EECS150 - Digital Design

Lecture 7- MIPS CPU Microarchitecture

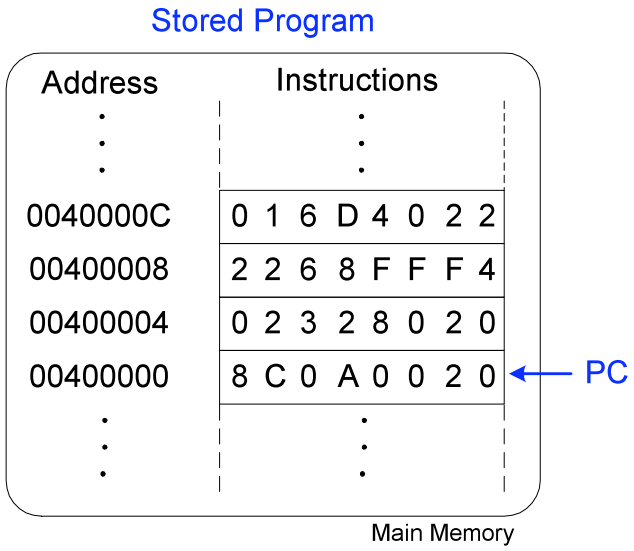
Feb 4, 2012

John Wawrzynek

Key 61c Concept: "Stored Program"

- Instructions and data stored in memory.
- Only difference between two applications (for example, a text editor and a video game), is the sequence of instructions.
- To run a new program:
  - No rewiring required
  - Simply store new program in memory
  - The processor hardware executes the program:
  - fetches (reads) the instructions from memory in sequence
  - performs the specified operation
- The program counter (PC) keeps track of the current instruction.

Assembly Code	Machine Code
`lw $t2, 32($0)`	`0x8C0A0020`
`add $s0, $s1, $s2`	`0x02328020`
`addi $t0, $s3, -12`	`0x2268FFF4`
`sub $t0, $t3, $t5`	`0x016D4022`



Key 61c Concept:

High-level languages help productivity.

High-level code

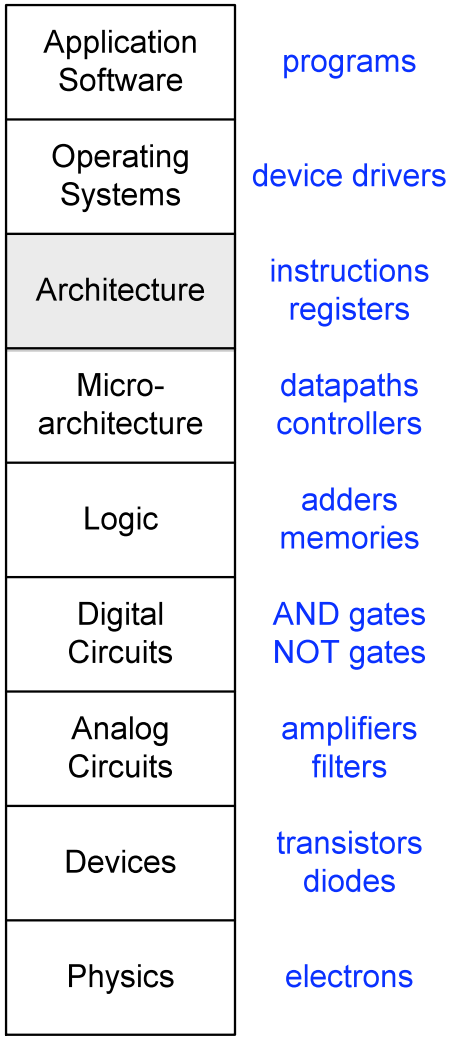
```
// add the numbers from 0 to 9
int sum = 0;
int i;

for (i=0; i!=10; i = i+1) {
  sum = sum + i;
}
```

MIPS assembly code

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

Therefore with the help of a compiler (and assembler), to run applications all we need is a means to interpret (or "execute") machine instructions. Usually the application calls on the operating system and libraries to provide special functions.

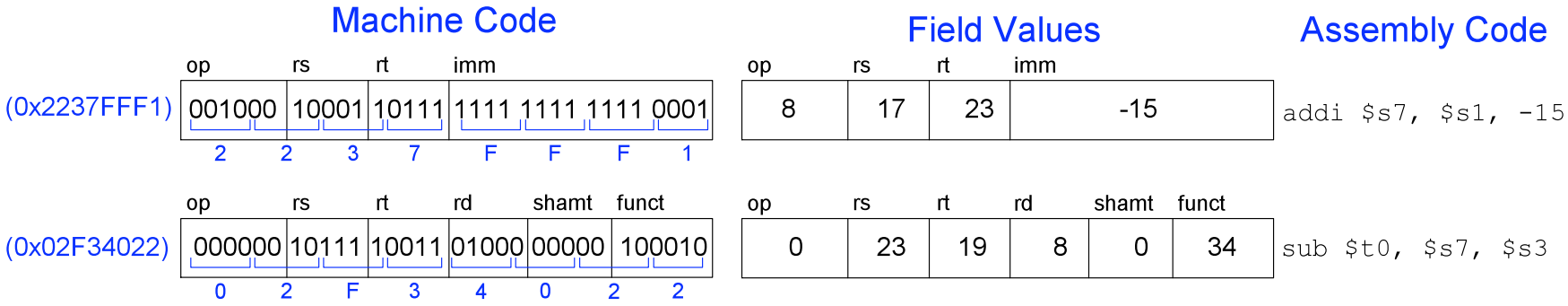
Abstraction Layers



- **Architecture**: the programmer's view of the computer
  - Defined by instructions (operations) and operand locations
- **Microarchitecture**: how to implement an architecture in hardware (covered in great detail later)
- The microarchitecture is built out of "logic" circuits and memory elements (this semester).
- All logic circuits and memory elements are implemented in the physical world with transistors.

Interpreting Machine Code

- Start with opcode
- Opcode tells how to parse the remaining bits
- If opcode is all 0's
  - R-type instruction
  - Function bits tell what instruction it is
- Otherwise
  - opcode tells what instruction it is



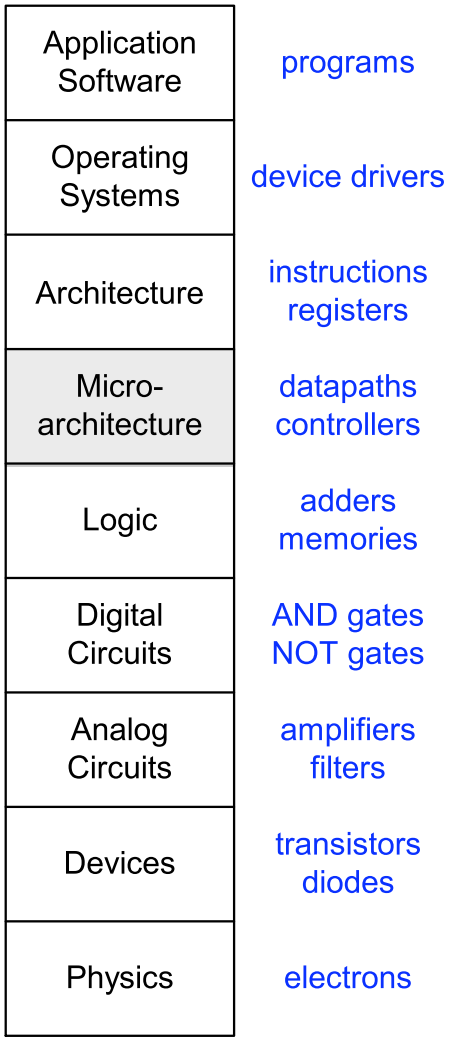
A processor is a machine code interpreter build in hardware!

Processor Microarchitecture Introduction

Microarchitecture: how to implement an architecture in hardware

Good examples of how to put principles of digital design to practice.

Introduction to final project.

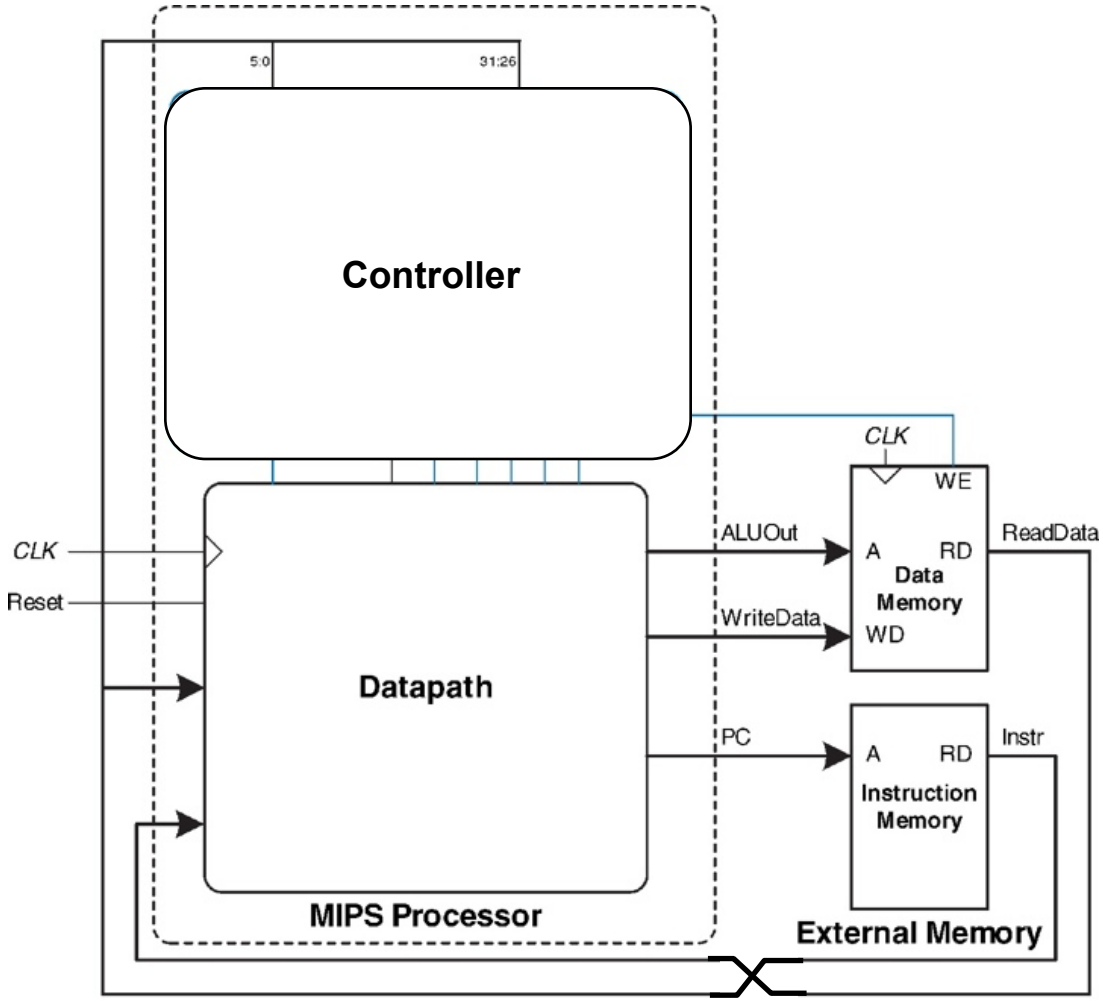


MIPS Processor Architecture

- For now we consider a subset of MIPS instructions:
  - R-type instructions: **`and, or, add, sub, slt`**
  - Memory instructions: **`lw, sw`**
  - Branch instructions: **`beq`**
- Later we'll add **`addi`** and **`j`**

MIPS Micrarchitecture Oganization

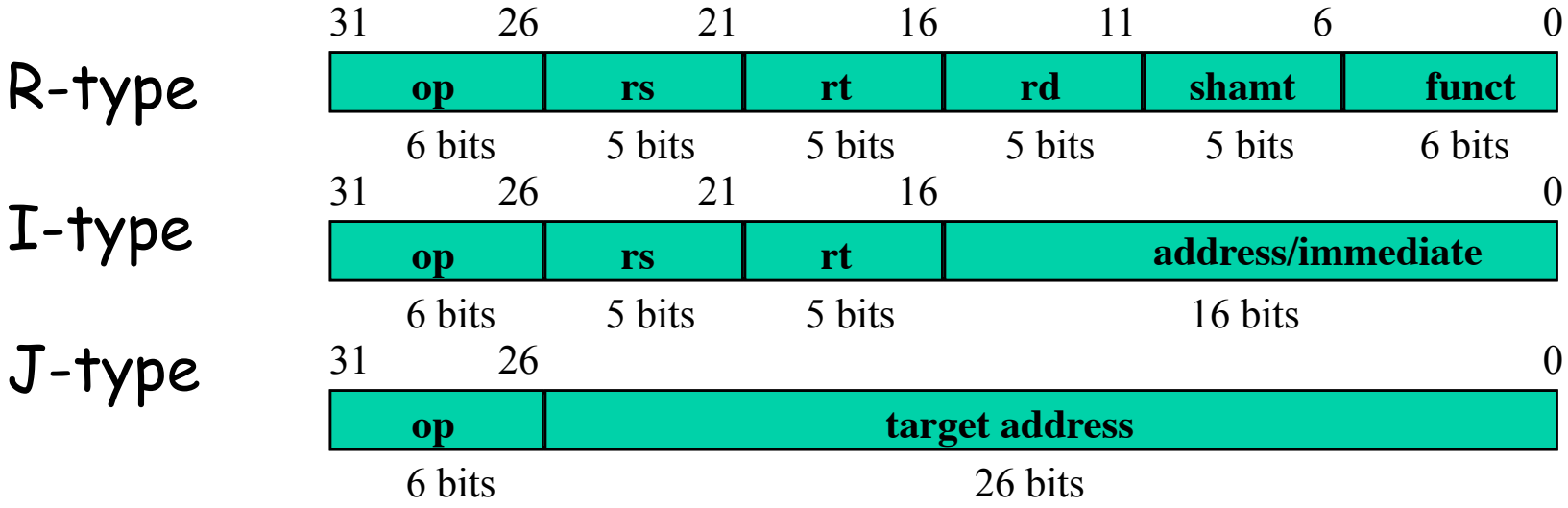
Datapath + Controller + External Memory



How to Design a Processor: step-by-step

1. Analyze instruction set architecture (ISA) ⟹ datapath requirements
   - meaning of each instruction is given by the ***data transfers (register transfers)***
   - datapath must include storage element for ISA registers
   - datapath must support each data transfer
2. Select set of datapath components and establish clocking methodology
3. Assemble datapath meeting requirements
4. Analyze implementation of each instruction to determine setting of control points that effects the data transfer.
5. Assemble the control logic.

Review: The MIPS Instruction



The different fields are:

op: operation ("opcode") of the instruction

rs, rt, rd: the source and destination register specifiers

shamt: shift amount

funct: selects the variant of the operation in the "op" field

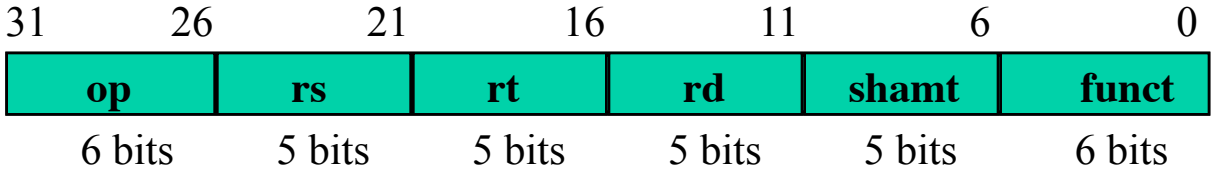
address / immediate: address offset or immediate value

target address: target address of jump instruction

Subset for Lecture

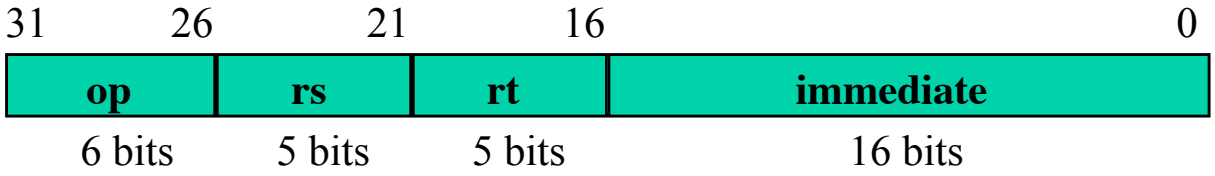
add, sub, or, slt

- **addu rd,rs,rt**
- **subu rd,rs,rt**



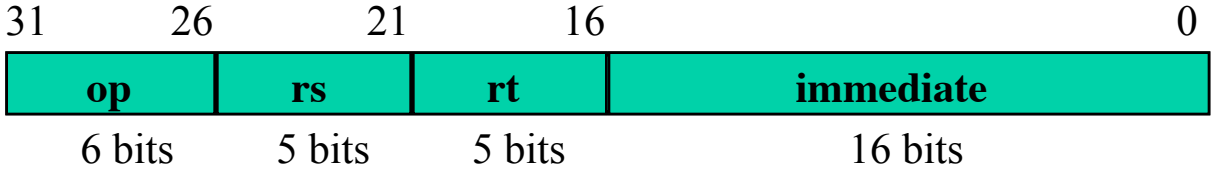
lw, sw

- **lw rt,rs,imm16**
- **sw rt,rs,imm16**



beq

- **beq rs,rt,imm16**



Register Transfer Descriptions

All start with instruction fetch:

{op , rs , rt , rd , shamt , funct} ← IMEM[PC] OR

{op , rs , rt , Imm16} ← IMEM[PC] THEN

inst	Register Transfers	
add	**R[rd] ← R[rs] + R[rt];**	**PC ← PC + 4**
sub	**R[rd] ← R[rs] – R[rt];**	**PC ← PC + 4**
or	**R[rd] ← R[rs] | R[rt];**	**PC ← PC + 4**
slt	**R[rd] ← (R[rs] < R[rt]) ? 1 : 0;**	**PC ← PC + 4**
lw	**R[rt] ← DMEM[R[rs] + sign_ext(Imm16)];**	**PC ← PC + 4**
sw	**DMEM[R[rs] + sign_ext(Imm16)] ← R[rt];**	**PC ← PC + 4**
beq	**if (R[rs] == R[rt]) then PC ← PC + 4 + {sign_ext(Imm16), 00} else PC ← PC + 4**	

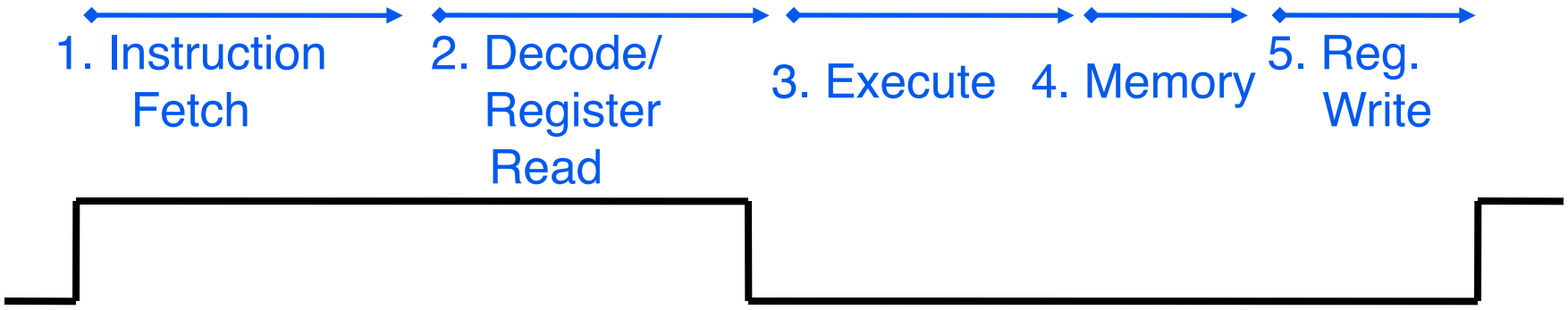
Microarchitecture

Multiple implementations for a single architecture:

- Single-cycle
  - Each instruction executes in a single clock cycle.
- Multicycle
  - Each instruction is broken up into a series of shorter steps with one step per clock cycle.
- Pipelined (variant on "multicycle")
  - Each instruction is broken up into a series of steps with one step per clock cycle
  - Multiple instructions execute at once.

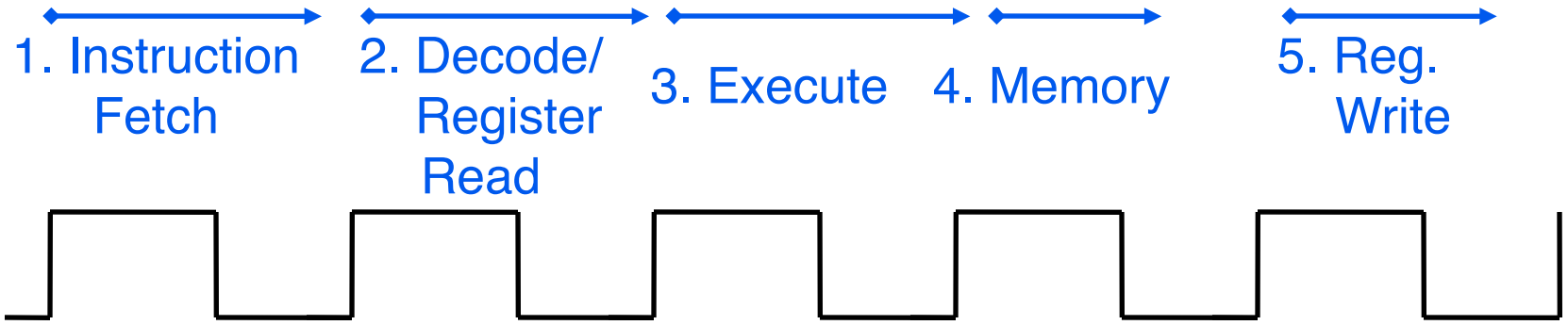
CPU clocking (1/2)

- Single Cycle CPU: All stages of an instruction are completed within one ***long*** clock cycle.
  - The clock cycle is made sufficient long to allow each instruction to complete all stages without interruption and within one cycle.



CPU clocking (2/2)

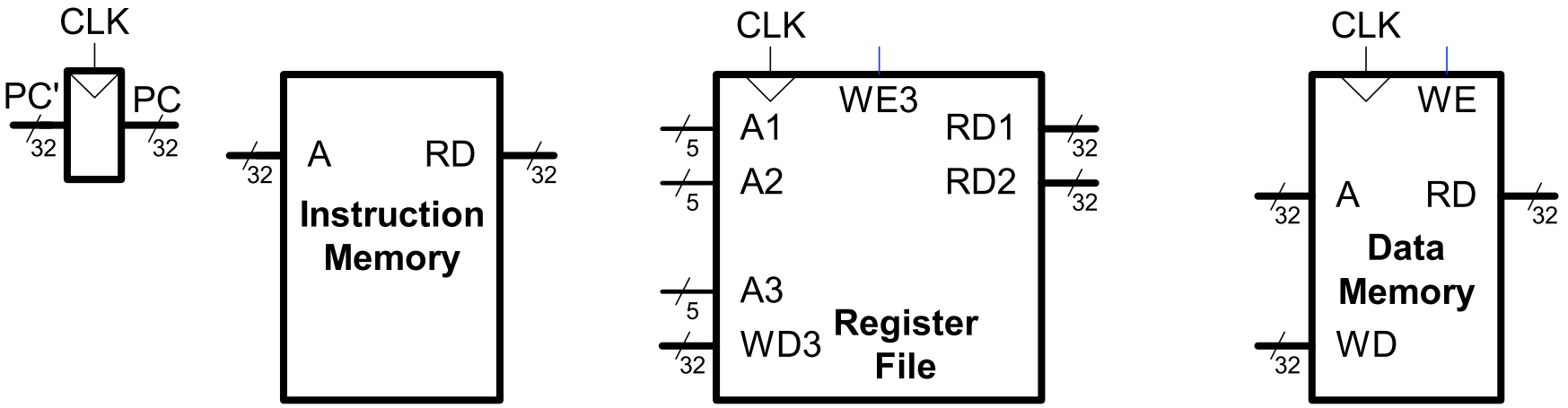
- Multiple-cycle CPU: Only one stage of instruction per clock cycle.
  - The clock is made as long as the slowest stage.



Several significant advantages over single cycle execution: Unused stages in a particular instruction can be skipped OR instructions can be pipelined (overlapped).

MIPS State Elements

- Determines everything about the execution status of a processor:
  - PC register
  - 32 registers
  - Memory



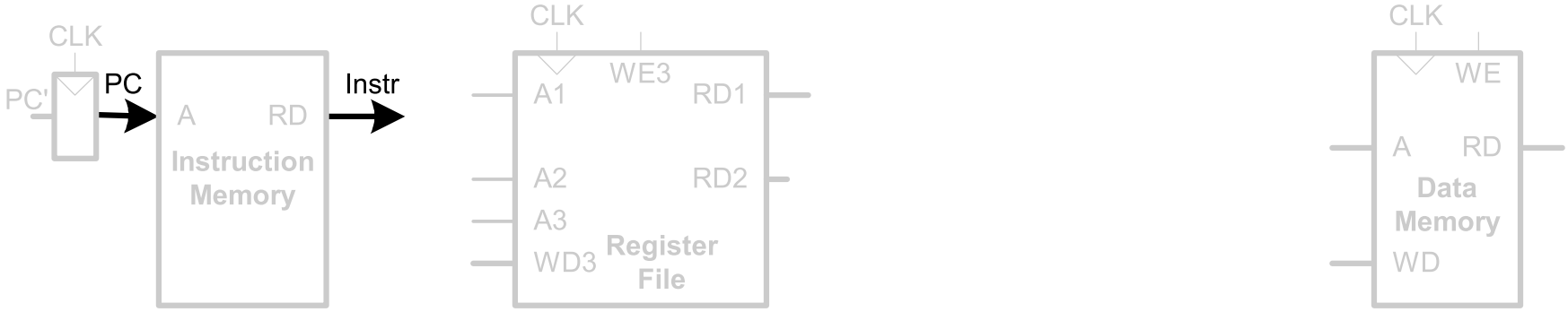
Note: for these state elements, clock is used for write but not for read (asynchronous read, synchronous write).

Single-Cycle Datapath: `lw` fetch

- First consider executing `lw`

R[rt] ← DMEM[R[rs] + sign_ext(Imm16)]

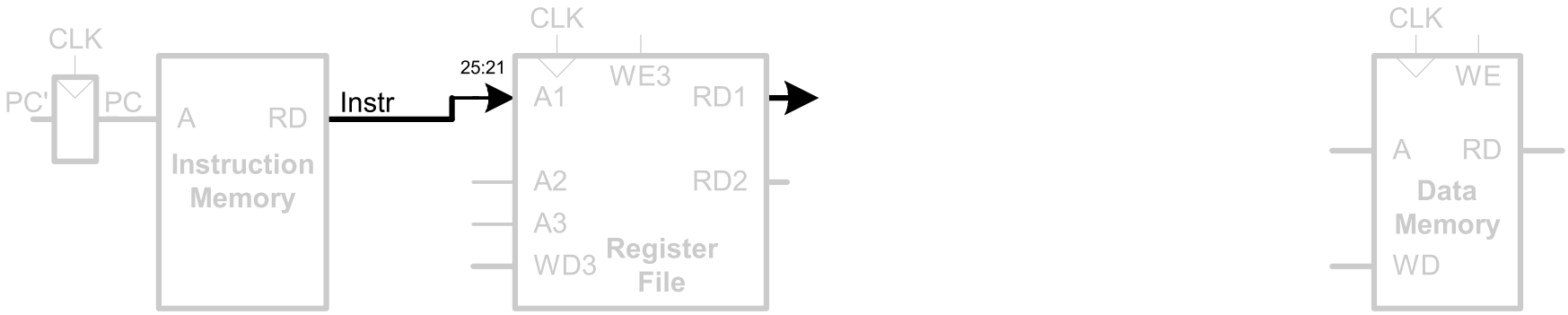
- **STEP 1:** Fetch instruction



Single-Cycle Datapath: `lw` register read

R[rt] ← DMEM[R[rs] + sign_ext(Imm16)]

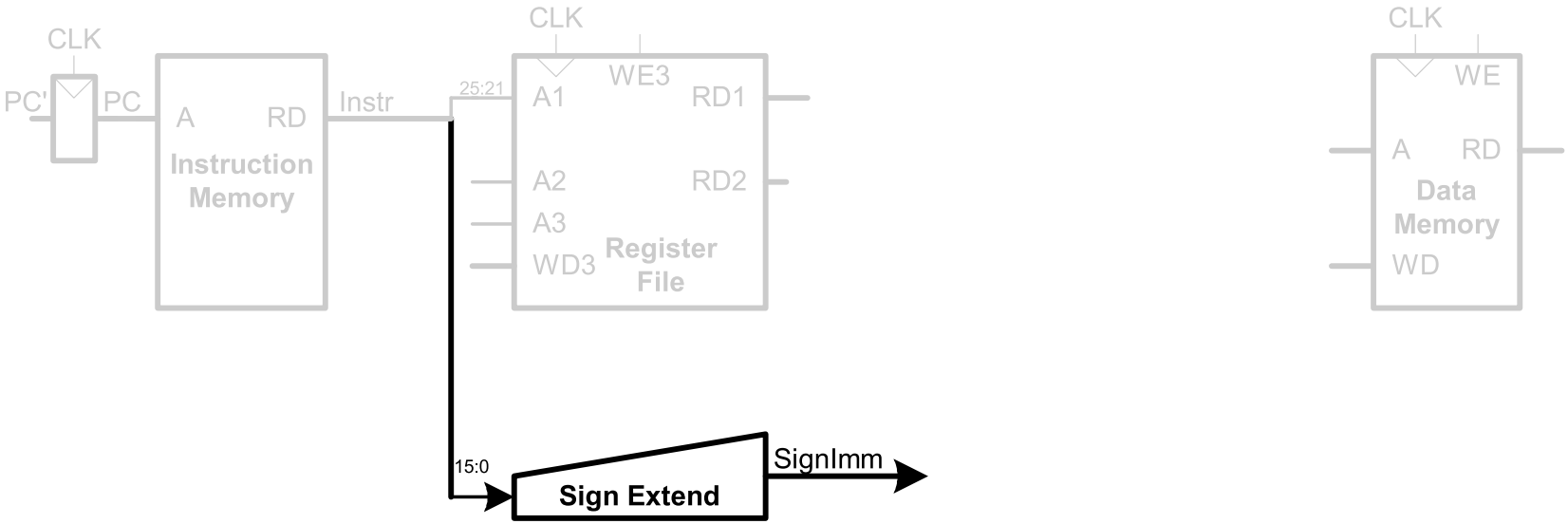
- **STEP 2:** Read source operands from register file



Single-Cycle Datapath: `lw` immediate

R[rt] ← DMEM[R[rs] + sign_ext(Imm16)]

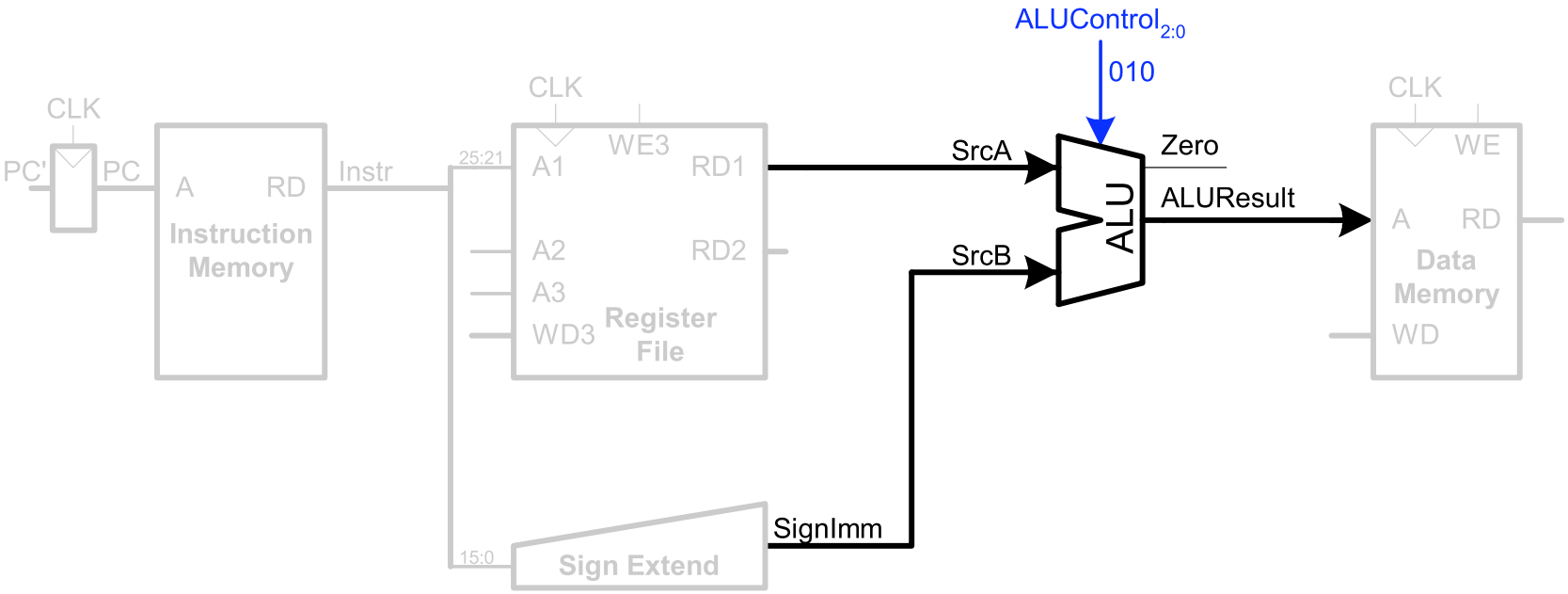
- **STEP 3:** Sign-extend the immediate



Single-Cycle Datapath: `lw` address

R[rt] ← DMEM[R[rs] + sign_ext(Imm16)]

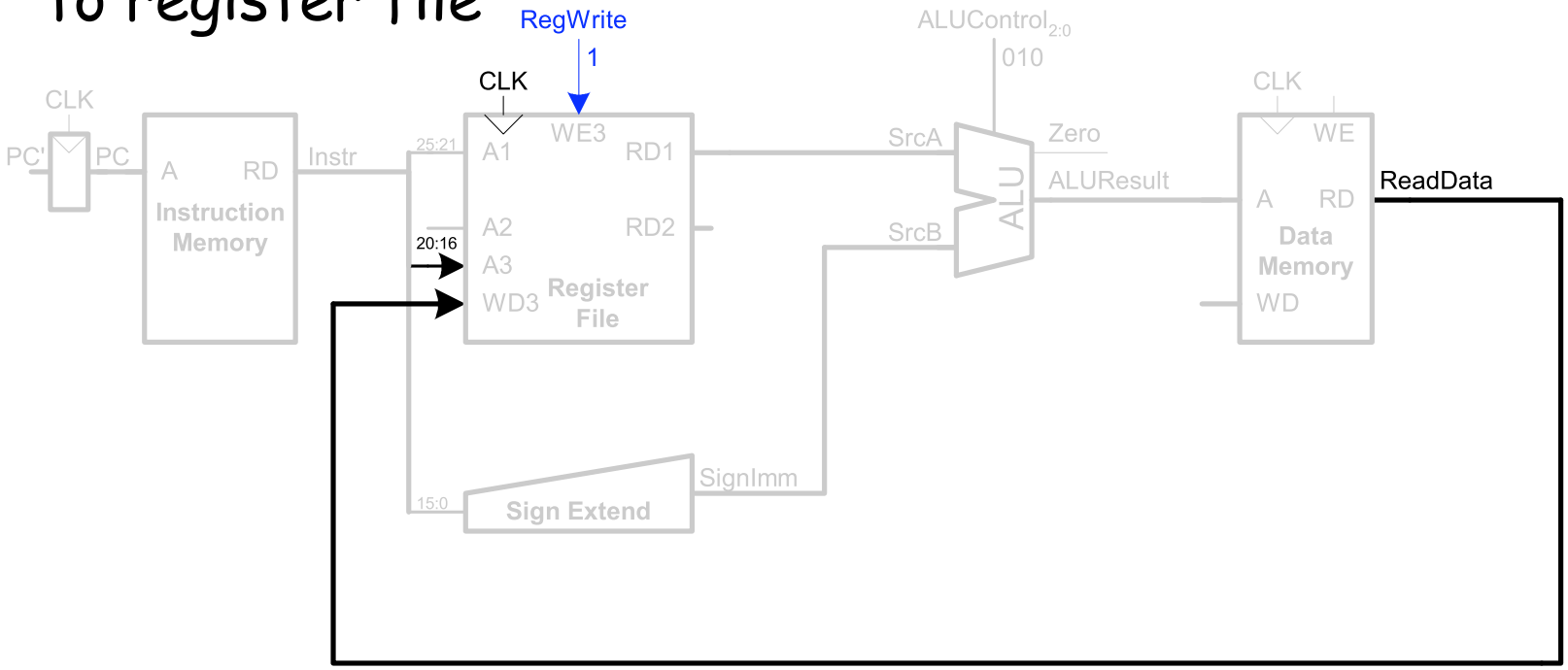
- **STEP 4:** Compute the memory address



Single-Cycle Datapath: `lw` memory read

R[rt] ← DMEM[R[rs] + sign_ext(Imm16)]

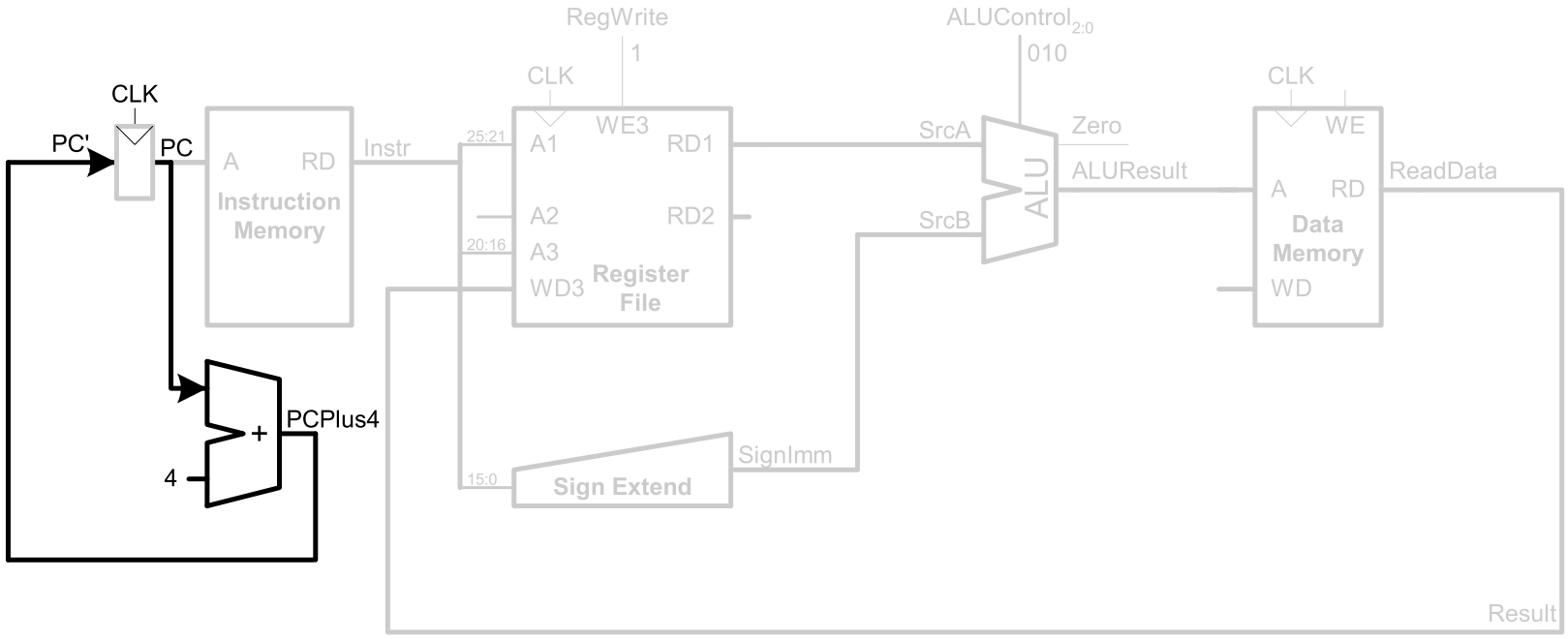
- **STEP 5:** Read data from memory and write it back to register file



Single-Cycle Datapath: `lw` PC increment

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

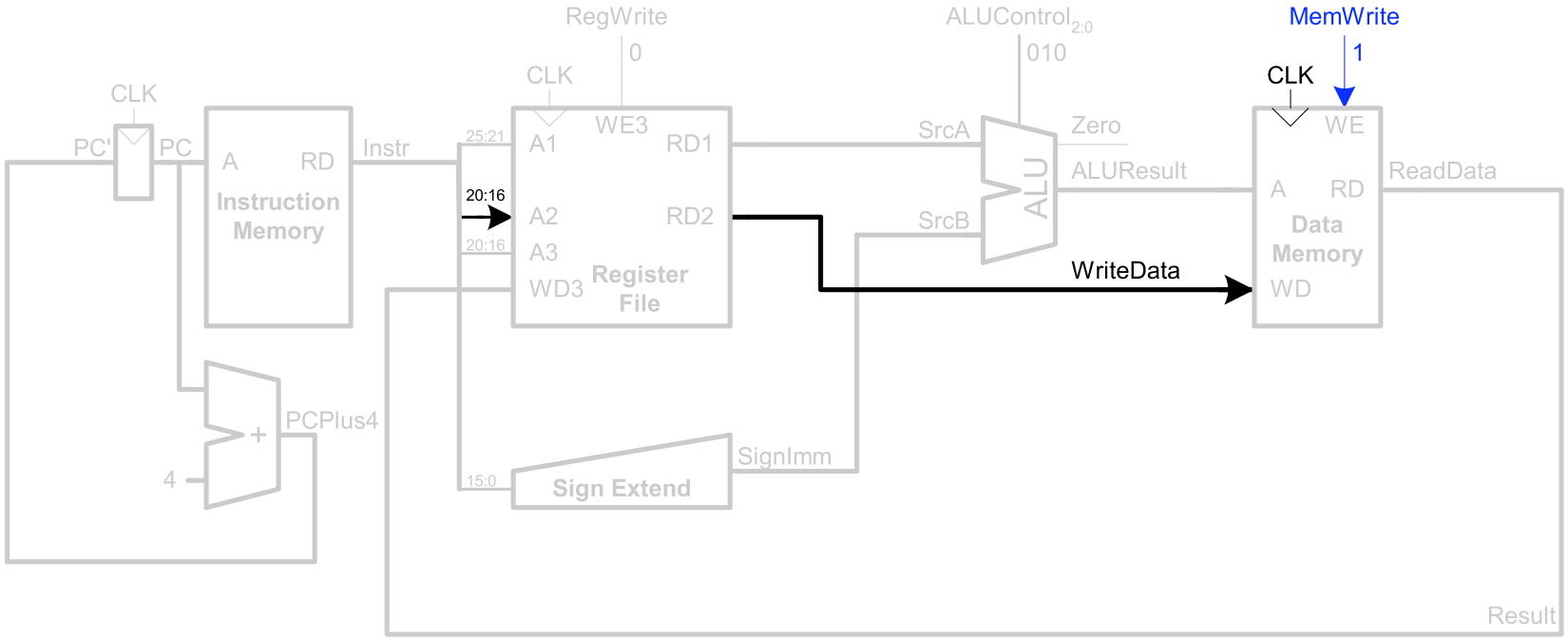
PC ← PC + 4



Single-Cycle Datapath: sw

DMEM[R[rs] + sign_ext(Imm16)] ← R[rt]

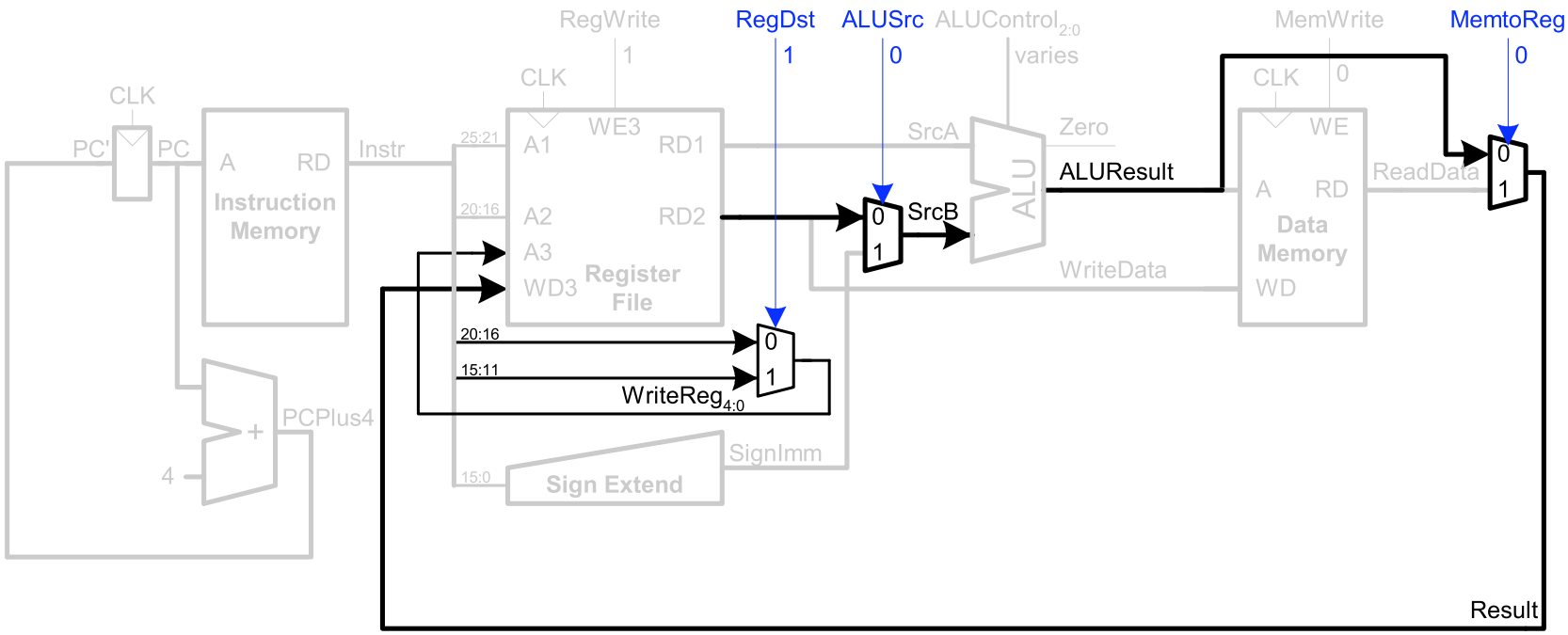
- Write data in `rt` to memory



Single-Cycle Datapath: R-type instructions

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

R[rd] ← R[rs] op R[rt]

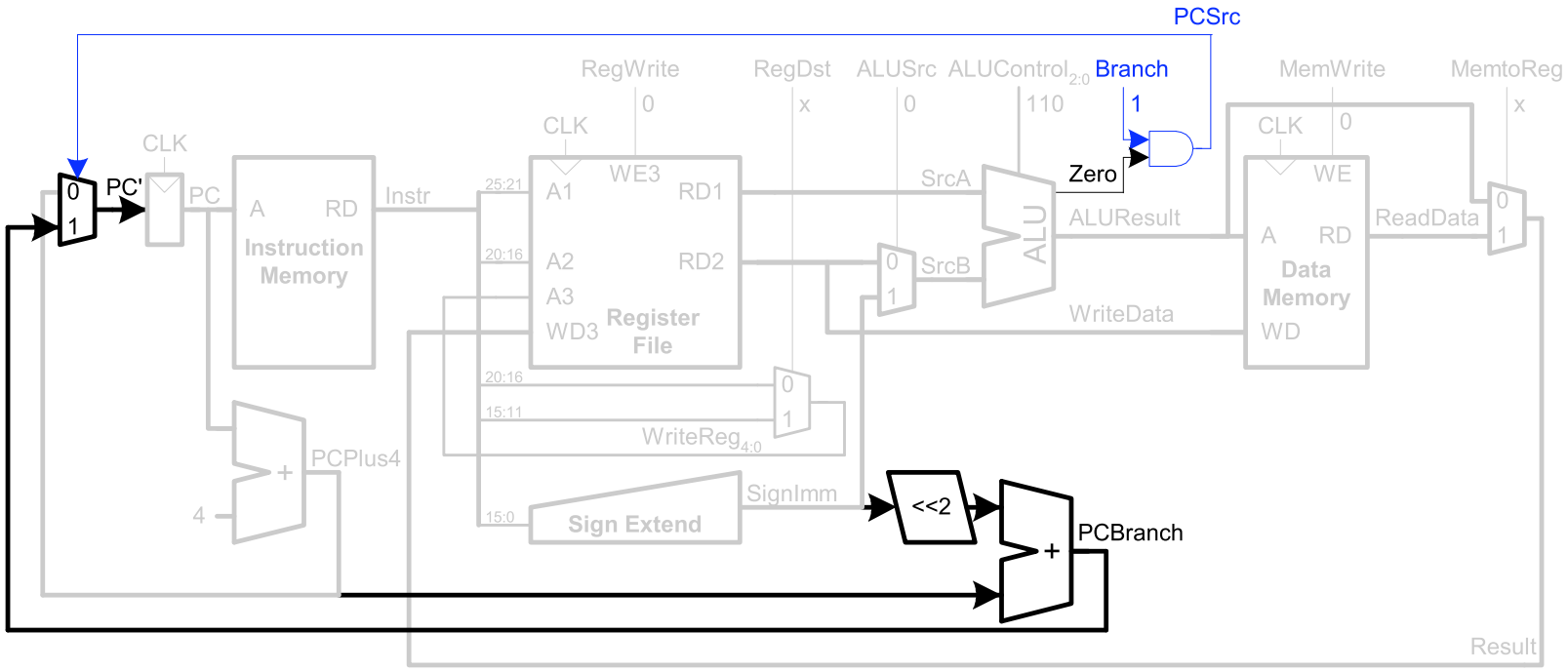


Single-Cycle Datapath: beq

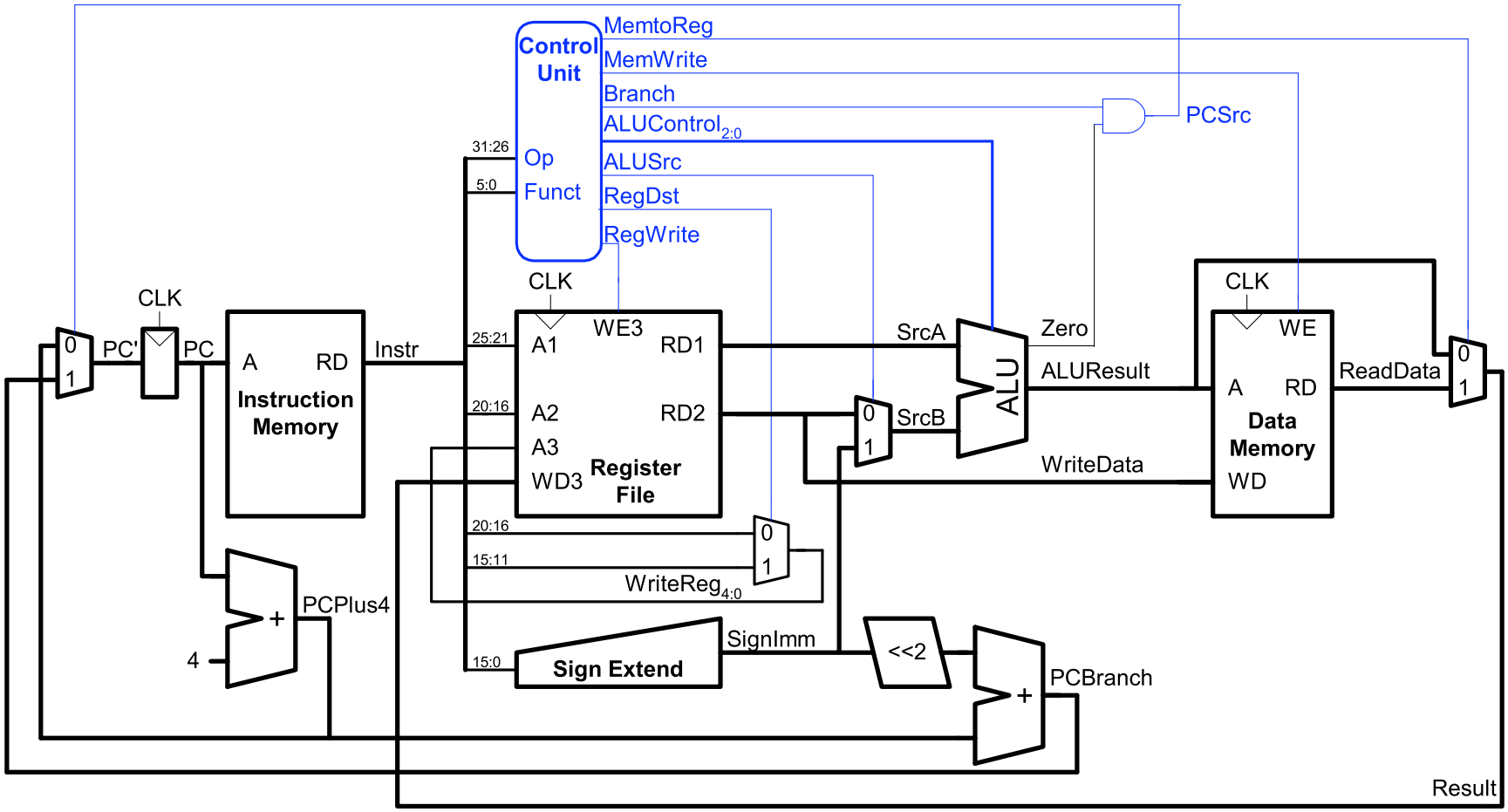
if (R[rs] == R[rt]) then PC ← PC + 4 + {sign_ext(Imm16), 00}

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

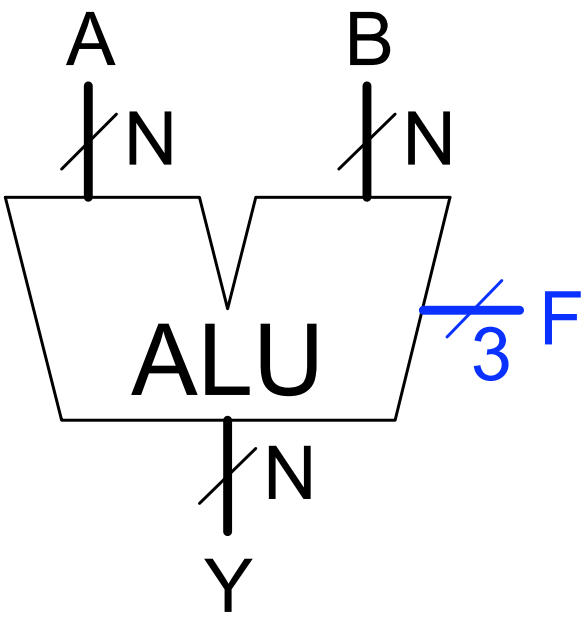
  BTA = (sign-extended immediate << 2) + (PC+4)



Complete Single-Cycle Processor

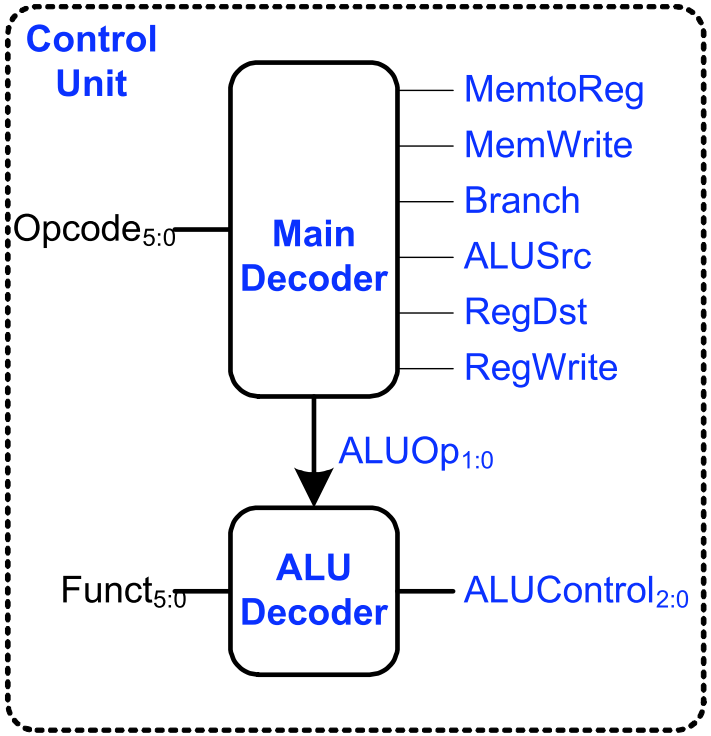


Review: ALU



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

Control Unit



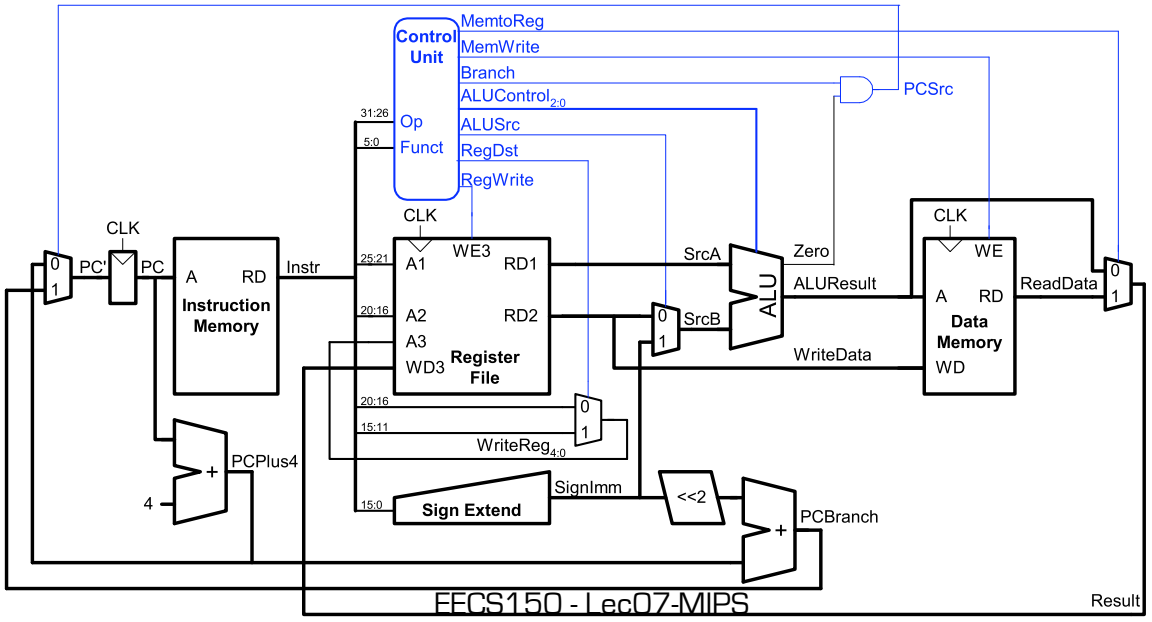
Control Unit: ALU Decoder

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

$ALUOp_{1:0}$	Funct	$ALUControl_{2:0}$
00	XXXXXX	010 (Add)
X1	XXXXXX	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)

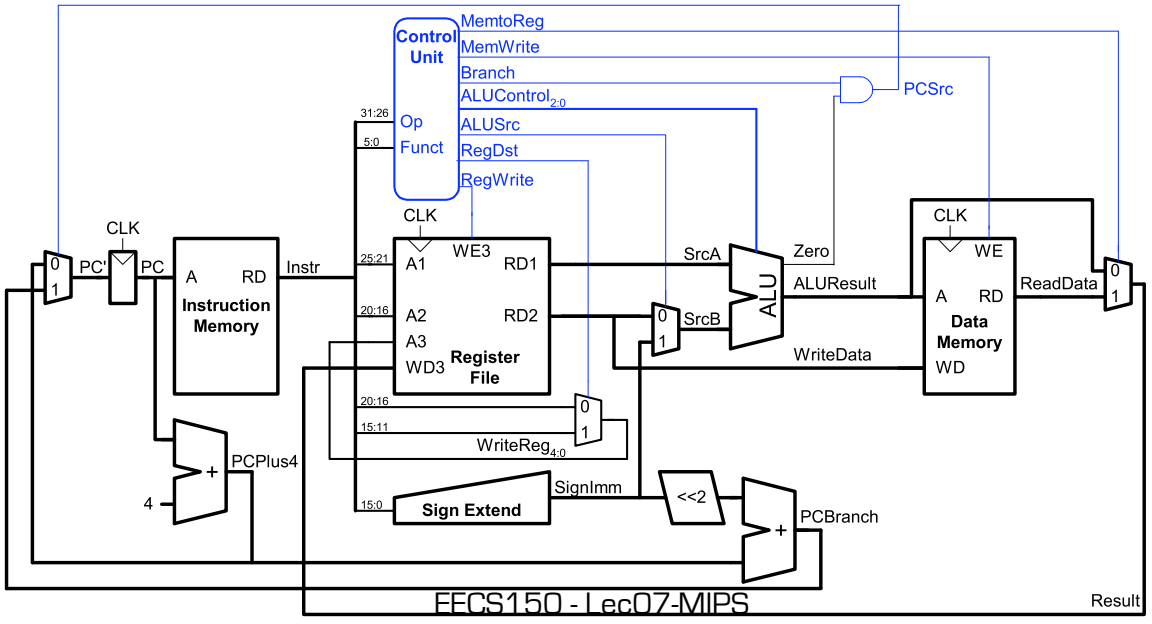
Control Unit: Main Decoder

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

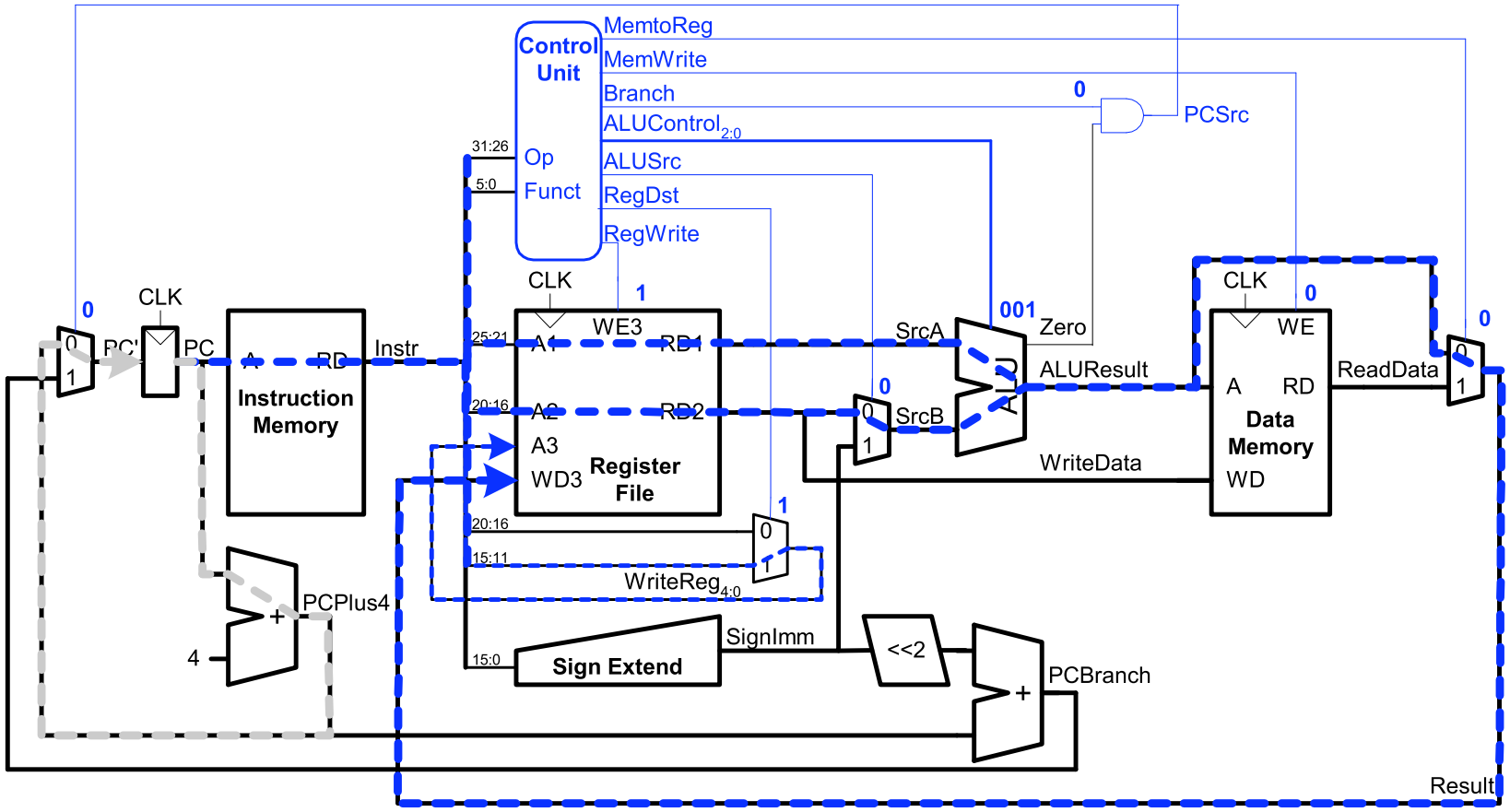


Control Unit: Main Decoder

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

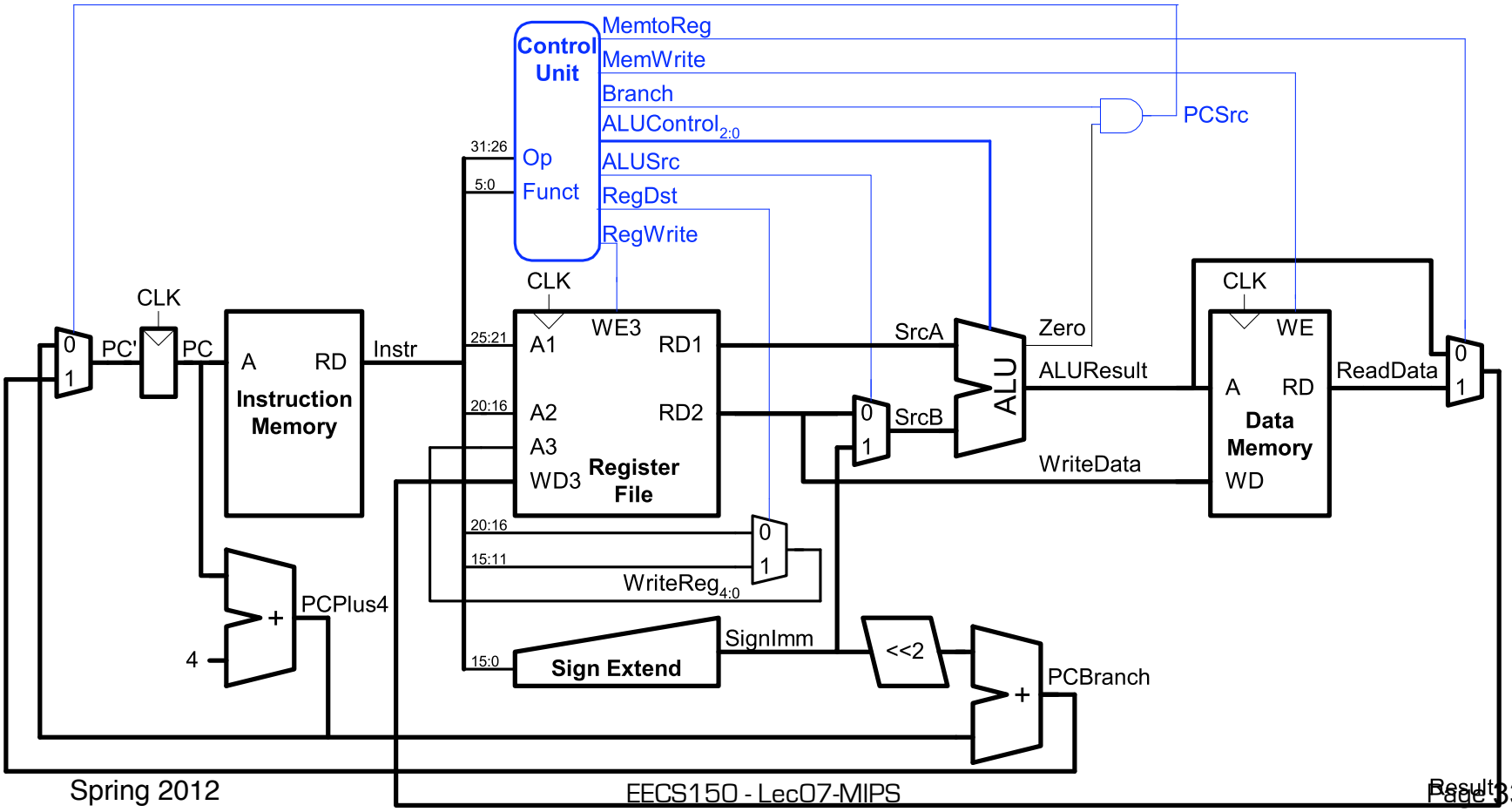


Single-Cycle Datapath Example: `or`



Extended Functionality: addi

- No change to datapath



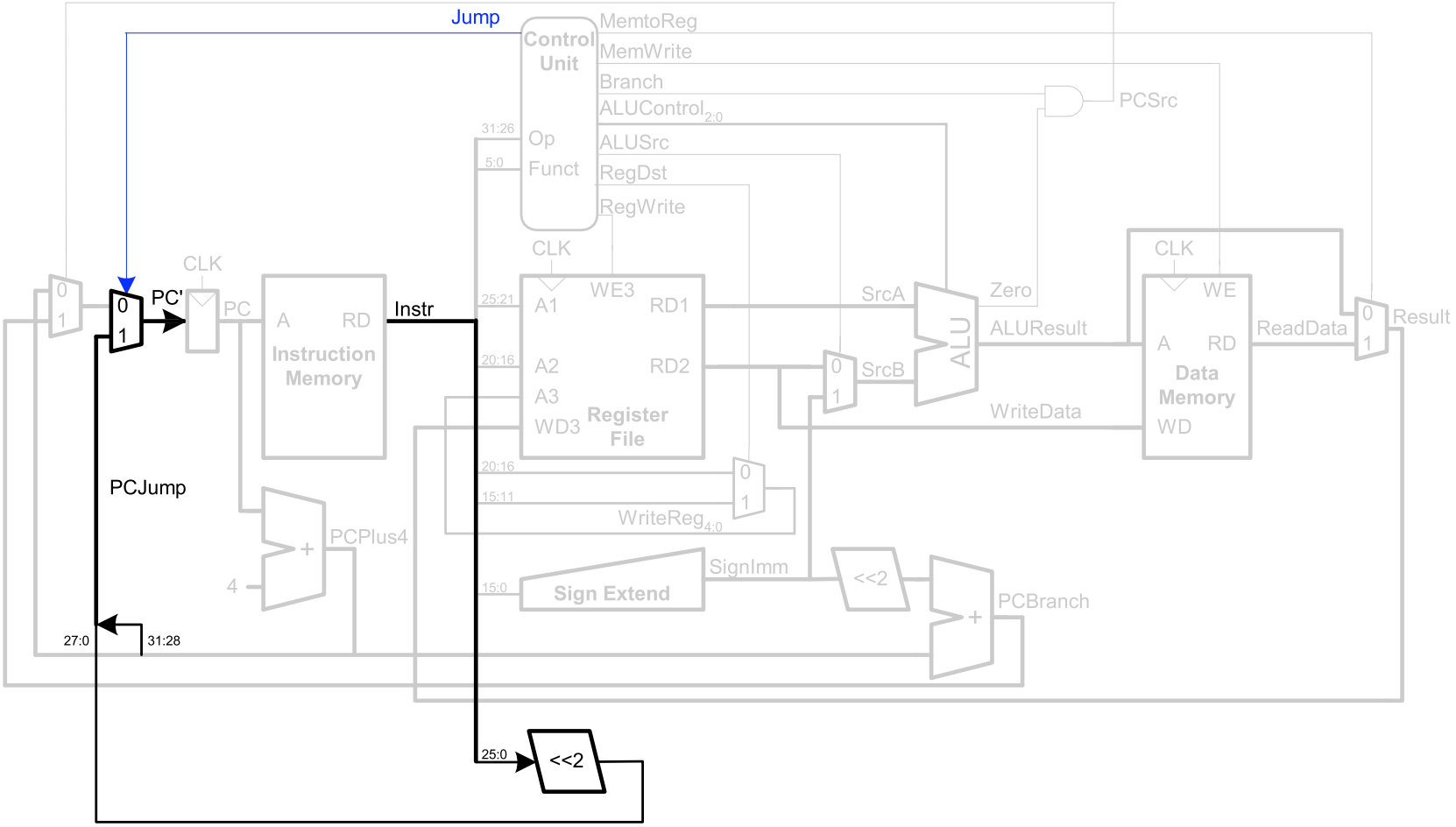
Control Unit: addi

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	0	1	1	0	0	0	0	10	0
`lw`	100011	1	0	1	0	0	1	0	0
`sw`	101011	0	X	1	0	1	X	0	0
`beq`	100	0	X	0	1	0	X	1	0
`j`	100	0	X	X	X	0	X	XX	1

Review: Processor Performance

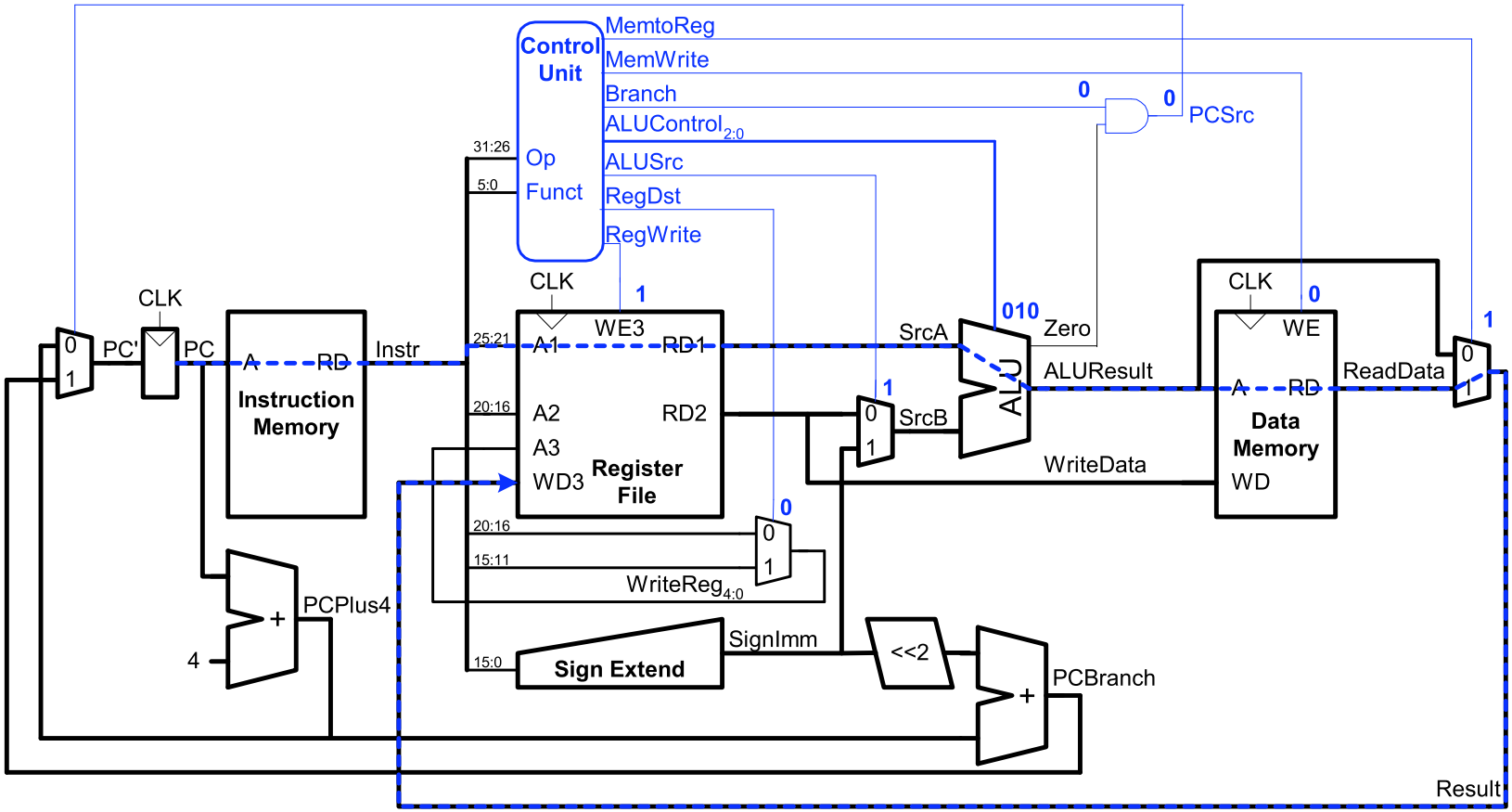
Program Execution Time

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

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_{RFread}	150
Register file setup	$t_{RFsetup}$	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_{RFread}	150
Register file setup	$t_{RFsetup}$	20

$$T_c = t_{pcq\_PC} + 2t_{mem} + t_{RFread} + t_{mux} + t_{ALU} + t_{RFsetup}$$
$$= [30 + 2(250) + 150 + 25 + 200 + 20] \text{ ps}$$
$$= 925 \text{ ps}$$

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,

Execution Time = # instructions x CPI x T_C

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

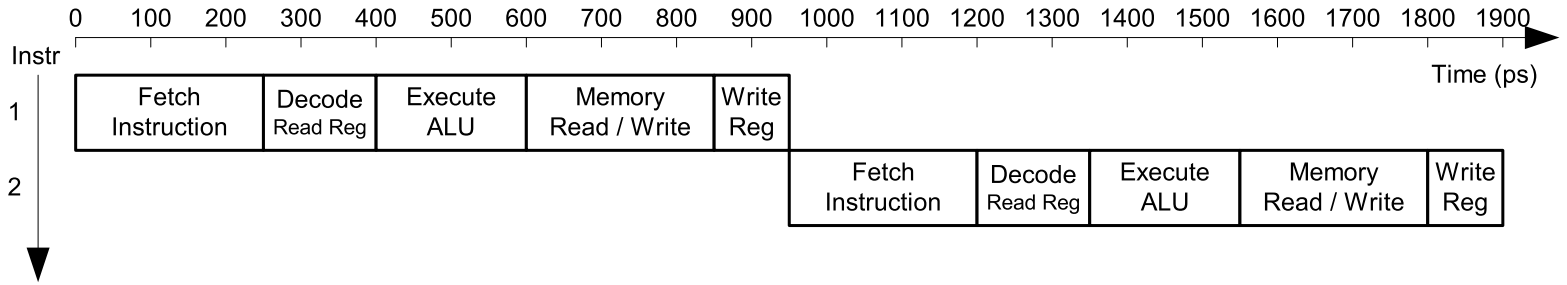
$= 92.5$ seconds

Pipelined MIPS Processor

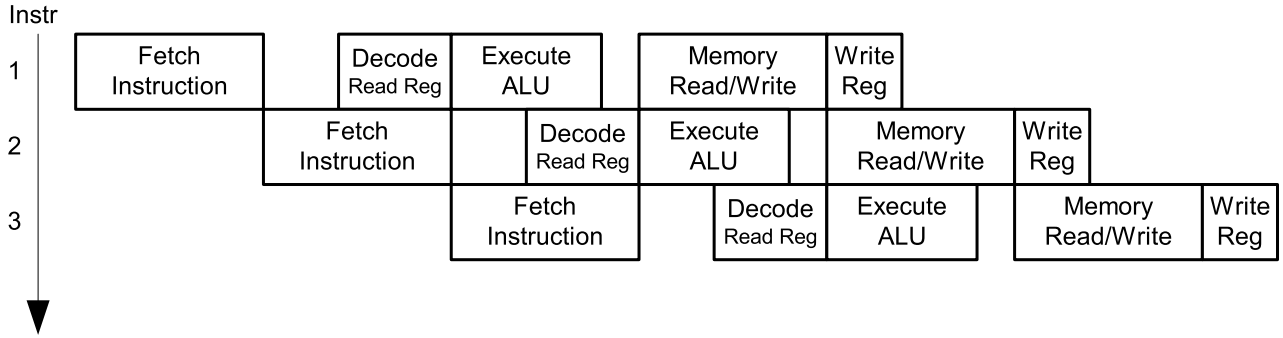
- Temporal parallelism
- Divide single-cycle processor into 5 stages:
  - Fetch
  - Decode
  - Execute
  - Memory
  - Writeback
- Add pipeline registers between stages

Single-Cycle vs. Pipelined Performance

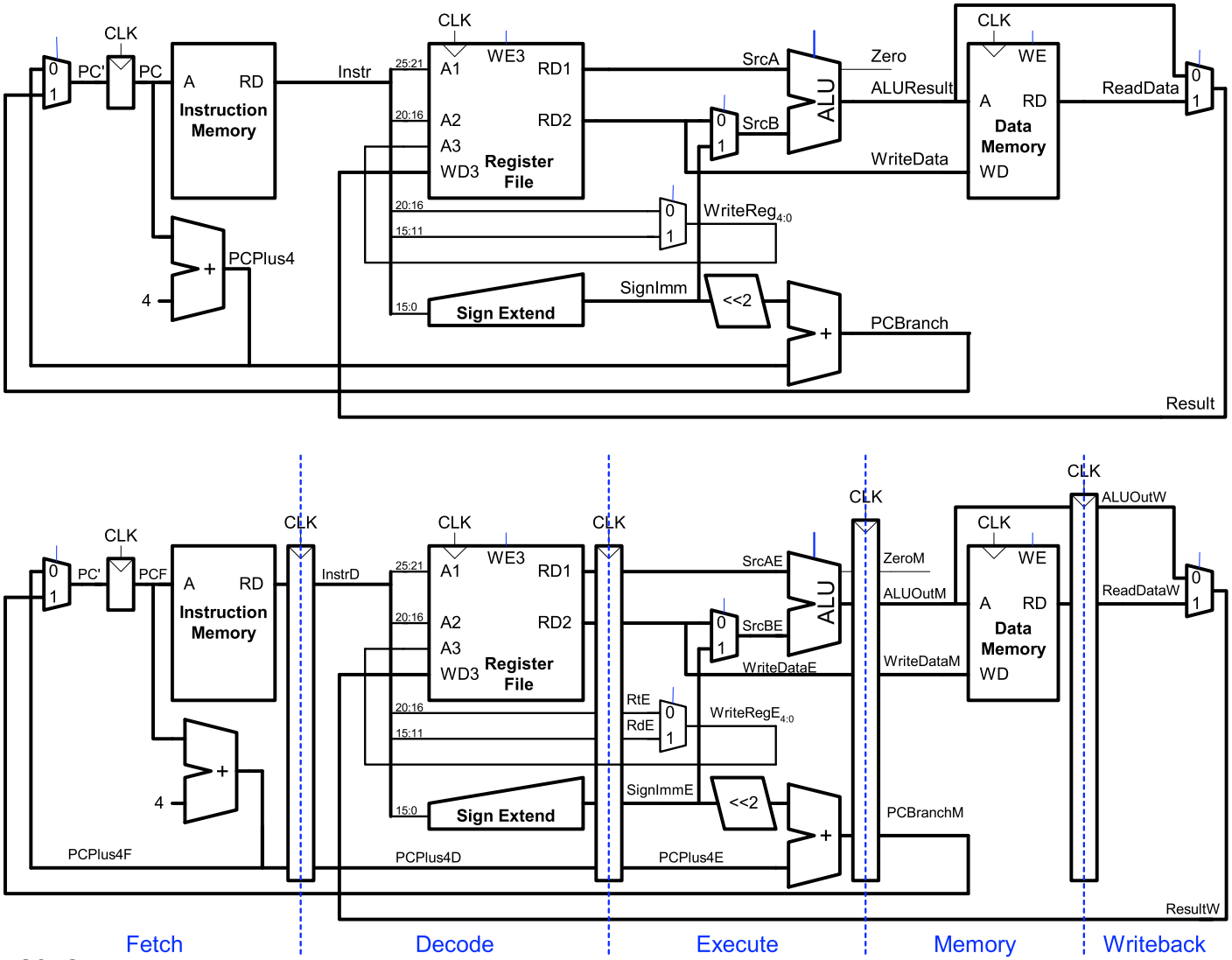
Single-Cycle



Pipelined

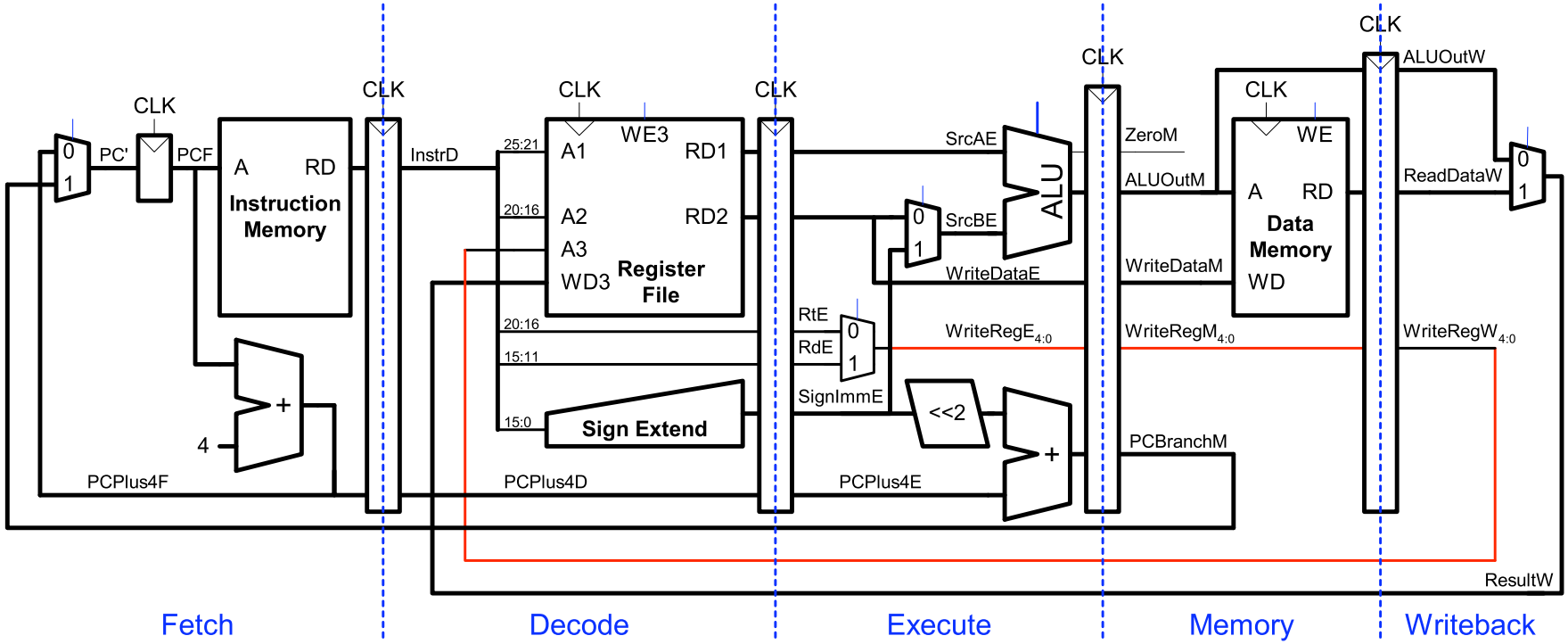


Single-Cycle and Pipelined Datapath

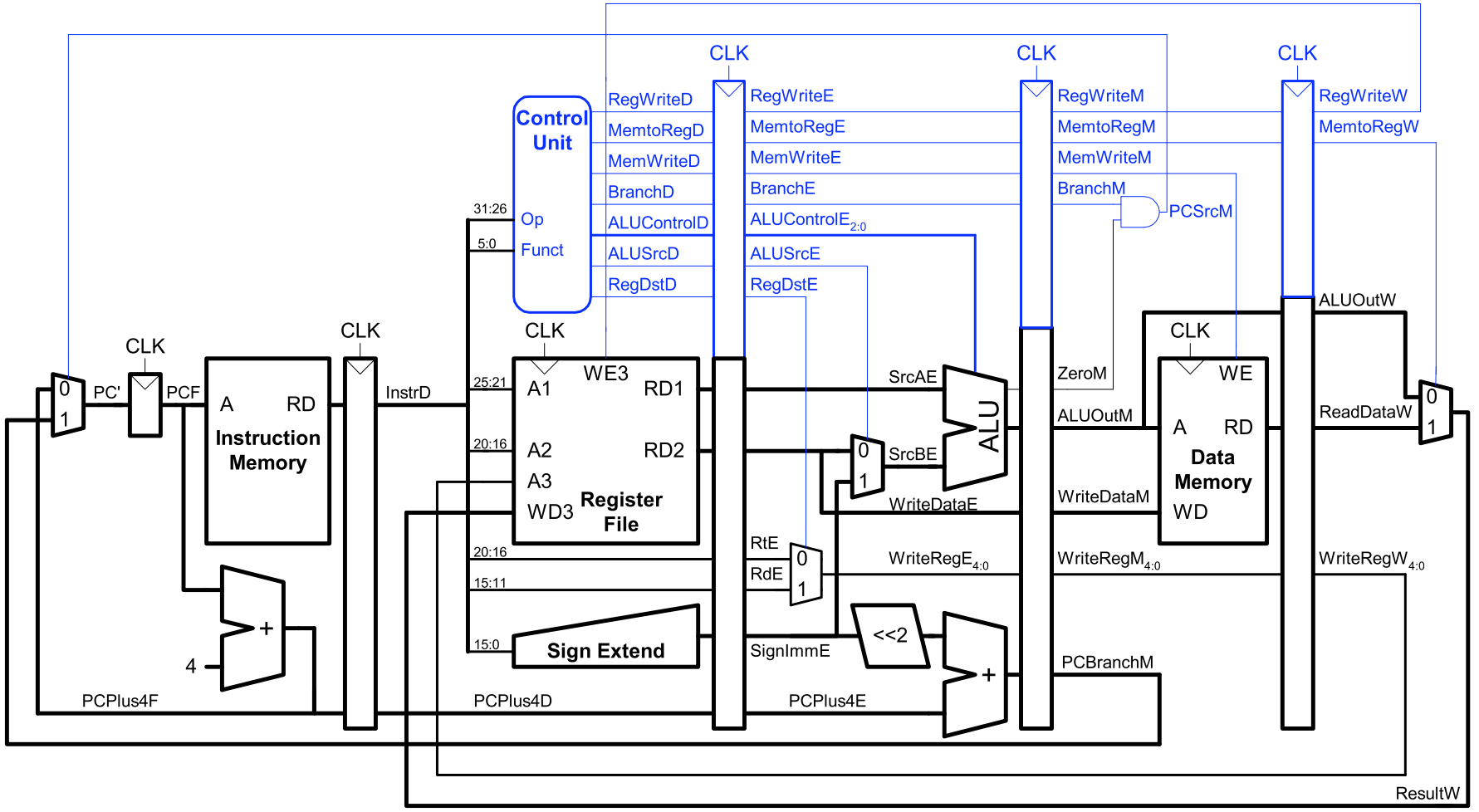


Corrected Pipelined Datapath

- 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 Hazards

- 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)

Pipelining Abstraction