CS222: Computer Architecture

Instructors:
Dr Ahmed Shalaby http://bu.edu.eg/staff/ahmedshalaby14#

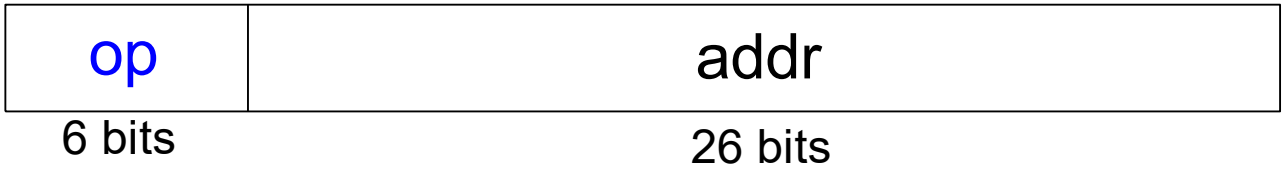
الاحترام - الادب - الاخلاق
الطالب - المعيد - الدكتور

Review: Instruction Formats

R-Type

I-Type

J-Type



R-Type

- *Register-type*
- 3 register operands:
  - `rs`, `rt`: source registers
  - `rd`: destination register
- Other fields:
  - `op`: the *operation code* or *opcode* (0 for R-type instructions)
  - `funct`: the *function* with opcode, tells computer what operation to perform
  - `shamt`: the *shift amount* for shift instructions, otherwise it's 0

R-Type

R-Type Examples

Table B.2 R-type instructions, sorted by funct field—Cont'd

Funct	Name	Description	Operation
100000 (32)	add rd, rs, rt	add	[rd] = [rs] + [rt]
100001 (33)	addu rd, rs, rt	add unsigned	[rd] = [rs] + [rt]
100010 (34)	sub rd, rs, rt	subtract	[rd] = [rs] - [rt]
100011 (35)	subu rd, rs, rt	subtract unsigned	[rd] = [rs] - [rt]

Assembly Code

```
add $s0, $s1, $s2
sub $t0, $t3, $t5
```

Field Values

op	rs	rt	rd	shamt	funct
0	17	18	16	0	32
0	11	13	8	0	34
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Machine Code

op	rs	rt	rd	shamt	funct	
000000	10001	10010	10000	00000	100000	(0x02328020)
000000	01011	01101	01000	00000	100010	(0x016D4022)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Name	Register
$0	0
$at	1
$v0-$v1	2-3
$a0-$a3	4-7
$t0-$t7	8-15
$s0-$s7	16-23
$t8-$t9	24-25

Note the order of registers in the assembly code:

```
add rd, rs, rt
```

I-Type

- *Immediate-type*
- 3 operands:
  - `rs`, `rt`: register operands
  - `imm`: 16-bit two's complement immediate
- Other fields:
  - `op`: the opcode
  - Simplicity favors regularity: all instructions have opcode
  - Operation is completely determined by opcode

I-Type

I-Type Examples

100011 (35)	lw rt, imm(rs)	load word	[rt] = [Address]
101011 (43)	sw rt, imm(rs)	store word	[Address] = [rt]
001000 (8)	addi rt, rs, imm	add immediate	[rt] = [rs] + SignImm

Name	Regist
$0	0
$at	1
$v0-$v1	2-3
$a0-$a3	4-7
$t0-$t7	8-15
$s0-$s7	16-23
$t8-$t9	24-25

Assembly Code

```
addi $s0, $s1, 5
addi $t0, $s3, -12
lw   $t2, 32($0)
sw   $s1,  4($t1)
```

Field Values

op	rs	rt	imm
8	17	16	5
8	19	8	-12
35	0	10	32
43	9	17	4
6 bits	5 bits	5 bits	16 bits

Note the differing order of registers in assembly and machine codes:

```
addi rt, rs, imm
lw   rt, imm(rs)
sw   rt, imm(rs)
```

Machine Code

op	rs	rt	imm	
001000	10001	10000	0000 0000 0000 0101	(0x22300005)
001000	10011	01000	1111 1111 1111 0100	(0x2268FFF4)
100011	00000	01010	0000 0000 0010 0000	(0x8C0A0020)
101011	01001	10001	0000 0000 0000 0100	(0xAD310004)
6 bits	5 bits	5 bits	16 bits	

I-Type Examples

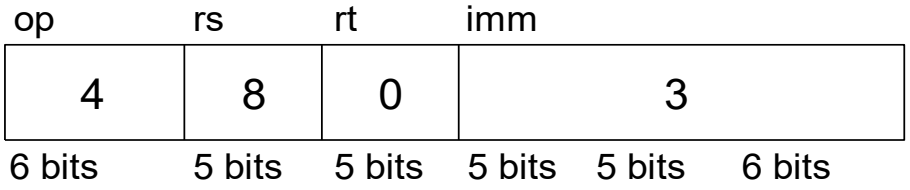
PC-Relative Addressing

```
0x10                beq   $t0, $0, else
0x14                addi  $v0, $0, 1
0x18                addi  $sp, $sp, i
0x1C                jr          $ra
0x20        else:   addi  $a0, $a0, -1
0x24                jal   factorial
```

Assembly Code

```
beq $t0, $0, else
(beq $t0, $0, 3)
```

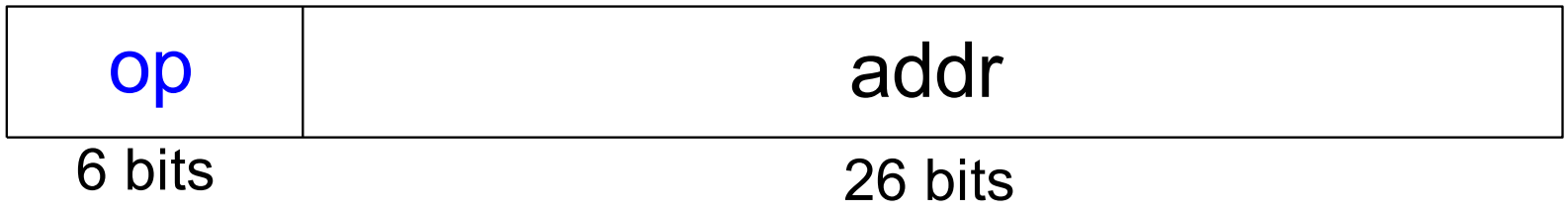
Field Values



J-Type

- *Jump-type*
- 26-bit address operand (`addr`)
- Used for jump instructions (`j`)

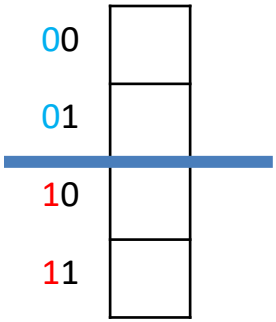
J-Type



000010 (2)	`j label`	jump	PC = JTA

J-Type Example

Pseudo-direct Addressing




```
0x0040005C          jal   sum
...
0x004000A0  sum:    add   $v0, $a0, $a1
```

JTA 0000 0000 0100 0000 0000 0000 1010 0000 (0x004000A0)

26-bit addr 0000 0000 0100 0000 0000 0000 1010 0000 (0x0100028)

0 1 0 0 0 2 8

0	1	2	3

Field Values

op	imm
3	0x0100028
6 bits	26 bits

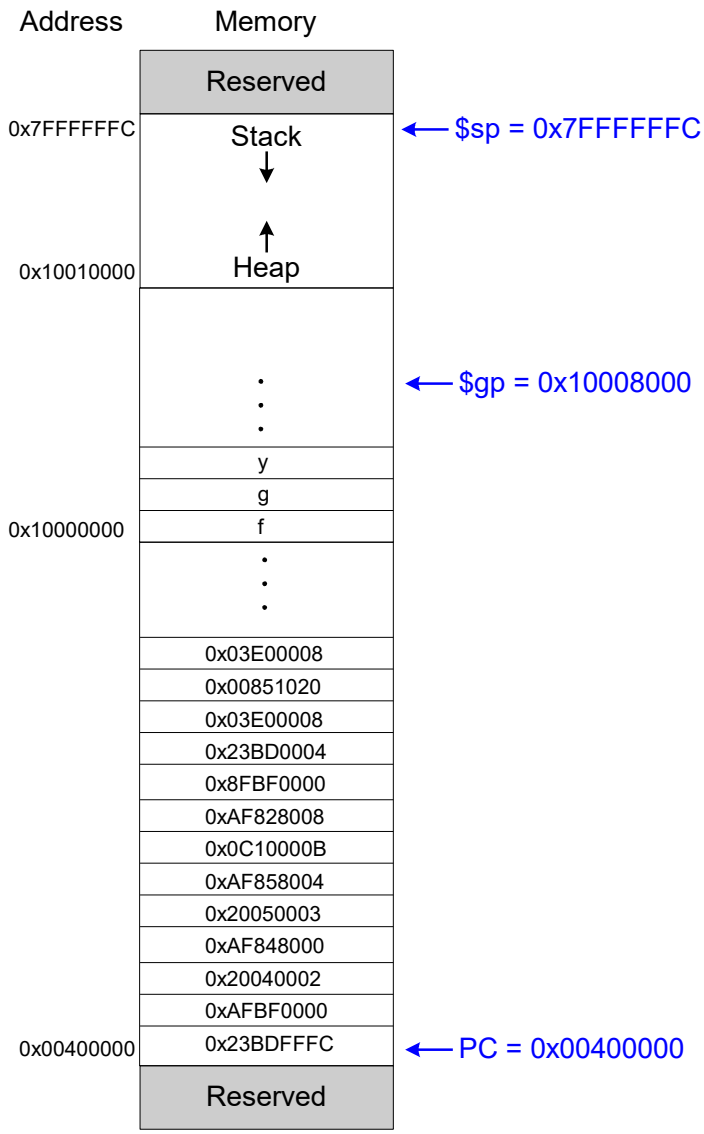
Machine Code

op	addr	
000011	00 0001 0000 0000 0000 0010 1000	(0x0C100028)
6 bits	26 bits	

000011 (3)	jal label	jump and link	$ra = PC + 4, PC = JTA

Example Program: In Memory

Instruction		Data
0x23BDFFFC	addi $sp, $sp, -4	f
0xAFBF0000	sw $ra, 0 ($sp)	g
0x20040002	addi $a0, $0, 2	y
0xAF848000	sw $a0, 0x8000 ($gp)	
0x20050003	addi $a1, $0, 3	
0xAF858004	sw $a1, 0x8004 ($gp)	
0x0C10000B	jal 0x0040002C	
0xAF828008	sw $v0, 0x8008 ($gp)	
0x8FBF0000	lw $ra, 0 ($sp)	
0x23BD0004	addi $sp, $sp, -4	
0x03E00008	jr $ra	
0x00851020	add $v0, $a0, $a1	
0x03E00008	jr $ra	



MICROARCHITECTURE

Chapter 7

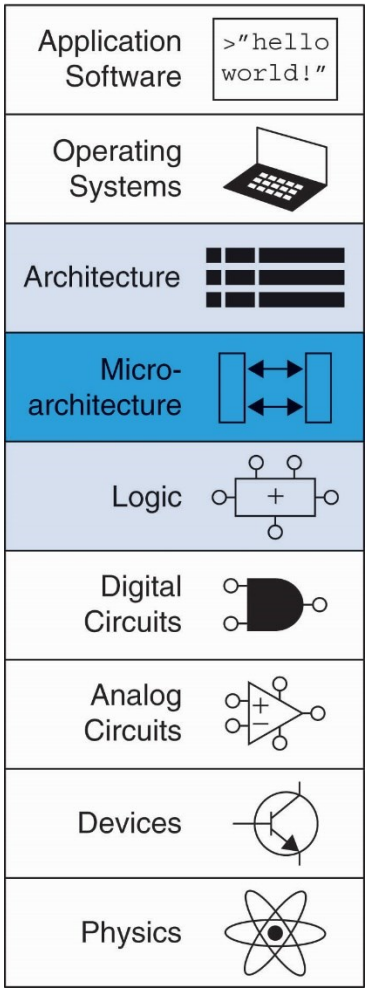
***Digital Design and Computer Architecture*, 2nd Edition**

David Money Harris and Sarah L. Harris

ELSEVIER

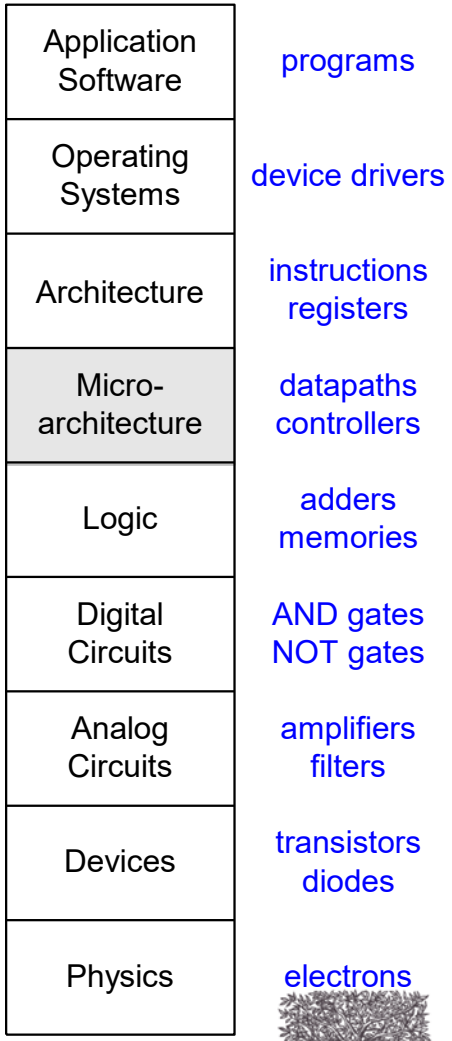
Chapter 7 :: Topics

- 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



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

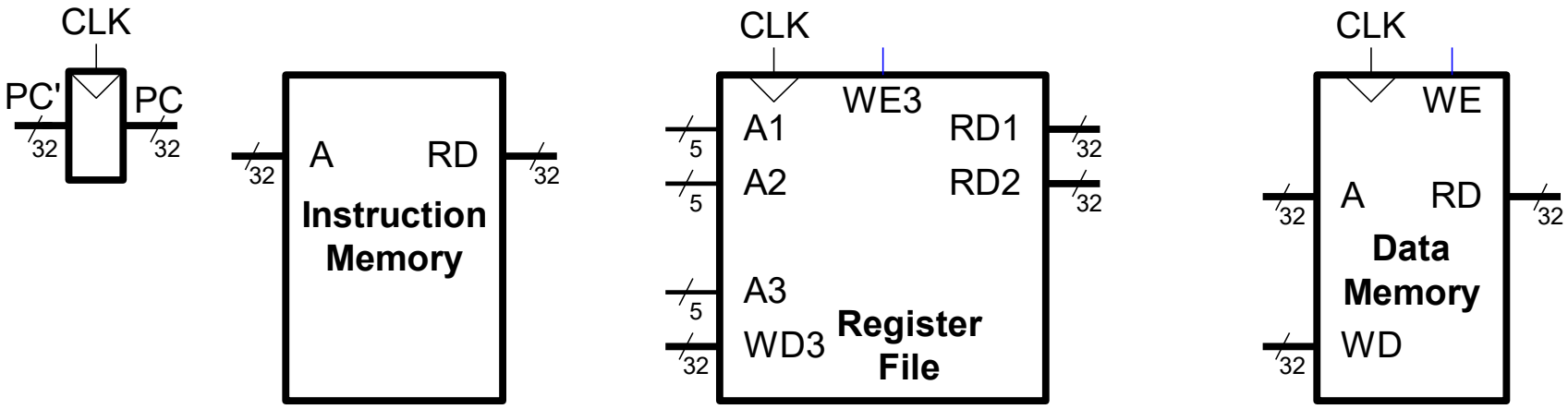
MIPS Processor

- Consider subset of MIPS instructions:
  - R-type instructions: `and`, `or`, `add`, `sub`, `slt`
  - Memory instructions: `lw`, `sw`
  - Branch instructions: `beq`

Architectural State

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

MIPS State Elements

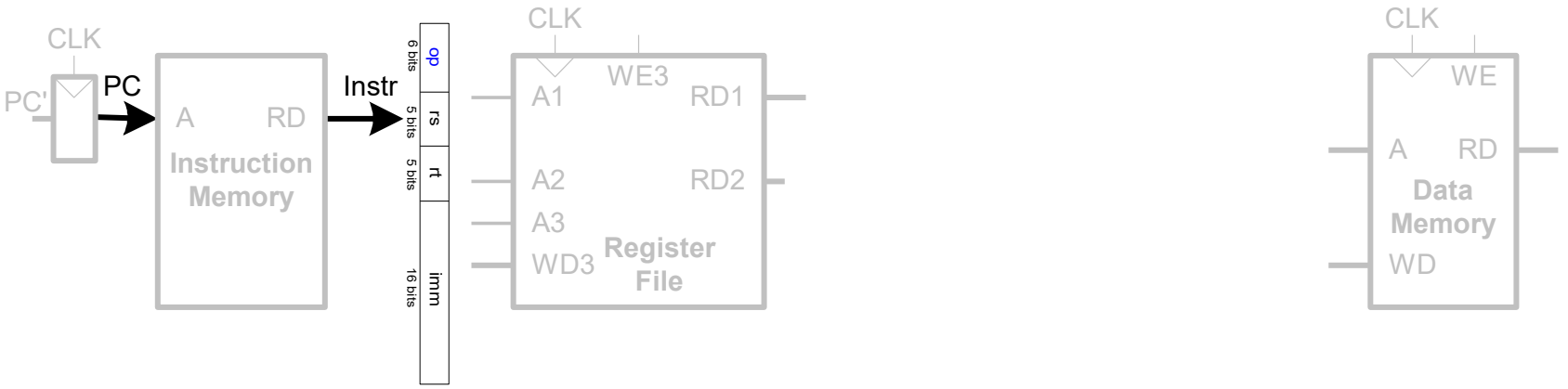


Single-Cycle MIPS Processor

- Datapath
- Control

Single-Cycle Datapath: `lw` fetch

STEP 1: Fetch instruction



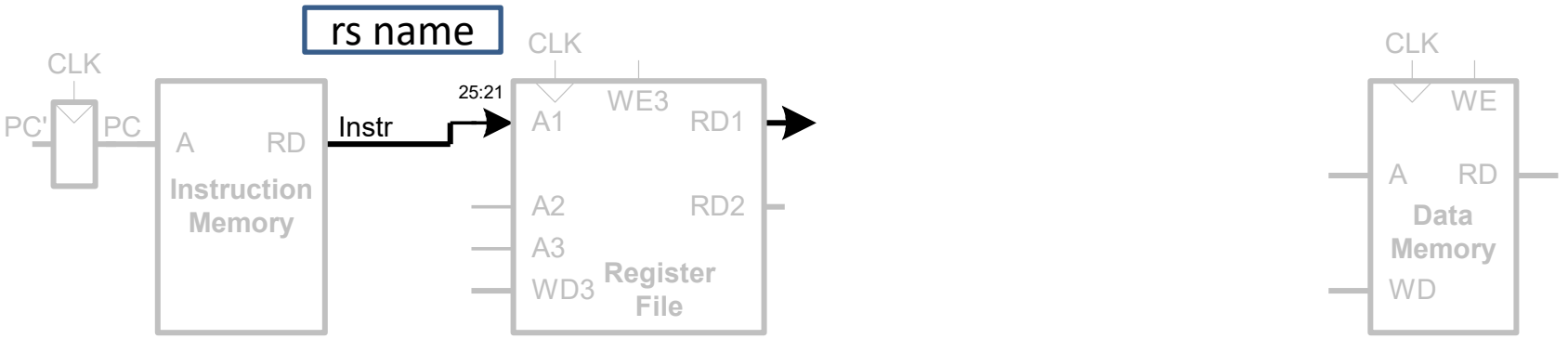
`lw rt, imm(rs)`

I-Type

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

Single-Cycle Datapath: `lw` Register Read

STEP 2: Read source operands from RF



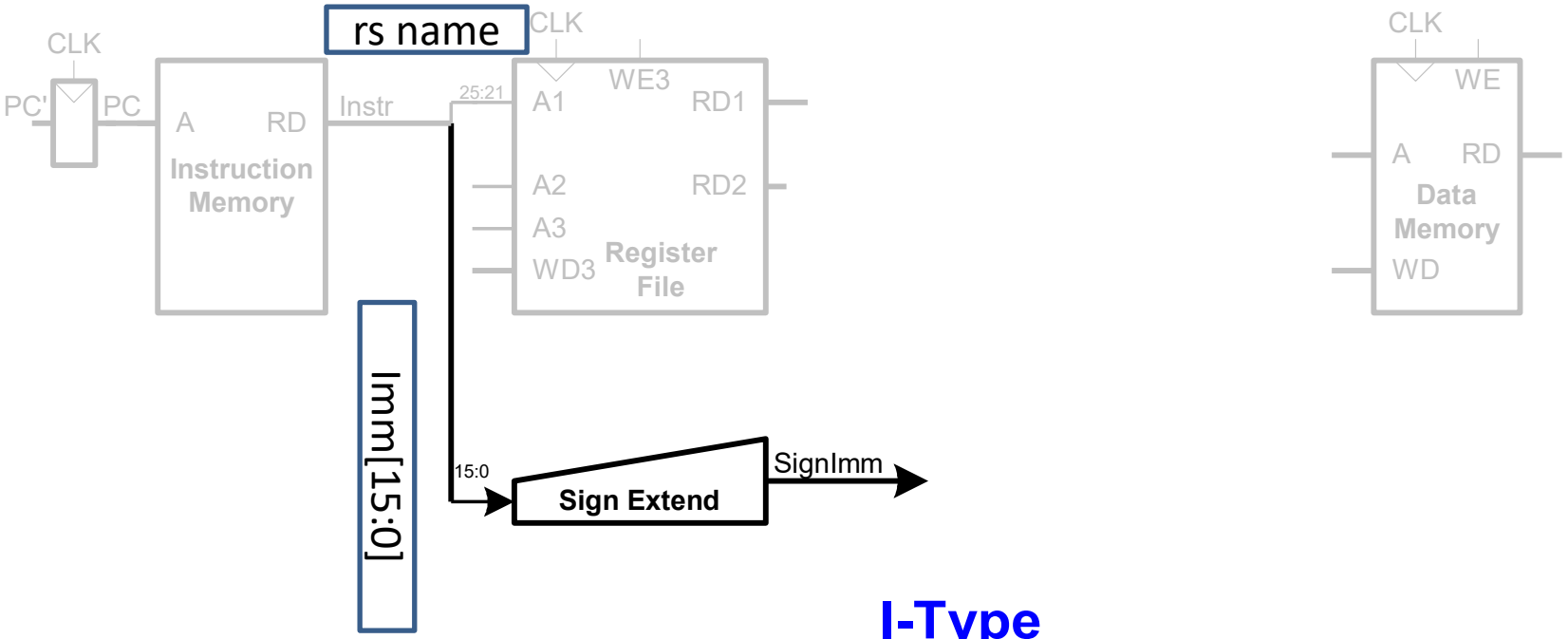
`lw rt, imm(rs)`

I-Type

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

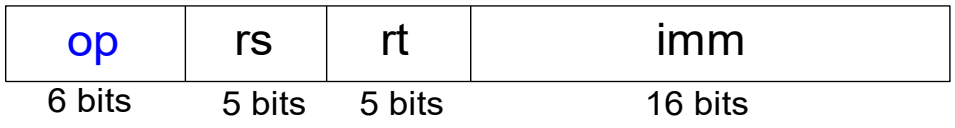
Single-Cycle Datapath: `lw` Immediate

STEP 3: Sign-extend the immediate



`lw rt, imm(rs)`

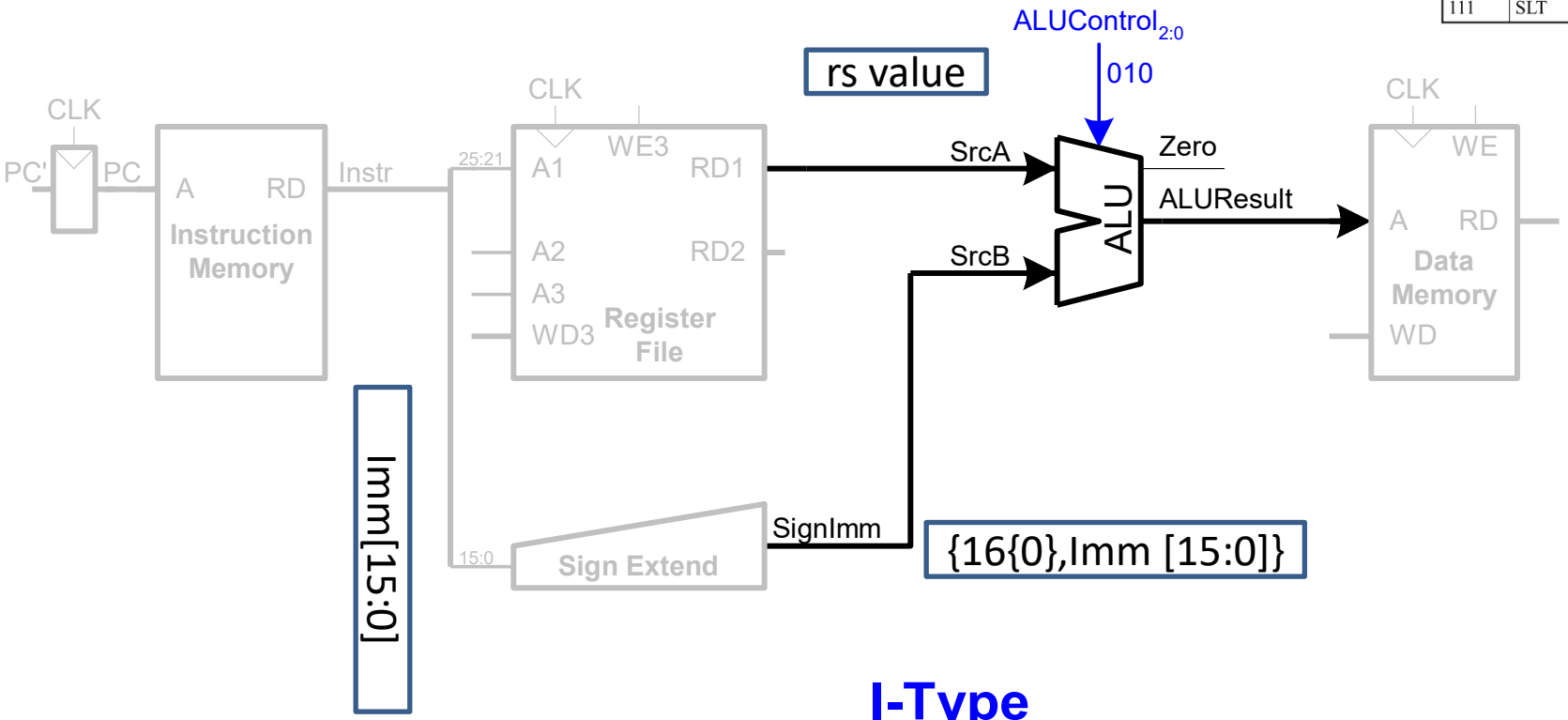
I-Type



Single-Cycle Datapath: `lw` address

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

STEP 4: Compute the memory address



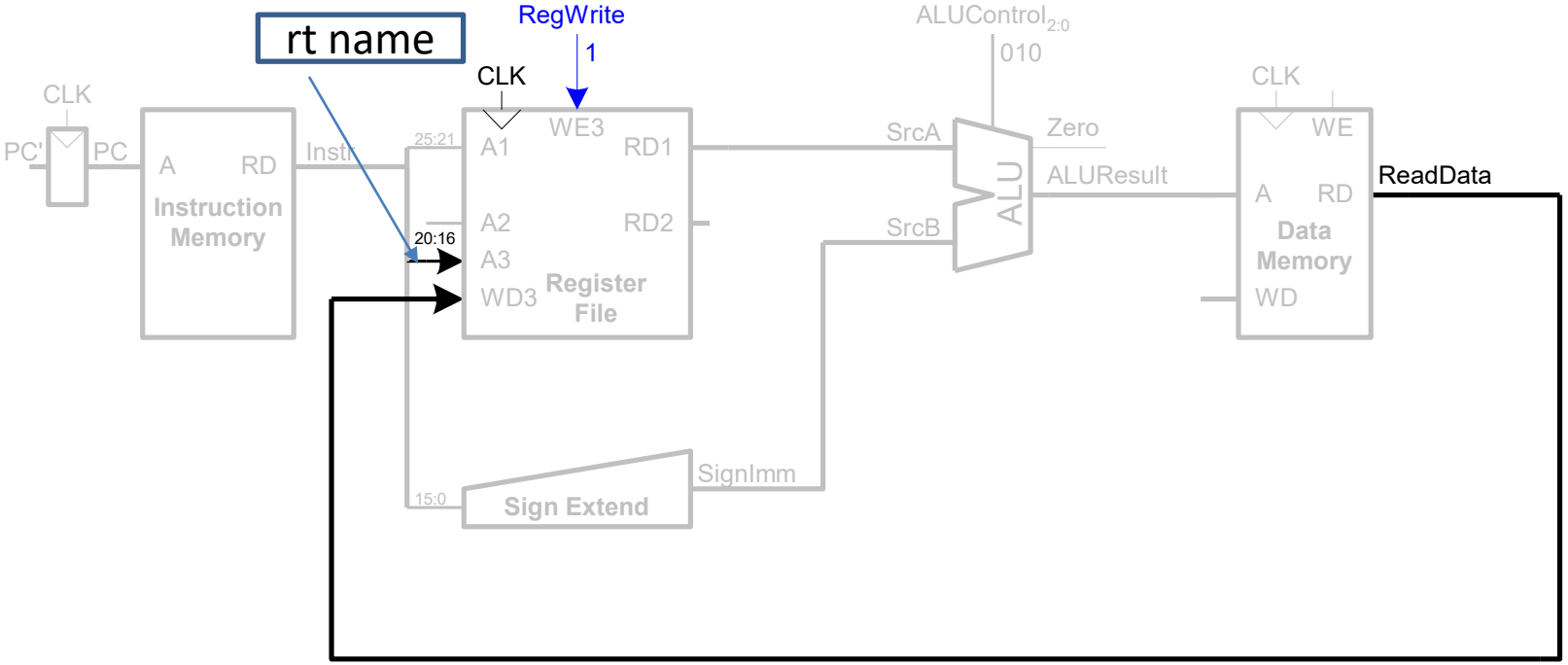
`lw rt, imm(rs)`

I-Type

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

Single-Cycle Datapath: `lw` Memory Read

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



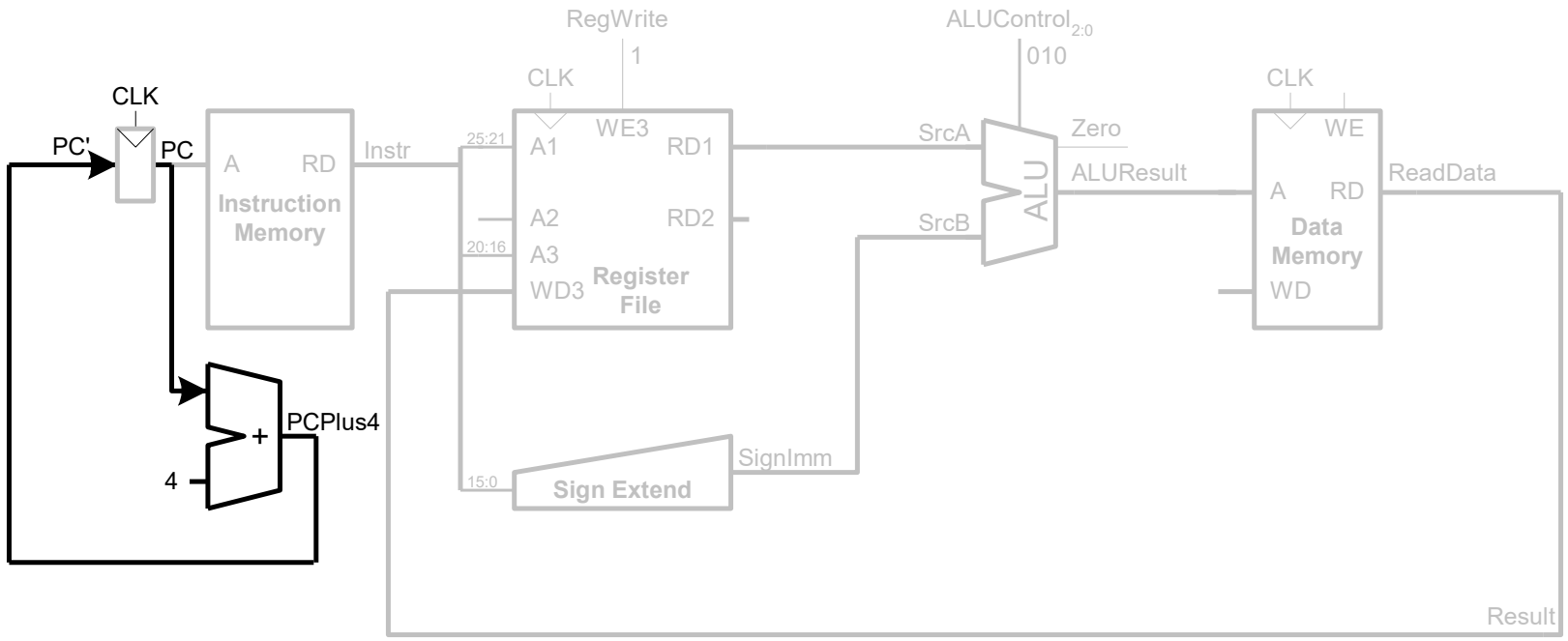
I-Type

`lw rt, imm(rs)`

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

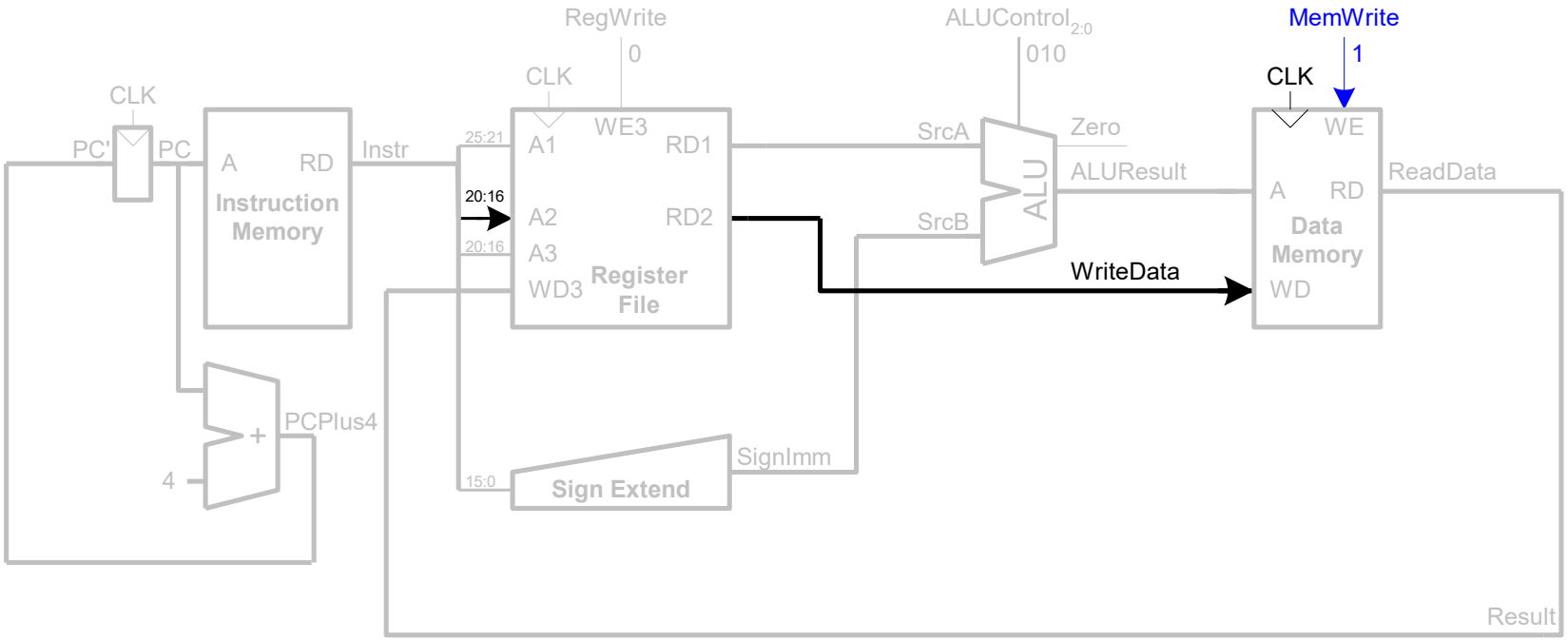
Single-Cycle Datapath: `lw` PC Increment

STEP 6: Determine address of next instruction



Single-Cycle Datapath: `sw`

Write data in `rt` to memory



I-Type

`sw rt, imm(rs)`

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

Single-Cycle Datapath: R-Type

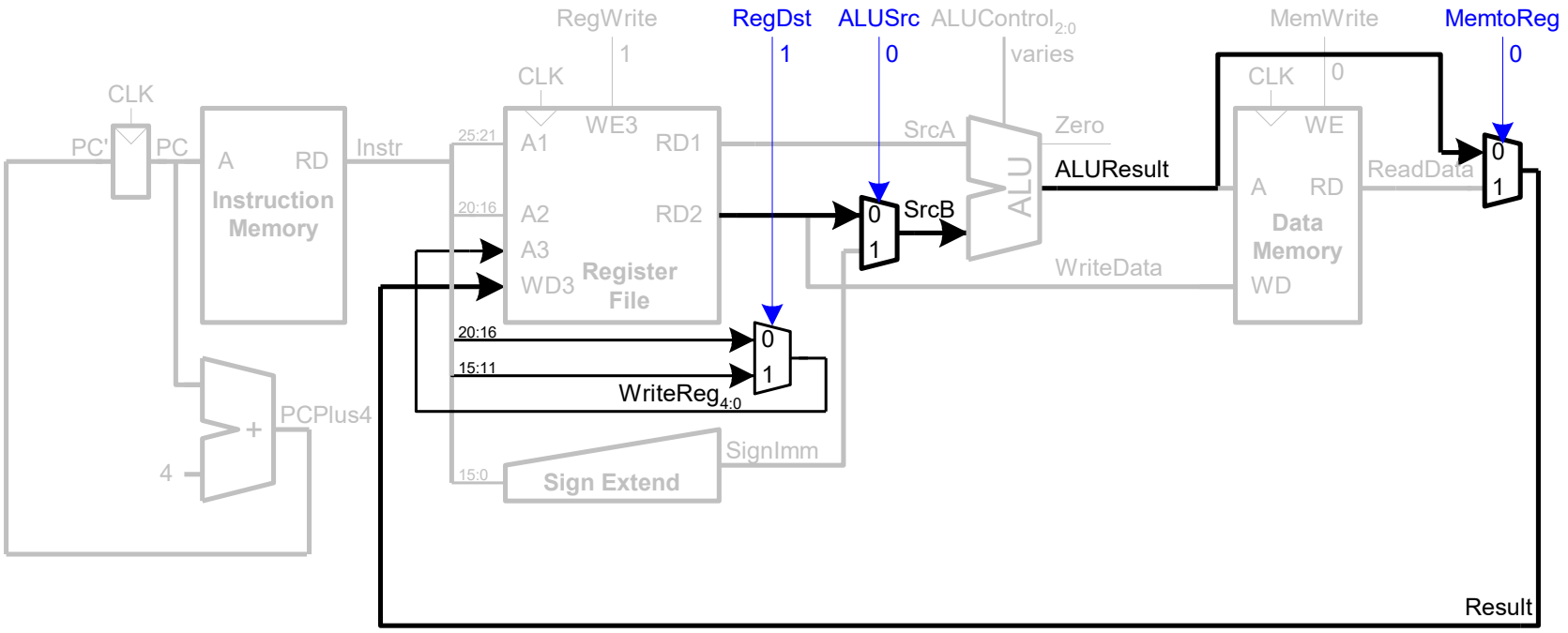
- Read from `rs` and `rt`
- Write *ALUResult* to register file
- Write to `rd` (instead of `rt`)

R-Type

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

I-Type

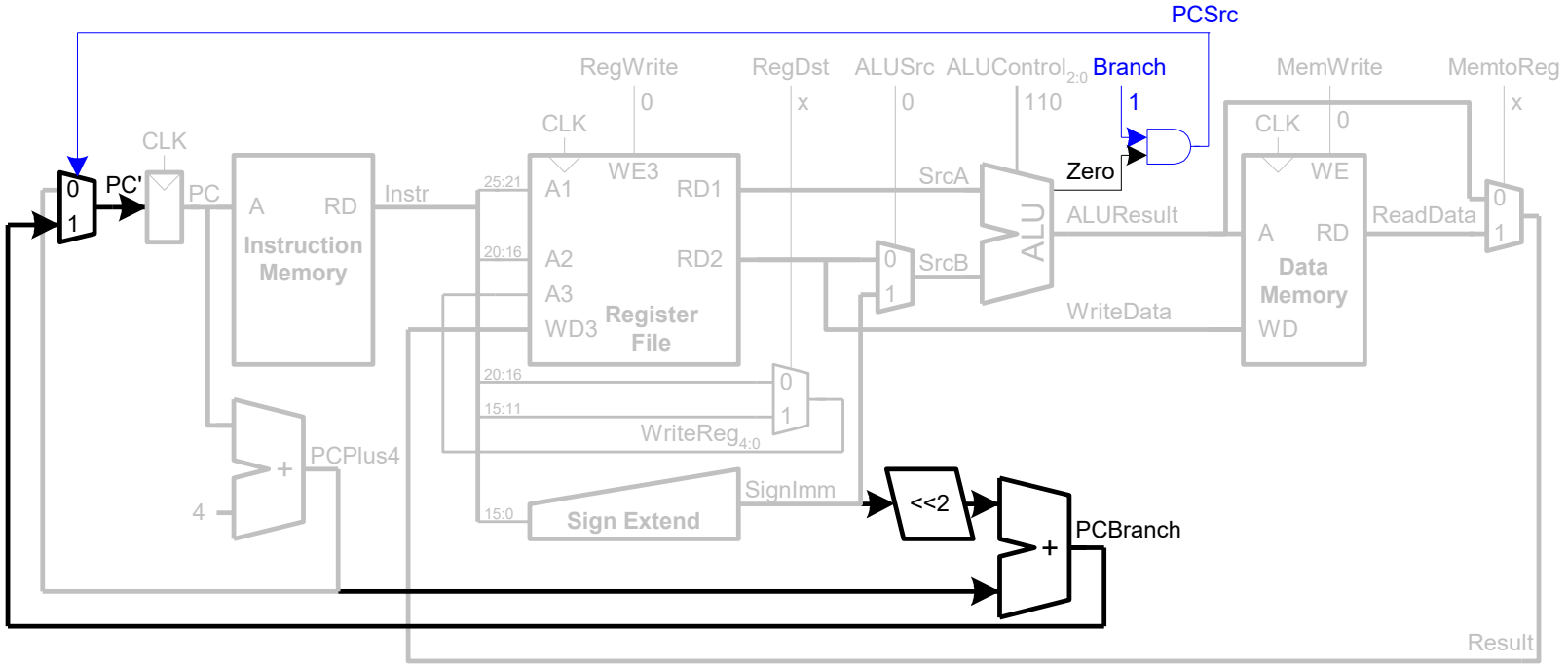
op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits



Single-Cycle Datapath: beq

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

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



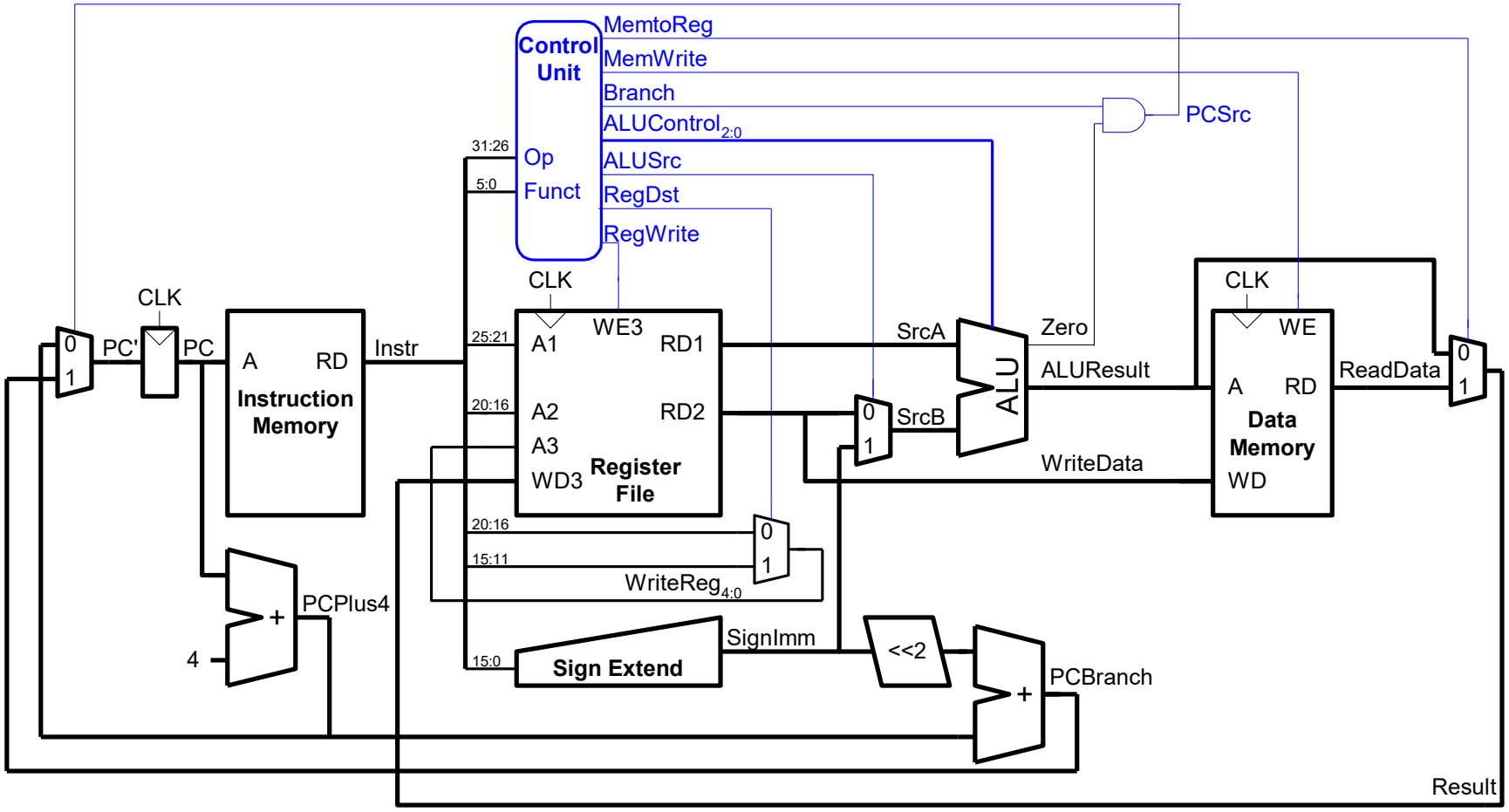
Assembly Code

```
beq $t0, $0, else
(beq $t0, $0, 3)
```

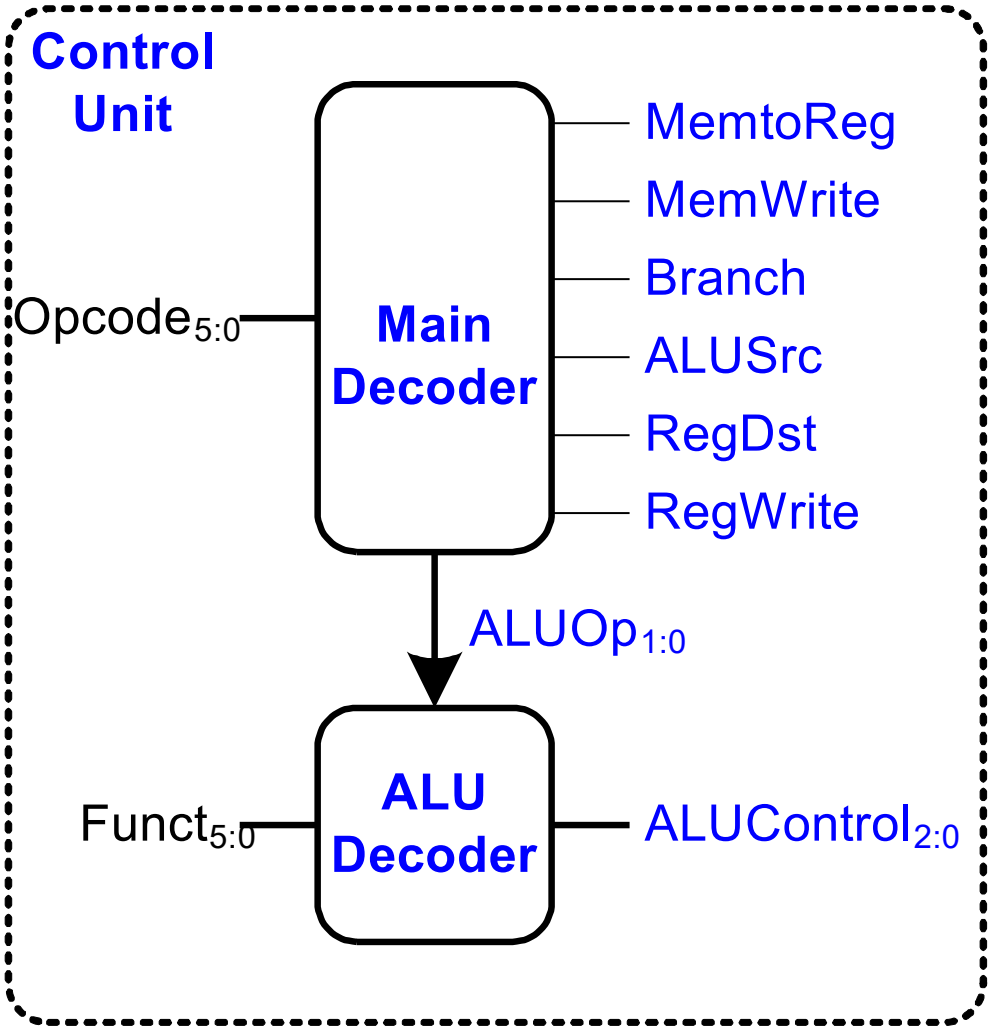
Field Values

op	rs	rt	imm		
4	8	0	3		
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

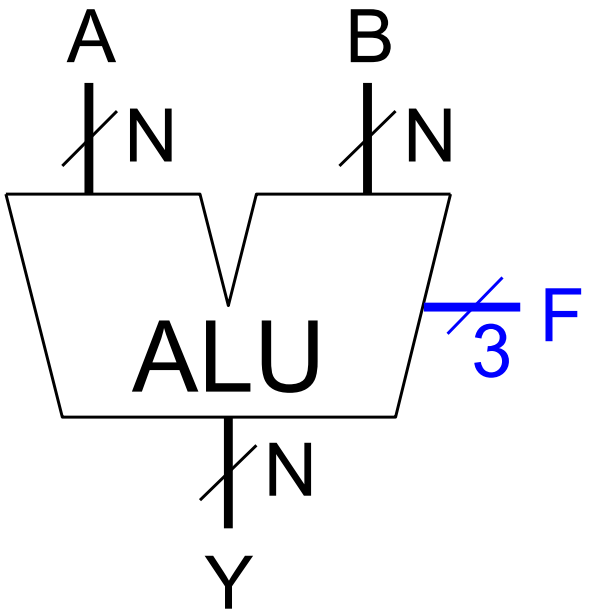
Single-Cycle Processor



Single-Cycle Control

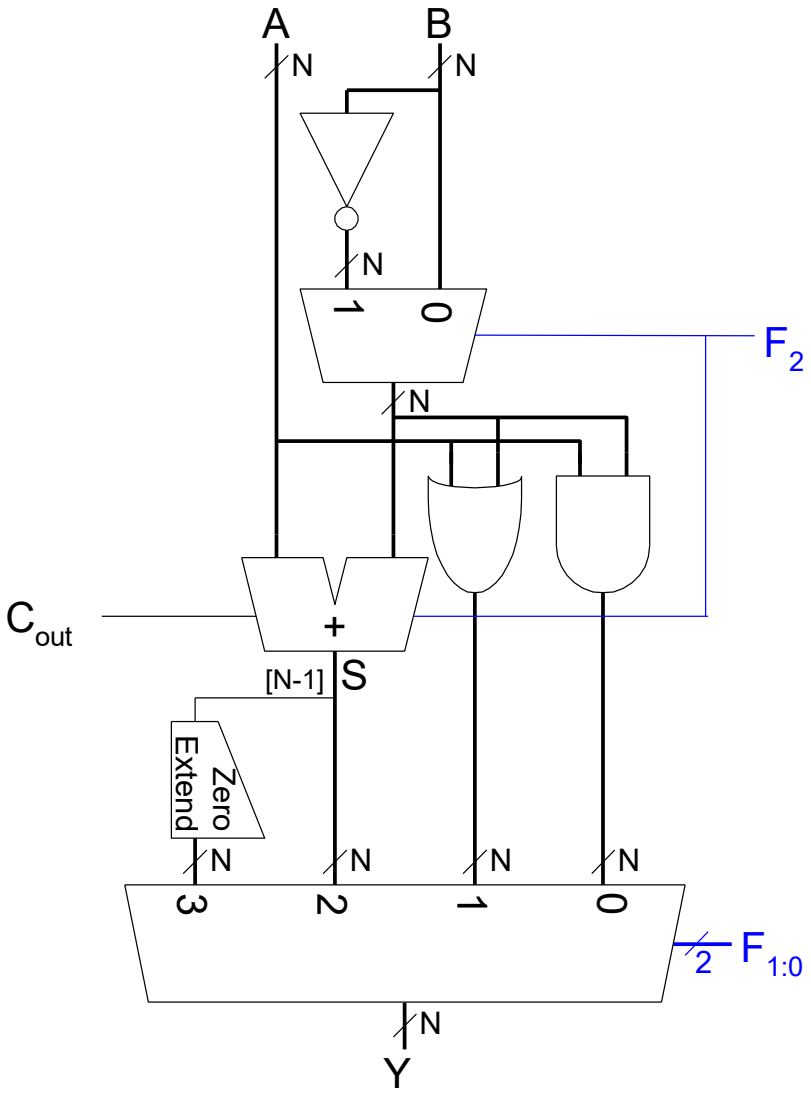


Review: ALU



$F_{2:0}$	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



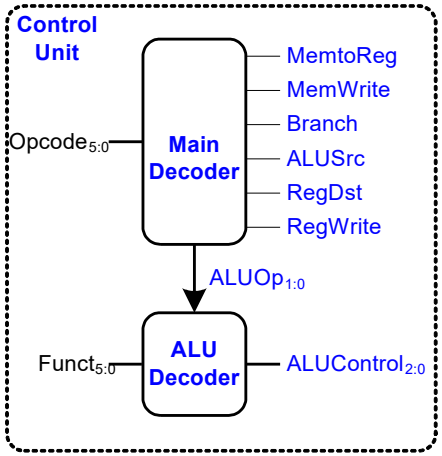
$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT

Control Unit: ALU Decoder

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

R-Type

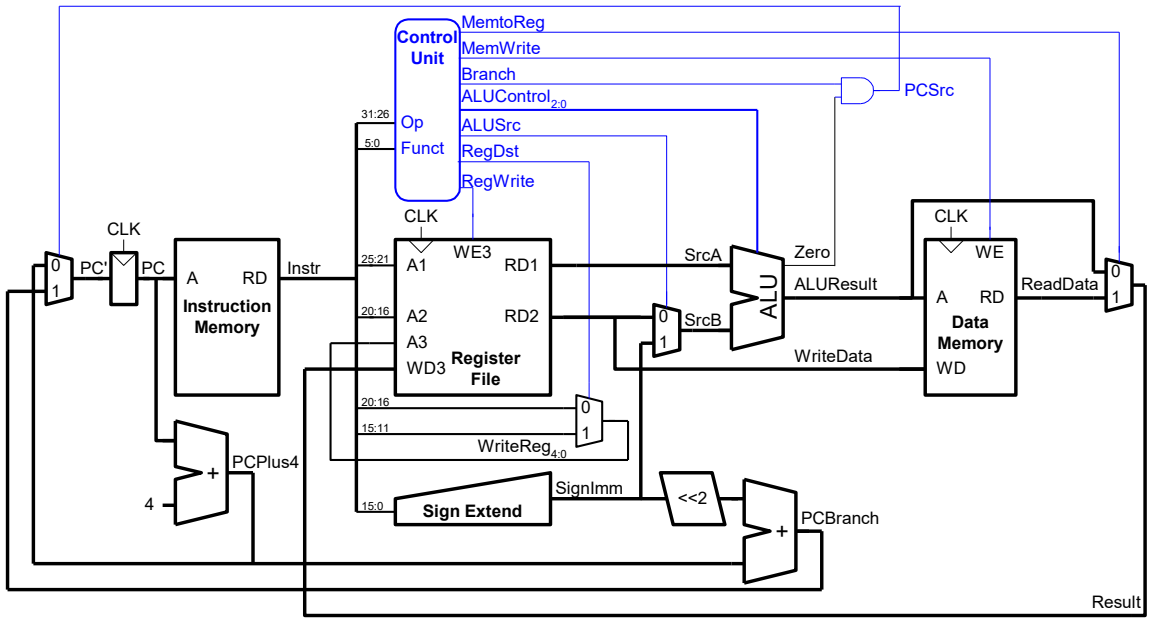
$ALUOp_{1:0}$	Funct	$ALUControl_{2:0}$
00	X	010 (Add)
X1	X	110 (Subtract)
1X	100000 (`add`)	010 (Add)
1X	100010 (`sub`)	110 (Subtract)
1X	100100 (`and`)	000 (And)
1X	100101 (`or`)	001 (Or)
1X	101010 (`slt`)	111 (SLT)



Control Unit Main Decoder

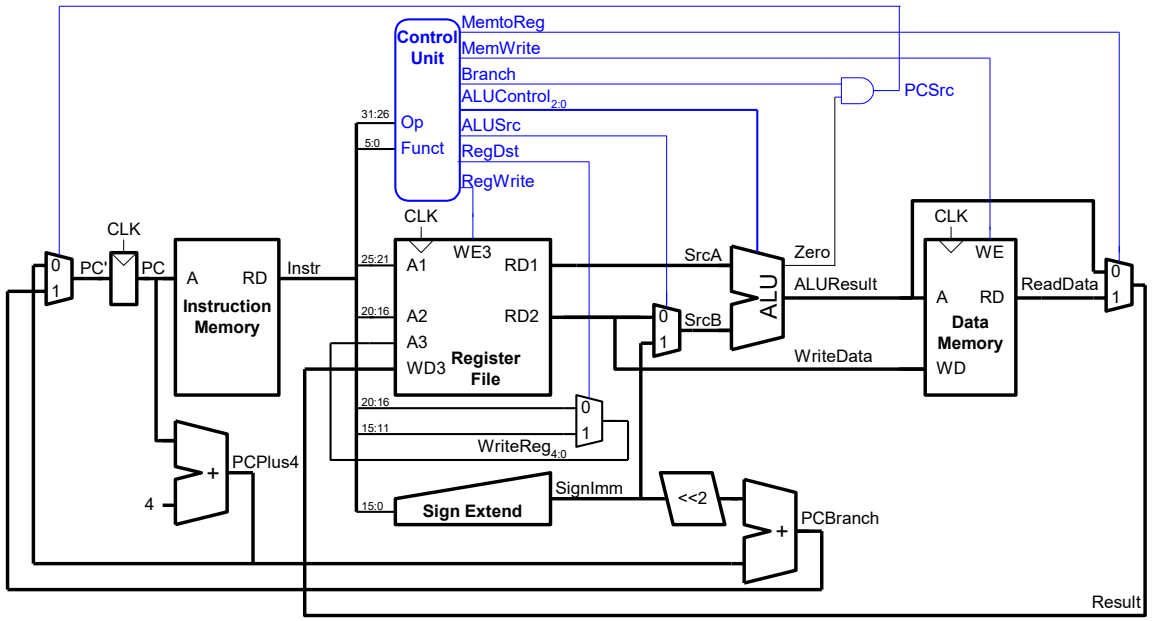
Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$
R-type	000000							
`lw`	100011							
`sw`	101011							
`beq`	000100							

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



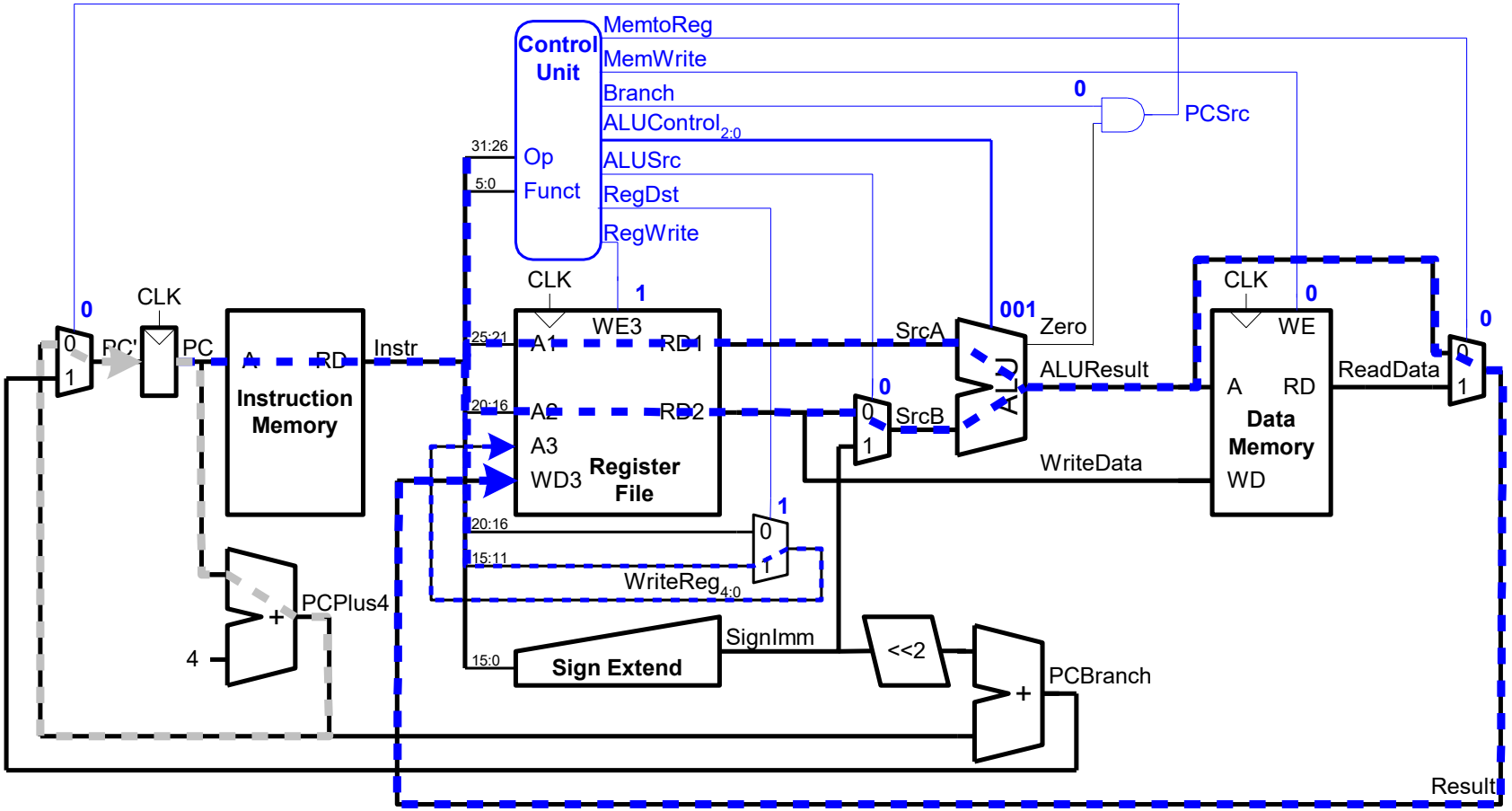
Control Unit: Main Decoder

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	1	00
`sw`	101011	0	X	1	0	1	X	00
`beq`	000100	0	X	0	1	0	X	01



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

Single-Cycle Datapath: or

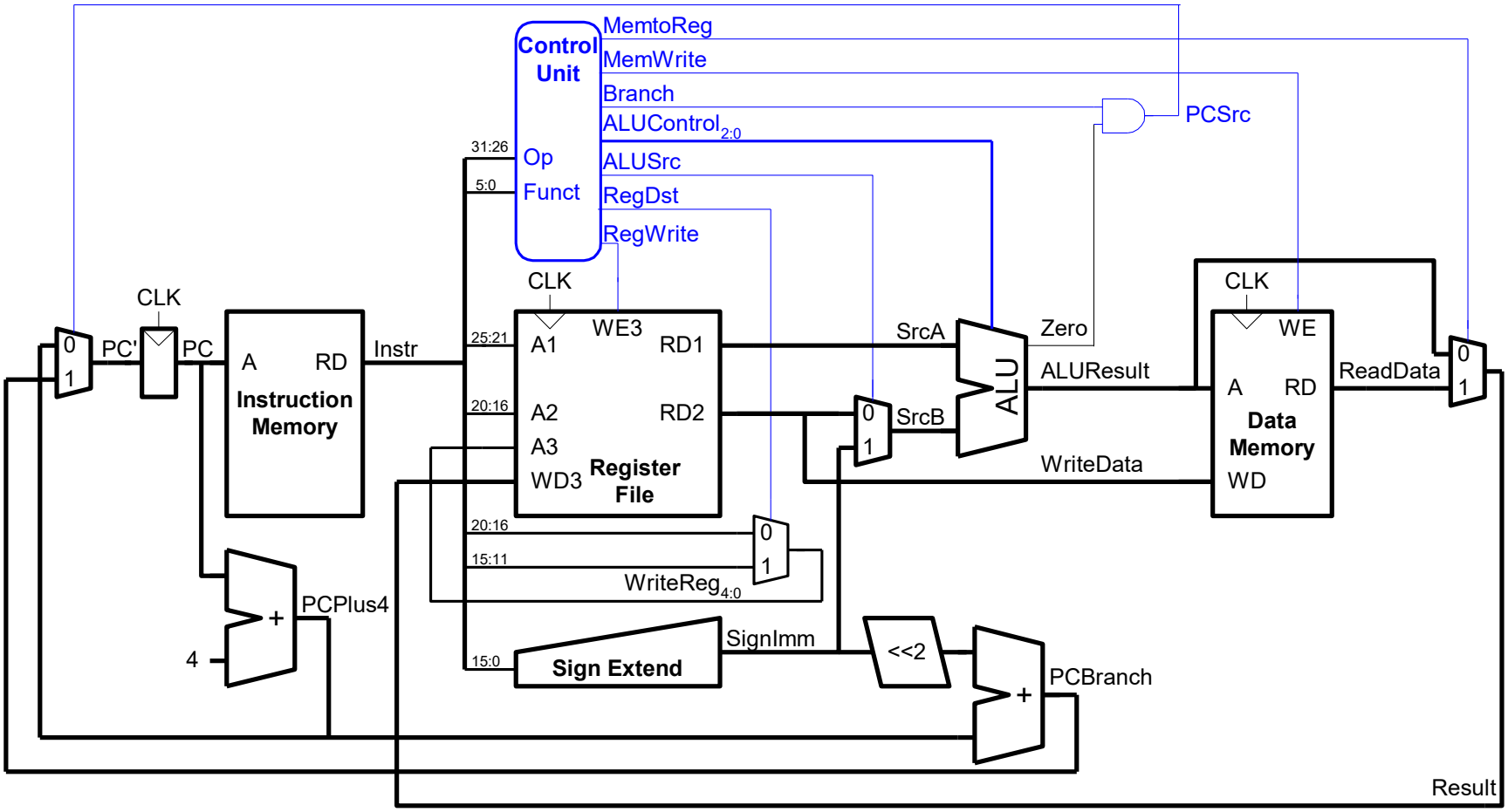


R-Type

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

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

Extended Functionality: addi

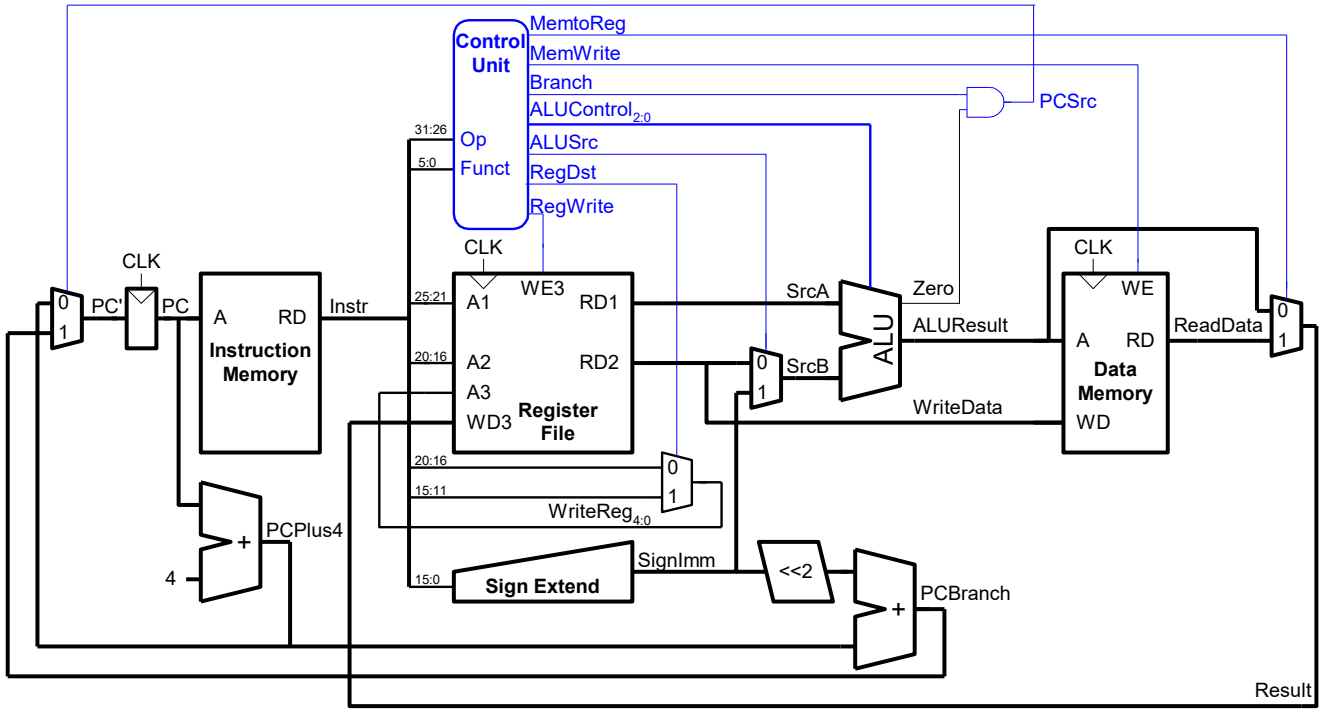


No change to datapath

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
addi	**001000**	**1**	**0**	**1**	**0**	**0**	**0**	**00**

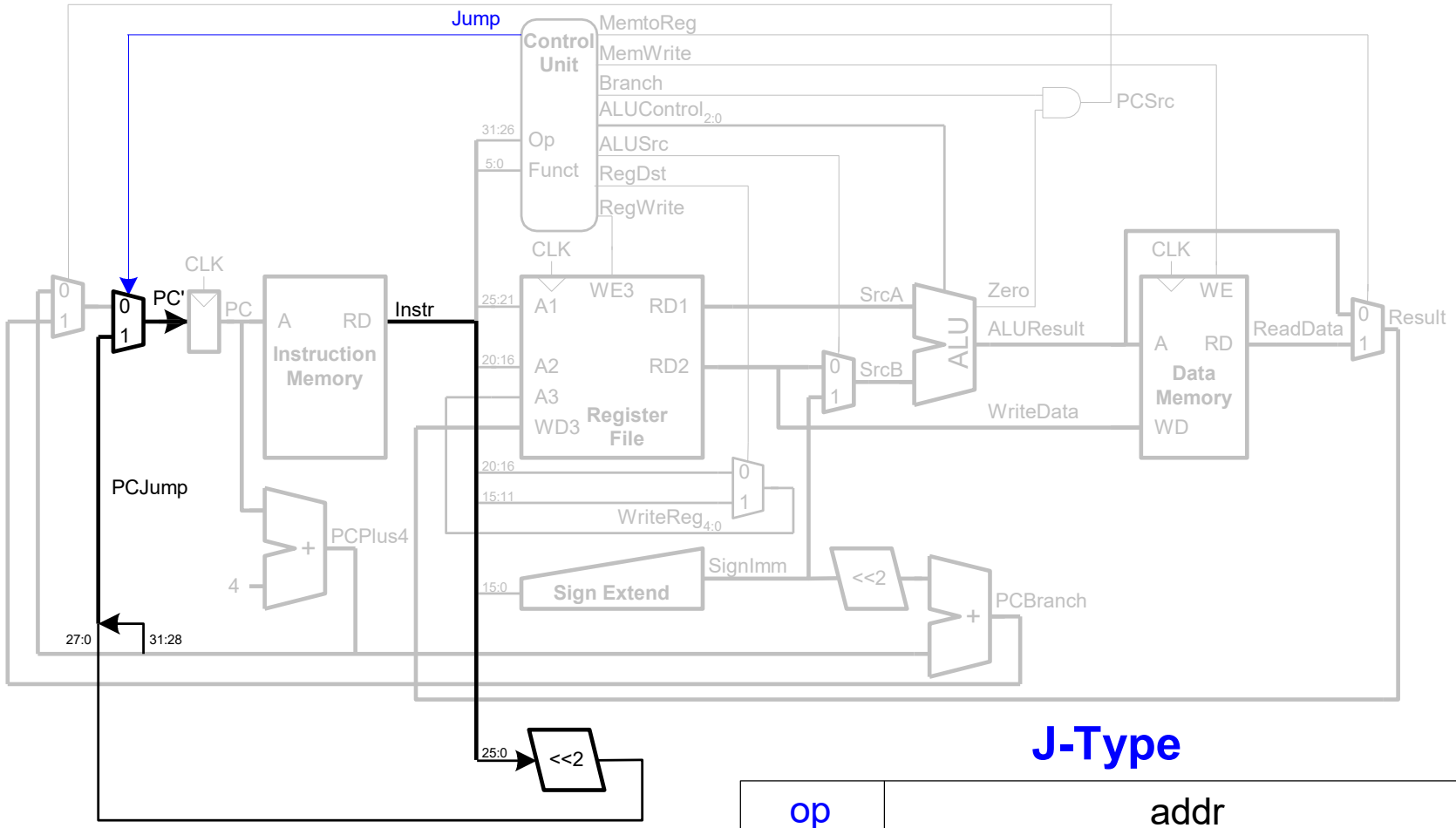
```
addi rt, rs, imm
```



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



J-Type

op	addr
6 bits	26 bits

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`	000010								

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`	**000010**	**0**	**X**	**X**	**X**	**0**	**X**	**XX**	**1**