361-1-4201/381-1-0107
Computer Architecture
Intro to Microarchitecture: Single-Cycle
Dr. Guy Tel-Zur

Based on slides by Prof. Onur Mutlu

Carnegie Mellon University

Spring 2015, 1/26/2015

With Dr. Danny Seidner modifications

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 – Lec #1
- Fundamental Concepts – Lec #1
  - Computer Arch = ISA (the definition) + uArch (the implementation)
- ISA basics and tradeoffs – Lec #2
  - Instruction length
  - Uniform vs. non-uniform decode
  - Number of registers
  - Addressing modes
  - Aligned vs. unaligned access
  - RISC vs. CISC properties
- MIPS ISA – Lec #3
  - MIPS ISA Overview
- Datapath & Microarchitecture – Lec #4

חומר למחשבה

- As you learn the MIPS 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?"

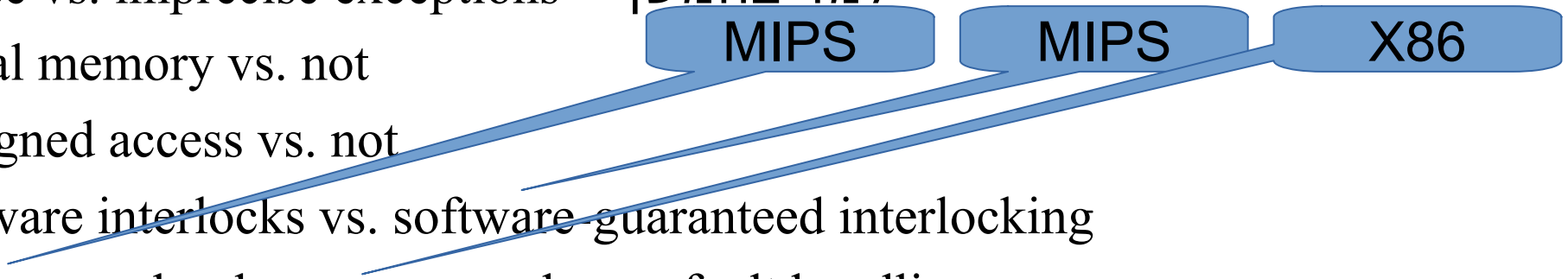
    - The 1st question to ask what is the Design Point / Use Case

      מה השאלה הראשונה שתרצו לשאול את המזמין?

Look Forward & Up

Review: Other Example ISA-level Tradeoffs

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



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

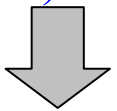
- How do we implement it?
- i.e., how do we design a system that obeys the hardware/software interface?
- Aside: "System" can be solely hardware or a combination of hardware and software
  - Remember "Translation of ISAs"
  - A virtual ISA can be converted by "software" into an implementation ISA
- We will assume "hardware" for most lectures

Implementing the ISA: Microarchitecture Basics

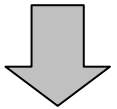
How Does a Machine Process Instructions?

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

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



Process instruction

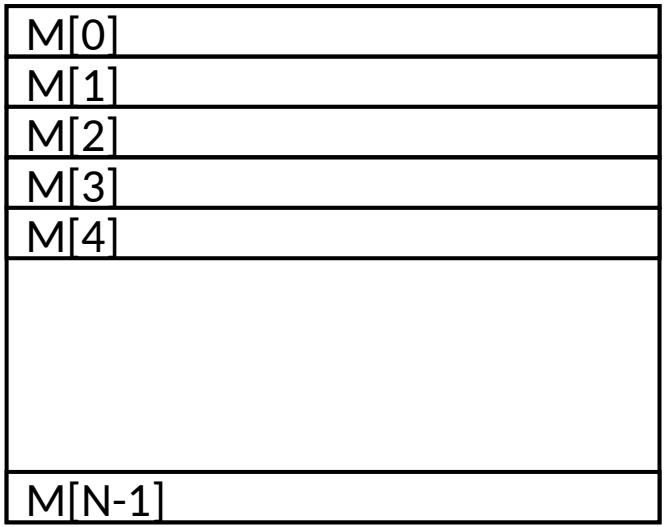


AS' = Architectural (programmer visible) state after an instruction is processed

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

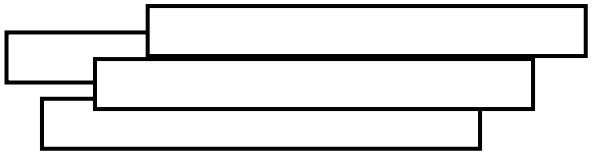
Remember: Programmer Visible (Architectural) State

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

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

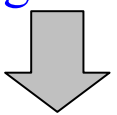
- Microarchitecture implements how AS is transformed to AS'
  - There are many choices in implementation
  - We can have programmer-invisible state to optimize the speed of instruction execution: multiple state transitions per instruction
    - Choice 1: AS → AS' (transform AS to AS' in a single clock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

AS = Architecture State
MS=Mictoarchitecture State

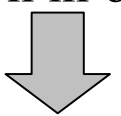
A Very Basic Instruction Processing Engine

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

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



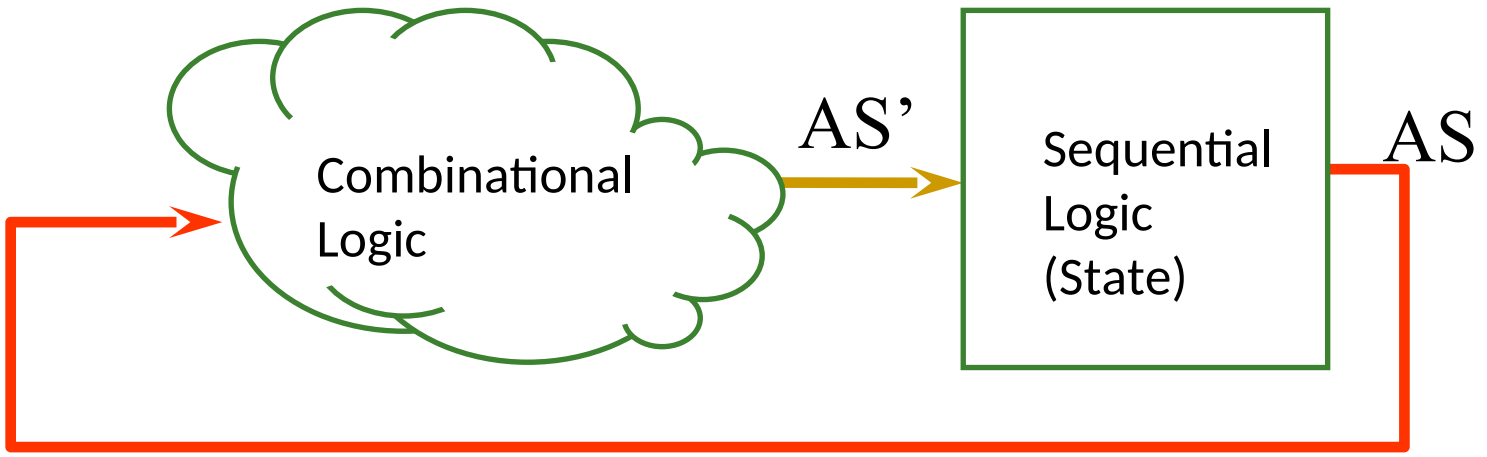
Process instruction in one clock cycle



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

A Very Basic Instruction Processing Engine

- Single-cycle machine



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

מישהו רוצה לענות?

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

Slowest combinational logic

- 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 מחזור של הוראה לבין 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 חוויה מתסכלת...

Instruction Processing Viewed Another Way

- 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

Data vs. Control separation

- 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

Flash-Forward: 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

We have one degree of freedom to optimize

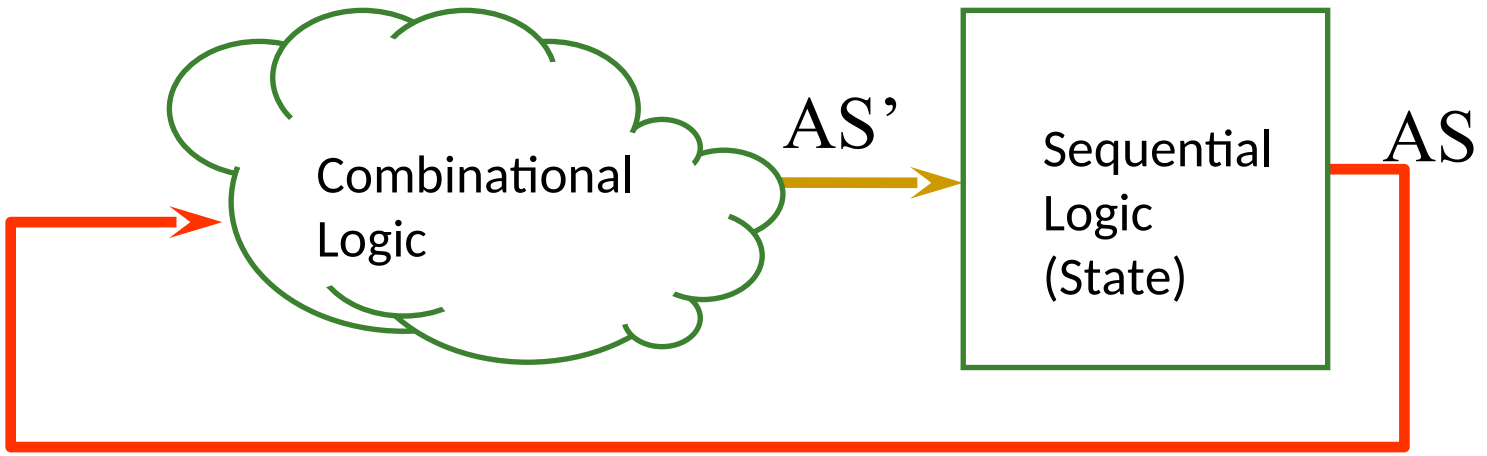
Now, we have two degrees of freedom to optimize independently

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

A Single-Cycle Microarchitecture
A Closer Look

Remember…

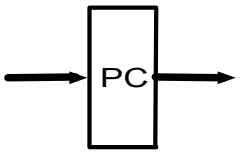
- Single-cycle machine

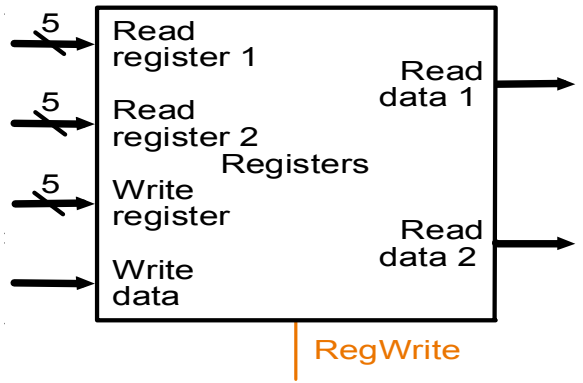


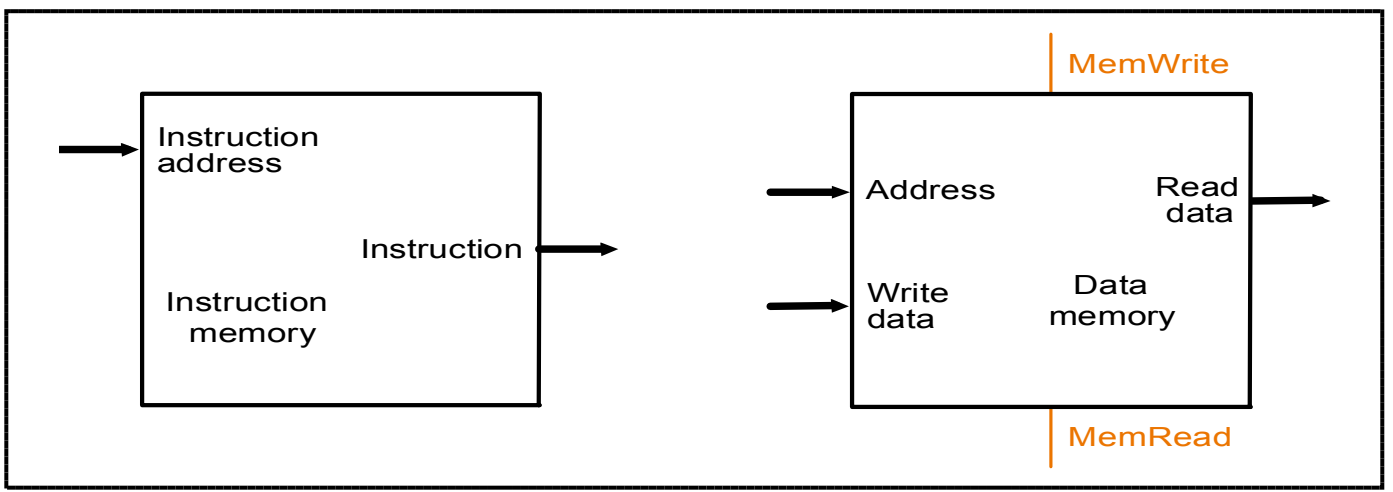
Let's Start with the State Elements

- Data and control inputs

נתחיל עם כל מה שהוא
programmer visible state







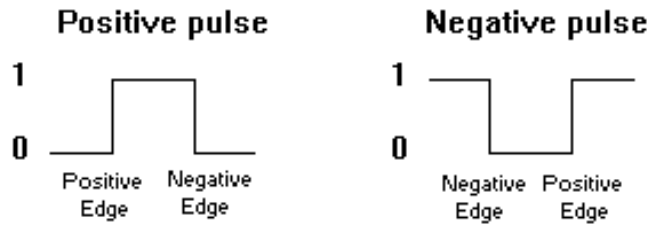
**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

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

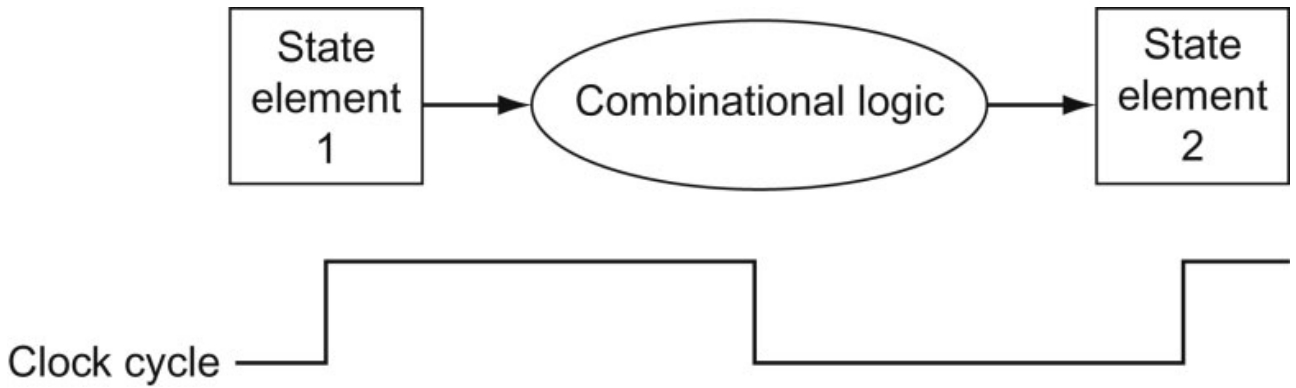


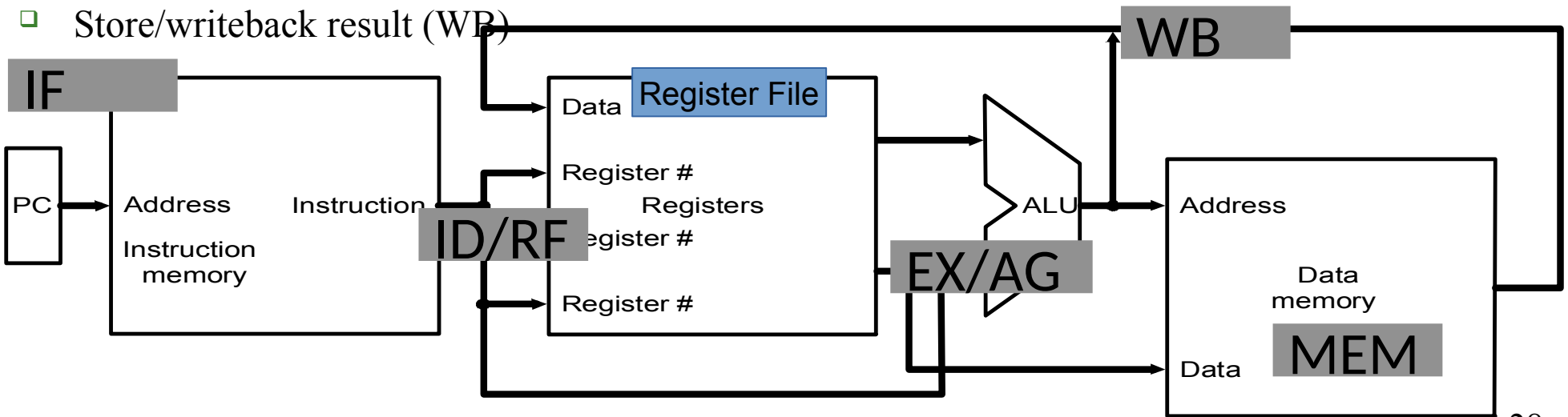
FIGURE 4.3 Combinational logic, state elements, and the clock are closely related. In a synchronous digital system, the clock determines when elements with state will write values into internal storage. Any inputs to a state element must reach a stable value (that is, have reached a value from which 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

- 5 generic steps (P&H book) –

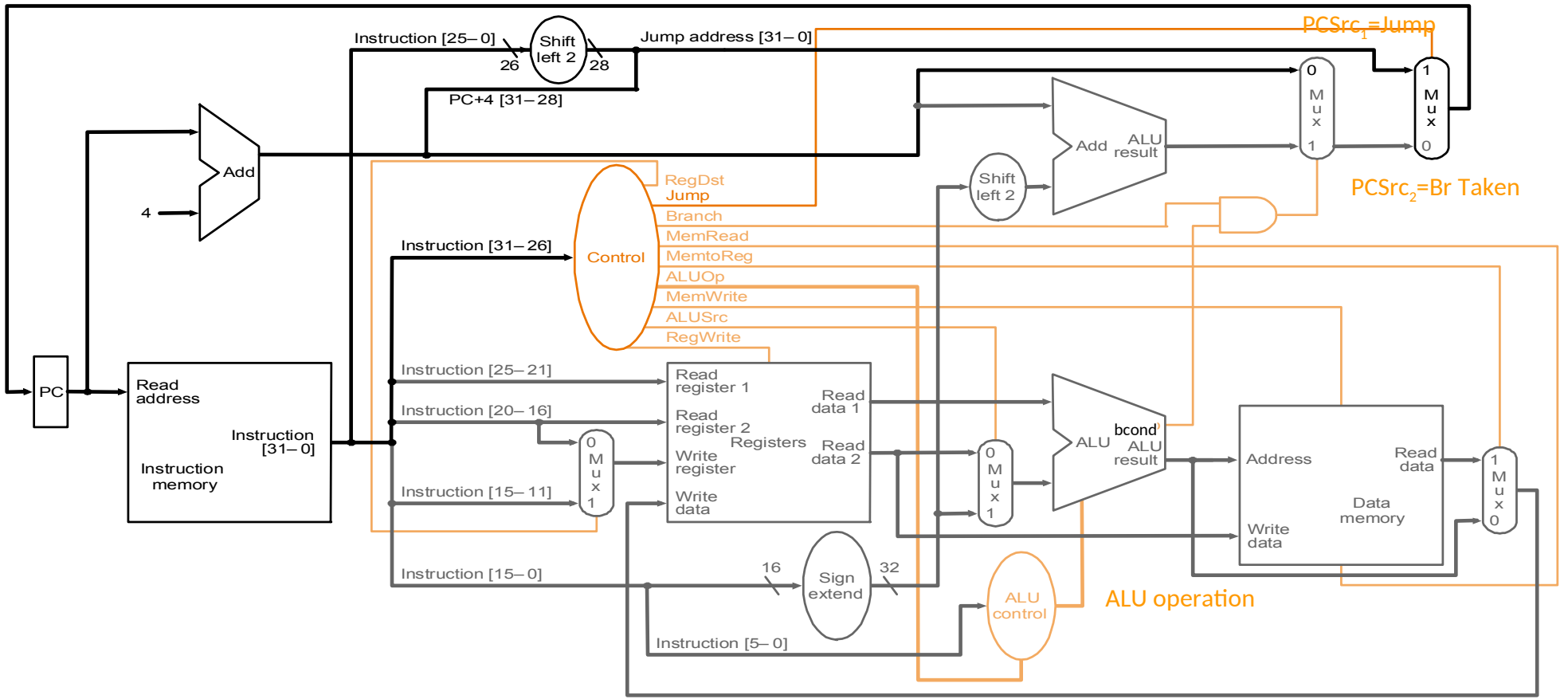
אמנם דיברנו על 6 שלבים אך כאן נעשה מיזוג ושינוי קל לעומת מה שנכתב לפני 10 שקפים

- 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



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

JAL, JR, JALR omitted

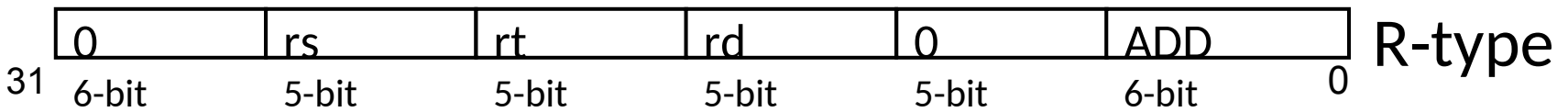
Single-Cycle Datapath for *Arithmetic and Logical Instructions*

R-Type ALU Instructions

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

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

- Machine encoding



- Semantics

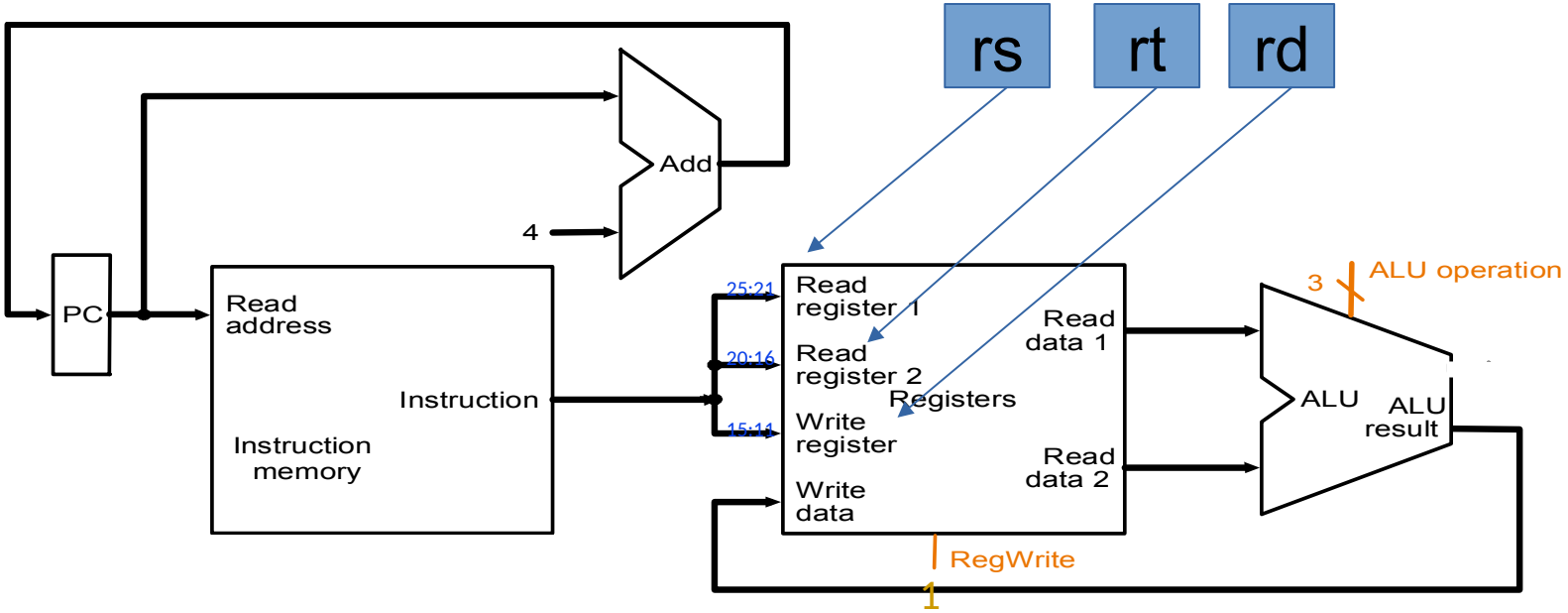
  if MEM[PC] == ADD rd rs rt

  GPR[rd] ← GPR[rs] + GPR[rt]

  PC ← PC + 4

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

ALU Datapath להלן המימוש



if MEM[PC] == ADD rd rs rt

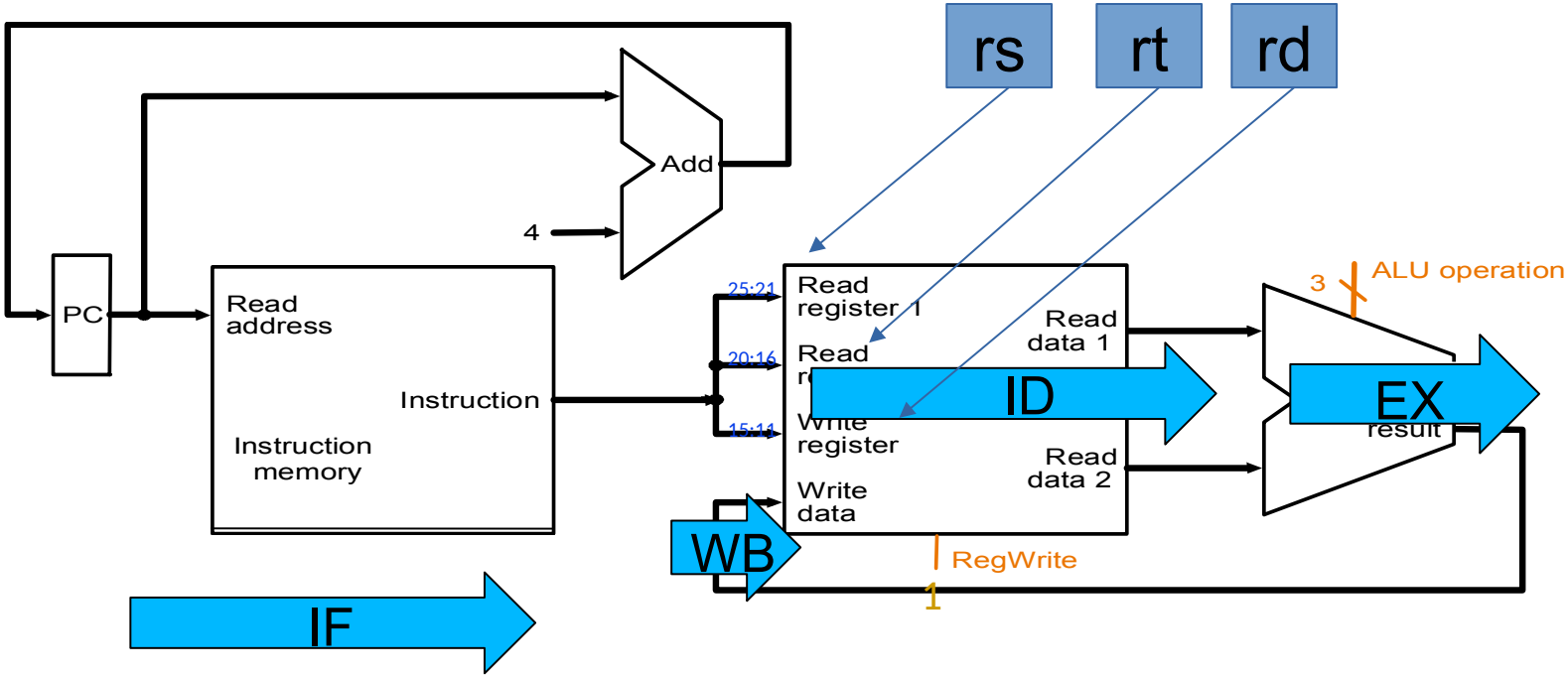
GPR[rd] ← GPR[rs] + GPR[rt]

PC ← PC + 4

IF	ID	EX	MEM	WB

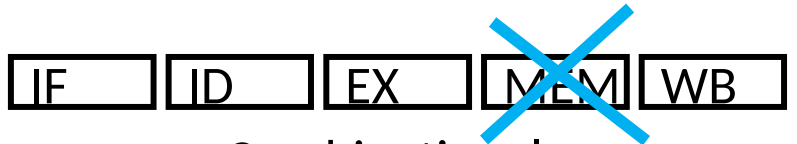
Combinational state update logic

R-Type ALU להלן המימוש



GPR[rd] ← GPR[rs] + GPR[rt]

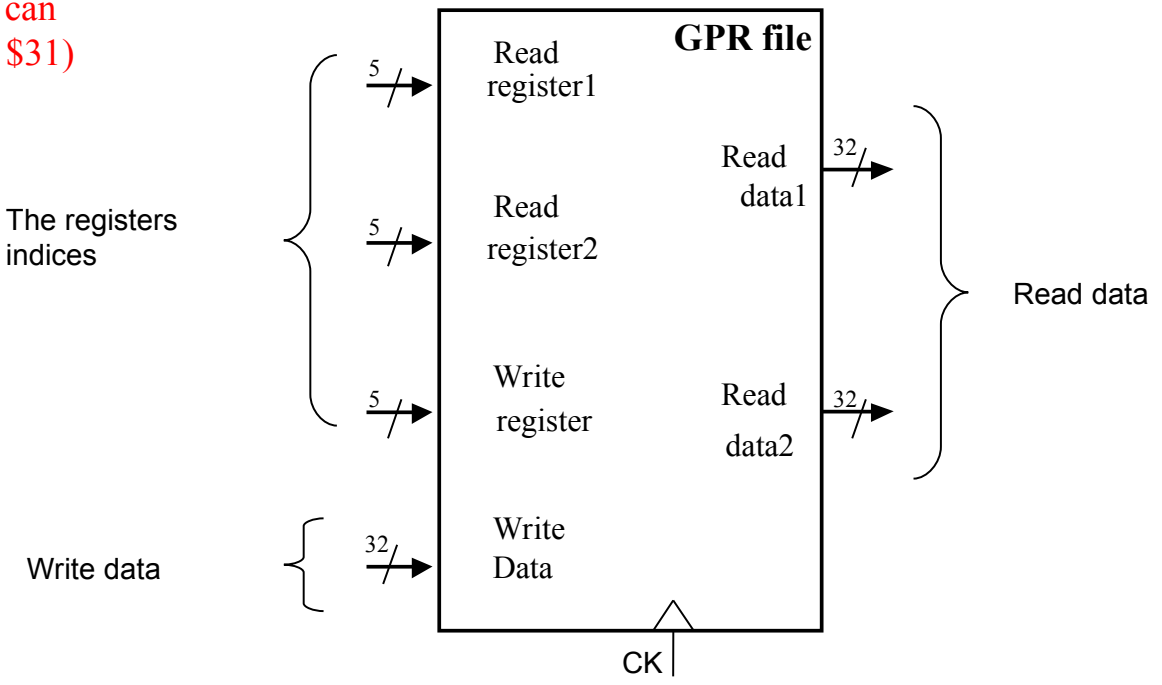
PC ← PC + 4



Combinational state update logic

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

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



Let's see how it is used:

The General-Purpose Register file (GPR).

There are 3 ports in this box

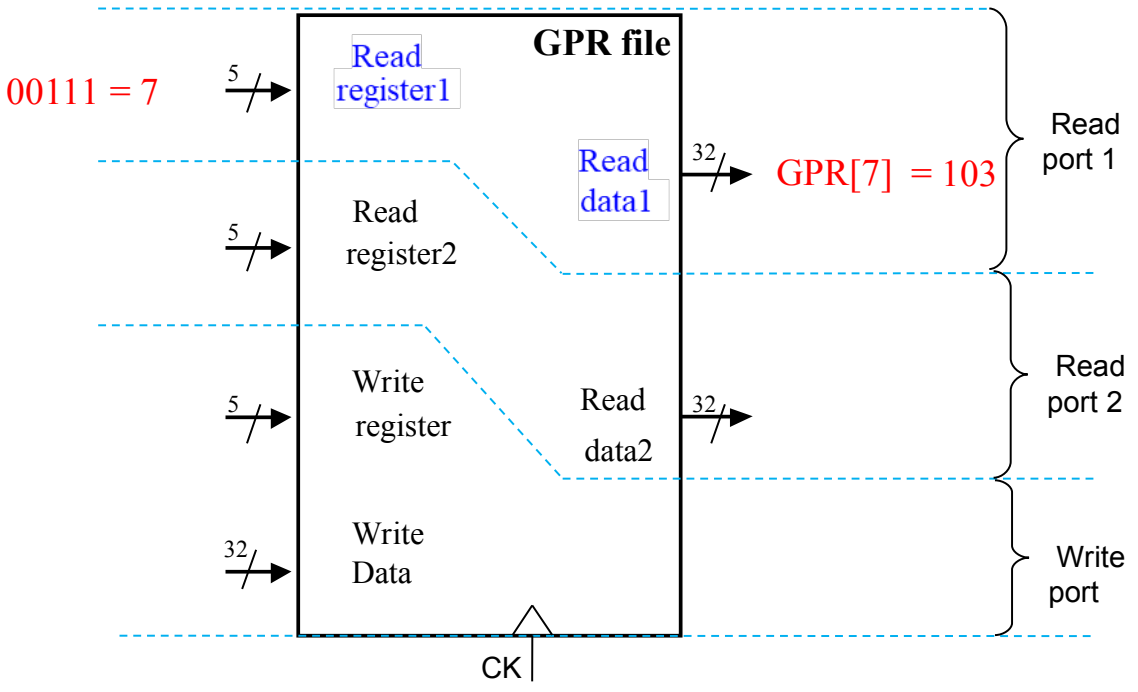
If we want to read from $7 via port 1

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).

So if we have the instruction:

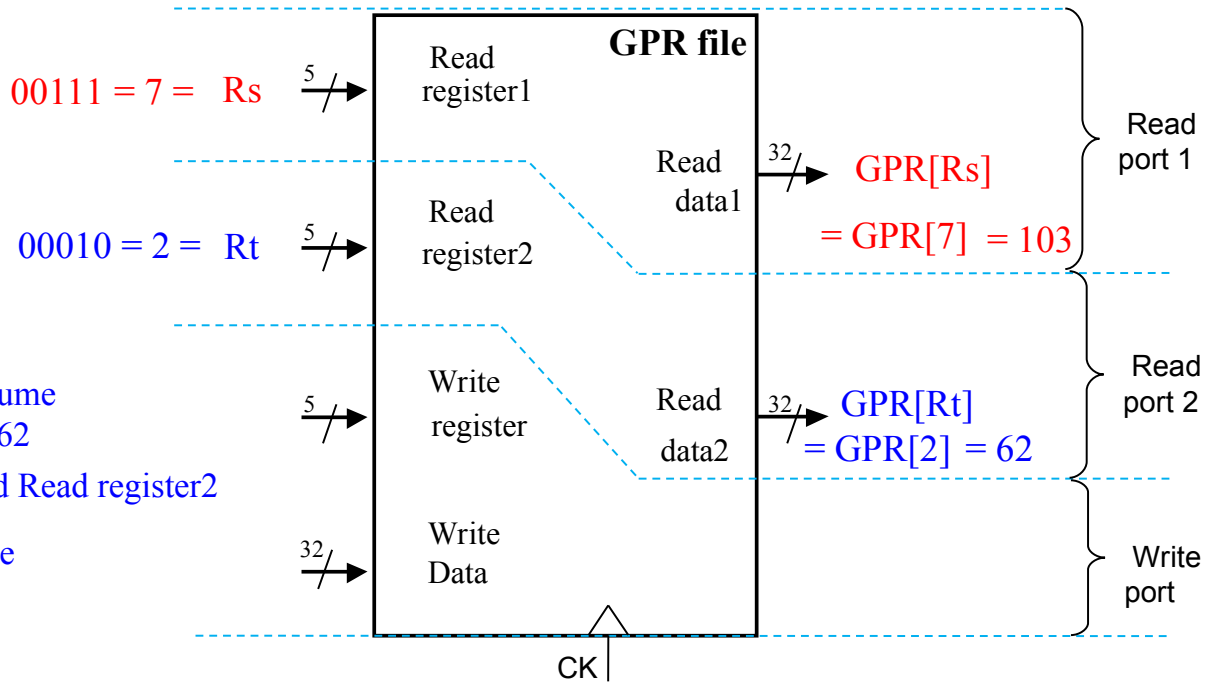
add $12, $7, $2

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).

R

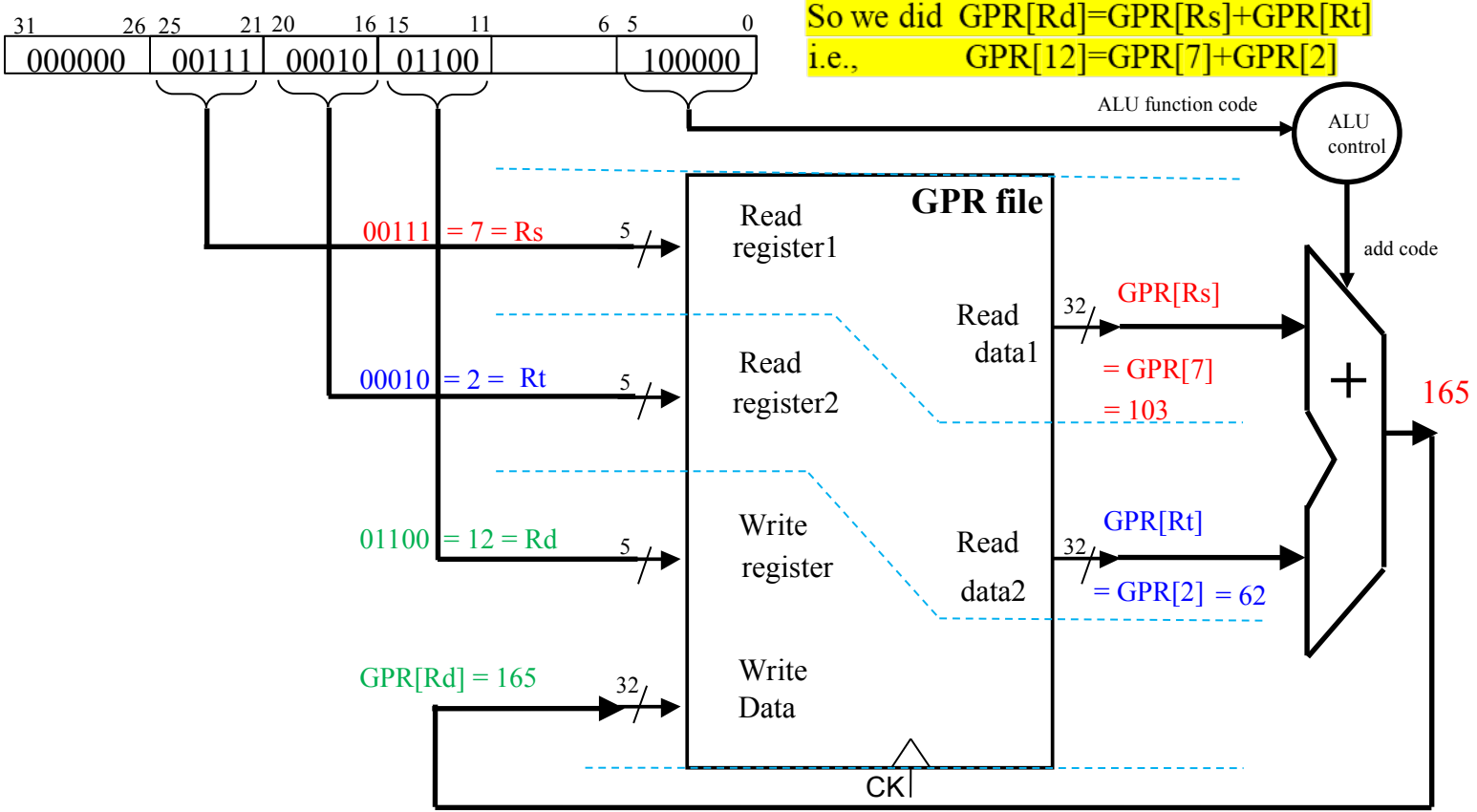
opcode	rs	rt	rd	shamt	funct
31 26	25 21	20 16	15 11	10 6	5 0

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

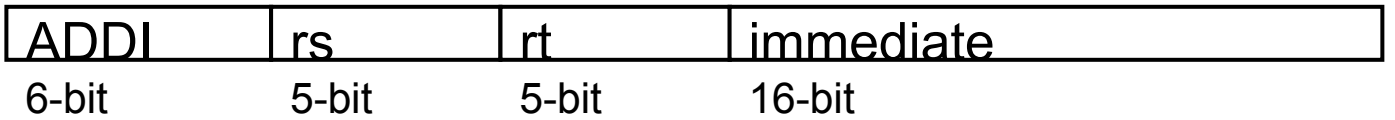
Now we want to write this to Rd

I-Type ALU Instructions

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

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

- Machine encoding



I-type

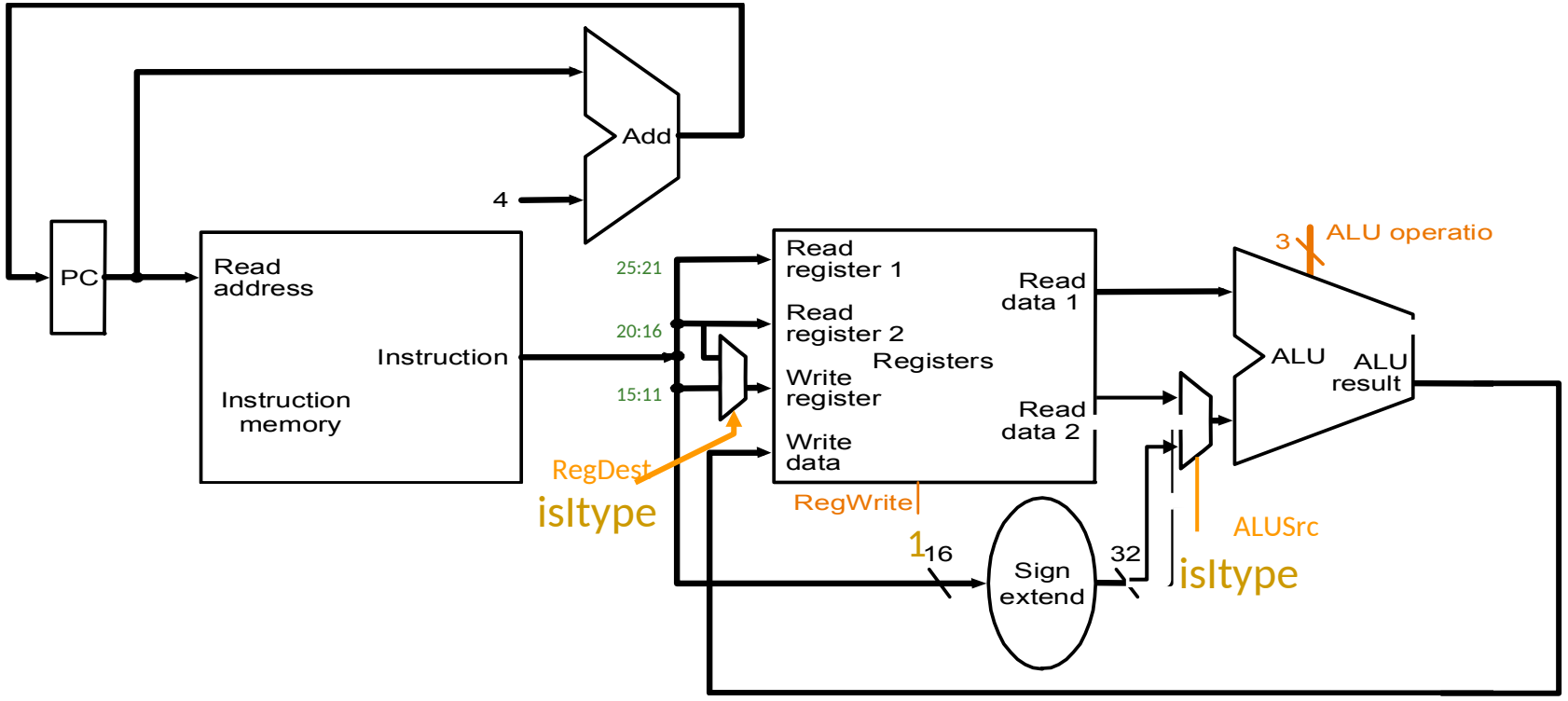
- Semantics

  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.



if MEM[PC] == ADDI rt rs immediate

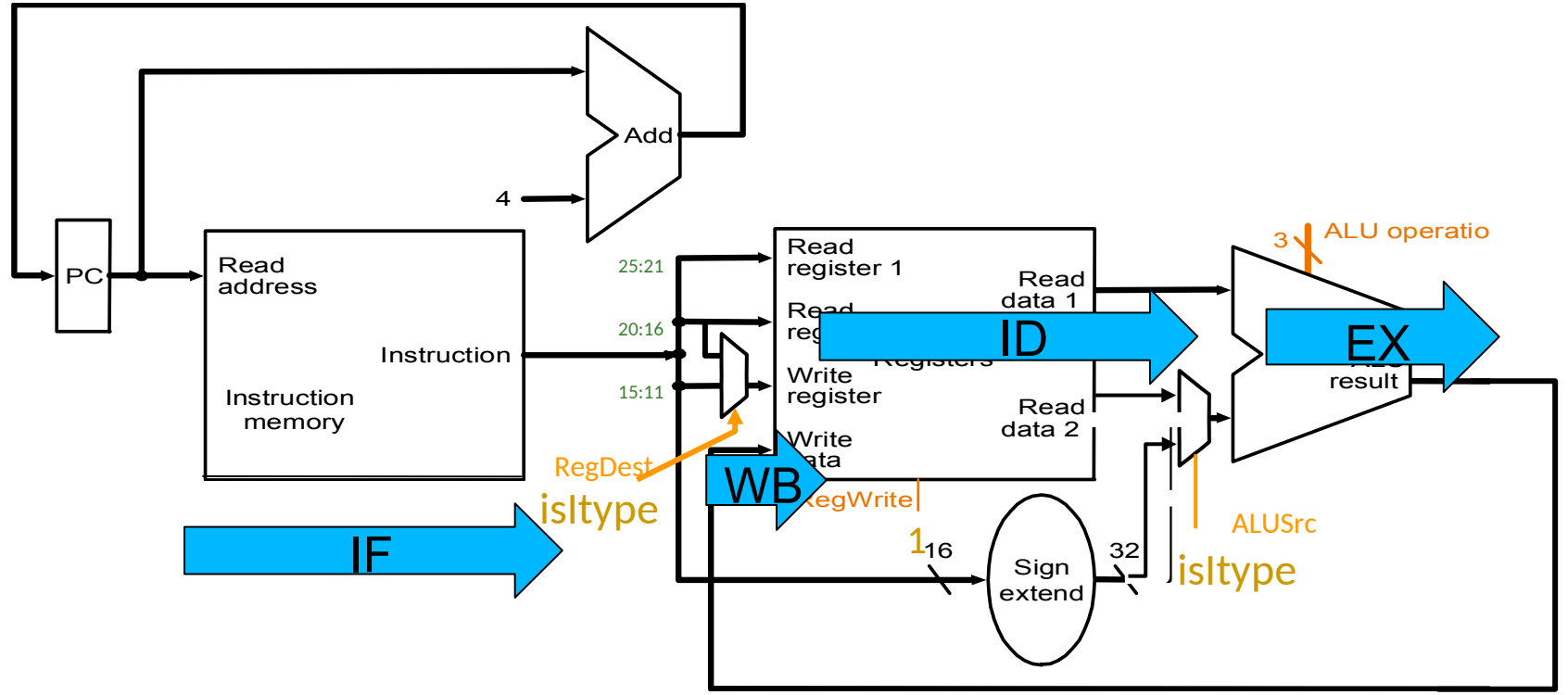
GPR[rt] ← GPR[rs] + sign-extend (immediate)

PC ← PC + 4

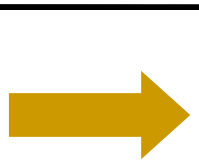
IF	ID	EX	MEM	WB

Combinational state update logic

Datapath for R and I-Type ALU Insts.



GPR[rt] ← GPR[rs] + sign-extend (immediate)
PC ← PC + 4



Combinational state update logic

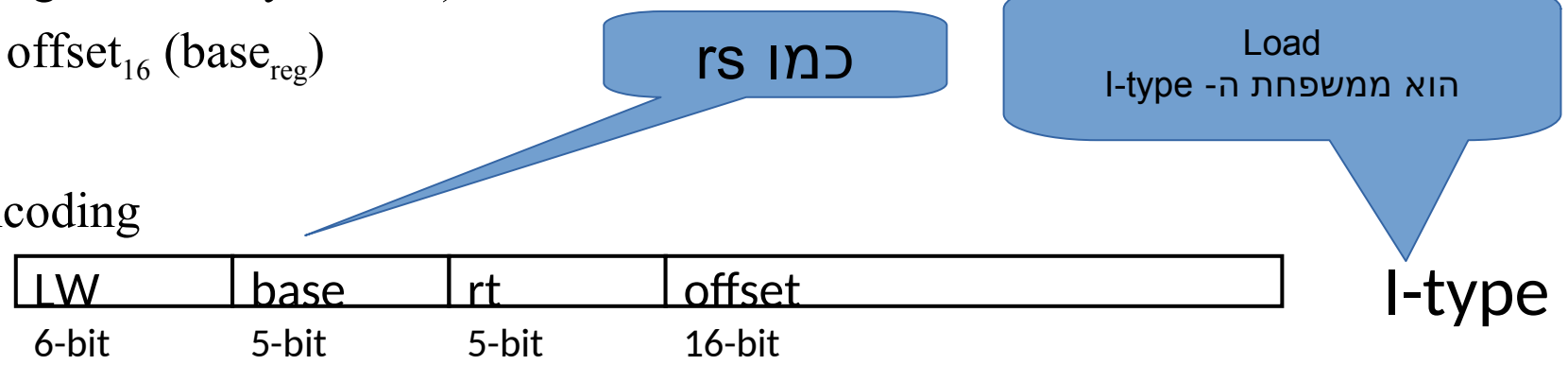
Single-Cycle Datapath for *Data Movement Instructions*

Load Instructions

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

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

- Machine encoding



- Semantics

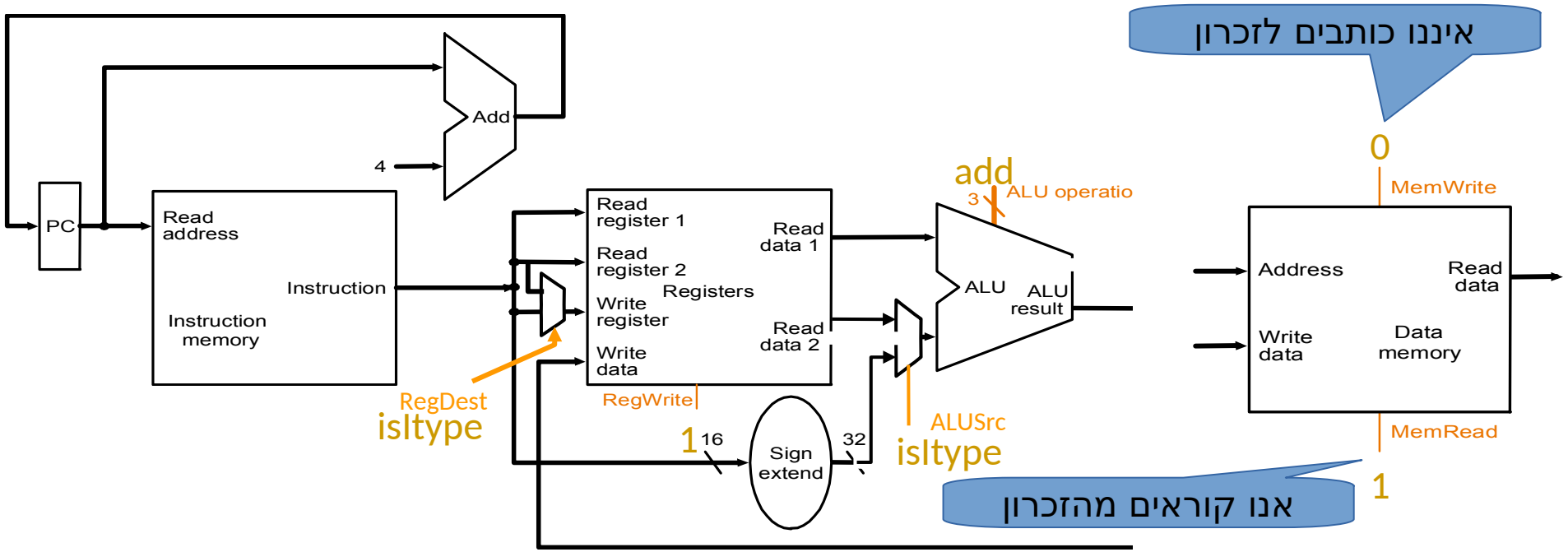
  if MEM[PC]==LW rt $offset_{16}$ (base)

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

  GPR[rt] ← MEM[translate(EA)]

  PC ← PC + 4

LW Datapath



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

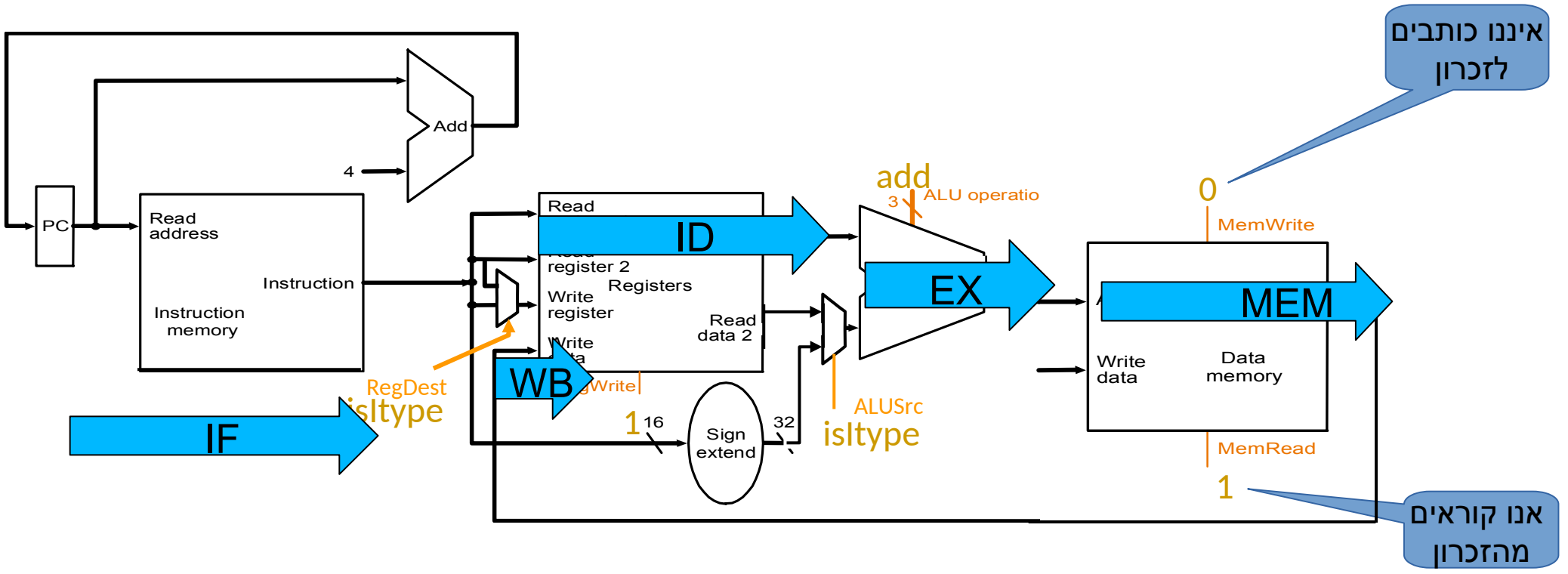
EA = sign-extend(offset) + GPR[base]

GPR[rt] ← MEM[translate(EA)]

IF	ID	EX	MEM	WB

Combinational state update logic

LW Datapath



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

GPR[rt] ← MEM[EA]

PC ← PC + 4

IF	ID	EX	MEM	WB

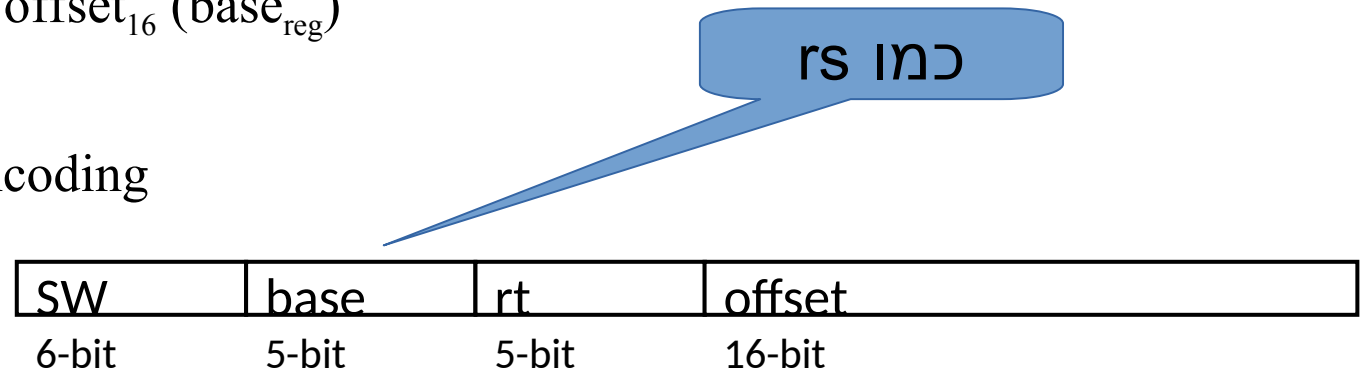
Combinational state update logic

Store Instructions

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

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

- Machine encoding



- Semantics

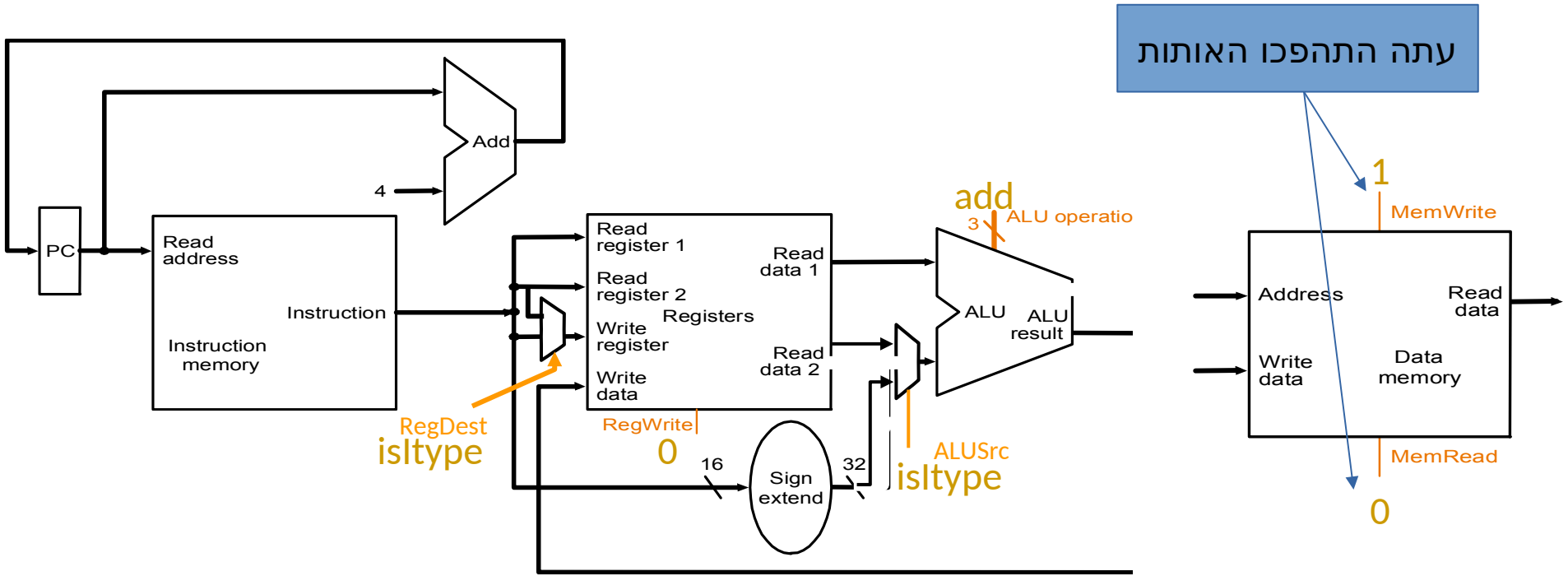
  if MEM[PC]==SW rt $offset_{16}$ (base)

  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]

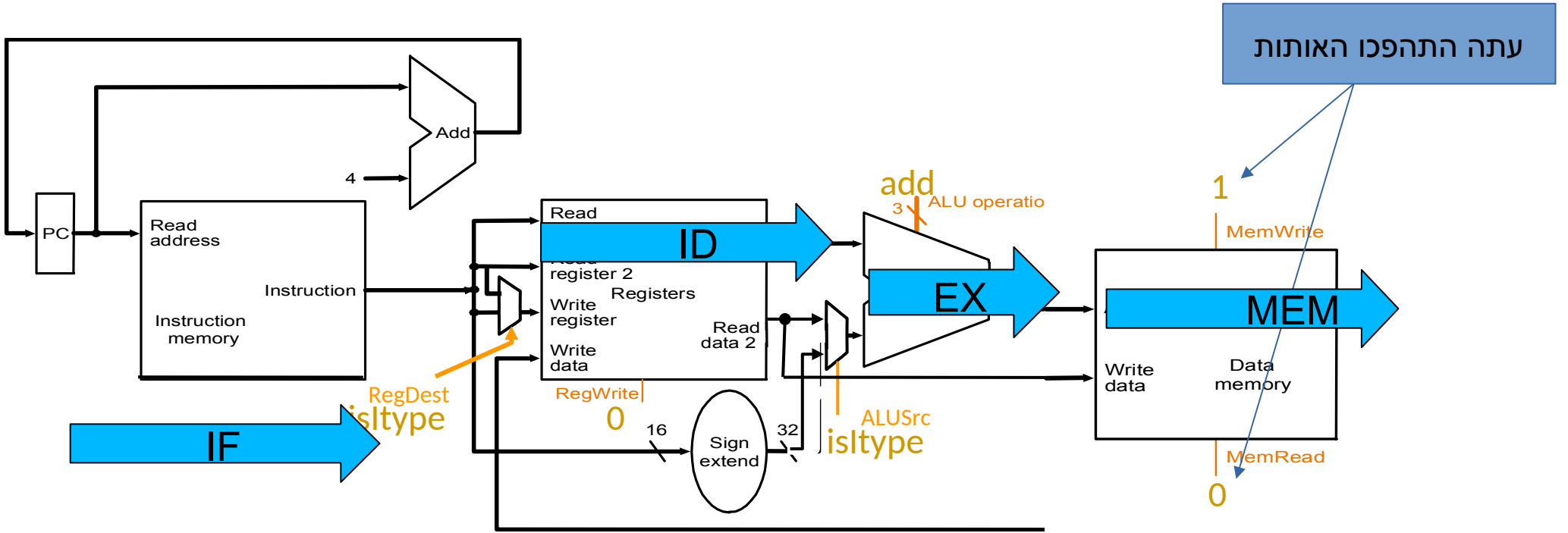
MEM[translate(EA)] ← GPR[rt]

PC ← PC + 4

IF	ID	EX	MEM	WB

Combinational state update logic

SW Datapath



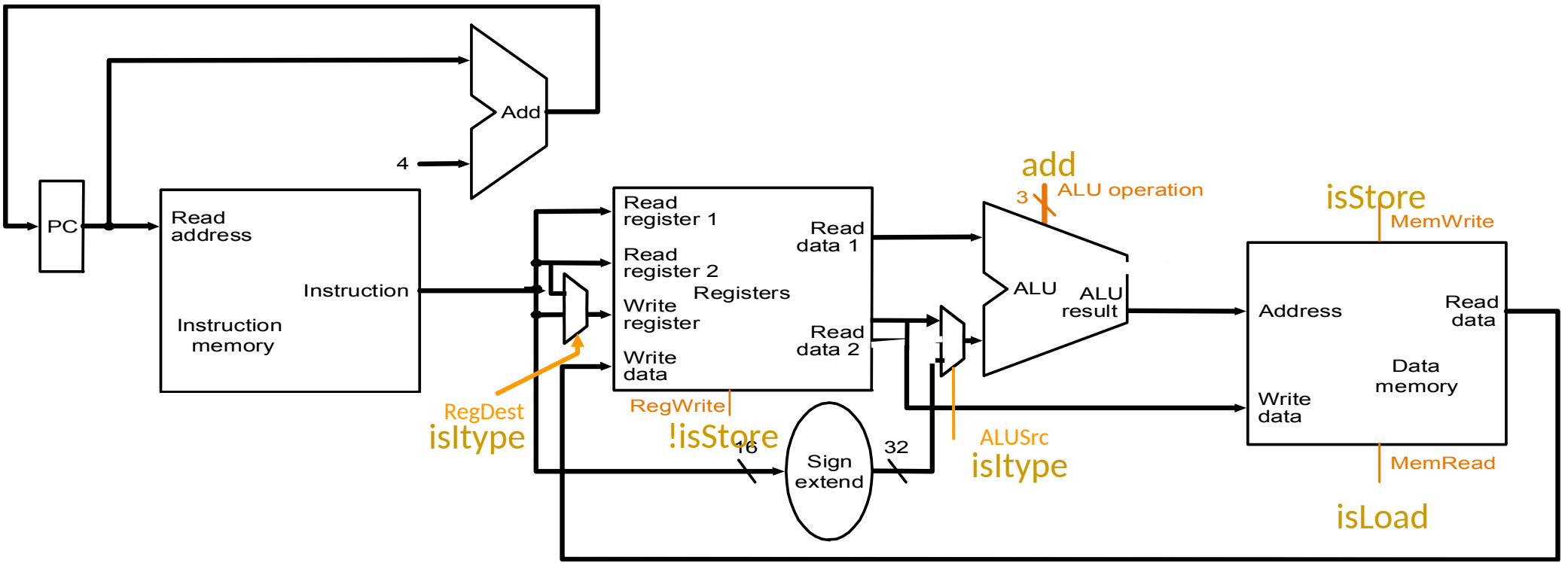
EA = GPR[rs] + sign-extend(offset)

MEM[EA] ← GPR[rt]

PC ← PC + 4

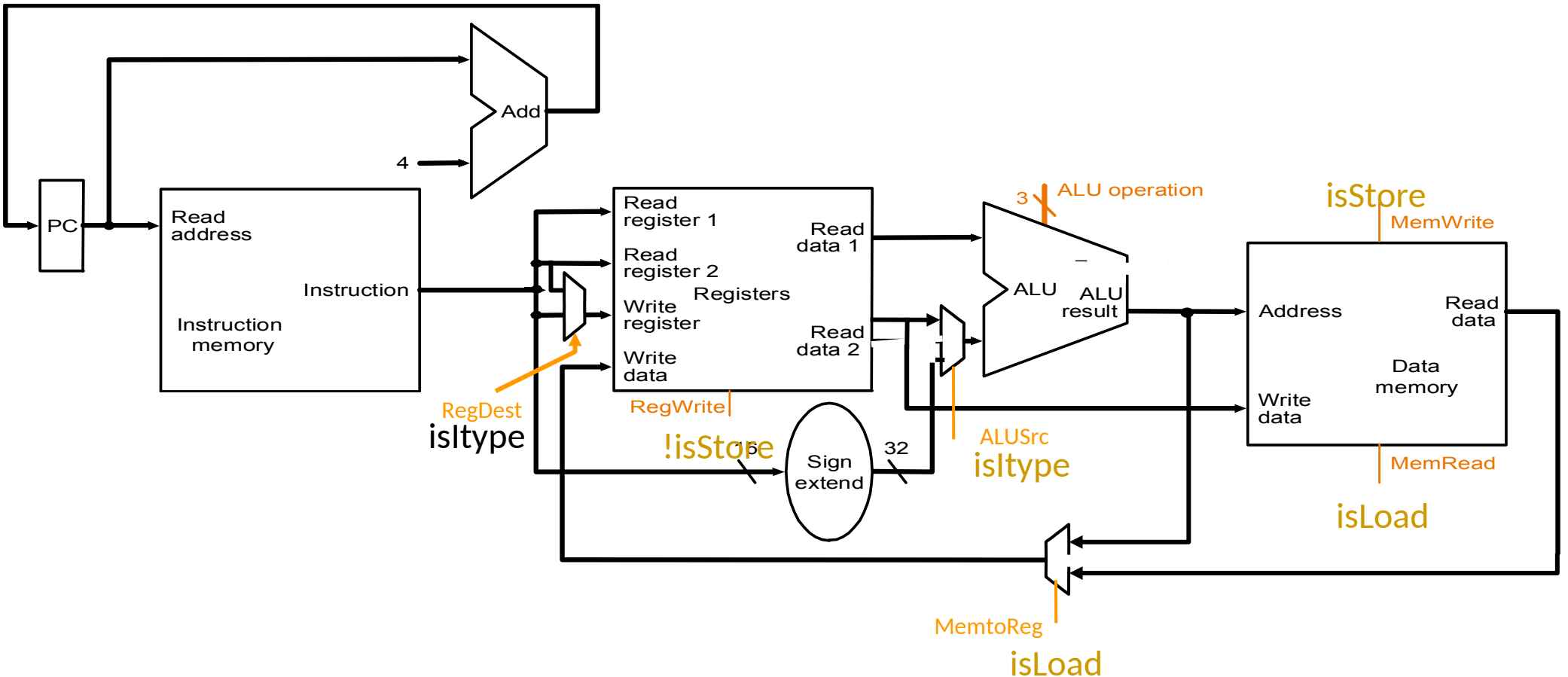
Combinational state update logic

Load-Store Datapath



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

Datapath for Non-Control-Flow Insts.



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

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

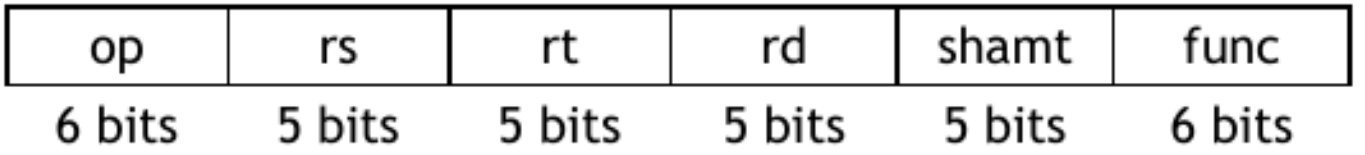
CSE378 University of Washington

https://courses.cs.washington.edu/courses/cse378/10sp/lectures.html

אנו לא 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.

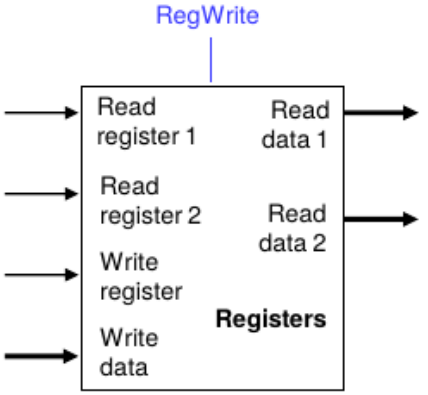


- An example instruction and its encoding:

add $s4, $t1, $t2

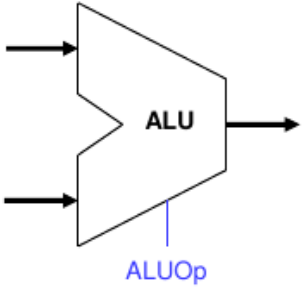
Registers and ALUs

- 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.

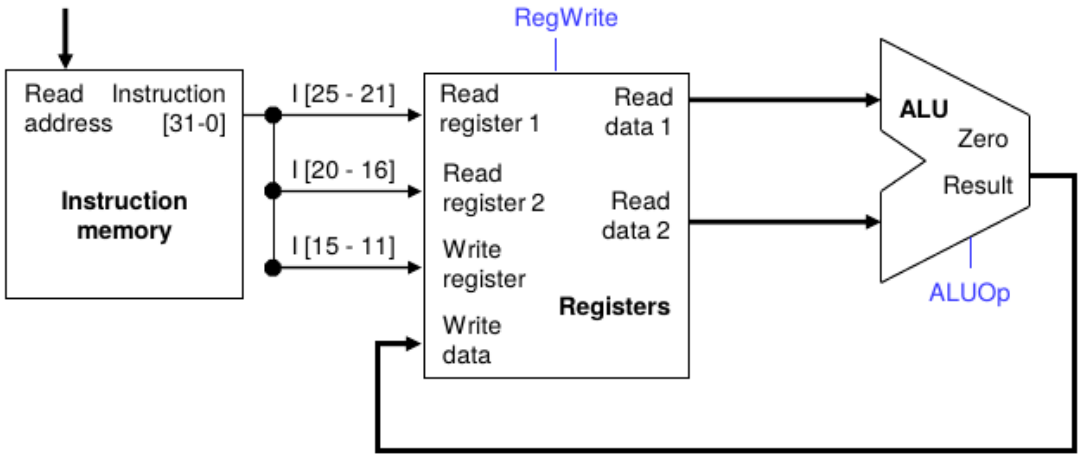
- Here's a simple ALU with five operations, selected by a 3-bit control signal ALUOp.

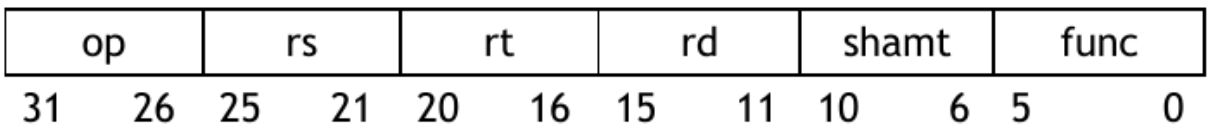


ALUOp	Function
000	and
001	or
010	add
110	subtract
111	slt

Executing an R-type instruction

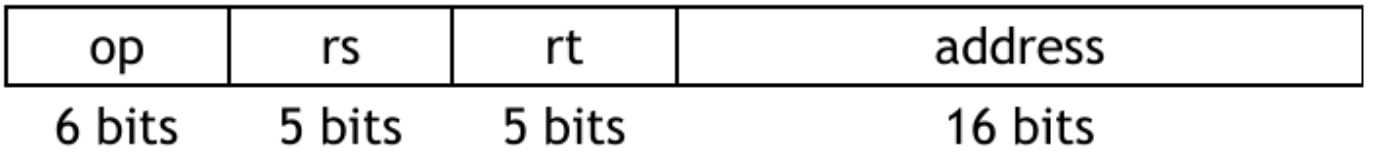
1. Read an instruction from the instruction memory.
2. The source registers, specified by instruction fields rs and rt, should be read from the register file.
3. The ALU performs the desired operation.
4. Its result is stored in the destination register, which is specified by field rd of the instruction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

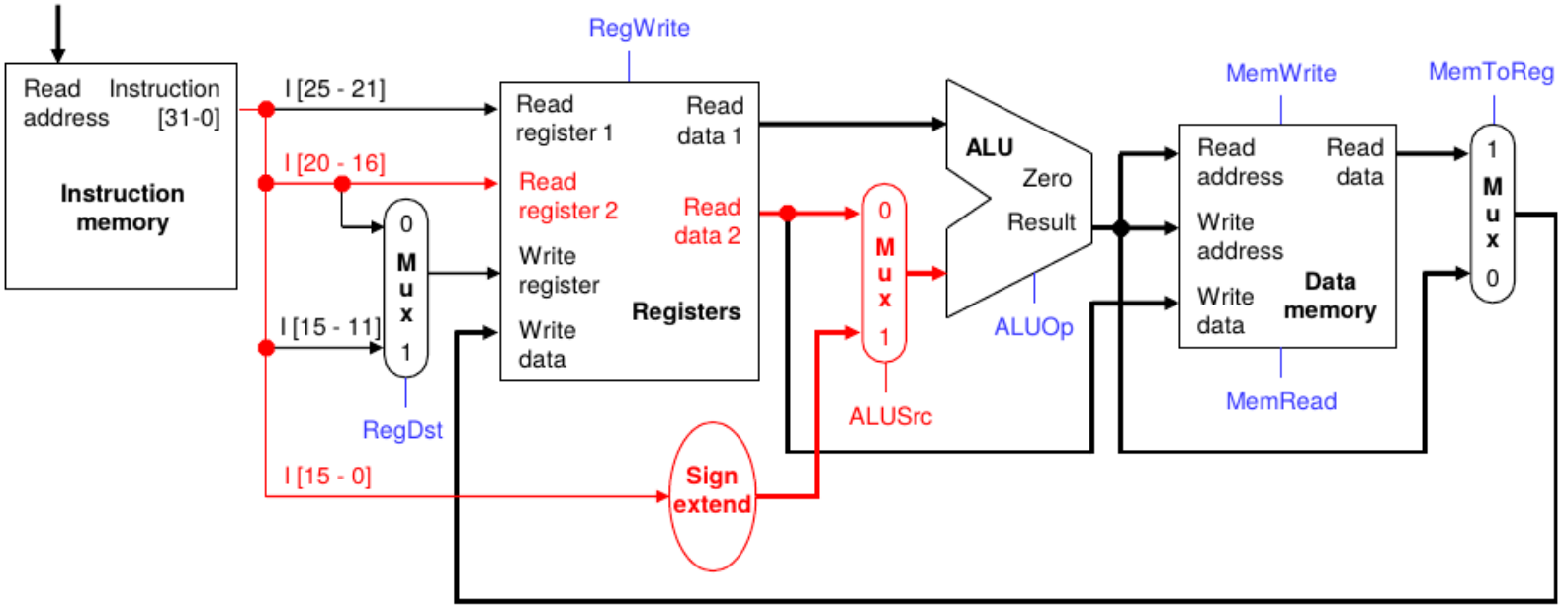


- Two example instructions:

lw $t0, -4($sp)	100011	11101	01000	1111 1111 1111 1100
sw $a0, 16($sp)	101011	11101	00100	0000 0000 0001 0000

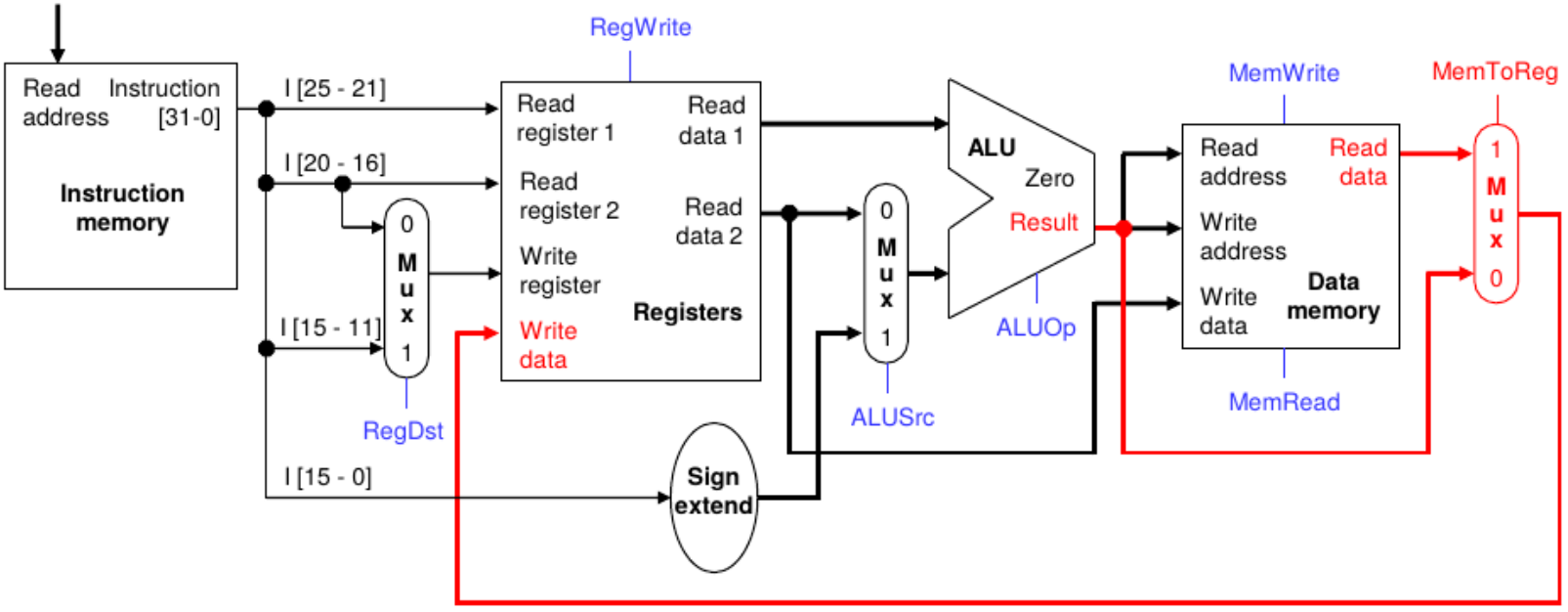
Accessing data memory

- 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.



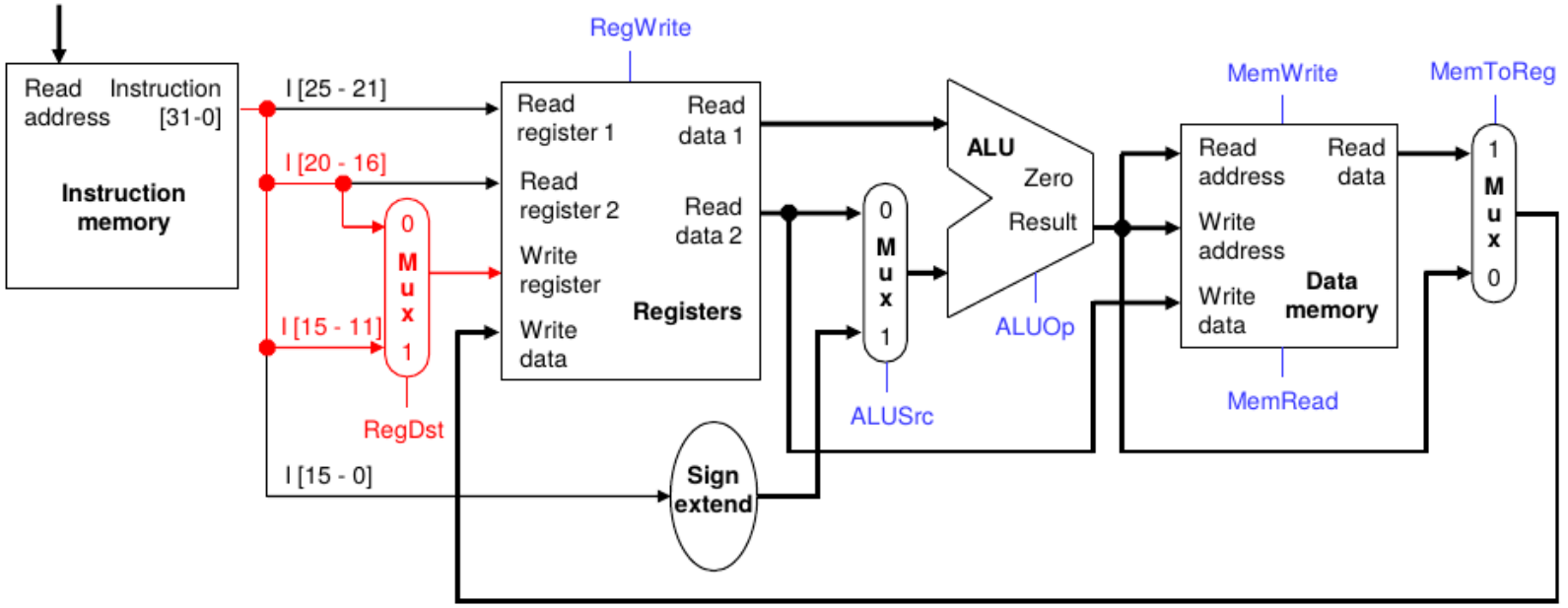
RegDst

- 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

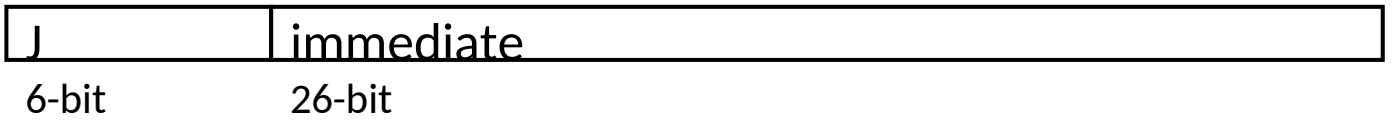
Single-Cycle Datapath for *Control Flow Instructions*

Unconditional Jump Instructions

- Assembly

  J immediate$_{26}$

- Machine encoding



J-type

- Semantics

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

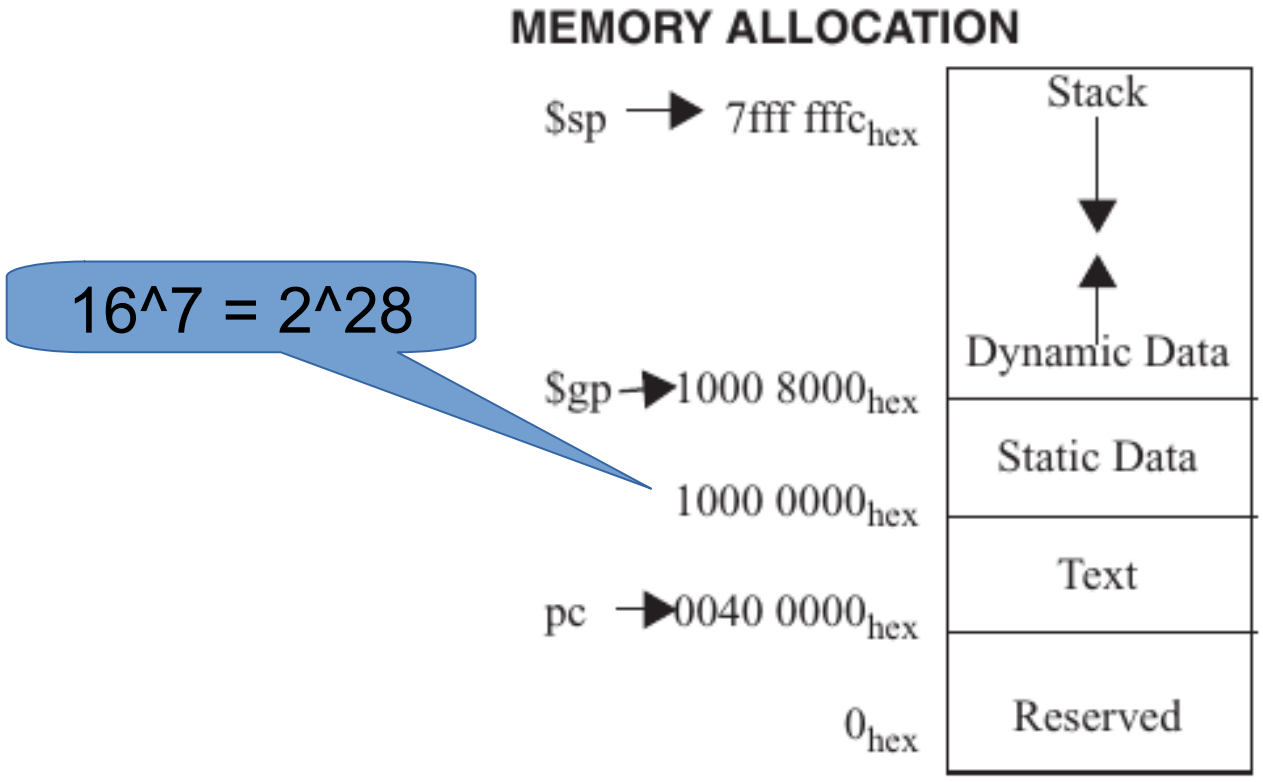
  target = { PC[31:28], immediate$_{26}$, 2'b00 }

  PC ← target

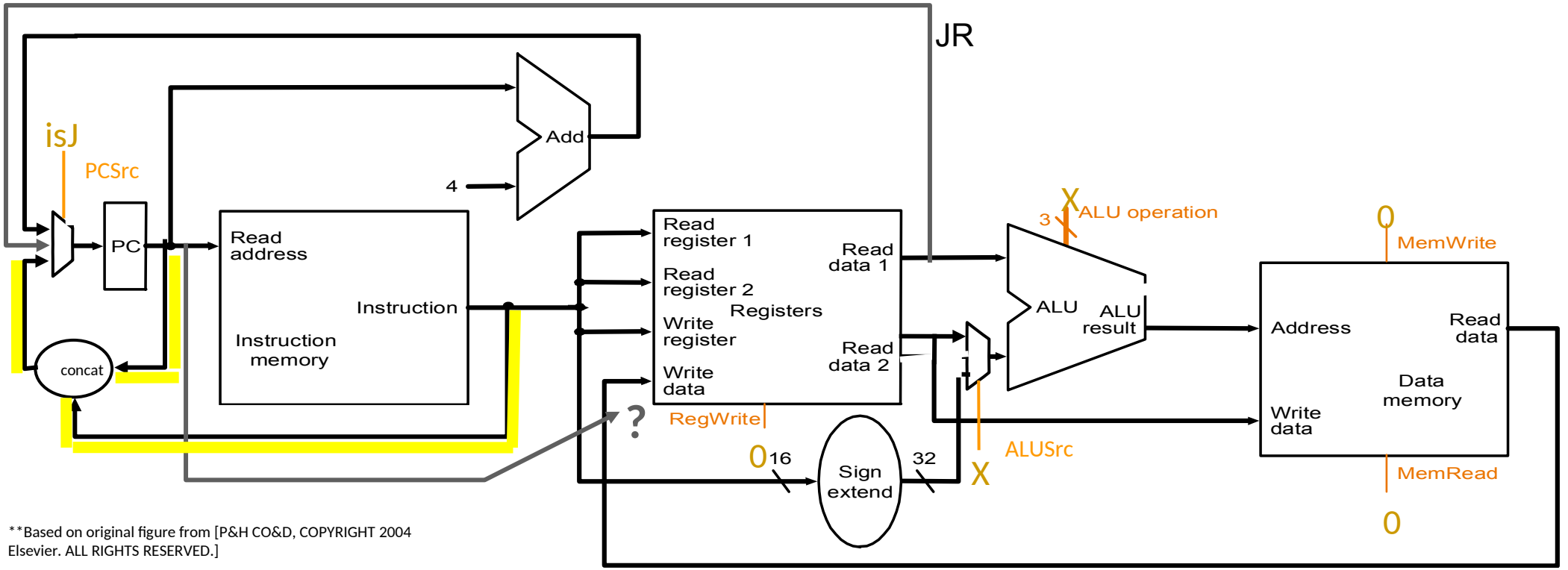
Guy> This is the addressing mode convention: concatenate the PC+imm+2'b00

תזכורת מפת זיכרון ה- MIPS

לקוח מה- MIPS reference card



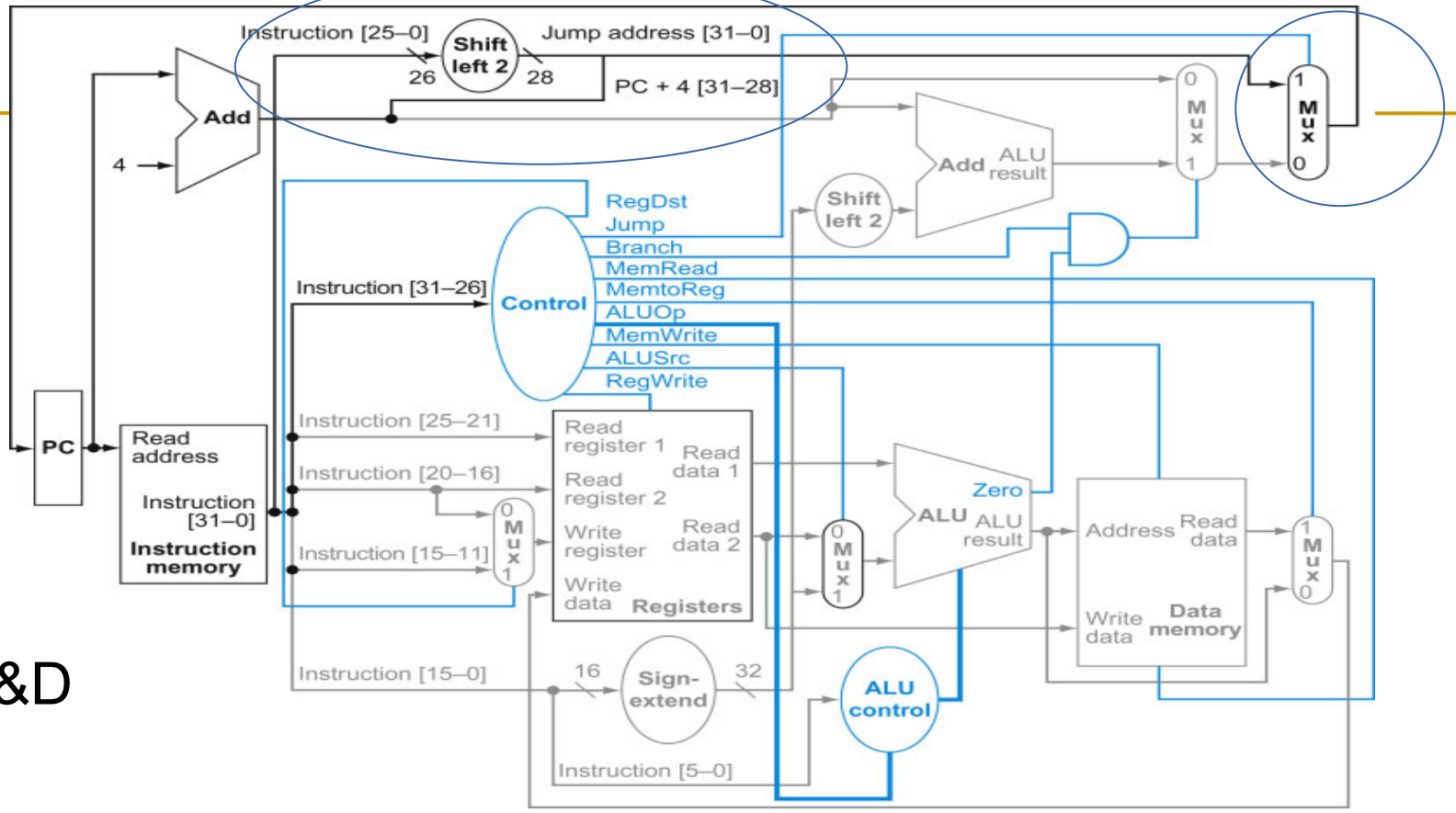
Unconditional Jump Datapath



**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?



H&P CO&D

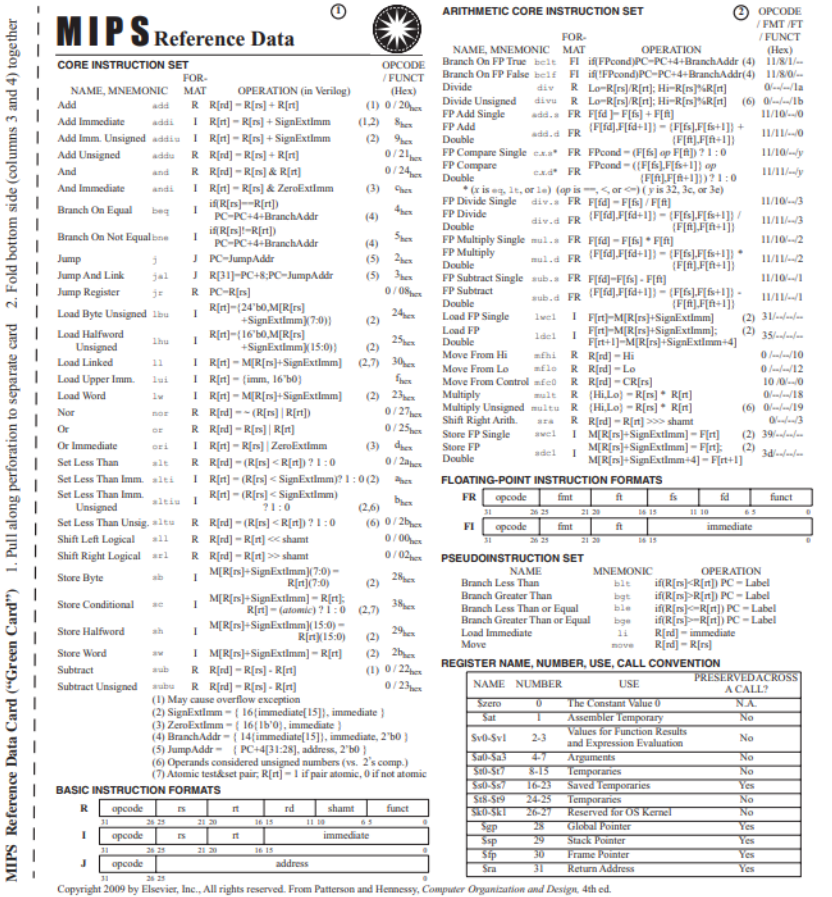
FIGURE 4.24 The simple control and datapath are extended to handle the jump instruction. An additional multiplexor (at the upper right) is used to choose between the jump target and either the branch target or the sequential instruction following this one. This multiplexor is controlled by the jump control 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 Cheat Sheet

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

להזכיר:

Pseudo Instructions



MIPS Reference Data Card ("Green Card") 1. Pull along perforation to separate card 2. Fold bottom side (columns 3 and 4) together

MIPS Reference Data

CORE INSTRUCTION SET

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

(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

R: opcode (31–26) | rs (25–21) | rt (20–16) | rd (15–11) | shamt (10–6) | funct (5–0)

I: opcode (31–26) | rs (25–21) | rt (20–16) | immediate (15–0)

J: opcode (31–26) | address (25–0)

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]; (2) F[rt+1]=M[R[rs]+SignExtImm+4]	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]; (2) M[R[rs]+SignExtImm+4] = F[rt+1]	3d/--/--/--

FLOATING-POINT INSTRUCTION FORMATS

FR: opcode (31–26) | fmt (25–21) | ft (20–16) | fs (15–11) | fd (10–6) | funct (5–0)

FI: opcode (31–26) | fmt (25–21) | ft (20–16) | immediate (15–0)

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

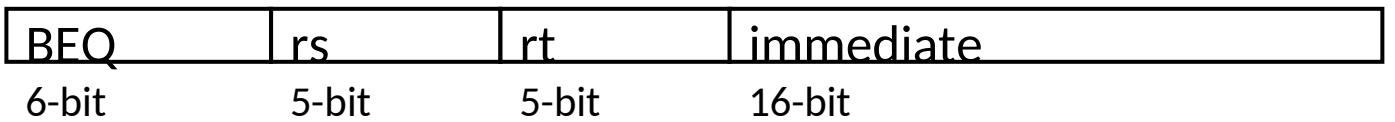
Please download!
נמצא במודל

Conditional Branch Instructions

- 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 $immediate_{16}$

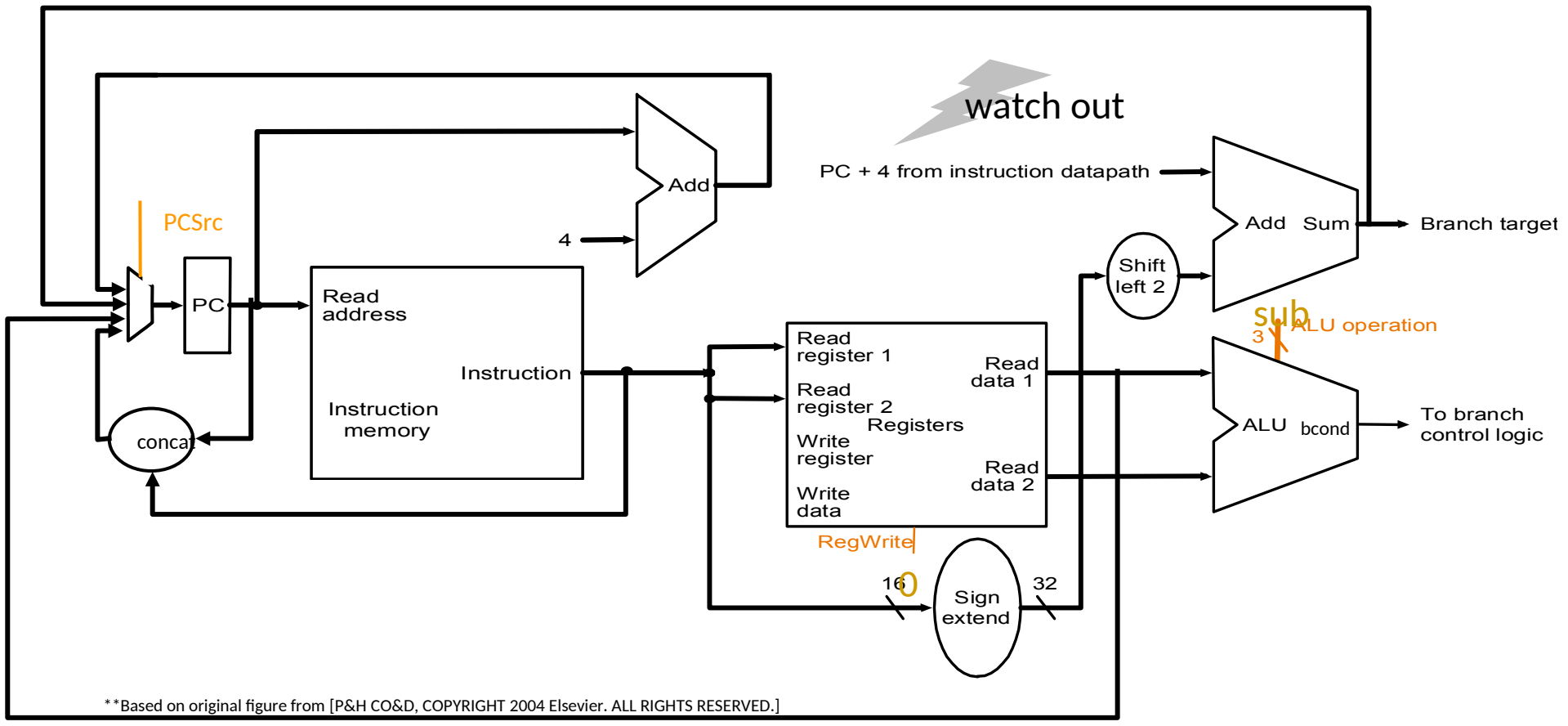
  target = PC + 4 + sign-extend(immediate) x 4

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

  else PC ← PC + 4

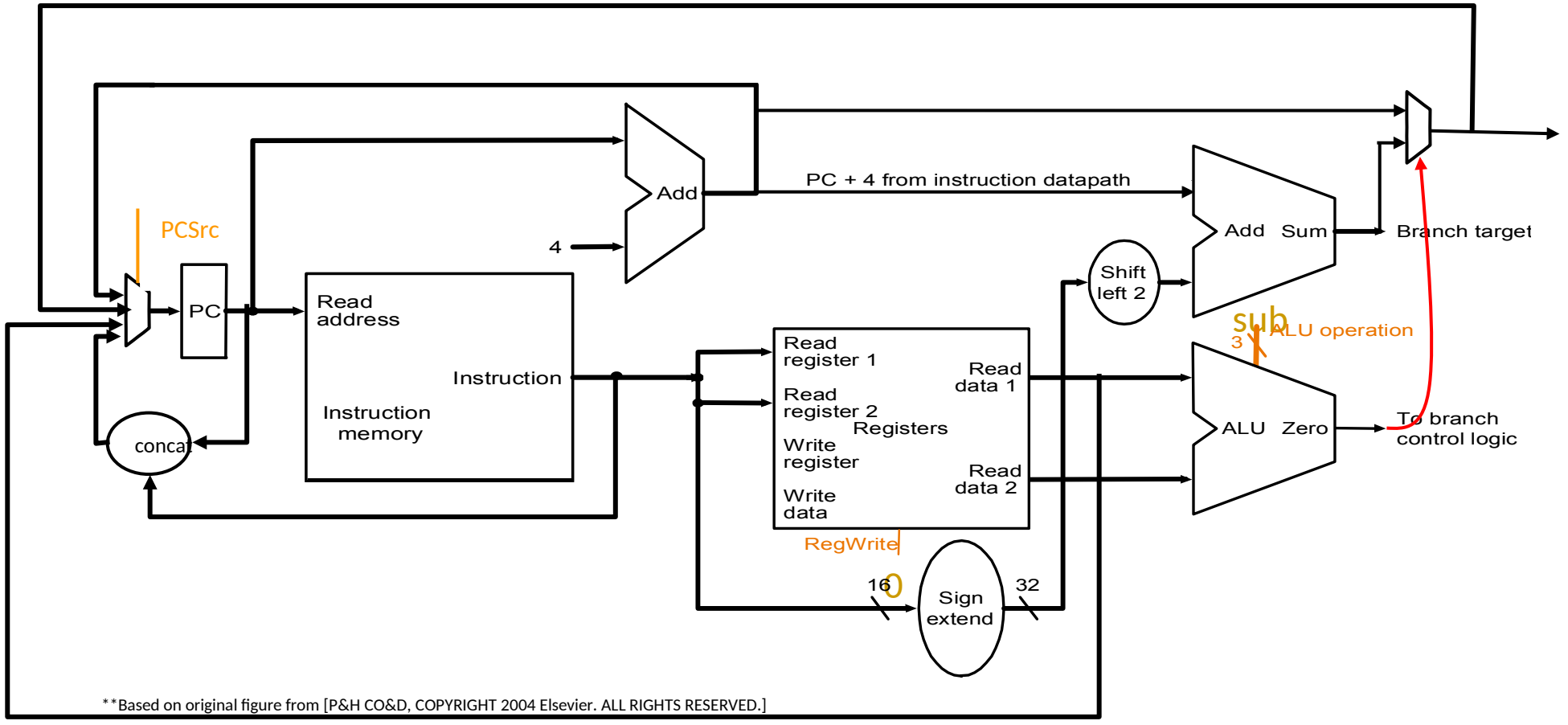
פעולת השוואה נעשית ב-ALU

Conditional Branch Datapath (for you to finish)

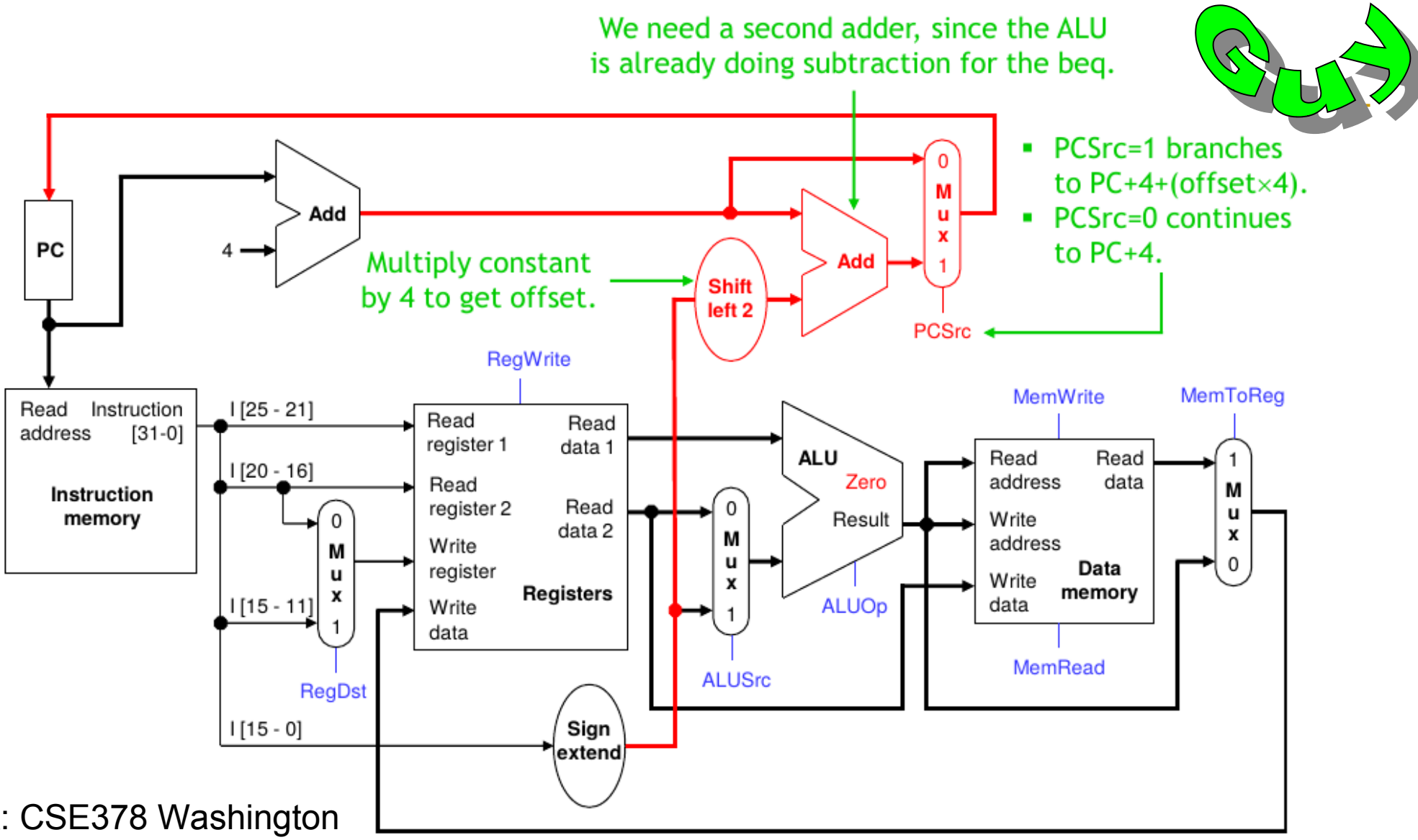


**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

Conditional Branch Datapath (finished :-))

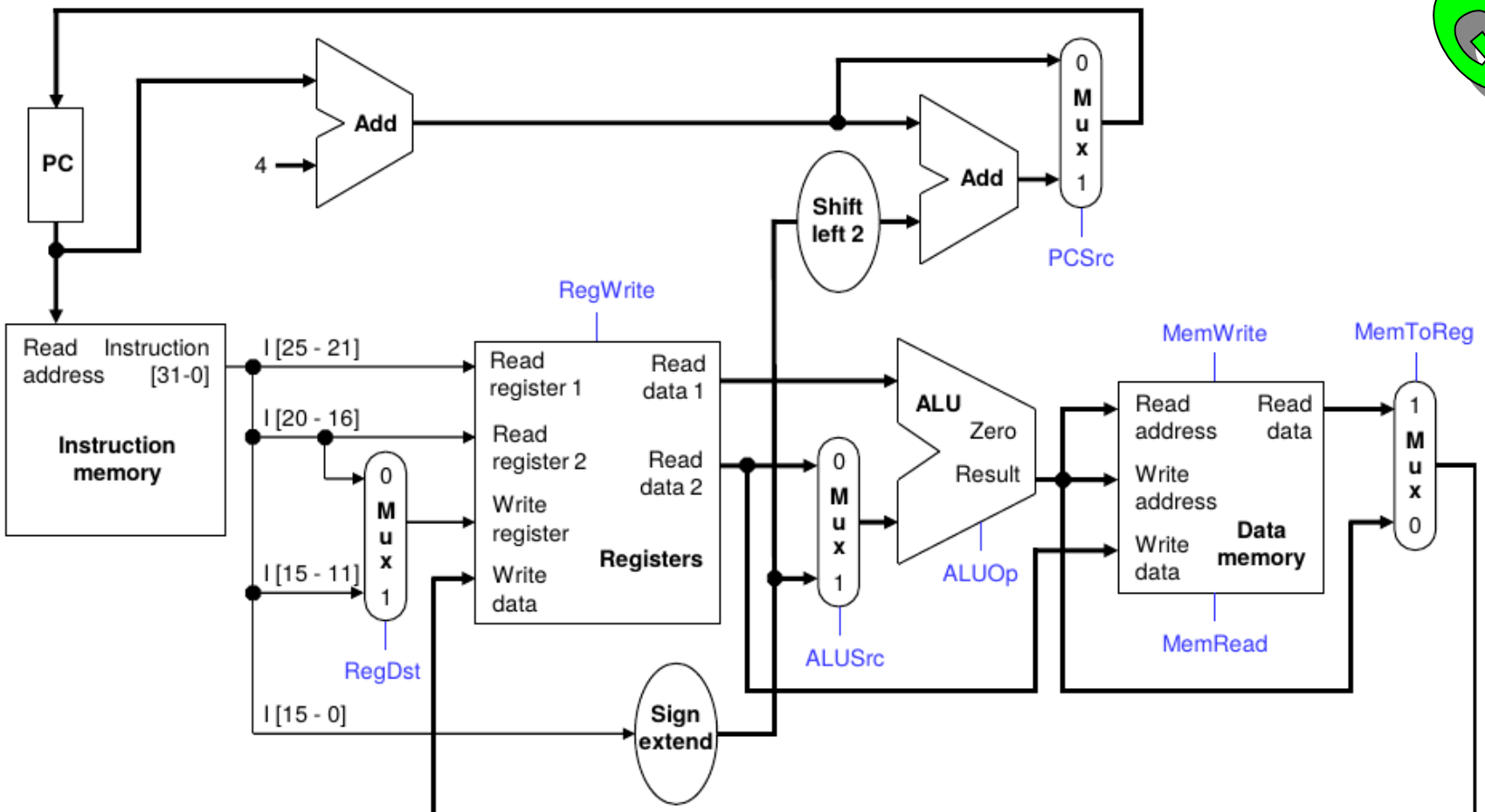


**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]



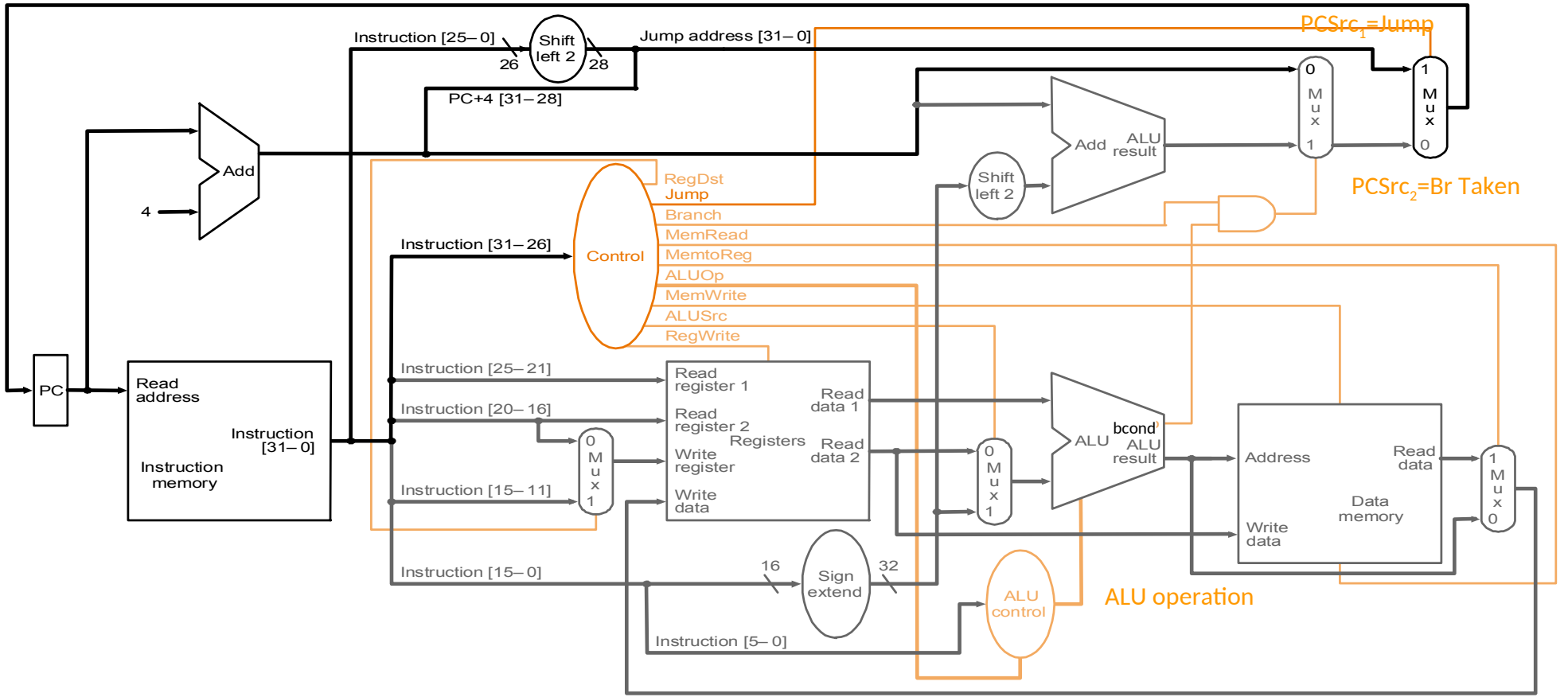
Credit: CSE378 Washington

The Final Datapath (without the controls)



Credit: CSE378 Washington

Putting It All Together



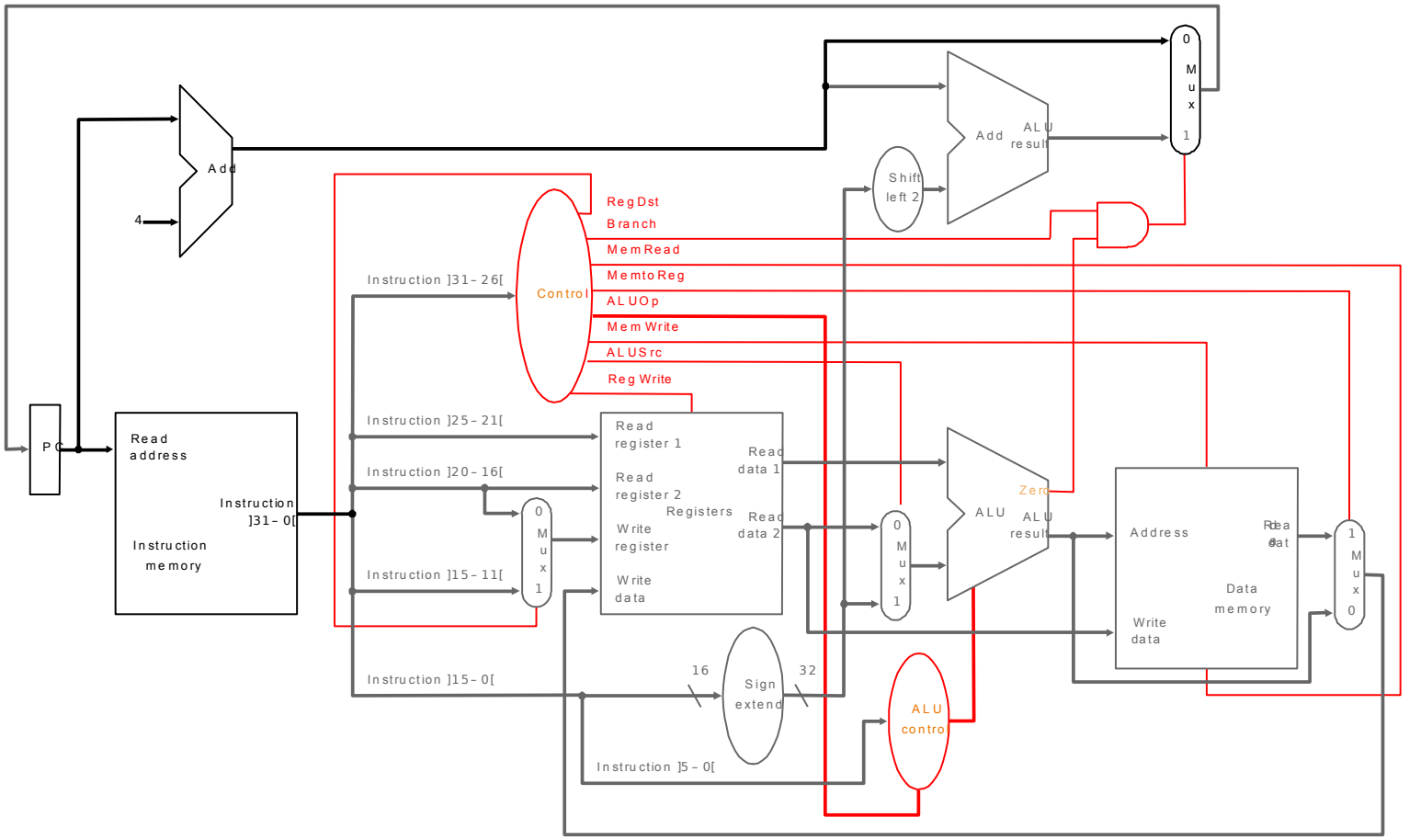
**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

JAL, JR, JALR omitted

Single-Cycle Control Logic

נסביר את אותות הבקרה ששולטים על ה- datapath ב-10 השקפים הבאים של דני

Control



Let's explain all Control Signals:

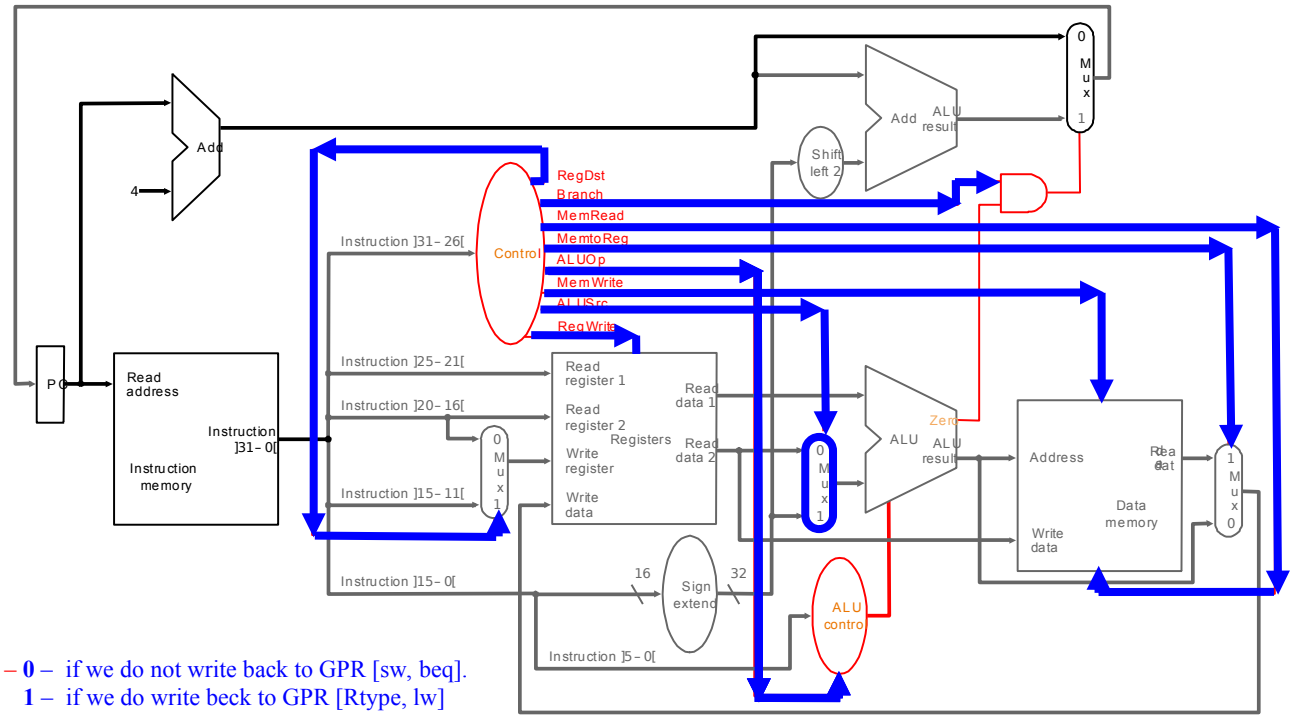
ALUSrc – **0** will connect GPR[Rt] to the ALU B input [Rtype, beq]

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



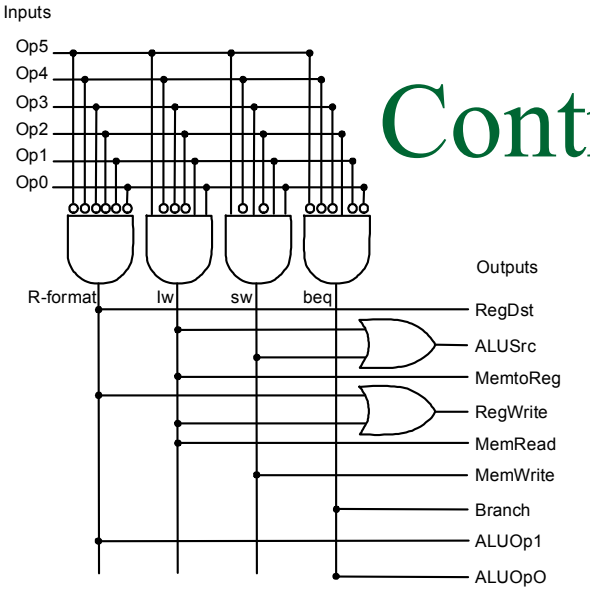
RegWrite – **0** – if we do not write back to GPR [sw, beq].
1 – if we do write beck to GPR [Rtype, lw]

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]

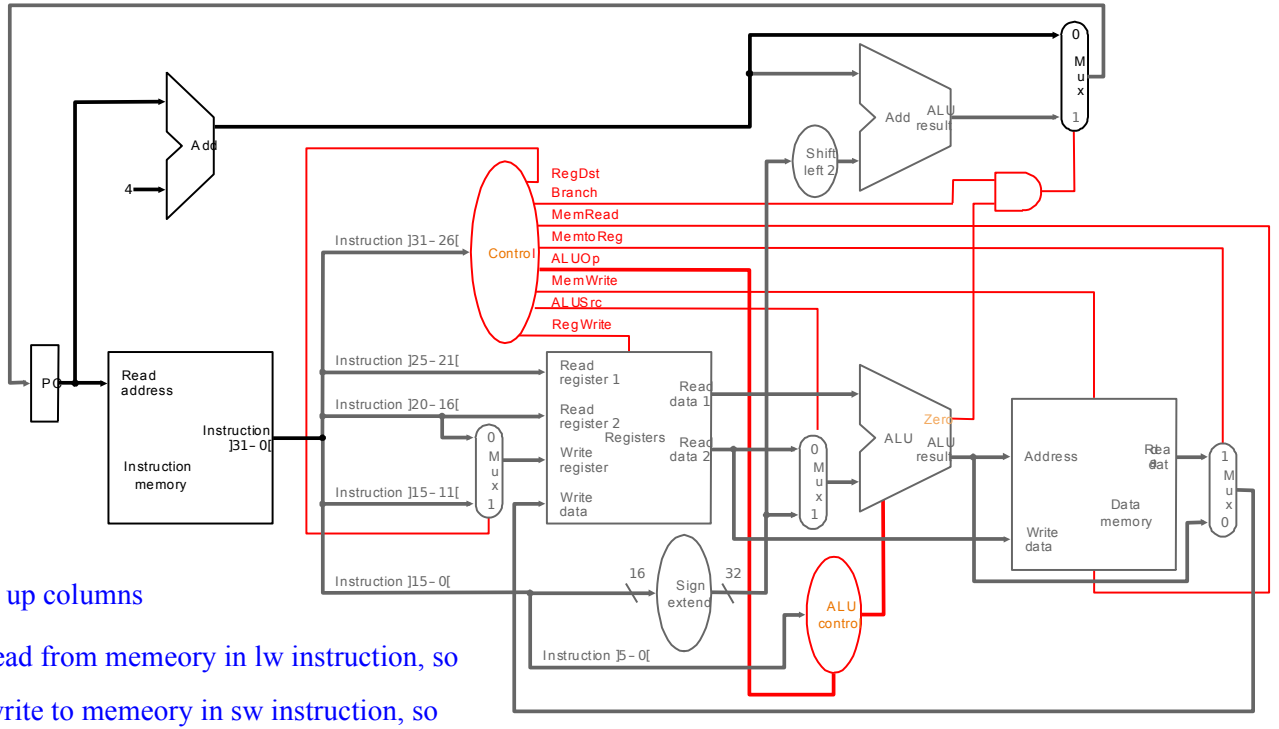
Branch – **0** – ia all instructions except beq. **1**- in beq. Selects whether to branch or not.

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									



We can fill up columns

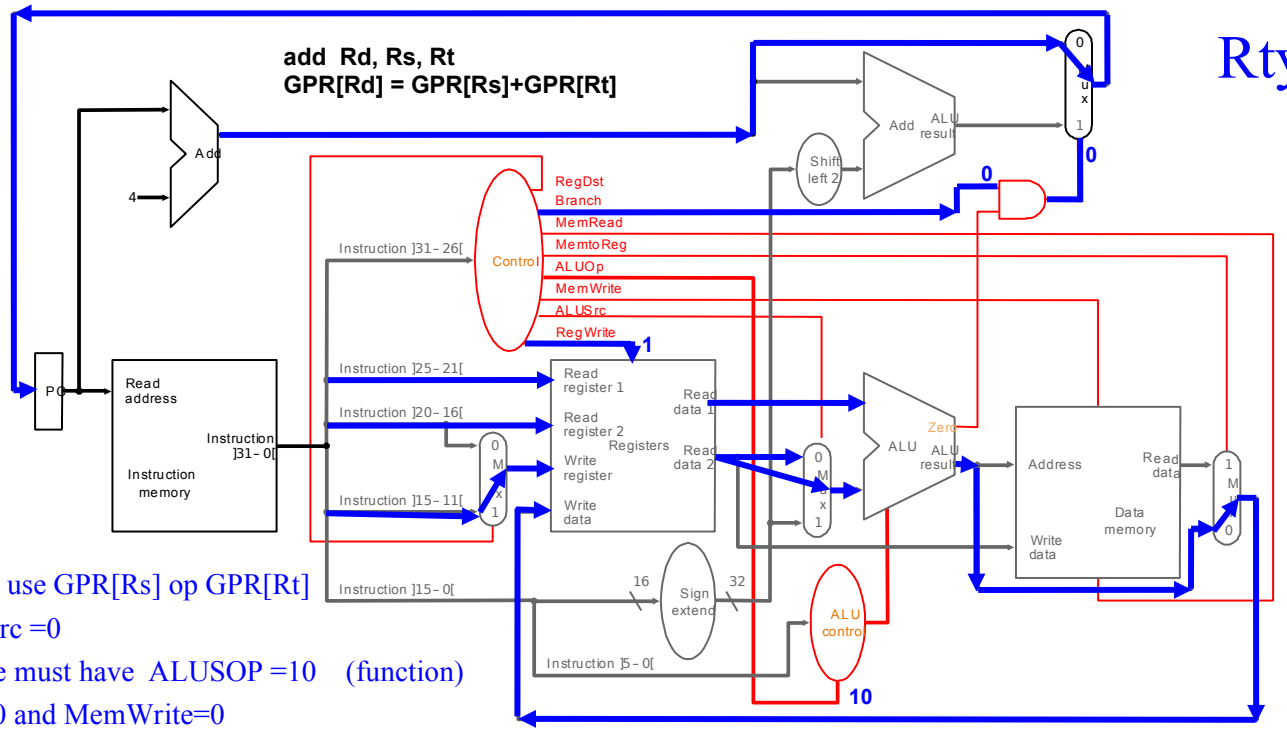
We only read from memeory in lw instruction, so

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

Rtype



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

Thus, ALUSrc =0

In Rtype we must have ALUSOP =10 (function)

MemRead=0 and MemWrite=0

MemToReg=0

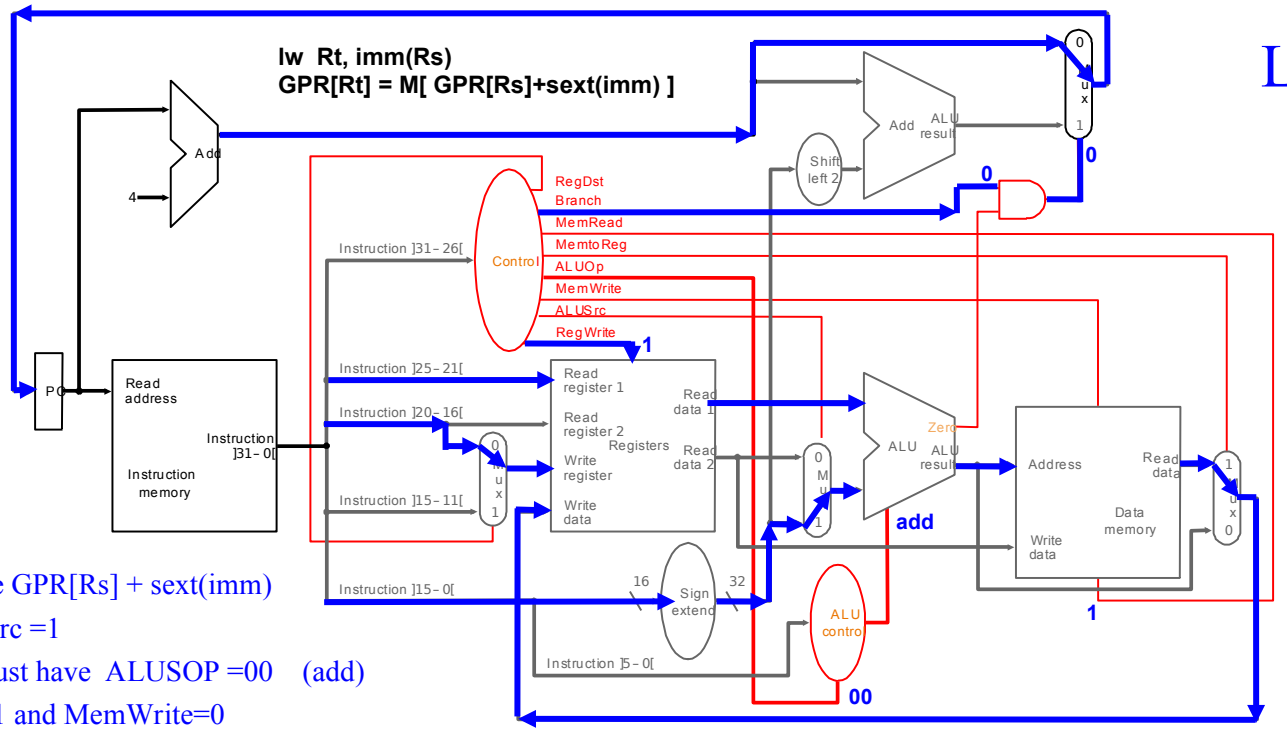
Instr.	ALU Src	ALUOp [1:0]		Mem Read	Mem Write	Mem to Reg	Reg Dst	Reg Write	Branch
R-type	0	1	0	0	0	0	1	1	0
lw				1	0				
sw				0	1				
beq				0	0				

RegDst=1 to select Rd

RegWrite=1 to write to the GPR

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

LW



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

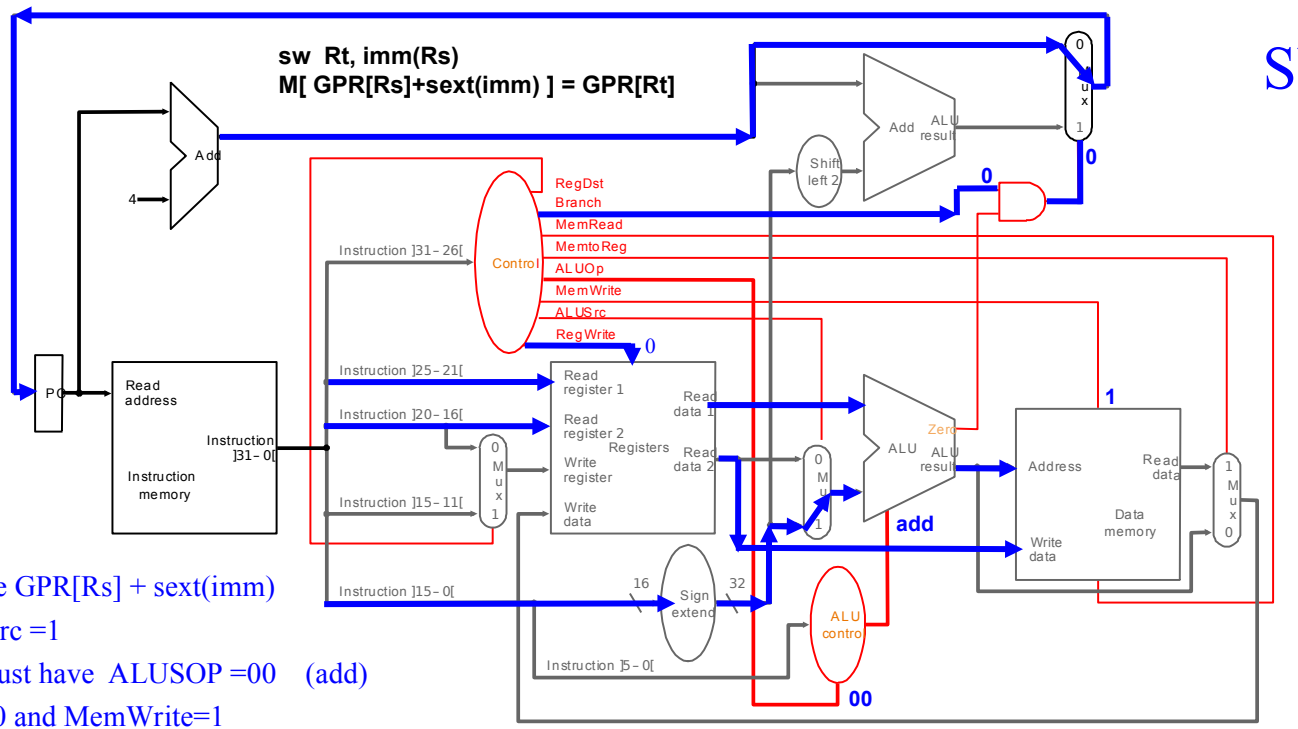
Instr.	ALU Src	ALUOp [1:0]		Mem Read	Mem Write	Mem to Reg	Reg Dst	Reg Write	Branch
R-type	0	1	0	0	0	0	1	1	0
lw	**1**	**0**	**0**	1	0	**1**	**0**	**1**	**0**
sw				0	1				
beq				0	0				

RegDst=0 to select Rt

RegWrite=1 to write to the GPR

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

SW



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 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				

RegDst=X. We do not write back!

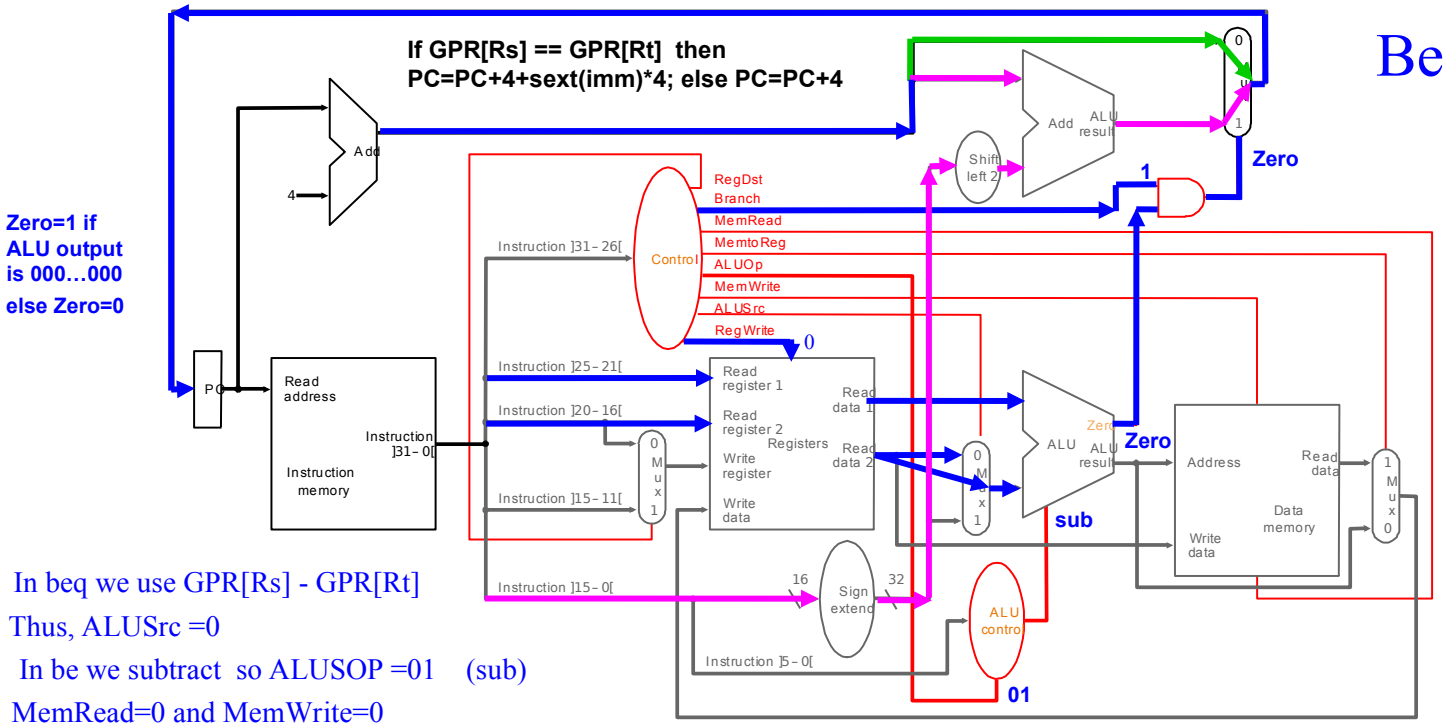
RegWrite=0. We do not write back!

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

Beq

If GPR[Rs] == GPR[Rt] then
PC=PC+4+sext(imm)*4; else PC=PC+4

Zero=1 if
ALU output
is 000…000
else Zero=0



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

Thus, ALUSrc =0

In be we subtract so ALUSOP =01 (sub)

MemRead=0 and MemWrite=0

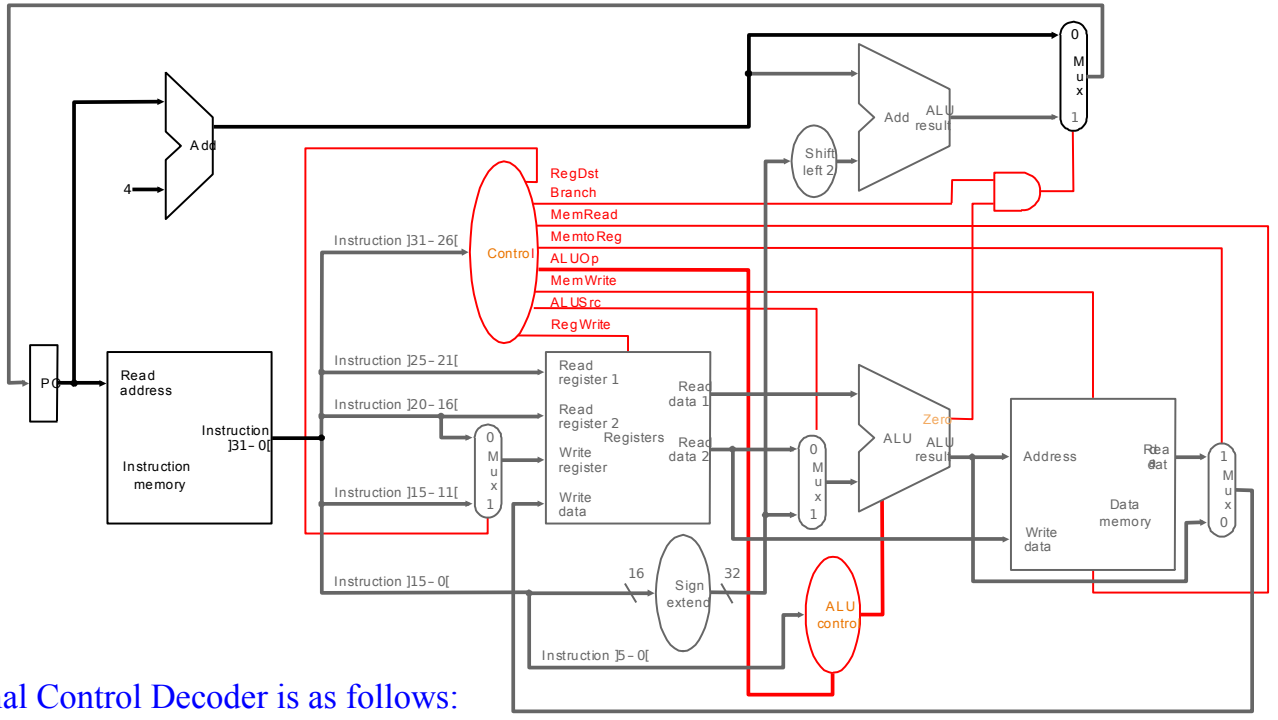
MemToReg=X. We do not write back!

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!

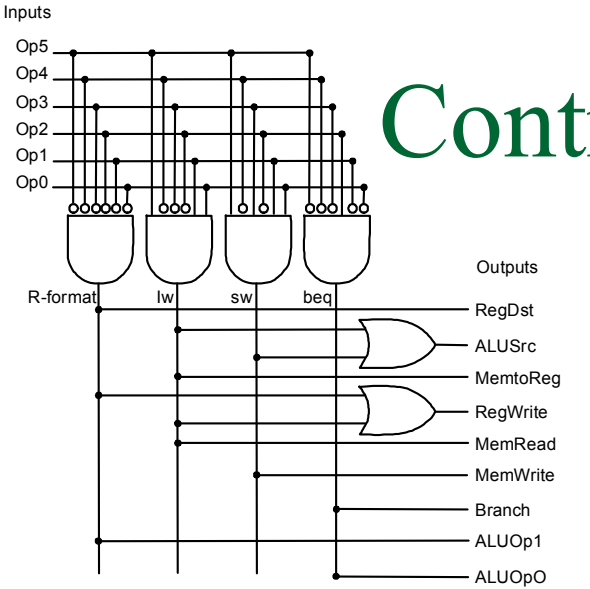
Branch=1 so we check Zero to
decide whether to
branch (PC= branch target)
or not to branch (PC=PC+4)



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

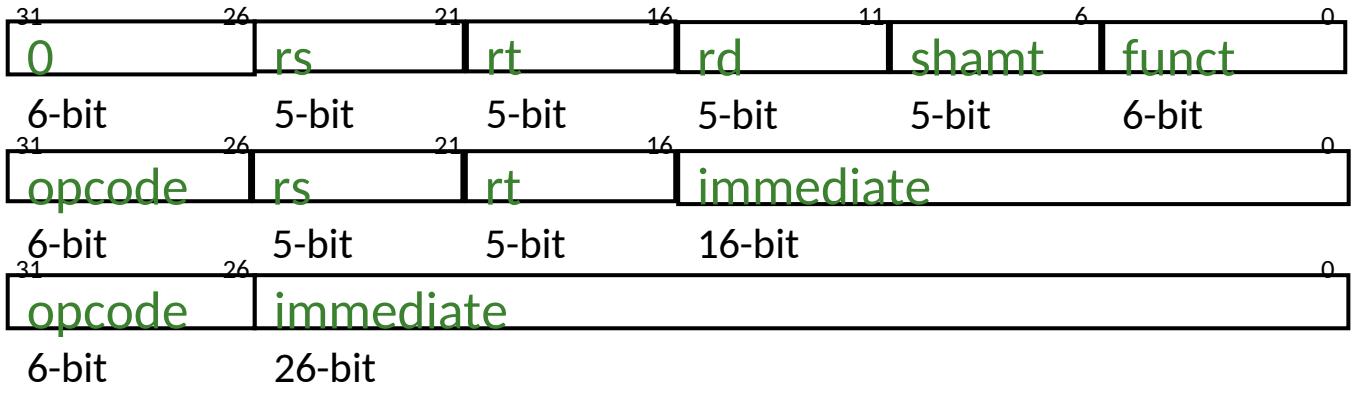


Instr.	ALU Src	ALUOp [1:0]		Mem Read	Mem Write	Mem to Reg	Reg Dst	Reg Write	Branch
R-type	0	1	0	0	0	0	1	1	0
lw	1	0	0	1	0	1	0	1	0
sw	1	0	0	0	1	X	X	0	0
beq	0	0	1	0	0	X	X	0	1

סיום השקפים של דני על אותות הבקרה ששולטים על ה- datapath

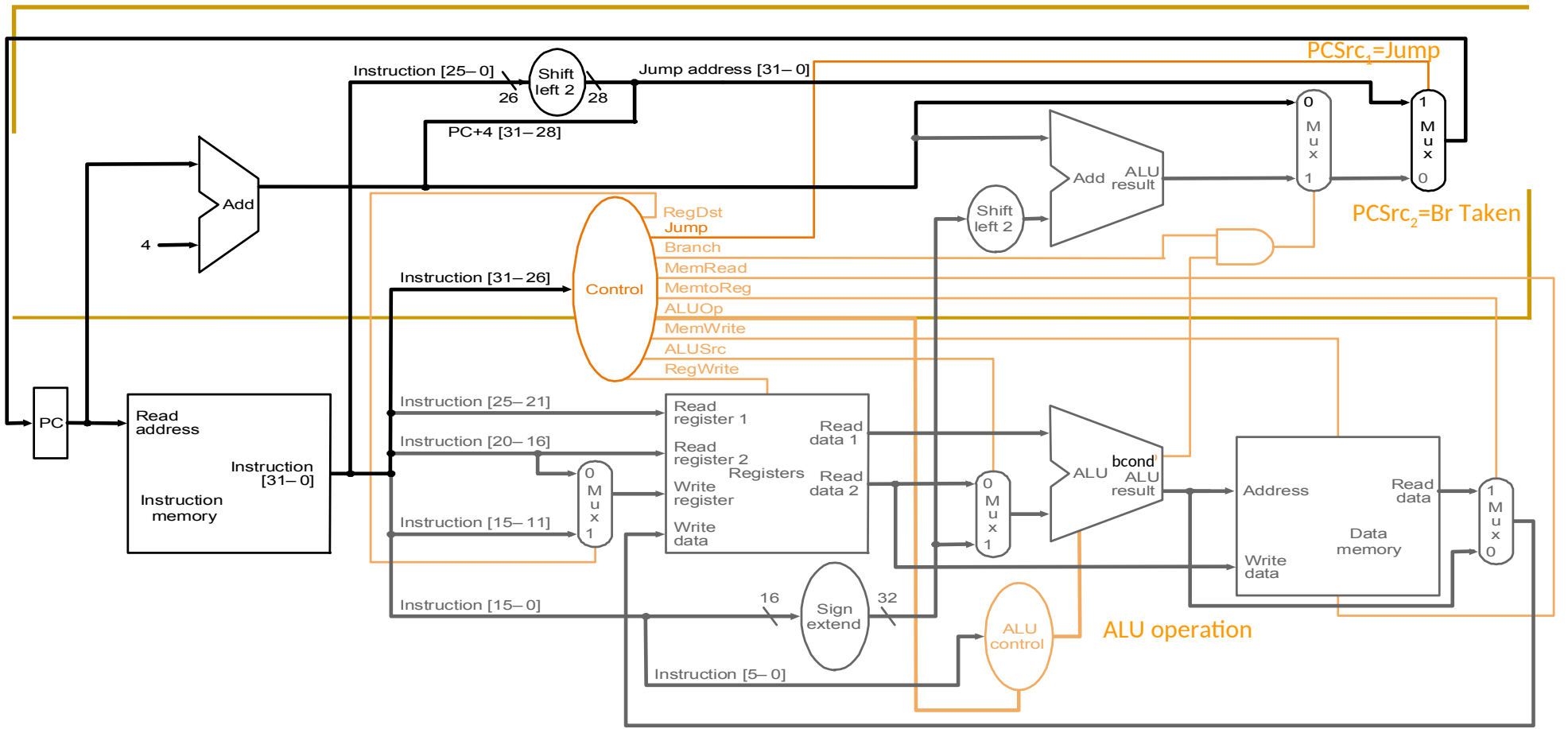
Single-Cycle Hardwired Control

- 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



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

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) && (opcode!=BNE)
MemtoReg	Steer ALU result to GPR write port	Steer memory load to GPR wr. port	opcode==LW
RegWrite	GPR write disabled	GPR write enabled	(opcode!=SW) && (opcode!=Bxx) && (opcode!=J) && (opcode!=JR))

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

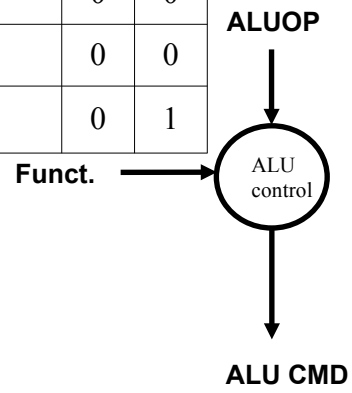
  'SW' ⇒ select addition

  '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.

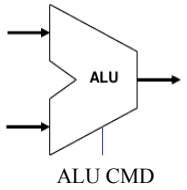
Instr.	ALUOp [1:0]	
R-type	1	0
lw	0	0
sw	0	0
beq	0	1



00 = Add
01= Sub
10 = Funct.

MIPS ALU CMDs

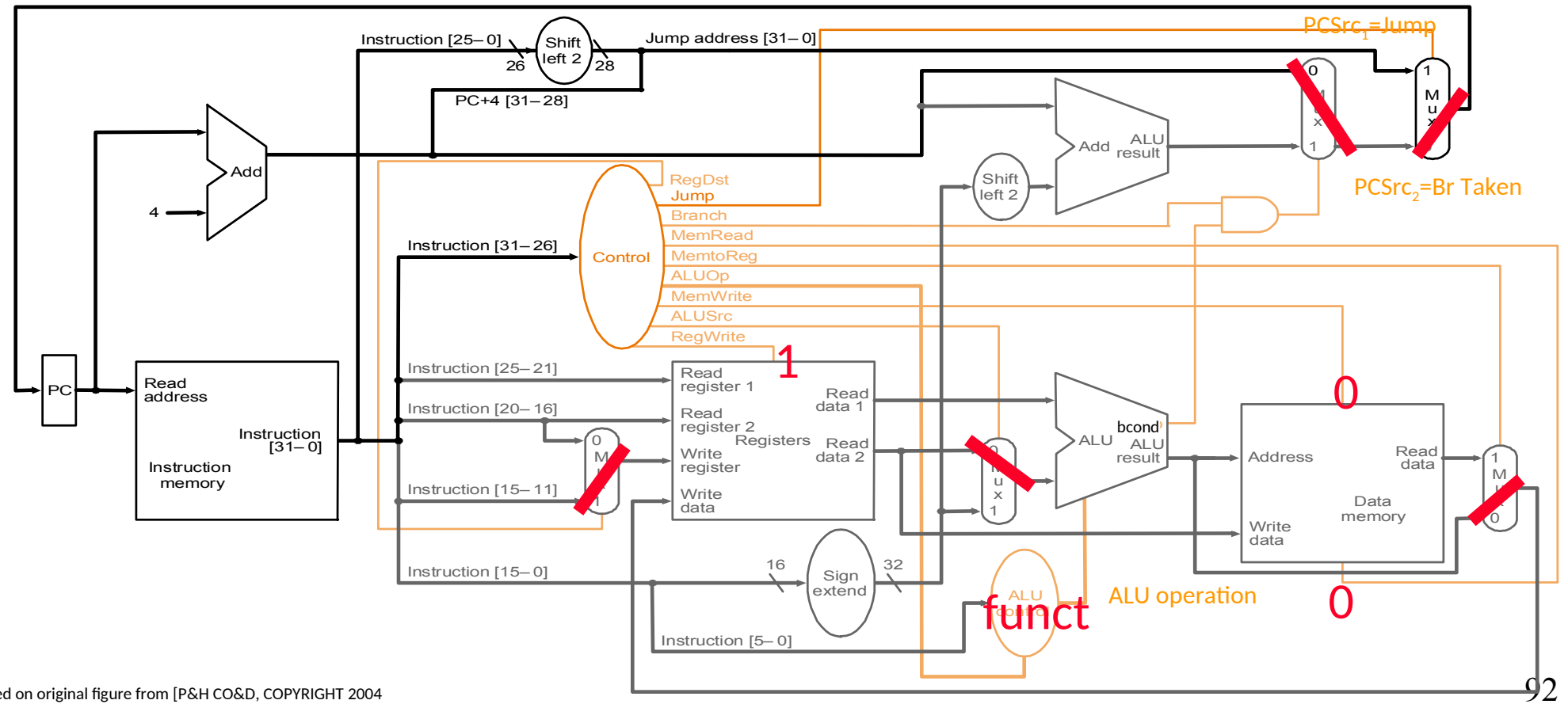
- Here's a simple ALU with five operations, selected by a 3-bit control signal ALU CMD



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

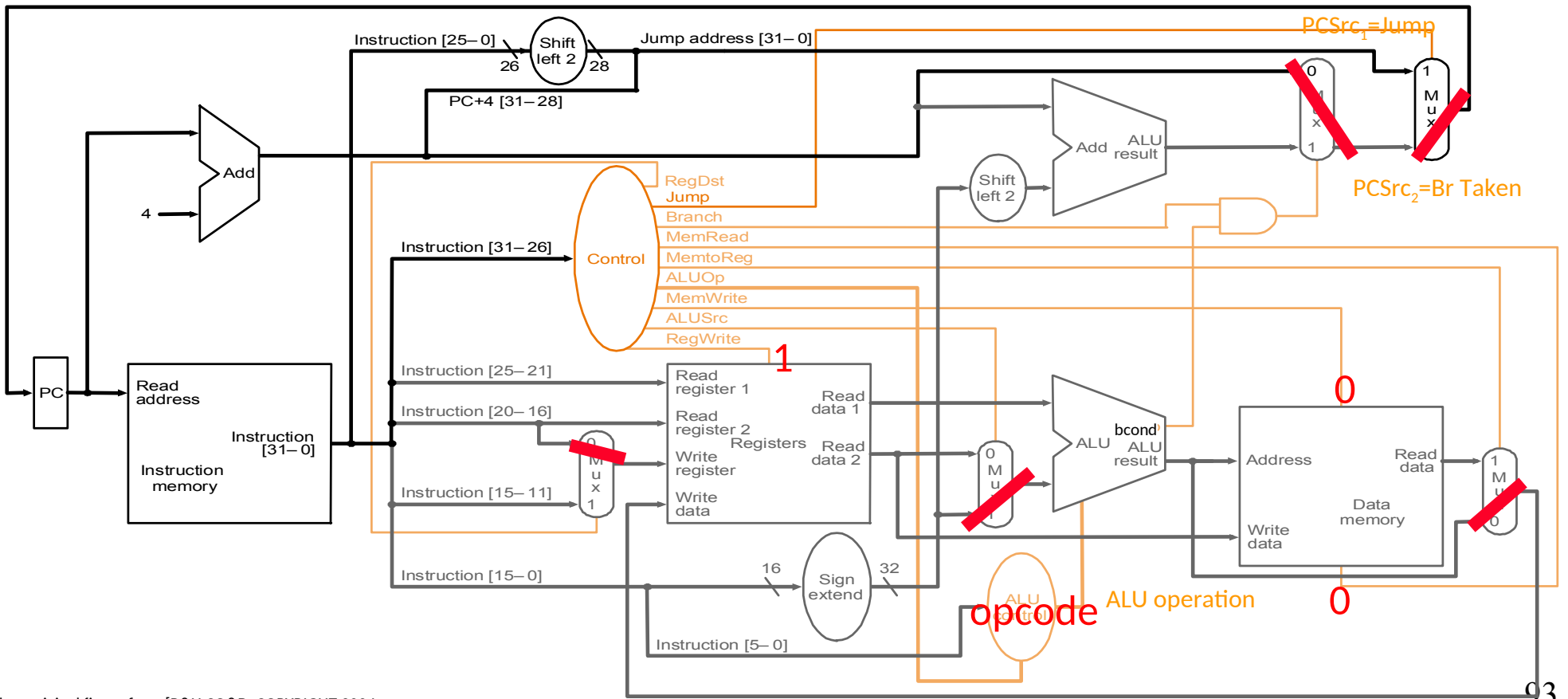
Taken from: **CSE378 University of Washington**
https://courses.cs.washington.edu/courses/cse378/10sp/lectures.html

R-Type ALU



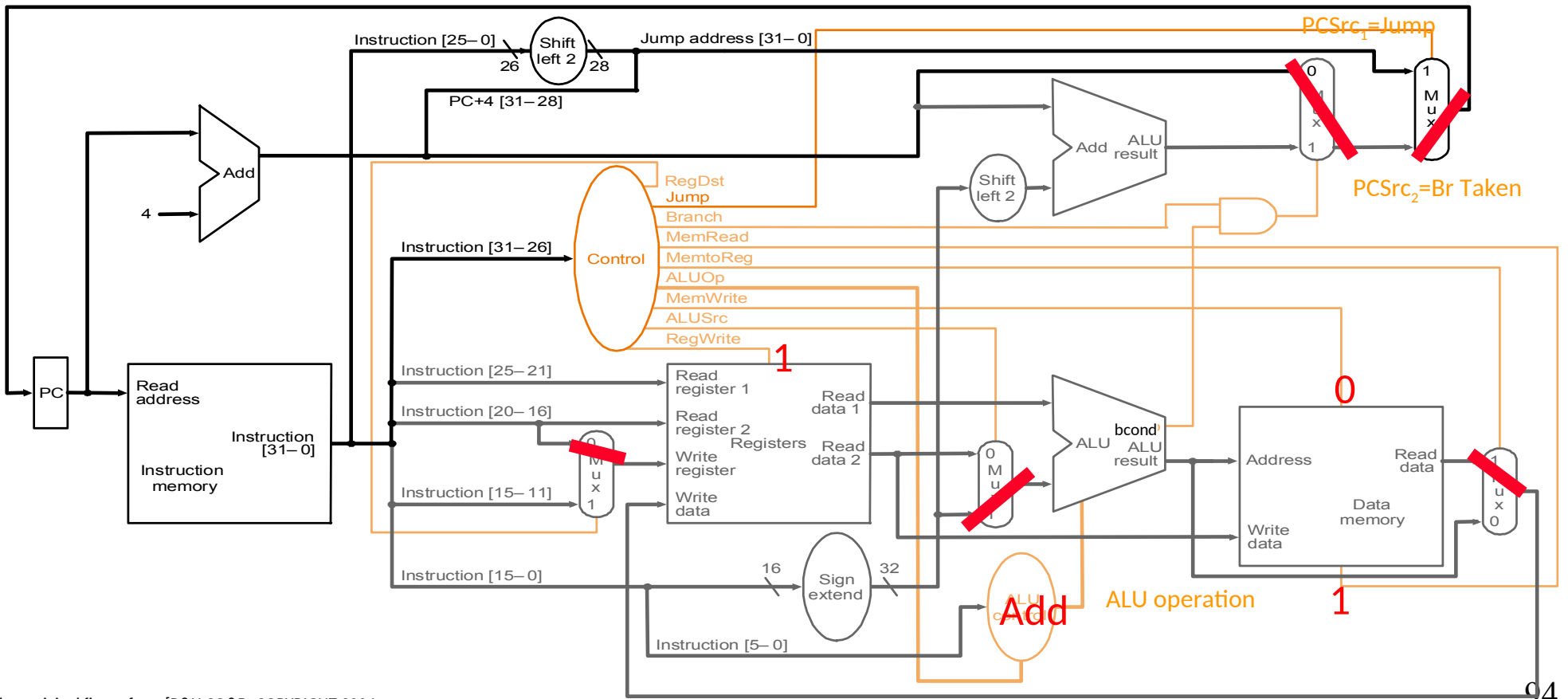
**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

I-Type ALU



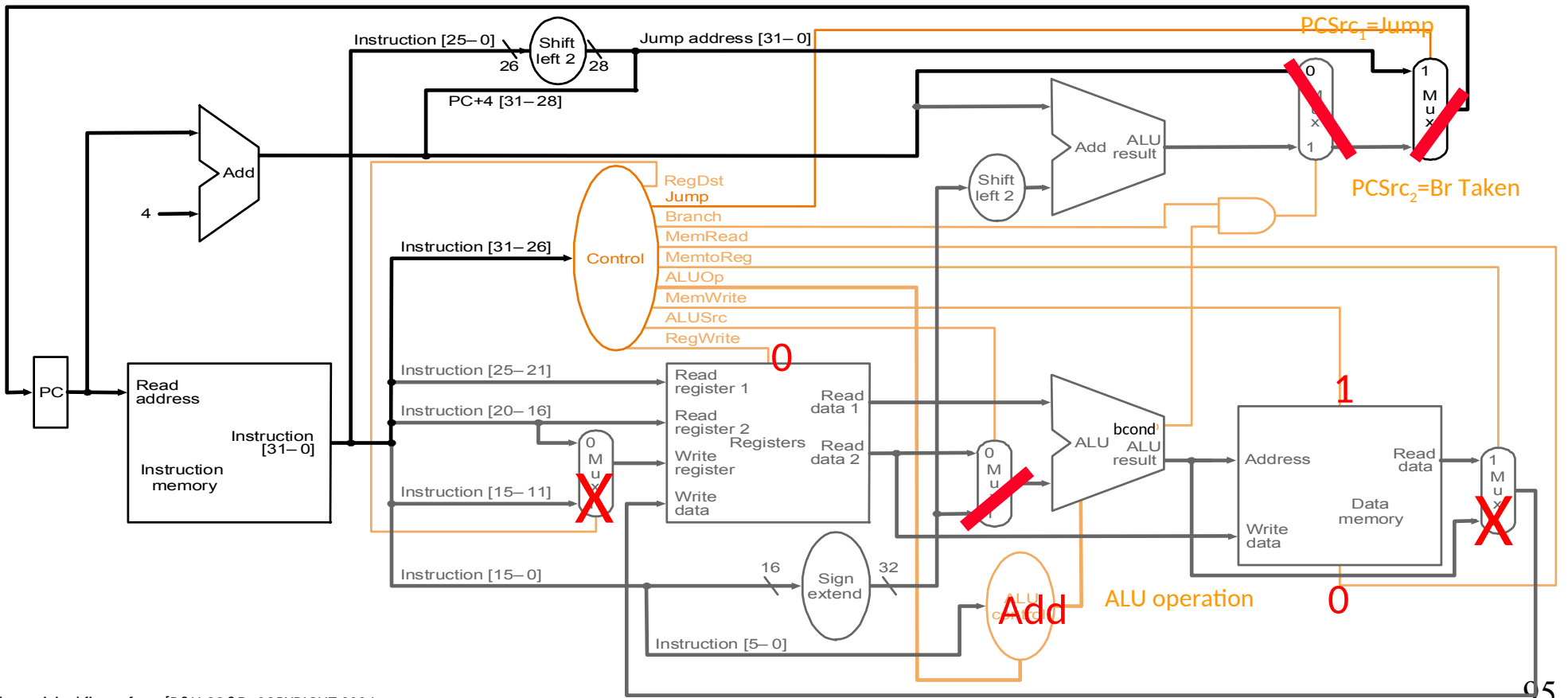
**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

LW



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

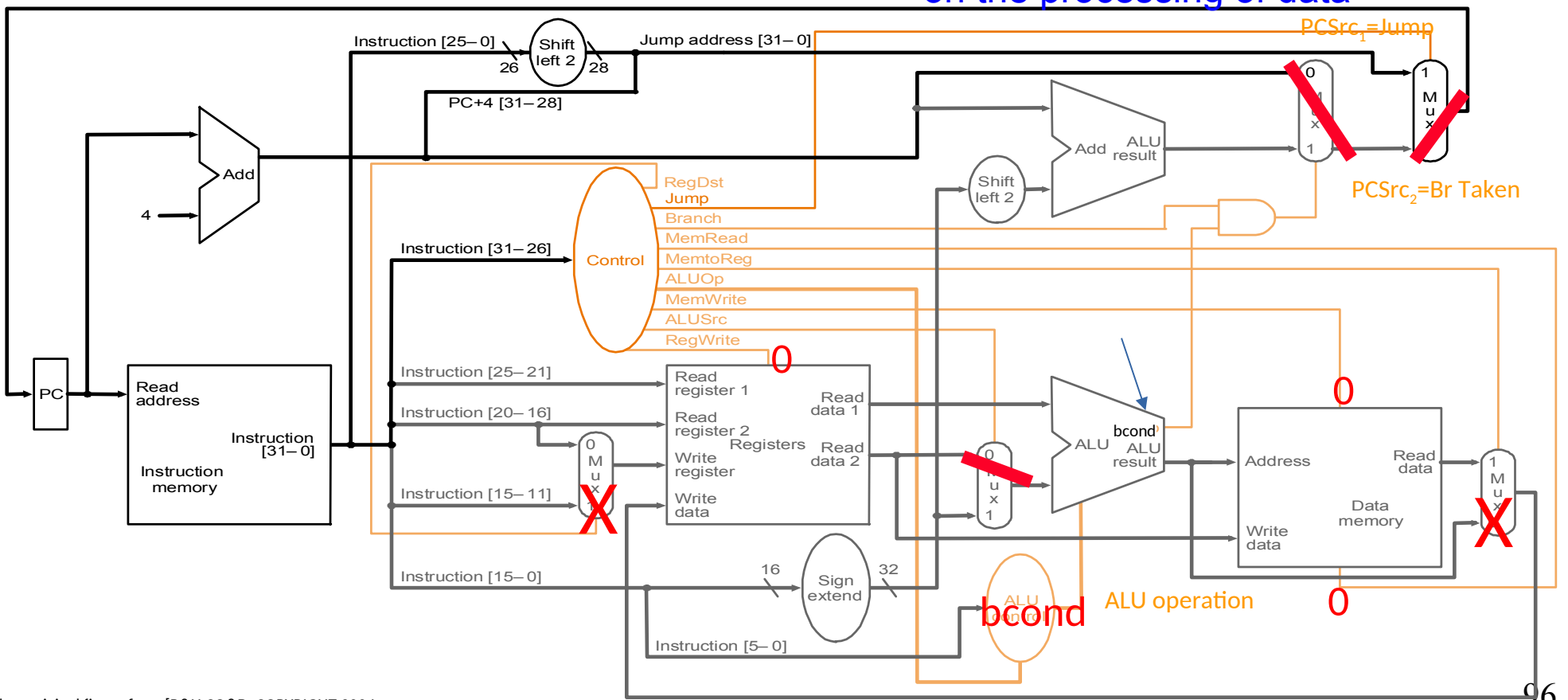
SW



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

Branch (Not Taken)

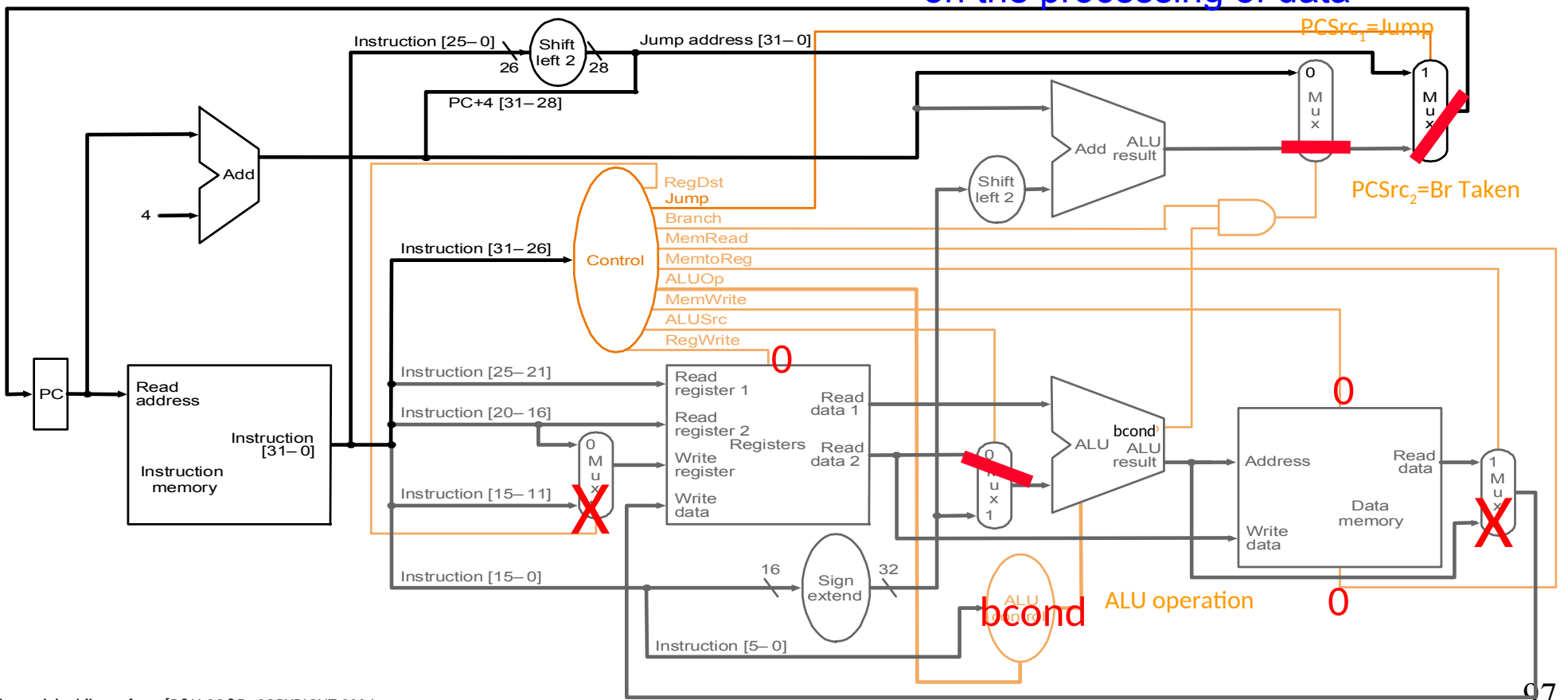
Some control signals are dependent on the processing of data



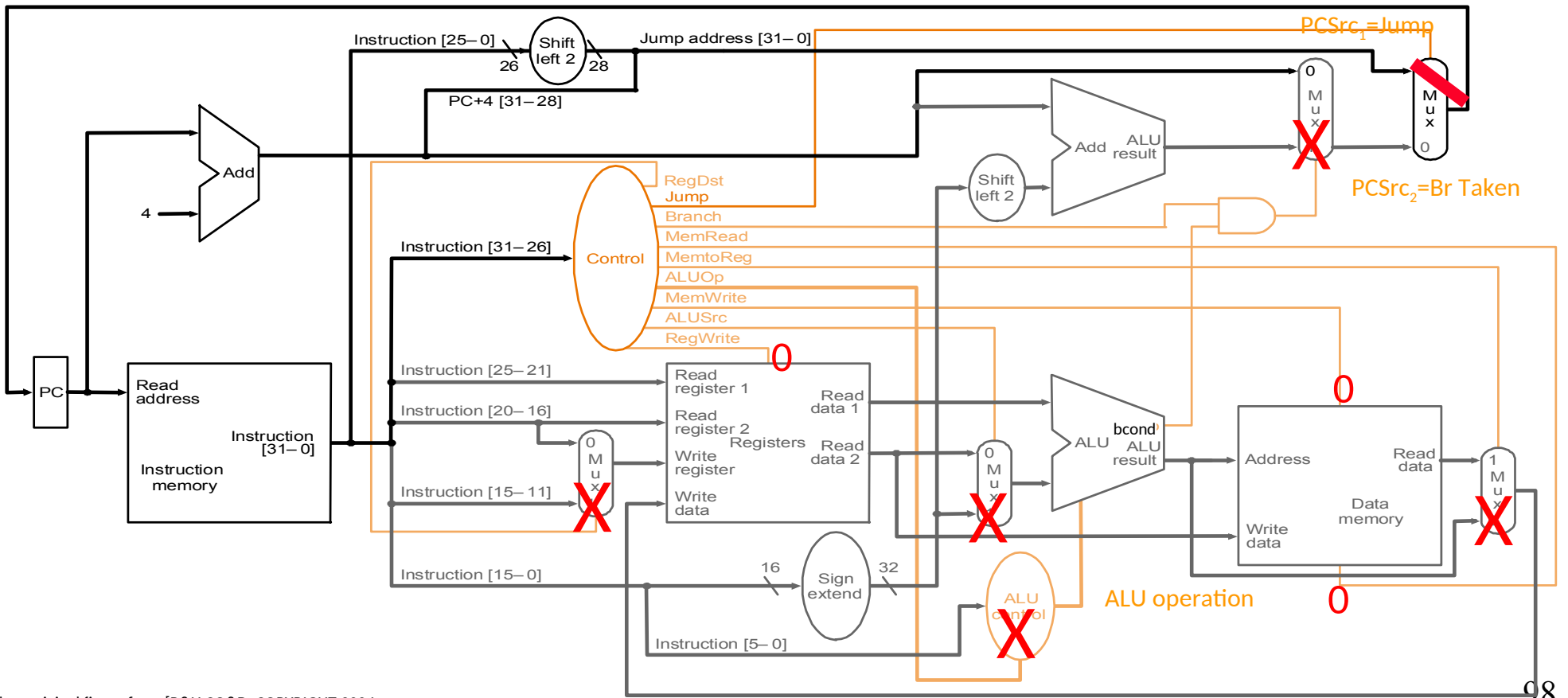
**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

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

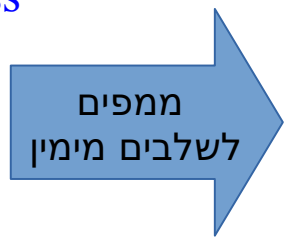
- How long each instruction takes is determined by how long the slowest instruction takes to execute
  - Even though many instructions do not need that long to execute

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



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

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

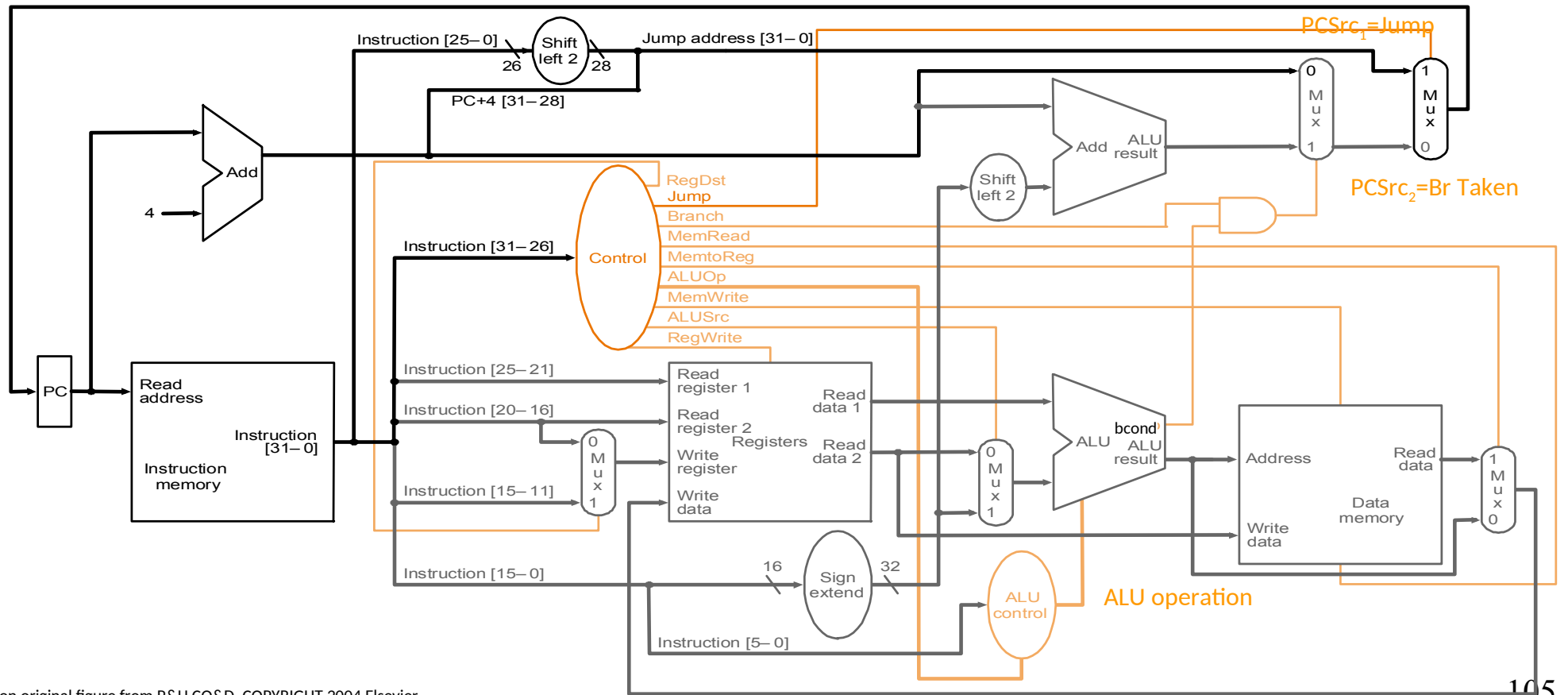
סביר שהפקודה האיטית ביותר תזדקק לכל השלבים הללו

Single-Cycle Datapath Analysis - דוגמה

- Assume – זוהי רק דוגמה
  - memory units (read or write): 200 ps
  - ALU and adders: 100 ps
  - register file (read or write): 50 ps
  - other combinational logic: 0 ps

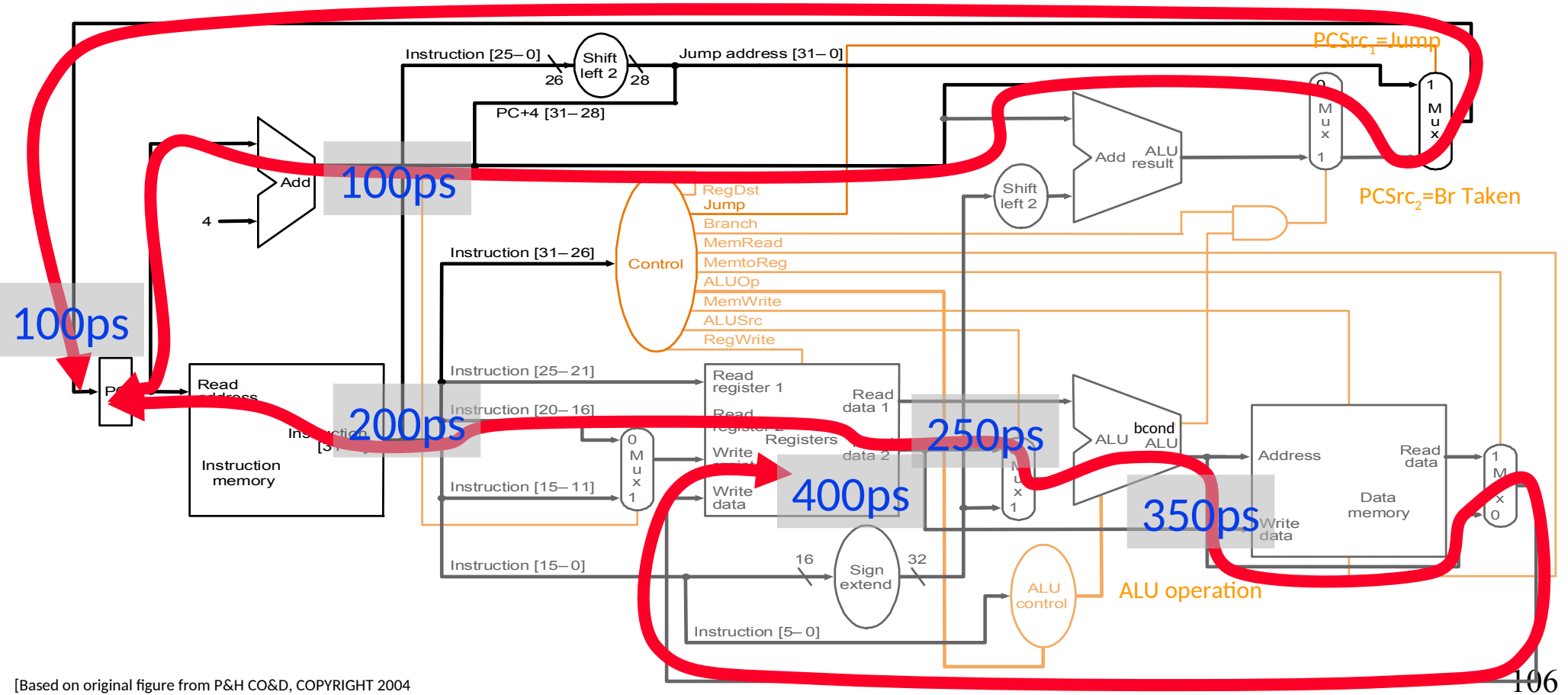
steps	IF	ID	EX	MEM	WB	Delay
resources	mem	RF	ALU	mem	RF	
R-type	200	50	100		50	400
I-type	200	50	100		50	400
LW	200	50	100	200	50	600
SW	200	50	100	200		550
Branch	200	50	100			350
Jump	200					200

Let's Find the Critical Path



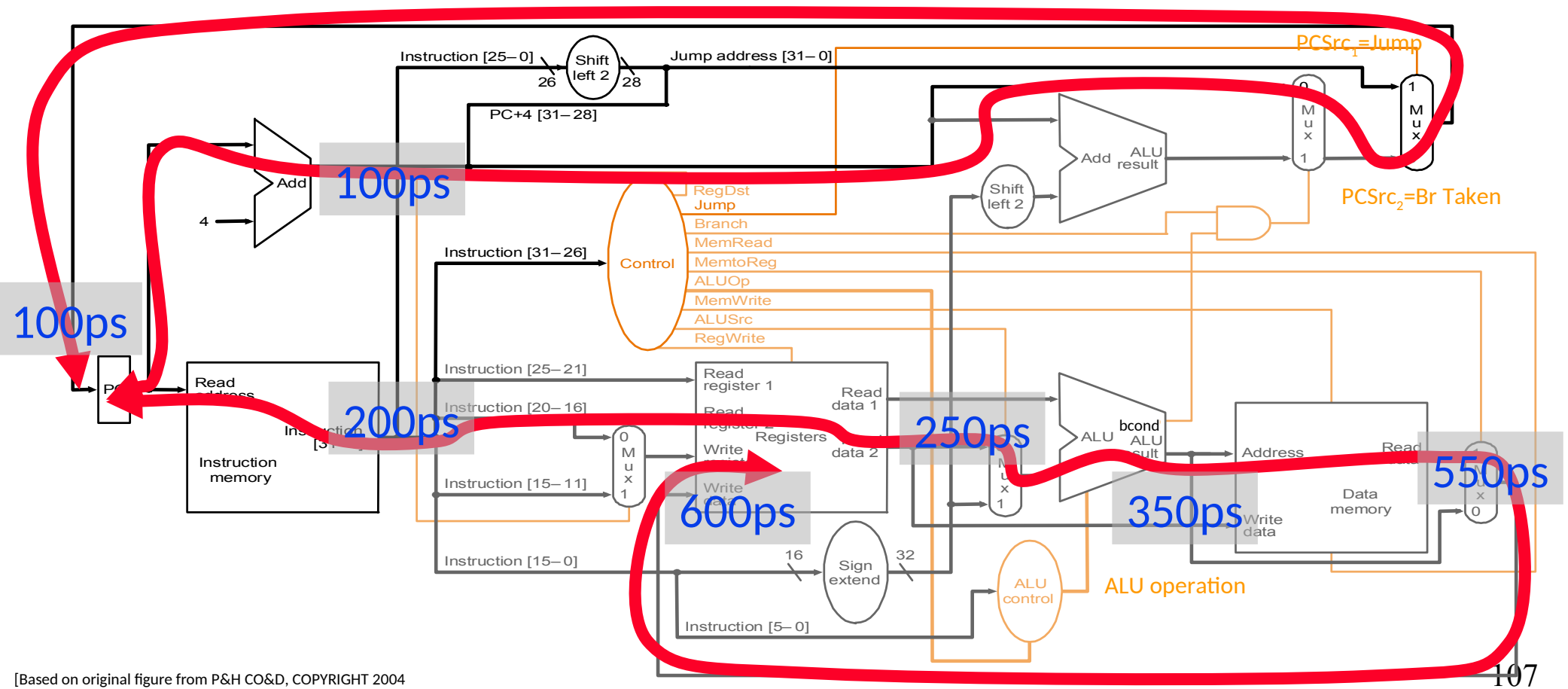
[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

R-Type and I-Type ALU → critical path



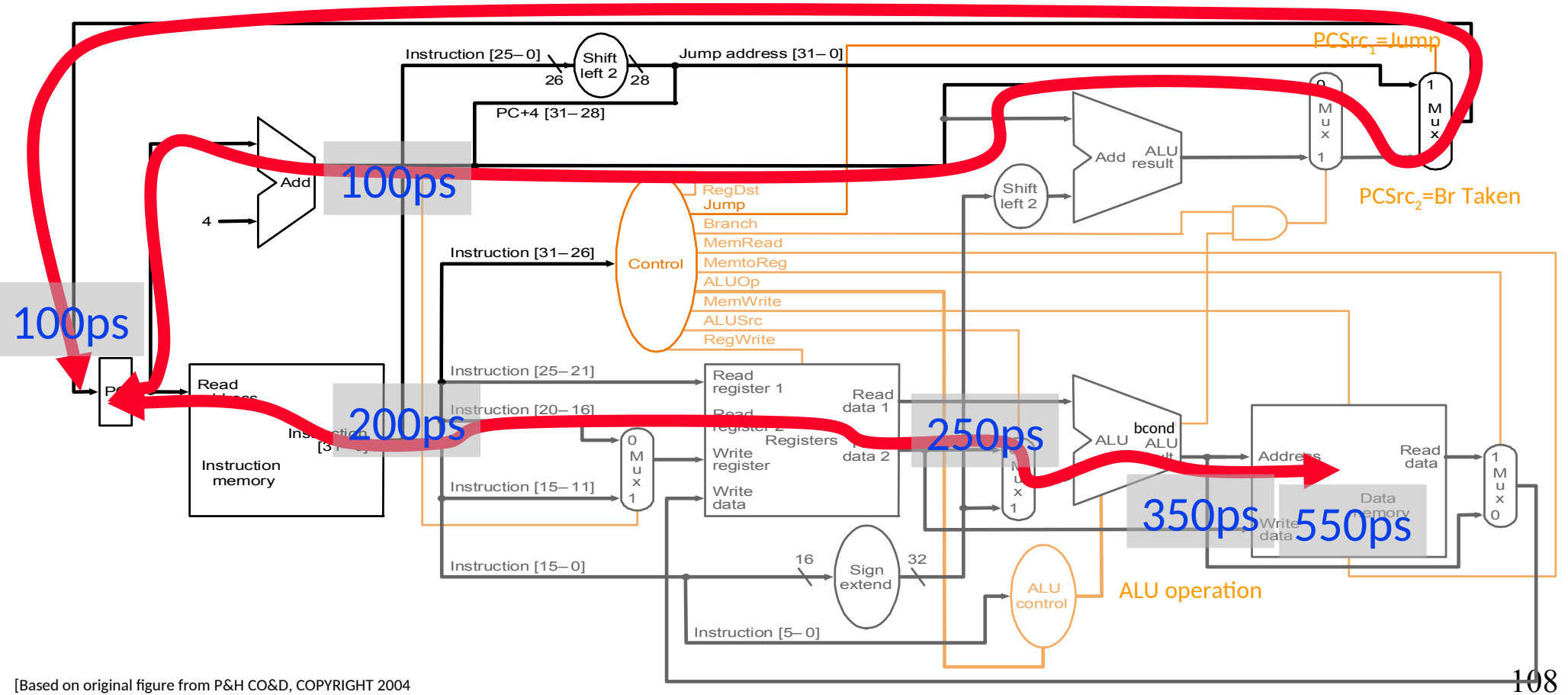
[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

LW



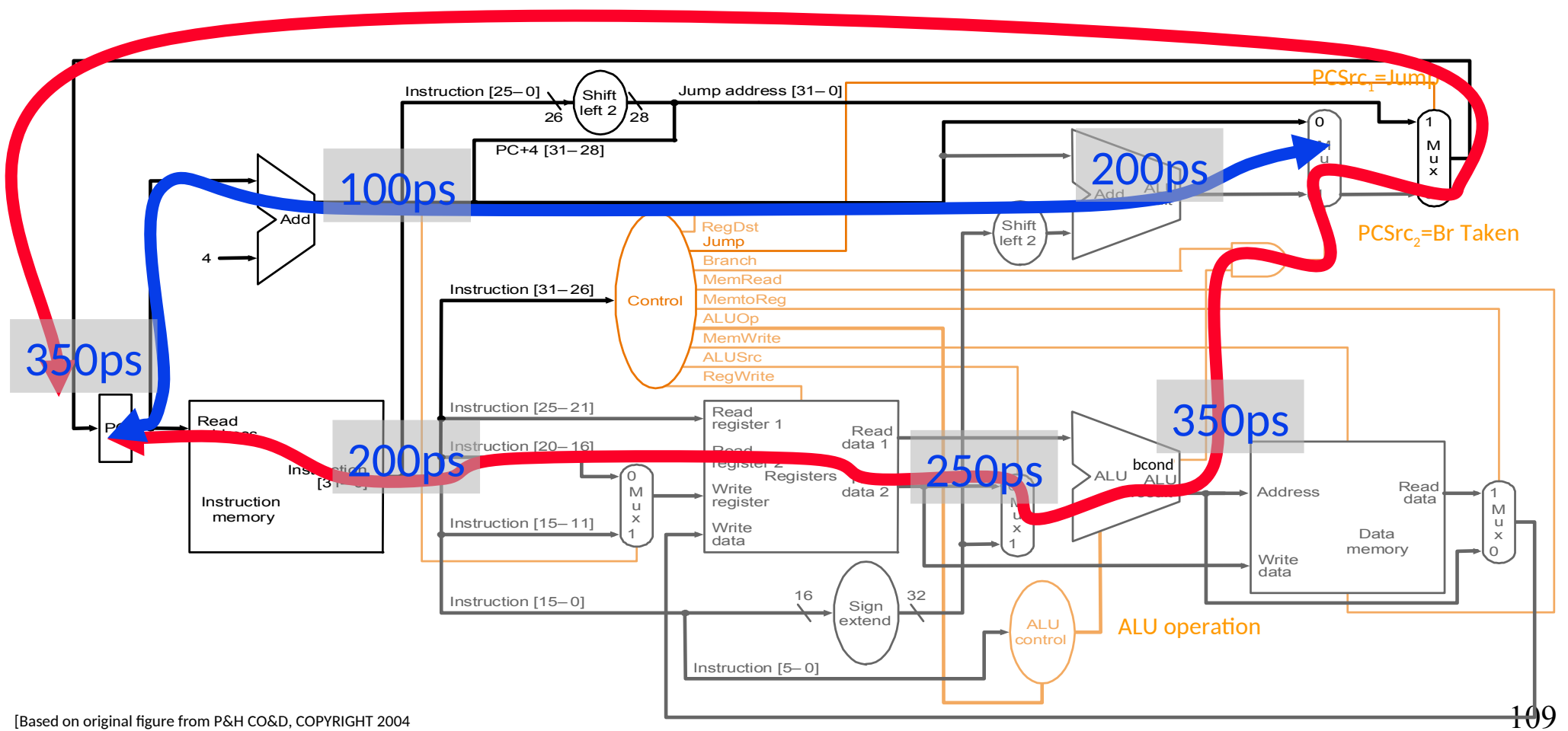
[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

SW



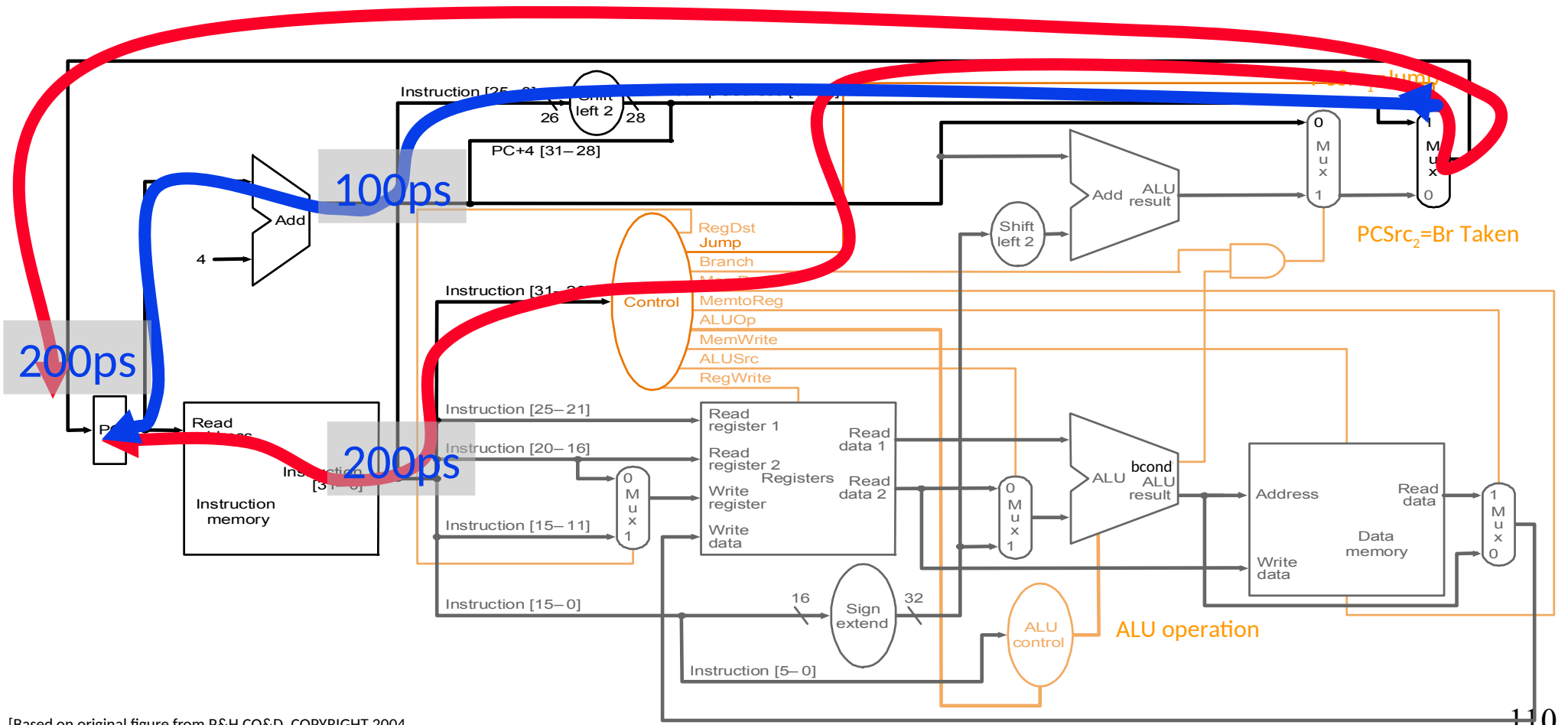
[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

Branch Taken



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

Jump



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

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?

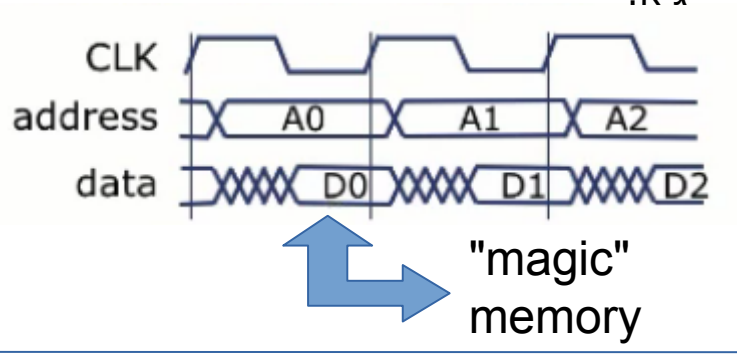
- Food for thought for you:
  - Can control logic be on the critical path? (*)
  - A note on CDC 5600: control store access too long…(**)

(*) 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. הם קיצרו את ה- pipeline כדי להתגבר על בעיית זמן הגישה הארוך

What is the Slowest Instruction to Process?

- 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?

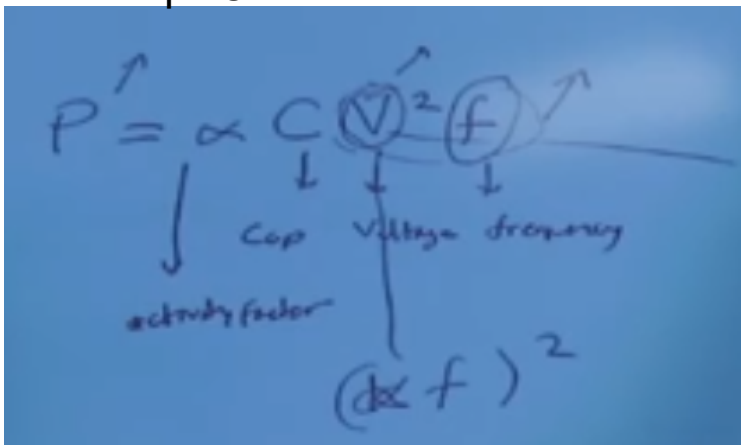
Single Cycle µArch: Complexity

- 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 do
  - **Common case vs. uncommon case**

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

Single-Cycle Design vs. Design Principles

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

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

Aside: System Design Principles

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

- Remember: "principled design" from our first lecture(*)
  - Frank Lloyd Wright: "architecture […] based upon principle, and not upon precedent"

(*) השנה דילגנו על פרק לויד רייט.... ראו השקף הבא

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;

Aside: From Lecture 1

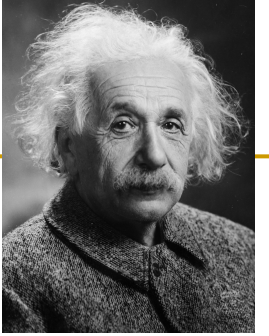
- "architecture […] based upon principle, and not upon precedent"

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

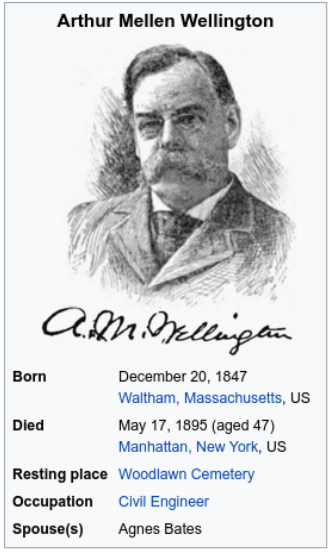


- Keep it simple

- "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"



Multi-Cycle Microarchitectures