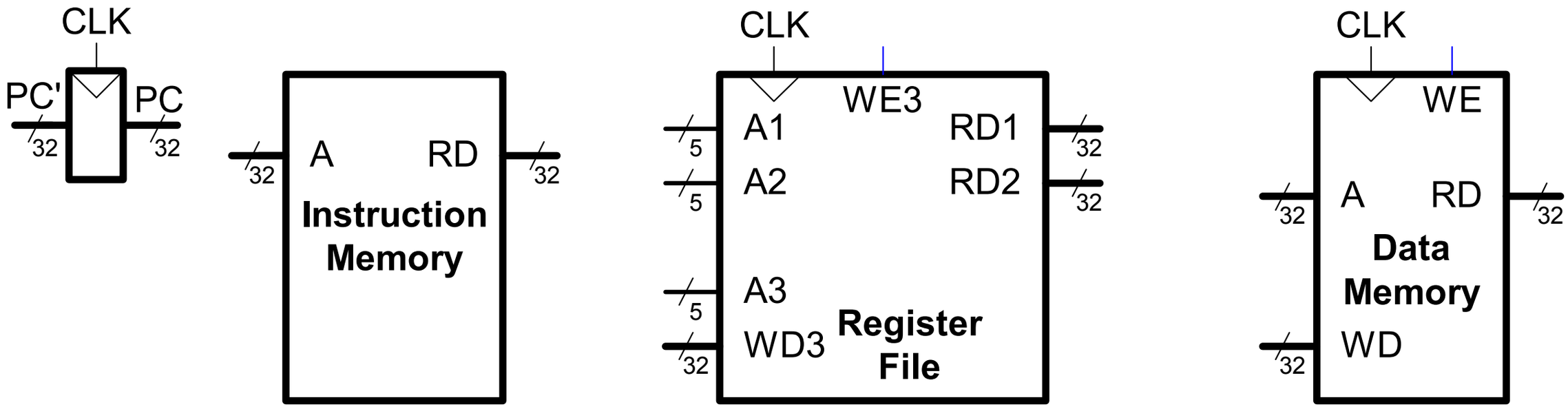
EECE 2322: Fundamentals of Digital Design and Computer Organization

Lecture 13_1: Microarchitecture

Xiaolin Xu
Department of ECE
Northeastern University

MIPS State Elements



- Instruction memory, register file, and data memory, are being **read** as **combinational circuit, i.e., data change follows address change, while the writing follows clock-edge**
- **Benefits: all elements are synchronized, while MIPS processor is a FSM!**

Single-Cycle MIPS Processor

- Datapath
- Control

Overview: Single-Cycle Control

R-Type

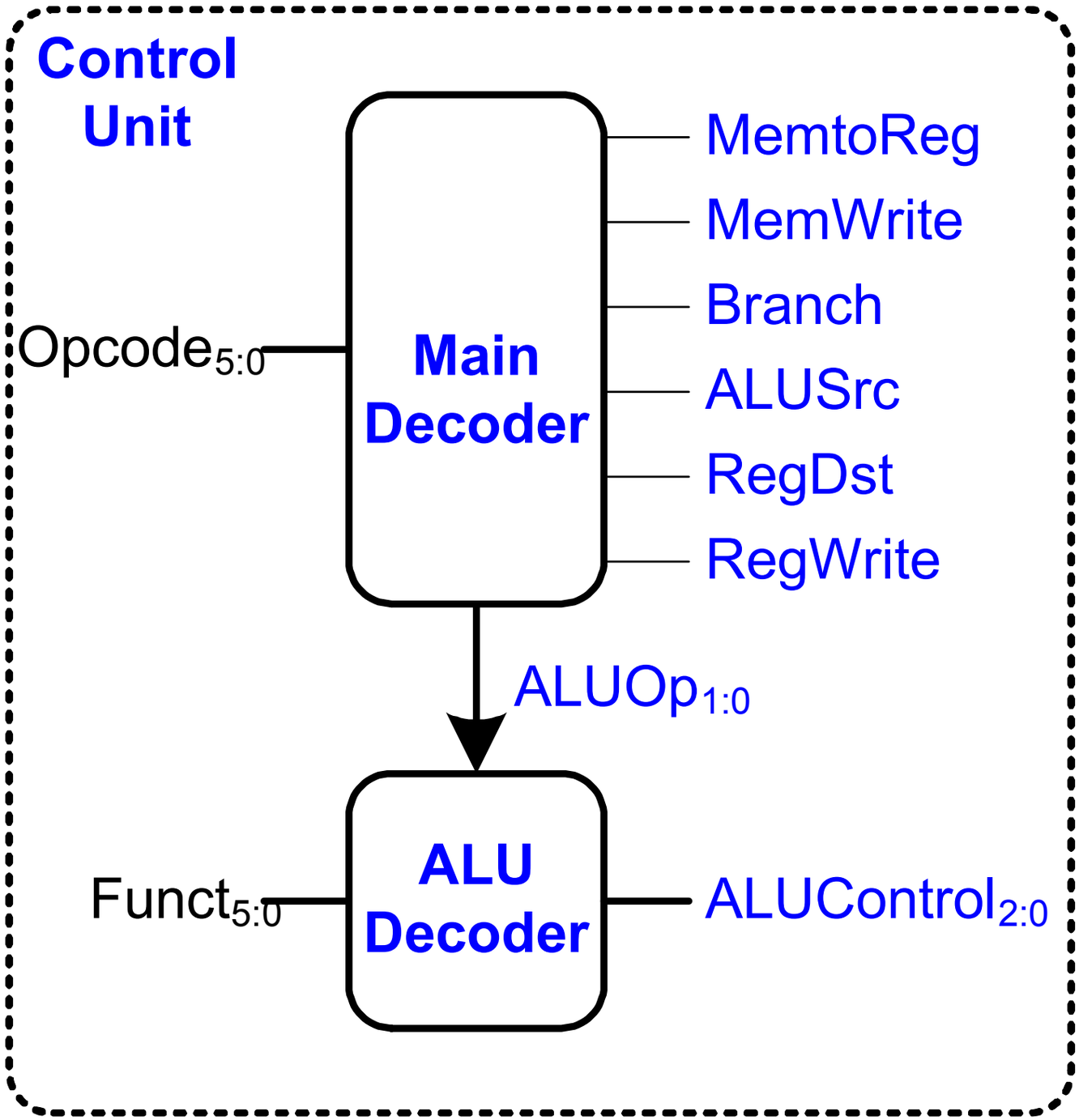
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

I-Type

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

J-Type

op	addr
6 bits	26 bits



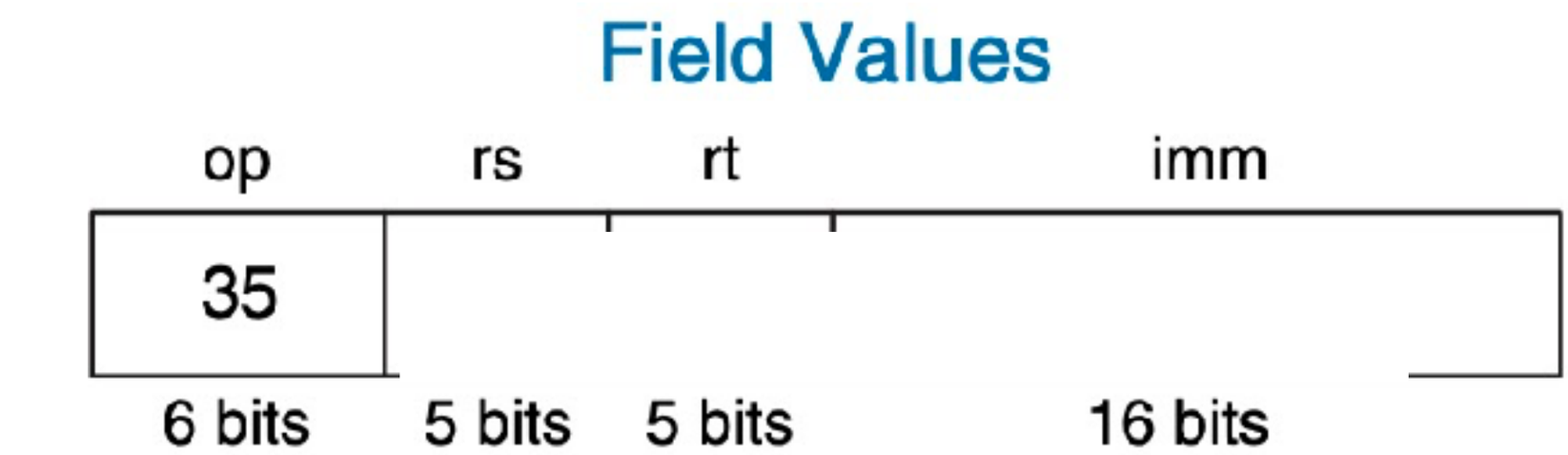
Single-Cycle Datapath Example: lw fetch

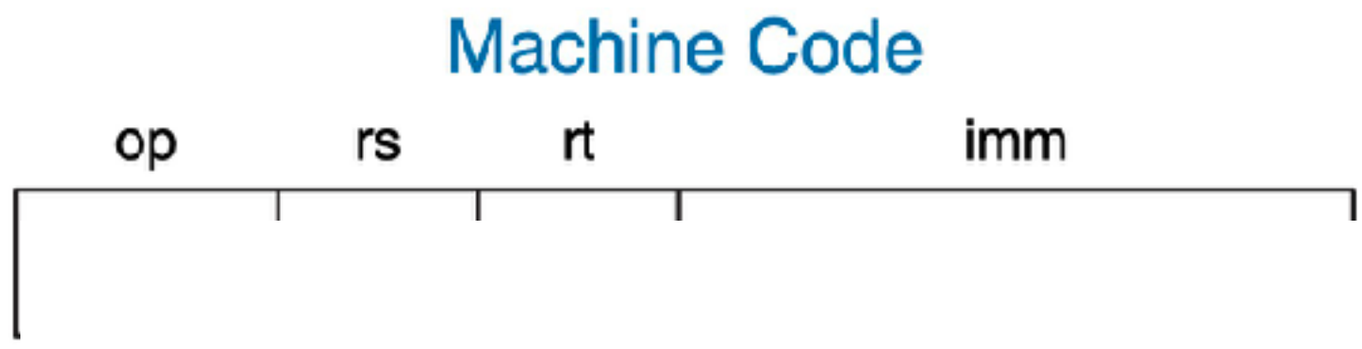
Revisit: Practice of I-type

```
lw $s3, -24($s4)
```

lw has opcode of 35

Name	Number
$0	0
$at	1
$v0–$v1	2–3
$a0–$a3	4–7
$t0–$t7	8–15
$s0–$s7	16–23
$t8–$t9	24–25
$k0–$k1	26–27
$gp	28
$sp	29
$fp	30
$ra	31





Revisit: Practice of I-type

`lw $s3, -24($s4)`

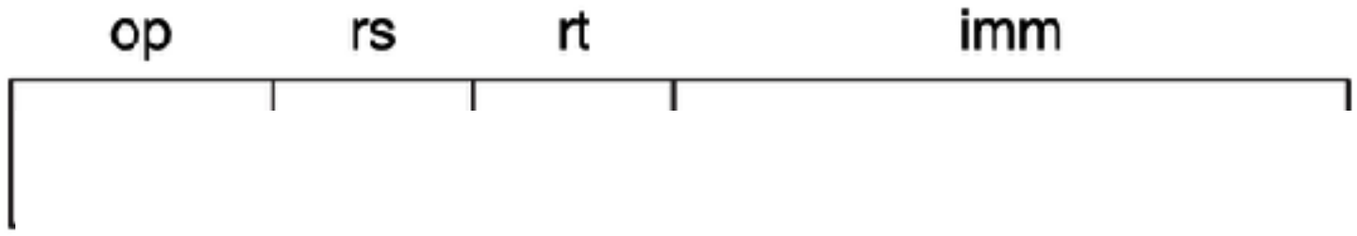
lw has opcode of 35

Name	Number
$0	0
$at	1
$v0–$v1	2–3
$a0–$a3	4–7
$t0–$t7	8–15
$s0–$s7	16–23
$t8–$t9	24–25
$k0–$k1	26–27
$gp	28
$sp	29
$fp	30
$ra	31

Field Values

op	rs	rt	imm
35	20	19	–24
6 bits	5 bits	5 bits	16 bits

Machine Code



Revisit: Practice of I-type

`lw $s3, -24($s4)`

lw has opcode of 35

Name	Number
$0	0
$at	1
$v0–$v1	2–3
$a0–$a3	4–7
$t0–$t7	8–15
$s0–$s7	16–23
$t8–$t9	24–25
$k0–$k1	26–27
$gp	28
$sp	29
$fp	30
$ra	31

Field Values

op	rs	rt	imm
35	20	19	–24
6 bits	5 bits	5 bits	16 bits

Machine Code

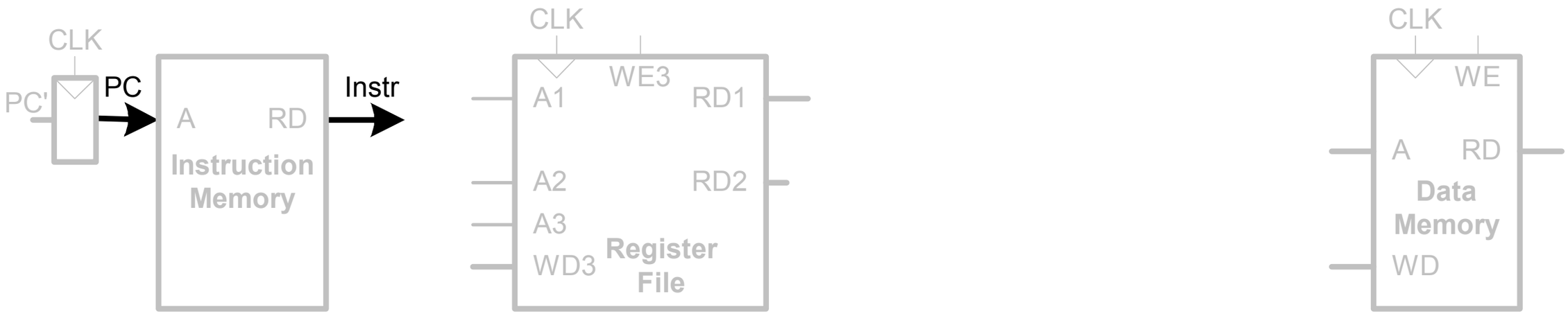
op	rs	rt	imm	
100011	10100	10011	1111 1111 1110 1000	(0x8E93FFE8)
8	E	9	3	F F E 8

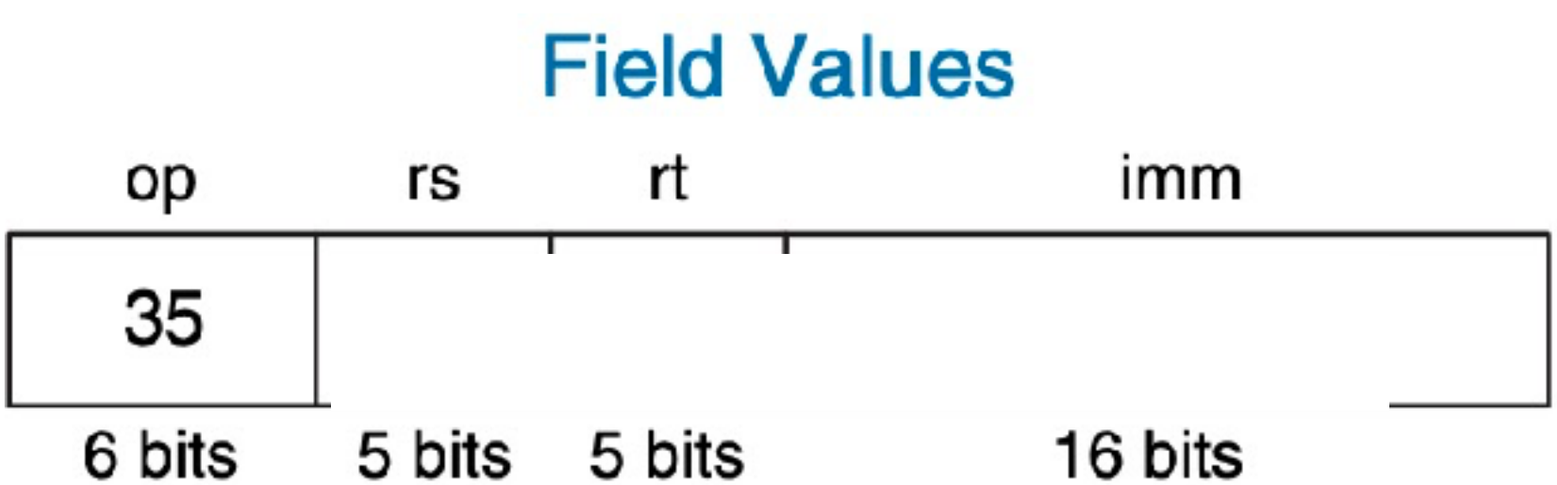
Single-Cycle Datapath Example: lw fetch

STEP 1: Fetch instruction

```
lw $s3, -24($s4)
```

lw has opcode of 35



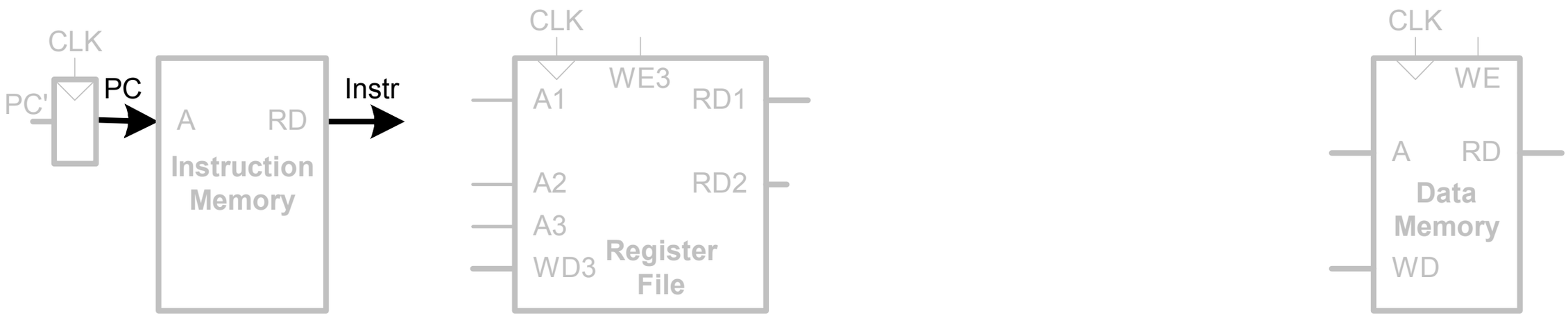


Single-Cycle Datapath Example: lw fetch

STEP 1: Fetch instruction

```
lw $s3, -24($s4)
```

lw has opcode of 35



Field Values

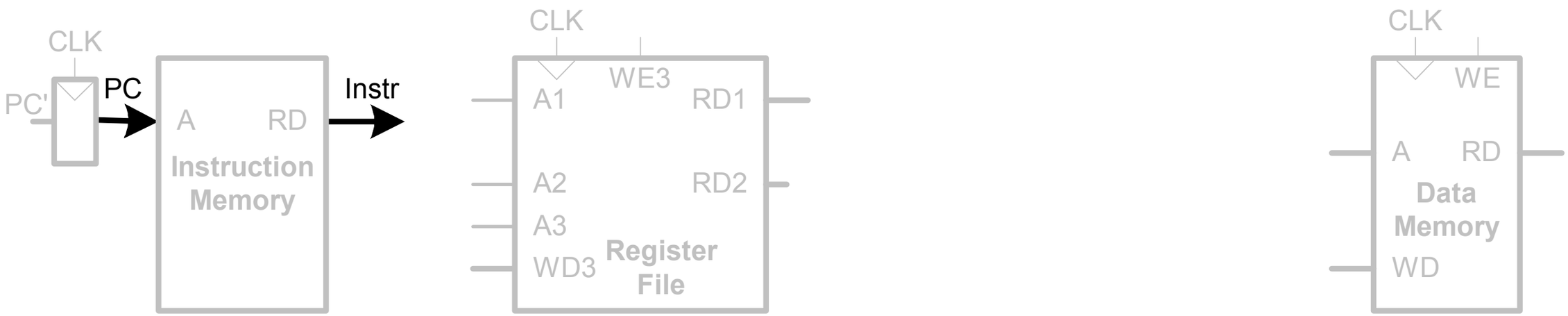
op	rs	rt	imm
35	20	19	–24
6 bits	5 bits	5 bits	16 bits

Single-Cycle Datapath Example: lw fetch

STEP 1: Fetch instruction

```
lw $s3, -24($s4)
```

lw has opcode of 35



The offset is stored in the immediate field of the instruction, $Instr_{15:0}$

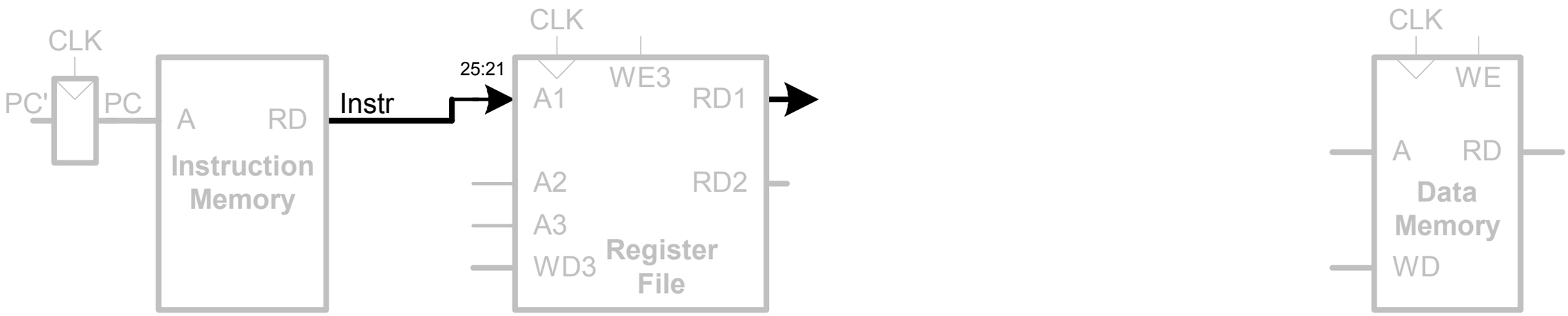
Field Values

op	rs	rt	imm
35	20	19	–24
6 bits	5 bits	5 bits	16 bits

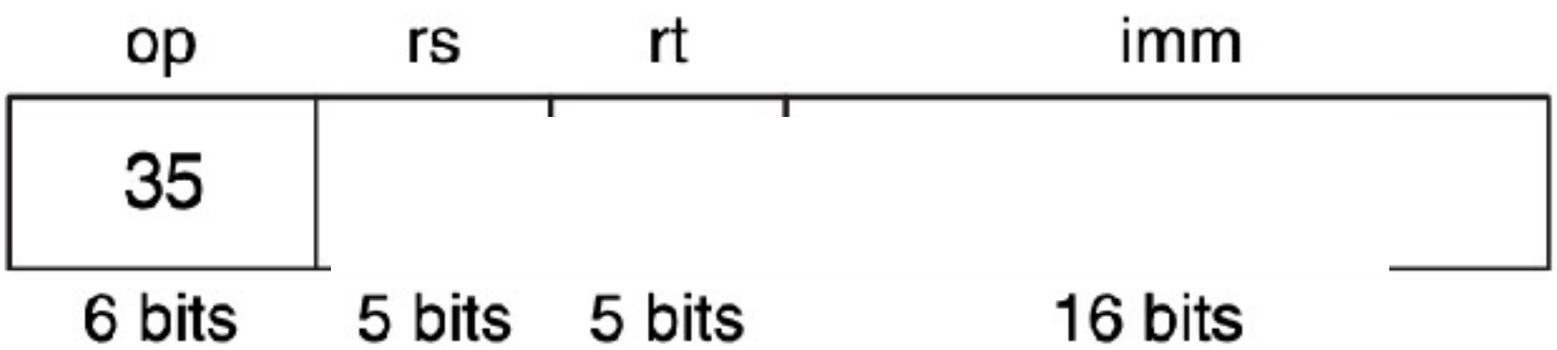
Single-Cycle Datapath: lw Register Read

STEP 2: Read source operands from RF

```
lw $s3, -24($s4)
```



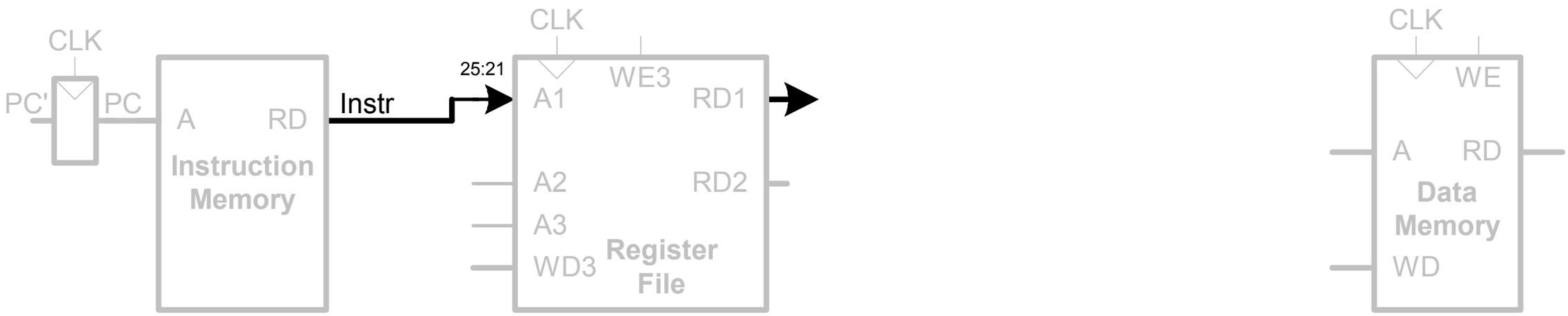
Field Values



Single-Cycle Datapath: lw Register Read

STEP 2: Read source operands from RF

```
lw $s3, -24($s4)
```



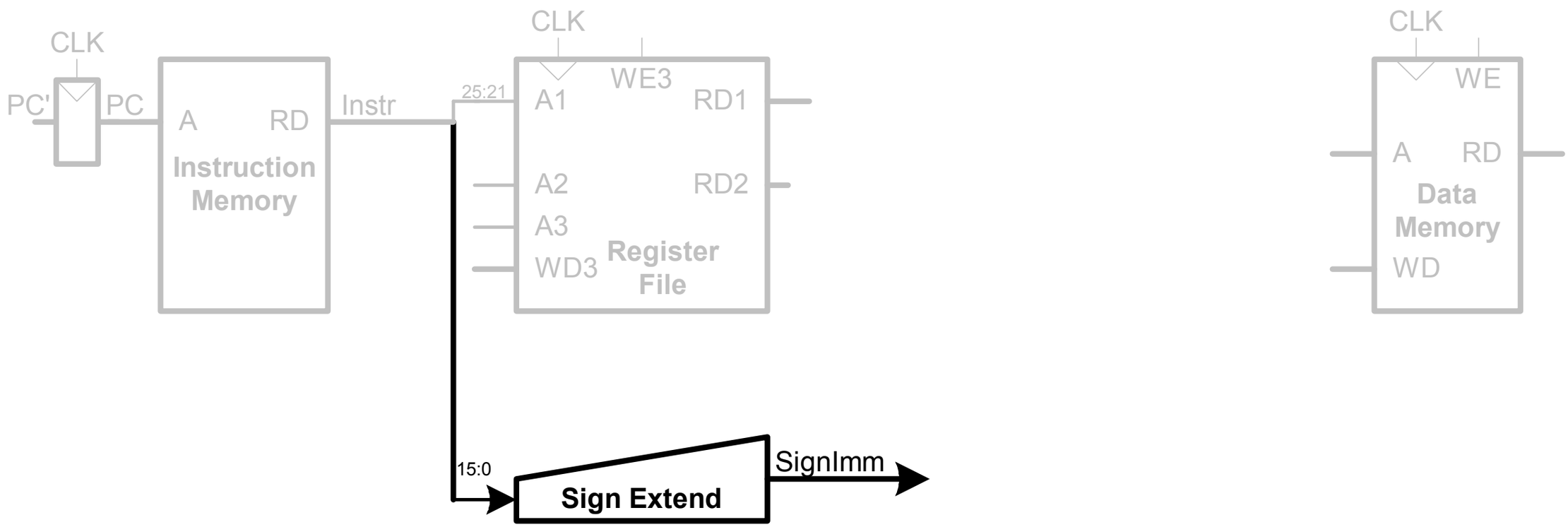
Field Values

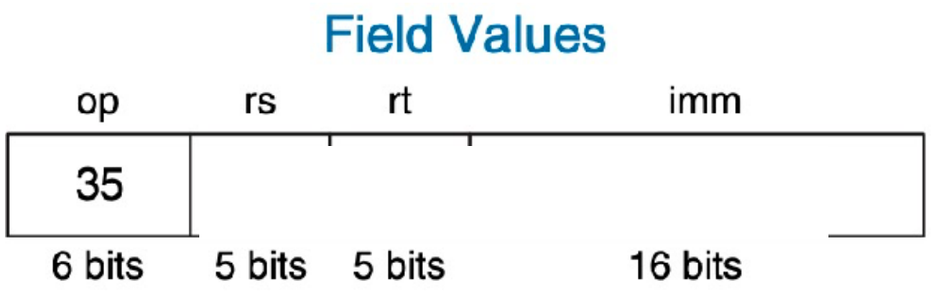
op	rs	rt	imm
35	20	19	–24
6 bits	5 bits	5 bits	16 bits

Single-Cycle Datapath: lw Immediate

STEP 3: Sign-extend the immediate

```
lw $s3, -24($s4)
```



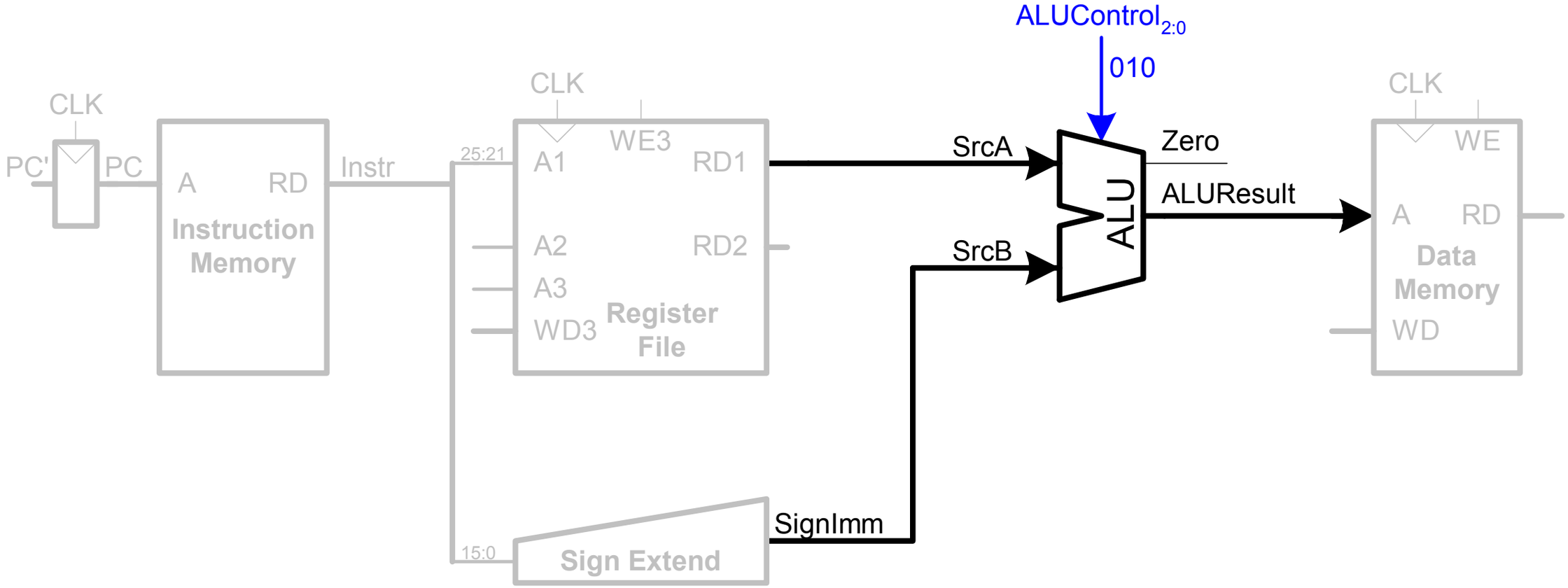


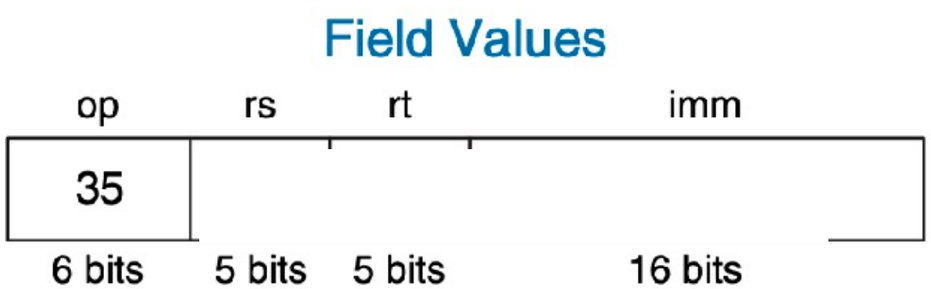
$$SignImm_{15:0} = Instr_{15:0}$$
$$SignImm_{31:16} = Instr_{15}$$

Single-Cycle Datapath: lw address

STEP 4: Compute the memory address

```
lw $s3, -24($s4)
```



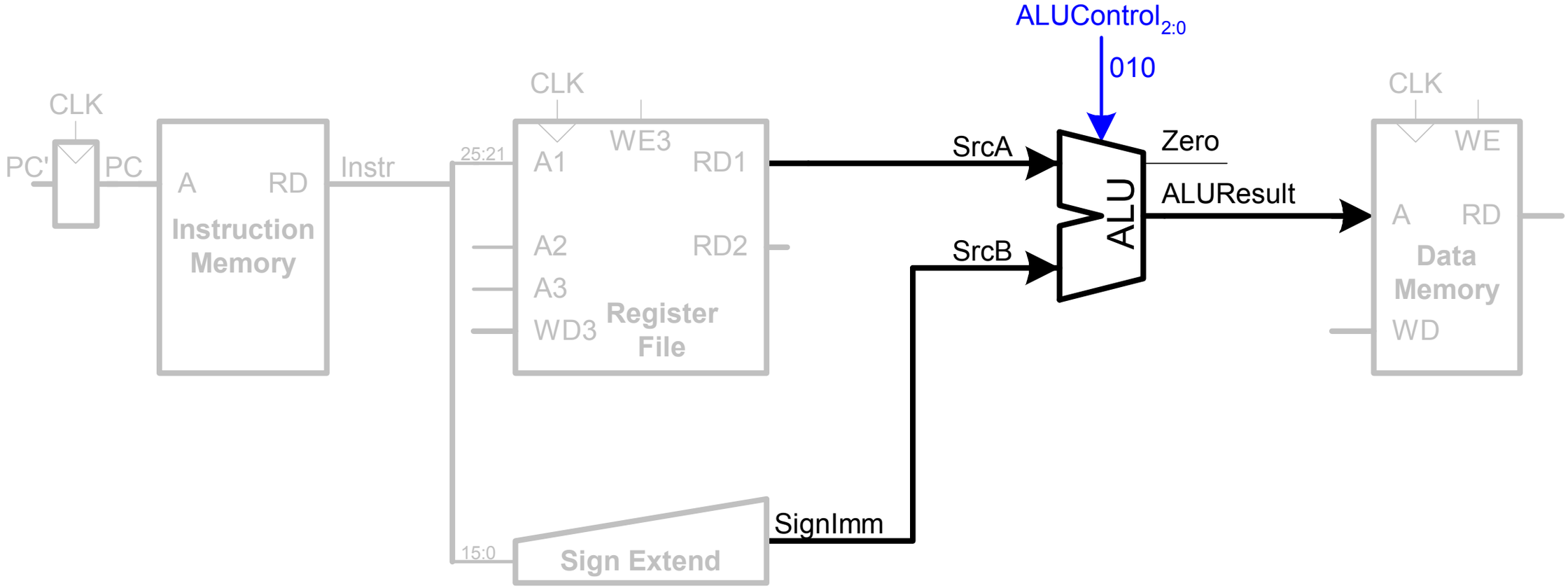


Single-Cycle Datapath: lw address

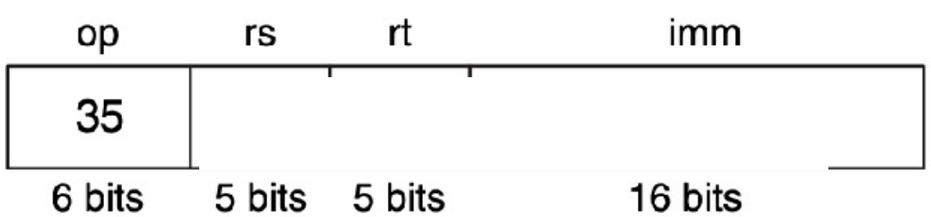
add the base address to the offset to find the address to read from memory

STEP 4: Compute the memory address

```
lw $s3, -24($s4)
```



Field Values

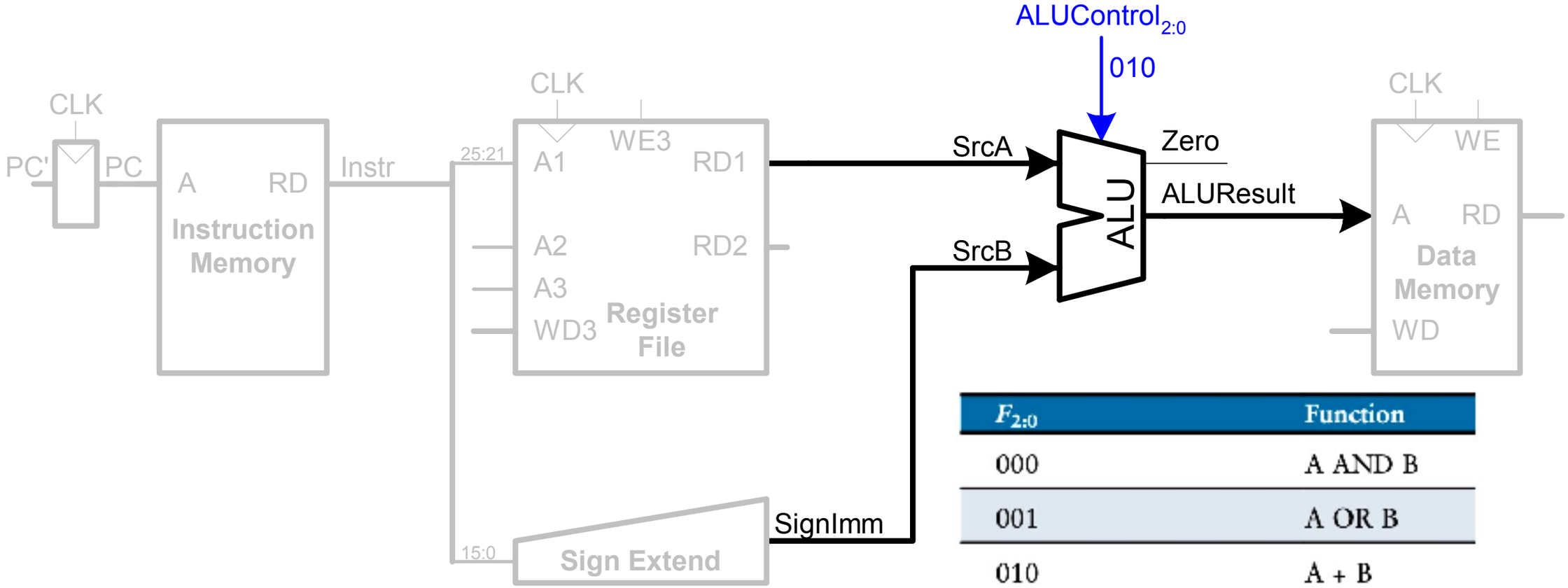


Single-Cycle Datapath: lw address

add the base address to the offset to find the address to read from memory

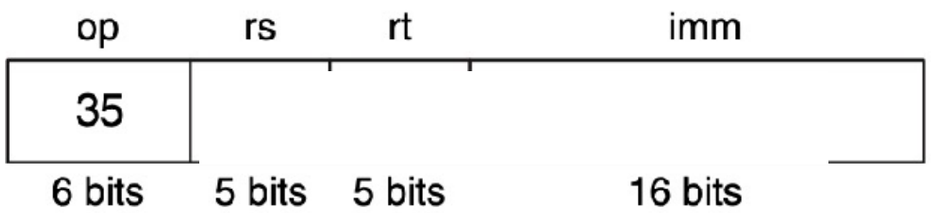
STEP 4: Compute the memory address

```
lw $s3, -24($s4)
```



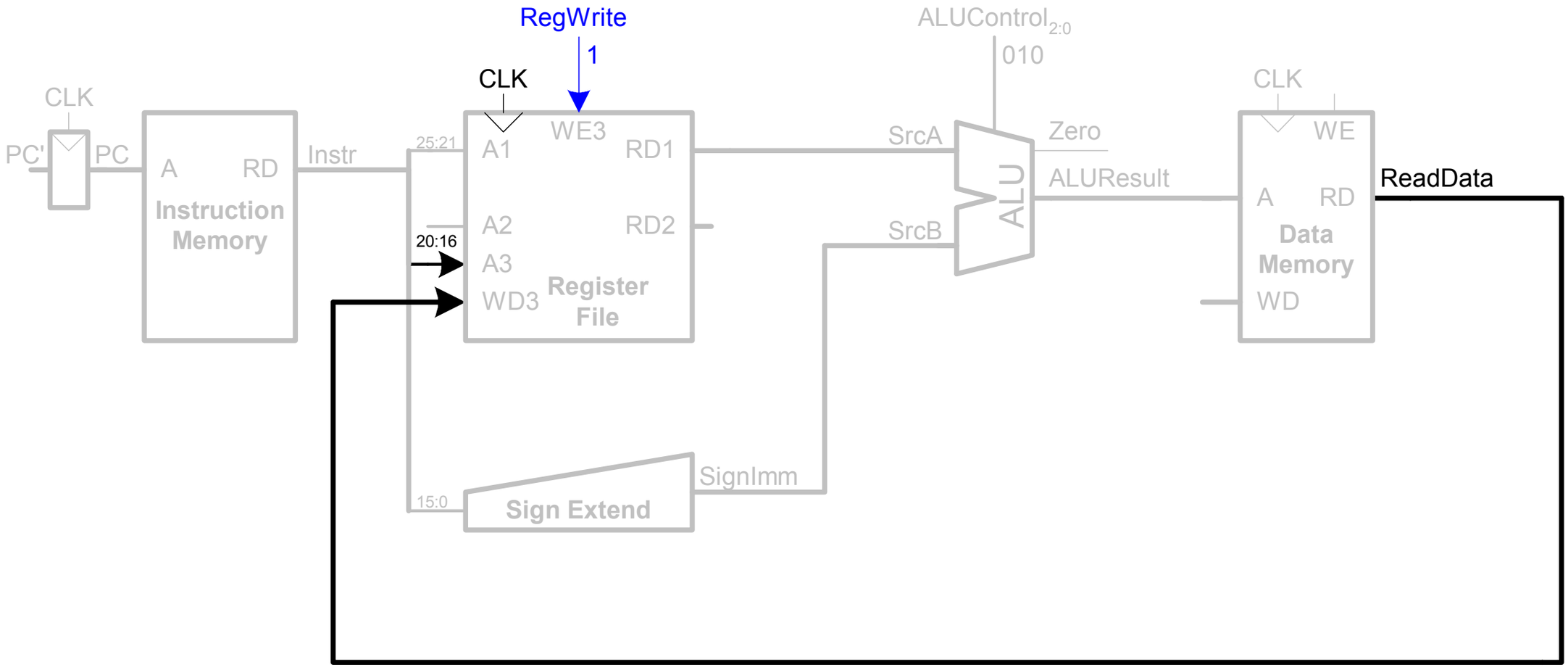
$F_{2:0}$	Function
000	A AND B
001	A OR B
010	A + B
011	not used
100	A AND $\overline{B}$
101	A OR $\overline{B}$
110	A – B
111	SLT

Field Values



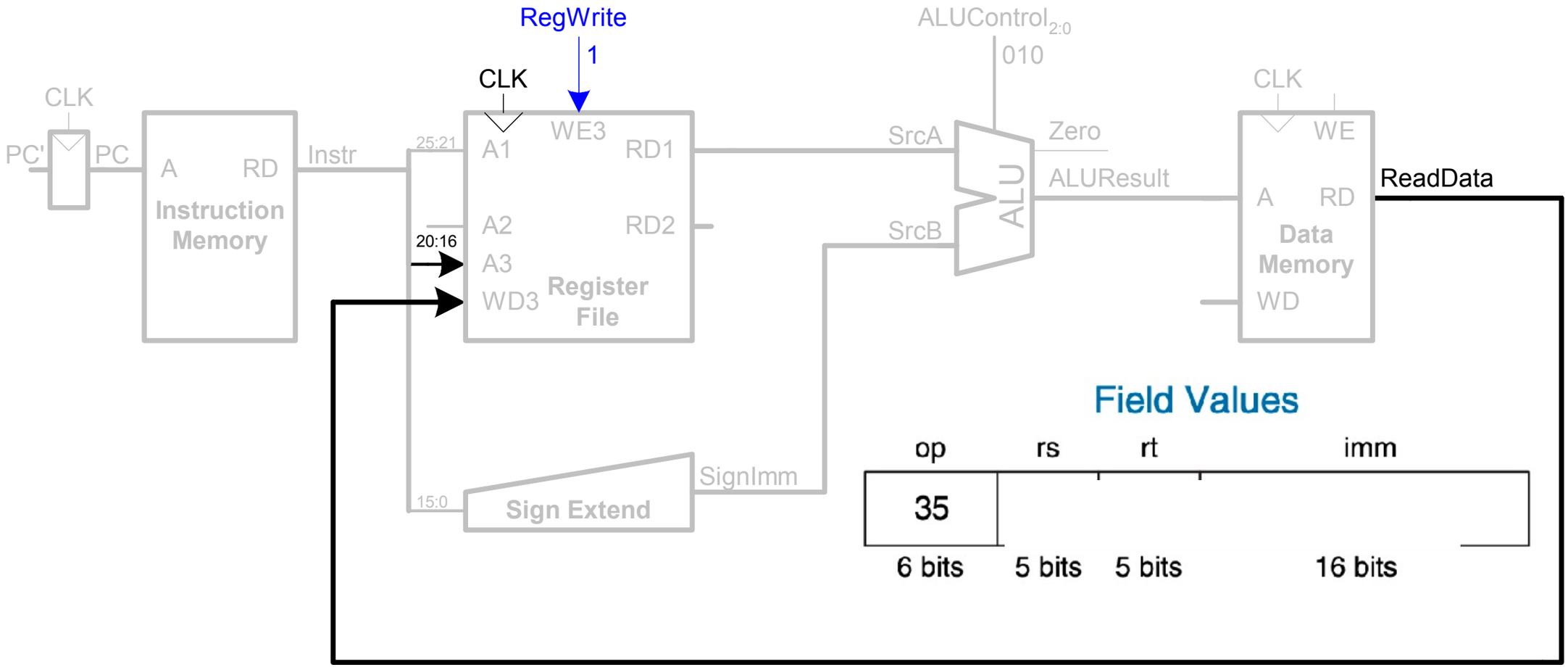
Single-Cycle Datapath: lw Memory Read

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



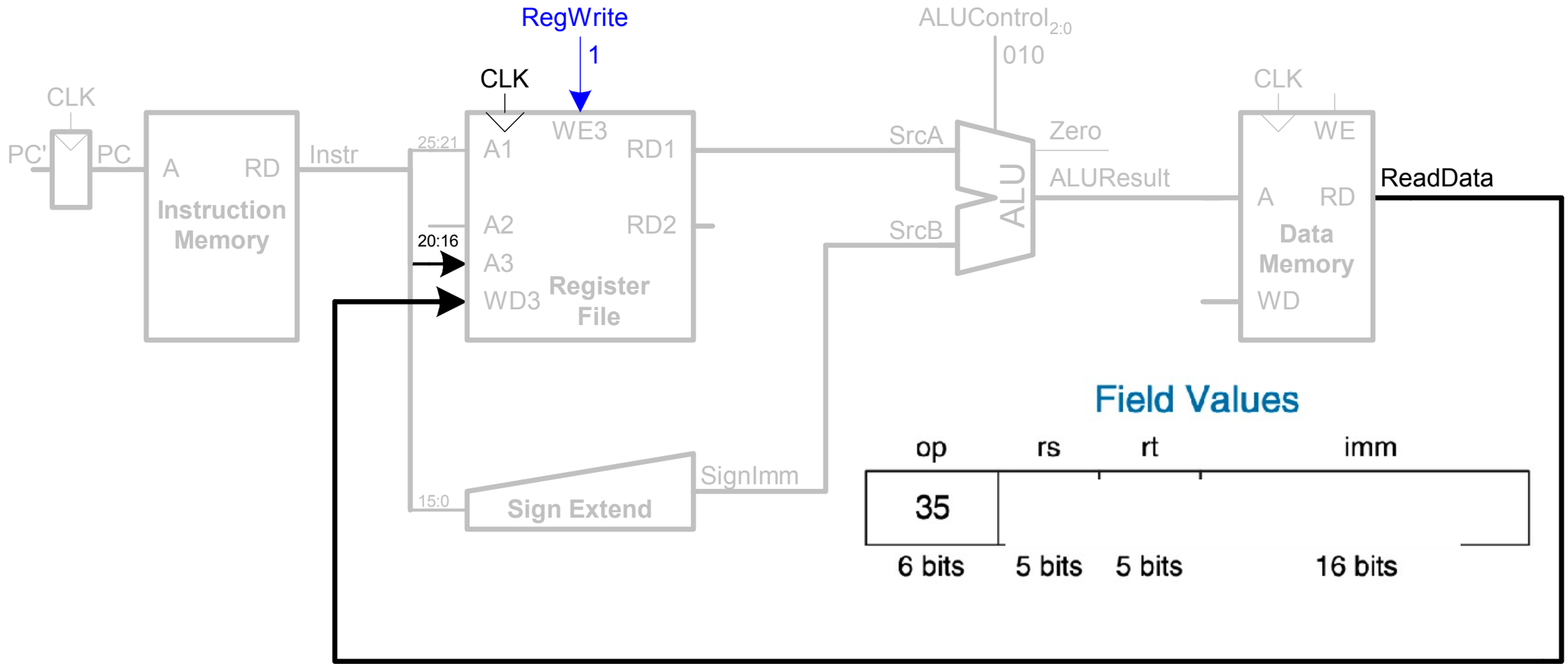
Single-Cycle Datapath: lw Memory Read

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



Single-Cycle Datapath: lw Memory Read

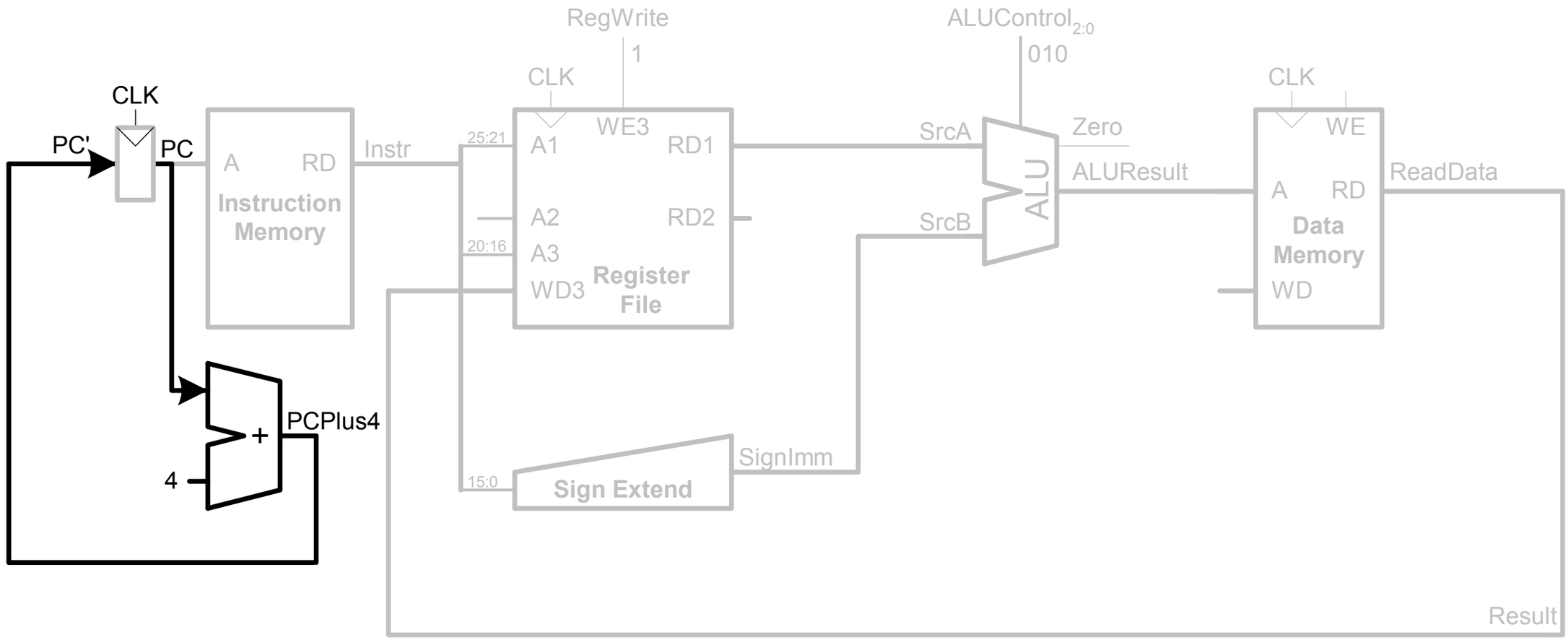
- **STEP 5:** Read data from memory and write it back to register file



The data is read from the data memory onto the ReadData bus, then written back to the destination register in the register file at the end of the cycle

Single-Cycle Datapath: lw PC Increment

STEP 6: Determine address of next instruction



Use another adder to increment the PC by 4

The new address is written into the program counter on the next rising edge of the clock. This completes the datapath for the lw instruction

Single-Cycle Datapath: sw

Single-Cycle Datapath: sw

Like the **lw** instruction, the **sw** instruction reads a base address from port 1 of the register file and sign-extends an immediate.

Single-Cycle Datapath: sw

Like the **lw** instruction, the **sw** instruction reads a base address from port 1 of the register file and sign-extends an immediate.

ALU adds the base address to the immediate to find the memory address

Name	Number
$0	0
$at	1
$v0–$v1	2–3
$a0–$a3	4–7
$t0–$t7	8–15
$s0–$s7	16–23
$t8–$t9	24–25
$k0–$k1	26–27
$gp	28
$sp	29
$fp	30
$ra	31

I-Type Examples

Assembly Code

Field Values

op	rs	rt	imm
43	9	17	4
6 bits	5 bits	5 bits	16 bits

```
sw    $s1,  4($t1)
```

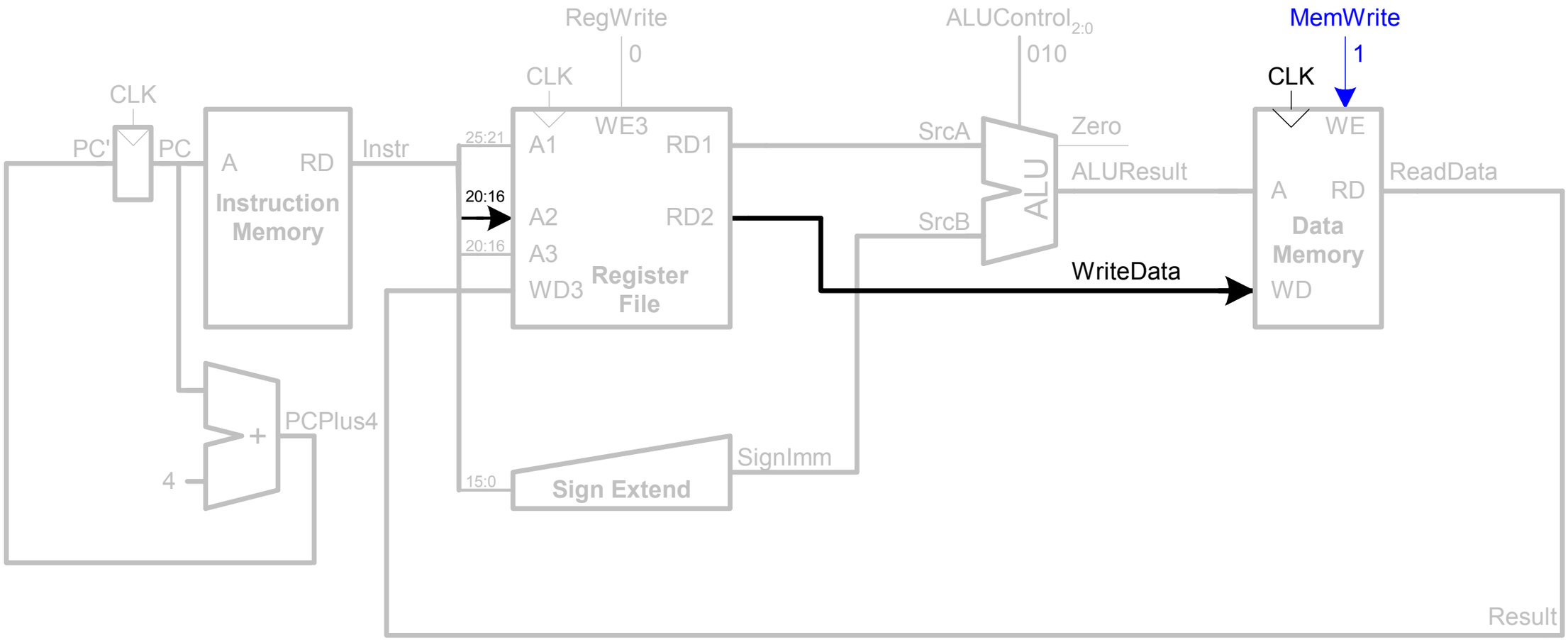
Machine Code

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

101011	01001	10001	0000 0000 0000 0100	(0xAD310004)
6 bits	5 bits	5 bits	16 bits	

Single-Cycle Datapath: sw

Write data in `rt` to memory



Single-Cycle Datapath: R-Type

- Read from `rs` and `rt`

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

Single-Cycle Datapath: R-Type

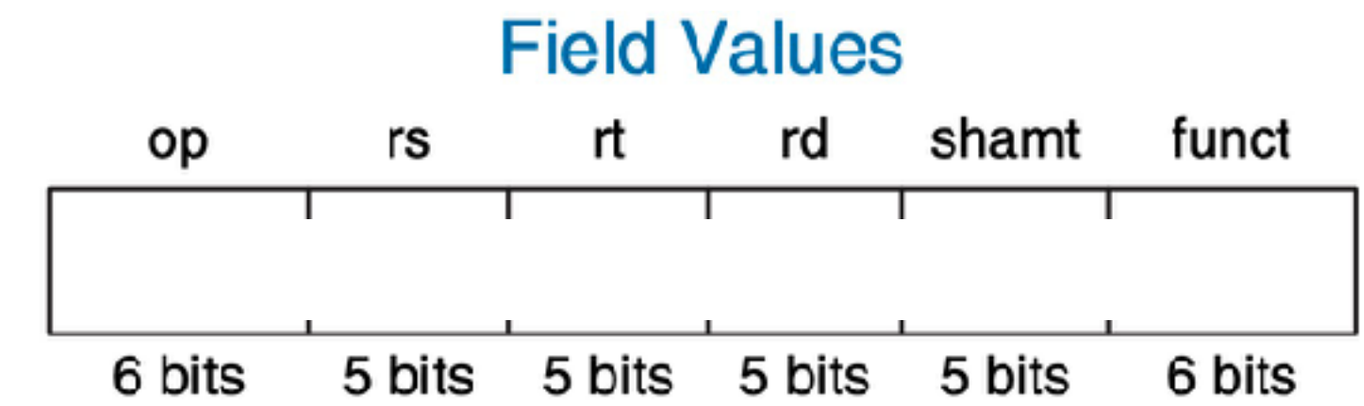
- Write *ALUResult* to register file
- Write to `rd` (instead of `rt`)

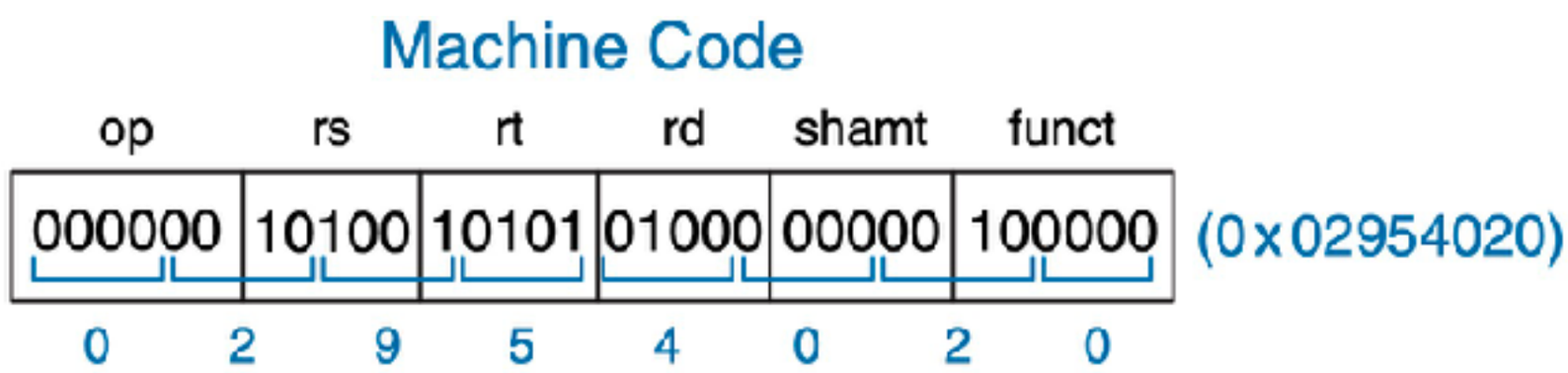
Practice of R-type

```
add $t4, $s4, $s5
```

add has op code of 0 and funct code of 32

Name	Number
$0	0
$at	1
$v0–$v1	2–3
$a0–$a3	4–7
$t0–$t7	8–15
$s0–$s7	16–23
$t8–$t9	24–25
$k0–$k1	26–27
$gp	28
$sp	29
$fp	30
$ra	31





Practice of R-type

```
add $t4, $s4, $s5
```

add has op code of 0 and funct code of 32

Name	Number
$0	0
$at	1
$v0–$v1	2–3
$a0–$a3	4–7
$t0–$t7	8–15
$s0–$s7	16–23
$t8–$t9	24–25
$k0–$k1	26–27
$gp	28
$sp	29
$fp	30
$ra	31

Field Values

op	rs	rt	rd	shamt	funct
0	20	21	8	0	32
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

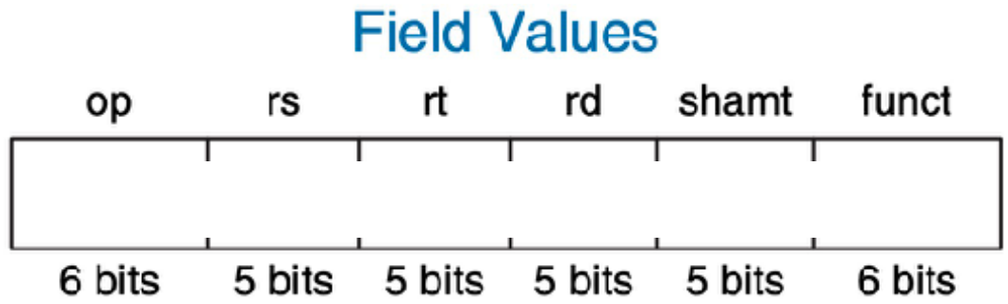
Machine Code

op	rs	rt	rd	shamt	funct
000000	10100	10101	01000	00000	100000
0	2	9	5	4	0 2 0

(0x02954020)

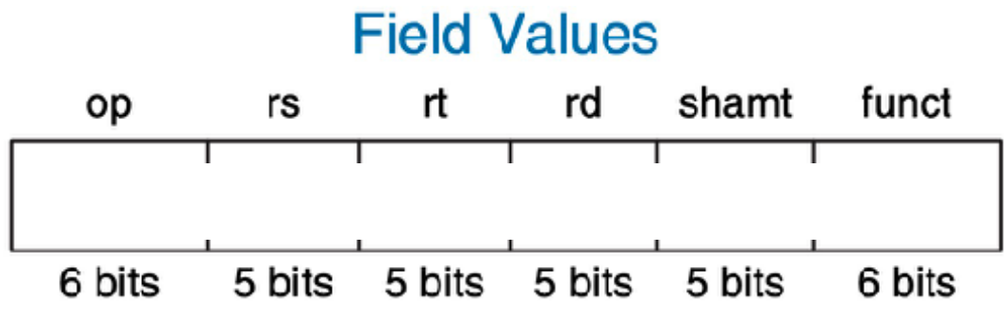
Single-Cycle Datapath: R-Type

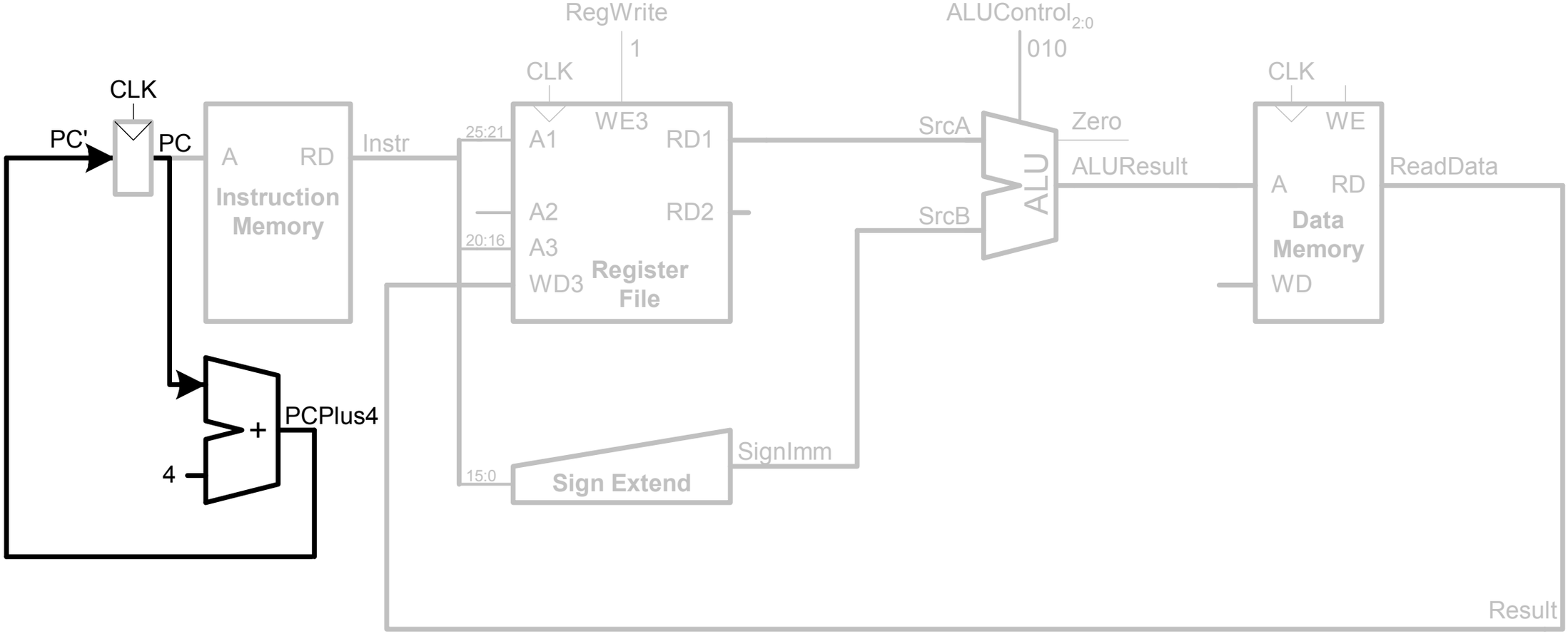
- Read from `rs` and `rt`
- Write *ALUResult* to register file
- Write to `rd` (instead of `rt`)



Single-Cycle Datapath: R-Type

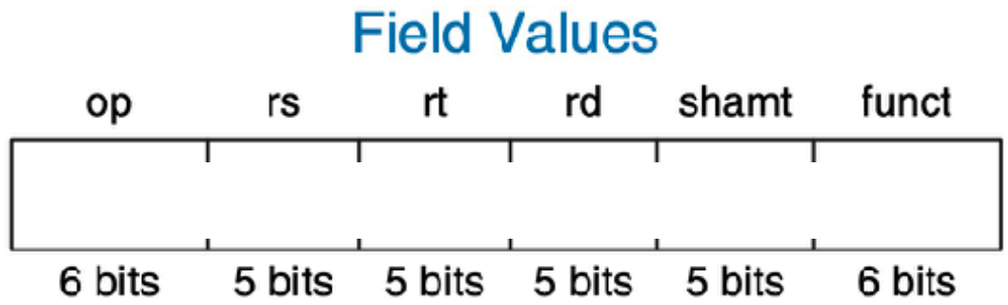
- Read from `rs` and `rt`
- Write *ALUResult* to register file
- Write to `rd` (instead of `rt`)

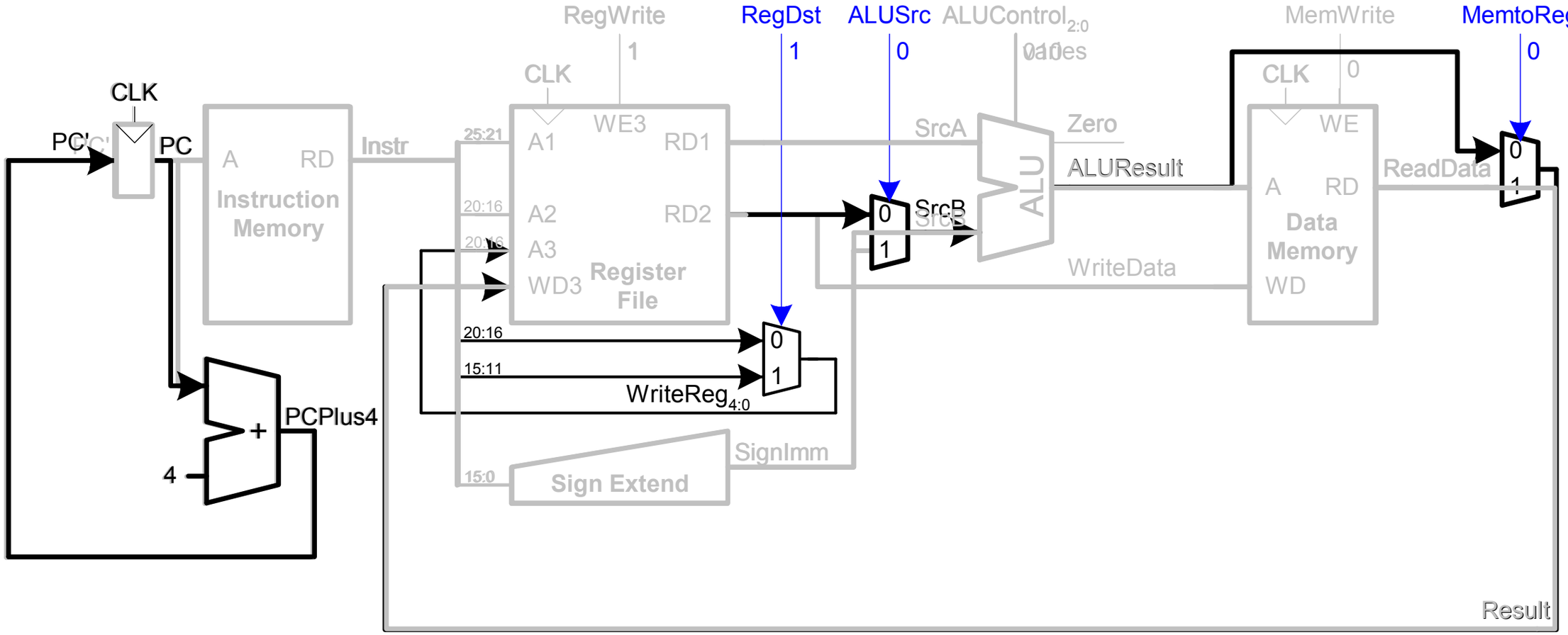




Single-Cycle Datapath: R-Type

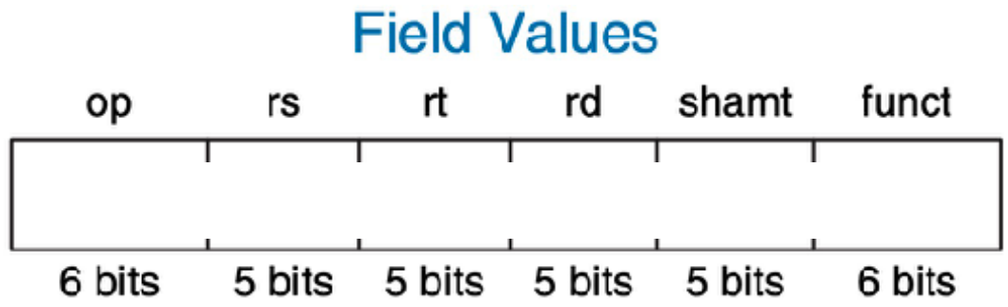
- Read from `rs` and `rt`
- Write *ALUResult* to register file
- Write to `rd` (instead of `rt`)

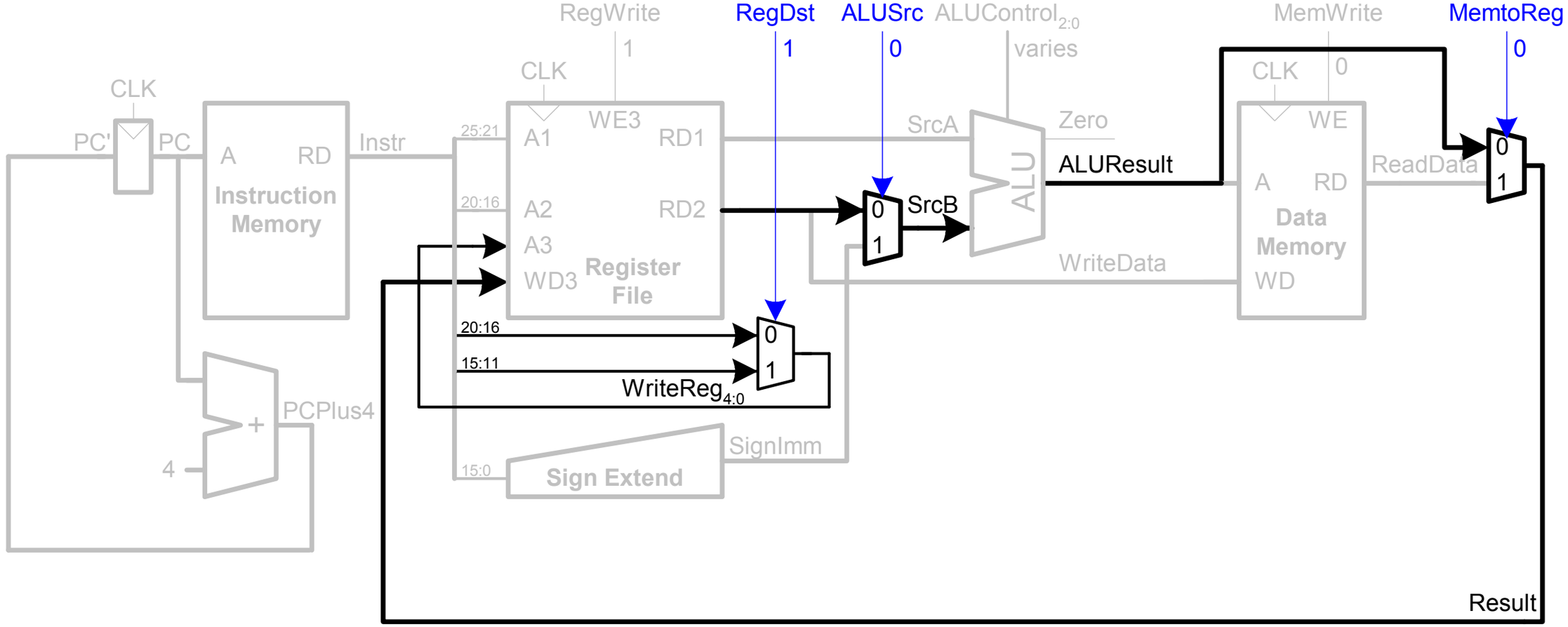




Single-Cycle Datapath: R-Type

- 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

Instruction	Example	Meaning	Comments
branch on equal	`beq $1,$2,100`	if($1==$2) go to PC+4+100	Test if registers are equal
branch on not equal	`bne $1,$2,100`	if($1!=$2) go to PC+4+100	Test if registers are not equal

Conditional Branching (beq)

 # MIPS assembly **Results per line?**

```
addi $s0, $0, 4
addi $s1, $0, 1
sll  $s1, $s1, 2
beq  $s0, $s1, target
addi $s1, $s1, 1
sub  $s1, $s1, $s0
```

Is this "targeted" instruction executed or not? and its results?

```
target:        # label
add  $s1, $s1, $s0
```

Labels indicate instruction location. They can't be reserved words and must be followed by colon (:)

Conditional Branching (beq)

```
# MIPS assembly
 addi $s0, $0, 4
 addi $s1, $0, 1
 sll  $s1, $s1, 2
 beq  $s0, $s1, target
 addi $s1, $s1, 1
 sub  $s1, $s1, $s0
```

```
# Results per line?
# $s0 = 0 + 4 = 4
# $s1 = 0 + 1 = 1
# $s1 = 1 << 2 = 4
# branch is taken
# not executed
# not executed
```

Is this "targeted" instruction executed or not? and its results?

```
target:        # label
 add  $s1, $s1, $s0
```

Labels indicate instruction location. They can't be reserved words and must be followed by colon (:)

Conditional Branching (beq)

```
# MIPS assembly                  Results per line?
 addi $s0, $0, 4                # $s0 = 0 + 4 = 4
 addi $s1, $0, 1                # $s1 = 0 + 1 = 1
 sll  $s1, $s1, 2               # $s1 = 1 << 2 = 4
 beq  $s0, $s1, target          # branch is taken
 addi $s1, $s1, 1               # not executed
 sub  $s1, $s1, $s0             # not executed
```

Is this "targeted" instruction executed or not? and its results?

```
target:          # label
 add  $s1, $s1, $s0      # $s1 = 4 + 4 = 8
```

Labels indicate instruction location. They can't be reserved words and must be followed by colon (:)

Single-Cycle Datapath: beq

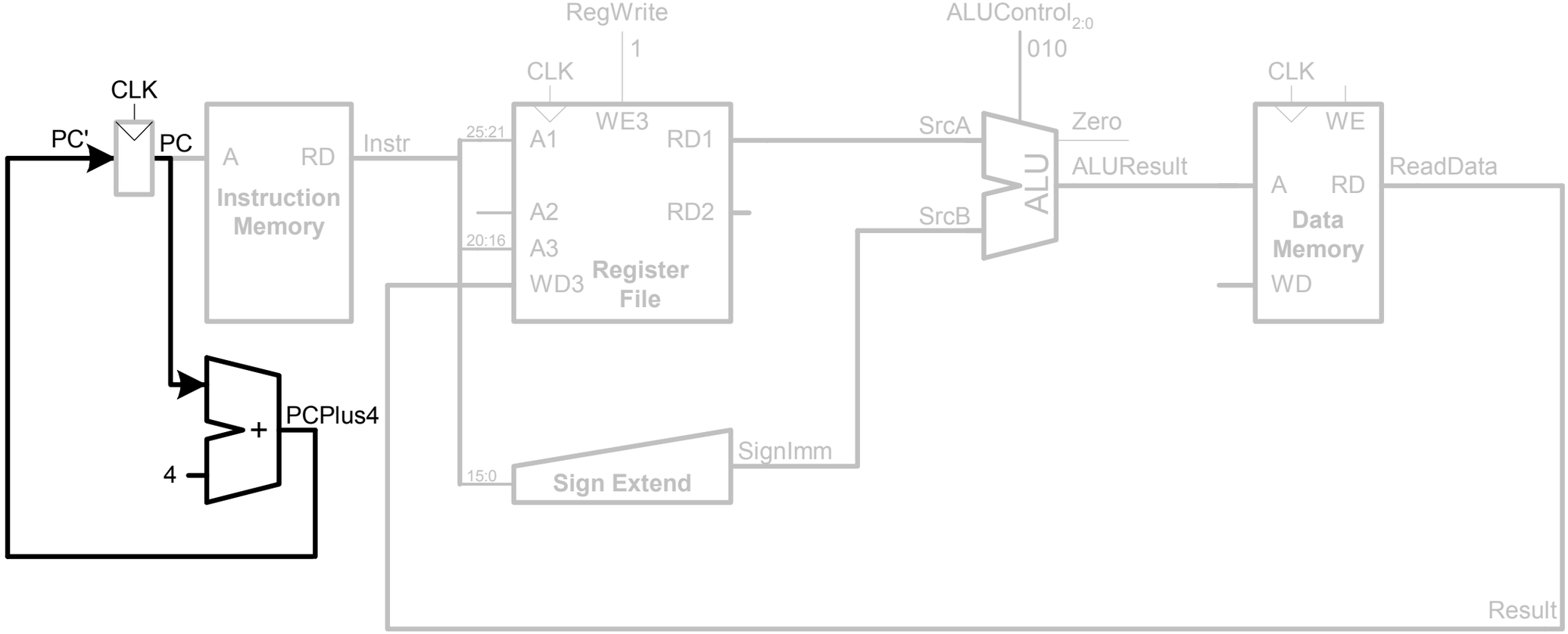
- Determine whether values in `rs` and `rt` are equal
- Calculate branch target address:

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

Single-Cycle Datapath: beq

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

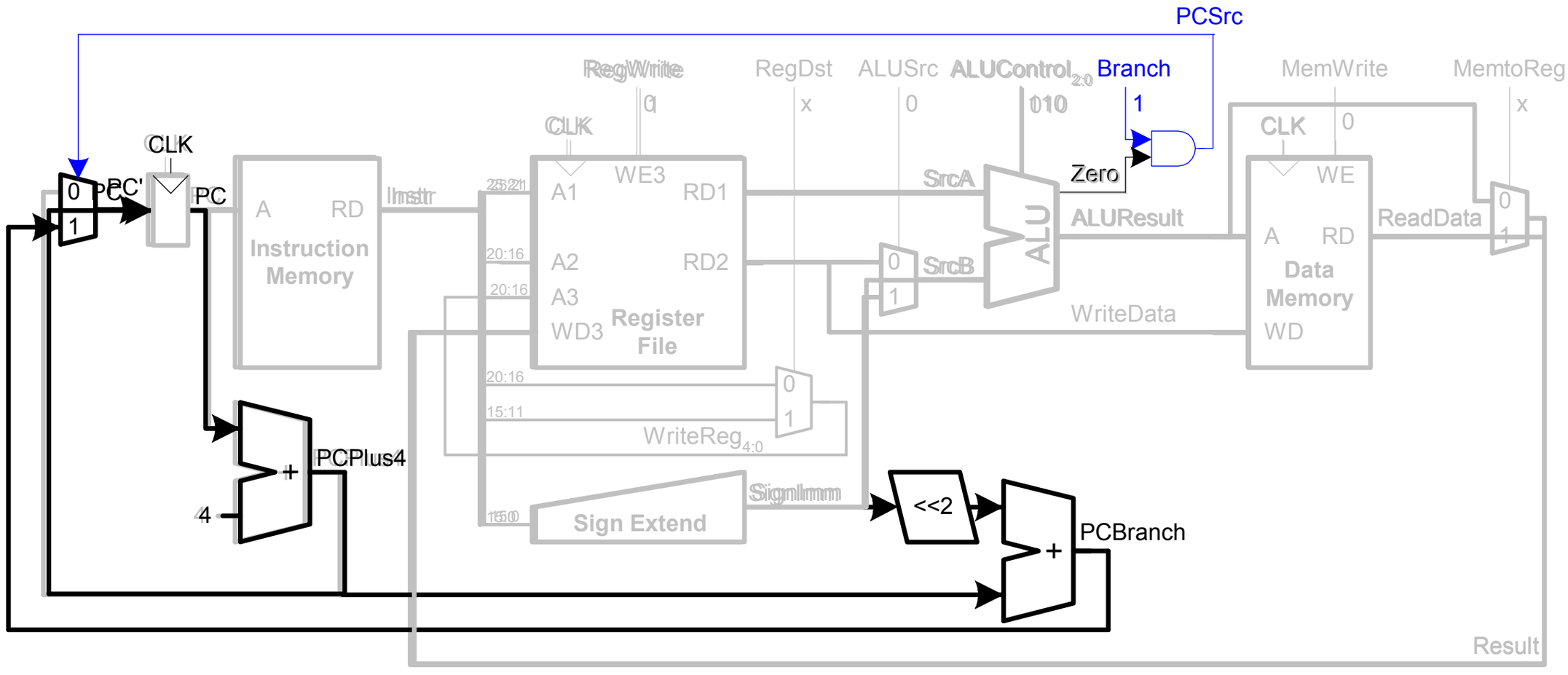
  BTA = (sign-extended immediate << 2) + (PC+4)



Single-Cycle Datapath: beq

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

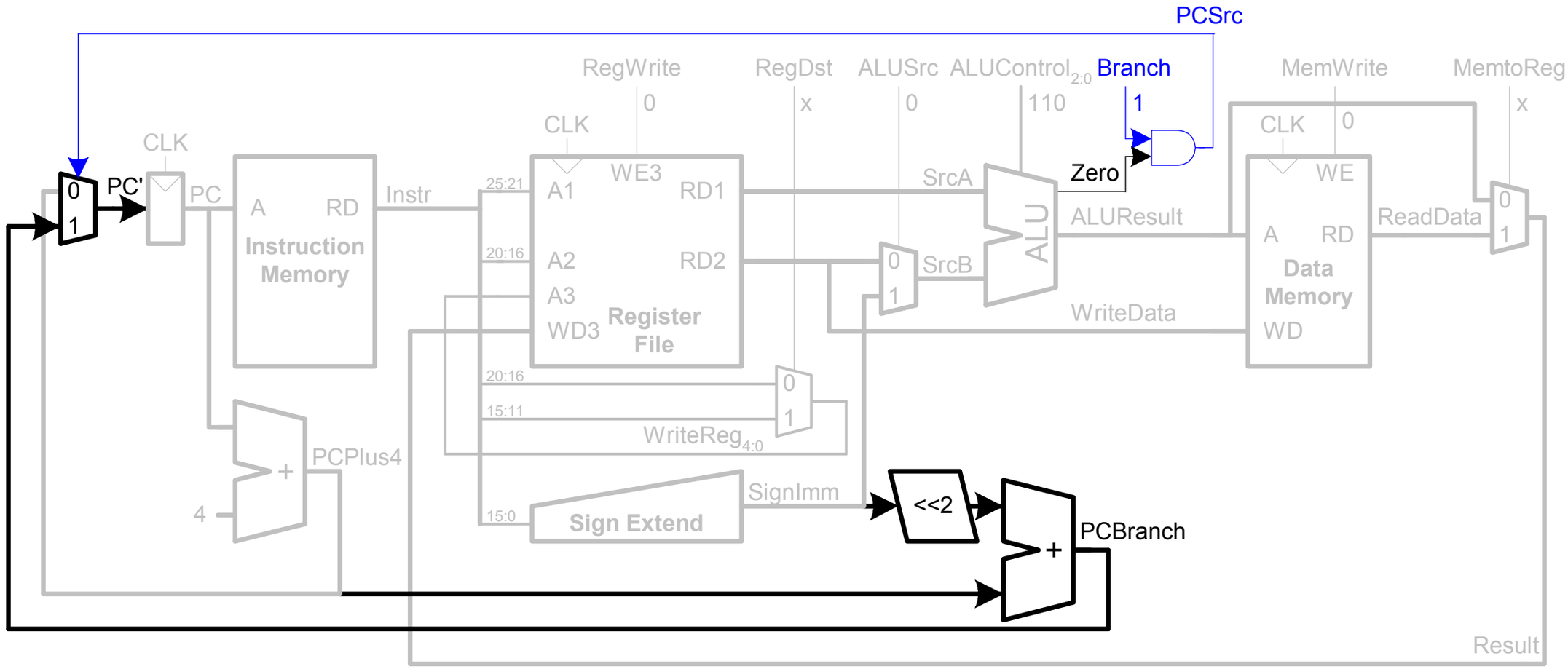
  BTA = (sign-extended immediate << 2) + (PC+4)



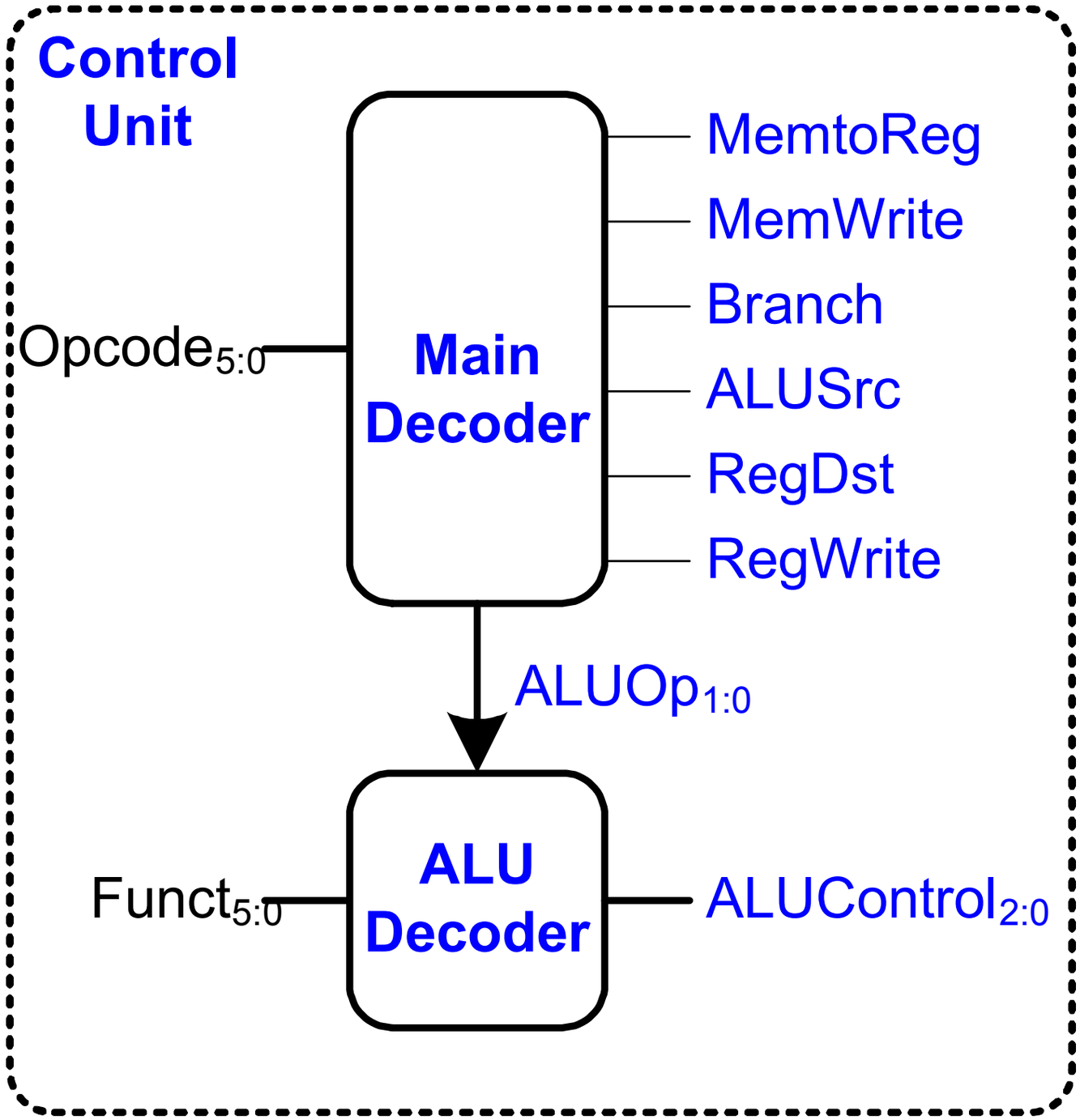
Single-Cycle Datapath: beq

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

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

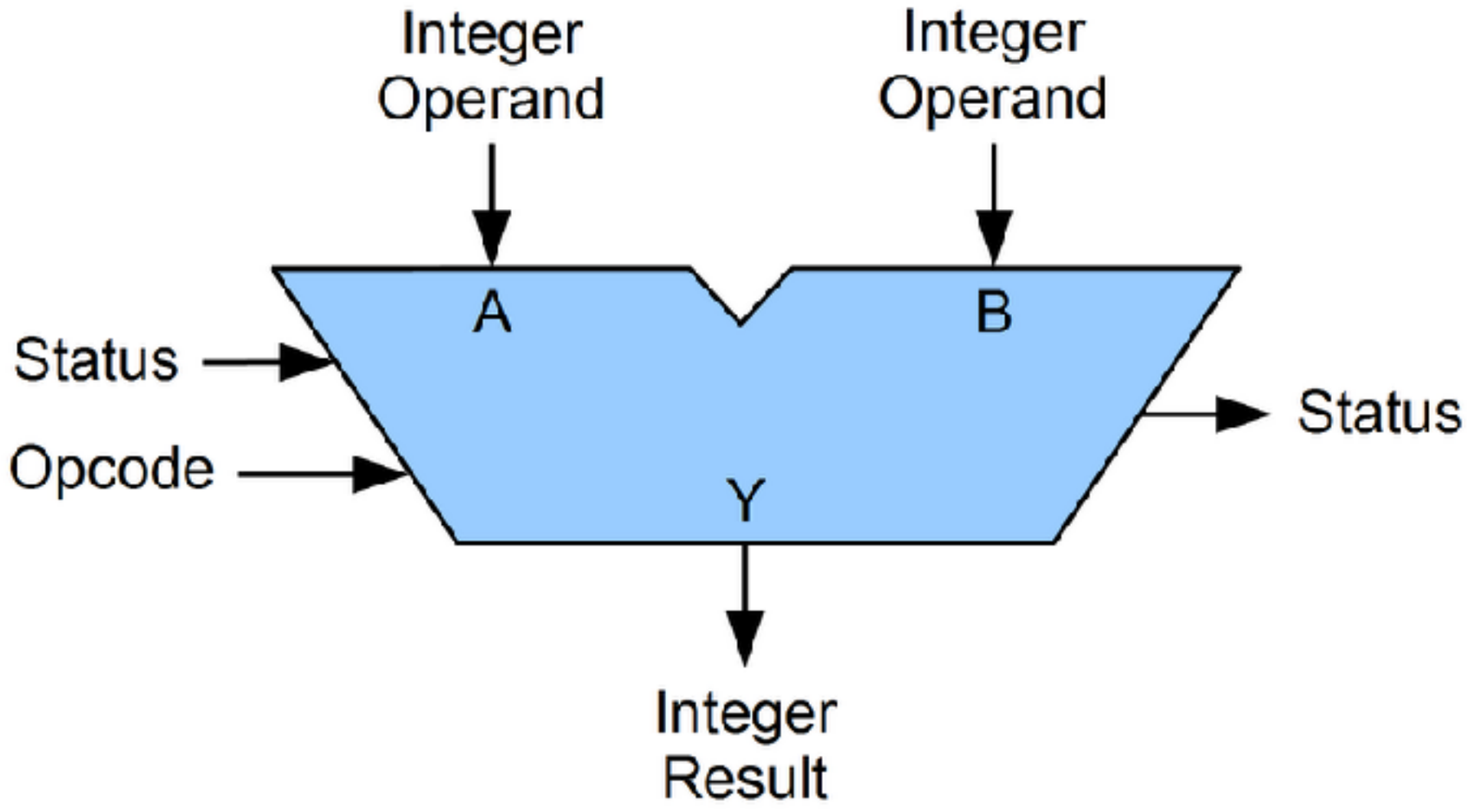


Design of the Single-Cycle Control



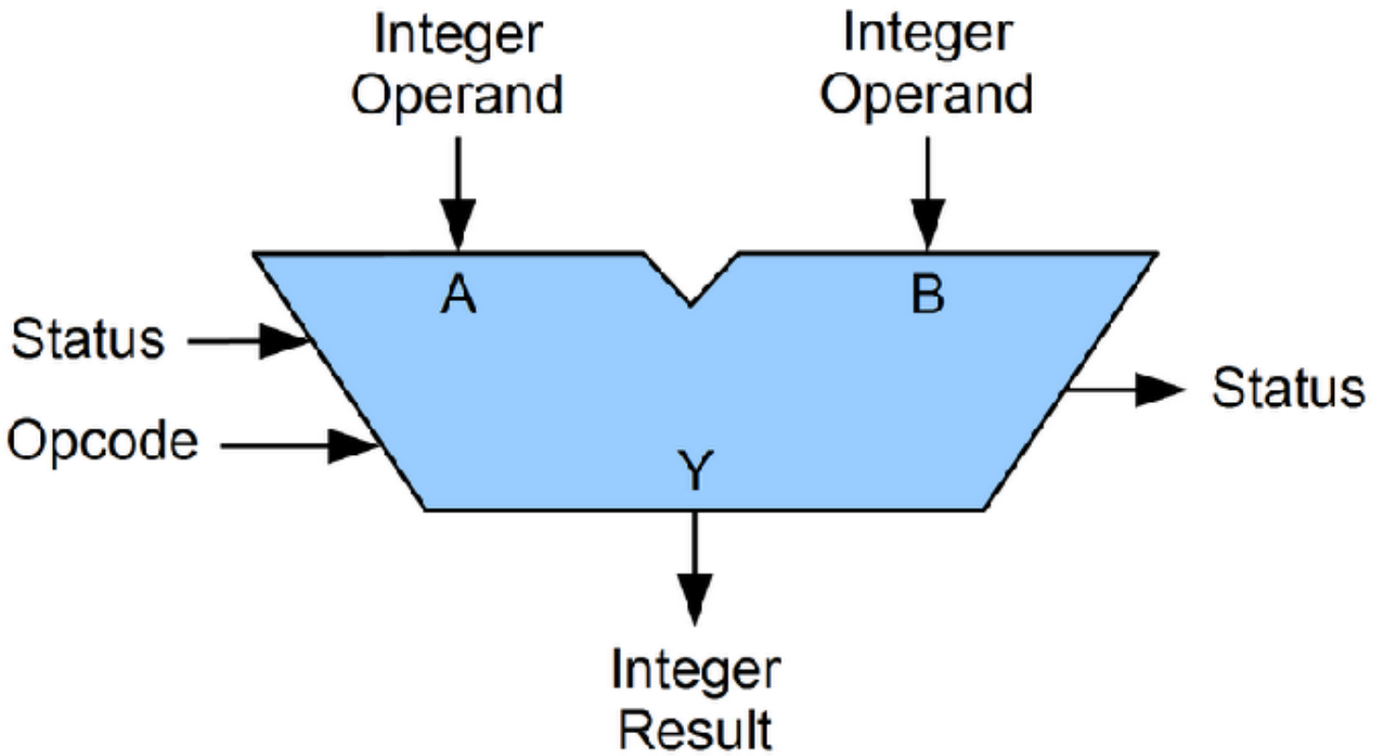
ALU: Arithmetic Logic Unit

- Digital circuit that performs arithmetic and bitwise operations



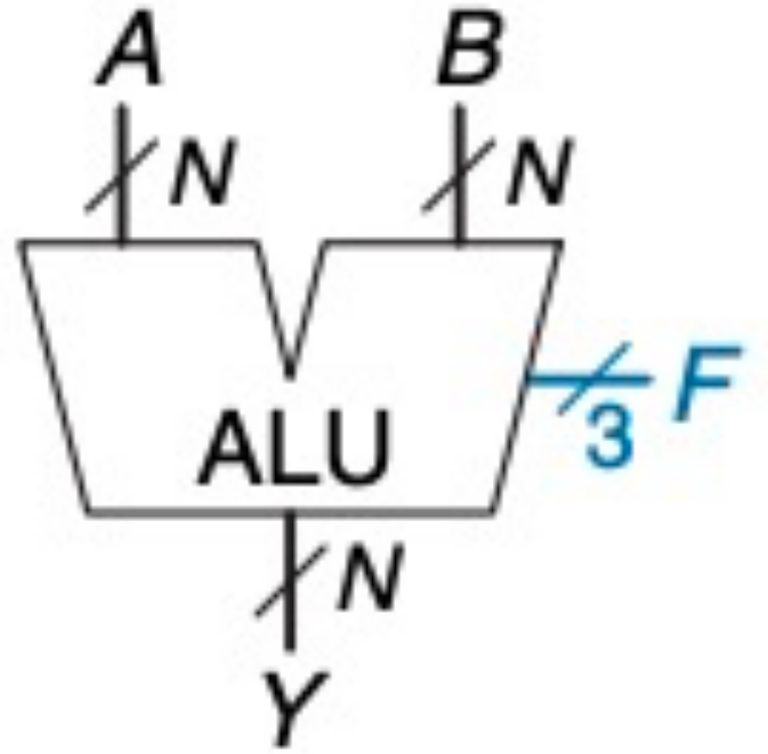
ALU: Arithmetic Logic Unit

- Digital circuit that performs arithmetic and bitwise operations
  - Data, Opcode, Status
- Arithmetic
  - Add, Add with carry, Subtract, Subtract with borrow …
- Bite-wise logic
  - AND, OR, XOR …



ALU Schematic

- Abstraction of ALU, three components
  - Two inputs (A, B)
  - One output (Y)
  - Control signal F (3-bit)

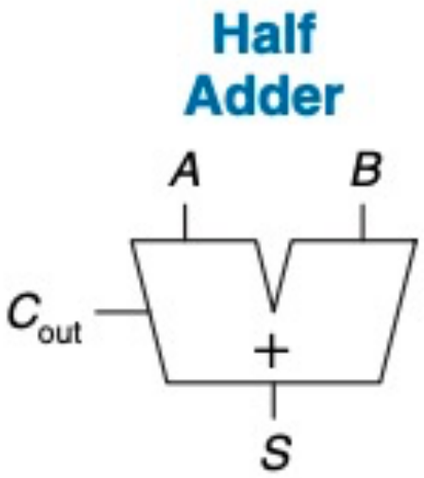


$F_{2:0}$	Function
000	A AND B
001	A OR B
010	A + B
011	not used
100	A AND $\overline{B}$
101	A OR $\overline{B}$
110	A – B
111	SLT

SLT: Set less than (later)

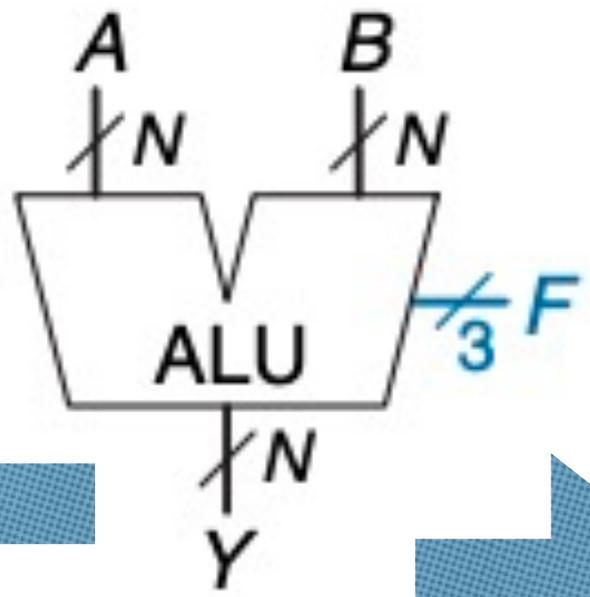
ALU Operations: Addition

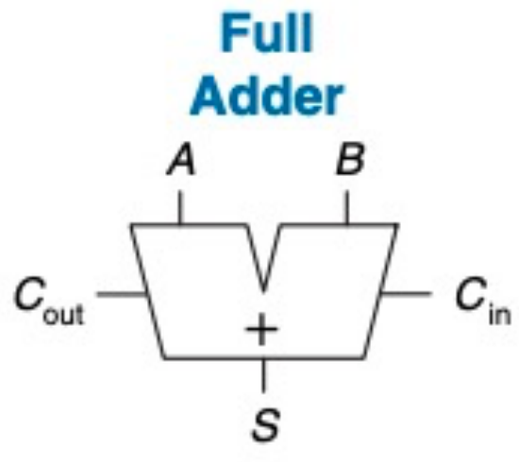
- Adder, already introduced
- Replace the ALU with +



A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$S = A \oplus B$$
$$C_{out} = AB$$

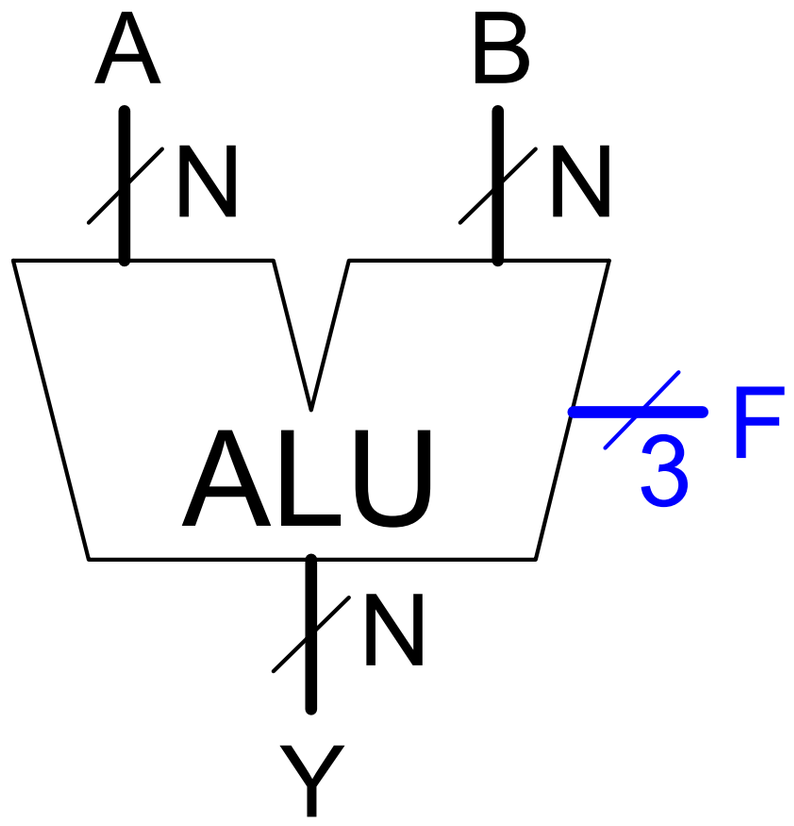




C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

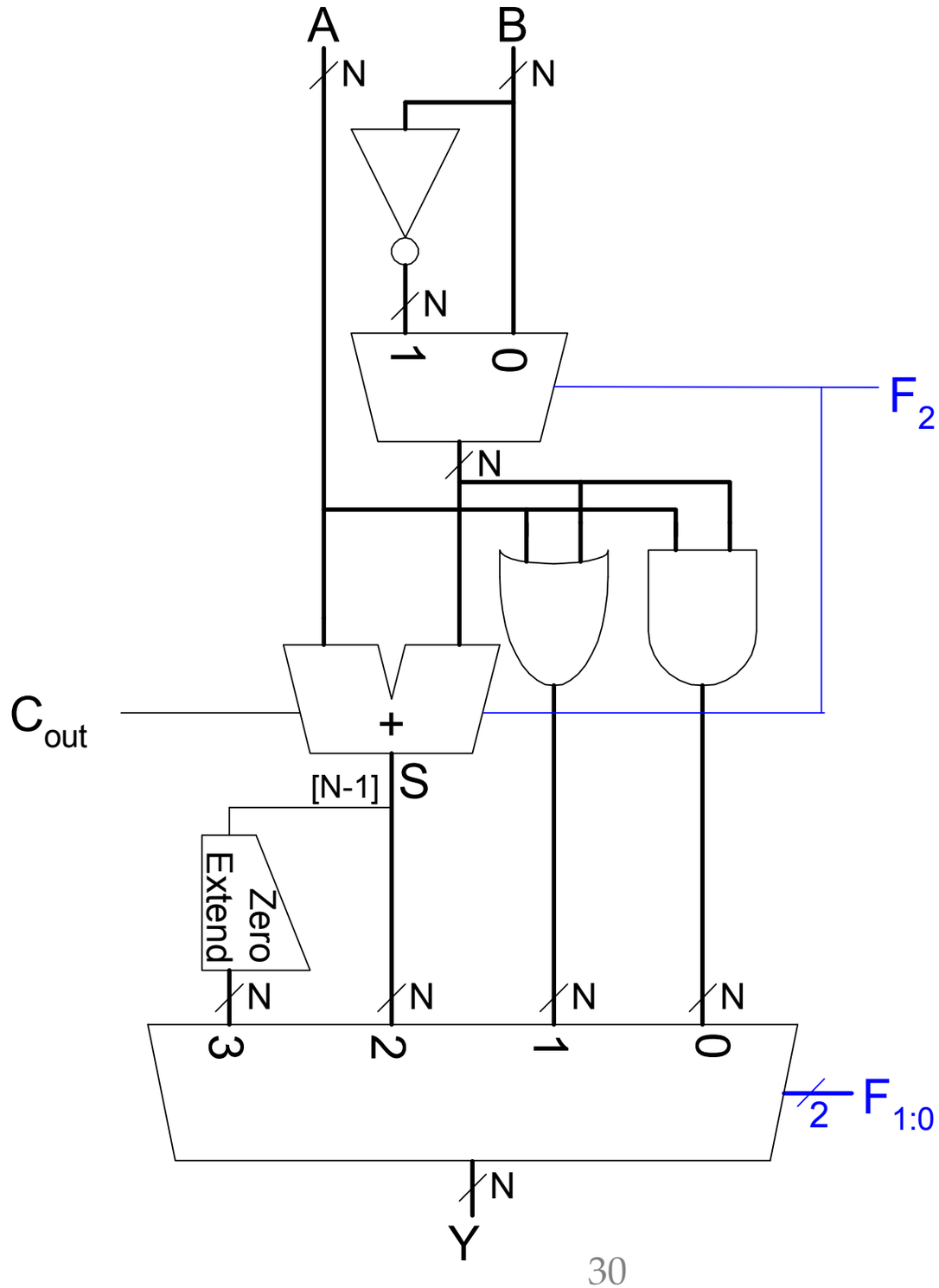
$$S = A \oplus B \oplus C_{in}$$
$$C_{out} = AB + AC_{in} + BC_{in}$$

Review: ALU



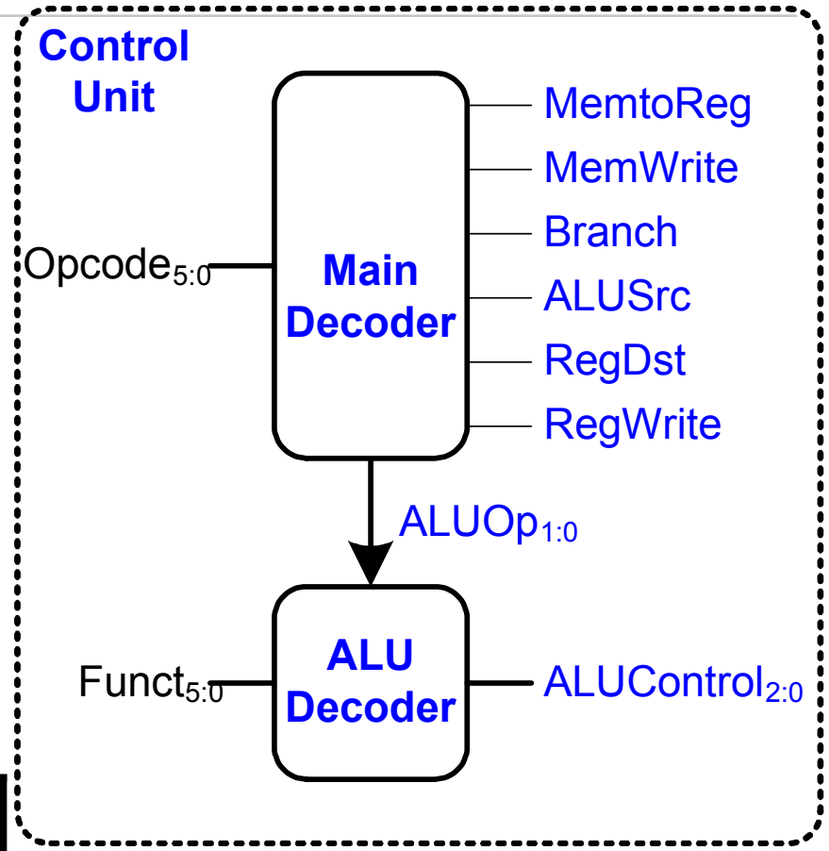
$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT

Review: ALU



Control Unit: ALU Decoder

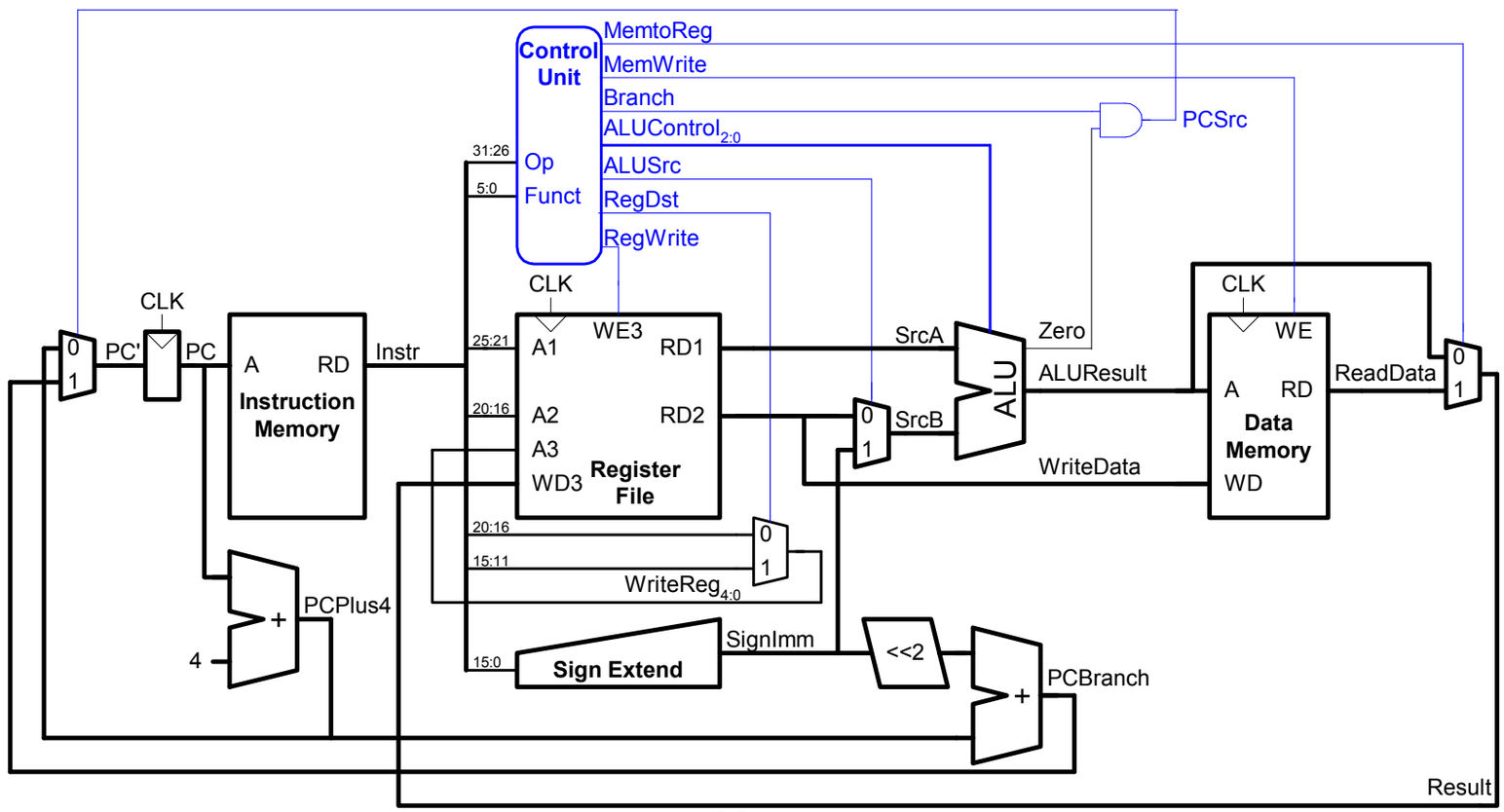
$ALUOp_{1:0}$	Meaning
00	Add
01	Subtract
10	Look at Funct
11	Not Used



$ALUOp_{1:0}$	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							

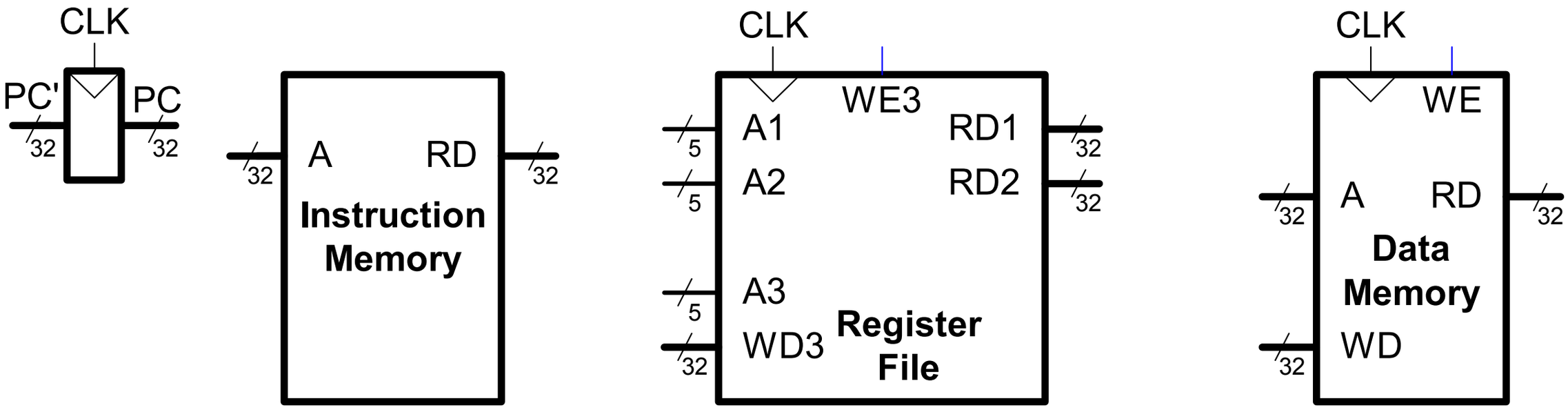


Control Unit: Main Decoder

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

Review: What We Have So Far

MIPS State Elements



- Instruction memory, register file, and data memory, are being **read** as **combinational circuit, i.e., data change follows address change, while the writing follows clock-edge**
- **Benefits: all elements are synchronized, while MIPS processor is a FSM!**

Overview: Single-Cycle Control

R-Type

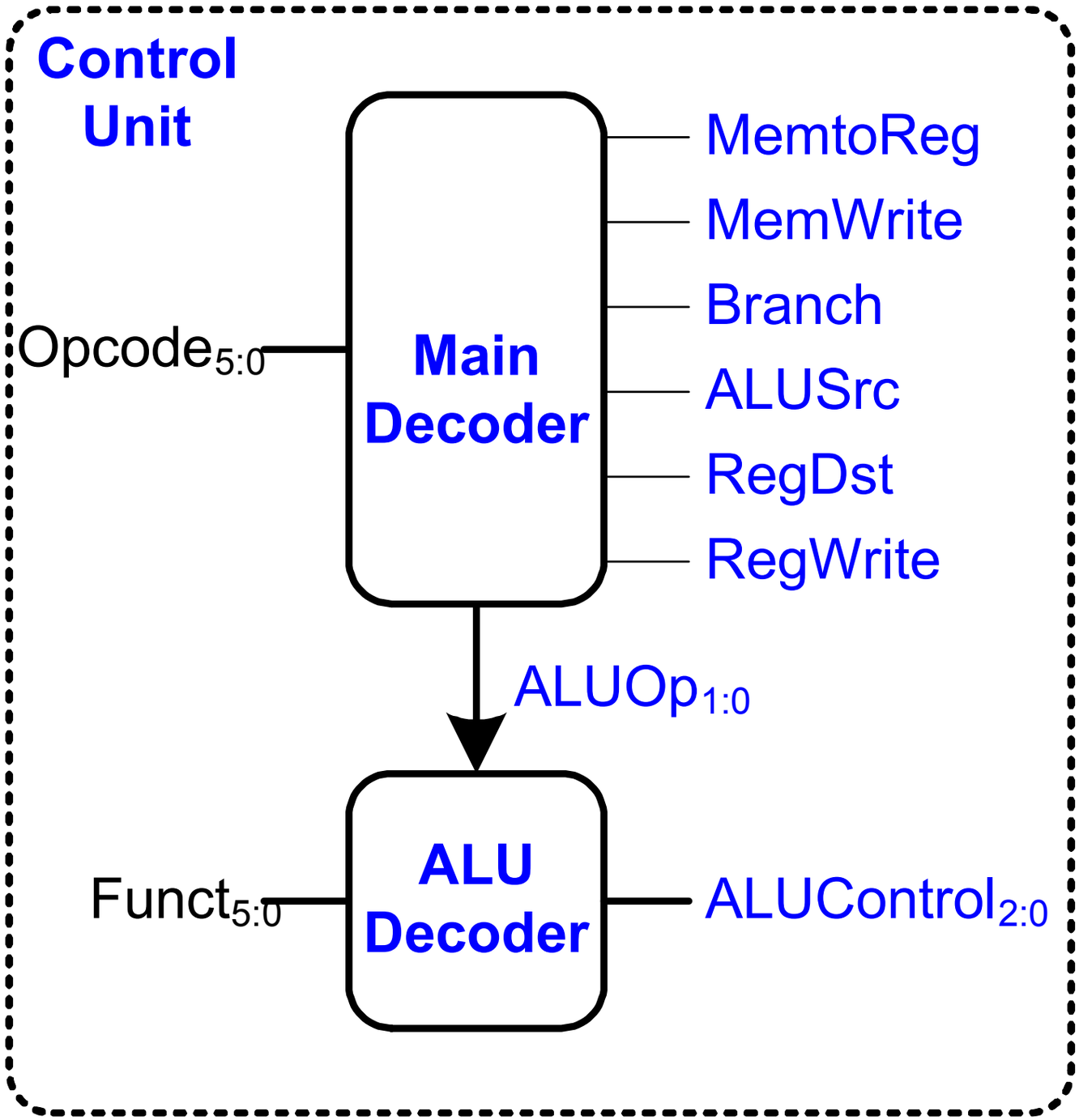
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

I-Type

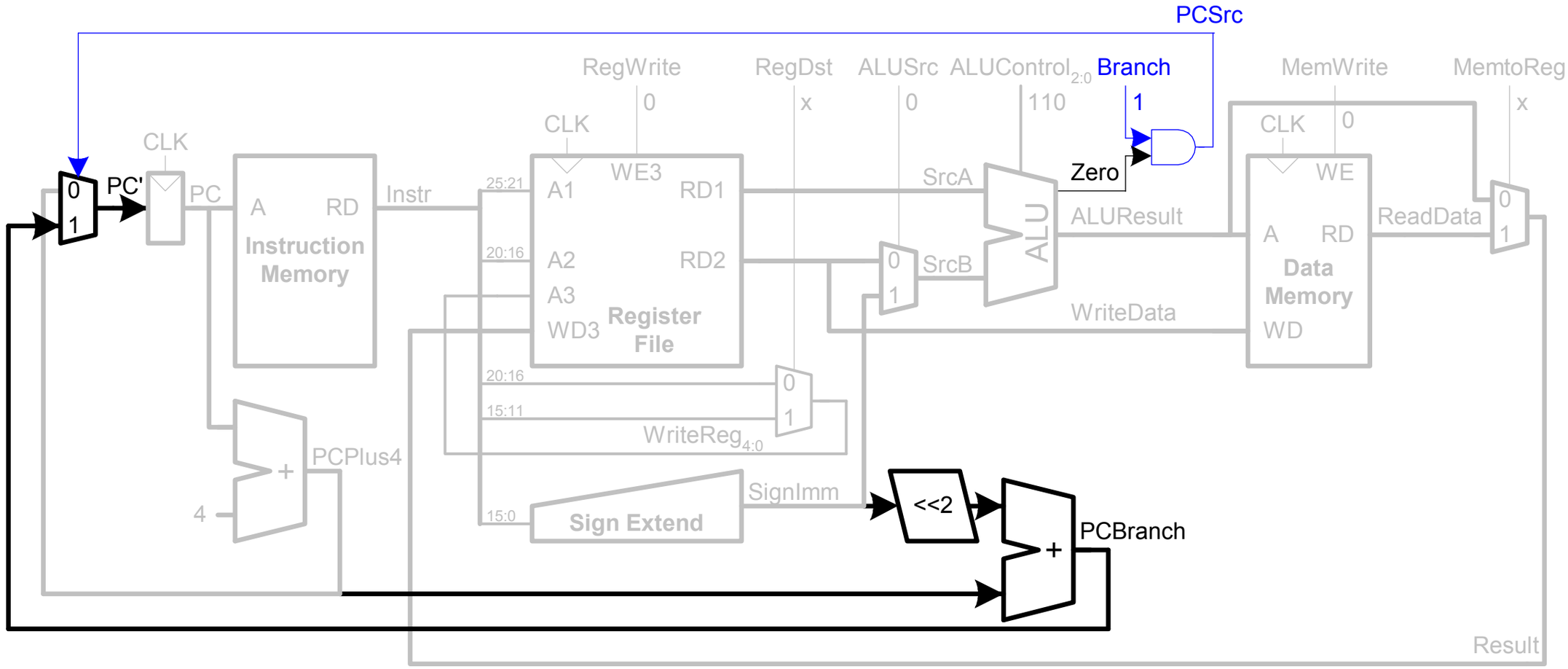
op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

J-Type

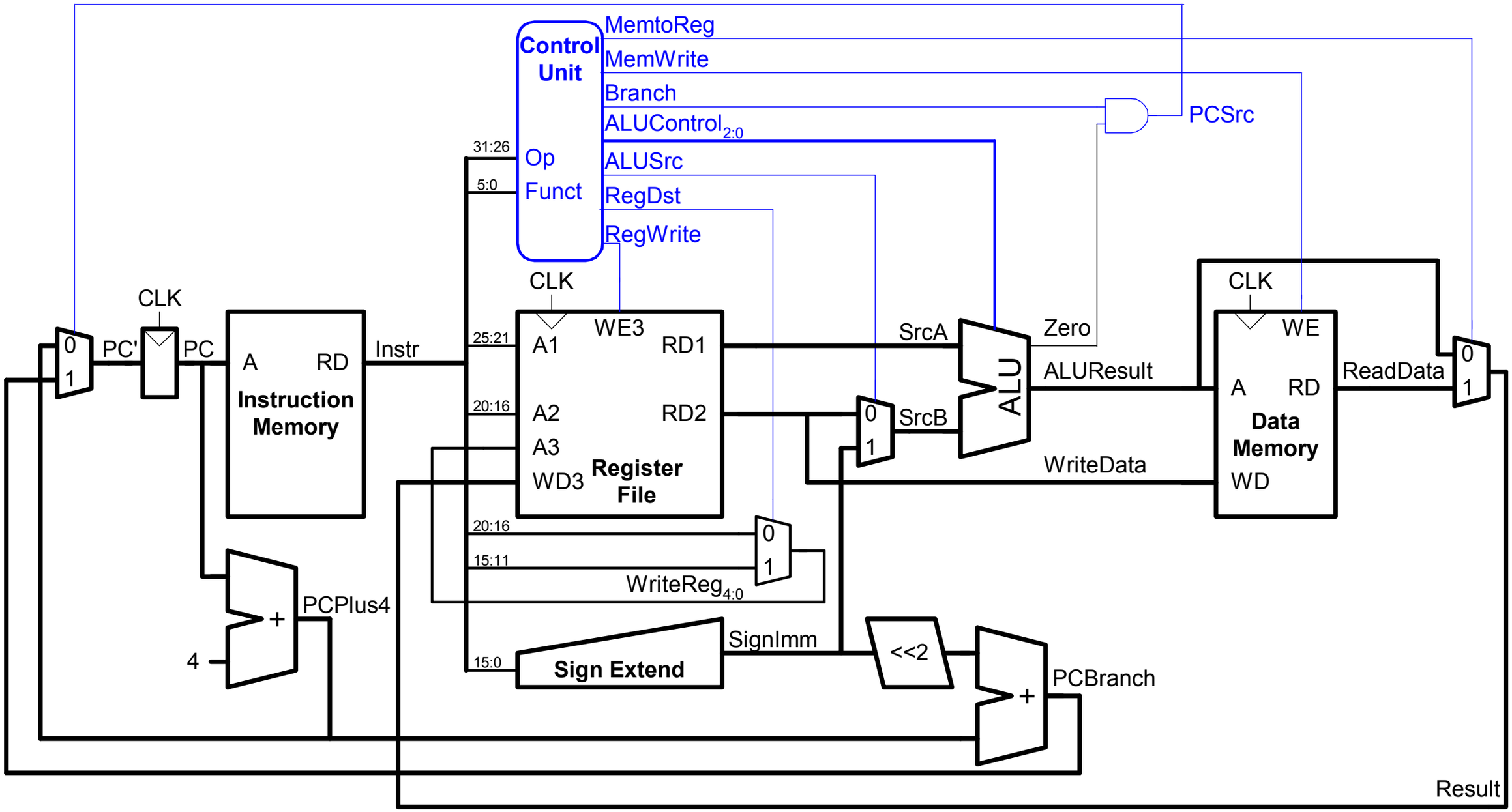
op	addr
6 bits	26 bits



Single-Cycle Datapath



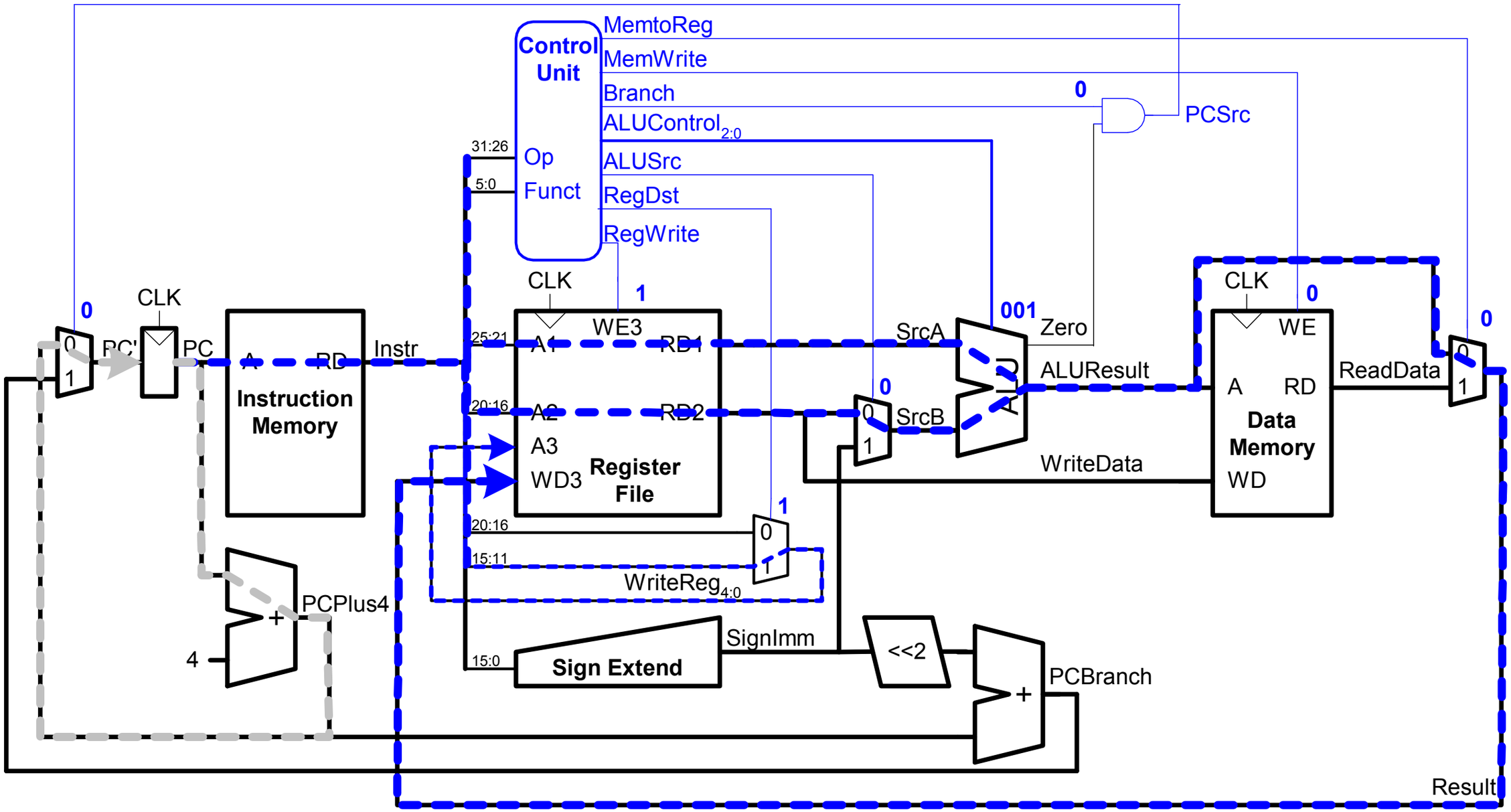
Single-Cycle Processor



Single-Cycle Datapath: or

or	`or $1,$2,$3`	$1=$2|$3	Bitwise OR

Single-Cycle Datapath: or

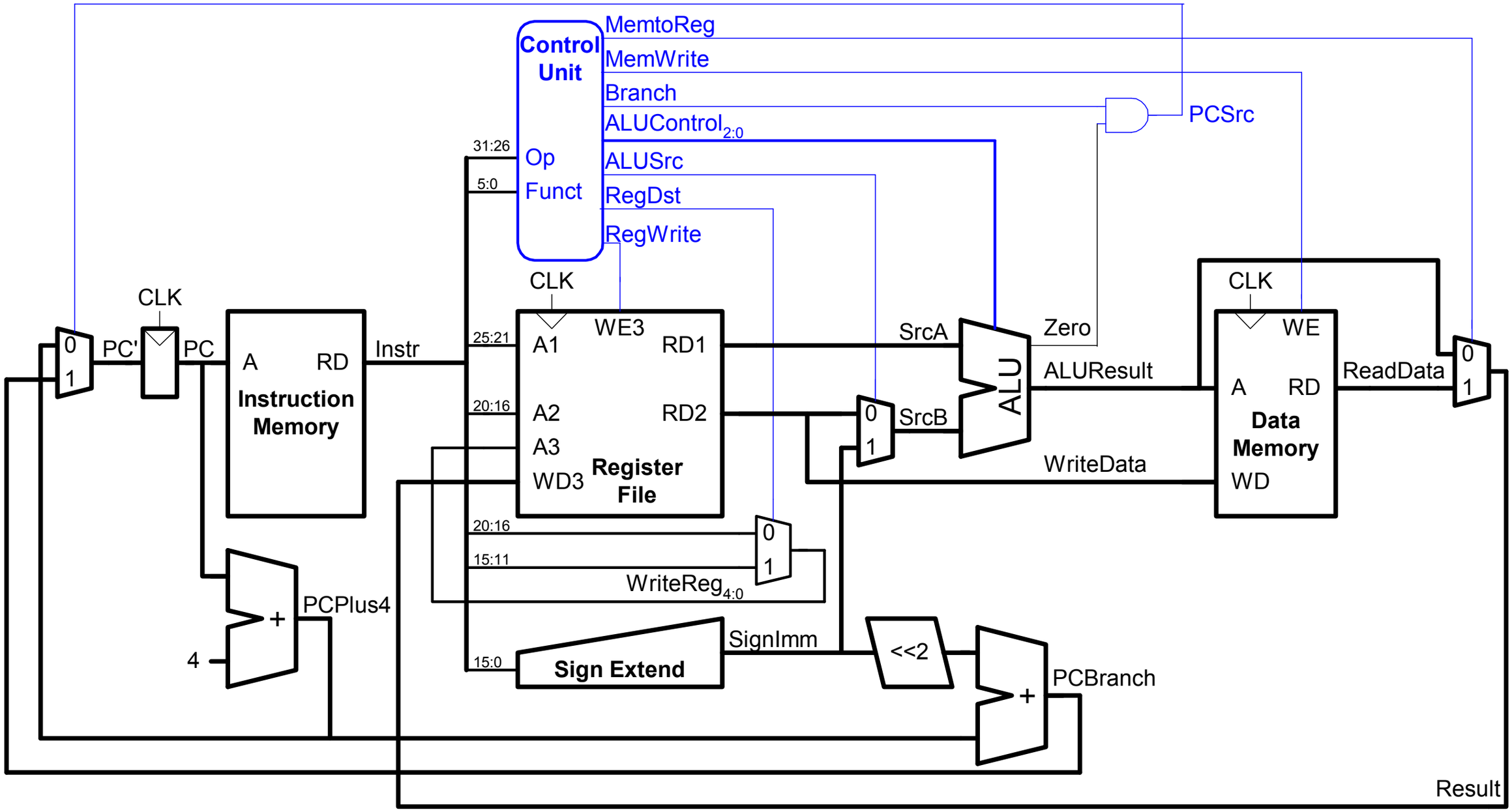


or	`or $1,$2,$3`	$1=$2|$3	Bitwise OR

Extended Functionality: addi

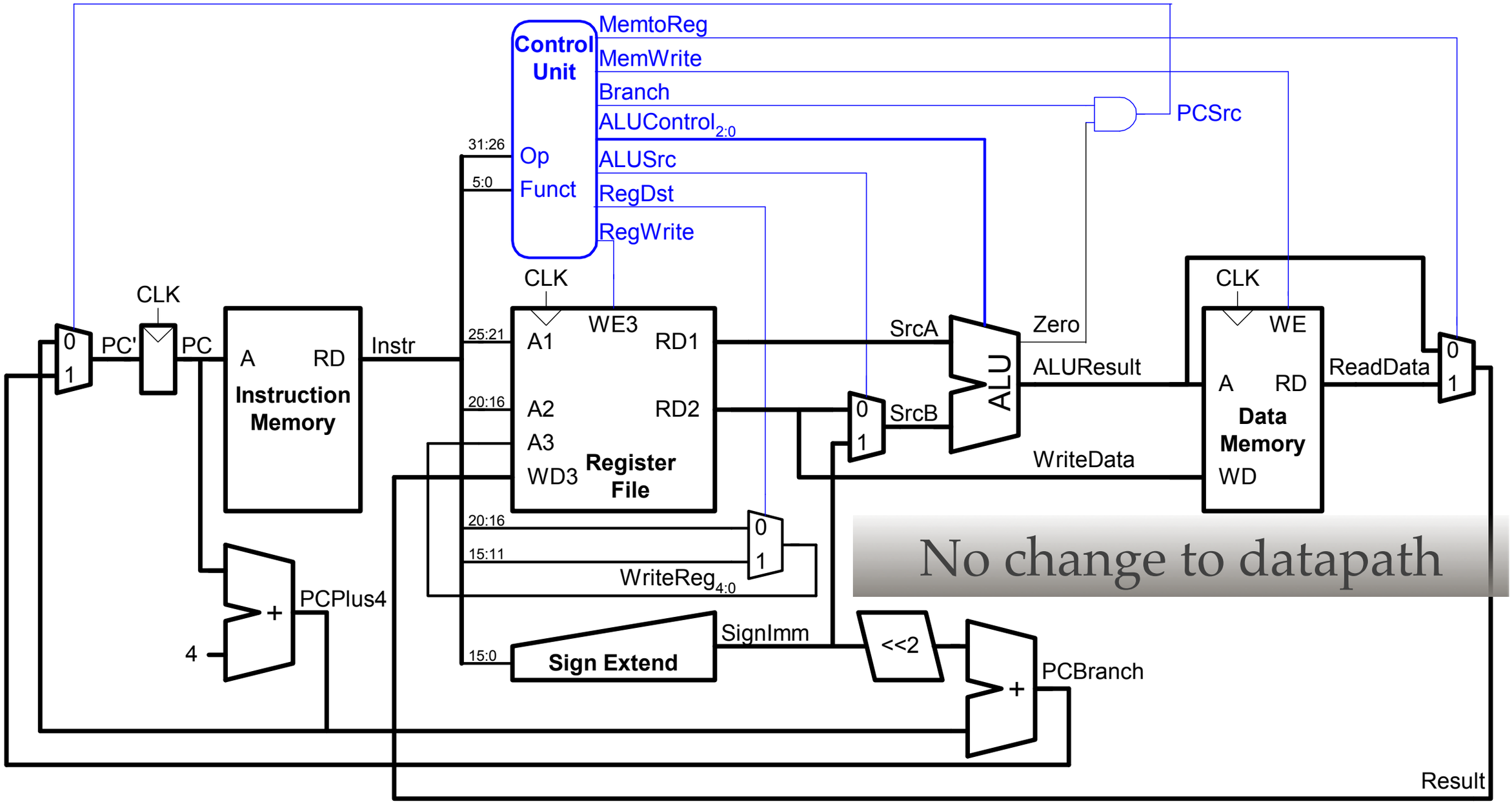
and immediate	`andi $1,$2,100`	$1=$2&100	Bitwise AND with immediate value

Extended Functionality: addi



and immediate	`andi $1,$2,100`	$1=$2&100	Bitwise AND with immediate value

Extended Functionality: addi



No change to datapath

and immediate	`andi $1,$2,100`	$1=$2&100	Bitwise AND with immediate value

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

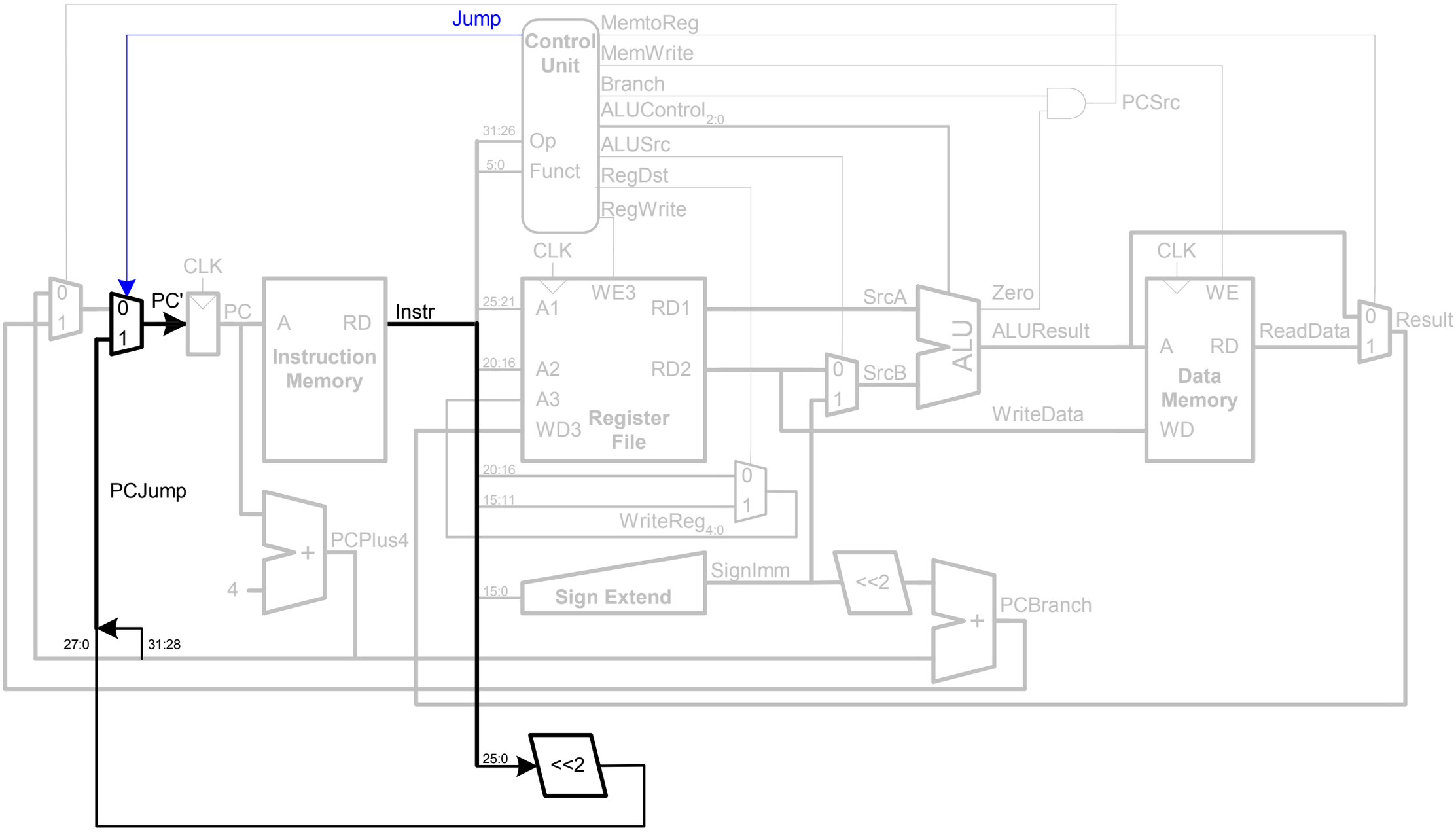
Unconditional Branching: jump (j)

```
# MIPS assembly
   addi $s0, $0, 4          # $s0 = 4
   addi $s1, $0, 1          # $s1 = 1
   j     target             # jump to target
   sra   $s1, $s1, 2
   addi  $s1, $s1, 1
   sub   $s1, $s1, $s0

target:
   add   $s1, $s1, $s0
```

Unconditional Branching: jump (j)

```
# MIPS assembly
  addi $s0, $0, 4          # $s0 = 4
  addi $s1, $0, 1          # $s1 = 1
  j    target              # jump to target
  sra  $s1, $s1, 2         # not executed
  addi $s1, $s1, 1         # not executed
  sub  $s1, $s1, $s0       # not executed

target:
  add  $s1, $s1, $s0       # $s1 = 1 + 4 = 5
```

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**

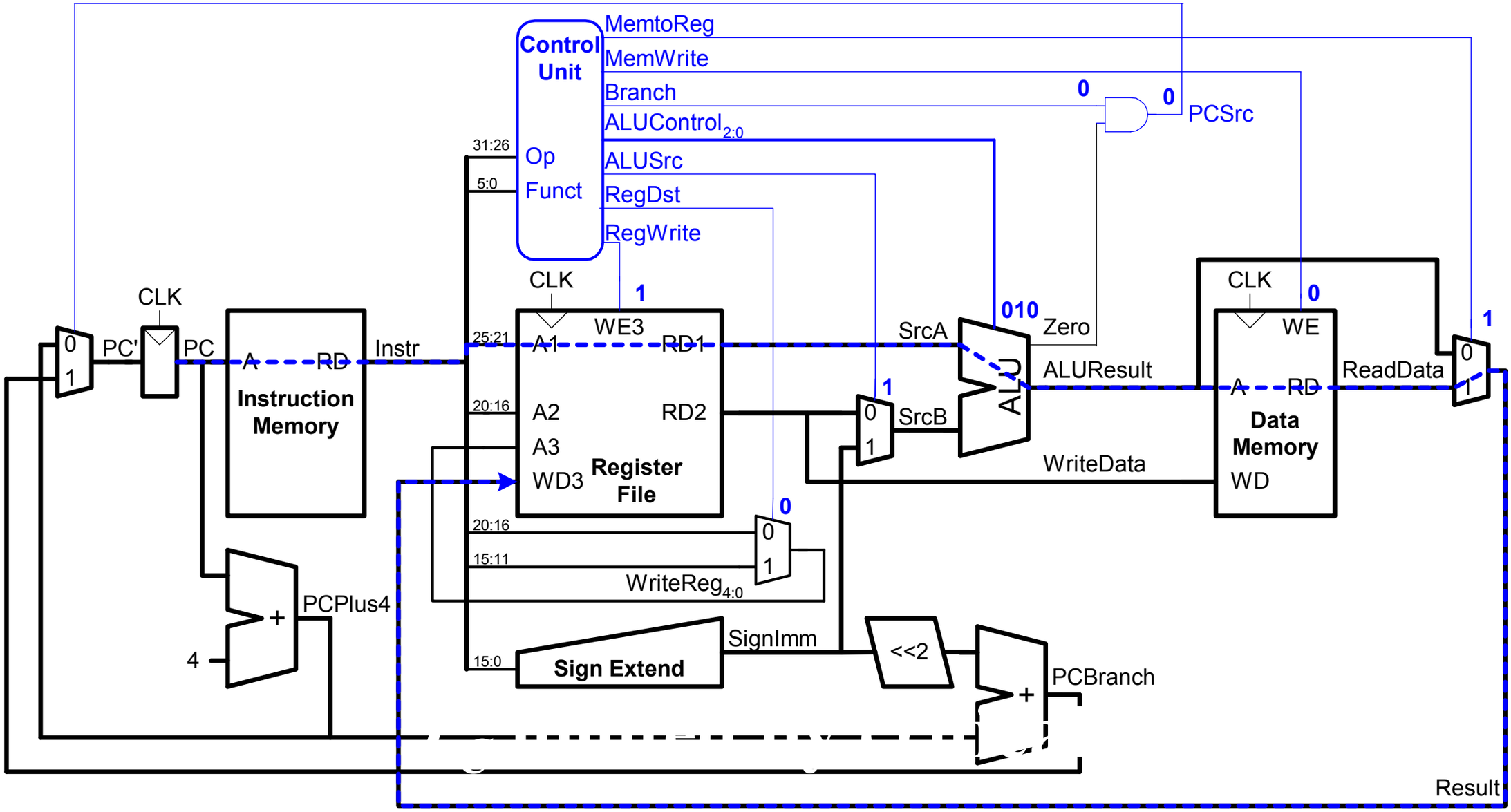
Review: Processor Performance

Program Execution Time

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

= # instructions x CPI x T_C

Single-Cycle Performance



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
  - $T_c = t_{pcq\_PC} + 2t_{\text{mem}} + t_{RF\text{read}} + t_{\text{mux}} + t_{\text{ALU}} + t_{RF\text{setup}}$

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

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

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**