Santa Clara University

COEN 210

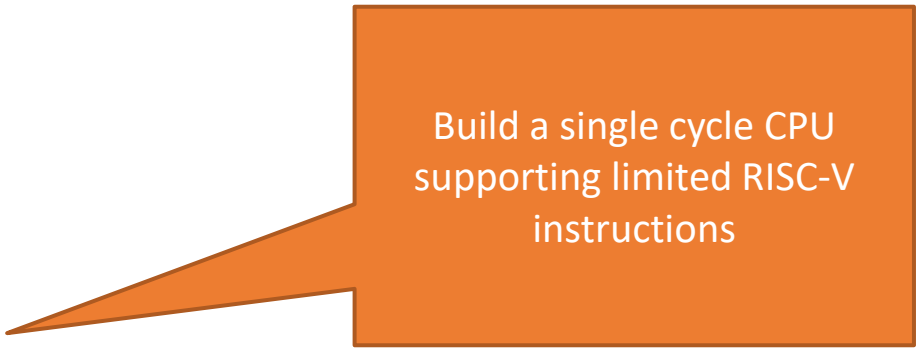
Chapter 4, Part 2 – Single Cycle Processor

Yan Cui

ycui4@scu.edu

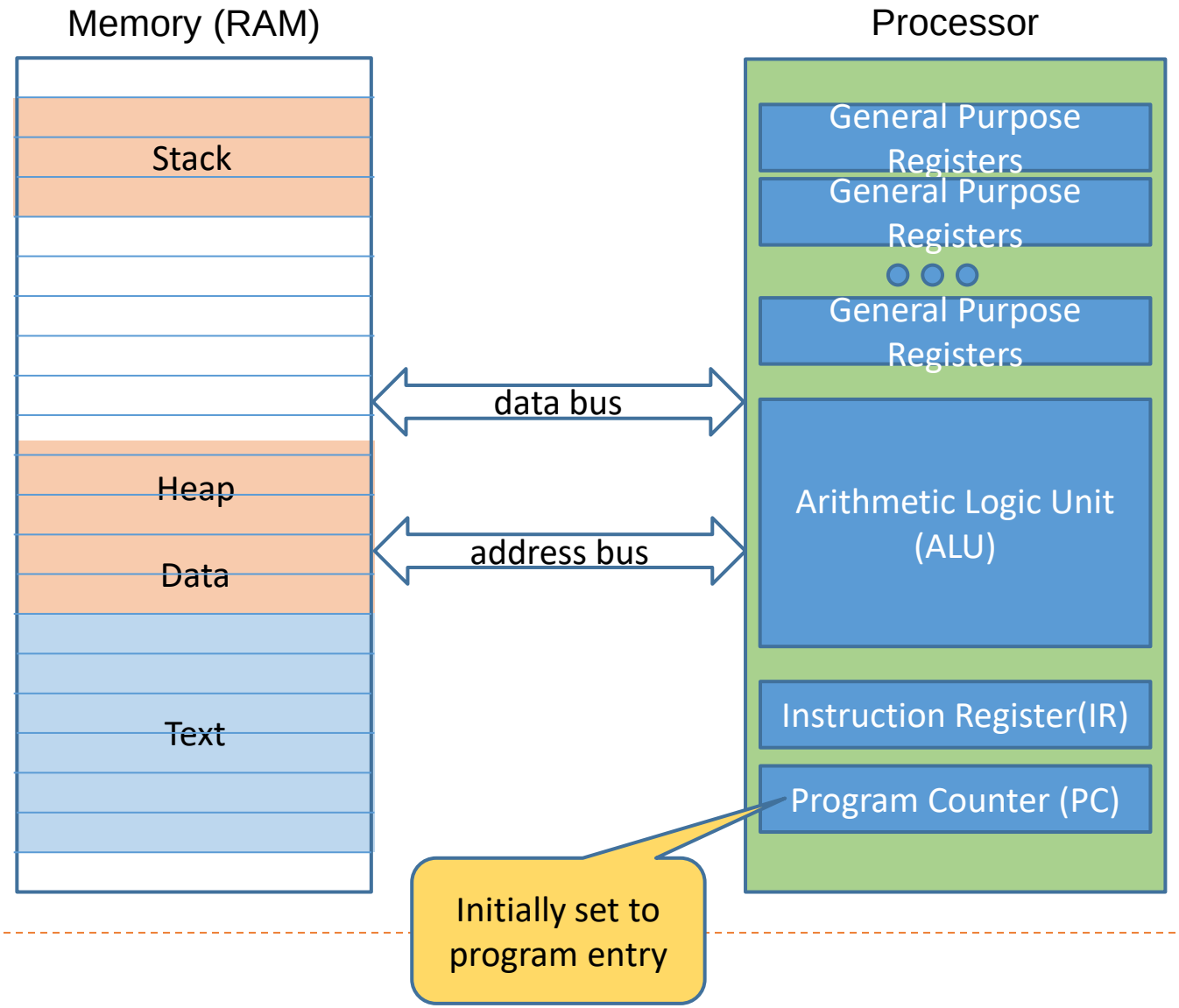
Where are we?

- Chapter 1 – Introduction
- Chapter 2 – RISC-V
- Chapter 3 - Arithmetic design
  - Chapter 3.1 - logic gates
  - Chapter 3.2 – Arithmetic
- Chapter 4 – Processor design
  - Chapter 4.1 - Sequential Logic & Memory
  - Chapter 4.2 - Single Cycle Processor
  - Chapter 4.3 - Pipelined Processor
- chapter 5 - memory hierarchy
  - Chapter 5.1 - Caches
  - Chapter 5.2 - Virtual Memory
- chapter 6 - Parallel Processors



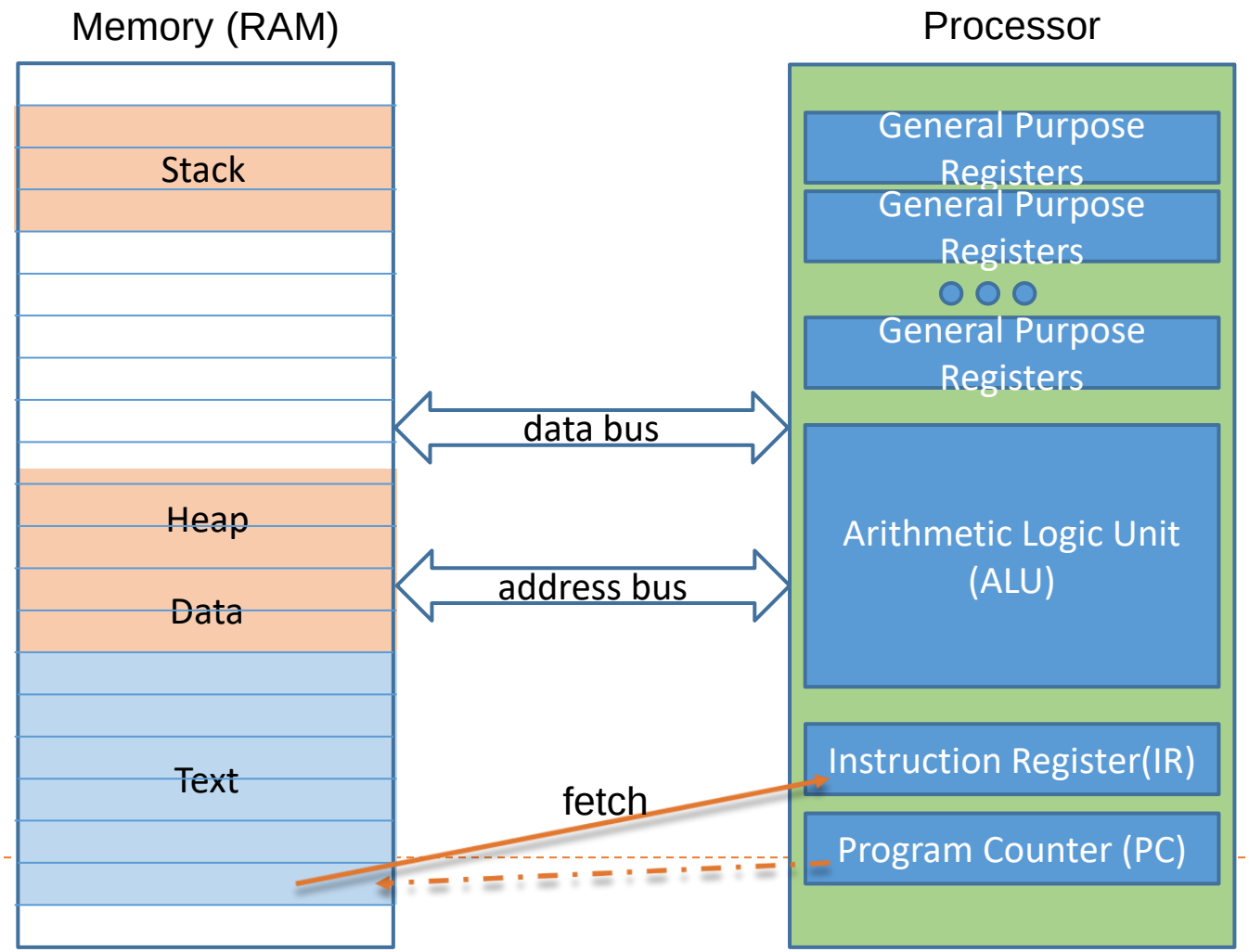
Execution Model - Running

How CPU Execute A Program



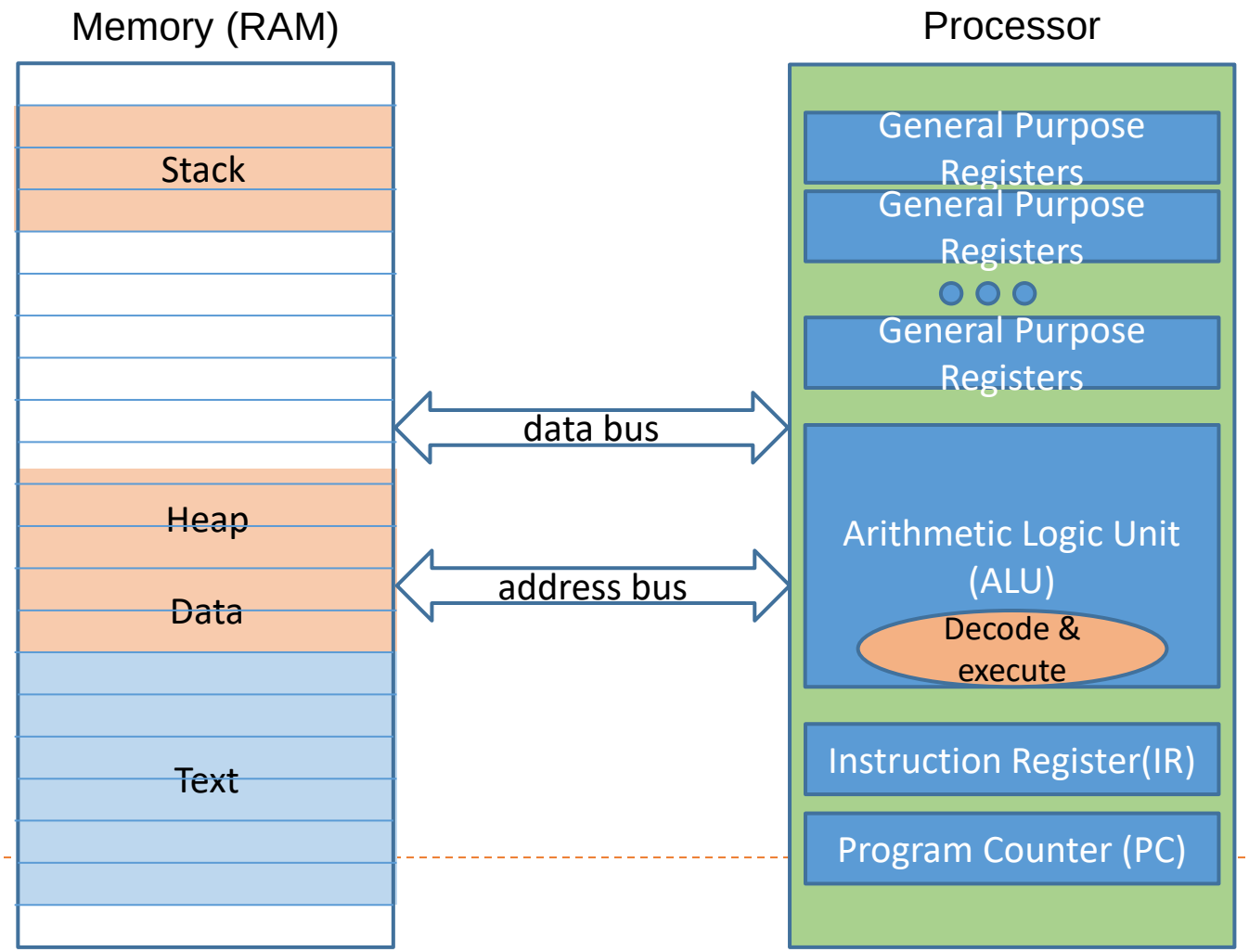
Step 1. Fetch Instruction

use the address in the PC to fetch the current instruction from instruction memory
Update the PC to hold the address of the next instruction



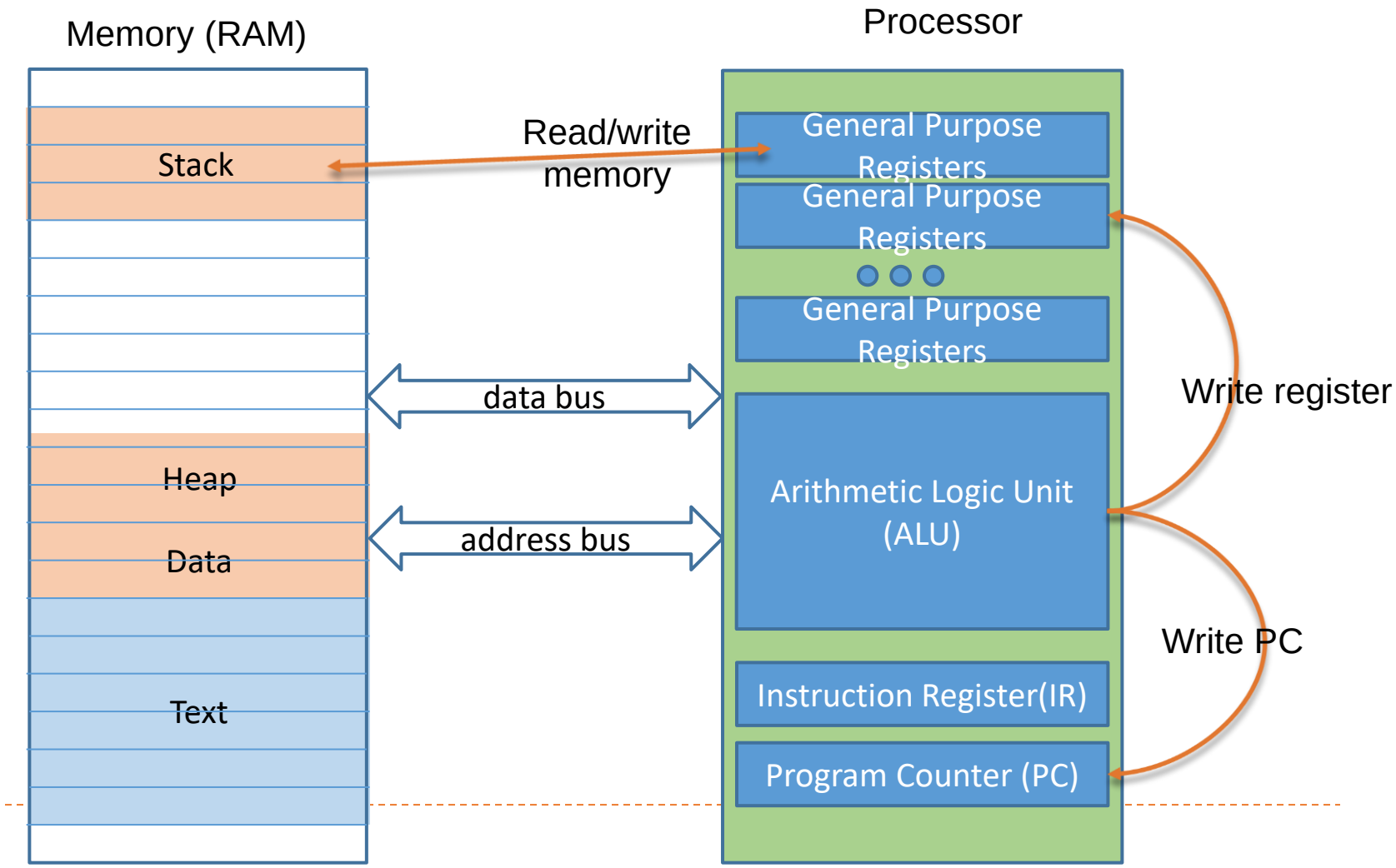
Step 2 : Decode and Execute

determine the fields within the instruction
perform the operation indicated by the instruction.

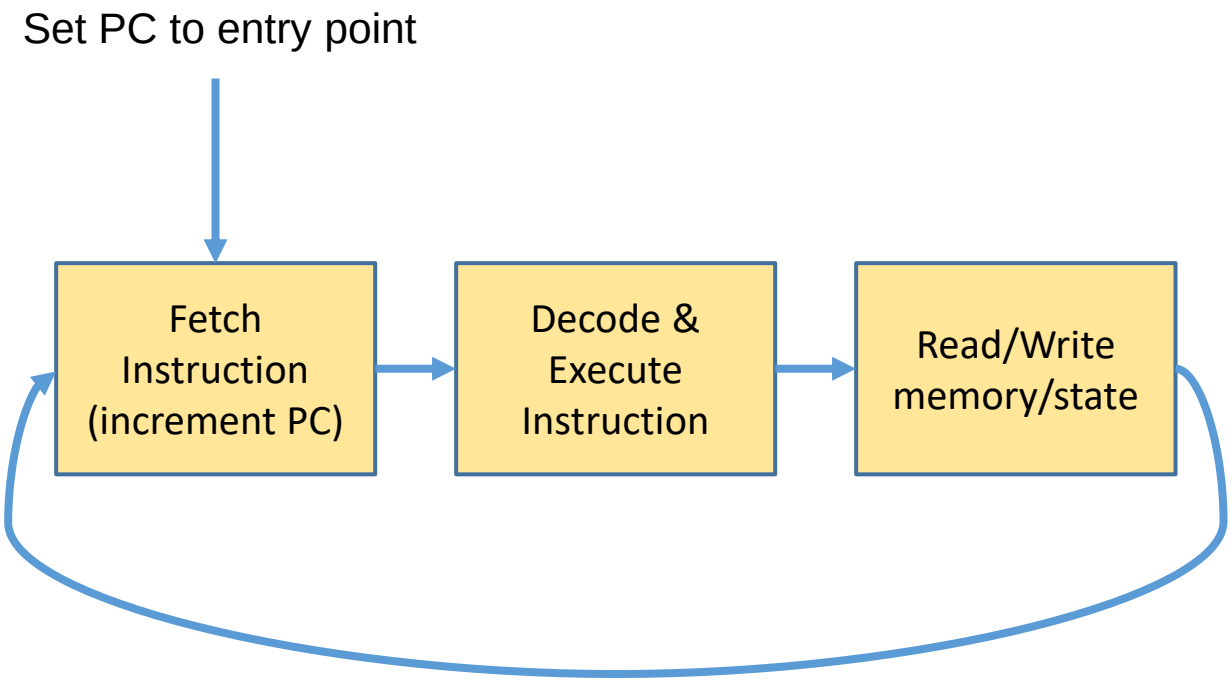


Step 3 : Read/Write Data

Storing results, if any. – write result to memory



Loop Forever until Powered Off



Review: Processors

- In lecture 1, we reminded ourselves that the datapath and control are the two components that come together to be collectively known as the processor.
- Datapath consists of the functional units of the processor
  - Elements that hold data.
    - Program counter, register file, instruction memory, etc.
  - Elements that operate on data.
    - ALU, adders, etc.
  - Buses for transferring data between elements.
- Control commands the datapath regarding when and how to route and operate on data.

Processor = datapath + control

A Single-cycle Processor

- An instruction set architecture (ISA) is an interface that defines the hardware operations which are available to software.
- Any instruction set can be implemented in many different ways. Over the next few weeks we'll see several possibilities.
  - In a basic single-cycle implementation all operations take the same amount of time—a single cycle.
  - Next, pipelining lets a processor overlap the execution of several instructions, potentially leading to big performance gains.

A Single-cycle implementation

- All instructions will execute in the same amount of time; this will determine the clock cycle time for our performance equations.
- We will describe a basic RISC-V Implementation supporting just the following instructions:

Arithmetic:	add	sub	and	or	slt
Data Transfer:	lw	sw			
Control:	beq				

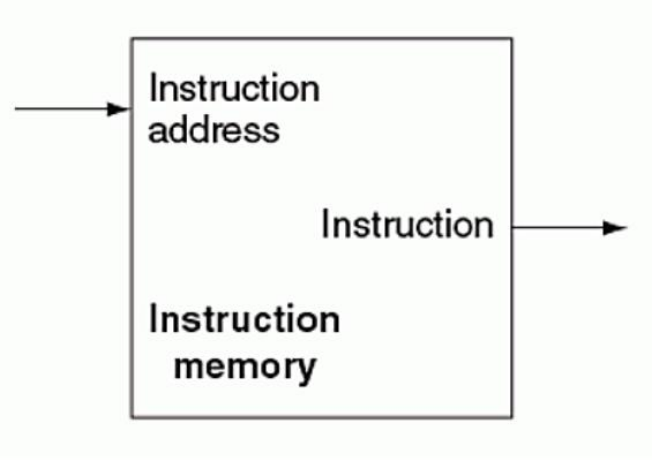
- We'll explain the datapath first, and then make the control unit.

Build Datapaths

Which datapath elements does each instruction need?

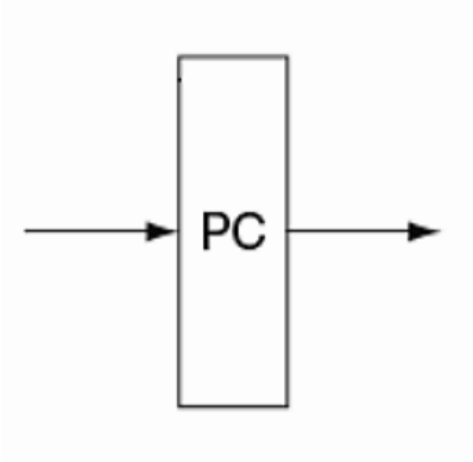
Datapath Element - Instruction Memory

- To start, we will look at the datapath elements needed by every instruction.
- First, we have instruction memory.
- Instruction memory is a memory unit to store the instructions
  - provides read-access to the instructions of a program
  - given an address as input, supplies the corresponding instruction at that address.



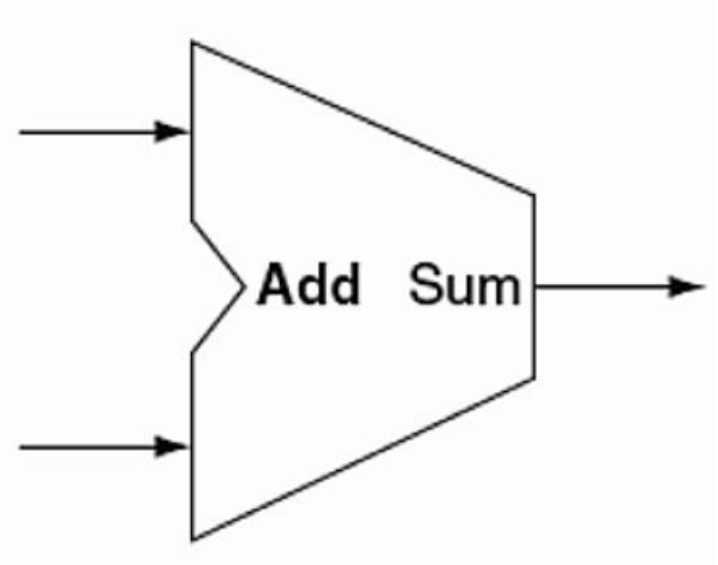
Datapath Element - Program Counter

- Next, we have the program counter or PC – a register.
- The PC holds the address of the instruction being executed.
  - Essentially, it is just a 32-bit register
  - holds the current instruction address
  - updated at the end of every clock cycle
  - Normally PC increments sequentially except for branch instructions
- The arrows on either side indicate that the PC is both readable and writeable.



Datapath Element - Adder

- The adder is responsible for incrementing the PC to the address of the next instruction
- Built from ALU by wiring the control lines so that the control always specifies an add operation
- It takes two input values, adds them together and outputs the result.

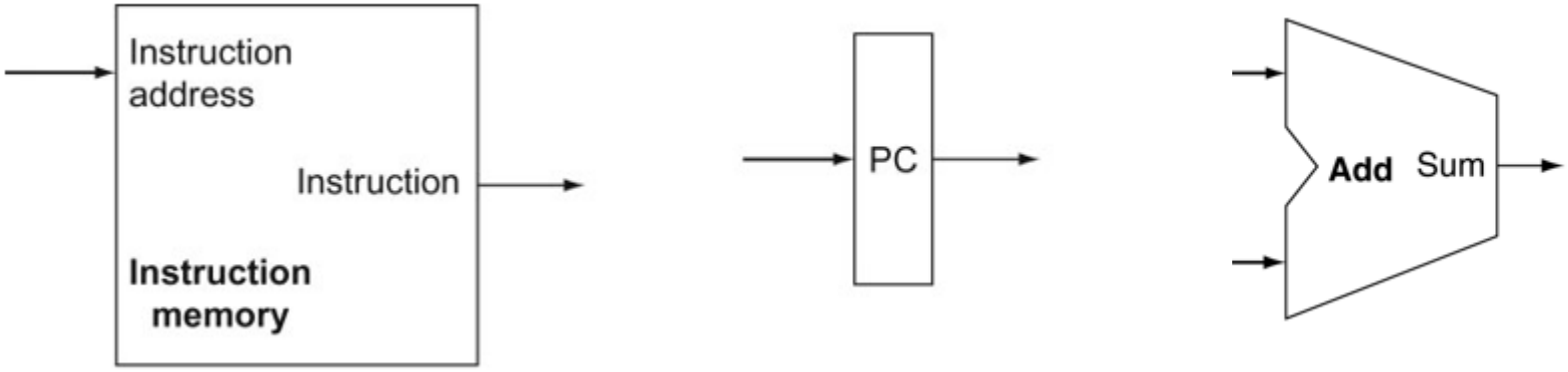


Datapath

- So now we have instruction memory, PC, and adder datapath elements.
- Now, we can talk about the general steps taken to execute a program.
  - Fetch Instruction from memory:
    - use the address in the PC to fetch the current instruction from instruction memory.
    - Update the PC to hold the address of the next instruction
  - Instruction decoding & Instruction execution
    - determine the fields within the instruction
    - perform the operation indicated by the instruction
  - Read/Write Data
    - Read or write data from memory
    - Storing results, if any. – write result to memory

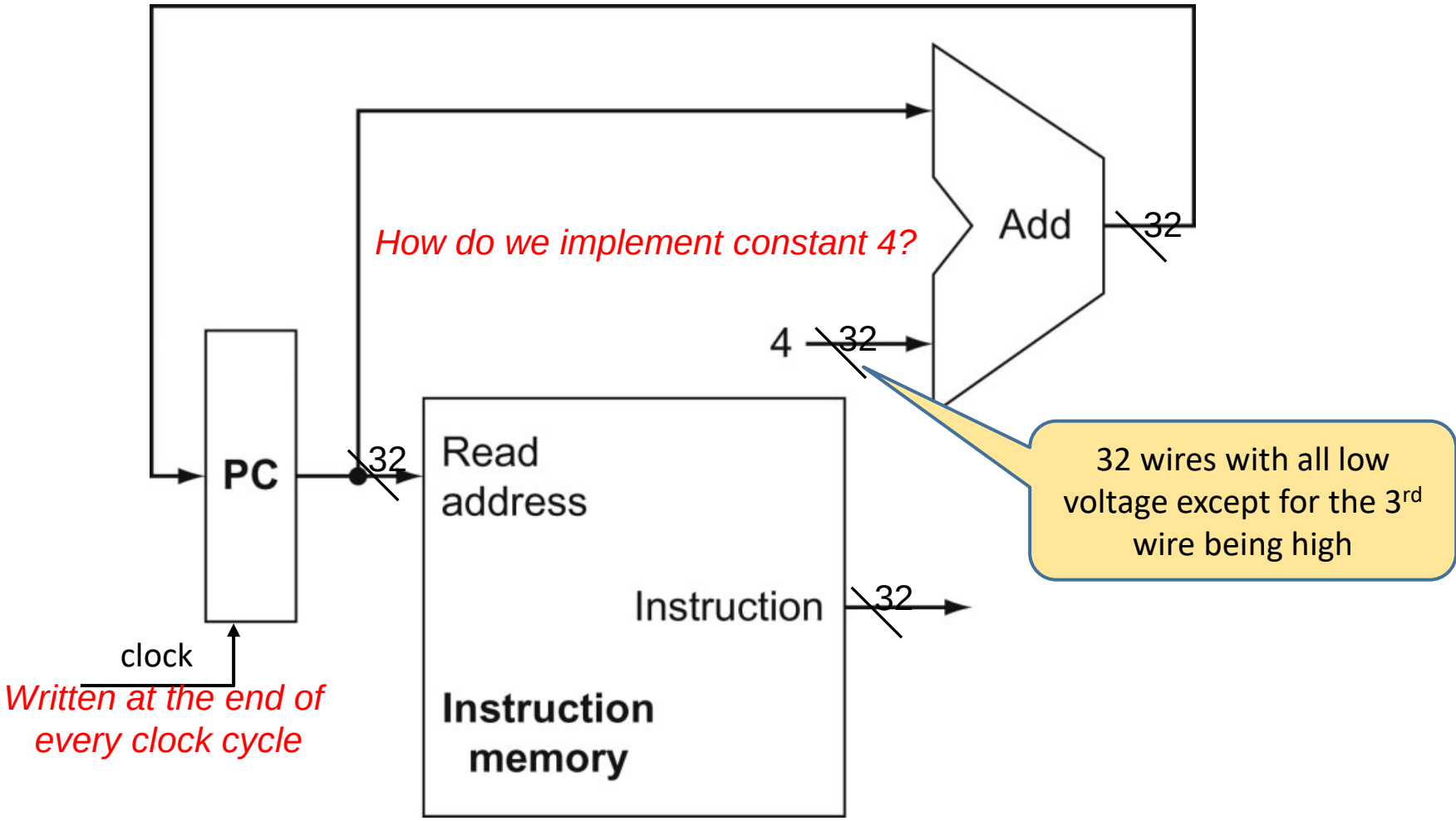
First thing's First - Fetch Instructions

- Goal
  - Fetch instruction from memory pointed by PC register
  - Increment PC by 4 pointing at the next instruction
- What components should we use?
  - **Instruction Memory** stores the instruction
  - **PC** address of current instruction
  - **ALU(adder)** increase PC by 4



Put It Together

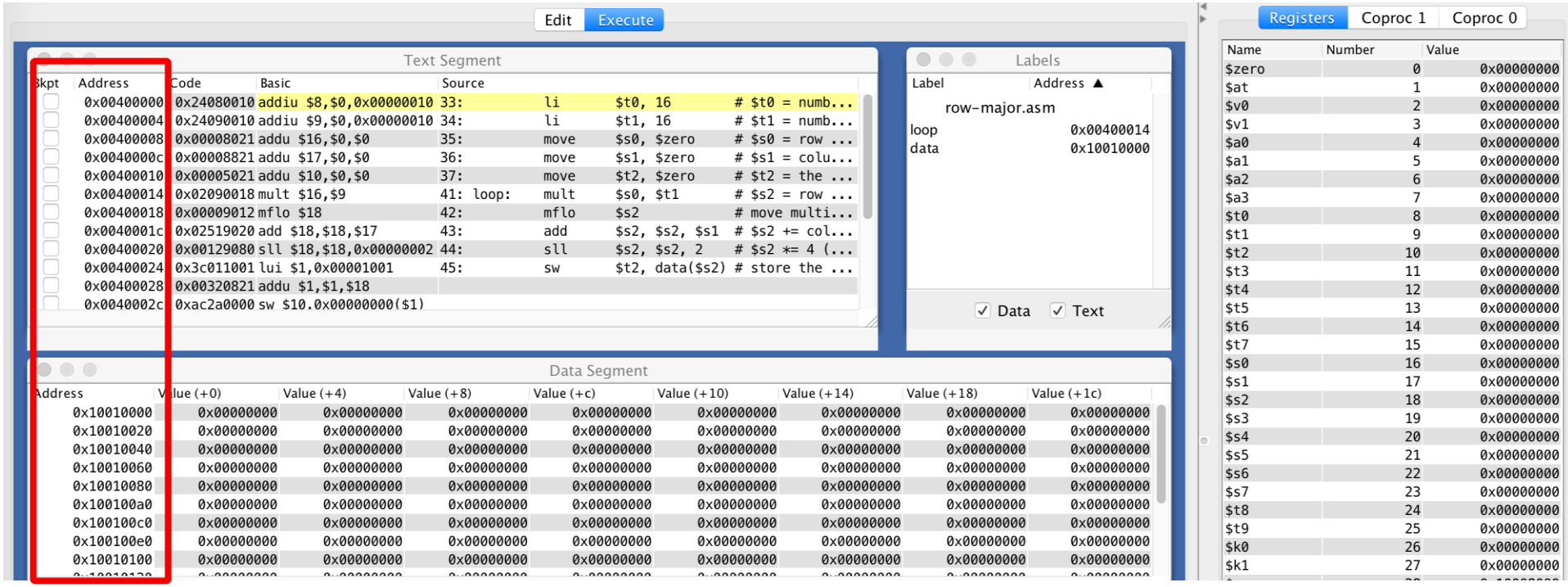
Why will PC NOT go crazy and keep on incrementing its value?



A portion of the datapath used for fetching instructions and incrementing the program counter

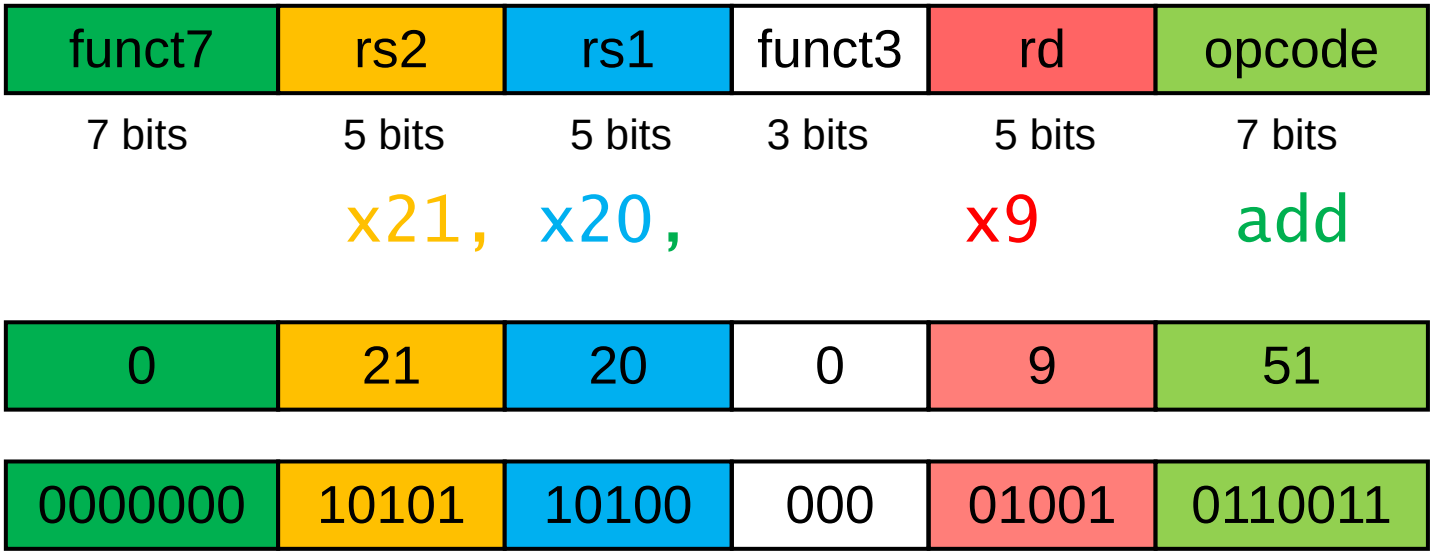
Instruction and Data

- **Each instruction is encoded as 32-bit numbers**
- **Each byte/word has its memory address**



R-Format Encoding for Arithmetic-Logic Instructions

`add x9,x20,x21` (add rd, rs1, rs2)



$0000\ 0001\ 0101\ 1010\ 0000\ 0100\ 1011\ 0011_{two} = 015A04B3_{16}$

5 bits for rd, rs1 and rs2 because we have 32 registers,

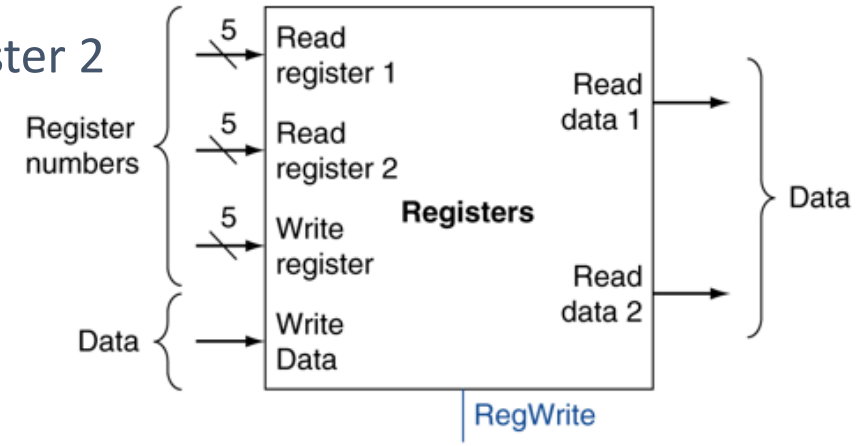
thus only needs 5 bit to address a register

R-Format Instructions – Building Blocks

- Goal
  - Perform 5 ops (add, sub, and, or, slt)
  - Read data from two read registers
  - Write result to the 3rd write register
- What components do we need?
  - Register file for reading/writing registers
  - ALU for performing 5 ops (add, sub, and, or, slt)

Register File

- The input ports are:
  - Read register 1 –first source register. 5 bits wide.
  - Read register 2 –second source register. 5 bits.
  - Write register –destination register. 5 bits wide.
  - Write data – data to be written to a register. 32 bits wide.
  - RegWrite is a control signal that is tied with clock signal
    - A writing operation only occurs when this bit is set.
- The two output ports are
  - Read data 1 –contents of source register 1
  - Read data 2 –contents of source register 2

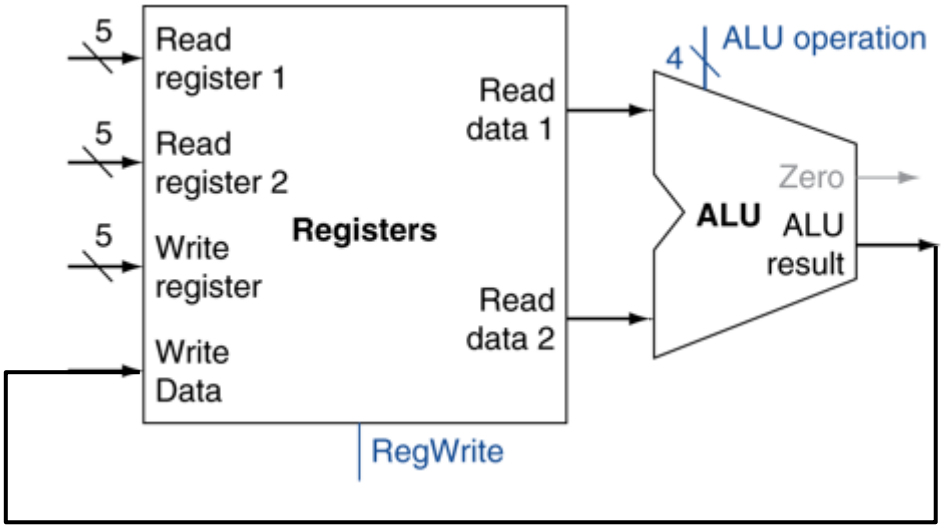


ALU

- To actually execute R-format instructions, we need to include the ALU element.
- The ALU performs the operation indicated by the instruction.
- It takes two operands, as well as a 4-bit wide operation selector value
- The result of the operation is the output value.
  - ALU operation is a part of the control. We discuss datapath first.
- We have an additional output specifically for branching –we will cover this in a minute.

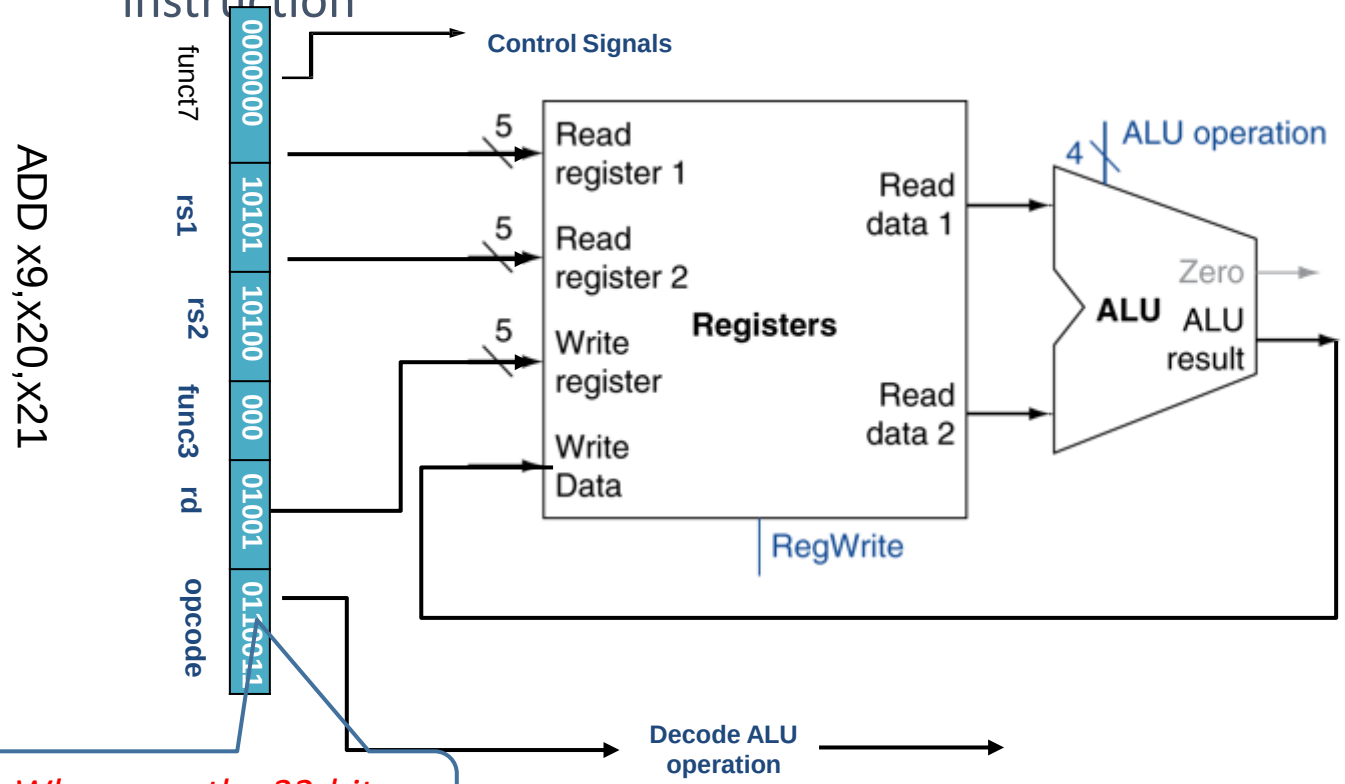
Put It Together for R-Format Instructions

- Notes:
  - RegWrite is a control signal that is tied with clock signal
  - ALU operation are control signals derived from "funct" field of the instruction



R-Format Instruction Example – ADD x9, x20, x21

- Notes
  - RegWrite is a control signal that is tied with clock signal
  - ALU operation are control signals derived from "funct" field of the instruction



Where are the 32-bit instruction coming from?

I-format Encoding Example – lw x9, 64(x22)

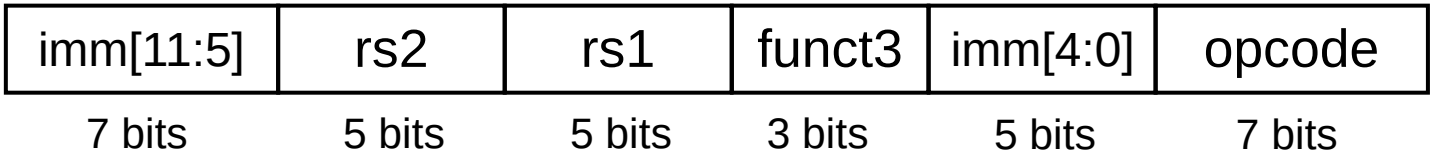
immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits
64	22	2	9	3
000001000000	10110	010	01001	0000011

Machine code:

$0000\ 0100\ 0000\ 1011\ 0010\ 0100\ 1000\ 0011_2$

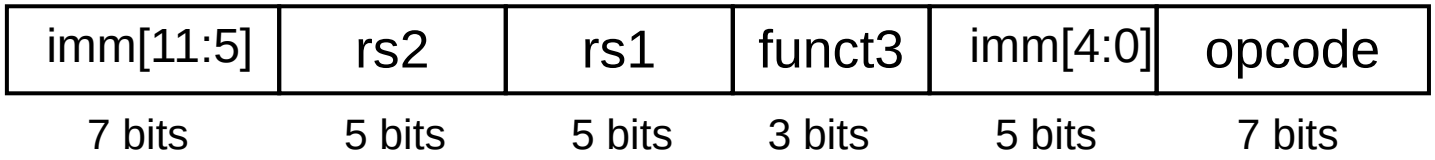
rs1: base address register number
immediate: offset added to base address
rd: destination register

RISC-V S-format Instructions

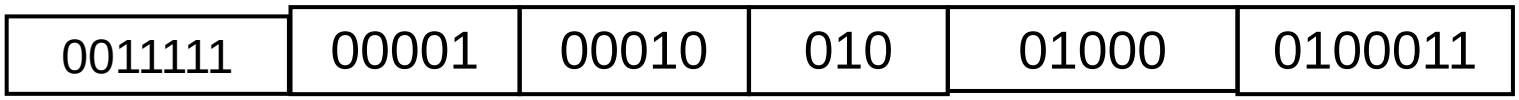


- Different immediate format for store instructions
  - rs1: base address register number
  - rs2: source operand register number
  - immediate: offset added to base address
    - Split so that rs1 and rs2 fields always in the same place

S-format Encoding Example - sw



sw x1, 1000(x2)

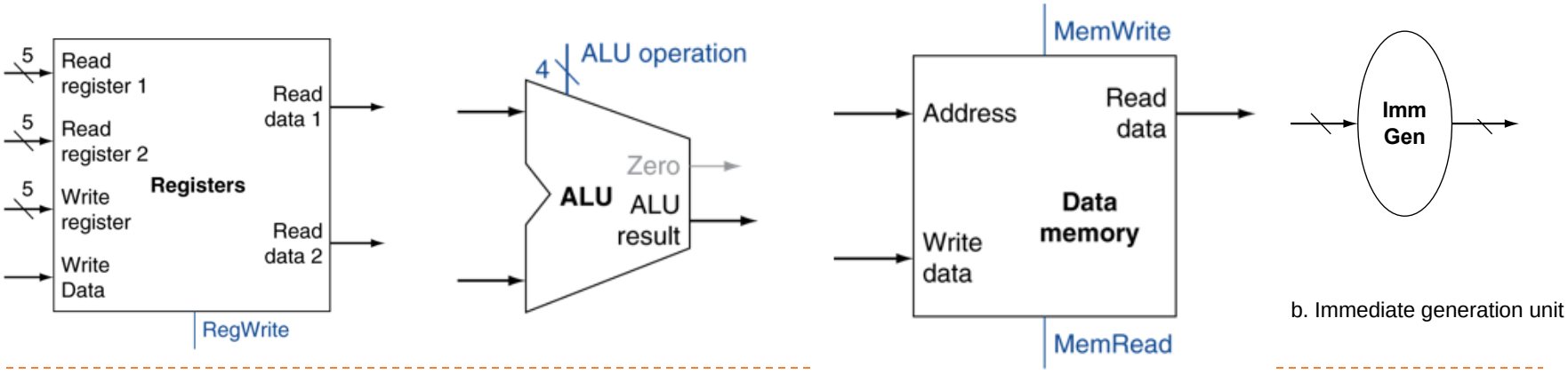


Machine code:

$0011\ 1110\ 0001\ 0001\ 0010\ 0100\ 0010\ 0011_2$

Load/Store Instructions – Building Blocks

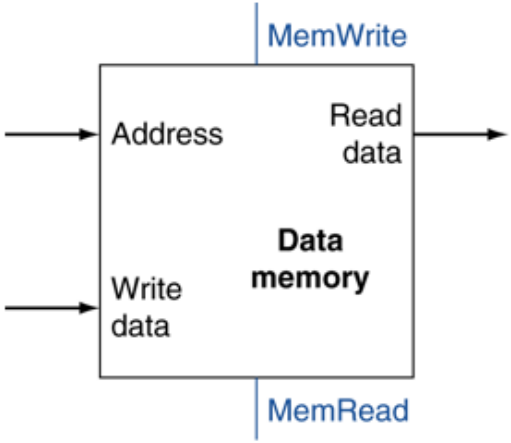
- Goal
  - Calculate target memory address (xrs1 + off12)
    - The memory address is computed by sign-extending the 12-bit immediate to 32-bits, which is added to the contents of xrs.
  - Read from memory and write to xrd; or read from xrs2 and write to memory
- What components do we need?
  - **Register file** reading/writing registers
  - **ALU** calculate the memory address
  - **Data memory** read / write data
  - **Sign extension** extend 12 bit to 32 bit for offset



b. Immediate generation unit

Data Memory Unit

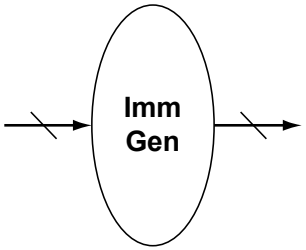
- The data memory element implements the functionality for reading and writing data to/from memory.
- There are two inputs. One for the address of the memory location to access, the other for the data to be written to memory if applicable.
- The output is the data read from the memory location accessed, if applicable.
- Reads and writes are signaled by MemRead and MemWrite, respectively, which must be asserted for the corresponding action to take place.



Immediate Generator

- Imm Gen: generate 32- or 64-bit immediate value (depending on whether we design 32-bit or 64-bit machine) from an instruction word.
  - Select the 12-bit from the instruction word and sign-extended to 32- or 64-bit.
  - **Used for for I-, S- and SB-format (I-format ALU, load, store, and beq)**

- **Elaboration on Imm generation:**
  - **Last paragraph of 4.3, page 269**



b. Immediate generation unit

Datapath for Load / Store Word

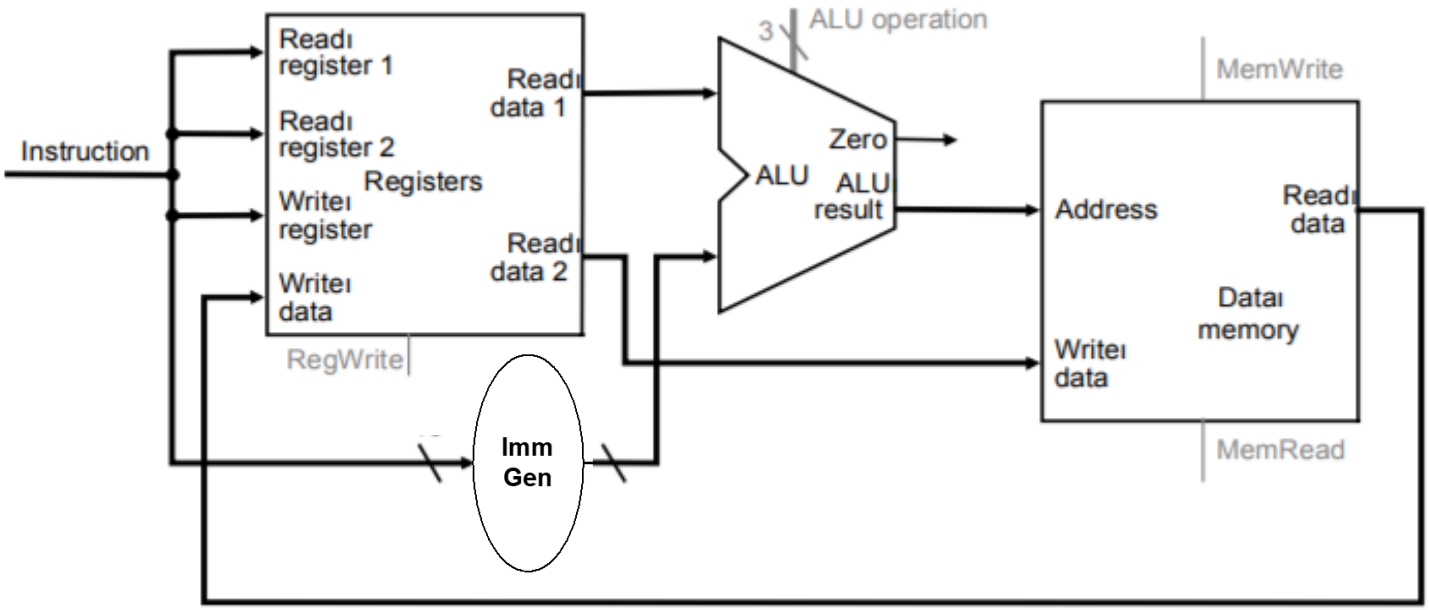
Basic steps: example: sw x1, 1000(x2)

- fetch instruction
- select two registers (rs1, rs2)
- calculate address, need ALU
- access memory (write memory)

Basic steps: example: lw x3, 64(x4)

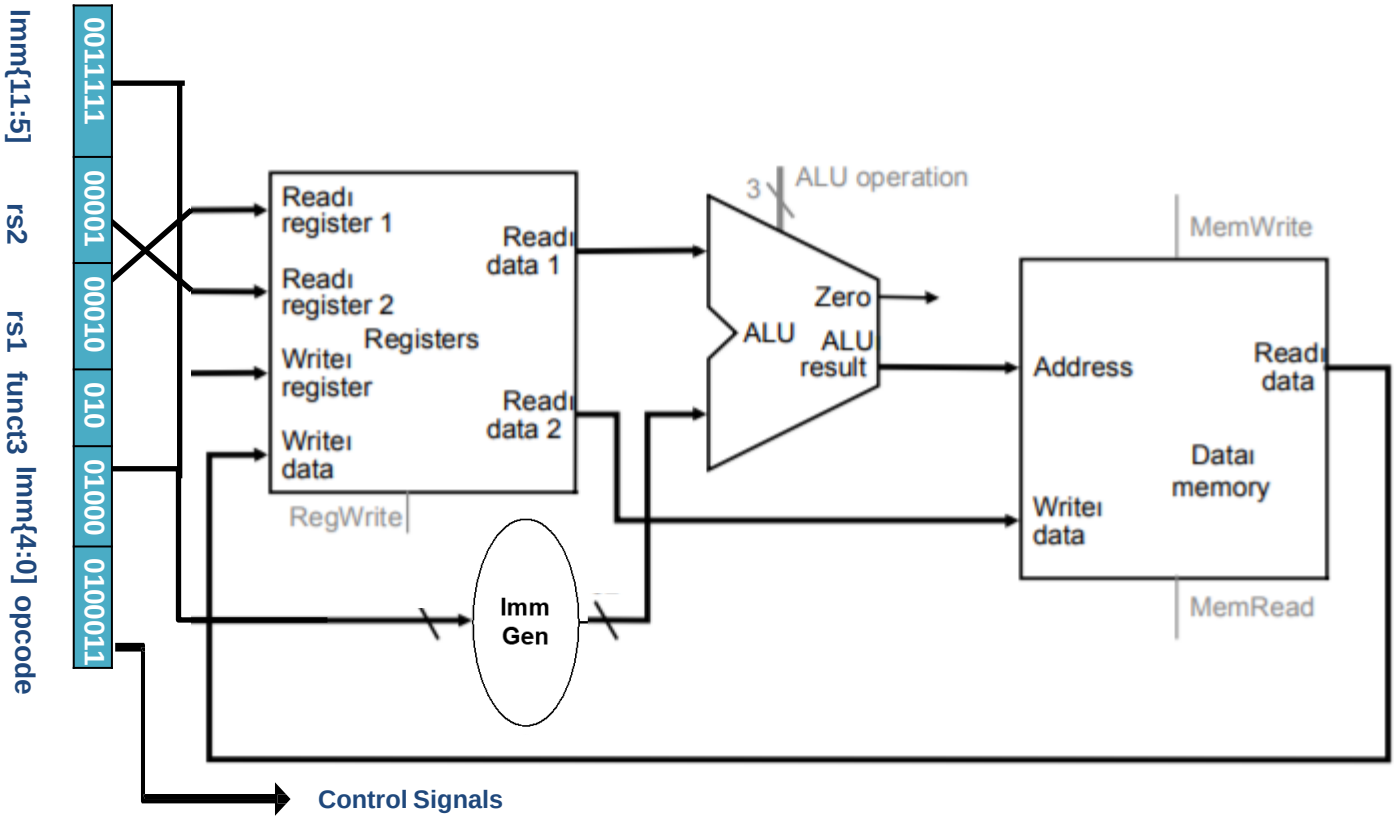
- fetch instruction
- select a register (rs1)
- calculate address, need ALU
- access memory (read memory)
- write register file (rd)

Put It Together for Load/Store



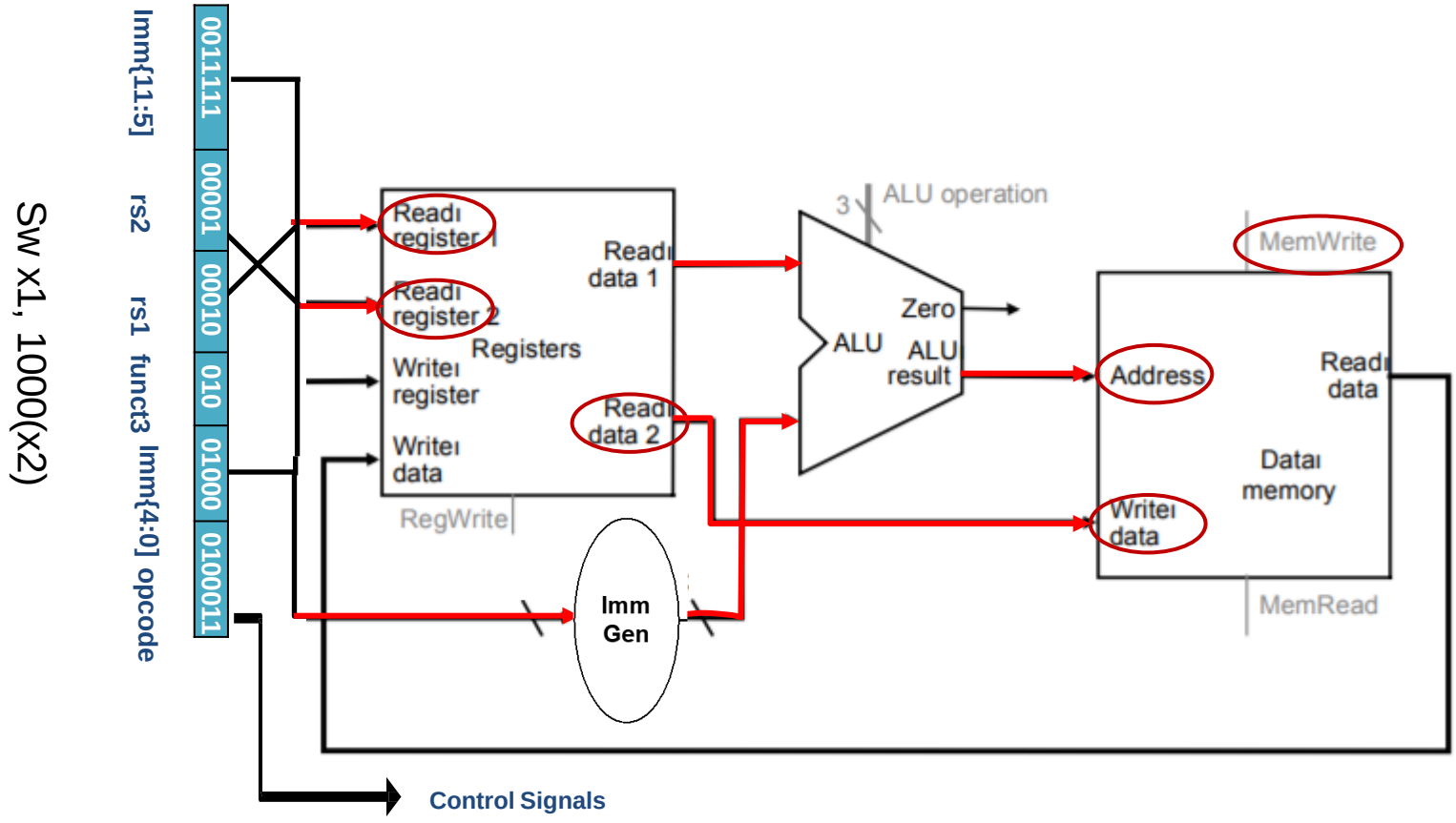
Store Example – sw X1, 1000(X2)

Sw x1, 1000(x2)



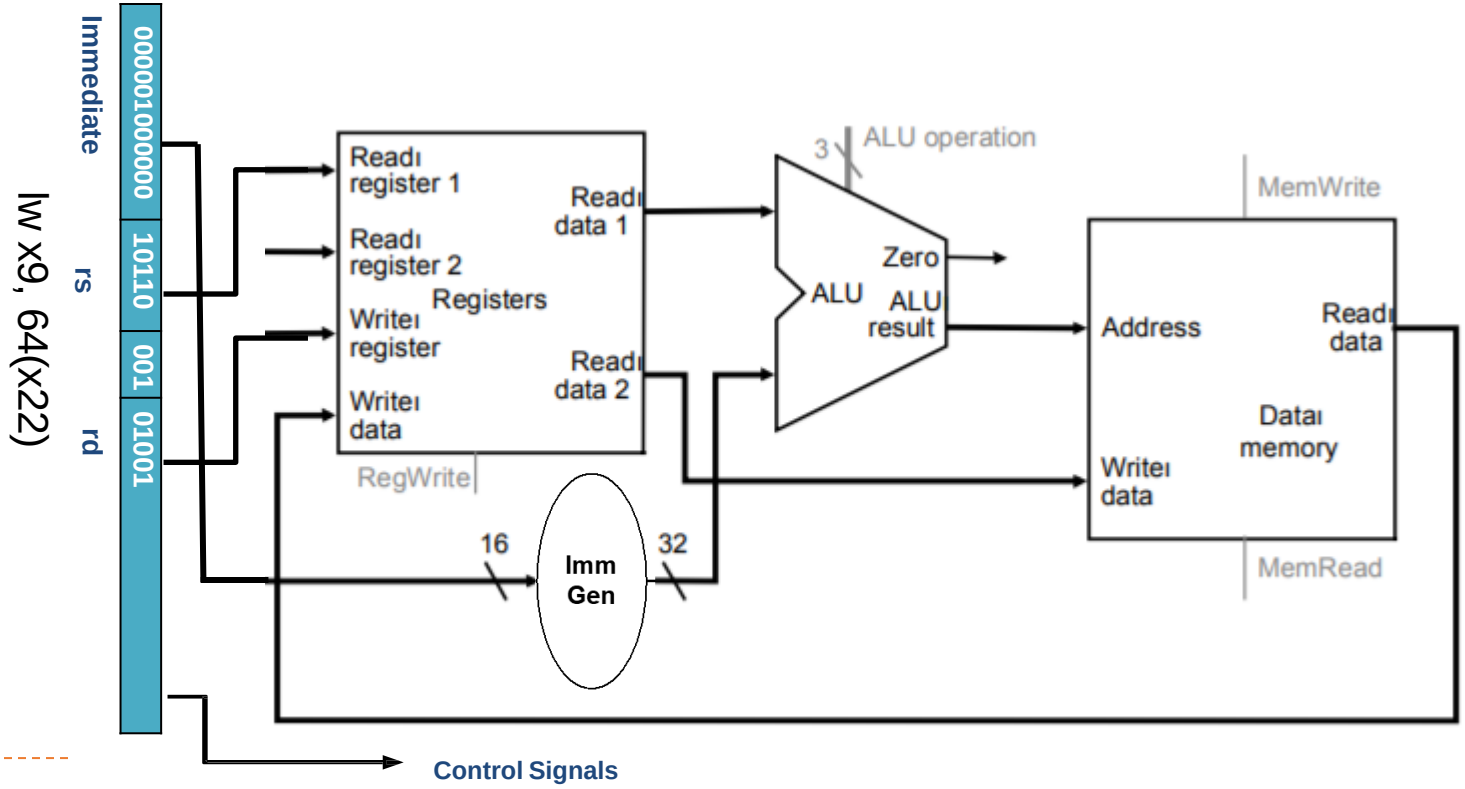
Datapath for Store

- A subset of all components and paths



Load Example – lw x9, 64(x22)

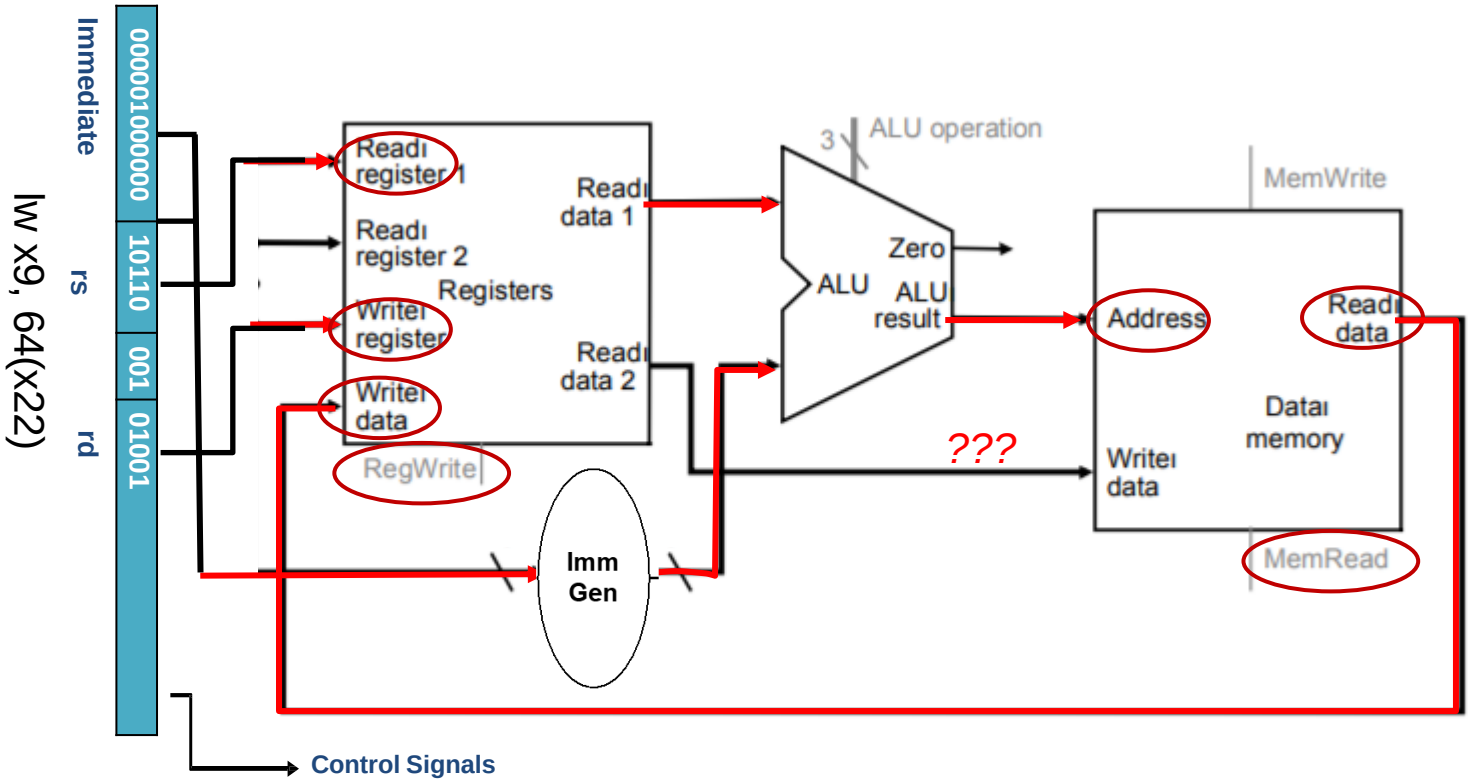
- Rs = 22, rd = 9
  - 22 = 10110_2 9 = 01001_2
- 64 = 000001000000_2, opcode = 0000011_2



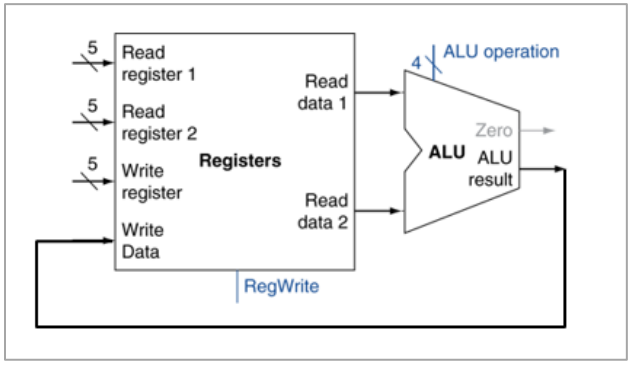
Datapath for Load

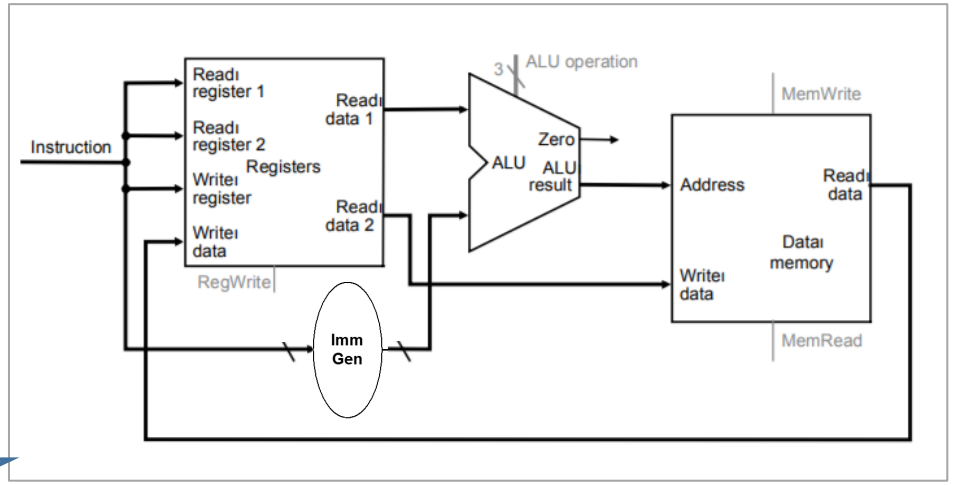
- A different subset of components and paths

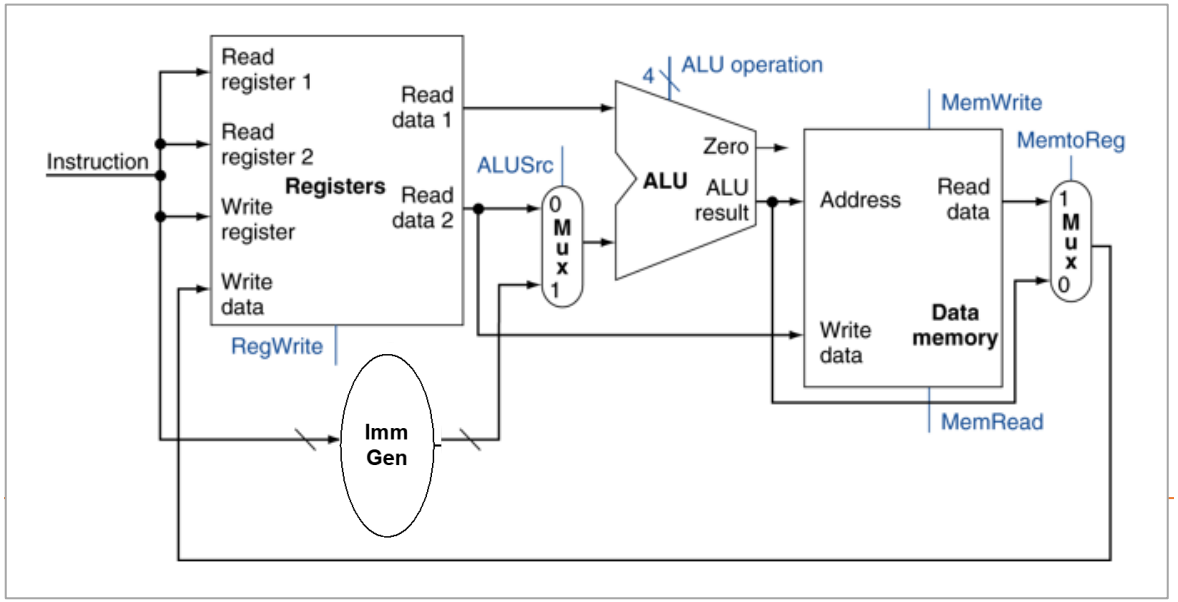
What happens to components/paths not used?



Merge R-Type/Load/Store Datapath





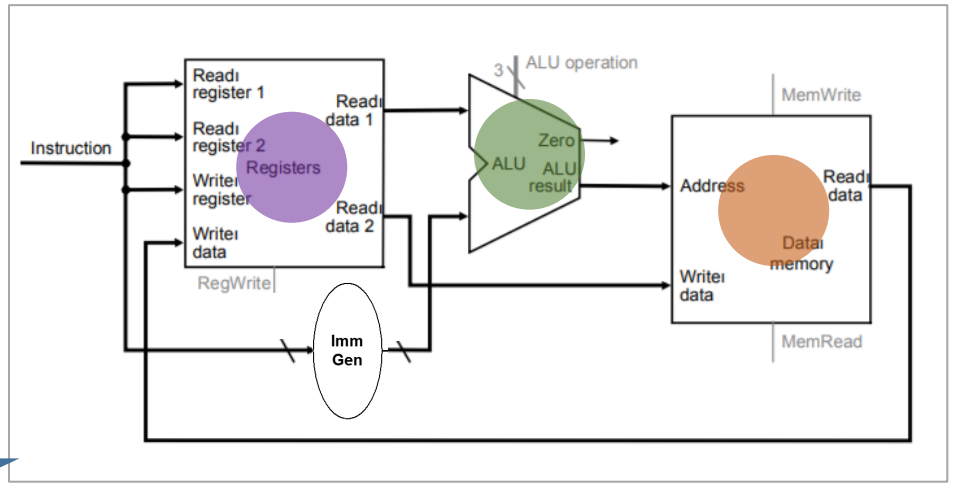


#1: Identified common components and paths.

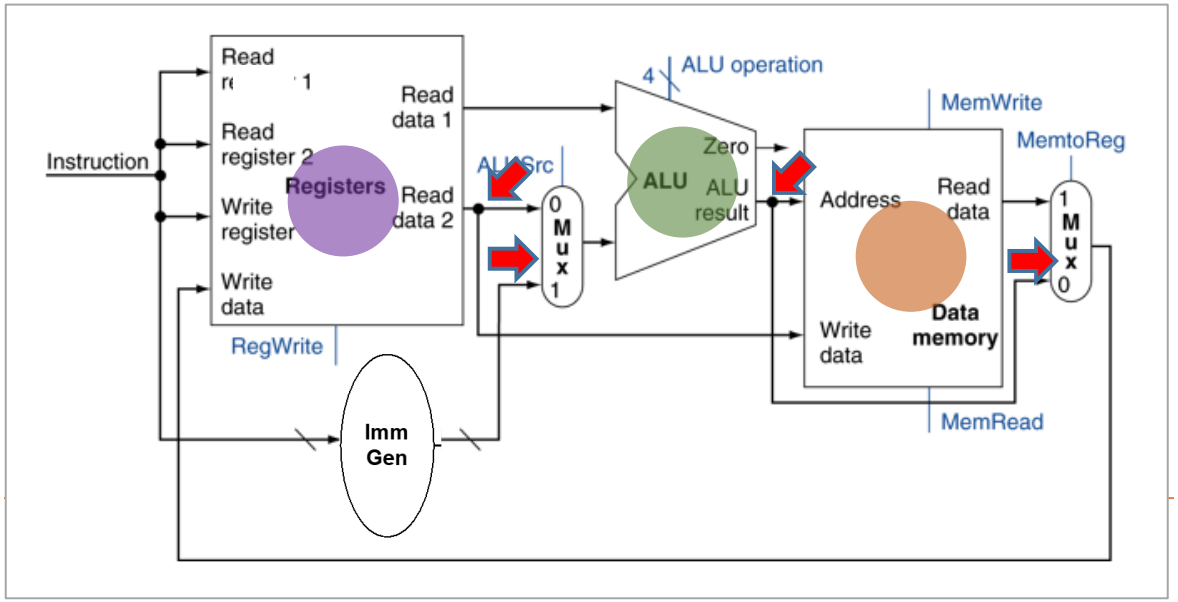
Merge R-Type/Load/Store Datap

#2: Use MUX to join incoming sources. Control signals control behavior.

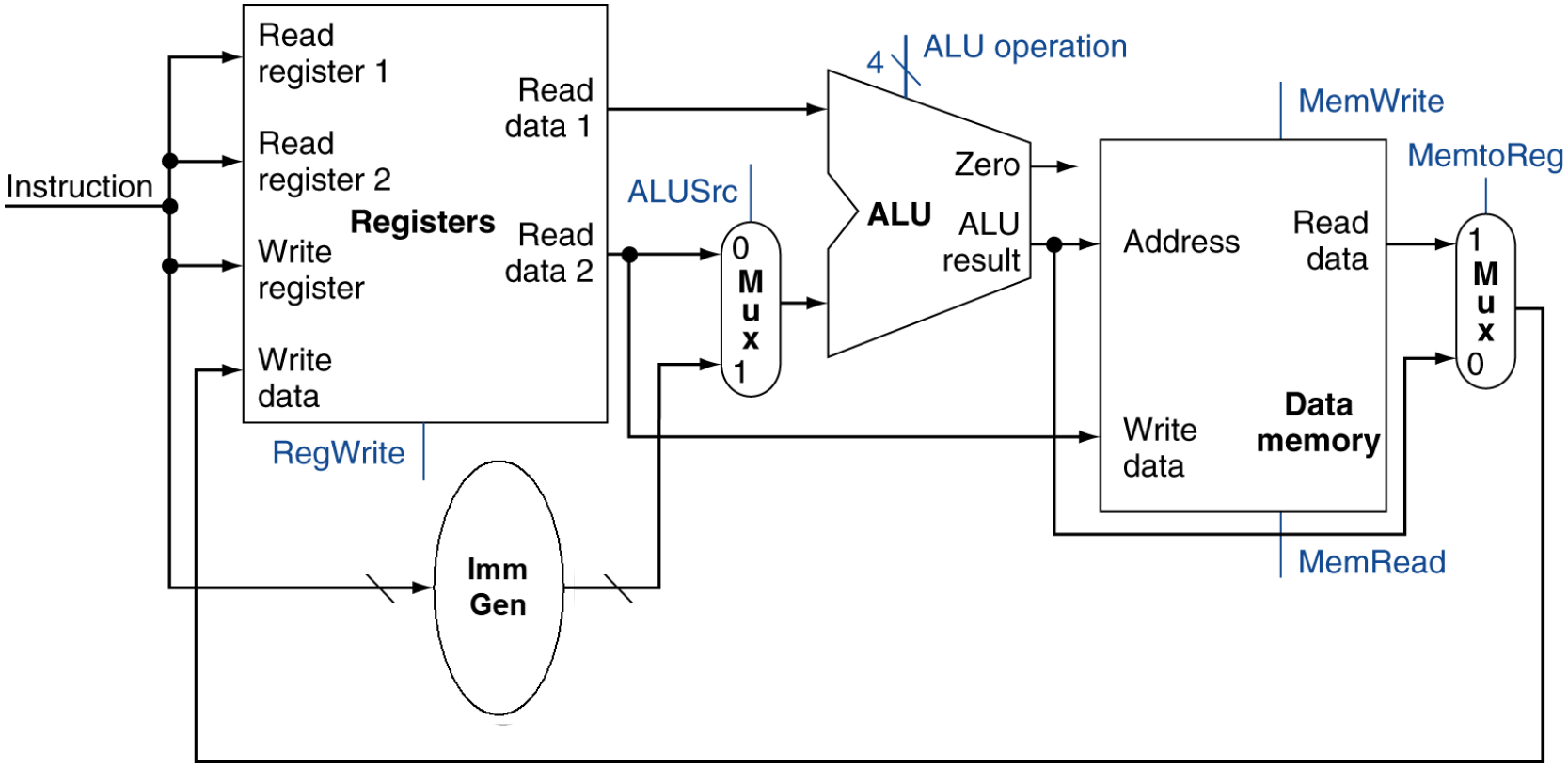
+



#3: Use splitting to forward data to multiple destinations



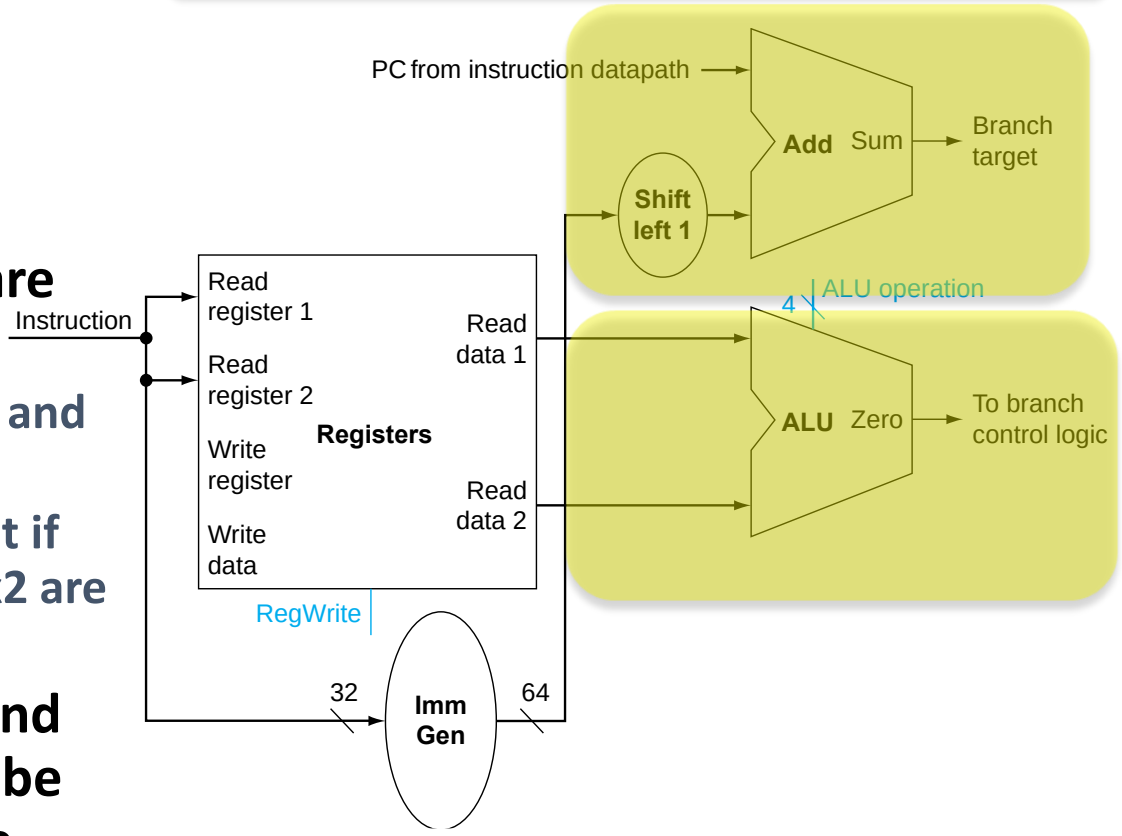
Merged R-Type/Load/Store Datapath



Branch Instructions

- Read register operands
  - x1 and x2
1. **Calculate target address (pc + offset * 2)**
   - **Shift left 1 places**
     - **Offset * 1**
   - **Add to PC**
2. **Branch condition: Compare operands**
   - **Use ALU, subtract ([x1] – [x2]) and check Zero output**
   - **The Zero output line is only set if x1 –x2 is 0, indicating x1 and x2 are equal**
- **Target address calculation and branch condition check can be performed at the same time**

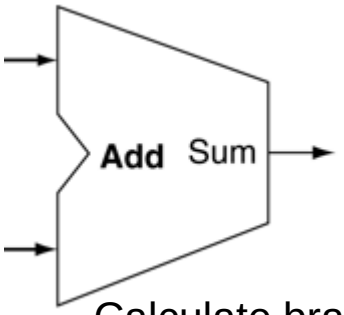
0x0FFE1238: beq x1, x2, offset



BEQ – Building Blocks

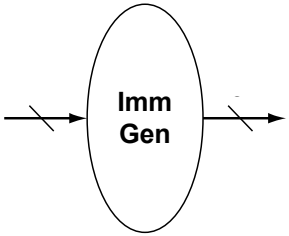
- What components should we use?

Read data

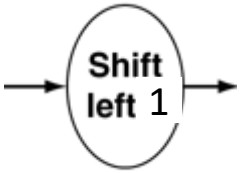


Calculate branch Target address

Determine whether or not the branch should be taken

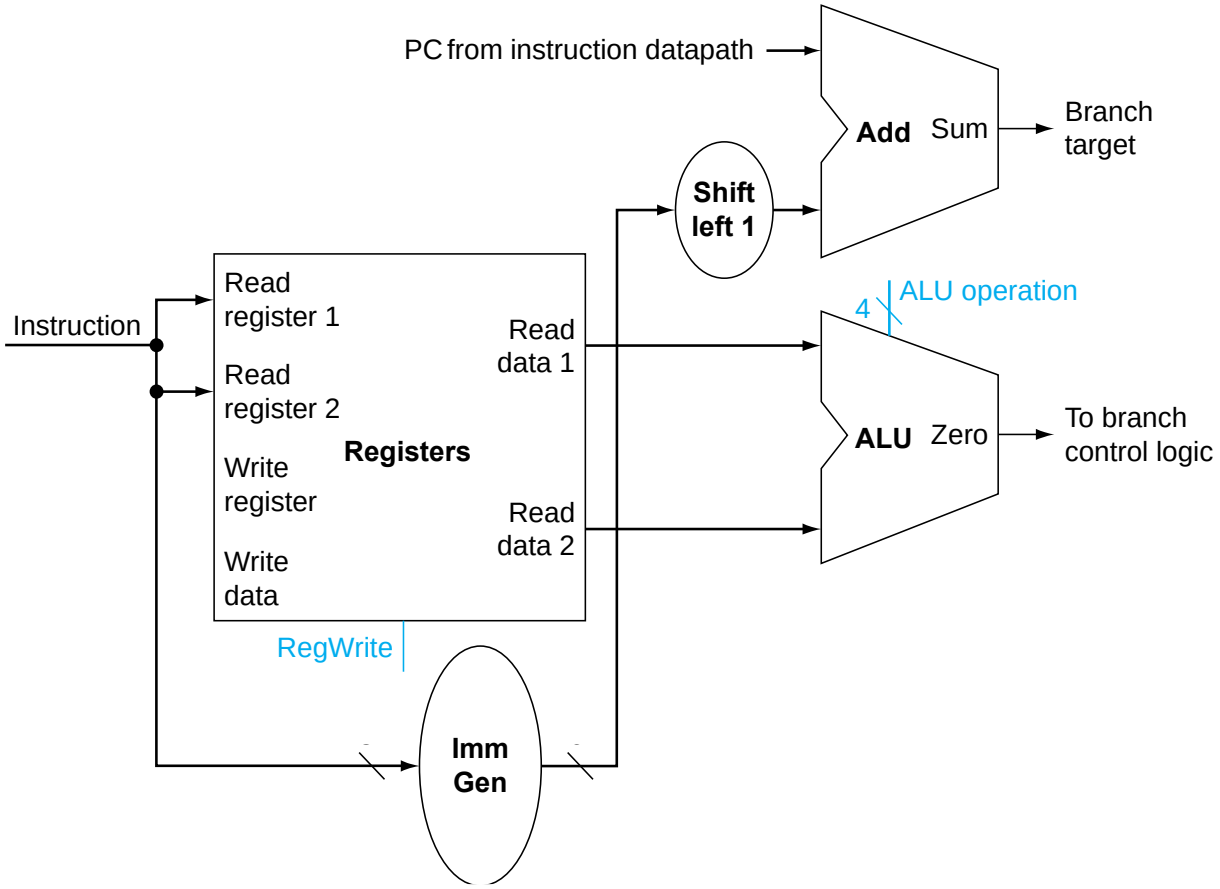


Immediate generation unit

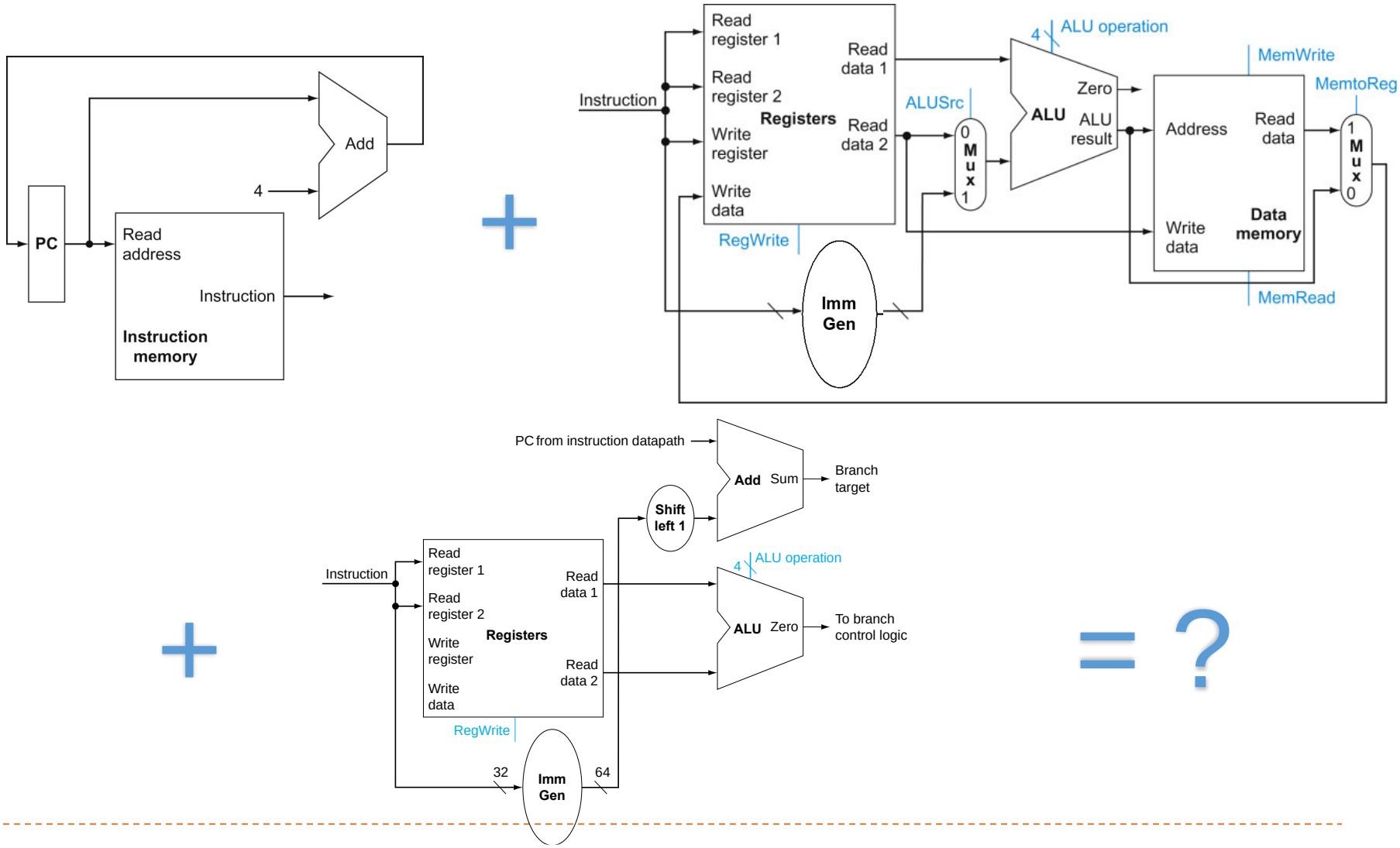


How to implement "shift left 1" logic?

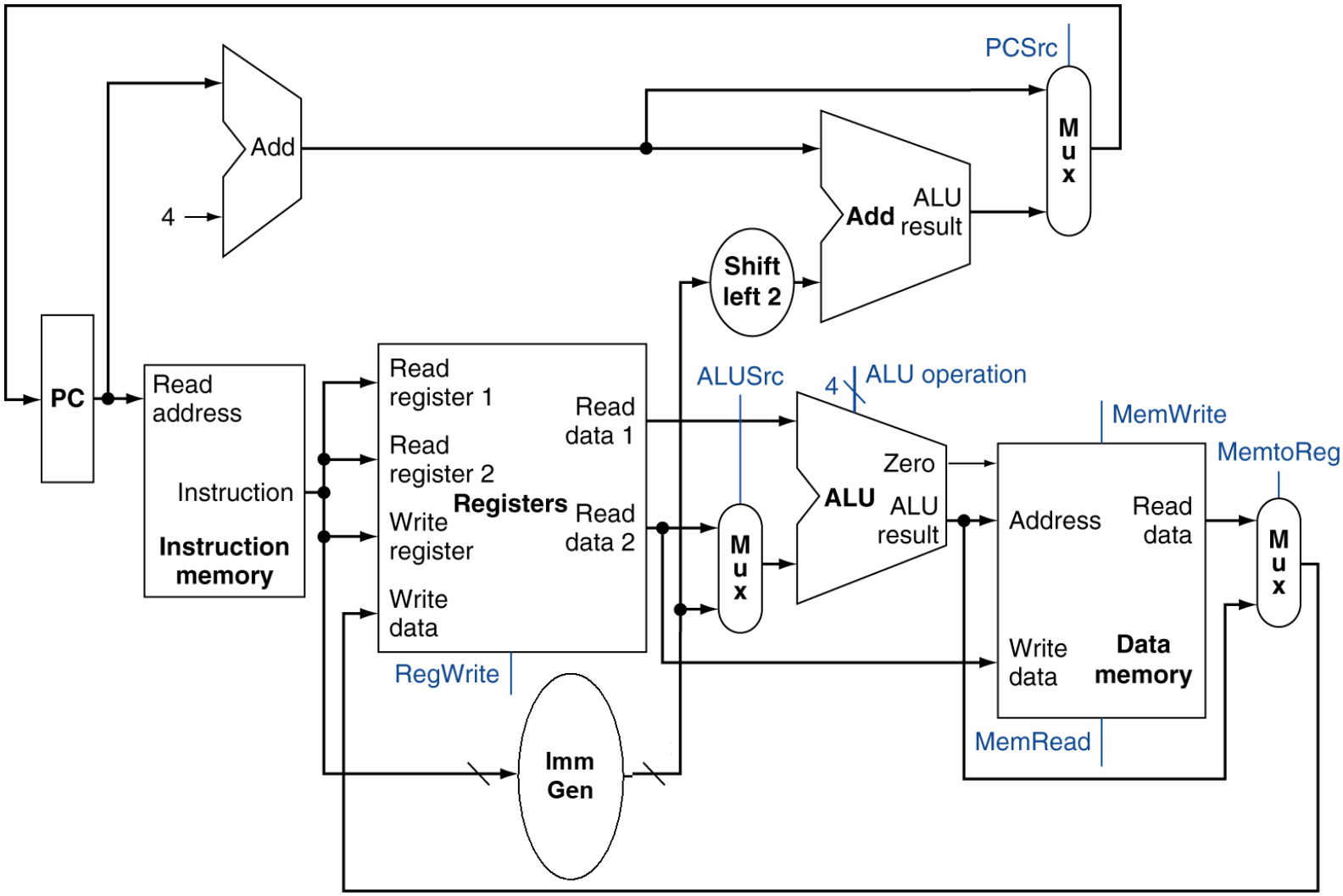
Datapath for Branch Instructions (BEQ)



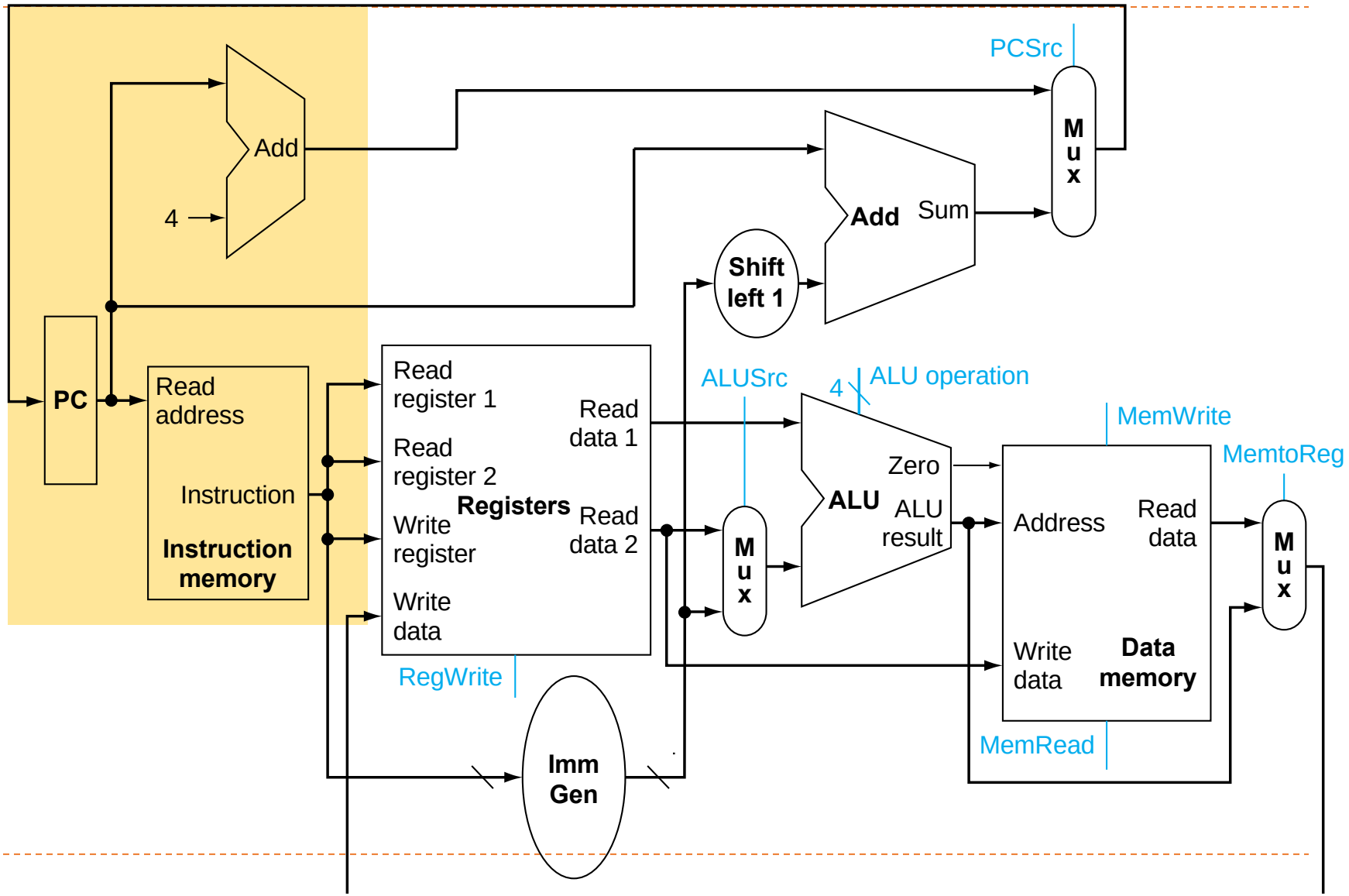
Merge Them All!



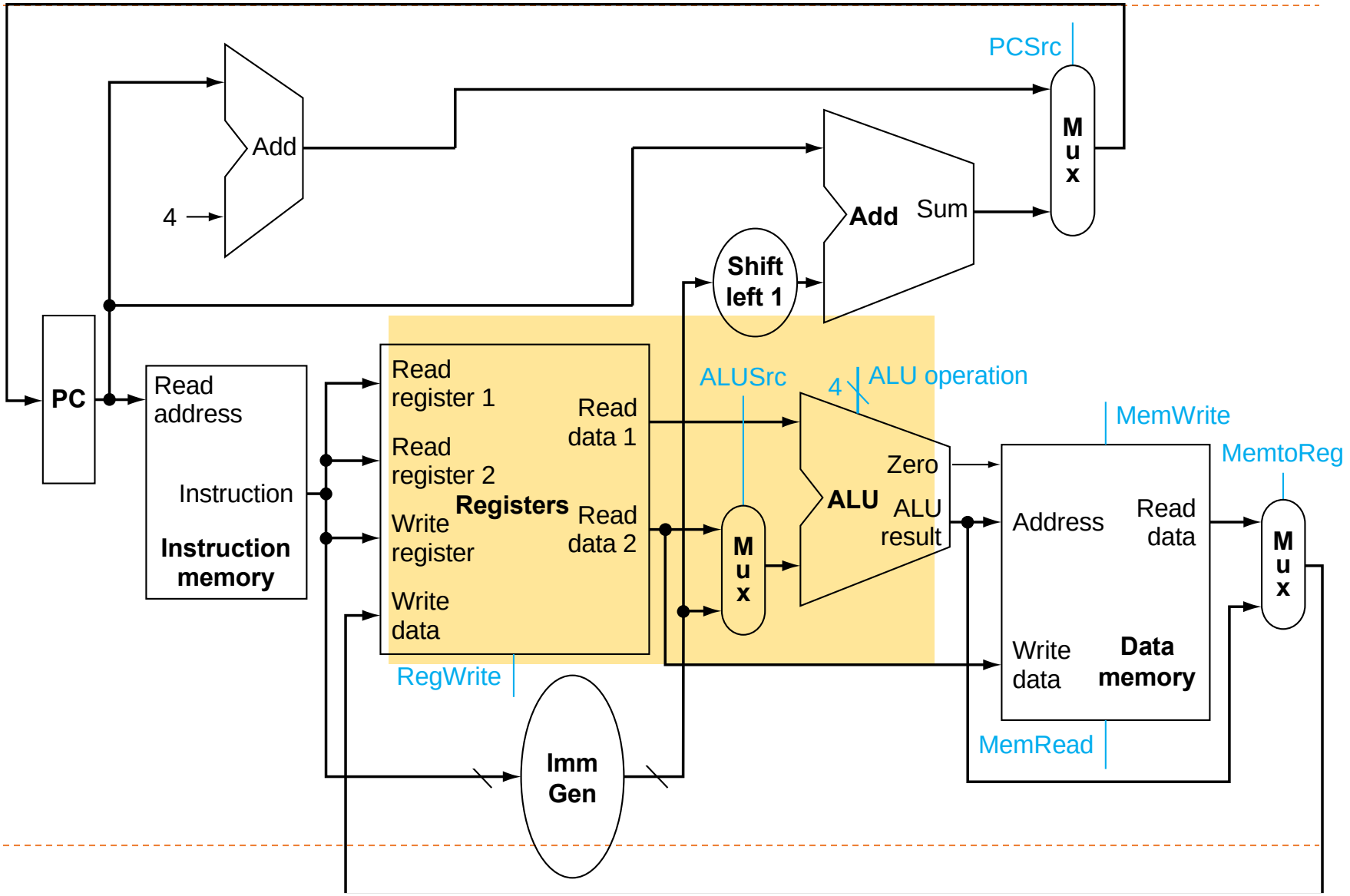
Full Datapath – Merge Them All!



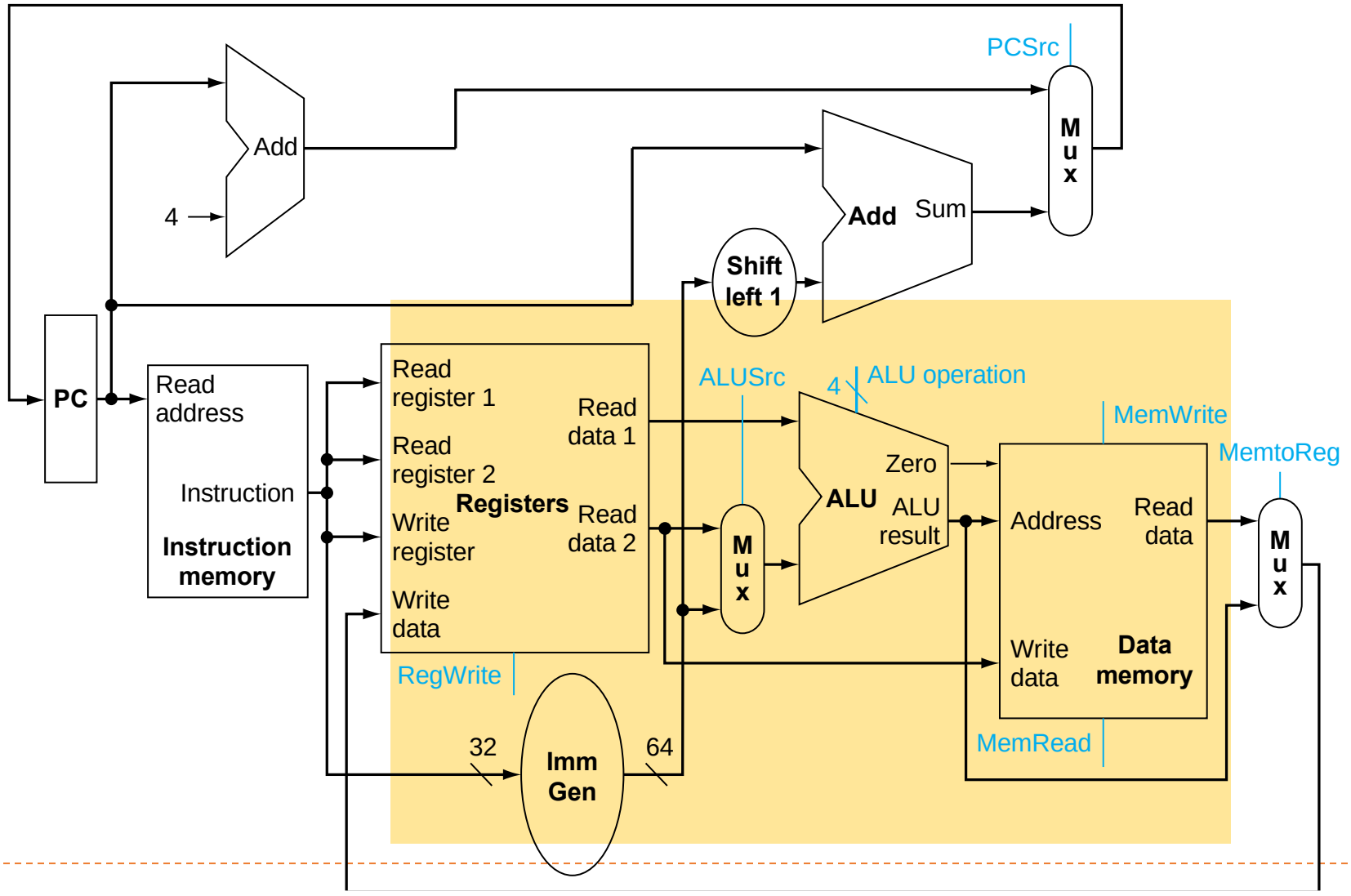
Full Datapath – PC and Instruction Fetching



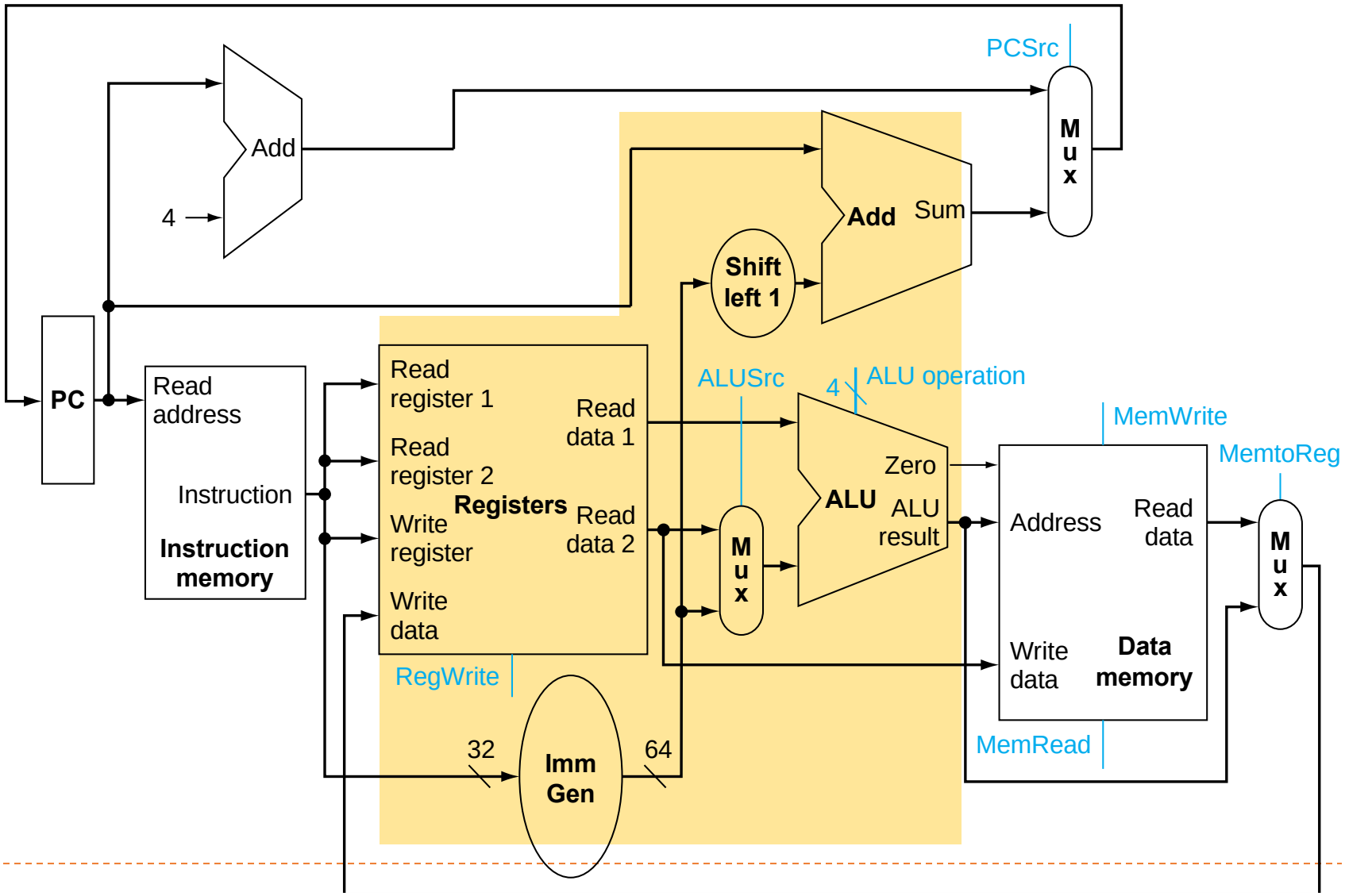
Full Datapath – R-Format



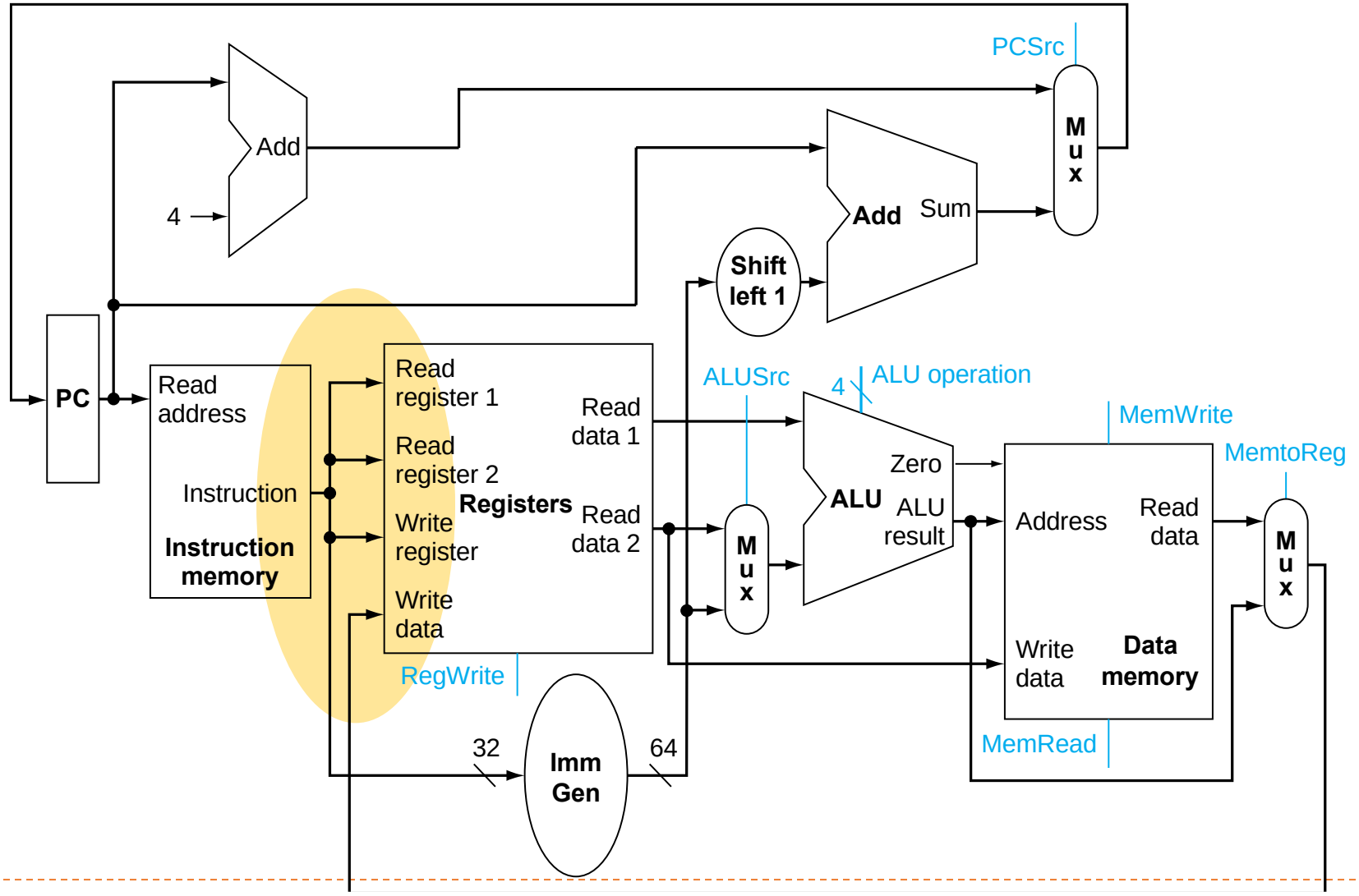
Full Datapath – Load/Store



Full Datapath – Branch (BEQ)



Something Still Fuzzy Here …

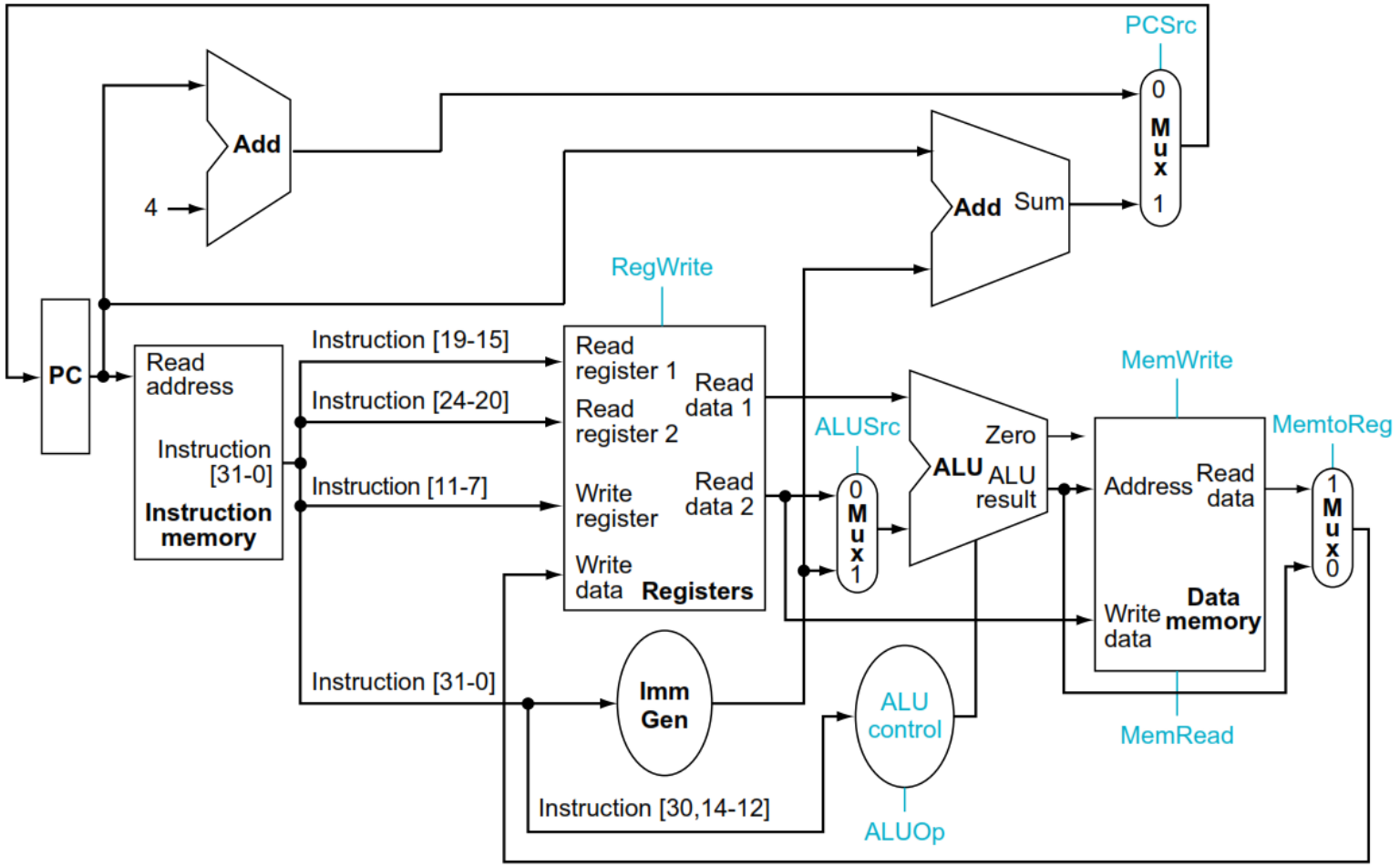


Four Different Instruction Formats

Name (Bit position)	Fields 31:25	24:20	19:15	14:12	11:7	6:0
(a) R-type	funct7	rs2	rs1	funct3	rd	opcode
(b) I-type	immediate[11:0]		rs1	funct3	rd	opcode
(c) S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode
(d) SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode

- The first register operand is always in bit positions 19:15 (rs1)
- The second register operand is always in bit positions 24:20 (rs2) for R, S, and branch instruction
- The destination register is always in bit positions 11:7 (rd) for R type and load instruction

Final Datapath with Control Signals and Multiplexors in RSIC-V



Single-cycle Controls

Now we have a complete datapath for our simple RISC-V subset – we will show the whole diagram in just a couple of minutes. Before that, we will add the control.

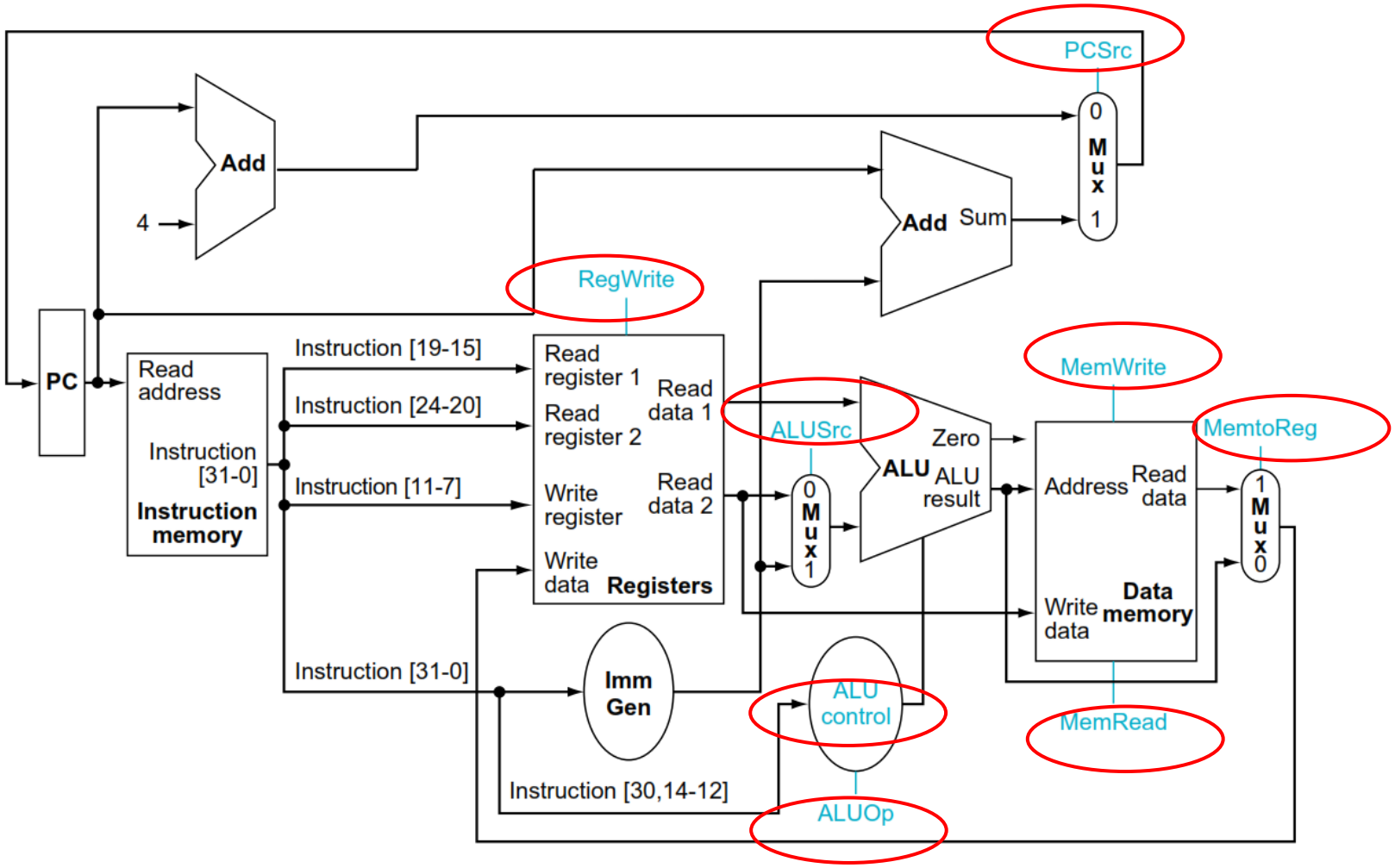
The *control unit* is responsible for taking the instruction and generating the appropriate signals for the datapath elements.

Signals that need to be generated include

- Operation to be performed by ALU.
- Whether register file needs to be written.
- Signals for multiple intermediate multiplexors.
- Whether data memory needs to be written.

For the most part, we can generate these signals using only the *opcode* and *funct* fields of an instruction.

Where are the Control Signals?

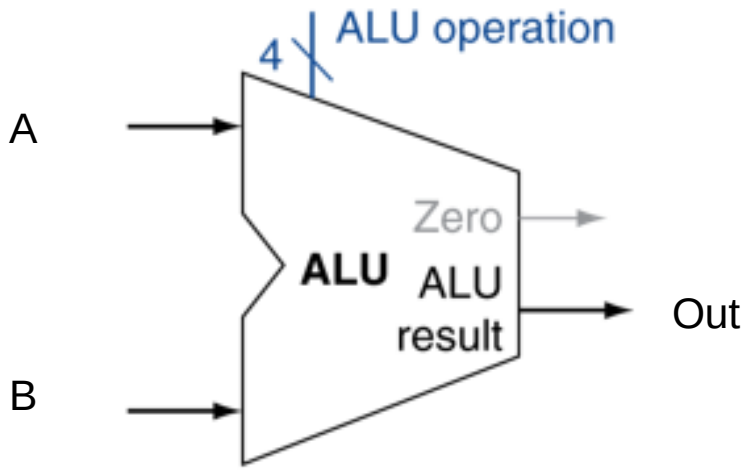


ALU Operations for subset RISC-V

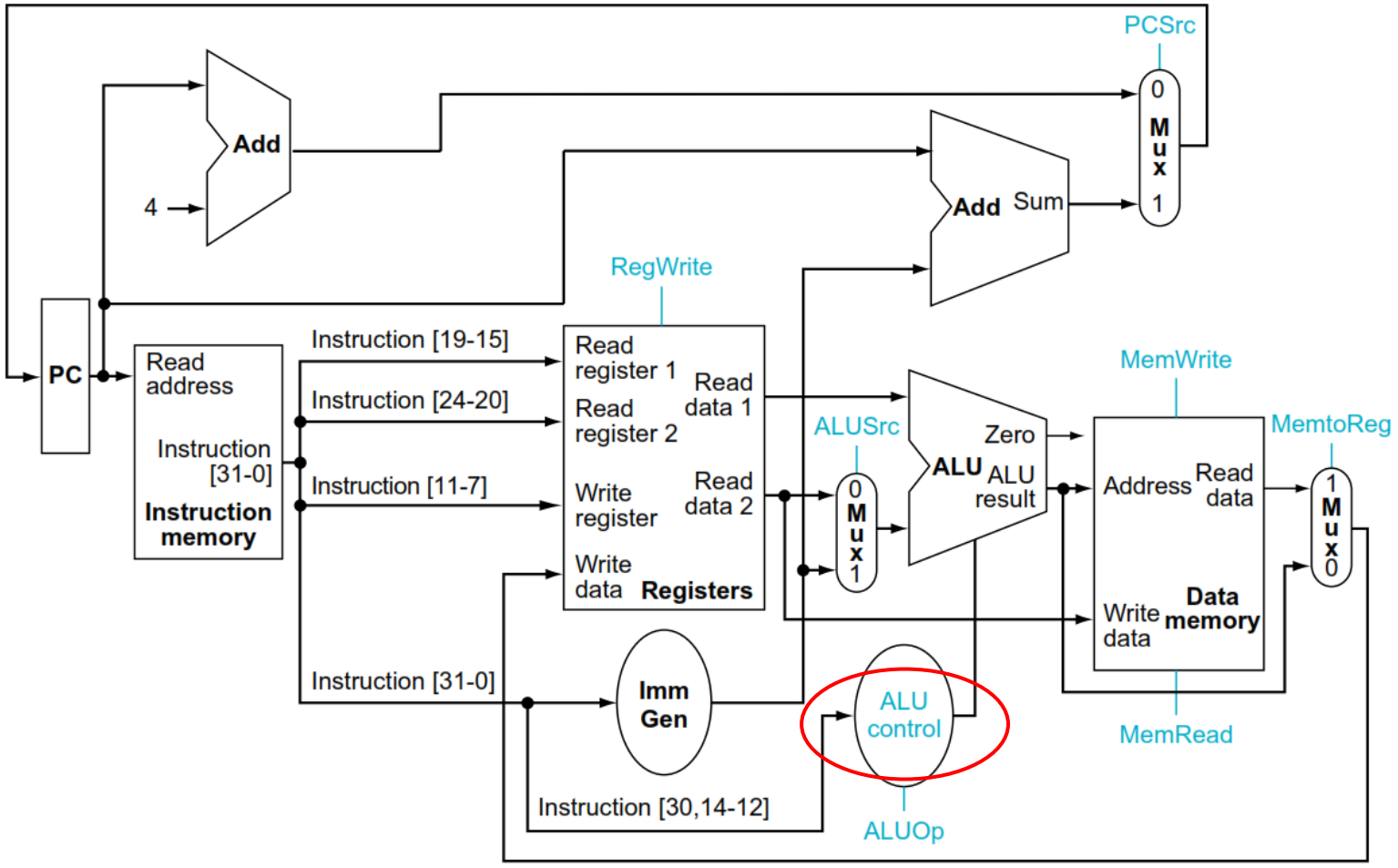
opcode	Operation	ALU function
lw	load word	add
sw	store word	add
beq	branch equal	subtract (zero flag)
R-type	add	add
	subtract	subtract
	AND	AND
	OR	OR
	set-on-less-than	set-on-less-than

Set-on-less-than ALU Operation

- Function
  - If A < B, then Out = 1
  - Otherwise Out = 0
- How to implement it?
  - Perform subtraction, A – B
  - Route sign to Out[0]
  - Set all other bits Out[31:1] to 0



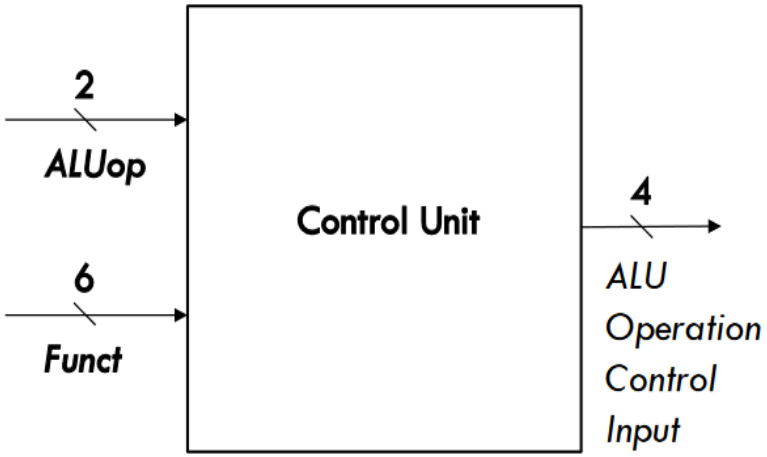
ALU control


PCSrc
Add
4
0
M
u
x
1
Add
Sum
RegWrite
PC
Read address
Instruction [31-0]
Instruction memory
Instruction [19-15]
Instruction [24-20]
Instruction [11-7]
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
ALUSrc
Zero
ALU
ALU result
MemWrite
MemtoReg
Address
Read data
Write data
Data memory
MemRead
Instruction [31-0]
Imm Gen
ALU control
Instruction [30,14-12]
ALUOp

ALU control Lines

How do we set these control lines? Consider the control unit below.

- The 2-bit *ALUop* input indicates whether an operation should be add (00) *for loads and stores*, subtract (01) *for beq*, or determined by the funct input (10).
  - Case 00 and 01 are determined by the op field
  - Case 10 is used by R-format instructions
- The 6-bit *Funct* input corresponds to the funct field of R-format instructions. Each unique funct field corresponds to a unique set of ALU control input lines.



How to generate those control signals?

Based on the opcode, func3 and func7 fields of an instruction word

Encoding Review:

R-Format Instruction Encoding (AL Instructions)

http://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf#page=116

RV32I Base Instruction Set

0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

Arithmetic instructions

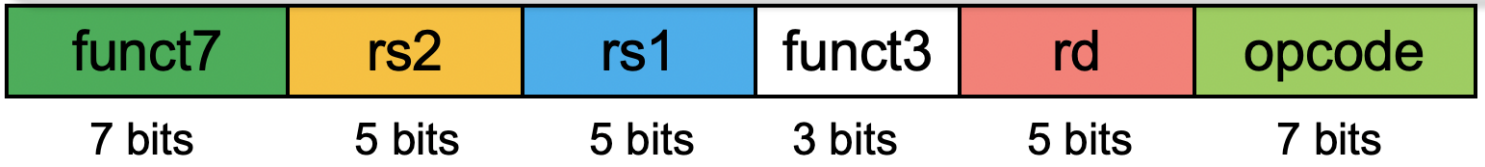
RV64I Base Instruction Set (in addition to RV32I)

0000000	rs2	rs1	000	rd	0111011	ADDW
0100000	rs2	rs1	000	rd	0111011	SUBW
0000000	rs2	rs1	001	rd	0111011	SLLW
0000000	rs2	rs1	101	rd	0111011	SRLW
0100000	rs2	rs1	101	rd	0111011	SRAW

RV32M Standard Extension

0000001	rs2	rs1	000	rd	0110011	MUL
0000001	rs2	rs1	001	rd	0110011	MULH
0000001	rs2	rs1	010	rd	0110011	MULHSU
0000001	rs2	rs1	011	rd	0110011	MULHU
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU

Logic instructions



I-Format Instruction Encoding (AL and Load)

http://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf#page=116

Immediate arithmetic/logic **load instructions**

immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

imm[11:0]	rs1	000	rd	0000011	LB
imm[11:0]	rs1	001	rd	0000011	LH
imm[11:0]	rs1	010	rd	0000011	LW
imm[11:0]	rs1	100	rd	0000011	LBU
imm[11:0]	rs1	101	rd	0000011	LHU

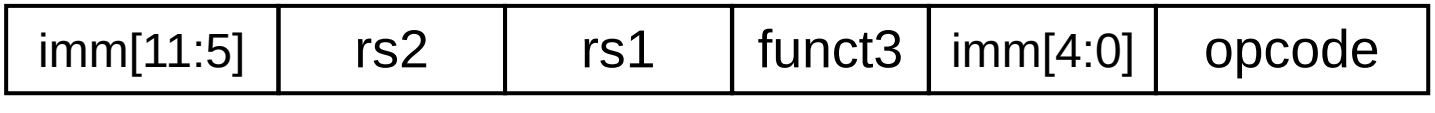
imm[11:0]	rs1	000	rd	0010011	ADDI
imm[11:0]	rs1	010	rd	0010011	SLTI
imm[11:0]	rs1	011	rd	0010011	SLTIU
imm[11:0]	rs1	100	rd	0010011	XORI
imm[11:0]	rs1	110	rd	0010011	ORI
imm[11:0]	rs1	111	rd	0010011	ANDI
0000000 shamt	rs1	001	rd	0010011	SLLI
0000000 shamt	rs1	101	rd	0010011	SRLI
0100000 shamt	rs1	101	rd	0010011	SRAI

imm[11:0]	rs1	110	rd	0000011	LWU
imm[11:0]	rs1	011	rd	0000011	LD

imm[11:0]	rs1	000	rd	0011011	ADDIW

S-Format Instruction Encoding (Store)

http://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf#page=116



imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:5]	rs2	rs1	011	imm[4:0]	0100011	SD

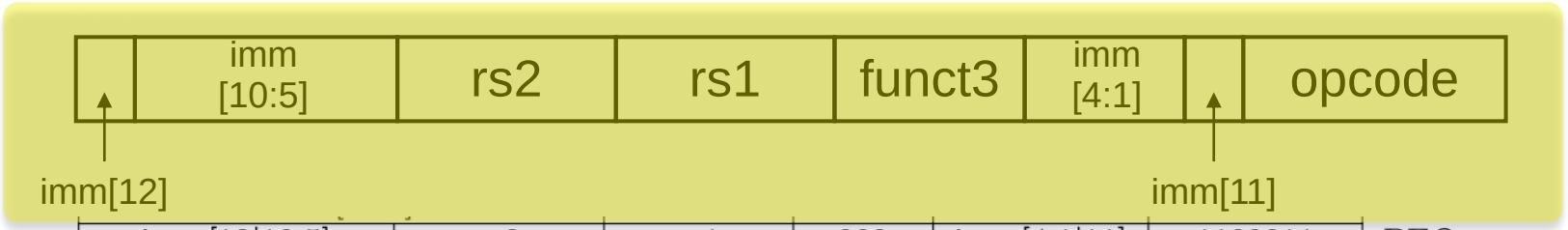
Store instructions

- Same opcode
- Func3 are different for different sizes of data
  - Byte, half-word, word, doubleword

SB-Format Encoding for Branch Instr (e.g. beq)

http://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf#page=116

- Branch instructions specify
  - Opcode, two registers, target address
  - Most branch targets are near branch, Forward or backward
- SB-Format instructions: beq x8, x9, 4



imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU

  - Same opcode, func3 are different for different branch instr
- PC-relative addressing
  - Branch target address is encoded as the offset off the the address of the branch instruction itself
  - Target address = PC (Branch address) + immediate × 4

Observation

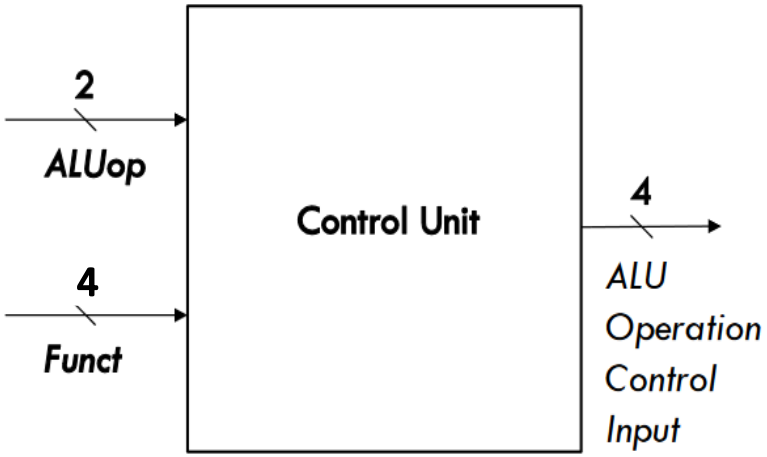
- Opcode are the same for each basic function category
  - R-format 32-bit AL
  - R-format 64-bit AL
  - I-format AL
  - Load (I-Format)
  - Store (S-Format)
  - Branch (SB-Format)

- Func3 and func7 are different for different operations with each categories
  - To determine the ALU action for the instructions
    - Add, sub, AND, or, etc.

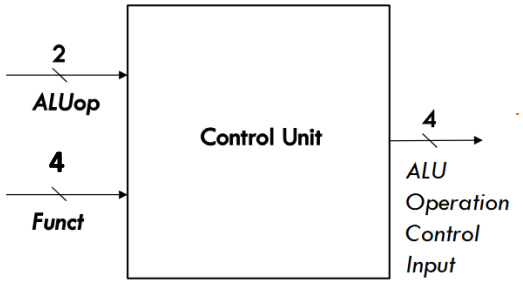
ALU control Lines

How do we set these control lines? Consider the control unit below.

- The 2-bit *ALUop* input indicates whether an operation should be add (00) *for loads and stores*, subtract (01) *for beq*, or determined by the funct input (10).
  - Case 00 and 01 are determined by the op field
  - Case 10 is used by R-format instructions
- The **4**-bit *Funct* input corresponds to the funct field of R-format instructions. Each unique funct field corresponds to a unique set of ALU control input lines.



Four Formats of Instructions



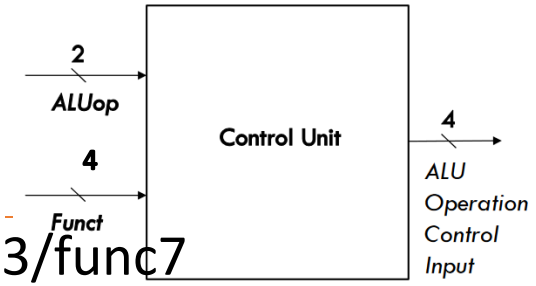
- ALU Control input = ALUOp + ALU action
  - **2-bit ALUOp determined by opcode**
  - **ALU action determinded by bit[30, 14-12] of func3/func7**

Name (Bit position)	Fields 31:25	24:20	19:15	14:12	11:7	6:0
(a) R-type	funct7	rs2	rs1	funct3	rd	opcode
(b) I-type	immediate[11:0]		rs1	funct3	rd	opcode
(c) S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode
(d) SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode

Instruction opcode	ALUOp	operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
ld	00	load doubleword	XXXXXXX	XXX	add	0010
sd	00	store doubleword	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract

The Truth Table for ALU Operation



- Control signals derived from instruction opcode/func3/func7
  - Nothing to do with operands (register or immediate)

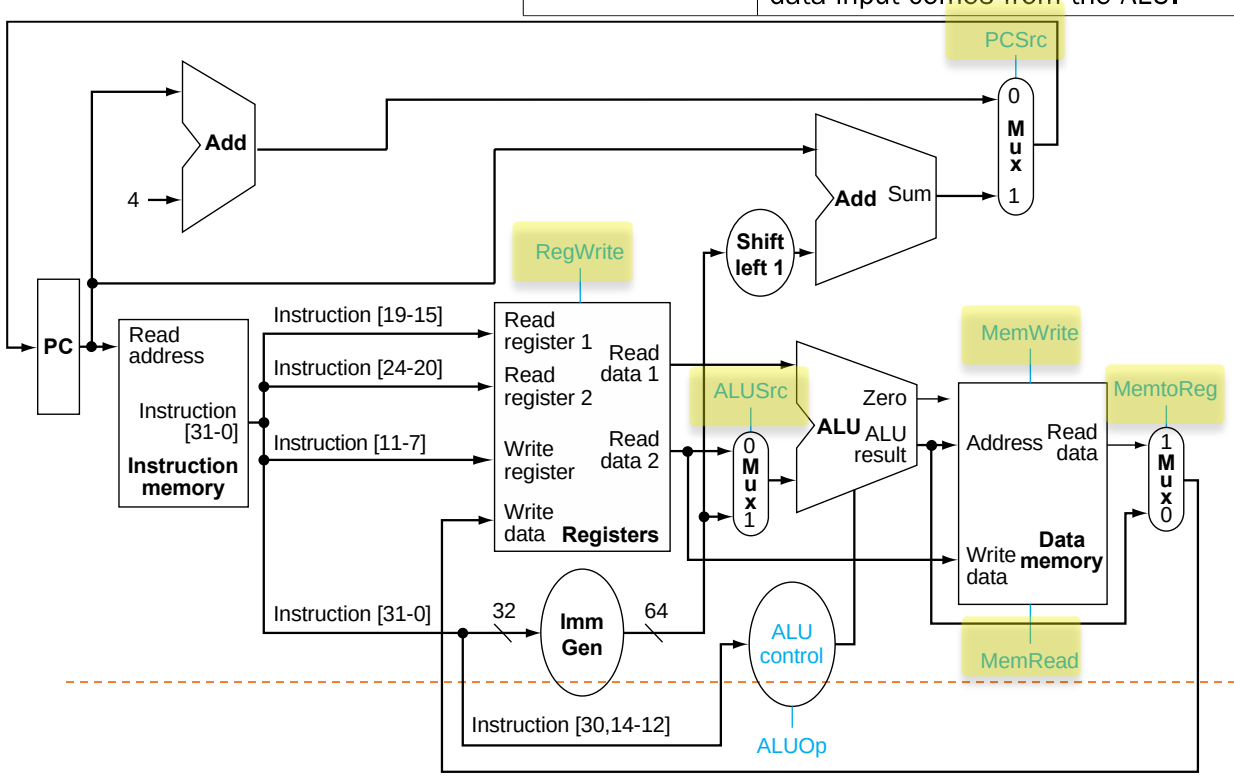
Instruction opcode	ALUOp	operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
ld	00	load doubleword	XXXXXXX	XXX	add	0010
sd	00	store doubleword	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

ALUOp		Funct7 field							Funct3 field			
ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[14]	I[13]	I[12]	Operation
0	0	X	X	X	X	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	X	X	X	X	0110
1	X	0	0	0	0	0	0	0	0	0	0	0010
1	X	0	1	0	0	0	0	0	0	0	0	0110
1	X	0	0	0	0	0	0	0	1	1	1	0000
1	X	0	0	0	0	0	0	0	1	1	0	0001

- The design of those logics can be done with programmable logic array (PLA)

Six Control Signals

Signal name	Effect when deasserted	Effect when asserted
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, 12 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.



Control Signals

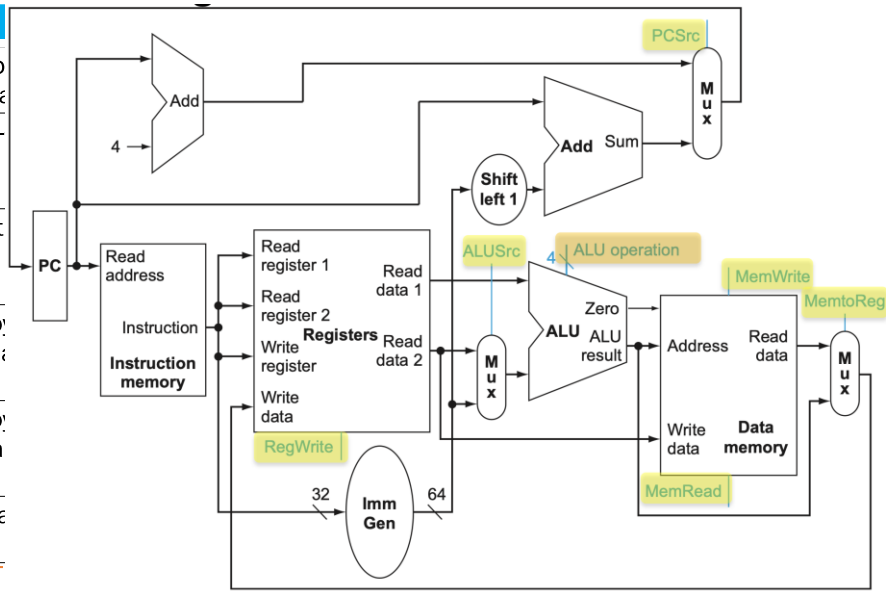
- 6 1-bit control:
  - PCSrc: Mux input to select PC+4 or PC+offset, for beq instruction to select next instruction
  - ALUSrc: Mux input to select input from rs2 or immediate, for R/I-type ALU and load instr
  - RegWrite: enable signal to enable write to register, for ALU, and load instr (write to register)
  - MemRead: enable signal to enable read from memory, for load instr
  - MemWrite: enable signal to enable write to memory, for store instr
  - MemtoReg: Mux input to select input to write to register from memory or ALU, for ALU and load instr
- 1 4-bit control: ALU operation
  - 2-bit ALUOp: for enabling certain input (Ainvert, Binvert, etc) of the ALU
  - 4-bit ALU Action: AL operation (add, AND, etc) to be performed by ALU

Setting of the 2-Bit ALUOp and the 6 1-bit Controls

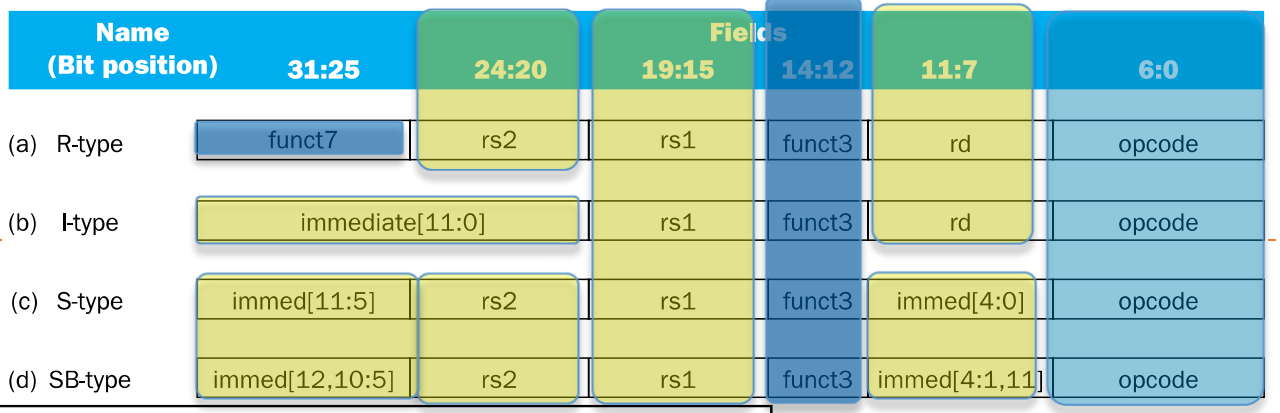
- **Are completely determined by the instruction opcode**
  - Check Figure 4.18 of the description

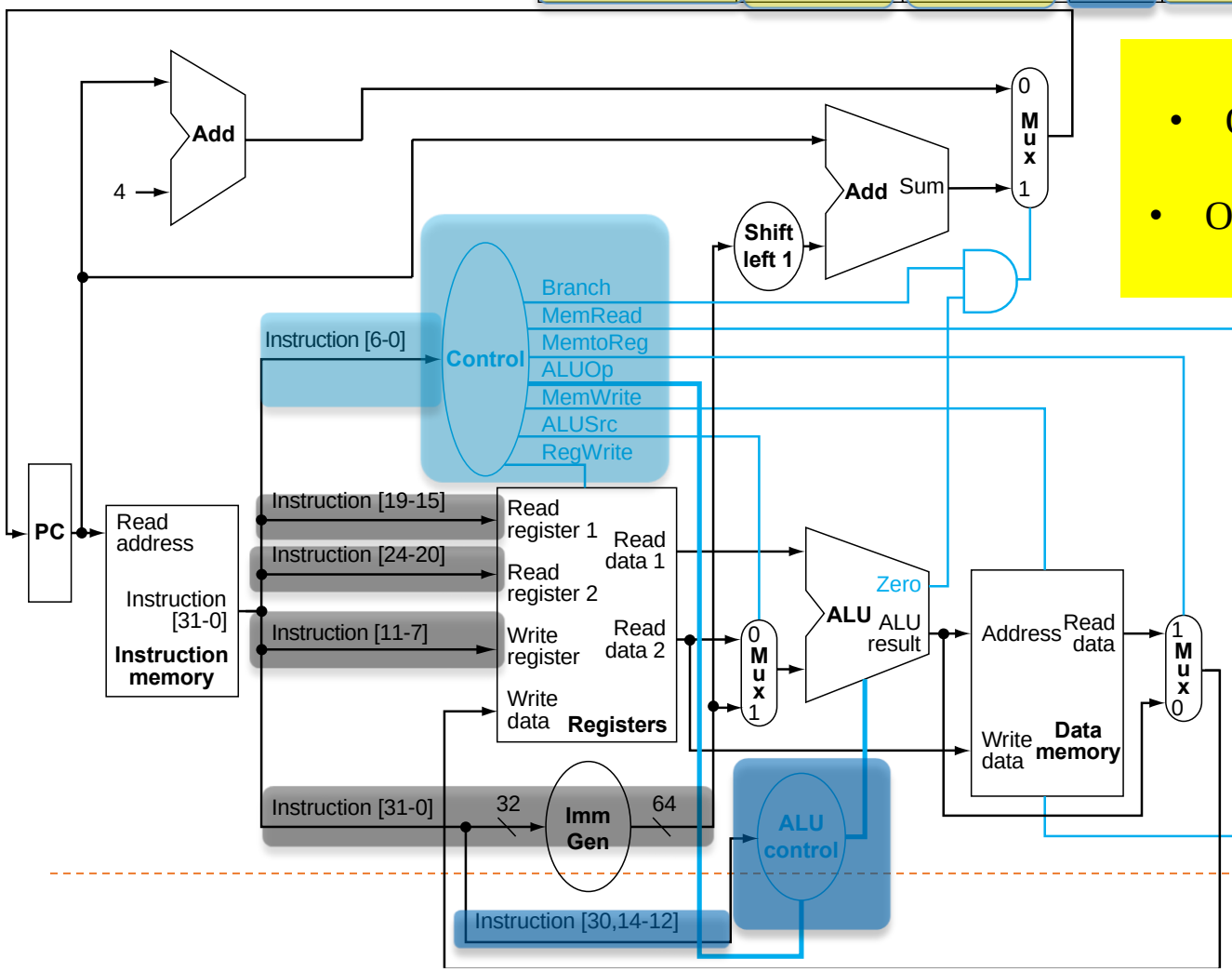
Instruction	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	0	0	1	0	0	0	1	0
ld	1	1	1	1	0	0	0	0
sd	1	X	0	0	1	0	0	0
beq	0	X	0	0	0	1	0	1

Signal name	Effect when deasserted	Effect when asserted
RegWrite	None.	The register on the Write register inp written with the value on the Write da
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign- 12 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of t that computes the branch target.
MemRead	None.	Data memory contents designated by address input are put on the Read da output.
MemWrite	None.	Data memory contents designated by address input are replaced by the va the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write da comes from the data memory.



Datapath With Control





Instruction Word:

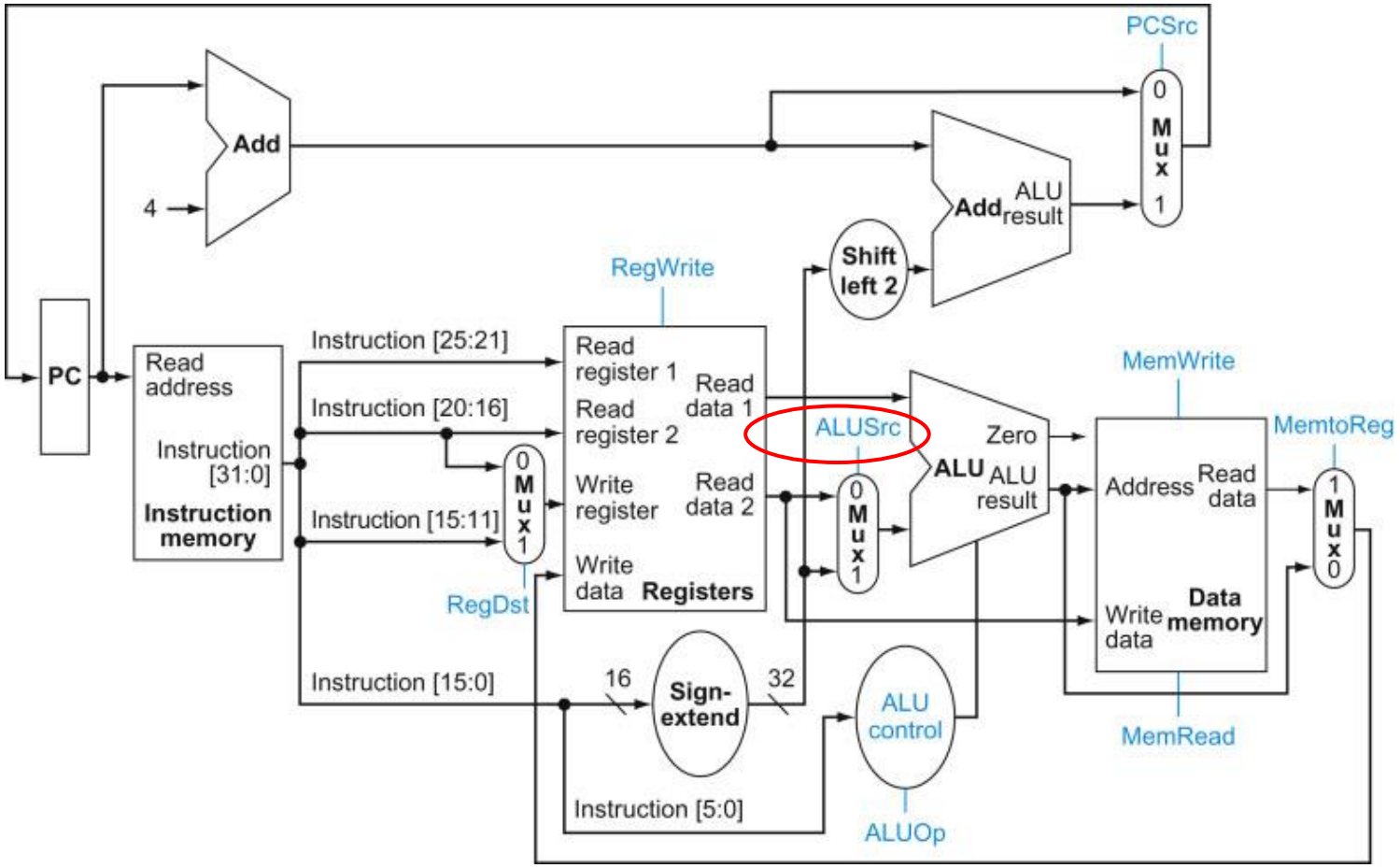
- Operation: opcode, func3 and func7) ➔ control logic
- Operands: register or immediate operands ➔ data path

The "Control" logic derives the 2-bit ALUOp and other six 1-bit control, solely based on Instruction[6-0] bits, which is the opcode of an instruction word.

"ALU Control" logic derives the 2-bit ALU action based on Instruction[30,14-12] bits (func3 and func7) and then combines the 2-bit ALUOp with the 2-bit ALUOp to create the 4-bit ALU input control.

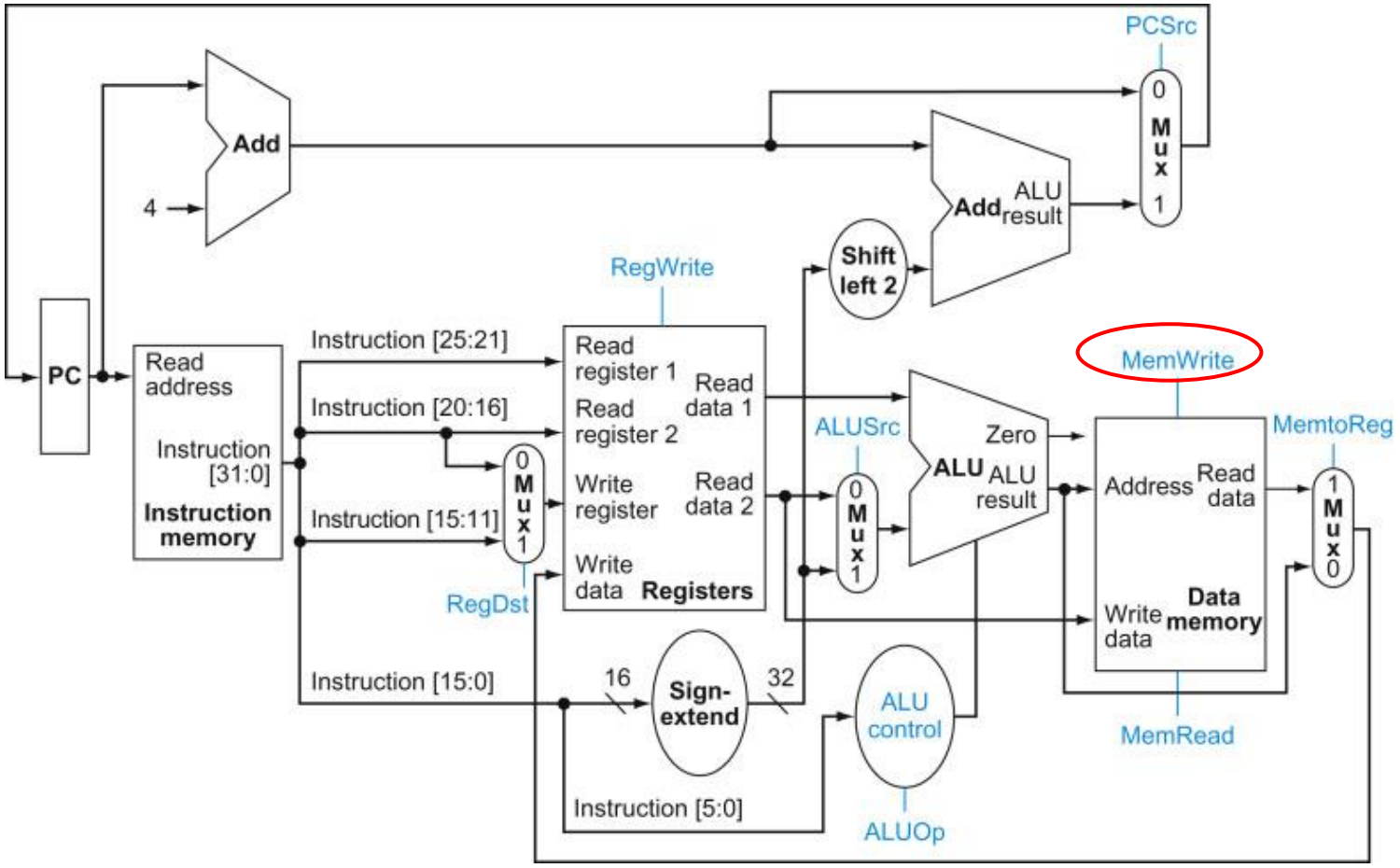
Control Signal - ALUSrc

Signal name	R-format	lw	sw	beq
ALUSrc	0	1	1	0



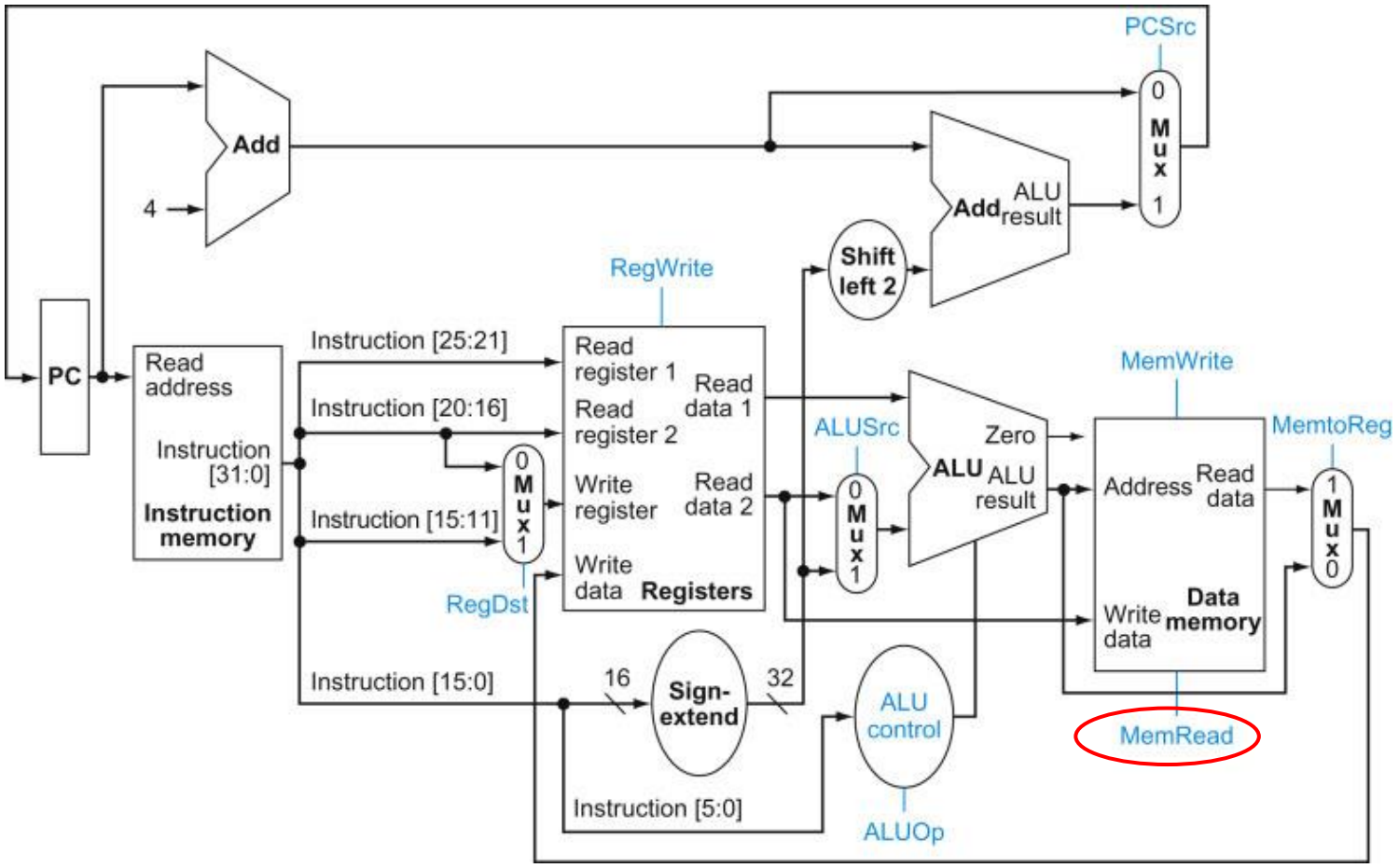
Control Signal - MemWrite

Signal name	R-format	lw	sw	beq
MemWrite	0	0	1	0



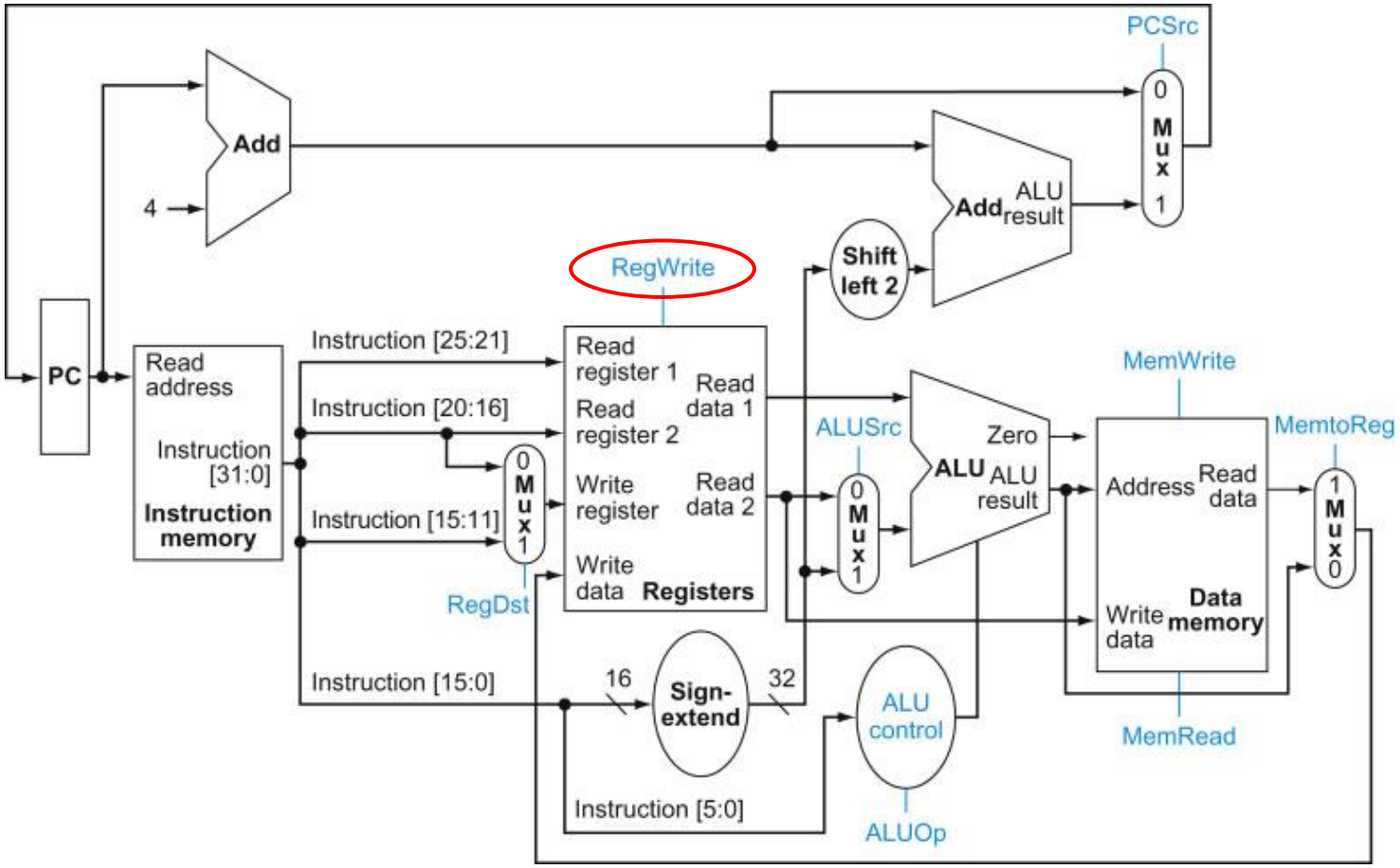
Control Signal - MemRead

Signal name	R-format	lw	sw	beq
MemRead	0	1	0	0



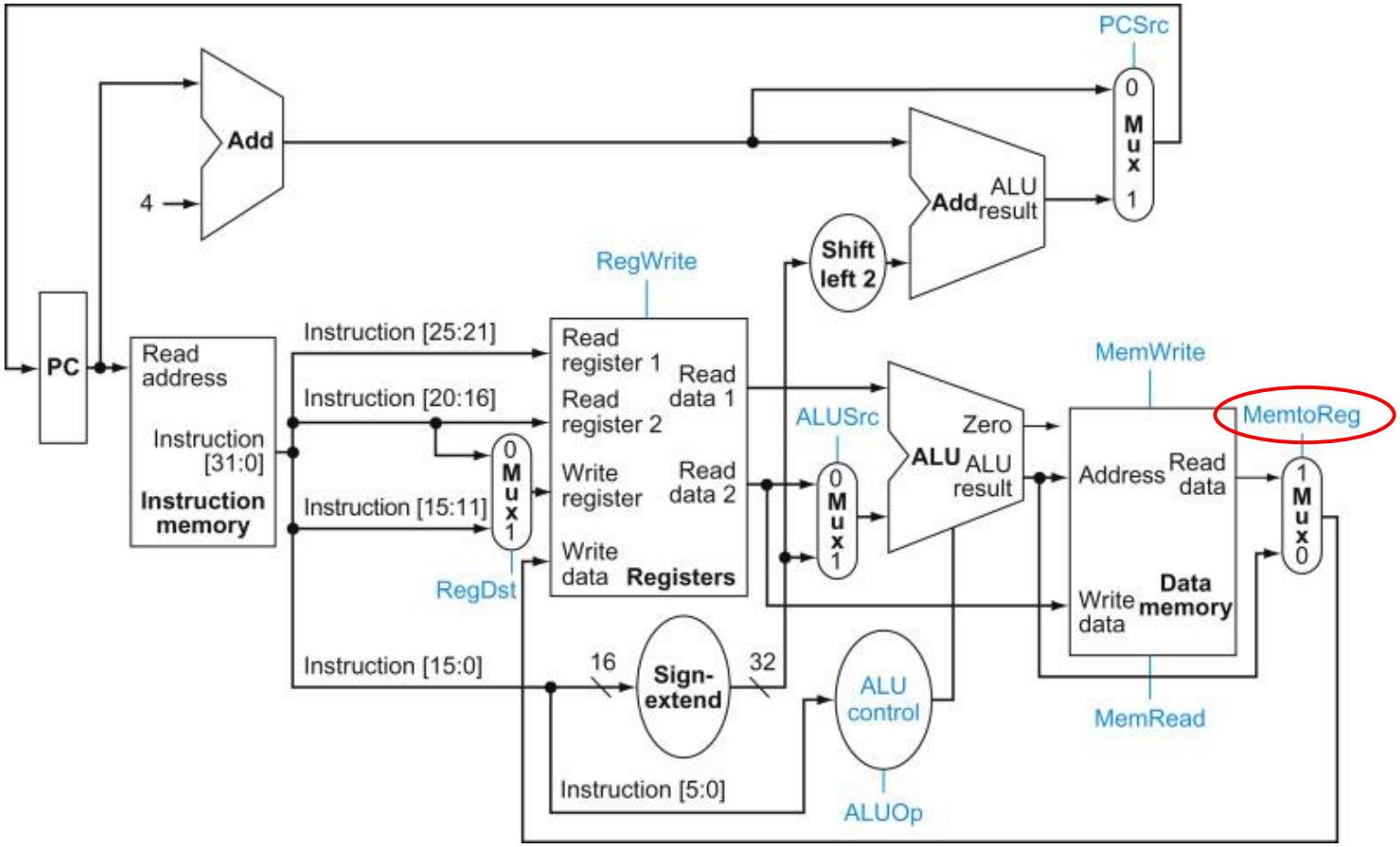
Control Signal - RegWrite

Signal name	R-format	lw	sw	beq
RegWrite	1	1	0	0



Control Signal - MemtoReg

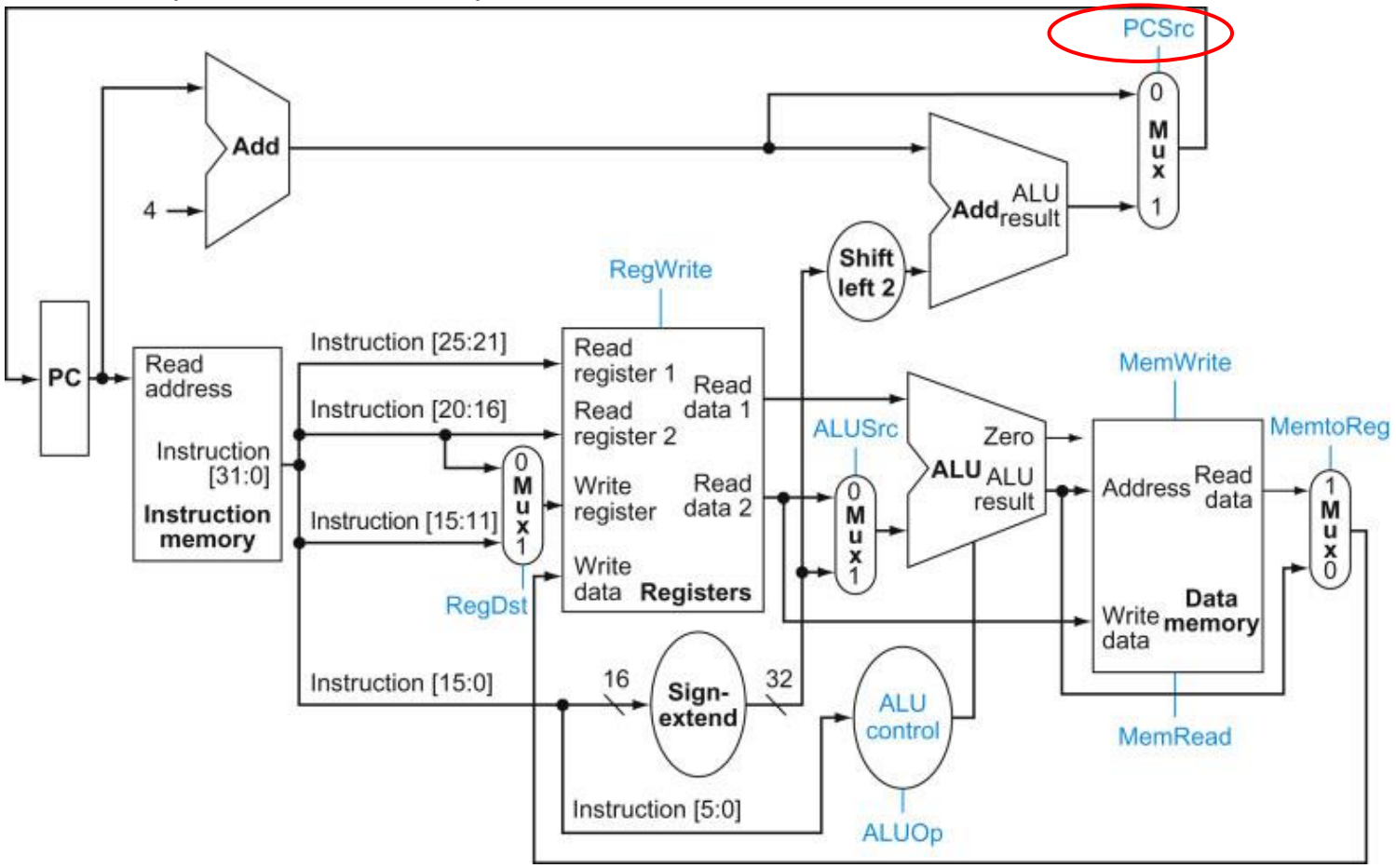
Signal name	R-format	lw	sw	beq
MemtoReg	0	1	X	X



Control Signal - PCSrc

- Introduce "Branch" signal for BEQ
  - PCSrc = (Branch AND Zero)

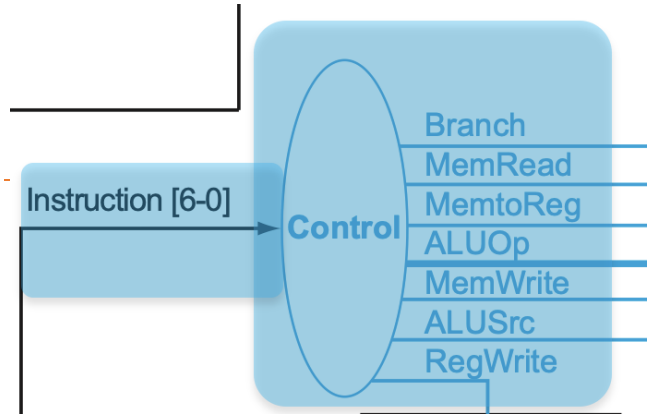
Signal name	R-format	lw	sw	beq
Branch	0	0	0	1



Summary of All Control Signals

Signal name	R-format	lw	sw	beq
ALUSrc	0	1	1	0
MemtoReg	0	1	X	X
RegWrite	1	1	0	0
MemRead	0	1	0	0
MemWrite	0	0	1	0
Branch	0	0	0	1
ALUOp1	1	0	0	0
ALUOp0	0	0	0	1

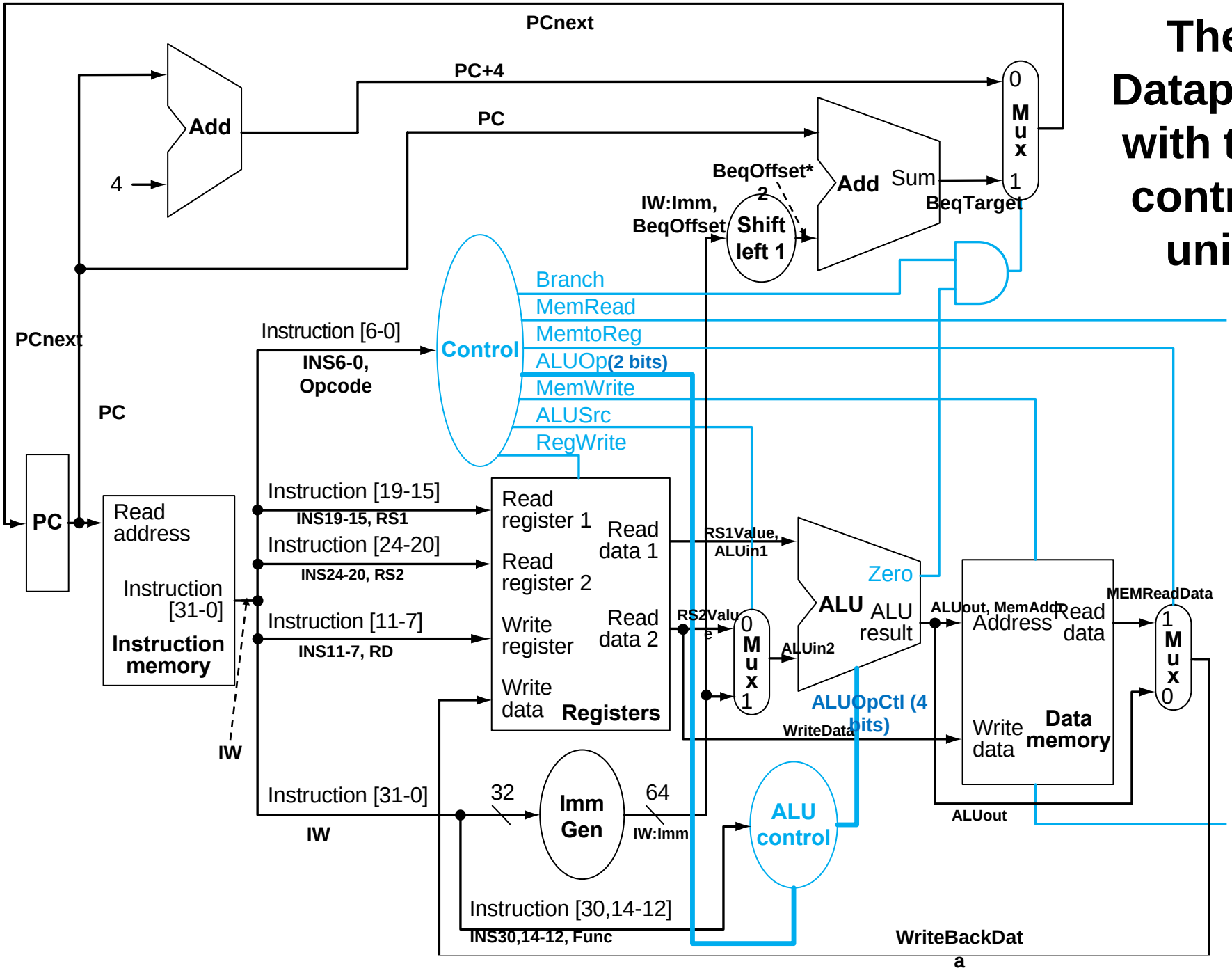
Truth Table for the Control Logic



- The "Control" logic derives the 2-bit ALUOp and other six 1-bit controls, solely based on Instruction[6-0] bits, which is the opcode of an instruction word.

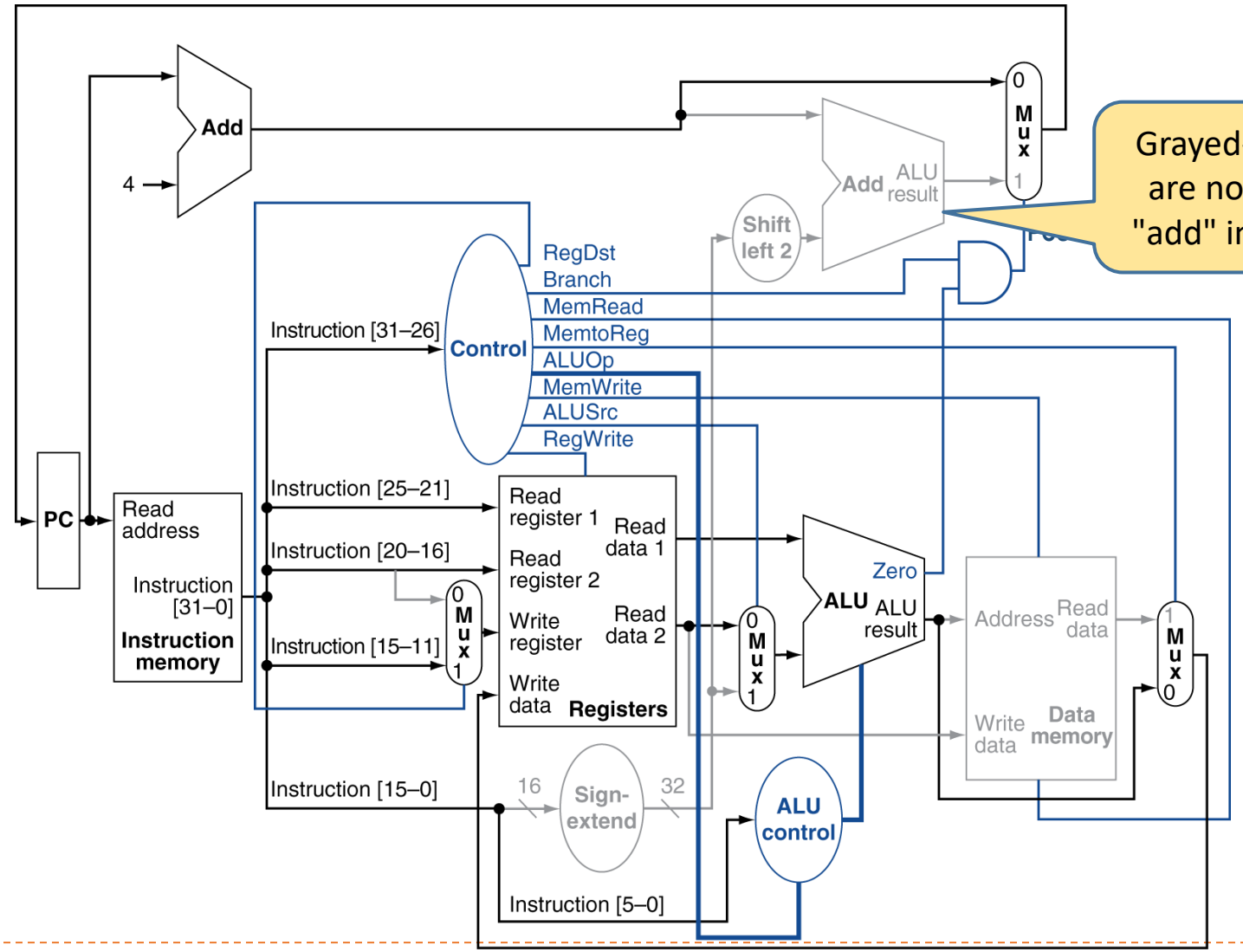
Input or output	Signal name	R-format	ld	sd	beq
Inputs	I[6]	0	0	0	1
	I[5]	1	0	1	1
	I[4]	1	0	0	0
	I[3]	0	0	0	0
	I[2]	0	0	0	0
	I[1]	1	1	1	1
	I[0]	1	1	1	1
Outputs	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

The Datapath with the control unit

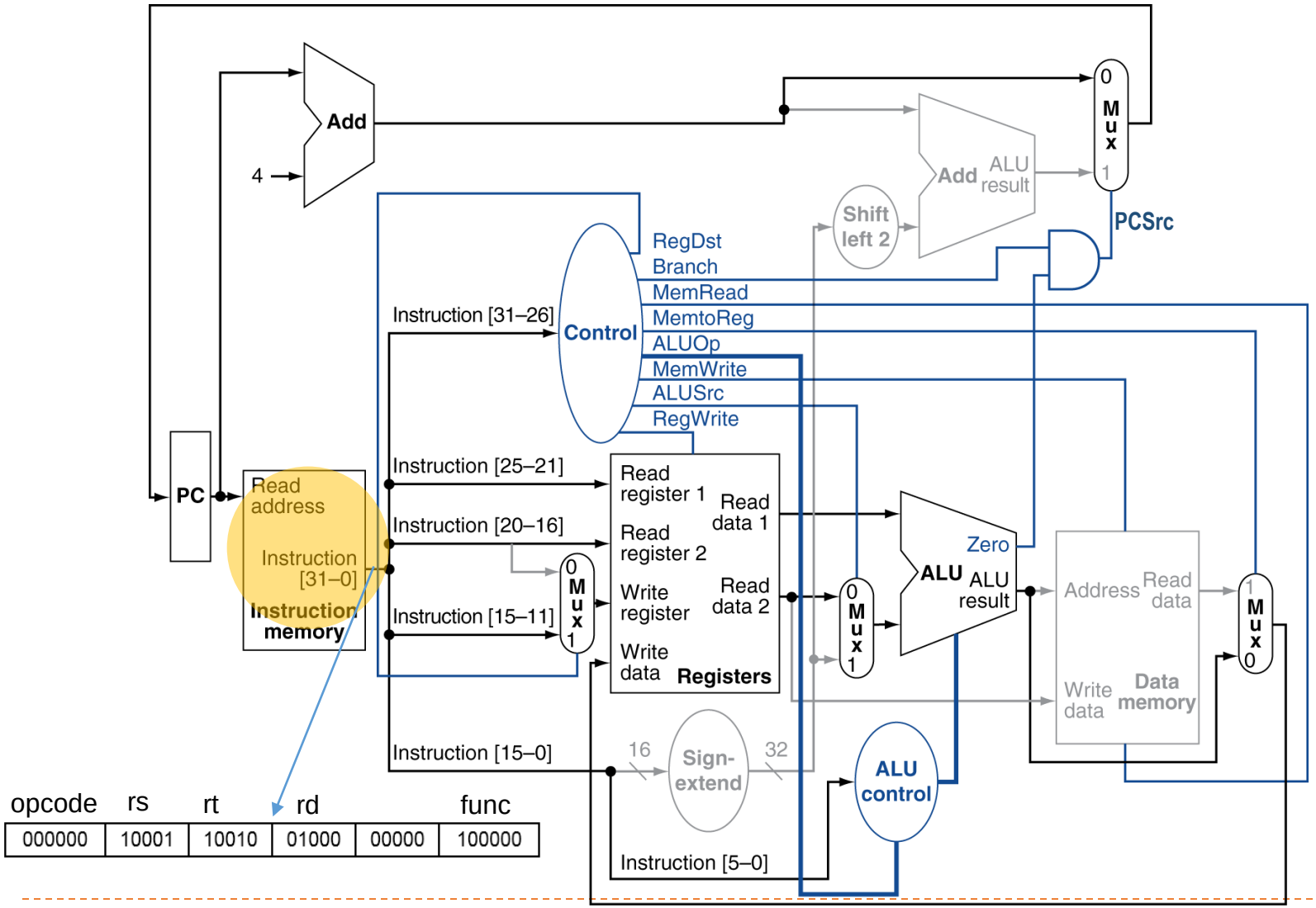


Example Walk-Through (add)

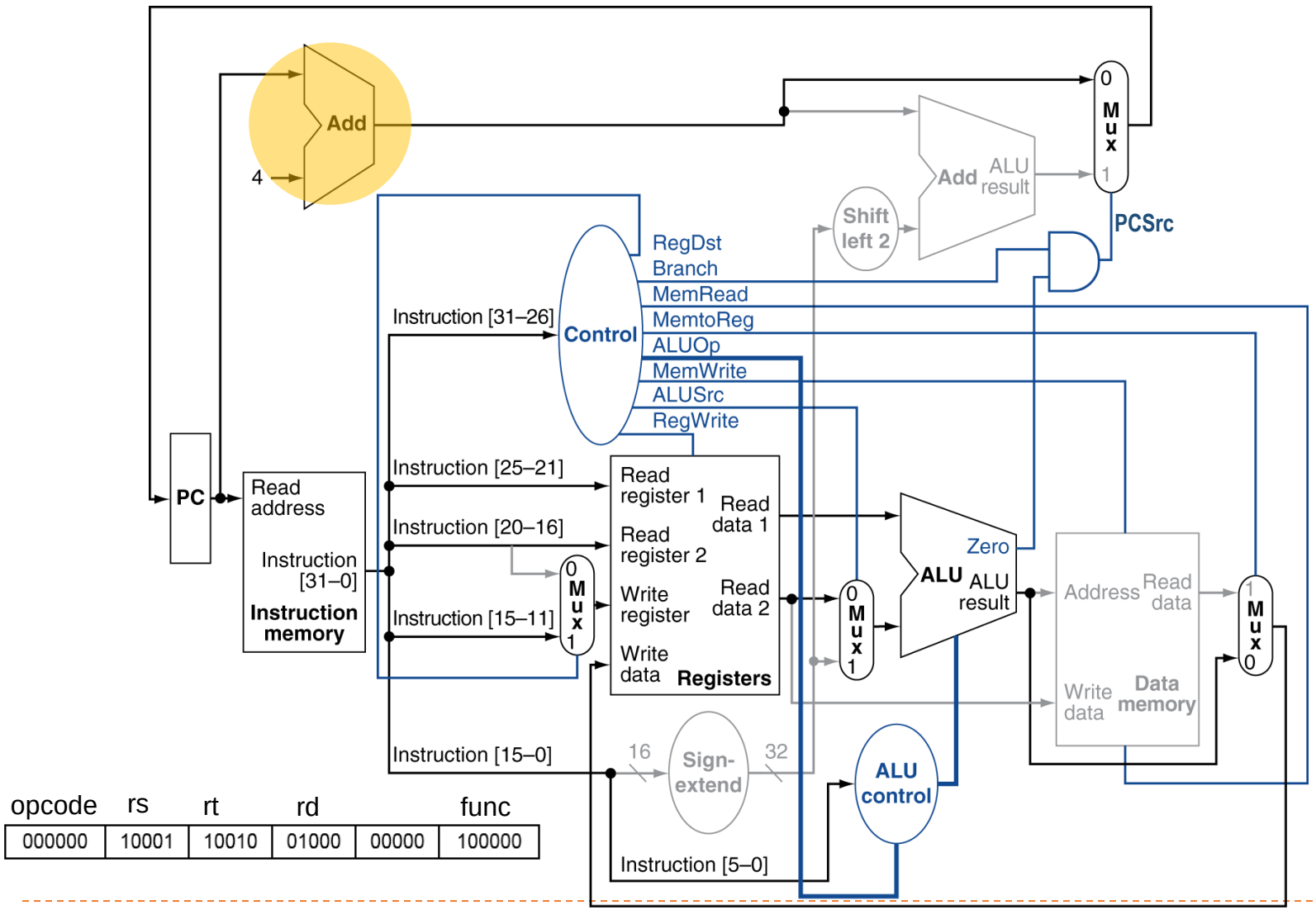
R-Type Instruction – add $t0, $s1, $s2



R-Type Instruction – add $t0, $s1, $s2

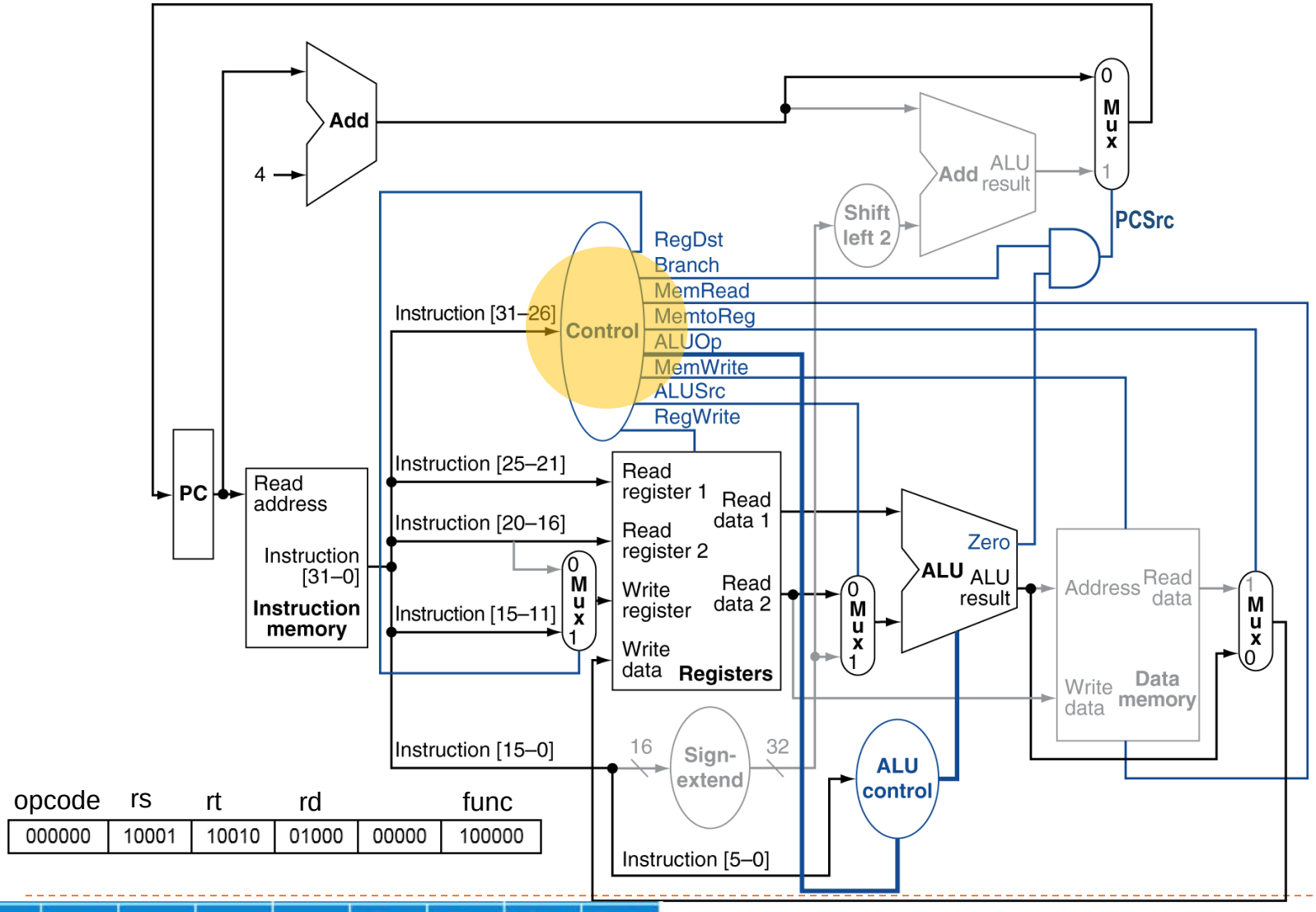


R-Type Instruction – add $t0, $s1, $s2



opcode	rs	rt	rd		func
000000	10001	10010	01000	00000	100000

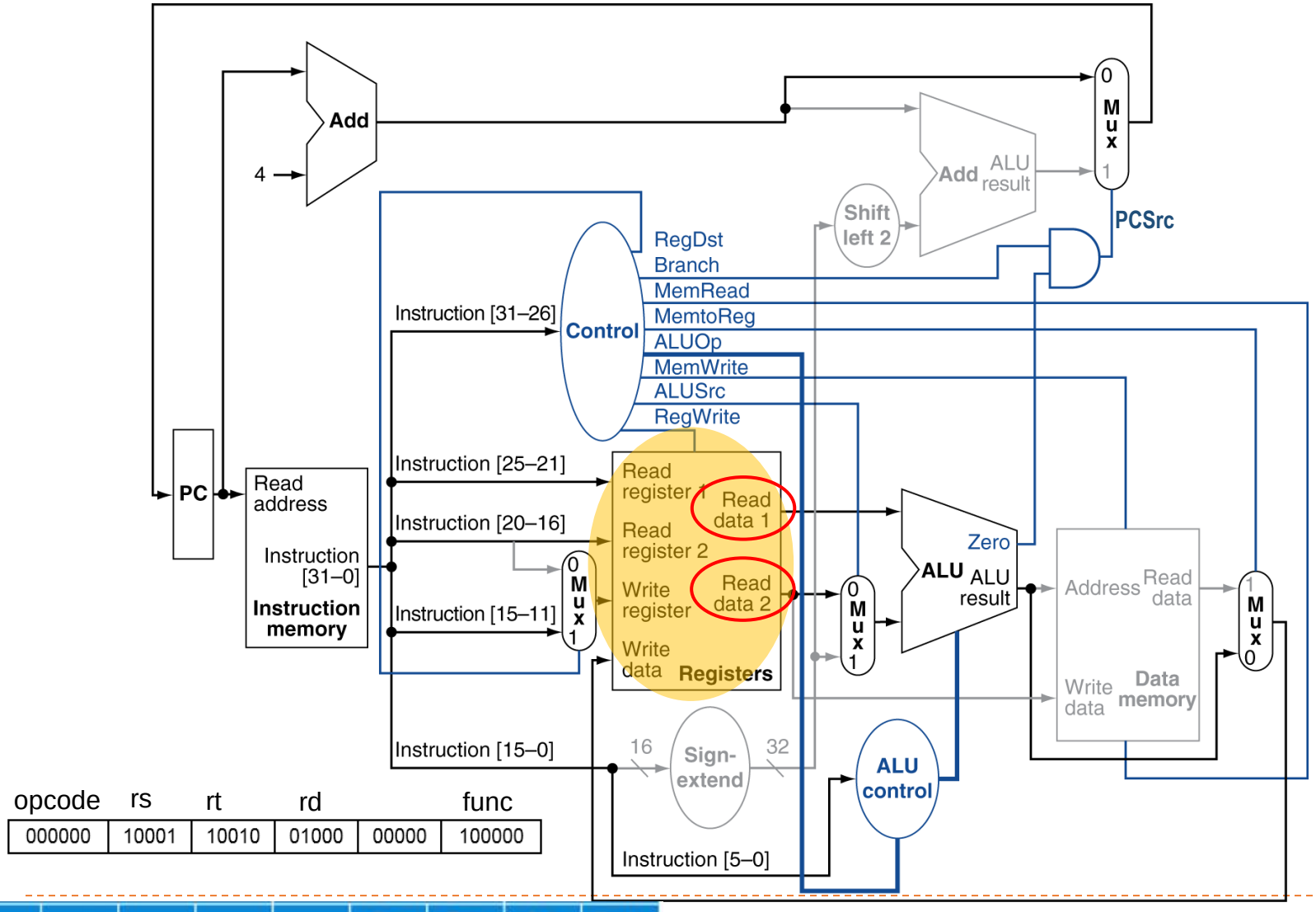
R-Type Instruction – add $t0, $s1, $s2



opcode	rs	rt	rd		func
000000	10001	10010	01000	00000	100000

RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
1	1	0	0	0	0	0	1	0

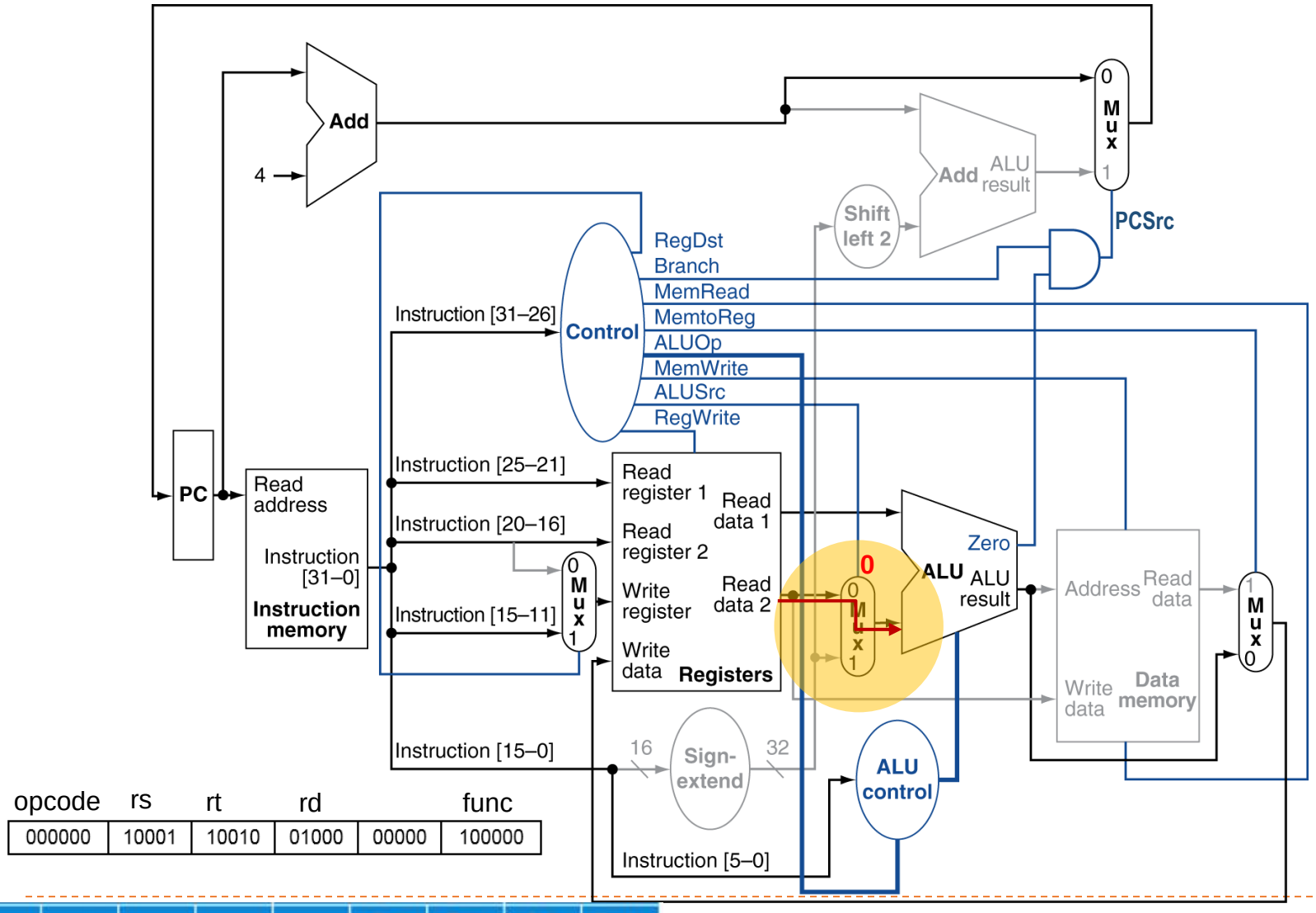
R-Type Instruction – add $t0, $s1, $s2



opcode	rs	rt	rd		func
000000	10001	10010	01000	00000	100000

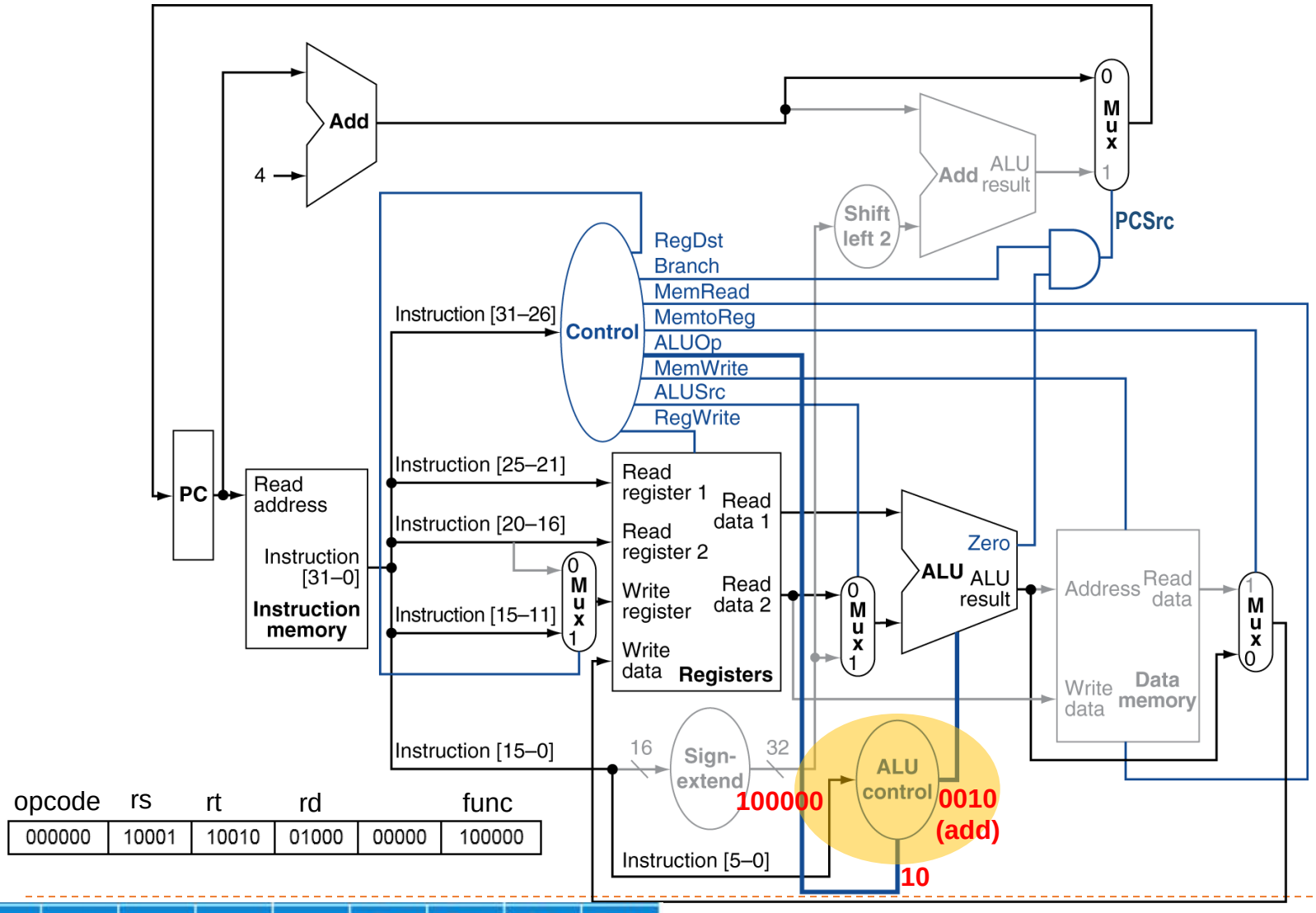
RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
1	1	0	0	0	0	0	1	0

R-Type Instruction – add $t0, $s1, $s2



RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
1	1	0	0	0	0	0	1	0

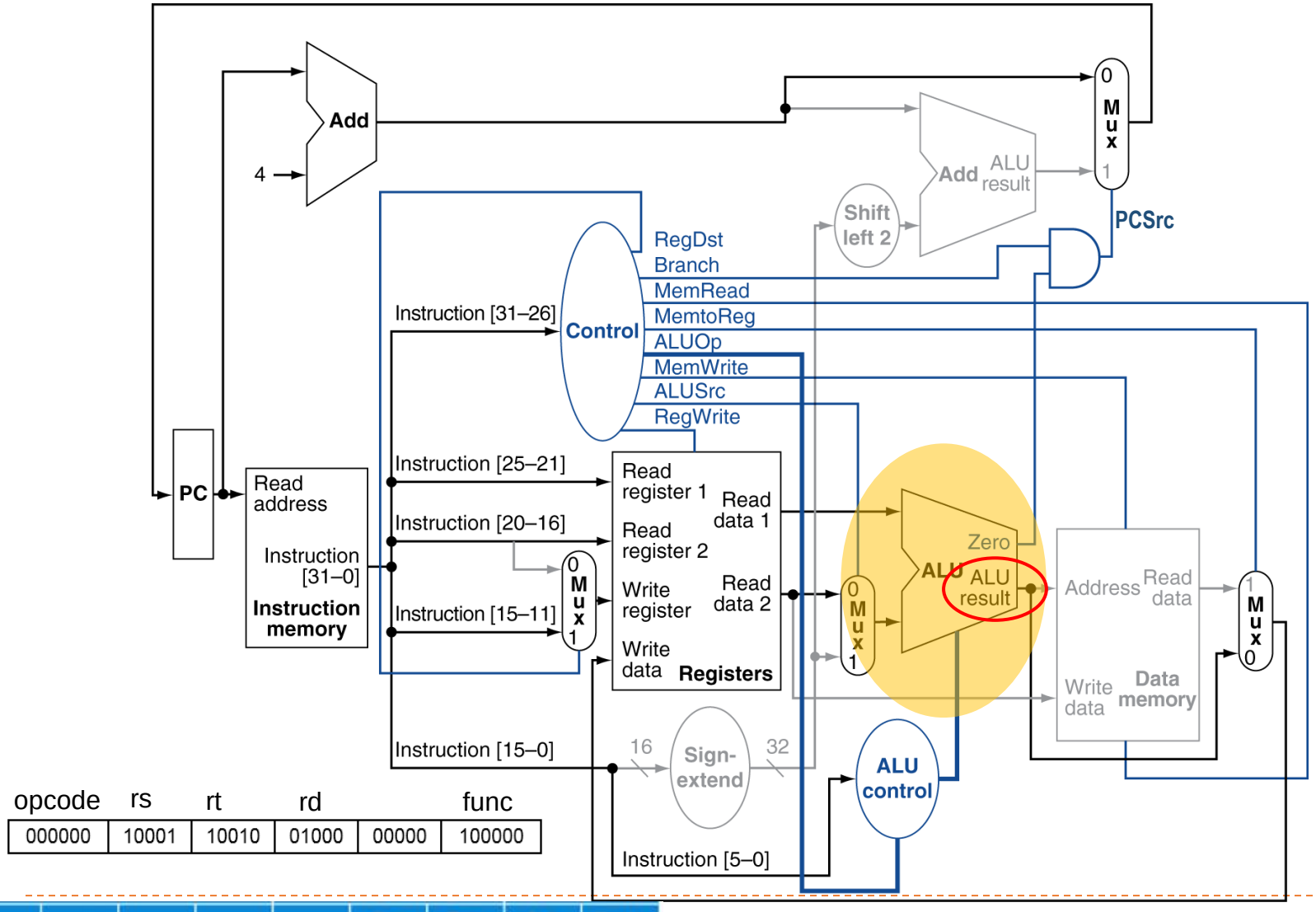
R-Type Instruction – add $t0, $s1, $s2



opcode	rs	rt	rd		func
000000	10001	10010	01000	00000	100000

RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
1	1	0	0	0	0	0	1	0

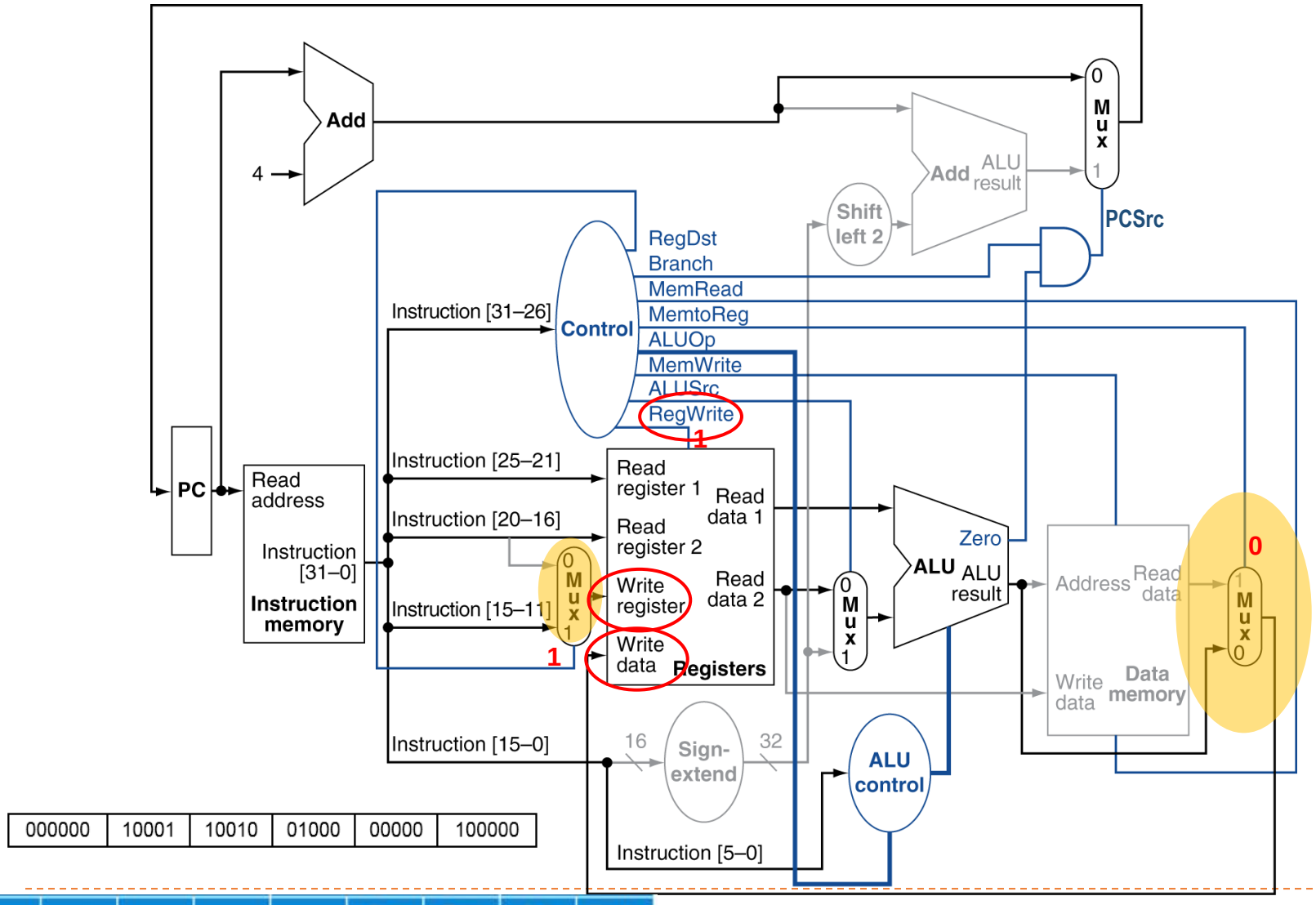
R-Type Instruction – add $t0, $s1, $s2



opcode	rs	rt	rd		func
000000	10001	10010	01000	00000	100000

RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
1	1	0	0	0	0	0	1	0

R-Type Instruction – add $t0, $s1, $s2

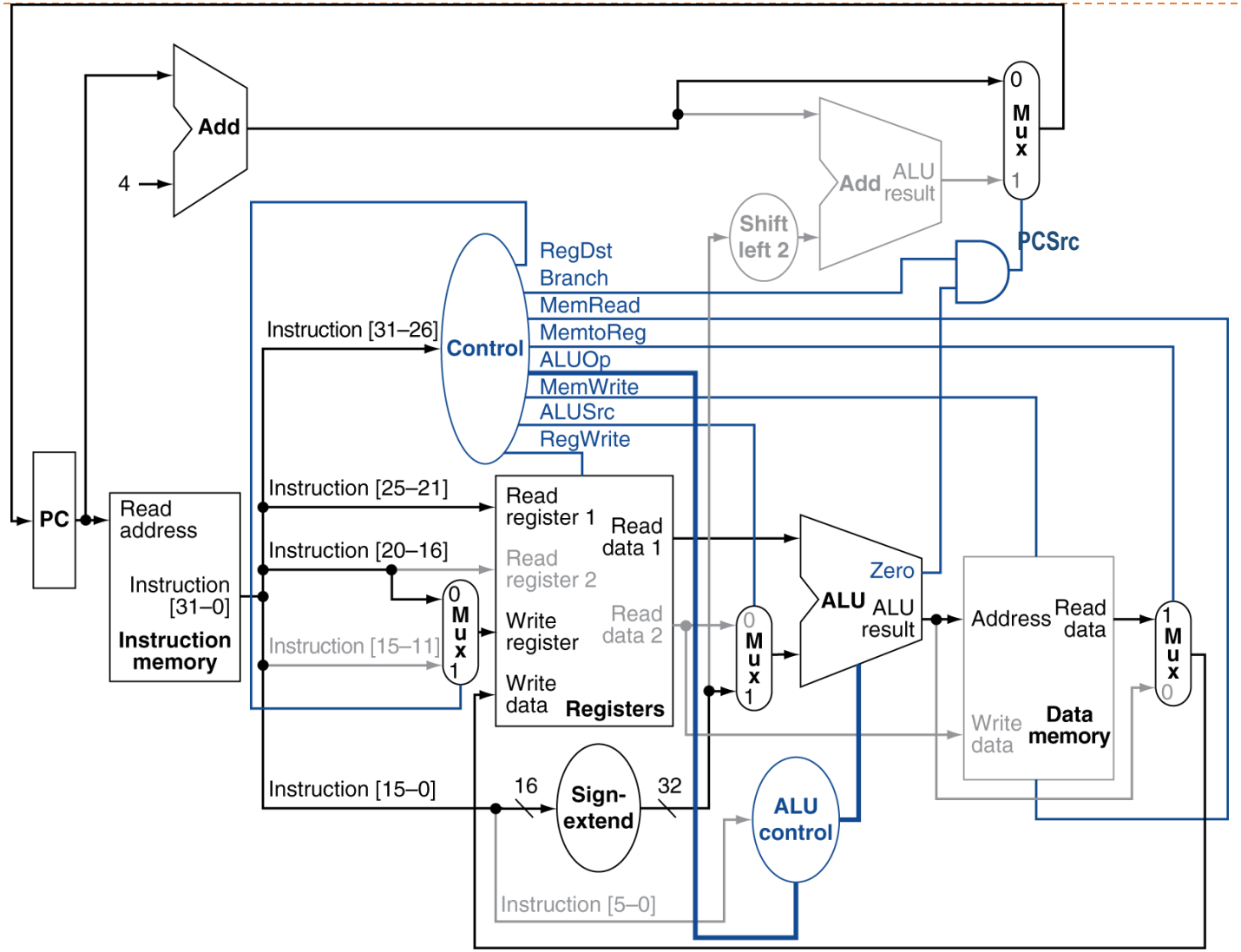


000000	10001	10010	01000	00000	100000

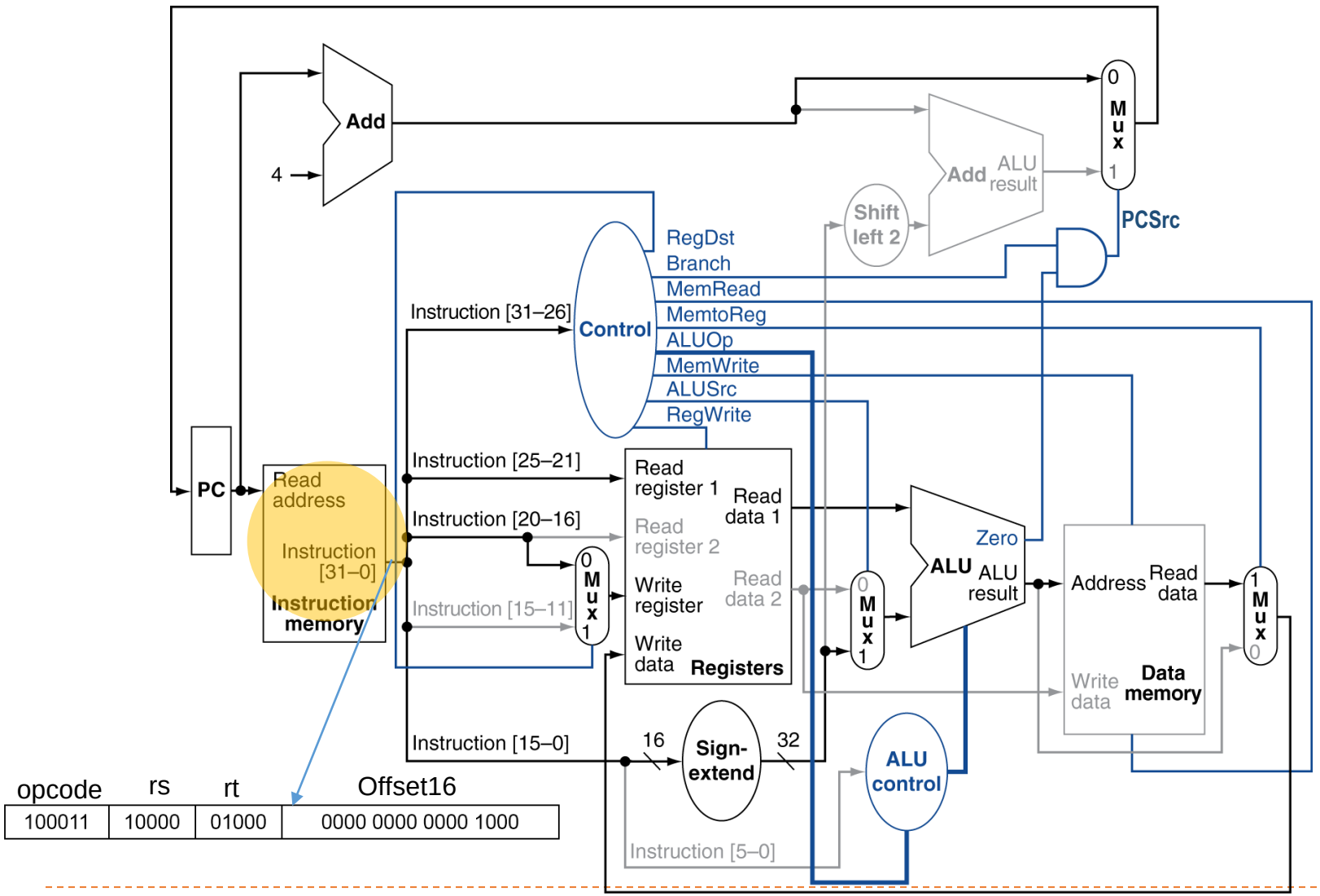
RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
1	1	0	0	0	0	0	1	0

Example Walk-Through (load)

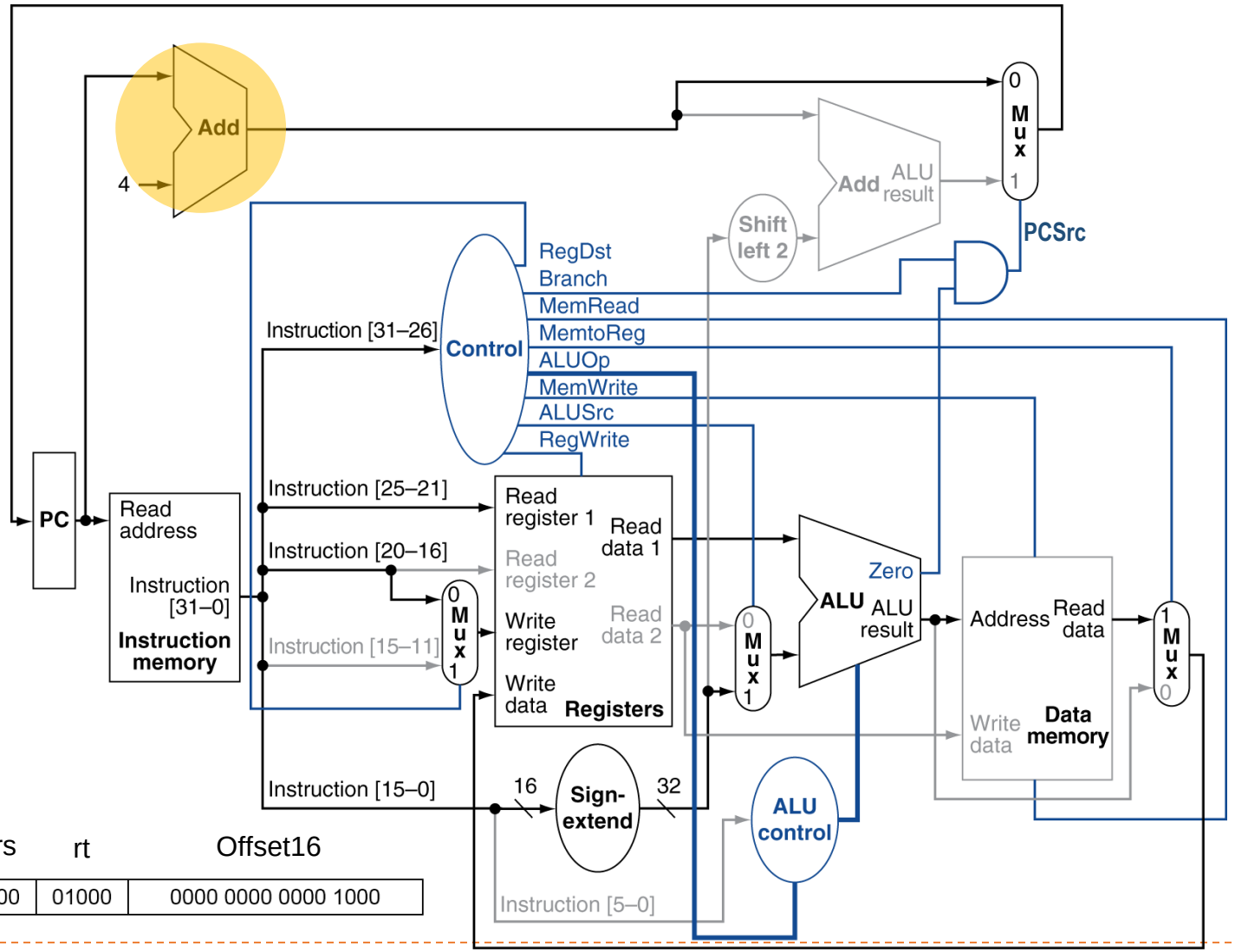
Load Instruction – lw $t0, 8($s0)



Load Instruction – lw $t0, 8($s0)

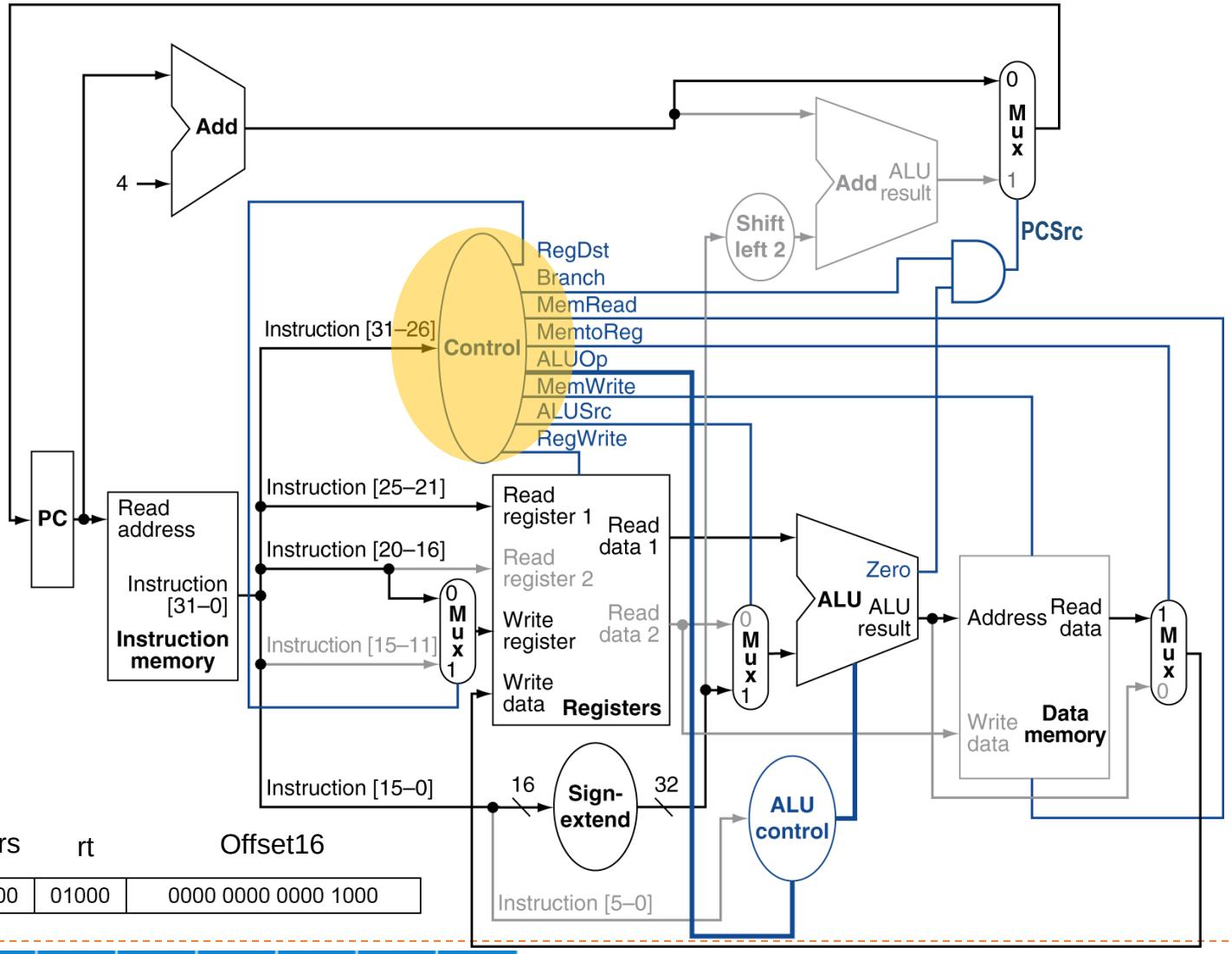


Load Instruction – lw $t0, 8($s0)



opcode	rs	rt	Offset16
100011	10000	01000	0000 0000 0000 1000

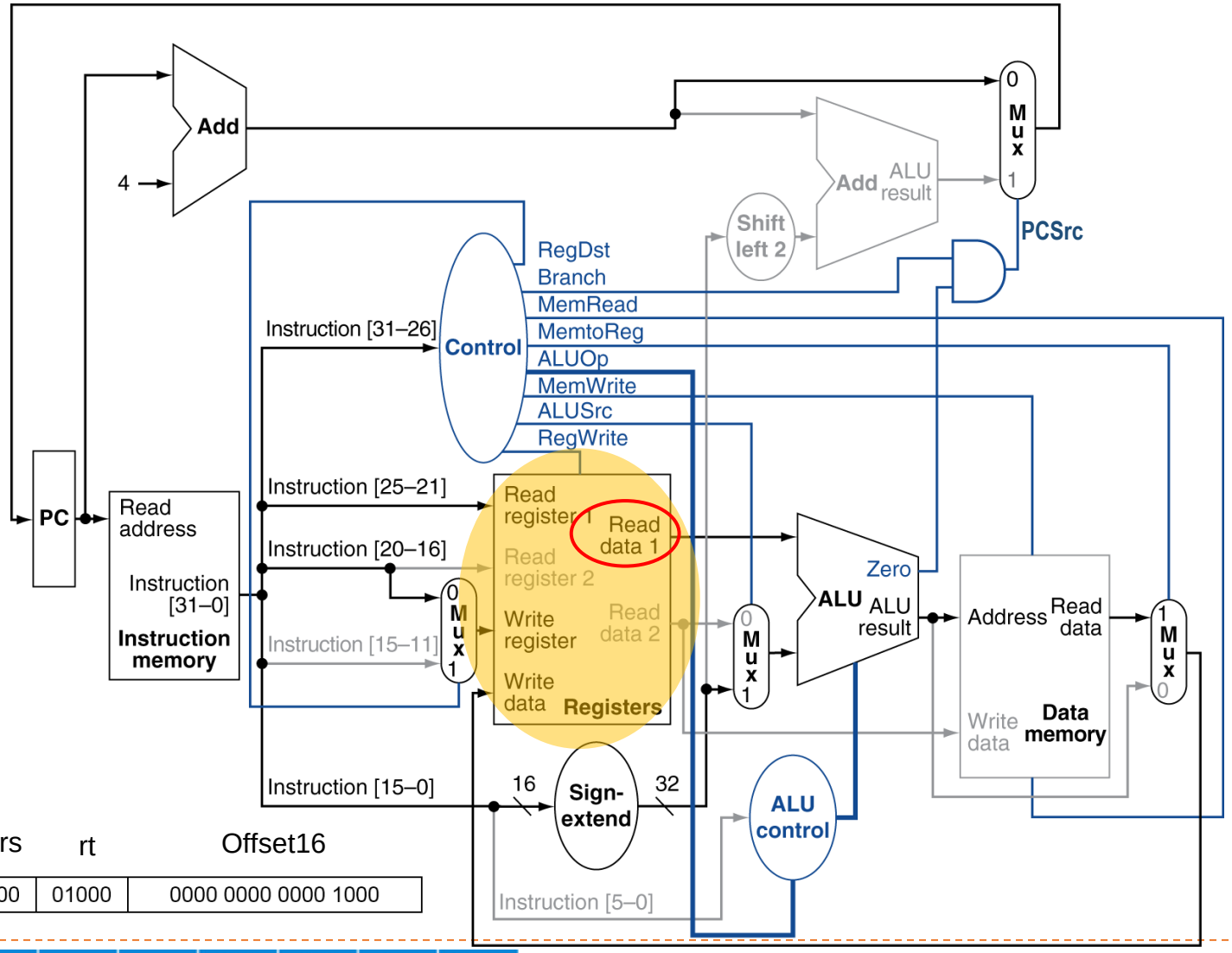
Load Instruction – lw $t0, 8($s0)



opcode	rs	rt	Offset16
100011	10000	01000	0000 0000 0000 1000

RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
0	0	0	1	0	1	0	1	1

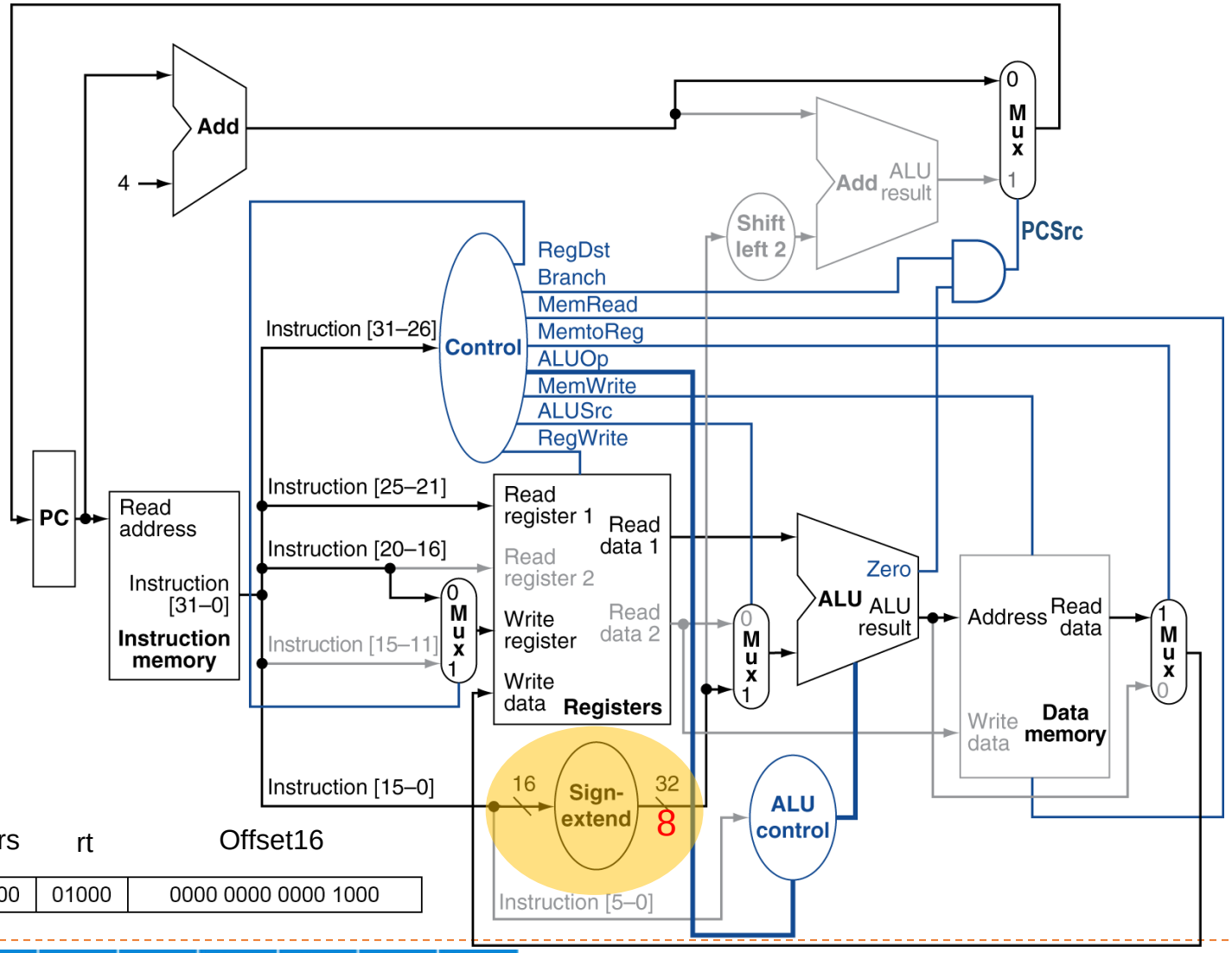
Load Instruction – lw $t0, 8($s0)



opcode	rs	rt	Offset16
100011	10000	01000	0000 0000 0000 1000

RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
0	0	0	1	0	1	0	1	1

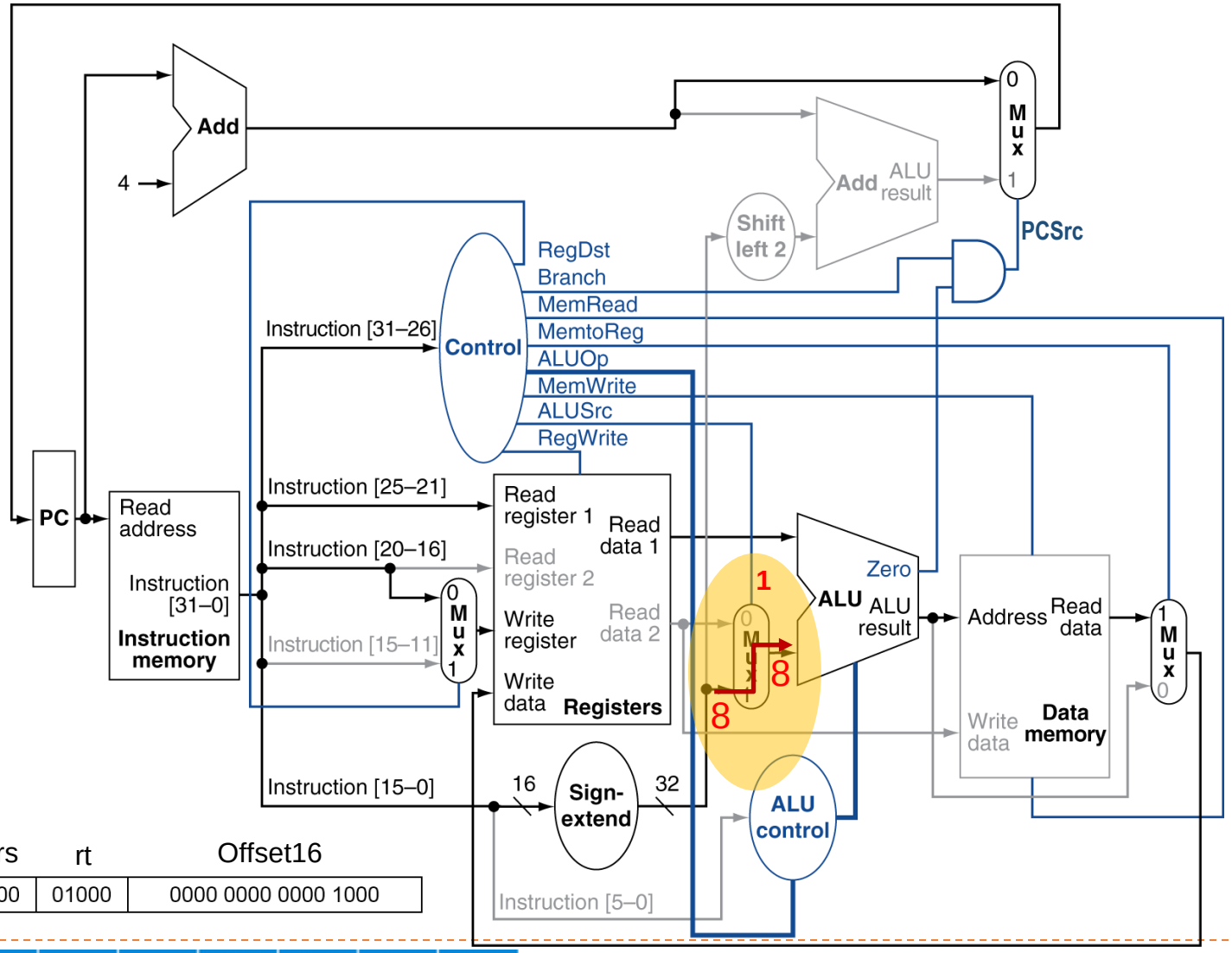
Load Instruction – lw $t0, 8($s0)



opcode	rs	rt	Offset16
100011	10000	01000	0000 0000 0000 1000

RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
0	0	0	1	0	1	0	1	1

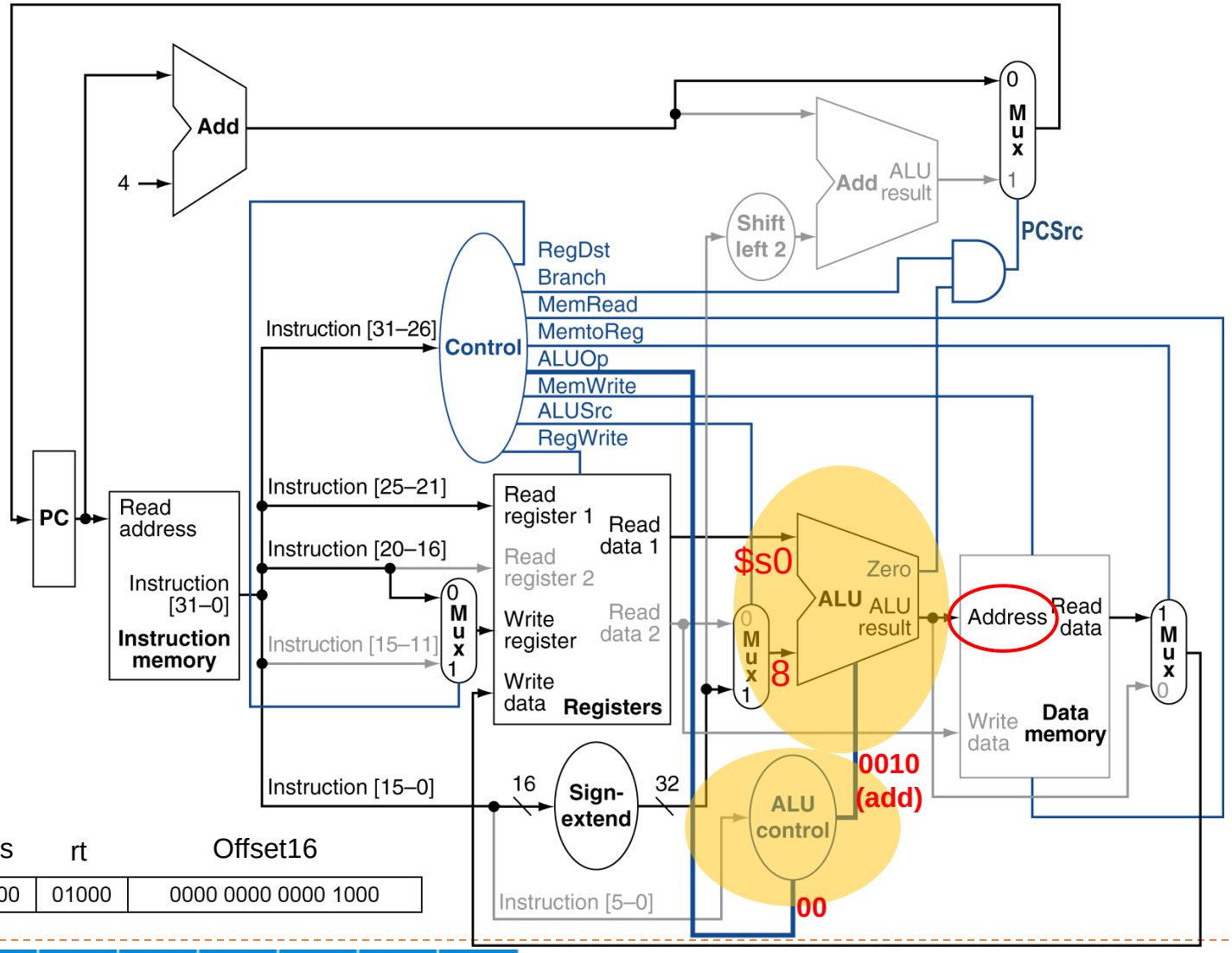
Load Instruction – lw $t0, 8($s0)



opcode	rs	rt	Offset16
100011	10000	01000	0000 0000 0000 1000

RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
0	0	0	1	0	1	0	1	1

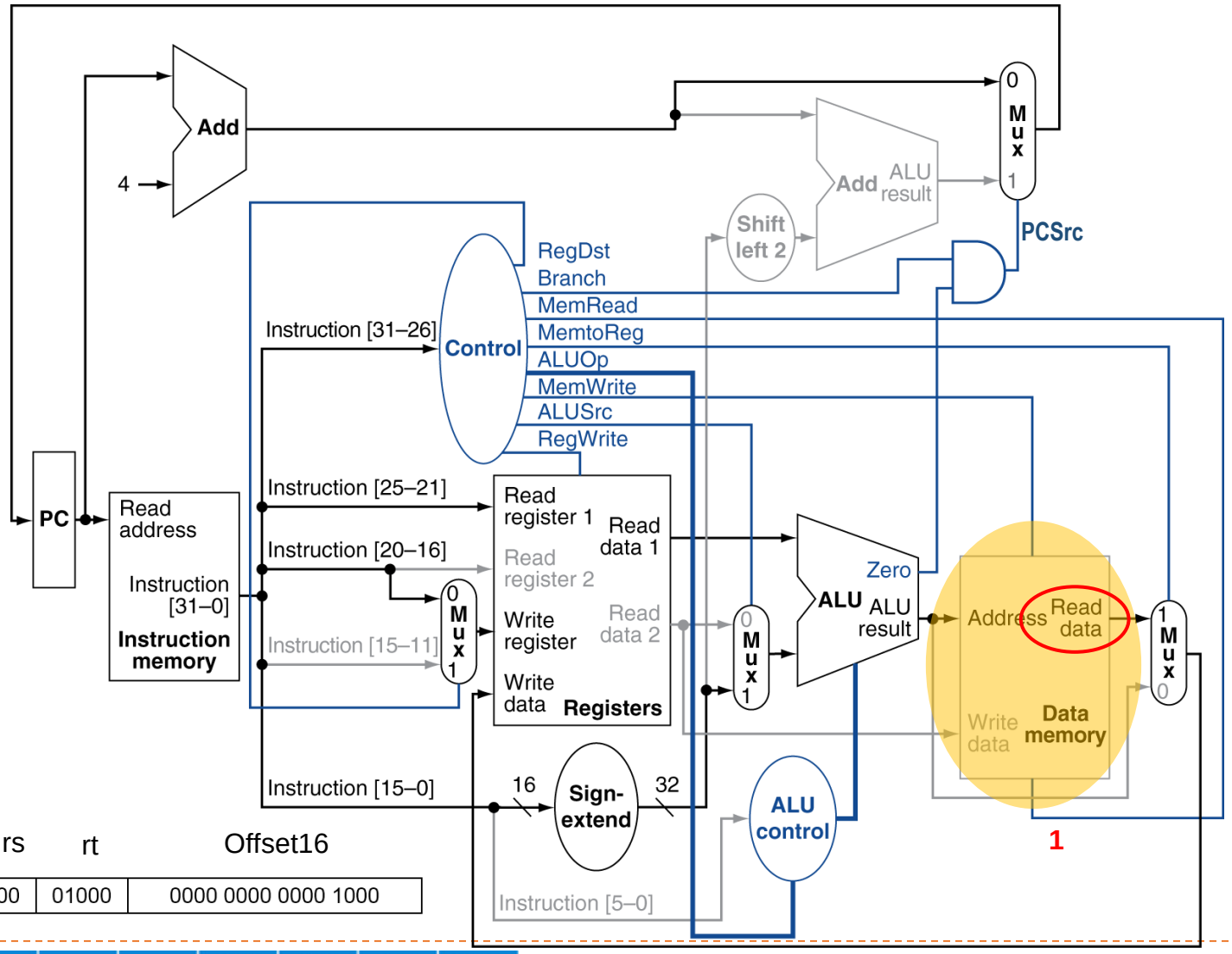
Load Instruction – lw $t0, 8($s0)



opcode	rs	rt	Offset16
100011	10000	01000	0000 0000 0000 1000

RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
0	0	0	1	0	1	0	1	1

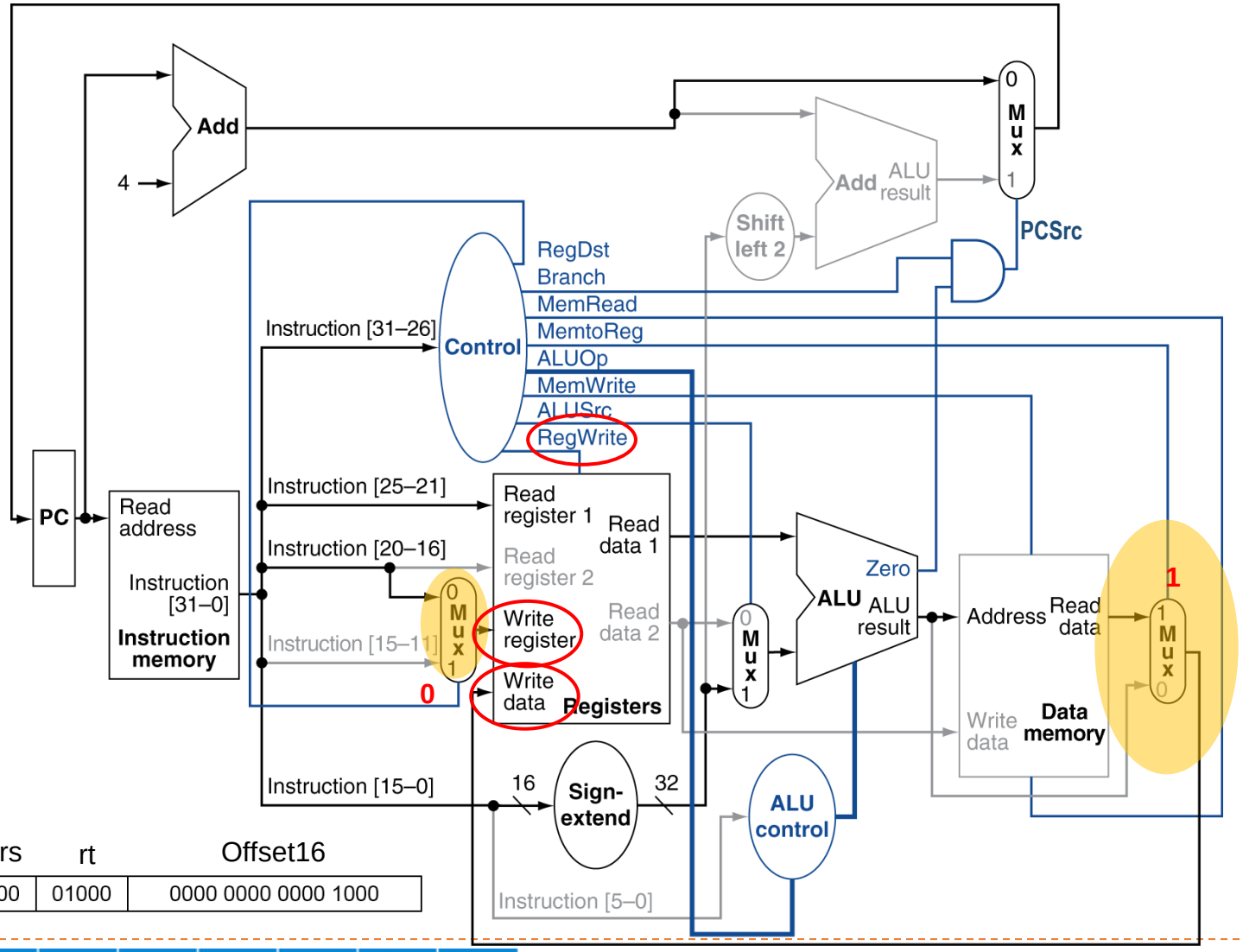
Load Instruction – lw $t0, 8($s0)



opcode	rs	rt	Offset16
100011	10000	01000	0000 0000 0000 1000

RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
0	0	0	1	0	1	0	1	1

Load Instruction – lw $t0, 8($s0)

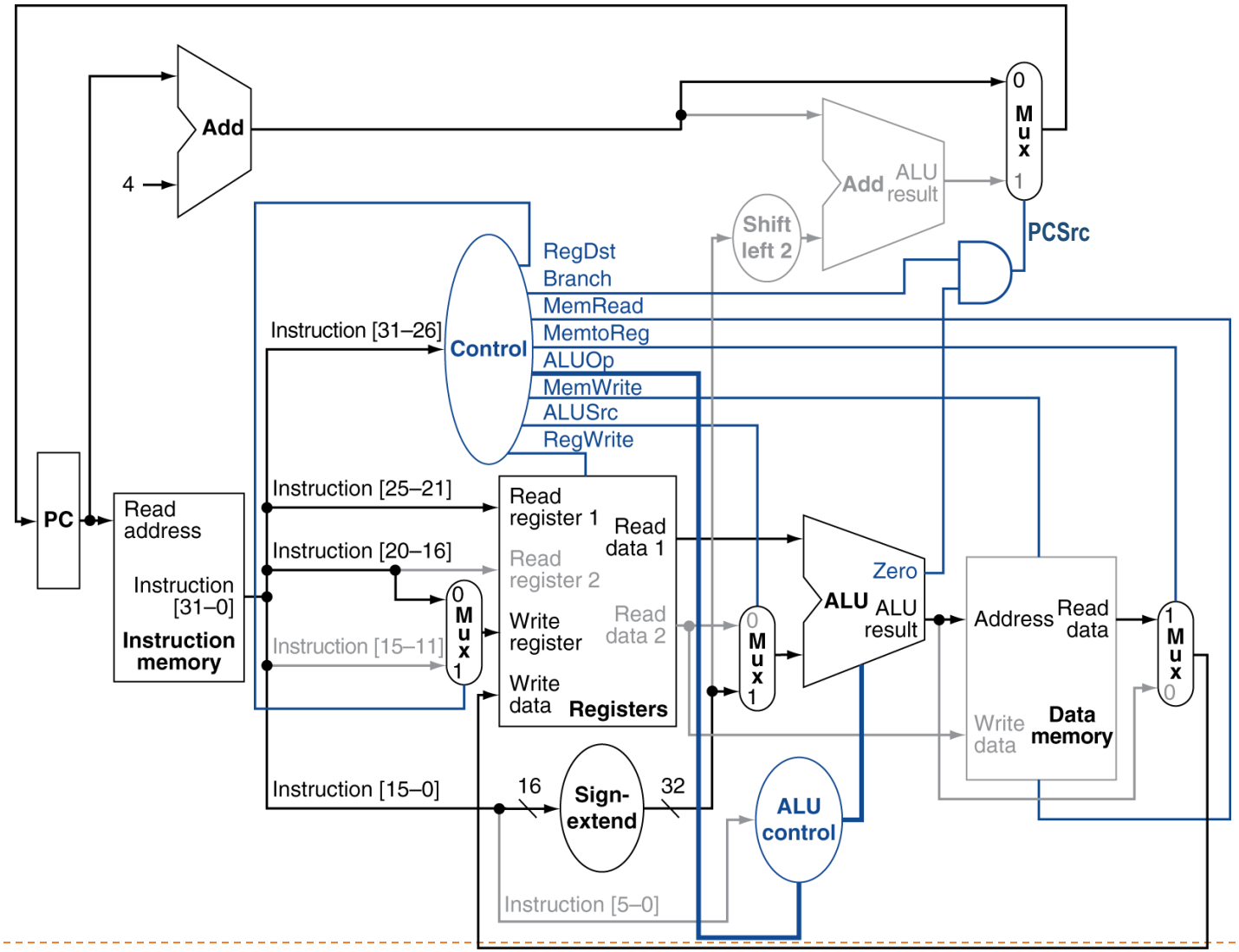


opcode	rs	rt	Offset16
100011	10000	01000	0000 0000 0000 1000

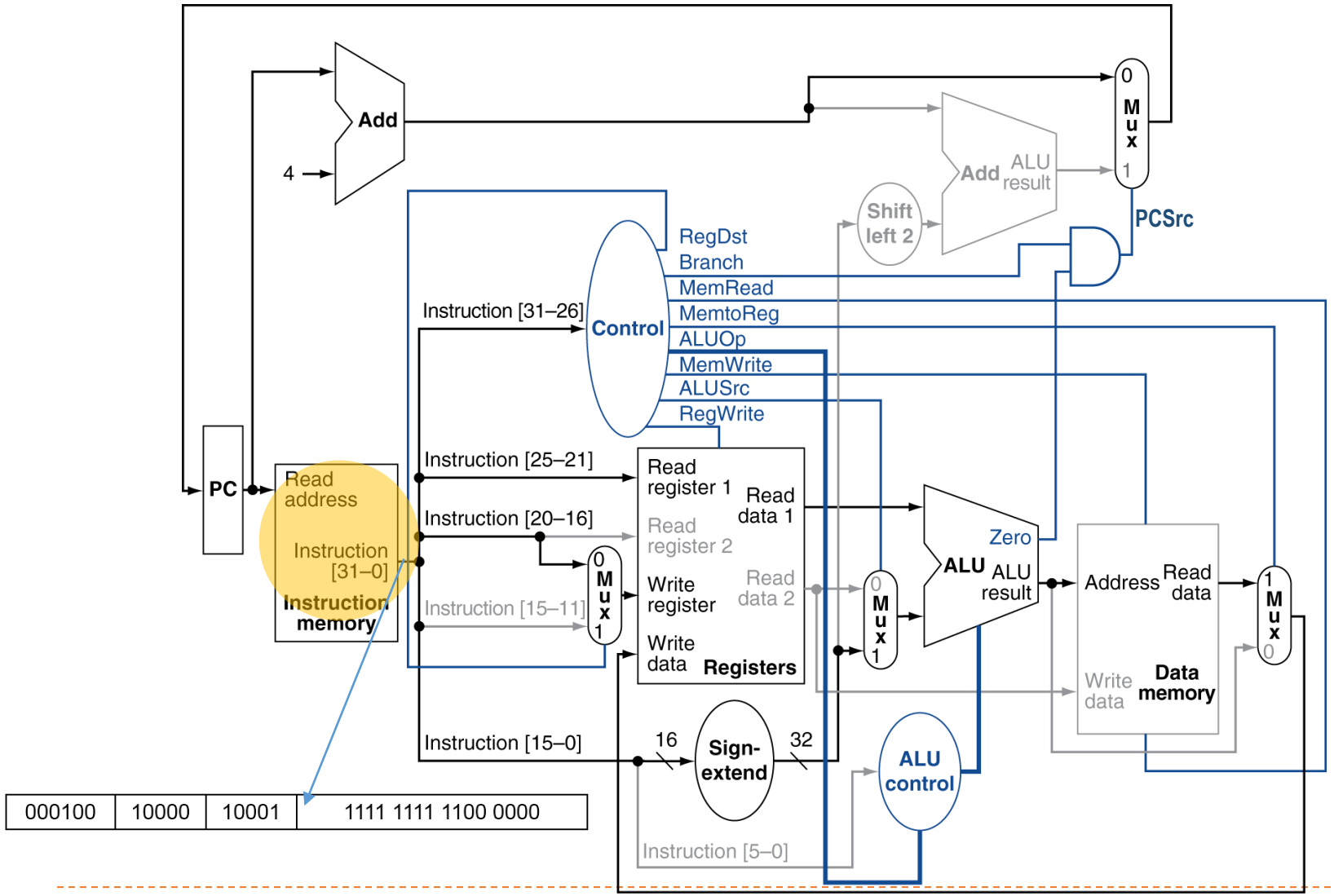
RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
0	0	0	1	0	1	0	1	1

Example Walk-Through (beq)

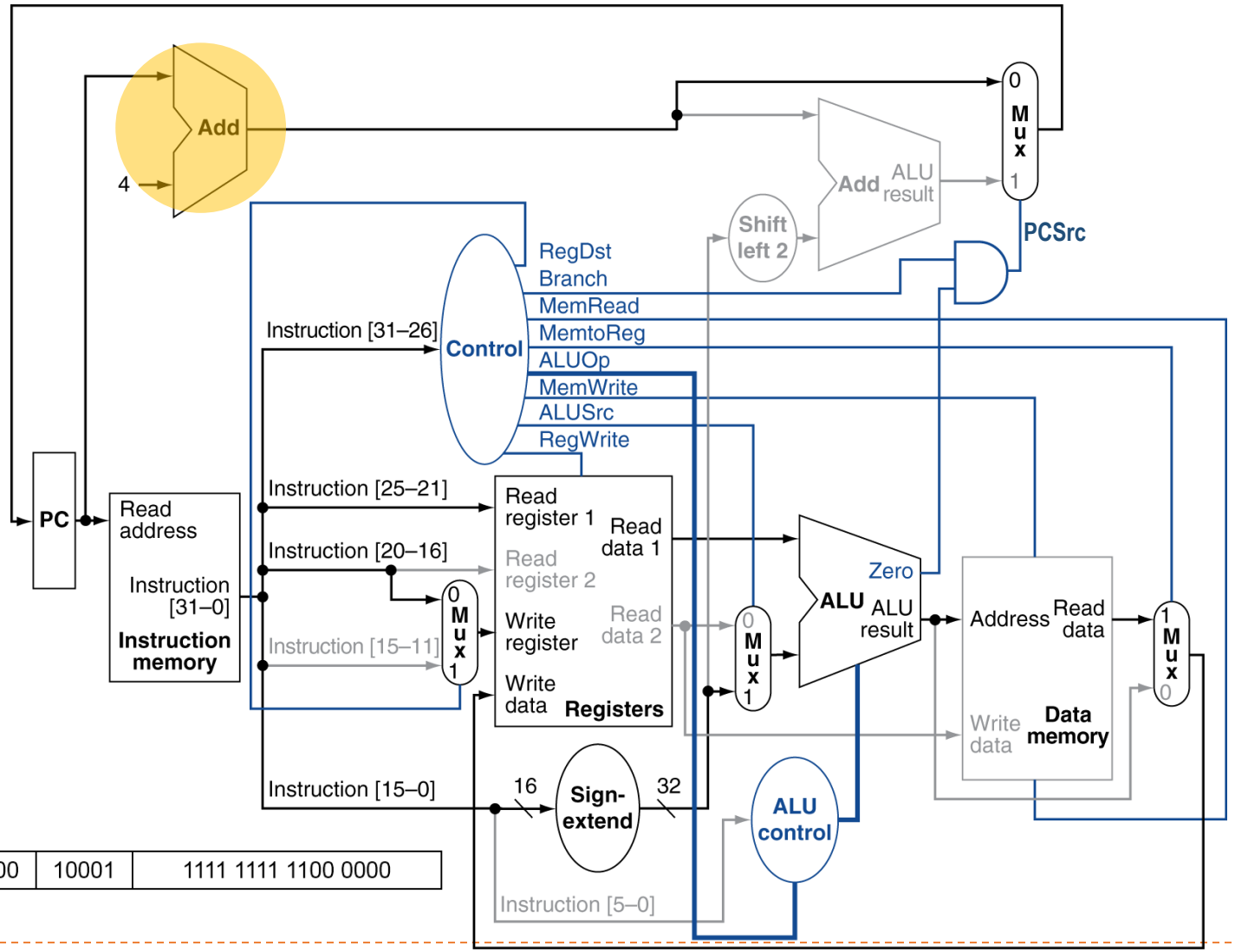
Branch-on-Equal Instruction – beq $s0, $s1, -64



Branch-on-Equal Instruction – beq $s0, $s1, -64

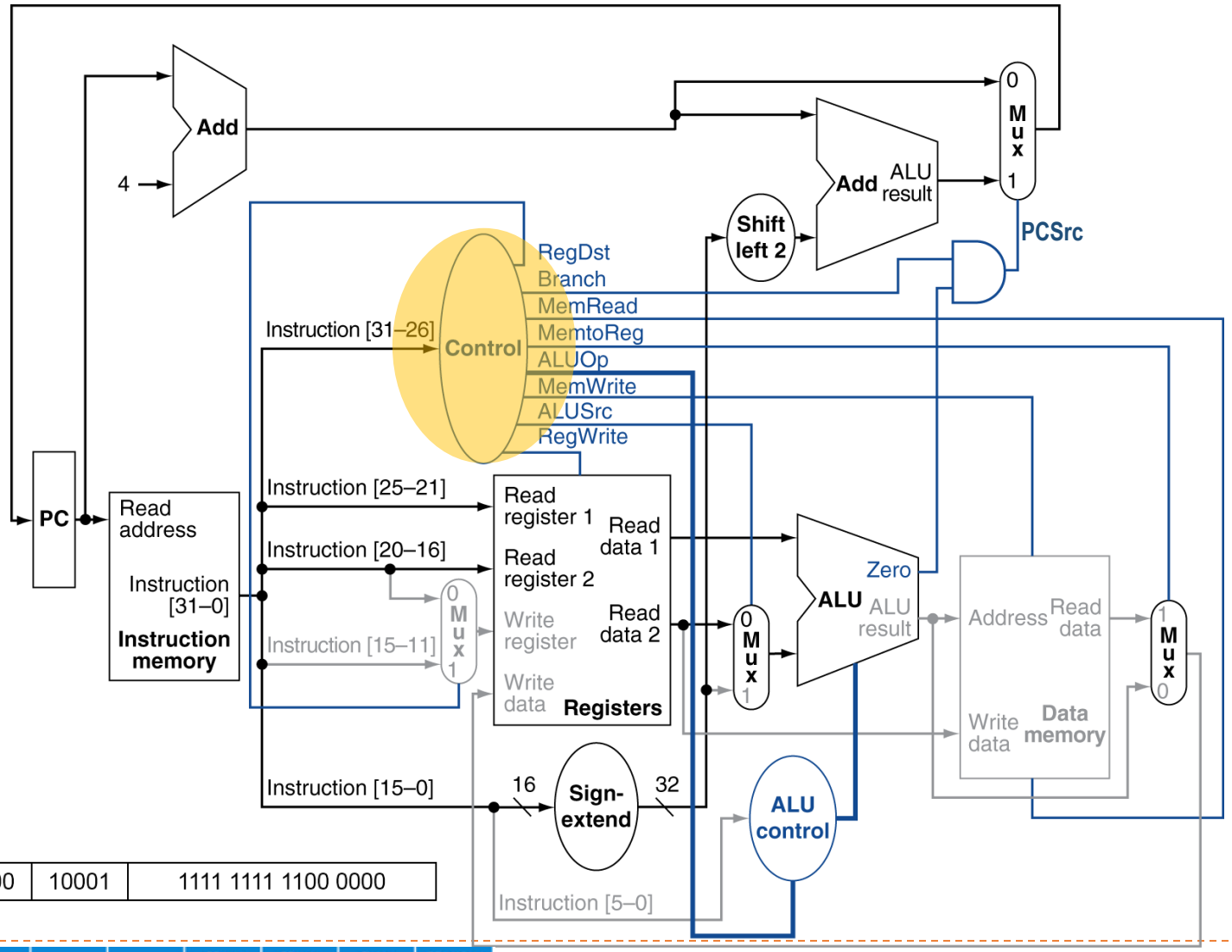


Branch-on-Equal Instruction – beq $s0, $s1, -64



000100	10000	10001	1111 1111 1100 0000

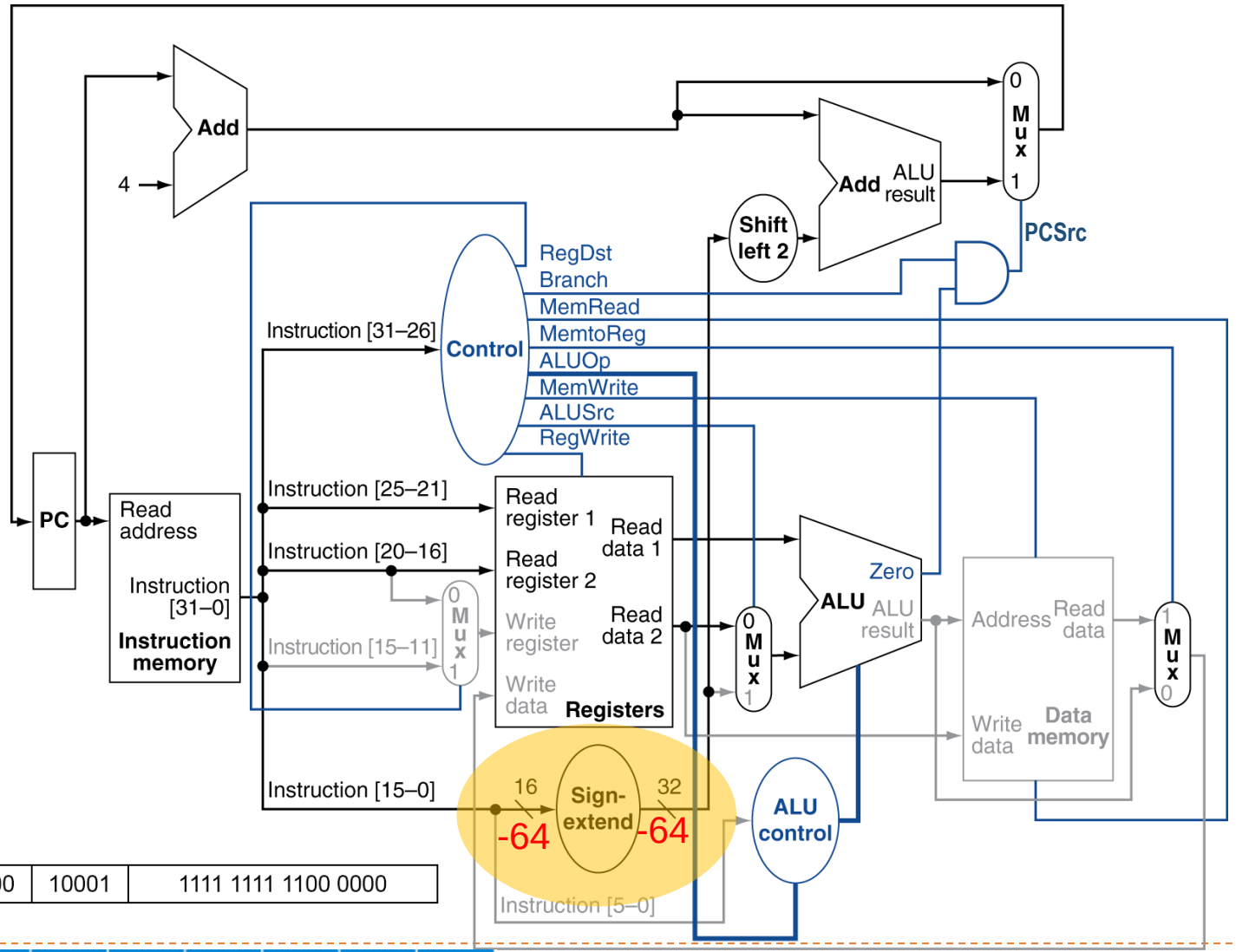
Branch-on-Equal Instruction – beq $s0, $s1, -64



000100	10000	10001	1111 1111 1100 0000

RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
X	0	1	0	1	0	0	0	X

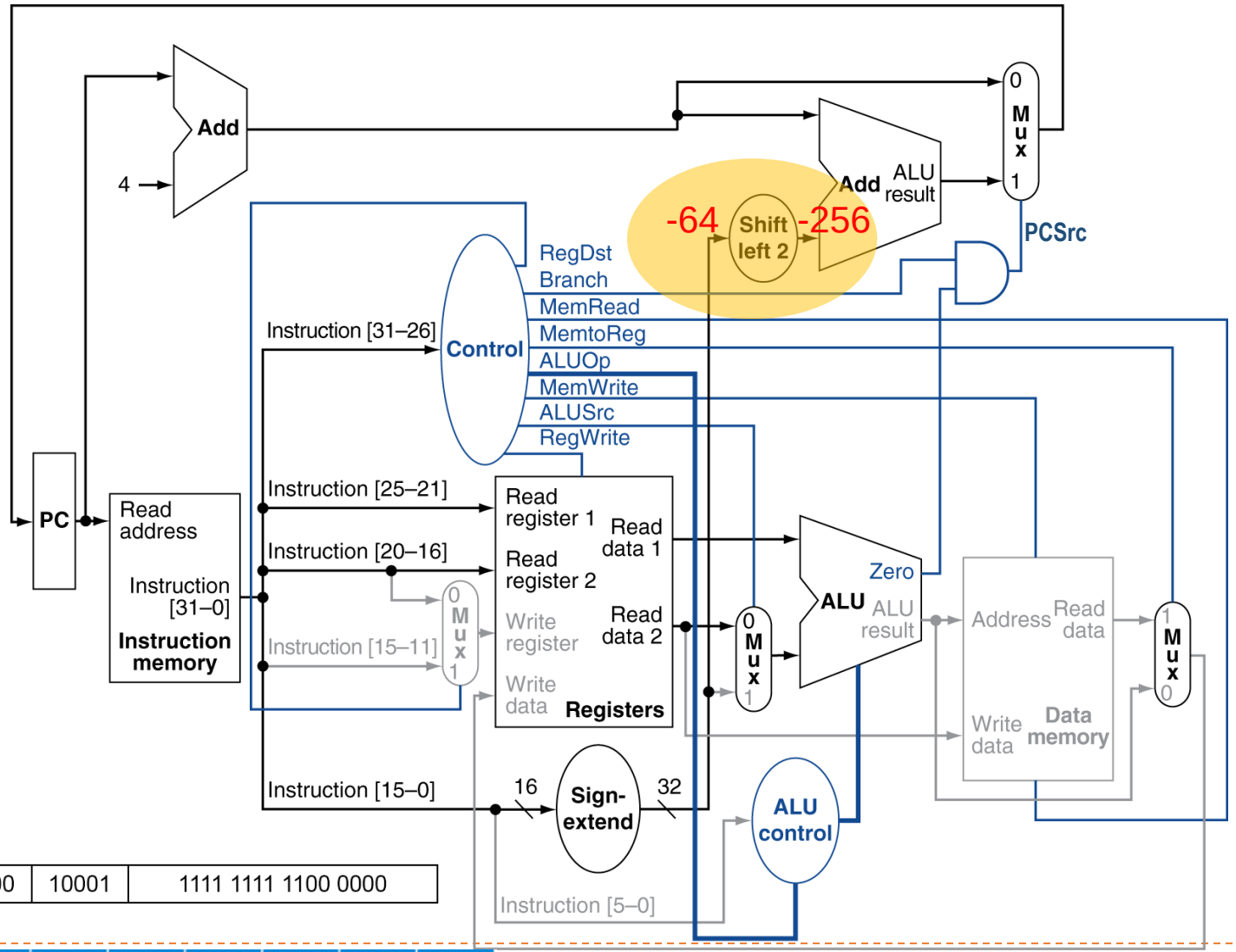
Branch-on-Equal Instruction – beq $s0, $s1, -64



000100	10000	10001	1111 1111 1100 0000

RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
X	0	1	0	1	0	0	0	X

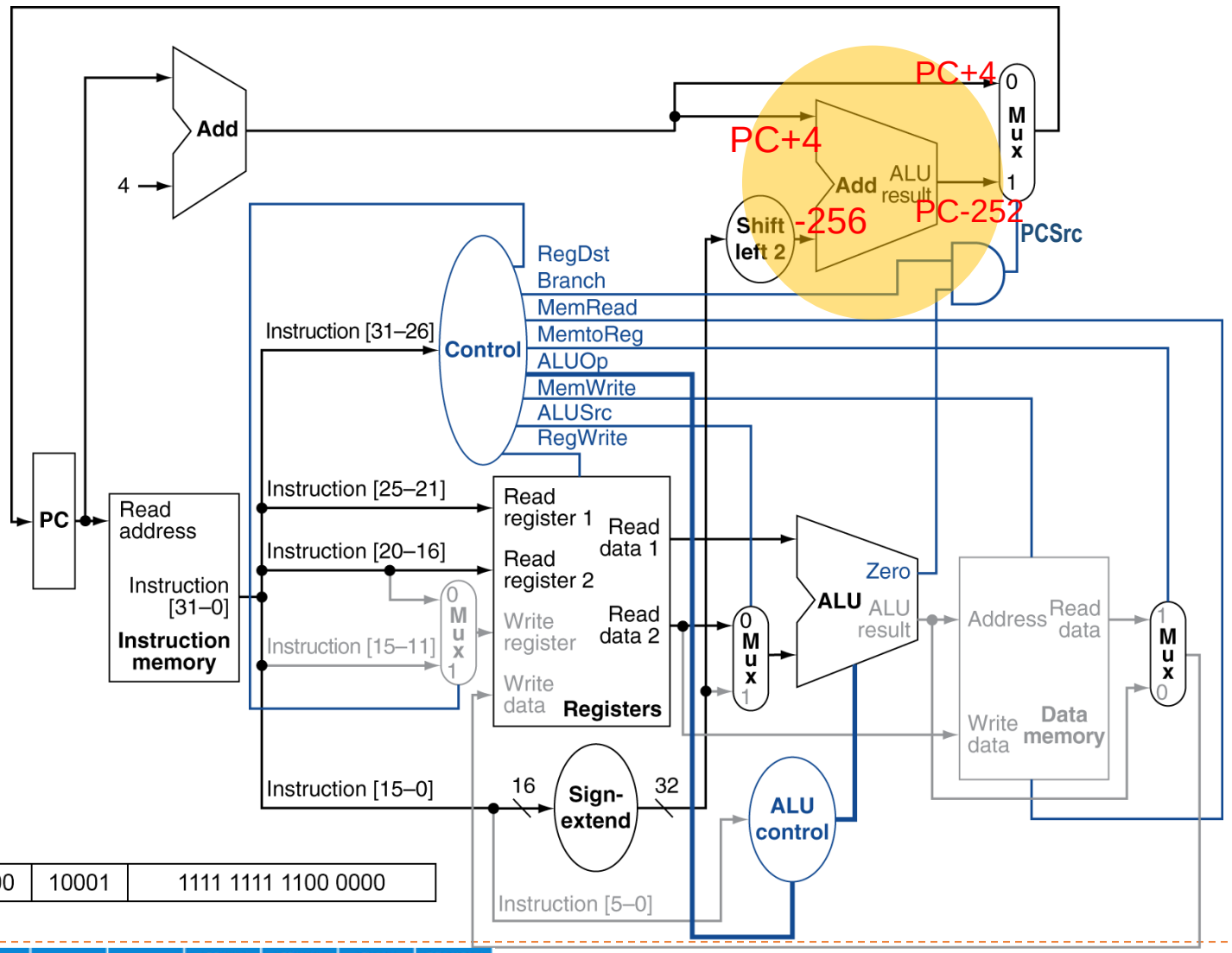
Branch-on-Equal Instruction – beq $s0, $s1, -64



000100	10000	10001	1111 1111 1100 0000

RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
X	0	1	0	1	0	0	0	X

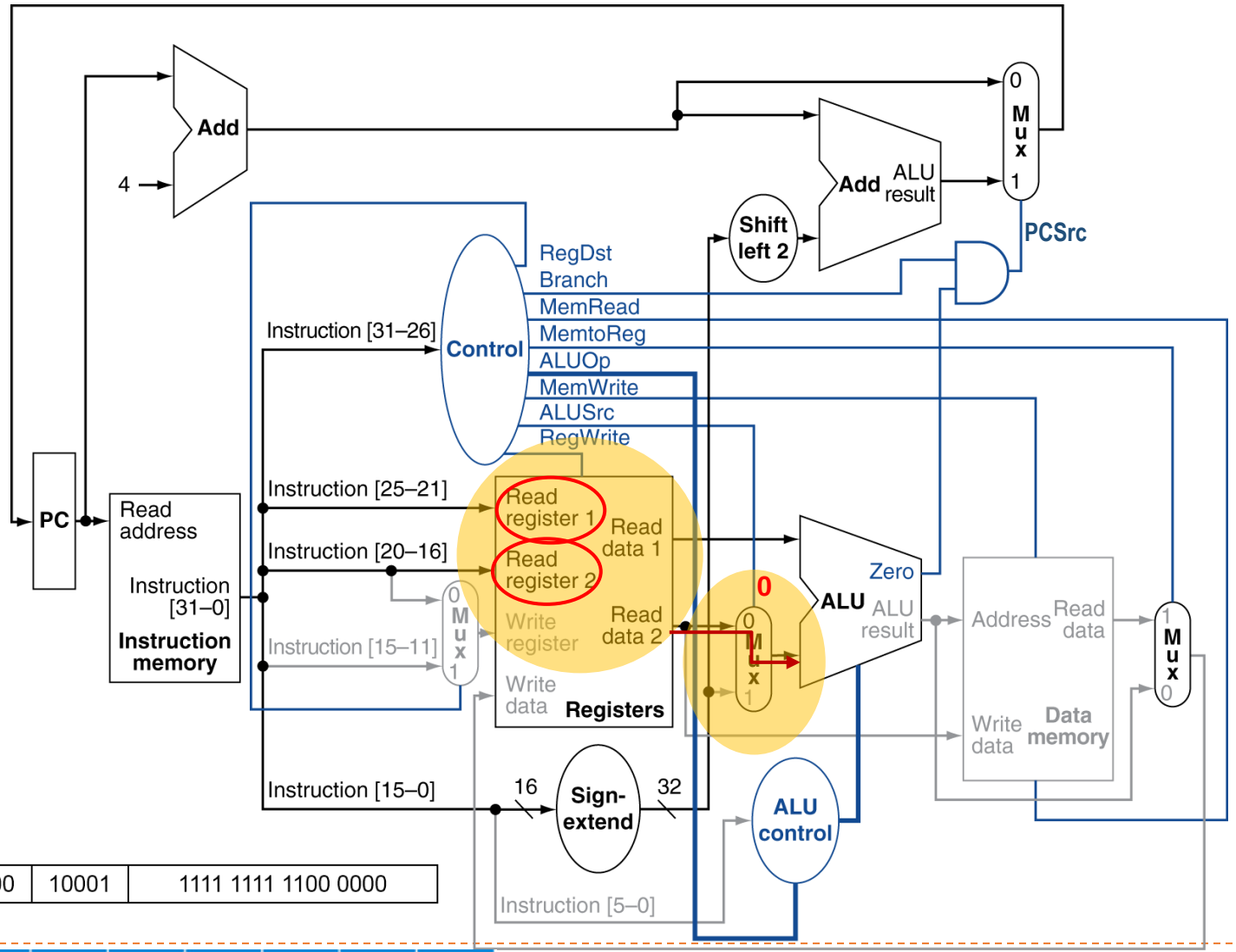
Branch-on-Equal Instruction – beq $s0, $s1, -64



000100	10000	10001	1111 1111 1100 0000

RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
X	0	1	0	1	0	0	0	X

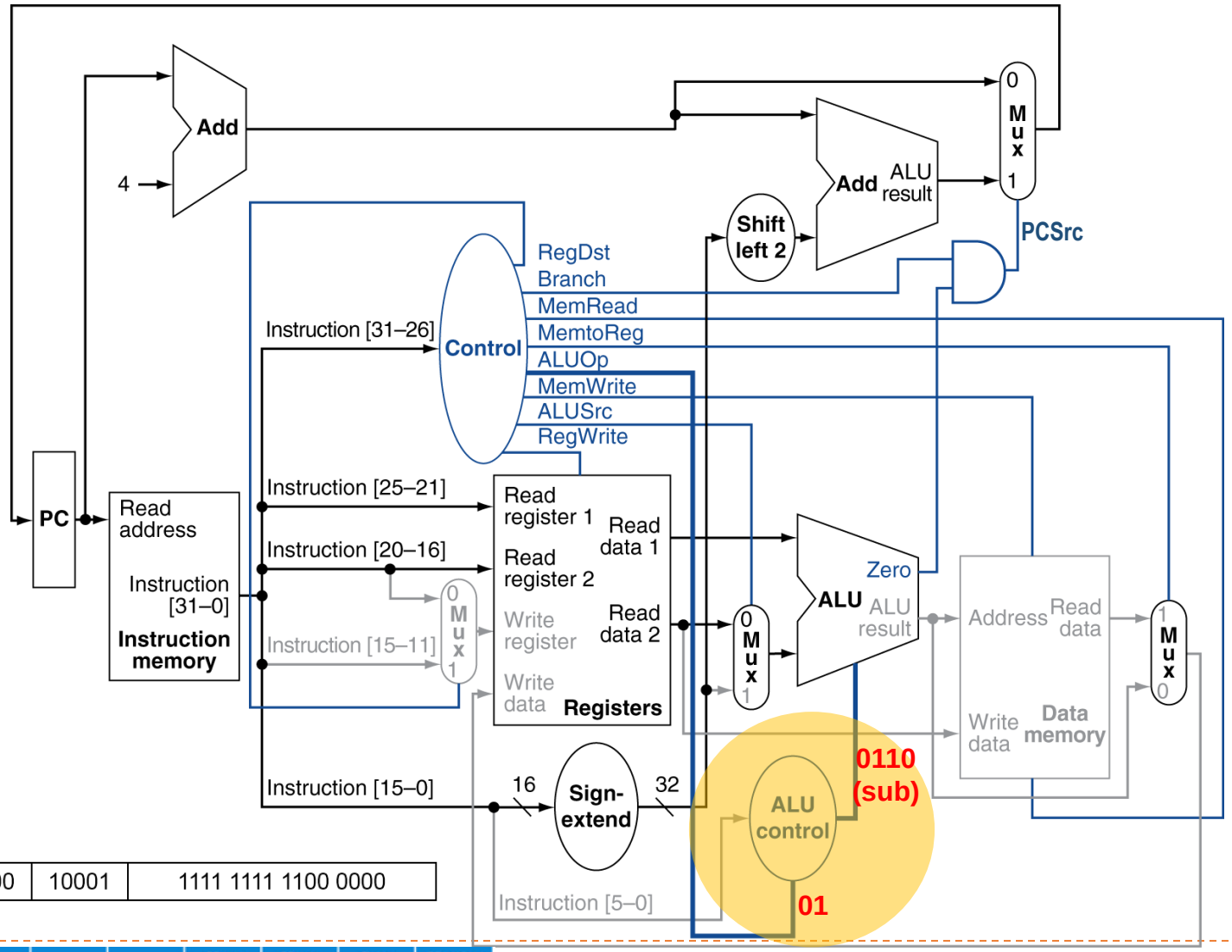
Branch-on-Equal Instruction – beq $s0, $s1, -64



000100	10000	10001	1111 1111 1100 0000

RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
X	0	1	0	1	0	0	0	X

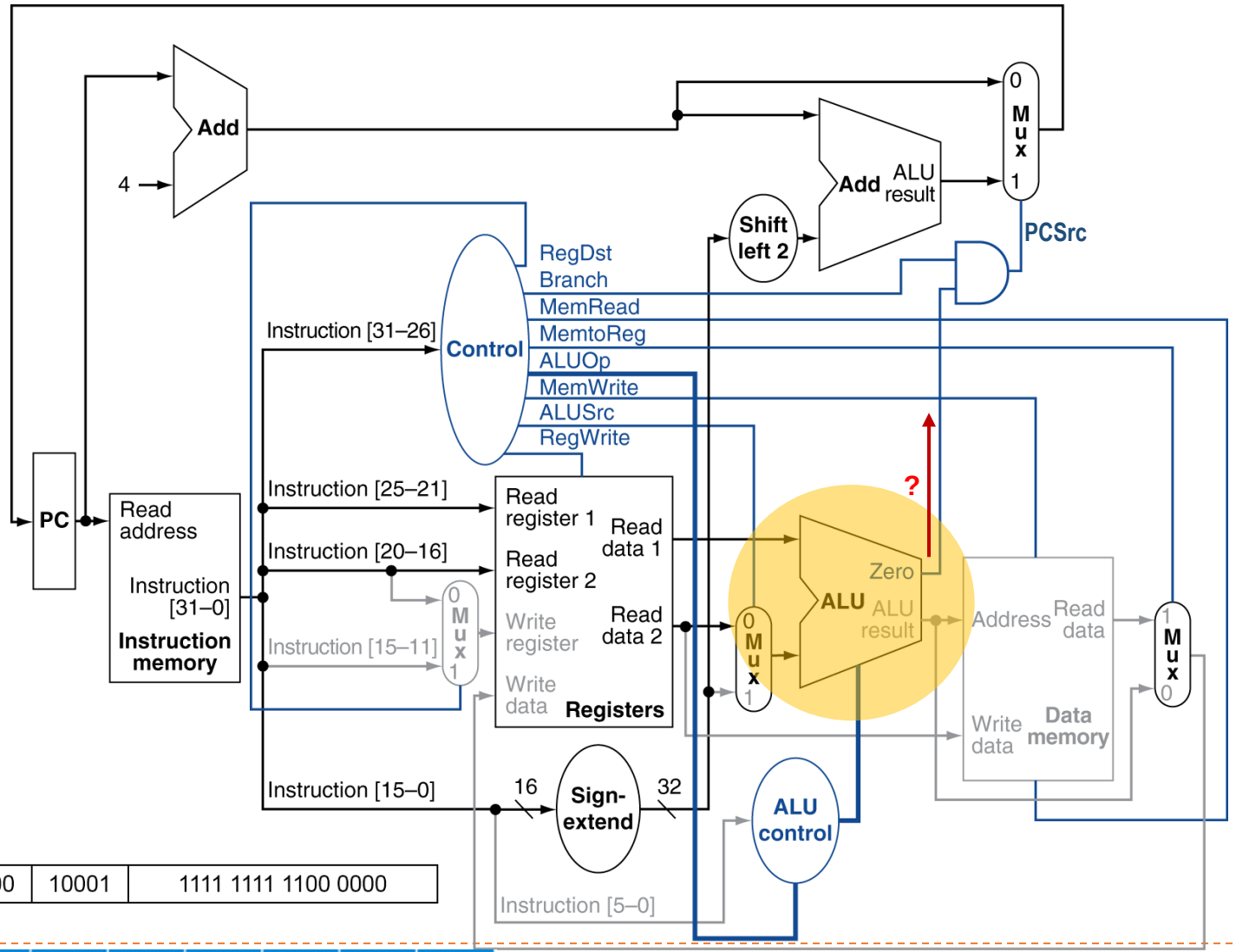
Branch-on-Equal Instruction – beq $s0, $s1, -64



000100	10000	10001	1111 1111 1100 0000

RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
X	0	1	0	1	0	0	0	X

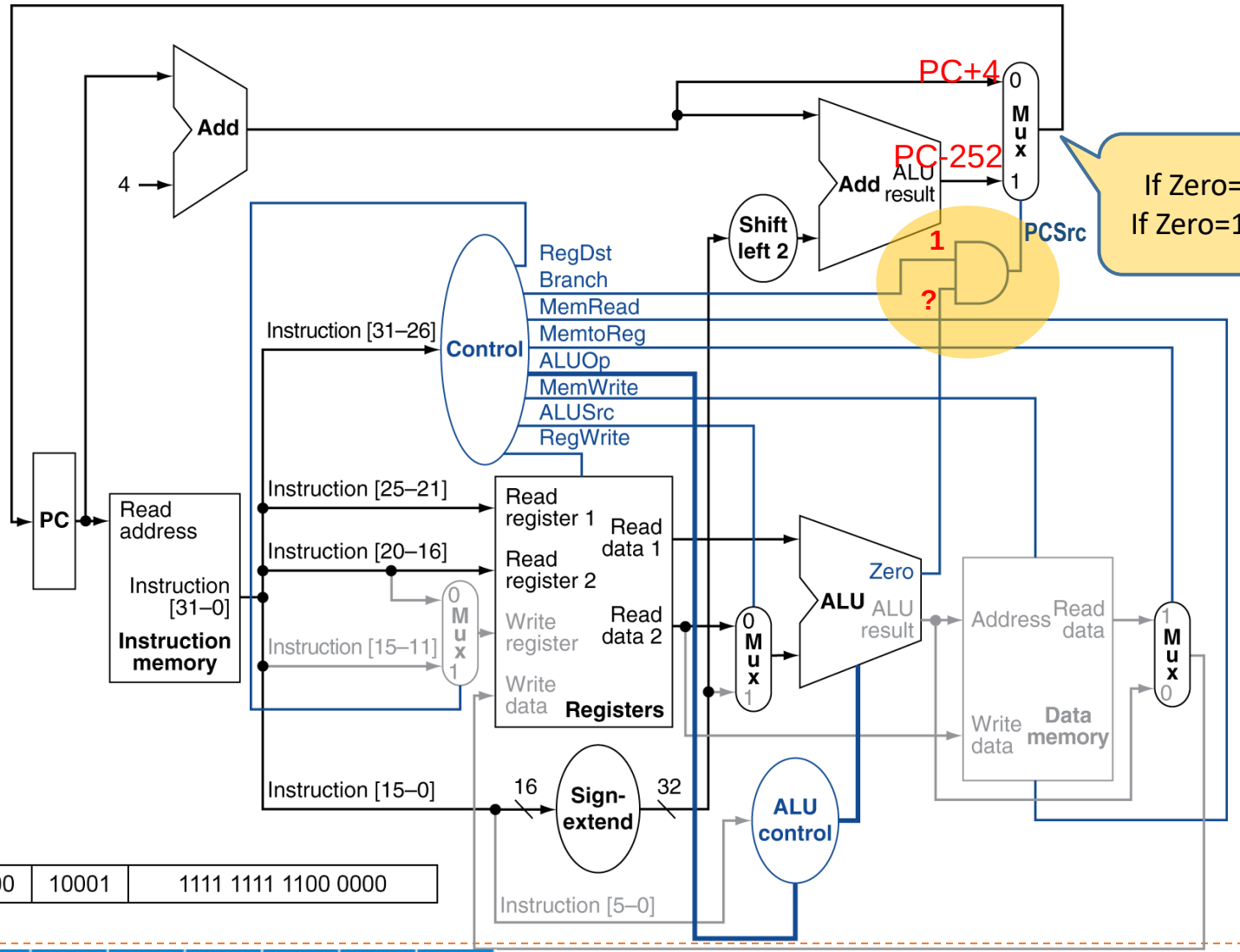
Branch-on-Equal Instruction – beq $s0, $s1, -64



000100	10000	10001	1111 1111 1100 0000

RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
X	0	1	0	1	0	0	0	X

Branch-on-Equal Instruction – beq $s0, $s1, -64



000100	10000	10001	1111 1111 1100 0000

RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
X	0	1	0	1	0	0	0	X

Reference Readings

- Patterson, "Computer Organization and Design RISC-V Edition"
  - Sec 4.1 – 4.4