361-1-4201
Computer Architecture
Intro to Microarchitecture: Single-Cycle, Multi-Cycle

Dr. Guy Tel-Zur

Based on slides by Prof. Onur Mutlu

Spring 2015

and

H&H Chapter 7

Agenda for Today & Next Few Lectures

- Start Microarchitecture
- Single-cycle Microarchitectures

Lectures 4-5

- Multi-cycle Microarchitectures
- Microprogrammed Microarchitectures

Lecture 6

- Pipelining
- Issues in Pipelining: Control & Data Dependence Handling, State Maintenance and Recovery, …

Lect 7

Recap of the Last Lectures

- Computer Architecture Today and Basics – Lecture #1
- Fundamental Concepts – Lecture #1
  - Computer Arch = ISA (the definition) + uArch (the implementation)
- ISA basics and tradeoffs – Lecture #2
  - Instruction length
  - Uniform vs. non-uniform decode
  - Number of registers
  - Addressing modes
  - Aligned vs. unaligned access
  - RISC vs. CISC properties
- RISC-V ISA – Lecture #3
  - RISC-V ISA Overview
- Datapath & Control, and an introduction to Pipeline – Lecture #4

חומרים למחשבה

- As you learn the RISC-V ISA, think about what **tradeoffs** the designers have made
  - in terms of the ISA properties we talked about

- And, think about the **pros and cons** of design choices
  - In comparison to ARM, Alpha
  - In comparison to x86, VAX

- And, think about the **potential mistakes**
  - Branch delay slot? (stall in the pipeline – טרם למדנו)
  - Load delay slot? (להבטיח שההוראה הבאה תתבצע ללא עיכוב)
  - No FP, no multiply, MIPS (initial)

Food for Thought for You עוד חומר למחשבה

- How would you design a new ISA?
- Where would you place it?
- What design choices would you make in terms of ISA properties?
- What would be the first question you ask in this process?
  - "What is my design point?"
    - The 1st question to ask what is the Design Point / Use Case

מה השאלה הראשונה שתרצו לשאול את המזמין?

Look Forward & Up

Review: Other Example ISA-level Tradeoffs

- Condition codes vs. not – יילמד בהמשך
- VLIW vs. single instruction – אם נספיק יילמד בהמשך
- SIMD (single instruction multiple data) vs. SISD – הטקסונומיה של פלין
- Precise vs. imprecise exceptions – יילמד בהמשך
- Virtual memory vs. not
- Unaligned access vs. not
- Hardware interlocks vs. software-guaranteed interlocking
- Software vs. hardware managed page fault handling - לא יילמד
- Cache coherence (hardware vs. software) – לא יילמד
- …

Think Programmer vs. (Micro)architect היכן למקם את המעמסה?

Review: A Note on RISC vs. CISC

- **RISC**
  - Simple instructions
  - Fixed length
  - Uniform decode
  - Few addressing modes

- **CISC**
  - Complex instructions
  - Variable length
  - Non-uniform decode
  - Many addressing modes

Now That We Have an ISA

- How do we implement it?

- i.e., how do we design a system that obeys the hardware/software interface?

- Aside: “System” can be solely hardware or a combination of hardware and software
  - Remember “Translation of ISAs”
  - A virtual ISA can be converted by “software” into an implementation ISA

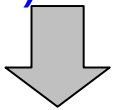
- We will assume “hardware” for most lectures

Implementing the ISA: Microarchitecture Basics

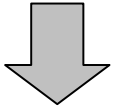
How Does a Machine Process Instructions?

- What does processing an instruction mean?
- Remember the von Neumann model

AS = Architectural (programmer visible) state before an instruction is processed



Process instruction

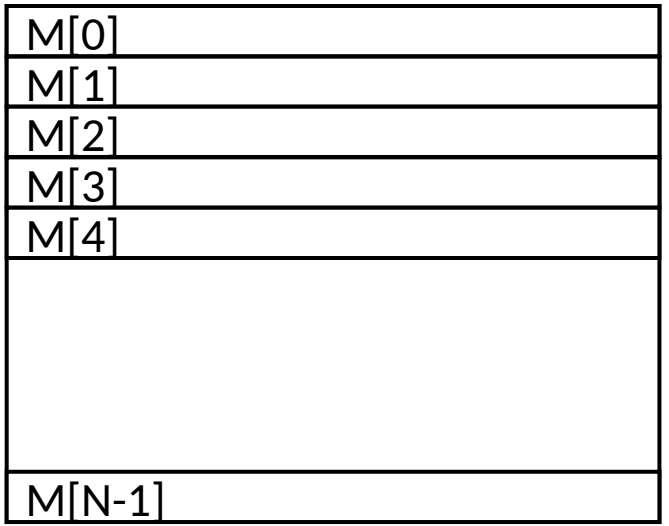


AS’ = Architectural (programmer visible) state after an instruction is processed

- Processing an instruction: Transforming AS to AS’ according to the ISA specification of the instruction

Remember: Programmer Visible (Architectural) State

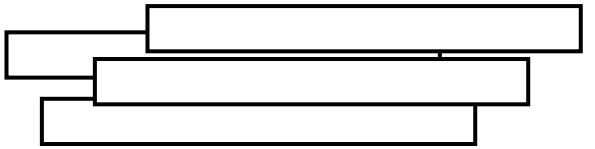
הגדרת AS



Memory

array of storage locations

indexed by an address



Registers

- given special names in the ISA
  (as opposed to addresses)
- general vs. special purpose

Program Counter

memory address

of the **current** instruction

Instructions (and programs) specify how to transform the values of programmer visible state

הערה לעצמי: הדגמת דבאגר
/home/telzur/science/Teaching/CPU/lectures/05/code/ddd_demo.c

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

Finite State Machine

AS = Architecture State

MS=Microarchitecture State

- 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')

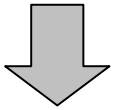
A Very Basic Instruction Processing Engine

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

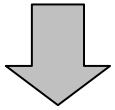
בהרצאה זו

AS = Architectural (programmer visible) state

at the beginning of a clock cycle



Process instruction in one clock cycle

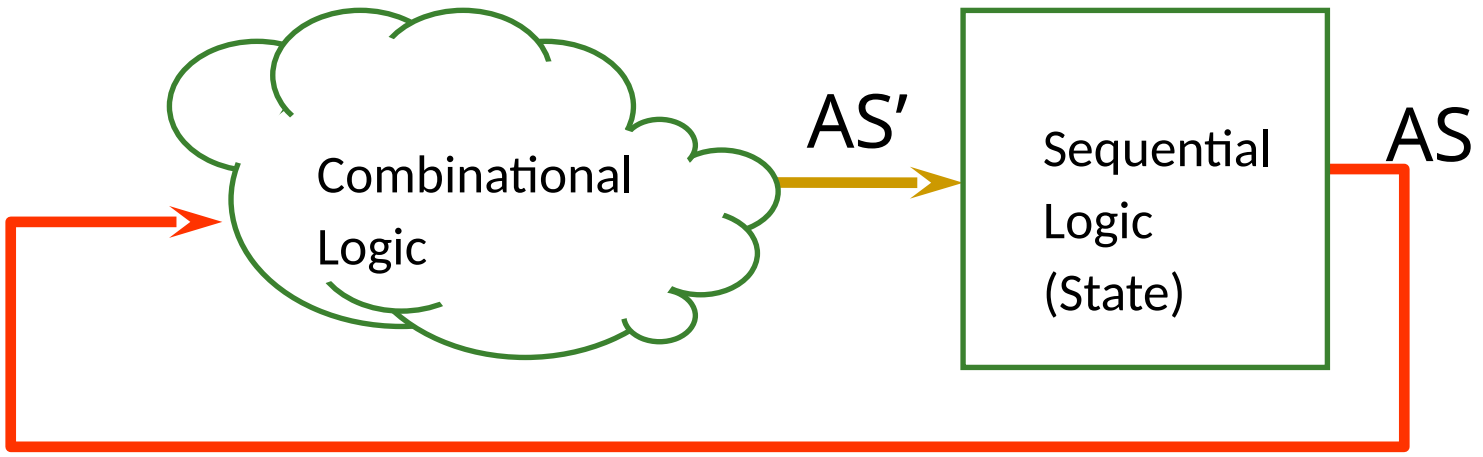


AS’ = Architectural (programmer visible) state

at the end of a clock cycle

A Very Basic Instruction Processing Engine

- Single-cycle machine



נשאל 2 שאלות,
מישהו רוצה לענות?

- What is the *clock cycle time* determined by?
- What is the *critical path* of the combinational logic determined by?

Single-cycle vs. Multi-cycle Machines

- Single-cycle machines
  - Each instruction takes a single clock cycle
  - All state updates made at the end of an instruction's execution
  - Big disadvantage: The slowest instruction determines cycle time ✉ long clock cycle time

- Multi-cycle machines
  - Instruction processing broken into multiple cycles (or stages)
  - State updates can be made during an instruction's execution
  - Architectural state(*) updates made only at the end of an instruction's execution
  - Advantage over single-cycle: The slowest "stage" determines cycle time

- Both single-cycle and multi-cycle machines literally follow the von Neumann model at the microarchitecture level

(*) "מצב" כפי שהוגדר בשקף הקודם

Instruction Processing “Cycle”

- Instructions are processed under the direction of a “control unit” step by step.
- Instruction cycle: Sequence of steps to process an instruction
- Fundamentally, there are six phases (steps):

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

➔ Not all instructions require all six stages

- IF - Fetch
- ID - Decode
- EX - Execute/Evaluate Address
- MEM - Fetch Operands
- WB - Store Result

We will later change to 5 stages

יש להבחין בין משך של הוראה, לבין שלב של הוראה, ולבין זמן מחזור של שעון!

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

למשל יתכן שביצוע FETCH יקח 5 מחזורי שעון...

- האינטראקציה בין המעבד לזכרון במחשב מסוג single cycle היא בעייתית...

Instruction Processing Viewed Another Way

Data vs. Control separation

- Instructions transform **Data** (AS) to **Data'** (AS')
- This transformation is done by **functional units**
  - Units that "operate" on data
- These units need to be told what to do to the data

- 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 and muxes) 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

Single-cycle vs. Multi-cycle: Control & Data

- **Single-cycle machine:**
  - Control signals are generated in the same clock cycle as the one during which data signals are operated on
  - Everything related to an instruction happens in one clock cycle (serialized processing)

- **Multi-cycle machine:**
  - Control signals needed in the next cycle can be generated in the current cycle
  - Latency of control processing can be overlapped with latency of datapath operation (more parallelism)

- We will see the difference clearly in *microprogrammed multi-cycle microarchitectures*

Many Ways of Datapath and Control Design

- There are many ways of designing the data path and control logic

- Single-cycle, multi-cycle, pipelined datapath and control
- Single-bus vs. multi-bus datapaths

- Hardwired/combinational vs. microcoded/microprogrammed control(*)
  - Control signals generated by combinational logic versus
  - Control signals stored in a memory structure

- Control signals and structure depend on the datapath design

(*)מיקרו-קוד – הרצאה מס' 7

Performance Analysis

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

CPI = 1/IPC

In single cycle we have one degree of freedom to optimize

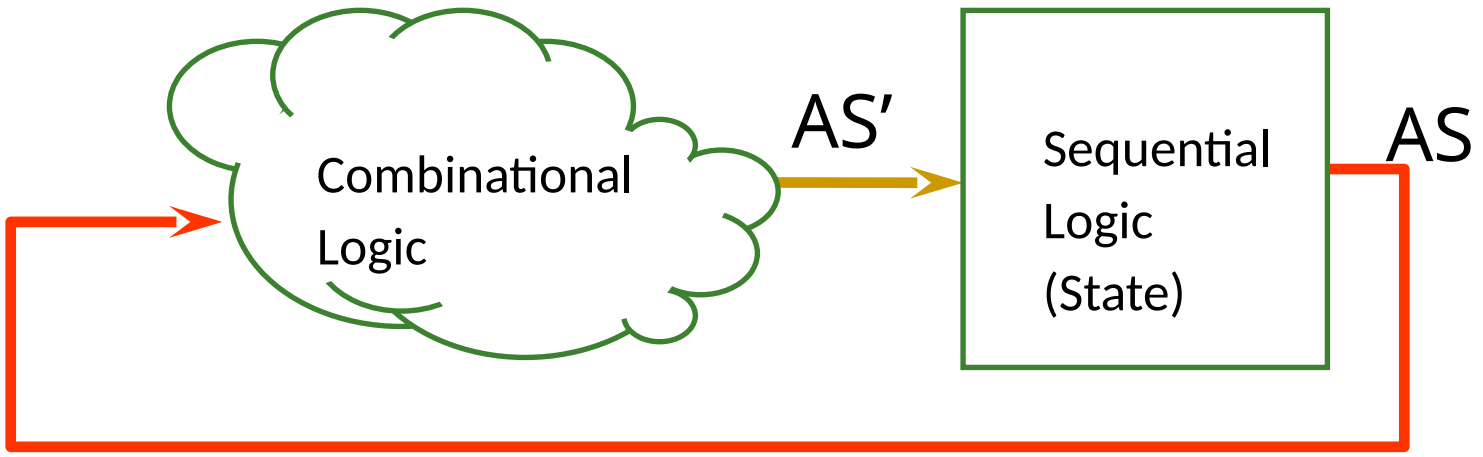
In multi-cycle we have two degrees of freedom to optimize independently

ניתן "לשחק" עם 2 גדלים: CPI ו- clock cycle time

A Single-Cycle Microarchitecture
A Closer Look

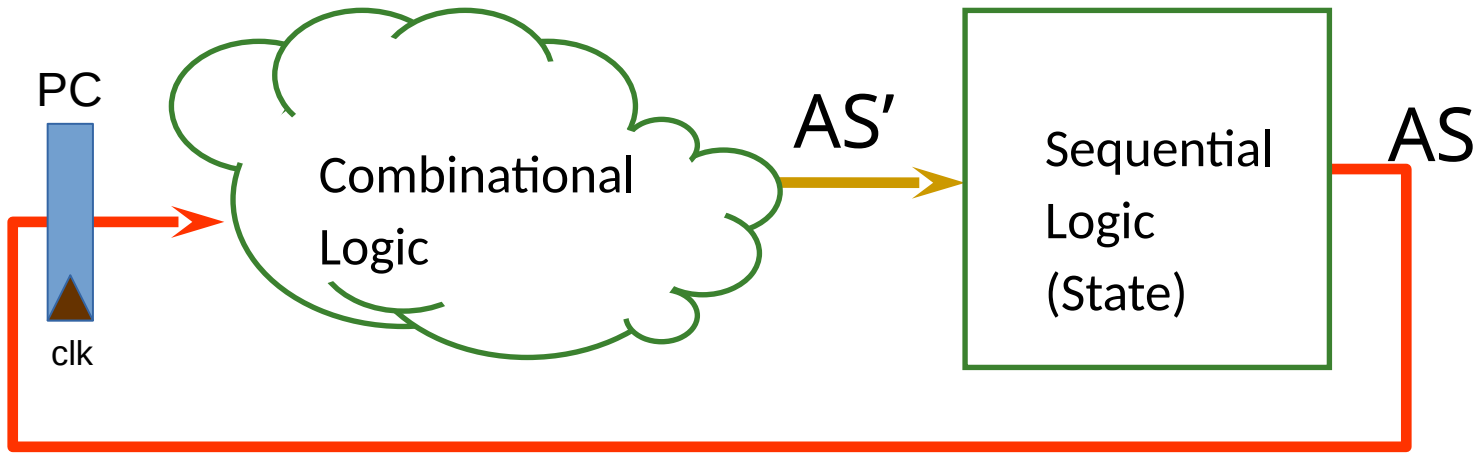
Remember…

- Single-cycle machine



למעשה:

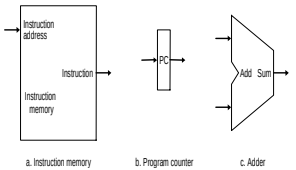
- Single-cycle machine

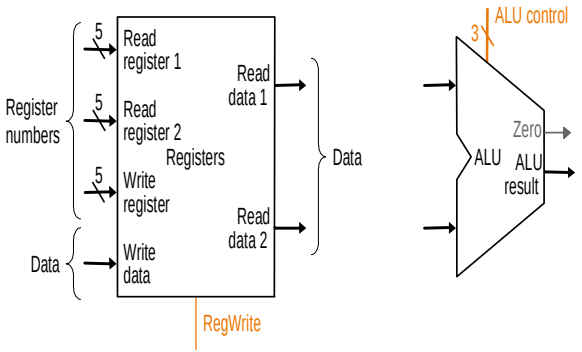


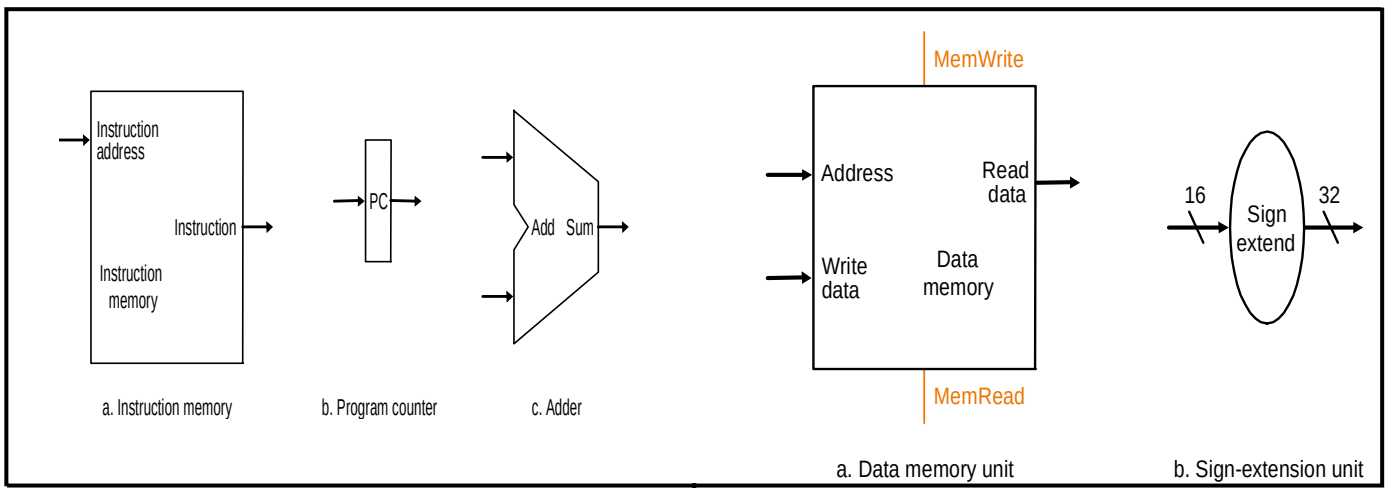
Let's Start with the State Elements

- Data and control inputs

נתחיל עם כל מה שהוא
programmer visible state







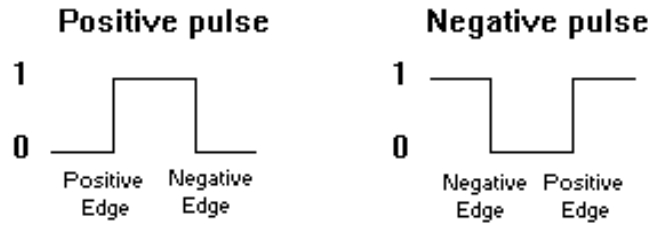
**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

For Now, We Will Assume

- "Magic" memory and register file

- Combinational read
  - output of the read data port is a combinational function of the register file contents and the corresponding read select port

- Synchronous write
  - the selected register is updated on the **positive** edge clock transition when write enable is asserted
    - Cannot affect read output in between clock edges



- Single-cycle, synchronous memory – זוהי הנחה לשם הפשטות
  - Contrast this with memory that tells when the data is ready
  - i.e., Ready bit: indicating the read or write is done

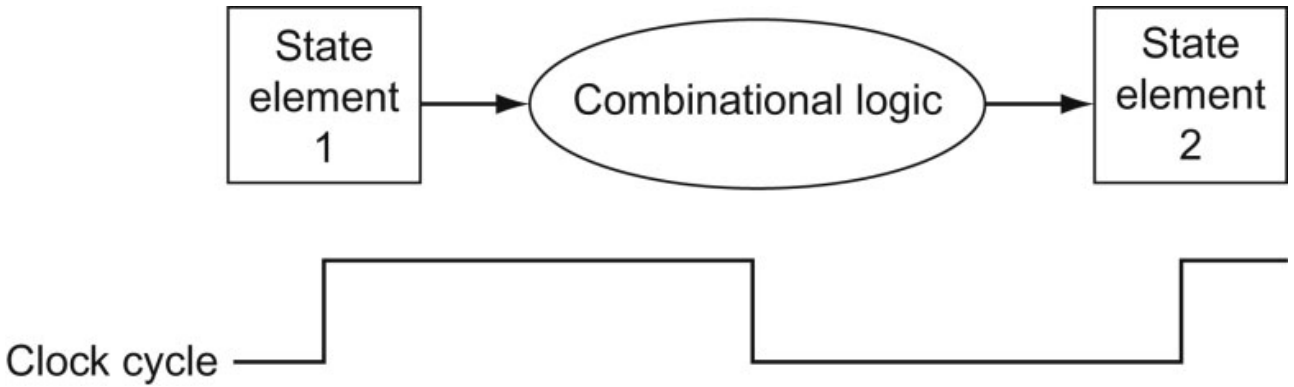


FIGURE 4.3 Combinational logic, state elements, and the clock are closely related. In a synchronous digital system, the clock determines when elements with state will write values into internal storage. Any inputs to a state element must reach a stable value (that is, have reached a value from which they will not change until after the clock edge) before the active clock edge causes the state to be updated. All state elements in this chapter, including memory, are assumed to be **positive edge-triggered**; that is, they change on the rising clock edge.

Instruction Processing

אנימציה

- 5 generic steps (P&H book) –

  אמנם דיברנו על 6 שלבים אך כאן נעשה מיזוג ושינוי קל לעומת מה שנכתב לפני מספר שקפים

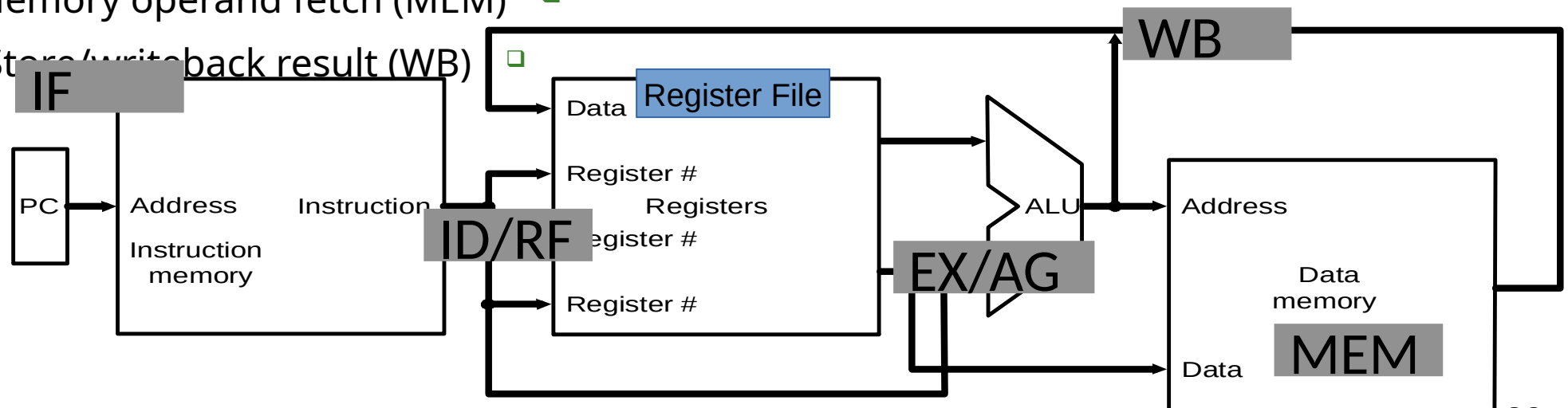
Instruction fetch (IF)

Instruction decode and register operand fetch (ID/RF = Register Fetch)

Execute/Evaluate memory address (EX/AG = Address Generate)

Memory operand fetch (MEM)

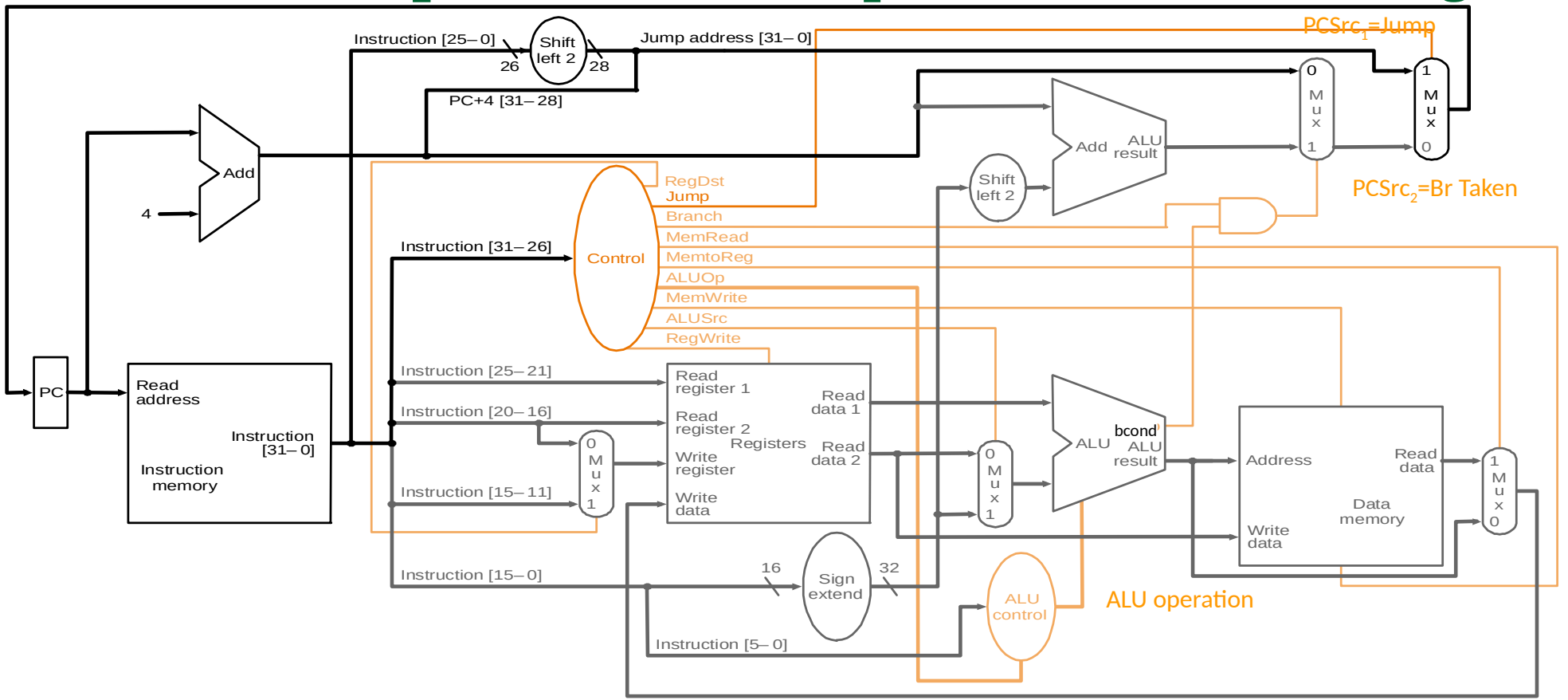
Store/writeback result (WB)



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

What Is To Come: The Full MIPS(*) Datapath

(* RISC-V is quite similar except the wires numbering)



JAL, JR, JALR omitted

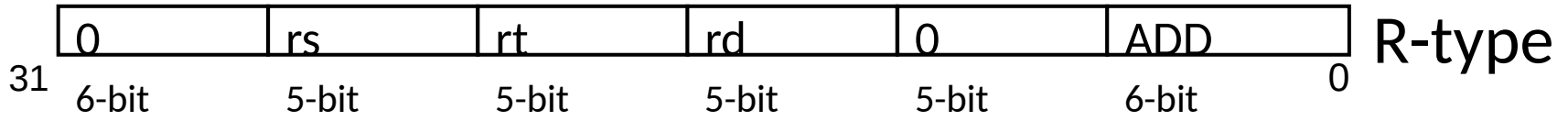
Single-Cycle Datapath for *Arithmetic and Logical Instructions*

R-Type ALU Instructions (MIPS)

- Assembly (e.g., register-register signed addition)

  ADD rd_{reg} rs_{reg} rt_{reg}

- Machine encoding



- Semantics

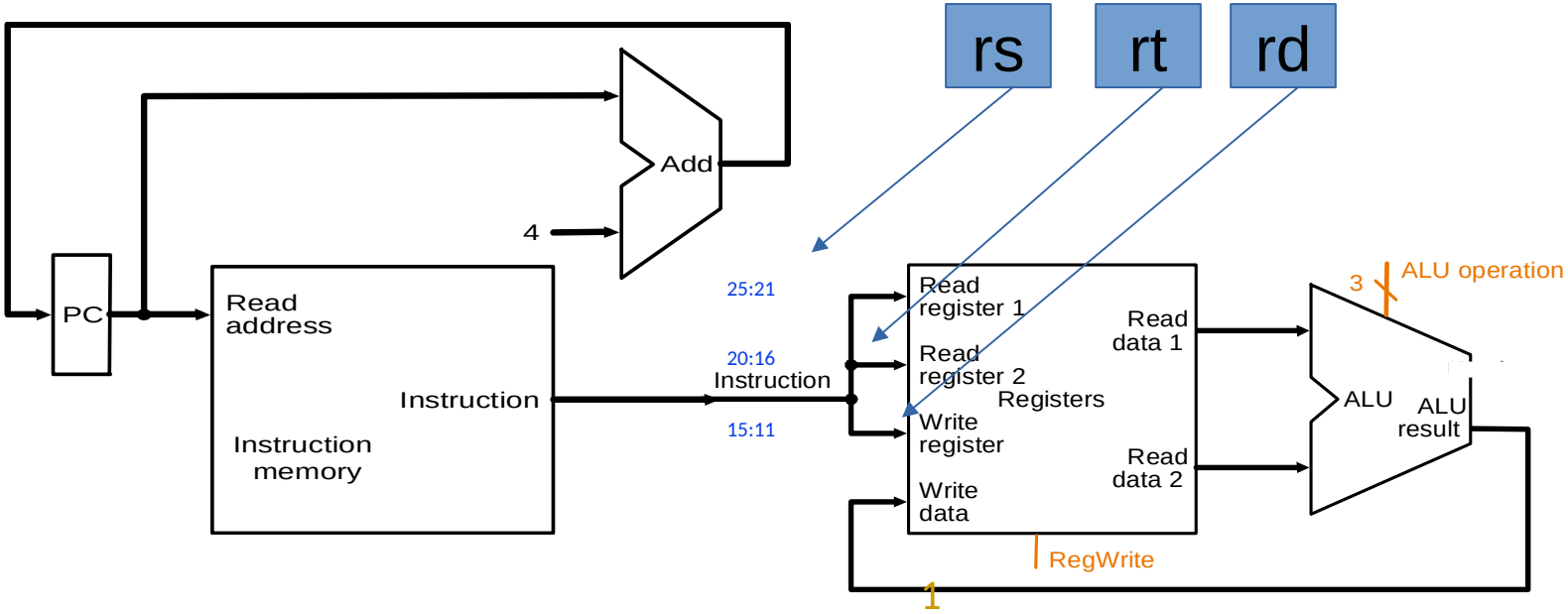
  if MEM[PC] == ADD rd rs rt

  GPR[rd] 🌎 GPR[rs] + GPR[rt]

  PC 🌎 PC + 4

אם מה שנמצא בזיכרון בכתובת PC שווה לתוכן שבאגף ימין אז...

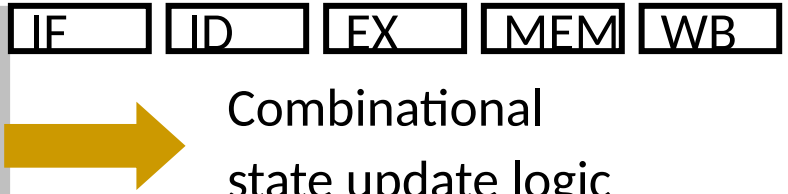
להלן המימוש ALU Datapath



if MEM[PC] == ADD rd rs rt

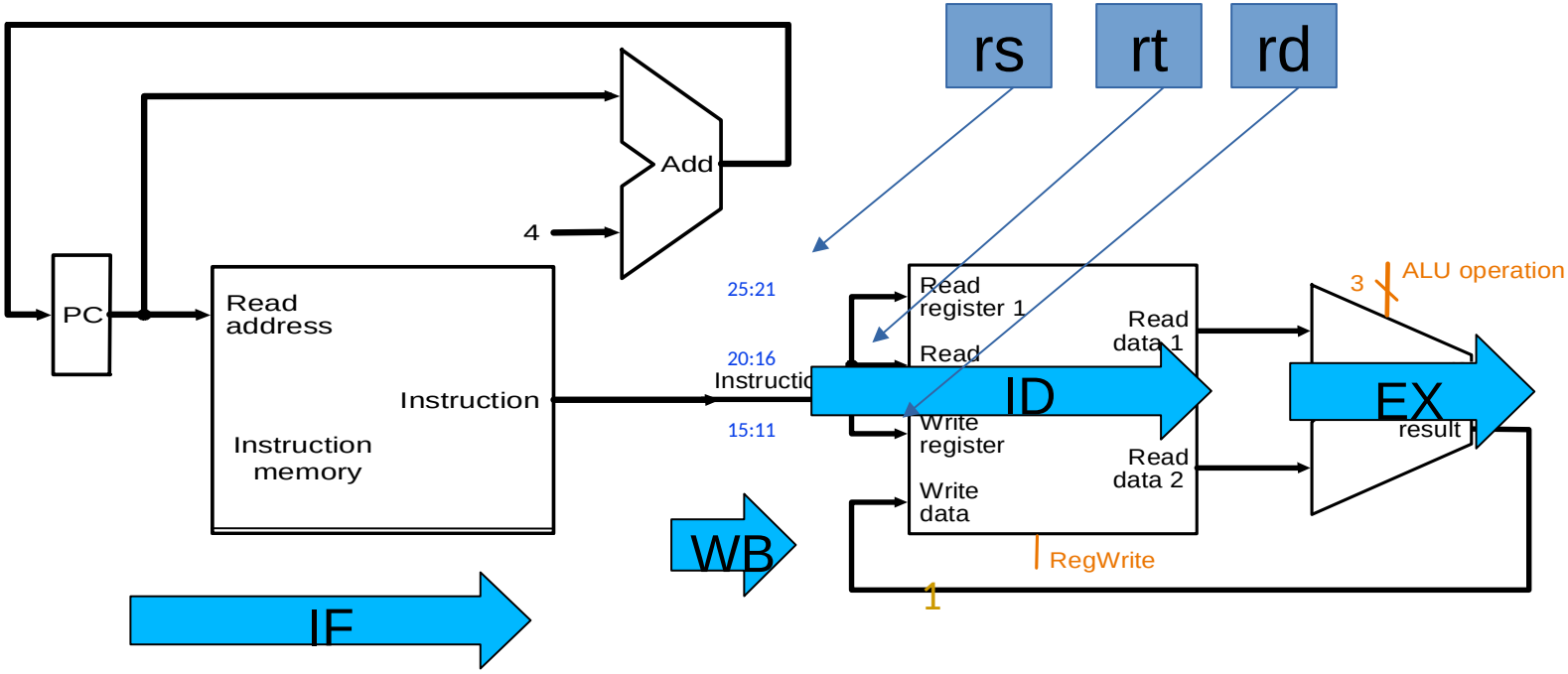
GPR[rd] 🌍 GPR[rs] + GPR[rt]

PC 🌍 PC + 4 (assuming no compressed instructions support)



להלן המימוש R-Type ALU

אנימציה



GPR[rd] ← GPR[rs] + GPR[rt]

PC ← PC + 4

IF | ID | EX | ~~MEM~~ | WB

Combinational state update logic

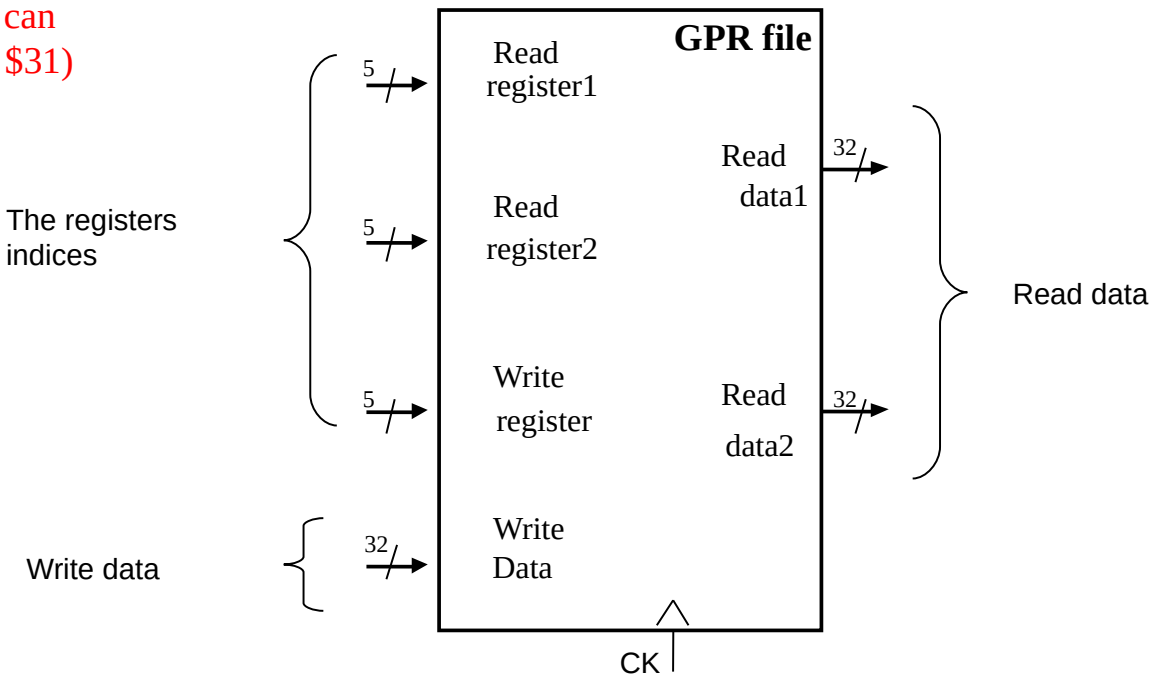
33

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

The General-Purpose Register file (GPR file). This unit is required for the decode phase (and for the Write Back phase)

אנימציה

It includes the
32 registers the
programmer can
access ($0 - $31)



Let's see how it is used:

The General-Purpose Register file (GPR).

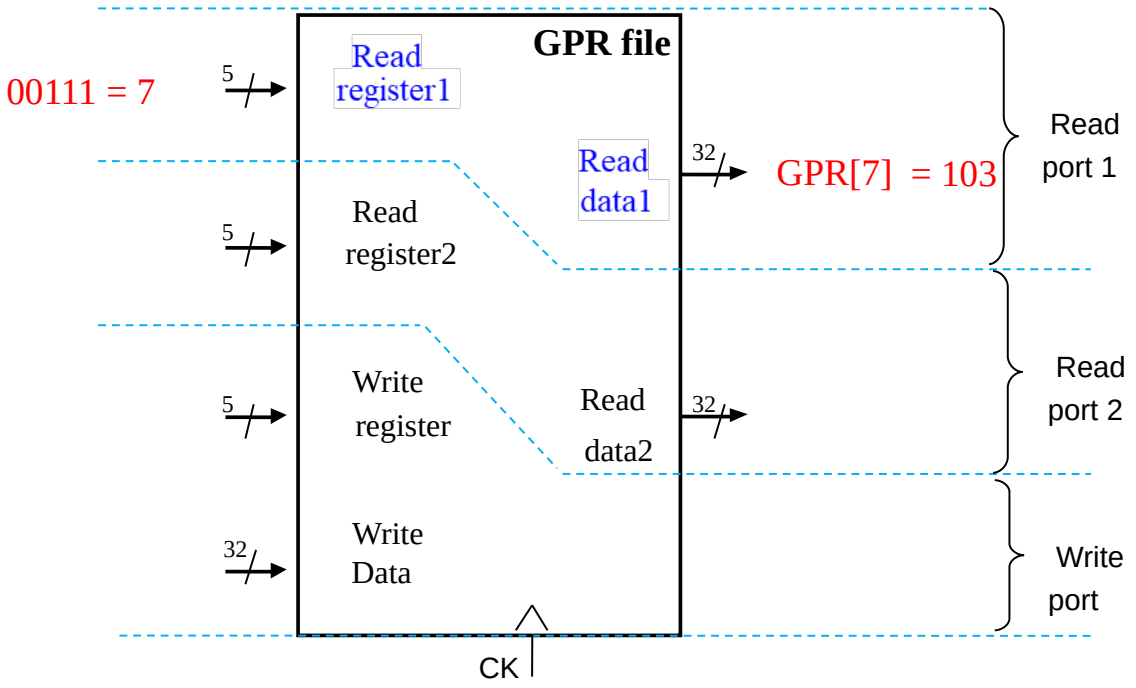
אנימציה

There are 3 ports in this box

If we want to read from $7 via port 1

we set Read register1 to 7 = 00111

we then gets the contents of $7 at the Read data1 output
we mark it as: GPR[7]
It could be 103 for example



The General-Purpose Register file (GPR).

אנימציה

So if we have the instruction:

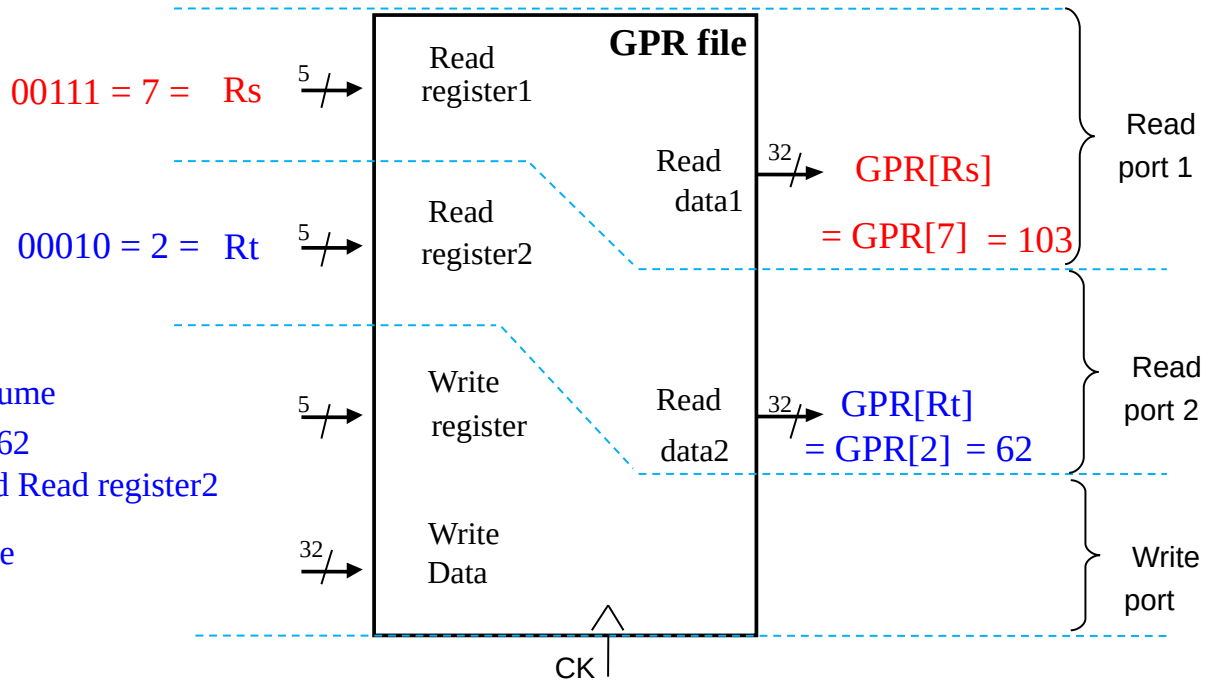
add $12, $7, $2

Rs Rt

we connect the Rs field Read reg1

and get GPR[Rs] at the Read data1 output

In our case we assumed GPR[7]=103



In our case we also assume

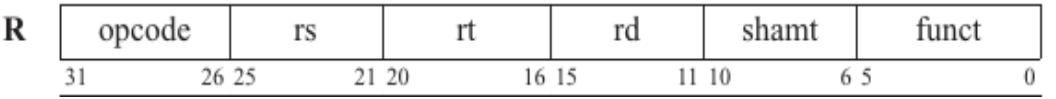
assumed that GPR[2]=62

we connect the Rt field Read register2

and get GPR[Rt] ate the

Read data2 output

The General-Purpose Register file (GPR).



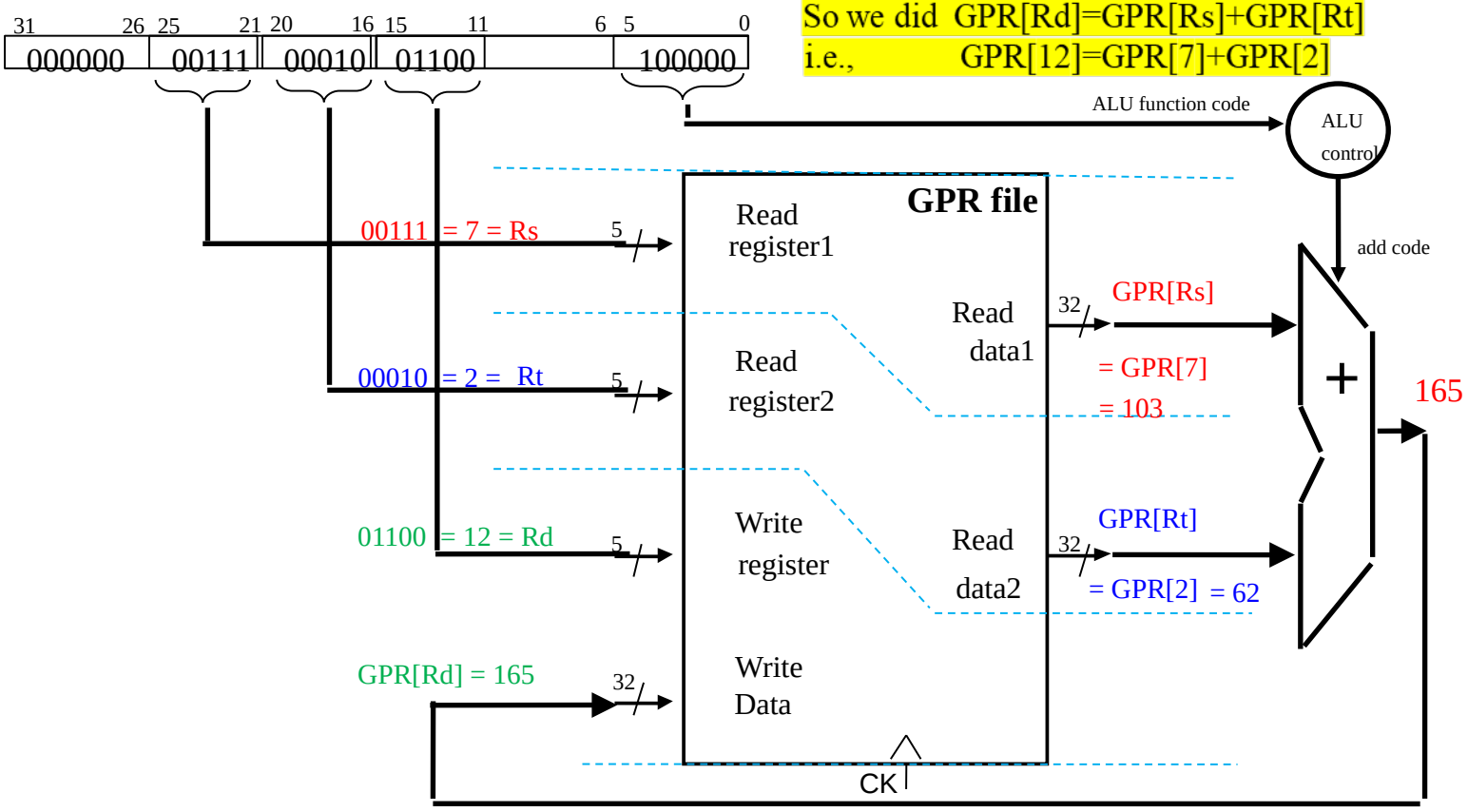
אנימציה

So this is the real picture:

add $12, $7, $2

So we did GPR[Rd]=GPR[Rs]+GPR[Rt]

i.e., GPR[12]=GPR[7]+GPR[2]



So what's next? We want to add the 103 & 62 103 + 62 = 165

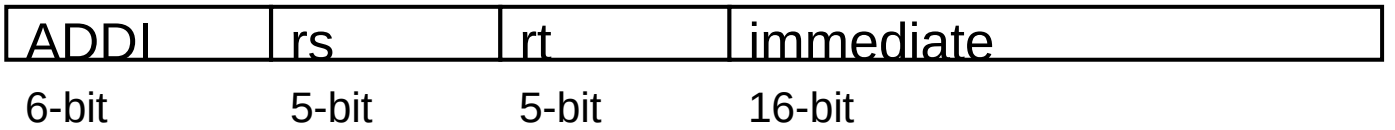
Now we want to write this to Rd

I-Type ALU Instructions

- Assembly (e.g., register-immediate signed additions)

  ADDI rt_{reg} rs_{reg} $immediate_{16}$

- Machine encoding



I-type

- Semantics

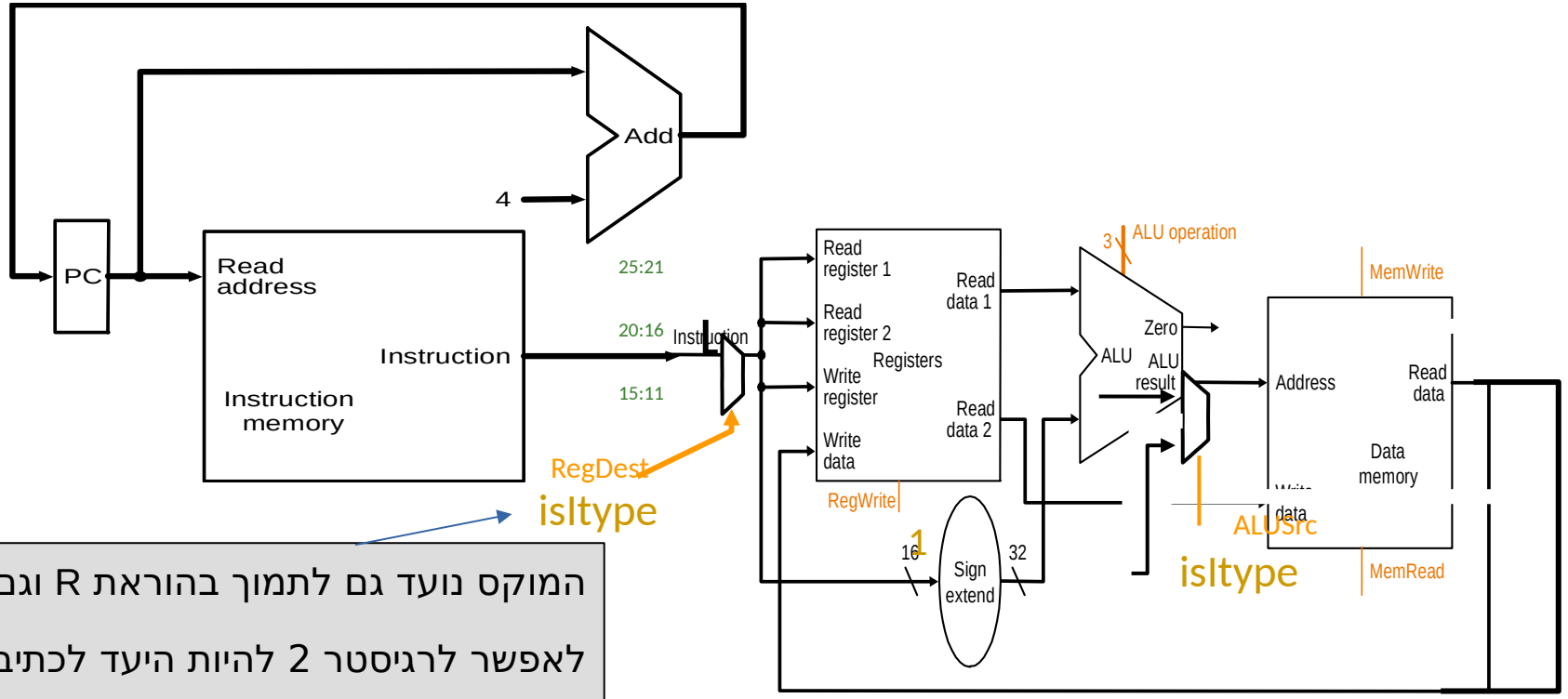
  if MEM[PC] == ADDI rt rs immediate

    GPR[rt] 🌎 GPR[rs] + sign-extend (immediate)

    PC 🌎 PC + 4

Datapath for R and I-Type ALU Insts.

אנימציה



המוקס נועד גם לתמוך בהוראת R וגם

לאפשר לרגיסטר 2 להיות היעד לכתיבה

בהוראת I.

if MEM[PC] == ADDI rt rs immediate

GPR[rt] 🌍 GPR[rs] + sign-extend (immediate)

PC 🌍 PC + 4

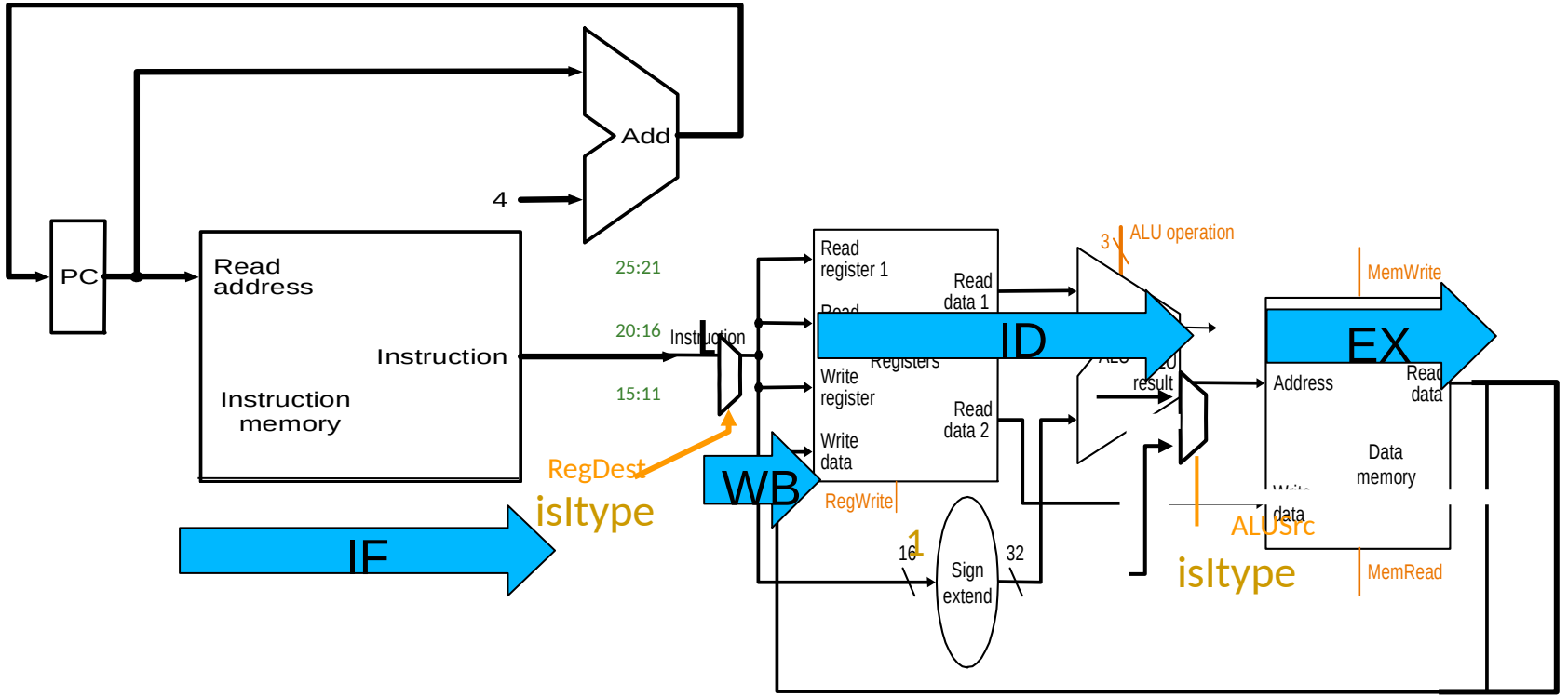
IF	ID	EX	MEM	WB

Combinational

state update logic

Datapath for R and I-Type ALU Insts.

אנימציה



GPR[rt] 🌎 GPR[rs] + sign-extend (immediate)

PC 🌎 PC + 4

Combinational state update logic

Single-Cycle Datapath for *Data Movement Instructions*

Load Instructions

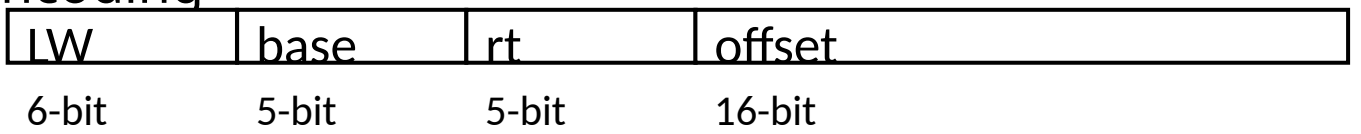
- Assembly (e.g., load 4-byte word)

  LW rt_{reg} $offset_{16}$ ($base_{reg}$)

כמו rs

Load
הוא ממשפחת ה- I-type

- Machine encoding



I-type

- Semantics

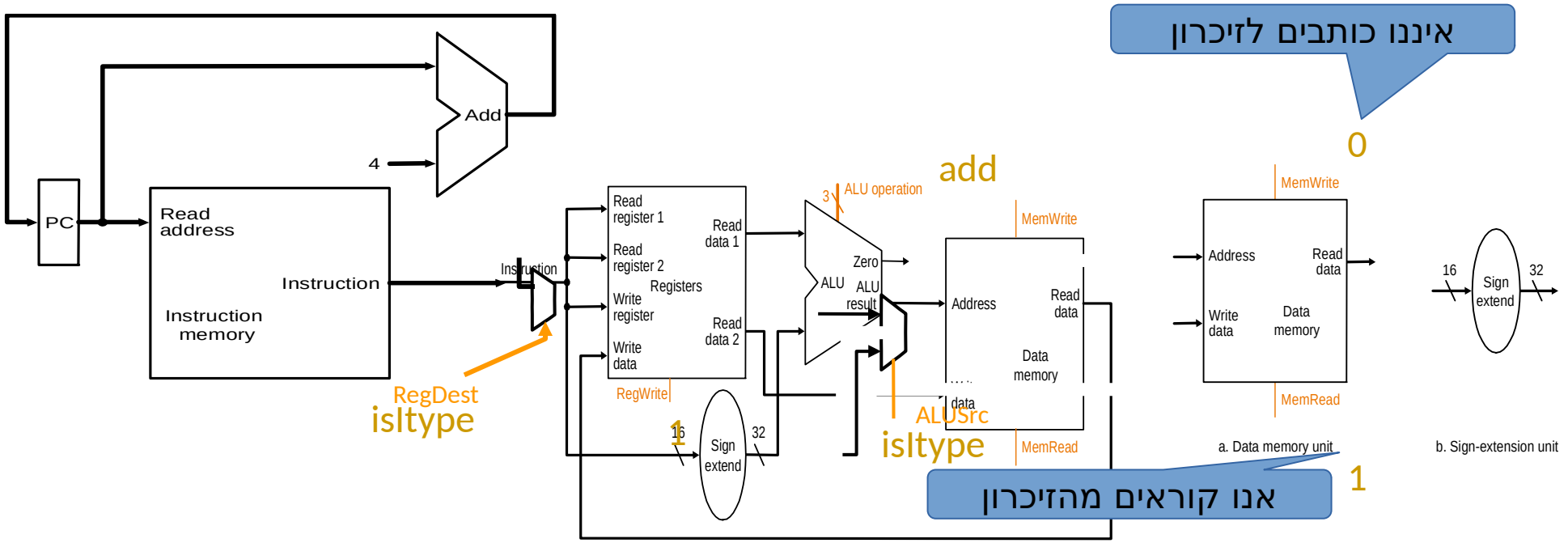
  if MEM[PC]==LW rt $offset_{16}$ (base)

  EA = sign-extend(offset) + GPR[base]

  GPR[rt] 🌍 MEM[translate(EA)]

  PC 🌍 PC + 4

LW Datapath



if MEM[PC]==LW rt offset$_{16}$ (base)

EA = sign-extend(offset) + GPR[base]

GPR[rt] 🌍 MEM[translate(EA)]

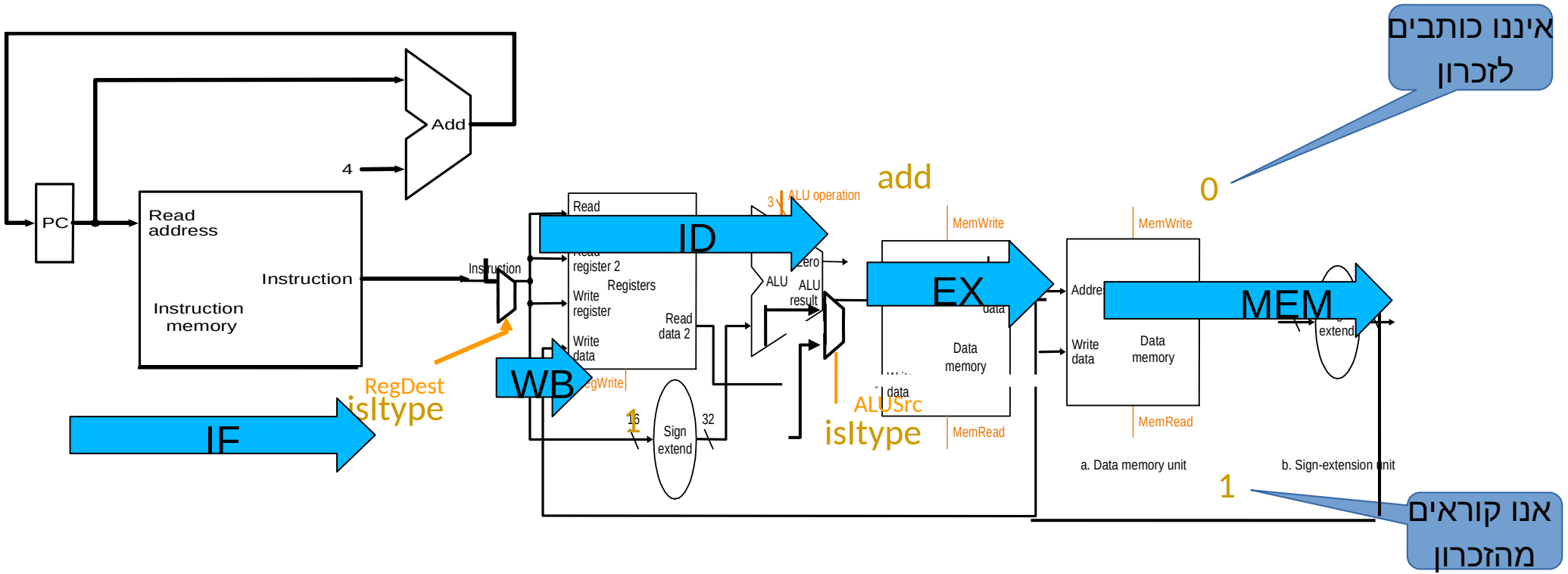
PC 🌍 PC + 4

IF	ID	EX	MEM	WB

Combinational state update logic

LW Datapath

אנימציה



EA = GPR[rs] + sign-extend(offset)

GPR[rt] 🌍 MEM[EA]

PC 🌍 PC + 4

IF	ID	EX	MEM	WB

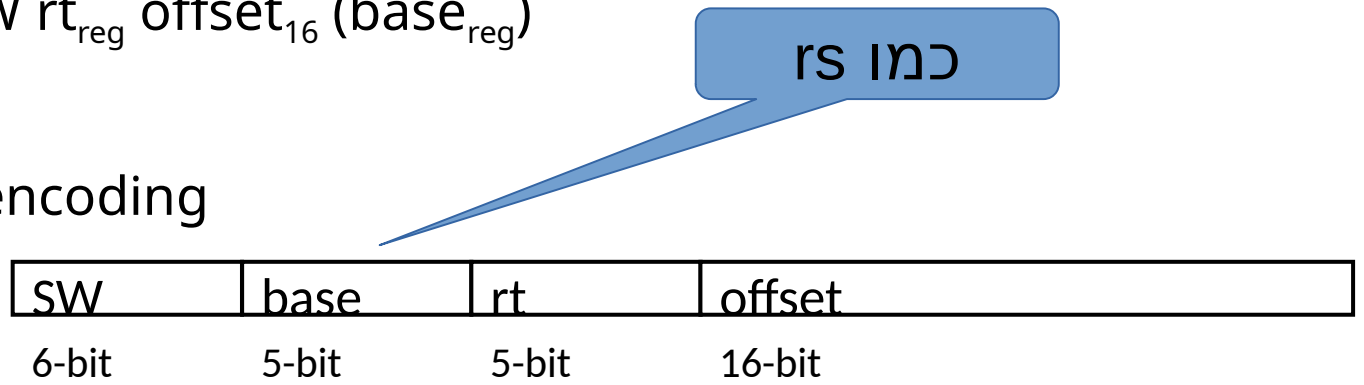
Combinational state update logic

Store Instructions

- Assembly (e.g., store 4-byte word)

  SW rt_{reg} $offset_{16}$ ($base_{reg}$)

- Machine encoding



- Semantics

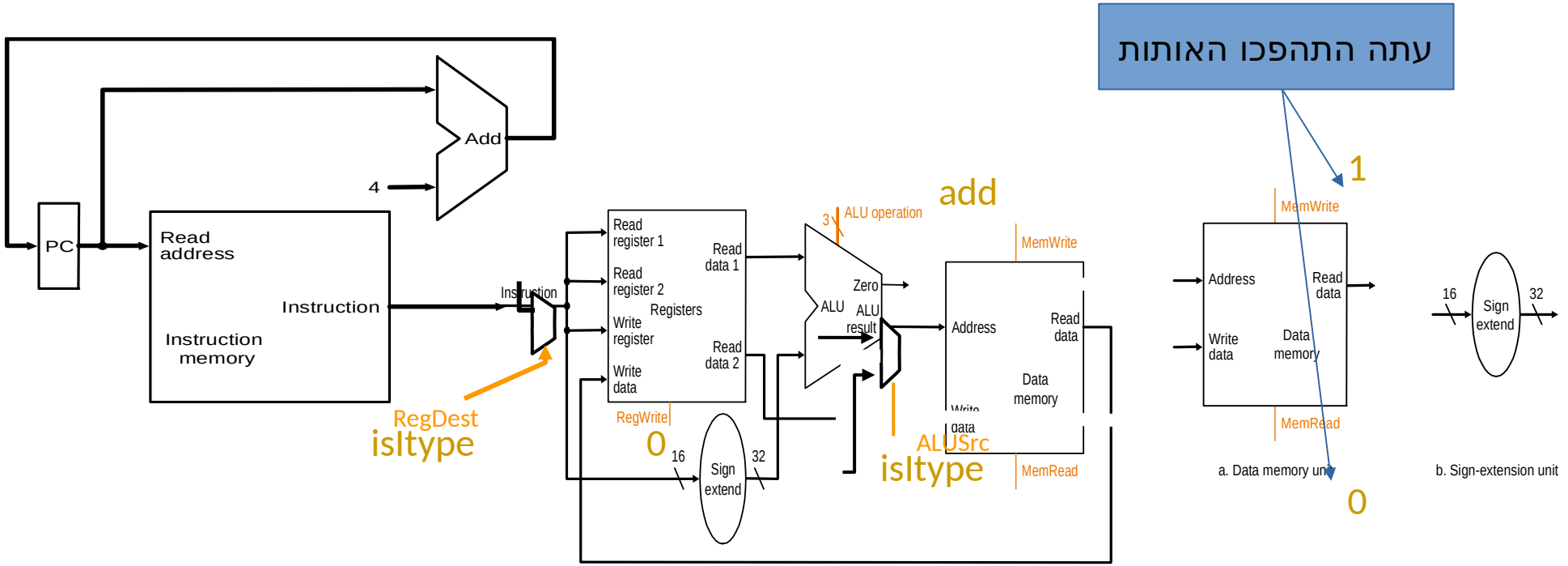
  if MEM[PC]==SW rt $offset_{16}$ (base)

  EA = sign-extend(offset) + GPR[base]

  MEM[translate(EA)] ← GPR[rt]

  PC ← PC + 4

SW Datapath



if MEM[PC]==SW rt $offset_{16}$ (base)

EA = sign-extend(offset) + GPR[base]

MEM[translate(EA)] 🌍 GPR[rt]

PC 🌍 PC + 4

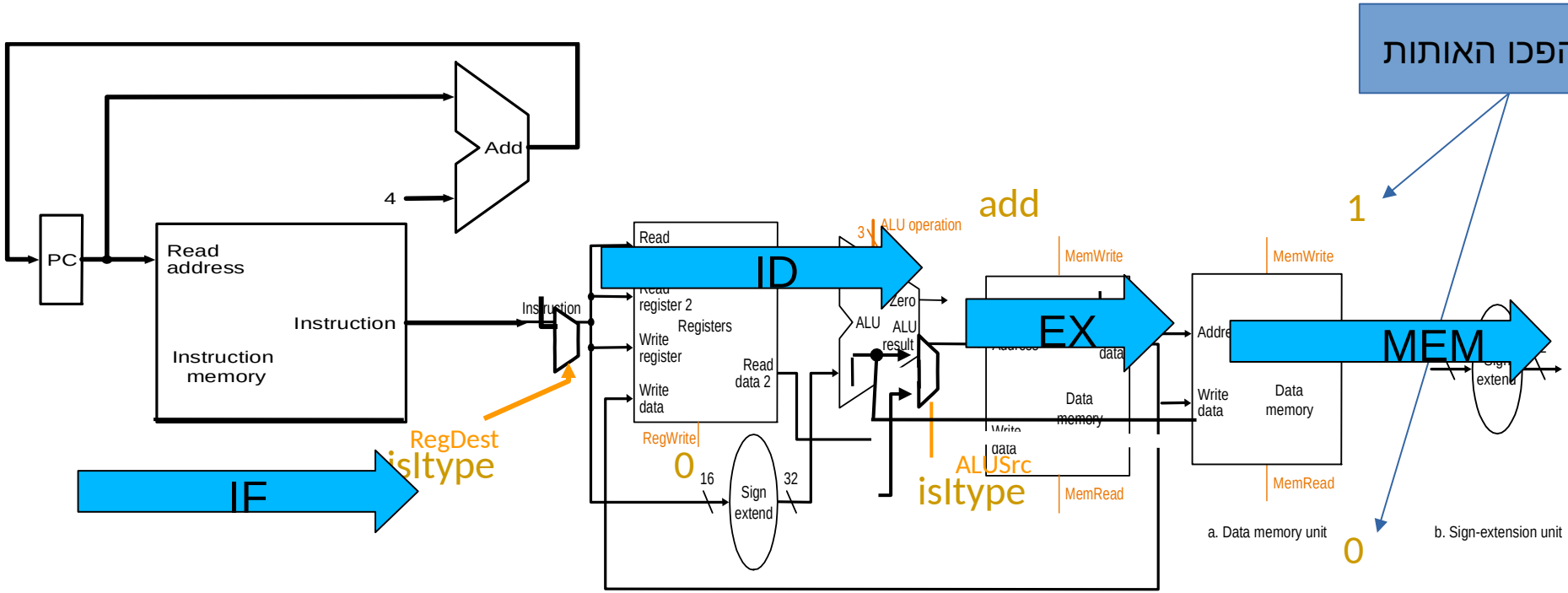
IF | ID | EX | MEM | WB

Combinational state update logic

SW Datapath

אנימציה

עתה התהפכו האותות



EA = GPR[rs] + sign-extend(offset)

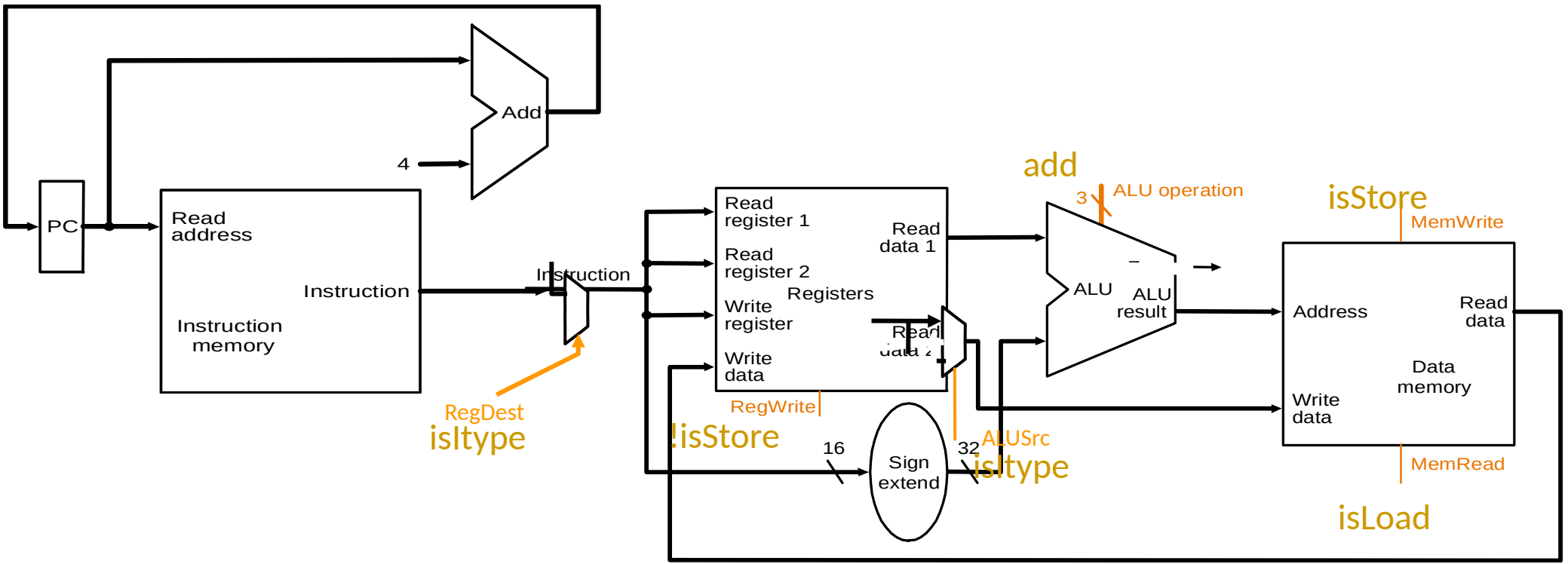
MEM[EA] 🌍 GPR[rt]

PC 🌍 PC + 4

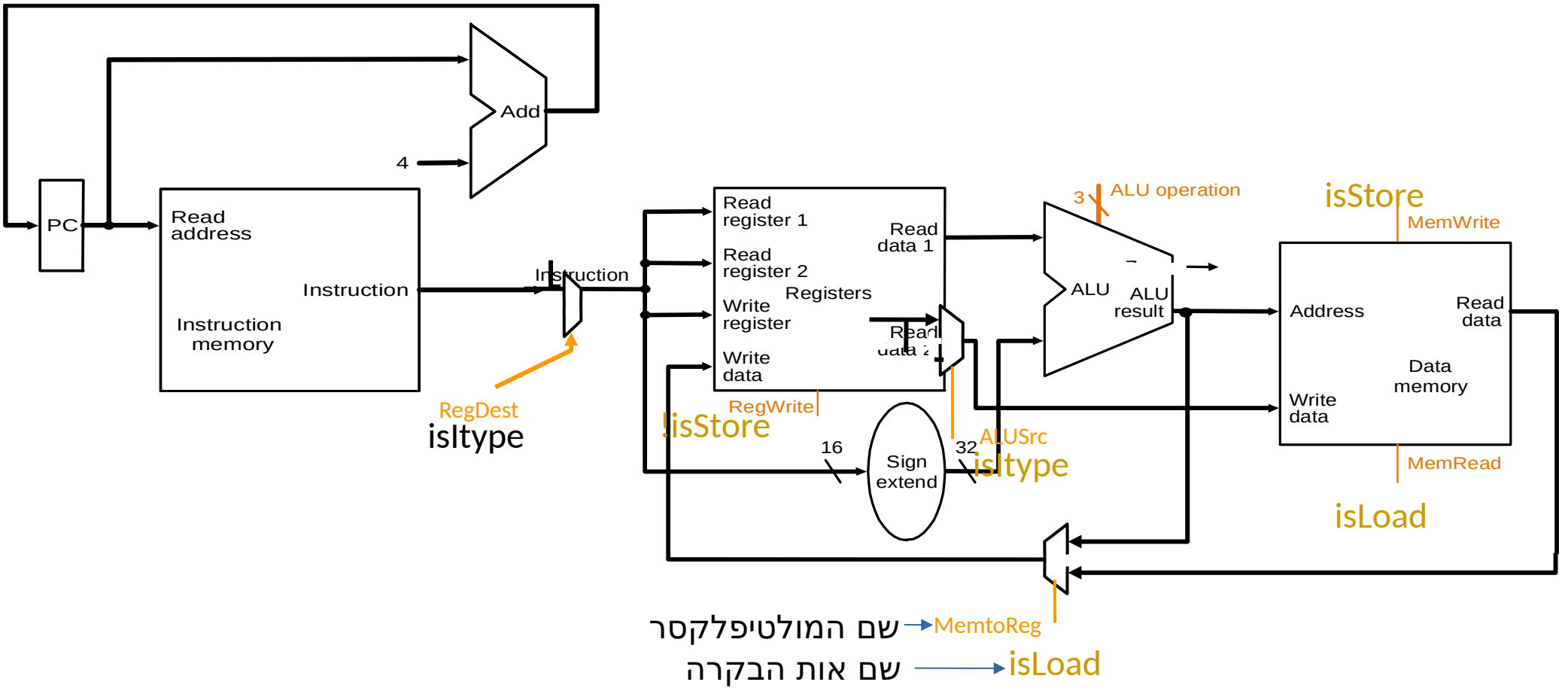
IF | ID | EX | MEM | ~~WB~~

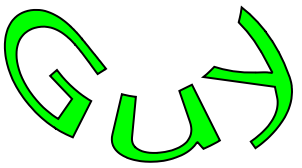
Combinational state update logic

Load-Store Datapath



Datapath for Non-Control-Flow Instructions





חזרה והסבר נוסף על ה- datapath עבור ההוראות שנלמדו עד-כה
ניתנת מחדש ב-7 השקפים הבאים הלקוחים מ-

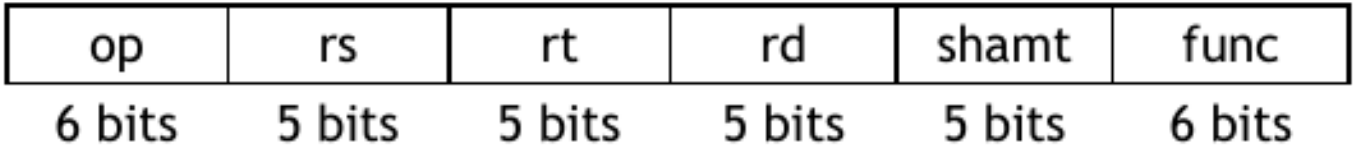
CSE378 University of Washington

//courses.cs.washington.edu/courses/cse378/10sp/lec
html

-במידה ולא נתעכב עליהם כאן והמעבר על השקפים יעשה בלימוד
עצמי בבית-

Encoding R-type instructions

- ~~Last lecture, we saw encodings of MIPS instructions as 32-bit values.~~
- Register-to-register arithmetic instructions use the R-type format.
  - op is the instruction opcode, and func specifies a particular arithmetic operation (see textbook).
  - rs, rt and rd are source and destination registers.

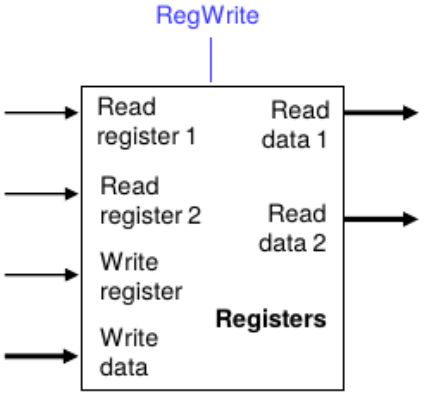


- An example instruction and its encoding:

add $s4, $t1, $t2

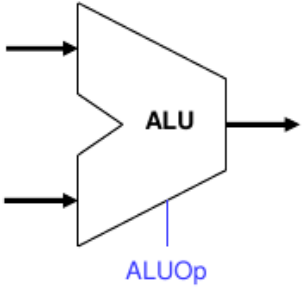
Registers and ALUs

- R-type instructions must access registers and an ALU.



- Our register file stores thirty-two 32-bit values.
  - Each register specifier is 5 bits long.
  - You can read from two registers at a time.
  - RegWrite is 1 if a register should be written.

- Here's a simple ALU with five operations, selected by a 3-bit control signal ALUOp.

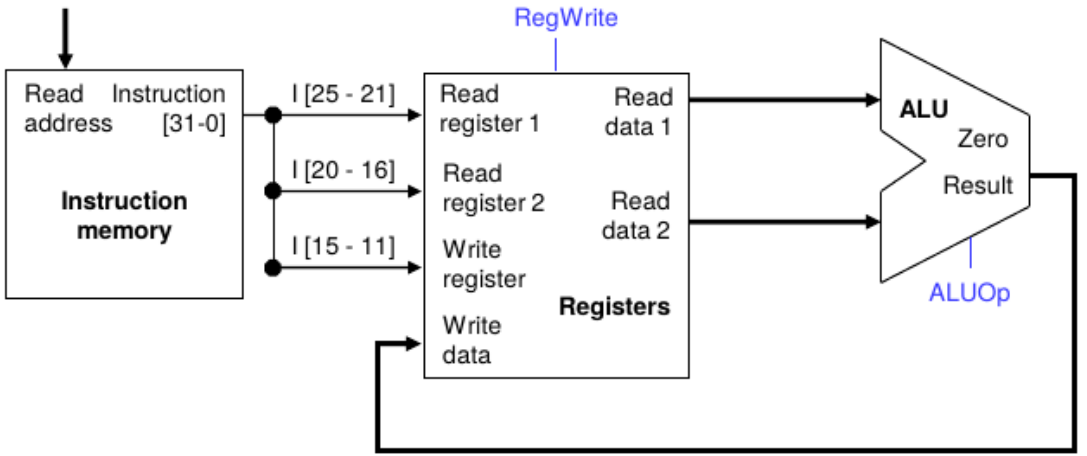


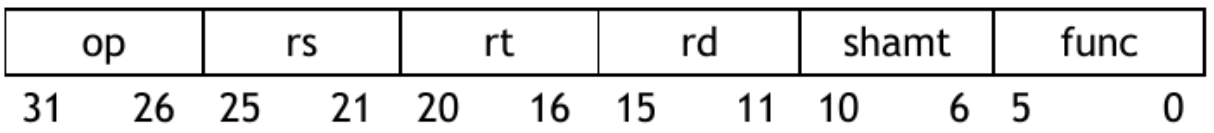
להמחשה בלבד

ALUOp	Function
000	and
001	or
010	add
110	subtract
111	slt

Executing an R-type instruction

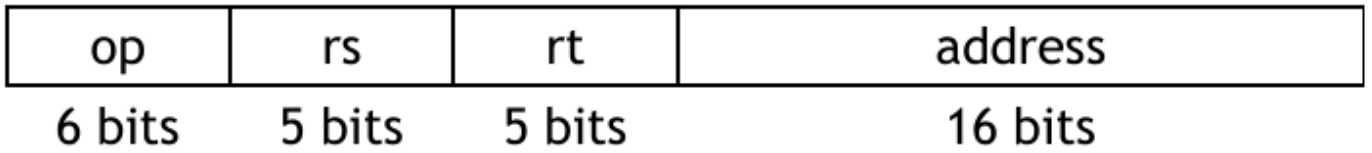
1. Read an instruction from the instruction memory.
2. The source registers, specified by instruction fields rs and rt, should be read from the register file.
3. The ALU performs the desired operation.
4. Its result is stored in the destination register, which is specified by field rd of the instruction word.



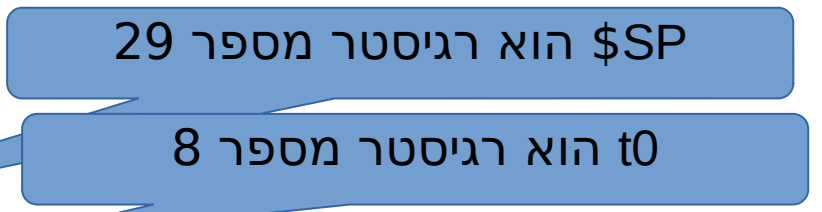


Encoding I-type instructions

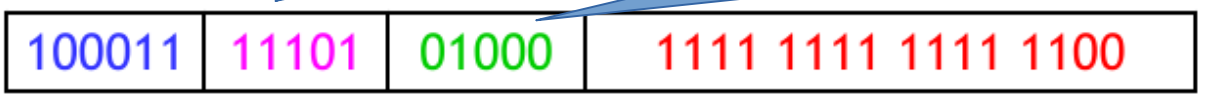
- The lw, sw and beq instructions all use the I-type encoding.
  - rt is the *destination* for lw, but a *source* for beq and sw.
  - address is a 16-bit signed constant.



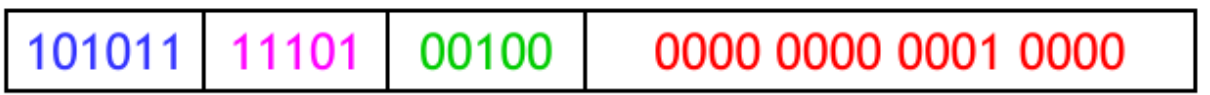
- Two example instructions:



lw $t0, -4($sp)

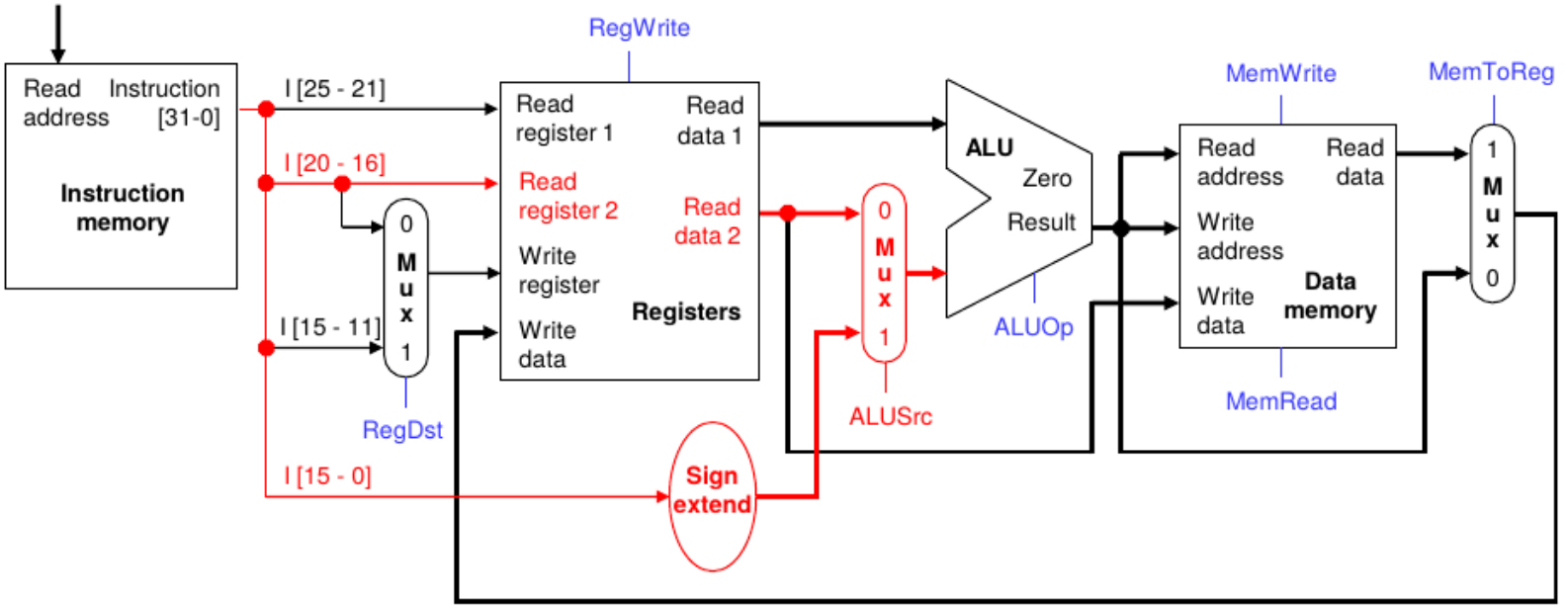


sw $a0, 16($sp)



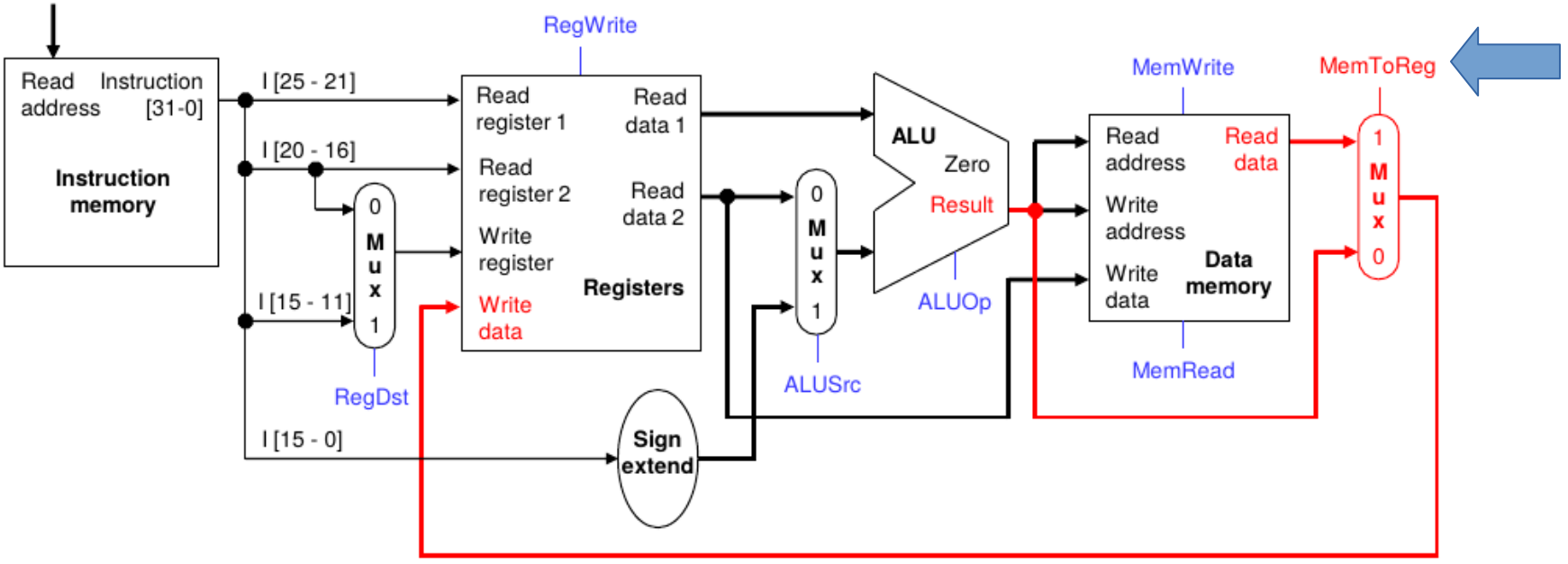
Accessing data memory

- For an instruction like lw $t0, -4($sp), the base register $sp is added to the *sign-extended* constant to get a data memory address.
- This means the ALU must accept *either* a register operand for arithmetic instructions, *or* a sign-extended immediate operand for lw and sw.
- We'll add a multiplexer, controlled by ALUSrc, to select either a register operand (0) or a constant operand (1).



MemToReg

- The register file's "Write data" input has a similar problem. It must be able to store *either* the ALU output of R-type instructions, *or* the data memory output for lw.
- We add a mux, controlled by MemToReg, to select between saving the ALU result (0) or the data memory output (1) to the registers.



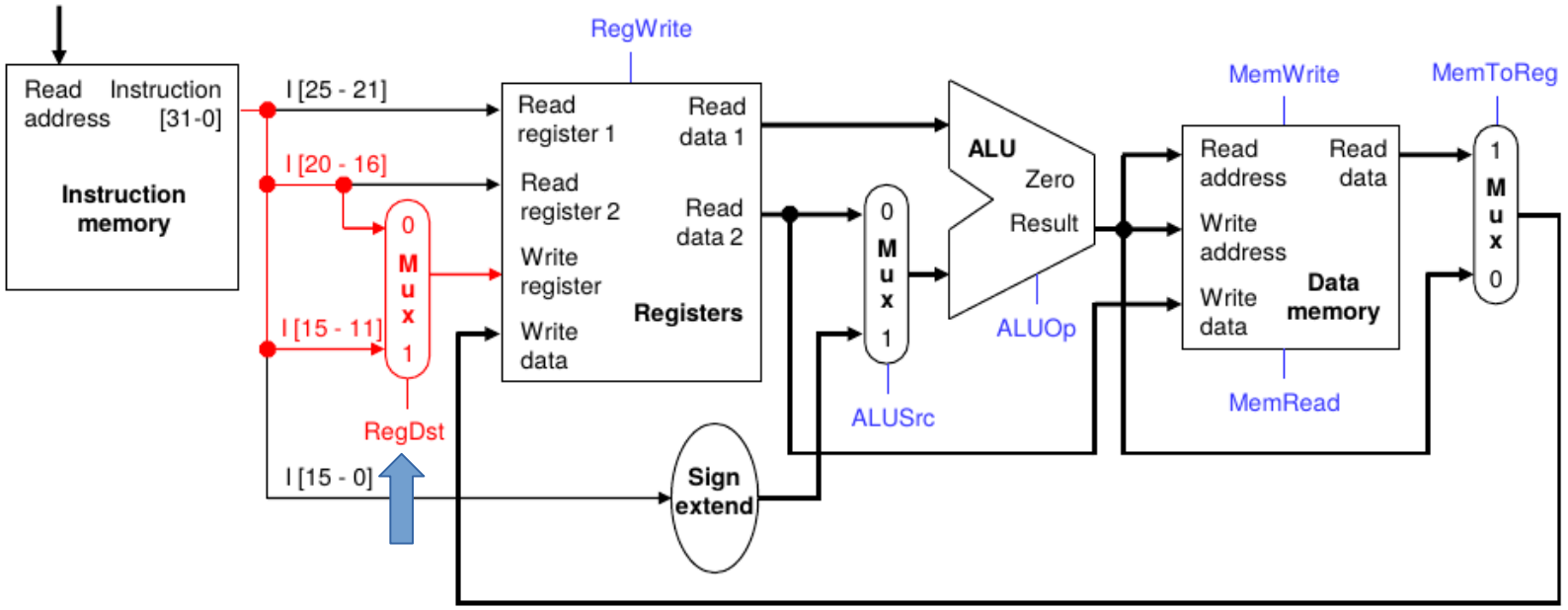
RegDst

- A final annoyance is the destination register of lw is in *rt* instead of rd.

op	rs	rt	address

lw $rt, address($rs)

- We'll add one more mux, controlled by RegDst, to select the destination register from either instruction field rt (0) or field rd (1).



עד כאן תוספת שקפים מתוך CSE378

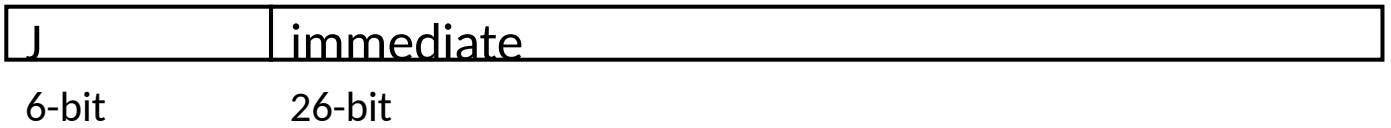
Single-Cycle Datapath for *Control Flow Instructions*

Unconditional Jump Instructions (MIPS)

- Assembly

  J immediate$_{26}$

- Machine encoding



J-type

- Semantics

  if MEM[PC]==J immediate$_{26}$

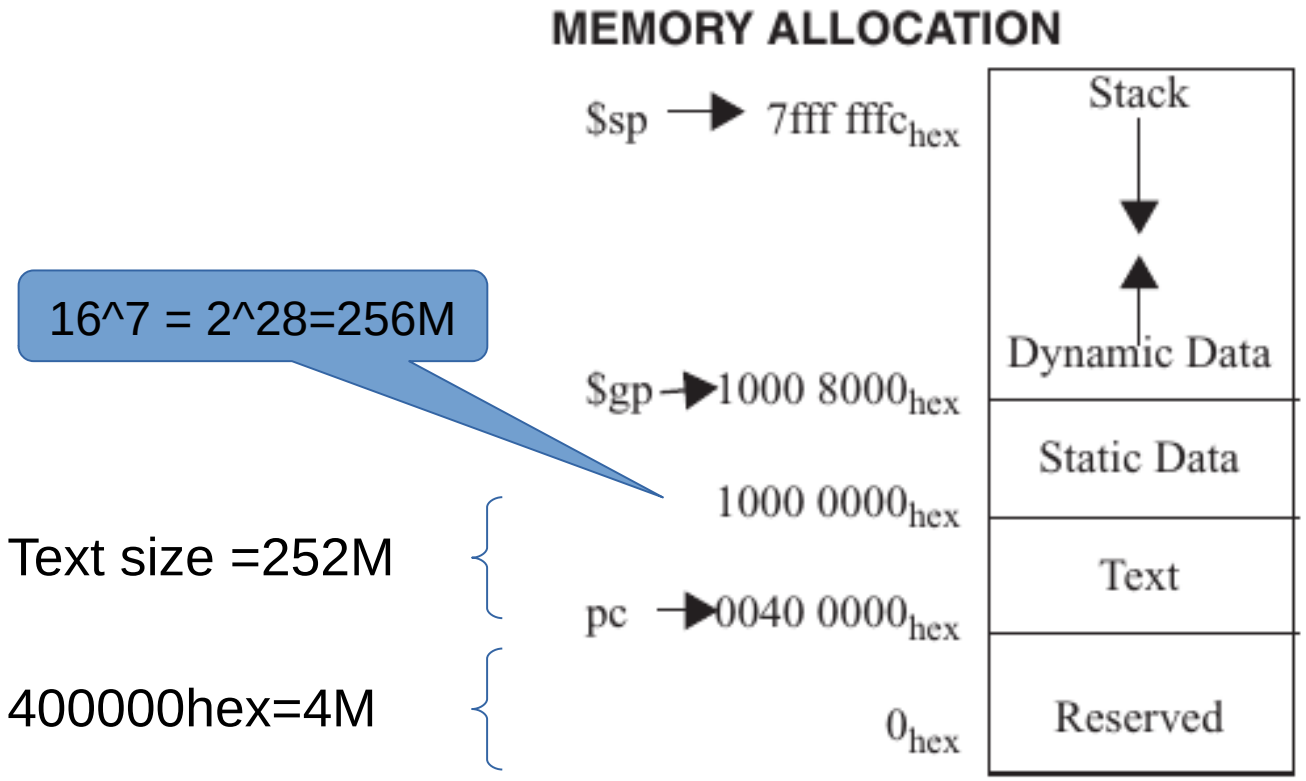
  target = { PC[31:28], immediate$_{26}$, 2'b00 }

  PC ← target

Guy> This is the addressing mode convention: [concatenate the PC+imm+2'b00]

4 ביטים עליונים == הספרה השמאלית של ה- PC בבסיס 16

תזכורת מפת זיכרון ה- MIPS(*)



* מפת הזיכרון עבור RISC-V הוצגה בשיעור 3 (פרק 6 H&H, וגם היא לימודית ולא מחייבת)

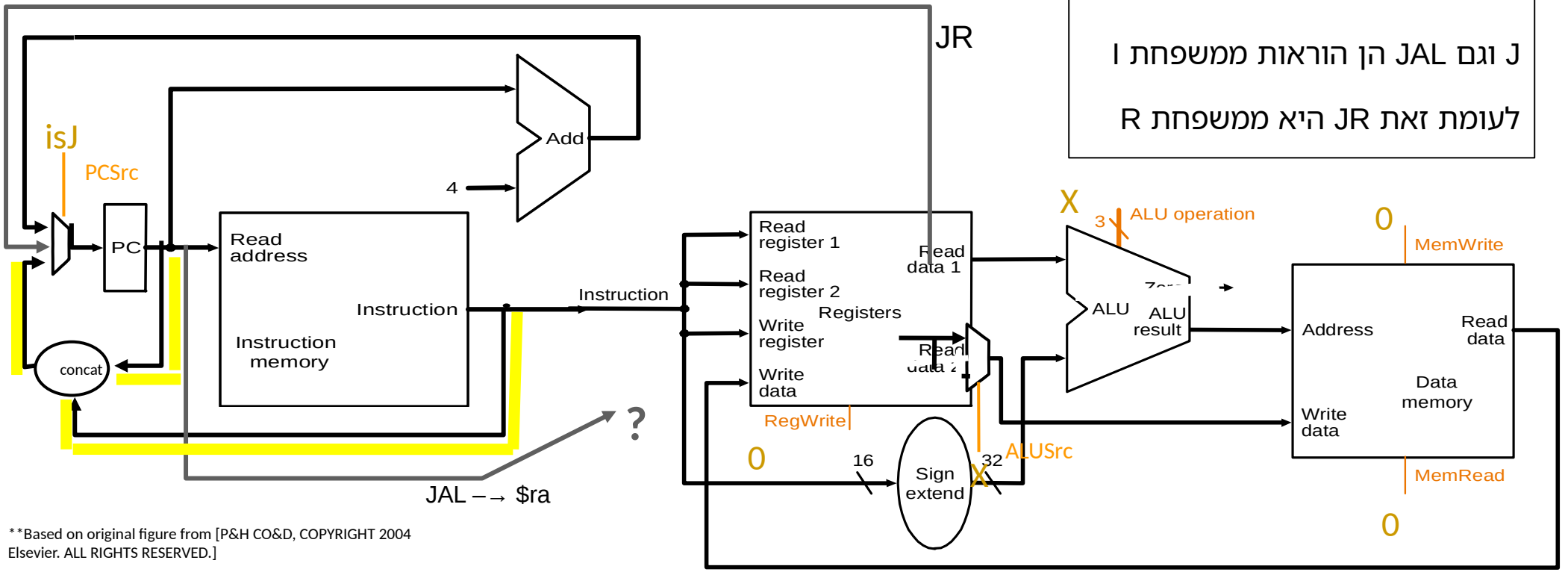
Unconditional Jump Datapath (MIPS)

אנימציה

שימו לב:

J וגם JAL הן הוראות ממשפחת I

לעומת זאת JR היא ממשפחת R



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

```
if MEM[PC]==J immediate26
    PC = { PC[31:28], immediate26, 2'b00 }
```

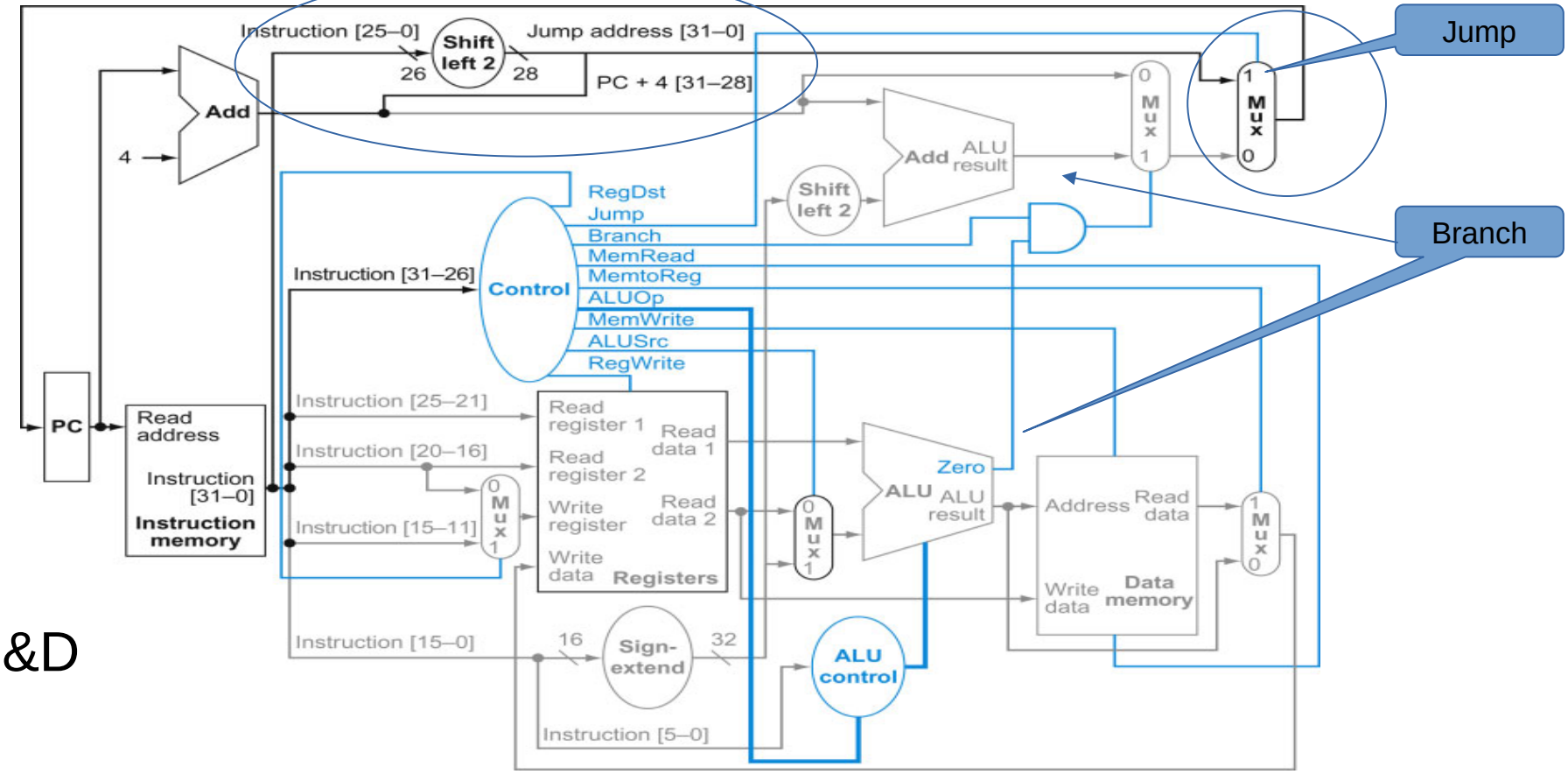
What about JR, JAL?

הערה לעצמי:

התרשים מופיע

גם בשקף הבא,

עליו ניתן לצייר



H&P CO&D

FIGURE 4.24 The simple control and datapath are extended to handle the **jump** instruction. An additional multiplexor (at the upper right) is used to choose between the jump target and either the branch target or the sequential instruction following this one. This multiplexor is controlled by the jump control signal. The jump target address is obtained by shifting the lower 26 bits of the jump instruction left 2 bits, effectively adding 00 as the low-order bits, and then concatenating the upper 4 bits of PC + 4 as the high-order bits, thus yielding a 32-bit address.

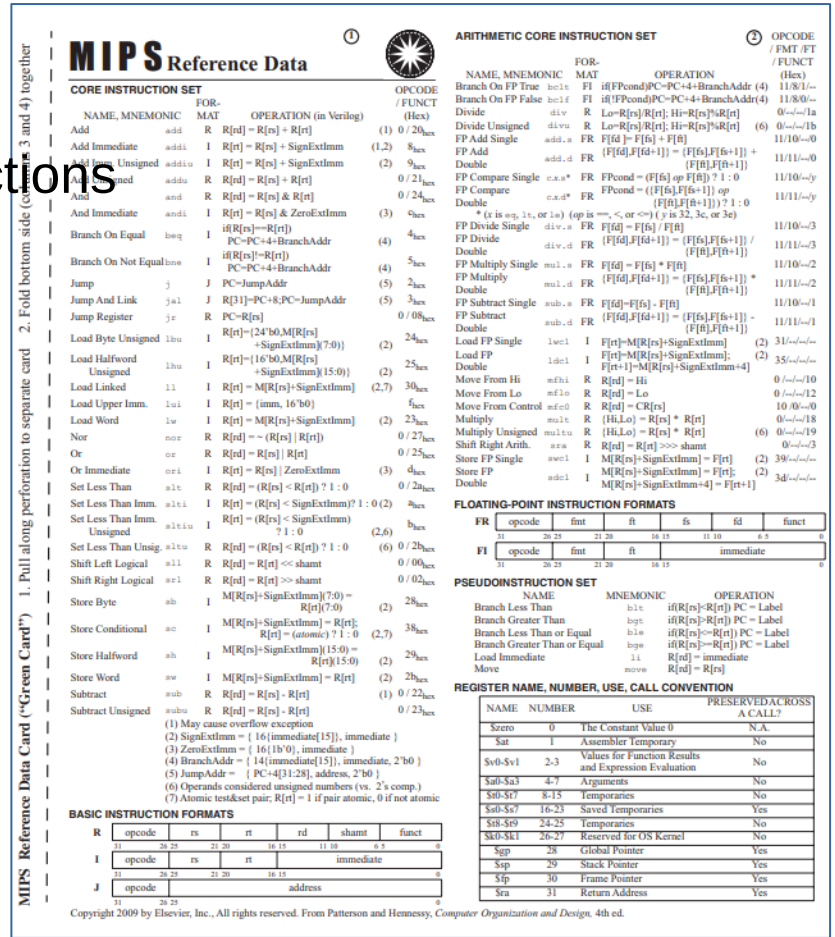
Aside: MIPS & RISC-V Cheat Sheets

נמצאים במודל

http://www.ece.cmu.edu/~ece447/s15/lib/exe/fetch.php?media=mips_reference_data.pdf

Pseudo Instructions

- abs
- blt
- bgt
- ble
- neg
- not
- bge
- li
- la
- move
- sge
- sgt



MIPS Reference Data

1. Pull along perforation to separate card 2. Fold bottom side (column 3 and 4) together

MIPS Reference Data Card ("Green Card")

CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Add	add	R	R[rd] = R[rs] + R[rt]	(1)	0 / 20_{hex}
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm	(1,2)	8_{hex}
Add Imm. Unsigned	addiu	I	R[rt] = R[rs] + SignExtImm	(2)	9_{hex}
Add Unsigned	addu	R	R[rd] = R[rs] + R[rt]		0 / 21_{hex}
And	and	R	R[rd] = R[rs] & R[rt]		0 / 24_{hex}
And Immediate	andi	I	R[rt] = R[rs] & ZeroExtImm	(3)	c_{hex}
Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4_{hex}
Branch On Not Equal	bne	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5_{hex}
Jump	j	J	PC=JumpAddr	(5)	2_{hex}
Jump And Link	jal	J	R[31]=PC+8;PC=JumpAddr	(5)	3_{hex}
Jump Register	jr	R	PC=R[rs]		0 / 08_{hex}
Load Byte Unsigned	lbu	I	R[rt]={24'b0,M[R[rs]+SignExtImm](7:0)}	(2)	24_{hex}
Load Halfword Unsigned	lhu	I	R[rt]={16'b0,M[R[rs]+SignExtImm](15:0)}	(2)	25_{hex}
Load Linked	ll	I	R[rt] = M[R[rs]+SignExtImm]	(2,7)	30_{hex}
Load Upper Imm.	lui	I	R[rt] = {imm, 16'b0}		f_{hex}
Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}
Nor	nor	R	R[rd] = ~ (R[rs] | R[rt])		0 / 27_{hex}
Or	or	R	R[rd] = R[rs] | R[rt]		0 / 25_{hex}
Or Immediate	ori	I	R[rt] = R[rs] | ZeroExtImm	(3)	d_{hex}
Set Less Than	slt	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0		0 / $2a_{hex}$
Set Less Than Imm.	slti	I	R[rt] = (R[rs] < SignExtImm)? 1 : 0	(2)	a_{hex}
Set Less Than Imm. Unsigned	sltiu	I	R[rt] = (R[rs] < SignExtImm) ? 1 : 0	(2,6)	b_{hex}
Set Less Than Unsig.	sltu	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0	(6)	0 / $2b_{hex}$
Shift Left Logical	sll	R	R[rd] = R[rt] << shamt		0 / 00_{hex}
Shift Right Logical	srl	R	R[rd] = R[rt] >> shamt		0 / 02_{hex}
Store Byte	sb	I	M[R[rs]+SignExtImm](7:0) = R[rt](7:0)	(2)	28_{hex}
Store Conditional	sc	I	M[R[rs]+SignExtImm] = R[rt]; R[rt] = (*atomic*) ? 1 : 0	(2,7)	38_{hex}
Store Halfword	sh	I	M[R[rs]+SignExtImm](15:0) = R[rt](15:0)	(2)	29_{hex}
Store Word	sw	I	M[R[rs]+SignExtImm] = R[rt]	(2)	$2b_{hex}$
Subtract	sub	R	R[rd] = R[rs] - R[rt]	(1)	0 / 22_{hex}
Subtract Unsigned	subu	R	R[rd] = R[rs] - R[rt]		0 / 23_{hex}

(1) May cause overflow exception
(2) SignExtImm = { 16{immediate[15]}, immediate }
(3) ZeroExtImm = { 16{1b'0}, immediate }
(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }
(5) JumpAddr = { PC+4[31:28], address, 2'b0 }
(6) Operands considered unsigned numbers (vs. 2's comp.)
(7) Atomic test&set pair; R[rt] = 1 if pair atomic, 0 if not atomic

BASIC INSTRUCTION FORMATS

ARITHMETIC CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION		OPCODE / FMT /FT / FUNCT (Hex)
Branch On FP True	bc1t	FI	if(FPcond)PC=PC+4+BranchAddr (4)		11/8/1/--
Branch On FP False	bc1f	FI	if(!FPcond)PC=PC+4+BranchAddr(4)		11/8/0/--
Divide	div	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]		0/--/--/1a
Divide Unsigned	divu	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]	(6)	0/--/--/1b
FP Add Single	add.s	FR	F[fd]= F[fs] + F[ft]		11/10/--/0
FP Add Double	add.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} + {F[ft],F[ft+1]}		11/11/--/0
FP Compare Single	c.x.s*	FR	FPcond = (F[fs] *op* F[ft]) ? 1 : 0		11/10/--/y
FP Compare Double	c.x.d*	FR	FPcond = ({F[fs],F[fs+1]} *op* {F[ft],F[ft+1]}) ? 1 : 0		11/11/--/y
* (*x* is eq, lt, or le) (*op* is ==, <, or <=) (*y* is 32, 3c, or 3e)					
FP Divide Single	div.s	FR	F[fd] = F[fs] / F[ft]		11/10/--/3
FP Divide Double	div.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} / {F[ft],F[ft+1]}		11/11/--/3
FP Multiply Single	mul.s	FR	F[fd] = F[fs] * F[ft]		11/10/--/2
FP Multiply Double	mul.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} * {F[ft],F[ft+1]}		11/11/--/2
FP Subtract Single	sub.s	FR	F[fd]=F[fs] - F[ft]		11/10/--/1
FP Subtract Double	sub.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} - {F[ft],F[ft+1]}		11/11/--/1
Load FP Single	lwc1	I	F[rt]=M[R[rs]+SignExtImm]	(2)	31/--/--/--
Load FP Double	ldc1	I	F[rt]=M[R[rs]+SignExtImm]; F[rt+1]=M[R[rs]+SignExtImm+4]	(2)	35/--/--/--
Move From Hi	mfhi	R	R[rd] = Hi		0 /--/--/10
Move From Lo	mflo	R	R[rd] = Lo		0 /--/--/12
Move From Control	mfc0	R	R[rd] = CR[rs]		10 /0/--/0
Multiply	mult	R	{Hi,Lo} = R[rs] * R[rt]		0/--/--/18
Multiply Unsigned	multu	R	{Hi,Lo} = R[rs] * R[rt]	(6)	0/--/--/19
Shift Right Arith.	sra	R	R[rd] = R[rt] >>> shamt		0/--/--/3
Store FP Single	swc1	I	M[R[rs]+SignExtImm] = F[rt]	(2)	39/--/--/--
Store FP Double	sdc1	I	M[R[rs]+SignExtImm] = F[rt]; M[R[rs]+SignExtImm+4] = F[rt+1]	(2)	3d/--/--/--

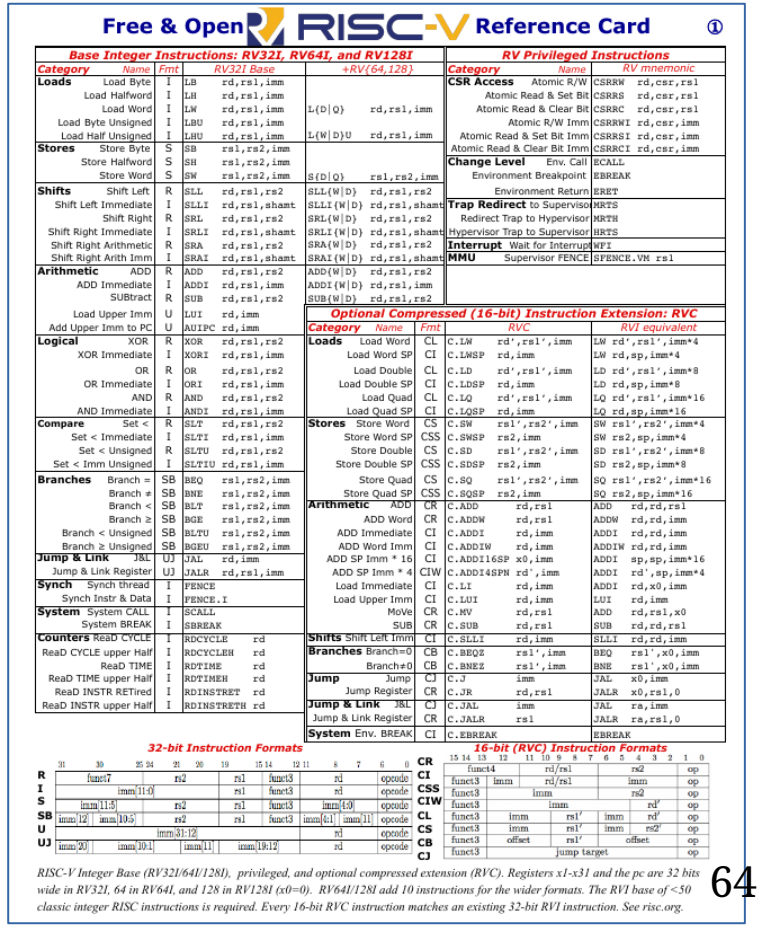
FLOATING-POINT INSTRUCTION FORMATS

PSEUDOINSTRUCTION SET

NAME	MNEMONIC	OPERATION
Branch Less Than	blt	if(R[rs]<R[rt]) PC = Label
Branch Greater Than	bgt	if(R[rs]>R[rt]) PC = Label
Branch Less Than or Equal	ble	if(R[rs]<=R[rt]) PC = Label
Branch Greater Than or Equal	bge	if(R[rs]>=R[rt]) PC = Label
Load Immediate	li	R[rd] = immediate
Move	move	R[rd] = R[rs]

REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	Yes



Free & Open RISC-V Reference Card

Base Integer Instructions: RV32I, RV64I, and RV128I

Category	Name	Fmt	RV32I Base	+RV{64,128}
Loads	Load Byte	I	LB rd,rs1,imm	
	Load Halfword	I	LH rd,rs1,imm	
	Load Word	I	LW rd,rs1,imm	L{D|Q} rd,rs1,imm
	Load Byte Unsigned	I	LBU rd,rs1,imm	
	Load Half Unsigned	I	LHU rd,rs1,imm	L{W|D}U rd,rs1,imm
Stores	Store Byte	S	SB rs1,rs2,imm	
	Store Halfword	S	SH rs1,rs2,imm	
	Store Word	S	SW rs1,rs2,imm	S{D|Q} rs1,rs2,imm
Shifts	Shift Left	R	SLL rd,rs1,rs2	SLL{W|D} rd,rs1,rs2
	Shift Left Immediate	I	SLLI rd,rs1,shamt	SLLI{W|D} rd,rs1,shamt
	Shift Right	R	SRL rd,rs1,rs2	SRL{W|D} rd,rs1,rs2
	Shift Right Immediate	I	SRLI rd,rs1,shamt	SRLI{W|D} rd,rs1,shamt
	Shift Right Arithmetic	R	SRA rd,rs1,rs2	SRA{W|D} rd,rs1,rs2
	Shift Right Arith Imm	I	SRAI rd,rs1,shamt	SRAI{W|D} rd,rs1,shamt
Arithmetic	ADD	R	ADD rd,rs1,rs2	ADD{W|D} rd,rs1,rs2
	ADD Immediate	I	ADDI rd,rs1,imm	ADDI{W|D} rd,rs1,imm
	SUBtract	R	SUB rd,rs1,rs2	SUB{W|D} rd,rs1,rs2
	Load Upper Imm	U	LUI rd,imm	
	Add Upper Imm to PC	U	AUIPC rd,imm	
Logical	XOR	R	XOR rd,rs1,rs2	
	XOR Immediate	I	XORI rd,rs1,imm	
	OR	R	OR rd,rs1,rs2	
	OR Immediate	I	ORI rd,rs1,imm	
	AND	R	AND rd,rs1,rs2	
	AND Immediate	I	ANDI rd,rs1,imm	
Compare	Set <	R	SLT rd,rs1,rs2	
	Set < Immediate	I	SLTI rd,rs1,imm	
	Set < Unsigned	R	SLTU rd,rs1,rs2	
	Set < Imm Unsigned	I	SLTIU rd,rs1,imm	
Branches	Branch =	SB	BEQ rs1,rs2,imm	
	Branch ≠	SB	BNE rs1,rs2,imm	
	Branch <	SB	BLT rs1,rs2,imm	
	Branch ≥	SB	BGE rs1,rs2,imm	
	Branch < Unsigned	SB	BLTU rs1,rs2,imm	
	Branch ≥ Unsigned	SB	BGEU rs1,rs2,imm	
Jump & Link	JAL	UJ	JAL rd,imm	
	Jump & Link Register	UJ	JALR rd,rs1,imm	
Synch	Synch thread	I	FENCE	
	Synch Instr & Data	I	FENCE.I	
System	System CALL	I	SCALL	
	System BREAK	I	SBREAK	
Counters	ReaD CYCLE	I	RDCYCLE rd	
	ReaD CYCLE upper Half	I	RDCYCLEH rd	
	ReaD TIME	I	RDTIME rd	
	ReaD TIME upper Half	I	RDTIMEH rd	
	ReaD INSTR RETired	I	RDINSTRET rd	
	ReaD INSTR upper Half	I	RDINSTRETH rd	

RV Privileged Instructions

Category	Name	RV mnemonic
CSR Access	Atomic R/W	CSRRW rd,csr,rs1
	Atomic Read & Set Bit	CSRRS rd,csr,rs1
	Atomic Read & Clear Bit	CSRRC rd,csr,rs1
	Atomic R/W Imm	CSRRWI rd,csr,imm
	Atomic Read & Set Bit Imm	CSRRSI rd,csr,imm
	Atomic Read & Clear Bit Imm	CSRRCI rd,csr,imm
Change Level	Env. Call	ECALL
	Environment Breakpoint	EBREAK
	Environment Return	ERET
Trap Redirect	to Supervisor	MRTS
	Redirect Trap to Hypervisor	MRTH
	Hypervisor Trap to Supervisor	HRTS
Interrupt	Wait for Interrupt	WFI
MMU	Supervisor FENCE	SFENCE.VM rs1

Optional Compressed (16-bit) Instruction Extension: RVC

Category	Name	Fmt	RVC	RVI equivalent
Loads	Load Word	CL	C.LW rd',rs1',imm*4	LW rd',rs1',imm*4
	Load Word SP	CI	C.LWSP rd,imm	LW rd,sp,imm*4
	Load Double	CL	C.LD rd',rs1',imm	LD rd',rs1',imm*8
	Load Double SP	CI	C.LDSP rd,imm	LD rd,sp,imm*8
	Load Quad	CL	C.LQ rd',rs1',imm	LQ rd',rs1',imm*16
	Load Quad SP	CI	C.LQSP rd,imm	LQ rd,sp,imm*16
Stores	Store Word	CS	C.SW rs1',rs2',imm	SW rs1',rs2',imm*4
	Store Word SP	CSS	C.SWSP rs2,imm	SW rs2,sp,imm*4
	Store Double	CS	C.SD rs1',rs2',imm	SD rs1',rs2',imm*8
	Store Double SP	CSS	C.SDSP rs2,imm	SD rs2,sp,imm*8
	Store Quad	CS	C.SQ rs1',rs2',imm	SQ rs1',rs2',imm*16
	Store Quad SP	CSS	C.SQSP rs2,imm	SQ rs2,sp,imm*16
Arithmetic	ADD	CR	C.ADD rd,rs1	ADD rd,rd,rs1
	ADD Word	CR	C.ADDW rd,rs1	ADDW rd,rd,rs1
	ADD Immediate	CI	C.ADDI rd,imm	ADDI rd,rd,imm
	ADD Word Imm	CI	C.ADDIW rd,imm	ADDIW rd,rd,imm
	ADD SP Imm * 16	CI	C.ADDI16SP x0,imm	ADDI sp,sp,imm*16
	ADD SP Imm * 4	CIW	C.ADDI4SPN rd',imm	ADDI rd',sp,imm*4
	Load Immediate	CI	C.LI rd,imm	ADDI rd,x0,imm
	Load Upper Imm	CI	C.LUI rd,imm	LUI rd,imm
	MoVe	CR	C.MV rd,rs1	ADD rd,rs1,x0
	SUB	CR	C.SUB rd,rs1	SUB rd,rd,rs1
Shifts	Shift Left Imm	CI	C.SLLI rd,imm	SLLI rd,rd,imm
Branches	Branch=0	CB	C.BEQZ rs1',imm	BEQ rs1',x0,imm
	Branch≠0	CB	C.BNEZ rs1',imm	BNE rs1',x0,imm
Jump	Jump	CJ	C.J imm	JAL x0,imm
	Jump Register	CR	C.JR rd,rs1	JALR x0,rs1,0
Jump & Link	JAL	CJ	C.JAL imm	JAL ra,imm
	Jump & Link Register	CR	C.JALR rs1	JALR ra,rs1,0
System	Env. BREAK	CI	C.EBREAK	EBREAK

32-bit Instruction Formats

16-bit (RVC) Instruction Formats

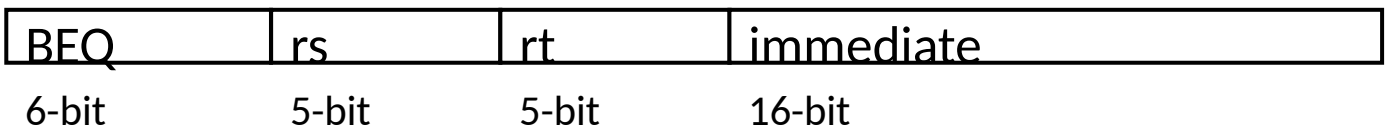
RISC-V Integer Base (RV32I/64I/128I), privileged, and optional compressed extension (RVC). Registers x1-x31 and the pc are 32 bits wide in RV32I, 64 in RV64I, and 128 in RV128I (x0=0). RV64I/128I add 10 instructions for the wider formats. The RVI base of <50 classic integer RISC instructions is required. Every 16-bit RVC instruction matches an existing 32-bit RVI instruction. See risc.org.

Conditional Branch Instructions (MIPS)

- Assembly (e.g., branch if equal)

  BEQ rs_{reg} rt_{reg} $immediate_{16}$

- Machine encoding



I-type

- Semantics (assuming no branch delay slot)

  if MEM[PC]==BEQ rs rt $immediate_{16}$

  target = PC + 4 + sign-extend(immediate) x 4

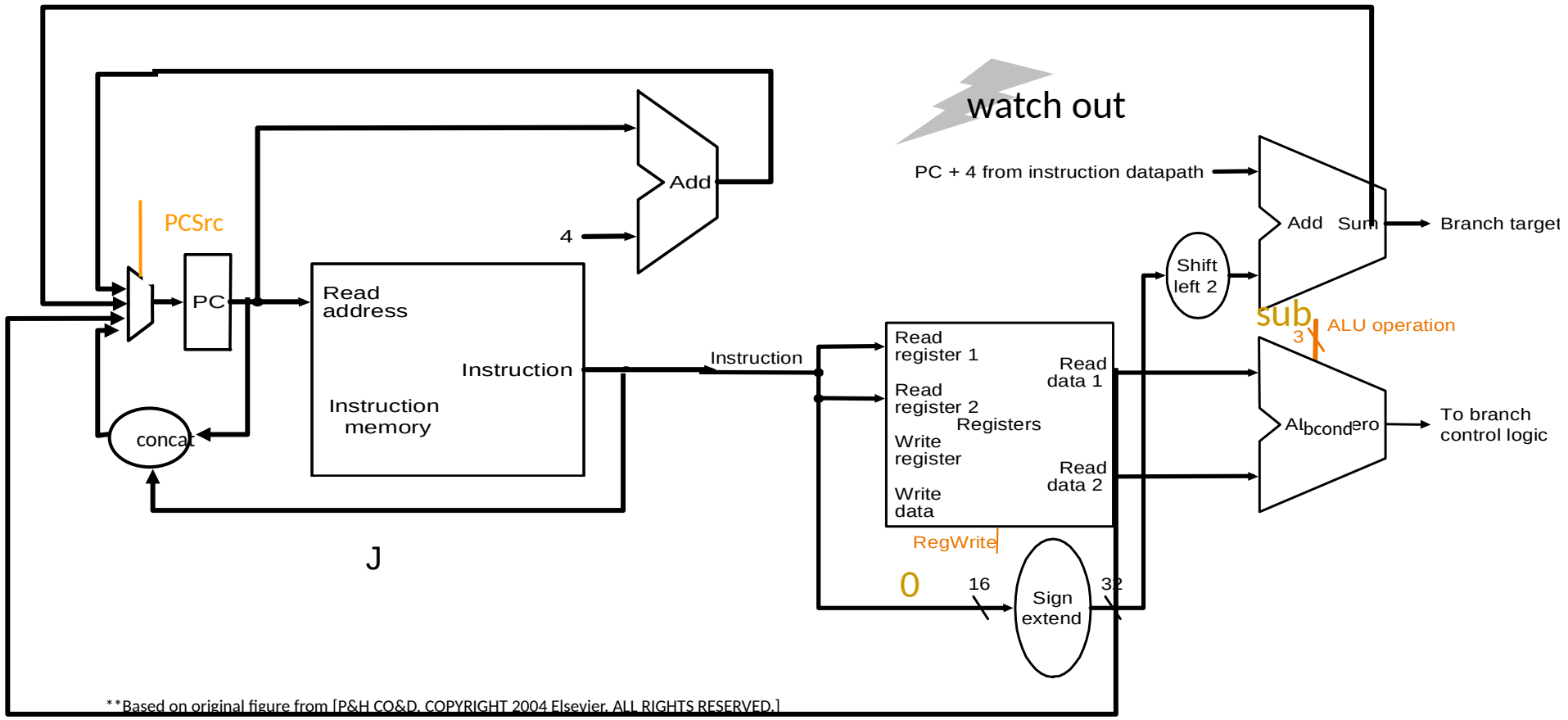
  if GPR[rs]==GPR[rt] then PC ← target

  else PC ← PC + 4

פעולת השוואה נעשית ב-ALU !

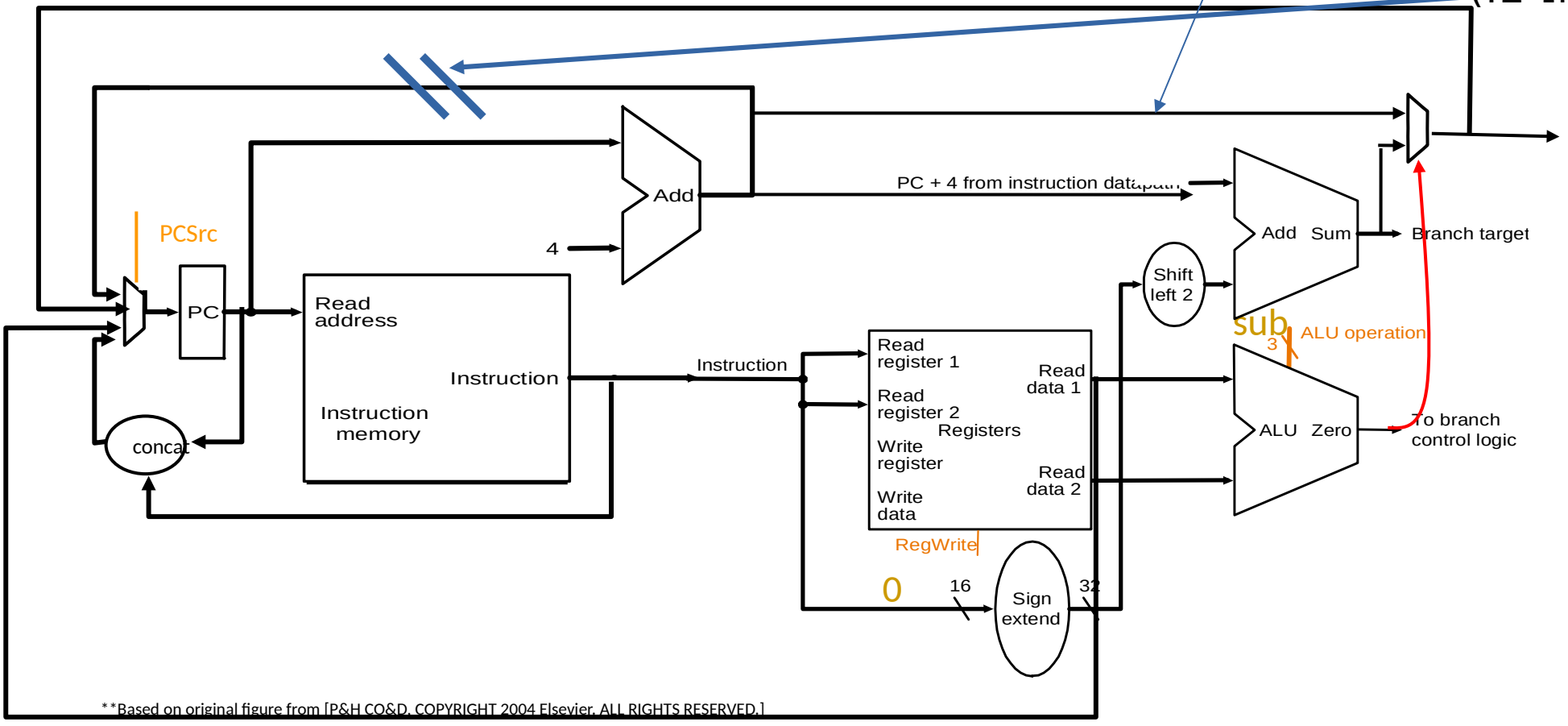
Conditional Branch Datapath (for you to finish)

פתרון בשקף הבא...

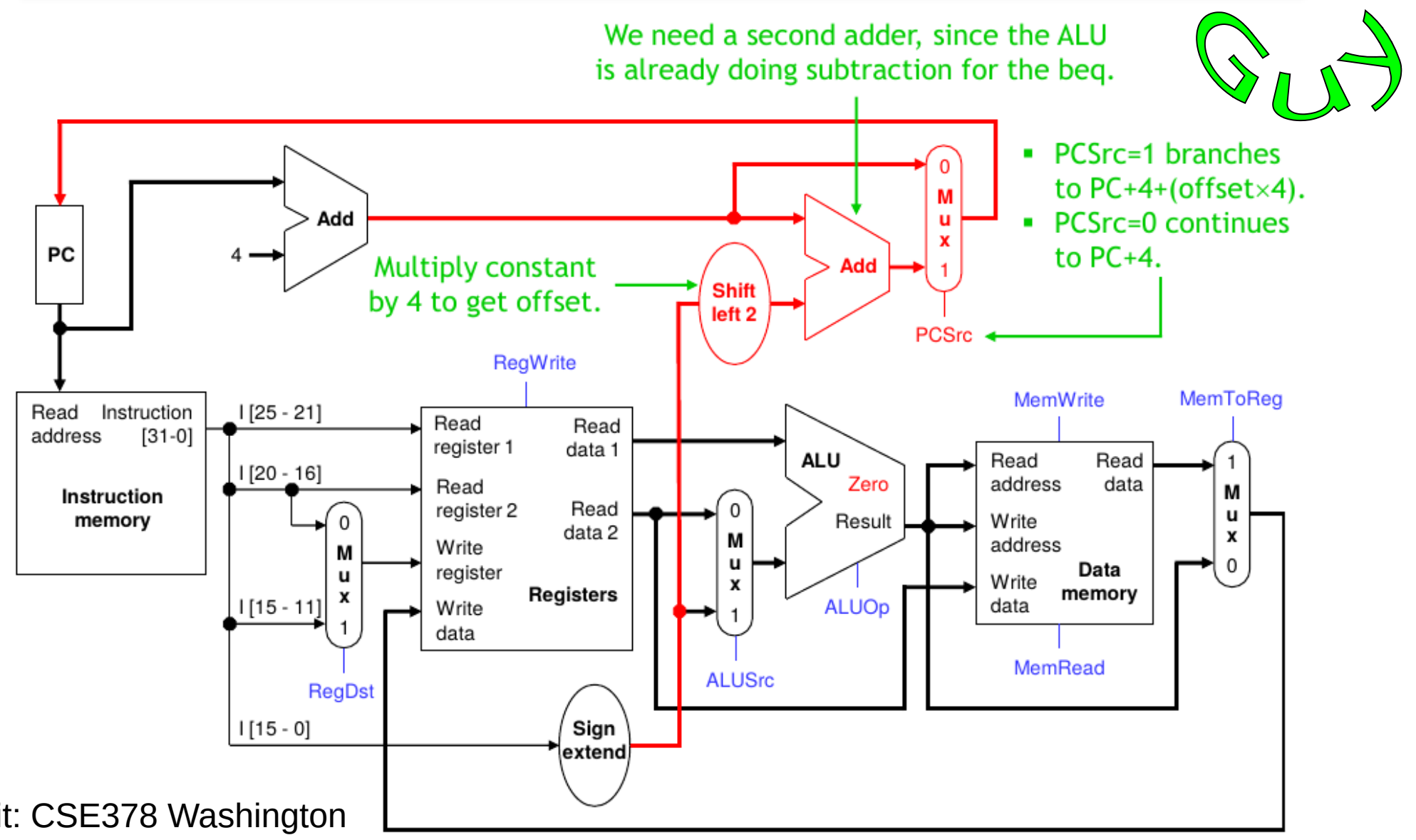


Conditional Branch Datapath (finished :-))

כאשר מוסיפים תמיכה בהסתעפות מותנה יש צורך לבטל את ההוראה שמשמאל (מסומן בשני קווים אלכסוניים)

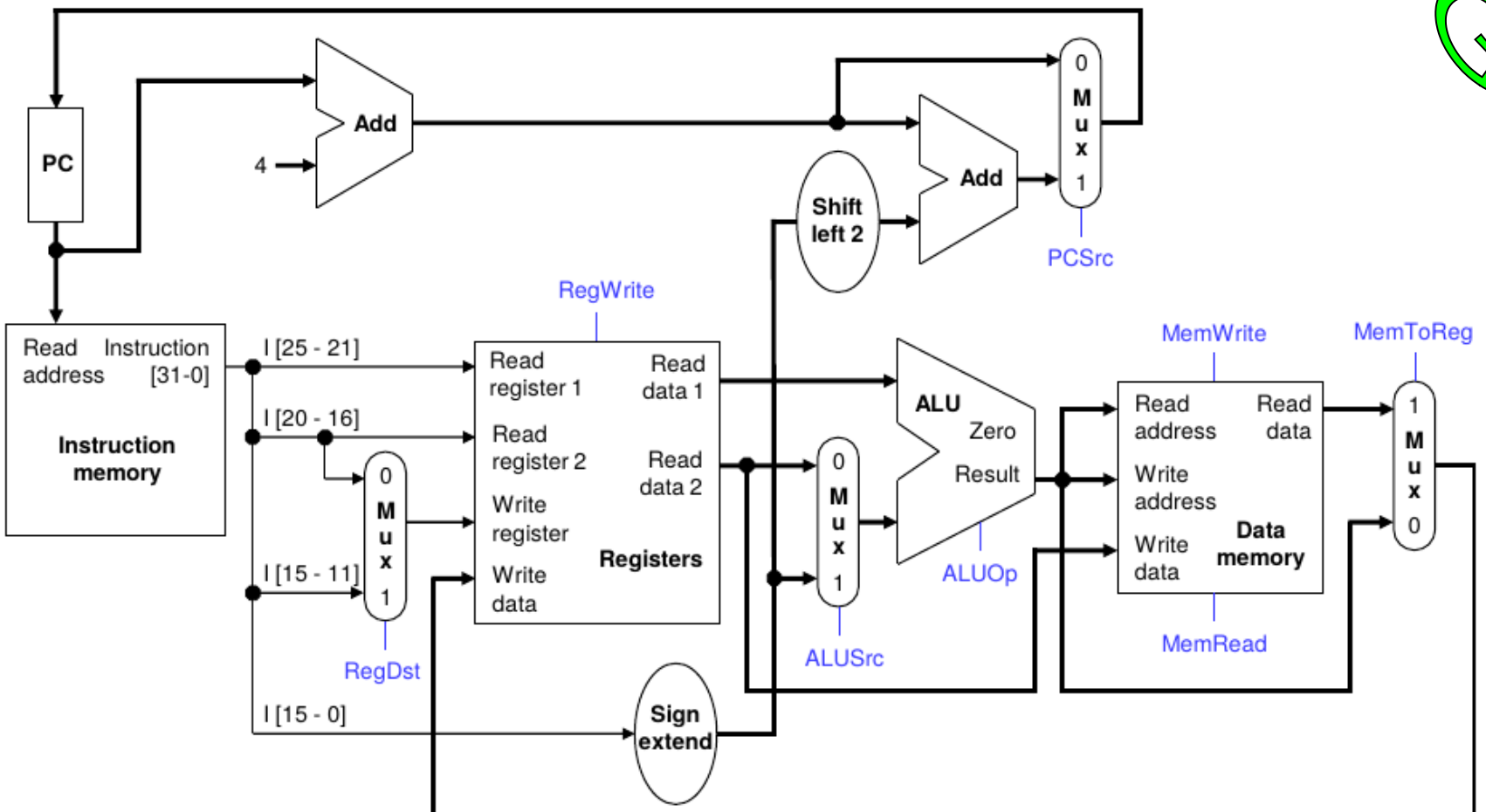


**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]



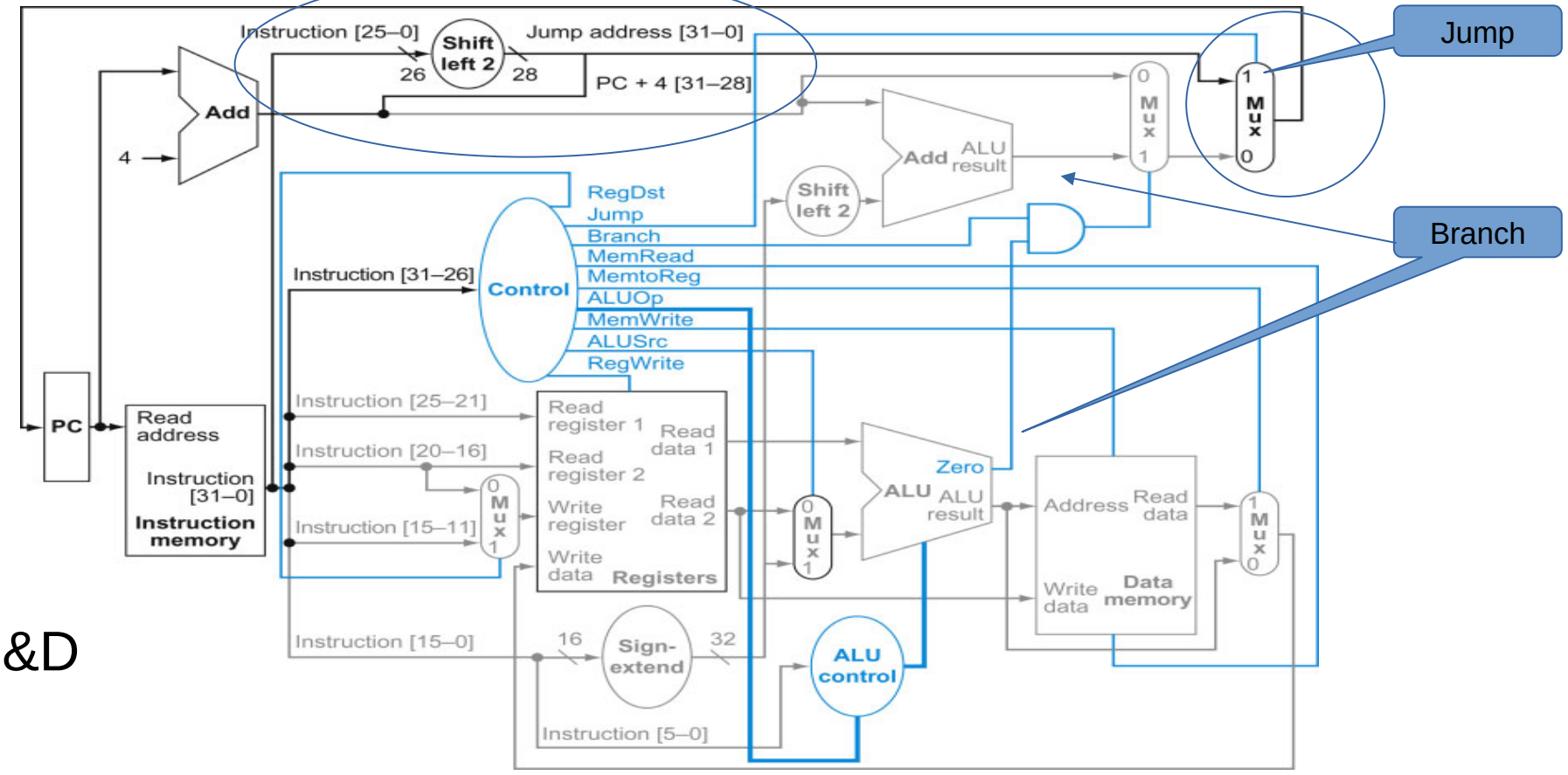
Credit: CSE378 Washington

The Final Datapath (without the controls)



Credit: CSE378 Washington

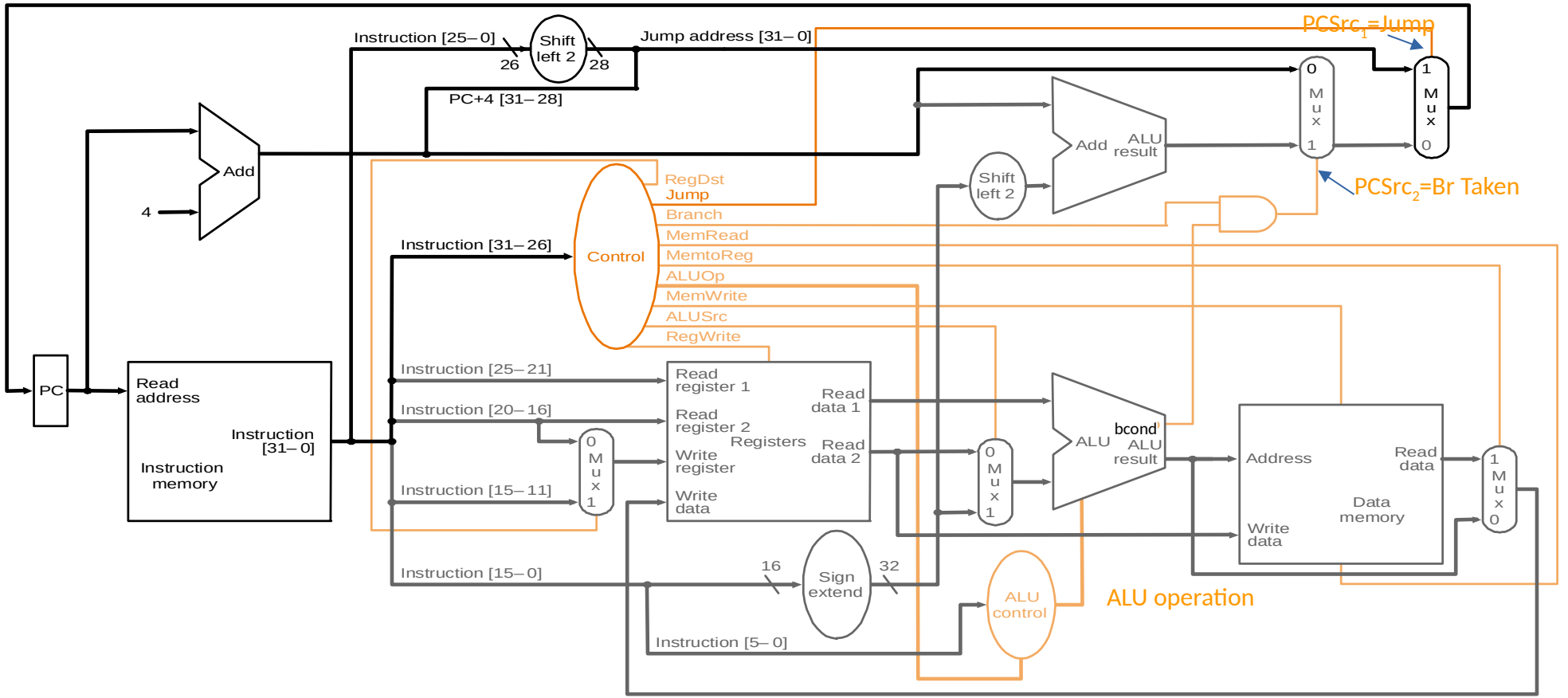
ניתן לצייר על שקף זה



H&P CO&D

FIGURE 4.24 The simple control and datapath are extended to handle the **jump** instruction. An additional multiplexor (at the upper right) is used to choose between the jump target and either the branch target or the sequential instruction following this one. This multiplexor is controlled by the jump control signal. The jump target address is obtained by shifting the lower 26 bits of the jump instruction left 2 bits, effectively adding 00 as the low-order bits, and then concatenating the upper 4 bits of PC + 4 as the high-order bits, thus yielding a 32-bit address.

Putting It All Together (MIPS)

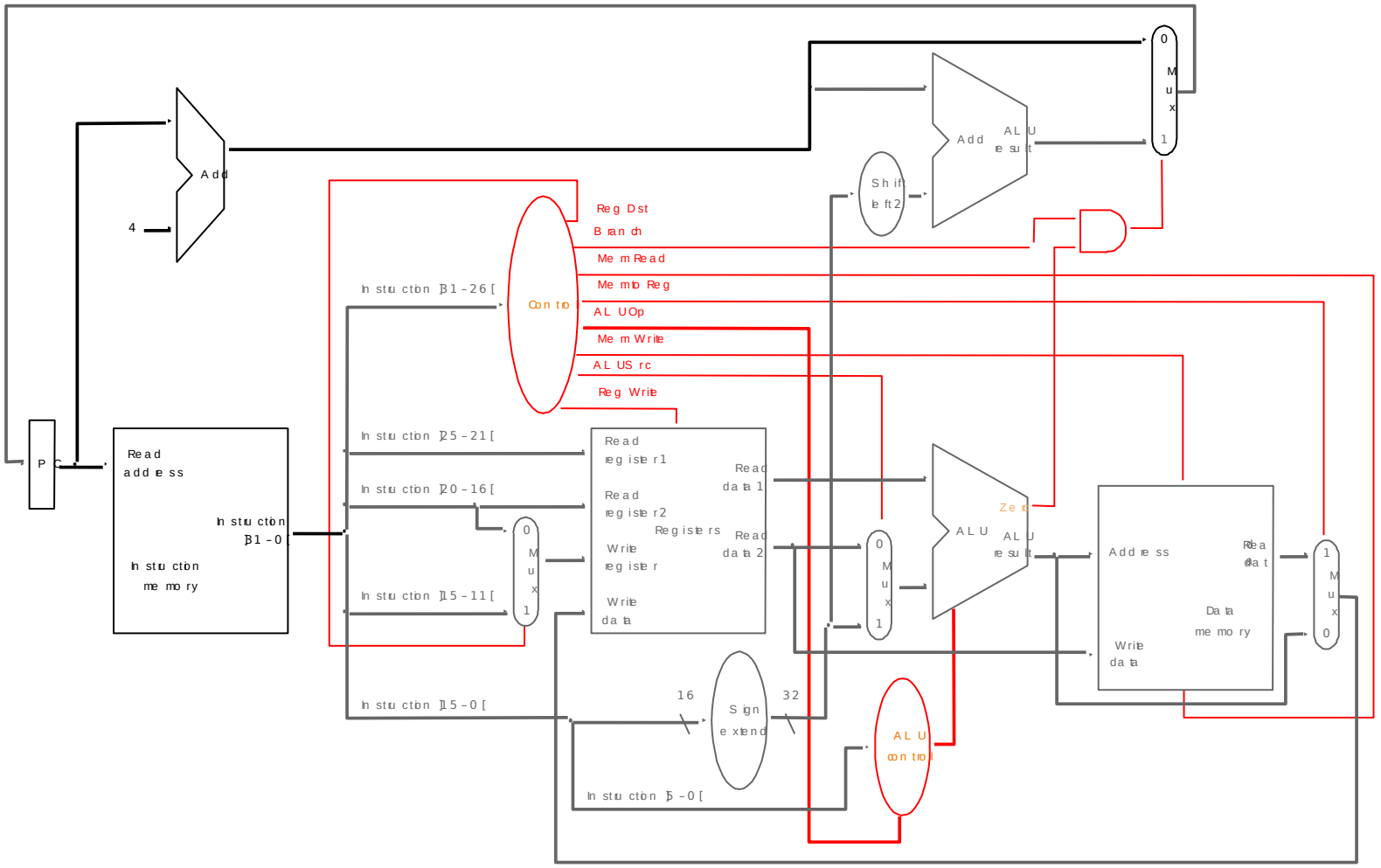


JAL, JR, JALR omitted

Single-Cycle Control Logic

Control

התרשים כאן מחדד את קווי הבקרה אך הוא חלקי (אין כאן תמיכה מלאה בכל ההוראות)



אנימציה

Let's explain all Control Signals:

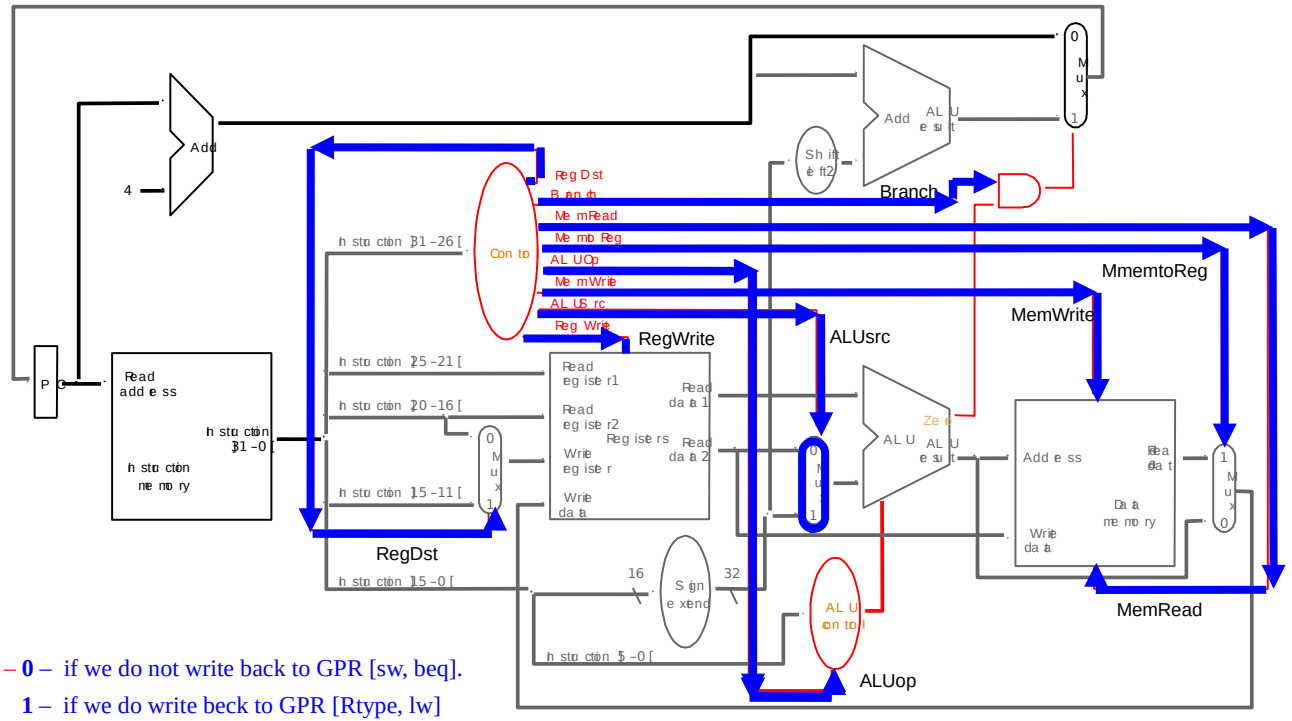
ALUSrc – **0** will connect GPR[Rt] to the ALU B input [Rtype, beq]

1- will connect sext(imm) to the ALU B input [in lw or sw inst.]

ALUOP – Two bits: **00** force the ALU to add [lw & sw]. **01** force the ALU to sub [beq]. **10** force the ALU to look at the funct. Field [Rtype].

MemRead – **0** - nothing happens. **1**- data is read from M[ALU output]

MemWrite – **0** - nothing happens. **1**- GPR[Rt] is written into M[ALU output] at the next rising edge of the CK



RegWrite – **0** – if we do not write back to GPR [sw, beq].

1 – if we do write beck to GPR [Rtype, lw]

MemToReg – **0** – The ALU output is fed to the GPR Write Data input [Rtype]. **1**- Data read from Memory is fed to the GPR Write Data input [lw]

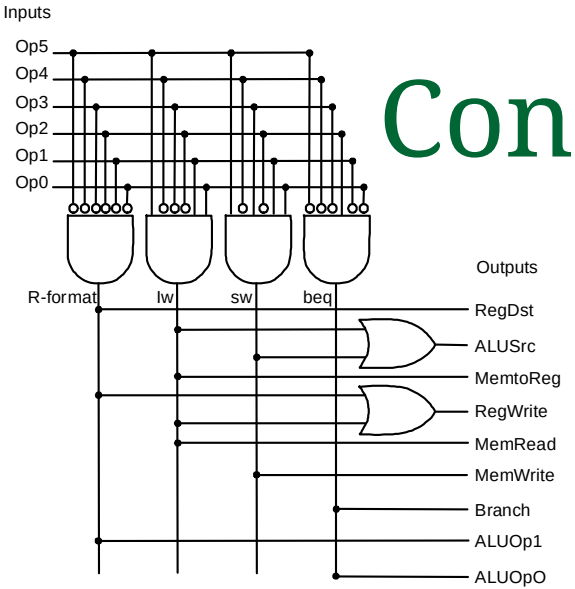
RegDst – **0** – The GPR Write Reg gets Rt field of the instruction [lw]. **1**- The GPR Write Reg gets Rd field of instruction [Rtype]

Branch – **0** – ia all instructions except beq. **1**- in beq. Selects whether to branch or not.

אנימציה

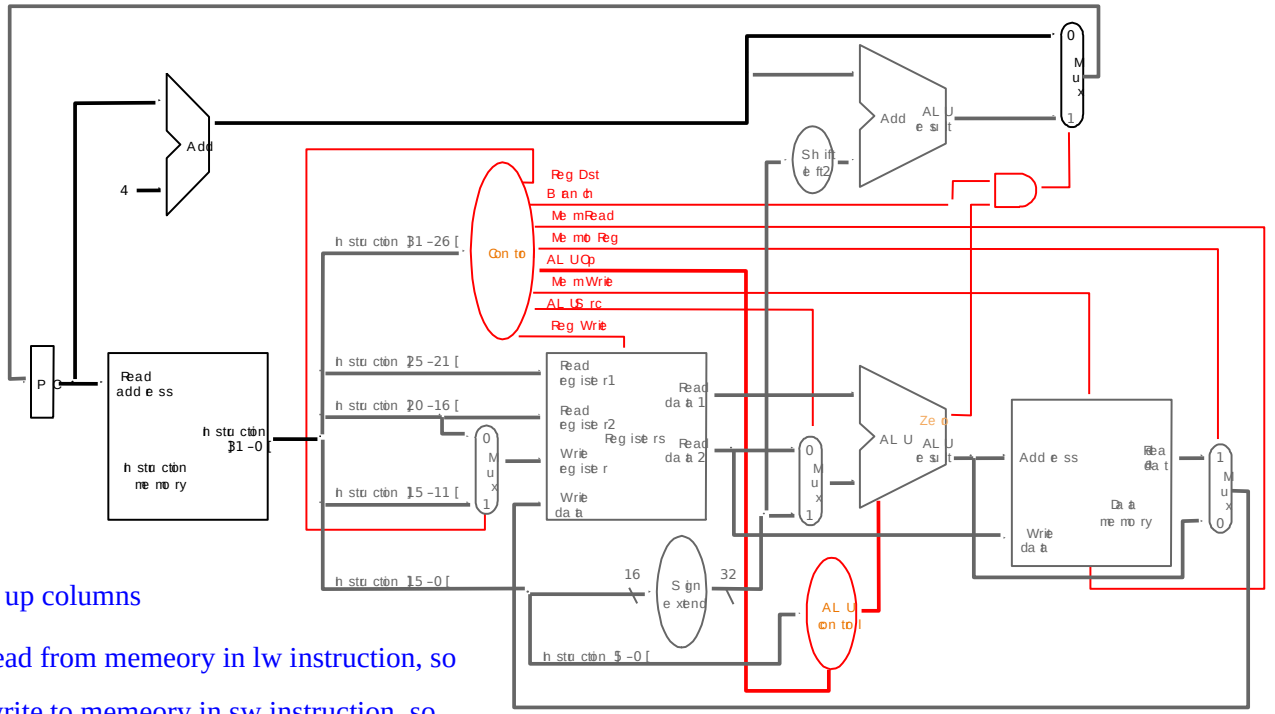
Control

We would like to build the Control Decoder together



Instr.	ALU Src	ALUOp [1:0]		Mem Read	Mem Write	Mem to Reg	Reg Dst	Reg Write	Branch
R-type									
lw									
sw									
beq									

אנימציה



We can fill up columns

We only read from memeory in lw instruction, so

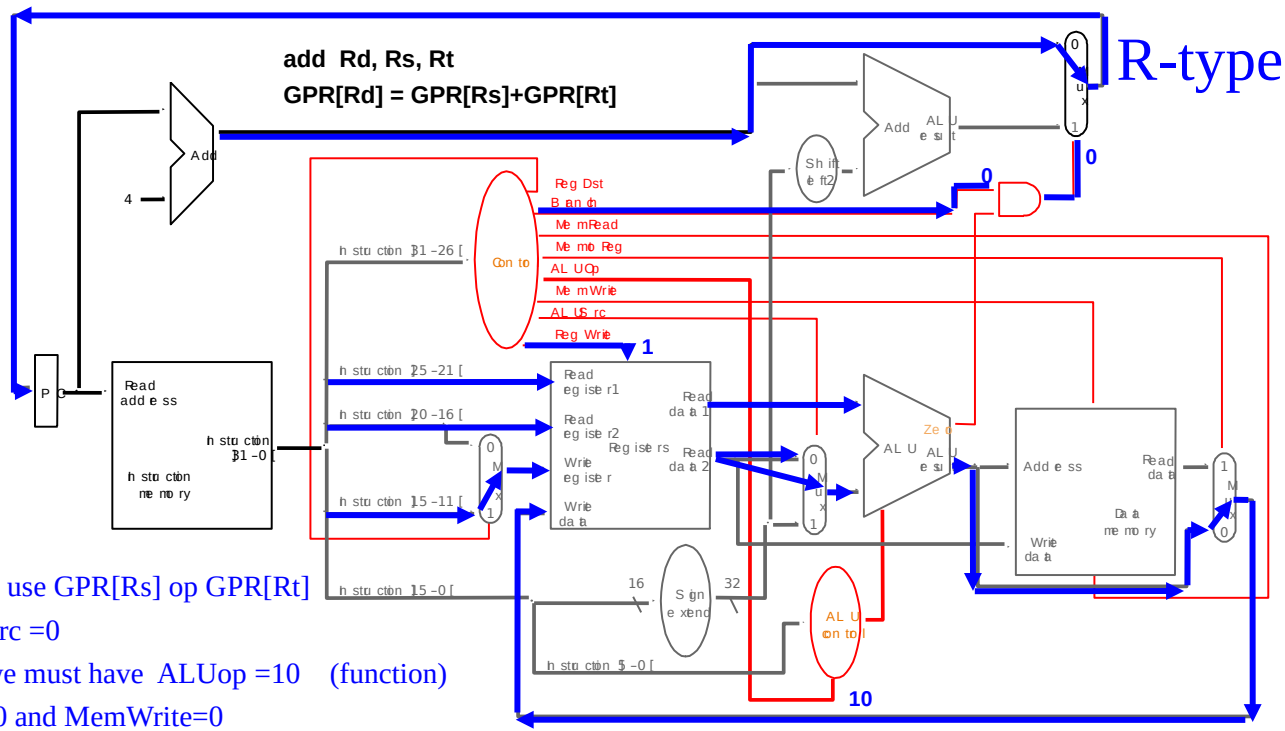
We only write to memeory in sw instruction, so

Instr.	ALU Src	ALUOp [1:0]		Mem Read	Mem Write	Mem to Reg	Reg Dst	Reg Write	Branch
R-type				0	0				
lw				1	0				
sw				0	1				
beq				0	0				

A better way is to follow
Each instruction and see
What happens in the
Data-Path during the
Execution of the instruction

R-type

אנימציה



In Rtype we use GPR[Rs] op GPR[Rt]

Thus, ALUSrc =0

In Rtype we must have ALUop =10 (function)

MemRead=0 and MemWrite=0

MemToReg=0

Instr.	ALU Src	ALUOp [1:0]		Mem Read	Mem Write	Mem Reg	Reg Dst	Reg Write	Branch
R-type	**0**	**1**	**0**	0	0	**0**	**1**	**1**	**0**
lw				1	0				
sw				0	1				
beq				0	0				

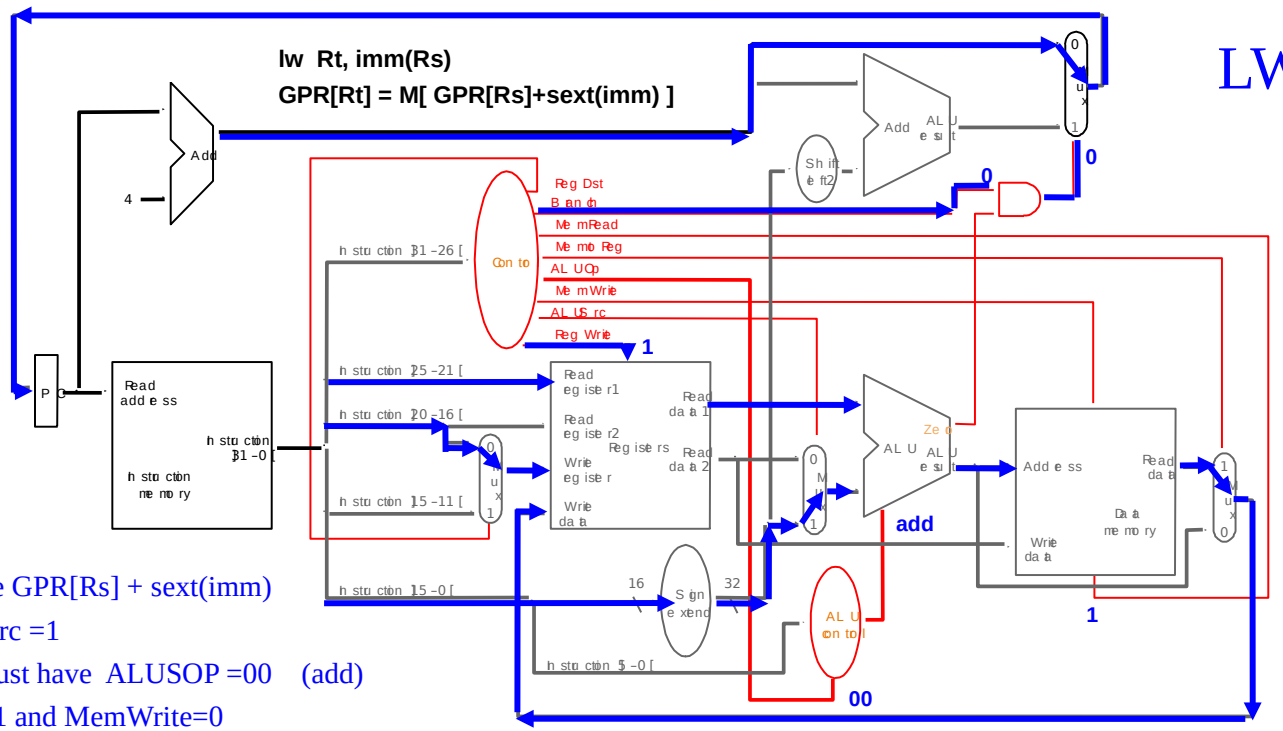
RegDst=1 to select Rd

RegWrite=1 to write to the GPR

Branch=0 so we always get PC=PC+4

LW

אנימציה



In lw we use GPR[Rs] + sext(imm)

Thus, ALUSrc =1

In lw we must have ALUSOP =00 (add)

MemRead=1 and MemWrite=0

MemToReg=1

Instr.	ALU Src	ALUOp [1:0]		Mem Read	Mem Write	Mem Reg	Reg Dst	Reg Write	Branch
R-type	0	1	0	0	0	0	1	1	0
lw	**1**	**0**	**0**	1	0	**1**	**0**	**1**	**0**
sw				0	1				
beq				0	0				

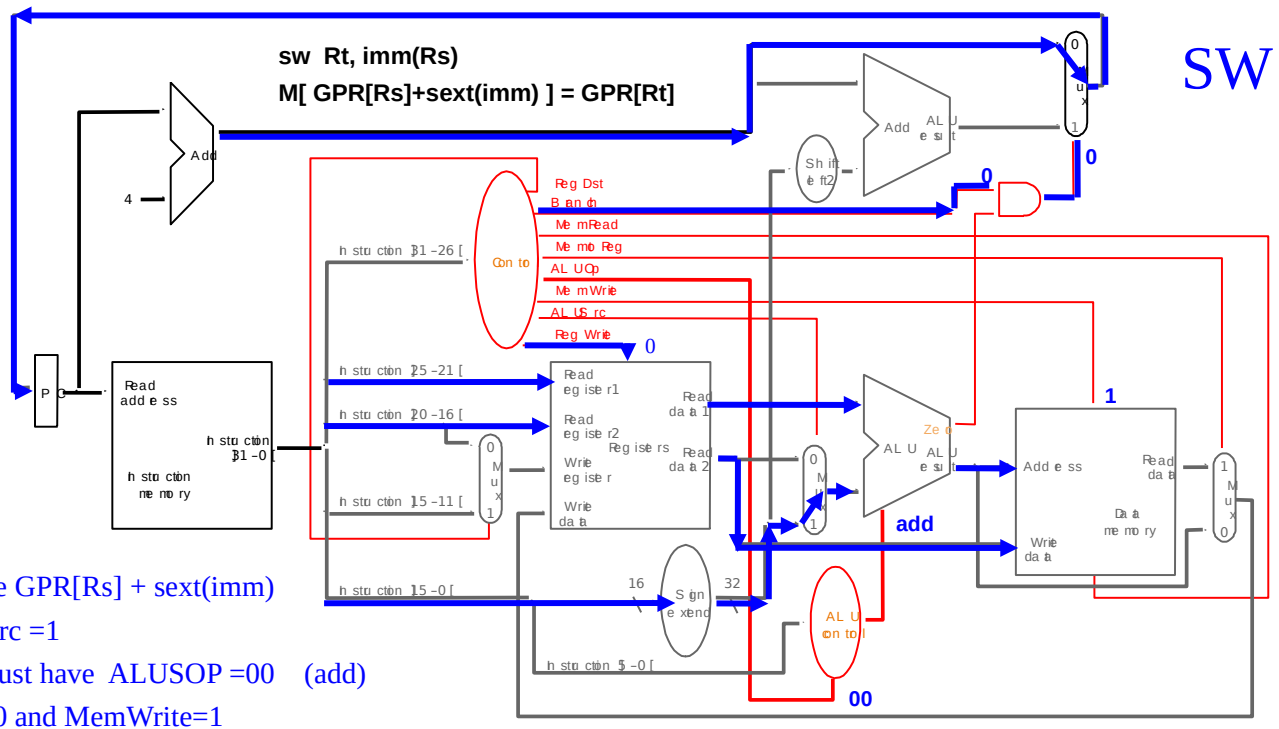
RegDst=0 to select Rt

RegWrite=1 to write to the GPR

Branch=0 so we always get PC=PC+4

SW

אנימציה



In sw we use GPR[Rs] + sext(imm)

Thus, ALUSrc =1

In sw we must have ALUSOP =00 (add)

MemRead=0 and MemWrite=1

MemToReg=X. We do not write back!

Instr.	ALU Src	ALUOp [1:0]		Mem Read	Mem Write	Mem Reg	Reg Dst	Reg Write	Branch
R-type	0	1	0	0	0	0	1	1	0
lw	1	0	0	1	0	1	0	1	0
sw	**1**	**0**	**0**	0	1	**X**	**X**	**0**	**0**
beq				0	0				

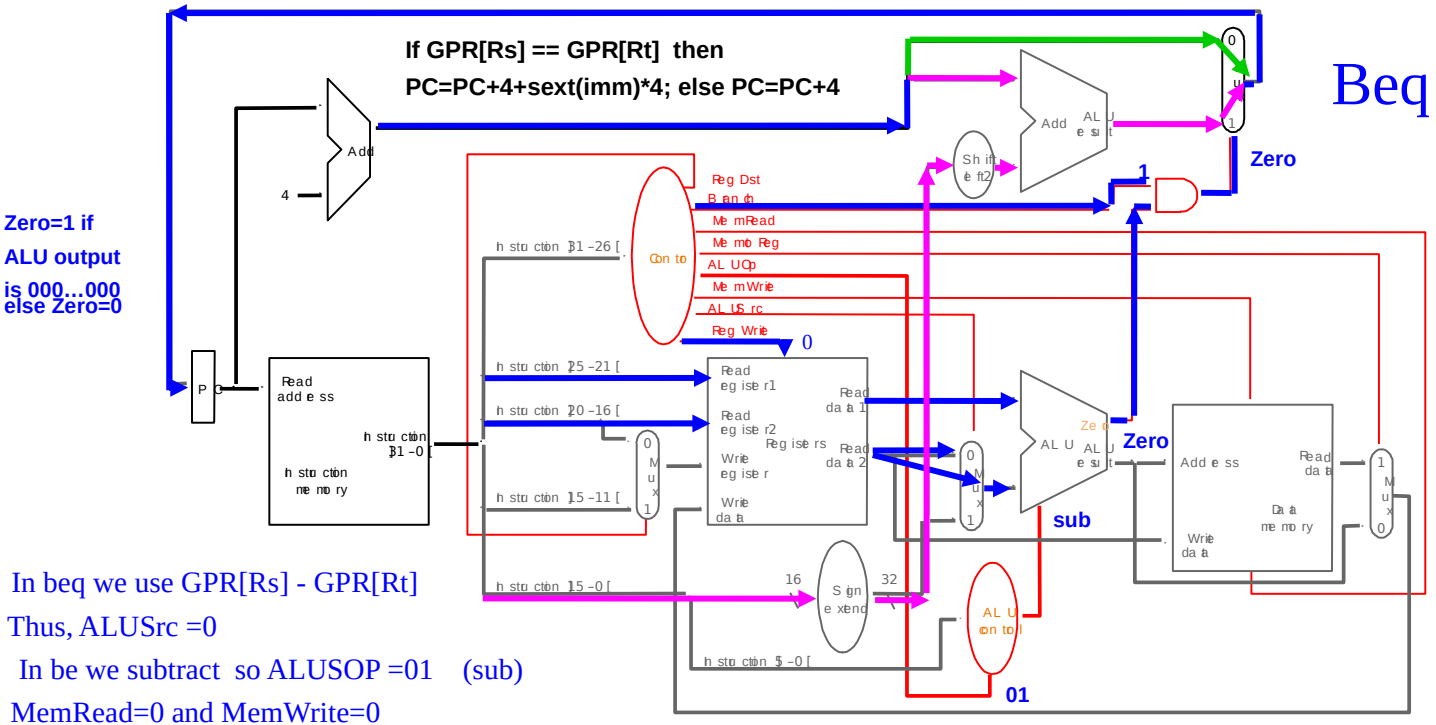
RegDst=X. We do not write back!

RegWrite=0. We do not write back!

Branch=0 so we always get PC=PC+4

Beq

אנימציה



In beq we use GPR[Rs] - GPR[Rt]

Thus, ALUSrc =0

In be we subtract so ALUSOP =01 (sub)

MemRead=0 and MemWrite=0

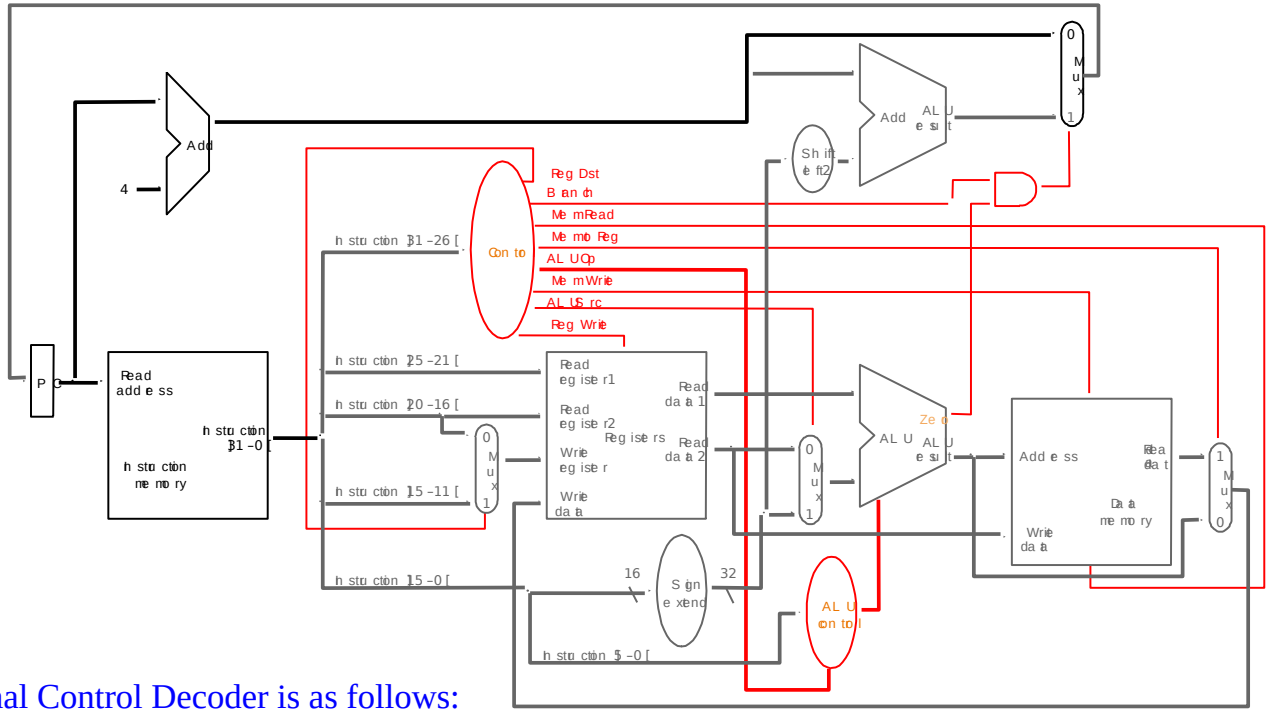
MemToReg=X. We do not write back!

Instr.	ALU Src	ALUOp [1:0]		Mem Read	Mem Write	Mem to Reg	Reg Dst	Reg Write	Branch
R-type	0	1	0	0	0	0	1	1	0
lw	1	0	0	1	0	1	0	1	0
sw	1	0	0	0	1	X	X	0	0
beq	**0**	**0**	**1**	0	0	**X**	**X**	**0**	**1**

RegDst= X. We do not write back!

RegWrite=0 We do not write back!

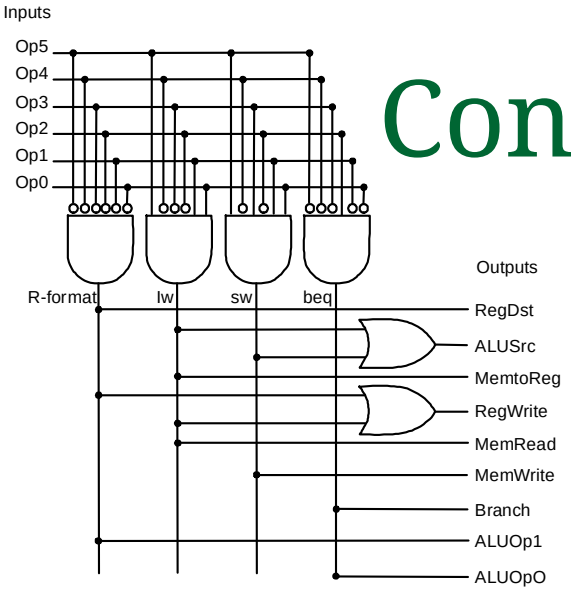
Branch=1 so we check Zero to decide whether to branch (PC= branch target) or not to branch (PC=PC+4)



So the final Control Decoder is as follows:

Instr.	ALU Src	ALUOp [1:0]		Mem Read	Mem Write	Mem to Reg	Reg Dst	Reg Write	Branch
R-type	0	1	0	0	0	0	1	1	0
lw	1	0	0	1	0	1	0	1	0
sw	1	0	0	0	1	X	X	0	0
beq	0	0	1	0	0	X	X	0	1

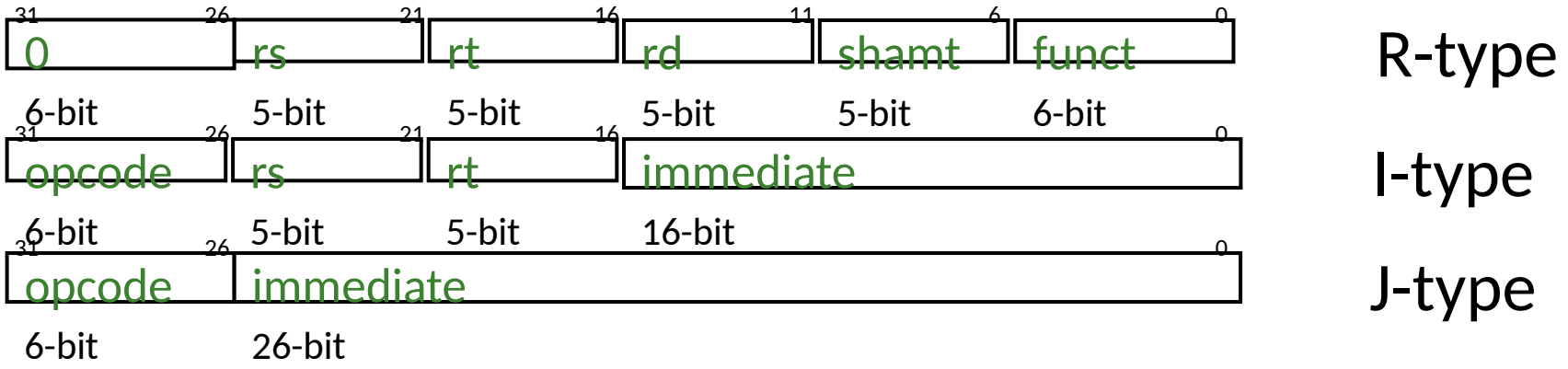
Control Decoder



Instr.	ALU Src	ALUOp [1:0]		Mem Read	Mem Write	Mem to Reg	Reg Dst	Reg Write	Branch
R-type	0	1	0	0	0	0	1	1	0
lw	1	0	0	1	0	1	0	1	0
sw	1	0	0	0	1	X	X	0	0
beq	0	0	1	0	0	X	X	0	1

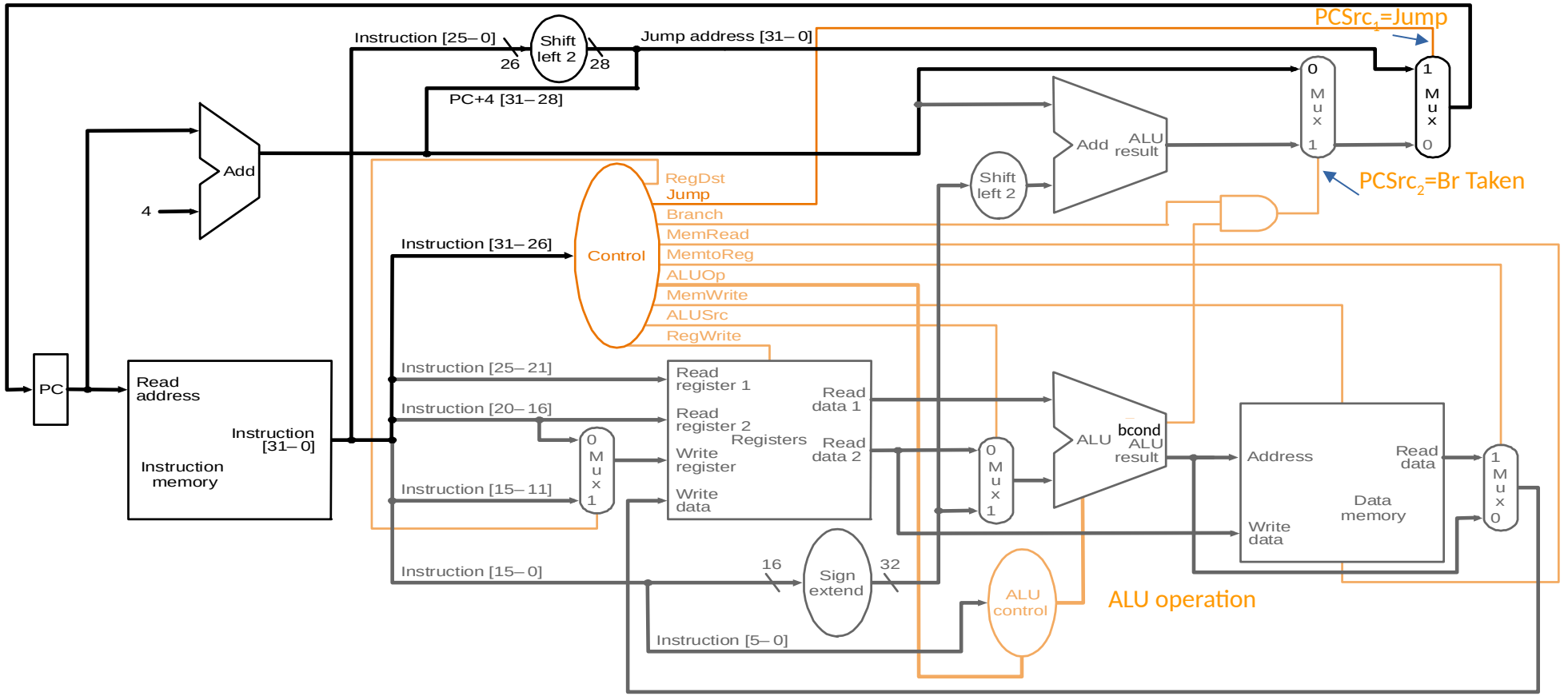
Single-Cycle Hardwired Control (MIPS)

- As combinational function of Inst=MEM[PC]



- Consider – סט מצומצם
  - All R-type and I-type ALU instructions
  - LW and SW
  - BEQ, BNE, BLEZ, BGTZ
  - J, JR, JAL, JALR

Putting It All Together (MIPS)



JAL, JR, JALR omitted

Single-Bit Control Signals

	When De-asserted	When asserted	Equation
RegDest	GPR write select according to rt, i.e., inst[20:16]	GPR write select according to rd, i.e., inst[15:11]	opcode==0
ALUSrc	2nd ALU input from 2nd GPR read port	2nd ALU input from sign-extended 16-bit immediate	(opcode!=0) && (opcode!=BEQ) &&

JAL and JALR require additional RegDest and MemtoReg options

Single-Bit Control Signals

	When De-asserted	When asserted	Equation
MemRead	Memory read disabled	Memory read port return load value	opcode==LW
MemWrite	Memory write disabled	Memory write enabled	opcode==SW
$PCSrc_1$	According to $PCSrc_2$	next PC is based on 26-bit immediate jump target	(opcode==J) || (opcode==JAL)
$PCSrc_2$	next PC = PC + 4	next PC is based on 16-bit immediate branch target	(opcode==Bxx) && "bcond is satisfied"

JR and JALR require additional PCSrc options

ALU Control

- case opcode

  R-type: '0' ▲ select operation according to funct

  'ALUi' ▲ selection operation according to opcode

  'LW' ▲ select addition

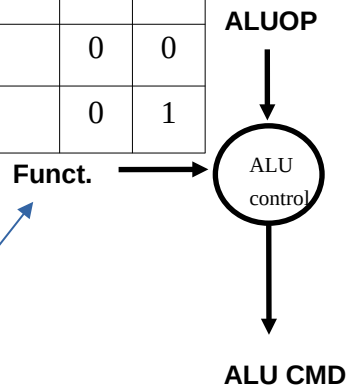
  'SW' ▲ select addition

  'Bxx' ▲ select bcond generation function

  other inst. ▲ don't care

- Example of ALU CMD (operations)
  - ADD, SUB, AND, OR, XOR, NOR, etc.
  - bcond on equal, not equal, LE zero, GT zero, etc.

Instr.	ALUOp [1:0]	
R-type	1	0
lw	0	0
sw	0	0
beq	0	1

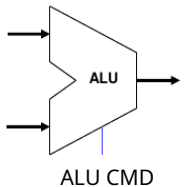


אנימציה

MIPS ALU CMDs

- Here's a simple ALU with five operations, selected by a 3-bit control signal ALU CMD

ALU CMD	Function
000	and
001	or
010	add
110	subtract
111	slt



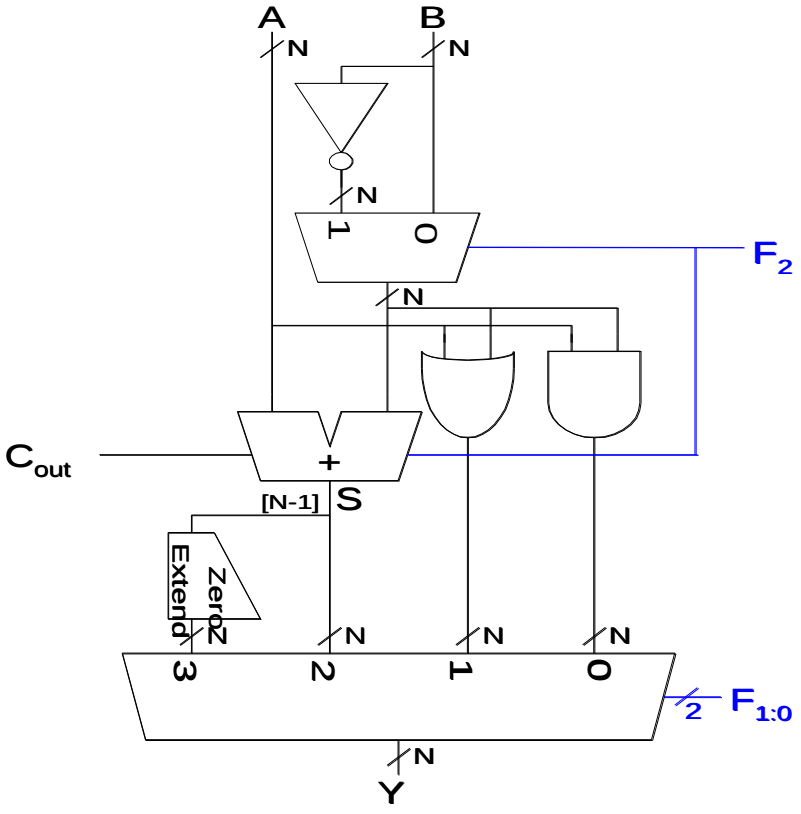
CORE INSTRUCTION SET				OPCODE / FUNCT
NAME, MNEMONIC	FOR-MAT	OPERATION (in Verilog)		(Hex)
Add	add	R	R[rd] = R[rs] + R[rt]	(1) 0 / 20_{hex}

מתוך דפי הסיכום של MIPS:

Taken from: **CSE378 University of Washington**
https://courses.cs.washington.edu/courses/cse378/10sp/lectures.html

Example: ALU Design – The Logic part

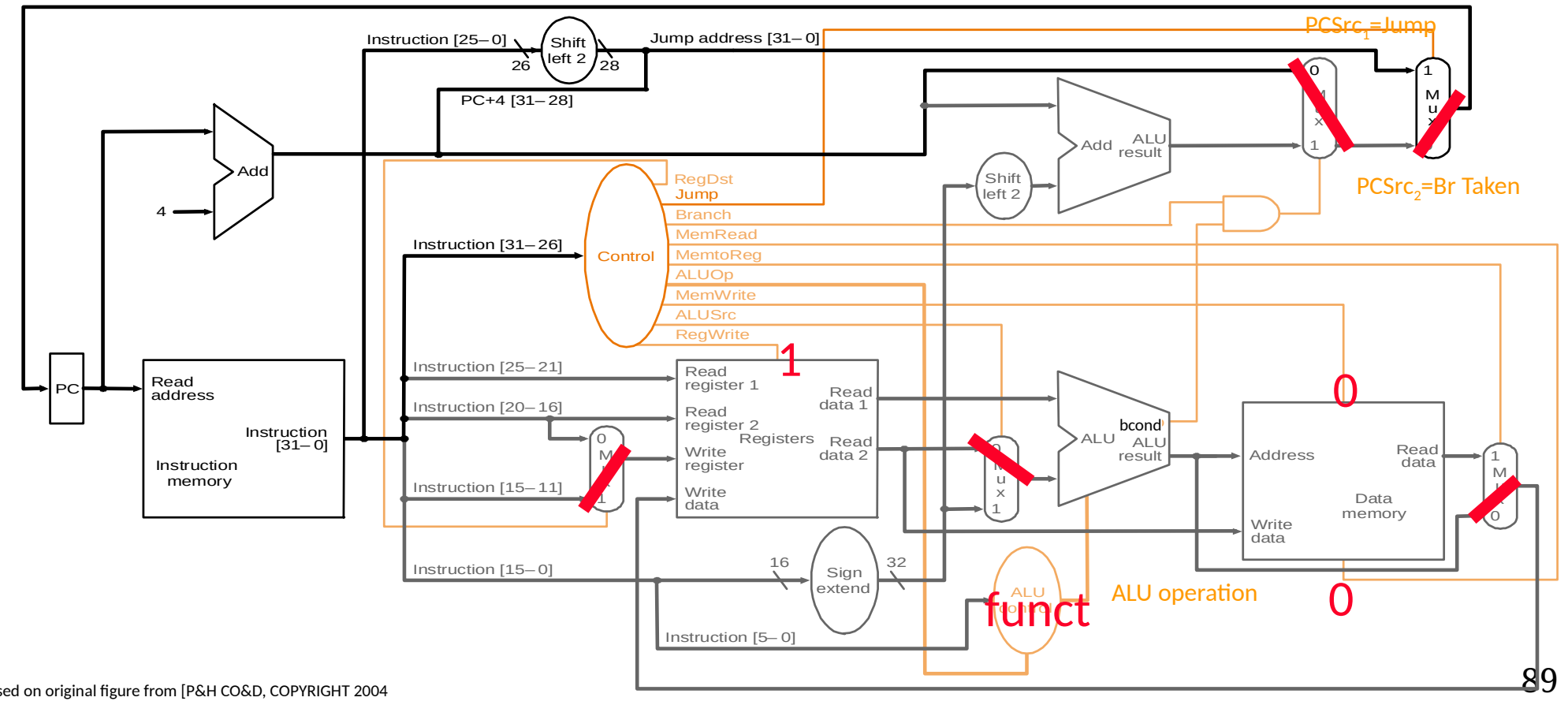
- ALU operation ($F_{2:0}$) comes from the control logic



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

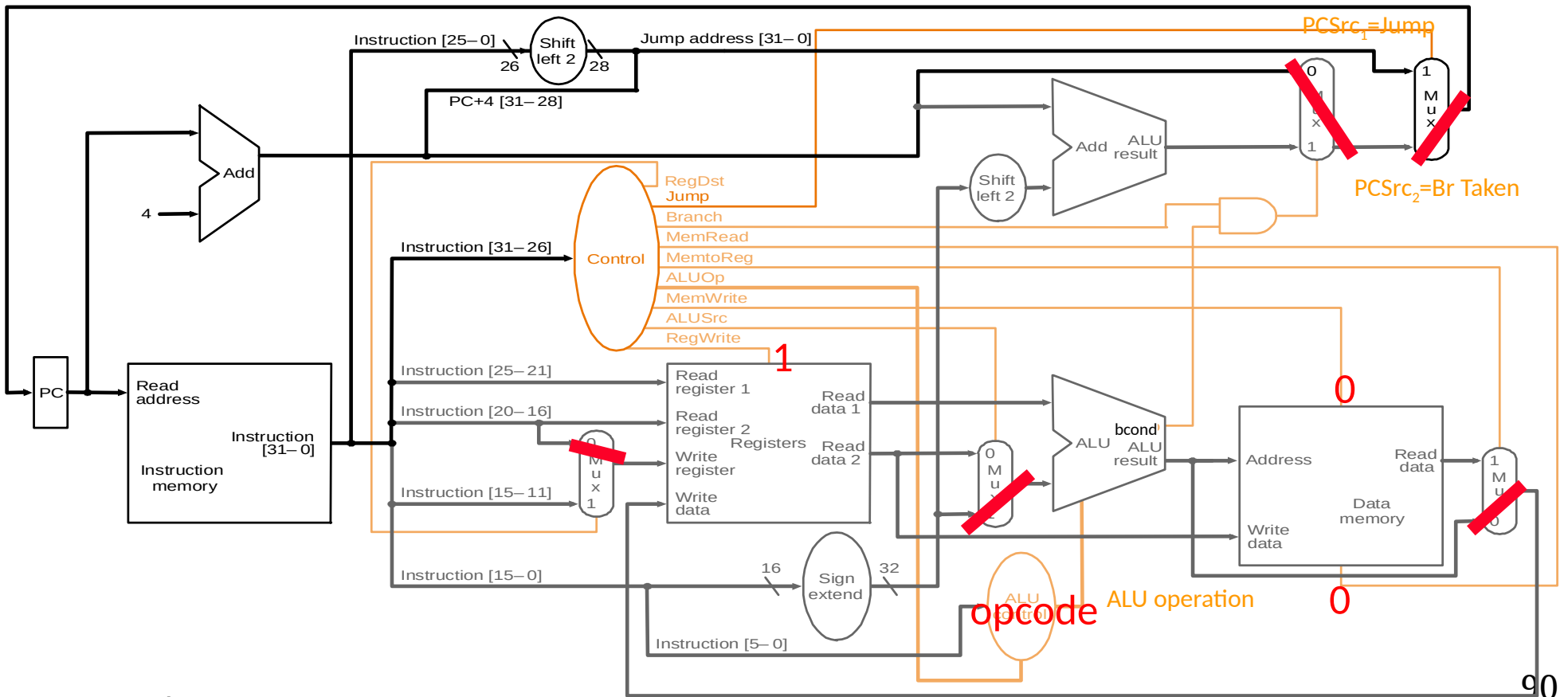
R-Type ALU

אנימציה



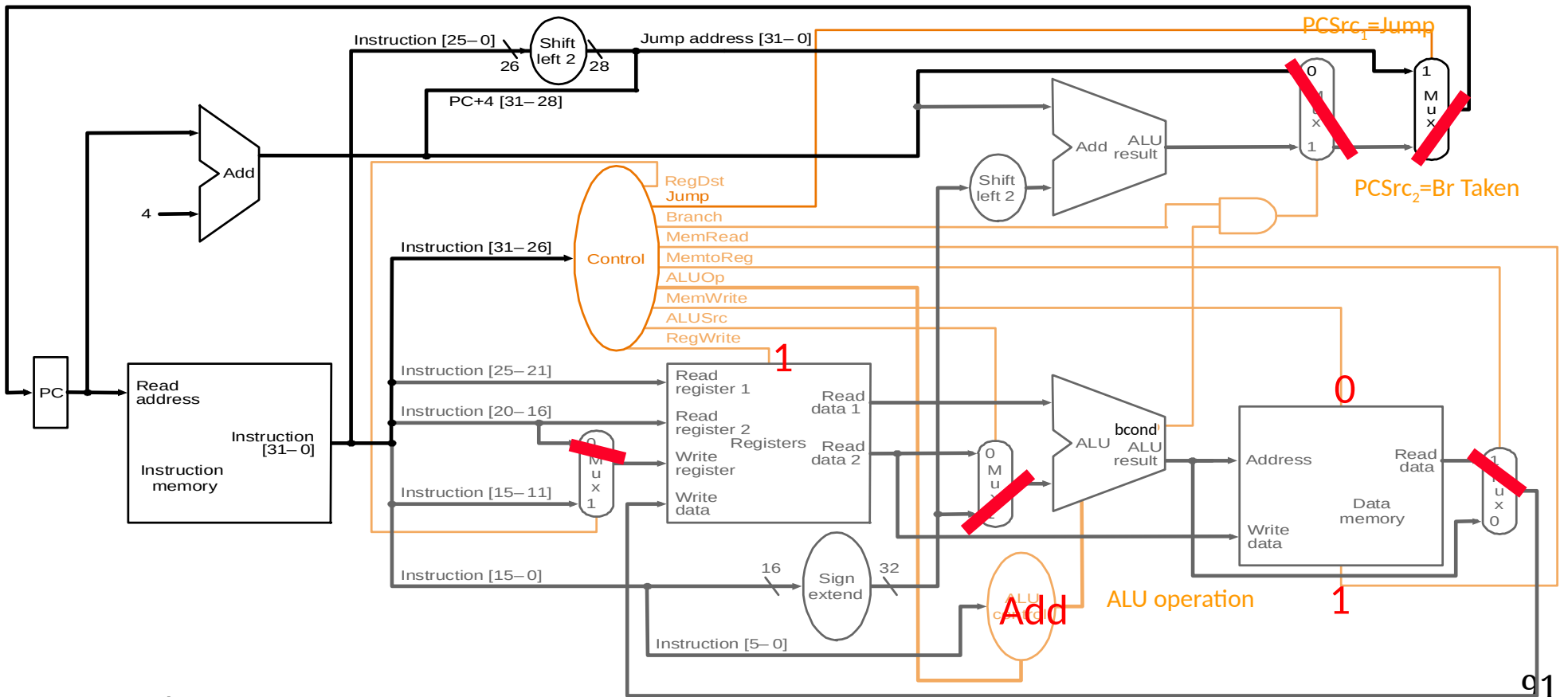
**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

I-Type ALU



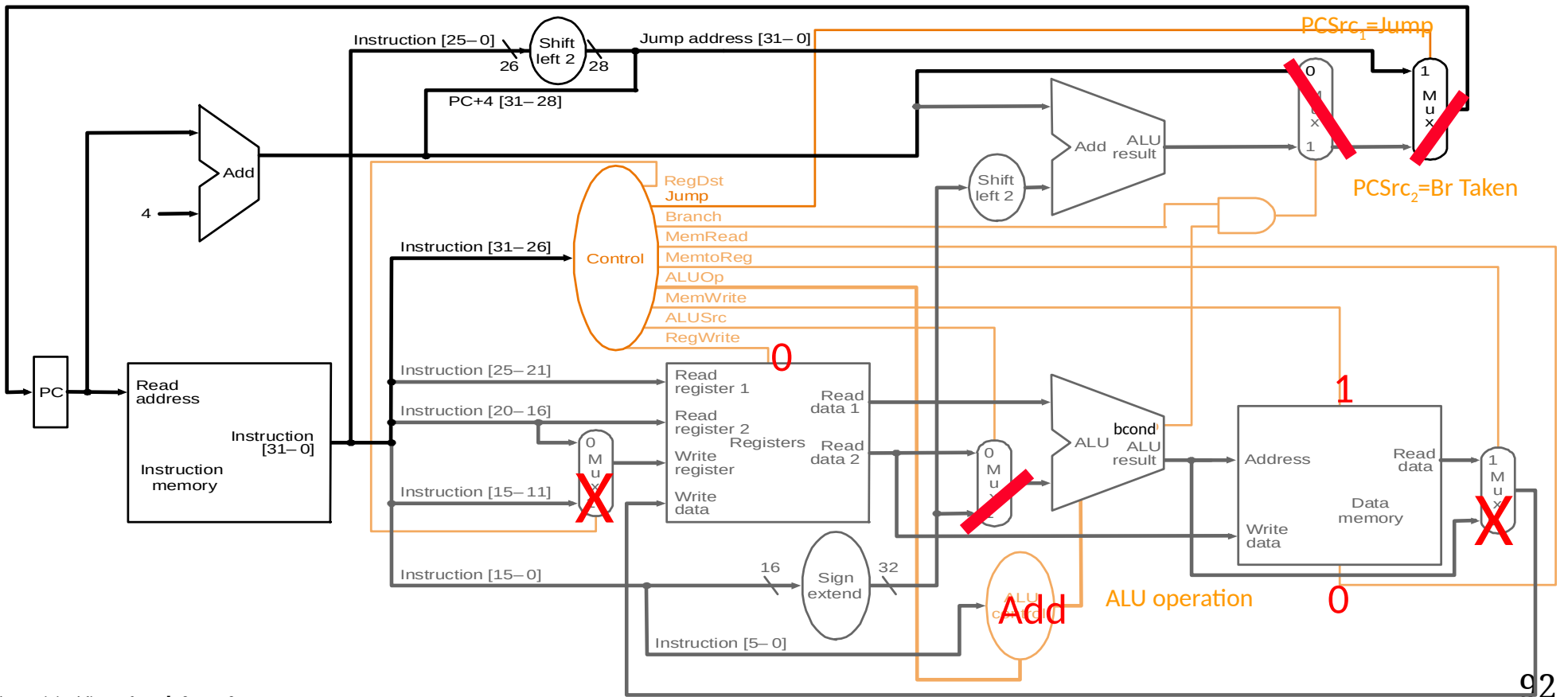
**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

LW



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

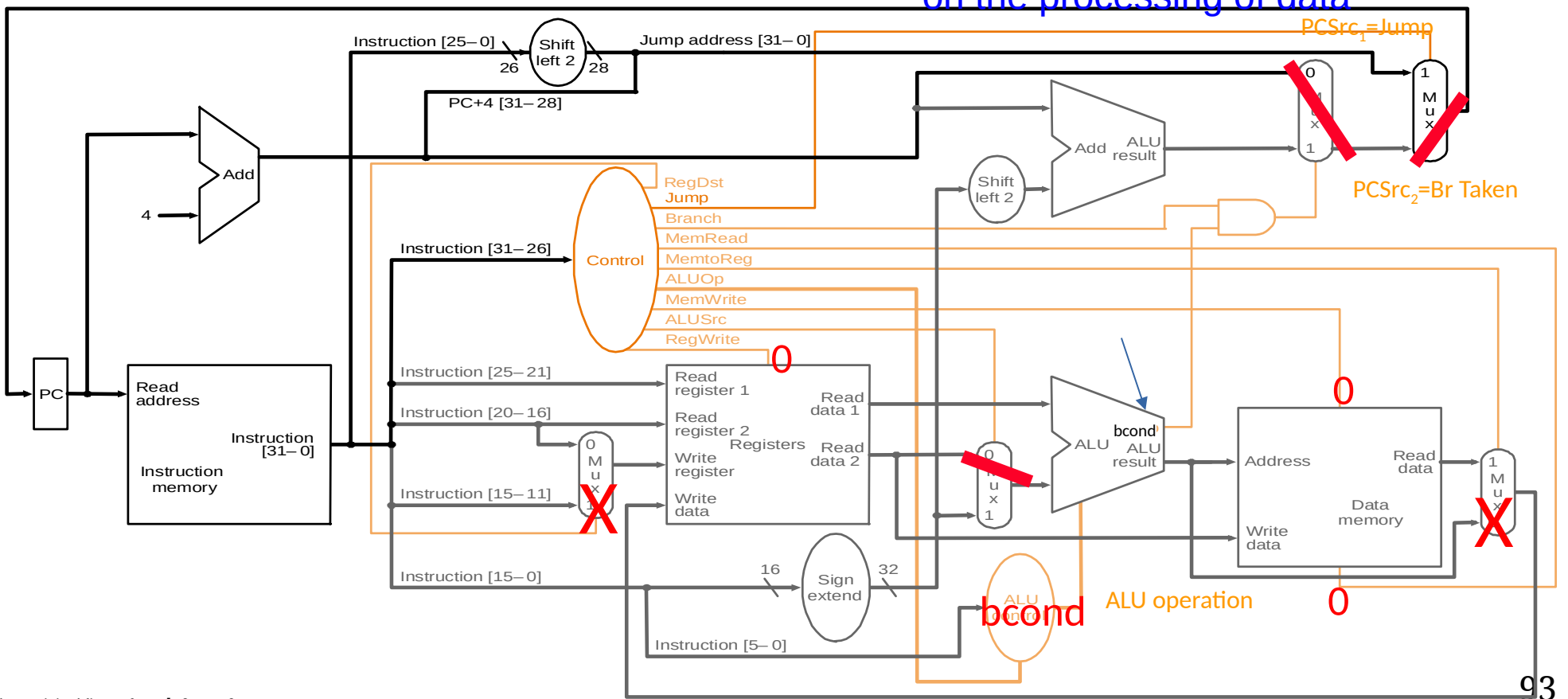
SW



Branch (Not Taken)

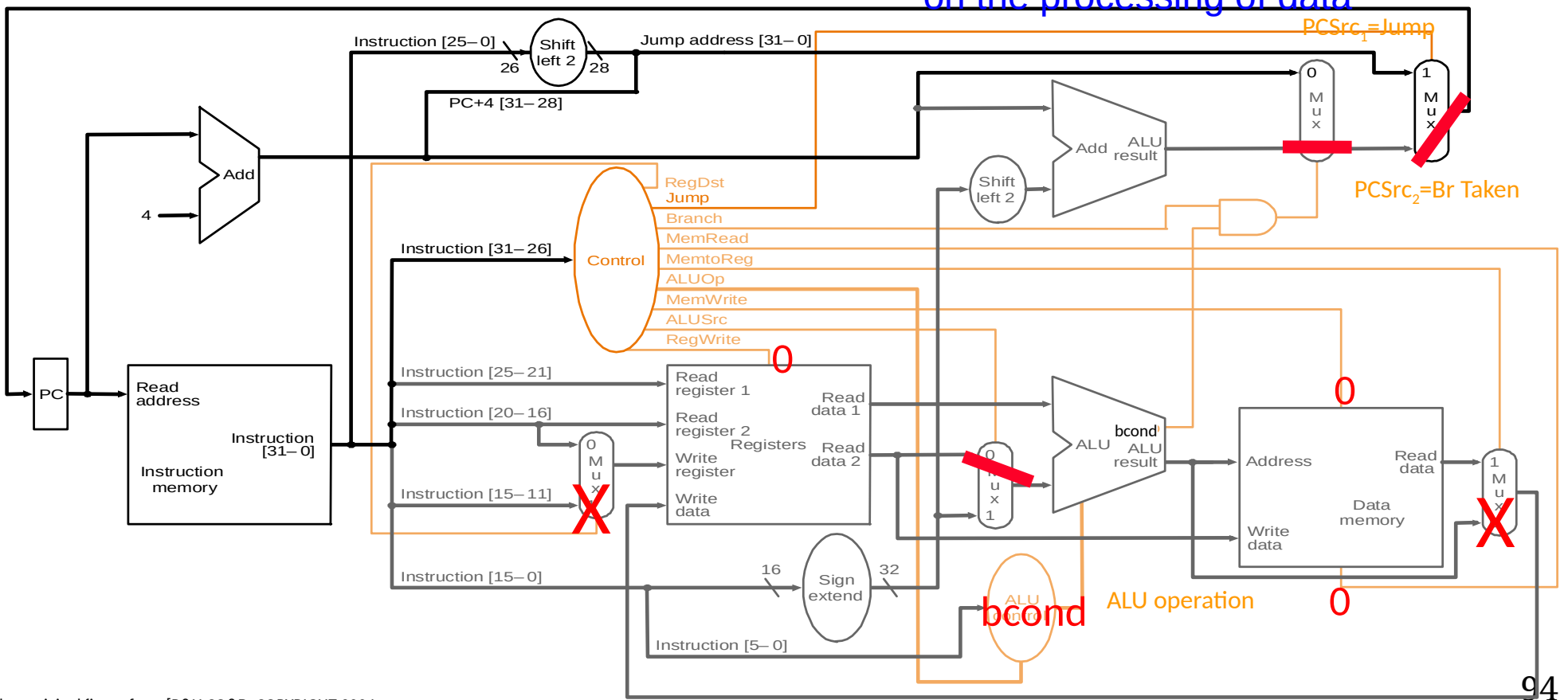
אנימציה

Some control signals are dependent on the processing of data

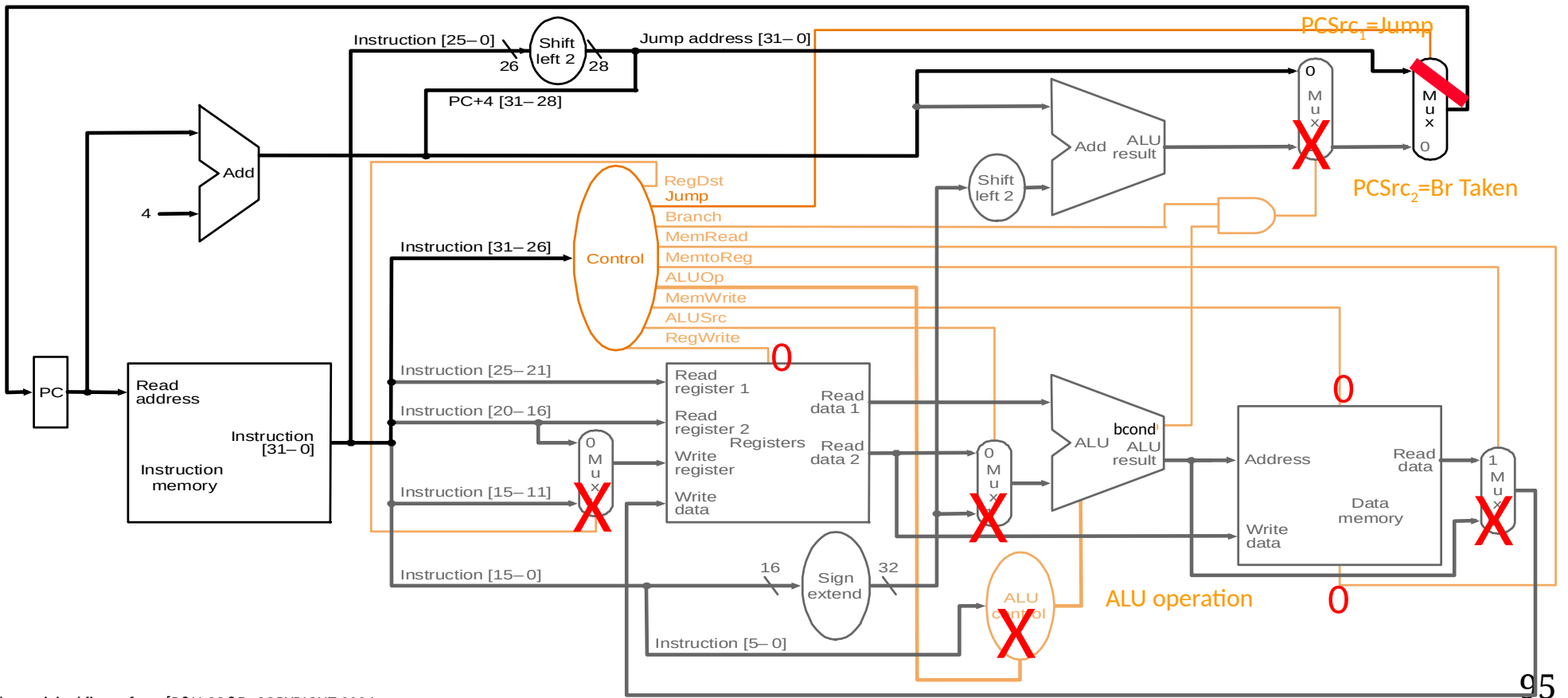


Branch (Taken)

Some control signals are dependent on the processing of data



Jump



What is in That Control Box?

- Combinational Logic ✉ Hardwired Control
  - Idea: Control signals generated combinationally based on instruction
  - Necessary in a single-cycle microarchitecture…

- Sequential Logic ✉ Sequential/Microprogrammed Control
  - Idea: A memory structure contains the control signals associated with an instruction
  - Control Store

יוסבר במצגת נפרדת!

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?

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 five phases of the instruction processing cycle take a *single machine clock cycle* to complete

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)

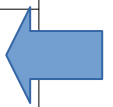
- Do each of the above phases take the same time (latency) for all instructions?

- לא כל ההוראות צריכות את כל 5 השלבים.
- לא כל שלב אורך אותו הזמן.
- סביר שהפקודה האיטית ביותר תזדקק לכל השלבים.

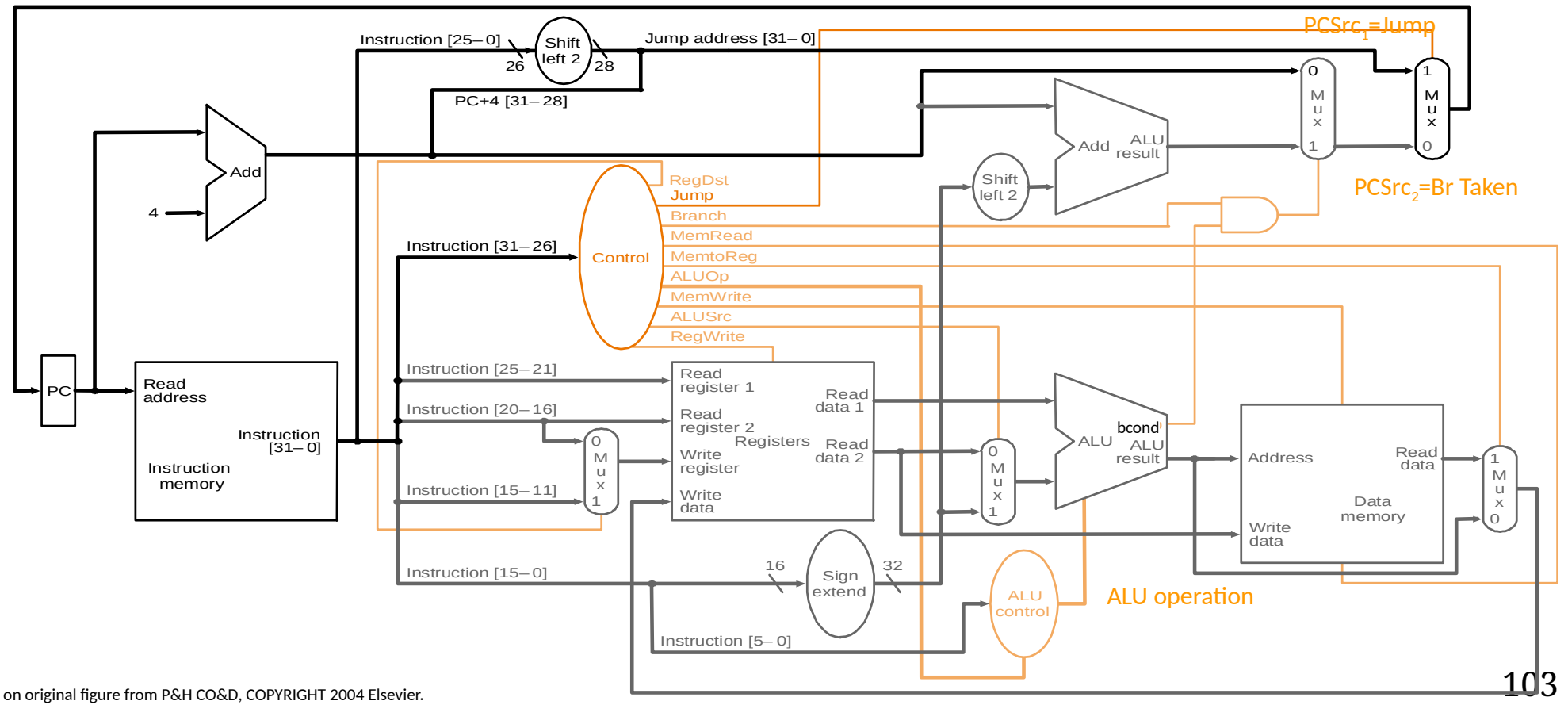
Single-Cycle Datapath Analysis - דוגמה

- Assume – זוהי רק דוגמה המניחה ערכים להשהיות בכל אחד מהשלבים
  - memory units (read or write): 200 ps
  - ALU and adders: 100 ps
  - register file (read or write): 50 ps
  - other combinational logic: 0 ps
  - בשקף הבא נציב את הזמנים ונחפש את ההוראה הכי איטית

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

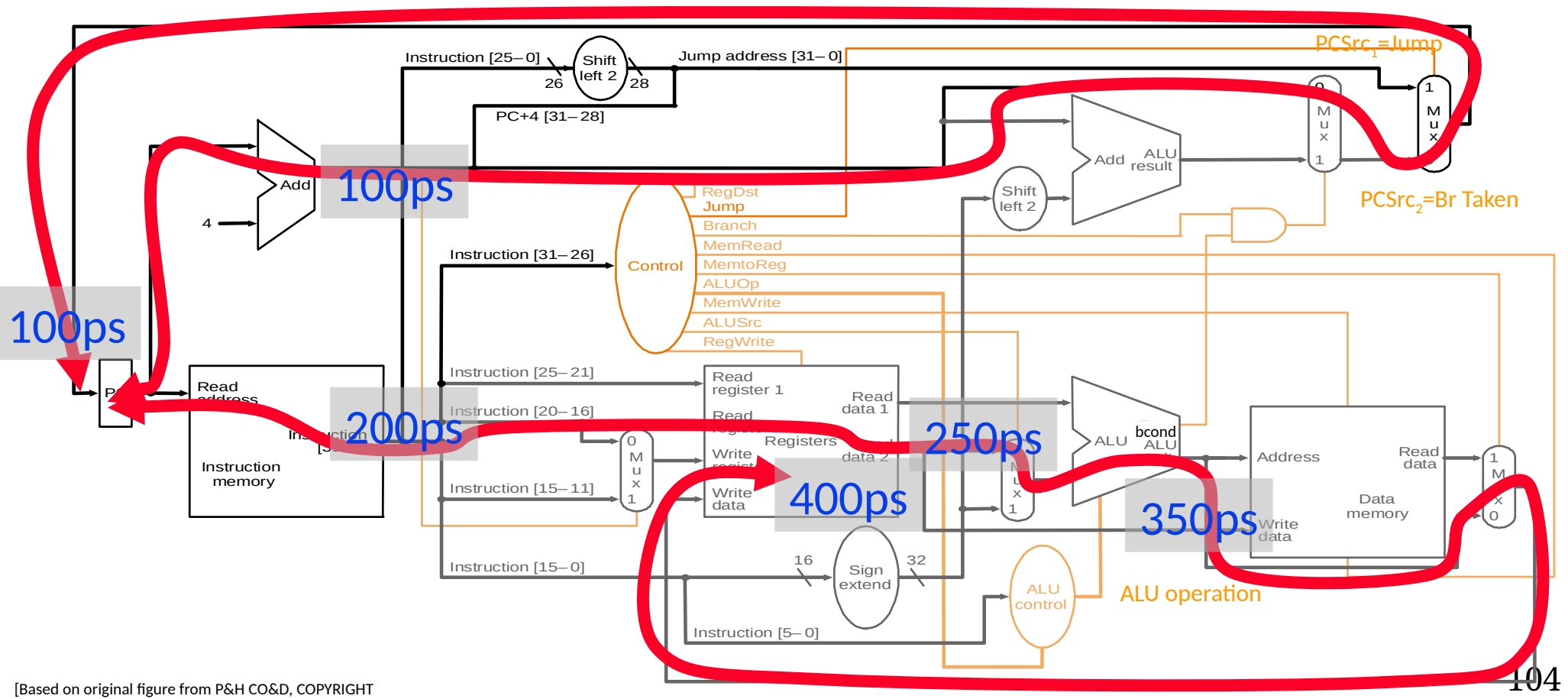


Let's Find the Critical Path



R-Type and I-Type ALU → critical path

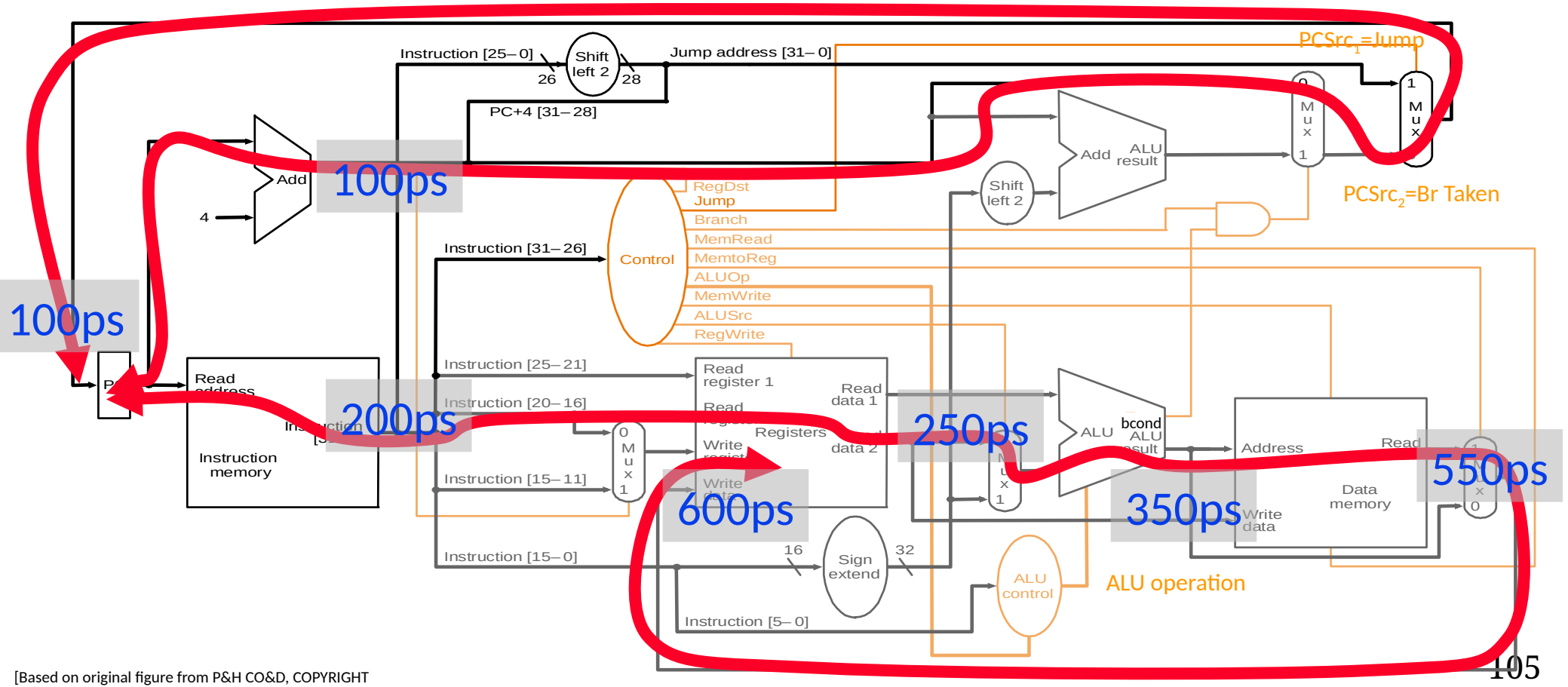
אנימציה



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

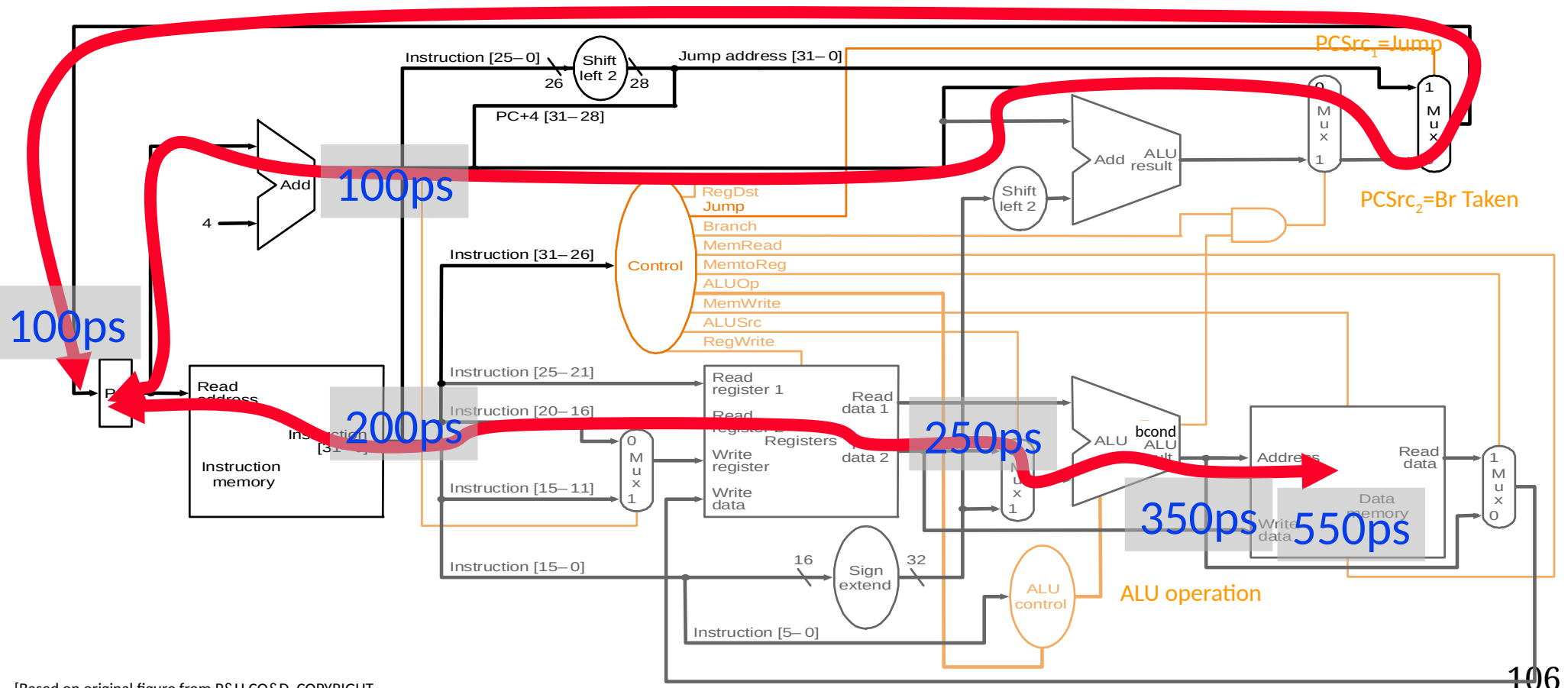
LW

אנימציה



SW

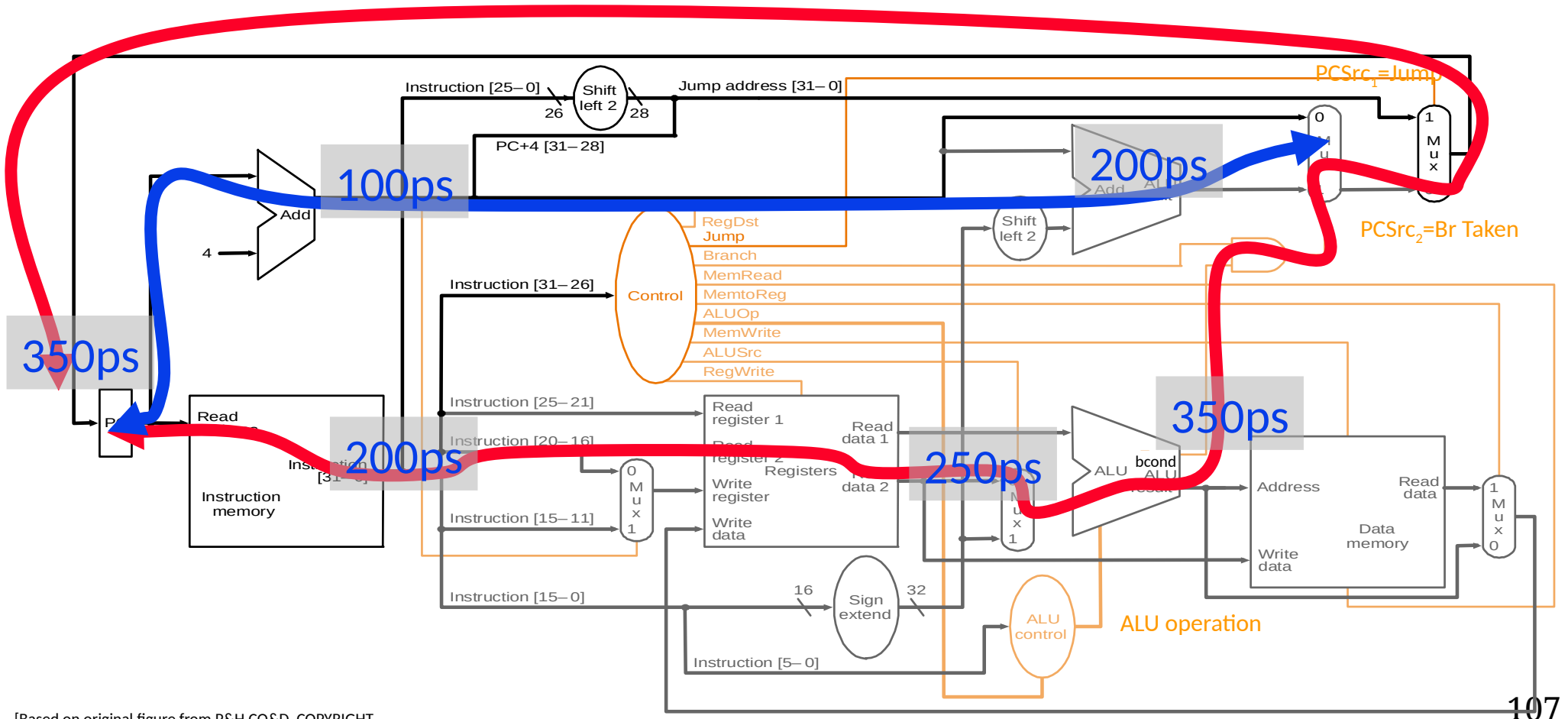
אנימציה



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

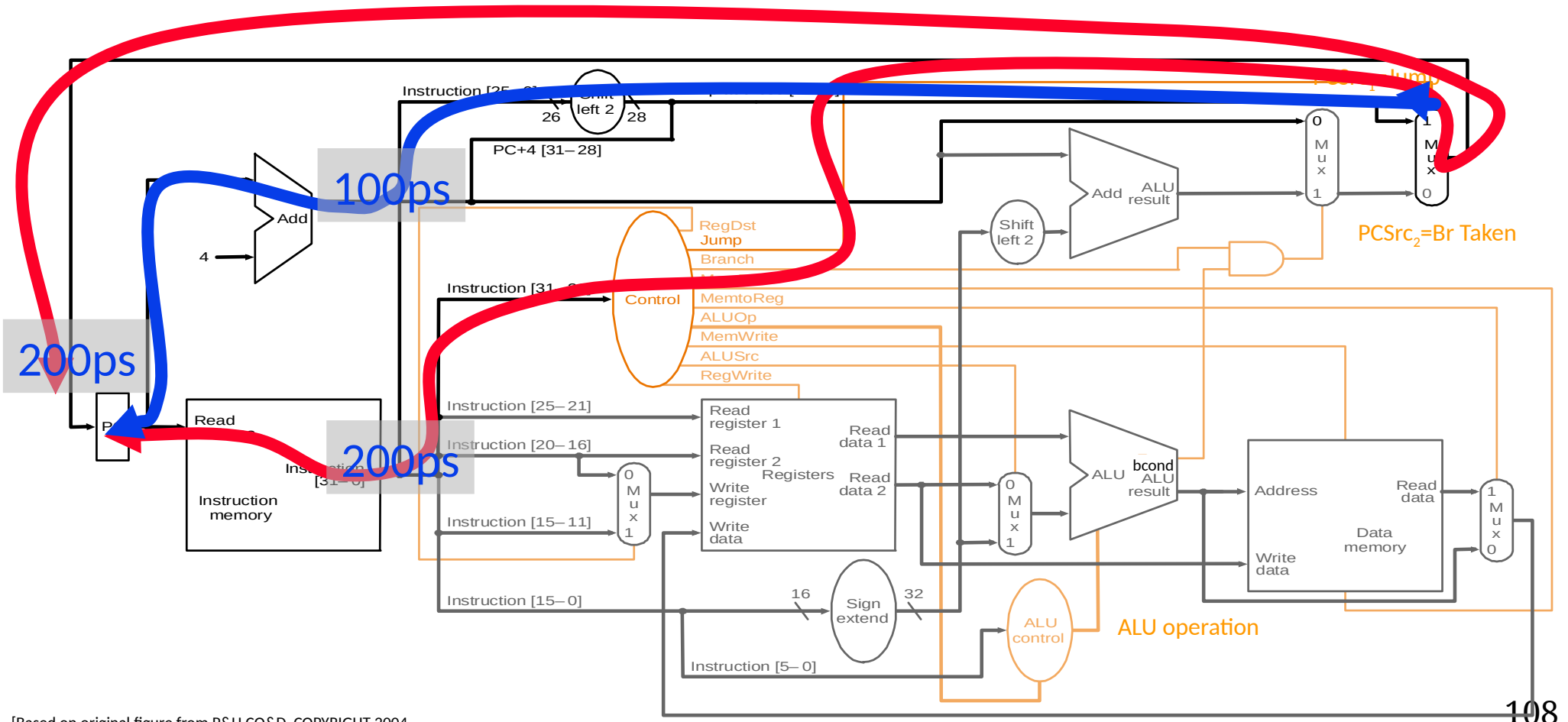
Branch Taken

אנימציה



Jump

אנימציה



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

What About Control Logic?

עד עכשיו הנחנו עיכובים רק בגלל ה- datapath מה קורה עם עיכובים בצד של הבקרה?

- How does that affect the critical path?

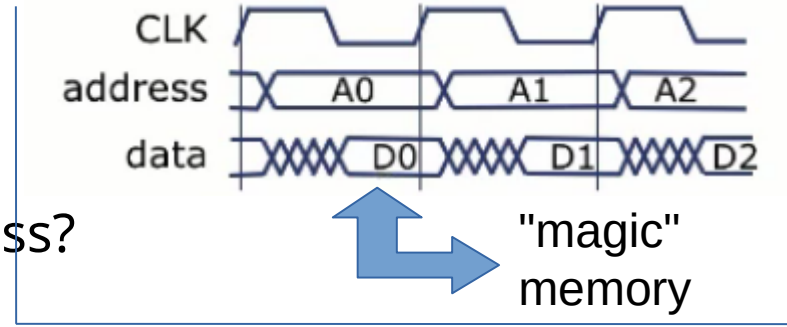
- Food for thought for you:
  - Can control logic be on the critical path? (*)
  - A note on CDC 5600: control store access too long.

(*) בהחלט כן. ראו לפני 2 שקפים עבורBranch Taken. אותות הבקרה תלויים בנתונים ונוצרות השהיות. דוגמה נוספת: אותות הבקרה שקשורים ב- cache hit/miss נמצאים על הנתיב הקריטי.

(**) Seymour Cray היה המהנדס הראשי ב- CDC. הוא ואנשיו קיצרו את ה- pipeline כדי להתגבר על בעיית זמן הגישה הארוך

What is the Slowest Instruction to Process?

גיא:



- Memory is not “magic”
- What if memory *sometimes* takes 100ms to access?
- Does it make sense to have a simple register to register add or jump to take {100ms+all else to do a memory operation}?
- And, what if you need to access memory more than once to process an instruction?
  - Which instructions need this?
  - Do you provide multiple ports to memory?

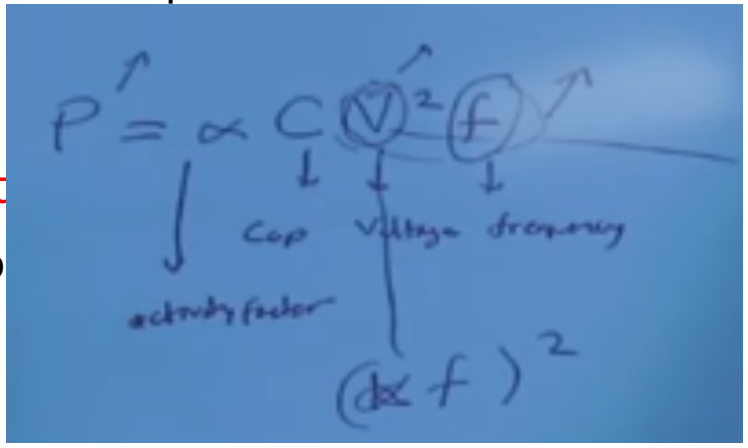
Single Cycle µArch: Complexity

- Contrived (=מועסקים)
  - All instructions run as slow as the slowest instruction
- Inefficient
  - All instructions run as slow as the slowest 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
  - Single-cycle implementation of REP MOVS (x86) or INDEX (VAX)?
- Not easy to optimize/improve performance
  - Optimizing the common case does not work (e.g. common instructions)
  - Need to optimize the worst case all the time

(Micro)architecture Design Principles

- Critical path design (Guy: optimize for performance)
  - Find and **decrease** the maximum **combinational logic delay**
  - Break a path into multiple cycles if it takes too long

למרות הרצון לאופטימיזציה של הנתיב הקריטי כדי שנוכל להעלות את התדר, לא תמיד הדבר רצוי בגלל צריכת ההספק



P is proportional to f^3

- 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
  - **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

אנימציה

- Critical path design
- Bread and butter (common case) design
- Balanced design

How does a single-cycle microarchitecture fare in light of these principles?

Aside: System Design Principles

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

- “principled design”

Frank Lloyd Wright: “architecture [...] based upon principle, and not upon precedent” = אדריכלות המבוססת על עיקרון, ולא על תקדים

Guy: **Patterson and Hennessy design principles:**

(1) Simplicity favors regularity;

(2) Make the common case fast;

(3) Smaller is faster;

(4) Good design demands good compromises;

Frank Lloyd Wright

אנימציה

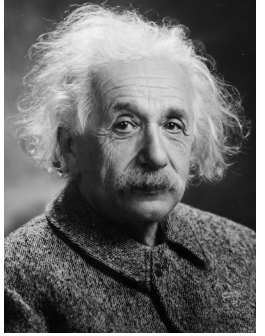
- “architecture [...] based upon principle, and not upon precedent”

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.
  - http://research.microsoft.com/pubs/68221/acrobat.pdf

Aside: One Important Principle



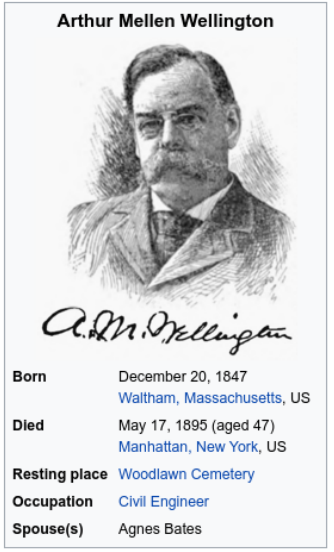
- Keep it simple

- “Everything should be made as simple as possible, but no simpler.”
  - Albert Einstein
- And, do not forget: “An engineer is a person who can do for a dime what any fool can do for a dollar.”

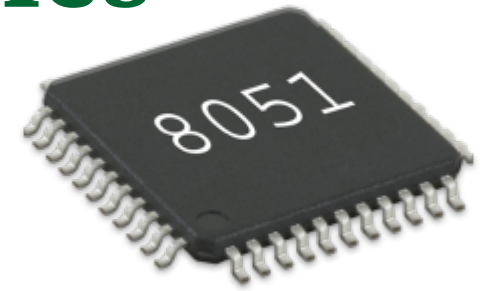
הציטוט המדויק:

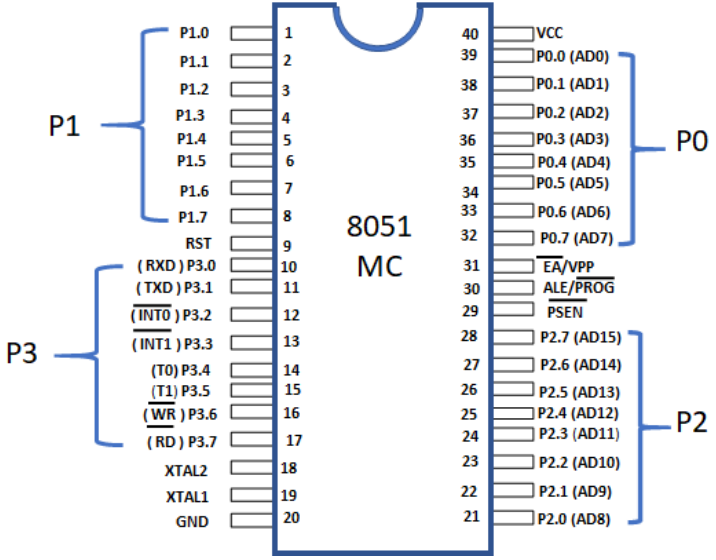
An engineer can do for a dollar“

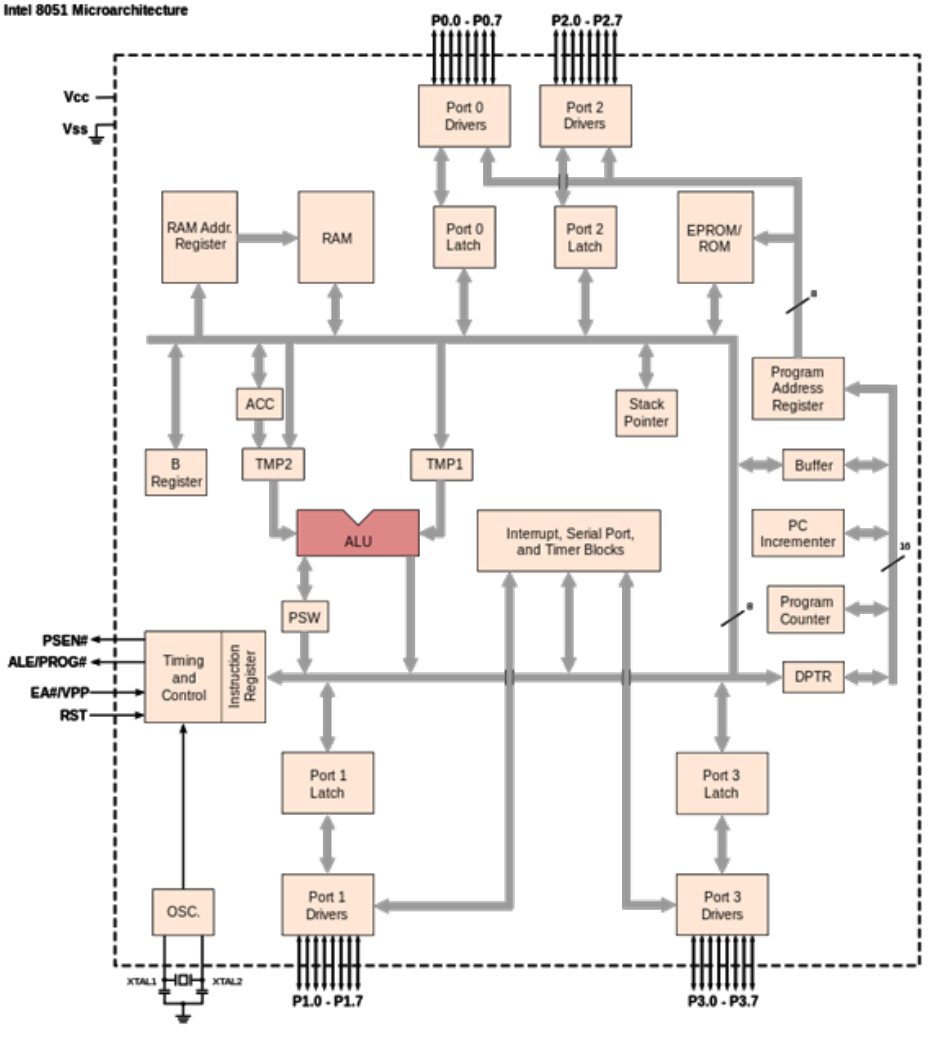
"what any fool can do for two



Single Cycle MCU Examples







ATMEL WHITE PAPER

Single Cycle 8051 Core—AT89LP Family of High Performance & Low Power Flash Microcontrollers

By David Lee, Marketing Manager
&
Ben Froemming, Design Manager

Summary

The Atmel AT89LP family of products features a single-cycle 8051 core within a highly integrated microcontroller allowing designers to achieve 6x to 12x more performance compared to classic 8051 devices. The 8051 architecture has been used in the industry for decades and remains very popular with system developers. AT89LP microcontrollers offer full binary code compatibility with the original MSC®51 instruction set, ensuring an easy migration path. The single-cycle core provides designers a unique opportunity to upgrade their application with more performance (up to 20 MIPS), less power consumption (up to 80% savings) and code size ranging from 2KB to 64KB. AT89LP microcontrollers reduce system cost and enable faster time-to-market by integrating more system features on chip.

Reference:
https://ww1.microchip.com/downloads/en/DeviceDoc/doc4088.pdf

ANALOG DEVICES

MicroConverter® Multichannel 24-/16-Bit ADCs with Embedded 62 kB Flash and Single-Cycle MCU

Data Sheet

ADuC845/ADuC847/ADuC848

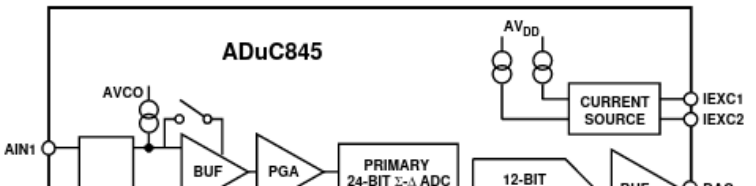
FEATURES

- High resolution Σ-Δ ADCs
  - 2 independent 24-bit ADCs on the ADuC845
  - Single 24-bit ADC on the ADuC847 and single 16-bit ADC on the ADuC848
  - Up to 10 ADC input channels on all devices
  - 24-bit no missing codes
  - 22-bit rms (19.5 bit p-p) effective resolution
  - Offset drift 10 nV/°C, gain drift 0.5 ppm/°C chop enabled
- Memory
  - 62-kbyte on-chip Flash/EE program memory
  - 4-kbyte on-chip Flash/EE data memory
  - Flash/EE, 100-year retention, 100 kcycle endurance
  - 3 levels of Flash/EE program memory security
  - In-circuit serial download (no external hardware)
  - High speed user download (5 sec)
  - 2304 bytes on-chip data RAM
- 8051-based core
  - 8051-compatible instruction set
  - High performance single-cycle core
  - 32 kHz external crystal
  - On-chip programmable PLL (12.58 MHz max)
- Power
  - Normal: 4.8 mA max at 3.6 V (core CLK = 1.57 MHz)
  - Power-down: 20 µA max with wake-up timer running
  - Specified for 3 V and 5 V operation
  - Package and temperature range:
  - 52-lead MQFP (14 mm × 14 mm), −40°C to +125°C
  - 56-lead LFCSP (8 mm × 8 mm), −40°C to +85°C

APPLICATIONS

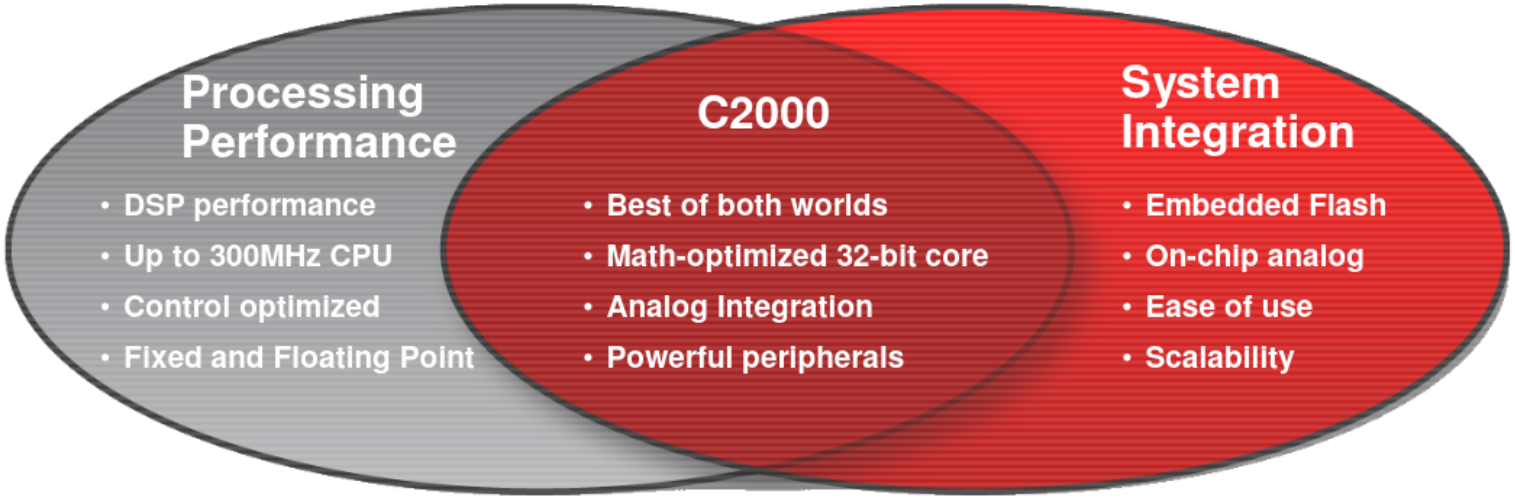
Multichannel sensor monitoring
Industrial/environmental instrumentation
Weigh scales, pressure sensors, temperature monitoring
Portable instrumentation, battery-powered systems
Data logging, precision system monitoring

FUNCTIONAL BLOCK DIAGRAM



What is C2000?

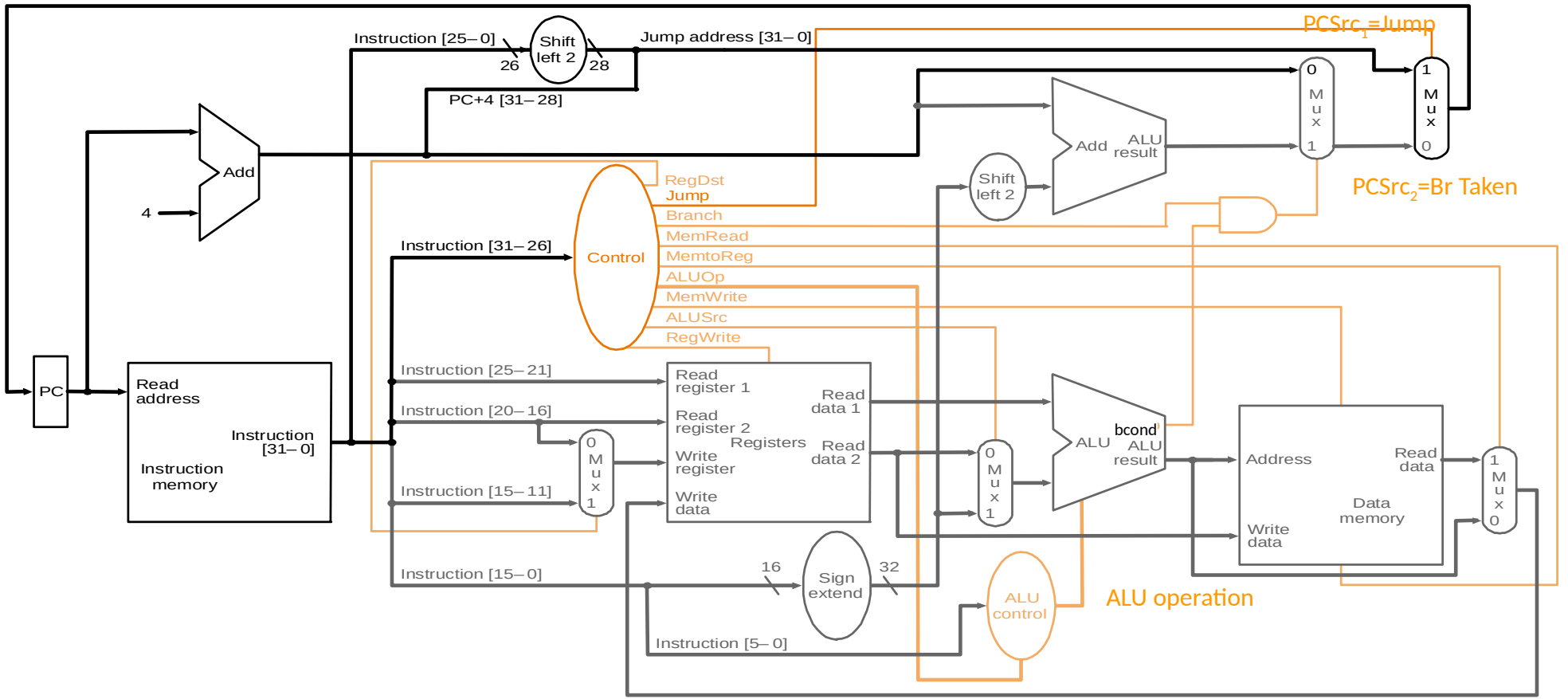
The 32-bit real-time microcontroller family



- **DSP performance within a Microcontroller architecture**
  - 40-300MHz C28x CPU
    - Built-in DSP functions
    - Single Cycle 32x32-bit MAC
  - Control Law Accelerator
  - Floating-Point Unit
  - Embedded Flash
- **Fine-tuned for real-time control**
  - Optimized core
  - Fast interrupts
  - Flexible interrupt system
- **Comprehensive Peripheral Set**
  - Best in class ADC performance
  - Flexible high resolution PWMs
  - Advanced Capture, Quadrature Encoder Interfaces
  - CAN, LIN, SPI, I2C, SCI/UART, McBSP
- **Broad portfolio of configurations**
  - 40-300 MHz
  - Fixed and Floating-point devices
  - 32-512KB of Flash
  - From sub $2 to $20
  - Software compatibility across C2000 family

Single Cycle - סיכום

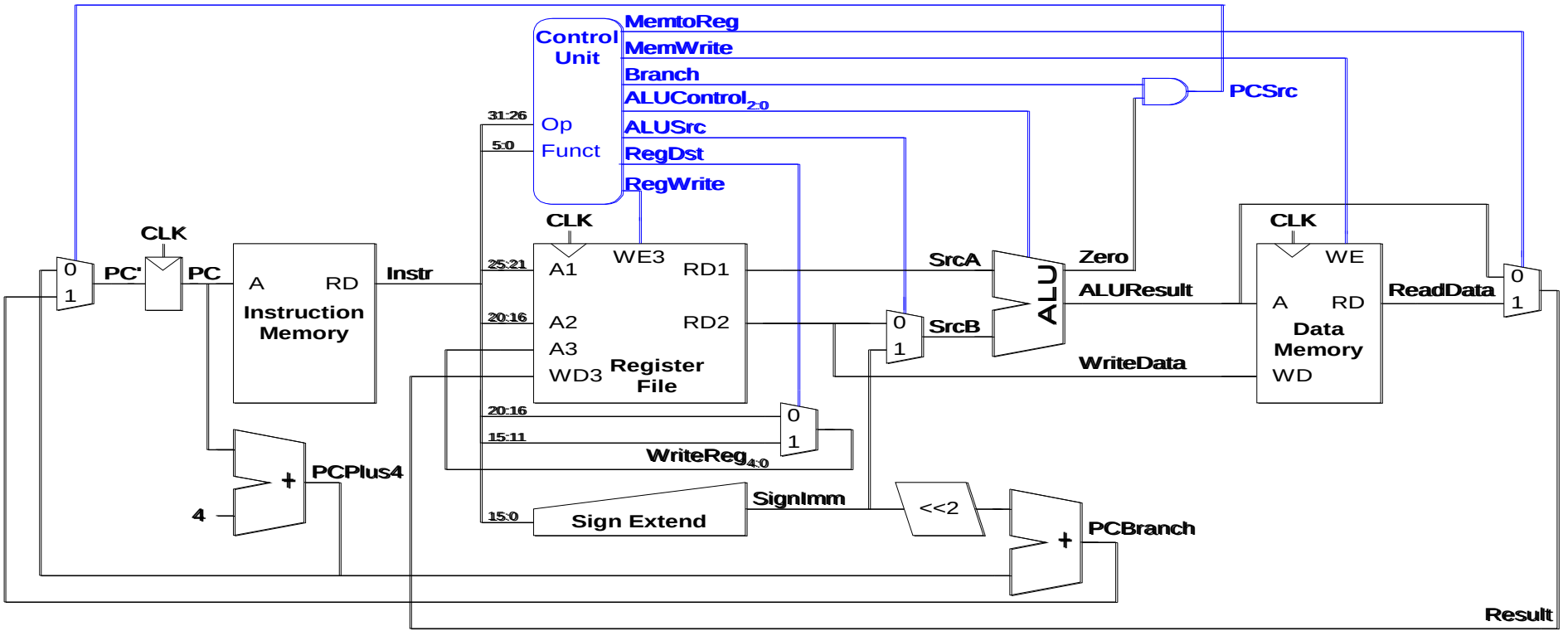
Single-Cycle µarch I (H&P עפ"י) - MIPS



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

JAL, JR, JALR omitted

Single-Cycle µarch II (H&H תצורה מעט שונה עפ"י)



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

הנושא הבא:

נעבור עתה לפרק 7 מתוך H&H, נתמקד במעבד RISC-V ונרחיב ונעסוק גם ב-Multi-Cycle Microarchitectures

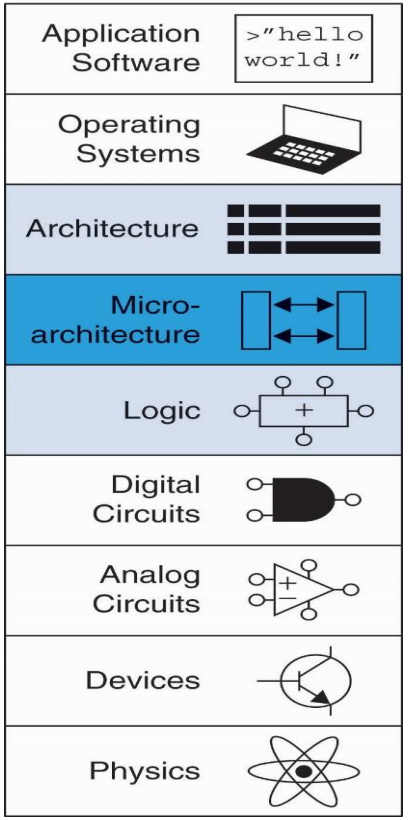
Digital Design & Computer Architecture

Sarah Harris & David Harris

Chapter 7: Microarchitecture

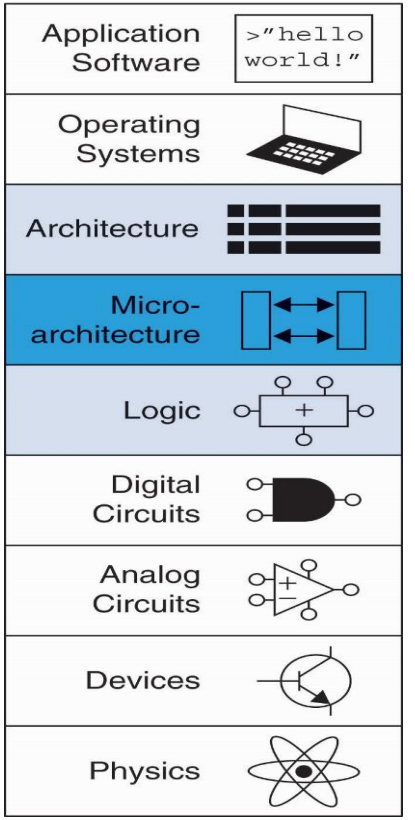
Chapter 7 :: Topics

- **Introduction**
- **Performance Analysis**
- **Single-Cycle Processor**
- **Multicycle Processor**
- **Pipelined Processor**
- **Advanced Microarchitecture**



Introduction

- **Microarchitecture:** how to implement an architecture in hardware
- Processor:
  - **Datapath:** functional blocks
  - **Control:** control signals



Microarchitecture

- **Multiple implementations** for a single architecture:
  - **Single-cycle:** Each instruction executes in a single cycle
  - **Multicycle:** Each instruction is broken up into series of shorter steps
  - **Pipelined:** Each instruction broken up into series of steps & multiple instructions execute at once

Processor Performance

- **Program execution time**

Execution Time = (#instructions)(cycles/instruction)(seconds/cycle)

- **Definitions:**
  - CPI: Cycles/instruction
  - clock period: seconds/cycle
  - IPC: instructions/cycle = IPC
- **Challenge is to satisfy constraints of:**
  - Cost
  - Power
  - Performance

RISC-V Processor

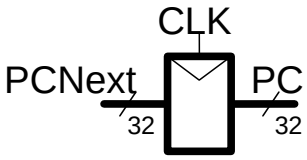
- Consider **subset** of RISC-V instructions:
  - **R-type ALU instructions:**
    - `add`, `sub`, `and`, `or`, `slt`
  - **Memory instructions:**
    - `lw`, `sw`
  - **Branch instructions:**
    - `beq`

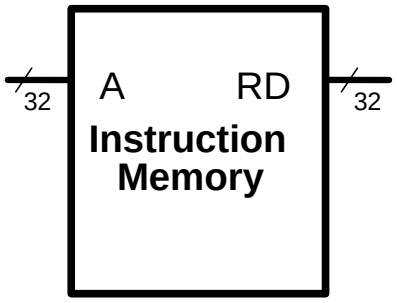
Architectural State Elements

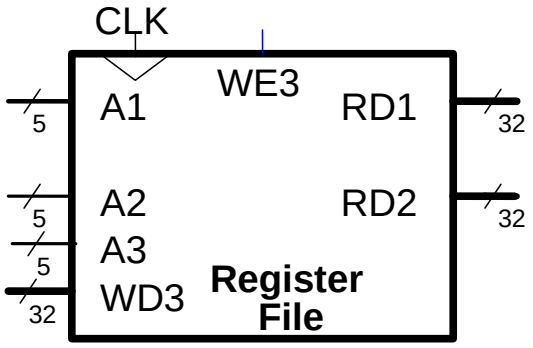
Determines everything about a processor:

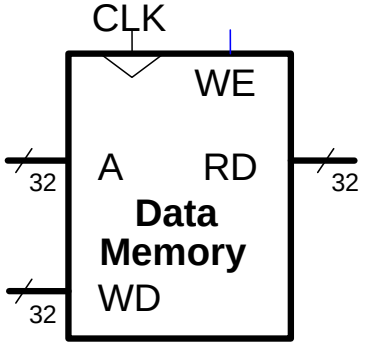
- **Architectural state:**
  - 32 registers
  - PC
  - Memory

RISC-V Architectural State Elements









Chapter 7: Microarchitecture

Single-Cycle RISC-V Processor

Single-Cycle RISC-V Processor

- Datapath
- Control

Example Program

- Design datapath
- View example program executing

Example Program:

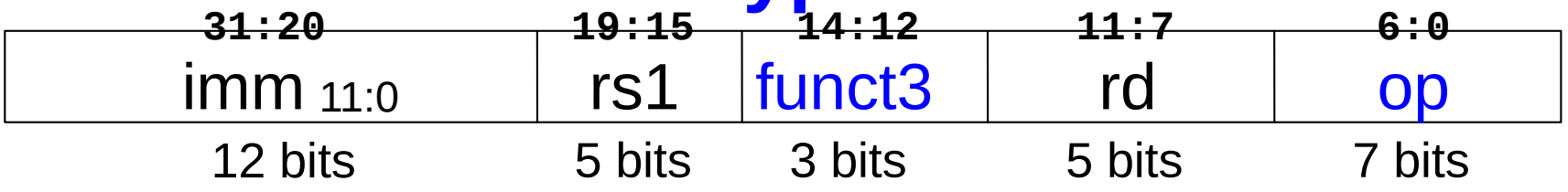
Address	Instruction	Type	Fields	Machine Language
			imm 11:0 rs1 f3 rd op	
0x1000	L7: lw x6, -4(x9)	I	111111111100 01001 010 00110 0000011	FFC4A303
			imm 11:5 rs2 rs1 f3 imm 4:0 op	
0x1004	sw x6, 8(x9)	S	0000000 00110 01001 010 01000 0100011	0064A423
			funct7 rs2 rs1 f3 rd op	
0x1008	or x4, x5, x6	R	0000000 00110 00101 110 00100 0110011	0062E233
			imm 12,10:5 rs2 rs1 f3 imm 4:1,11 op	
0x100C	beq x4, x4, L7	B	1111111 00100 00100 000 10101 1100011	FE420AE3

Single-Cycle RISC-V Processor

- **Datapath:** start with `lw` instruction
- **Example:** `lw x6, -4(x9)`

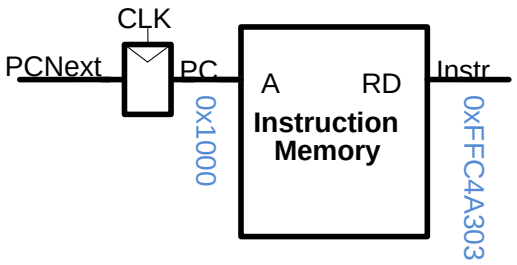
`lw rd, imm(rs1)`

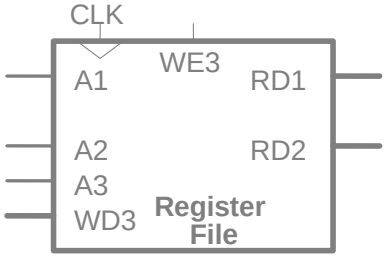
I-Type

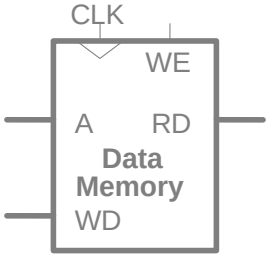


Single-Cycle Datapath: `lw` fetch

STEP 1: Fetch instruction



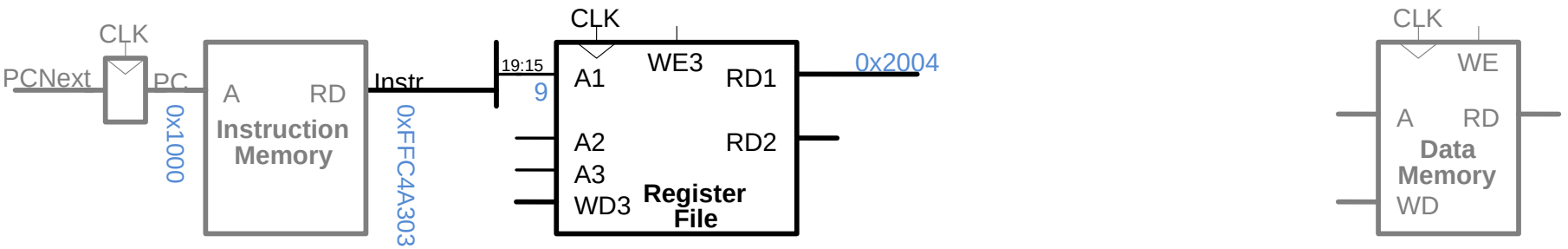




Address	Instruction	Type	Fields					Machine Language
			$\text{imm}_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Single-Cycle Datapath: `lw` Reg Read

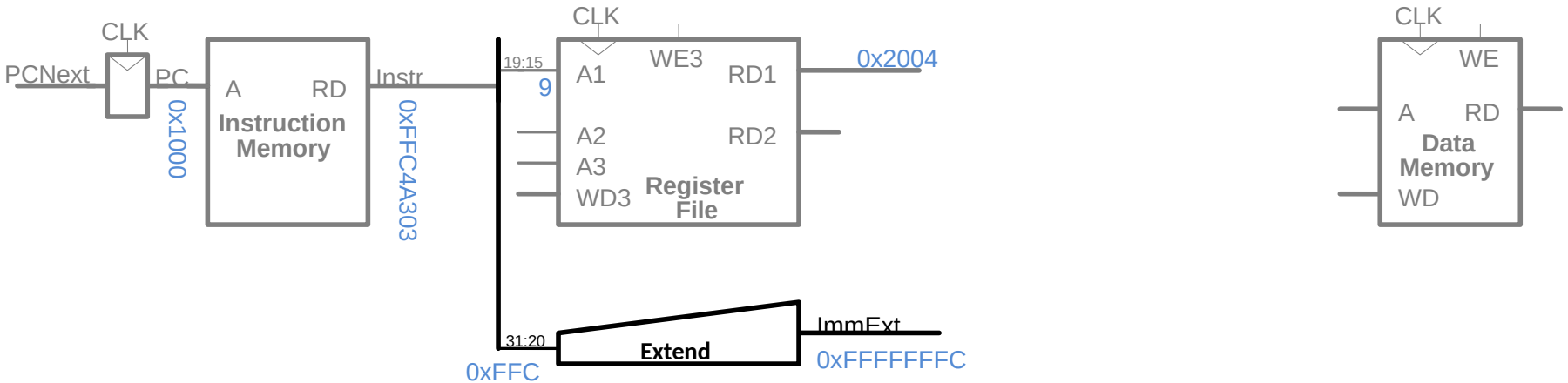
STEP 2: Read source operand (**rs1**) from RF



Address	Instruction	Type	Fields					Machine Language
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Single-Cycle Datapath: `lw` Immediate

STEP 3: Extend the immediate

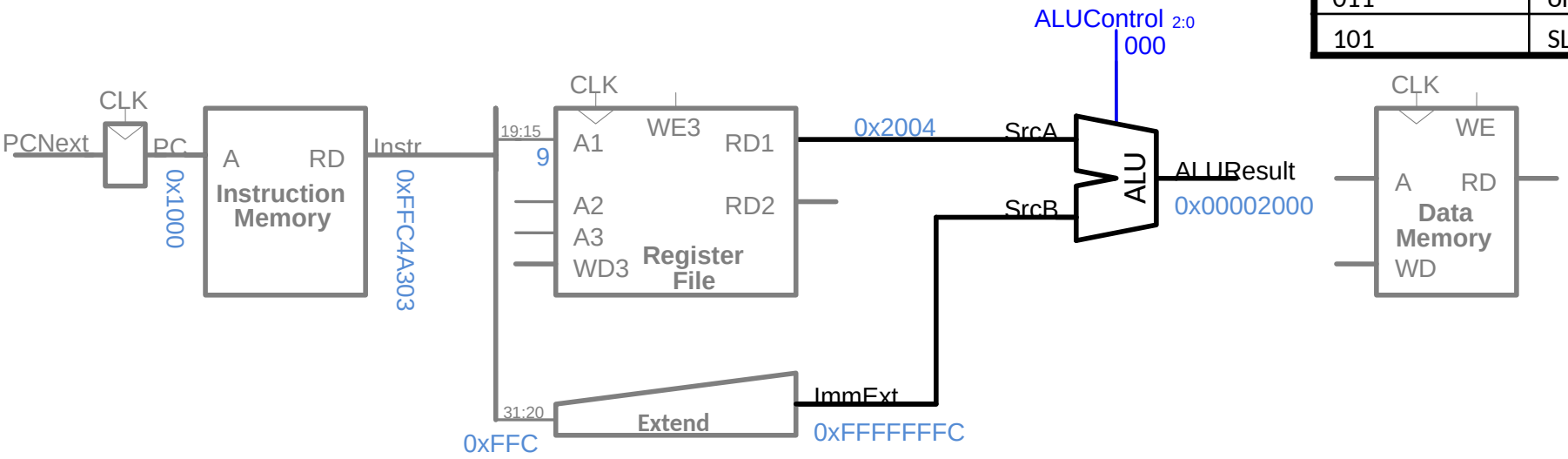


Address	Instruction	Type	Fields					Machine Language
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Single-Cycle Datapath: lw Address

STEP 4: Compute the memory address

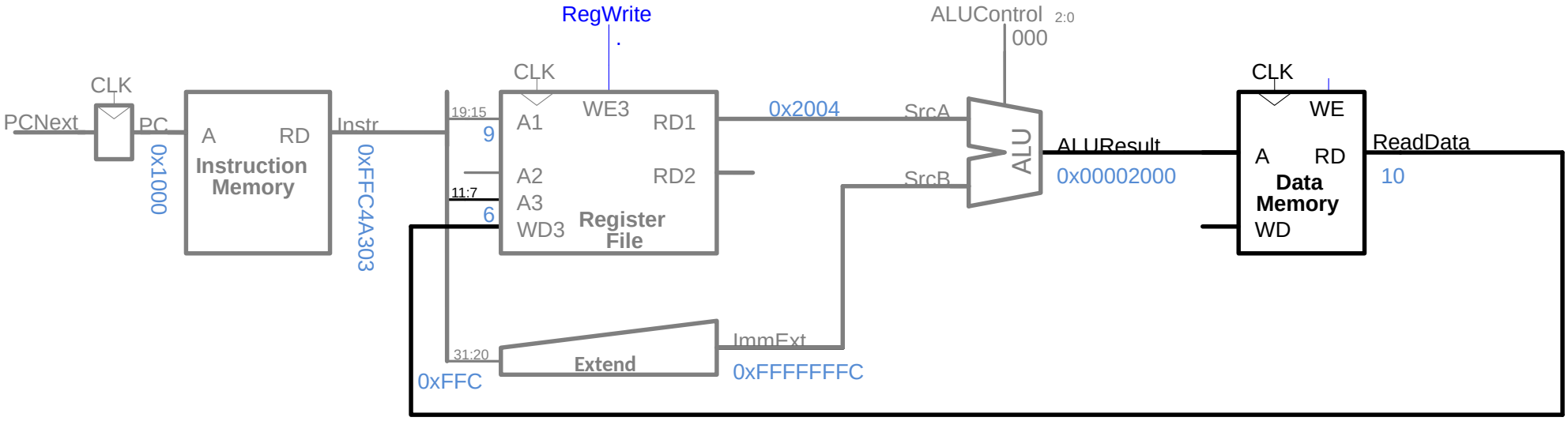
ALUControl 2:0	Function
000	add
001	subtract
010	and
011	or
101	SLT



Address	Instruction	Type	Fields					Machine Language
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Single-Cycle Datapath: lw Mem Read

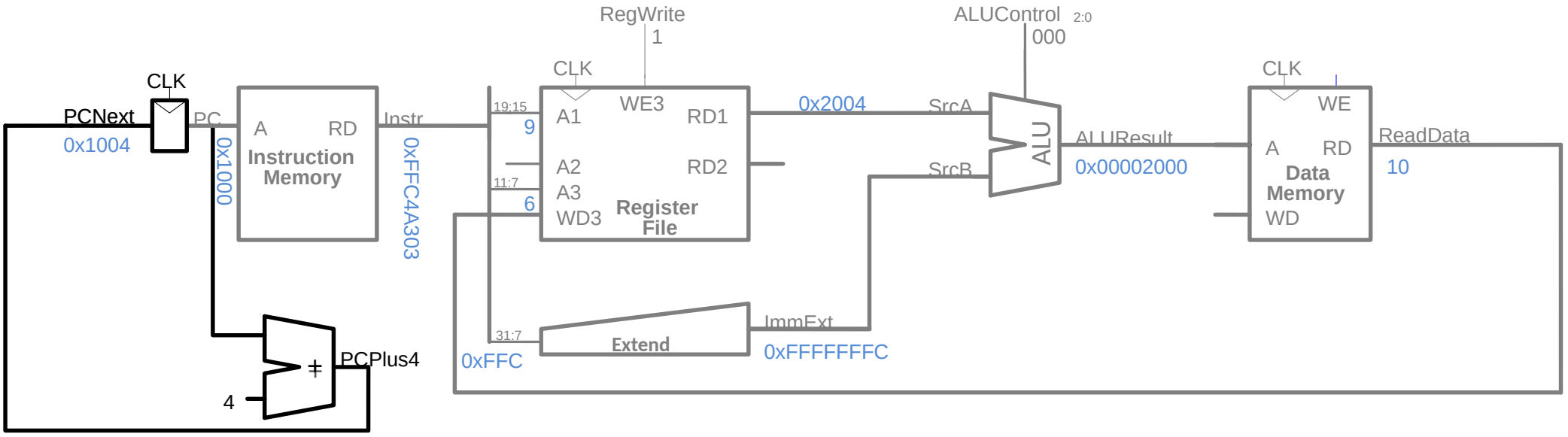
STEP 5: Read data from memory and write it back to register file



Address	Instruction	Type	Fields					Machine Language
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Single-Cycle Datapath: PC Increment

STEP 6: Determine address of next instruction



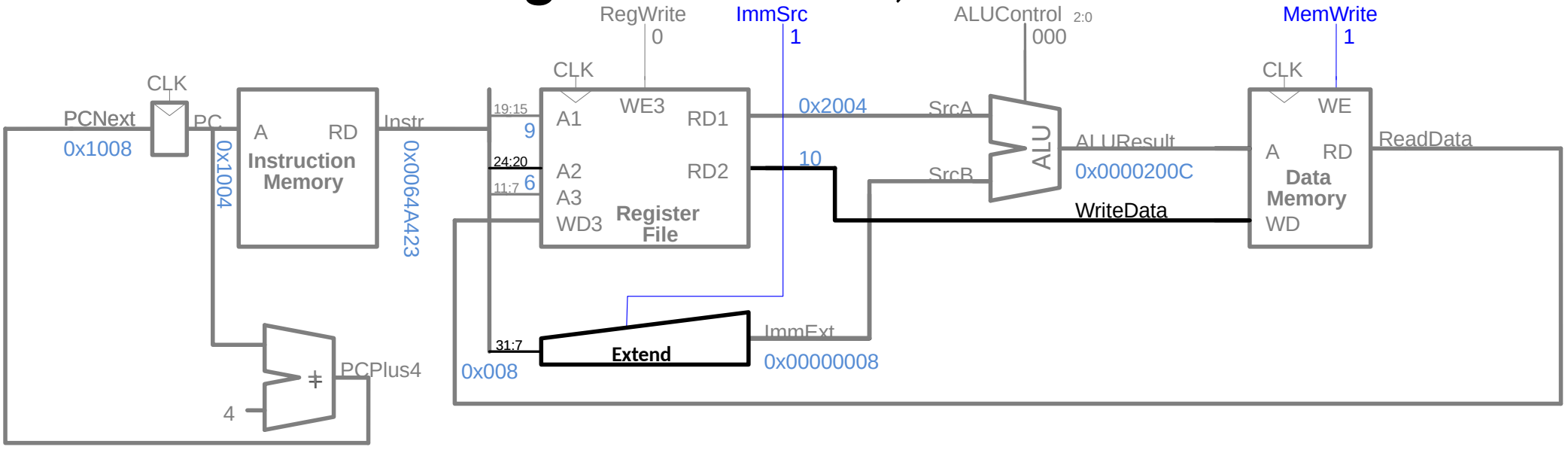
Address	Instruction	Type	Fields					Machine Language
			imm 11:0	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Chapter 7: Microarchitecture

Single-Cycle Datapath: Other Instructions

Single-Cycle Datapath: sw

- **Immediate:** now in {instr[31:25], instr[11:7]}
- **Add control signals:** ImmSrc, MemWrite

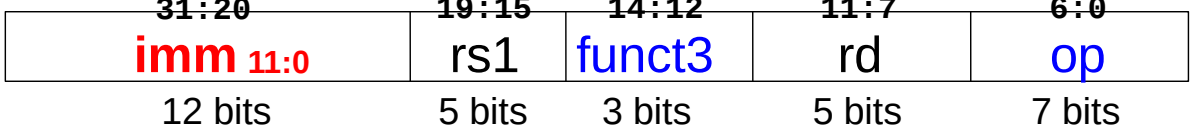


Address	Instruction	Type	Fields						Machine Language
			$imm_{11:5}$	rs2	rs1	f3	$imm_{4:0}$	op	
0x1004	sw x6, 8(x9)	S	0000000	00110	01001	010	01000	0100011	0064A423

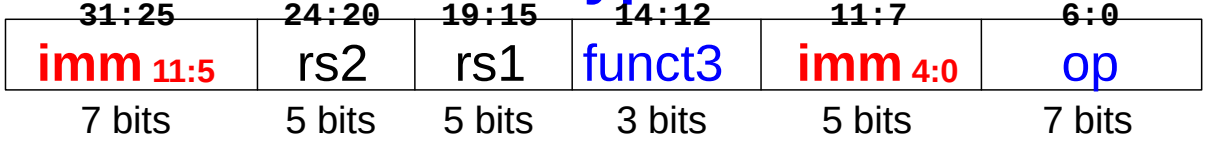
Single-Cycle Datapath: Immediate

ImmSrc	ImmExt	Instruction Type
0	{{20{instr[31]}}, **instr[31:20]**}	I-Type
1	{{20{instr[31]}}, **instr[31:25], instr[11:7]**}	S-Type

I-Type

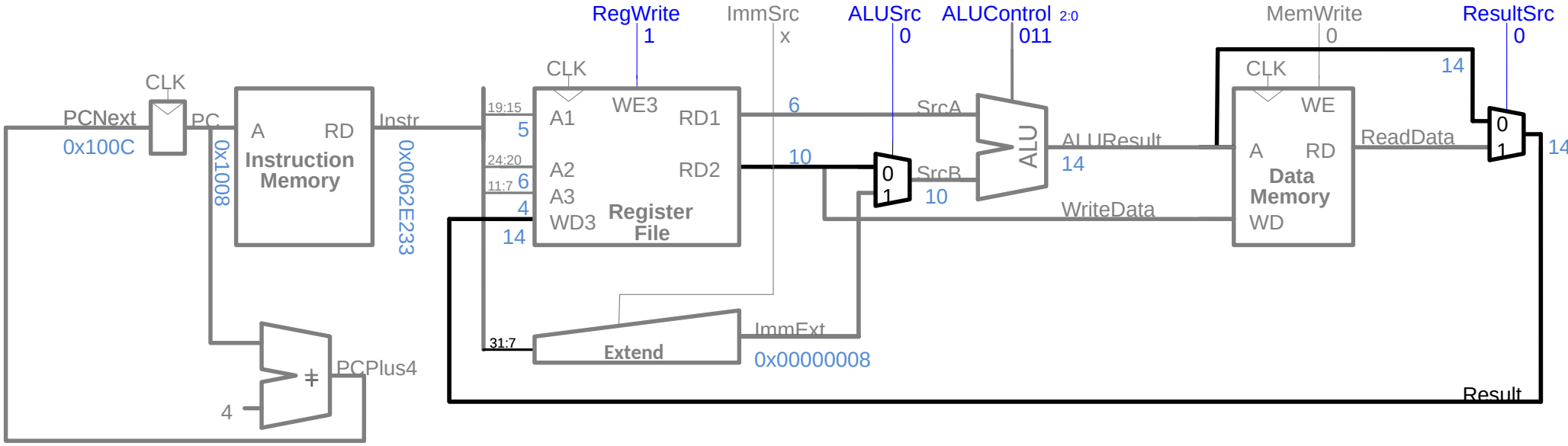


S-Type



Single-Cycle Datapath: R-type

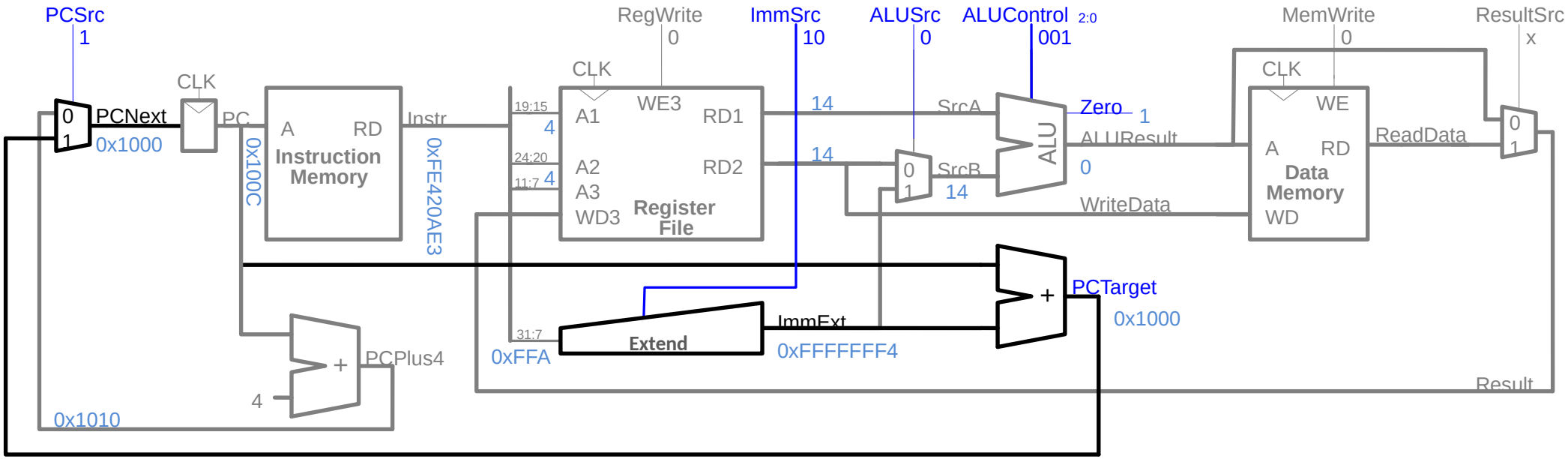
- Read from **rs1** and **rs2** (instead of **imm**)
- Write *ALUResult* to **rd**



Address	Instruction	Type	Fields						Machine Language
			funct7	rs2	rs1	f3	rd	op	
0x1008	or x4, x5, x6	R	0000000	00110	00101	110	00100	0110011	0062E233

Single-Cycle Datapath: beq

Calculate **target address:** PCTarget = PC + imm

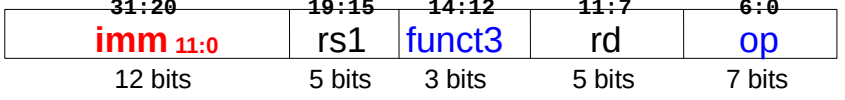


Address	Instruction	Type	Fields						Machine Language
			imm 12,10:5	rs2	rs1	f3	imm 4:1,11	op	
0x100C	beq x4, x4, L7	B	1111111	00100	00100	000	10101	1100011	FE420AE3

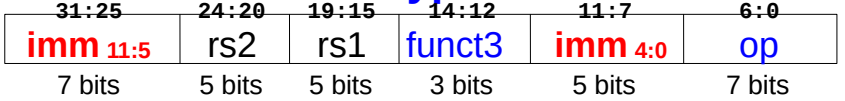
Single-Cycle Datapath: ImmExt

ImmSrc $_{1:0}$	ImmExt	Instruction Type
00	{{20{instr[31]}}, **instr[31:20]**}	I-Type
01	{{20{instr[31]}}, **instr[31:25], instr[11:7]**}	S-Type
10	{{19{instr[31]}}, **instr[31], instr[7], instr[30:25], instr[11:8], 1'b0**}	B-Type

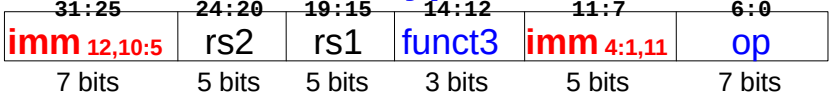
I-Type



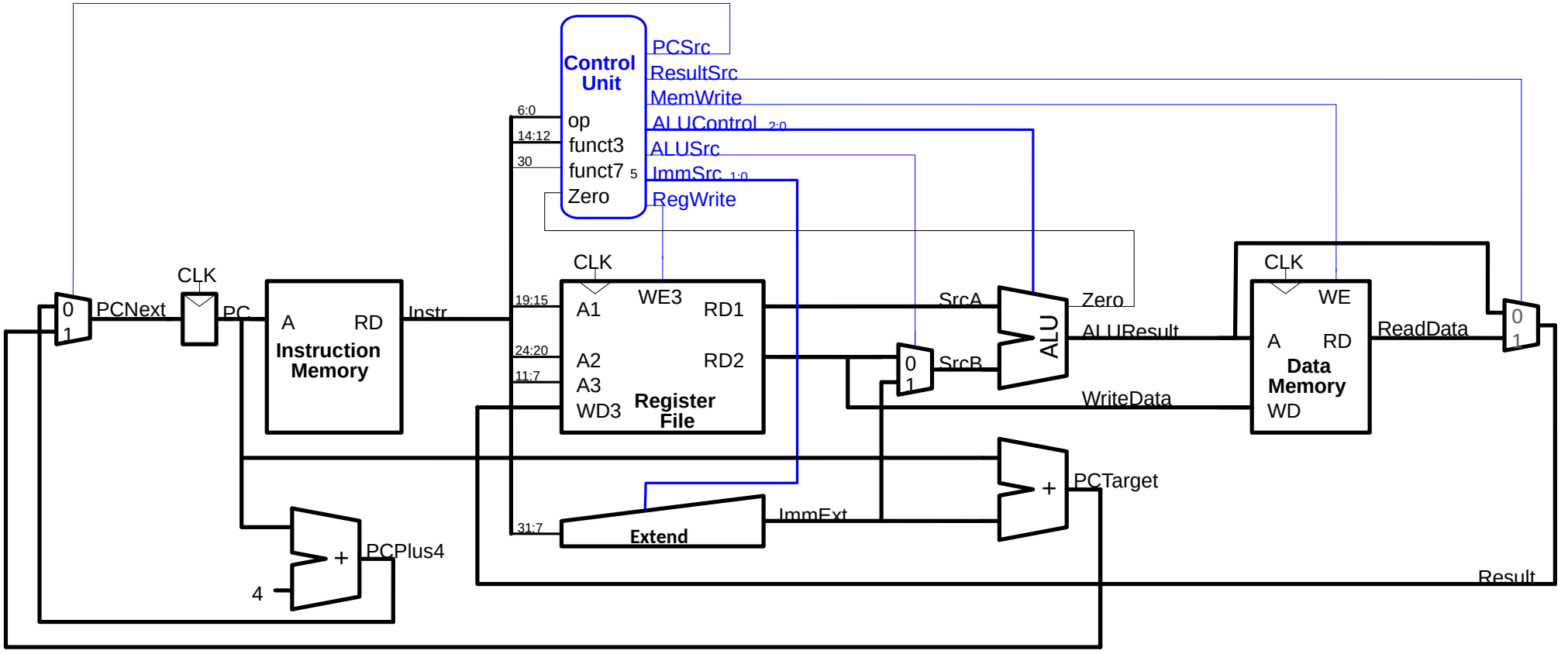
S-Type



B-Type



Single-Cycle RISC-V Processor

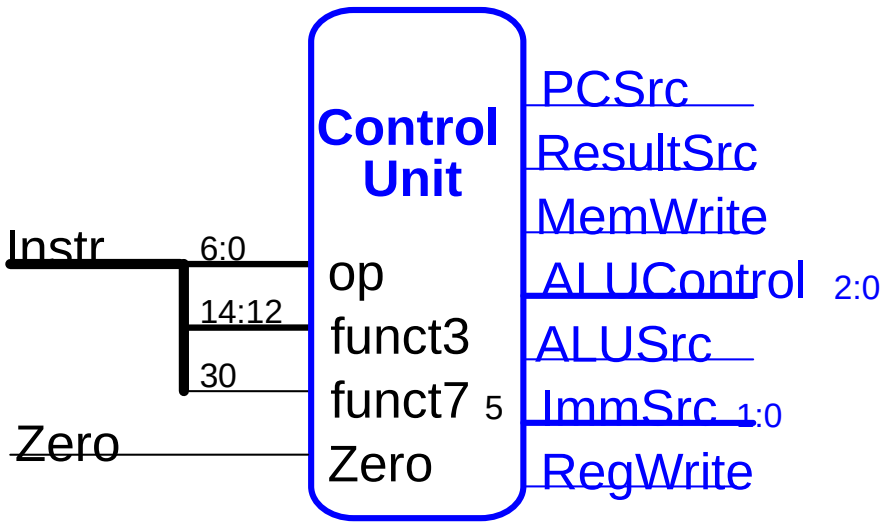


Chapter 7: Microarchitecture

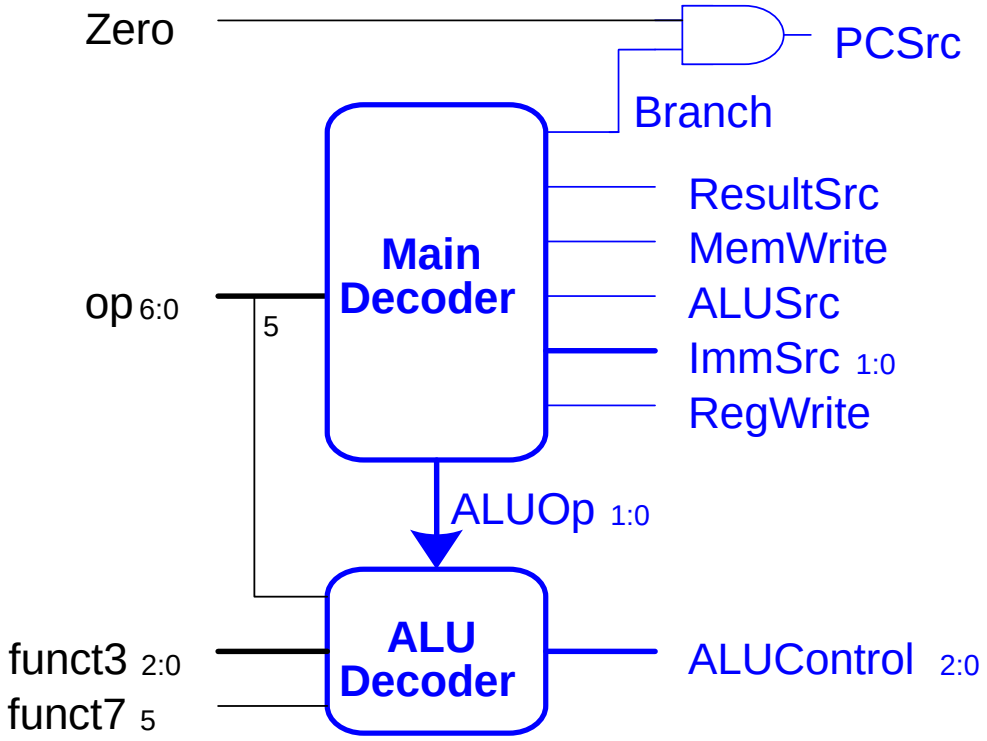
Single-Cycle Control

Single-Cycle Control

High-Level View

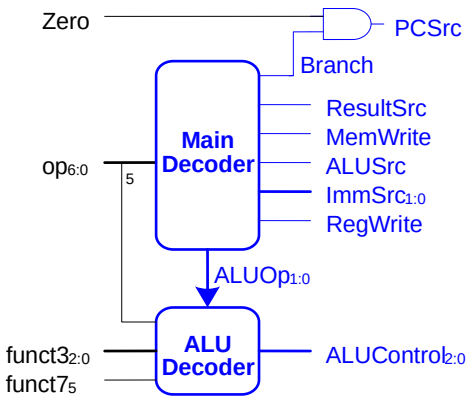


Low-Level View



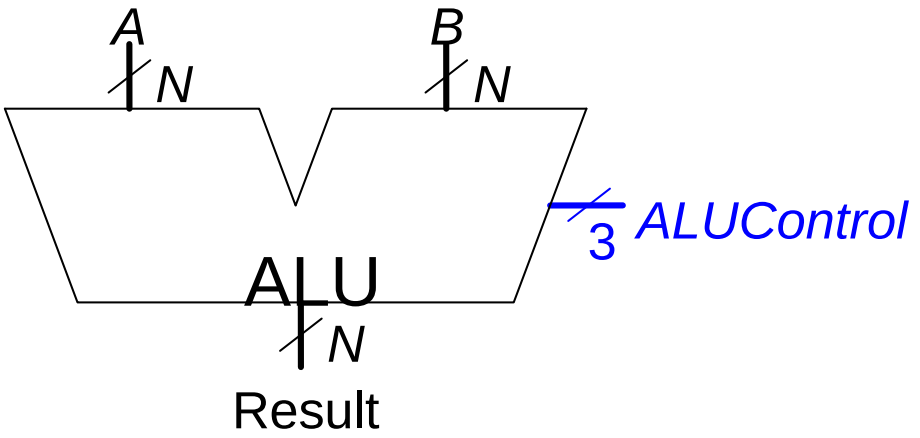
Single-Cycle Control: Main Decoder

op	Instr.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
3	`lw`							
35	`sw`							
51	R-type							
99	`beq`							



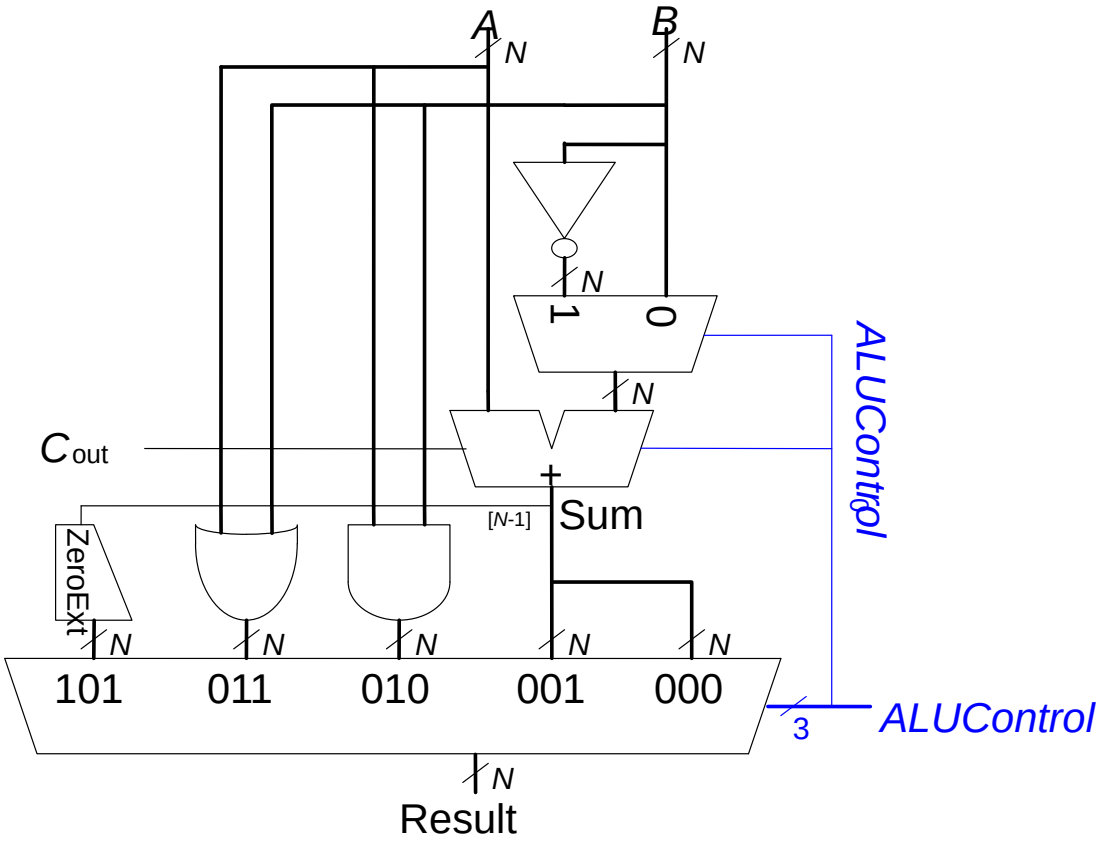
Review: ALU

ALUControl $_{2:0}$	Function
000	add
001	subtract
010	and
011	or
101	SLT

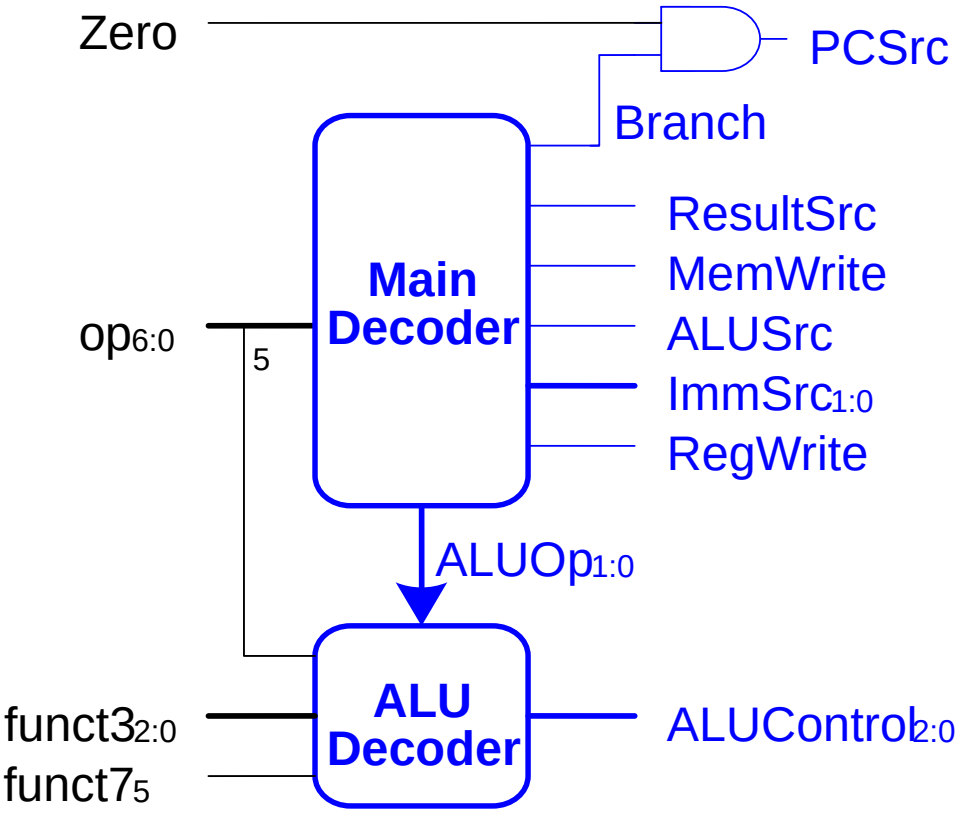


Review: ALU

ALUControl $_{2:0}$	Function
000	add
001	subtract
010	and
011	or
101	SLT

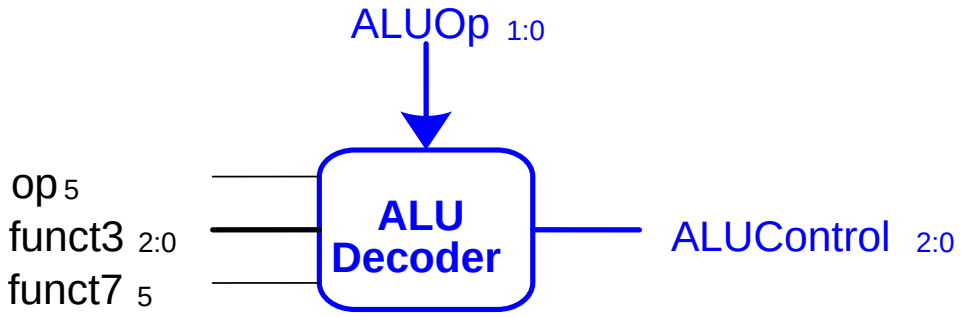


Single-Cycle Control: ALU Decoder



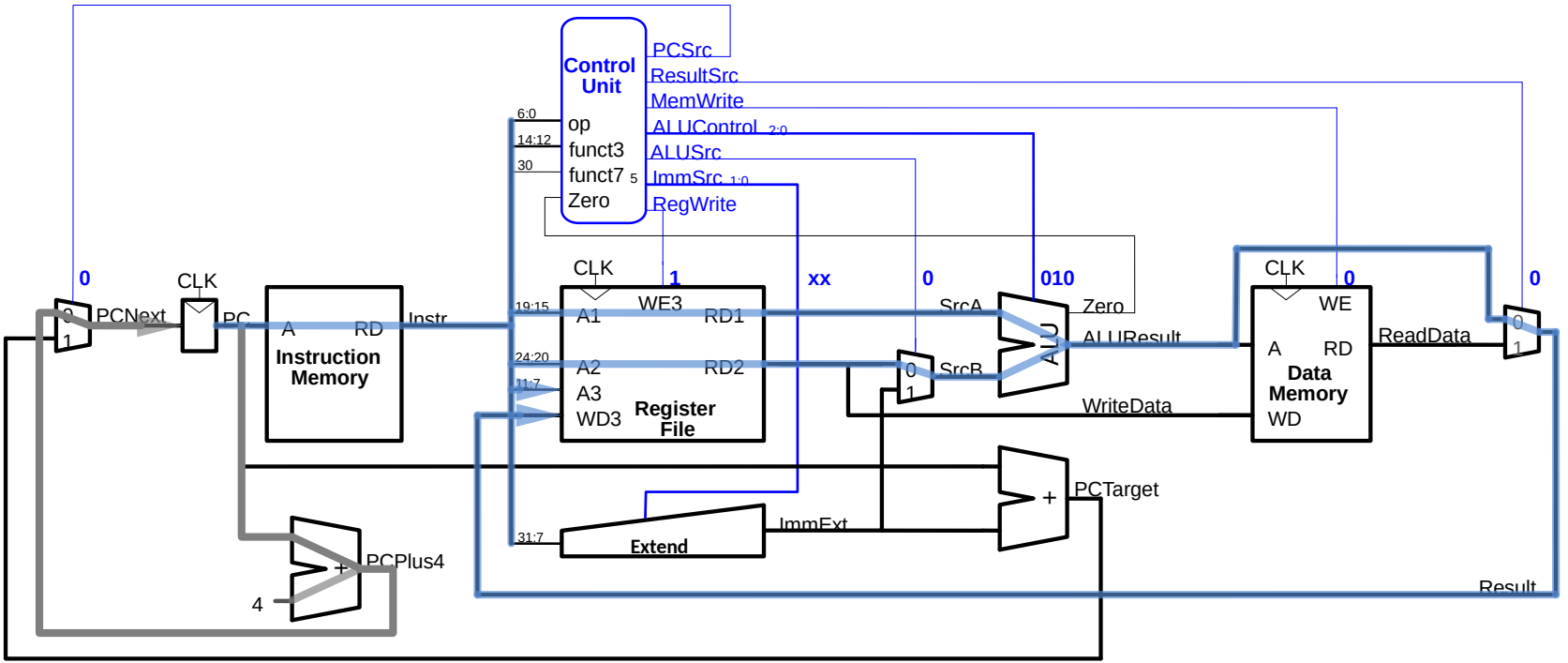
Single-Cycle Control: ALU Decoder

ALUOp	funct3	op $_5$, funct7 $_5$	Instruction	*ALUControl* $_{2:0}$
00	x	x	`lw, sw`	000 (add)
01	x	x	`beq`	001 (subtract)
10	000	00, 01, 10	`add`	000 (add)
	000	11	`sub`	001 (subtract)
	010	x	`slt`	101 (set less than)
	110	x	`or`	011 (or)
	111	x	`and`	010 (and)



Example: and

op	Instruct	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
51	**R-type**	1	XX	0	0	0	0	10



and x5, x6, x7

Chapter 7: Microarchitecture

Extending the Single-Cycle Processor

Extended Functionality: I-Type ALU

Enhance the single-cycle processor to handle **I-Type ALU instructions**: `addi`, `andi`, `ori`, and `slti`

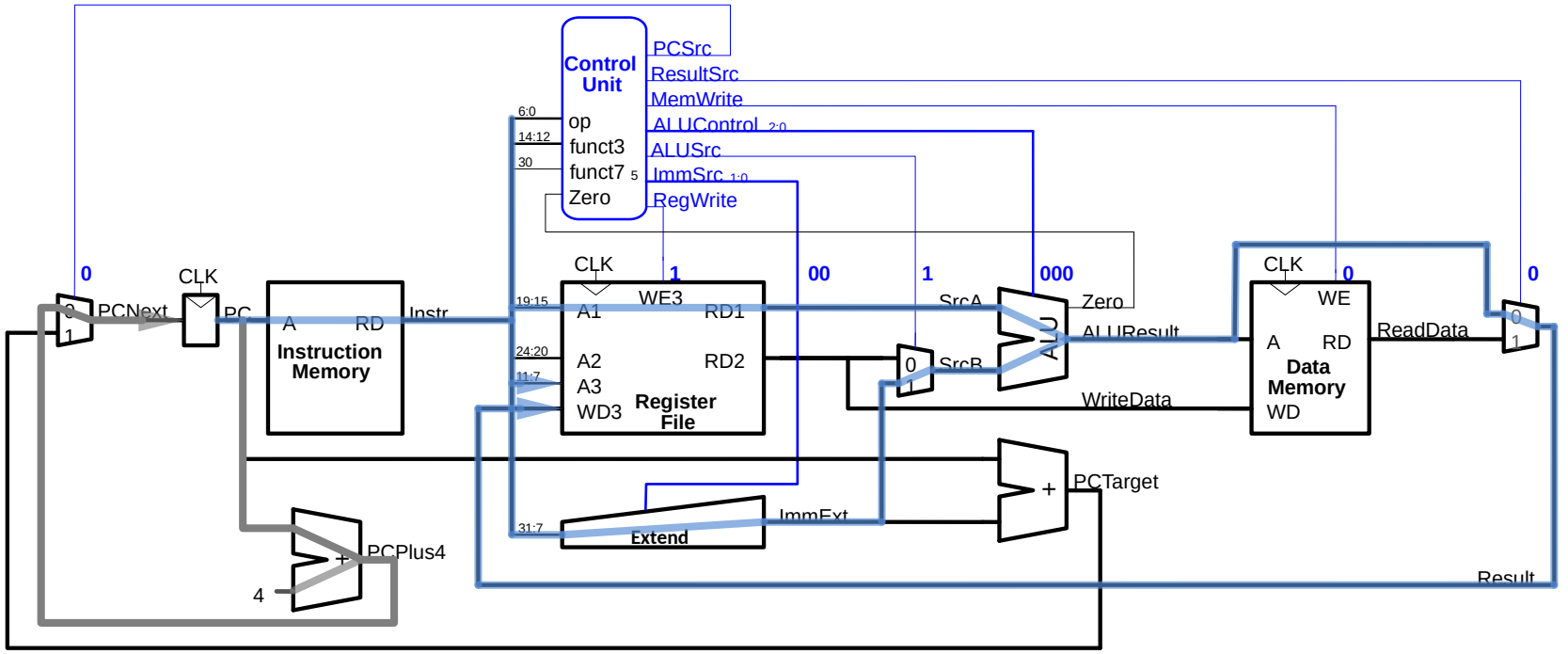
- **Similar to R-type** instructions
- But **second source** comes from **immediate**
- Change ***ALUSrc*** to select the immediate
- And ***ImmSrc*** to pick the correct immediate

Extended Functionality: I-Type ALU

op	Instruct.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
3	**lw**	1	00	1	0	1	0	00
35	**sw**	0	01	1	1	X	0	00
51	**R-type**	**1**	**XX**	**0**	**0**	**0**	**0**	**10**
99	**beq**	0	10	0	0	X	1	01
19	**I-type**	**1**	**00**	**1**	**0**	**0**	**0**	**10**

Extended Functionality: addi

op	Instruct.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
19	I-type	1	00	1	0	0	0	10



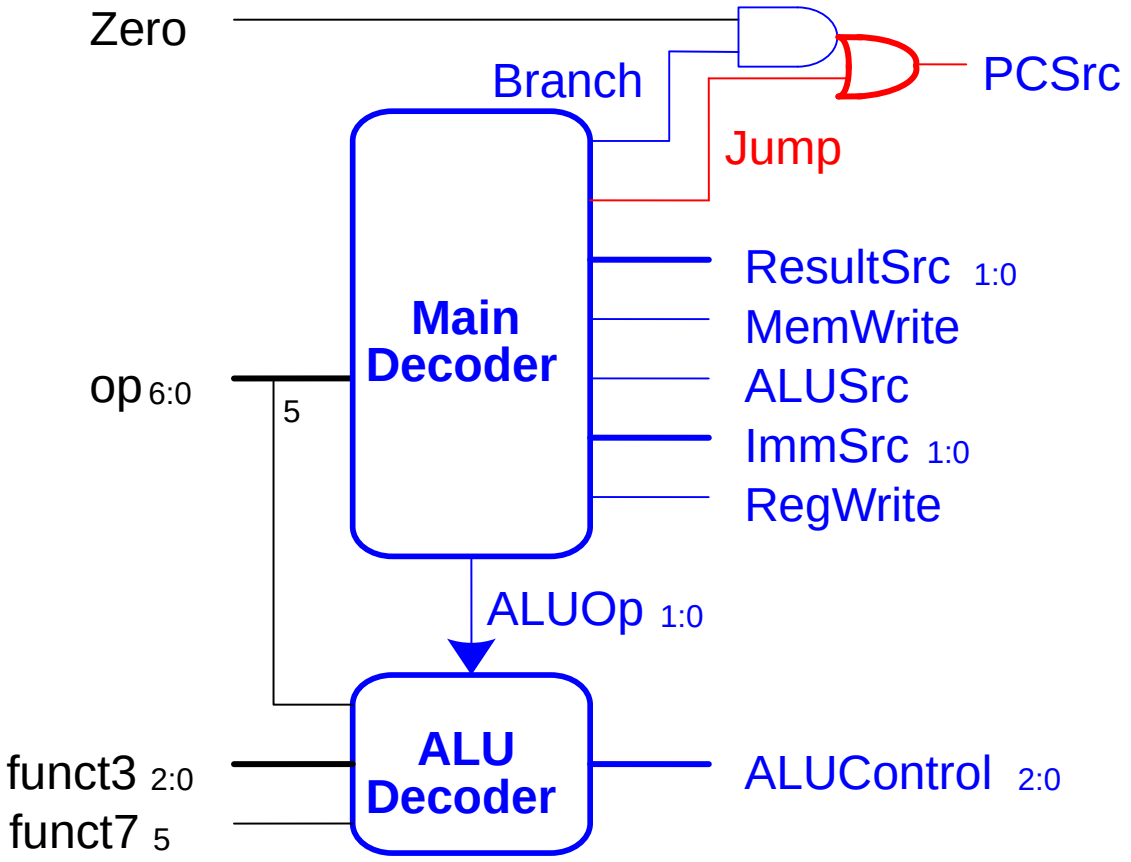
addi x5, x6, -33

Extended Functionality: `jal`

Enhance the single-cycle processor to handle `jal`

- **Similar to `beq`**
- But jump is **always taken**
  - *PCSrc* should be 1
- **Immediate format** is different
  - Need a new *ImmSrc* of 11
- And `jal` must **compute PC+4** and **store in `rd`**
  - Take PC+4 from adder through ResultMux

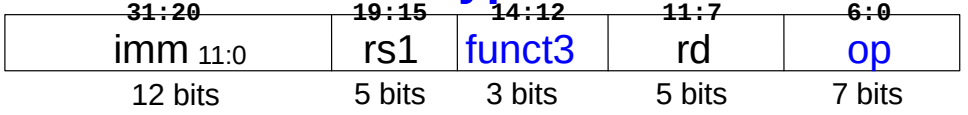
Extended Functionality: jal



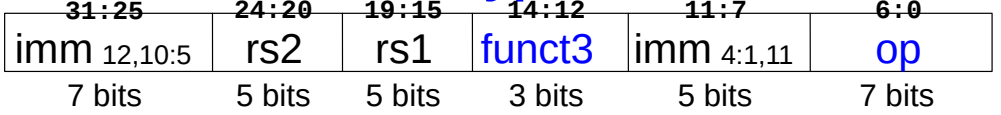
Extended Functionality: *ImmExt*

ImmSrc $_{1:0}$	ImmExt	Instruction Type
00	{{20{instr[31]}}, instr[31:20]}	I-Type
01	{{20{instr[31]}}, instr[31:25], instr[11:7]}	S-Type
10	{{19{instr[31]}}, instr[31], instr[7], instr[30:25], instr[11:8], 1'b0}	B-Type
11	**{{12{instr[31]}}, instr[19:12], instr[20], instr[30:21], 1'b0}**	**J-Type**

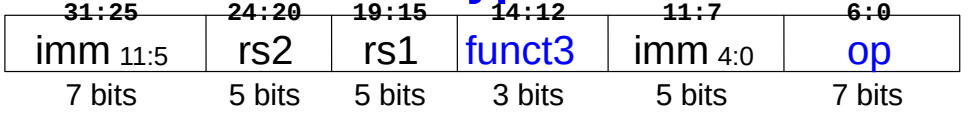
I-Type



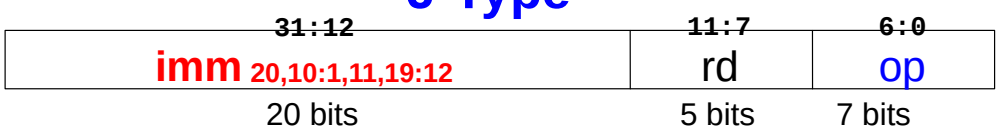
B-Type



S-Type



J-Type

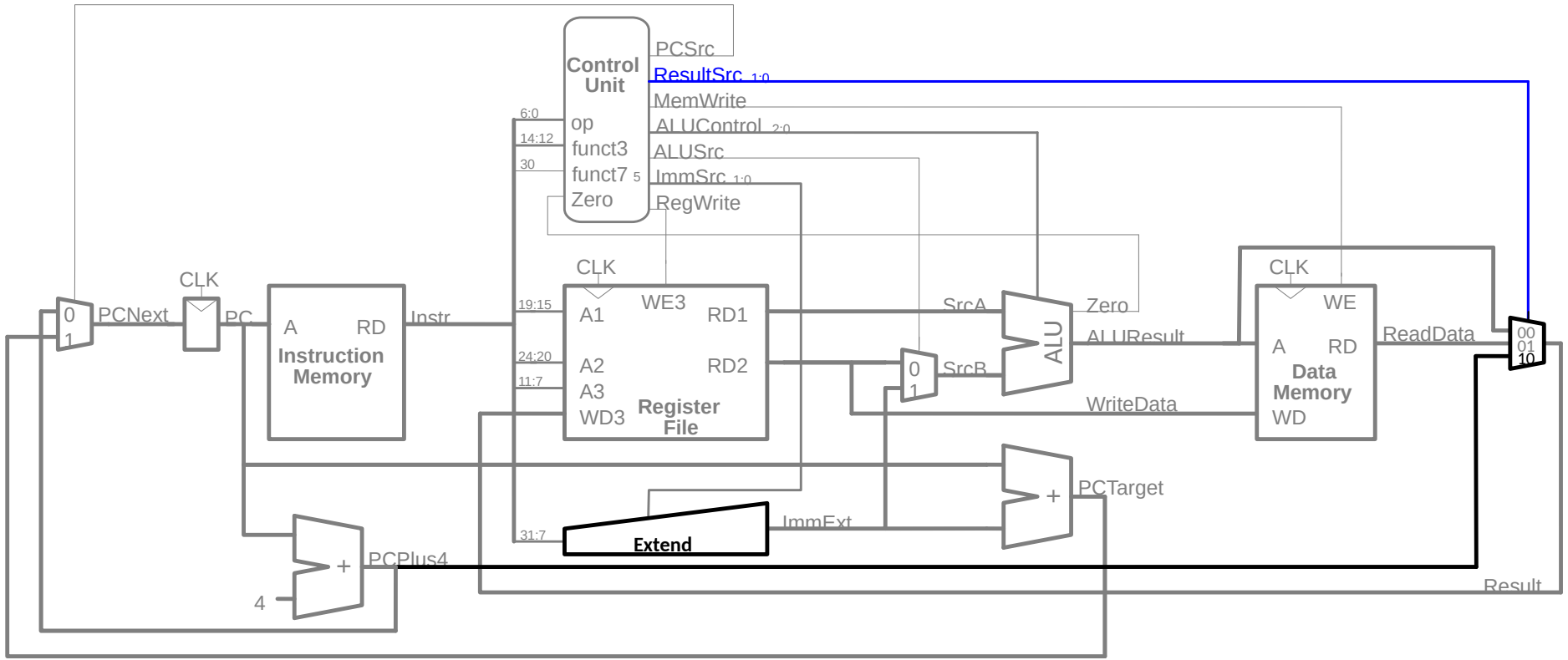


Extended Functionality: jal

op	Instruct.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp	Jump
3	lw	1	00	1	0	01	0	00	0
35	sw	0	01	1	1	XX	0	00	0
51	R-type	1	XX	0	0	00	0	10	0
99	beq	0	10	0	0	XX	1	01	0
19	I-type	1	00	1	0	00	0	10	0
111	jal	1	11	X	0	10	0	XX	1

Extended Functionality: jal

op	Instruct.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp	Jump
111	jal	1	11	X	0	10	0	XX	1



Chapter 7: Microarchitecture

Single-Cycle Performance

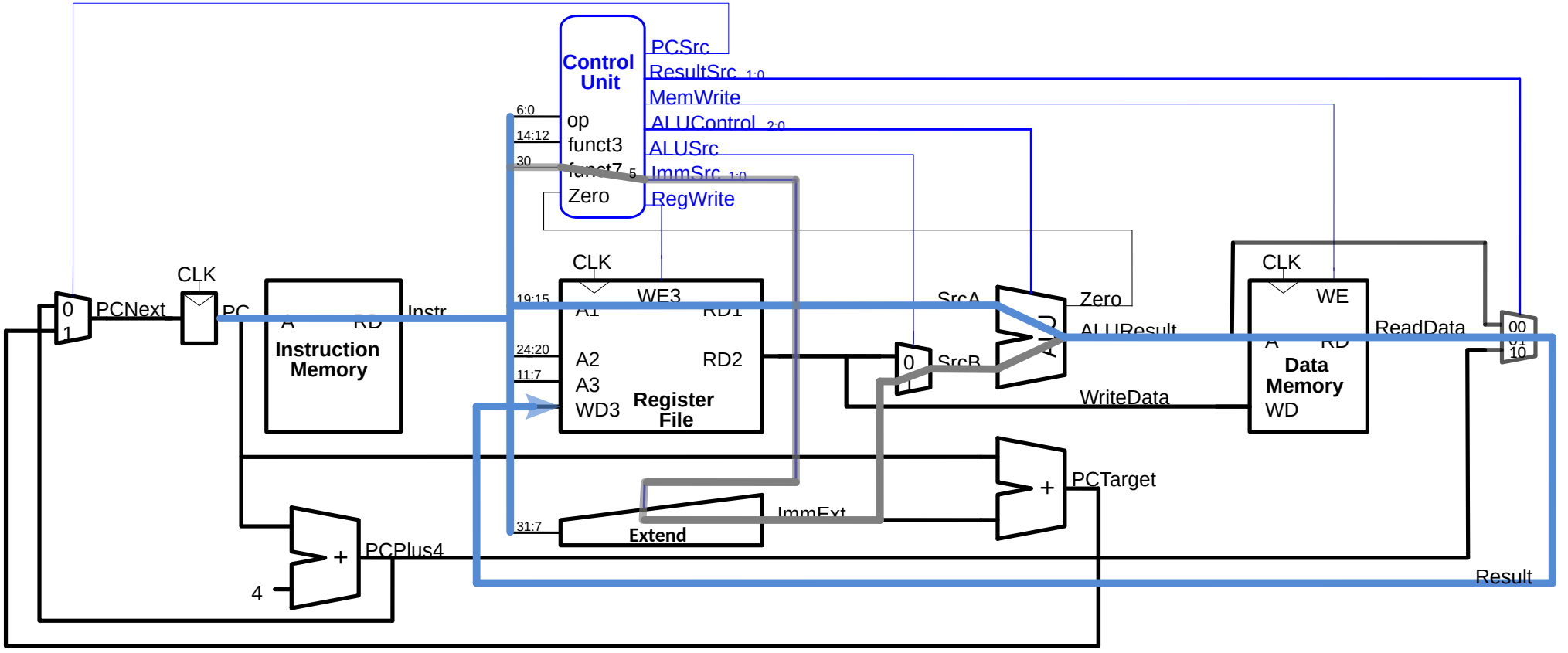
Processor Performance

Program Execution Time

= (#instructions)(cycles/instruction)(seconds/cycle)

= # instructions x CPI x T_C

Single-Cycle Processor Performance



T_C limited by critical path (lw)

Single-Cycle Processor Performance

- **Single-cycle critical path:**

$$T_{c\_single} = t_{pcq\_PC} + t_{\text{mem}} + \max[t_{RF\text{read}}, t_{dec} + t_{ext} + t_{\text{mux}}] + t_{\text{ALU}} + t_{\text{mem}} + t_{\text{mux}} + t_{RF\text{setup}}$$

- **Typically, limiting paths are:**
  - memory, ALU, register file
  - *So,*

$$\begin{aligned} T_{c\_single} &= t_{pcq\_PC} + t_{\text{mem}} + t_{RF\text{read}} + t_{\text{ALU}} + t_{\text{mem}} + t_{\text{mux}} + t_{RF\text{setup}} \\ &= t_{pcq\_PC} + 2t_{\text{mem}} + t_{RF\text{read}} + t_{\text{ALU}} + t_{\text{mux}} + t_{RF\text{setup}} \end{aligned}$$

Single-Cycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	30
AND-OR gate	t_{AND-OR}	20
ALU	t_{ALU}	120
Decoder (Control Unit)	t_{dec}	25
Extend unit	t_{ext}	35
Memory read	t_{mem}	200
Register file read	t_{RFread}	100
Register file setup	$t_{RFsetup}$	60

$$T_{c\_single} = t_{pcq\_PC} + 2t_{mem} + t_{RFread} + t_{ALU} + t_{mux} + t_{RFsetup}$$

$$=$$

Single-Cycle Performance Example

Program with 100 billion instructions:

Execution Time = # instructions x CPI x T_C

= **75 seconds**

Chapter 7: Microarchitecture

Multicycle RISC-V Processor

Note to self:
For a few additional slides switch to my "old" Muti-cycle presentation

Single- vs. Multicycle Processor

- **Single-cycle:**
  + simple
  - cycle time limited by longest instruction (`lw`)
  - separate memories for instruction and data
  - 3 adders/ALUs
- **Multicycle processor** addresses these issues by breaking instruction into **shorter steps**
  - shorter instructions take fewer steps
  - can re-use hardware
  - cycle time is faster

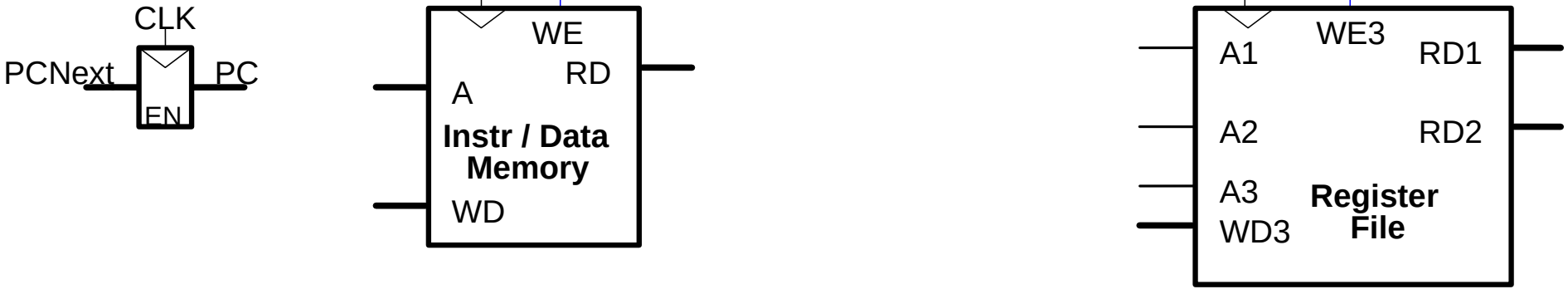
Single- vs. Multicycle Processor

- **Single-cycle:**
  + simple
  - cycle time limited by longest instruction (`lw`)
  - separate memories for instruction and data
  - 3 adders/ALUs
- **Multicycle:**
  + higher clock speed
  + simpler instructions run faster
  + reuse expensive hardware on multiple cycles
  - sequencing overhead paid many times

Same design steps as single-cycle:
- first datapath
- then control

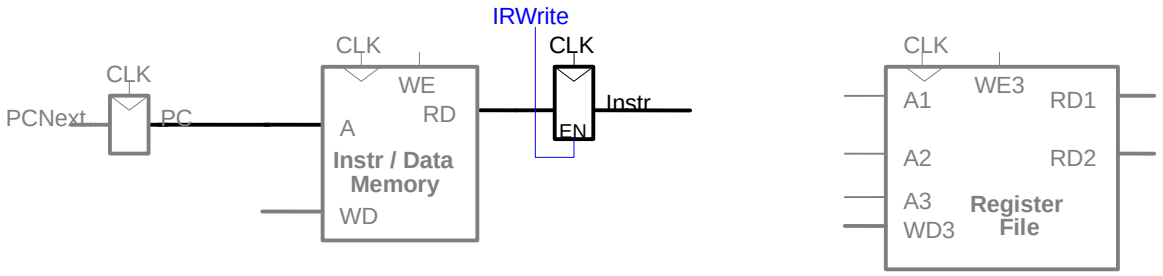
Multicycle State Elements

Replace separate Instruction and Data memories with a **single unified memory** – more realistic



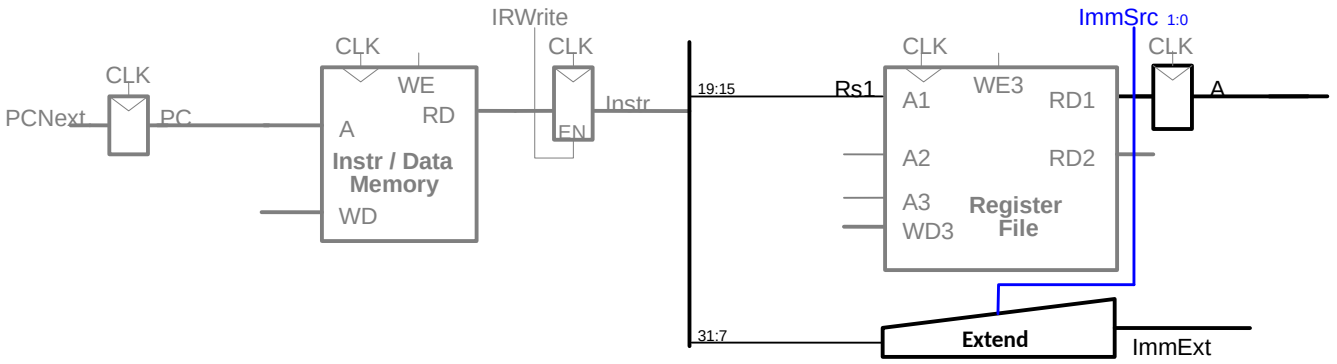
Multicycle Datapath: Instruction Fetch

STEP 1: Fetch instruction



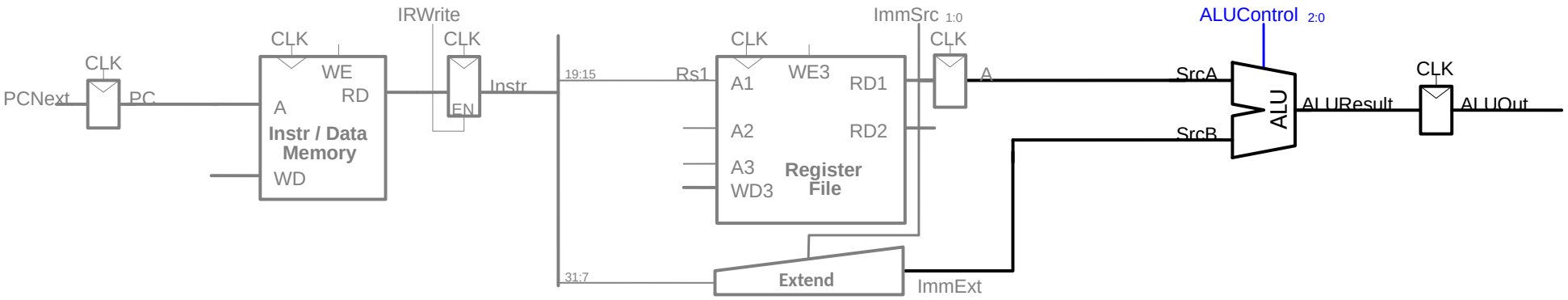
Multicycle Datapath: lw Get Sources

STEP 2: Read source operand from RF and extend immediate



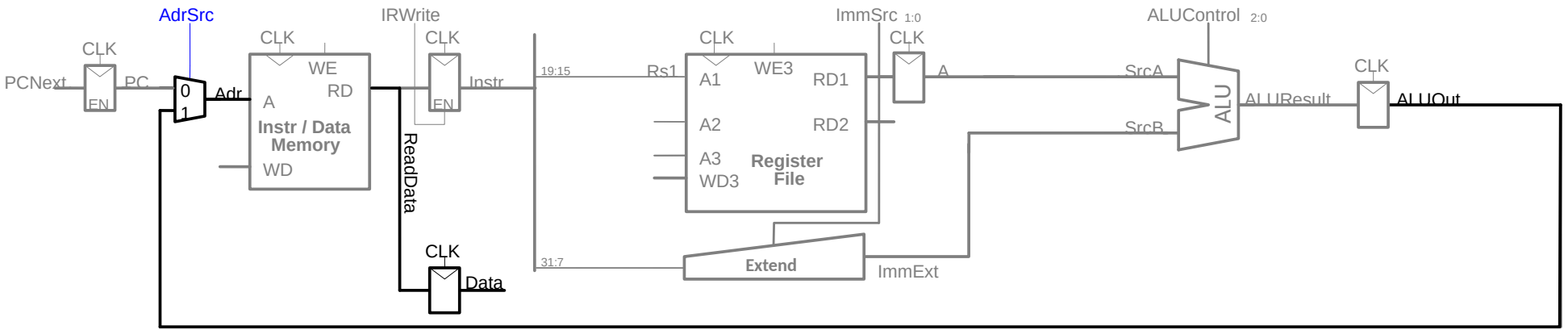
Multicycle Datapath: `lw` Address

STEP 3: Compute the memory address



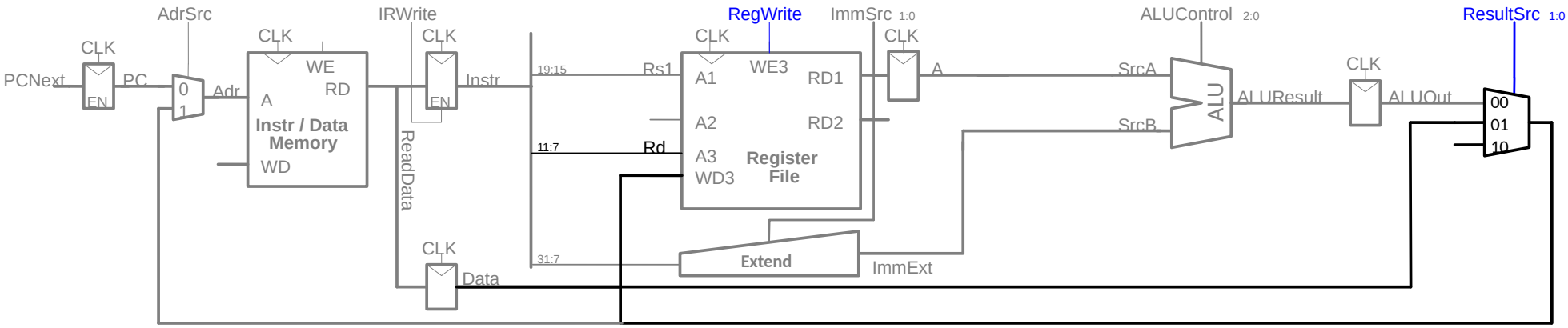
Multicycle Datapath: `lw` Memory Read

STEP 4: Read data from memory



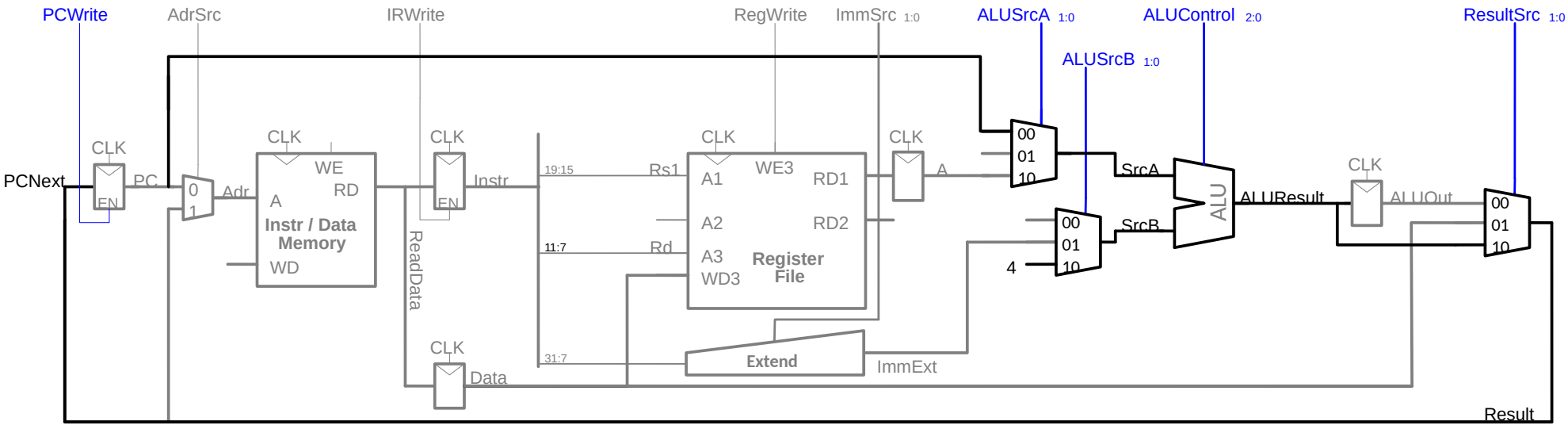
Multicycle Datapath: `lw` Write Register

STEP 5: Write data back to register file



Multicycle Datapath: Increment PC

STEP 6: Increment PC: PC = PC+4

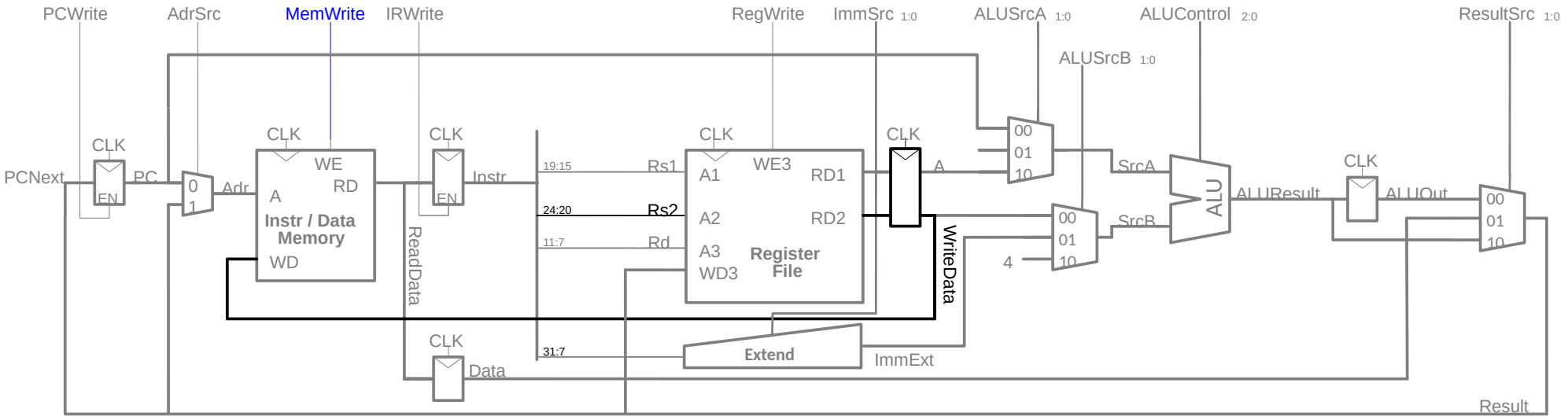


Chapter 7: Microarchitecture

Multicycle Datapath: Other Instructions

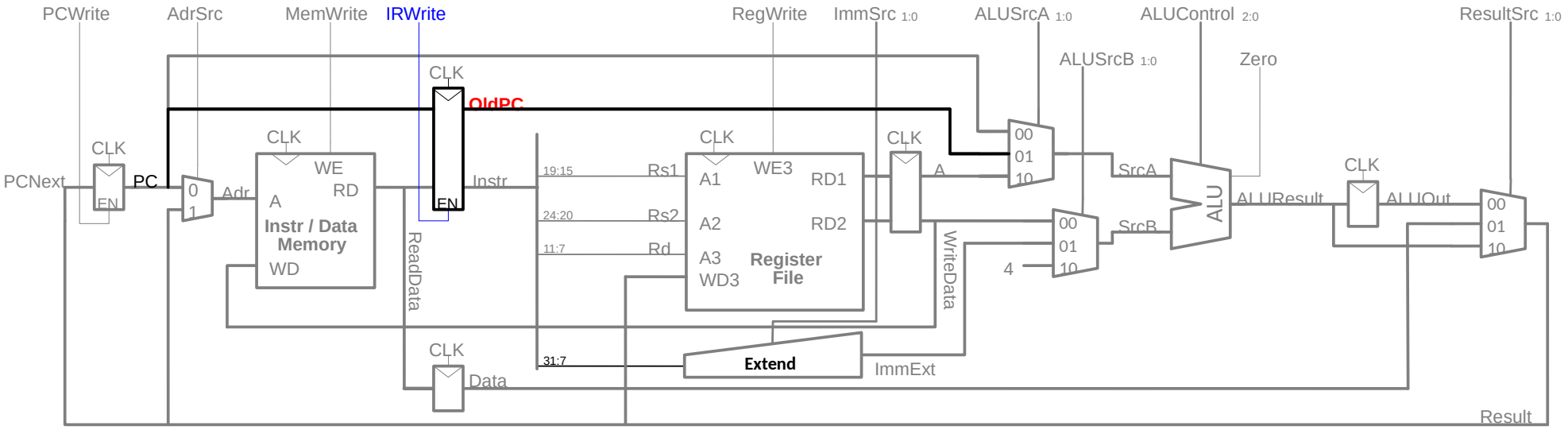
Multicycle Datapath: sw

Write data in **rs2** to memory



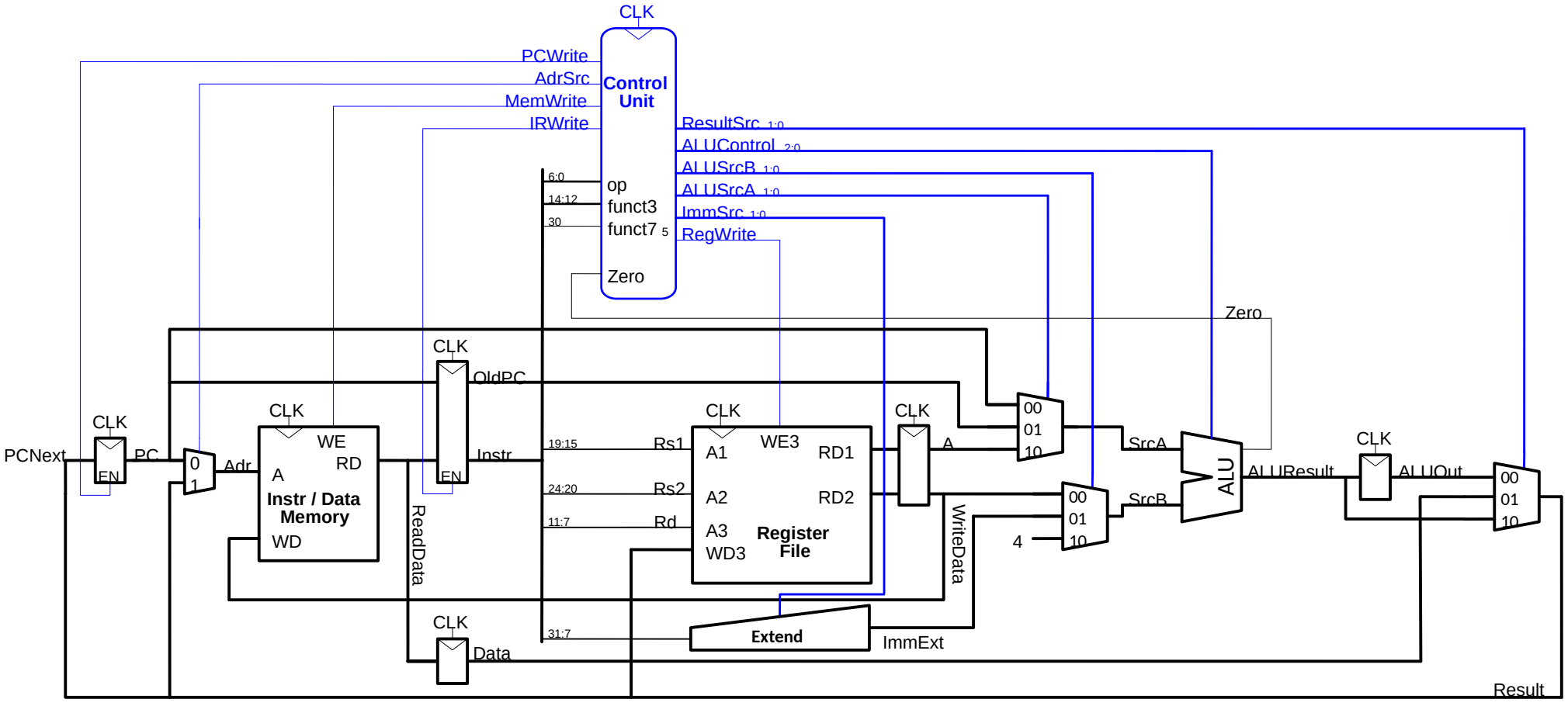
Multicycle Datapath: beq

Calculate branch target address:
BTA = PC + imm



PC is updated in Fetch stage, so need to save **old (current) PC**

Multicycle RISC-V Processor

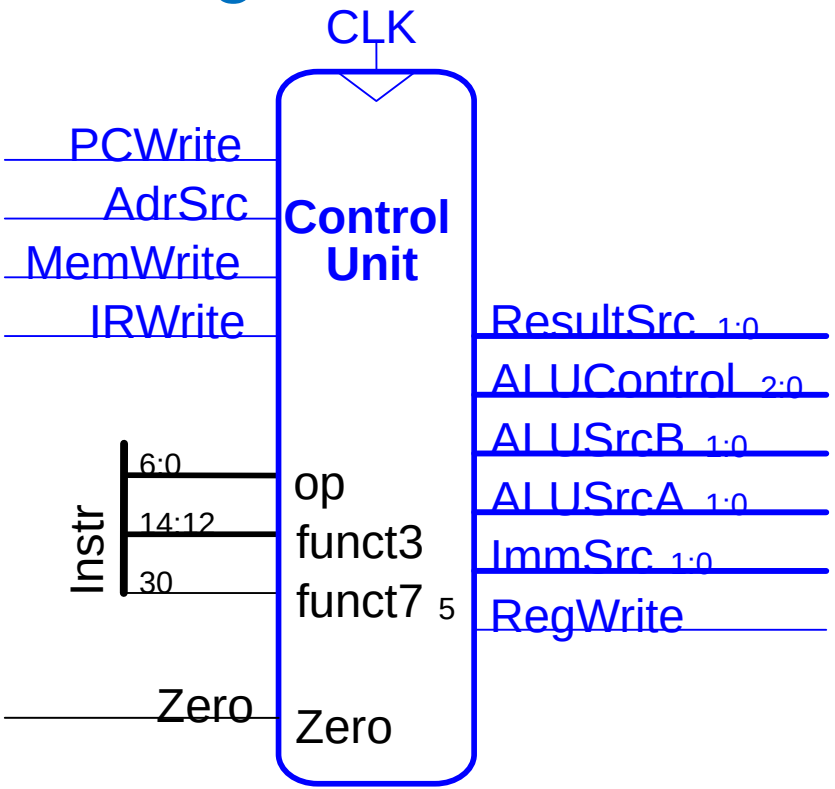


Chapter 7: Microarchitecture

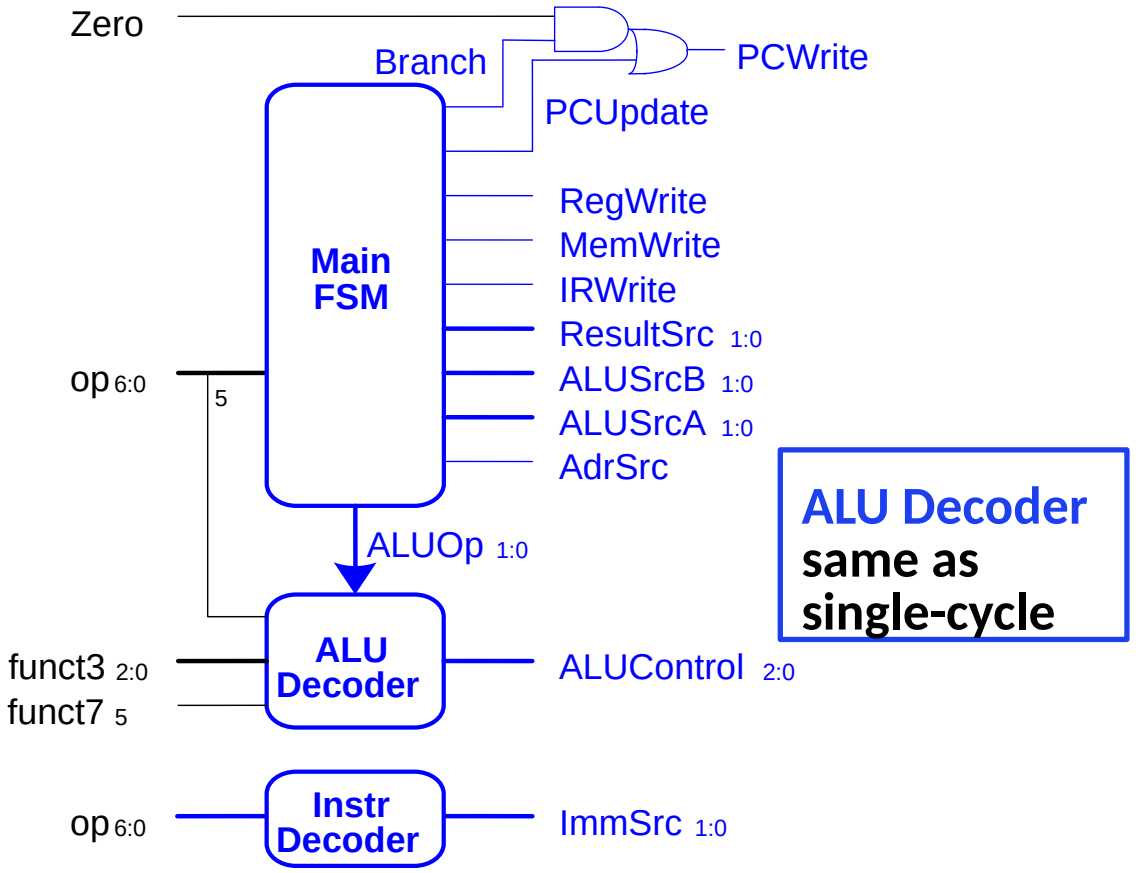
Multicycle Control

Multicycle Control

High-Level View

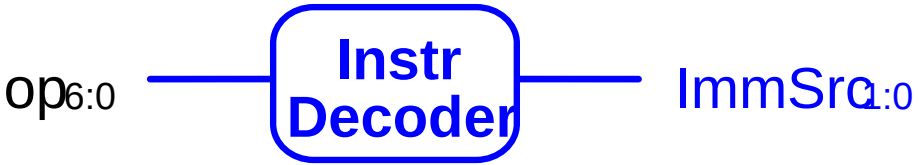


Low-Level View



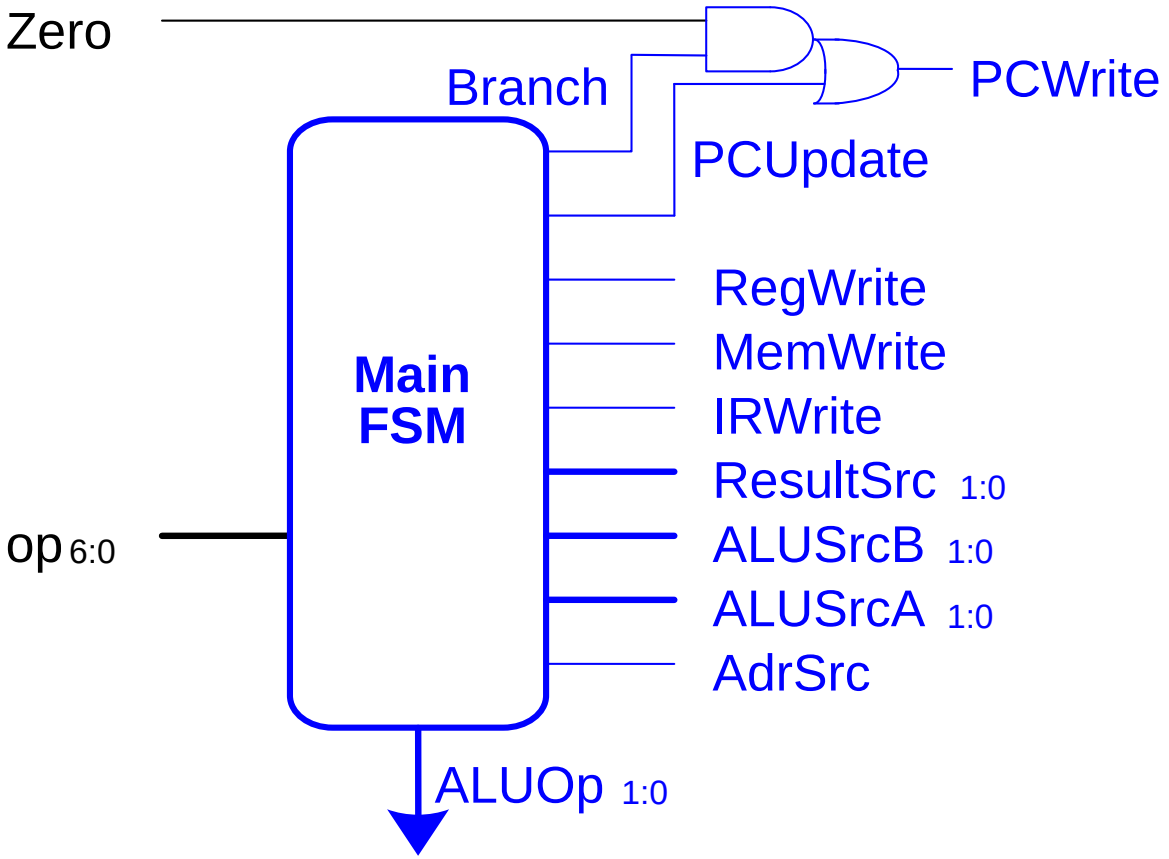
ALU Decoder same as single-cycle

Multicycle Control: Instruction Decoder



op	Instruction	ImmSrc
3	**lw**	00
35	**sw**	01
51	**R-type**	XX
99	**beq**	10

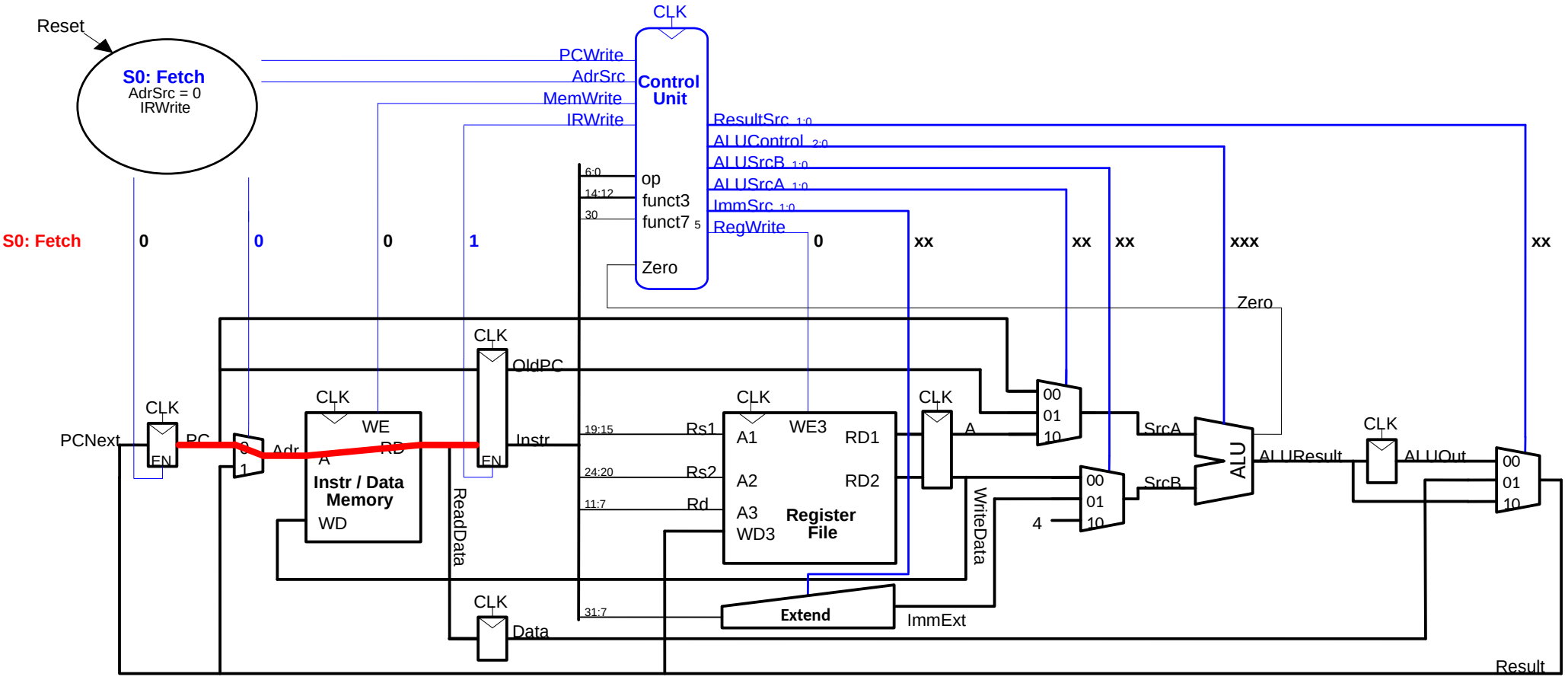
Multicycle Control: Main FSM



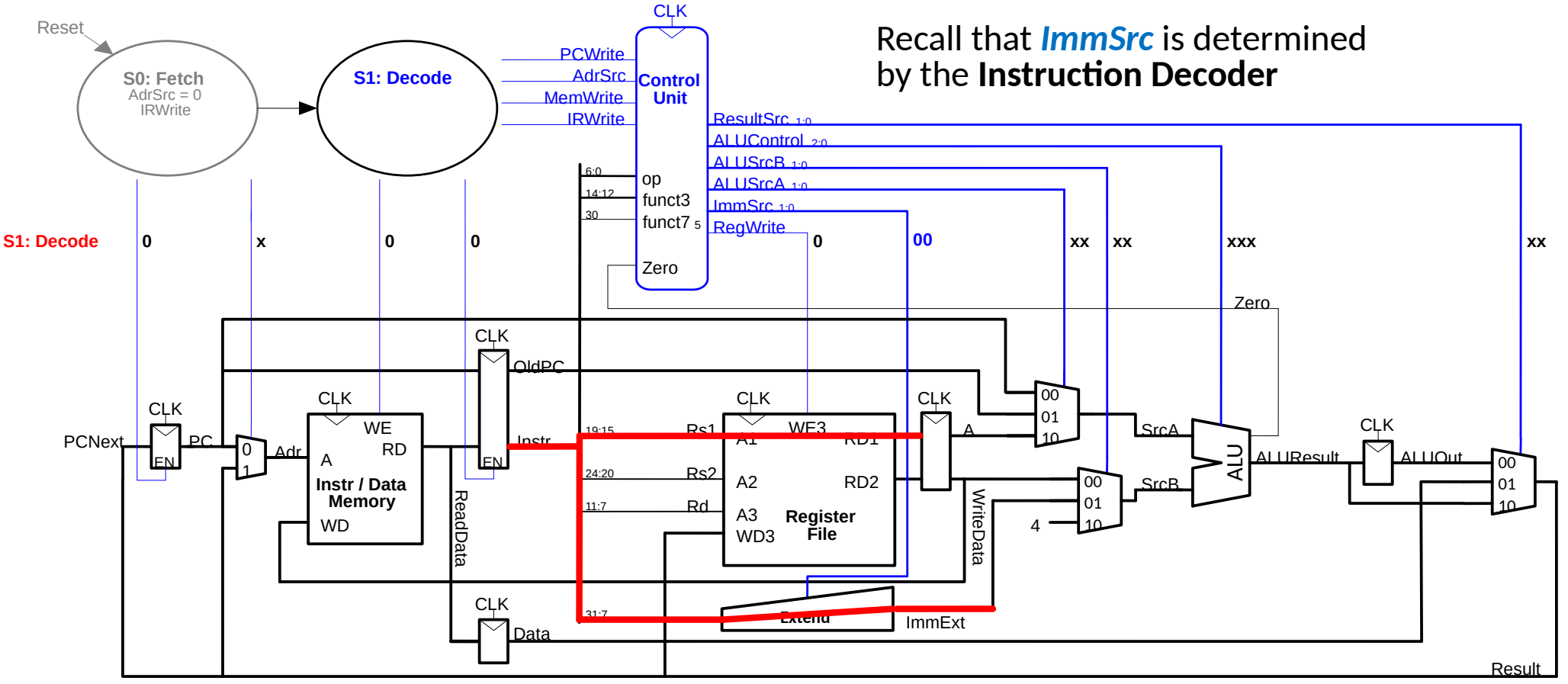
To declutter FSM:

- **Write enable signals** (RegWrite, MemWrite, IRWrite, PCUpdate, and Branch) are **0** if not listed in a state.
- **Other signals are don't care** if not listed in a state

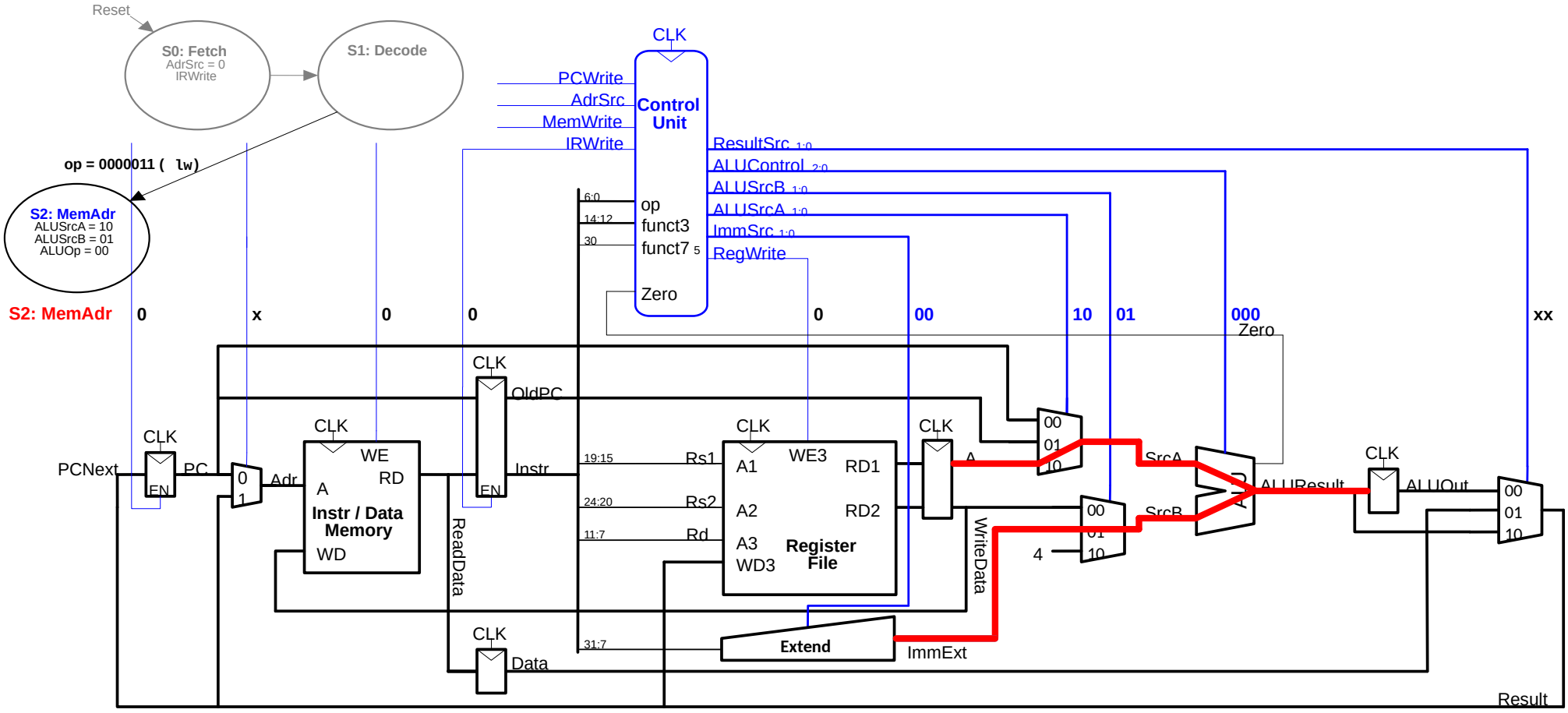
Main FSM: Fetch



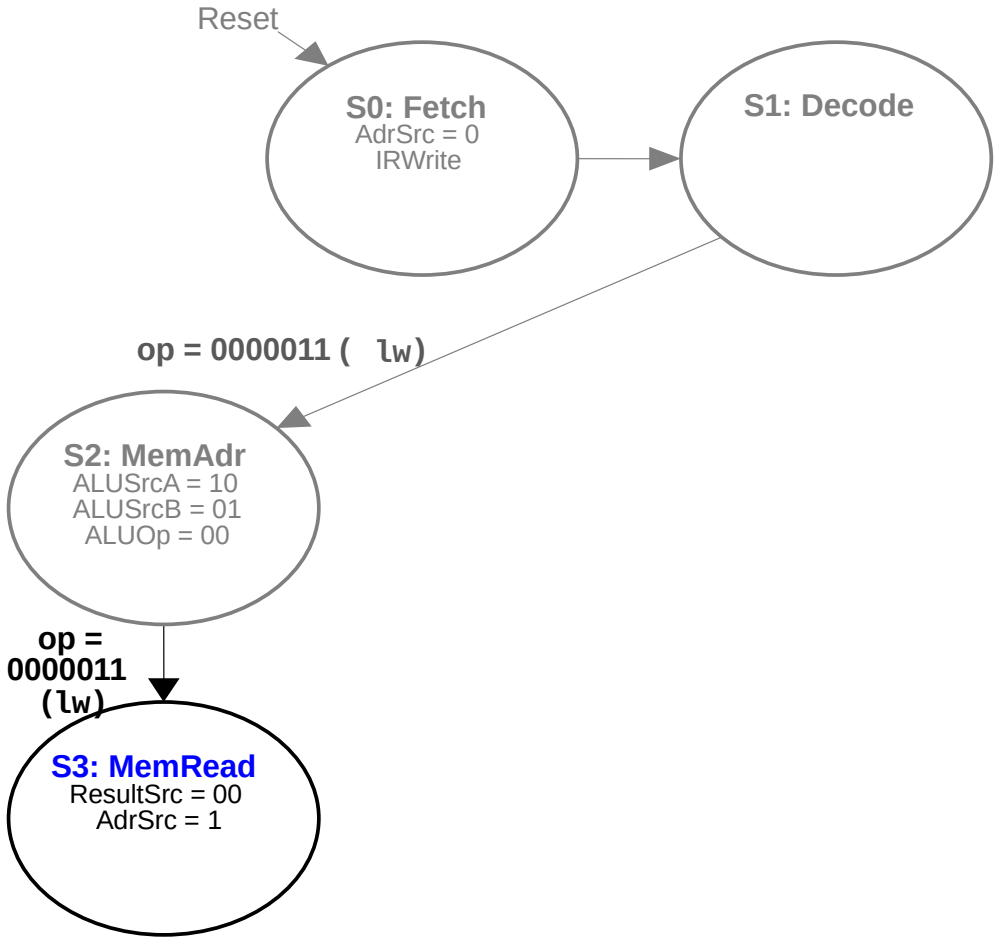
Main FSM: Decode



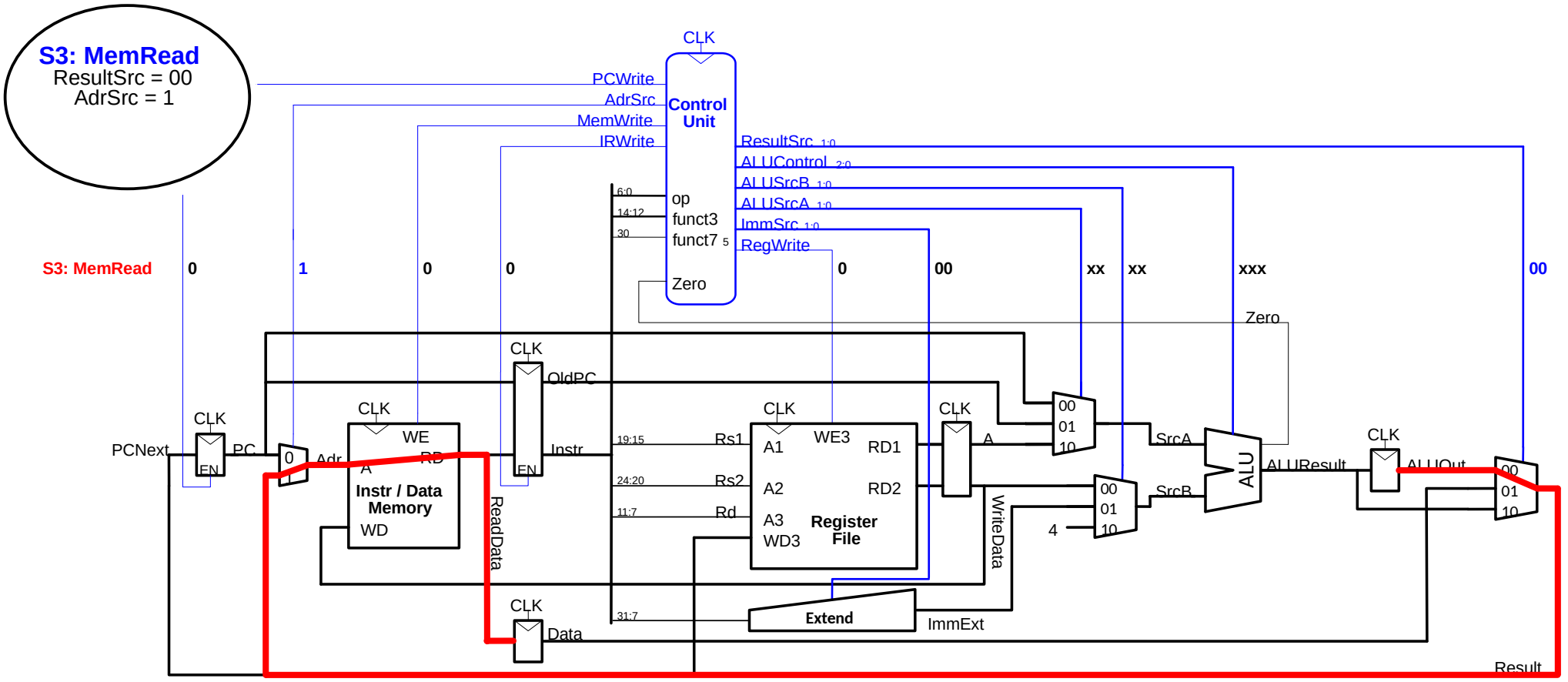
Main FSM: Address



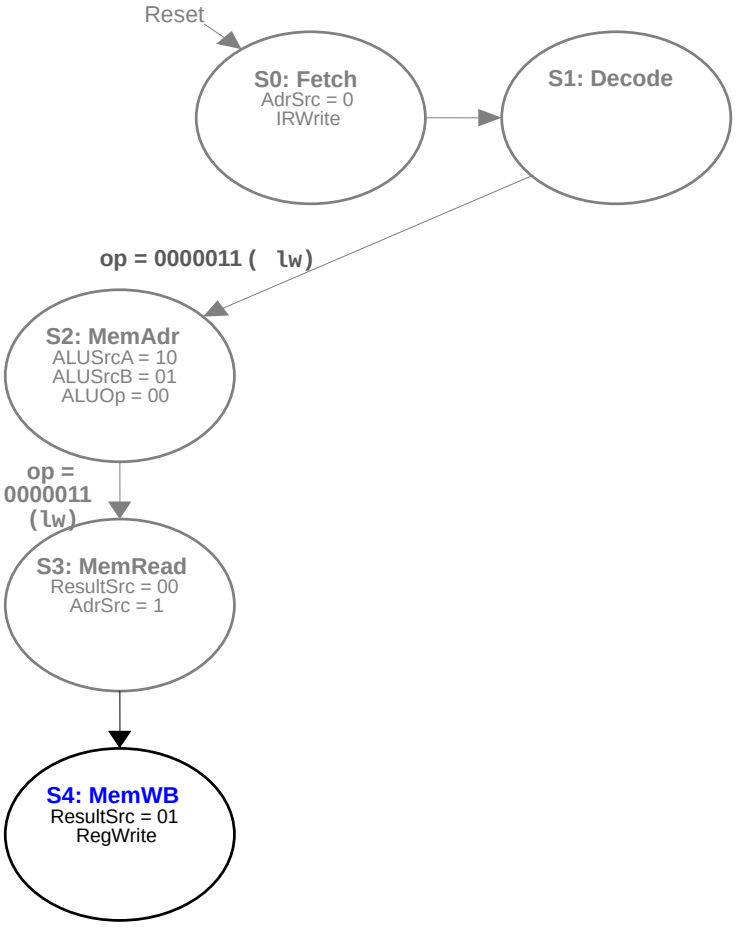
Main FSM: Read Memory



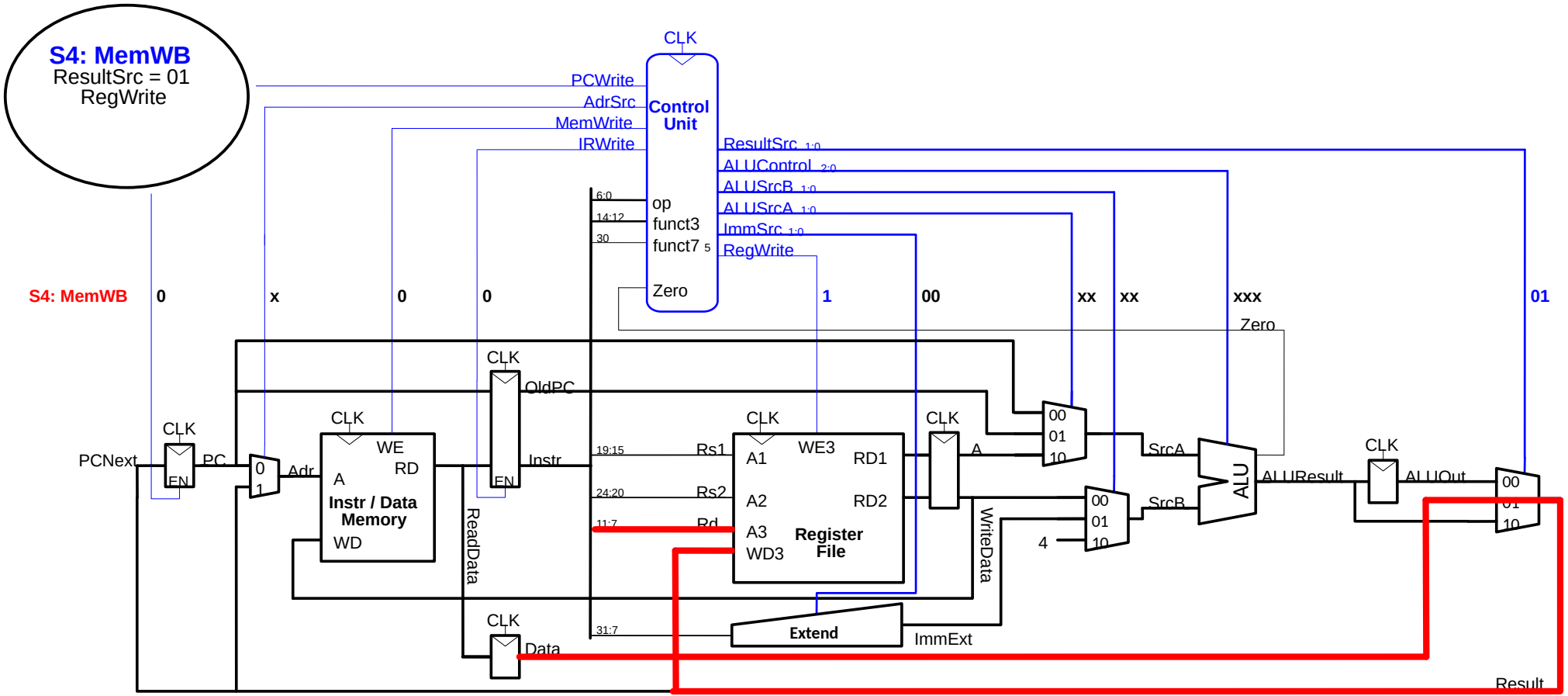
Main FSM: Read Memory Datapath



Main FSM: Write RF

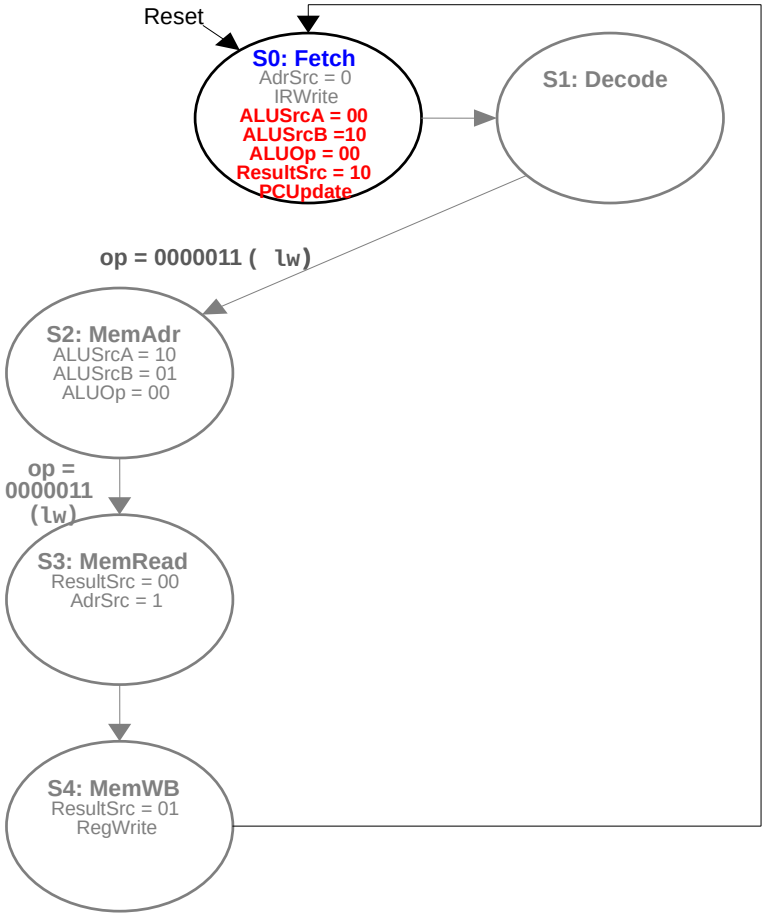


Main FSM: Write RF Datapath

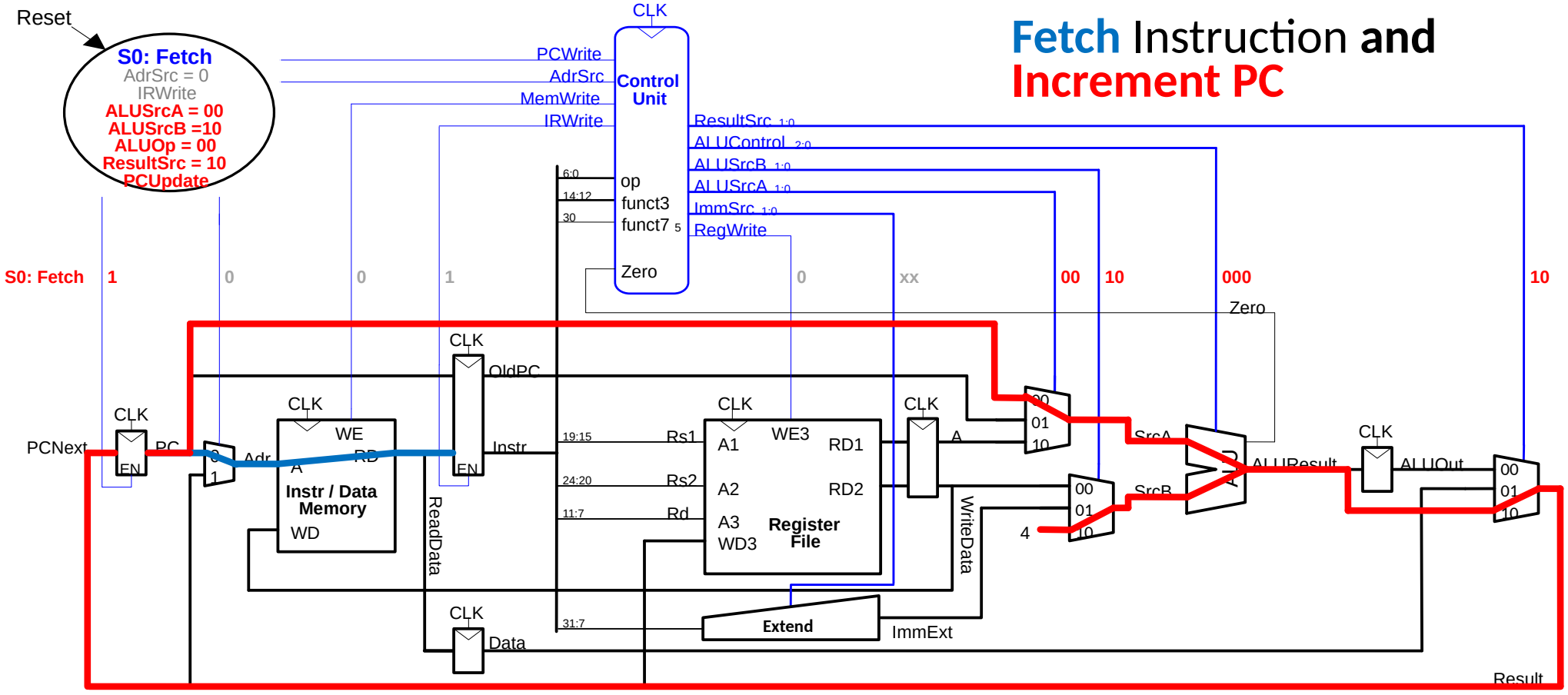


Main FSM: Fetch Revisited

Calculate **PC+4** during Fetch stage (ALU isn't being used)



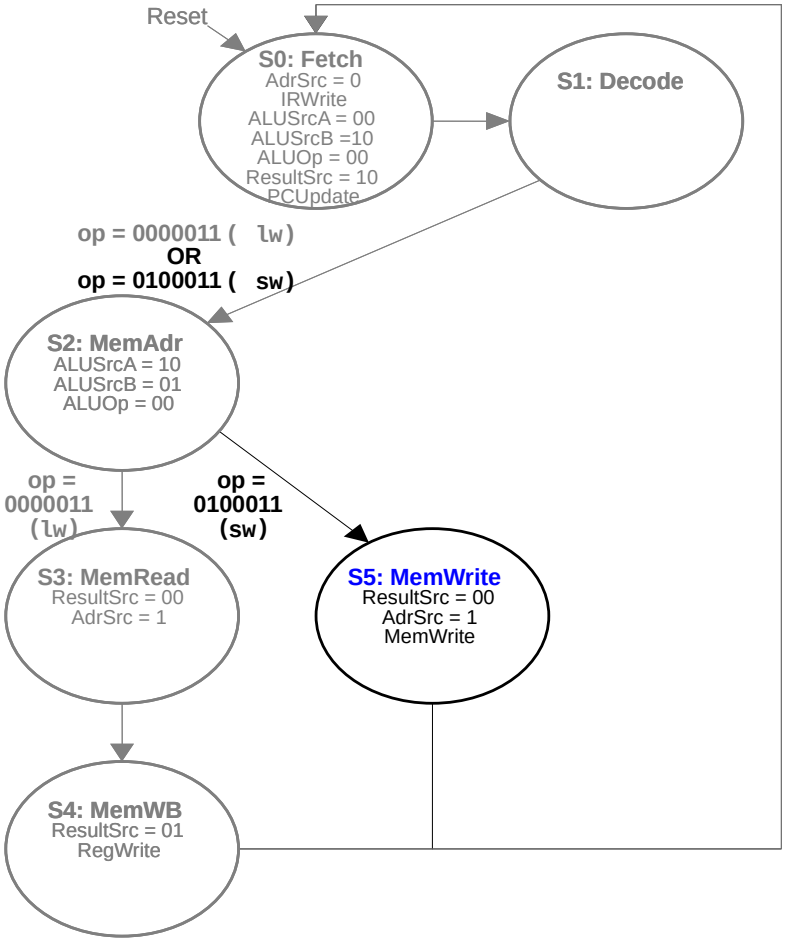
Main FSM: Fetch (PC+4) Datapath



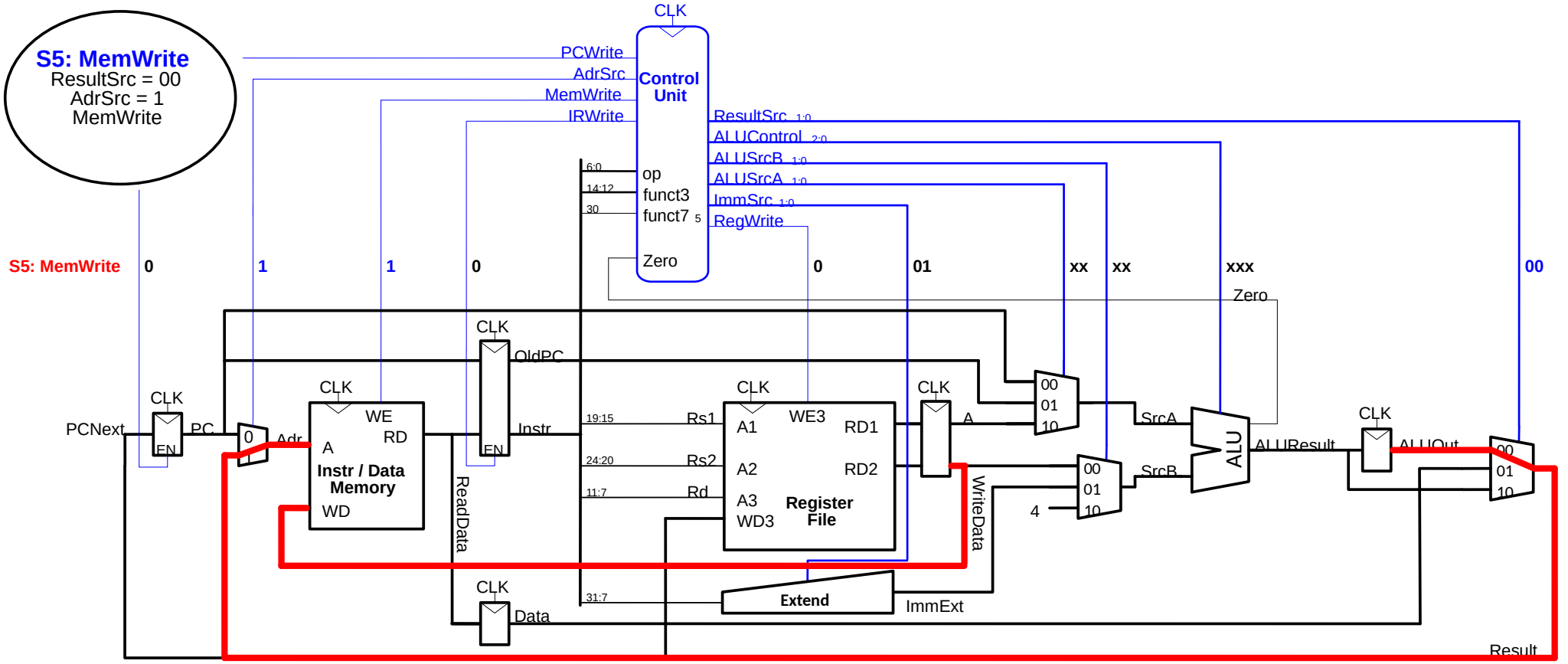
Chapter 7: Microarchitecture

Multicycle Control: Other Instructions

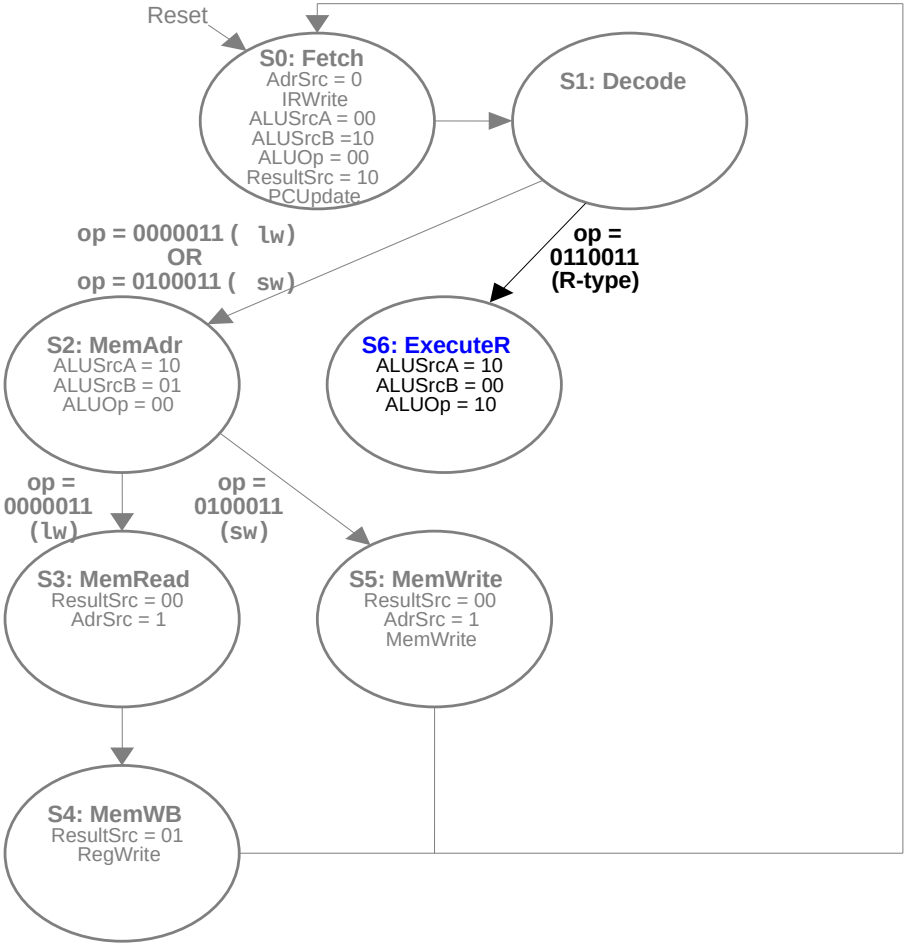
Main FSM: sw


Reset
S0: Fetch
AdrSrc = 0
IRWrite
ALUSrcA = 00
ALUSrcB =10
ALUOp = 00
ResultSrc = 10
PCUpdate
S1: Decode
op = 0000011 (lw)
OR
op = 0100011 (sw)
S2: MemAdr
ALUSrcA = 10
ALUSrcB = 01
ALUOp = 00
op = 0000011 (lw)
op = 0100011 (sw)
S3: MemRead
ResultSrc = 00
AdrSrc = 1
S5: MemWrite
ResultSrc = 00
AdrSrc = 1
MemWrite
S4: MemWB
ResultSrc = 01
RegWrite

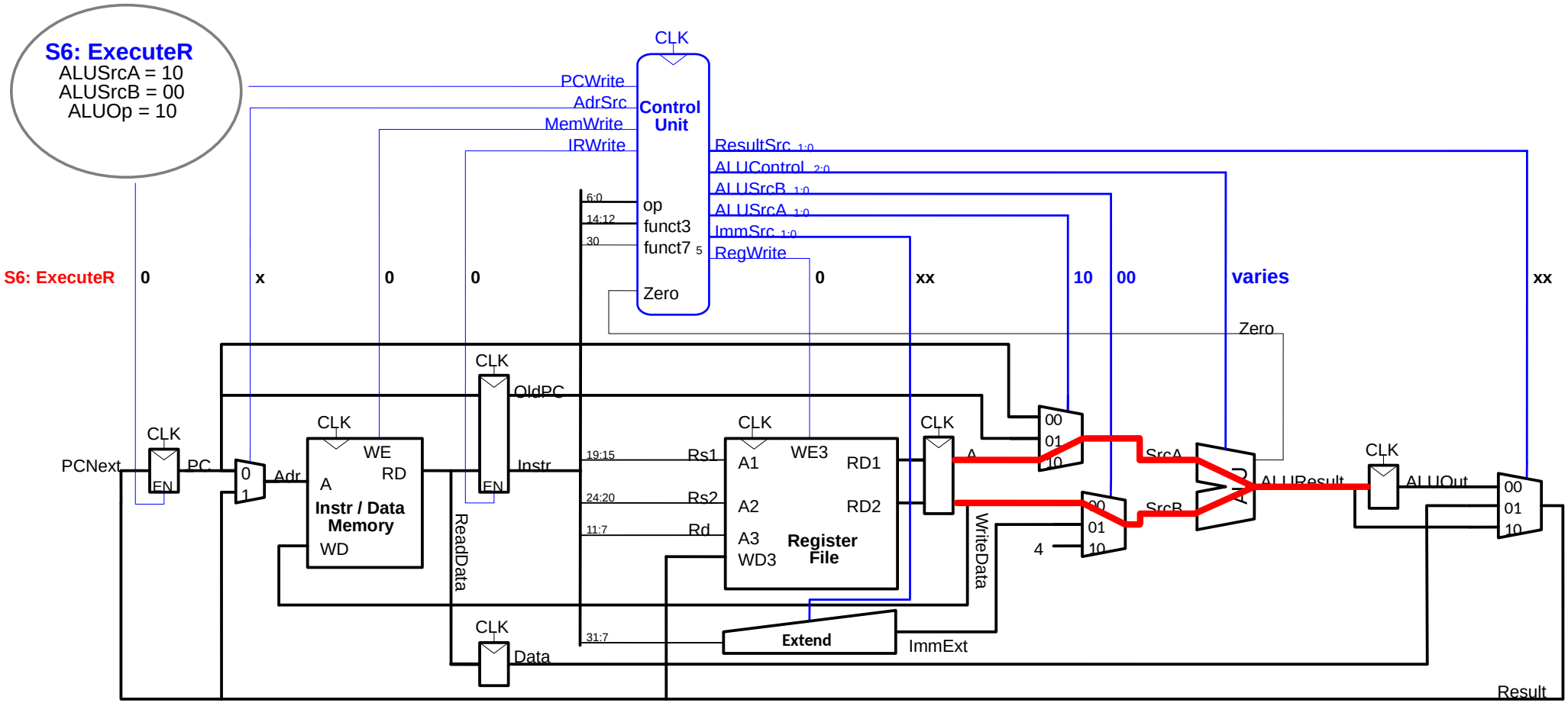
Main FSM: sw Datapath



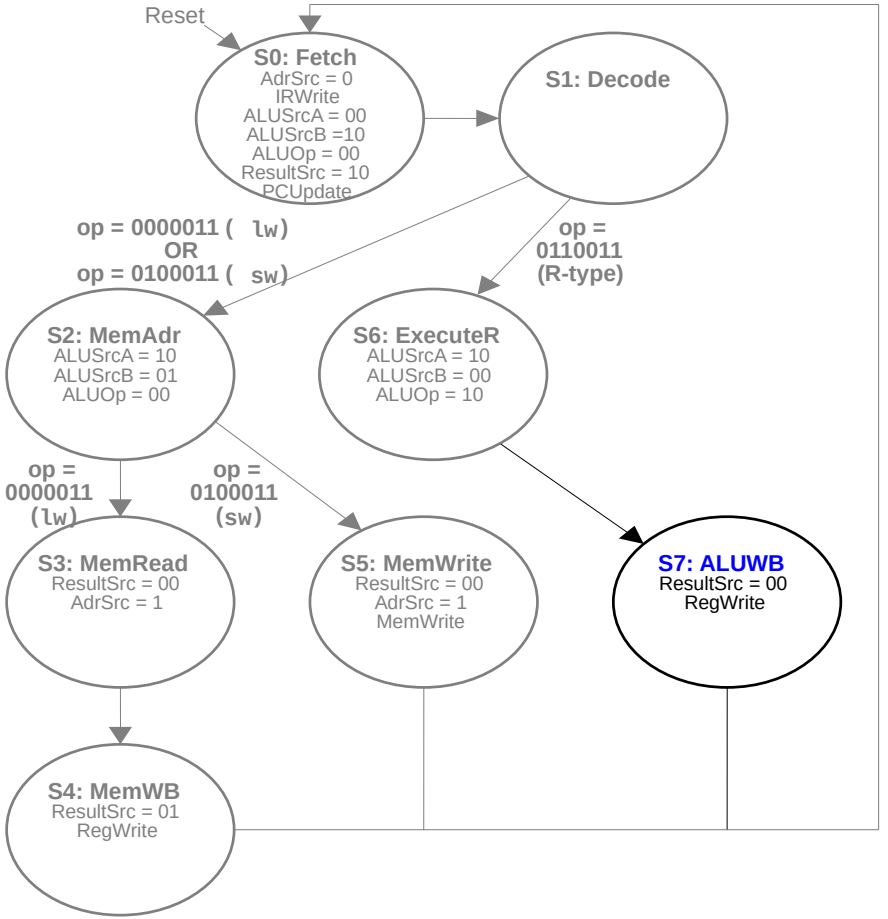
Main FSM: R-Type Execute



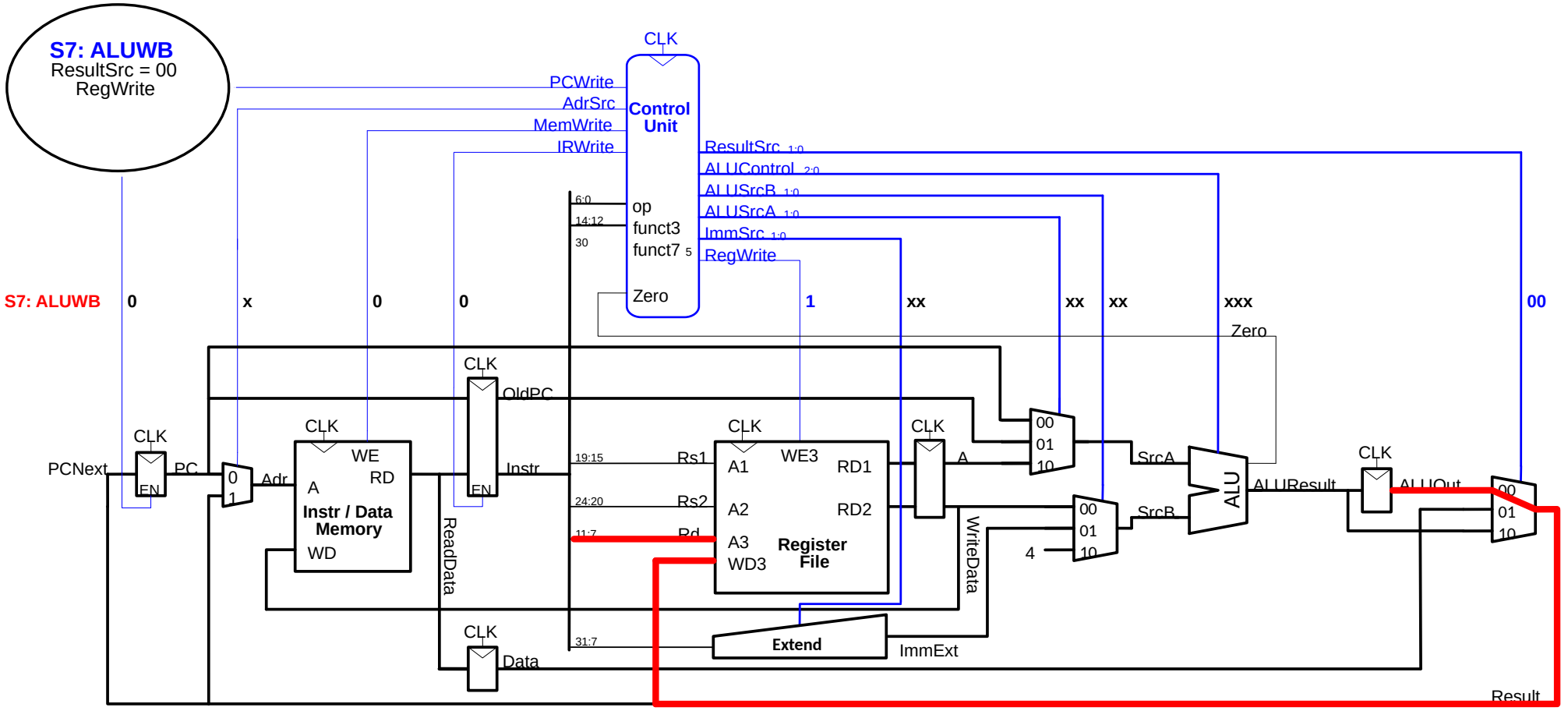
Main FSM: R-Type Execute Datapath



Main FSM: R-Type ALU Write Back



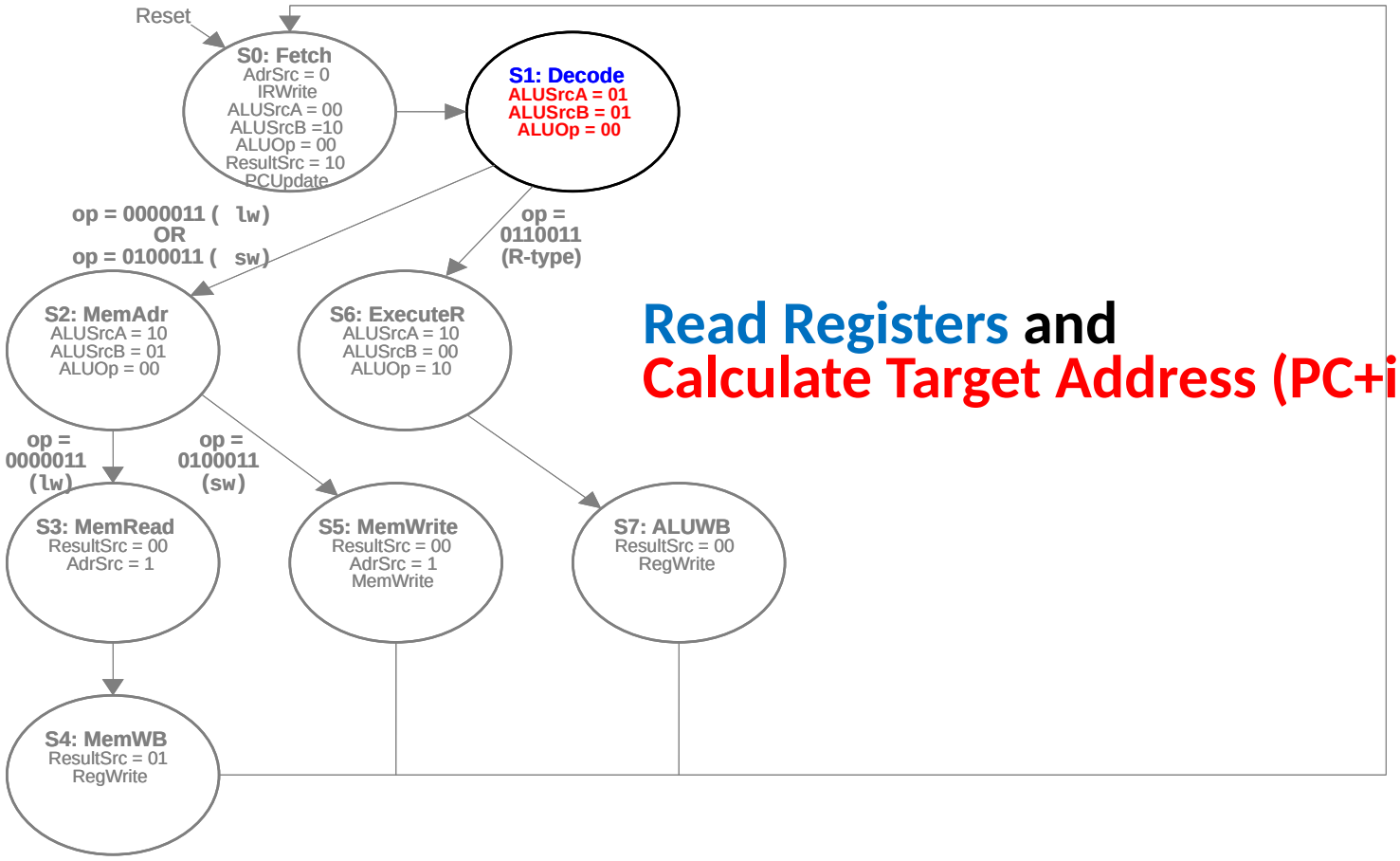
Main FSM: R-Type ALU Write Back



Main FSM: beq

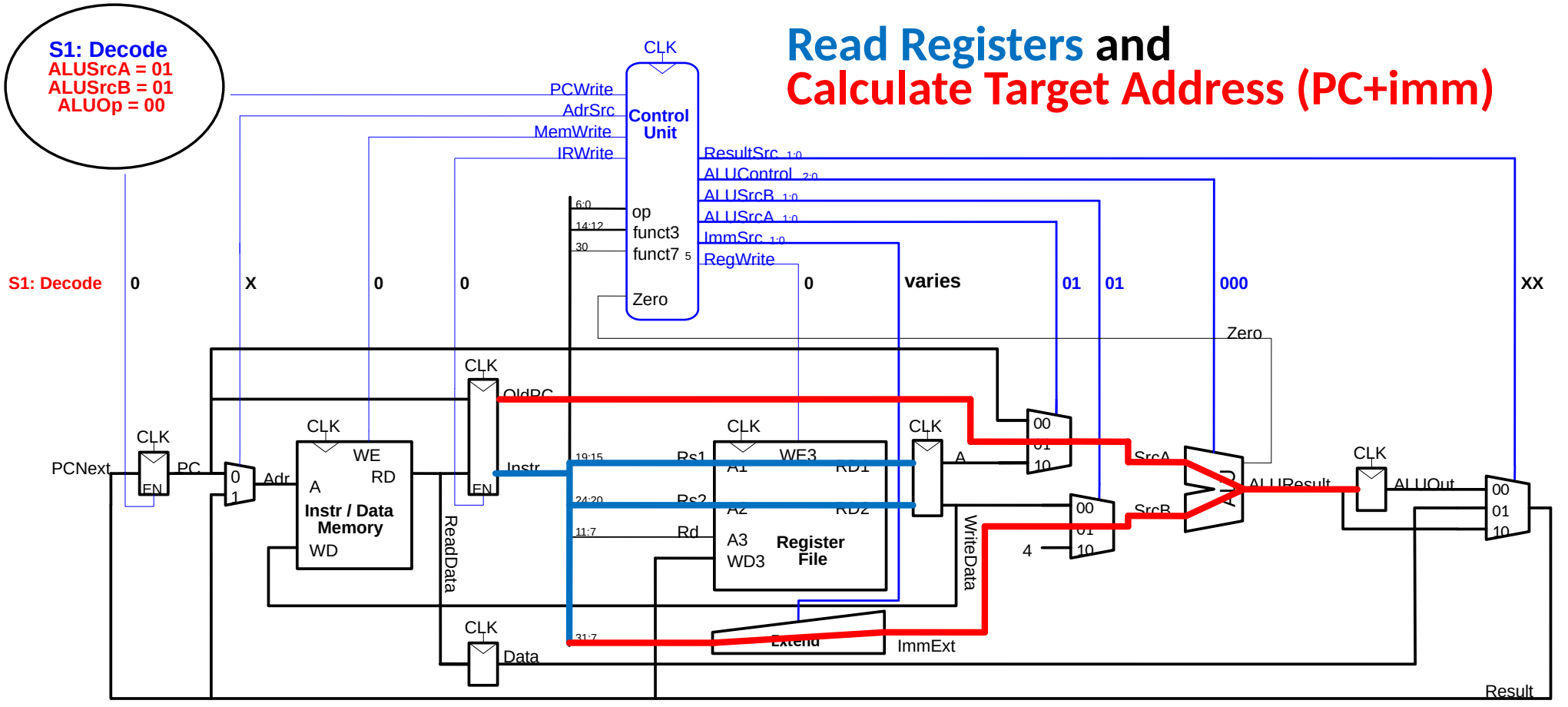
- **Need to calculate:**
  - Branch Target Address
  - **rs1 - rs2** (to see if equal)
- **ALU** isn't being used in Decode stage
  - Use it to calculate Target Address (PC + imm)

Main FSM: Decode Revisited

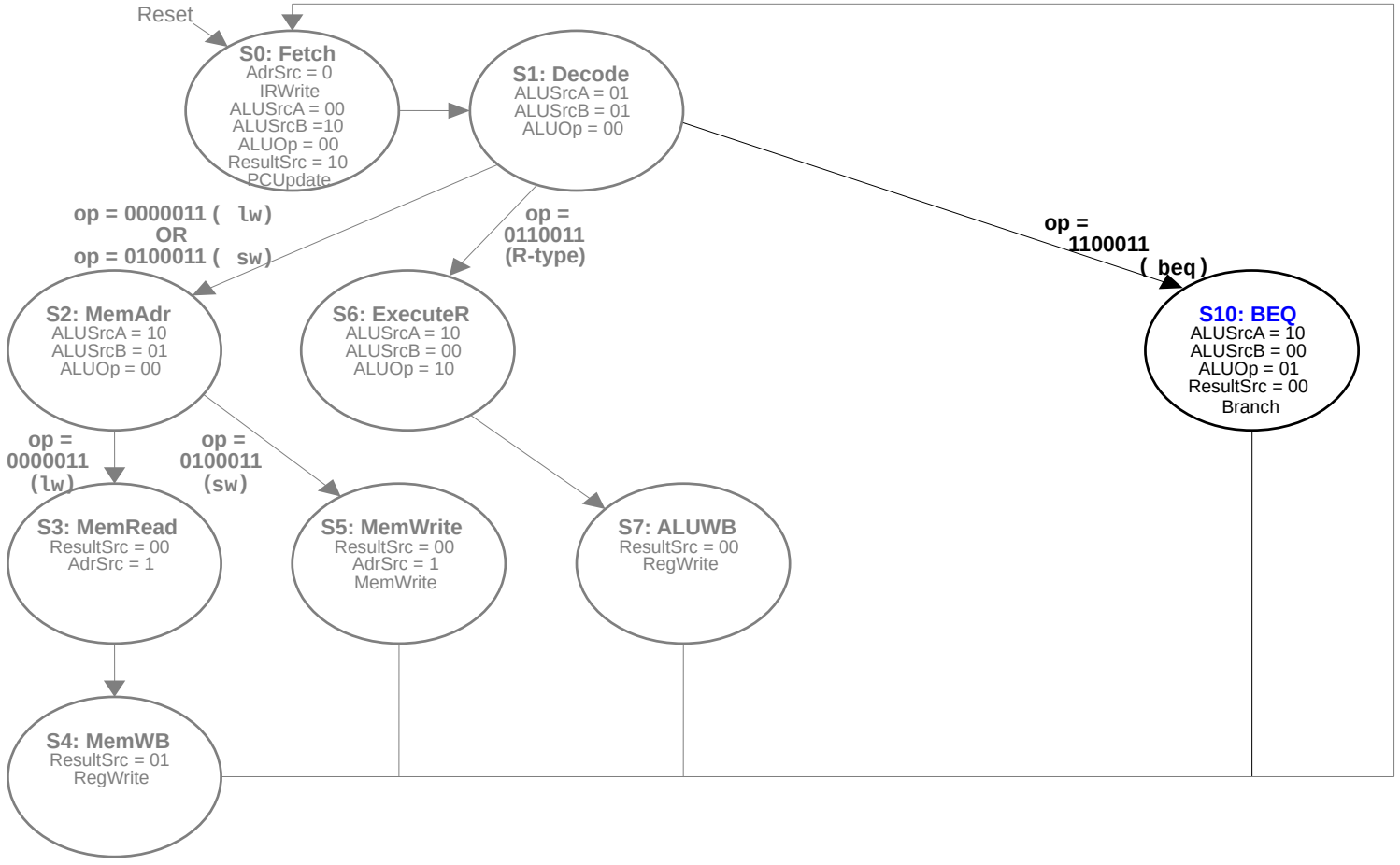


Read Registers and
Calculate Target Address (PC+imm)

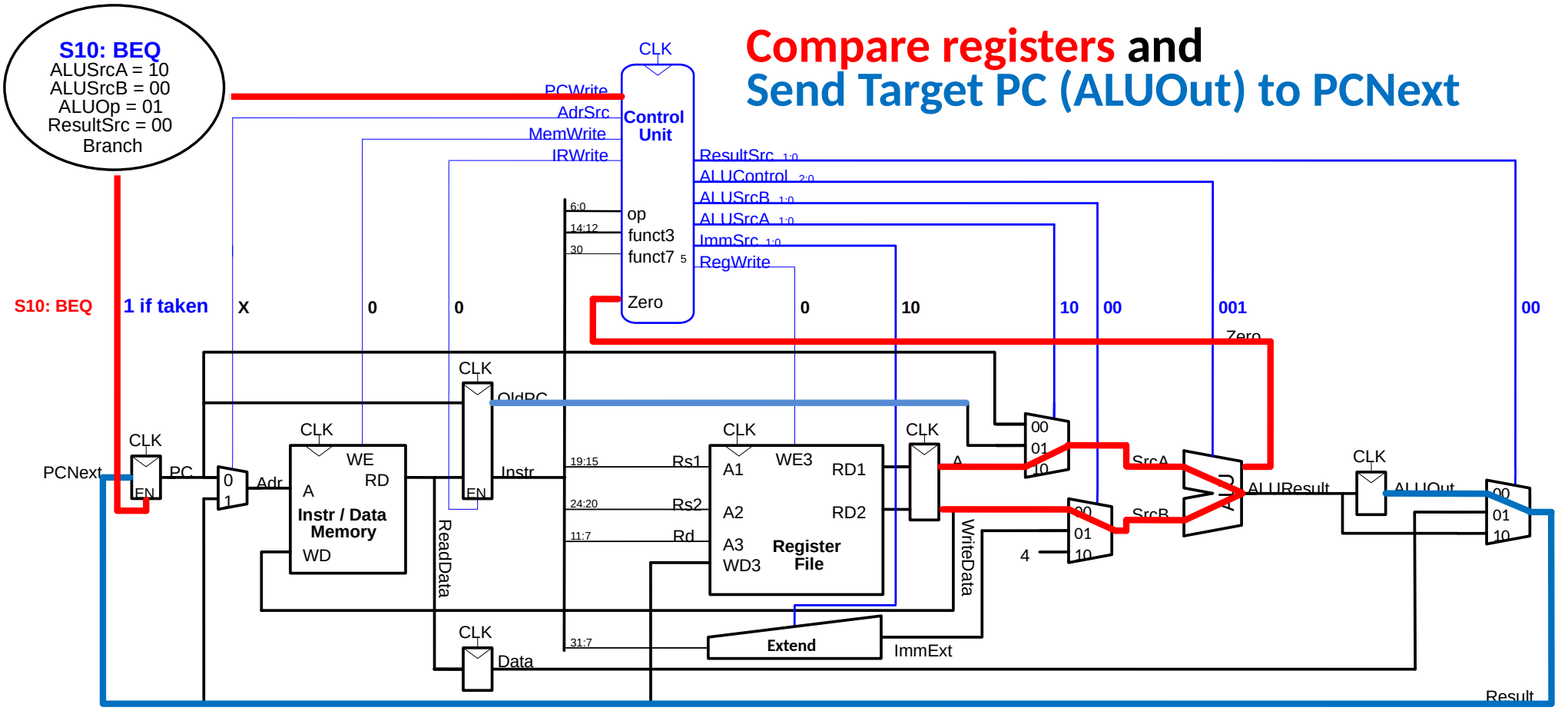
Main FSM: Decode (Target Address)



Main FSM: beq


Reset
S0: Fetch
AdrSrc = 0
IRWrite
ALUSrcA = 00
ALUSrcB =10
ALUOp = 00
ResultSrc = 10
PCUpdate
S1: Decode
ALUSrcA = 01
ALUSrcB = 01
ALUOp = 00
op = 0000011 (lw)
OR
op = 0100011 (sw)
op = 0110011 (R-type)
op = 1100011 (beq)
S2: MemAdr
ALUSrcA = 10
ALUSrcB = 01
ALUOp = 00
S6: ExecuteR
ALUSrcA = 10
ALUSrcB = 00
ALUOp = 10
S10: BEQ
ALUSrcA = 10
ALUSrcB = 00
ALUOp = 01
ResultSrc = 00
Branch
op = 0000011 (lw)
op = 0100011 (sw)
S3: MemRead
ResultSrc = 00
AdrSrc = 1
S5: MemWrite
ResultSrc = 00
AdrSrc = 1
MemWrite
S7: ALUWB
ResultSrc = 00
RegWrite
S4: MemWB
ResultSrc = 01
RegWrite

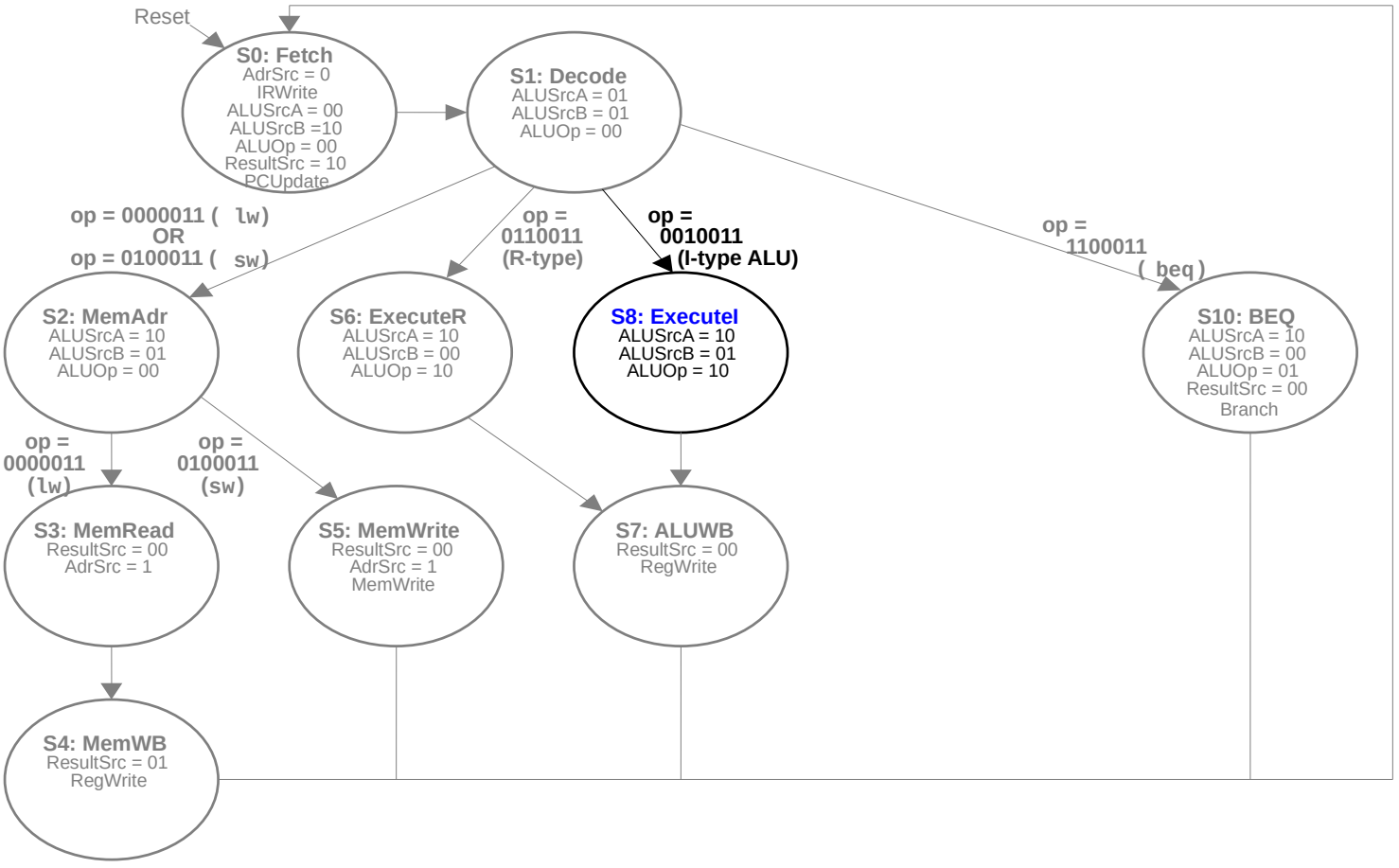
Main FSM: beq Datapath



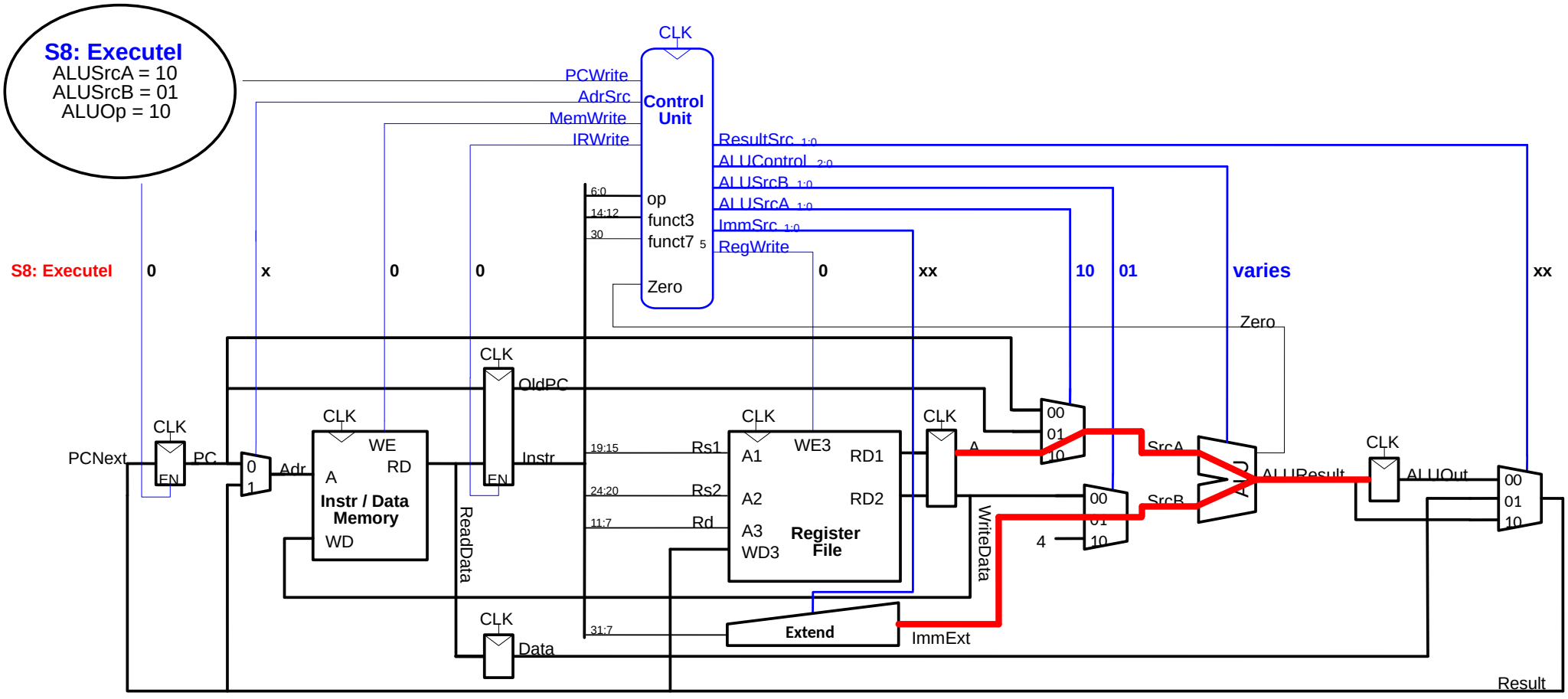
Chapter 7: Microarchitecture

Extending the RISC-V Multicycle Processor

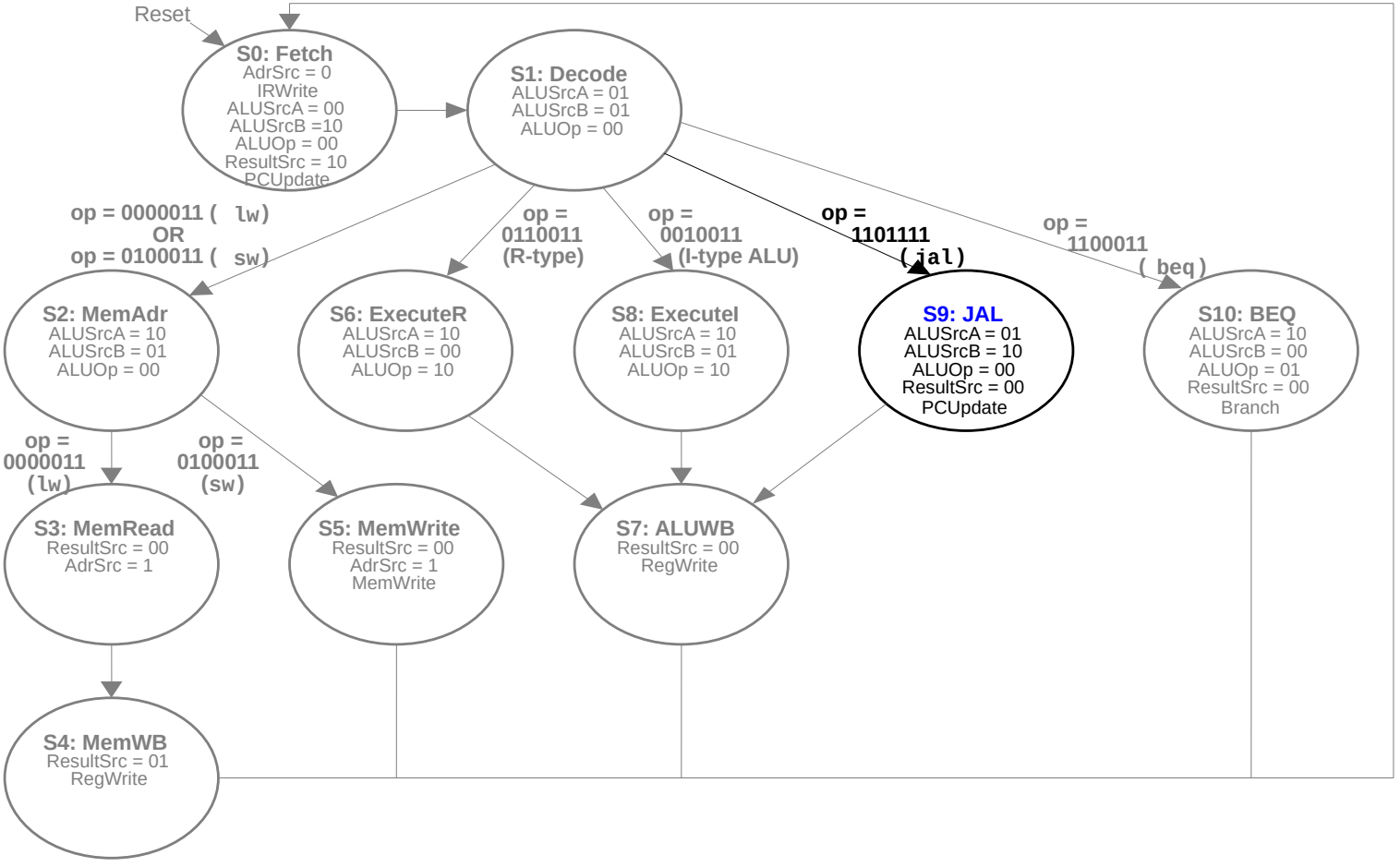
Main FSM: I-Type ALU Execute



Main FSM: I-Type ALU Exec. Datapath



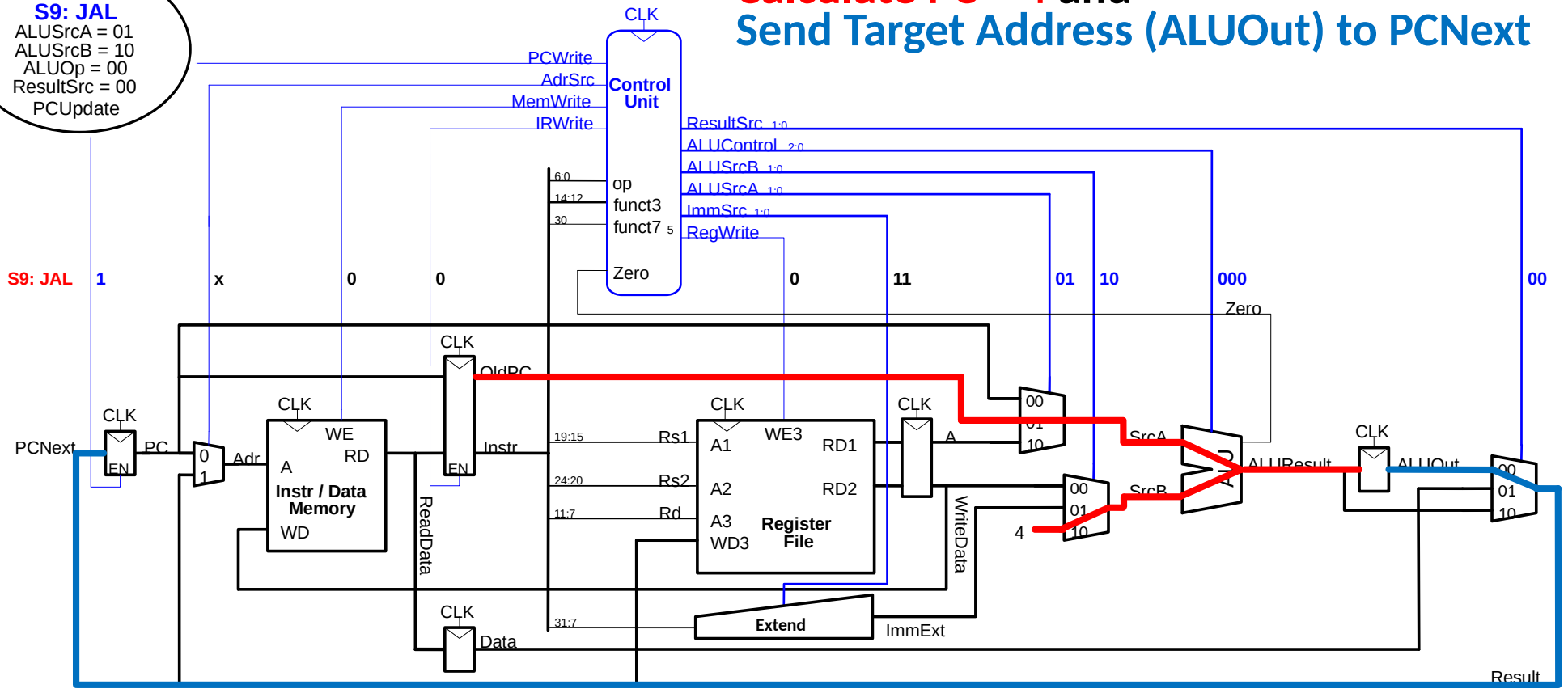
Main FSM: jal


Reset
S0: Fetch
AdrSrc = 0
IRWrite
ALUSrcA = 00
ALUSrcB =10
ALUOp = 00
ResultSrc = 10
PCUpdate
S1: Decode
ALUSrcA = 01
ALUSrcB = 01
ALUOp = 00
op = 0000011 (lw)
OR
op = 0100011 (sw)
op = 0110011 (R-type)
op = 0010011 (I-type ALU)
op = 1101111 (jal)
op = 1100011 (beq)
S2: MemAdr
ALUSrcA = 10
ALUSrcB = 01
ALUOp = 00
S6: ExecuteR
ALUSrcA = 10
ALUSrcB = 00
ALUOp = 10
S8: ExecuteI
ALUSrcA = 10
ALUSrcB = 01
ALUOp = 10
S9: JAL
ALUSrcA = 01
ALUSrcB = 10
ALUOp = 00
ResultSrc = 00
PCUpdate
S10: BEQ
ALUSrcA = 10
ALUSrcB = 00
ALUOp = 01
ResultSrc = 00
Branch
op = 0000011 (lw)
op = 0100011 (sw)
S3: MemRead
ResultSrc = 00
AdrSrc = 1
S5: MemWrite
ResultSrc = 00
AdrSrc = 1
MemWrite
S7: ALUWB
ResultSrc = 00
RegWrite
S4: MemWB
ResultSrc = 01
RegWrite

Main FSM: jal Datapath

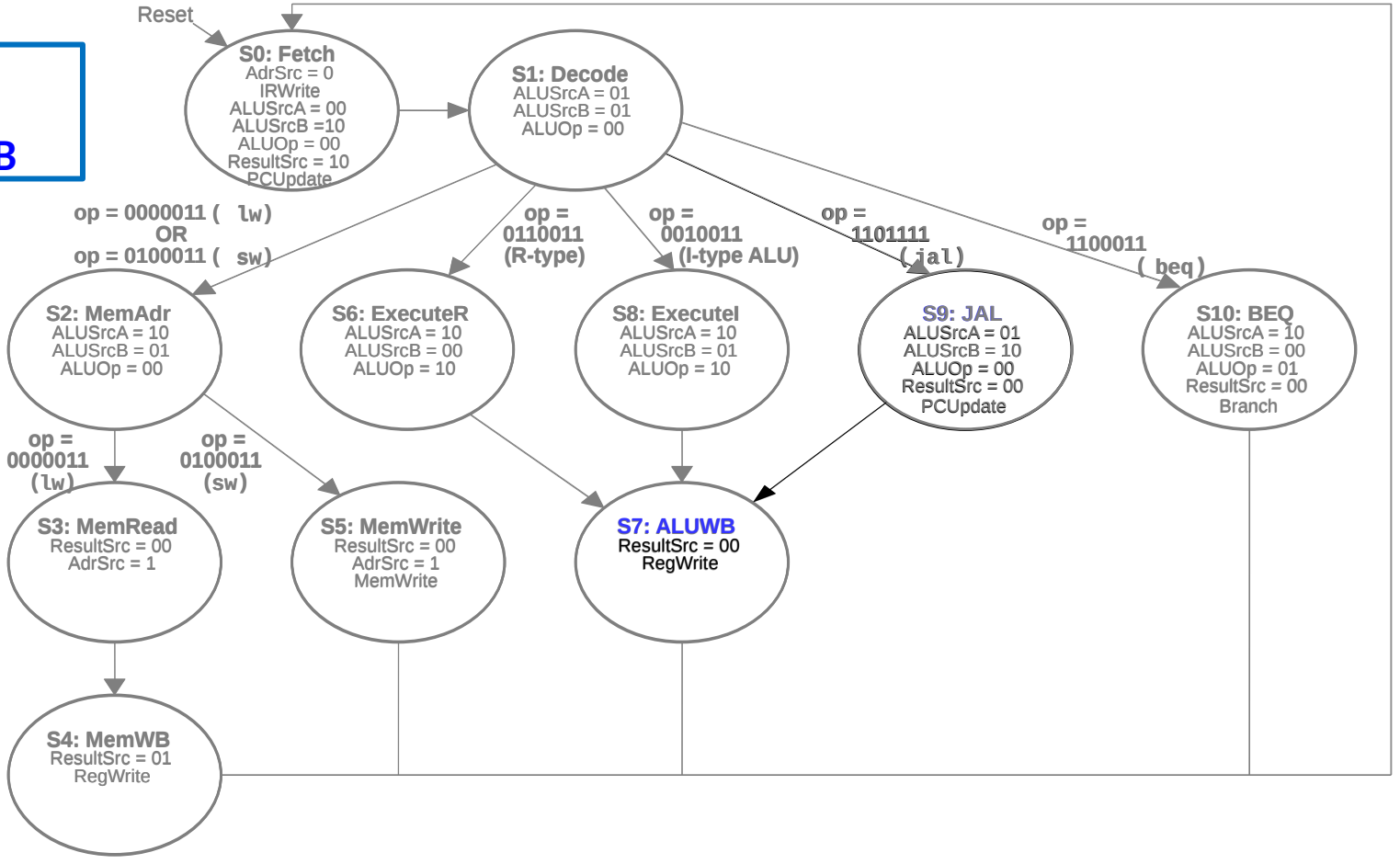
S9: JAL
ALUSrcA = 01
ALUSrcB = 10
ALUOp = 00
ResultSrc = 00
PCUpdate

Calculate PC + 4 and
Send Target Address (ALUOut) to PCNext



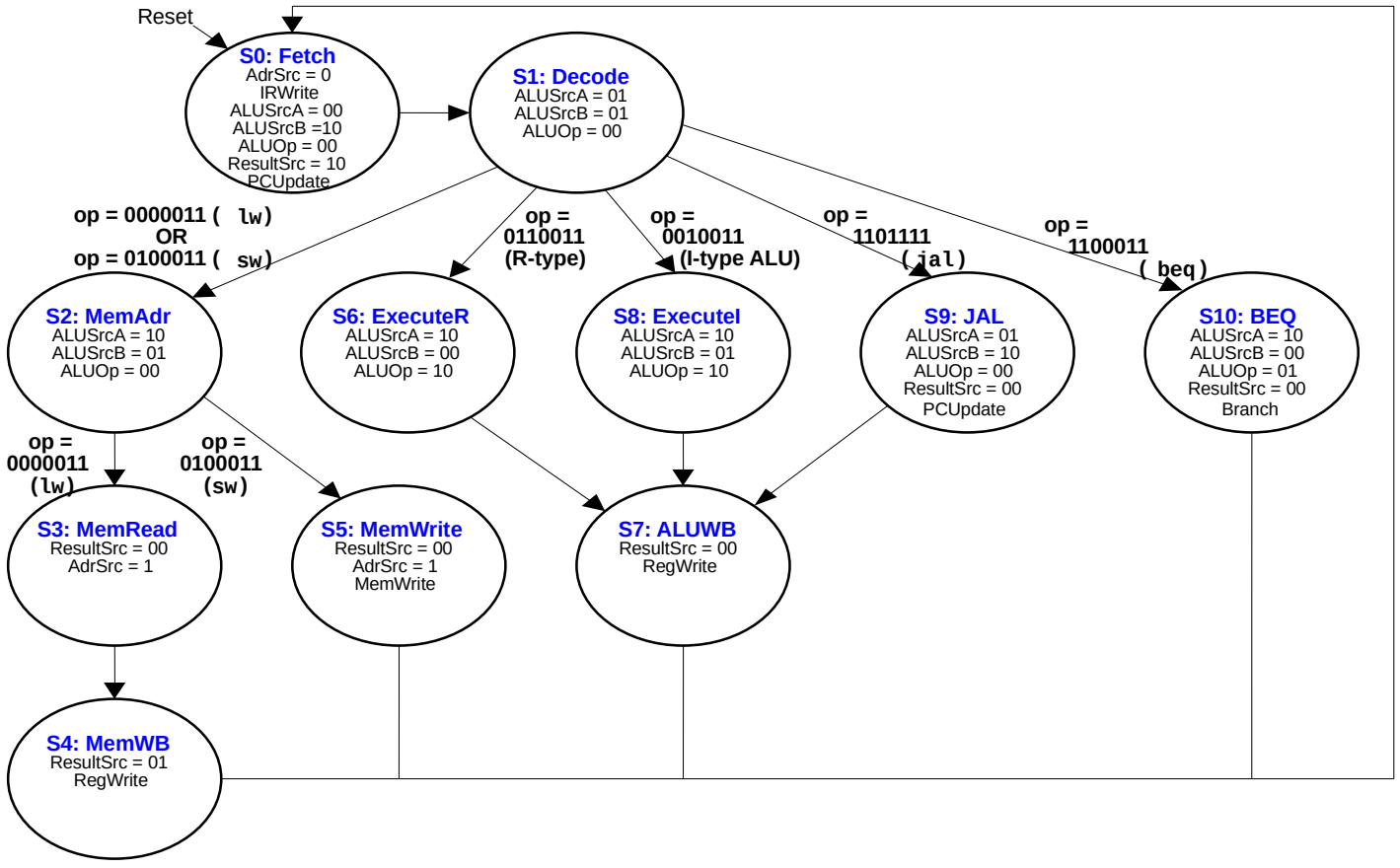
Main FSM: jal

PC + 4 is written to **rd** in **S7: ALUWB**



Multicycle Processor Main FSM

State	Datapath μOp
Fetch	Instr ← Mem[PC]; PC ← PC+4
Decode	ALUOut ← PCTarget
MemAdr	ALUOut ← rs1 + imm
MemRead	Data ← Mem[ALUOut]
MemWB	rd ← Data
MemWrite	Mem[ALUOut] ← rd
ExecuteR	ALUOut ← rs1 op rs2
ExecuteI	ALUOut ← rs1 op imm
ALUWB	rd ← ALUOut
BEQ	ALUResult = rs1-rs2; if Zero, PC ← ALUOut
JAL	PC ← ALUOut; ALUOut ← PC+4



Chapter 7: Microarchitecture

Multicycle Performance

Multicycle Processor Performance

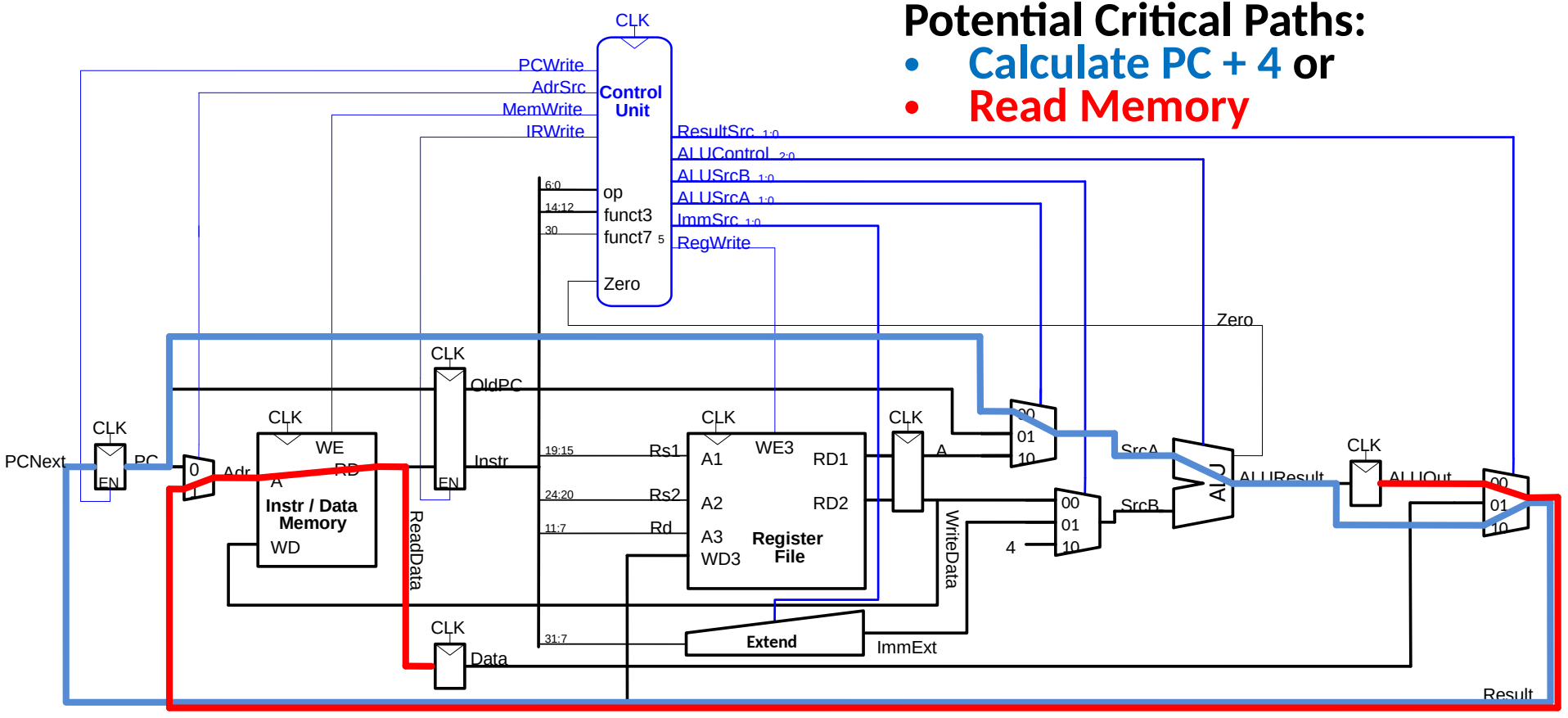
- Instructions take different number of cycles:
  - 3 cycles: `beq`
  - 4 cycles: R-type, `addi`, `sw`, `jal`
  - 5 cycles: `lw`
- CPI is weighted average
- SPECINT2000 benchmark:
  - **25%** loads
  - **10%** stores
  - **13%** branches
  - **52%** R-type

Average CPI = (**0.13**)(3) + (**0.52 + 0.10**)(4) + (**0.25**)(5) = **4.12**

Multicycle Critical Path

Potential Critical Paths:

- **Calculate PC + 4 or**
- **Read Memory**



Multicycle Processor Performance

Multicycle critical path:

- **Assumptions:**
  - RF is faster than memory
  - Writing memory is faster than reading memory

$$T_{c\_multi} = t_{pcq} + t_{\text{dec}} + 2t_{\text{mux}} + \max(t_{\text{ALU}}, t_{\text{mem}}) + t_{\text{setup}}$$

Multicycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	30
AND-OR gate	t_{AND-OR}	20
ALU	t_{ALU}	120
Decoder (Control Unit)	t_{dec}	25
Extend unit	t_{dec}	35
Memory read	t_{mem}	200
Register file read	t_{RFread}	100
Register file setup	$t_{RFsetup}$	60

$$T_{c\_multi} = t_{pcq} + t_{dec} + 2t_{mux} + \max(t_{ALU}, t_{mem}) + t_{setup}$$

$$=$$

Multicycle Performance Example

For a program with **100 billion** instructions executing on a **multicycle** RISC-V processor

- **CPI** = 4.12 cycles/instruction
- **Clock cycle time:** T_{c_multi} = 375 ps

Execution Time = (# instructions) × CPI × T_c

This is slower than the single-cycle processor (75 sec.)

Chapter 7: Microarchitecture

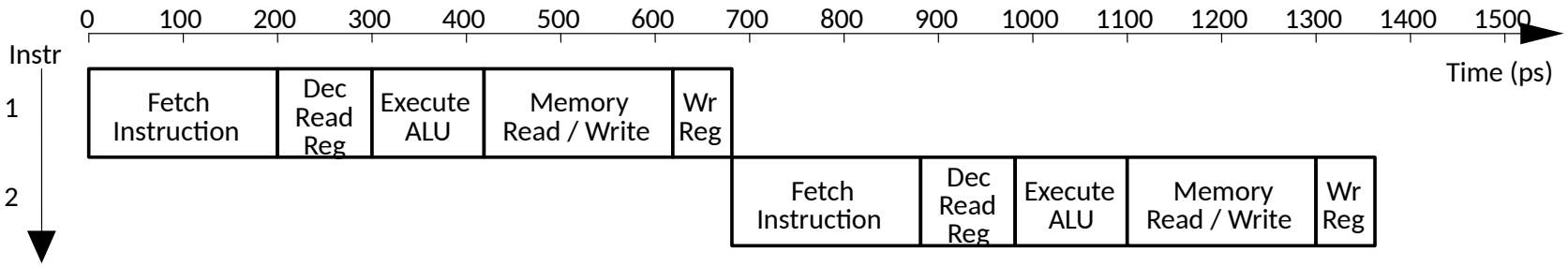
Pipelined RISC-V Processor

Pipelined RISC-V Processor

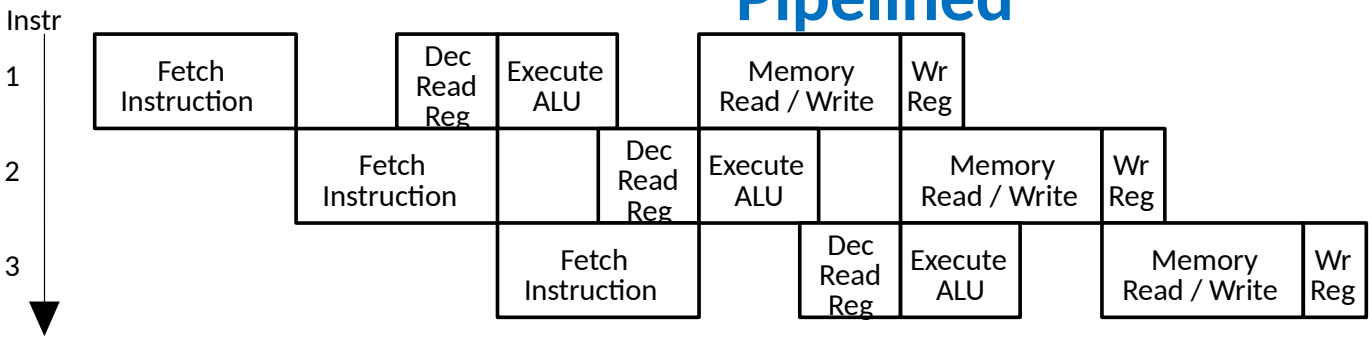
- **Temporal parallelism**
- Divide single-cycle processor into **5 stages**:
  - Fetch
  - Decode
  - Execute
  - Memory
  - Writeback
- Add **pipeline registers** between stages

Single-Cycle vs. Pipelined Processor

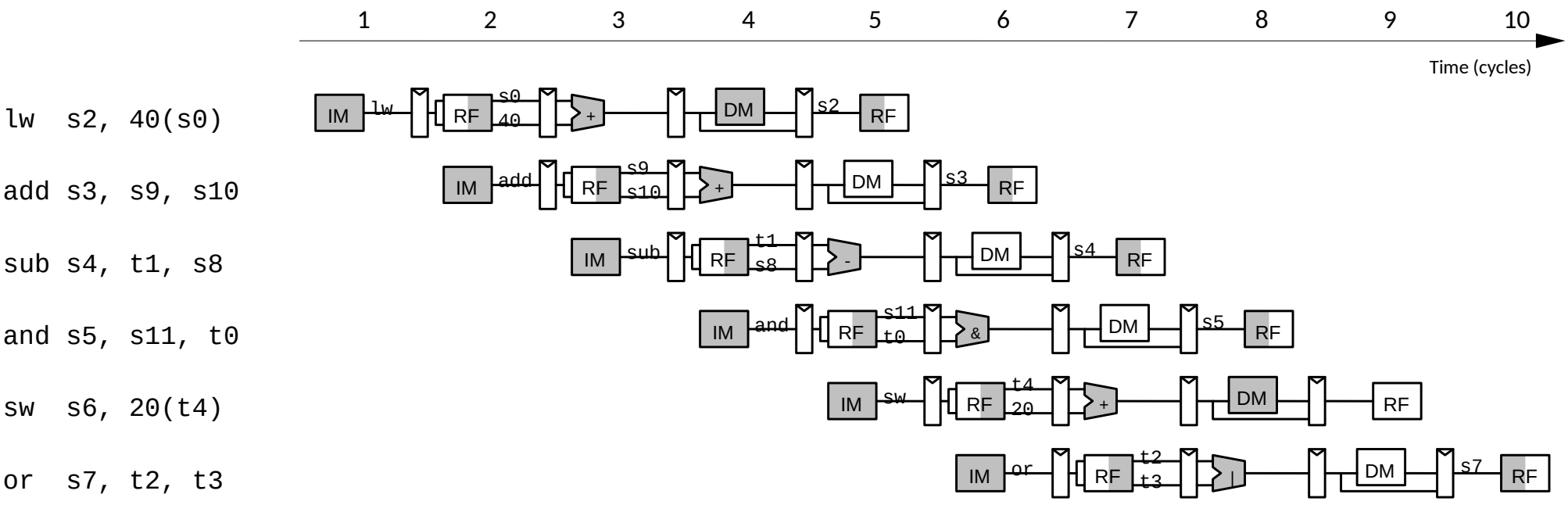
Single-Cycle



Pipelined

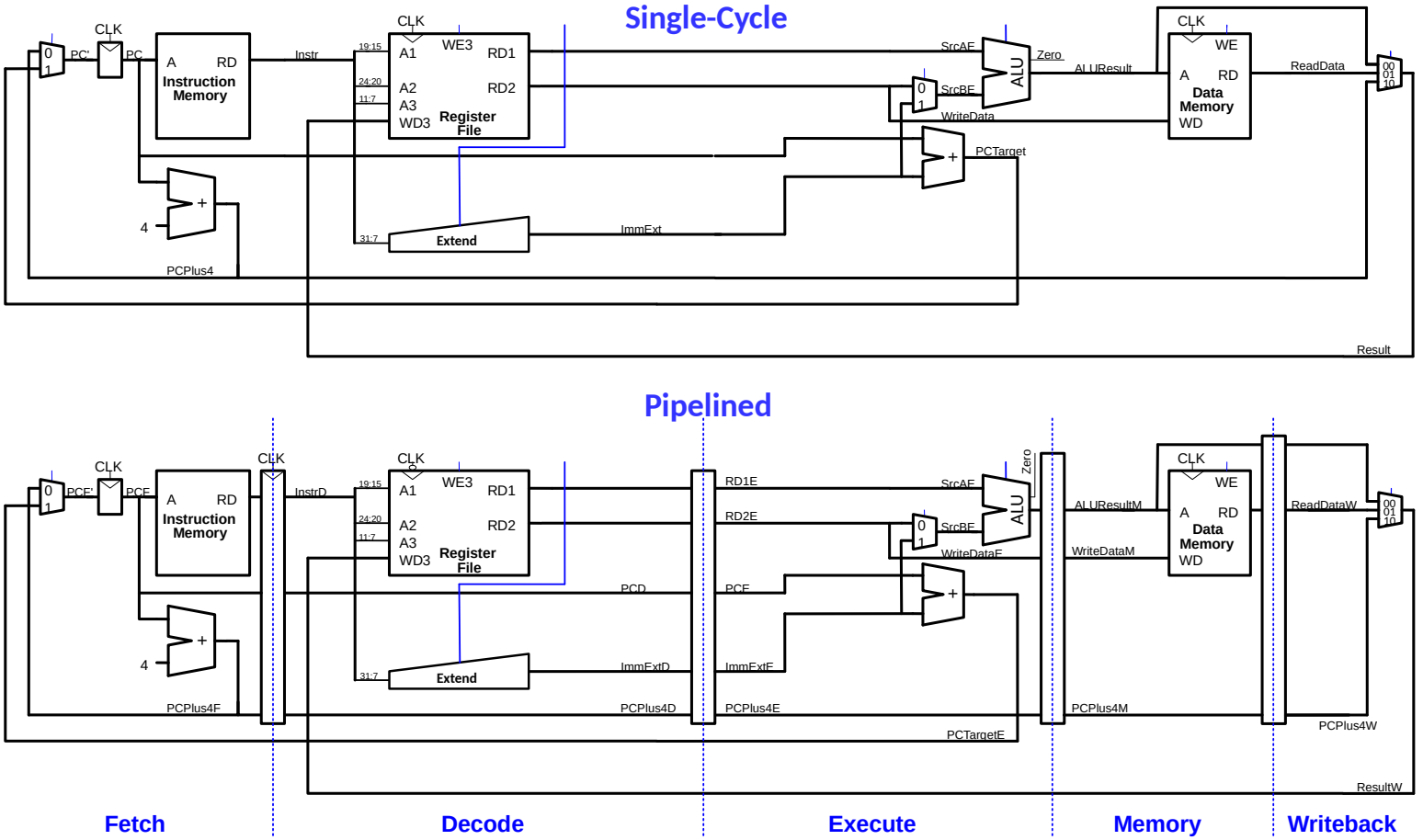


Pipelined Processor Abstraction

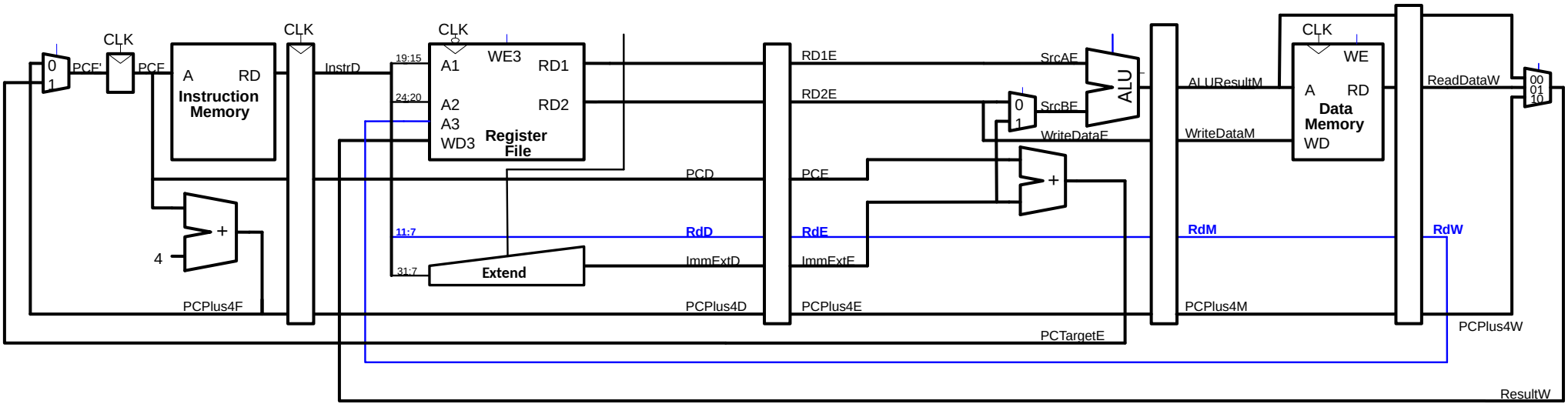


Single-Cycle & Pipelined Datapaths

Signals in Pipelined Processor are appended with first letter of stage (i.e., PC**F**, PC**D**, PC**E**).

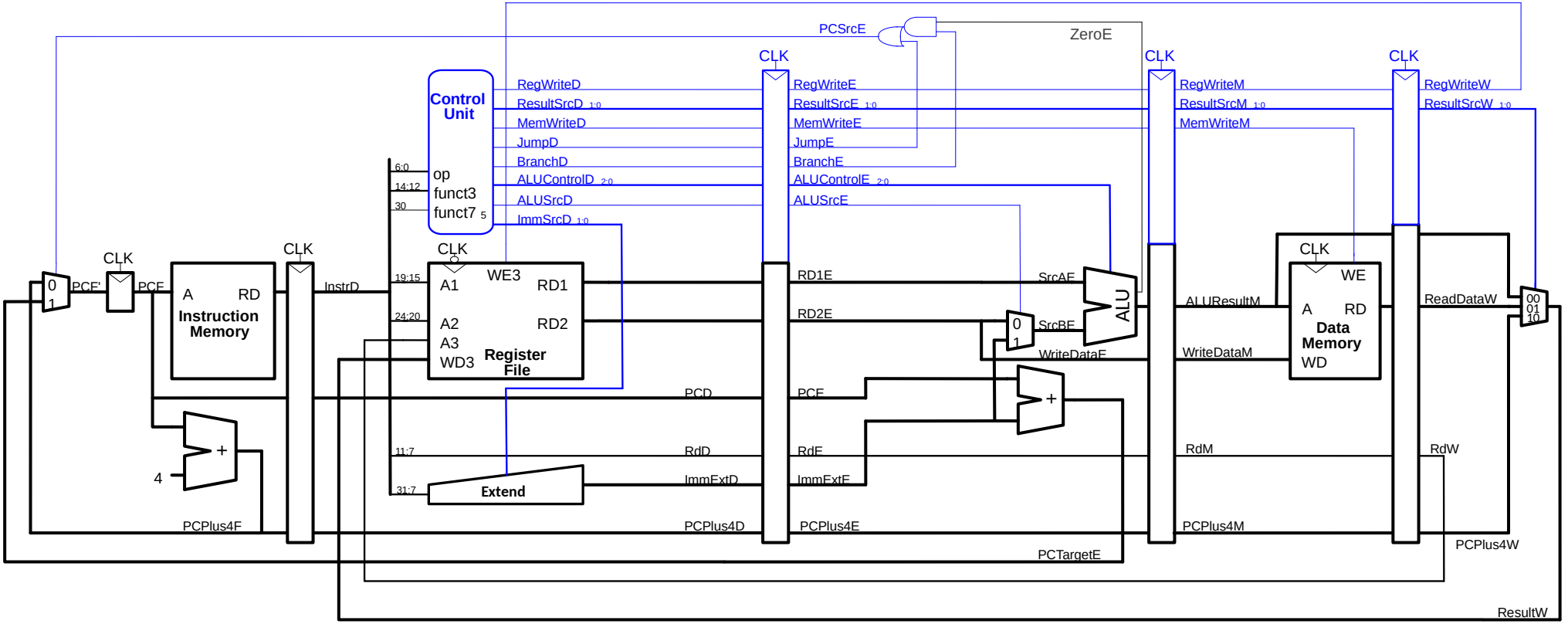


Corrected Pipelined Datapath



- ***Rd*** must arrive at same time as ***Result***
- Register file written on **falling edge** of *CLK*

Pipelined Processor with Control



- **Same control unit** as single-cycle processor
- **Control signals travel with** the instruction (drop off when used)

Chapter 7: Microarchitecture

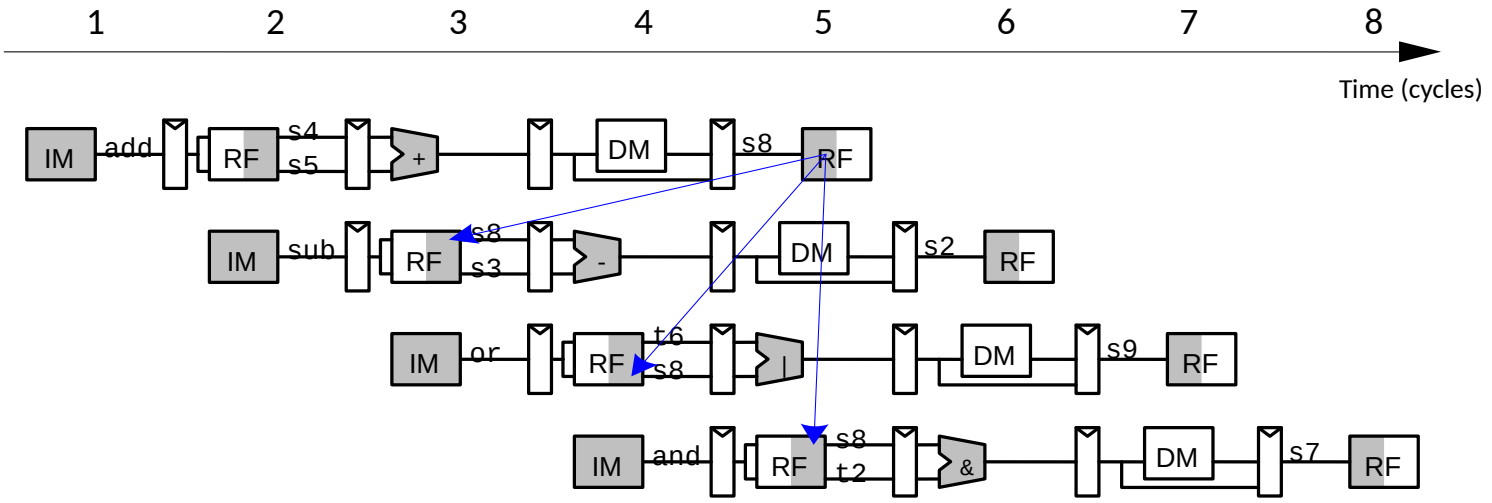
Pipelined Processor Hazards

Pipelined Hazards

- When an instruction depends on result from instruction that hasn't completed
- Types:
  - **Data hazard:** register value not yet written back to register file
  - **Control hazard:** next instruction not decided yet (caused by branch)

Data Hazard

```
add s8, s4, s5
sub s2, s8, s3
or  s9, t6, s8
and s7, s8, t2
```



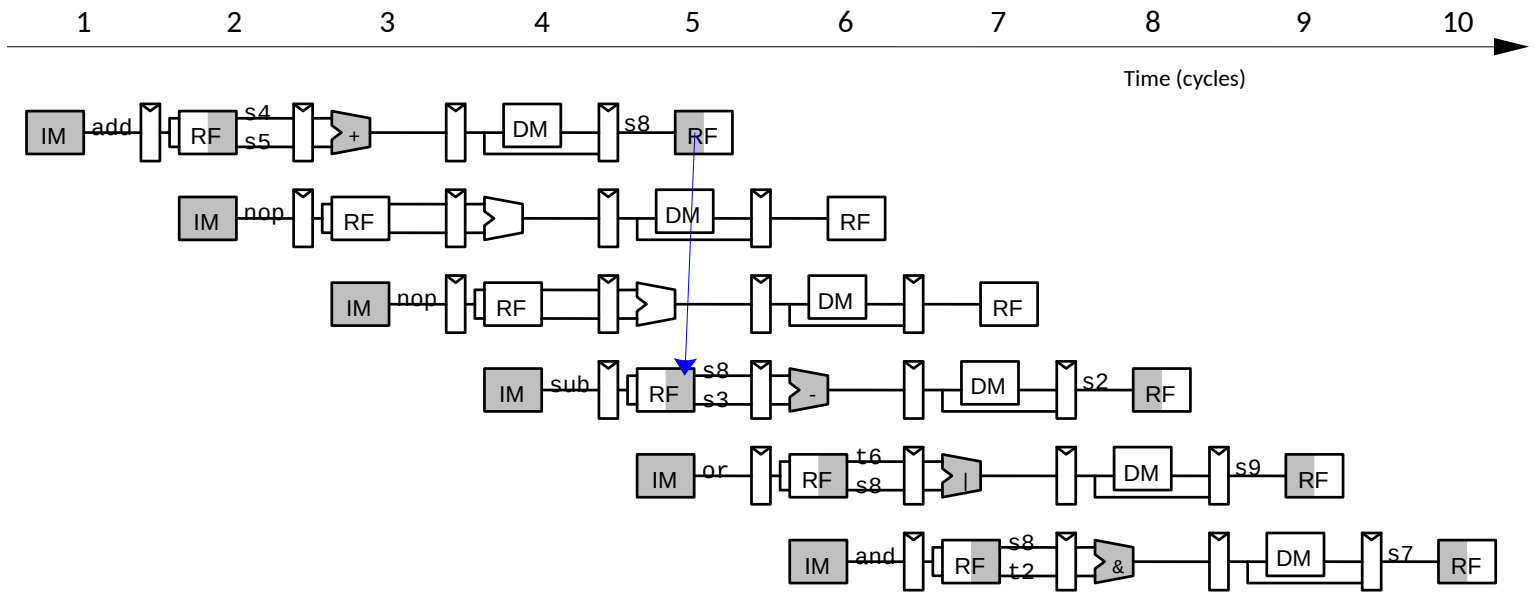
Handling Data Hazards

Stall the processor at run time

Handling Data Hazards

- **Insert** enough **nops** for result to be ready
- Or move independent useful instructions forward

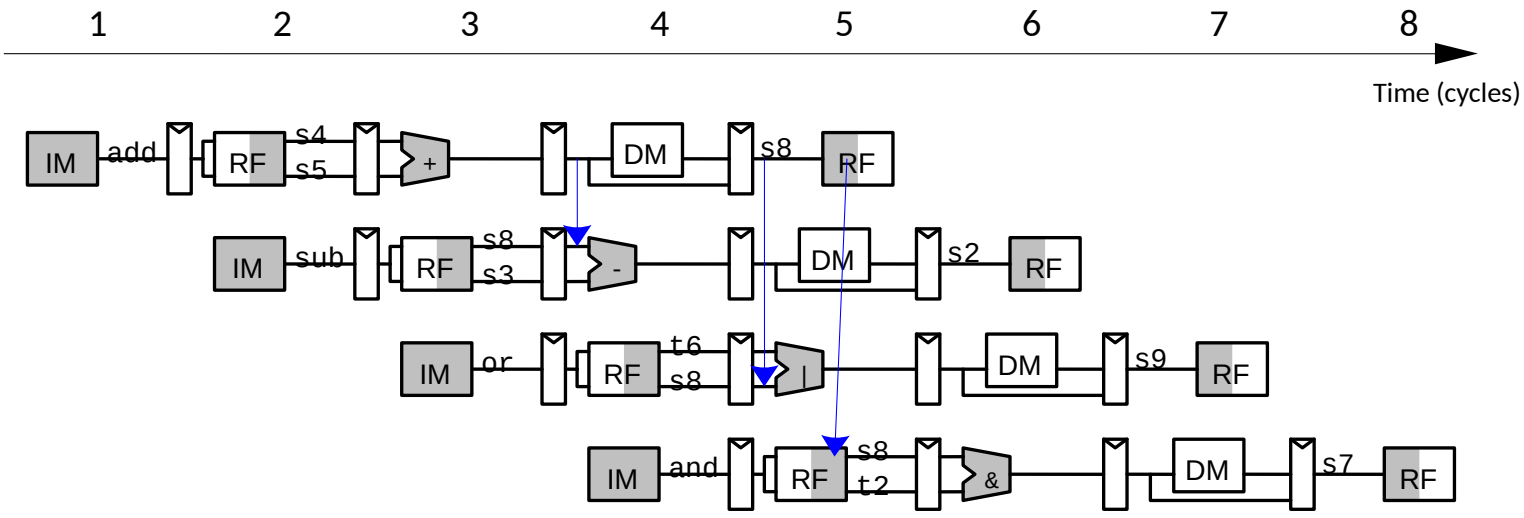
```
add s8, s4, s5
nop
nop
sub s2, s8, s3
or  s9, t6, s8
and s7, s8, t2
```



Data Forwarding

- Data is **available on internal busses** before it is written back to the register file (RF).
- **Forward data** from internal busses **to Execute stage**.

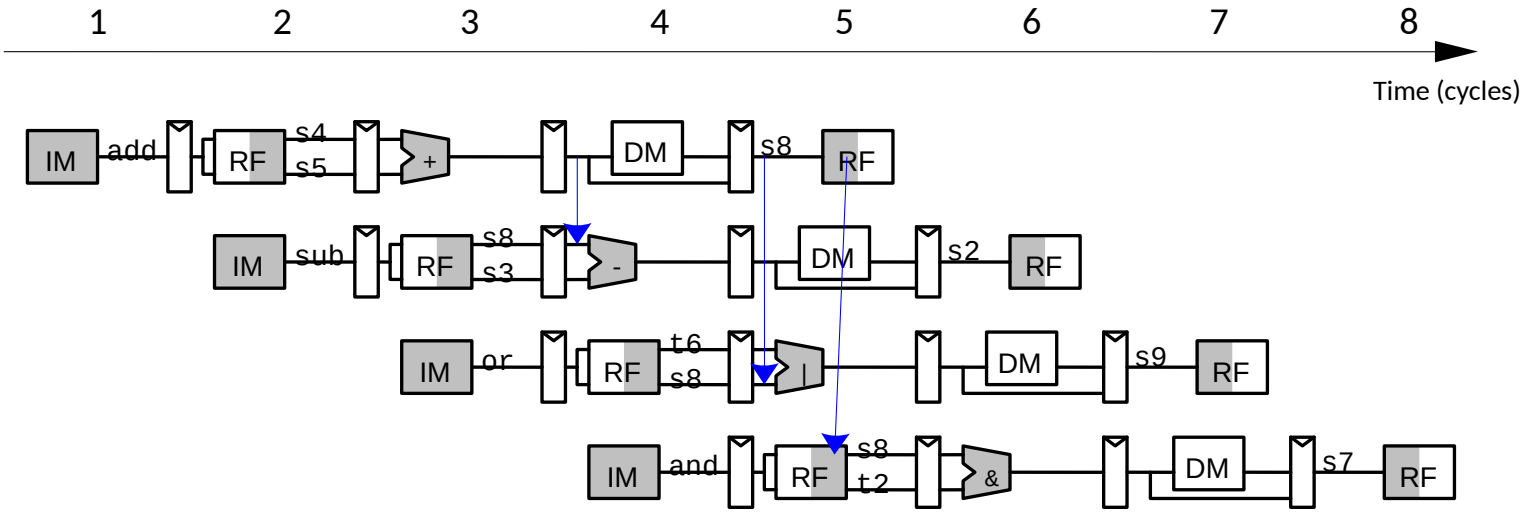
```
add s8, s4, s5
sub s2, s8, s3
or  s9, t6, s8
and s7, s8, t2
```



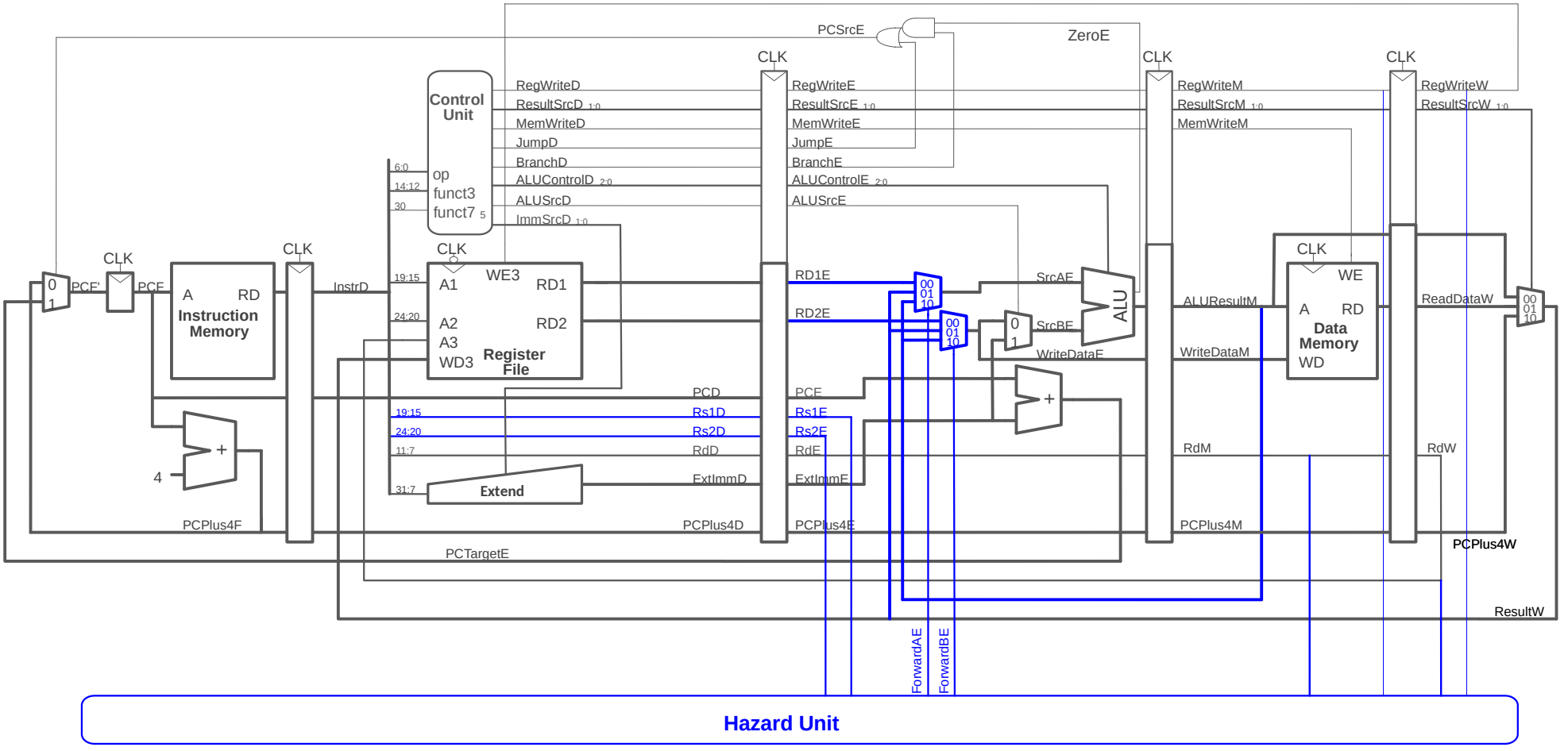
Data Forwarding

- Check if source register **in Execute stage** **matches** destination register of instruction **in Memory or Writeback stage**.
- If so, forward result.

```
add s8, s4, s5
sub s2, s8, s3
or  s9, t6, s8
and s7, s8, t2
```



Data Forwarding: Hazard Unit



Data Forwarding

- **Case 1:** **Execute** stage *Rs1* or *Rs2* matches **Memory** stage *Rd*?
  Forward from Memory stage
- **Case 2:** **Execute** stage *Rs1* or *Rs2* matches **Writeback** stage *Rd*?
  Forward from Writeback stage
- **Case 3:** Otherwise use value read from register file (as usual)

Equations for *Rs1*:

```
if      ((Rs1E == RdM) AND RegWriteM)        // Case 1
            ForwardAE = 10
else if ((Rs1E == RdW) AND RegWriteW)        // Case 2
            ForwardAE = 01
else        ForwardAE = 00                   // Case 3
```

***ForwardBE** equations are similar (replace **Rs1E** with **Rs2E**)*

Data Forwarding

- **Case 1:** **Execute** stage *Rs1* or *Rs2* matches **Memory** stage *Rd*?
  Forward from Memory stage
- **Case 2:** **Execute** stage *Rs1* or *Rs2* matches **Writeback** stage *Rd*?
  Forward from Writeback stage
- **Case 3:** Otherwise use value read from register file (as usual)

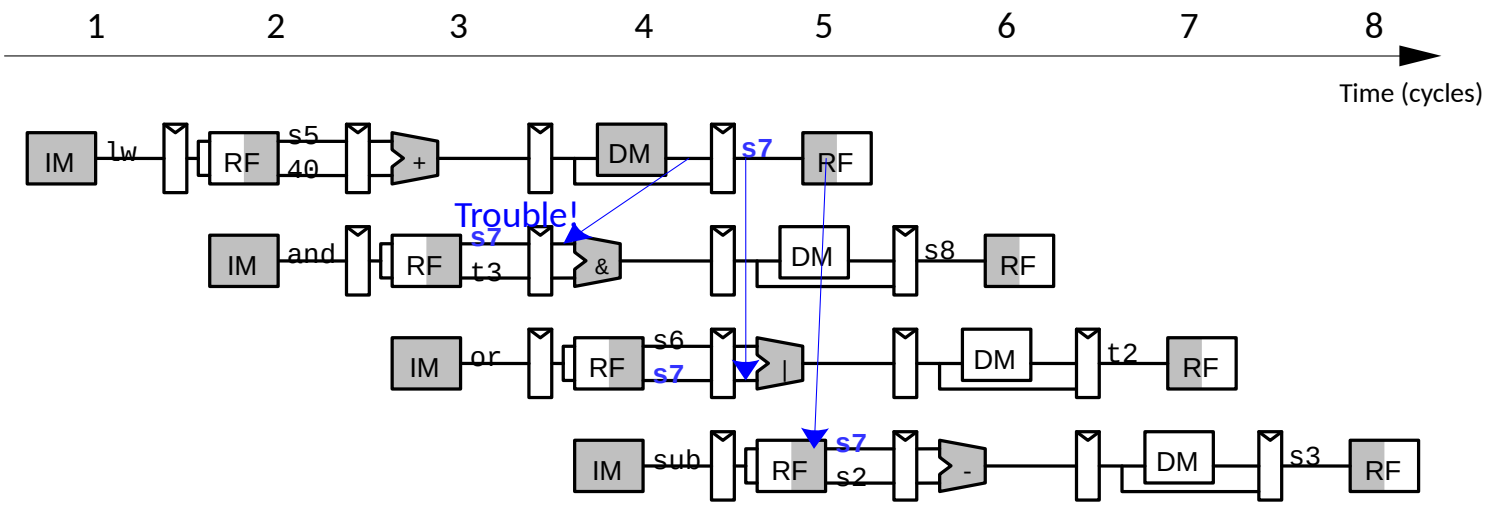
Equations for *Rs1*:

if ((*Rs1E* == *RdM*) AND *RegWriteM*) AND (Rs1E != 0) // Case 1
  ForwardAE = 10
else if ((*Rs1E* == *RdW*) AND *RegWriteW*) AND (Rs1E != 0) // Case 2
  ForwardAE = 01
else *ForwardAE* = 00 // Case 3

ForwardBE *equations are similar (replace* ***Rs1E*** *with* ***Rs2E****)*

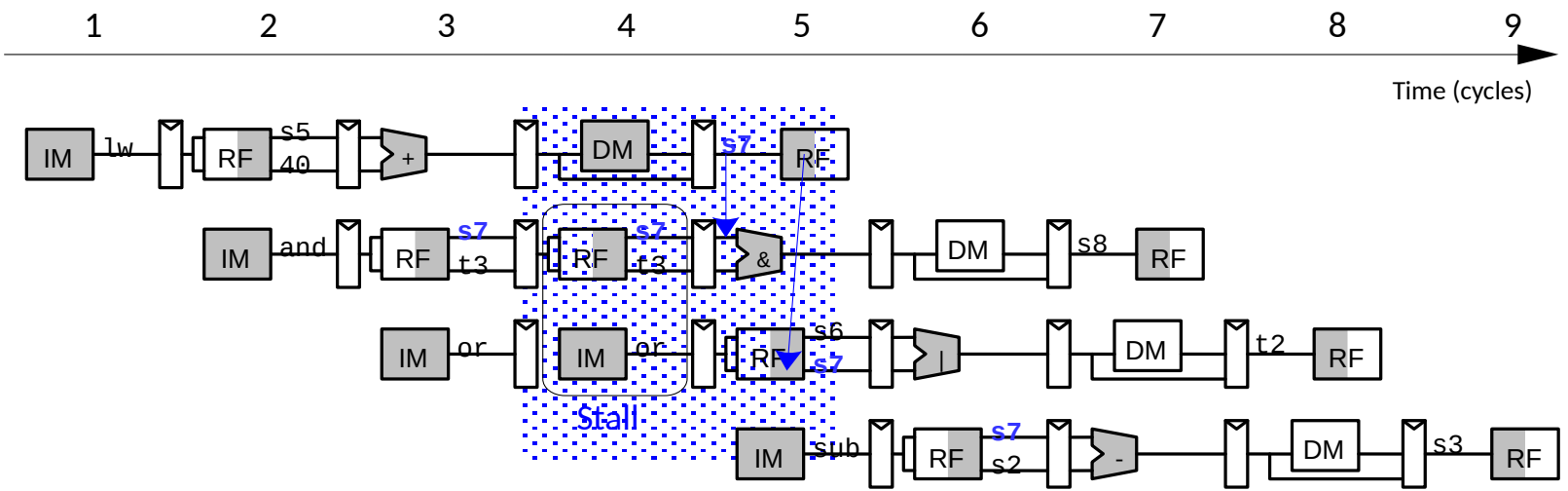
Data Hazard due to `lw` Dependency

```
lw  s7, 40(s5)
and s8, s7, t3
or  t2, s6, s7
sub s3, s7, s2
```



Stalling to solve `lw` Data Dependency

```
lw  s7, 40(s5)
and s8, s7, t3
or  t2, s6, s7
sub s3, s7, s2
```



Stalling Logic

- Is either **source register in the Decode stage** the same as the **destination register in the Execute stage**?

  AND

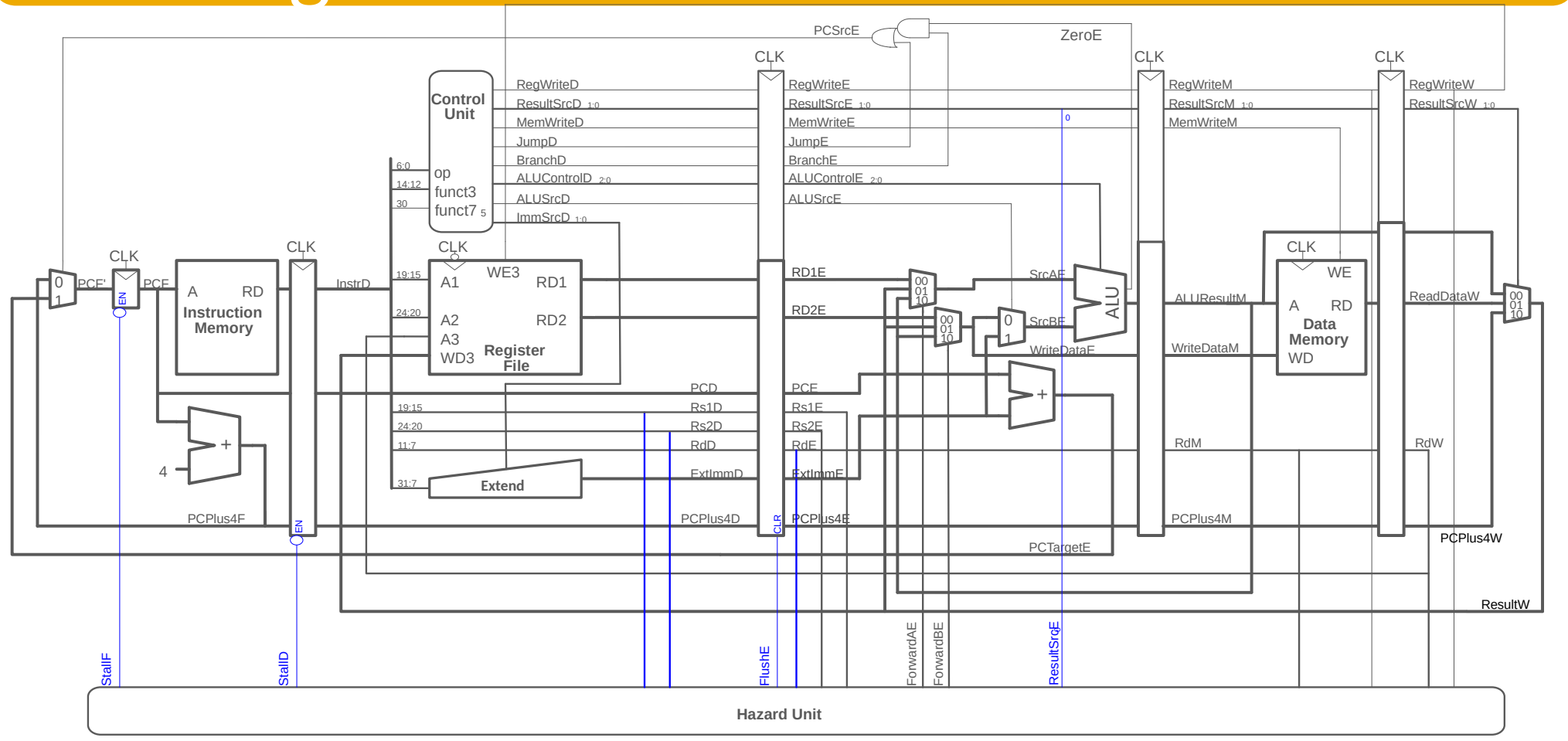
- Is the instruction in the **Execute stage a** `lw`?

lwStall = ((*Rs1D* == *RdE*) OR (*Rs2D* == *RdE*)) AND *ResultSrcE*$_0$

StallF = ***StallD*** = ***FlushE*** = ***lwStall***

(Stall the Fetch and Decode stages, and flush the Execute stage.)

Stalling Hardware



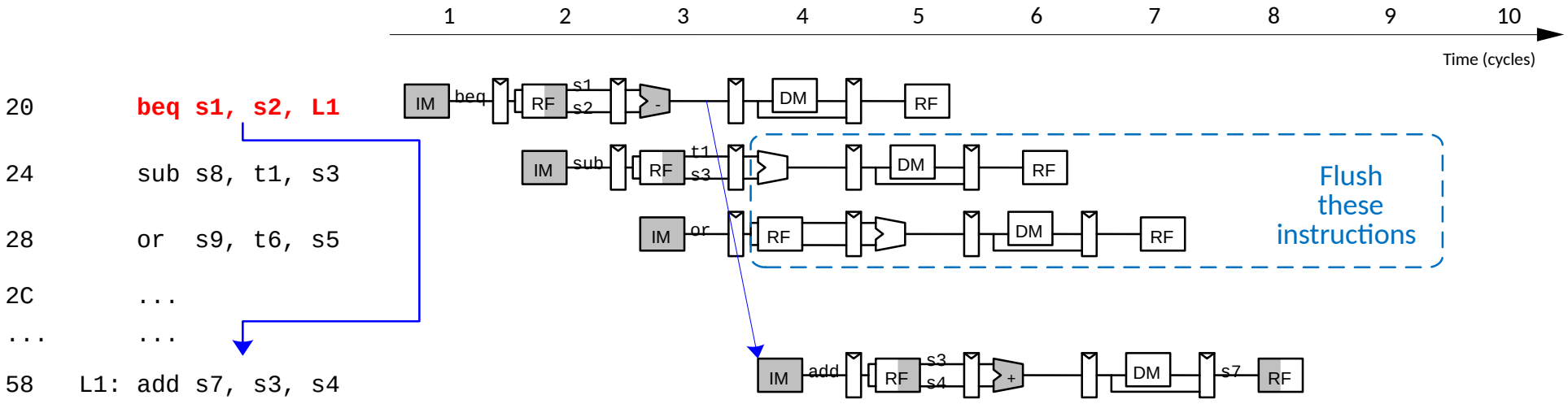
Chapter 7: Microarchitecture

Pipelined Processor Control Hazards

Control Hazards

- **beq**:
  - Branch **not determined until the Execute stage** of pipeline
  - **Instructions** after branch **fetched** before branch occurs
  - These **2 instructions must be flushed** if branch happens

Control Hazards



Branch misprediction penalty:

The number of instructions flushed when a branch is taken (in this case, 2 instructions)

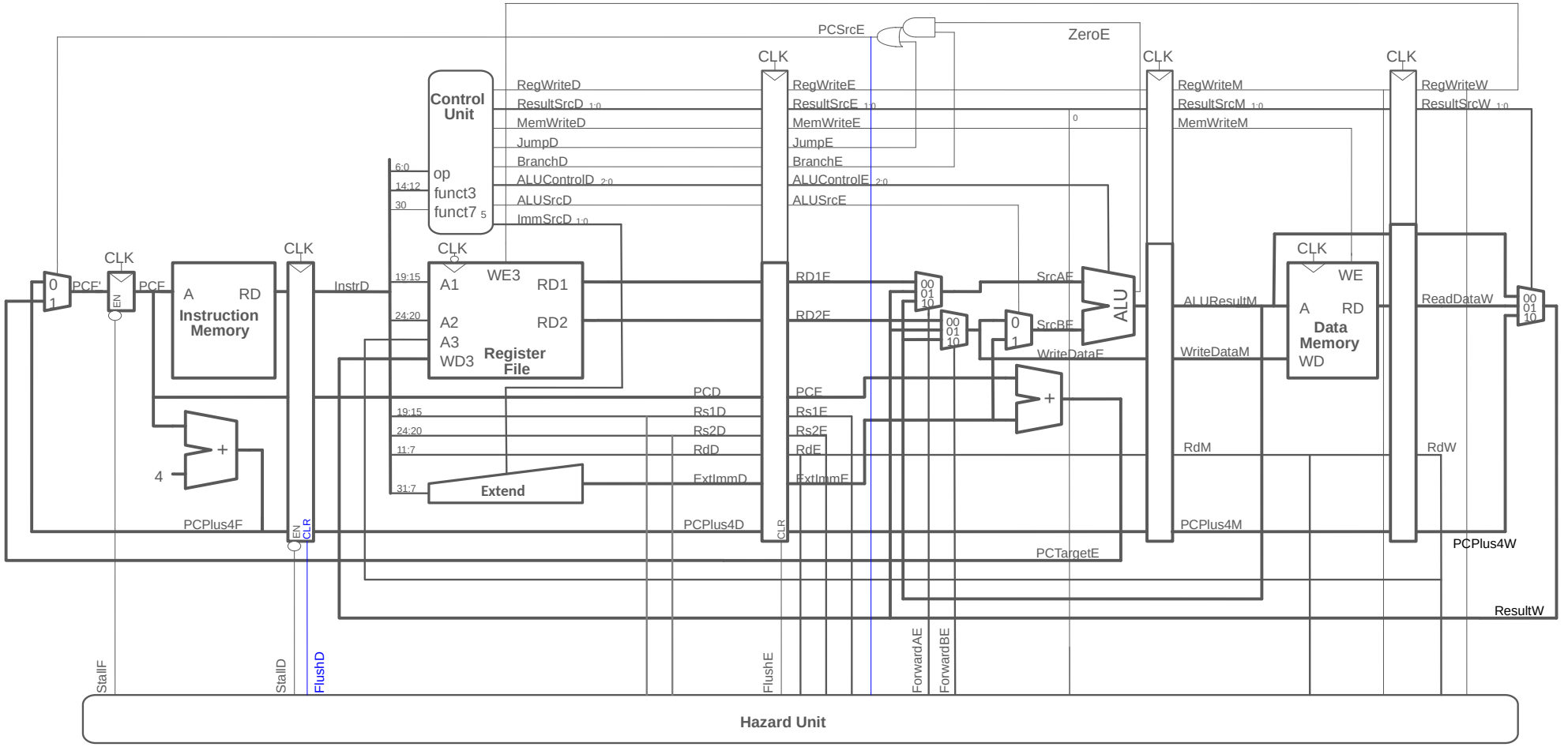
Control Hazards: Flushing Logic

- If branch is taken in execute stage, need to flush the instructions in the Fetch and Decode stages
  - Do this by clearing Decode and Execute Pipeline registers using *FlushD* and *FlushE*
- **Equations:**

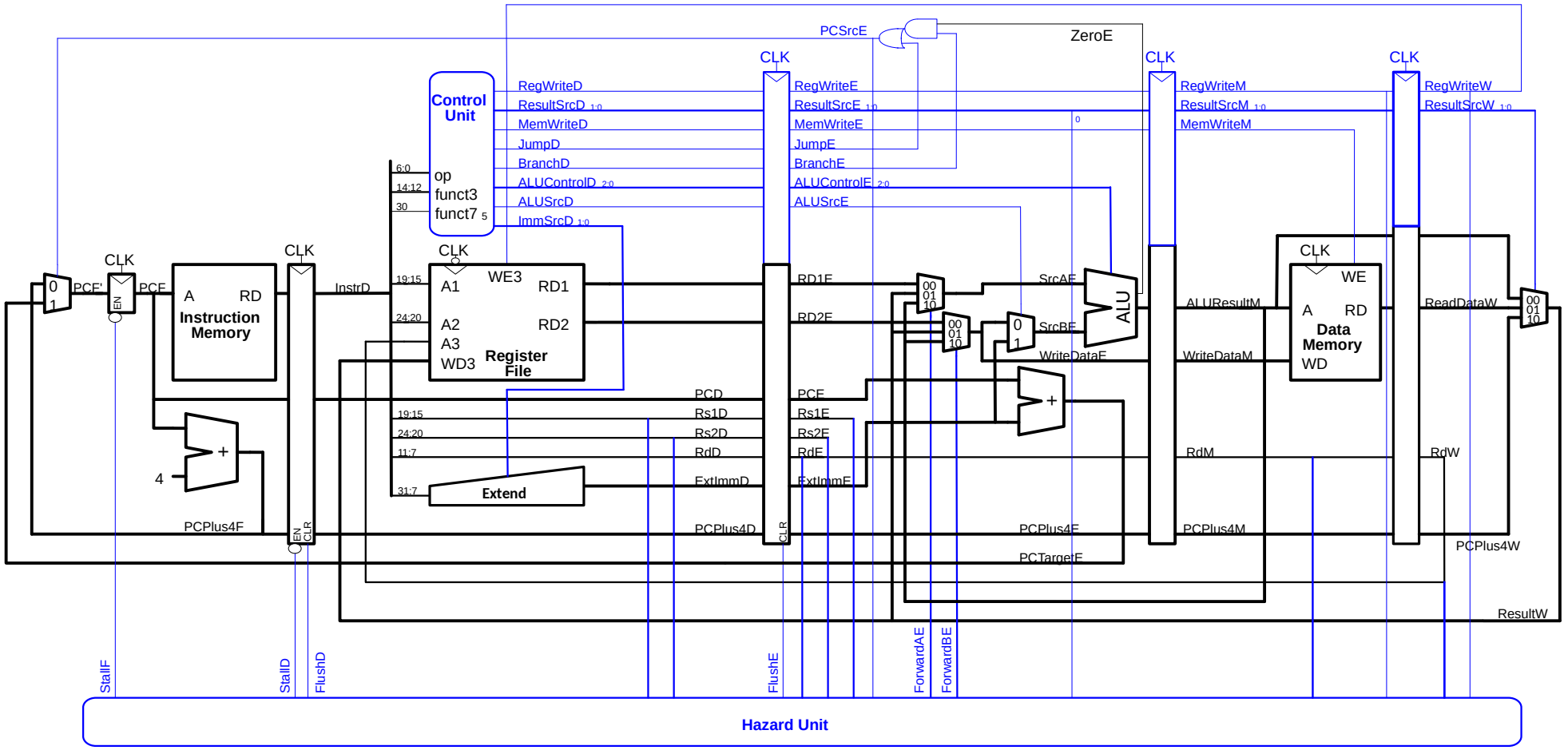
  FlushD = *PCSrcE*

  FlushE = *lwStall* OR *PCSrcE*

Control Hazards: Flushing Hardware



RISC-V Pipelined Processor with Hazard Unit



Summary of Hazard Logic

Data hazard logic (shown for SrcA of ALU):

if ((*Rs1E* == *RdM*) AND *RegWriteM*) AND (Rs1E != 0) // Case 1
ForwardAE = 10
else if ((*Rs1E* == *RdW*) AND *RegWriteW*) AND (Rs1E != 0) // Case 2
ForwardAE = 01
else ***ForwardAE*** = 00 // Case 3

Load word stall logic:

lwStall = ((*Rs1D* == *RdE*) OR (*Rs2D* == *RdE*)) AND *ResultSrcE*$_0$
StallF = ***StallD*** = *lwStall*

Control hazard flush:

FlushD = *PCSrcE*
FlushE = *lwStall* OR *PCSrcE*

Chapter 7: Microarchitecture

Pipelined Performance

Pipelined Processor Performance Example

- **SPECINT2000 benchmark:**
  - 25% loads
  - 10% stores
  - 13% branches
  - 52% R-type
- **Suppose:**
  - 40% of loads used by next instruction
  - 50% of branches mispredicted
- **What is the average CPI?** (Ideally it's 1, but...)

= 1.23

Pipelined Processor Performance Example

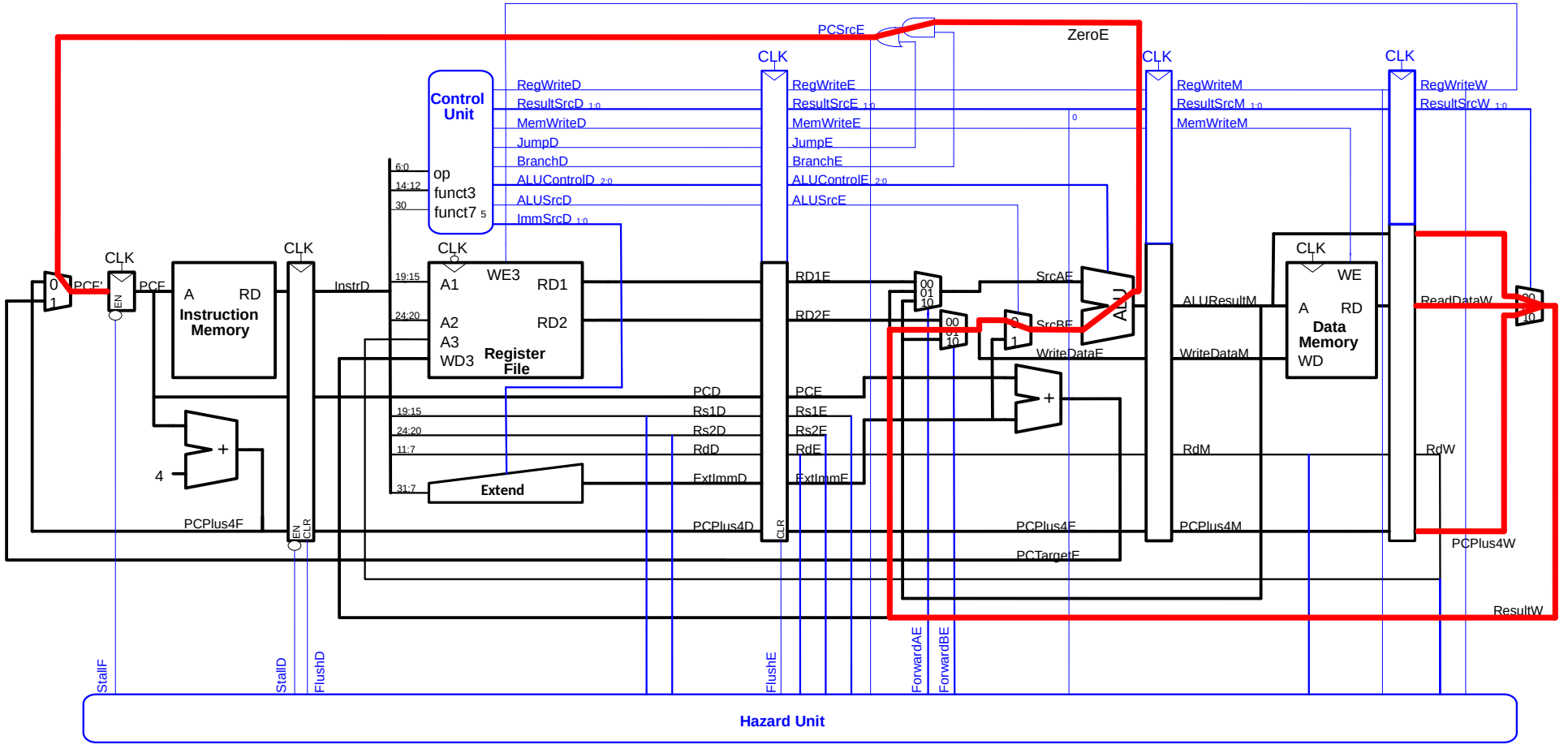
Pipelined processor critical path:

$T_{c\_pipelined}$ = max of

$t_{pcq} + t_{mem} + t_{setup}$	**Fetch**
$2(t_{RFread} + t_{setup})$	**Decode**
$t_{pcq} + 4t_{mux} + t_{ALU} + t_{AND\text{-}OR} + t_{setup}$	**Execute**
$t_{pcq} + t_{mem} + t_{setup}$	**Memory**
$2(t_{pcq} + t_{mux} + t_{RFwrite})$	**Writeback**

- Decode and Writeback stages **both use the register file** in each cycle
- So each stage gets half of the cycle time ($T_c/2$) to do their work
- Or, stated a different way, **2x of their work** must fit in a cycle (T_c)

Pipelined Critical Path: Execute Stage



Pipelined Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	30
AND-OR gate	t_{AND-OR}	20
ALU	t_{ALU}	120
Decoder (Control Unit)	t_{dec}	25
Extend unit	t_{dec}	35
Memory read	t_{mem}	200
Register file read	t_{RFread}	100
Register file setup	$t_{RFsetup}$	60

$$T_{c\_pipelined} = t_{pcq} + 4t_{mux} + t_{ALU} + t_{AND-OR} + t_{setup}$$

$$=$$

Pipelined Performance Example

Program with 100 billion instructions

Execution Time = (# instructions) × CPI × T_c

= $(100 \times 10^9)(1.23)(350 \times 10^{-12})$

= 43 seconds

Processor Performance Comparison

Processor	Execution Time (seconds)	Speedup (single-cycle as baseline)
Single-cycle	75	1
Multicycle	155	0.5
Pipelined	43	1.7

Chapter 7: Microarchitecture

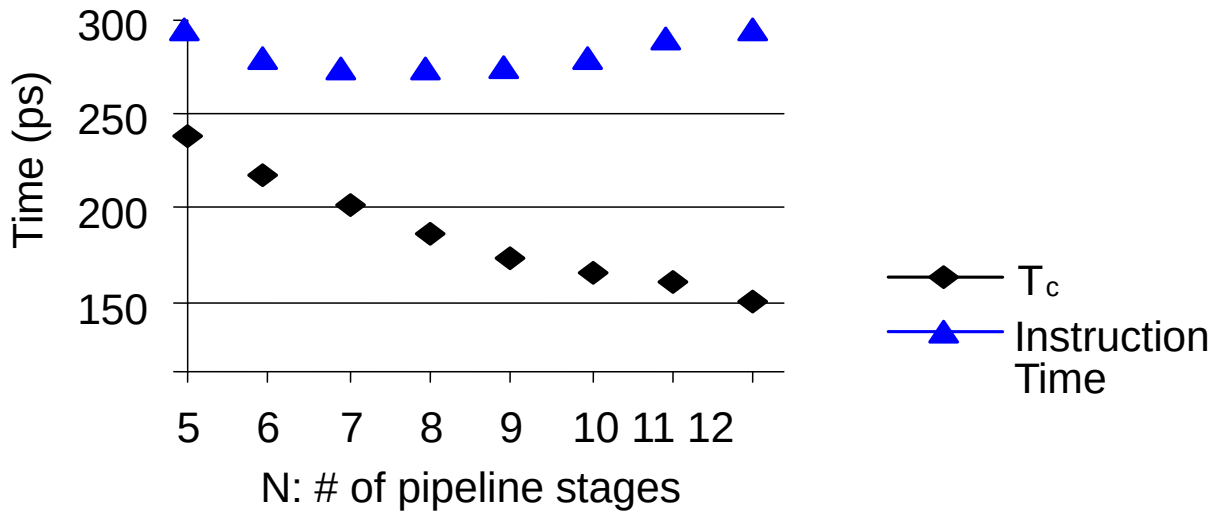
Advanced Microarchitecture

Advanced Microarchitecture

- Deep Pipelining
- Micro-operations
- Branch Prediction
- Superscalar Processors
- Out of Order Processors
- Register Renaming
- SIMD
- Multithreading
- Multiprocessors

Deep Pipelining

- **10-20 stages typical**
- Number of stages limited by:
  - Pipeline hazards
  - Sequencing overhead
  - Power
  - Cost



Micro-operations

- Decompose complex instructions into series of simple instructions called ***micro-operations*** (*micro-ops* or *μ-ops*)
- **At run-time**, complex instructions are decoded into one or more micro-ops
- Used heavily in **CISC** (complex instruction set computer) architectures (e.g., x86)

Complex Op

```
lw s1, 0(s2), postincr 4
```

Micro-op Sequence

```
lw   s1, 0(s2)
addi s2, s2, 4
```

Without μ-ops, would need 2nd write port on the register file

Branch Prediction

- **Guess** whether branch will be taken
  - Backward branches are usually taken (loops)
  - Consider history to improve guess
- Good prediction **reduces fraction of branches requiring a flush**

Branch Prediction

- Ideal pipelined processor: CPI = 1
- Branch misprediction increases CPI
- **Static branch prediction:**
  - Check direction of branch (forward or backward)
  - If backward, predict taken
  - Else, predict not taken
- **Dynamic branch prediction:**
  - Keep **history** of last several hundred (or thousand) branches in *branch target buffer*, record:
    - Branch destination
    - Whether branch was taken

Dynamic Branch Prediction

- 1-bit branch predictor
- 2-bit branch predictor

Branch Prediction Example

```
    addi s1, zero, 0          # s1 = sum
    addi s0, zero, 0          # s0 = i
    addi t0, zero, 10      # t0 = 10

For:                          # for (i=0; i<10; i=i+1)
    bge   s0, t0, Done
    add   s1, s1, s0          # sum = sum + i
    addi s0, s0, 1            # i = i + 1
    j     For

Done:
```

1-Bit Branch Predictor

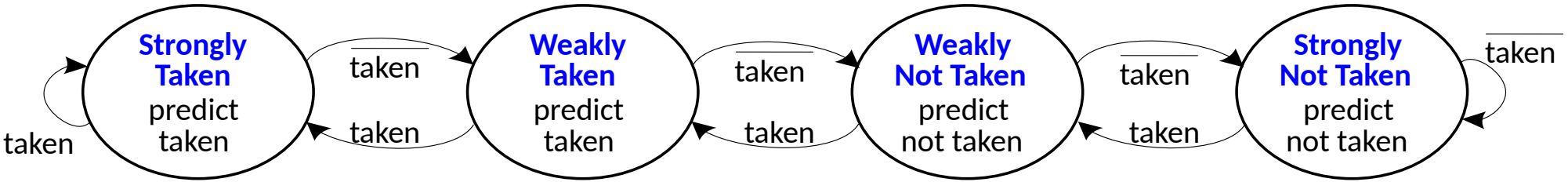
- **Remembers** whether branch was taken the last time and **does the same thing**
- Mispredicts first and last branch of loop

```
    addi s1, zero, 0      # s1 = sum
    addi s0, zero, 0      # s0 = i
    addi t0, zero, 10     # t0 = 10

For:                      # for (i=0; i<10; i=i+1)
    bge   s0, t0, Done
    add   s1, s1, s0      # sum = sum + i
    addi s0, s0, 1        # i = i + 1
    j     For

Done:
```

2-Bit Branch Predictor




```
   addi s1, zero, 0      # s1 = sum
   addi s0, zero, 0      # s0 = i
   addi t0, zero, 10     # t0 = 10

For:                     # for (i=0; i<10; i=i+1)
   bge   s0, t0, Done
   add   s1, s1, s0      # sum = sum + i
   addi s0, s0, 1        # i = i + 1
   j     For

Done:
```

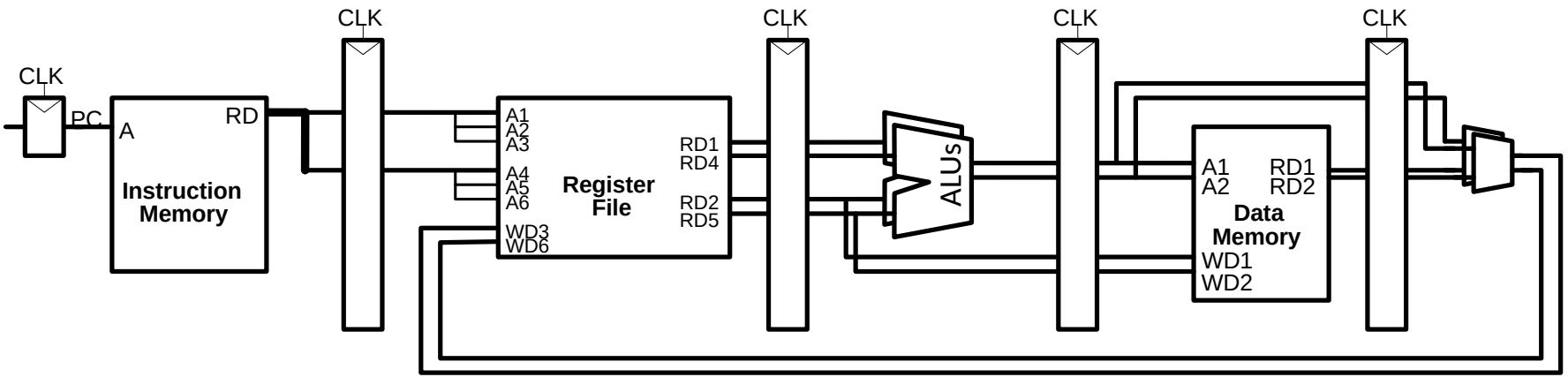
Only mispredicts **last branch** of loop

Chapter 7: Microarchitecture

Superscalar & Out of Order Processors

Superscalar Processors

- Multiple copies of datapath execute multiple instructions at once
- Dependencies make it tricky to issue multiple instructions at once



Superscalar Example

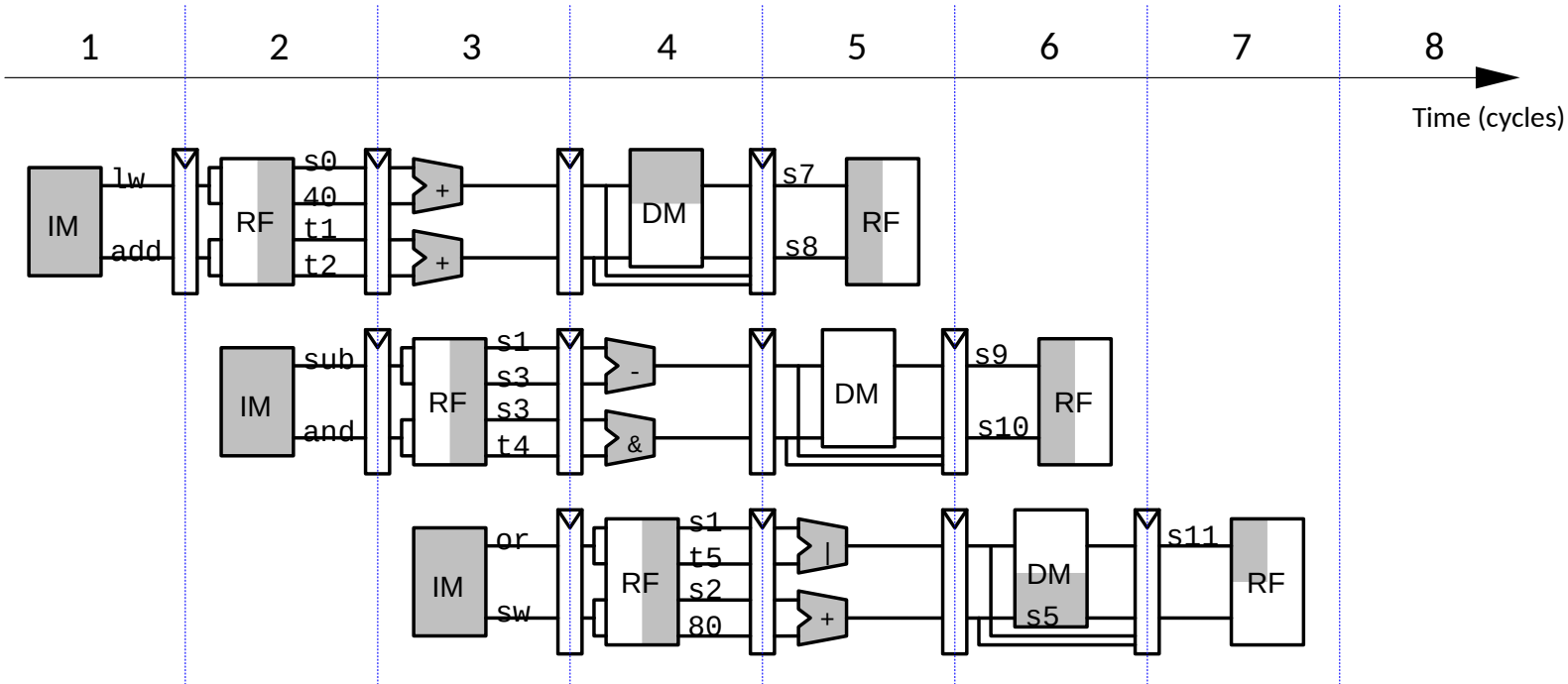
Ideal IPC: 2

Actual IPC: 2

```
lw  s7, 40(s0)
add s8, t1, t2

sub s9, s1, s3
and s10, s3, t4

or  s11, s1, t5
sw  s5, 80(s2)
```



Superscalar with Dependencies

Ideal IPC: 2

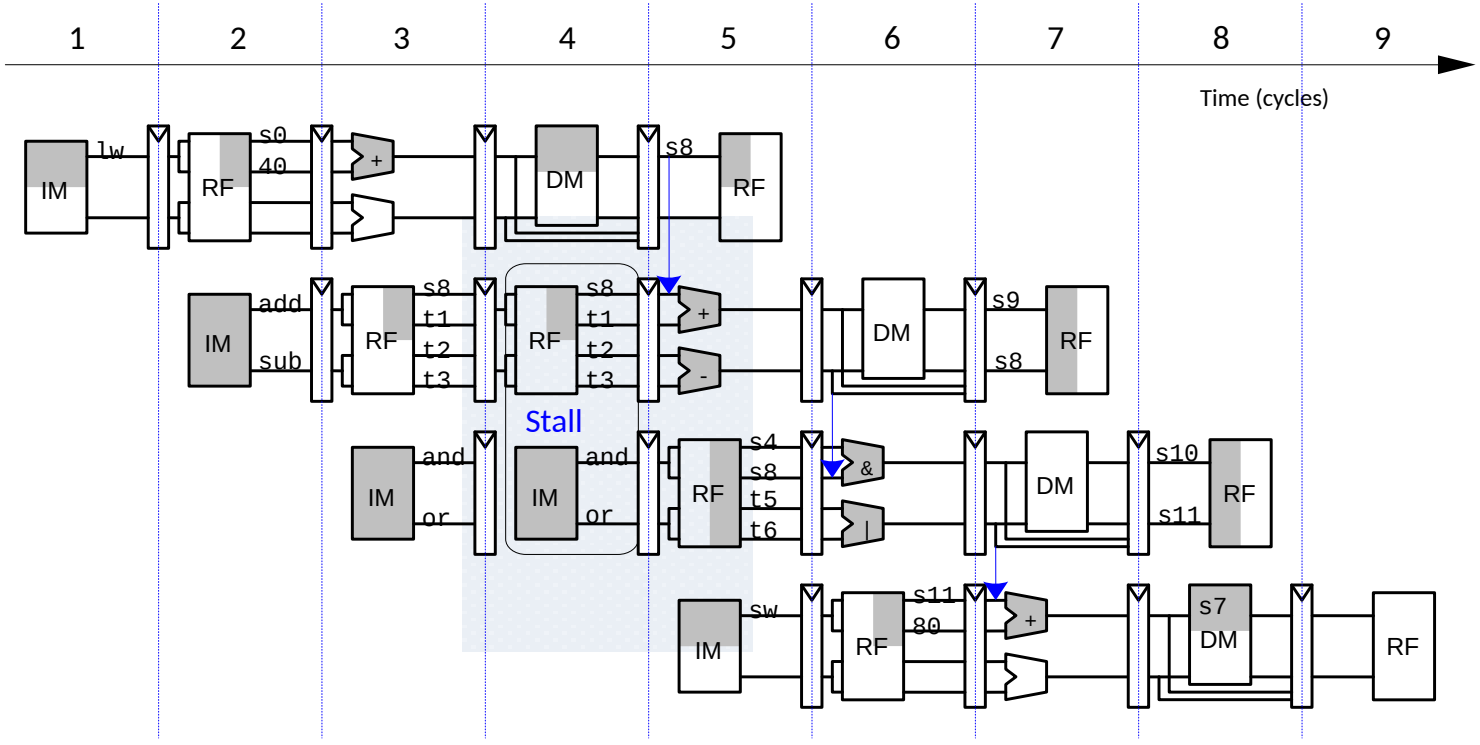
Actual IPC: 6/5 = 1.2

```
lw  s8, 40(s0)

add s9, s8, t1
sub s8, t2, t3

and s10, s4, s8
or  s11, t5, t6

sw  s7, 80(s11)
```



Out of Order (OOO) Processor

- Looks ahead across multiple instructions
- Issues as many instructions as possible at once
- Issues instructions out of order (as long as no dependencies)
- **Dependencies:**
  - **RAW** (read after write): one instruction writes, later instruction reads a register
  - **WAR** (write after read): one instruction reads, later instruction writes a register
  - **WAW** (write after write): one instruction writes, later instruction writes a register

Out of Order (OOO) Processor

- **Instruction level parallelism (ILP):** number of instruction that can be issued simultaneously (average < 3)
- **Scoreboard:** table that keeps track of:
  - Instructions waiting to issue
  - Available functional units
  - Dependencies

Out of Order Processor Example

Ideal IPC: 2

Actual IPC: 6/4 = 1.5

```
lw  s8, 40(s0)
or  s11, t5, t6
sw  s7, 80(s11)
add s9, s8, t1
sub s8, t2, t3
and s10, s4, s8
```

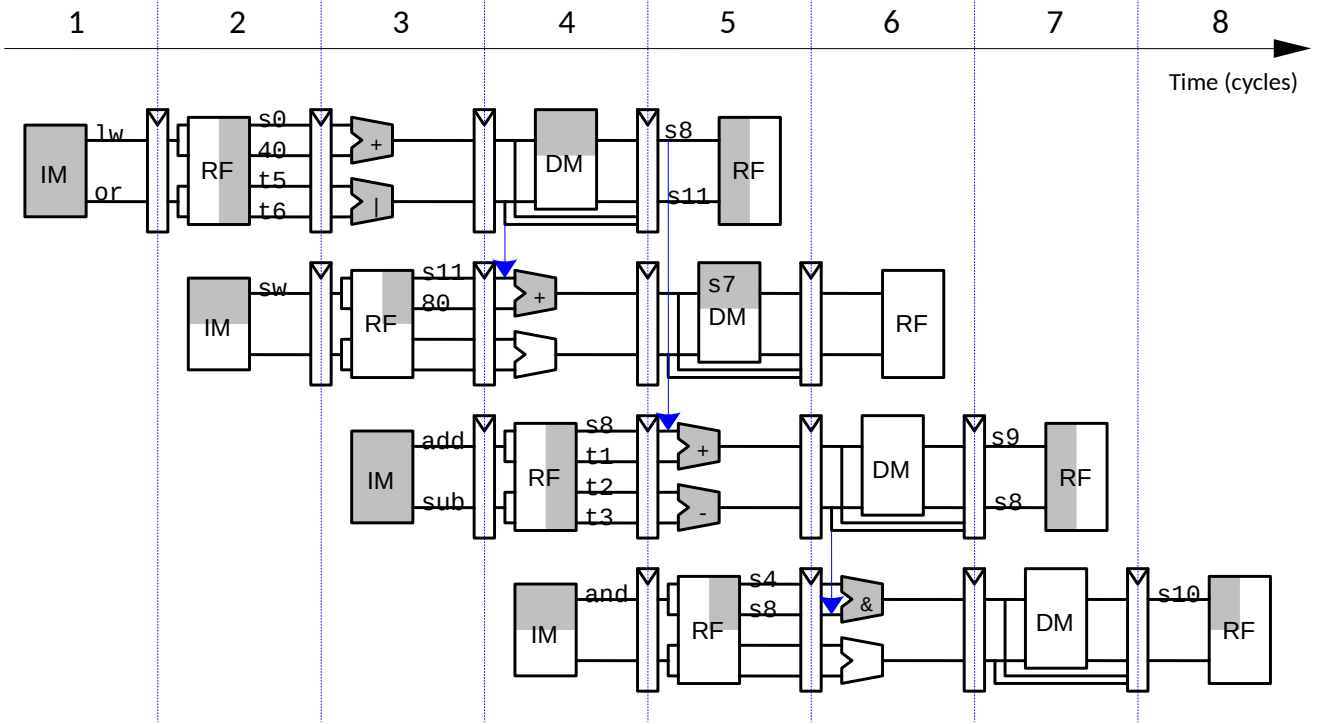
RAW

RAW

WAR

RAW

two cycle latency between load and use of **s8**



Register Renaming

Ideal IPC: 2

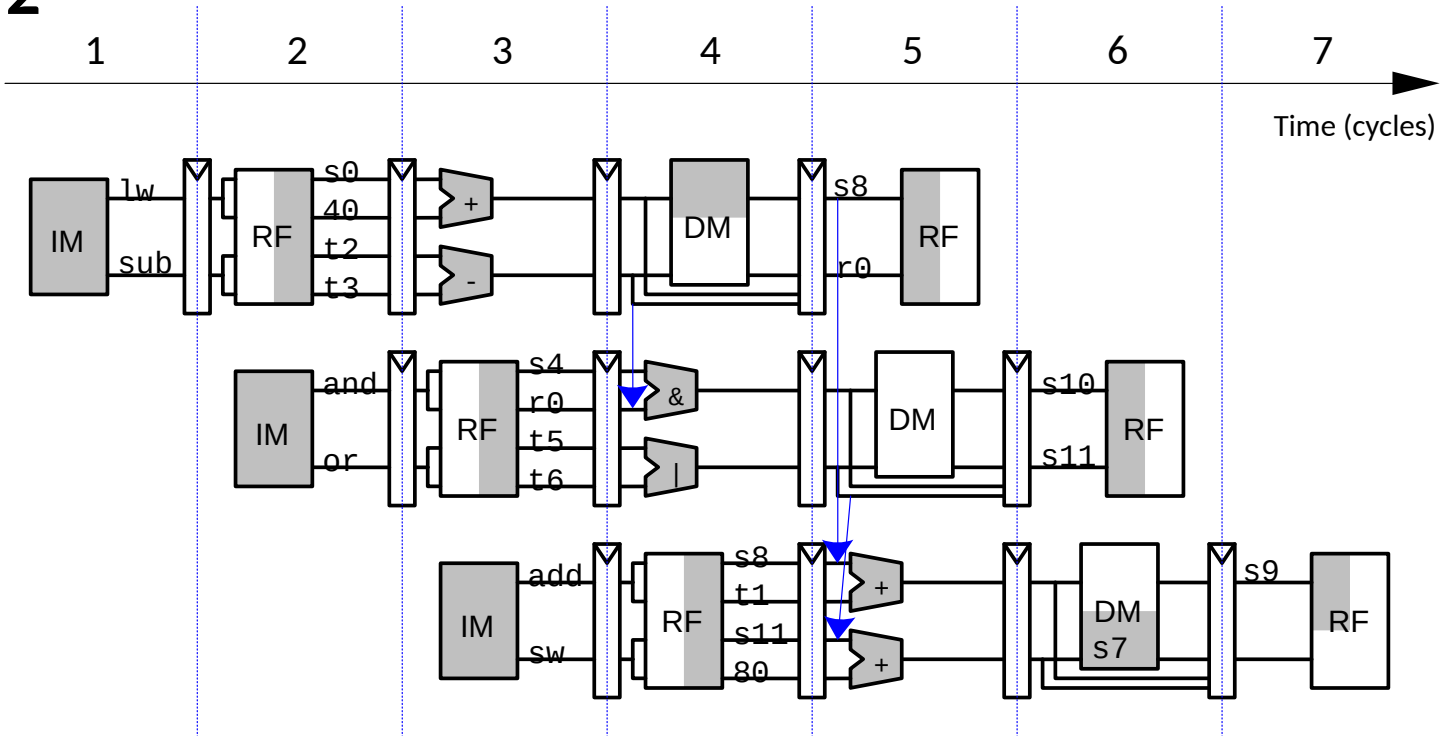
Actual IPC: 6/3 = 2

```
lw   s8, 40(s0)
sub  r0, t2, t3
and  s10, s4, r0
or   s11, t5, t6
add  s9, s8, t1
sw   s7, 80(s11)
```

2-cycle RAW

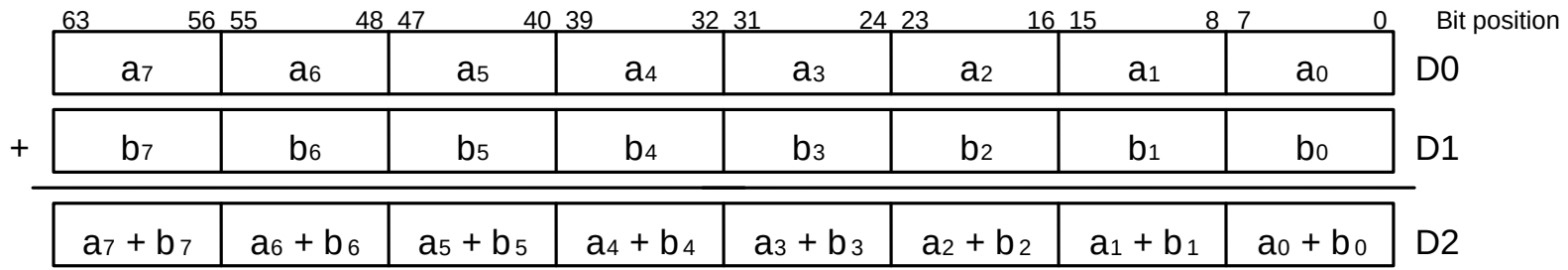
RAW

RAW



SIMD

- **Single Instruction Multiple Data** (SIMD)
  - Single instruction **acts on multiple pieces of data at once**
  - Common application: **graphics**
  - Can apply to short arithmetic operations (also called *packed arithmetic*)
- For example, add eight 8-bit elements



Chapter 7: Microarchitecture

Multithreading & Multiprocessors

Advanced Architecture Techniques

- **Multithreading**
  - Wordprocessor: thread for typing, spell checking, printing
- **Multiprocessors**
  - Multiple processors (cores) on a single chip

Threading: Definitions

- **Process:** program running on a computer
  - Multiple processes can run at once: e.g., surfing Web, playing music, writing a paper
- **Thread:** part of a program
  - Each process has multiple threads: e.g., a word processor may have threads for typing, spell checking, printing

Threads in a Conventional Processor

Single-core system:

- One thread runs at once
- When one thread stalls (for example, waiting for memory):
  - Architectural state of that thread stored
  - Architectural state of waiting thread loaded into processor and it runs
  - Called **context switching**
- Appears to user like all threads running simultaneously

Multithreading

- Multiple copies of architectural state
- Multiple threads **active** at once:
  - When one thread stalls, another runs immediately
  - If one thread can't keep all execution units busy, another thread can use them
- Does not increase instruction-level parallelism (ILP) of single thread, but increases throughput

Intel calls this "hyperthreading"

Multiprocessors

- Multiple processors (cores) with a method of communication between them
- Types:
  - **Homogeneous:** multiple cores with shared main memory
  - **Heterogeneous:** separate cores for different tasks (for example, DSP and CPU in cell phone)
  - **Clusters:** each core has own memory system

About these Notes

Digital Design and Computer Architecture Lecture Notes