More on Single-Cycle Processor
Multi-Cycle Processor

Dr. Igor Ivkovic

iivkovic@uwaterloo.ca

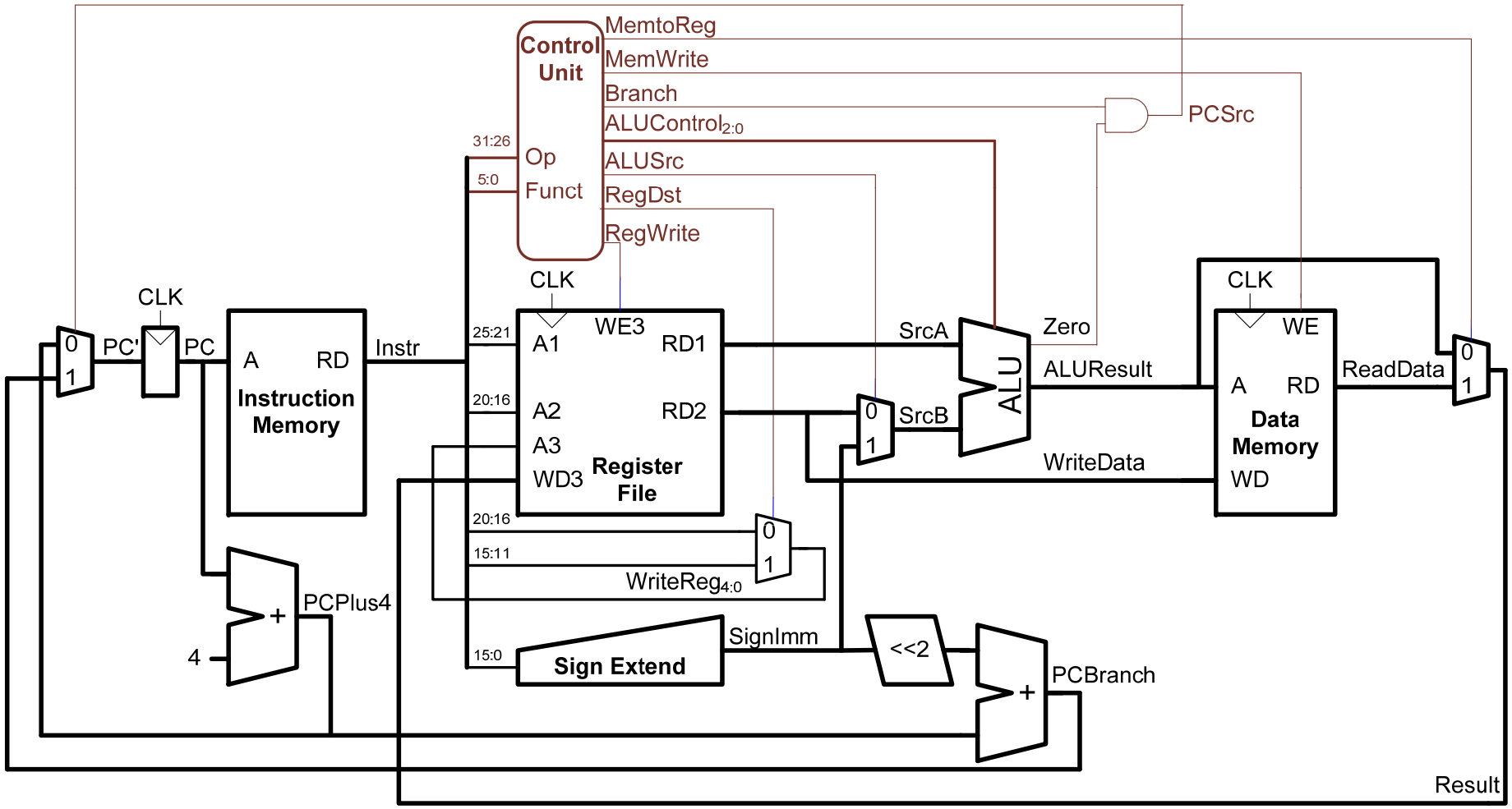
[with material from “Computer Organization and Design” by Patterson and Hennessy, and “Digital Design and Computer Architecture” by Harris and Harris, both published by Morgan Kaufmann]

Objectives

- More on Single-Cycle Processor
- Control Unit Decoder
- Multi-Cycle Processor

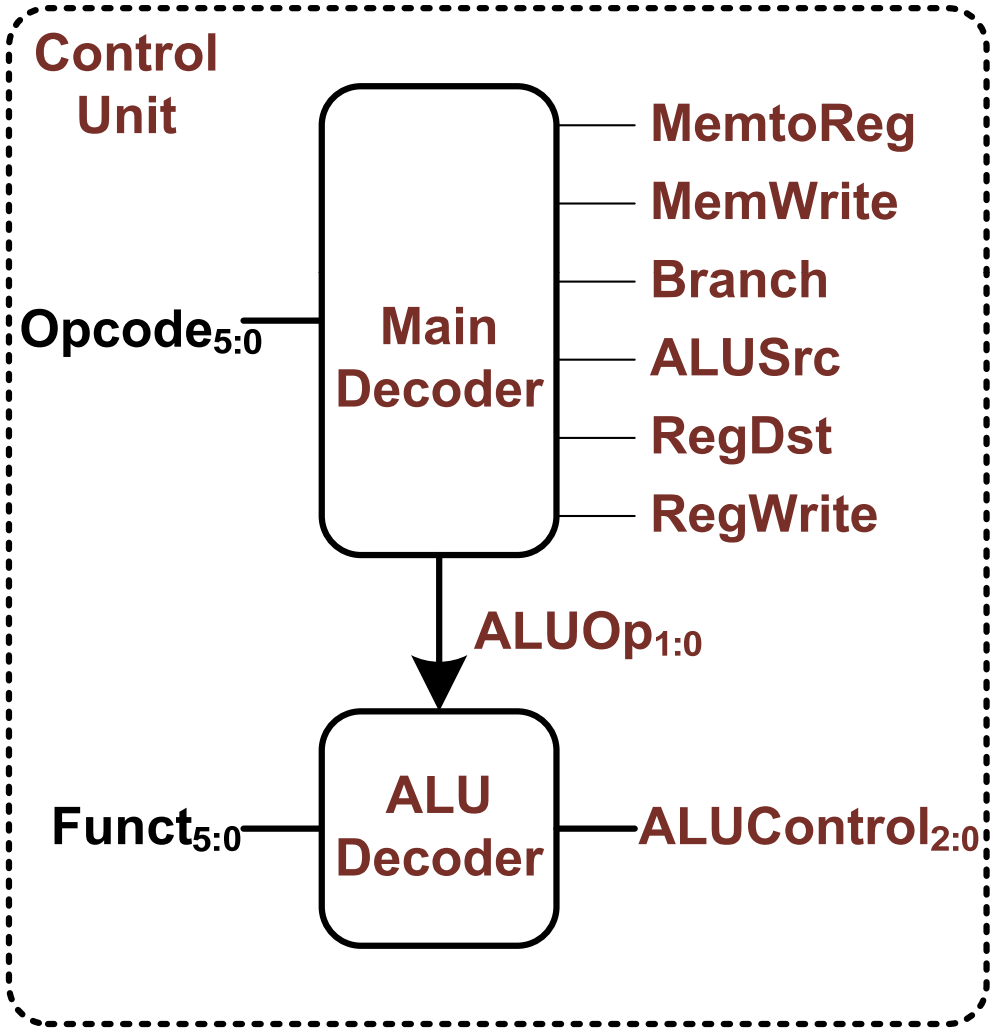
More on Single-Cycle Processor /1

- **Control Unit Added:**



More on Single-Cycle Processor /2

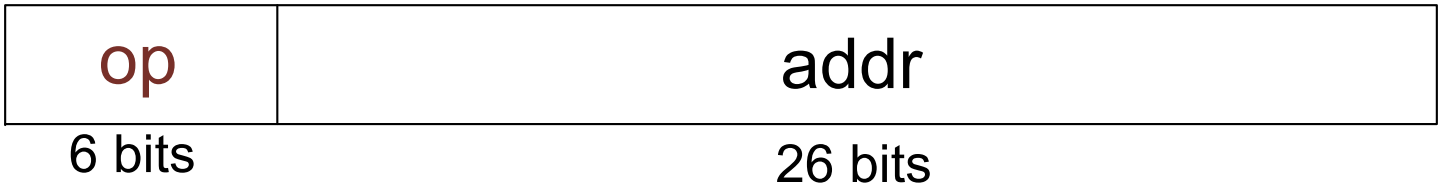
- **Control Unit as a factored FSM:**



More on Single-Cycle Processor /3

- **J-Type (Jump-Type) Instruction:**
  - Used for jump instructions
    - beq and bne (branch if not equal) are both I-type instructions
  - **j (jump) is a J-type instruction**
    - jr (jump register) is a R-type instruction
  - **For j, update PC to the concatenation of**
    - Top 4 bits of old PC
    - 26-bit jump address
    - 00
- **3 register operands:**
  - `op`: the opcode
  - `addr`: 26-bit address operand

J-Type



More on Single-Cycle Processor /4

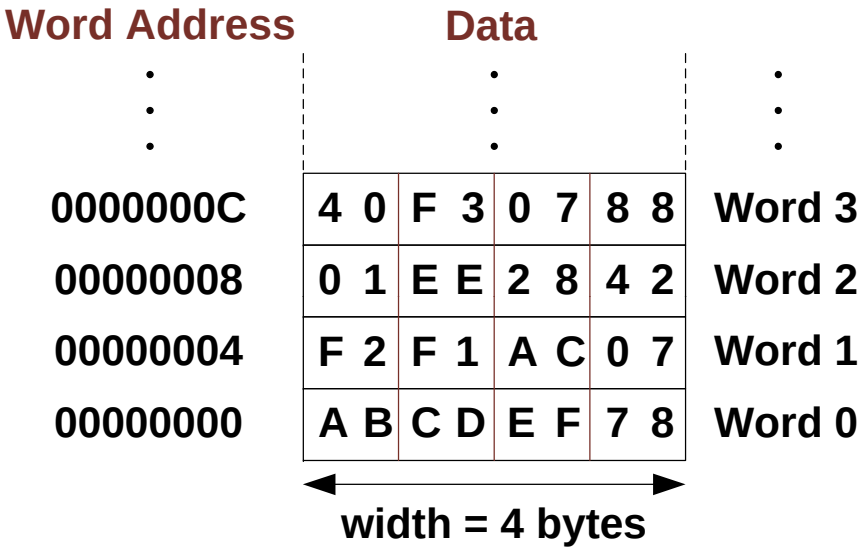
- **J-Type Instruction MIPS Example:**

```
addi $s0, $0, 4       # $s0 = 4
addi $s1, $0, 1       # $s1 = 1
j    target           # jump to target
# load label address using la operation
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
```

More on Single-Cycle Processor /5

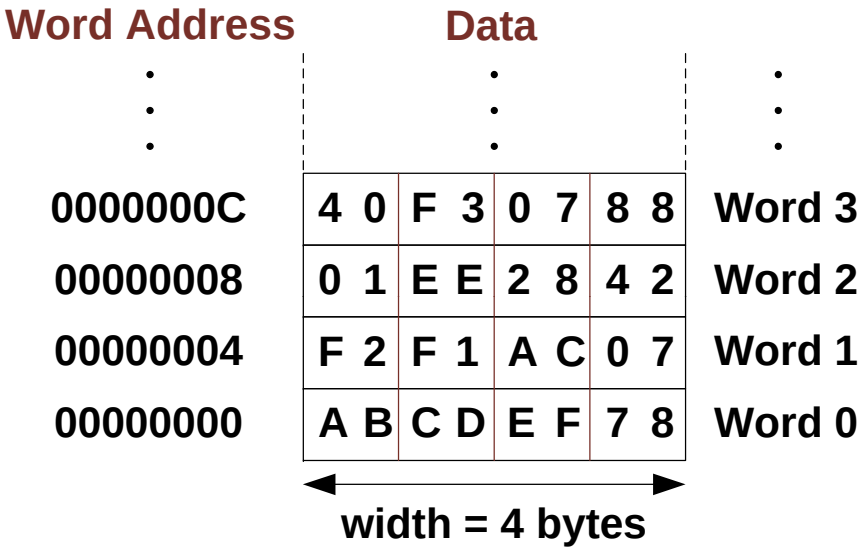
- **Each 32-bit data word has a unique address:**



- **Reading Byte-Addressable Memory:**
  - 32-bit word = 4 bytes, so word address increments by 4
  - The address of a memory word must now be multiplied by 4
    - The address of memory word 2 is 2 × 4 = 8, word 10 is 10 × 4 = 40
  - **MIPS is byte-addressed, not word-addressed**
  - **Example:** `lw $s0, 4($0)`, **Result:** $s0 = 0xF2F1AC07

More on Single-Cycle Processor /6

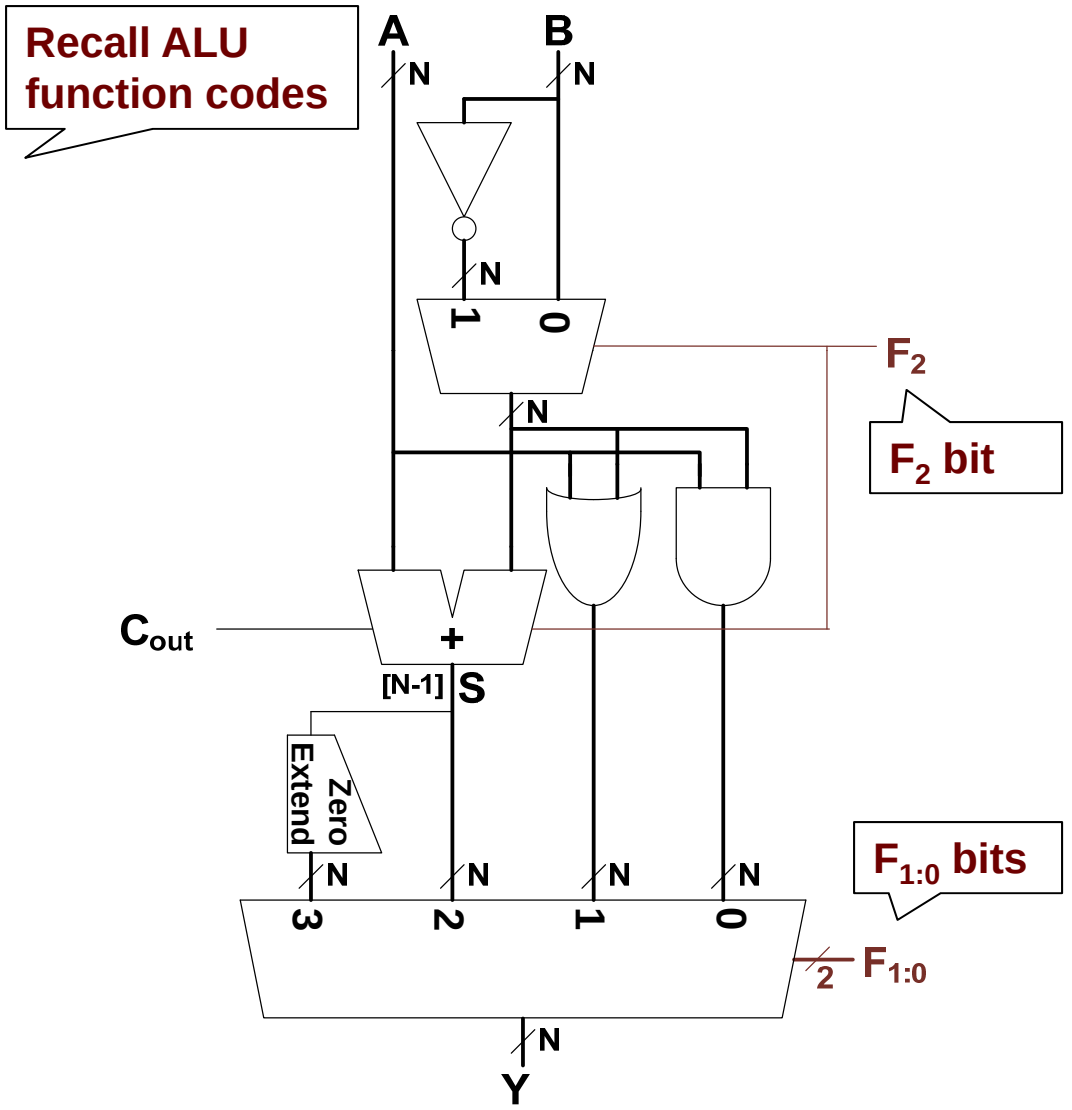
- **Each 32-bit data word has a unique address:**



- **Writing Byte-Addressable Memory:**
  - 32-bit word = 4 bytes, so word address increments by 4
  - **Example:** `sw $s0,44($0)`, **Result:** address 44 holds $s0

More on Single-Cycle Processor /7

$F_{2:0}$	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

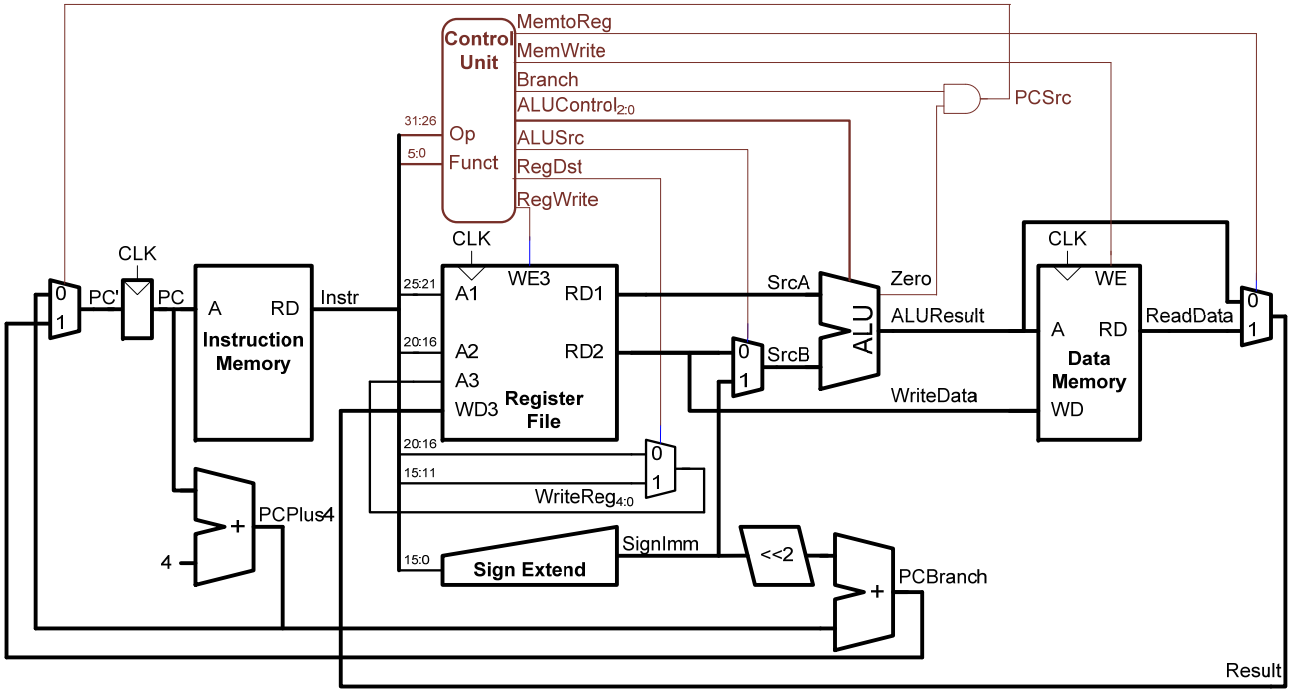
$ALUOp_{1:0}$	Meaning
00	Add
01	Subtract
10	Look at funct
11	Not used

$ALUOp_{1:0}$	Funct	$ALUControl_{2:0}$
00	X	010 (Add)
01	X	110 (Subtract)
10	100000 (add)	010 (Add)
10	100010 (sub)	110 (Subtract)
10	100100 (and)	000 (And)
10	100101 (or)	001 (Or)
10	101010 (slt)	111 (SLT)

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

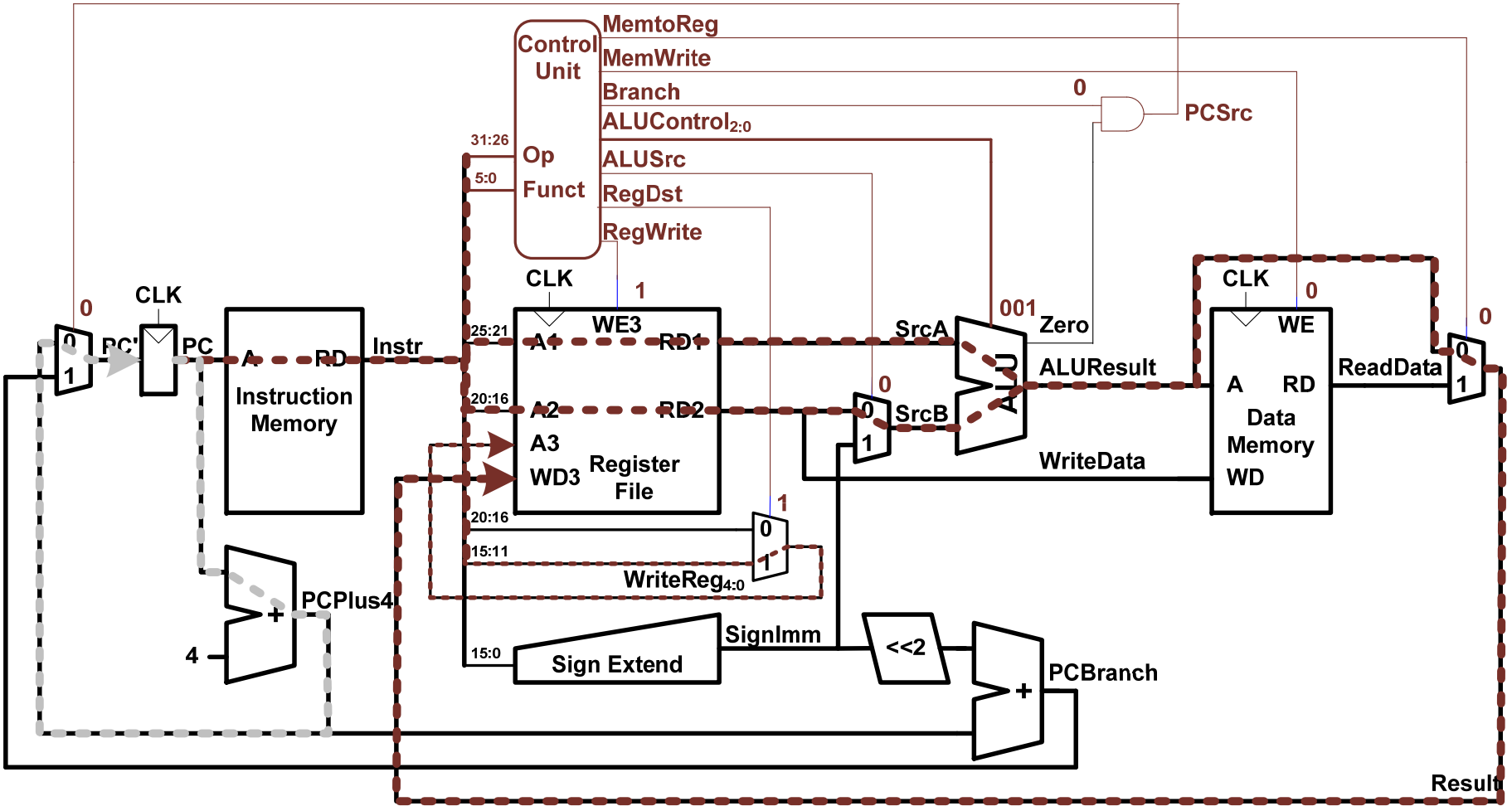
Control Unit: Main Decoder /1

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**



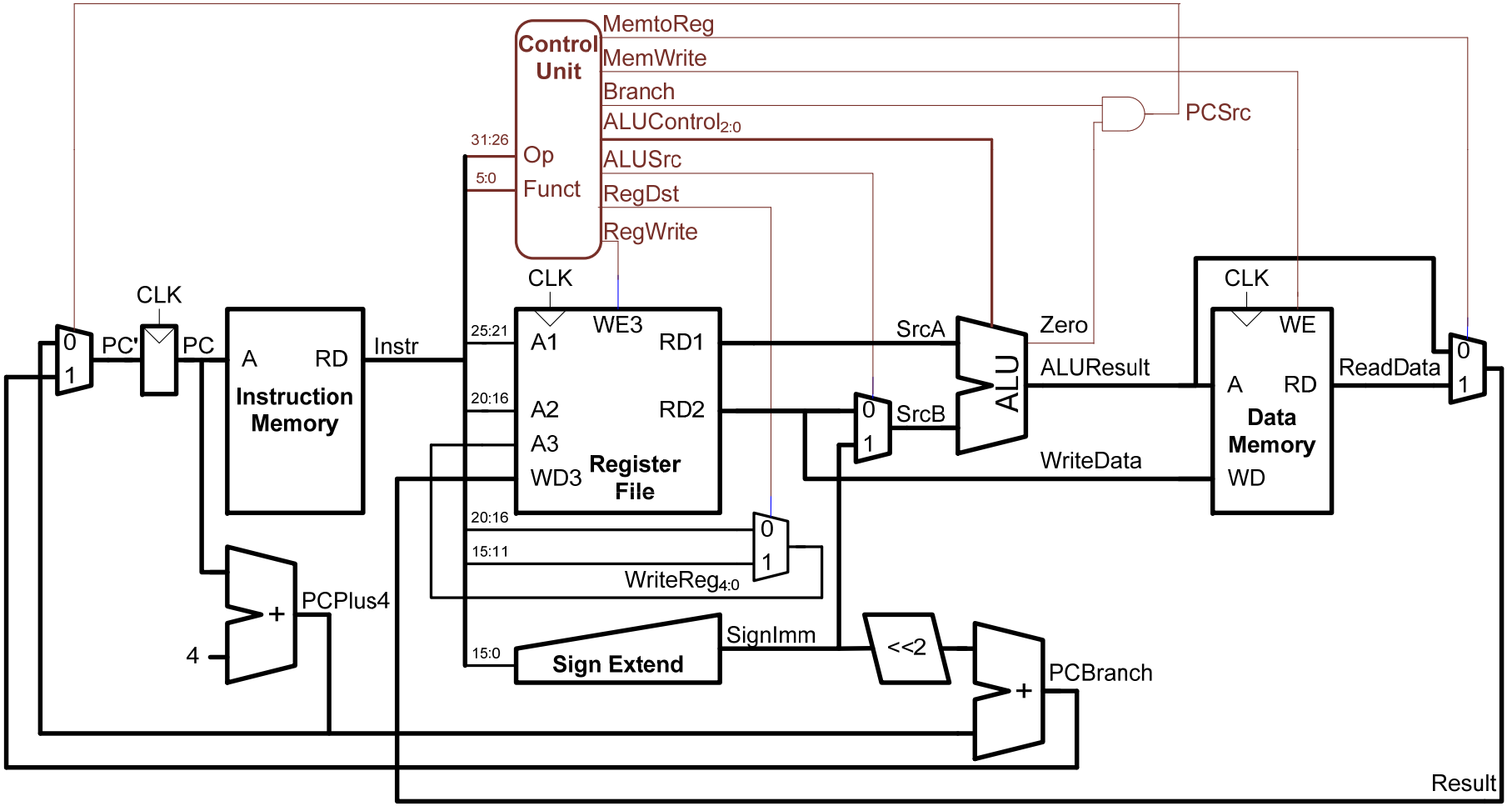
Control Unit: Main Decoder /2

- **Single-Cycle Datapath:** `or $s0, $s1, $s2`



Control Unit: Main Decoder /3

- **Single-Cycle Datapath:** `addi $s0, $s1, 5`
  - No change to datapath



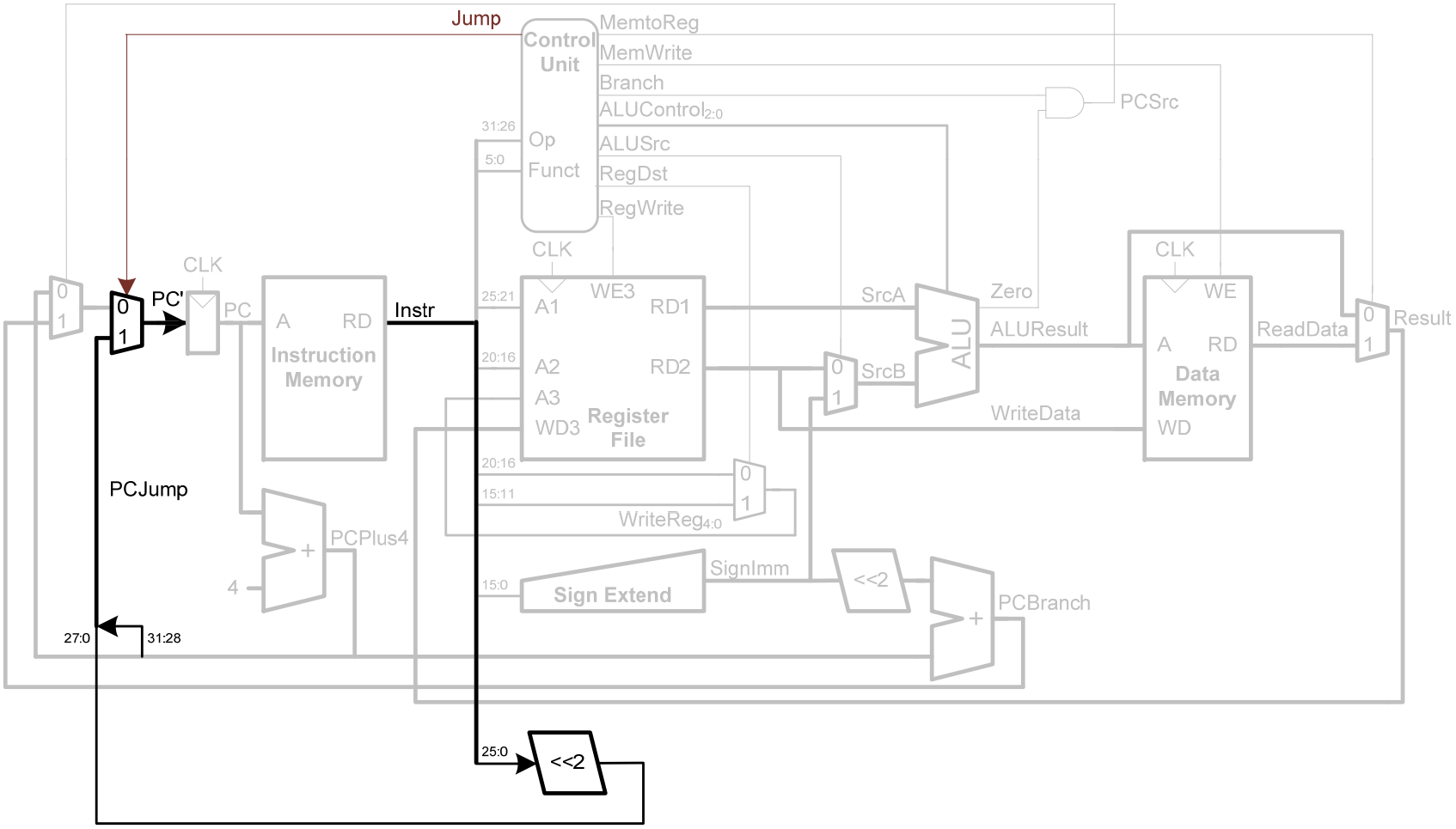
Control Unit: Main Decoder /4

- **Single-Cycle Datapath:** `addi $s0, $s1, 5`

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 /5

- **Single-Cycle Datapath Extended Functionality: `j`**
  - Additional circuitry



Control Unit: Main Decoder /6

- **Single-Cycle Datapath Extended Functionality: `j`**

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

More on Single-Cycle Processor /8

- **Performance of Single Cycle Machines**
  - Suppose memory unit requires 200ps (picoseconds), ALU 100ps, register file 50ps, and no delay on other units
  - Furthermore, let jump take 200ps, branch take 350ps, R-format instructions 400ps, store 550ps, and load 600 ps
  - **As the result, the clock period must be increased to 600 ps (the maximum length of an instruction) or more to ensure that each instruction can be performed in a single cycle**
    - Critical path typically represented by the `lw` path
  - Even worse when floating-point instructions are implemented
  - Better idea: Use multi-cycle processor implementation

More on Single-Cycle Processor /9

- Typically design datapath for all instructions all together
  - **If a new instruction is needed after the design, we will need to modify the datapath**
- **For example, perform the following steps:**
  1. Determine what datapath is needed for the new command
  2. Check if any components in the current datapath can be used
  3. Integrate components of the new datapath into the existing datapath, most likely requiring the use of MUXes
  4. Add new control signals to the control unit
  5. Adjust old control signals to account for new command

Multi-Cycle Processor /1

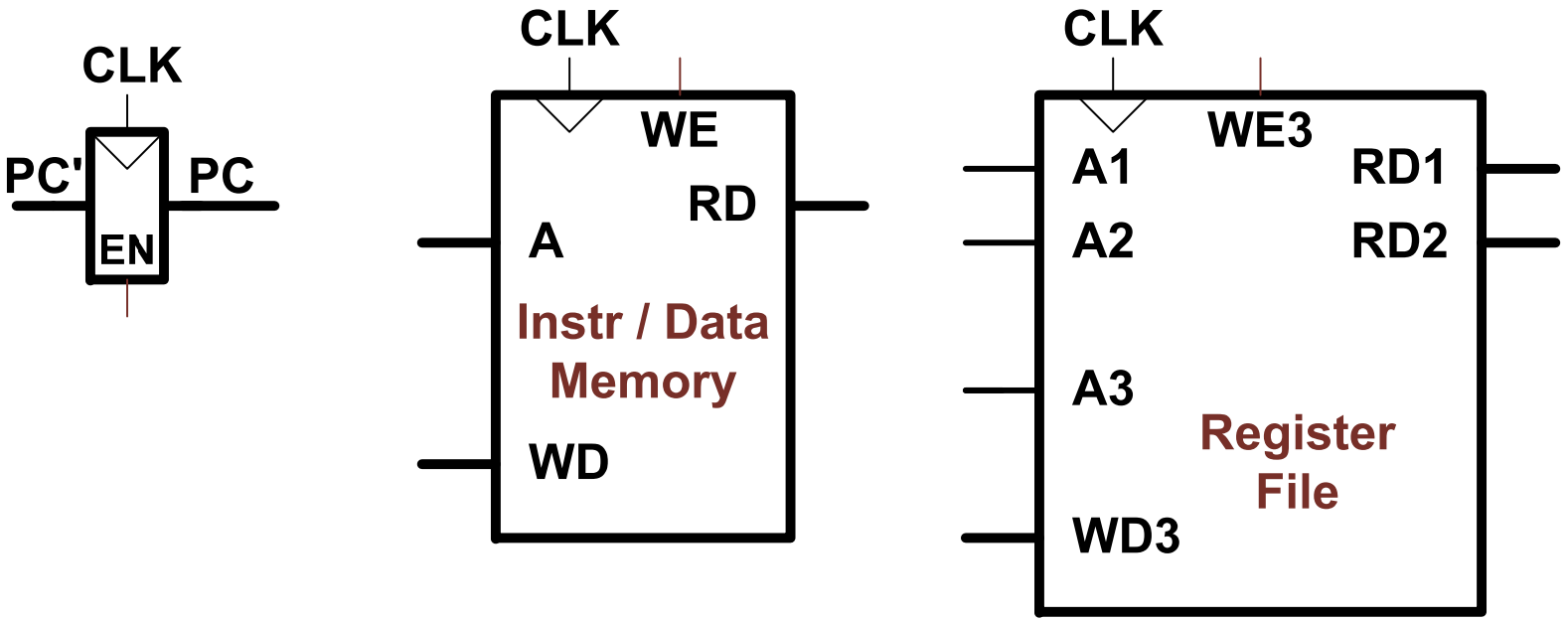
- **Single-Cycle Processor:**
  - Simple design, but cycle time (T_c) limited by the longest instruction (`lw`)
  - The design also requires multiple adders/ALUs, which can be expensive especially if these need to be fast (e.g., CLAs)
  - Two memory units are used, but typical computer design uses single memory block to store both data and instructions
- **Multi-Cycle Processor:**
  - Break the instruction into smaller steps, such as reading or writing memory or using the ALU
  - As the result, instructions that require less processing steps can perform faster
  - Allow higher clock speeds since cycle time is not limited by the longest instruction

Multi-Cycle Processor /2

- **More on Multi-Cycle Processor:**
  - Expensive hardware, such as adders/ALUs, can be reused on multiple cycles
  - Use only one memory unit are used to store both data and instructions
  - An instruction is fetched from memory on the first step, but data may be read from or written to memory in later steps
  - **At end of clock cycle, all data used in subsequent cycles must be stored in state element**
  - **We assume one clock cycle can contain one memory access, a register file access, or one ALU operation**

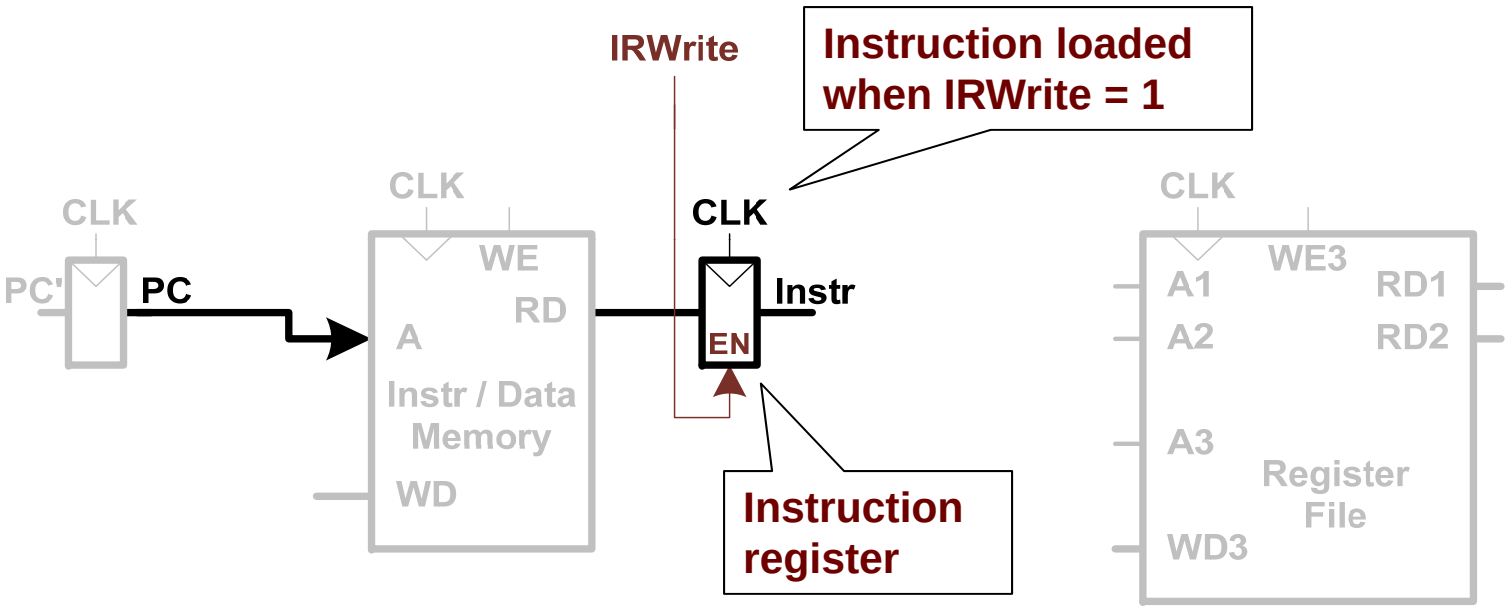
Multi-Cycle Processor /3

- **Multi-Cycle Processor Design:**
  - Use combinational and sequential circuitry for the datapath, but also add state elements to store results between steps
  - Design the control unit as a finite state machine
  - As the first step, replace instruction and data memory units with a single unified memory unit



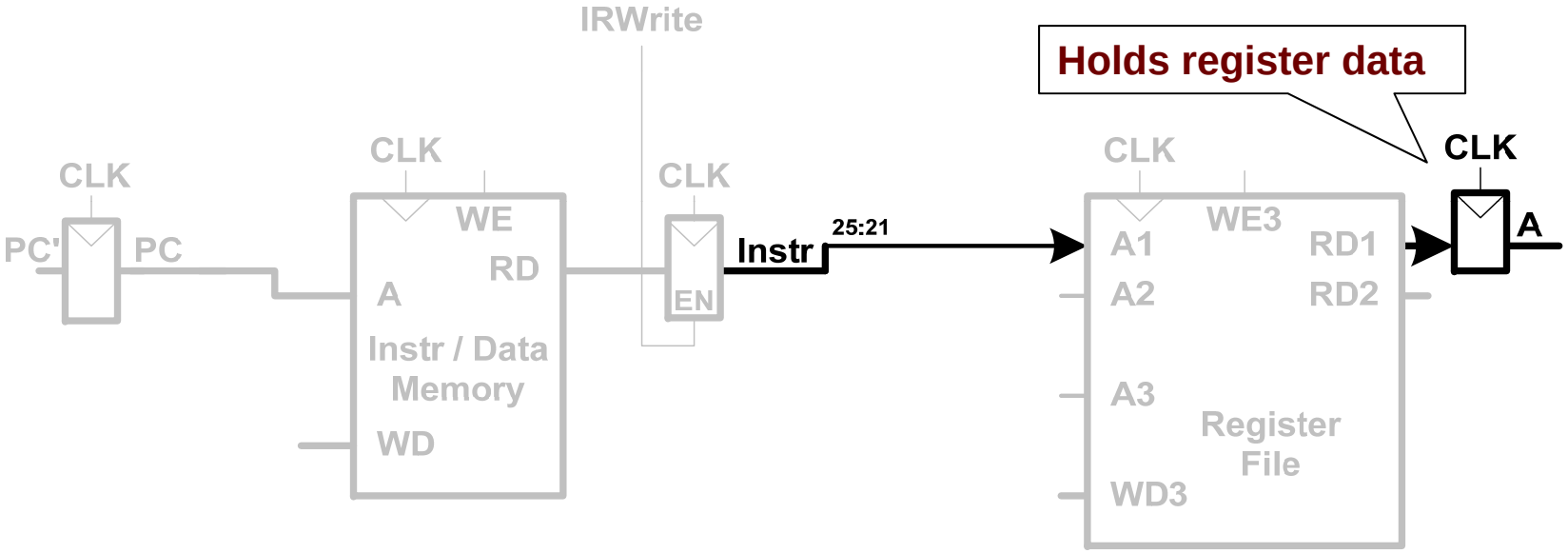
Multi-Cycle Datapath Trace for `lw` /1

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



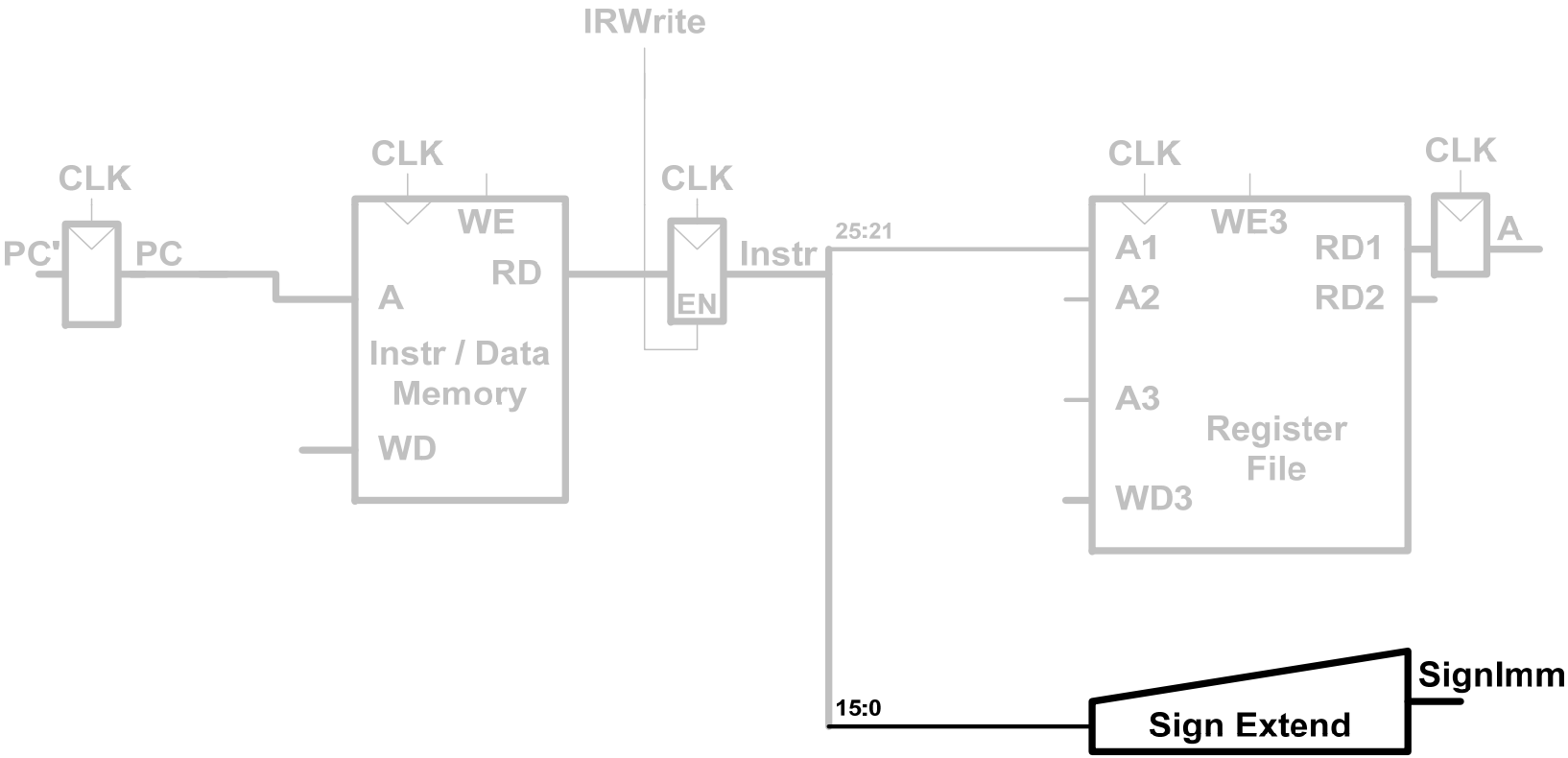
Multi-Cycle Datapath Trace for `lw` /2

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



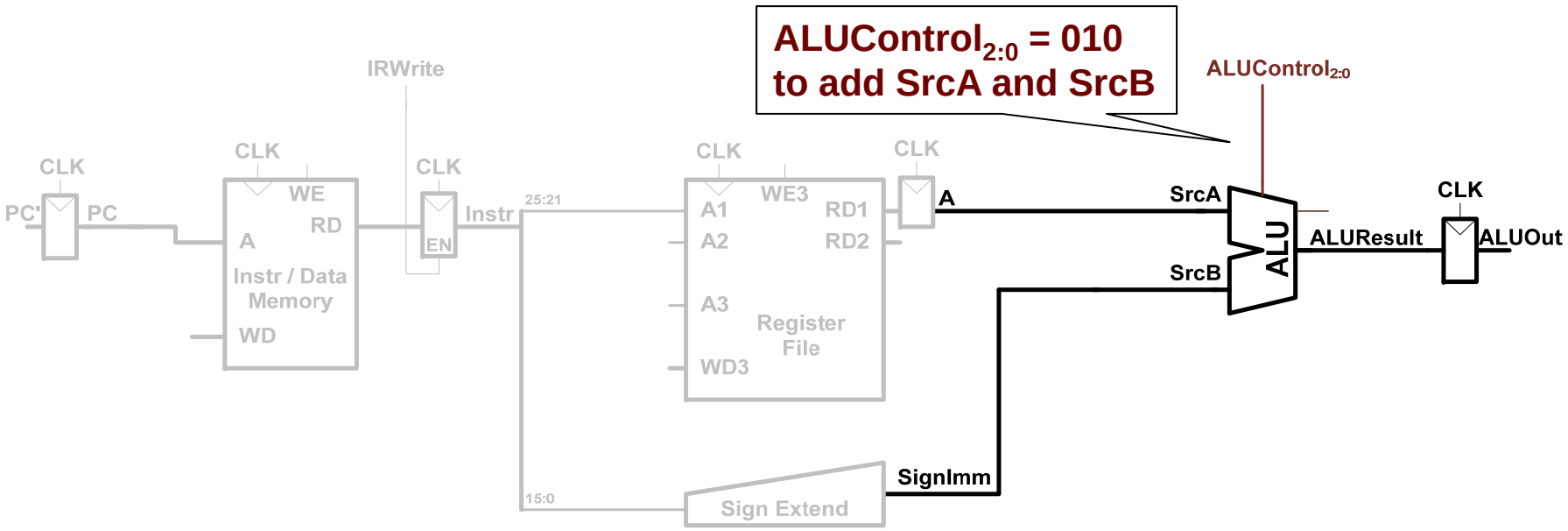
Multi-Cycle Datapath Trace for `lw` /3

- **Step 2b. Sign-extend the immediate value:**



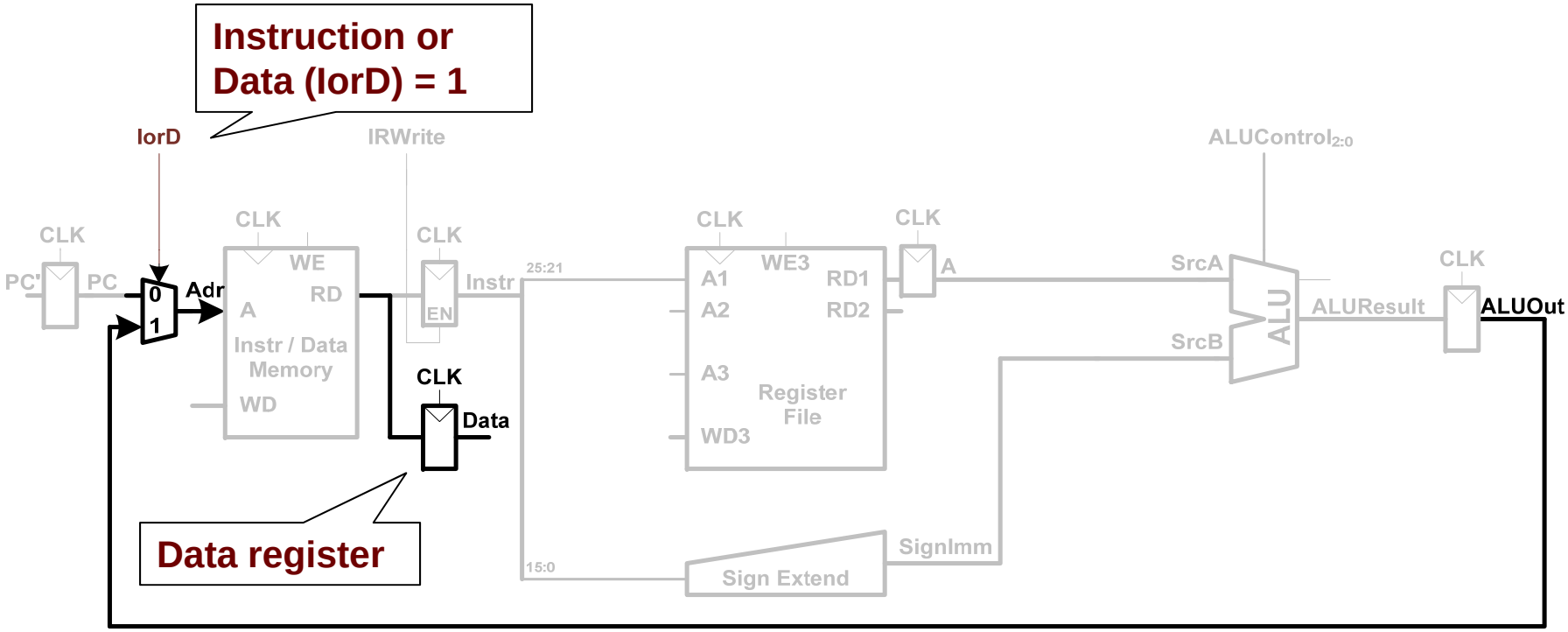
Multi-Cycle Datapath Trace for `lw` /4

- **Step 3. Compute the memory address (add srcA/B):**



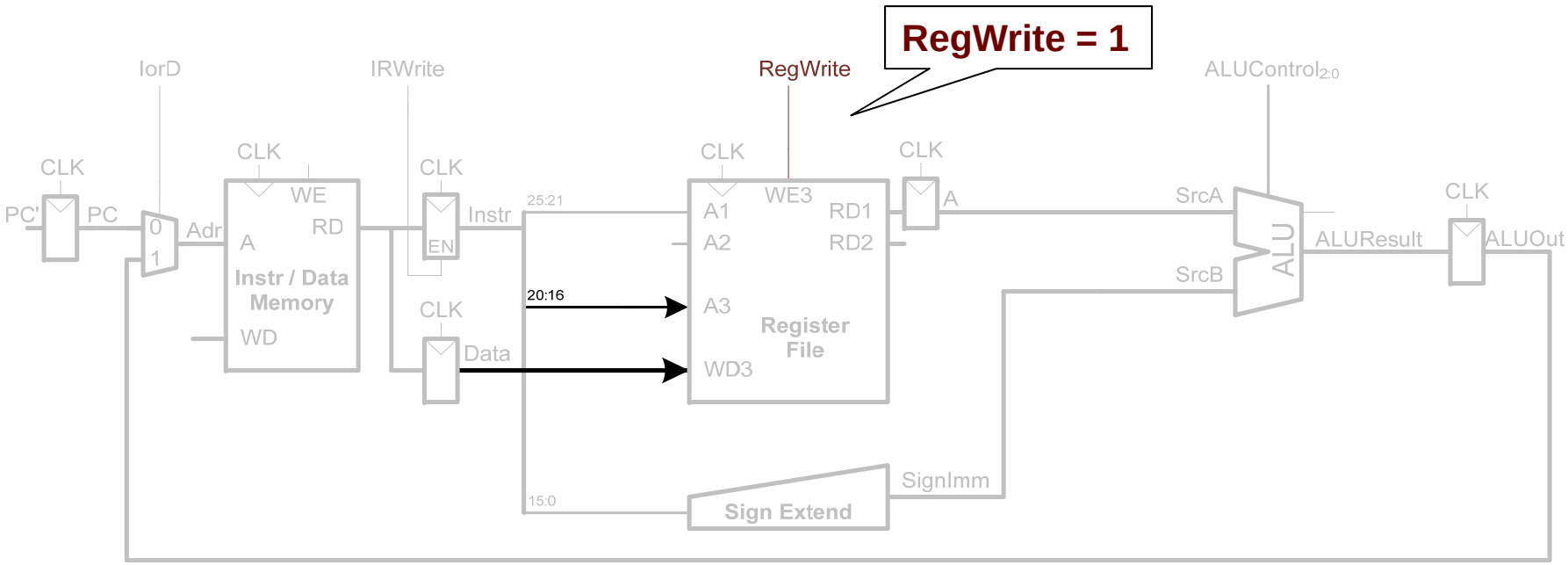
Multi-Cycle Datapath Trace for `lw` /5

- **Step 4. Read data from Memory Unit:**



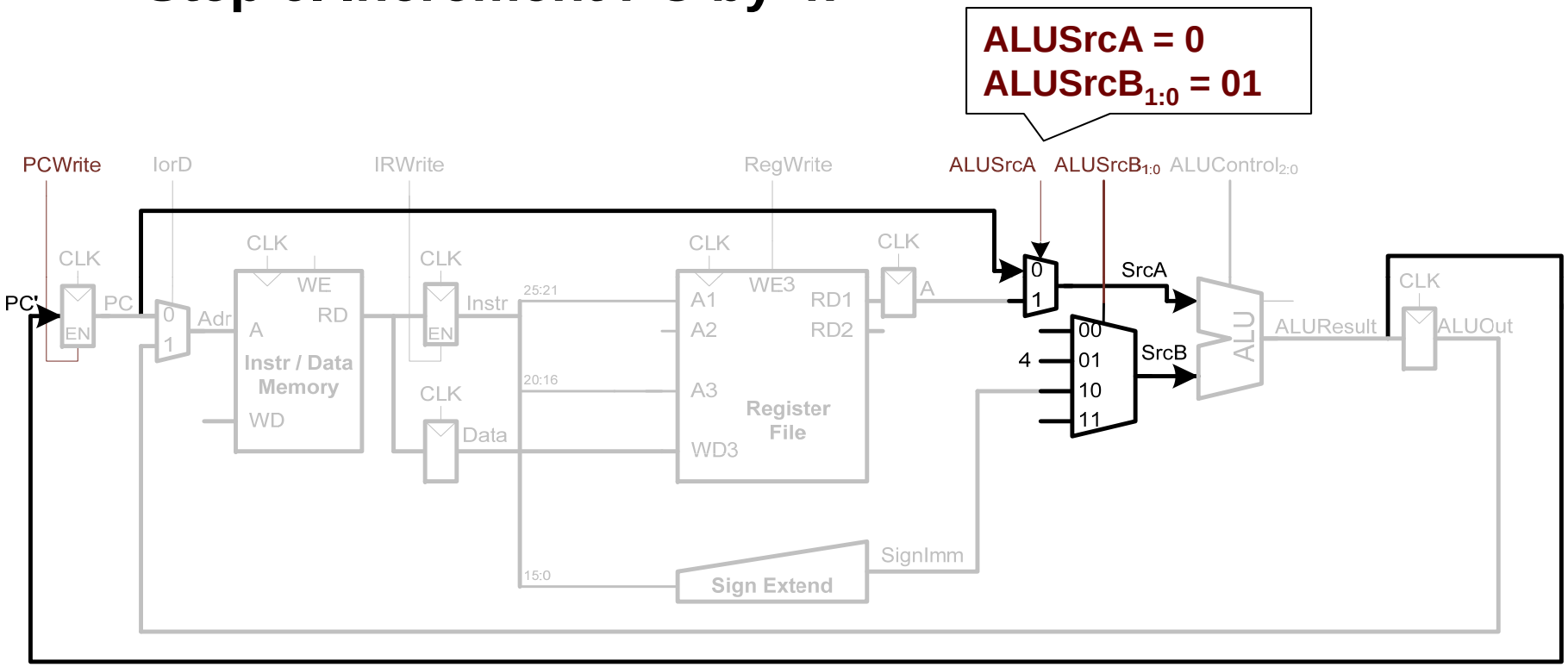
Multi-Cycle Datapath Trace for `lw` /6

- **Step 5. Write data back to Register File:**



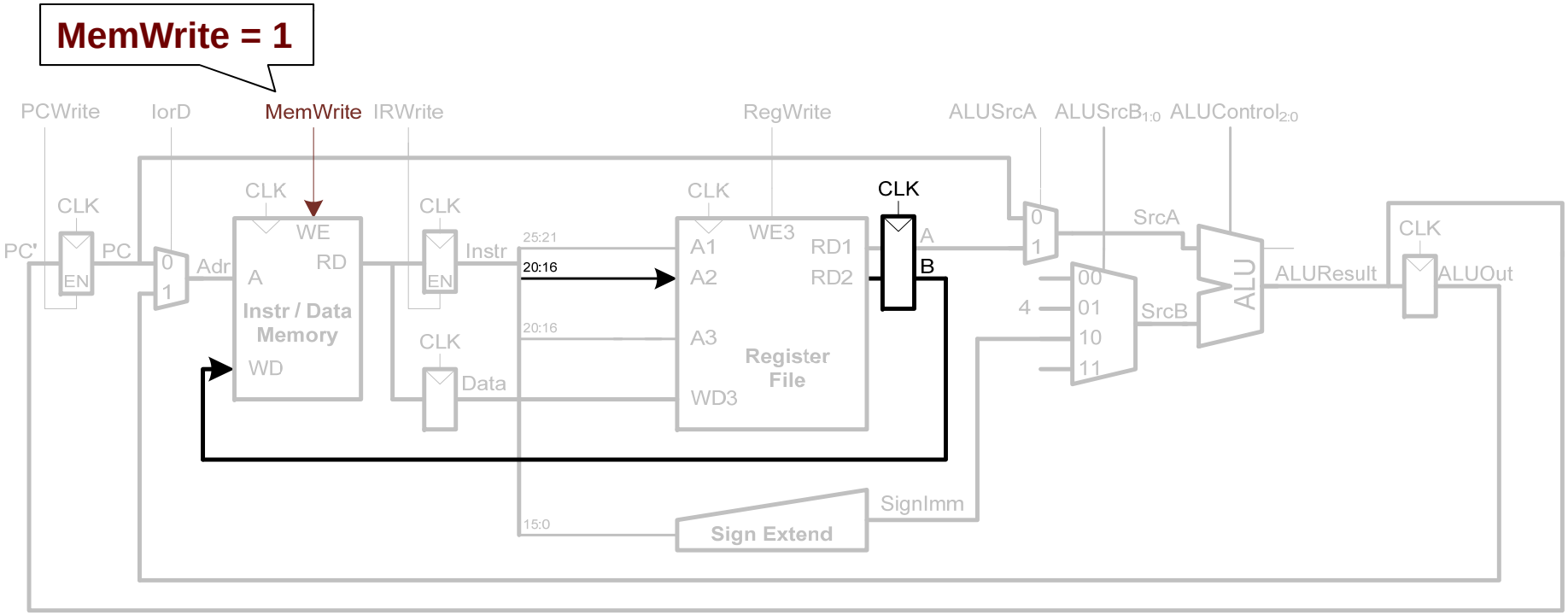
Multi-Cycle Datapath Trace for `lw` /7

- **Step 6. Increment PC by 4:**



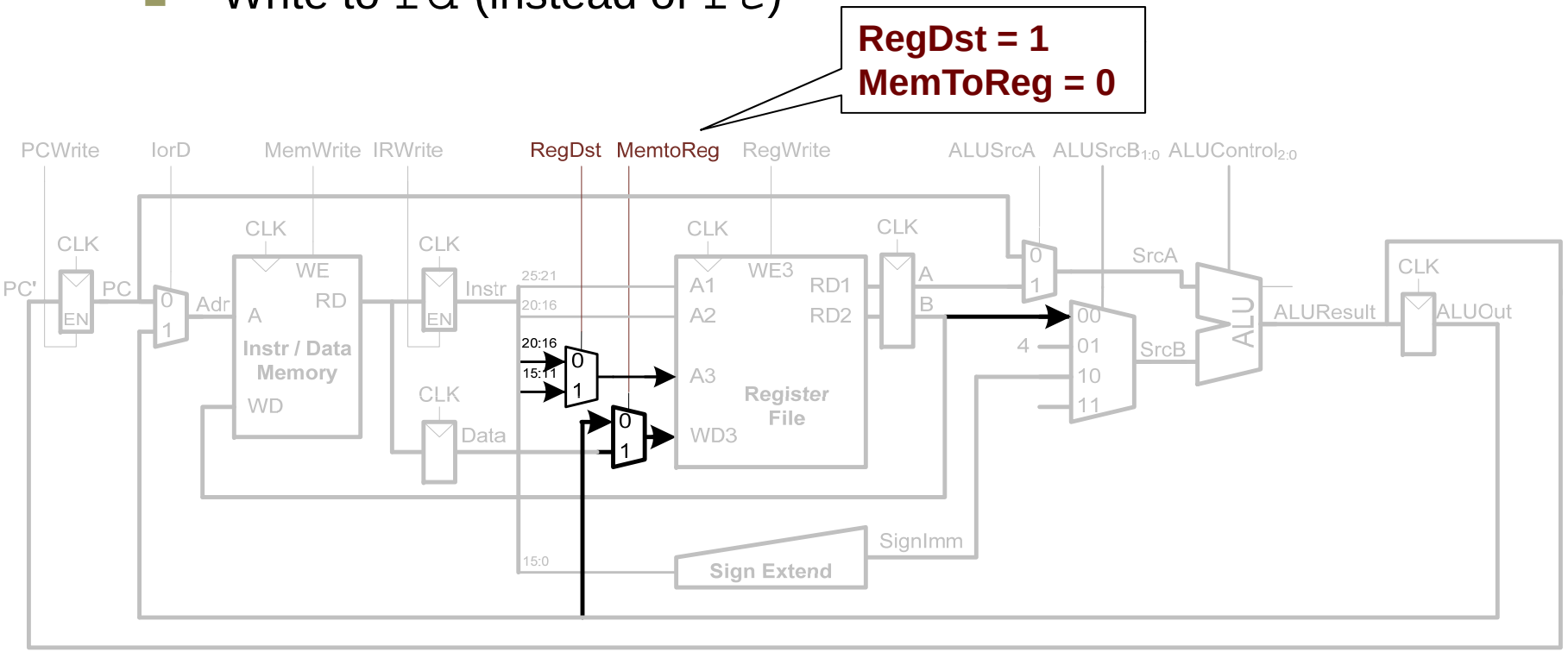
Multi-Cycle Datapath Trace for `sw`

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



Multi-Cycle Datapath Trace for `R-type`

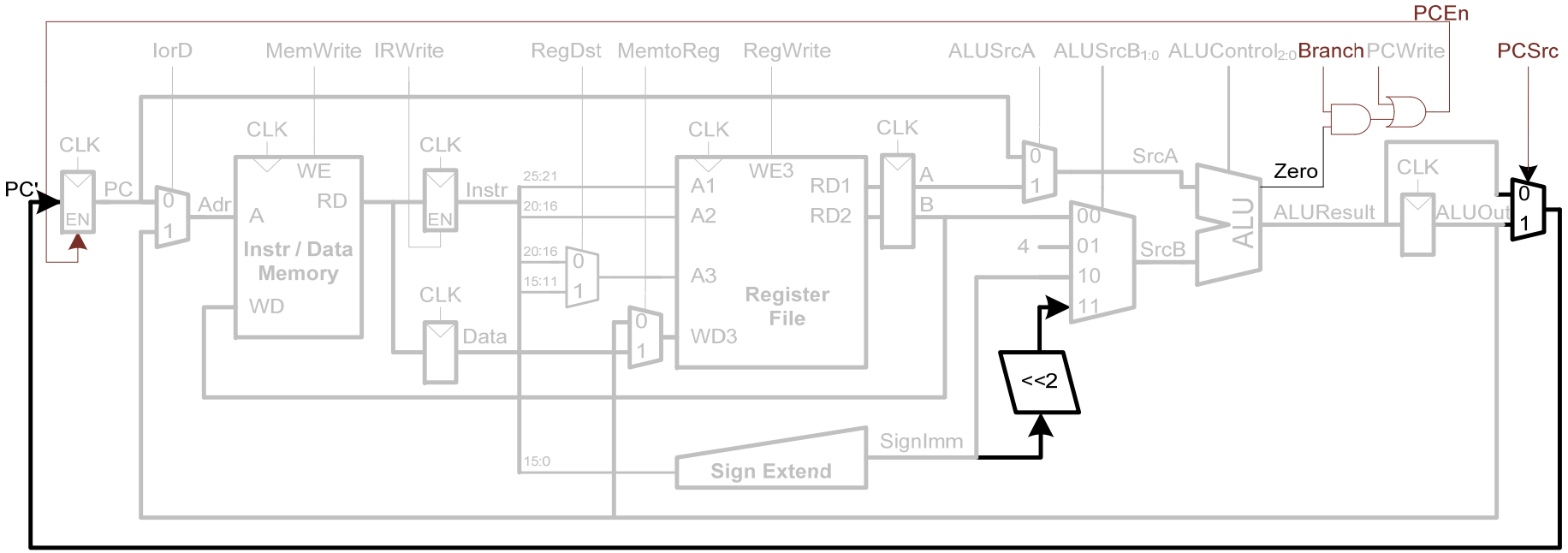
- **Steps:**
  - Read from `rs` and `rt`
  - Write `ALUResult` to register file
  - Write to `rd` (instead of `rt`)



Multi-Cycle Datapath Trace for `beq`

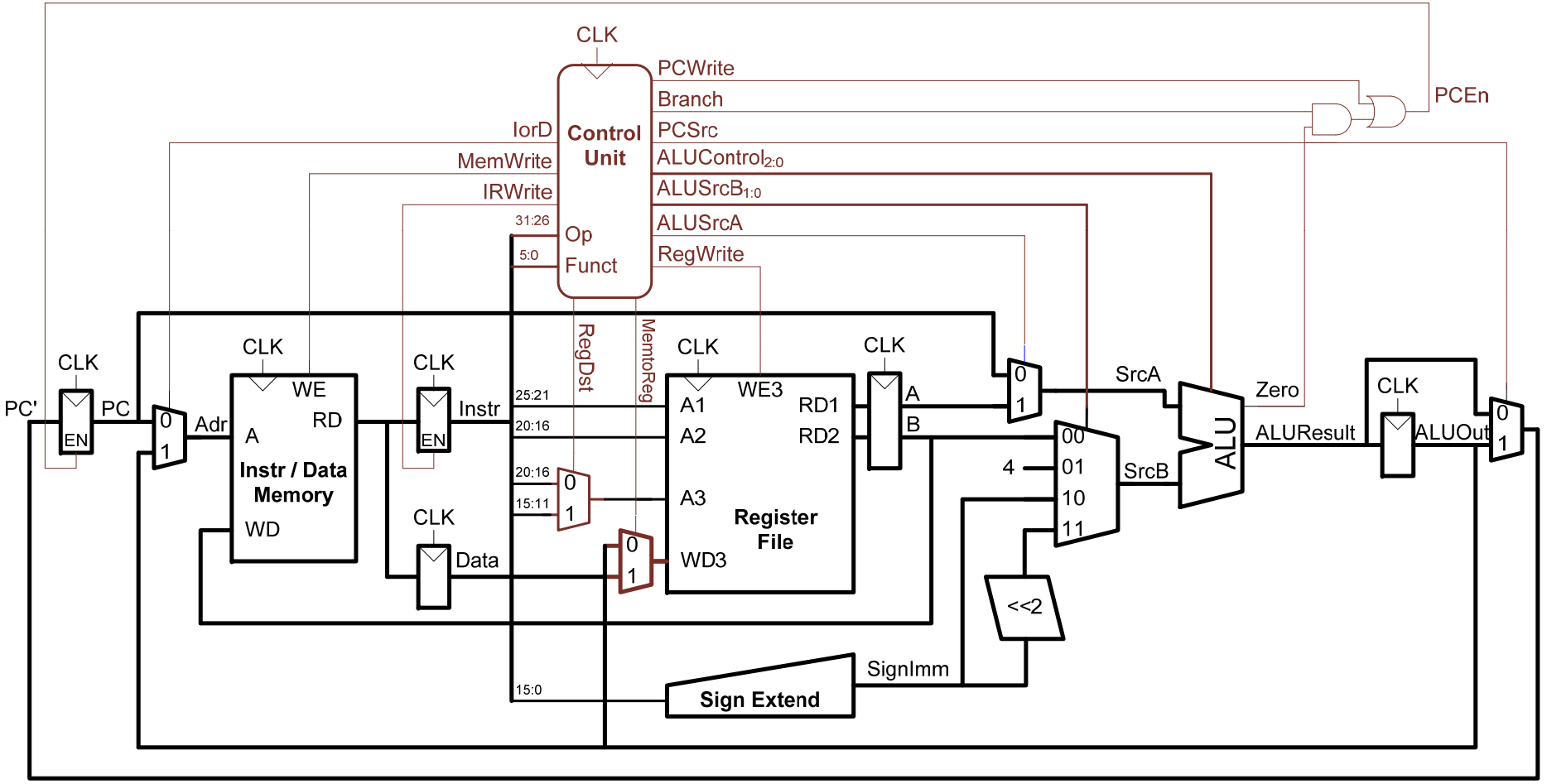
- **Steps:** `beq $s0, $s1, target`
  - Determine whether values in `rs` and `rt` are equal
  - Calculate branch target address (BTA):

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



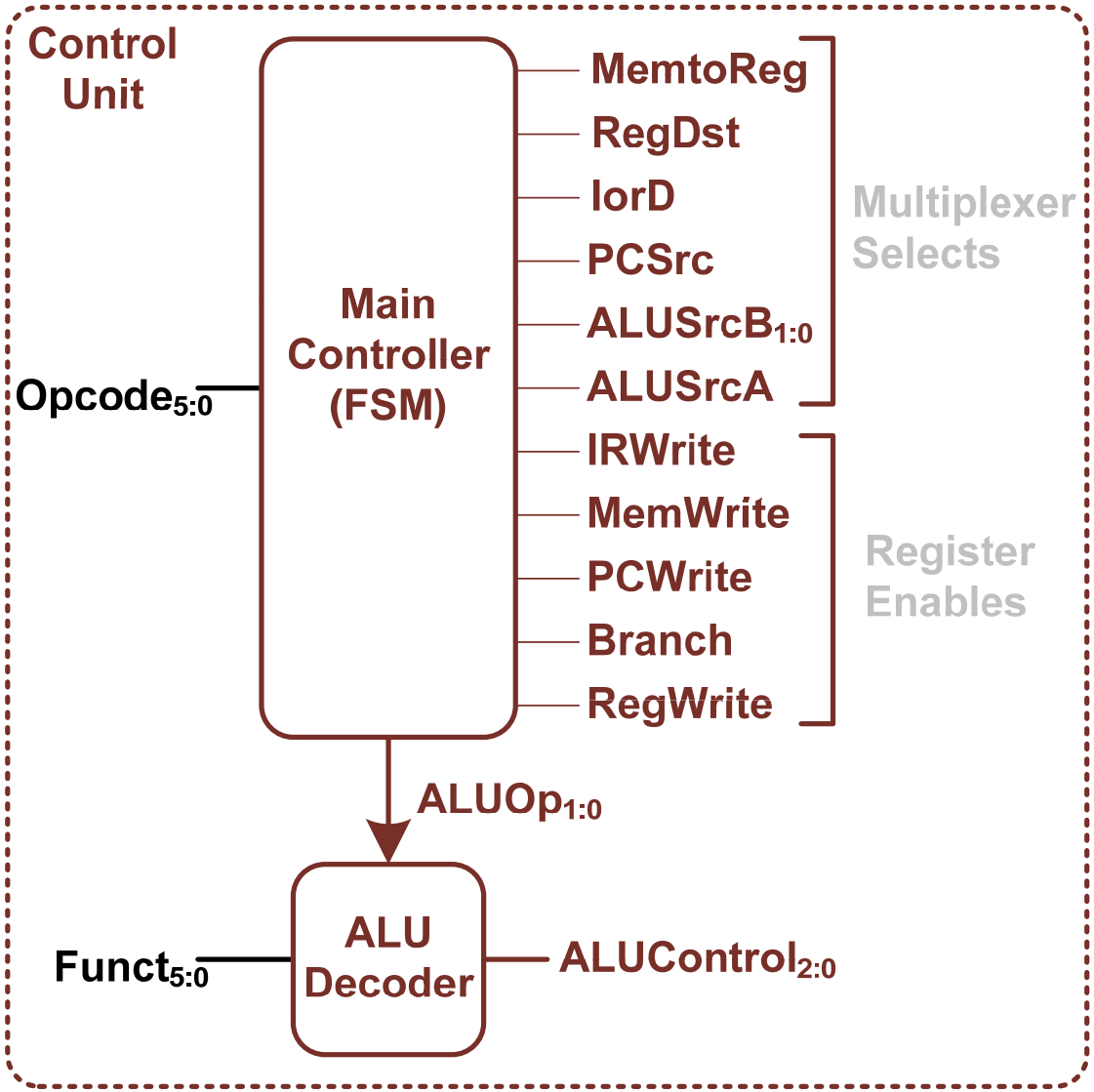
Multi-Cycle Processor with Control Unit /1

- **Control Unit Added:**



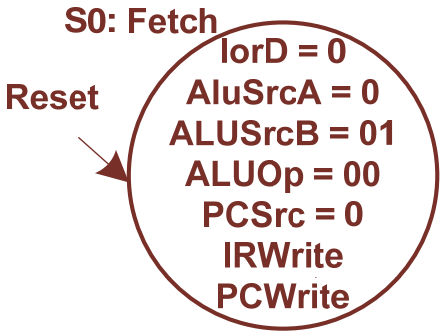
Multi-Cycle Processor with Control Unit /2

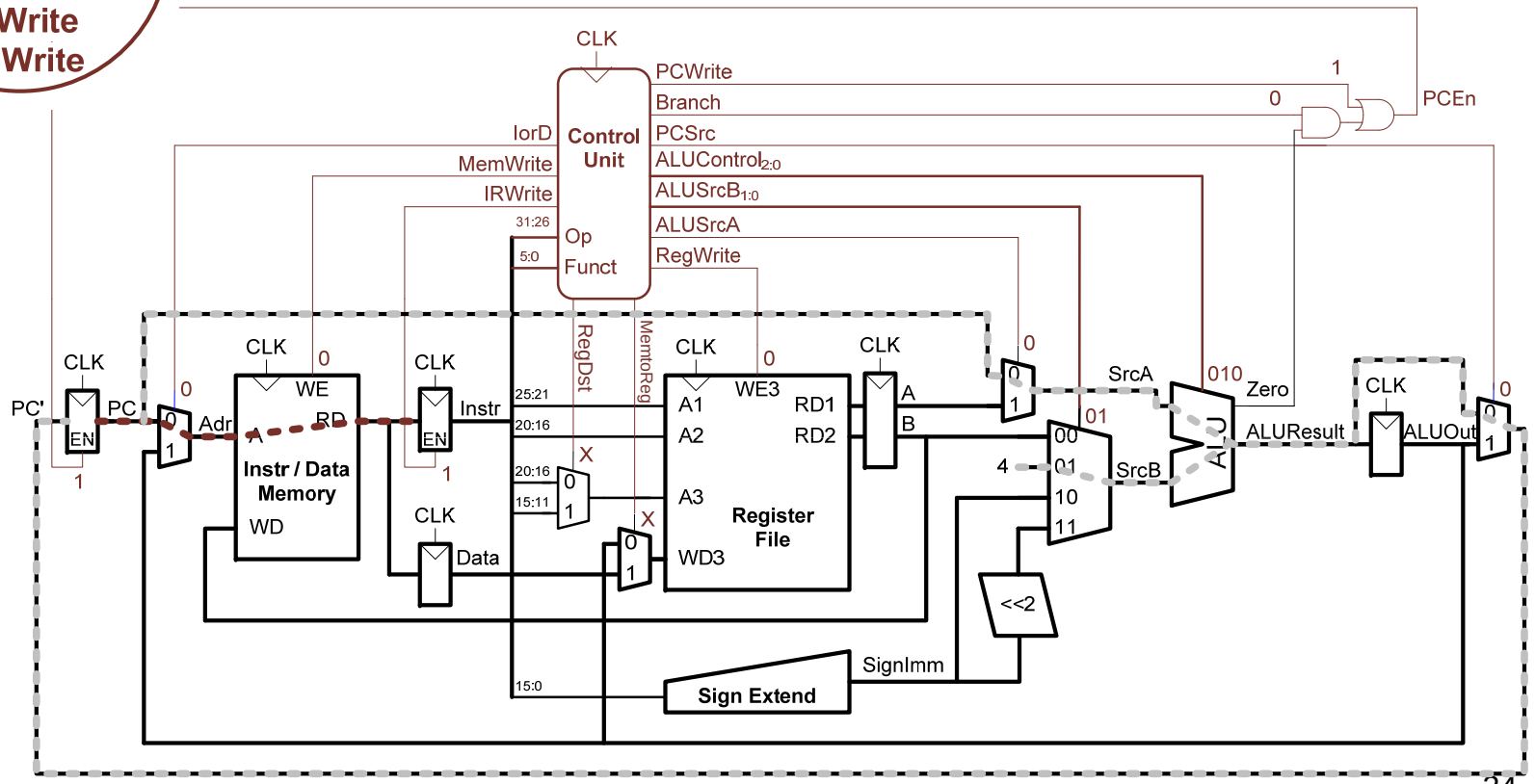
- **Control Unit as a Factored FSM:**



Multi-Cycle Control Unit: Main Decoder /1

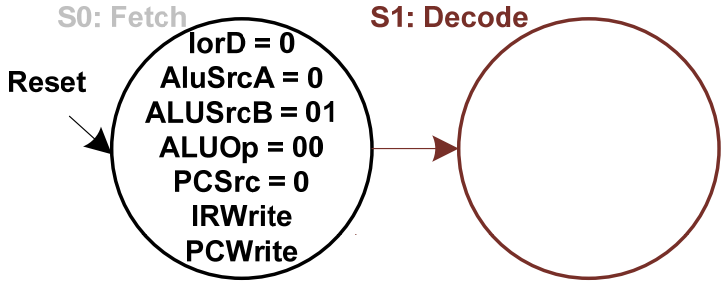
- **Tracing – S0: Fetch**
  - Fetch the instructions from Memory Unit at the address held in PC
  - Use ALU at the same time to increment PC by 4

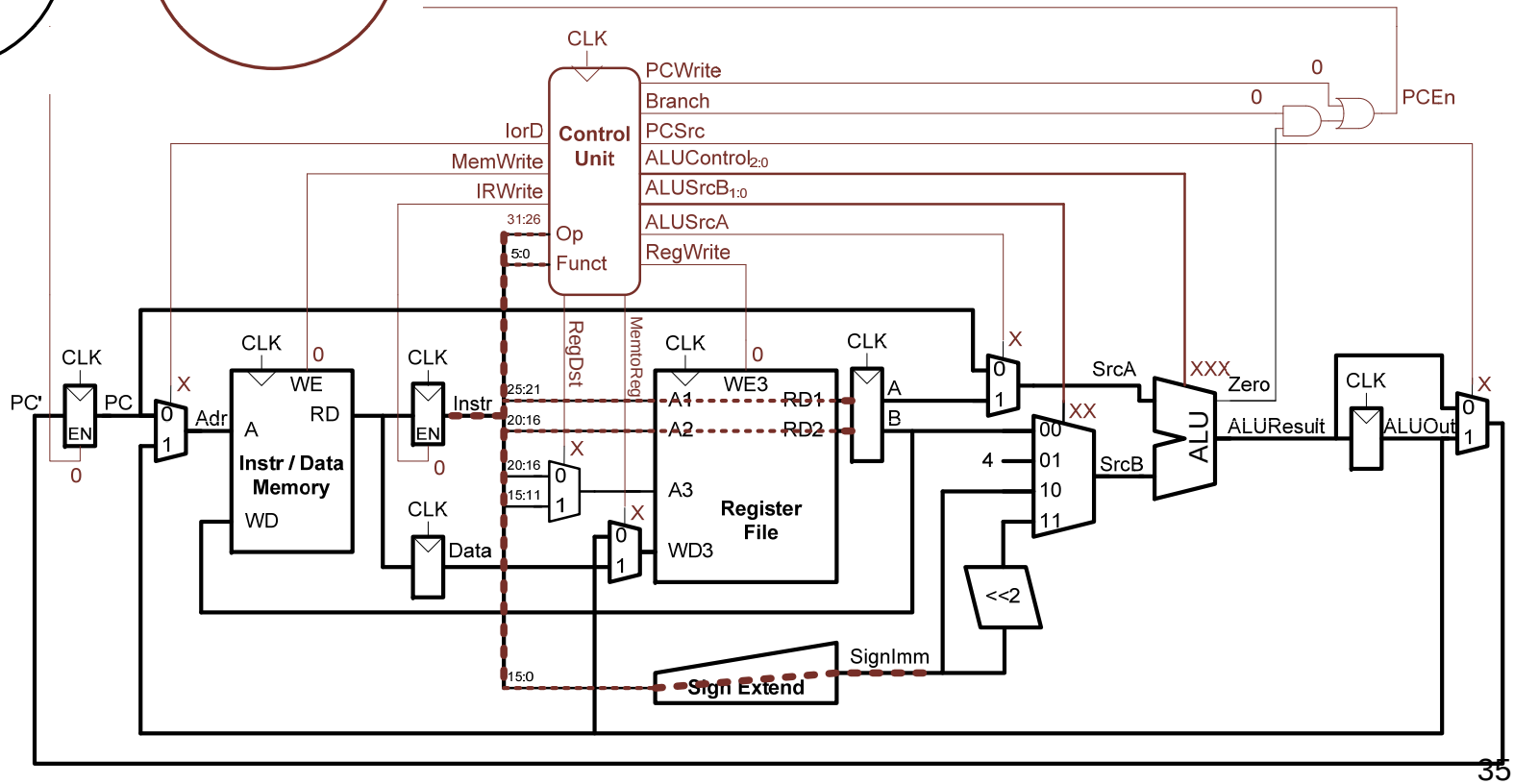




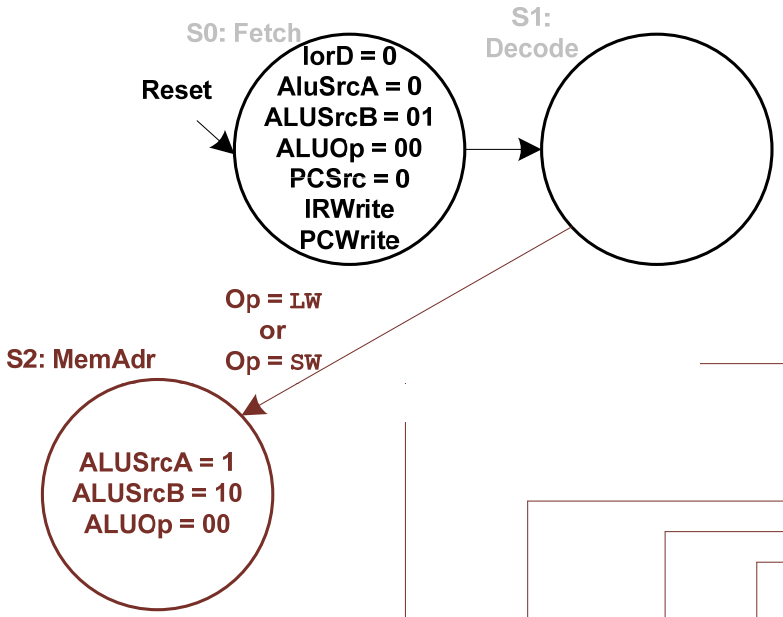
Multi-Cycle Control Unit: Main Decoder /2

- **Tracing – S1: Decode**
  - Read data specified by `rs` and `rt`
  - Decode the `opcode` to decide what to do, but must wait one clock cycle

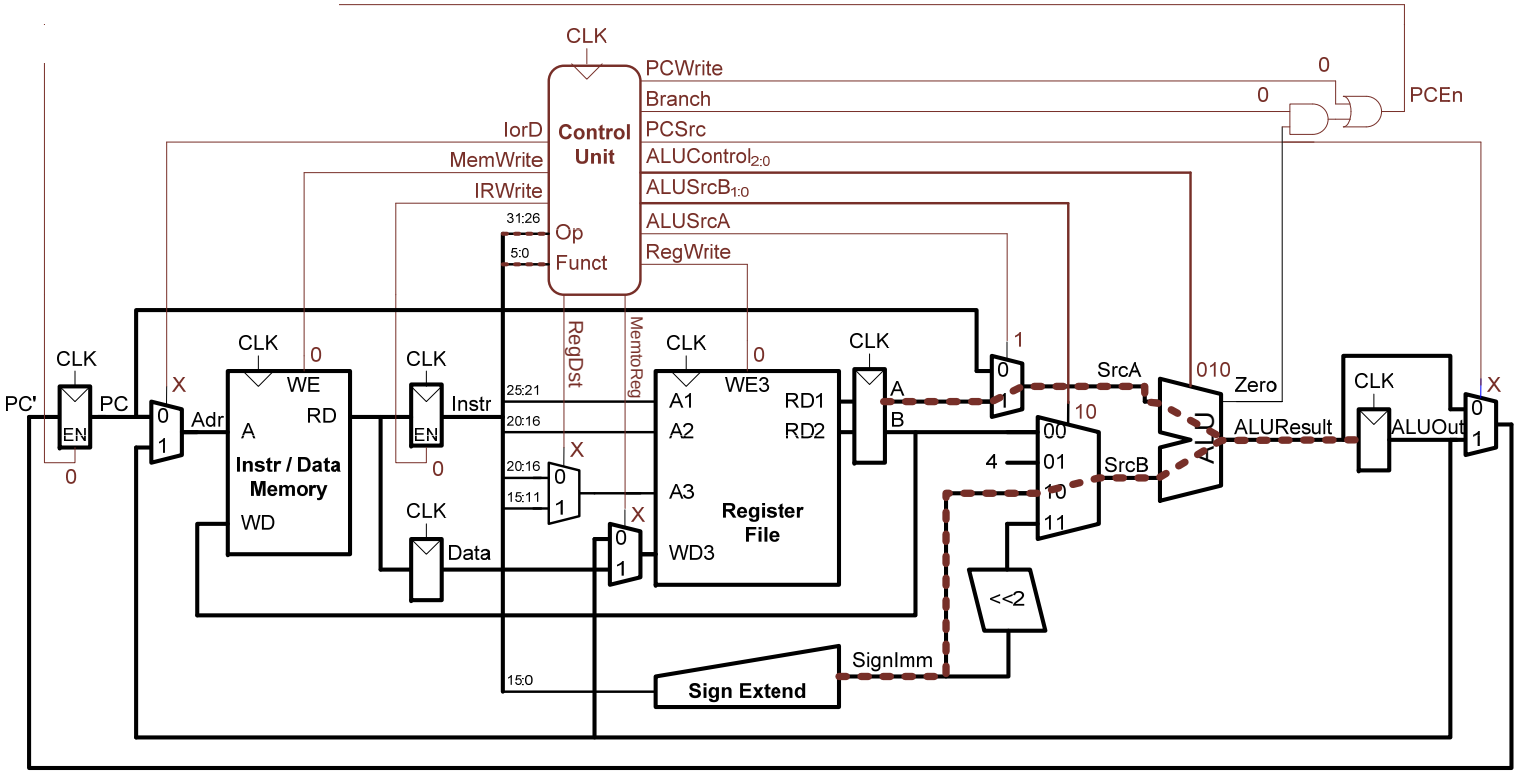




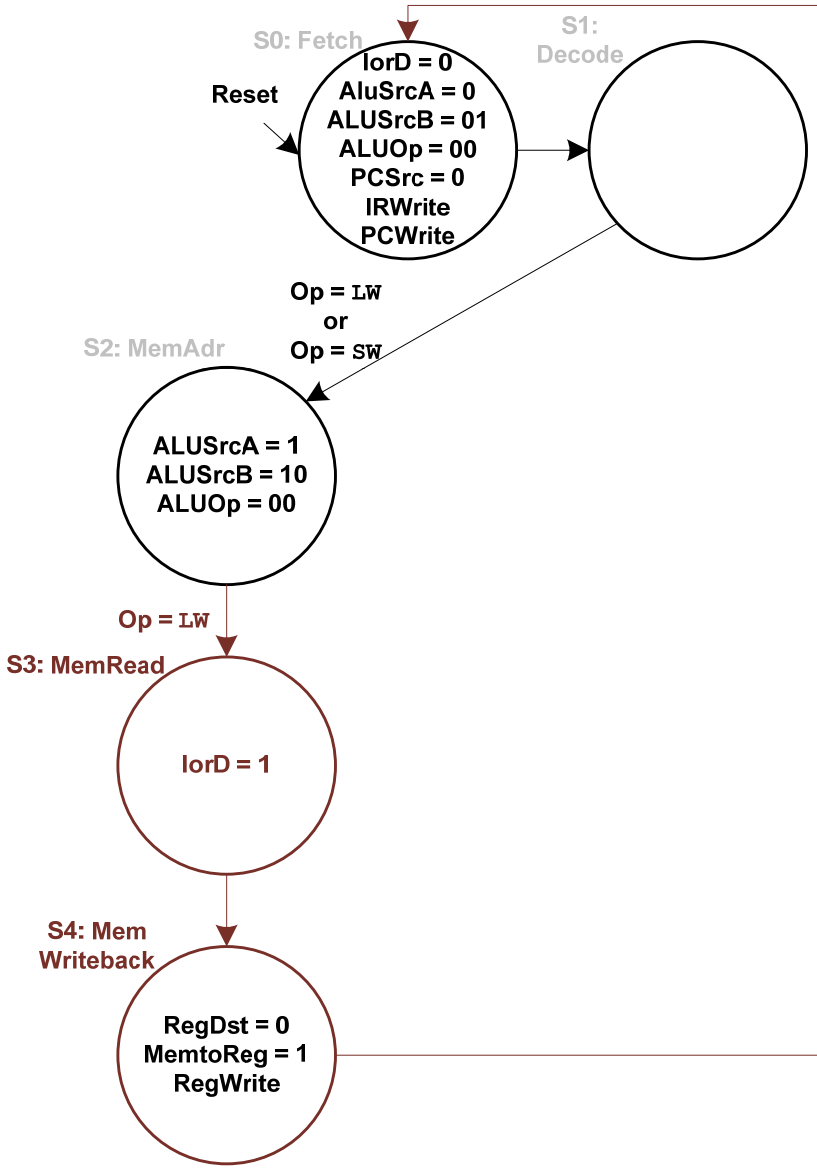
Multi-Cycle Control Unit: Main Decoder /3



- **Tracing – S2: MemAdr**
  - Add the base address to the immediate value, which was sign extended previously

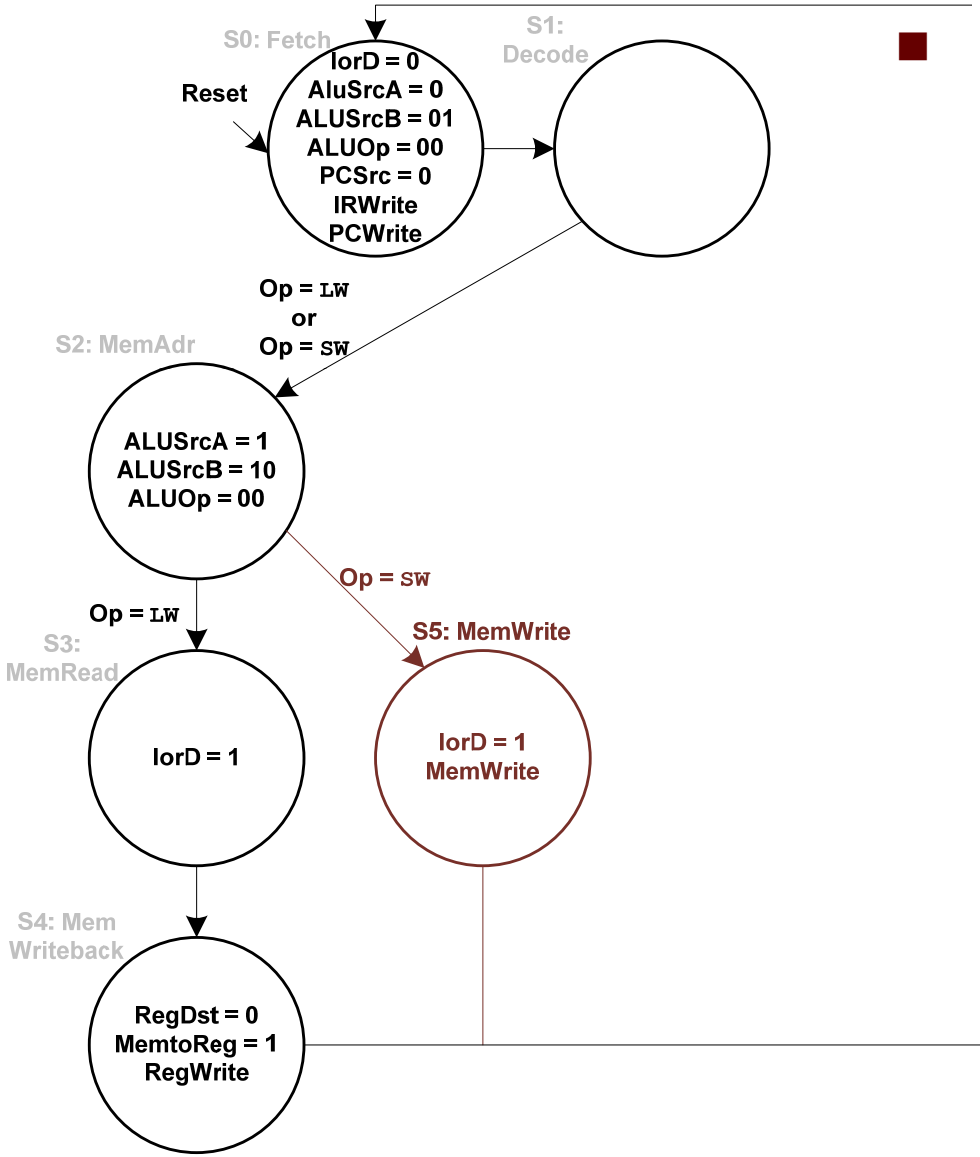


Multi-Cycle Control Unit: Main Decoder /4



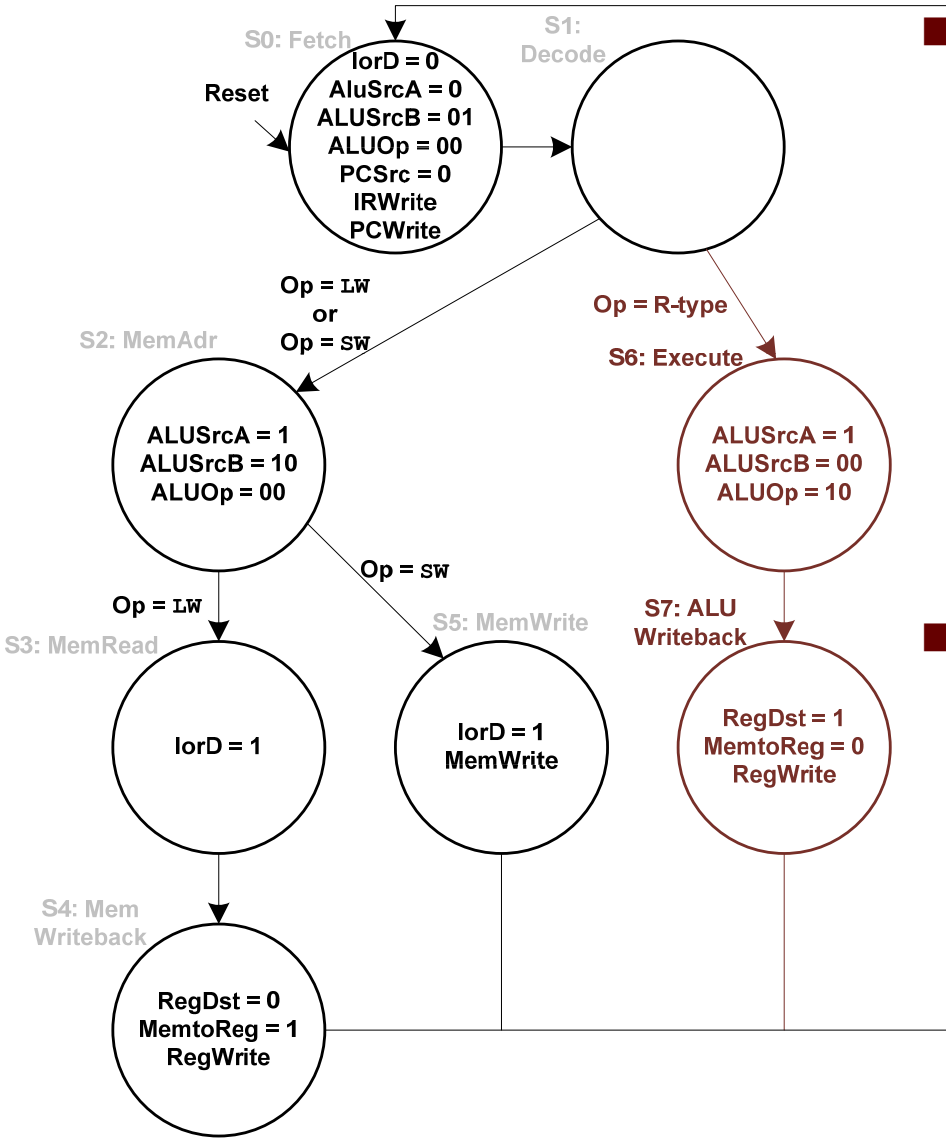
- **Tracing – S3: MemRead**
  - For Op = `lw`
  - Read the address from `ALUOut` register
  - Read the data at this address and store it in Data register
- **Tracing – S4: Mem Writeback**
  - Data is written to Register File
  - `RegWrite` is asserted to enable the write operation
  - Finally, when done with `lw`, the FSM returns to S0

Multi-Cycle Control Unit: Main Decoder /5



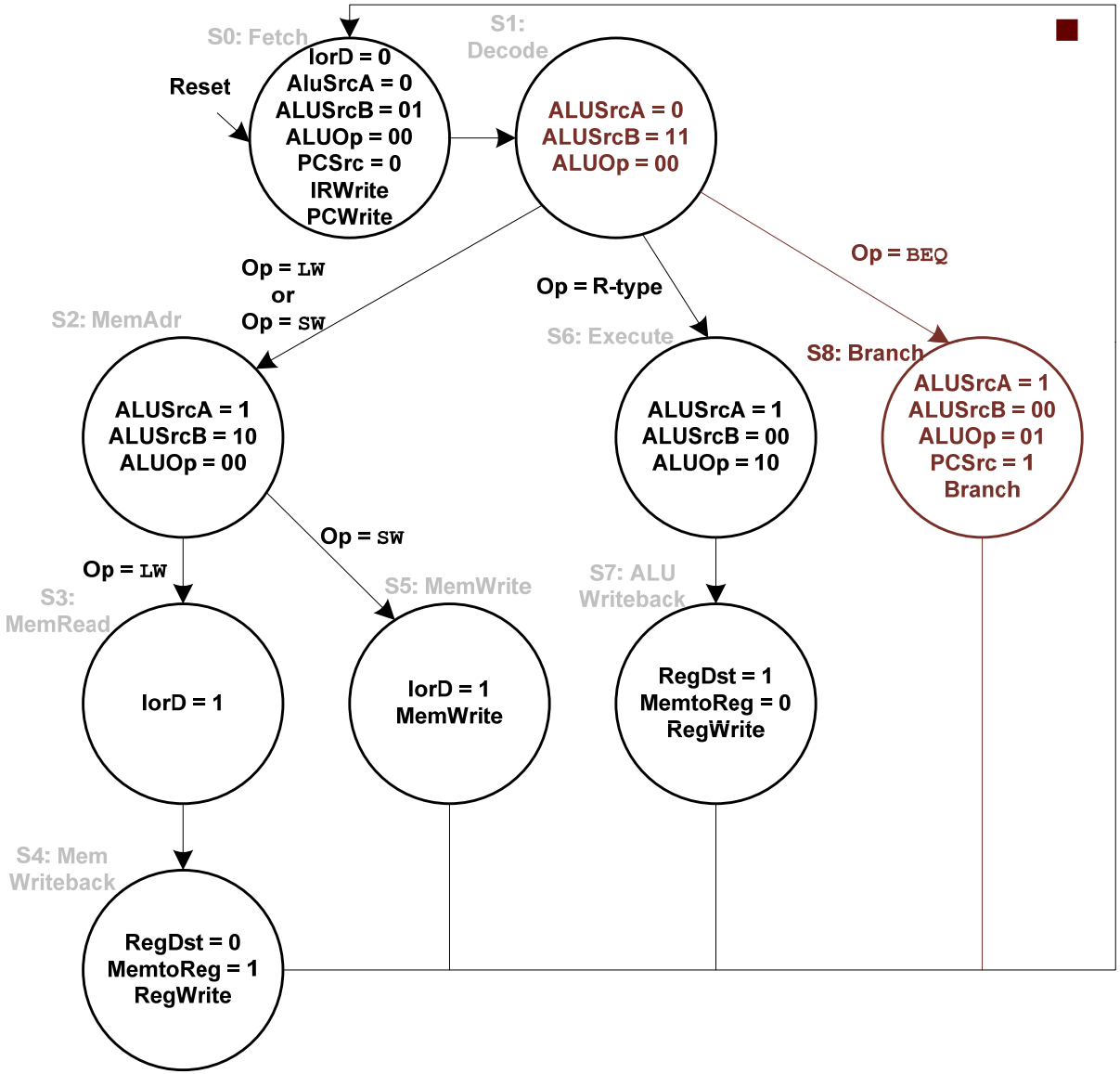
- **Tracing – S5: MemWrite**
  - Op = `sw`
  - Read the address from `ALUOut` register
  - Read the data at this address and store it Memory Unit
  - When done with `sw`, the FSM returns to S0

Multi-Cycle Control Unit: Main Decoder /6



- **Tracing – S6: Execute**
  - Op = `R-type`
  - Select A and B registers and perform the ALU operation indicated by the `funct` field
  - The `ALUResult` is stored in `ALUOut`
- **Tracing – S7: ALU Writeback**
  - Write `ALUOut` to the register file into the destination register specified by `rd` field

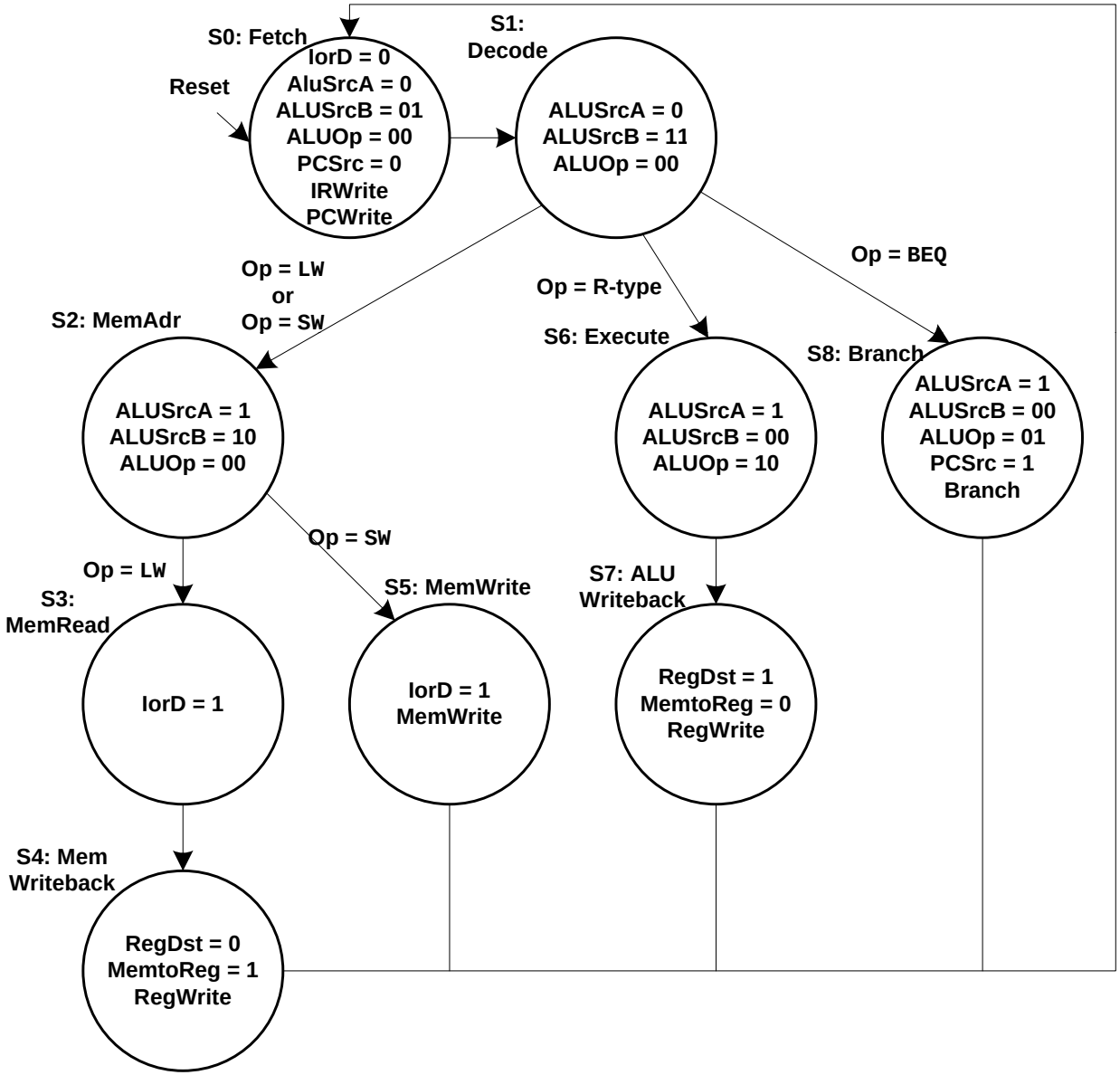
Multi-Cycle Control Unit: Main Decoder /7



Tracing – S8: Branch

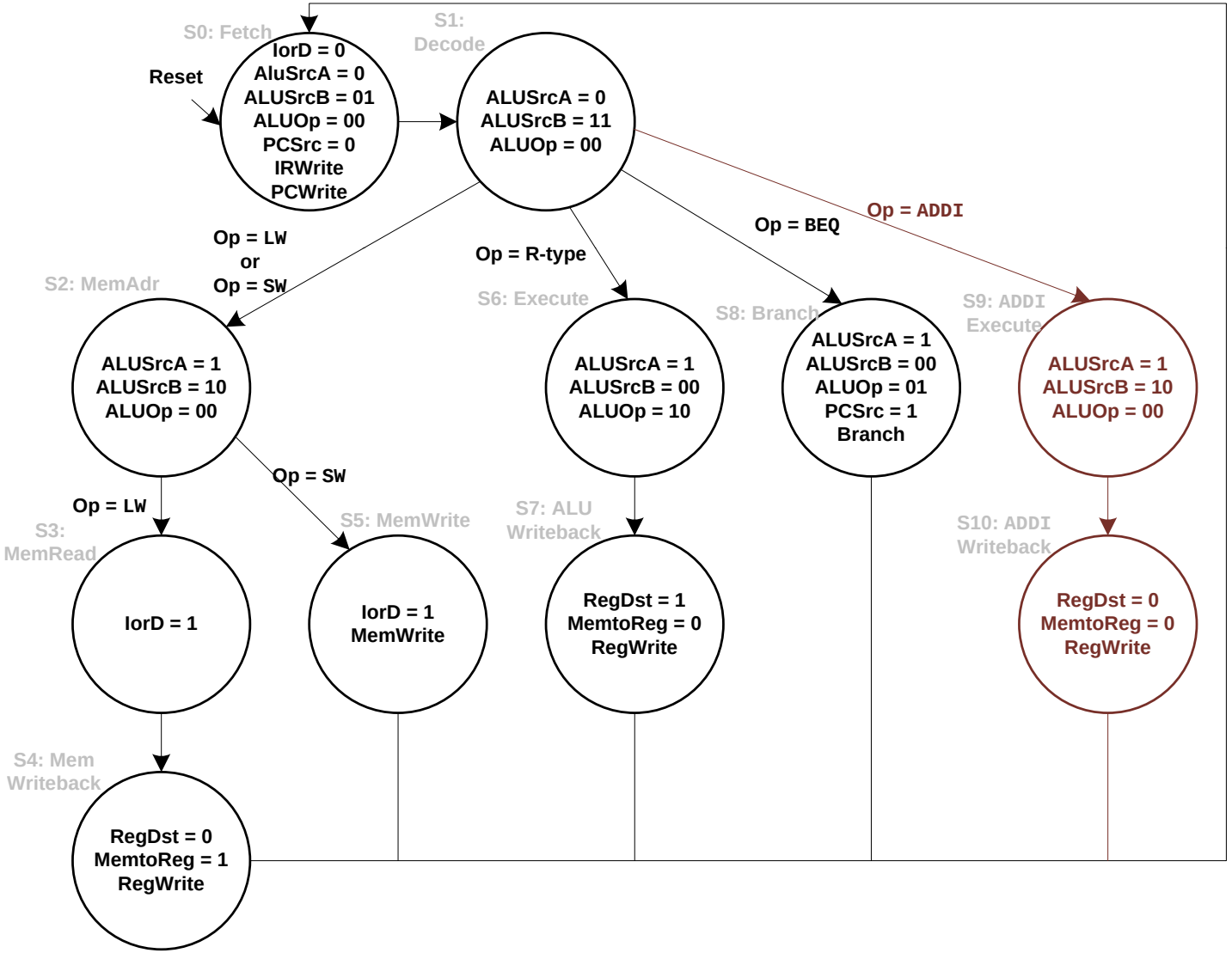
- Op = `beq`
- Calculate the destination address and compare the two source registers to decide whether to branch
- **Use the ALU in S1 (not used otherwise) to compute BTA**
  - If the instruction is not `beq`, this result is ignored
- **Use the ALU again in S8 to subtract the two source registers and check if equal (i.e., `ALUResult` = 0)**

Multi-Cycle Control Unit: Main Decoder /8



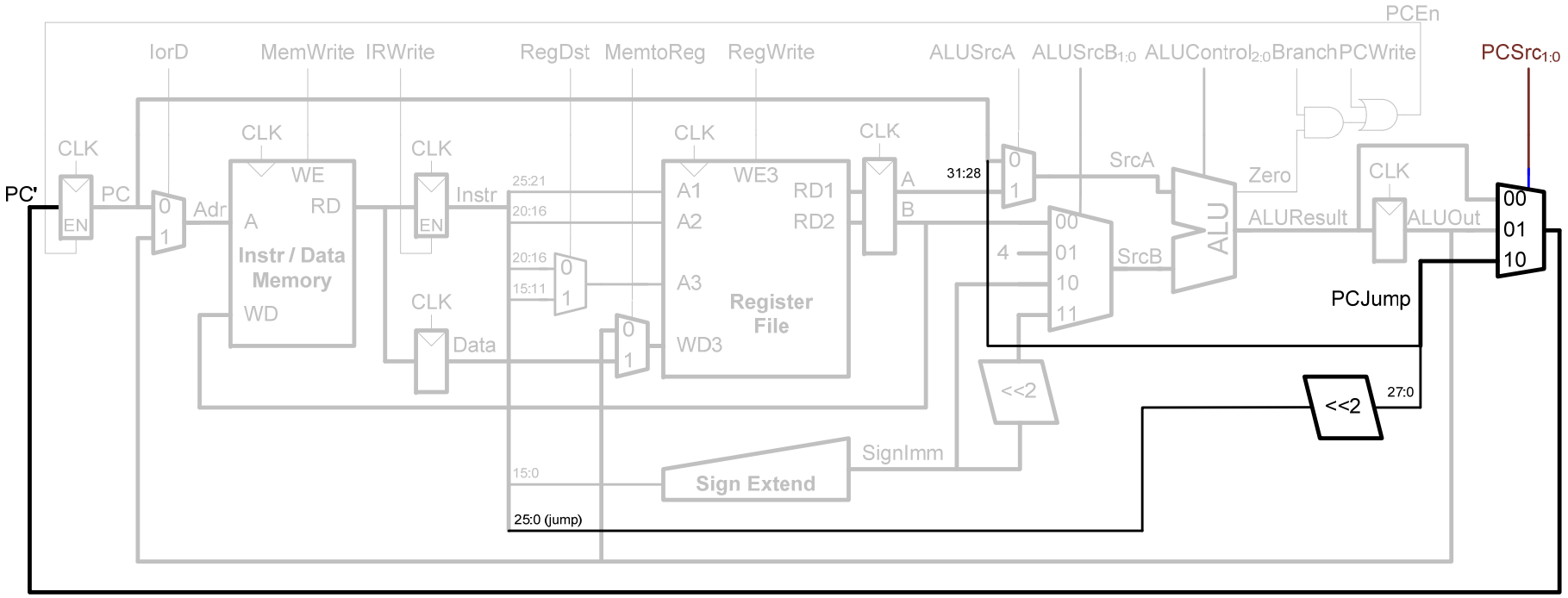
Multi-Cycle Control Unit: Main Decoder /9

- **Extending the Main Decoder FSM for `addi`**



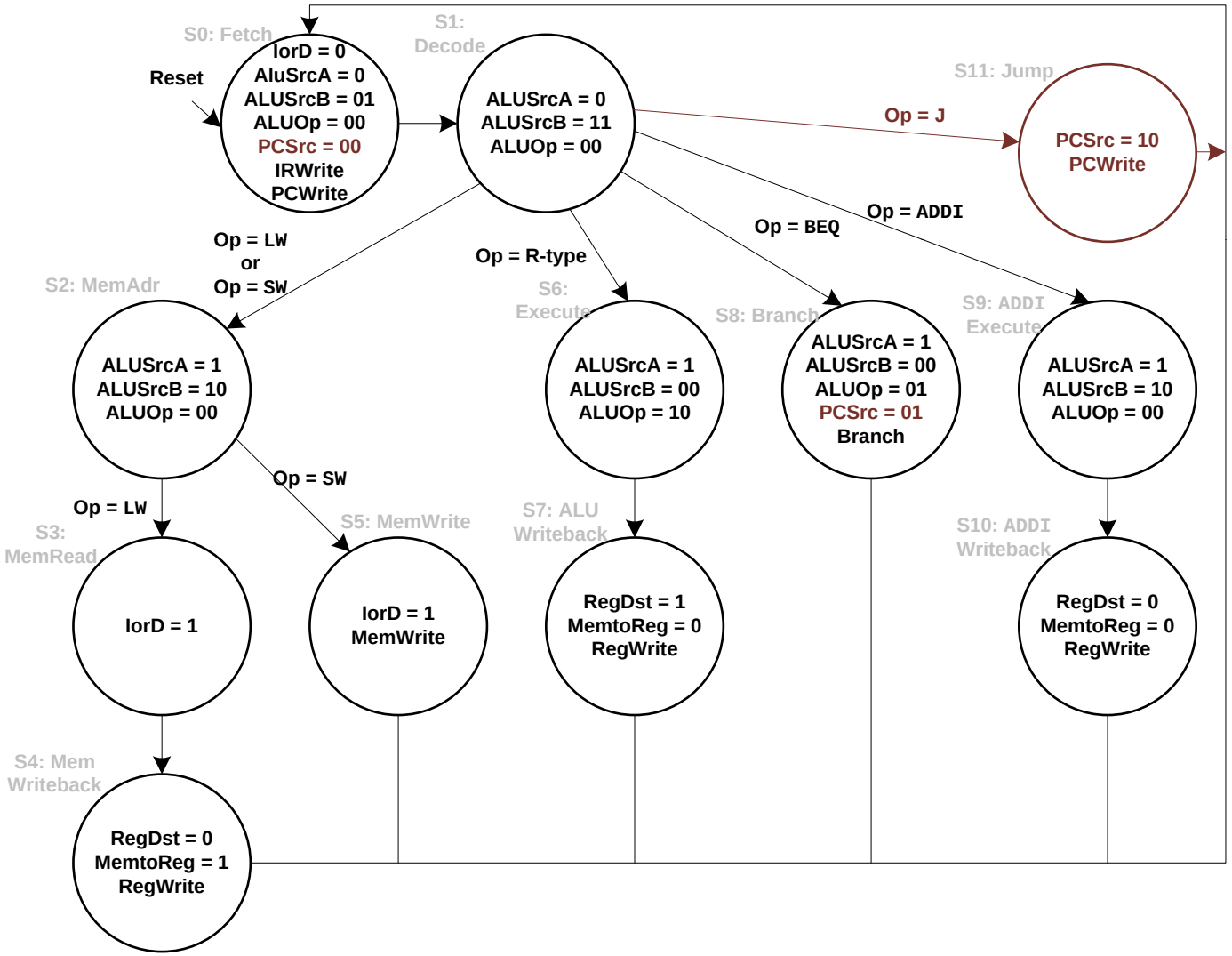
Multi-Cycle Datapath Trace for `j`

- **Multi-Cycle Datapath Extended Functionality: `j`**
  - Additional circuitry



Multi-Cycle Control Unit: Main Decoder /10

- **Extending the Main Decoder FSM for `j`**



Multi-Cycle Datapath Performance /1

- **Sample values for the clock-cycle times (T_c):**
  - **Single-cycle processor T_c: 925 ps**
  - **Multi-cycle processor T_c: 325 ps**
    - See Section 7.4.4 in the Harris textbook for computation details
  - **So the multi-cycle processor can perform faster**
- However, depending on the types of instructions run, multi-cycle processor will not always perform faster
  - For example, experiments show that gcc (GNU C compiler) uses on average 22% loads, 11% stores, 49% R-format, 16% branches, 2% jumps
  - Loads take 5 cycles, stores and R-format take 4 cycles, branches and jumps take 3 cycles
  - Average number of cycles per instruction (CPI) is:
    0.22 x 5 + 0.11 x 4 + 0.49 x 4 + 0.16 x 3 + 0.02 x 3 = 4.04 CPI

Multi-Cycle Datapath Performance /2

- **So which of the two processor designs will perform faster when executing 100 billion gcc instructions?**
  - Single-cycle processor T_c: 925 ps
  - Single-cycle processor CPI: 1
  - Multi-cycle processor T_c: 325 ps
  - Multi-cycle processor CPI: 4.04
  - **Single-cycle processor execution time =**

    # of instructions × CPI × T_c =

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

    92.5 seconds
  - **Multi-cycle processor execution time =**

    # of instructions × CPI × T_c =

    $(100 \times 10^9) \times 4.04 \times (325 \times 10^{-12})$ =

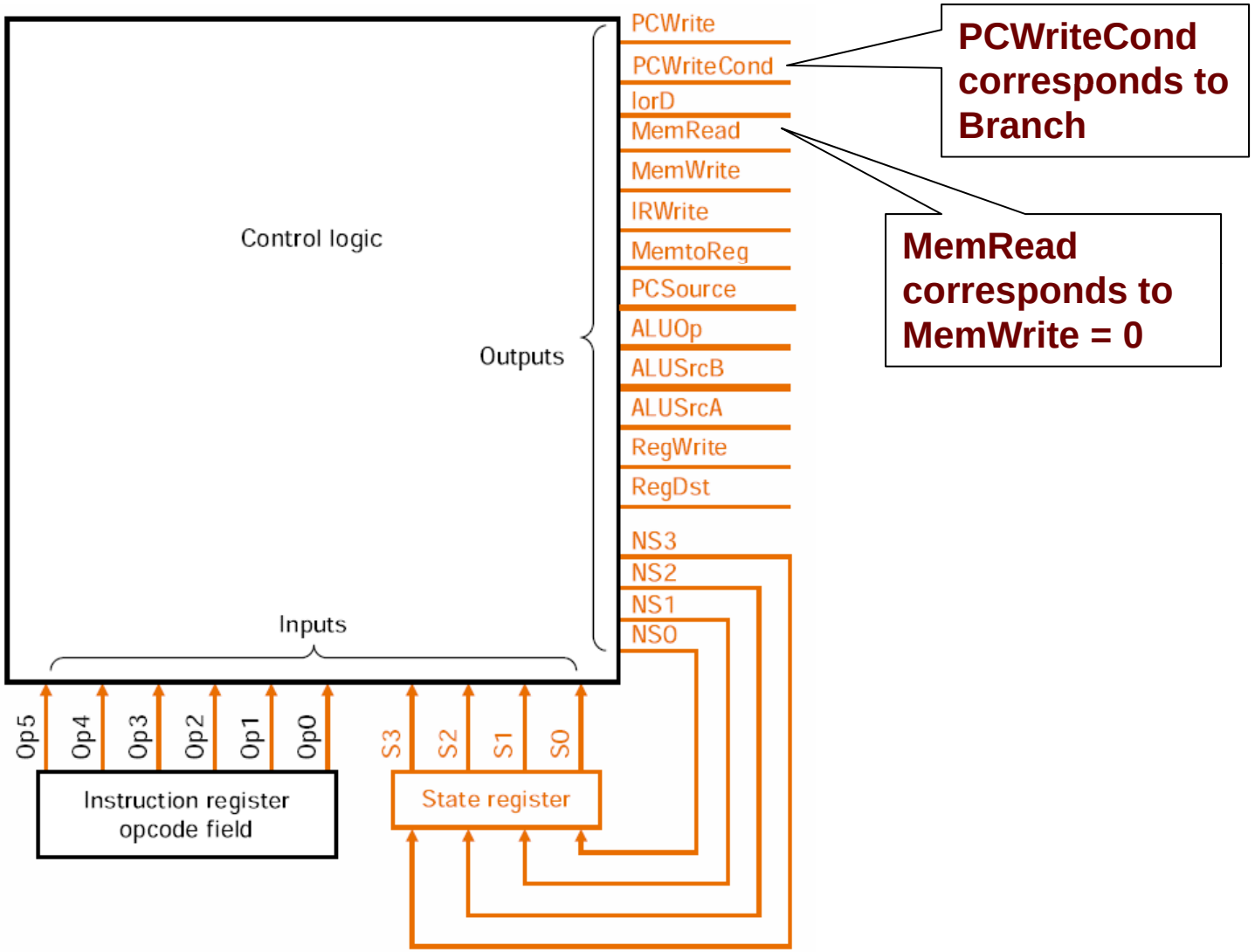
    131.3 seconds

Multi-Cycle Datapath Implementation /1

- **Multi-Cycle Control Unit FSM Implementation:**
  - 10 states in the FSM require 4 bits to encode
  - State changes depend only on the 6 opcode bits
  - Outputs include all signals plus the next state
  - Control logic can be implemented by ROM or PLA
- **Recall that PLA implies sum-of-products ordering of inputs to outputs**

Multi-Cycle Datapath Implementation /2

- **Multi-Cycle Control Unit FSM Implementation:**



Multi-Cycle Datapath Implementation /3

- **ROM:**

  - ROM corresponds to a table of 2^n m-bit words
  - ROM can be viewed as m one-bit functions of n variables
  - Internally, includes a decoder plus an OR gate for each output
  - For multi-cycle datapath, ROM is of size 2^{10} x 20
    - 10 for (6 + 4) input bits and 20 for (16 + 4) output bits
  - At the same time, PLA requires 17 terms for 17 output signals
  - **PLA design can be significantly more efficient**

Food for Thought

- **Download and Read Assignment #3 Specifications**
- **Read:**
  - Chapters 4 and 5 from the Course Notes
    - Review the material discussed in the lecture notes in more detail
    - Our course schedule follows the material in the Course Notes
  - (Optional) Chapter 7 of the Harris and Harris textbook