Floating-Point Representation Single-Cycle Processor

Dr. Igor Ivkovic

iivkovic@uwaterloo.ca

[with material from “Computer Organization and Design” by Patterson and Hennessy, and “Digital Design and Computer Architecture” by Harris and Harris, both published by Morgan Kaufmann]

Objectives

- Binary Representations of Numbers with Fractions
- Floating-Point Unit (FPU) Circuitry
- Single-Cycle Processor

Representing Numbers with Fractions /1

- **Two common notations for representing fractions:**
  - **Fixed-point:** Binary point fixed
  - **Floating-point:** Binary point floats to the right of the most significant 1
- **Fixed-Point Numbers:**
  - 6.75 using 4 integer bits and 4 fraction bits:

01101100

0110.1100

$$2^2 + 2^1 + 2^{-1} + 2^{-2} = 6.75$$

  - **Binary point is implied**
  - The number of integer and fraction bits must be agreed upon beforehand

Representing Numbers with Fractions /2

- Example1:
  - **Represent 7.510 using 4 integer bits and 4 fraction bits:**
  - $0111.1000_2 = (2^2 + 2^1 + 2^0 + 2^{-1})$
  - Why not add 2^{-3}? Since 0.0625 exceeds 0.01
- **Example2:**
  - **How about 0.84332 using 4 fraction bits?**
  - $0.84332 \geq 2^{-1}$ so place 1 for 2^{-1} bit
    - Update 0.84332 – 0.5 = 0.34332
  - $0.34332 \geq 2^{-2}$ so place 1 for 2^{-2} bit
    - Update 0.34332 – 0.25 = 0.09332
  - $0.09332 < 2^{-3}$ so place 0 for 2^{-3} bit
  - $0.09332 \geq 2^{-4}$ so place 1 for 2^{-4} bit
    - Update 0.09332 – 0.0625 = 0.03082
  - **The result is 0.1101_2 (0.8125_{10})**

Representing Numbers with Fractions /3

- **Two Representations:**
  - Signed / magnitude representation
  - Two's complement representation
- **Example:**
  - Represent -7.510 using 4 integer and 4 fraction bits
  - **Signed / magnitude:** 1111.1000 ($-2^2 - 2^1 - 2^0 - 2^{-1}$)
  - **Two's complement:**

    Step1. Represent absolute value: $\underline{01111000}$

    Step2. Invert the bits: 10000111

    Step3. Add 1 to LSB: $\underline{+\quad\quad 1}$

    10001000

Representing Numbers with Fractions /4

- **Binary point floats to the right of the most significant 1**
  - Similar to decimal scientific notation
  - For example, write 273_{10} in scientific notation:

    $273 = 2.73 \times 10^2$
  - **In general, a number is written in scientific notation as:**

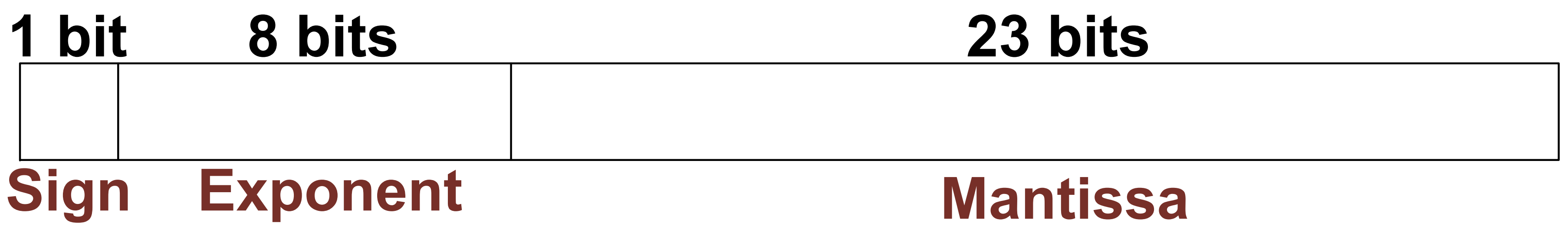
    $\pm M \times B^E$

    M = mantissa

    B = base

    E = exponent
  - In the example, M = 2.73, B = 10, and E = 2

Representing Numbers with Fractions /5



- **Example:**
  - Represent the value 228_{10} using a 32-bit floating point representation (128 + 64 +32 + 4 = 228)
  - We shall examine three different representations for the above
  - The final version is called the IEEE 754 floating-point standard

Representing Numbers with Fractions /6

- **Representation1.**
  - Step1. Convert decimal to binary: $228_{10} = 11100100_2$
  - Step2. Write the number in binary scientific notation:

    $11100100_2 = 1.11001_2 \times 2^7$
  - Step3. Fill in each field of the 32-bit floating point number:
    - The sign bit is positive (0)
    - The 8 exponent bits represent the value 7
    - The remaining 23 bits are the mantissa

1 bit	**8 bits**	**23 bits**
0	00000111	11 1001 0000 0000 0000 0000
Sign	**Exponent**	**Mantissa**

Representing Numbers with Fractions /7

- **Representation2.**
  - The first bit of the mantissa is always 1:

    $228_{10} = 11100100_2 = 1.11001 \times 2^7$
  - So no need to store the first: implicit leading 1
  - Store just the fraction bits in the 23-bit field

1 bit	8 bits	23 bits
0	00000111	110 0100 0000 0000 0000 0000
Sign	**Exponent**	**Fraction**

Representing Numbers with Fractions /8

- **Representation3.**
  - **Biased exponent:** bias = 127 (01111111_2)
  - **Biased exponent = bias + exponent**
    - Ensures that the exponent is unsigned
    - To store –4 as exponent, store –4 + 127 = 123 (01111011_2)
  - Exponent of 7 is stored as:

    127 + 7 = 134 = 10000110_2
  - The IEEE 754 32-bit floating-point (FP) representation of 228_{10}

1 bit	8 bits	23 bits
0	10000110	110 0100 0000 0000 0000 0000
Sign	Biased Exponent	Fraction

Representing Numbers with Fractions /9

1 bit	8 bits	23 bits
0	10000110	110 0100 0000 0000 0000 0000
Sign	**Biased Exponent**	**Fraction**

- **The above is represented in compliance with IEEE Std 754**
- Developed in response to divergence of representations, and to ensure portability issues for scientific code
- Now it is almost universally adopted as a standard

- **Two IEEE Std 754 floating point representations:**
  - Single precision (32-bit), with 1/8/23 bits
  - Double precision (64-bit), with 1/11/52 bits

Representing Numbers with Fractions /10

- **Example1: Write -58.25_{10} in IEEE 754 FP Std**
  - Convert decimal to binary: $58.25_{10} = 111010.01_2$
  - Write in binary scientific notation: 1.1101001×2^5
  - **Significand/Mantissa is Fraction with the "1." restored**
  - Fill in the fields:

    Sign bit: 1 (for negative)

    8 exponent bits: $(127 + 5) = 132 = 1000\ 0100_2$

    23 fraction bits: 110 1001 0000 0000 0000 0000

1 bit	8 bits	23 bits
1	100 0010 0	110 1001 0000 0000 0000 0000
Sign	**Exponent**	**Fraction**

Representing Numbers with Fractions /11

- **Example2: Write $–0.75_{10}$ in IEEE 754 FP Std**
  - Convert decimal to binary: $0.75_{10} = 0.11_2$
  - Write in binary scientific notation: $1.1_2 \times 2^{-1}$
  - Fill in the fields:

    Sign bit: 1 (for negative)

    8 exponent bits: (127 – 1) = 126 = $0111\ 1110_2$

    23 fraction bits: $100\ 0000\ 0000\ 0000\ 0000_2$

1 bit	**8 bits**	**23 bits**
1	011 1111 0	100 0000 0000 0000 0000 0000
Sign	**Exponent**	**Fraction**

Representing Numbers with Fractions /12

- **Example3: What is 11000000101000…00 in decimal?**
  - Sign = 1 (implies negative number)
  - Exponent = $10000001_2 = 129_{10}$
  - Bias removed form the exponent = 129 – 127 = **2**
  - Fraction = $(1)01000\ldots00_2 = 1.25_{10}$
  - **Result: x = $(-1) \times 1.25 \times 2^2 = -5.0$**

Representing Numbers with Fractions /13

- **IEEE 754 FP special cases:**

Number	Sign	Exponent	Fraction
0	X	0000 0000	000 0000 0000 0000 0000 0000
∞	0	1111 1111	000 0000 0000 0000 0000 0000
–∞	1	1111 1111	000 0000 0000 0000 0000 0000
NaN	X	1111 1111	non-zero

Illegal values such as SQRT(-1)

Representing Numbers with Fractions /14

- **Single Floating-Point Precision: 32-bit**
  - 1 sign bit, 8 exponent bits, 23 fraction bits, bias = 127
  - **Also called single-precision, single, or float**
- **Double Floating-Point Precision: 64-bit**
  - 1 sign bit, 11 exponent bits, 52 fraction bits, bias = 1023
  - **Also called double-precision or double**

Representing Numbers with Fractions /15

- **Single-Precision Range:**
  - Exponents 0000 0000 and 1111 1111 are reserved as shown
  - **Has a precision of <u>about</u> 7 decimal digits since $\log_{10}(2^{-24}) = -7.225 \approx -7$ (can rely on the 7th digit)**
- **Smallest value has exponent: 00000001**
  - This leads to the actual exponent of $1 - 127 = -126$
  - Fraction: 000…00 leads to significand of 1.0
  - Hence, the range is defined as $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- **Largest value has exponent: 11111110**
  - This leads to the actual exponent $= 254 - 127 = +127$
  - Fraction: 111…11 leads to significand of 1.999999… ≈ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Representing Numbers with Fractions /16

- **Double-Precision Range:**
  - Exponents 0000…00 and 1111…11 are reserved as shown
  - **Has a precision of <u>about</u> 15 decimal digits since $\log_{10}(2^{-53}) = -15.955 \approx -15$ (cannot rely on the 16th digit)**
- **Smallest value has exponent: 00000000001**
  - This leads to the actual exponent of 1 – 1023 = –1022
  - Fraction: 000…00 leads to significand of 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- **Largest value has exponent: 11111111110**
  - This leads to the actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 leads to significand of 1.999999… ≈ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Representing Numbers with Fractions /17

- **Floating-Point Rounding:**
  - **Overflow:** The number is too large to be represented
  - **Underflow:** The number is too small to be represented
- **Rounding modes:**
  - Round Down
  - Round Up
  - Toward zero
  - To nearest
- **Example:** Round 1.100101 (1.578125) to only 3 fraction bits

Round Down:	1.100
Round Up:	1.101
Towards zero:	1.100 (1.625 is farther from 0 than 1.5)
To nearest:	1.101 (1.578125 is closer to 1.625 than 1.5)

Floating-Point Addition /1

- **Floating-Point (FP) Addition Steps:**
  1. Extract exponent and fraction bits
  2. Prepend leading 1 to form mantissa
  3. Compare exponents
  4. Shift smaller mantissa if necessary
  5. Add mantissas
  6. Normalize mantissa and adjust exponent if necessary
  7. Round result
  8. Assemble exponent and fraction back into floating-point format

Floating-Point Addition /2

- **Floating-Point (FP) Example:**
  - **Step1. Extract exponent and fraction bits**

1 bit	8 bits	23 bits
0	01111111	100 0000 0000 0000 0000 0000
Sign	**Exponent**	**Fraction**

What numbers are these in decimal?

1 bit	8 bits	23 bits
0	10000000	101 0000 0000 0000 0000 0000
Sign	**Exponent**	**Fraction**

For the first number N1: S = 0, E = 127, F = .1

For the second number N2: S = 0, E = 128, F = .101

  - **Step2. Prepend leading 1 to form mantissa**

N1: 1.1

N2: 1.101

Floating-Point Addition /3

- **Floating-Point (FP) Example:**
  - **Step3. Compare exponents**

    127 – 128 = -1 so shift N1 right by 1 bit
  - **Step4. Shift smaller mantissa if necessary**

    N1's mantissa: 1.1 >> 1 = 0.11 (equals the original x 2^1)
  - **Step 5. Add mantissas**

    $$\begin{array}{r l} & 0.11 \quad \times 2^1 \\ + & \underline{1.101 \times 2^1} \\ & 10.011 \times 2^1 \end{array}$$
  - **Step6. Normalize mantissa and adjust exponent if needed**

    $10.011 \times 2^1 = 1.0011 \times 2^2$

Floating-Point Addition /4

- **Floating-Point (FP) Example:**
  - **Step7. Round result**

    No need since it fits into 23 bits

  - **Step8. Assemble exponent and fraction into FP format**

    S = 0, E = 2 + 127 = 129 = 10000001_2, F = $001100..0_2$

1 bit	8 bits	23 bits
0	10000001	001 1000 0000 0000 0000 0000
Sign	**Exponent**	**Fraction**

Floating-Point Addition /5

- **Abbreviated Example:**
  - Add: $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ $(0.5 + -0.4375)_{10}$
  - **Step1. Align binary points and shift number with the smaller exponent**

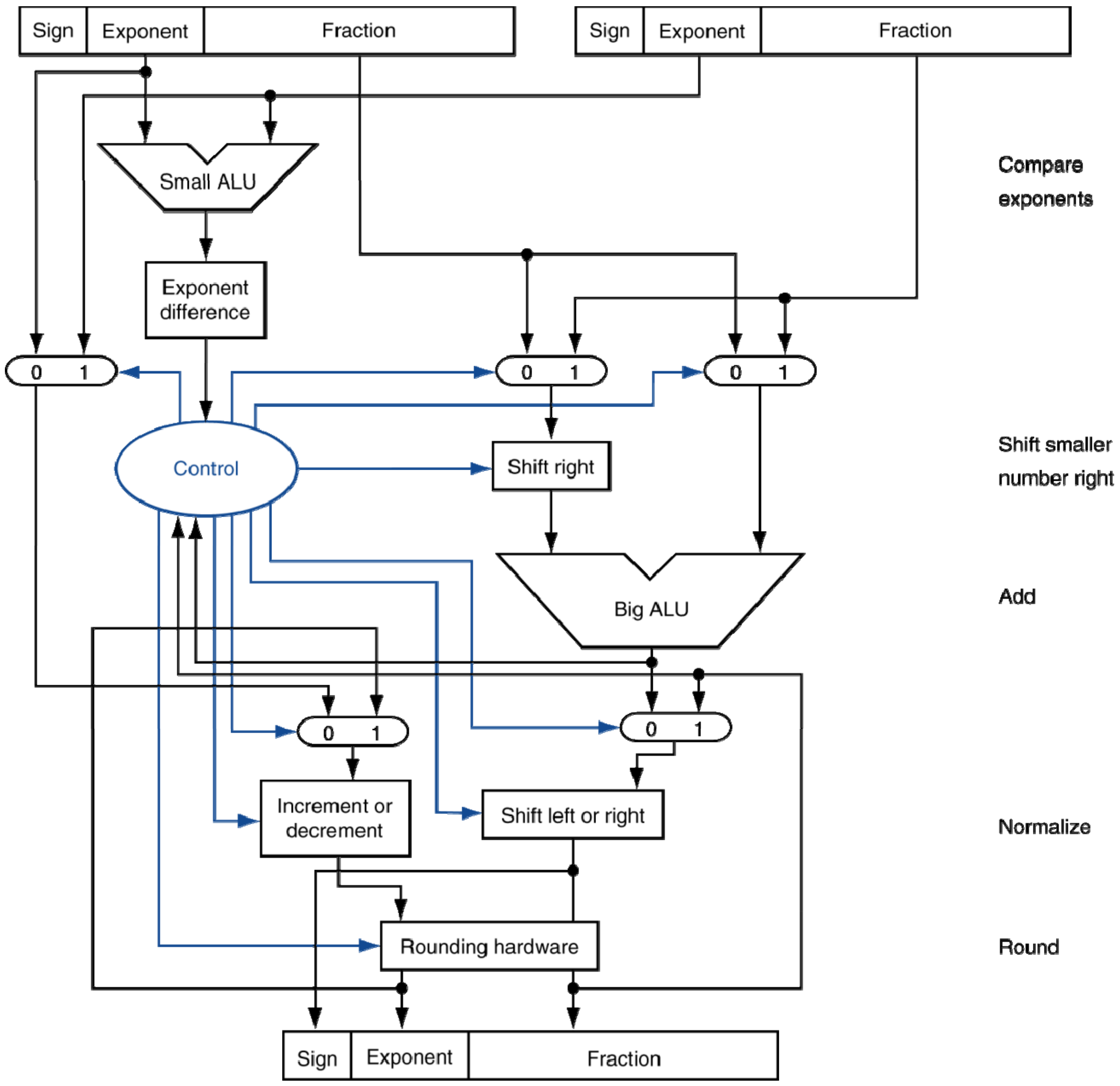
    $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
  - **Step2. Add mantissas**

    $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$
  - **Step3. Normalize result and check for over/underflow**

    $1.000_2 \times 2^{-4}$, no change needed
  - **Step4. Round and renormalize if necessary**

    $1.000_2 \times 2^{-4}$ (no change needed) $= 0.0625_{10}$

Floating-Point Unit (FPU) Circuitry /1



Step1.

Step2.

Step3.

Step4. (go to Step3 until done)

Floating-Point Multiplication Overview

- **Let us consider a 4-digit binary example:**
  - Multiply: $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ $(0.5 \times -0.4375)_{10}$
  - **Step1. Add exponents**

    Unbiased: $-1 + -2 = -3$

    Biased: $(-1 + 127) + (-2 + 127) = -3 + 254 - 127 = -3 + 127$
  - **Step2. Multiply mantissas**

    $1.000_2 \times 1.110_2 = 1.1102 \Rightarrow 1.110_2 \times 2^{-3}$
  - **Step3. Normalize result and check for over/underflow**

    $1.110_2 \times 2^{-3}$ (no change) with no over/underflow
  - **Step4. Round and renormalize if necessary**

    $1.110_2 \times 2^{-3}$ (no change)
  - **Step5. Determine sign: +ve × –ve ⇒ –ve**

    $-1.110_2 \times 2^{-3} = -0.21875$

Floating-Point Unit (FPU) Circuitry /2

- **Floating-Point Adder:**
  - Can be significantly more complex than the integer adder
  - Performing the addition in one clock cycle would take too long
  - Much longer than integer operations
  - Slower clock would penalize all instructions
  - **Floating-point adder usually takes several cycles**
- **FP multiplier is of similar complexity to FP adder:**
  - It uses a multiplier for significands instead of an adder
- **Floating-Point Unit (FPU) circuitry usually does:**
  - Addition, subtraction, multiplication, division, reciprocal, square-root, and FP-to-integer conversion
  - Operations usually takes several cycles
  - FP operations can be pipelined (as discussed later)

Floating-Point Unit (FPU) Circuitry /3

- **IEEE Std 754 specifies additional rounding control:**
  - Choice of rounding modes
  - Allows programmer to fine-tune numerical behavior of a computation
  - Not all FP circuits implement all options
  - Most programming languages and FP libraries just use defaults (e.g., round to nearest)

FP Instructions in MIPS /1

- **FP circuitry is usually a coprocessor (i.e., number 1) that extends the system architecture**
  - It includes separate FP registers
  - 32 single-precision: \$f0, \$f1, … \$f31
  - Paired for double-precision: \$f0/\$f1, \$f2/\$f3, …
    - Release 2 of MIPS supports 32 × 64-bit FP registers
- **FP instructions operate only on FP registers:**
  - Programs typically do no perform integer operations on FP data, or vice versa
  - More registers are available with minimal code-size impact
- **FP load and store instructions (d is for double):**
  - `lwc1, ldc1, swc1, sdc1`
    - e.g., `lwc1 $f8, 32($sp)`

Load/store word on coprocessor 1

FP Instructions in MIPS /2

- **Single-precision arithmetic:**
  - `add.s, sub.s, mul.s, div.s`
    - `e.g., add.s $f0, $f1, $f6`
- **Double-precision arithmetic:**
  - `add.d, sub.d, mul.d, div.d`
    - `e.g., mul.d $f4, $f4, $f6`
- **Single- and double-precision comparison:**
  - `c.xx.s, c.xx.d (xx is eq, lt, le, …)`
  - Sets or clears FP condition-code bit
    - `e.g. c.lt.s $f3, $f4`
- **Branch on FP condition code true or false:**
  - `bc1t, bc1f`
    - `e.g., bc1t TargetLabel`

FP Instructions in MIPS /3

- **Example: °F to °C**
  - **C code:**

```
float f2c (float fahr) {
  return ((5.0/9.0)*(fahr - 32.0));
}
```

    - **fahr in $f12, result in $f0, literals in global memory space**
  - **Compiled MIPS code:**

```
f2c: lwc1  $f16, const5($gp)
     lwc2  $f18, const9($gp)
     div.s $f16, $f16, $f18
     lwc1  $f18, const32($gp)
     sub.s $f18, $f12, $f18
     mul.s $f0,  $f16, $f18
     jr    $ra
```

Single-Cycle Processor /1

- **Microarchitecture:**
  - How to implement an architecture in hardware
- **Processor includes:**
  - **Datapath:** Functional blocks
  - **Control:** Control signals (more on these later)
- **Datapath:**
  - Elements that process data and addresses in the CPU
    - Includes registers, ALUs, multiplexers, memory units, etc
  - We will build a MIPS datapath incrementally

Single-Cycle Processor /2

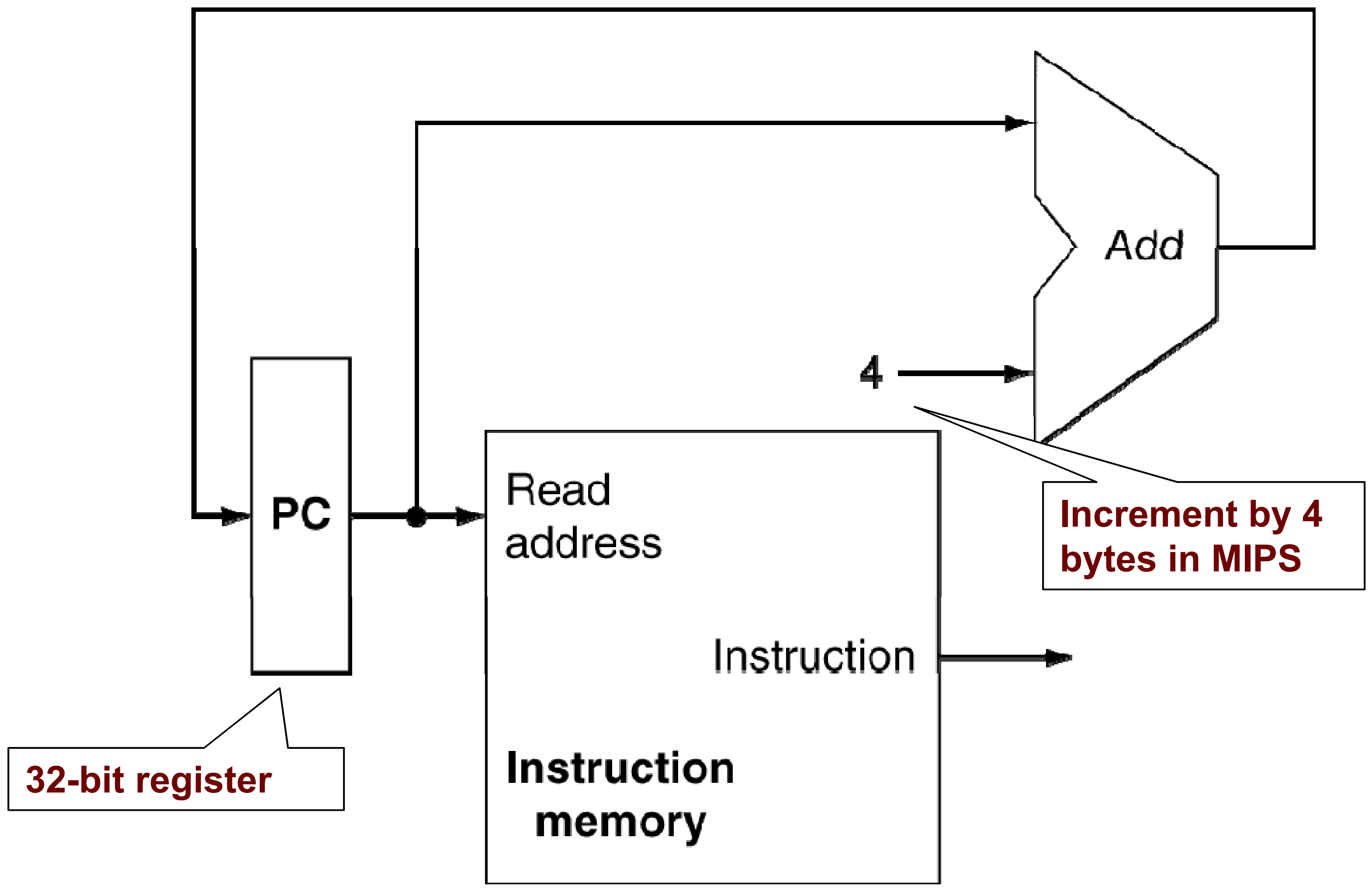
- **Program execution time:**
  - Execution Time = (#instructions)(cycles/instruction)(seconds/cycle)
- Recall these definitions:
  - **CPI:** Cycles per Instruction
  - **Clock period:** Seconds per clock cycle
  - **IPC:** Instructions per cycle
- **The challenge of architecture design is to meet the constraints of cost, power, and performance**

Single-Cycle Processor /3

- **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 and multiple instructions are executed at once
- **The processor state in MIPS is determined by:**
  - Program Counter (PC): Address value of the current instruction being executed
  - 32 registers and Memory
  - MIPS has 32 32-bit registers

Single-Cycle Processor /4

- **Instruction Fetch by adding 4 to PC address:**

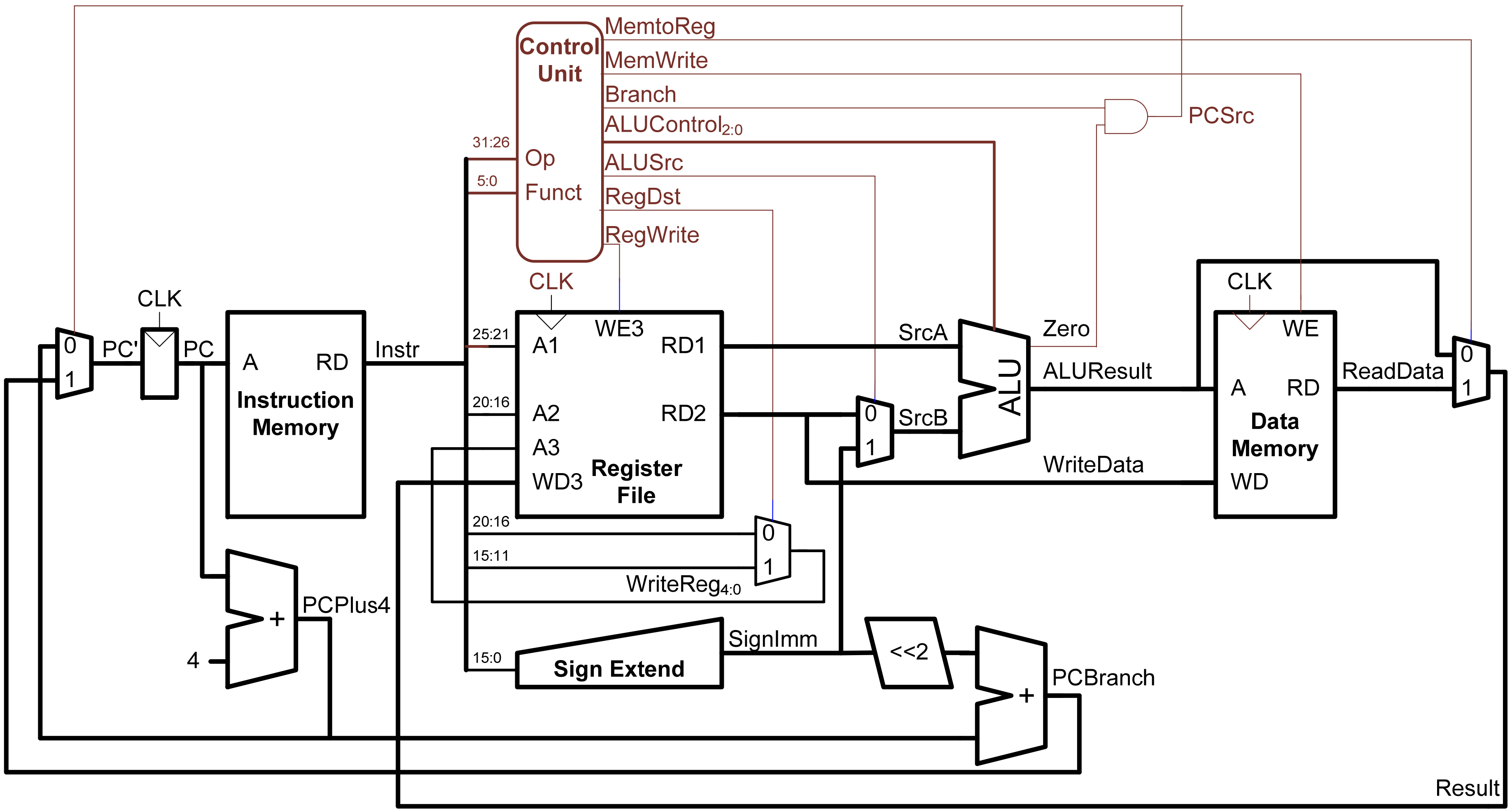


Single-Cycle Processor /5

- We shall start with a subset of MIPS instructions:
  - R-type instructions: `and, or, add, sub, slt`
  - Memory instructions (I-type instructions): `lw, sw`
  - Branch instructions (J-type instructions): `beq`
- **Our goal is to show how these instructions are executed on a single-cycle processor**

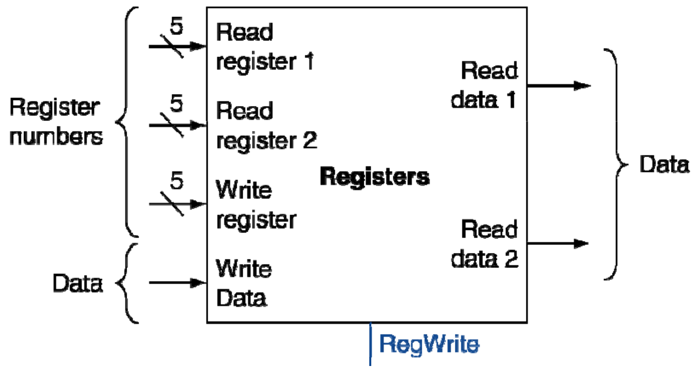
Single-Cycle Processor /6

- **Our goal: explain the processor architecture below**



Single-Cycle Processor /7

- **R-Type (Register-Type) Instruction:**
  - Reads two register operands
  - Performs arithmetic/logical operation
  - Writes register result



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

R-Type

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

Single-Cycle Processor /8

- **R-Type Assembly Code:**
  - **Order of registers:** `add rd, rs, rt`

Field Values

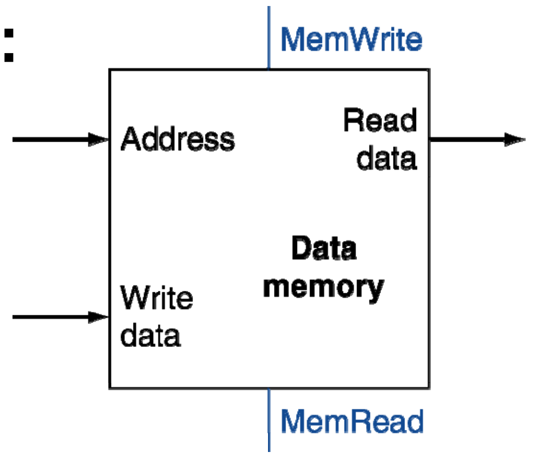
	op	rs	rt	rd	shamt	funct
add $r16, $r17, $r18	0	17	18	16	0	32
sub $r8, $r11, $r13	0	11	13	8	0	34
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Machine Code

op	rs	rt	rd	shamt	funct	Hexadecimal Value
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	

Single-Cycle Processor /9

- **I-Type (Immediate-Type) Instruction:**
  - Uses immediate-value operand
- **3 register operands:**
  - `rs, rt`: register operands
  - `imm`: 16-bit two's complement immediate
  - `op`: the opcode
  - The operation is completely determined by the opcode



I-Type

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

Single-Cycle Processor /10

- **I-Type Assembly Code:**
  - **Order of registers:** `lw rt, imm(rs)` & `addi rt, rs, imm`

Assembly Code / **Field Values**

Assembly Code	op	rs	rt	imm
`addi $s0, $s1, 5`	8	17	16	5
`addi $t0, $s3, -12`	8	19	8	-12
`lw $t2, 32($0)`	35	0	10	32
`sw $s1, 4($t1)`	43	9	17	4
	6 bits	5 bits	5 bits	16 bits

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	

addi rt, rs, imm

lw rt, imm(rs)

sw rt, imm(rs)

Single-Cycle Processor /11

- **MIPS Register Set:**

Name	Register Number	Usage
$0	0	the constant value 0
$at	1	assembler temporary
$v0-$v1	2-3	Function return values
$a0-$a3	4-7	Function arguments
$t0-$t7	8-15	temporaries
$s0-$s7	16-23	saved variables
$t8-$t9	24-25	more temporaries
$k0-$k1	26-27	OS temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	Function return address

Single-Cycle Processor /12

- **Identifying registers by using $ before name:**
  - Example: $0, “register zero”, “dollar zero”
- **Registers used for specific purposes:**
  - $0 always holds the constant value 0
  - The saved registers, $s0-$s7, are used to hold variables
  - The temporary registers, $t0 - $t9, are used to hold intermediate values during a larger computation
- **Too much data to fit into only 32 registers:**
  - Store more data in memory
  - Memory is large but it is slow
  - Commonly used variables can be kept in registers
  - **Use load and store operations to interact with the memory**

Single-Cycle Processor /13

- **Each 32-bit data word has a unique address:**

Word Address	Data	
⋮	⋮	⋮
00000003	4 0 F 3 0 7 8 8	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

- **Reading Word-Addressable Memory:**
  - **Load word (lw):** `lw $s0, 2($t1)`
  - Add base address ($t1) and the offset (2); address = ($t1 + 2)
  - Result: $s0 holds the value at address ($t1 + 2)
  - **Example:** `lw $s0, 2($0)`, **Result:** $s0 = 0x01EE2842

Single-Cycle Processor /14

- **Each 32-bit data word has a unique address:**

Word Address	Data	
⋮	⋮	⋮
00000003	4 0 F 3 0 7 8 8	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

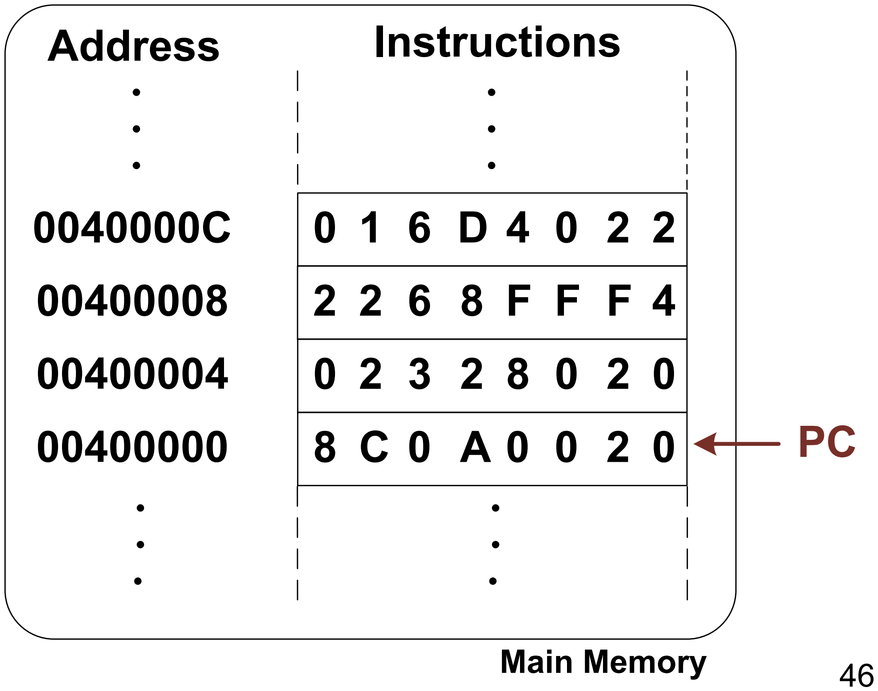
- **Writing Word-Addressable Memory:**
  - **Store word (sw):** `sw $s0, 4($t4)`
  - Add base address ($t4) and the offset (4); address = ($t4 + 4)
  - Result: stores the value in $s0 to the address ($t4 + 4)
  - **Example:** `sw $s0, 4($0)`, **Result:** address 4 holds $s0

Single-Cycle Processor /15

- **32-bit instructions and data are stored in memory**
  - **Sequence of instructions:** only difference between two programs
  - **To run a new program:** Simply store the new program in memory
  - **Program Execution:** Processor fetches (reads) instructions from memory in sequence
  - Processor performs the specified operation

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



Single-Cycle Processor /16

- **Start with opcode:**
  - Tells us how to parse rest
  - If opcode all 0s then it is an R-type instruction
  - Function bits tell the operation
  - If opcode is not 0s, then opcode tells the operation
- **Logical Instructions Sample:**

What are these in Hexadecimal?

Source Registers

$s1	1111	1111	1111	1111	0000	0000	0000	0000
$s2	0100	0110	1010	0001	1111	0000	1011	0111

Assembly Code		Result							
`and $s3, $s1, $s2`	$s3	0100	0110	1010	0001	0000	0000	0000	0000
`or  $s4, $s1, $s2`	$s4	1111	1111	1111	1111	1111	0000	1011	0111
`xor $s5, $s1, $s2`	$s5	1011	1001	0101	1110	1111	0000	1011	0111
`nor $s6, $s1, $s2`	$s6	0000	0000	0000	0000	0000	1111	0100	1000

Single-Cycle Processor /17

- **Shift Operations:**
  - **sll: shift left logical**
  - Example:
    ```
    sll $t0, $t1, 5  # $t0 <= $t1 << 5
    ```
  - **srl: shift right logical**
  - Example:
    ```
    srl $t0, $t1, 5  # $t0 <= $t1 >> 5
    ```
  - **sra: shift right arithmetic**
  - Example:
    ```
    sra $t0, $t1, 5  # $t0 <= $t1 >>> 5
    ```

Single-Cycle Processor /18

- **Shift Operations Code:**

Assembly Code / **Field Values**

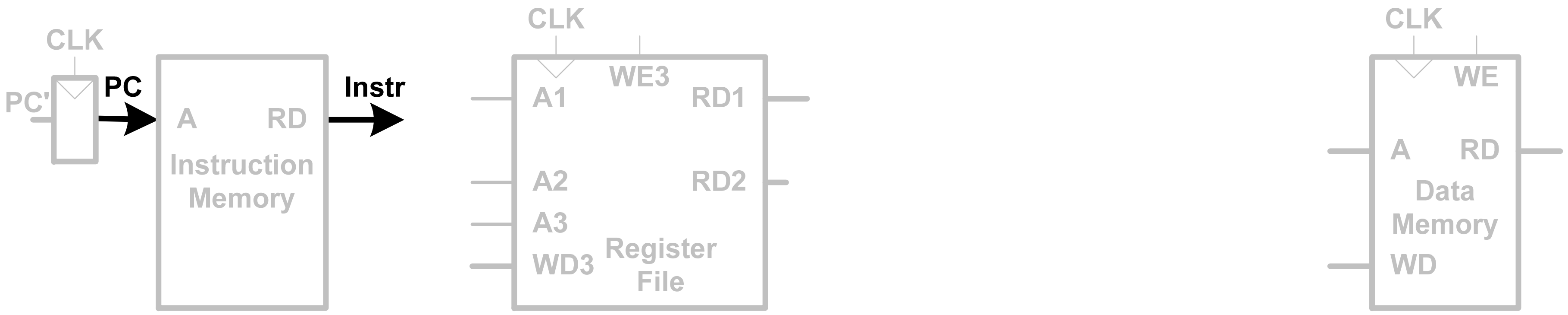
Assembly Code	op	rs	rt	rd	shamt	funct
`sll $t0, $s1, 2`	0	0	17	8	2	0
`srl $s2, $s1, 2`	0	0	17	18	2	2
`sra $s3, $s1, 2`	0	0	17	19	2	3
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Machine Code

op	rs	rt	rd	shamt	funct	
000000	00000	10001	01000	00010	000000	(0x00114080)
000000	00000	10001	10010	00010	000010	(0x00119082)
000000	00000	10001	10011	00010	000011	(0x00119883)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

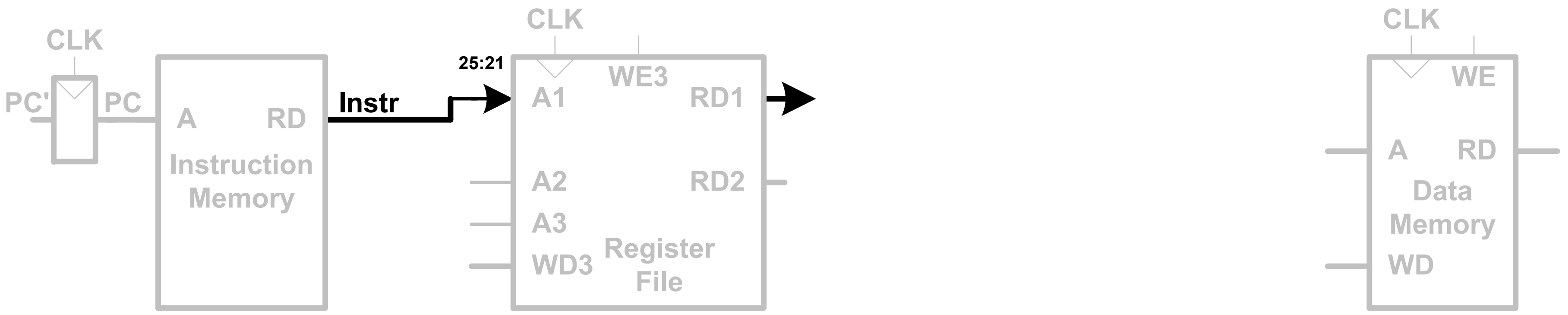
Single-Cycle Datapath Trace for `lw` /1

- **Step 1. Fetch instruction from Instruction Memory:**



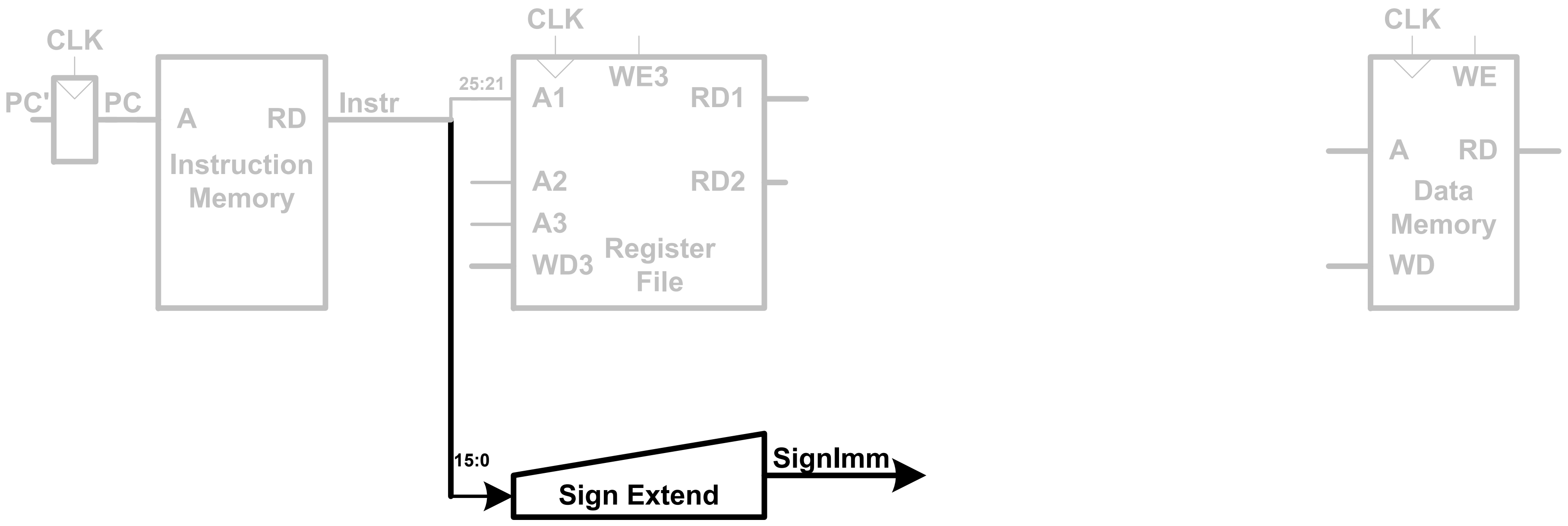
Single-Cycle Datapath Trace for `lw` /2

- **Step 2. Read source operands from Register File:**



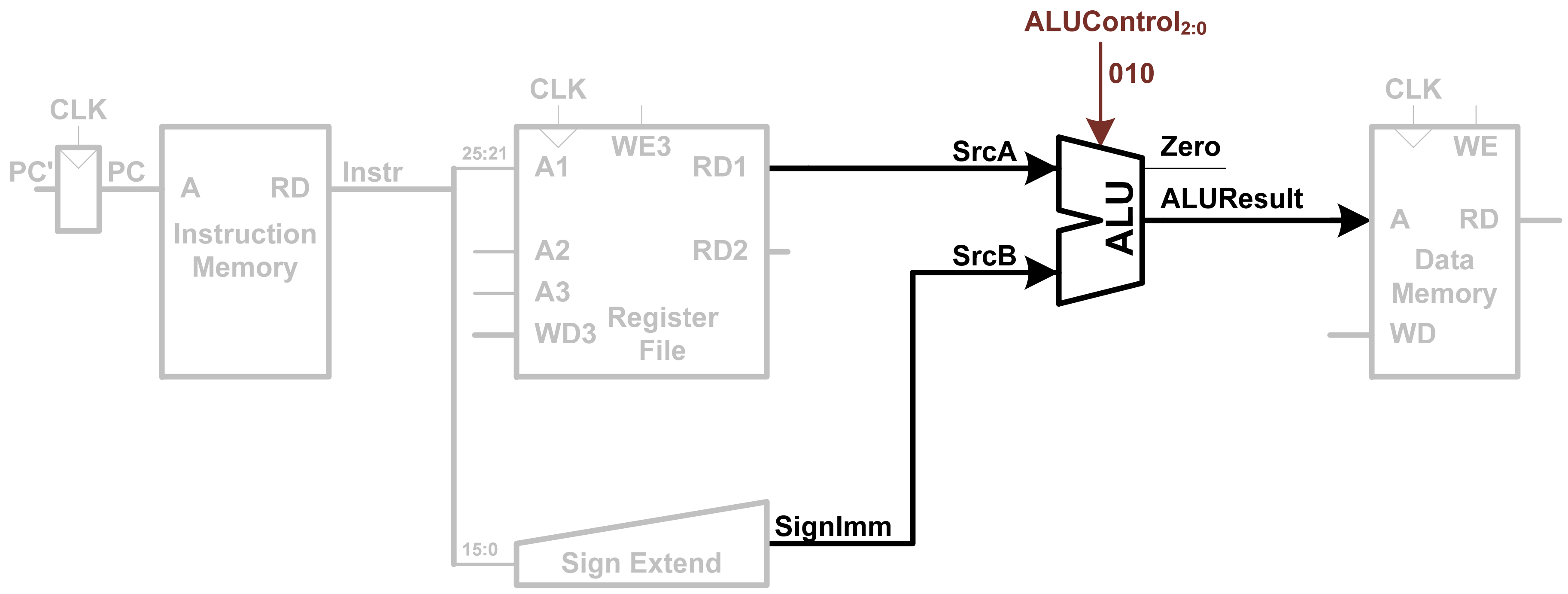
Single-Cycle Datapath Trace for `lw` /3

- **Step 3. Sign-extend the immediate:**



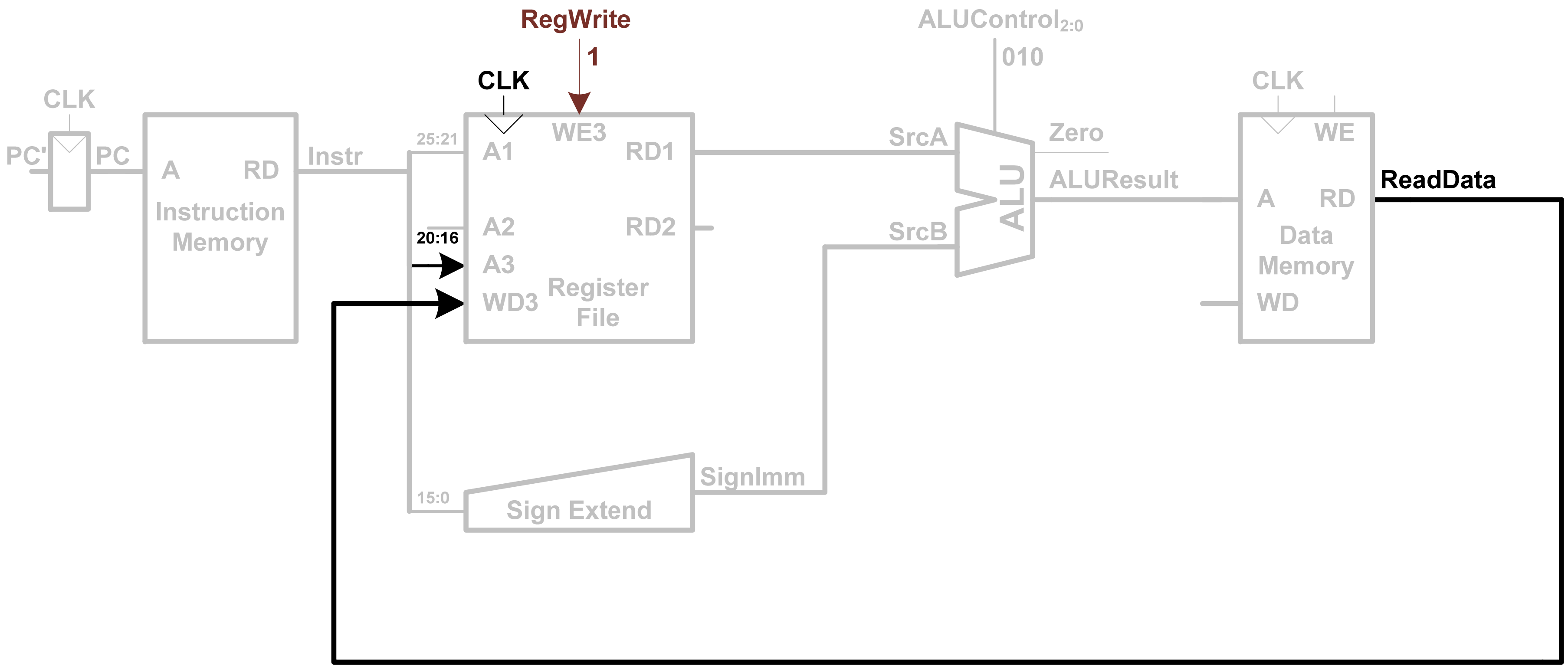
Single-Cycle Datapath Trace for `lw` /4

- **Step 4. Compute the memory address (add srcA/B):**



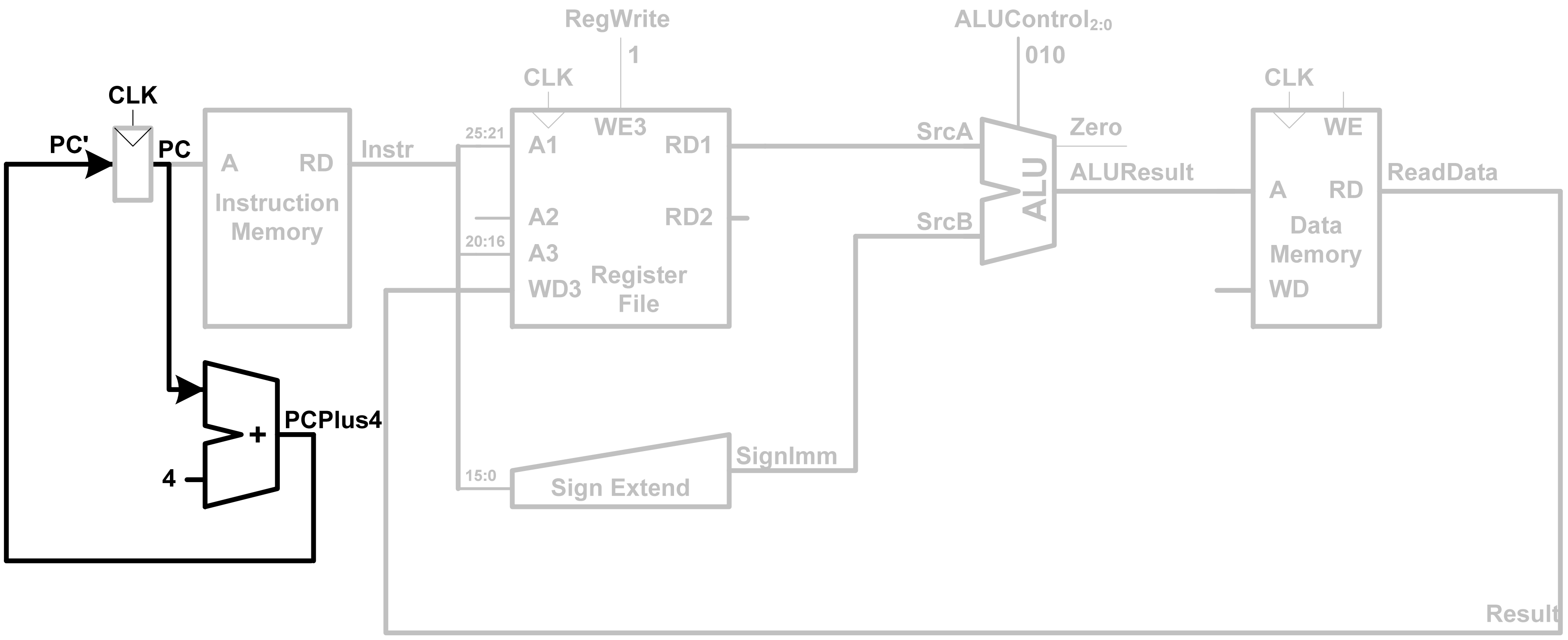
Single-Cycle Datapath Trace for `lw` /5

- **Step 5. Read data from Data Memory and write it back to Register File:**



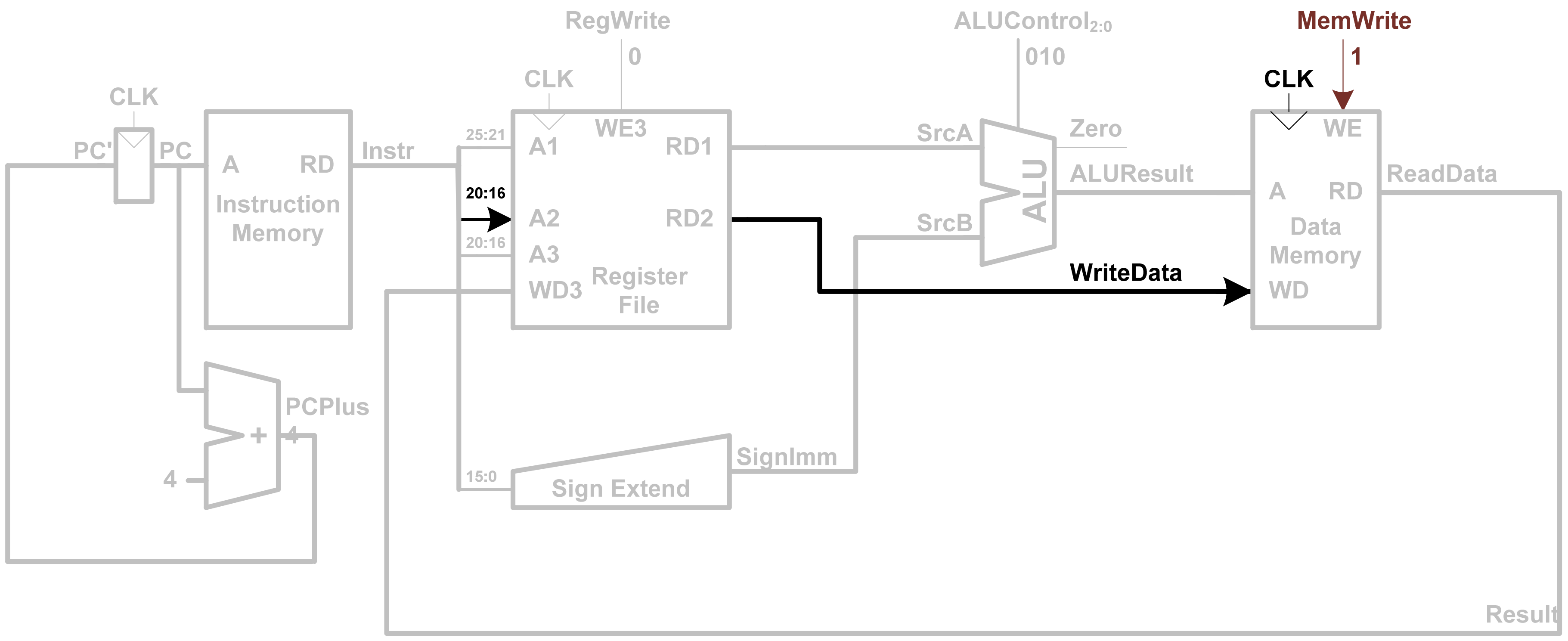
Single-Cycle Datapath Trace for `lw` /6

- **Step 6. Determine the address of next instruction:**



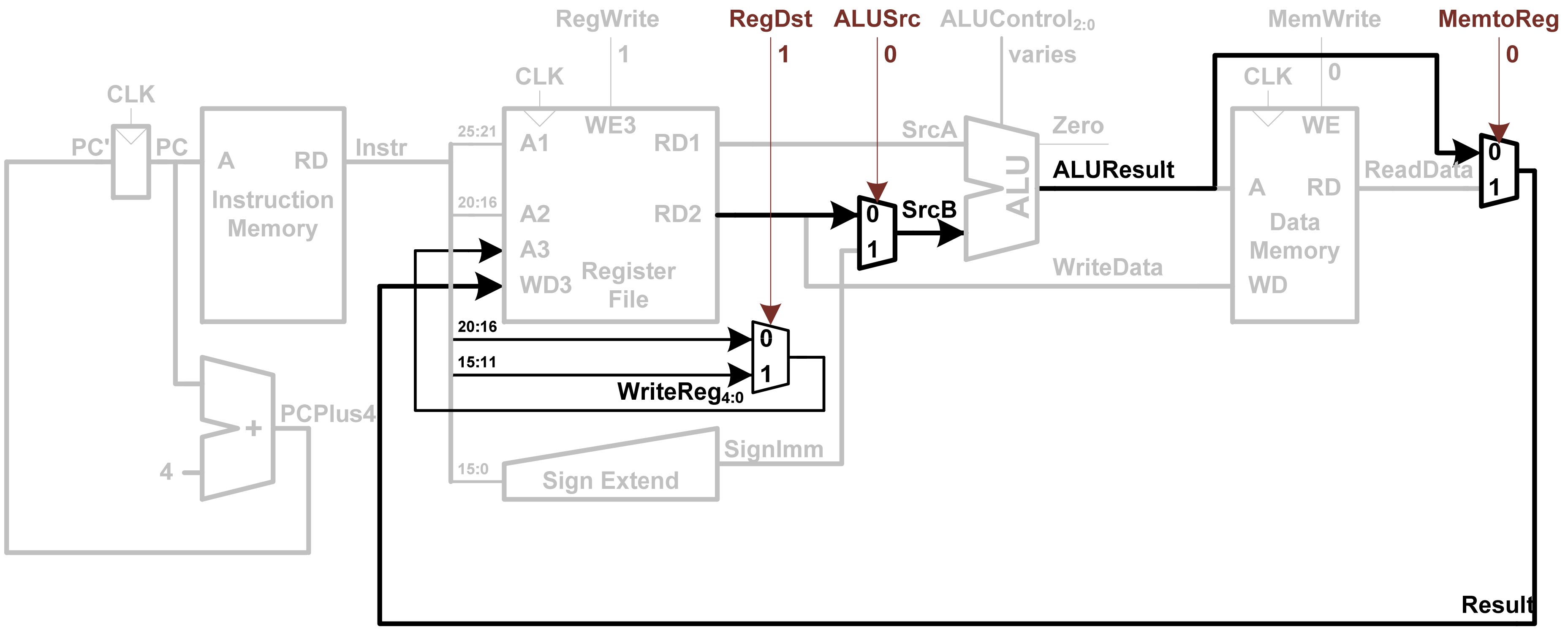
Single-Cycle Datapath Trace for `sw`

- **Steps:**
  - Compute the address the same as for `lw`
  - Write data in `rt` to memory



Single-Cycle Datapath Trace for `R-type`

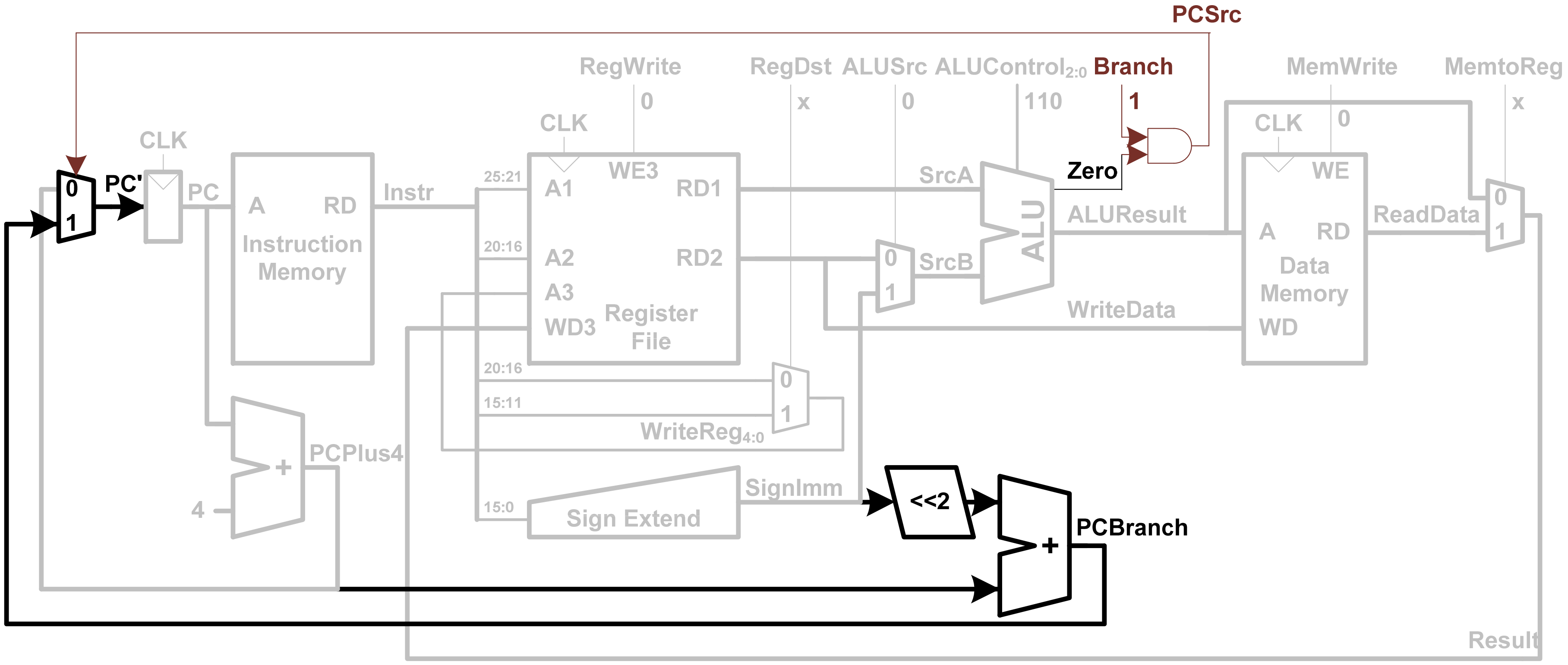
- **Steps:**
  - Read from `rs` and `rt`
  - Write ALUResult to register file
  - Write to `rd` (instead of `rt`)



Single-Cycle Datapath Trace for `beq`

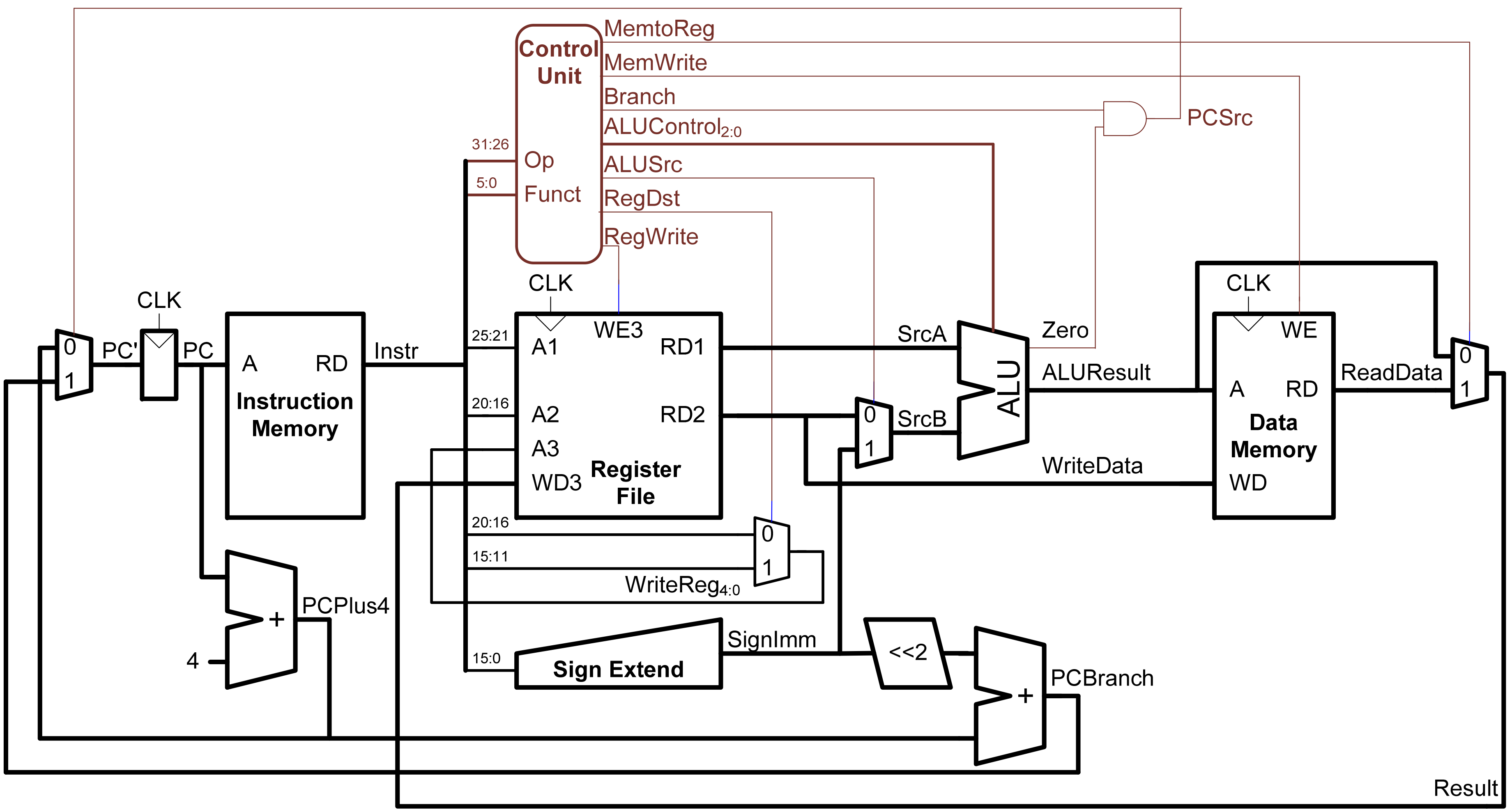
- **Steps:** `beq $s0, $s1, target`
  - Determine whether values in `rs` and `rt` are equal
    - More on J-Type instructions in the coming lectures
  - Calculate branch target address (BTA):

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



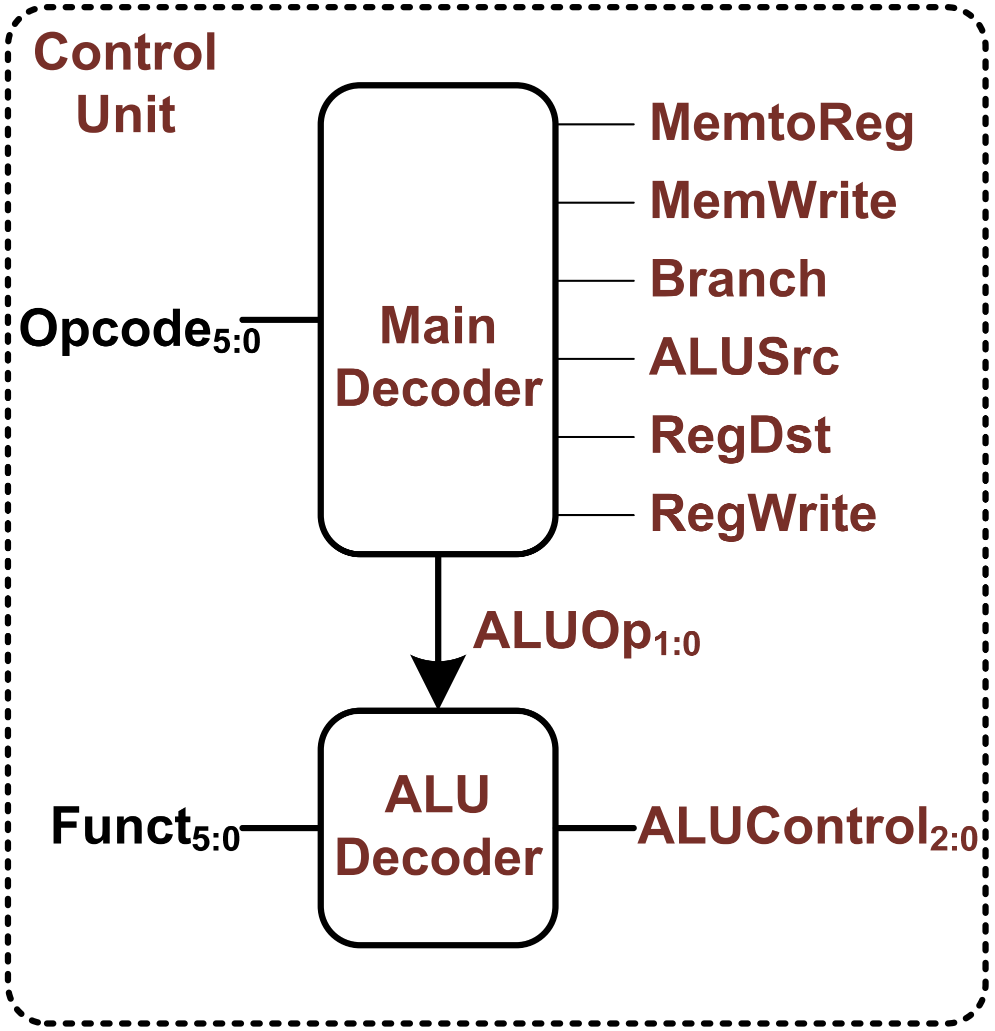
Single-Cycle Processor with Control Unit /1

- **Control Unit Added:**



Single-Cycle Processor with Control Unit /2

- **Control Unit as a factored FSM:**



Food for Thought

- **Download and Read Assignment #2 Specifications**
- **Read:**
  - Chapter 5 of the course textbook
    - Review the material discussed in the lecture notes in more detail
  - (Optional) Chapter 6 and 7 of the Harris and Harris textbook

Midterm Overview /1

- **Midterm will cover:**
  - Lecture Notes #1 to #5
  - Related Food for Thought readings
    - Review Assignment #1 and #2
  - Review Course Notes up to Chapter 4, Section 4.5 (inclusive)
    - No calculators are allowed but we will provide a table with fraction values for FP calculations
- **Main Topics:**
  - Performance measurements
  - Combinational logic circuitry
  - Simplification of Boolean equations
  - **Karnaugh maps**
  - Timing and glitches
  - …

Midterm Overview /2

- **Main Topics Continued:**
  - Sequential logic circuitry
  - **Finite state machines**
  - Two’s complement representation of integers
  - Shift/AND/OR/NOT operations
  - Ripple-carry adder and CLA adder
  - **Arithmetic Logic Unit and its components**
  - **Binary representation of numbers with fractions**
  - Floating-point addition and multiplication
  - Single-cycle processor (trace R-type and I-type instructions)