EECE 2322: Fundamentals of Digital Design and Computer Organization

Lecture 14_3: Microarchitecture

Xiaolin Xu
Department of ECE
Northeastern University

Control Hazards

- **`beq`:**
  - branch not determined until 4th stage of pipeline
  - Instructions after branch fetched before branch occurs
  - These instructions must be flushed if branch happens

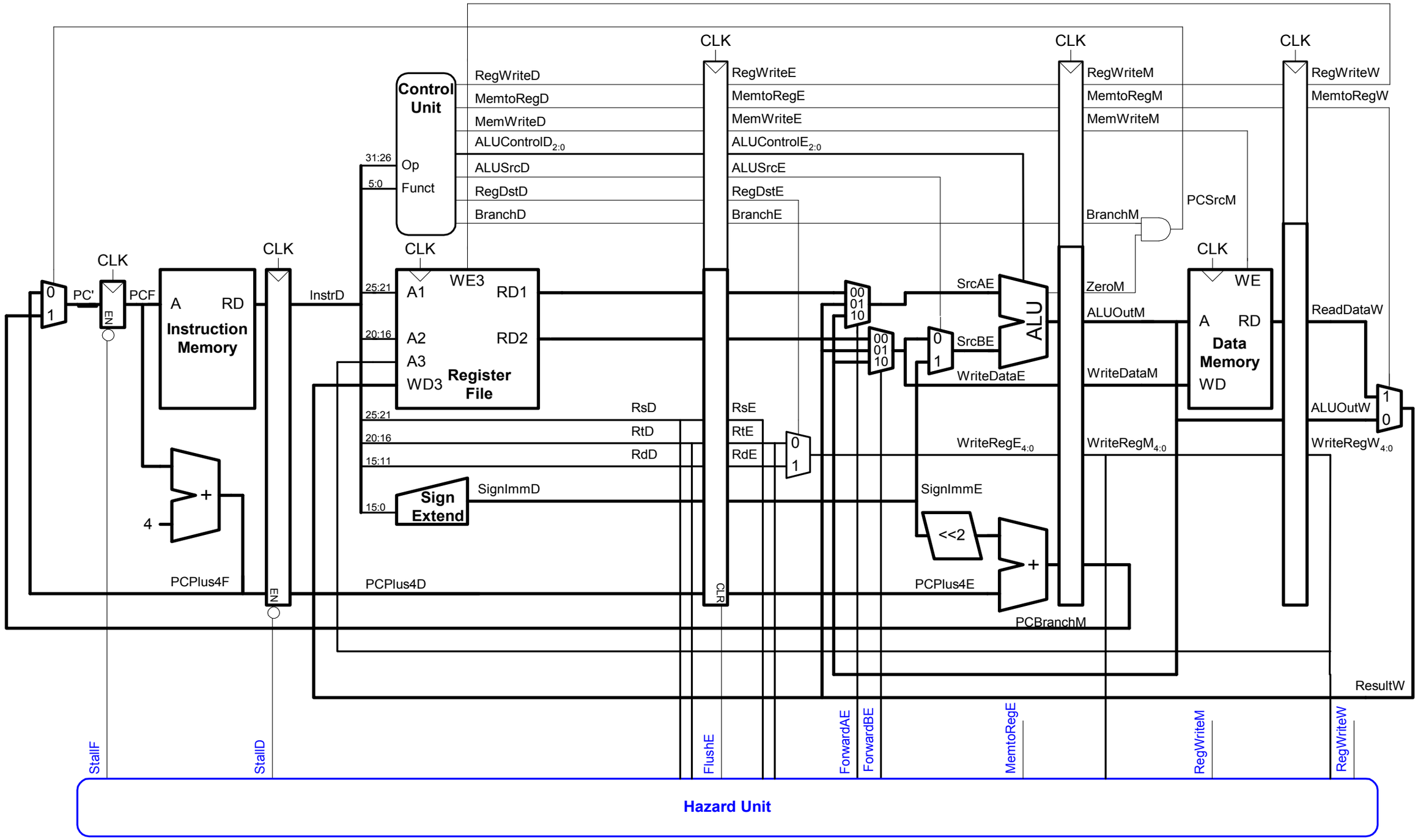
Control Hazards

- **`beq`:**
  - branch not determined until 4th stage of pipeline
  - Instructions after branch fetched before branch occurs
  - These instructions must be flushed if branch happens
- **naive solution:**
  - stall all pipelines until knowing which branch taken
  - Too slow —> memory stage —> stall pipeline for 3 cycles!

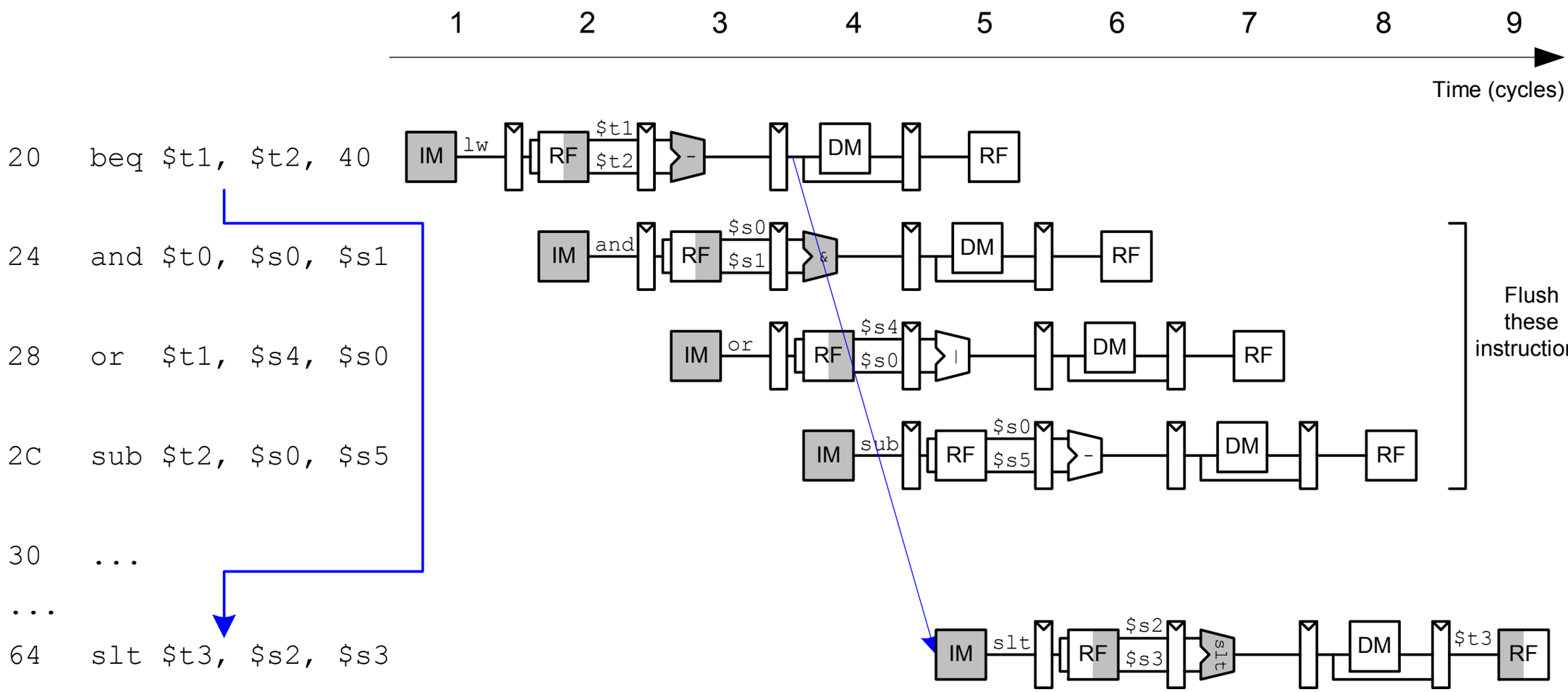
Control Hazards

- **Branch misprediction penalty**
  - number of instruction flushed when branch is taken
  - May be reduced by determining branch earlier

Control Hazards: Original Pipeline

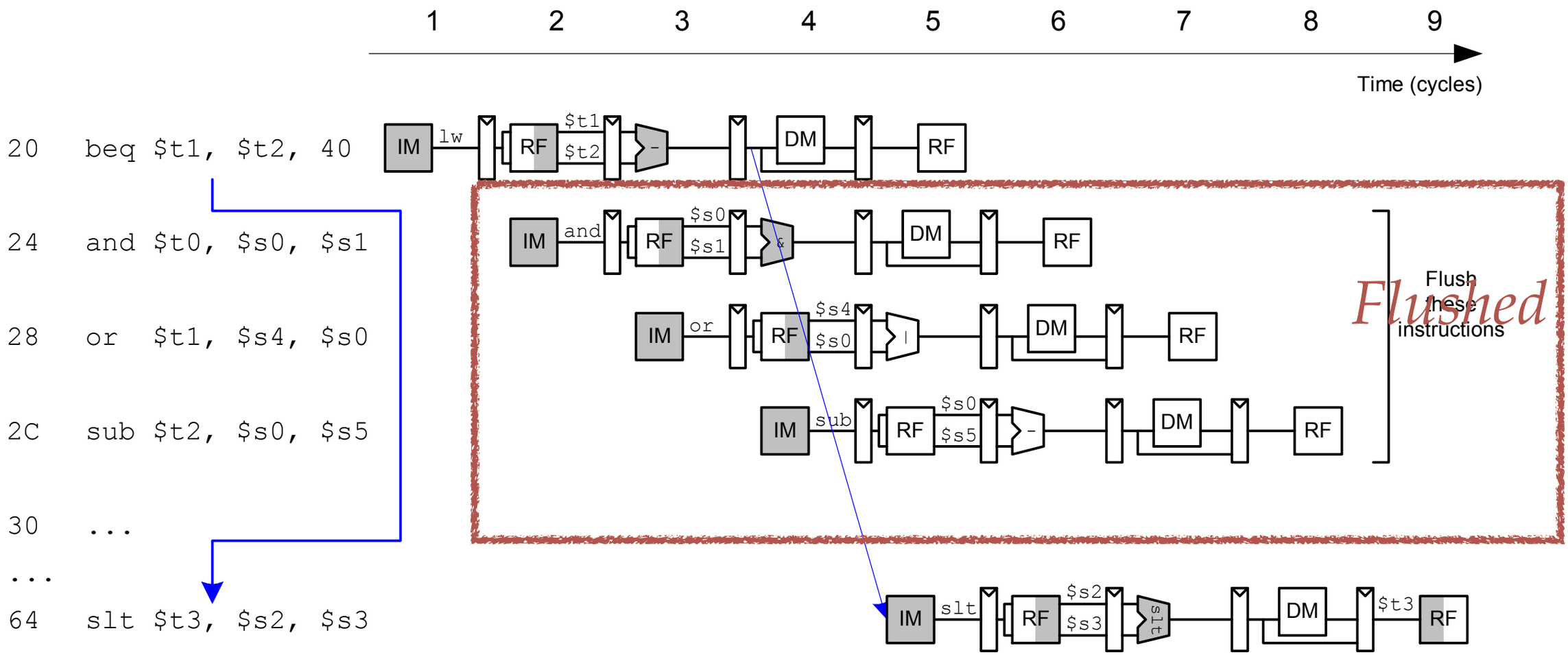


Control Hazards: An Improved Solution



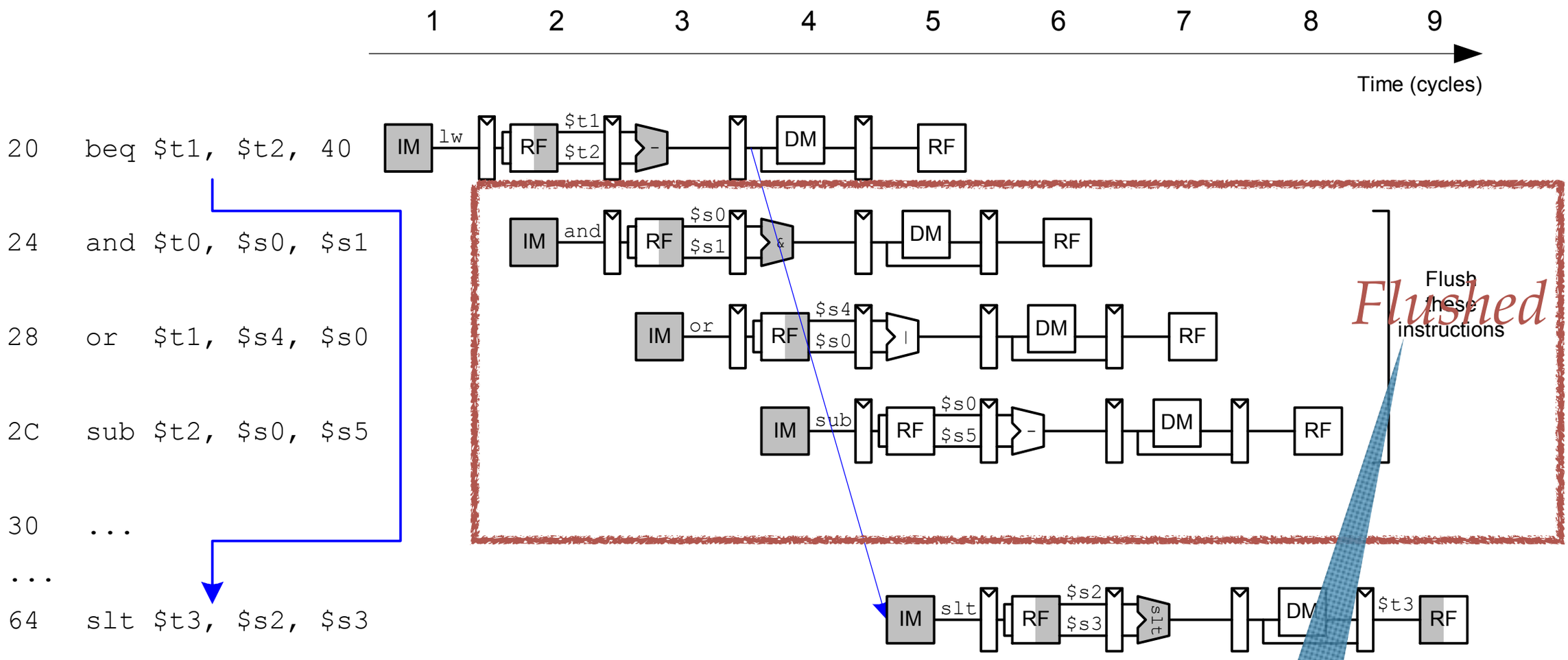
- Assuming branch is NOT taken
  - Decision of beq at #20 can be known in Cycle 4
  - Three instructions (24, 28, 2C) fetched by then
  - Fetch *slt* in Cycle 5 —> What will happen?

Control Hazards: An Improved Solution



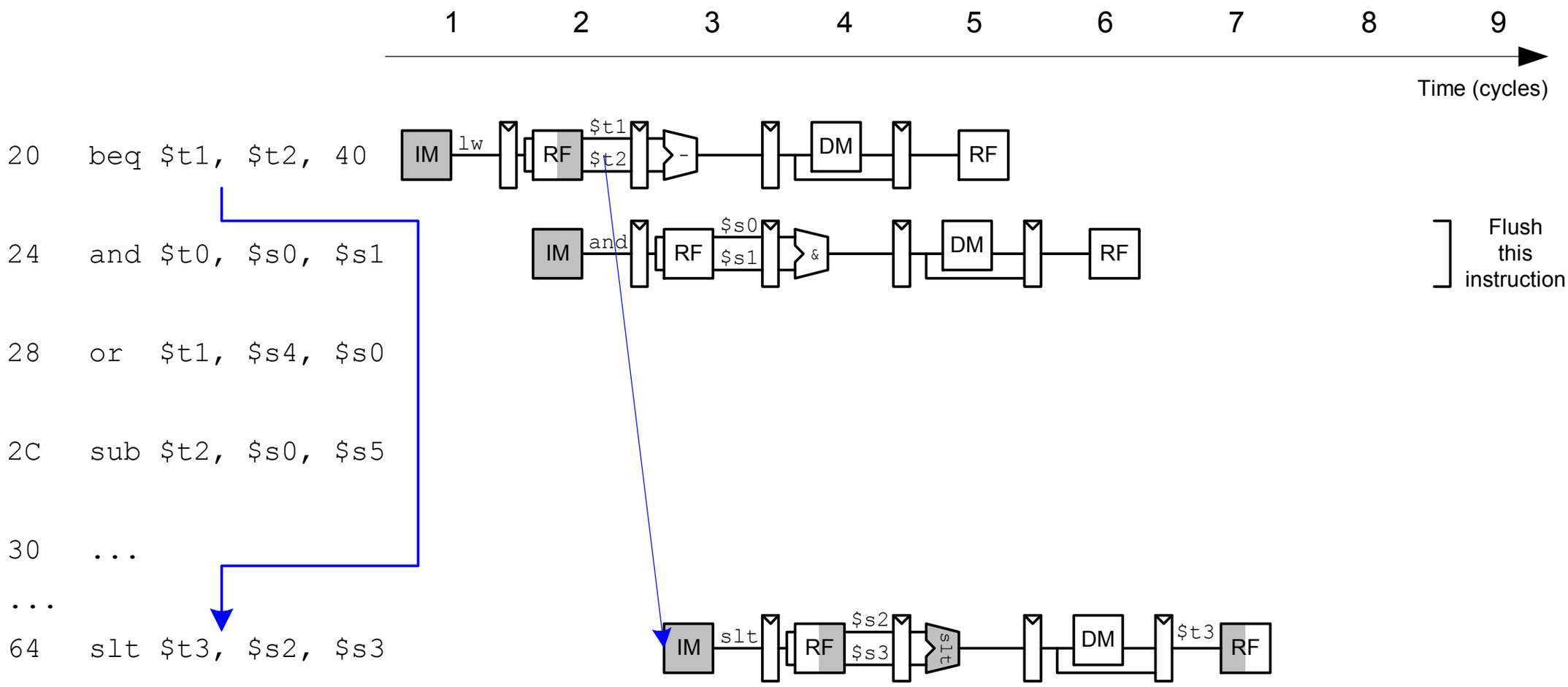
- Assuming branch is NOT taken
  - Decision of beq at #20 can be known in Cycle 4
  - Three instructions (24, 28, 2C) fetched by then
  - Fetch *slt* in Cycle 5 —> What will happen?

Control Hazards: An Improved Solution



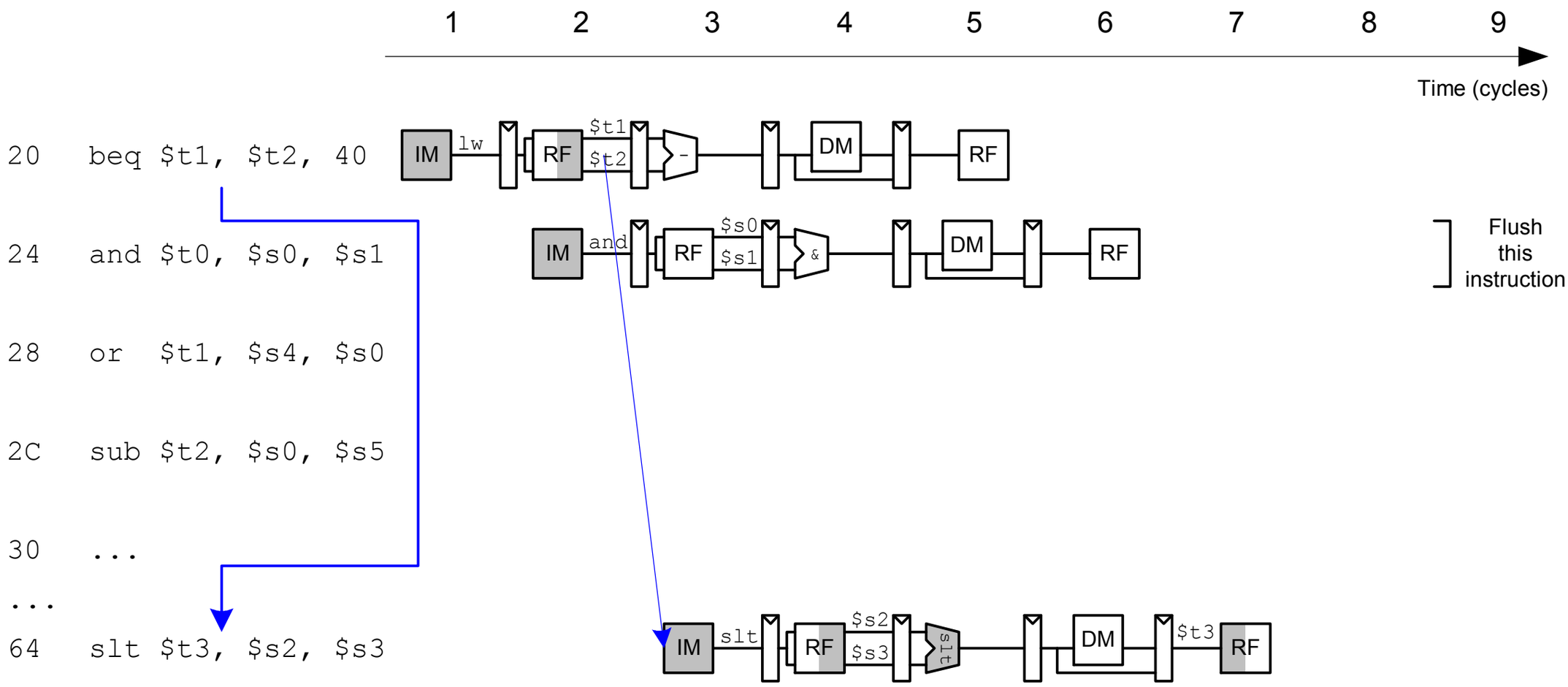
- Assuming branch is NOT taken
  - Decision of beq at #20 can be known in Cycle 4
  - Three instructions (24, 28, 2C) fetched by then
  - Fetch *slt* in Cycle 5 —> What will happen?

Early Branch Resolution



- How early can we push back?

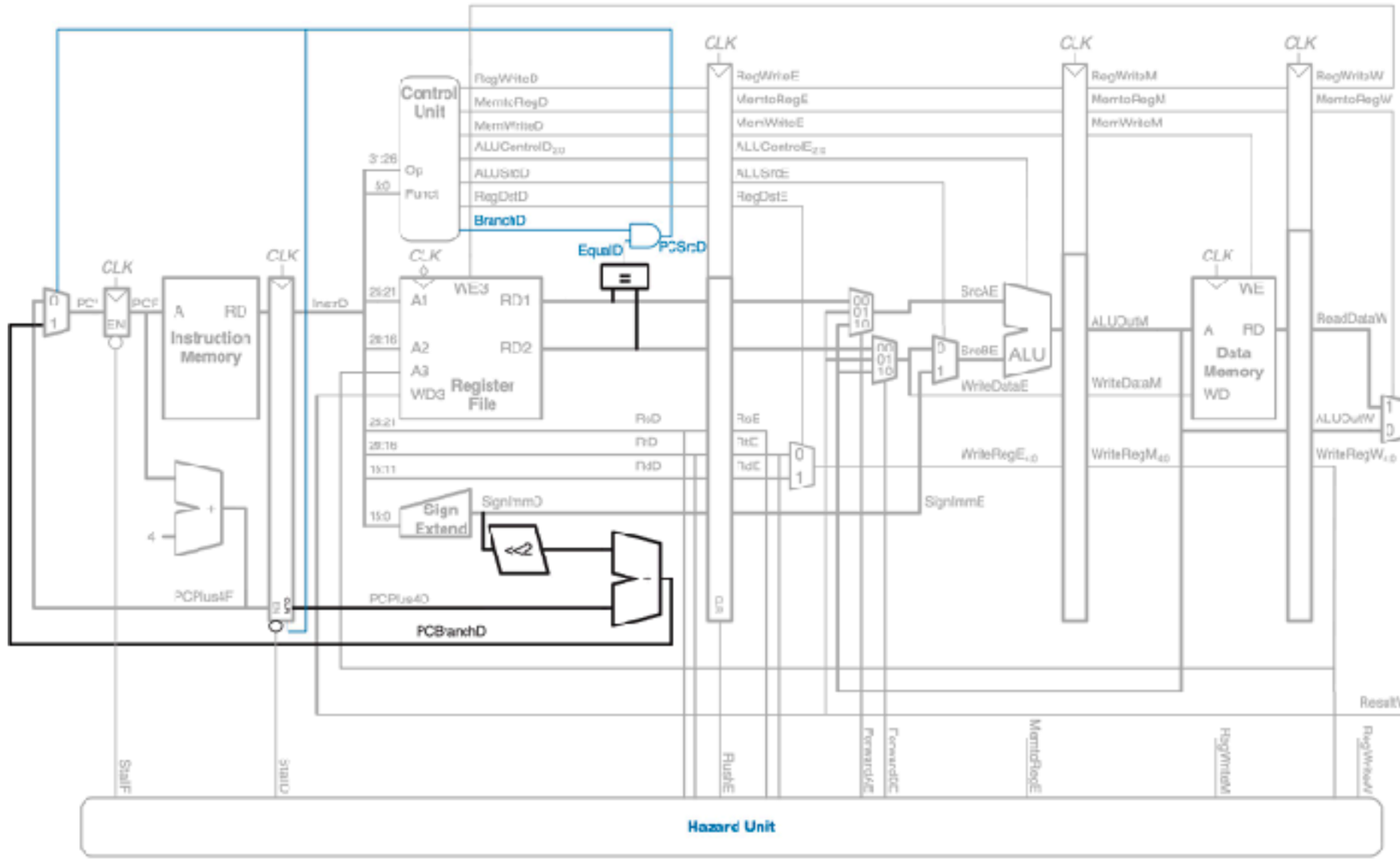
Early Branch Resolution



- How early can we push back?
- Decide earlier
  - Original: ALU
  - Dedicated hardware setup —> equality comparator

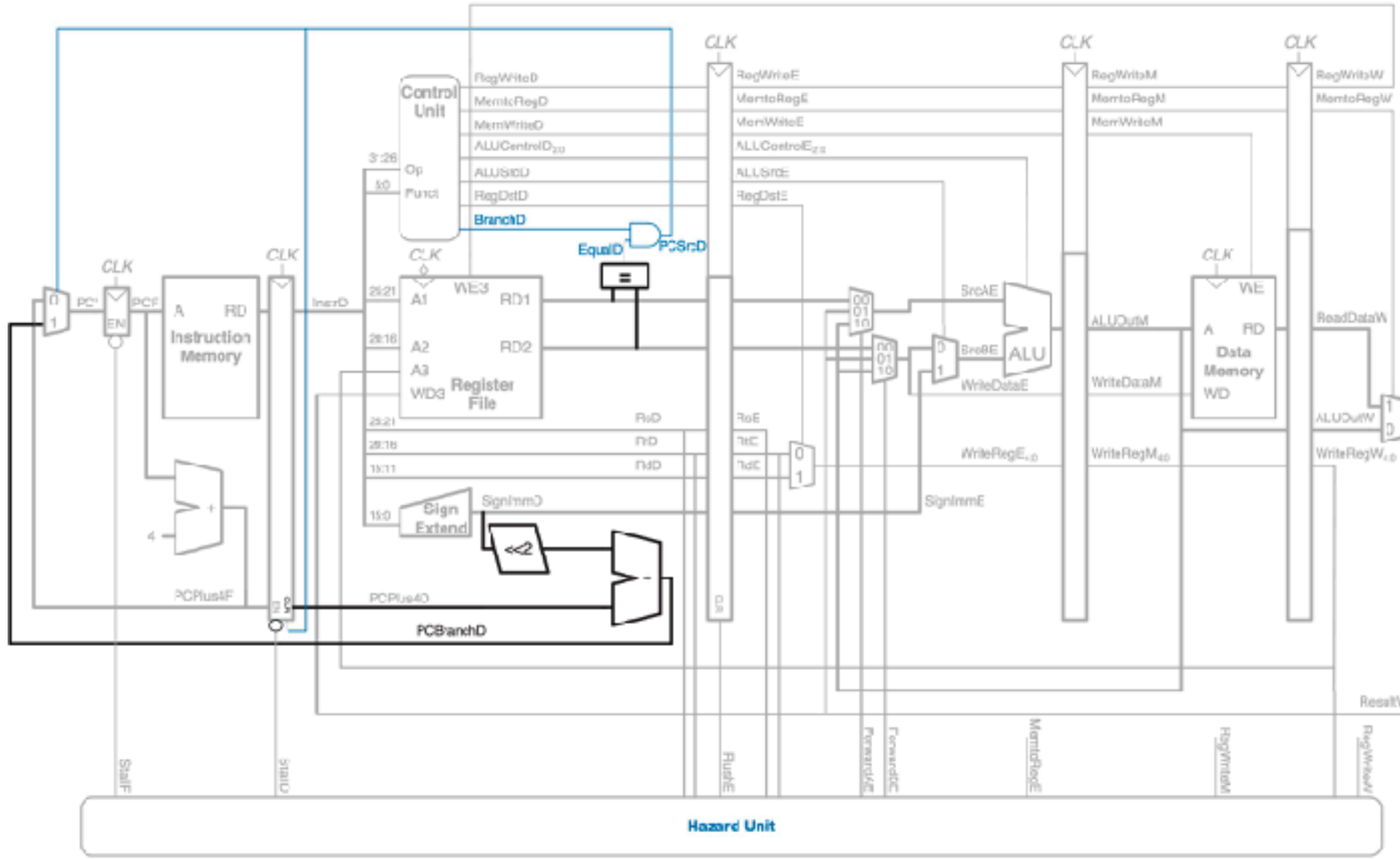
Early Branch Resolution

- Equality compactor added in the Decode stage
- *PCSrc* AND gate moved earlier
  - Now Decode stage in stead of Memory stage



Early Branch Resolution

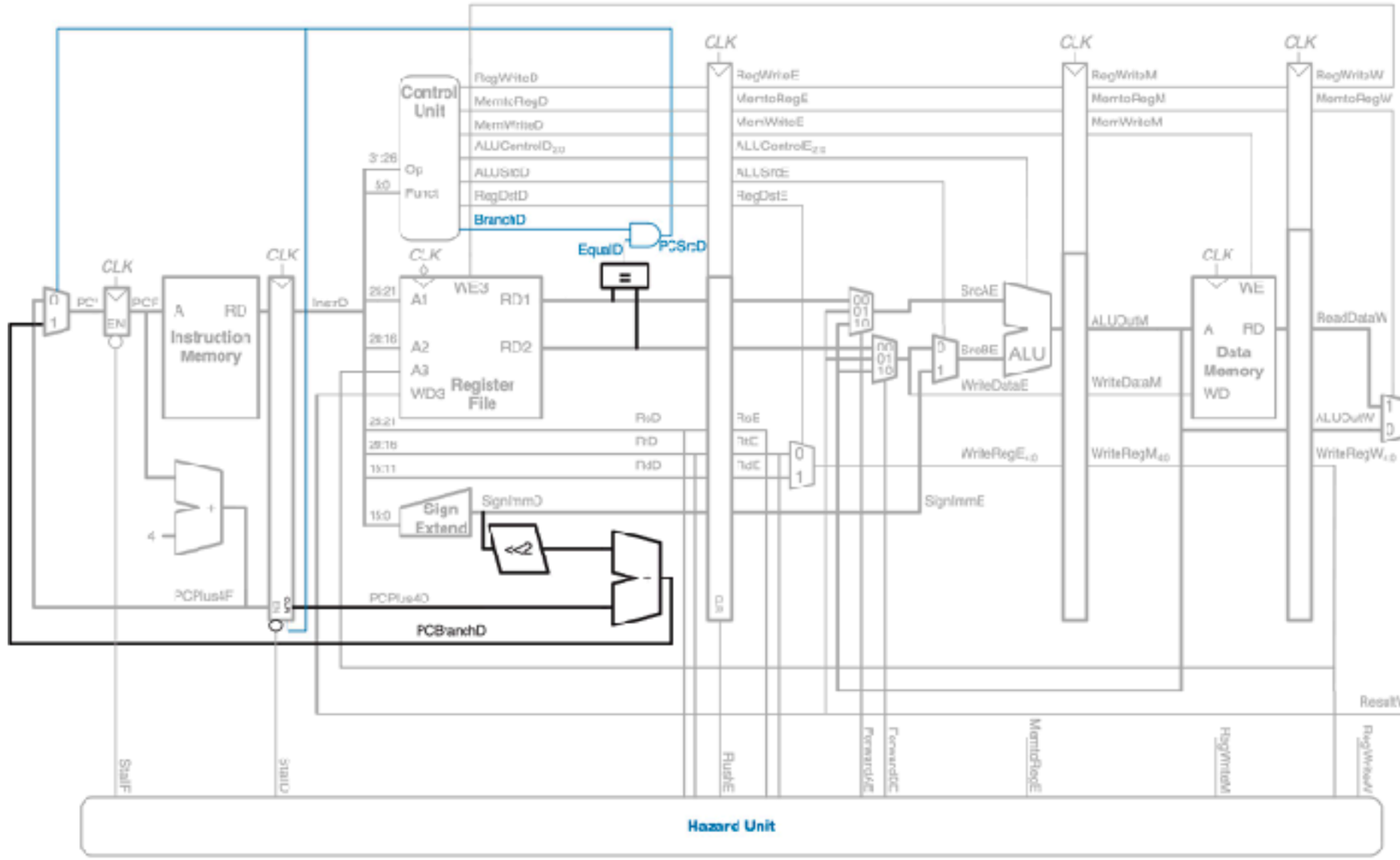
- Equality compactor added in the Decode stage
- *PCSrc* AND gate moved earlier
  - Now Decode stage in stead of Memory stage
- Good?



Early Branch Resolution

- Equality compactor added in the Decode stage
- *PCSrc* AND gate moved earlier
  - Now Decode stage in stead of Memory stage
- Good?

Introduced another data hazard in Decode stage

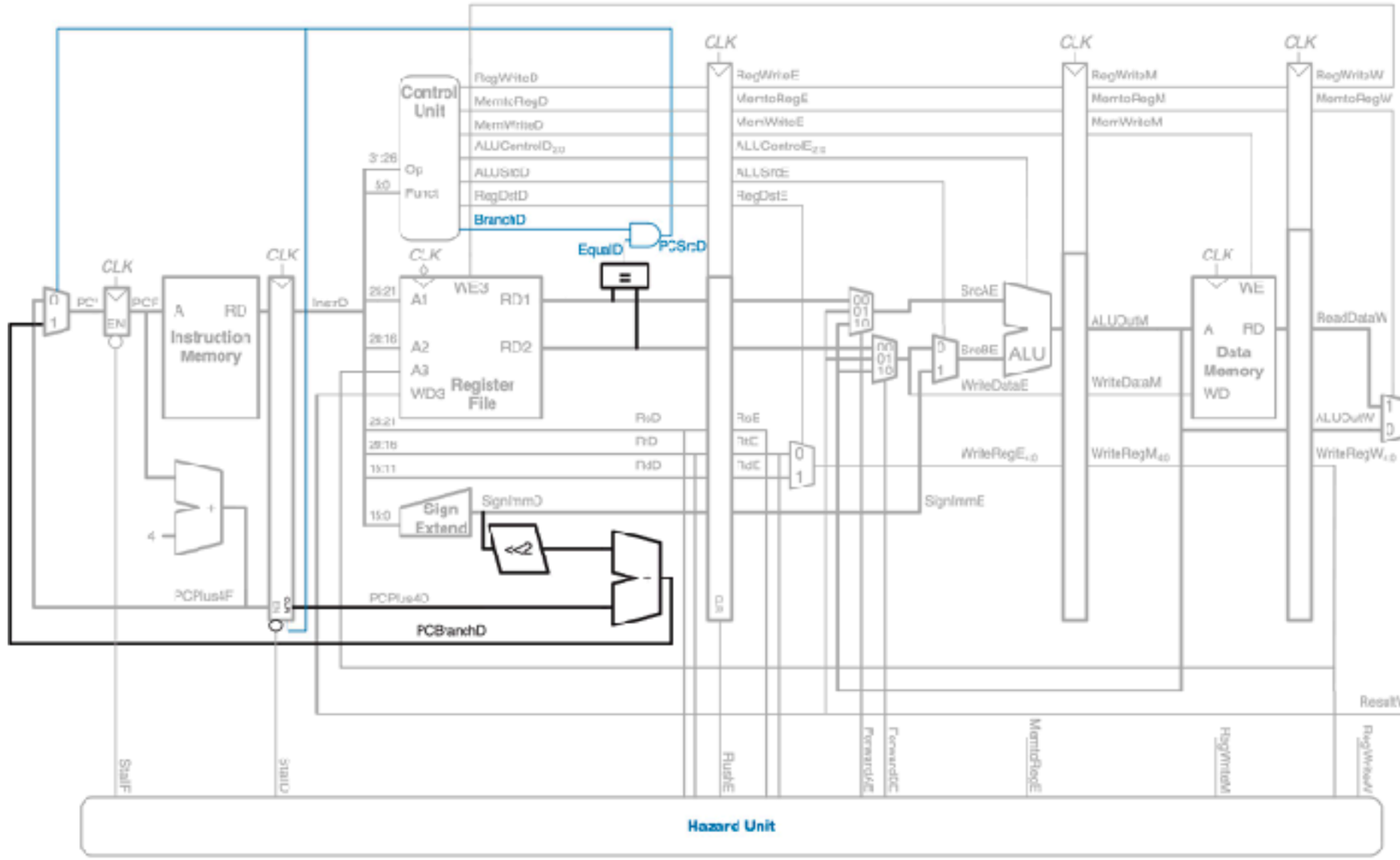


Early Branch Resolution

- Equality compactor added in the Decode stage
- *PCSrc* AND gate moved earlier
  - Now Decode stage in stead of Memory stage
- Good?

Introduced another data hazard in Decode stage

What if the source operand(s) of the branch is dependent on others?



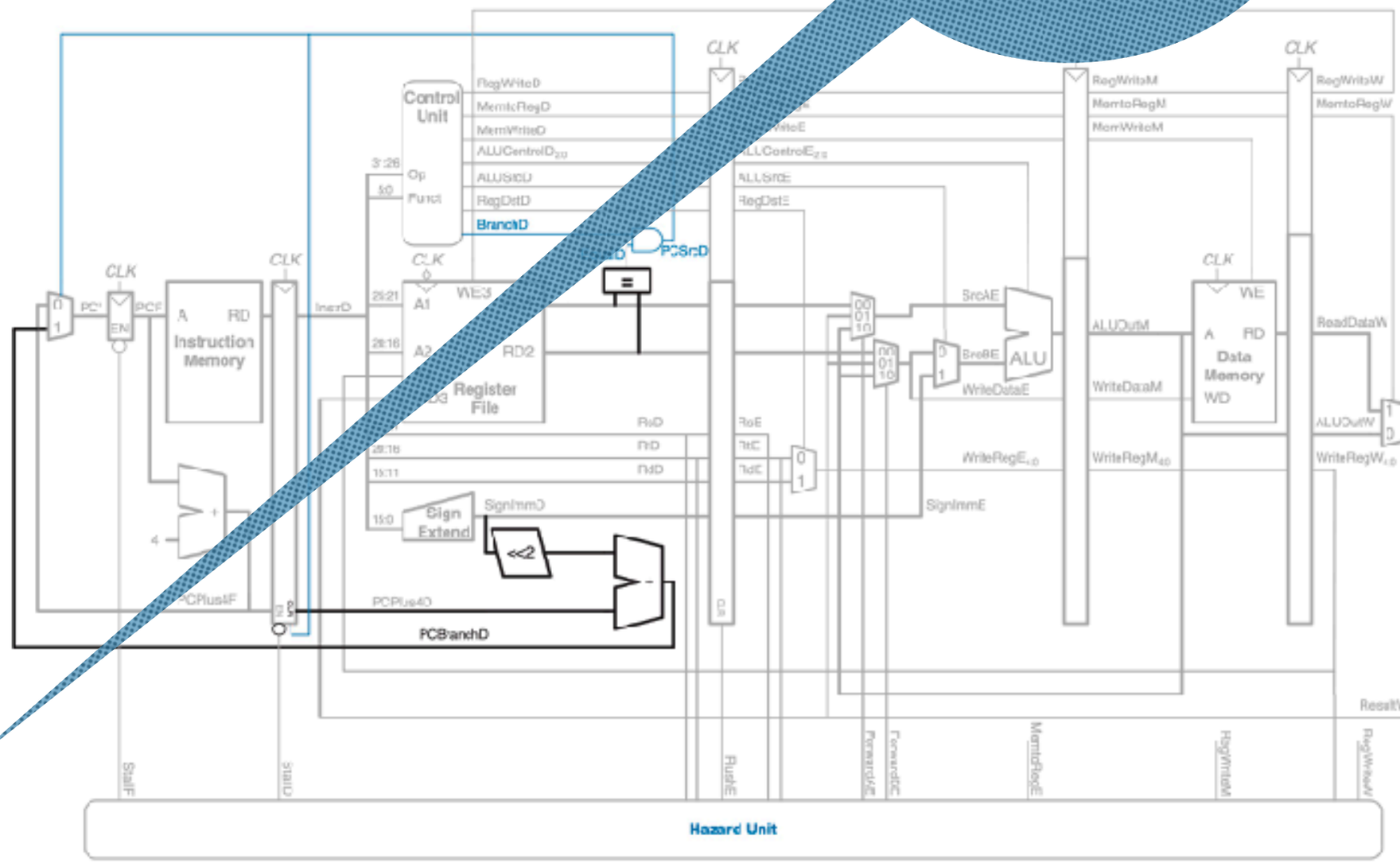
Early Branch Resolution

- Equality compactor added in the Decode stage
- *PCSrc* AND gate moved earlier
  - Now Decode stage in stead of Memory stage
- Good?

Introduced another data hazard in Decode stage

What if the source operand(s) of the branch is dependent on others?

Decode stage data dependency



Decode Stage Data Dependency Hazard

- Forwarding (available) + Stall (when ready)
- Case 1: the result is currently in the Writeback stage
  - No hazard, writing 1st half, reading 2nd half

Decode Stage Data Dependency Hazard

- Forwarding (available) + Stall (when ready)
- Case 2: the result is currently in the Memory stage
  - Forwarding to the equality comparator (new HW needed)

- **Forwarding logic:**

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

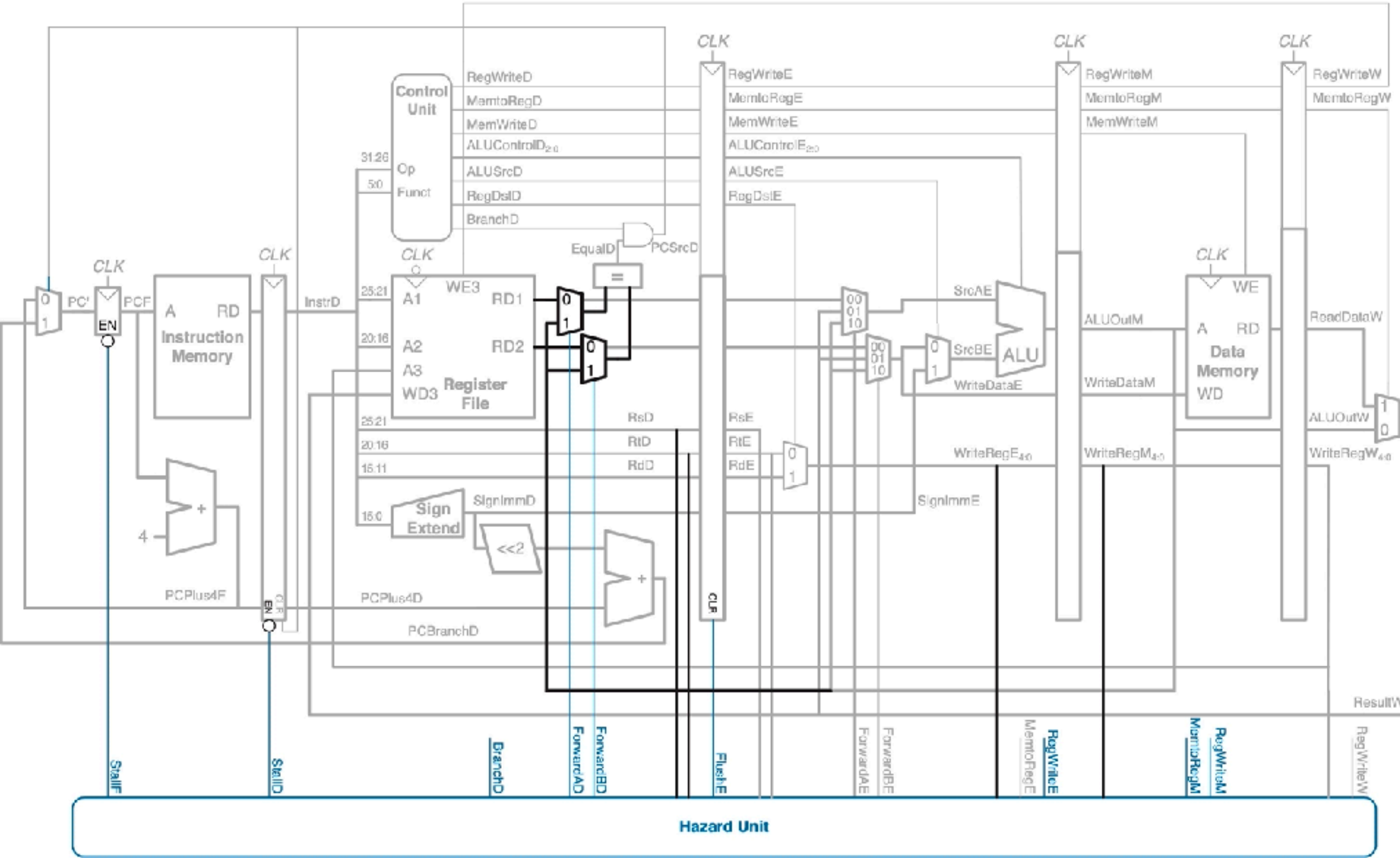
Decode Stage Data Dependency Hazard

- Forwarding (available) + Stall (when ready)
- Case 3: the result is currently in the Execution stage
  - Must stall
- Case 4: *lw* instruction, and the result is currently in the Memory stage
  - Must stall
- **Stalling logic:**

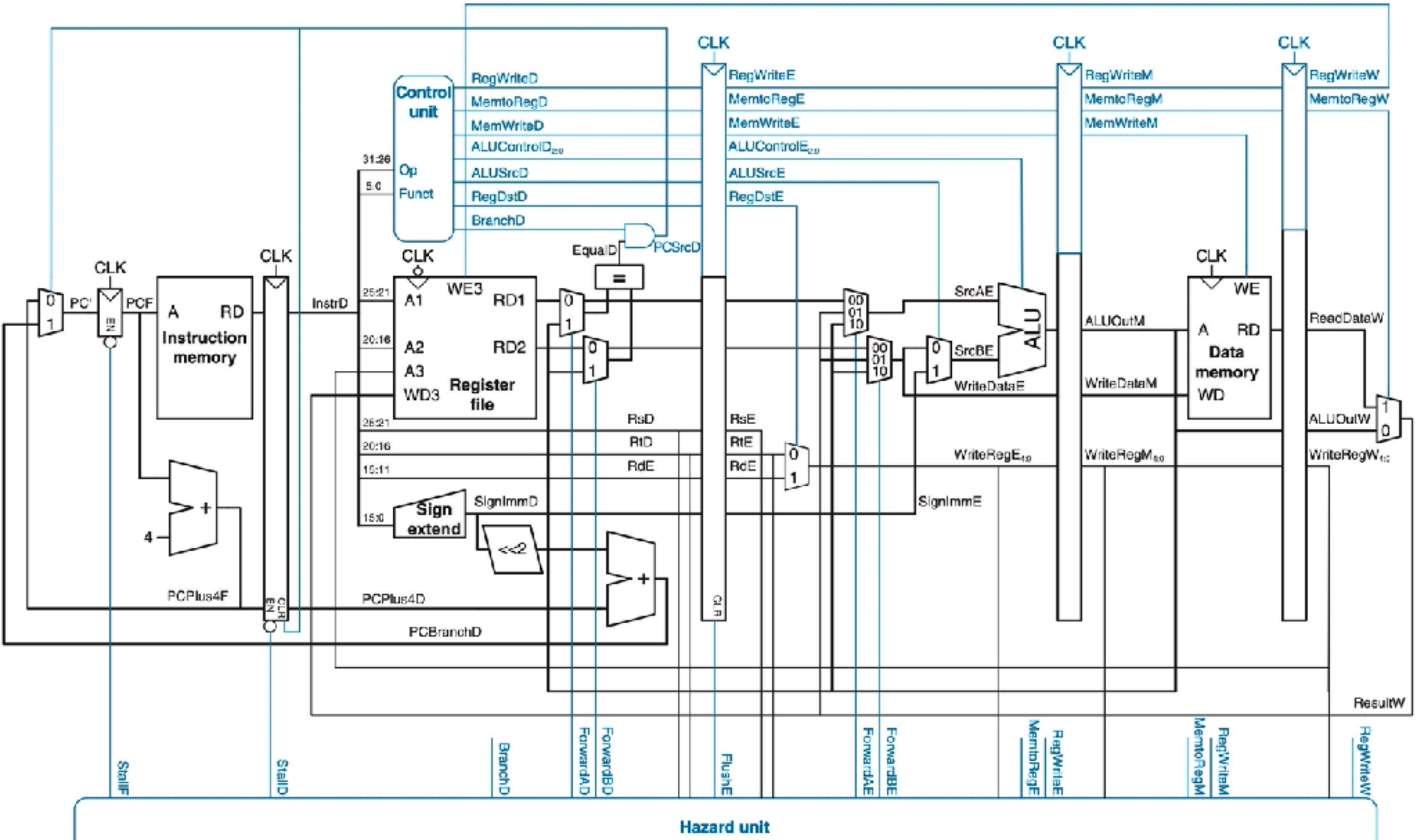
```
branchstall = BranchD AND RegWriteE AND
                  (WriteRegE == rsD OR WriteRegE == rtD)
              OR
        BranchD AND MemtoRegM AND
                  (WriteRegM == rsD OR WriteRegM == rtD)


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

Handling Control Hazards



Full Hazard Handling



More about 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

Branch Prediction

- Ideal pipelined processor: CPI = 1

Branch Prediction

- Ideal pipelined processor: CPI = 1
- Branch misprediction increases CPI

Branch Prediction

- Ideal pipelined processor: CPI = 1
- Branch misprediction increases CPI
- **Static branch prediction:**

Branch Prediction

- Ideal pipelined processor: CPI = 1
- Branch misprediction increases CPI
- **Static branch prediction:**
  - Check direction of branch (forward or backward)

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

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

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:**

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) branches in *branch target buffer*, record:

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) branches in *branch target buffer,* record:
    - Branch destination

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) branches in *branch target buffer,* record:
    - Branch destination
    - Whether branch was taken

1-Bit Branch Predictor

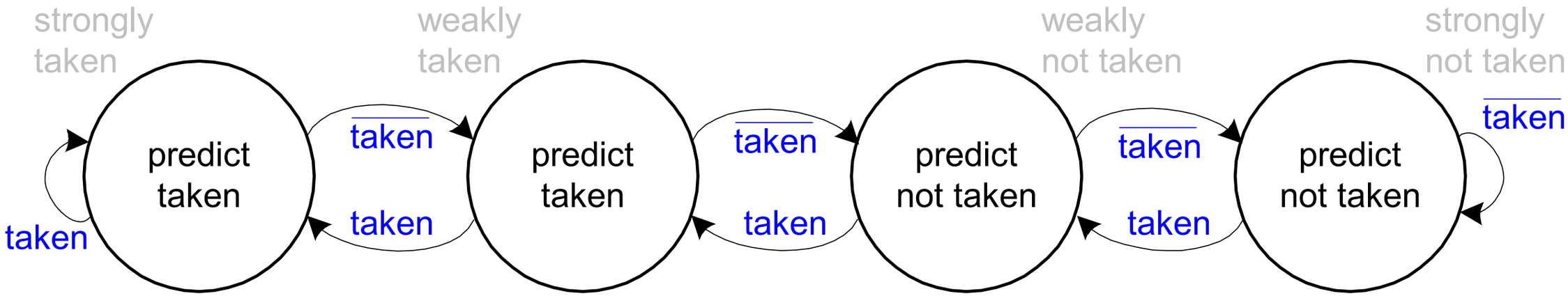
- Remembers whether branch was taken the last time and does the same thing
- Mispredicts first and last branch of loop

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

for:
   beq     $s0,   $t0,   done      # if i == 10, branch to done
   add     $s1,   $s1,   $s0       # sum = sum + i
   addi    $s0,   $s0,   1         # increment i
   j       for
done:
```

2-Bit Branch Predictor

Only mispredicts last branch of loop




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

for:
   beq     $s0,   $t0,   done      # if i == 10, branch to done
   add     $s1,   $s1,   $s0       # sum = sum + i
   addi    $s0,   $s0,   1         # increment i
   j       for
done:
```

Pipelined Performance Example

- 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
  - All jumps flush next instruction
- What is the average CPI?

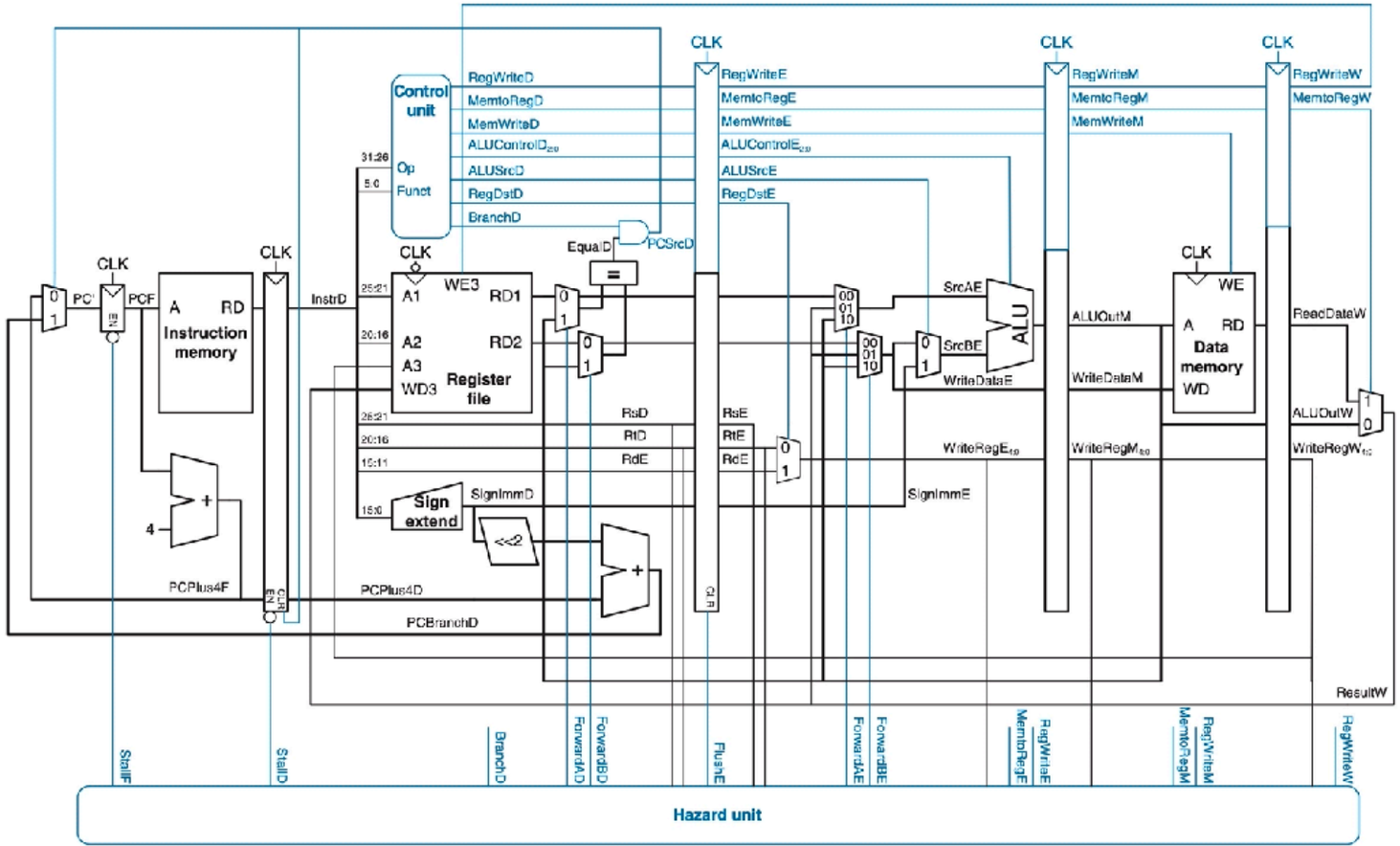
Pipelined Performance Example

- Suppose:
  - 40% of loads followed by the next instruction —> results dependency
  - 25% of branches mispredicted
  - All jumps flush next instruction
- What is the average CPI?
  - Load/Branch CPI = 1 when no stalling, 2 when stalling
  - CPIlw = 1(0.6) + 2(0.4) = 1.4
  - CPIbeq = 1(0.75) + 2(0.25) = 1.25
  - Average CPI = (0.25)(1.4) + (0.1)(1) + (0.11)(1.25) + (0.02)(2) + (0.52)(1) = 1.15

Pipelined Performance

- Pipelined processor critical path:

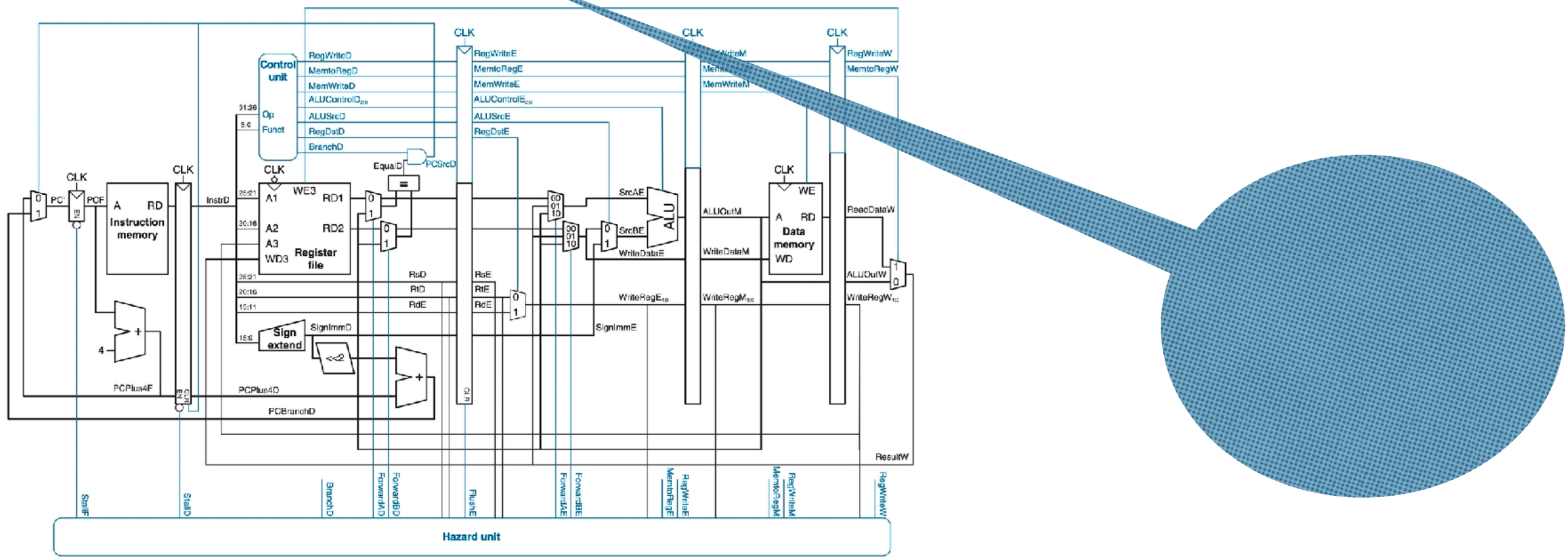
$$T_c = max \left(\begin{array}{c} t_{pcq} + t_{\text{mem}} + t_{\text{setup}} \\ 2(t_{RF\text{read}} + t_{\text{mux}} + t_{\text{eq}} + t_{\text{AND}} + T_{\text{mux}} + t_{\text{setup}}) \\ t_{pcq} + t_{\text{mux}} + t_{\text{mux}} + t_{\text{ALU}} + t_{\text{setup}} \\ t_{pcq} + t_{\text{memwrite}} + t_{\text{setup}} \\ 2(t_{pcq} + t_{\text{mux}} + t_{RF\text{write}}) \end{array} \right\} \begin{array}{c} \text{Fetch} \\ \text{Decode} \\ \text{Execute} \\ \text{Memory} \\ \text{Writeback} \end{array}$$



Pipelined Performance

- Pipelined processor critical path:

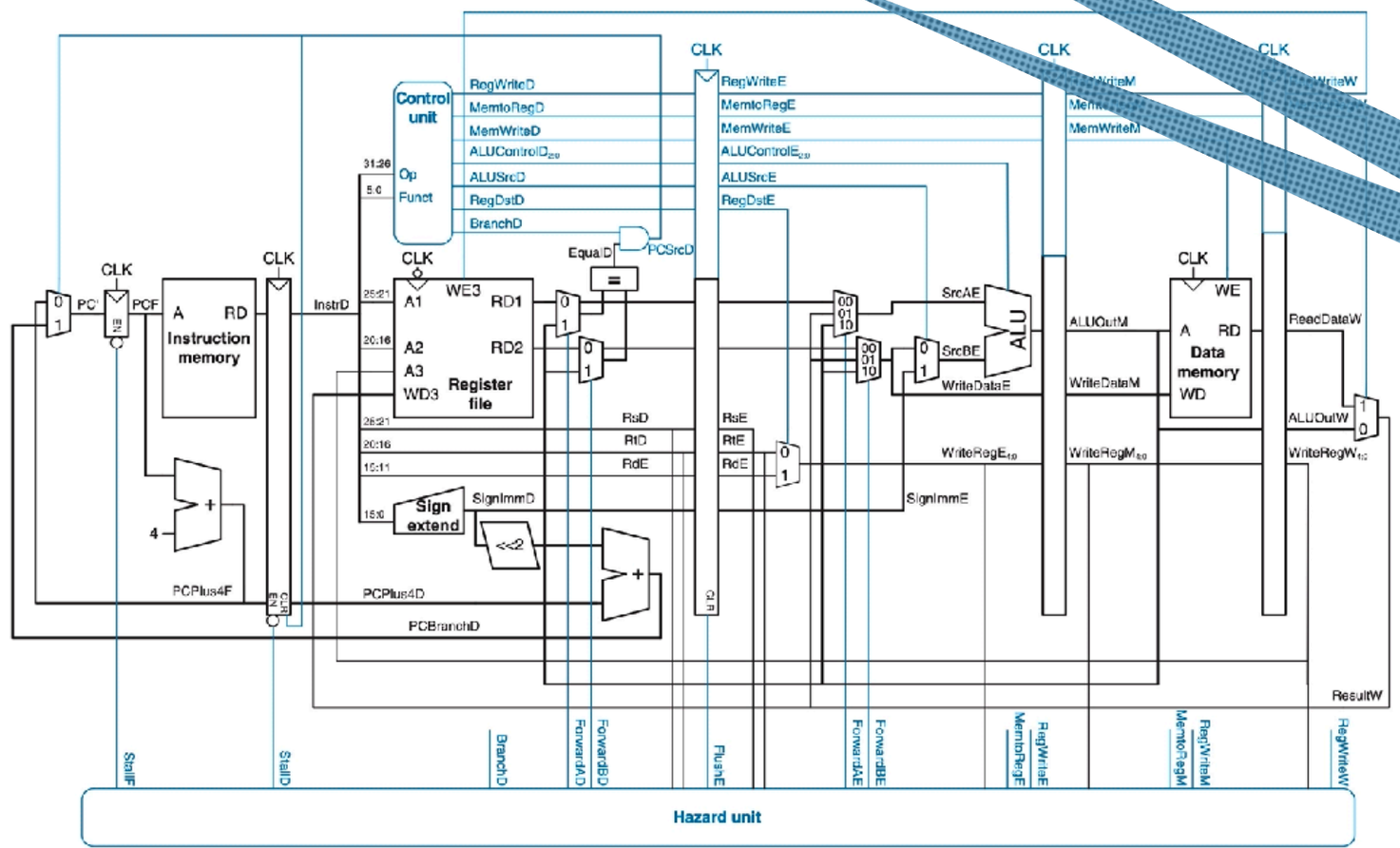
$$T_c = max \left\{ \begin{array}{c} t_{pcq} + t_{\text{mem}} + t_{\text{setup}} \\ 2(t_{RF\text{read}} + t_{\text{mux}} + t_{\text{eq}} + t_{\text{AND}} + T_{\text{mux}} + t_{\text{setup}}) \\ t_{pcq} + t_{\text{mux}} + t_{\text{mux}} + t_{\text{ALU}} + t_{\text{setup}} \\ t_{pcq} + t_{\text{memwrite}} + t_{\text{setup}} \\ 2(t_{pcq} + t_{\text{mux}} + t_{RF\text{write}}) \end{array} \right\} \begin{array}{c} \text{Fetch} \\ \text{Decode} \\ \text{Execute} \\ \text{Memory} \\ \text{Writeback} \end{array}$$



Pipelined Performance

- Pipelined processor critical path:

$$T_c = max \left(\begin{array}{c} t_{pcq} + t_{\text{mem}} + t_{\text{setup}} \\ 2(t_{RF\text{read}} + t_{\text{mux}} + t_{\text{eq}} + t_{\text{AND}} + T_{\text{mux}} + t_{\text{setup}}) \\ t_{pcq} + t_{\text{mux}} + t_{\text{mux}} + t_{\text{ALU}} + t_{\text{setup}} \\ t_{pcq} + t_{\text{memwrite}} + t_{\text{setup}} \\ 2(t_{pcq} + t_{\text{mux}} + t_{RF\text{write}}) \end{array} \right) \begin{array}{l} \text{Fetch} \\ \text{Decode} \\ \text{Execute} \\ \text{Memory} \\ \text{Writeback} \end{array}$$



Write in 1st half
Read in 2nd half

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_{RFread}	150
Register file setup	$t_{RFsetup}$	20
Equality comparator	t_{eq}	40
AND gate	t_{AND}	15
Memory write	T_{memwrite}	220
Register file write	$t_{RFwrite}$	100 ps

$$T_c = 2(t_{RFread} + t_{\text{mux}} + t_{\text{eq}} + t_{\text{AND}} + t_{\text{mux}} + t_{\text{setup}})$$

$$= 2[150 + 25 + 40 + 15 + 25 + 20] \text{ ps} = \mathbf{550 \text{ ps}}$$

Pipelined Performance Example

Program with 100 billion instructions

Execution Time = (# instructions) × CPI × T_c

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

= 63 seconds

Processor Performance Comparison

Processor	Execution Time (seconds)	Speedup (single-cycle as baseline)
Single-cycle	92.5	1
Multicycle	133	0.70
Pipelined	63	1.47