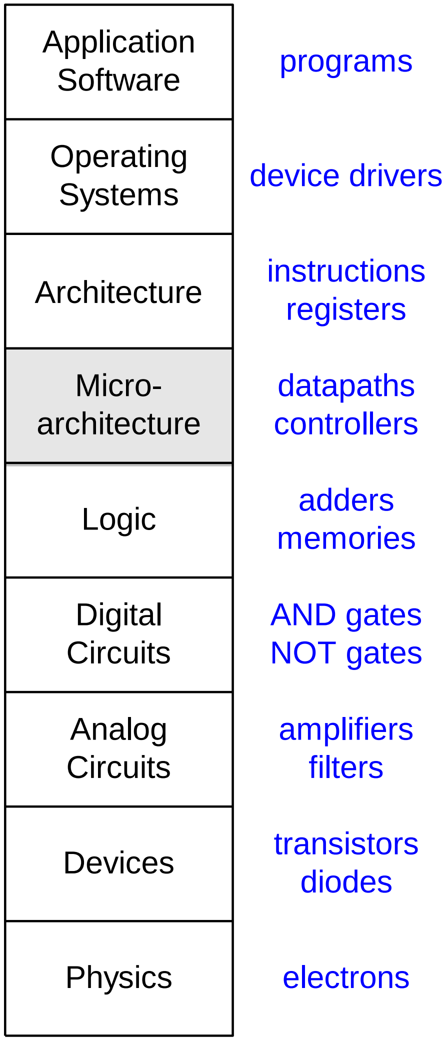
# Microarchitecture

# Topics

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

# Introduction

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



# Space and Time Metrics

- Two important metrics for any program
  - Space: How much memory does the program code and data require? (Memory footprint)
  - Time: What is the execution time for the program?
- Different design methodologies
  - CISC
  - RISC
- Memory footprint and execution time are not necessarily correlated

# What determines execution time?

- **Execution time = ($\sum$ CPI$_j$) * clock cycle time**, where $1 \leq j \leq n$
- **Execution time = n * CPI$_{Avg}$ * clock cycle time**, where n is the number of instructions (executed not static instruction count)

# Instruction Frequency

- *Static* instruction frequency refers to number of times a particular instruction occurs in compiled code.
  - Impacts memory footprint
  - If a particular instruction appears a lot in a program, can try to optimize amount of space it occupies by clever instruction encoding techniques in the instruction format.
- *Dynamic* instruction frequency refers to number of times a particular instruction is executed when program is run.
  - Impacts execution time of program
  - If dynamic frequency of an instruction is high then can try to make enhancements to datapath and control to ensure that CPI taken for its execution is minimized.

# Benchmarks

- **Benchmarks** are a set of programs that are representative of the workload for a processor.
- The key difficulty is to be sure that the benchmark program selected really are representative.
- A radical new design is hard to benchmark because there may not yet be a compiler or much code.

# SPECint2006

## 12 programs for quantifying performance of processors on integer programs

Intel Core 2 Duo
E6850 (3 GHz)

| Program name | Description | Time in seconds |
|---|---|---|
| 400.perlbench | Applications in Perl | 510 |
| 401.bzip2 | Data compression | 602 |
| 403.gcc | C Compiler | 382 |
| 429.mcf | Optimization | 328 |
| 445.gobmk | Game based on AI | 548 |
| 456.hmmer | Gene sequencing | 593 |
| 458.sjeng | Chess based on AI | 679 |
| 462.libquantum | Quantum computing | 422 |
| 464.h264ref | Video compression | 708 |
| 471.omnetpp | Discrete event simulation | 362 |
| 473.astar | Path-finding algorithm | 466 |
| 483.xalancbmk | XML processing | 302 |

# Processor Performance

- Program execution time

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

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

# Increasing the Processor Performance

- **Execution time = n * $\mathbf{CPI_{Avg}}$ * clock cycle time**
- Reduction in the number of executed instructions
- Datapath organization leading to lower CPI
- Increasing clock speed

# Speedup

- Assume a base case execution time of 10 sec.
- Assume an improved case execution time of 5 sec.
- Percent improvement = (base-new)/base
- Percent improvement = (10-5)/5 = 100%
- Speedup = base/new
- Speedup = 10/5 = 2
- Speedup is preferred by advertising copy writers

# Recall: What is Computer Architecture?

- **Computer Architecture:**

The science and art of designing, selecting, and interconnecting hardware components and designing the hardware/software interface to create a computing system that meets functional, performance, energy consumption, cost, and other specific goals.

- **Traditional definition:**

The term architecture is used here to describe the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behavior as distinct fror
dataflow and controls, the logic design,
implementation." Gene Amdahl, IBM
1964

Dr. Amdahl holding a 100gate LSI air-cooled chip. On his desk is a circuit board with the chips on it. This circuit board was for an Amdahl 470 V/6 (photograph dated March 1973).

# Amdahl’s Law

**Amdahl's law**, named after computer architect Gene Amdahl, is used to find the maximum expected improvement to an overall system when only part of the system is improved.

Amdhal's law can be interpreted more technically, but in simplest terms it means that it is the algorithm that decides the speedup not the number of processors.

Wikipedia

# Amdahl's Law

- f: Parallelizable fraction of a program
- P: Number of processors

$$\text{Speedup} = \frac{1}{1 - f + \frac{f}{P}}$$

- Maximum speedup limited by serial portion:

  Serial bottleneck

# Increasing the Throughput of the Processor

- Don’t focus on trying to speedup individual instructions
- Instead focus on throughput i.e. number of instructions executed per unit time

# Microarchitecture

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

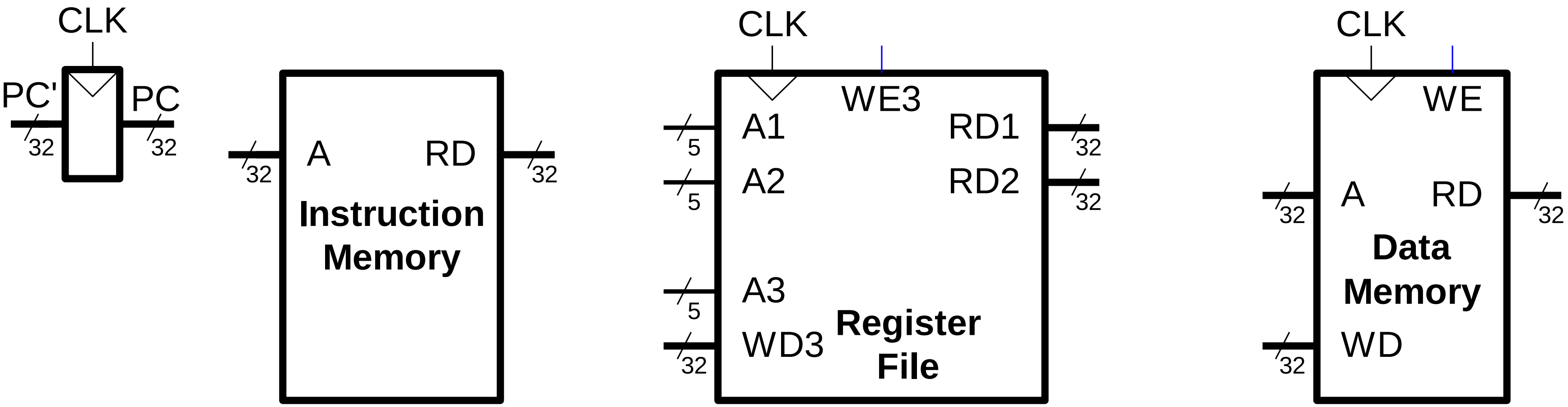
# MIPS Processor

- We consider a subset of MIPS instructions:
  - R-type instructions: `and`, `or`, `add`, `sub`, `slt`
  - Memory instructions: `lw`, `sw`
  - Branch instructions: `beq`
- Later consider adding `addi` and `j`

# Architectural State

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

# MIPS State Elements

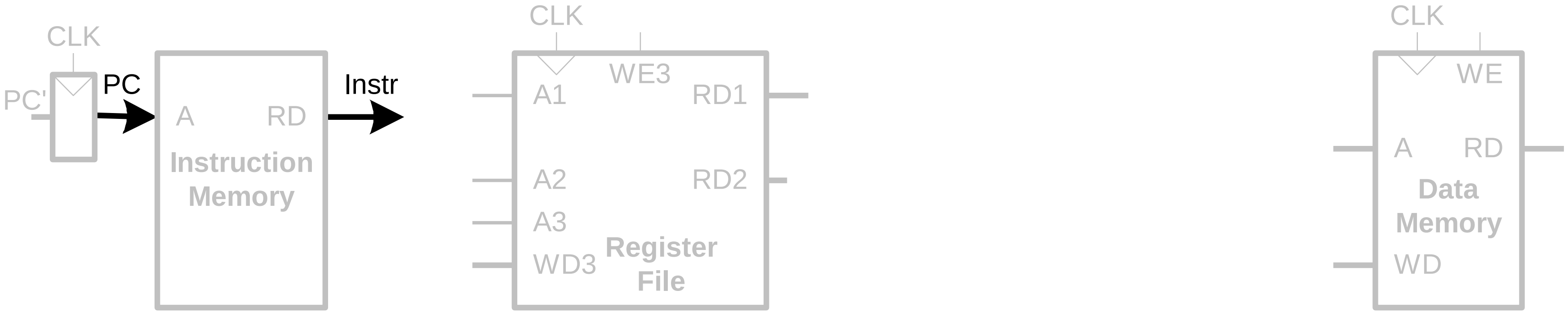


# Single-Cycle MIPS Processor

- Datapath
- Control

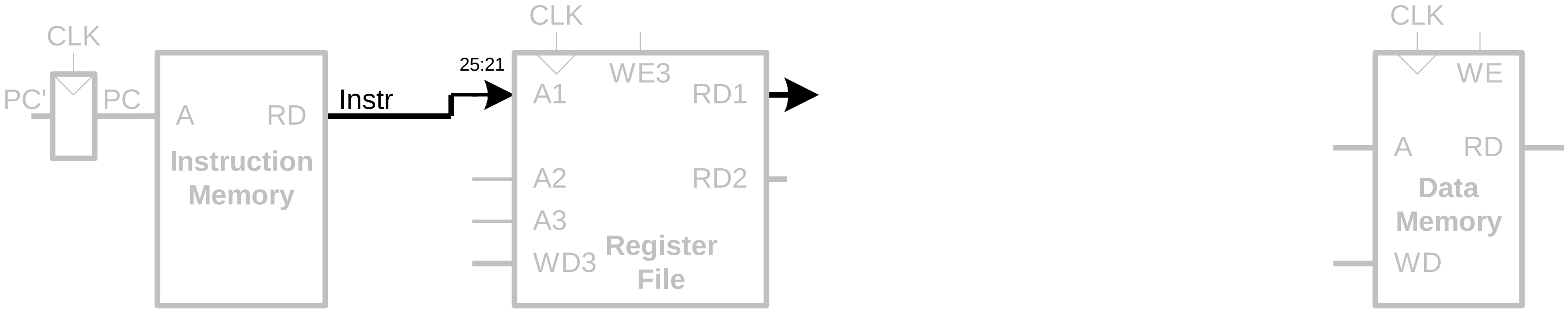
# Single-Cycle Datapath: `lw` fetch

- First consider executing `lw`
- **STEP 1:** Fetch instruction



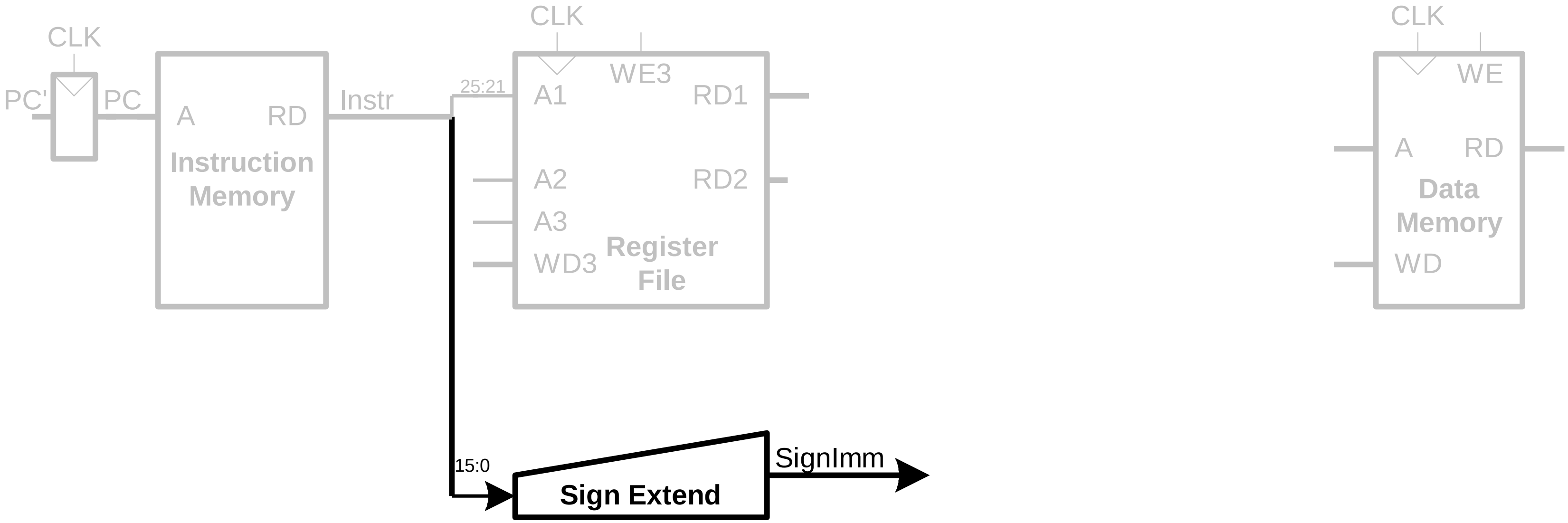
# Single-Cycle Datapath: `lw` register read

- **STEP 2:** Read source operands from register file



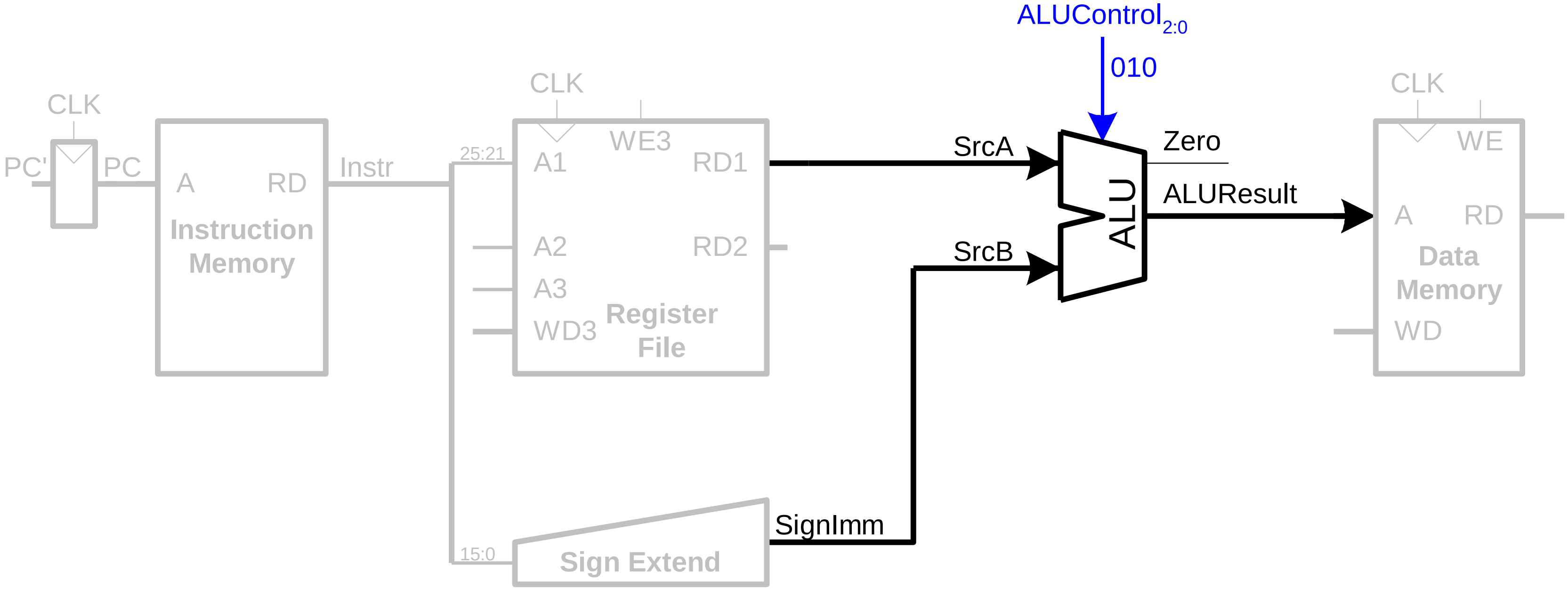
# Single-Cycle Datapath: `lw` immediate

- **STEP 3:** Sign-extend the immediate



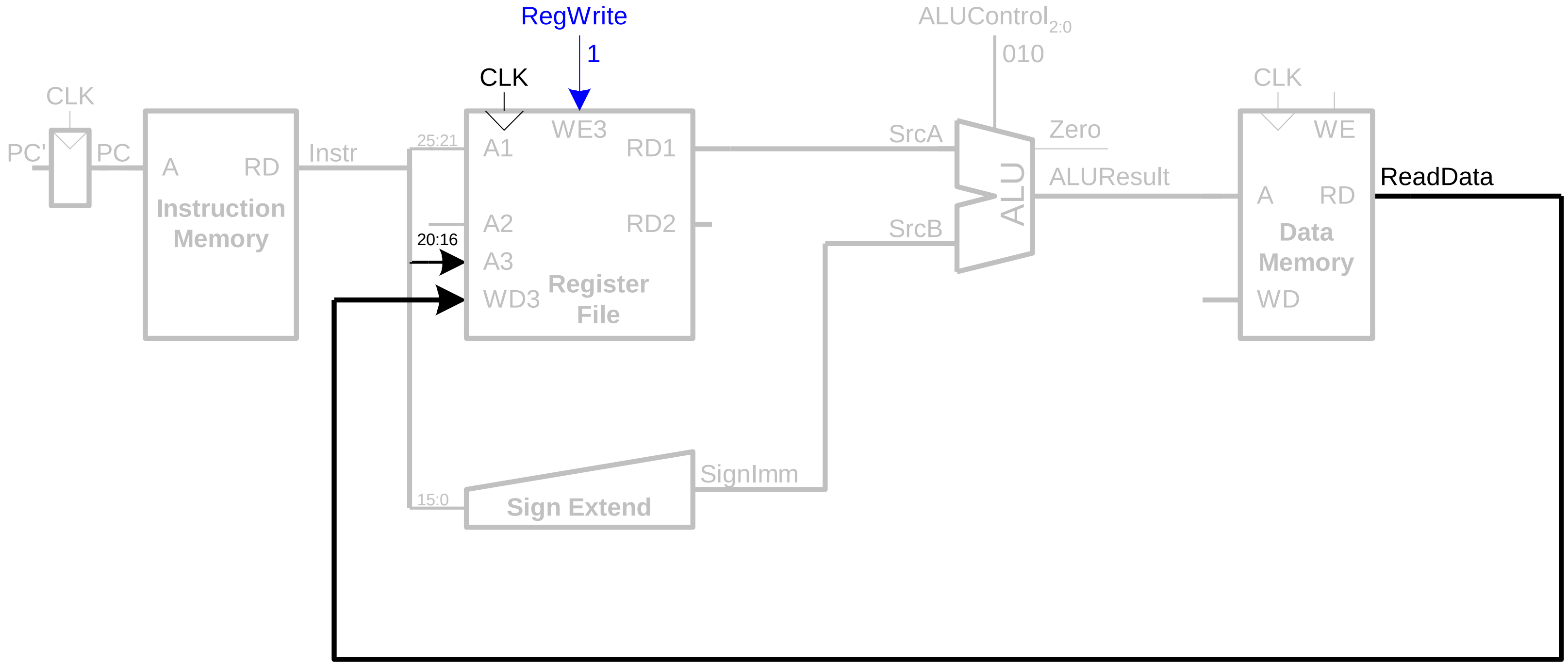
# Single-Cycle Datapath: `lw` address

- **STEP 4:** Compute the memory address



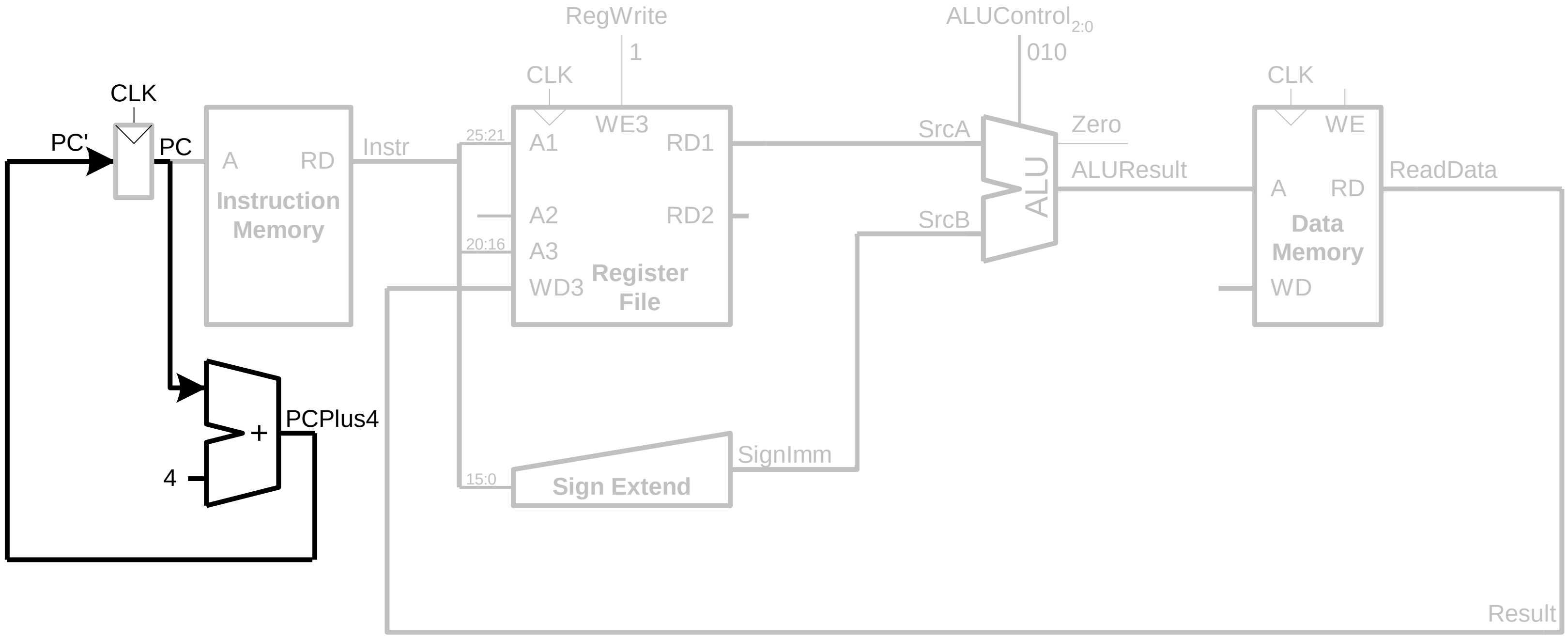
# Single-Cycle Datapath: `lw` memory read

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



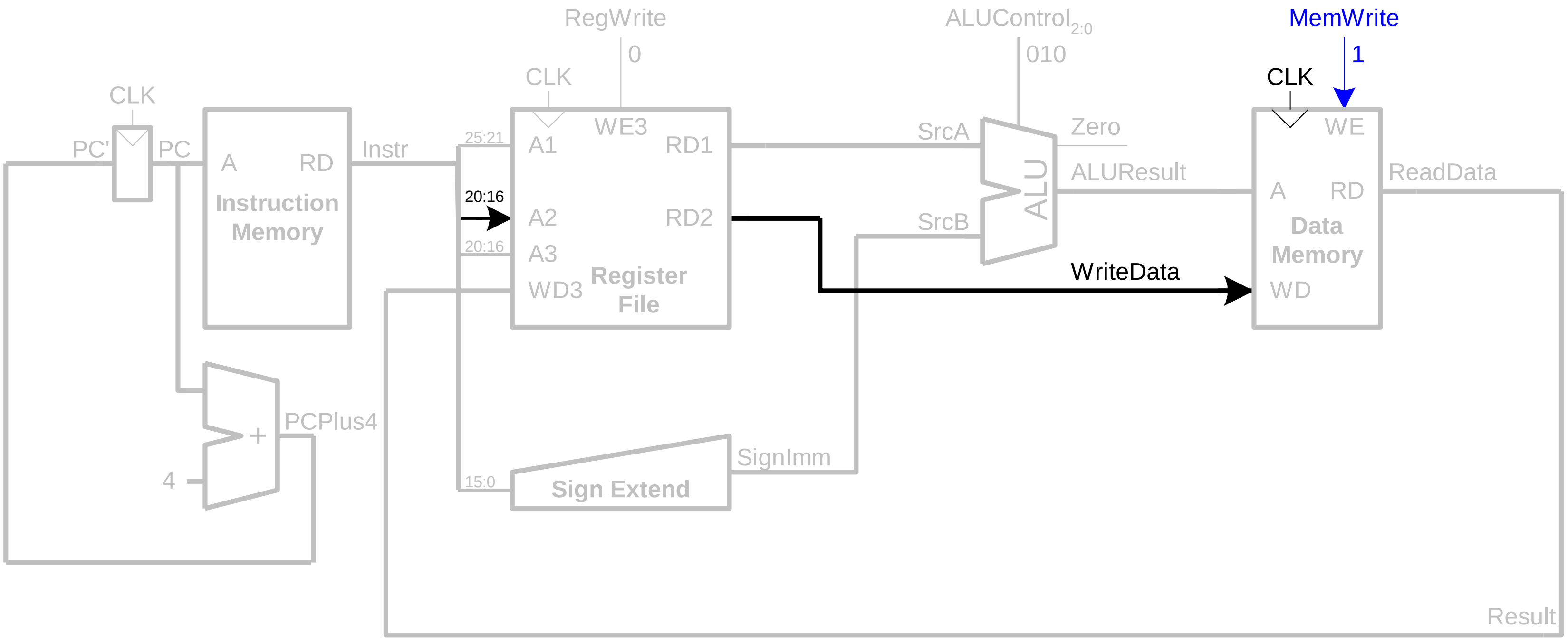
# Single-Cycle Datapath: `lw` PC increment

- **STEP 6:** Determine the address of the next instruction



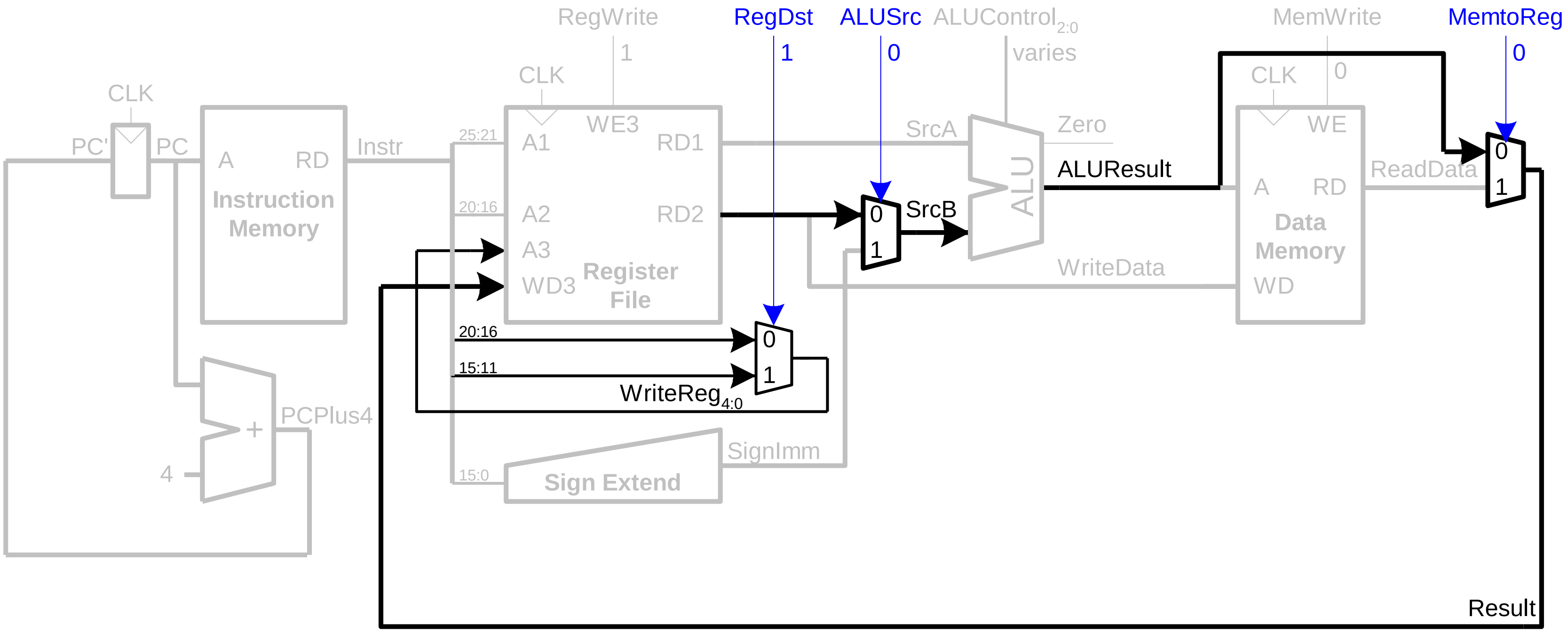
# Single-Cycle Datapath: `sw`

- Write data in `rt` to memory



# Single-Cycle Datapath: R-type instructions

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



## R-Type

| op | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

## I-Type

| op | rs | rt | imm |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 16 bits |

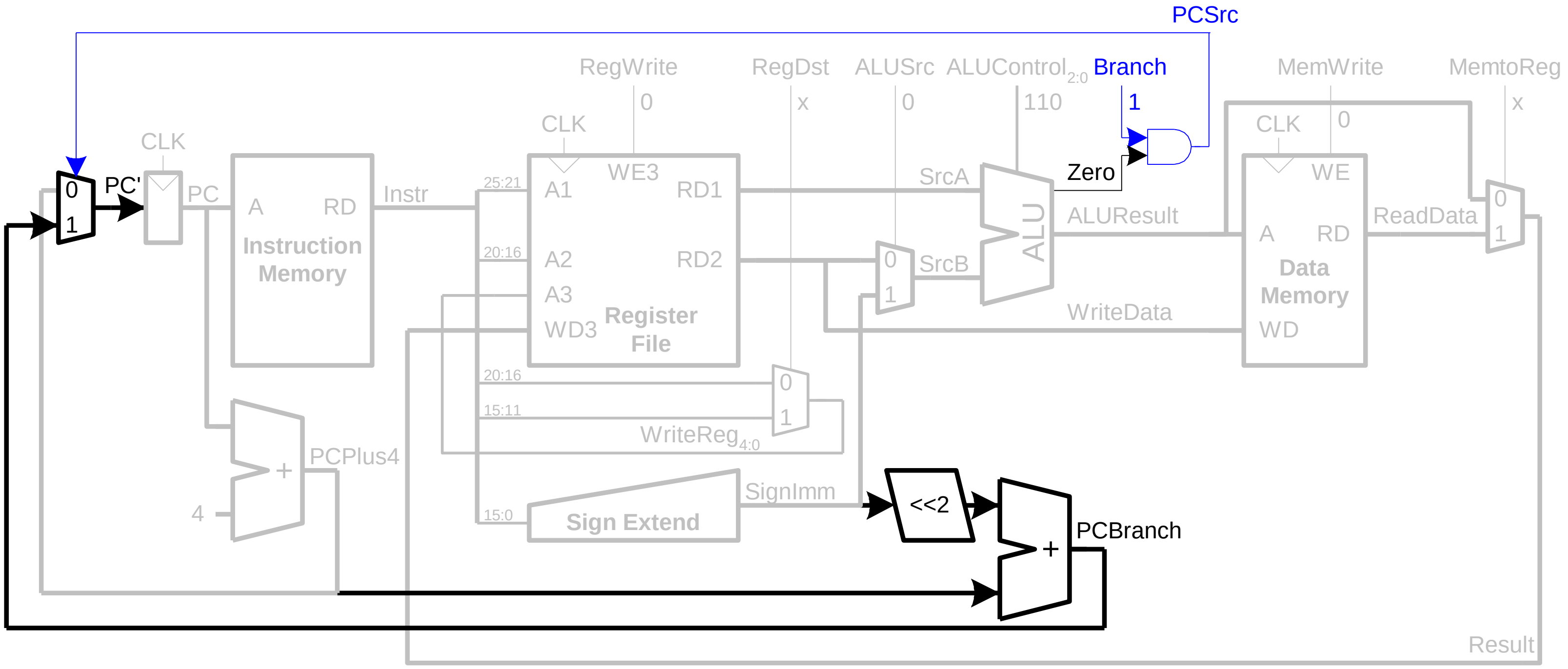
## J-Type

| op | addr |
|---|---|
| 6 bits | 26 bits |

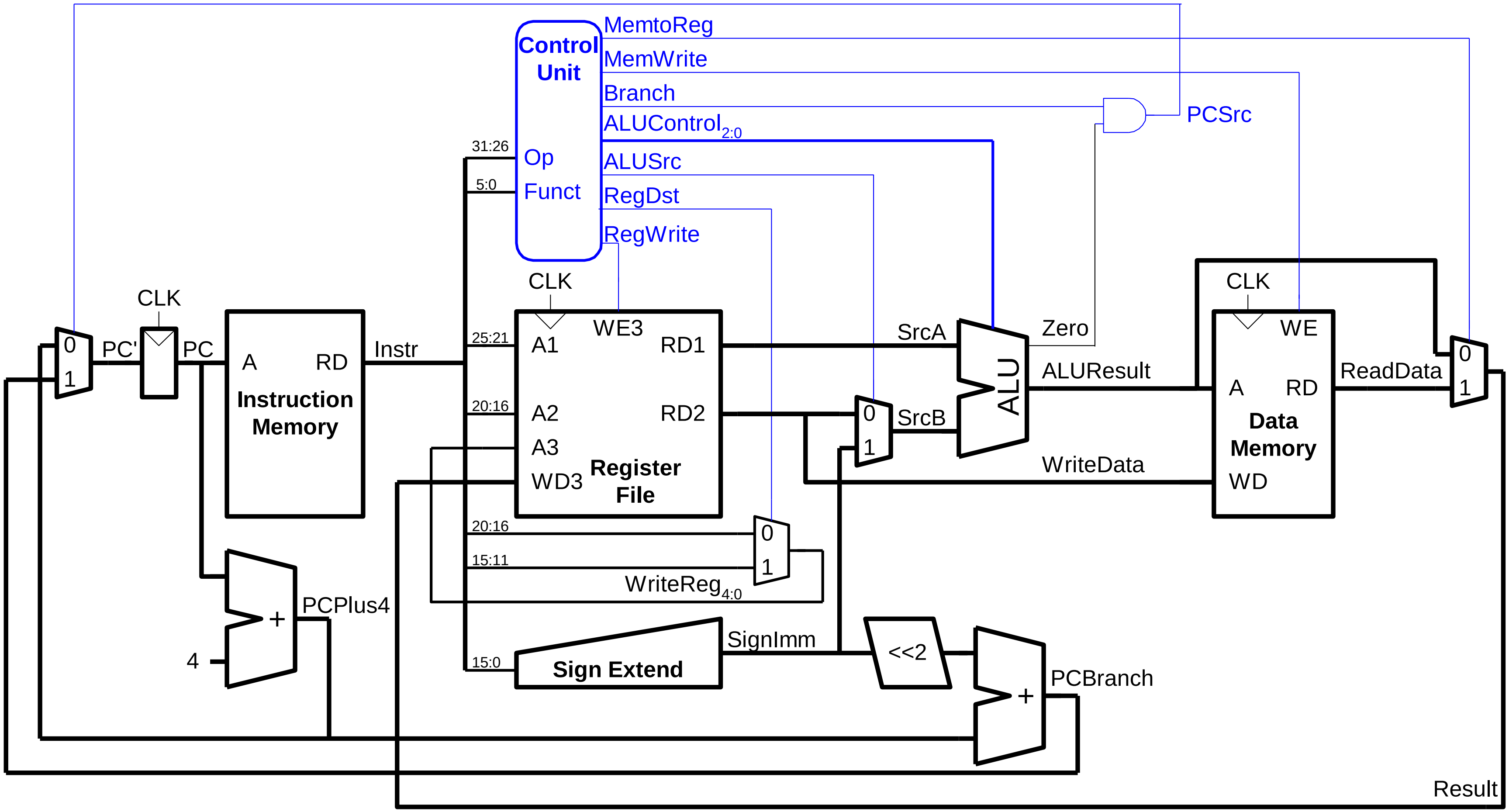
# Single-Cycle Datapath: `beq`

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

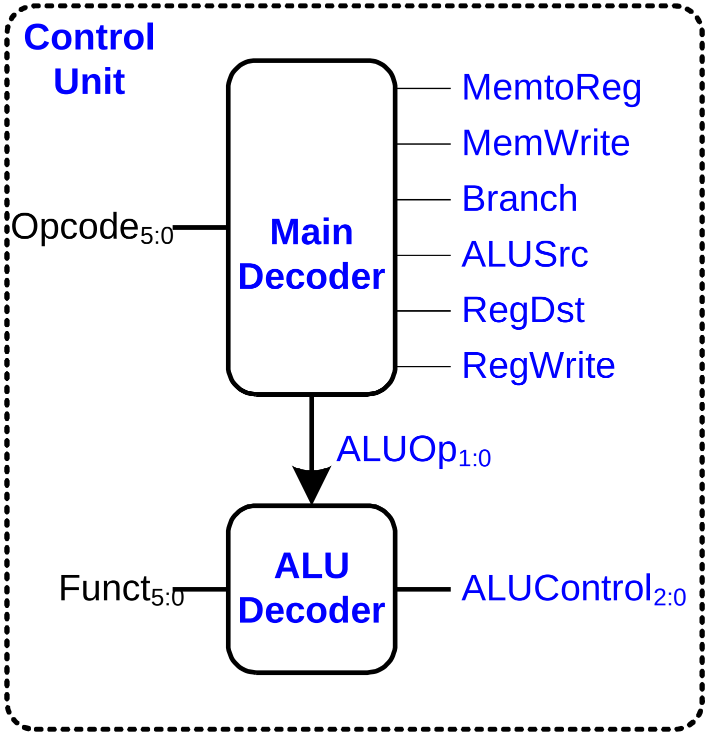
  BTA = (sign-extended immediate << 2) + (PC+4)



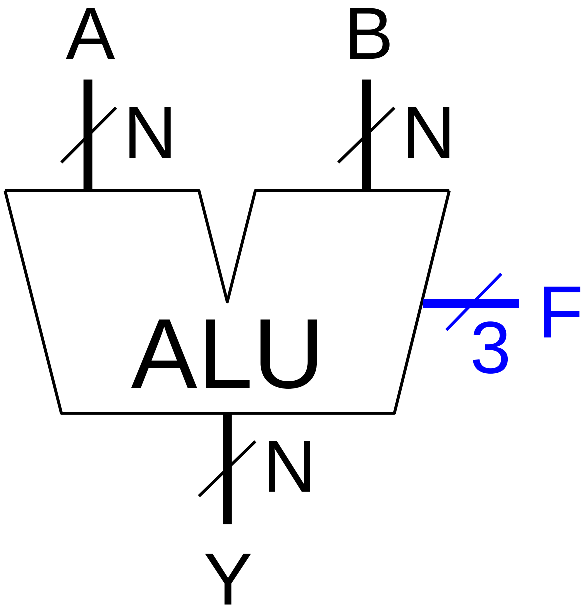
# Complete Single-Cycle Processor



# Control Unit

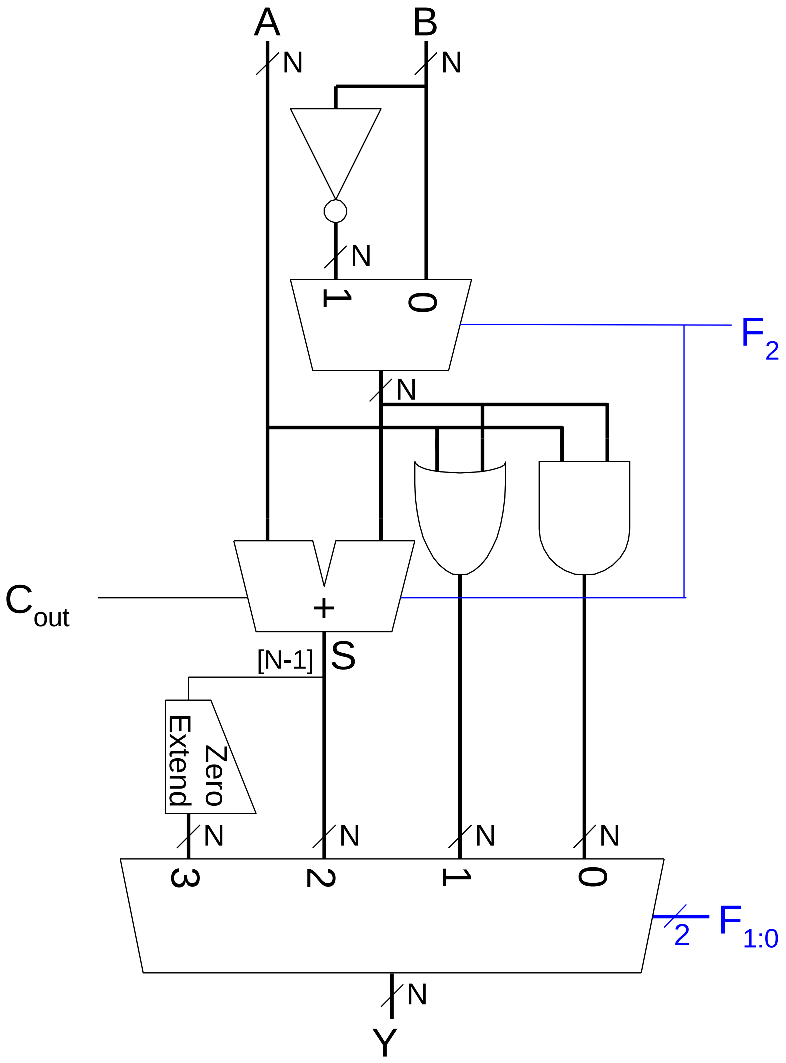


# Review: ALU



| $F_{2:0}$ | Function |
|---|---|
| 000 | A & B |
| 001 | A \| B |
| 010 | A + B |
| 011 | not used |
| 100 | A & ~B |
| 101 | A \| ~B |
| 110 | A - B |
| 111 | SLT |

# Review: ALU



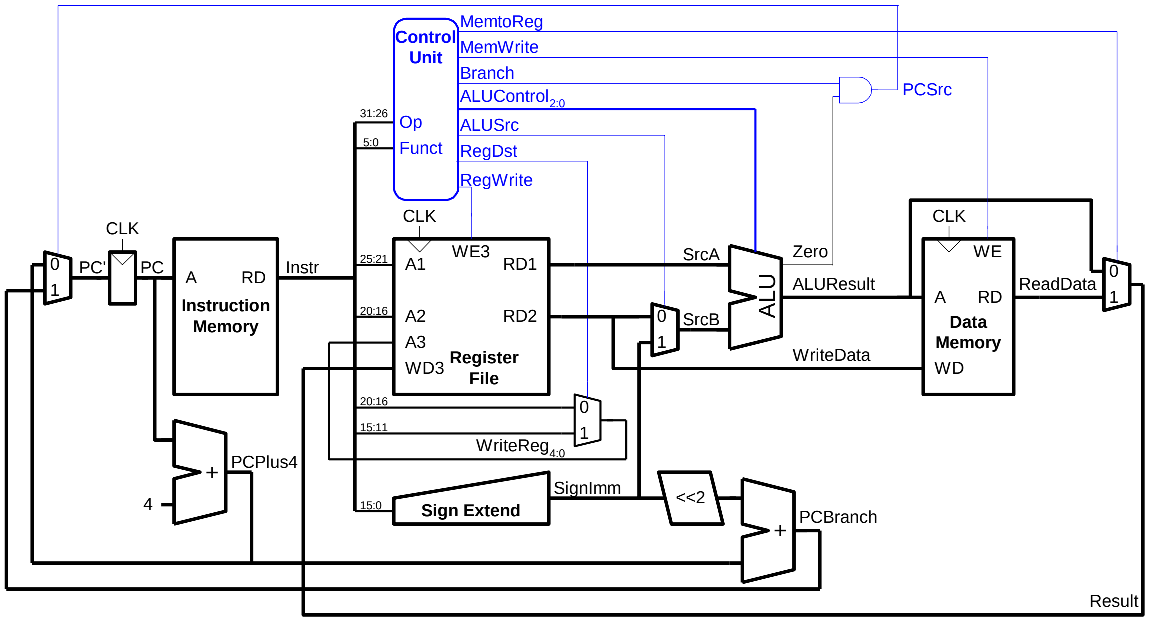
# Control Unit: ALU Decoder

| $ALUOp_{1:0}$ | Meaning |
|---|---|
| 00 | Add |
| 01 | Subtract |
| 10 | Look at Funct |
| 11 | Not Used |

| $ALUOp_{1:0}$ | Funct | $ALUControl_{2:0}$ |
|---|---|---|
| 00 | X | 010 (Add) |
| X1 | X | 110 (Subtract) |
| 1X | 100000 (add) | 010 (Add) |
| 1X | 100010 (sub) | 110 (Subtract) |
| 1X | 100100 (and) | 000 (And) |
| 1X | 100101 (or) | 001 (Or) |
| 1X | 101010 (slt) | 111 (SLT) |

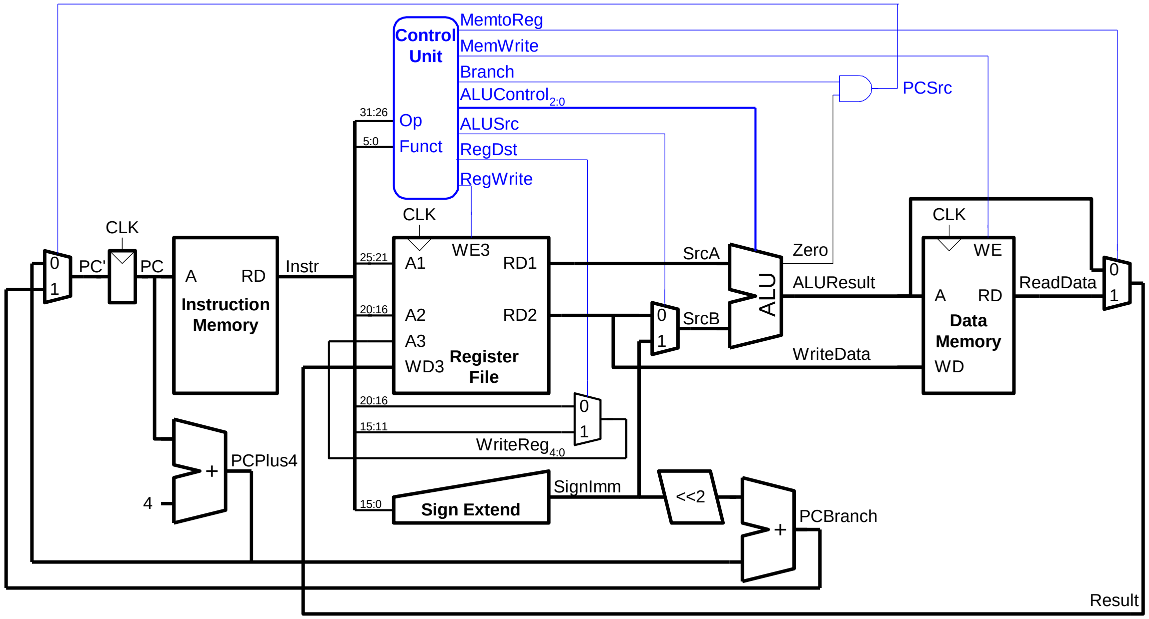
# Control Unit: Main Decoder

| Instruction | $Op_{5:0}$ | RegWrite | RegDst | AluSrc | Branch | MemWrite | MemtoReg | $ALUOp_{1:0}$ |
|---|---|---|---|---|---|---|---|---|
| R-type | 000000 | | | | | | | |
| lw | 100011 | | | | | | | |
| sw | 101011 | | | | | | | |
| beq | 000100 | | | | | | | |

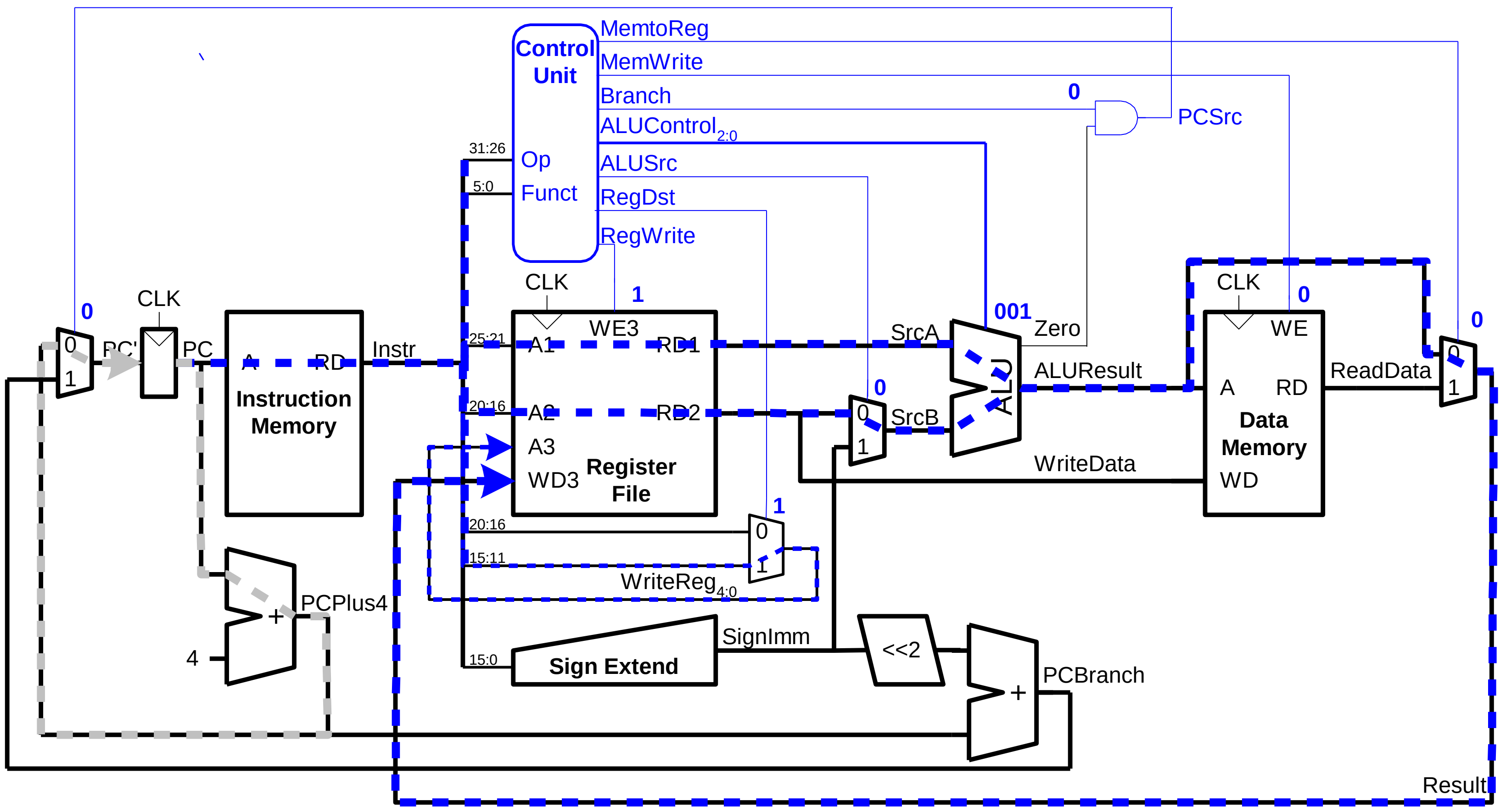


# Control Unit: Main Decoder

| Instruction | $Op_{5:0}$ | RegWrite | RegDst | AluSrc | Branch | MemWrite | MemtoReg | $ALUOp_{1:0}$ |
|---|---|---|---|---|---|---|---|---|
| R-type | 000000 | 1 | 1 | 0 | 0 | 0 | 0 | 10 |
| lw | 100011 | 1 | 0 | 1 | 0 | 0 | 0 | 00 |
| sw | 101011 | 0 | X | 1 | 0 | 1 | X | 00 |
| beq | 000100 | 0 | X | 0 | 1 | 0 | X | 01 |

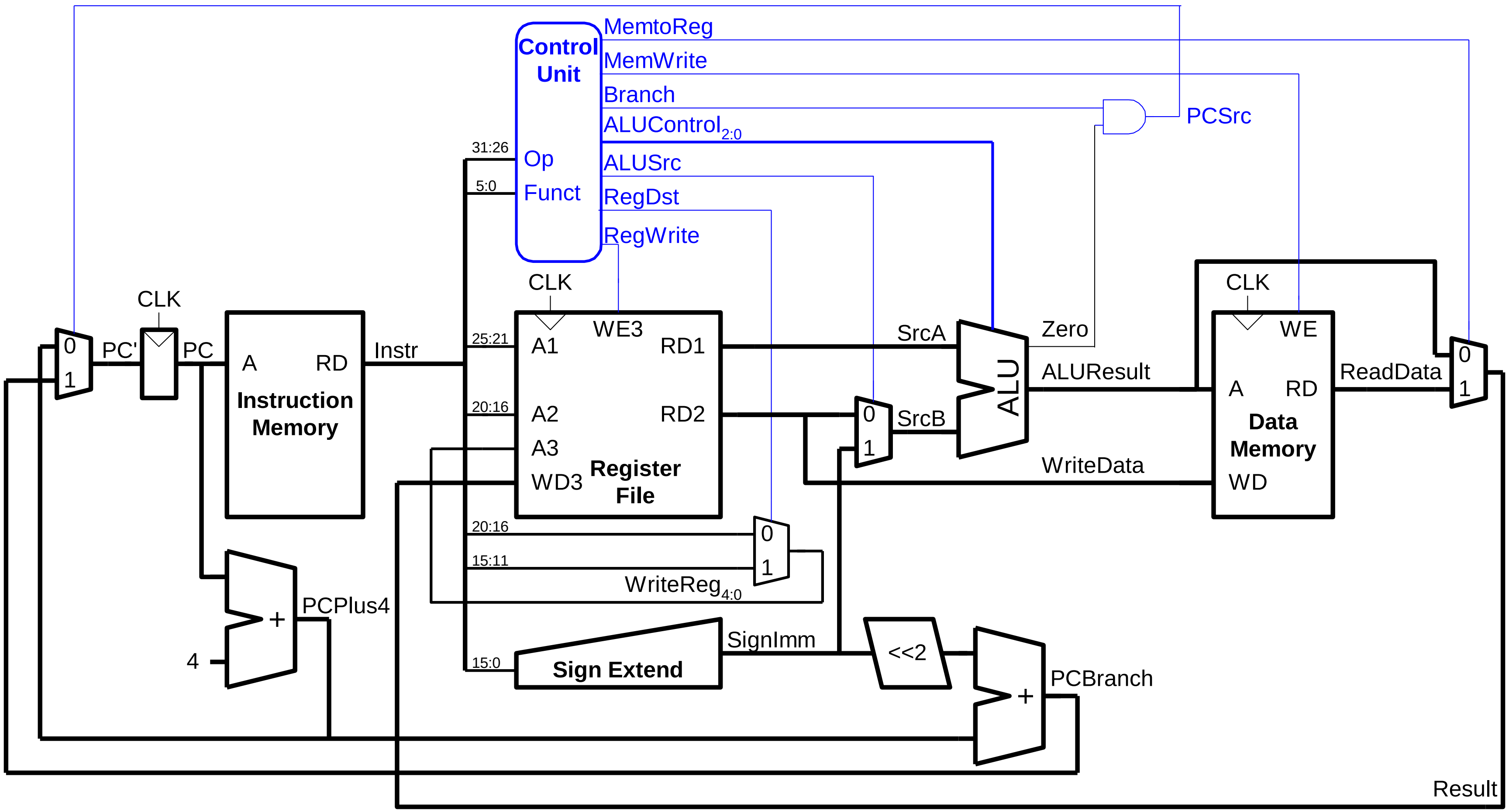


# Single-Cycle Datapath Example: `or`



# Extended Functionality: addi

- No change to datapath



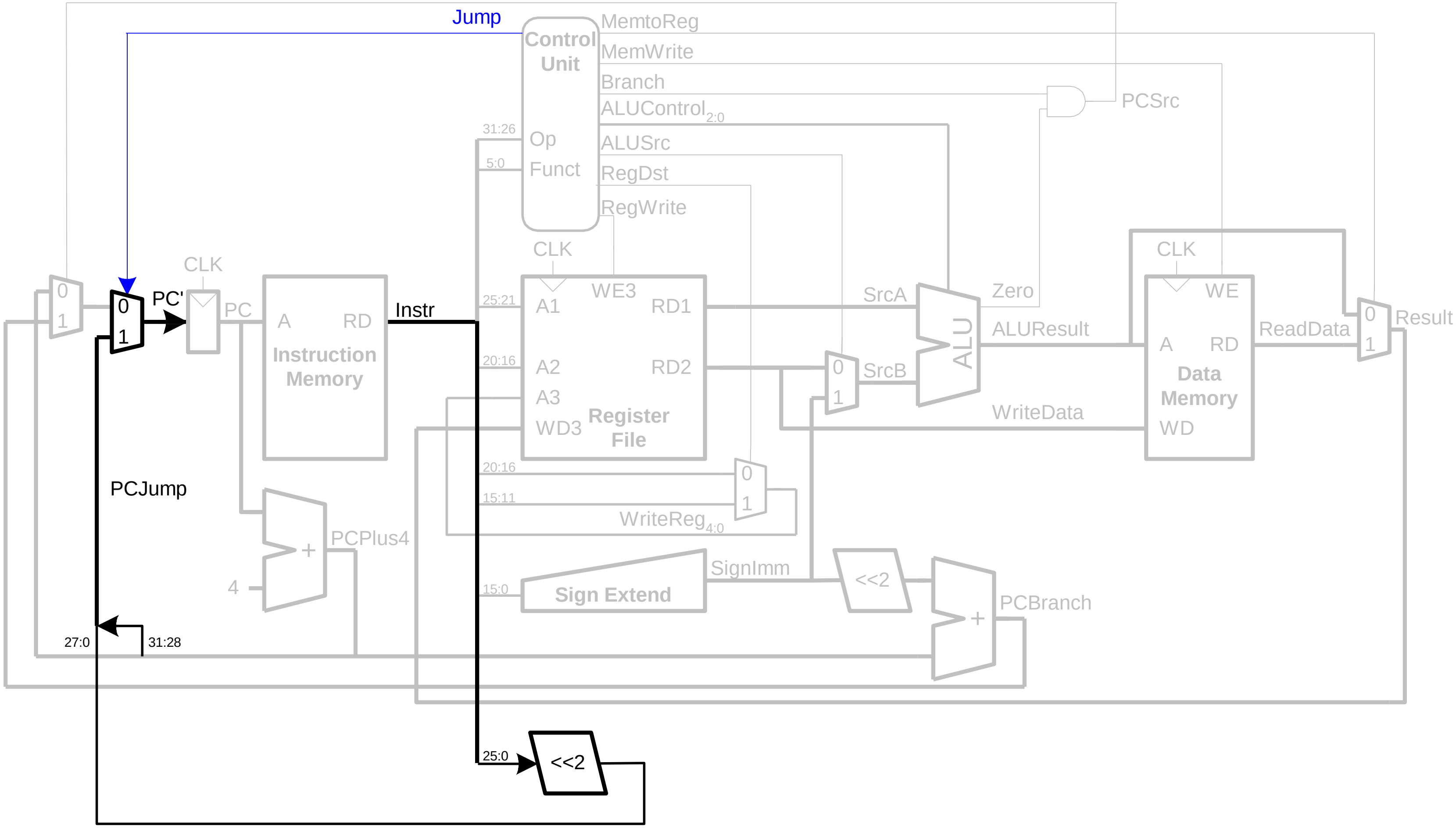
# Control Unit: addi

| Instruction | $Op_{5:0}$ | RegWrite | RegDst | AluSrc | Branch | MemWrite | MemtoReg | $ALUOp_{1:0}$ |
|---|---|---|---|---|---|---|---|---|
| R-type | 000000 | 1 | 1 | 0 | 0 | 0 | 0 | 10 |
| lw | 100011 | 1 | 0 | 1 | 0 | 0 | 1 | 00 |
| sw | 101011 | 0 | X | 1 | 0 | 1 | X | 00 |
| beq | 000100 | 0 | X | 0 | 1 | 0 | X | 01 |
| **addi** | **001000** | | | | | | | |

# **Control Unit:** `addi`

| Instruction | $Op_{5:0}$ | RegWrite | RegDst | AluSrc | Branch | MemWrite | MemtoReg | $ALUOp_{1:0}$ |
|---|---|---|---|---|---|---|---|---|
| R-type | 000000 | 1 | 1 | 0 | 0 | 0 | 0 | 10 |
| `lw` | 100011 | 1 | 0 | 1 | 0 | 0 | 1 | 00 |
| `sw` | 101011 | 0 | X | 1 | 0 | 1 | X | 00 |
| `beq` | 000100 | 0 | X | 0 | 1 | 0 | X | 01 |
| **`addi`** | **001000** | **1** | **0** | **1** | **0** | **0** | **0** | **00** |

# Extended Functionality: j



# Control Unit: Main Decoder

| Instruction | $Op_{5:0}$ | RegWrite | RegDst | AluSrc | Branch | MemWrite | MemtoReg | $ALUOp_{1:0}$ | Jump |
|---|---|---|---|---|---|---|---|---|---|
| R-type | 000000 | 1 | 1 | 0 | 0 | 0 | 0 | 10 | 0 |
| `lw` | 100011 | 1 | 0 | 1 | 0 | 0 | 1 | 00 | 0 |
| `sw` | 101011 | 0 | X | 1 | 0 | 1 | X | 00 | 0 |
| `beq` | 000100 | 0 | X | 0 | 1 | 0 | X | 01 | 0 |
| `j` | **000100** | | | | | | | | |

# Control Unit: Main Decoder

| Instruction | Op$_{5:0}$ | RegWrite | RegDst | AluSrc | Branch | MemWrite | MemtoReg | ALUOp$_{1:0}$ | Jump |
|---|---|---|---|---|---|---|---|---|---|
| R-type | 000000 | 1 | 1 | 0 | 0 | 0 | 0 | 10 | 0 |
| lw | 100011 | 1 | 0 | 1 | 0 | 0 | 1 | 00 | 0 |
| sw | 101011 | 0 | X | 1 | 0 | 1 | X | 00 | 0 |
| beq | 000100 | 0 | X | 0 | 1 | 0 | X | 01 | 0 |
| j | **000100** | **0** | **X** | **X** | **X** | **0** | **X** | **XX** | **1** |

# Review: Processor Performance

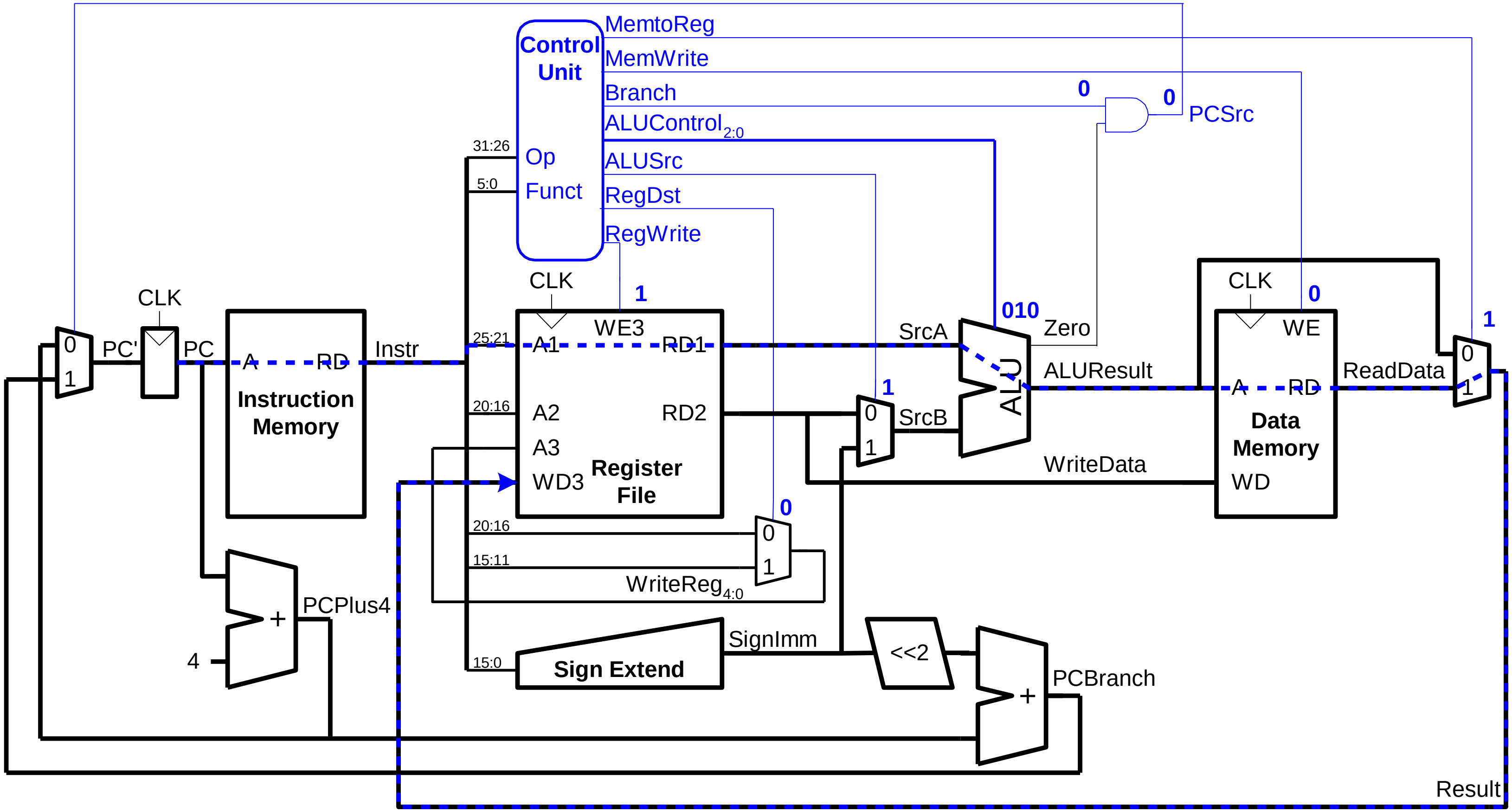
Program Execution Time

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

= # instructions x CPI x $T_C$

# Single-Cycle Performance

- $T_C$ is limited by the critical path (`lw`)



# Single-Cycle Performance

- Single-cycle critical path:

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

- In most implementations, limiting paths are:
  - memory, ALU, register file.
  - $T_c = t_{pcq\_PC} + 2t_{\mathrm{mem}} + t_{RF\mathrm{read}} + t_{\mathrm{mux}} + t_{\mathrm{ALU}} + t_{RF\mathrm{setup}}$

# Single-Cycle Performance Example

| Element | Parameter | Delay (ps) |
|---|---|---|
| Register clock-to-Q | $t_{pcq\_\mathrm{PC}}$ | 30 |
| Register setup | $t_{\mathrm{setup}}$ | 20 |
| Multiplexer | $t_{\mathrm{mux}}$ | 25 |
| ALU | $t_{\mathrm{ALU}}$ | 200 |
| Memory read | $t_{\mathrm{mem}}$ | 250 |
| Register file read | $t_{RF\mathrm{read}}$ | 150 |
| Register file setup | $t_{RF\mathrm{setup}}$ | 20 |

$T_c =$

# Single-Cycle Performance Example

| **Element** | **Parameter** | **Delay (ps)** |
|---|---|---|
| Register clock-to-Q | $t_{pcq\_\mathrm{PC}}$ | 30 |
| Register setup | $t_{\mathrm{setup}}$ | 20 |
| Multiplexer | $t_{\mathrm{mux}}$ | 25 |
| ALU | $t_{\mathrm{ALU}}$ | 200 |
| Memory read | $t_{\mathrm{mem}}$ | 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

- For a program with 100 billion instructions executing on a single-cycle MIPS processor,

Execution Time =

# Single-Cycle Performance Example

- For a program with 100 billion instructions executing on a single-cycle MIPS processor,

$$
\begin{aligned}
\text{Execution Time} &= \#\text{ instructions x CPI x } T_C \\
&= (100 \times 10^{9})(1)(925 \times 10^{-12}\,\text{s}) \\
&= 92.5 \text{ seconds}
\end{aligned}
$$

# Multicycle MIPS Processor

- Single-cycle microarchitecture:
  - + simple
  - - cycle time limited by longest instruction (`lw`)
  - - two adders/ALUs and two memories
- Multicycle microarchitecture:
  - + higher clock speed
  - + simpler instructions run faster
  - + reuse expensive hardware on multiple cycles
  - - sequencing overhead paid many times
- Same design steps: datapath & control

# Performance of Single-Cycle Machines

- Memory Unit 2 ns
- ALU and Adders 2 ns
- Register file (Read or Write) 1 ns

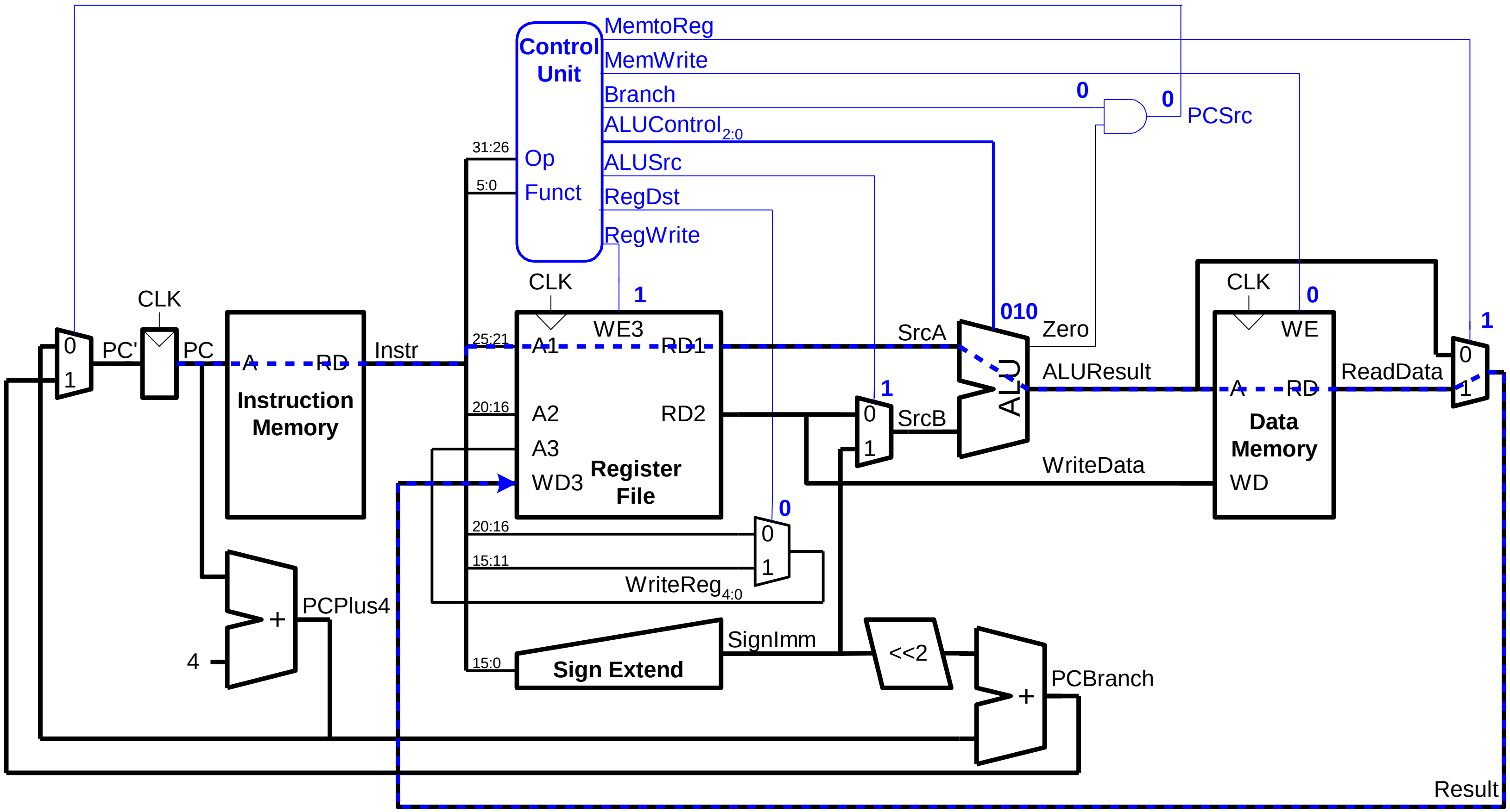
| Class | Fetch | Decode | ALU | Memory | Write Back | Total |
|---|---|---|---|---|---|---|
| R-format | 2 | 1 | 2 | 0 | 1 | 6 |
| LW | 2 | 1 | 2 | 2 | 1 | 8 |
| SW | 2 | 1 | 2 | 2 |  | 7ns |
| Branch | 2 | 1 | 2 |  |  | 5ns |
| Jump | 2 |  |  |  |  | 2ns |

**Rtype: 44%, LW: 24%, SW: 12% , BRANCH: 18%, JUMP: 2%**

Execution=I*T*CPI= 8*24%+7*12%+6*44%+5*18%+2*2%=6.3 ns

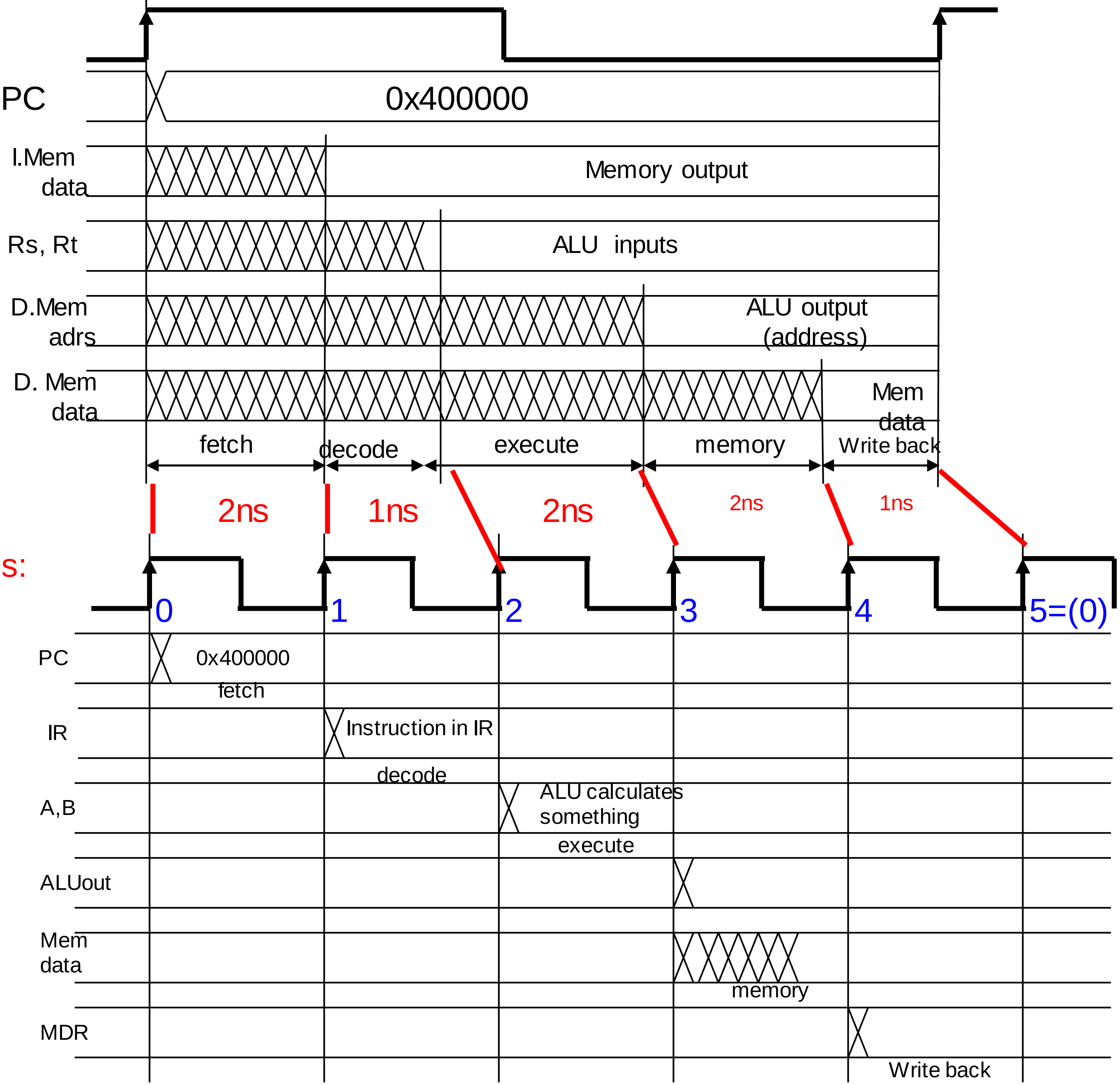
# Single-Cycle Performance

- $T_C$ is limited by the critical path (`lw`)



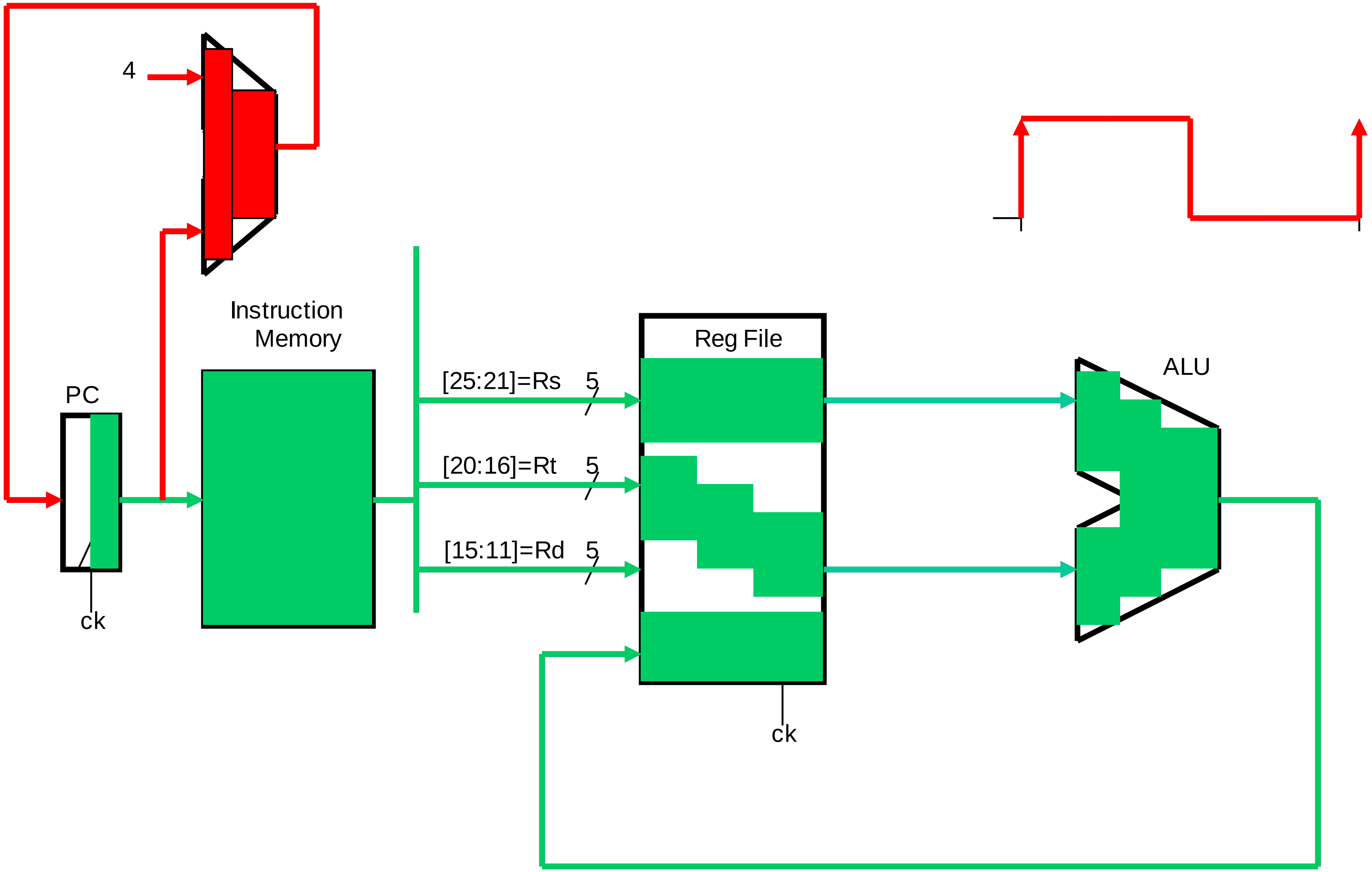
## Timing of a lw instruction in a single cycle CPU

We want to replace a long single CK cycle with 5 short ones:



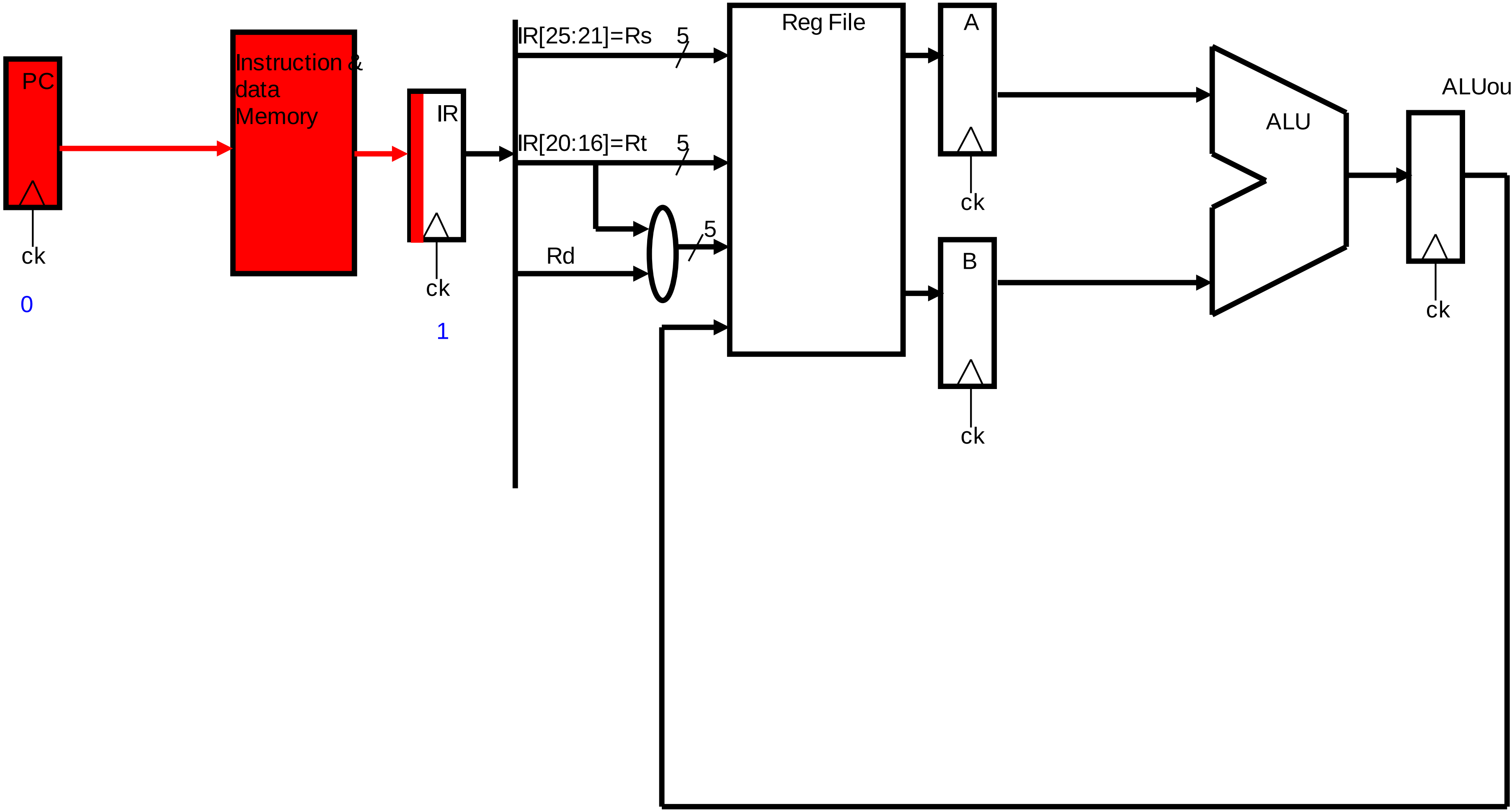
**Timing of a lw instruction in a multi-cycle CPU**

## A single cycle CPU demo:  R-type instruction



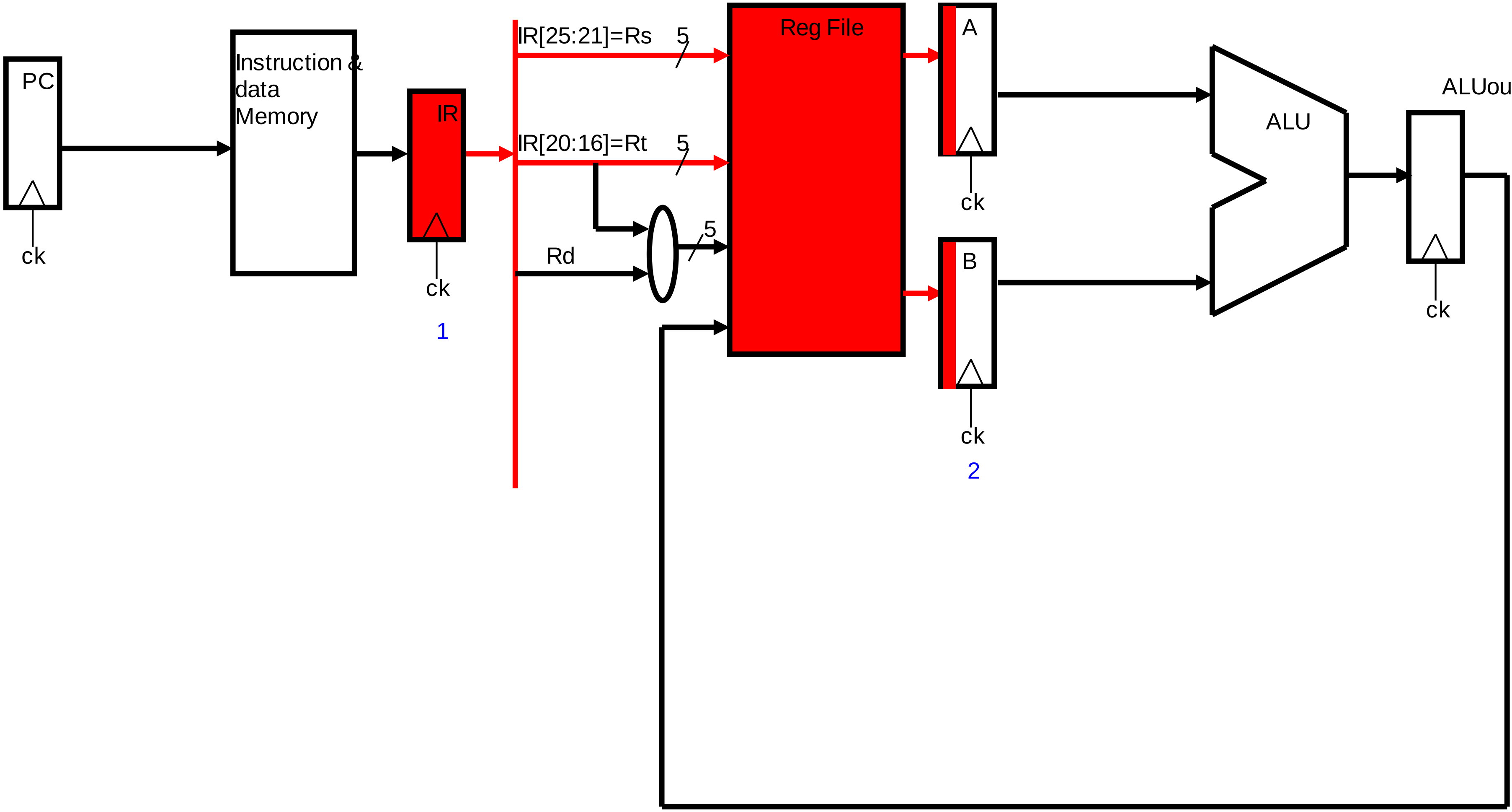
# A multi cycle CPU capable of R-type & instructions

## fetch



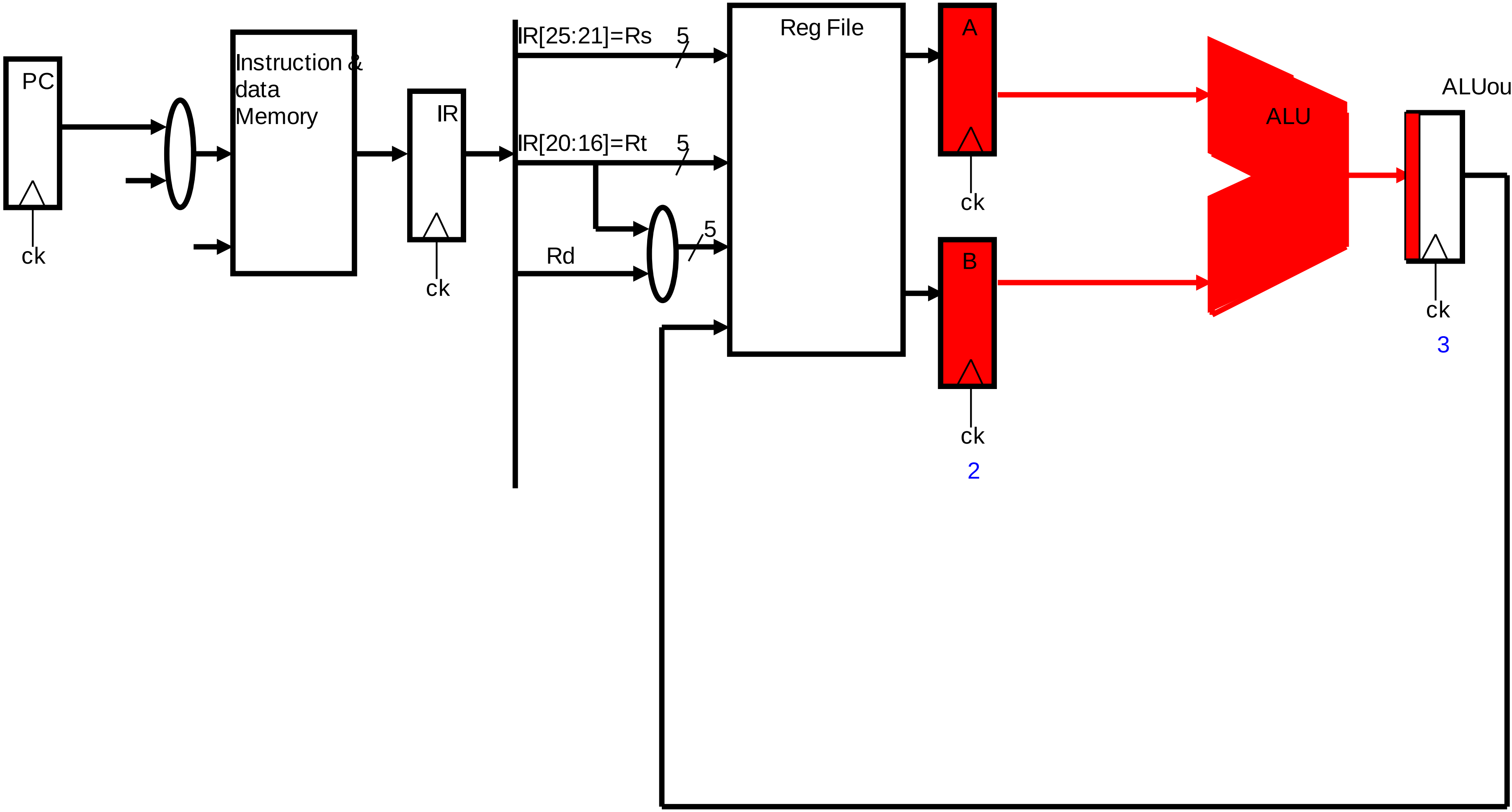
# A multi cycle CPU capable of R-type & instructions

# decode



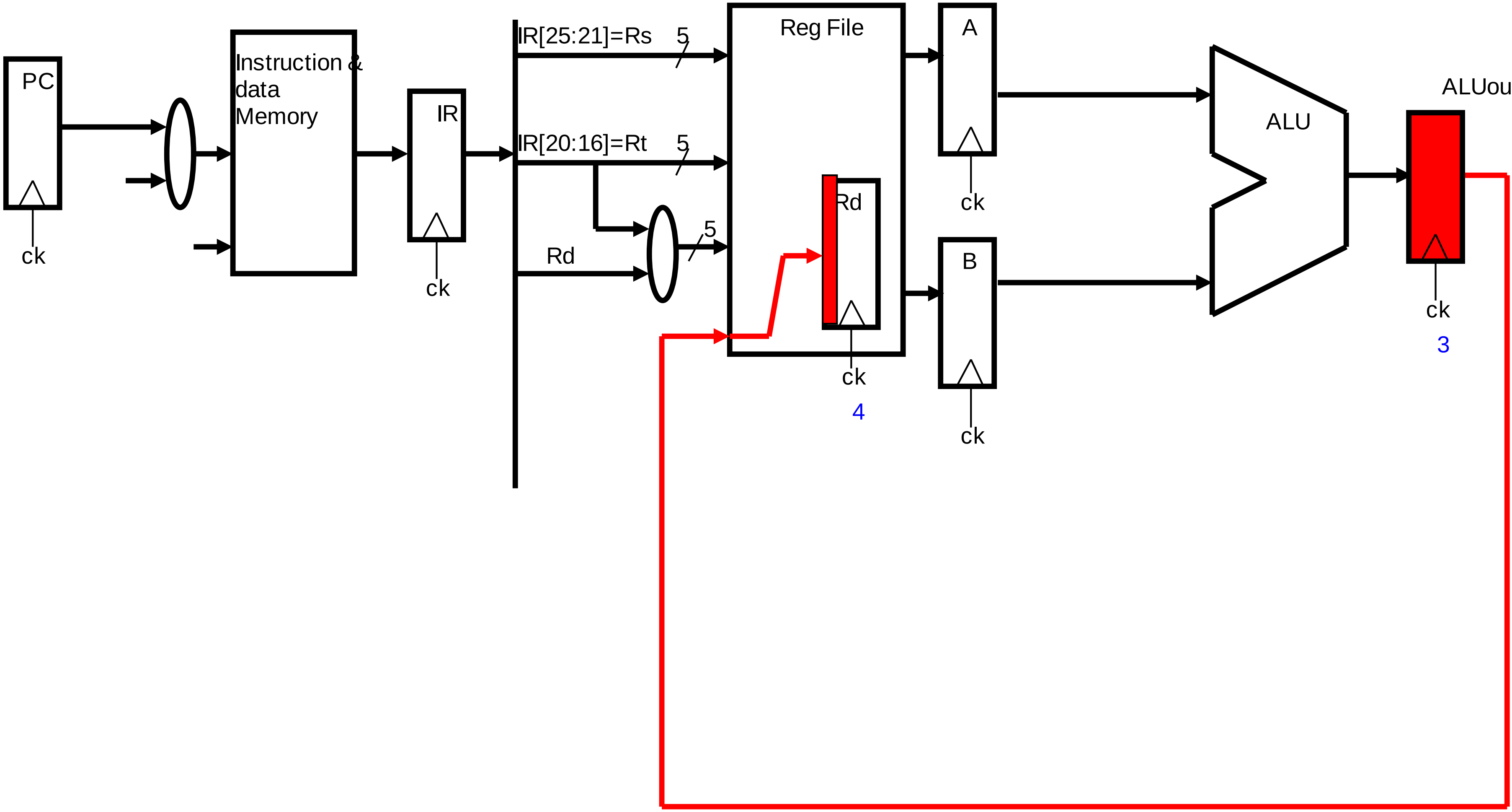
# A multi cycle CPU capable of R-type & instructions

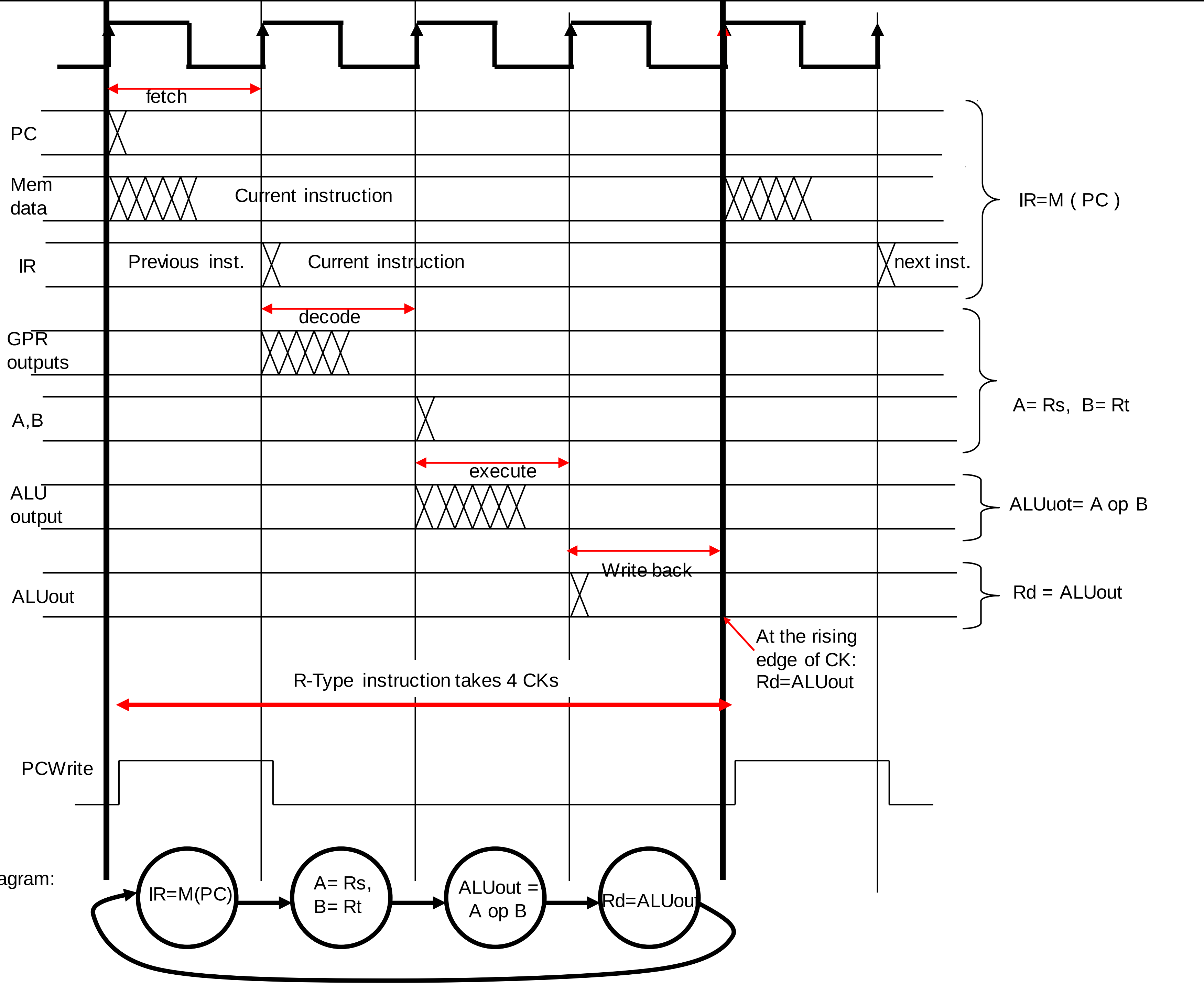
## execute



# A multi cycle CPU capable of R-type & instructions

# write back



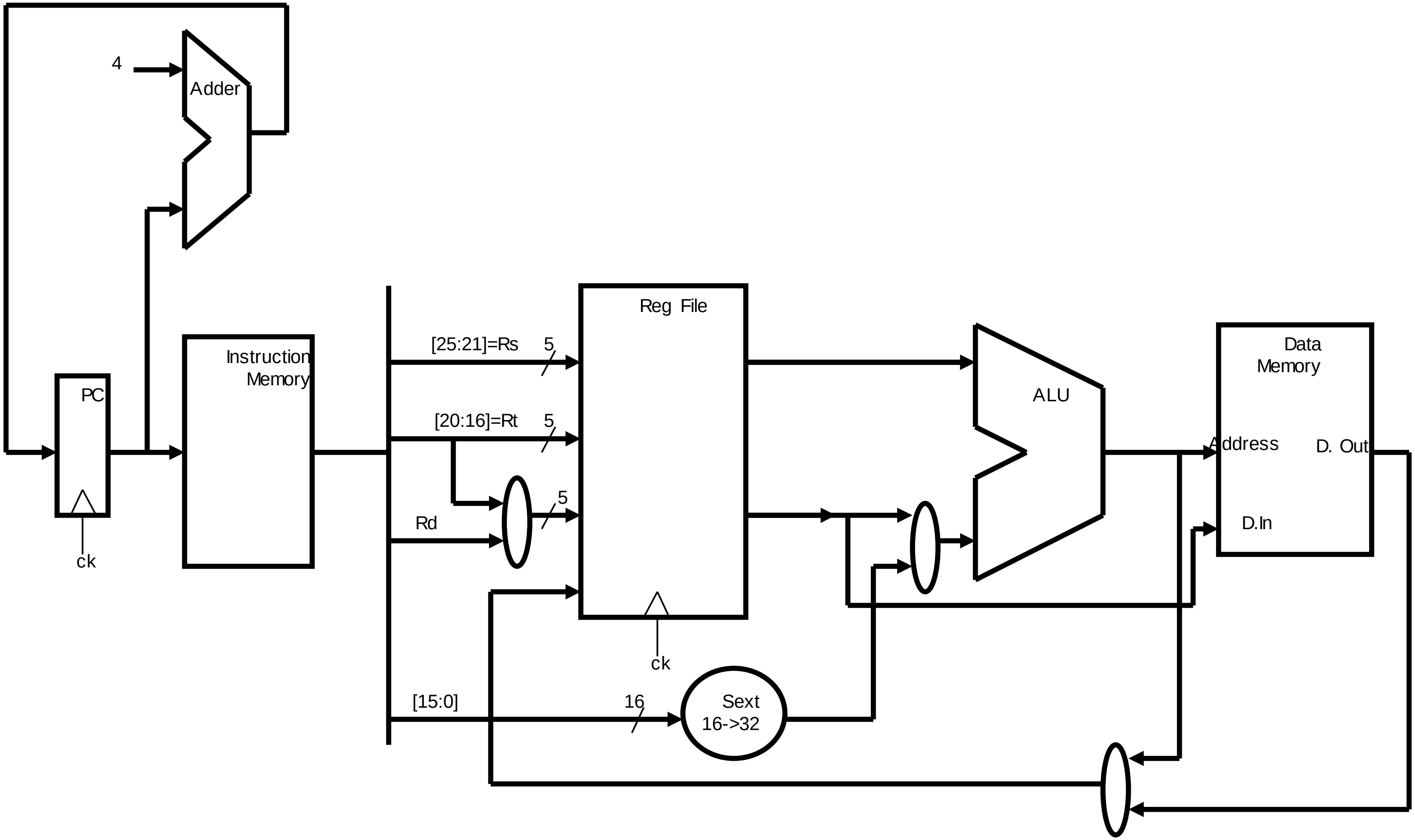


At the rising edge of CK: Rd=ALUout

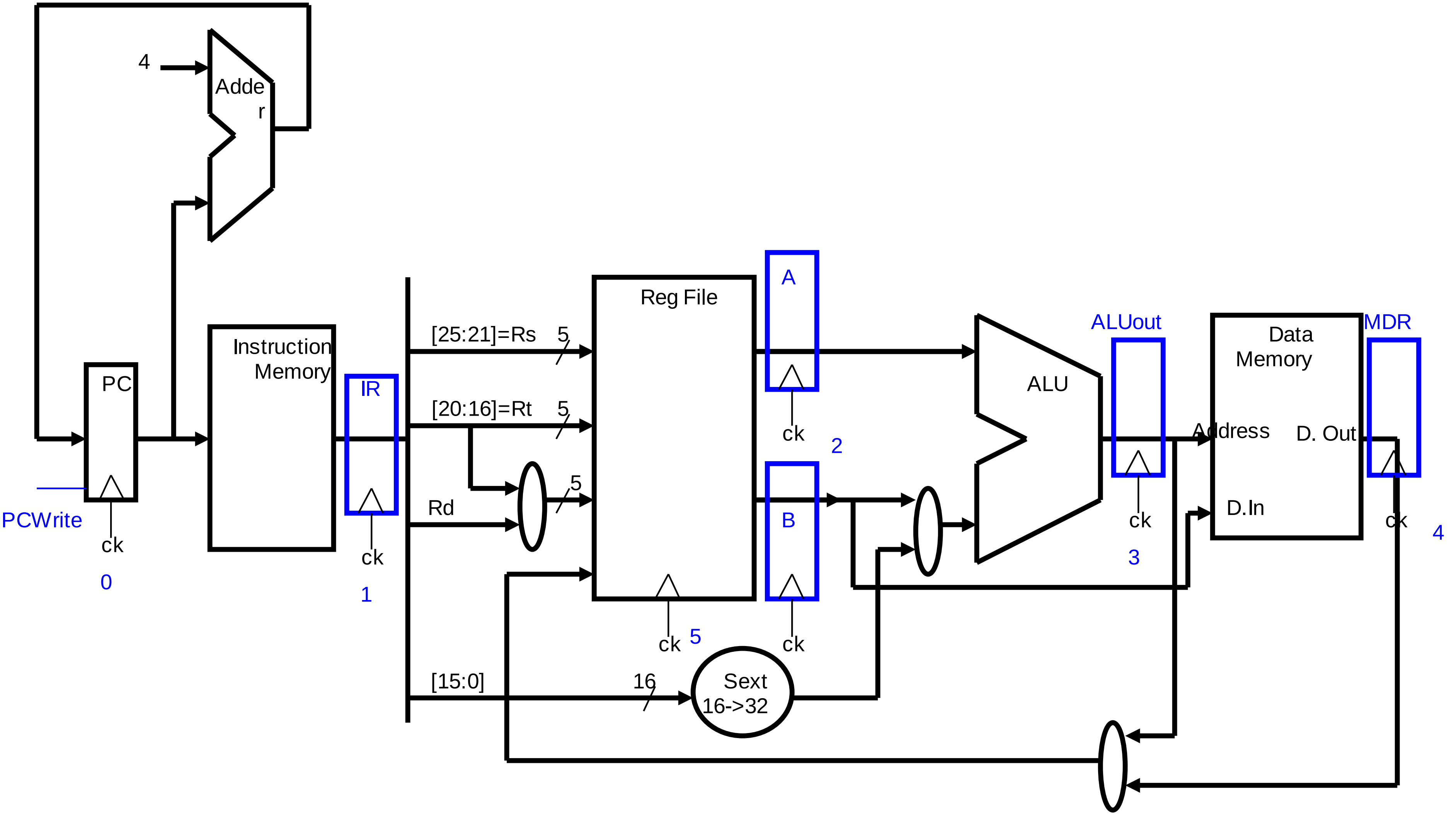
R-Type instruction takes 4 CKs

The state diagram:

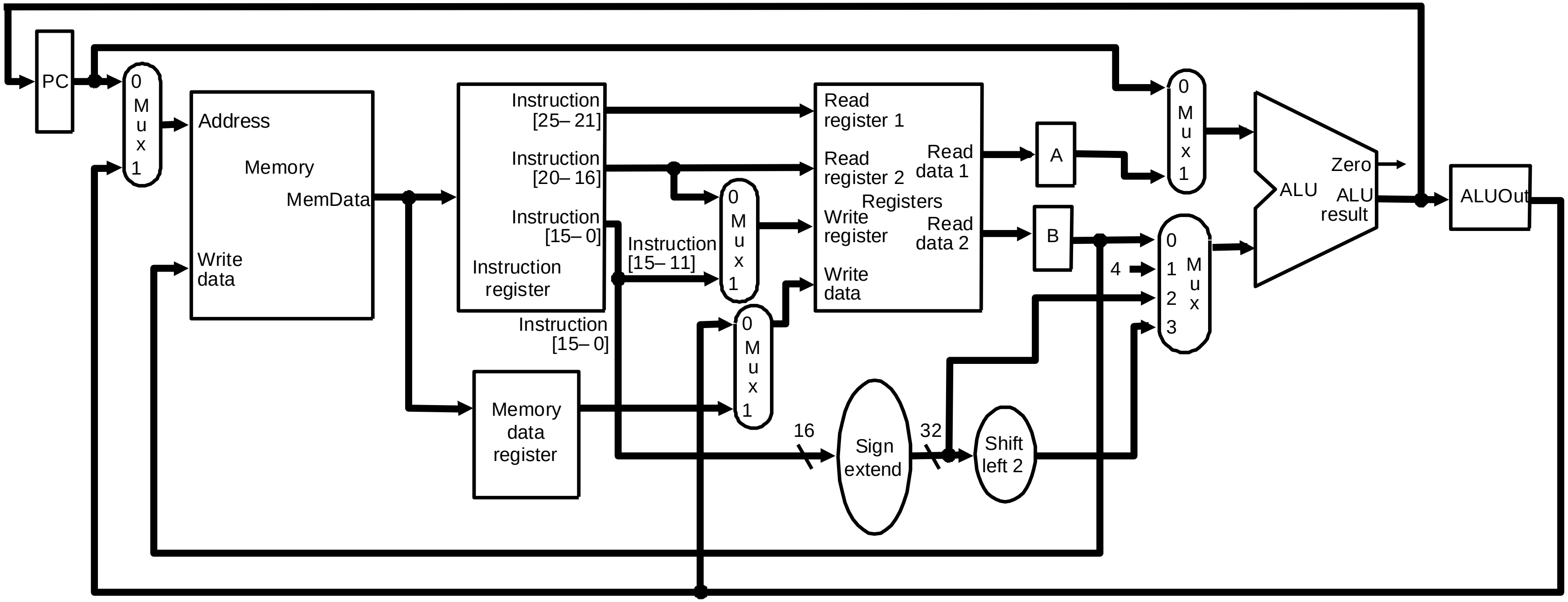
# single cycle CPU -> multicycle



# Adding registers to “split” the instruction to 5 stages:

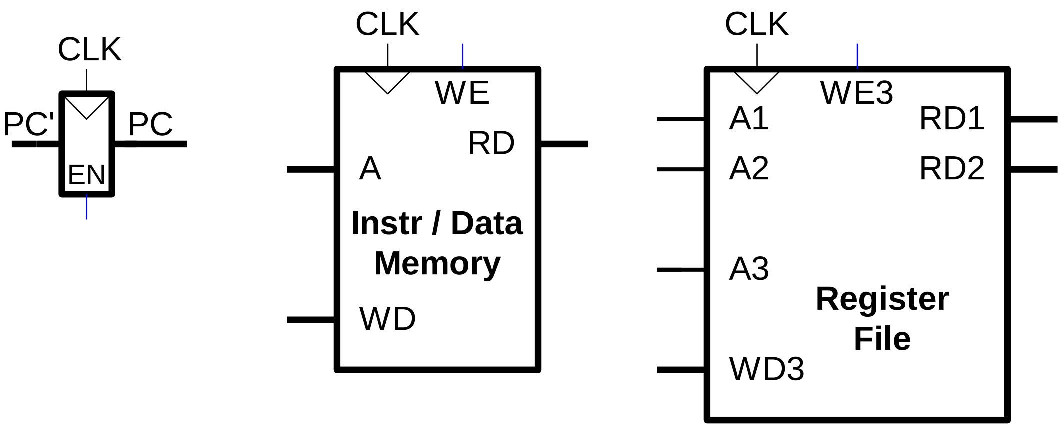


# Here is the book's version of the multi-cycle CPU:



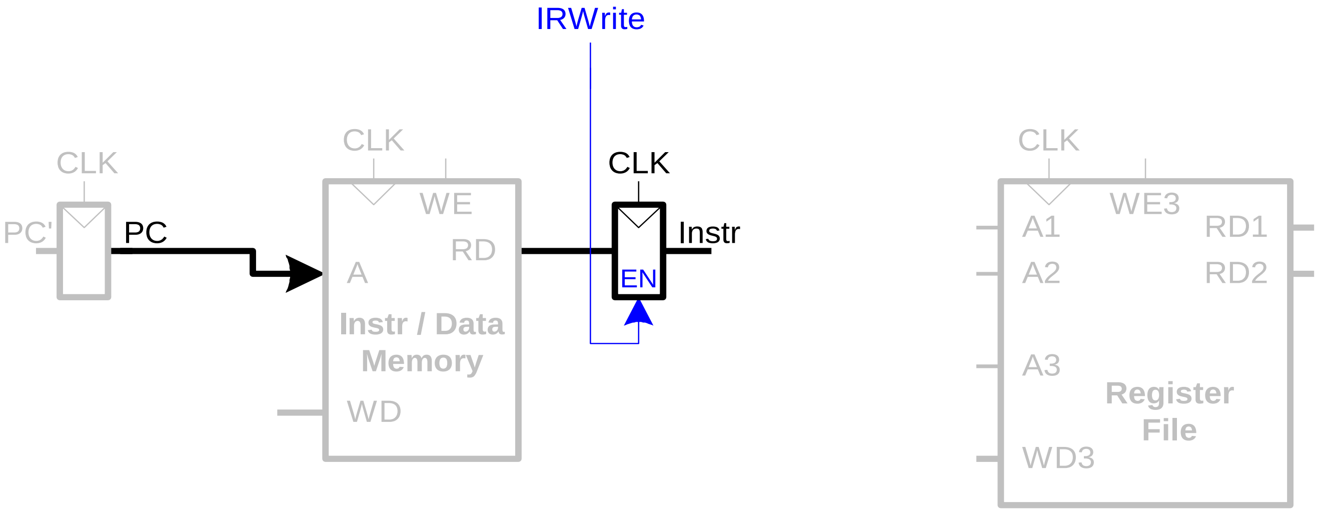
# Multicycle State Elements

- Replace Instruction and Data memories with a single unified memory
  - More realistic

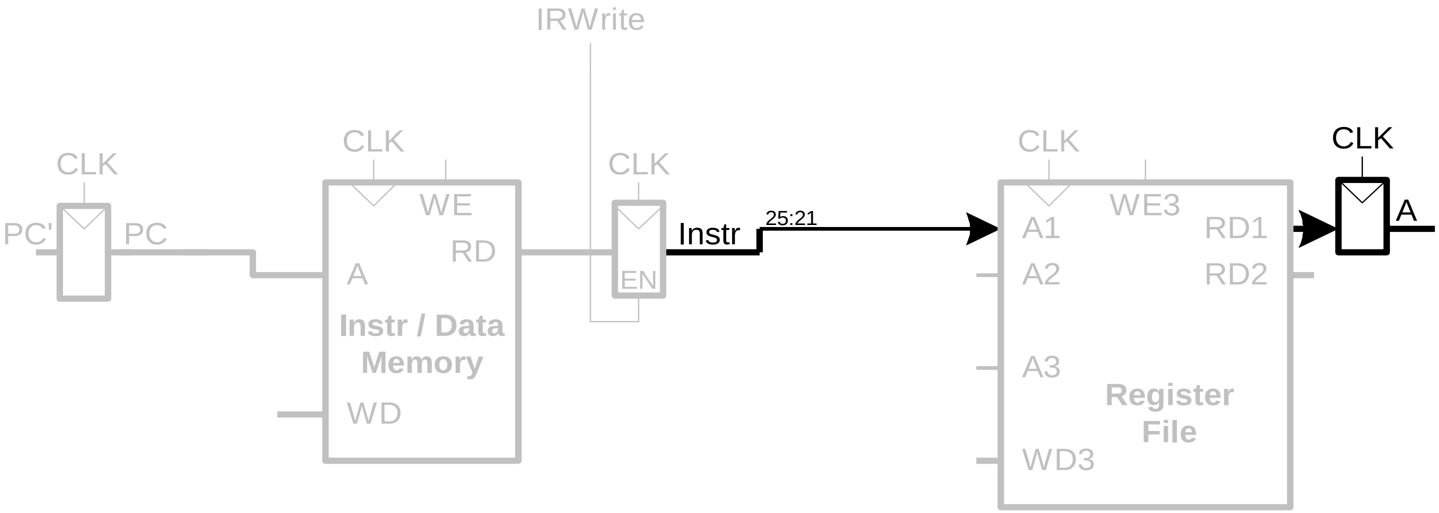


# Multicycle Datapath: instruction fetch

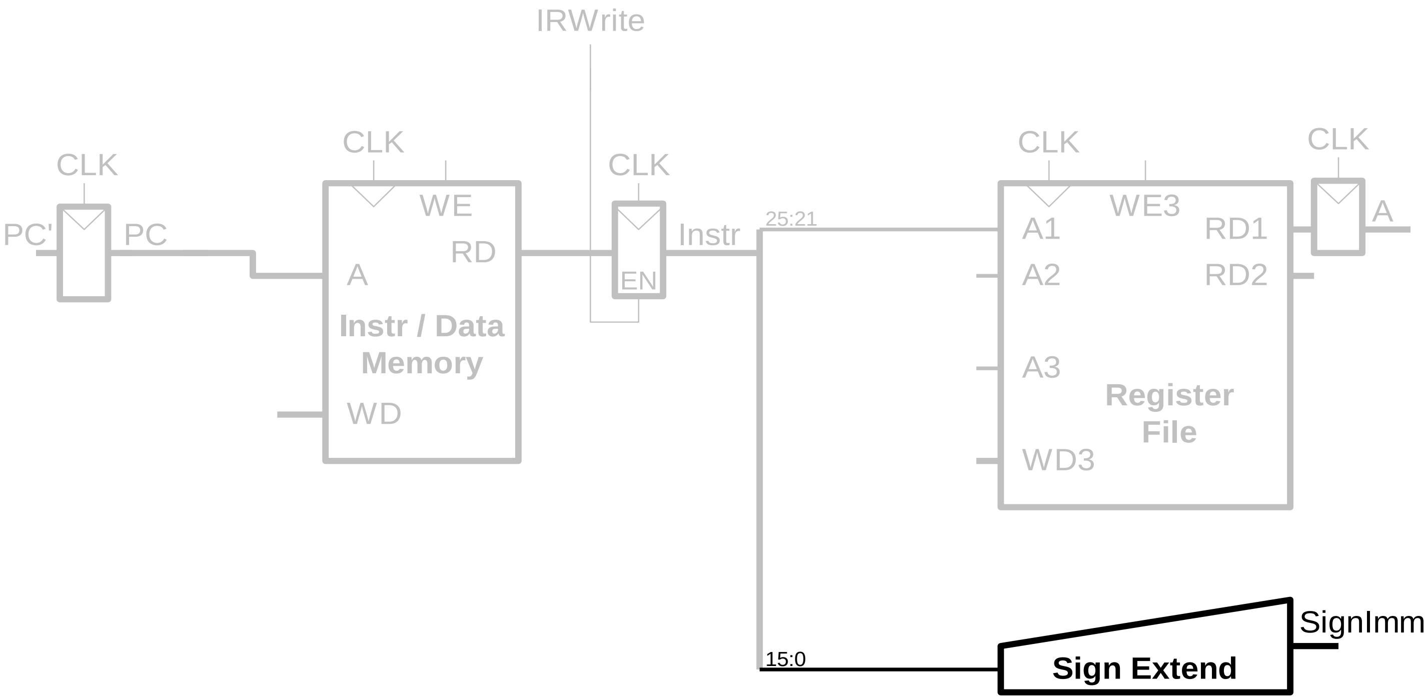
- First consider executing `lw`
- **STEP 1:** Fetch instruction



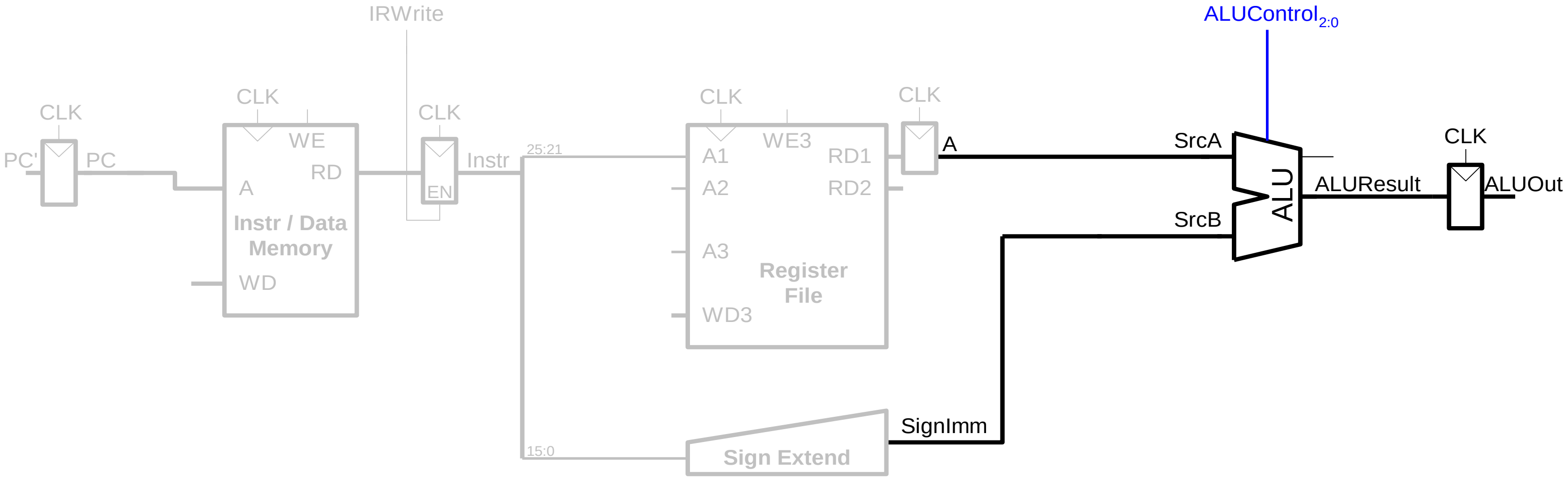
# Multicycle Datapath: `lw` register read



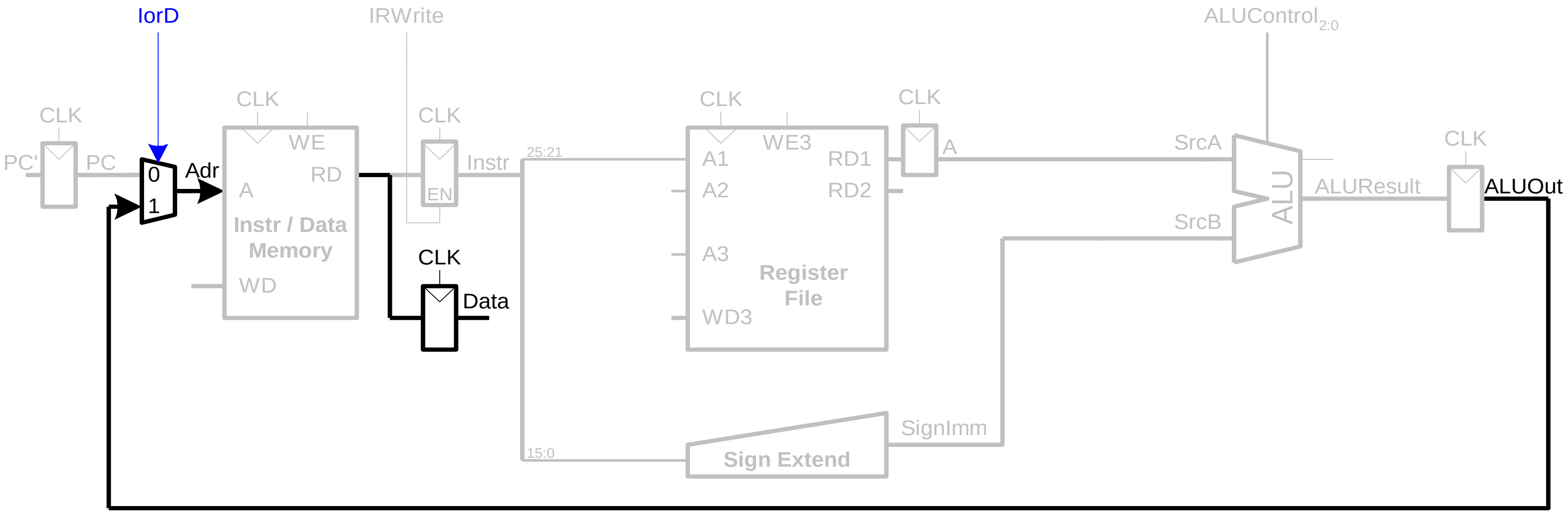
# Multicycle Datapath: `lw` immediate



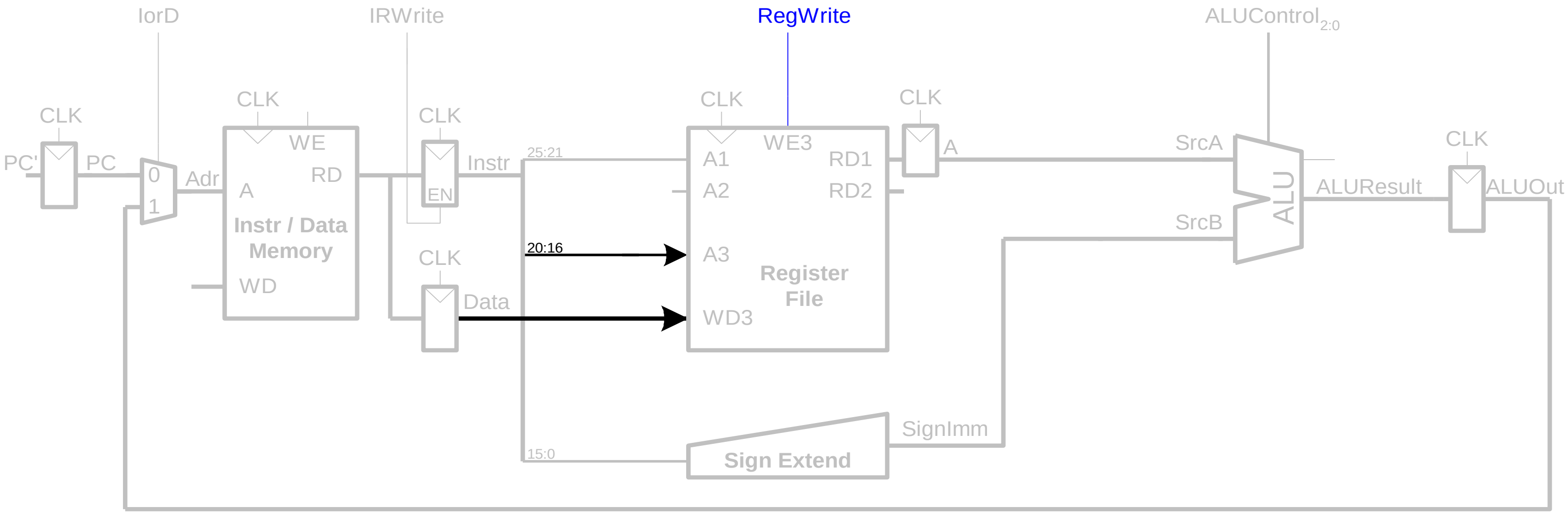
# Multicycle Datapath: `lw` address



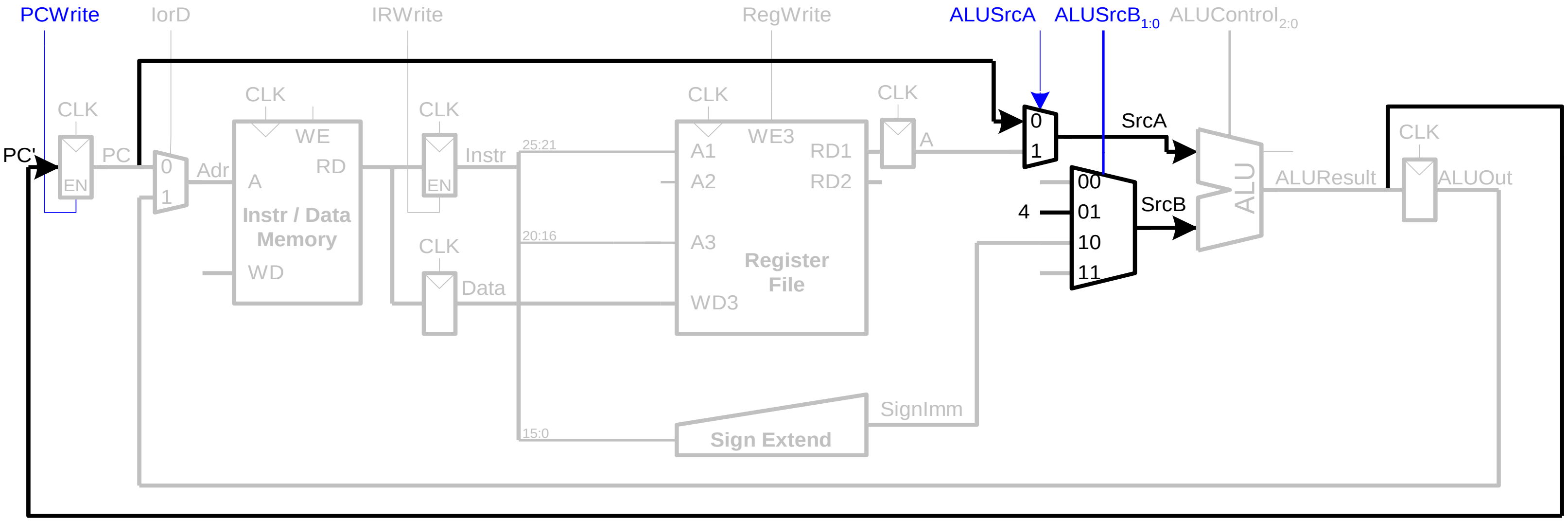
# Multicycle Datapath: `lw` memory read



# Multicycle Datapath: `lw` write register

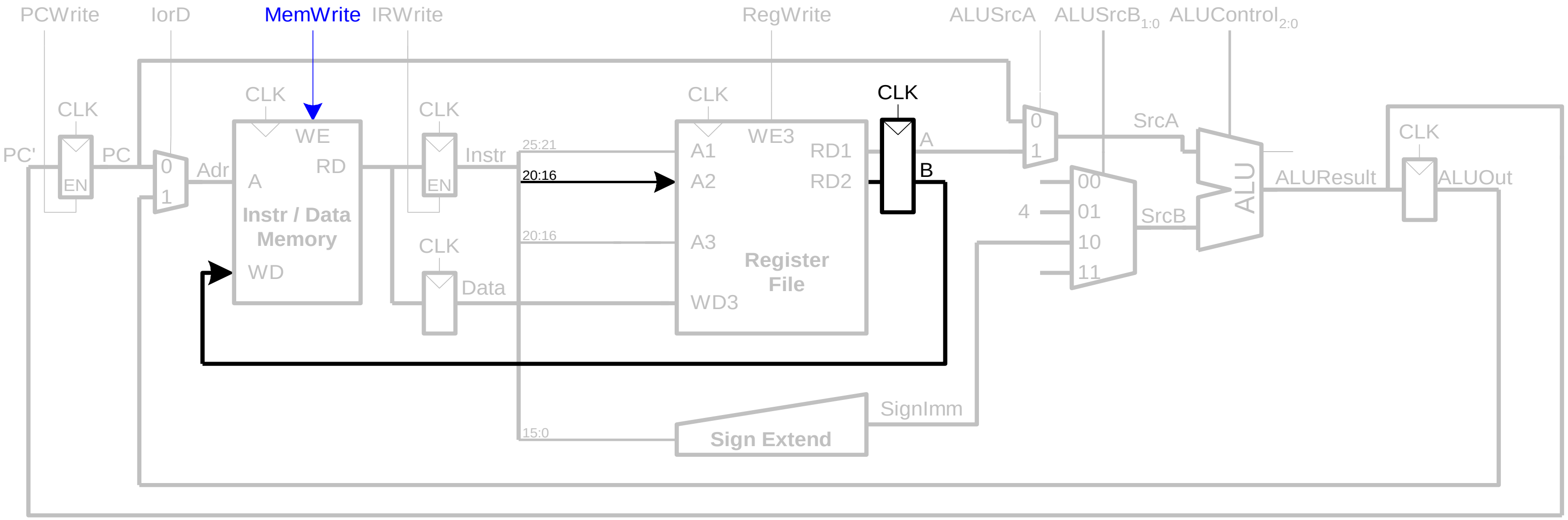


# Multicycle Datapath: increment PC



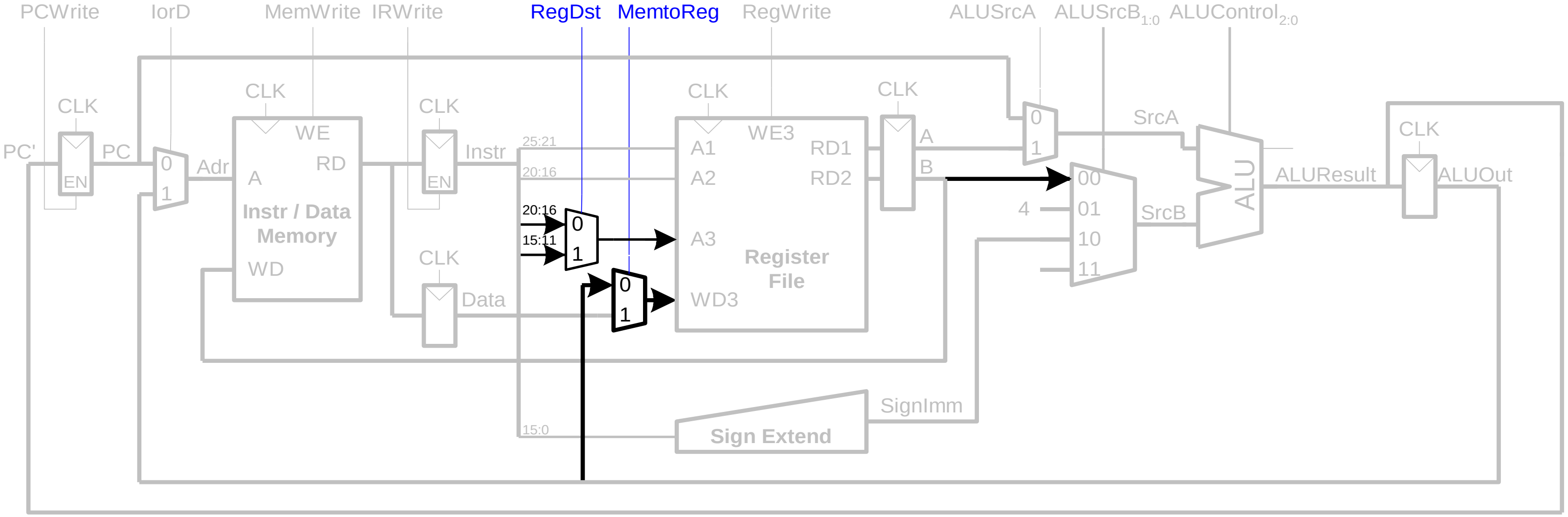
# Multicycle Datapath: `sw`

- Write data in `rt` to memory



# Multicycle Datapath: R-type Instructions

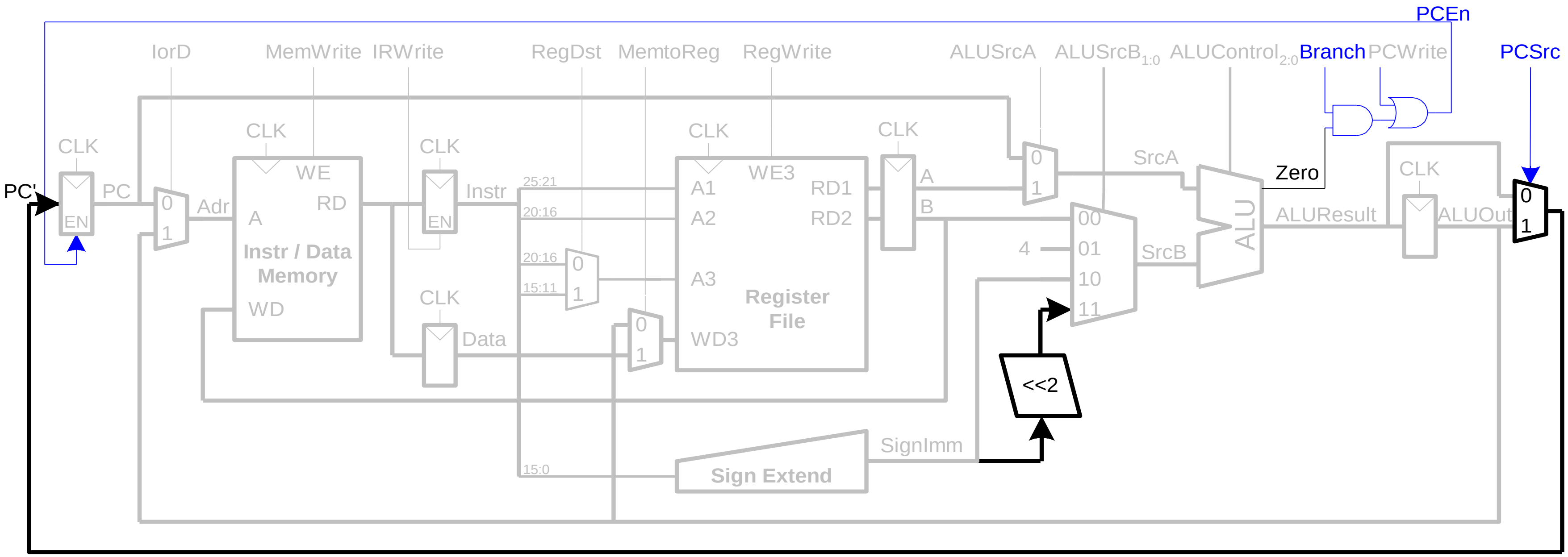
- Read from `rs` and `rt`
- Write *ALUResult* to register file
- Write to `rd` (instead of `rt`)



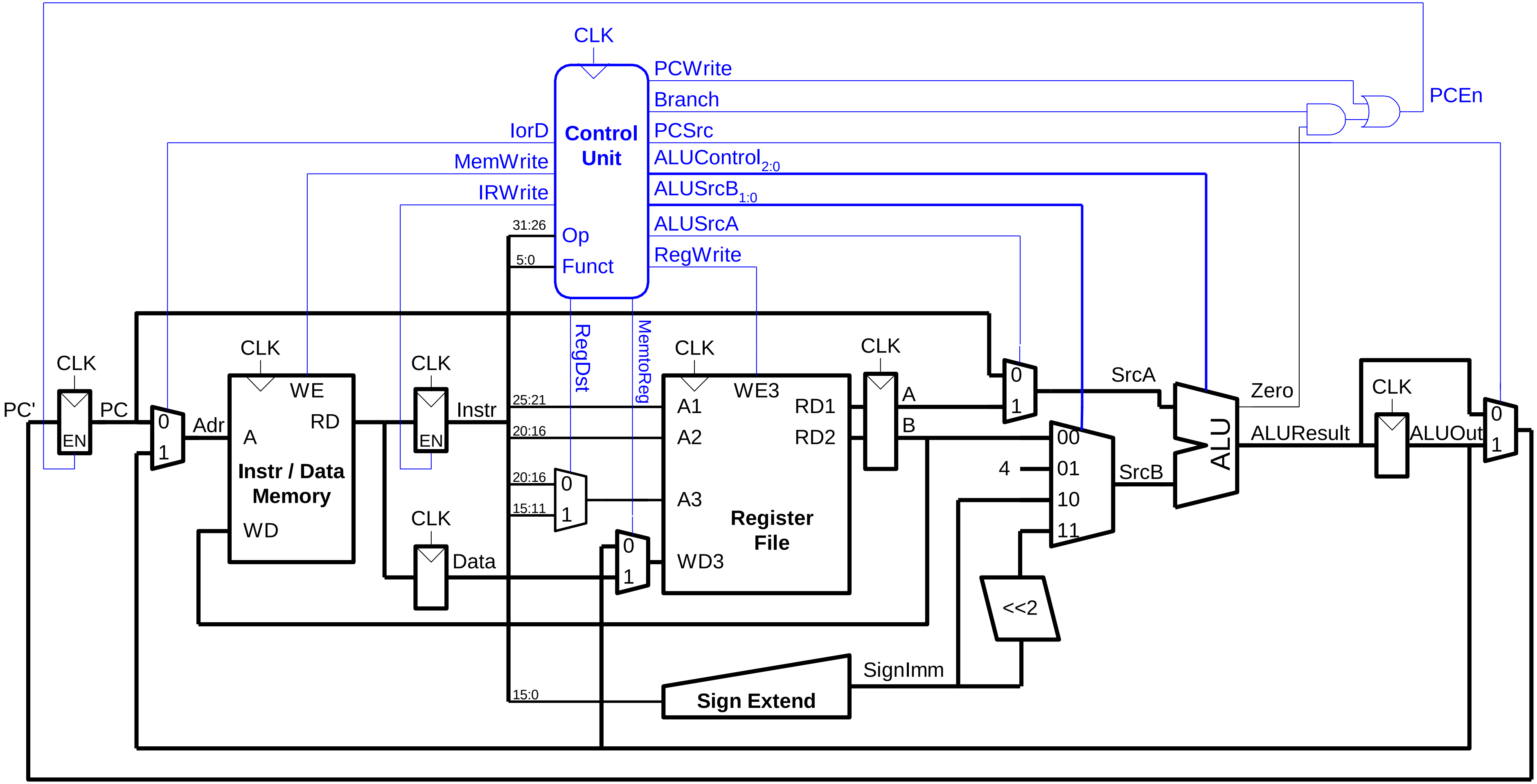
# Multicycle Datapath: `beq`

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

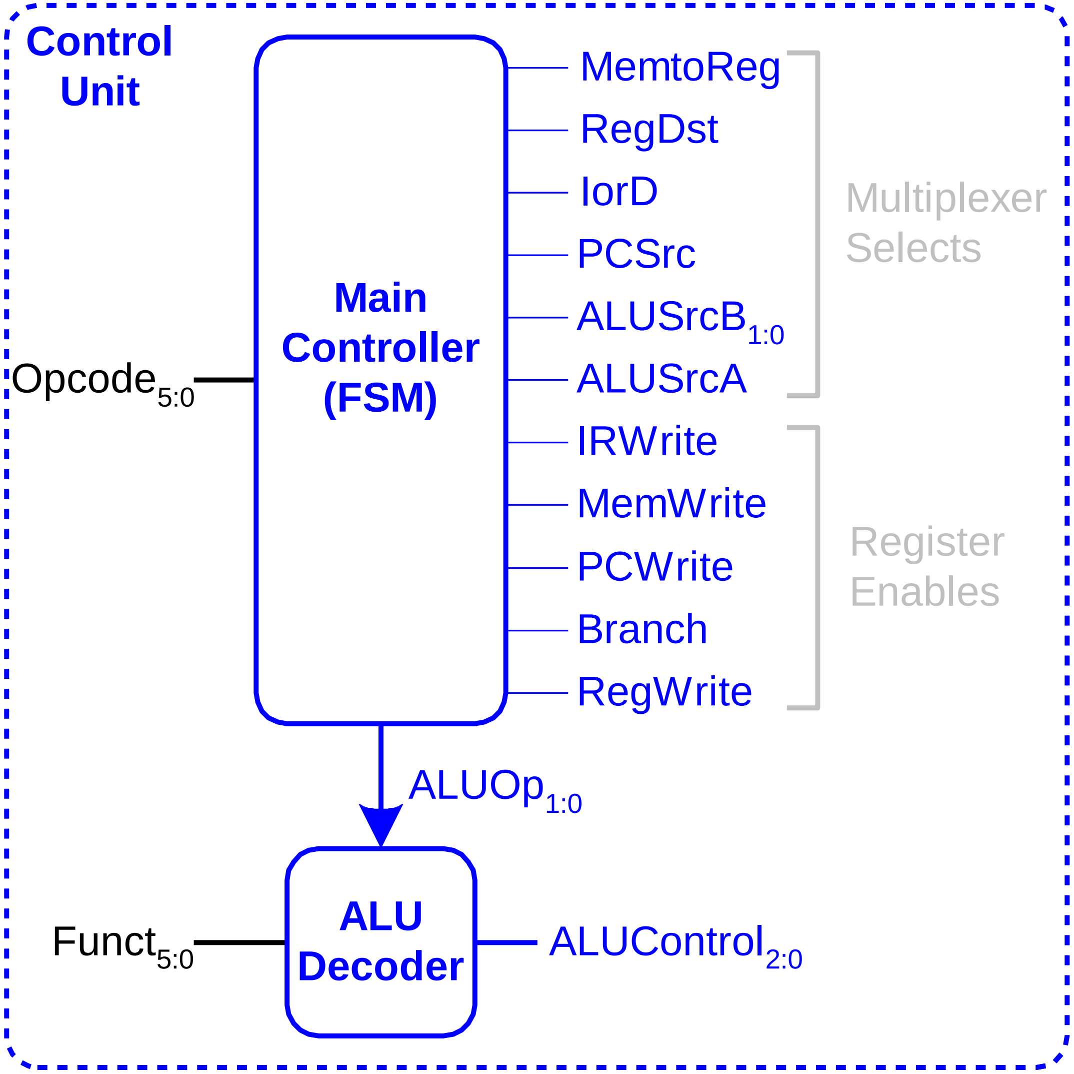
  BTA = (sign-extended immediate << 2) + (PC+4)



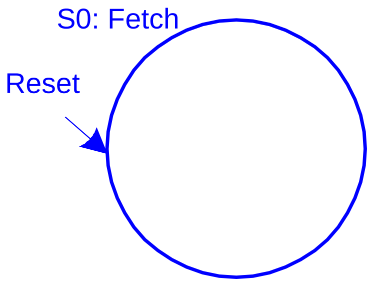
# Complete Multicycle Processor

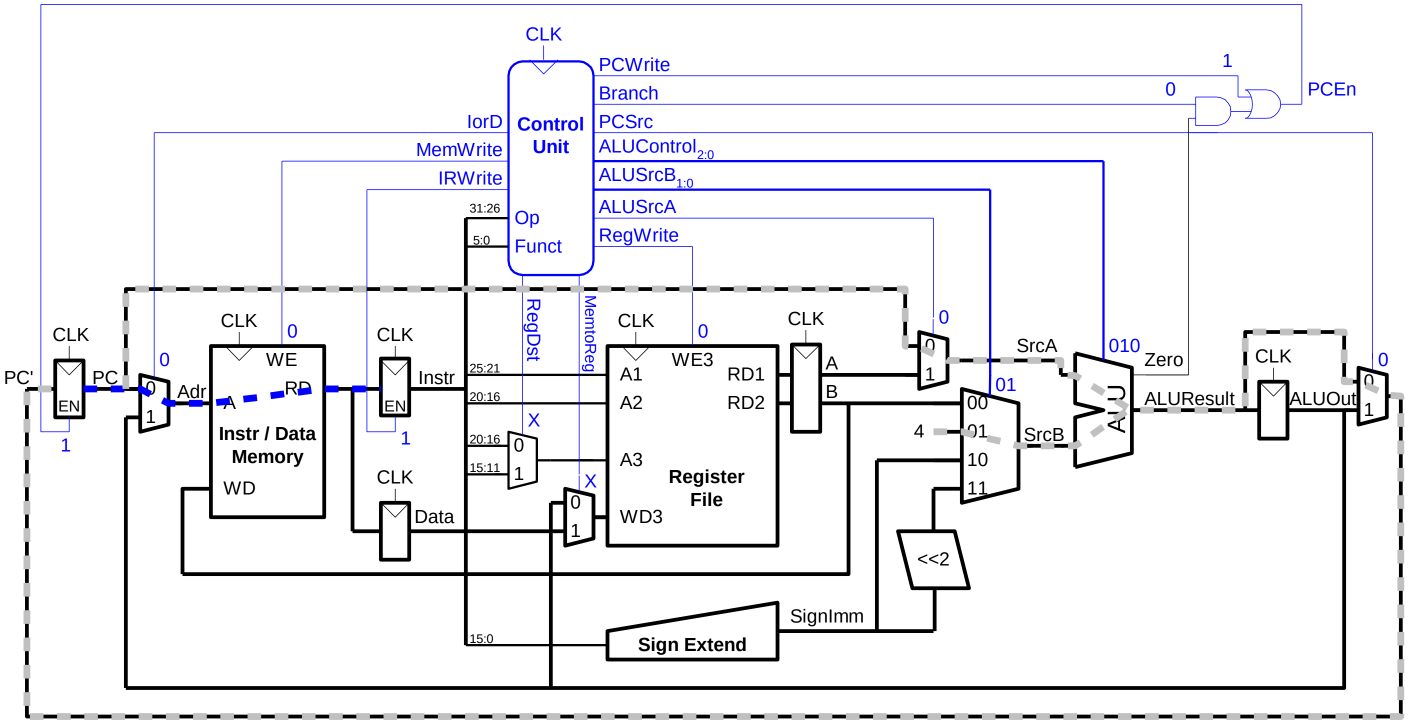


# Control Unit

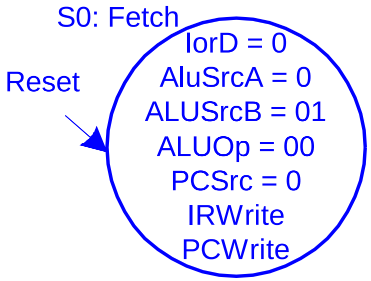


# Main Controller FSM: Fetch





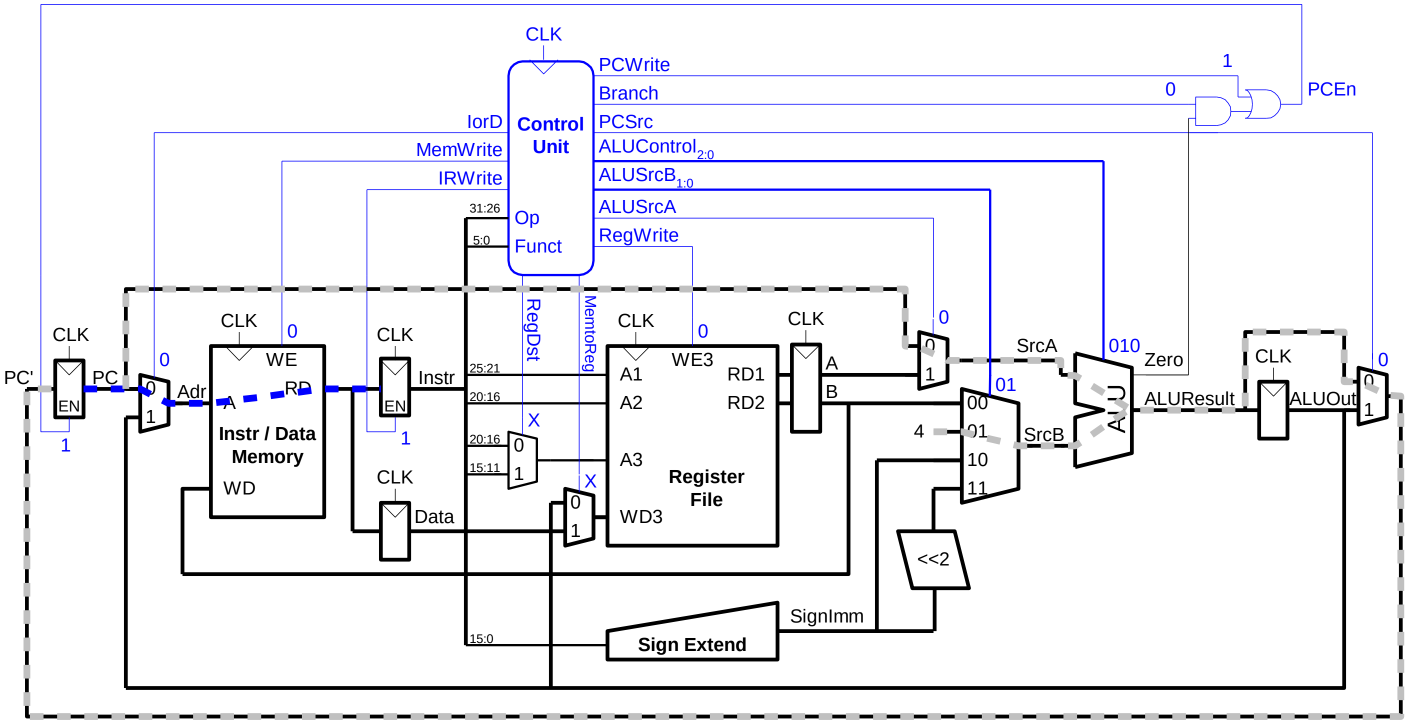
# Main Controller FSM: Fetch



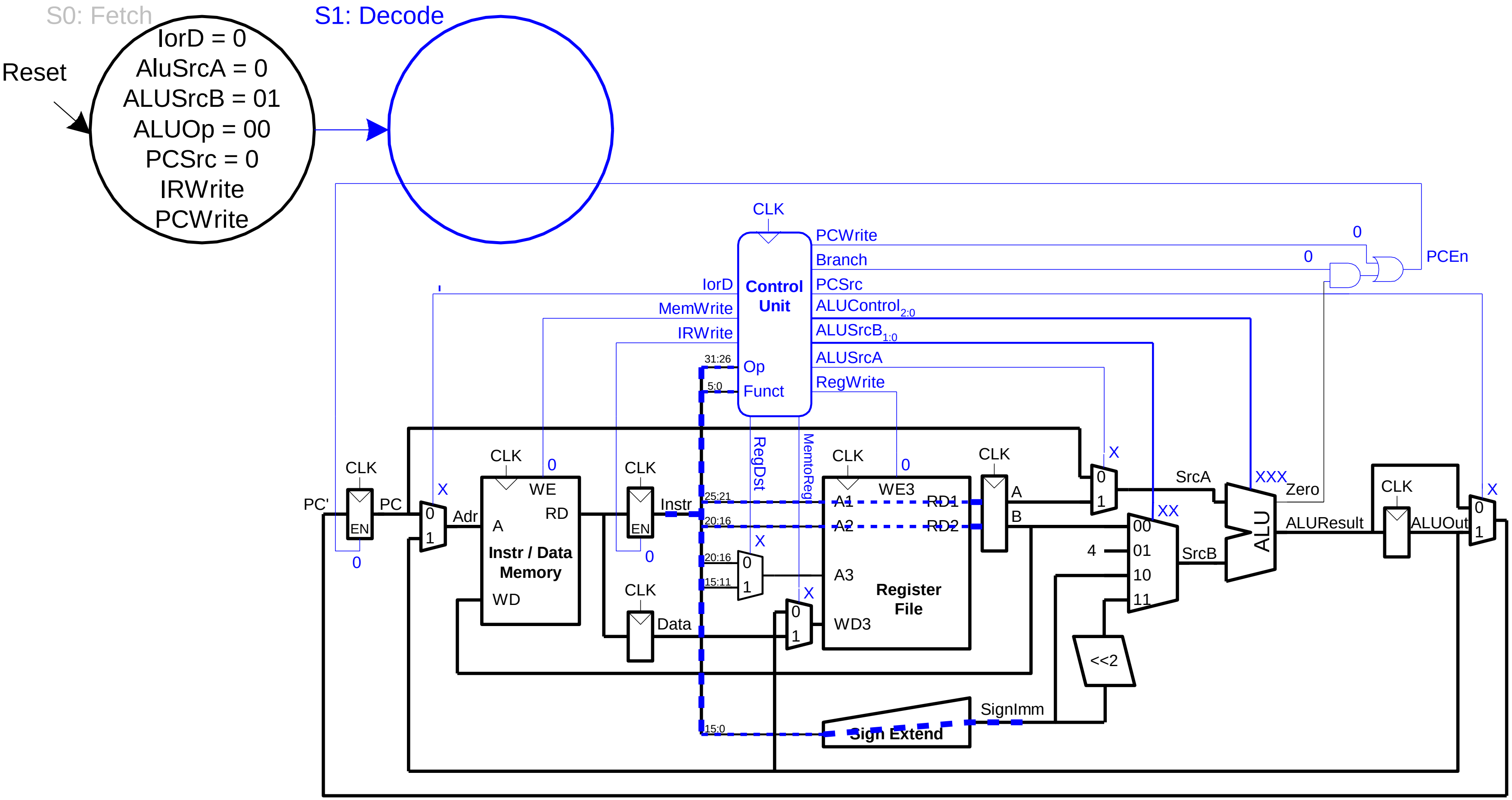
S0: Fetch

Reset

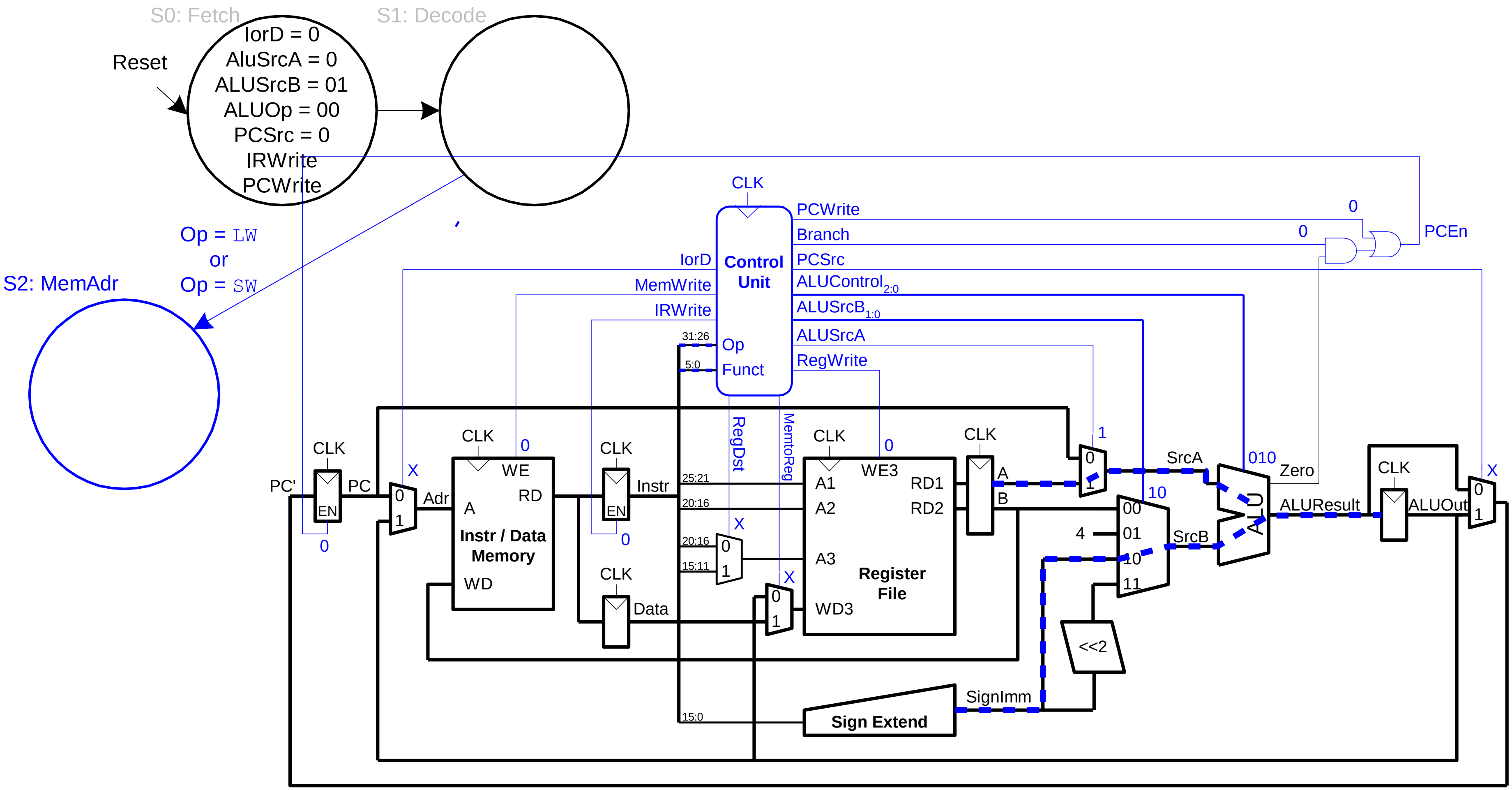
IorD = 0
AluSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCSrc = 0
IRWrite
PCWrite



# Main Controller FSM: Decode



# Main Controller FSM: Address Calculation



S0: Fetch

Reset

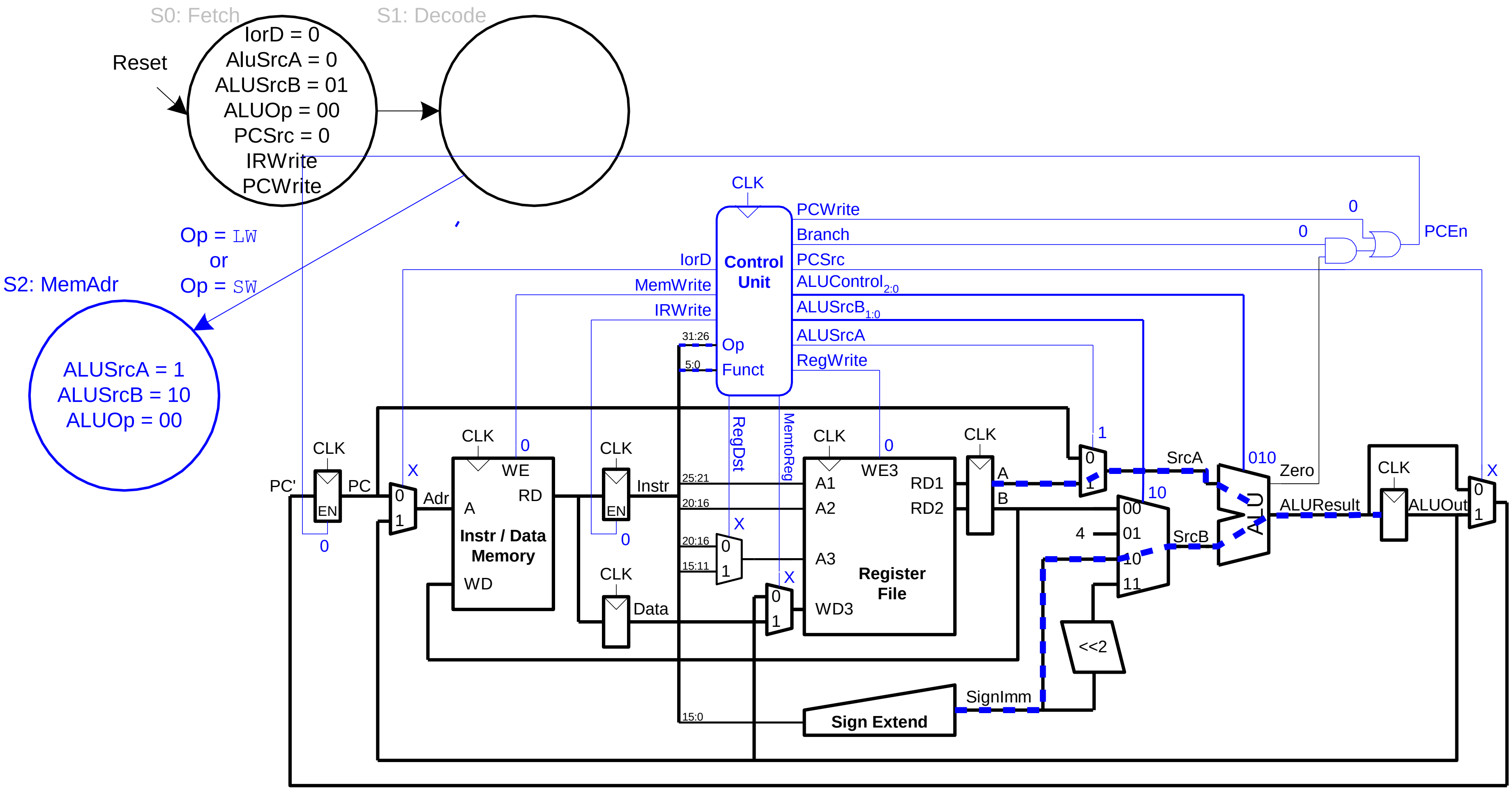
IorD = 0
AluSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCSrc = 0
IRWrite
PCWrite

S1: Decode

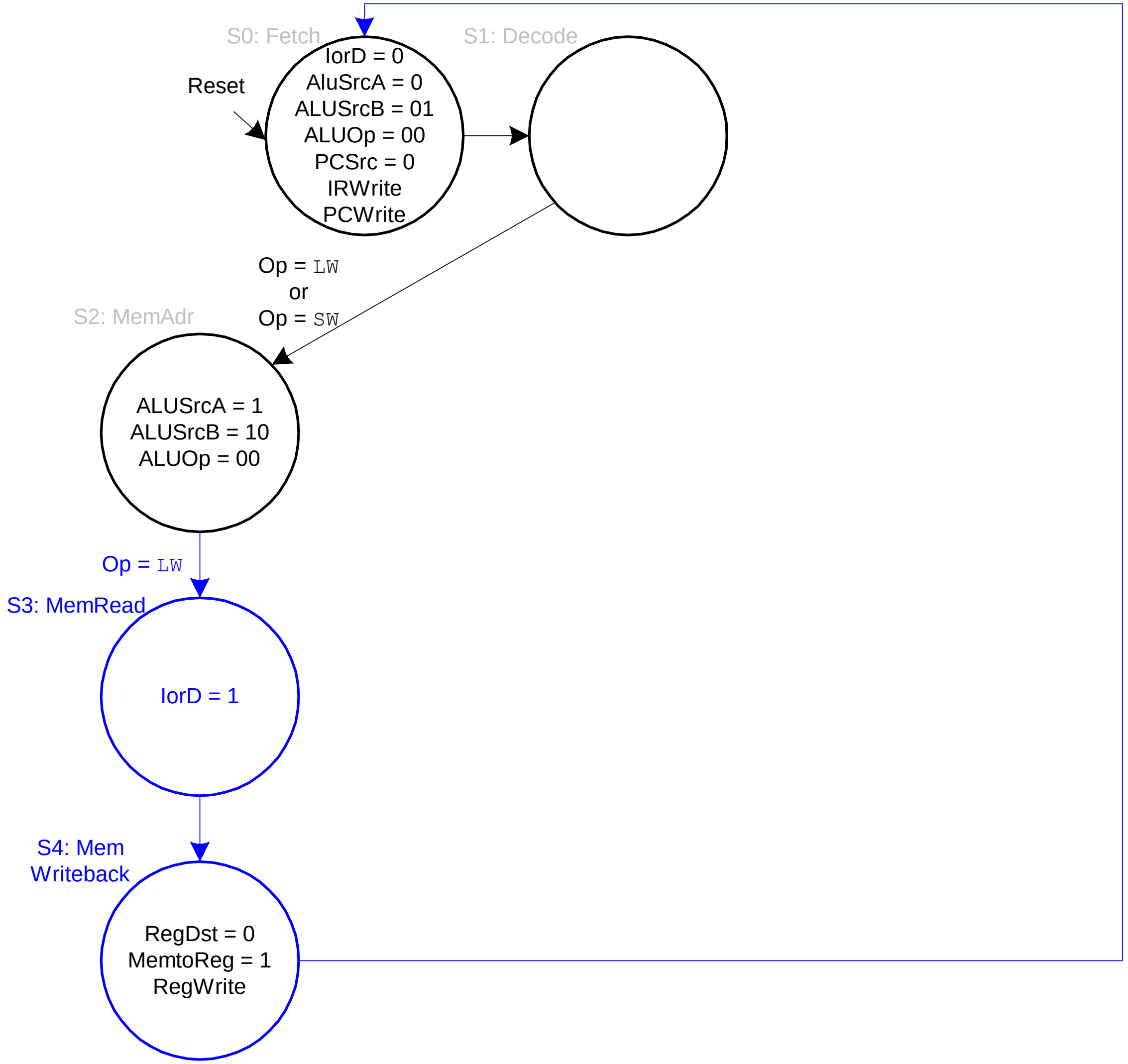
Op = LW
or
Op = SW

S2: MemAdr

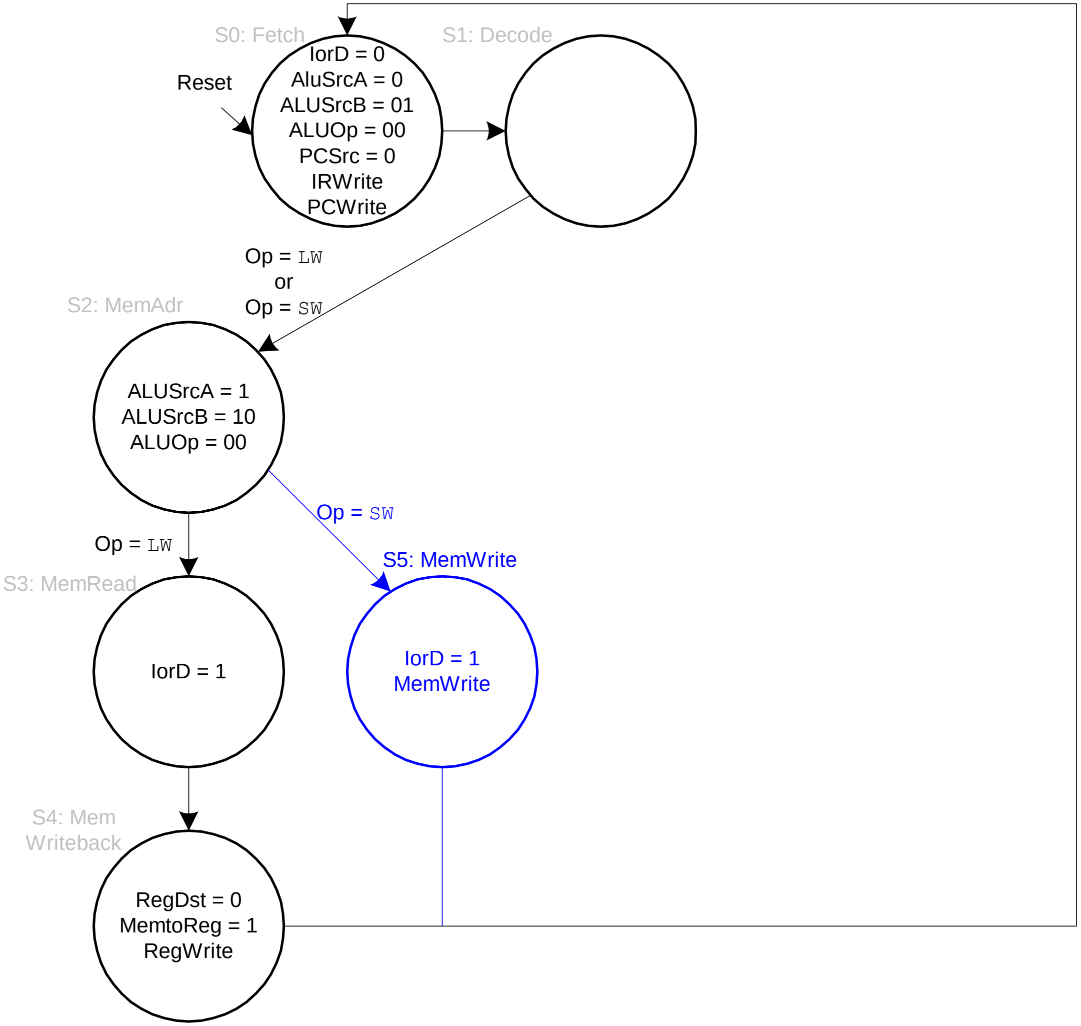
# Main Controller FSM: Address Calculation



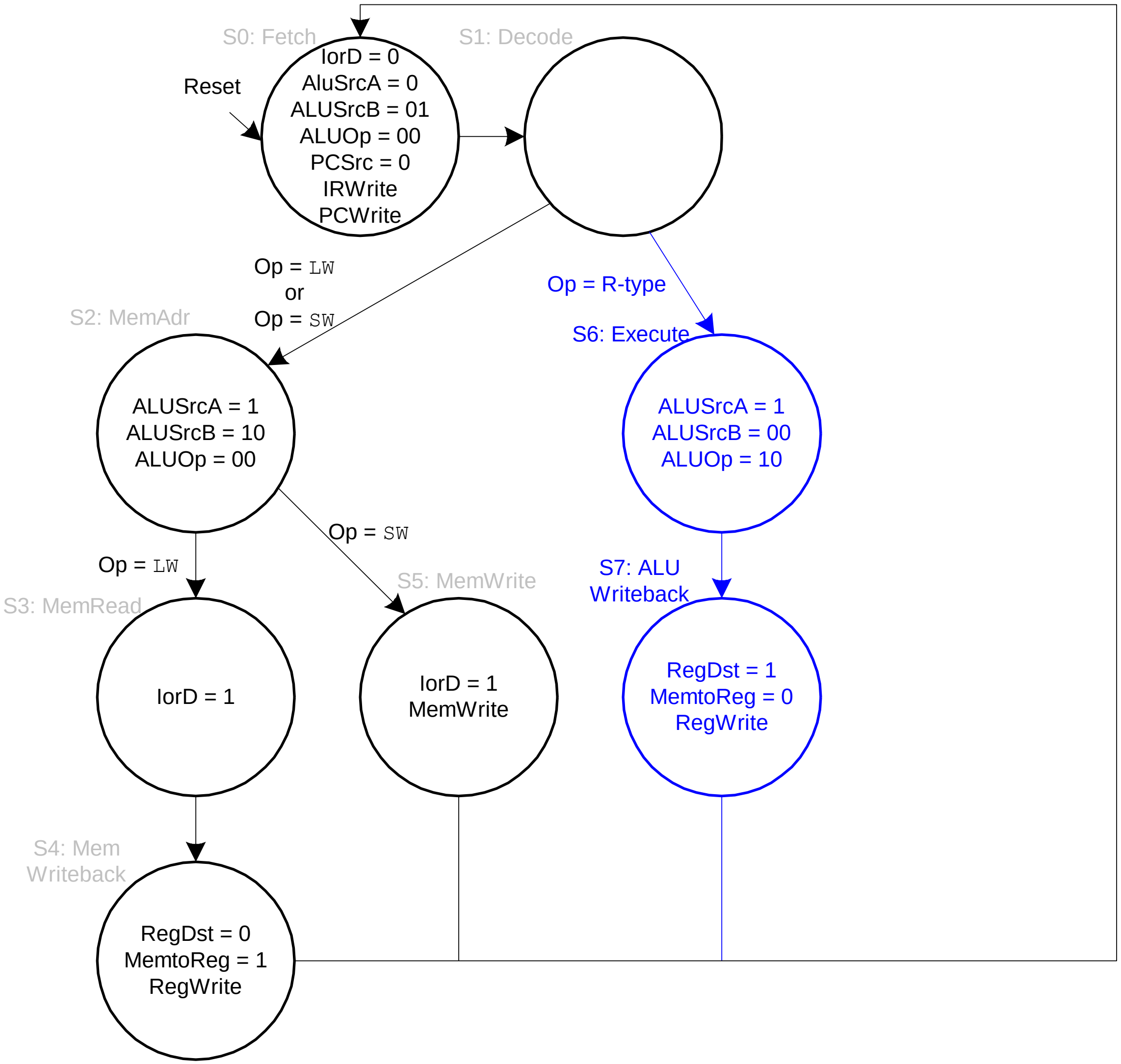
# Main Controller FSM: `lw`



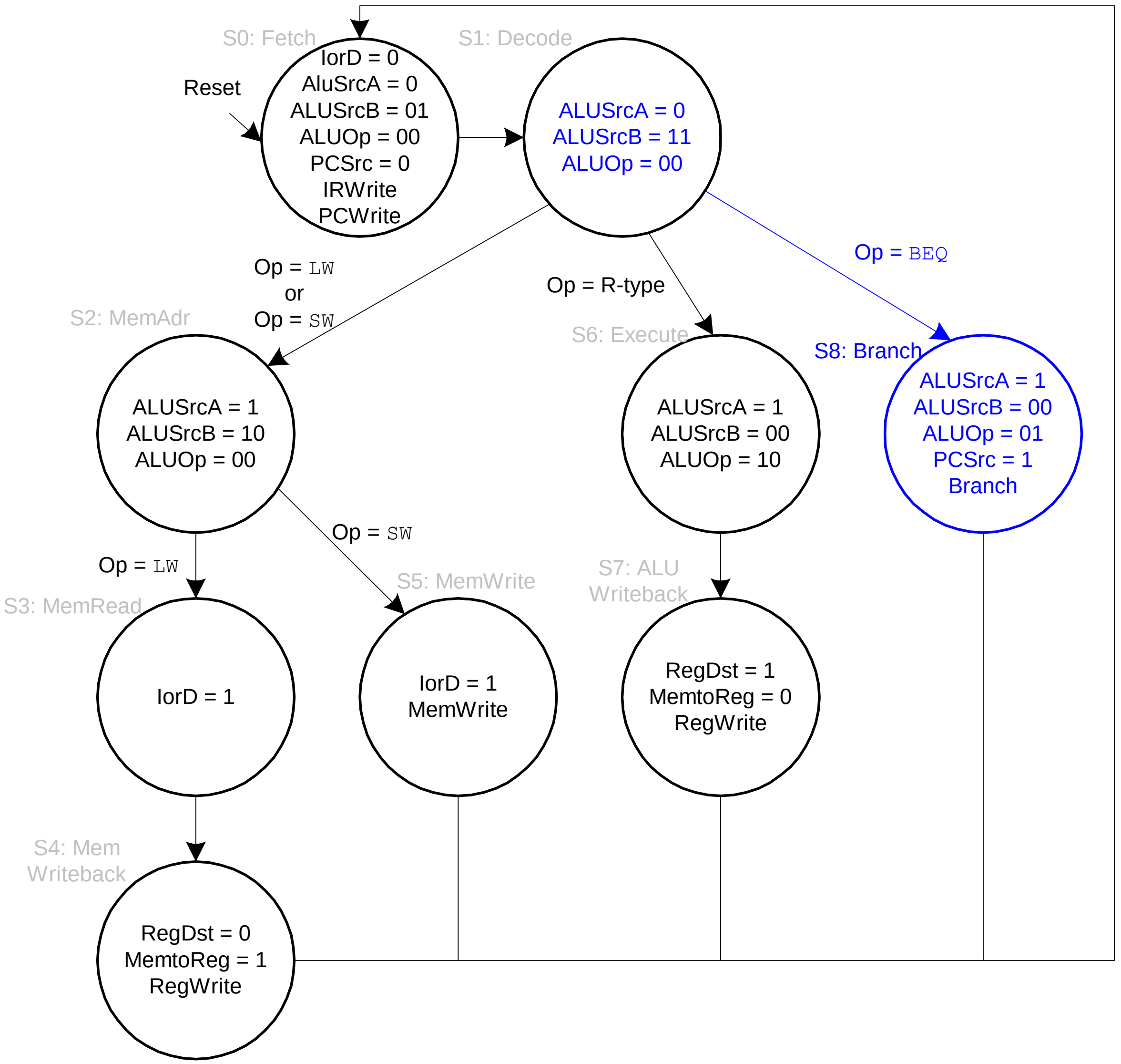
# Main Controller FSM: `SW`



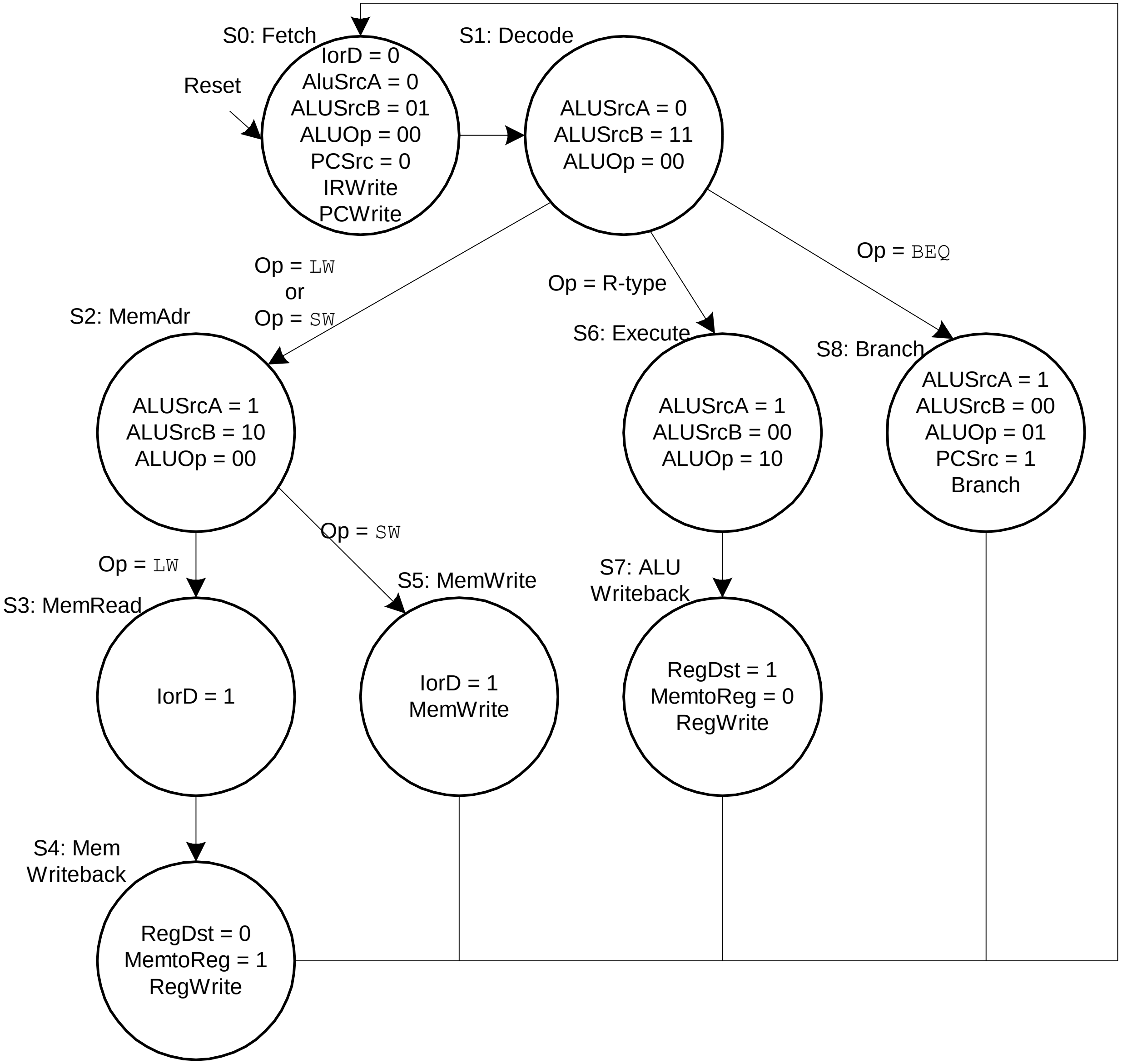
# Main Controller FSM: R-Type



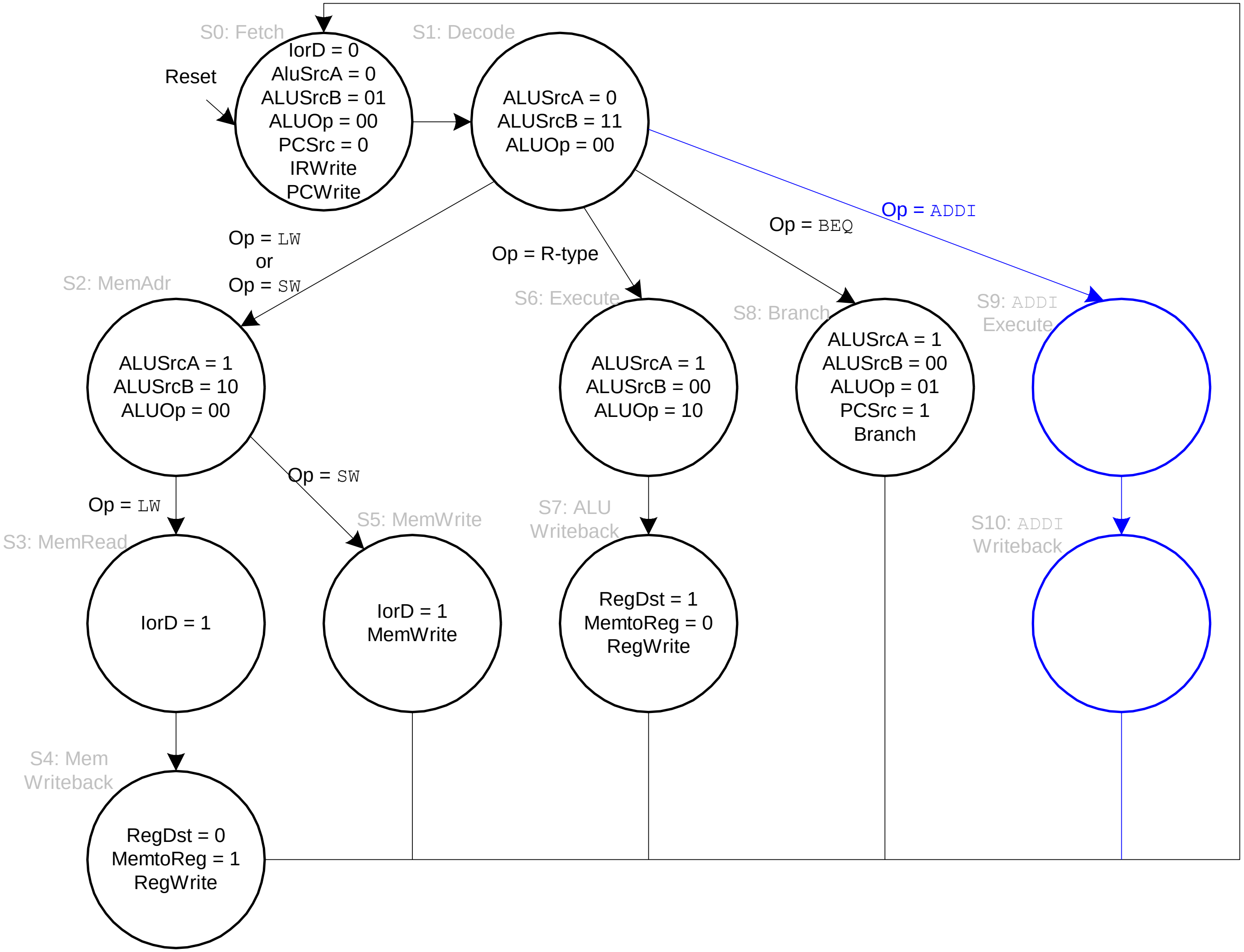
# Main Controller FSM: beq



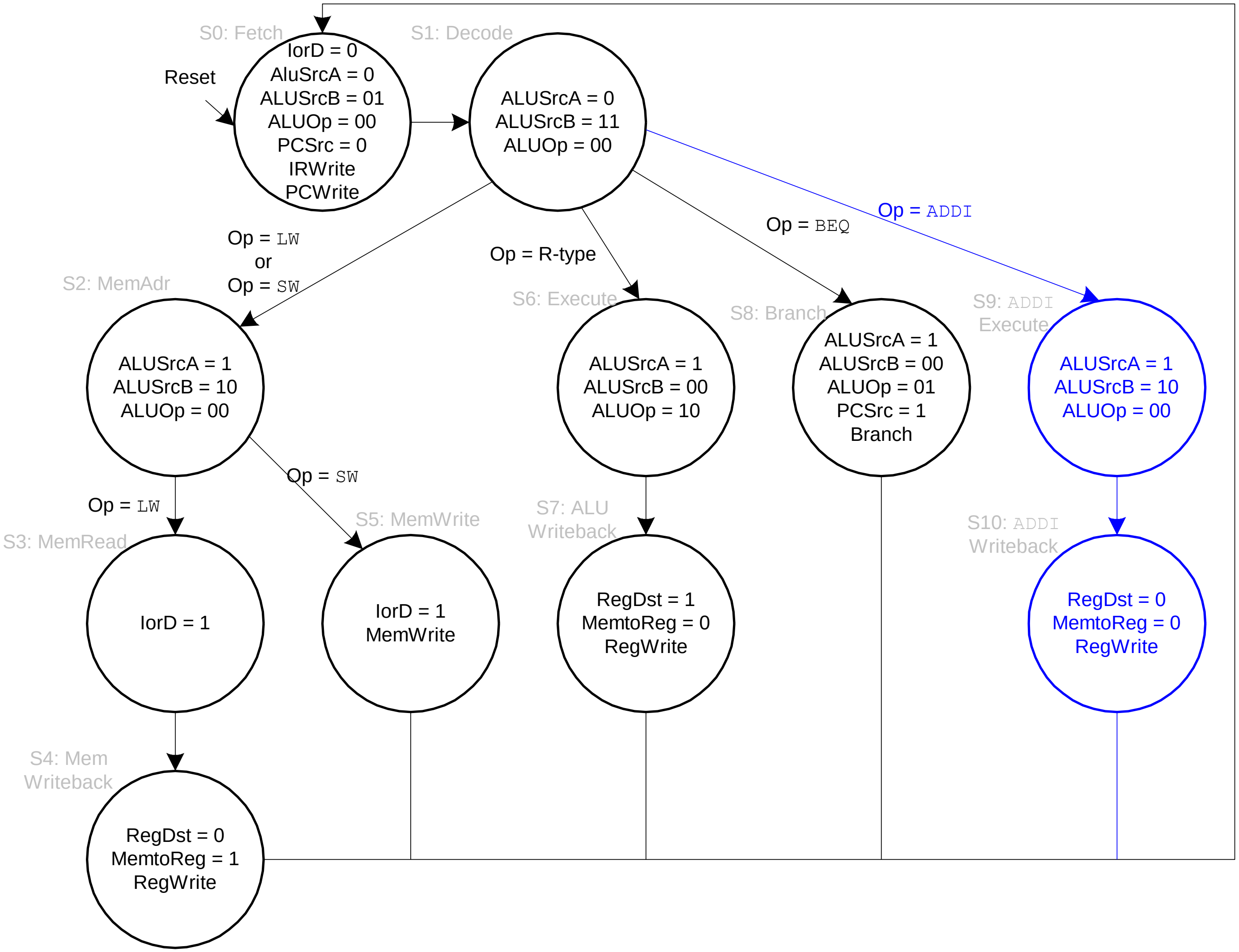
# Complete Multicycle Controller FSM



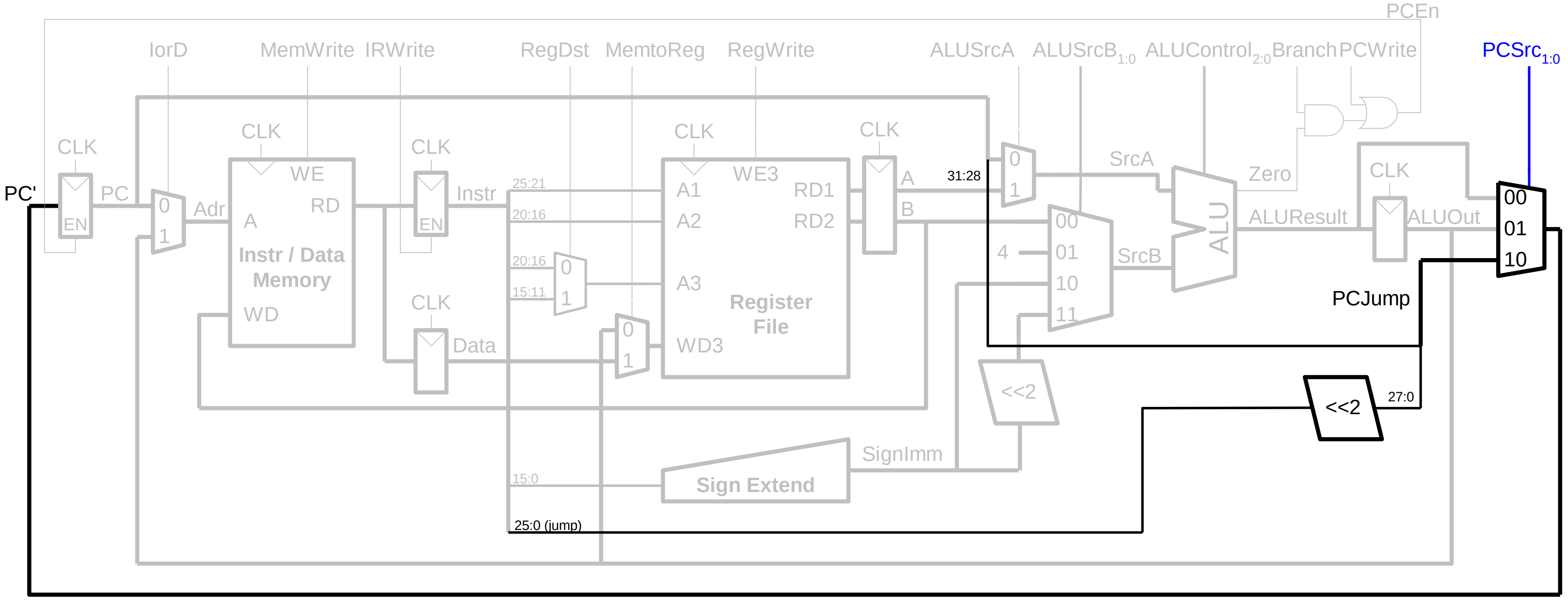
# Main Controller FSM: addi



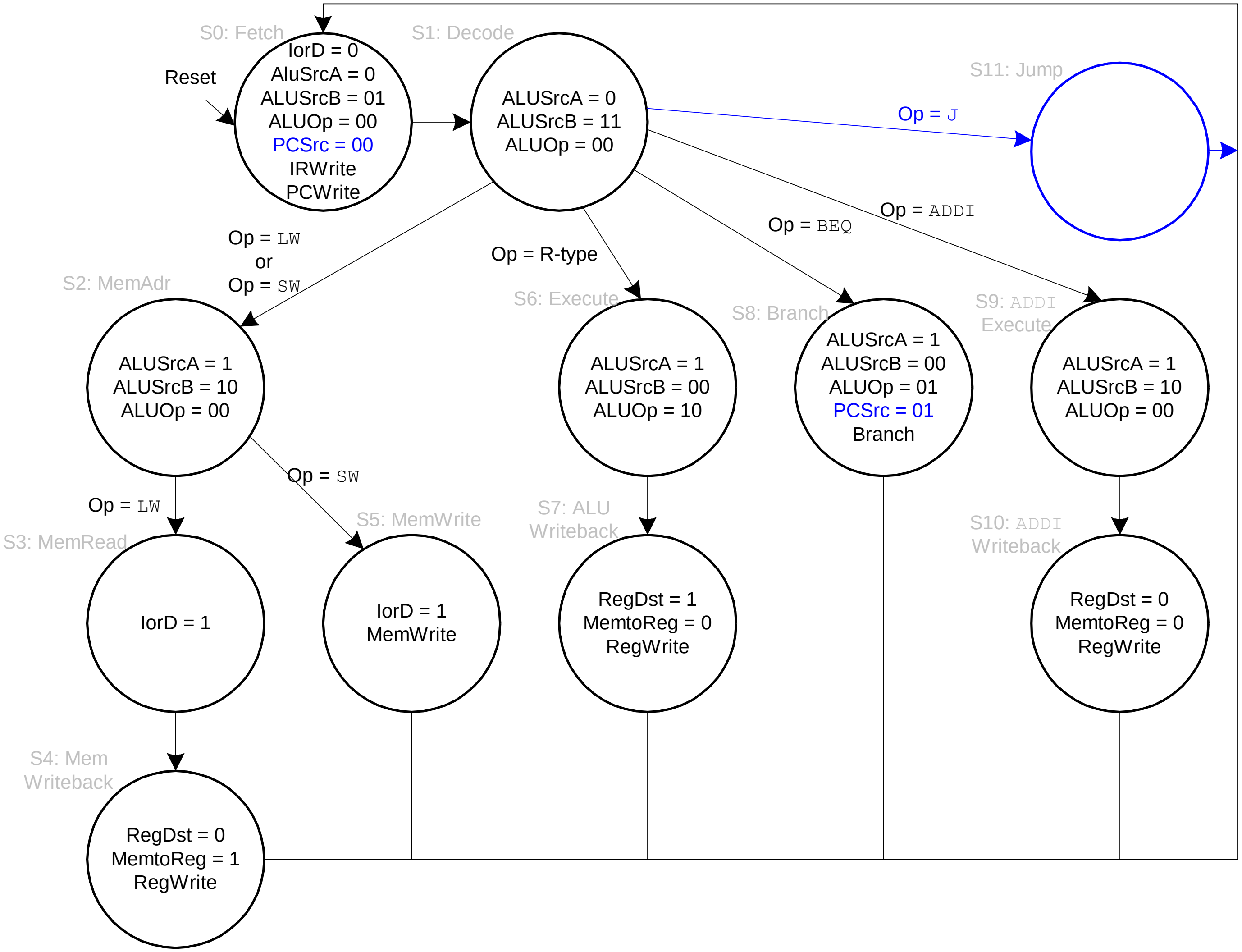
# Main Controller FSM: addi



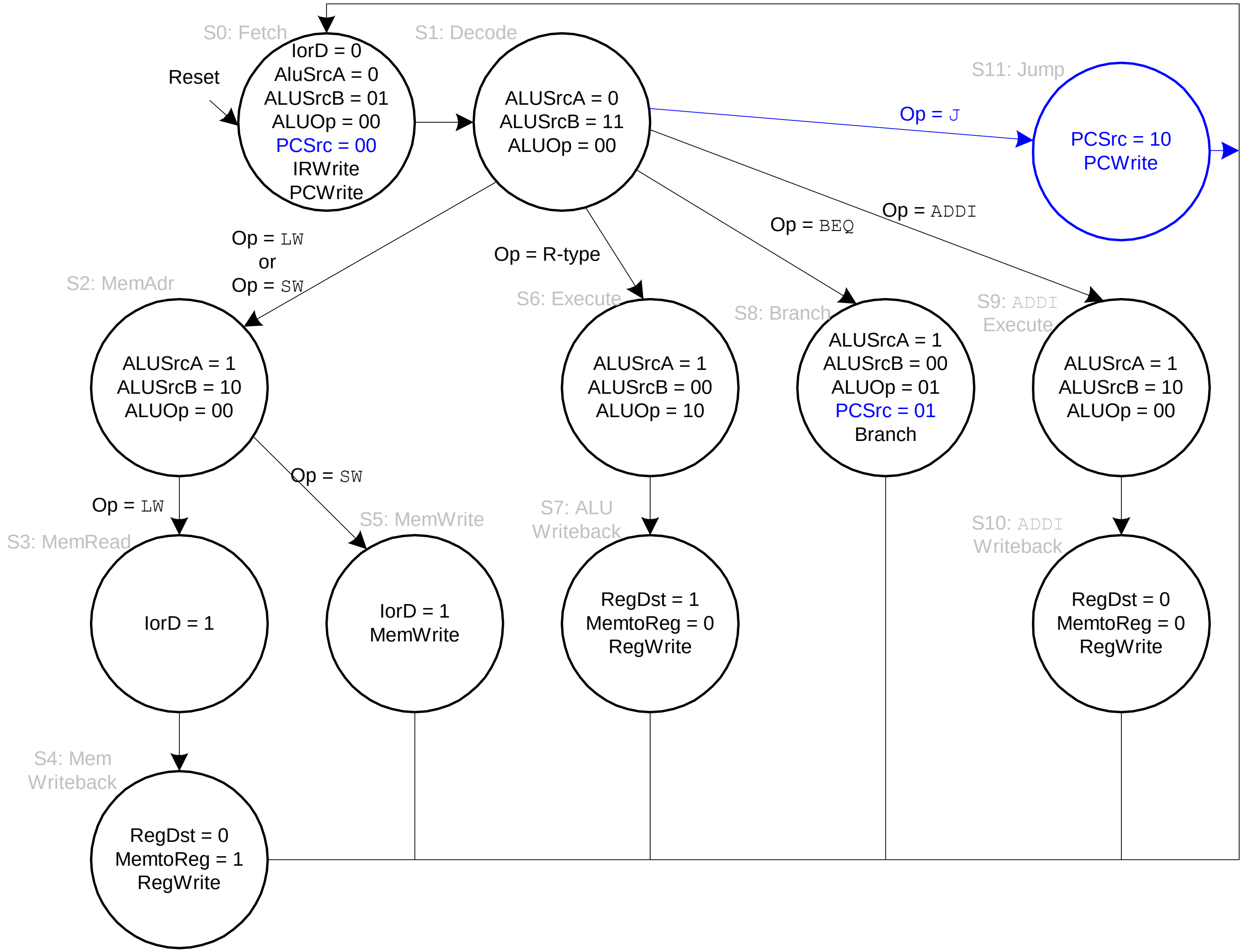
# Extended Functionality: `j`



# Control FSM: `j`



# Control FSM: j



# Multicycle Performance

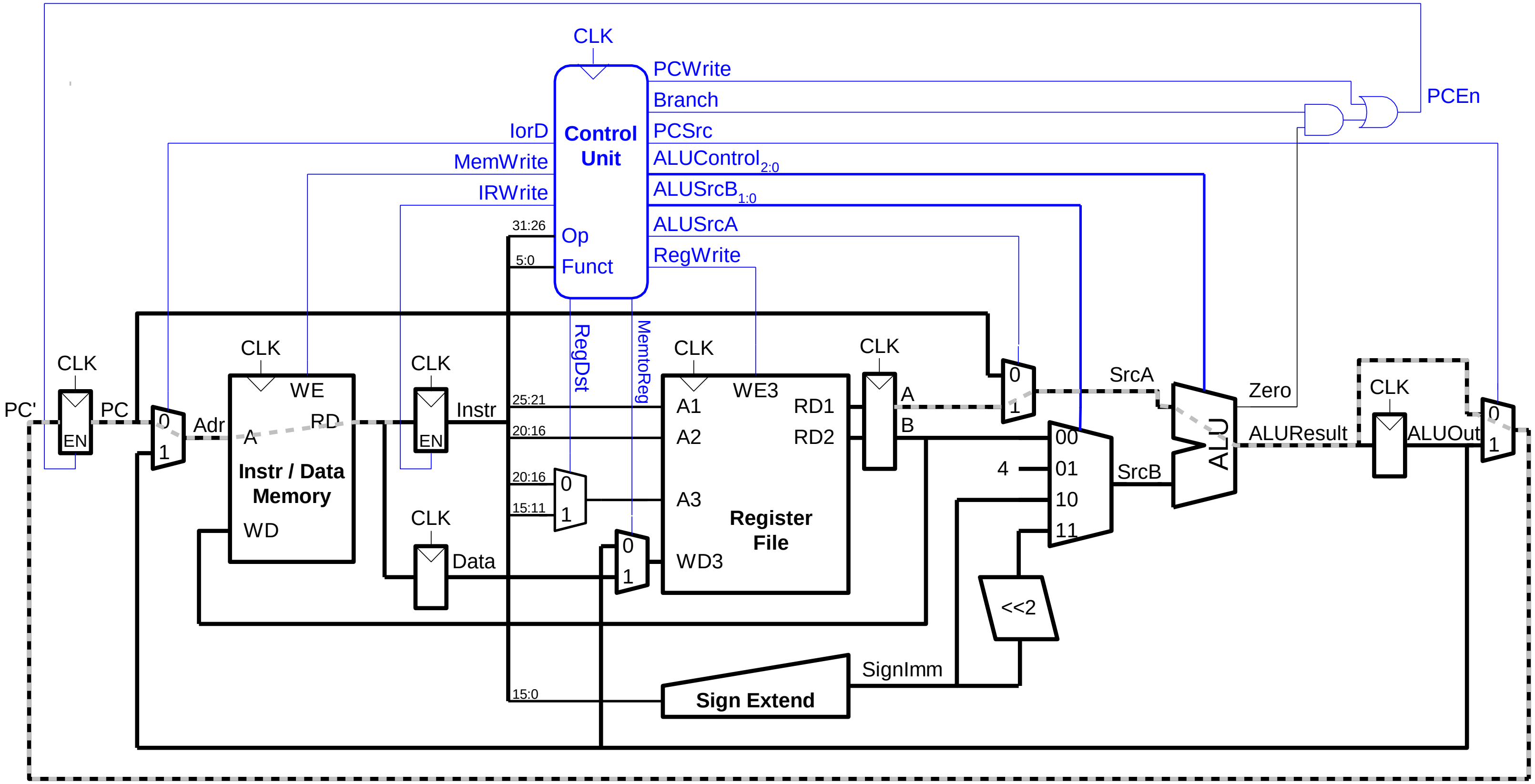
- Instructions take different number of cycles:
  - 3 cycles: `beq`, `j`
  - 4 cycles: R-Type, `sw`, `addi`
  - 5 cycles: `lw`
- CPI is weighted average
- SPECINT2000 benchmark:
  - 25% loads
  - 10% stores
  - 11% branches
  - 2% jumps
  - 52% R-type

**Average CPI = (0.11 + 0.2)(3) + (0.52 + 0.10)(4) + (0.25)(5) = 4.12**

# Multicycle Performance

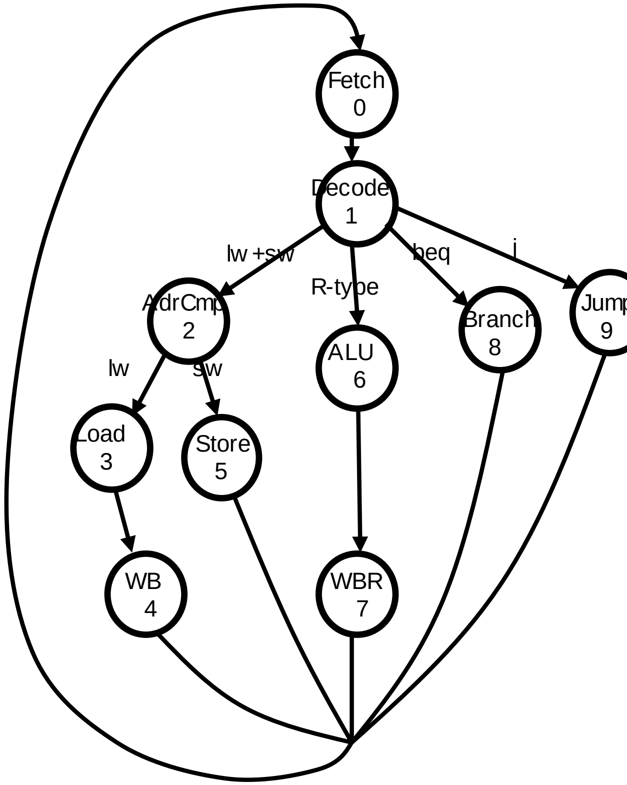
- Multicycle critical path:

$T_c =$



# The state machine

| | opcode | | | | | | current state | | | | next state | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | IR31 | IR30 | IR29 | IR28 | IR27 | IR26 | S3 | S2 | S1 | S0 | S3 | S2 | S1 | S0 |
| | X | X | X | X | X | X | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| R-type | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| lw+sw | 1 | X | X | X | X | X | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| lw | X | X | 0 | X | X | X | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| sw | X | X | 1 | X | X | X | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| | | | | | | | | | | | | | | |

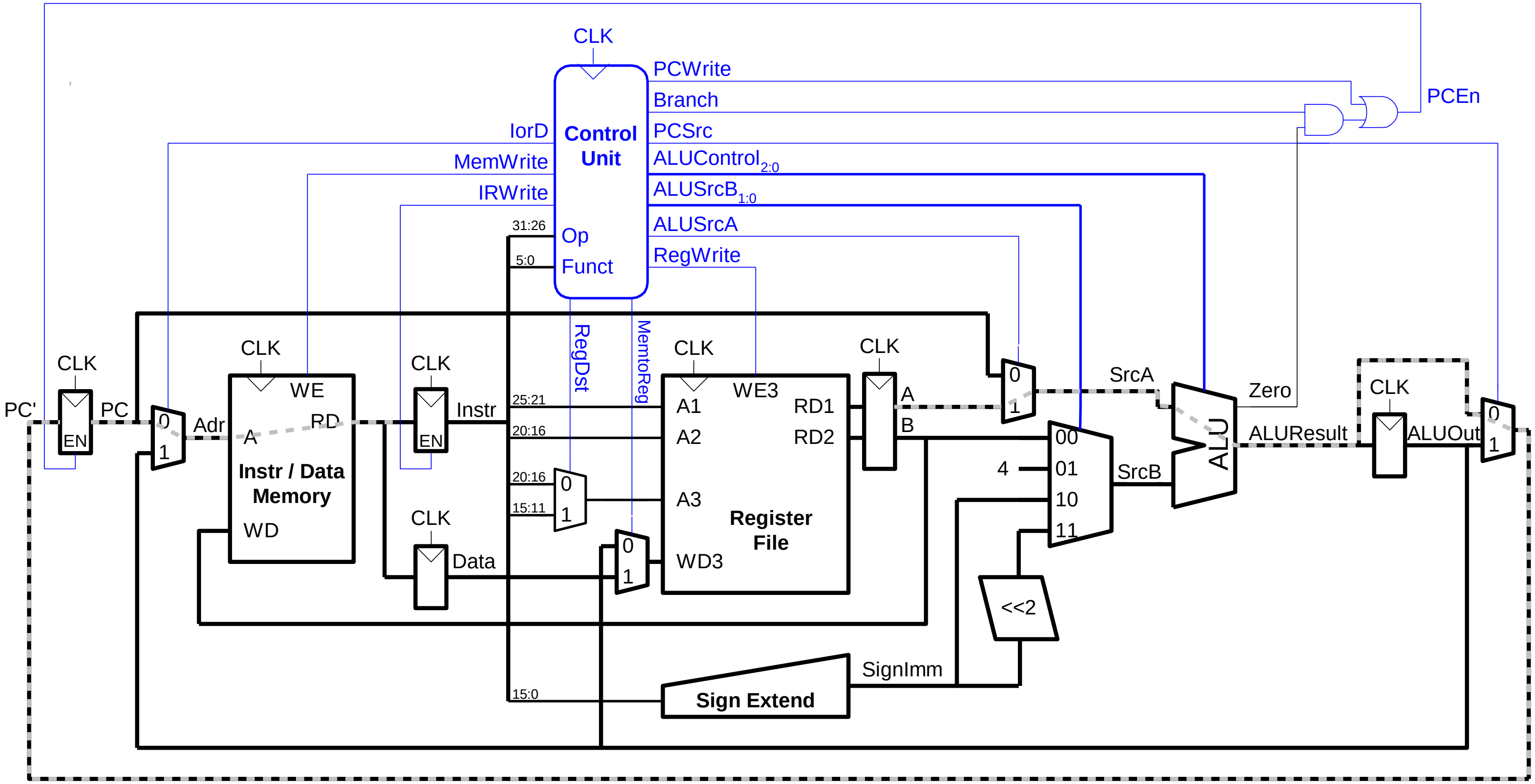


R-type=000000, lw=100011, sw=101011, beq=000100, bne=000101, lui=001111, j=0000010, jal=000011, addi=001000

# Multicycle Performance

- Multicycle critical path:

$$T_c = t_{pcq} + t_{mux} + \max(t_{ALU} + t_{mux}, t_{mem}) + t_{setup}$$



# Multicycle Performance Example

| Element | Parameter | Delay (ps) |
|---|---|---|
| Register clock-to-Q | $t_{pcq}$ | 30 |
| Register setup | $t_{\text{setup}}$ | 20 |
| Multiplexer | $t_{\text{mux}}$ | 25 |
| ALU | $t_{\text{ALU}}$ | 200 |
| Memory read | $t_{\text{mem}}$ | 250 |
| Register file read | $t_{RF\text{read}}$ | 150 |
| Register file setup | $t_{RF\text{setup}}$ | 20 |

$T_c =$

# Multicycle Performance Example

| Element | Parameter | Delay (ps) |
|---|---|---|
| Register clock-to-Q | $t_{pcq\_\mathrm{PC}}$ | 30 |
| Register setup | $t_{\mathrm{setup}}$ | 20 |
| Multiplexer | $t_{\mathrm{mux}}$ | 25 |
| ALU | $t_{\mathrm{ALU}}$ | 200 |
| Memory read | $t_{\mathrm{mem}}$ | 250 |
| Register file read | $t_{RF\mathrm{read}}$ | 150 |
| Register file setup | $t_{RF\mathrm{setup}}$ | 20 |

$$\begin{aligned}
T_c &= t_{pcq\_PC} + t_{\mathrm{mux}} + \max(t_{\mathrm{ALU}} + t_{\mathrm{mux}}, t_{\mathrm{mem}}) + t_{\mathrm{setup}} \\
&= t_{pcq\_PC} + t_{\mathrm{mux}} + t_{\mathrm{mem}} + t_{\mathrm{setup}} \\
&= [30 + 25 + 250 + 20] \text{ ps} \\
&= 325 \text{ ps}
\end{aligned}$$

# Multicycle Performance Example

- For a program with 100 billion instructions executing on a multicycle MIPS processor
  - CPI = 4.12
  - $T_c$ = 325 ps

Execution Time =

# Multicycle Performance Example

- For a program with 100 billion instructions executing on a multicycle MIPS processor
  - CPI = 4.12
  - $T_c$ = 325 ps

$$\begin{aligned}\text{Execution Time} &= (\#\text{ instructions}) \times \text{CPI} \times T_c \\ &= (100 \times 10^{9})(4.12)(325 \times 10^{-12}) \\ &= 133.9 \text{ seconds}\end{aligned}$$

- This is slower than the single-cycle processor (92.5 seconds). Why?

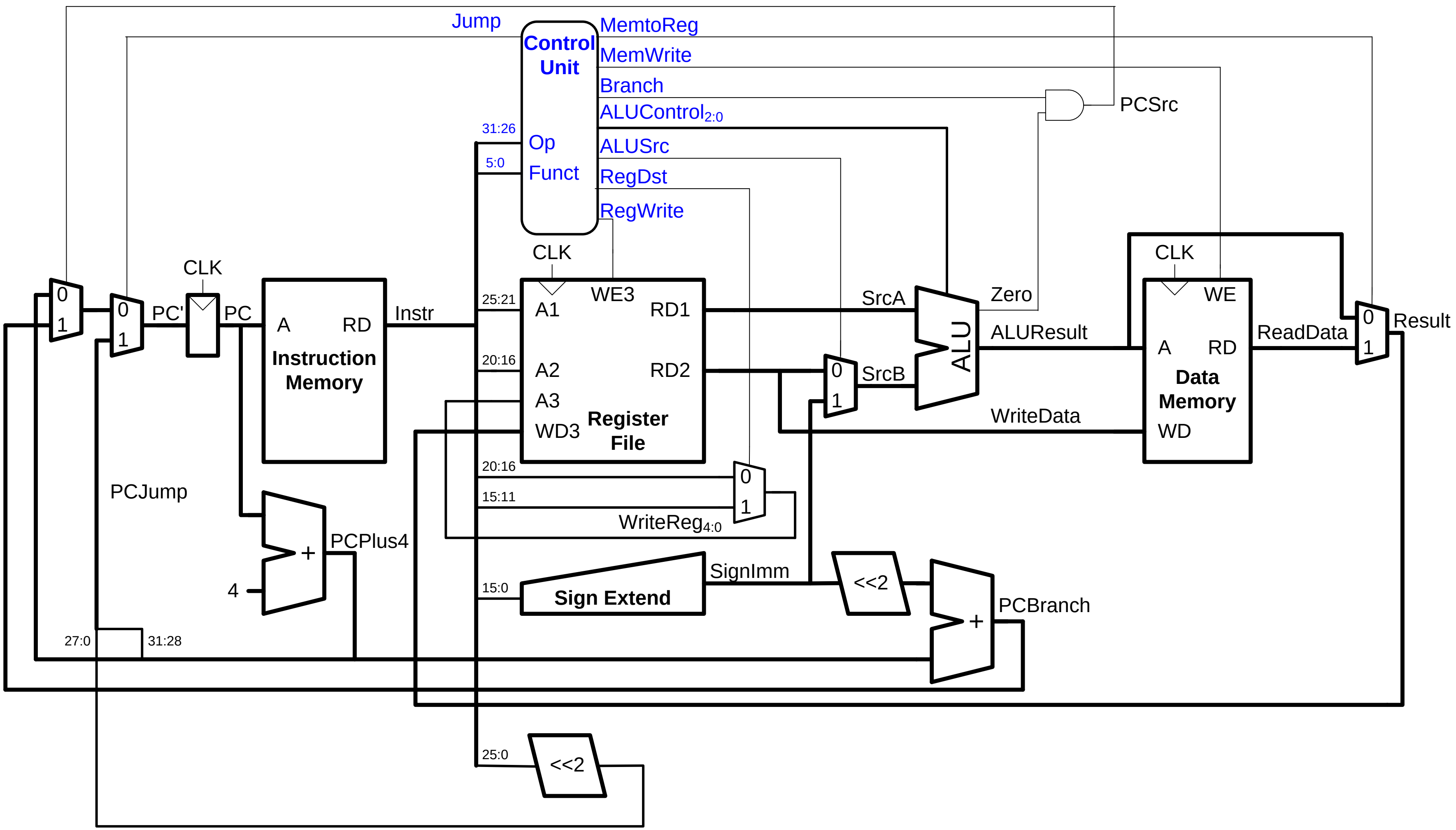
# Multicycle Performance Example

- For a program with 100 billion instructions executing on a multicycle MIPS processor
  - CPI = 4.12
  - $T_c$ = 325 ps

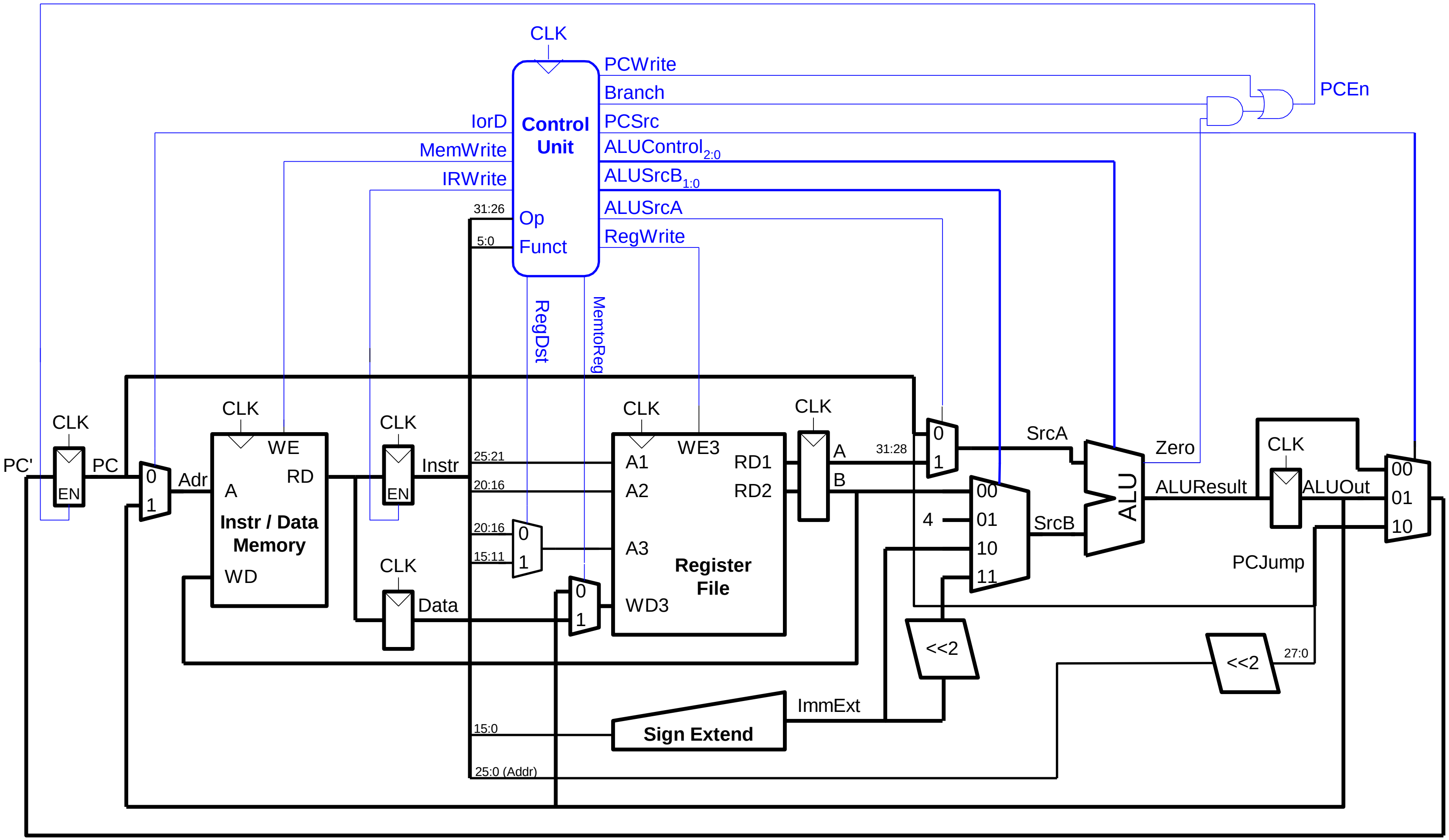
$$
\begin{aligned}
\text{Execution Time} &= (\#\ \text{instructions}) \times \text{CPI} \times T_c \\
&= (100 \times 10^{9})(4.12)(325 \times 10^{-12}) \\
&= 133.9\ \text{seconds}
\end{aligned}
$$

- This is slower than the single-cycle processor (92.5 seconds). Why?
  - Not all steps the same length
  - Sequencing overhead for each step ($t_{pcq} + t_{\text{setup}} = 50$ ps)

# Review: Single-Cycle MIPS Processor



# Review: Multicycle MIPS Processor



# Exception Registers

- Not part of the register file.
  - Cause
    - Records the cause of the exception
    - Coprocessor 0 register 13
  - EPC (Exception PC)
    - Records the PC where the exception occurred
    - Coprocessor 0 register 14
- Move from Coprocessor 0
  - mfc0 $t0, Cause
  - Moves the contents of Cause into $t0

**mfc0**

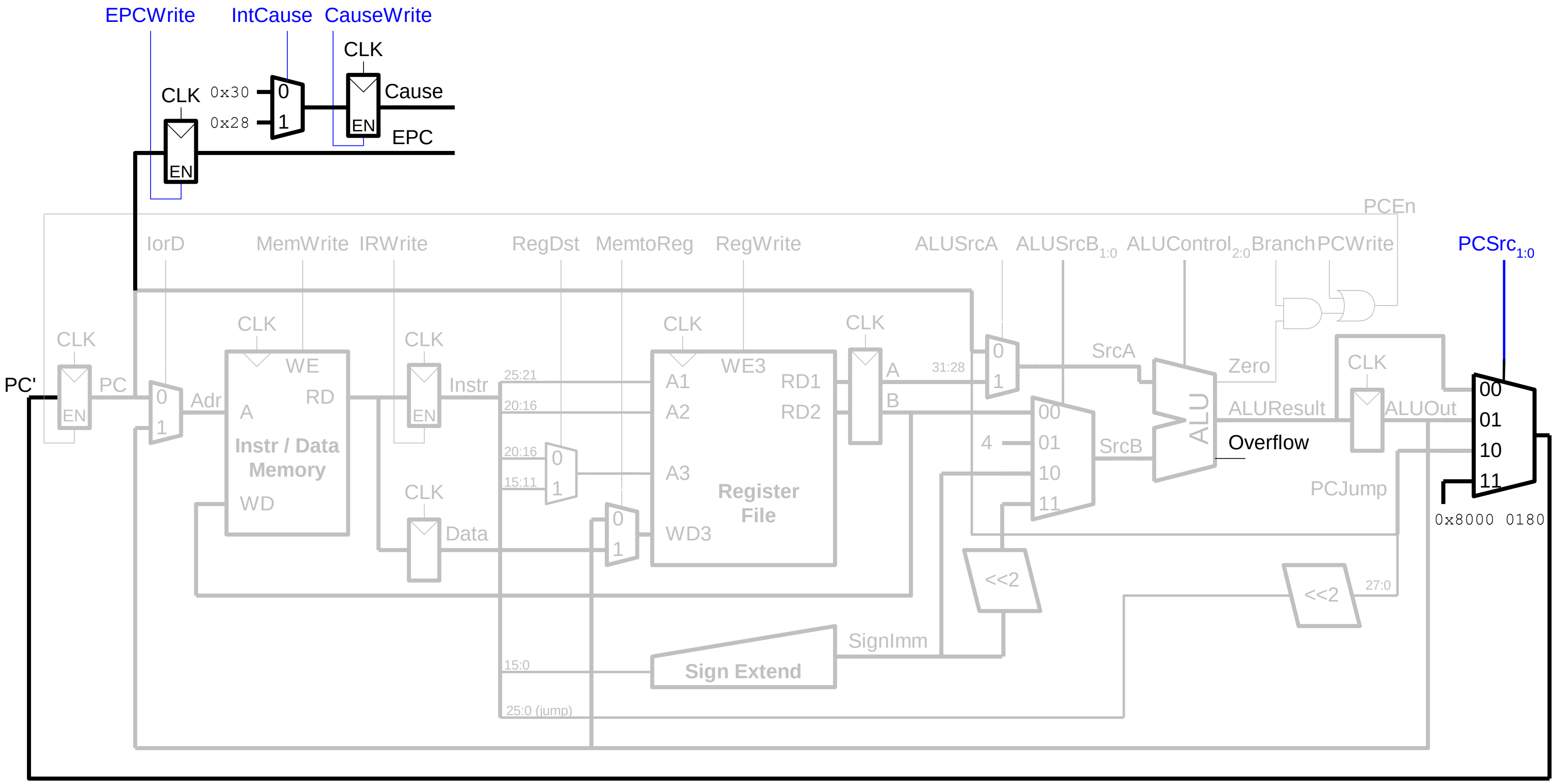
| | | | | |
|---|---|---|---|---|
| 010000 | 00000 | $t0 (8) | Cause (13) | 00000000000 |
| 31:26 | 25:21 | 20:16 | 15:11 | 10:0 |

# Exception Causes

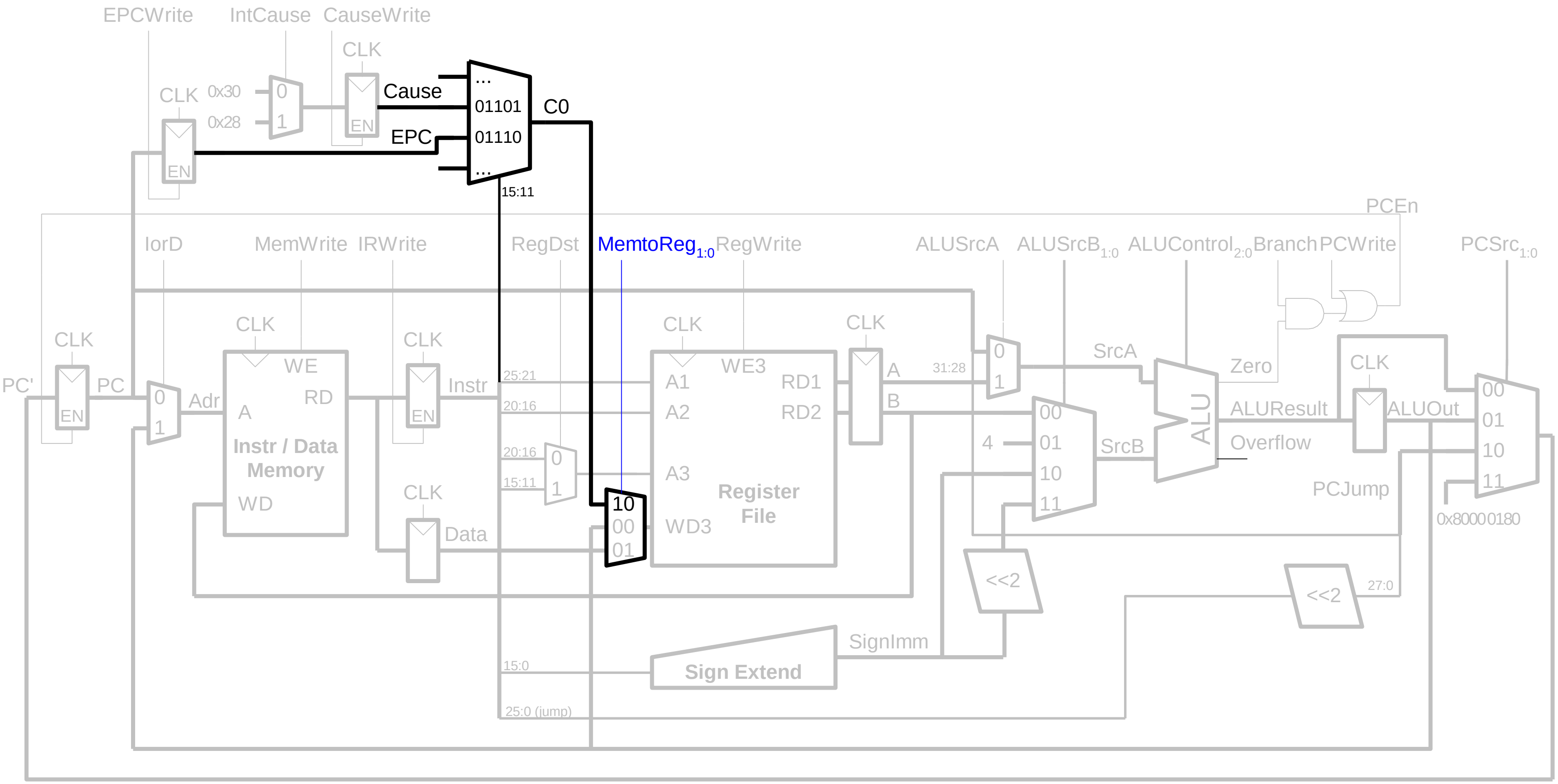
| Exception | Cause |
|---|---|
| Hardware Interrupt | 0x00000000 |
| System Call | 0x00000020 |
| Breakpoint / Divide by 0 | 0x00000024 |
| Undefined Instruction | 0x00000028 |
| Arithmetic Overflow | 0x00000030 |

**We extend the multicycle MIPS processor to handle the last two types of exceptions.**

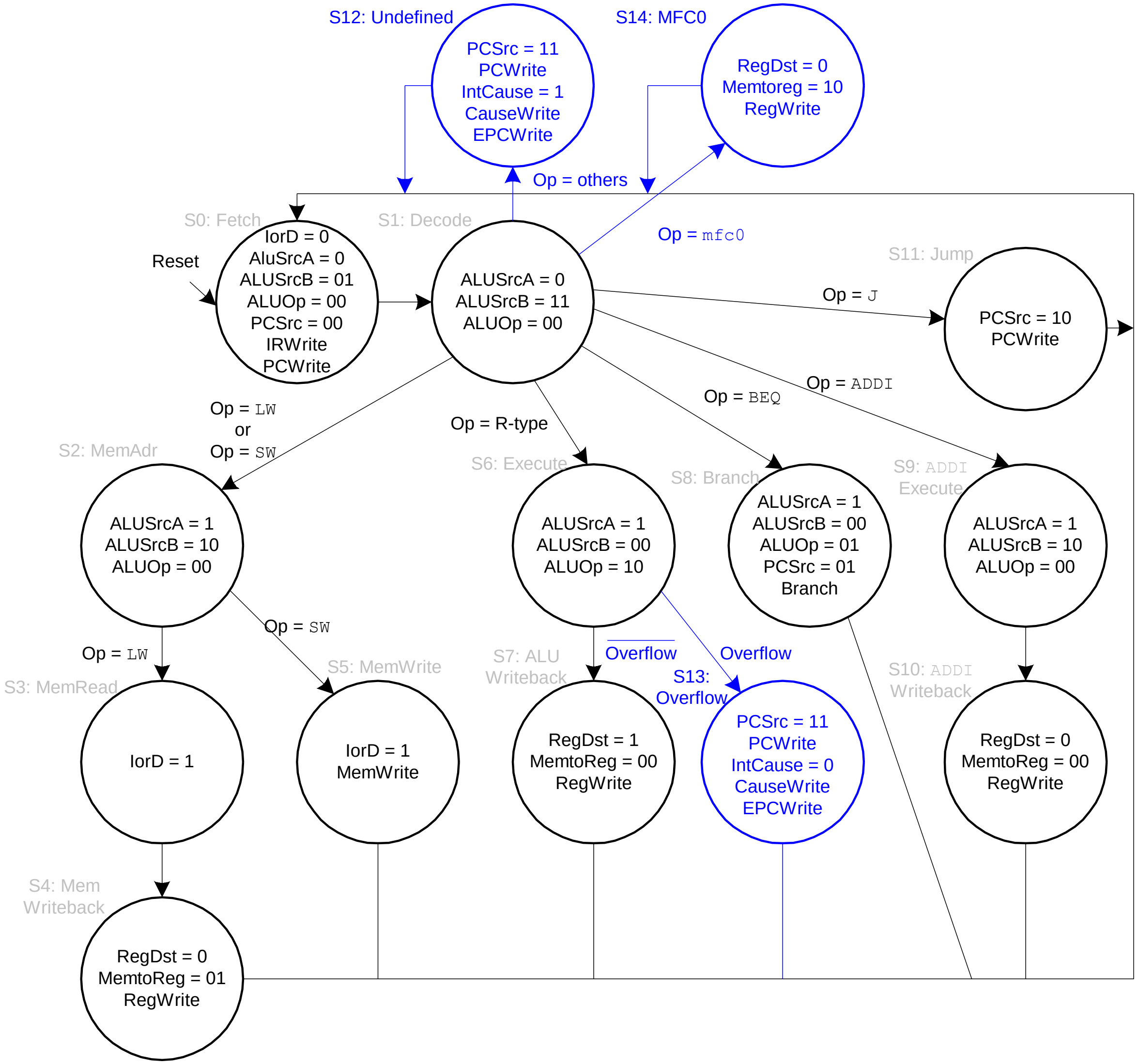
# Exception Hardware: EPC & Cause



# Exception Hardware: `mfc0`



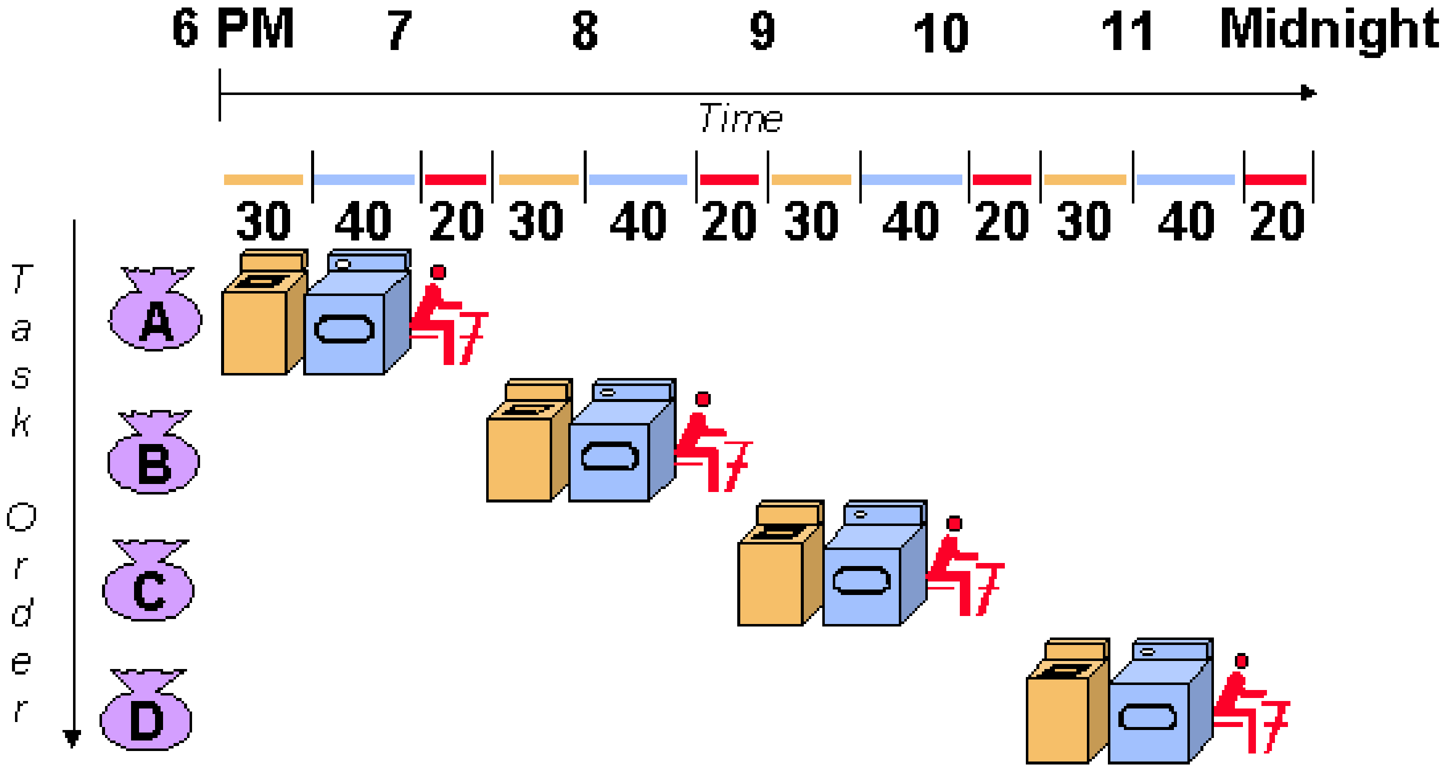
# Control FSM with Exceptions



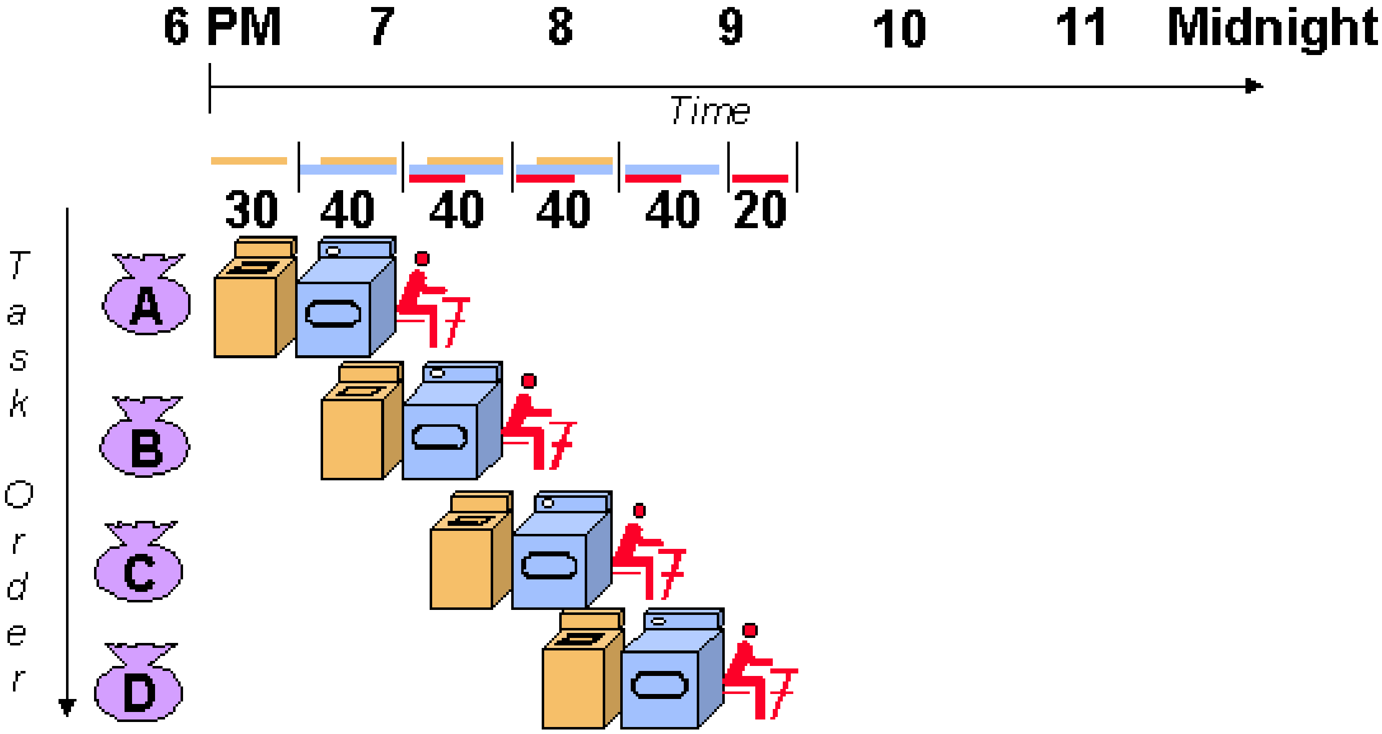
# Pipelined MIPS Processor

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

# How Pipelining Works

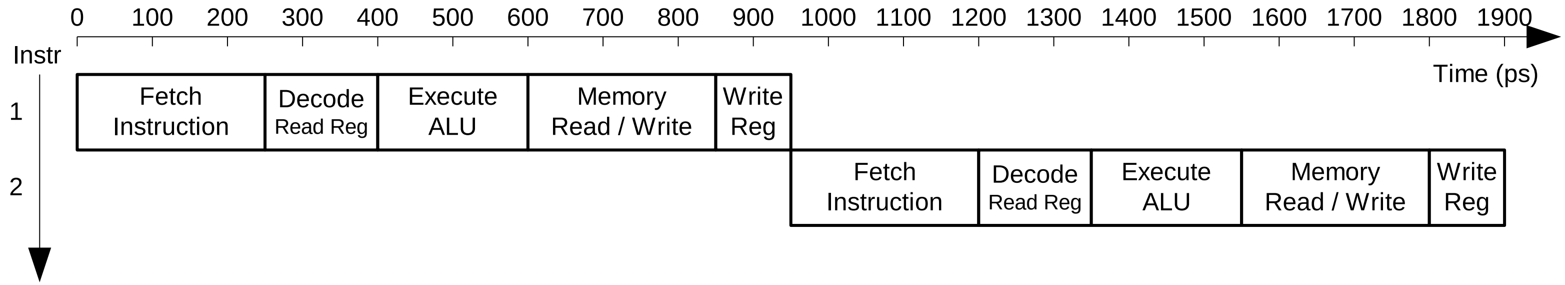


# How Pipelining Works

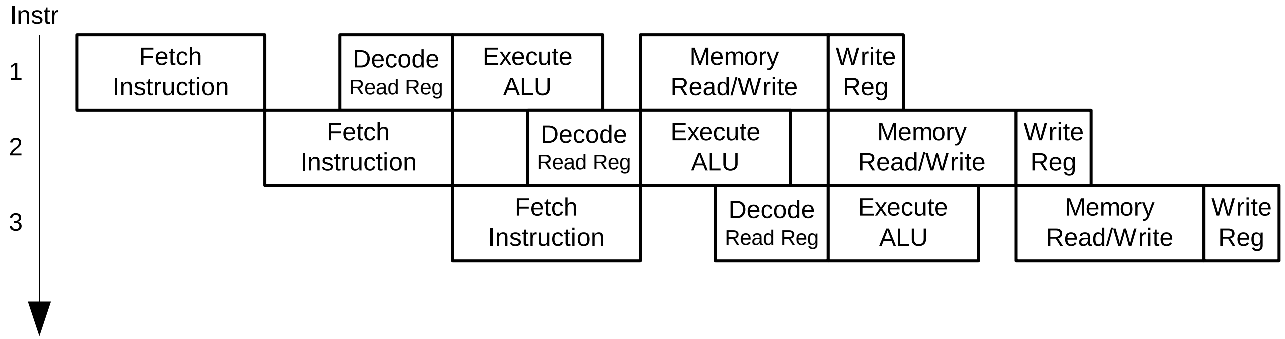


# Single-Cycle vs. Pipelined Performance

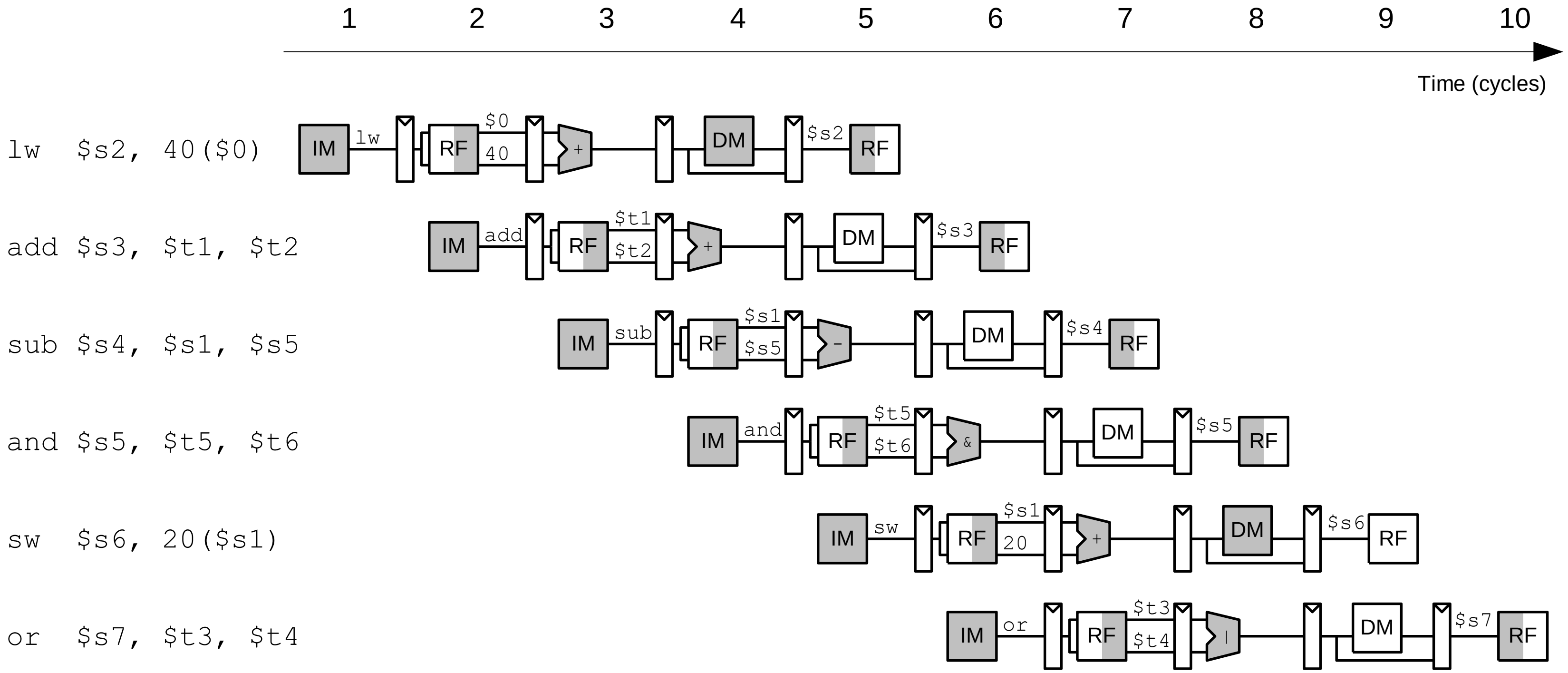
## Single-Cycle



## Pipelined

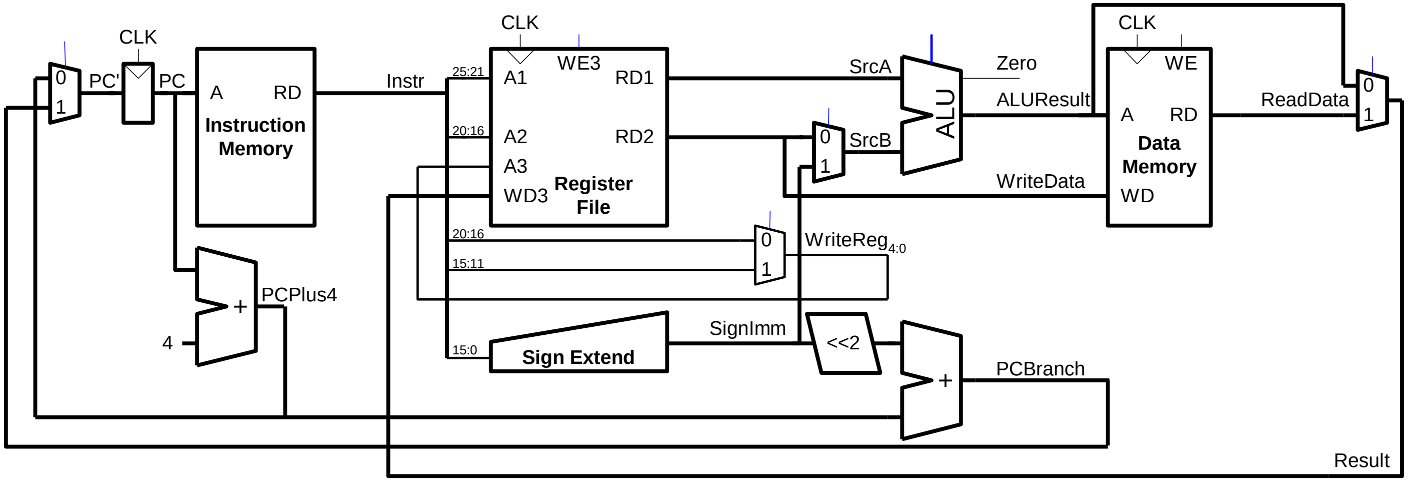


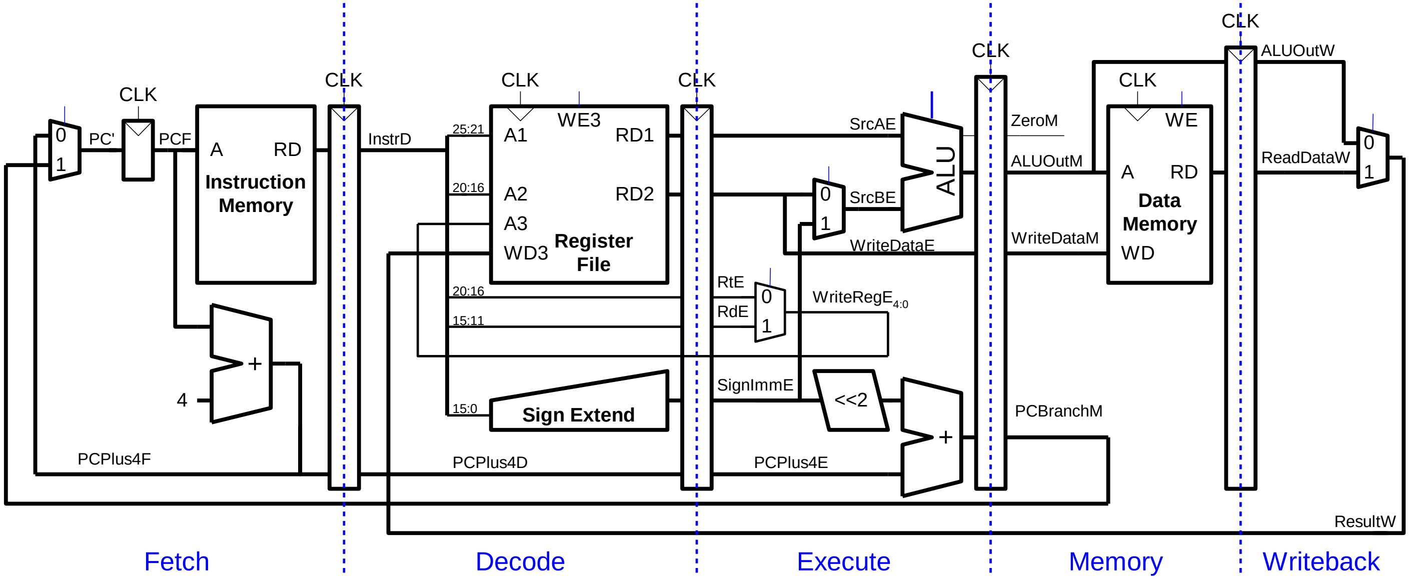
# Pipelining Abstraction



```
lw  $s2, 40($0)
add $s3, $t1, $t2
sub $s4, $s1, $s5
and $s5, $t5, $t6
sw  $s6, 20($s1)
or  $s7, $t3, $t4
```

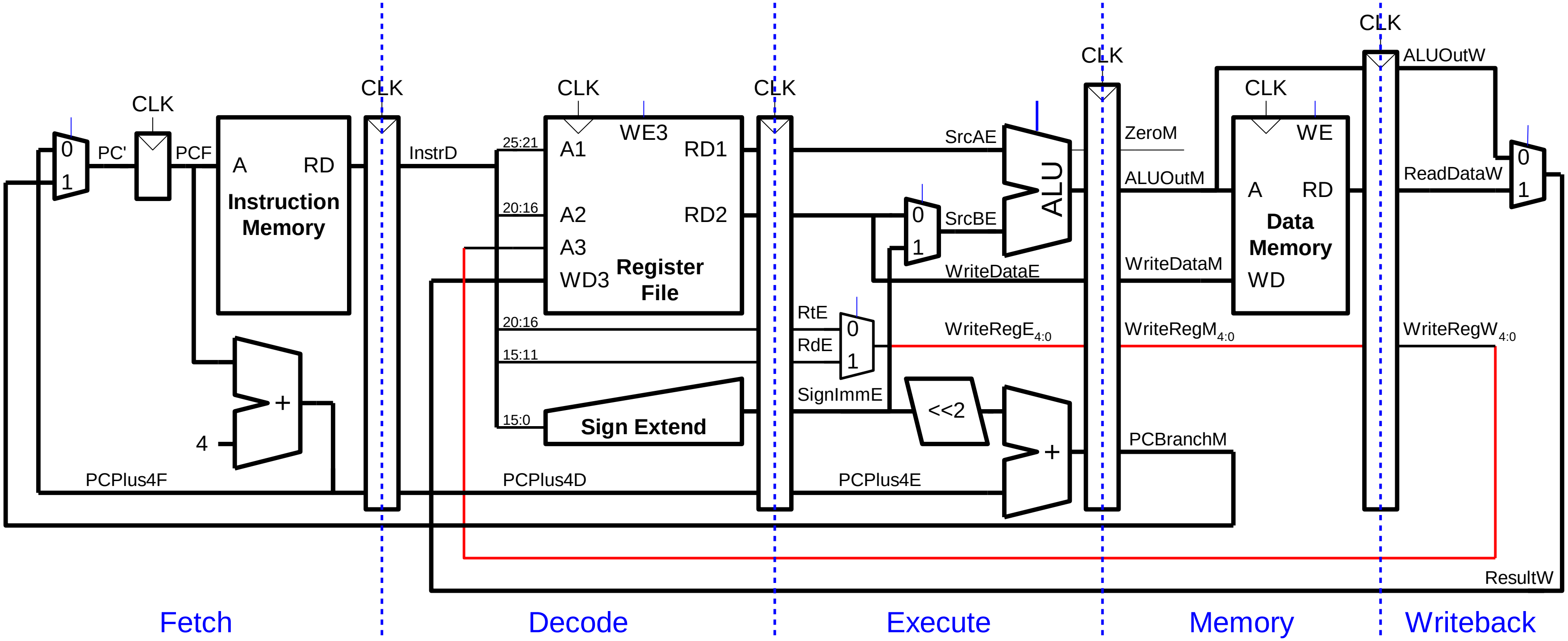
# Single-Cycle and Pipelined Datapath



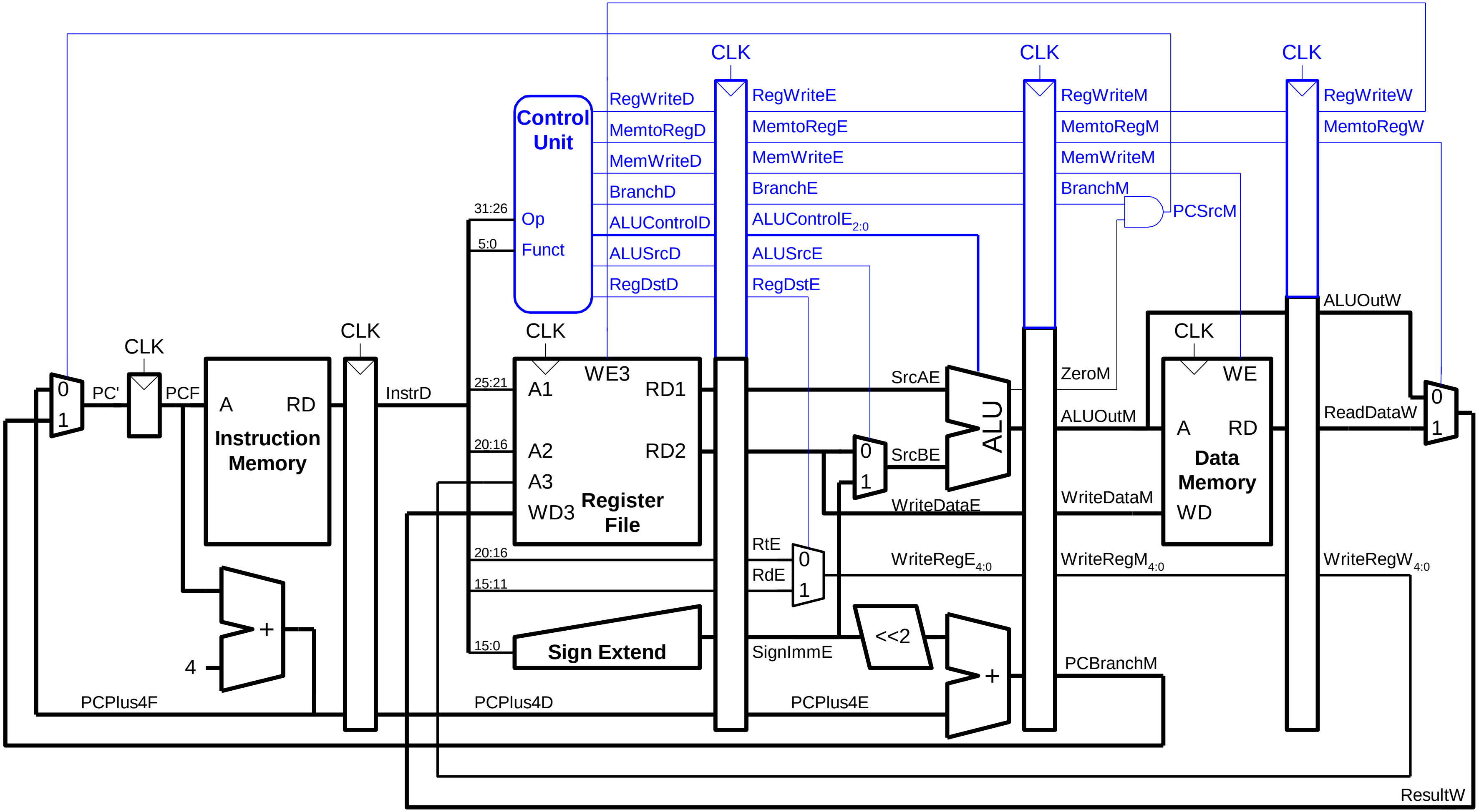


# Corrected Pipelined Datapath

- WriteReg must arrive at the same time as Result



# Pipelined Control



Same control unit as single-cycle processor

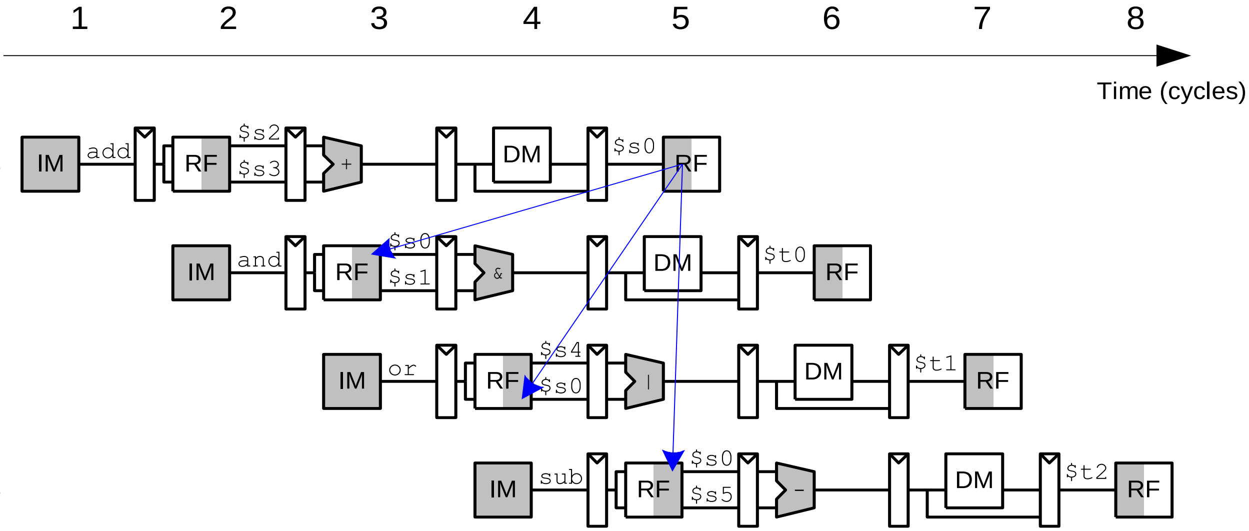
Control delayed to proper pipeline stage

# 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

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

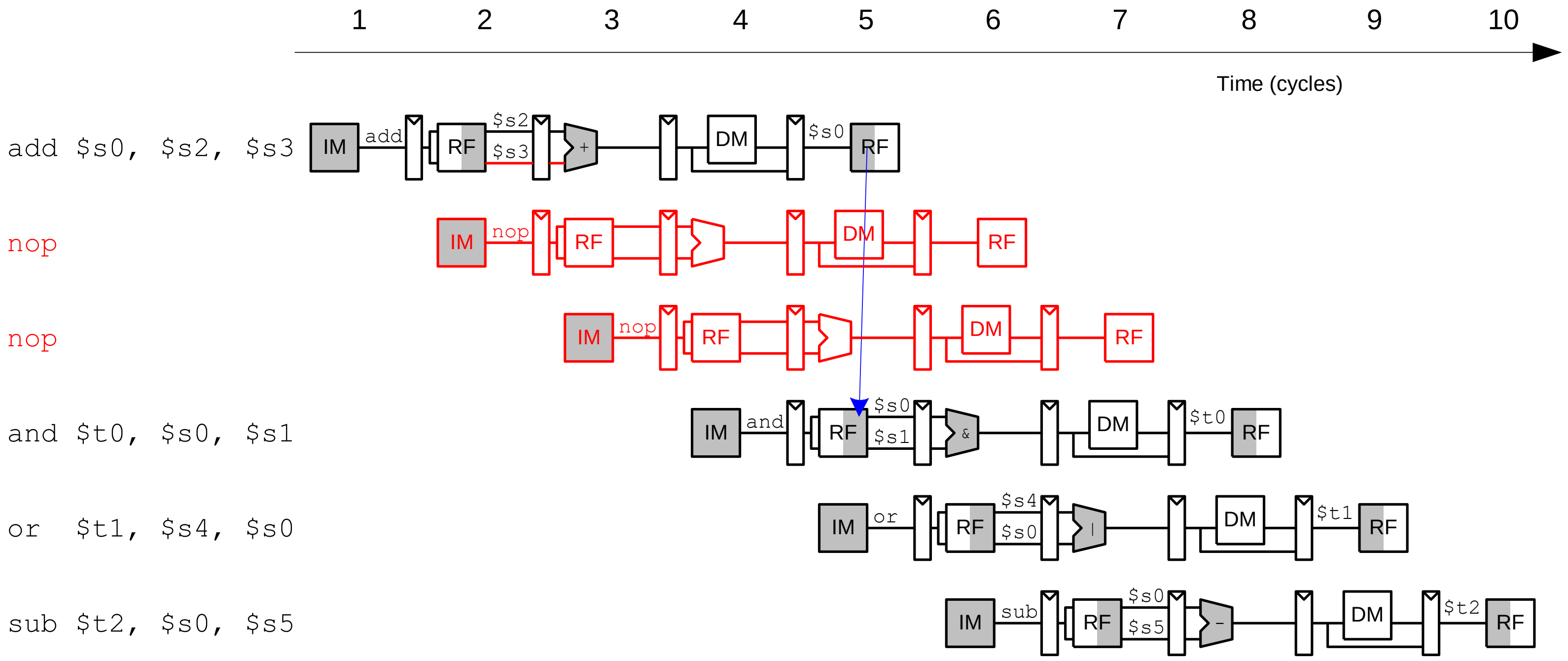


# Handling Data Hazards

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

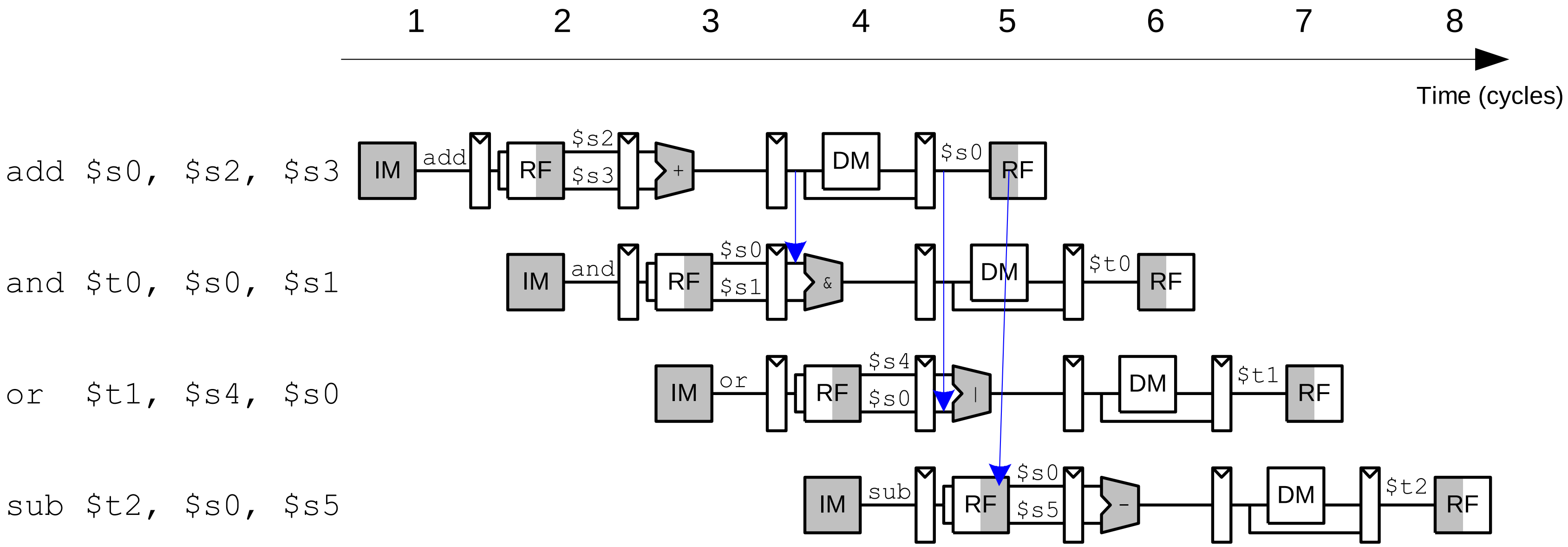
# Compile-Time Hazard Elimination

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

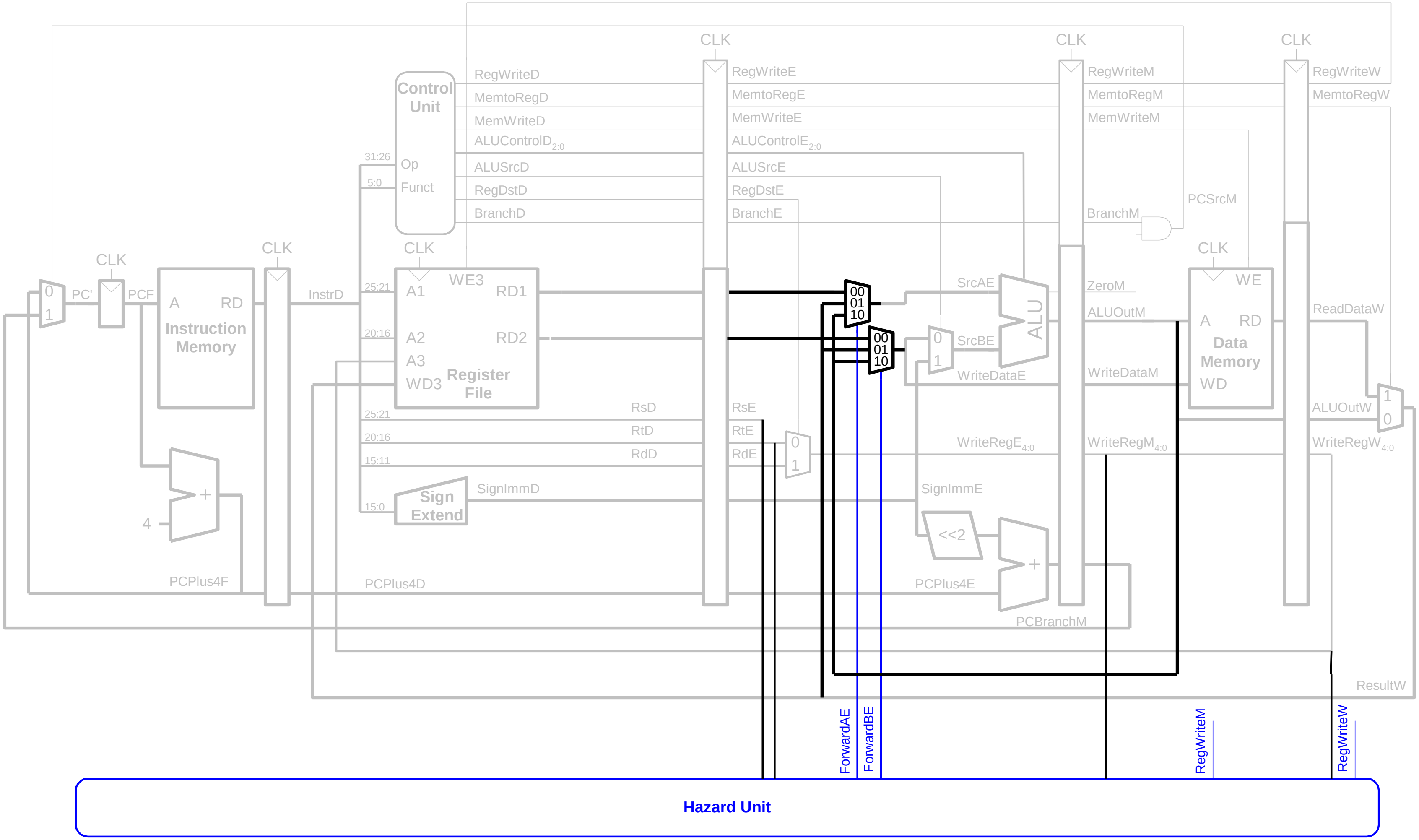


```
add $s0, $s2, $s3
nop
nop
and $t0, $s0, $s1
or  $t1, $s4, $s0
sub $t2, $s0, $s5
```

# Data Forwarding



# Data Forwarding



# Data Forwarding

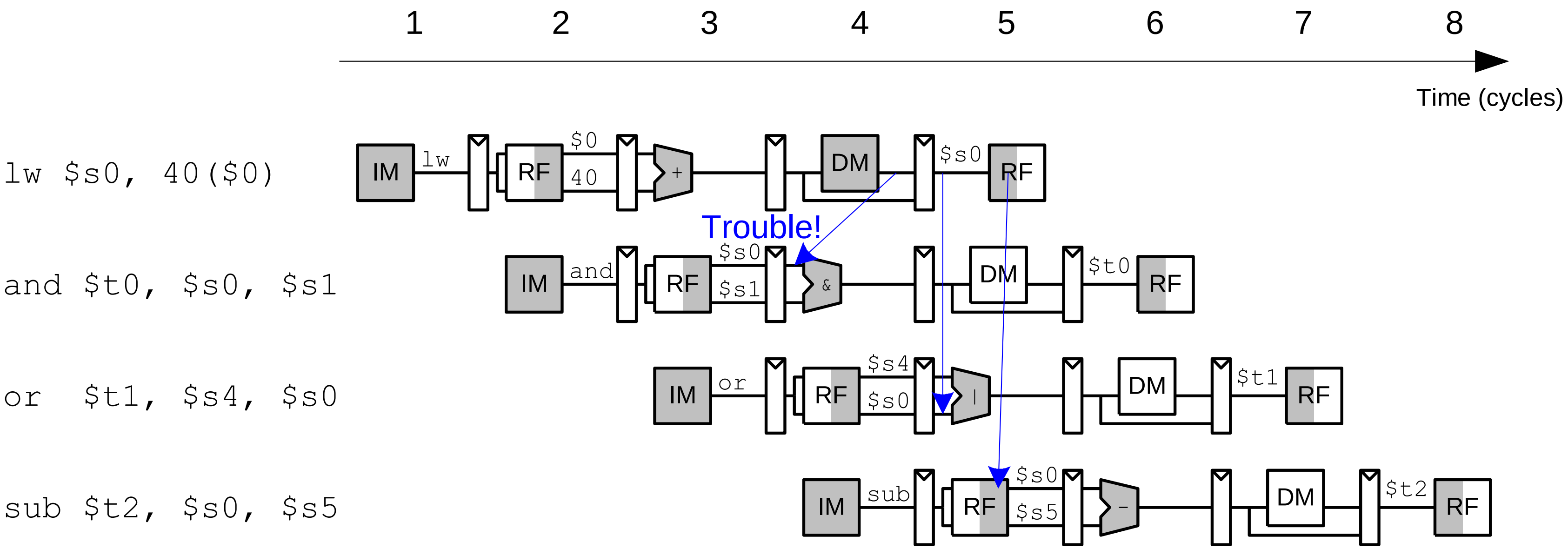
- Forward to Execute stage from either:
  - Memory stage or
  - Writeback stage

- 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
```

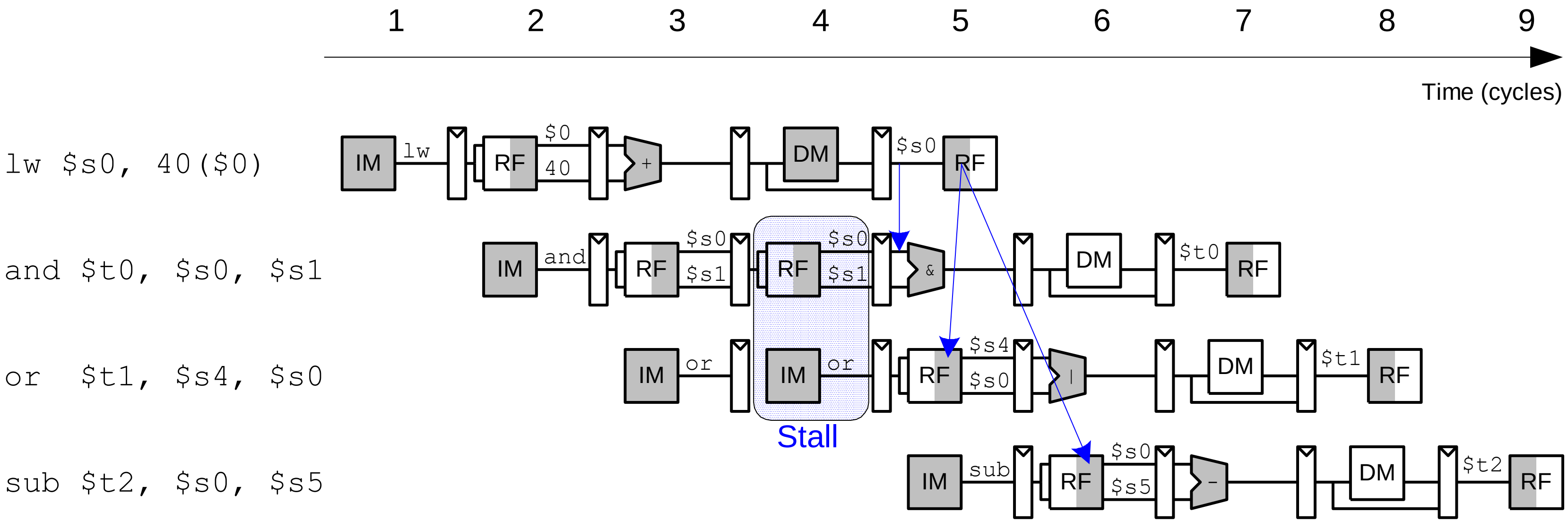
- Forwarding logic for *ForwardBE* same, but replace *rsE* with *rtE*

# Stalling

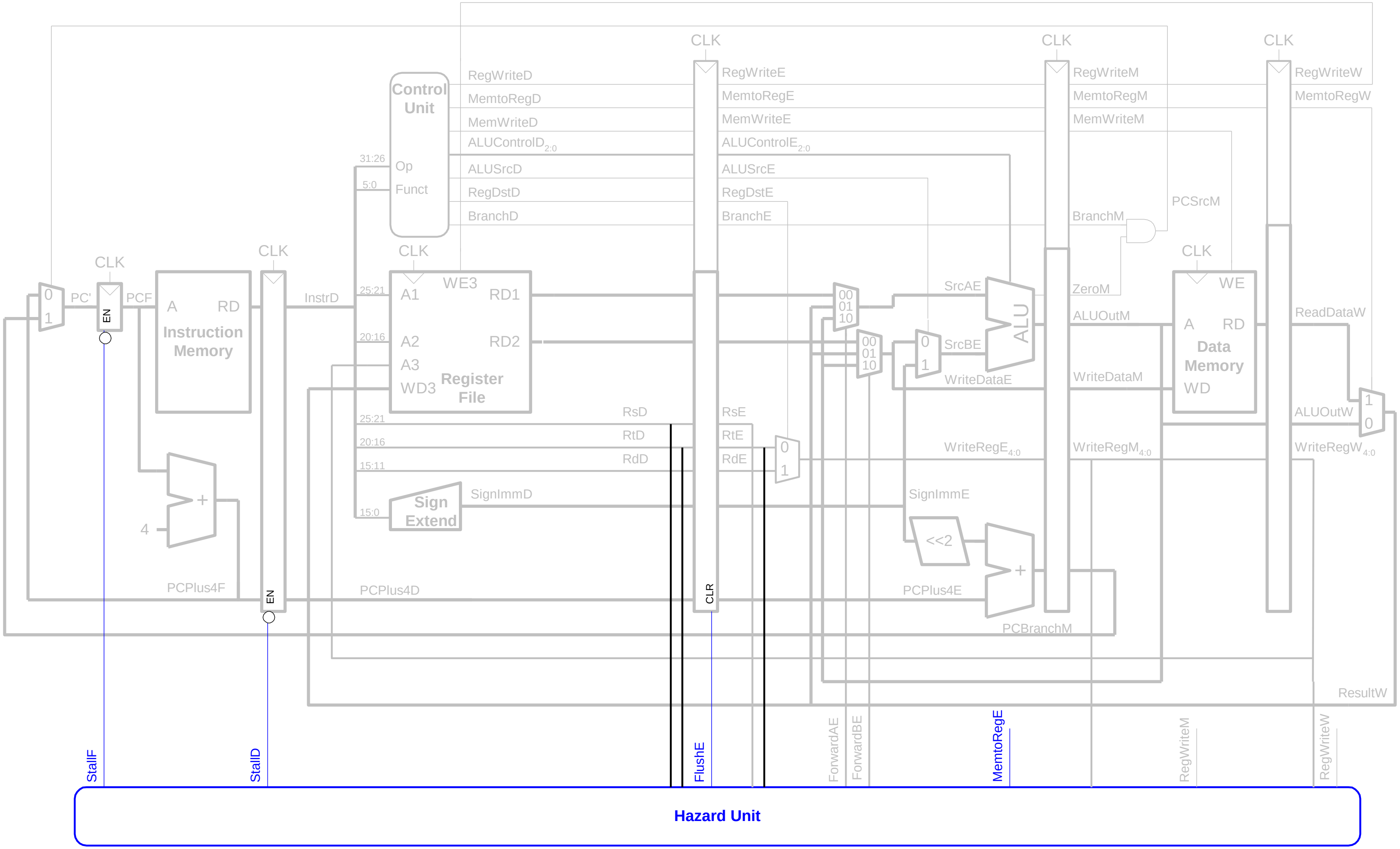


```
lw $s0, 40($0)
and $t0, $s0, $s1
or  $t1, $s4, $s0
sub $t2, $s0, $s5
```

# 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
IM
sub
RF
$s0
$s5
-
DM
$t2
RF
Stall

# Stalling Hardware



# Stalling Hardware

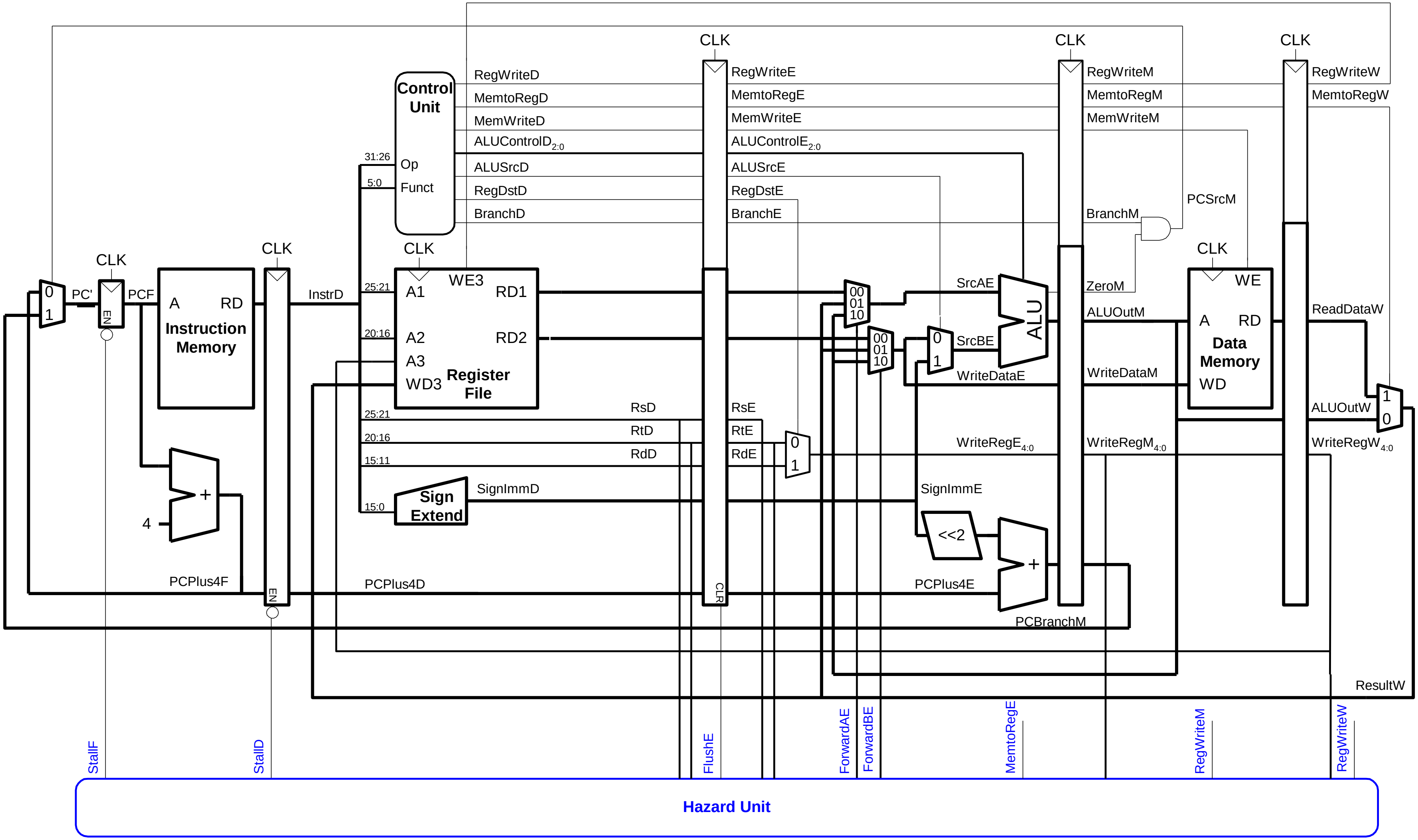
- Stalling logic:

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

StallF = StallD = FlushE = lwstall
```

# Control Hazards

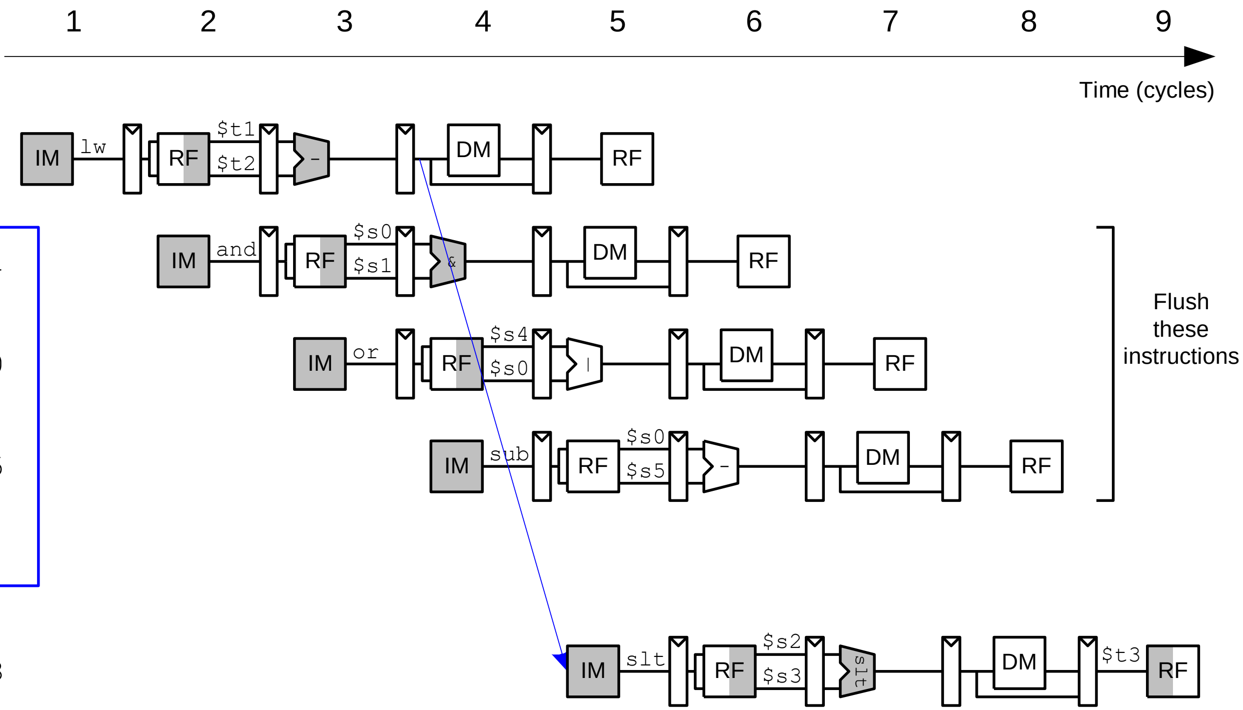
- `beq`:
  - branch is not determined until the fourth stage of the pipeline
  - Instructions after the branch are fetched before branch occurs
  - These instructions must be flushed if the branch happens
- Branch misprediction penalty
  - number of instruction flushed when branch is taken
  - May be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

# Control Hazards: Original Pipeline



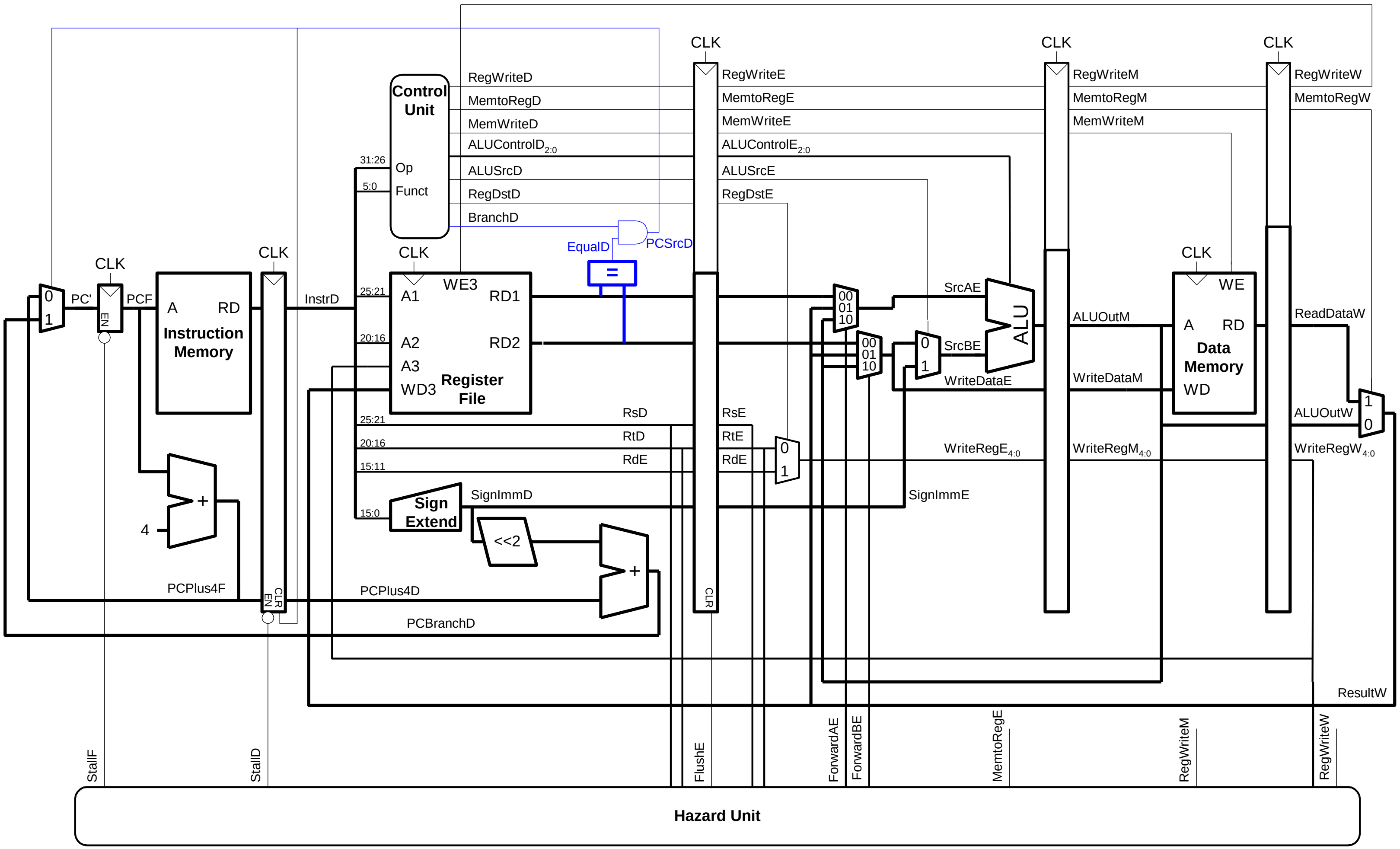
# Control Hazards

```
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
```



Flush these instructions

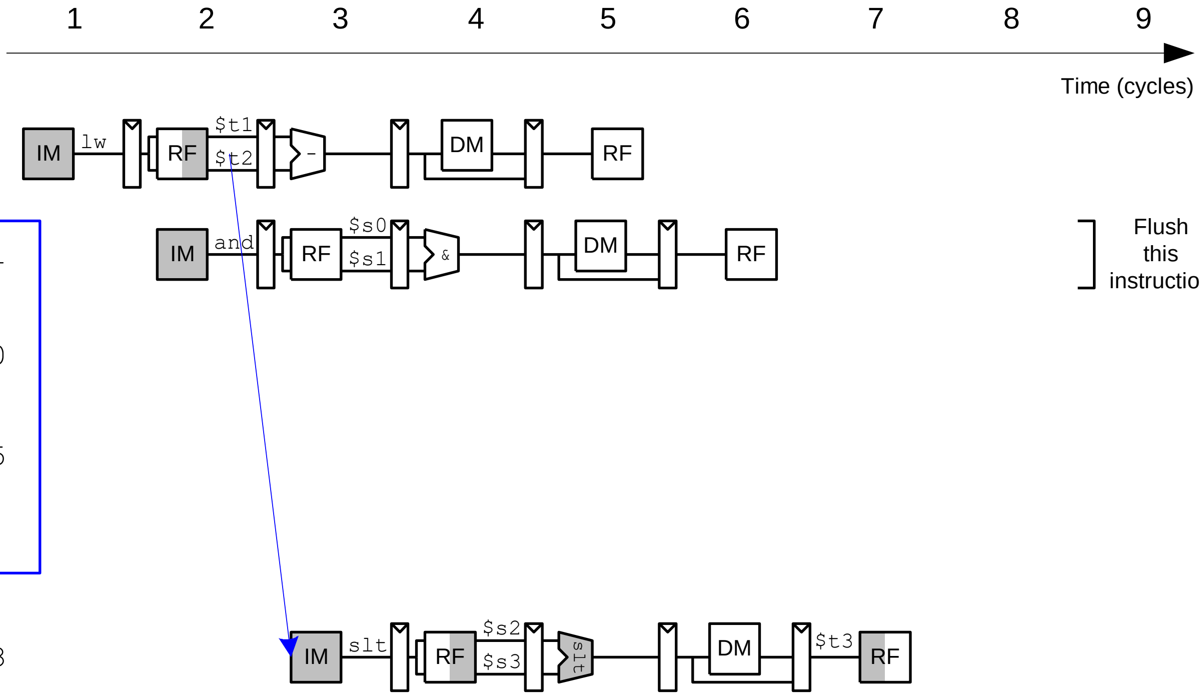
# Control Hazards: Early Branch Resolution



Introduced another data hazard in Decode stage

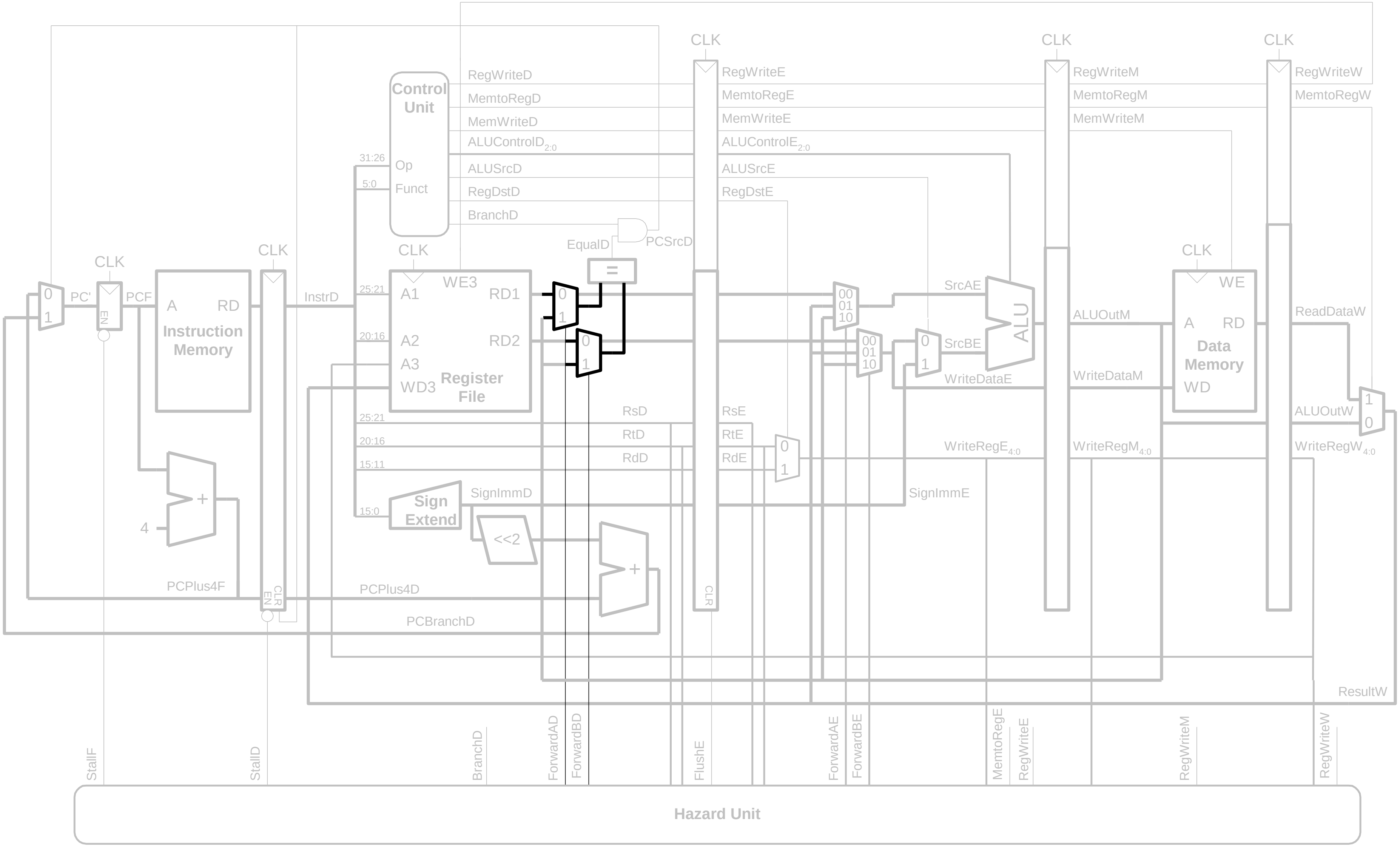
# Control Hazards with Early Branch Resolution

```
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
```



Flush this instruction

# 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

- 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. 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 Performance

- Pipelined processor critical path:

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

# Pipelined Performance Example

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

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

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

# Pipelined Performance Example

- For a program with 100 billion instructions executing on a pipelined MIPS processor,
- CPI = 1.15
- $T_c$ = 550 ps

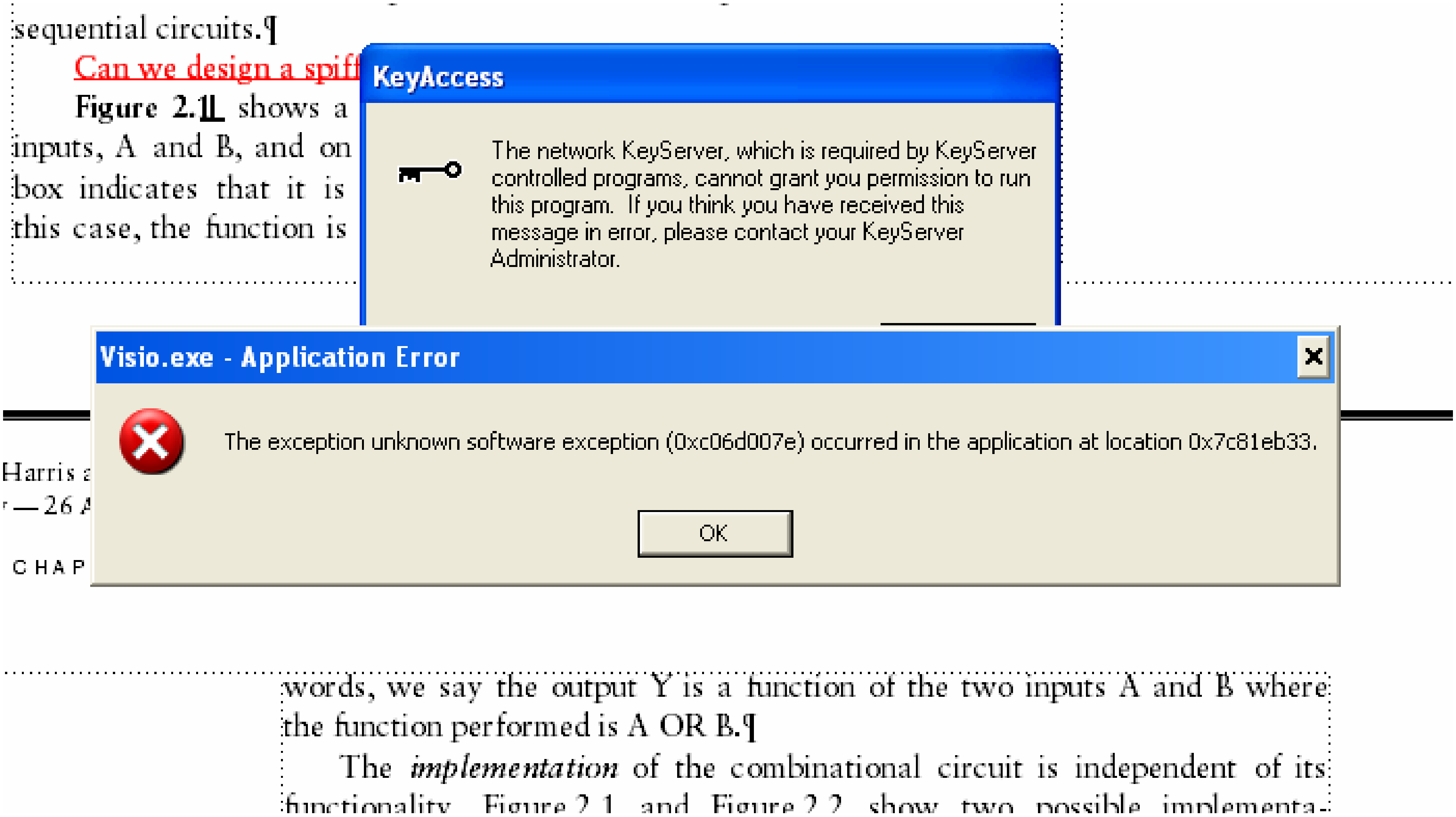
$$
\begin{aligned}
\text{Execution Time} &= (\#\ \text{instructions}) \times \text{CPI} \times T_c \\
&= (100 \times 10^{9})(1.15)(550 \times 10^{-12}) \\
&= 63\ \text{seconds}
\end{aligned}
$$

| **Processor** | **Execution Time (seconds)** | **Speedup (single-cycle is baseline)** |
|---|---|---|
| Single-cycle | 95 | 1 |
| Multicycle | 133 | 0.71 |
| Pipelined | 63 | 1.51 |

# Review: Exceptions

- Unscheduled procedure call to the *exception handler*
- Casued by:
  - Hardware, also called an *interrupt*, e.g. keyboard
  - Software, also called *traps*, e.g. undefined instruction
- When exception occurs, the processor:
  - Records the cause of the exception (`Cause` register)
  - Jumps to the exception handler at instruction address 0x80000180
  - Returns to program (`EPC` register)

# Example Exception



sequential circuits.¶

Can we design a spiff

Figure 2.1L shows a inputs, A and B, and on box indicates that it is this case, the function is

**KeyAccess**

The network KeyServer, which is required by KeyServer controlled programs, cannot grant you permission to run this program. If you think you have received this message in error, please contact your KeyServer Administrator.

**Visio.exe - Application Error**

The exception unknown software exception (0xc06d007e) occurred in the application at location 0x7c81eb33.

Harris a

— 26 A

CHAP

words, we say the output Y is a function of the two inputs A and B where the function performed is A OR B.¶

The *implementation* of the combinational circuit is independent of its functionality. Figure 2.1 and Figure 2.2 show two possible implementa-

# Exception Registers

- Not part of the register file.
  - `Cause`
    - Records the cause of the exception
    - Coprocessor 0 register 13
  - `EPC` (Exception PC)
    - Records the PC where the exception occurred
    - Coprocessor 0 register 14
- Move from Coprocessor 0
  - `mfc0 $t0, Cause`
  - Moves the contents of `Cause` into `$t0`

**mfc0**

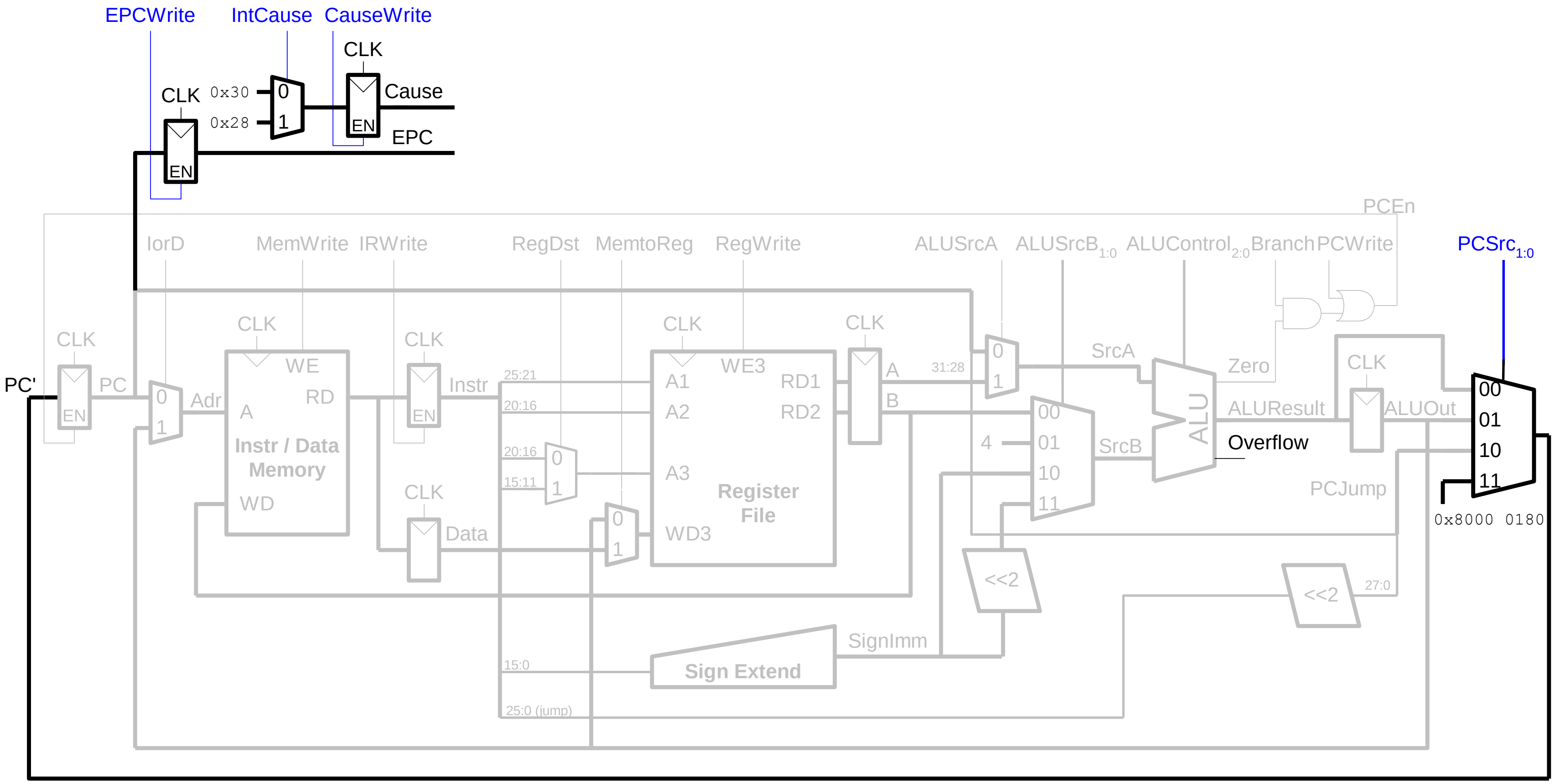
| | | | | |
|---|---|---|---|---|
| 010000 | 00000 | `$t0` (8) | `Cause` (13) | 00000000000 |
| 31:26 | 25:21 | 20:16 | 15:11 | 10:0 |

# Exception Causes

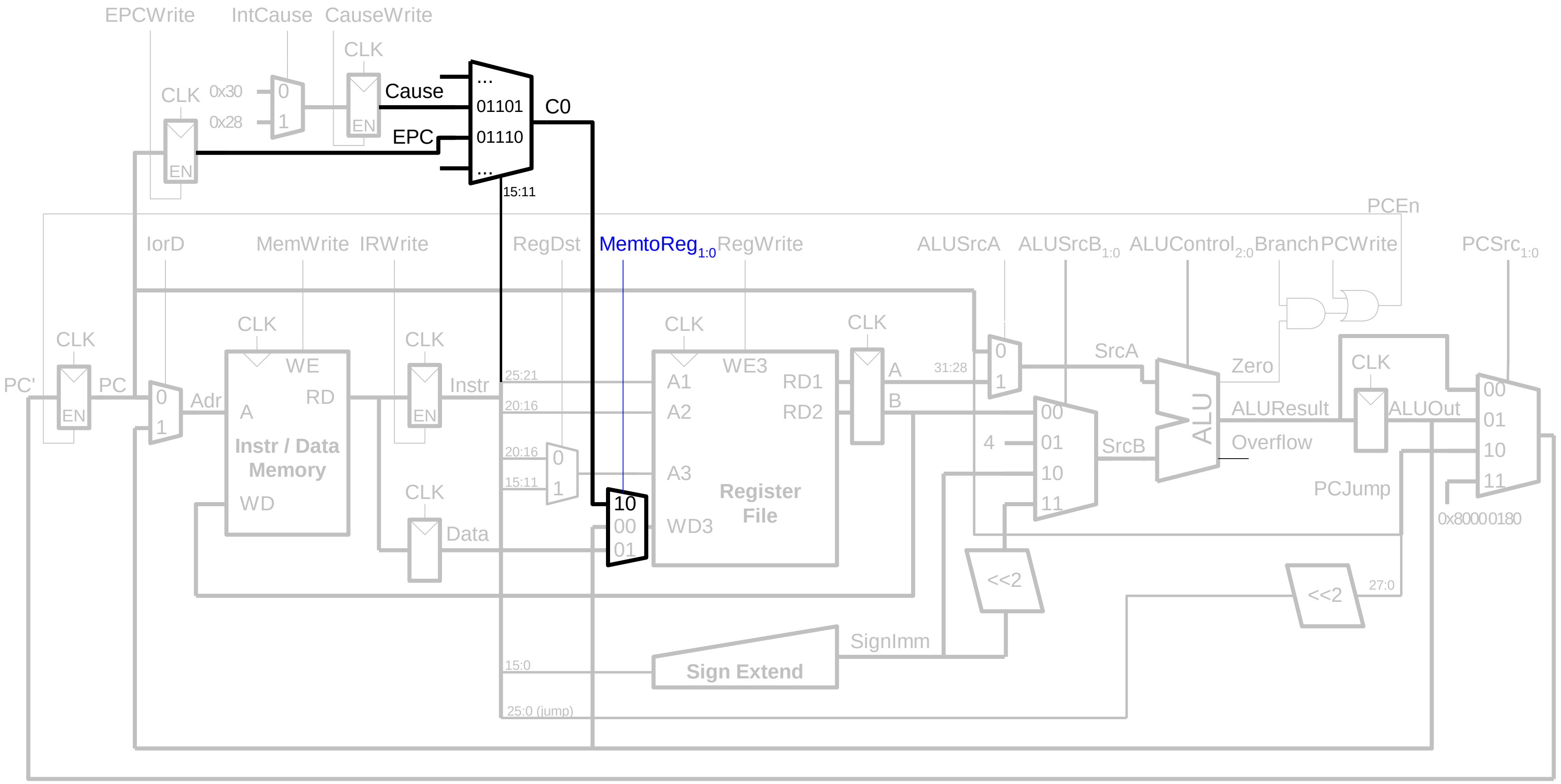
| Exception | Cause |
|---|---|
| Hardware Interrupt | 0x00000000 |
| System Call | 0x00000020 |
| Breakpoint / Divide by 0 | 0x00000024 |
| Undefined Instruction | 0x00000028 |
| Arithmetic Overflow | 0x00000030 |

**We extend the multicycle MIPS processor to handle the last two types of exceptions.**

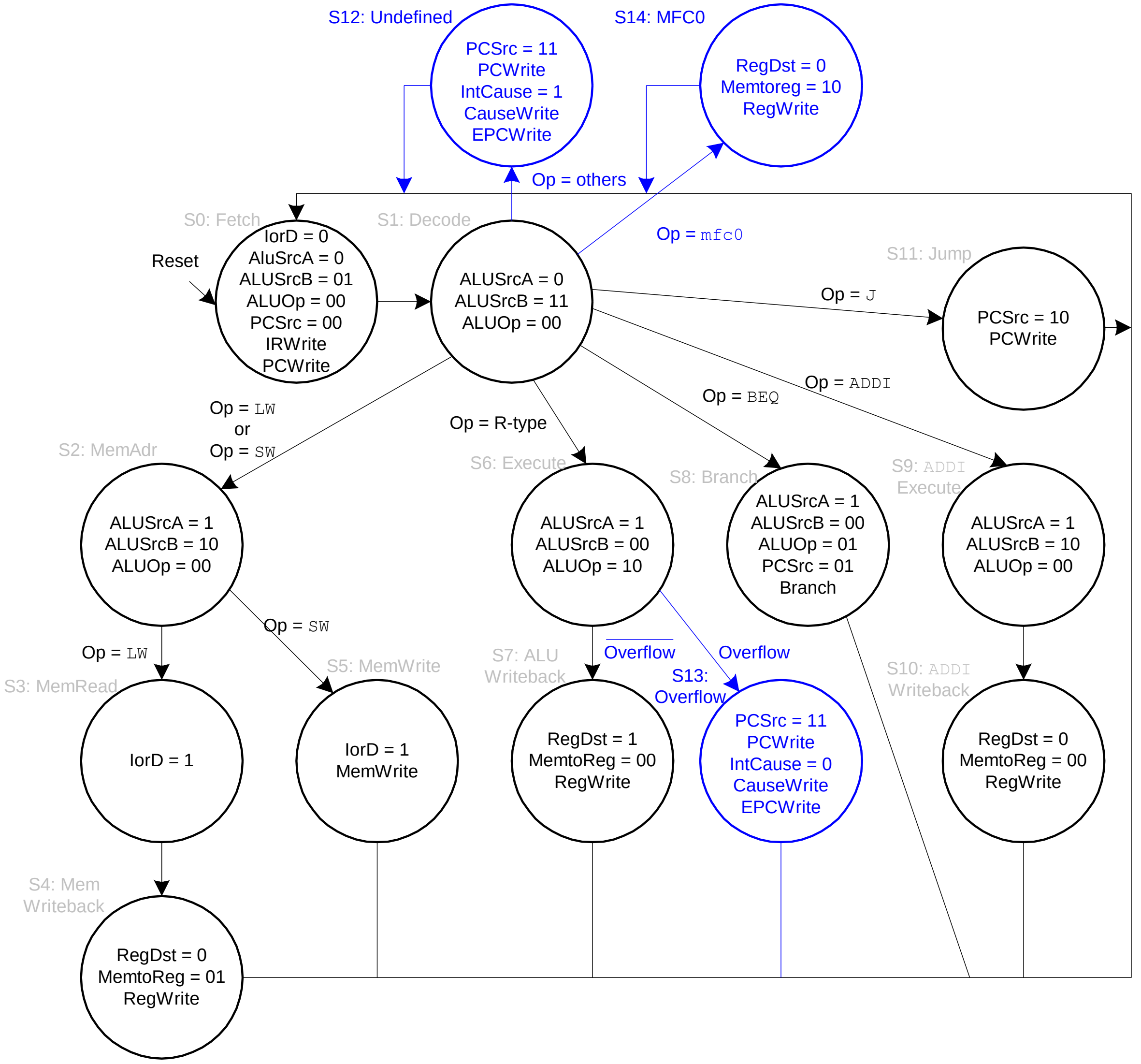
# Exception Hardware: EPC & Cause



# Exception Hardware: `mfc0`



# Control FSM with Exceptions



# 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
  - Otherwise, predict not taken
- Dynamic branch prediction:
  - Keep history of last (several hundred) branches in a *branch target buffer* which holds:
    - Branch destination
    - Whether branch was taken

# Branch Prediction Example

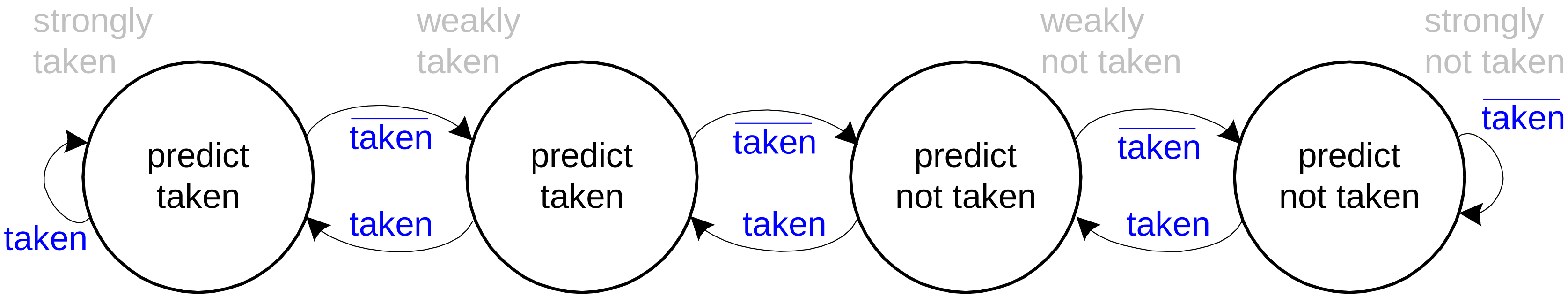
```
  add  $s1, $0, $0      # sum = 0
  add  $s0, $0, $0      # i   = 0
  addi $t0, $0, 10      # $t0 = 10
for:
  beq  $t0, $t0, done   # if i == 10, branch
  add  $s1, $s1, $s0    # sum = sum + i
  addi $s0, $s0, 1      # increment i
  j    for
done:
```

# 1-Bit Branch Predictor

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

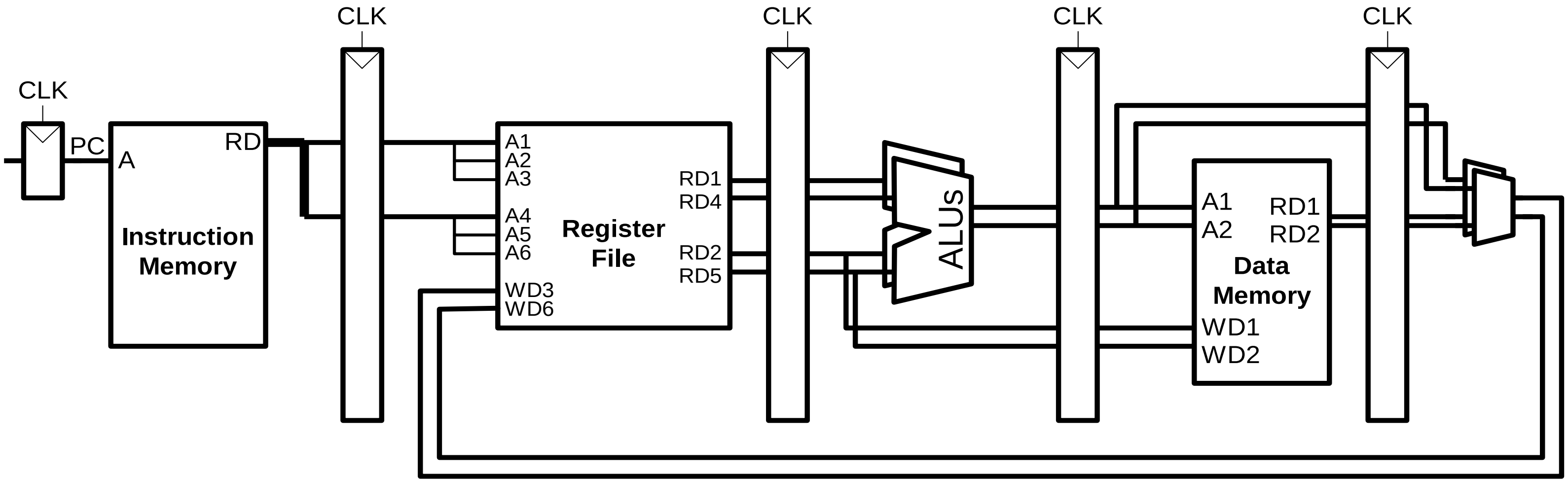
# 2-Bit Branch Predictor

- 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

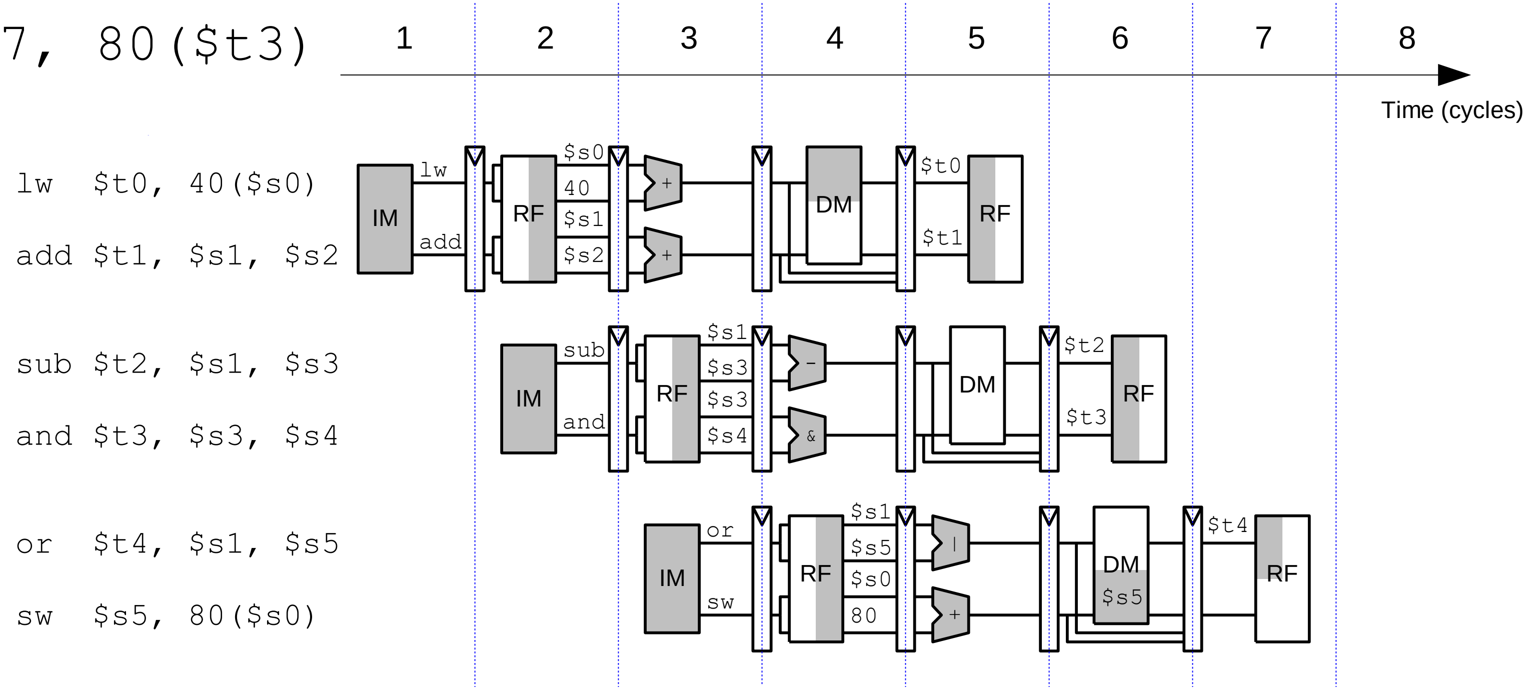


# Superscalar Example

```
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**

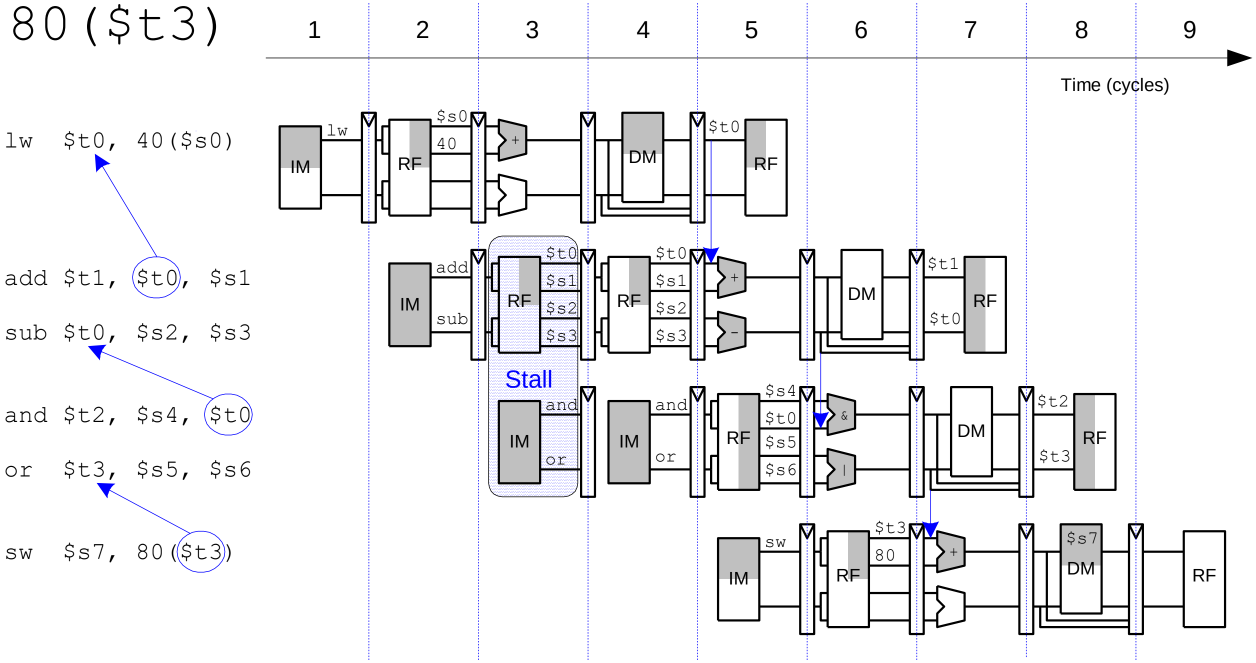


# Superscalar Example with Dependencies

```
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 = 1.17**



```
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 to issue as many as possible at once
- Issues instructions out of order as long as no dependencies
- Dependencies:
  - **RAW** (read after write): one instruction writes, and later instruction reads a register
  - **WAR** (write after read): one instruction reads, and a later instruction writes a register (also called an *antidependence*)
  - **WAW** (write after write): one instruction writes, and a later instruction writes a register (also called an *output dependence*)

# Out of Order Processor

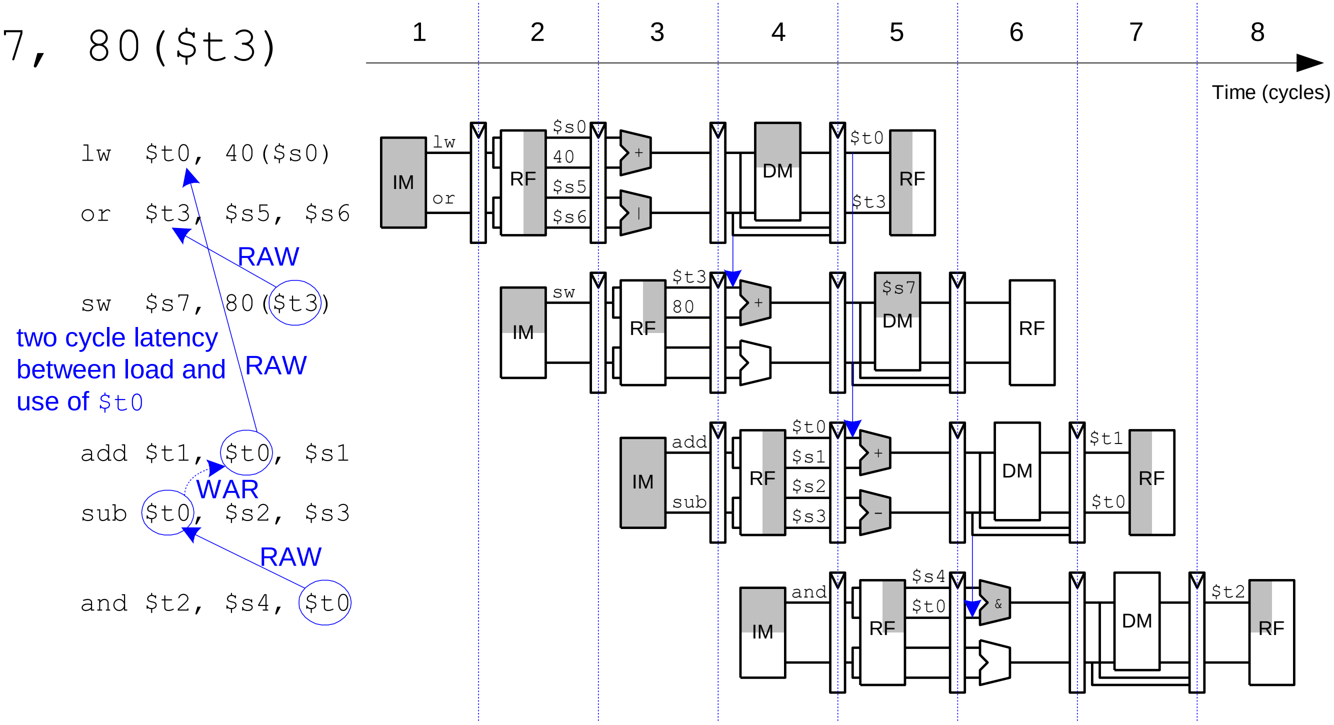
- **Instruction level parallelism:** the number of instruction that can be issued simultaneously (in practice average < 3)
- **Scoreboard:** table that keeps track of:
  - Instructions waiting to issue
  - Available functional units
  - Dependencies

# 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
```

RAW

two cycle latency between load and use of $t0

RAW

WAR

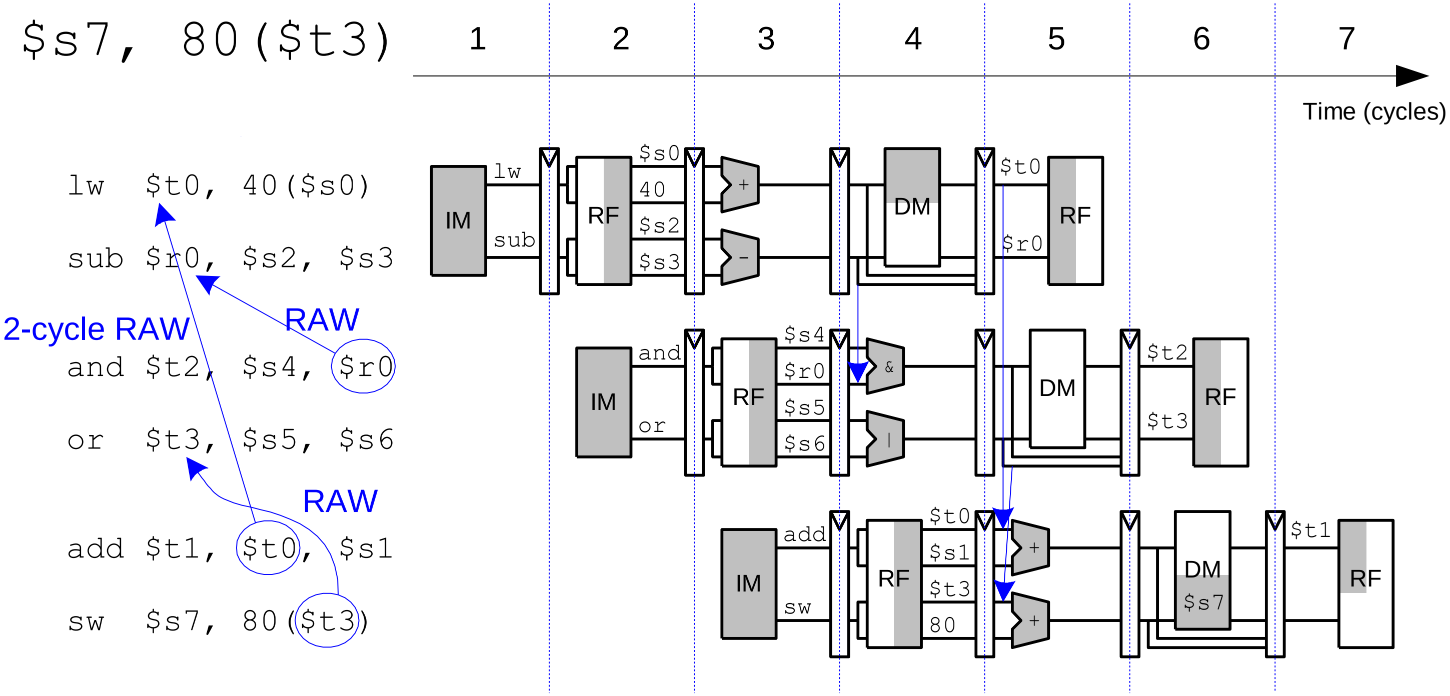
RAW

# 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**

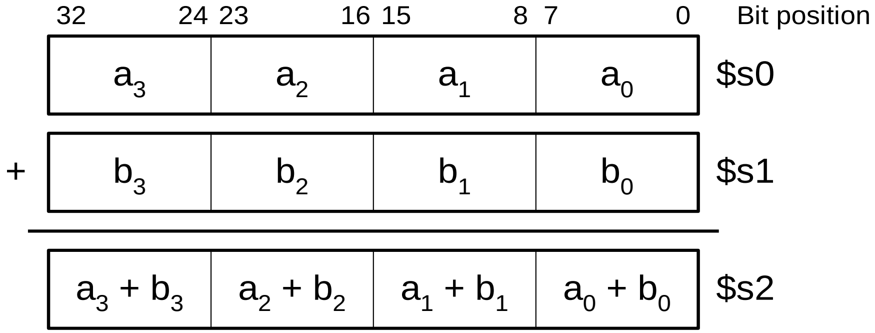


```
lw   $t0, 40($s0)
sub  $r0, $s2, $s3
and  $t2, $s4, $r0
or   $t3, $s5, $s6
add  $t1, $t0, $s1
sw   $s7, 80($t3)
```

# 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
- Must modify ALU to eliminate carries between 8-bit values

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



# Advanced Architecture Techniques

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