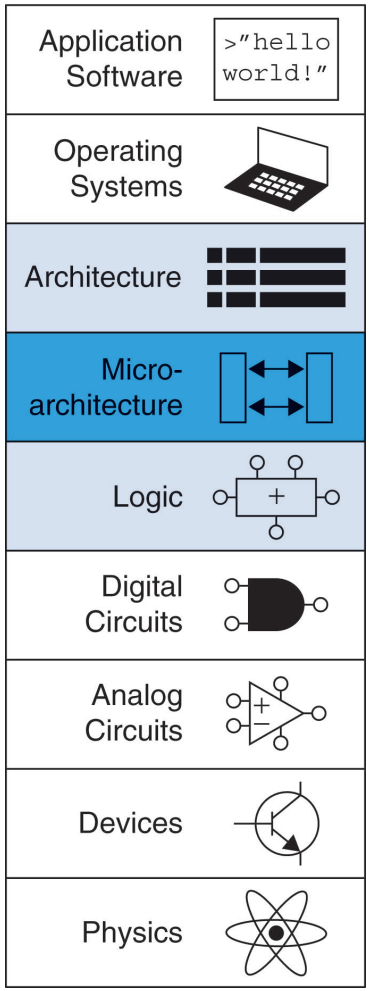
Digital Design and Computer Architecture, RISC-V Edition

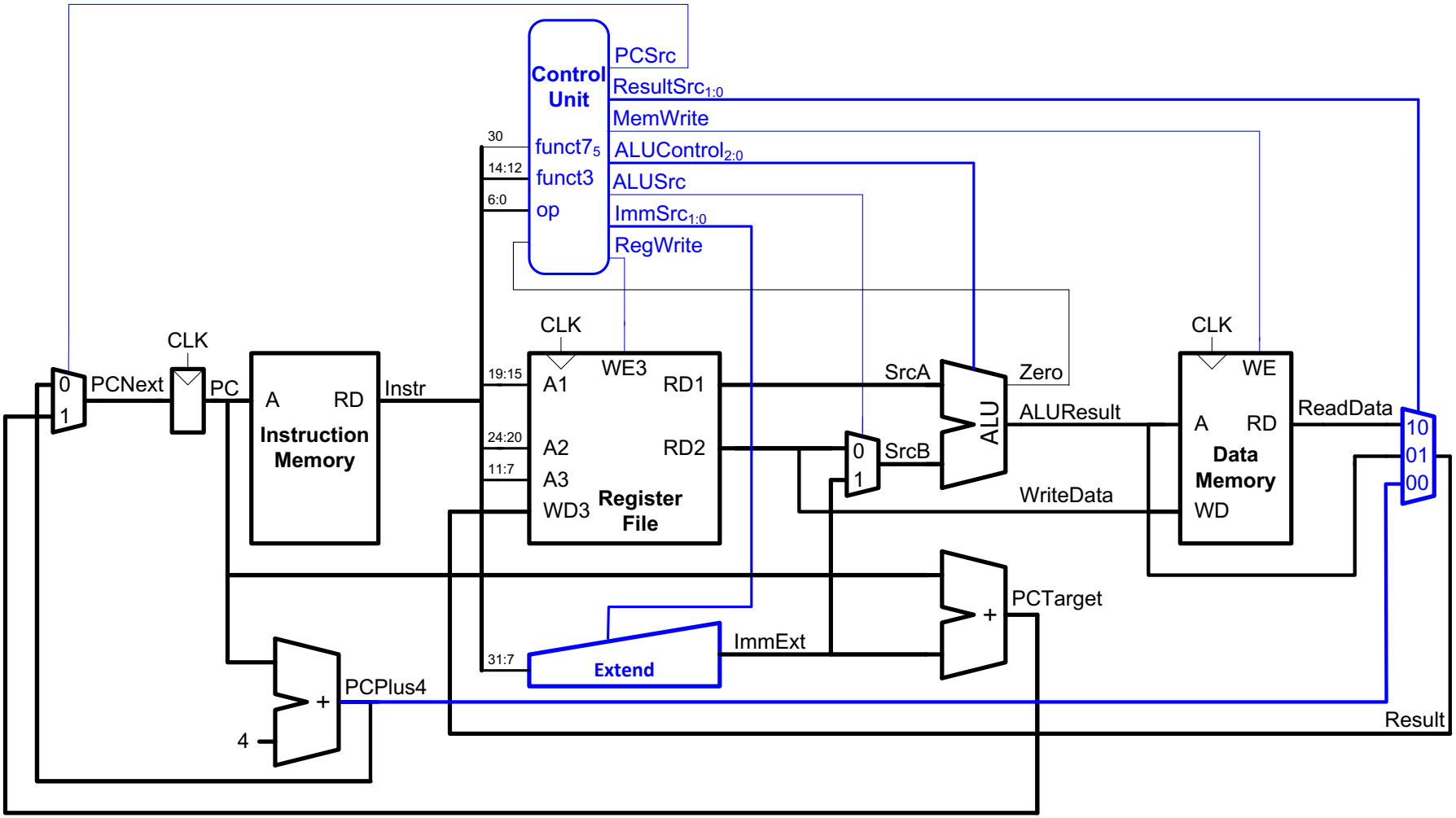
David M. Harris and Sarah L. Harris

Chapter 7 :: Microarchitecture

- **Introduction**
- **Performance Analysis**
- **Single-Cycle Processor**
- **Multicycle Processor**
- **Pipelined Processor**
- **Advanced Microarchitecture**



Single Cycle Processor



Single Cycle Main Decoder

op	Instruct	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp	PCUpdate
3	`lw`	1	00	1	0	10	0	00	**0**
35	`sw`	0	01	1	1	XX	0	00	**0**
51	**R-type**	1	XX	0	0	01	0	10	**0**
99	`beq`	0	10	0	0	XX	1	01	**0**
19	`addi`	1	00	1	0	01	0	10	**0**
111	`jal`	**1**	**11**	**X**	**0**	**00**	**0**	**XX**	**1**

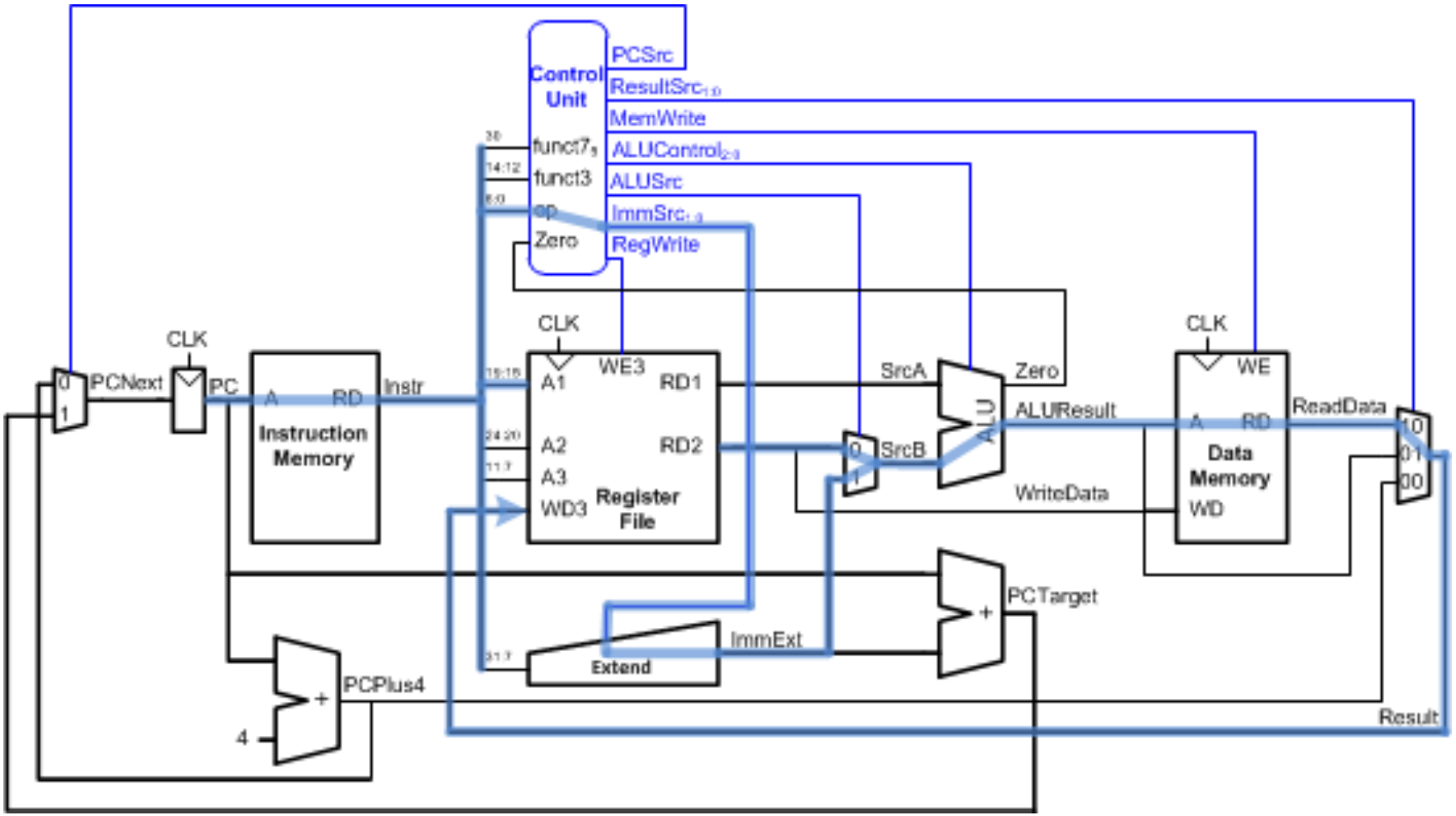
Processor Performance

Program Execution Time

= (#instructions)(cycles/instruction)(seconds/cycle)

= # instructions x CPI x T_C

Single-Cycle Performance



T_c limited by critical path (`lw`)

Single-Cycle Performance

- **Single-cycle critical path:**

$$T_{c1} = t_{pcq\_PC} + t_{\text{mem}} + \max[t_{RF\text{read}}, t_{dec} + t_{ext} + t_{\text{mux}}] + t_{\text{ALU}} + t_{\text{mem}} + t_{\text{mux}} + t_{RF\text{setup}}$$

- **Typically, limiting paths are:**
  - memory, ALU, register file
  - $T_{c1} = t_{pcq\_PC} + 2t_{\text{mem}} + t_{RF\text{read}} + t_{\text{ALU}} + t_{\text{mux}} + t_{RF\text{setup}}$

Multicycle RISC-V Processor

- **Single-cycle:**
  + simple
  - cycle time limited by longest instruction (`lw`)
  - separate memories for instruction and data
  - 3 adders/ALUs
- **Multicycle processor addresses these issues by breaking instruction into shorter steps**
  - shorter instructions take fewer steps
  - can re-use hardware
  - cycle time is faster

Single-Cycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	30
ALU	t_{ALU}	120
Decoder (Control Unit)	t_{dec}	35
Memory read	t_{mem}	200
Register file read	t_{RFread}	100
Register file setup	$t_{RFsetup}$	60

$$T_{c1} = ?$$

Single-Cycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	30
ALU	t_{ALU}	120
Decoder (Control Unit)	t_{dec}	35
Memory read	t_{mem}	200
Register file read	t_{RFread}	100
Register file setup	$t_{RFsetup}$	60

$$T_{c1} = t_{pcq\_PC} + 2t_{mem} + t_{RFread} + t_{ALU} + t_{mux} + t_{RFsetup}$$

$$= [40 + 2(200) + 100 + 120 + 30 + 60]\ \text{ps}$$

$$= \mathbf{750\ ps}$$

Single-Cycle Performance Example

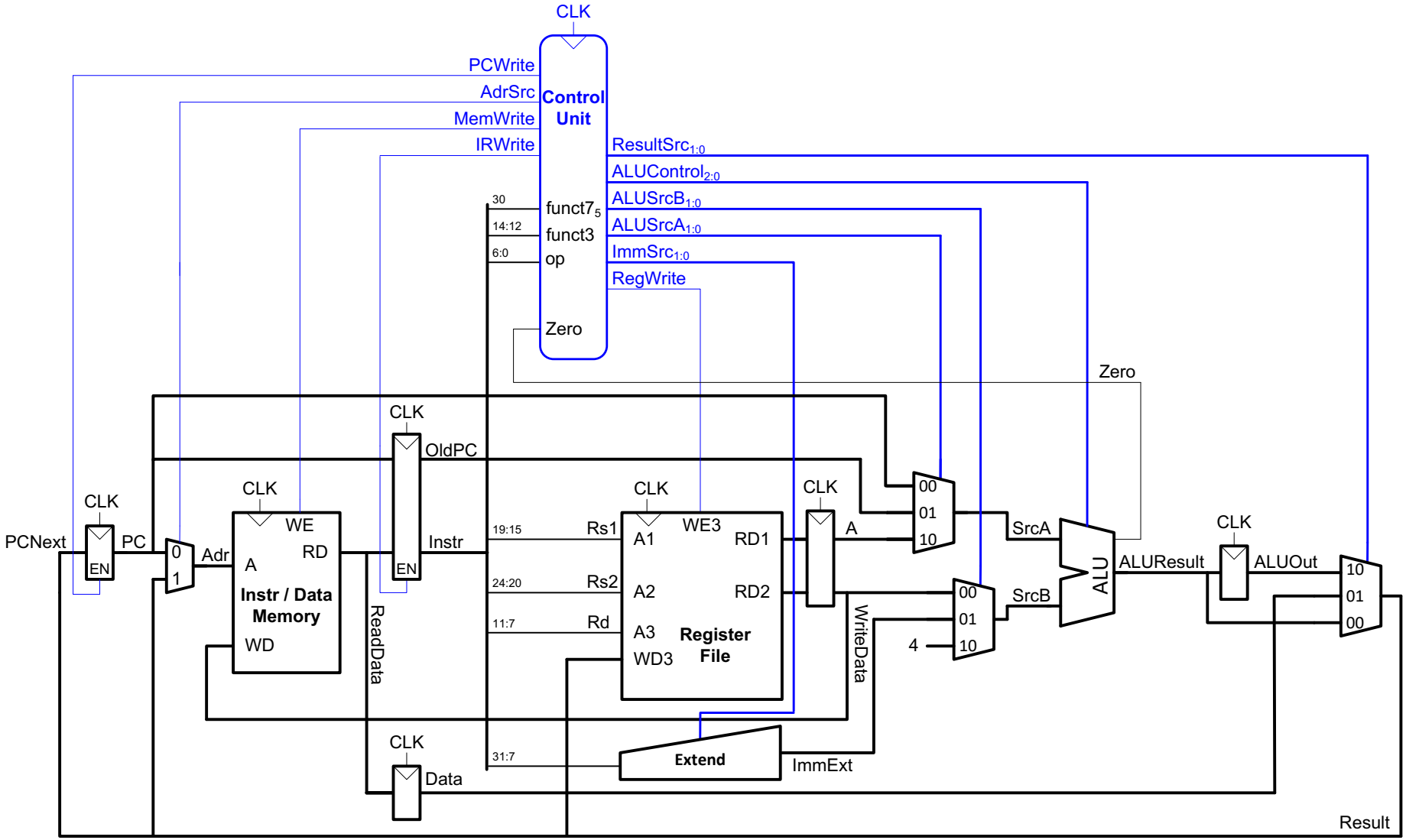
Program with 100 billion instructions:

Execution Time = # instructions x CPI x T_C

$= (100 \times 10^9)(1)(750 \times 10^{-12}\ \text{s})$

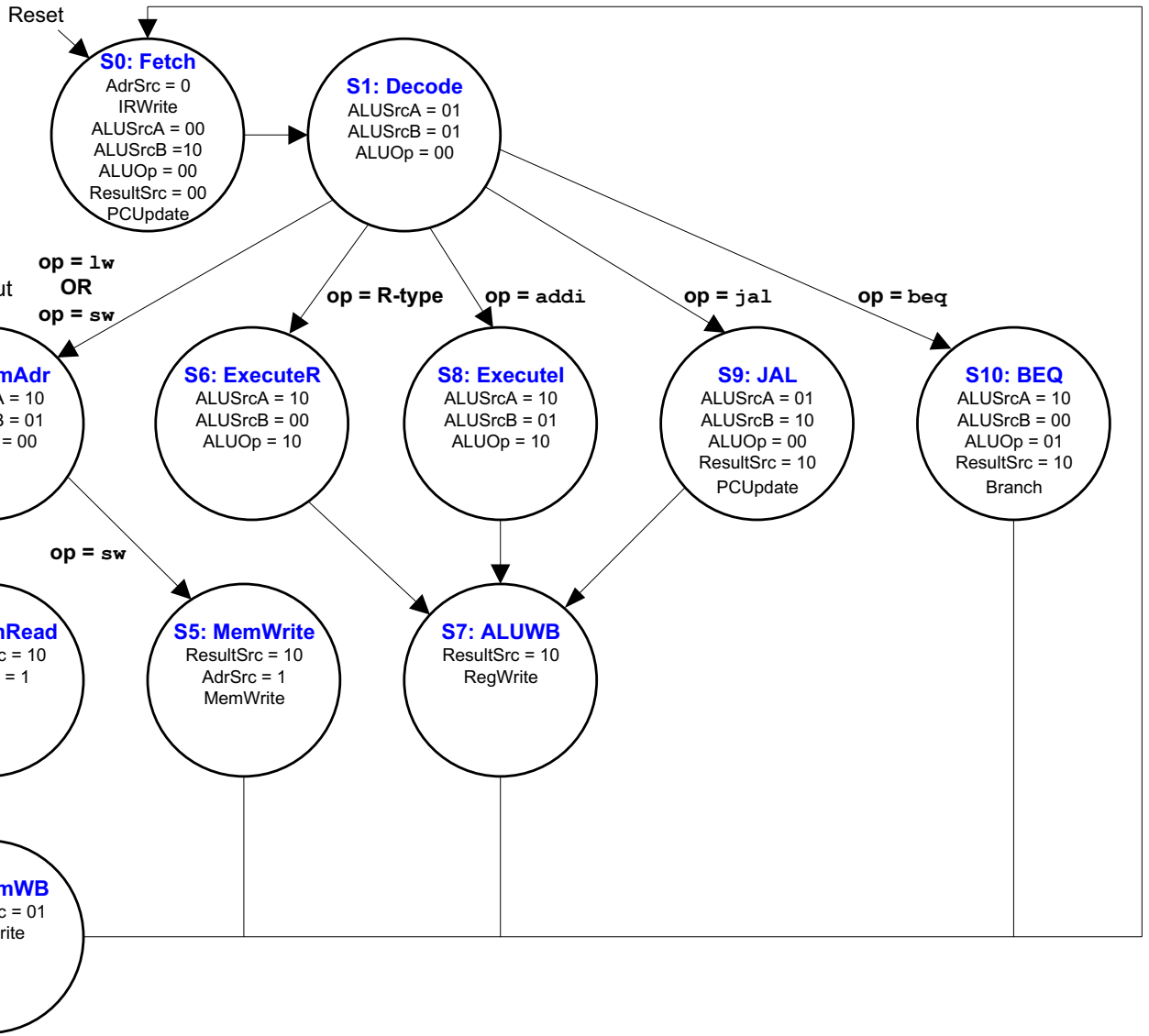
= **75 seconds**

Review: Multicycle RISC-V Processor



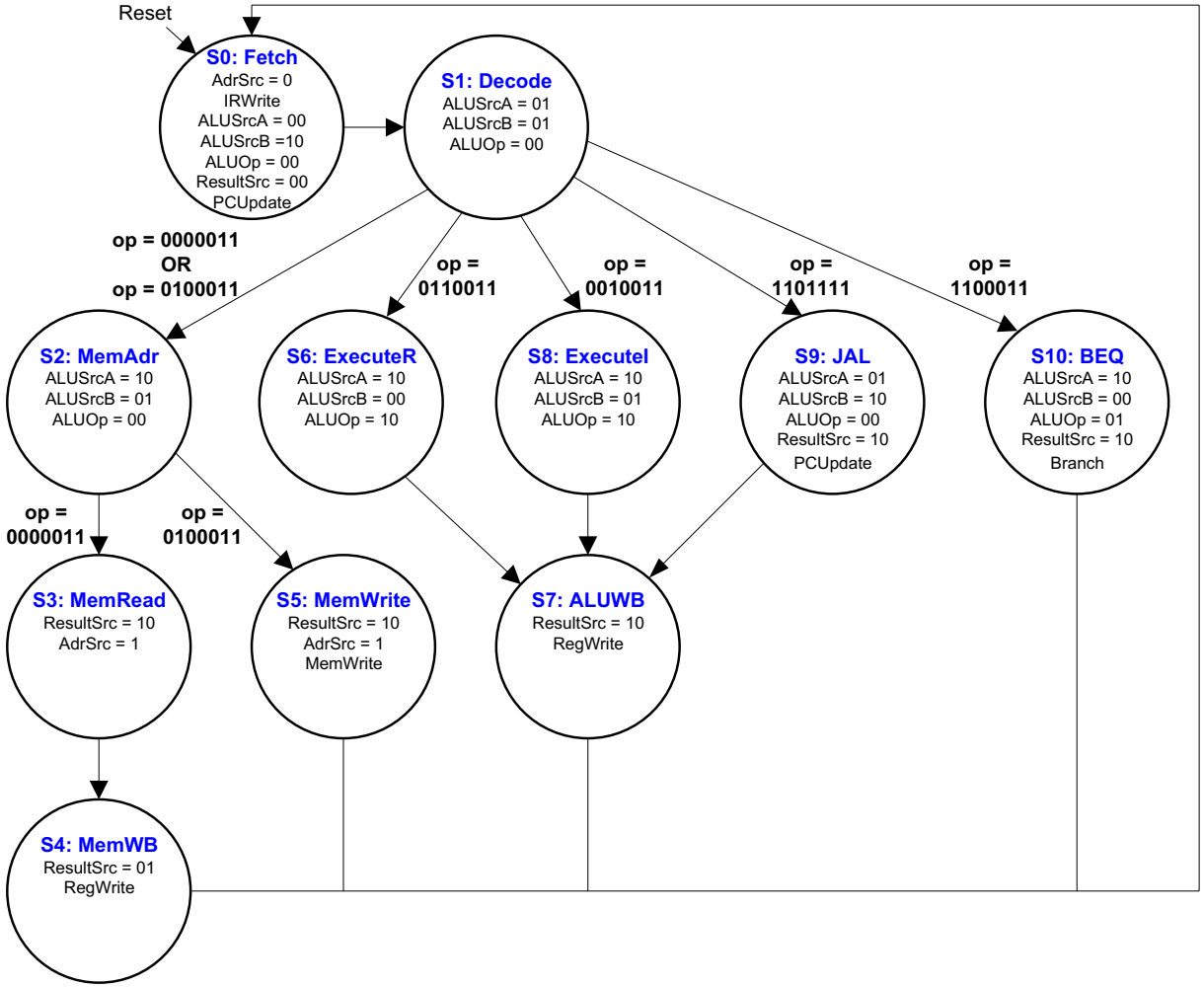
Review: Multicycle Main FSM

State	Datapath μOp
Fetch	Instr ←Mem[PC]; PC ← PC+4
Decode	ALUOut ← PCTarget
MemAdr	ALUOut ← rs1 + imm
MemRead	Data ← Mem[ALUOut]
MemWB	rd ← Data
MemWrite	Mem[ALUOut] ← rd
ExecuteR	ALUOut ← rs1 op rs2
ExecuteI	ALUOut ← rs1 op imm
ALUWB	rd ← ALUOut
BEQ	ALUResult = rs1-rs2; if Zero, PC = ALUOut
JAL	PC = ALUOut, ALUOut = PC+4



Multicycle Processor Performance

- Instructions take different number of cycles:



Multicycle Processor Performance

- Instructions take different number of cycles:
  - 3 cycles:
  - 4 cycles:
  - 5 cycles:

Multicycle Processor Performance

- Instructions take different number of cycles:
  - 3 cycles: `beq`
  - 4 cycles: R-type, `addi`, `sw`, `jal`
  - 5 cycles: `lw`

Multicycle Processor Performance

- Instructions take different number of cycles:
  - 3 cycles: `beq`
  - 4 cycles: R-type, `addi`, `sw`, `jal`
  - 5 cycles: `lw`
- CPI is weighted average
- SPECINT2000 benchmark:
  - 25% loads
  - 10% stores
  - 13% branches
  - 52% R-type

Multicycle Processor Performance

- Instructions take different number of cycles:
  - 3 cycles: `beq`
  - 4 cycles: R-type, `addi`, `sw`, `jal`
  - 5 cycles: `lw`
- CPI is weighted average
- SPECINT2000 benchmark:
  - **25%** loads
  - **10%** stores
  - **13%** branches
  - **52%** R-type

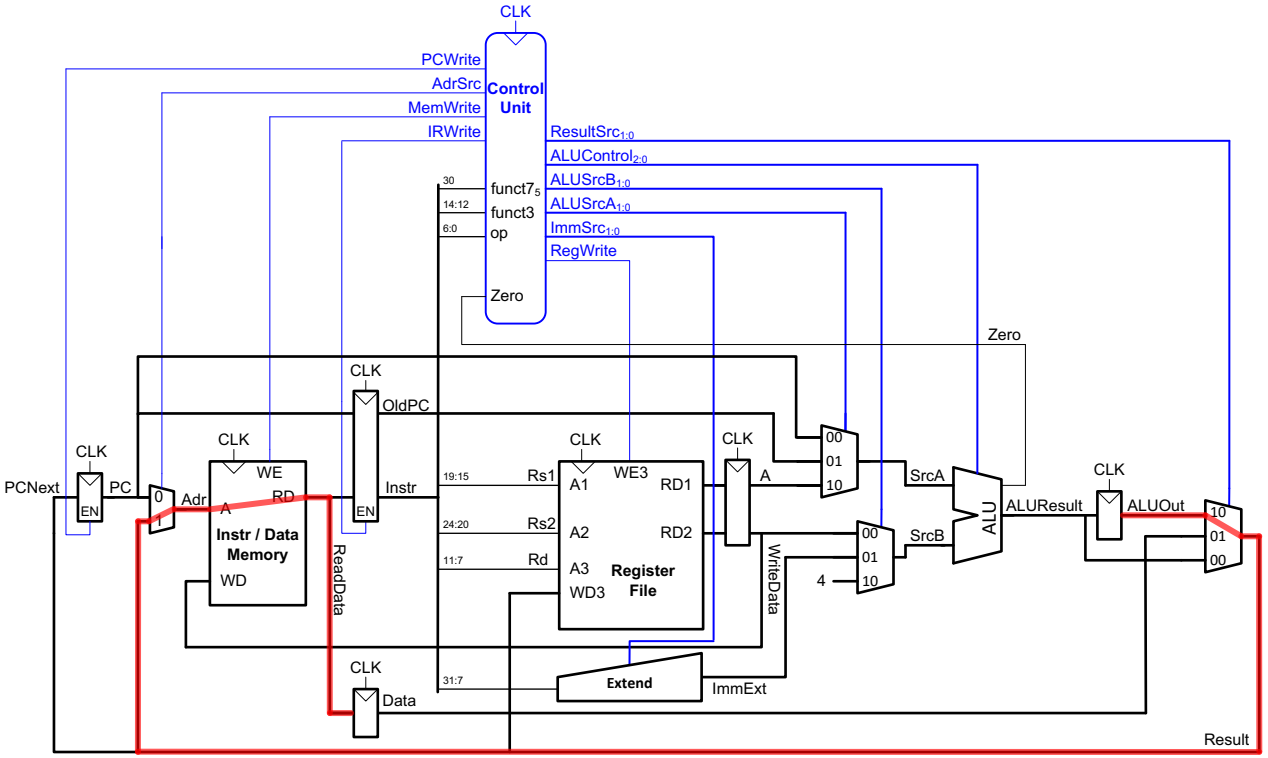
$$\textbf{Average CPI} = (0.13)(3) + (0.52 + 0.10)(4) + (0.25)(5) = \mathbf{4.12}$$

Multicycle Processor Critical Path

- Assumptions:
  - RF is faster than memory
  - writing memory is faster than reading memory
- Two possibilities:
  - Read memory (MemRead state)
  - PC = PC + 4 path (Fetch state)

Multicycle Processor Critical Path

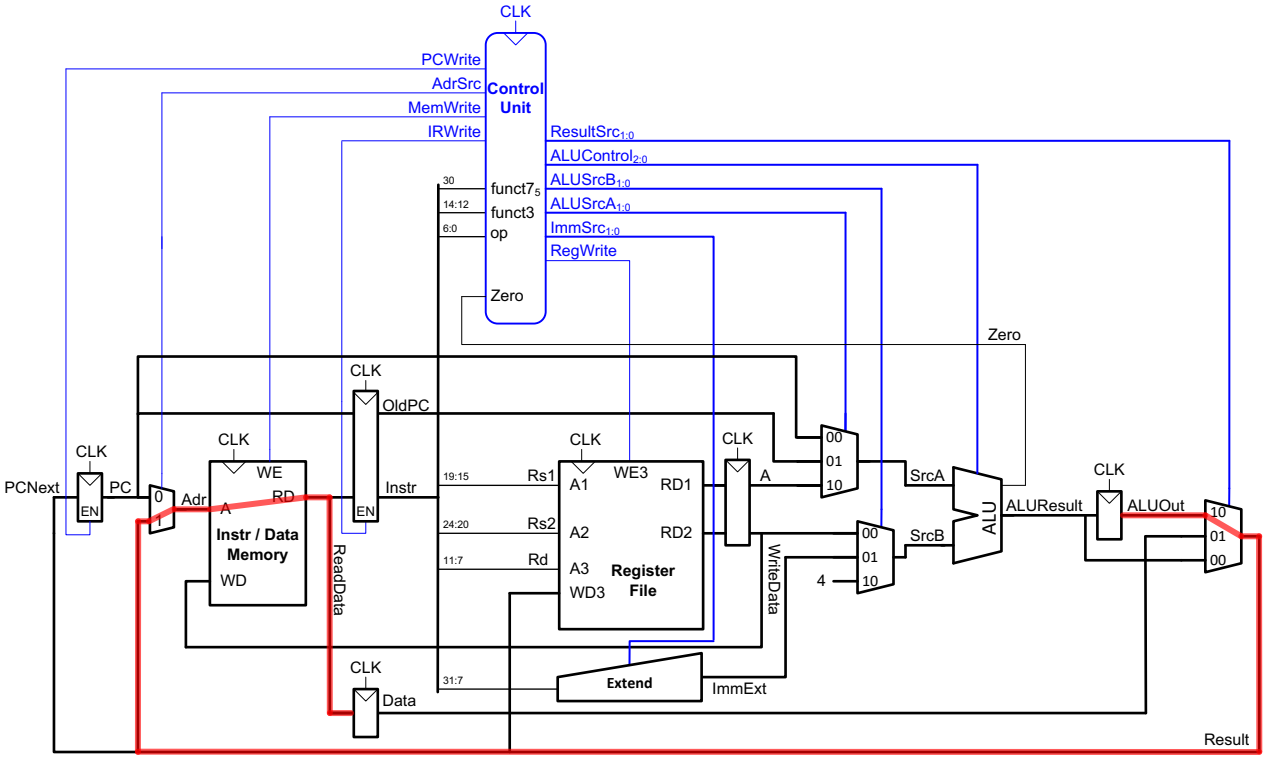
Option 1: Read memory (MemRead state)



$$T_{c2} = t_{pcq} + t_{\text{mux}} + t_{\text{mux}} + t_{\text{mem}} + t_{\text{setup}}$$

Multicycle Processor Critical Path

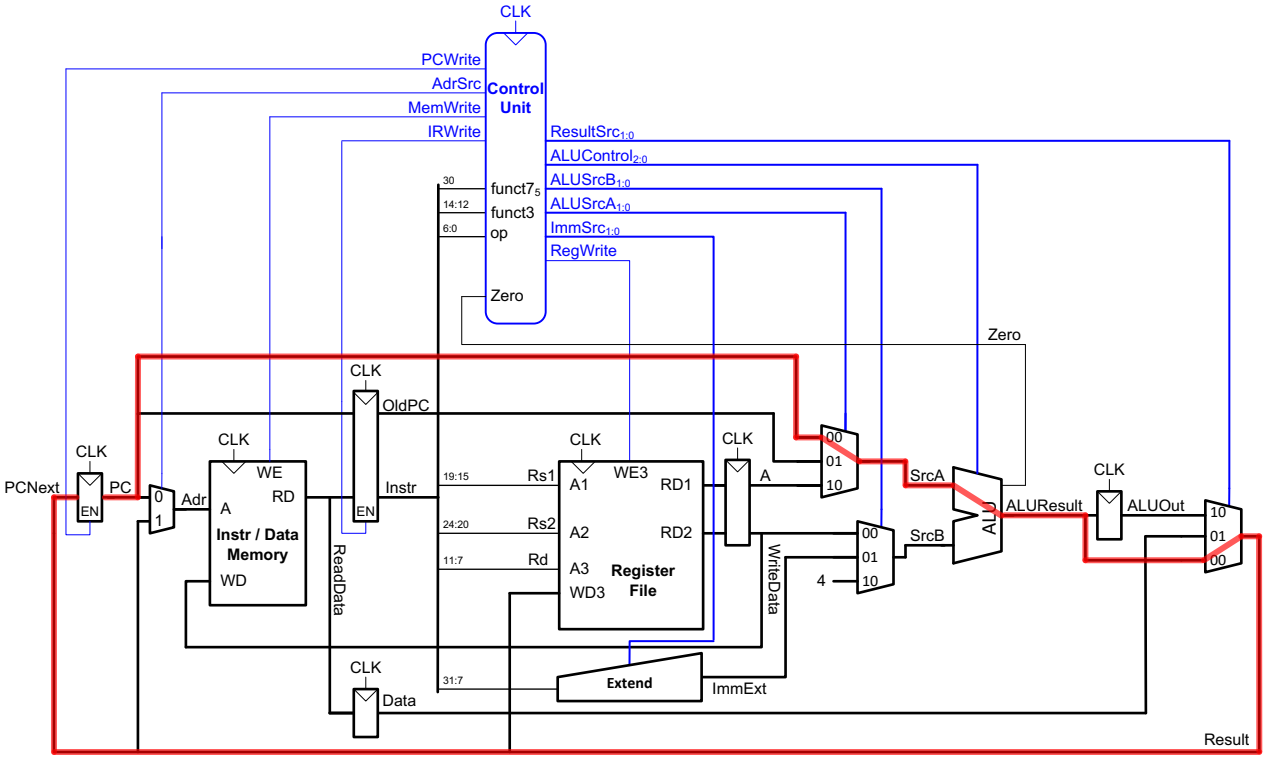
Option 1: Read memory (MemRead state)



$$T_{c2} = t_{pcq} + t_{\text{mux}} + t_{\text{mux}} + t_{\text{mem}} + t_{\text{setup}}$$
$$= t_{pcq} + 2t_{\text{mux}} + t_{\text{mem}} + t_{\text{setup}}$$

Multicycle Processor Critical Path

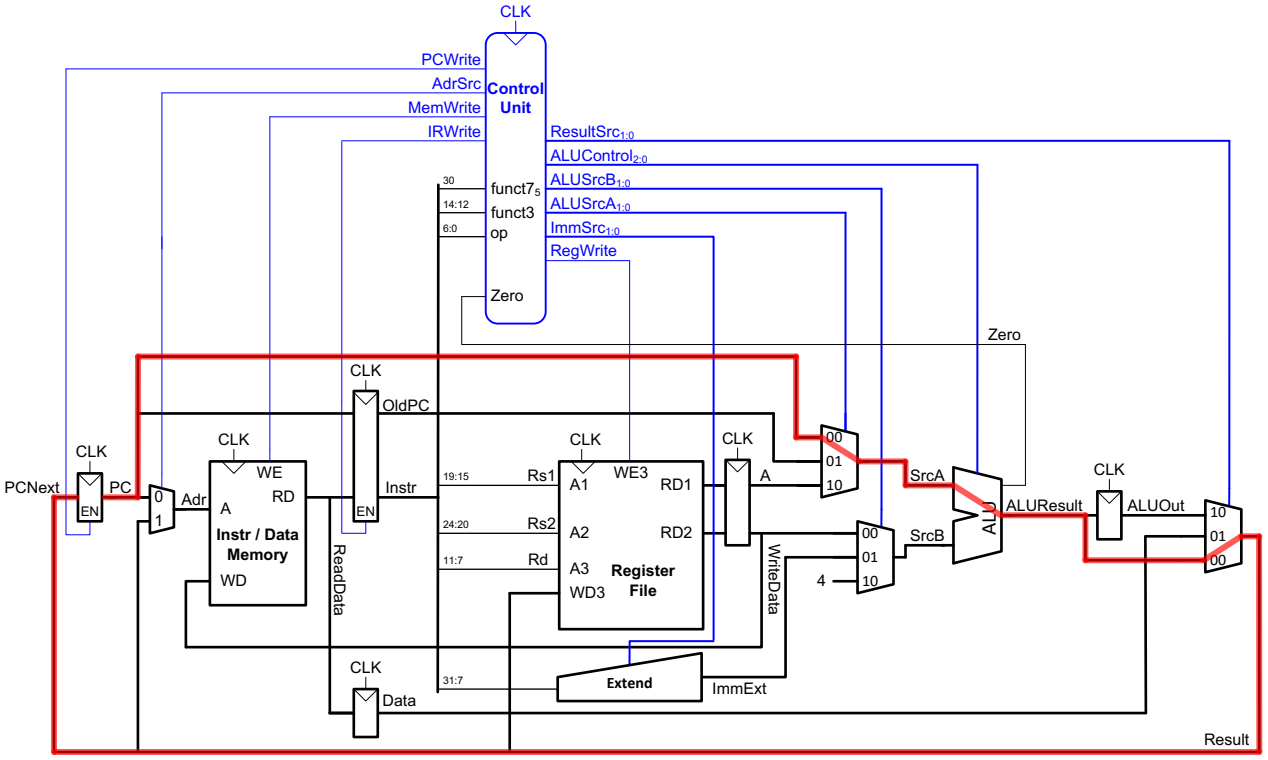
Option 2: PC = PC + 4 path (Fetch state)



$$T_{c2} = t_{pcq} + t_{\text{mux}} + t_{\text{ALU}} + t_{\text{mux}} + t_{\text{setup}}$$

Multicycle Processor Critical Path

Option 2: PC = PC + 4 path (Fetch state)



$$T_{c2} = t_{pcq} + t_{\text{mux}} + t_{\text{ALU}} + t_{\text{mux}} + t_{\text{setup}}$$
$$= t_{pcq} + 2t_{\text{mux}} + t_{\text{ALU}} + t_{\text{setup}}$$

Multicycle Processor Critical Path

- Two possibilities:
  - Read memory (MemRead state)
  - PC = PC + 4 path (Fetch state)

$$T_{c2} = t_{pcq} + 2t_{\text{mux}} + \max[t_{\text{ALU}}, t_{\text{mem}}] + t_{\text{setup}}$$

Multi-Cycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	30
AND-OR gate	$t_{AND\text{-}OR}$	20
ALU	t_{ALU}	120
Decoder (control unit)	t_{dec}	35
Memory read	t_{mem}	200
Register file read	t_{RFread}	100
Register file setup	$t_{RFsetup}$	60

$$T_{c2} = t_{pcq} + 2t_{mux} + \max[t_{ALU}, t_{mem}] + t_{setup}$$
$$= (40 + 2(30) + 200 + 50)\ \text{ps} = \mathbf{350\ ps}$$

Multicycle Performance Example

For a program with **100 billion** instructions executing on a **multicycle** RISC-V processor

- **CPI** = 4.12 cycles/instruction
- **Clock cycle time:** T_{c2} = 350 ps

Execution Time = ?

Multicycle Performance Example

For a program with **100 billion** instructions executing on a **multicycle** RISC-V processor

- **CPI** = 4.12 cycles/instruction
- **Clock cycle time:** T_{c2} = 350 ps

Execution Time = (# instructions) × CPI × T_c

$= (100 \times 10^9)(4.12)(350 \times 10^{-12})$

$=$ **144 seconds**

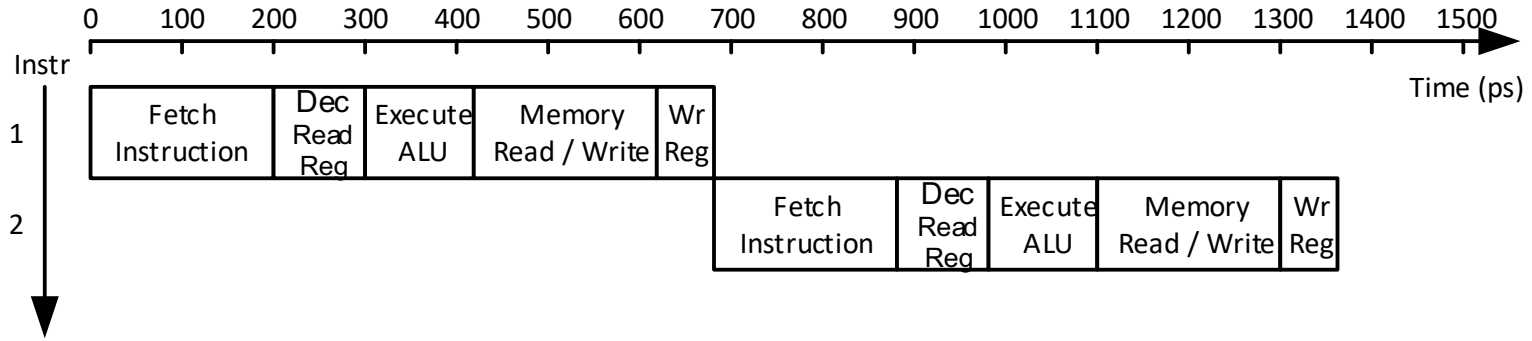
This is slower than the single-cycle processor (75 sec.)

Pipelined RISC-V Processor

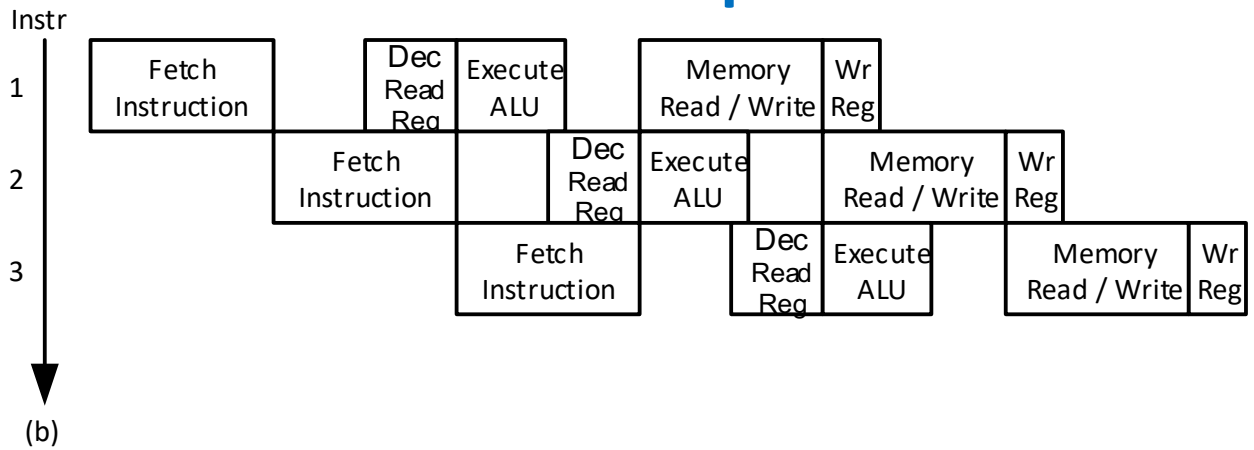
- Temporal parallelism
- Divide single-cycle processor into 5 stages:
  - Fetch
  - Decode
  - Execute
  - Memory
  - Writeback
- Add pipeline registers between stages

Single-Cycle vs. Pipelined

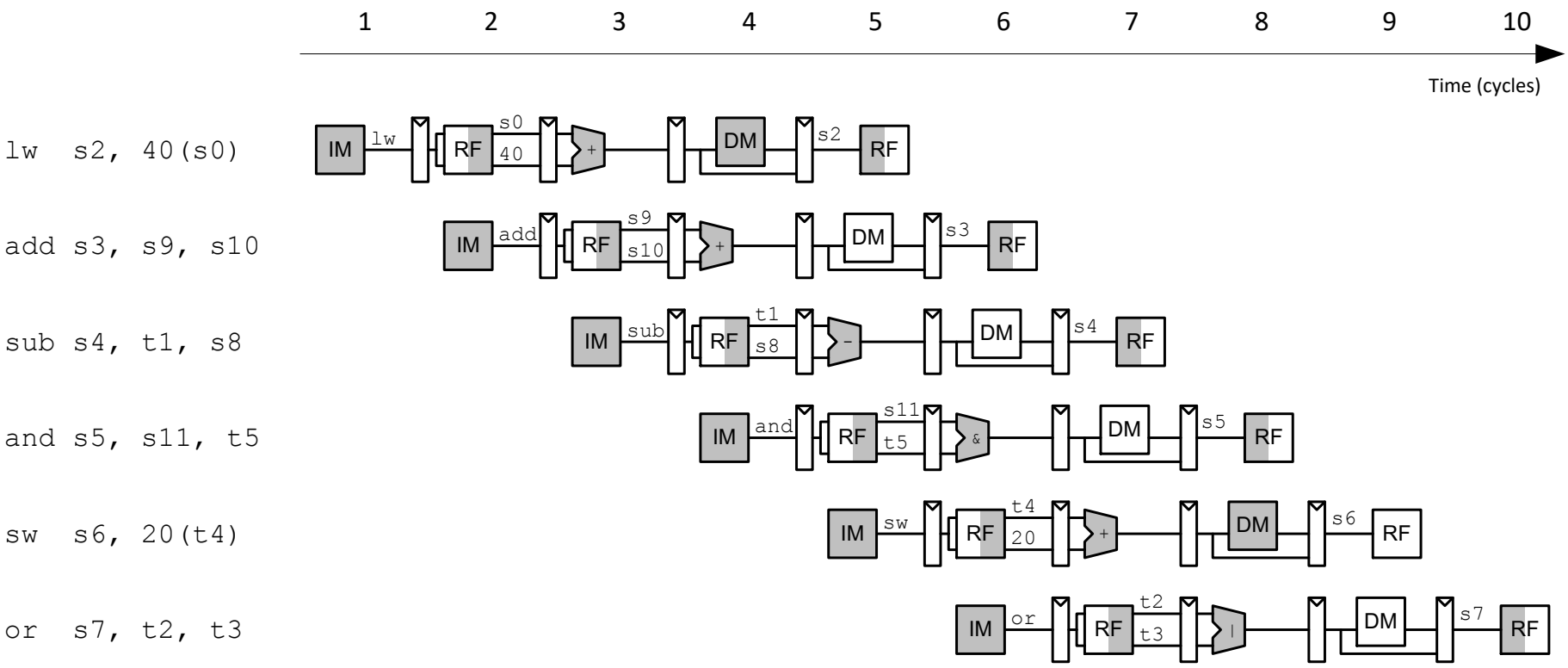
Single-Cycle



Pipelined

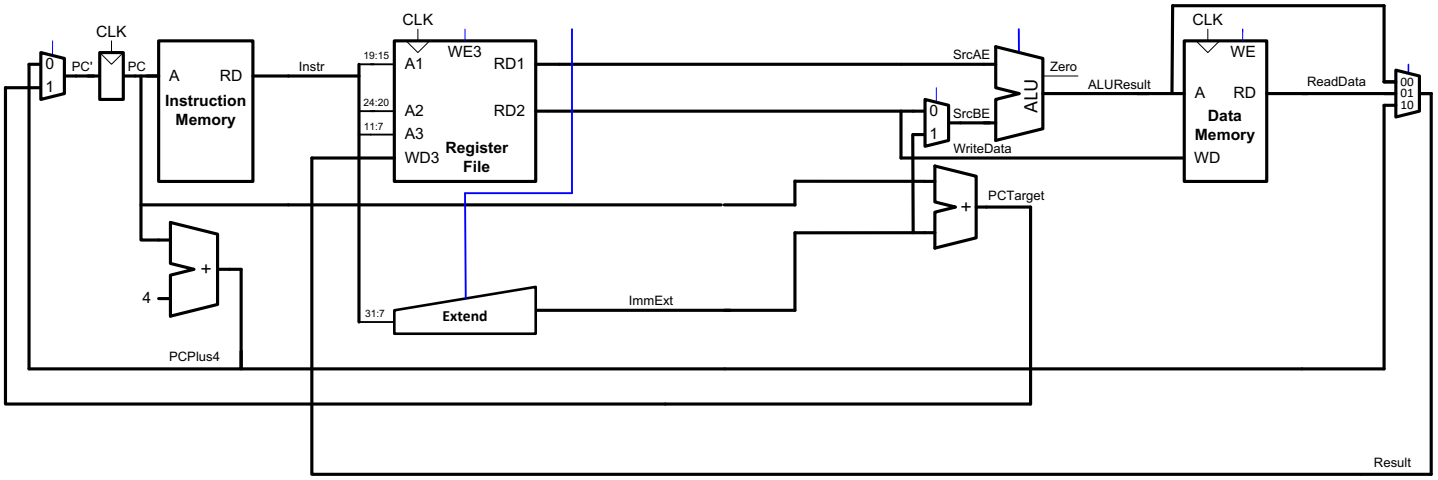


Pipelined Processor Abstraction

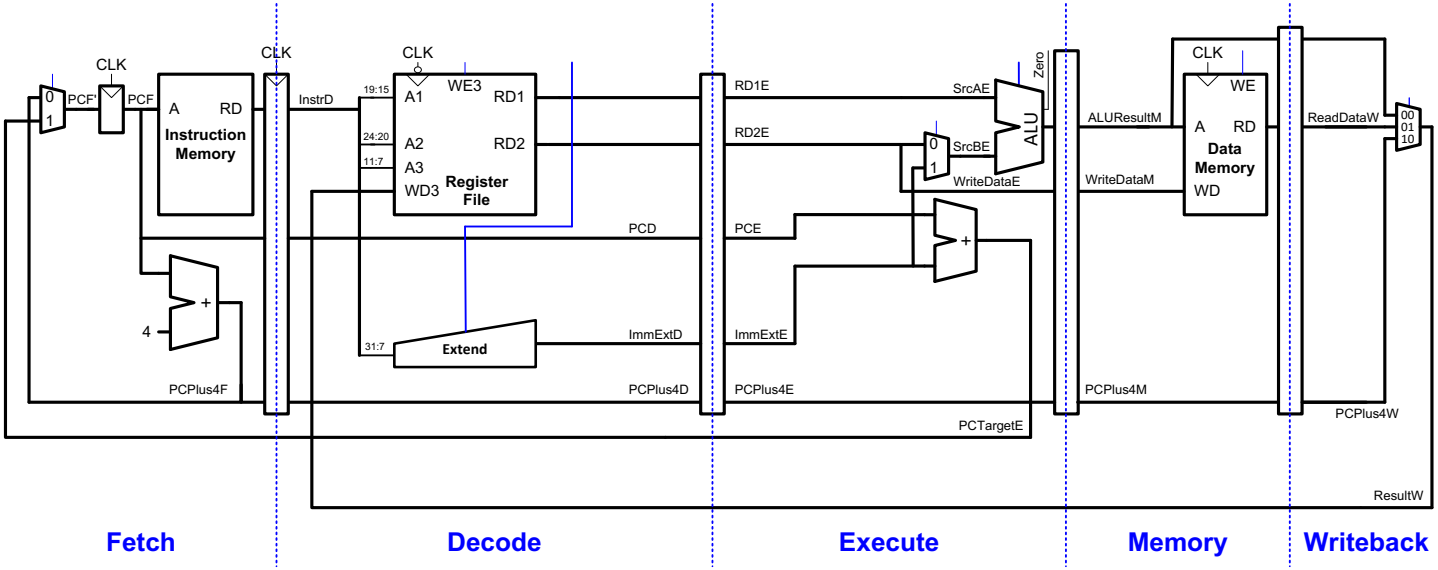


Single-Cycle & Pipelined Datapath

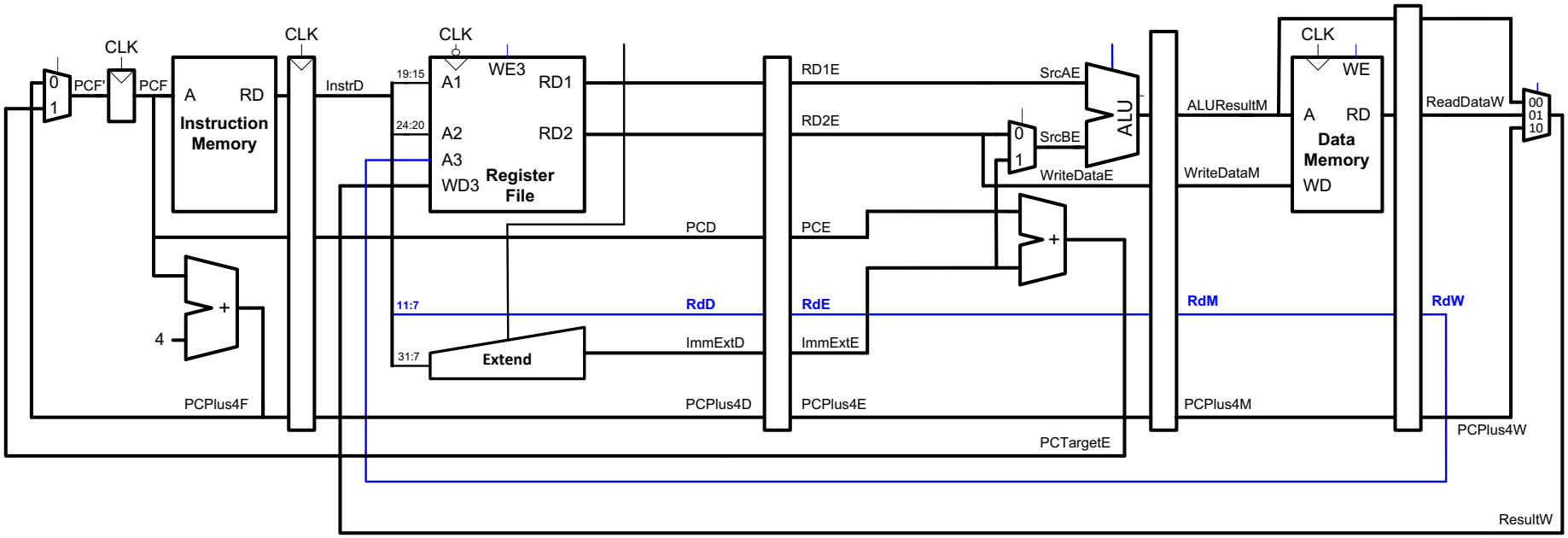
Single-Cycle



Pipelined

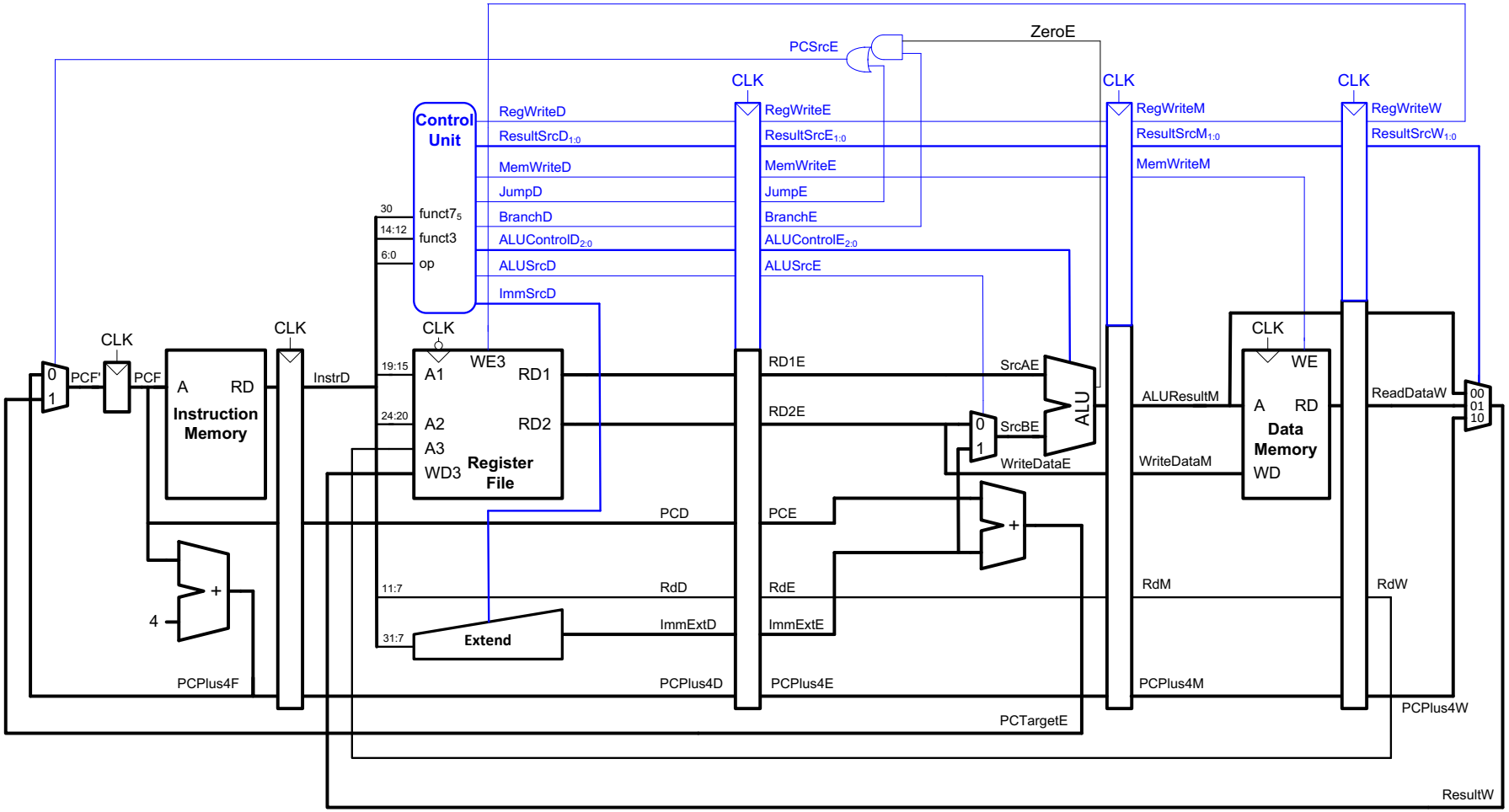


Corrected Pipelined Datapath



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

Pipelined Processor with Control



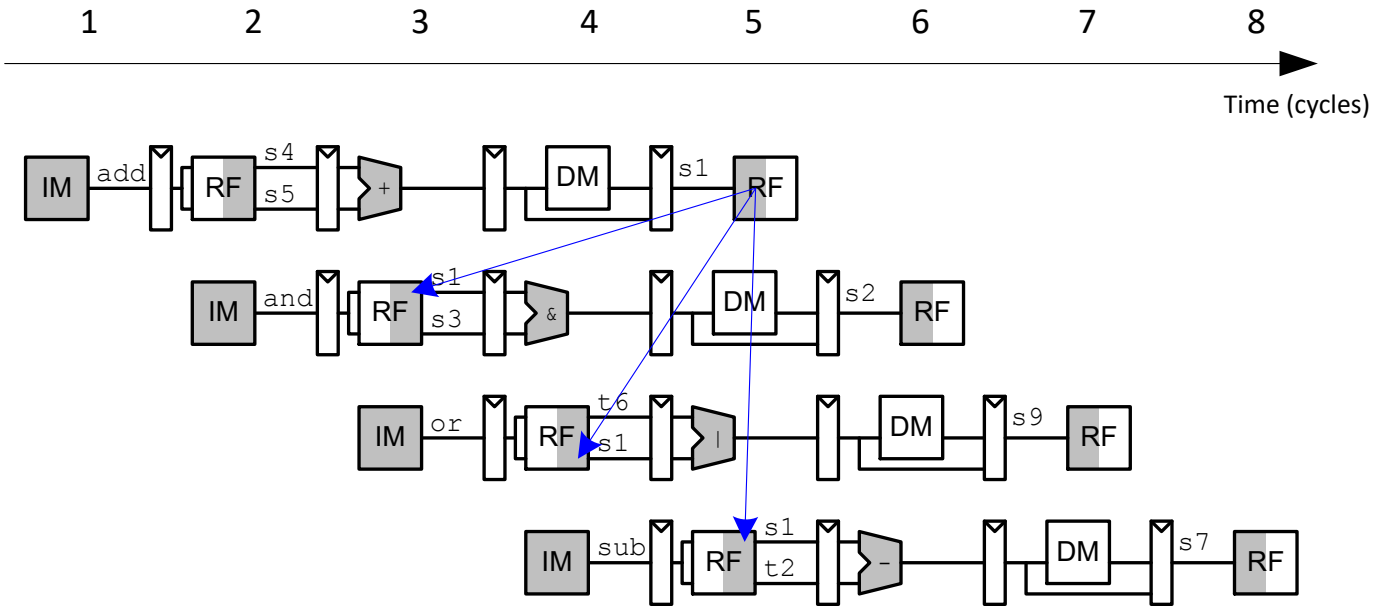
- **Same control unit** as single-cycle processor
- **Control signals travel with** the instruction (drop off when used)

Pipeline 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 s1, s4, s5
and s2, s1, s3
or  s9, t6, s1
sub s7, s1, t2
```



Handling Data Hazards

How do we ensure that our programs run correctly?

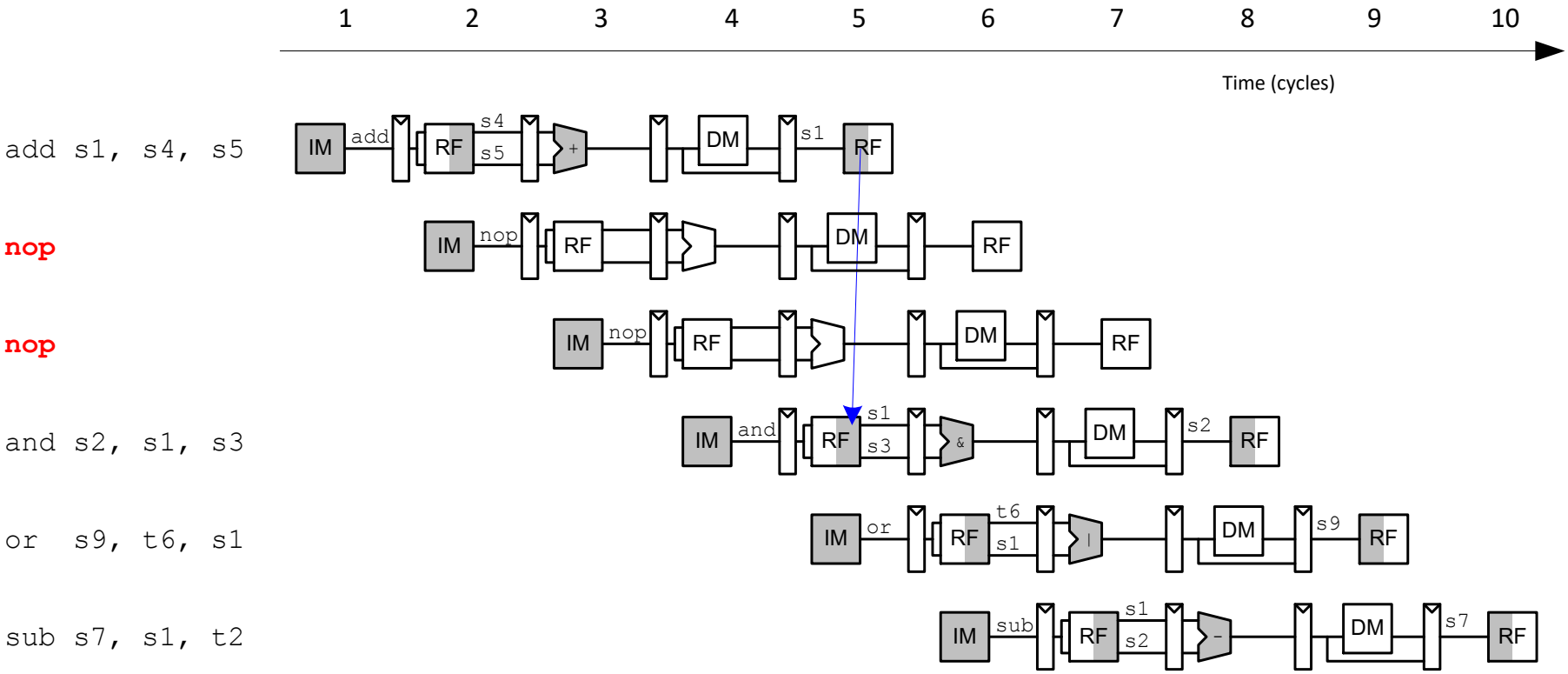
Handling Data Hazards

How do we ensure that our programs run correctly?

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

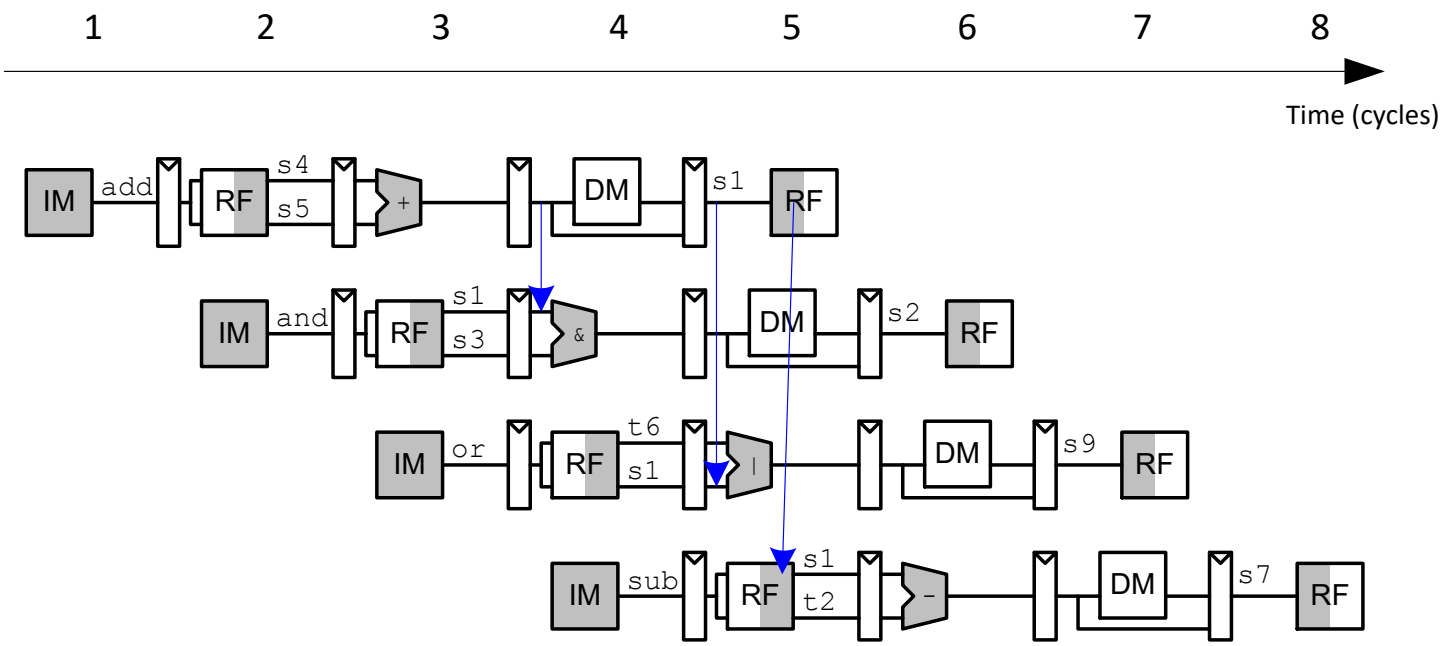
Compile-Time Hazard Elimination

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

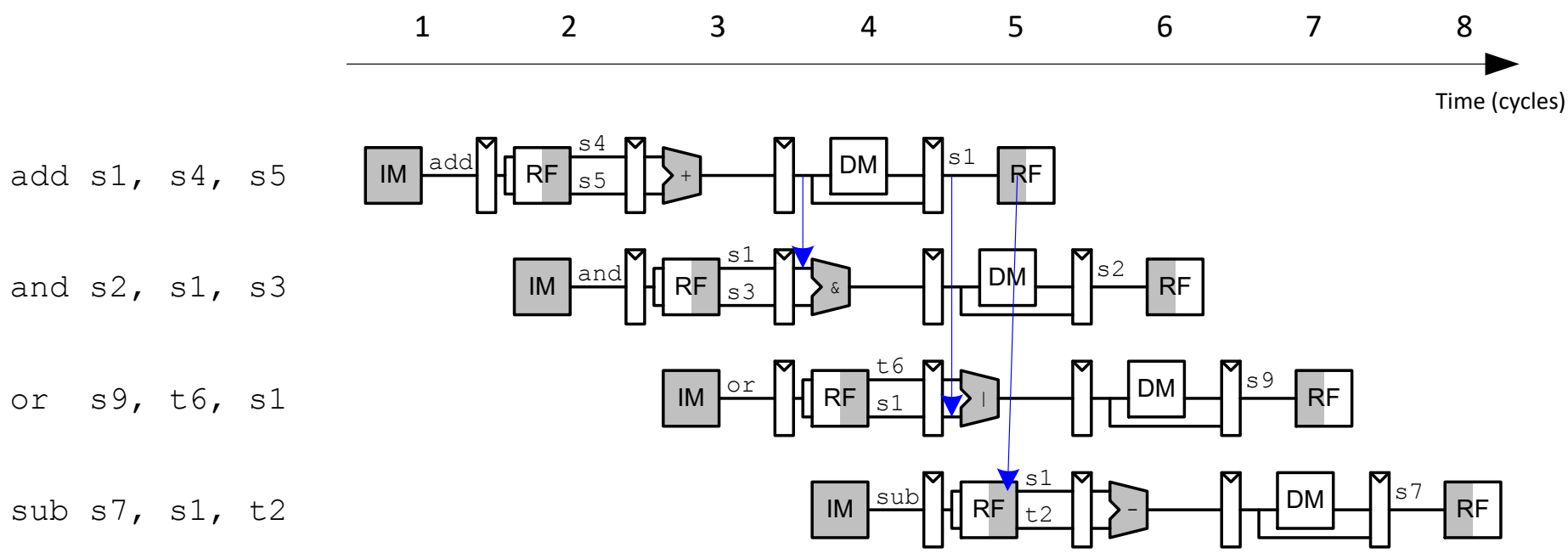


Data Forwarding

```
add s1, s4, s5
and s2, s1, s3
or  s9, t6, s1
sub s7, s1, t2
```

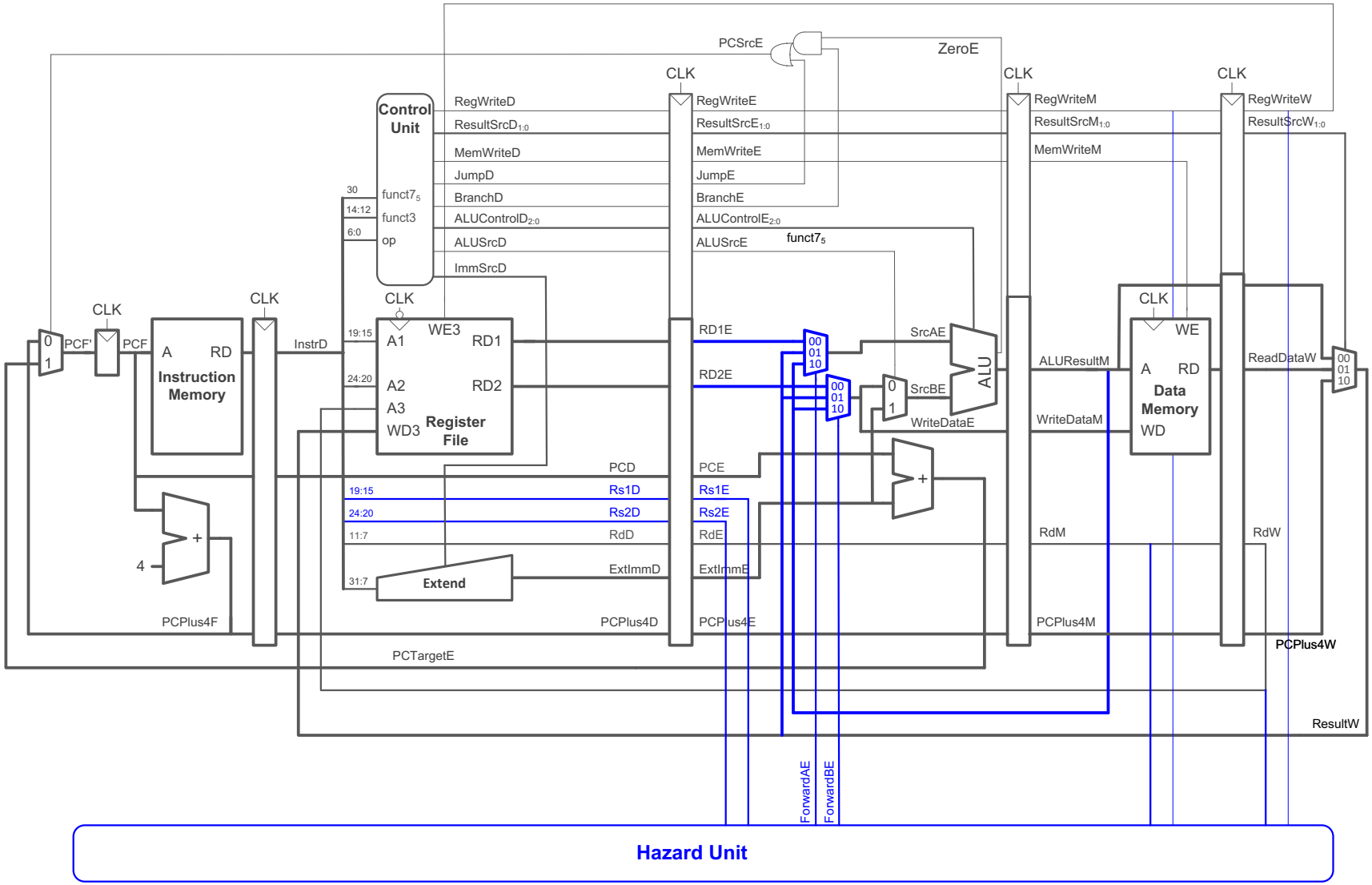


Data Forwarding



- Check if register read in Execute stage matches register written in Memory or Writeback stage
- If so, forward result

Data Forwarding



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

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

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)

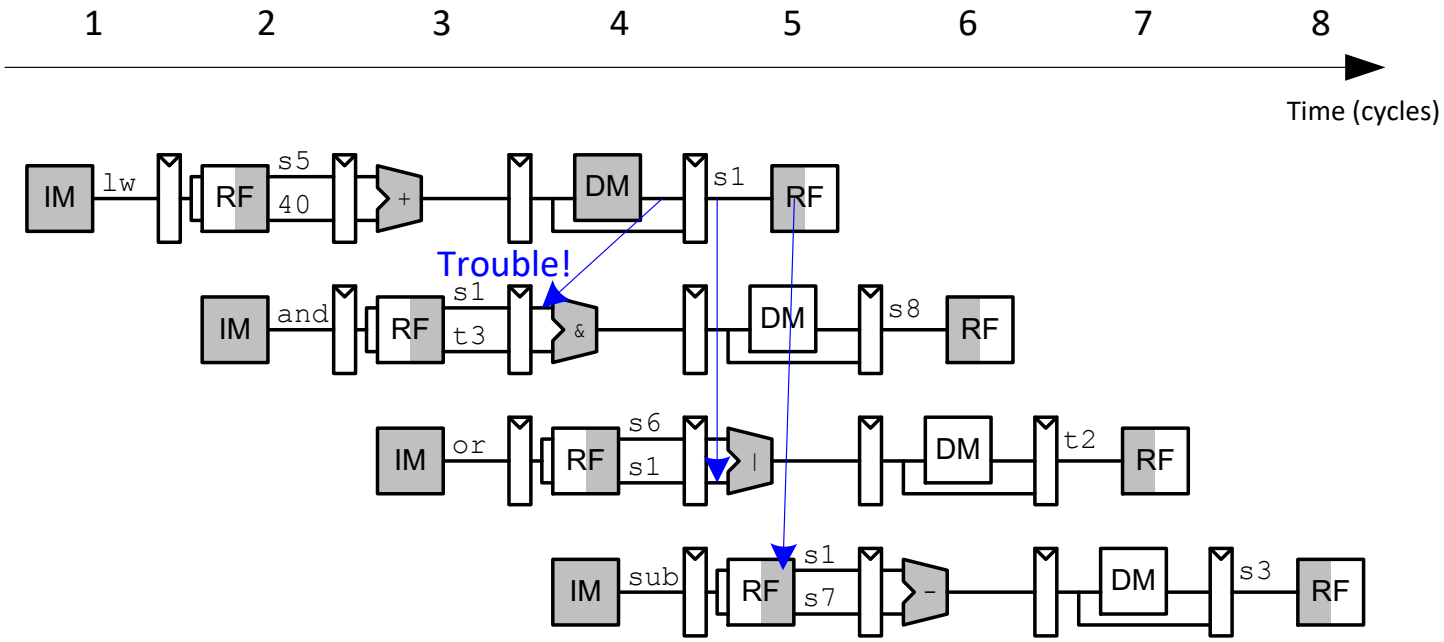
Equations for *Rs1*:

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

***ForwardBE* equation is similar (replace *Rs1E* with *Rs2E*)**

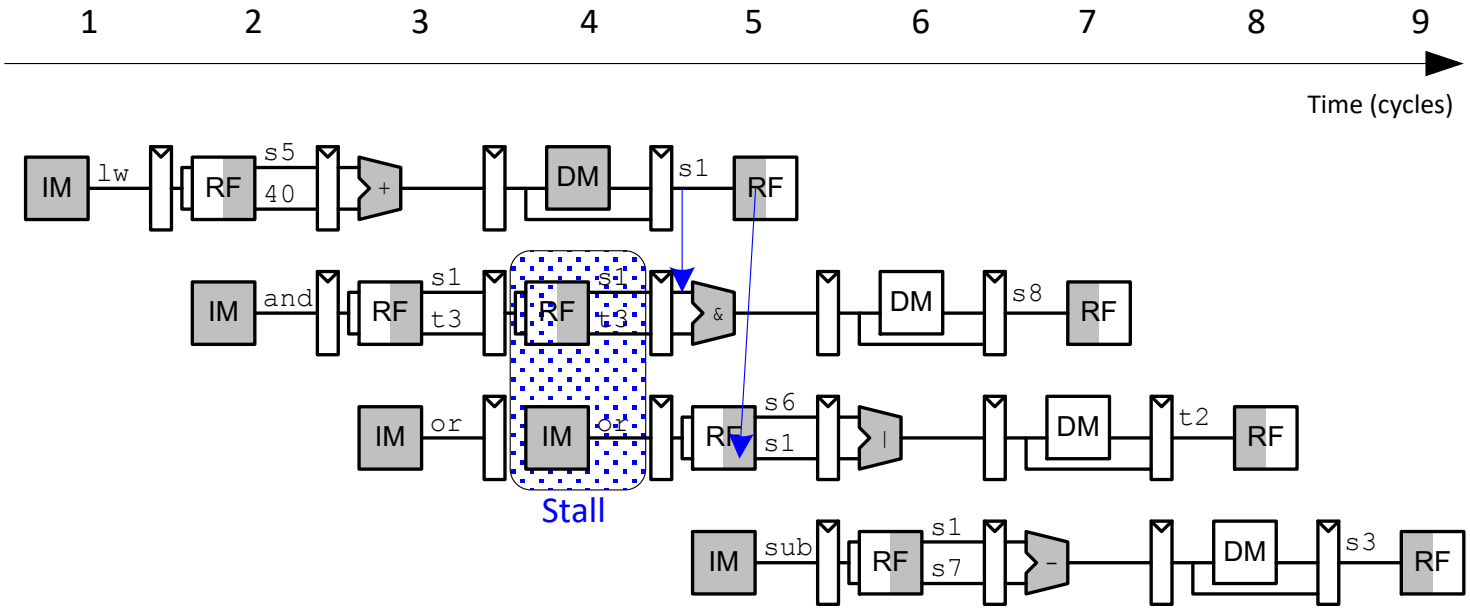
Stalling

```
lw   s1, 40(s5)
and  s8, s1, t3
or   t2, s6, s1
sub  s3, s1, s7
```

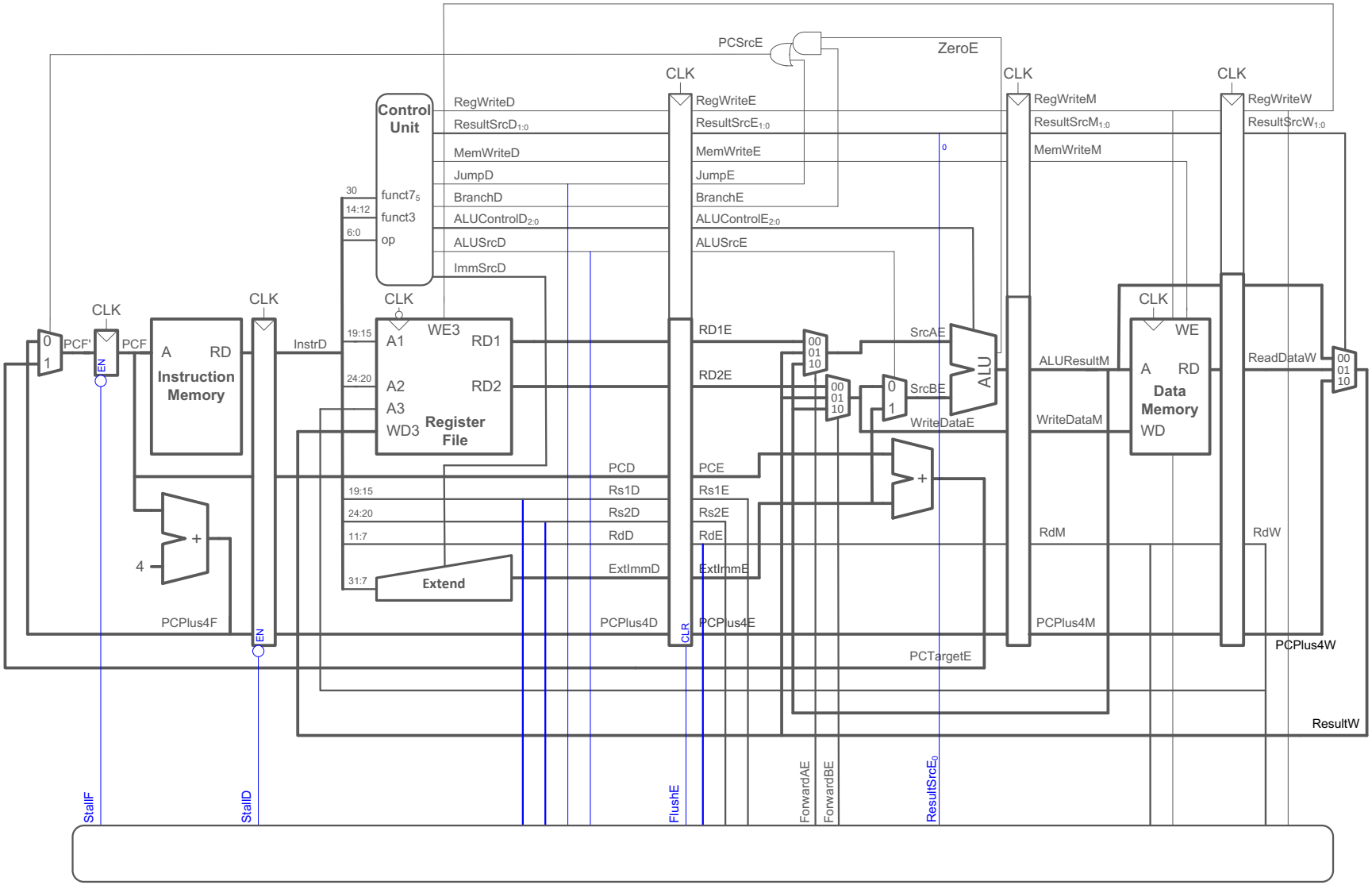


Stalling

```
lw   s1, 40(s5)
and  s8, s1, t3
or   t2, s6, s1
sub  s3, s1, s7
```



Stalling Hardware



Stalling Logic

- Is either source register in Decode stage the same as the one to be written by instruction in Execute stage? **AND**
- Is the instruction in Execute stage a `lw`?

lwStall = ((*Rs1D* == *RdE*) | (*Rs2D* == *RdE*)) & $ResultSrcE_0$

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

Stalling Logic

- Is **either source register** in Decode stage the same as the one to be written by instruction in Execute stage? **AND**
- Is the instruction in Execute stage a **`lw`**?

$$lwStall = ((Rs1D == RdE) \mid (Rs2D == RdE)) \;\&\; ResultSrcE_0$$

$$StallF = StallD = FlushE = lwStall$$

Stalling Logic

- Is either source register in Decode stage the same as the one to be written by instruction in Execute stage? **AND**
- Is the instruction in Execute stage a `lw`? **AND**
- Is the `lw`'s destination register (*RdE*) **not x0**?

lwStall = ((*Rs1D* == *RdE*) | (*Rs2D* == *RdE*) & ~*ALUSrcD*) & $ResultSrcE_0$ & (***RdE != 0***)

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

Stalling Logic

- Is either source register in Decode stage the same as the one to be written by instruction in Execute stage? **AND**
- Is the instruction in Execute stage a `lw`? **AND**
- Is the `lw`'s destination register (*RdE*) not x0? **AND**
- Are the source registers (*Rs1D* and *Rs2D*) **used**?

lwStall = ((*Rs1D* == *RdE*) | (*Rs2D* == *RdE*) & ***~ALUSrcD***) & $ResultSrcE_0$ & (*RdE != 0*) & ***(~JumpD)***

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

Stalling Logic

- Is either source register in Decode stage the same as the one to be written by instruction in Execute stage? **AND**
- Is the instruction in Execute stage a `lw`? **AND**
- Is the `lw`'s destination register (*RdE*) not x0? **AND**
- Are the source registers (*Rs1D* and *Rs2D*) **used**?

lwStall = ((*Rs1D* == *RdE*) | (*Rs2D* == *RdE*) & **~*ALUSrcD***) & *ResultSrc*$_0$ & (*RdE != 0*) & **(~*JumpD*)**

`JAL` doesn't use **rs1**, **rs2**

`I-type` instructions don't use **rs2** (*ALUSrcD* = 1 selects *ExtImm* as *SrcB* of ALU)

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

Stalling Logic

- Is **either source register** in Decode stage the same as the one to be written by instruction in Execute stage? **AND**
- Is the instruction in Execute stage a **`lw`**? **AND**
- Is the `lw`'s destination register (*RdE*) **not x0**? **AND**
- Are the source registers (*Rs1D* and *Rs2D*) **used**?

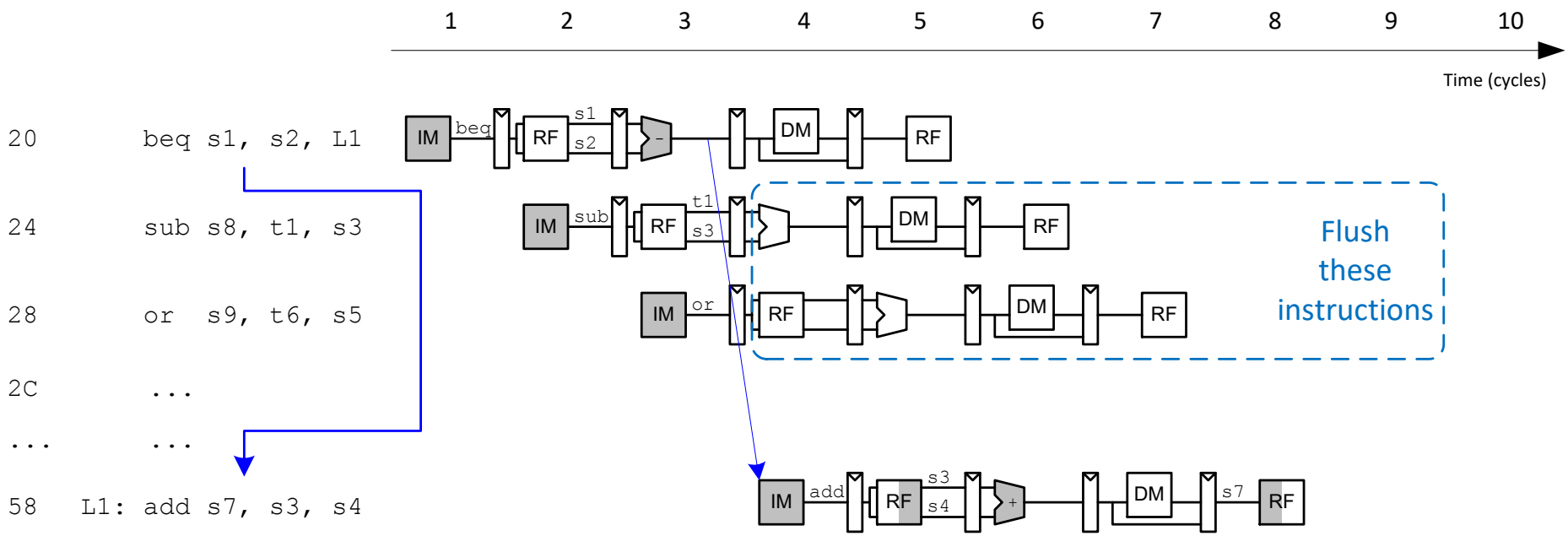
lwStall = (***Rs1D == RdE***) | (***Rs2D == RdE***) & ***~ALUSrcD***) &
ResultSrcE$_0$ &
(***RdE != 0***) &
(***~JumpD***)

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

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

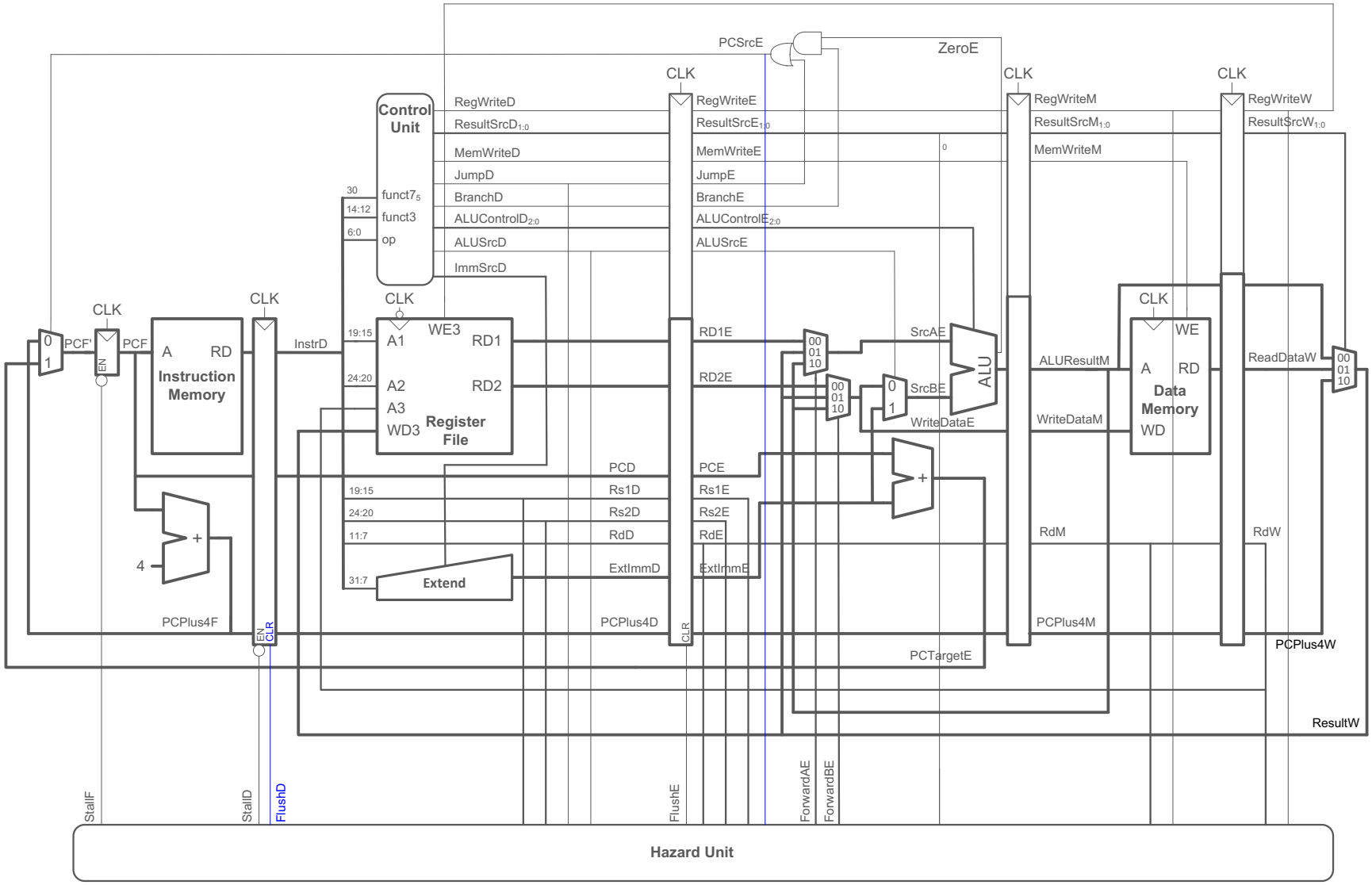
Control Hazards



Branch misprediction penalty

- number of instruction flushed when branch is taken (2)

Flushing Hardware for Control Hazards



Control 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 | PCSrcE

Pipeline Hazard Summary

Forward to solve data hazards when possible

if ((*Rs1E* == *RdM*) & *RegWriteM*) & (*Rs1E* != 0) then
ForwardAE = 10

else if ((*Rs1E* == *RdW*) & *RegWriteW*) & (*Rs1E* != 0) then
ForwardAE = 01

else *ForwardAE* = 00

Stall when a load hazard occurs

lwStall = ((*Rs1D* == *RdE*) | (*Rs2D* == *RdE*) & ~*ALUSrcD*) & $ResultSrcE_0$ &
(*RdE* != 0) & ~*JumpD*

StallF = *lwStall*

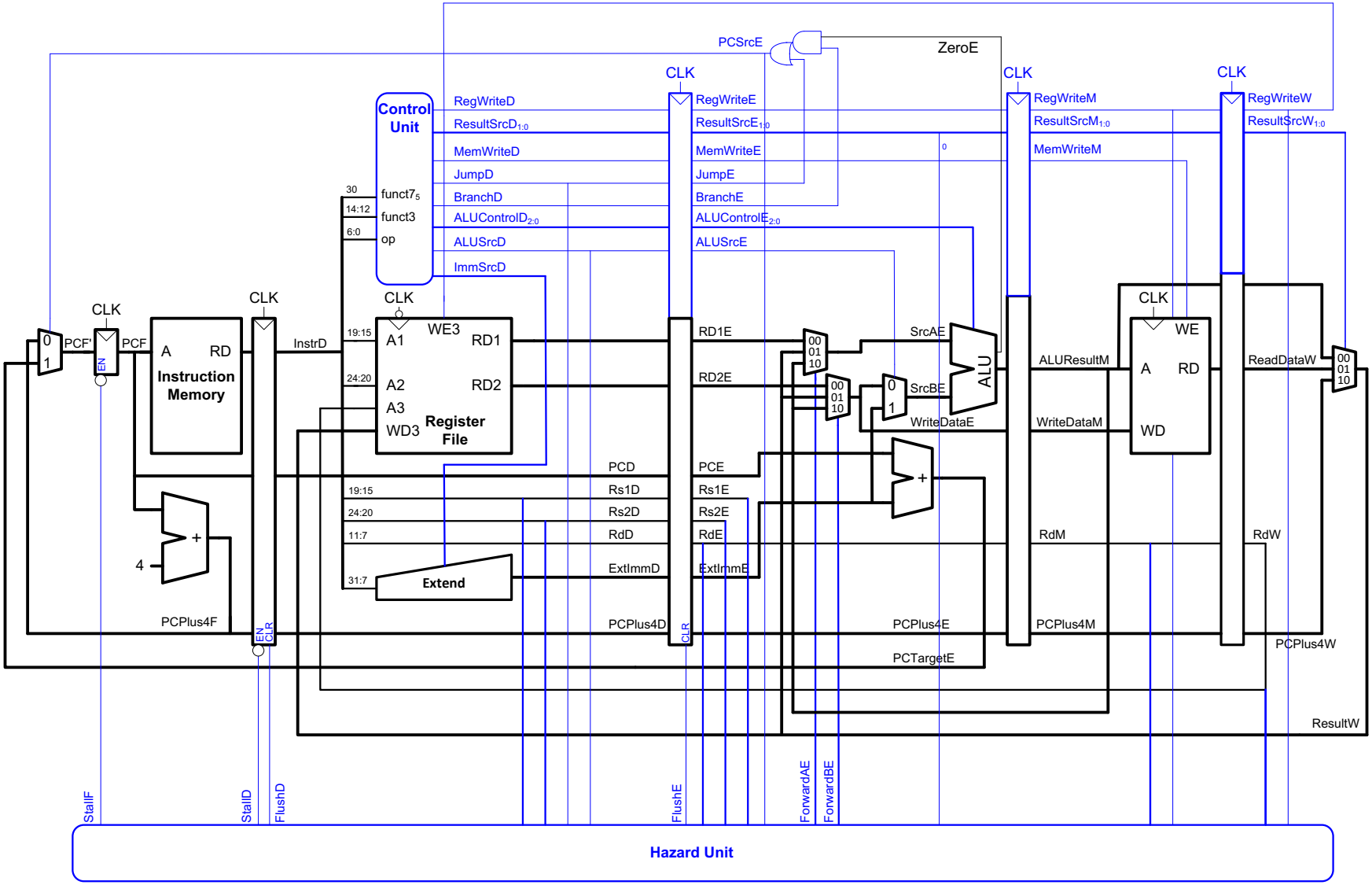
StallD = *lwStall*

Flush when a branch is taken or a load introduces a bubble

FlushD = *PCSrcE*

FlushE = *lwStall* | *PCSrcE*

RISC-V Pipelined Processor with Hazard Unit



Pipelined Performance Example

- **SPECINT2000 benchmark:**
  - 25% loads
  - 10% stores
  - 13% branches
  - 52% R-type
- **Suppose:**
  - 40% of loads used by next instruction
  - 50% of branches mispredicted
- **What is the average CPI?**

Pipelined Performance Example

- **SPECINT2000 benchmark:**
  - 25% loads
  - 10% stores
  - 13% branches
  - 52% R-type
- **Suppose:**
  - 40% of loads used by next instruction
  - 50% of branches mispredicted
- **What is the average CPI?**
  - Load CPI = 1 when not stalling, 2 when stalling

    So, $\mathbf{CPI_{lw}} = 1(0.6) + 2(0.4) = 1.4$
  - Branch CPI = 1 when not stalling, 3 when stalling

    So, $\mathbf{CPI_{beq}} = 1(0.5) + 3(0.5) = 2$

Average CPI = $(0.25)(1.4) + (0.1)(1) + (0.13)(2) + (0.52)(1)$ **= 1.23**

Pipelined Performance

Pipelined processor critical path:

T_{c3} = max of

$t_{pcq} + t_{mem} + t_{setup}$	Fetch
$2(t_{RFread} + t_{setup})$	Decode
$t_{pcq} + 4t_{mux} + t_{ALU} + t_{AND\text{-}OR} + t_{setup}$	Execute
$t_{pcq} + t_{mem} + t_{setup}$	Memory
$2(t_{pcq} + t_{mux} + t_{RFwrite})$	Writeback

Pipelined Performance

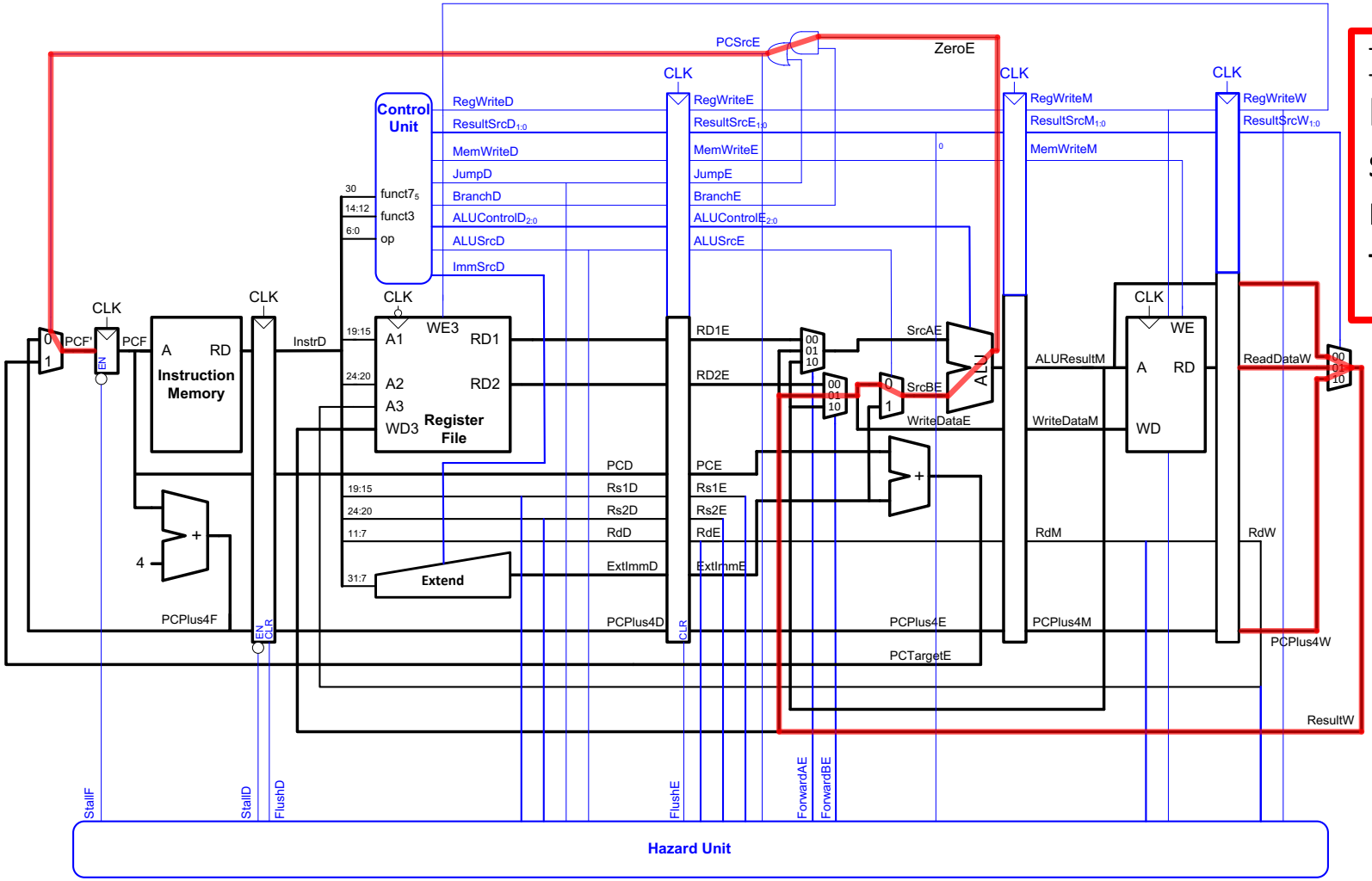
Pipelined processor critical path:

T_{c3} = max of

$t_{pcq} + t_{mem} + t_{setup}$	**Fetch**
$2(t_{RFread} + t_{setup})$	**Decode**
$t_{pcq} + 4t_{mux} + t_{ALU} + t_{AND\text{-}OR} + t_{setup}$	**Execute**
$t_{pcq} + t_{mem} + t_{setup}$	**Memory**
$2(t_{pcq} + t_{mux} + t_{RFwrite})$	**Writeback**

- Decode and Writeback stages both use the register file in each cycle
- So each stage gets half of the cycle time ($T_c/2$) to do their work
- Or, stated a different way, **2x of their work** must fit in a cycle (T_c)

Pipelined Processor Critical Path



`beq` in Execute stage that requires forwarding

$$t_{pcq} + 4t_{mux} + t_{ALU} + t_{AND\text{-}OR} + t_{setup}$$

Execute

Pipelined Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	30
AND-OR gate	$t_{AND\text{-}OR}$	20
ALU	t_{ALU}	120
Decoder (control unit)	t_{dec}	35
Memory read	t_{mem}	200
Register file read	t_{RFread}	100
Register file setup	$t_{RFsetup}$	60

Cycle time: $T_{c3} = t_{pcq} + 4t_{mux} + t_{ALU} + t_{AND\text{-}OR} + t_{setup}$

$= (40 + 4(30) + 120 + 20 + 50)\ \text{ps} = \mathbf{350\ ps}$

Pipelined Performance Example

Program with 100 billion instructions

Execution Time = (# instructions) $\times$ CPI $\times$ T_c

= $(100 \times 10^9)(1.23)(350 \times 10^{-12})$

= 43 seconds

Processor Performance Comparison

Processor	Execution Time (seconds)	Speedup (single-cycle as baseline)
Single-cycle	75	1
Multicycle	144	0.5
Pipelined	43	1.7