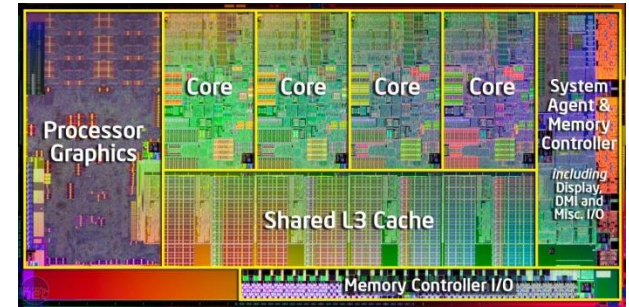




고려대학교
KOREA UNIVERSITY



COSE222, COMP212 Computer Architecture

Lecture 5. MIPS Processor Design

Single-Cycle MIPS #2

Prof. Taeweon Suh

Computer Science & Engineering

Korea University

Single-Cycle MIPS

- Again, keep in mind that microarchitecture is composed of 2 interacting parts
 - **Datapath**
 - **Control**
- Let's execute some example instructions on what we have designed so far
- Then, we are going to design control logic in detail

Single-Cycle MIPS - 1w

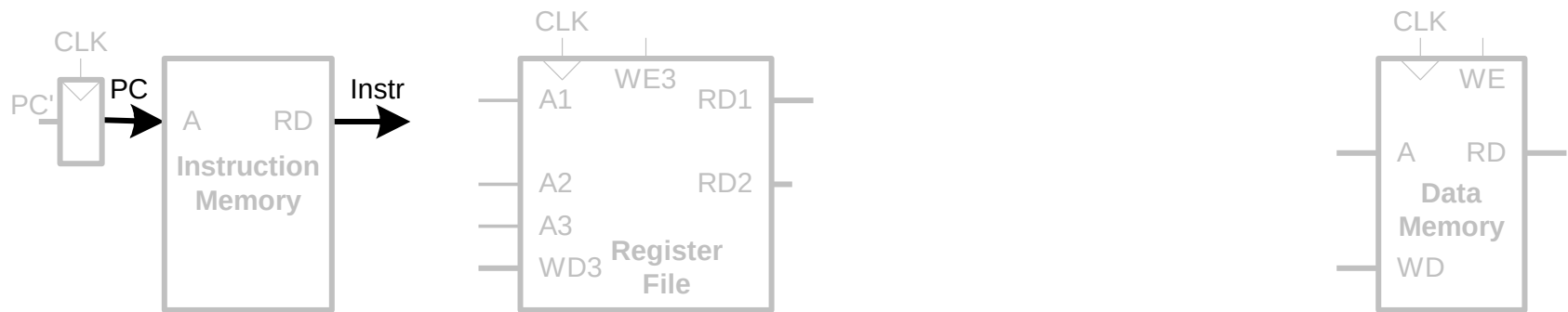
- Let's start with a memory access instruction - lw

$lw \$2, 80(\$0)$

- STEP 1:** Instruction Fetch

I-Type

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



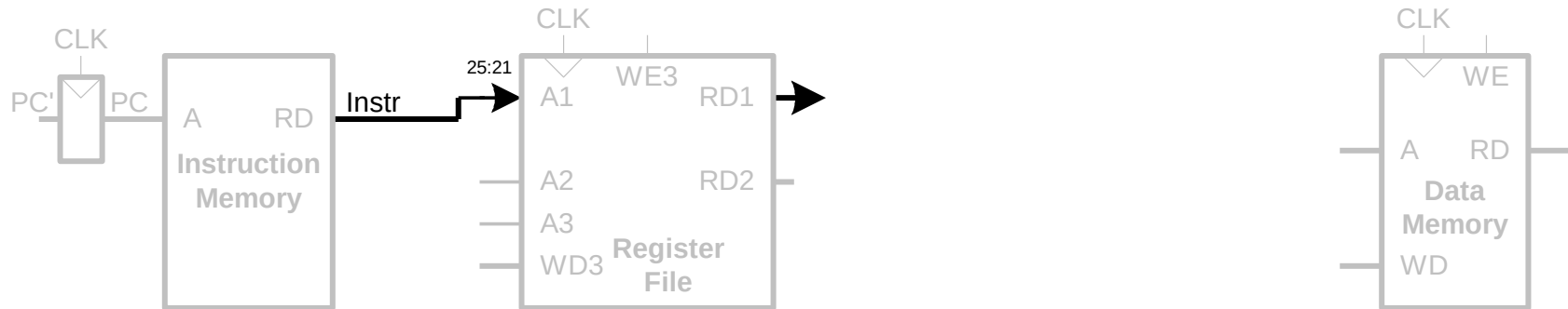
Single-Cycle MIPS - 1w

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

lw \$2, 80(\$0)

I-Type

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



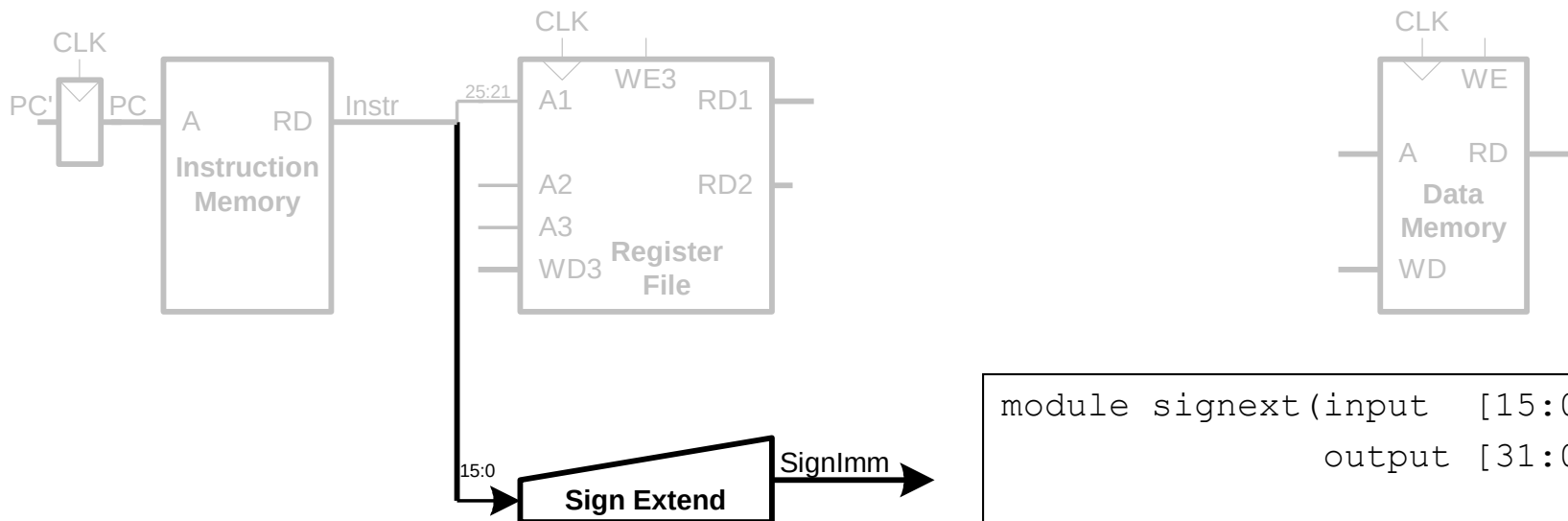
Single-Cycle MIPS - 1w

- **STEP 2: Decoding**
 - Sign-extend the immediate

lw \$2, 80(\$0)

I-Type

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



```
module signext(input  [15:0] a,
               output [31:0] y);

    assign y = {{16{a[15]}}, a};
endmodule
```

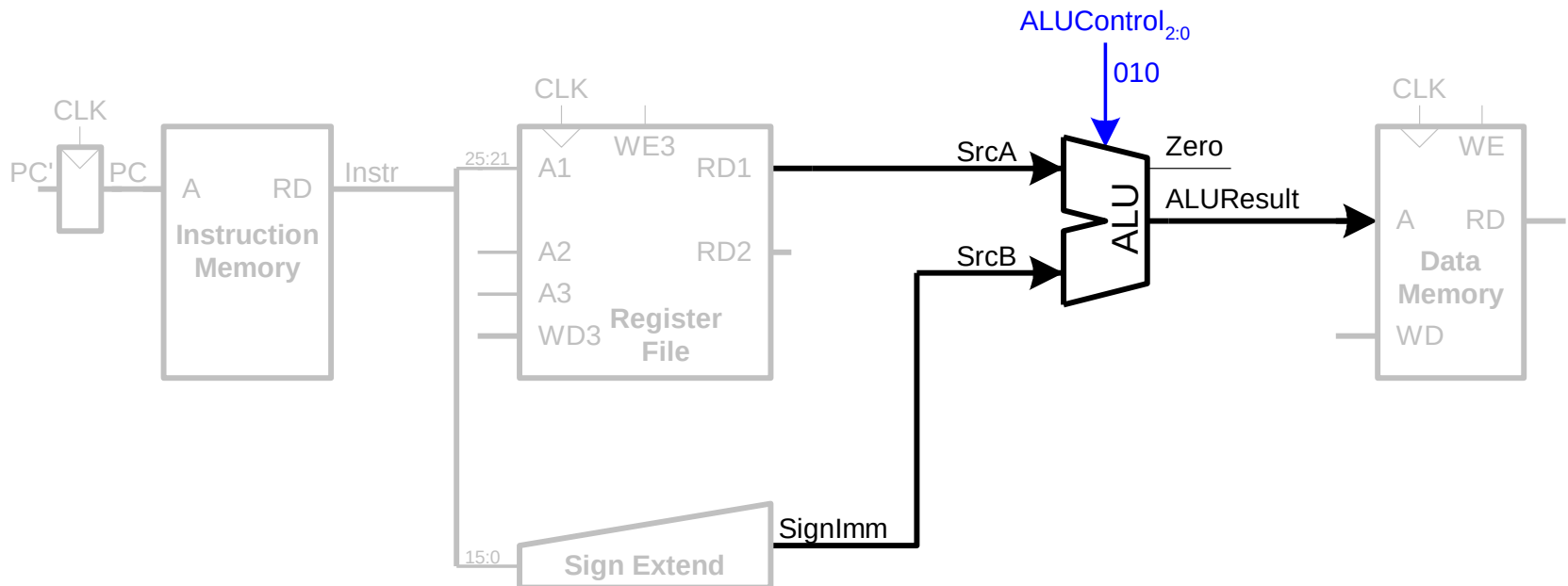
Single-Cycle MIPS - 1w

- **STEP 3: Execution**
 - Compute the memory address

1w \$2, 80 (\$0)

I-Type

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



Single-Cycle MIPS - 1w

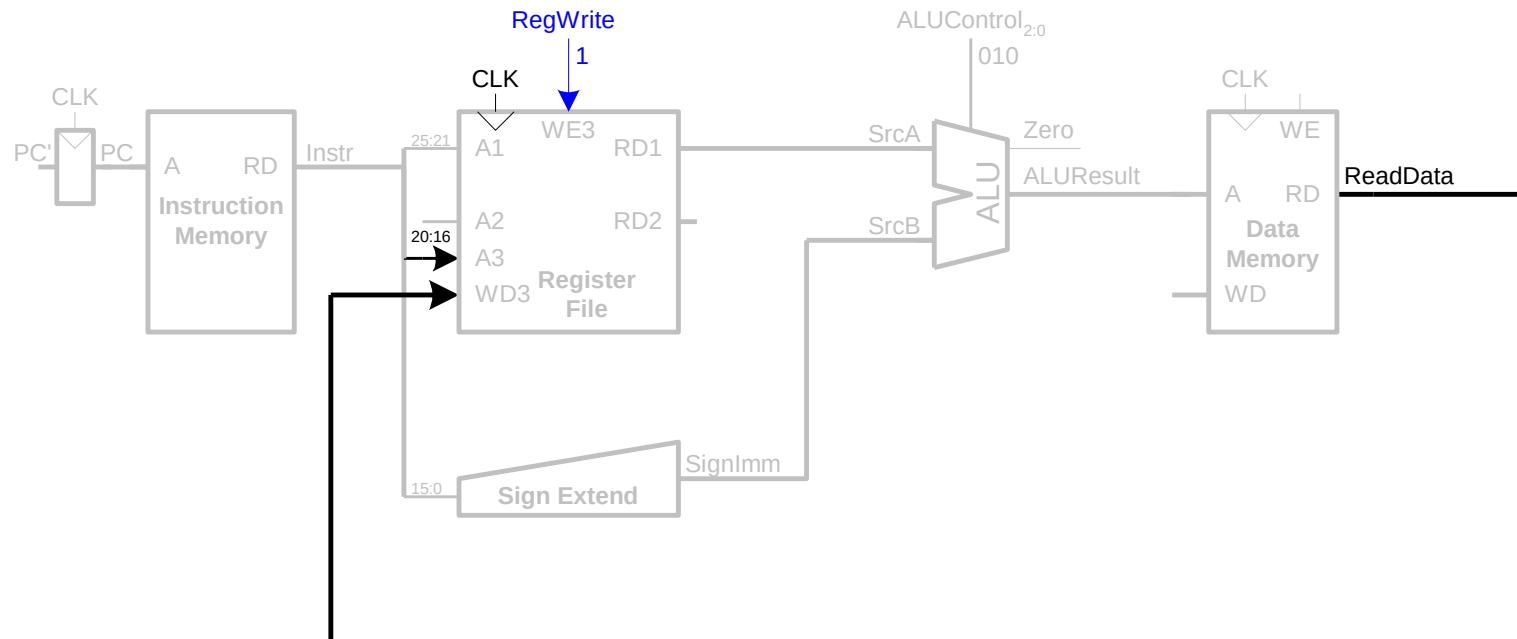
- **STEP 4: Execution**

- Read data from memory and write it to register file

1w \$2, 80 (\$0)

I-Type

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

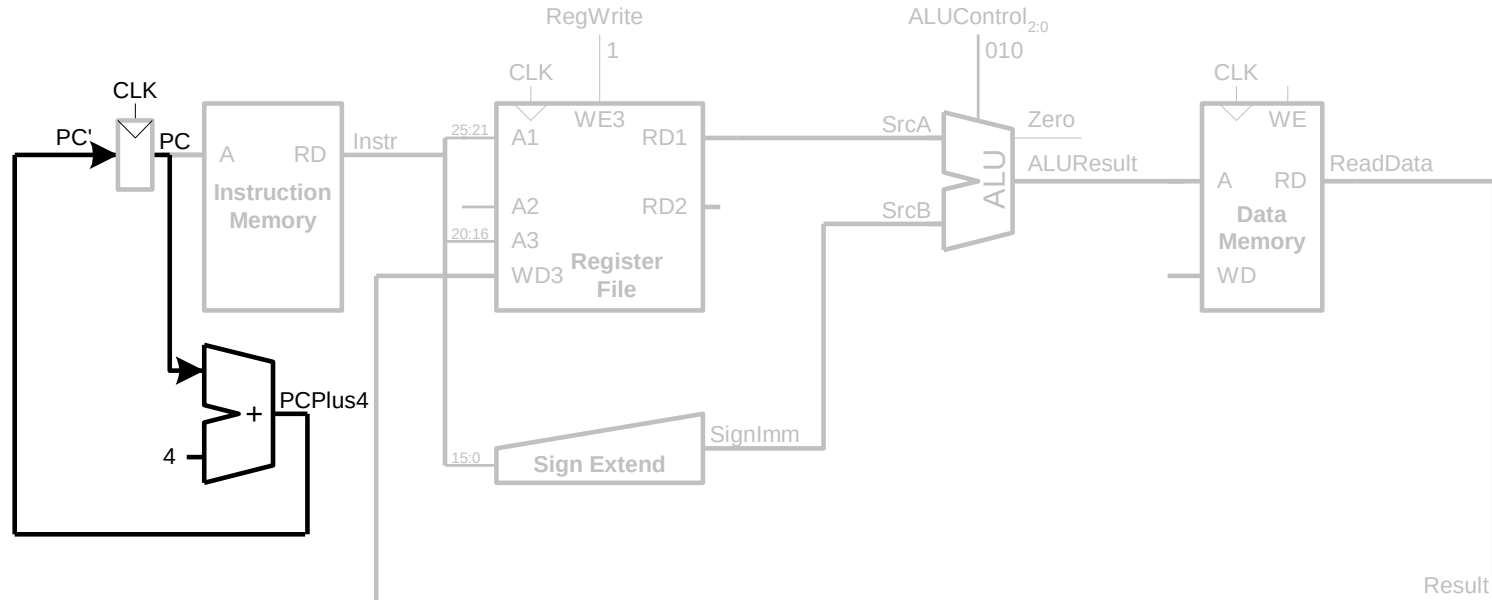


Single-Cycle MIPS – PC

- CPU starts fetching the next instruction from $PC+4$

```
module adder(input [31:0] a, b,  
             output [31:0] y);  
    assign y = a + b;  
endmodule
```

```
adder pcadd1(.a (pc),  
            .b (32'b100)  
            .y (pcplus4));
```



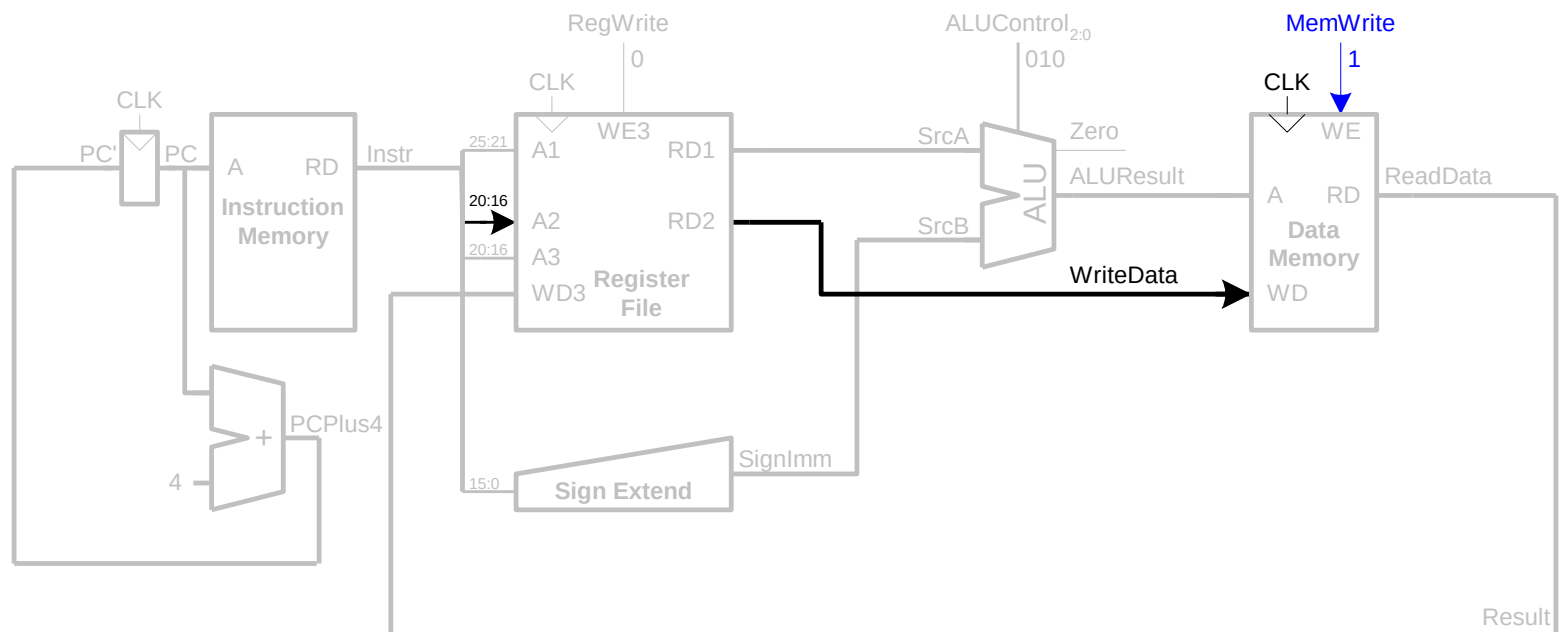
Single-Cycle MIPS - sw

- Let's execute another memory access instruction - sw
 - sw instruction needs to write data to memory

Example: **sw \$2, 84(\$0)**

I-Type

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

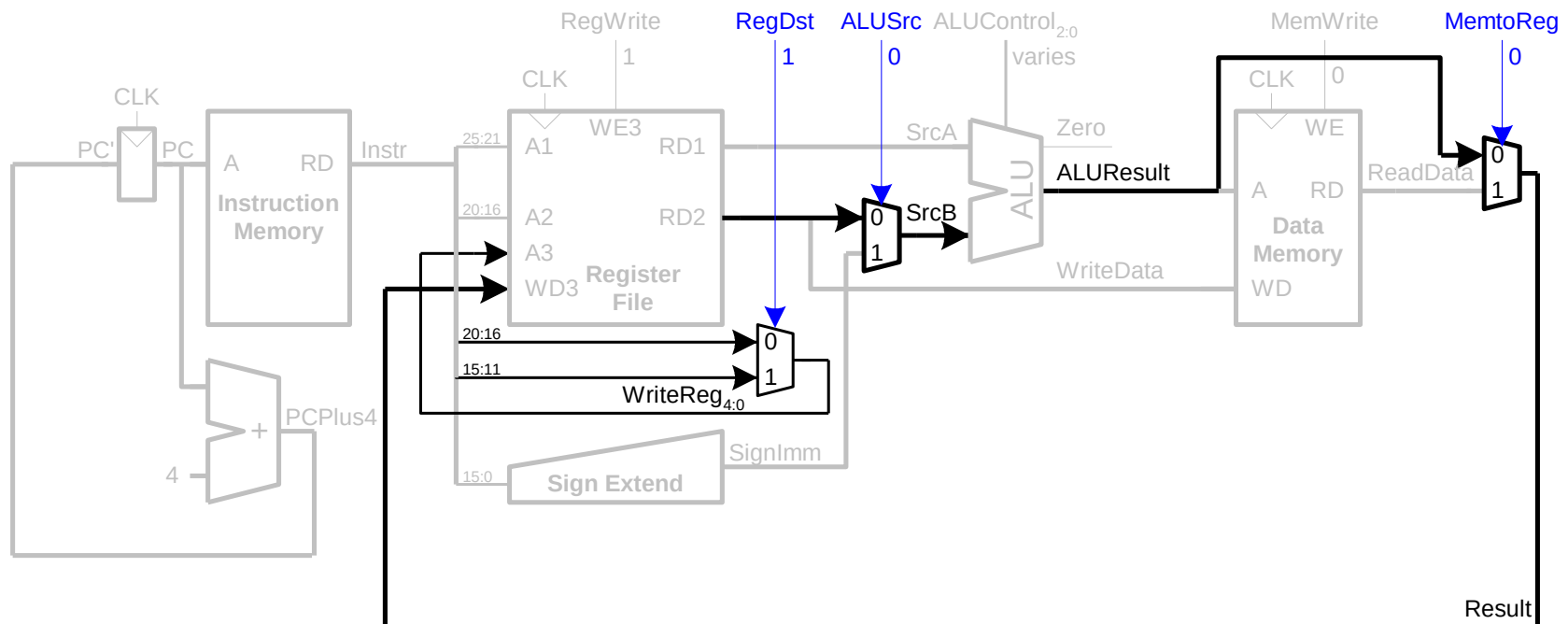


Single-Cycle MIPS - add, sub, and, or

- Let's consider arithmetic and logical instructions - add, sub, and, or
 - Write `ALUResult` to register file
 - Note that R-type instructions write to `rd` field of instruction (instead of `rt`)

R-Type

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



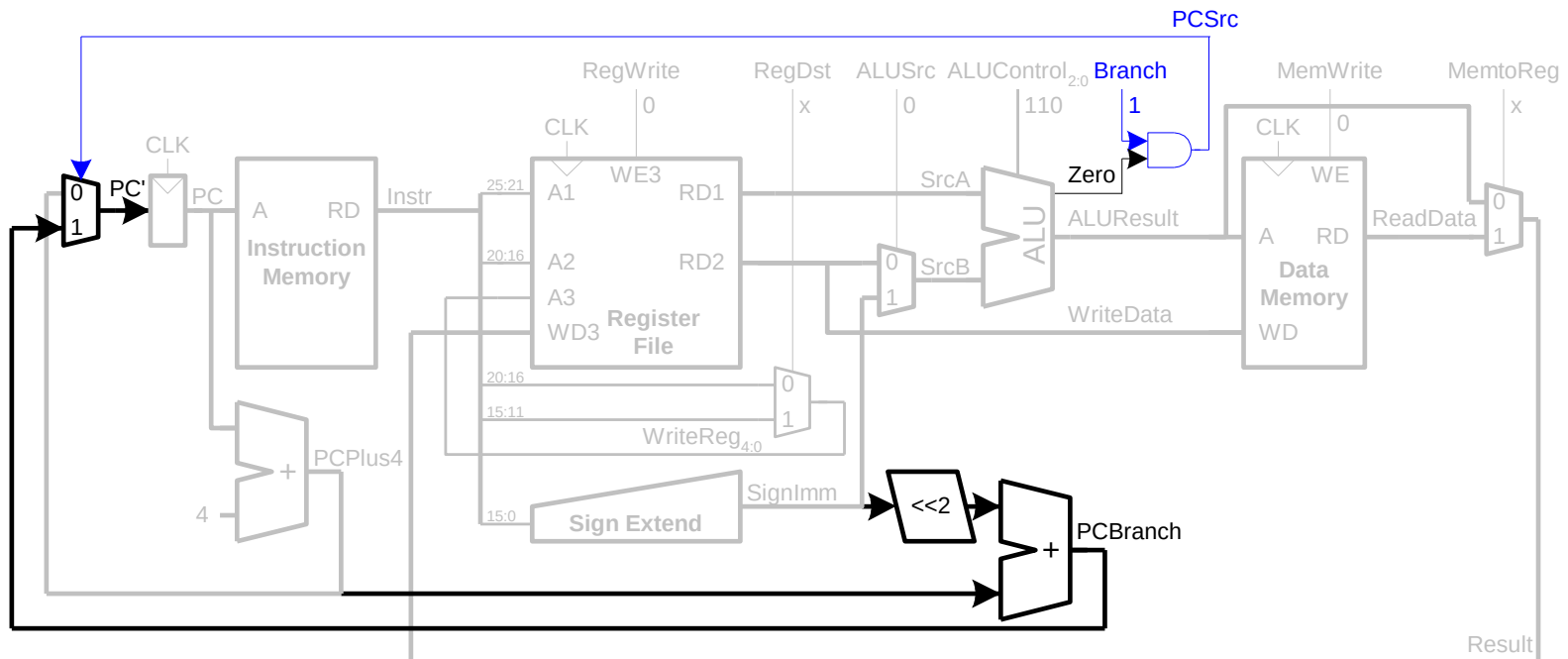
Single-Cycle MIPS - beq

- Let's consider a branch instruction - `beq`
 - Determine whether register values are equal
 - Calculate branch target address (BTA) from sign-extended immediate and PC+4

Example: **beq \$4, \$0, around**

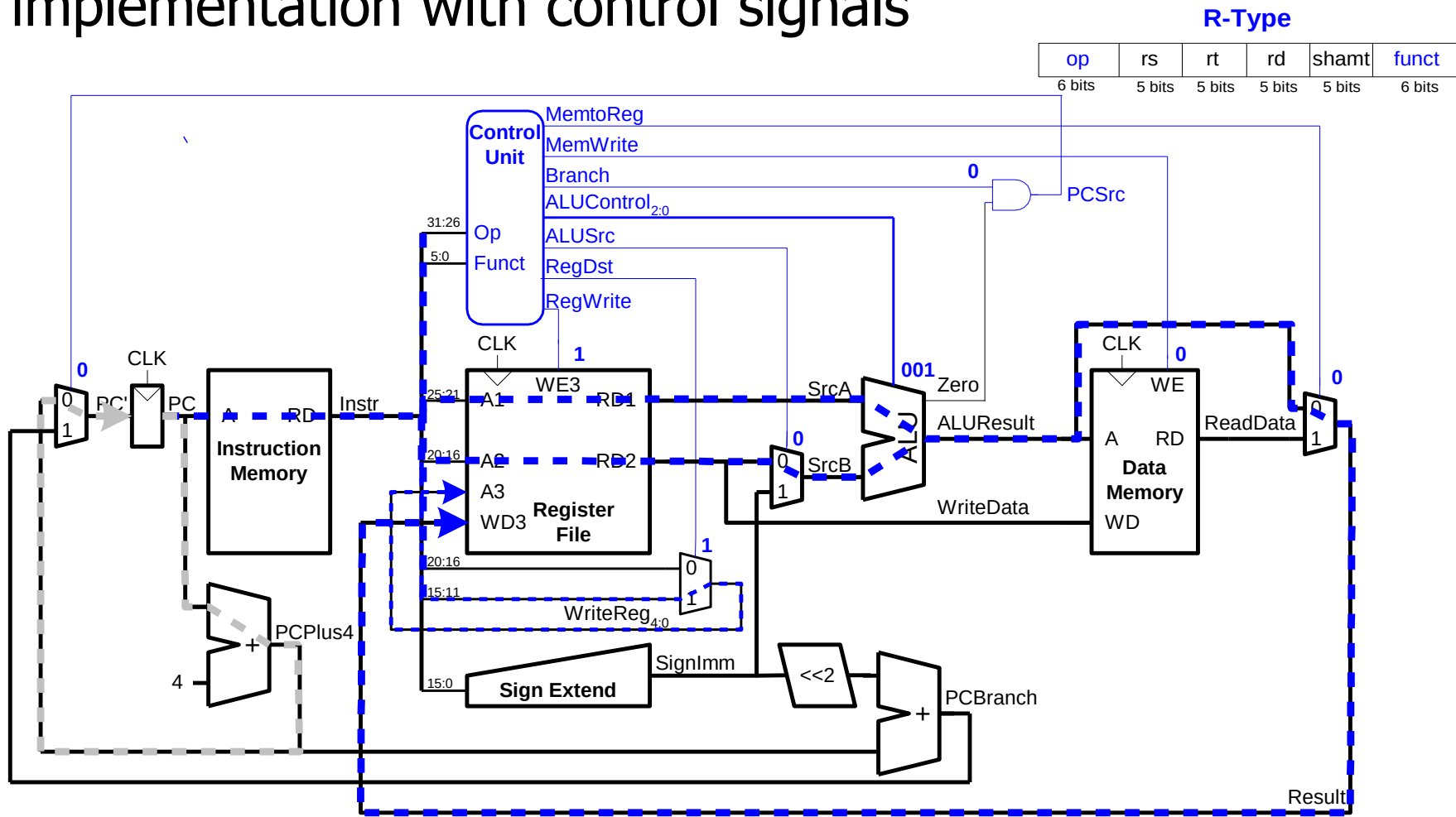
I-Type

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



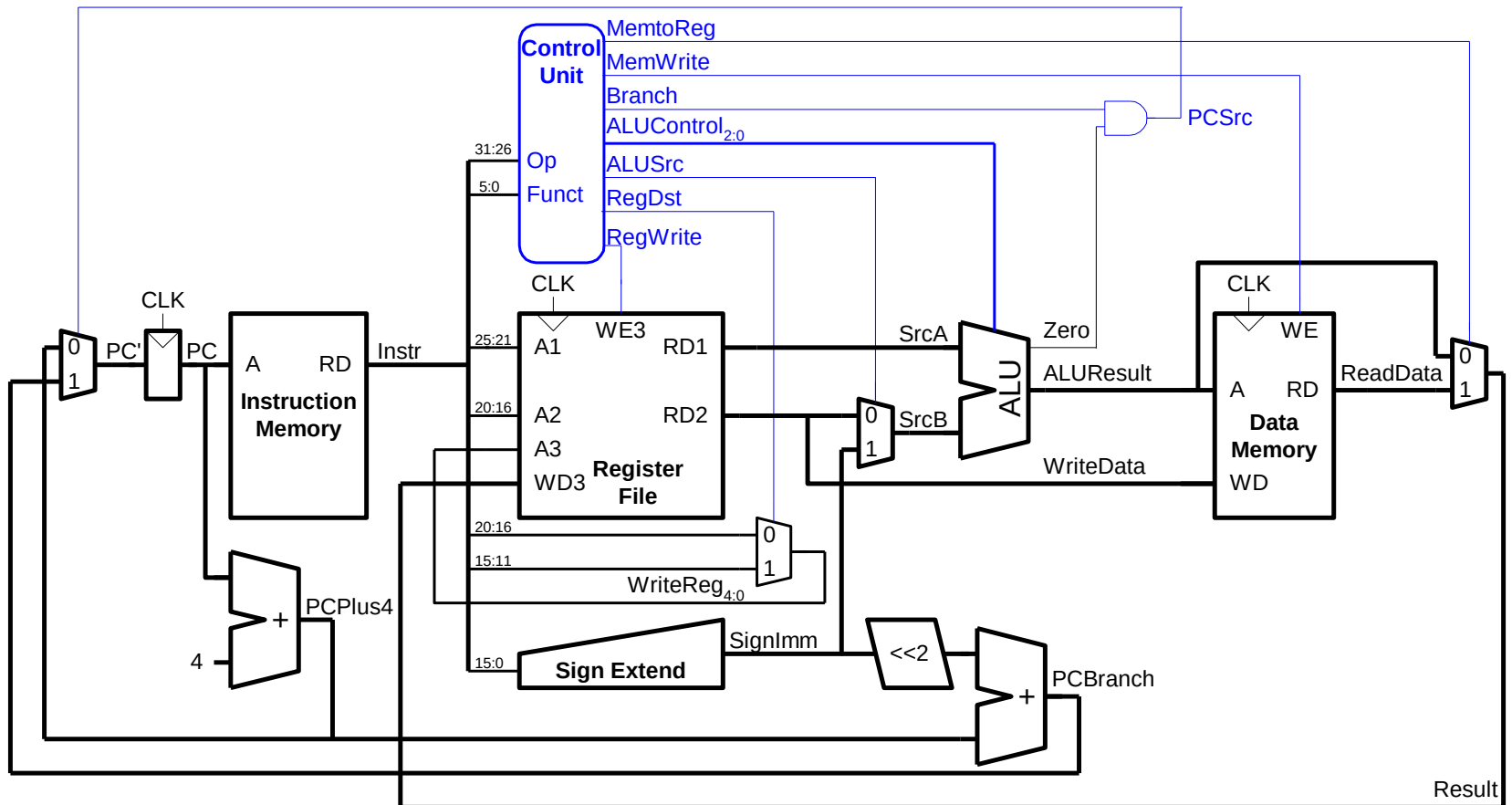
Single-Cycle MIPS - or

- Let's see how **or** instruction works out in the implementation with control signals

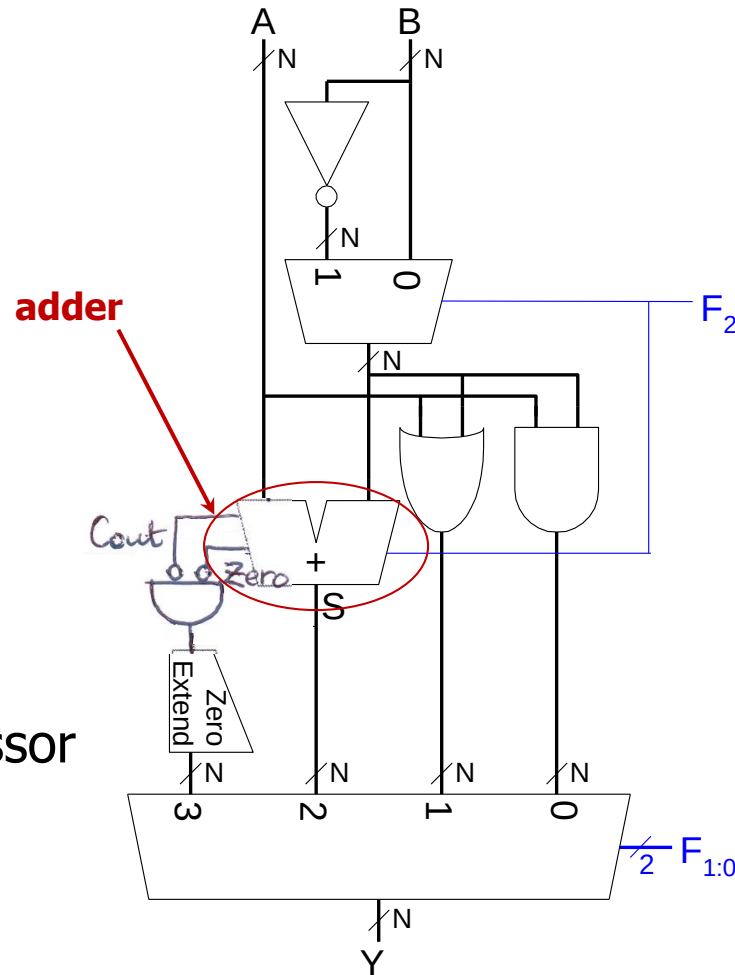
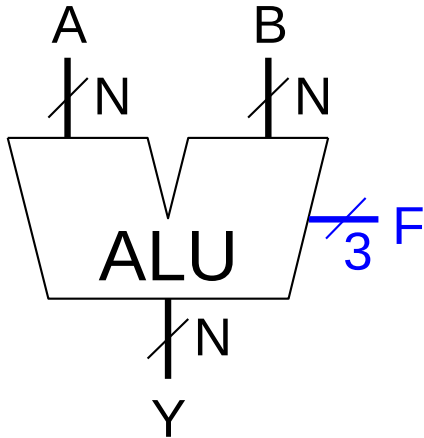


Single-Cycle MIPS

- As mentioned, CPU is designed with **datapath** and **control**
- Now, let's delve into the ALU and control part design



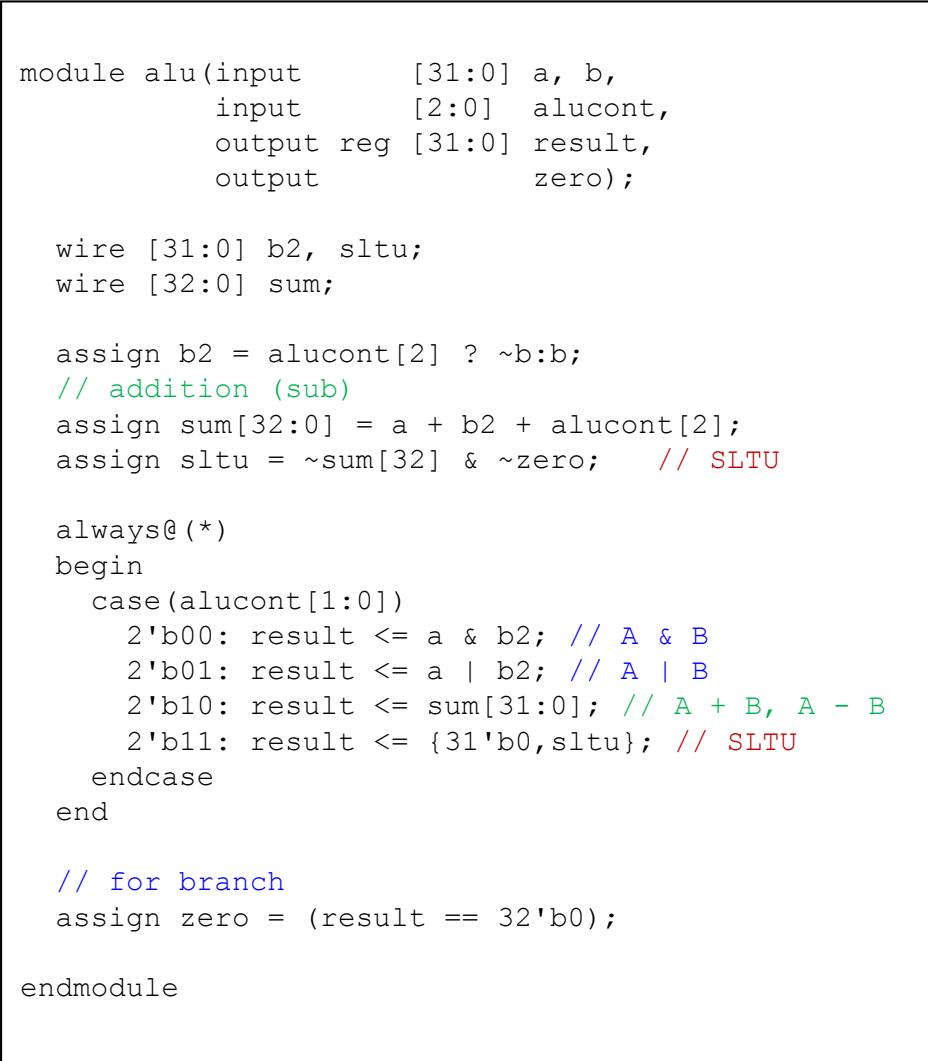
ALU (Arithmetic Logic Unit)



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	SLTU

```
// sltu (set less than
//      unsigned)
// $t0 = 1 if $t1 < $t2
sltu $t0, $t1, $t2
```

N = 32 in 32-bit processor



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	SLTU

Control Unit

R-Type

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

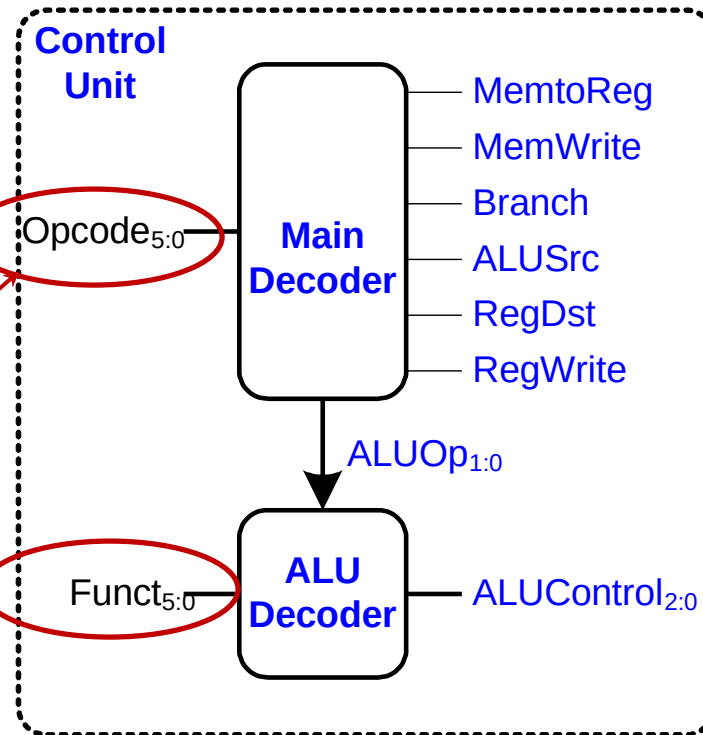
Opcode and **funct** fields come from the fetched instruction

I-Type

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

J-Type

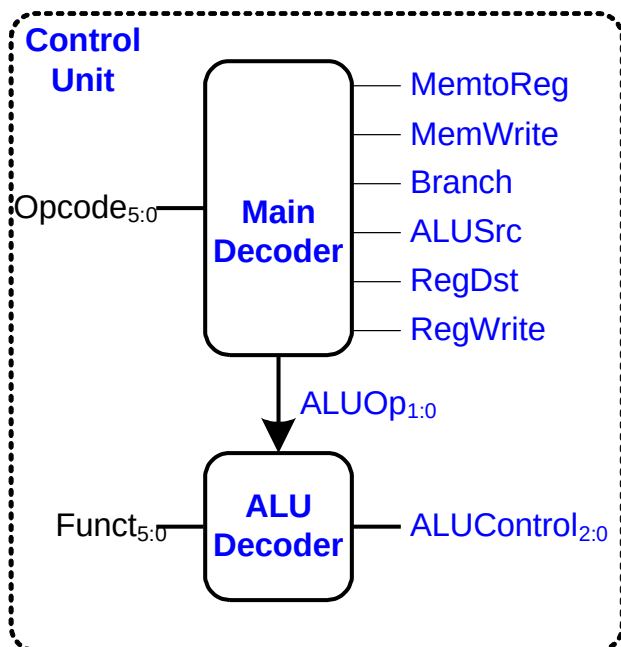
op	addr
6 bits	26 bits



Control Unit - ALU Control

- Implementation is completely dependent on hardware designers
- But, the designers should make sure the implementation is reasonable enough
- Memory access instructions (`lw`, `sw`) need to use ALU to calculate memory target address (**addition**)
- Branch instructions (`beq`, `bne`) need to use ALU for the equality check (**subtraction**)

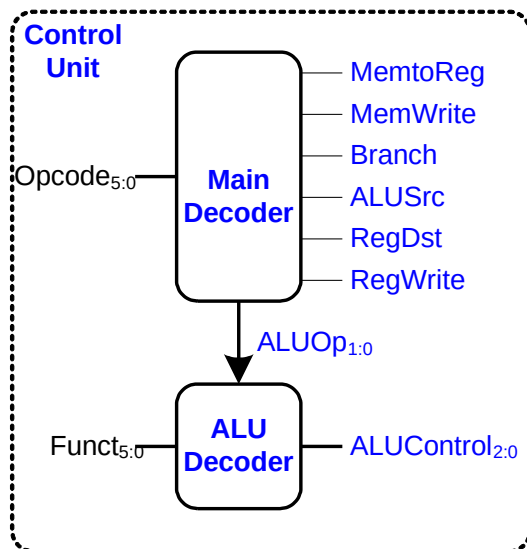
ALUOp _{1:0}	Meaning
00	Add
01	Subtract
10	Look at Funct
11	Not Used



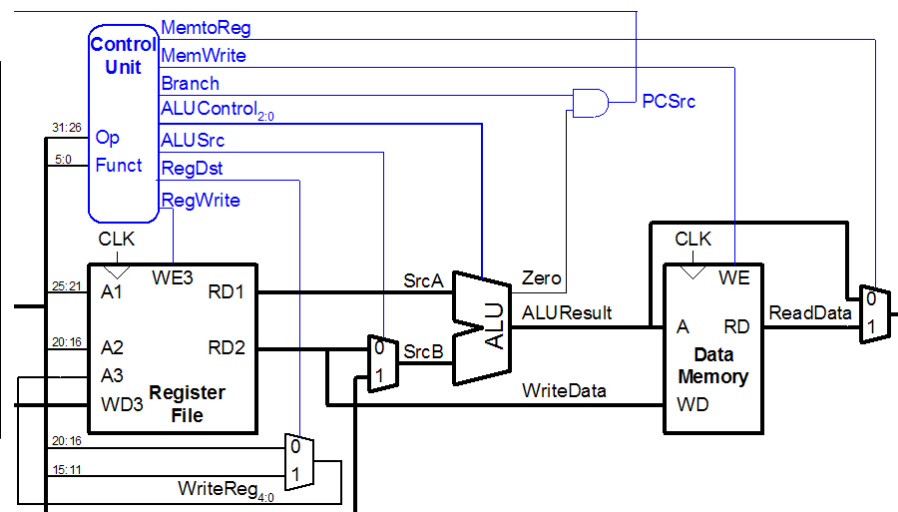
ALUOp _{1:0}	Funct	ALUControl _{2:0}
00	X	010 (add)
X1	X	110 (subtract)
1X	100000 (add)	010 (add)
1X	100010 (sub)	110 (subtract)
1X	100100 (and)	000 (and)
1X	100101 (or)	001 (or)
1X	101010 (slt)	111 (slt)

Control Unit - Main Decoder

Instruction	Op _{5:0}	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUOp _{1:0}
R-type	000000	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



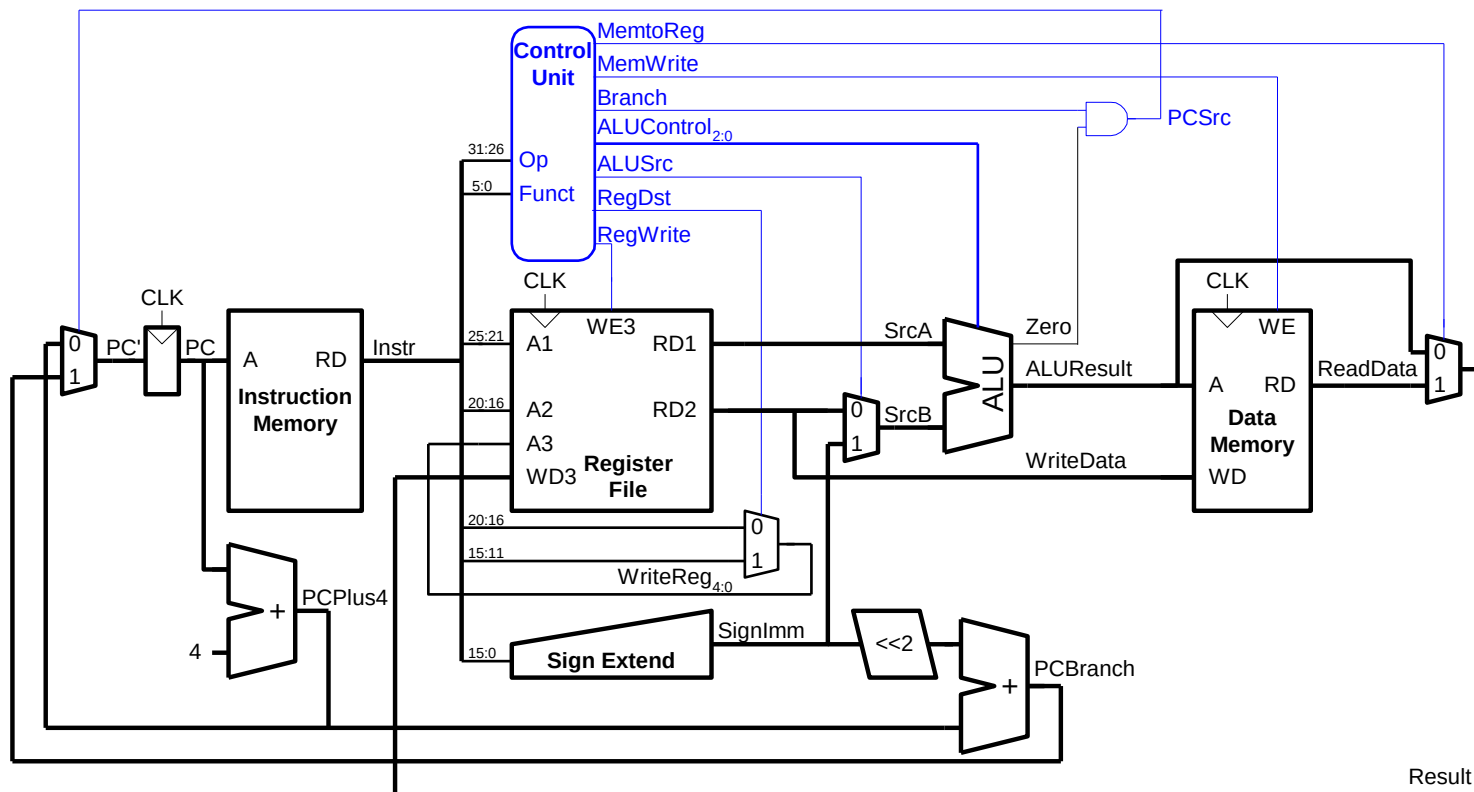
ALUOp _{1:0}	Meaning
00	Add
01	Subtract
10	Look at Funct field
11	Not Used



How about Other Instructions?

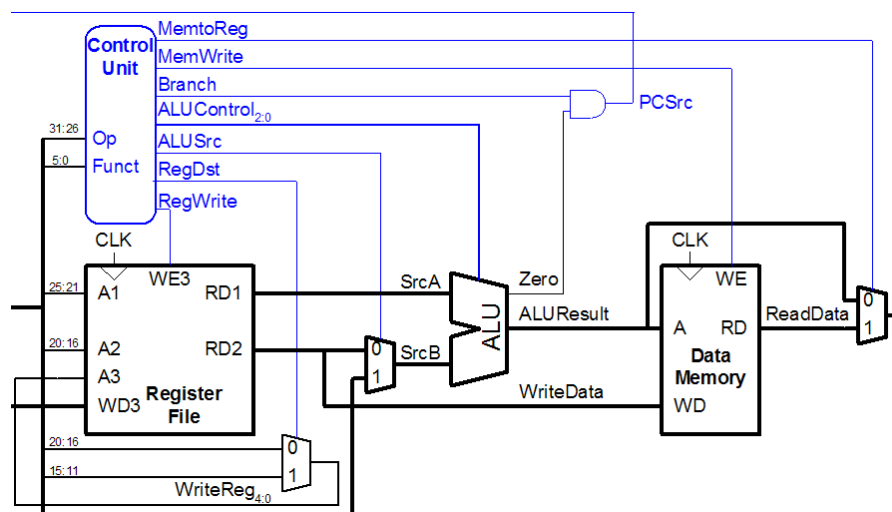
- Now, we are done with the control part design
- Let's examine if the design is able to execute other instructions

Example: **addi \$t0, \$t1, -14**



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
addi	001000	1	0	1	0	0	0	00



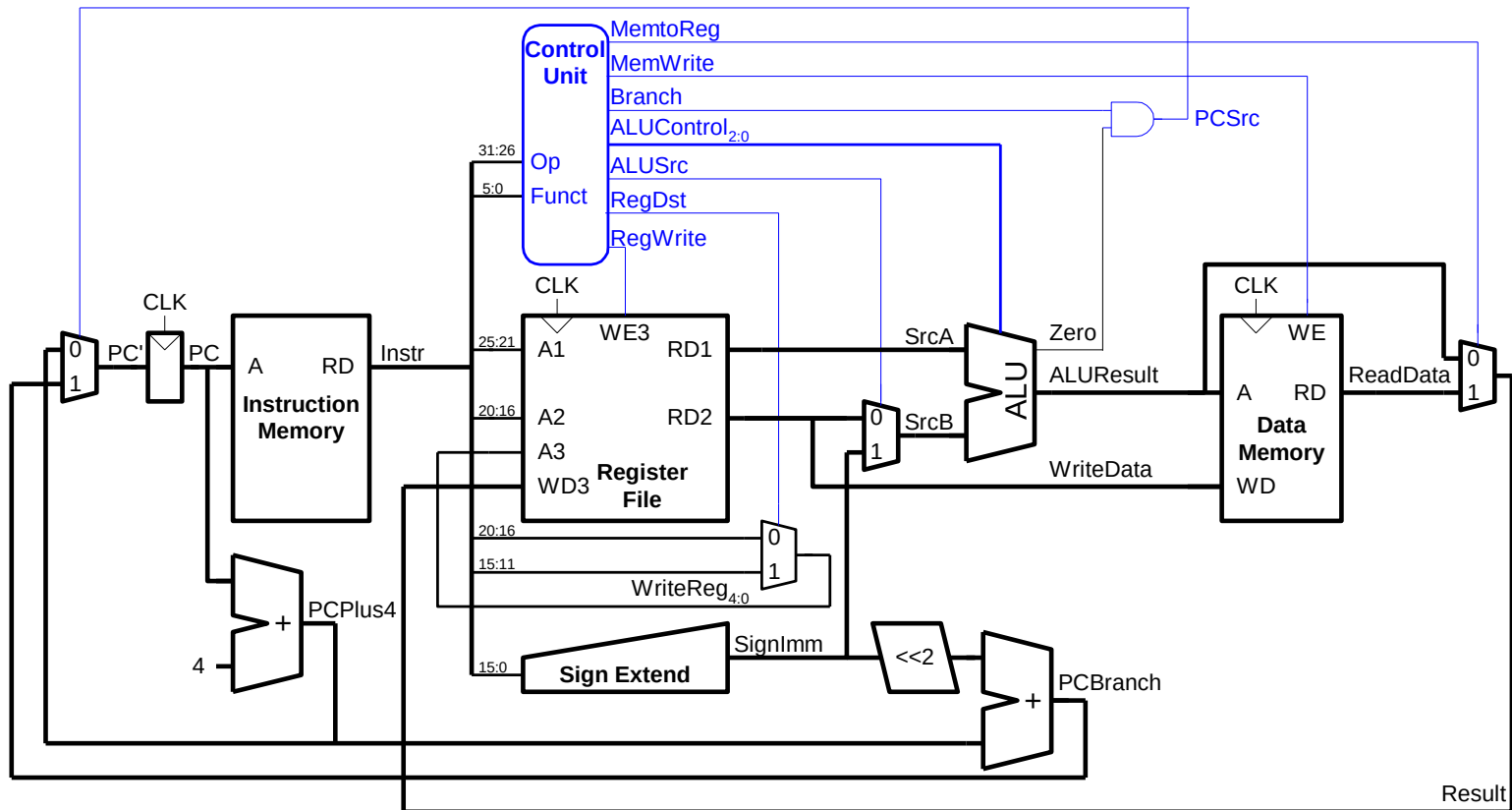
Control Unit - Main Decoder

- How about jump instructions?

- j

J-Type

op	addr
6 bits	26 bits

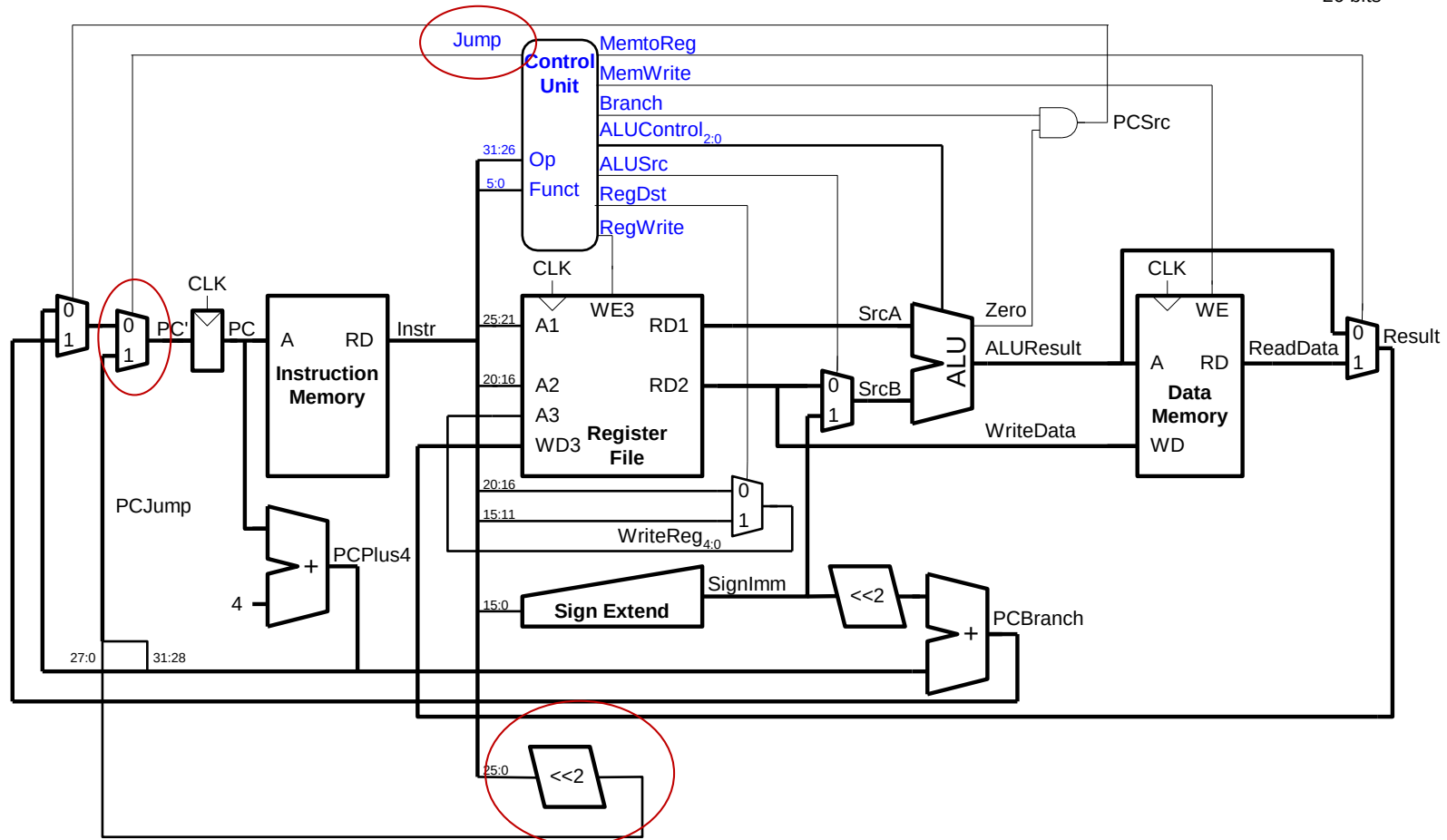


Control Unit - Main Decoder

- We added new hardware to support the j instruction
 - A logic to compute the target address
 - Mux and control signal

J-Type

op	addr
6 bits	26 bits



Control Unit - Main Decoder

- There should be one more output (**jump**) in the main decoder to support the jump instructions

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
addi	001000	1	0	1	0	0	0	00	0
j	000100	0	X	X	0	0	X	XX	1

Verilog Code - Main Decoder and ALU Control

```
module maindec(input [5:0] op,
               output memtoreg, memwrite,
               output branch, alusrc,
               output regdst, regwrite,
               output jump,
               output [1:0] aluop);
```

```
    reg [8:0] controls;
```

```
    assign {regwrite, regdst, alusrc,
            branch, memwrite,
            memtoreg, jump, aluop} = controls;
```

```
    always @(*)
    begin
```

```
        case(op)
```

```
            6'b000000: controls <= 9'b110000010; // R-type
```

```
            6'b100011: controls <= 9'b101001000; // lw
```

```
            6'b101011: controls <= 9'b001010000; // sw
```

```
            6'b000100: controls <= 9'b000100001; // beq
```

```
            6'b001000: controls <= 9'b101000000; // addi
```

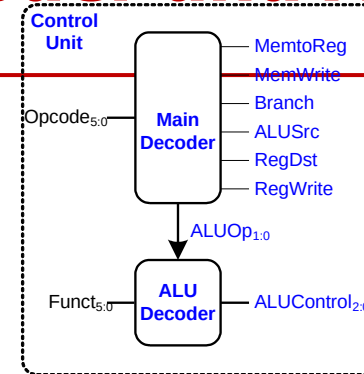
```
            6'b000010: controls <= 9'b0000000100; // j
```

```
            default: controls <= 9'bxxxxxxxx; // ???
```

```
        endcase
```

```
    end
```

```
endmodule
```



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

```
module aludec(input [5:0] funct,
               input [1:0] aluop,
               output reg [2:0] alucontrol);
```

```
    always @(*)
```

```
    begin
```

```
        case(aluop)
```

```
            2'b00: alucontrol <= 3'b010; // add
```

```
            2'b01: alucontrol <= 3'b110; // sub
```

```
            default: case(funct) // RTYPE
```

```
                6'b100000: alucontrol <= 3'b010; // ADD
```

```
                6'b100010: alucontrol <= 3'b110; // SUB
```

```
                6'b100100: alucontrol <= 3'b000; // AND
```

```
                6'b100101: alucontrol <= 3'b001; // OR
```

```
                6'b101010: alucontrol <= 3'b111; // SLT
```

```
                default: alucontrol <= 3'bxxx; // ???
```

```
            endcase
```

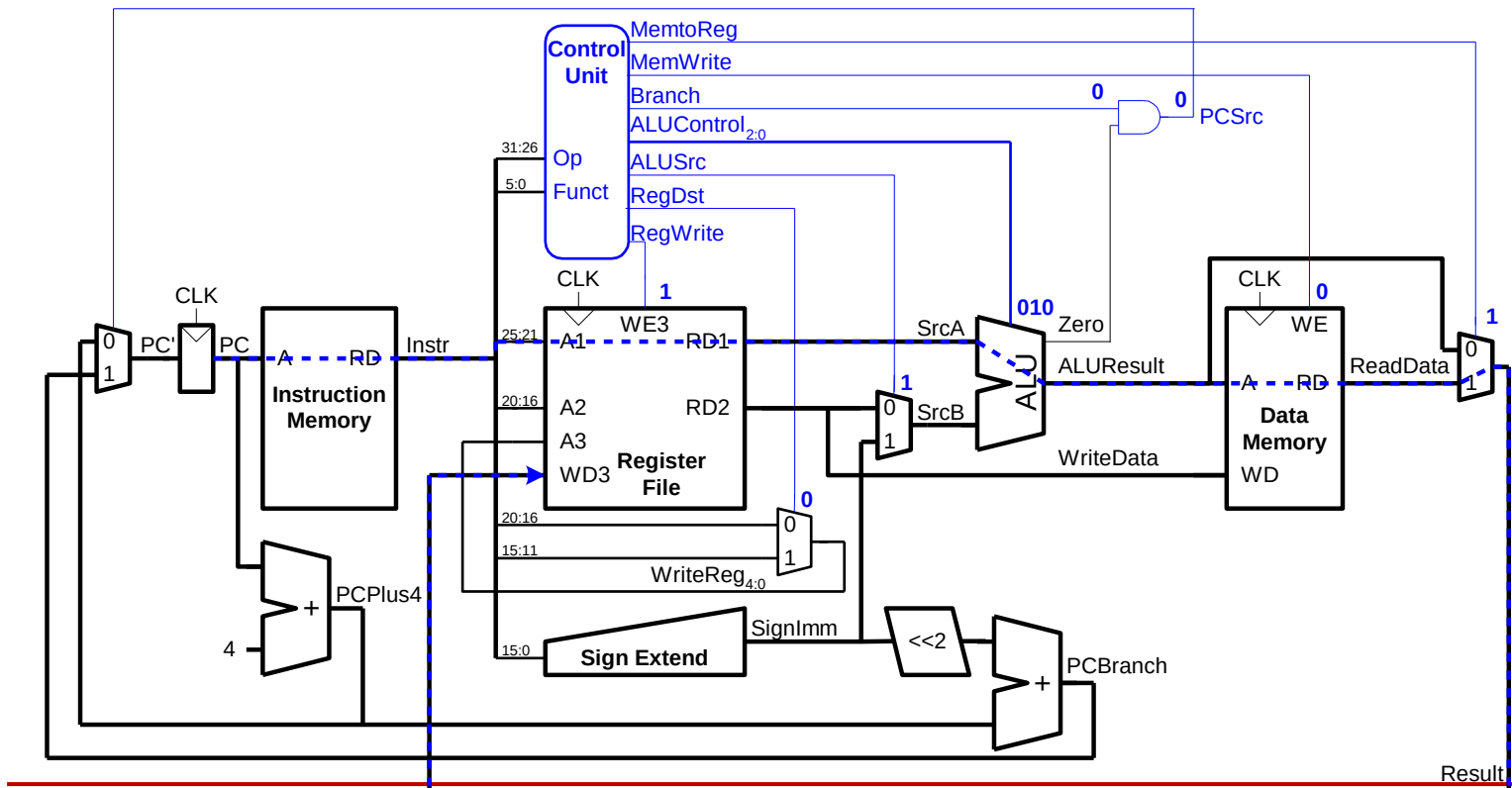
```
        endcase
```

```
    end
```

```
endmodule
```

Single-Cycle MIPS Performance

- How fast is the single-cycle processor?
- Clock cycle time (frequency) is limited by the **critical path**
 - The critical path is the path that takes the longest time
 - What do you think the critical path is?
 - The path that **lw** instruction goes through



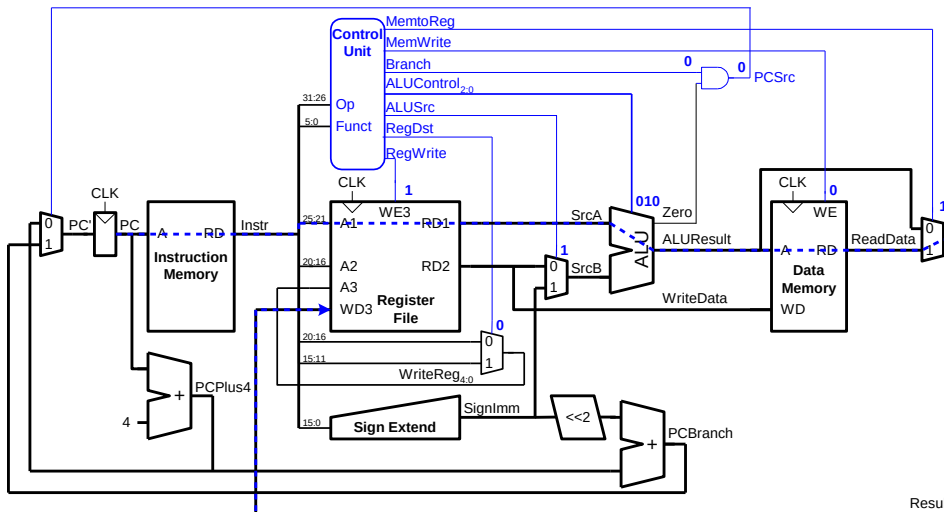
Single-Cycle MIPS Performance

- Critical path of single-cycle MIPS

$$T_c = t_{pcq_PC} + t_{mem} + \max(t_{RFread}, t_{sext}) + t_{mux} + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup}$$

- In most implementations, limiting paths are: memory (instruction and data), ALU, register file.

$$T_c = t_{pcq_PC} + 2t_{mem} + t_{RFread} + 2t_{mux} + t_{ALU} + t_{RFsetup}$$



Elements	Parameter
Register clock-to-Q	t_{pcq_PC}
Multiplexer	t_{mux}
ALU	t_{ALU}
Memory read	t_{mem}
Register file read	t_{RFread}
Register file setup	$t_{RFsetup}$

Example

Elements	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	30
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

$$\begin{aligned}T_c &= t_{pcq_PC} + 2t_{mem} + t_{RFread} + 2t_{mux} + t_{ALU} + t_{RFsetup} \\&= [30 + 2(250) + 150 + 2(25) + 200 + 20] \text{ ps} \\&= 950 \text{ ps}\end{aligned}$$



$$\begin{aligned}f_c &= 1/T_c \\f_c &= 1/950\text{ps} \\&= \mathbf{1.052\text{GHz}}\end{aligned}$$

- Assuming that the CPU should execute 100 billion instructions to run your program, what is the execution time of the program on a single-cycle MIPS processor?

$$\begin{aligned}\text{Execution Time} &= (\# \text{instructions}) \times (\text{cycles/instruction}) \times (\text{seconds/cycle}) \\&= (100 \times 10^9) \times (1) \times (950 \times 10^{-12} \text{ s}) \\&= 95 \text{ seconds}\end{aligned}$$