Microarchitecture

Digital Design and Computer Architecture

Mohammad Sadrosadati
Frank K. Gürkaynak

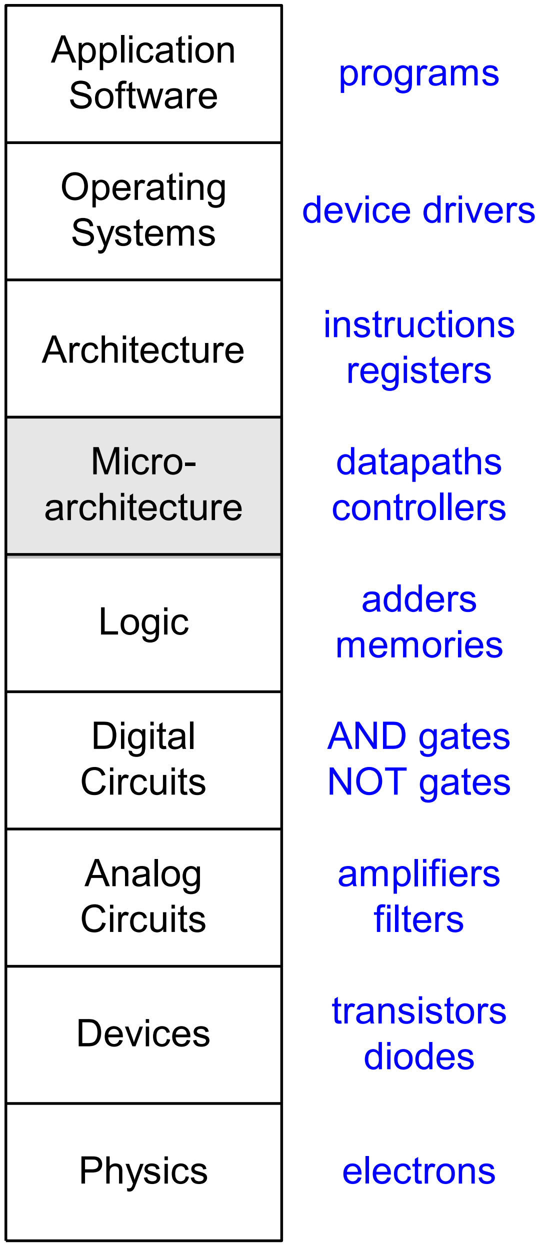
http://safari.ethz.ch/ddca

In This Lecture

- Overview of the MIPS Microprocessor
- RTL Design Revisited
- Single-Cycle Processor
- Performance Analysis

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*** **(this week)**
  - Each instruction executes in a single cycle
- ***Multicycle*** **(in the book, we will not cover it)**
  - Each instruction is broken up into a series of shorter steps
  - One instruction is executed piece-wise
  - Allows re-use of components (ALU, memory)
- ***Pipelined*** **(just after we finish this one)**
  - Each instruction is broken up into a series of steps
  - Multiple instructions execute at once
  - Better for performance

What does our Microprocessor do ?

- **Program is in memory, stored as a series of instructions.**
- **Basic execution is:**
  - Read one instruction
  - Decode what it wants
  - Find the operands either from registers or from memory
  - Perform the operation as dictated by the instruction
  - Write the result (if necessary)
  - Go to the next instruction

What are the Basic Components of MIPS?

- **The main pieces**
  - Memory to store program
  - Registers to access data fast
  - Data memory to store more data
  - An ALU to perform the core function

What are the Basic Components of MIPS?

- **The main pieces**
  - Memory to store program
  - Registers to access data fast
  - Data memory to store more data
  - An ALU to perform the core function
- **Some auxiliary pieces**
  - A program counter to tell where to find next instruction
  - Some logic to decode instructions
  - A way to manipulate the program counter for branches

What are the Basic Components of MIPS?

- **The main pieces**
  - Memory to store program
  - Registers to access data fast
  - Data memory to store more data
  - An ALU to perform the core function

- **Some auxiliary pieces**
  - A program counter to tell where to find next instruction
  - Some logic to decode instructions
  - A way to manipulate the program counter for branches

- ***Today we will see how it all comes together***

RTL Design Revisited

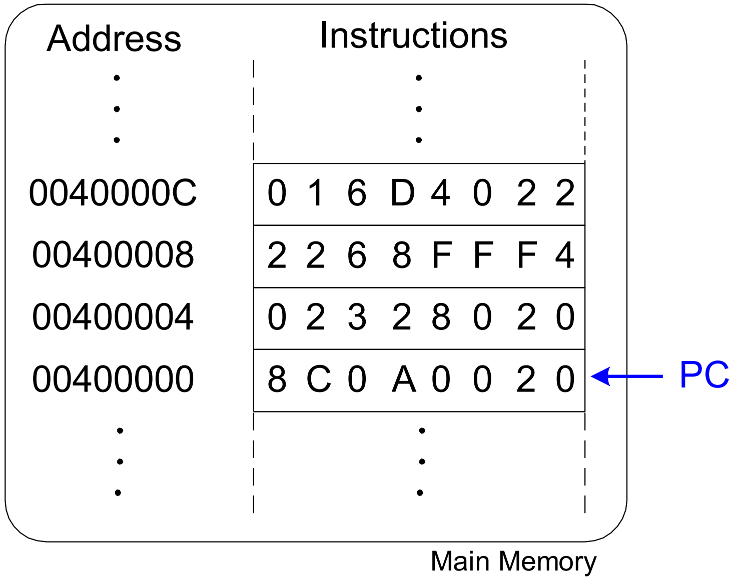
- **Registers store the current state**
- **Combinational logic determines next state**
  - Data moves from one register to another (datapath)
  - Control signals move data through the datapath
  - Control signals depend on state and input
- **Clock moves us from state to another**

Let us Start with Building the Processor

- Program is stored in a (read-only) memory as 32-bit binary values.
- Memory addresses are also 32-bit wide, allowing (in theory) 2^{32} = 4 Gigabytes of addressed memory.
- The actual address is stored in a register called PC.

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

Stored Program



What to do with the Program Counter?

- **The PC needs to be incremented by 4 during each cycle (for the time being).**
- **Initial PC value (after reset) is `0x00400000`**

```
reg [31:0] PC_p, PC_n;      // Present and next state of PC

// […]

  assign PC_n <= PC_p + 4;                   // Increment by 4;

  always @ (posedge clk, negedge rst)
    begin
       if (rst == ‘0’) PC_p <= 32’h00400000; // default
       else            PC_p <= PC_n;         // when clk
    end
```

We have an Instruction Now What ?

- **There are different types of instructions**
  - ***R type*** (three registers)
  - ***I type*** (the one with immediate values)
  - ***J type*** (for changing the flow)
- **First only a subset of MIPS instructions:**
  - R type instructions: **`and`, `or`, `add`, `sub`, `slt`**
  - Memory instructions: **`lw`, `sw`**
  - Branch instructions: **`beq`**
- **Later we will add `addi` and `j`**

R-Type MIPS instructions

- **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 together, the opcode and function tell the computer what operation to perform
  - ***shamt***: the shift amount for shift instructions, otherwise it's 0

R-Type

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

R-Type Examples

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	

Note the order of registers in the assembly code:

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

We Need a Register File

- **Store 32 registers, each 32-bit**
  - 2^5 == 32, we need 5 bits to address each
- **Every R-type instruction uses 3 register**
  - Two for reading (RS, RT)
  - One for writing (RD)
- **We need a special memory with:**
  - 2 read ports (address x2, data out x2)
  - 1 write port (address, data in)

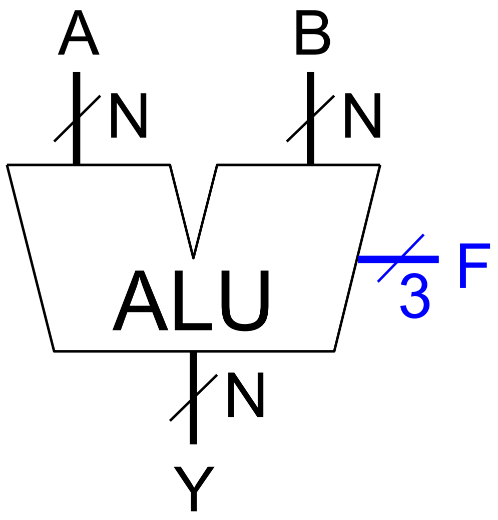
Register File

```
input [4:0]   a_rs, a_rt, a_rd;
input [31:0]  di_rd;
input         we_rd;
output [31:0] do_rs, do_rt;

  reg [31:0] R_arr [31:0]; // Array that stores regs

  // Circuit description
  assign do_rs = R_arr[a_rs];           // Read RS


  assign do_rt = R_arr[a_rt];           // Read RT


  always @ (posedge clk)
      if (we_rd) R_arr[a_rd] <= di_rd; // write RD
```

Register File

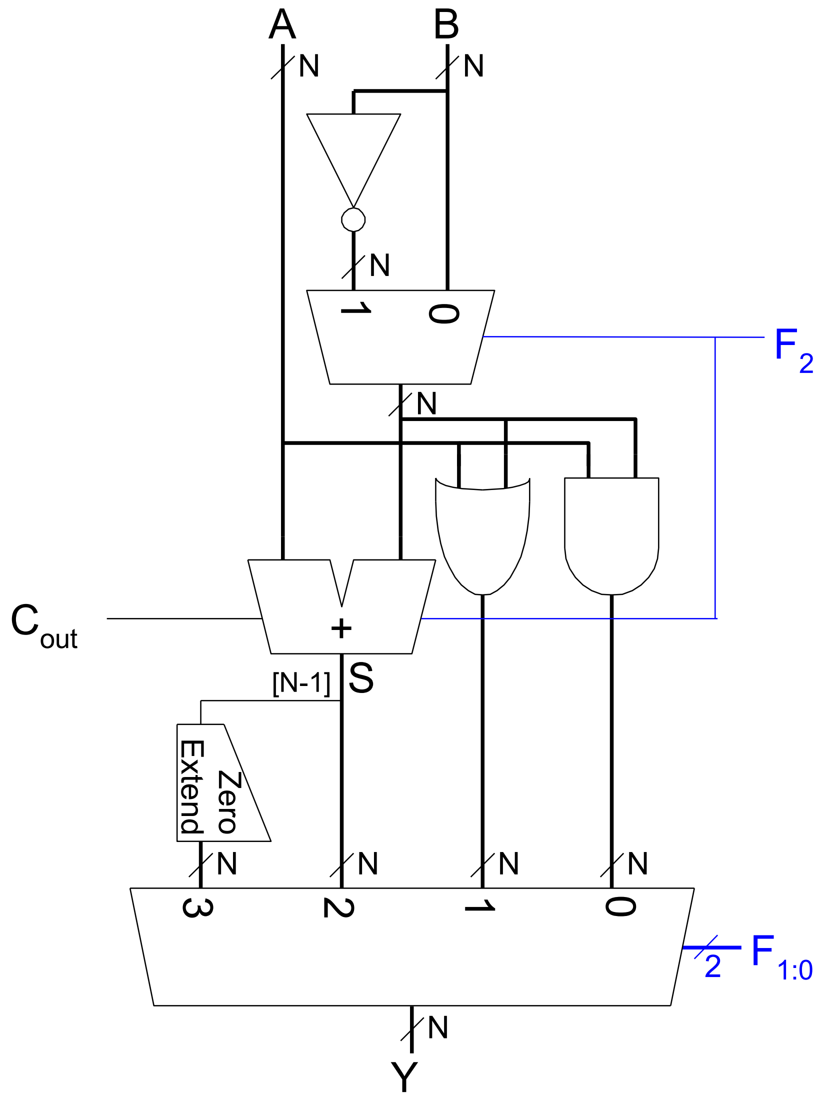
```
input [4:0]   a_rs, a_rt, a_rd;
input [31:0]  di_rd;
input         we_rd;
output [31:0] do_rs, do_rt;

  reg [31:0] R_arr [31:0]; // Array that stores regs

  // Circuit description; add the trick with $0
  assign do_rs = (a_rs != 5’b00000)?    // is address 0?
                  R_arr[a_rs] : 0;      // Read RS or 0

  assign do_rt = (a_rt != 5’b00000)?    // is address 0?
                  R_arr[a_rt] : 0;      // Read RT or 0

  always @ (posedge clk)
      if (we_rd) R_arr[a_rd] <= di_rd; // write RD
```

ALU Does the Real Work in a 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	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

So far so good

- **We have most of the components for MIPS**
  - We still do not have data memory

So far so good

- **We have most of the components for MIPS**
  - We still do not have data memory
- **We can implement all R-type instructions**
  - Assuming our ALU can handle all of the functions
  - At the moment we do not have the shifter
  - We know what to do for more functions.

So far so good

- **We have most of the components for MIPS**
  - We still do not have data memory
- **We can implement all R-type instructions**
  - Assuming our ALU can handle all of the functions
  - At the moment we do not have the shifter
  - We know what to do for more functions.
- **…but not much else**
  - ***No memory access***
  - ***No immediate values***
  - ***No conditional branches, no absolute jumps***

Data Memory

- **Will be used to store the bulk of data**

```
input [15:0]  addr; // Only 16 bits in this example
input [31:0]  di;
input         we;
output [31:0] do;

  reg [65535:0] M_arr [31:0];            // Array for Memory

  // Circuit description
  assign do = M_arr[addr];               // Read memory

  always @ (posedge clk)
      if (we) M_arr[addr] <= di;         // write memory
```

Let us Read (then Write) to Memory

- **The main memory can have up to 2^{32} bytes**
  - It needs an 32-bit address for the entire memory.
  - In practice much smaller memories are used
- **We need the `lw` instruction to read from memory**
  - I-type instruction
  - We need to calculate the address from an immediate value and a register.
  - Maybe we can use the ALU to add immediate to the register?

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 the opcode

I-Type

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

Differences between R-type and I-type

- **Changes to Register file**
  - ***RS + sign extended immediate*** is the address
  - RS is always read.
  - For some instructions RT is read as well
  - If one register is written it is RT, not RD
  - The write_data_in can come from Memory or ALU
  - Not all I-type instructions write to register file

- **Changes to ALU**
  - Used to calculate memory address, add function needed during non R-type operations
  - Result is also used as address for the data memory
  - One input is no longer RT, but the immediate value

Now Writing to Memory is Similar

- **We need the `sw` instruction to write to memory**
  - I-type instruction
  - Address calculated the same way
  - Unlike, previous step, there is no writing back to register.
  - But we need to enable the write enable of the memory

Our MIPS Datapath has Several Options

- **ALU inputs**
  - Either RT or Immediate
- **Write Address of Register File**
  - Either RD or RT
- **Write Data In of Register File**
  - Either ALU out or Data Memory Out
- **Write enable of Register File**
  - Not always a register write
- **Write enable of Memory**
  - Only when writing to memory (sw)

Our MIPS Datapath has Several Options

- **ALU inputs**
  - Either RT or Immediate ***(MUX)***
- **Write Address of Register File**
  - Either RD or RT ***(MUX)***
- **Write Data In of Register File**
  - Either ALU out or Data Memory Out ***(MUX)***
- **Write enable of Register File**
  - Not always a register write ***(MUX)***
- **Write enable of Memory**
  - Only when writing to memory (sw) ***(MUX)***

All these options are our control signals

Let us Develop our Control Table

Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	MemWrite	MemtoReg	ALUOp

▪ ***RegWrite:***	Write enable for the register file
▪ ***RegDst:***	Write to register RD or RT
▪ ***AluSrc:***	ALU input RT or immediate
▪ ***MemWrite:***	Write Enable
▪ ***MemtoReg:***	Register data in from Memory or ALU
▪ ***ALUOp:***	What operation does ALU do

Let us Develop our Control Table

Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	MemWrite	MemtoReg	ALUOp
R-type	000000	1	1	0	0	0	funct

- ***RegWrite:*** Write enable for the register file
- ***RegDst:*** Write to register RD or RT
- ***AluSrc:*** ALU input RT or immediate
- ***MemWrite:*** Write Enable
- ***MemtoReg:*** Register data in from Memory or ALU
- ***ALUOp:*** What operation does ALU do

Let us Develop our Control Table

Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	MemWrite	MemtoReg	ALUOp
R-type	000000	1	1	0	0	0	funct
`lw`	100011	1	0	1	0	1	add

- ***RegWrite:*** Write enable for the register file
- ***RegDst:*** Write to register RD or RT
- ***AluSrc:*** ALU input RT or immediate
- ***MemWrite:*** Write Enable
- ***MemtoReg:*** Register data in from Memory or ALU
- ***ALUOp:*** What operation does ALU do

Let us Develop our Control Table

Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	MemWrite	MemtoReg	ALUOp
R-type	000000	1	1	0	0	0	funct
`lw`	100011	1	0	1	0	1	add
`sw`	101011	0	X	1	1	X	add

- ***RegWrite:*** Write enable for the register file
- ***RegDst:*** Write to register RD or RT
- ***AluSrc:*** ALU input RT or immediate
- ***MemWrite:*** Write Enable
- ***MemtoReg:*** Register data in from Memory or ALU
- ***ALUOp:*** What operation does ALU do

Now we are much better

- **We have almost all components for MIPS**
- **We already had (nearly) all R-type instructions**
  - Our ALU can implement some functions
  - We know what to do for more functions.
- **Now we can also Read and Write to memory**
- **Conditional Branches are still missing**
  - Until know, PC just increases by 4 every time.
  - We need to somehow be able to influence this.

Branch if equal: BEQ

- **From Appendix B:**

  Opcode: `000100`

  `If ([rs]==[rt]) PC= BTA`

  BTA = Branch Target Address
  = PC+4 +(SignImm <<2)

Branch if equal: BEQ

- **From Appendix B:**

  Opcode: `000100`

  `If ([rs]==[rt]) PC= BTA`

  BTA = Branch Target Address
  = PC+4 +(SignImm <<2)

- **We need ALU for comparison**
  - Subtract RS – RT
  - We need a ‘zero’ flag, if RS equals RT

Branch if equal: BEQ

- **From Appendix B:**

  Opcode: `000100`

  `If ([rs]==[rt]) PC= BTA`

  BTA = Branch Target Address
  = PC+4 +(SignImm <<2)

- **We need ALU for comparison**
  - Subtract RS – RT
  - We need a ‘zero’ flag, if RS equals RT

- **We also need a *Second Adder***
  - The immediate value has to be added to PC+4

Branch if equal: BEQ

- **From Appendix B:**

  Opcode: `000100`

  `If ([rs]==[rt]) PC= BTA`

  BTA = Branch Target Address
  = PC+4 +(SignImm <<2)

- **We need ALU for comparison**
  - Subtract RS – RT
  - We need a ‘zero’ flag, if RS equals RT
- **We also need a *Second Adder***
  - The immediate value has to be added to PC+4
- **PC has now two options**
  - Either PC+4 or the new BTA

More Control Signals

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

- **New Control Signal**
  - ***Branch*:** Are we jumping or not ?

We are almost done

- **We have almost all components for MIPS**
- **We already had (nearly) all R-type instructions**
  - Our ALU can implement some functions
  - We know what to do for more functions.
- **Now we can also Read and Write to memory**
- **We also have conditional branches**
- **Absolute jumps are still missing**
  - J-type instructions
  - The next PC address is directly determined

Machine Language: J-Type

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

J-Type

op	addr
6 bits	26 bits

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

26-bit addr $0000\ \underbrace{00}_{0}\underbrace{00\ 01}_{1}\underbrace{00\ 00}_{0}\underbrace{00\ 00}_{0}\underbrace{00\ 00}_{0}\underbrace{00\ 10}_{2}\underbrace{10\ 00}_{8}00$ (0x0100028)

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	

Jump: J

- **From Appendix B:**

  Opcode: `000010`

  `PC= JTA`

  `JTA` = Jump Target Address
  = `{(PC+4)[31:28], addr,2’b0}`

- **No need for ALU**
  - Assemble the JTA from PC and 26-bit immediate

- **No need for memory, register file**

- **One more option (mux + control signal) for PC**

Even More Control Signals

Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUOp	Jump
R-type	000000	1	1	0	0	0	0	funct	0
lw	100011	1	0	1	0	0	1	add	0
sw	101011	0	X	1	0	1	X	add	0
beq	000100	0	X	0	1	0	X	sub	0
j	000010	0	X	X	X	0	X	X	1

- **New Control Signal**
  - ***Jump***: Direct jump yes or no

Now we are Almost Done

- **We have all major parts**
  - We can implement, R, I, J type instructions
  - We can calculate, branch, and jump
  - We would be able to run most programs

Now we are Almost Done

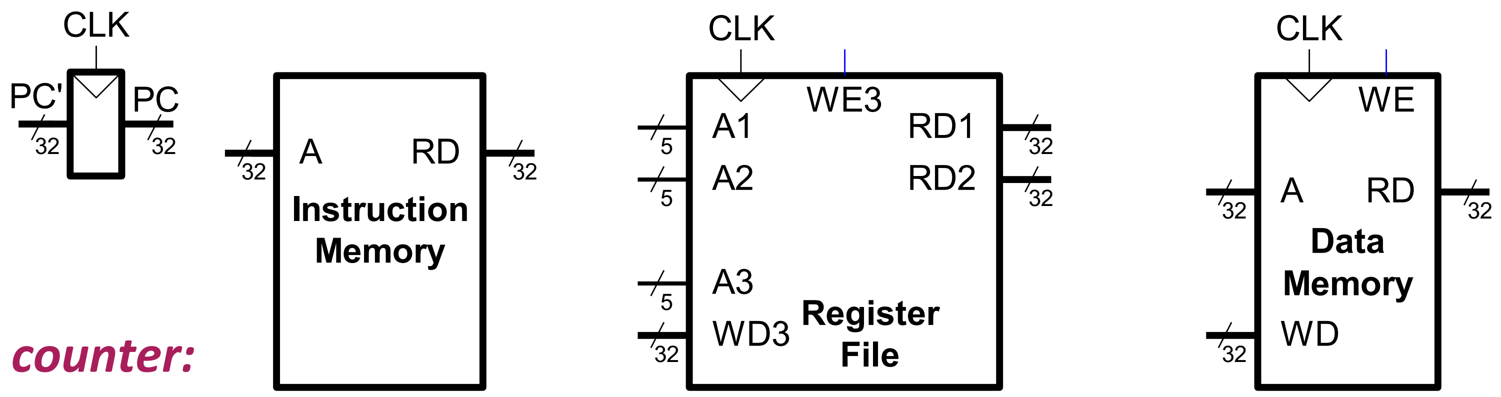
- **We have all major parts**
  - We can implement, R, I, J type instructions
  - We can calculate, branch, and jump
  - We would be able to run most programs

- **Still we can improve:**
  - ALU can be improved for more ‘funct’
  - We do not have a shifter.
  - Or a multiplier.
  - More I-type functions (only `lw` and `sw` supported)
  - More J-type functions (`jal`, `jr`)

Now the original slides once again

- The following slides will repeat what we just did
- Consistent with the text book

MIPS State Elements



- ***Program counter:***

  32-bit register

- ***Instruction memory:***

  Takes input 32-bit address A and reads the 32-bit data (i.e., instruction) from that address to the read data output RD.

- ***Register file:***

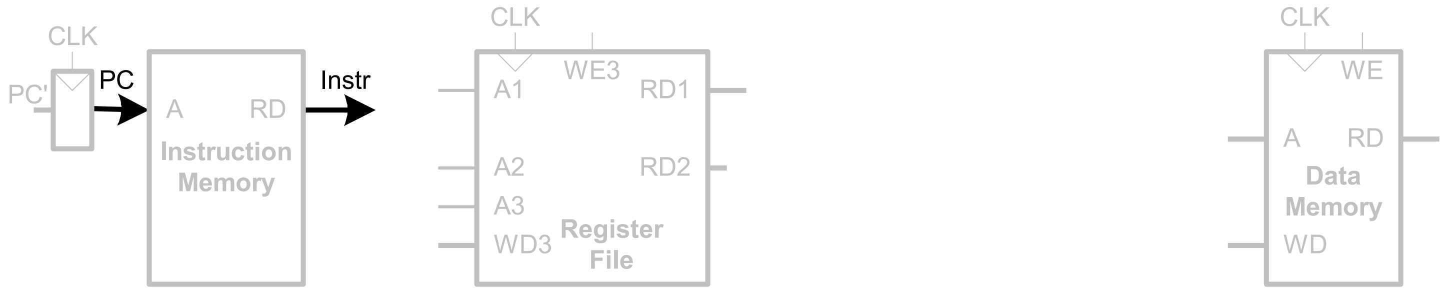
  The 32-element, 32-bit register file has 2 read ports and 1 write port

- ***Data memory:***

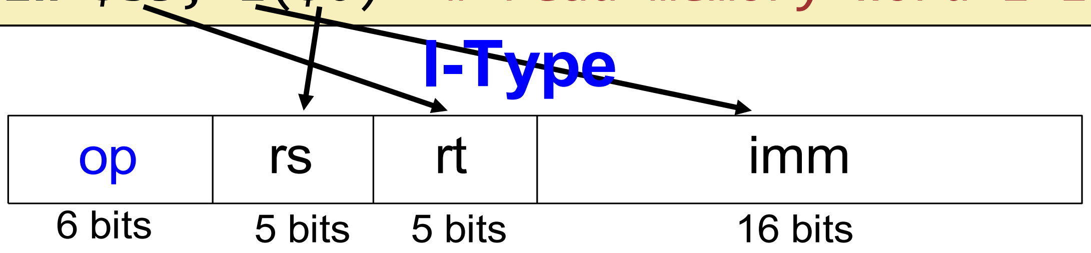
  Has a single read/write port. If the write enable, WE, is 1, it writes data WD into address A on the rising edge of the clock. If the write enable is 0, it reads address A onto RD.

Single-Cycle Datapath: `lw` fetch

- *STEP 1:* **Fetch instruction**

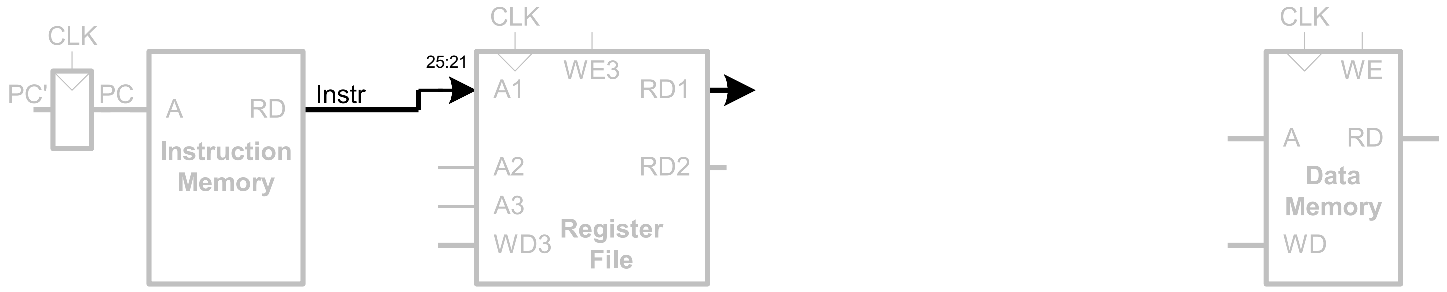


```
lw $s3, 1($0)  # read memory word 1 into $s3
```



Single-Cycle Datapath: `lw` register read

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



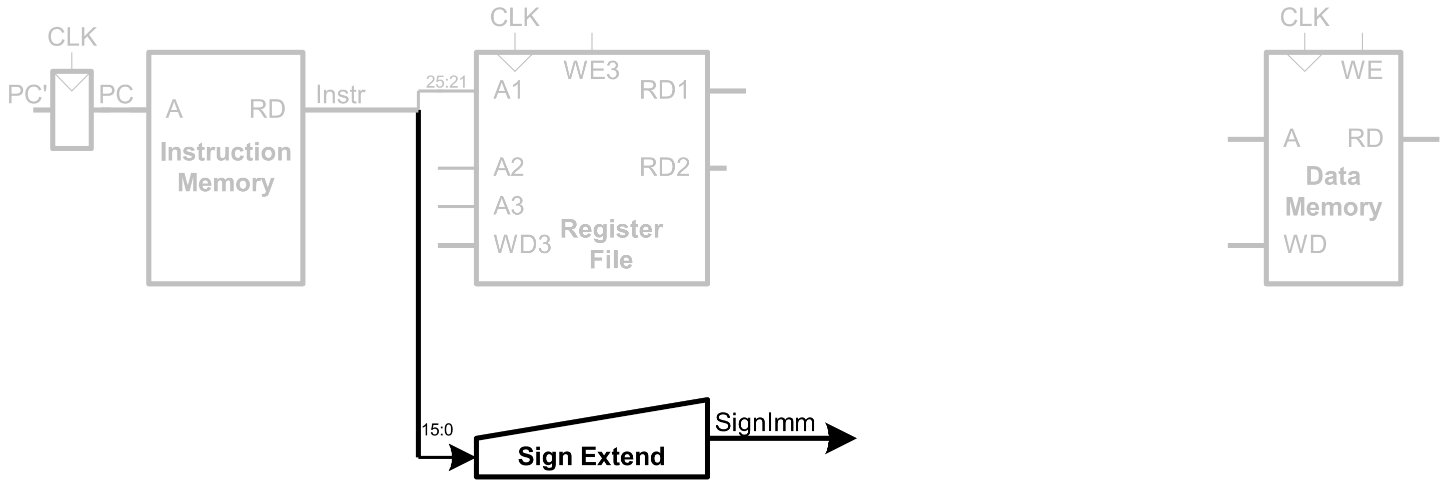
```
lw $s3, 1($0)  # read memory word 1 into $s3
```

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**



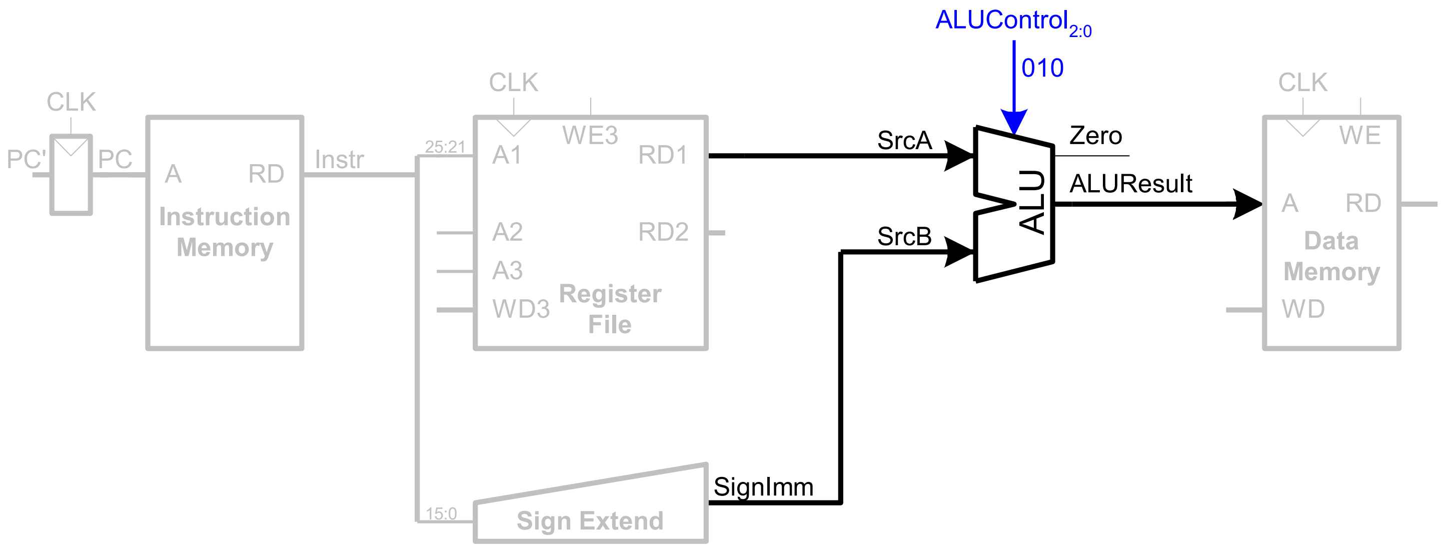
```
lw $s3, 1($0)   # read memory word 1 into $s3
```

I-Type

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

Single-Cycle Datapath: `lw` address

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



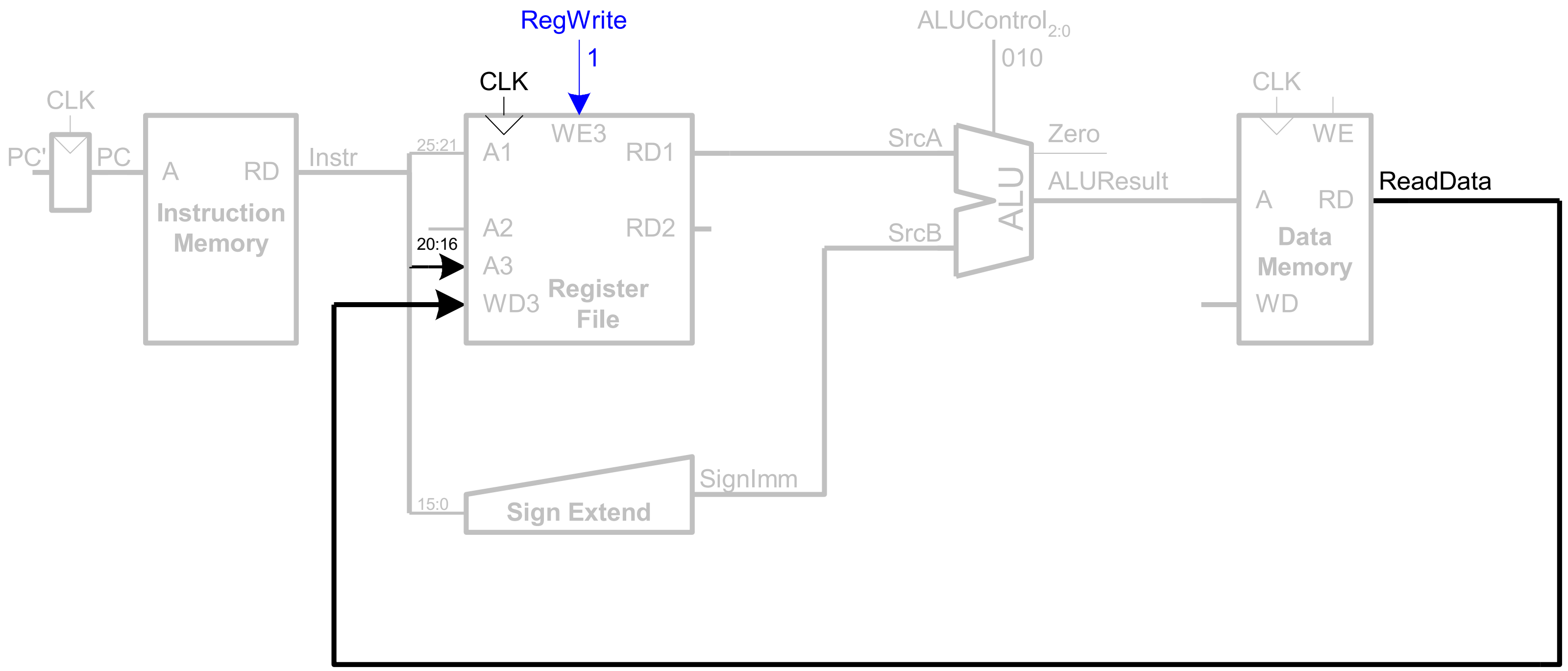
```
lw $s3, 1($0)   # read memory word 1 into $s3
```

I-Type

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

Single-Cycle Datapath: `lw` memory read

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



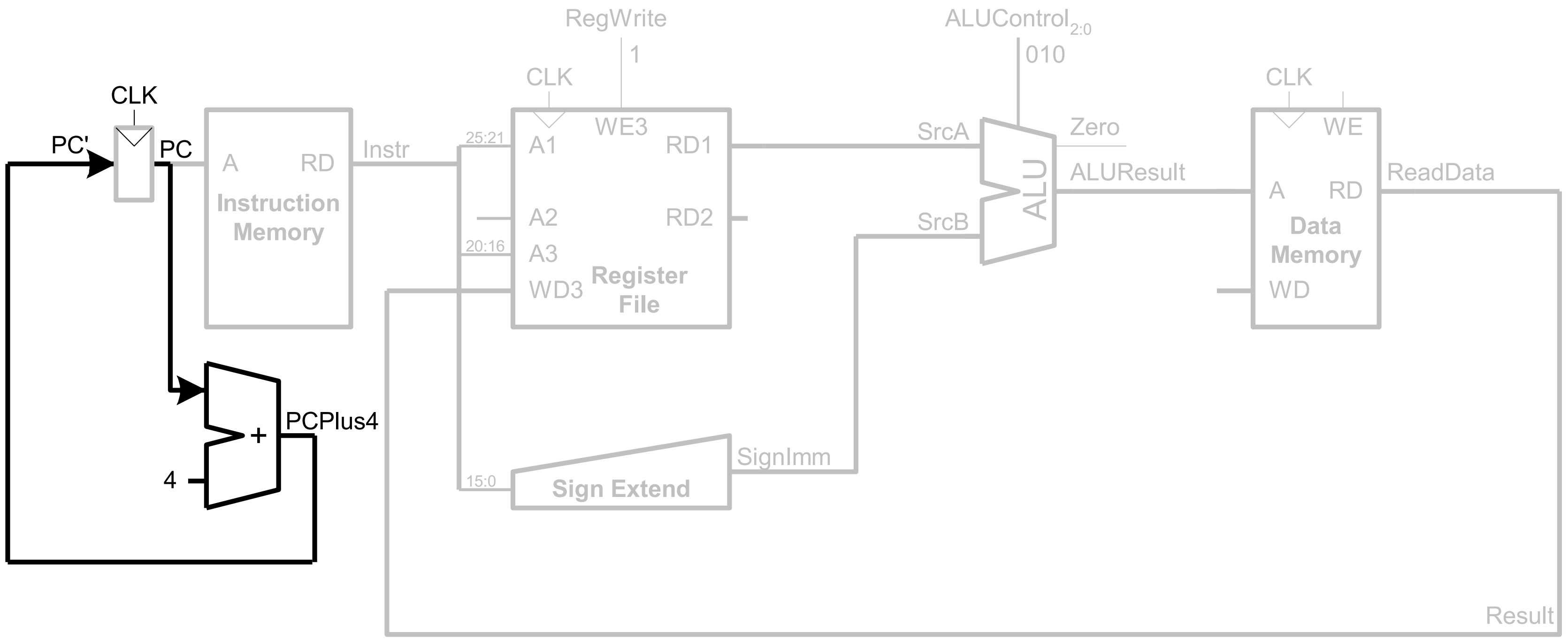
```
lw $s3, 1($0)   # read memory word 1 into $s3
```

I-Type

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**



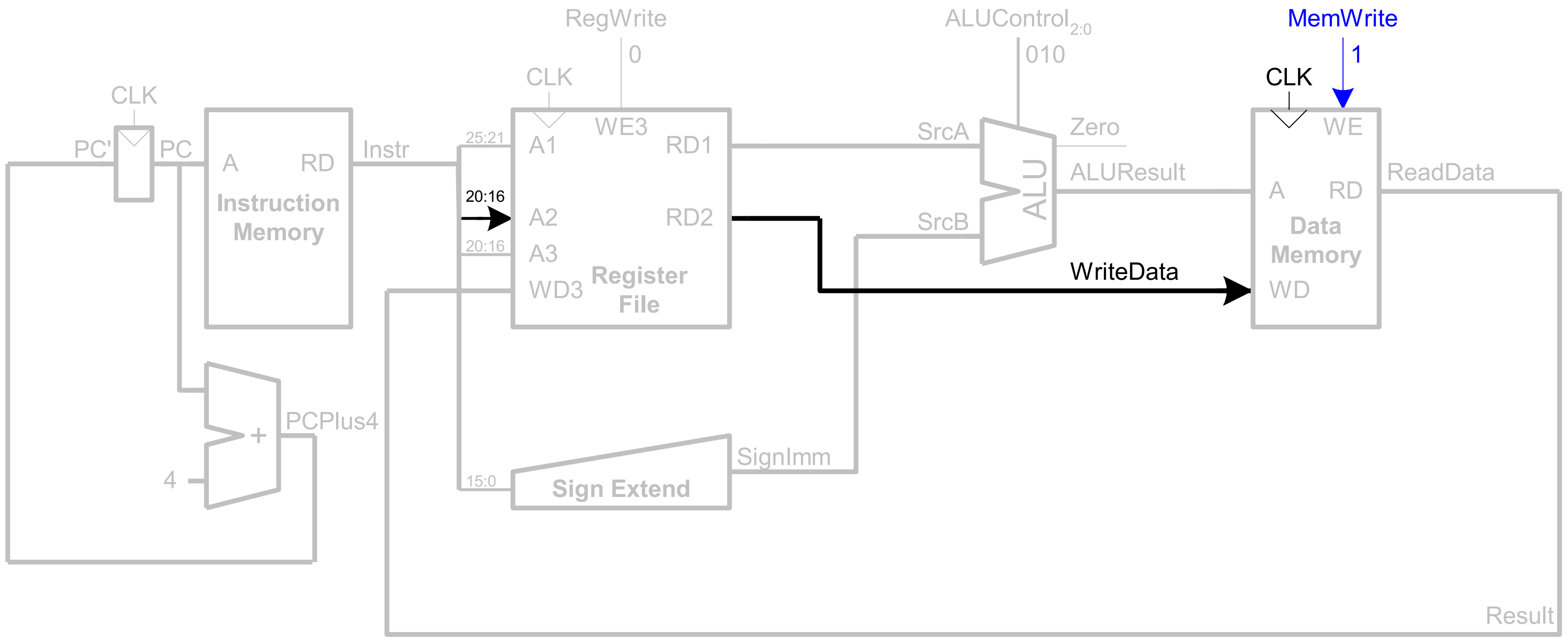
```
lw $s3, 1($0)   # read memory word 1 into $s3
```

I-Type

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

Single-Cycle Datapath: sw

- **Write data in rt to memory**



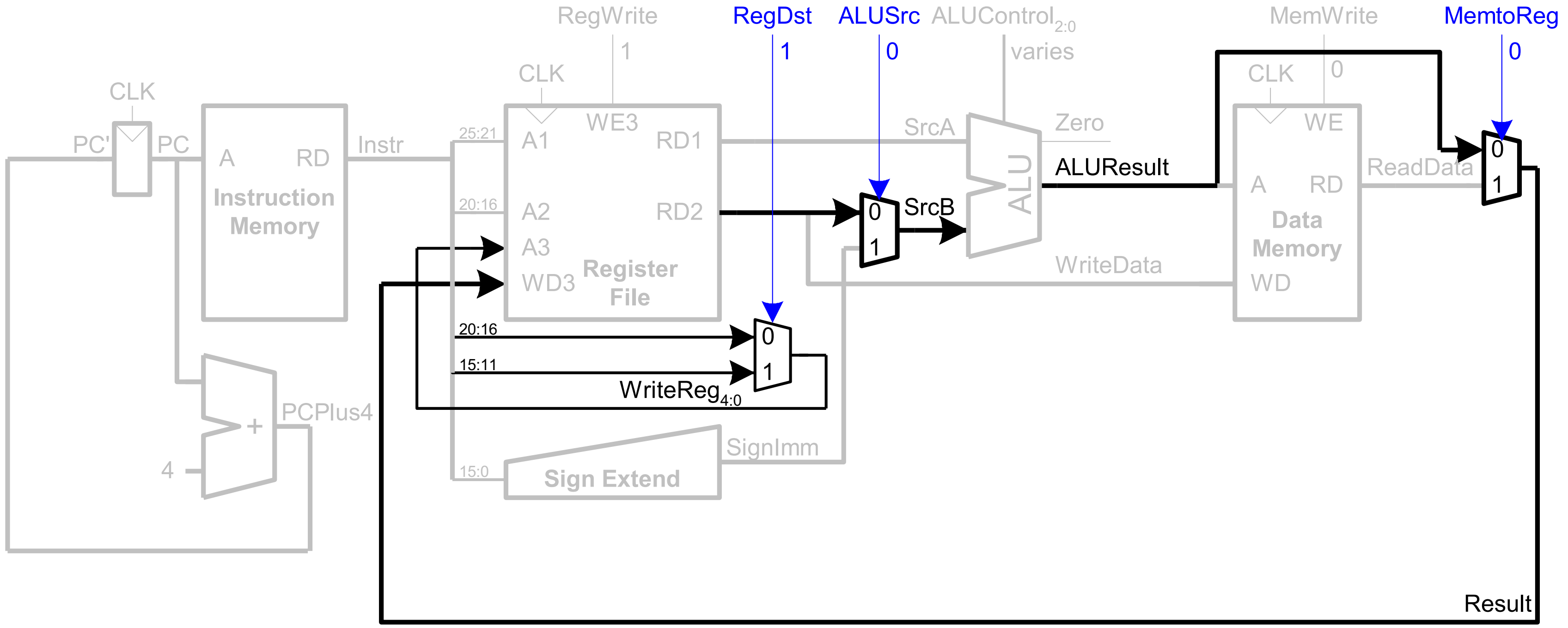
```
sw $t7, 44($0)  # write t7 into memory address 44
```

I-Type

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

Single-Cycle Datapath: R-type Instructions

- **Read from rs and rt, write ALUResult to register file**

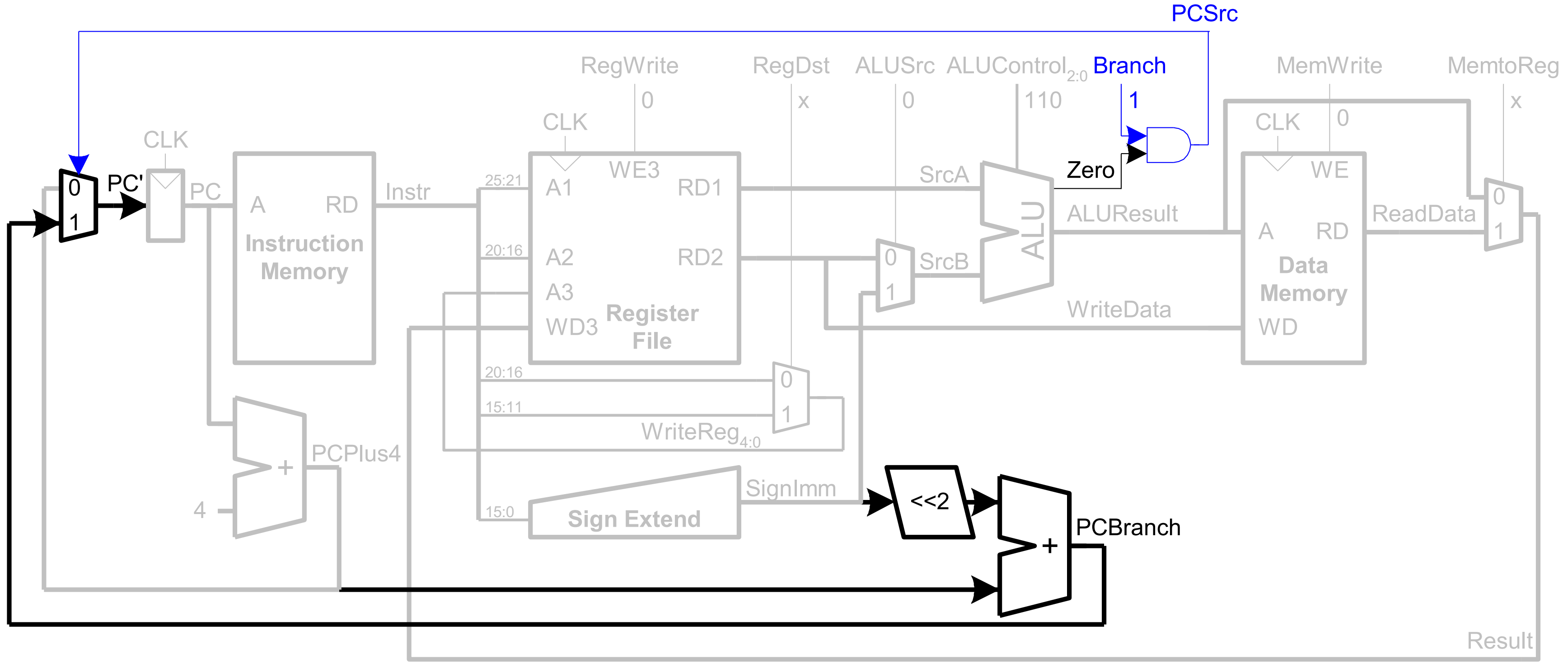


```
add t, b, c  # t = b + c
```

R-Type

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

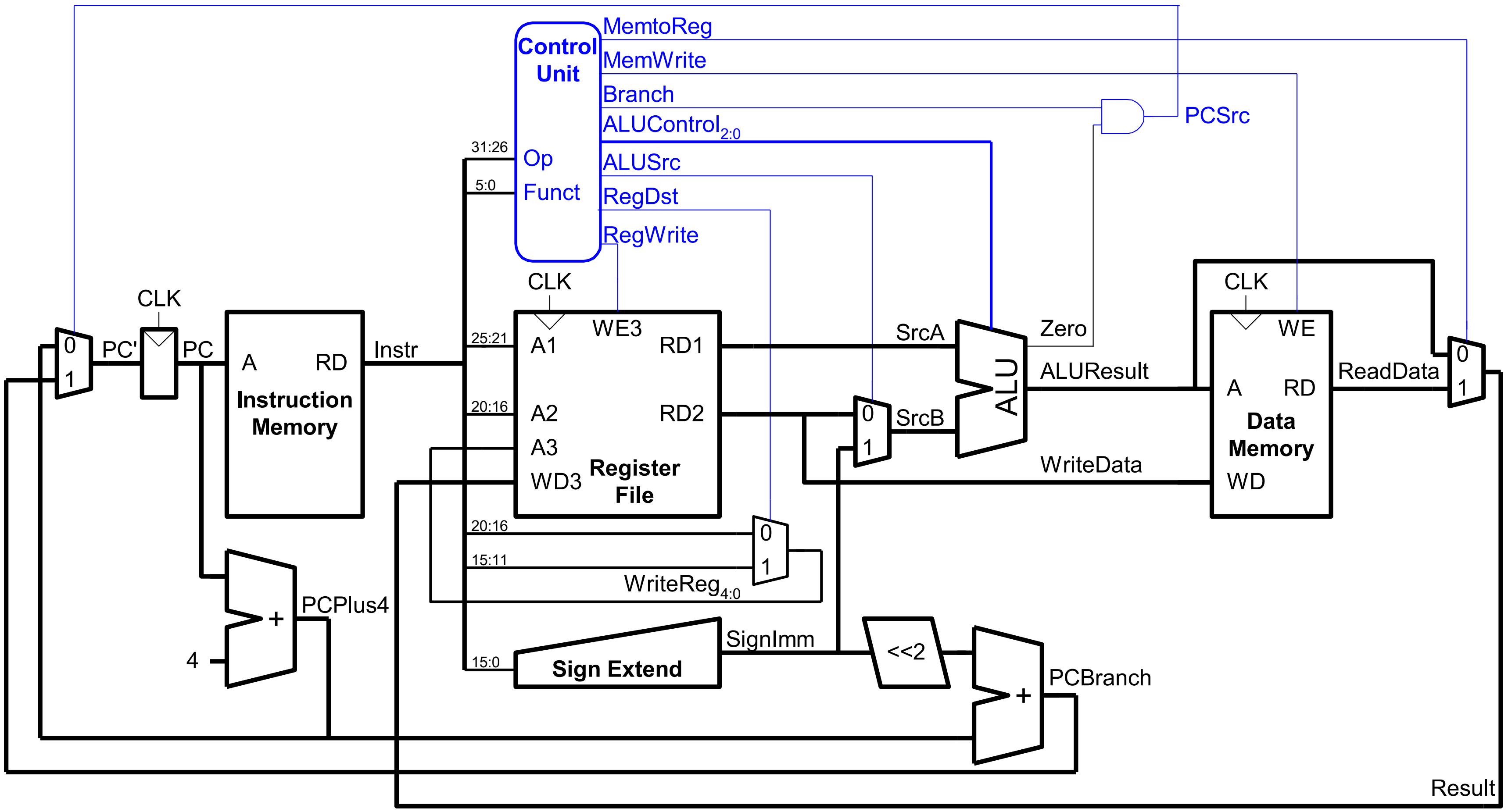
Single-Cycle Datapath: beq



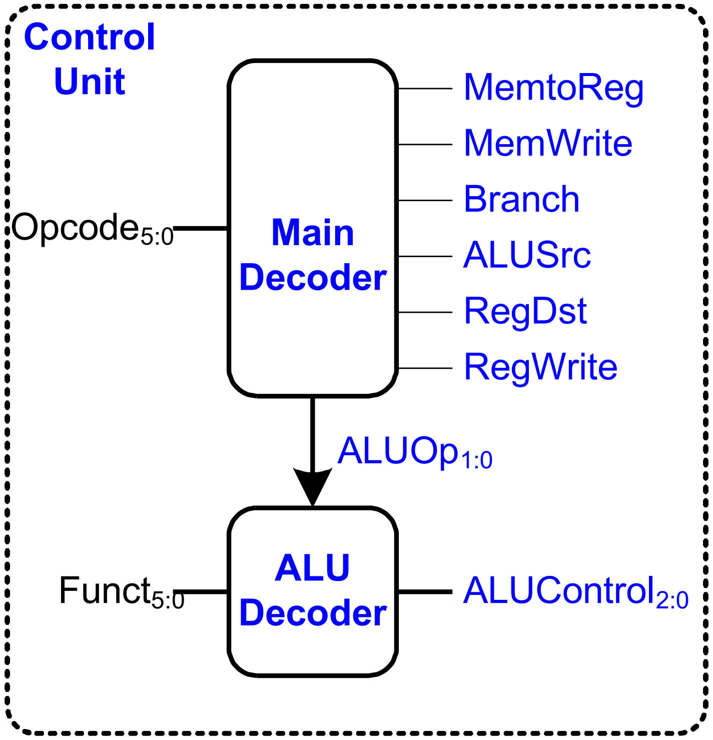
```
beq  $s0, $s1, target  # branch is taken
```

- **Determine whether values in rs and rt are equal**
  Calculate BTA = (sign-extended immediate << 2) + (PC+4)

Complete Single-Cycle Processor

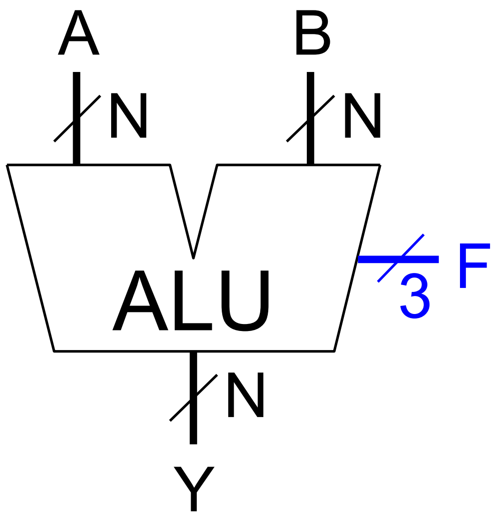


Control Unit



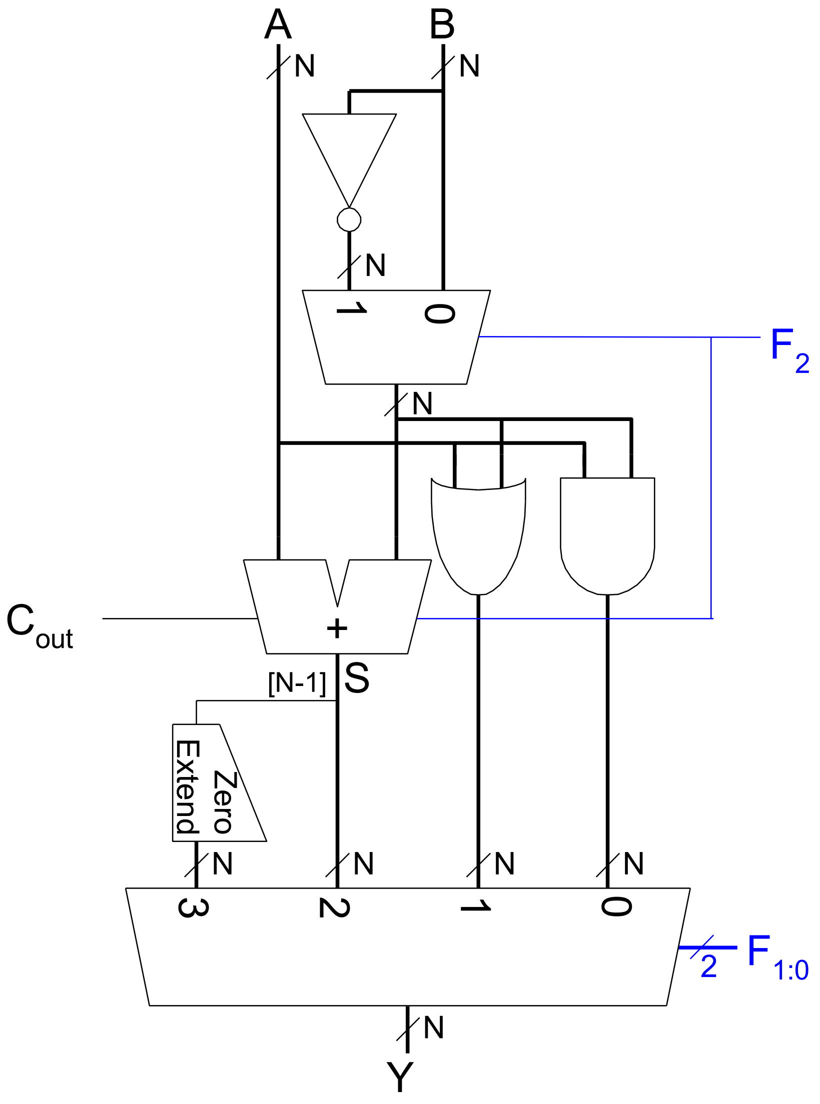
ALUOp	Meaning
00	add
01	subtract
10	look at `funct` field
11	n/a

ALU Does the Real Work in a Processor



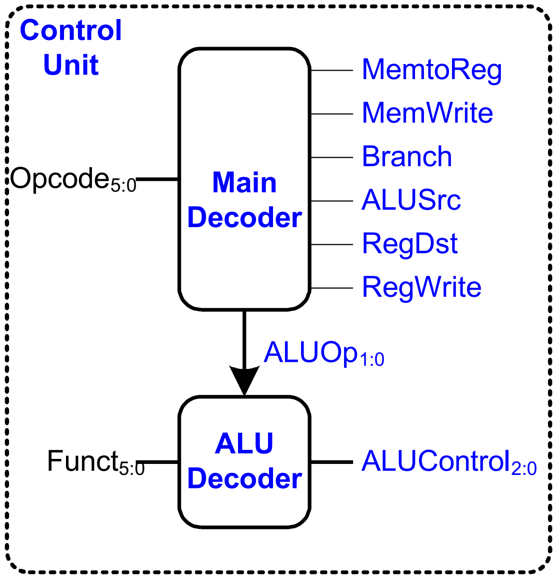
$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT

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

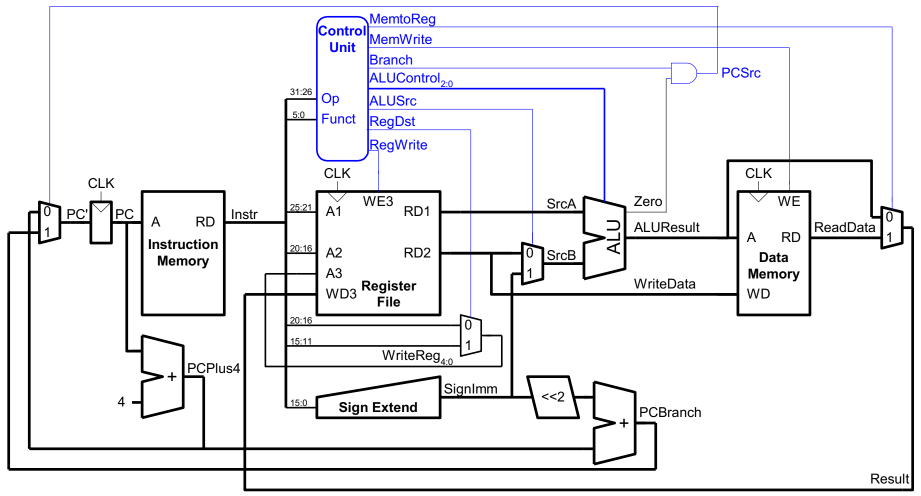


ALUOp$_{1:0}$	Meaning
00	Add
01	Subtract
10	Look at Funct
11	Not Used

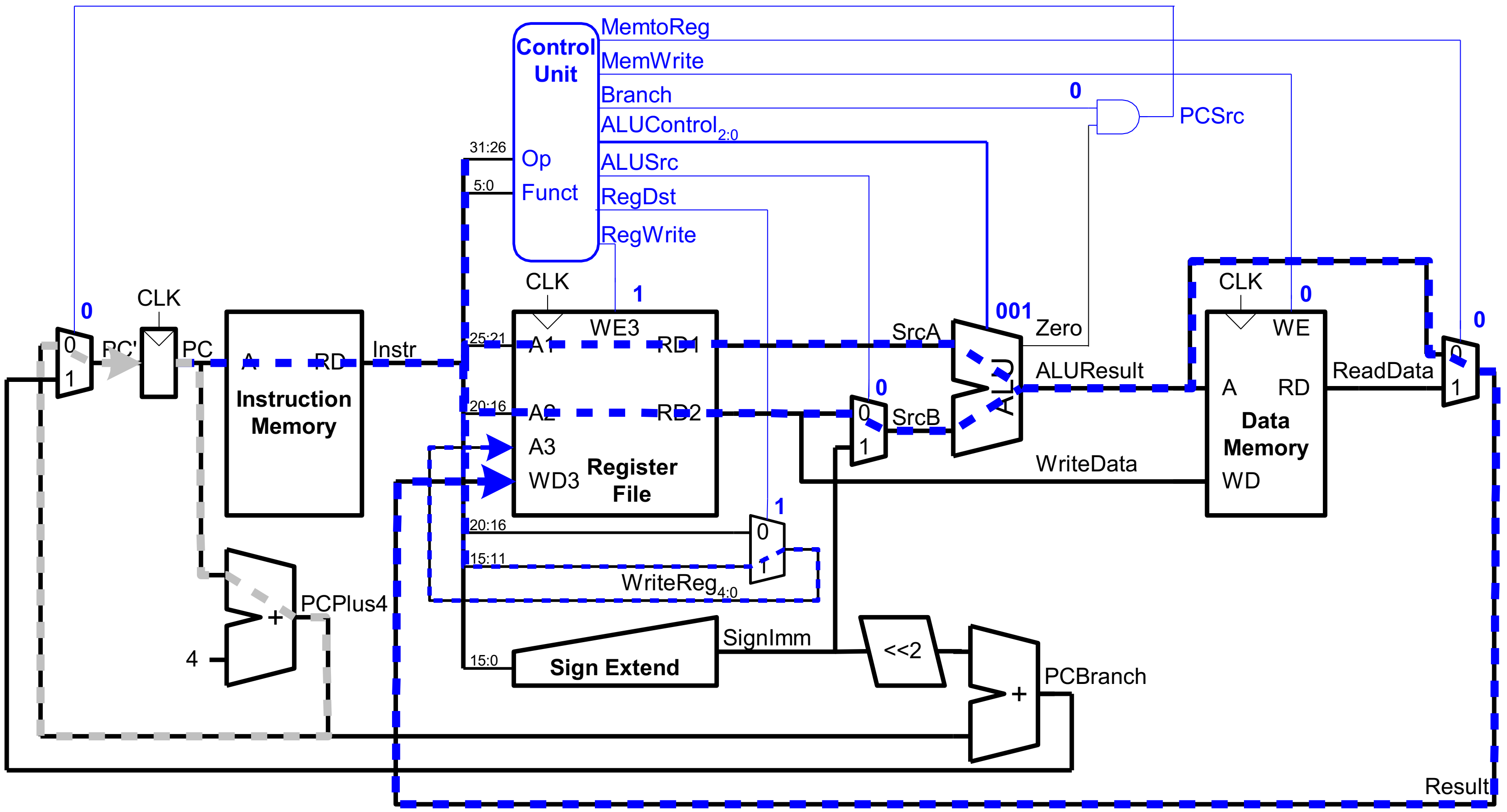
ALUOp$_{1:0}$	Funct	ALUControl$_{2:0}$
00	X	010 (Add)
X1	X	110 (Subtract)
1X	100000 (add)	010 (Add)
1X	100010 (sub)	110 (Subtract)
1X	100100 (and)	000 (And)
1X	100101 (or)	001 (Or)
1X	101010 (slt)	111 (SLT)

Control Unit: Main Decoder

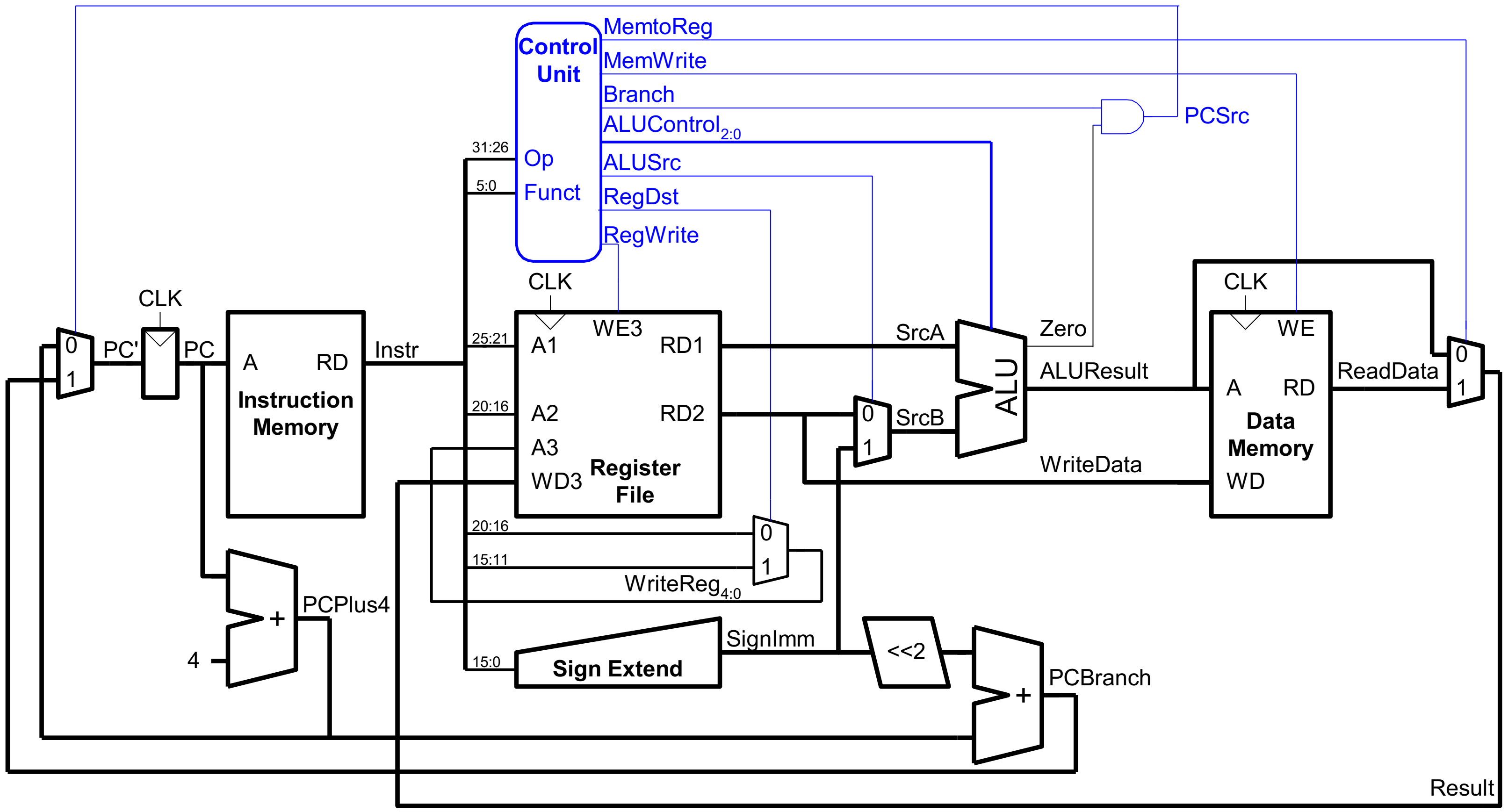
Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01



Single-Cycle Datapath Example: or



Extended Functionality: addi

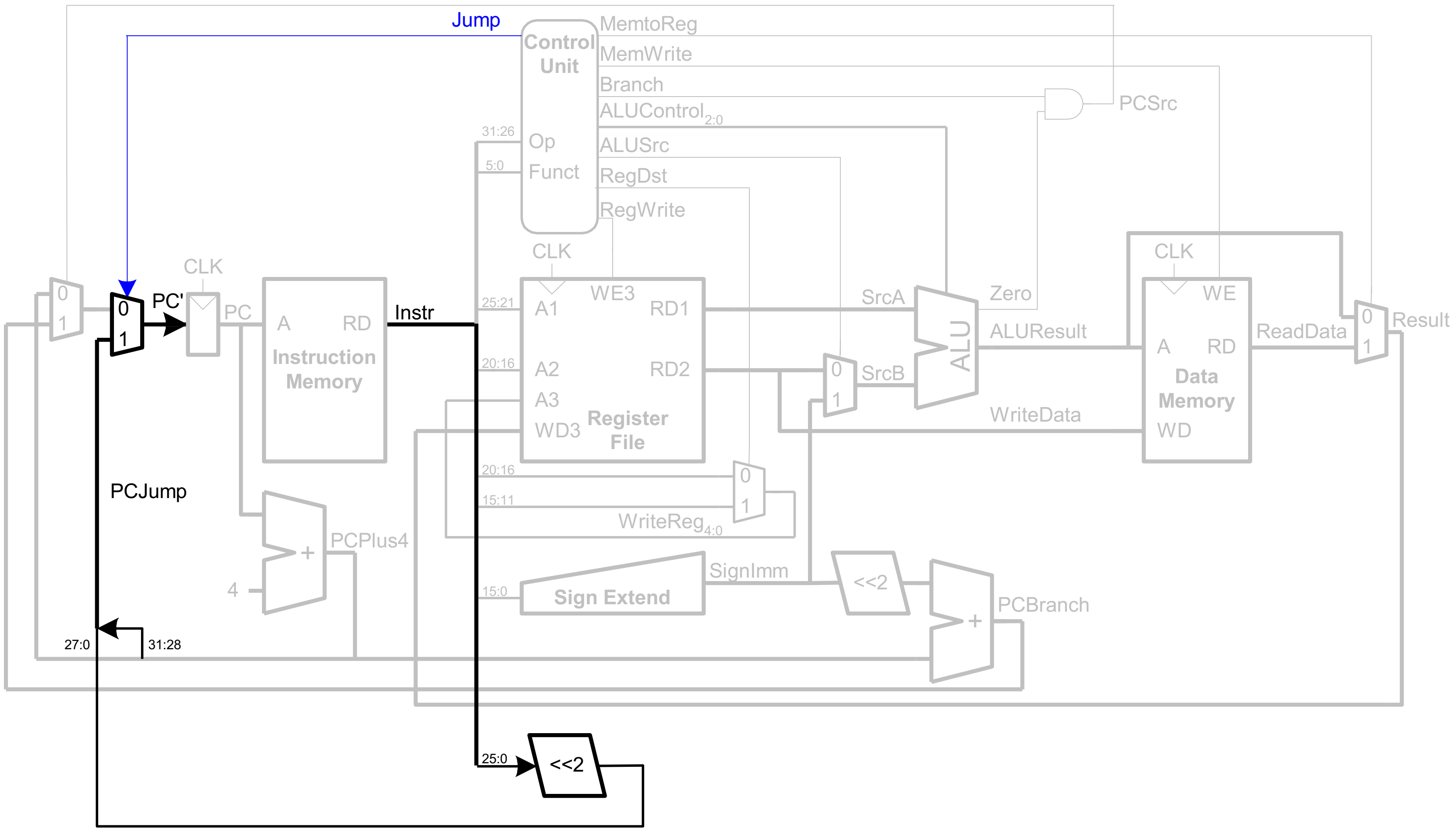


- No change to datapath

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	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	0	X	X	X	0	X	XX	1

Processor Performance

- **How fast is my program?**
  - Every program consists of a series of instructions
  - Each instruction needs to be executed.

Processor Performance

- **How fast is my program?**
  - Every program consists of a series of instructions
  - Each instruction needs to be executed.

- **So how fast are my instructions ?**
  - Instructions are realized on the hardware
  - They can take one or more clock cycles to complete
  - ***Cycles per Instruction = CPI***

Processor Performance

- **How fast is my program?**
  - Every program consists of a series of instructions
  - Each instruction needs to be executed.

- **So how fast are my instructions ?**
  - Instructions are realized on the hardware
  - They can take one or more clock cycles to complete
  - ***Cycles per Instruction = CPI***

- **How much time is one clock cycle?**
  - The critical path determines how much time one cycle requires = ***clock period***.
  - 1/clock period = ***clock frequency*** = how many cycles can be done each second.

Processor Performance

- **Now as a general formula**
  - Our program consists of executing **N** instructions.
  - Our processor needs **CPI** cycles for each instruction.
  - The maximum clock speed of the processor is **f**,
    and the clock period is therefore **T**=1/f

Processor Performance

- **Now as a general formula**
  - Our program consists of executing **N** instructions.
  - Our processor needs **CPI** cycles for each instruction.
  - The maximum clock speed of the processor is **f**,
    and the clock period is therefore **T**=1/f
- **Our program will execute in**

$$N \times CPI \times (1/f) = N \times CPI \times T \text{ seconds}$$

How can I Make the Program Run Faster?

$$N \times CPI \times (1/f)$$

How can I Make the Program Run Faster?

N x CPI x (1/f)

- **Reduce the number of instructions**
  - Make instructions that ‘do’ more (CISC)
  - Use better compilers

How can I Make the Program Run Faster?

N x CPI x (1/f)

- **Reduce the number of instructions**
  - Make instructions that ‘do’ more (CISC)
  - Use better compilers
- **Use less cycles to perform the instruction**
  - Simpler instructions (RISC)
  - Use multiple units/ALUs/cores in parallel

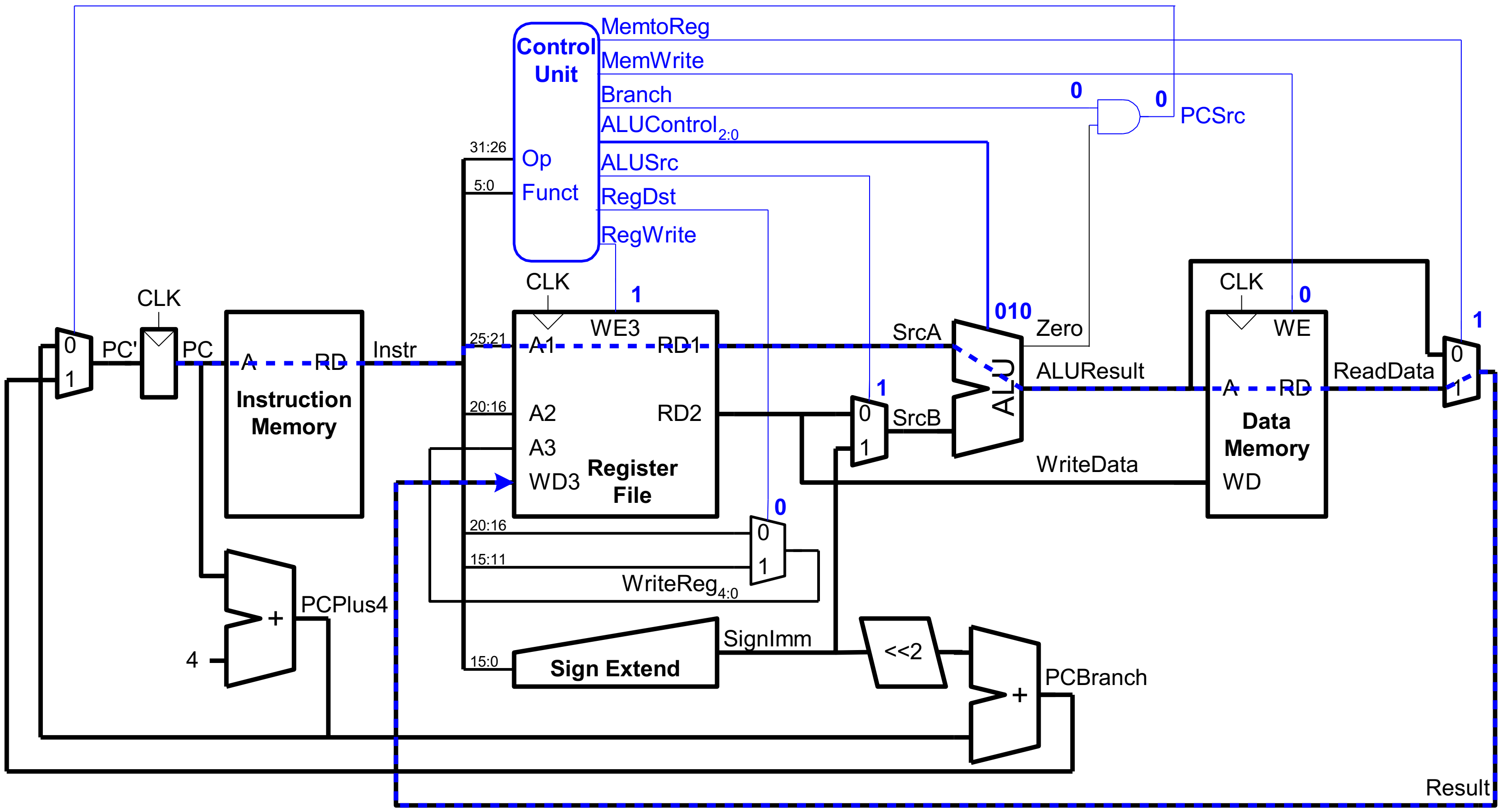
How can I Make the Program Run Faster?

N x CPI x (1/f)

- **Reduce the number of instructions**
  - Make instructions that ‘do’ more (CISC)
  - Use better compilers
- **Use less cycles to perform the instruction**
  - Simpler instructions (RISC)
  - Use multiple units/ALUs/cores in parallel
- **Increase the clock frequency**
  - Find a ‘newer’ technology to manufacture
  - Redesign time critical components
  - Adopt pipelining

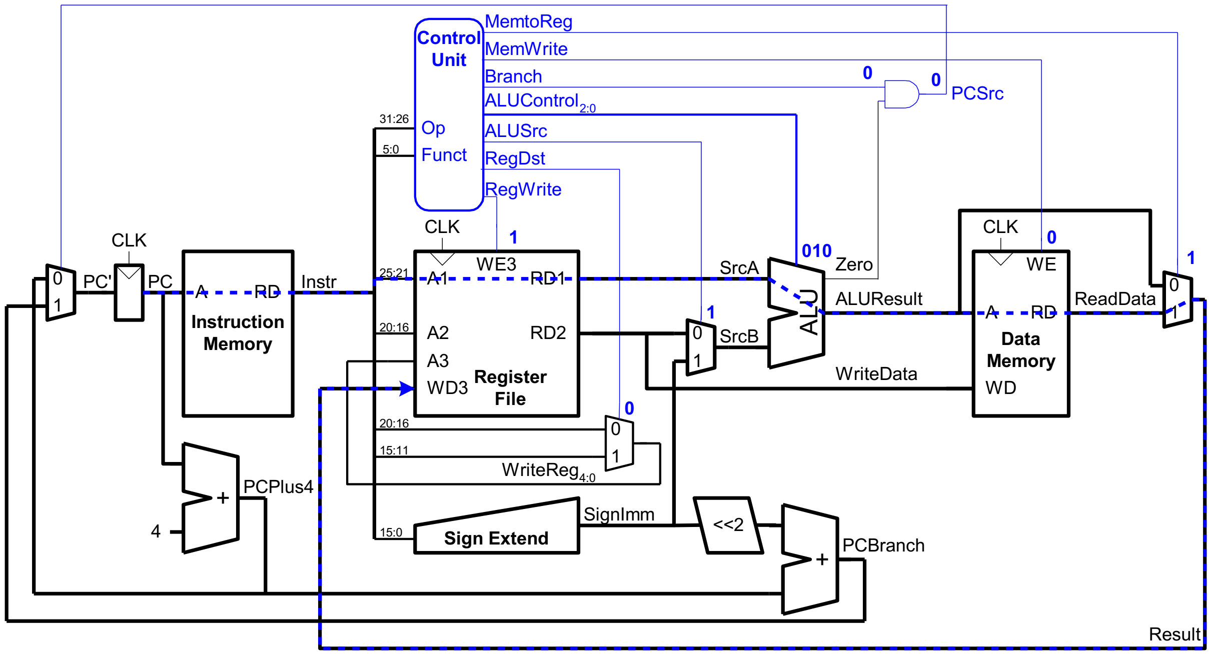
Single-Cycle Performance

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



Single-Cycle Performance

- **Single-cycle critical path:**
  - **$T_c = t_{pcq\_PC} + t_{mem} + \max(t_{RFread}, t_{sext} + t_{mux}) + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup}$**
- **In most implementations, limiting paths are:**
  - memory, ALU, register file.
  - $T_c = t_{pcq\_PC} + 2t_{mem} + t_{RFread} + t_{mux} + t_{ALU} + t_{RFsetup}$



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

$$\begin{aligned} T_c &= t_{pcq\_PC} + 2t_{\mathrm{mem}} + t_{RF\mathrm{read}} + t_{\mathrm{mux}} + t_{\mathrm{ALU}} + t_{RF\mathrm{setup}} \\ &= [30 + 2(250) + 150 + 25 + 200 + 20]\ \mathrm{ps} \\ &= 925\ \mathrm{ps} \end{aligned}$$

Single-Cycle Performance Example

- **Example:**

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

Single-Cycle Performance Example

- **Example:**

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

Execution Time = # instructions x CPI x TC

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

= 92.5 seconds

What Did We Learn?

- **How to determine the performance of processors**
  - CPI
  - Clock speed
  - Number of instructions
- **Single-cycle MIPS architecture**
  - Learned individual components
    - ALU
    - Data Memory
    - Instruction Memory
    - Register File
  - Discussed implementation of some instructions
  - Explained the control flow