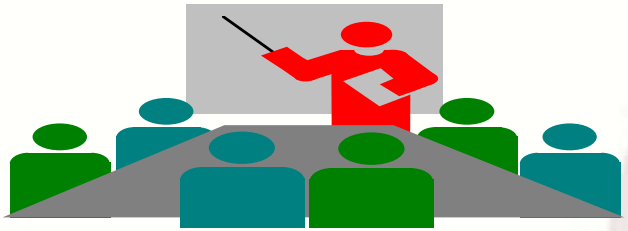


计算机组成与设计

Computer Organization & Design

The Hardware/Software Interface

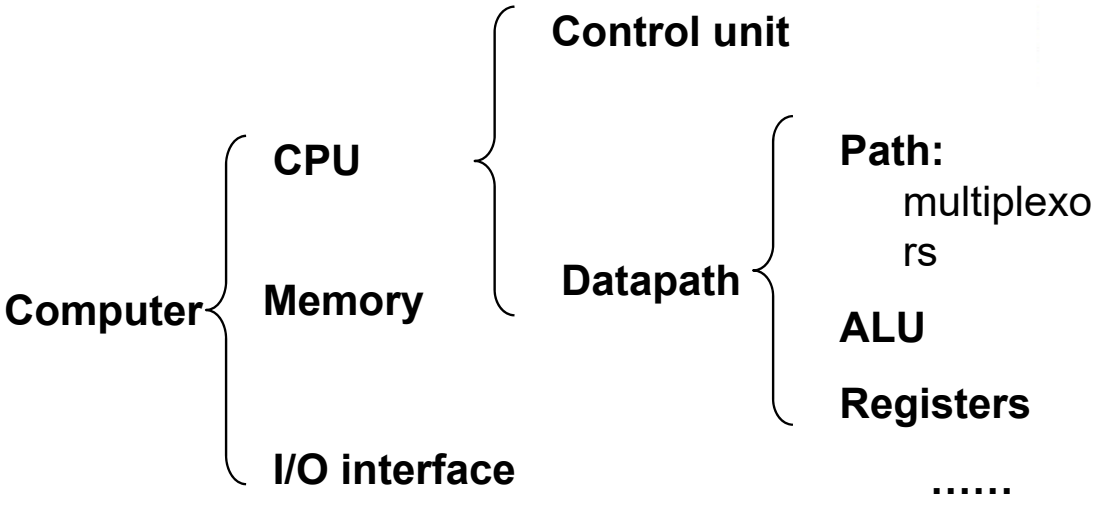
Chapter 4

The processor

Part 1

Haifeng Liu
College of Computer Science and Technology, Zhejiang University
haifengliu@zju.edu.cn

Computer Organization



Contents

- **Introduction & Logic Design Conventions**
- Building a datapath
- A Simple Implementation Scheme
- Pipelining

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, slt
  - Control transfer: beq，jal

- 实现不少于下列指令

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

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

S-Type： sw；

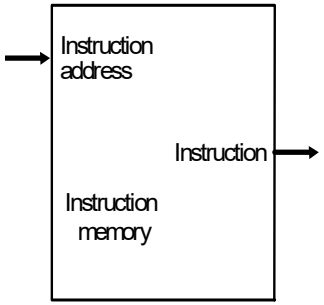
B-Type： beq；

J-Type： Jal；

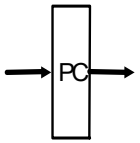
Instruction Execution Overview

- **For every instruction, the first two steps 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

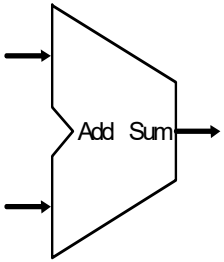
Simple Implementation



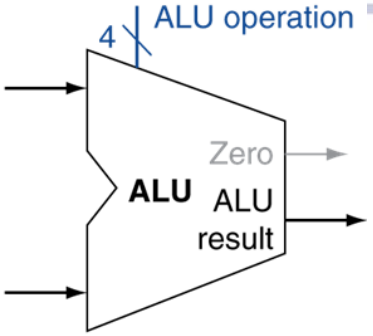
a. Instruction memory

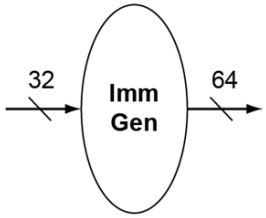


b. Program counter

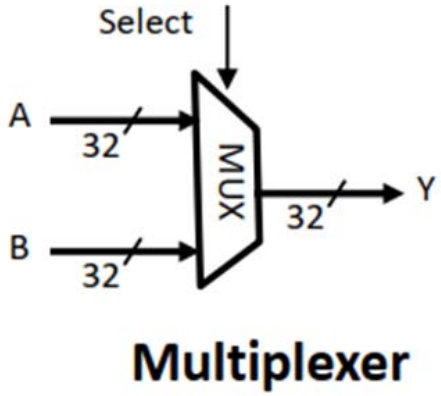


c. Adder

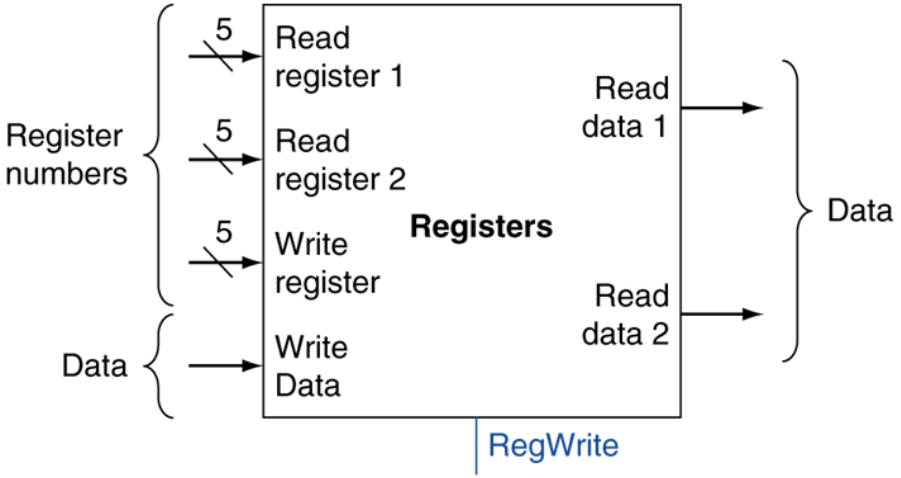




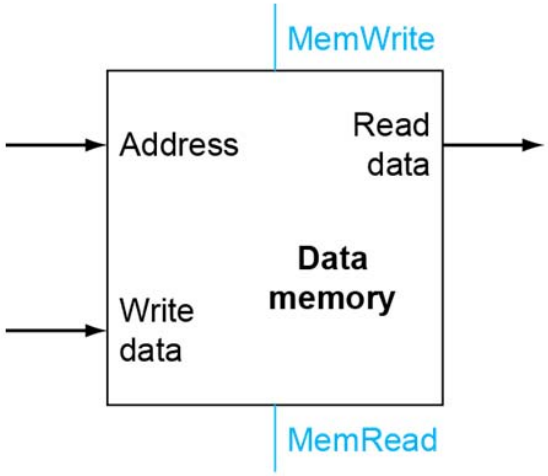
b. Immediate generation unit



Multiplexer



a. Registers



a. Data memory unit

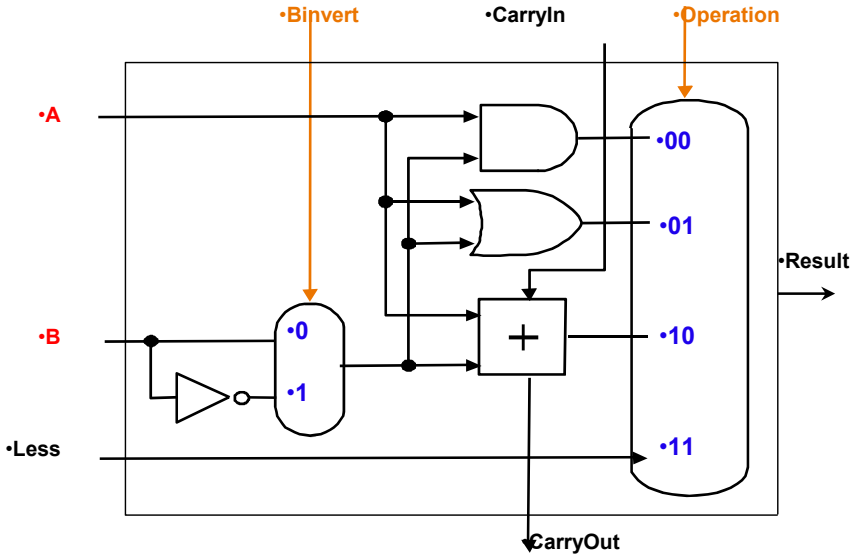
ALU

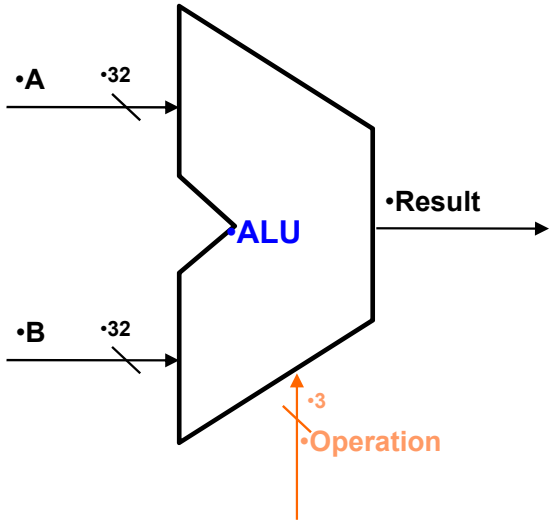
- 算术逻辑运算器ALU：即运算器
  - 5 Operations
  - “Set on less than”：

    if A<B then Result=1；

    else Result=0。

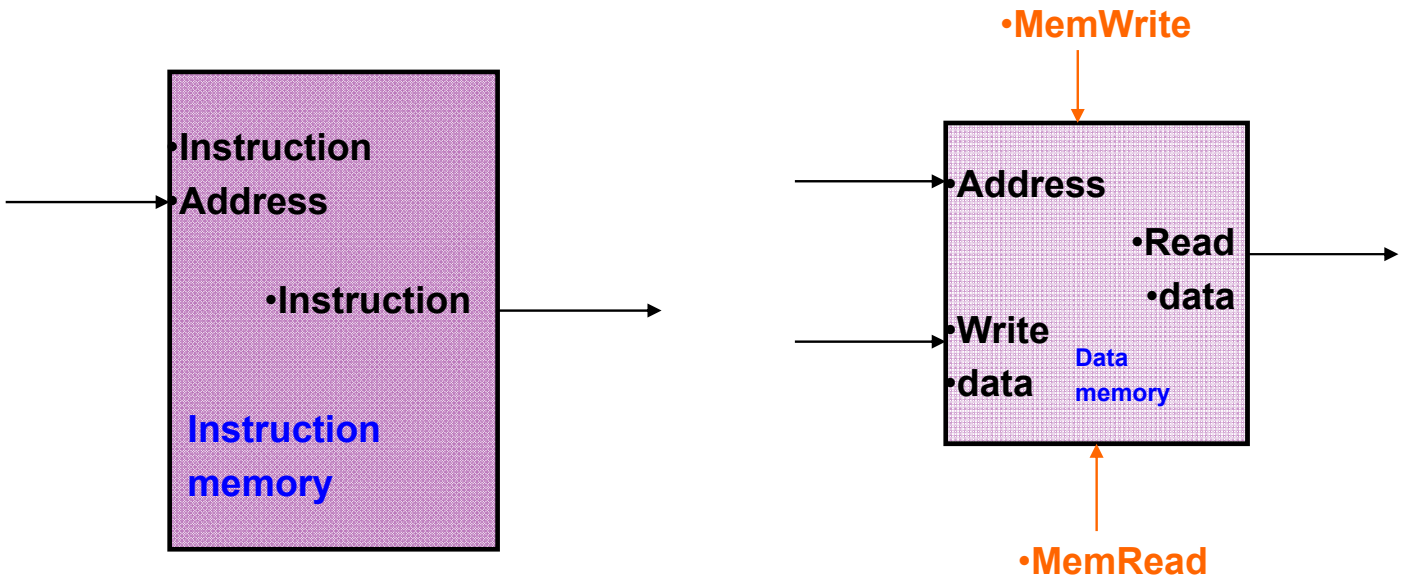
Operation	Function
000	And
001	Or
010	Add
110	Sub
111	Slt





Memory

- 存储器：
  - 可分为指令存储器与数据存储器；
  - 指令存储器设为只读；输入指令地址，输出指令。
  - 数据存储器可以读写，由MemRead和MemWrite控制。按地址读出数据输出，或将写数据写入地址所指存储器单元。



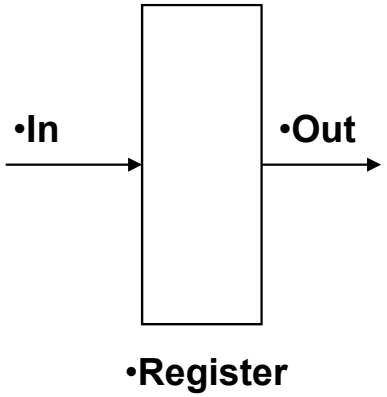
REGISTER

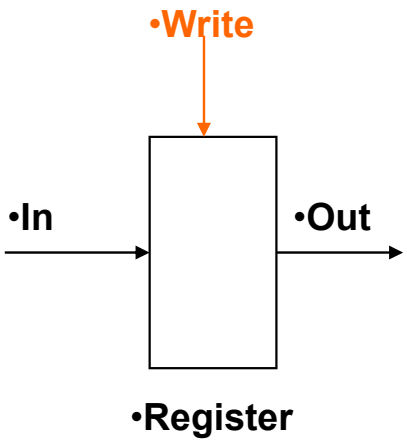
Register

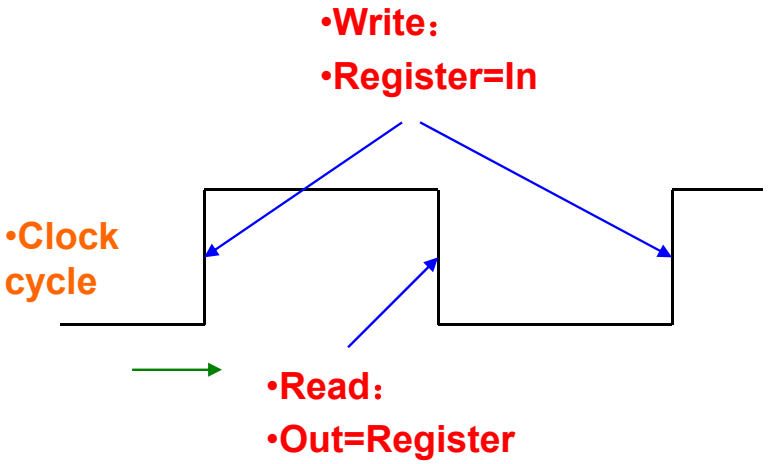
- State element。
- Can be controlled by Write signal.

  置0，数据输出保持原状态不变

  置1，在有效时钟边沿到来，数据输出为数据输入值

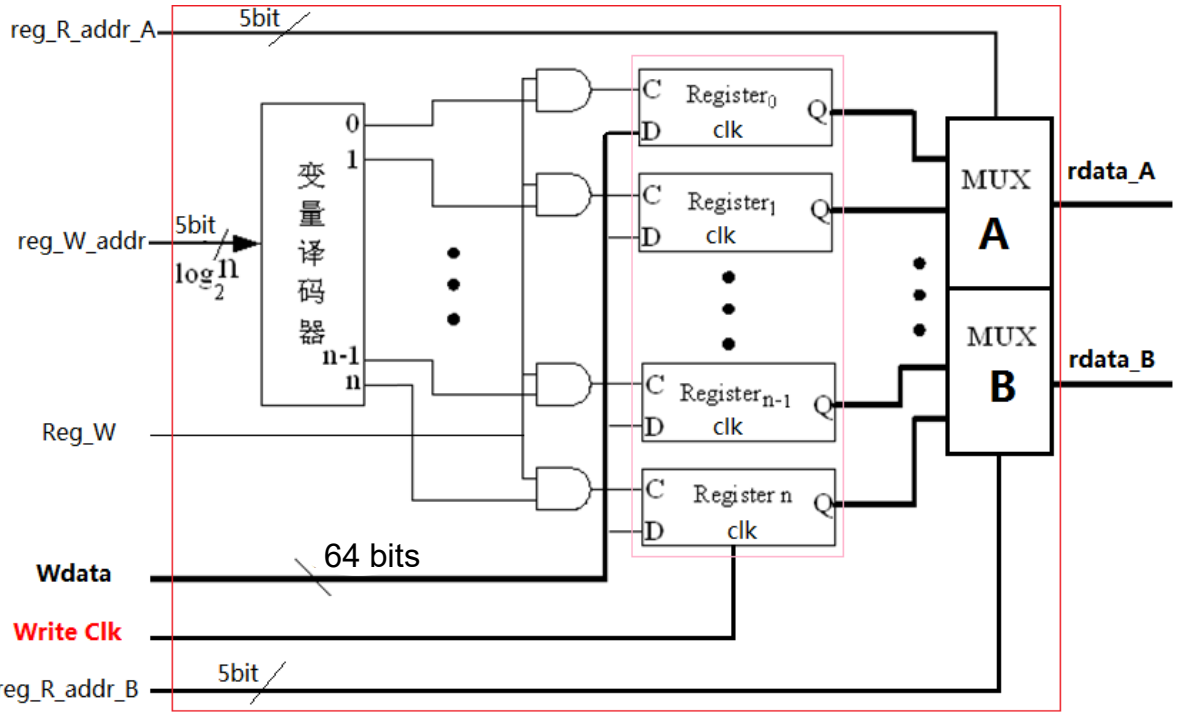






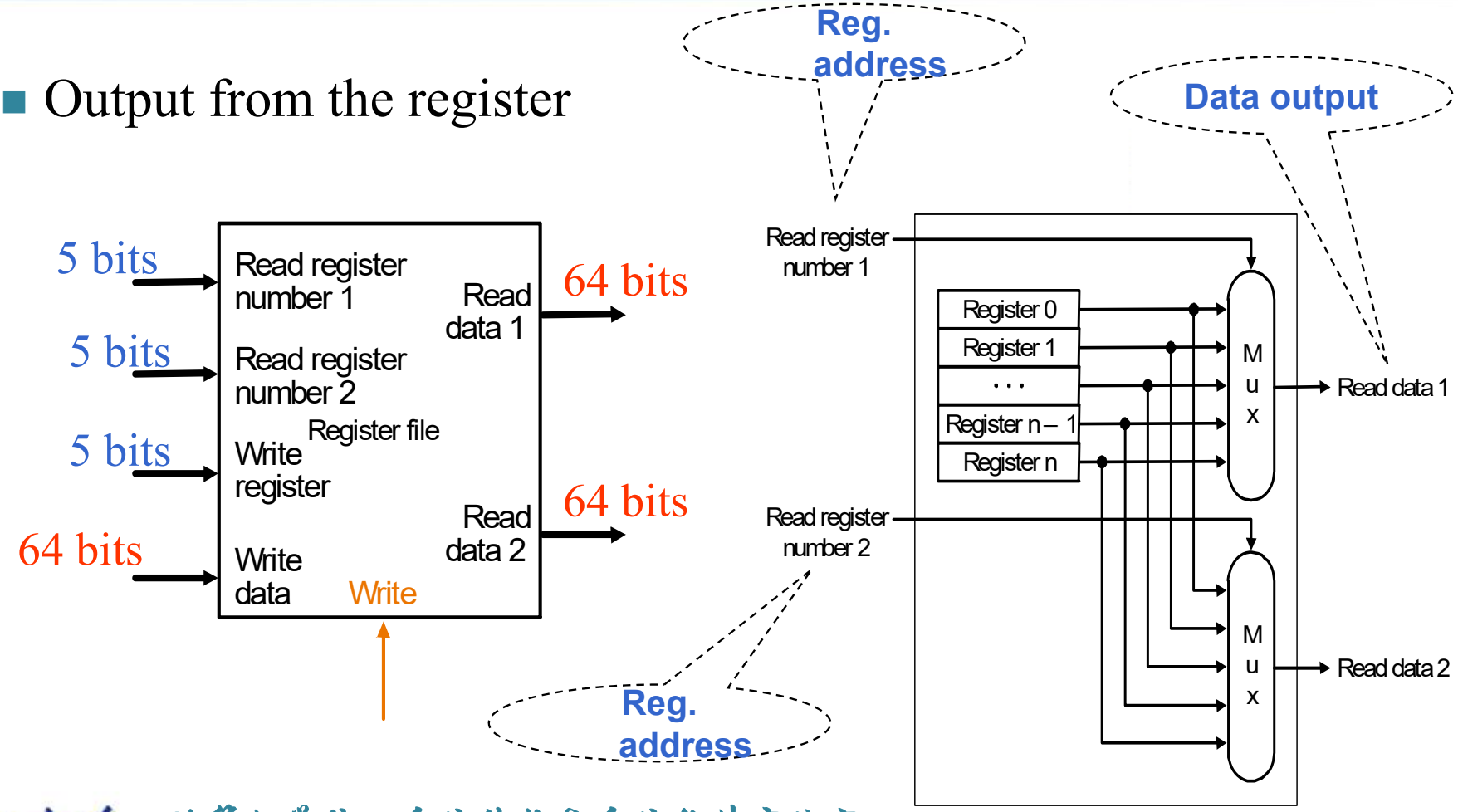
Register Files--Built using D flip-flops

- 32 64-bit Registers；
- Input: 2 5-bit register number/ one 5-bit register number and 64-bit data;
- Output: 64-bit data;
- Register write control。



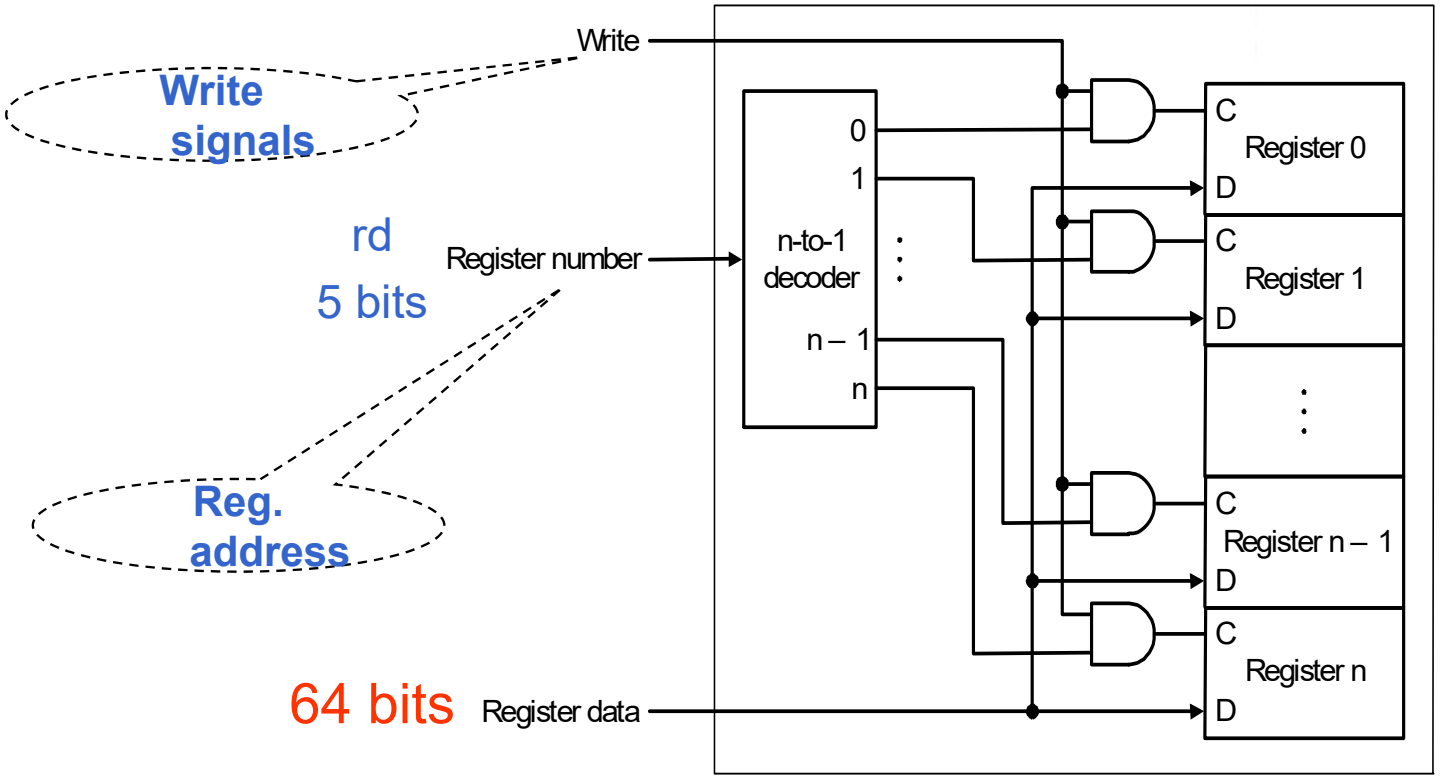
Register File: Read-Output

- Output from the register



Register File: Write-Input

- Written to the register



Description: 32×64bits Register files

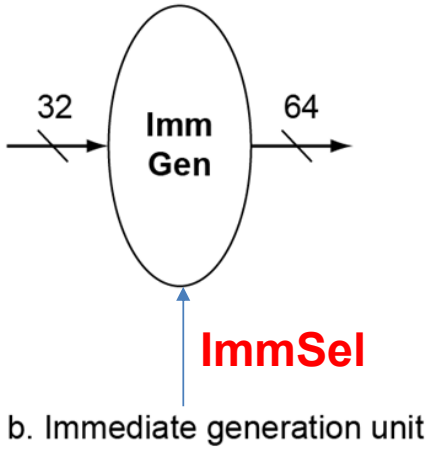
```
Module  regs( input              clk,  rst,  RegWrite,
              input    [4:0]    Rs1_addr, Rs2_addr, Wt_addr,
              input    [63:0]  Wt_data,
              output [63:0]  Rs1_data,  Rs2_data
            );
        reg [63:0] register [1:31];                    // r1 - r31
        integer i;

        assign rdata_A = (Rs1_addr== 0) ? 0 : register[Rs1_addr];              // read
        assign rdata_B = (Rs2_addr== 0) ? 0 : register[Rs2_addr];                 // read

        always @(posedge clk or posedge rst)
        begin   if (rst==1)
                        for (i=1; i<32; i=i+1)  register[i] <= 0;             // reset
                else if ((Wt_addr != 0) && (RegWrite == 1))
                        register[Wt_addr] <= Wt_data;                                 // write
        end
endmodule
```

The other elements

- Immediate generation unit：
  - 输入指令产生立即数的逻辑功能
    - 根据指令类型（加载，存储或者分支指令），产生相应的立即数
  - 转移指令偏移量左移位的功能
    - 立即数字段符号扩展为64位结果输出



b. Immediate generation unit

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

Immediate generation

- Load: 0000011

  L_imm = {{52{inst[31]}}, Inst[31:20]};

- Save: 0100011

  S_imm = {{52{inst[31]}}, Inst[31:25], inst[11:7]};

- Branch: 1100011

  SB_imm = {{51{inst [31]}}, inst[31], inst [7], inst[30:25], inst[11:8],1'b0};

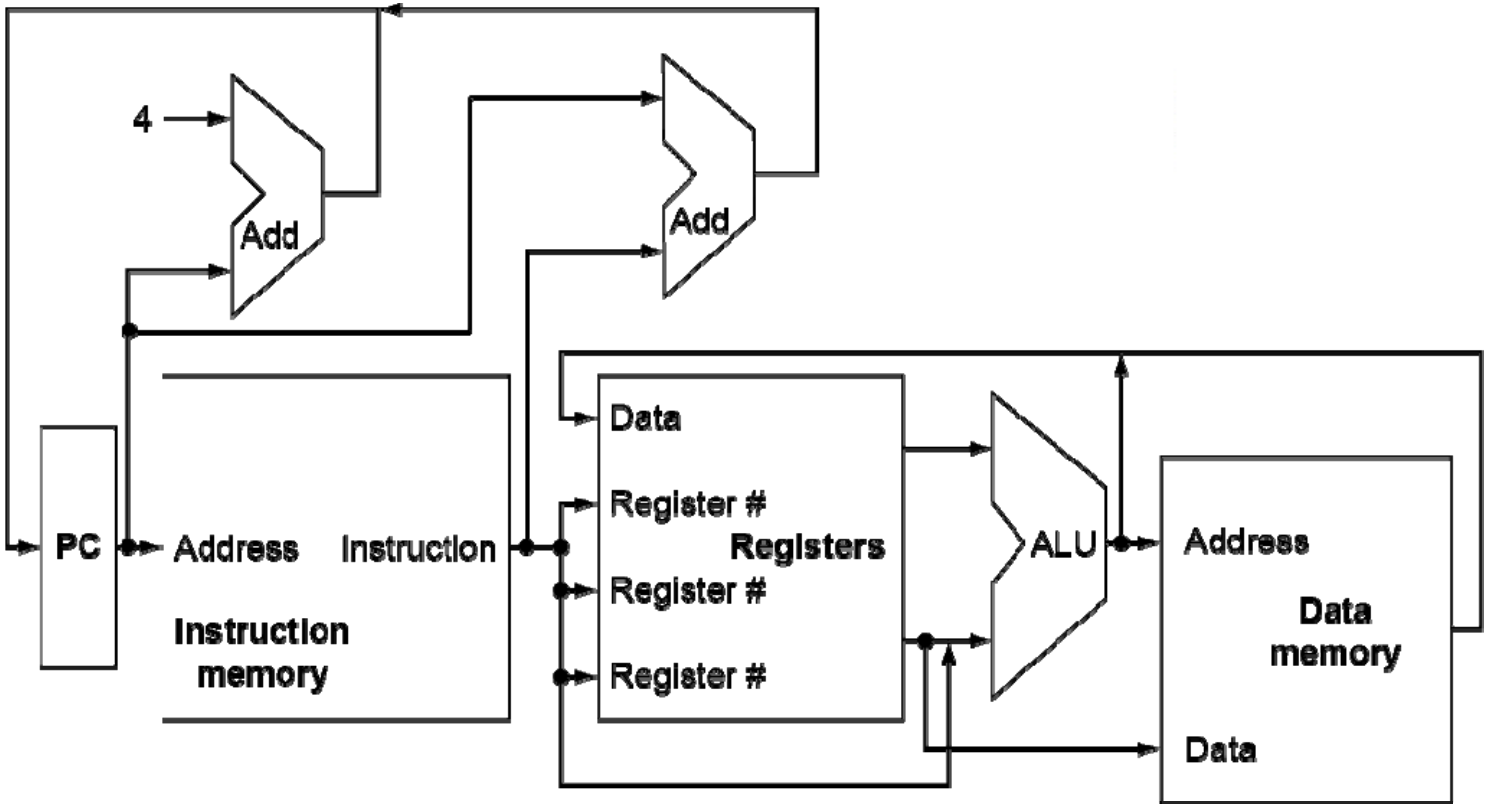
- Jal: 1101111

  UJ = {{43{inst[31]}}, inst [31], inst[19:12], inst[20], inst[30:21],1'b0};

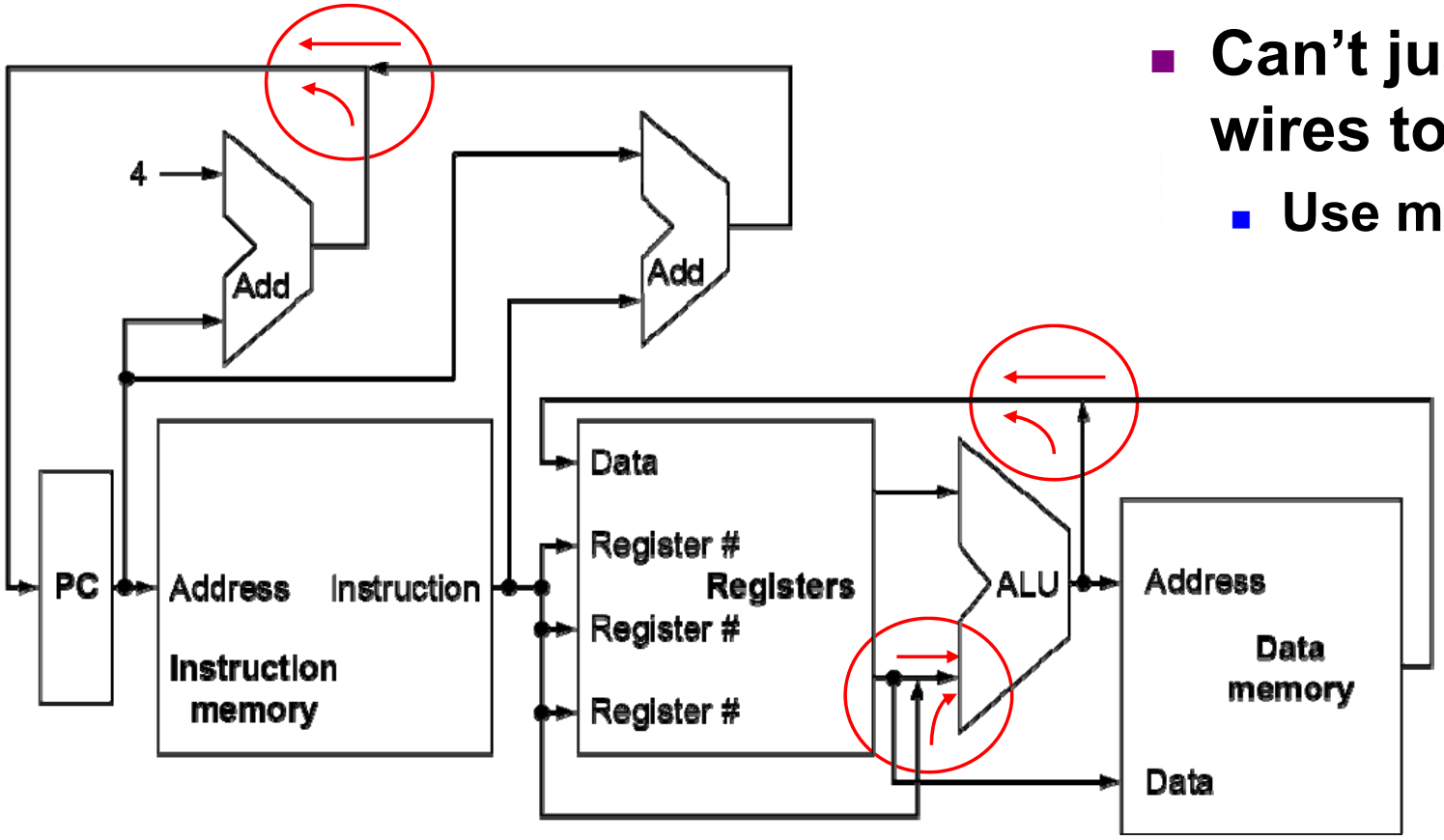
ImmSel

Instruction type	Instruction opcode[6:0]	Instruction operation	(sign-extend)immediate	ImmSel
I-type	0000011	Lw;lbu;lh; lb;lhu	(sign-extend) instr[31:20]	00
	0010011	Addi;slti;sltiu;xori;ori; andi;		
	1100111	jalr		
S-type	0100011	Sw;sb;sh	(sign-extend) instr[31:25],[11:7]	01
B-type	1100011	Beq;bne;blt;bge;bltu; bgeu	(sign-extend) instr[31],[7],[30:25],[11:8],1'b0	10
J-type	1101111	jal	(sign-extend) instr[31],[19:12],[20],[30:21],1'b0	11

CPU Overview

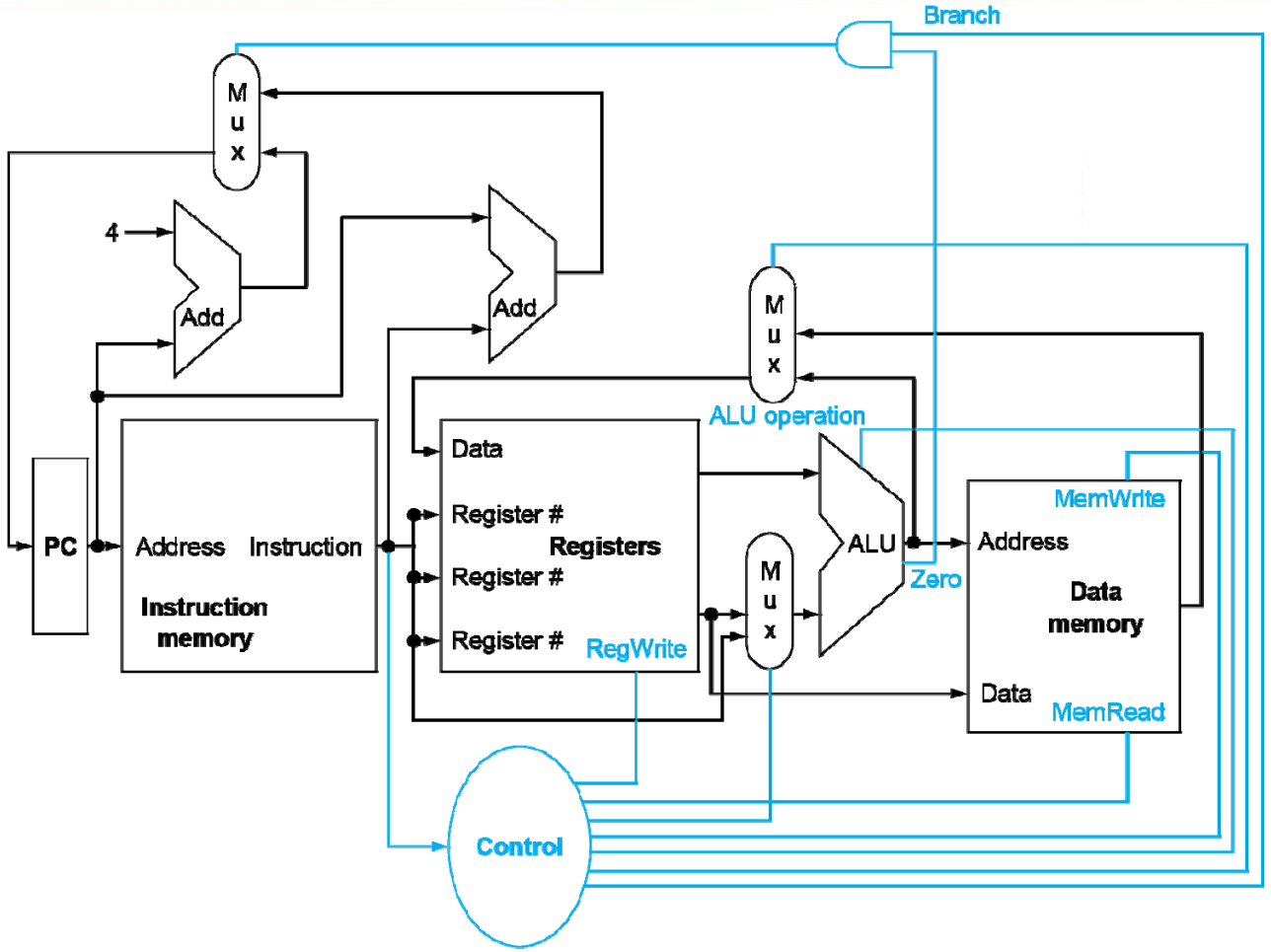


Multiplexers



- Can't just join wires together
  - Use multiplexers

Control

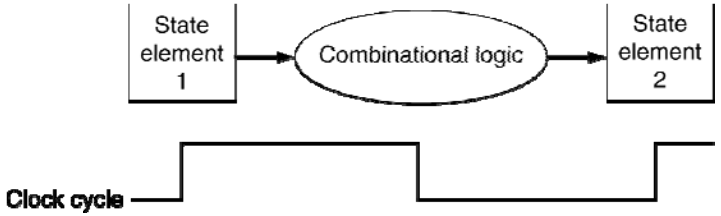

Branch
M
u
x
4
Add
Add
M
u
x
ALU operation
PC
Address
Instruction
Instruction
memory
Data
Register #
Registers
Register #
Register #
RegWrite
M
u
x
ALU
Zero
MemWrite
Address
Data
memory
Data
MemRead
Control

Logic Design Conventions

- **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

Clocking Methodology

- An edge triggered methodology
- Typical execution:
  - read contents of some state elements,
  - send values through some combinational logic
  - write results to one or more state elements



Contents

- Introduction & Logic Design Conventions
- **Building a datapath**
- A Simple Implementation Scheme
- Pipelining
- Exceptions

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 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
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*

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.

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

R-type Instructions	funct7	rs2	rs1	funct3	rd	opcode	Example
add (Add)	0000000	00011	00010	000	00001	0110011	add x1,x2,x3
sub (Sub)	010000	00011	00010	000	00001	0110011	sub x1,x2,x3
I-type Instructions	**immediate**		**rs1**	**funct3**	**rd**	**opcode**	**Example**
addi (Add Immediate)	001111101000		00010	000	00001	0010011	addi x1,x2, 1000
ld (Load doubleword)	001111101000		00010	011	00001	0000011	ld x1,1000 (x2)
S-type Instructions	**immed -iate**	**rs2**	**rs1**	**funct3**	**immed -iate**	**opcode**	**Example**
sd (Store doubleword)	0011111	00001	00010	011	01000	0100011	sd x1,1000(x2)

SB-type instruction	**111110**	**01011**	**01010**	**001**	**10000**	**1100011**

bne x10, x11, 2000

UJ-type instruction	**01111101000000000000**	**00000**	**1101111**

jal x0, 2000

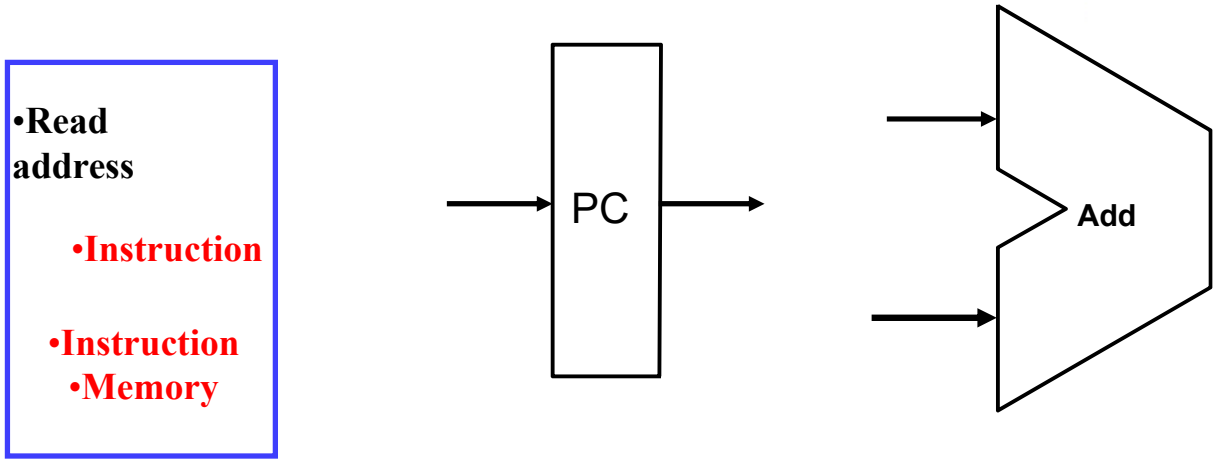
Instruction execution in RISC-V

- **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 three elements

Data Stream of Instruction fetching

How to connect? Who?

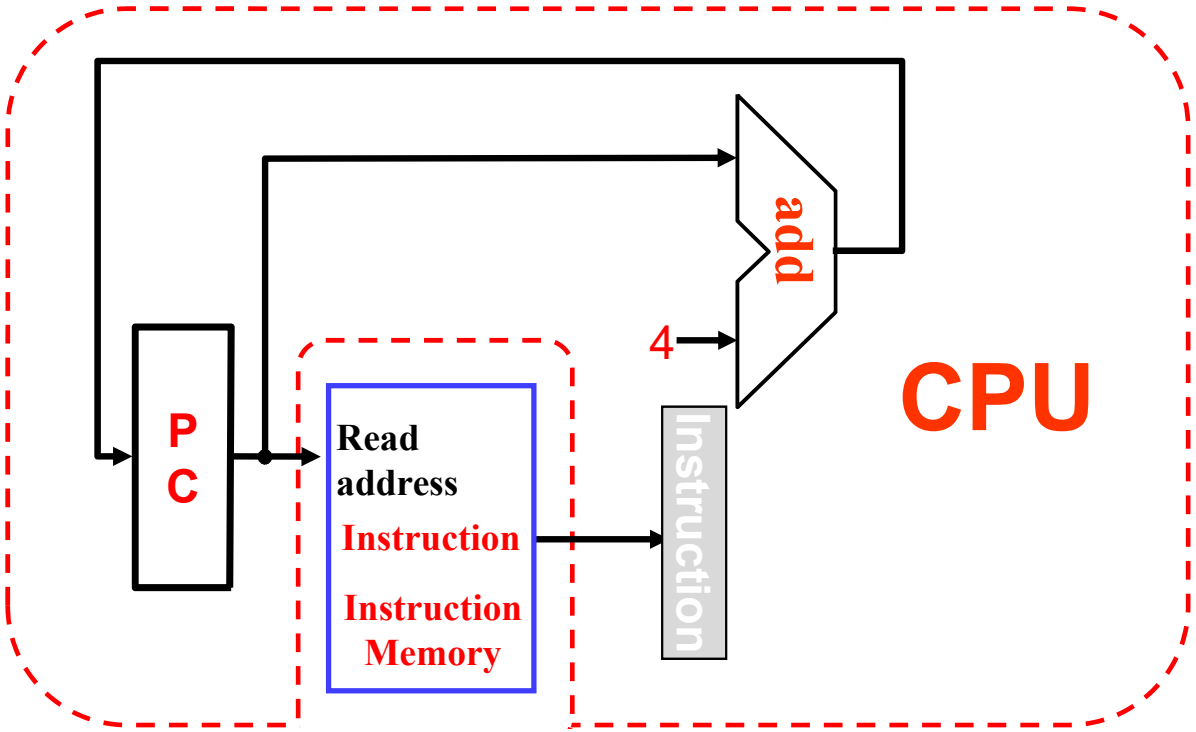


Instruction memory Program counter Adder

Instruction fetching unit

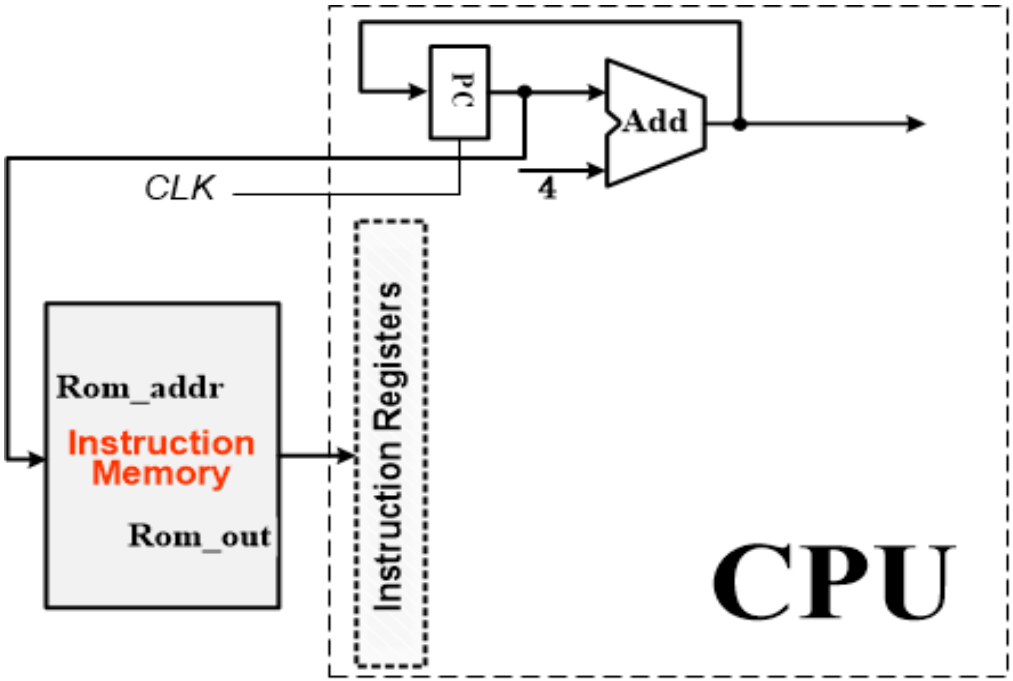
Instruction Register

- Can you omit it?



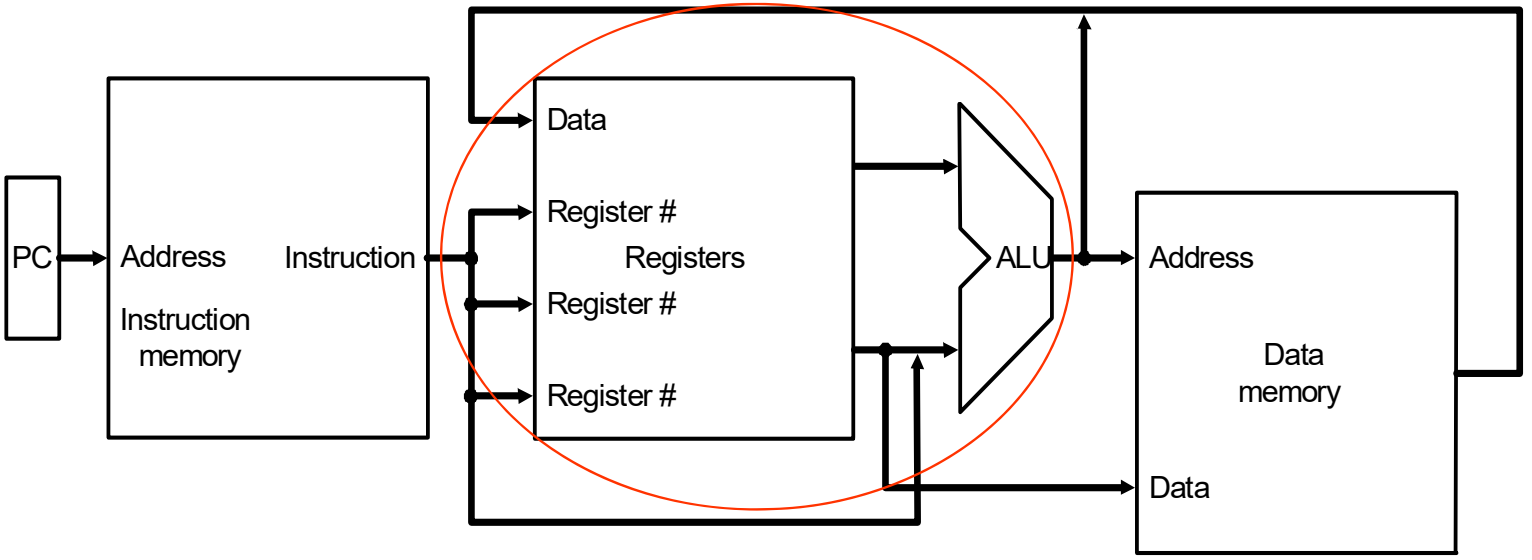
How simple is!

- Why PC+4?



More Implementation Details

Abstract / Simplified View:



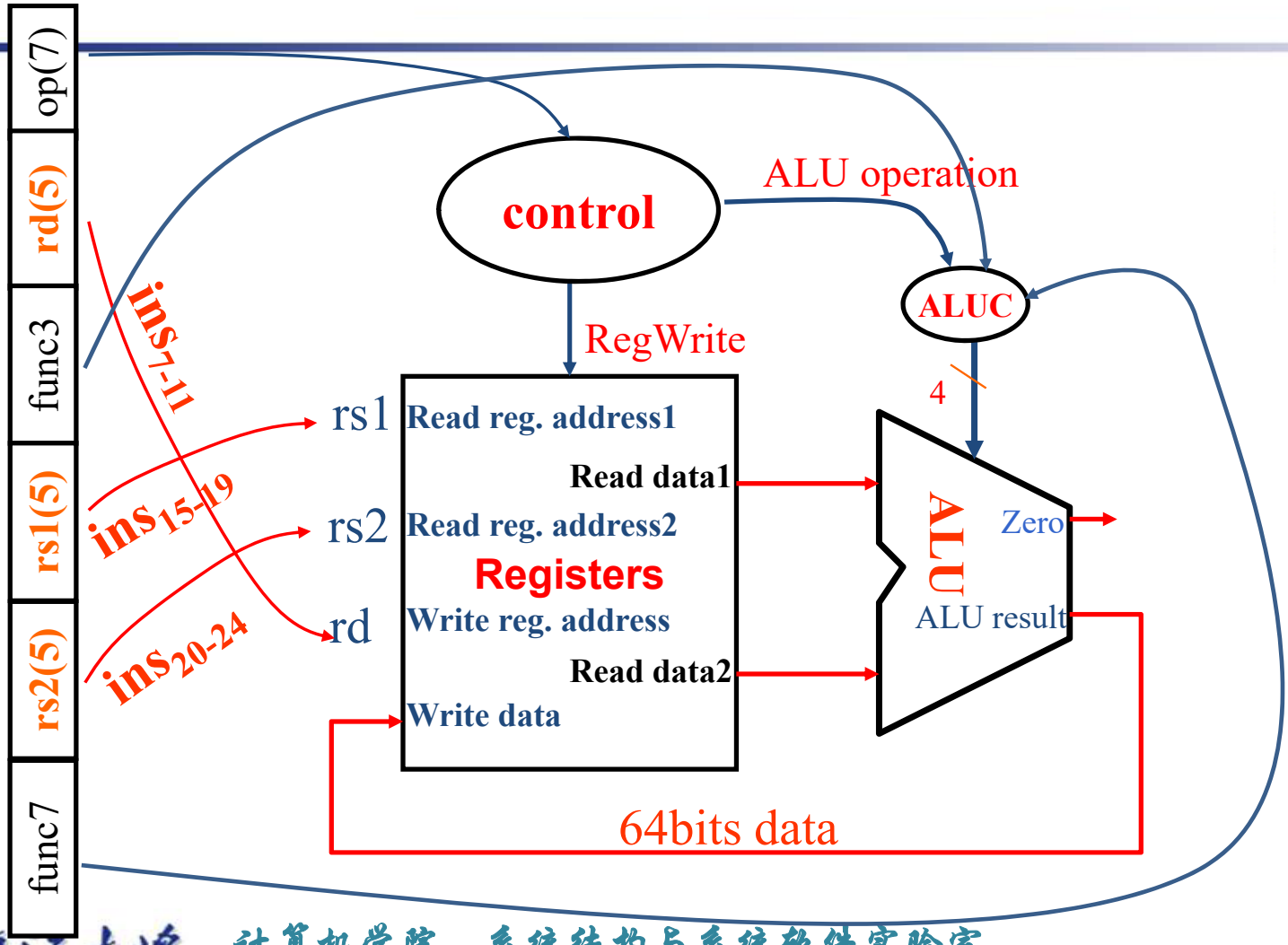
Path Built using Multiplexer

Data Stream of Instruction executing

- 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

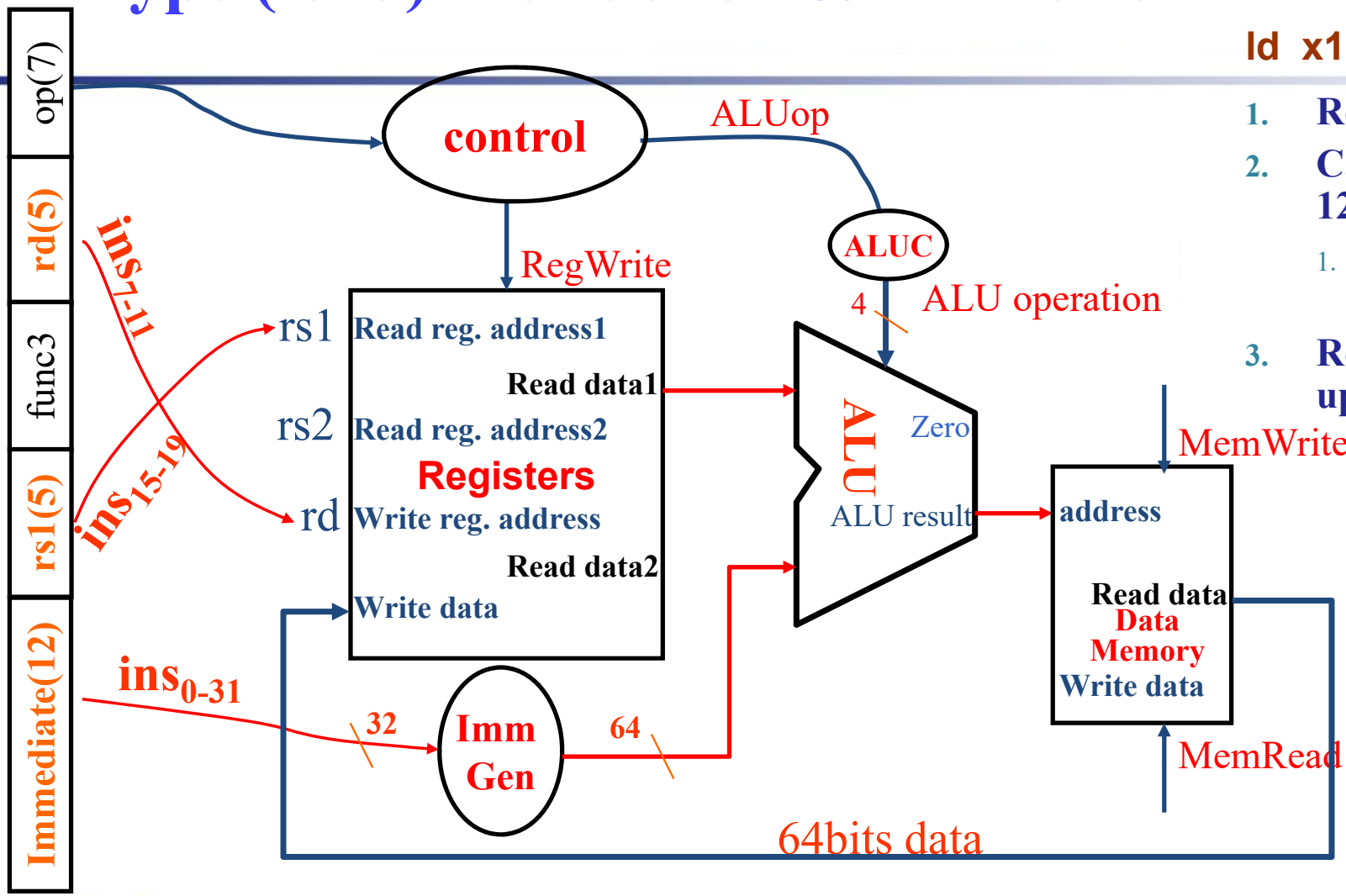
R type Instruction & Data stream



add x9, x20, x21

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

I type (load) Instruction & Data stream



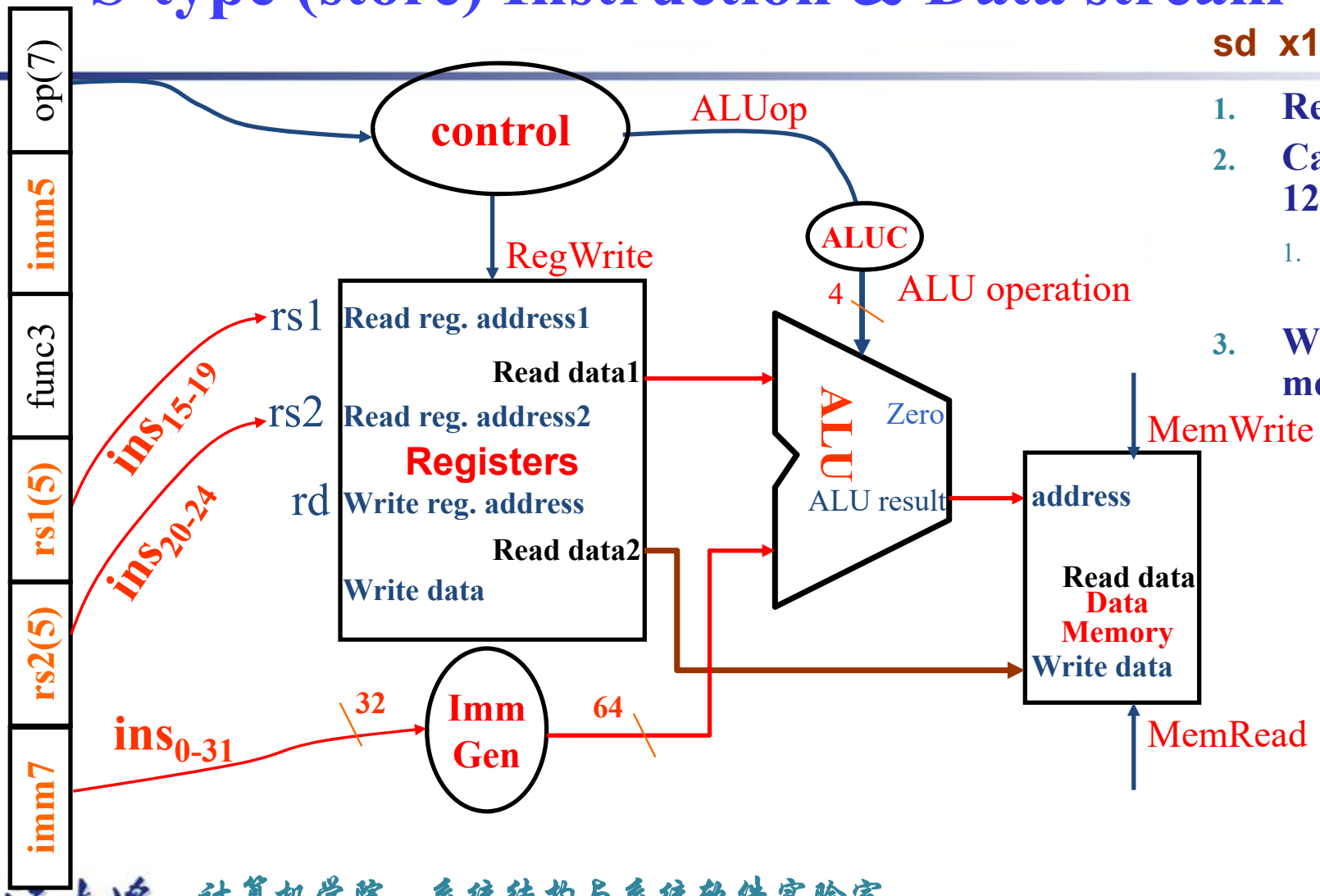
ld x1, 200(x2)

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

addi x1, x2, 4?

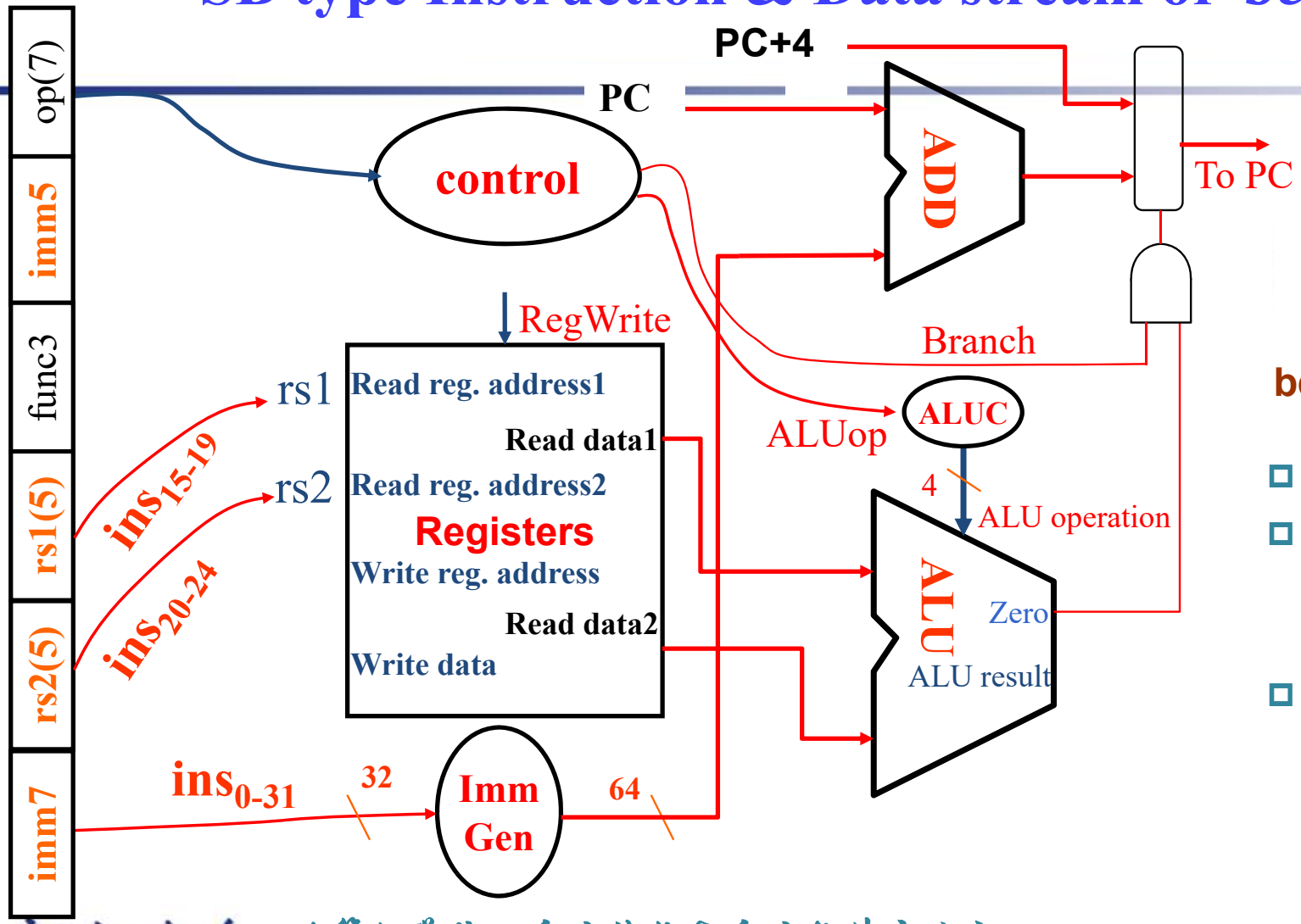
S-type (store) Instruction & Data stream

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

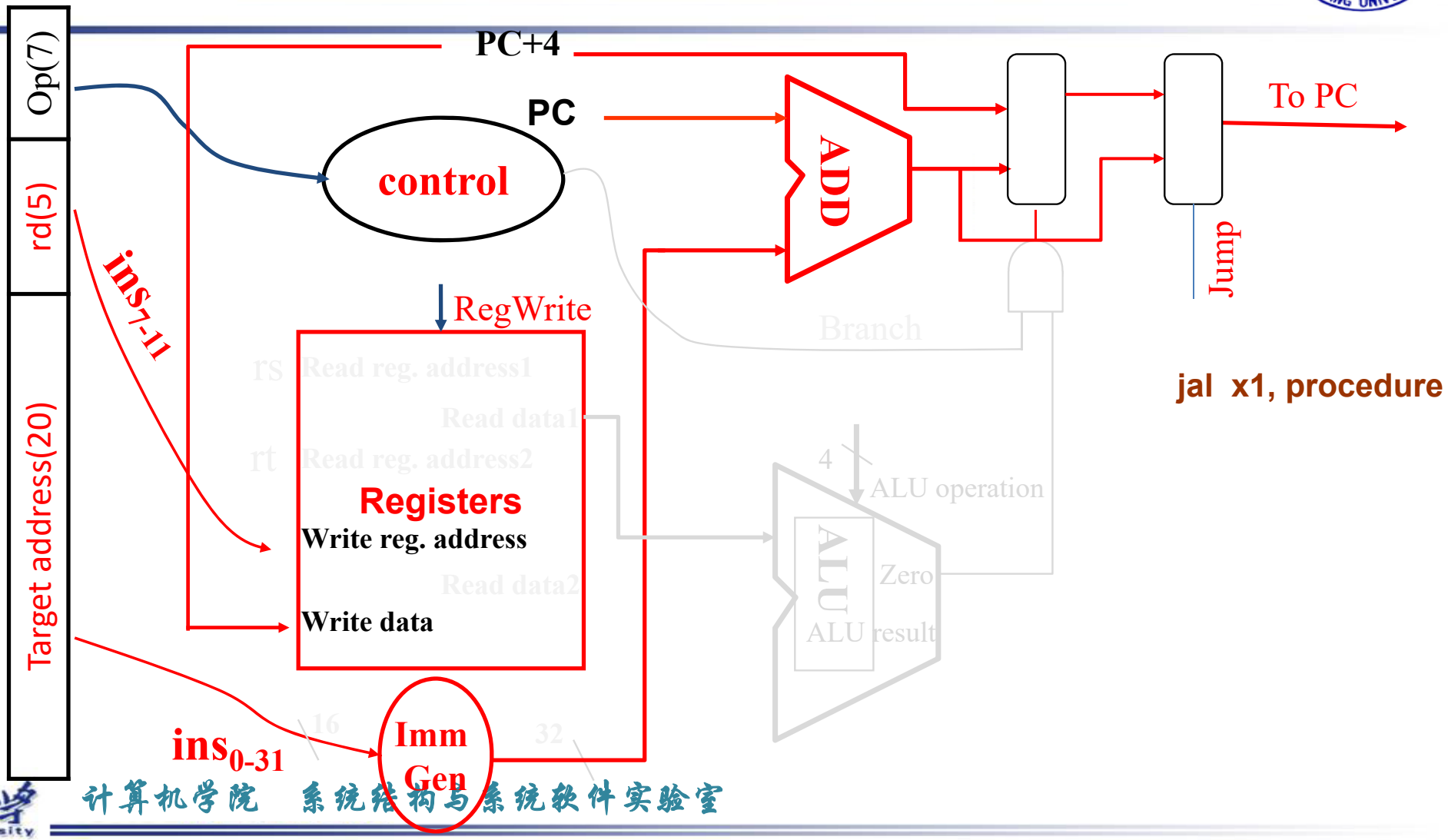
SB type Instruction & Data stream of *beq*



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

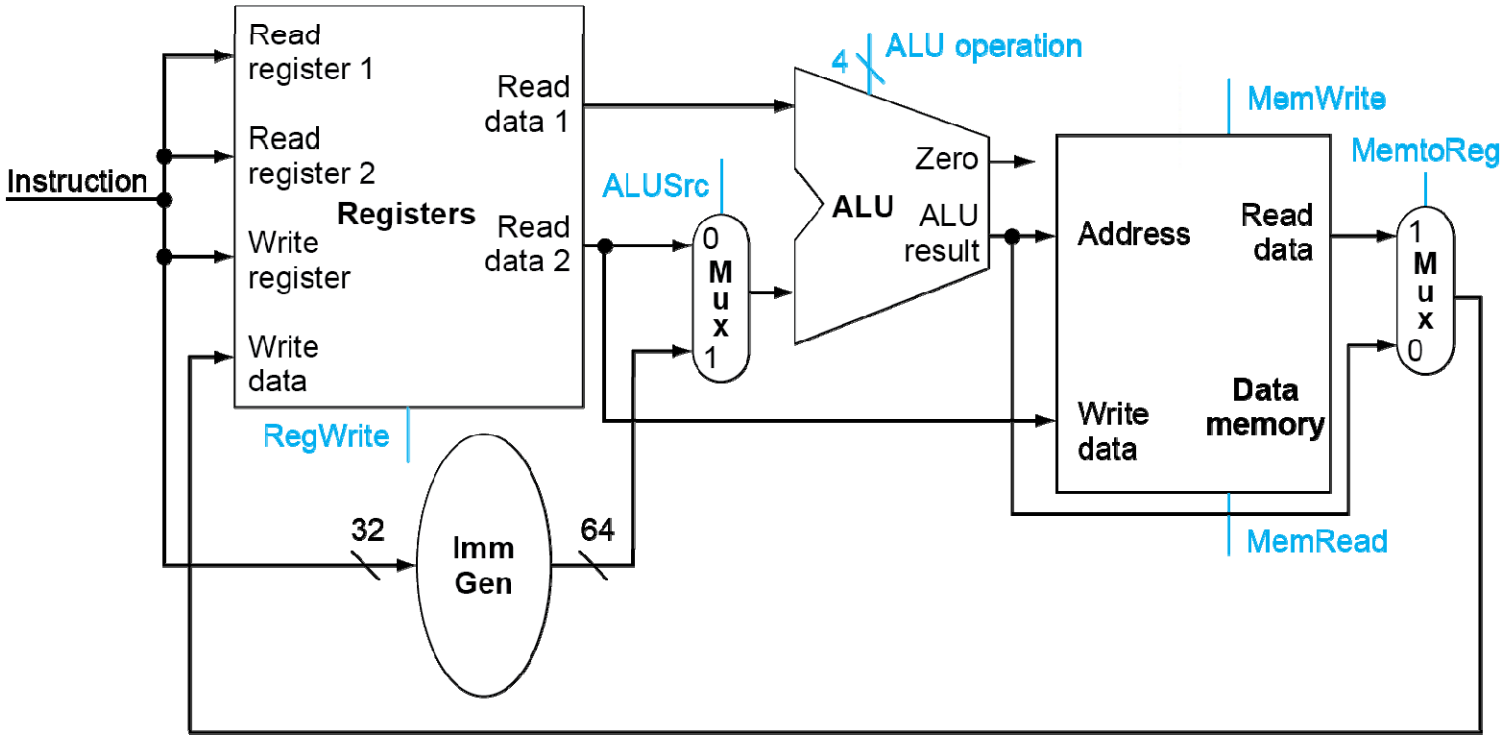
UJ type Instruction



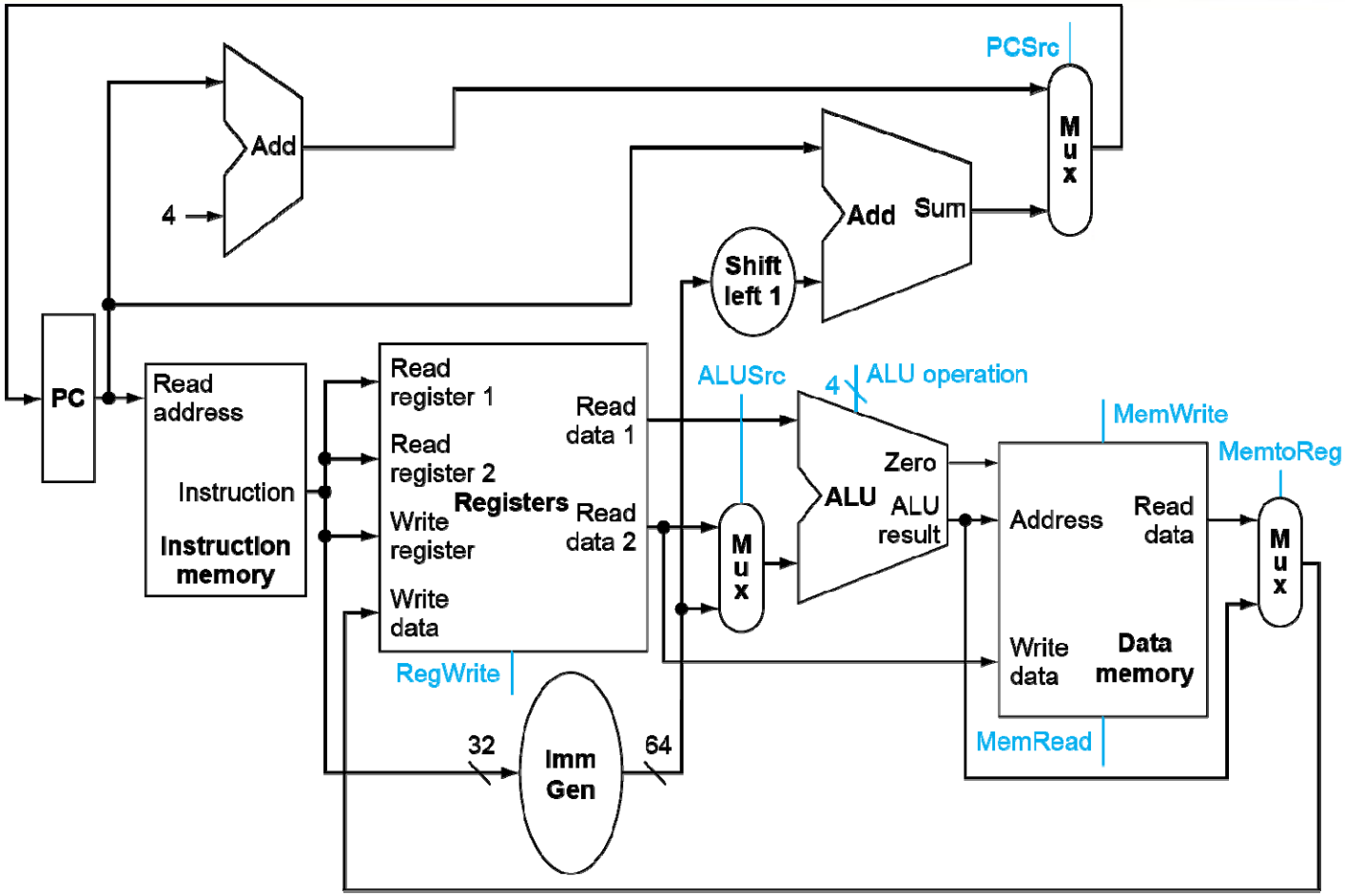
Composing the Elements

- **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**

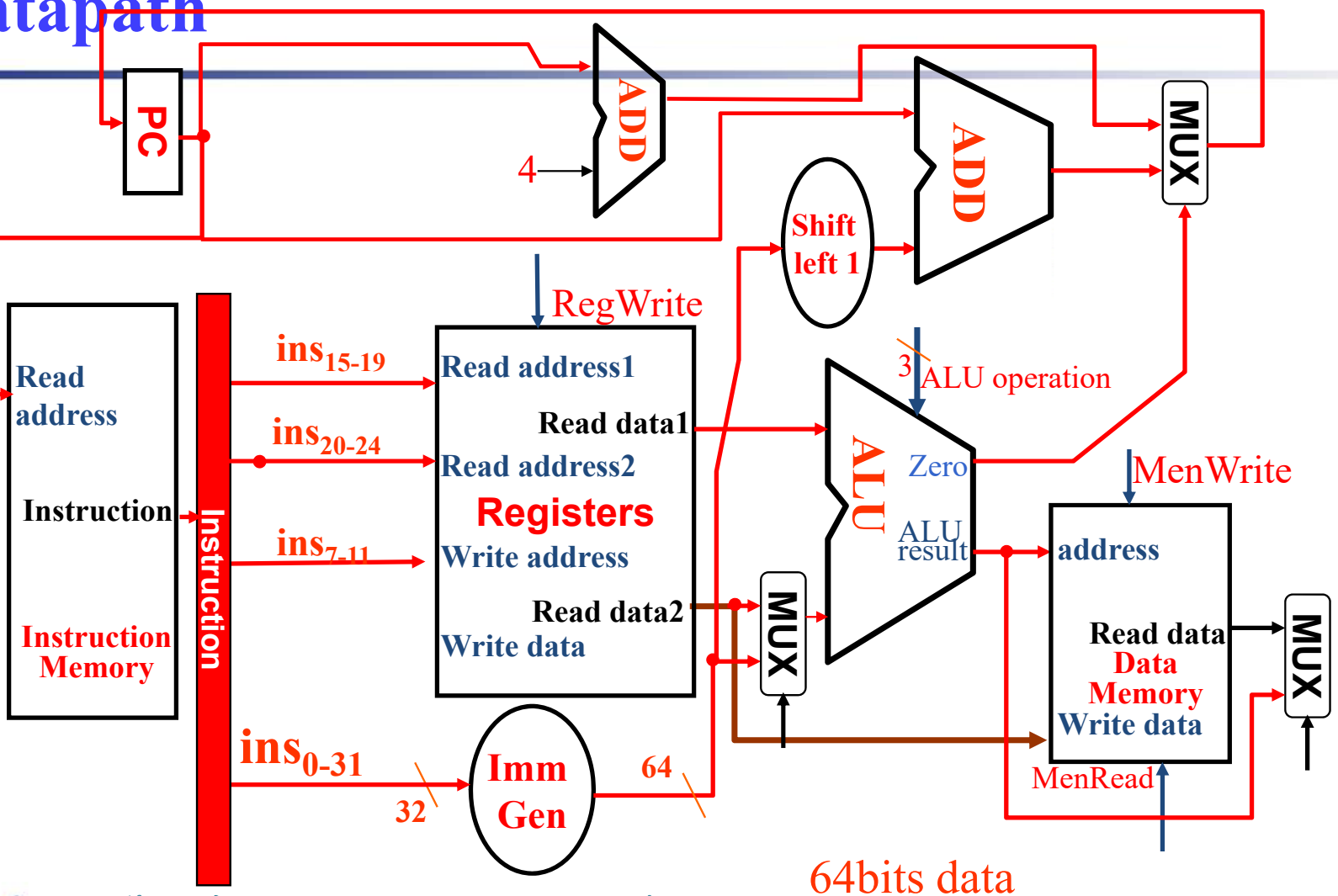
R-Type/Load/Store Datapath



Full Datapath

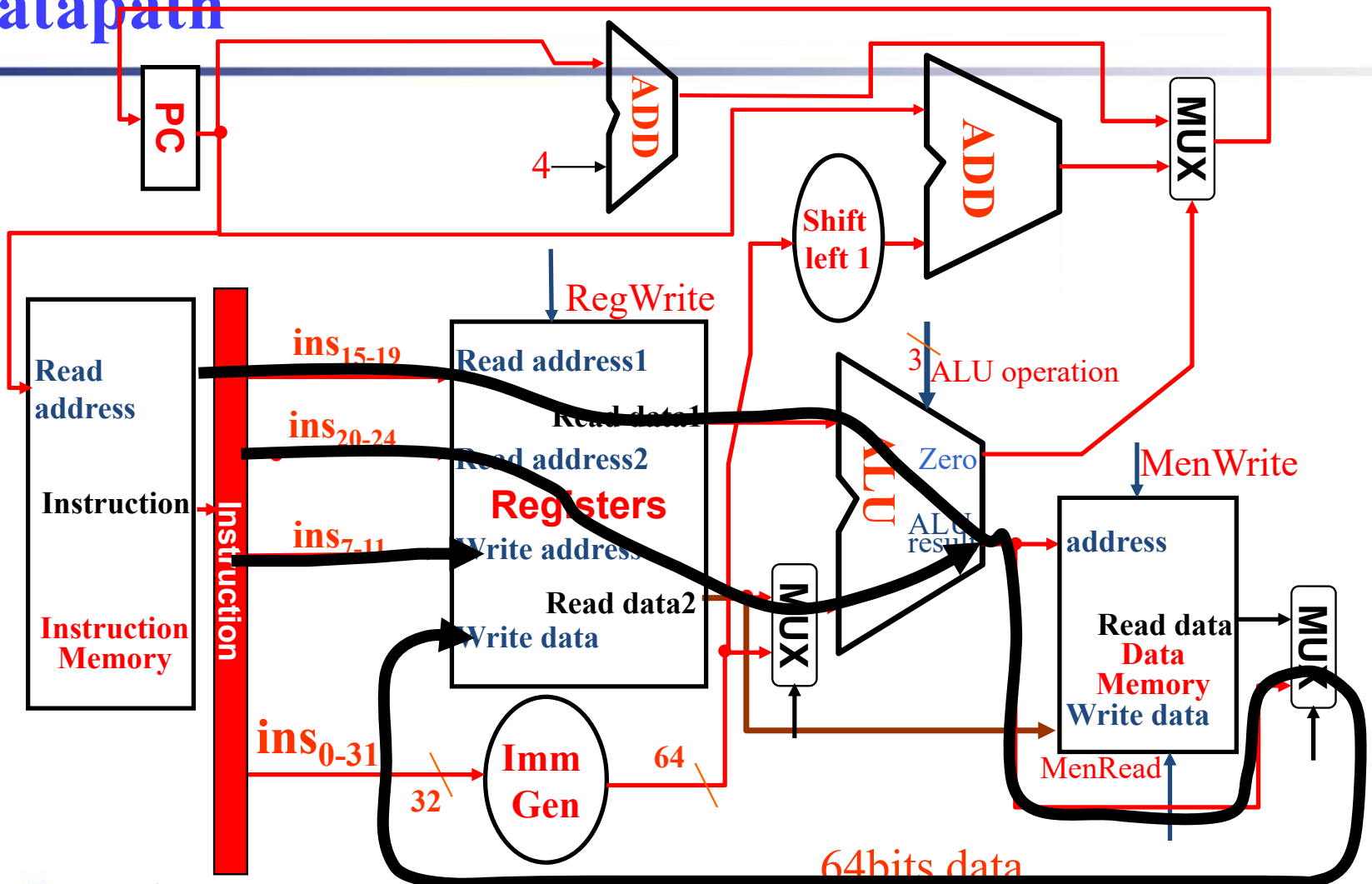


Full datapath



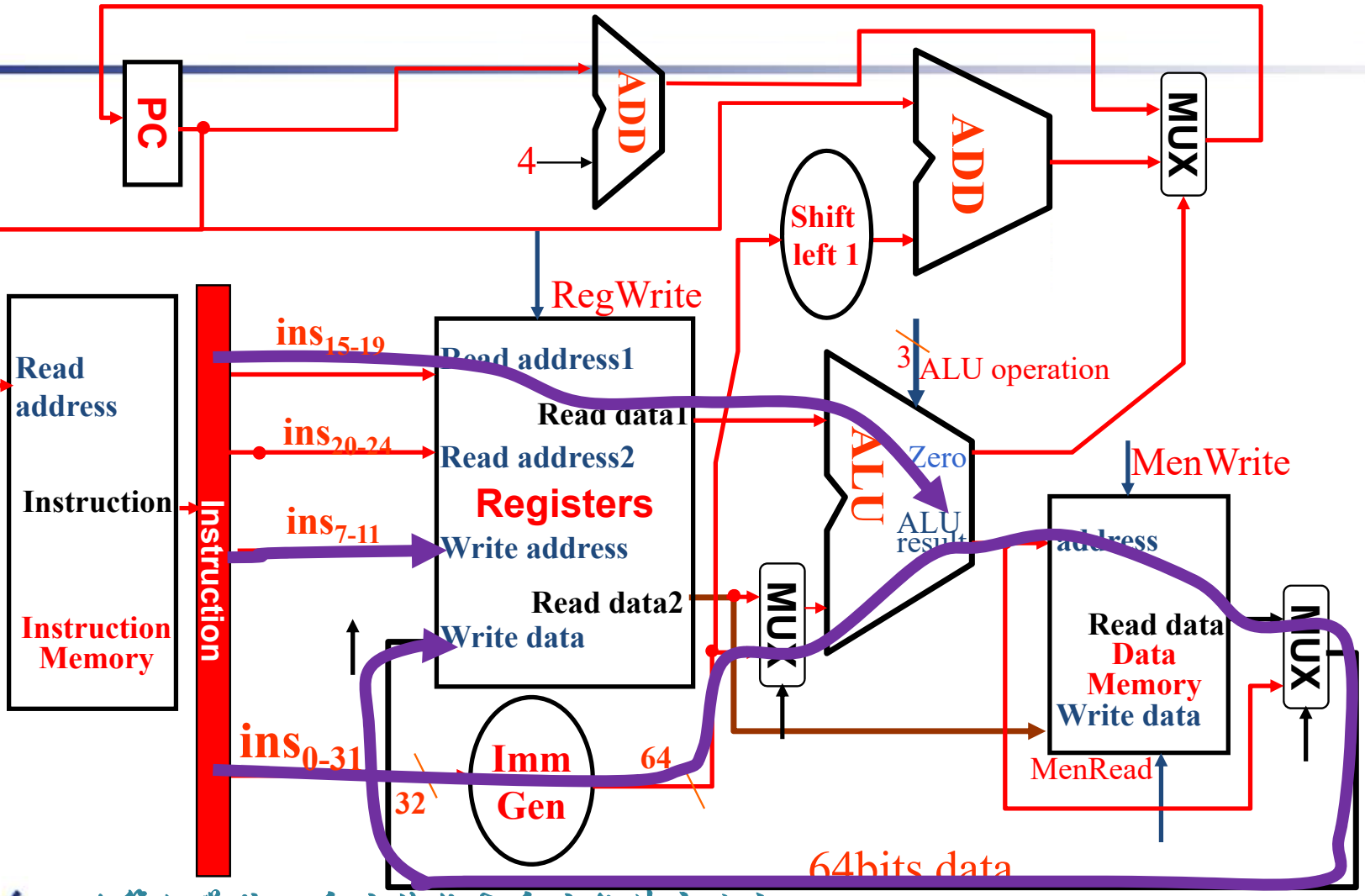
Full datapath

R



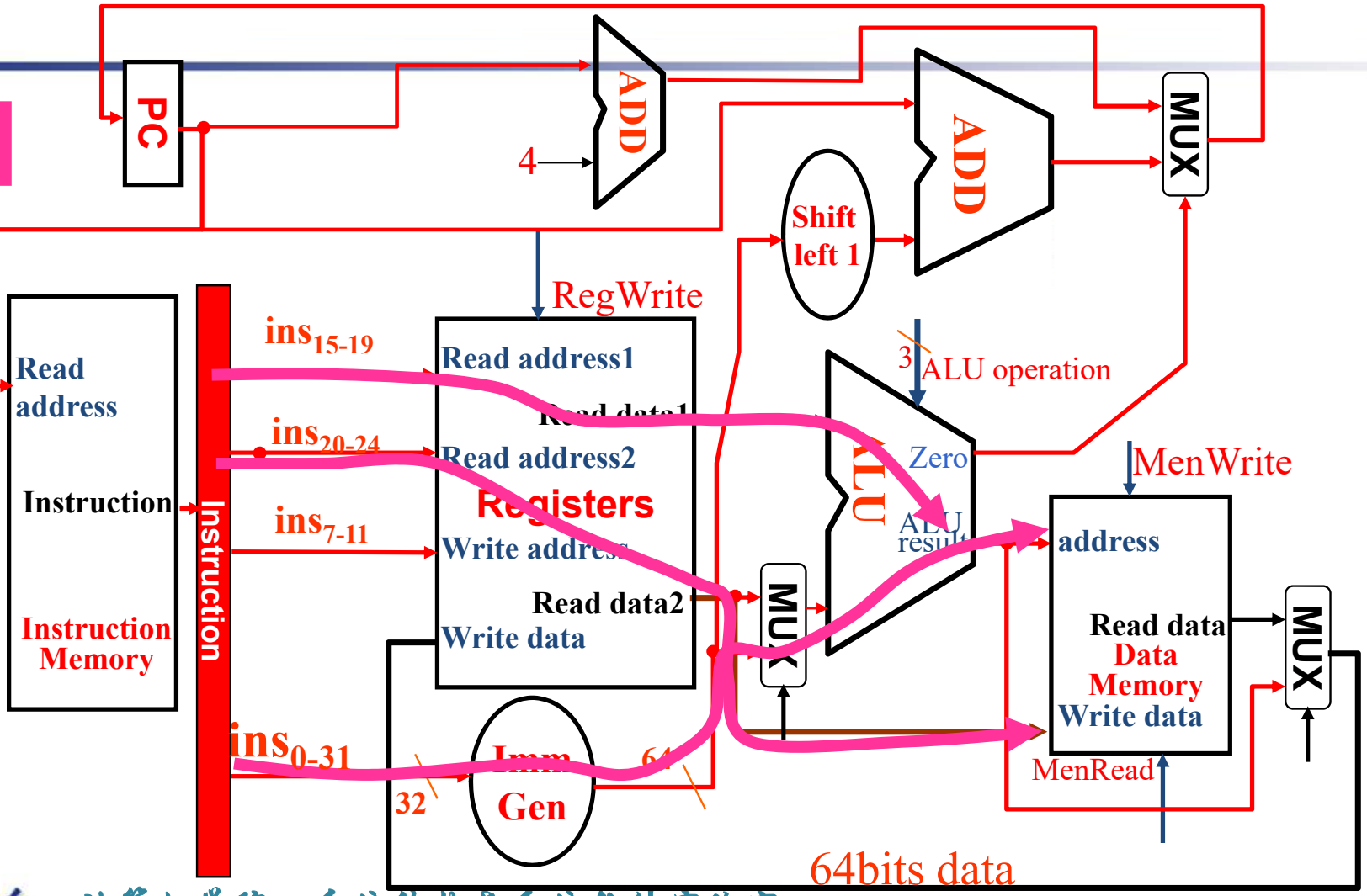
Full datapath

I-ld



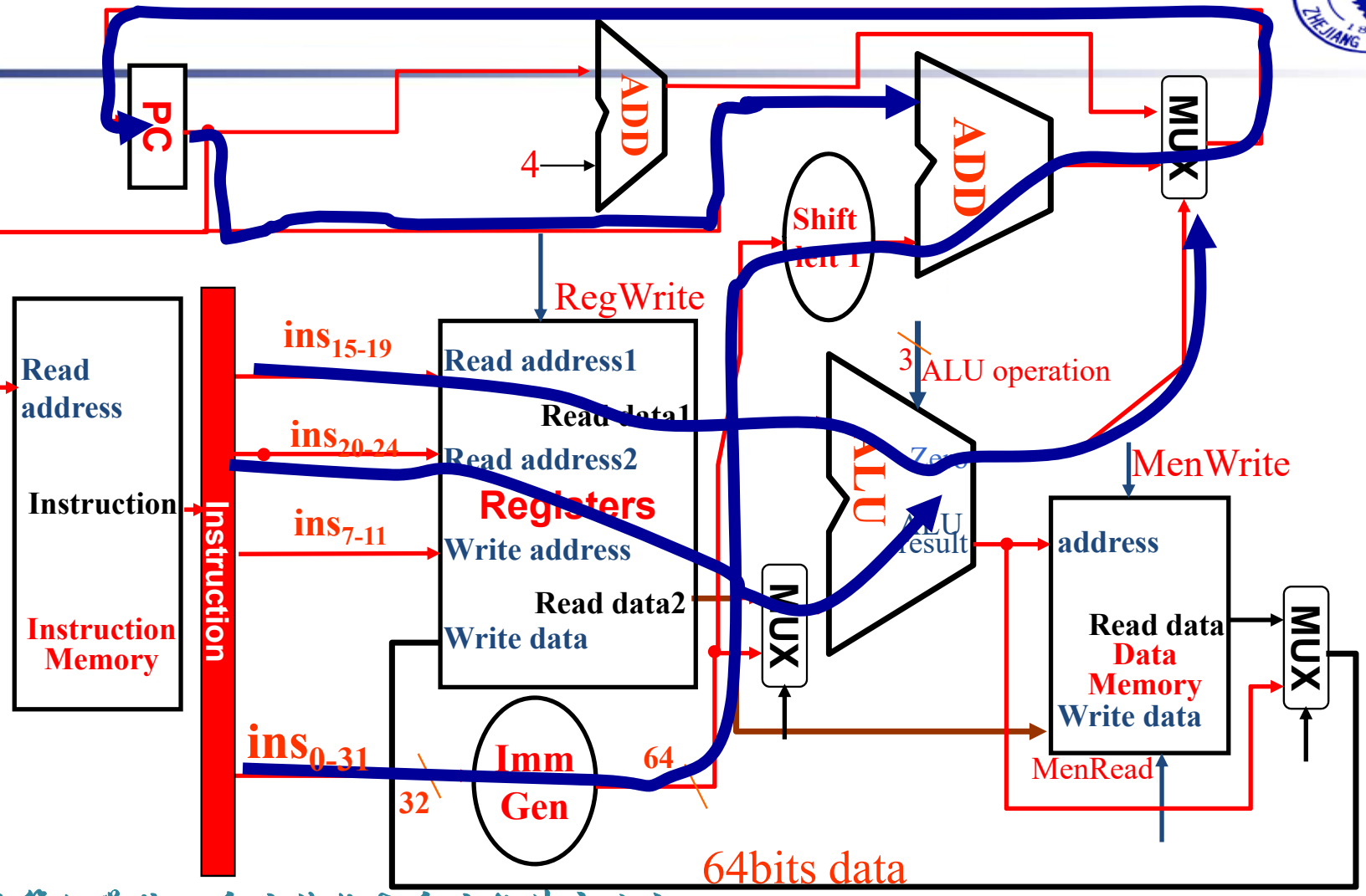
Full datapath

S-sd



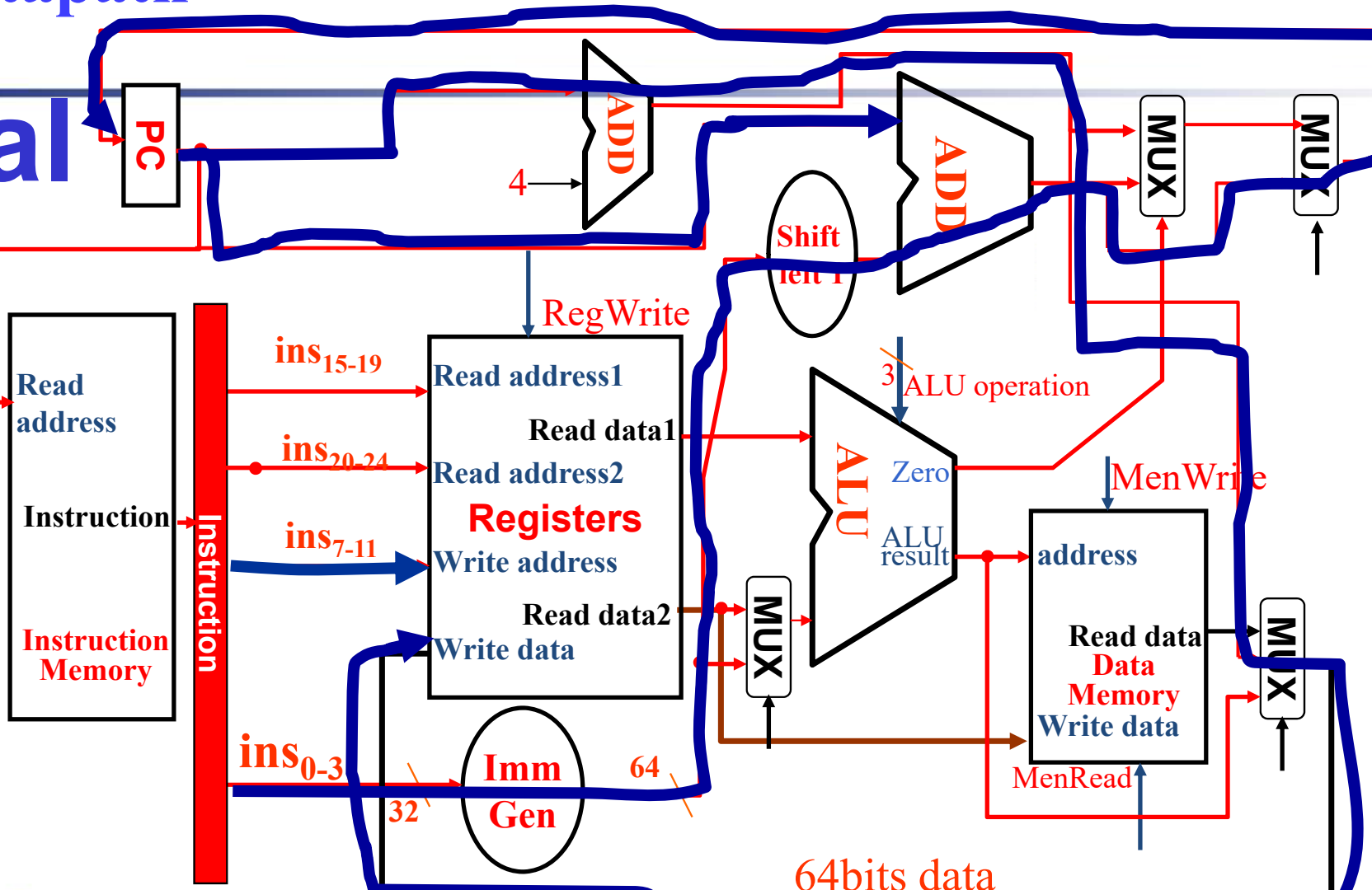
Full datapath

SB-beq

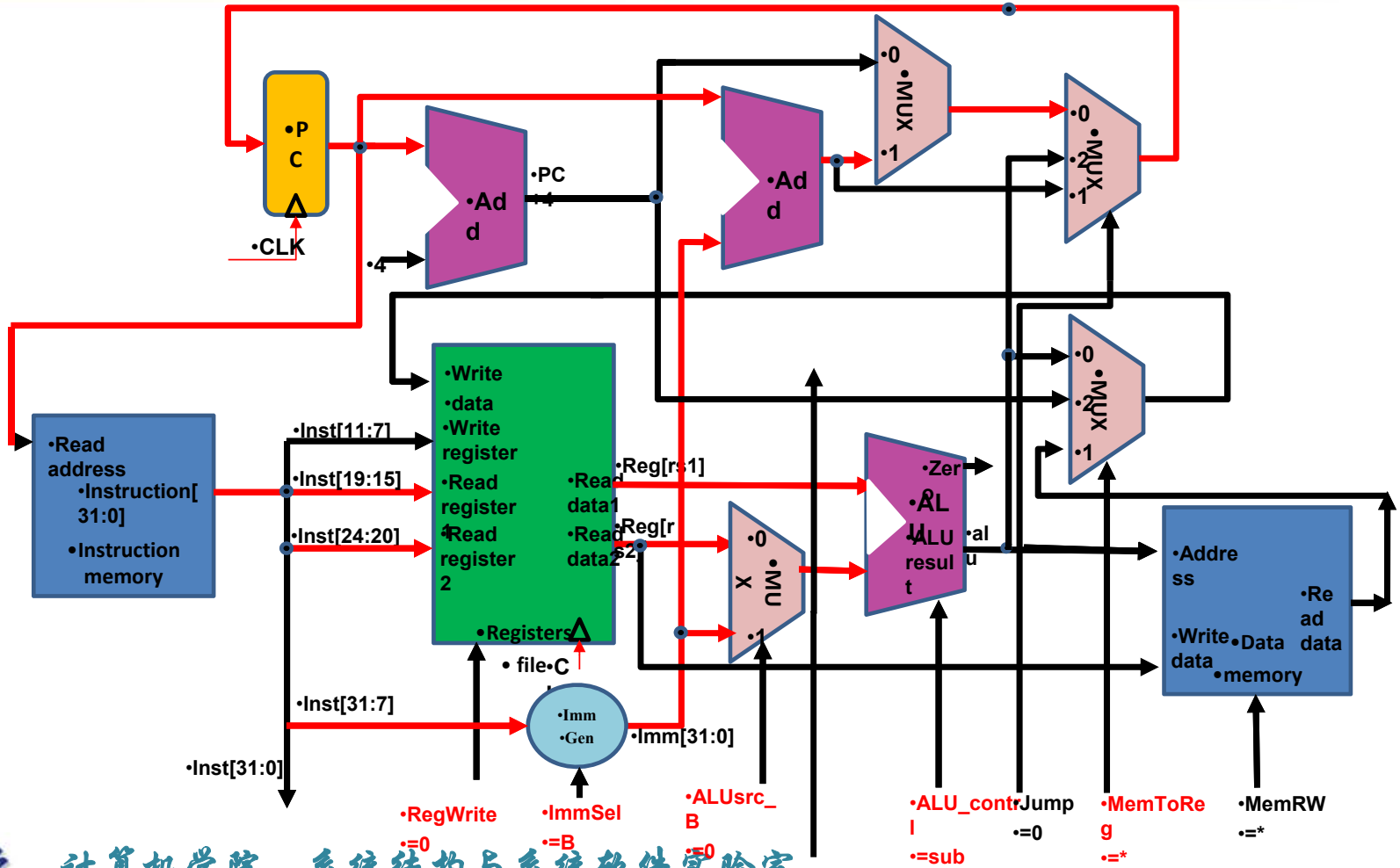


Full datapath

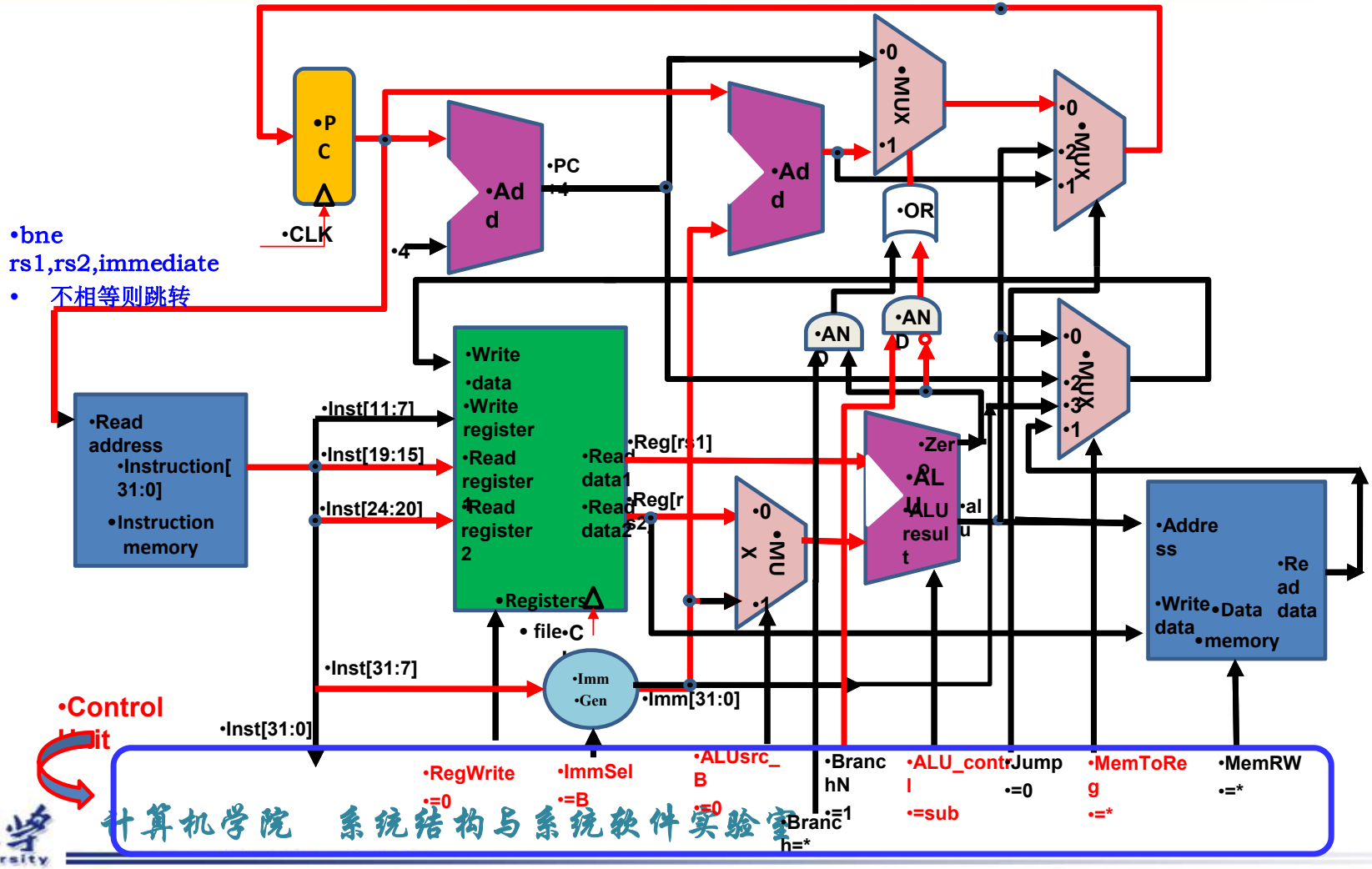
UJ-jal



单周期数据通路结构（add jalr)



单周期数据通路结构 (add lui, bne)

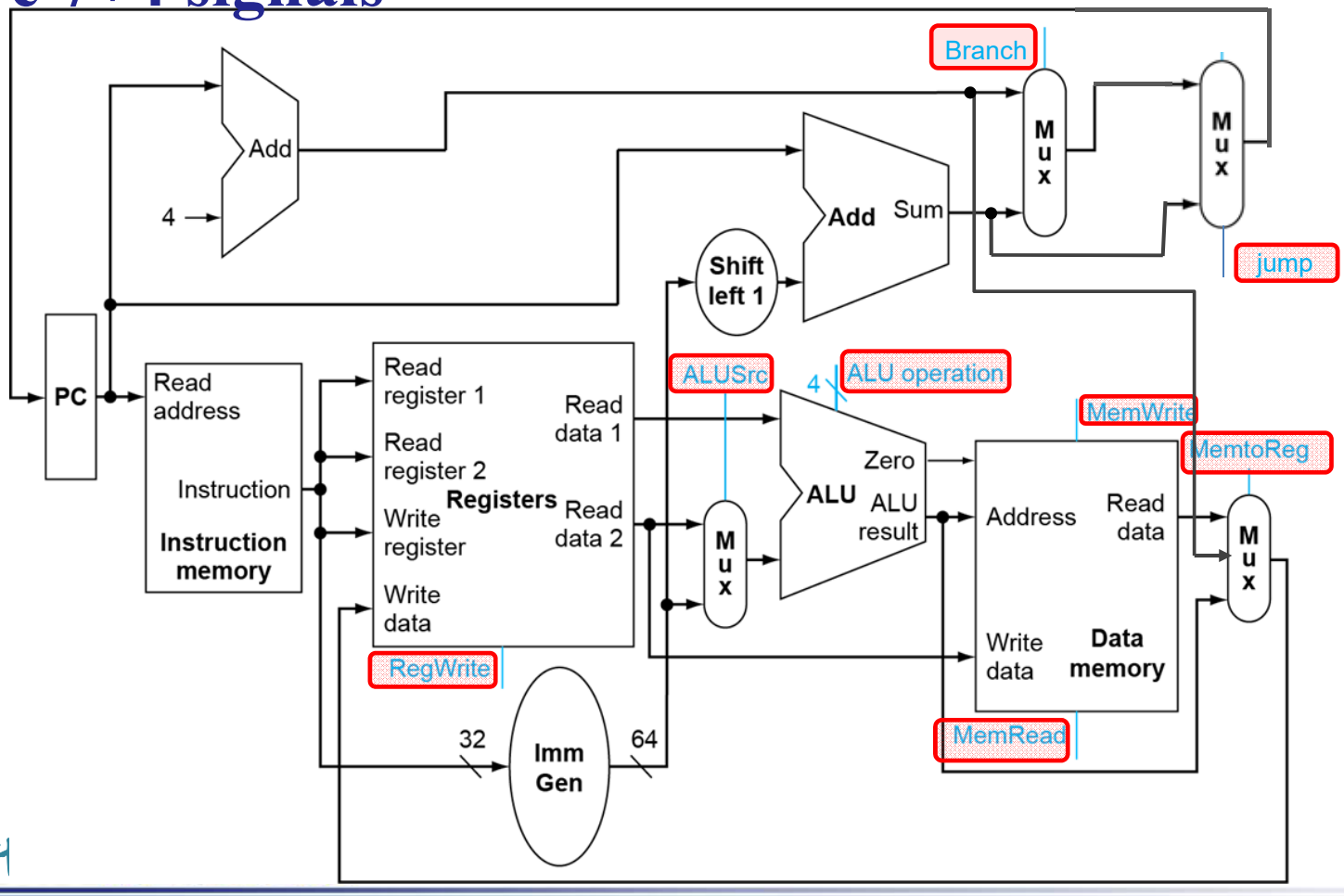


Contents

- Introduction & Logic Design Conventions
- Building a datapath
- **A Simple Implementation Scheme**
- Pipelining

Building the Datapath & Controller

- There are 7+4 signals



Building Controller

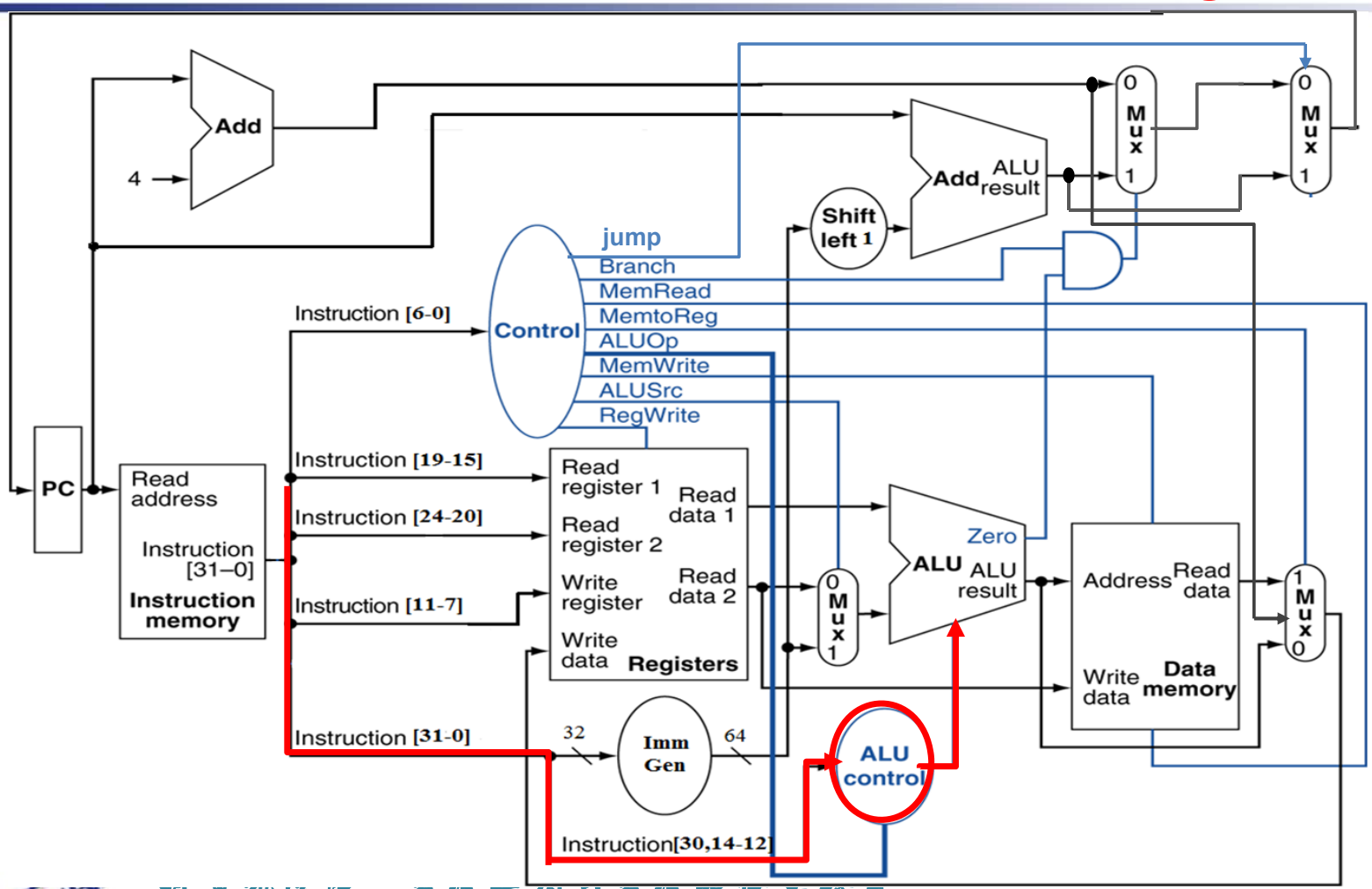
Analyse for cause and effect

- 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

The ALU control is where and other signals(7)

Output signals



What should ALU do ?

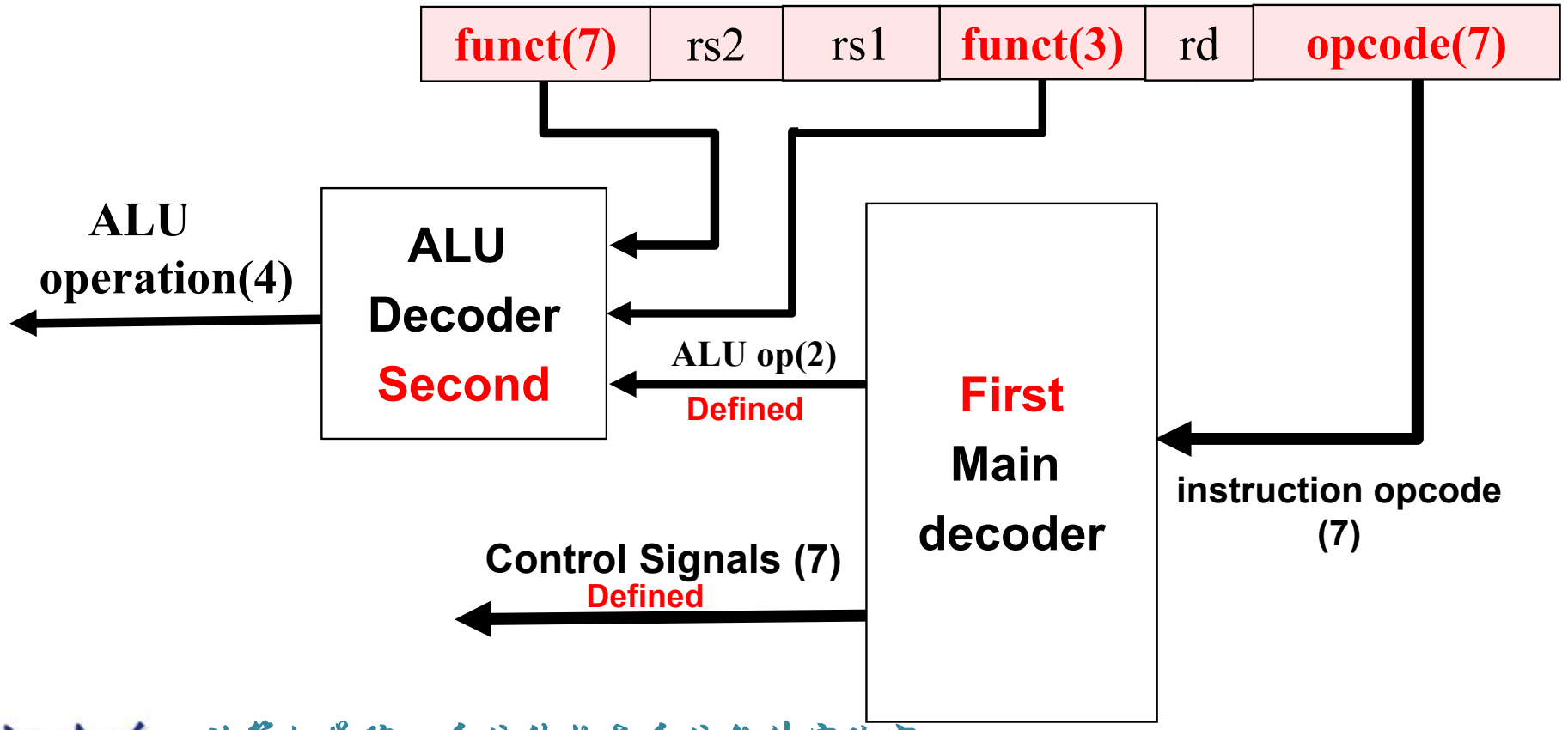
- **ALU used for**
  - Load/Store: F = add
  - Branch: F = subtract
  - R-type: F depends on opcode
- **Assume 2-bit ALUOp derived from opcode**
  - Combinational logic derives ALU control

Operation	Function
000	And
001	Or
010	Add
110	Sub
111	Slt

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

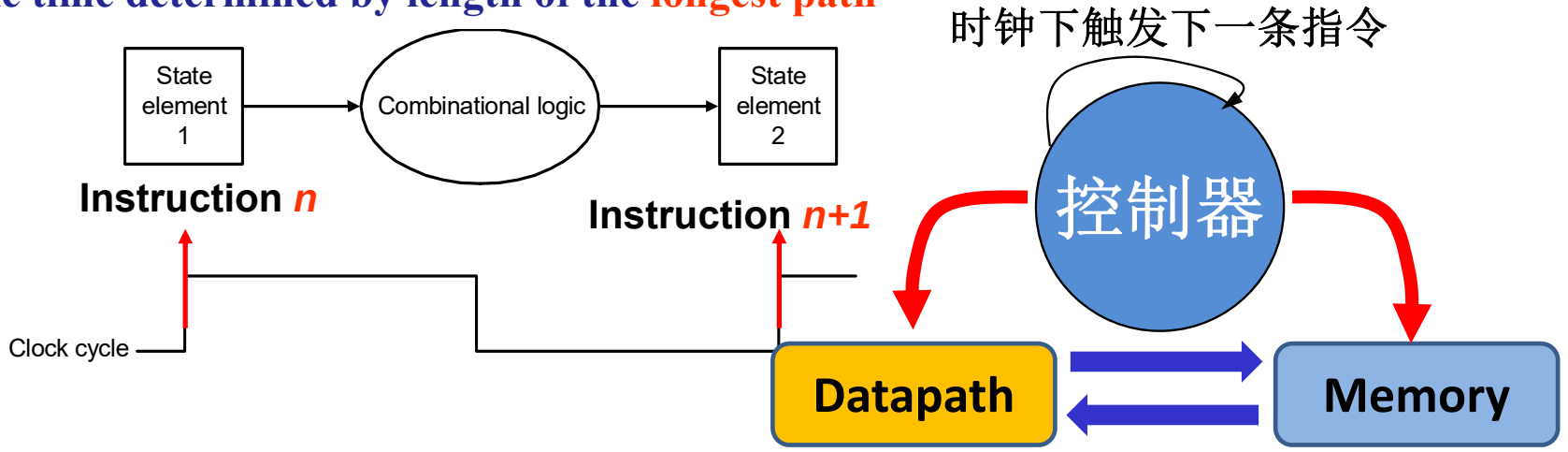
Scheme of Controller

- 2-level decoder



Our Simple 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
  - we use write signals along with clock to determine when to write
- Cycle time determined by length of the longest path

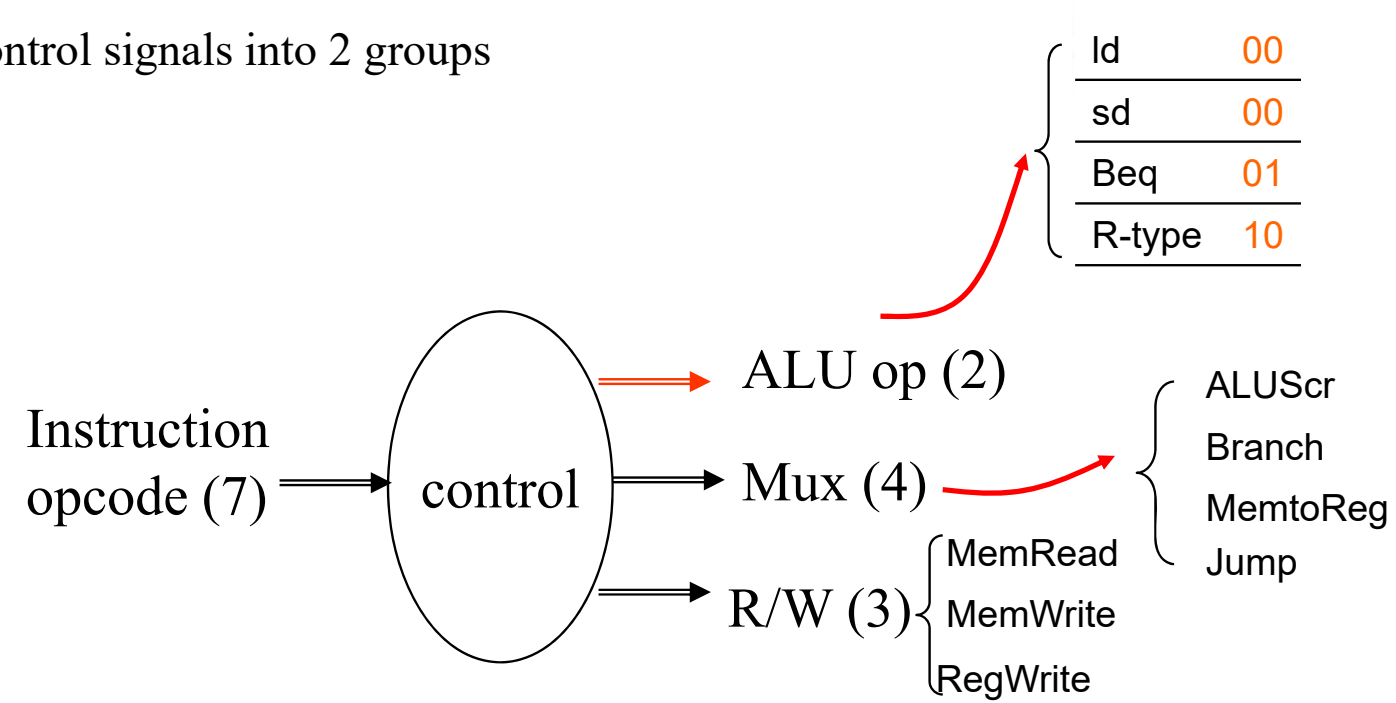


We are ignoring some details like setup and hold times

Designing the Main Control Unit

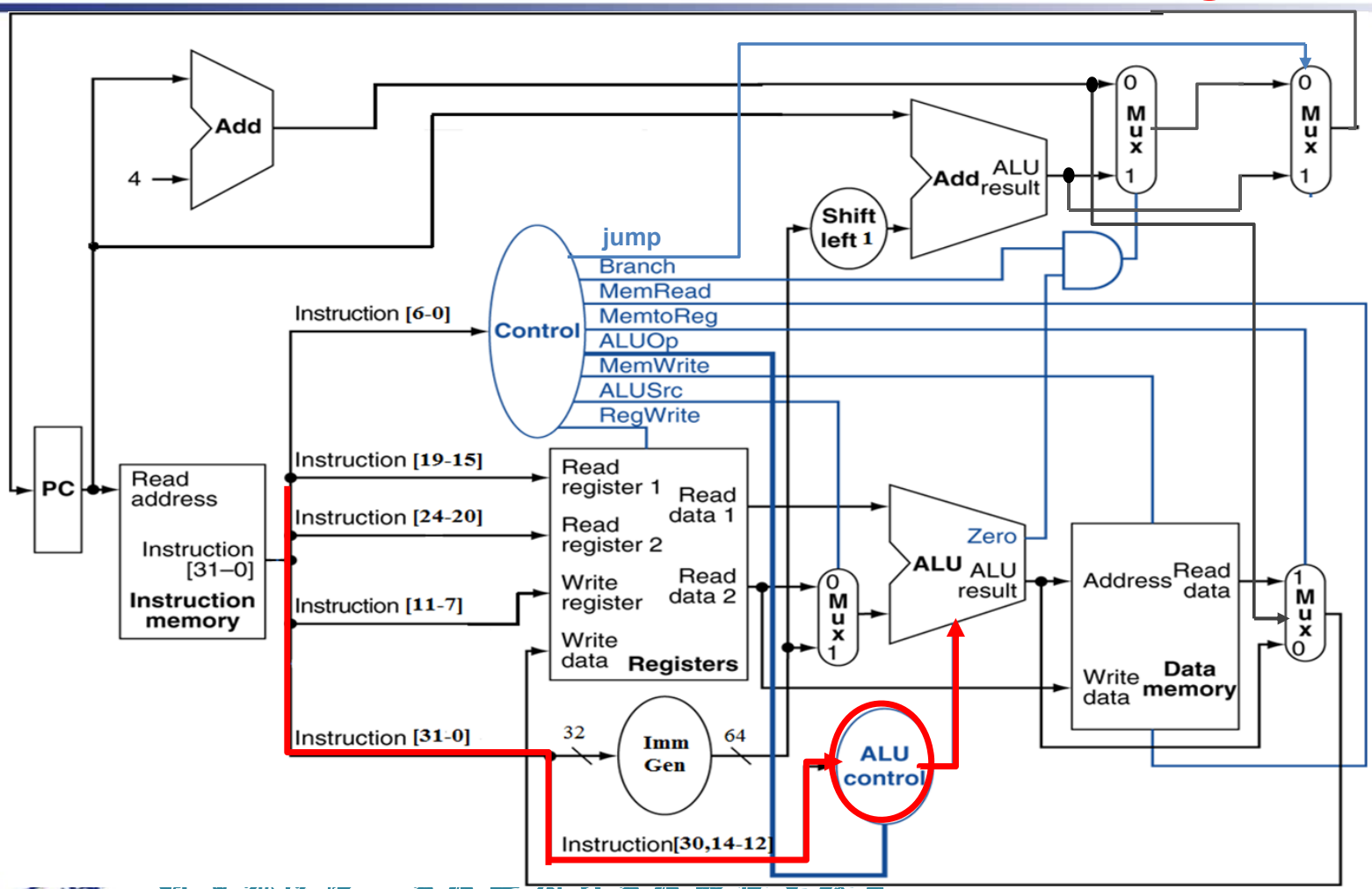
First level

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



The ALU control is where and other signals(7)

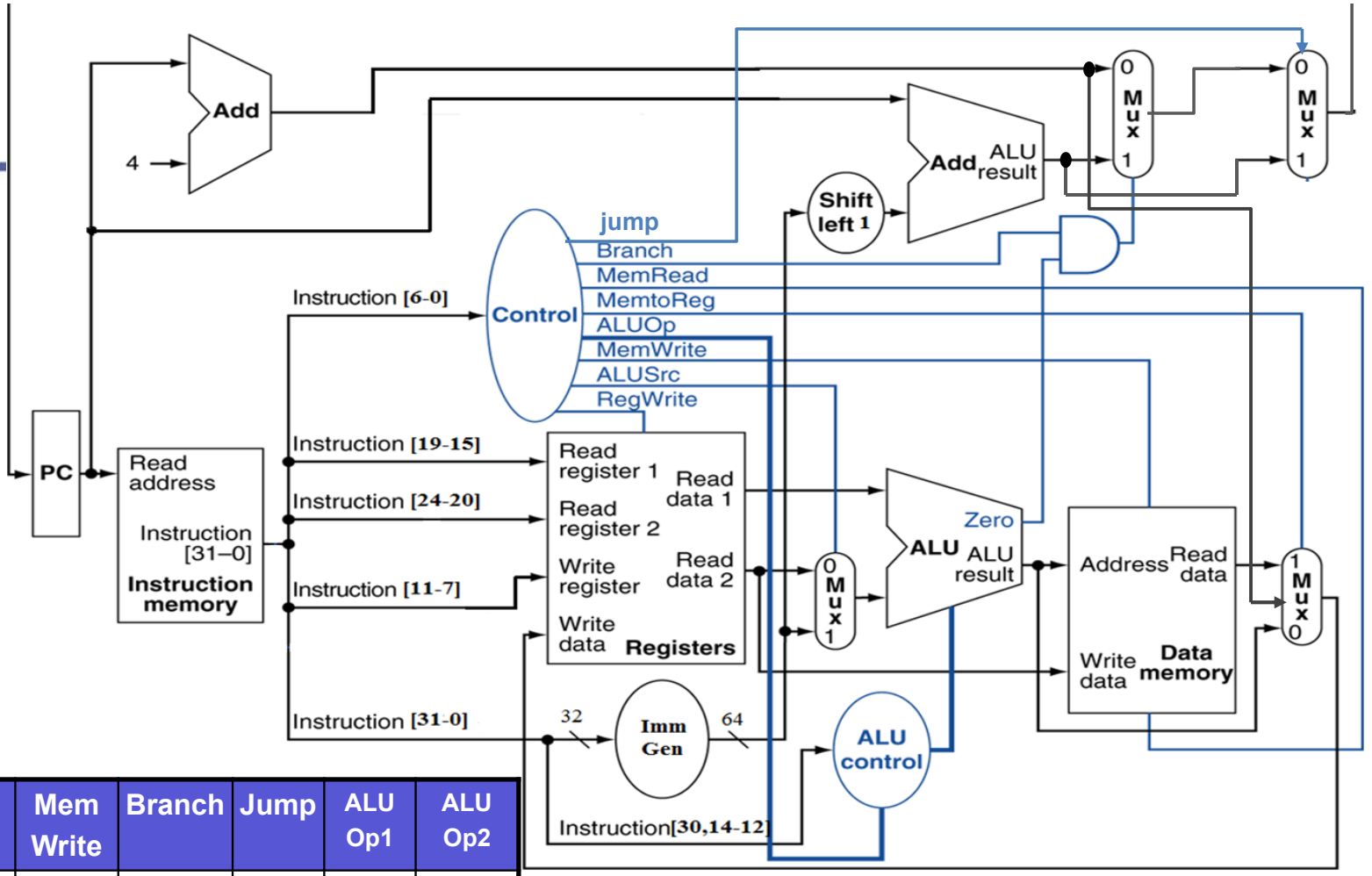
Output signals



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 comes from the output of the Immediate Generator
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

Truth Table for Main decoder



Instruction	ALUSrc	Memto Reg	Reg Write	Mem Read	Mem Write	Branch	Jump	ALU Op1	ALU Op2
R-format									
Ld									
Sd									
beq									
Jal									

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**

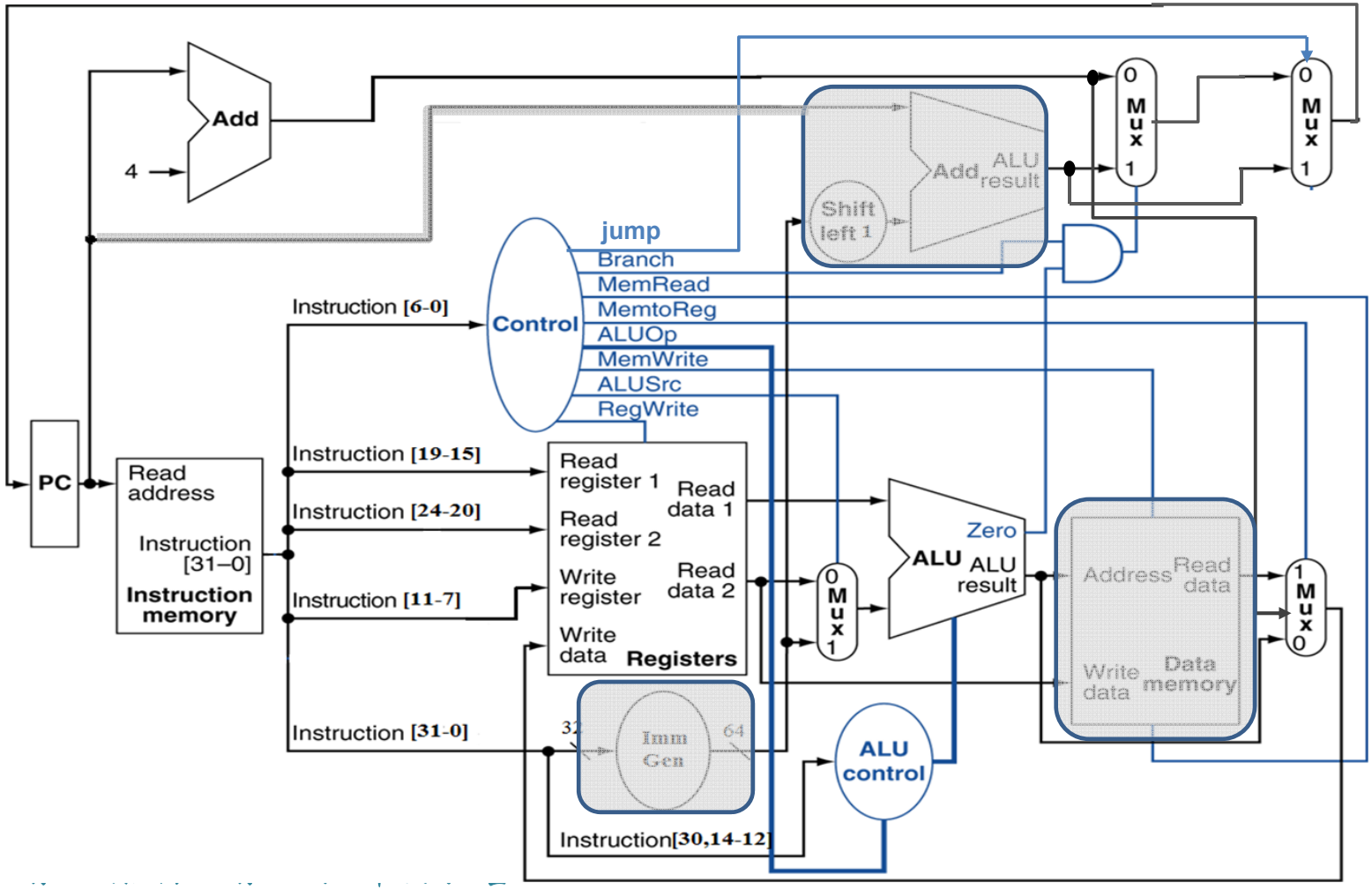
```
assign Fun = {Fun3,Fun7};
always @* begin
    case(ALUop)
    2’b00: ALU_Control = ? ;                    //add计算地址
    2’b01: ALU_Control = ? ;                    //sub比较条件
    2’b10:
        case(Fun)
            4'b0000: ALU_Control = 3'b010 ;     //add
            4’b0001: ALU_Control = ? ;  //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’b11:
        case(Fun3)
                ……
    endcase
```

R-Type Instruc

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

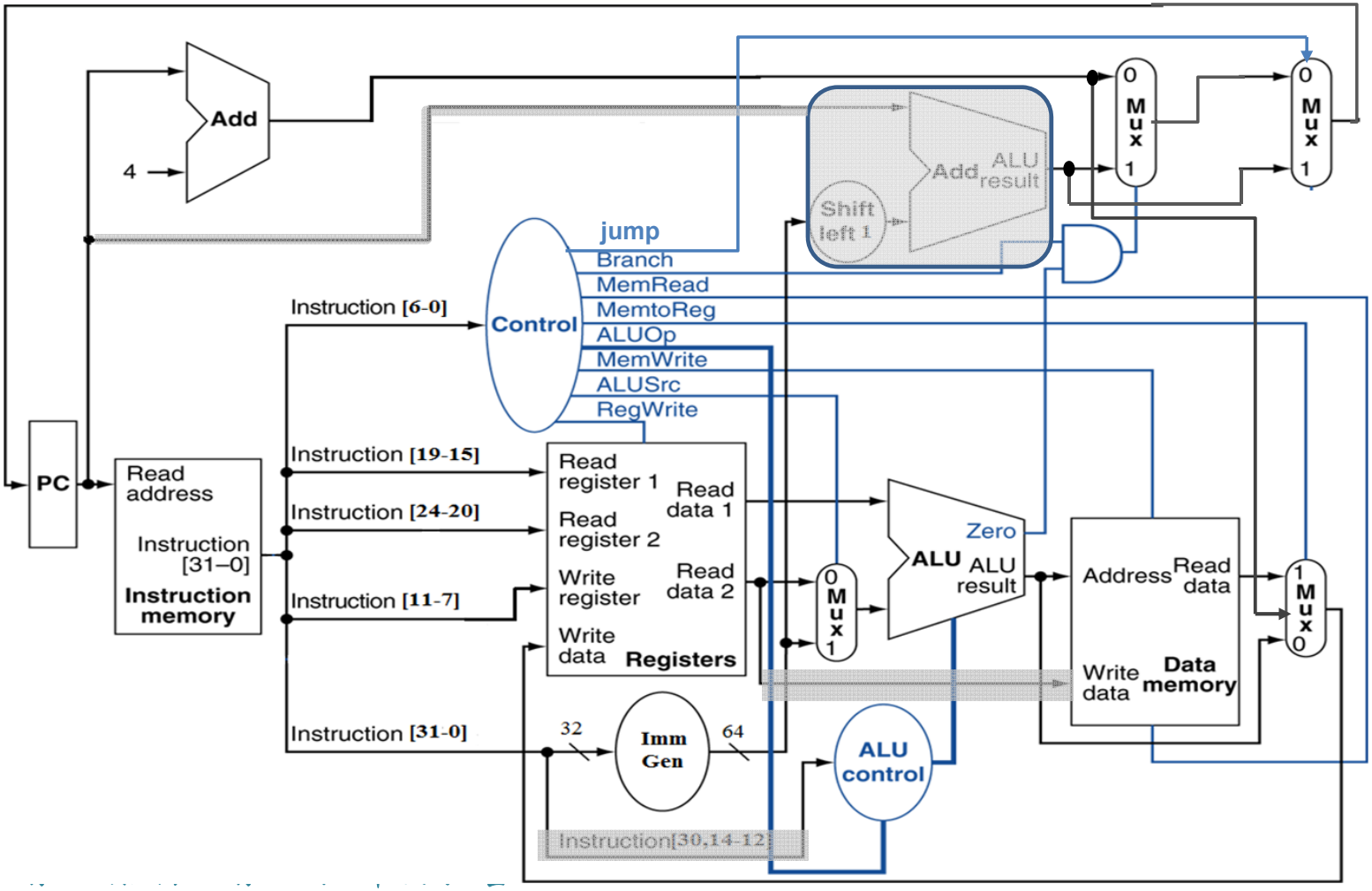


Load Instruction

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

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**

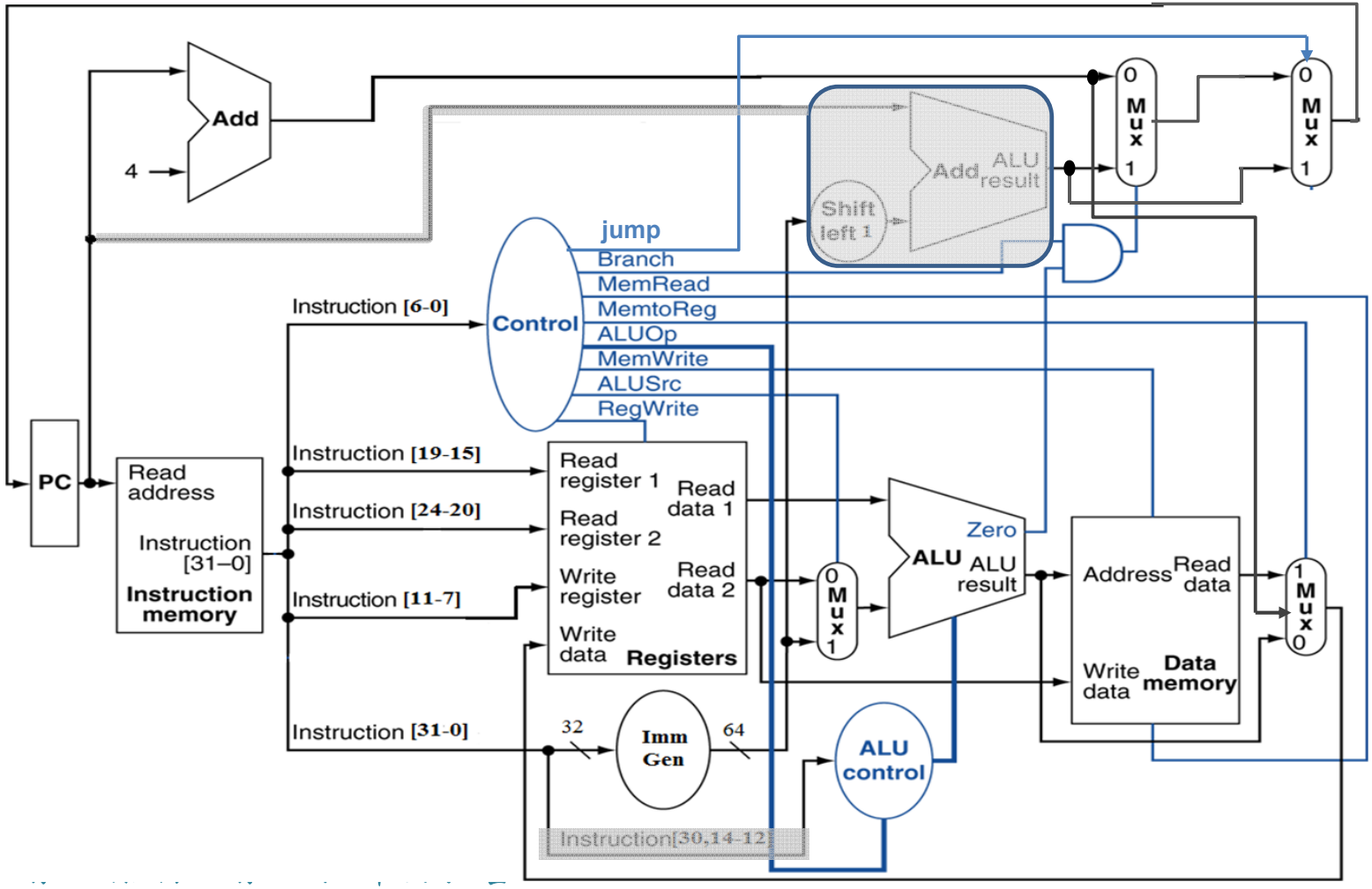


Save Instructi

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

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**

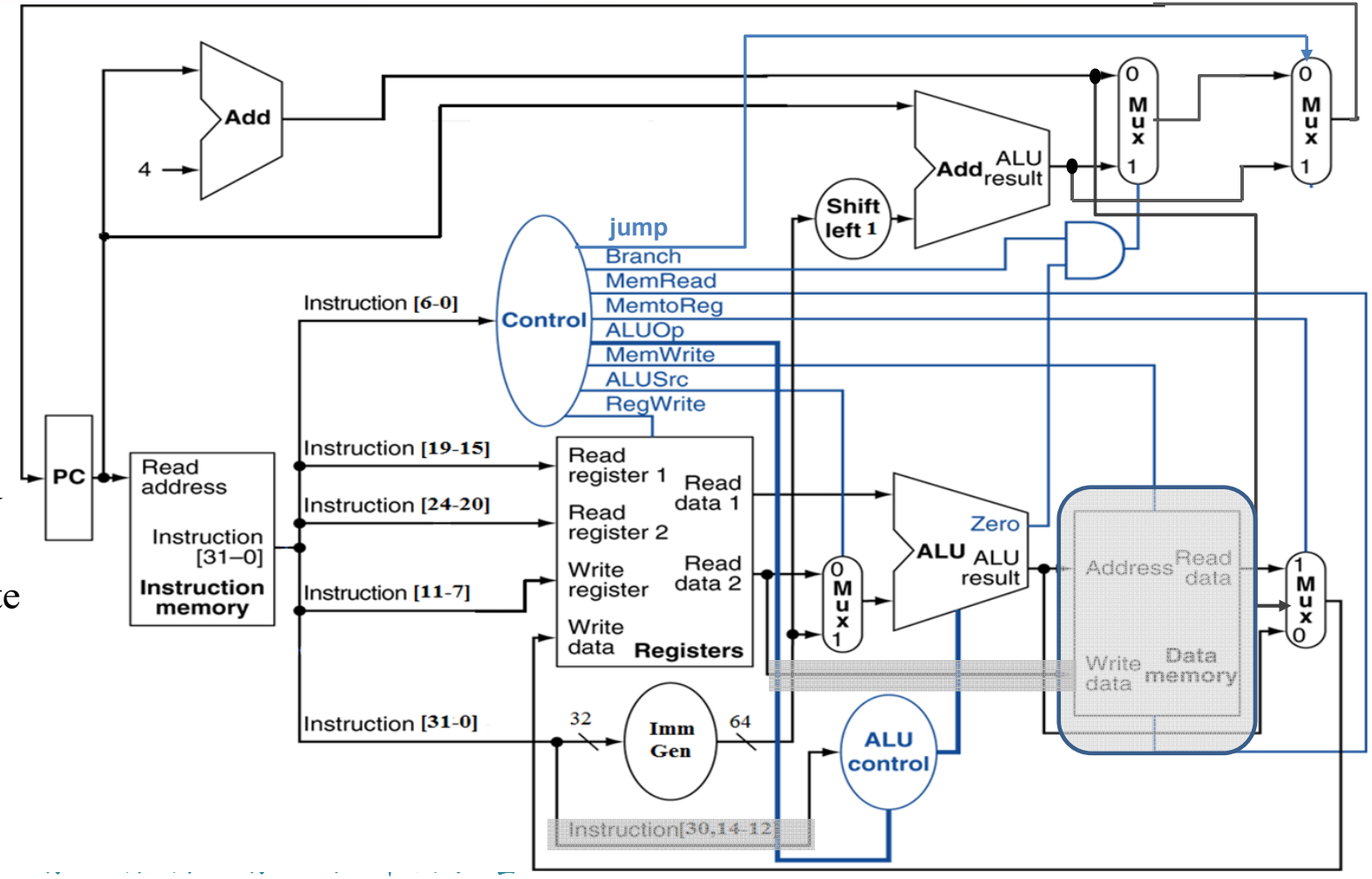


BEQ Instructio

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

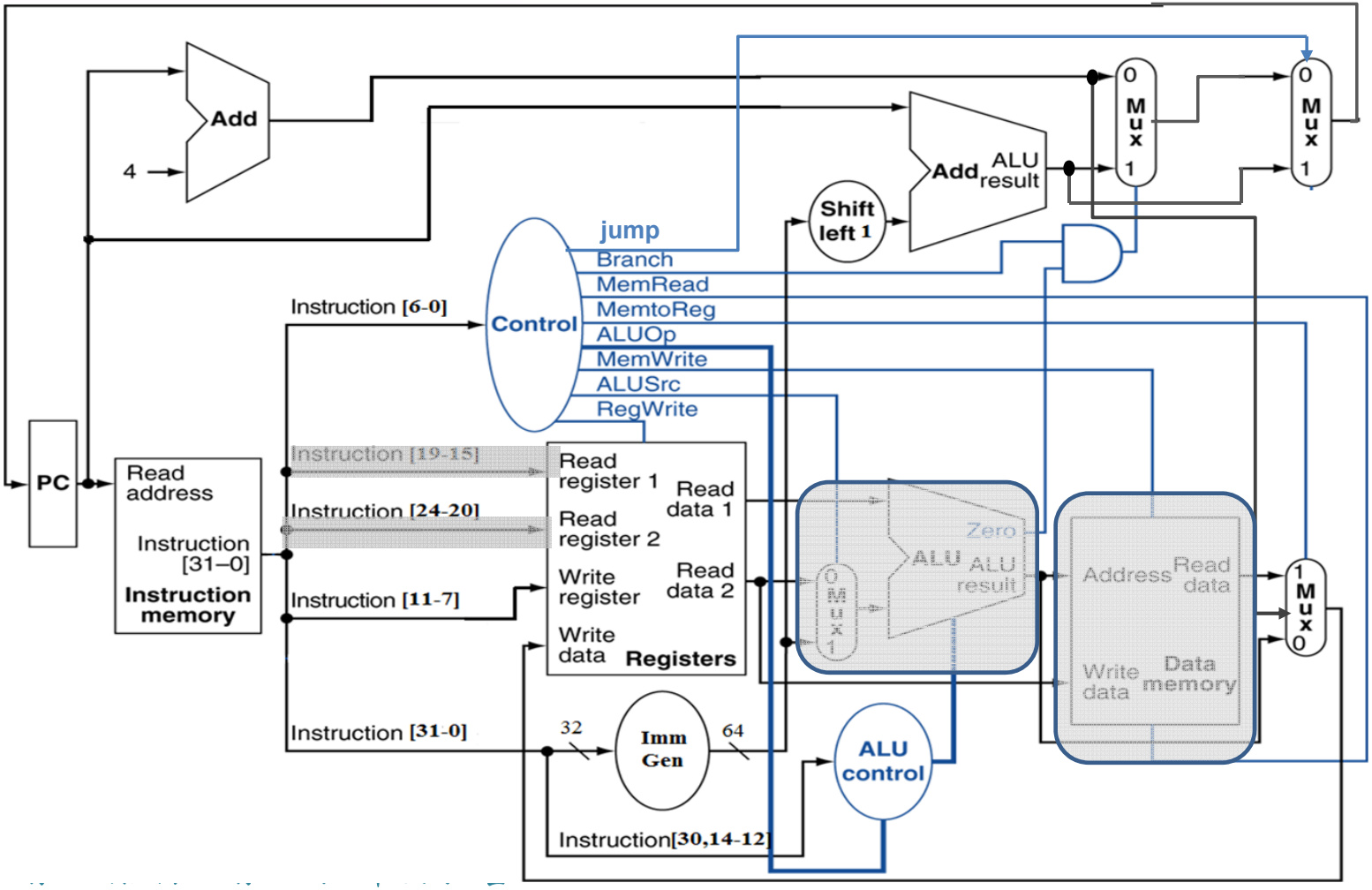


Jal Instruction

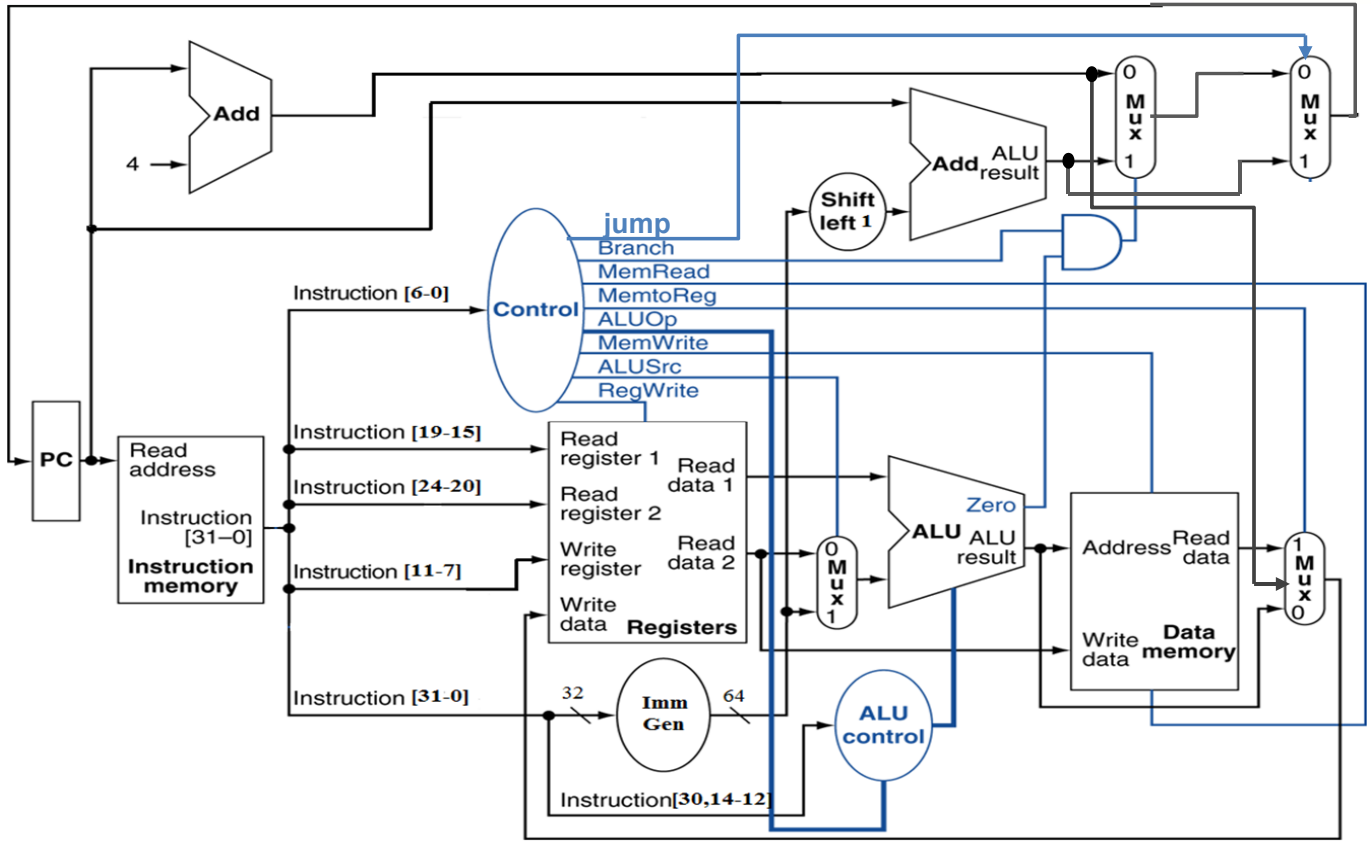
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode

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



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)

Performance in Single Cycle Implementation

- **Let's see the following table:**

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
ld	200ps	100 ps	200ps	200ps	100 ps	800ps
sd	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

- **The conclusion:**

  Different instructions needs different time.

  The clock cycle must meet the need of the slowest instruction.

  So, some time will be wasted.

Performance Issues

- **Longest delay determines clock period**
  - Critical path: load instruction
  - Instruction memory → register file → ALU → data memory → register file
- **Wasteful of area. If the instruction needs to use some functional unit multiple times.**
  - E.g., the instruction ‘mult’ needs to use the ALU repeatedly. So, the CPU will be very large.
- **Violates design principle**
  - Making the common case fast
- **We will improve performance by pipelining**

4.6 Exception

- **The cause of changing CPU's work flow :**
  - Control instructions in program (bne/beq, jal , etc)

    It is foreseeable in programming flow
  - Something happen suddenly (Exception and Interruption)

    It is unpredictable
    - Call Instructions triggered by hardware
- **Exception**
  - Arises within the CPU（e.g.,
- **Interrupt**
  - From an external I/O controller
- **Dealing with them without sacrificing performance is hard**

Type of event	From where?	RISC-V terminology
System reset	External	Exception
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Either

Handling Exceptions

- 保护CPU现场，进入异常
  - Save PC of offending (or interrupted) instruction
    - In RISC-V: Supervisor Exception Program Counter (mEPC)
  - Save indication of the problem
    - In RISC-V: Supervisor Exception Cause Register (mCAUSE)
  - 64 bits, but most bits unused
    - Exception code field: 2 for undefined opcode, 12 for hardware malfunction, …
- 处理中断事件
  - Jump to handler，mtvec寄存器提供地址
  - Assume at 0000 0000 1C09 0000_{hex}
- 退出异常，恢复正常操作
  - 当异常程序处理完成后，最终要从异常服务程序中退出，并返回主程序。对于机器模式，使用MRET退出指令，返回到SEPC存储的pc地址开始执行。

Handler Actions

- **Read cause, and transfer to relevant handler**
- **Determine action required**
- **If restartable**
  - Take corrective action
  - use SEPC to return to program
- **Otherwise**
  - Terminate program
  - Report error using SEPC, SCAUSE, …

Exceptions in RISC V

- Transfer control to exception handler & return from exception
- Control status registers
- CSR instructions
- How to write an exception handler?

RISC-V Privileged

RISC-V Privileged Architecture

All hardware implementations must provide M-mode

- The machine level has the highest privileges
  - and is the only mandatory privilege level for a RISC-V hardware platform.
  - Code run in machine-mode (M-mode) is usually inherently trusted, as it has low-level access to the machine implementation.
  - M-mode can be used to manage secure execution environments on RISC-V.

- User-mode (U-mode) and supervisor-mode (S-mode) are intended for conventional application and operating system usage respectively.

Level	Encoding	Name	Abbreviation	
0	00	User/Application	U	用户模式
1	01	Supervisor	S	监督模式
2	10	Reserved		保留
3	11	Machine	M	机器模式

RISC-V Privilege Modes Usage

- **Each privilege level has**
  - a core set of privileged ISA extensions
    - with optional extensions and variants.
- **Combinations of privilege modes**
  - Implementations might provide anywhere from 1 to 3 privilege modes
    - trading off reduced isolation for lower implementation cost

Number of levels	Supported Modes	Intended Usage
1	M	Simple embedded systems
2	M, U	Secure embedded systems
3	M, S, U	Systems running Unix-like operating systems

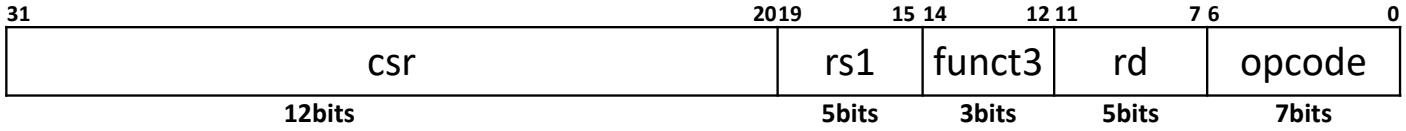
  - For example, machine-mode supports an optional standard extension for memory protection.

RISC-V interrupt structure

- **All hardware implementations must provide M-mode**
  - as this is the only mode that has unfettered access to the whole machine.
  - The simplest RISC-V implementations may provide only M-mode
    - though this will provide no protection against incorrect or malicious application code
- **Machine mode(M-mode) most important task**
  - that is to intercept and handle interrupts/exceptions
  - There are 4096 Control and Status Registers (CSRs)
    - Similar to CP0 of MIPS

Control and Status Registers (CSRs)

- **CSRs, are additional set of registers**
  - accessible by some subset of the privilege levels using the CSR instructions
  - These can be divided into two main classes:
    - those that atomically read-modify-write control and status registers
    - and all other privileged instructions.
- **All privileged instructions encode by the SYSTEM major opcode**
  - CSRs and instructions are associated with one privilege level, they are also accessible at all higher privilege levels
    - The standard RISC-V ISA sets aside a 12-bit encoding space (csr[11:0]) for up to 4,096 CSRs.



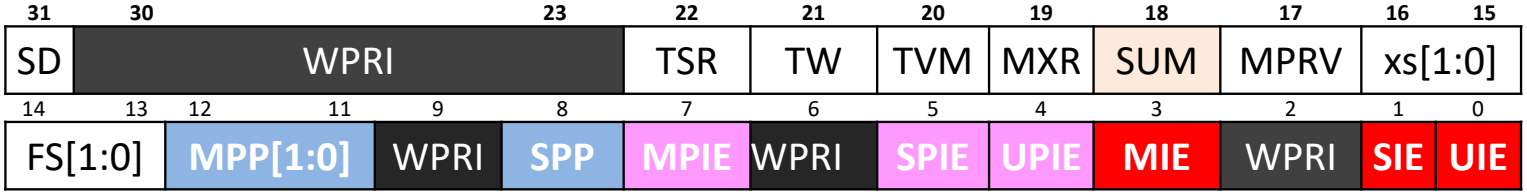
Exception & Interrupt related registers

CSR	Privilege	Abbr.	Name	Description
0x300	MRW	mstatus	Machine STATUS register 机器模式状态寄存器	MIE、MPIE域标记中断全局使能
0x304	MRW	mie	Machine Interrupt Enable register 机器模式中断使能寄存器	控制不同类型中断的局部使能
0x305	MRW	mtvec	Machine trap-handler base address 机器模式异常入口基地址寄存器	进入异常服务程序基地址
0x341	MRW	mepc	Machine exception program counter 机器模式异常PC寄存器	异常断点PC地址
0x342	MRW	mcause	Machine trap cause register 机器模式原因寄存器	处理器异常原因
0x343	MRW	mtval	Machine Trap Value register 机器模式异常值寄存器	处理器异常值地址或指令
0x344	MRW	mip	Machine interrupt pending 机器模式中断挂起寄存器	处理器中断等待处理

Interrupt Registers： mstatus(0x300)

- Machine STATUS register
  - These bits are primarily used to guarantee atomicity with respect to interrupt handlers in the current privilege mode.

bit	Name	Attributes	Description
0	UIE	RW	User interrupt enable
1	SIE	RW	Supervisor interrupt enable
3	MIE	RW	Machine interrupt enable
4	UPIE	RW	previous user-level interrupts enable
5	SPIE	RW	Previous Supervisor interrupt-enable
7	MPIE	RW	Previous Machine interrupt enable
8	SPP	RW	Supervisor Previous Privilege mode
12:11	MPP	RW	Machine Previous Privilege mode
……			
31:23,9,6,2	WPRI		Reserved



Interrupt Registers： mie/mip(*0x304/344*)

- **Machine Interrupt Enable register**
  - MIE register controls whether it can respond to interrupts, corresponding different modes
    - MEIE、SEIE and UEIE enable external interrupt
    - MSIE、SSIE & USIE enable software interrupts
    - MTIE、STIE and UTIE enable timer interrupts
- **Machine interrupt-pending register**
  - The mip register is an MXLEN-bit read/write register containing information on pending interrupts

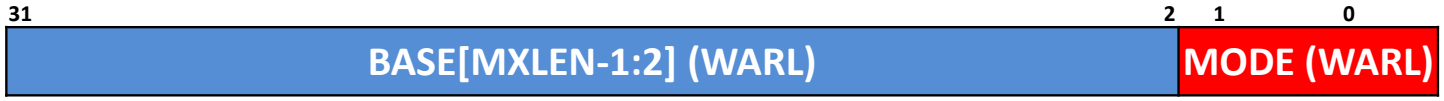
	31 … 12	11	10	9	8	7	6	5	4	3	2	1	0
mie	WPRI	**MEIE**	WPRI	**SEIE**	UEIE	**MTIE**	WPRI	**STIE**	UTIE	**MSIE**	WPRI	**SSIE**	USIE
mip	WPRI	**MEIP**	WPRI	**SEIP**	UEIP	**MTIP**	WPRI	**STIP**	UTIP	**MSIP**	WPRI	**SSIP**	USIP

M, S and U correspond to M mode, S mode and U mode interrupt respectively

Interrupt Registers： mtvec *(0x305)*

Machine Trap-Vector Base-Address Register

- The mtvec register holds trap vector configuration, consisting of a vector base address (BASE) and a vector mode (MODE)



- The value in the BASE field must always be aligned on a 4-byte boundary, and the MODE setting may impose additional alignment constraints on the value in the BASE field.

MODE Value	Name	Description
0	Direct(查询)	All exceptions set pc to BASE
1	Vectored(向量)	Asynchronous interrupts set pc to BASE+4×cause
≥2	--	Reserved

BASE+ 4×Exception Code

Interrupt Registers： mepc *(0x341)*

Machine Exception Program Counter

- mepc is a WARL register that must be able to hold all valid physical and virtual addresses.
  - When a trap is taken into M-mode, mepc is written with the address of the instruction that was interrupted or exception.

- The low bit of mepc (mepc[0]) is always zero.
  - On implementations that support only IALIGN=32, the two low bits (mepc[1:0]) are always zero.
- Exception： `mepc ← pc`
- Interrupted： `mepc ← pc + 4`

Interrupt Registers： mcause (0x342)

Machine Cause Register (mcause)

- When a trap is taken mcause register is written with a code indicating the event that caused the Exception/Interruption.

- Exception Code与mtvec的向量模式相对应
  - 在异步中断时，不同的模式会跳转到不同的入口
- The Exception Code is a WLRL
  - Write/Read Only Legal Values
  - 如果写入值不合法可以引发非法指令异常

Exception Code mcause *(0x342)*

INT	E Code	Description
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2	Reserved for future standard use
1	**3**	**Machine software interrupt**
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6	Reserved for future standard use
1	7	Machine timer interrupt
1	8	User external interrupt
1	9	Supervisor external interrupt
1	10	Reserved for future standard use
1	**11**	**Machine external interrupt**
1	12-15	Reserved for future standard use
1	≥16	Reserved for platform use
0	16-23	Reserved for future standard use
0	32-47	Reserved for future standard use
0	14/≥64	Reserved for future standard use

INT	E Code	Description
0	0	Instruction address misaligned
0	1	Instruction access fault
0	**2**	**Illegal instruction**
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	Reserved
0	11	**Environment call from M-mode**
0	12	Instruction page fault
0	13	Load page fault
0	15	Store/AMO page fault
0	24-31	Reserved for custom use
0	48-63	Reserved for custom use

RISV-V Interrupt priority

External interrupt > Software interrupt > Timer interrupt

Synchronous exception priority

Priority	Exception Code	Description
Highest	3	Instruction address breakpoint
	12	Instruction page fault
	1	Instruction access fault
	2	Illegal instruction
	0	Instruction address misaligned
	8，9，11	Environment call
	3	Environment break
	3	Load/Store/AMO address breakpoint
	6	Store/AMO address misaligned
	4	Load address misaligned
	15	Store/AMO page fault
	13	Load page fault
	7	Store/AMO access fault
Lowest	5	Load access fault

CSR Instruction

- **All CSR instructions atomically read-modify-write a single CSR**
  - whose CSR specifier is encoded in the 12-bit csr field of the instruction held in bits 31–20
    - The immediate forms use a 5-bit zero-extended immediate encoded in the rs1 field.

	31 … 20	19 … 15	14 … 12	11 … 7	6 … 0
	csr	rs1	funct3	rd	opcode
	12bits	5bits	3bits	5bits	7bits
I：CSRRW	csr	rs1	001	rd	1110011
I： CSRRS	csr	rs1	010	rd	1110011
I： CSRRC	csr	rs1	011	rd	1110011
I：CSRRWI	csr	zimm	101	rd	1110011
I： CSRRSI	csr	zimm	110	rd	1110011
I： CSRRCI	csr	zimm	111	rd	1110011

Interrupts Instruction

- **Exception return**
  - MRET
    - PC <= MEPC；(MStatus寄存器有变化)

	31–25	24–20	19–15	14–12	11–7	6–0
R：MRET	0011000	00010	00000	000	00000	1110011
R：SRET	0001000	00010	00000	000	00000	1110011
R：wfi	0001000	00101	00000	000	00000	1110011

- **Environment call**
  - ecall
    - MEPC = ecall指令本身的PC值
- **Breakpoint**
  - ebreak
    - MEPC = ebreak指令本身的PC值

	31–25	24–20	19–15	14–12	11–7	6–0
I：ecall	0000000	00000	00000	000	00000	1110011
I：ebreak	0000000	00001	00000	000	00000	1110011

How Control Checks for Exceptions

- **Add test logic**
  - Illegal instruction, load/store address misaligned,
- **add control signal**
  - CauseWrite for mcause
  - EPCWrite for mepc
  - TVALWrite for mtval
- **process of control**
  - mepc ← pc (exception) or pc+4 (interruption)
  - mcause ← set corresponding bit
  - mtval ← memory address or illegal instruction
  - Mstatus.mpie ← mstatus.mie, mstatus.mie ← 0; mstatus.mpp ← mp, mp ← 11
  - PC ← address of process routine (mtvec, eg. 0000 0000 1C09 0000)

When jump to exception handler?

- **Jump to handler**
  - Assume at 0000 0000 1C09 0000
- **Jump when**
  - Mstatus.mie = 1 && mie[i] =1 && mip[i] = 1

How to back to the exception breakpoint

- **mret**
  - PC ← CSRs[mepc]
  - mstatus.MIE ← mstatus. MPIE
  - mp ← mstatus. mpp

An Alternate Mechanism

- **Vectored Interrupts（e.g., x86, ARM）**
  - Handler address determined by the cause
- **Exception vector address to be added to a vector table base register:**
  - Undefined opcode $00\ 0100\ 0000_{two}$
  - Hardware malfunction: $01\ 1000\ 0000_{two}$
  - …: …
- **Instructions either**
  - Deal with the interrupt, or
  - Jump to real handler

典型处理器中断结构：向量模式

- **Intel x86中断结构**
  - 间接向量：000～3FF，占内存最底1KB空间
    - 每个向量由二个16位生成20位中断地址
    - 共256个中断向量，向量编号n=0～255
    - 分硬中断和软中断，响应过程类同，触发方式不同
    - 硬中断响应由控制芯片8259产生中断号n(接口原理课深入学习)
- **ARM中断结构**
  - 固定向量方式(嵌入式课程深入学习)

异常类型	偏移地址(低)	偏移地址(高)	
复 位	00000000	FFFF0000	
未定义指令	00000004	FFFF0004	
软中断	00000008	FFFF0008	
预取指令终	0000000C	FFFF000C	
数据终止	00000010	FFFF0010	
保留	00000014	FFFF0014	
中断请求(IRQ)	00000018	FFFF0018	
快速中断请求(FIQ)	0000001C	FFFF001C	

Simplify Interrupt Design

- 简化中断设计
  - 采用ARM中断向量(不兼容RISC-V)
    - 实现非法指令异常和外中断
    - 设计EPC
  - 兼容RISC-V*
    - 仅M-Mode中断寄存器(MCause、Mstatus、MIE、MIP、MEPC和MTVEC)
    - 设计mret、CSRRW (csrw rd, csr, rs1)和ecall指令
- ARM中断向量表

向量地址	ARM异常名称	ARM系统工作模式	本课程定义
0x0000000	复位	超级用户Svc	复位(M-MODE)
0x0000004	未定义指令终止	未定义指令终止Und	非法指令异常
0x0000008	软中断（SWI）	超级用户Svc	ECALL
0x000000c	Prefetch abort	指令预取终止Abt	Int外部中断（硬件）
0x0000010	Data abort	数据访问终止Abt	Reserved自定义
0x0000014	Reserved	Reserved	Reserved自定义
0x0000018	IRQ	外部中断模式IRQ	Reserved自定义
0x000001C	FIQ	快速中断模式FIQ	Reserved自定义

Assignment

- **Reading**
  - Textbook 4.1-4.4
- **Assignment**
  - 4.1, 4.4, 4.6, 4.7, 4.8, 4.13


END