Multi-cycle Microarchitecture Design

Recall: Performance Analysis Basics

Excution time of a single instruction

{CPI} x {clock cycle time}

Excution time of an entire program

Sum over all instruction [{CPI} x {clock cycle time}]

Also: {# of instructions} x {Average CPI} x {clock cycle time}

Processor Performance

How fast is my program?

- Every program consists of a series of instrs.
- Each instr needs to be excuted.

How fast are my instructions?

- Instrs are realized on the hardware
- Each instr can take one or more clock cycles to complete
- CPI = cycles per instr

How long is one clock cycle?

- The critical path determines how much time one cycle requires = clock period
- 1/clock period = clock frequency = how many clock cycles are in each second.

So our program executes in:

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

- N: N instrs
- CPI: on average
- T: clock period

Analyse the single-cycle uarch:

① CPI is strictly 1

② How long each instr takes is determined by how long the slowest instr takes to execute
Even though many instrs don't need that long to execute

③ Clock cycle time of the uarch is determined by how long it takes to complete the slowest instr

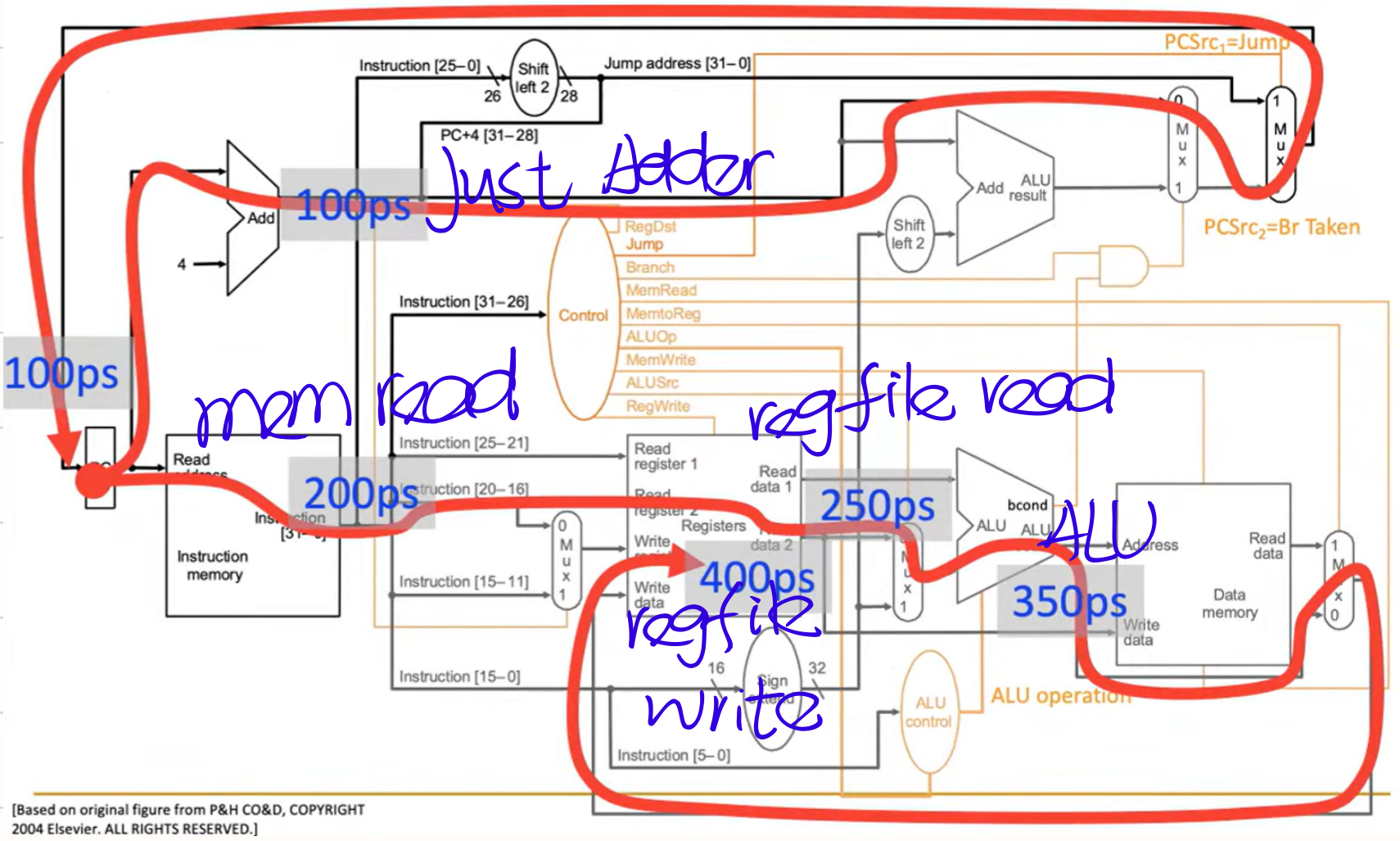
Example Single-Cycle Datapath Analysis

- Assume (for the design in the previous slide)
  - memory units (read or write): 200 ps
  - ALU and adders: 100 ps
  - register file (read or write): 50 ps
  - other logic or wire delay: 0 ps

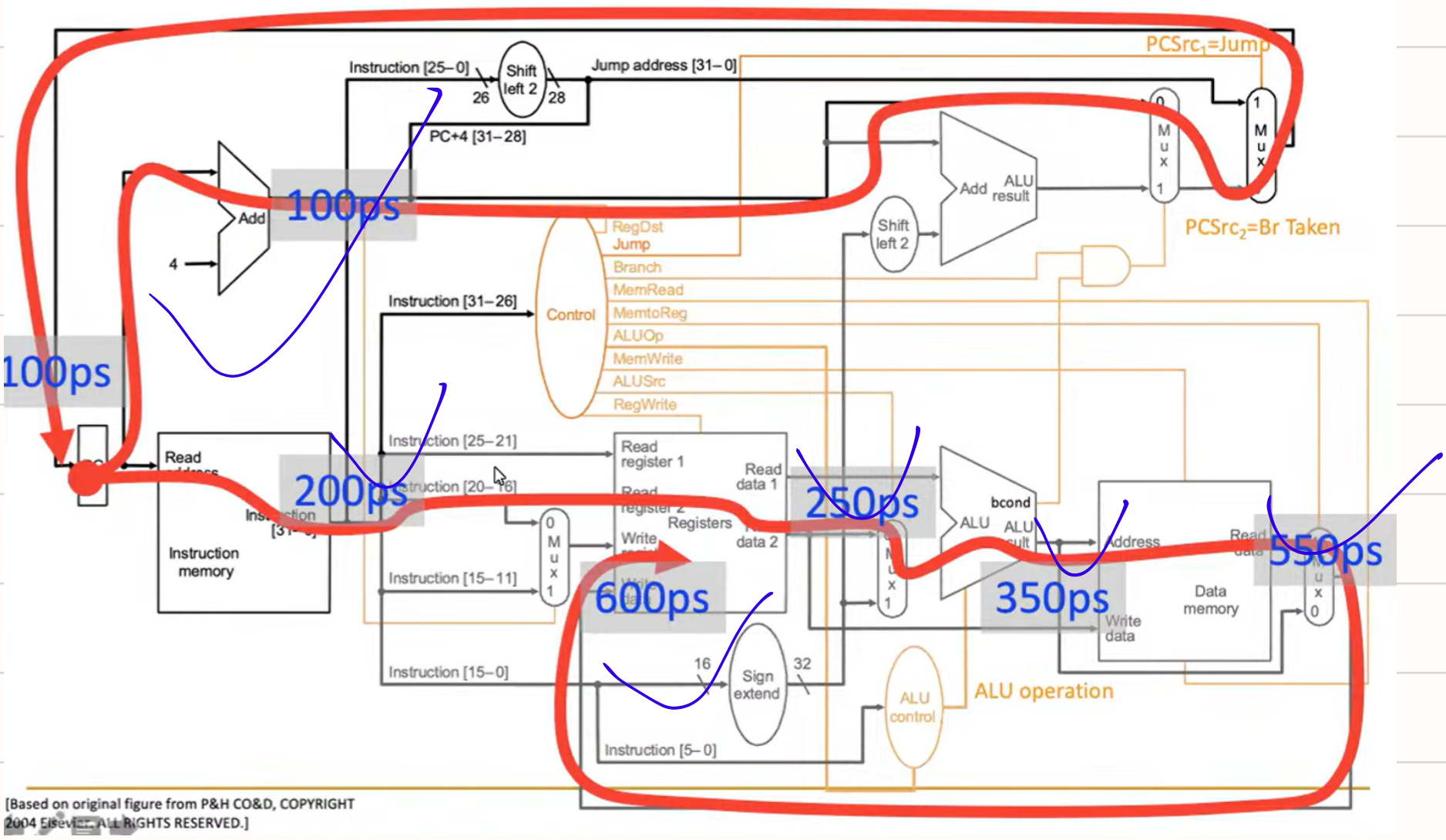
steps	IF	ID	EX	MEM	WB	Delay
resources	mem	RF	ALU	mem	RF	
R-type	200	50	100		50	400
I-type	200	50	100		50	400
LW	200	50	100	200	50	600
SW	200	50	100	200		550
Branch	200	50	100			350
Jump	200					200

R-Type and I-Type ALU

400 PS

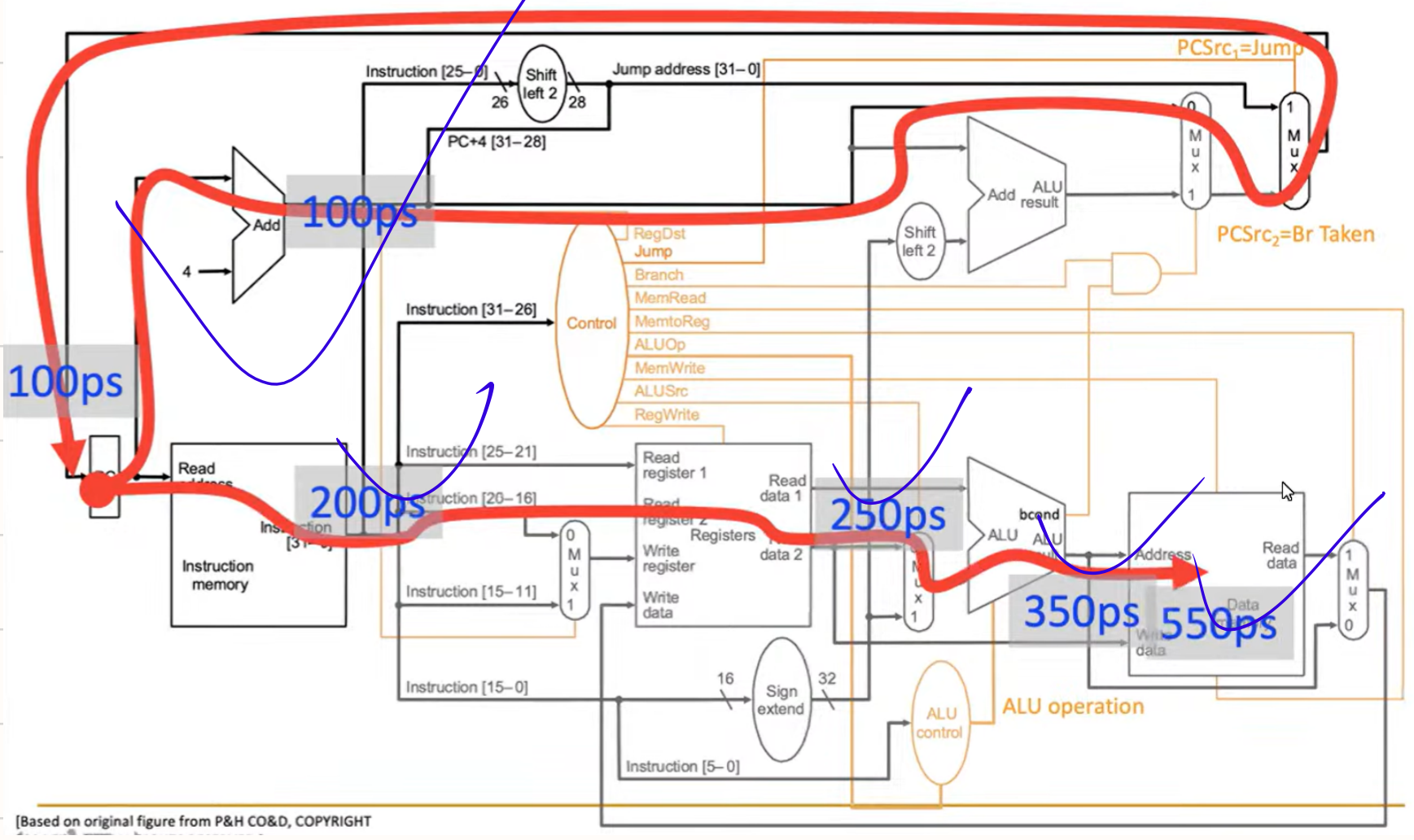


LW 600ps



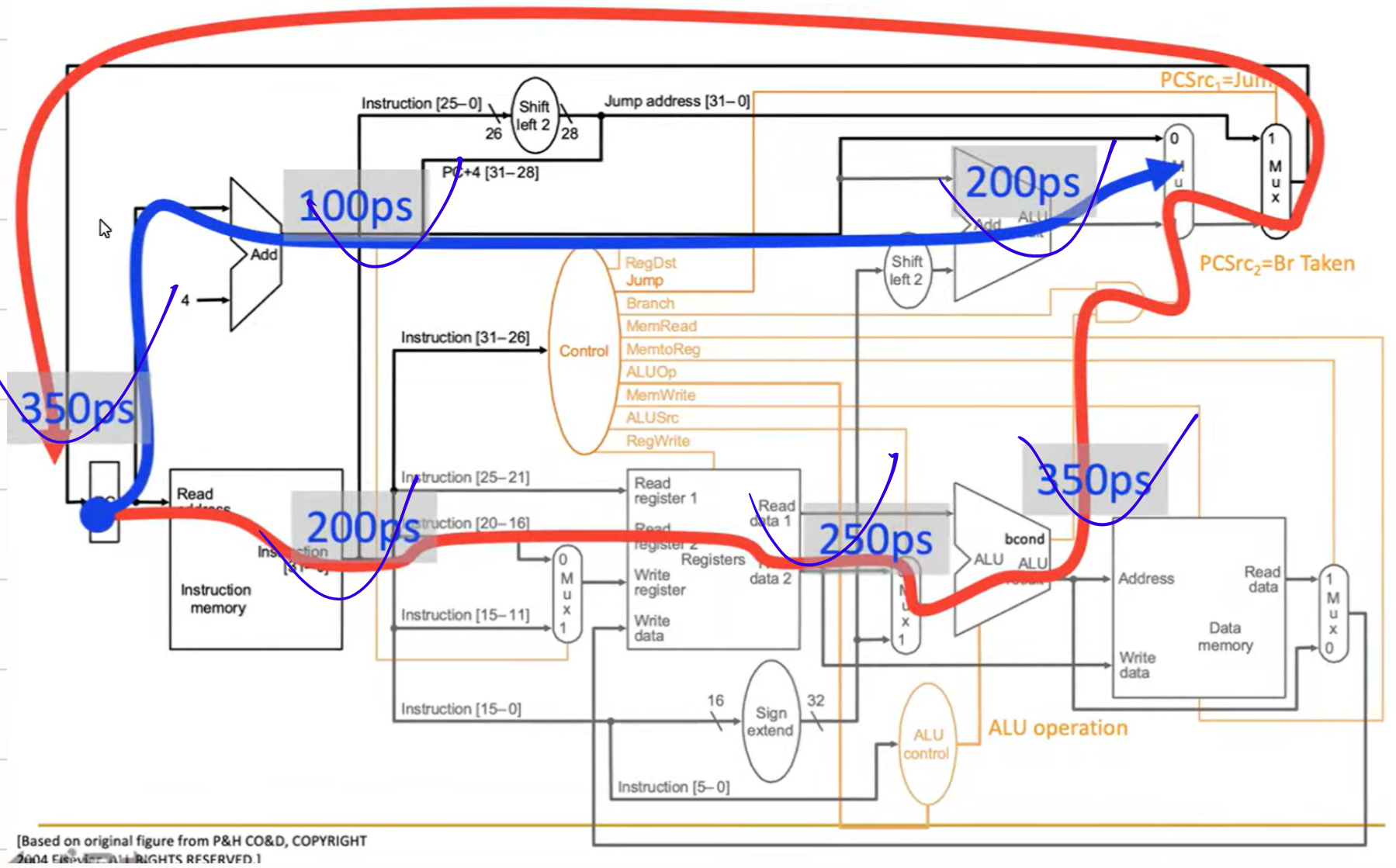
[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

SW 550ps



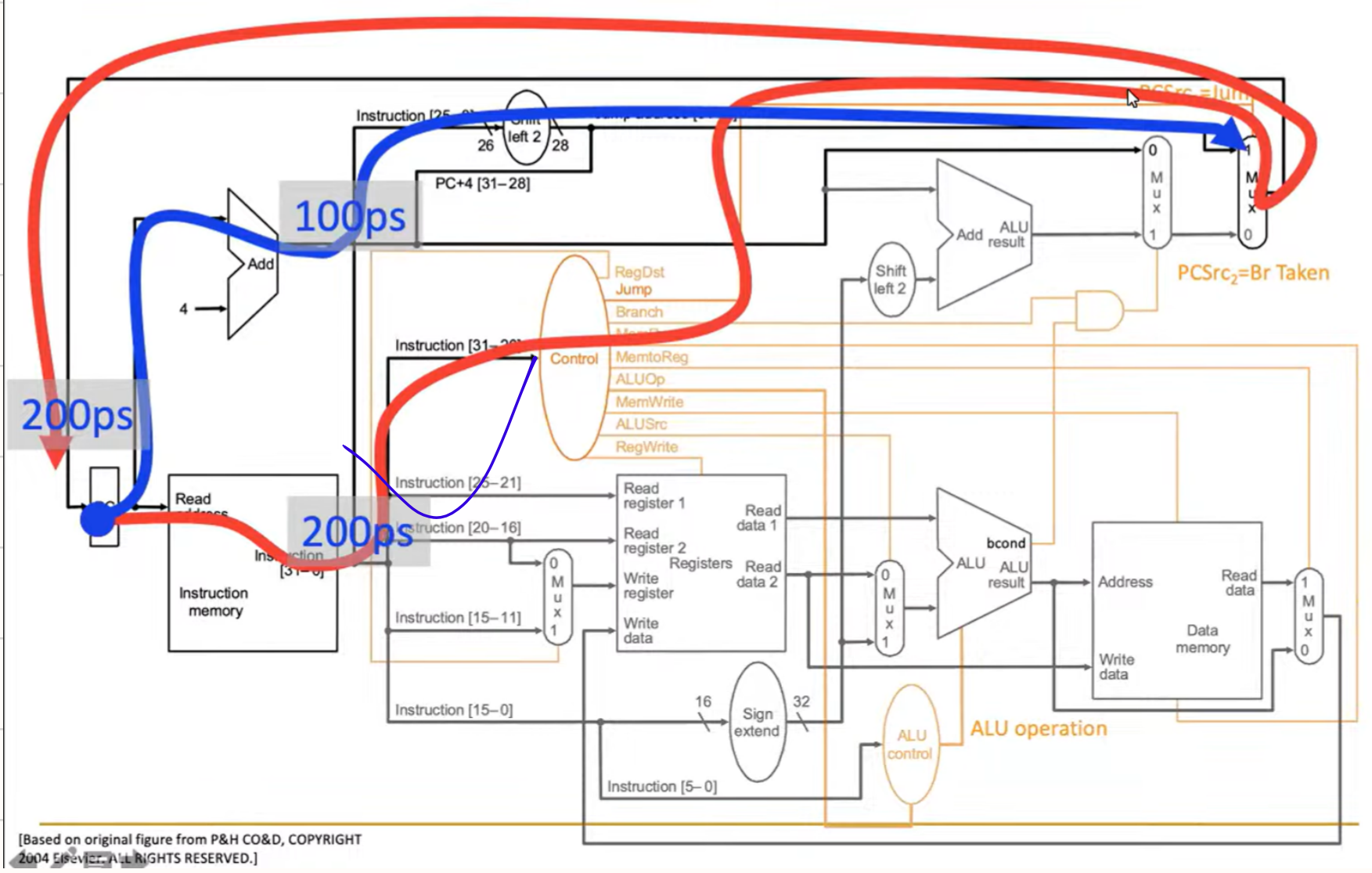
[Based on original figure from P&H CO&D, COPYRIGHT

Branch Taken 350ps



[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

Jump 200ps



[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

Memory is really the slowest Instr to process

(Micro)architecture Design Principles

Critical path design

Find and decrease the maximum combinati-onal logic delay.

Break a path into multiple cycles if it takes too long

Bread and butter (common case) design

Spend time and resources on where it matters most

Common case vs. uncommon case

Balanced design:

Balance instr/data flow through hardware components

Design to eliminate bottlenecks:

balance the hardware for the work

When designing computer systems/architectures it is important to follow good principles!

Architecture based upon principle, and not upon precedent

Multi-cycle Uarch

Goal: Let each instruction take (close to) only as much time it really needs.

Idea:

Determine clock cycle time independently of instruction processing time.

Each instruction takes as many clock cycles as it needs to take.

Multiple state transitions per instruction

The states followed by each instruction is different.

Multi-Cycle Microarchitecture

AS = Architectural (programmer visible) state at the beginning of an instruction

↓

Step 1: Process part of instruction in one clock cycle

↓

Step 2: Process part of instruction in the next clock cycle

↓

...

↓

AS' = Architectural (programmer visible) state at the end of a clock cycle

Benefits of Multi-Cycle Design

- Critical path design
  - Can keep reducing the critical path independently of the worst-case processing time of any instruction
- Bread and butter (common case) design
  - Can optimize the number of states it takes to execute "important" instructions that make up much of the execution time
- Balanced design
  - No need to provide more capability or resources than really needed
    - An instruction that needs resource X multiple times does **not** require multiple X's to be implemented
    - Leads to more efficient hardware: Can reuse hardware components needed multiple times for an instruction

Downsides:

Need to store the intermediate results at the end of each clock cycle.

Hardware overhead for microarchitectural registers.

Register setup/load overhead (i.e., sequencing overhead) is paid multiple times for an instr

Key Idea for Realization

One can implement the "process instruction" step as a finite state machine that sequences between states and eventually returns back to the "fetch instruction" state

A state is defined by the control signals asserted in it.

Control signals for the next state are determined in current state

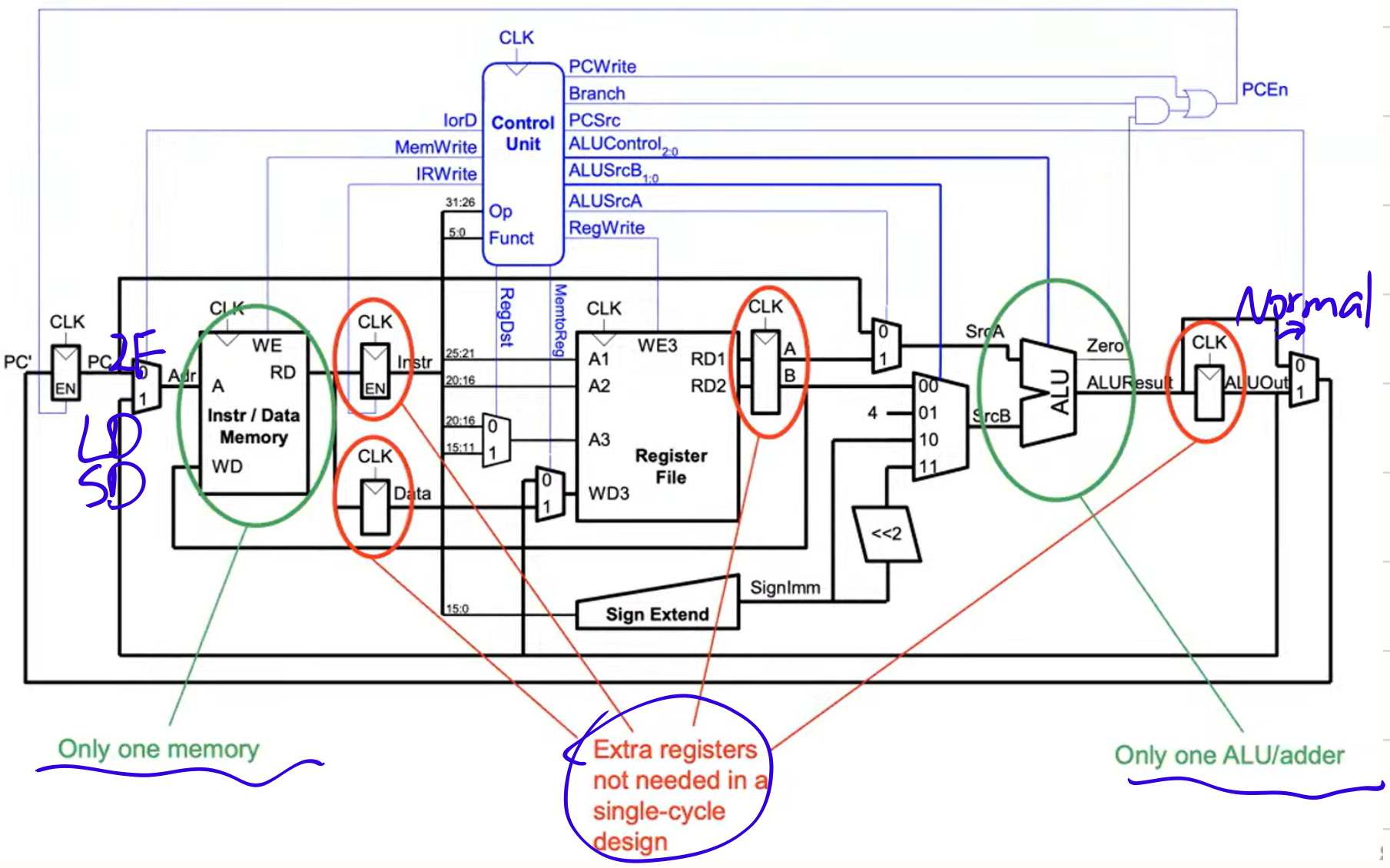
A Basic Multi-Cycle Microarchitecture

- Instruction processing cycle divided into "states"
  - A stage in the instruction processing cycle can take multiple states
- A multi-cycle microarchitecture sequences from state to state to process an instruction
  - The behavior of the machine in a state is completely determined by control signals in that state
- The behavior of the entire processor is specified fully by a *finite state machine*
- In a state (clock cycle), control signals control two things:
  - How the datapath should process the data
  - How to generate the control signals for the (next) clock cycle

What Can We Optimize with Multi-Cycle

- Single-cycle microarchitecture uses two memories
  - One memory stores instructions, the other data
  - We want to use a single memory (lower cost)
- Single-cycle microarchitecture needs three adders
  - ALU, PC, Branch address calculation
  - We want to use only one ALU for all operations (lower cost)
- Single-cycle microarchitecture: each instruction takes one cycle
  - The slowest instruction slows down every single instruction
  - We want to determine clock cycle time independently of instruction processing time
    - Divide each instruction into multiple clock cycles
    - Simpler instructions can be very fast (compared to the slowest)

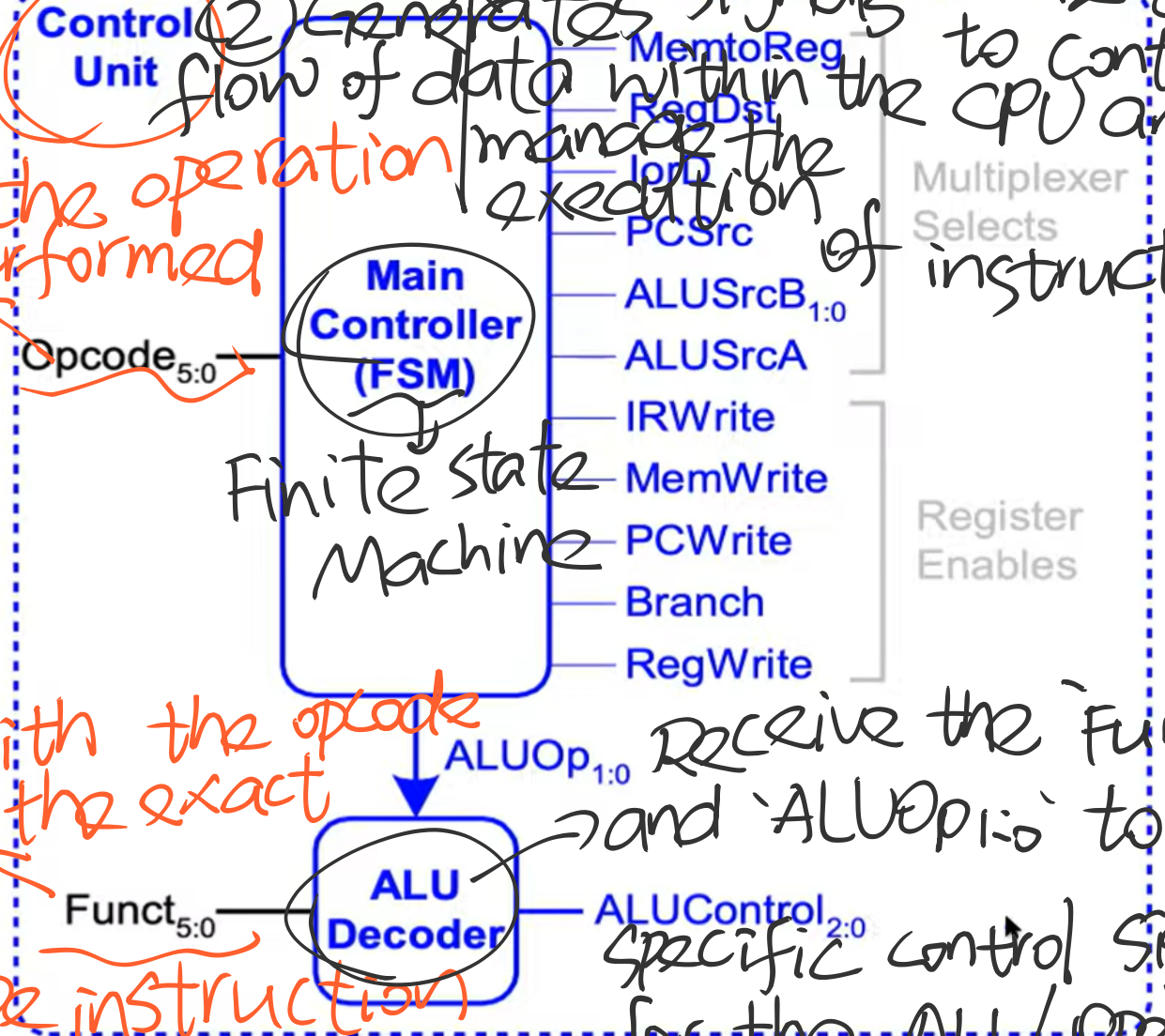
Complete Multi-Cycle Processor



Control Unit

① Receive the opcode and dictate the sequence of operations to be executed

② Generates signals to control the flow of data within the CPU and manage the execution of instructions



Dictate the operation to be performed

Finite state Machine

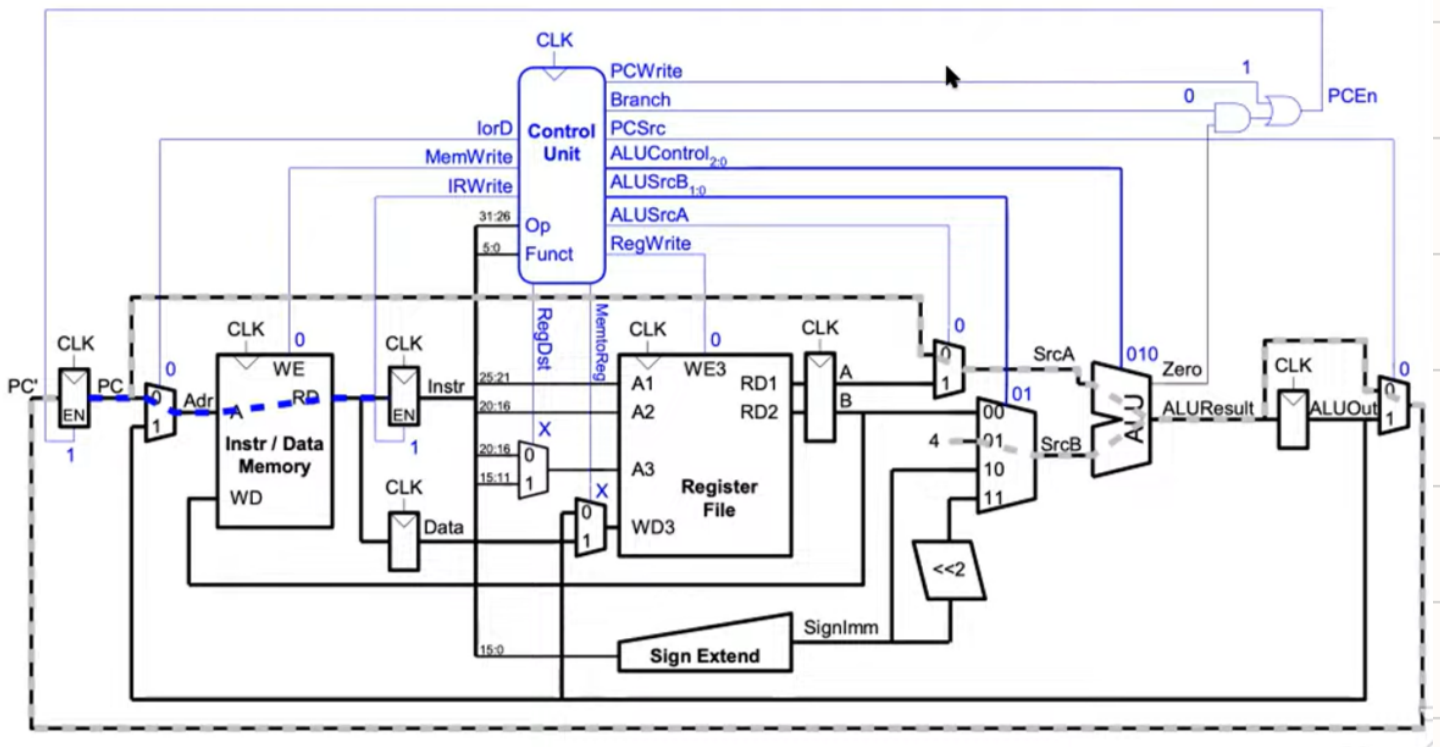
Together with the opcode determine the exact operation in R-type instruction

Receive the `Funct1:0` and `ALUOp1:0` to generate specific control signals for the ALU operation `ALUcontrol`

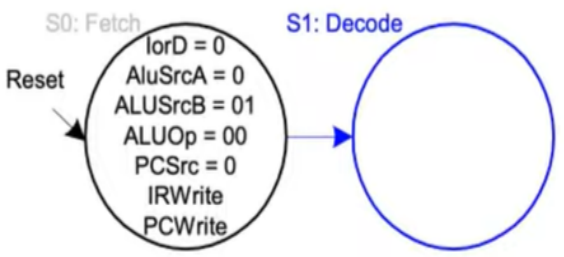
Control Signal	Description
MemtoReg	Determines whether the data to be written to a register comes from memory or the ALU.
RegDst	Selects the register in which the ALU result or the data loaded from memory will be stored.
IorD	Determines whether the address to the memory is from the instruction register or from the ALU.
PCSrc	Determines the source of the Program Counter's next value (e.g., branch, jump, or sequential).
ALUSrcB	Selects the second operand of the ALU between the sign-extended immediate value and a register.
ALUSrcA	Selects the first operand of the ALU between the Program Counter and a register.
IRWrite	Enables writing to the Instruction Register.
MemWrite	Enables writing data to the memory.
PCWrite	Enables writing to the Program Counter, updating its value.
Branch	Controls whether a branch should be taken, combining with PCSrc to compute the next PC value.
RegWrite	Enables writing to the register file.

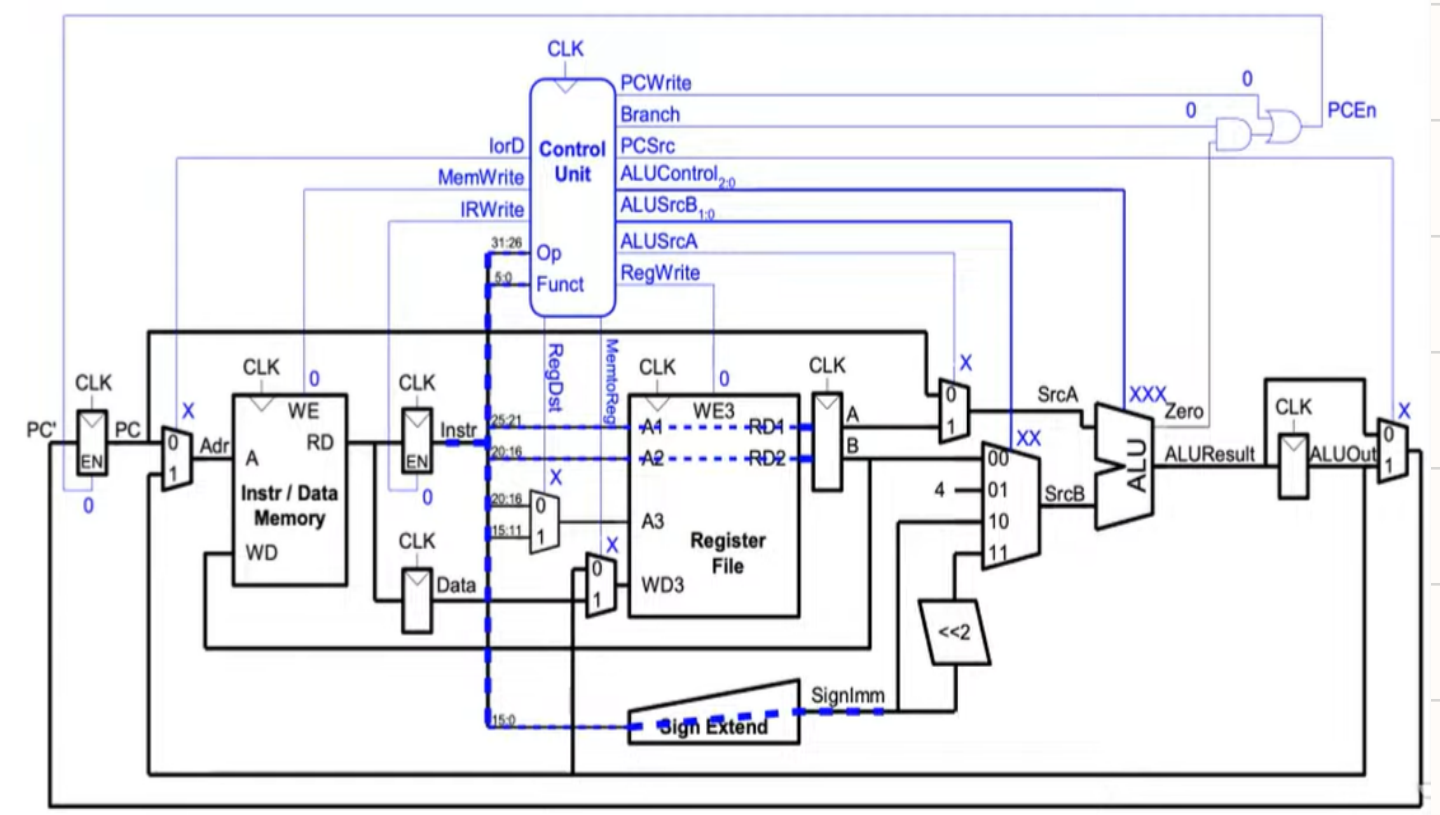
Instruction or Data

Main Controller FSM: Fetch

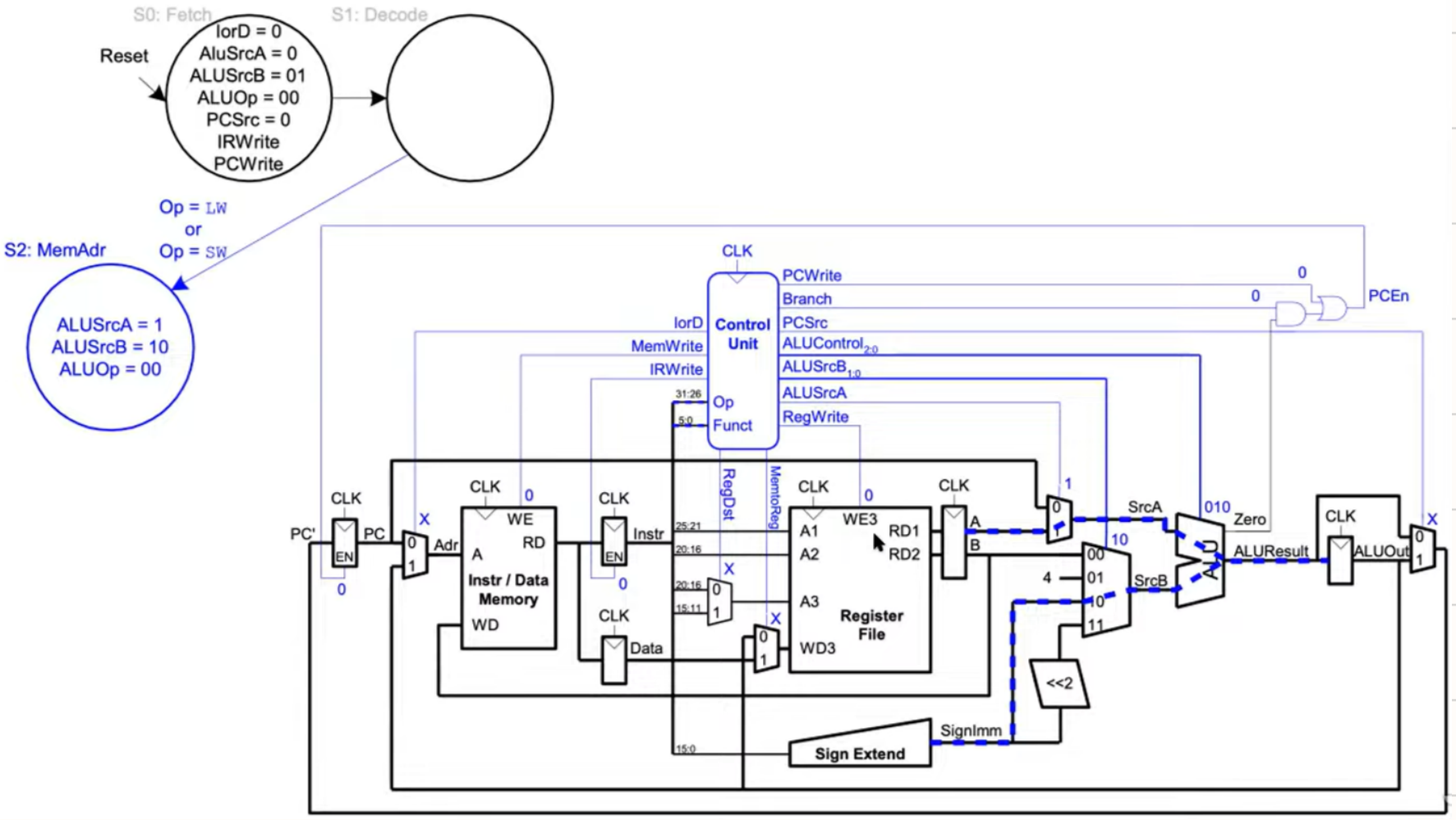


Main Controller FSM: Decode

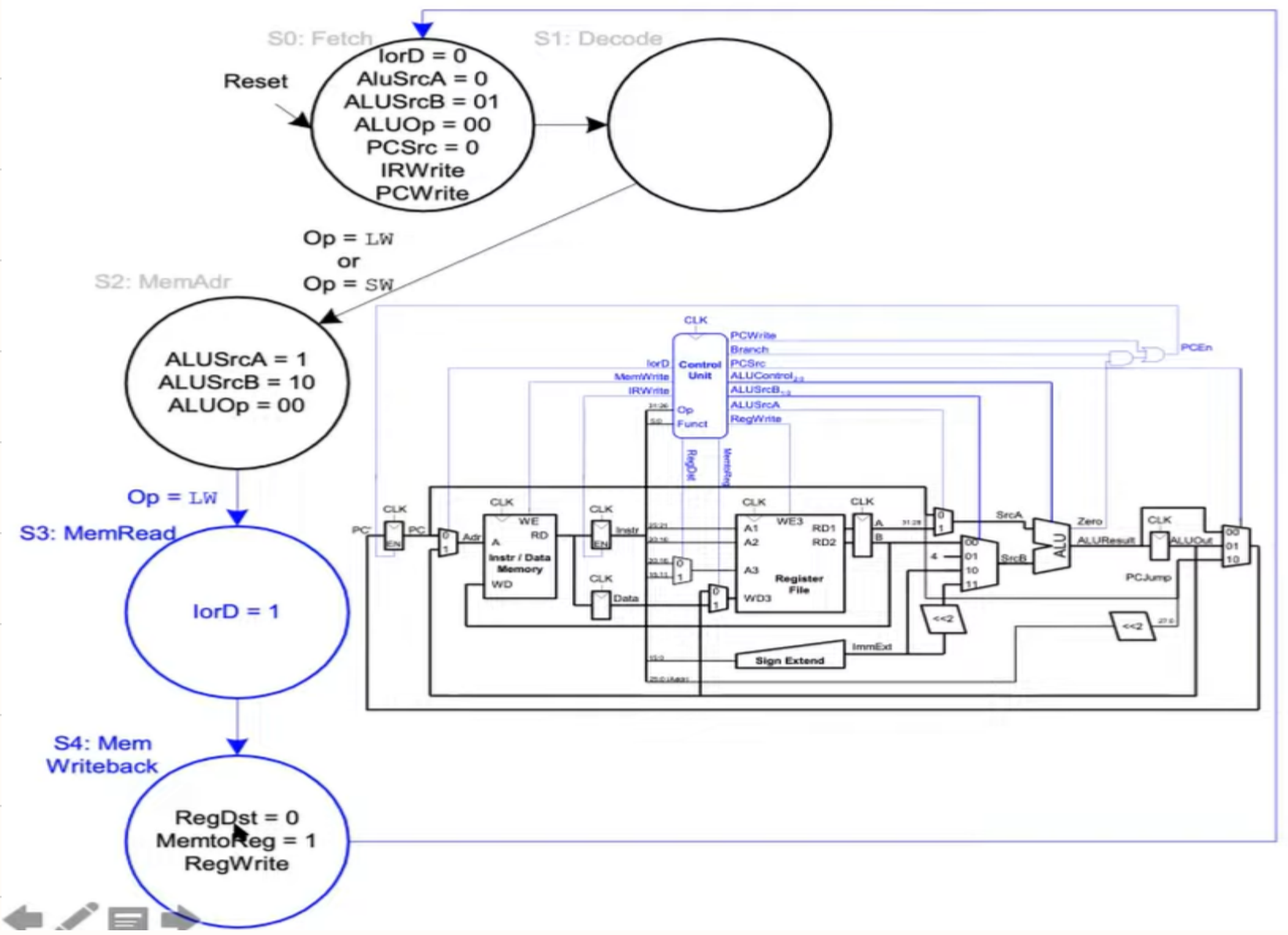




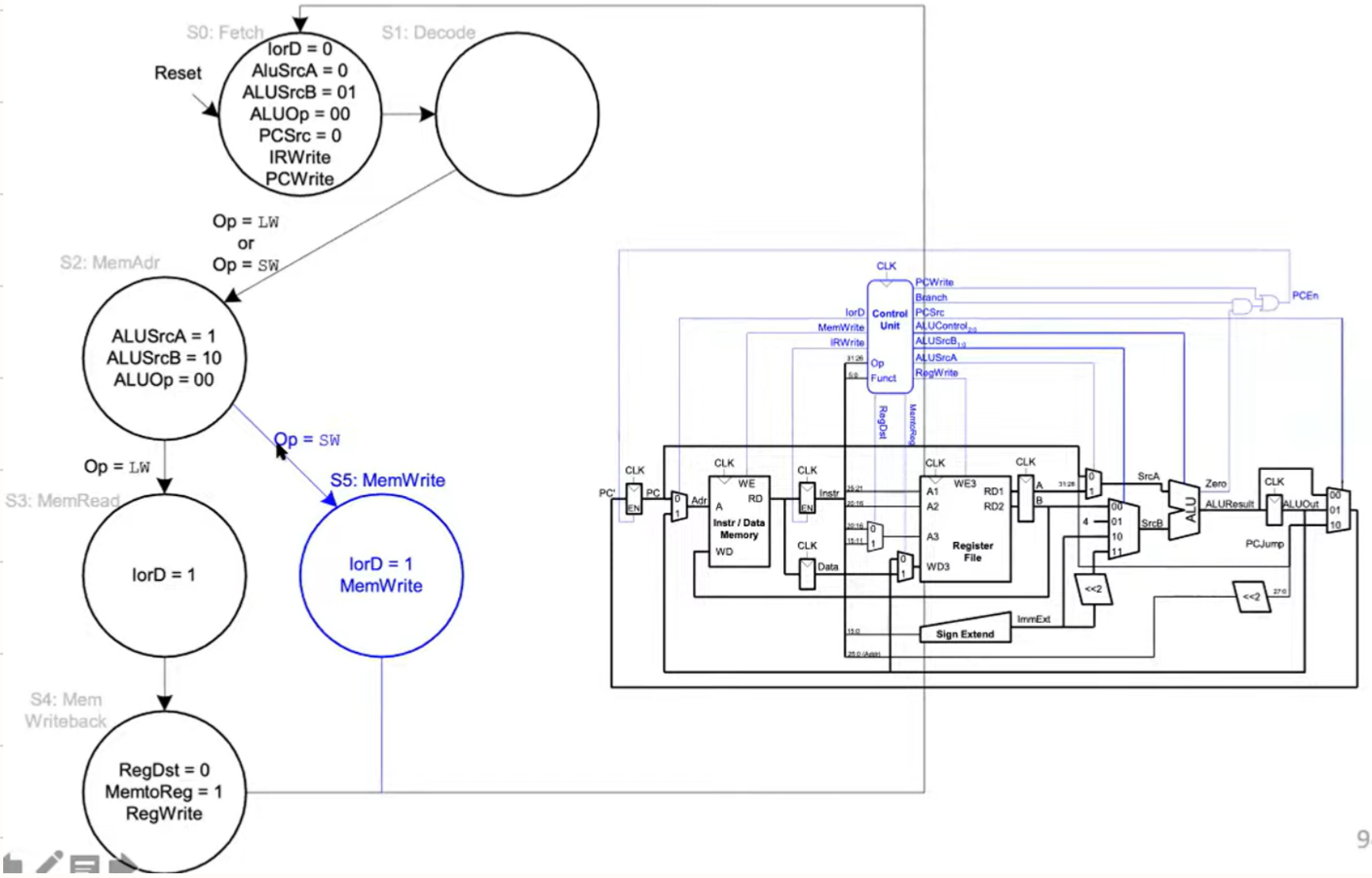
Main Controller FSM: Address Calculation



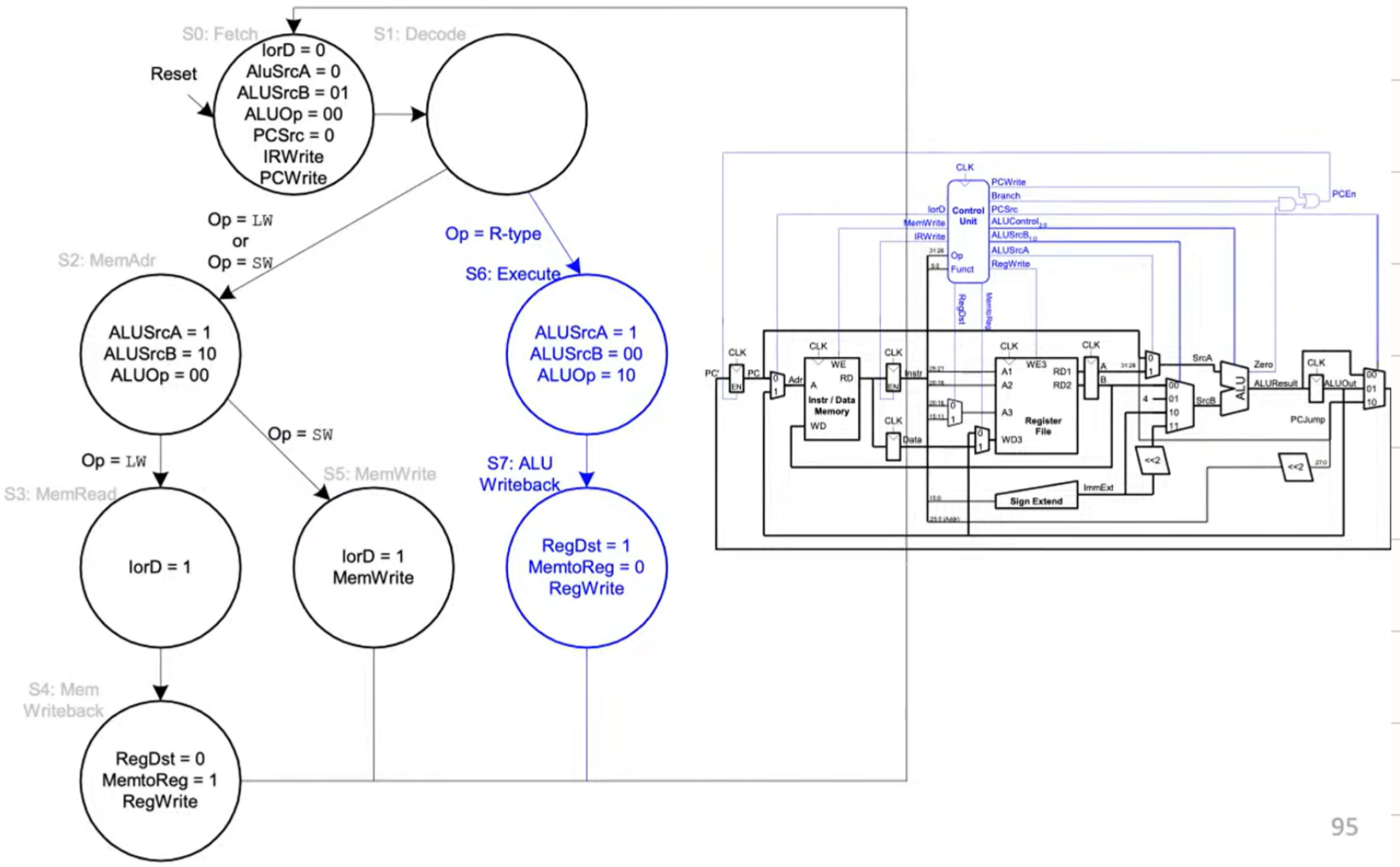
Main Controller FSM: lw



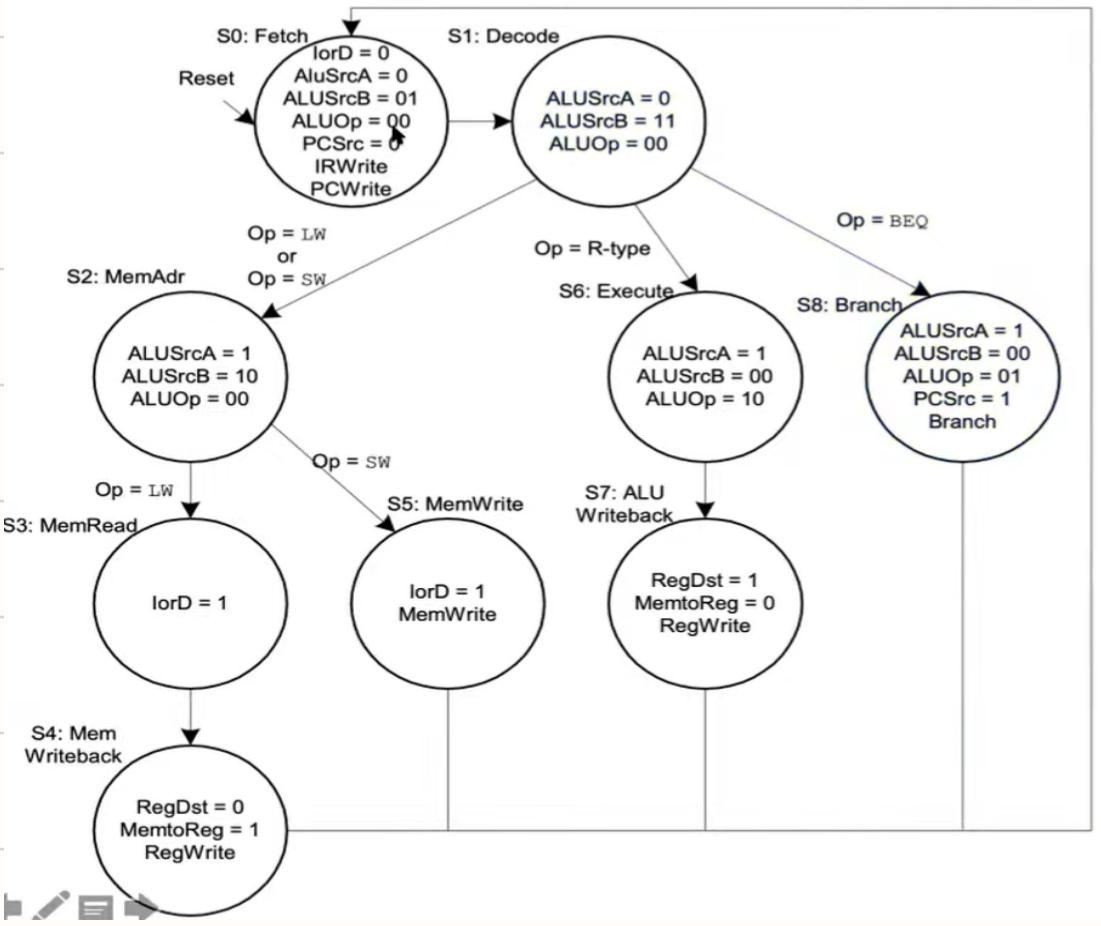
Main Controller FSM: sw

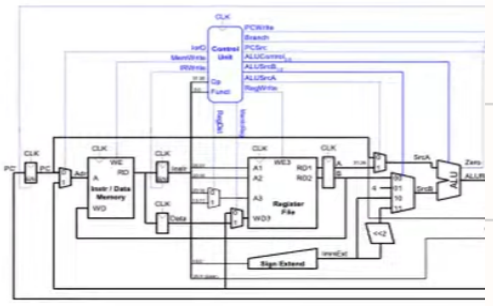


Main Controller FSM: R-Type

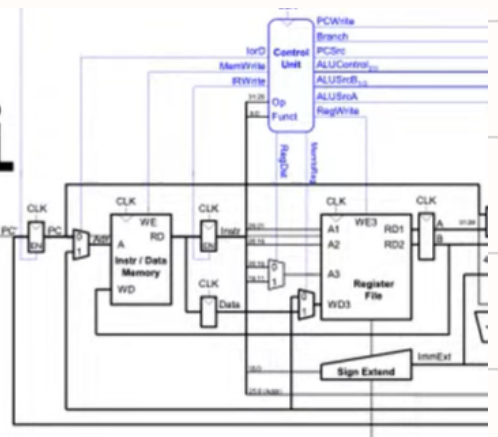


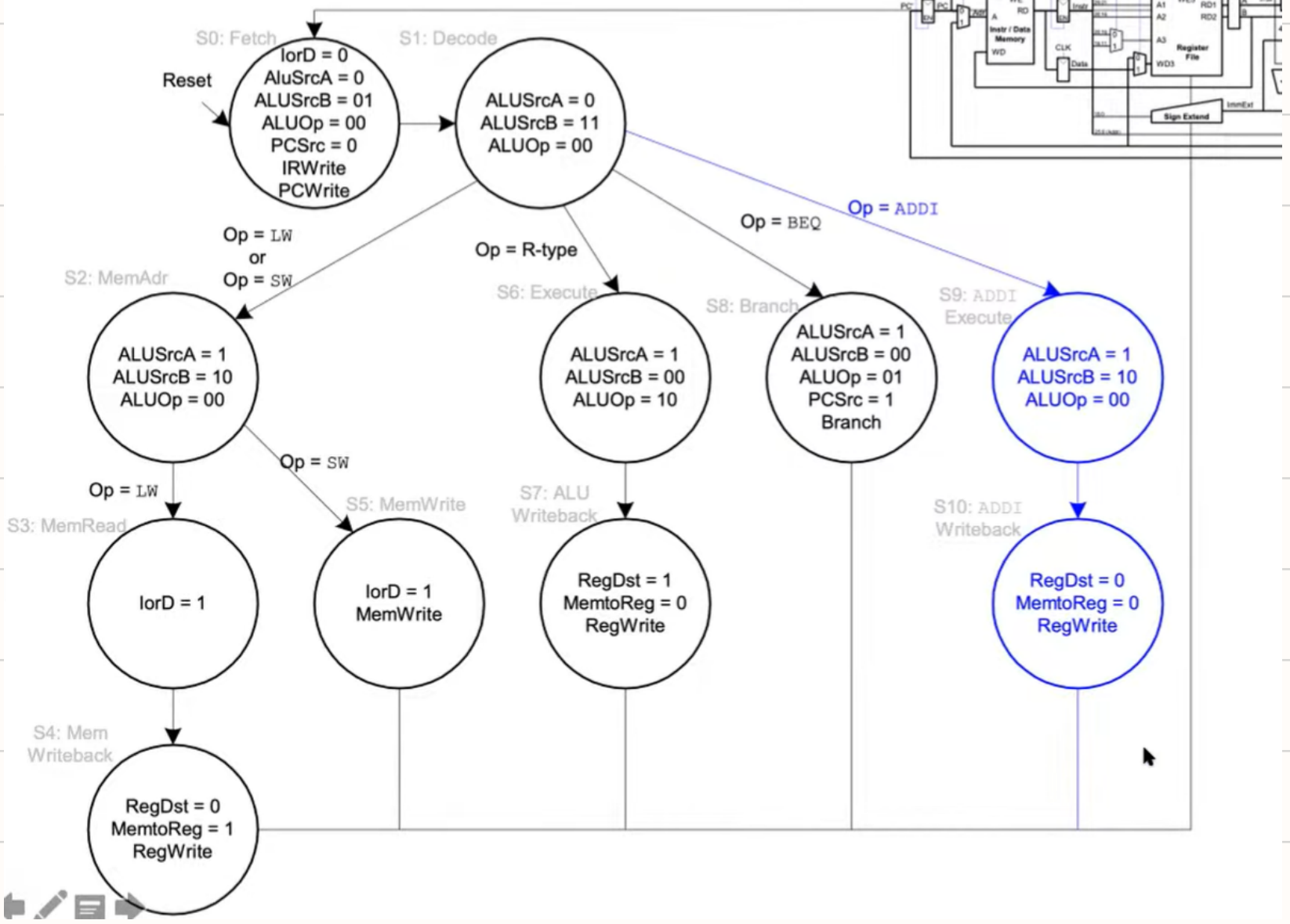
Complete Multi-Cycle Controller FSM



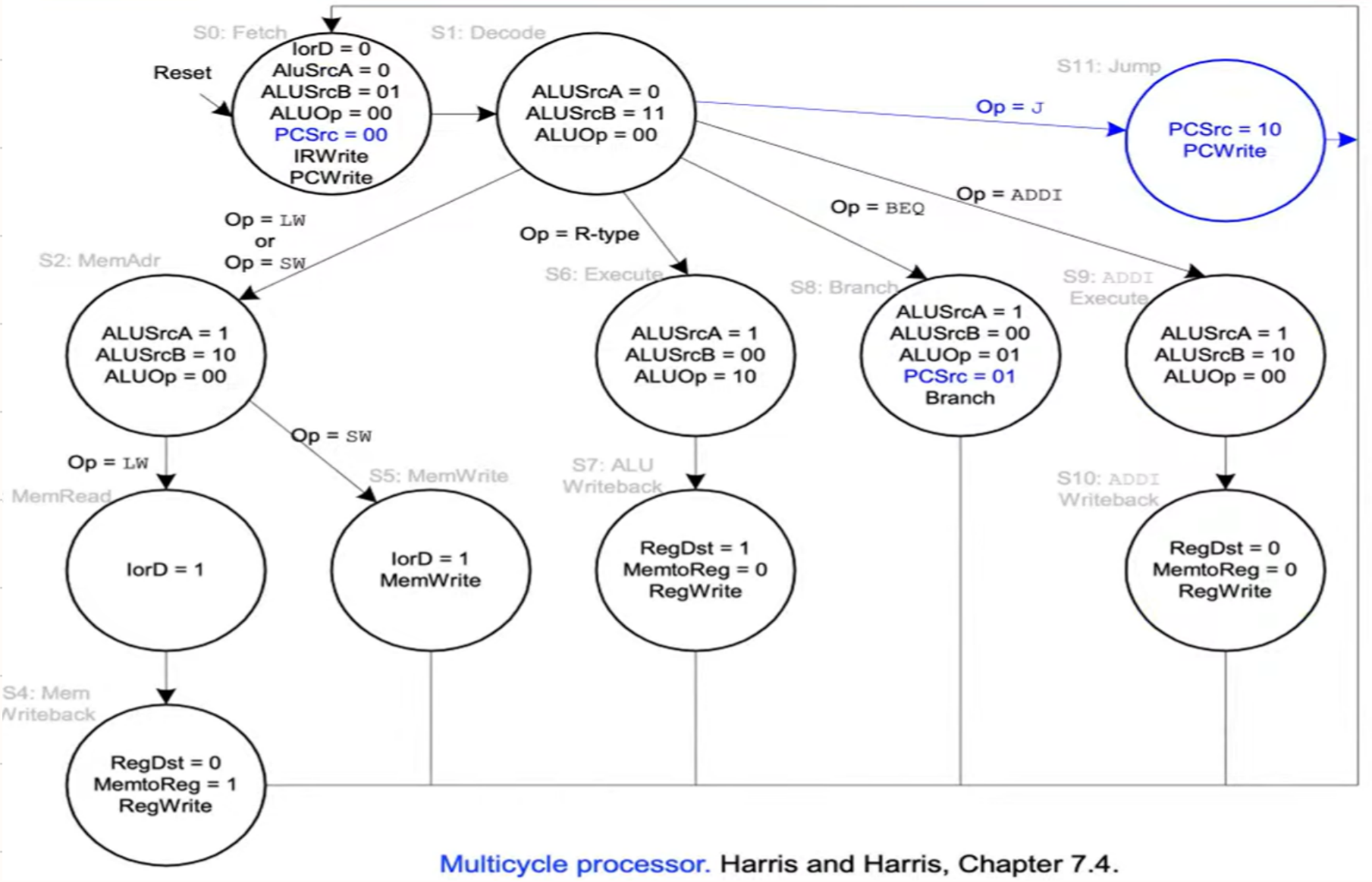


Main Controller FSM: addi



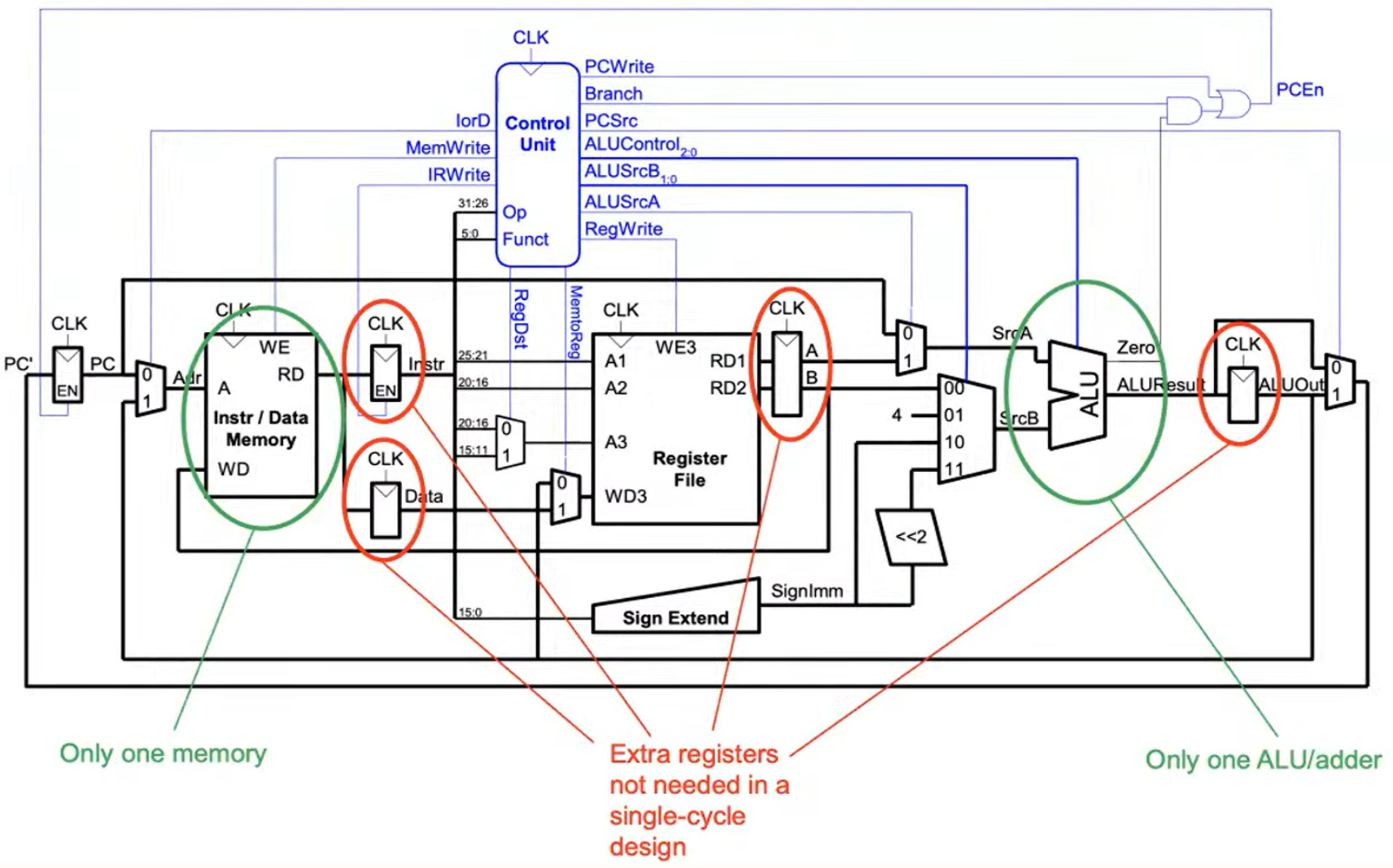


Review: Multi-Cycle MIPS FSM



Multicycle processor. Harris and Harris, Chapter 7.4.

Complete Multi-Cycle Processor



Microprogrammed Control Terminology

- Control signals associated with the current state
  - Microinstruction
- Act of transitioning from one state to another
  - Determining the next state and the microinstruction for the next state
  - Microsequencing
- Control store stores control signals for every possible state
  - Store for microinstructions for the entire FSM
- Microsequencer determines which set of control signals will be used in the next clock cycle (i.e., next state)

Microprogrammed Control Structure

- Three components Microinstruction, control store, microsequencer
- Microinstruction: control signals that control the datapath (26 of them) and help determine the next state (9 of them)
- Each microinstruction is stored in a *unique location* in the control store (a special memory structure)
  - *Unique location*: address of the state corresponding to the microinstruction
  - Each state in the FSM corresponds to one microinstruction
- Microsequencer determines the address of the next microinstruction (i.e., next state)

The Power of Abstraction

- The concept of a control store of microinstructions enables the hardware designer with a new abstraction: microprogramming

- The designer can translate any desired operation to a sequence of microinstructions
- All the designer needs to provide is
  - The sequence of microinstructions needed to implement the desired operation
  - The ability for the control logic to correctly sequence through the microinstructions
  - Any additional datapath elements and control signals needed (no need if the operation can be "translated" into existing control signals)

Advantages of Microprogrammed Control

- Allows a very simple design to do powerful computation by controlling the datapath (using a sequencer)
  - High-level ISA translated into microcode (sequence of u-instructions)
  - Microcode (u-code) enables a minimal datapath to emulate an ISA
  - Microinstructions can be thought of as a user-invisible ISA (u-ISA)

- Enables easy extensibility of the ISA
  - Can support a new instruction by changing the microcode
  - Can support complex instructions as a sequence of simple microinstructions (e.g., REP MOVS, MultiDimensional Array Updates)

- Enables update of machine behavior
  - A buggy implementation of an instruction can be fixed by changing the microcode in the field
    - Easier if datapath provides ability to do the same thing in different ways