Introduction

- **Microarchitecture:** how to implement an architecture in hardware
- Processor:
  - **Datapath:** functional blocks
  - **Control:** control signals

Application Software	programs
Operating Systems	device drivers
Architecture	instructions registers
Micro-architecture	datapaths controllers
Logic	adders memories
Digital Circuits	AND gates NOT gates
Analog Circuits	amplifiers filters
Devices	transistors diodes
Physics	electrons

Microarchitecture

- Multiple implementations for a single architecture:
  - **Single-cycle:** Each instruction executes in a single cycle
  - **Multicycle:** Each instruction is broken into series of shorter steps
  - **Pipelined:** Each instruction broken up into series of steps & multiple instructions execute at once

Processor Performance

- Program execution time

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

- Definitions:
  - CPI: Cycles/instruction
  - clock period: seconds/cycle
  - IPC: instructions/cycle = IPC
- Challenge is to satisfy constraints of:
  - Cost
  - Power
  - Performance

MIPS Processor

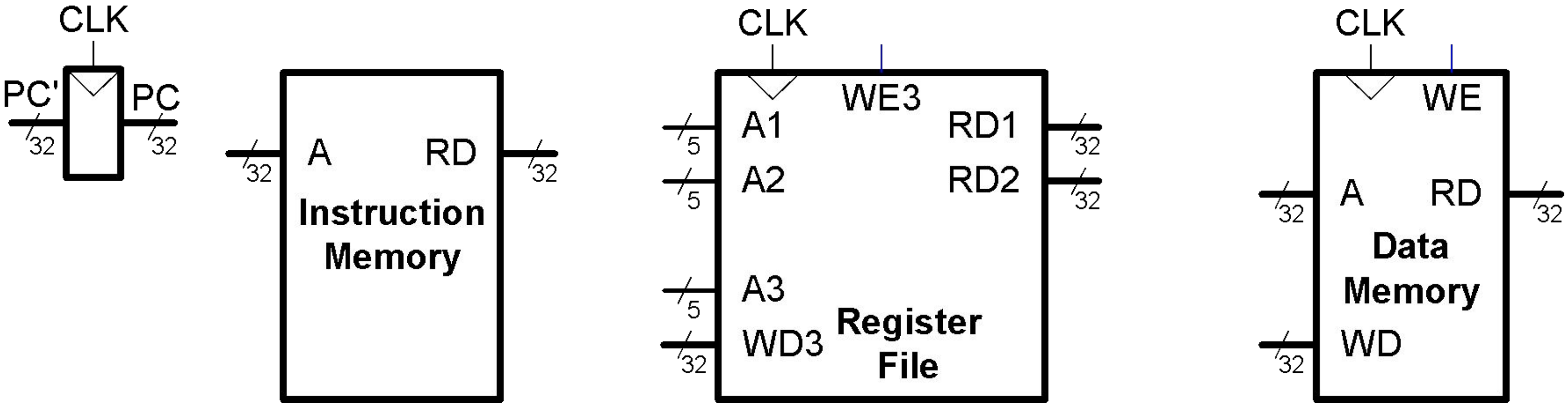
- Consider subset of MIPS instructions:
  - R-type instructions: `and`, `or`, `add`, `sub`, `slt`:
  - `x = y binop z`
  - Memory instructions:
  - `x = *(y+imm) (lw for short)`,
  - `*(x+imm) = y (sw for short)`
  - Branch instructions: `beq`:
  - `if x relop y goto L`

ELSEVIER

Architectural State

- Determines everything about a processor:
  - PC
  - 32 registers
  - Memory

MIPS State Elements

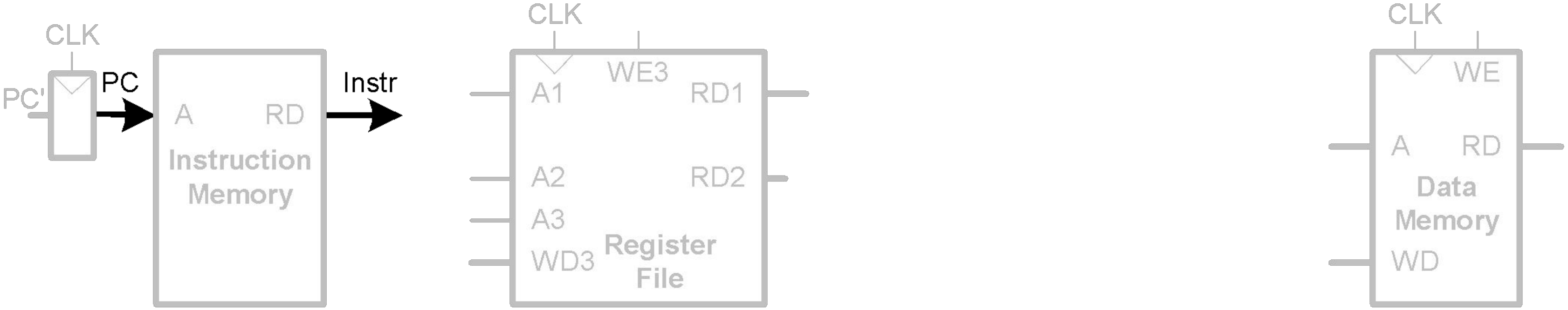


Single-Cycle MIPS Processor

- Datapath
- Control

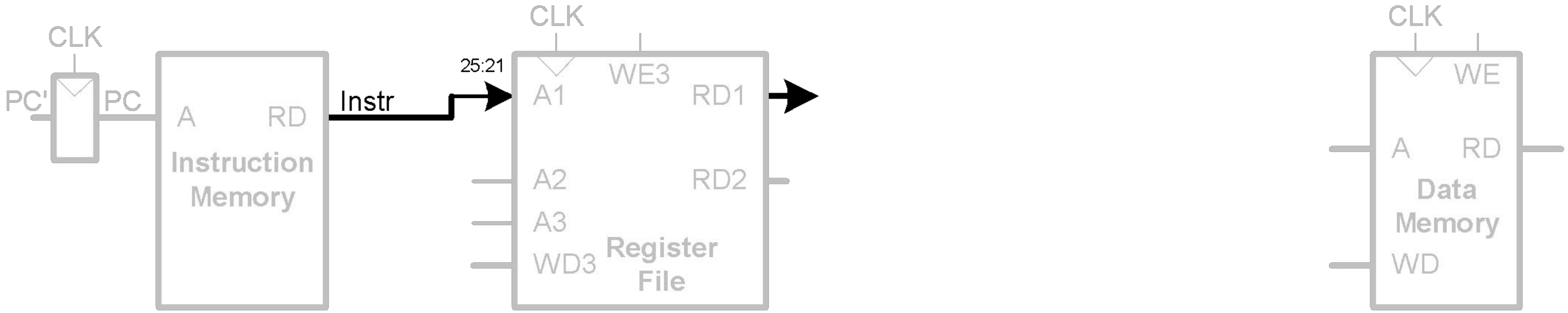
Single-Cycle Datapath: `lw` fetch

STEP 1: Fetch instruction



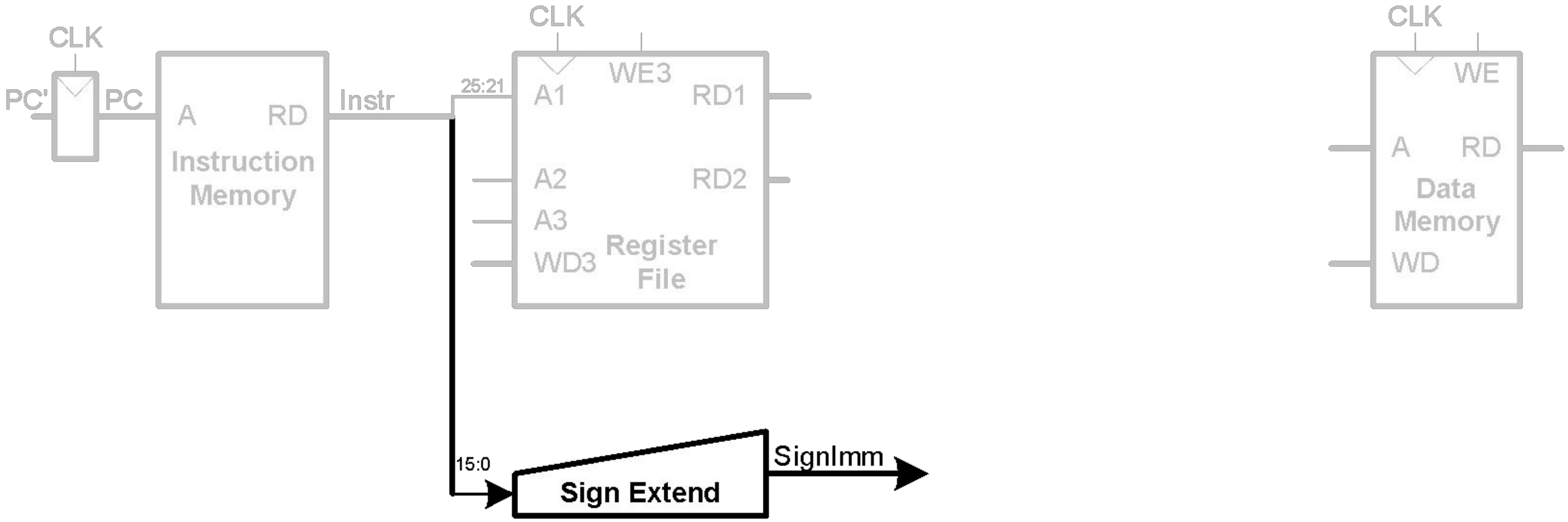
Single-Cycle Datapath: `lw` Register Read

STEP 2: Read source operands from RF



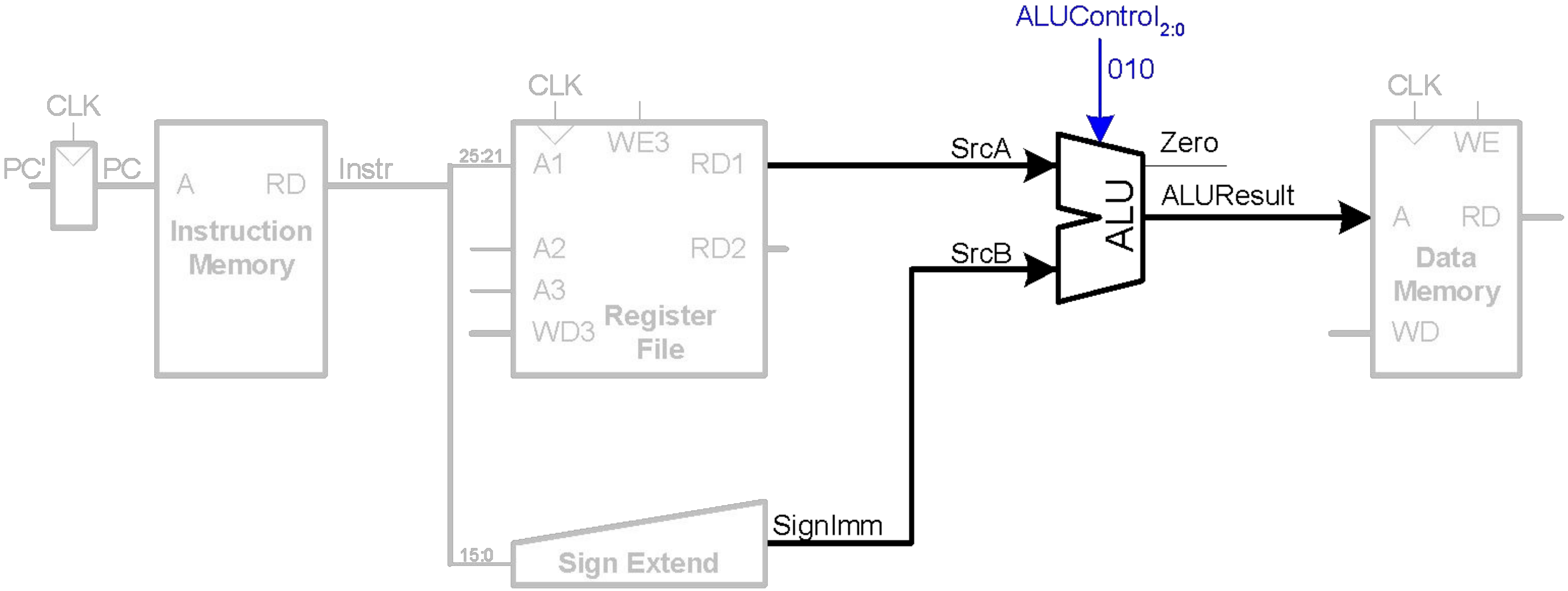
Single-Cycle Datapath: `lw` Immediate

STEP 3: Sign-extend the immediate



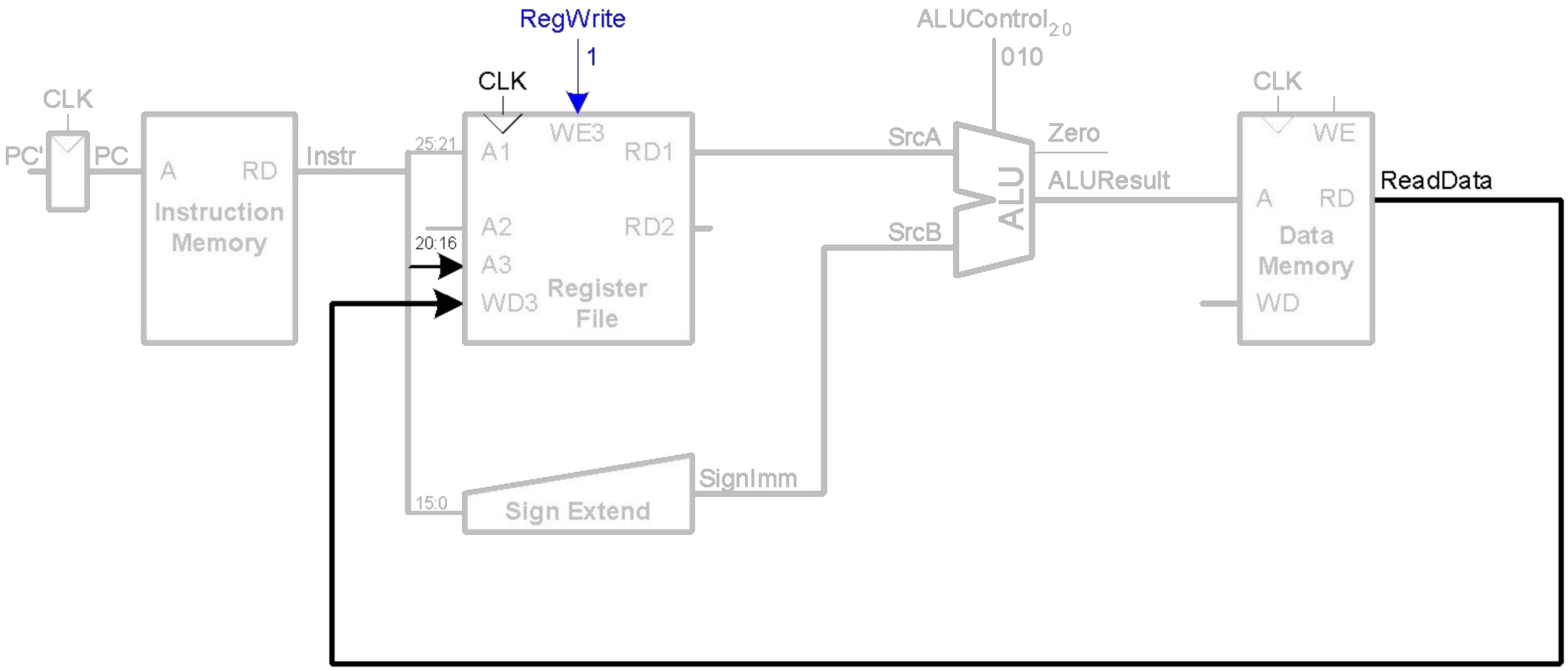
Single-Cycle Datapath: `lw` address

STEP 4: Compute the memory address



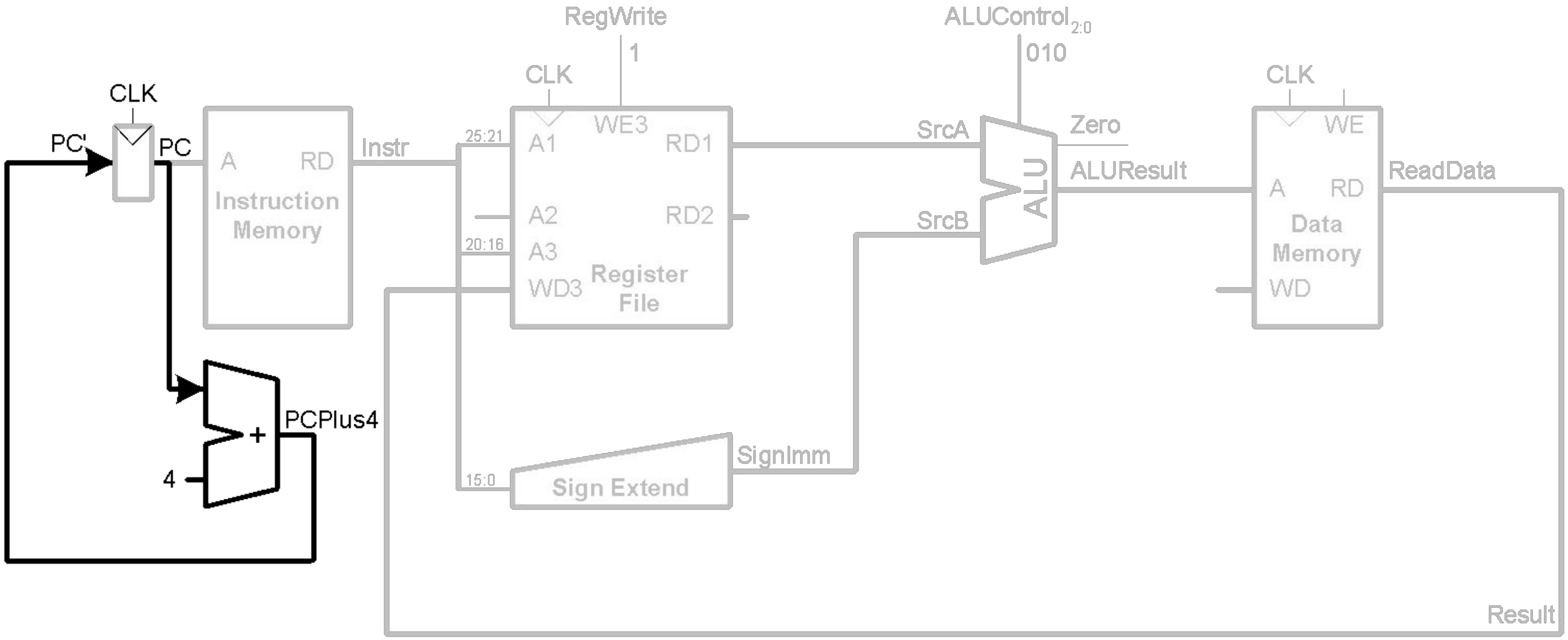
Single-Cycle Datapath: `lw` Memory Read

- **STEP 5:** Read data from memory and write it back to register file



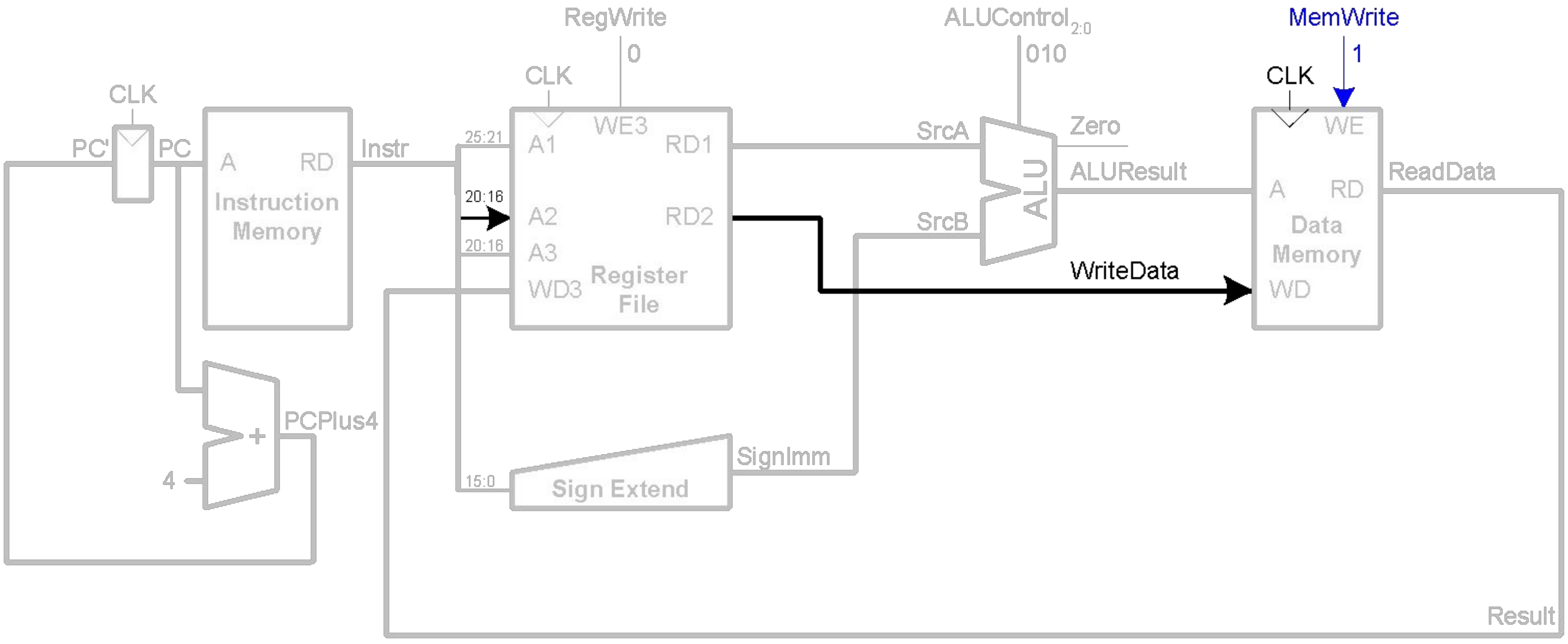
Single-Cycle Datapath: lw PC Increment

STEP 6: Determine address of next instruction



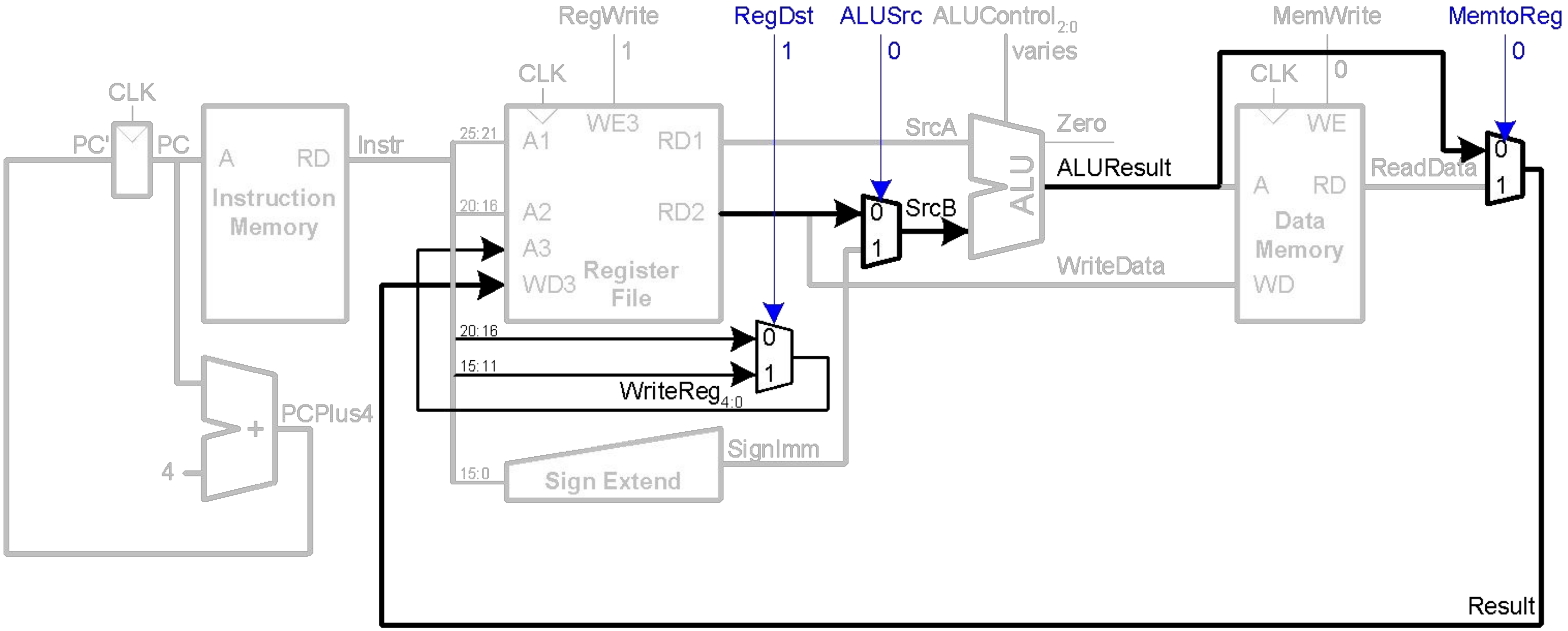
Single-Cycle Datapath: `sw`

Write data in `rt` to memory



Single-Cycle Datapath: R-Type

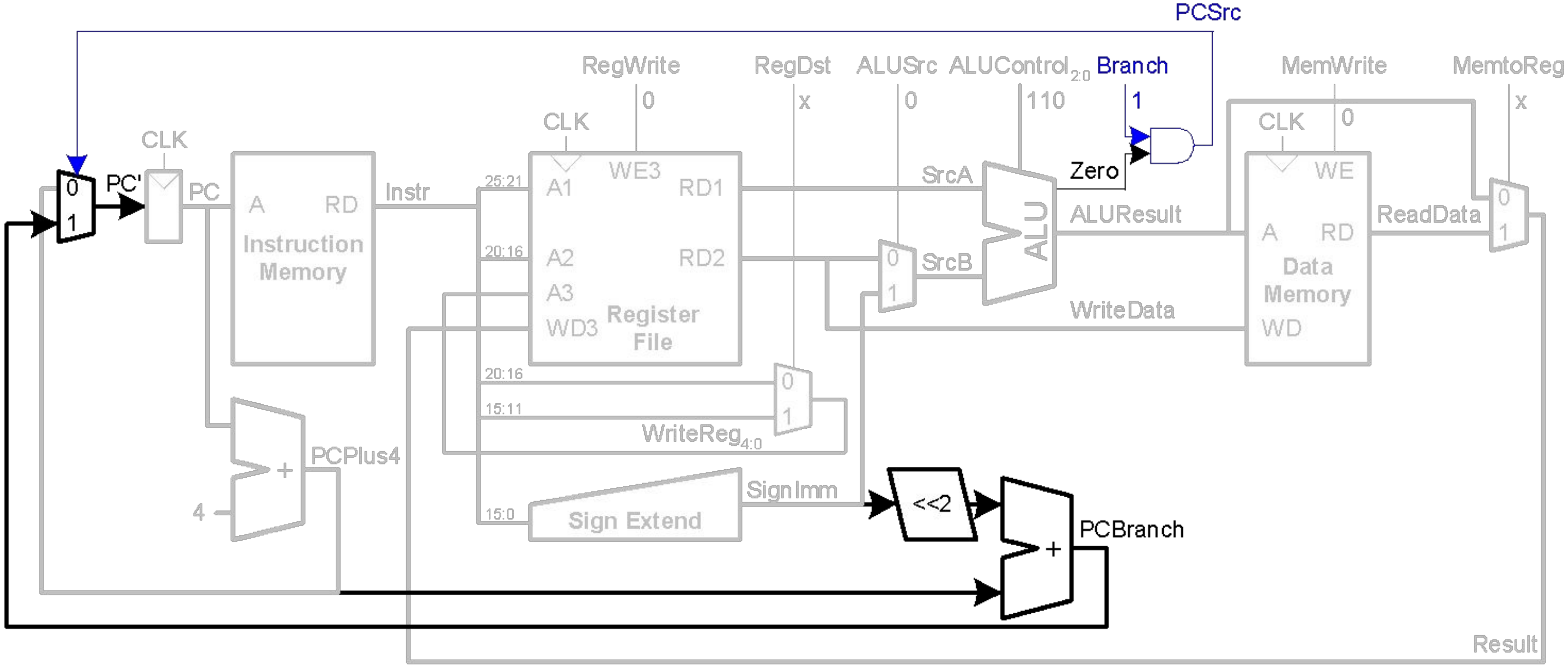
- Read from `rs` and `rt`
- Write *ALUResult* to register file
- Write to `rd` (instead of `rt`)



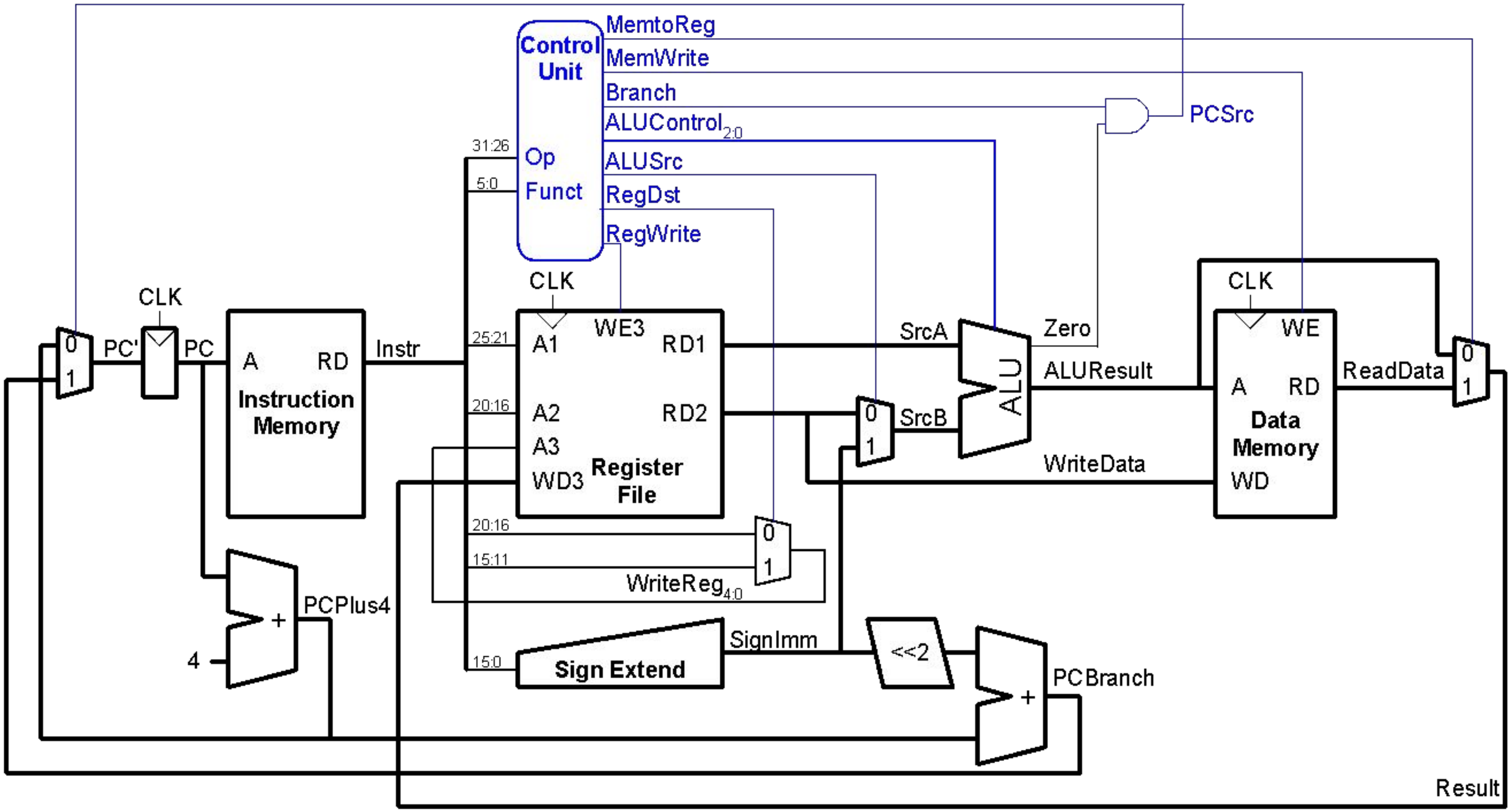
Single-Cycle Datapath: beq

- Determine whether values in rs and rt are equal
- Calculate branch target address:

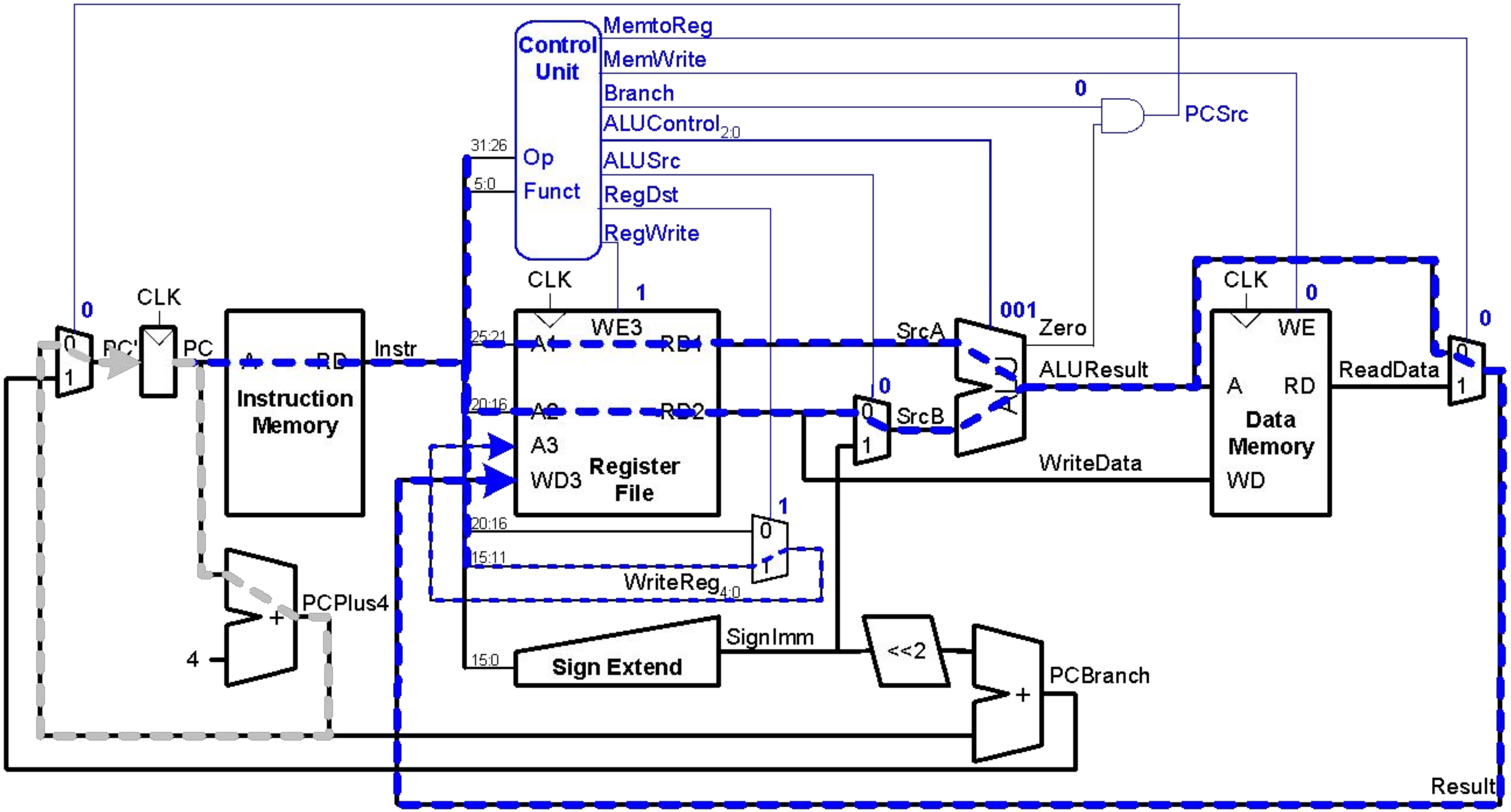
  BTA = (sign-extended immediate << 2) + (PC+4)



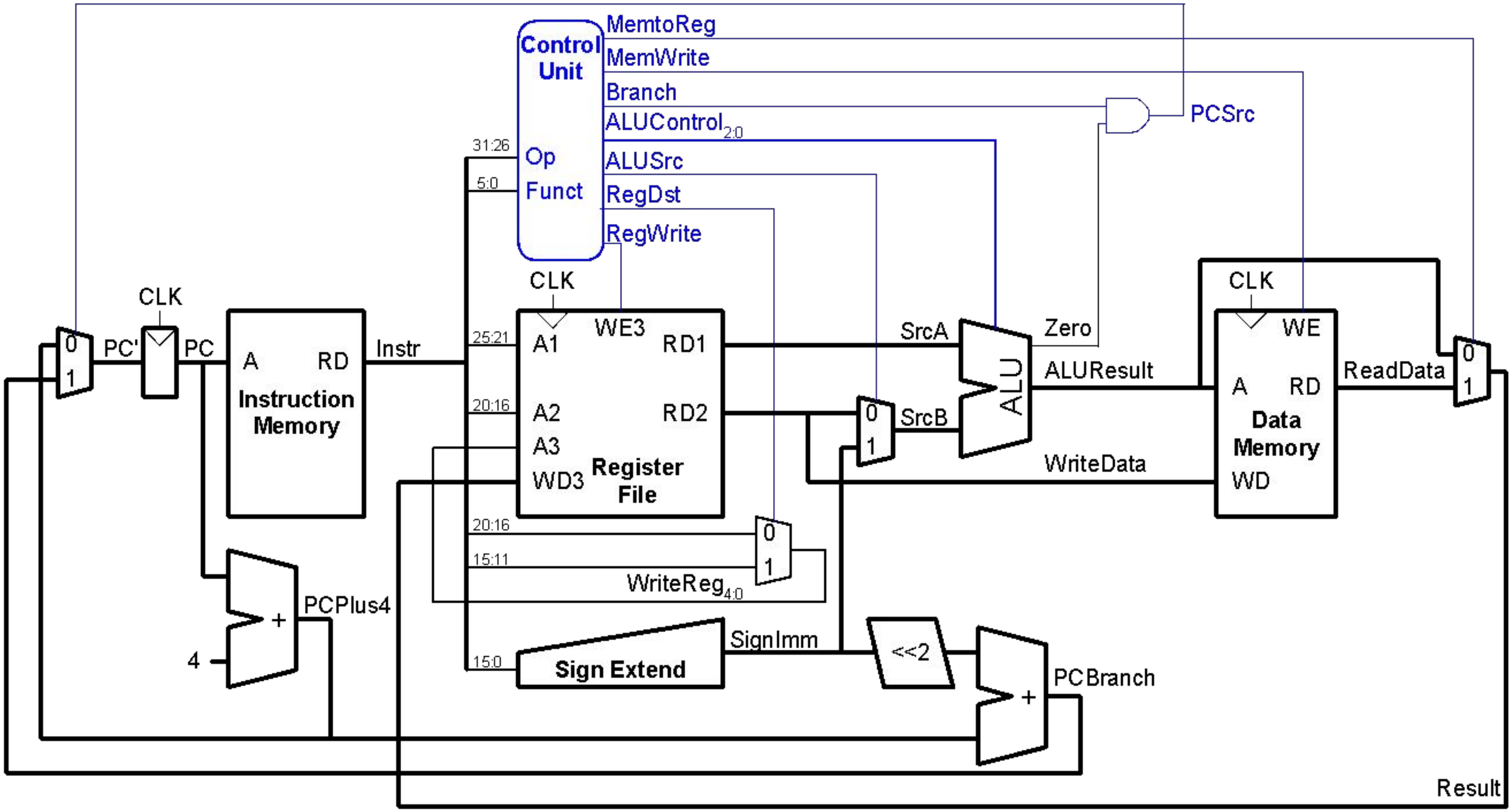
Single-Cycle Processor



Single-Cycle Datapath: `or`

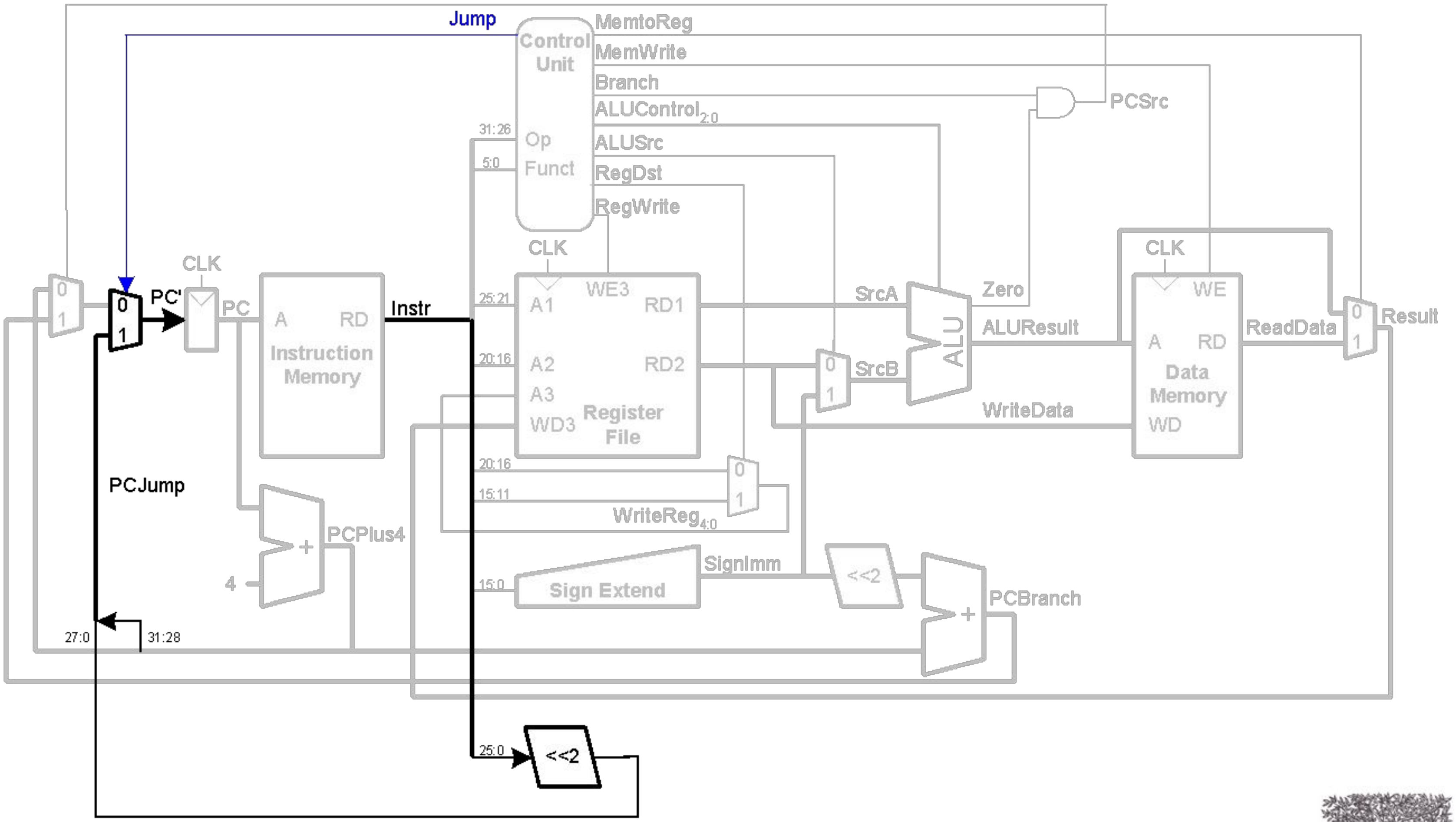


Extended Functionality: `addi`



No change to datapath

Extended Functionality: j



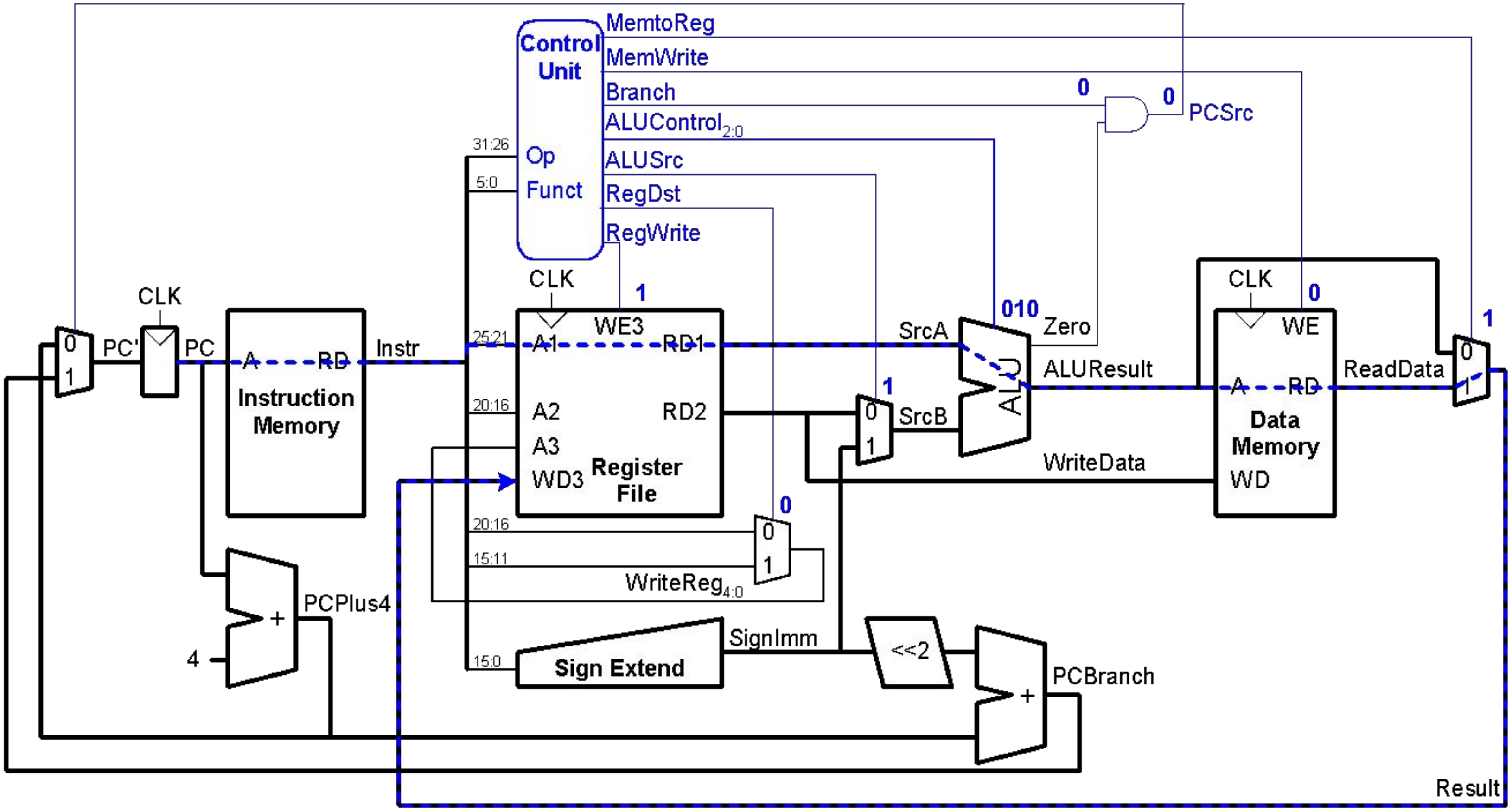
Review: Processor Performance

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:

$$T_c = t_{pcq\_PC} + t_{\text{mem}} + \max(t_{RF\text{read}}, t_{sext} + t_{\text{mux}}) + t_{\text{ALU}} + t_{\text{mem}} + t_{\text{mux}} + t_{RF\text{setup}}$$

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

MICROARCHITECTURE

Single-Cycle 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

$T_c = ?$

Single-Cycle 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	$t_{RF\mathrm{read}}$	150
Register file setup	$t_{RF\mathrm{setup}}$	20

$$
\begin{aligned}
T_c &= t_{pcq\_PC} + 2t_{\mathrm{mem}} + t_{RF\mathrm{read}} + t_{\mathrm{mux}} + t_{\mathrm{ALU}} + t_{RF\mathrm{setup}} \\
&= [30 + 2(250) + 150 + 25 + 200 + 20]\ \mathrm{ps} \\
&= 925\ \mathrm{ps}
\end{aligned}
$$

Single-Cycle Performance Example

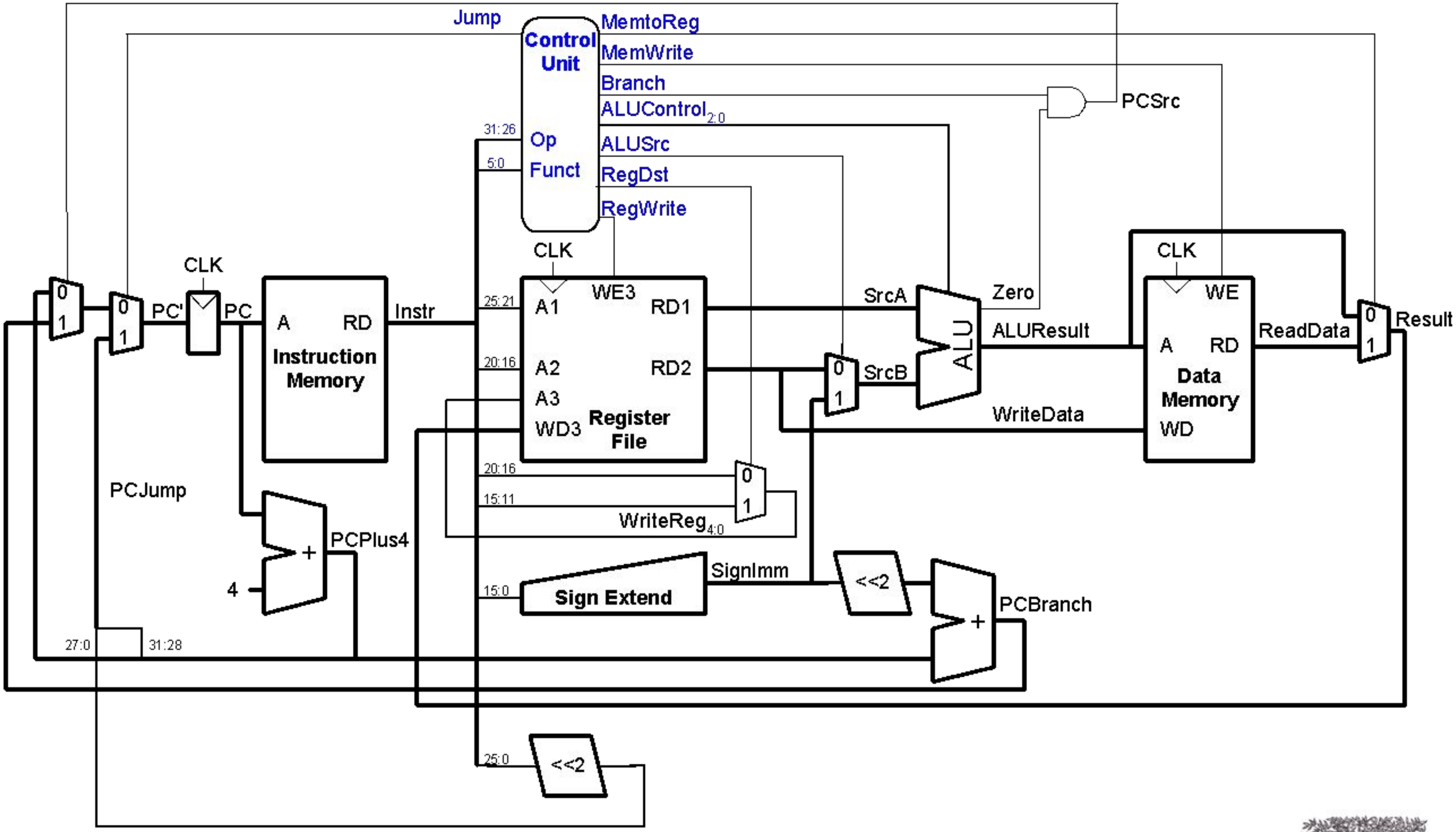
Program with 100 billion instructions:

Execution Time = # instructions x CPI x T_C

$= (100 \times 10^{9})(1)(925 \times 10^{-12}\ \text{s})$

= **92.5 seconds**

Review: Single-Cycle Processor



Parallelism

- **Two types of parallelism:**
  - **Spatial parallelism**
    - duplicate hardware performs multiple tasks at once
  - **Temporal parallelism**
    - task is broken into multiple stages
    - also called pipelining
    - for example, an assembly line

Parallelism Definitions

- **Token:** Group of inputs processed to produce group of outputs
- **Latency:** Time for one token to pass from start to end
- **Throughput:** Number of tokens produced per unit time

Parallelism increases throughput

Parallelism Example

- Ben Bitdiddle bakes cookies to celebrate traffic light controller installation
- 5 minutes to roll cookies
- 15 minutes to bake
- What is the latency and throughput without parallelism?

Parallelism Example

- Ben Bitdiddle bakes cookies to celebrate traffic light controller installation
- 5 minutes to roll cookies
- 15 minutes to bake
- What is the latency and throughput without parallelism?

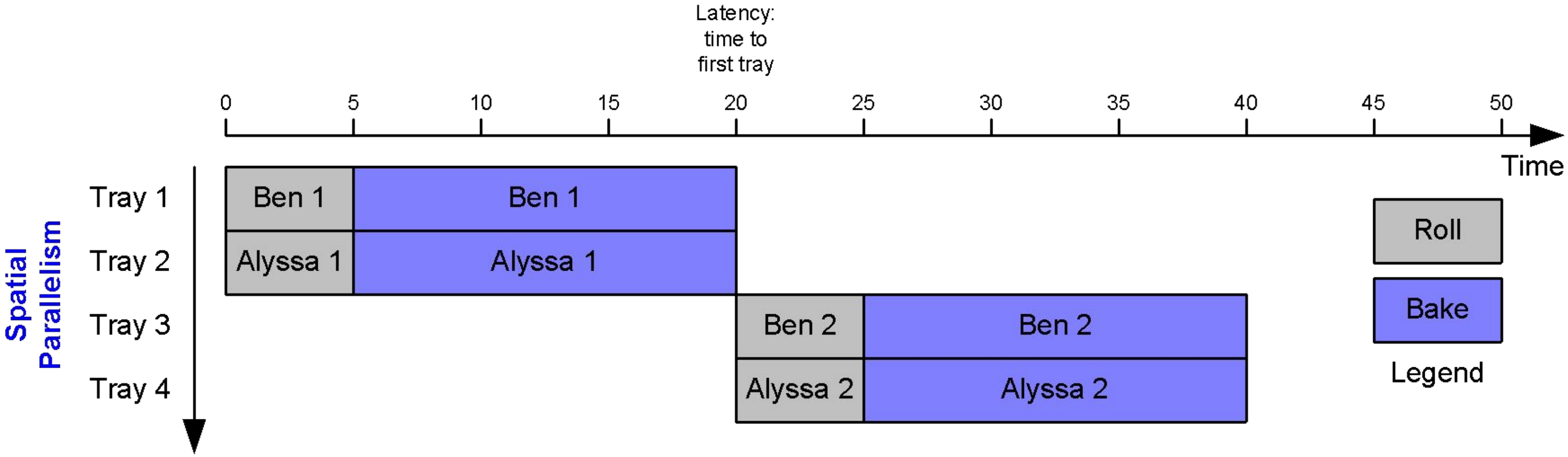
Latency = 5 + 15 = 20 minutes = **1/3 hour**

Throughput = 1 tray/ 1/3 hour = **3 trays/hour**

Parallelism Example

- What is the latency and throughput if Ben uses parallelism?
  - **Spatial parallelism:** Ben asks Allysa P. Hacker to help, using her own oven
  - **Temporal parallelism:**
    - two stages: rolling and baking
    - He uses two trays
    - While first batch is baking, he rolls the second batch, etc.

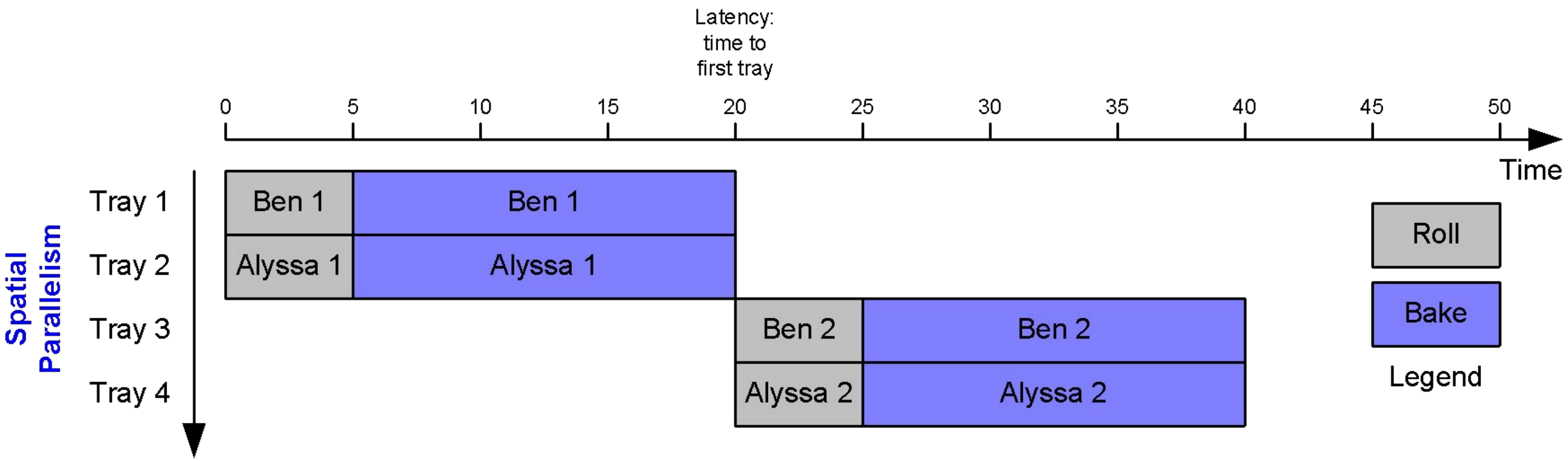
Spatial Parallelism



Latency = ?

Throughput = ?

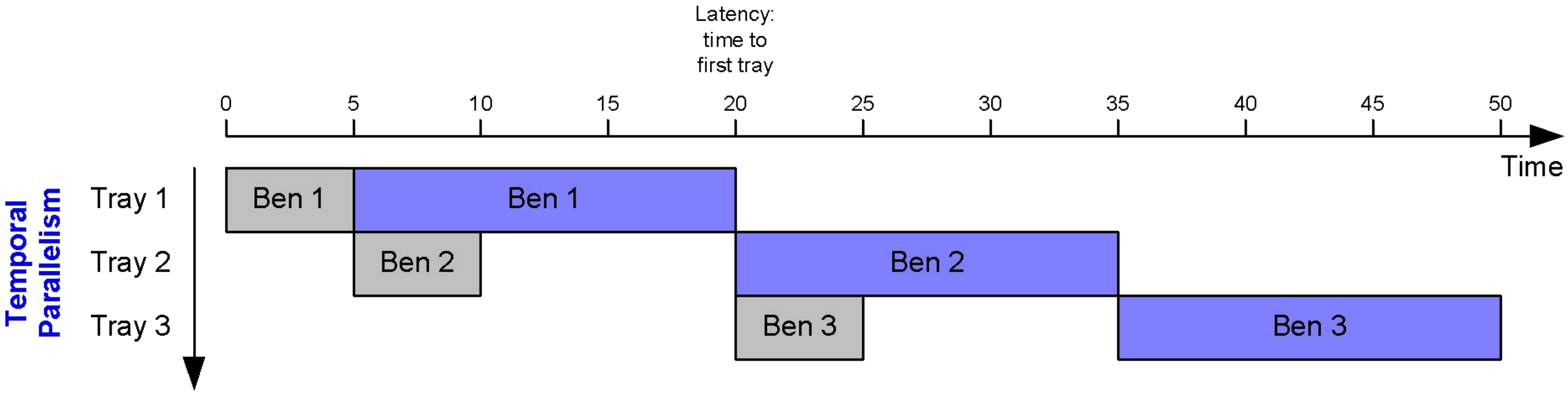
Spatial Parallelism



Latency = 5 + 15 = 20 minutes = **1/3 hour**

Throughput = 2 trays/ 1/3 hour = **6 trays/hour**

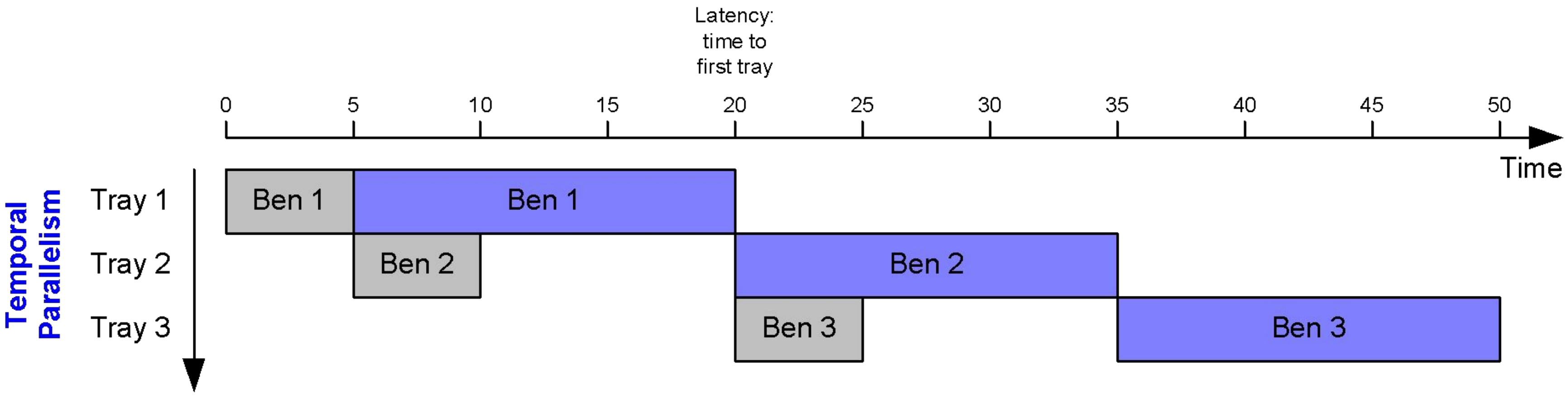
Temporal Parallelism



Latency = ?

Throughput = ?

Temporal Parallelism



Latency = 5 + 15 = 20 minutes = **1/3 hour**

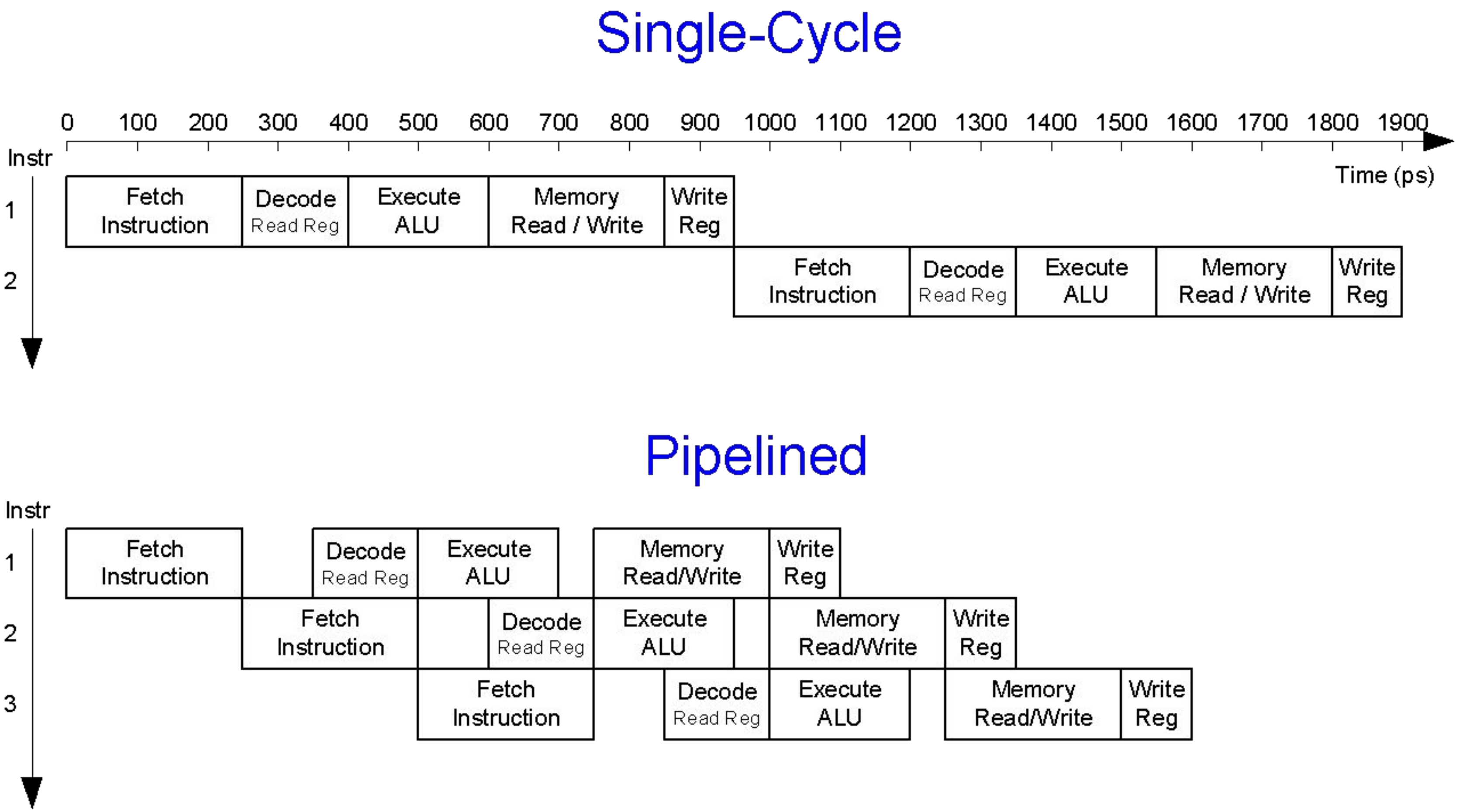
Throughput = 1 trays/ 1/4 hour = **4 trays/hour**

Using both techniques, the throughput would be **8 trays/hour**

Pipelined MIPS Processor

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

Single-Cycle vs. Pipelined



Pipelined Processor Abstraction

```
lw  $s2, 40($0)
```
$s2 = *($0 + 40)

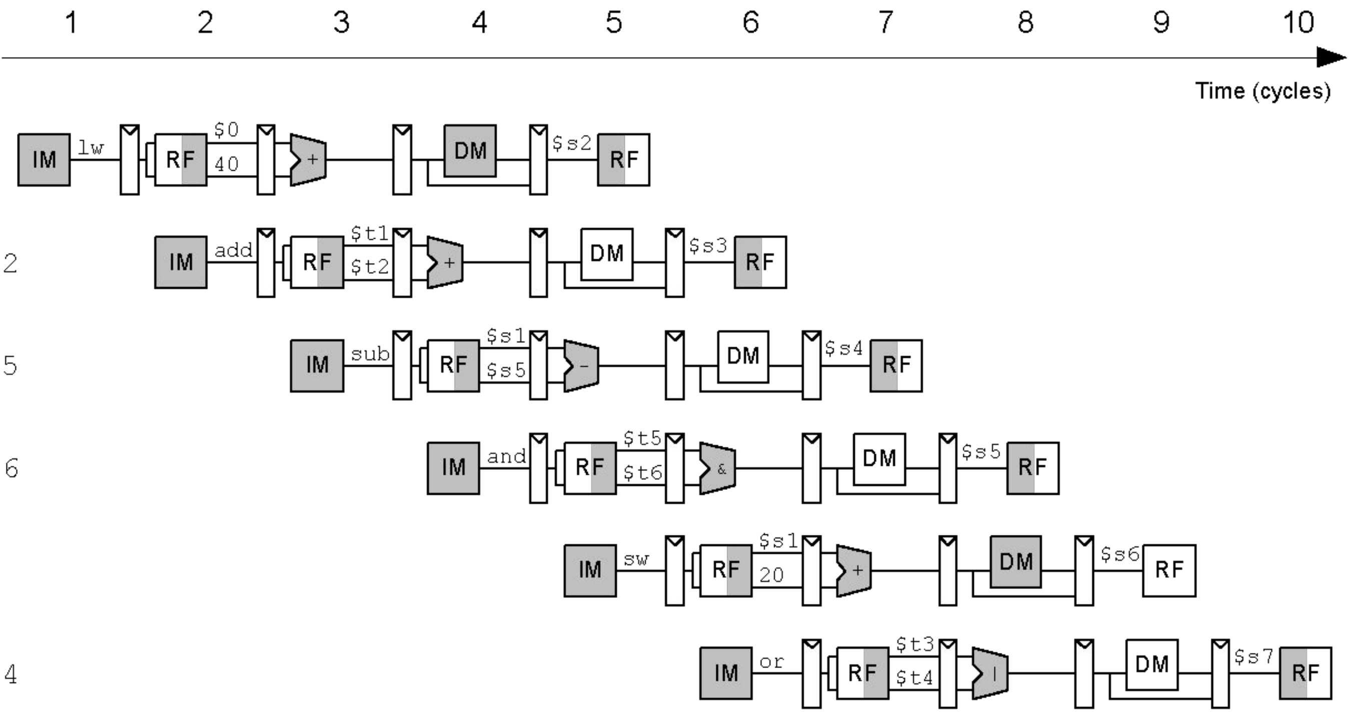
```
add $s3, $t1, $t2
```
$s3 = $t1 + $t2

```
sub $s4, $s1, $s5
```
$s4 = $s1 - $s5

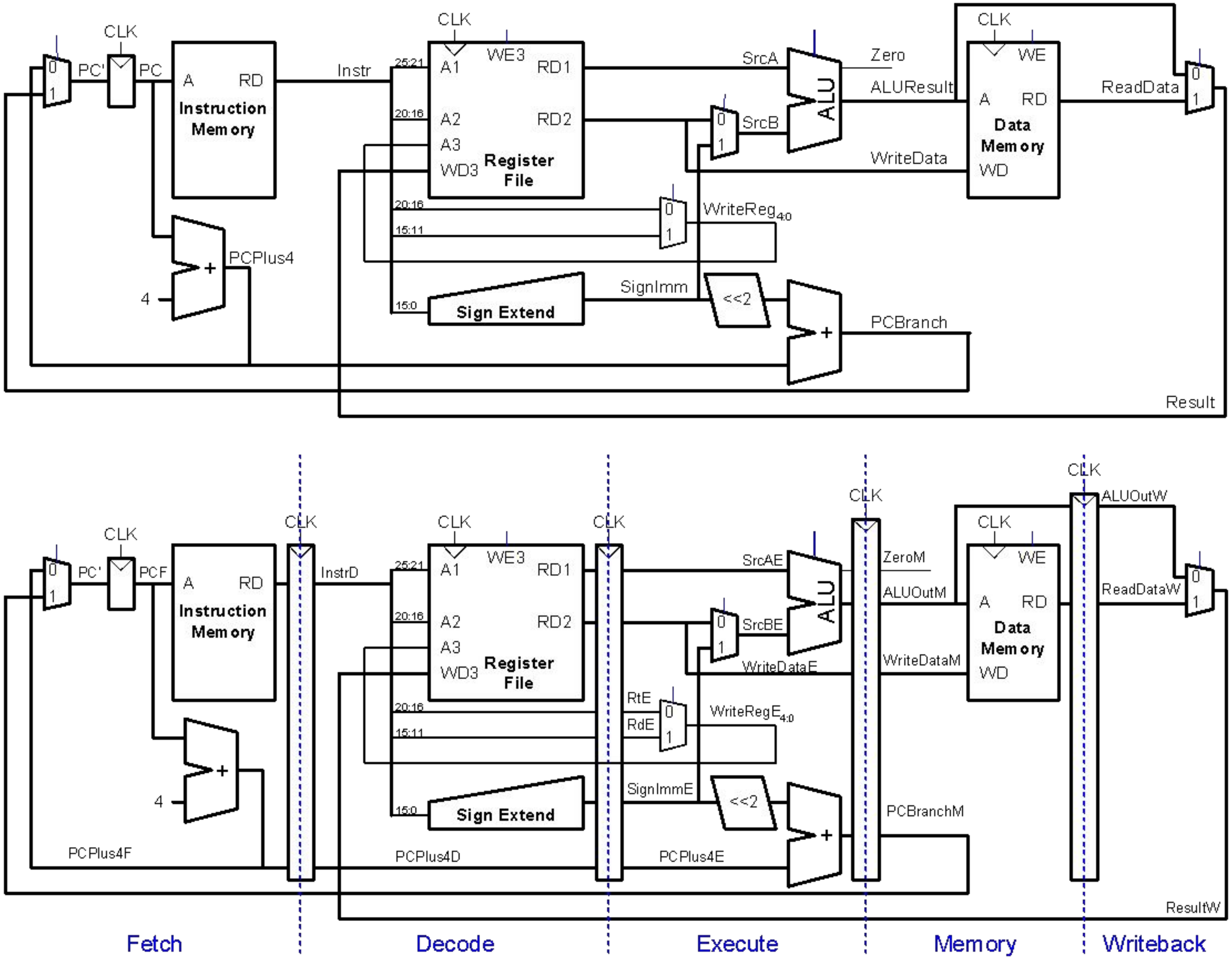
```
and $s5, $t5, $t6
```
$s5 = $t5 & $t6

```
sw  $s6, 20($s1)
```
*($s1 + 20) = $s6

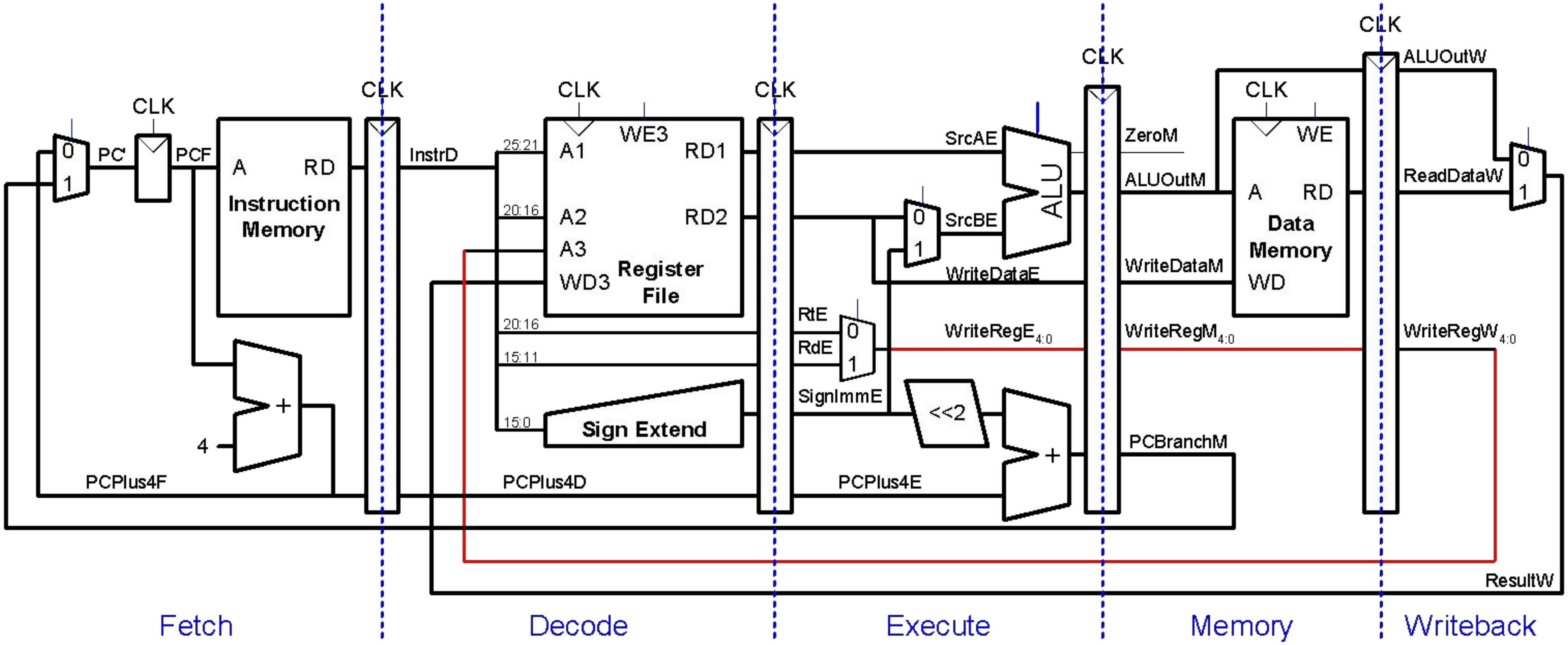
```
or  $s7, $t3, $t4
```
$s7 = $t3 | $t4



Single-Cycle & Pipelined Datapath

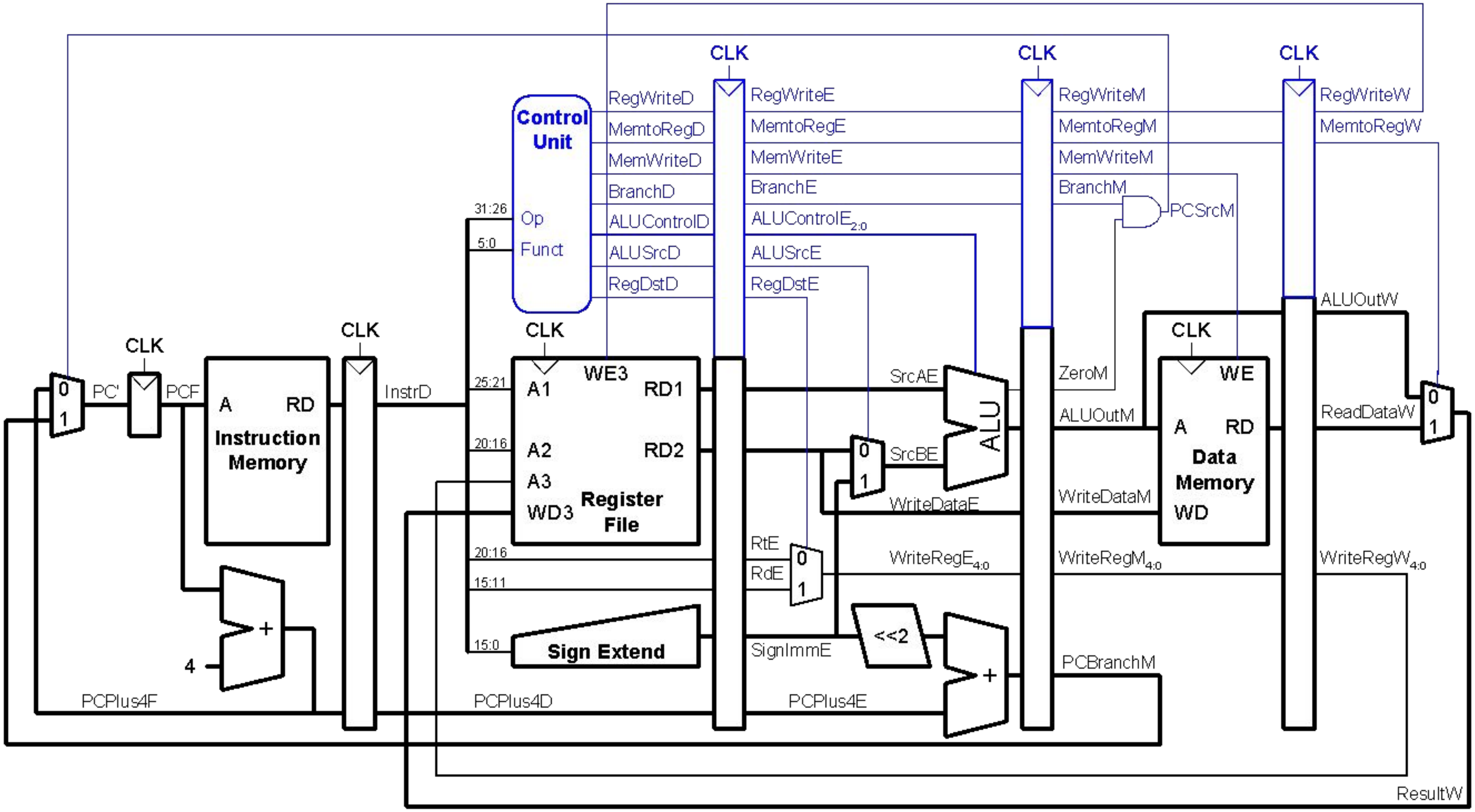


Corrected Pipelined Datapath



WriteReg* must arrive at same time as *Result

Pipelined Processor Control



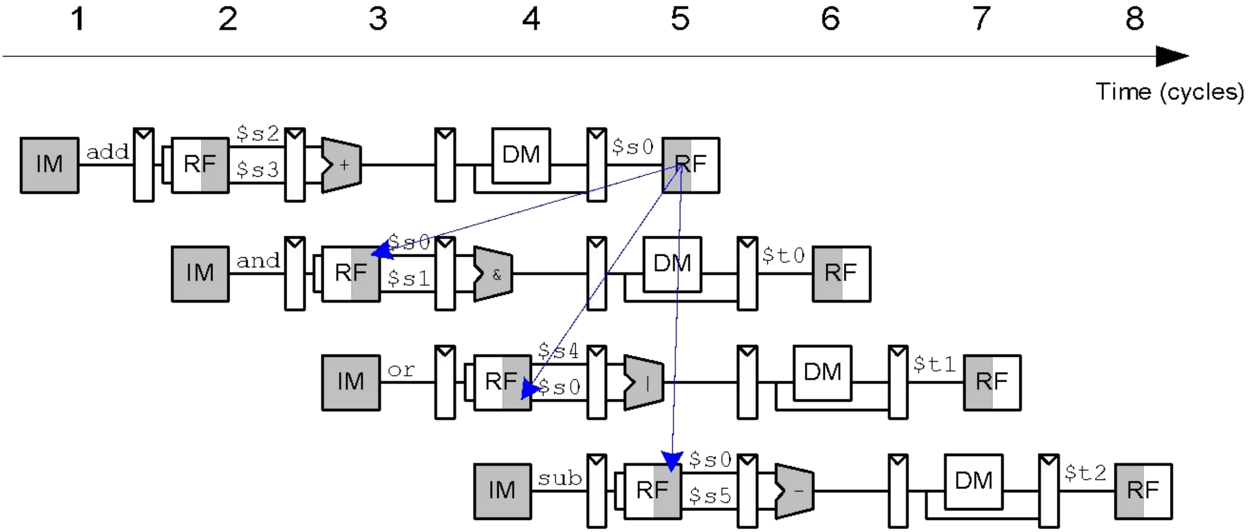
- **Same control unit as single-cycle processor**
- **Control delayed to proper pipeline stage**

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 branches)

Data Hazard

```
add $s0, $s2, $s3

and $t0, $s0, $s1

or  $t1, $s4, $s0

sub $t2, $s0, $s5
```

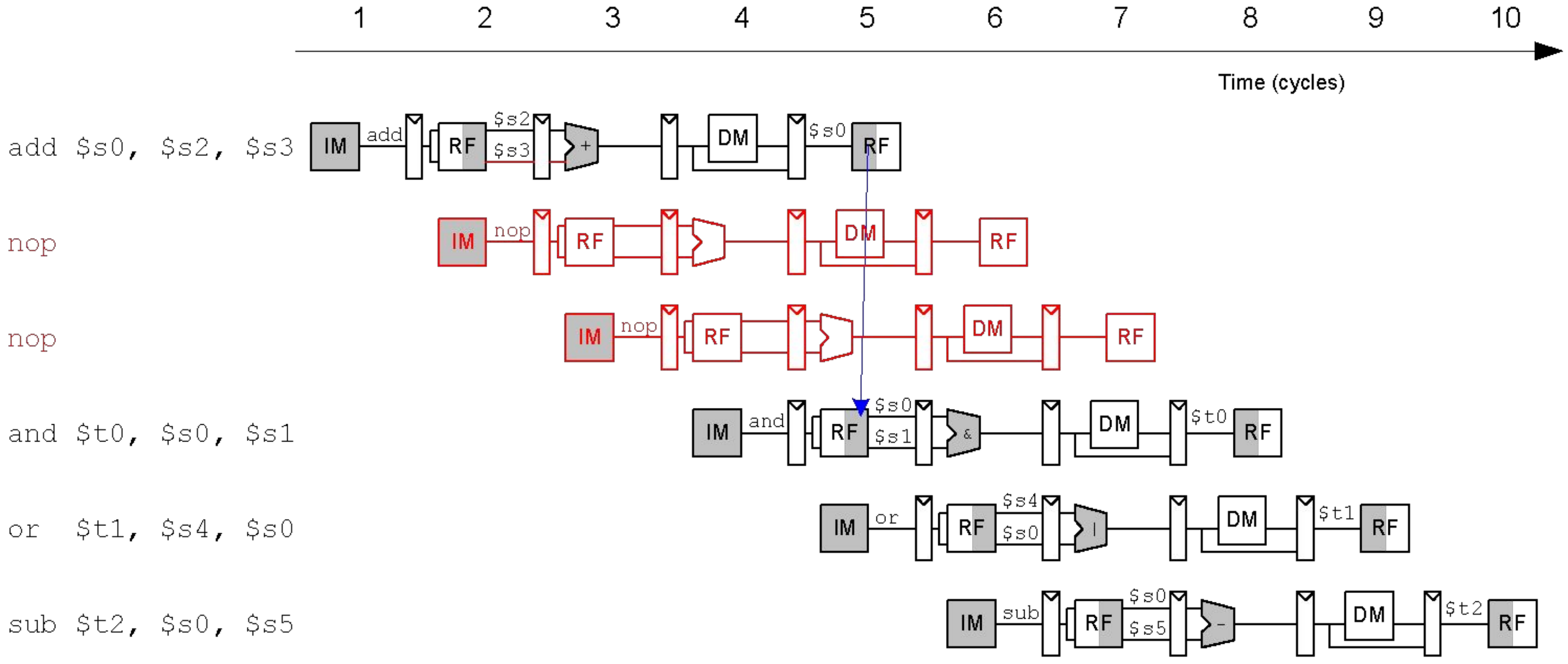


Handling Data Hazards

- 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

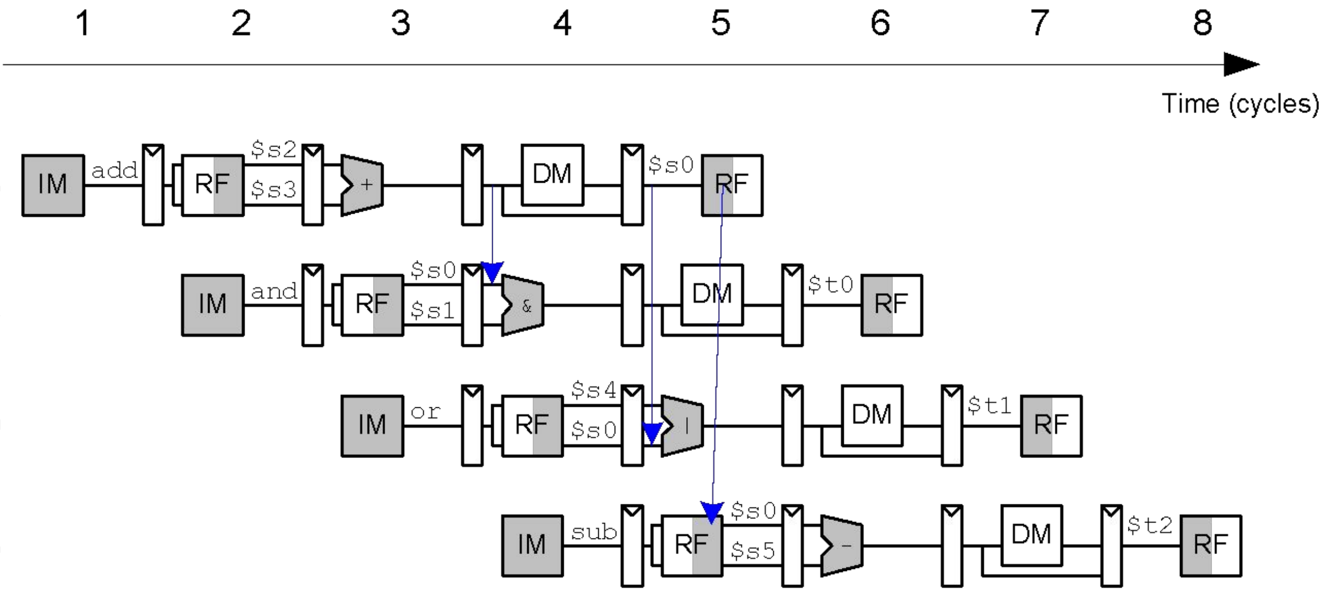
- Insert enough `nops` for result to be ready
- Or move independent useful instructions forward



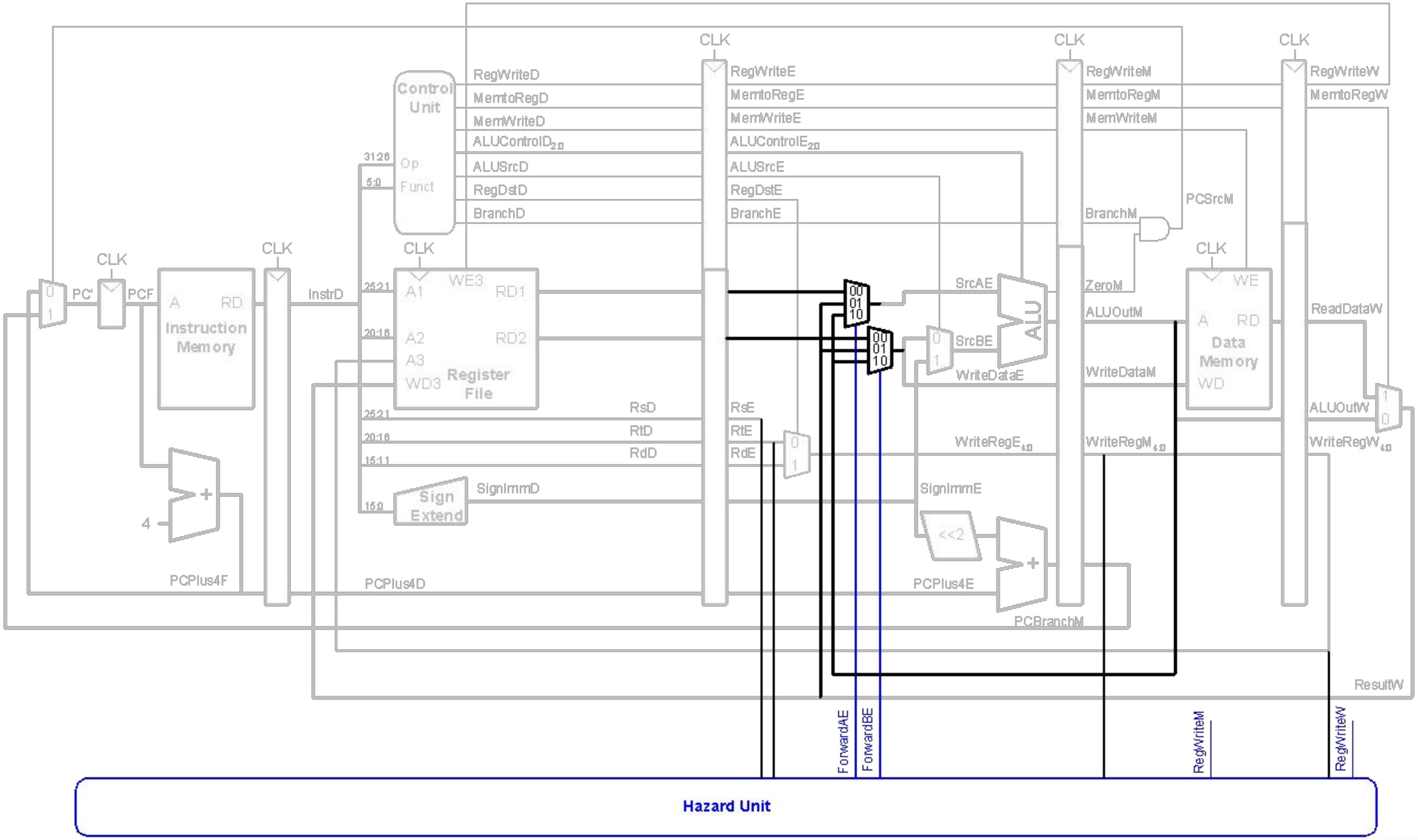
```
add $s0, $s2, $s3
nop
nop
and $t0, $s0, $s1
or  $t1, $s4, $s0
sub $t2, $s0, $s5
```

Data Forwarding

```
add $s0, $s2, $s3
and $t0, $s0, $s1
or  $t1, $s4, $s0
sub $t2, $s0, $s5
```



Data Forwarding



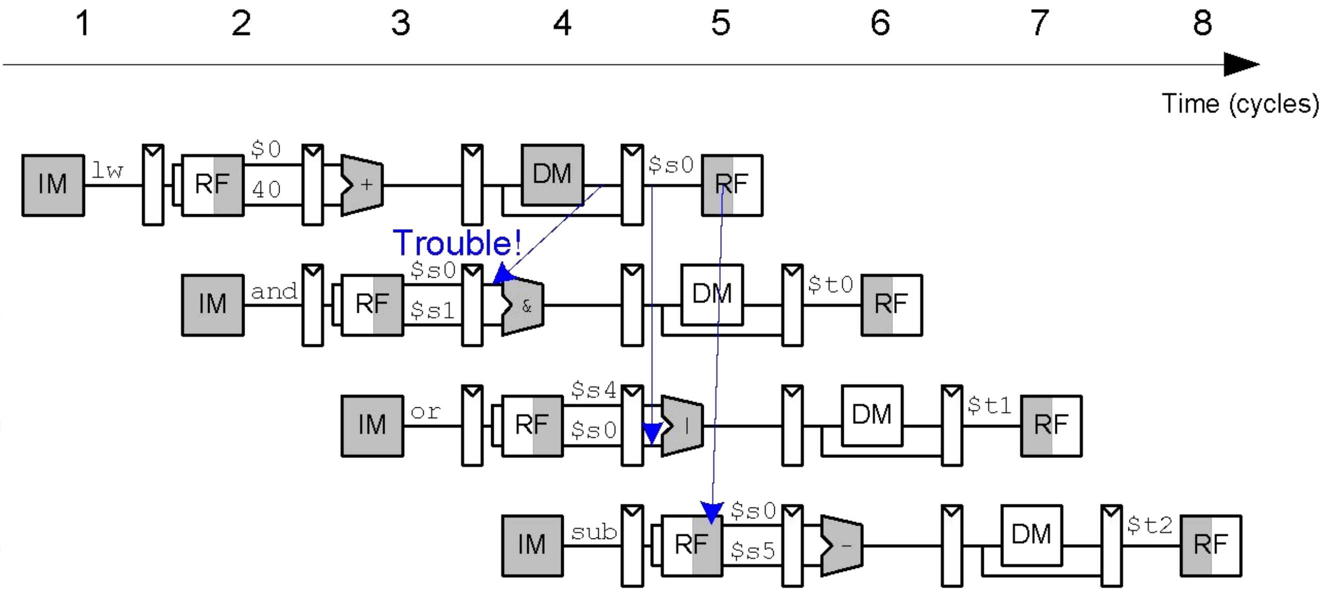
Data Forwarding

- 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
```

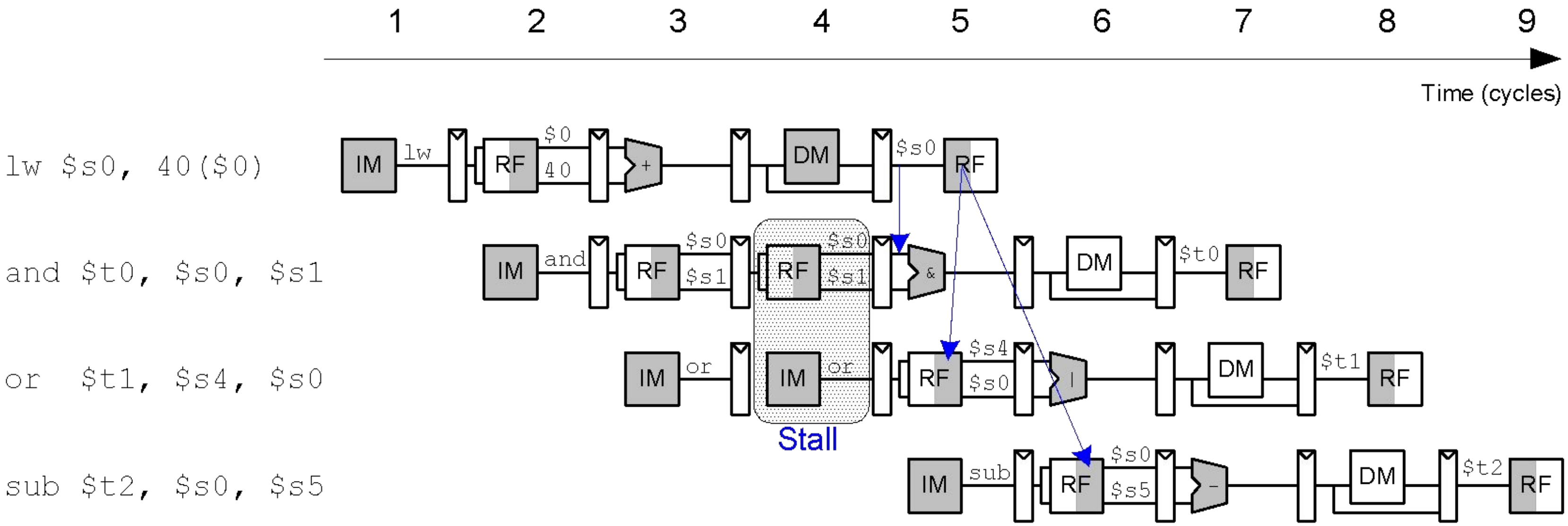
Forwarding logic for *ForwardBE* same, but replace *rsE* with *rtE*

Stalling

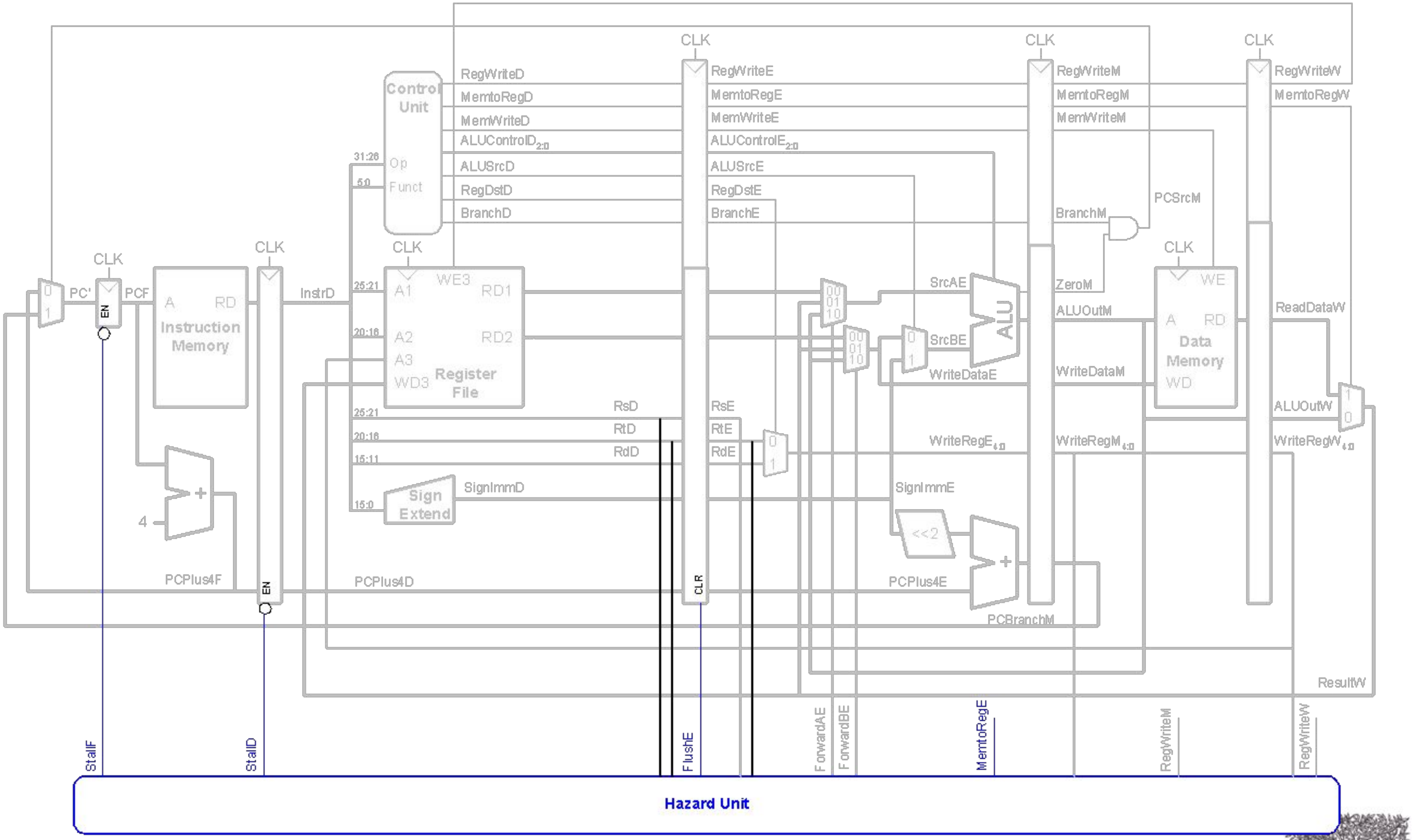


```
lw  $s0, 40($0)
and $t0, $s0, $s1
or  $t1, $s4, $s0
sub $t2, $s0, $s5
```

Stalling


1
2
3
4
5
6
7
8
9
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
IM
and
RF
$s0
$s1
RF
$s0
$s1
&
DM
$t0
RF
IM
or
IM
or
RF
$s4
$s0
|
DM
$t1
RF
Stall
IM
sub
RF
$s0
$s5
-
DM
$t2
RF

Stalling Hardware



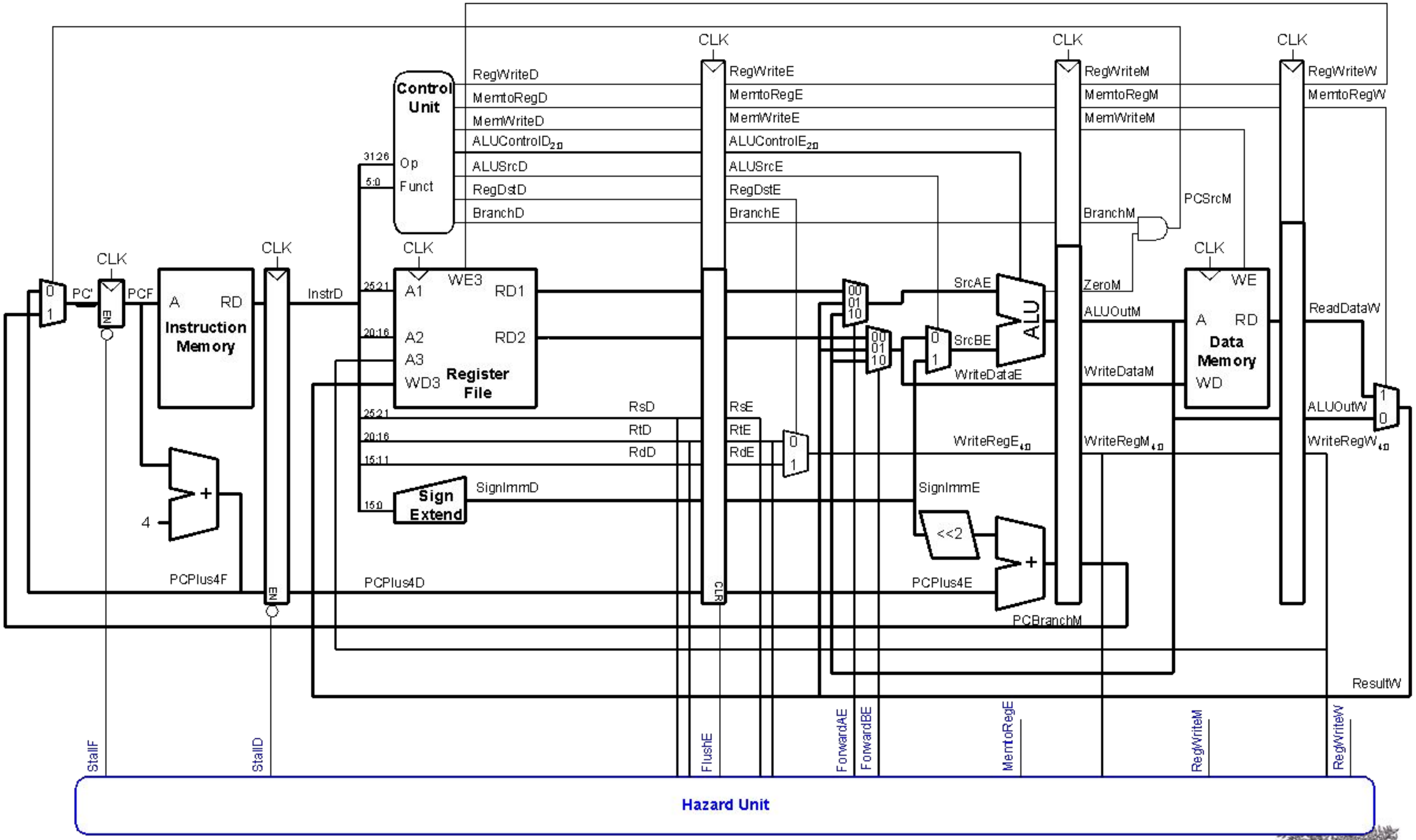
Stalling Logic

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

StallF = StallD = FlushE = lwstall
```

Control Hazards

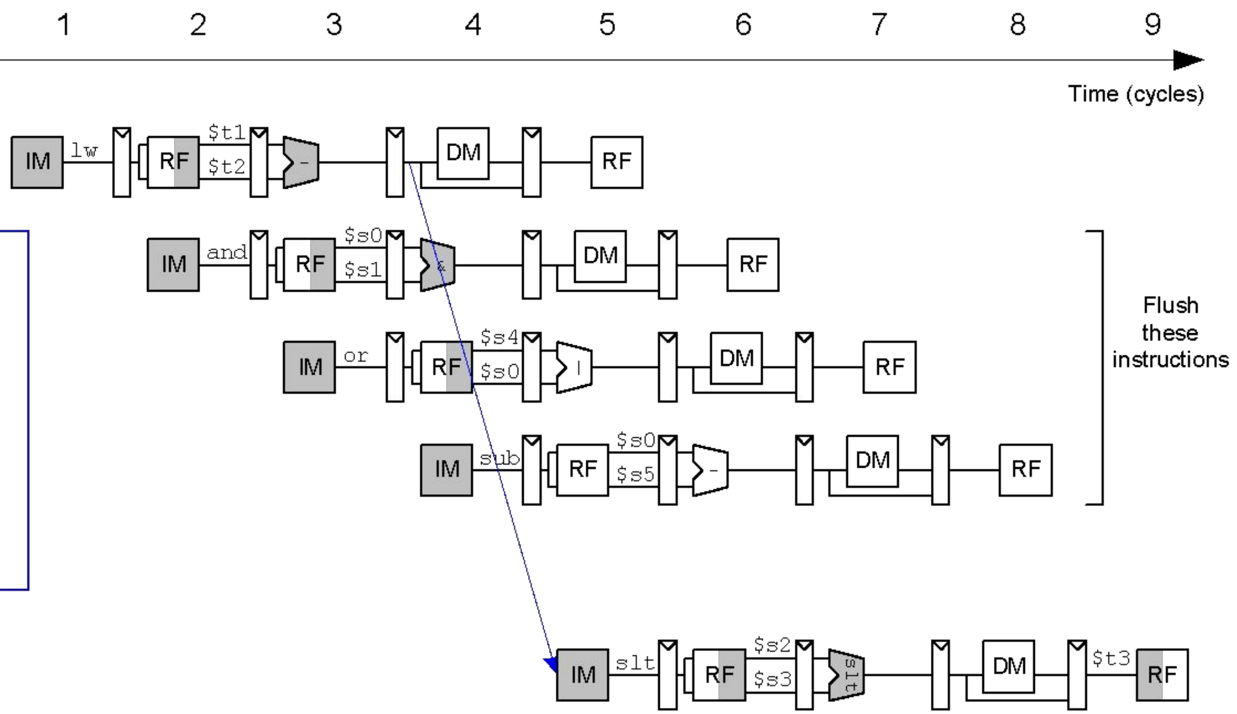
- **`beq`:**
  - branch not determined until 4th stage of pipeline
  - Instructions after branch fetched before branch occurs
  - These instructions must be flushed if branch happens
- **Branch misprediction penalty**
  - number of instruction flushed when branch is taken
  - May be reduced by determining branch earlier

Control Hazards: Original Pipeline

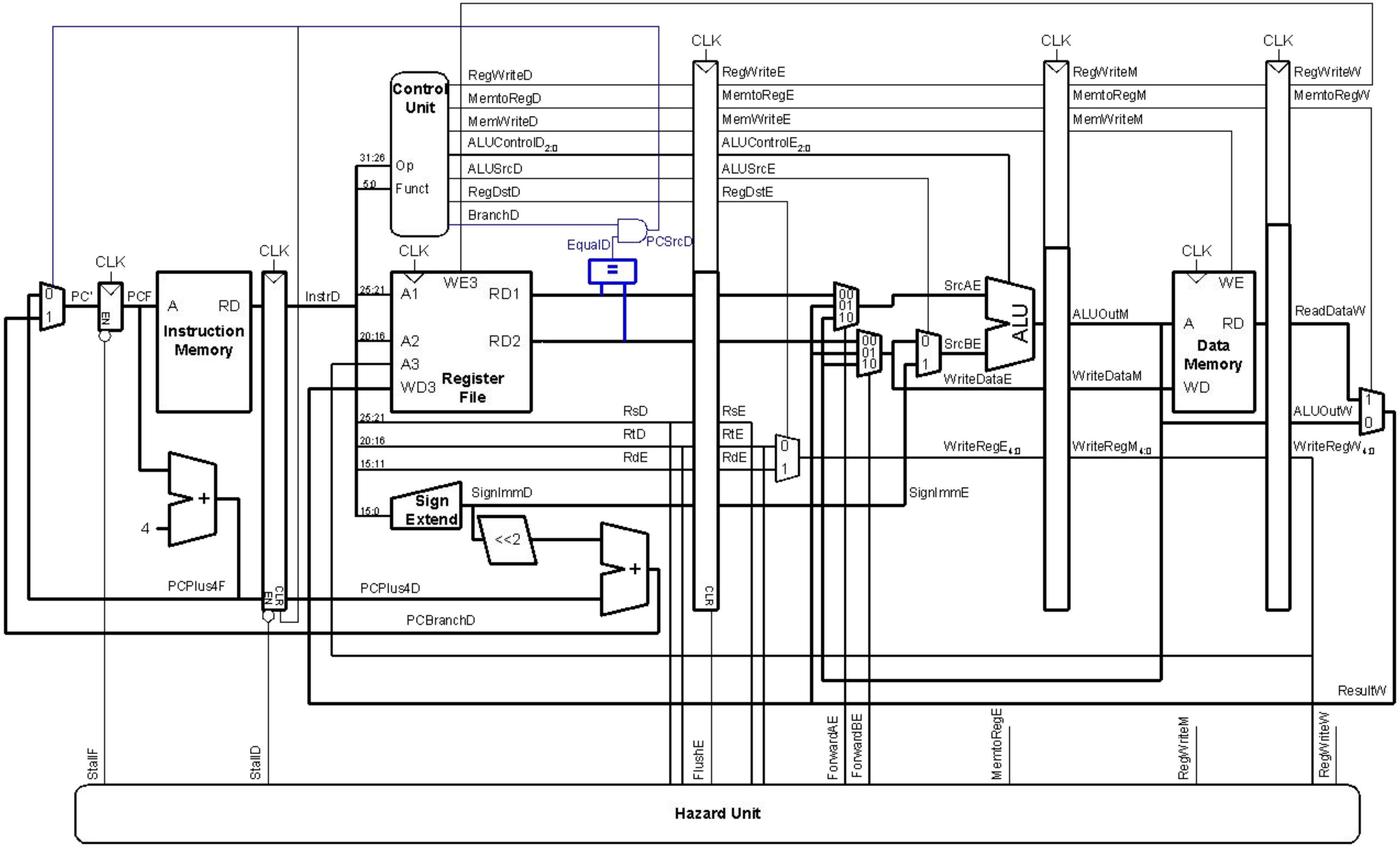


Control Hazards

```
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
```



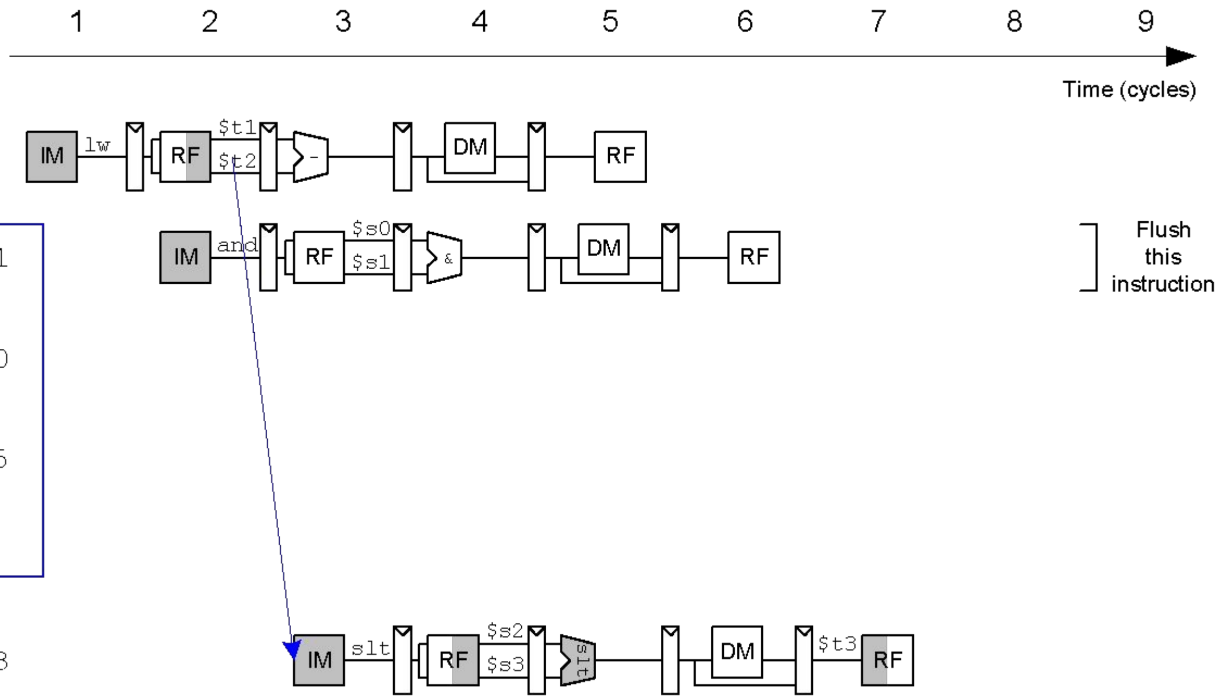
Early Branch Resolution



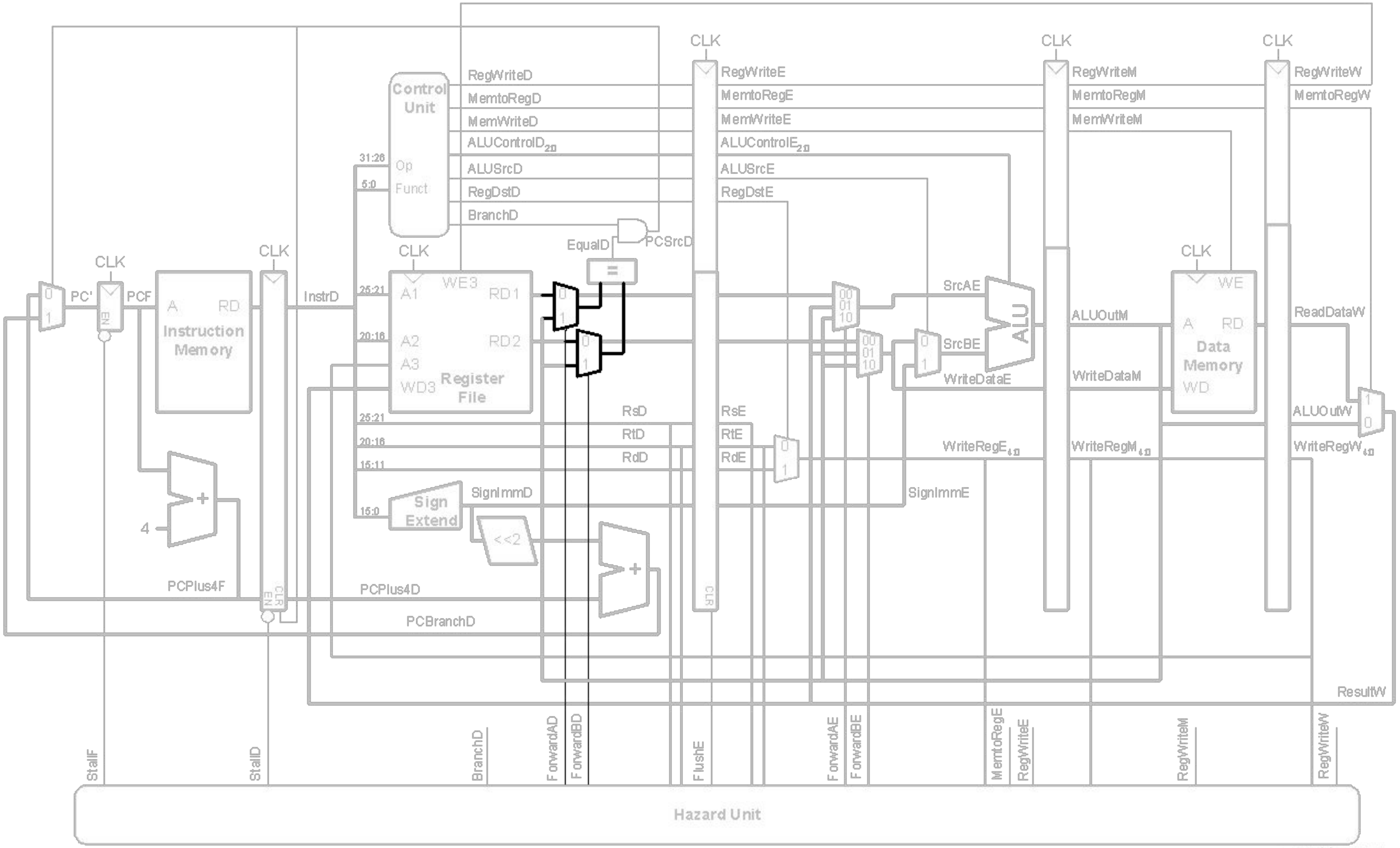
Introduced another data hazard in Decode stage

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 & Control Hazards



Control Forwarding & Stalling Logic

- **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
 [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)
  - Consider history to improve guess
- Good prediction reduces fraction of branches requiring a flush

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

- 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?**
  - Load/Branch CPI = 1 when no stalling, 2 when stalling
  - $\text{CPI}_{\text{lw}} = 1(0.6) + 2(0.4) = 1.4$
  - $\text{CPI}_{\text{beq}} = 1(0.75) + 2(0.25) = 1.25$

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:

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

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

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

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

Advanced Microarchitecture

- Deep Pipelining
- 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

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

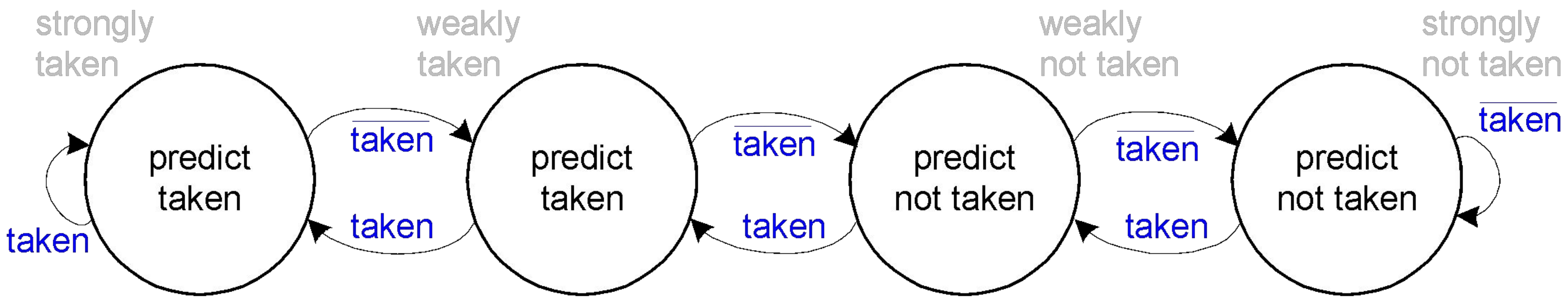
Branch Prediction Example

```
   add  $s1, $0, $0        # sum = 0
   add  $s0, $0, $0        # i   = 0
   addi $t0, $0, 10        # $t0 = 10
for:
   beq  $s0, $t0, done     # if i == 10, branch
   add  $s1, $s1, $s0      # sum = sum + i
   addi $s0, $s0, 1        # increment i
   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

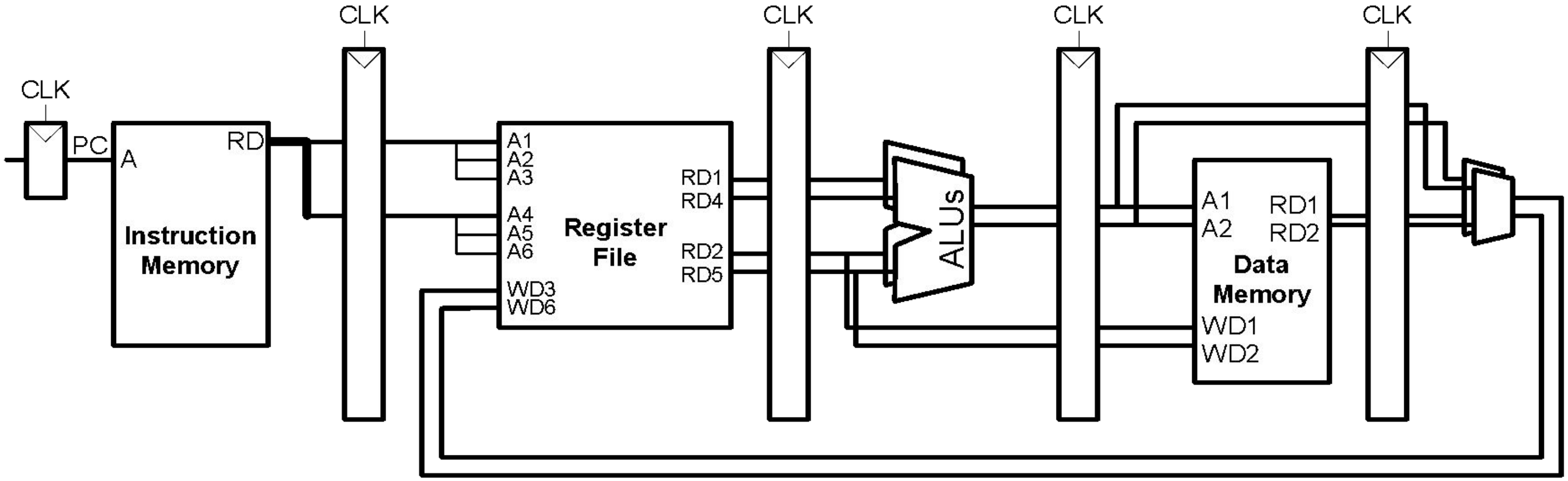
2-Bit Branch Predictor



Only mispredicts last branch of loop

Superscalar

- Multiple copies of datapath execute multiple instructions at once
- Dependencies make it tricky to issue multiple instructions at once

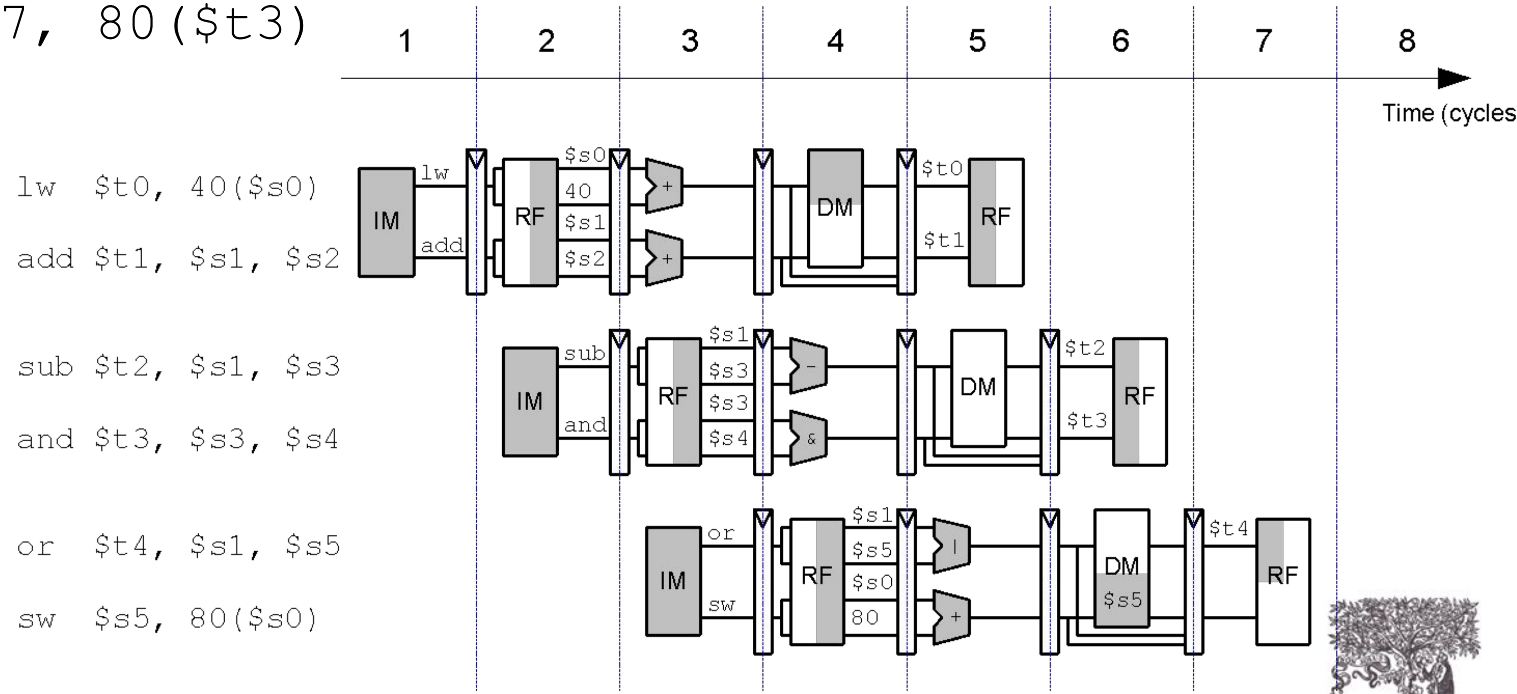


Superscalar Example

```
lw  $t0, 40($s0)
add $t1, $t0, $s1
sub $t0, $s2, $s3
and $t2, $s4, $t0
or  $t3, $s5, $s6
sw  $s7, 80($t3)
```

Ideal IPC: 2

Actual IPC: 2



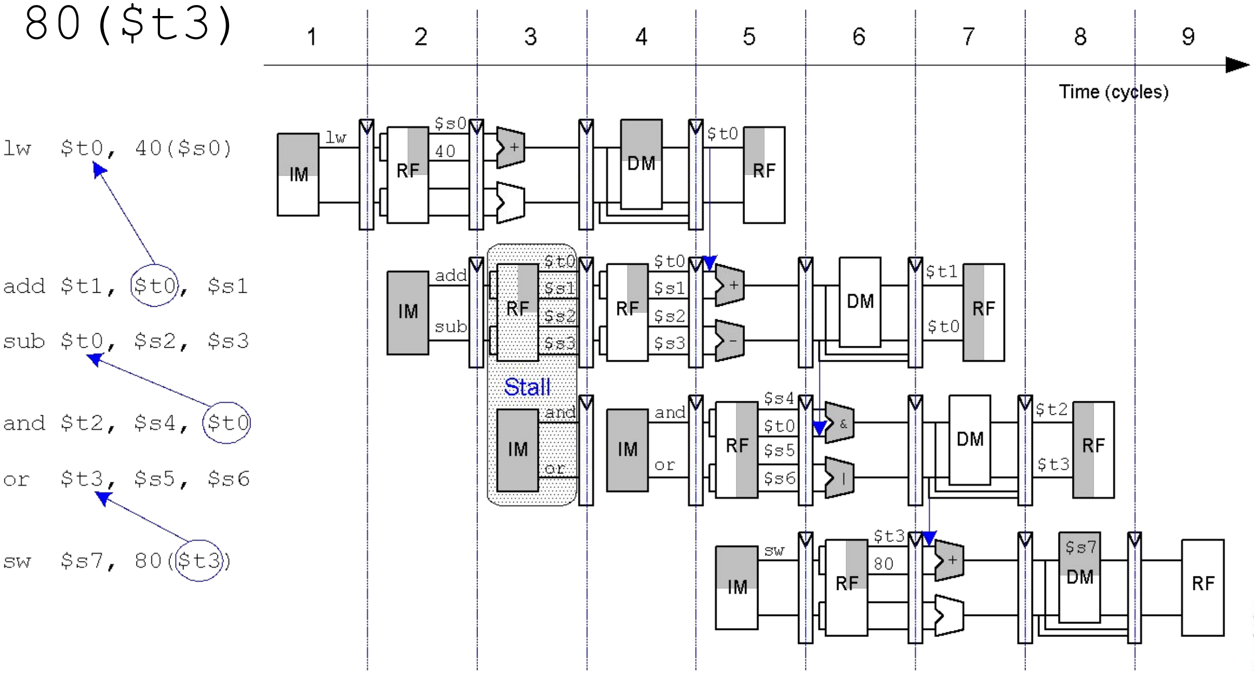
ELSEVIER

Superscalar with Dependencies

```
lw   $t0, 40($s0)
add  $t1, $t0, $s1
sub  $t0, $s2, $s3
and  $t2, $s4, $t0
or   $t3, $s5, $s6
sw   $s7, 80($t3)
```

Ideal IPC: 2

Actual IPC: 6/5 = 1.2



```
lw   $t0, 40($s0)

add  $t1, $t0, $s1
sub  $t0, $s2, $s3

and  $t2, $s4, $t0
or   $t3, $s5, $s6

sw   $s7, 80($t3)
```

Out of Order Processor

- Looks ahead across multiple instructions
- Issues as many instructions as possible at once
- Issues instructions out of order (as long as no dependencies)
- **Dependencies:**
  - **RAW** (read after write): one instruction writes, later instruction reads a register
  - **WAR** (write after read): one instruction reads, later instruction writes a register
  - **WAW** (write after write): one instruction writes, later instruction writes a register

Out of Order Processor

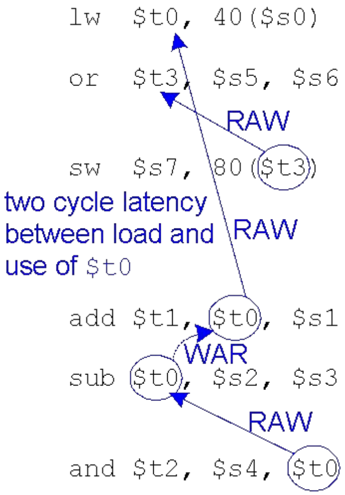
- **Instruction level parallelism (ILP):** number of instruction that can be issued simultaneously (average < 3)
- **Scoreboard:** table that keeps track of:
  - Instructions waiting to issue
  - Available functional units
  - Dependencies

Out of Order Processor Example

```
lw   $t0, 40($s0)
add  $t1, $t0, $s1
sub  $t0, $s2, $s3
and  $t2, $s4, $t0
or   $t3, $s5, $s6
sw   $s7, 80($t3)
```

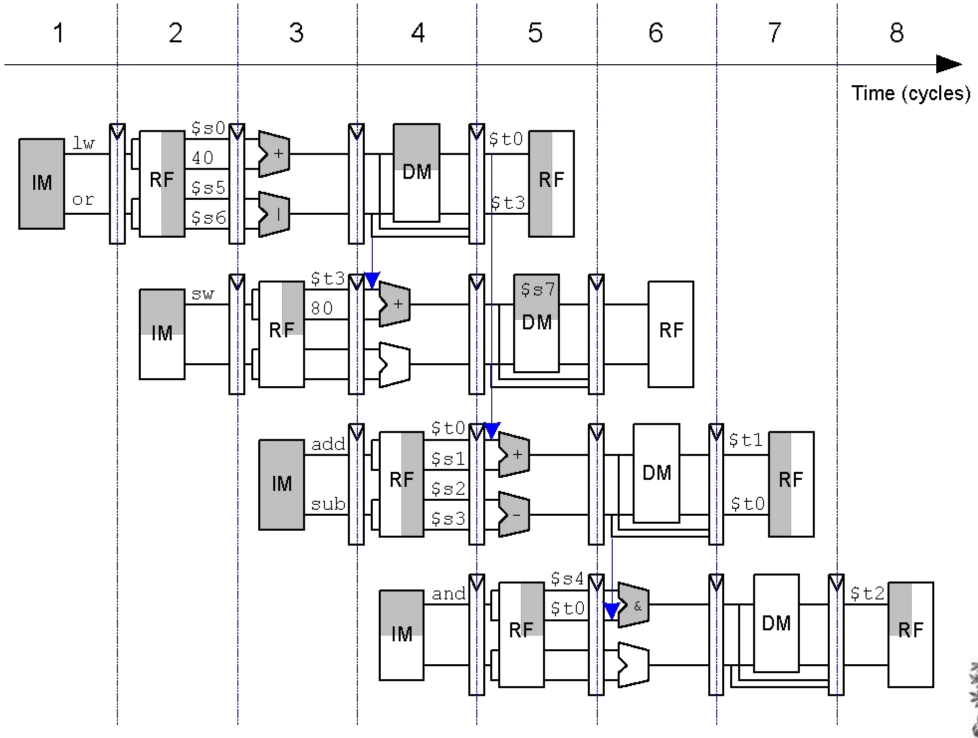
Ideal IPC: 2

Actual IPC: 6/4 = 1.5



```
lw   $t0, 40($s0)
or   $t3, $s5, $s6
sw   $s7, 80($t3)
add  $t1, $t0, $s1
sub  $t0, $s2, $s3
and  $t2, $s4, $t0
```

two cycle latency between load and use of $t0

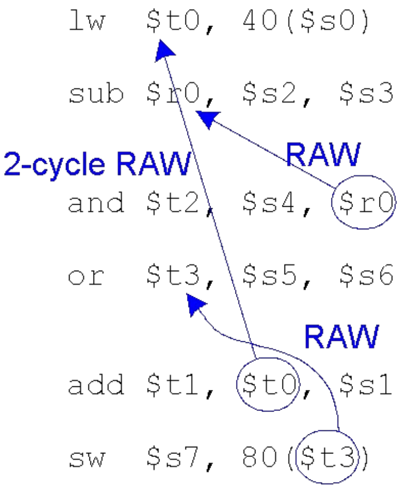


Register Renaming

```
lw   $t0, 40($s0)
add  $t1, $t0, $s1
sub  $t0, $s2, $s3
and  $t2, $s4, $t0
or   $t3, $s5, $s6
sw   $s7, 80($t3)
```

Ideal IPC: 2

Actual IPC: 6/3 = 2

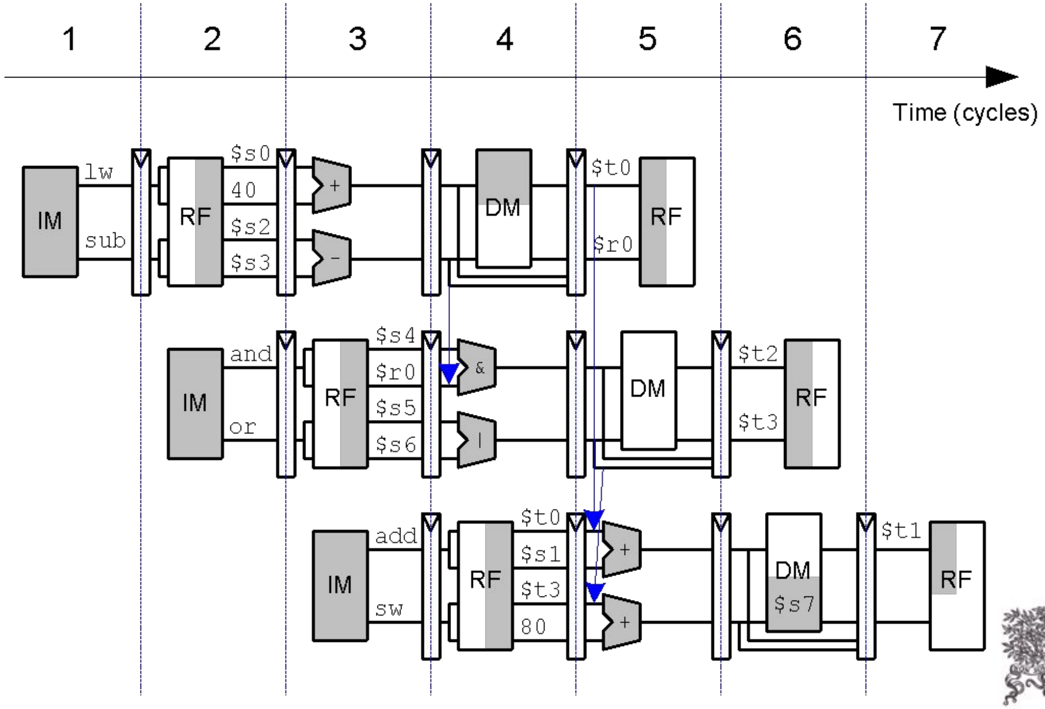


```
lw   $t0, 40($s0)
sub  $r0, $s2, $s3
and  $t2, $s4, $r0
or   $t3, $s5, $s6
add  $t1, $t0, $s1
sw   $s7, 80($t3)
```

2-cycle RAW

RAW

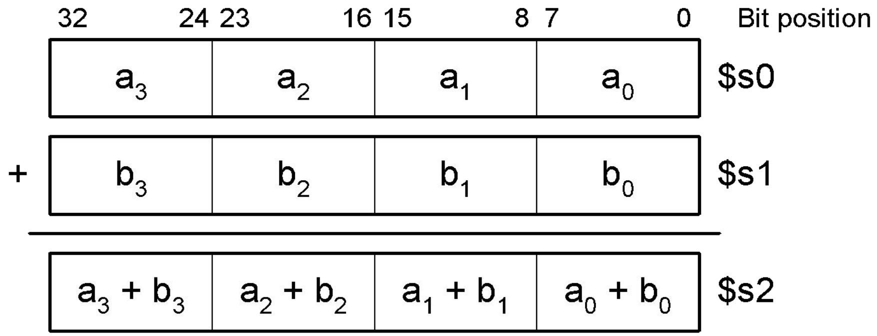
RAW



SIMD

- Single Instruction Multiple Data (SIMD)
  - Single instruction acts on multiple pieces of data at once
  - Common application: graphics
  - Perform short arithmetic operations (also called *packed arithmetic*)
- For example, add four 8-bit elements

```
padd8 $s2, $s0, $s1
```



	32 24	23 16	15 8	7 0	Bit position
	a_3	a_2	a_1	a_0	$s0
+	b_3	b_2	b_1	b_0	$s1
	$a_3 + b_3$	$a_2 + b_2$	$a_1 + b_1$	$a_0 + b_0$	$s2

Advanced Architecture Techniques

- **Multithreading**
  - Wordprocessor: thread for typing, spell checking, printing
- **Multiprocessors**
  - Multiple processors (cores) on a single chip

Threading: Definitions

- **Process:** program running on a computer
  - Multiple processes can run at once: e.g., surfing Web, playing music, writing a paper
- **Thread:** part of a program
  - Each process has multiple threads: e.g., a word processor may have threads for typing, spell checking, printing

Threads in Conventional Processor

- One thread runs at once
- When one thread stalls (for example, waiting for memory):
  - Architectural state of that thread stored
  - Architectural state of waiting thread loaded into processor and it runs
  - Called **context switching**
- Appears to user like all threads running simultaneously

Multithreading

- Multiple copies of architectural state
- Multiple threads **active** at once:
  - When one thread stalls, another runs immediately
  - If one thread can't keep all execution units busy, another thread can use them
- Does not increase instruction-level parallelism (ILP) of single thread, but increases throughput

Intel calls this "hyperthreading"

Multiprocessors

- Multiple processors (cores) with a method of communication between them
- Types:
  - **Homogeneous:** multiple cores with shared memory
  - **Heterogeneous:** separate cores for different tasks (for example, DSP and CPU in cell phone)
  - **Clusters:** each core has own memory system

Other Resources

- Patterson & Hennessy's: *Computer Architecture: A Quantitative Approach*
- Conferences:
  - www.cs.wisc.edu/~arch/www/
  - ISCA (International Symposium on Computer Architecture)
  - HPCA (International Symposium on High Performance Computer Architecture)