计算机组成与设计

Computer Organization & Design

The Hardware/Software Interface

Chapter 4

The processor-Part1

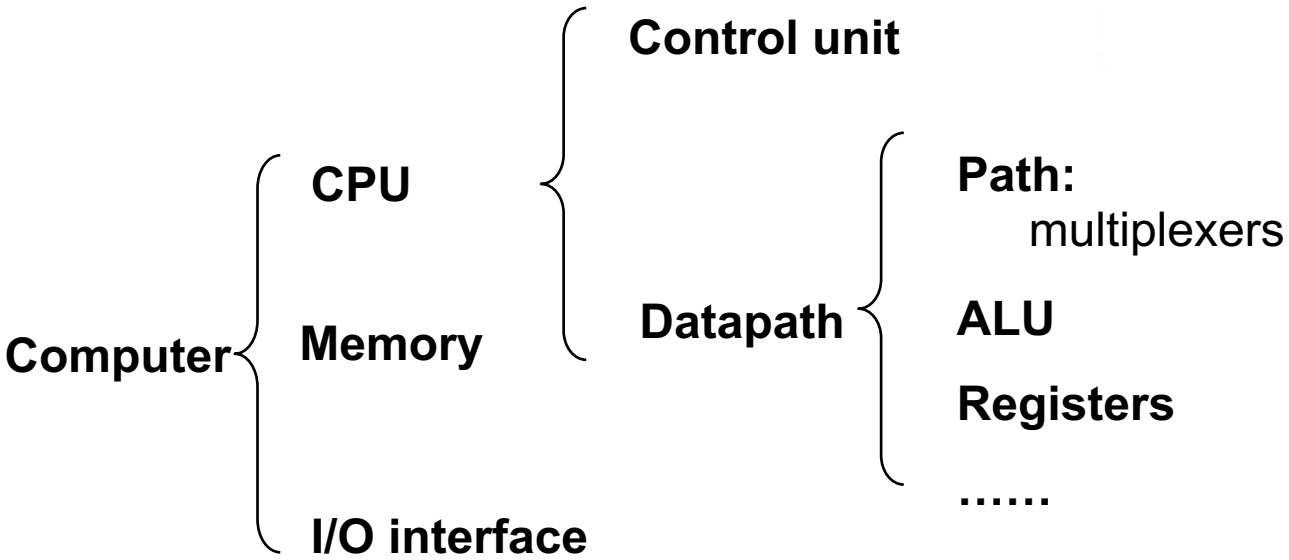
马德

College of Computer Science and Technology

Zhejiang University

made@zju.edu.cn

Computer Organization



Outline

- Introduction
- Logic Design Conventions
- Building a Datapath
- A Simple Implementation Scheme

4.1 Introduction

- **CPU performance factors**
  - Instruction count
    - Determined by ISA and compiler
  - CPI and Cycle time
    - Determined by CPU hardware
- **We will examine two RISC-V implementations**
  - A simplified version
  - A more realistic pipelined version
- **Simple subset, shows most aspects**
  - Memory reference: ld, sd
  - Arithmetic/logical: add, sub, and, or
  - Control transfer: beq

实验要求：

R-Type：add, sub, and, or, xor, slt,srl；

I-Type：addi, andi, ori, xori, slti, srli,**lw**;

S-Type：sw；

SB-Type：beq；

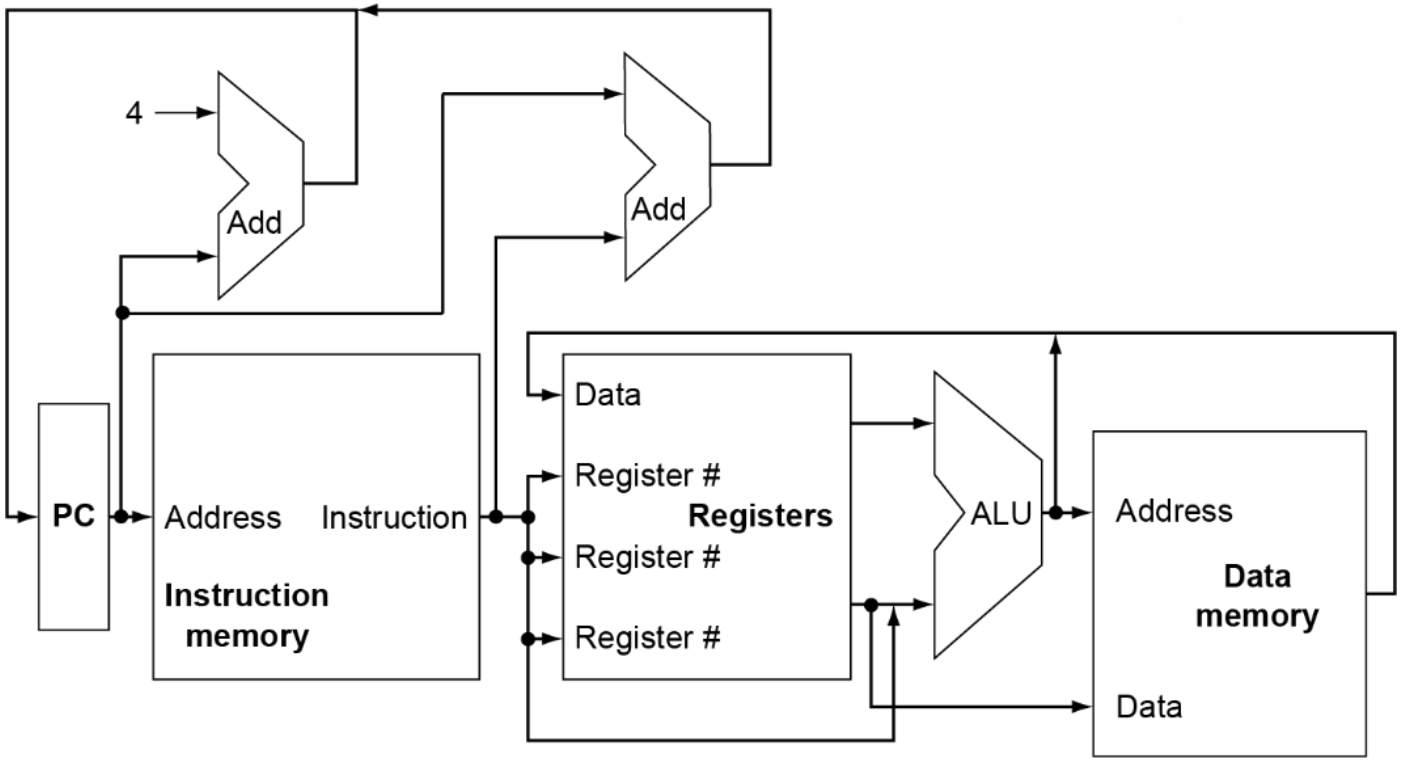
UJ-Type：Jal；

Instruction Execution Overview

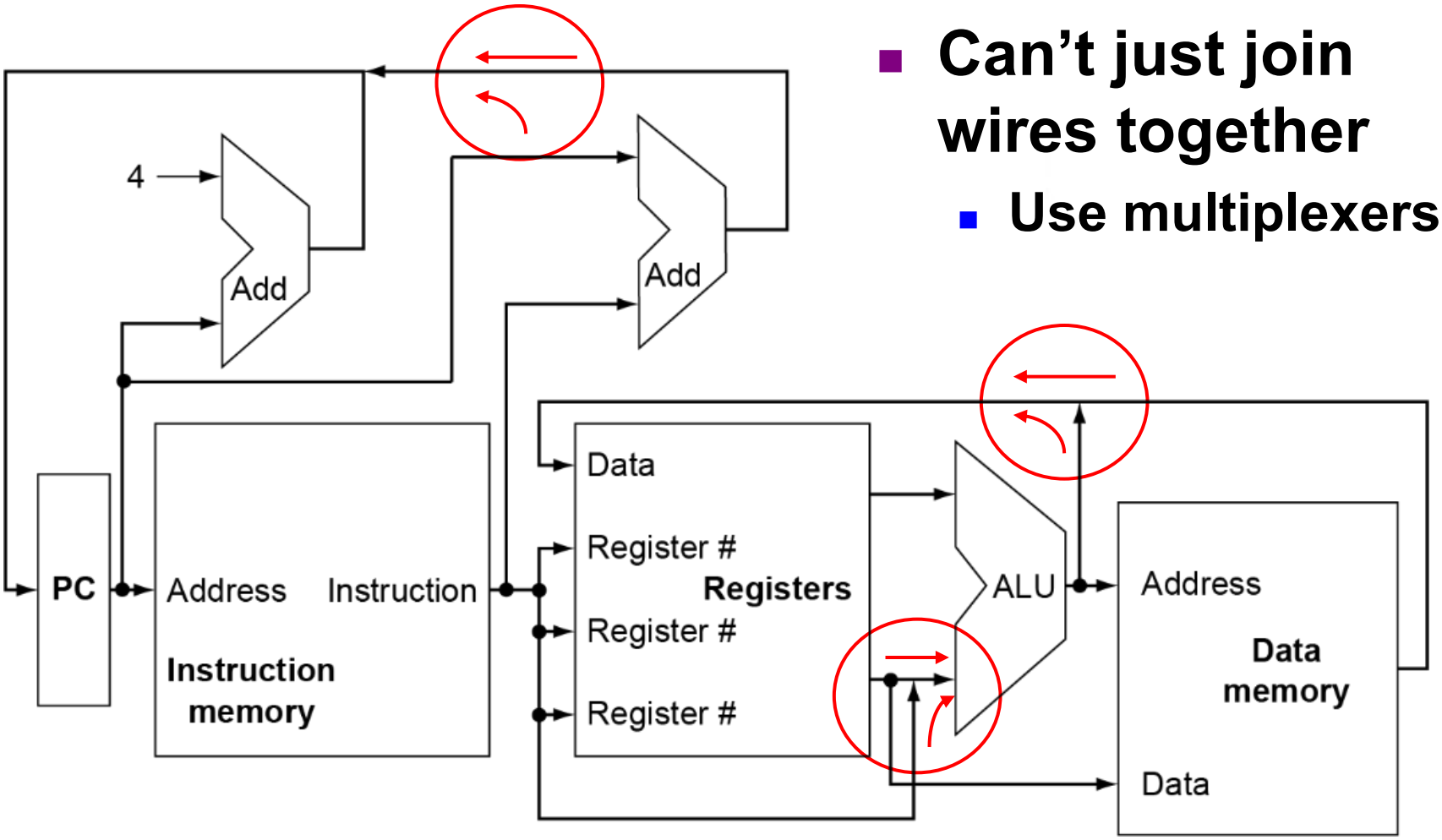
- **For every instruction, the first two step are identical**
  - **Fetch the instruction from the memory**
  - **Decode and read the registers**
- **Next steps depend on the instruction class**
  - **Memory-reference Arithmetic-logical branches**
- **Depending on instruction class**
  - Use ALU to calculate
    - Arithmetic result
    - Memory address for load/store
    - Branch comparison
  - Access data memory for load/store
  - PC ← target address or PC + 4

An overview of Implementation

- Send PC to the code memory.
- Read one or two register.



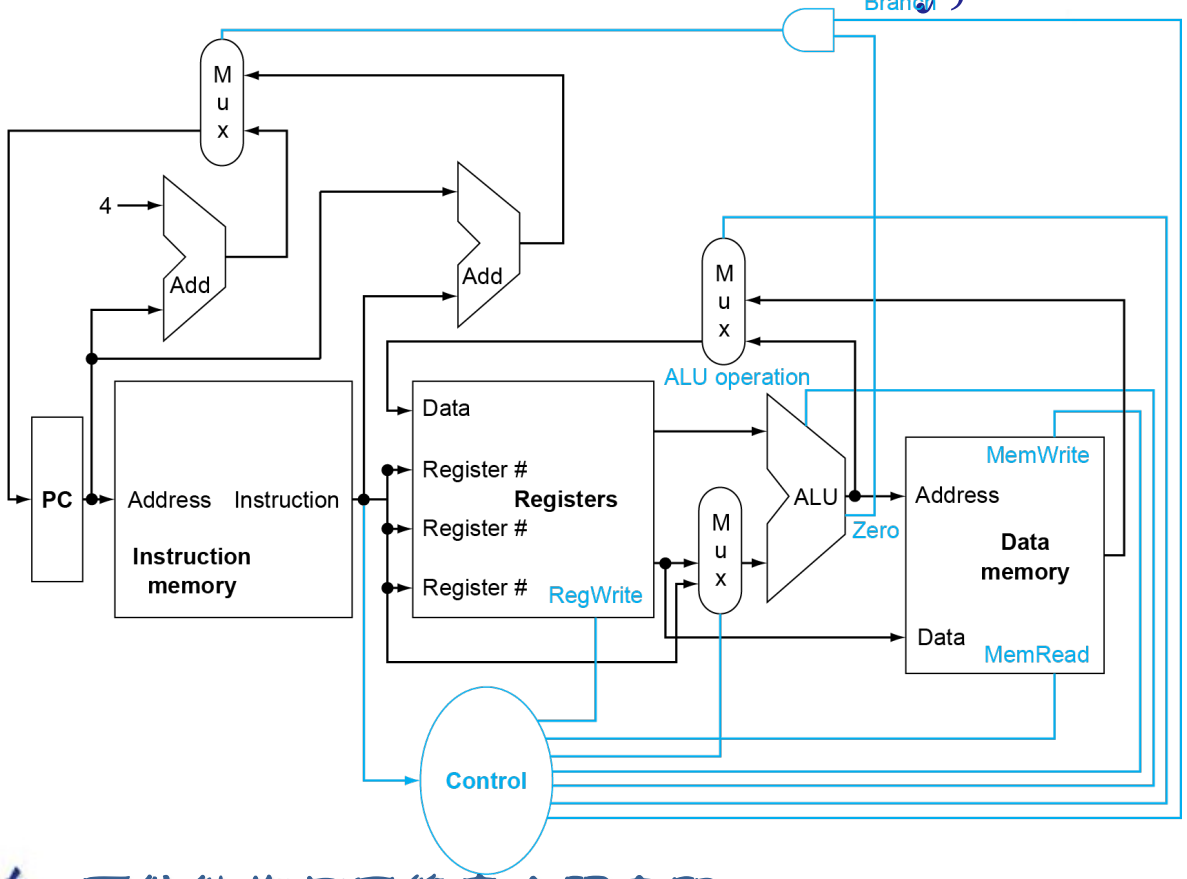
Multiplexers



- Can't just join wires together
  - Use multiplexers

Control

- Different Sources for unit.
- Control the units: Read Memory, Write Memory.

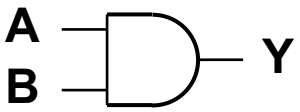


4.2 Logic Design Convention

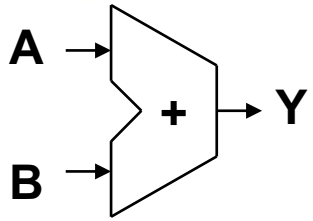
- **Information encoded in binary**
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- **Combinational element**
  - Operate on data
  - Output is a function of input
- **State (sequential) elements**
  - Store information

Combinational Elements

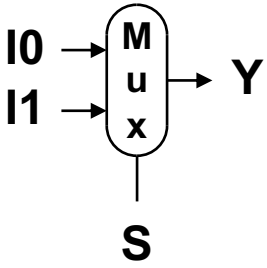
- **AND-gate**
  - $Y = A \& B$



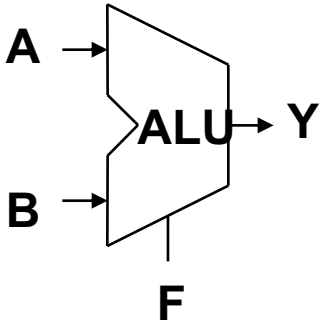
- **Adder**
  - **Y = A + B**



- **Multiplexer**
  - **Y = S ? I1 : I0**



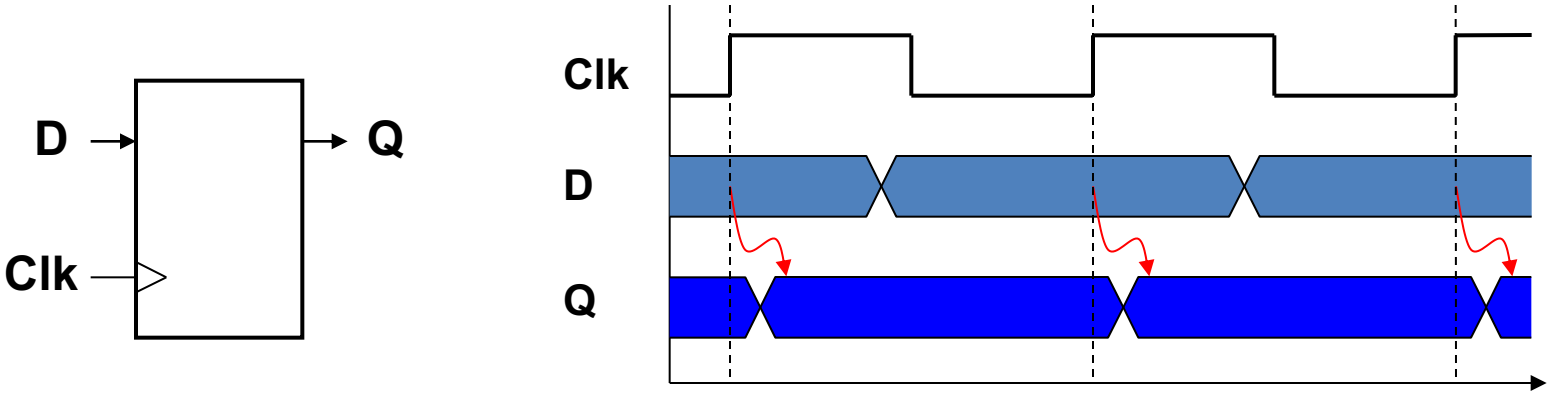
- **Arithmetic/Logic Unit**
  - **Y = F(A, B)**



Sequential Elements

Register: stores data in a circuit

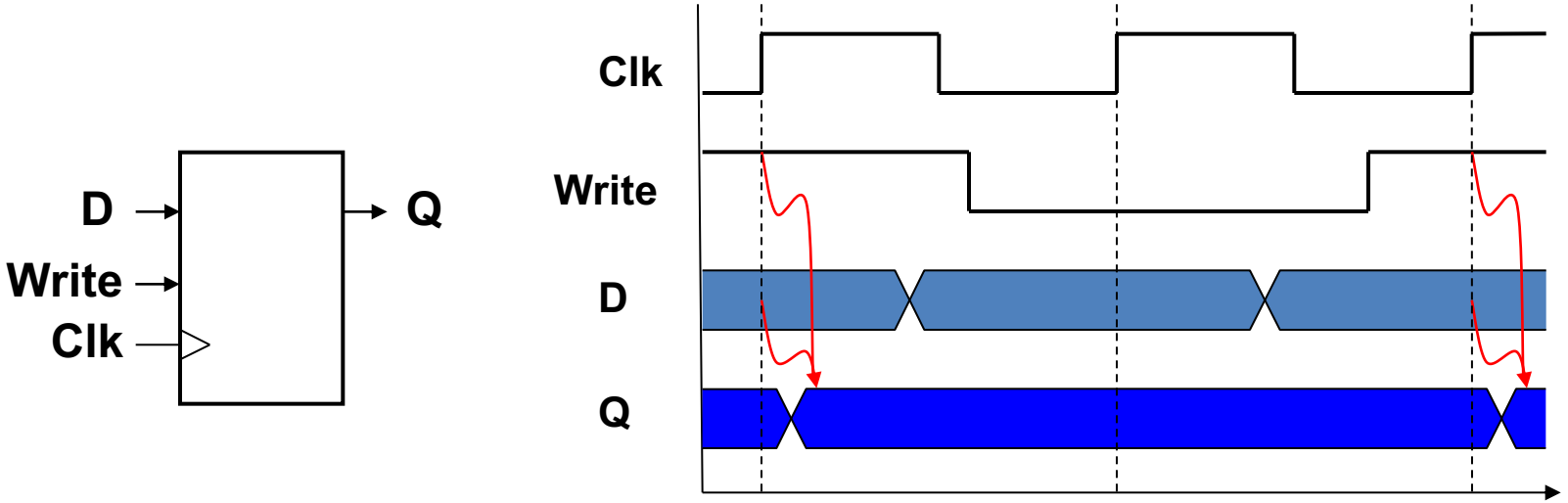
- Uses a clock signal to determine when to update the stored value
- Edge-triggered: update when Clk changes from 0 to 1



Sequential Elements

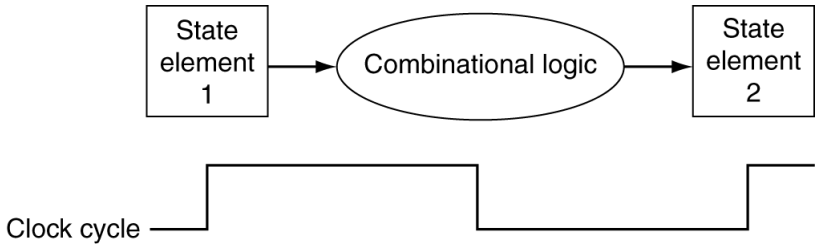
Register with write control

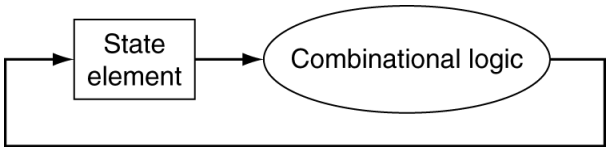
- Only updates on clock edge when write control input is 1
- Used when stored value is required later



Clocking Methodology

- **Combinational logic transforms data during clock cycles**
  - Between clock edges
  - Input from state elements, output to state element
  - Longest delay determines clock period





4.3 Building a Datapath

- **Datapath**
  - Elements that process data and addresses in the CPU
    - Registers, ALUs, mux's, memories, …
- **We will build a RISC-V datapath incrementally**
  - Refining the overview design

RISC-V field (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
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-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.*
- ***Immediate**: address or immediate*

The datapath there are ……

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[18,446,744,073,709,551,608]]	Accessed only by data transfer instructions. RISC-V uses byte addresses, so sequential double word accesses differ by 8. Memory holds data structures, arrays, and spilled registers.

Name	Register no.	Usage	Preserved on call
x0(zero)	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(t0-t2)	5-7	Temporaries	no
x8(s0/fp)	8	Saved/frame point	Yes
x9(s1)	9	Saved	Yes
x10-x17(a0-a7)	10-17	Arguments/results	no
x18-x27(s2-s11)	18-27	Saved	yes
x28-x31(t3-t6)	28-31	Temporaries	No
PC	-	Auipc(Add Upper Immediate to PC)	

RISC-V assembly language

Category	Instruction	Example	Meaning	Comments
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
	exclusive or	xor x5,x6,x7	x5=x6 ^ x7	Bit-by-bit XOR
	and immediate	andi x5,x6,20	x5=x6 & 20	Bit-by-bit AND reg. with constant
	inclusive or immediate	ori x5,x6,20	x5=x6 | 20	Bit-by-bit OR reg. with constant
	exclusive or immediate	xori x5,x6,20	X5=x6 ^ 20	Bit-by-bit XOR reg. with constant
Shift	shift left logical	sll x5, x6, x7	x5=x6 << x7	Shift left by register
	shift right logical	srl x5, x6, x7	x5=x6 >> x7	Shift right by register
	shift right arithmetic	sra x5, x6, x7	x5=x6 >> x7	Arithmetic shift right by register
	shift left logical immediate	slli x5, x6, 3	x5=x6 << 3	Shift left by immediate
Shift	shift right logical immediate	srli x5,x6,3	x5=x6 >> 3	Shift right by immediate
	shift right arithmetic immediate	srai x5,x6,3	x5=x6 >> 3	Arithmetic shift right by immediate
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
	branch if less than	blt x5, x6, 100	if(x5 < x6) go to PC+100	PC-relative branch if registers less
	branch if greater or equal	bge x5, x6, 100	if(x5 >= x6) go to PC+100	PC-relative branch if registers greater or equal
	branch if less, unsigned	bltu x5, x6, 100	if(x5 >= x6) go to PC+100	PC-relative branch if registers less, unsigned
	branch if greater or equal, unsigned	bgeu x5, x6, 100	if(x5 >= x6) go to PC+100	PC-relative branch if registers greater or equal, unsigned
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

Reg OP for RISC machine language

Format	Instruction	Opcode	Funct3	Funct6/7
R-type	add	0110011	000	0000000
	sub	0110011	000	0100000
	sll	0110011	001	0000000
	xor	0110011	100	0000000
	srl	0110011	101	0000000
	sra	0110011	101	0000000
	or	0110011	110	0000000
	and	0110011	111	0000000
	lr.d	0110011	011	0001000
	sc.d	0110011	011	0001100

Imm OP for RISC machine language

Format	Instruction	Opcode	Funct3	Funct6/7
I-type	lb	0000011	000	n.a.
	lh	0000011	001	n.a.
	lw	0000011	010	n.a.
	ld	0000011	011	n.a.
	lbu	0000011	100	n.a.
	lhu	0000011	101	n.a.
	lwu	0000011	110	n.a.
	addi	0010011	000	n.a.
	slli	0010011	001	000000
	xori	0010011	100	n.a.
	srli	0010011	101	000000
	srai	0010011	101	010000
	ori	0010011	110	n.a.
	andi	0010011	111	n.a.
	jalr	1100111	000	n.a.

S/U OP for RISC machine language

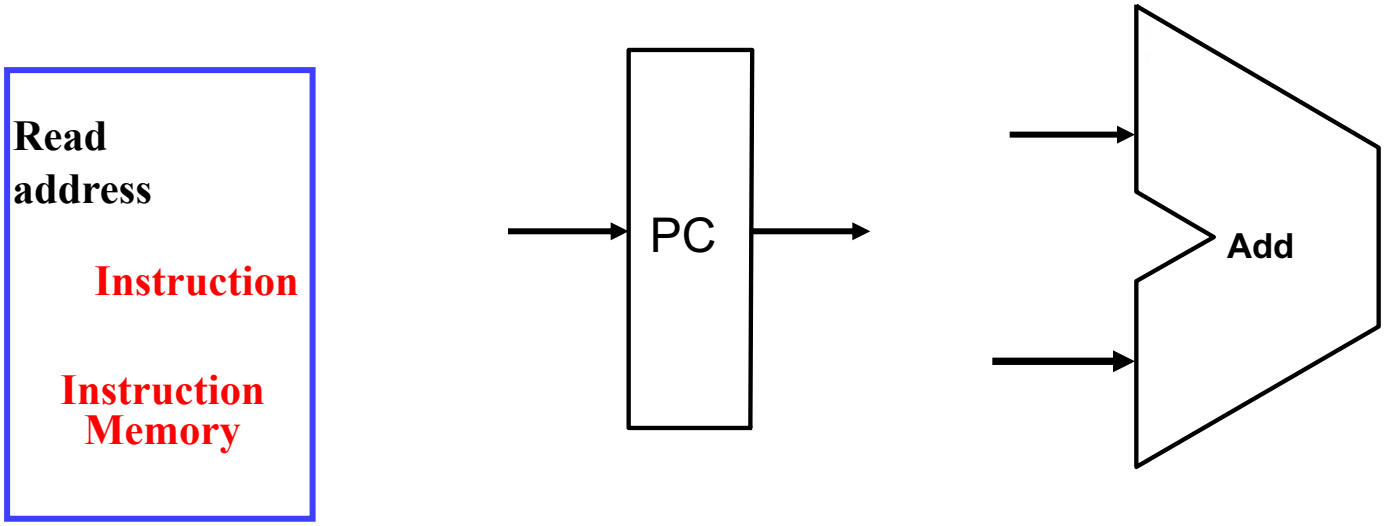
Format	Instruction	Opcode	Funct3	Funct6/7
S-type	sb	0100011	000	n.a.
	sh	0100011	001	n.a.
	sw	0100011	010	n.a.
	sd	0100011	111	n.a.
SB-type	beq	1100111	000	n.a.
	bne	1100111	001	n.a.
	blt	1100111	100	n.a.
	bge	1100111	101	n.a.
	bltu	1100111	110	n.a.
	bgeu	1100111	111	n.a.
U-type	lui	0110111	n.a.	n.a.
UJ-type	jal	1101111	n.a.	n.a.

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
  - 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

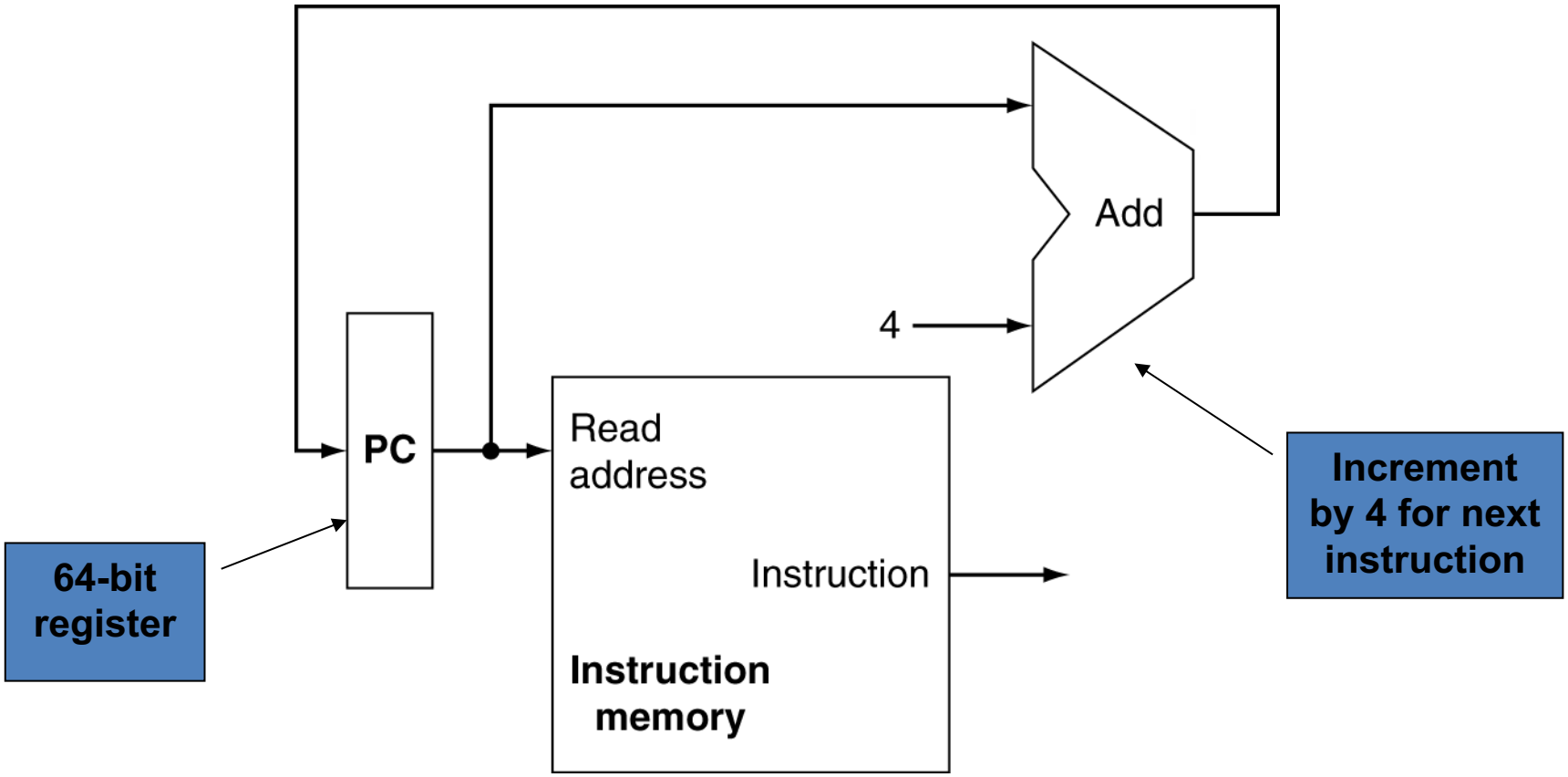
Instruction fetching three elements

How to connect?



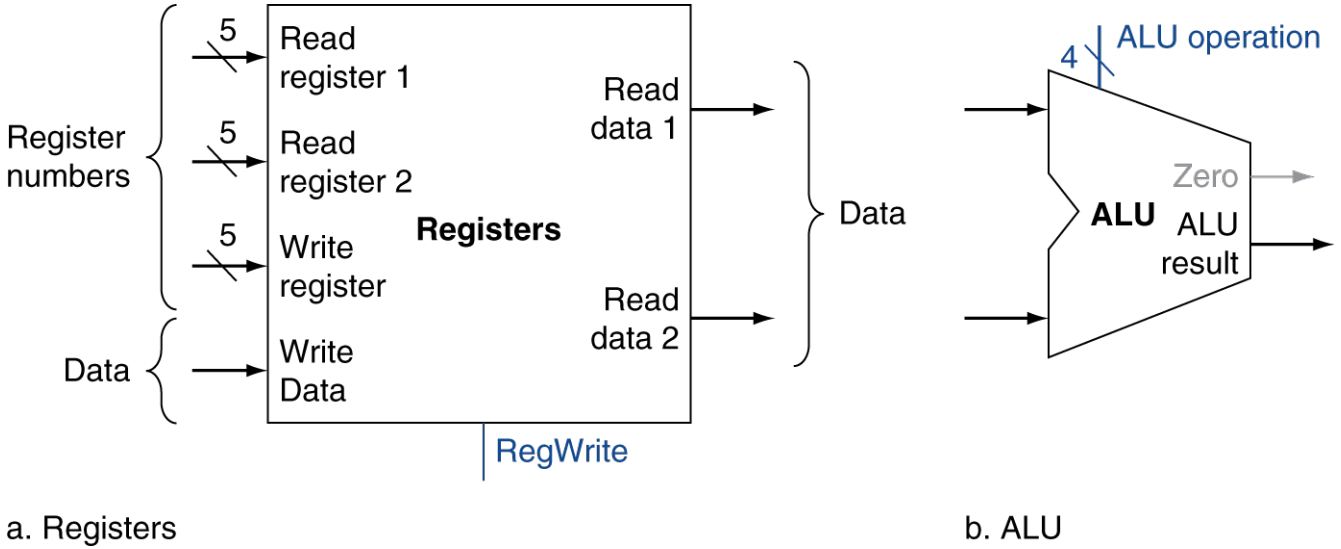
Instruction memory　　Program counter　　Adder

Instruction Fetch



R-Format Instructions

- Read two register operands
- Perform arithmetic/logical operation
- Write register result

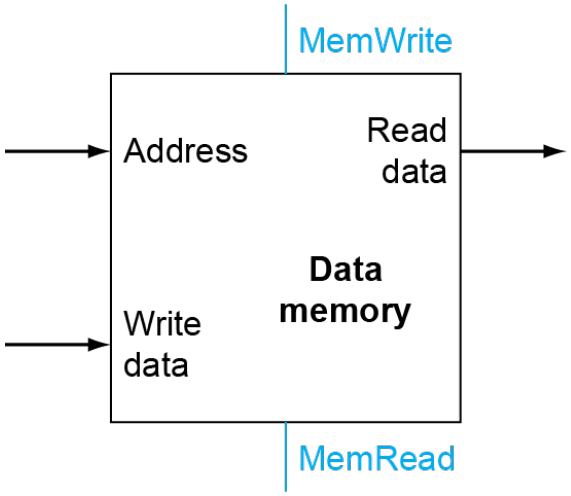


a. Registers

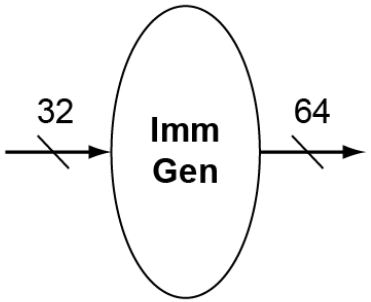
b. ALU

Load/Store Instructions

- Read register operands
- Calculate address using 12-bit offset
  - Use ALU, but sign-extend offset
- Load: Read memory and update register
- Store: Write register value to memory



a. Data memory unit

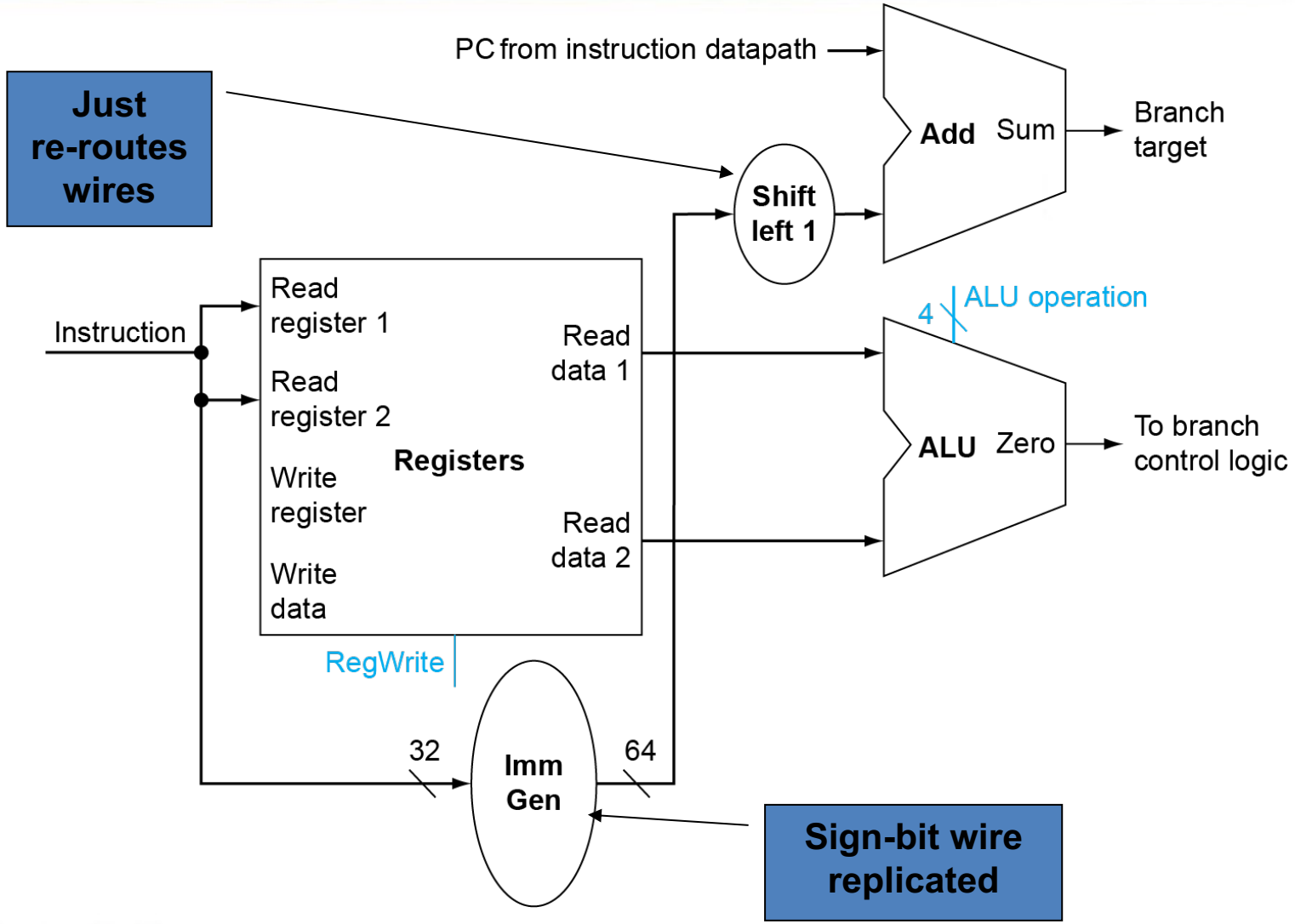


b. Immediate generation unit

Branch Instructions

- **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

Branch Instructions



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**

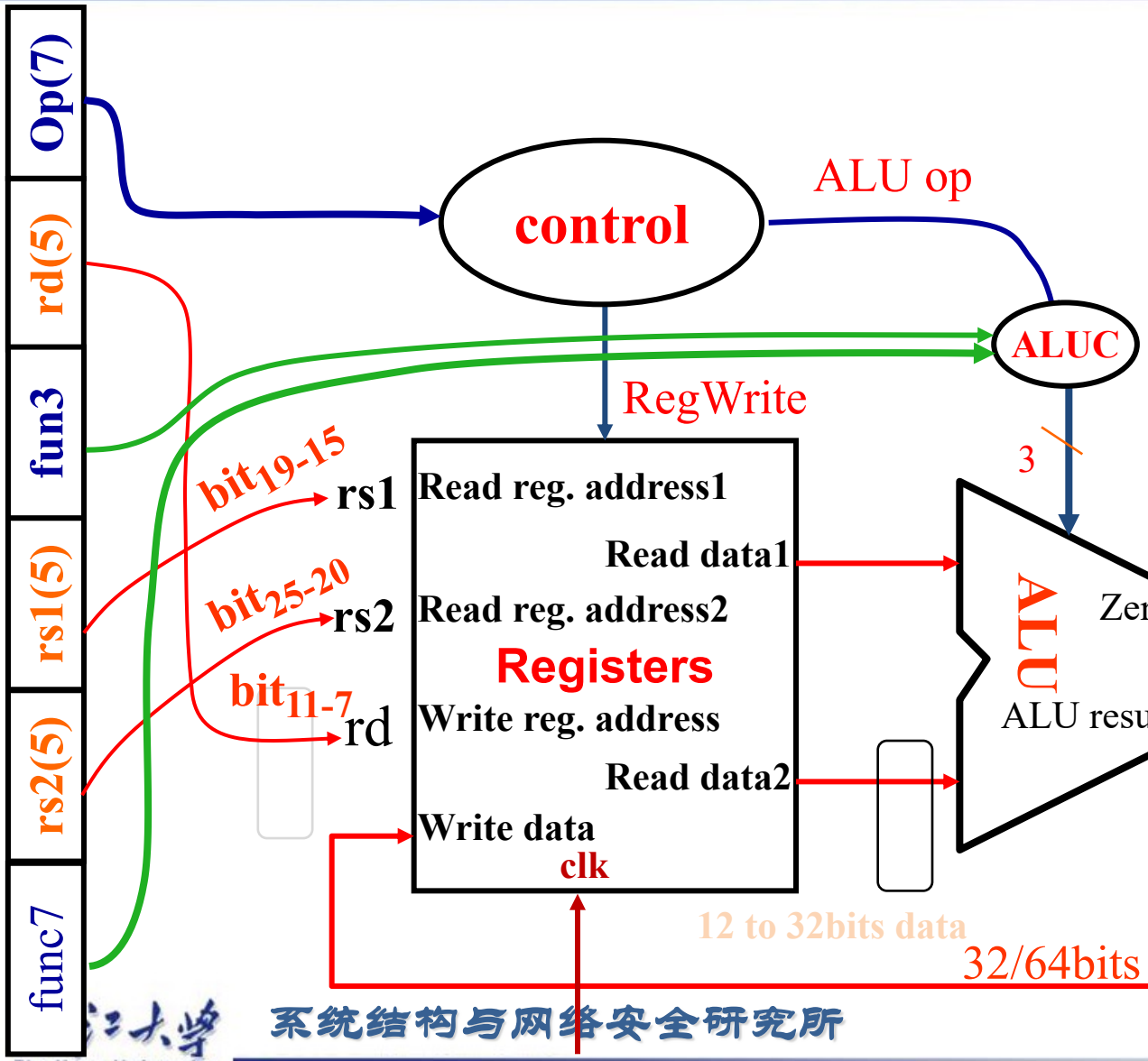
Path Built using Multiplexer

- R-type instruction Datapath
- I-type instruction Datapath
  - For ALU
  - For load
- S-type (store) instruction Datapath
- SB-type (branch) instruction Datapath
- UJ-type instruction Datapath
  - For Jump

First, Look at the data flow within instruction execution

R type Instruction & Data stream

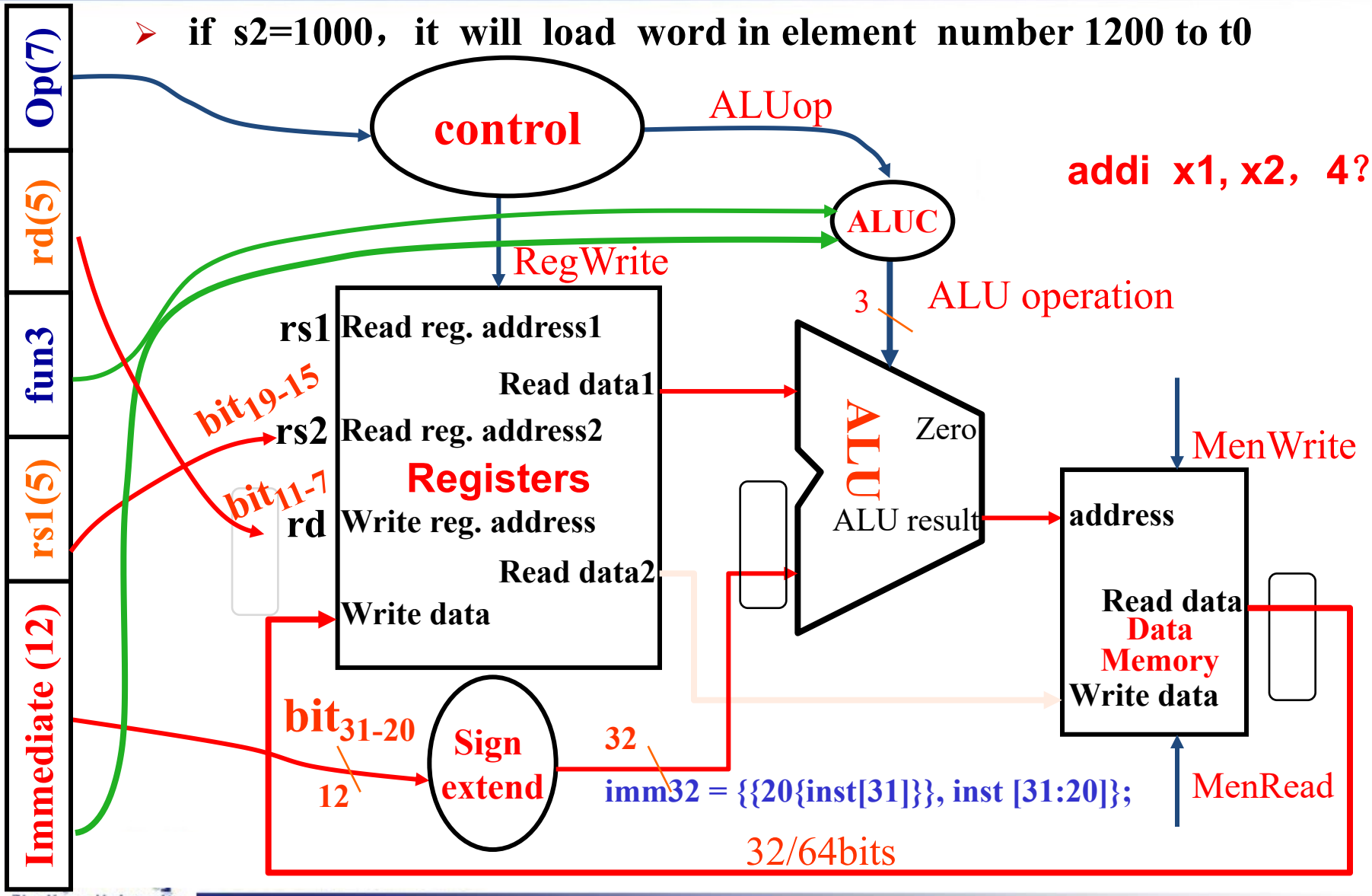
add s1, s4, s5



Bnegate	op	function
0	00	and
0	01	Or
0	10	Add
1	10	Sub
1	11	Slt

I type Instruction & Data stream

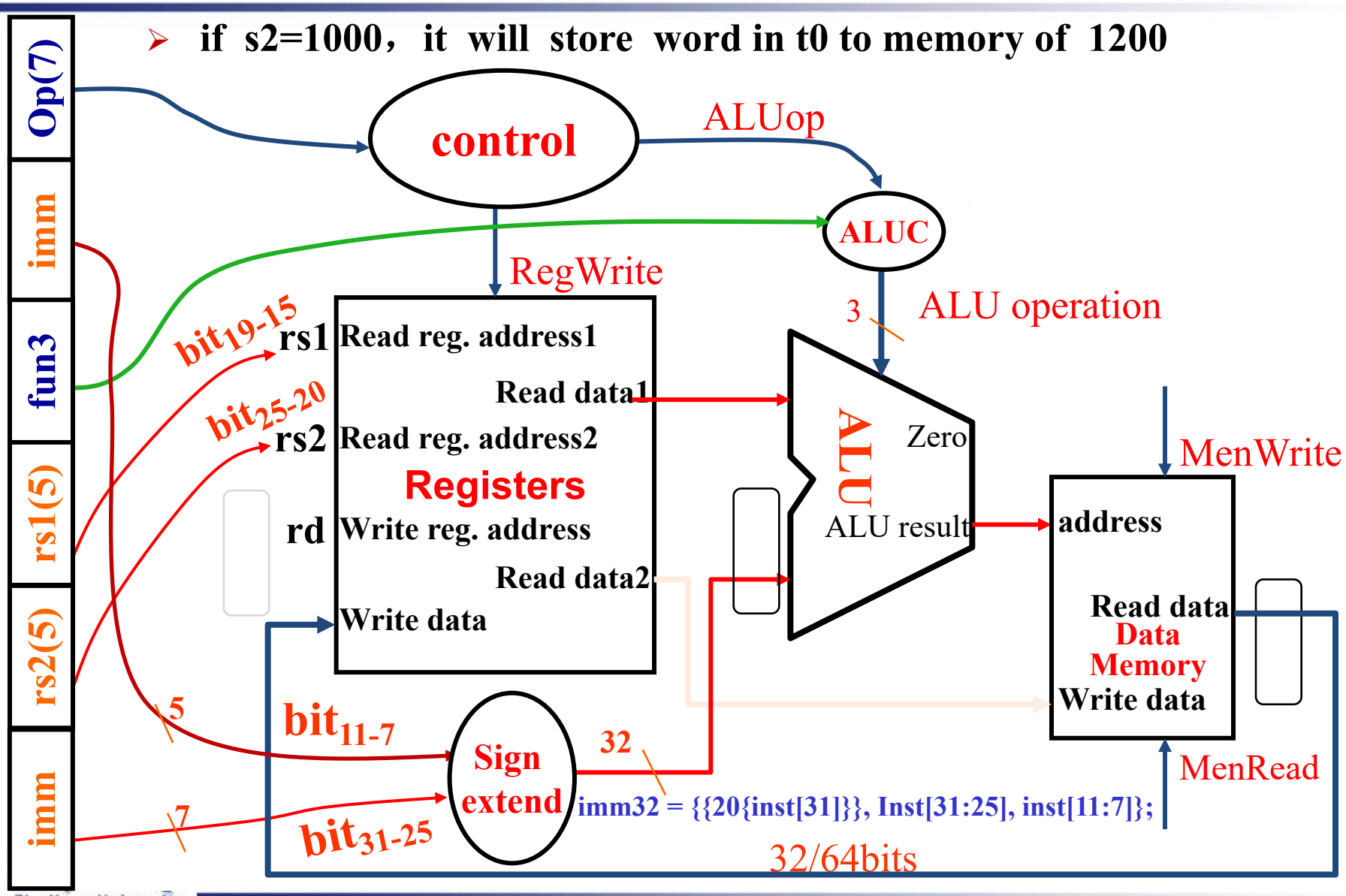
lw t0, 200(s2)



S type Instruction & Data stream

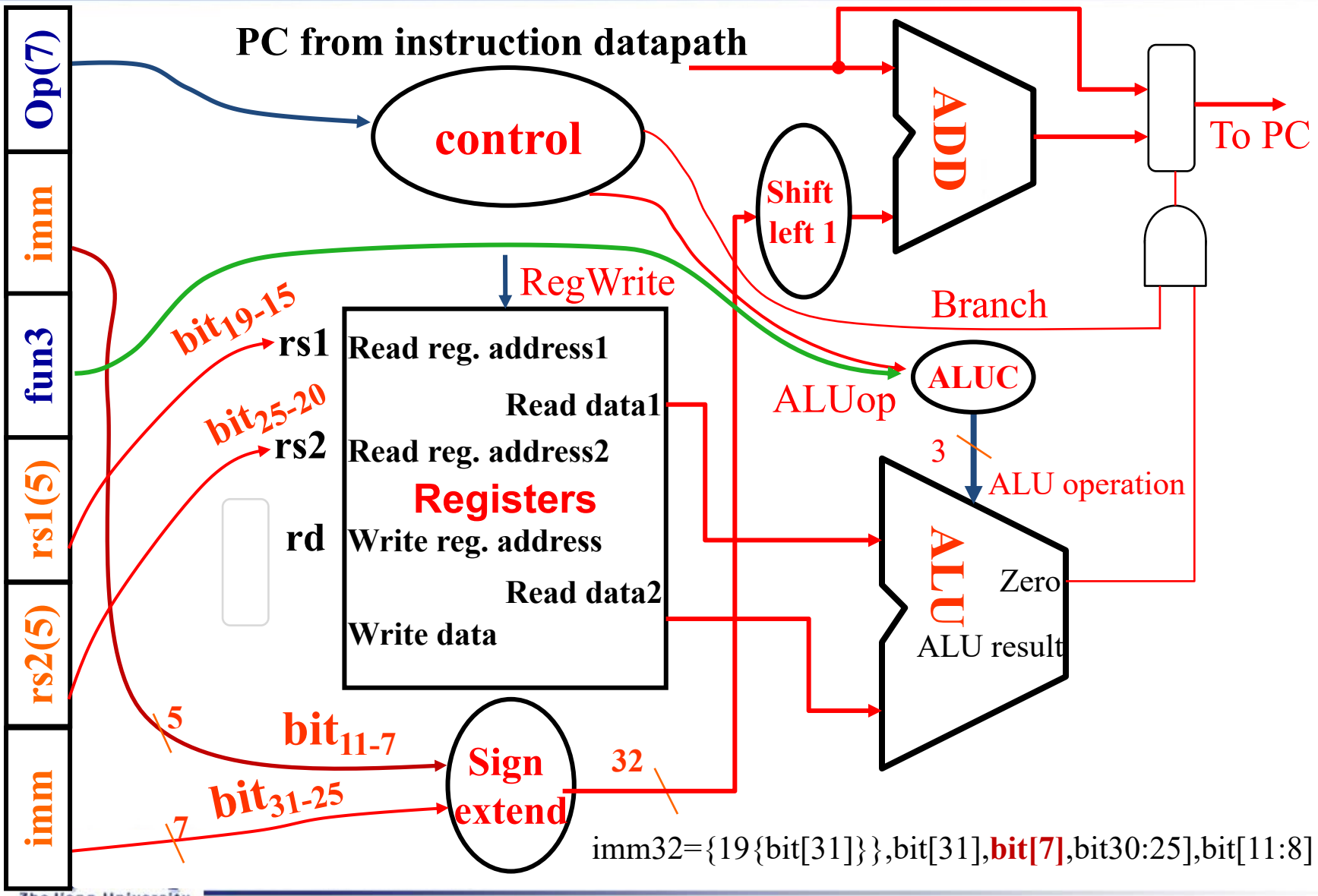
sw t0, 200(s2)

- if s2=1000，it will store word in t0 to memory of 1200

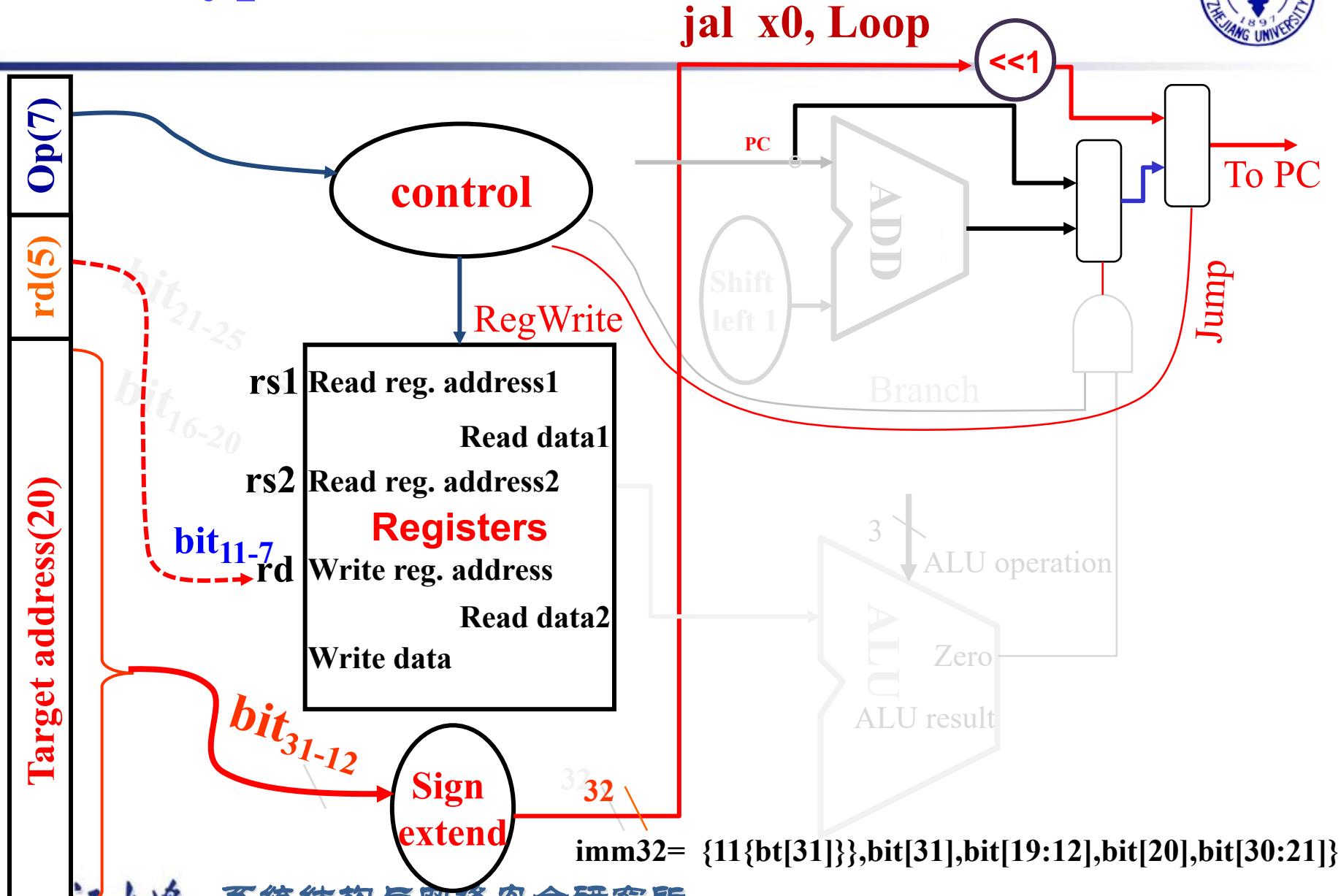


SB type Instruction & Data stream

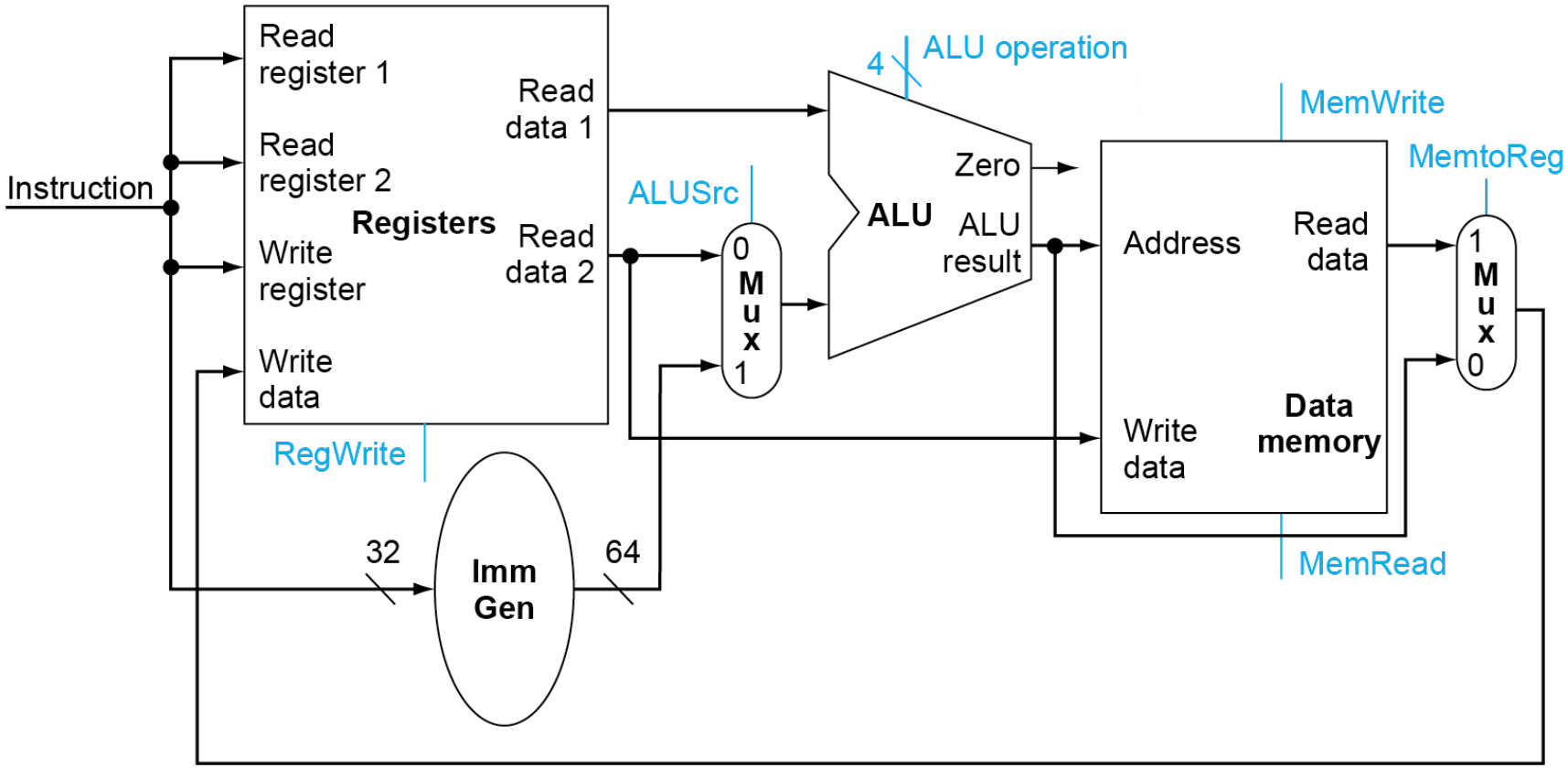
beq t0, t2， 200



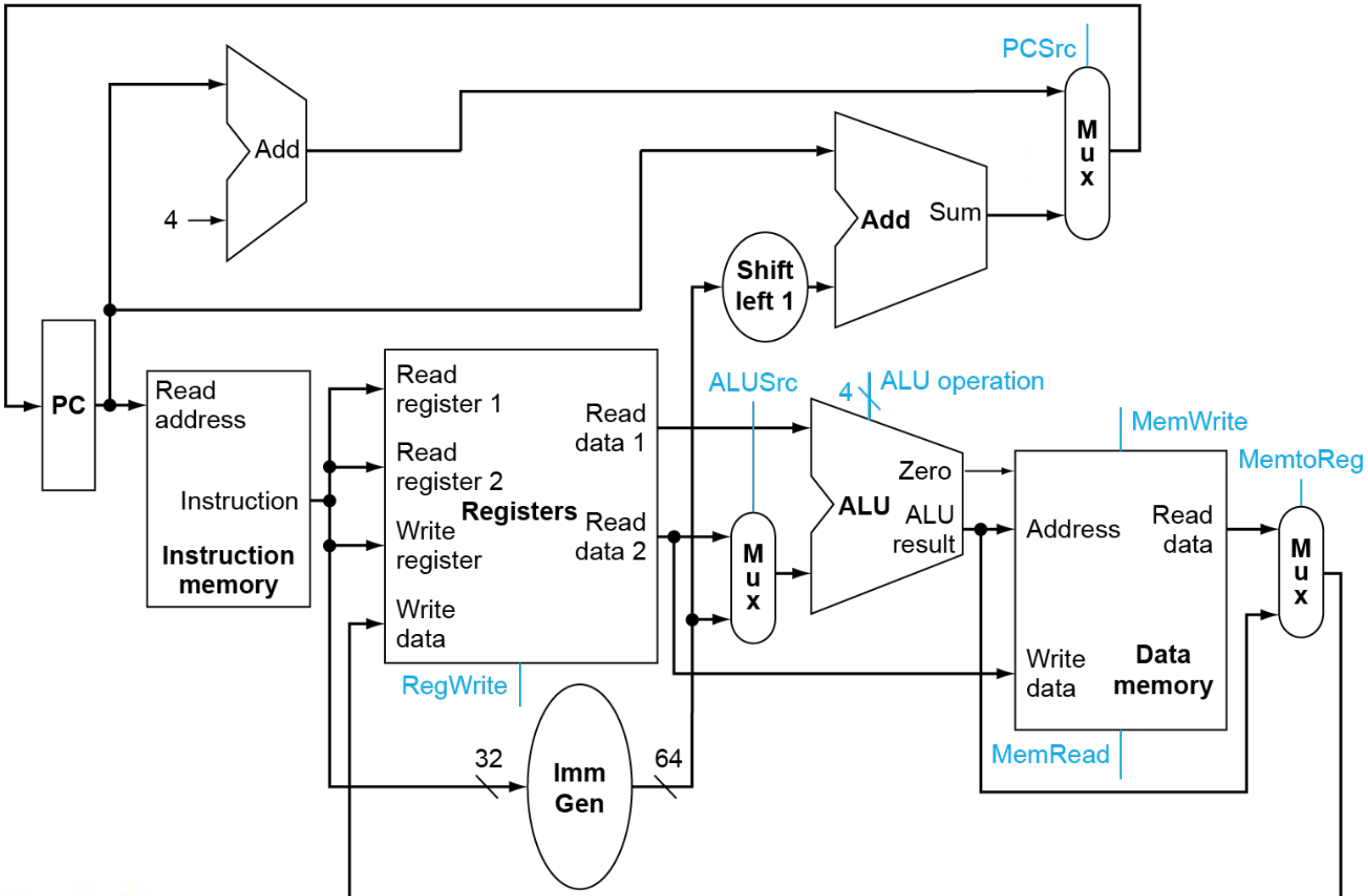
Jal/J type Instruction & Data stream



R-Type/Load/Store Datapath

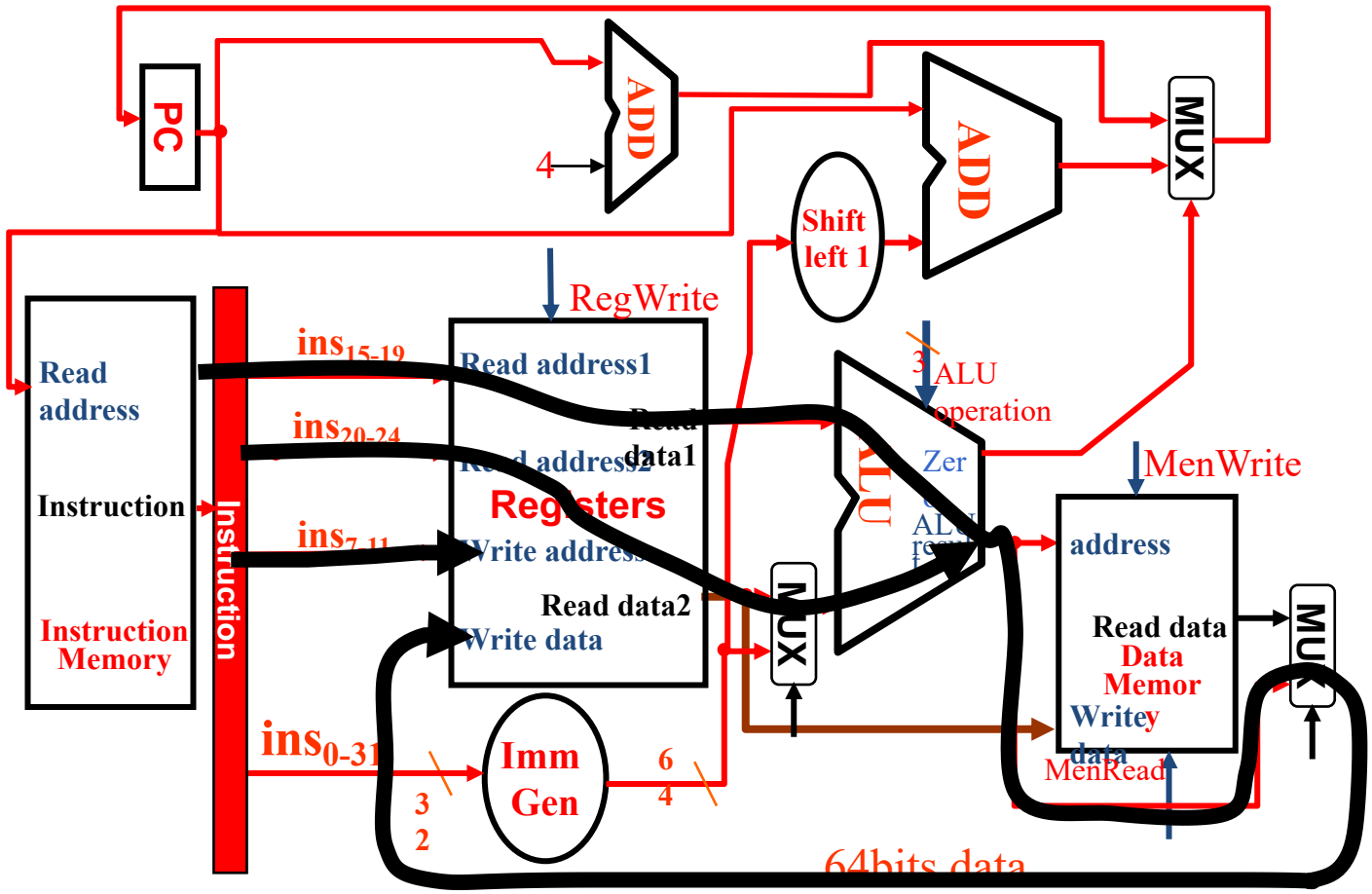


Full Datapath



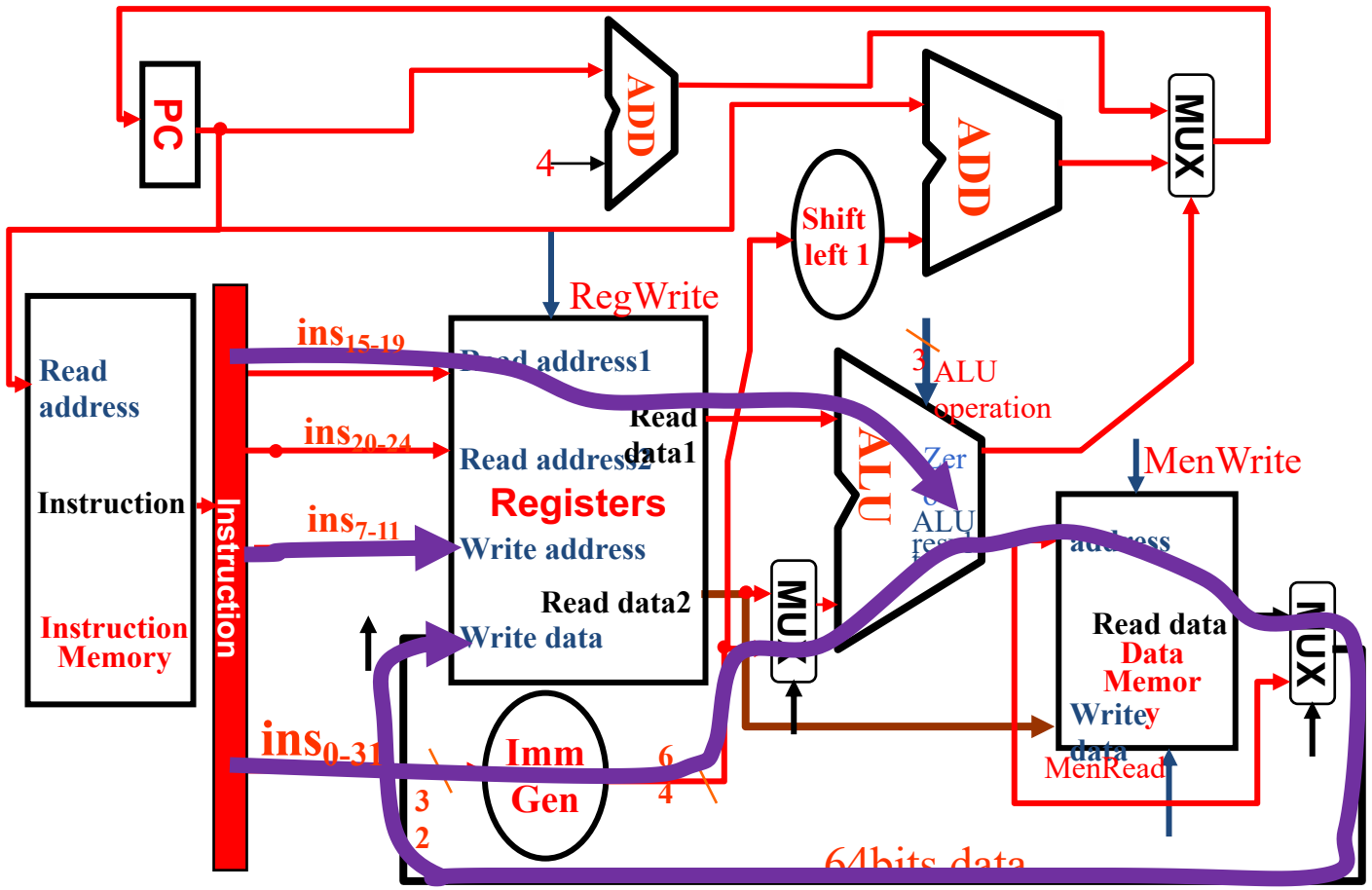
Full datapath

R



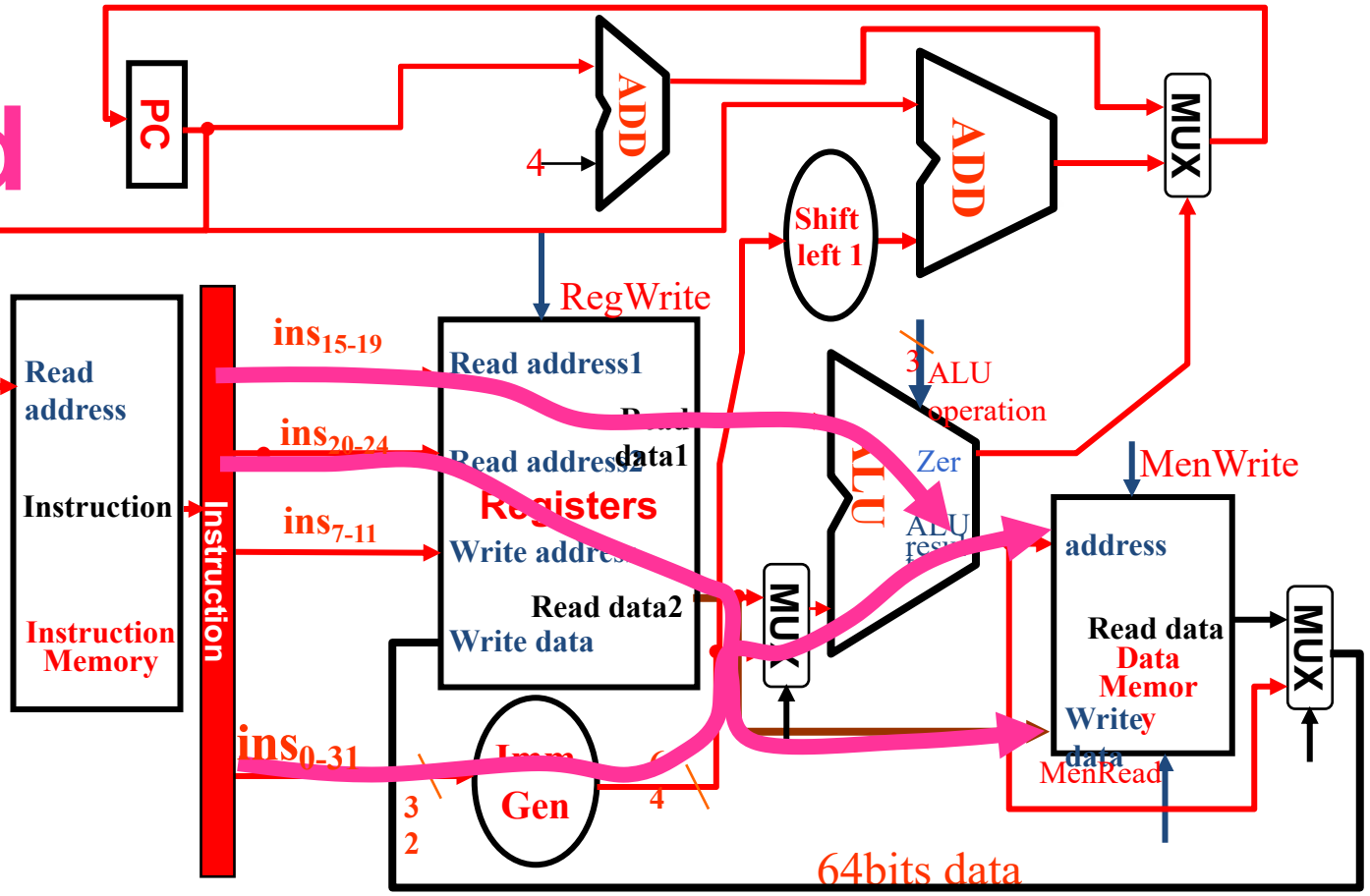
Full datapath

I-ld

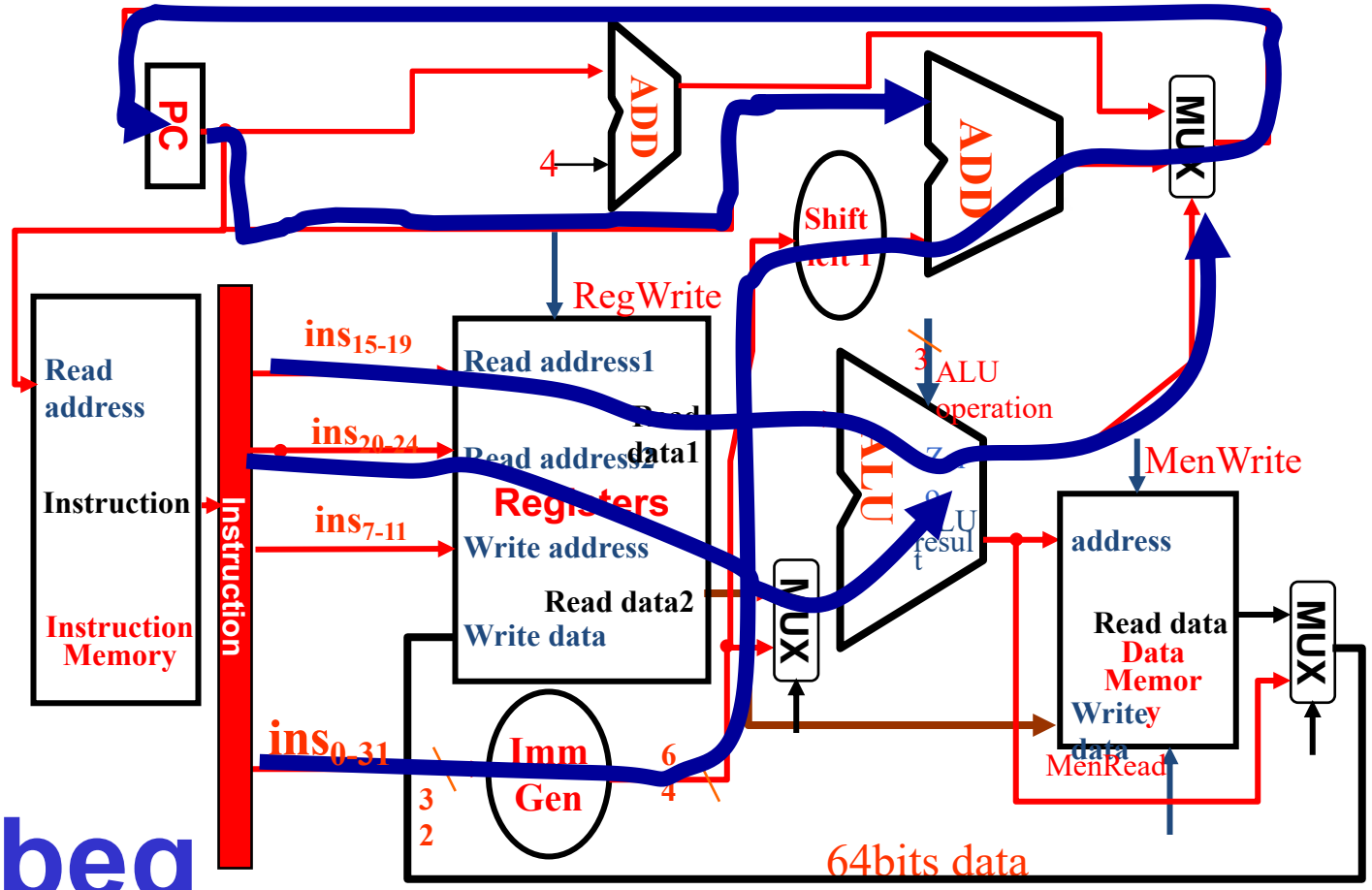


Full datapath

S-sd



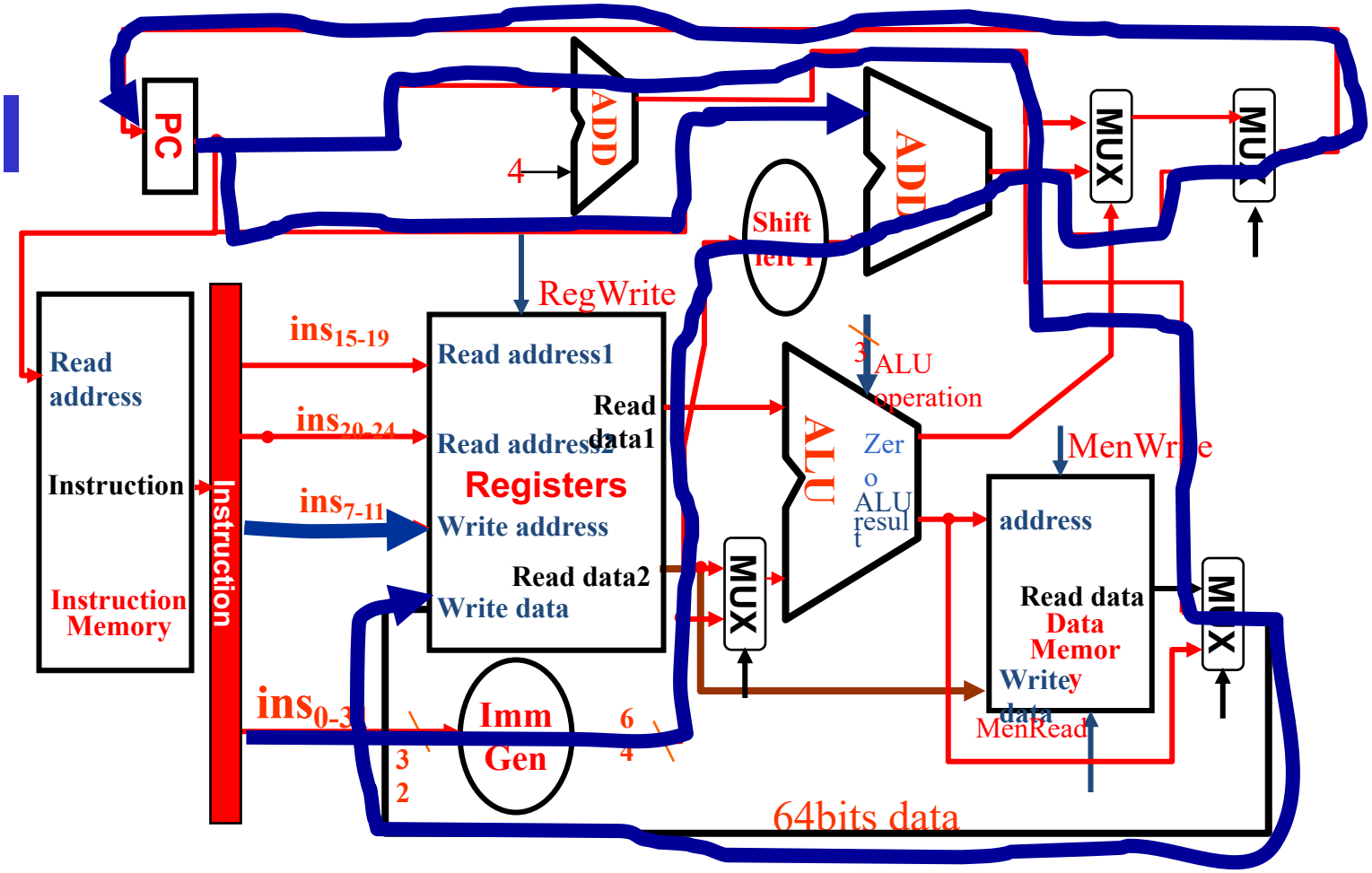
Full datapath



SB-beq

Full datapath

J-jal



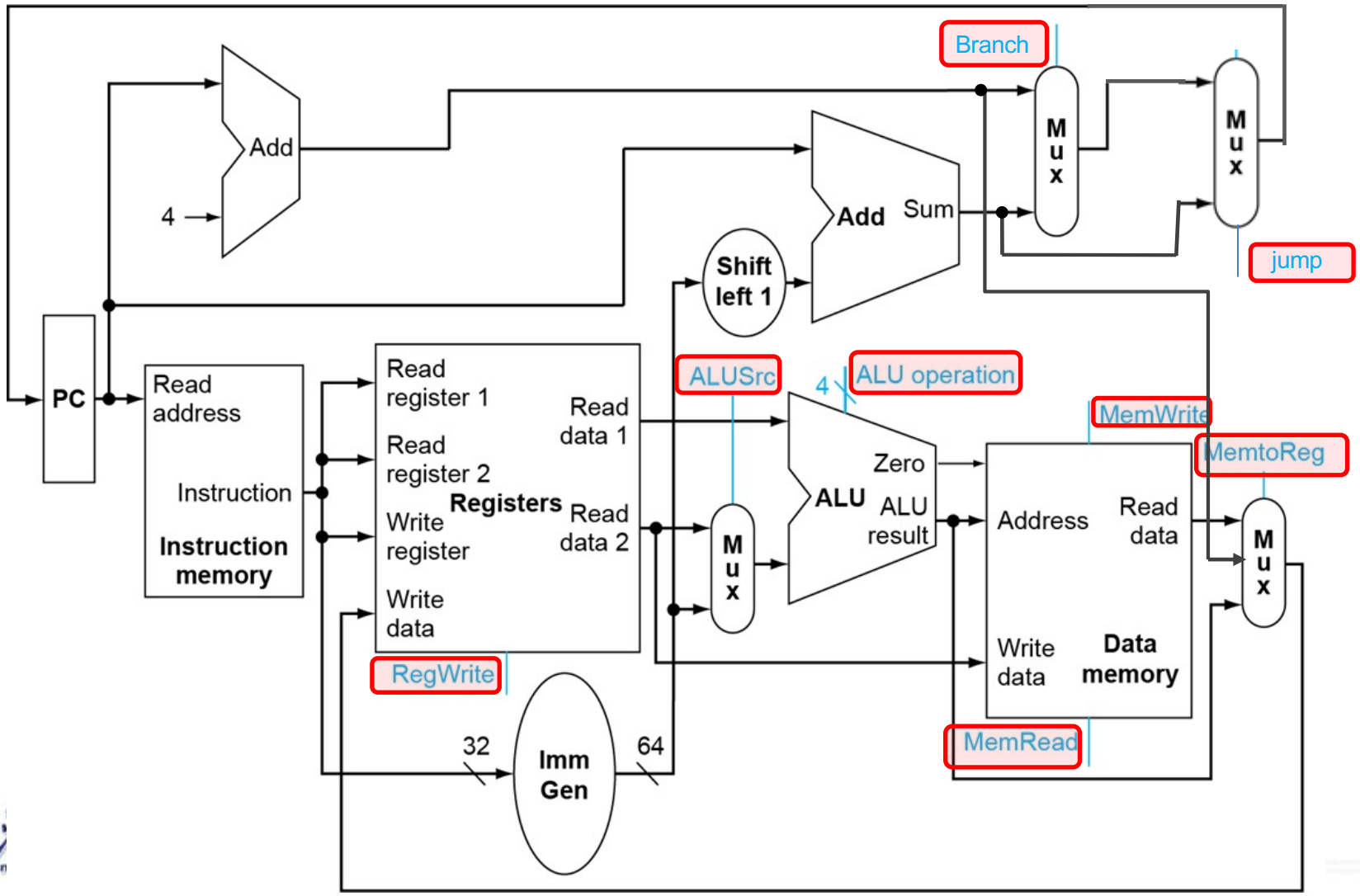
4.4 A simple Implementation Scheme

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

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
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

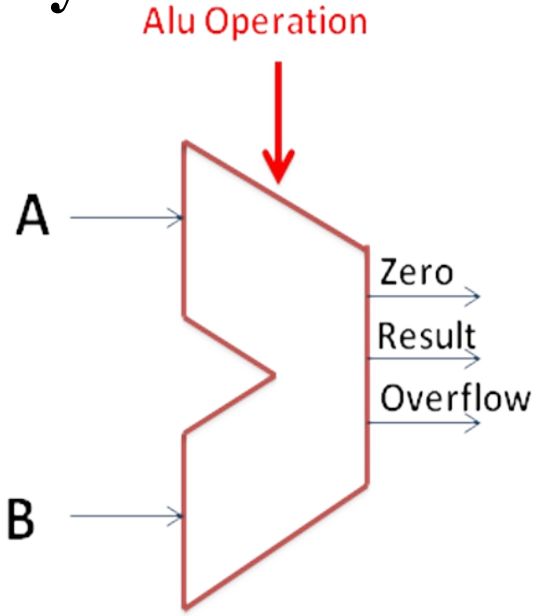
Building Controller

- There are 7+4 signals



ALU symbol & Control

- Symbol of the ALU



Control： Function table

ALU Control Lines	Function
000	And
001	Or
010	Add
110	Sub
111	Set on less than
100	nor
101	srl
011	xor

Signals for datapath Defined 7 control signals

Signal name	Effect when deasserted(=0)	Effect when asserted(=1)
RegWrite	None	Register destination input is written with the value on the Write data input
ALUScr	The second ALU operand come from the second register file output (Read data 2)	The second ALU operand is the sign-extended lower 16 bits of the instruction..
Branch (PCSrc)	The PC is replaced by the output of the adder that computers the value PC+4	The PC is replaced by the output of the adder that computers the branch target.
Jump	The PC is replaced by PC+4 or branch target	The PC is updated by jump address computed by adder
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 value on the Write data input.
MemtoReg (2位)	00：The value fed to register Write data input comes from the Alu	01：The value fed to the register Write data input comes from the data memory.
		10：The value fed to the register Write data input comes from PC+4

ALU Control

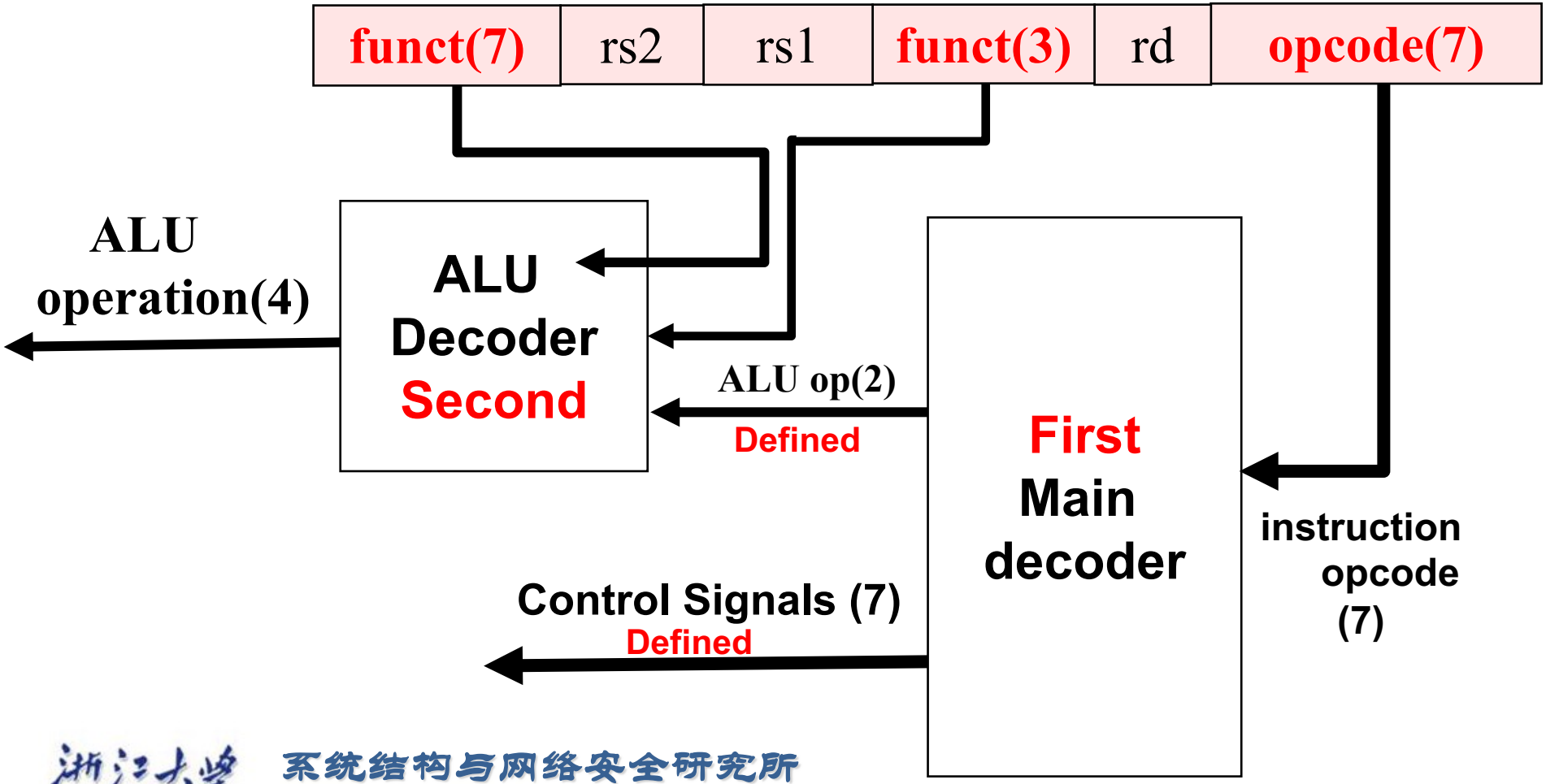
ALU used for

- Load/Store: F = add
- Branch: F = subtract
- R-type: F depends on opcode

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract

Scheme of Controller

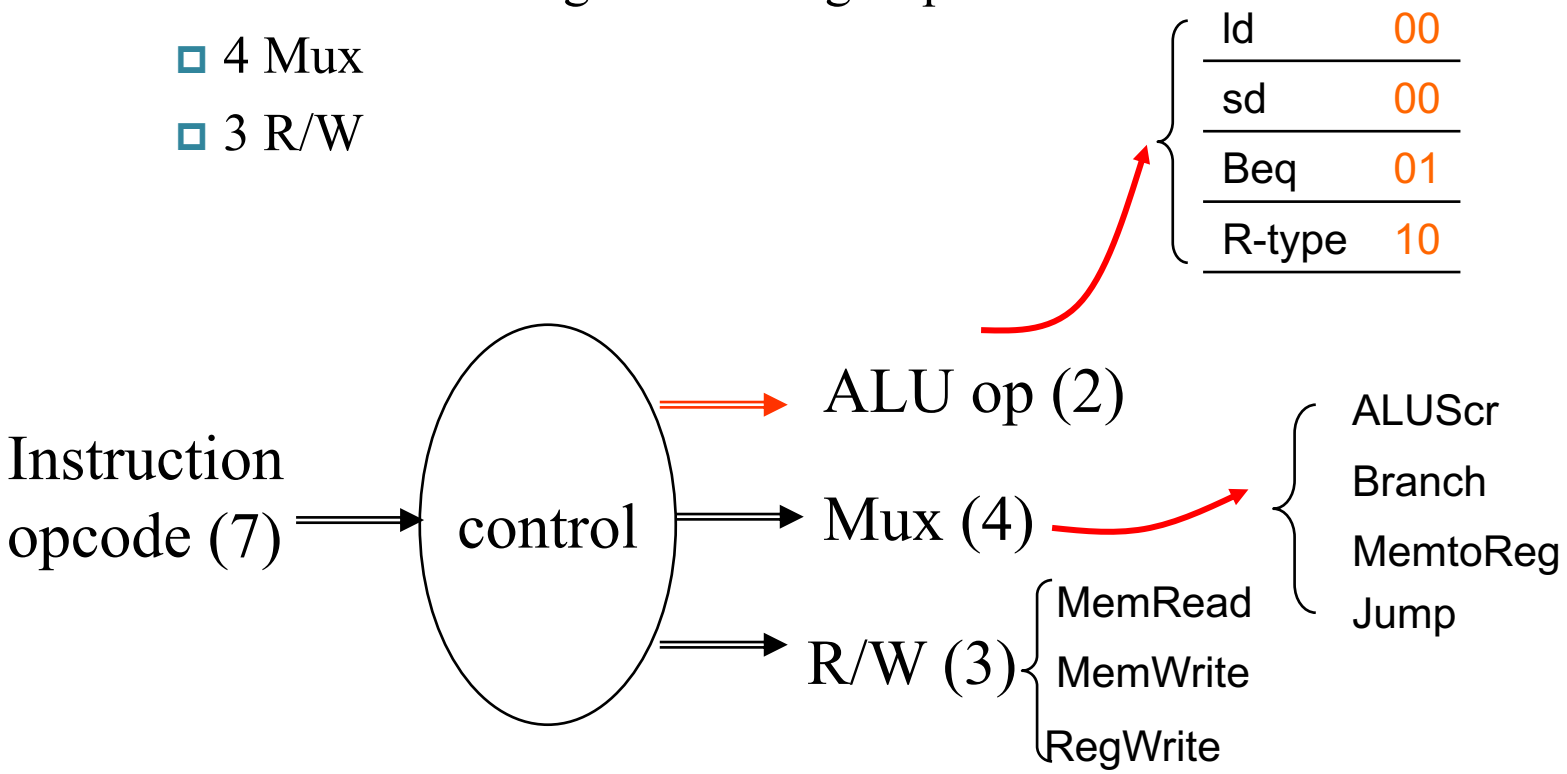
2-level decoder



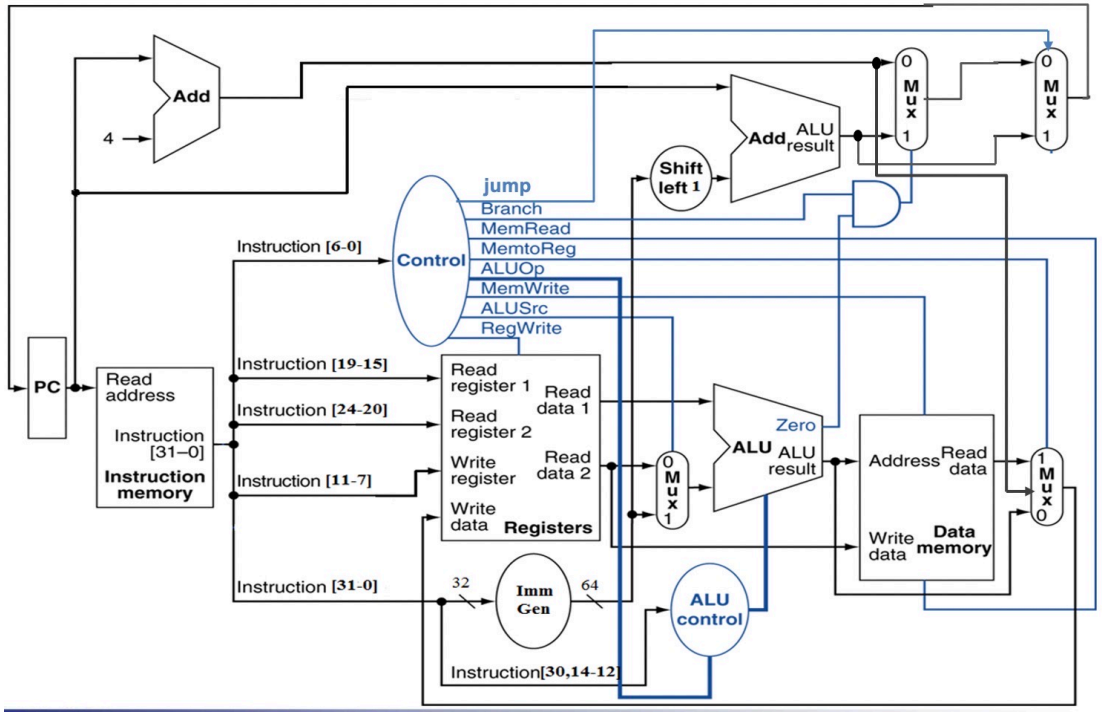
Designing the Main Control Unit

First level

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



Truth tables & Circuitry of main Controller



输入		输出								
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(SB-Type)	1100111	0	X	0	0	0	1	0	0	1
Jal(UJ-Type)	1101111	X	10	1	0	0	0	1	X	X

Main Controller Code

指令译码器参考描述

```
`define CPU_ctrl_signals {ALUSrc_B,MemtoReg,RegWR,MemWrite,Branch,Jump,ALUop}
        always @* begin
                case(OPcode)
                5‘b01100: begin CPU_ctrl_signals = ?;  end          //ALU
                5’b00000: begin CPU_ctrl_signals = ?;   end         //load
                5’b01000: begin CPU_ctrl_signals = ?;   end         //store
                5’b11000: begin CPU_ctrl_signals = ?;   end         //beq
                5’b11011: begin CPU_ctrl_signals = ?;   end         //jump
            5’b00100: begin CPU_ctrl_signals = ?;   end   //ALU(addi;;;;)
                                                  ……
                default:        begin  CPU_ctrl_signals = ?;  end
                endcase
        end
```

Design 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

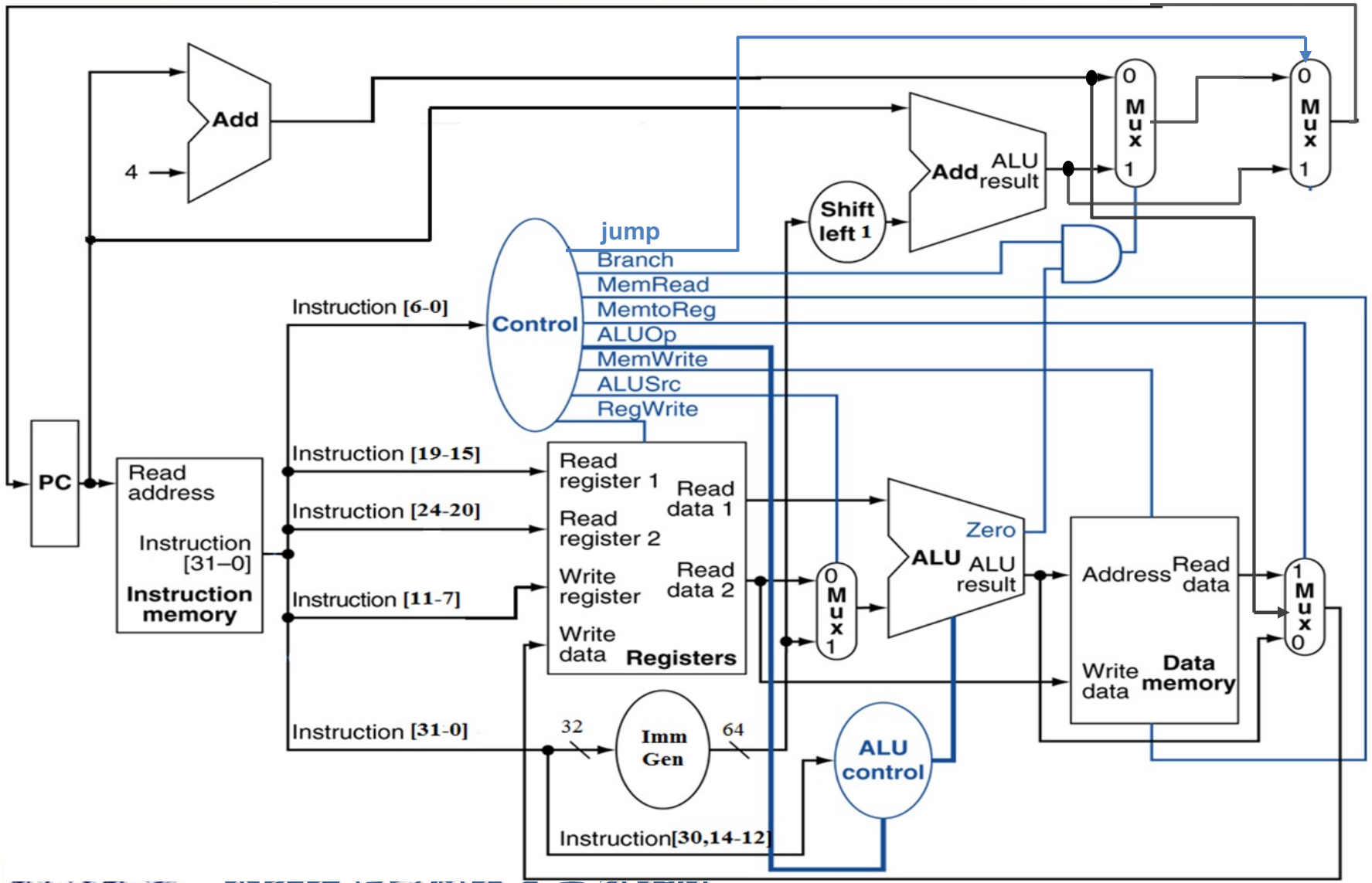
opcode	ALUOp	Operation	Funct7	funct3	ALU function	ALU control
ld	00	load register	XXXXXXX	xxx	add	0010
sd	00	store register	XXXXXXX	xxx	add	0010
beq	01	branch on equal	XXXXXXX	xxx	subtract	0110
R-type	10	add	0000000	000	add	0010
		subtract	0100000	000	subtract	0110
		AND	0000000	111	AND	0000
		OR	0000000	110	OR	0001
		SLT	0000000	010	Slt	0111

ALU Controller Code

- **ALU Control HDL Description**

```
assign Fun = {Fun3,Fun7};
always @* begin
    case(ALUop)
    2’b10: ALU_Control = ? ;                                  //add计算地址
    2’b11: ALU_Control = ? ;                                  //sub比较条件
    2’b00:
        case(Fun)
                4'b0000: ALU_Control = 3'b010 ;       //add
                4’b0001: ALU_Control = ? ;            //sub
                4’b1110: ALU_Control = ? ;            //and
                4’b1100: ALU_Control = ? ;            //or
                4’b0100: ALU_Control = ? ;            //slt
                4’b1010: ALU_Control = ? ;            //srl
                4’b1000: ALU_Control = ? ;            //xor
                        ……
                default:   ALU_Control=3'bx;
                endcase
2’b01:
        case(Fun3)
                ……
    endcase
```

Datapath with Control

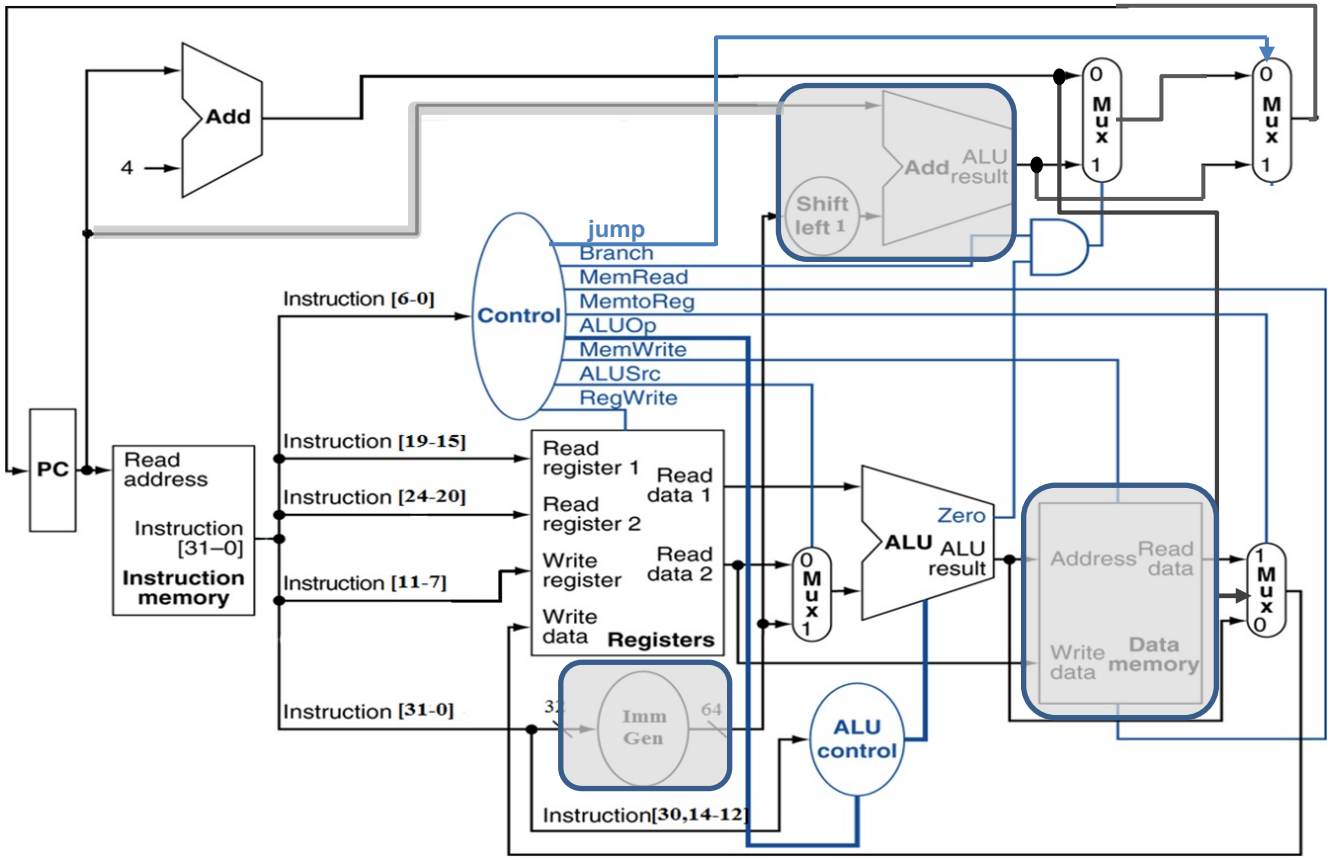


R-Type Instruction

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

add x9, x20, x21

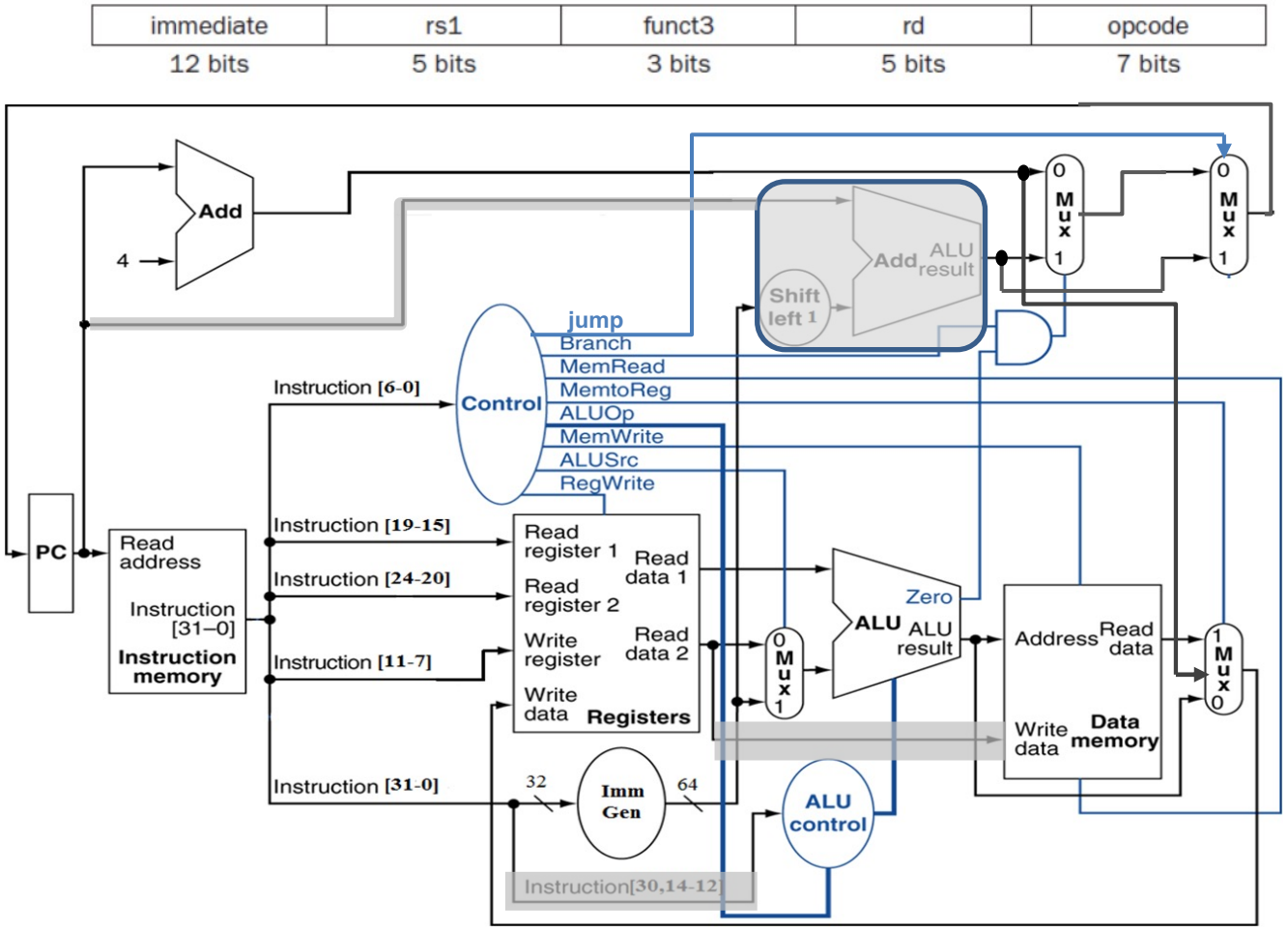
1. Read two register operands
2. Perform arithmetic/logical operation
3. Write register result



Load Instruction

ld x1, 200(x2)

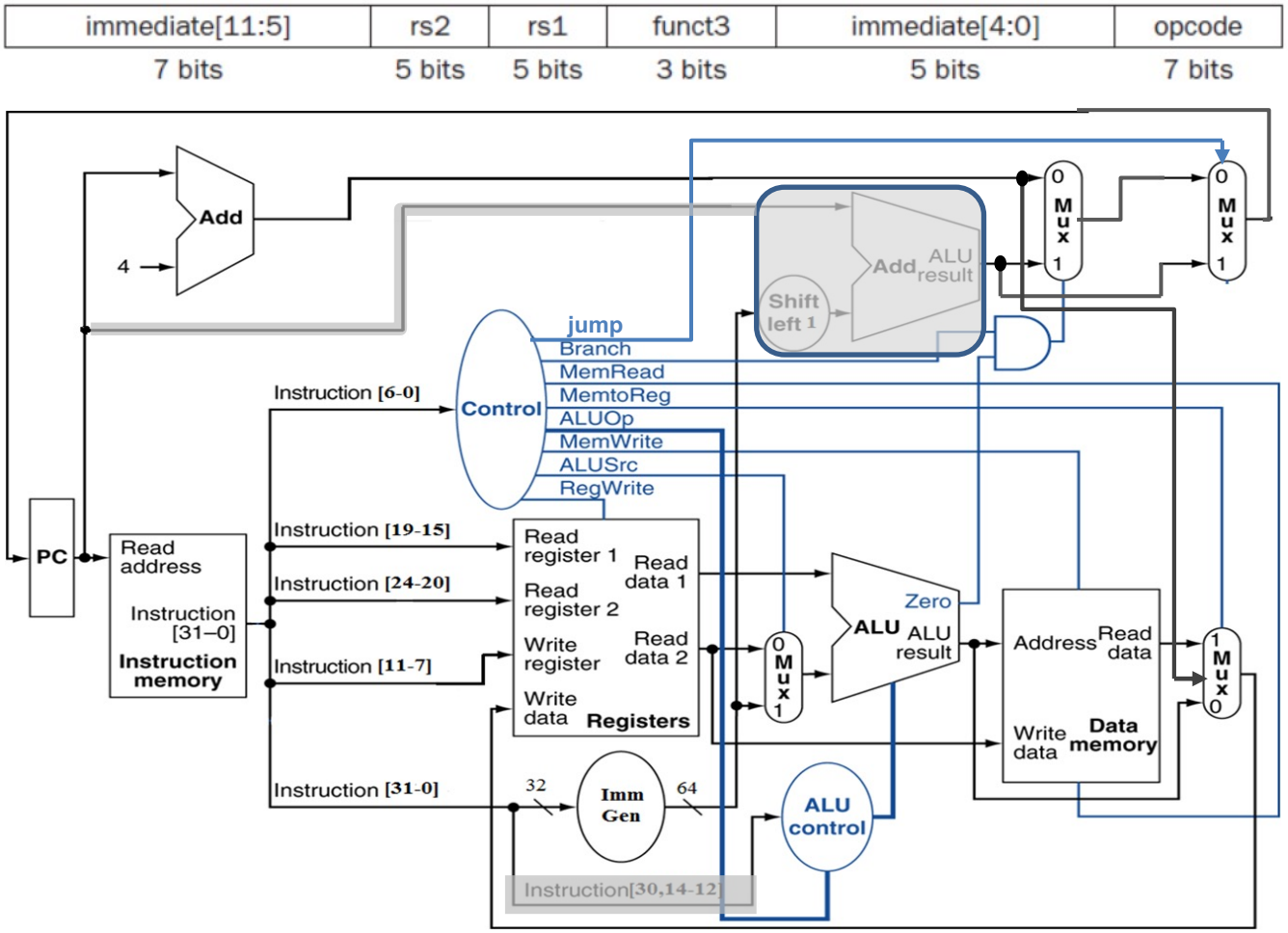
1. **Read register operands**
2. **Calculate address using 12-bit offset**
   1. Use ALU, but sign-extend offset
3. **Read memory and update register**



Store Instruction

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**

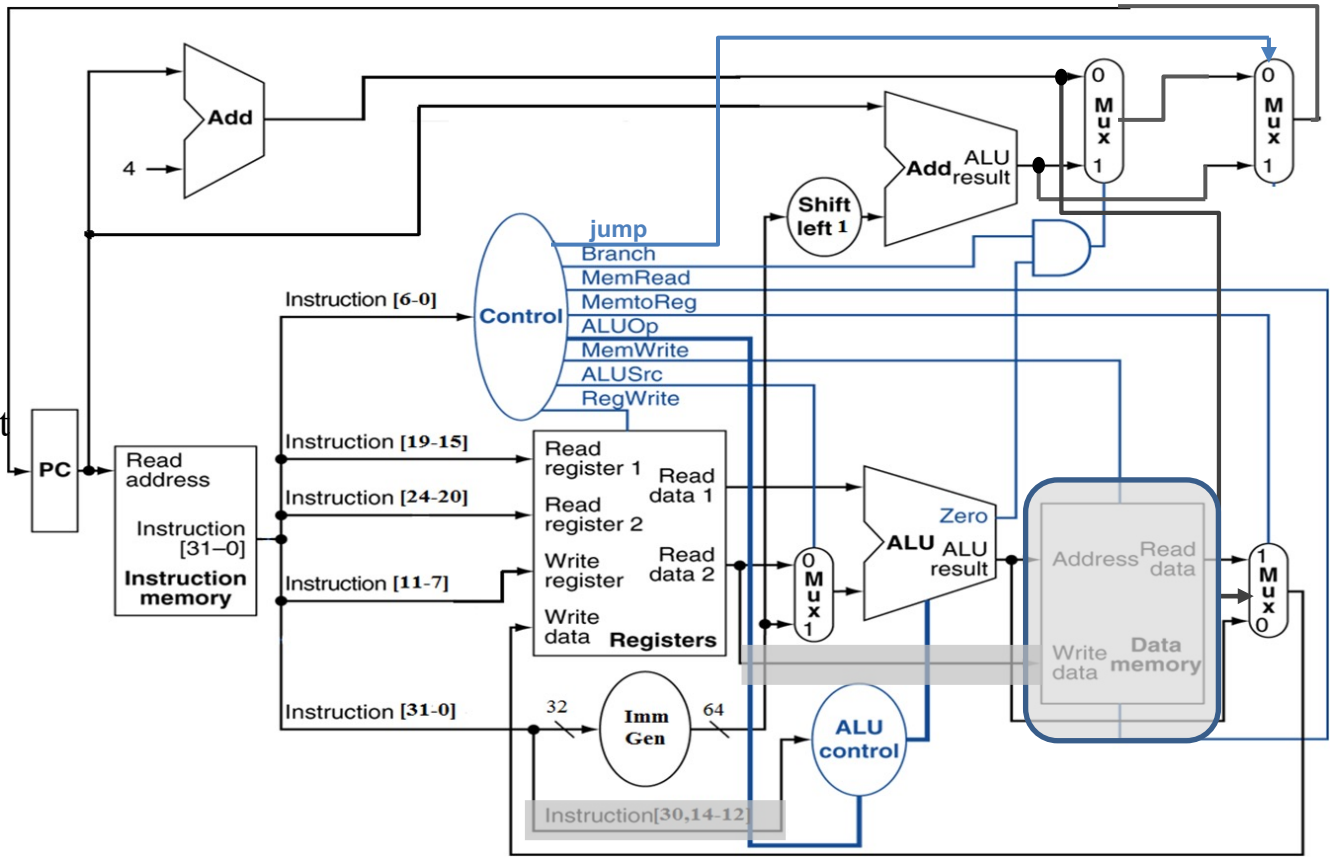


BEQ Instruction

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

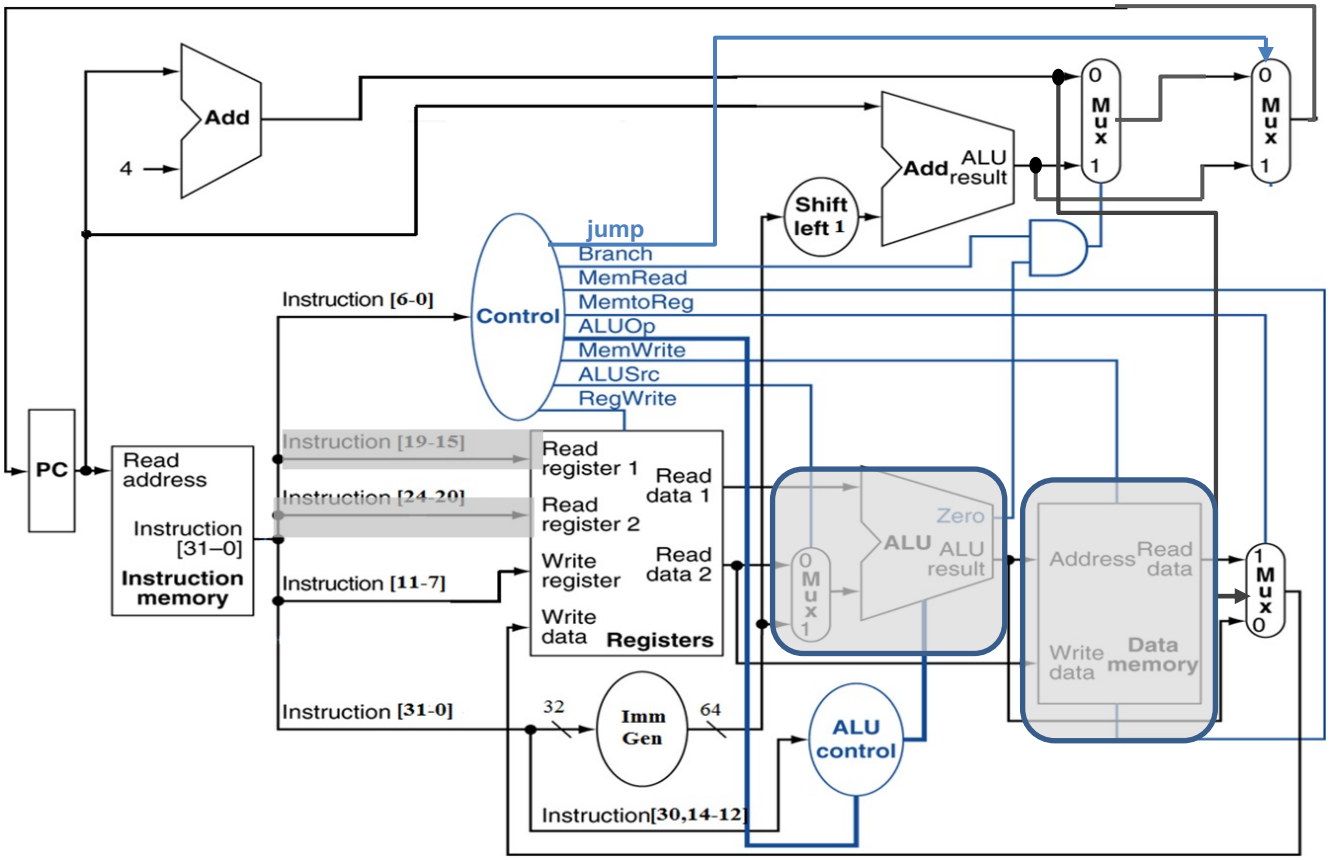


Jal Instruction

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




END