Digital Design & Computer Arch.

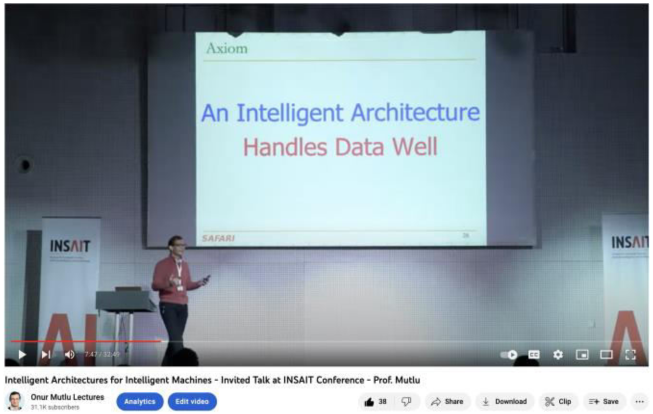
Lecture 11: Multi-Cycle Microarchitecture Design

Prof. Onur Mutlu

ETH Zürich
Spring 2023
30 March 2023

Extra Credit Assignment 1: Talk Analysis

- Intelligent Architectures for Intelligent Machines
- **Watch and analyze this short lecture (33 minutes)**
  - https://www.youtube.com/watch?v=WxHribseelw (Oct 2022)



Intelligent Architectures for Intelligent Machines - Invited Talk at INSAIT Conference - Prof. Mutlu

Onur Mutlu Lectures
31.1K subscribers

- **Assignment – for 1% extra credit**
  - **Write a good 1-page summary (following our guidelines)**
    - What are your key takeaways?
    - What did you learn?
    - What did you like or dislike?
    - Submit your summary to Moodle – deadline April 1

Extra Credit Assignment 2: Moore's Law

- **Paper review**
- G.E. Moore. "Cramming more components onto integrated circuits," Electronics magazine, 1965

- **Optional Assignment – for 1% extra credit**
  - **Write a 1-page review**
  - Upload PDF file to Moodle – Deadline: April 1

- I strongly recommend that you follow my guidelines for (paper) review (see next slide)

Extra Credit Assignment 2: Moore's Law

- **Guidelines on how to review papers critically**

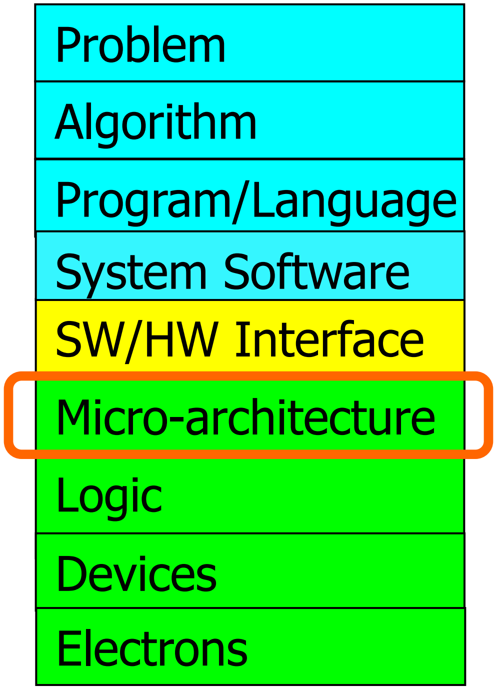
  - Guideline slides: pdf ppt
  - Video: https://www.youtube.com/watch?v=tOL6FANAJ8c

  - Example reviews on "Main Memory Scaling: Challenges and Solution Directions" (link to the paper)
    - Review 1
    - Review 2

  - Example review on "Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems" (link to the paper)
    - Review 1

Agenda for Today & Next Few Lectures

- Instruction Set Architectures (ISA): LC-3 and MIPS
- Assembly programming: LC-3 and MIPS
- Microarchitecture (principles & single-cycle uarch)
- Multi-cycle microarchitecture
- Pipelining
- Issues in Pipelining:
  - Control & Data Dependence Handling
  - State Maintenance and Recovery
- Out-of-Order Execution



Problem
Algorithm
Program/Language
System Software
SW/HW Interface
Micro-architecture
Logic
Devices
Electrons

Readings

- This week
  - Introduction to microarchitecture and single-cycle microarchitecture
    - H&H, Chapter 7.1-7.3
    - P&P, Appendices A and C
  - Multi-cycle microarchitecture
    - H&H, Chapter 7.4
    - P&P, Appendices A and C

- Also this week
  - Pipelining
    - H&H, Chapter 7.5
  - Pipelining Issues
    - H&H, Chapter 7.7, 7.8.1-7.8.3

Implementing the ISA: Microarchitecture Basics

Recall: A Very Basic Instruction Processing Engine

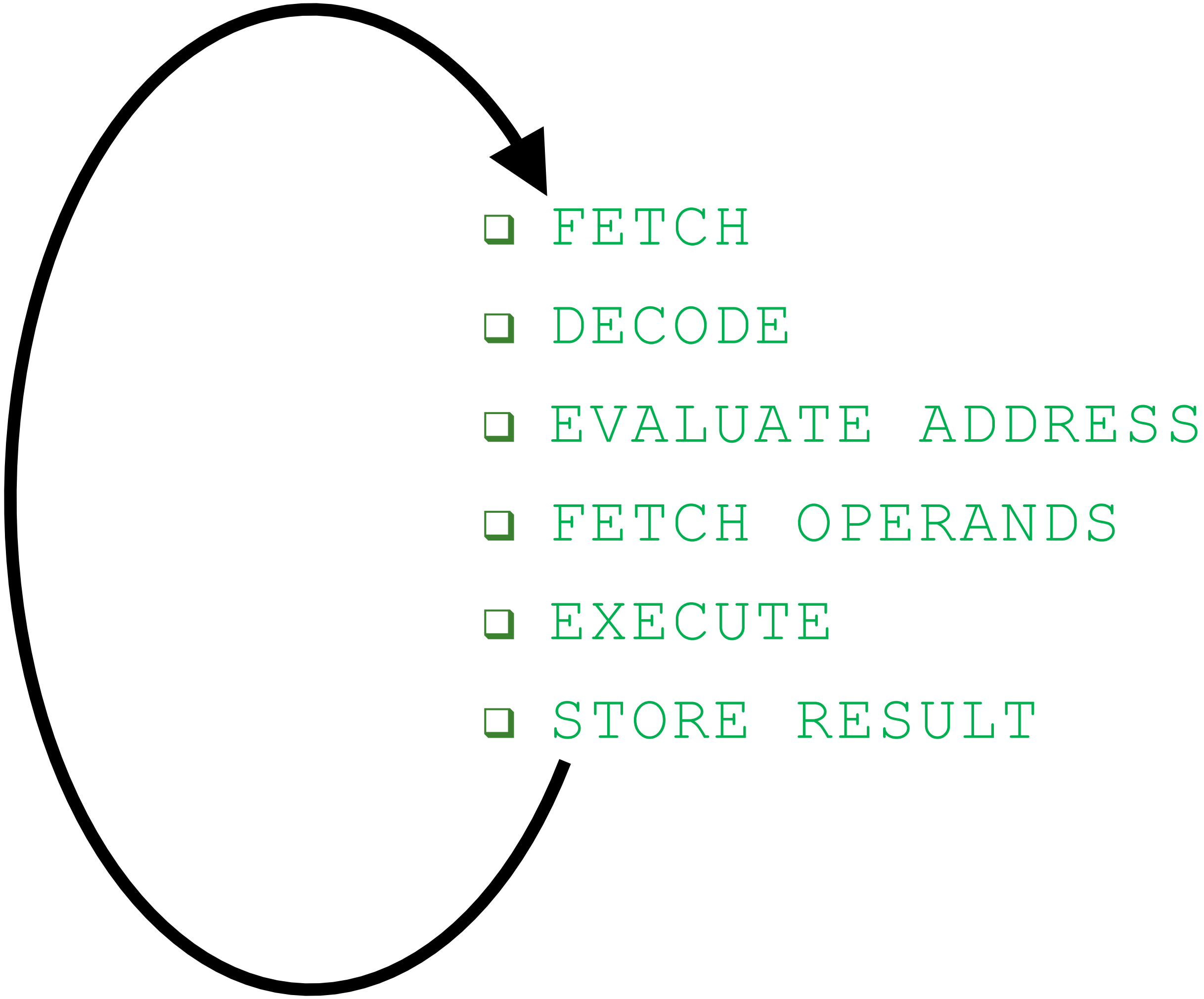
- Each instruction takes a single clock cycle to execute
- Only combinational logic is used to implement instruction execution
  - *No intermediate, programmer-invisible state updates*

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

Process instruction in **one clock cycle**

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

Recall: The Instruction Processing “Cycle”



- FETCH
- DECODE
- EVALUATE ADDRESS
- FETCH OPERANDS
- EXECUTE
- STORE RESULT

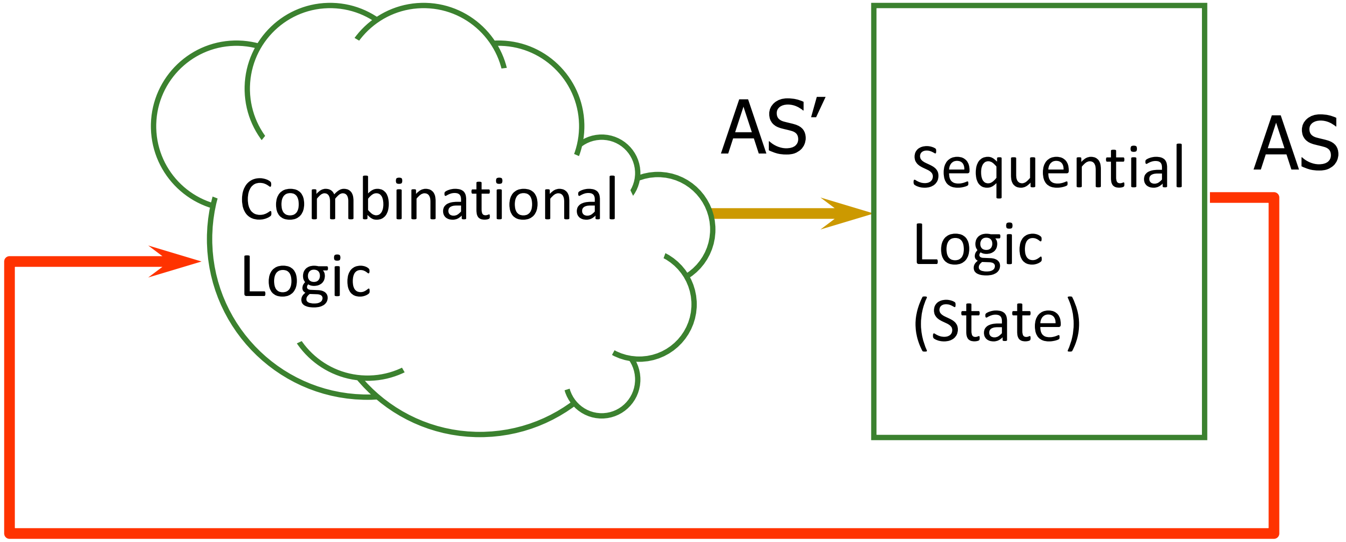
Instruction Processing "Cycle" vs. Machine Clock Cycle

- Single-cycle machine:
  - All six phases of the instruction processing cycle take a *single machine clock cycle* to complete

- Multi-cycle machine:
  - All six phases of the instruction processing cycle can take *multiple machine clock cycles* to complete
  - In fact, each phase can take multiple clock cycles to complete

Recall: Single-Cycle Machine

- Single-cycle machine



AS: Architectural State

Recall: Datapath and Control Logic

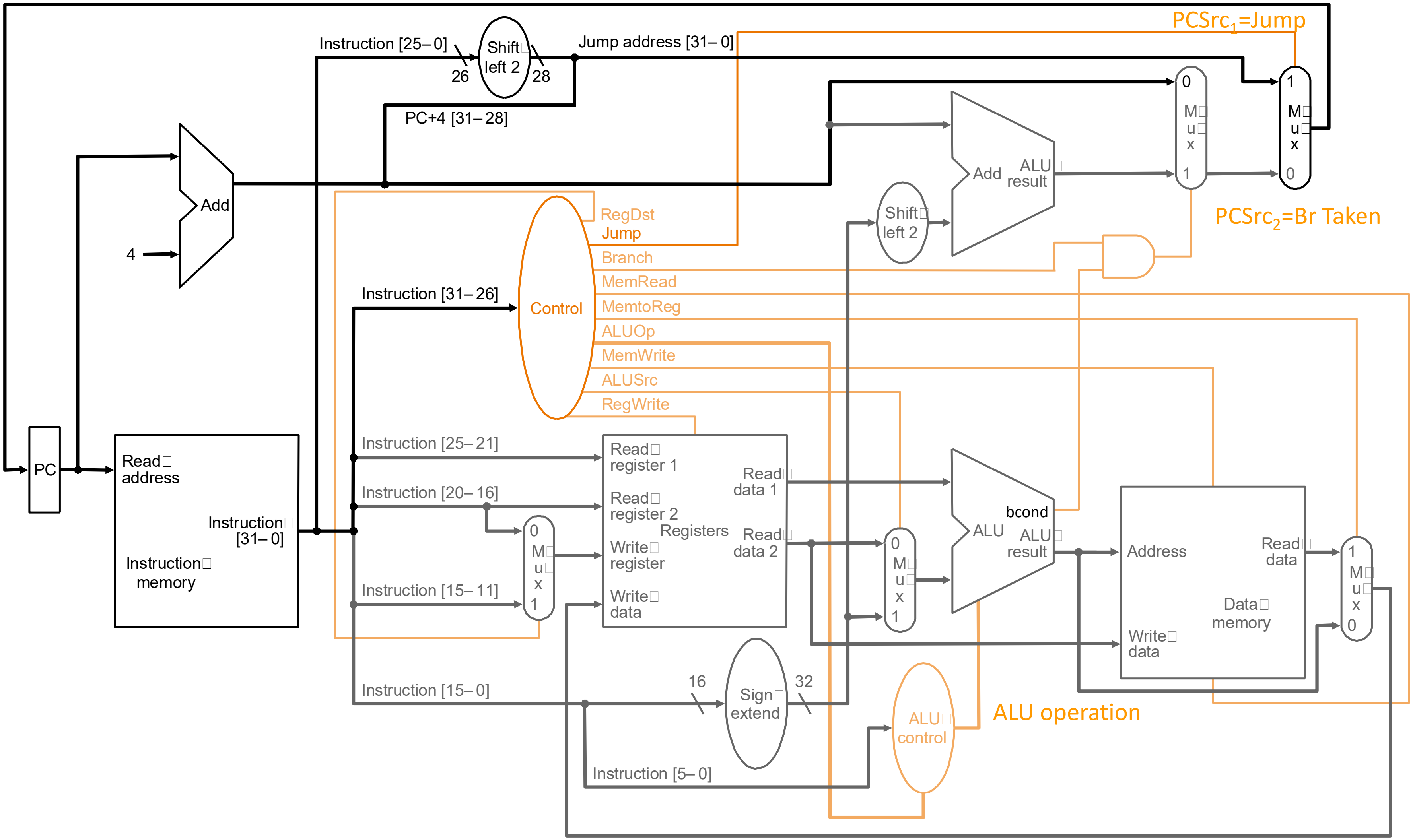
- An instruction processing engine consists of two components
  - Datapath: Consists of hardware elements that deal with and transform data signals
    - **functional units** that operate on data
    - **hardware structures** (e.g., wires, muxes, decoders, tri-state bufs) that enable the flow of data into the functional units and registers
    - **storage units** that store data (e.g., registers)
  - Control logic: Consists of hardware elements that determine control signals, i.e., signals that specify what the datapath elements should do to the data

A Single-Cycle Microarchitecture

From the Ground Up

Single-Cycle Control Logic

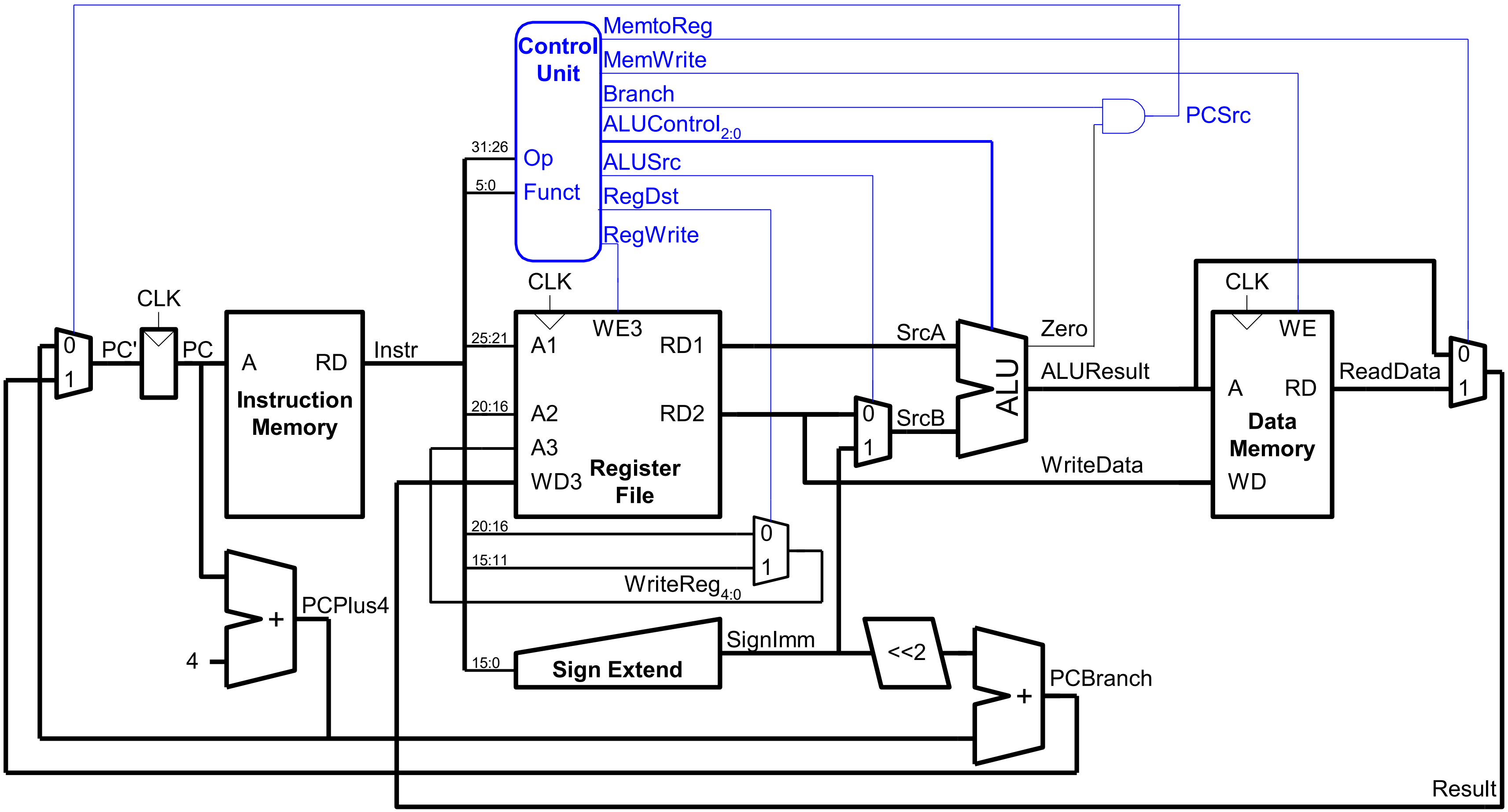
Single-Cycle Uarch I (We Developed in Lectures)



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

JAL, JR, JALR omitted

Single-Cycle Uarch II (In Your Readings)



Single-cycle processor. Harris and Harris, Chapter 7.3.

Evaluating the Single-Cycle Microarchitecture

A Single-Cycle Microarchitecture

- Is *this* a good idea/design?

- When is this a good design?

- When is this a bad design?

- How can we design a better microarchitecture?

Performance Analysis Basics

Recall: Performance Analysis Basics

- Execution time of a single instruction
  - **{CPI} x {clock cycle time}**
    - CPI: Number of cycles it takes to execute an instruction

- Execution time of an entire program
  - Sum over all instructions [{CPI} x {clock cycle time}]
  - **{# of instructions} x {Average CPI} x {clock cycle time}**

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

- **How fast are my instructions?**
  - Instructions are realized on the hardware
  - Each instruction 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

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

- **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

Processor Performance

- **As a general formula**
  - Our program consists of executing **N** instructions
  - Our processor needs **CPI** cycles (on average) for each instruction
  - The clock frequency of the processor is **f**

    → the clock period is therefore **T**=1/f

Processor Performance

- **As a general formula**
  - Our program consists of executing **N** instructions
  - Our processor needs **CPI** cycles (on average) for each instruction
  - The clock frequency of the processor is **f**
    → the clock period is therefore **T**=1/f

- **Our program executes in**

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

$$N \times CPI \times T \text{ seconds}$$

Performance Analysis of Our Single-Cycle Design

A Single-Cycle Microarchitecture: Analysis

- Every instruction takes 1 cycle to execute
  - CPI (Cycles per instruction) is strictly 1

- How long each instruction takes is determined by how long the slowest instruction takes to execute
  - Even though many instructions do not need that long to execute

- Clock cycle time of the microarchitecture is determined by how long it takes to complete the slowest instruction
  - Critical path of the design is determined by the processing time of the slowest instruction

What is the Slowest Instruction to Process?

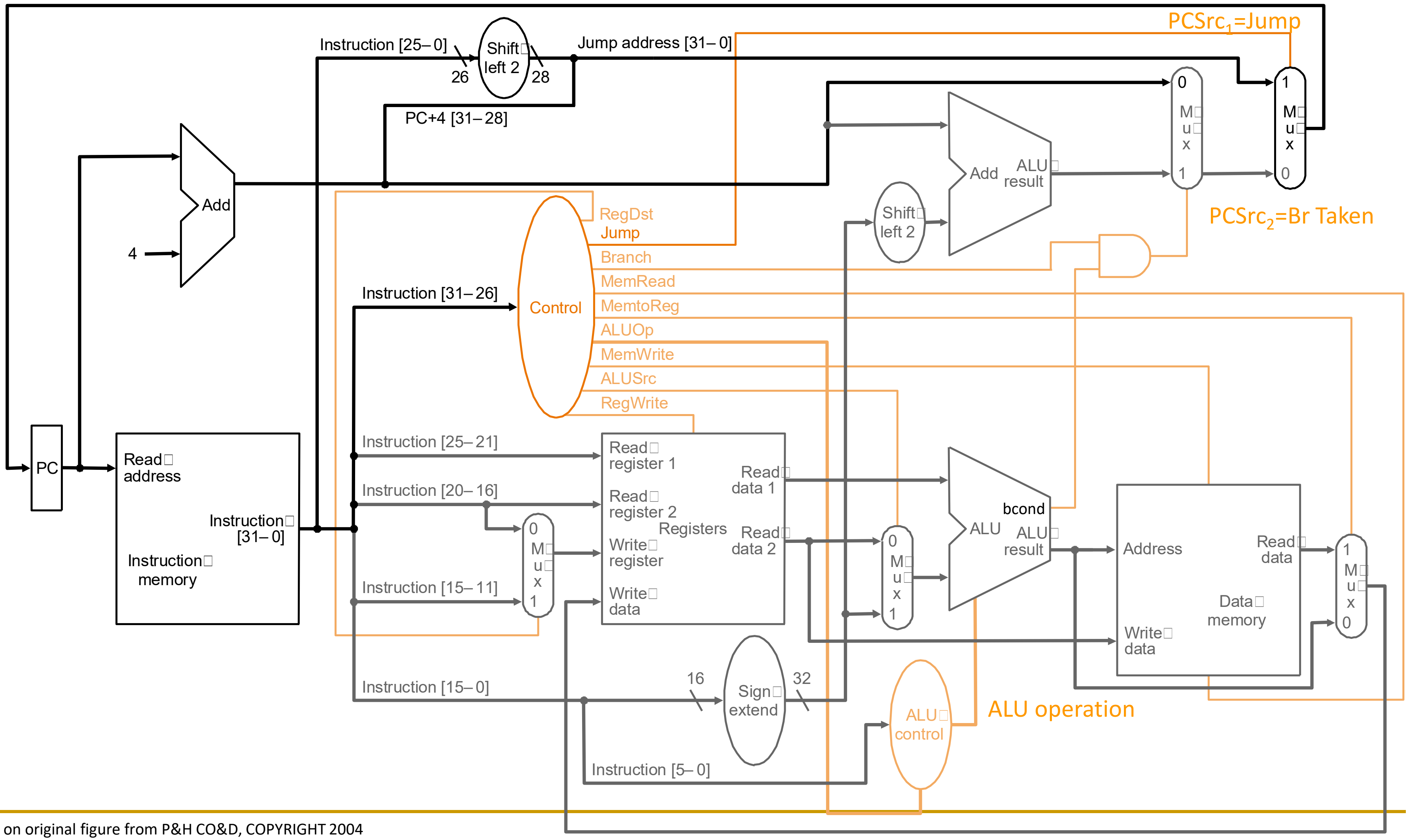
- Let’s go back to the basics

- All six phases of the instruction processing cycle take a *single machine clock cycle* to complete
  - ☐ Fetch
  - ☐ Decode
  - ☐ Evaluate Address
  - ☐ Fetch Operands
  - ☐ Execute
  - ☐ Store Result

1. Instruction fetch (IF)
2. Instruction decode and register operand fetch (ID/RF)
3. Execute/Evaluate memory address (EX/AG)
4. Memory operand fetch (MEM)
5. Store/writeback result (WB)

- Does every instruction take the same time (latency) to complete?

Let’s Find the Critical Path



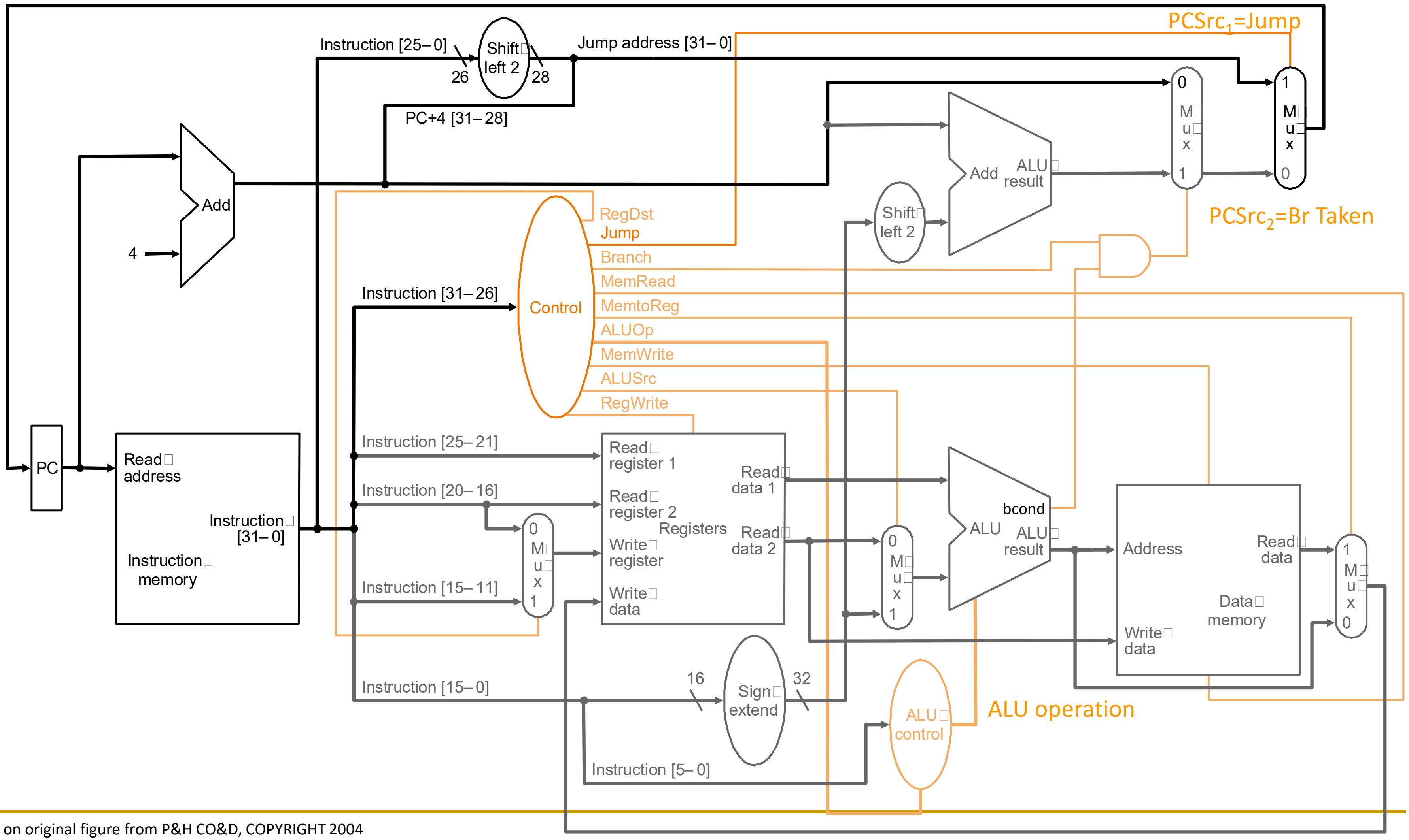
[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

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

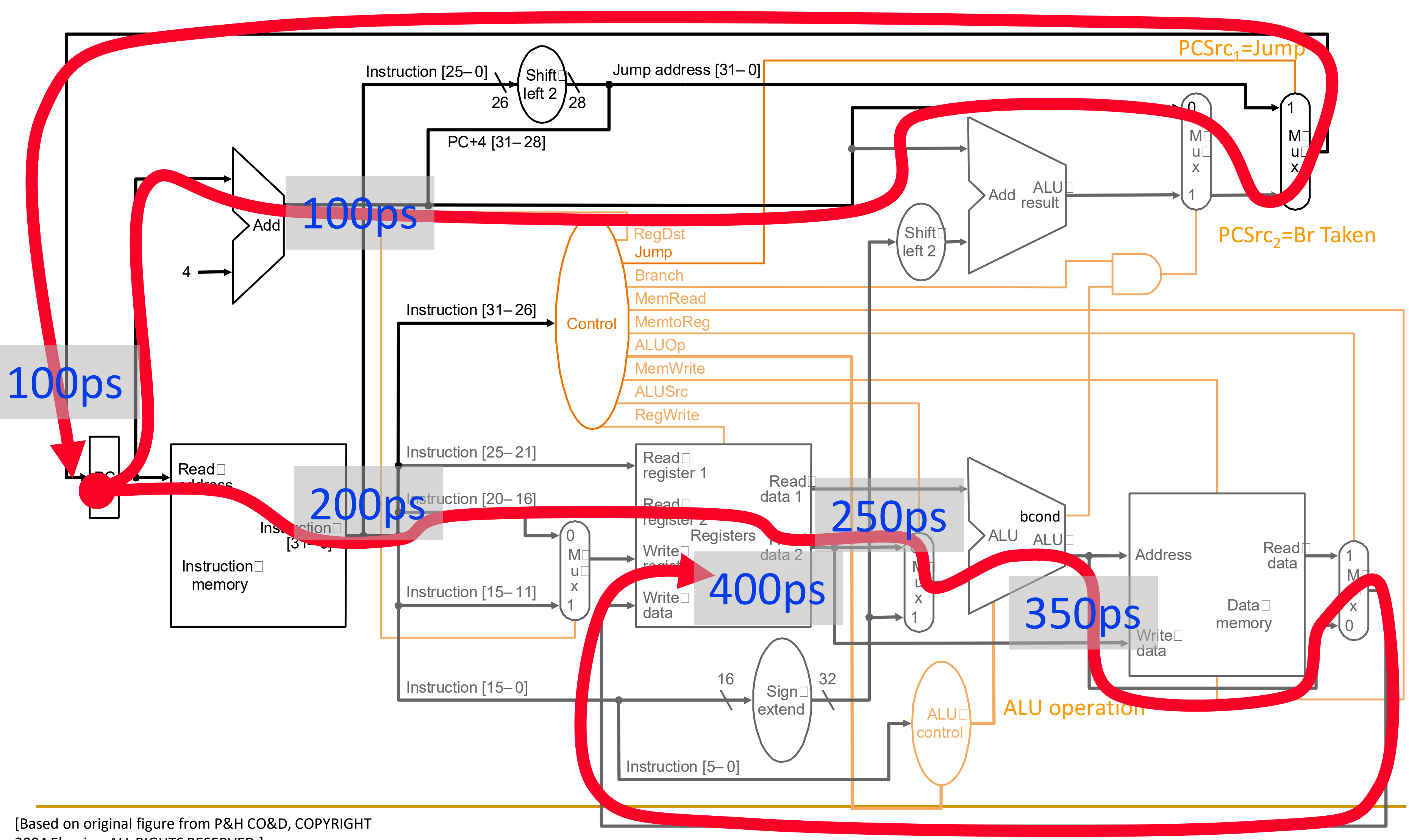
steps	IF	ID	EX	MEM	WB	Delay
resources	mem	RF	ALU	mem	RF	
R-type	200	50	100		50	400
I-type	200	50	100		50	400
LW	200	50	100	200	50	600
SW	200	50	100	200		550
Branch	200	50	100			350
Jump	200					200

Let’s Find the Critical Path



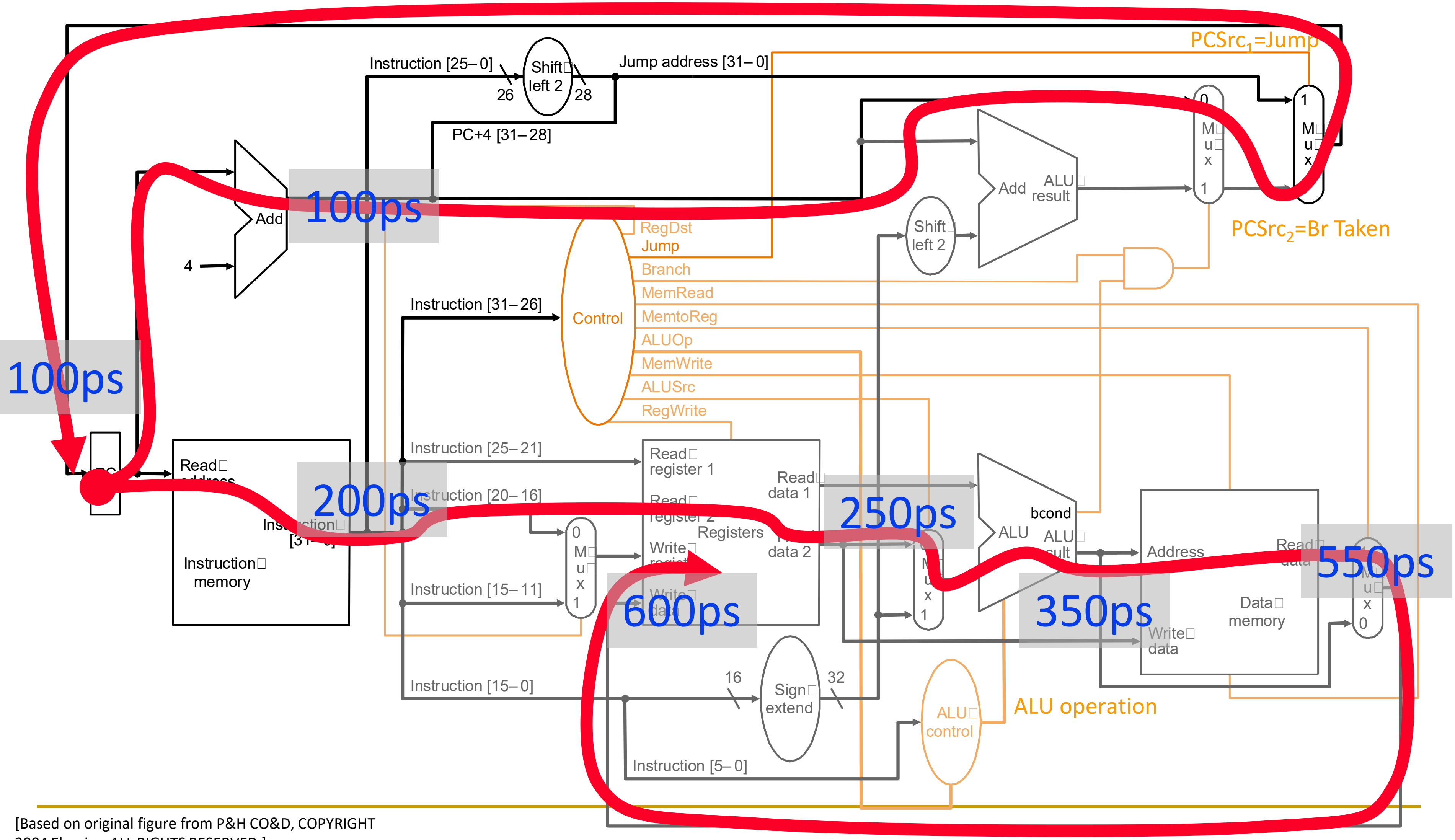
[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

R-Type and I-Type ALU



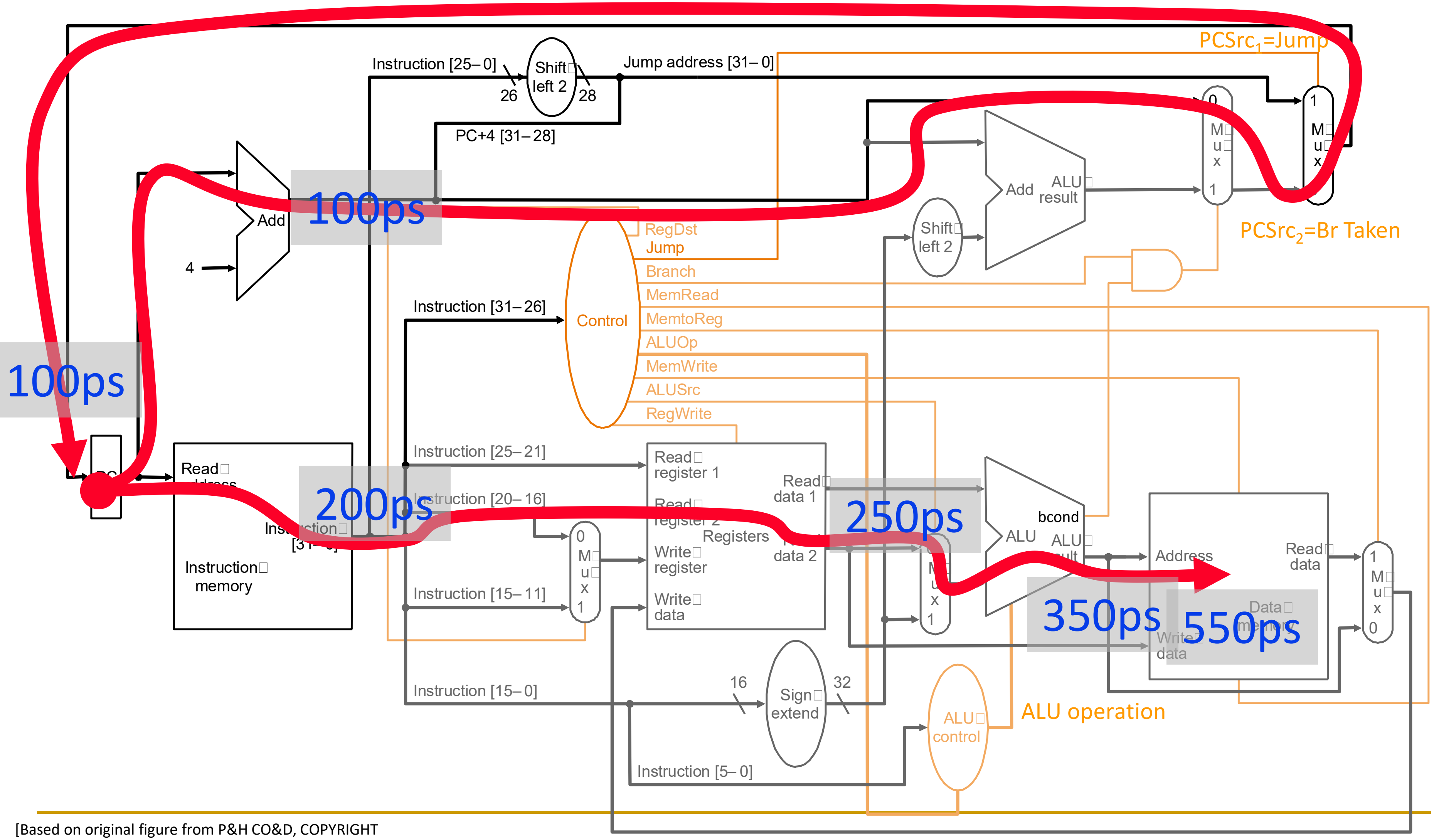
[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

LW



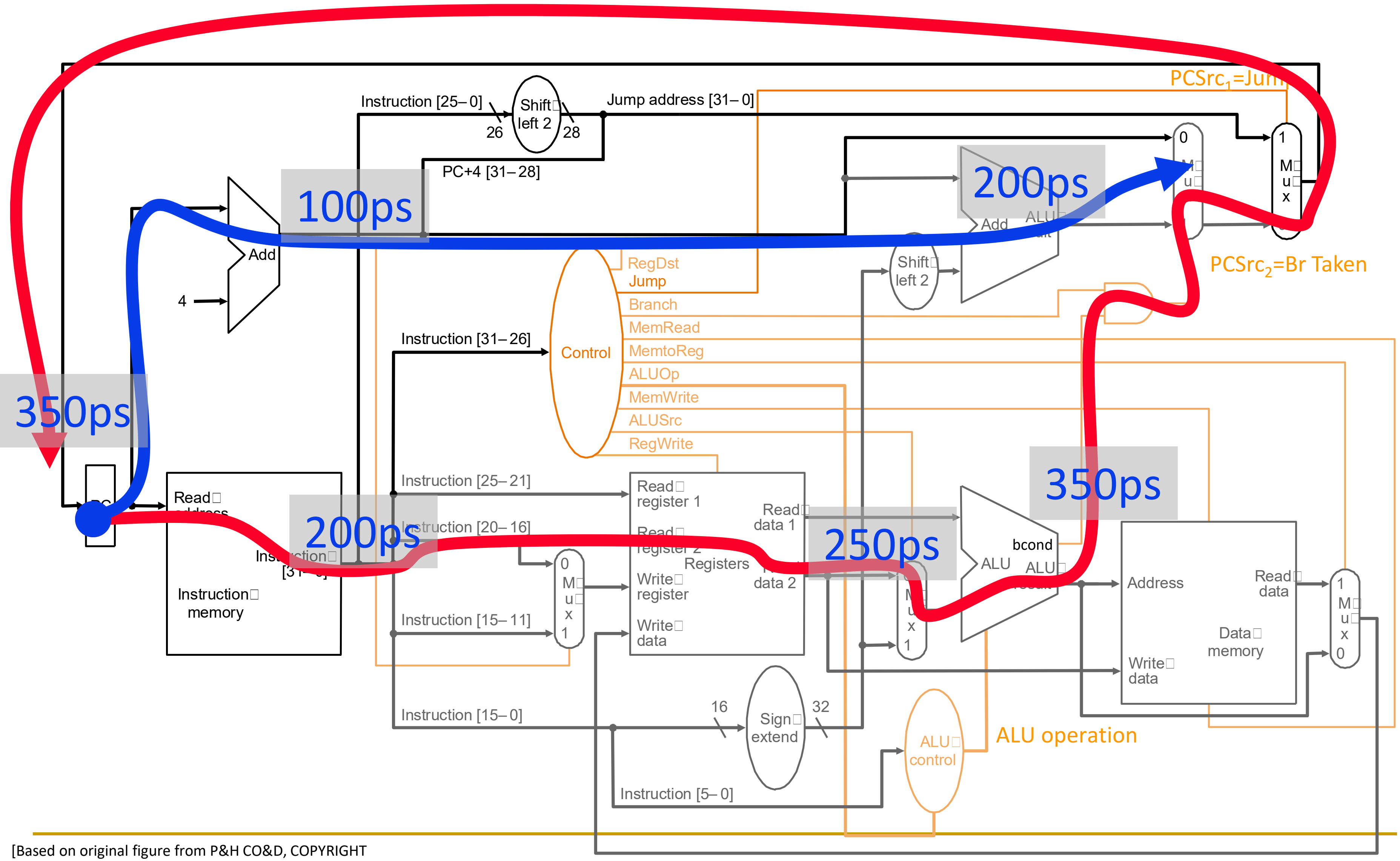
[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

SW



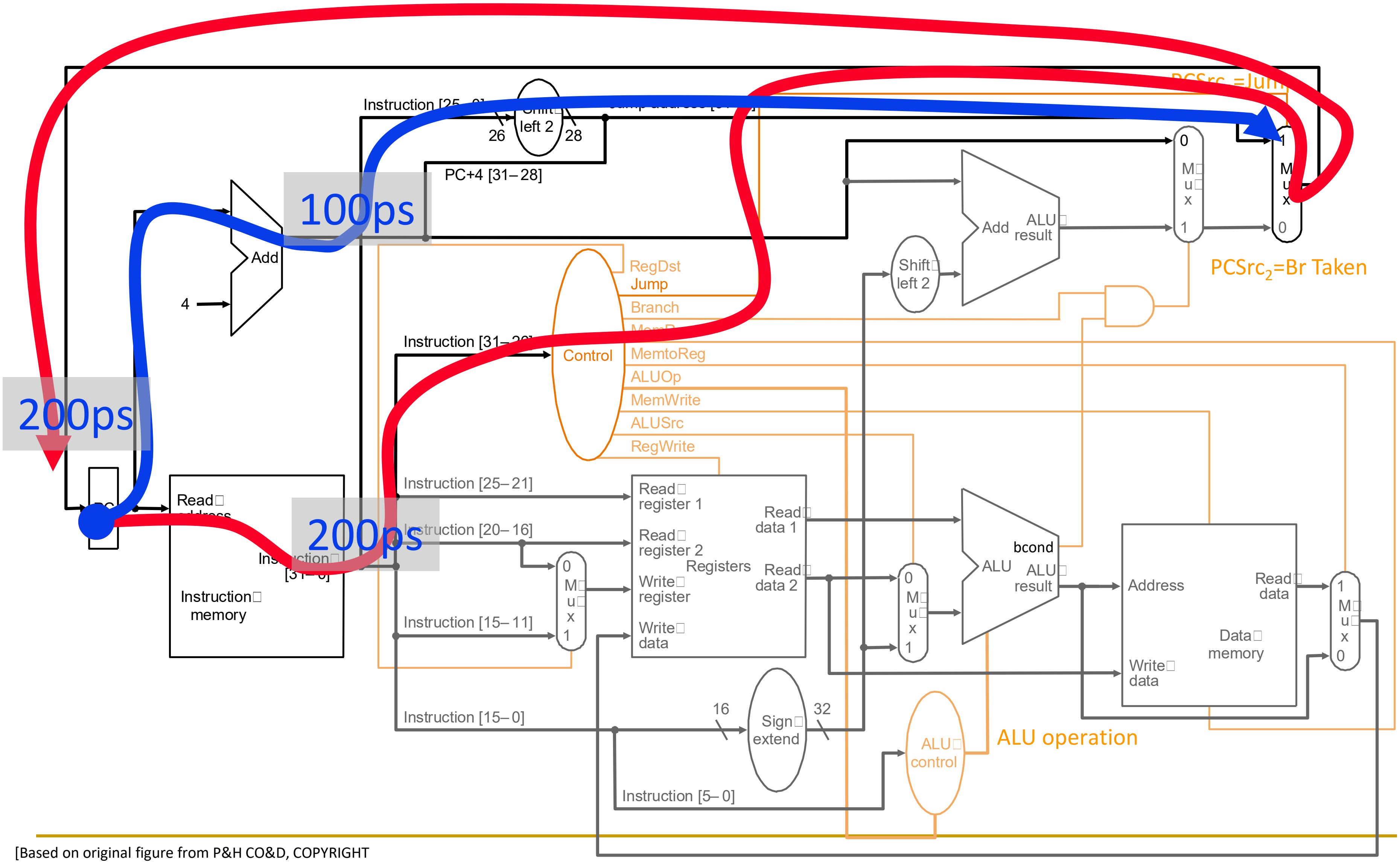
[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

Branch Taken



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

Jump



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

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

What About Control Logic?

- How does that affect the critical path?

- Food for thought for you:
  - Can control logic be on the critical path?
  - Historical example:
    - CDC 5600: control store access took too long…

What is Really the Slowest Instruction to Process?

- Real world: **Memory is slow (not magic)**
- What if memory *sometimes* takes 150ns to access?
- Does it make sense to have a simple register to register add or jump to take {150ns + all else to perform a memory operation}?
- And, what if you need to access memory more than once to process an instruction?
  - Which instructions require this?
  - Do you provide multiple ports to memory?

Single Cycle uArch: Complexity

- Contrived
  - All instructions run as slow as the slowest instruction

- Inefficient
  - All instructions run as slow as the slowest instruction
  - Must provide worst-case combinational resources in parallel as required by any instruction
  - Need to replicate a resource if it is needed more than once by an instruction during different parts of the instruction processing cycle

- Not necessarily the simplest way to implement an ISA
  - Tough for complex instructions, e.g., REP MOVS (x86) or INDEX (VAX)

- Not easy to optimize/improve performance
  - Optimizing the common case (frequent instructions) does not work
  - Need to optimize the worst case all the time

(Micro)architecture Design Principles

- Critical path design
  - Find and decrease the maximum combinational 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
    - i.e., improve what the machine is really designed to do
  - Common case vs. uncommon case

- Balanced design
  - Balance instruction/data flow through hardware components
  - Design to eliminate bottlenecks: balance the hardware for the work

Single-Cycle Design vs. Design Principles

- Critical path design

- Bread and butter (common case) design

- Balanced design

How does a single-cycle microarchitecture fare with respect to these principles?

Aside: System Design Principles

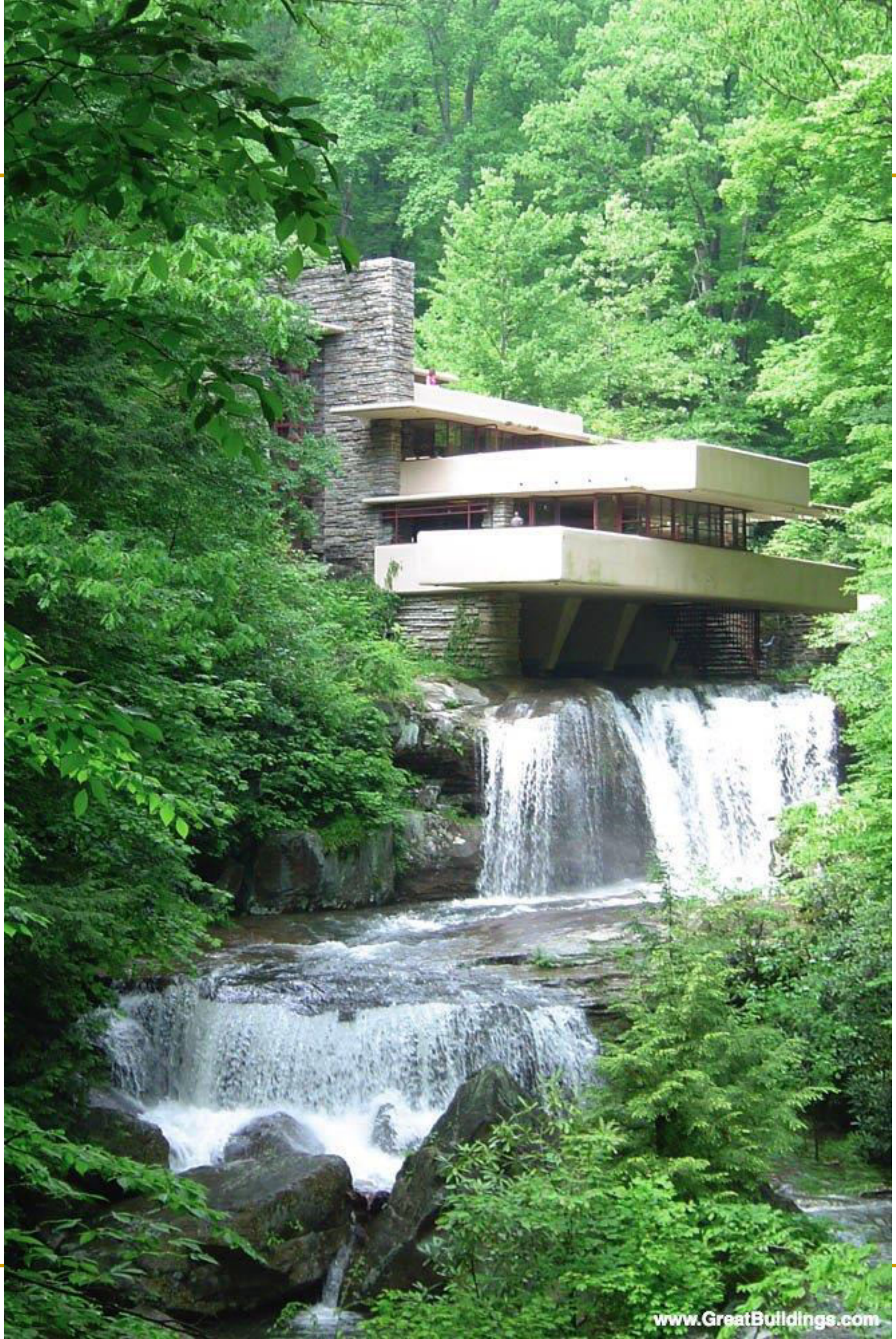
- When designing computer systems/architectures, it is important to follow good principles
    - Actually, this is true for *any* system design
        - Real architectures, buildings, bridges, train stations, …
        - Good consumer products
        - Security & safety-critical systems
        - Decision making systems
        - …

- Remember: “principled design” from our second lecture
    - Frank Lloyd Wright: “architecture […] based upon principle, and not upon precedent”

Aside: From Lecture 2

- “architecture […] based upon principle, and not upon precedent”

This



That

Recall: Takeaways

- It all starts from the basic building blocks and design principles

- And, knowledge of how to use, apply, enhance them

- Underlying technology might change (e.g., steel vs. wood)
    - but methods of taking advantage of technology bear resemblance
    - methods used for design depend on the principles employed

Aside: System Design Principles

- We will continue to cover key principles in this course
- Here are some references where you can learn more

- Yale Patt, "Requirements, Bottlenecks, and Good Fortune: Agents for Microprocessor Evolution," Proc. of IEEE, 2001. (Levels of transformation, design point, etc)

- Mike Flynn, "Very High-Speed Computing Systems," Proc. of IEEE, 1966. (Flynn's Bottleneck → Balanced design)

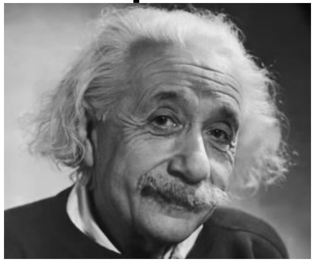
- Gene M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS Conference, April 1967. (Amdahl's Law → Common-case design)

- Butler W. Lampson, "Hints for Computer System Design," ACM Operating Systems Review, 1983.

A Key System Design Principle

- **Keep it simple**

- "Everything should be made as simple as possible, but no simpler."
  - Albert Einstein (paraphrased)



- And, **keep it low cost**: "An engineer is a person who can do for a dime what any fool can do for a dollar."

- For more, see:
  - Butler W. Lampson, "Hints for Computer System Design," ACM Operating Systems Review, 1983.
  - http://research.microsoft.com/pubs/68221/acrobat.pdf

Can We Do Better?

Multi-Cycle Microarchitectures

Multi-Cycle Microarchitectures

- 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

Recall: The "Process Instruction" Step

- ISA specifies abstractly what AS' should be, given an instruction and AS
    - It defines an **abstract finite state machine** where
        - State = programmer-visible state
        - Next-state logic = instruction execution specification
    - From ISA point of view, there are no "intermediate states" between AS and AS' during instruction execution
        - One state transition per instruction

- Microarchitecture implements how AS is transformed to AS'
    - There are many choices in implementation
    - We can have programmer-invisible state to optimize the speed of instruction execution: **multiple** state transitions per instruction
        - Choice 1: AS → AS' (transform AS to AS' in a single clock cycle)
        - Choice 2: AS → AS+MS1 → AS+MS2 → AS+MS3 → AS' (take multiple clock cycles to transform AS to AS')

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

Recall: Control of the Instruction Cycle

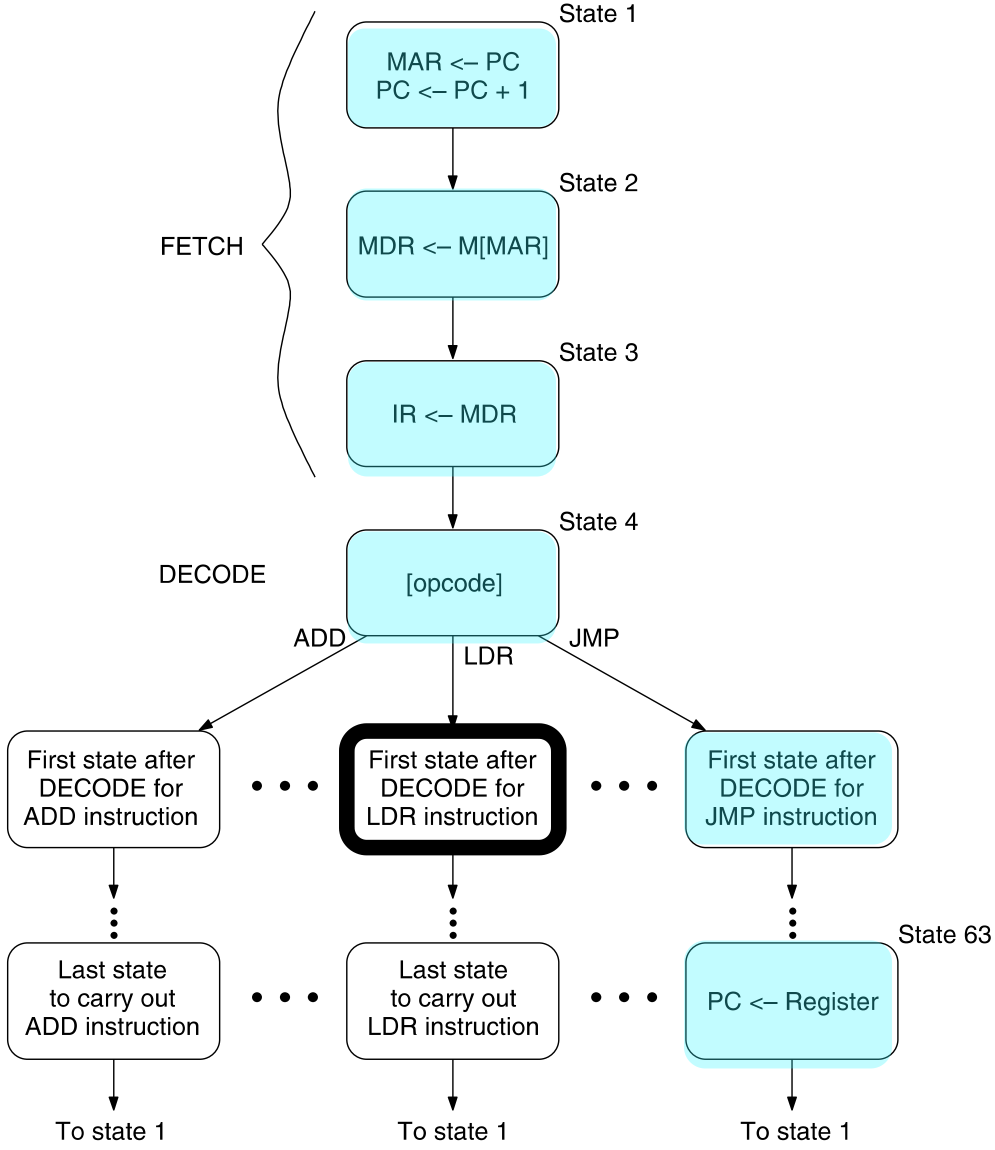
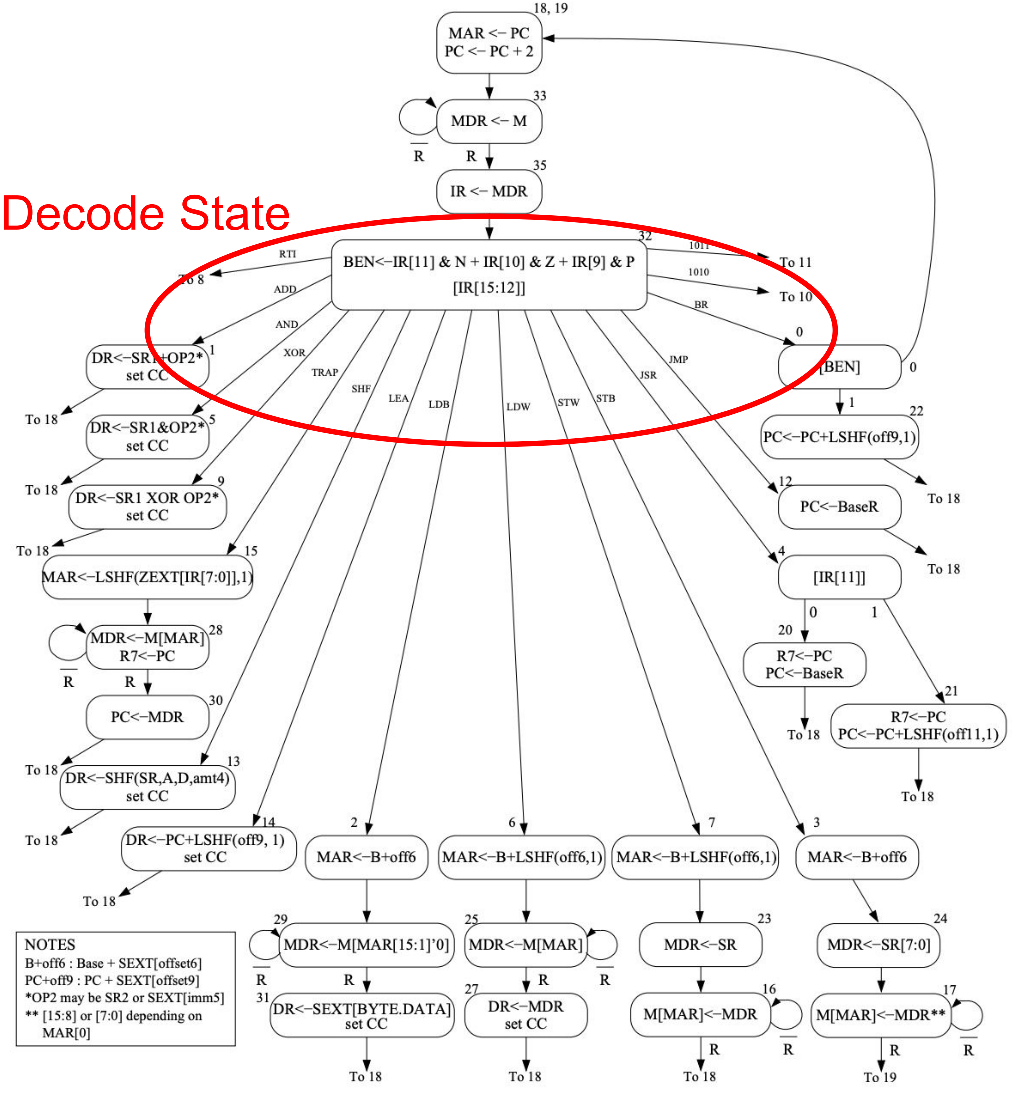


Figure 4.4 An abbreviated state diagram of the LC-3

- State 1
    - The FSM asserts GatePC and LD.MAR
    - It selects input (+1) in PCMUX and asserts LD.PC
- State 2
    - MDR is loaded with the instruction
- State 3
    - The FSM asserts GateMDR and LD.IR
- State 4
    - The FSM goes to next state depending on opcode
- State 63
    - JMP loads register into PC
- Full state diagram in Patt&Pattel, Appendix C

This is an FSM Controlling a Multi-Cycle LC-3 Microarchitecture

Recall: Full State Machine for LC-3b



Decode State

Figure C.2: A state machine for the LC-3b

Full FSM Controlling a Multi-Cycle LC-3b Microarchitecture

https://safari.ethz.ch/digitaltechnik/spring2022/lib/exe/fetch.php?media=pp-appendixc.pdf

Benefits of Multi-Cycle Design

- Critical path design
  - Can keep reducing the critical path independently of the worst-case processing time of any instruction

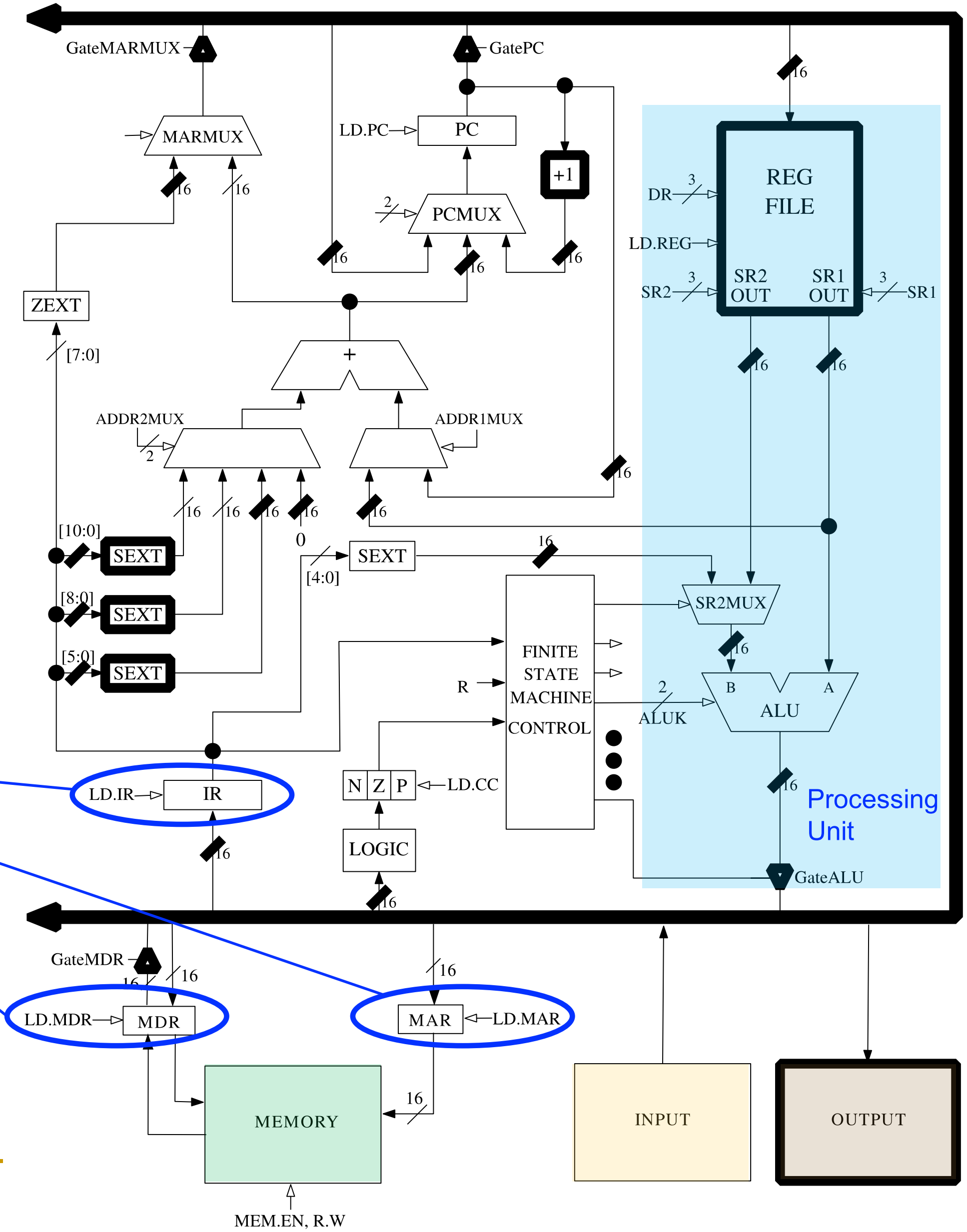
- Bread and butter (common case) design
  - Can optimize the number of states it takes to execute "important" instructions that make up much of the execution time

- 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 of Multi-Cycle Design

- Need to store the intermediate results at the end of each clock cycle
  - Hardware overhead for microarchitectural registers
  - Register setup/hold overhead (i.e., sequencing overhead) is paid multiple times for an instruction

Multi-Cycle LC-3 Data Path



Extra registers not needed in a single-cycle design

Remember: Performance Analysis

- Execution time of a single instruction
  - **{CPI} x {clock cycle time}**

CPI: Cycles Per Instruction

- Execution time of an entire program
  - Sum over all instructions [{CPI} x {clock cycle time}]
  - **{# of instructions} x {Average CPI} x {clock cycle time}**

- Single-cycle microarchitecture performance
  - CPI = 1
  - Clock cycle time = long
- Multi-cycle microarchitecture performance
  - CPI = different for each instruction
    - Average CPI → hopefully small
  - Clock cycle time = short

In multi-cycle, we have two degrees of freedom to optimize independently

A Multi-Cycle Microarchitecture

A Closer Look

An Elegant Multi-Cycle Processor Design

- Maurice Wilkes, "The Best Way to Design an Automatic Calculating Machine," Manchester Univ. Computer Inaugural Conf., 1951.

THE BEST WAY TO DESIGN AN AUTOMATIC
CALCULATING MACHINE

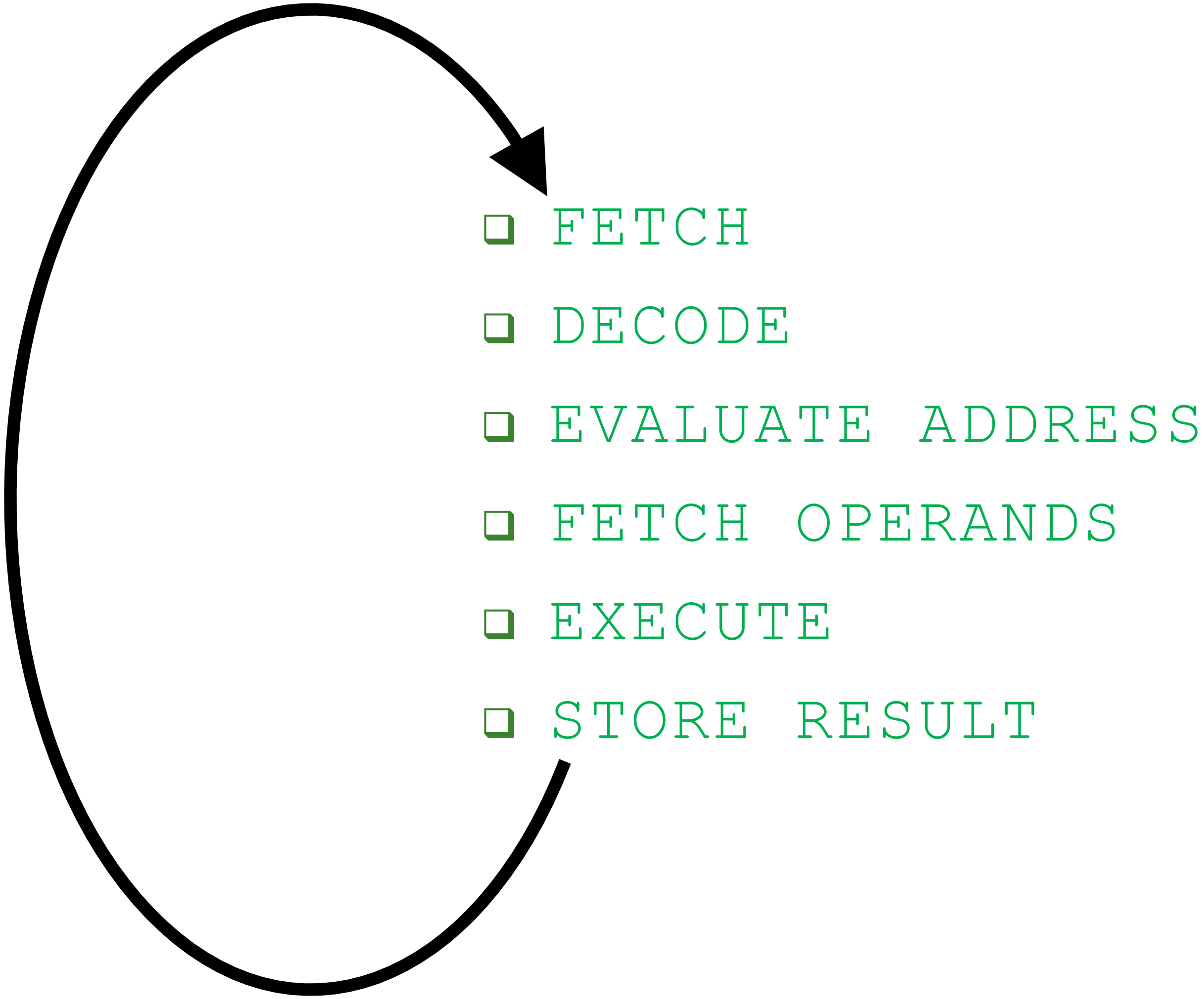
By M. V. Wilkes, M.A., Ph.D., F.R.A.S.

- An elegant implementation:
  - The concept of microcoded/microprogrammed machines

Multi-Cycle Microarchitectures

- 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

Recall: The Instruction Processing "Cycle"



- FETCH
- DECODE
- EVALUATE ADDRESS
- FETCH OPERANDS
- EXECUTE
- STORE RESULT

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

Recall: Control of the Instruction Cycle

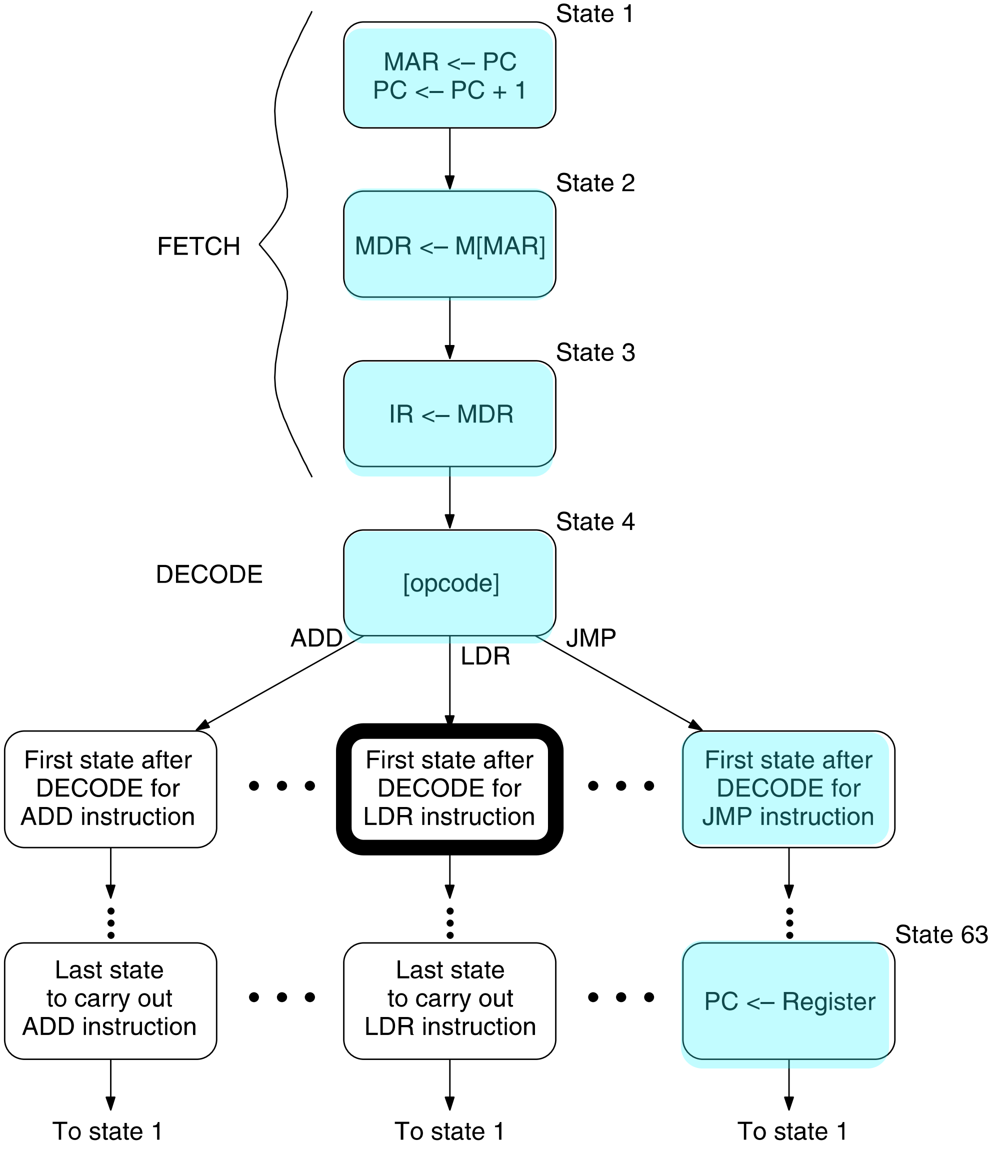


Figure 4.4 An abbreviated state diagram of the LC-3

- State 1
  - The FSM asserts GatePC and LD.MAR
  - It selects input (+1) in PCMUX and asserts LD.PC
- State 2
  - MDR is loaded with the instruction
- State 3
  - The FSM asserts GateMDR and LD.IR
- State 4
  - The FSM goes to next state depending on opcode
- State 63
  - JMP loads register into PC
- Full state diagram in Patt&Pattel, Appendix C

This is an FSM Controlling a Multi-Cycle LC-3 Microarchitecture

Recall: Full State Machine for LC-3b

Decode State

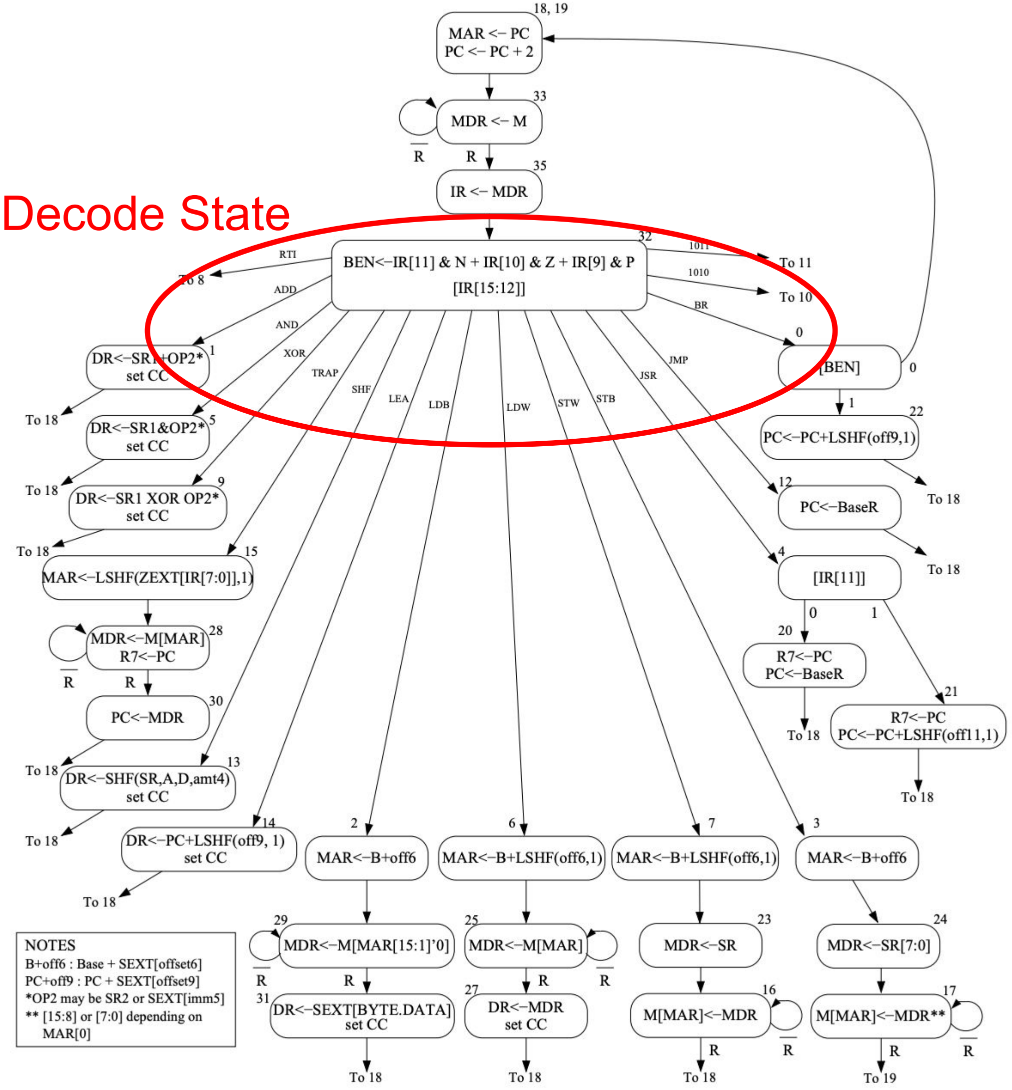
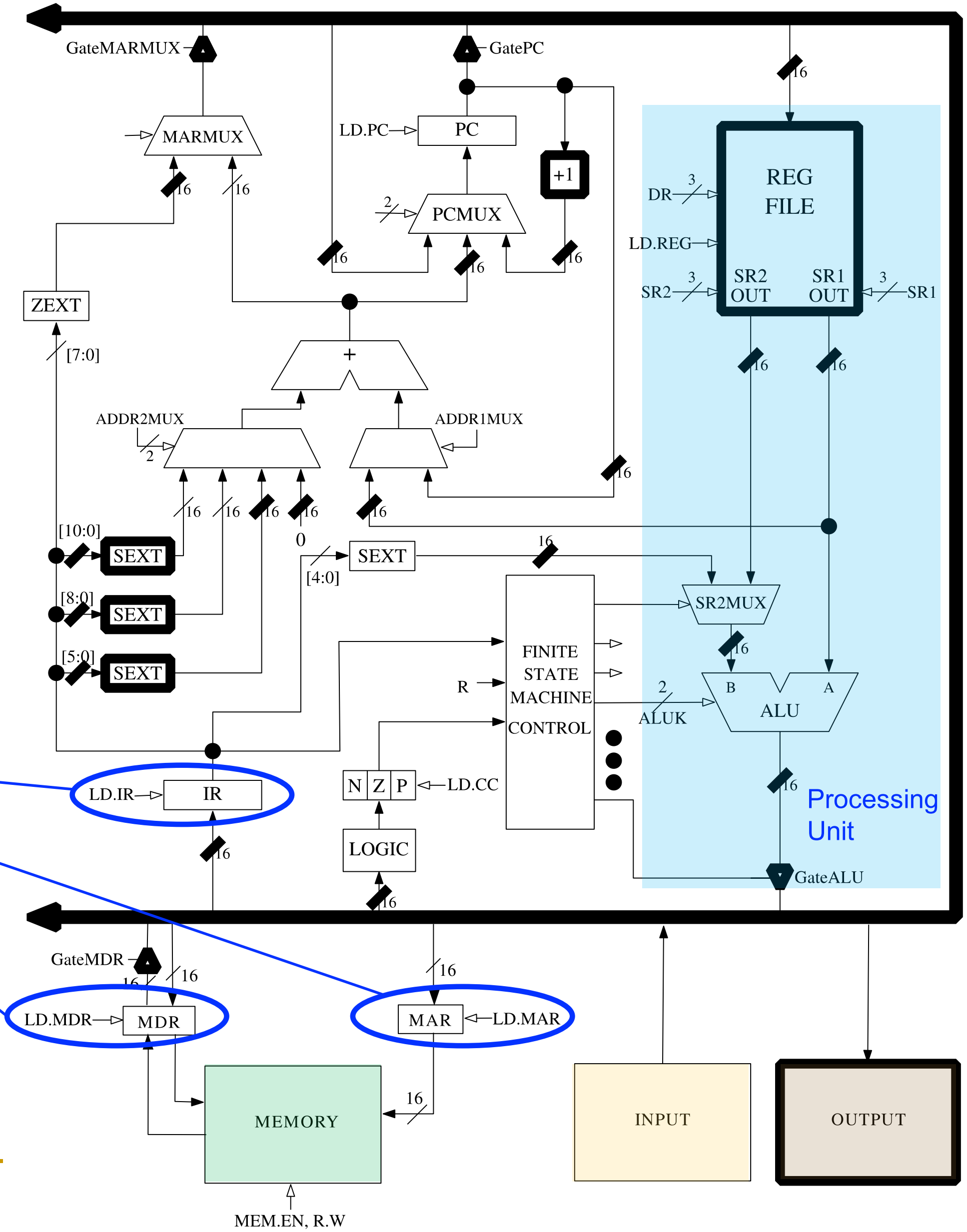


Figure C.2: A state machine for the LC-3b

Full FSM Controlling a Multi-Cycle LC-3b Microarchitecture

https://safari.ethz.ch/digitaltechnik/spring2022/lib/exe/fetch.php?media=pp-appendixc.pdf

Recall: Multi-Cycle LC-3 Data Path

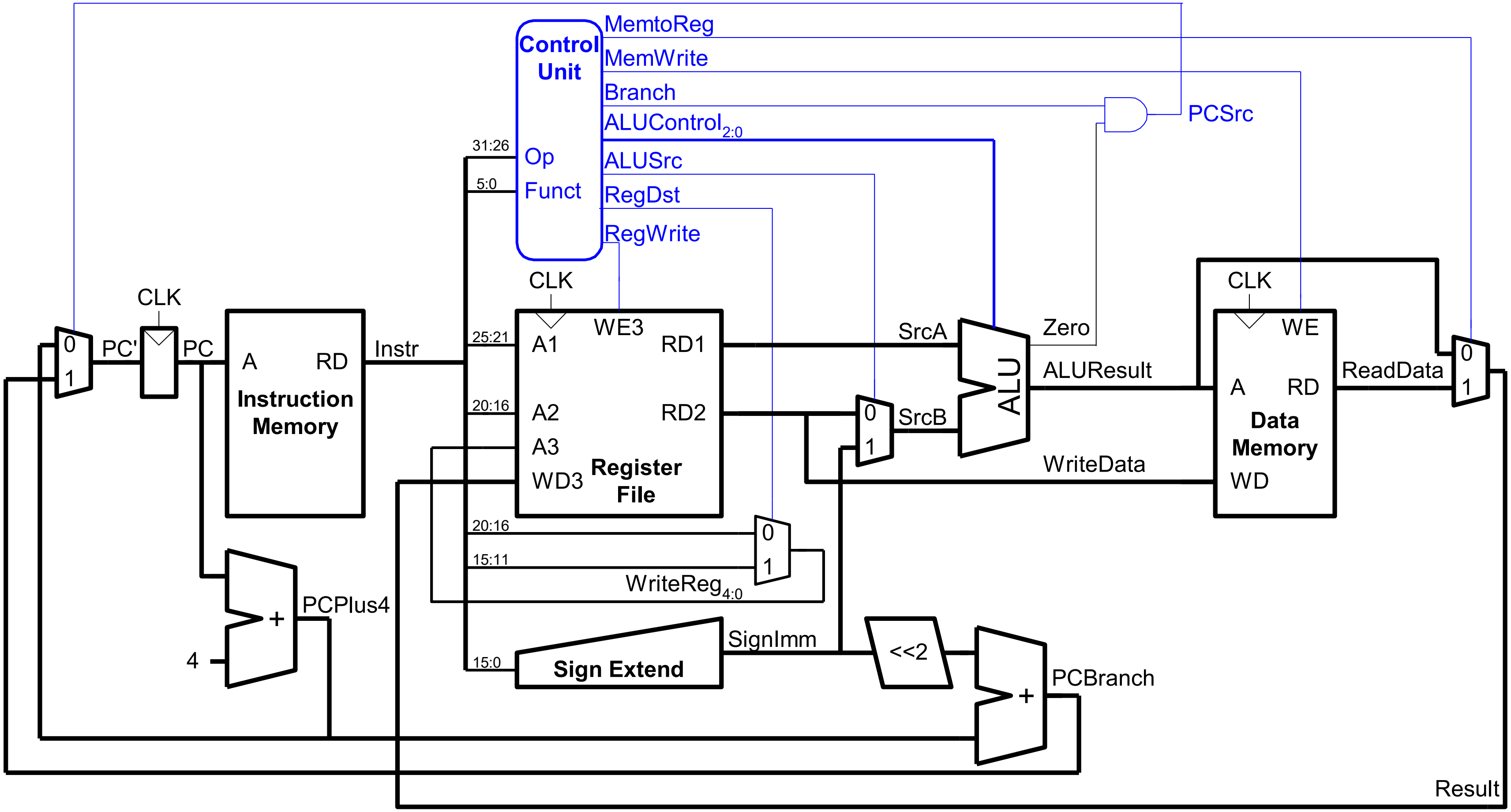


Extra registers not needed in a single-cycle design

Processing Unit

Another Example Multi-Cycle Microarchitecture

Remember the Single-Cycle Uarch II (Readings)



Single-cycle processor. Harris and Harris, Chapter 7.3.

Why Do We Want Multi-Cycle?

- Single-cycle microarchitecture:
  -- cycle time limited by longest instruction (`lw`) → low clock frequency
  -- three adders/ALUs and two memories → high hardware cost

- Multi-cycle microarchitecture:
  + higher clock frequency
  + simpler instructions take only a few clock cycles
  + reuse expensive hardware across multiple cycles
  -- hardware overhead for storing intermediate results
  -- sequential logic overhead paid many times for each instruction

- Multi-cycle requires the same design steps as single cycle:
  - datapath
  - control logic

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)

Let's Construct the Multi-Cycle Datapath

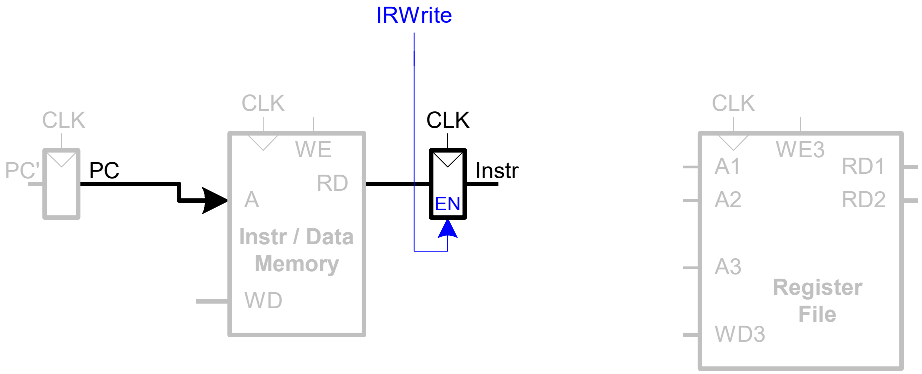
Consider the lw Instruction

- **For an instruction such as: `lw $t0, 0x20($t1)`**

- **We need to:**
  - Read the instruction from memory
  - Then read **`$t1`** from register array
  - Add the immediate value (**`0x20`**) to calculate the memory address
  - Read the content of this address
  - Write to the register **`$t0`** this content

Multi-Cycle Datapath: Instruction Fetch

- **We will consider lw, but fetch is the same for all instructions**
  - STEP 1: Fetch instruction

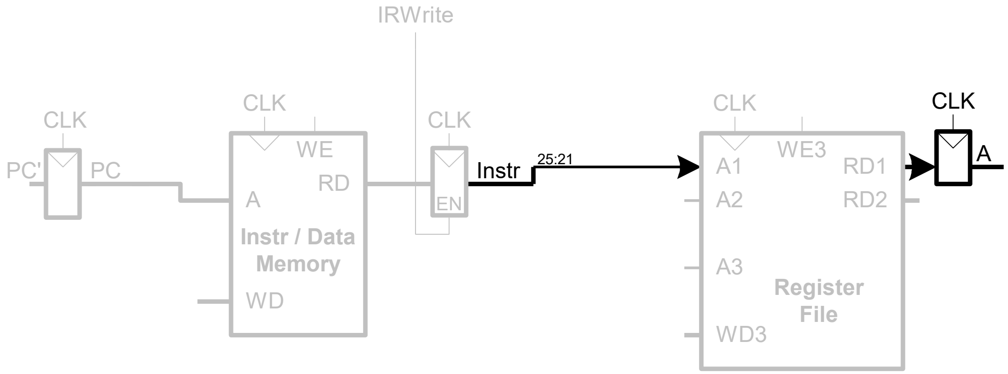


`read from the memory location [rs]+imm to location [rt]`

I-Type

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

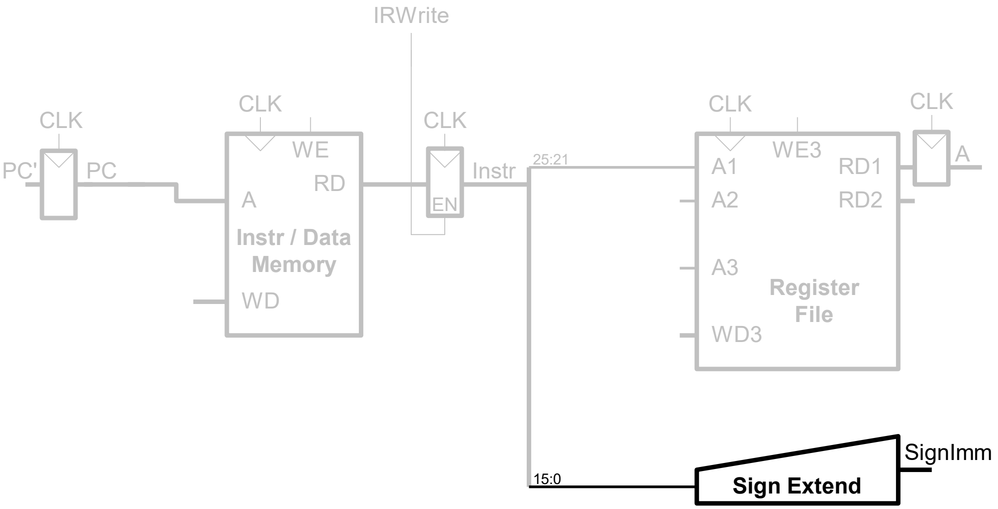
Multi-Cycle Datapath: `lw` register read



I-Type

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

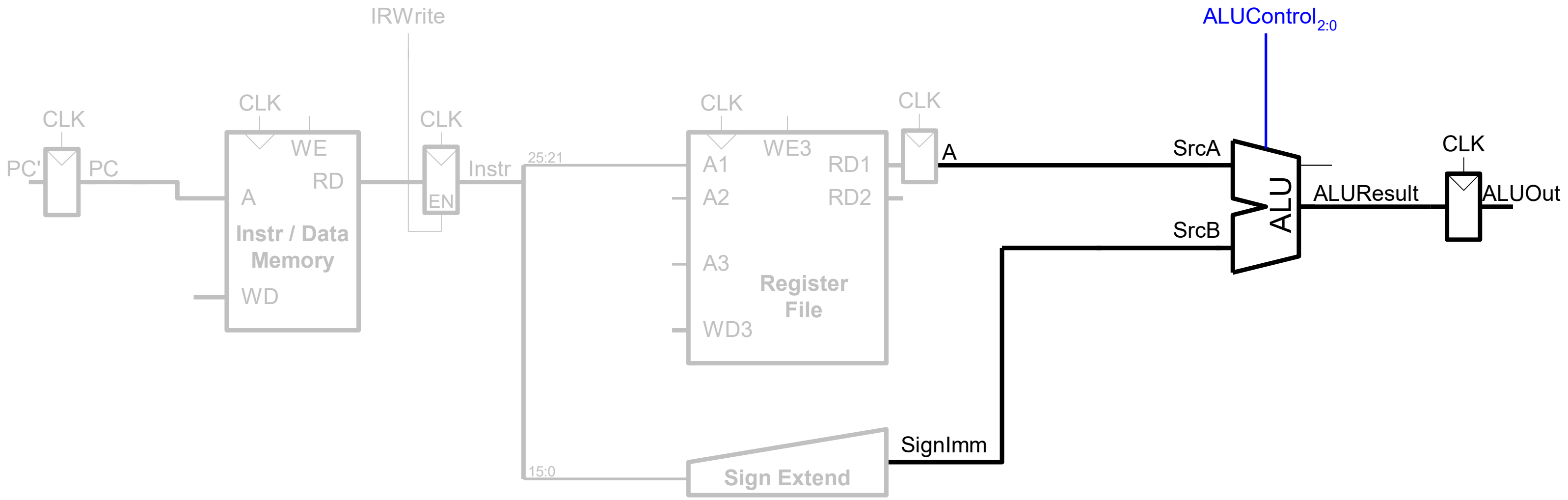
Multi-Cycle Datapath: `lw` immediate



I-Type

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

Multi-Cycle Datapath: `lw` address

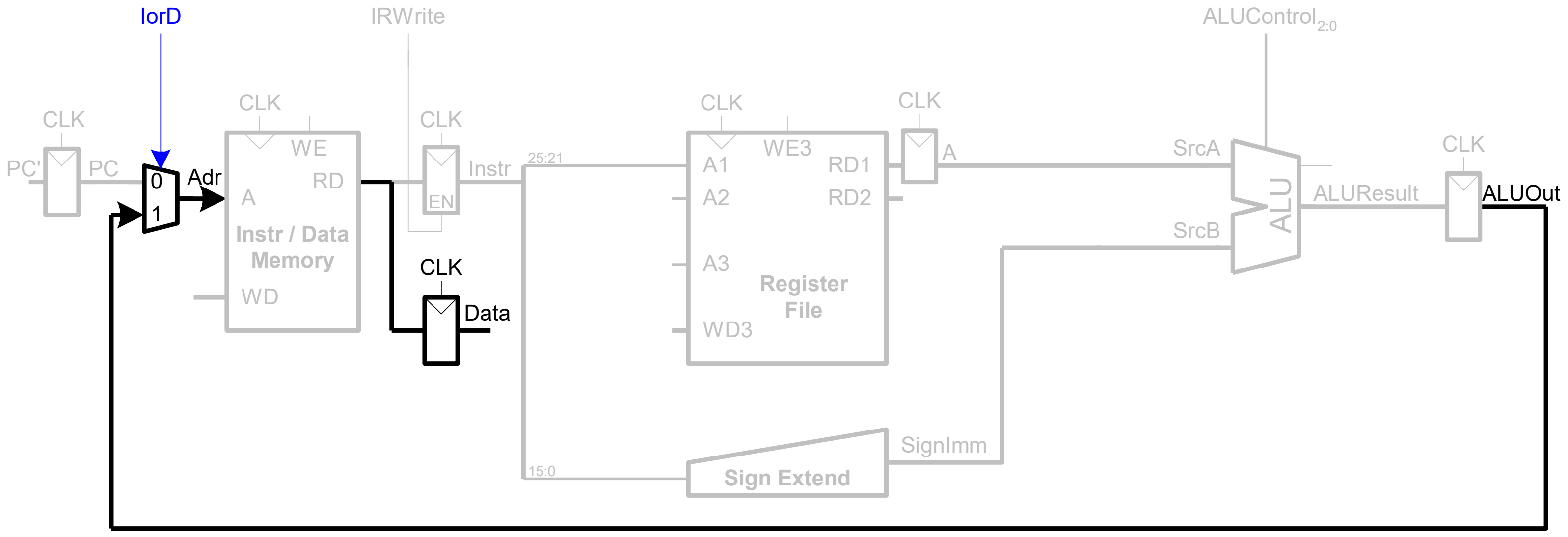


I-Type

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

Multi-Cycle Datapath: `lw` memory read

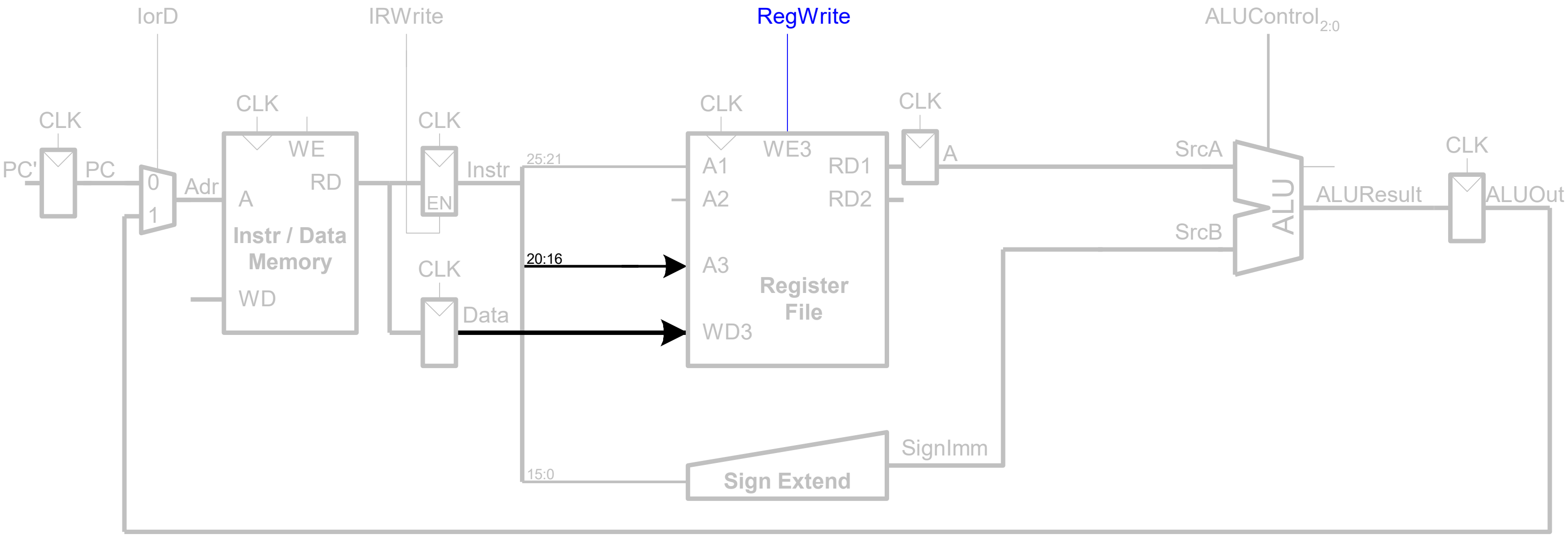
Now we can use **a single memory** to
both fetch an instruction and fetch an operand
(in different clock cycles)



I-Type

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

Multi-Cycle Datapath: `lw` write register

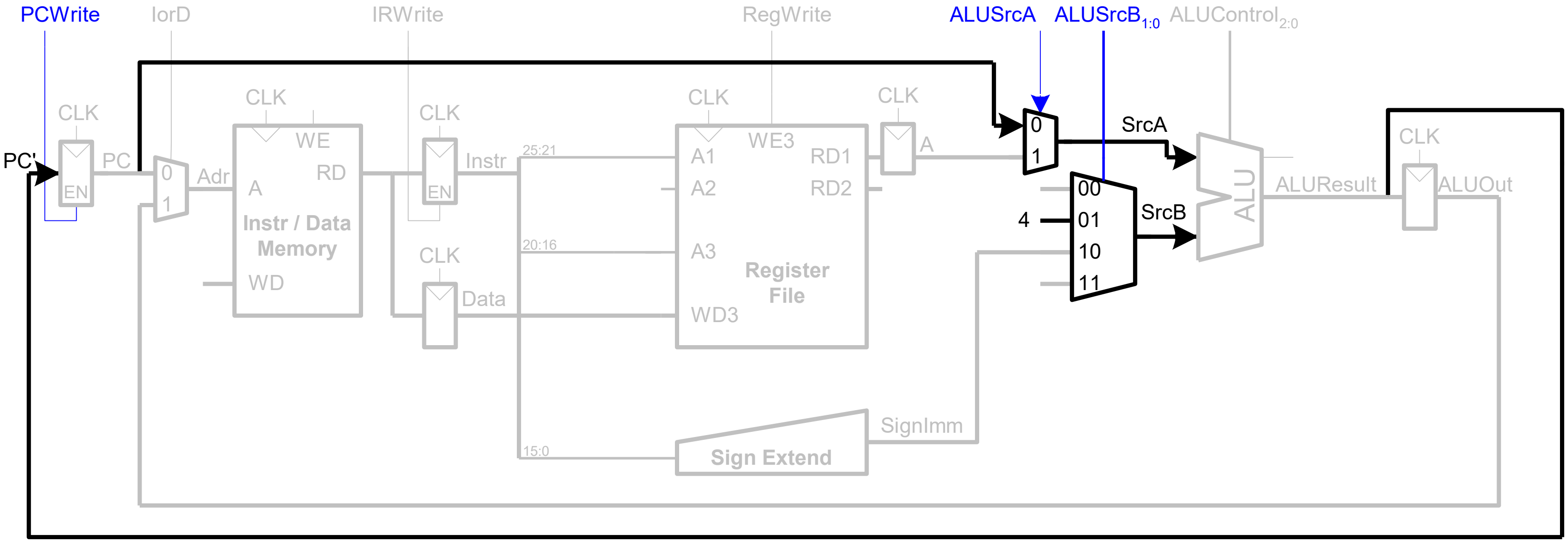


I-Type

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

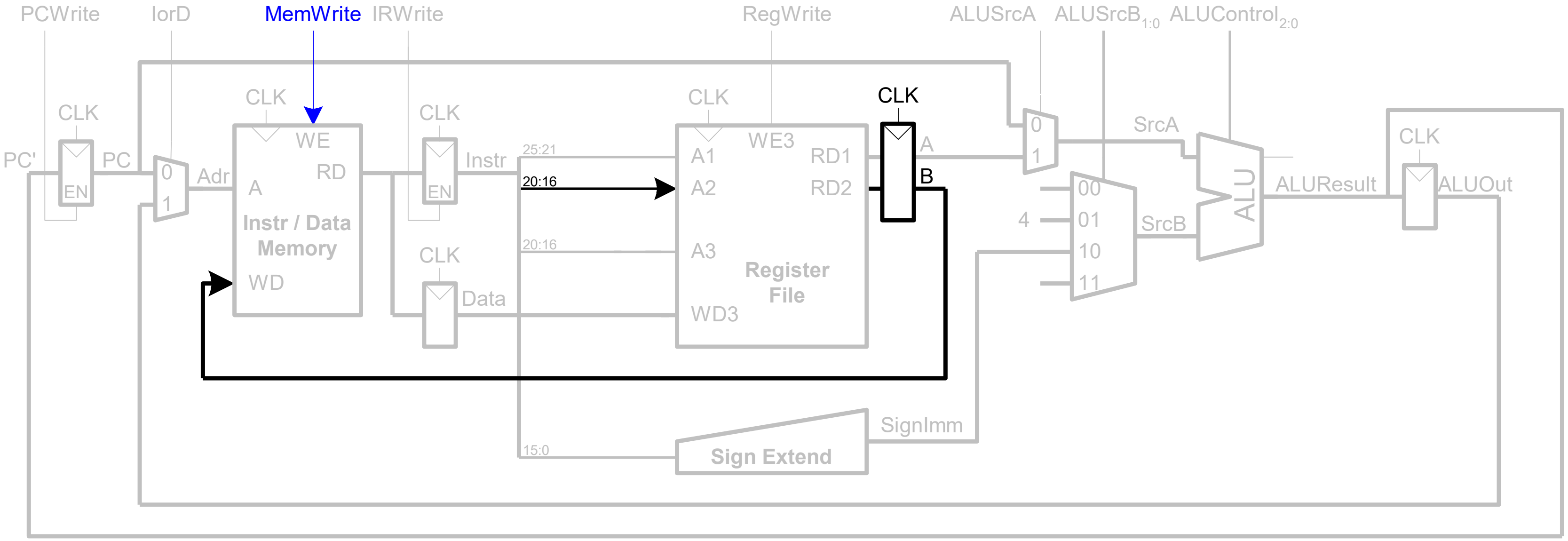
Multi-Cycle Datapath: increment PC

Now we can use **a single ALU** to both increment PC and do address calculation or arithmetic operations (in different clock cycles)



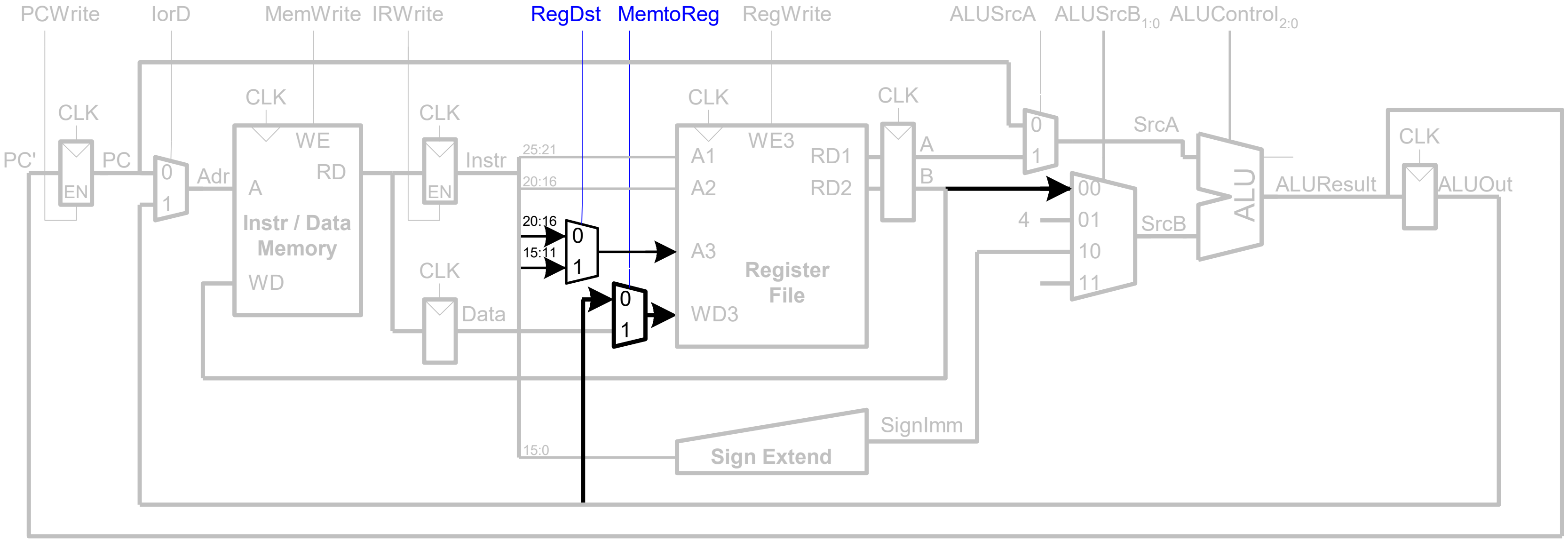
Multi-Cycle Datapath: sw

- **Write data in rt to memory**



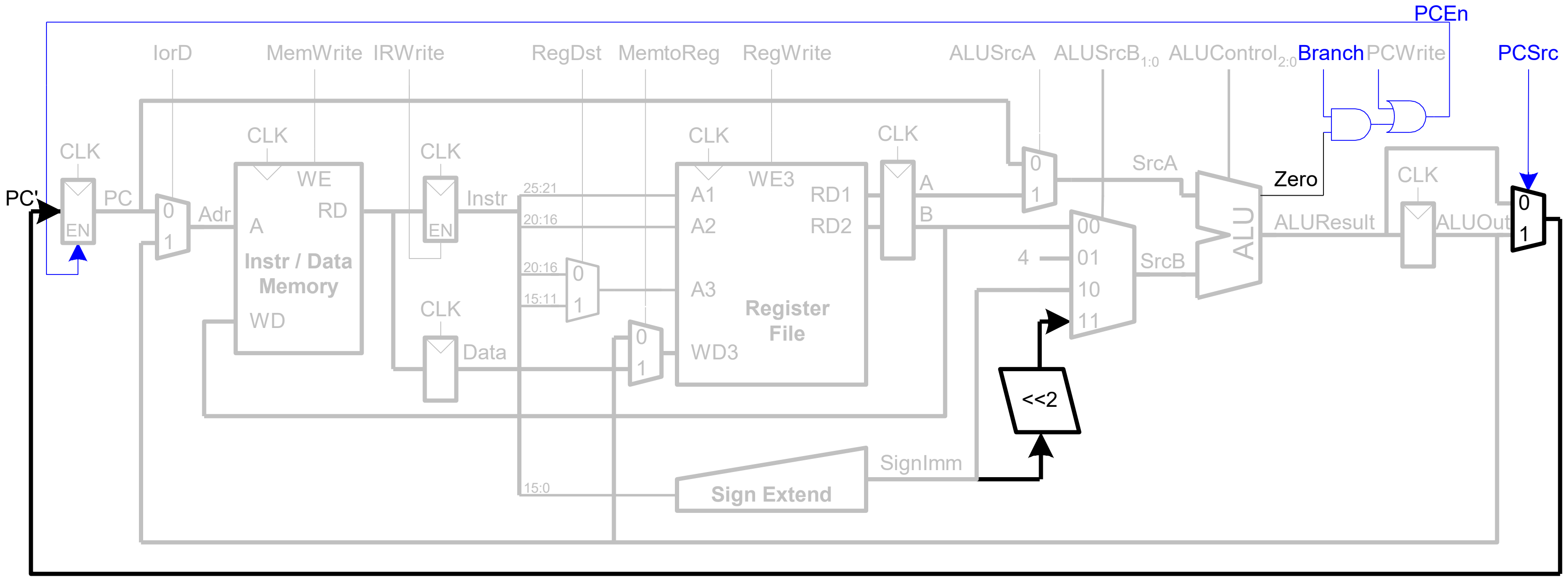
Multi-Cycle Datapath: R-type Instructions

- **Read from rs and rt**
  - Write ALUOut to register file
  - Write to rd (instead of rt)

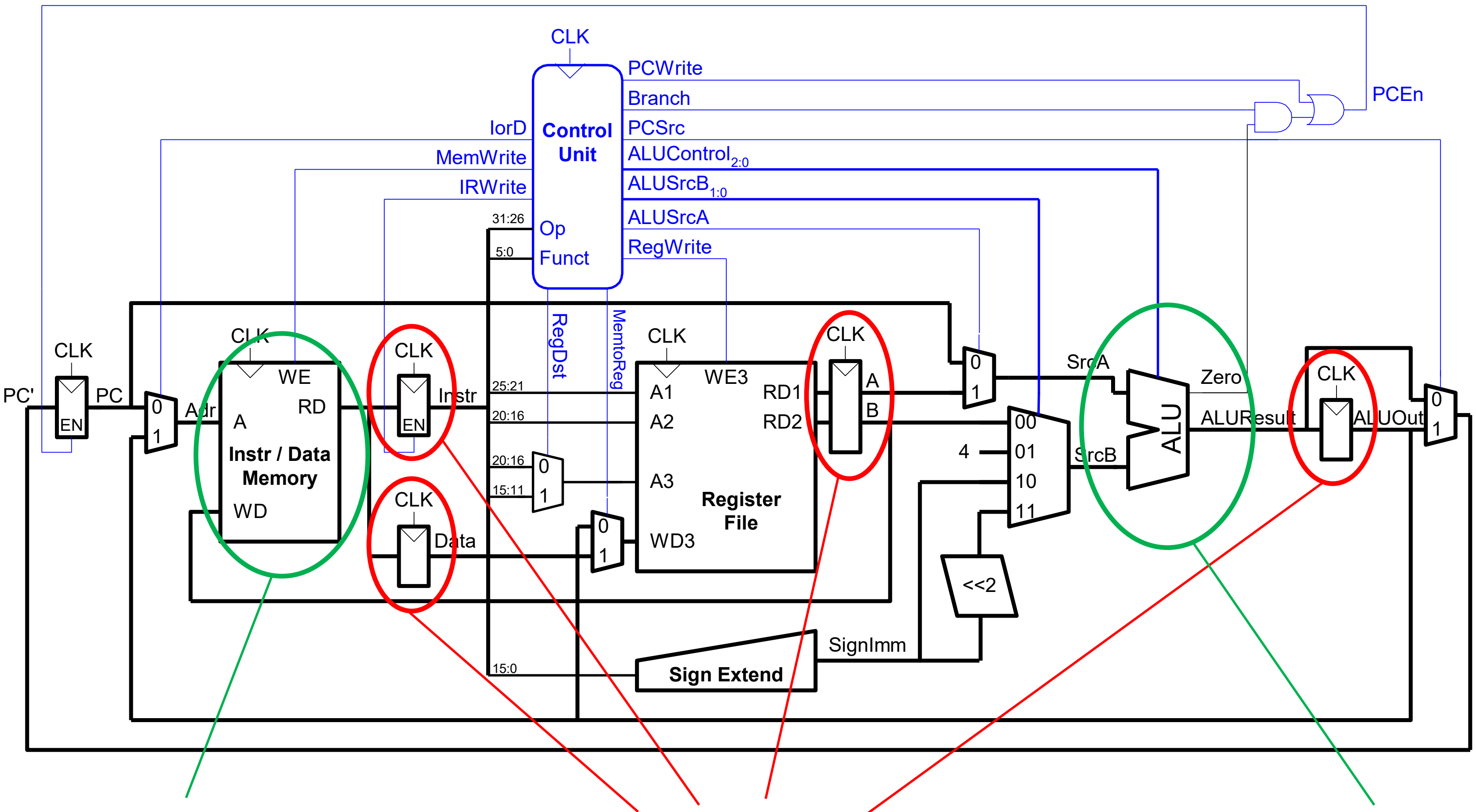


Multi-Cycle Datapath: beq

- **Determine whether values in rs and rt are equal**
  - Calculate branch target address:
    Target Address = (sign-extended immediate << 2) + (PC+4)



Complete Multi-Cycle Processor



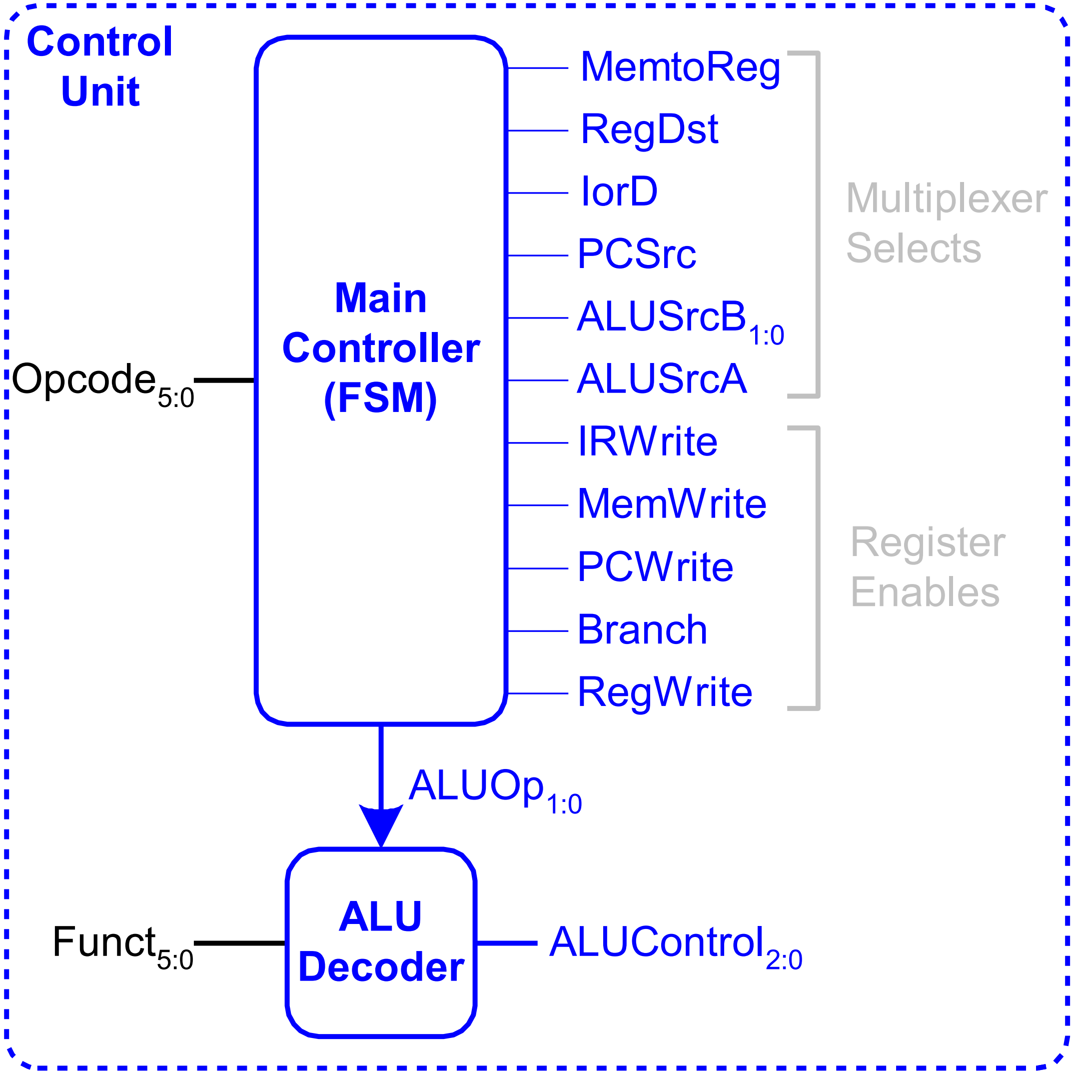
Only one memory

Extra registers not needed in a single-cycle design

Only one ALU/adder

Let's Construct the Multi-Cycle Control Logic

Control Unit



Control Unit

Opcode$_{5:0}$

Main Controller (FSM)

MemtoReg

RegDst

IorD

PCSrc

ALUSrcB$_{1:0}$

ALUSrcA

Multiplexer Selects

IRWrite

MemWrite

PCWrite

Branch

RegWrite

Register Enables

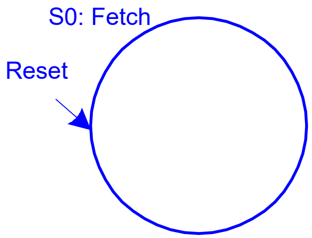
ALUOp$_{1:0}$

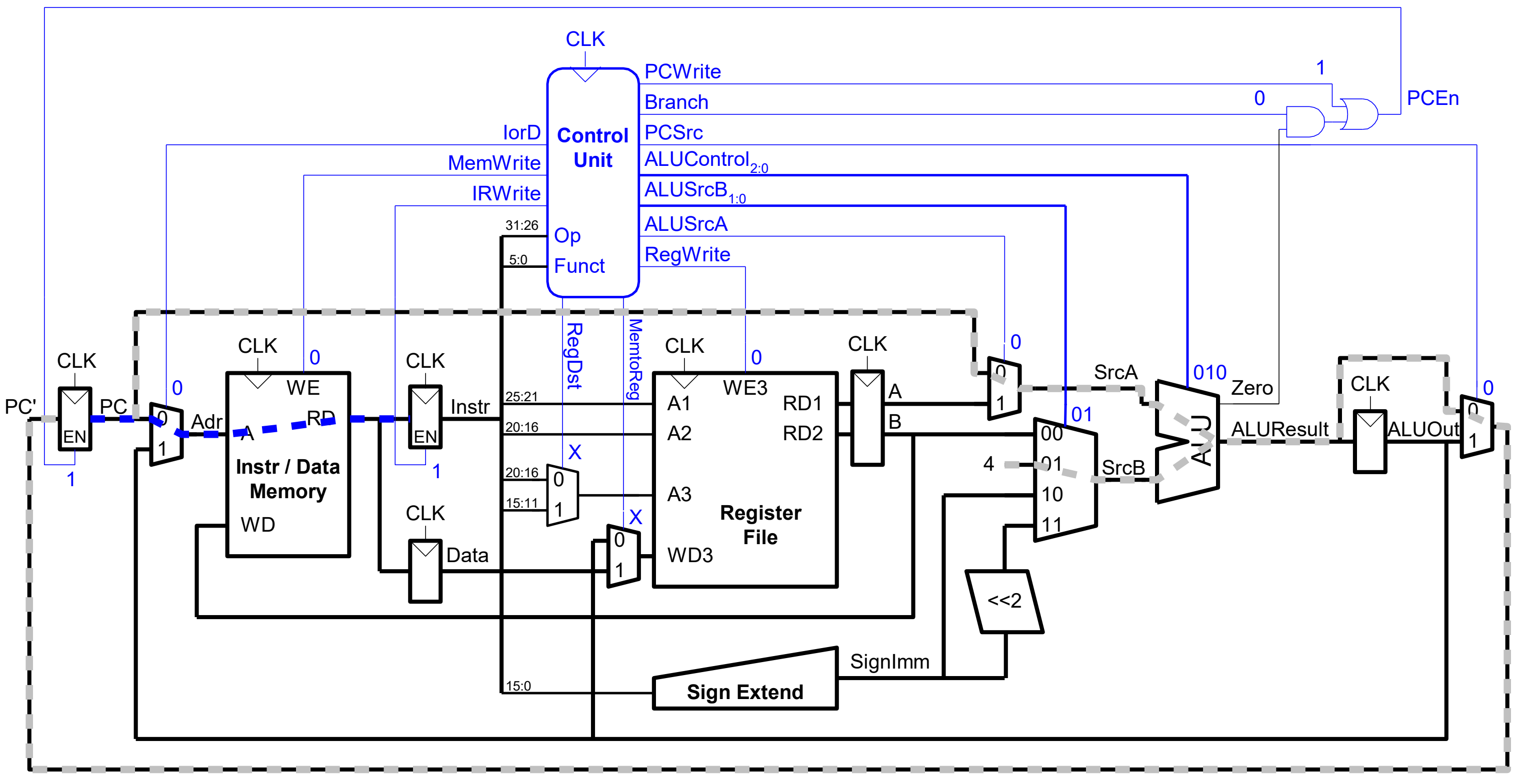
Funct$_{5:0}$

ALU Decoder

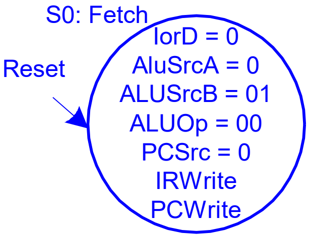
ALUControl$_{2:0}$

Main Controller FSM: Fetch





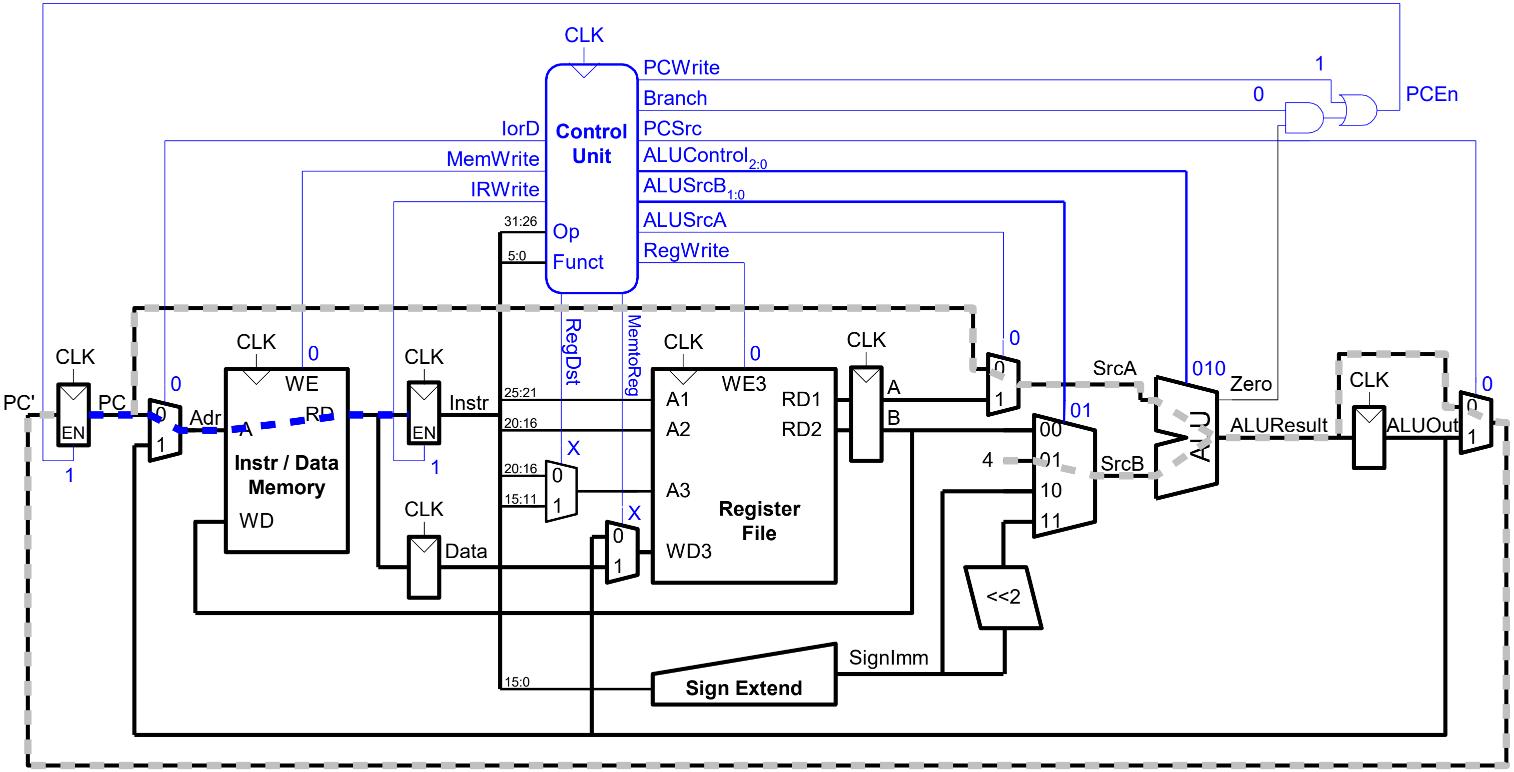
Main Controller FSM: Fetch



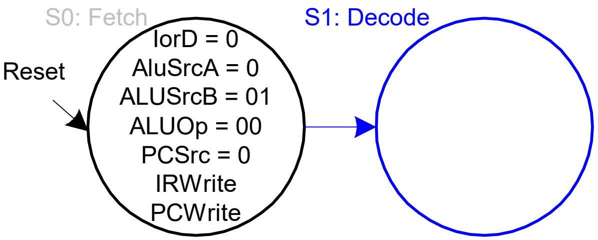
Reset

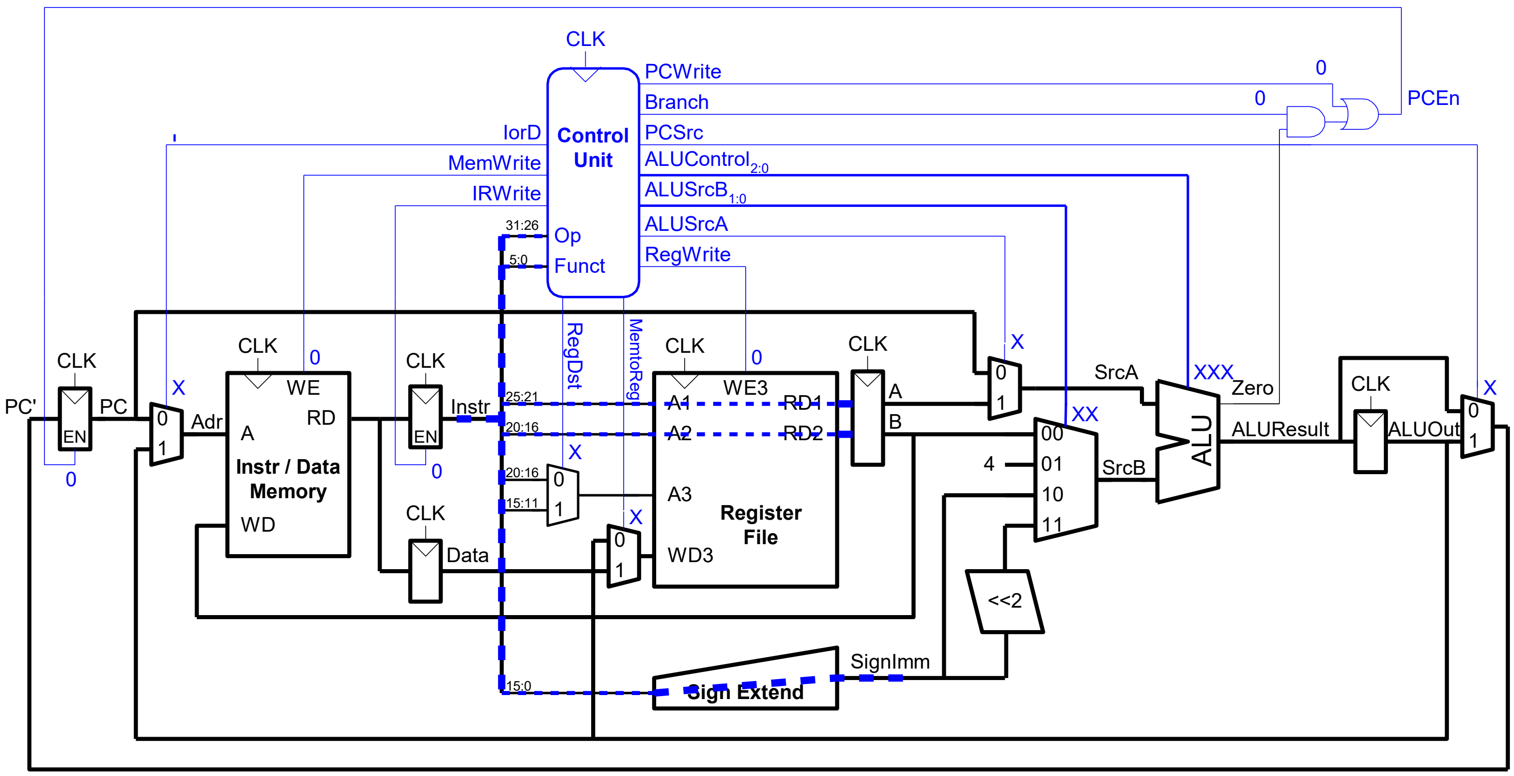
S0: Fetch

IorD = 0
AluSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCSrc = 0
IRWrite
PCWrite

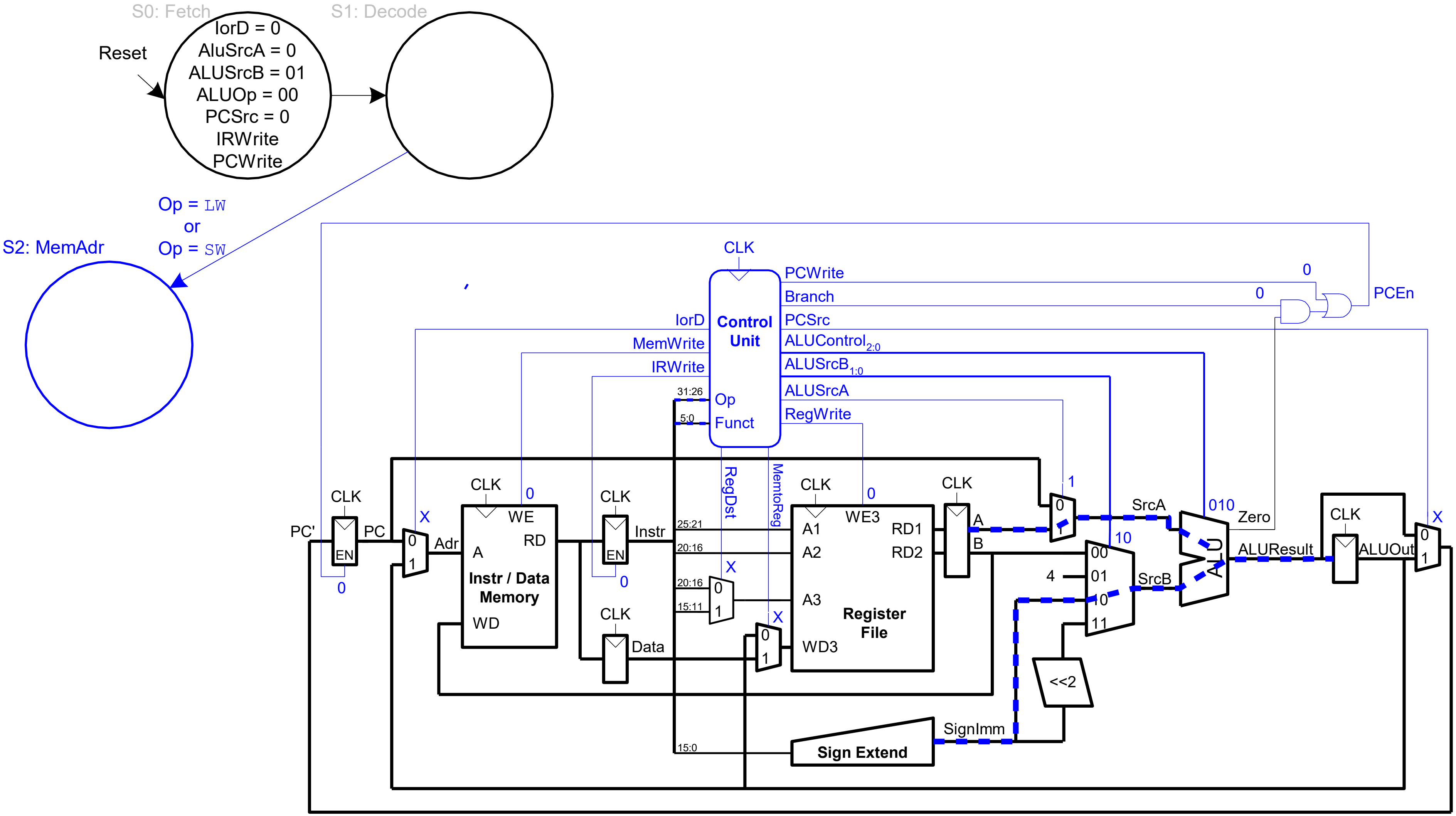


Main Controller FSM: Decode

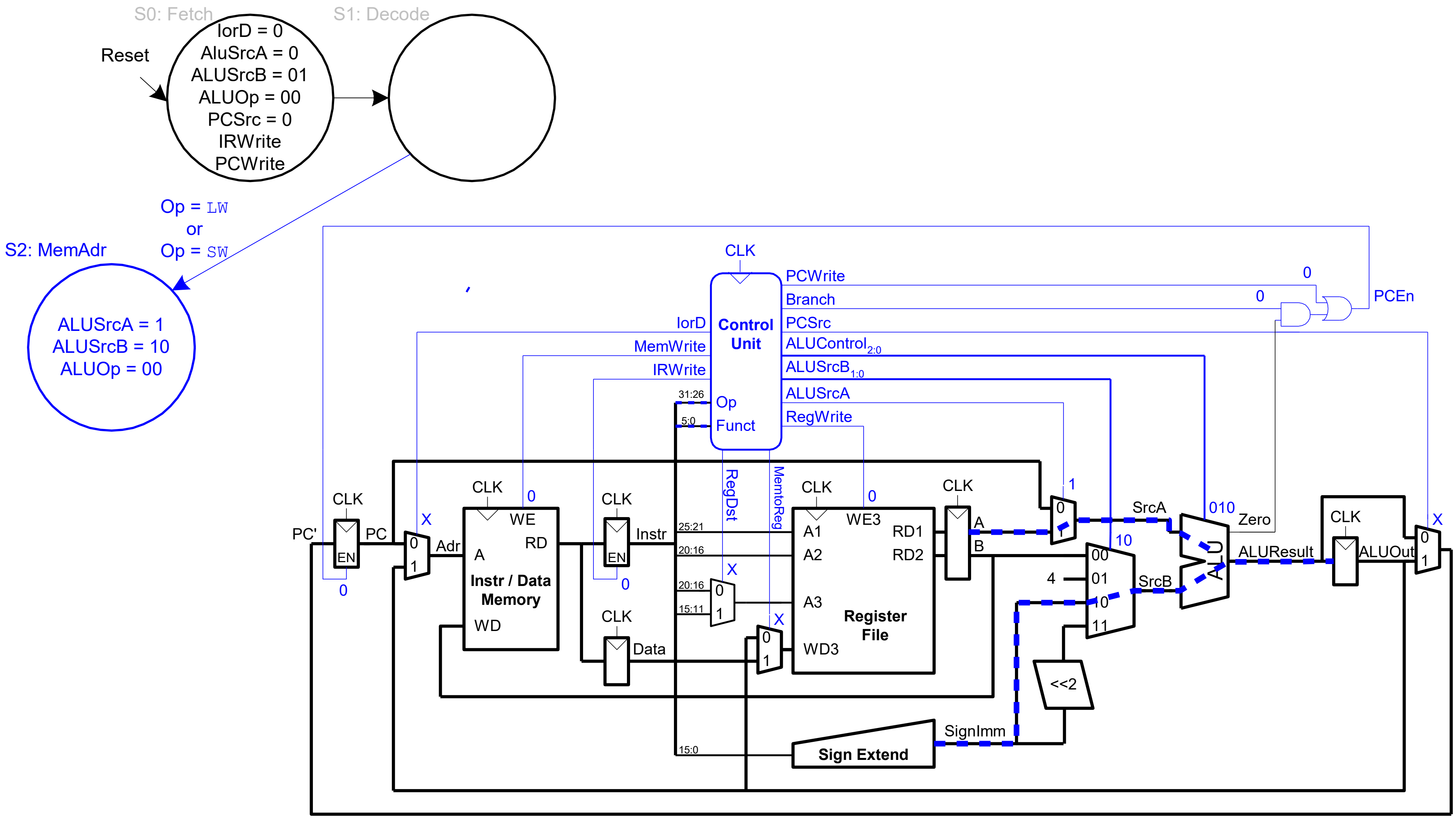




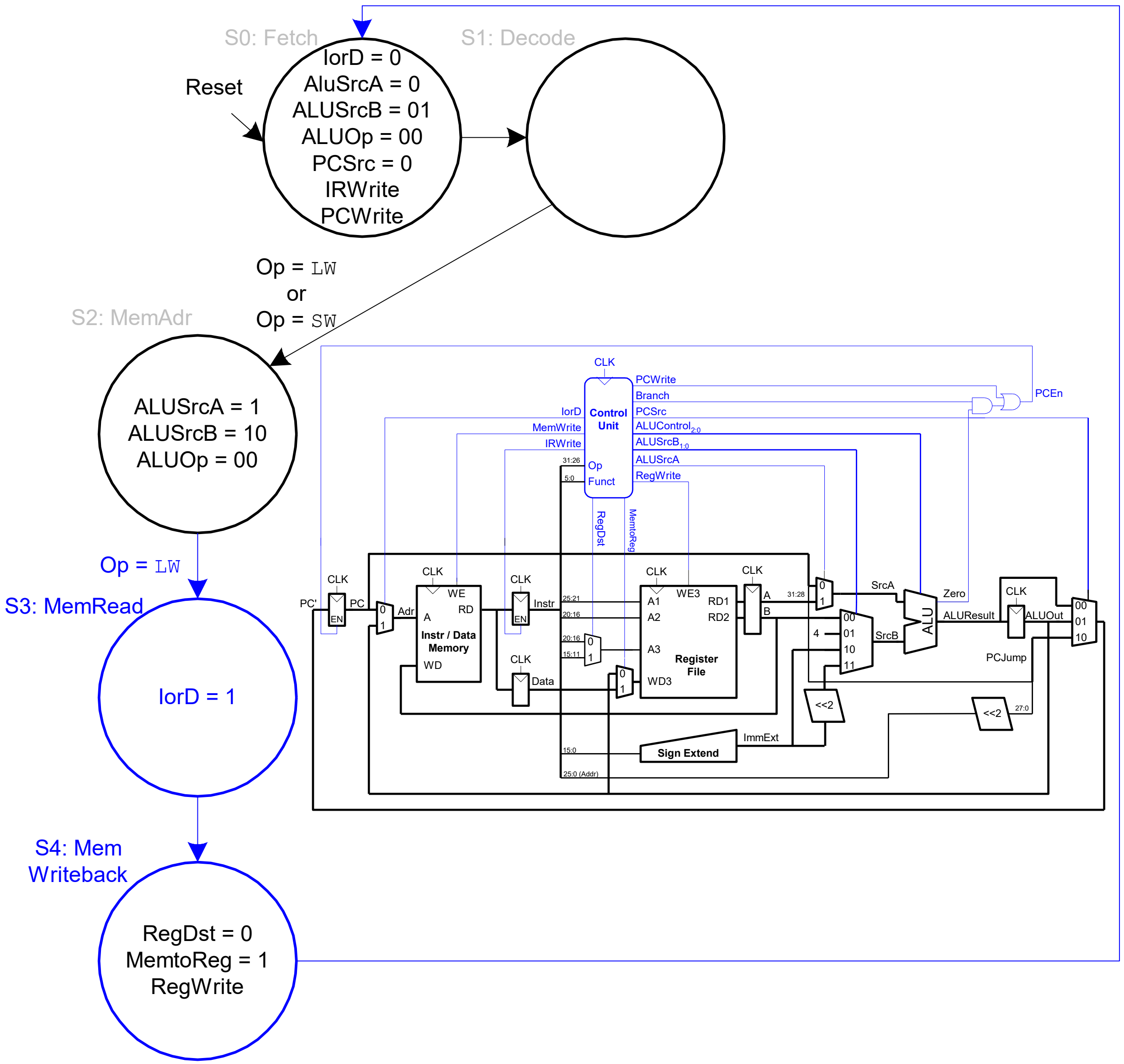
Main Controller FSM: Address Calculation



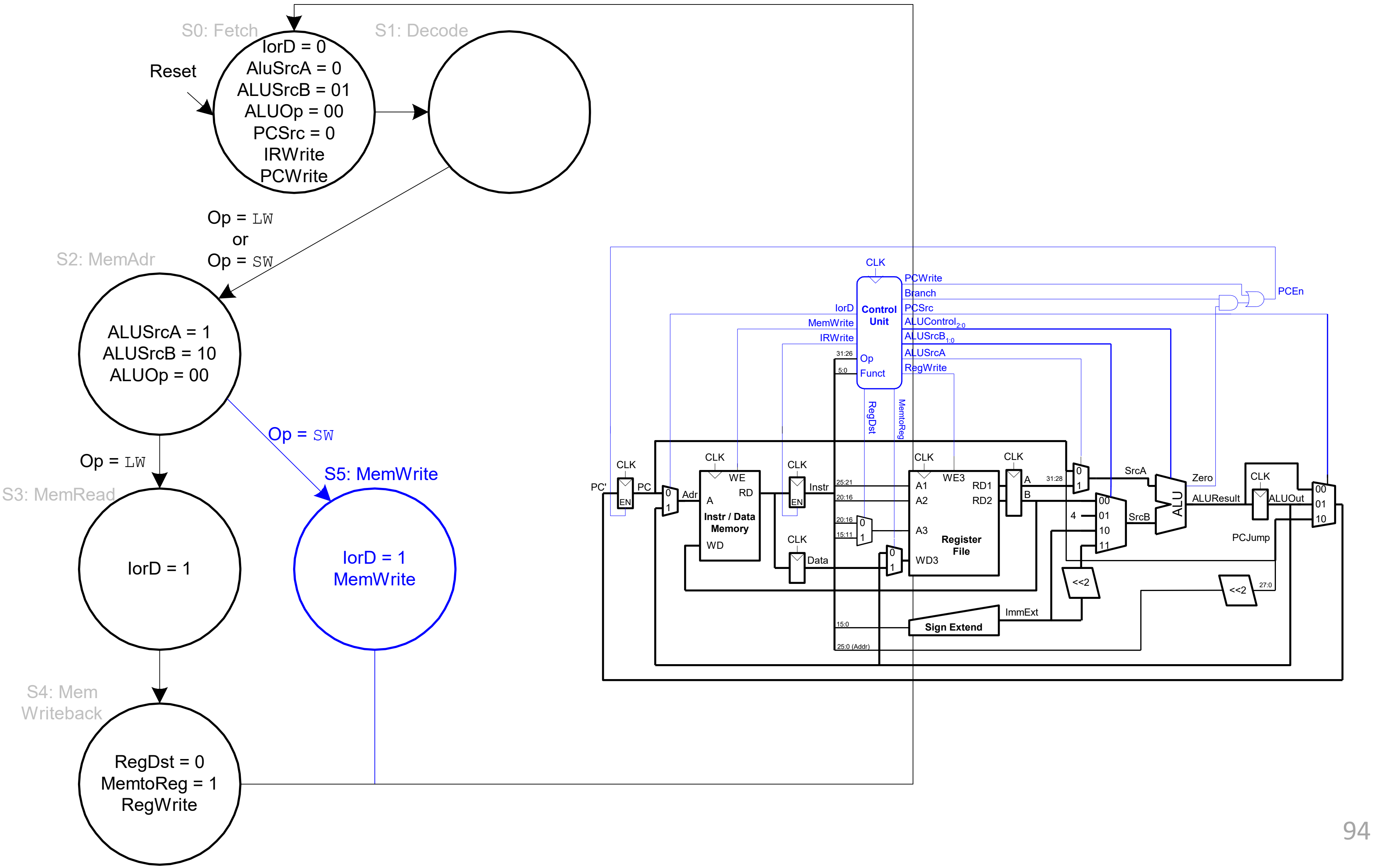
Main Controller FSM: Address Calculation



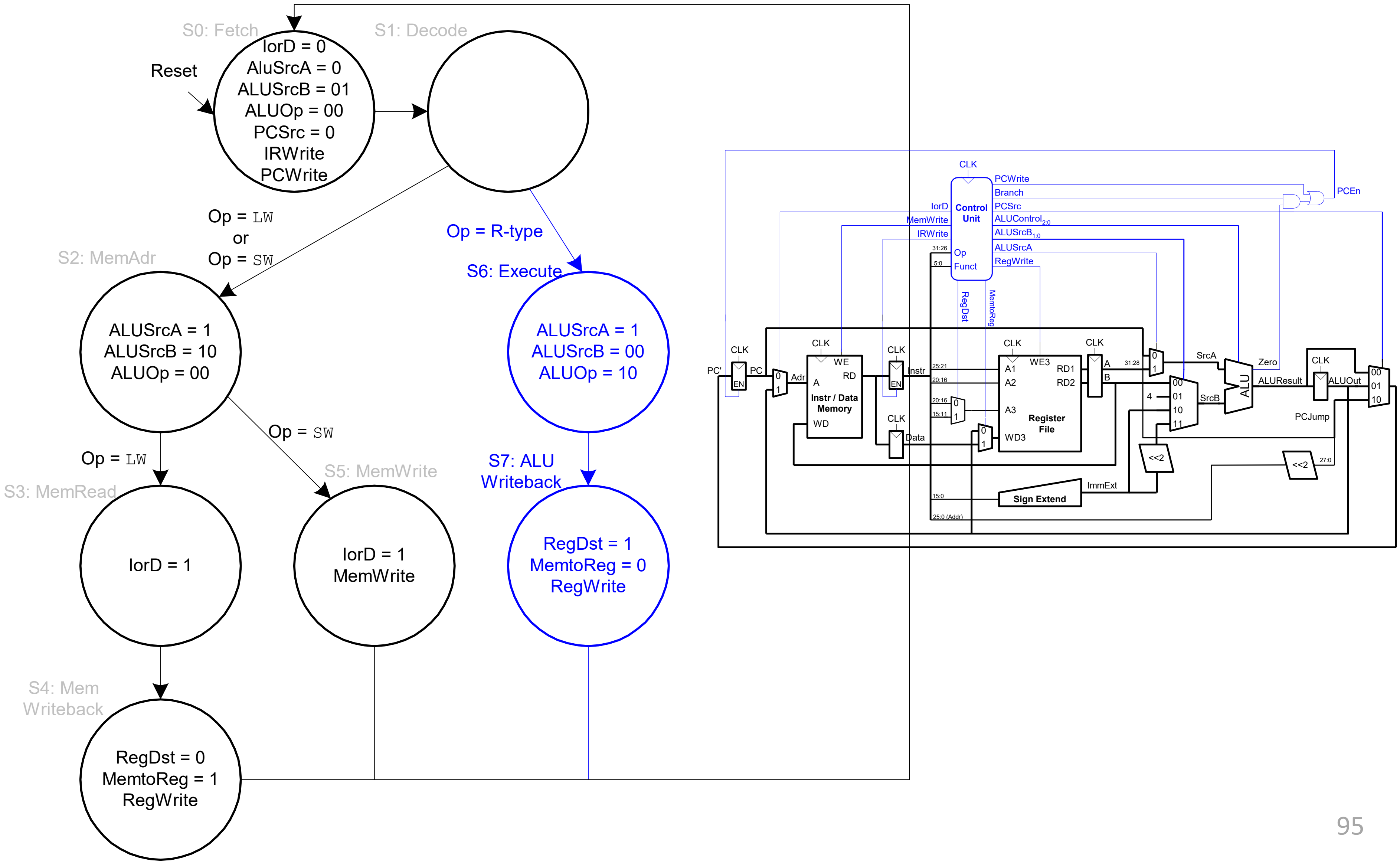
Main Controller FSM: `lw`



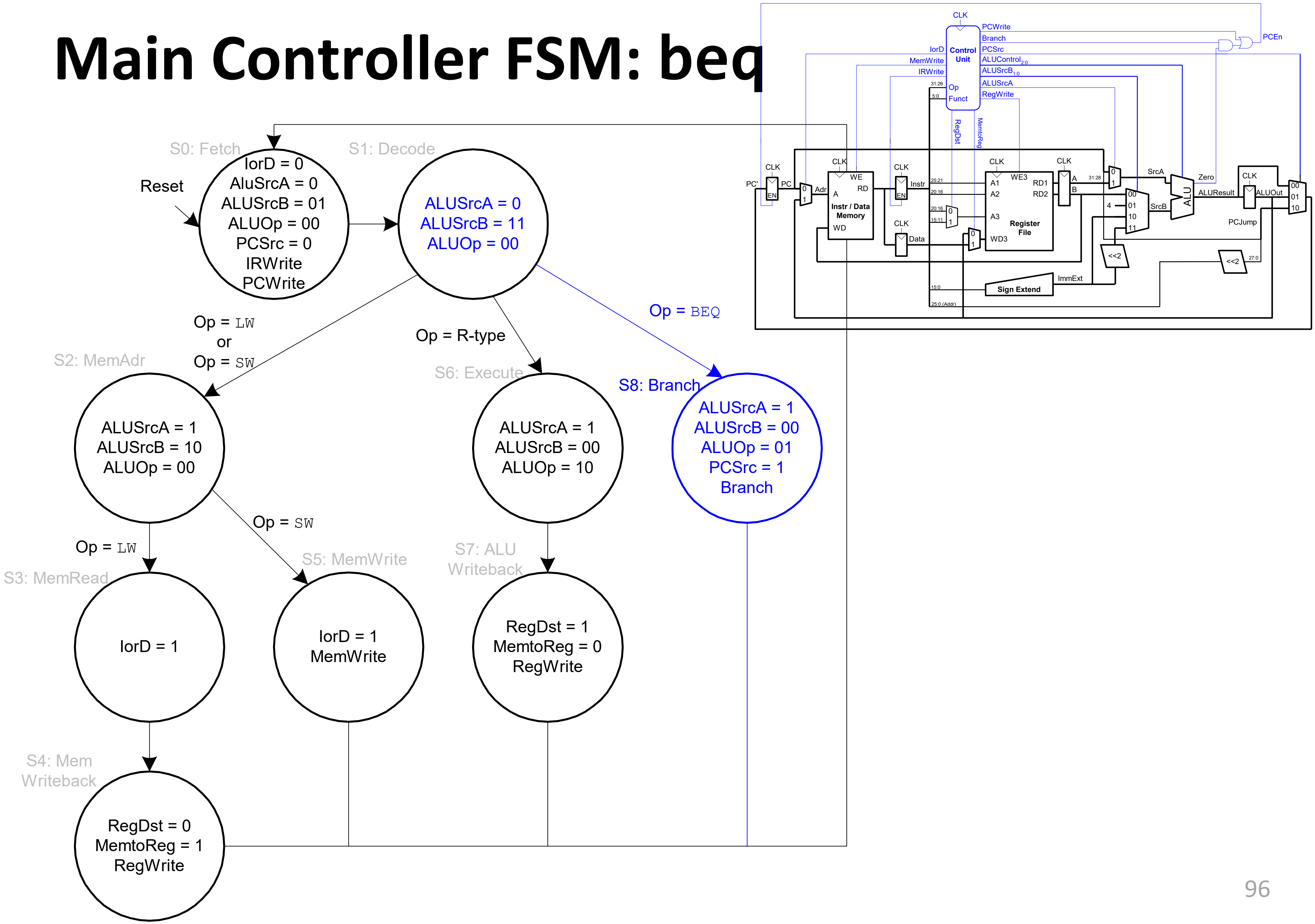
Main Controller FSM: `sw`



Main Controller FSM: R-Type



Main Controller FSM: beq



S0: Fetch

Reset

IorD = 0
AluSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCSrc = 0
IRWrite
PCWrite

S1: Decode

ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00

Op = LW
or
Op = SW

Op = R-type

Op = BEQ

S2: MemAdr

ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00

S6: Execute

ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10

S8: Branch

ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCSrc = 1
Branch

Op = LW

Op = SW

S3: MemRead

IorD = 1

S5: MemWrite

IorD = 1
MemWrite

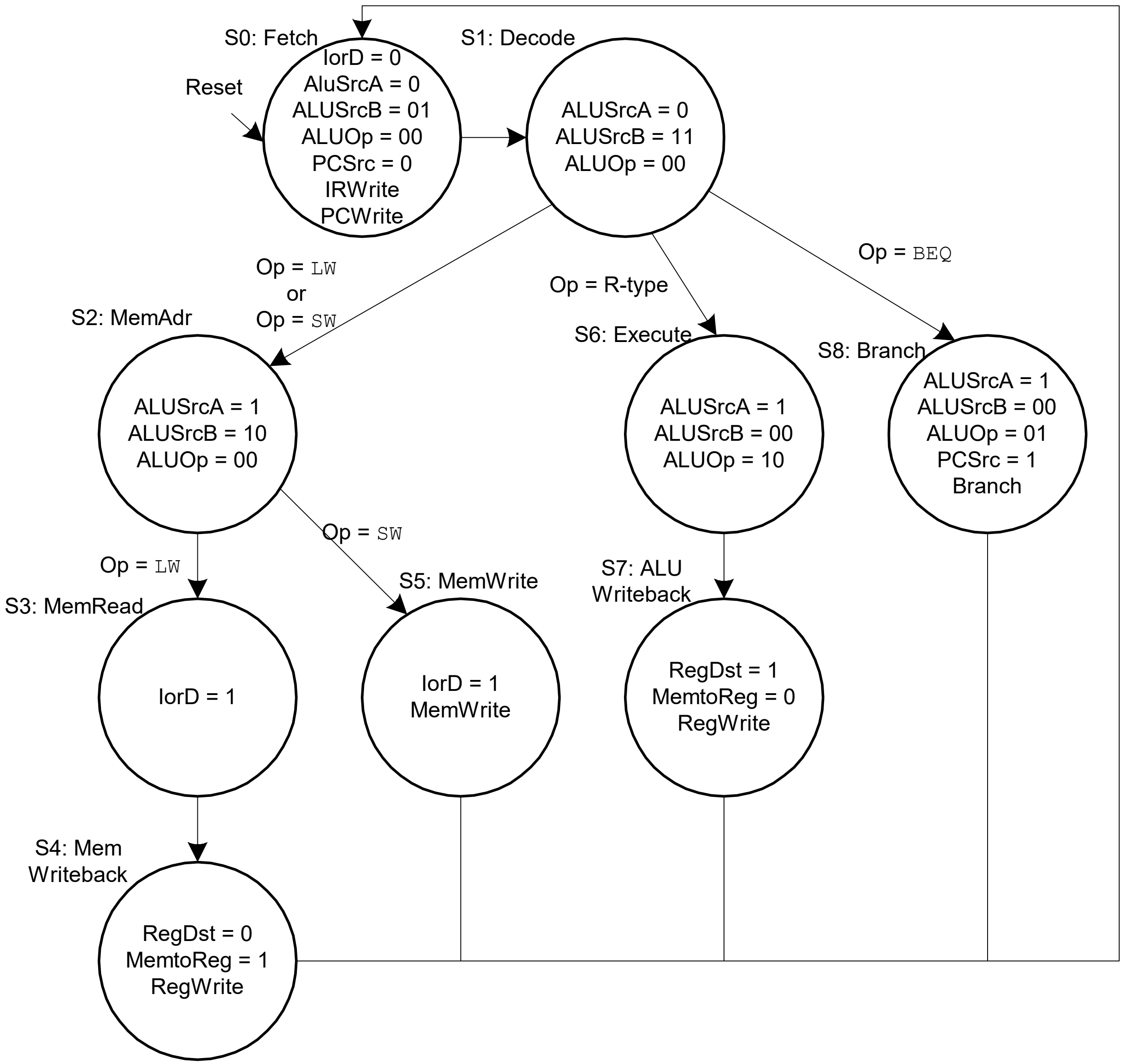
S7: ALU Writeback

RegDst = 1
MemtoReg = 0
RegWrite

S4: Mem Writeback

RegDst = 0
MemtoReg = 1
RegWrite

Complete Multi-Cycle Controller FSM



Reset

S0: Fetch
IorD = 0
AluSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCSrc = 0
IRWrite
PCWrite

S1: Decode
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00

Op = LW
or
Op = SW

Op = R-type

Op = BEQ

S2: MemAdr
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00

S6: Execute
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10

S8: Branch
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCSrc = 1
Branch

Op = LW

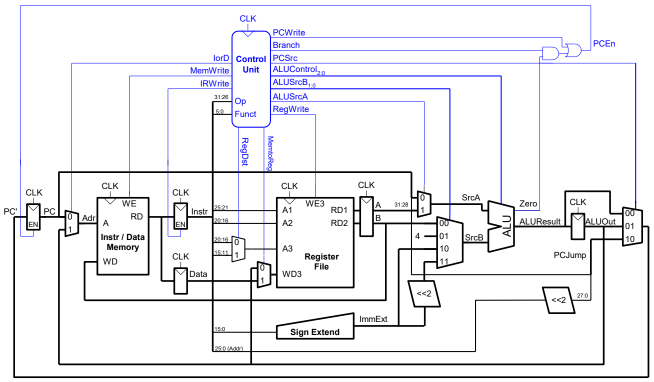
Op = SW

S3: MemRead
IorD = 1

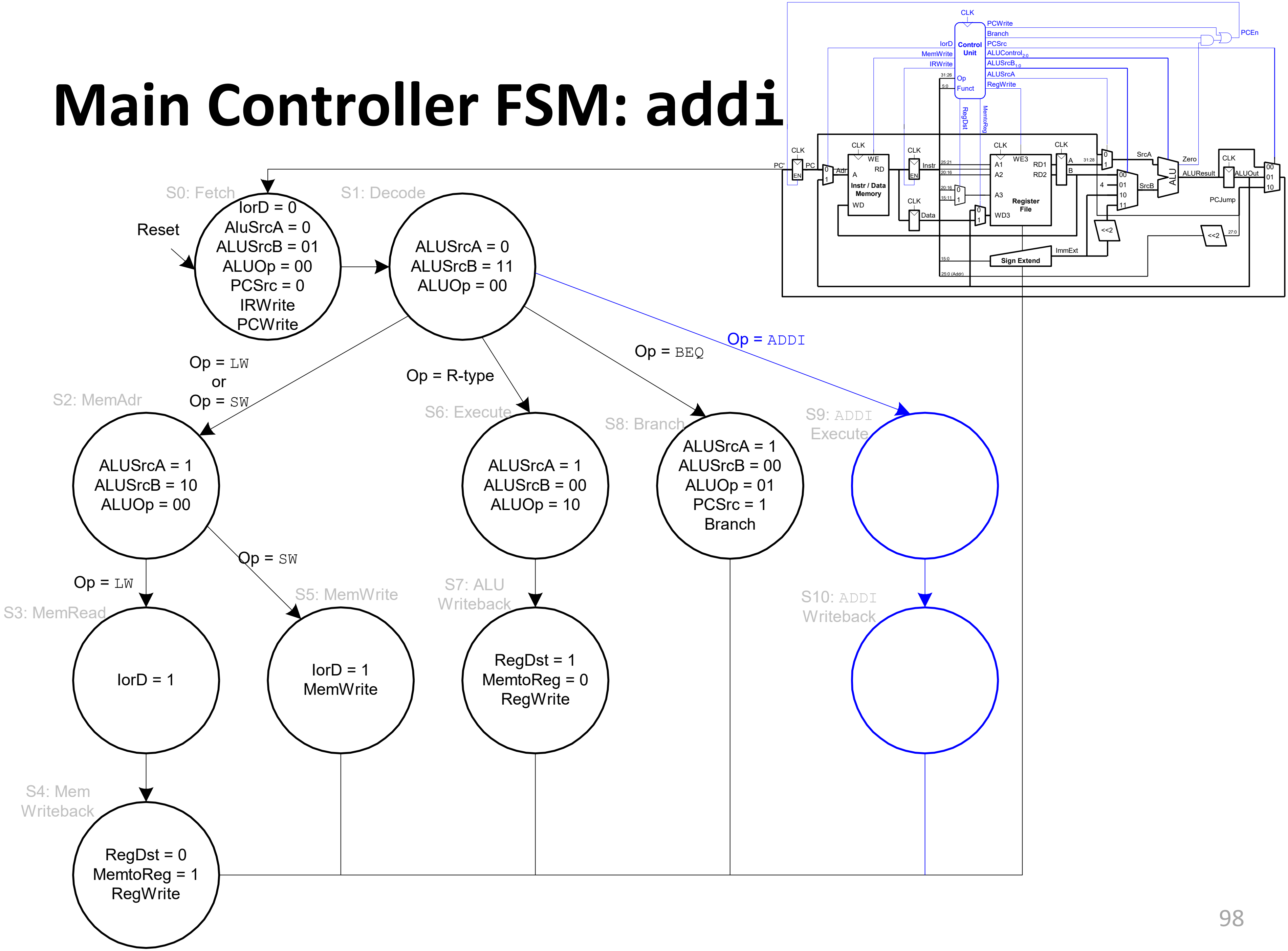
S5: MemWrite
IorD = 1
MemWrite

S7: ALU
Writeback
RegDst = 1
MemtoReg = 0
RegWrite

S4: Mem
Writeback
RegDst = 0
MemtoReg = 1
RegWrite



Main Controller FSM: addi



S0: Fetch

Reset

IorD = 0
AluSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCSrc = 0
IRWrite
PCWrite

S1: Decode

ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00

Op = `LW`
or
Op = `SW`

Op = R-type

Op = `BEQ`

Op = `ADDI`

S2: MemAdr

ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00

S6: Execute

ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10

S8: Branch

ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCSrc = 1
Branch

S9: `ADDI`
Execute

Op = `SW`

Op = `LW`

S3: MemRead

IorD = 1

S5: MemWrite

IorD = 1
MemWrite

S7: ALU
Writeback

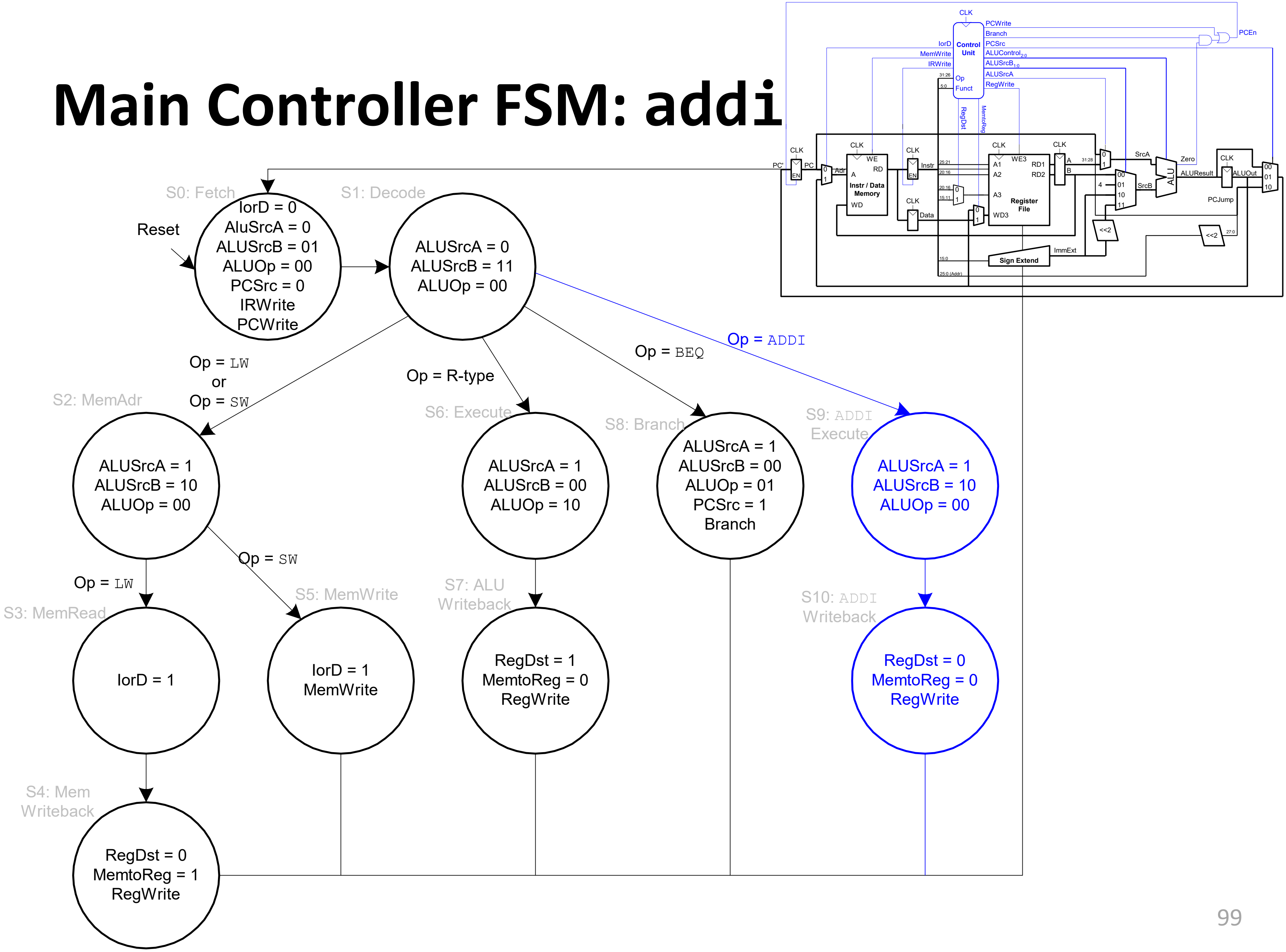
RegDst = 1
MemtoReg = 0
RegWrite

S10: `ADDI`
Writeback

S4: Mem
Writeback

RegDst = 0
MemtoReg = 1
RegWrite

Main Controller FSM: addi



S0: Fetch

Reset

IorD = 0
AluSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCSrc = 0
IRWrite
PCWrite

S1: Decode

ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00

Op = LW
or
Op = SW

Op = R-type

Op = BEQ

Op = ADDI

S2: MemAdr

ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00

S6: Execute

ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10

S8: Branch

ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCSrc = 1
Branch

S9: ADDI
Execute

ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00

Op = LW

Op = SW

S3: MemRead

IorD = 1

S5: MemWrite

IorD = 1
MemWrite

S7: ALU
Writeback

RegDst = 1
MemtoReg = 0
RegWrite

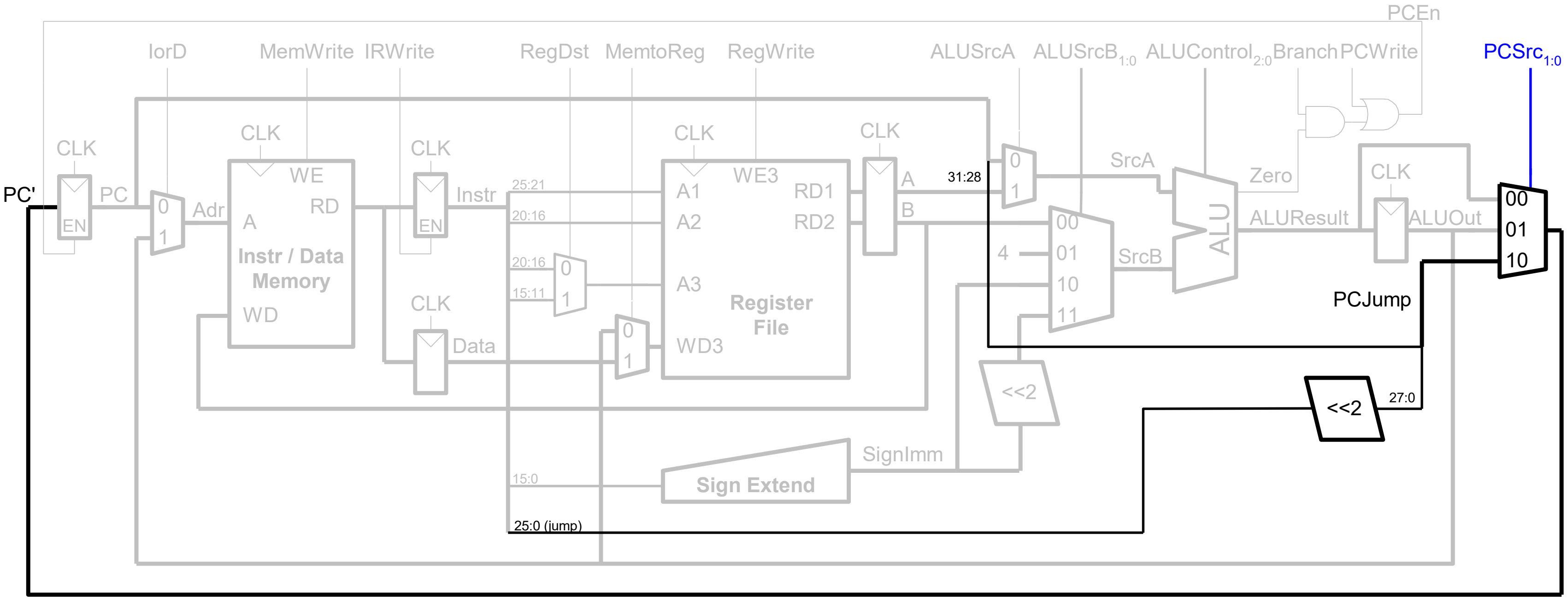
S10: ADDI
Writeback

RegDst = 0
MemtoReg = 0
RegWrite

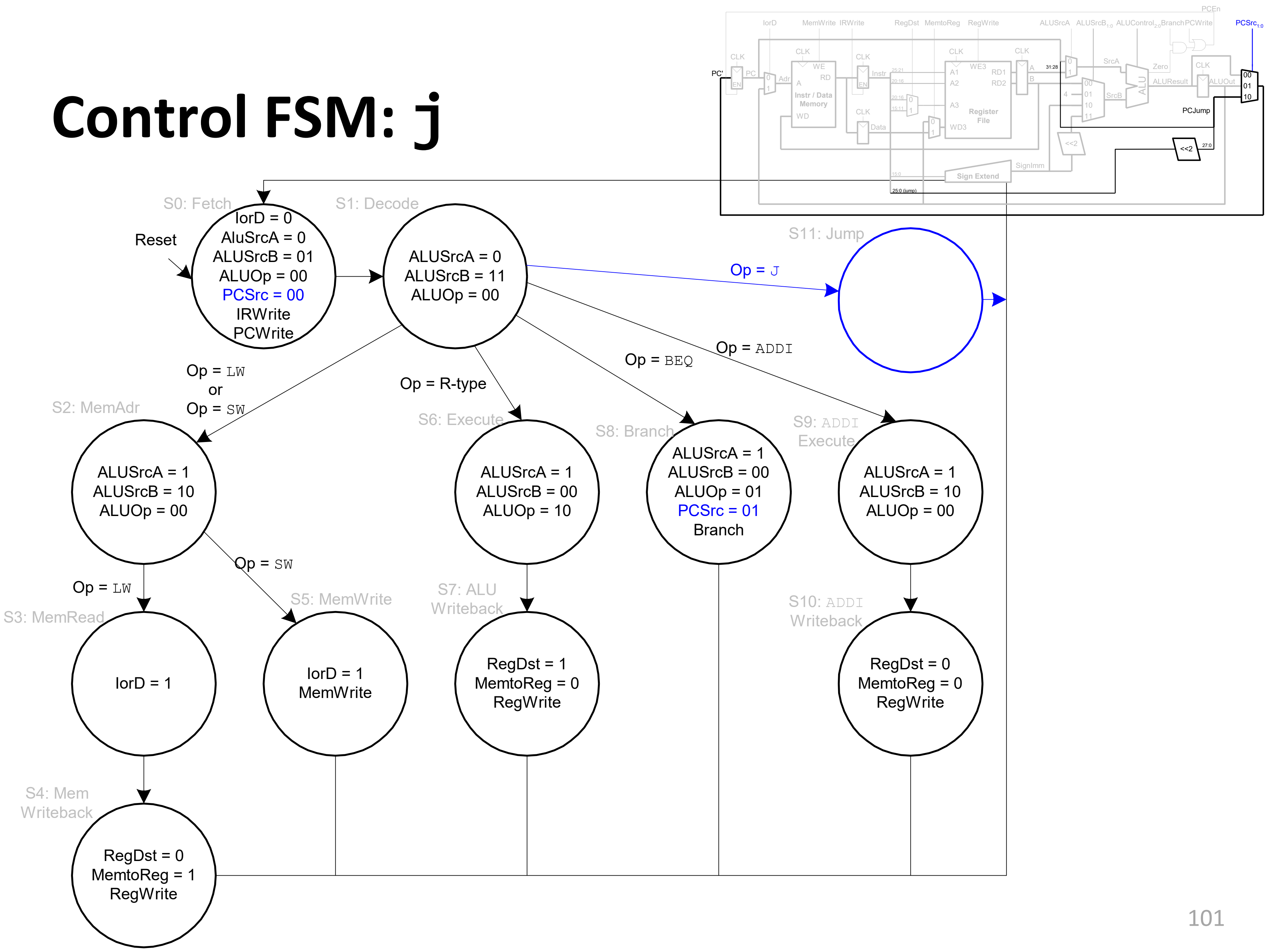
S4: Mem
Writeback

RegDst = 0
MemtoReg = 1
RegWrite

Extended Functionality: `j`



Control FSM: j



S0: Fetch

Reset

IorD = 0
AluSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCSrc = 00
IRWrite
PCWrite

S1: Decode

ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00

Op = J

S11: Jump

Op = LW
or
Op = SW

Op = R-type

Op = BEQ

Op = ADDI

S2: MemAdr

ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00

S6: Execute

ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10

S8: Branch

ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCSrc = 01
Branch

S9: ADDI Execute

ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00

Op = LW

Op = SW

S3: MemRead

IorD = 1

S5: MemWrite

IorD = 1
MemWrite

S7: ALU Writeback

RegDst = 1
MemtoReg = 0
RegWrite

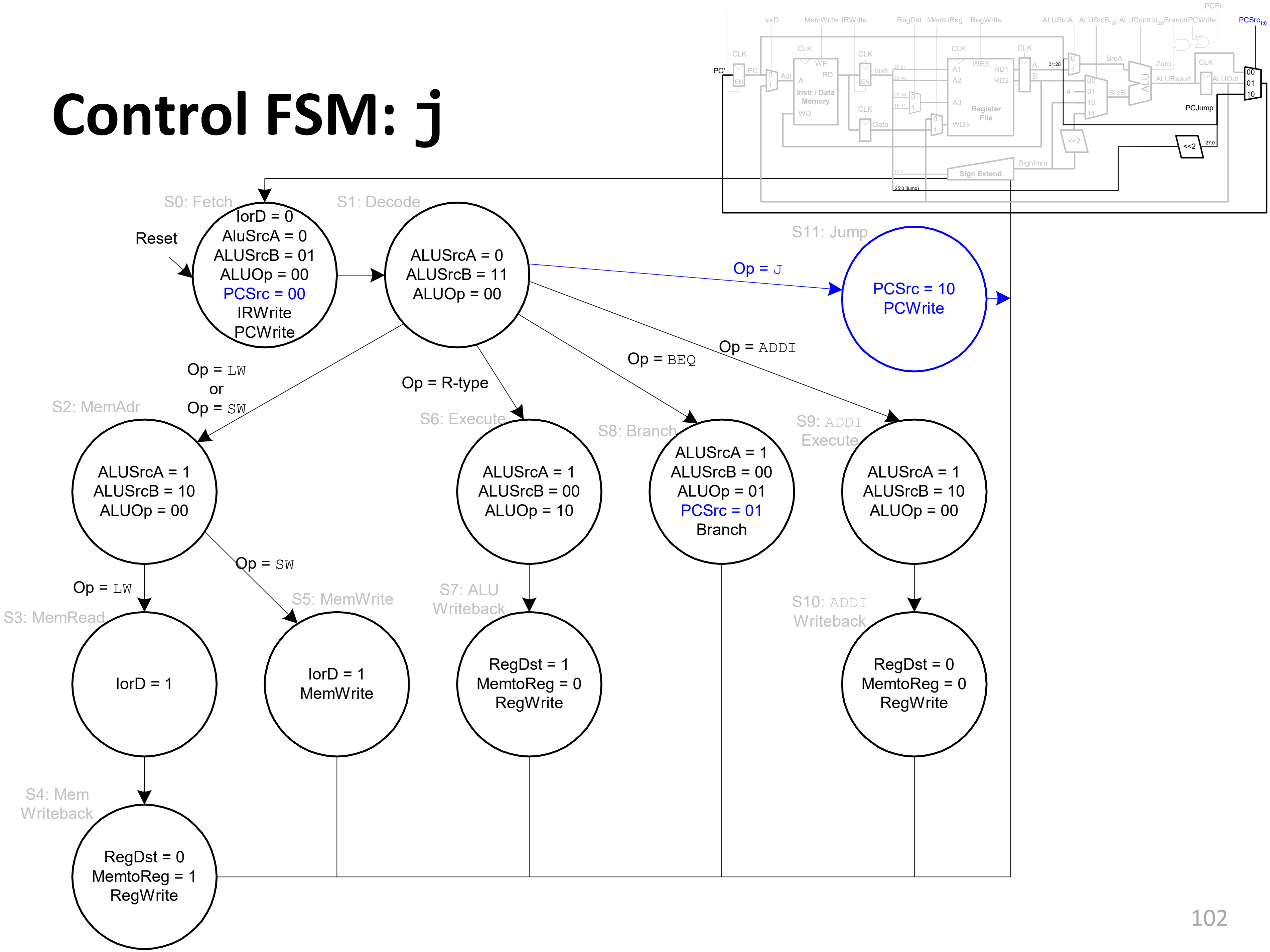
S10: ADDI Writeback

RegDst = 0
MemtoReg = 0
RegWrite

S4: Mem Writeback

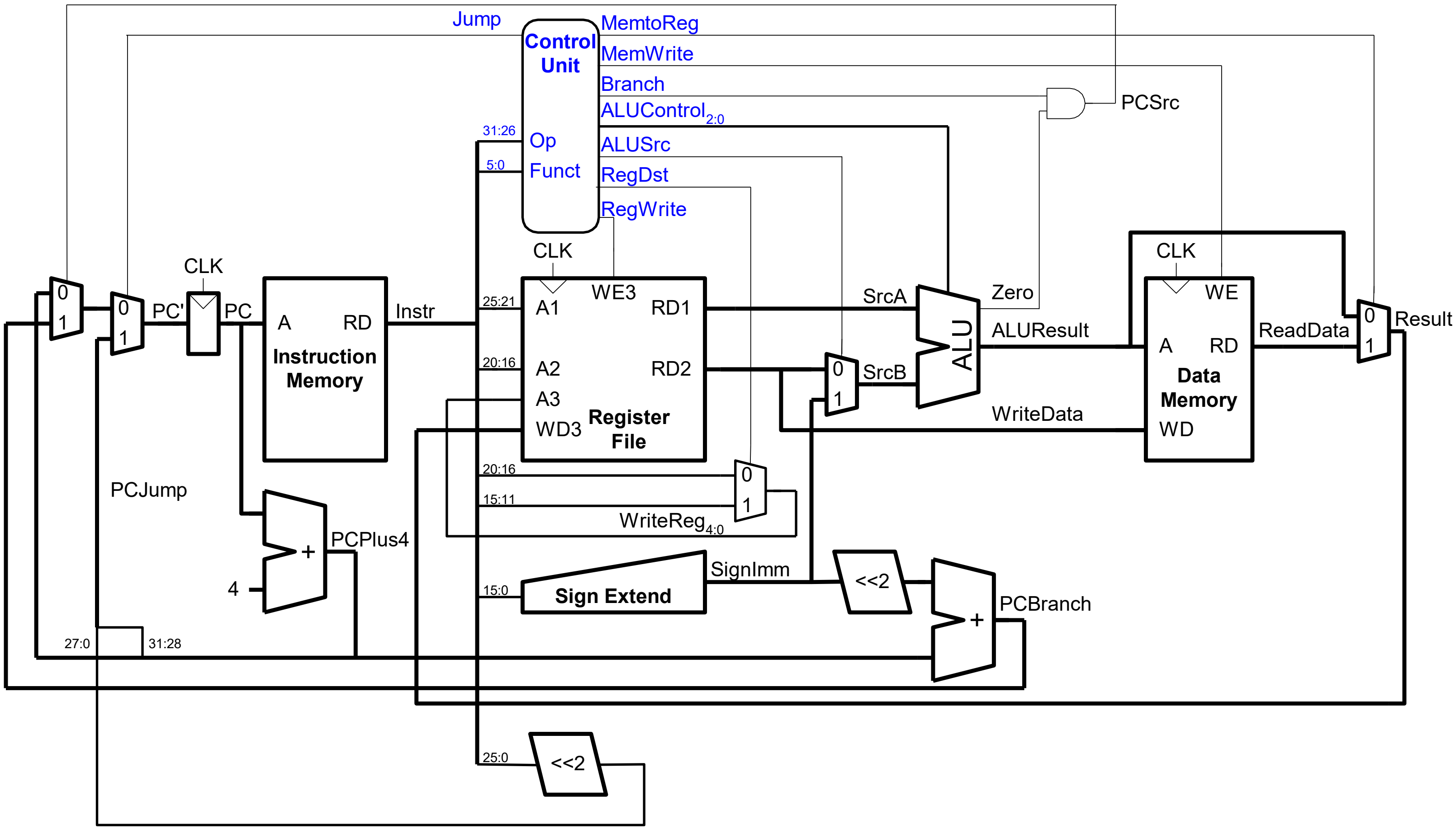
RegDst = 0
MemtoReg = 1
RegWrite

Control FSM: j



We Constructed a Multi-Cycle MIPS Microarchitecture

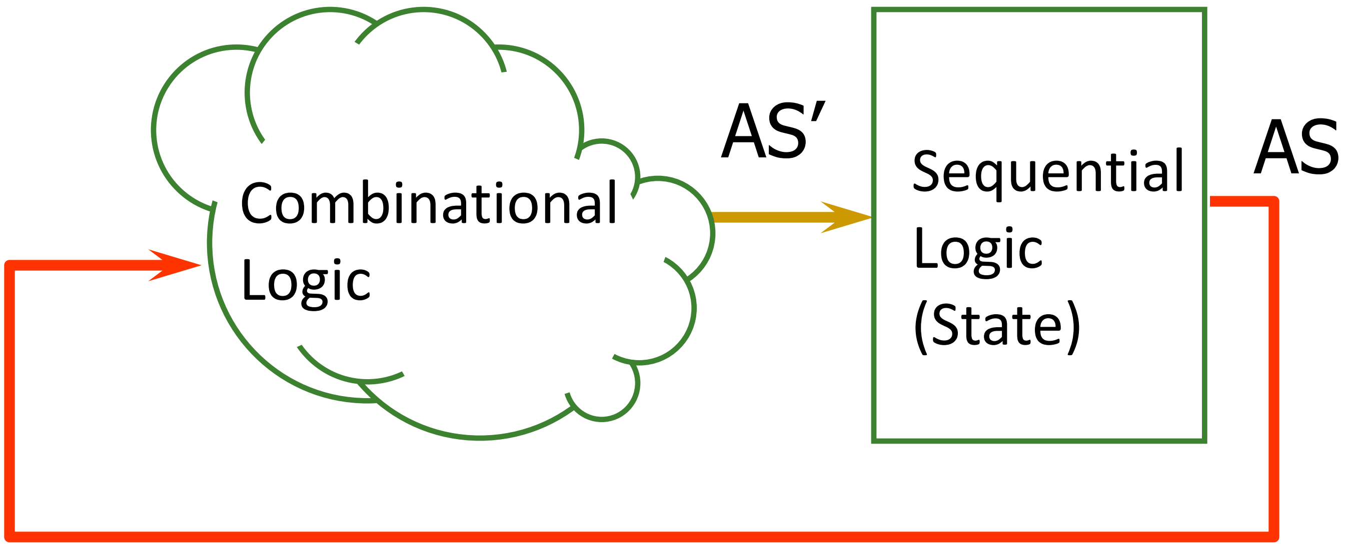
Review: Single-Cycle MIPS Microarchitecture



[Single-cycle processor.] Harris and Harris, Chapter 7.3.

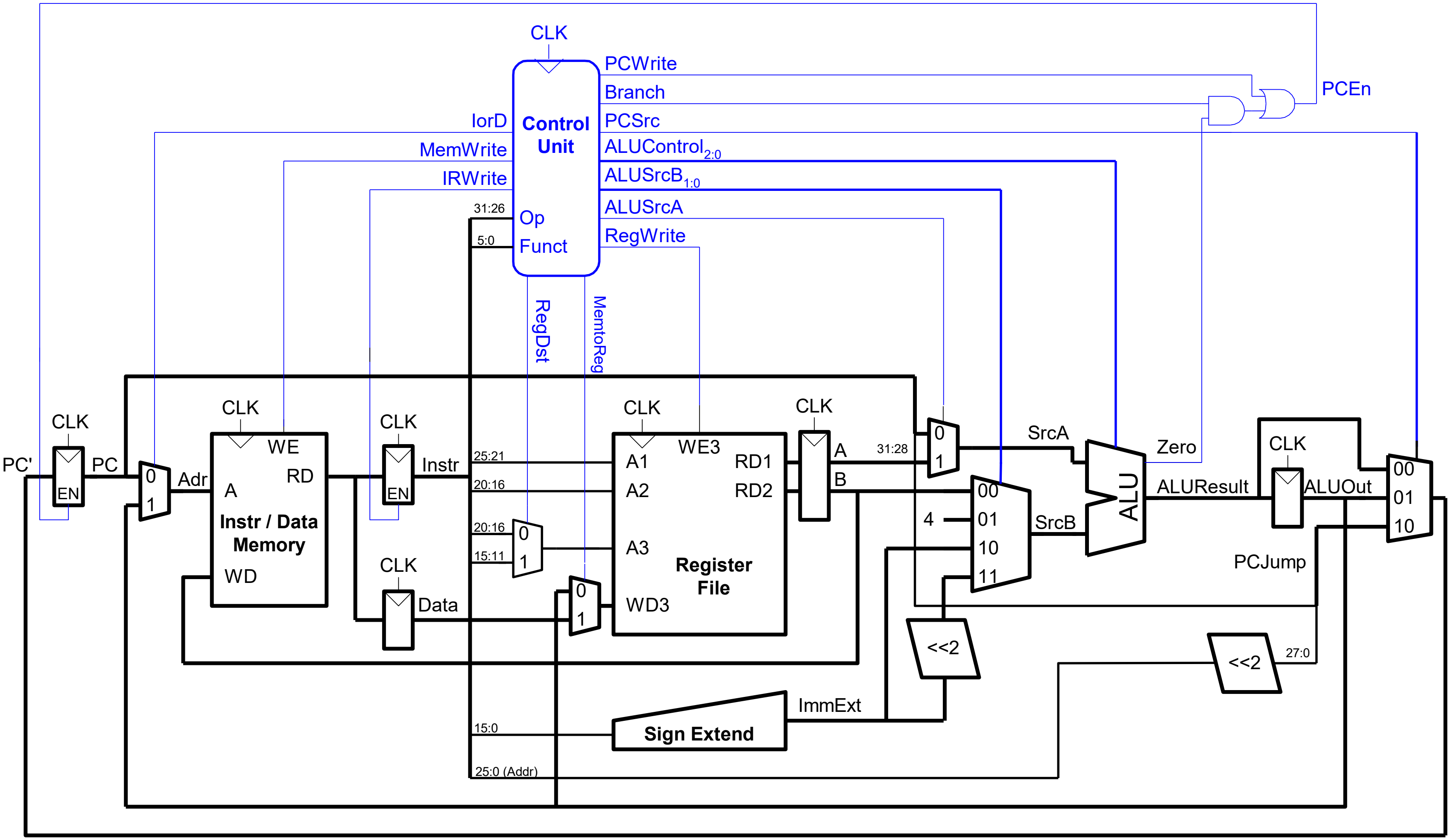
Review: Single-Cycle MIPS FSM

- Single-cycle machine



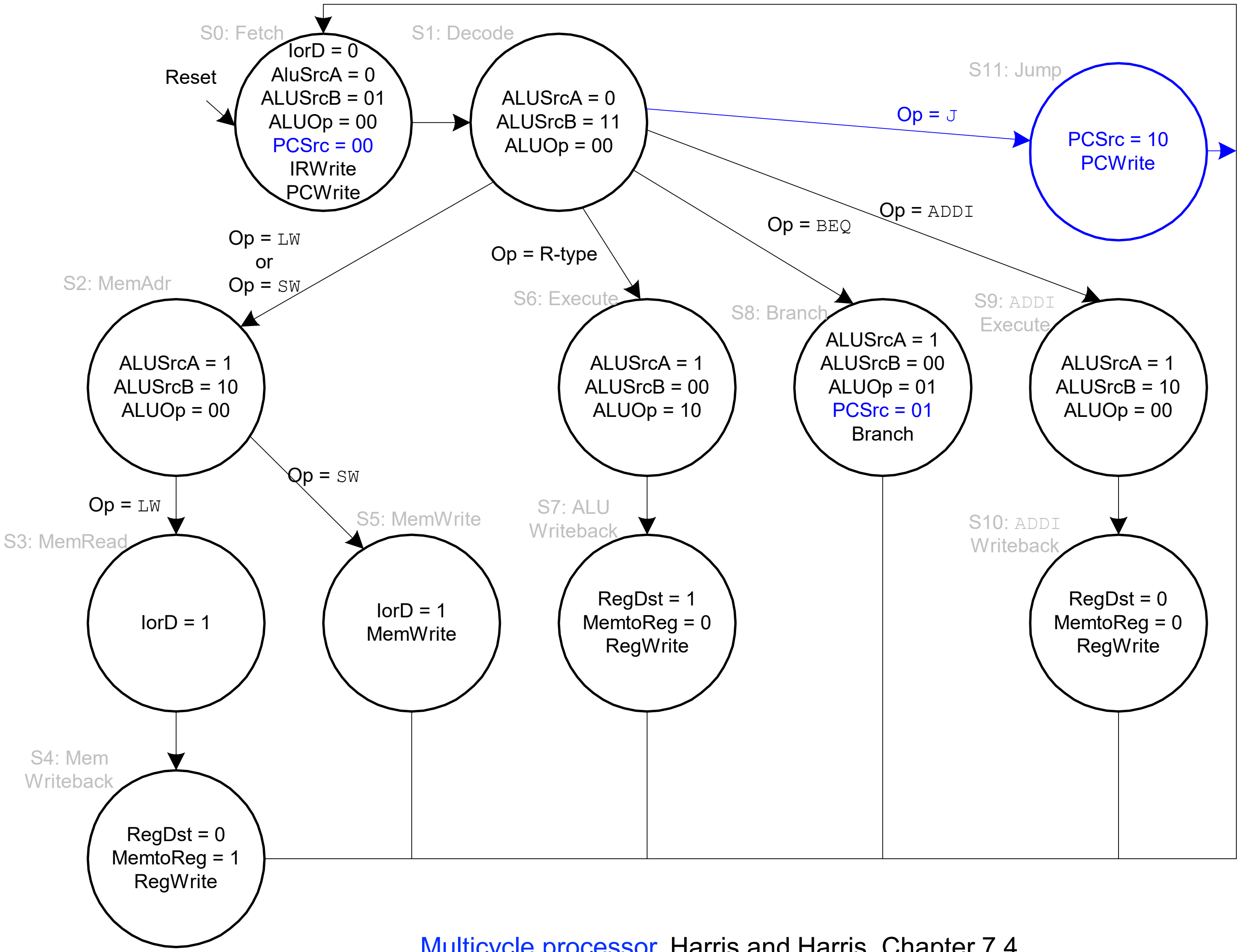
AS: Architectural State

Review: Multi-Cycle MIPS Microarchitecture



Multicycle processor. Harris and Harris, Chapter 7.4.

Review: Multi-Cycle MIPS FSM



What is the shortcoming of this design?

What does this design assume about memory?

Multicycle processor. Harris and Harris, Chapter 7.4.

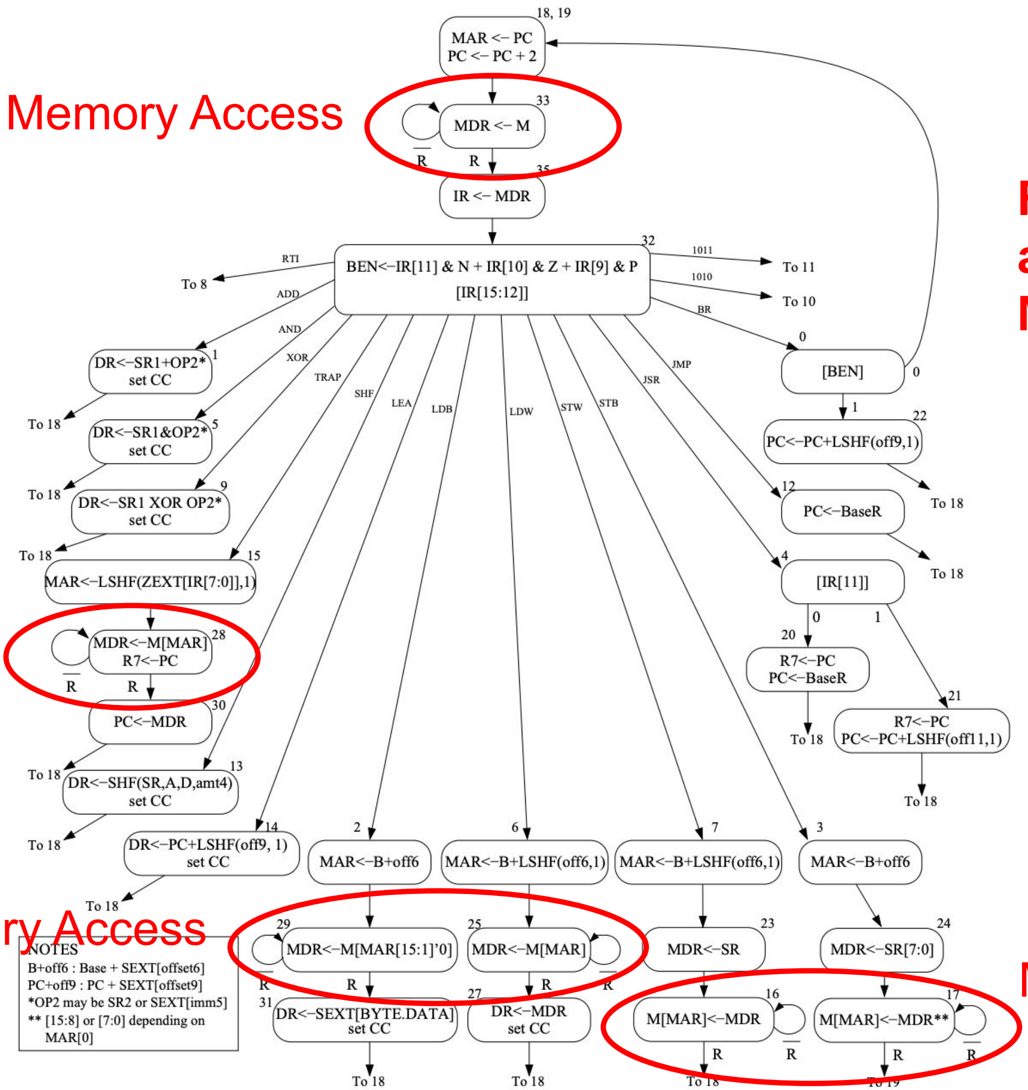
What If Memory Takes > One Cycle?

- Stay in the same “memory access” state until memory returns the data
- “Memory Ready?” bit is an input to the control logic that determines the next state

Recall: Full State Machine for LC-3b

Memory Access

Full FSM Controlling a Multi-Cycle LC-3b Microarchitecture



Memory Access

Memory Access

Memory Access

Figure C.2: A state machine for the LC-3b

https://safari.ethz.ch/digitaltechnik/spring2022/lib/exe/fetch.php?media=pp-appendixc.pdf

Another Example: **Microprogrammed** Multi-Cycle Microarchitecture

Recall: An Elegant Multi-Cycle Processor Design

- Maurice Wilkes, "The Best Way to Design an Automatic Calculating Machine," Manchester Univ. Computer Inaugural Conf., 1951.

THE BEST WAY TO DESIGN AN AUTOMATIC CALCULATING MACHINE

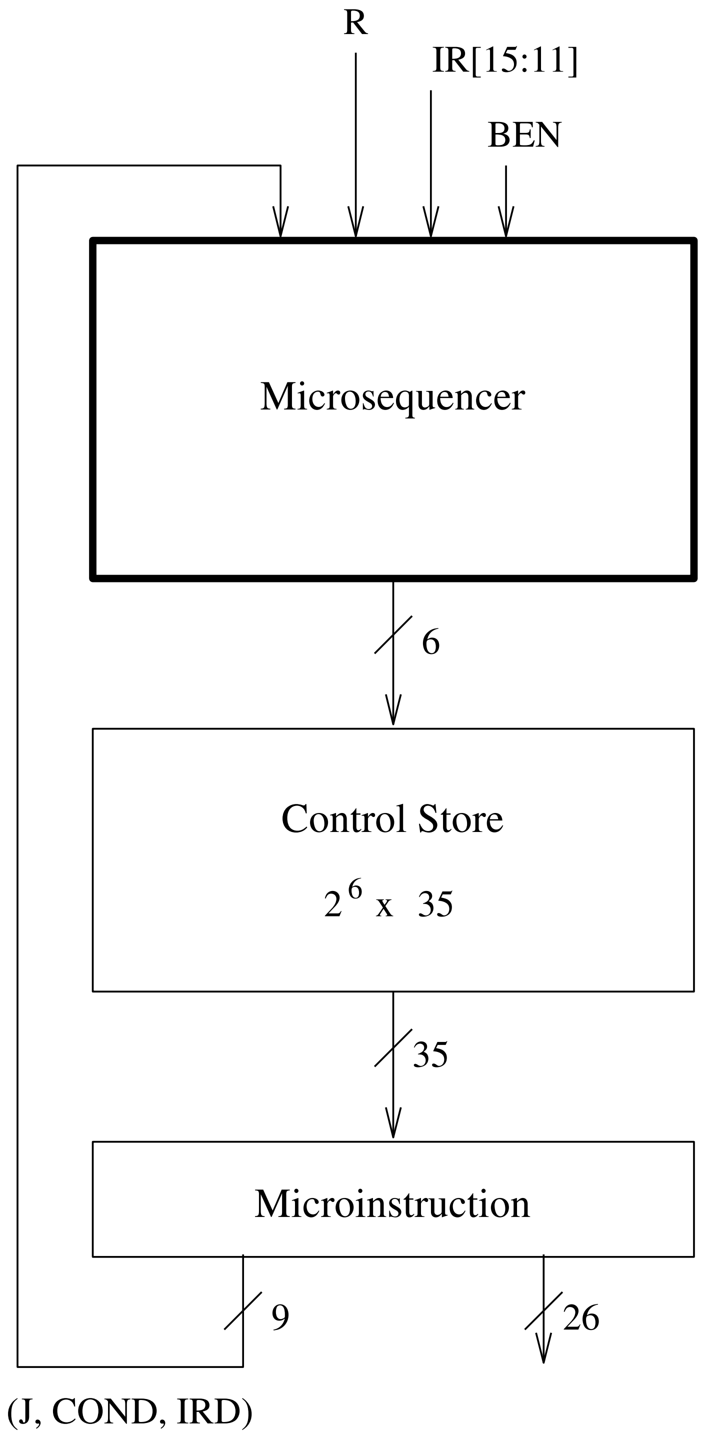
By M. V. Wilkes, M.A., Ph.D., F.R.A.S.

- An elegant implementation:
  - The concept of microcoded/microprogrammed machines

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)

Example Control Structure



Simple Design
of the Control Structure

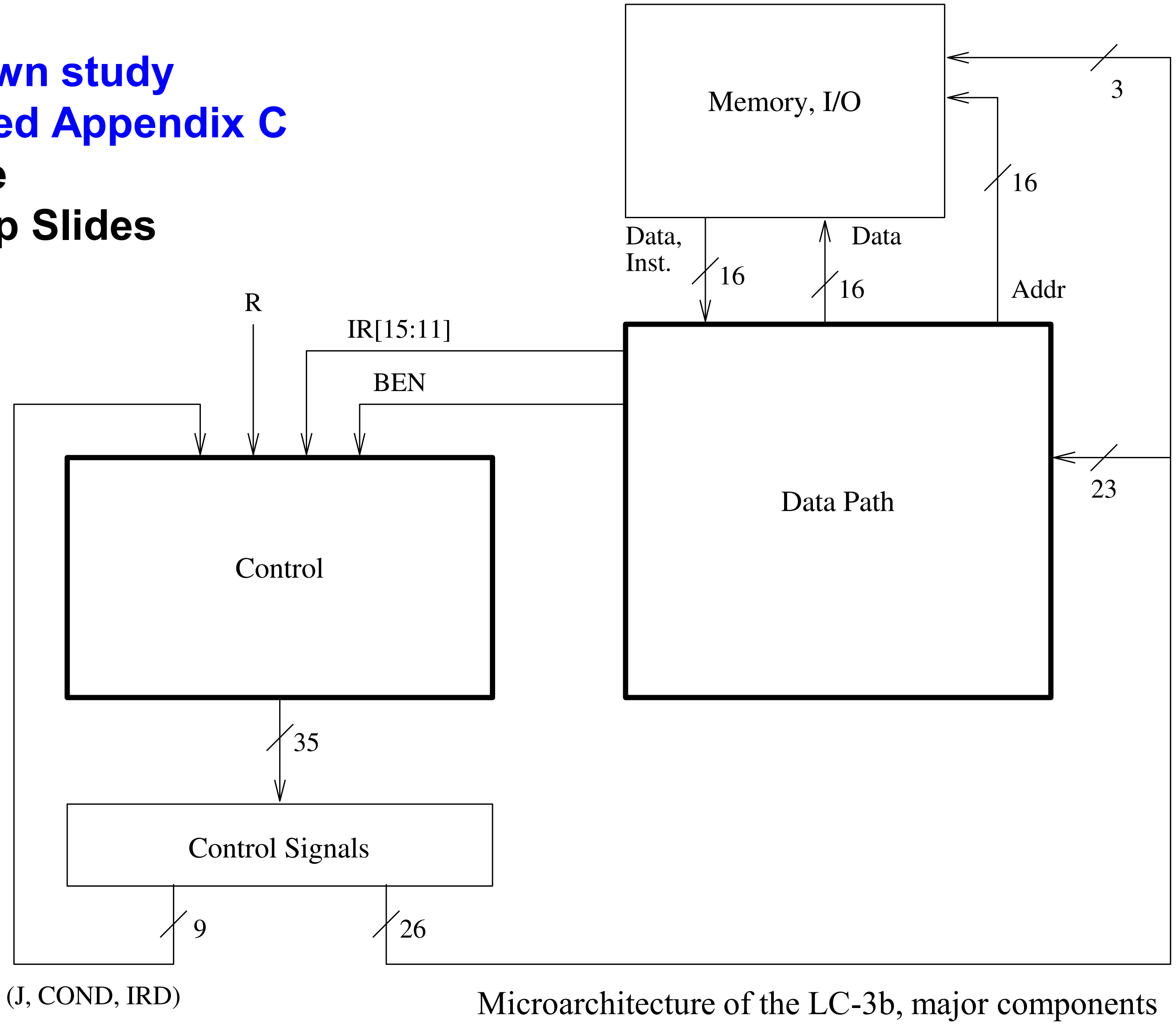
Example uProgrammed Control & Datapath

For your own study

P&P Revised Appendix C

On website

+ In Backup Slides



Microarchitecture of the LC-3b, major components

What Happens In A Clock Cycle?

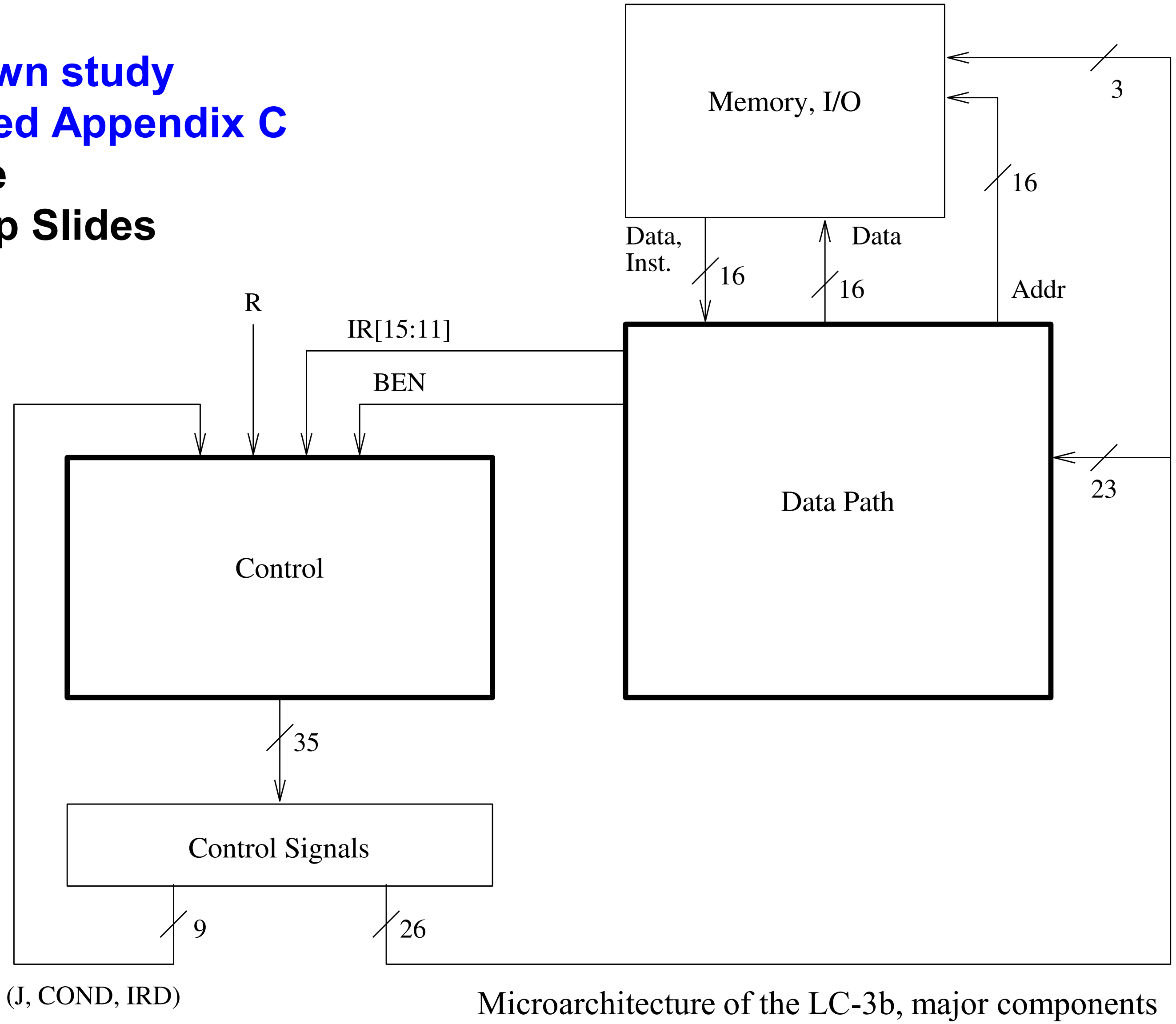
- The control signals (microinstruction) for the current state control two things:
  - Processing in the data path
  - Generation of control signals (microinstruction) for the next cycle

- Datapath and microsequencer operate concurrently

- Question: why not generate control signals for the current cycle in the current cycle?
  - This could lengthen the clock cycle

Example uProgrammed Control & Datapath

For your own study
P&P Revised Appendix C
On website
+ In Backup Slides



Microarchitecture of the LC-3b, major components

Microprogrammed Control Structure

- Three components: Microinstruction, Control store, Microsequencer

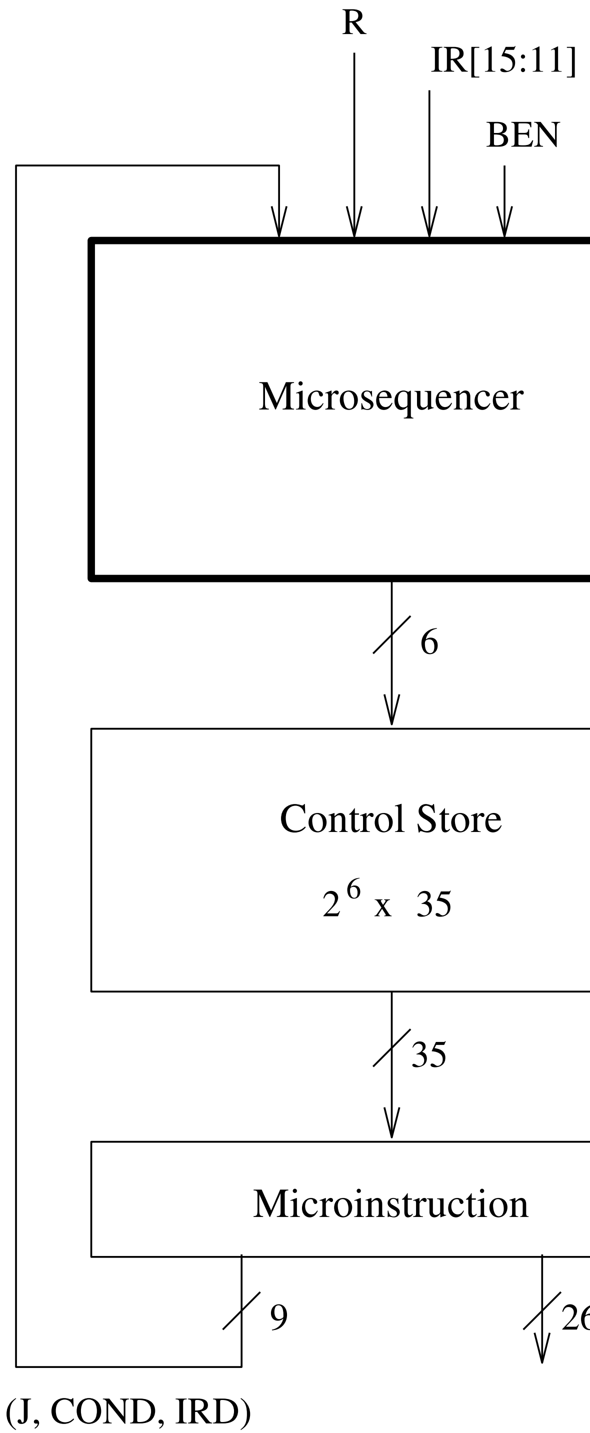
- Microinstruction: control signals that control the datapath (26 of them) and help determine the next state (9 of them)

- 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)

Patt and Patel, Appendix C, Figure C.4

uProgrammed Control Structure



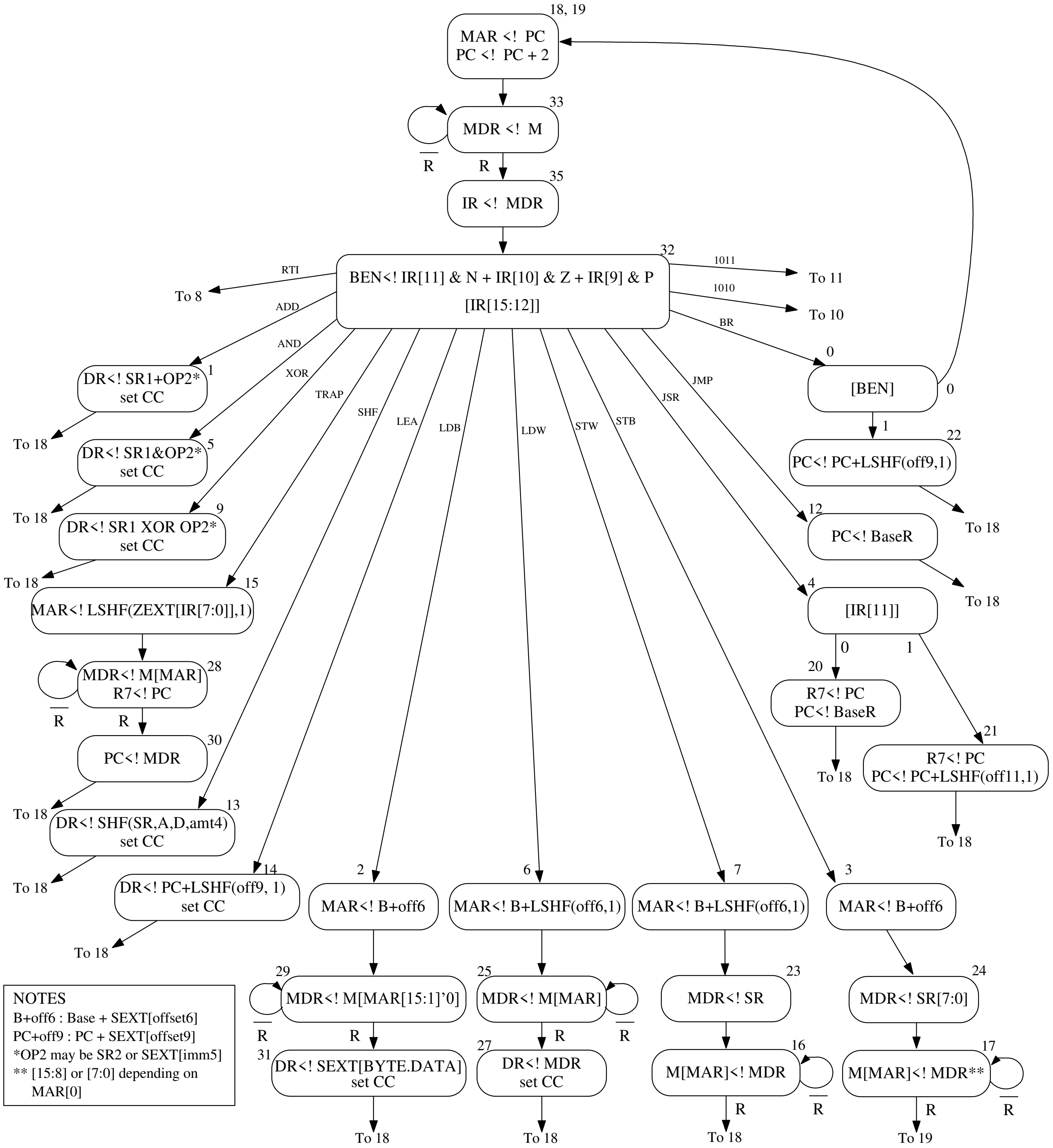
Simple Design
of the Control Structure

Control Store

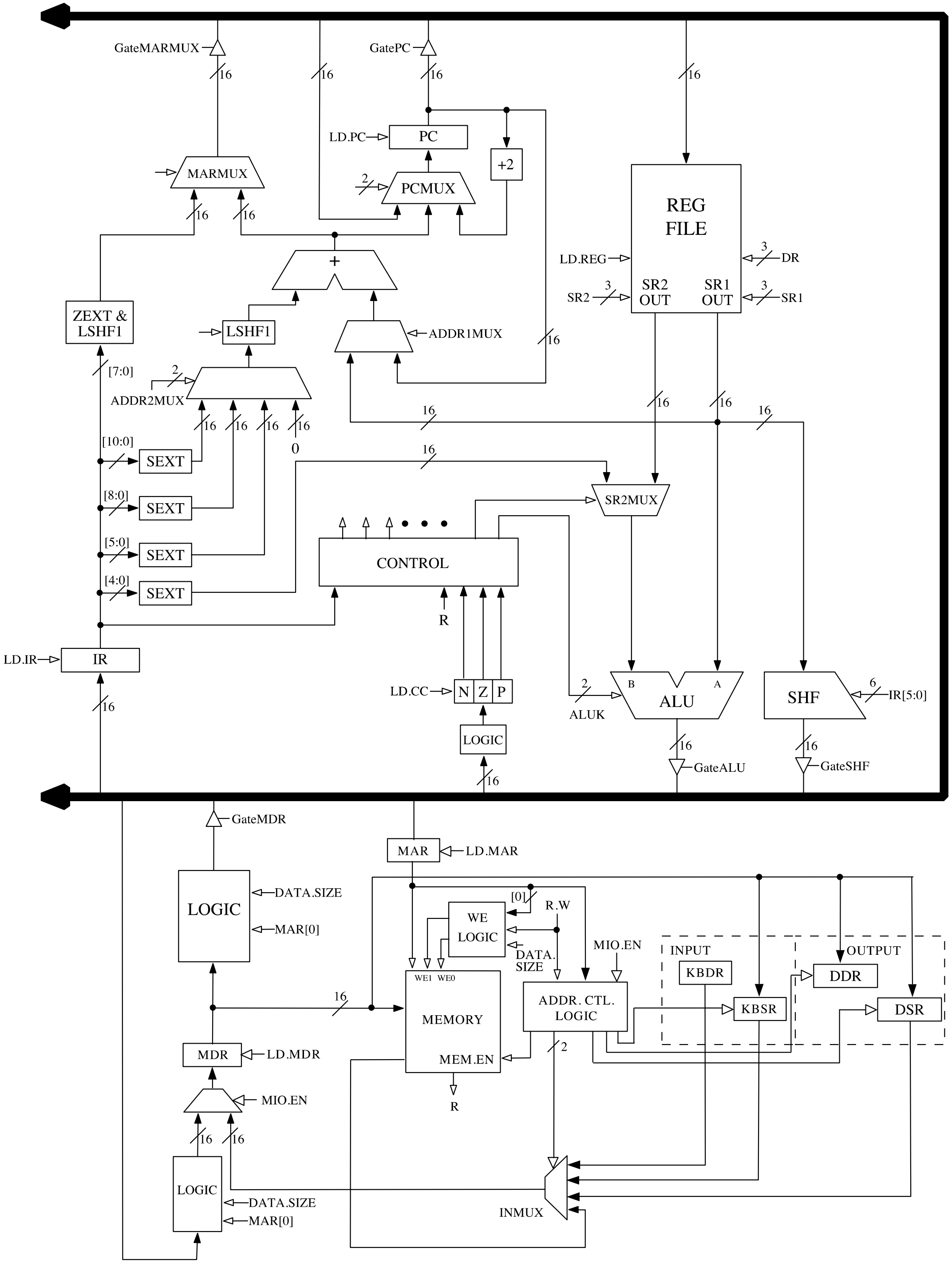
IRD	Cond	J	LD.MAR	LD.MDR	LD.IR	LD.BEN	LD.REG	LD.CC	LD.PC	GatePC	GateMDR	GateALU	GateMARMUX	GateSHF	PCMUX	DRMUX	SR1MUX	ADDR1MUX	ADDR2MUX	MARMUX	ALUK	MIO.EN	R.W	DATA.SIZE	LSHF1	
																										000000 (State 0)
																										000001 (State 1)
																										000010 (State 2)
																										000011 (State 3)
																										000100 (State 4)
																										000101 (State 5)
																										000110 (State 6)
																										000111 (State 7)
																										001000 (State 8)
																										001001 (State 9)
																										001010 (State 10)
																										001011 (State 11)
																										001100 (State 12)
																										001101 (State 13)
																										001110 (State 14)
																										001111 (State 15)
																										010000 (State 16)
																										010001 (State 17)
																										010010 (State 18)
																										010011 (State 19)
																										010100 (State 20)
																										010101 (State 21)
																										010110 (State 22)
																										010111 (State 23)
																										011000 (State 24)
																										011001 (State 25)
																										011010 (State 26)
																										011011 (State 27)
																										011100 (State 28)
																										011101 (State 29)
																										011110 (State 30)
																										011111 (State 31)
																										100000 (State 32)
																										100001 (State 33)
																										100010 (State 34)
																										100011 (State 35)
																										100100 (State 36)
																										100101 (State 37)
																										100110 (State 38)
																										100111 (State 39)
																										101000 (State 40)
																										101001 (State 41)
																										101010 (State 42)
																										101011 (State 43)
																										101100 (State 44)
																										101101 (State 45)
																										101110 (State 46)
																										101111 (State 47)
																										110000 (State 48)
																										110001 (State 49)
																										110010 (State 50)
																										110011 (State 51)
																										110100 (State 52)
																										110101 (State 53)
																										110110 (State 54)
																										110111 (State 55)
																										111000 (State 56)
																										111001 (State 57)
																										111010 (State 58)
																										111011 (State 59)
																										111100 (State 60)
																										111101 (State 61)
																										111110 (State 62)
																										111111 (State 63)

Each entry in the control store is a microinstruction corresponding to the FSM state

FSM state number is used to address the control store to get the relevant microinstruction

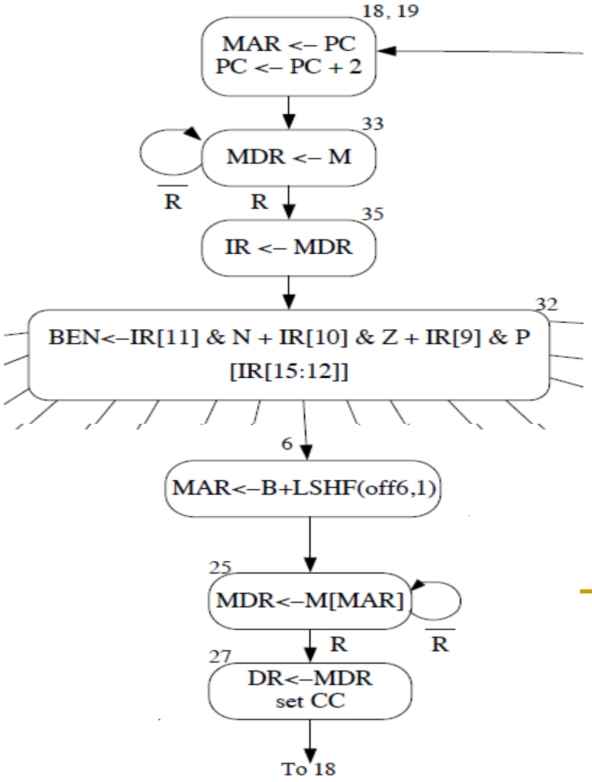


NOTES
B+off6 : Base + SEXT[offset6]
PC+off9 : PC + SEXT[offset9]
*OP2 may be SR2 or SEXT[imm5]
** [15:8] or [7:0] depending on
MAR[0]

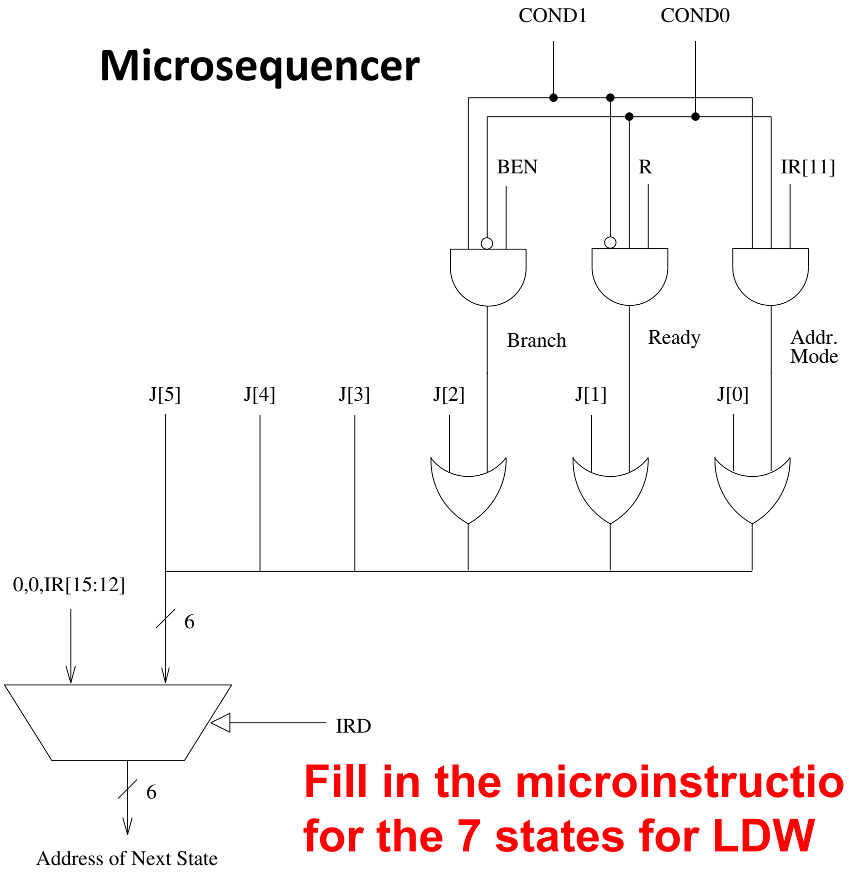


A Simple Datapath Can Become Very Powerful

State Machine for LDW



Microsequencer



Fill in the microinstructions for the 7 states for LDW

	IRD	Cond	J	LD.MAR	LD.MDR	LD.IR	LD.BEN	LD.REG	LD.CC	LD.PC	GatePC	GateMDR	GateALU	GateMARMUX	GateSHF	PCMUX	DRMUX	SR1MUX	ADDR1MUX	ADDR2MUX	MARMUX	ALUK	MIO.EN	R.W	DATA.SIZE	LSHF1
State 18 (010010)																										
State 33 (100001)																										
State 35 (100011)																										
State 32 (100000)																										
State 6 (000110)																										
State 25 (011001)																										
State 27 (011011)																										

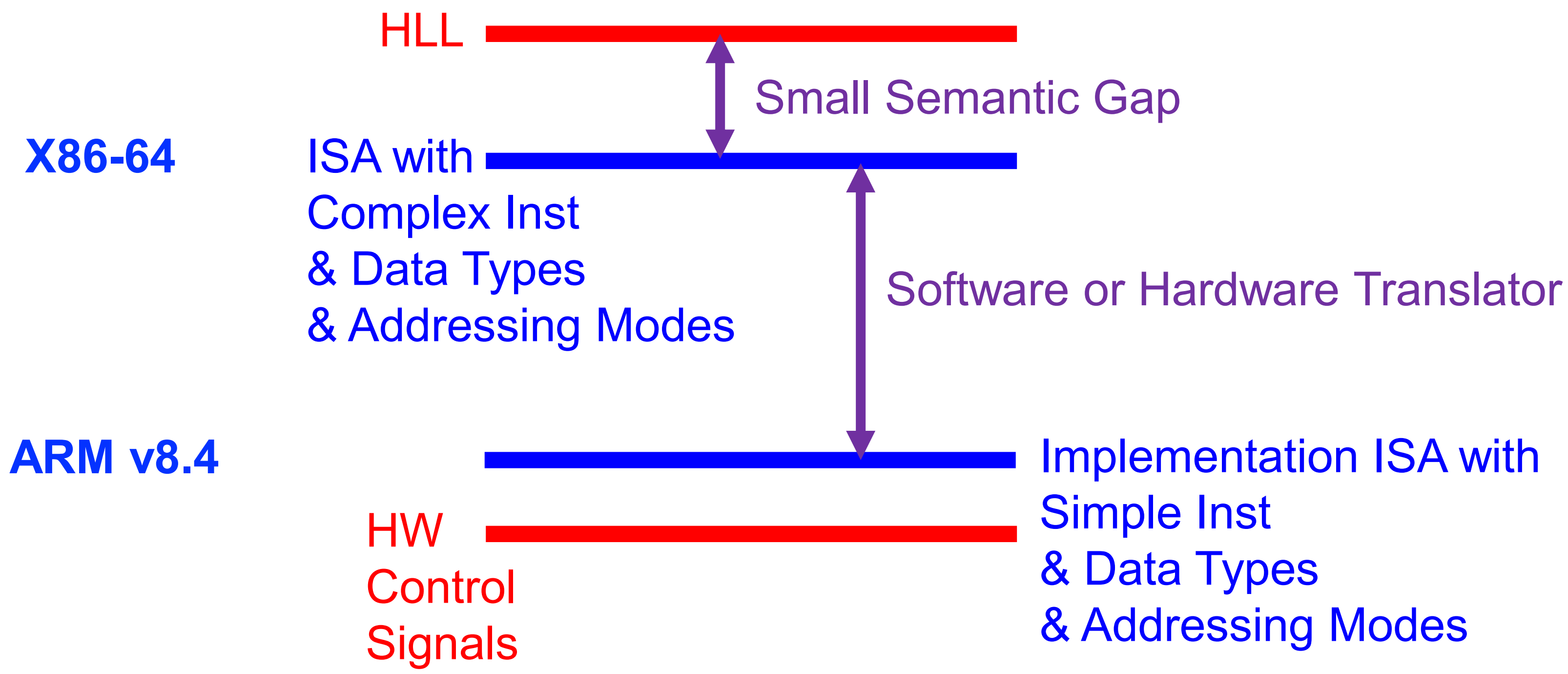
The Power of Abstraction

- 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)

Recall: How to Change the Semantic Gap Tradeoffs

- Translate from one ISA into a different "implementation" ISA



HLL

Small Semantic Gap

X86-64

ISA with
Complex Inst
& Data Types
& Addressing Modes

Software or Hardware Translator

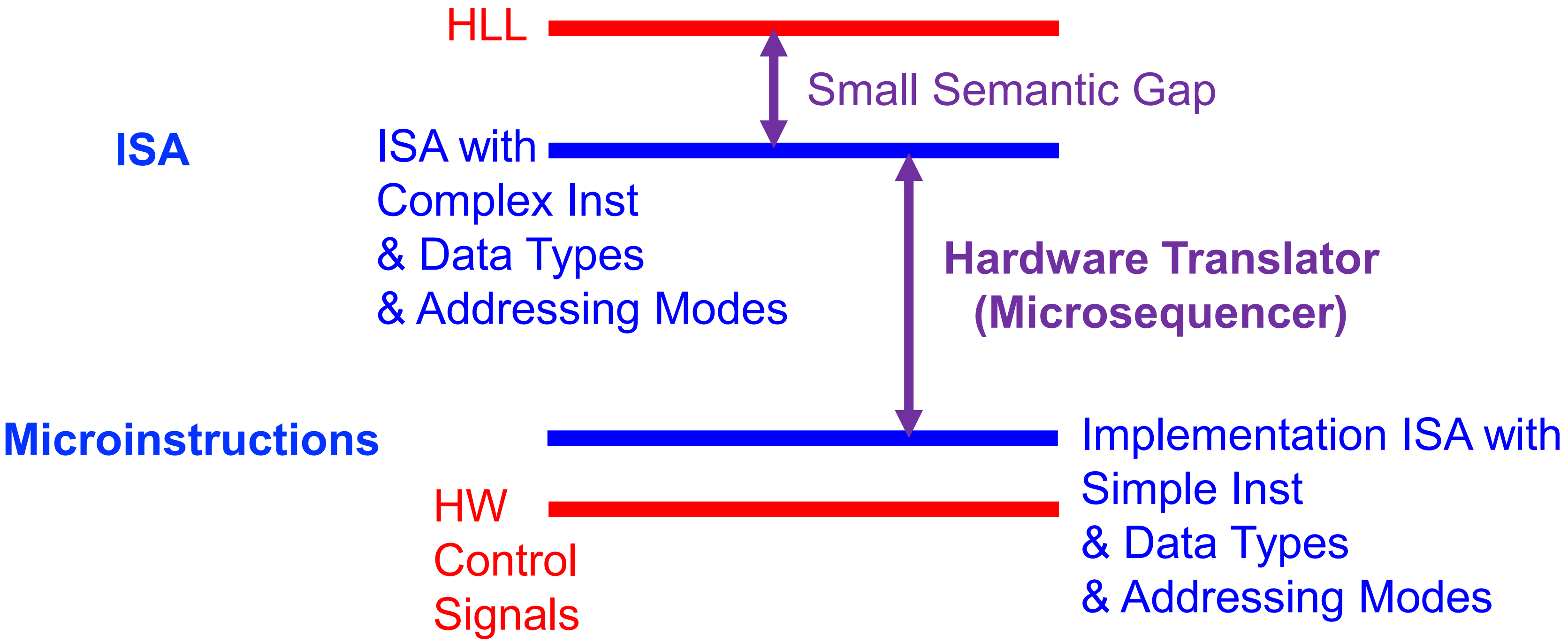
ARM v8.4

Implementation ISA with
Simple Inst
& Data Types
& Addressing Modes

HW
Control
Signals

How to Change the Semantic Gap Tradeoffs

- Translate from one ISA into a different "implementation" ISA



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)

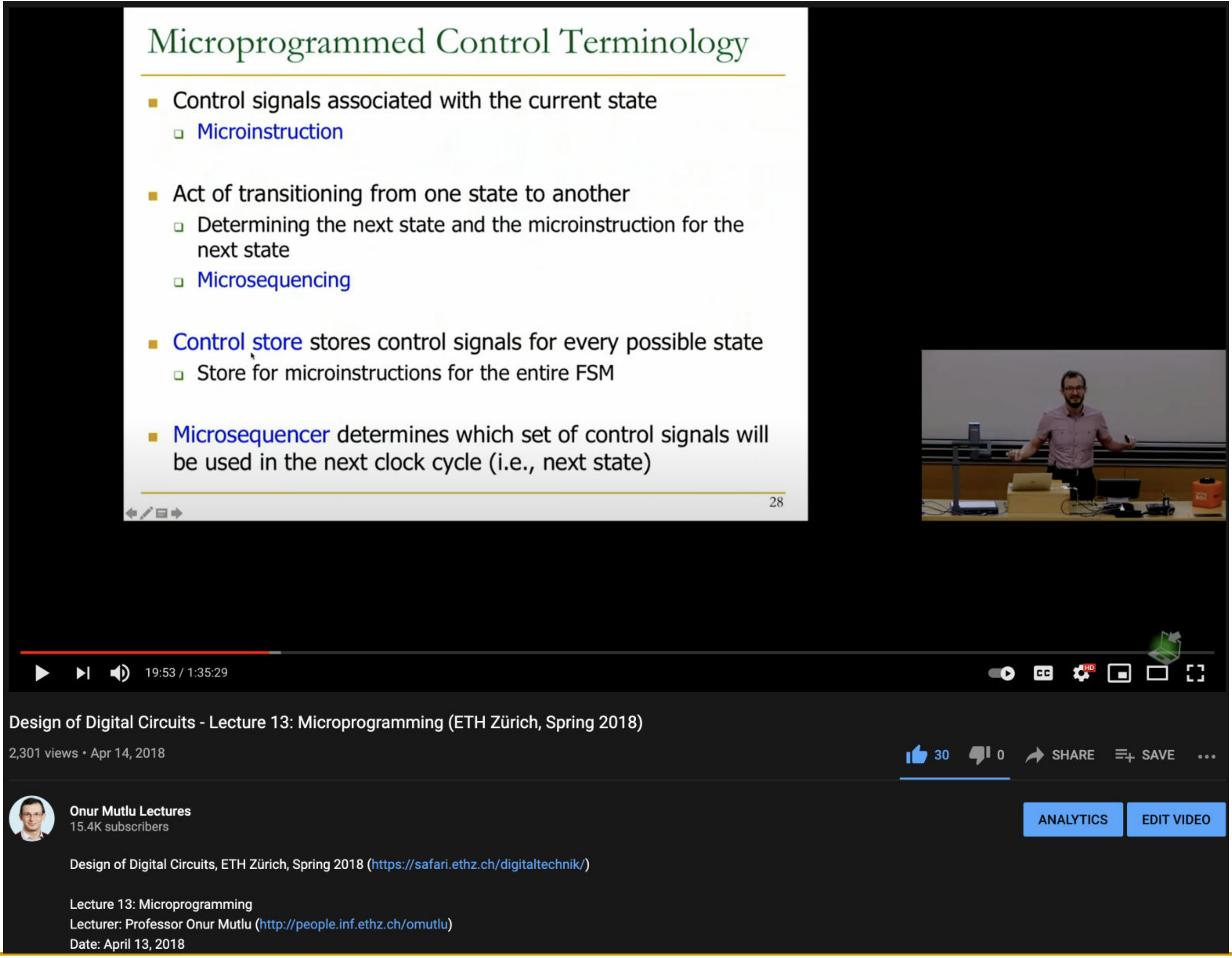
- Enables update of machine behavior
  - A buggy implementation of an instruction can be fixed by changing the microcode in the field
    - Easier if datapath provides ability to do the same thing in different ways

Update of Machine Behavior

- The ability to update/patch microcode in the field (after a processor is shipped) enables
  - Ability to add new instructions without changing the processor!
  - Ability to "fix" buggy hardware implementations

- Historical Examples
  - IBM 370 Model 145: microcode stored in main memory, can be updated after a reboot
  - IBM System z: Similar to 370/145.
    - Heller and Farrell, "Millicode in an IBM zSeries processor," IBM JR&D, May/Jul 2004.
  - B1700 microcode can be updated while the processor is running
    - User-microprogrammable machine!
    - Wilner, "Microprogramming environment on the Burroughs B1700", CompCon 1972.
  - Systems today use microcode patches to fix HW bugs/issues

For More on Microprogrammed Designs



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)

28

Design of Digital Circuits - Lecture 13: Microprogramming (ETH Zürich, Spring 2018)

2,301 views • Apr 14, 2018

Onur Mutlu Lectures
15.4K subscribers

Design of Digital Circuits, ETH Zürich, Spring 2018 (https://safari.ethz.ch/digitaltechnik/)

Lecture 13: Microprogramming
Lecturer: Professor Onur Mutlu (http://people.inf.ethz.ch/omutlu)
Date: April 13, 2018

https://www.youtube.com/onurmutlulectures

Detailed Lectures on Microprogramming

- Design of Digital Circuits, Spring 2018, Lecture 13
  - Microprogramming (ETH Zürich, Spring 2018)
  - https://www.youtube.com/watch?v=u4GhShuBP3Y&list=PL5Q2soXY2Zi_QedyPWtRmFUJ2F8DdYP7l&index=13

- Computer Architecture, Spring 2013, Lecture 7
  - Microprogramming (CMU, Spring 2013)
  - https://www.youtube.com/watch?v=_igvSl5h8cs&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=7

Digital Design & Computer Arch.

Lecture 11: Multi-Cycle Microarchitecture Design

Prof. Onur Mutlu

ETH Zürich
Spring 2023
30 March 2023

Backup Slides

A Bit More on Performance Analysis

How can I Make the Program Run Faster?

N x CPI x (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 – complex instruction sets)
  - 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 – complex instruction sets)
  - Use better compilers

- **Use fewer cycles to perform each instruction**
  - Simpler instructions (RISC – reduced instruction sets)
  - Use multiple units/ALUs/cores in parallel

How can I Make the Program Run Faster?

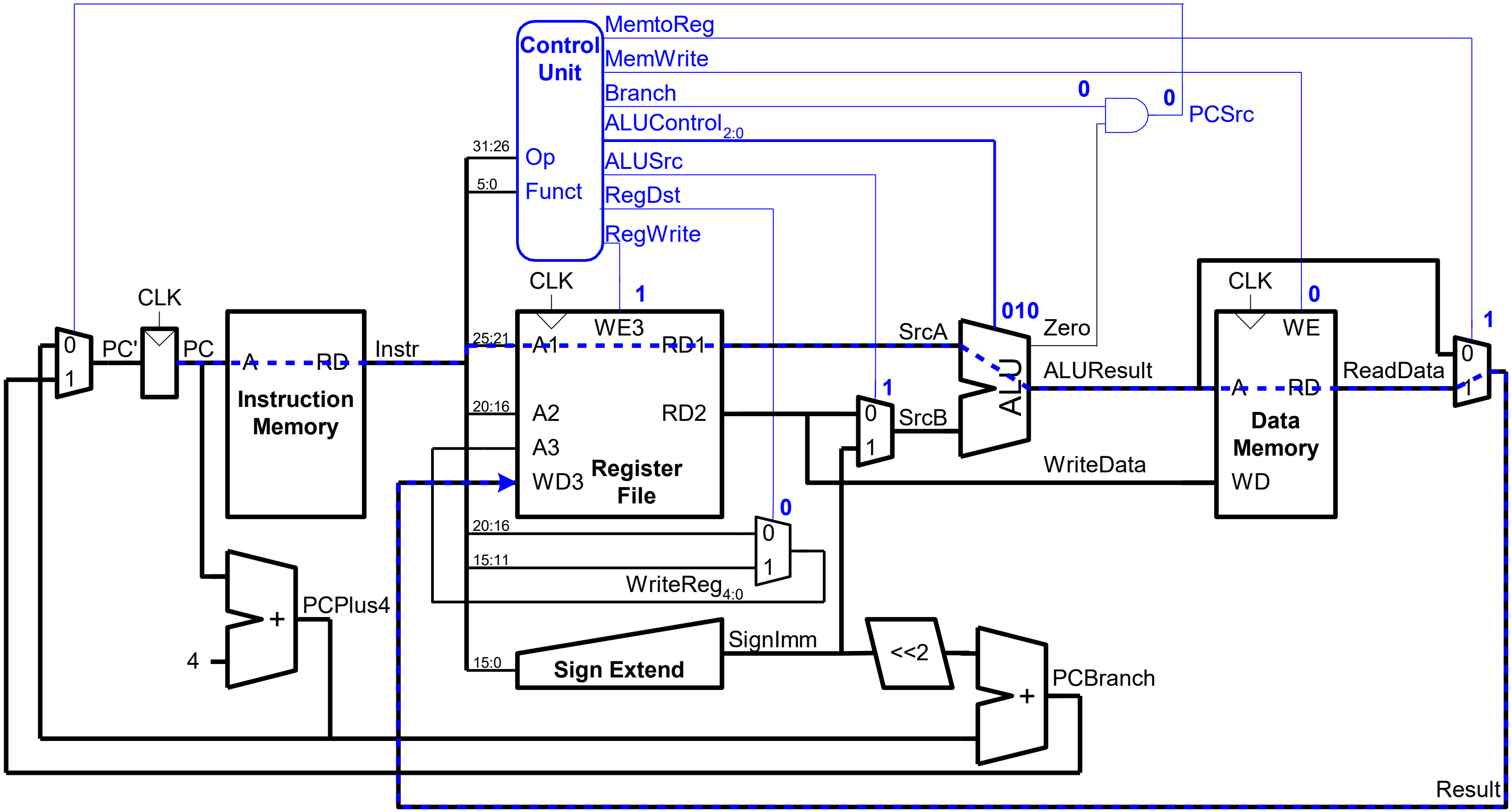
N x CPI x (1/f)

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

Performance Analysis of Single-Cycle vs. Multi-Cycle Designs

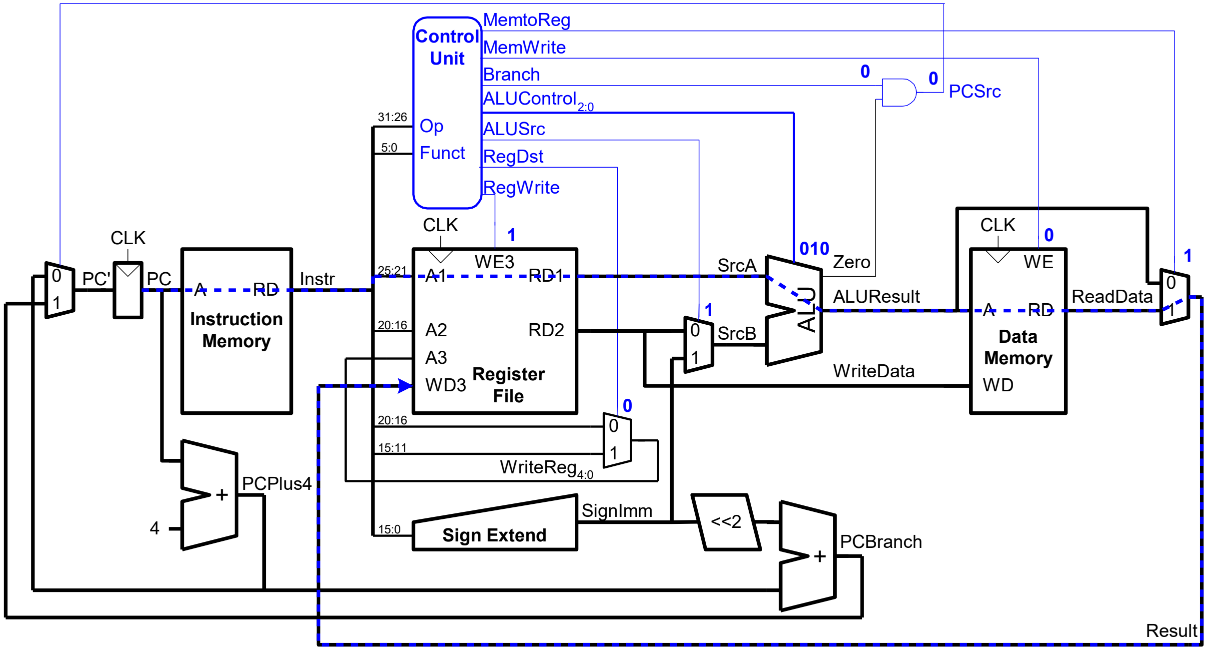
Single-Cycle Performance

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



Single-Cycle Performance

- Single-cycle critical path:
  - $\mathbf{T_c = t_{pcq\_PC} + t_{mem} + \max(t_{RFread}, t_{sext} + t_{mux}) + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup}}$
- 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

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

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 $= \# \text{ instructions} \times \text{CPI} \times T_c$

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

$= 92.5$ seconds

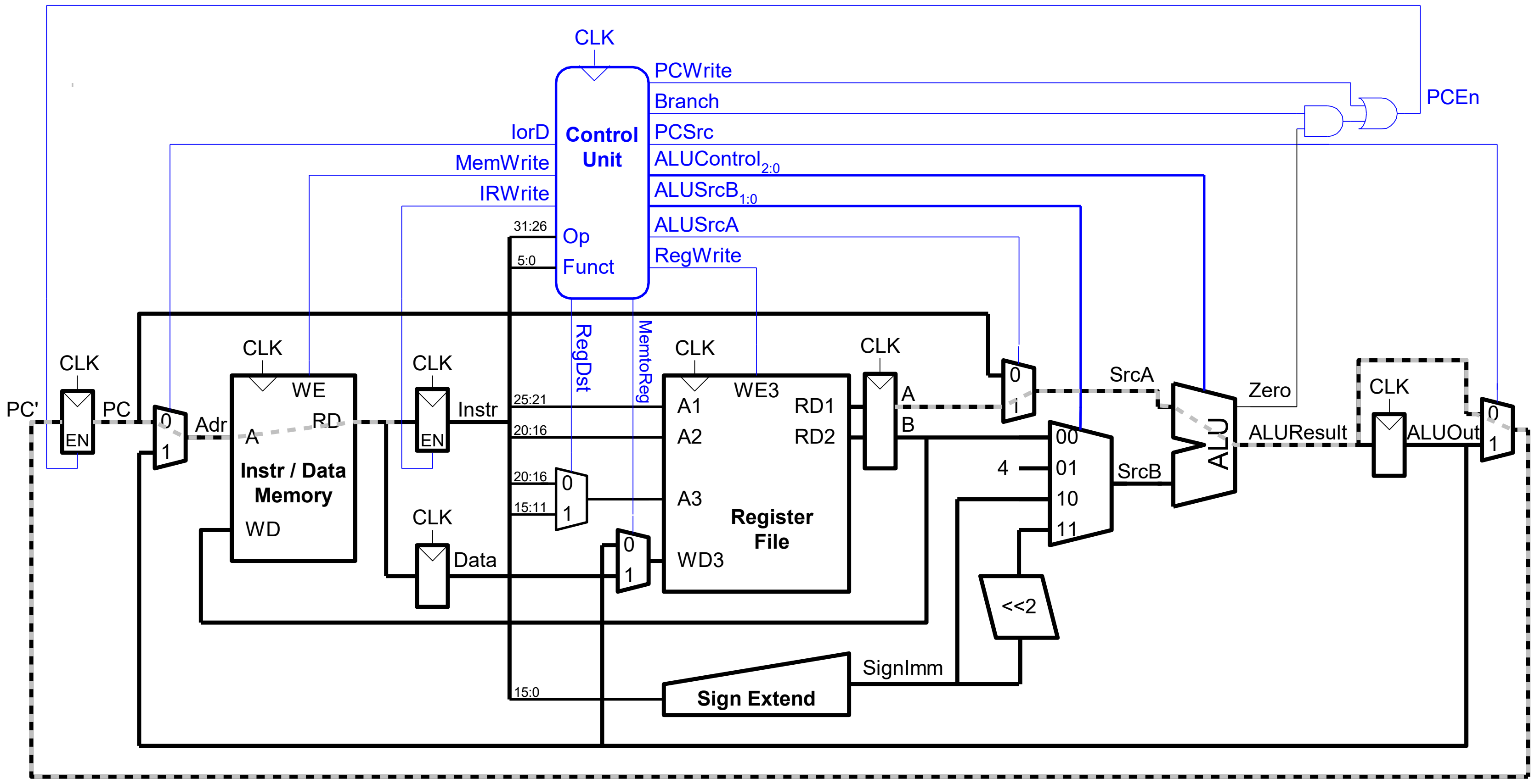
Multi-Cycle Performance: CPI

- Instructions take different number of cycles:
  - 3 cycles: `beq, j`
  - 4 cycles: `R-Type, sw, addi`
  - 5 cycles: `lw` **Realistic?**
- CPI is weighted average, e.g. SPECINT2000 benchmark:
  - 25% loads
  - 10% stores
  - 11% branches
  - 2% jumps
  - 52% R-type
- *Average CPI* = (0.11 + 0.02) 3 +(0.52 + 0.10) 4 +(0.25) 5
  = 4.12

Multi-Cycle Performance: Cycle Time

- Multi-cycle critical path:

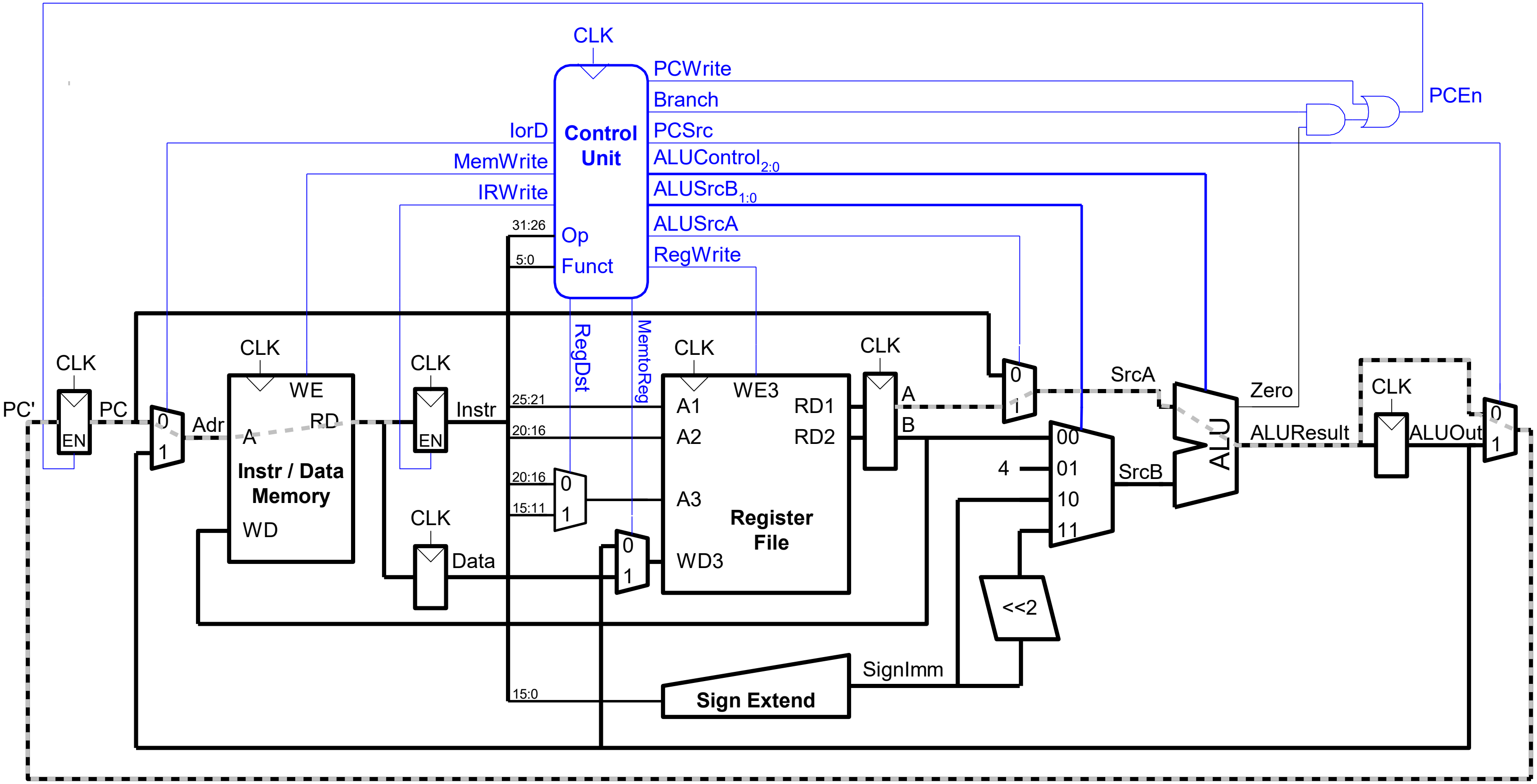
  $T_c =$



Multi-Cycle Performance: Cycle Time

- Multi-cycle critical path:

$$T_c = t_{pcq} + t_{mux} + \max(t_{ALU} + t_{mux}, t_{mem}) + t_{setup}$$



Multi-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 =

Multi-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

$$T_c = t_{pcq\_PC} + t_{mux} + \max(t_{ALU} + t_{mux}, t_{mem}) + t_{setup}$$

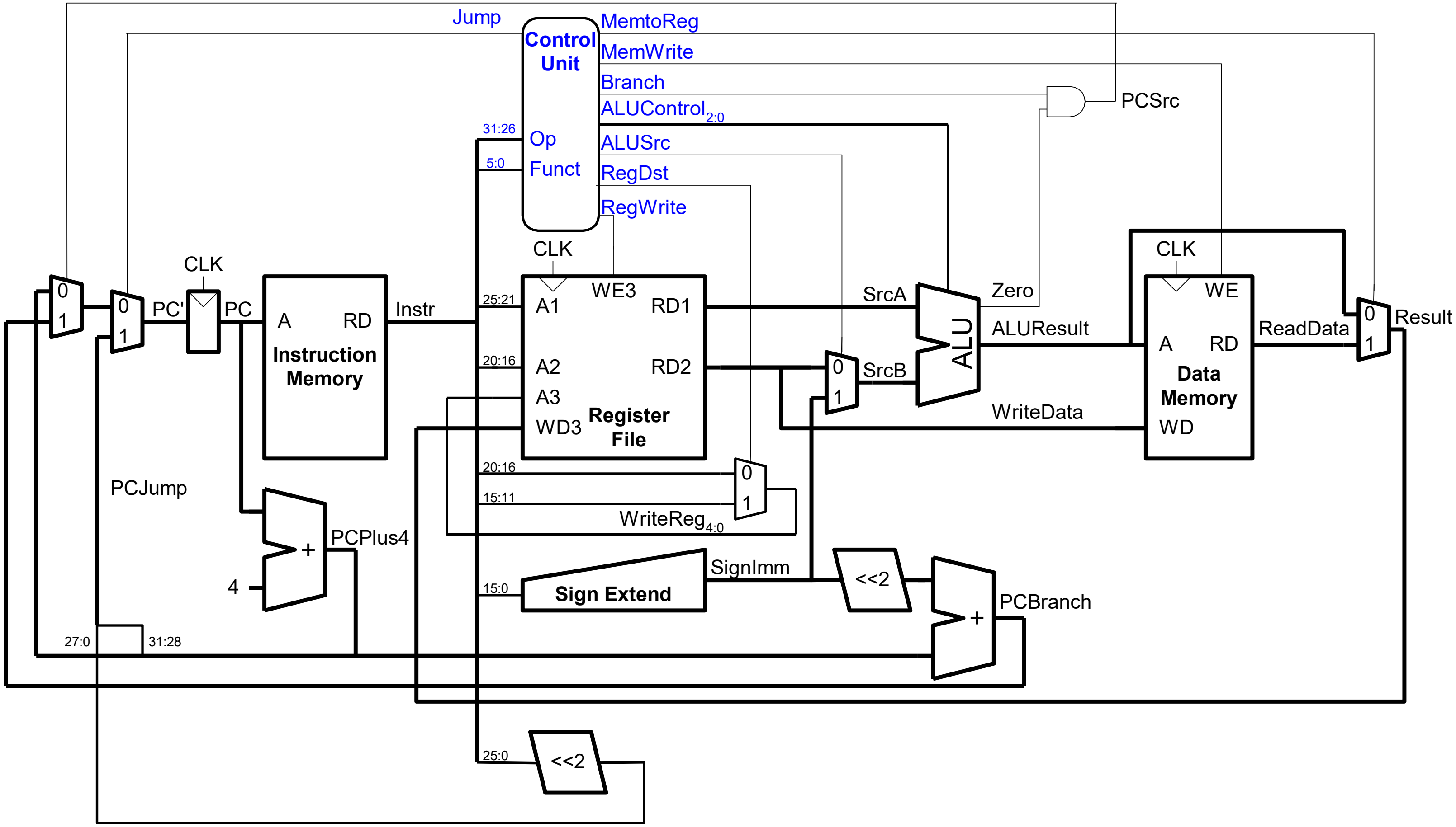
$$= [30 + 25 + 250 + 20] \text{ ps}$$

$$= 325 \text{ ps}$$

Multi-Cycle Performance Example

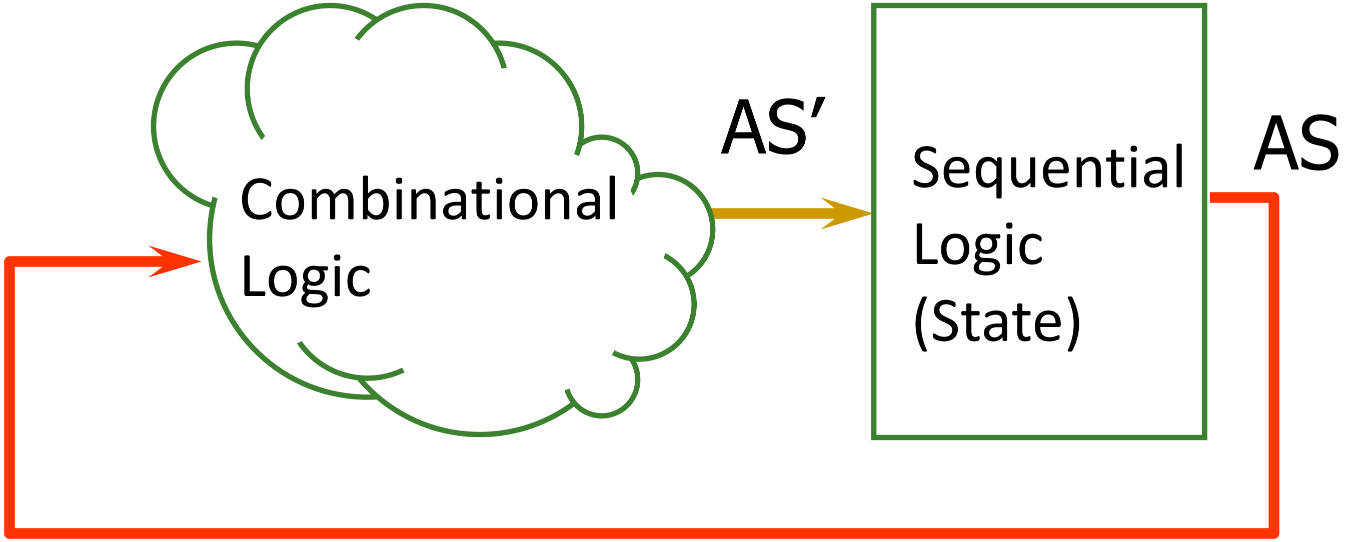
- For a program with 100 billion instructions executing on a multi-cycle MIPS processor
  - CPI = 4.12
  - T_c = 325 ps
- *Execution Time* = (# instructions) × CPI × T_c
  = $(100 \times 10^9)(4.12)(325 \times 10^{-12})$
  = 133.9 seconds
- This is slower than the single-cycle processor (92.5 seconds). Why?
- Did we break the stages in a balanced manner?
- Overhead of register setup/hold paid many times
- How would the results change with different assumptions on memory latency and instruction mix?

Review: Single-Cycle MIPS Processor



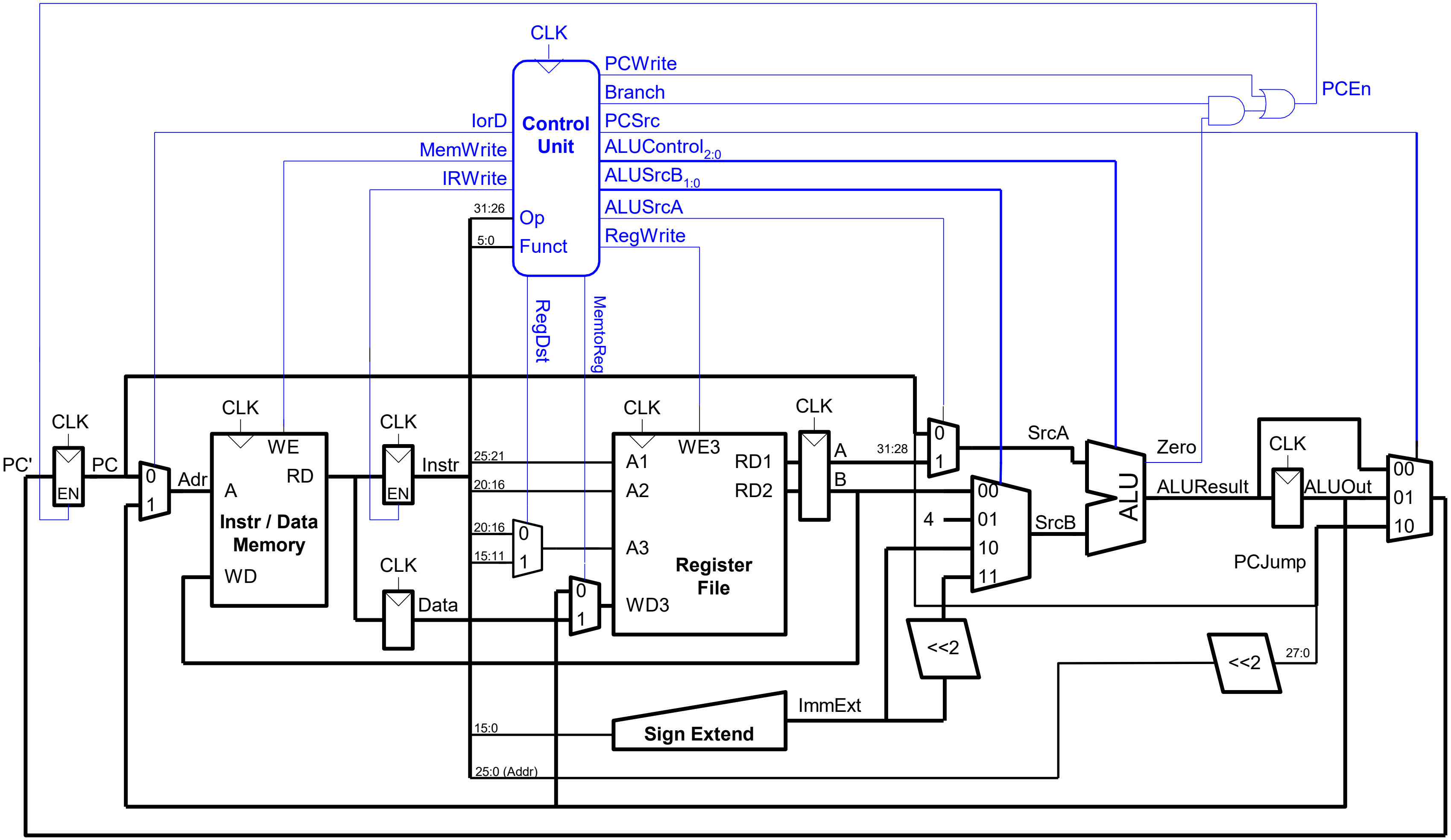
Review: Single-Cycle MIPS FSM

- Single-cycle machine

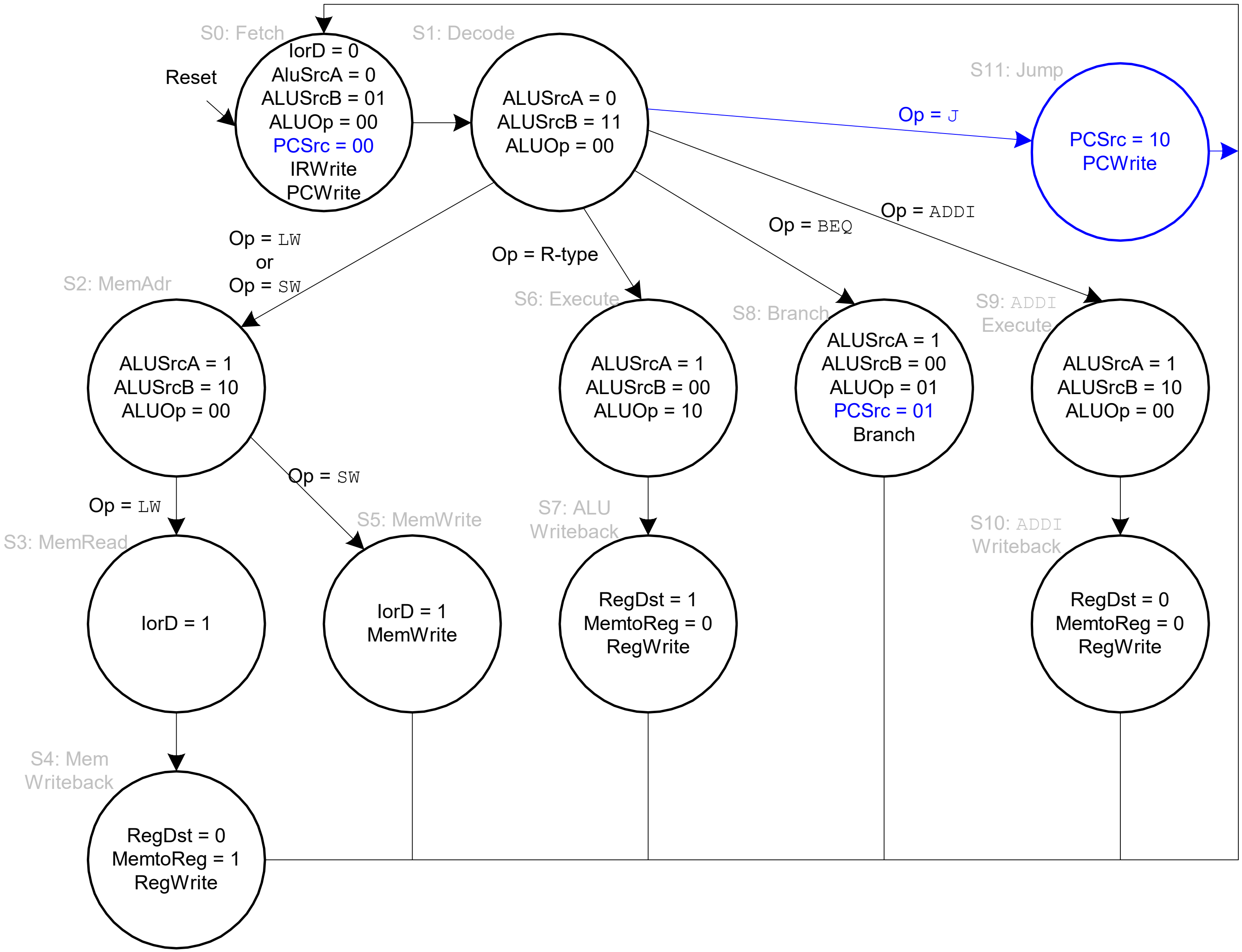


AS: Architectural State

Review: Multi-Cycle MIPS Processor



Review: Multi-Cycle MIPS FSM



What is the shortcoming of this design?

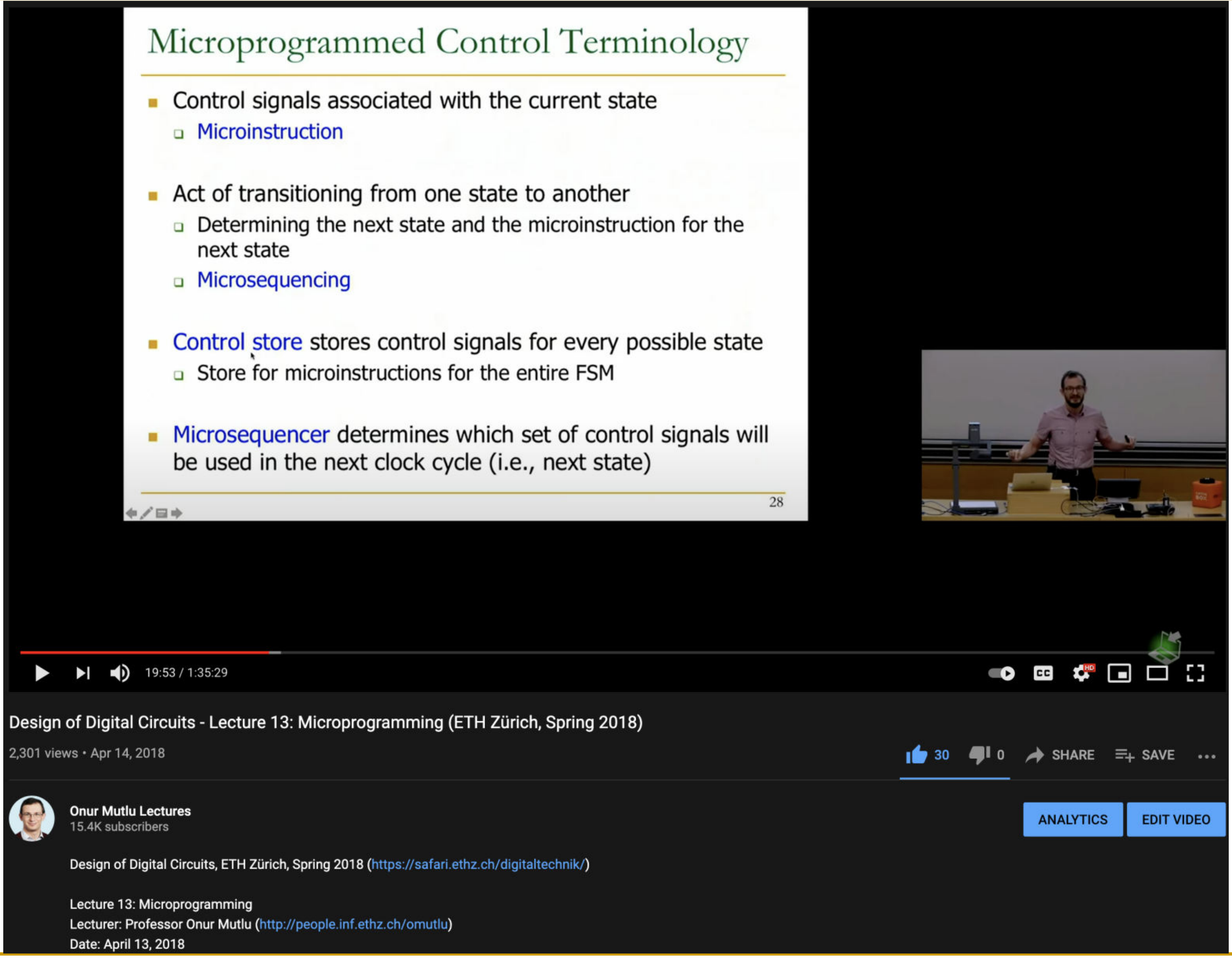
What does this design assume about memory?

What If Memory Takes > One Cycle?

- Stay in the same “memory access” state until memory returns the data
- “Memory Ready?” bit is an input to the control logic that determines the next state

Backup Slides on **Microprogrammed** Multi-Cycle Microarchitectures

These Slides Are Covered in A Past Lecture



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)

28

Design of Digital Circuits - Lecture 13: Microprogramming (ETH Zürich, Spring 2018)

2,301 views • Apr 14, 2018

Onur Mutlu Lectures
15.4K subscribers

Design of Digital Circuits, ETH Zürich, Spring 2018 (https://safari.ethz.ch/digitaltechnik/)

Lecture 13: Microprogramming
Lecturer: Professor Onur Mutlu (http://people.inf.ethz.ch/omutlu)
Date: April 13, 2018

https://www.youtube.com/onurmutlulectures

Lectures on Microprogrammed Designs

- Design of Digital Circuits, Spring 2018, Lecture 13
  - Microprogramming (ETH Zürich, Spring 2018)
  - https://www.youtube.com/watch?v=u4GhShuBP3Y&list=PL5Q2soXY2Zi_QedyPWtRmFUJ2F8DdYP7l&index=13

- Computer Architecture, Spring 2013, Lecture 7
  - Microprogramming (CMU, Spring 2013)
  - https://www.youtube.com/watch?v=_igvSl5h8cs&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=7

Another Example: **Microprogrammed** Multi-Cycle Microarchitecture

An Elegant Multi-Cycle Processor Design

- Maurice Wilkes, "The Best Way to Design an Automatic Calculating Machine," Manchester Univ. Computer Inaugural Conf., 1951.

THE BEST WAY TO DESIGN AN AUTOMATIC
CALCULATING MACHINE

By M. V. Wilkes, M.A., Ph.D., F.R.A.S.

- An elegant implementation:
  - The concept of microcoded/microprogrammed machines

Recall: 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

Microprogrammed Control Terminology

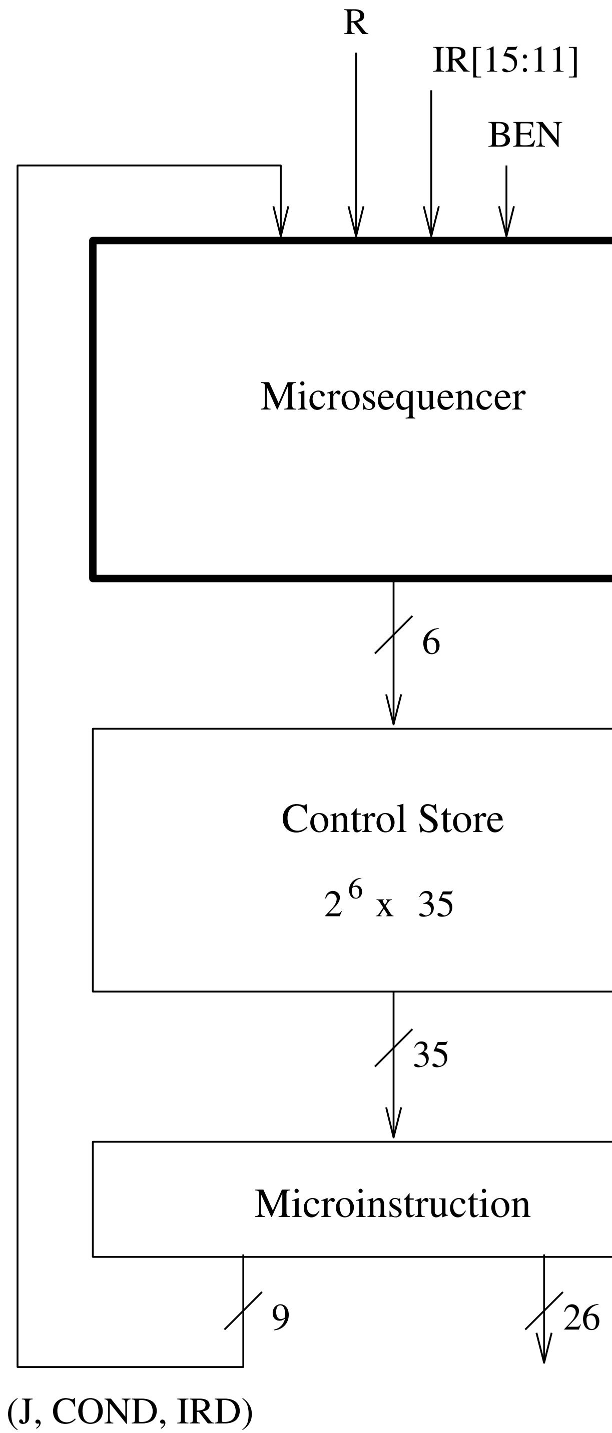
- Control signals associated with the current state
  - Microinstruction

- 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)

Example Control Structure



Simple Design
of the Control Structure

What Happens In A Clock Cycle?

- The control signals (microinstruction) for the current state control two things:
  - Processing in the data path
  - Generation of control signals (microinstruction) for the next cycle
  - *See Supplemental Figure 1 (next-next slide)*

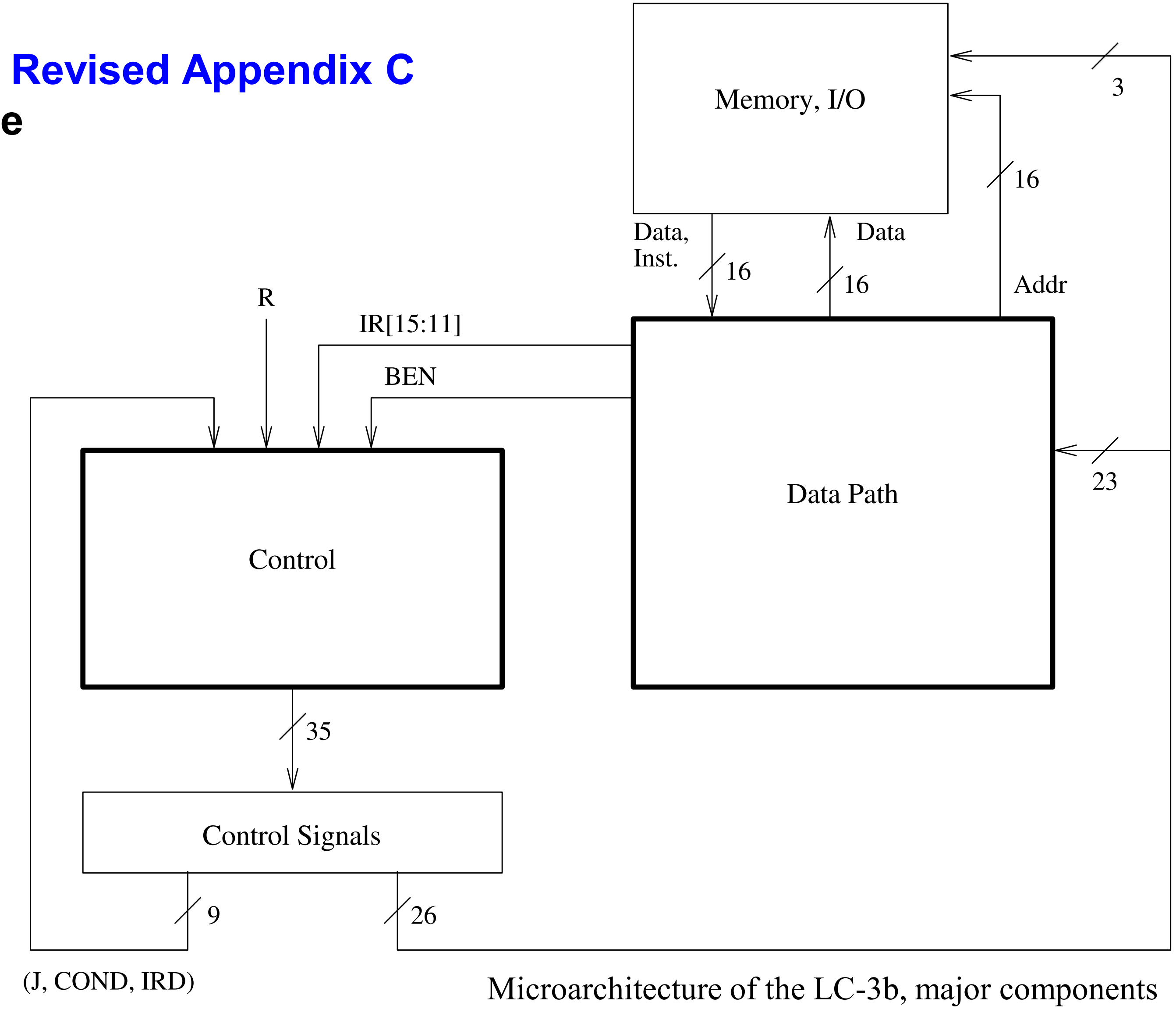
- Datapath and microsequencer operate concurrently

- Question: why not generate control signals for the current cycle in the current cycle?
  - This could lengthen the clock cycle
  - Why could it lengthen the clock cycle?
  - *See Supplemental Figure 2*

Example uProgrammed Control & Datapath

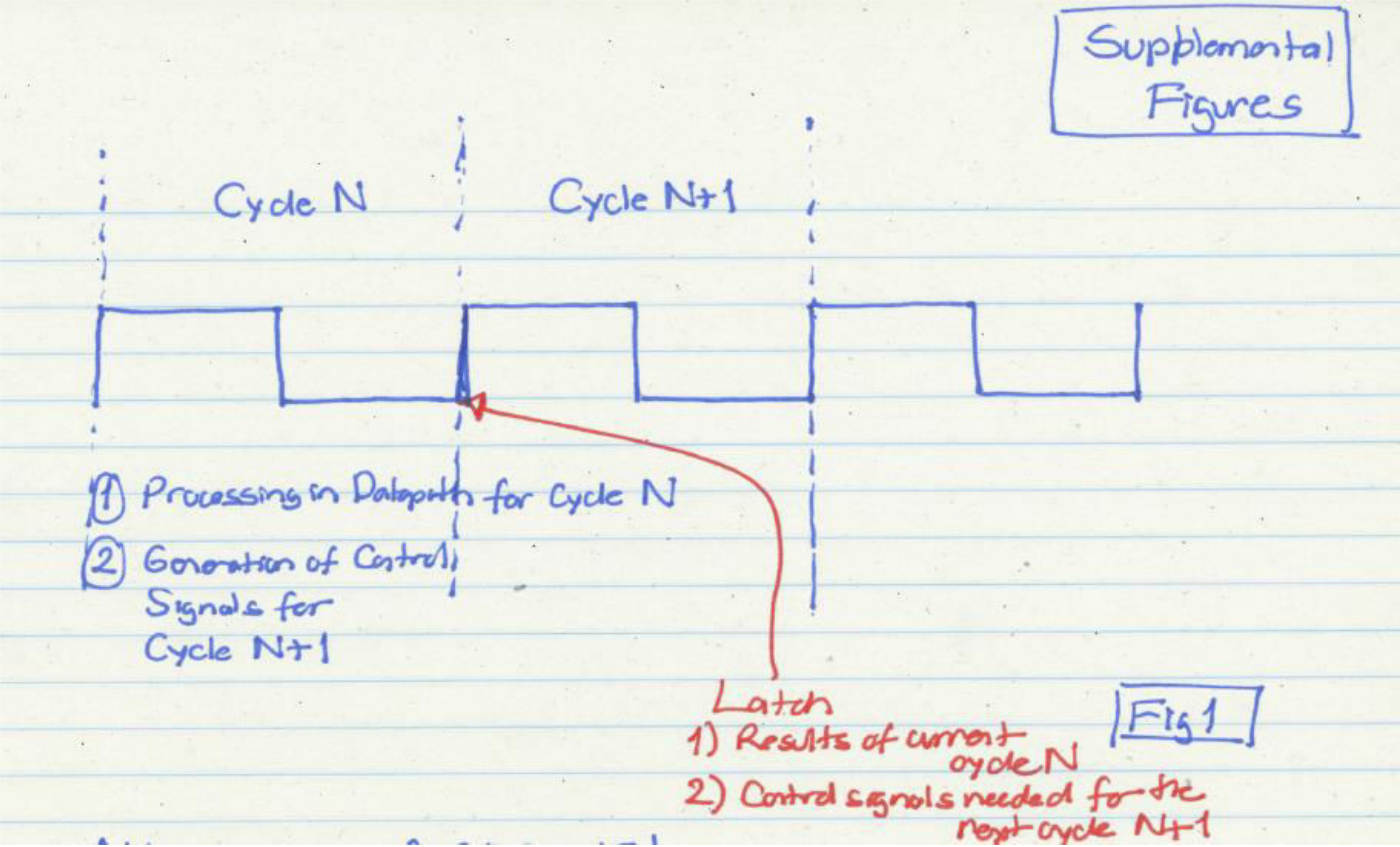
Read P&P Revised Appendix C

On website



Microarchitecture of the LC-3b, major components

A Clock Cycle

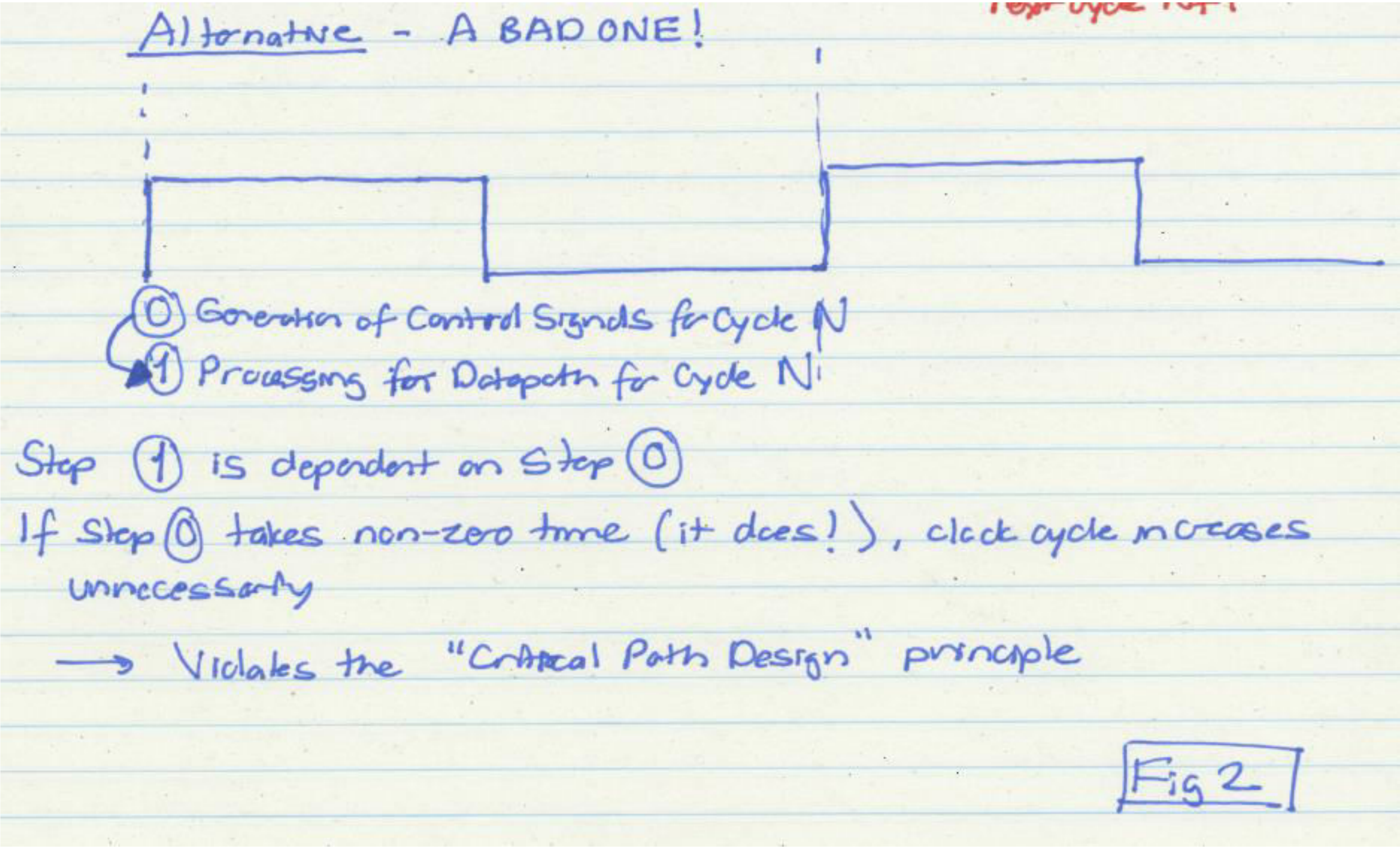


① Processing in Datapath for Cycle N

② Generation of Control Signals for Cycle N+1

Latch
1) Results of current cycle N
2) Control signals needed for the next cycle N+1

A Bad Clock Cycle!



Alternative - A BAD ONE!

⓪ Generation of Control Signals for Cycle N

① Processing for Datapath for Cycle N

Step ① is dependent on Step ⓪

If Step ⓪ takes non-zero time (it does!), clock cycle increases unnecessarily

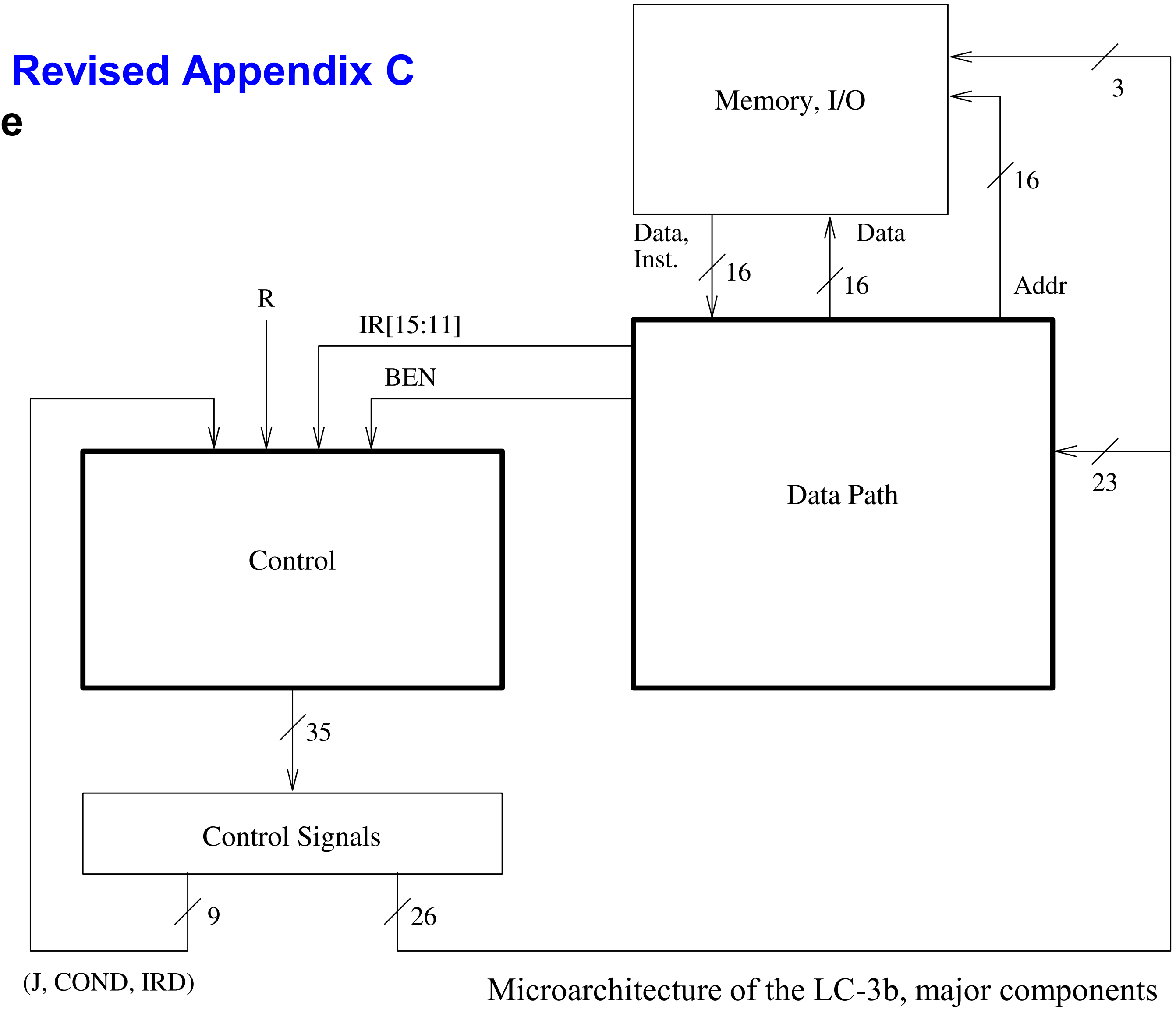
→ Violates the "Critical Path Design" principle

Fig 2

A Simple LC-3b Control and Datapath

Read P&P Revised Appendix C

On website

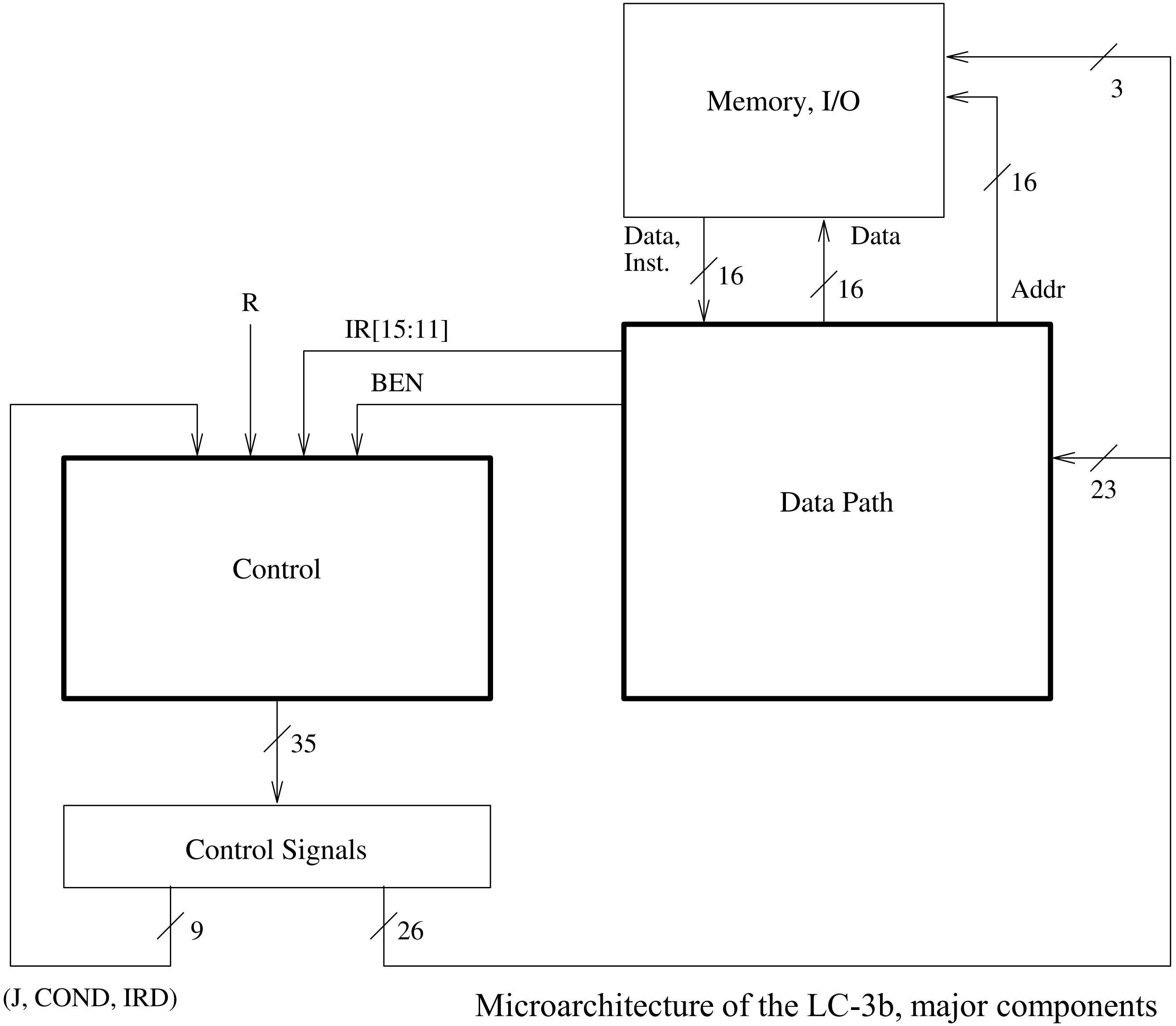


Microarchitecture of the LC-3b, major components

What Determines Next-State Control Signals?

- What is happening in the current clock cycle
  - See the 9 control signals coming from "Control" block
    - What are these for?
- The instruction that is being executed
  - IR[15:11] coming from the Data Path
- Whether the condition of a branch is met, if the instruction being processed is a branch
  - BEN bit coming from the datapath
- Whether the memory operation is completing in the current cycle, if one is in progress
  - R bit coming from memory

A Simple LC-3b Control and Datapath



Microarchitecture of the LC-3b, major components

The State Machine for Multi-Cycle Processing

- The behavior of the LC-3b uarch is completely determined by
  - the 35 control signals and
  - additional 7 bits that go into the control logic from the datapath

- 35 control signals completely describe the state of the control structure

- We can completely describe the behavior of the LC-3b as a state machine, i.e. a directed graph of
  - Nodes (one corresponding to each state)
  - Arcs (showing flow from each state to the next state(s))

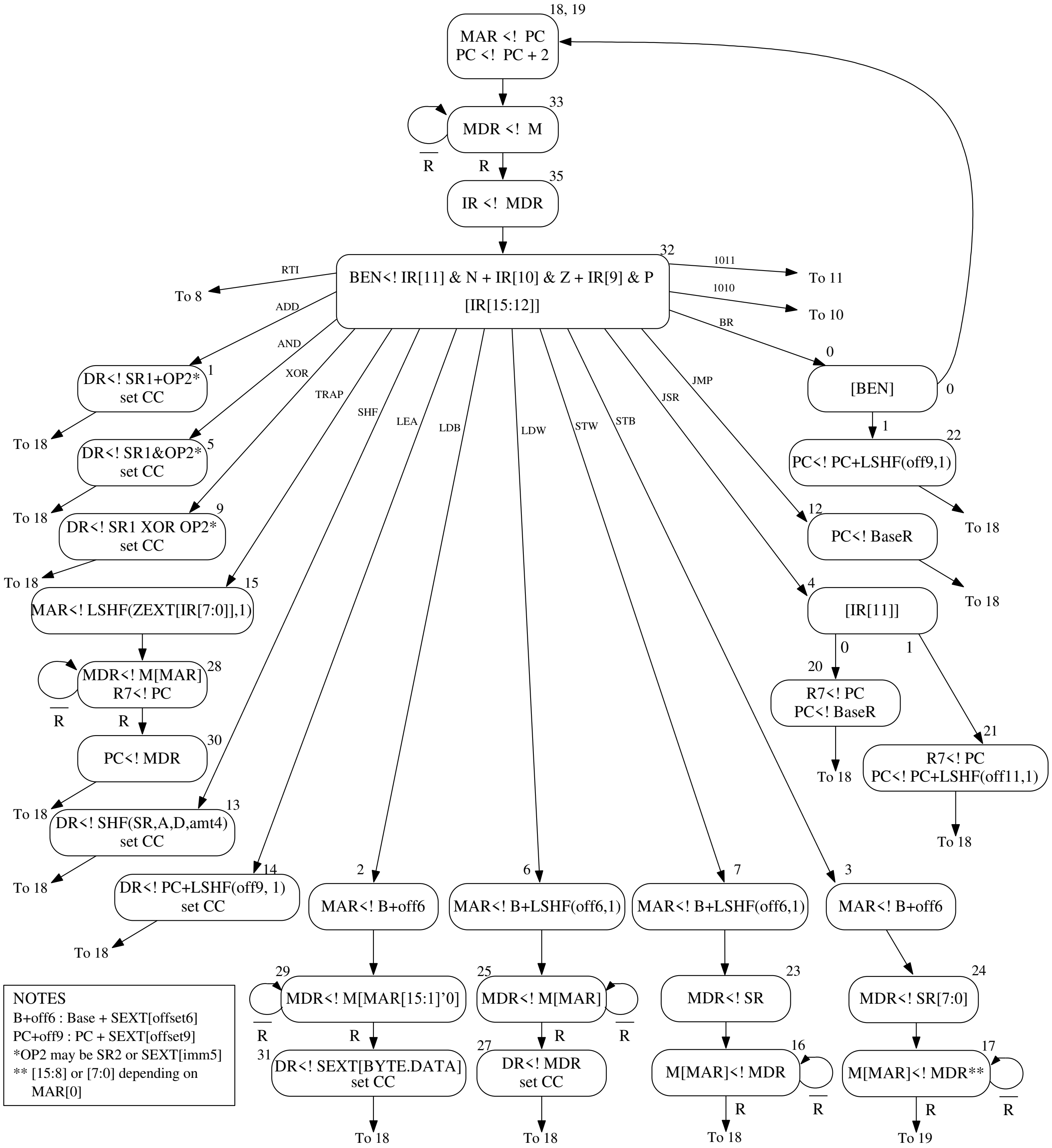
An LC-3b State Machine

- Patt and Patel, Revised Appendix C, Figure C.2

- Each state must be uniquely specified
  - Done by means of *state variables*

- 31 distinct states in this LC-3b state machine
  - Encoded with 6 state variables

- Examples
  - State 18,19 correspond to the beginning of the instruction processing cycle
  - Fetch phase: state 18, 19 → state 33 → state 35
  - Decode phase: state 32



18, 19
MAR <! PC
PC <! PC + 2

33
MDR <! M
$\overline{R}$
R

35
IR <! MDR

32
BEN<! IR[11] & N + IR[10] & Z + IR[9] & P
[IR[15:12]]

RTI
To 8
1011
To 11
1010
To 10
BR
ADD
AND
XOR
TRAP
SHF
LEA
LDB
LDW
STW
STB
JSR
JMP

0
[BEN]
0
1

22
PC<! PC+LSHF(off9,1)
To 18

12
PC<! BaseR
To 18

4
[IR[11]]
0
1

20
R7<! PC
PC<! BaseR
To 18

21
R7<! PC
PC<! PC+LSHF(off11,1)
To 18

1
DR<! SR1+OP2*
set CC
To 18

5
DR<! SR1&OP2*
set CC
To 18

9
DR<! SR1 XOR OP2*
set CC
To 18

15
MAR<! LSHF(ZEXT[IR[7:0]],1)

28
MDR<! M[MAR]
R7<! PC
$\overline{R}$
R

30
PC<! MDR
To 18

13
DR<! SHF(SR,A,D,amt4)
set CC
To 18

14
DR<! PC+LSHF(off9, 1)
set CC
To 18

2
MAR<! B+off6

29
MDR<! M[MAR[15:1]'0]
$\overline{R}$
R

31
DR<! SEXT[BYTE.DATA]
set CC
To 18

6
MAR<! B+LSHF(off6,1)

25
MDR<! M[MAR]
R
$\overline{R}$

27
DR<! MDR
set CC
To 18

7
MAR<! B+LSHF(off6,1)

23
MDR<! SR

16
M[MAR]<! MDR
R
$\overline{R}$
To 18

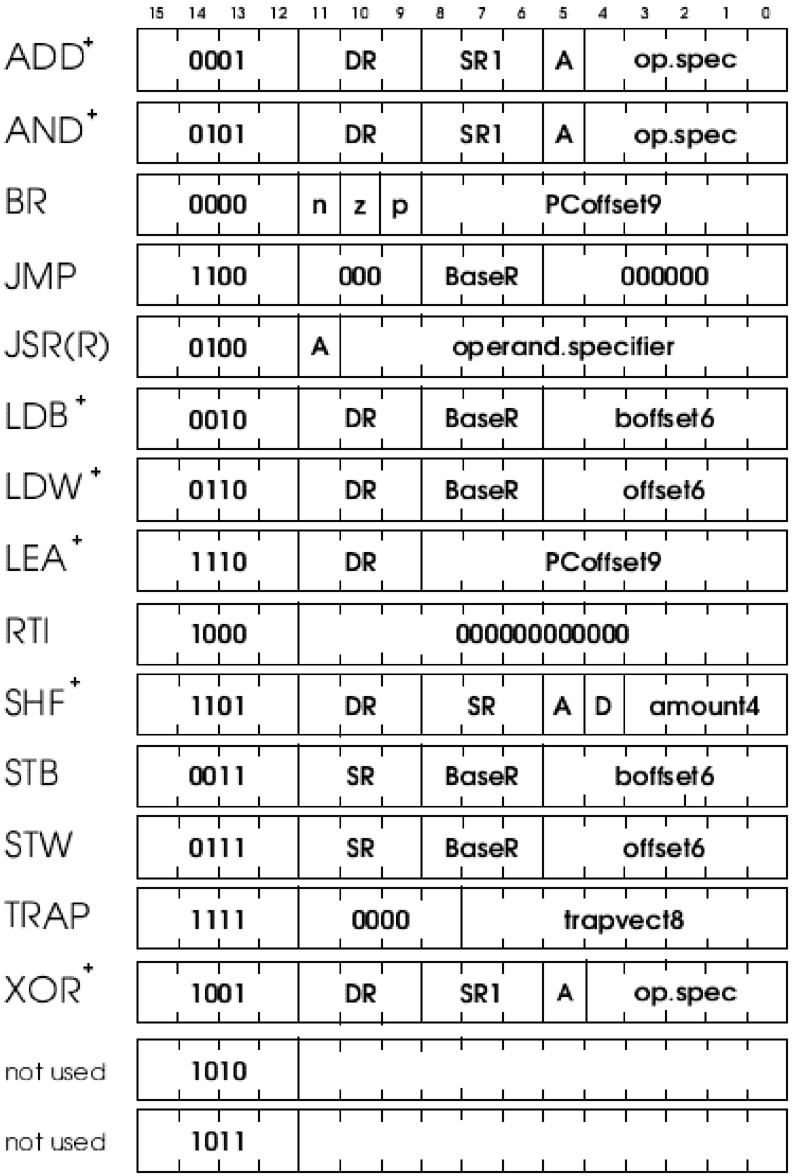
3
MAR<! B+off6

24
MDR<! SR[7:0]

17
M[MAR]<! MDR**
R
$\overline{R}$
To 19

NOTES
B+off6 : Base + SEXT[offset6]
PC+off9 : PC + SEXT[offset9]
*OP2 may be SR2 or SEXT[imm5]
** [15:8] or [7:0] depending on
MAR[0]

The FSM Implements the LC-3b ISA



	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD^{+}	0001				DR			SR1			A	op.spec				
AND^{+}	0101				DR			SR1			A	op.spec				
BR	0000				n	z	p	PCoffset9								
JMP	1100				000			BaseR			000000					
JSR(R)	0100				A	operand.specifier										
LDB^{+}	0010				DR			BaseR			boffset6					
LDW^{+}	0110				DR			BaseR			offset6					
LEA^{+}	1110				DR			PCoffset9								
RTI	1000				000000000000											
SHF^{+}	1101				DR			SR			A	D	amount4			
STB	0011				SR			BaseR			boffset6					
STW	0111				SR			BaseR			offset6					
TRAP	1111				0000				trapvect8							
XOR^{+}	1001				DR			SR1			A	op.spec				
not used	1010															
not used	1011															

- P&P Appendix A (revised):
  - https://safari.ethz.ch/digitaltechnik/spring2018/lib/exe/fetch.php?media=pp-appendixa.pdf

LC-3b State Machine: Some Questions

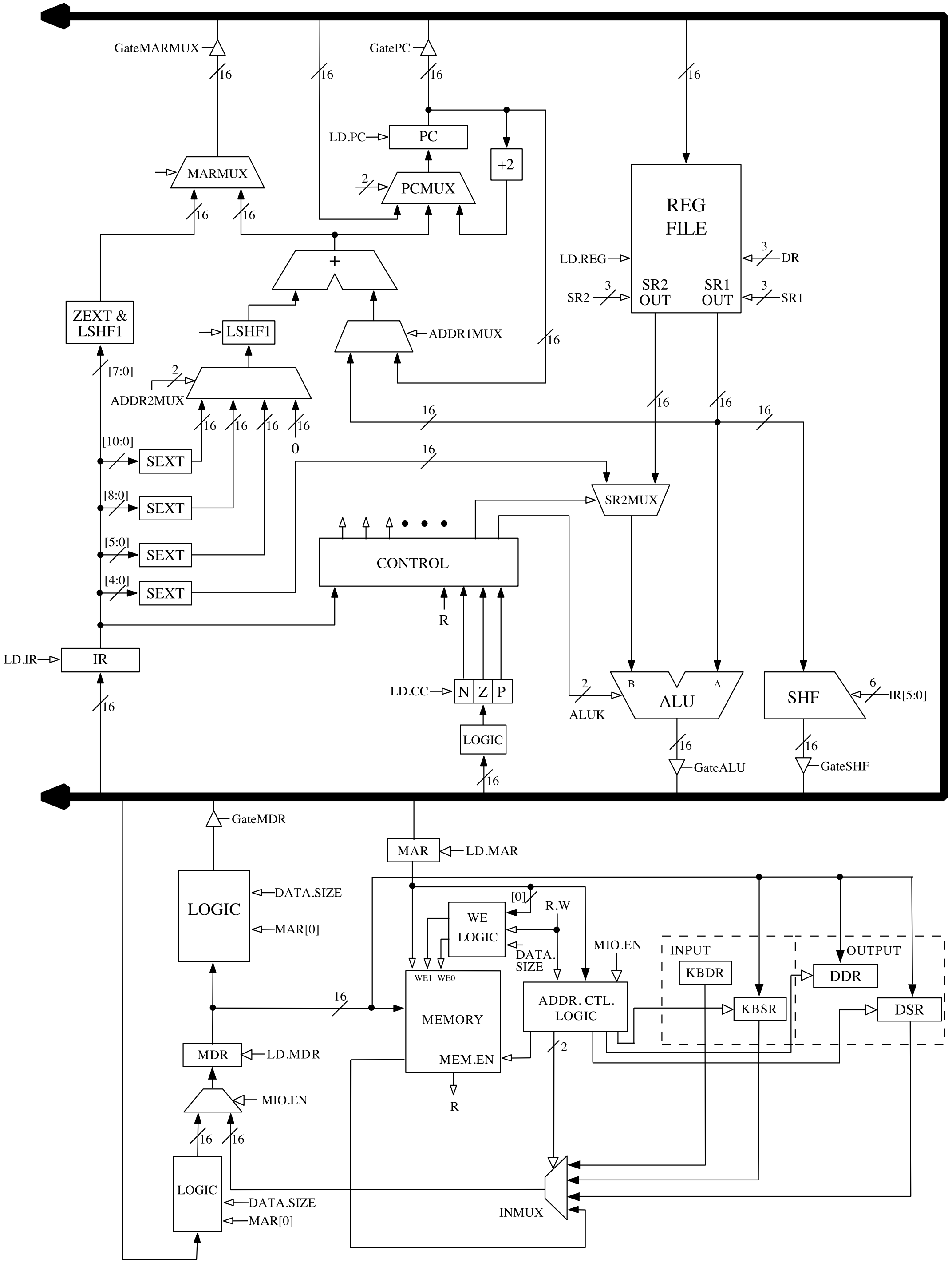
- How many cycles does the fastest instruction take?
- How many cycles does the slowest instruction take?
- Why does the BR take as long as it takes in the FSM?
- What determines the clock cycle time?

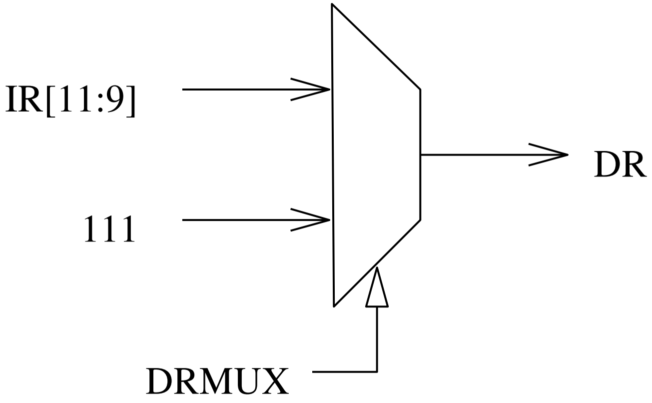
LC-3b Datapath

- Patt and Patel, Revised Appendix C, Figure C.3

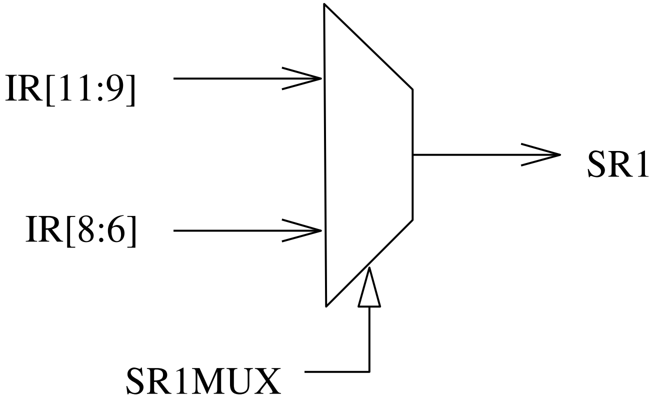
- Single-bus datapath design
  - At any point only one value can be "gated" on the bus (i.e., can be driving the bus)
  - Advantage: Low hardware cost: one bus
  - Disadvantage: Reduced concurrency – if instruction needs the bus twice for two different things, these need to happen in different states

- Control signals (26 of them) determine what happens in the datapath in one clock cycle
  - Patt and Patel, Revised Appendix C, Table C.1


GateMARMUX
GatePC
16
LD.PC
PC
+2
MARMUX
2
PCMUX
REG
FILE
LD.REG
3
DR
SR2
SR2
OUT
SR1
OUT
SR1
+
ZEXT &
LSHF1
LSHF1
ADDR1MUX
[7:0]
2
ADDR2MUX
0
[10:0]
SEXT
[8:0]
SEXT
[5:0]
SEXT
[4:0]
SEXT
SR2MUX
CONTROL
R
LD.IR
IR
LD.CC
N
Z
P
LOGIC
ALUK
B
A
ALU
SHF
6
IR[5:0]
GateALU
GateSHF
GateMDR
MAR
LD.MAR
LOGIC
DATA.SIZE
MAR[0]
[0]
R.W
WE
LOGIC
DATA.
SIZE
MIO.EN
WE1 WE0
MEMORY
MEM.EN
ADDR. CTL.
LOGIC
INPUT
KBDR
KBSR
OUTPUT
DDR
DSR
MDR
LD.MDR
MIO.EN
LOGIC
DATA.SIZE
MAR[0]
INMUX

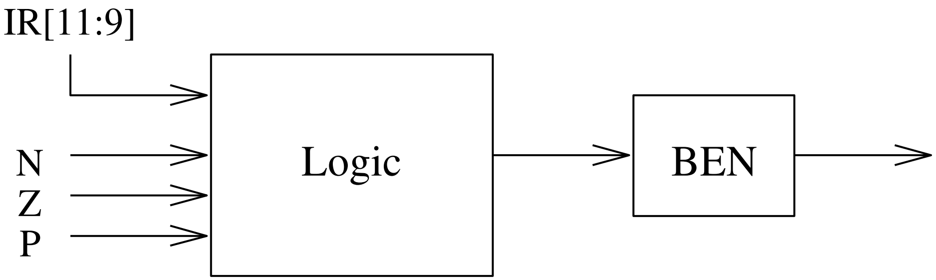


(a)

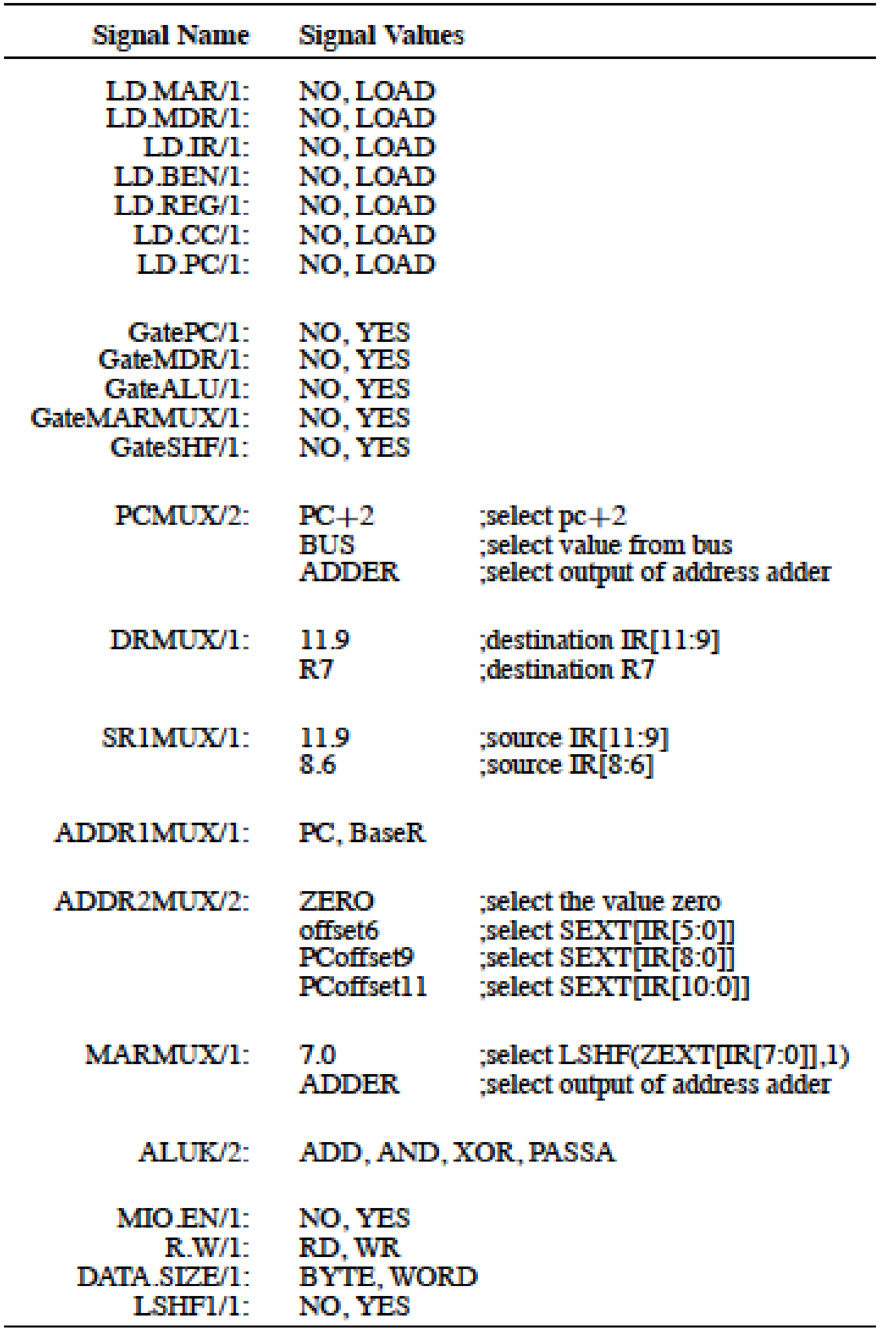


(b)

Remember the MIPS datapath



(c)



Signal Name	Signal Values	
LD.MAR/1:	NO, LOAD	
LD.MDR/1:	NO, LOAD	
LD.IR/1:	NO, LOAD	
LD.BEN/1:	NO, LOAD	
LD.REG/1:	NO, LOAD	
LD.CC/1:	NO, LOAD	
LD.PC/1:	NO, LOAD	
GatePC/1:	NO, YES	
GateMDR/1:	NO, YES	
GateALU/1:	NO, YES	
GateMARMUX/1:	NO, YES	
GateSHF/1:	NO, YES	
PCMUX/2:	PC+2	;select pc+2
	BUS	;select value from bus
	ADDER	;select output of address adder
DRMUX/1:	11.9	;destination IR[11:9]
	R7	;destination R7
SR1MUX/1:	11.9	;source IR[11:9]
	8.6	;source IR[8:6]
ADDR1MUX/1:	PC, BaseR	
ADDR2MUX/2:	ZERO	;select the value zero
	offset6	;select SEXT[IR[5:0]]
	PCoffset9	;select SEXT[IR[8:0]]
	PCoffset11	;select SEXT[IR[10:0]]
MARMUX/1:	7.0	;select LSHF(ZEXT[IR[7:0]],1)
	ADDER	;select output of address adder
ALUK/2:	ADD, AND, XOR, PASSA	
MIO.EN/1:	NO, YES	
R.W/1:	RD, WR	
DATA.SIZE/1:	BYTE, WORD	
LSHF1/1:	NO, YES	

Table C.1: Data path control signals

LC-3b Datapath: Some Questions

- How does instruction fetch happen in this datapath according to the state machine?

- What is the difference between gating and loading?
  - Gating: Enable/disable an input to be connected to the bus
    - Combinational: during a clock cycle
  - Loading: Enable/disable an input to be written to a register
    - Sequential: e.g., at a clock edge (assume at the end of cycle)

- Is this the smallest hardware you can design?

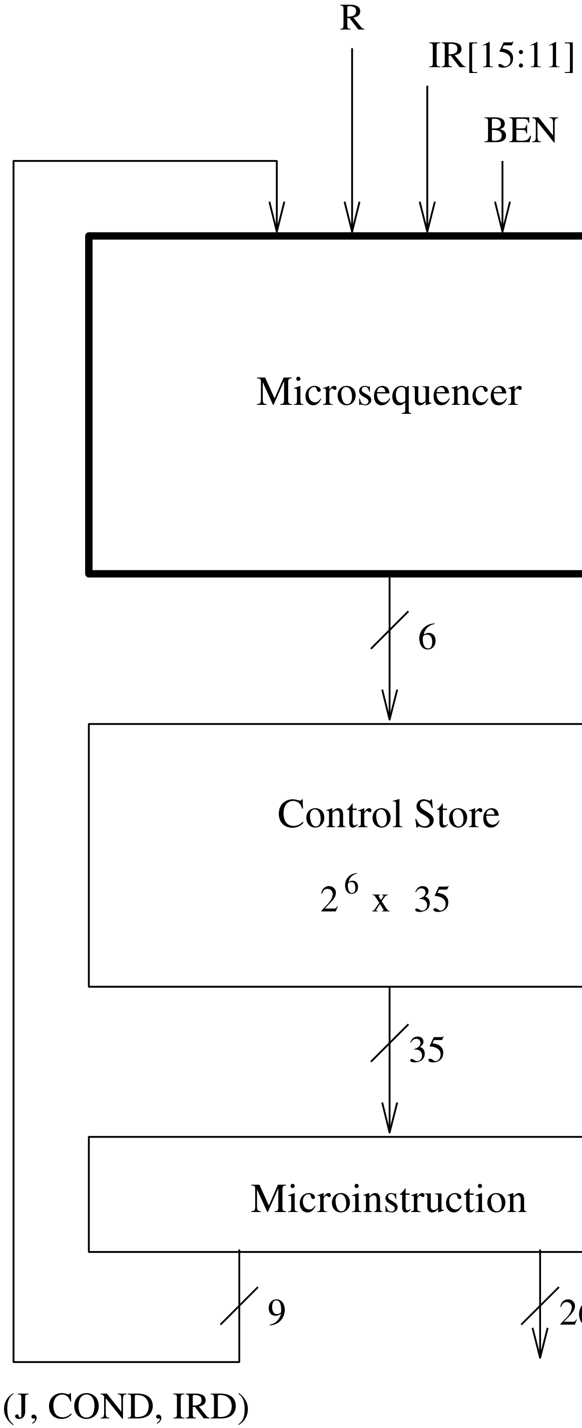
LC-3b Microprogrammed Control Structure

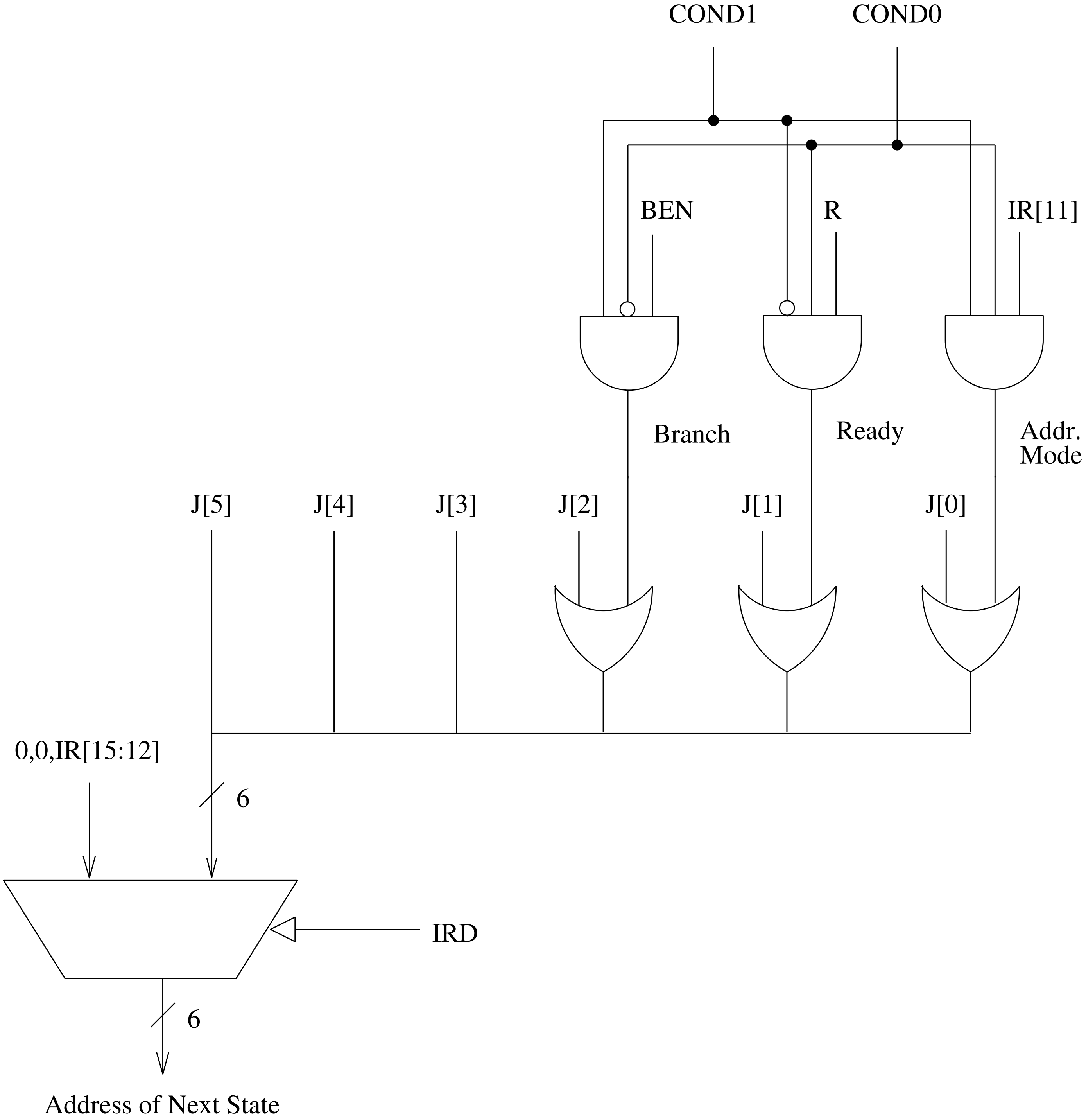
- Patt and Patel, Appendix C, Figure C.4

- 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
  - Remember each state corresponds to one microinstruction
- Microsequencer determines the address of the next microinstruction (i.e., next state)

Simple Design of the Control Structure

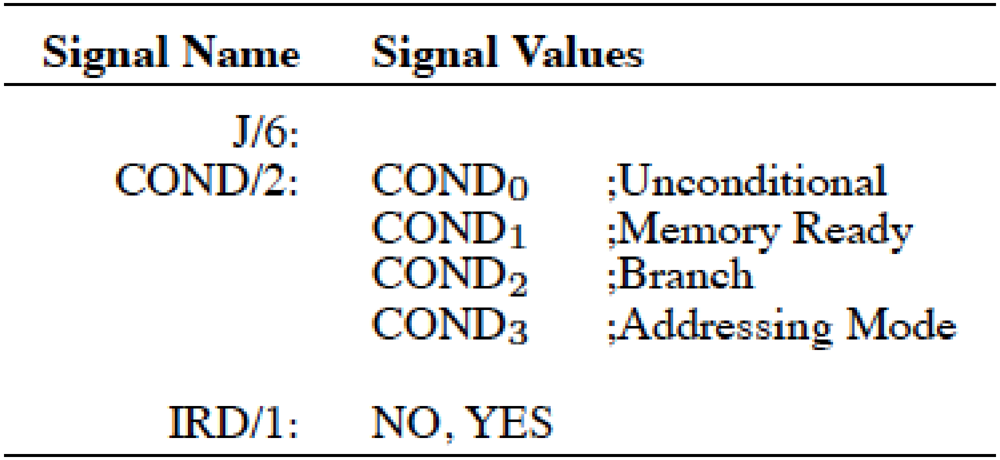



COND1
COND0
BEN
R
IR[11]
Branch
Ready
Addr.
Mode
J[5]
J[4]
J[3]
J[2]
J[1]
J[0]
0,0,IR[15:12]
6
IRD
6
Address of Next State

IRD	Cond	J	LD.MAR	LD.MDR	LD.IR	LD.BEN	LD.REG	LD.CC	LD.PC	GatePC	GateMDR	GateALU	GateMARMUX	GateSHF	PCMUX	DRMUX	SR1MUX	ADDR1MUX	ADDR2MUX	MARMUX	ALUK	MIO.EN	R.W	DATA.SIZE	LSHF1	
																										000000 (State 0)
																										000001 (State 1)
																										000010 (State 2)
																										000011 (State 3)
																										000100 (State 4)
																										000101 (State 5)
																										000110 (State 6)
																										000111 (State 7)
																										001000 (State 8)
																										001001 (State 9)
																										001010 (State 10)
																										001011 (State 11)
																										001100 (State 12)
																										001101 (State 13)
																										001110 (State 14)
																										001111 (State 15)
																										010000 (State 16)
																										010001 (State 17)
																										010010 (State 18)
																										010011 (State 19)
																										010100 (State 20)
																										010101 (State 21)
																										010110 (State 22)
																										010111 (State 23)
																										011000 (State 24)
																										011001 (State 25)
																										011010 (State 26)
																										011011 (State 27)
																										011100 (State 28)
																										011101 (State 29)
																										011110 (State 30)
																										011111 (State 31)
																										100000 (State 32)
																										100001 (State 33)
																										100010 (State 34)
																										100011 (State 35)
																										100100 (State 36)
																										100101 (State 37)
																										100110 (State 38)
																										100111 (State 39)
																										101000 (State 40)
																										101001 (State 41)
																										101010 (State 42)
																										101011 (State 43)
																										101100 (State 44)
																										101101 (State 45)
																										101110 (State 46)
																										101111 (State 47)
																										110000 (State 48)
																										110001 (State 49)
																										110010 (State 50)
																										110011 (State 51)
																										110100 (State 52)
																										110101 (State 53)
																										110110 (State 54)
																										110111 (State 55)
																										111000 (State 56)
																										111001 (State 57)
																										111010 (State 58)
																										111011 (State 59)
																										111100 (State 60)
																										111101 (State 61)
																										111110 (State 62)
																										111111 (State 63)

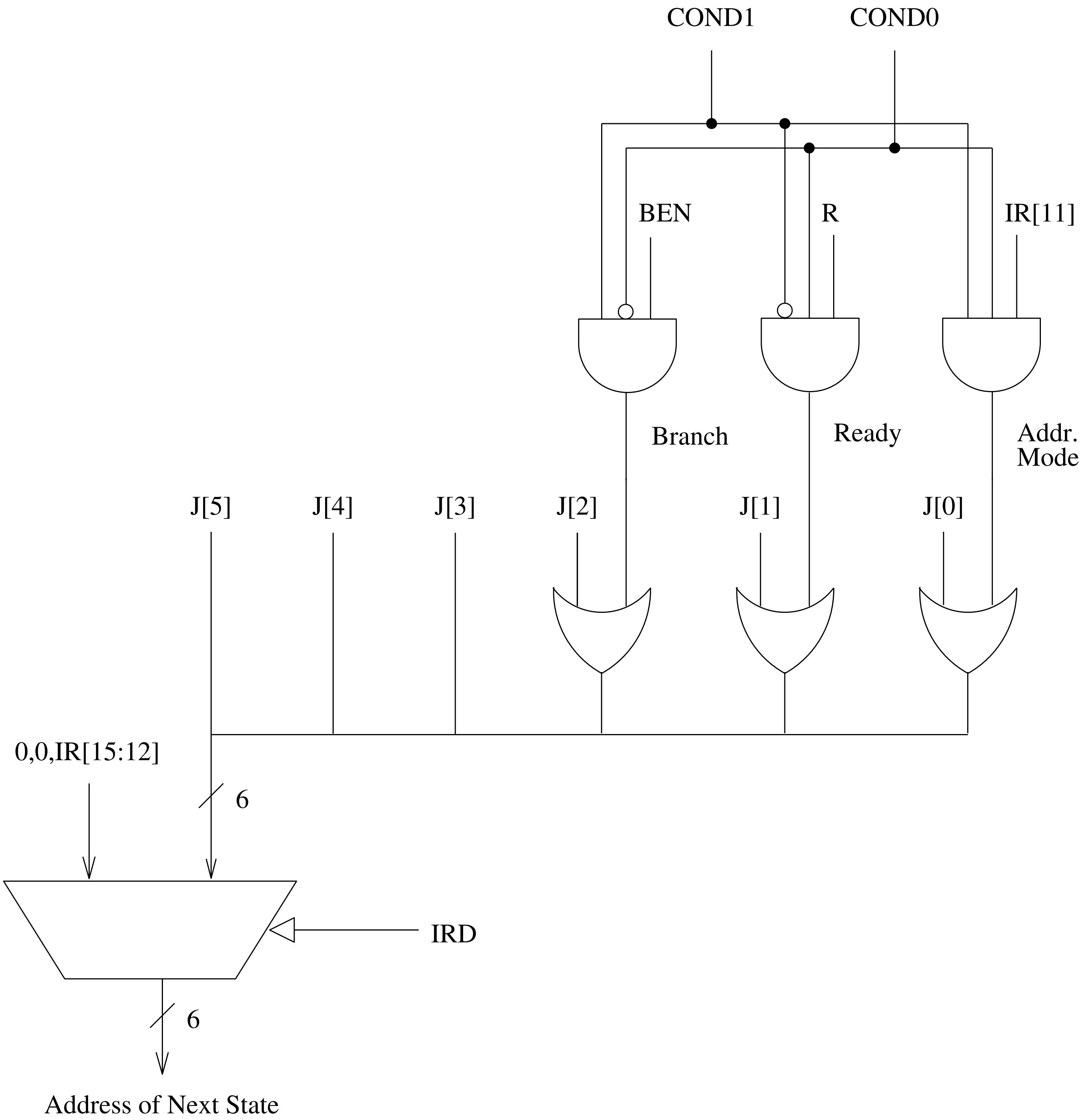
LC-3b Microsequencer

- Patt and Patel, Appendix C, Figure C.5
- The purpose of the microsequencer is to determine the address of the next microinstruction (i.e., next state)
  - Next state could be conditional or unconditional
- Next state address depends on 9 control signals (plus 7 data signals)



Signal Name	Signal Values	
J/6:		
COND/2:	$COND_0$	;Unconditional
	$COND_1$	;Memory Ready
	$COND_2$	;Branch
	$COND_3$	;Addressing Mode
IRD/1:	NO, YES	

Table C.2: Microsequencer control signals


COND1
COND0
BEN
R
IR[11]
Branch
Ready
Addr.
Mode
J[5]
J[4]
J[3]
J[2]
J[1]
J[0]
0,0,IR[15:12]
6
IRD
6
Address of Next State

The Microsequencer: Some Questions

- When is the IRD signal asserted?

- What happens if an illegal instruction is decoded?

- What are condition (COND) bits for?

- How is variable latency memory handled?

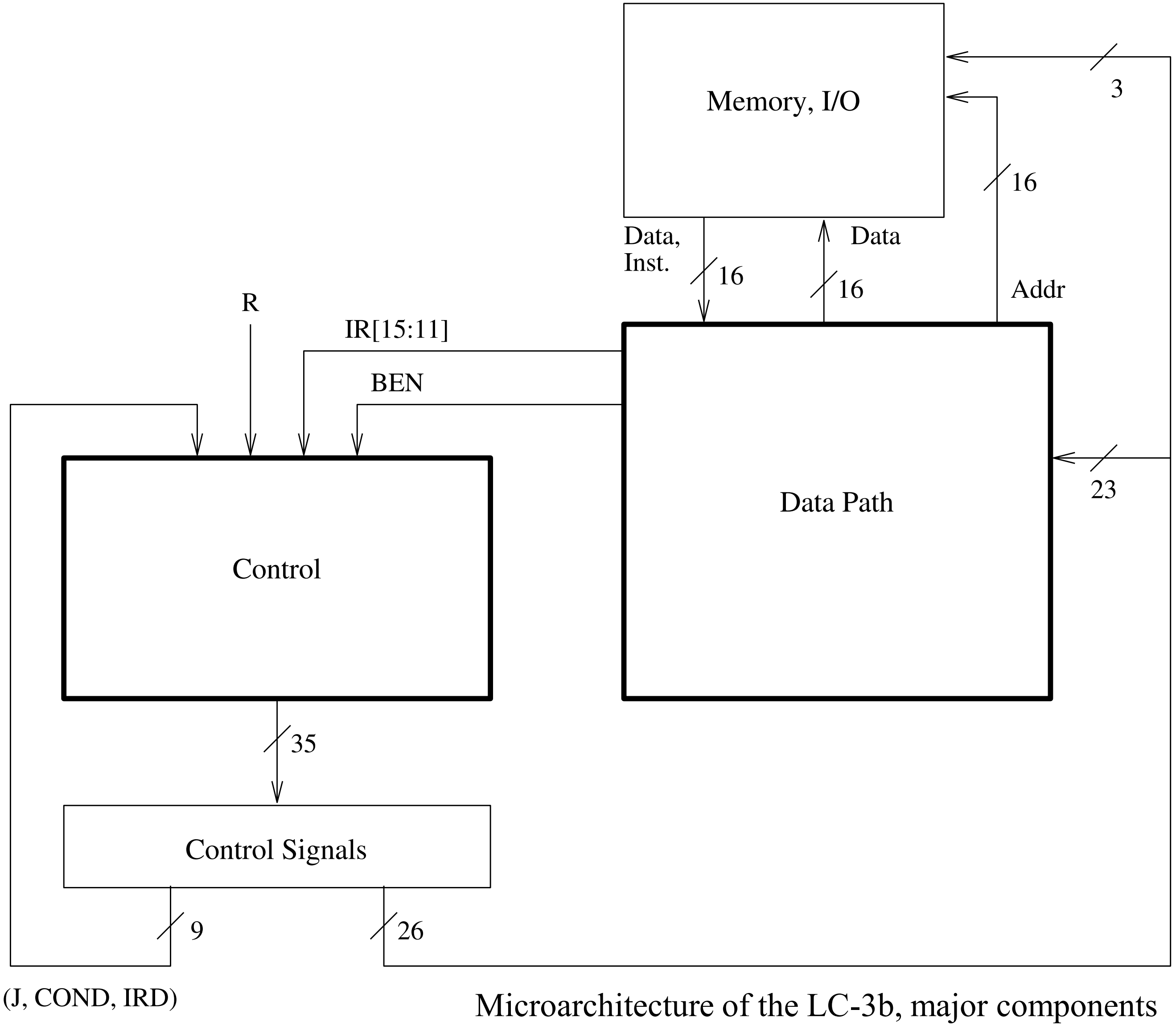
- How do you do the state encoding?
  - Minimize number of state variables (~ control store size)
  - Start with the 16-way branch
  - Then determine constraint tables and states dependent on COND

An Exercise in Microprogramming

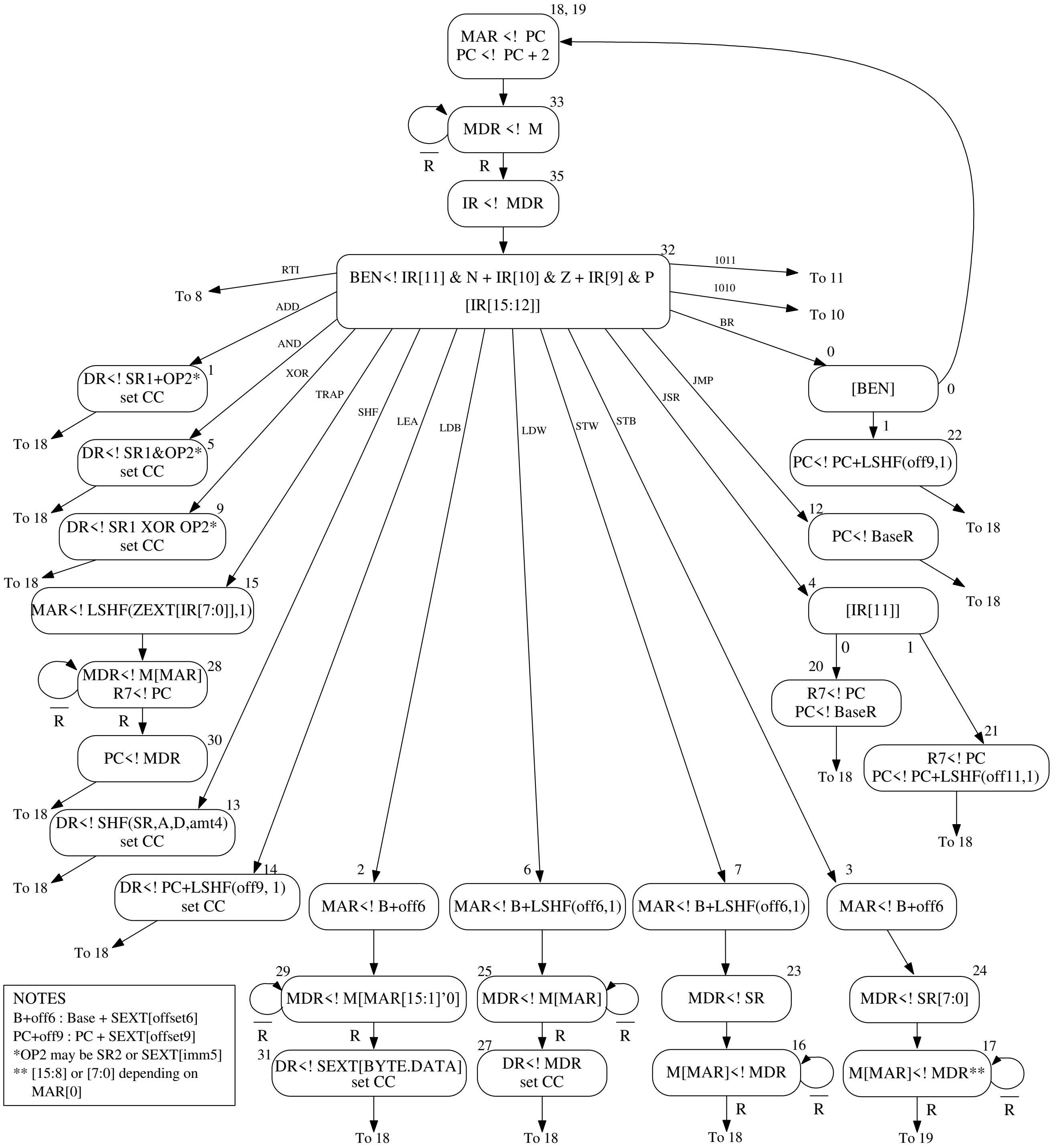
Handouts

- 7 pages of Microprogrammed LC-3b design
- https://safari.ethz.ch/digitaltechnik/spring2018/lib/exe/fetch.php?media=lc3b-figures.pdf

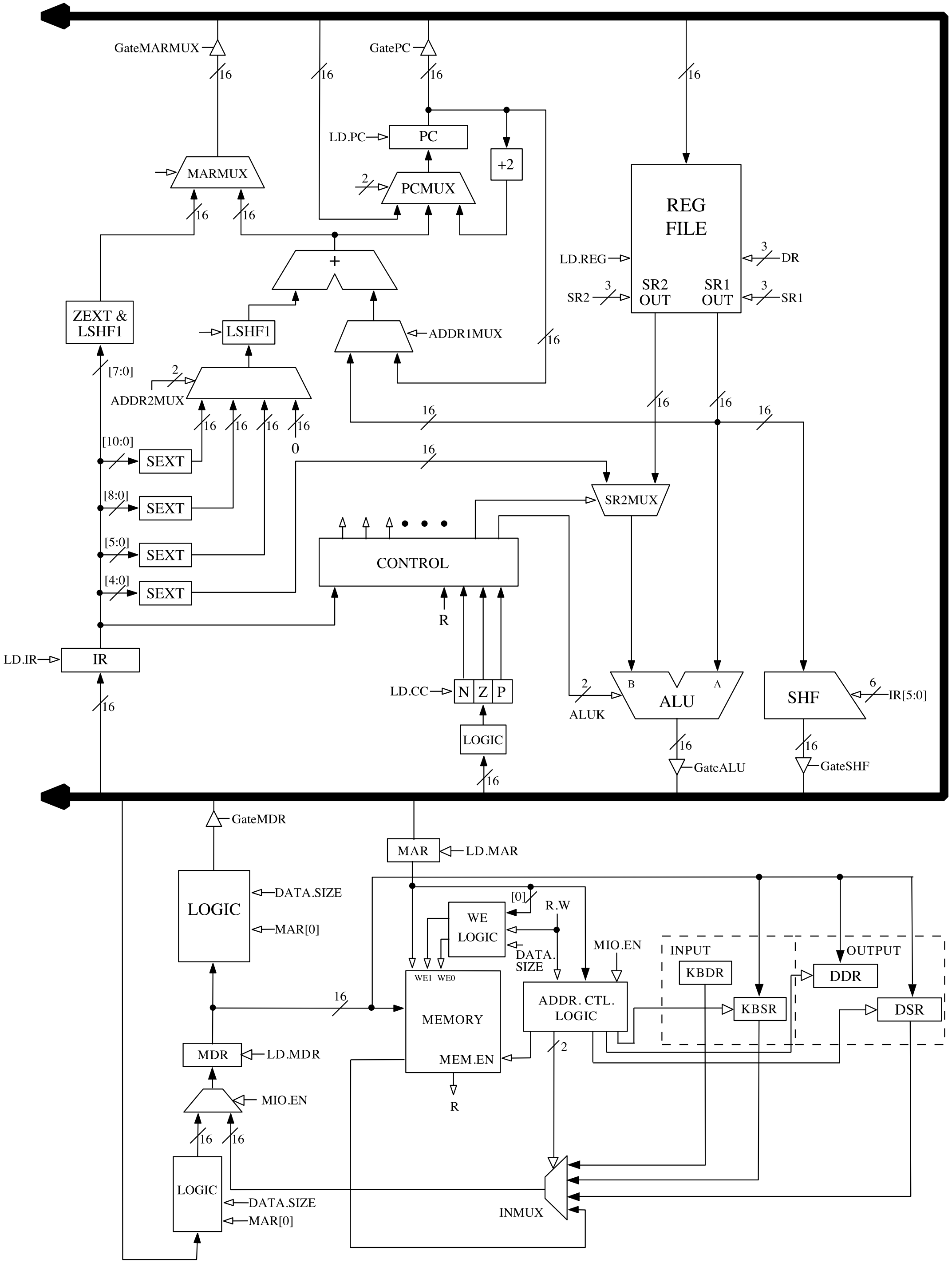
A Simple LC-3b Control and Datapath



Microarchitecture of the LC-3b, major components

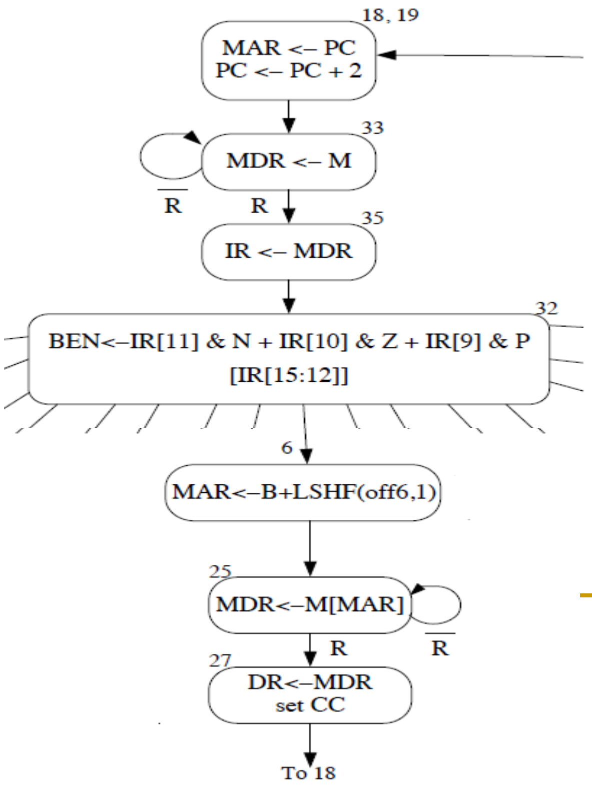


NOTES
B+off6 : Base + SEXT[offset6]
PC+off9 : PC + SEXT[offset9]
*OP2 may be SR2 or SEXT[imm5]
** [15:8] or [7:0] depending on
MAR[0]

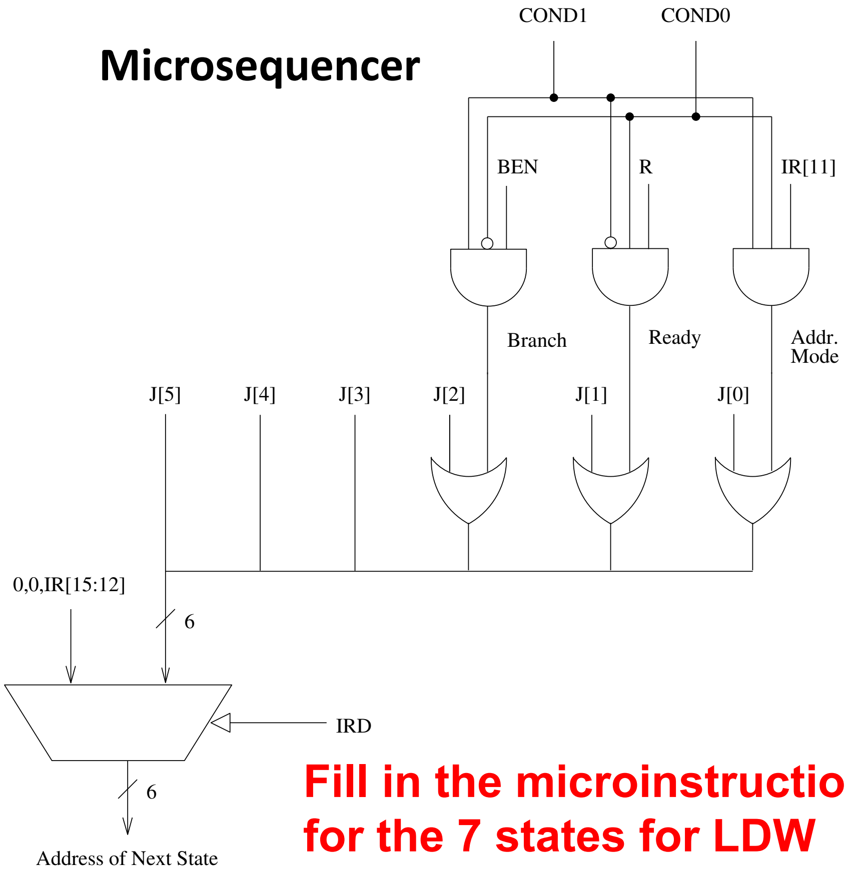


A Simple Datapath Can Become Very Powerful

State Machine for LDW

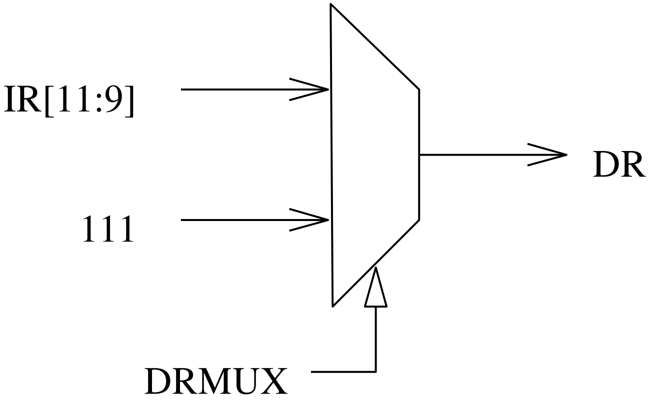


Microsequencer

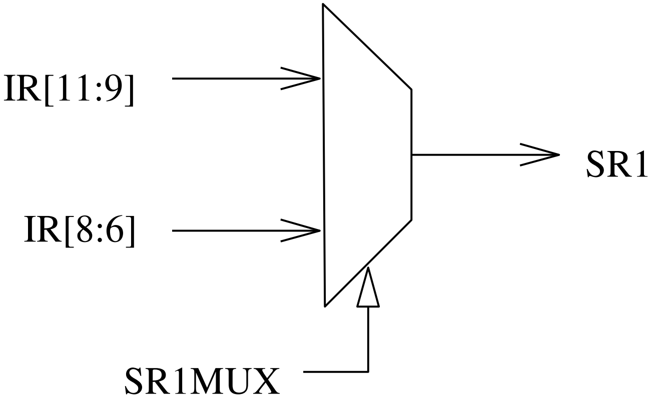


Fill in the microinstructions for the 7 states for LDW

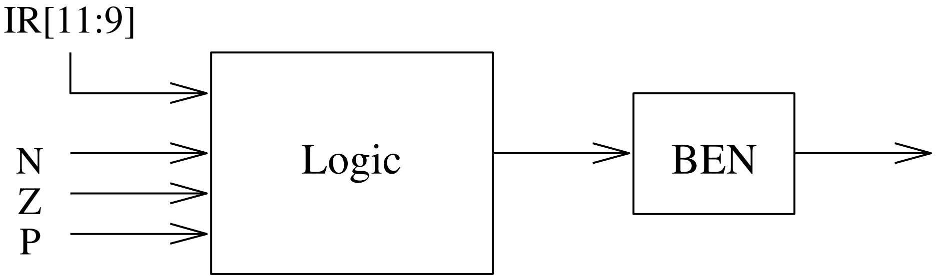
	IRD	Cond	J	LD.MAR	LD.MDR	LD.IR	LD.BEN	LD.REG	LD.CC	LD.PC	GatePC	GateMDR	GateALU	GateMARMUX	GateSHF	PCMUX	DRMUX	SR1MUX	ADDR1MUX	ADDR2MUX	MARMUX	ALUK	MIO.EN	R.W	DATA.SIZE	LSHF1
State 18 (010010)																										
State 33 (100001)																										
State 35 (100011)																										
State 32 (100000)																										
State 6 (000110)																										
State 25 (011001)																										
State 27 (011011)																										



(a)



(b)

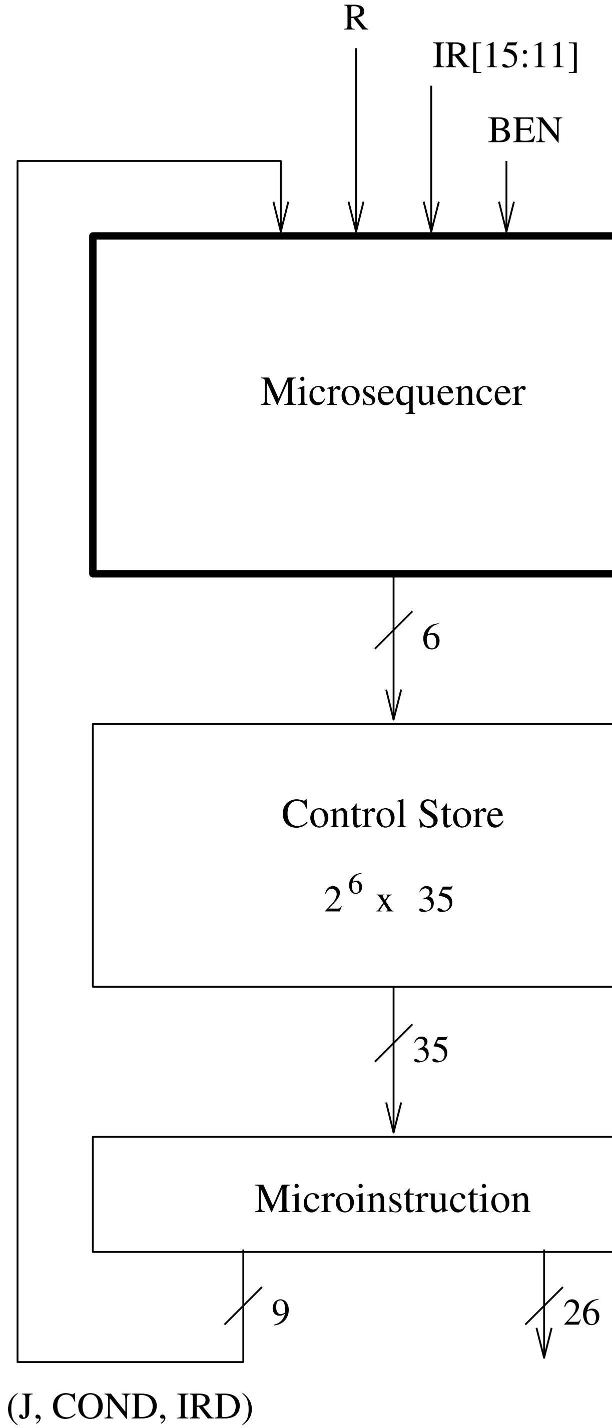


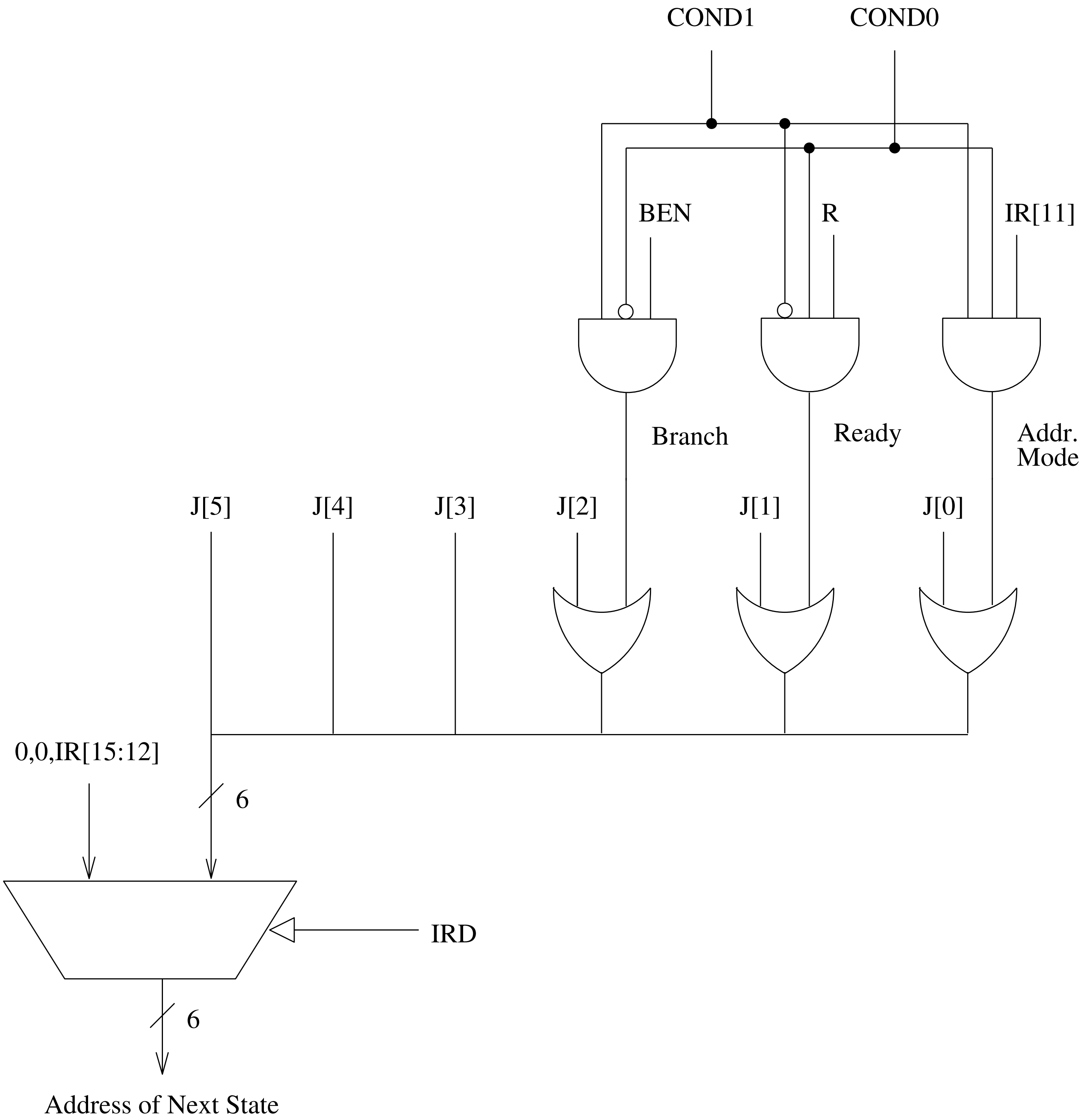
(c)

Signal Name	Signal Values	
LD.MAR/1:	NO, LOAD	
LD.MDR/1:	NO, LOAD	
LD.IR/1:	NO, LOAD	
LD.BEN/1:	NO, LOAD	
LD.REG/1:	NO, LOAD	
LD.CC/1:	NO, LOAD	
LD.PC/1:	NO, LOAD	
GatePC/1:	NO, YES	
GateMDR/1:	NO, YES	
GateALU/1:	NO, YES	
GateMARMUX/1:	NO, YES	
GateSHF/1:	NO, YES	
PCMUX/2:	PC+2	;select pc+2
	BUS	;select value from bus
	ADDER	;select output of address adder
DRMUX/1:	11.9	;destination IR[11:9]
	R7	;destination R7
SR1MUX/1:	11.9	;source IR[11:9]
	8.6	;source IR[8:6]
ADDR1MUX/1:	PC, BaseR	
ADDR2MUX/2:	ZERO	;select the value zero
	offset6	;select SEXT[IR[5:0]]
	PCoffset9	;select SEXT[IR[8:0]]
	PCoffset11	;select SEXT[IR[10:0]]
MARMUX/1:	7.0	;select LSHF(ZEXT[IR[7:0]],1)
	ADDER	;select output of address adder
ALUK/2:	ADD, AND, XOR, PASSA	
MIO.EN/1:	NO, YES	
R.W/1:	RD, WR	
DATA.SIZE/1:	BYTE, WORD	
LSHF1/1:	NO, YES	

Table C.1: Data path control signals

Simple Design of the Control Structure




COND1
COND0
BEN
R
IR[11]
Branch
Ready
Addr.
Mode
J[5]
J[4]
J[3]
J[2]
J[1]
J[0]
0,0,IR[15:12]
6
IRD
6
Address of Next State

IRD	Cond	J	LD.MAR	LD.MDR	LD.IR	LD.BEN	LD.REG	LD.CC	LD.PC	GatePC	GateMDR	GateALU	GateMARMUX	GateSHF	PCMUX	DRMUX	SR1MUX	ADDR1MUX	ADDR2MUX	MARMUX	ALUK	MIO.EN	R.W	DATA.SIZE	LSHF1	
																										000000 (State 0)
																										000001 (State 1)
																										000010 (State 2)
																										000011 (State 3)
																										000100 (State 4)
																										000101 (State 5)
																										000110 (State 6)
																										000111 (State 7)
																										001000 (State 8)
																										001001 (State 9)
																										001010 (State 10)
																										001011 (State 11)
																										001100 (State 12)
																										001101 (State 13)
																										001110 (State 14)
																										001111 (State 15)
																										010000 (State 16)
																										010001 (State 17)
																										010010 (State 18)
																										010011 (State 19)
																										010100 (State 20)
																										010101 (State 21)
																										010110 (State 22)
																										010111 (State 23)
																										011000 (State 24)
																										011001 (State 25)
																										011010 (State 26)
																										011011 (State 27)
																										011100 (State 28)
																										011101 (State 29)
																										011110 (State 30)
																										011111 (State 31)
																										100000 (State 32)
																										100001 (State 33)
																										100010 (State 34)
																										100011 (State 35)
																										100100 (State 36)
																										100101 (State 37)
																										100110 (State 38)
																										100111 (State 39)
																										101000 (State 40)
																										101001 (State 41)
																										101010 (State 42)
																										101011 (State 43)
																										101100 (State 44)
																										101101 (State 45)
																										101110 (State 46)
																										101111 (State 47)
																										110000 (State 48)
																										110001 (State 49)
																										110010 (State 50)
																										110011 (State 51)
																										110100 (State 52)
																										110101 (State 53)
																										110110 (State 54)
																										110111 (State 55)
																										111000 (State 56)
																										111001 (State 57)
																										111010 (State 58)
																										111011 (State 59)
																										111100 (State 60)
																										111101 (State 61)
																										111110 (State 62)
																										111111 (State 63)

End of the Exercise in Microprogramming

Variable-Latency Memory

- The ready signal (R) enables memory read/write to execute correctly
  - Example: transition from state 33 to state 35 is controlled by the R bit asserted by memory when memory data is available

- Could we have done this in a single-cycle microarchitecture?

- What did we assume about memory and registers in a single-cycle microarchitecture?

The Microsequencer: Advanced Questions

- What happens if the machine is interrupted?

- What if an instruction generates an exception?

- How can you implement a complex instruction using this control structure?
  - Think REP MOVS instruction in x86
    - string copy of N elements starting from address A to address B

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)

Let's Do Some More Microprogramming

- Implement REP MOVS in the LC-3b microarchitecture

- What changes, if any, do you make to the
  - state machine?
  - datapath?
  - control store?
  - microsequencer?

- Show all changes and microinstructions
- Optional HW Assignment

x86 REP MOVS (String Copy) Instruction

REP MOVS **(DEST SRC)**

```
IF AddressSize = 16
    THEN
        Use CX for CountReg;
    ELSE IF AddressSize = 64 and REX.W used
        THEN Use RCX for CountReg; FI;
    ELSE
        Use ECX for CountReg;
FI;
WHILE CountReg ≠ 0
    DO
        Service pending interrupts (if any);
        Execute associated string instruction;
        CountReg ← (CountReg – 1);
        IF CountReg = 0
            THEN exit WHILE loop; FI;
        IF (Repeat prefix is REPZ or REPE) and (ZF = 0)
        or (Repeat prefix is REPNZ or REPNE) and (ZF = 1)
            THEN exit WHILE loop; FI;
    OD;
```

```
DEST ← SRC;
IF (Byte move)
    THEN IF DF = 0
        THEN
            (R|E)SI ← (R|E)SI + 1;
            (R|E)DI ← (R|E)DI + 1;
        ELSE
            (R|E)SI ← (R|E)SI – 1;
            (R|E)DI ← (R|E)DI – 1;
        FI;
    ELSE IF (Word move)
        THEN IF DF = 0
            (R|E)SI ← (R|E)SI + 2;
            (R|E)DI ← (R|E)DI + 2;
            FI;
        ELSE
            (R|E)SI ← (R|E)SI – 2;
            (R|E)DI ← (R|E)DI – 2;
        FI;
    ELSE IF (Doubleword move)
        THEN IF DF = 0
            (R|E)SI ← (R|E)SI + 4;
            (R|E)DI ← (R|E)DI + 4;
            FI;
        ELSE
            (R|E)SI ← (R|E)SI – 4;
            (R|E)DI ← (R|E)DI – 4;
        FI;
    ELSE IF (Quadword move)
        THEN IF DF = 0
            (R|E)SI ← (R|E)SI + 8;
            (R|E)DI ← (R|E)DI + 8;
            FI;
        ELSE
            (R|E)SI ← (R|E)SI – 8;
            (R|E)DI ← (R|E)DI – 8;
        FI;
FI;
```

How many instructions does this take in MIPS ISA?

How many microinstructions does this take to add to the LC-3b microarchitecture?

Aside: Alignment Correction in Memory

- Unaligned accesses

- LC-3b has byte load and byte store instructions that move data not aligned at the word-address boundary
  - Convenience to the programmer/compiler

- How does the hardware ensure this works correctly?
  - Take a look at state 29 for LDB
  - States 24 and 17 for STB
  - Additional logic to handle unaligned accesses

- P&P, Revised Appendix C.5

Aside: Memory Mapped I/O

- Address control logic determines whether the specified address of LDW and STW are to memory or I/O devices
- Correspondingly enables memory or I/O devices and sets up muxes
- An instance where the final control signals of some datapath elements (e.g., MEM.EN or INMUX/2) **cannot** be stored in the control store
  - These signals are dependent on memory address
- P&P, Revised Appendix C.6

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, INC [MEM])

- 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

Update of Machine Behavior

- The ability to update/patch microcode in the field (after a processor is shipped) enables
  - Ability to add new instructions without changing the processor!
  - Ability to "fix" buggy hardware implementations

- Examples
  - IBM 370 Model 145: microcode stored in main memory, can be updated after a reboot
  - IBM System z: Similar to 370/145.
    - Heller and Farrell, "Millicode in an IBM zSeries processor," IBM JR&D, May/Jul 2004.
  - B1700 microcode can be updated while the processor is running
    - User-microprogrammable machine!
    - Wilner, "Microprogramming environment on the Burroughs B1700", CompCon 1972.

Multi-Cycle vs. Single-Cycle uArch

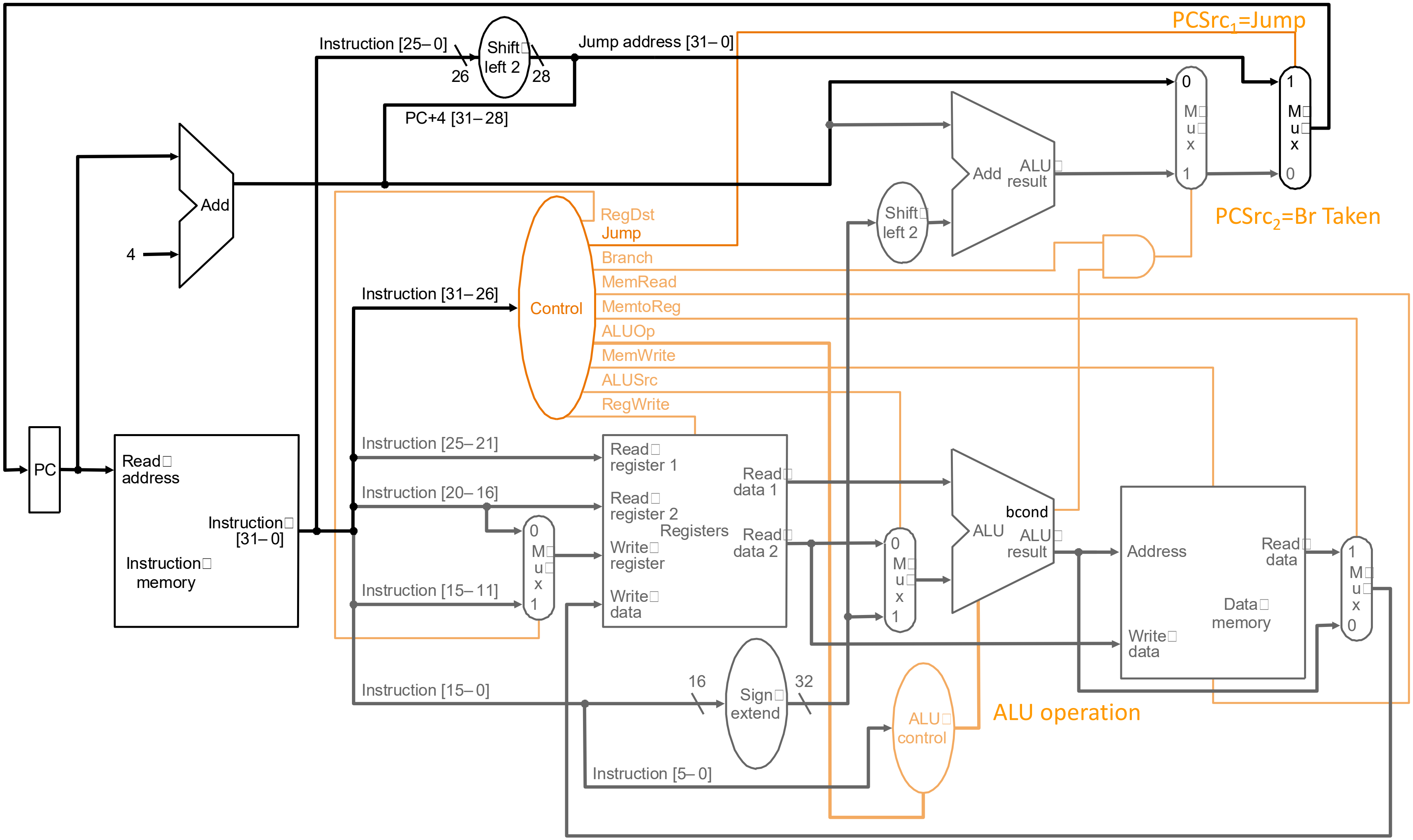
- Advantages

- Disadvantages

- For you to fill in

Segue into Pipelining

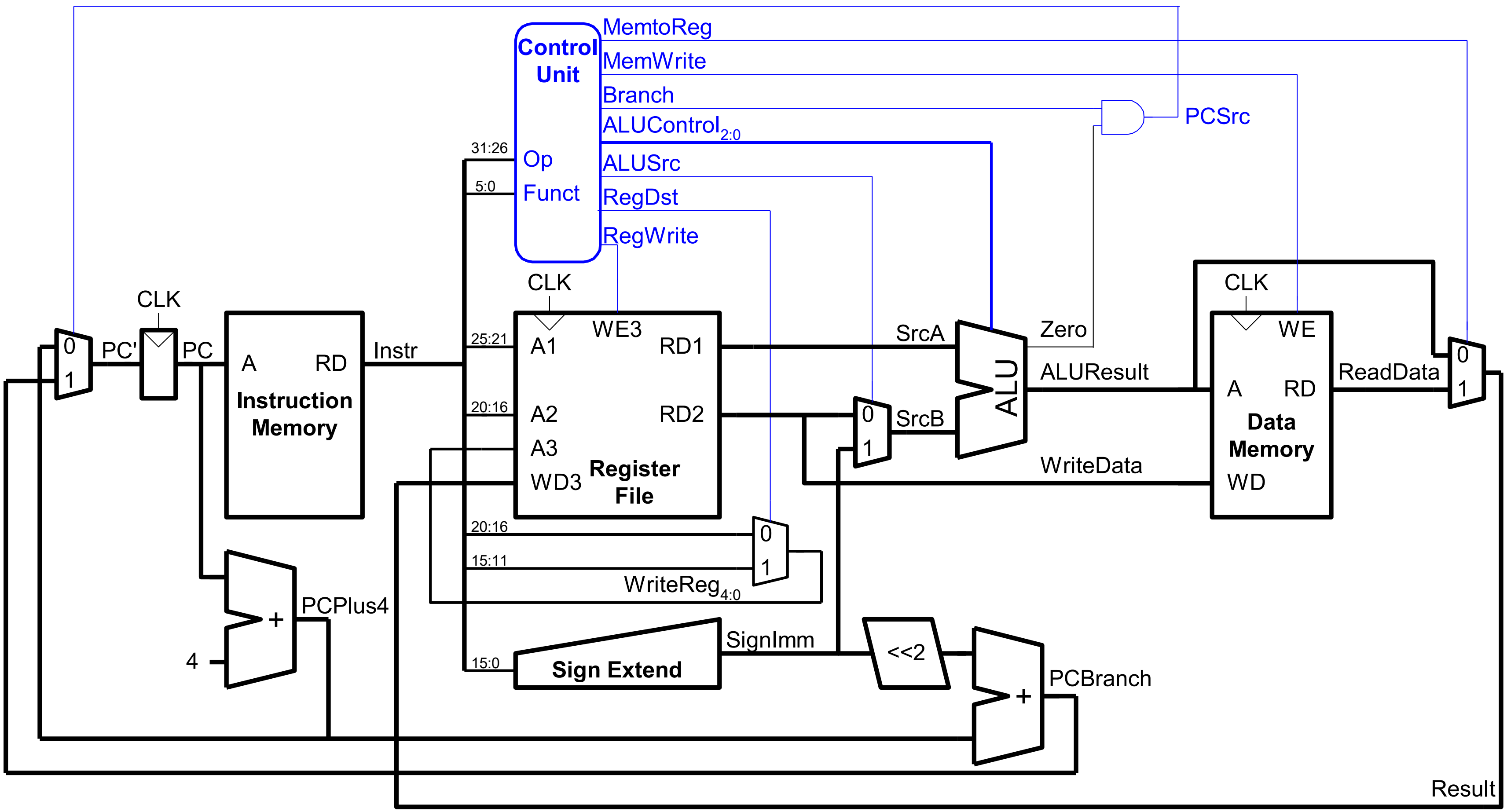
Review: Single-Cycle MIPS Processor (I)



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

JAL, JR, JALR omitted

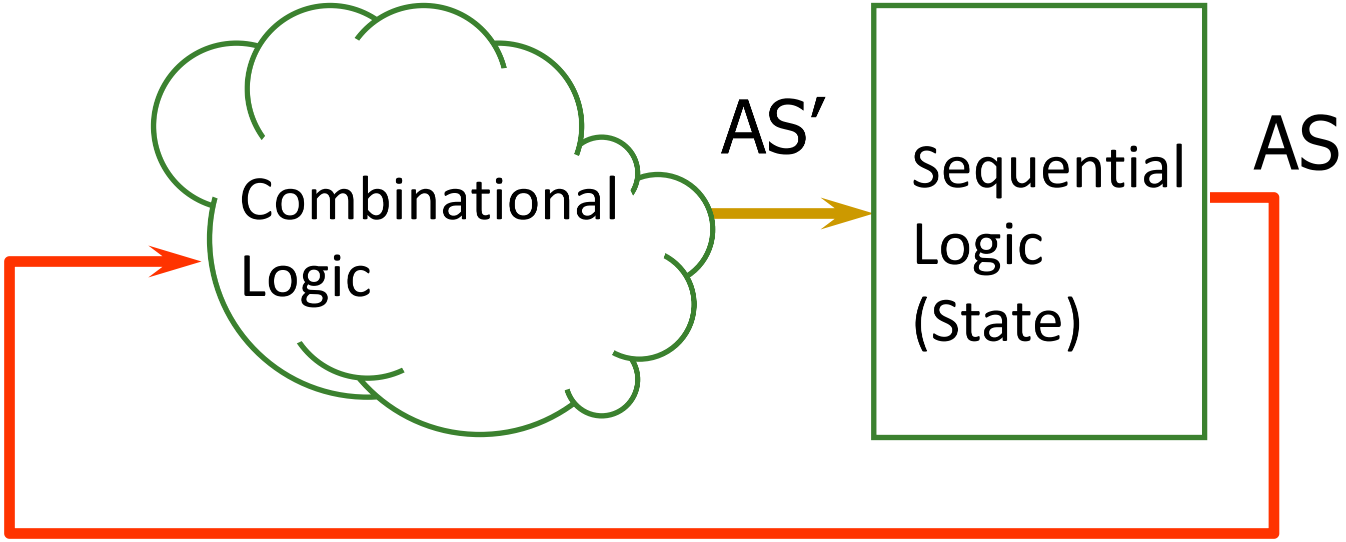
Review: Single-Cycle MIPS Processor (II)



Single-cycle processor. Harris and Harris, Chapter 7.3.

Review: Single-Cycle MIPS FSM

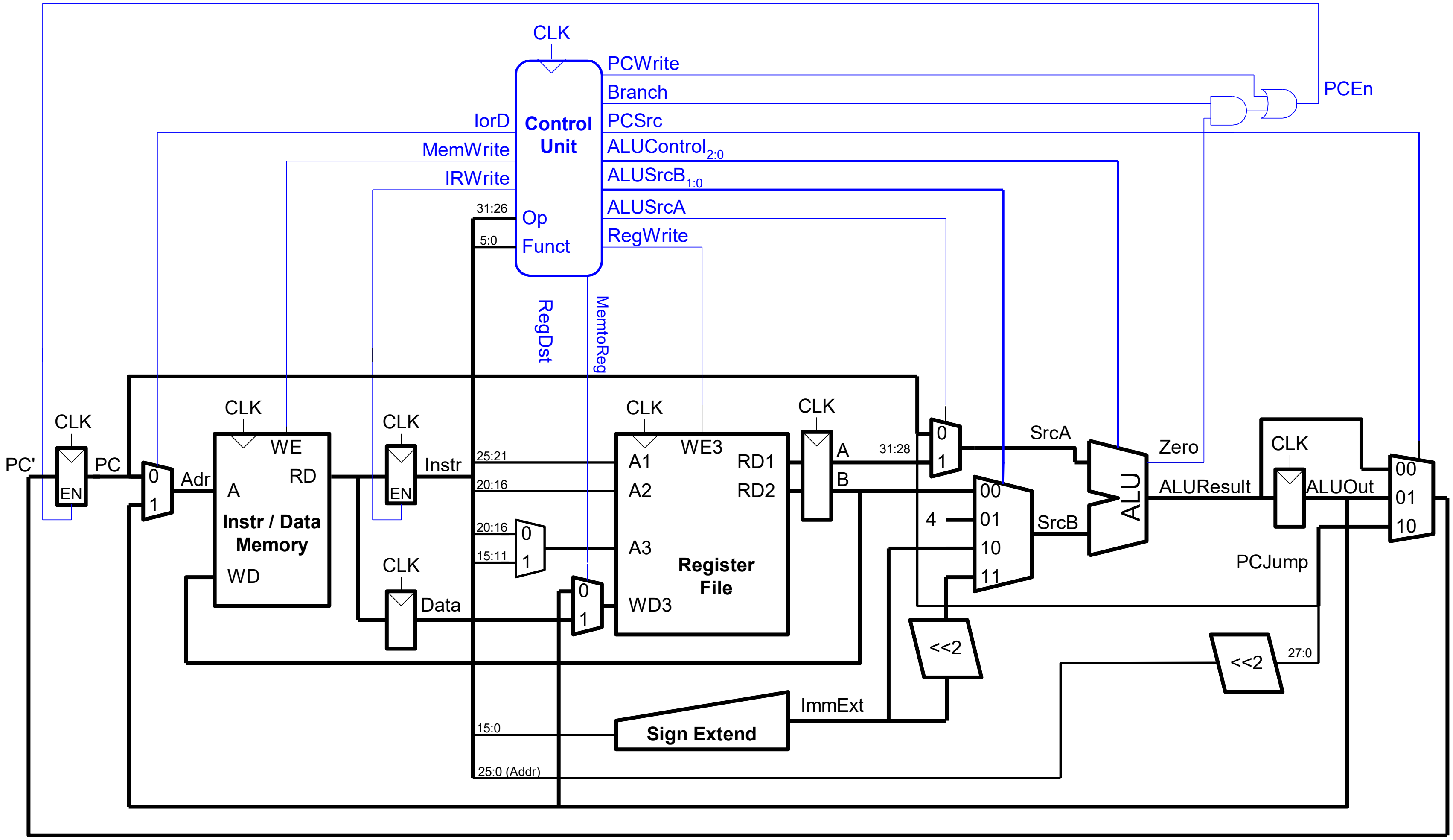
- Single-cycle machine



AS: Architectural State

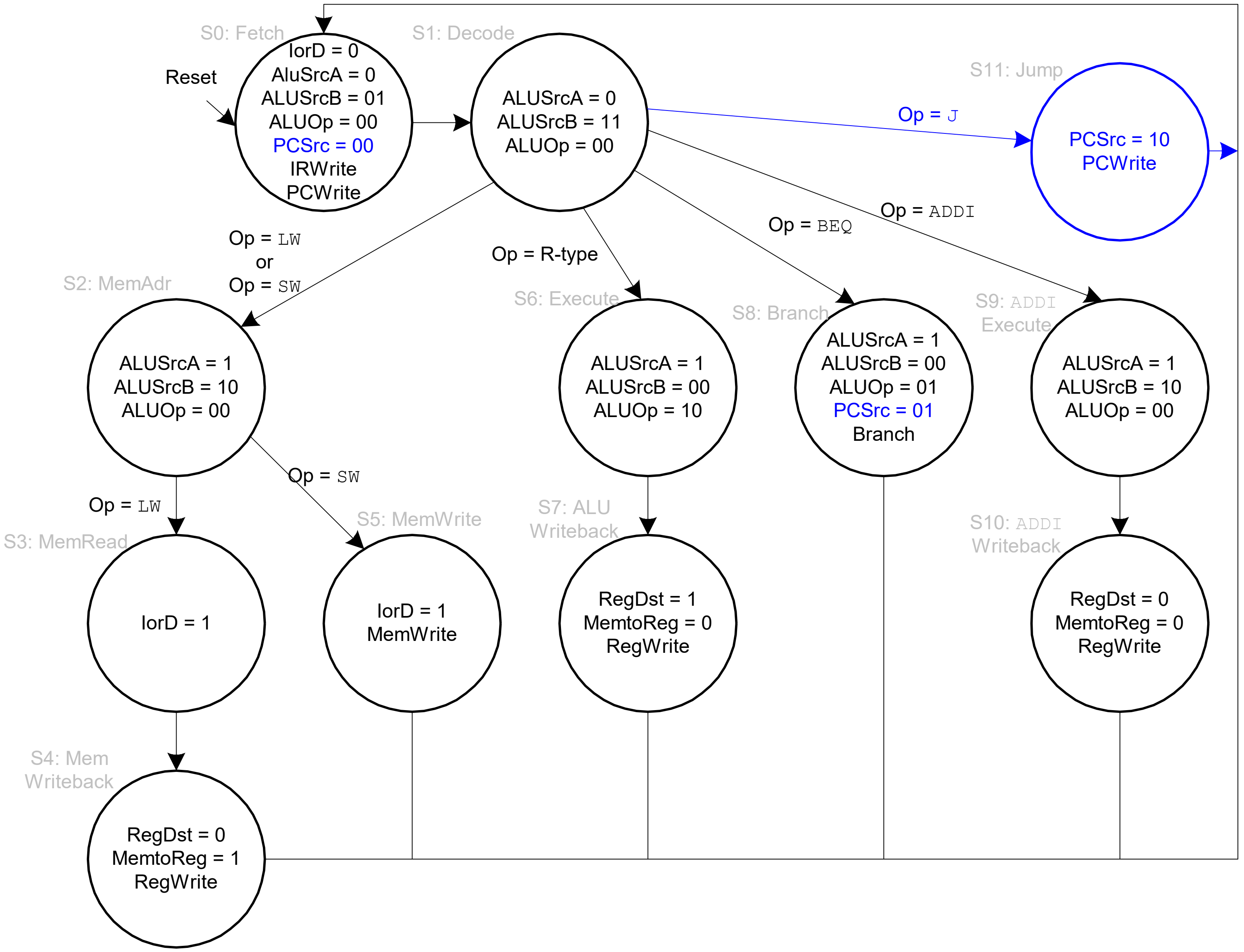
Can We Do Better?

Review: Multi-Cycle MIPS Processor



Multi-cycle processor. Harris and Harris, Chapter 7.4.

Review: Multi-Cycle MIPS FSM



What is the shortcoming of this design?

What does this design assume about memory?

Can We Do Better?

Can We Do Better?

- What limitations do you see with the multi-cycle design?

- Limited concurrency
  - Some hardware resources are idle during different phases of instruction processing cycle
  - “Fetch” logic is idle when an instruction is being “decoded” or “executed”
  - Most of the datapath is idle when a memory access is happening

Can We Use the Idle Hardware to Improve Concurrency?

- Goal: More concurrency → Higher instruction throughput (i.e., more "work" completed in one cycle)

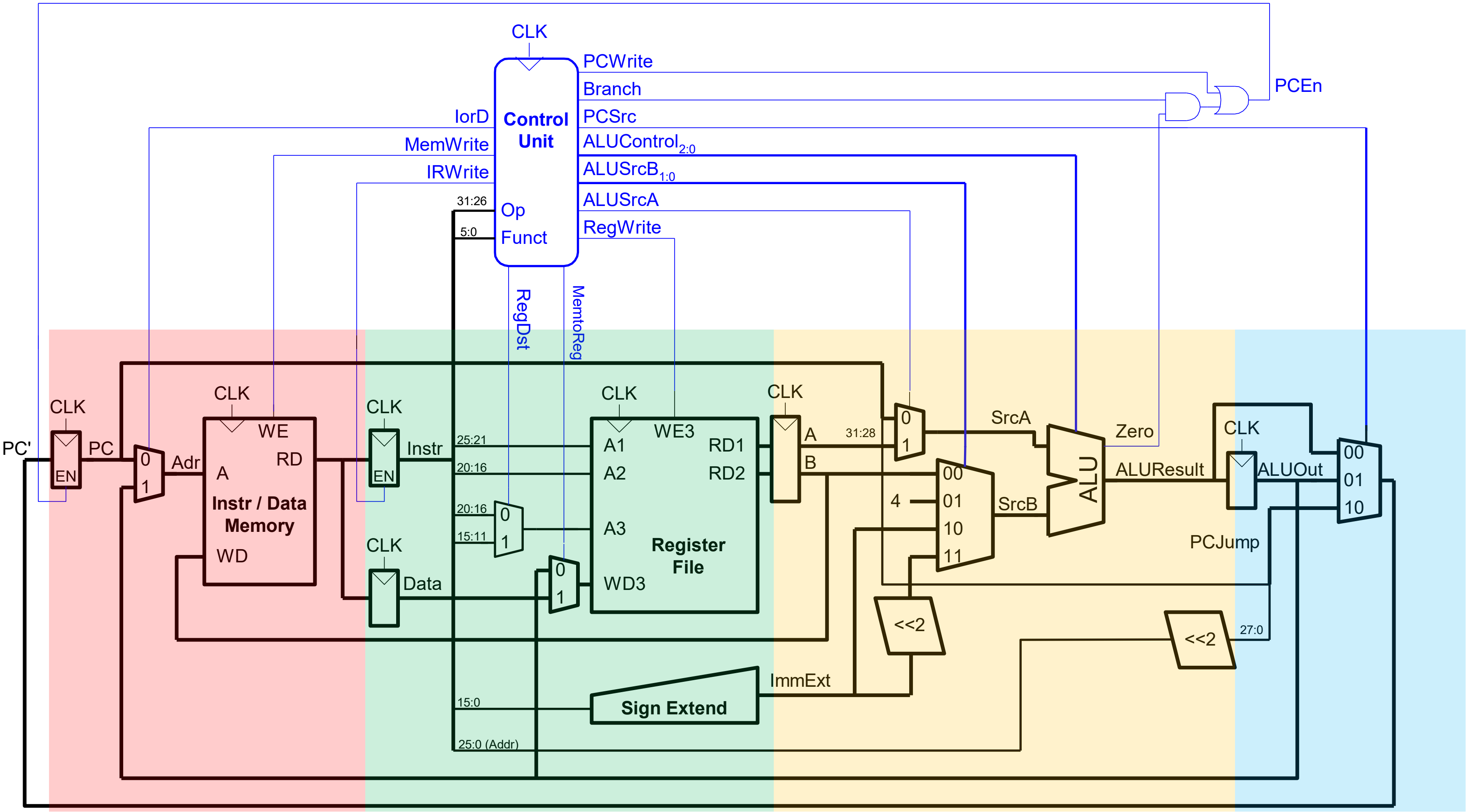
- Idea: When an instruction is using some resources in its processing phase, process other instructions on idle resources not needed by that instruction
  - E.g., when an instruction is being decoded, fetch the next instruction
  - E.g., when an instruction is being executed, decode another instruction
  - E.g., when an instruction is accessing data memory (ld/st), execute the next instruction
  - E.g., when an instruction is writing its result into the register file, access data memory for the next instruction

Can Have Different Instructions in Different Stages

- Fetch
- Decode
- Evaluate Address
- Fetch Operands
- Execute
- Store Result

1. Instruction fetch (IF)
2. Instruction decode and register operand fetch (ID/RF)
3. Execute/Evaluate memory address (EX/AG)
4. Memory operand fetch (MEM)
5. Store/writeback result (WB)

Can Have Different Instructions in Different Stages



Of course, we need to be more careful than this!

Pipelining

Pipelining: Basic Idea

- More systematically:
  - Pipeline the execution of multiple instructions
  - Analogy: “Assembly line processing” of instructions
- Idea:
  - Divide the instruction processing cycle into distinct “stages” of processing
  - Ensure there are enough hardware resources to process one instruction in each stage
  - Process a **different** instruction in each stage
    - Instructions consecutive in program order are processed in consecutive stages
- Benefit: Increases instruction processing throughput (1/CPI)
- Downside: Start thinking about this…