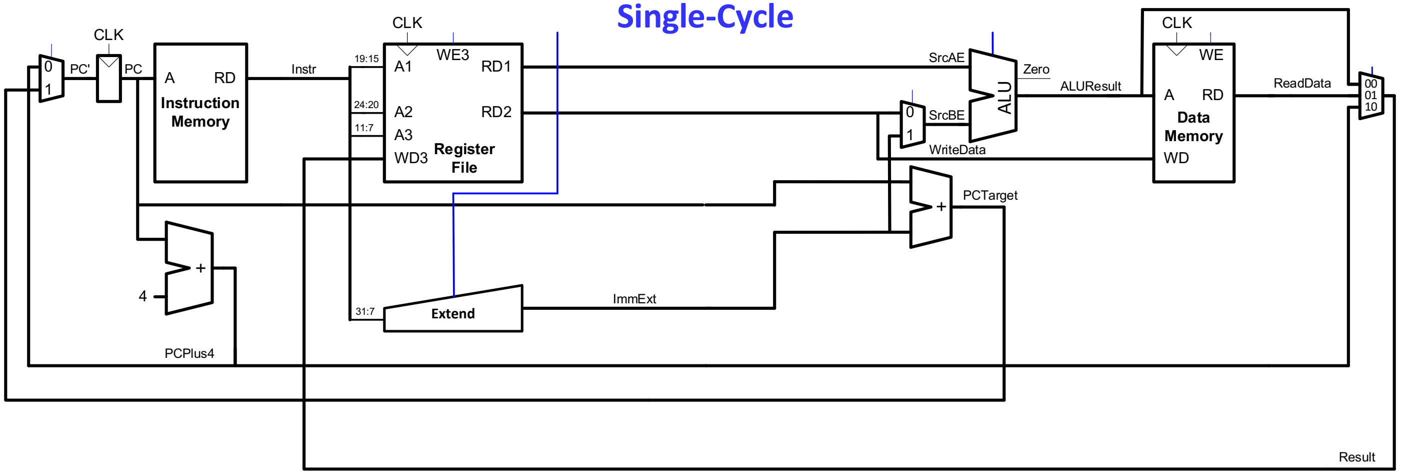
Pipelined Processor Abstraction

```
lw   s2, 40(s0)

add s3, s9, s10

sub s4, t1, s8

and s5, s11, t0

sw   s6, 20(t4)

or   s7, t2, t3
```

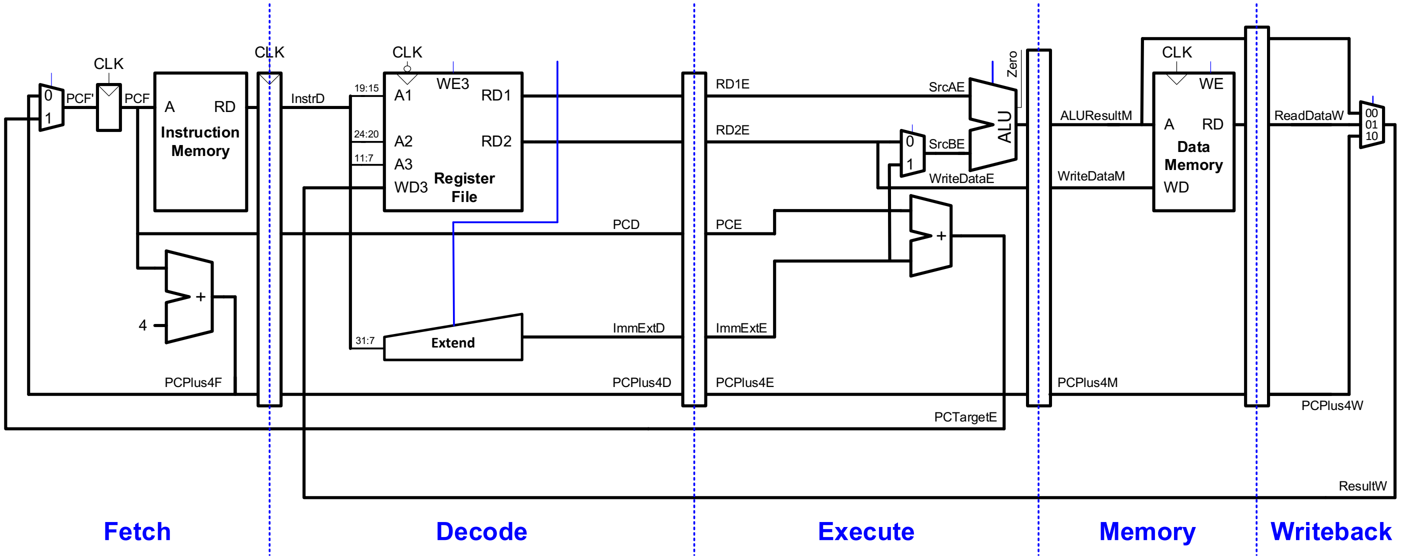


Single-Cycle & Pipelined Datapaths

Single-Cycle

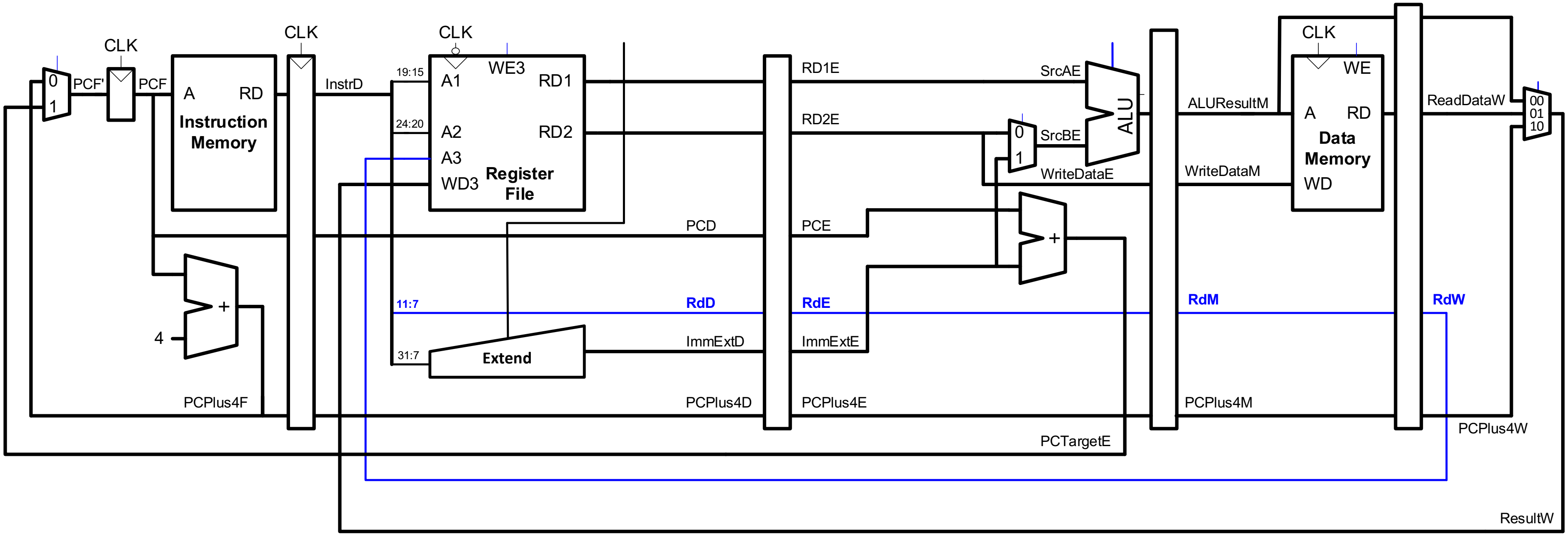


Pipelined



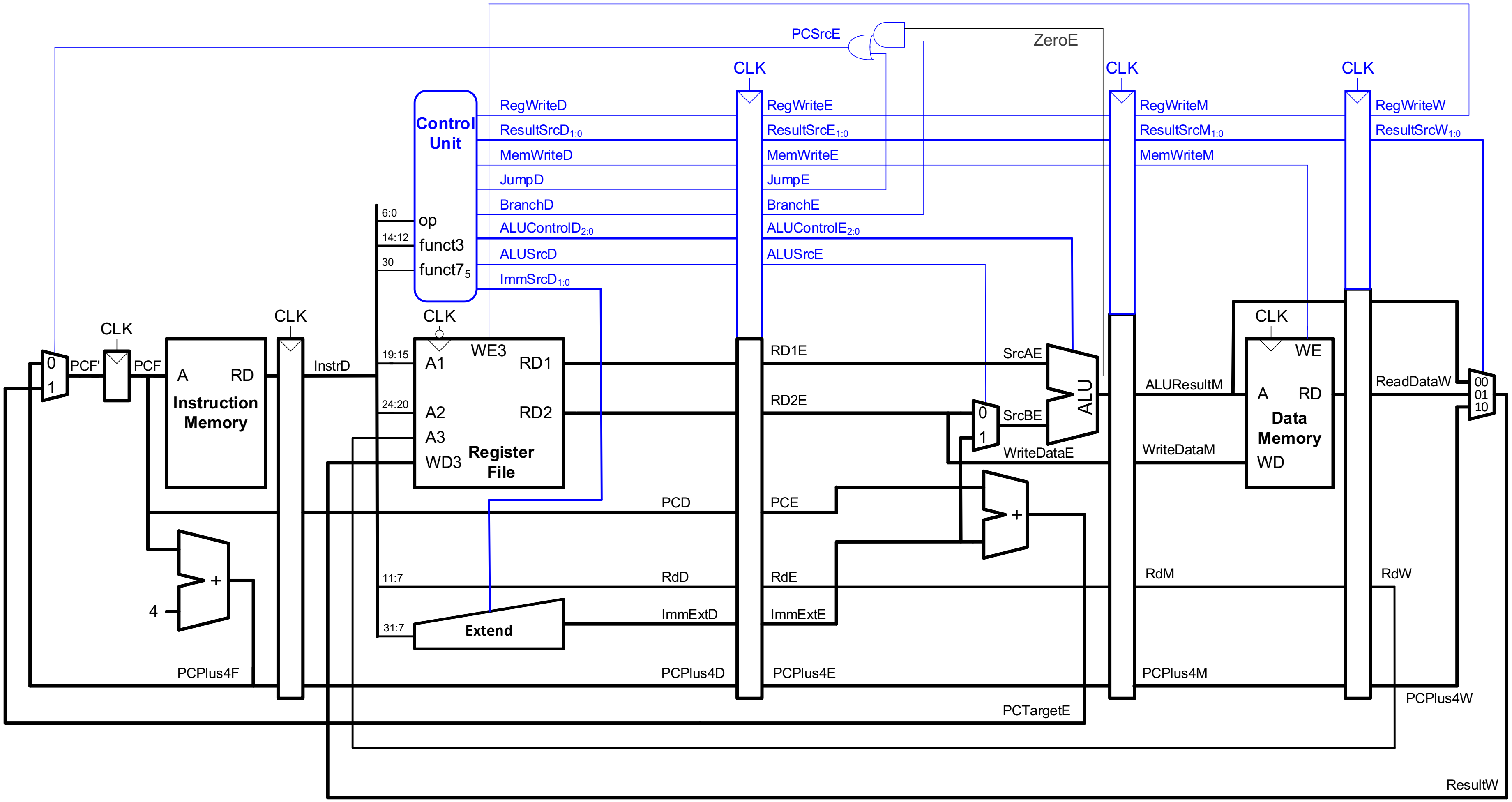
Signals in Pipelined Processor are appended with first letter of stage (i.e., PC**F**, PC**D**, PC**E**).

Corrected Pipelined Datapath



- ***Rd*** must arrive at same time as ***Result***
- Register file written on **falling edge** of *CLK*

Pipelined Processor with Control



- **Same control unit** as single-cycle processor
- **Control signals travel with** the instruction (drop off when used)

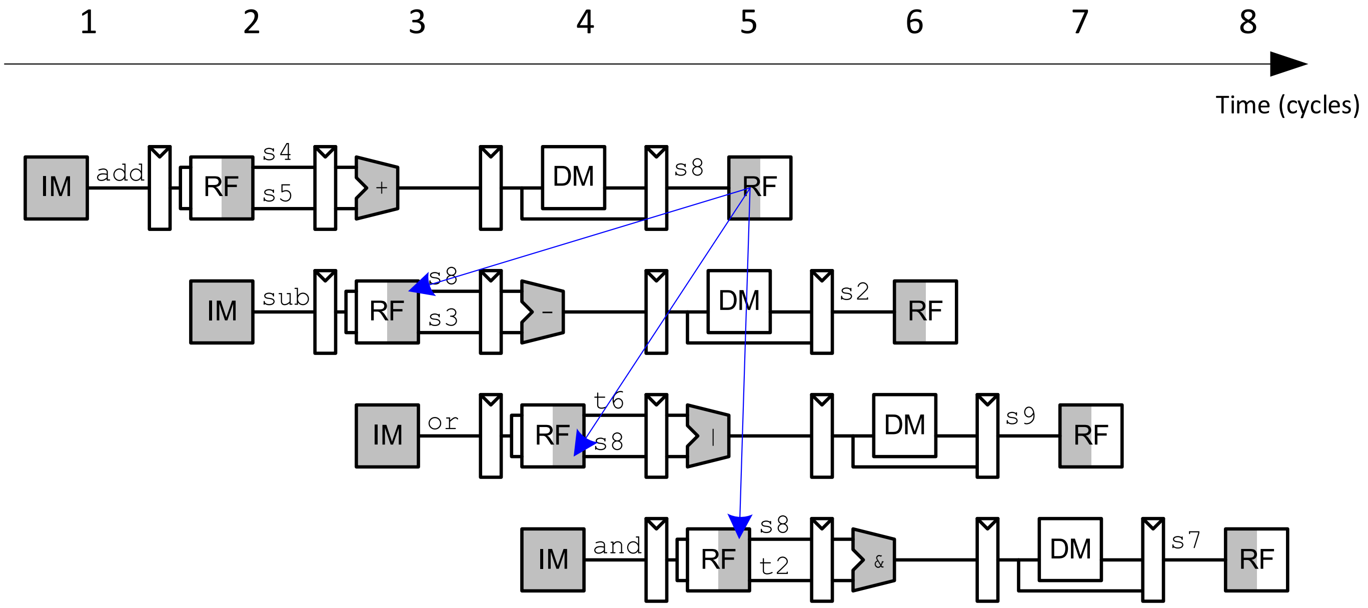
Pipelined Processor Hazards

Pipelined Hazards

- When 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 branch)

Data Hazard

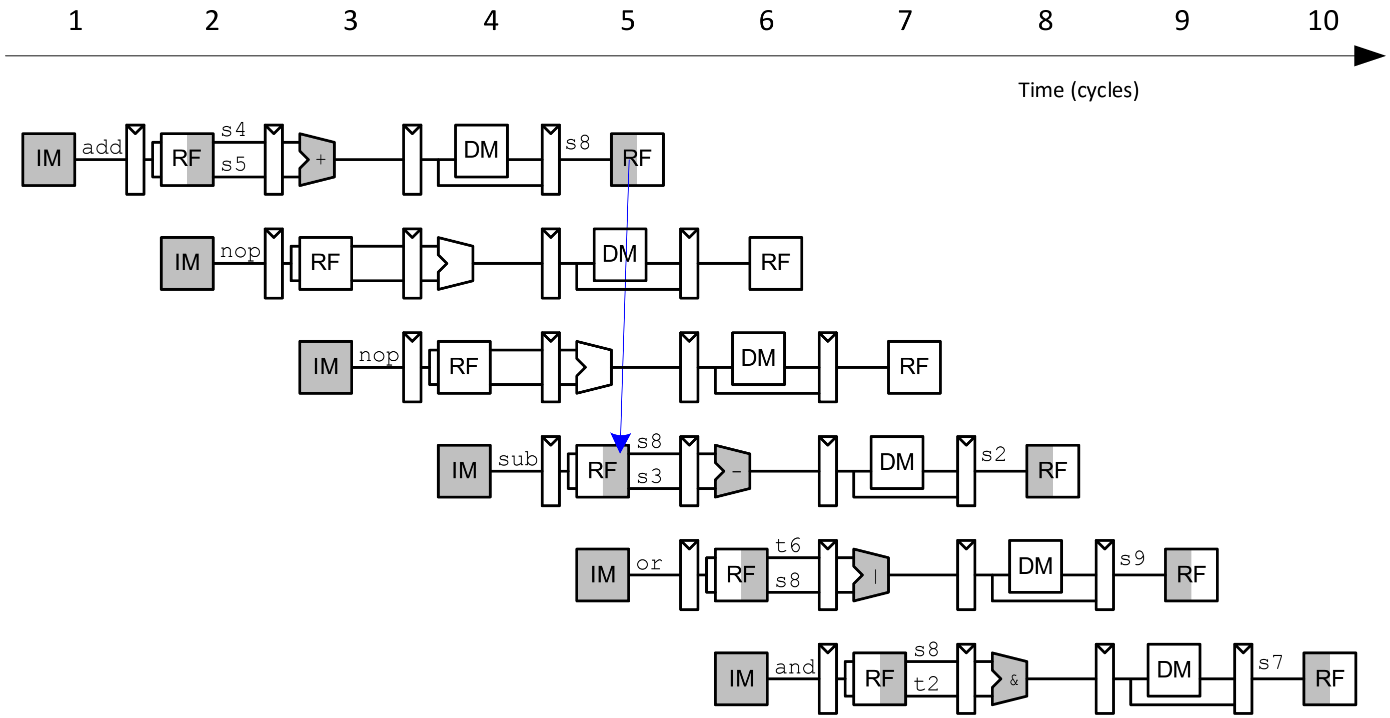
```
add s8, s4, s5
sub s2, s8, s3
or  s9, t6, s8
and s7, s8, t2
```



Handling Data Hazards

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

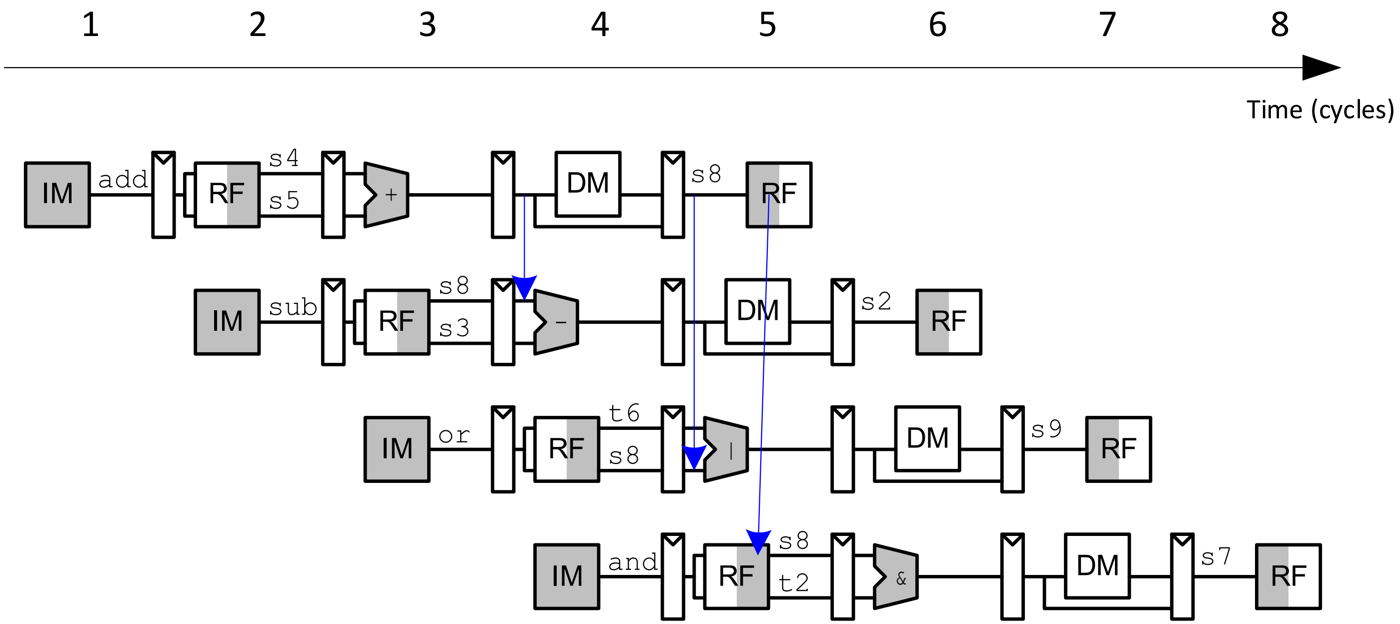
```
add s8, s4, s5
nop
nop
sub s2, s8, s3
or  s9, t6, s8
and s7, s8, t2
```



Data Forwarding

- Data is **available on internal busses** before it is written back to the register file (RF).
- **Forward data** from internal busses **to Execute stage**.

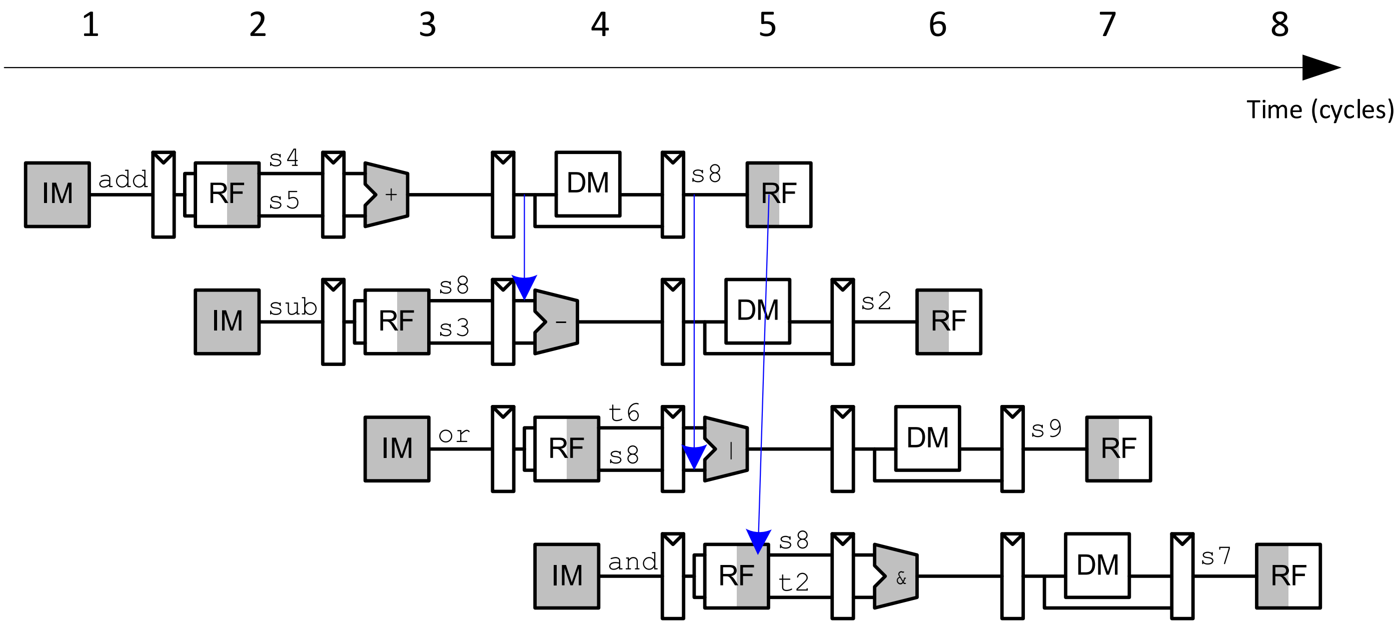
```
add s8, s4, s5

sub s2, s8, s3

or  s9, t6, s8

and s7, s8, t2
```



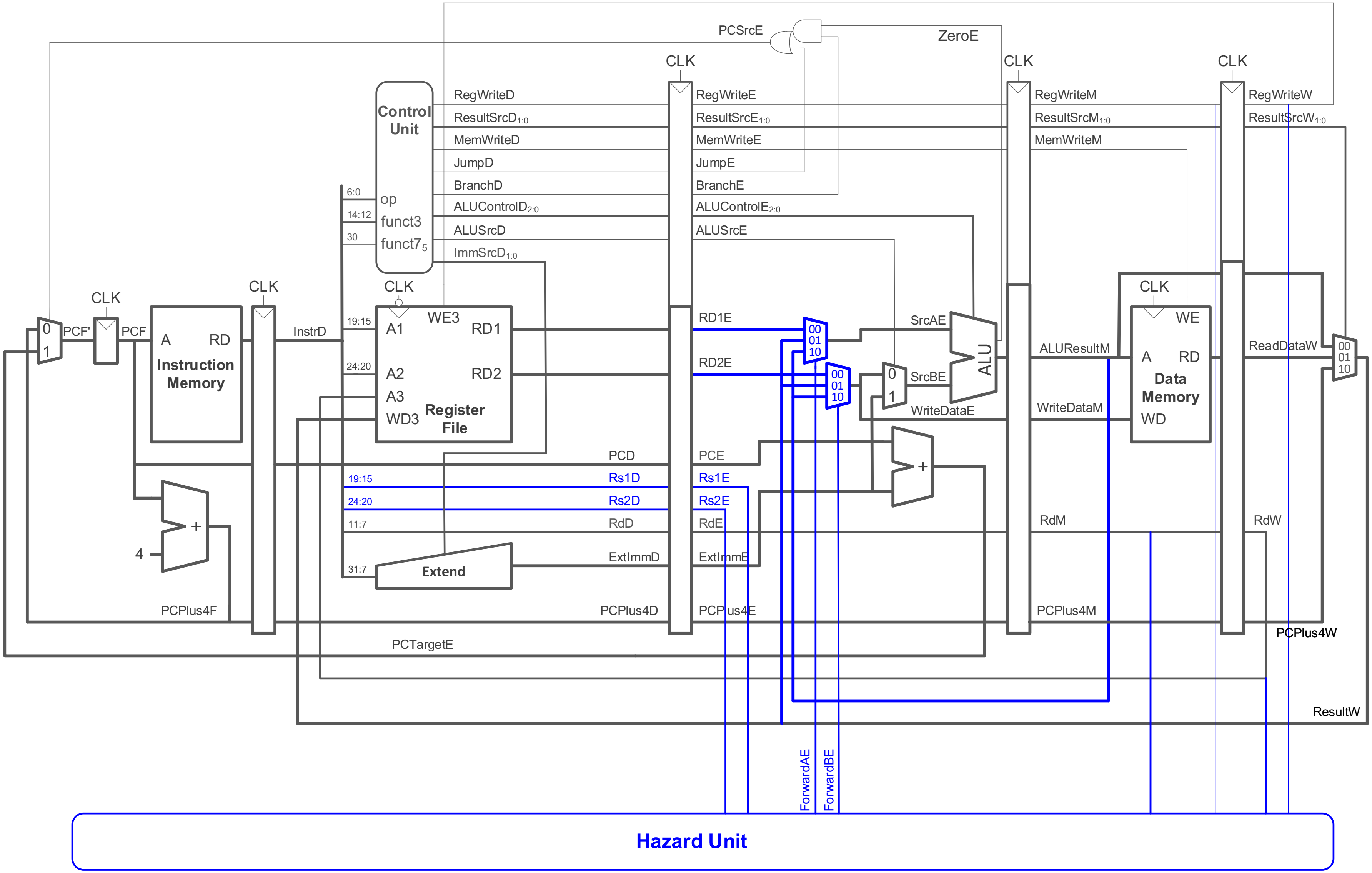
Data Forwarding

- Check if source register **in Execute stage matches** destination register of instruction **in Memory or Writeback stage**.
- If so, forward result.

```
add s8, s4, s5

sub s2, s8, s3

or  s9, t6, s8

and s7, s8, t2
```



Data Forwarding: Hazard Unit



Data Forwarding

- **Case 1: Execute** stage *Rs1* or *Rs2* matches **Memory** stage *Rd*?
  Forward from Memory stage
- **Case 2: Execute** stage *Rs1* or *Rs2* matches **Writeback** stage *Rd*?
  Forward from Writeback stage
- **Case 3:** Otherwise use value read from register file (as usual)

Equations for *Rs1*:

```
if      ((Rs1E == RdM) AND RegWriteM)          // Case 1
              ForwardAE = 10
else if ((Rs1E == RdW) AND RegWriteW)          // Case 2
              ForwardAE = 01
else          ForwardAE = 00                   // Case 3
```

***ForwardBE** equations are similar (replace **Rs1E** with **Rs2E**)*

Data Forwarding

- **Case 1: Execute** stage *Rs1* or *Rs2* matches **Memory** stage *Rd*?
  Forward from Memory stage
- **Case 2: Execute** stage *Rs1* or *Rs2* matches **Writeback** stage *Rd*?
  Forward from Writeback stage
- **Case 3:** Otherwise use value read from register file (as usual)

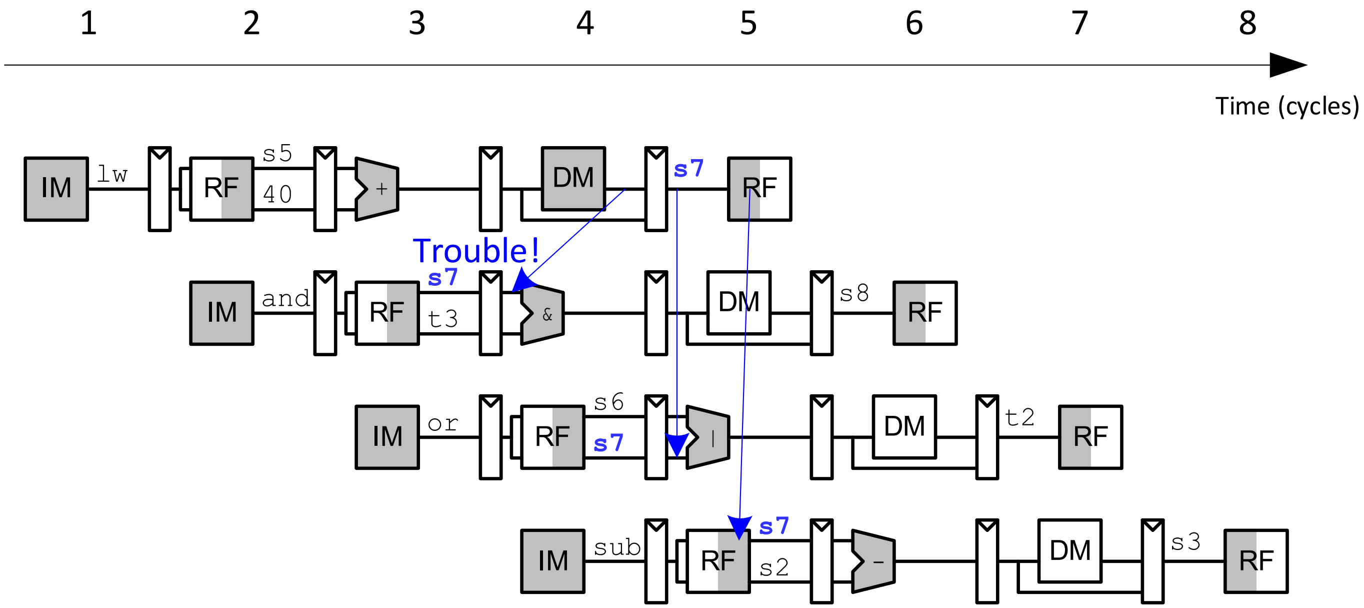
Equations for *Rs1*:

```
if       ((Rs1E == RdM) AND RegWriteM) AND (Rs1E != 0) // Case 1
                    ForwardAE = 10
else if ((Rs1E == RdW) AND RegWriteW) AND (Rs1E != 0) // Case 2
                    ForwardAE = 01
else                ForwardAE = 00                       // Case 3
```

***ForwardBE** equations are similar (replace **Rs1E** with **Rs2E**)*

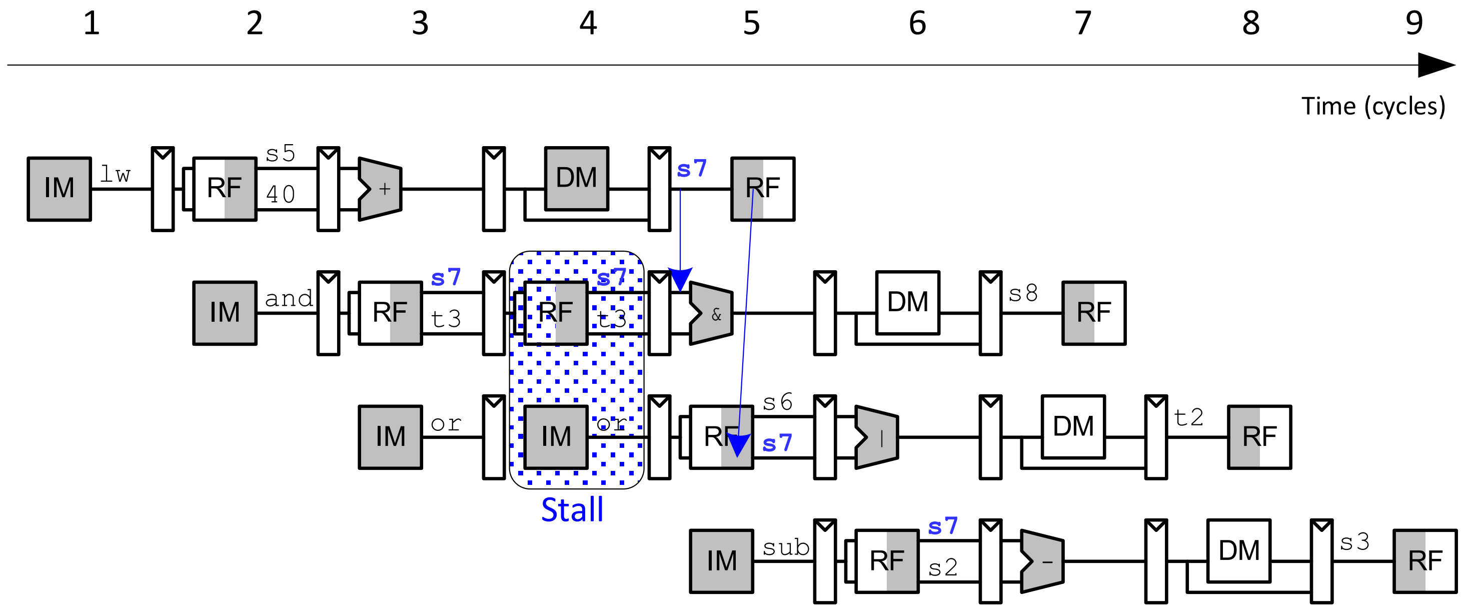
Data Hazard due to `lw` Dependency

```
lw   s7, 40(s5)
and s8, s7, t3
or   t2, s6, s7
sub s3, s7, s2
```



Stalling to solve `lw` Data Dependency

```
lw   s7, 40(s5)

and  s8, s7, t3

or   t2, s6, s7

sub  s3, s7, s2
```



Stalling Logic

- Is either **source register in the Decode stage** the same as the **destination register in the Execute stage**?

  AND

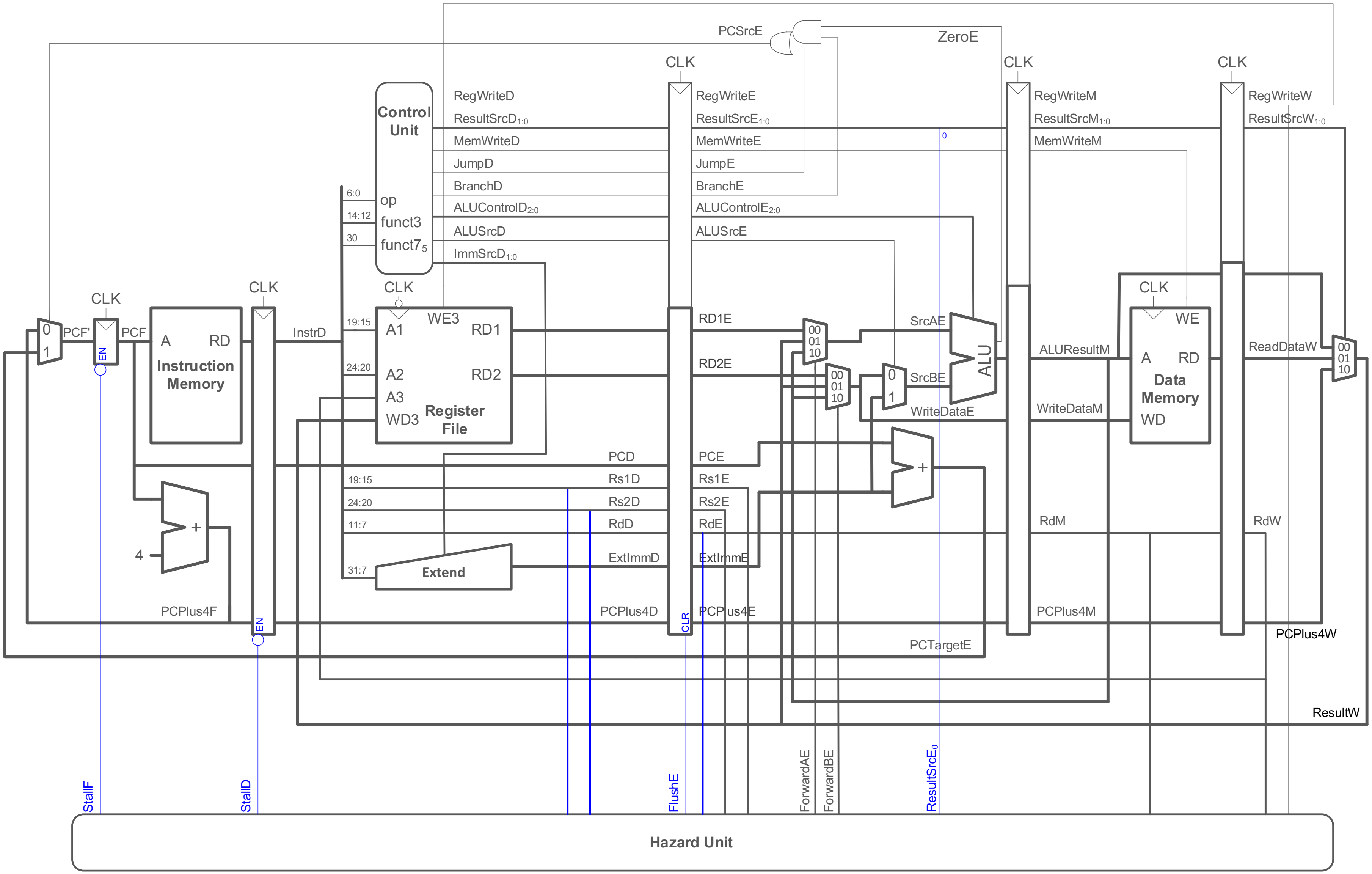
- Is the instruction in the **Execute stage a `lw`**?

lwStall = ((*Rs1D* == *RdE*) OR (*Rs2D* == *RdE*)) **AND** ***ResultSrcE***$_{0}$

StallF = ***StallD*** = ***FlushE*** = ***lwStall***

(Stall the Fetch and Decode stages, and flush the Execute stage.)

Stalling Hardware



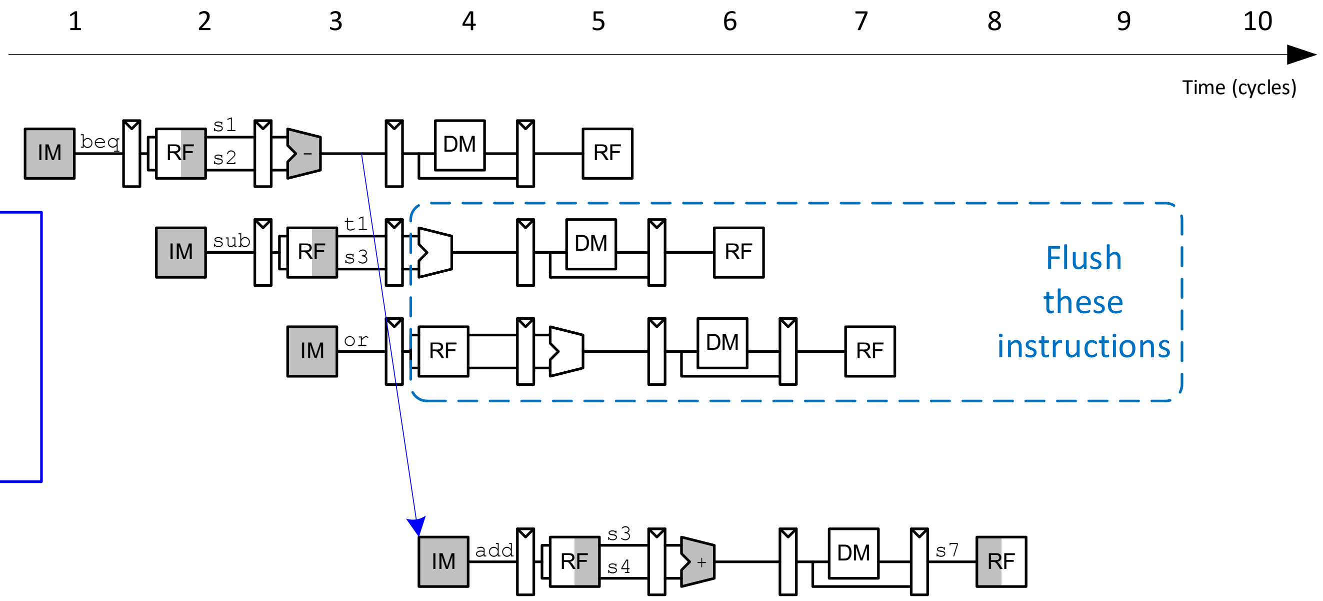
Pipelined Processor Control Hazards

Control Hazards

- `beq`:
  - Branch **not determined until the Execute stage** of pipeline
  - **Instructions** after branch **fetched** before branch occurs
  - These **2 instructions must be flushed** if branch happens

Control Hazards

```
20        beq s1, s2, L1

24        sub s8, t1, s3

28        or  s9, t6, s5

2C        ...
...       ...

58    L1: add s7, s3, s4
```



Flush these instructions

Branch misprediction penalty:

The number of instructions flushed when a branch is taken (in this case, 2 instructions)

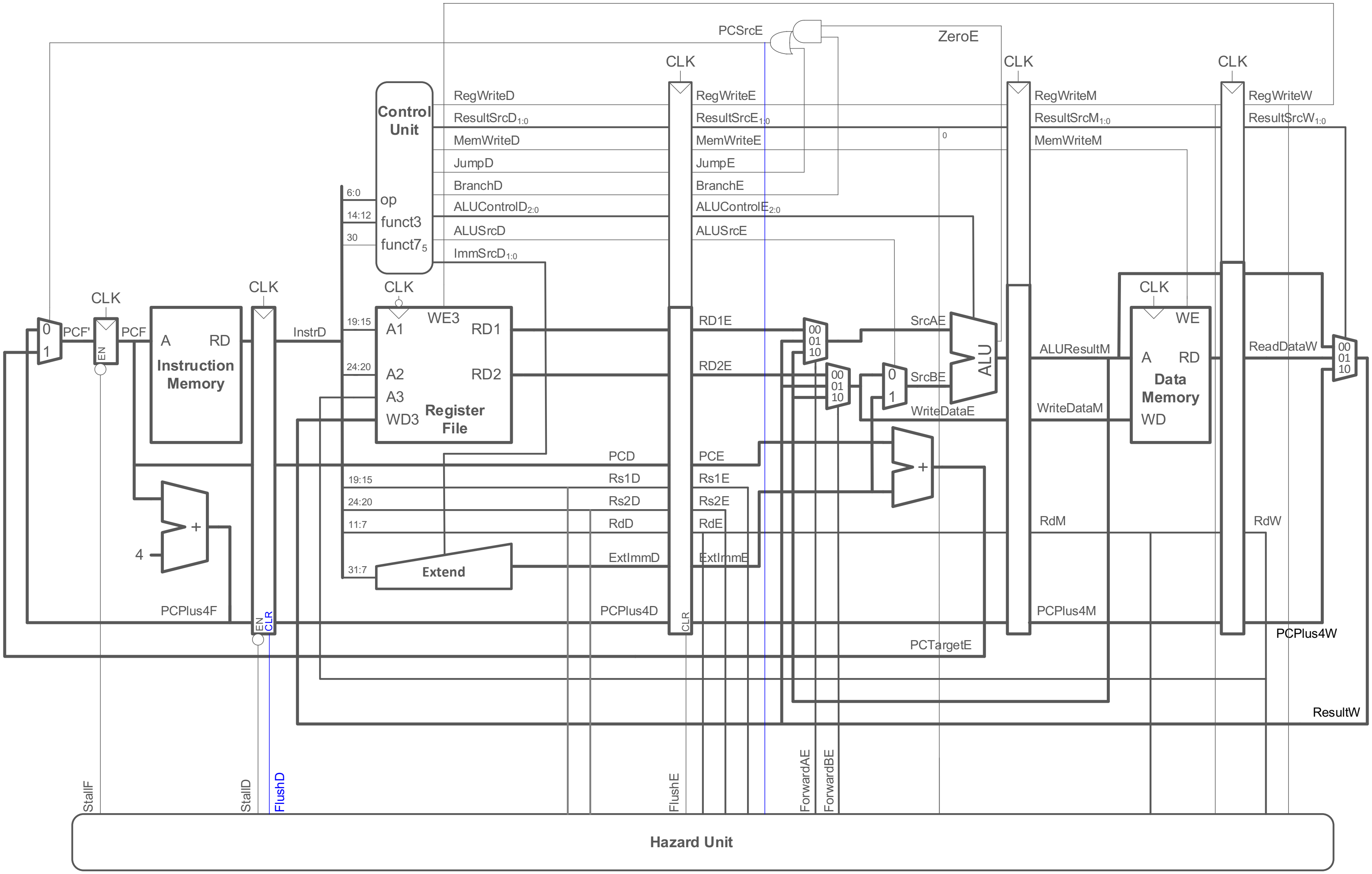
Control Hazards: Flushing Logic

- If branch is taken in execute stage, need to flush the instructions in the Fetch and Decode stages
  - Do this by clearing Decode and Execute Pipeline registers using *FlushD* and *FlushE*
- **Equations:**

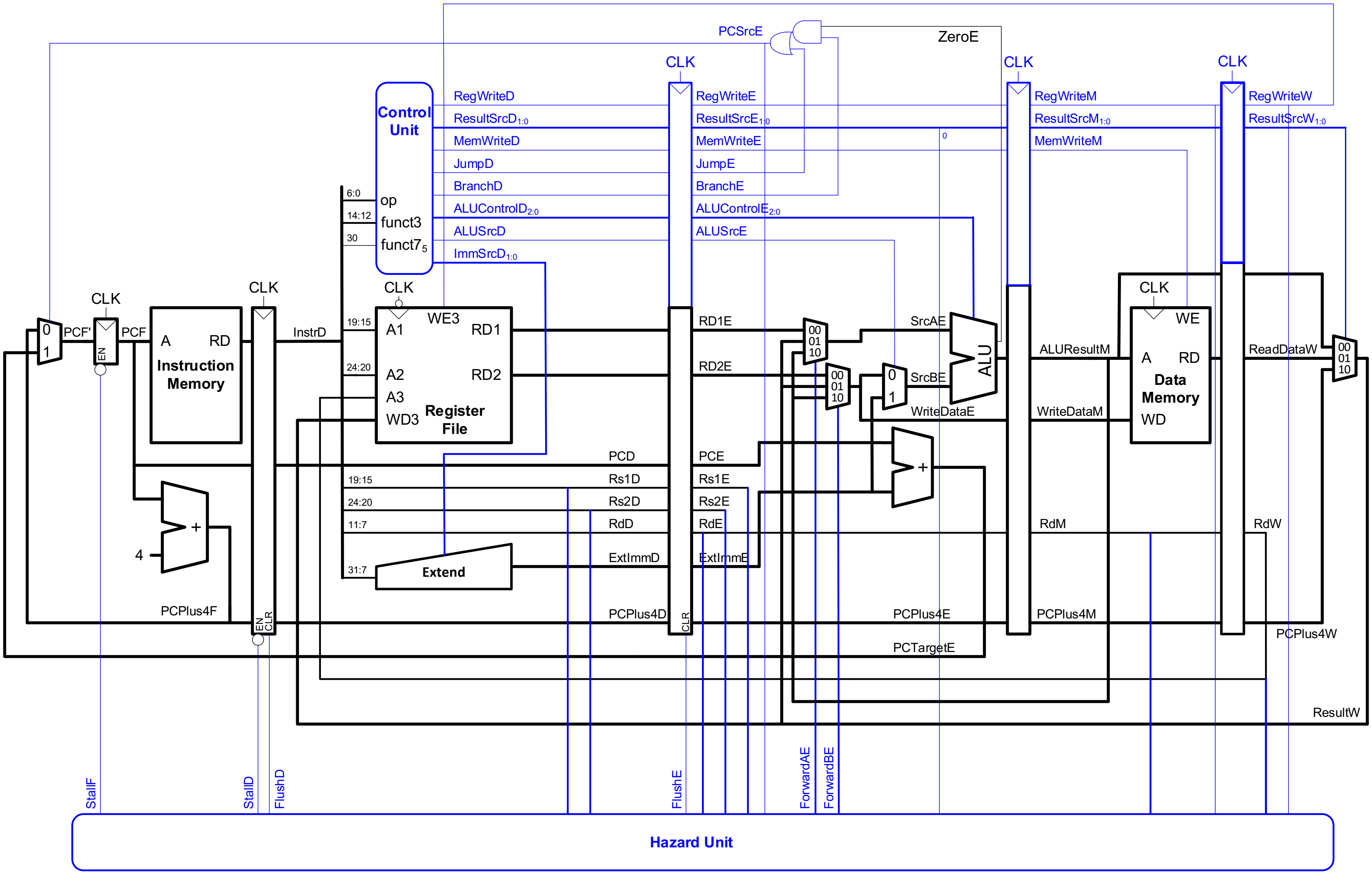
FlushD = PCSrcE

FlushE = *lwStall* OR *PCSrcE*

Control Hazards: Flushing Hardware



RISC-V Pipelined Processor with Hazard Unit



Summary of Hazard Logic

Data hazard logic (shown for SrcA of ALU):

```
if      ((Rs1E == RdM) AND RegWriteM) AND (Rs1E != 0)     // Case 1
                    ForwardAE = 10
else if ((Rs1E == RdW) AND RegWriteW) AND (Rs1E != 0)     // Case 2
                    ForwardAE = 01
else                ForwardAE = 00                        // Case 3
```

Load word stall logic:

lwStall = ((*Rs1D* == *RdE*) OR (*Rs2D* == *RdE*)) AND $ResultSrcE_0$

StallF = ***StallD*** = *lwStall*

Control hazard flush:

FlushD = *PCSrcE*

FlushE = *lwStall* OR *PCSrcE*

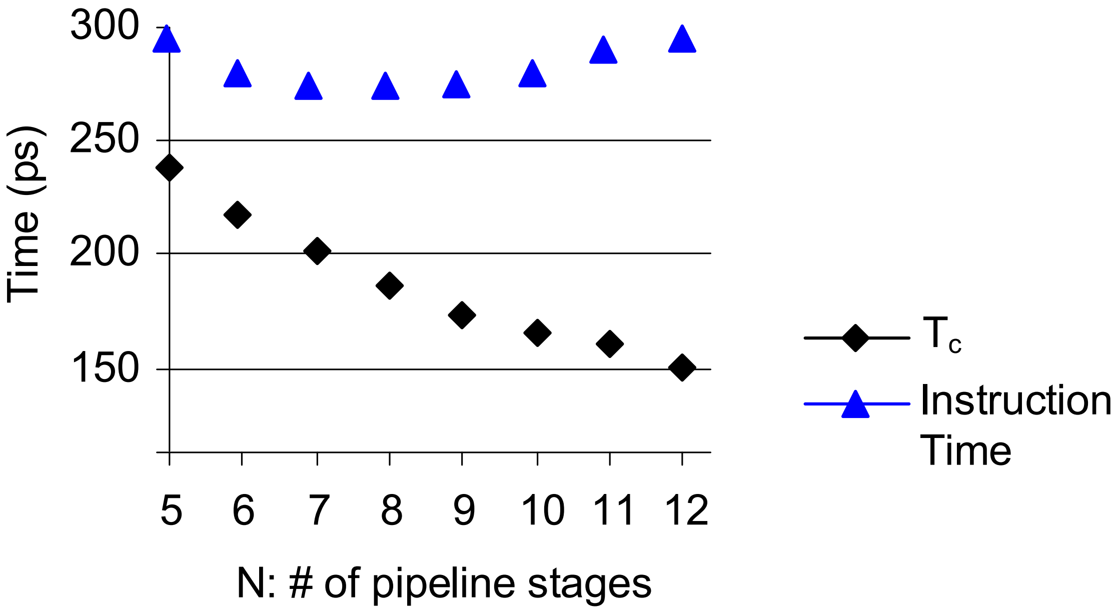
Advanced Microarchitecture

Advanced Microarchitecture

- Deep Pipelining
- Micro-operations
- Branch Prediction
- Superscalar Processors
- Out of Order Processors
- Register Renaming
- SIMD
- Multithreading
- Multiprocessors

Deep Pipelining

- **10-20 stages typical**
- Number of stages limited by:
  - – Pipeline hazards
  - – Sequencing overhead
  - – Power
  - – Cost



Micro-operations

- Decompose complex instructions into series of simple instructions called ***micro-operations*** (*micro-ops* or *μ-ops*)
- **At run-time**, complex instructions are decoded into one or more micro-ops
- Used heavily in **CISC** (complex instruction set computer) architectures (e.g., x86)

Complex Op	**Micro-op Sequence**

`lw s1, 0(s2), postincr 4`	`lw   s1, 0(s2)`
	`addi s2, s2, 4`

Without μ-ops, would need 2nd write port on the register file

Branch Prediction

- **Guess** whether branch will be taken
  - Backward branches are usually taken (loops)
  - Consider history to improve guess
- Good prediction **reduces fraction of branches requiring a flush**

Branch Prediction

- Ideal pipelined processor: CPI = 1
- Branch misprediction increases CPI
- **Static branch prediction:**
  - Check direction of branch (forward or backward)
  - If backward, predict taken
  - Else, predict not taken
- **Dynamic branch prediction:**
  - Keep **history** of last several hundred (or thousand) branches in *branch target buffer*, record:
    - Branch destination
    - Whether branch was taken

Dynamic Branch Prediction

- 1-bit branch predictor
- 2-bit branch predictor

Branch Prediction Example

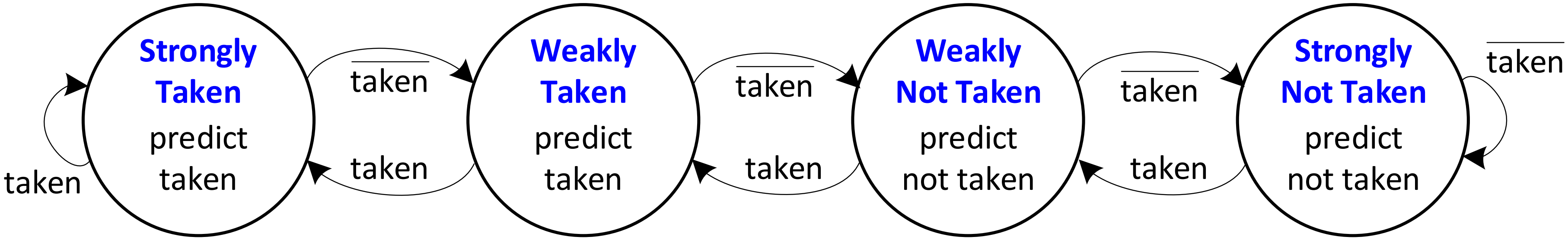
```
  addi s1, zero, 0        # s1 = sum
  addi s0, zero, 0        # s0 = i
  addi t0, zero, 10       # t0 = 10

For:                      # for (i=0; i<10; i=i+1)
  bge  s0, t0, Done
  add  s1, s1, s0         # sum = sum + i
  addi s0, s0, 1          # i = i + 1
  j    For

Done:
```

1-Bit Branch Predictor

- **Remembers** whether branch was taken the last time and **does the same thing**
- Mispredicts first and last branch of loop

```
  addi s1, zero, 0      # s1 = sum
  addi s0, zero, 0      # s0 = i
  addi t0, zero, 10     # t0 = 10

For:                    # for (i=0; i<10; i=i+1)
  bge  s0, t0, Done
  add  s1, s1, s0       # sum = sum + i
  addi s0, s0, 1        # i = i + 1
  j    For

Done:
```

2-Bit Branch Predictor



taken

Strongly Taken
predict taken

$\overline{\text{taken}}$

taken

Weakly Taken
predict taken

$\overline{\text{taken}}$

taken

Weakly Not Taken
predict not taken

$\overline{\text{taken}}$

taken

Strongly Not Taken
predict not taken

$\overline{\text{taken}}$

```
   addi s1, zero, 0       # s1 = sum
   addi s0, zero, 0       # s0 = i
   addi t0, zero, 10      # t0 = 10

 For:                     # for (i=0; i<10; i=i+1)
   bge  s0, t0, Done
   add  s1, s1, s0        # sum = sum + i
   addi s0, s0, 1         # i = i + 1
   j    For

 Done:
```

Only mispredicts **last branch** of loop