System I

The Processor: Design

Li Lu

Zhejiang University

Overview

- Instruction execution in RISC-V
- Datapath
- Controller

RISC-V fields (format)

Name (Field Size)	Field 7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	Comments
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
B-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
J-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

- **Opcode**: basic operation of the instruction.
- **rs1** : the first register source operand.
- **rs2** : the second register source operand.
- **rd** : the register destination operand.
- **funct** : function, this field selects the specific variant of the operation in the op field.
- **Imm** : address or immediate

RISC-V operands

Name	Example	Comments
32 registers	x0-x31	Fast locations for data. In RISC-V, data must be in registers to perform arithmetic. Register x0 always equals 0.
2^{61} memory words	Memory[0], Memory[8] , …… , Memory[18446744073709551608]	Accessed only by data transfer instructions. RISC-V uses byte addresses, so sequential doubleword accesses differ by 8. Memory holds data structures, arrays, and spilled registers.

RISC-V register conventions

Name	Register name	Usage	Preserved On call?
x0	0	The constant value 0	n.a.
x1(ra)	1	Return address(link register)	yes
x2(sp)	2	Stack pointer	yes
x3(gp)	3	Global pointer	yes
x4(tp)	4	Thread pointer	yes
x5-x7	5-7	Temporaries	no
x8-x9	8-9	Saved	yes
x10-x17	10-17	Arguments/results	no
x18-x27	18-27	Saved	yes
x28-x31	28-31	Temporaries	no

RISC-V assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add x5,x6,x7	x5=x6 + x7	Add two source register operands
	subtract	sub x5,x6,x7	x5=x6 - x7	First source register subtracts second one
	add immediate	addi x5,x6,20	x5=x6+20	Used to add constants
Data transfer	load doubleword	ld x5, 40(x6)	x5=Memory[x6+40]	doubleword from memory to register
	store doubleword	sd x5, 40(x6)	Memory[x6+40]=x5	doubleword from register to memory
Logical	and	and x5, x6, 3	x5=x6 & 3	Arithmetic shift right by register
	inclusive or	or x5,x6,x7	x5=x6 | x7	Bit-by-bit OR
Conditional Branch	branch if equal	beq x5, x6, 100	if(x5 == x6) go to PC+100	PC-relative branch if registers equal
	branch if not equal	bne x5, x6, 100	if(x5 != x6) go to PC+100	PC-relative branch if registers not equal
Unconditional Branch	jump and link	jal x1, 100	x1 = PC + 4; go to PC+100	PC-relative procedure call
	jump and link register	jalr x1, 100(x5)	x1 = PC + 4; go to x5+100	procedure return; indirect call

Overview

- Instruction execution in RISC-V
- Datapath
- Controller

Instruction execution in RISC-V

- **Fetch** :
  - Take instructions from the instruction memory
  - Modify PC to point the next instruction
- **Instruction decoding & Read Operand**:
  - Will be translated into machine control command
  - Reading Register Operands, whether or not to use
- **Executive Control**:
  - Control the implementation of the corresponding ALU operation
- **Memory access**:
  - Write or Read data from memory
  - Only ld/sd
- **Write results to register**:
  - If it is R-type instructions, ALU results are written to rd
  - If it is I-type instructions, memory data are written to rd
- **Modify PC** for branch instructions

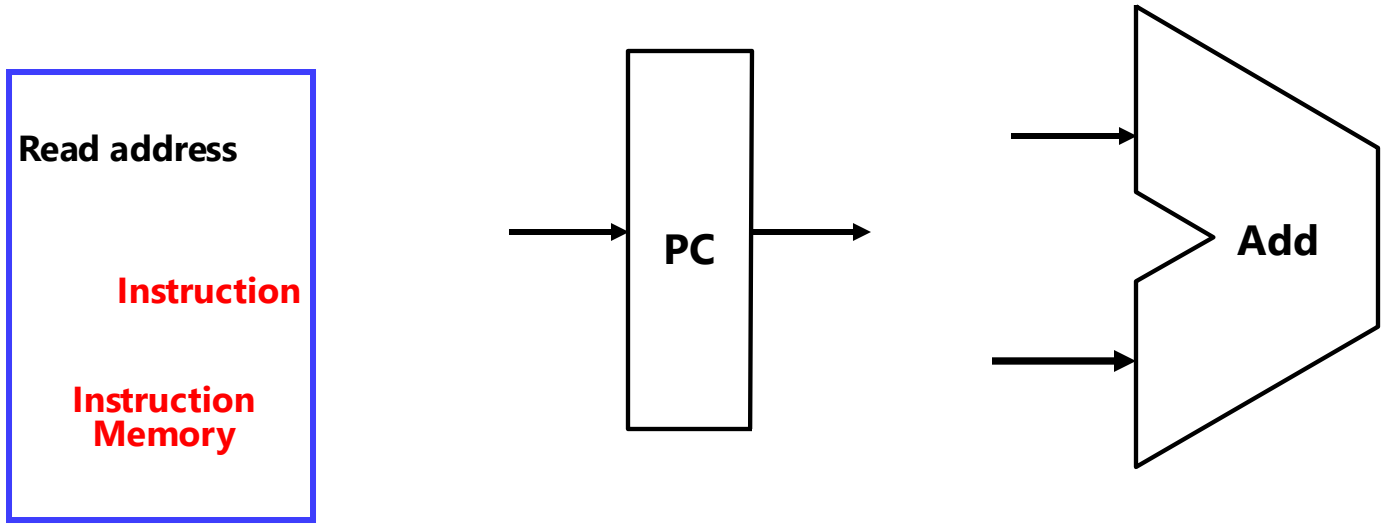
Instruction Fetching

- **Fetch** :
  - Take instructions from the instruction memory
  - Modify PC to point the next instruction
- **Instruction decoding & Read Operand**:
  - Will be translated into machine control command
  - Reading Register Operands, whether or not to use

Instruction fetching three elements

Data Stream of Instruction fetching

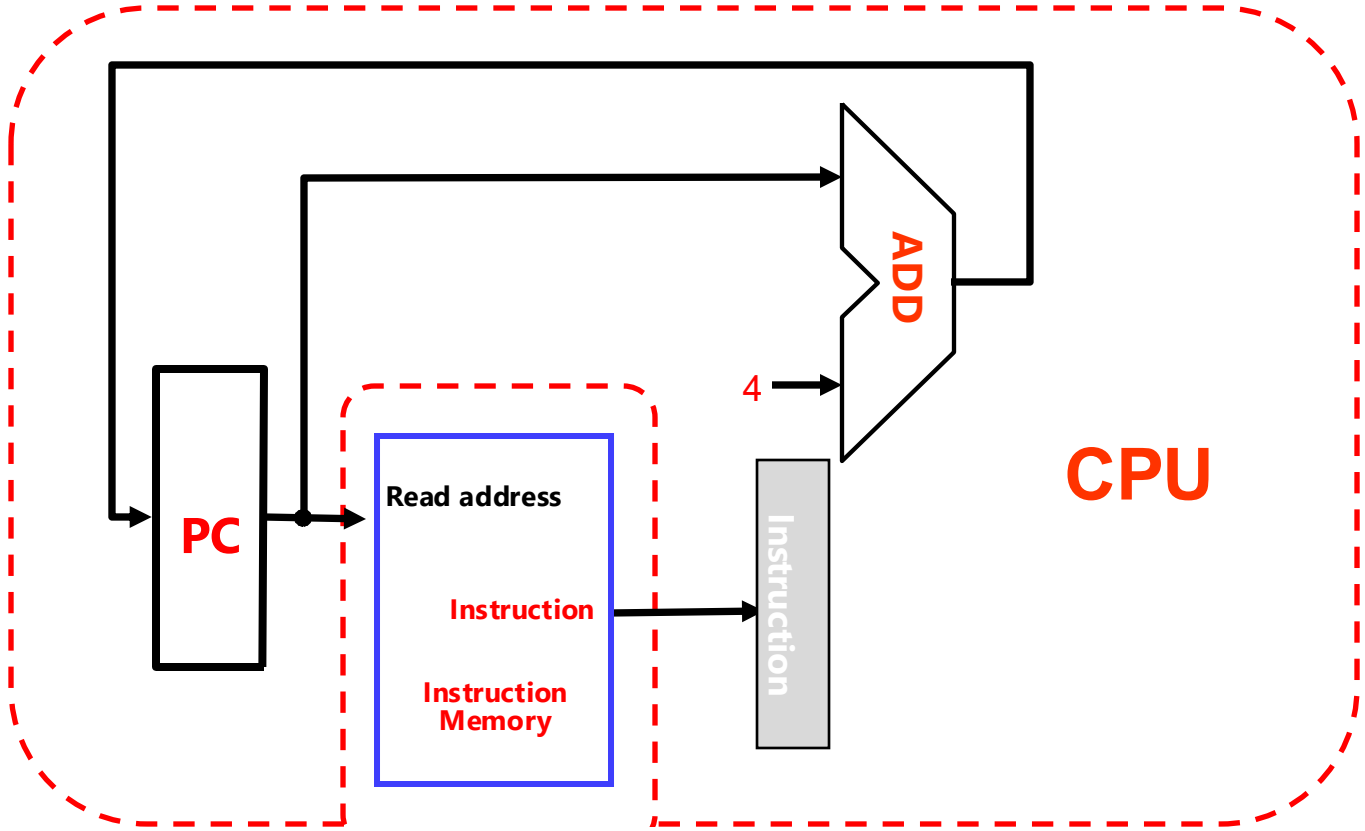
How to connect? **Who?**



Instruction memory **Program counter** **Adder**

Instruction fetching unit

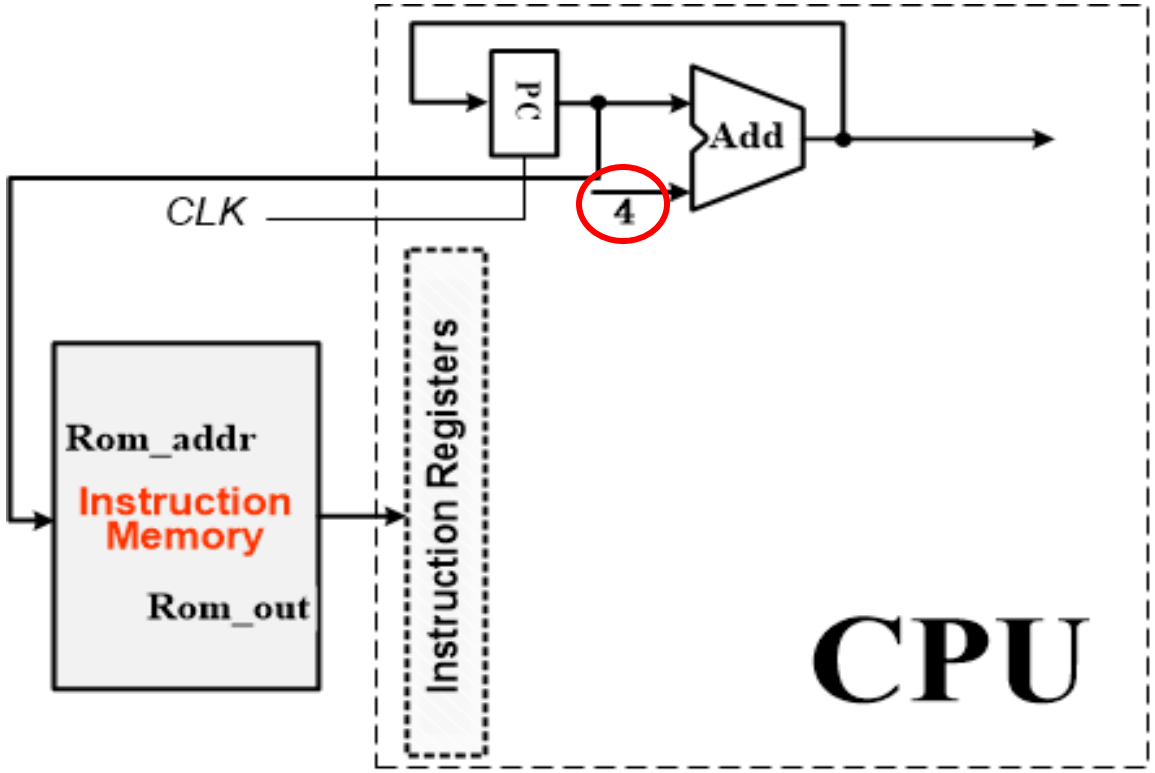
- Instruction Register



Where is the sequential logic?

How simple it is!

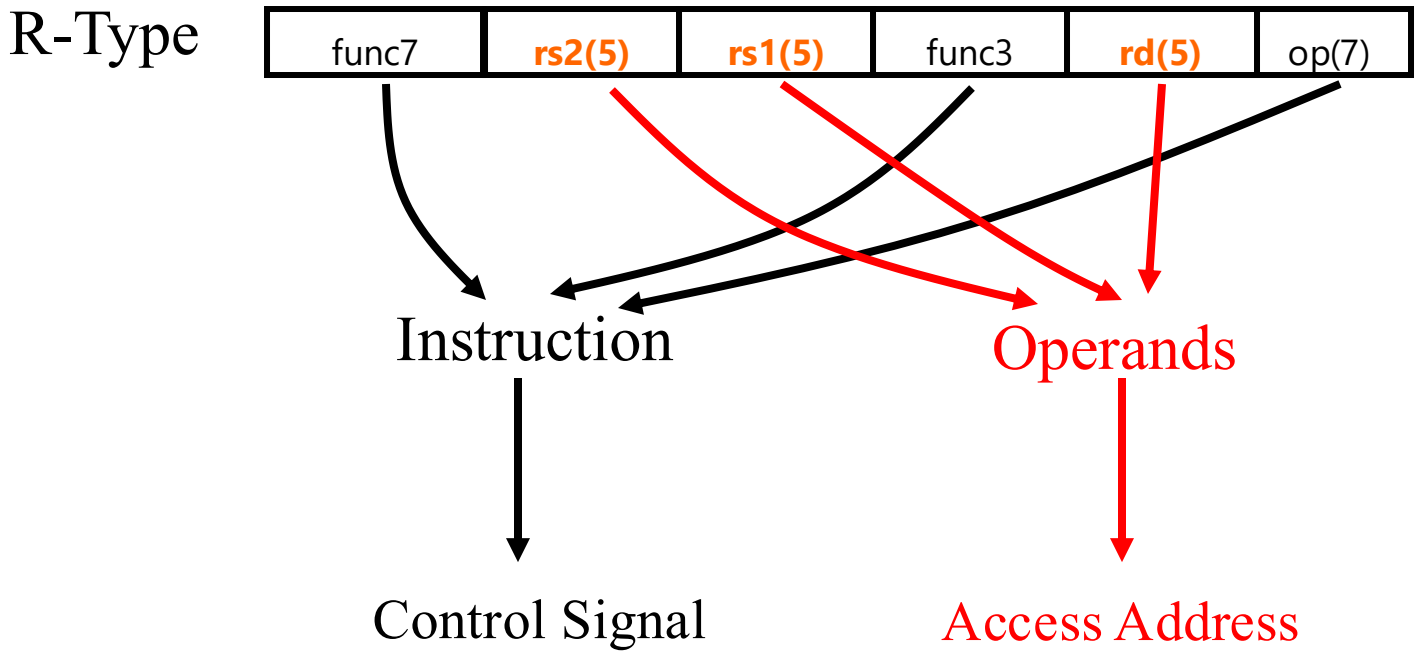
- Why PC+4?



Instruction execution in RISC-V

- **Fetch** :
  - Take instructions from the instruction memory
  - Modify PC to point the next instruction
- **Instruction decoding & Read Operand**:
  - Will be translated into machine control command
  - Reading Register Operands, whether or not to use

Instruction Decoding



How about other types of instructions?

Instruction Decoding

Name (Field Size)	Field 7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	Comments
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
B-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
J-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

Instruction

Control Signal

Operands

Access Address

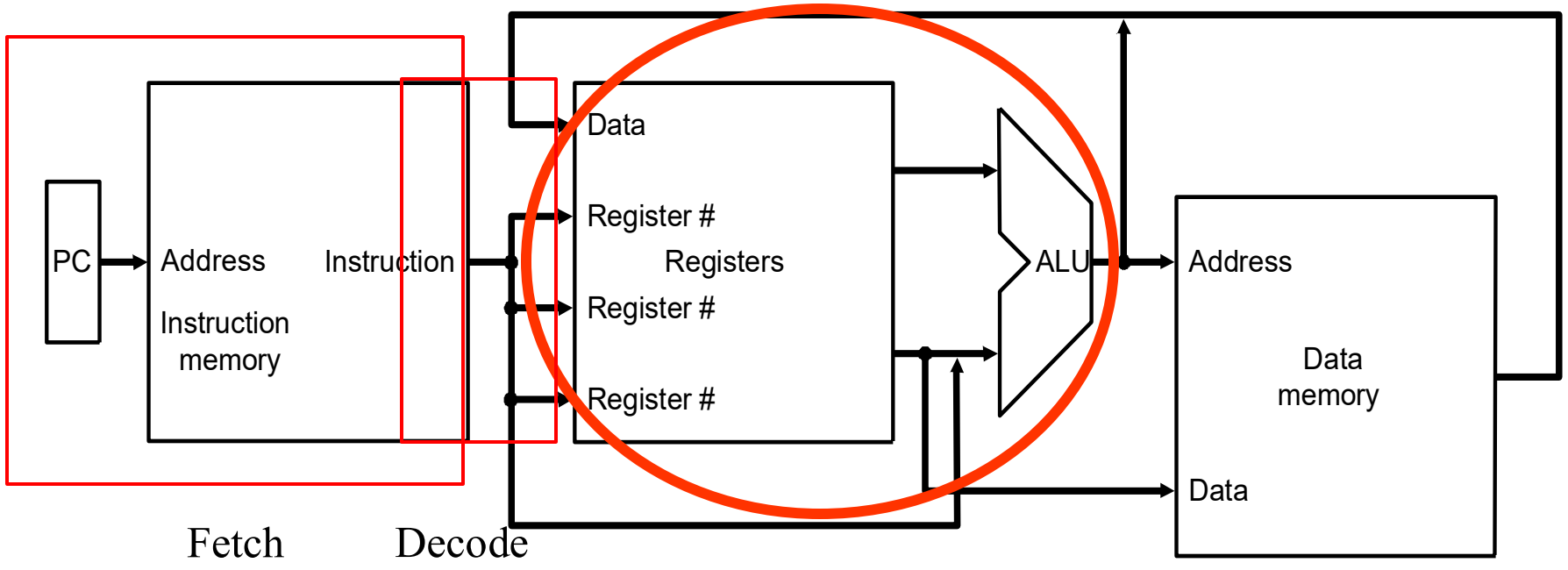
Why separate them in decoding?

Following execution

- **Executive Control**:
  - Control the implementation of the corresponding ALU operation
- **Memory access**:
  - Write or Read data from memory
  - Only ld/sd
- **Write results to register**:
  - If it is R-type instructions, ALU results are written to rd
  - If it is I-type instructions, memory data are written to rd
- **Modify PC** for branch instructions

More Implementation Details

- Abstract / Simplified View:



Path Built using Multiplexer

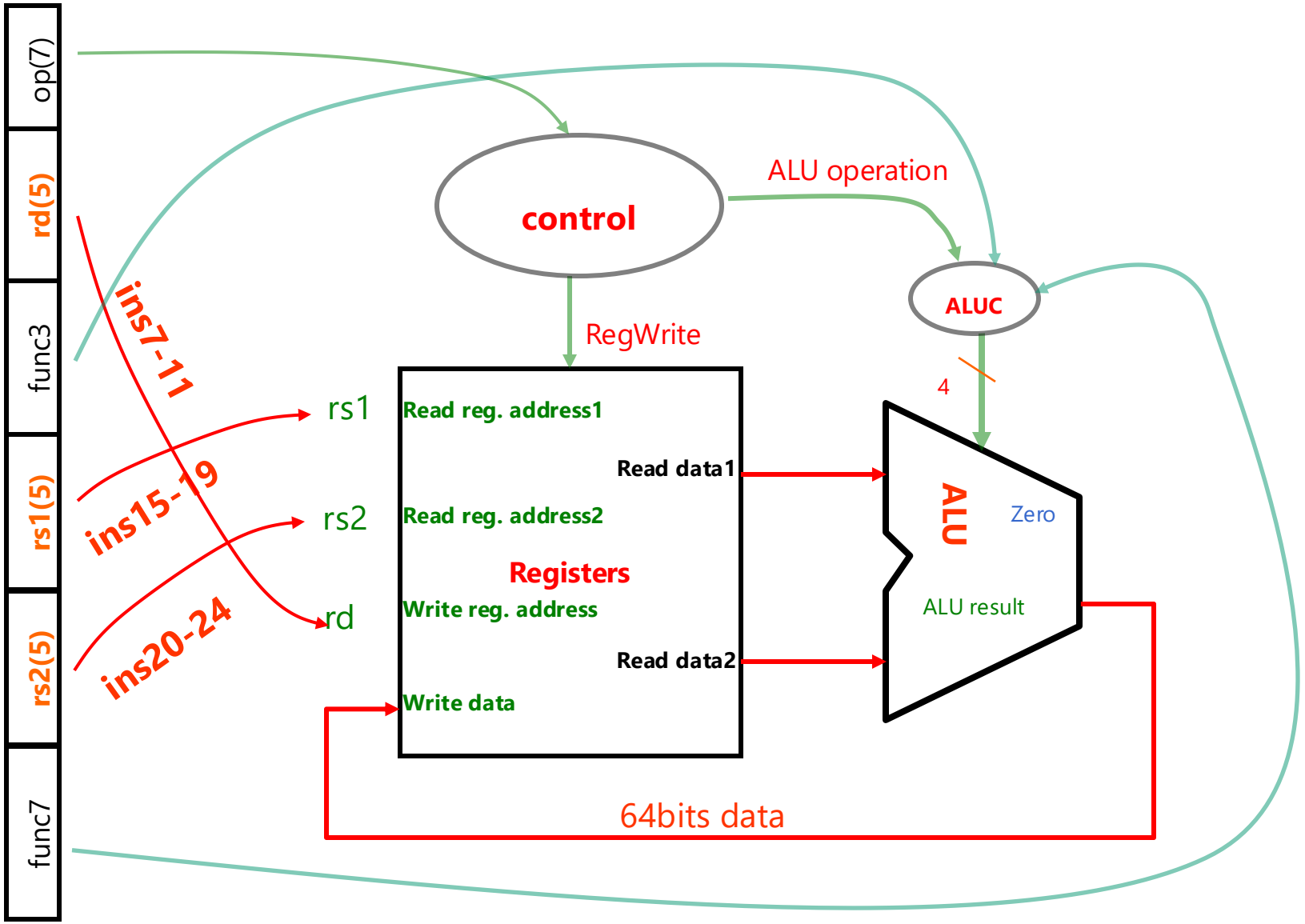
- R-type instruction Datapath
- I-type instruction Datapath
  - For ALU
  - For load
- S-type (store) instruction Datapath
- B-type (branch) instruction Datapath
- J-type instruction Datapath
  - For Jump
- First, revisit the instruction format

Instruction Format Revisit

	31 25	24 20	19 15	14 12	11 7	6 0
R	func7	rs2	rs1	func3	rd	opcode
I	imm[11:0]		rs1	func3	rd	opcode
S	imm[11:5]	rs2	rs1	func3	imm[4:0]	opcode
B	imm[12:10:5]	rs2	rs1	func3	imm[4:1|11]	opcode
J	imm[20|10:1|11|19:12]				rd	opcode

- Then, look at the data flow within instruction execution

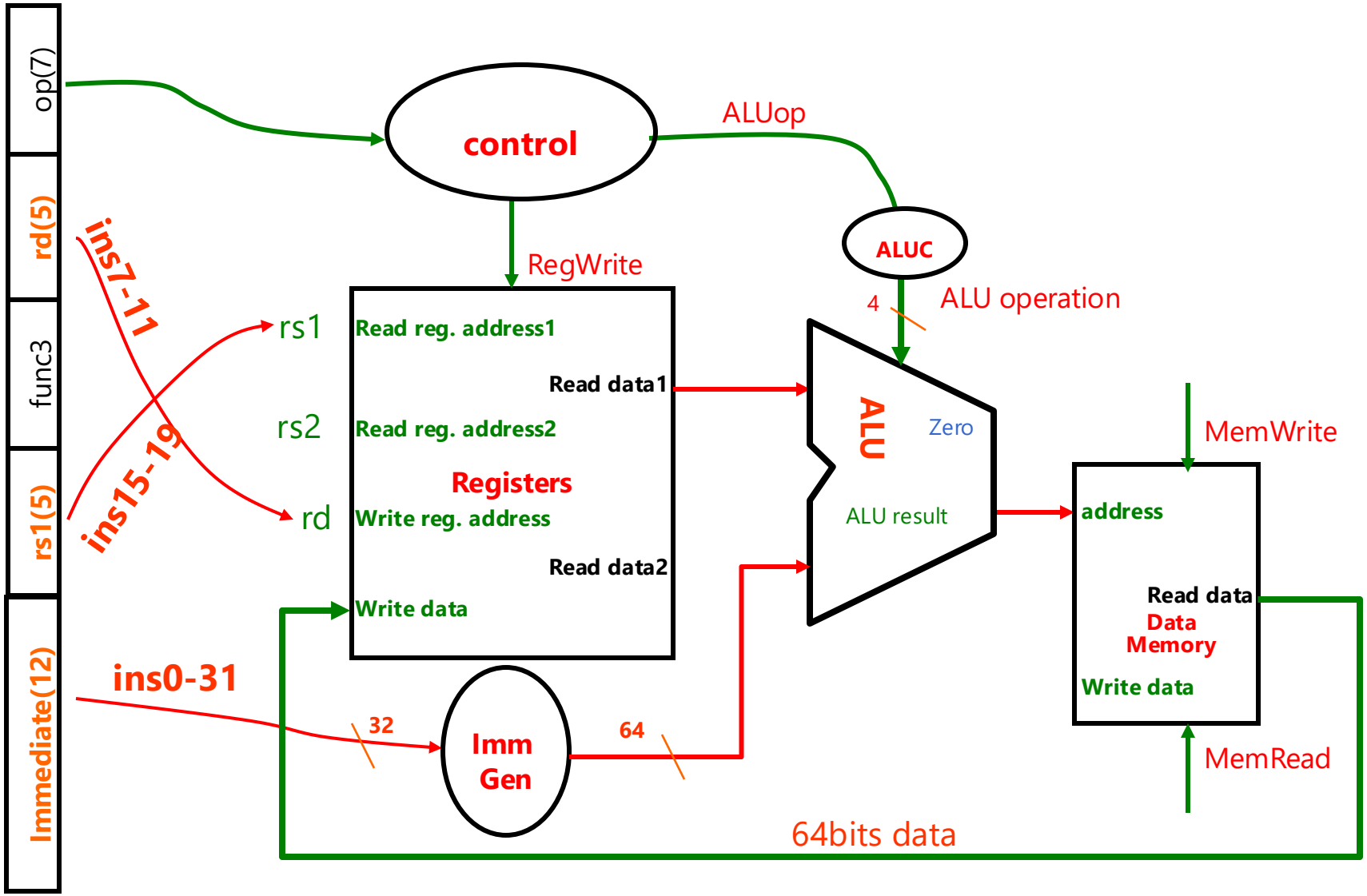
R-type Instruction & Data Stream



add x9, x20, x21

1. Read two register operands
2. Perform arithmetic/logical operation
3. Write register result

I-type(load) Instruction & Data Stream

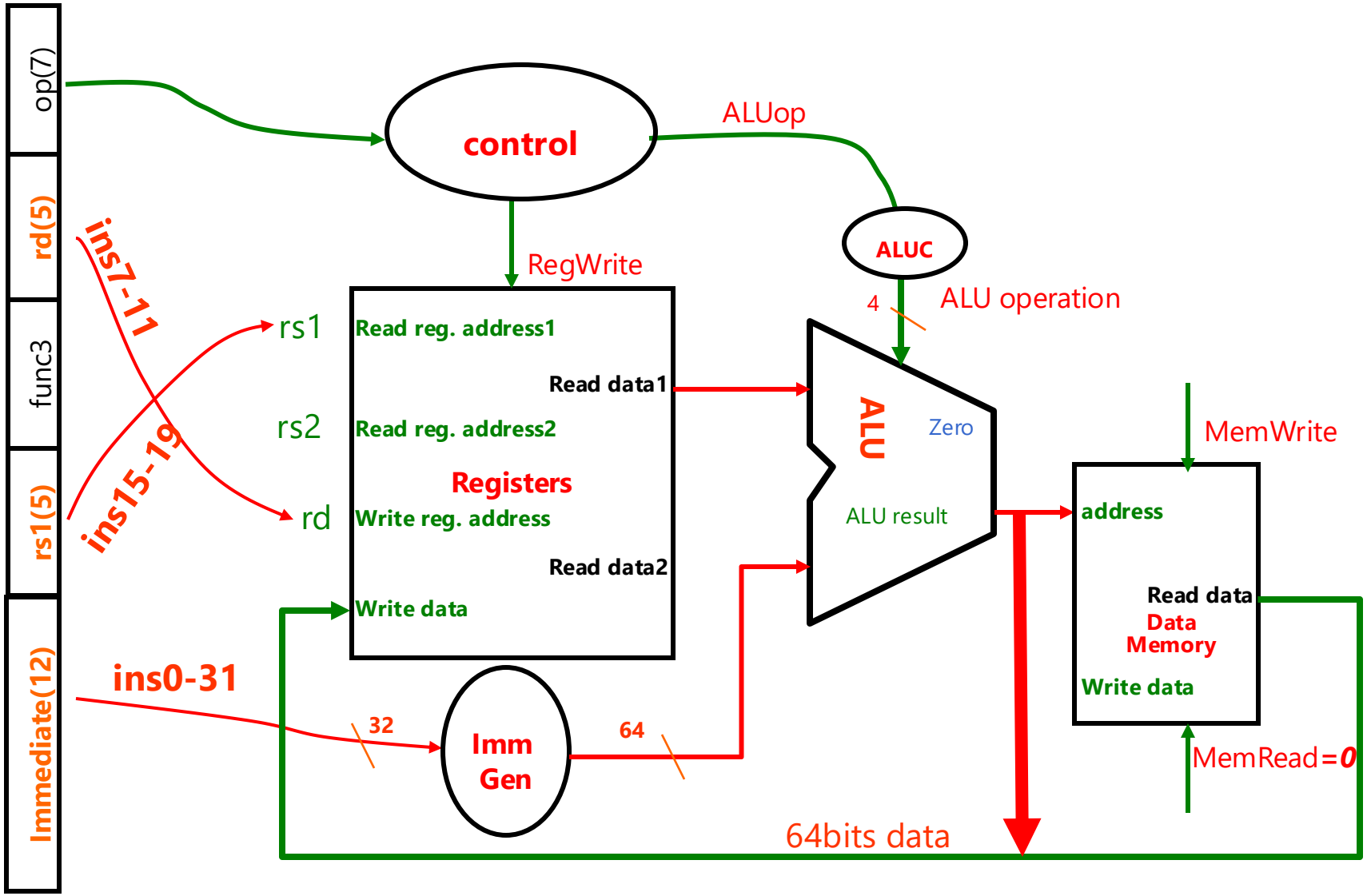


ld x1, 200(x2)

1. Read register operands
2. Calculate address using 12-bit offset
   1. Use ALU, but sign-extend offset
3. Read memory
4. Update register

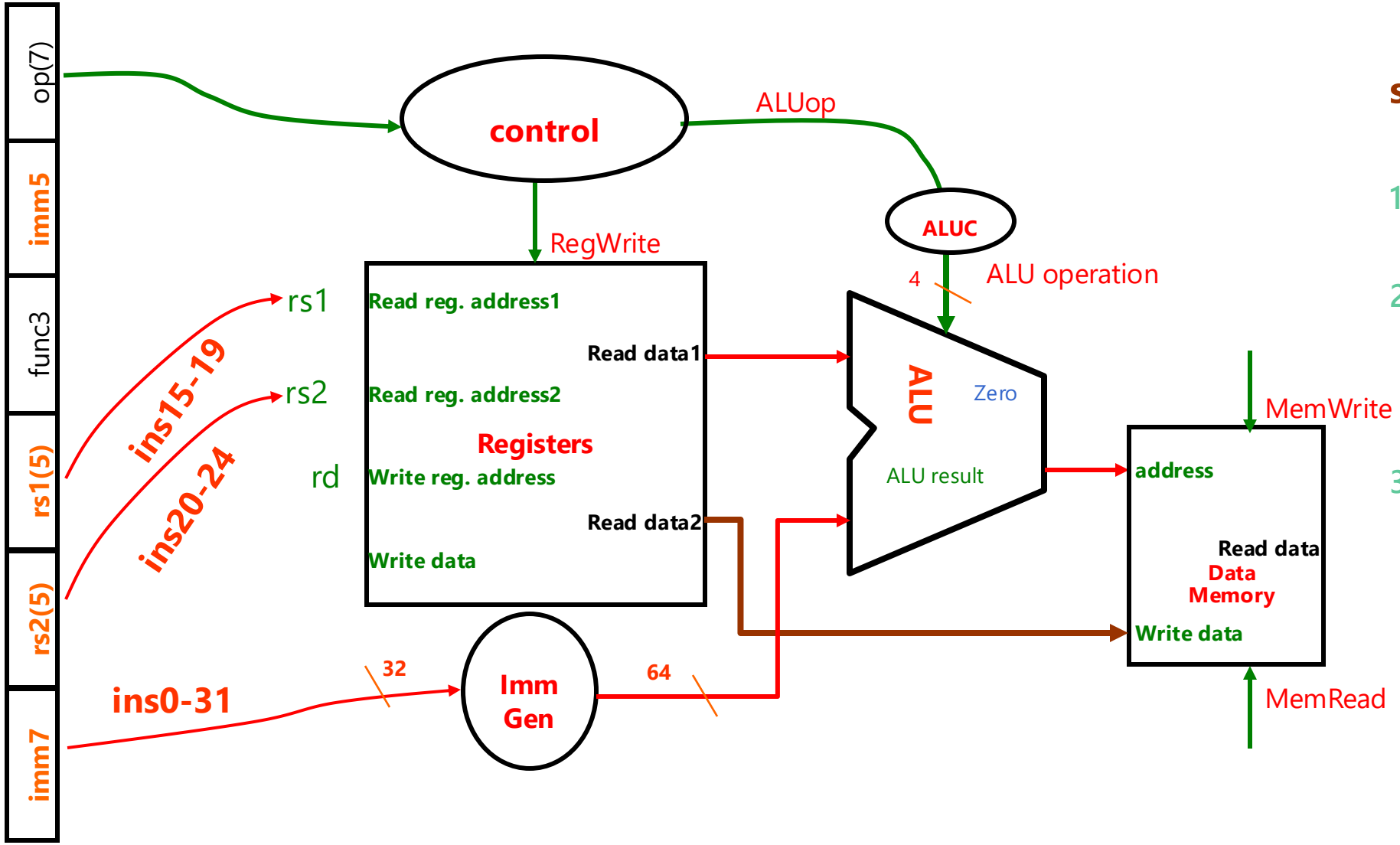
addi x1, x2, 4?

I-type Instruction & Data Stream



addi x1, x2, 4

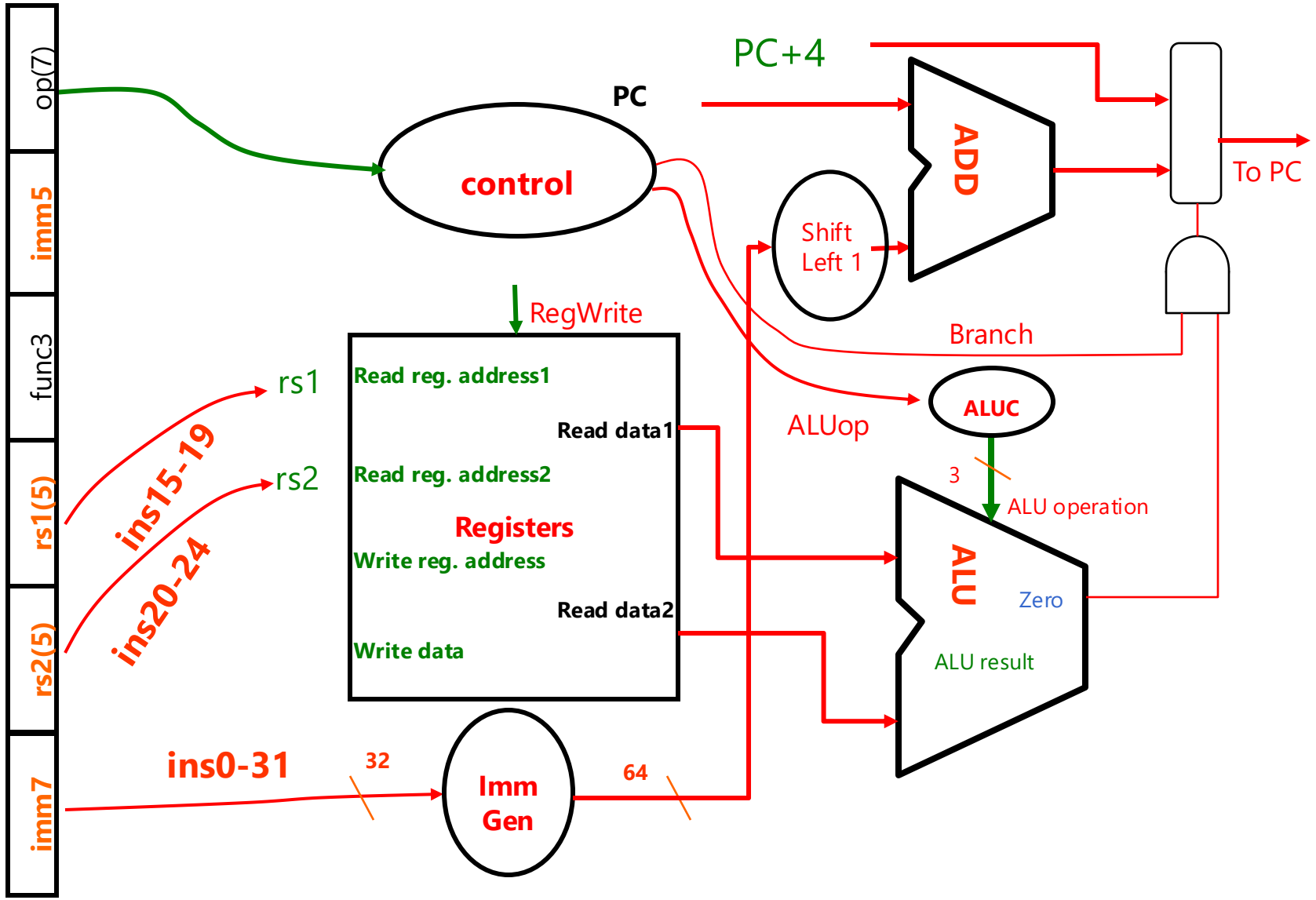
S-type(store) Instruction & Data Stream



sd x1, 200(x2)

1. Read register operands
2. Calculate address using 12-bit offset
   1. Use ALU, but sign-extend offset
3. Write register value to memory

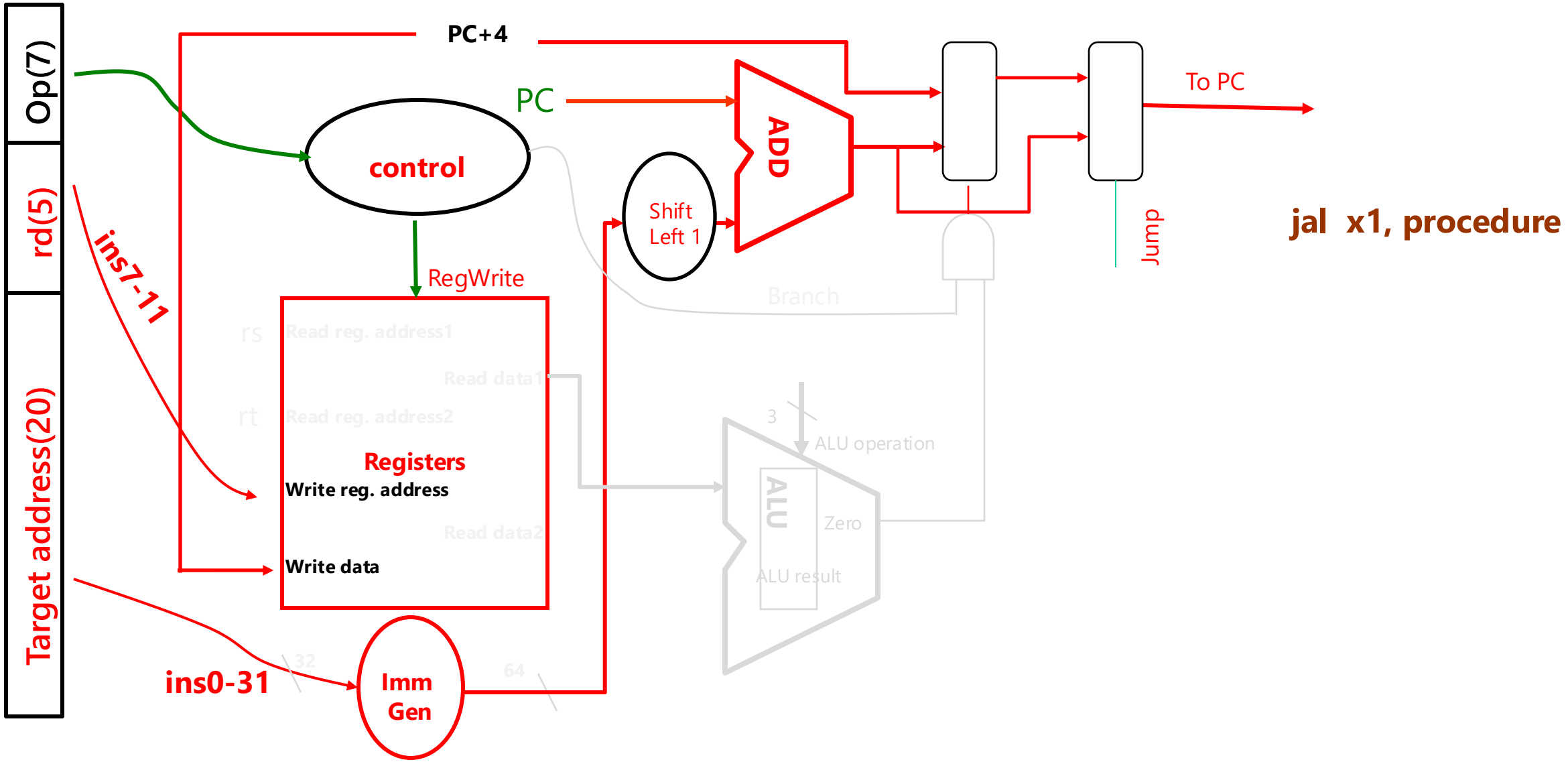
B-type Instruction & Data Stream of beq



beq x1, x2，200

- **Read register operands**
- **Compare operands**
  - Use ALU, subtract and check Zero output
- **Calculate target address**
  - Sign-extend displacement
  - Shift left 1 place (halfword displacement)
  - Add to PC value

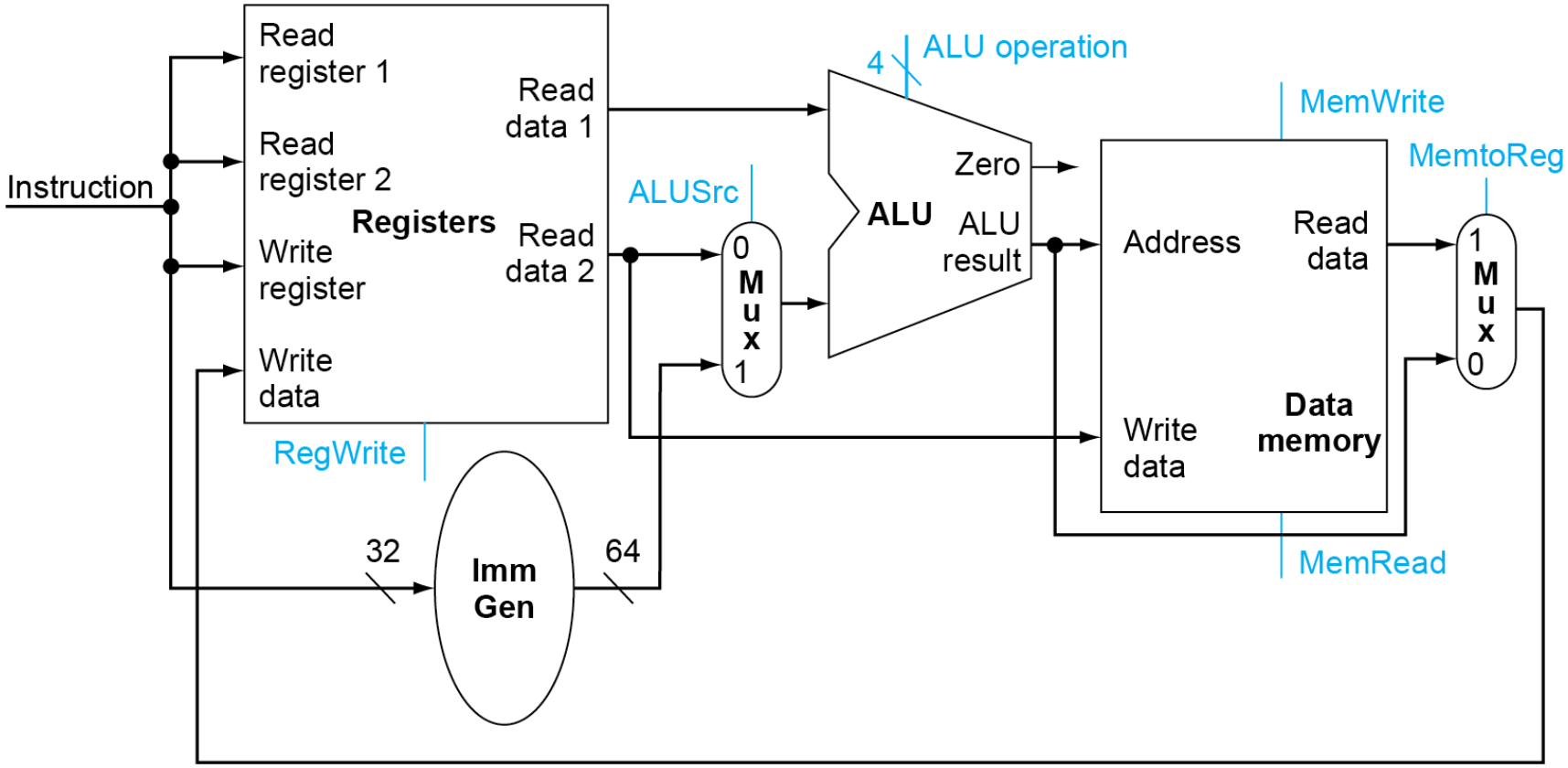
J-type Instruction



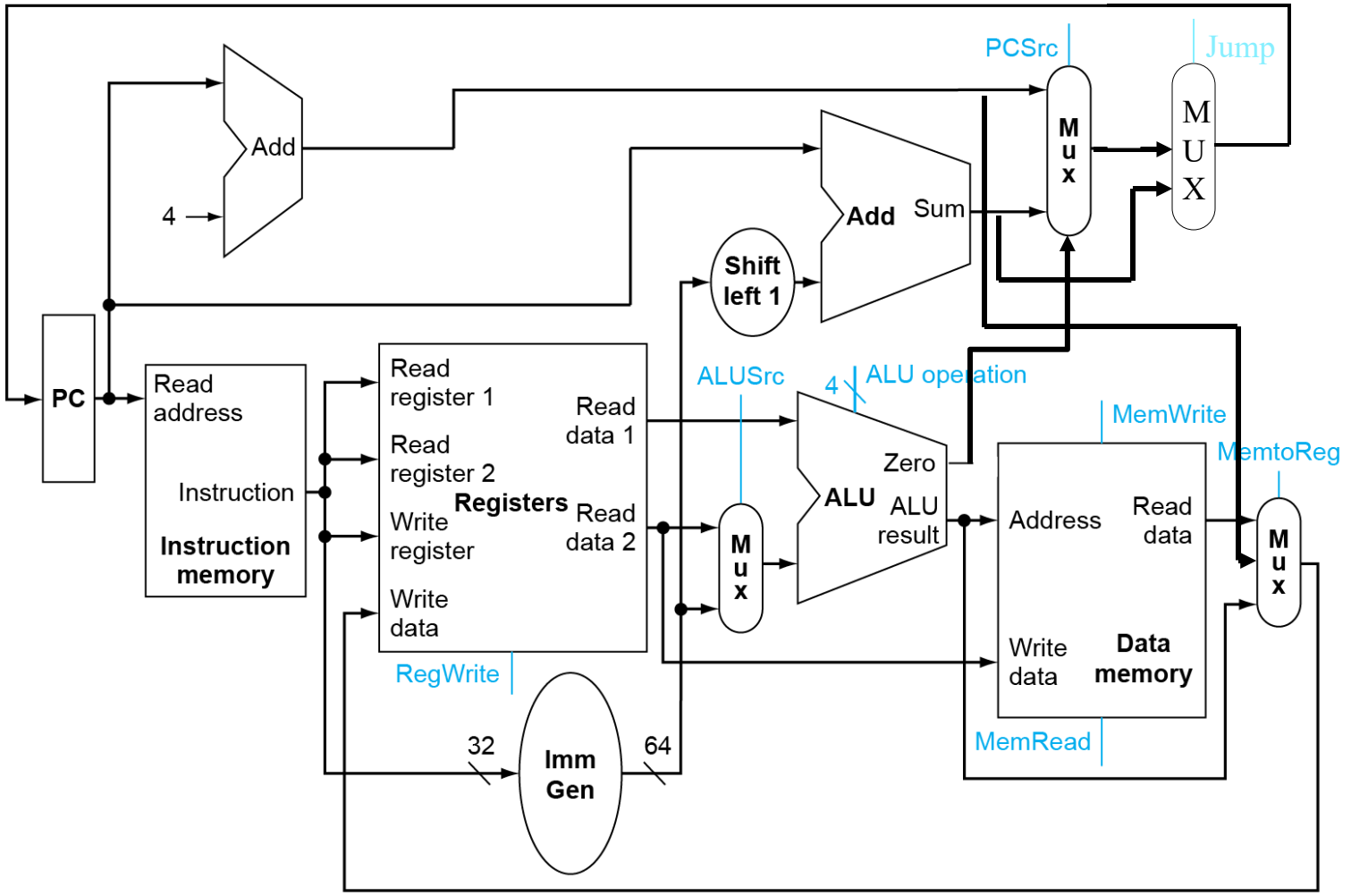
Composing the Elements

- First-cut data path does an instruction in one clock cycle
  - Each datapath element can **only** do **one** function at a time
  - Hence, we need separate instruction and data memories
- Use multiplexers where alternate data sources are used for different instructions

R-Type/Load/Store Datapath

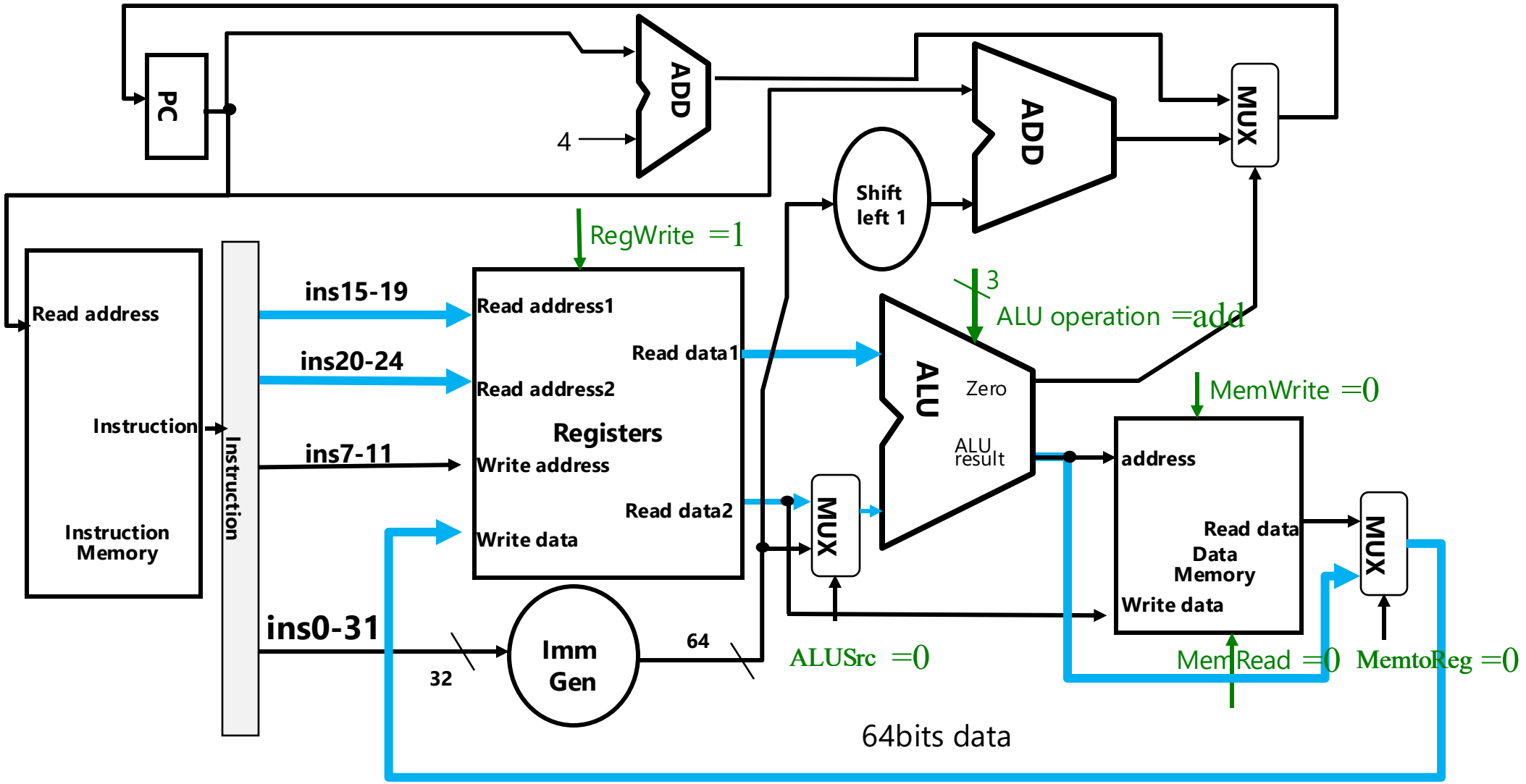


Full Datapath



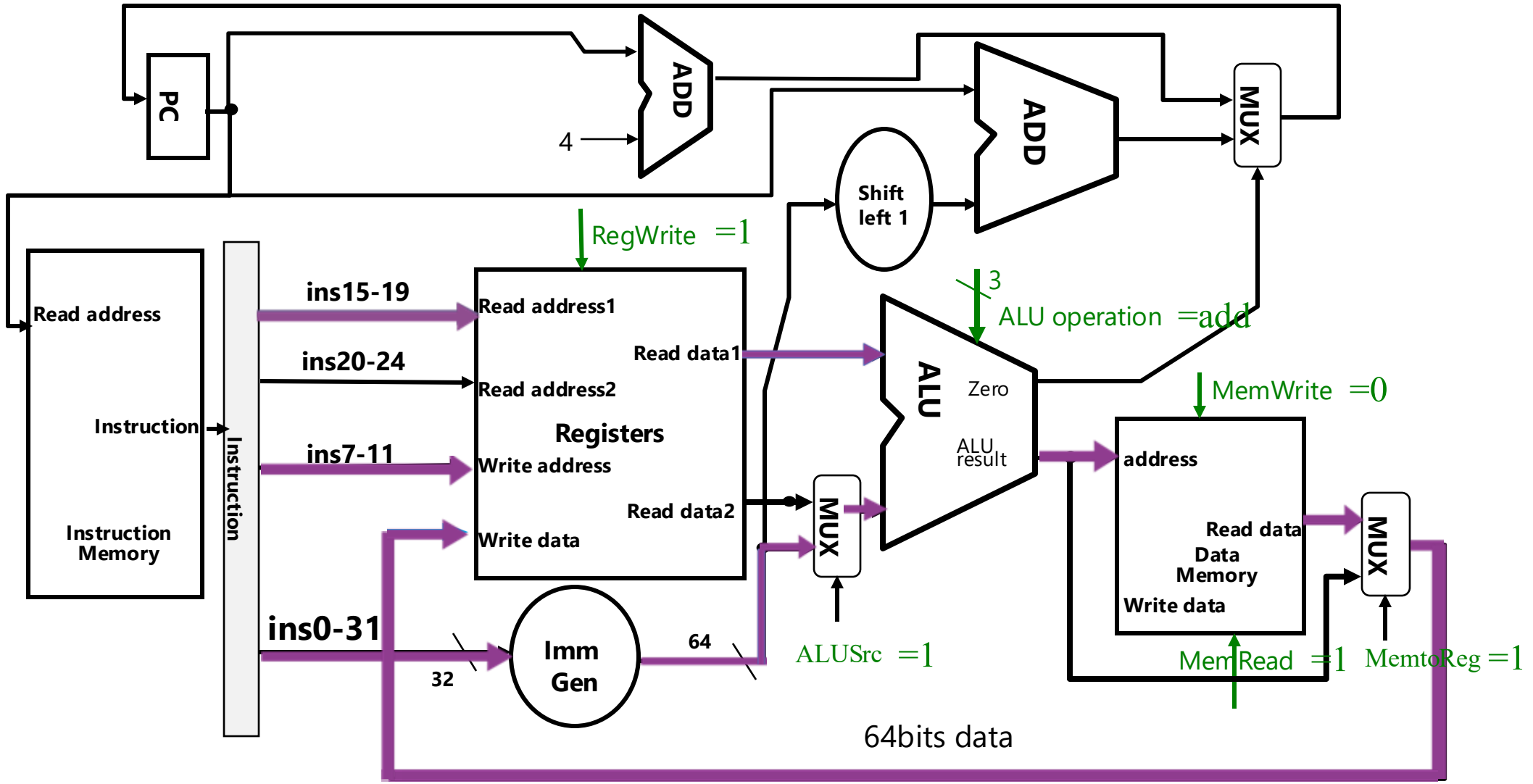
Full datapath

R



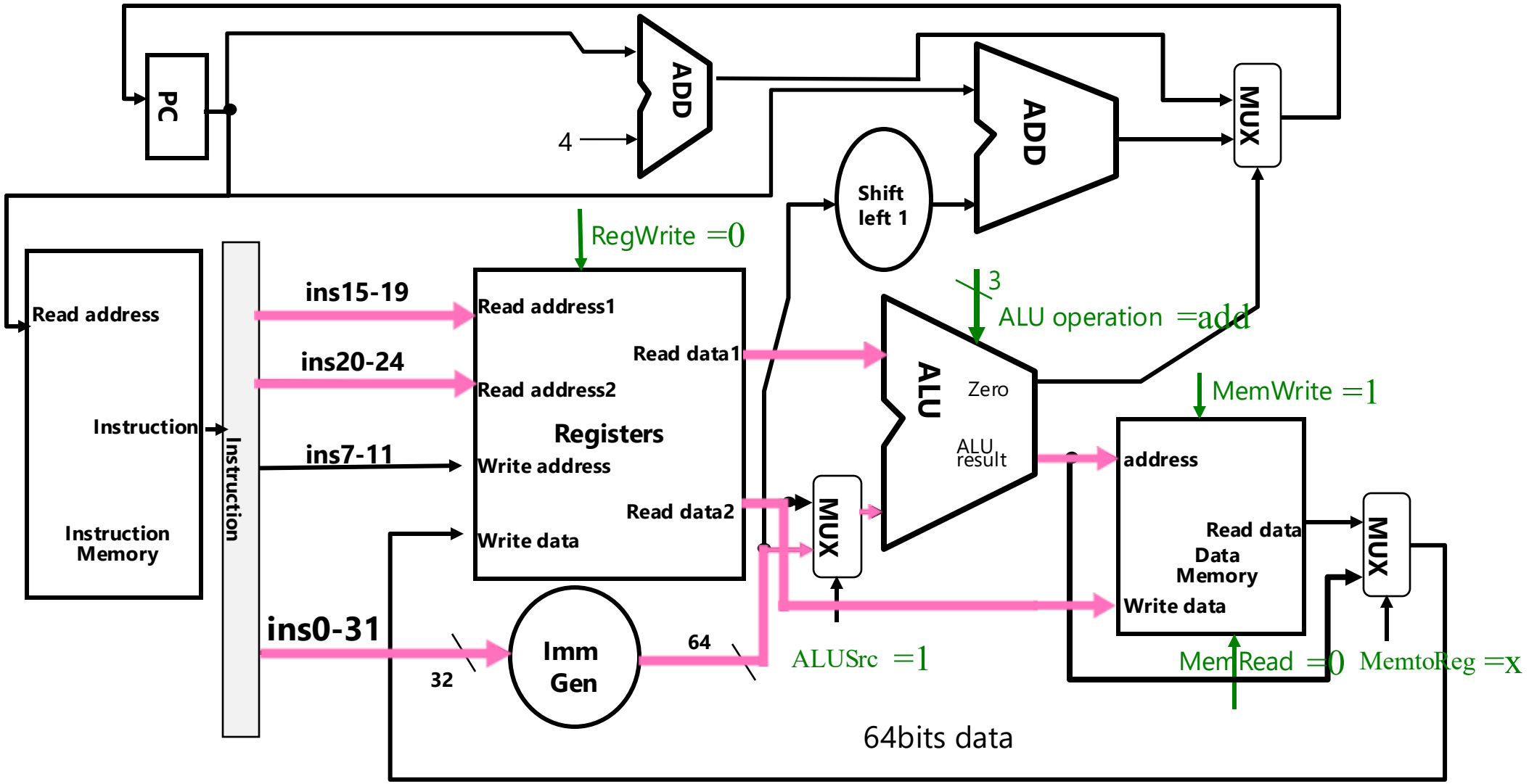
Full datapath

I-ld



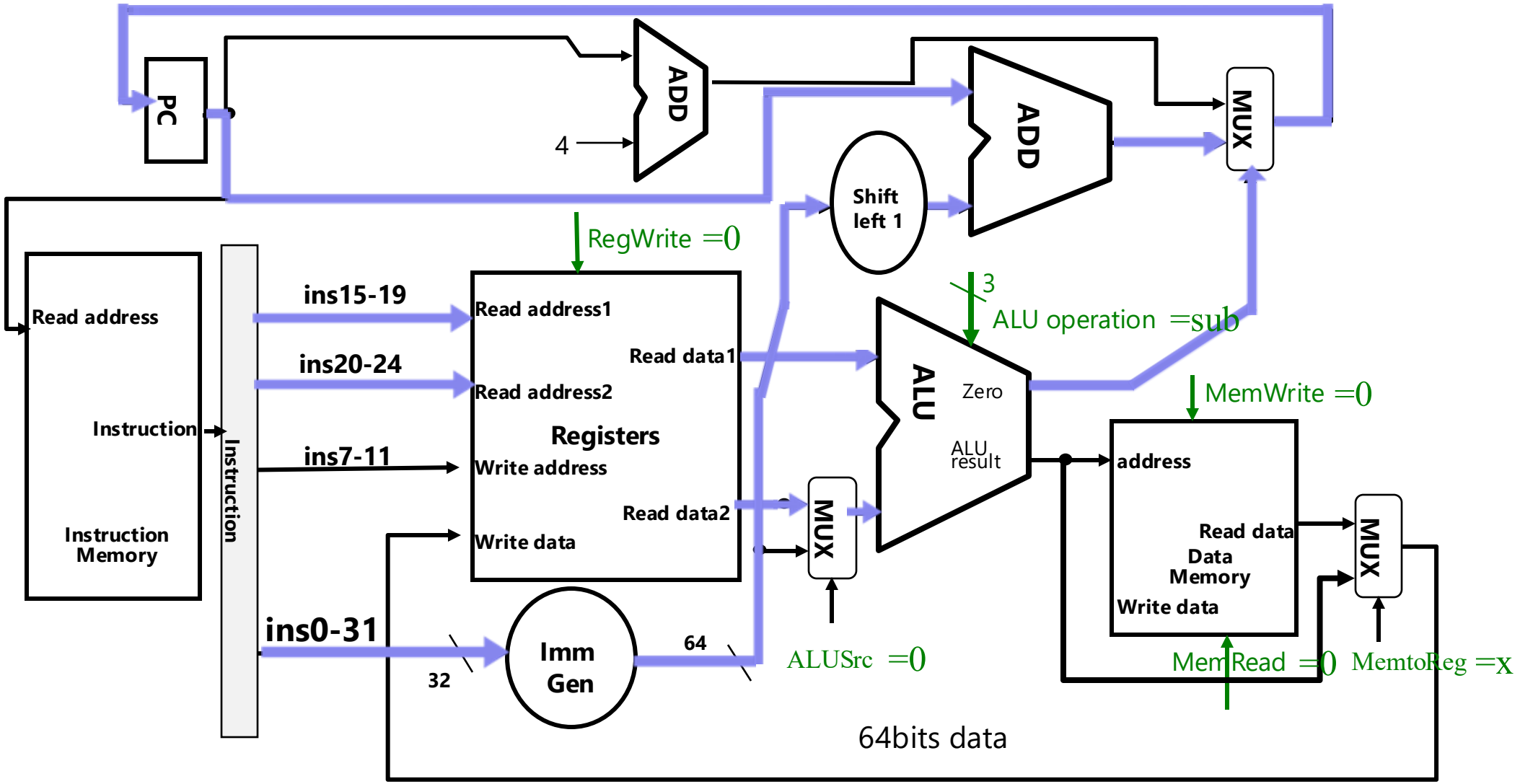
Full datapath

S-sd



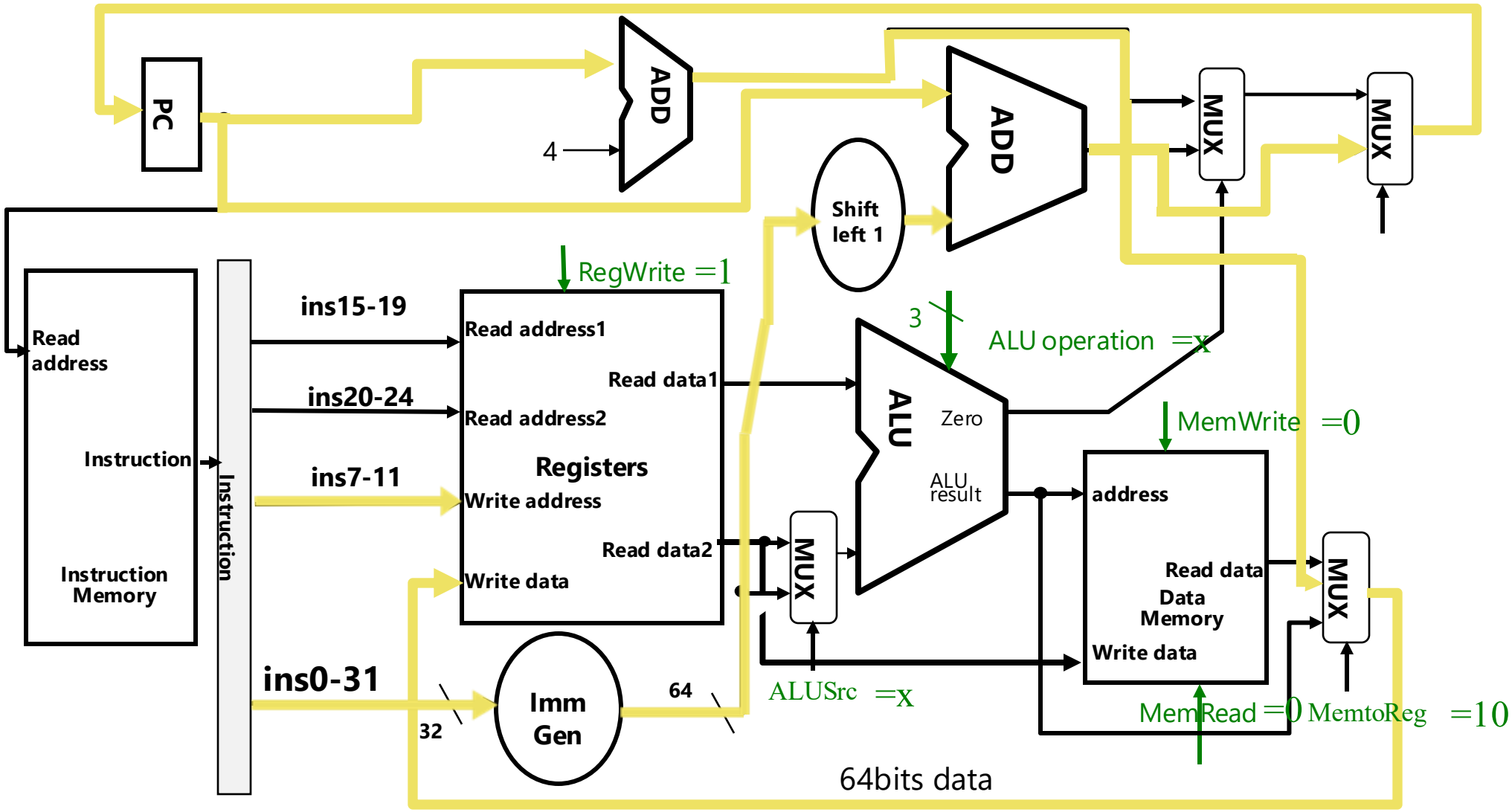
Full datapath

B-beq



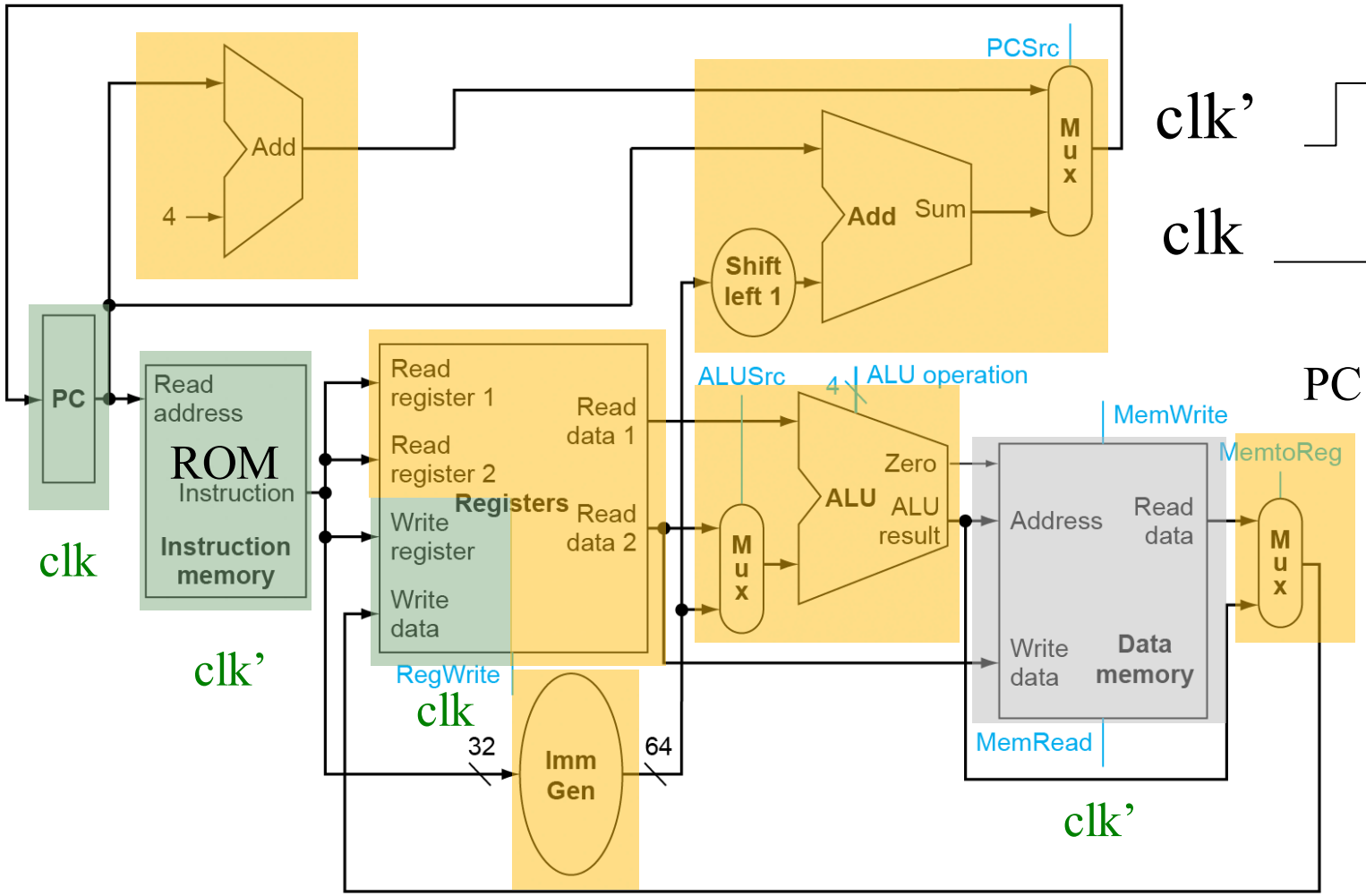
Full datapath

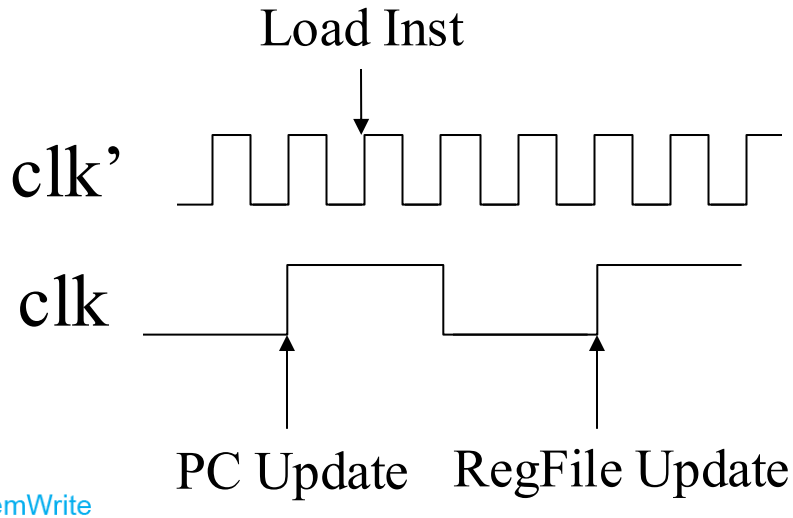
J-jal



Combinational or Sequential Logic Circuit

R Type





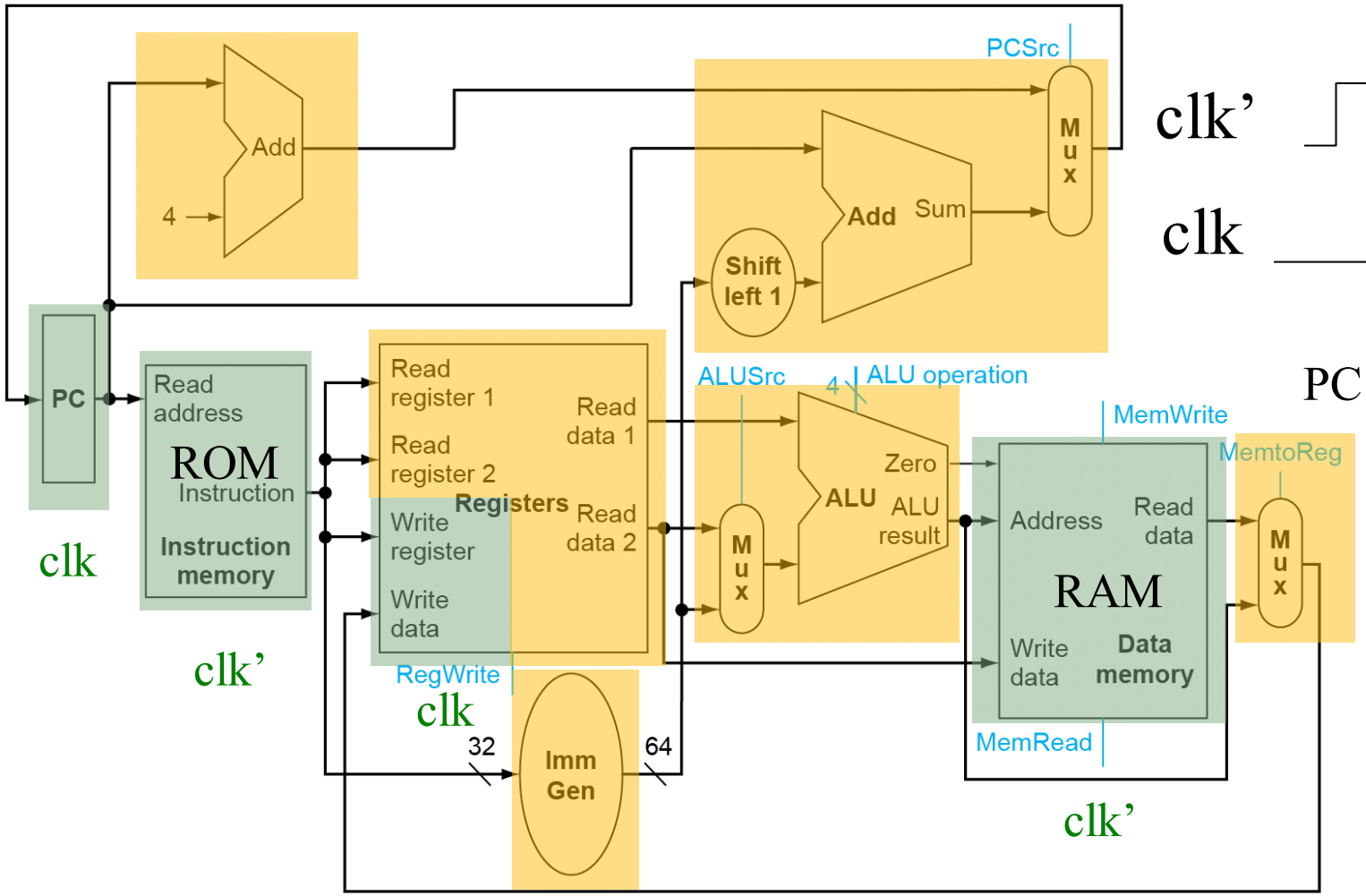
Seq

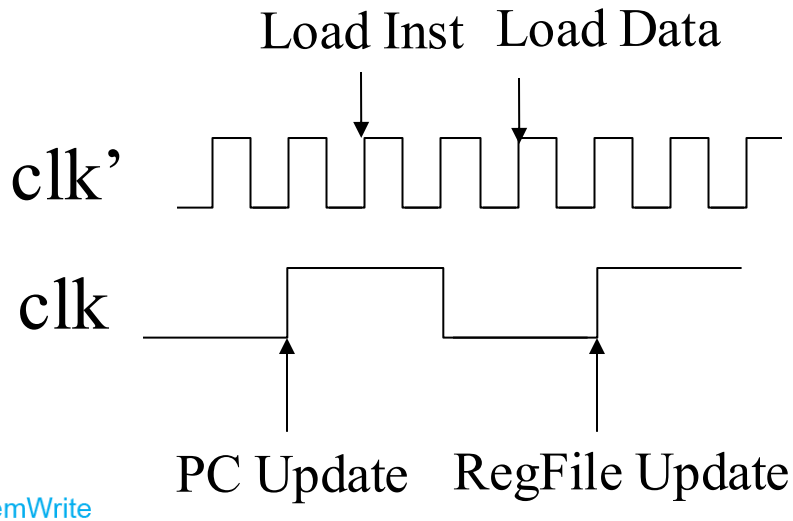
Comb

Not Used

Combinational or Sequential Logic Circuit

I-ld





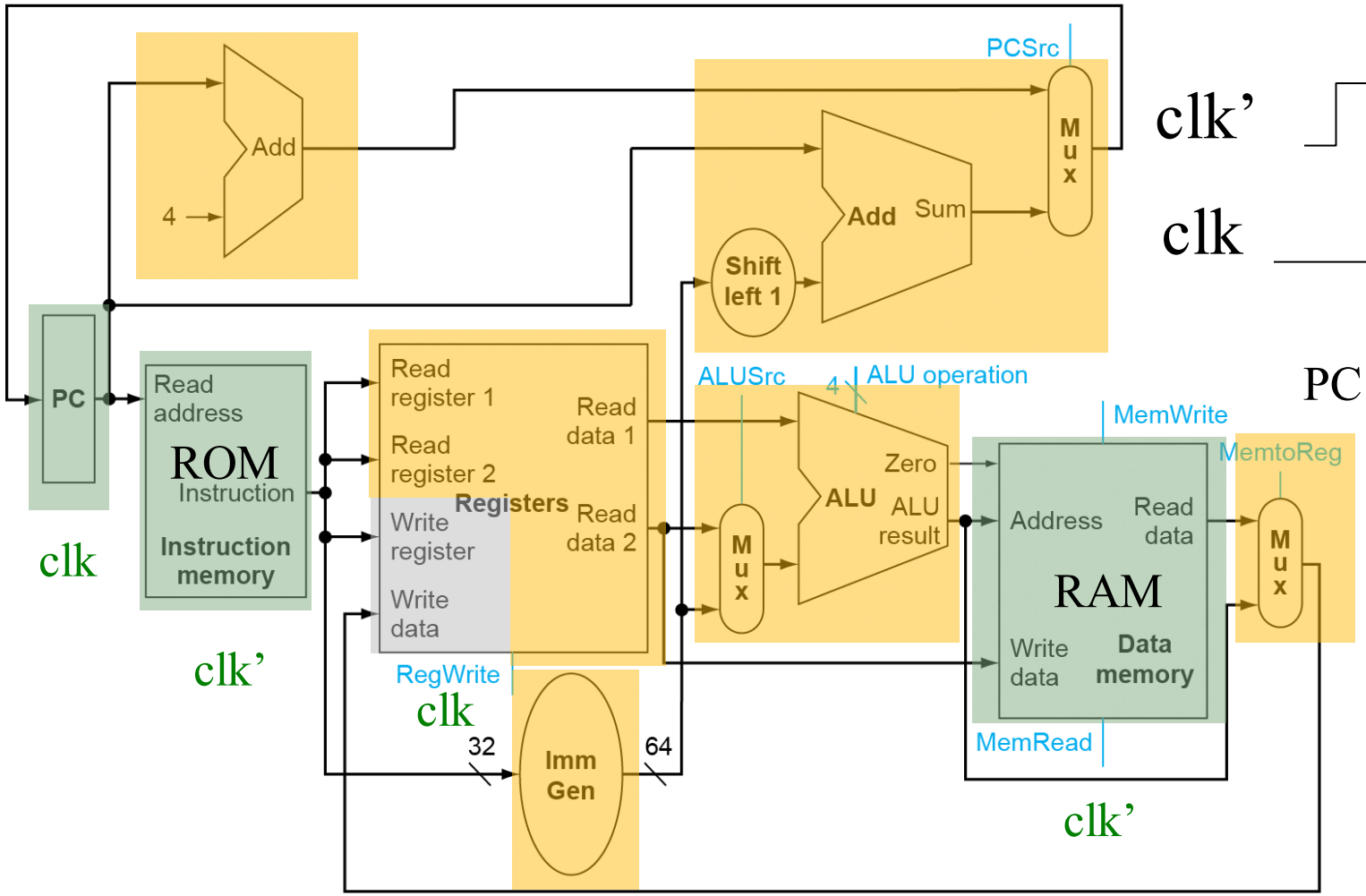
Seq

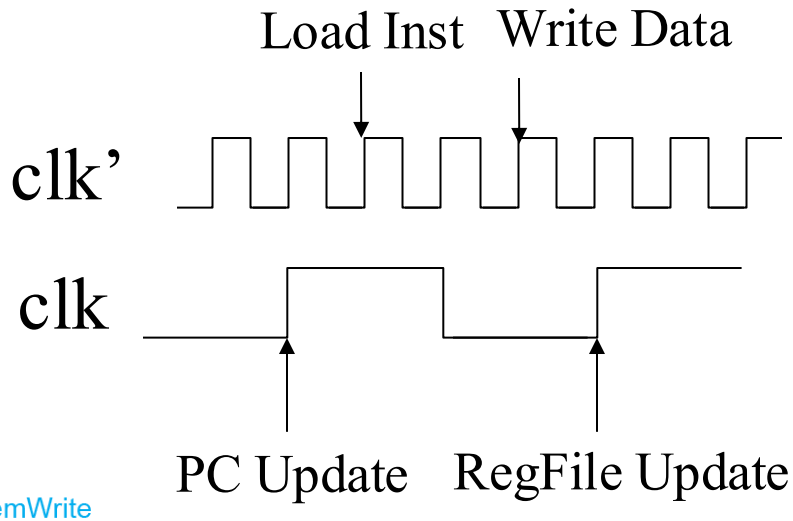
Comb

Not Used

Combinational or Sequential Logic Circuit

S-sd





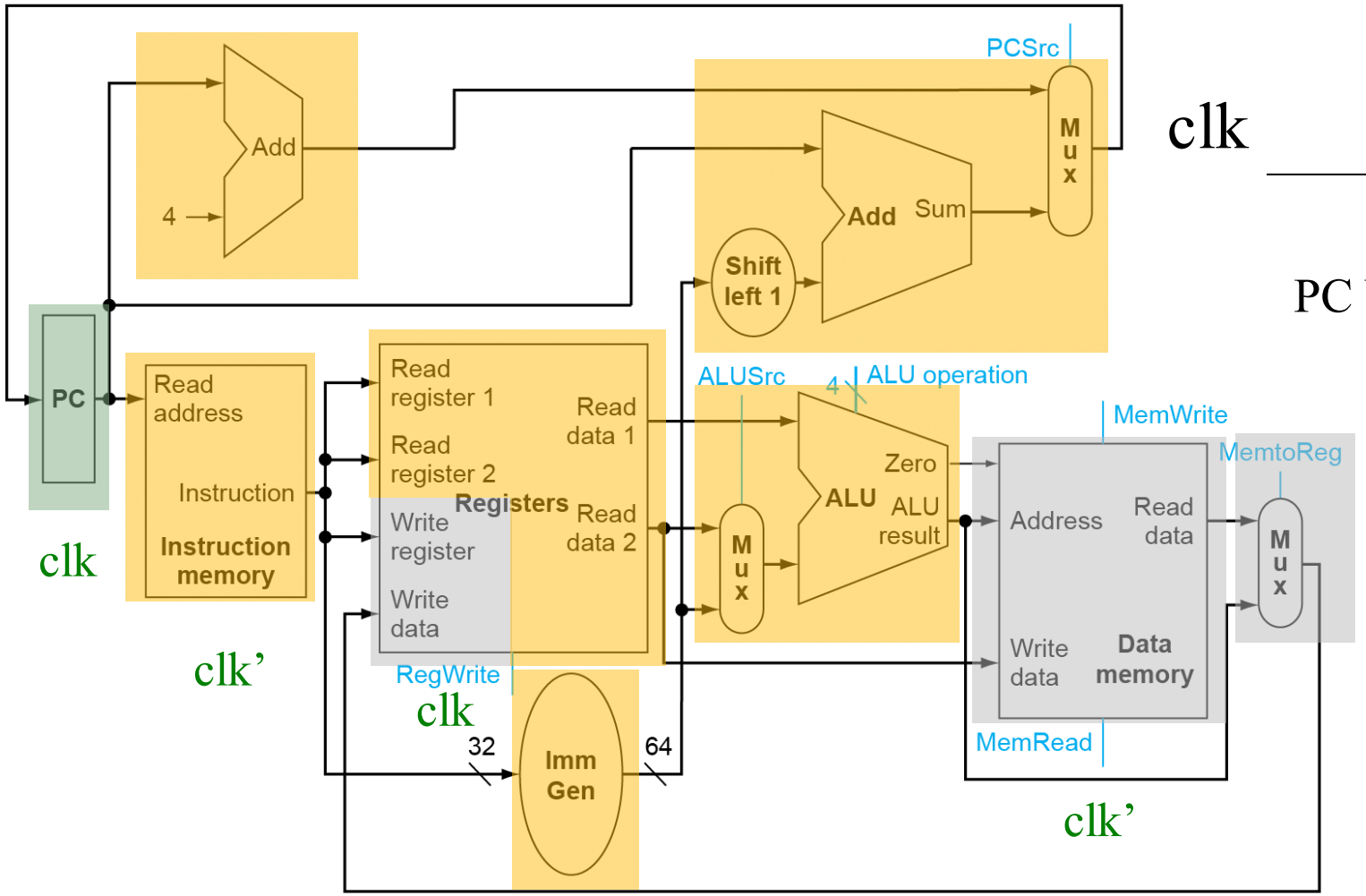
Seq

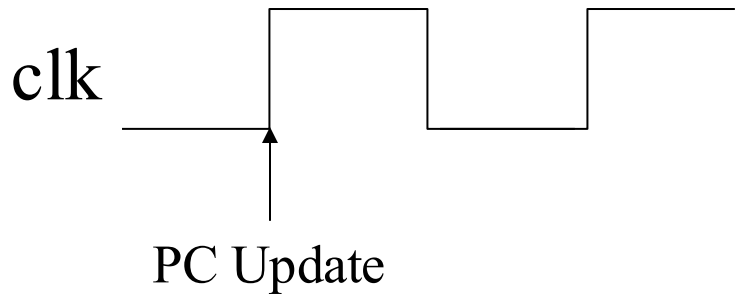
Comb

Not Used

Combinational or Sequential Logic Circuit

B-beq



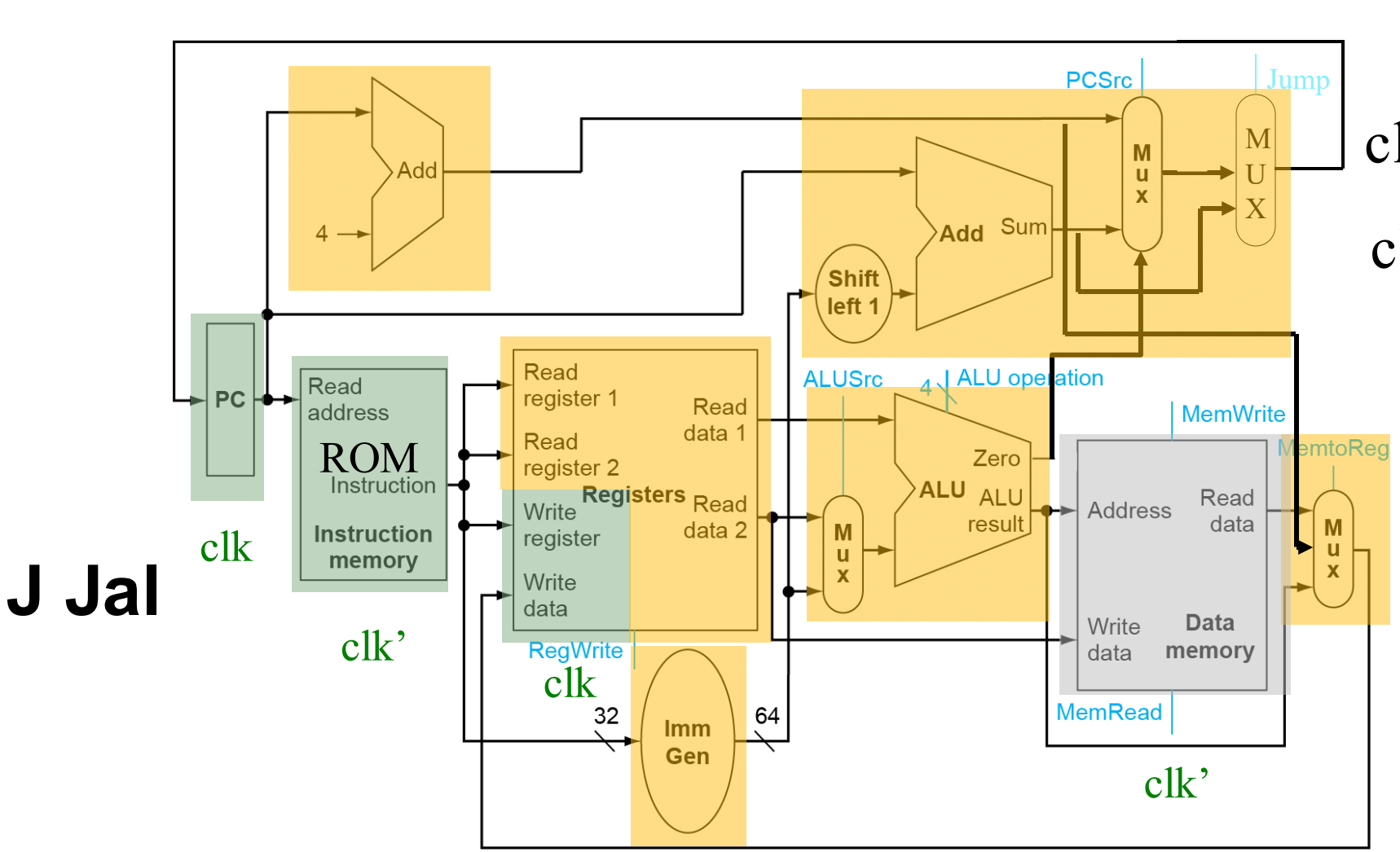


Seq

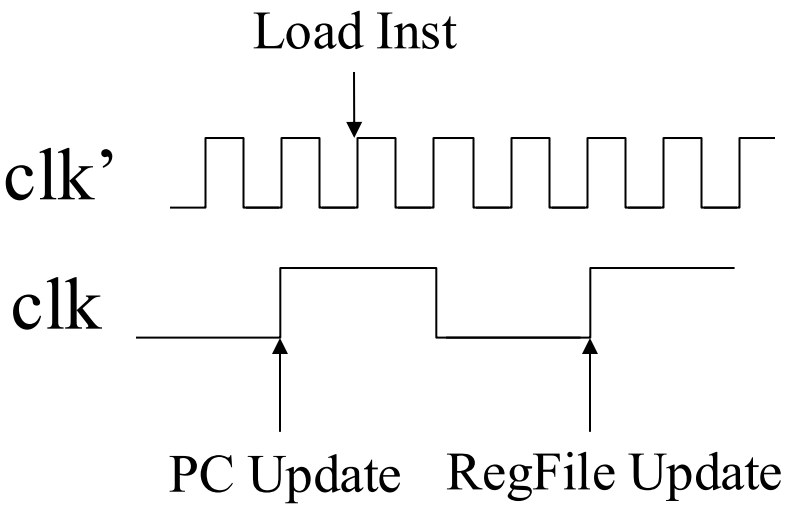
Comb

Not Used

Combinational or Sequential Logic Circuit



J Jal



Seq

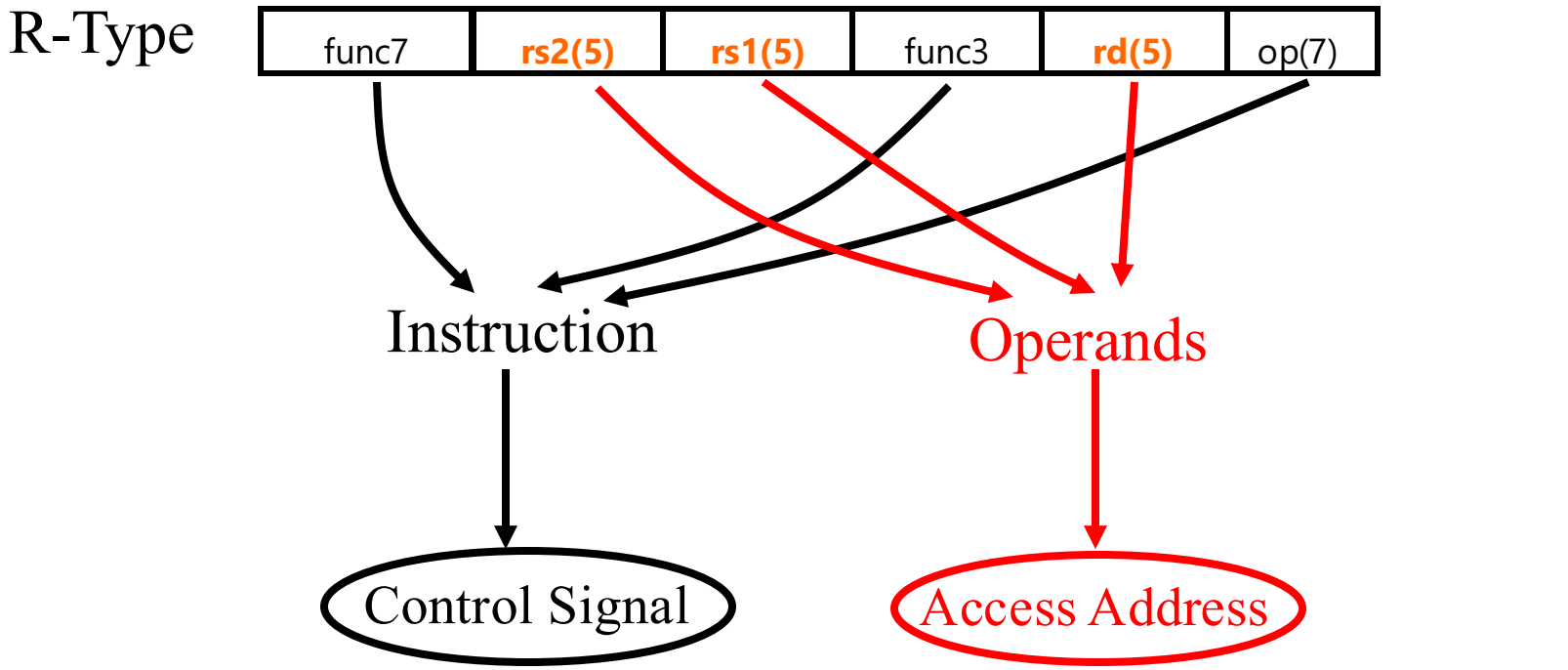
Comb

Not Used

Overview

- Instruction execution in RISC-V
- Datapath
- Controller

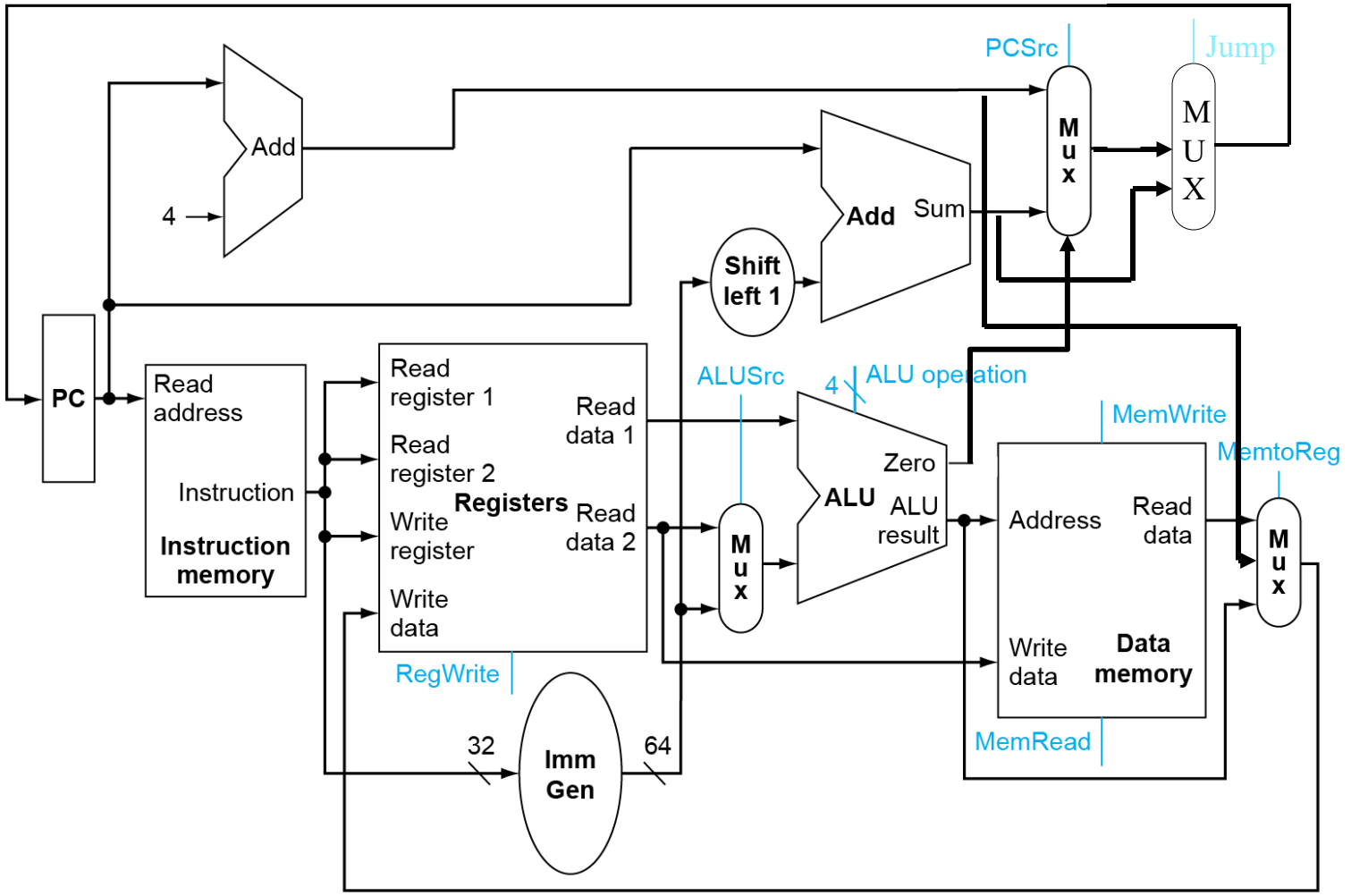
Instruction Revisit



Controller Design ！

Datapath Design

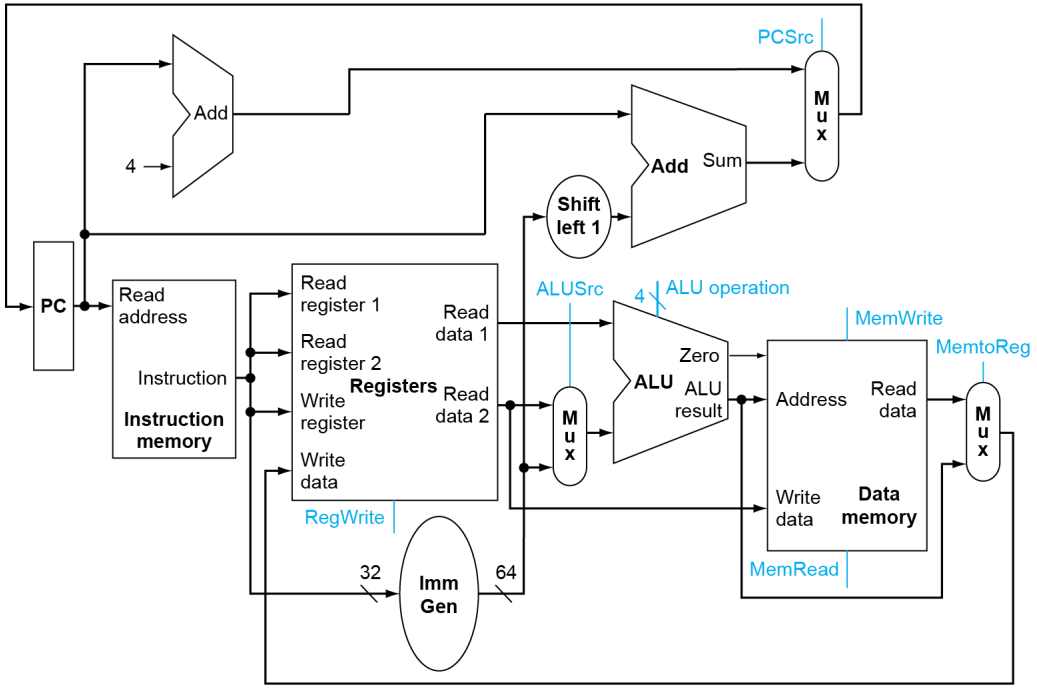
Full Datapath



Control Signals

	RegWrite	ALUSrc	PCSrc	MemRead	MemWrite	MemtoReg
0	/	The second ALU operand come from the second register file output (Read data 2)	The PC is replaced by the output of the adder that computers the value PC+4	/	/	The value fed to register Write data input comes from the ALU
1	Register destination input is written with the value on the Write data input	The second ALU operand is the sign-extended lower 16 bits of the instruction	The PC is replaced by the output of the adder that computers the branch target.	Data memory contents designated by the address input are put on the Read data output.	Data memory contents designated by the address input are replaced by value on the Write data input.	The value fed to the register Write data input comes from the data memory.

Control Signals



	RegWrite	ALUSrc	PCSrc	MemRead	MemWrite	MemtoReg
R	1	0	0	0	0	0
ld	1	1	0	1	0	1
sd	0	1	0	0	1	x
beq	0	0	1	0	0	x

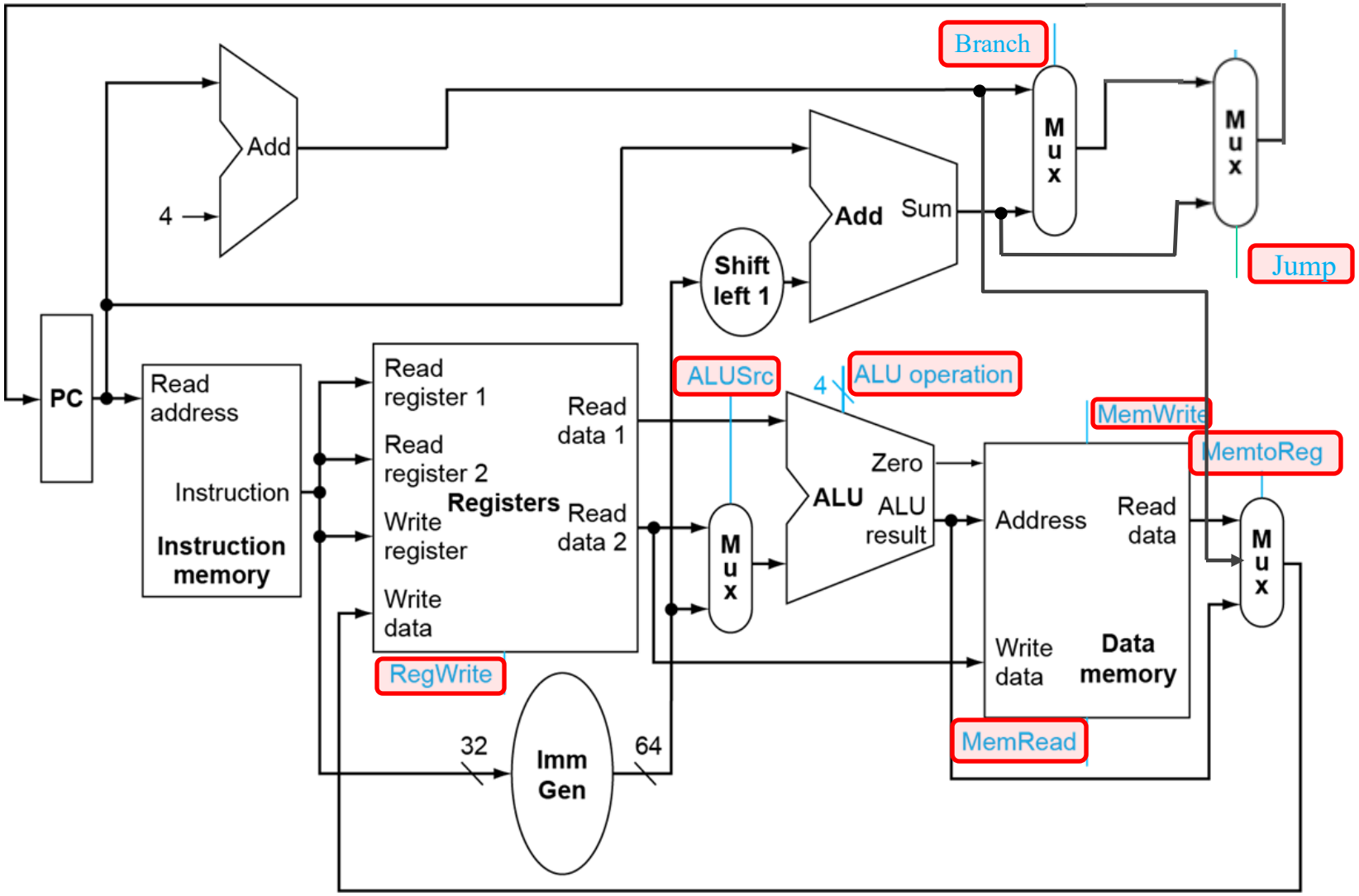
Building Controller

- Analyses for cause and effect
  - Information comes from the 32 bits of the instruction

Name (Field Size)	Field 7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	Comments
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
B-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
J-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format

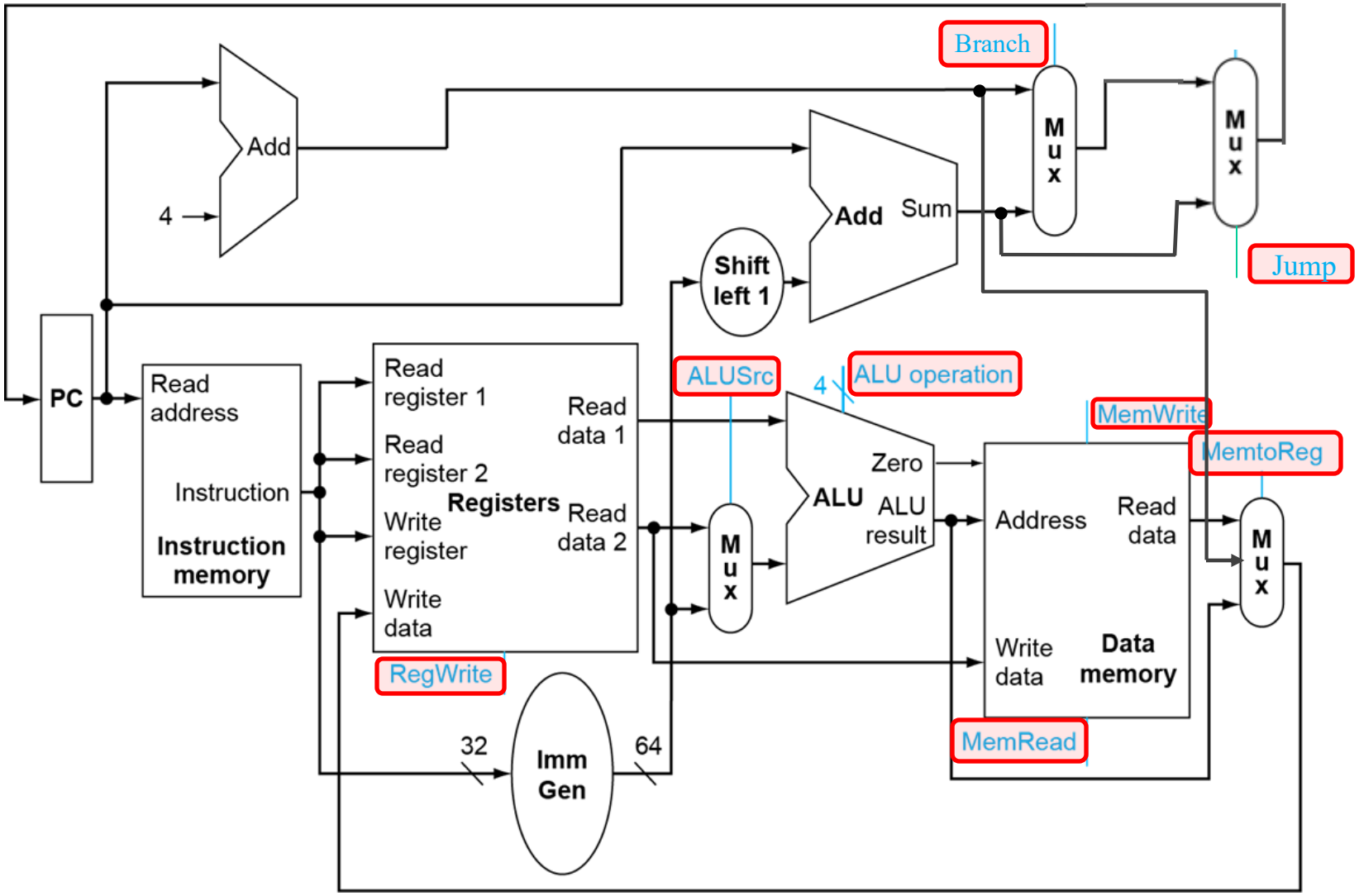
Building Controller

- Analyses for cause and effect
  - Information comes from the 32 bits of the instruction
  - Selecting the operations to perform by sequential components (read/write, etc.)



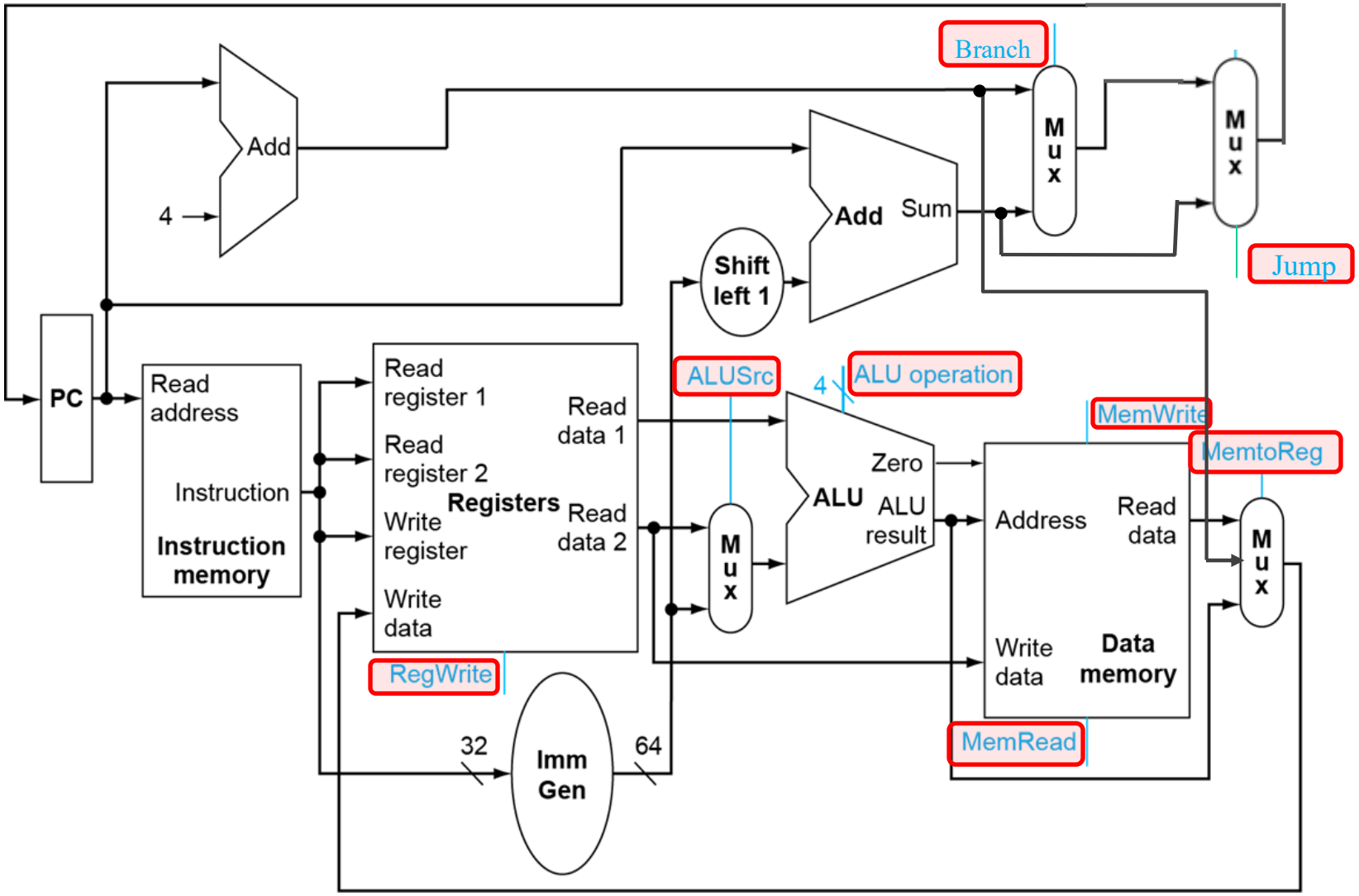
Building Controller

- Analyses for cause and effect
  - Information comes from the 32 bits of the instruction
  - Selecting the operations to perform by sequential components (read/write, etc.)
  - Controlling the flow of instruction (multiplexor inputs)



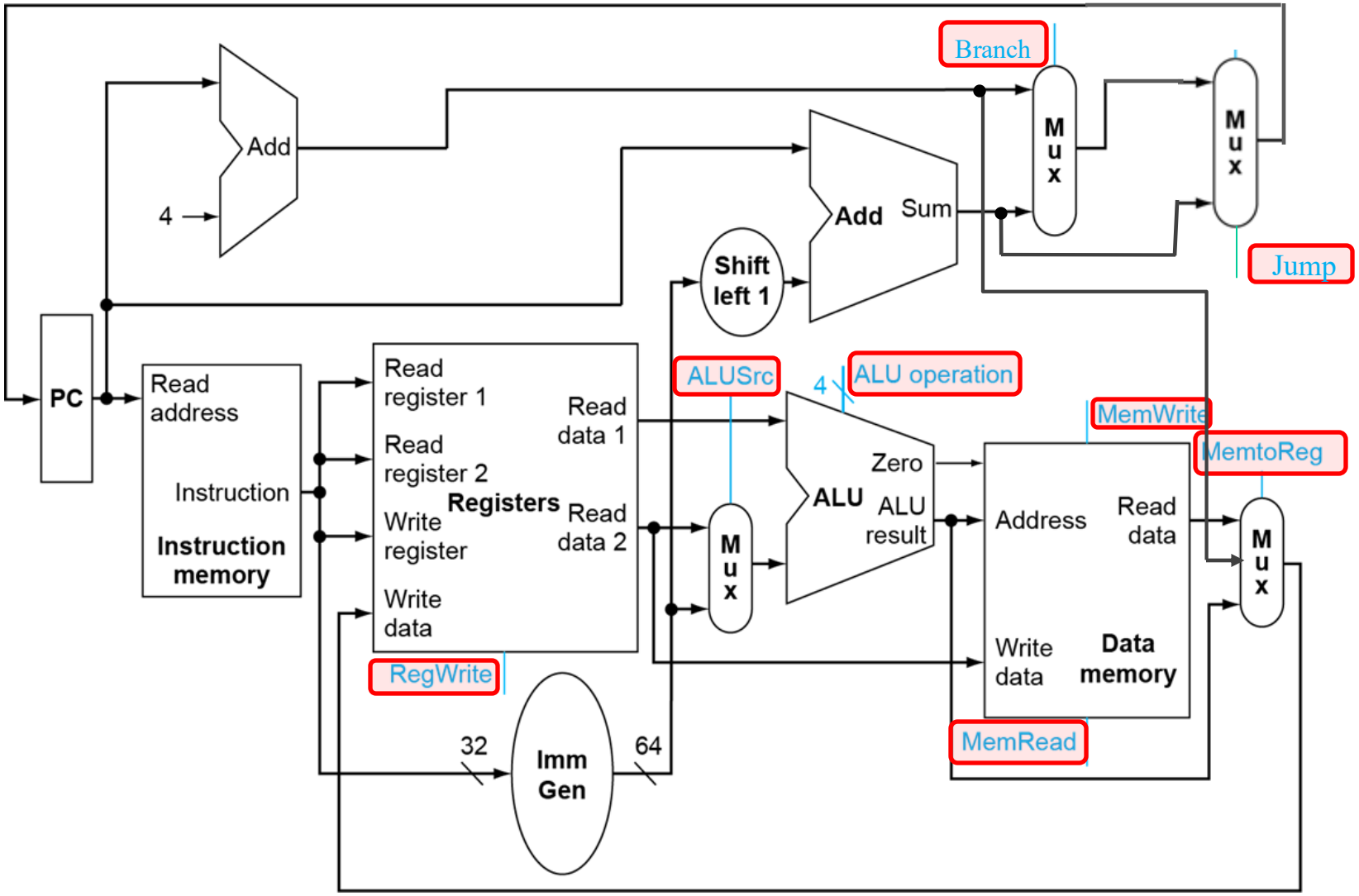
Building Controller

- Analyses for cause and effect
  - Information comes from the 32 bits of the instruction
  - Selecting the operations to perform by sequential components (read/write, etc.)
  - Controlling the flow of instruction (multiplexor inputs)
  - Controlling the flow of data (multiplexor inputs)

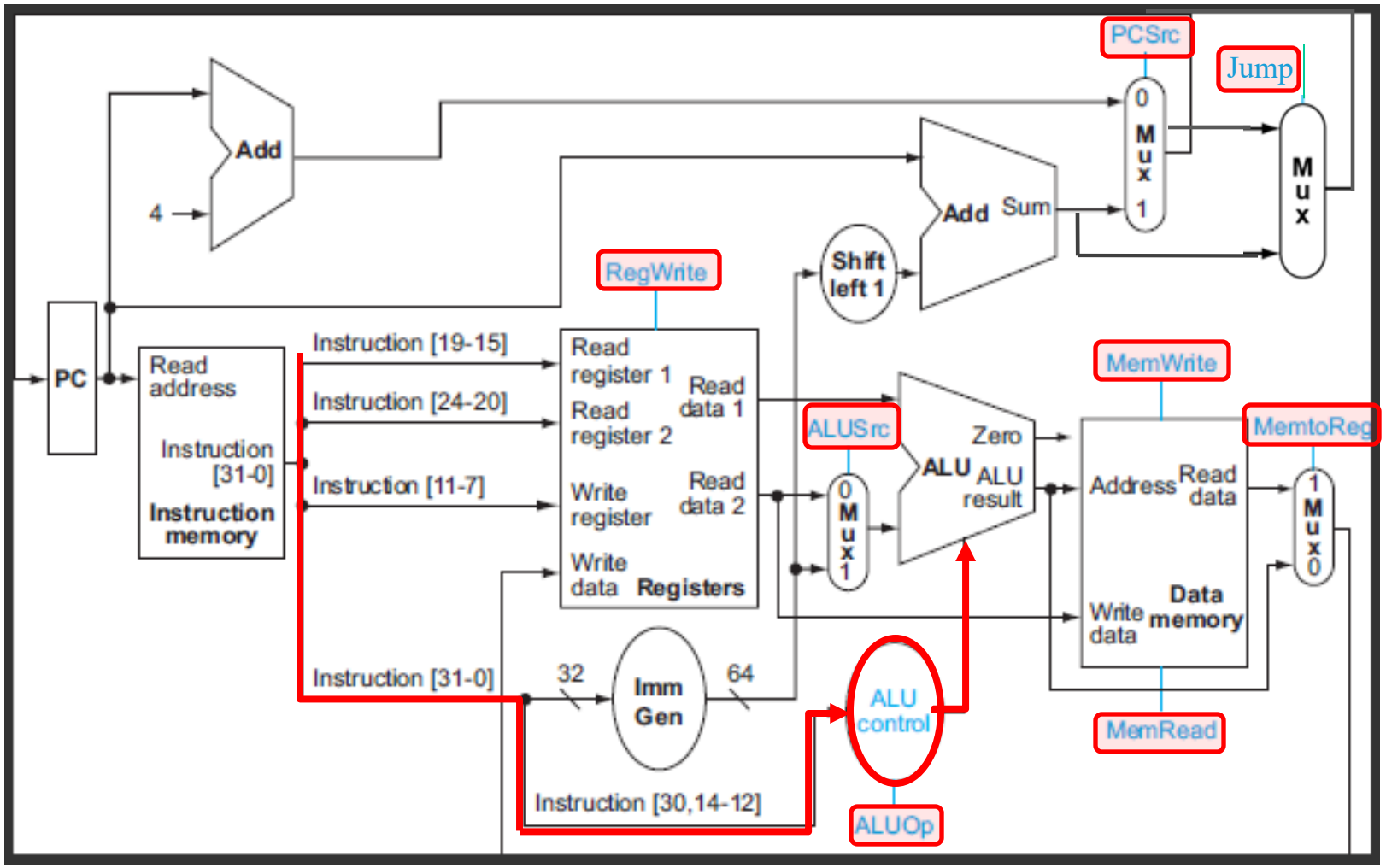


Building Controller

- Analyses for cause and effect
  - Information comes from the 32 bits of the instruction
  - Selecting the operations to perform by sequential components (read/write, etc.)
  - Controlling the flow of instruction (multiplexor inputs)
  - Controlling the flow of data (multiplexor inputs)
  - ALU's operation based on instruction type and function code

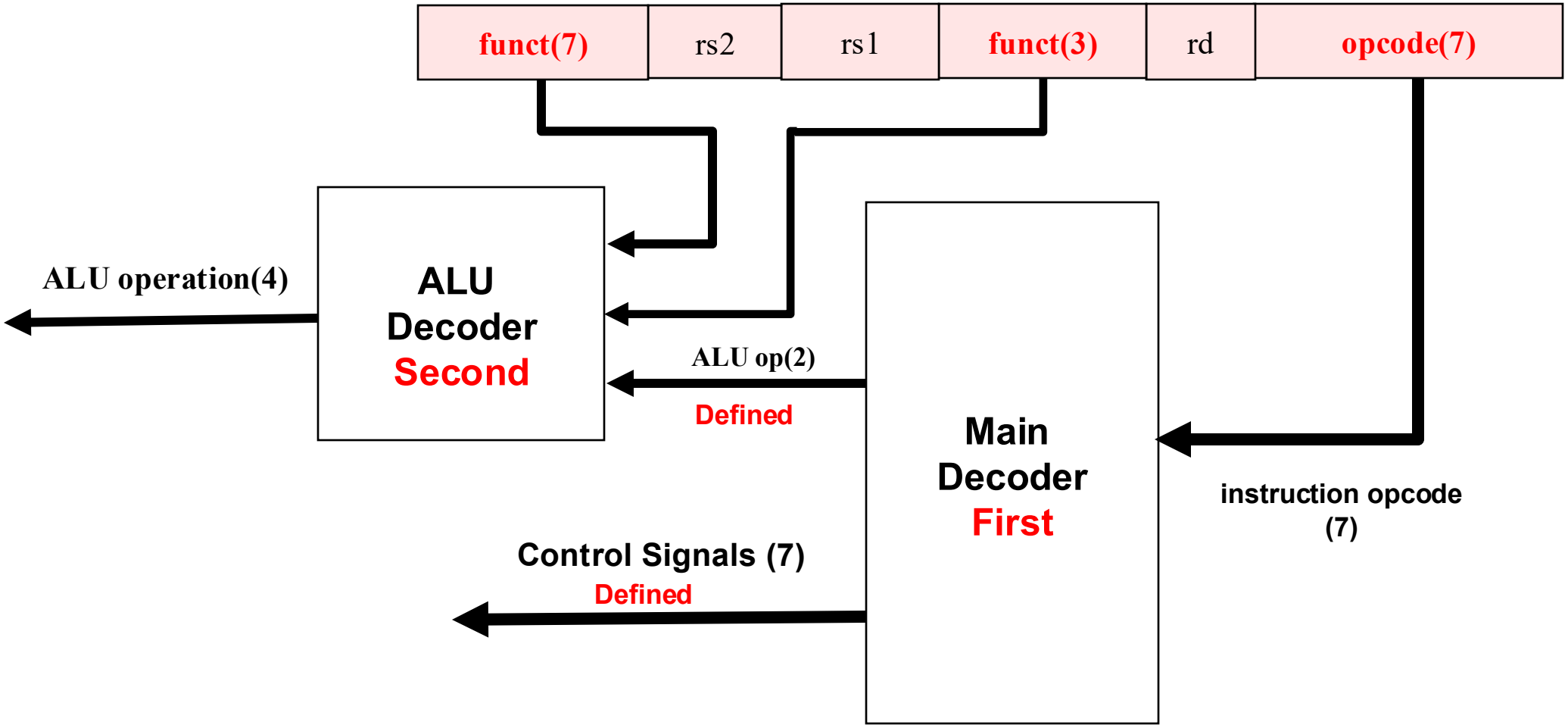


Where is ALU control and other signals



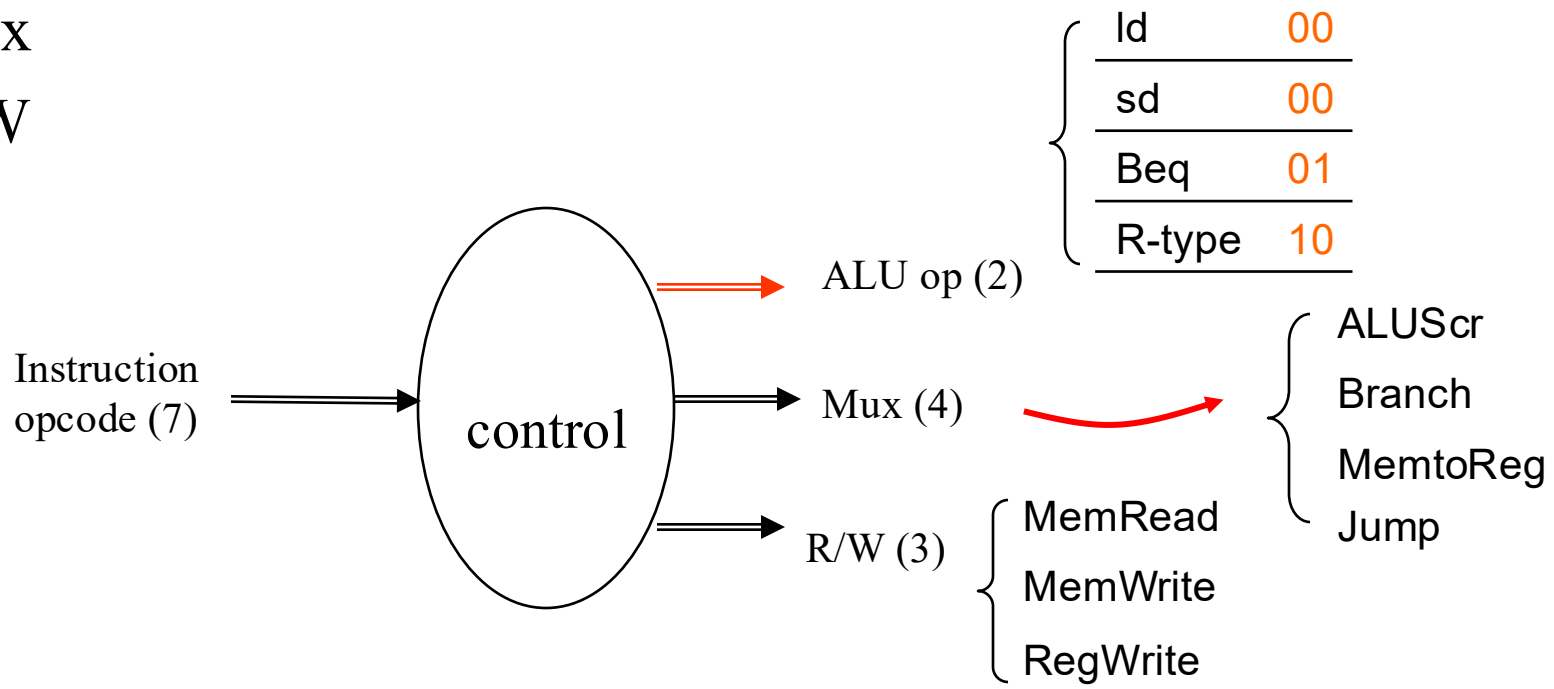
Scheme of Controller

- 2-level decoder

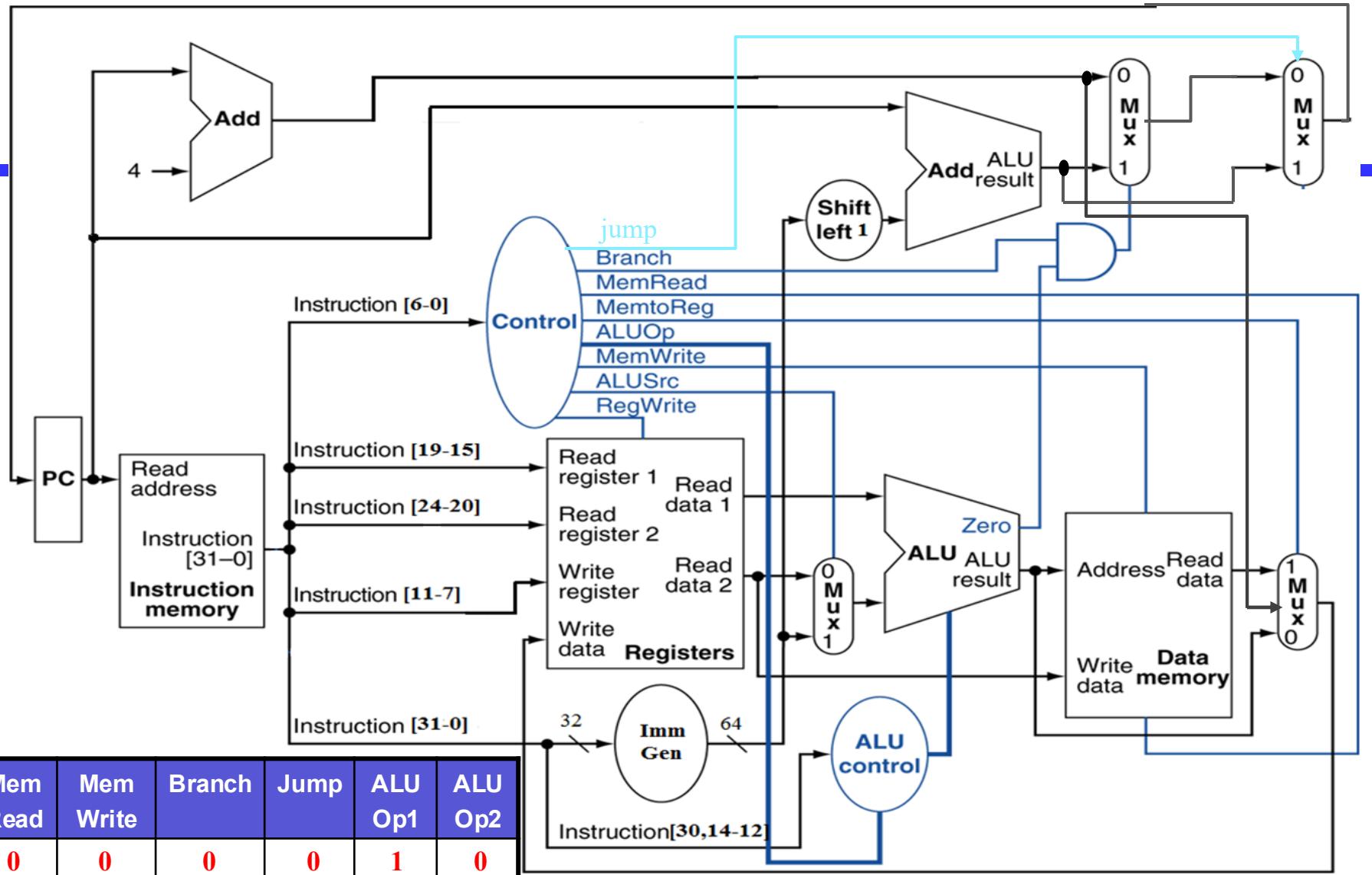


Designing the Main Control Unit – First Level

- Main Control Unit function
  - ALU op (2)
  - Divided 7 control signals into 2 groups
    - 4 Mux
    - 3 R/W



Truth Table for Main Decoder



Instruction	Opcode	ALUSrc	Memto Reg	Reg Write	Mem Read	Mem Write	Branch	Jump	ALU Op1	ALU Op2
R-format	0110011	0	00	1	0	0	0	0	1	0
Ld	0000011	1	01	1	1	0	0	0	0	0
Sd	0100011	1	xx	0	0	1	0	0	0	0
beq	1100111	0	xx	0	0	0	1	0	0	1
Jal	1101111	x	10	1	0	0	0	1	x	x

Derive Control Signals from Opcode

Input		Output								
Instruction	OPCode	ALUSrcB	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	Jump	ALU Op1	ALU Op0
R-format	0110011	0	00	1	0	0	0	0	1	0
Ld(I-Type)	0000011	1	01	1	1	0	0	0	0	0
sd(S-Type)	0100011	1	x	0	0	1	0	0	0	0
beq(B-Type)	1100111	0	x	0	0	0	1	0	0	1
Jal(J-Type)	1101111	x	10	1	0	0	0	1	x	x

Example: MemRead=~Op[6] & ~Op[5] & ~Op[4] & ~Op[3] & ~Op[2] & Op[1] & Op[0]

What about RegWrite?

Designing the ALU Decoder – Second Level

- ALU operation is decided by 2-bit ALUOp derived from opcode, and funct7 & funct3 fields of the instruction
  - Combinational logic derives ALU control

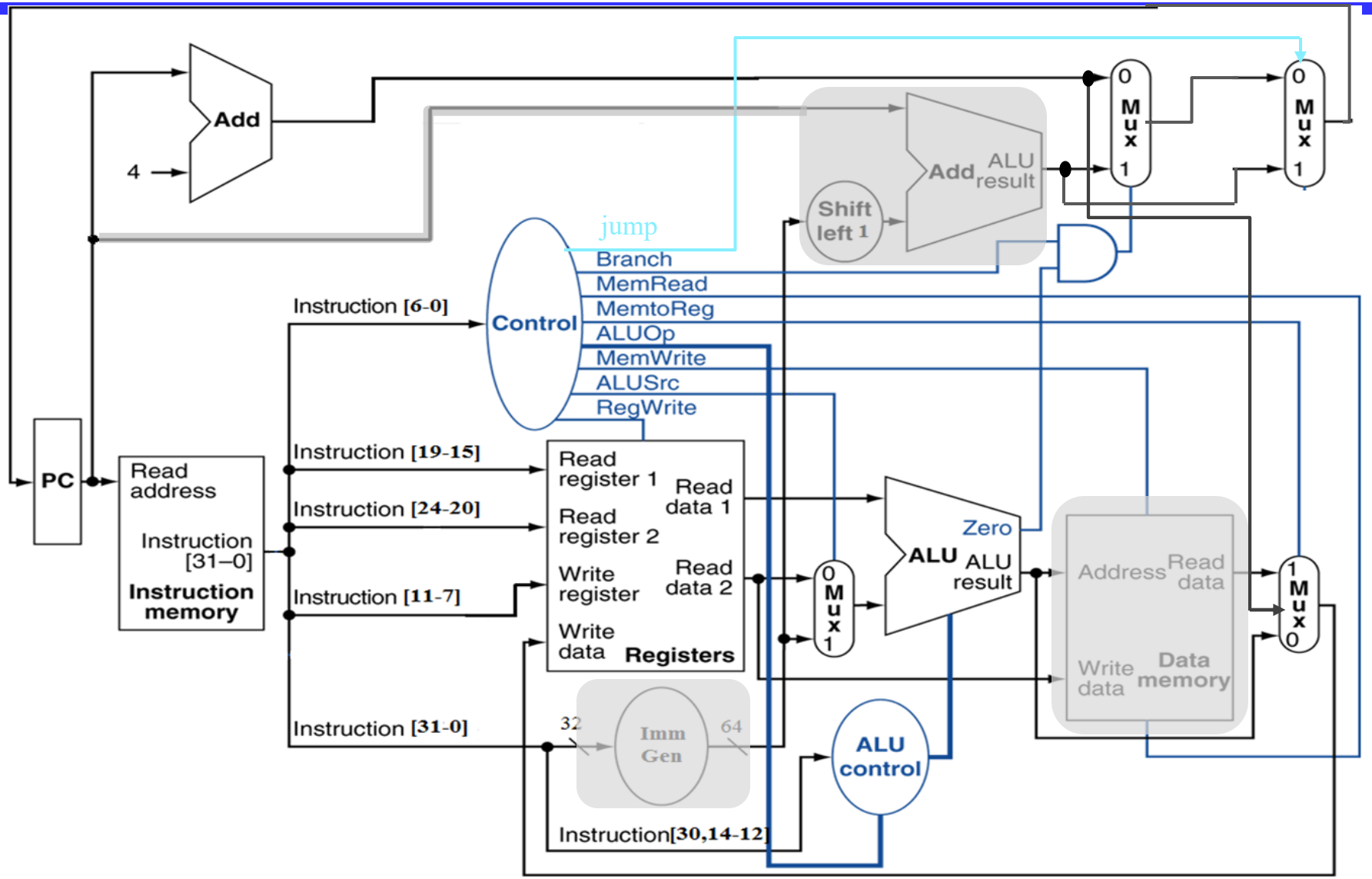
opcode	ALUOp	Operation	Funct7	funct3	ALU function	ALU control
ld	00	load register	XXXXXXX	xxx	add	0010
sd	00	store register	XXXXXXX	xxx	add	0010
beq	01	branch on equal	XXXXXXX	xxx	subtract	0110
R-type	10	add	0000000	000	add	0010
		subtract	0100000	000	subtract	0110
		AND	0000000	111	AND	0000
		OR	0000000	110	OR	0001

R Instruction

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

add x9, x20, x21

1. Read two register operands
2. Perform arithmetic/logical operation
3. Write register result

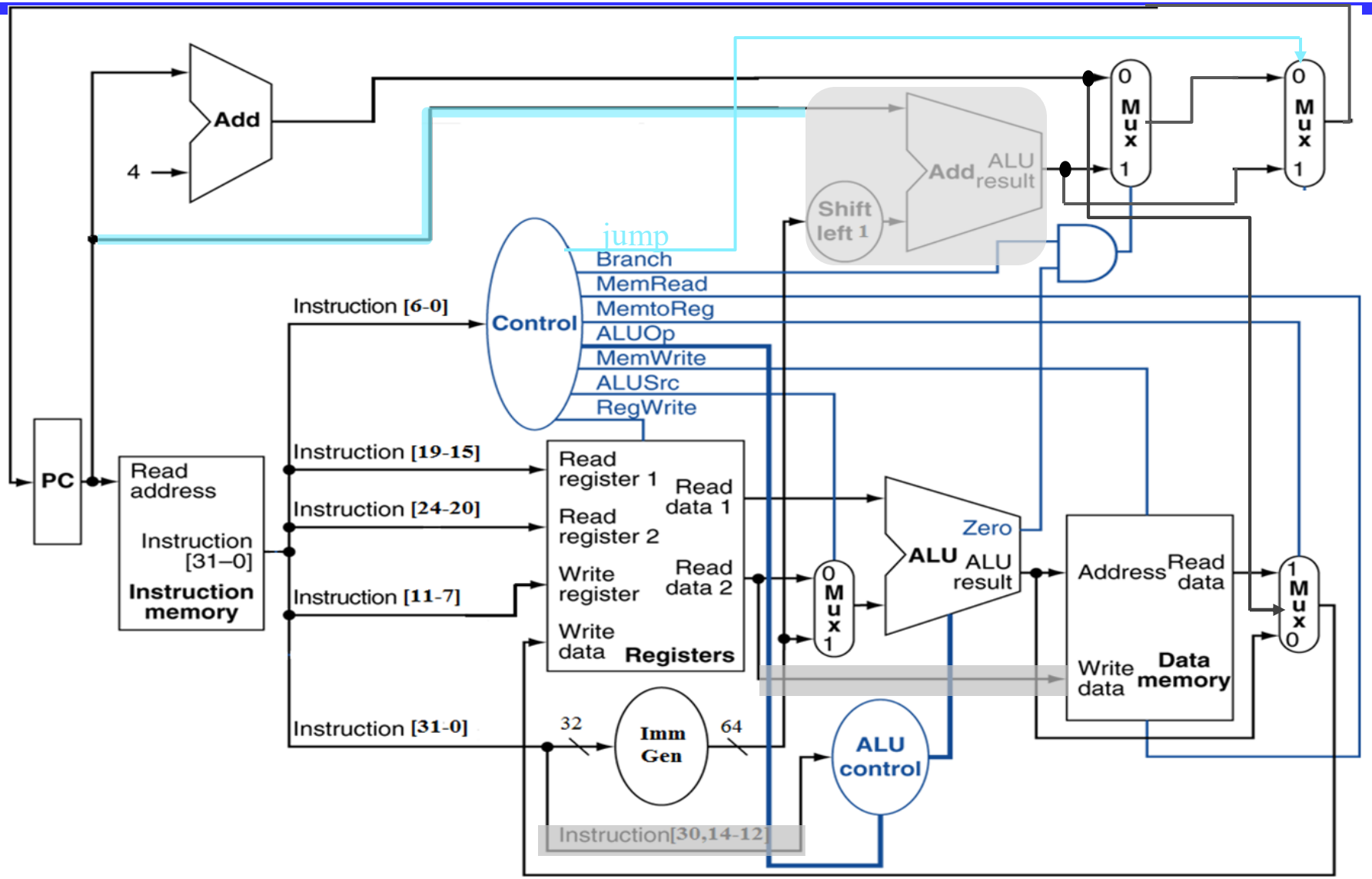


Load Instruction

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

lw x1, 200(x2)

1. **Read register operands**
2. **Calculate address using 12-bit offset**
   - Use ALU, but sign-extend offset
3. **Read memory and update register**

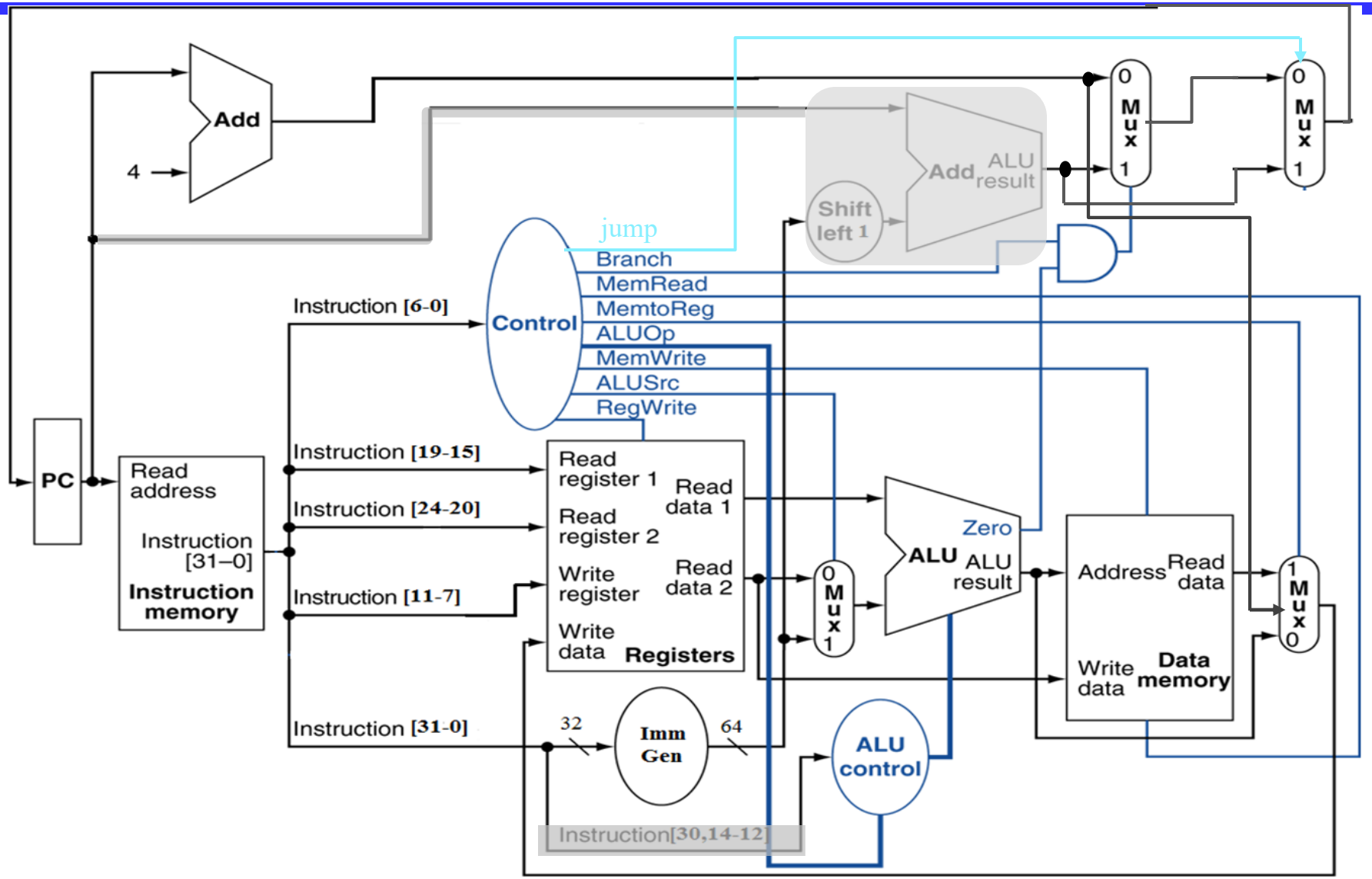


Store Instruction

immediate[11:5]	rs2	rs1	funct3	immediate[4:0]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

sw x1, 200(x2)

1. **Read register operands**
2. **Calculate address using 12-bit offset**
   - Use ALU, but sign-extend offset
3. **Write register value to memory**

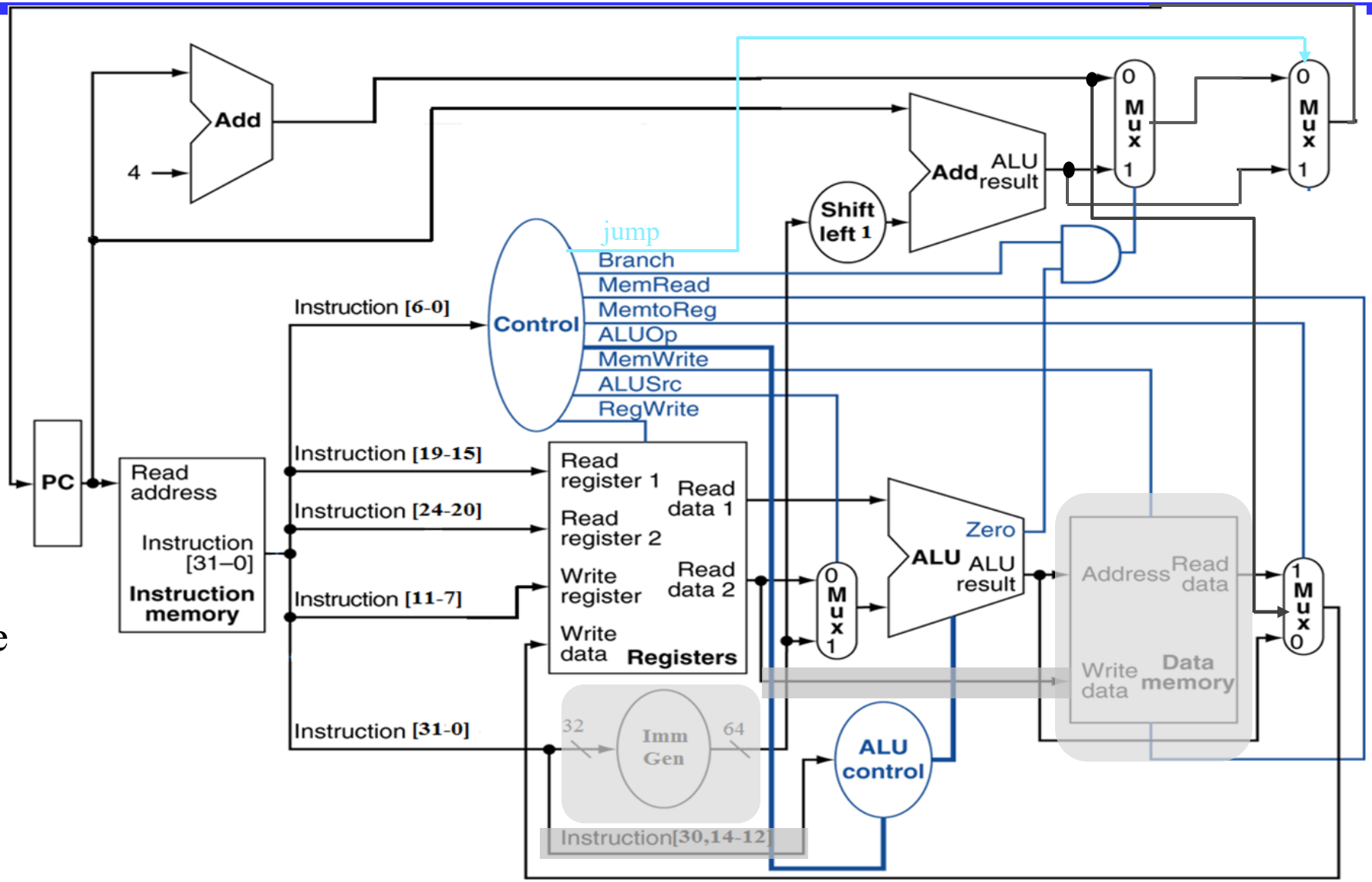


Beq Instruction

imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode

beq x1, x2，200

- **Read register operands**
- **Compare operands**
  - Use ALU, subtract and check Zero output
- **Calculate target address**
  - Sign-extend displacement
  - Shift left 1 place (halfword displacement)
  - Add to PC value and update PC

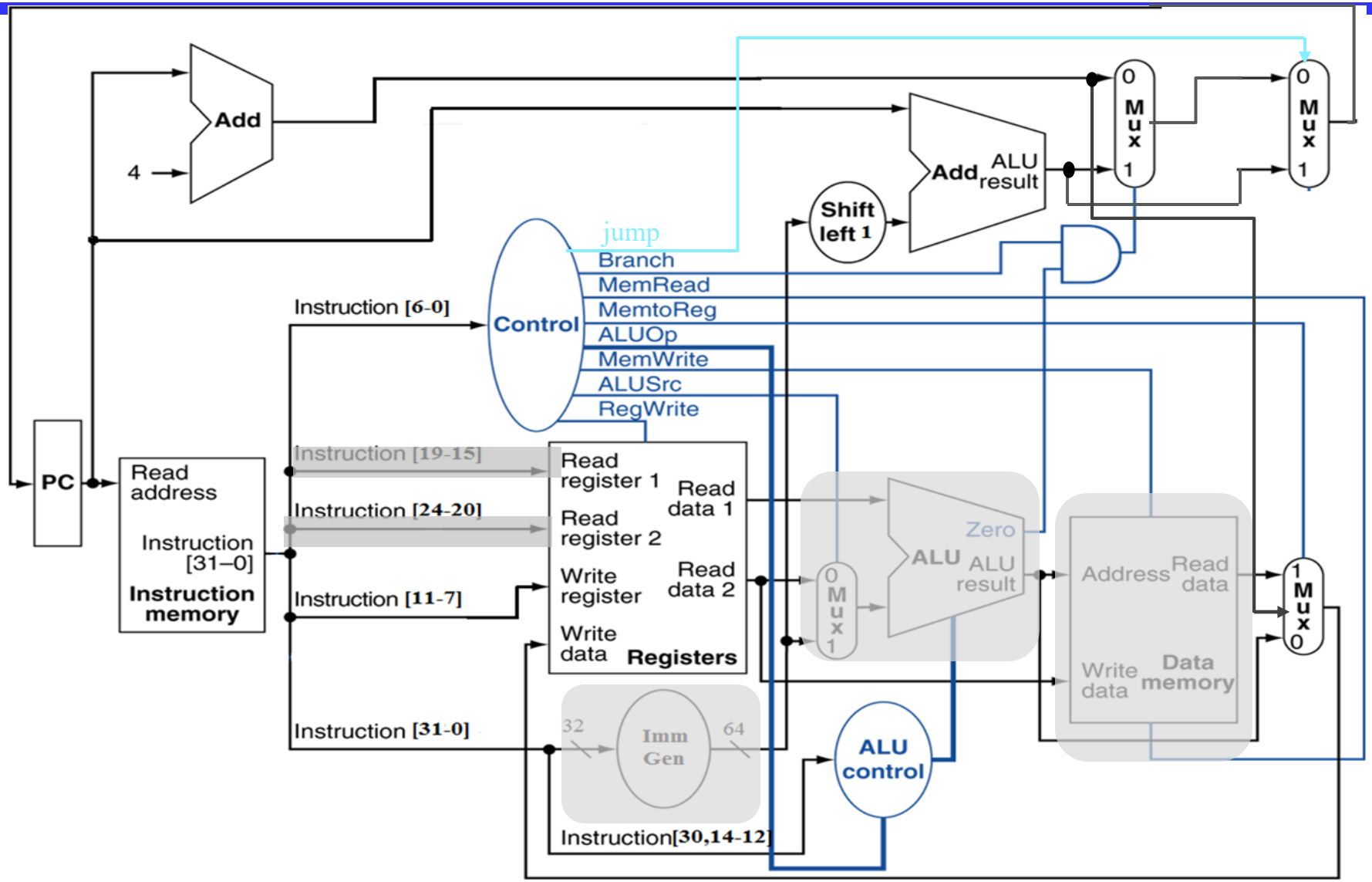


Jal Instruction

imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode

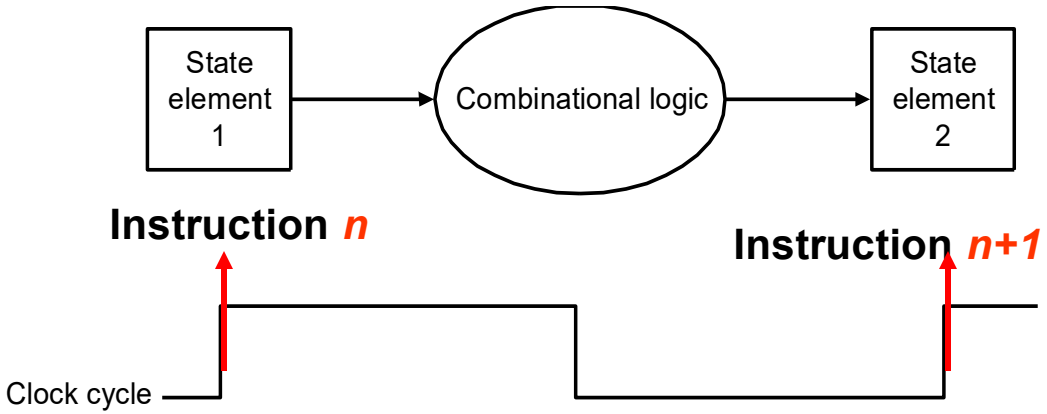
jal x1, procedure

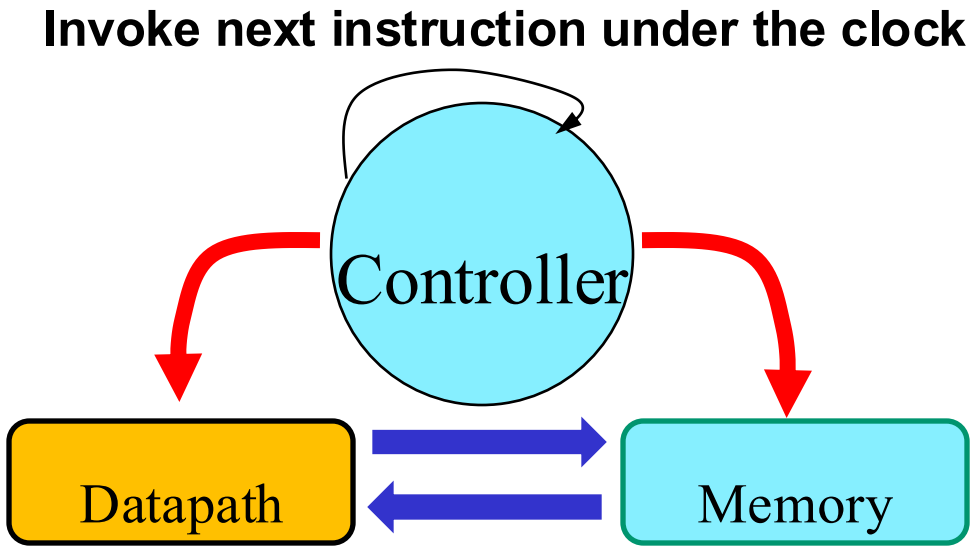
- **Write PC+4 to rd**
- **Calculate target address**
  - Sign-extend displacement
  - Shift left 1 place (halfword displacement)
  - Add to PC value and update PC



Control Structure

- All of the logic is combinational
- We wait for everything to settle down, and the right thing to be done
  - ALU might not produce right answer? right away theoretically
  - We use write signals along with clock to determine when to write
- Cycle time determined by length of the longest path

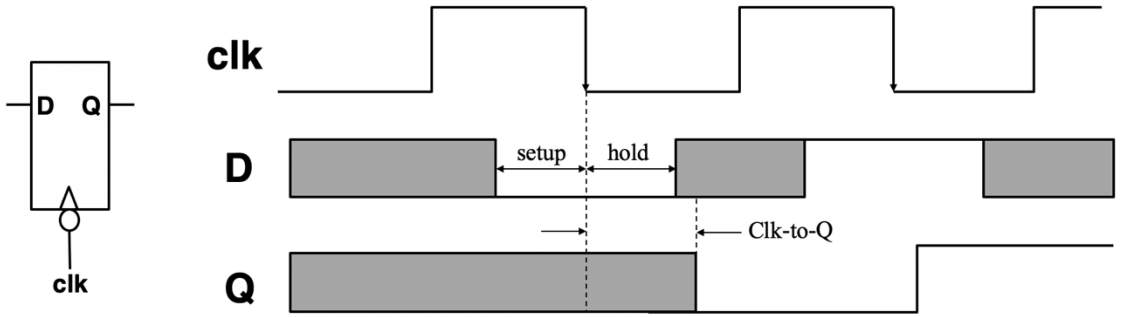




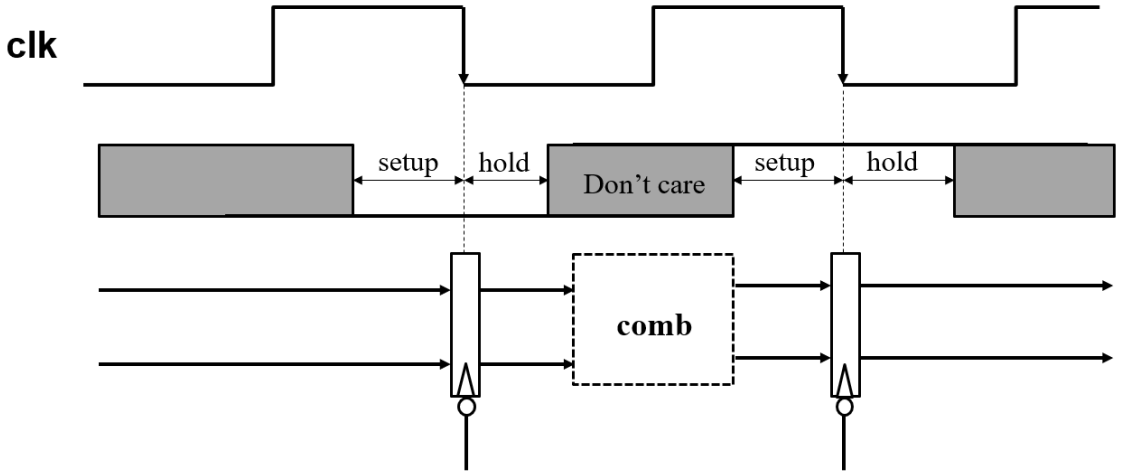
We are ignoring some details like setup and hold times

Time Flow – R type

- Combinational vs. Sequential
  - Revisit the time delay in register writing

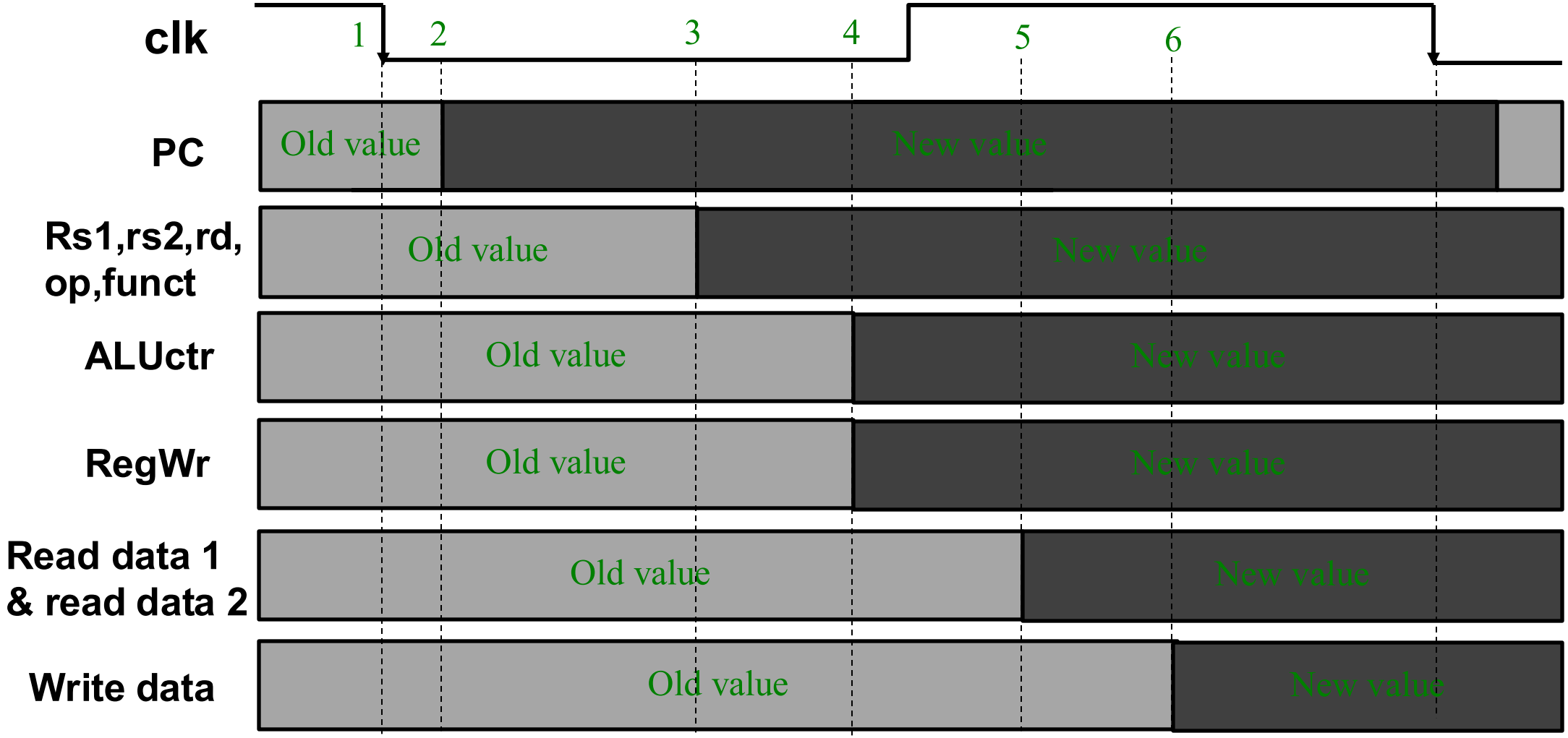


  - Revisit the time flow in datapath

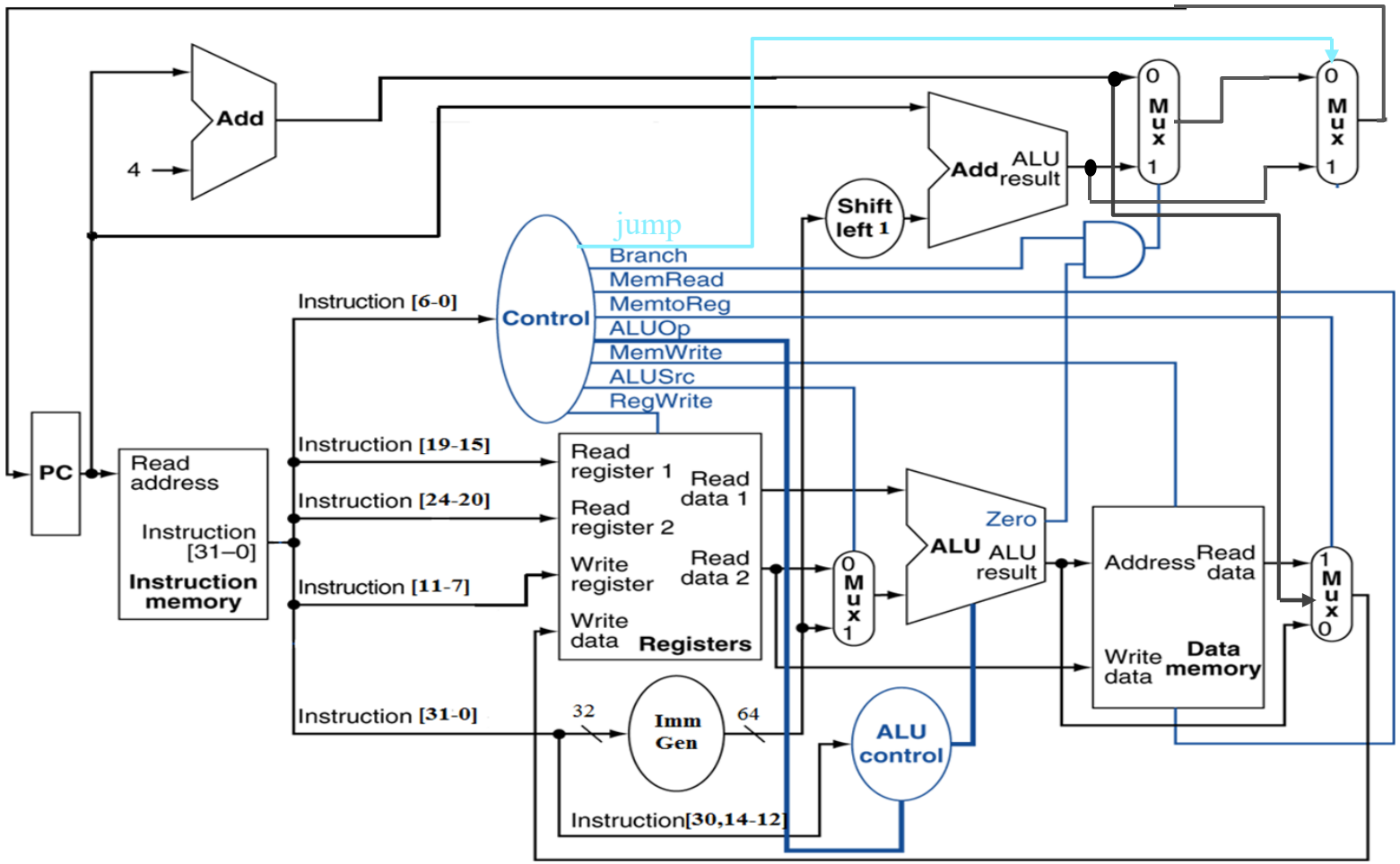


Time Flow – R type

- Combinational vs. Sequential



Single Cycle Implementation – Performance for lw



200ps 100+100=200ps 200ps 200ps

- Calculate cycle time assuming negligible delays except:
  - memory (200ps), ALU and adders (200ps), register file access (100ps)