CSED311 Computer Architecture – Lecture 5

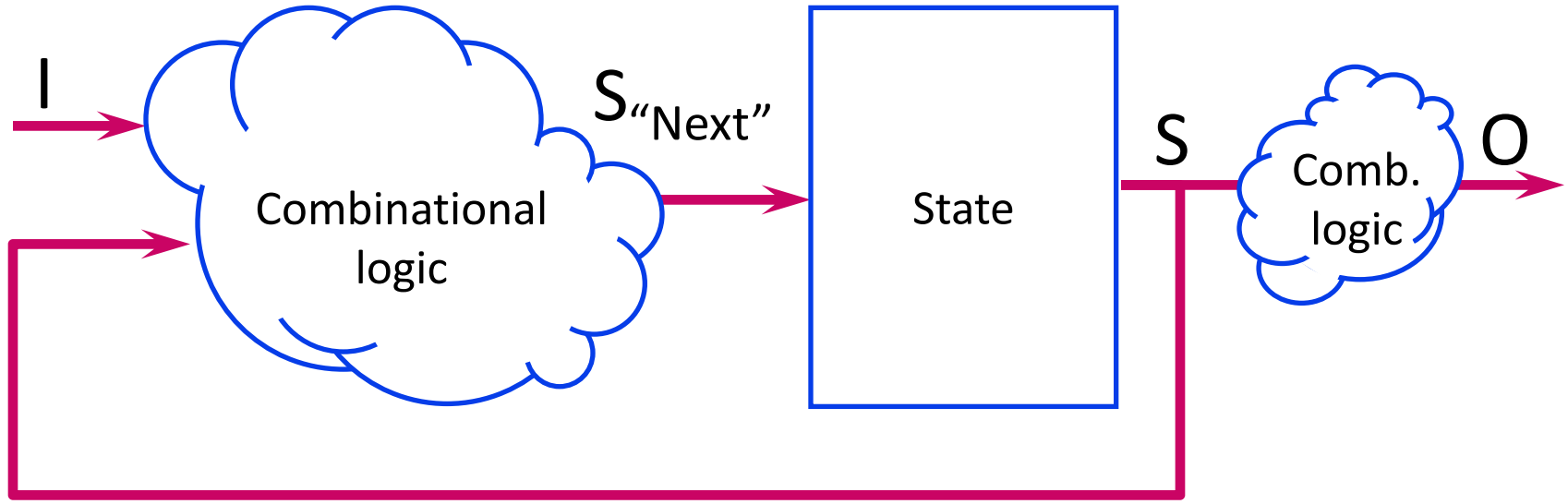
Single-Cycle CPU

Eunhyeok Park

Department of Computer Science and Engineering
POSTECH

Disclaimer: Slides developed in part by Profs. Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, and Wenisch @ Carnegie Mellon University, University of Michigan, Purdue University, University of Pennsylvania, University of Wisconsin and POSTECH.

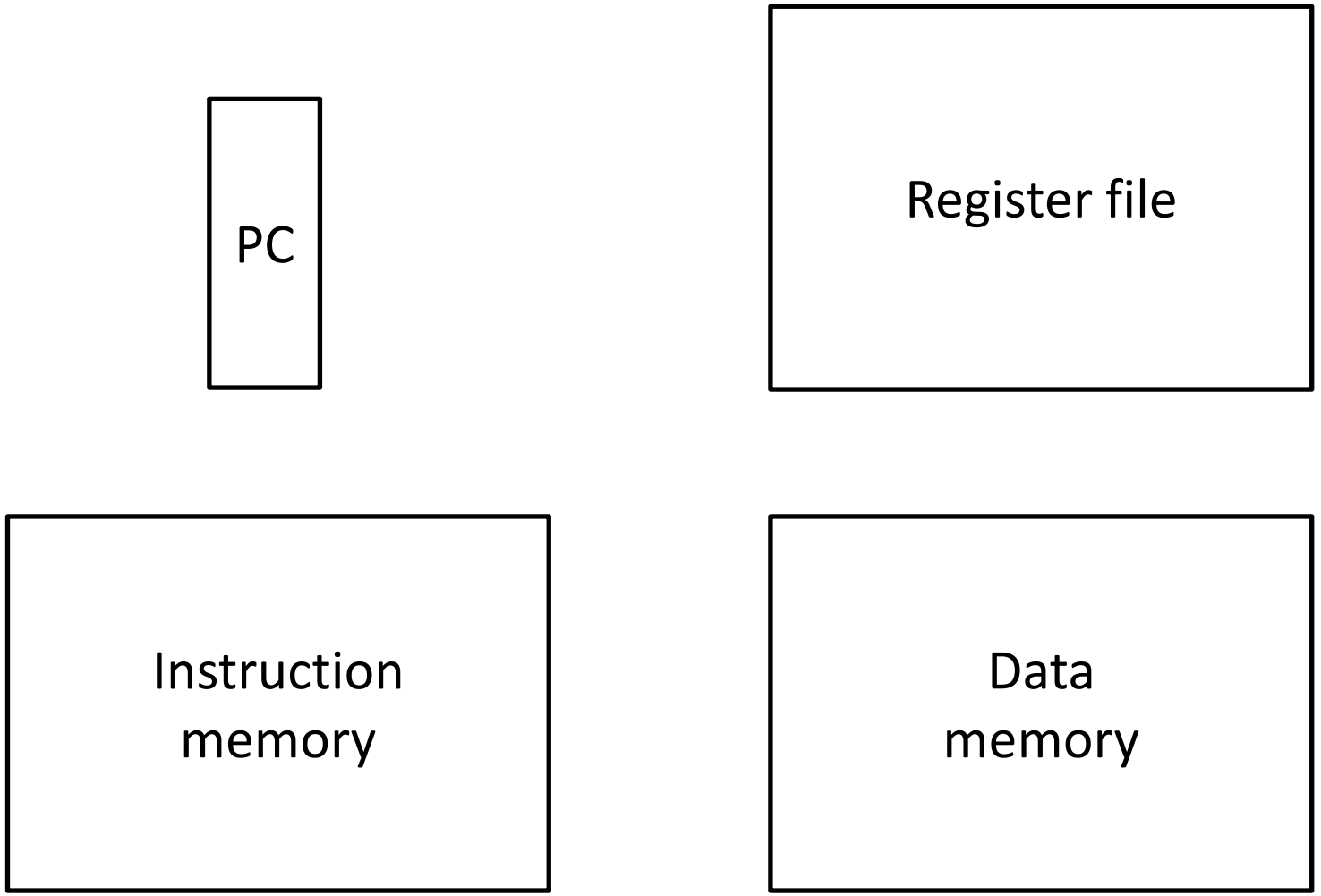
CPU: Instruction Processing FSM



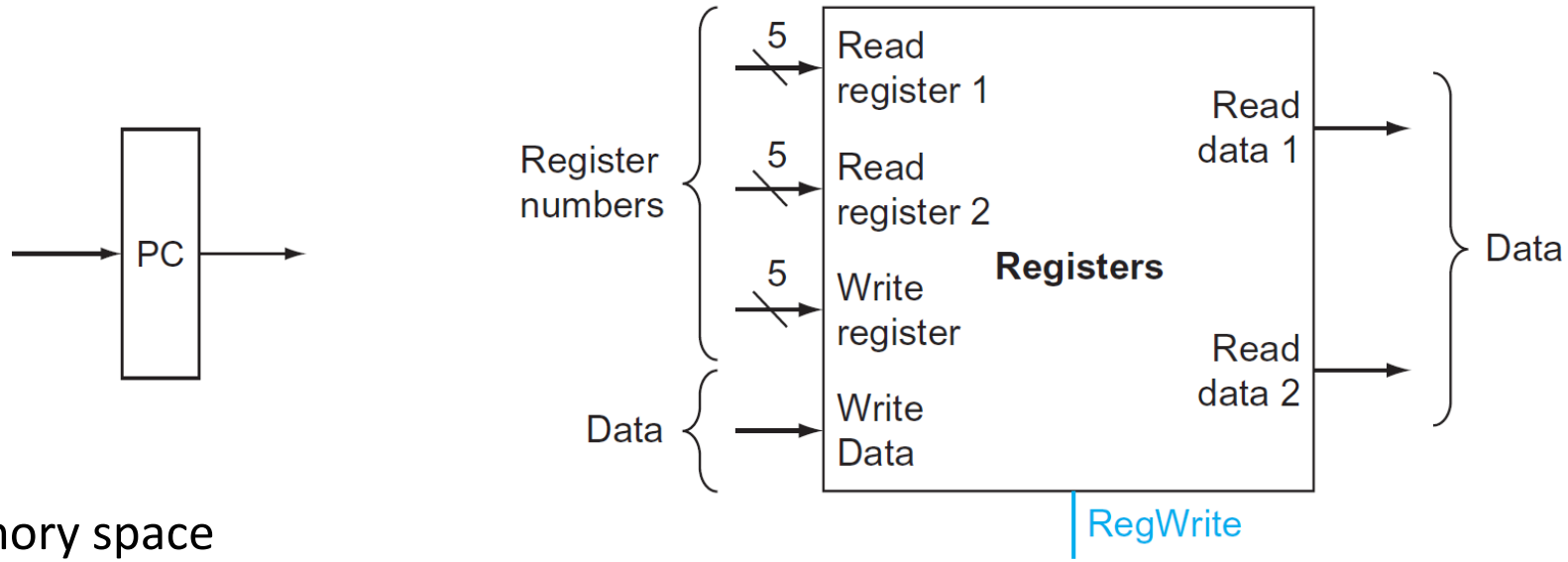
- An ISA describes an abstract finite-state machine (FSM)
  - State = program visible state
  - Next-state logic = instruction execution
- Nice ISAs have atomic instruction semantics
  - One state transition per instruction in abstract FSM
- Implementation of FSMs can vary

Programmer-Visible (Architectural) State

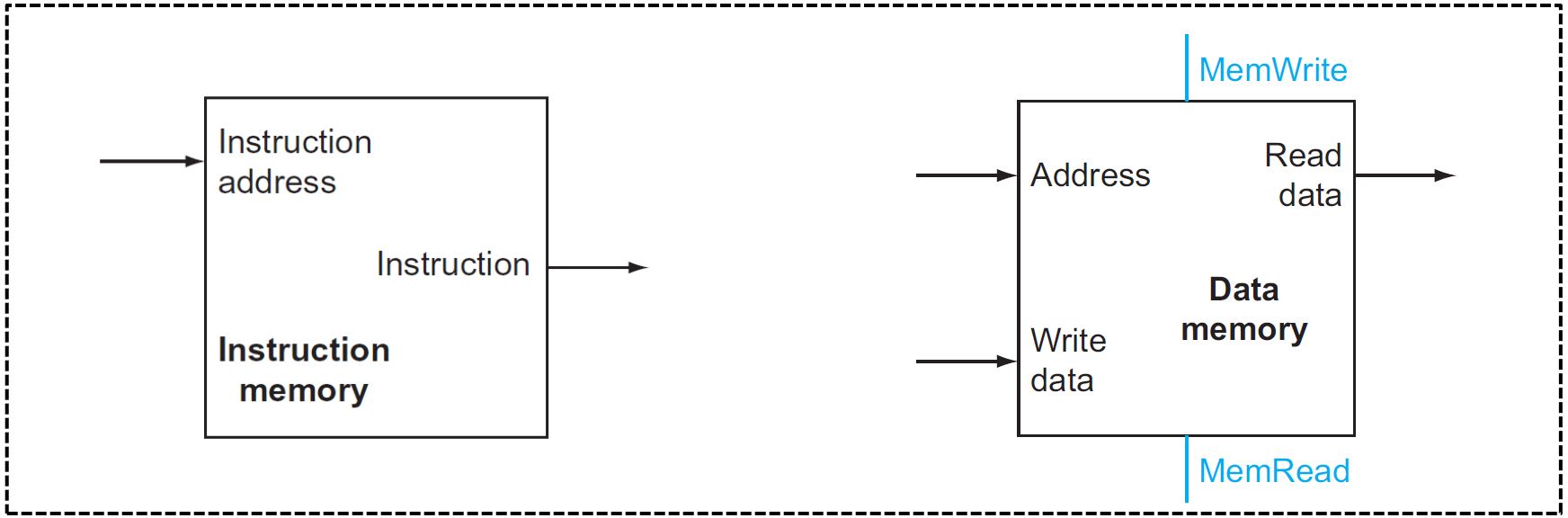
- What are the inputs and outputs for each component?



Programmer-Visible (Architectural) State



Single memory space



"Magic" Register File and Memory

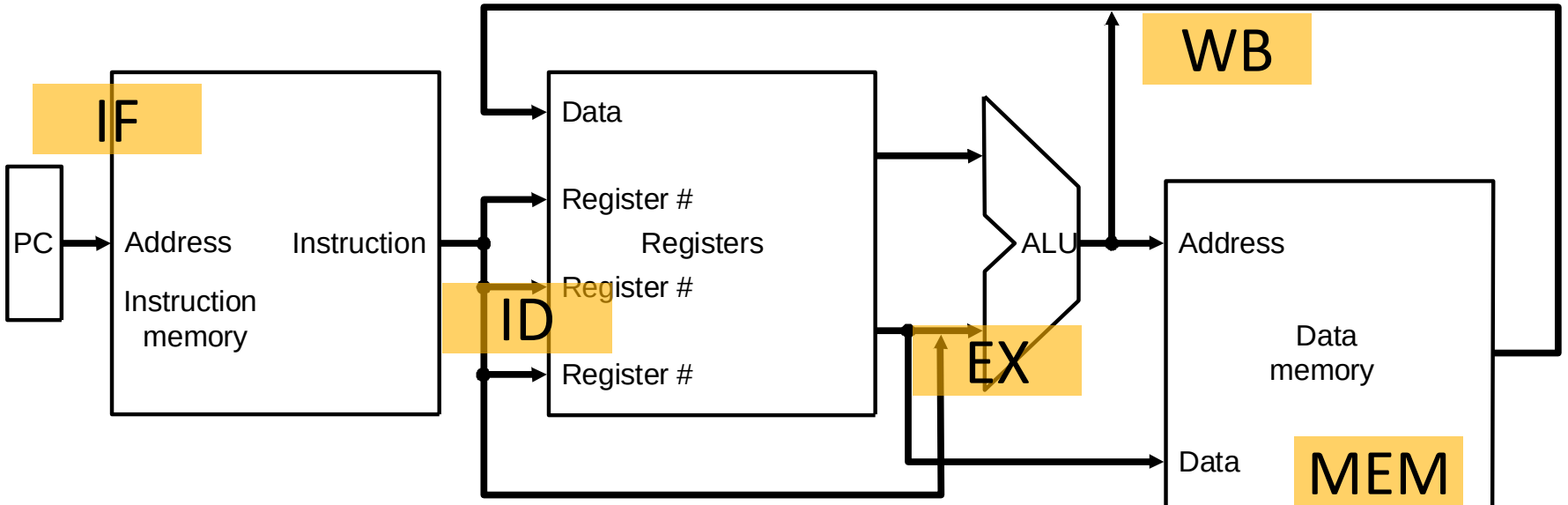
- Combinational read
  - The output of the read data port is a combinational function of the register file contents and the corresponding read select port
  - Similarly, instruction/data memory work like combinational logic

- Synchronous write
  - The selected register (or memory location) is updated **only** on the positive-edge clock transition when write enable is asserted
    Cannot affect read output in between clock edges

- The instruction memory also needs to be written to when the program is loaded, but let's consider it read-only for simplicity for now
- We will learn about more realistic design of memory hierarchy later

Instruction Processing

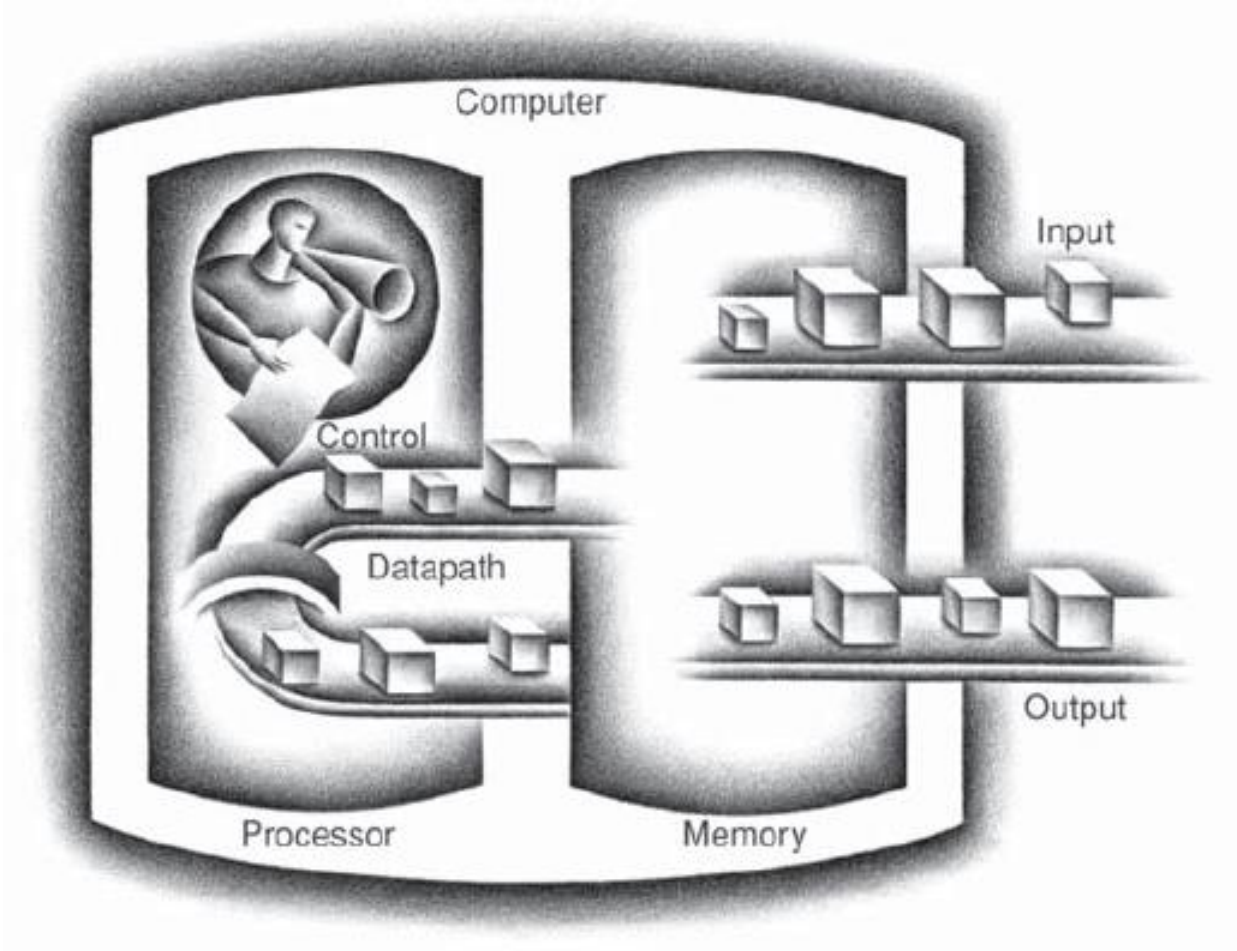
- 5 generic steps
  - **IF**: Instruction fetch
  - **ID**: Instruction decode and operand fetch
  - **EX**: ALU/execute
  - **MEM**: (Data) Memory access (only for load & store instructions)
  - **WB**: Write-back



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

Contents

- Datapath for ALU instructions
- Datapath for LD/ST instructions
- Datapath for control flow instructions
- Control



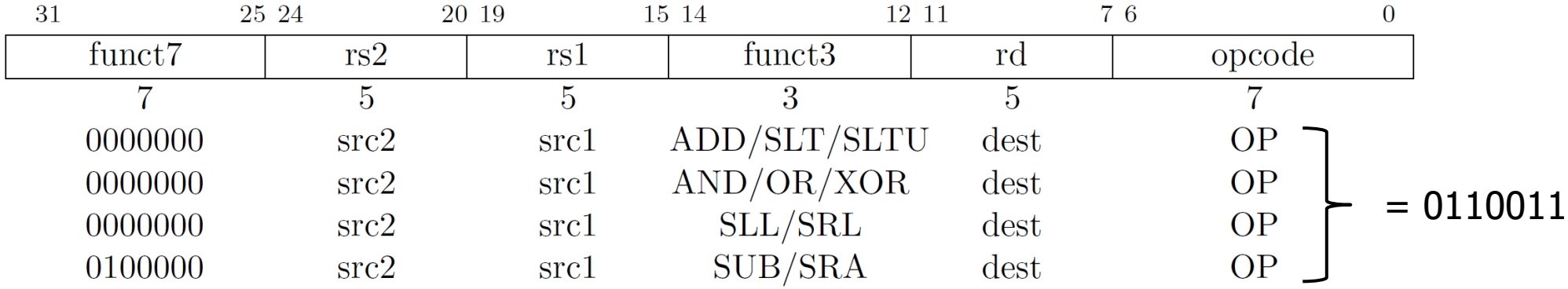
Today's lecture is about *microarchitecture*!

The microarchitecture *implements* the RISC-V *architecture*

Single-Cycle Datapath for Arithmetic and Logical Instructions

Reminder: R-Type Instruction

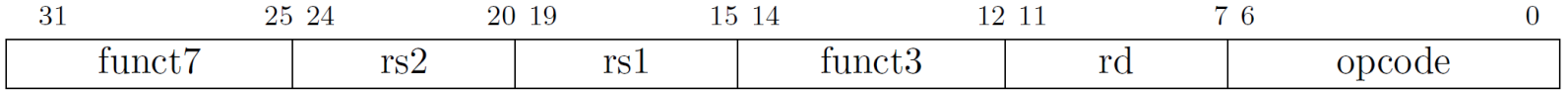
- Assembly example: ADD `rd, rs1, rs2`
- Encoding:



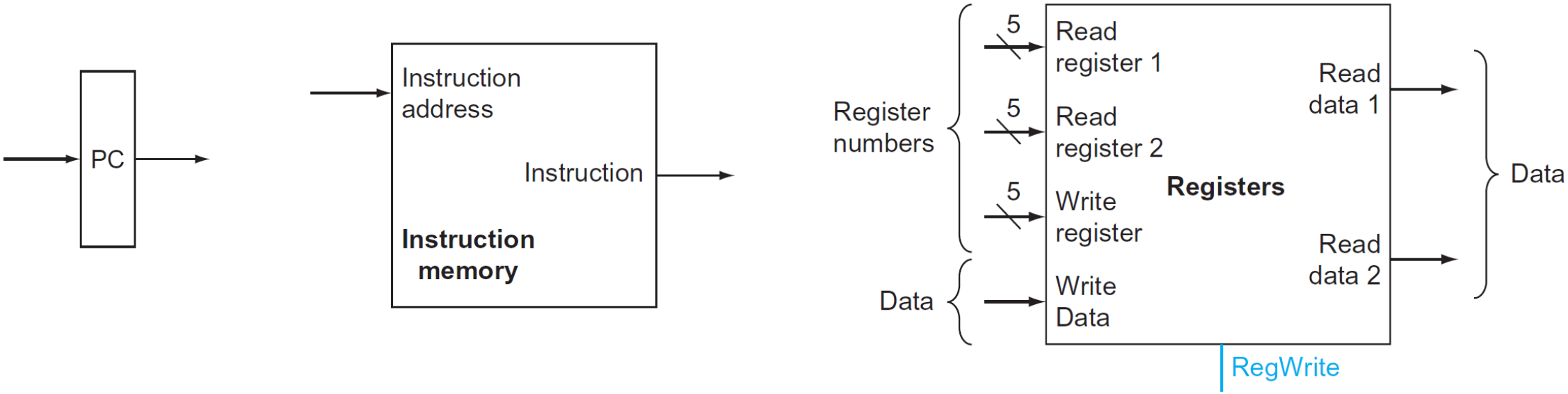
31–25	24–20	19–15	14–12	11–7	6–0	
funct7	rs2	rs1	funct3	rd	opcode	
7	5	5	3	5	7	
0000000	src2	src1	ADD/SLT/SLTU	dest	OP	= 0110011
0000000	src2	src1	AND/OR/XOR	dest	OP	
0000000	src2	src1	SLL/SRL	dest	OP	
0100000	src2	src1	SUB/SRA	dest	OP	

- Semantics
  - GPR[rd] ← GPR[rs1] (OP) GPR[rs2] [(OP): `+, -, |, &, <<, >>, …`]
  - PC ← PC + 4
- Variations
  - Arithmetic: ADD, SUB
  - Compare (Set if Less Than): SLT (signed), SLTU (unsigned) ← GPR[rd] ← 1 if GPR[rs1] < GPR[rs2]; GPR[rd] ← 0 otherwise
  - Logical: AND, OR, XOR
  - Shift: SLL (left), SRL (right-logical), SRA (right-arithmetic)
- Exception: none (ignore carry and overflow) – see page 13 of the ISA manual

R-Type Instruction Datapath

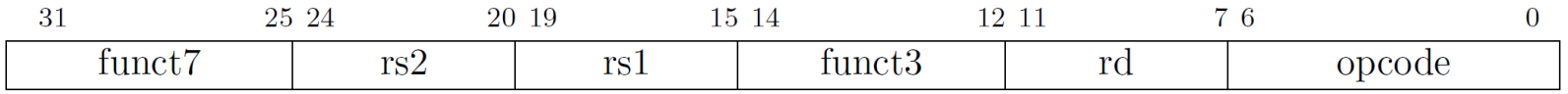


- Semantics (ADD)
  - GPR[rd] ← GPR[rs1] + GPR[rs2]
  - PC ← PC + 4

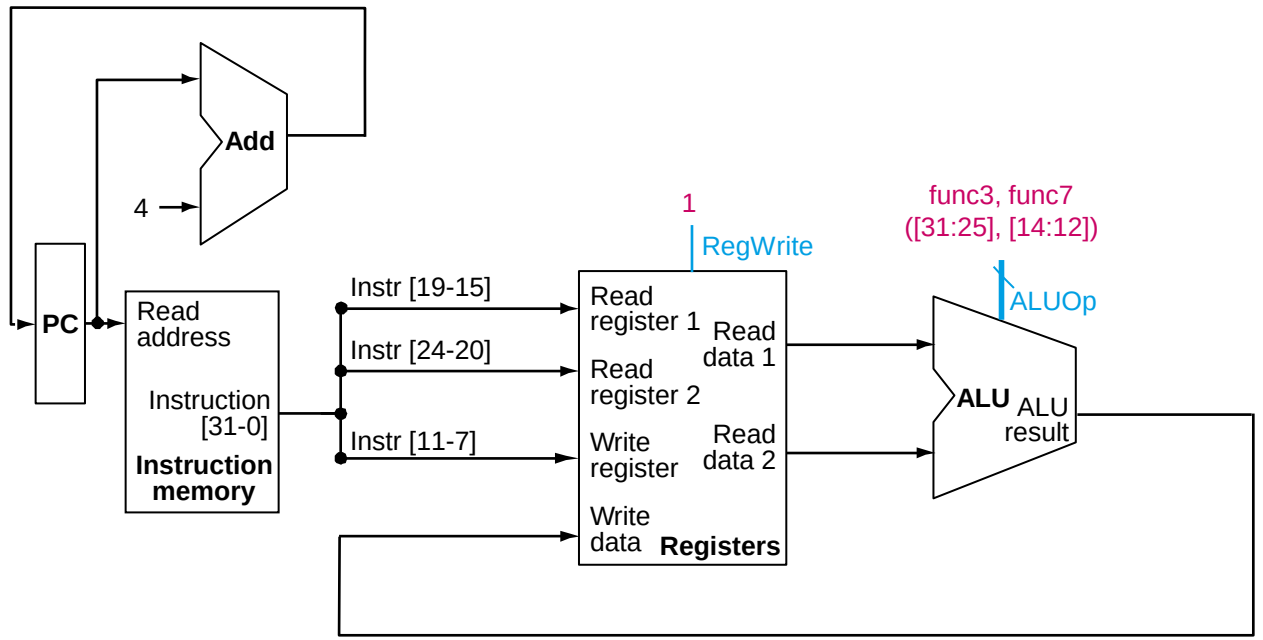


Frist, we are going to build the datapath and identify the control signals that need to be generated.
Then, we can work on the control signals at the end.

R-Type Instruction Datapath

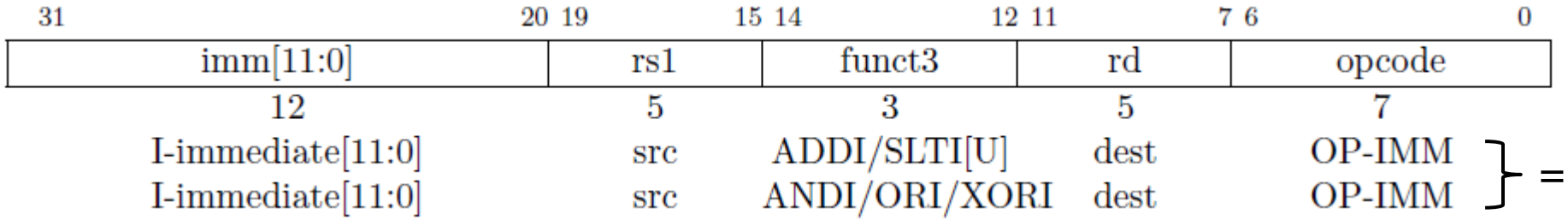


- Semantics (ADD)
  - GPR[rd] ← GPR[rs1] + GPR[rs2]
  - PC ← PC + 4



Reminder: I-Type ALU Instruction

- Assembly example: `ADDI rd, rs1, imm`$_{12}$
- Encoding:

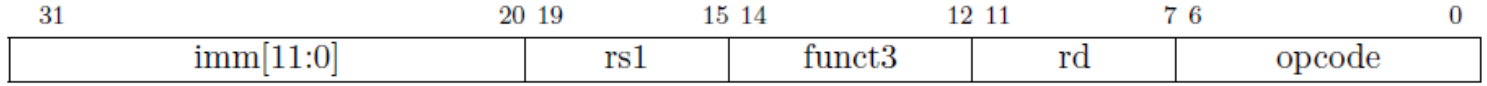


31 — 20	19 — 15	14 — 12	11 — 7	6 — 0	
imm[11:0]	rs1	funct3	rd	opcode	
12	5	3	5	7	
I-immediate[11:0]	src	ADDI/SLTI[U]	dest	OP-IMM	} = 0010011
I-immediate[11:0]	src	ANDI/ORI/XORI	dest	OP-IMM	

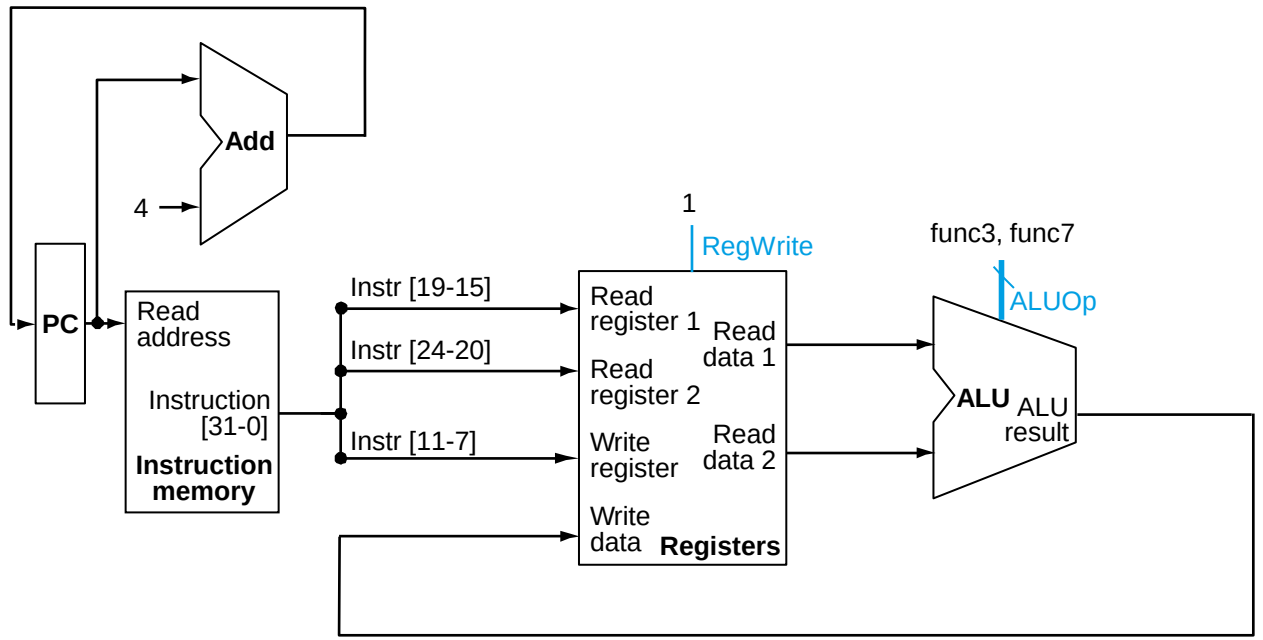
- Semantics
  - GPR[rd] ← GPR[rs1] (OP) sign-extend (immediate) [(OP): `+`, `|`, `&`, …]
  - PC ← PC + 4
- Variations:
  - Arithmetic: ADDI, ~~SUBI~~ – why no SUBI?
  - Logical: ANDI, ORI, XORI
  - Compare (Set if Less Than Immediate): SLTI (signed), SLTIU (unsigned)
- Exception: none (ignore carry and overflow)

I-Type ALU Instruction Datapath

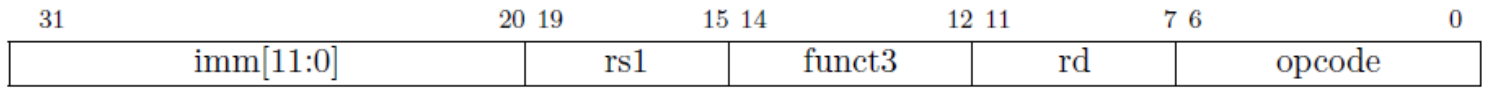
- How should this change?



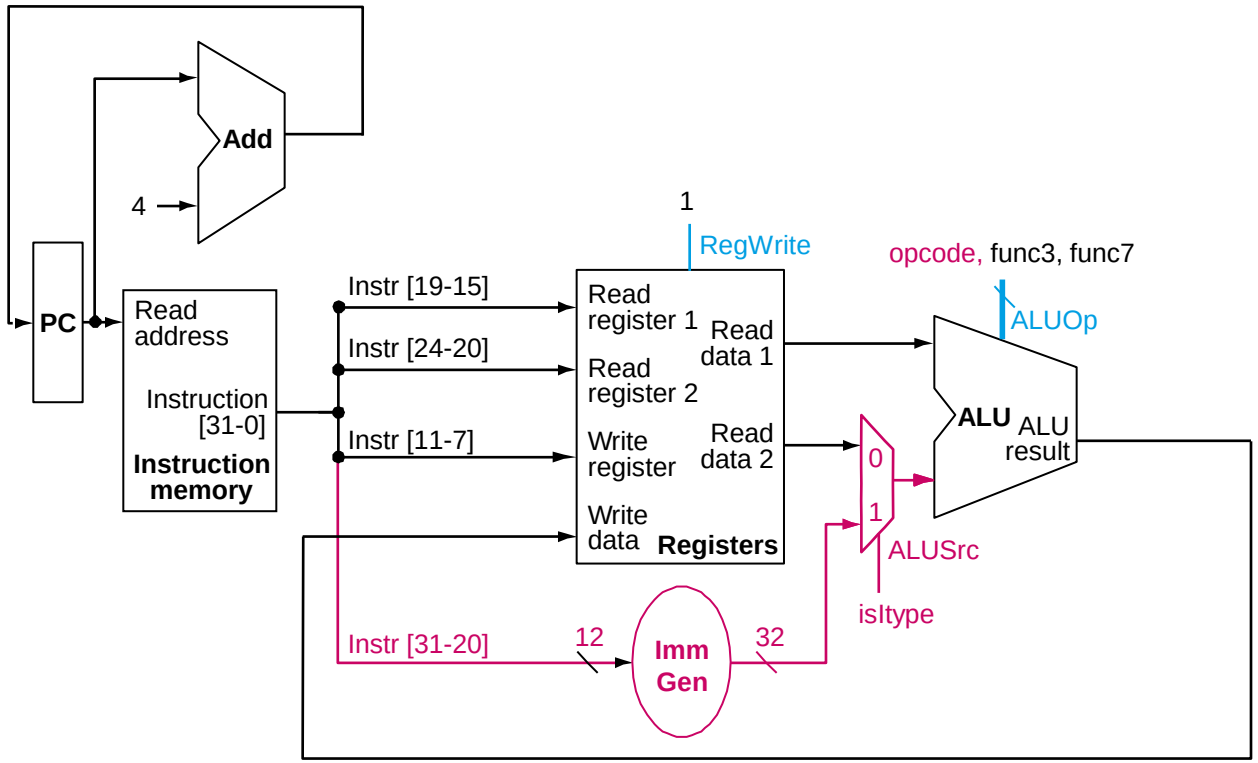
- Semantics
  - GPR[rd] ← GPR[rs1] + sign-extend(imm)
  - PC ← PC + 4

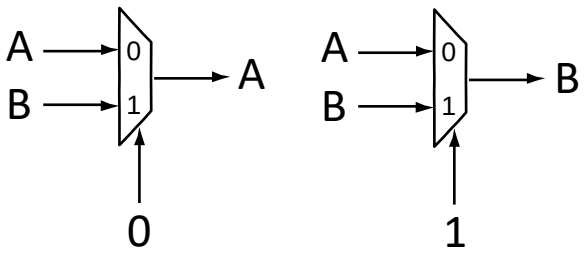


I-Type ALU Instruction Datapath



- Semantics
  - GPR[rd] ← GPR[rs1] + sign-extend(imm)
  - PC ← PC + 4



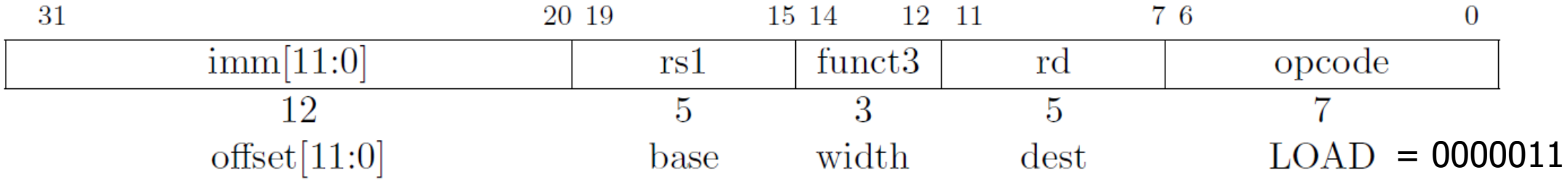


**Based on figures from [P&H CO&D, COPYRIGHT 2017 Elsevier. ALL RIGHTS RESERVED.]

Single-Cycle Datapath for Data Movement Instructions

Reminder: Load Instructions (I-type)

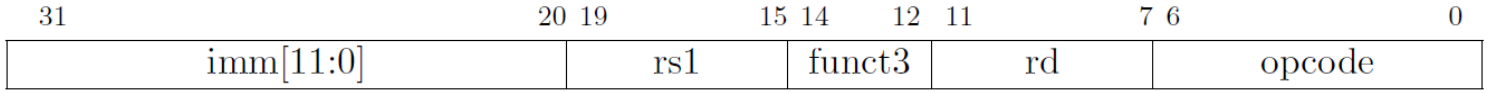
- Assembly example: `LW rd, offset`$_{12}$`(base)`
  - imm
  - rs1
- Encoding:



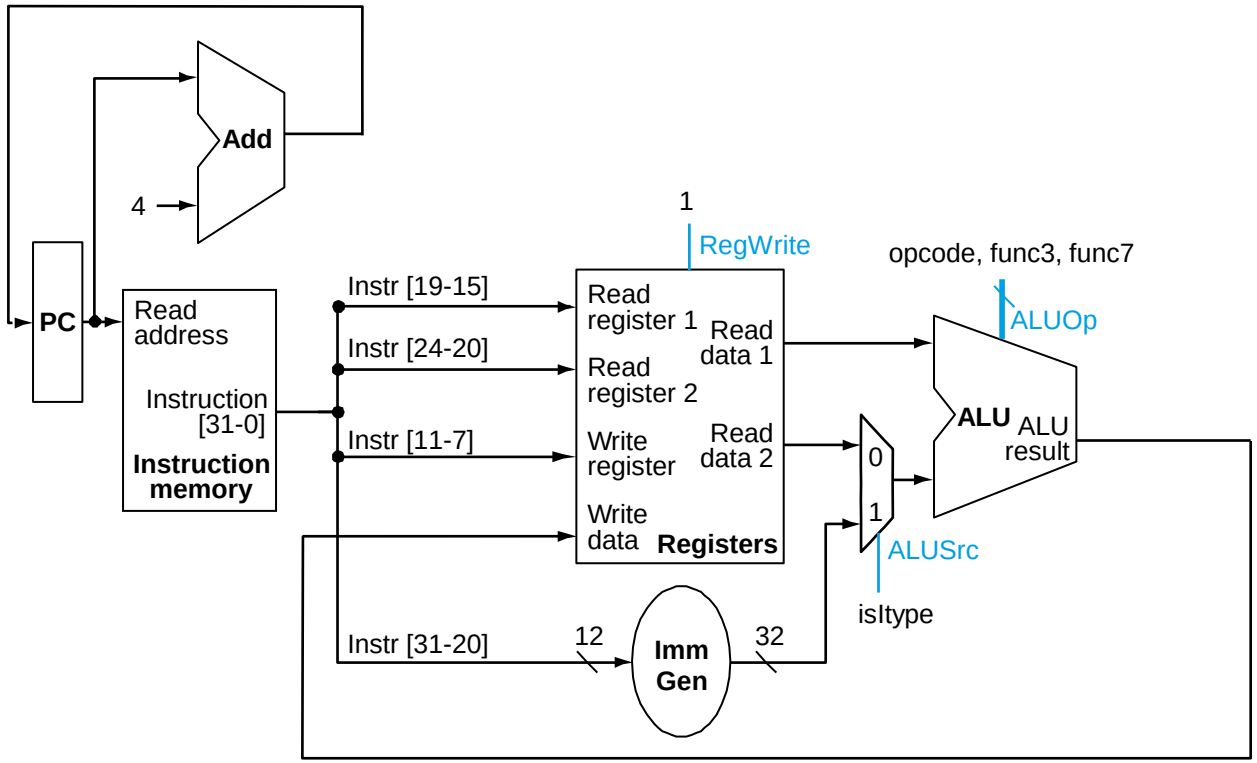
- Semantics:
  - byte_address$_{32}$ = sign-extend(offset$_{12}$) + GPR[base]
  - GPR[rd] ← MEM$_{32}$ [byte_address$_{32}$]
  - PC ← PC + 4
- Exceptions: will be discussed later
- Variations:
  - LW (word), LH (halfword), LB (byte): sign-extend
  - LHU, LBU: zero-extend

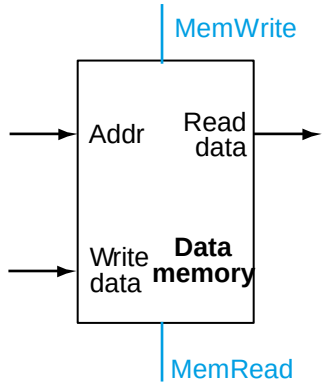
Load Datapath

- How should this change for load?

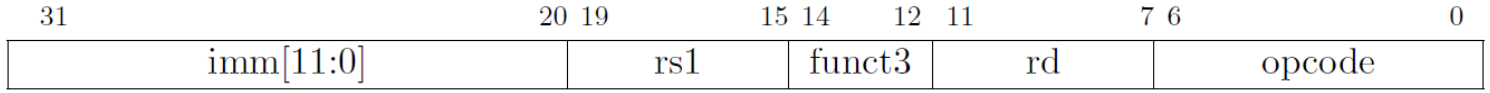


- Semantics:
  - $byte\_addr_{32}$ = sign-extend($offset_{12}$) + GPR[base]
  - GPR[rd] ← MEM_{32} [$byte\_addr_{32}$]
  - PC ← PC + 4

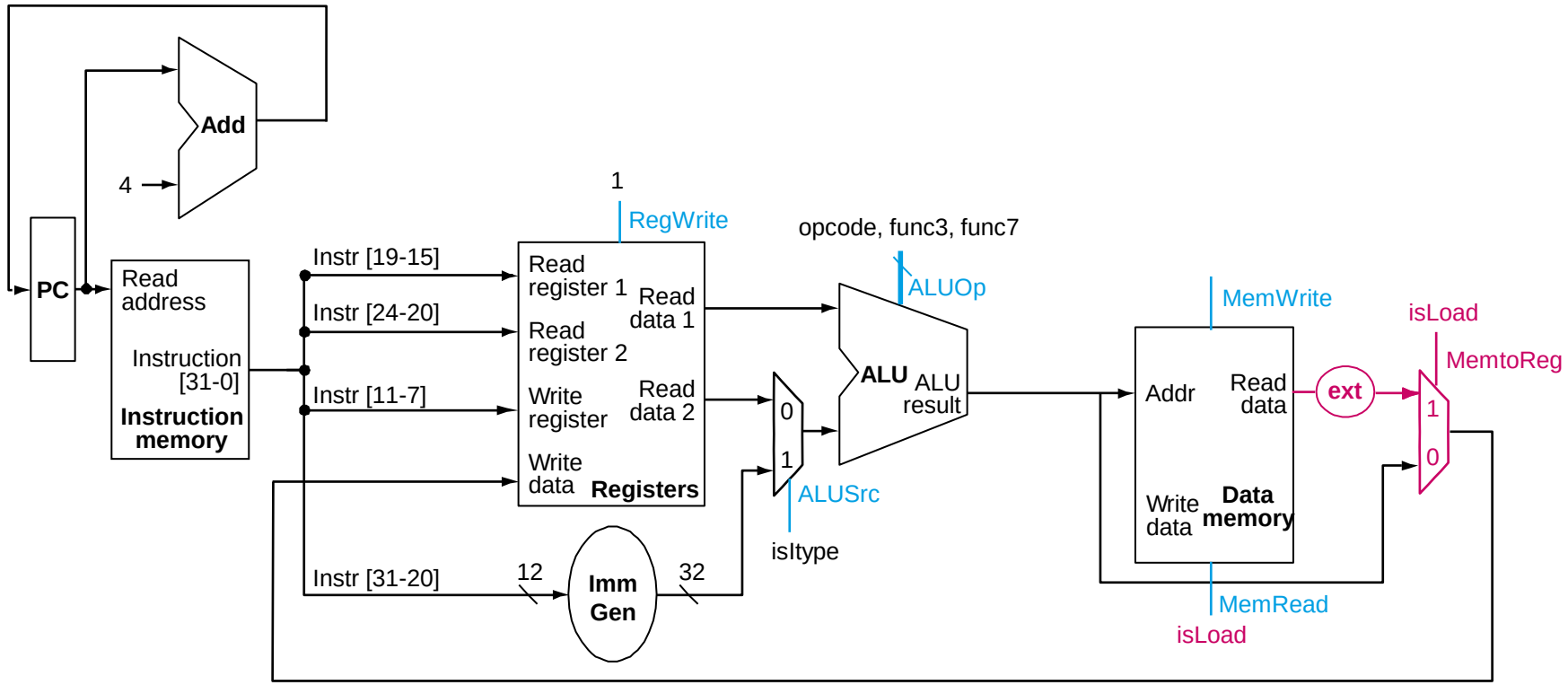




Load Datapath

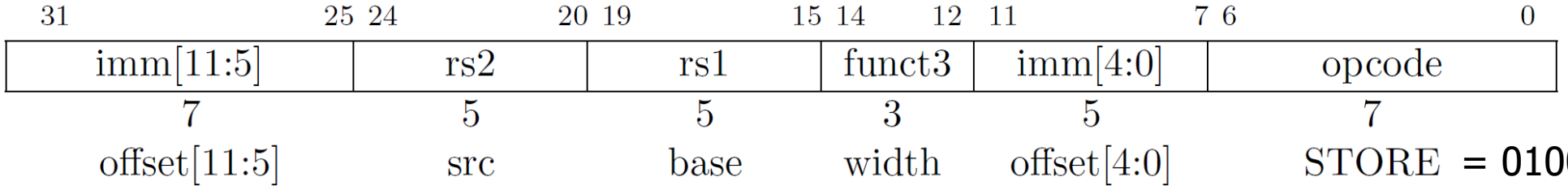


- Semantics:
  - $byte\_addr_{32} = sign\text{-}extend(offset_{12}) + GPR[base]$
  - $GPR[rd] \leftarrow MEM_{32}[byte\_addr_{32}]$
  - $PC \leftarrow PC + 4$



Reminder: Store Instructions (S-Type)

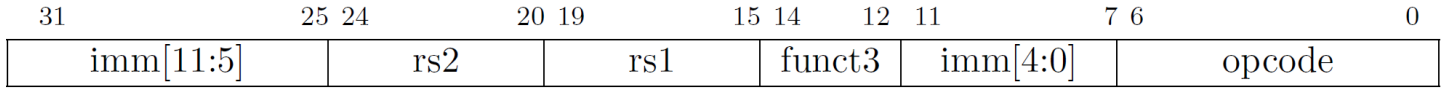
- Assembly example (Store Word): `SW rs2, offset`$_{12}$`(base)`
  - imm
  - rs1
- Encoding:



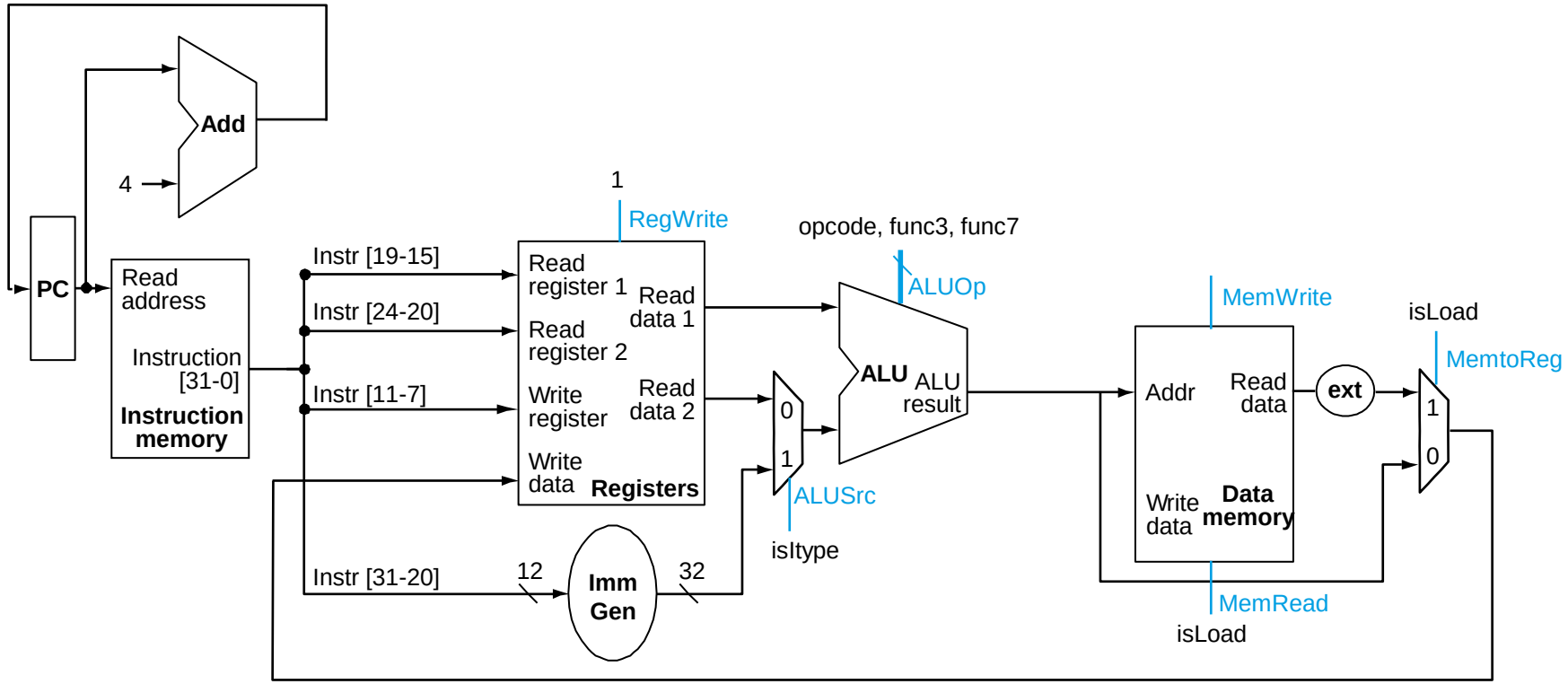
- Semantics:
  - byte_address$_{32}$ = sign-extend(offset$_{12}$) + GPR[base]
  - MEM$_{32}$ [byte_address$_{32}$] ← GPR[rs2]
  - PC ← PC + 4
- Exceptions: will be discussed later
- Variations:
  - SW (word), SH (halfword), SB (byte)
- Why split immediate?

Store Datapath

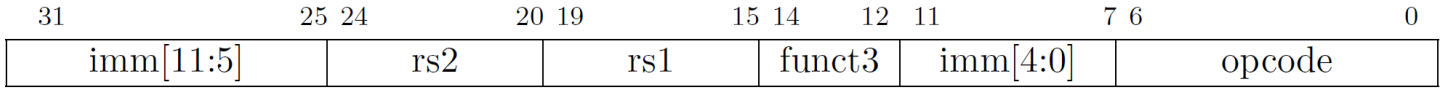
- How should this change for store?



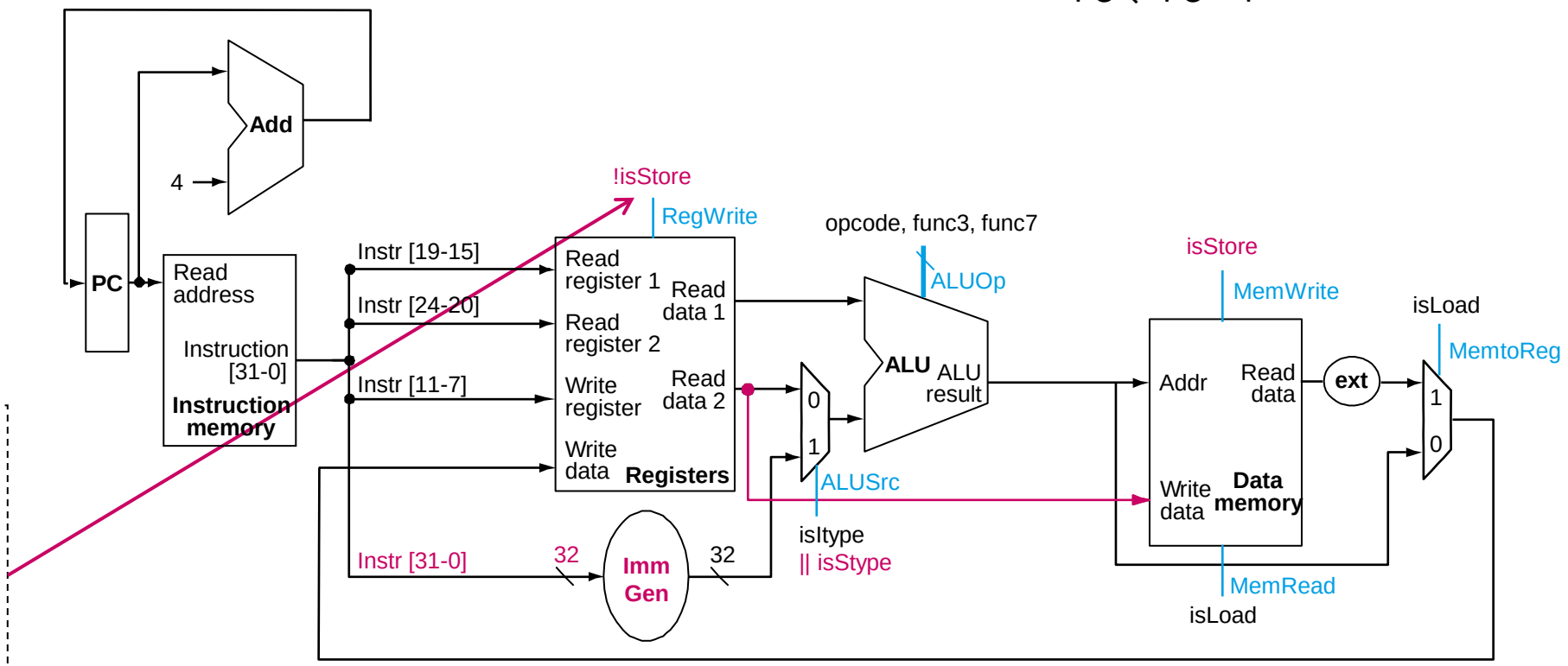
- Semantics:
  - $byte\_address_{32}$ = sign-extend($offset_{12}$) + GPR[base]
  - MEM_{32} [$byte\_address_{32}$] ← GPR[rs2]
  - PC ← PC + 4



Load/Store Datapath



- Semantics:
  - $byte\_address_{32}$ = sign-extend($offset_{12}$) + GPR[base]
  - MEM_{32} [$byte\_address_{32}$] ← GPR[rs2]
  - PC ← PC + 4

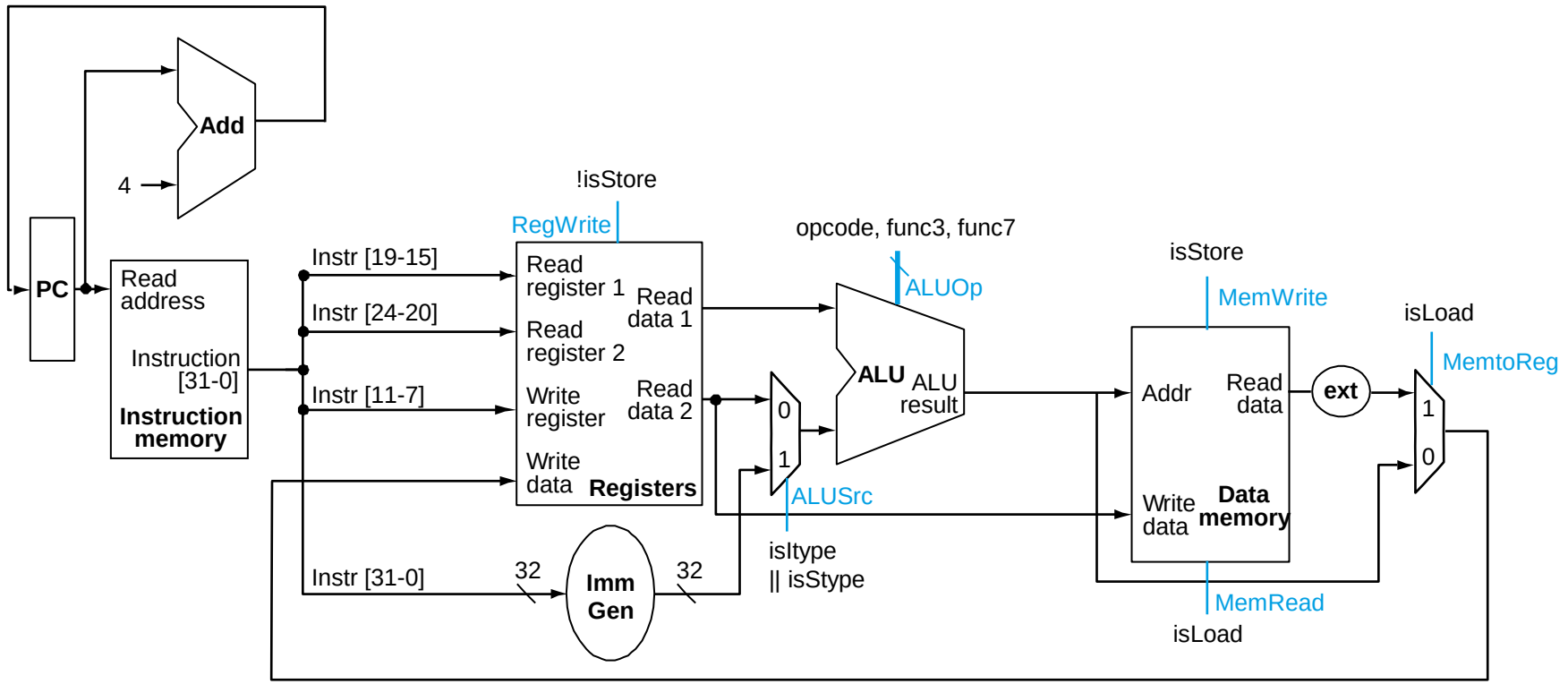


If STORE,
isStore = 1
→ RegWrite = 0

For others,
isStore = 0
→ RegWrite = 1

**Based on figures from [P&H CO&D, COPYRIGHT 2017 Elsevier. ALL RIGHTS RESERVED.]

Datapath for Non-Control Flow Instructions

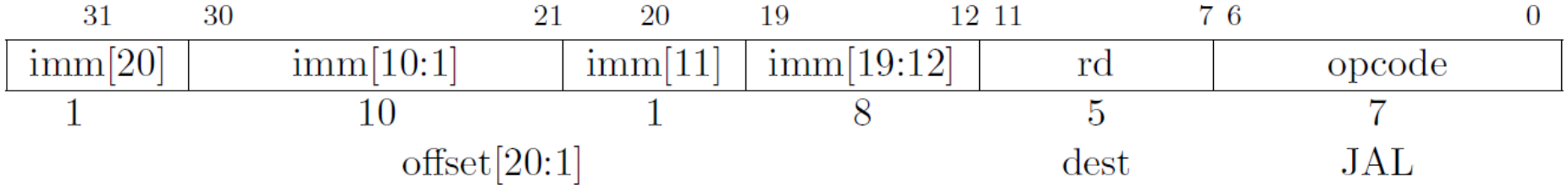


**Based on figures from [P&H CO&D, COPYRIGHT 2017 Elsevier. ALL RIGHTS RESERVED.]

Single-Cycle Datapath for Control Flow Instructions

Reminder: Jump Instruction (UJ-Type)

- Jump and Link assembly: `JAL rd, imm`$_{21}$
- Encoding:

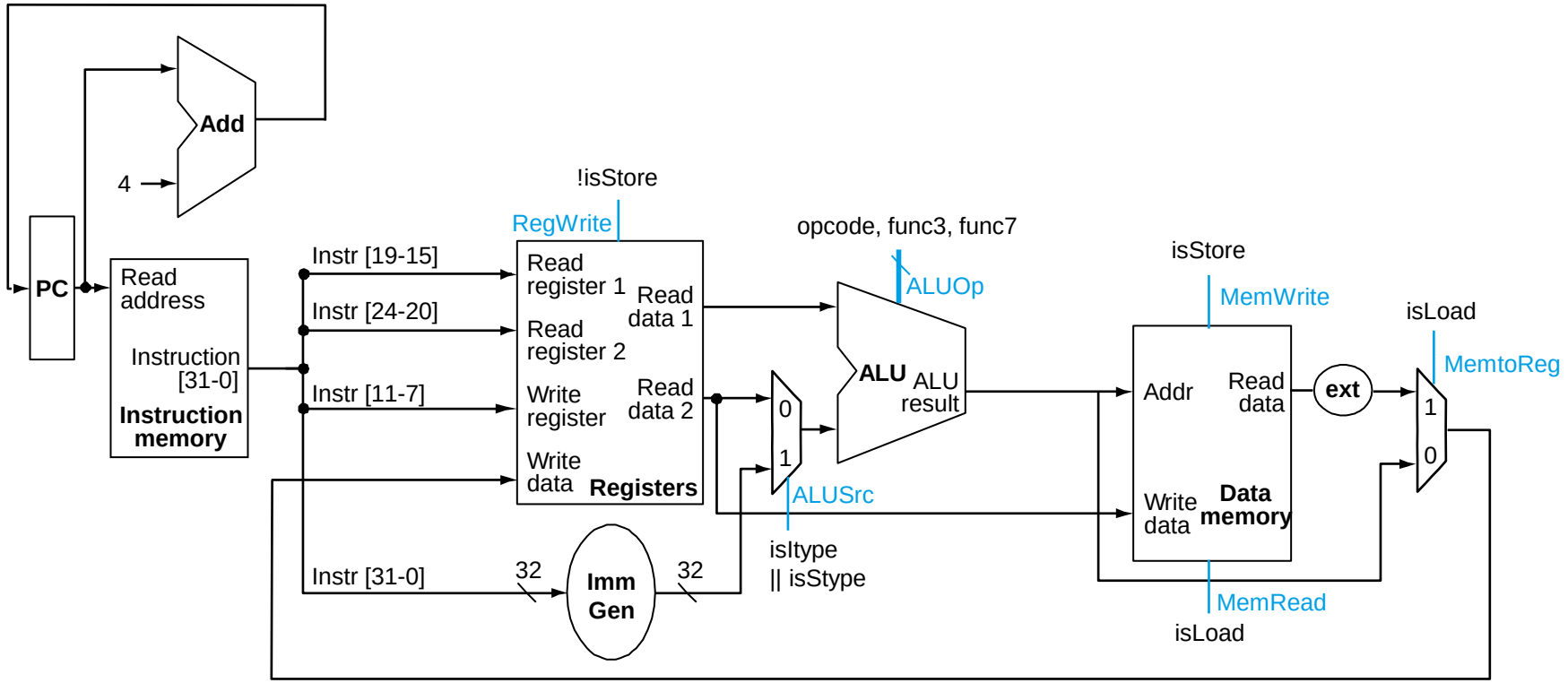


  - Note: imm[0] == 0 (only branch to even addresses)
- Semantics:
  - target = PC + sign-extend(imm$_{21}$) // PC-relative addressing
  - GPR[rd] ← PC + 4 // store return address in a register (cf. BEQ/BNE/…)
  - PC ← target // always jump to the same target
- Exception: misaligned target (4-byte)

Unconditional Jump Datapath (JAL)

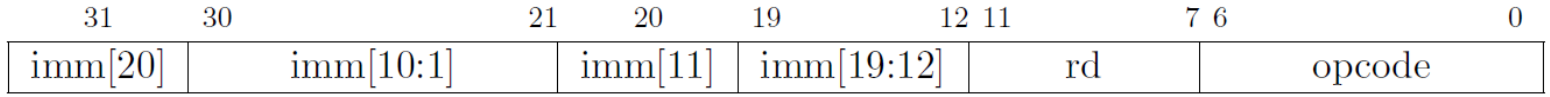
- How should this change for JAL?

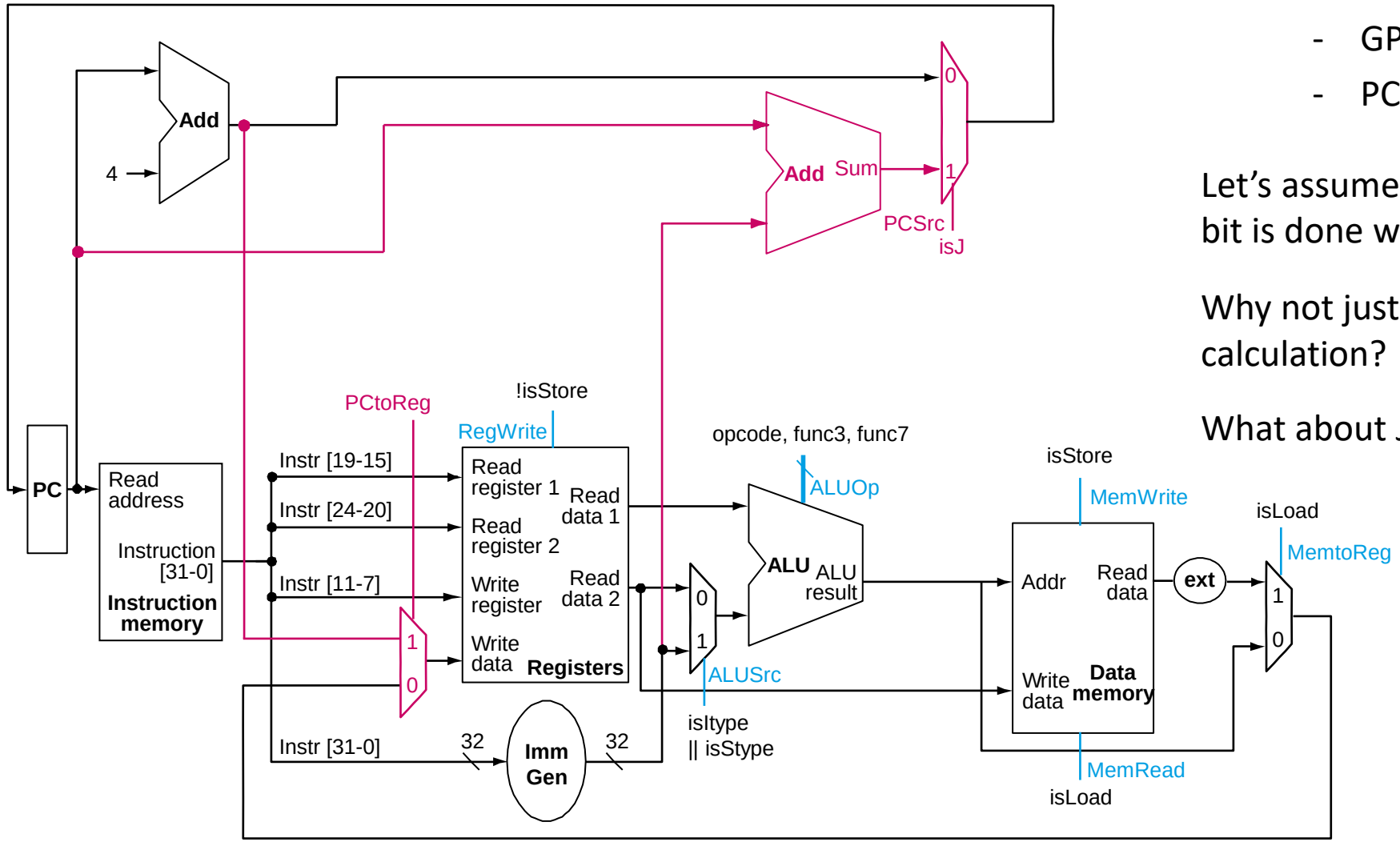
31	30 … 21	20	19 … 12	11 … 7	6 … 0
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode



- Semantics:
  - target = PC + sign-extend(imm_{21})
  - GPR[rd] ← PC + 4
  - PC ← target

Unconditional Jump Datapath (JAL)





- Semantics:
  - target = PC + sign-extend(imm_{21})
  - GPR[rd] ← PC + 4
  - PC ← target

Let's assume shifting immediate by one bit is done within "Imm. gen."

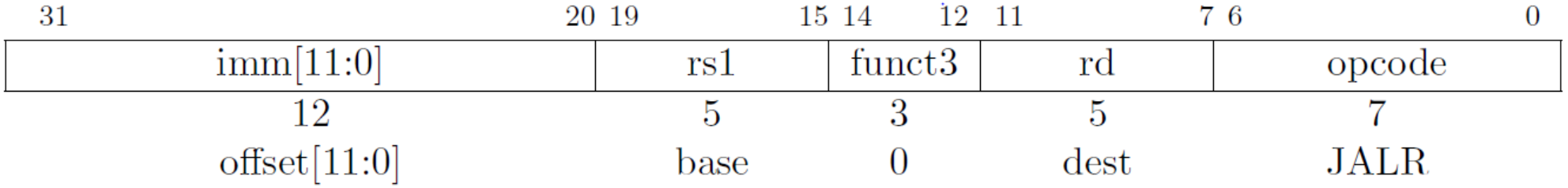
Why not just use existing ALU for target calculation?

What about JALR?

**Based on figures from [P&H CO&D, COPYRIGHT 2017 Elsevier. ALL RIGHTS RESERVED.]

Reminder: Jump Indirect Instruction (I-Type)

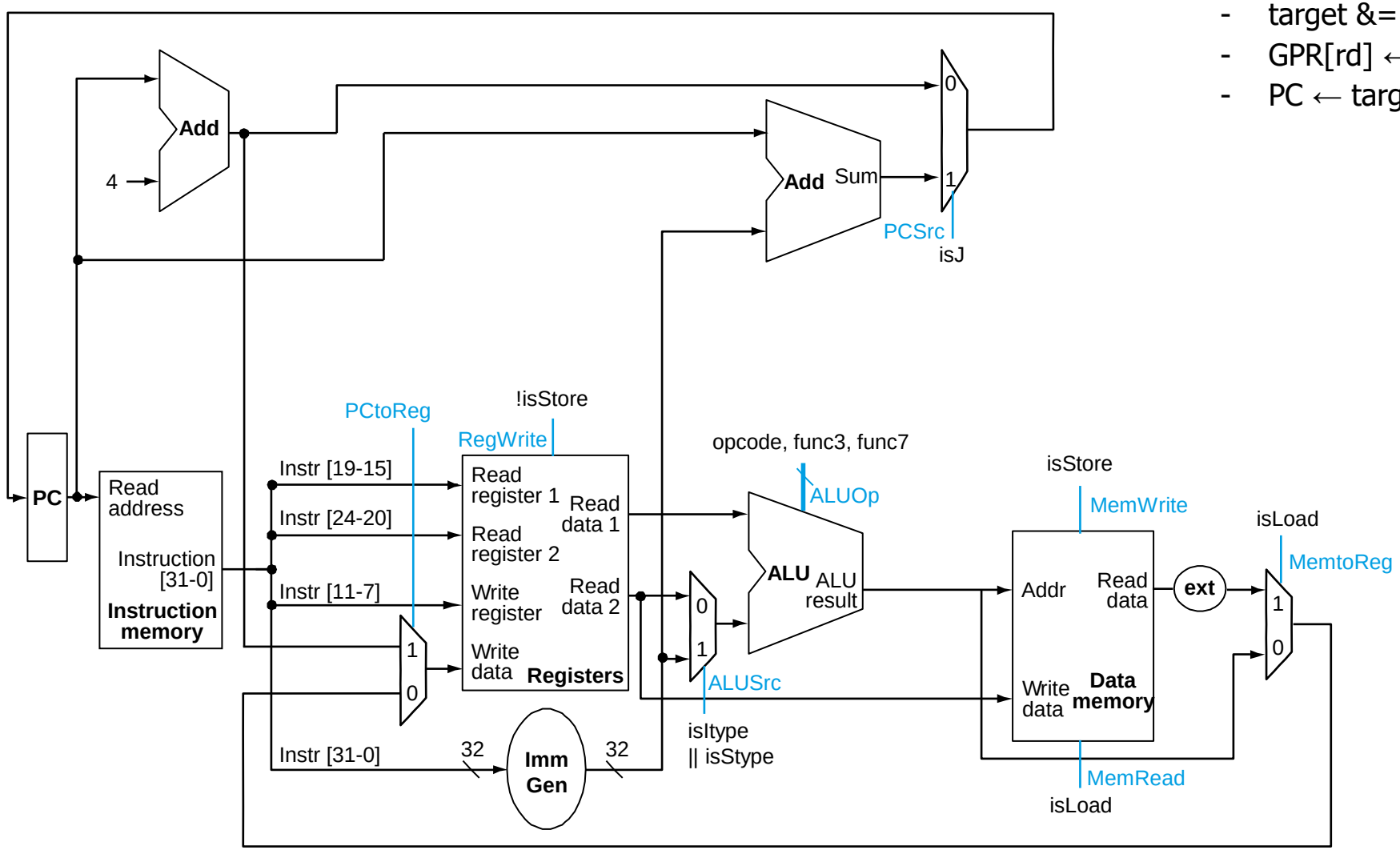
- Jump and Link Register assembly: `JALR rd, imm12(rs1)`
- Encoding:



- Semantics:
  - target = GPR[rs1] + sign-extend(imm_{12}) // can jump to different targets
  - target &= 0xFFFFFFFE // set the lowest bit to 0 (but let's ignore this for now)
  - GPR[rd] ← PC + 4
  - PC ← target
- Usages: - Procedure return: `jalr x0, 0(x1)` // return to GPR[x1]. x0 can't change
  - Computed jumps (e.g., case/switch statements)
- Exception: misaligned target (4-byte)

Unconditional Jump Datapath (JALR)

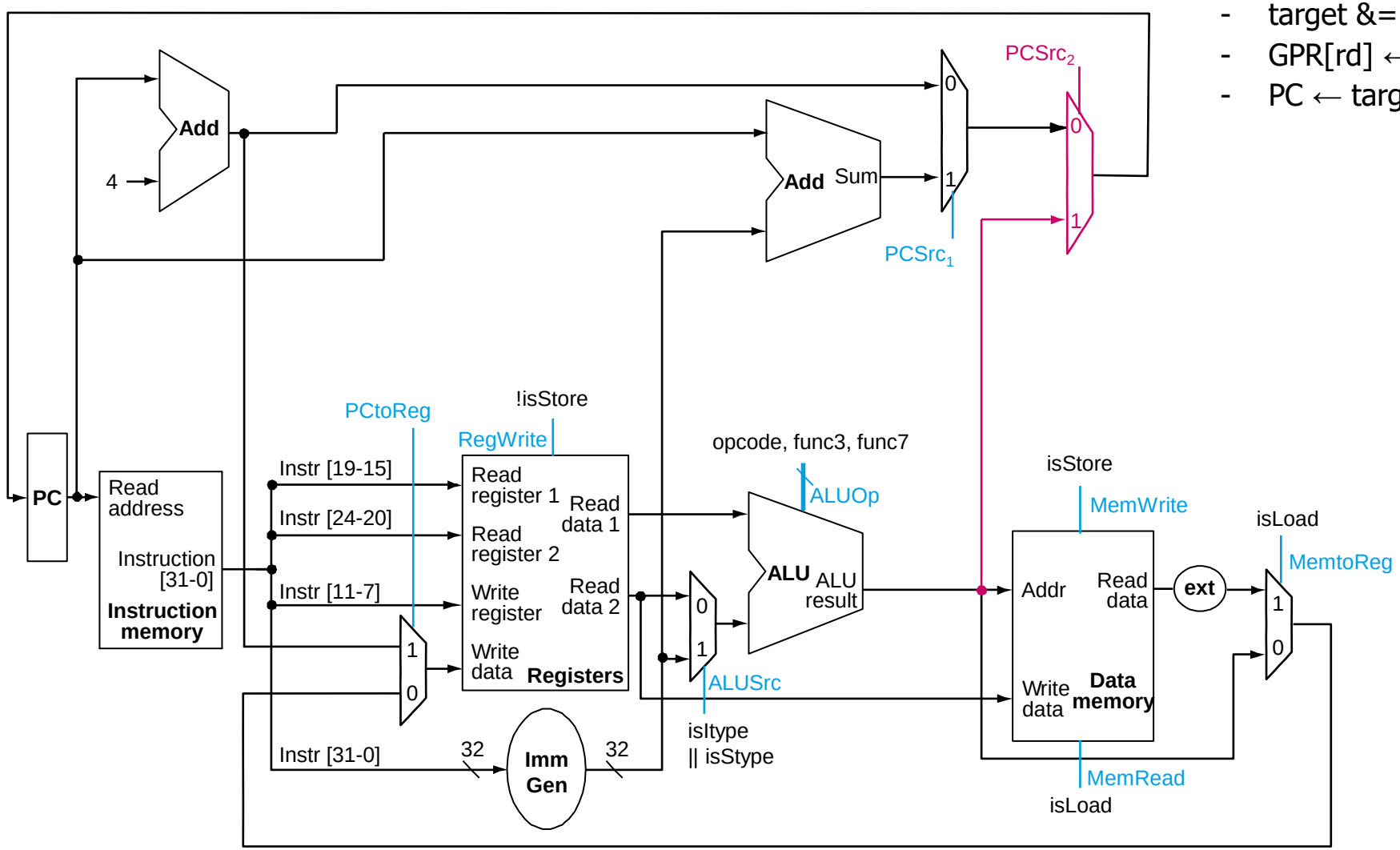
- How should this change for JALR?



- Semantics:
  - target = GPR[rs1] + sign-extend(imm_{12})
  - target &= 0xFFFFFFFE //ignored for simplicity
  - GPR[rd] ← PC + 4
  - PC ← target

Unconditional Jump Datapath (JALR)

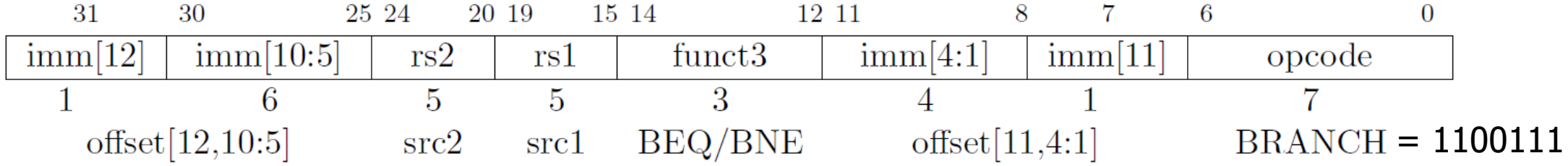
- Semantics:
  - target = GPR[rs1] + sign-extend(imm_{12})
  - target &= 0xFFFFFFFE //ignored for simplicity
  - GPR[rd] ← PC + 4
  - PC ← target



**Based on figures from [P&H CO&D, COPYRIGHT 2017 Elsevier. ALL RIGHTS RESERVED.]

Reminder: Cond. Branch Instructions (SB-Type)

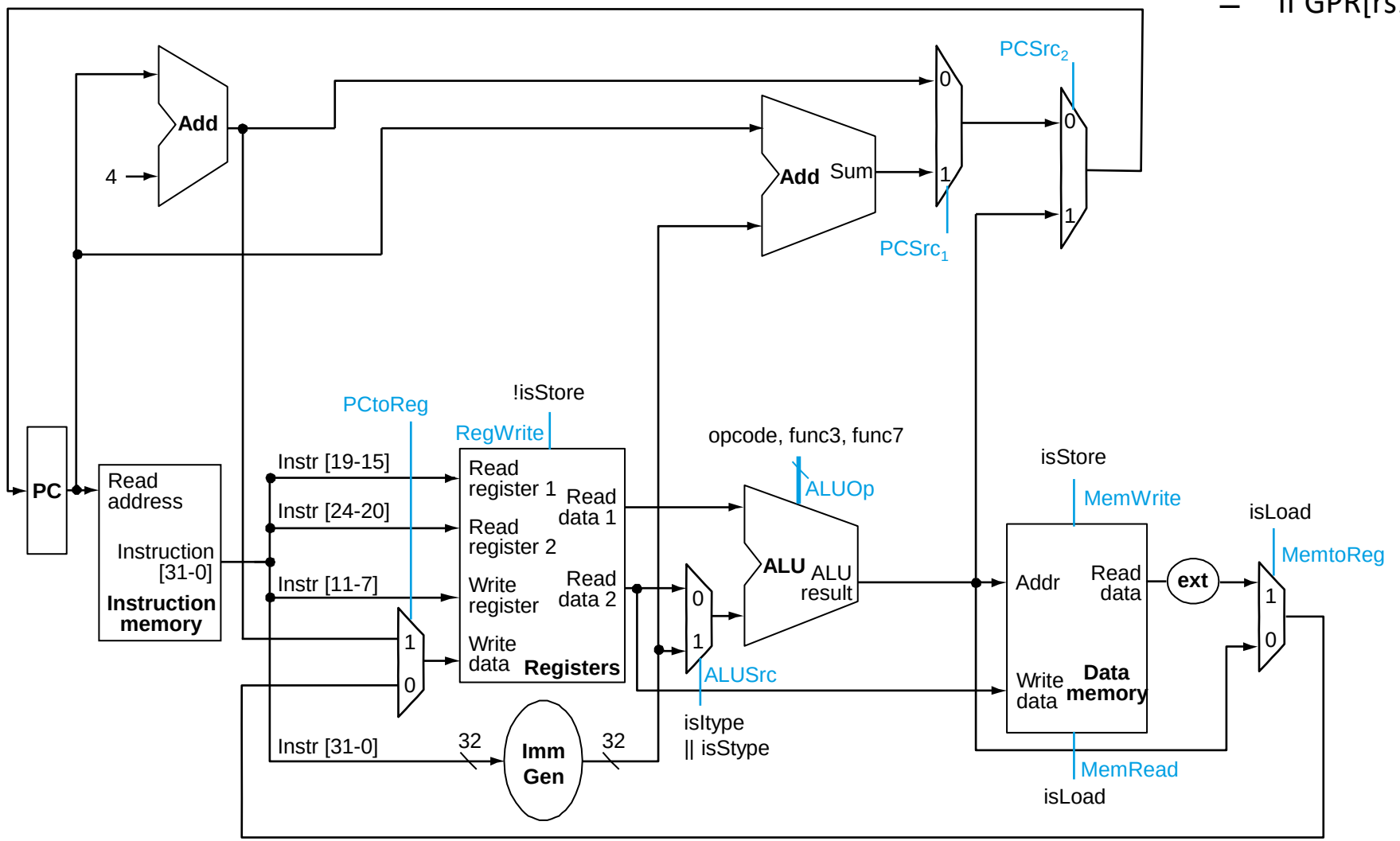
- Assembly example (Branch if EQual): `BEQ rs1, rs2, imm`$_{13}$
- Encoding:



  - Note: imm[0] == 0 (only branch to even addresses) – is this okay?
- Semantics:
  - Target = PC + sign-extend(imm$_{13}$) // PC-relative addressing -- how far can you jump?
  - If GPR[rs1] == GPR[rs2] then PC ← target
    else PC ← PC + 4
- Exception: misaligned target (4-byte) if taken
- Variations
  - BEQ, BNE, BLT, BGE (signed variations) – why no BGT (Greater Than)?
  - BLTU, BGEU (unsigned variations)

Conditional Branch Datapath

- How should this change for conditional branches?

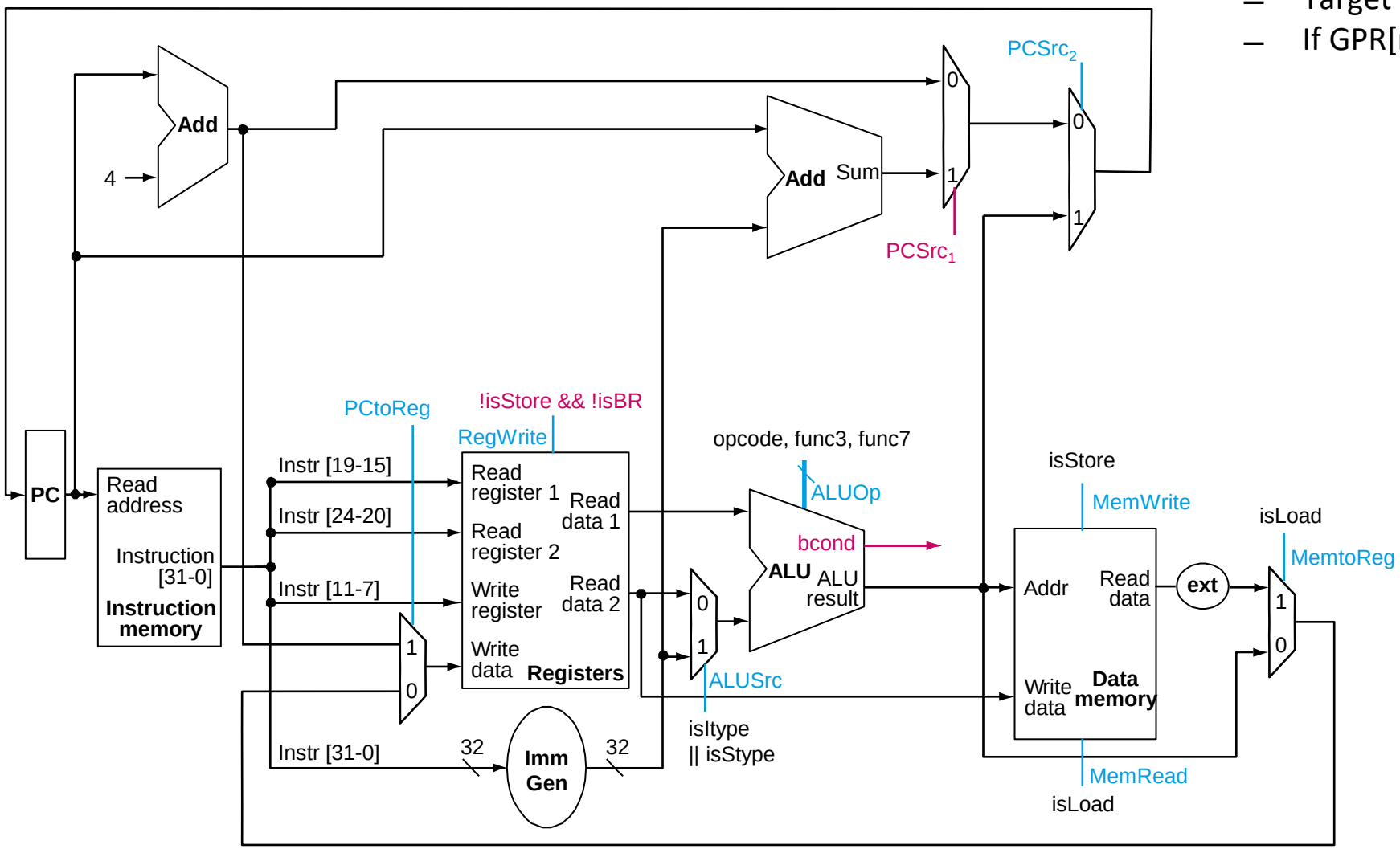


- Semantics (BEQ):
  - Target = PC + sign-extend(imm_{13})
  - If GPR[rs1] == GPR[rs2] then PC ← target
    else PC ← PC + 4

**Based on figures from [P&H CO&D, COPYRIGHT 2017 Elsevier. ALL RIGHTS RESERVED.]

Conditional Branch Datapath

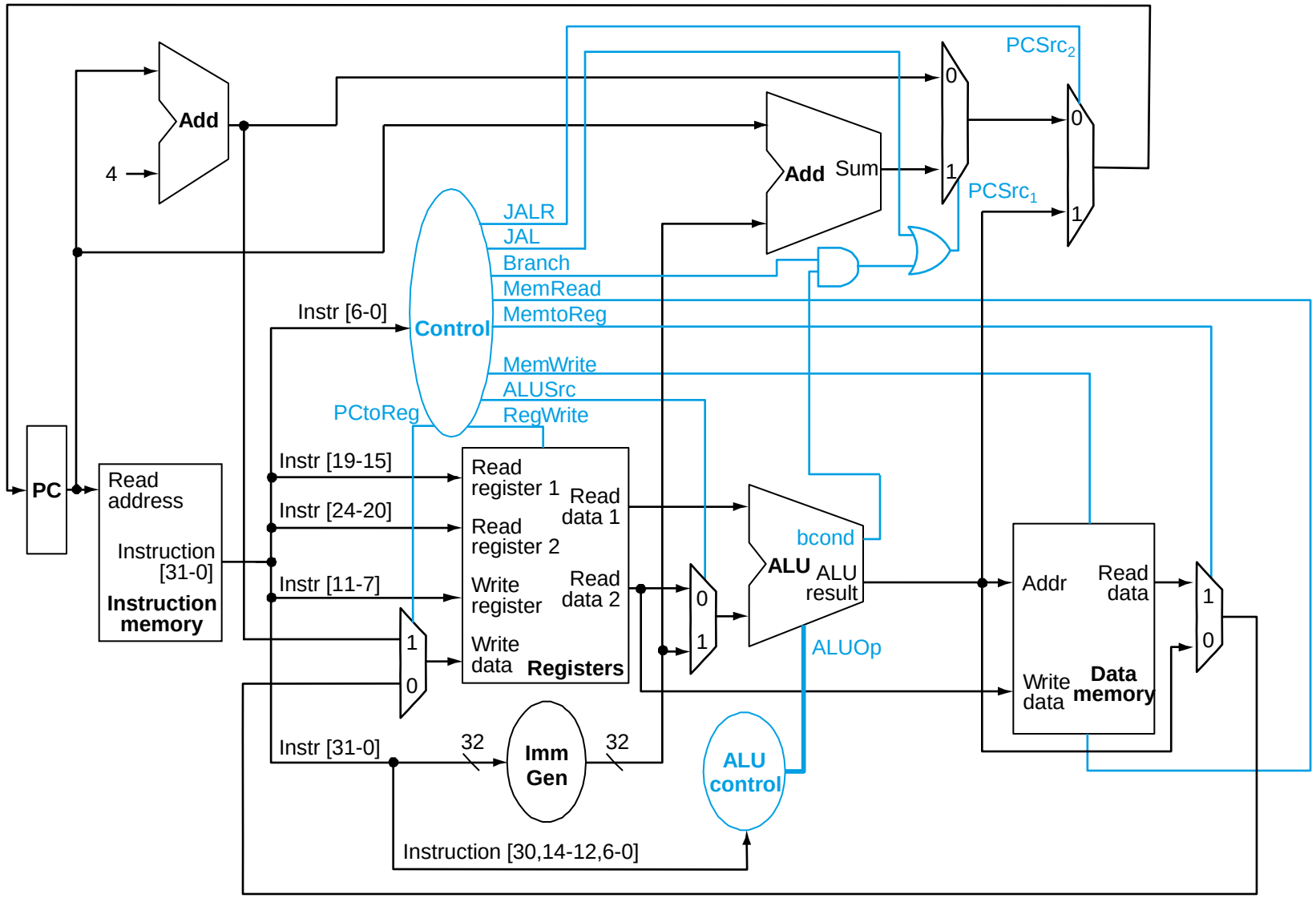
- Semantics (BEQ):
  - Target = PC + sign-extend(imm_{13})
  - If GPR[rs1] == GPR[rs2] then PC ← target
    else PC ← PC + 4



**Based on figures from [P&H CO&D, COPYRIGHT 2017 Elsevier. ALL RIGHTS RESERVED.]

Control

Datapath with Control



Single-bit Control Values

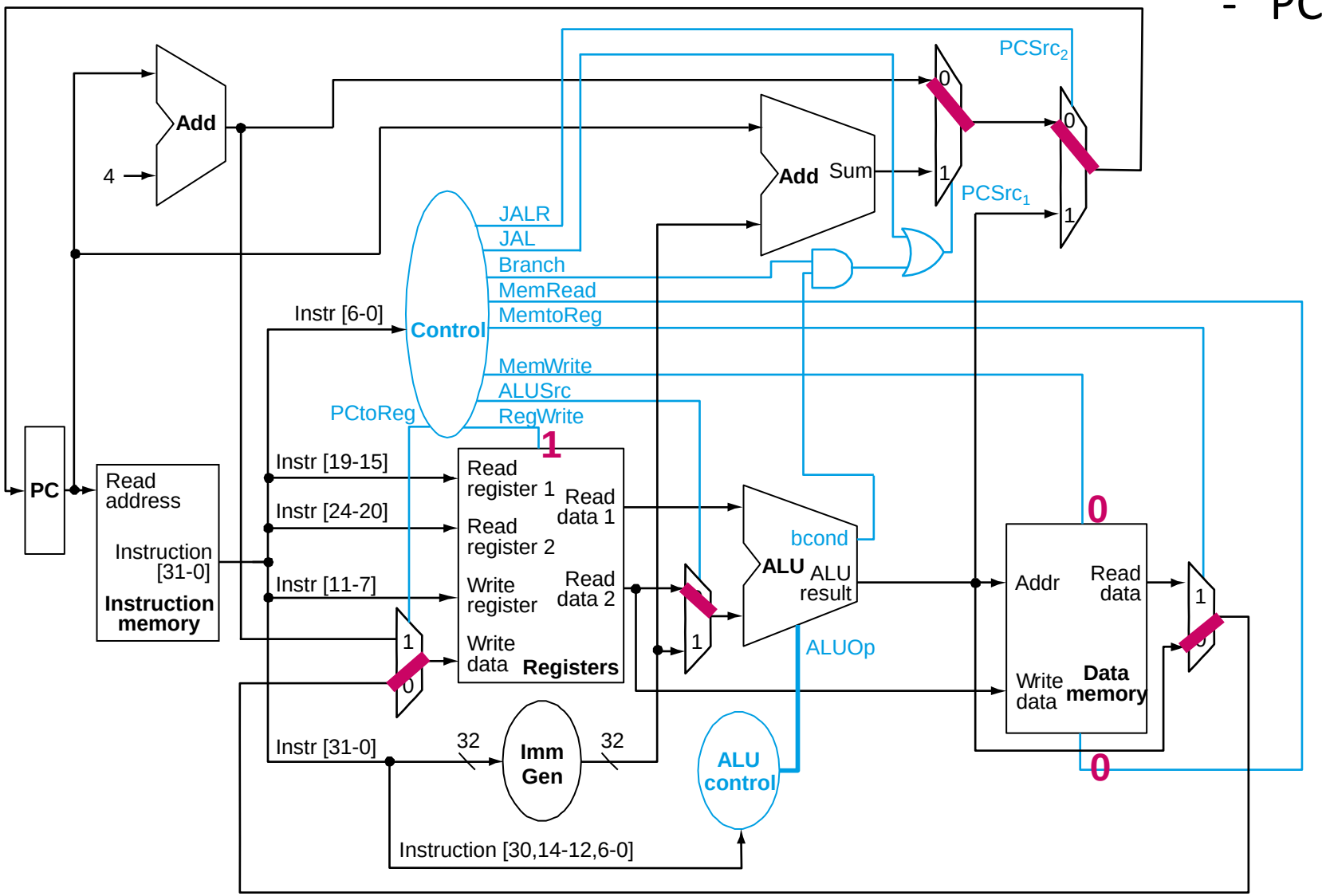
	When de-asserted	When asserted	Equation
RegWrite	GPR write disabled	GPR write enabled	(opcode!=SW/SH/SB) && (opcode!=Bxx)
ALUSrc	2nd ALU input from 2nd GPR read port	2nd ALU input from immediate	(opcode!=isRtype) && (opcode!=isSBtype)
MemRead	Memory read disabled	Memory read port return load value	opcode==LW/LH/LB
MemWrite	Memory write disabled	Memory write enabled	opcode==SW/SH/SB
MemtoReg	Steer ALU result to GPR write port	Steer memory load to GPR write port	opcode==LW/LH/LB
PCtoReg	Steer above result to GPR write port	Steer PC+4 to GPR write port	opcode==JAL/JALR
$PCSrc_1$	Next PC = PC + 4	Next PC = PC + immediate	opcode==JAL || (opcode==isSBtype && bcond)
$PCSrc_2$	Next PC is determined by $PCSrc_1$	Next PC = GPR + immediate	opcode==JALR

Multi-bit Control Values

	Options	Equation
ALUOp	• ADD, SUB, AND, OR, XOR, NOR, LT, and Shift • btype: EQ, NE, GE, LT	case opcode RTypeALU : according to funct3, funct7[5] ITypeALU : according to funct3 only (except shift) LW/SW/JALR : Add Bxx : Subtract and select btype
ImmGen	I-type, I-type(unsigned), S-type, SB-type, U-type, UJ-type	• Select based on instruction format type

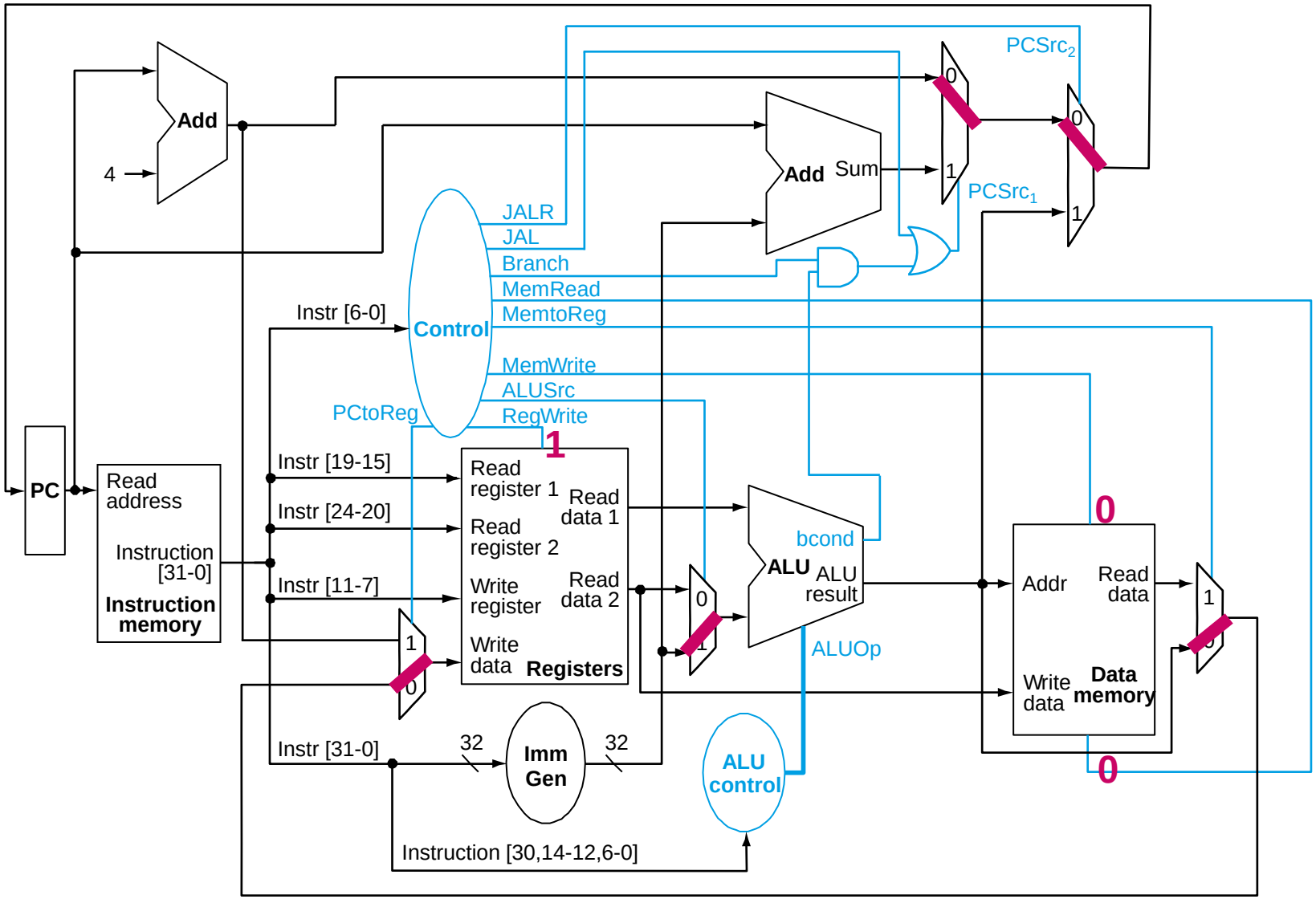
R-Type ALU

- GPR[rd] ← GPR[rs1] (op) GPR[rs2]
- PC ← PC + 4



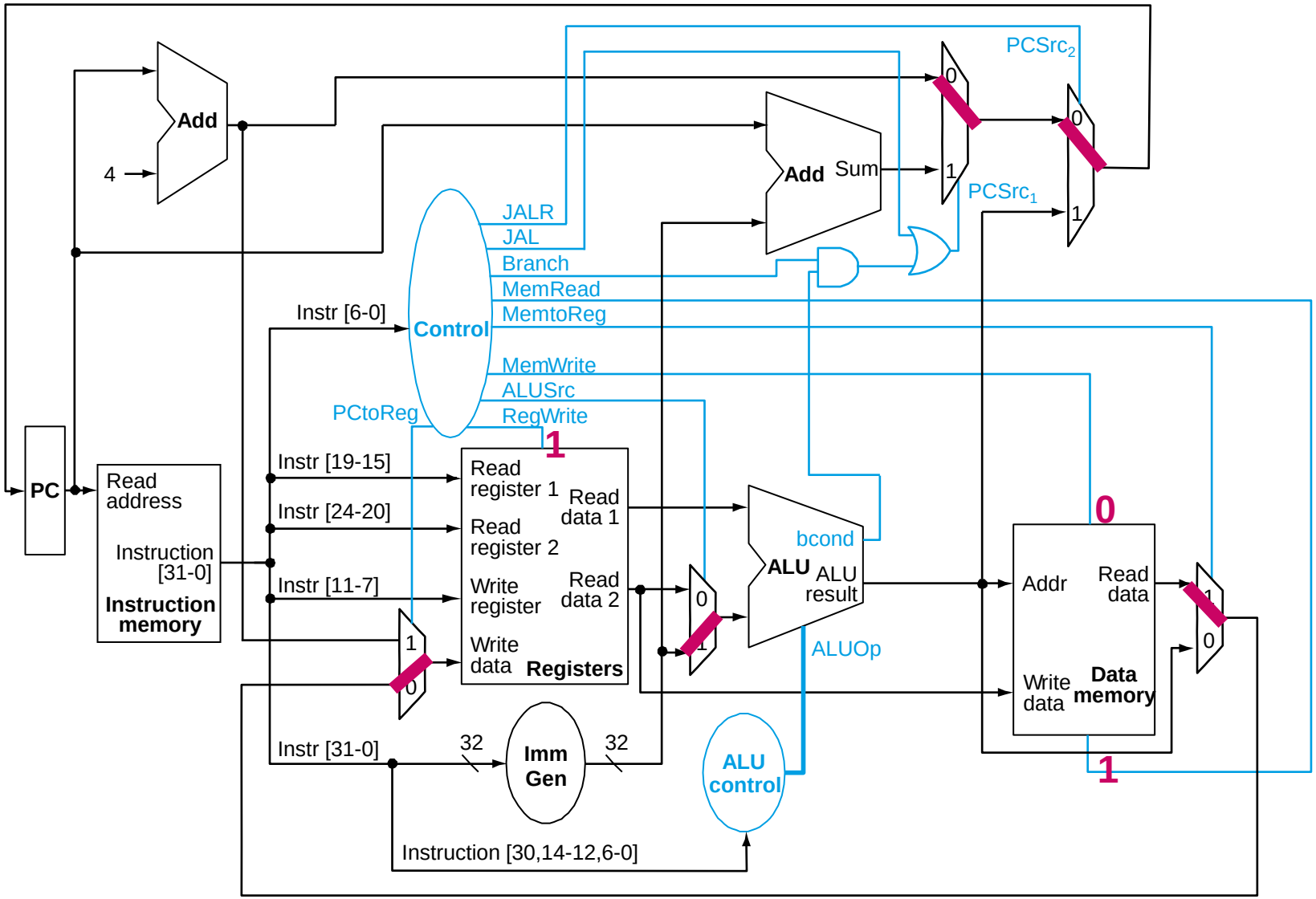
I-Type ALU

- GPR[rd] ← GPR[rs1] (op) sign-extend(imm)
- PC ← PC + 4



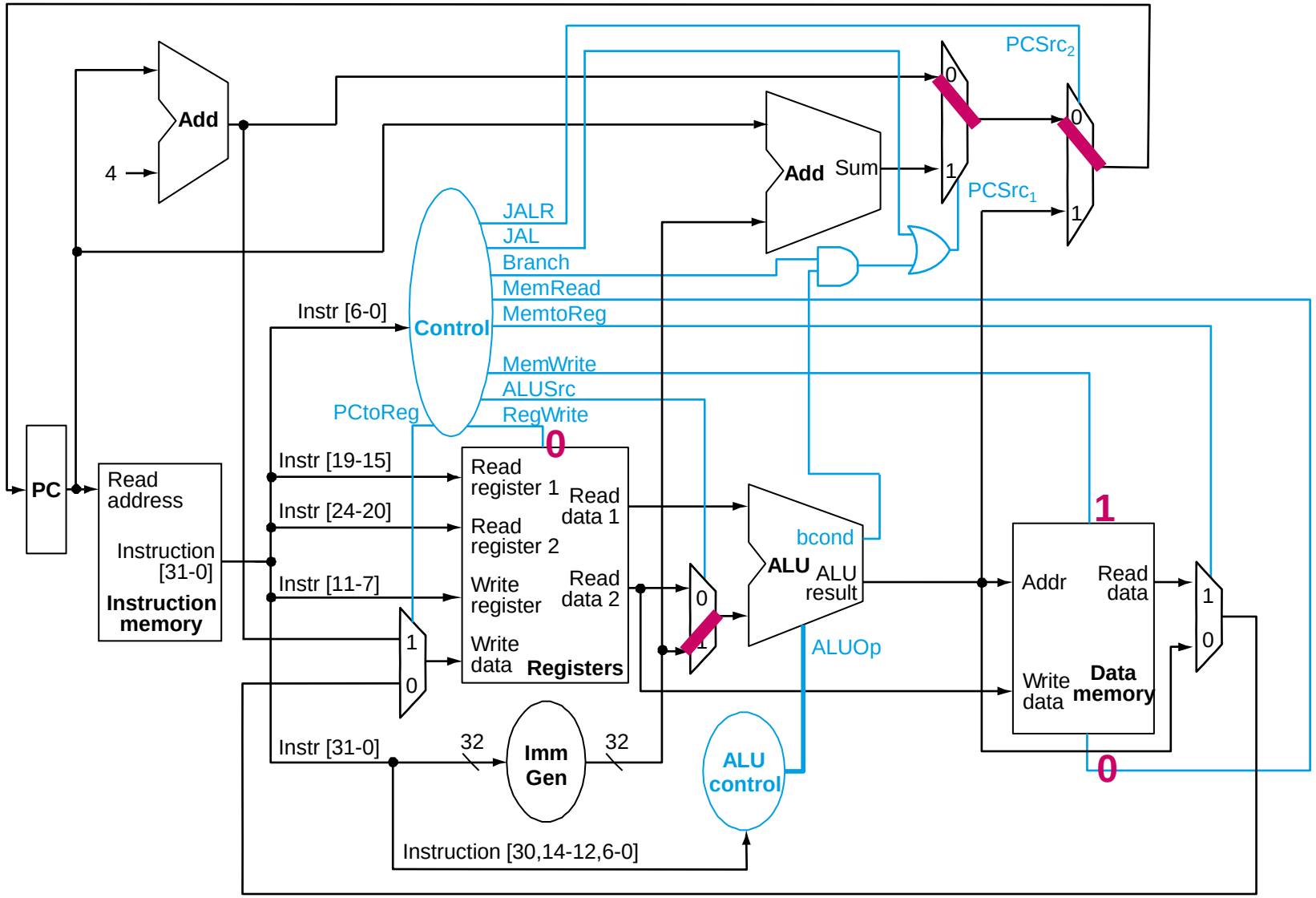
LD

- $byte\_addr_{32} = sign\text{-}extend(offset_{12}) + GPR[base]$
- $GPR[rd] \leftarrow MEM_{32}[byte\_addr_{32}]$
- $PC \leftarrow PC + 4$



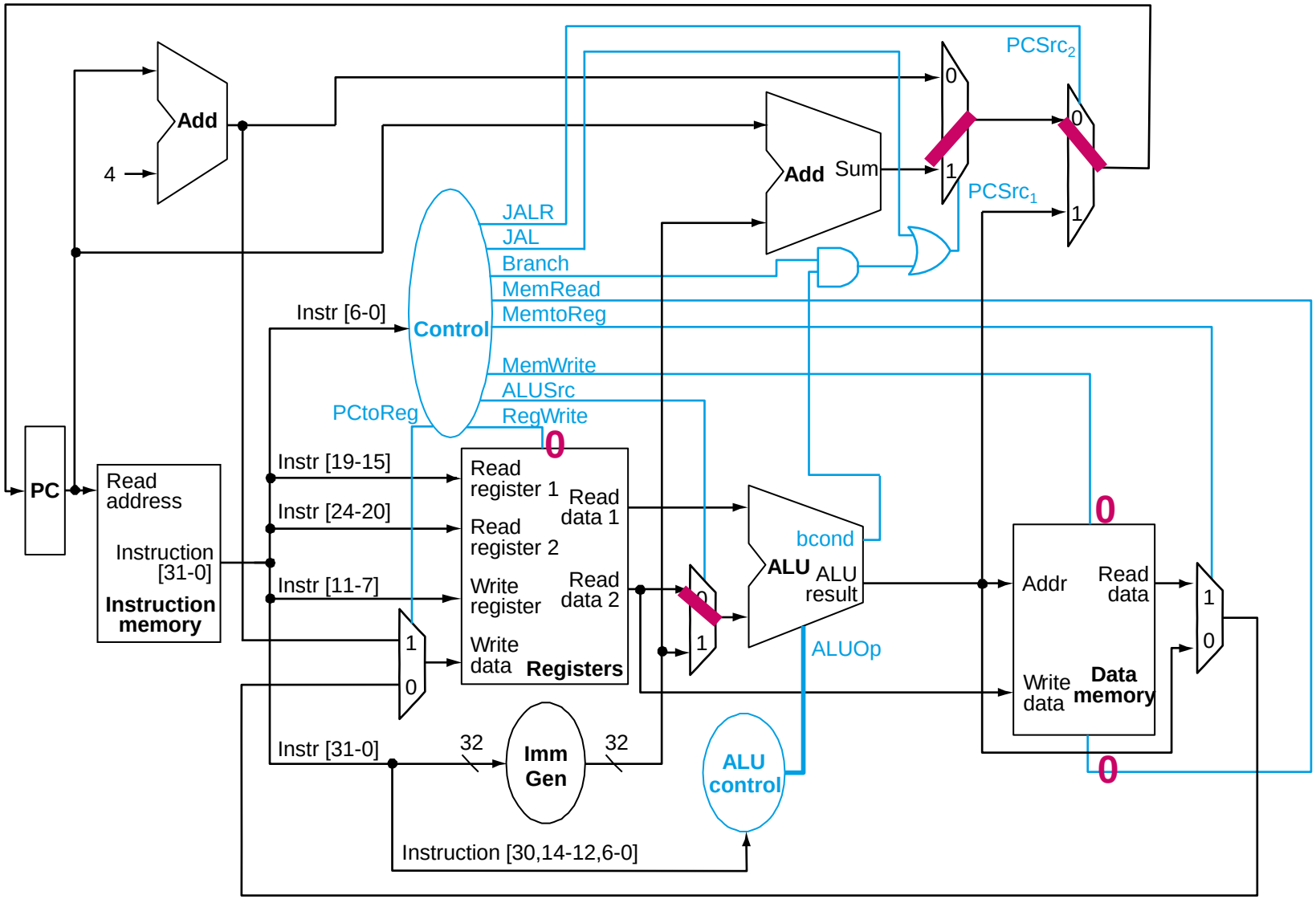
SD

- $byte\_address_{32}$ = sign-extend($offset_{12}$) + GPR[base]
- MEM_{32} [$byte\_address_{32}$] ← GPR[rs2]
- PC ← PC + 4



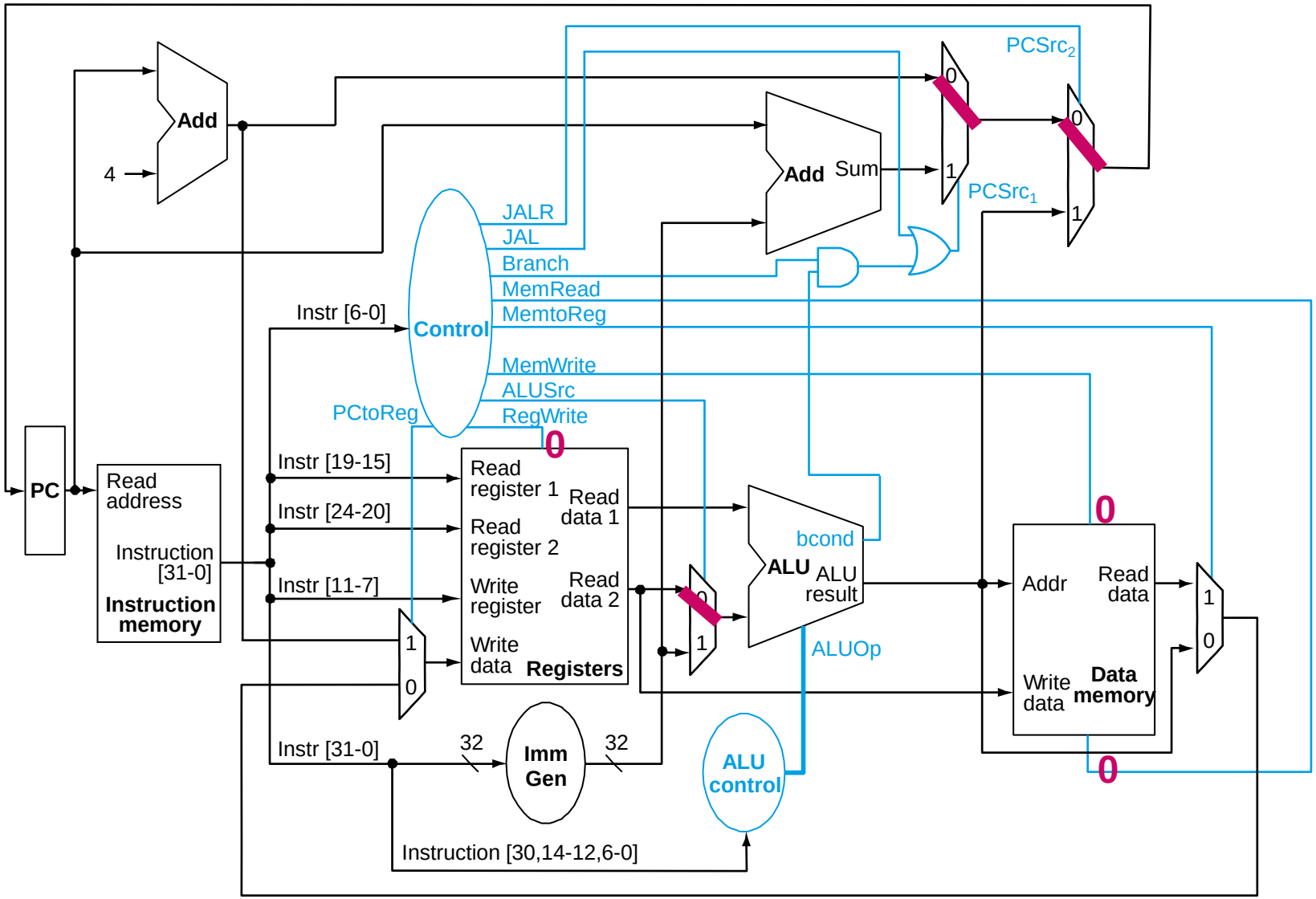
Branch Taken

- Target = PC + sign-extend(imm_{13})
- If GPR[rs1] == GPR[rs2] then PC ← target
  ~~else PC ← PC + 4~~



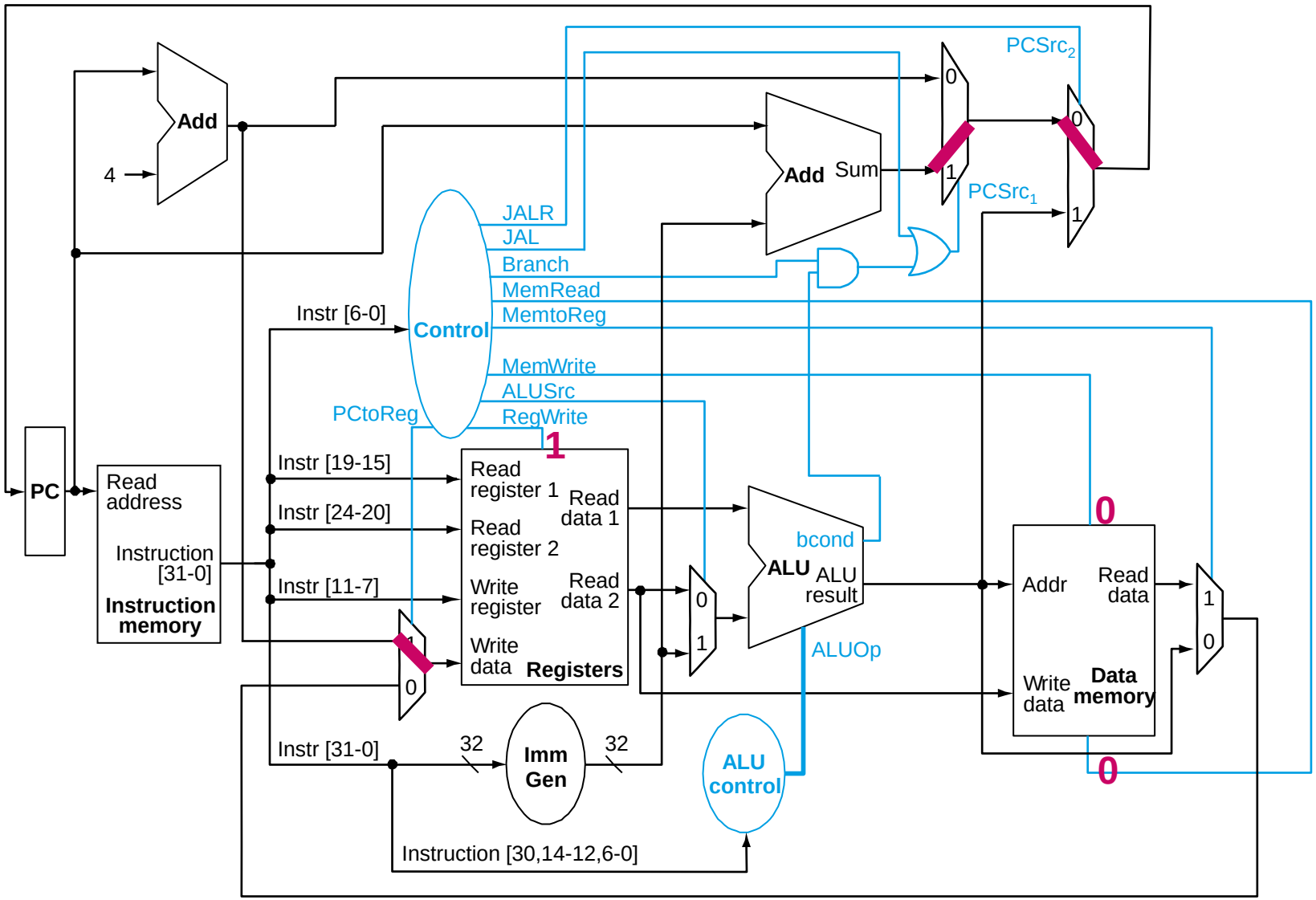
Branch Not Taken

- Target = PC + sign-extend(imm_{13})
- If GPR[rs1] == GPR[rs2] ~~then PC ← target~~
  else PC ← PC + 4



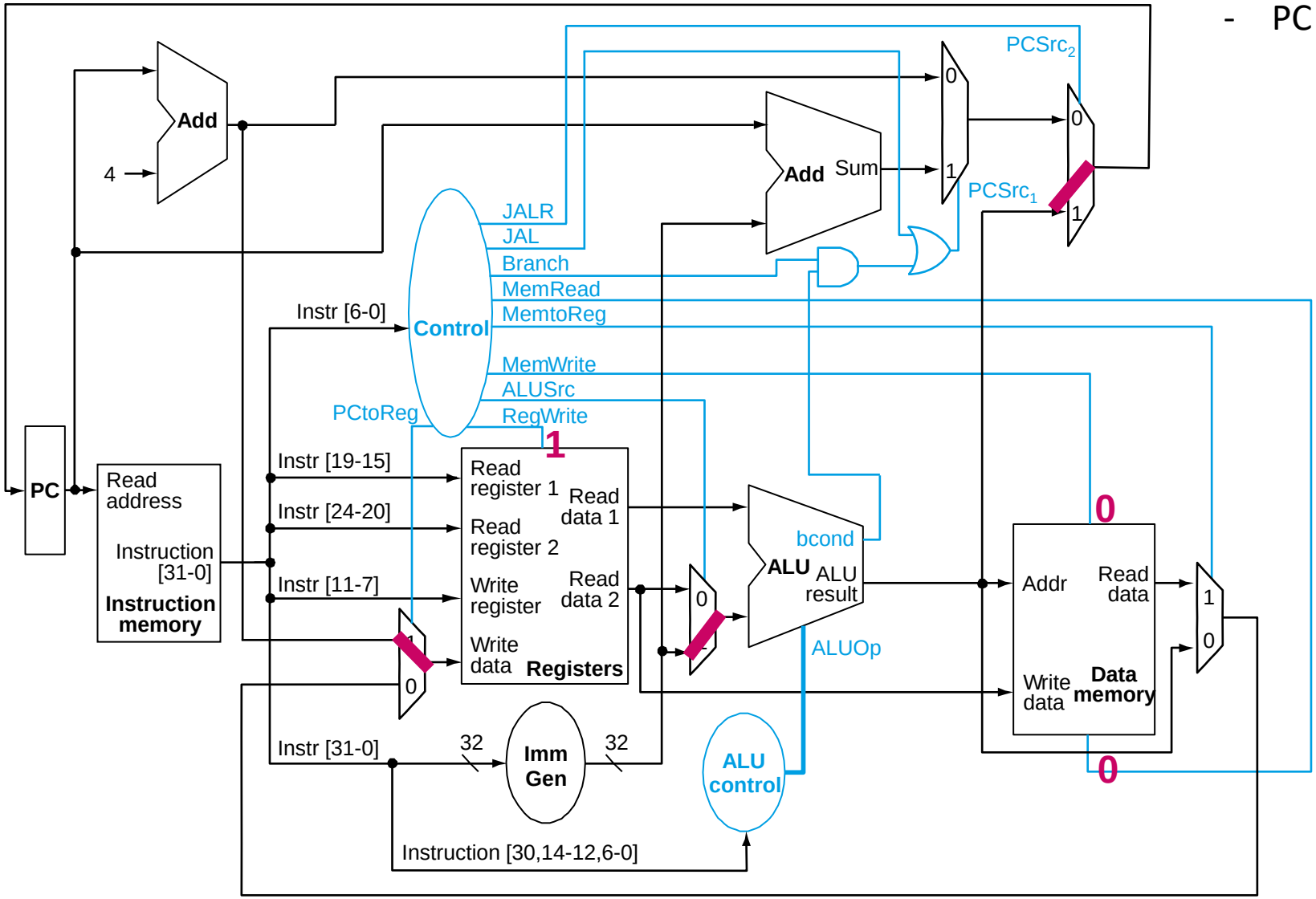
JAL

- target = PC + sign-extend(imm_{21})
- GPR[rd] ← PC + 4
- PC ← target



JALR

- target = GPR[rs1] + sign-extend(imm_{12})
- target &= 0xFFFFFFFE //ignored for simplicity
- GPR[rd] ← PC + 4
- PC ← target



Now you know how to design a single-cycle RISC-V CPU !!

- However, this is a very slow CPU
- We will cover multi-cycle CPU implementations (faster!) in the next lecture

Question?

Announcements

- Reading: P&H (RISC-V ed.) Appendix C