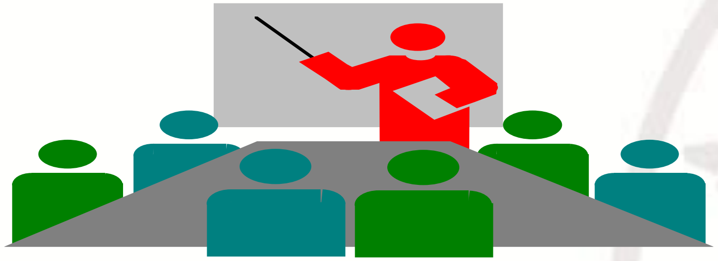
计算机组成与设计



Computer Organization & Design

The Hardware/Software Interface

Chapter 4

The Processor : Datapath and Control

A General Purpose Digital Circuit

施青松 Asso. Prof. Shi Qingsong
College of Computer Science and Technology, Zhejiang University
zjsqs@zju.edu.cn

Contents of Chapter 4

Introduction

- **CPU performance factors**
  - Instruction count
    - Determined by ISA and compiler
  - CPI and Cycle time
    - Determined by CPU hardware

- 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, jalr;
  - S-Type： sw；
  - B-Type： beq；
  - J-Type： Jal。

- **We will examine two RISC-V implementations**
  - A simplified version
  - A more realistic pipelined version
- **Simple subset, shows most aspects**
  - Memory reference: ld/lw, sd/sw
  - Arithmetic/logical: add, sub, and, or, slt
  - Control transfer: beq， jal

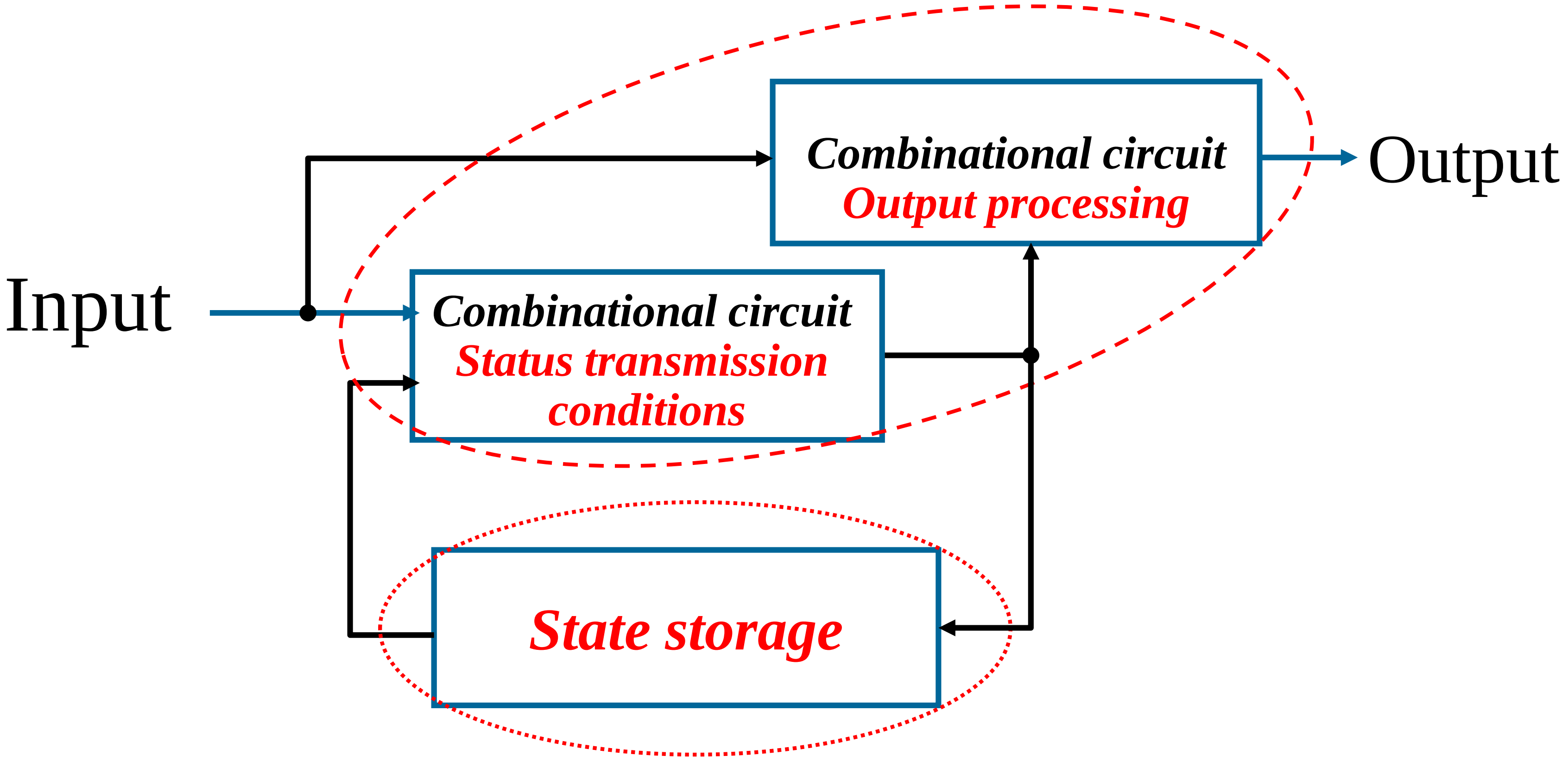
Watch the video：处理器设计-从数字系统到计算机.mp4

Why do?

◎Using the CPU to solve the problem

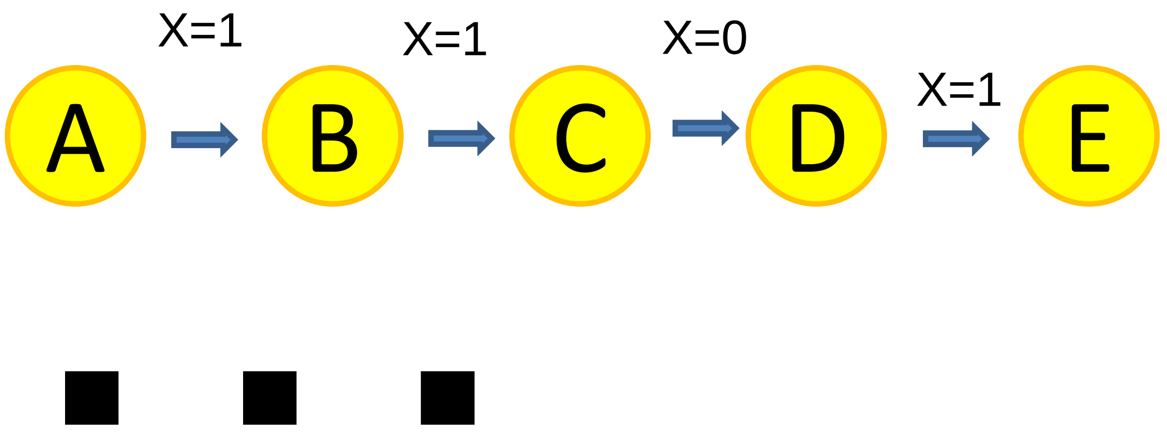
Design methodology 1: Common digital system

◎Finite state machine（FSM）



Detection successive occurrence "1101" number of times

◎How use to FSM ?

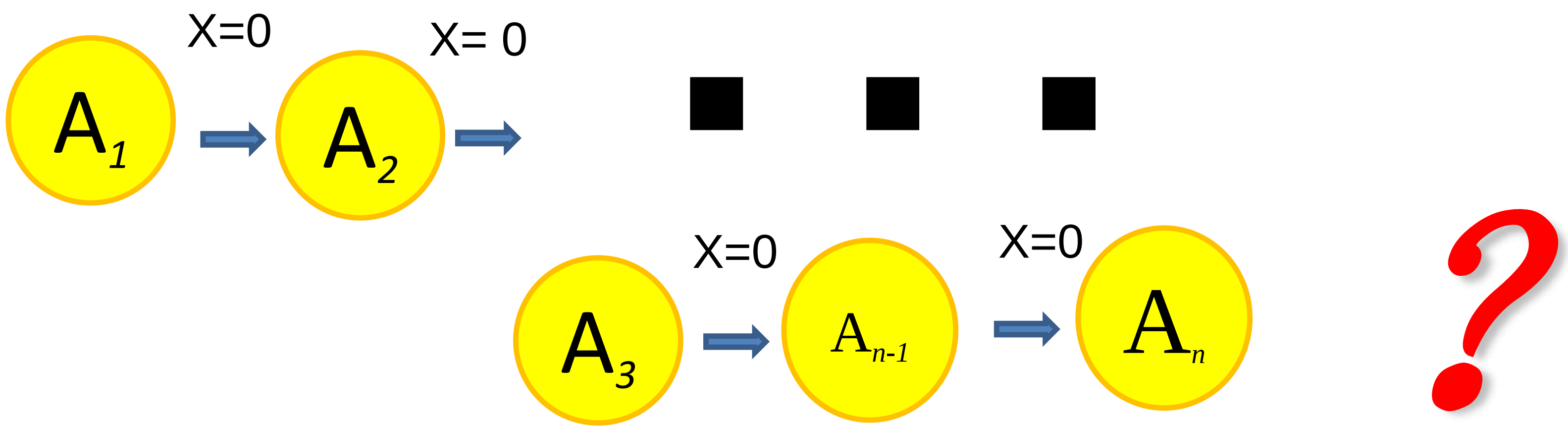


?

Uncertain state machine

Infinite state machine

Detection consecutive "0" number



To Isolate uncertainties

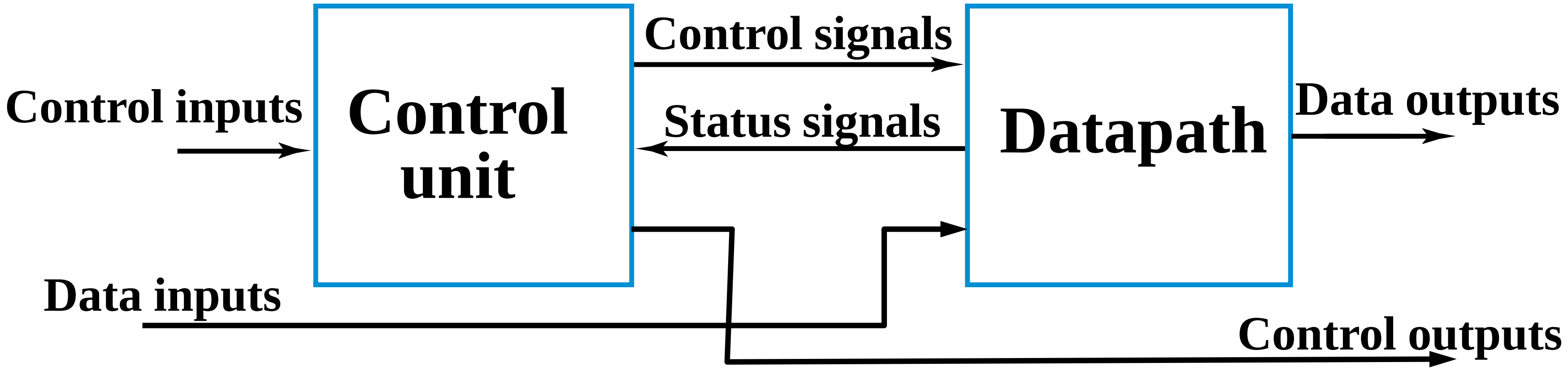
Design methodology 2: *Non-programmable* Control of Register Transfers

◎ Control of Register Transfers (传输状态机：RSM)

- Register transfers performed on registers Control that supervises the sequencing of the register transfers

◎ Three essential elements

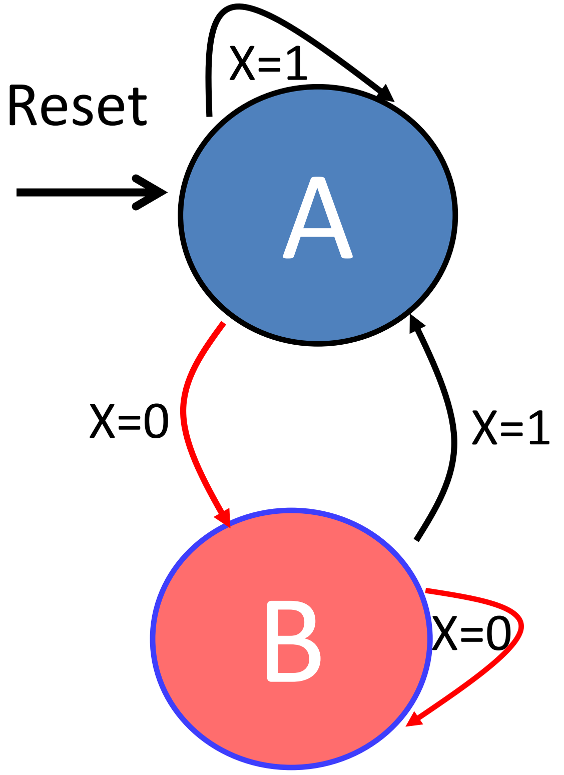
- Set of registers: mostly in Datapath with some in Control Unit
- Basic operation (micromanipulation): Register transfers performed
- Control: that supervises the sequencing of the register transfers

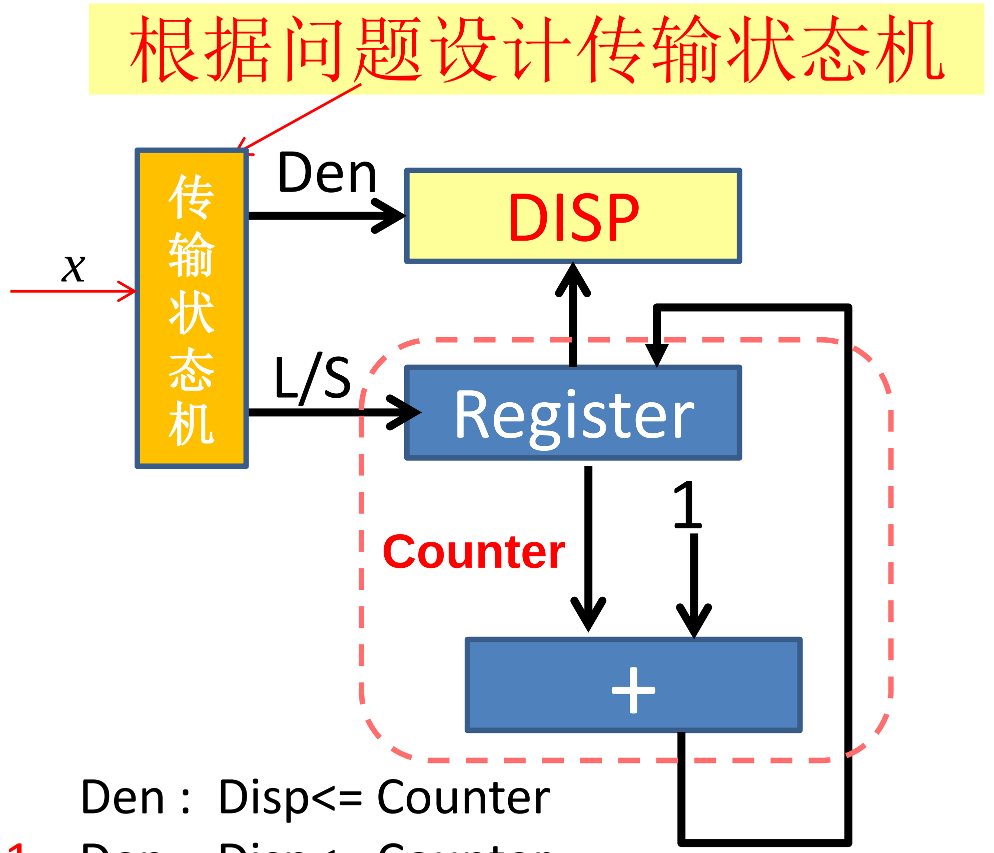


◎ Register Transfer Language: RTL

Transmission state machine processing

Detection consecutive "0" number



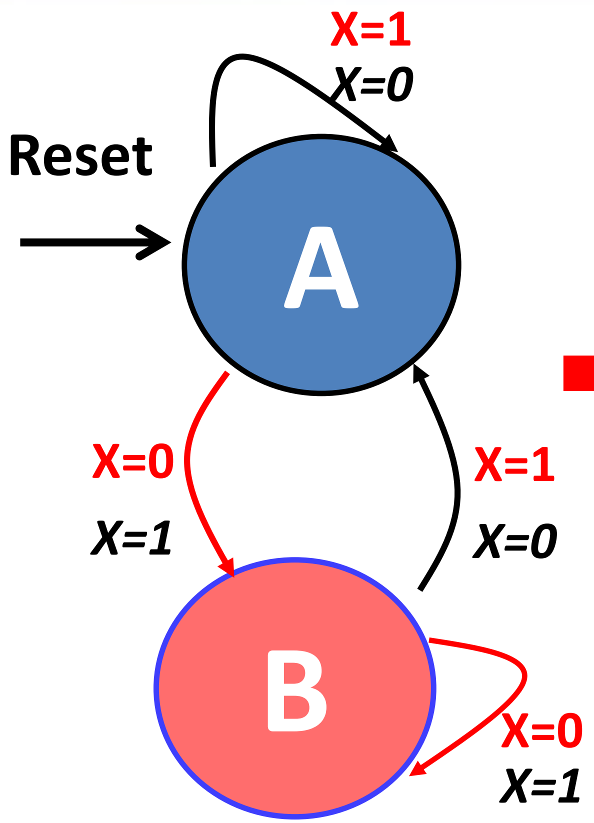


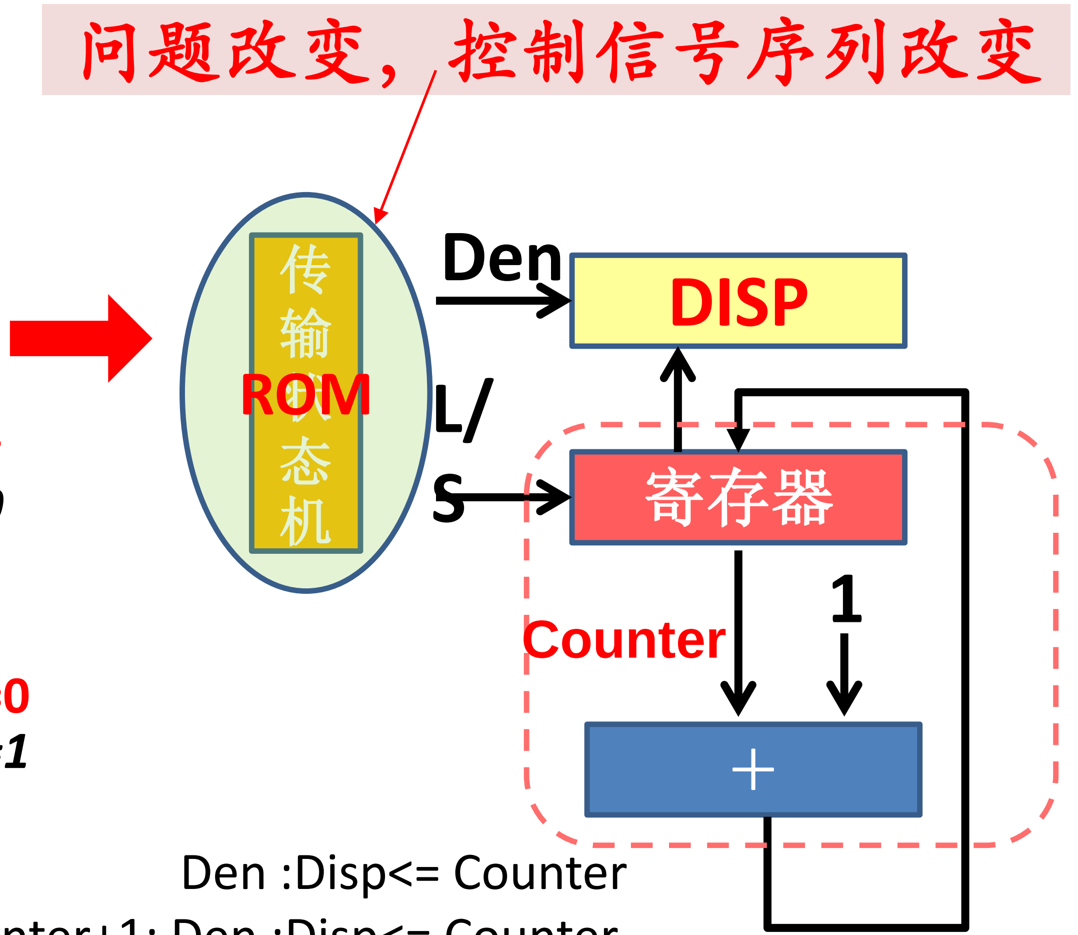
Reset: L/S:Counter<=0; Den : Disp<= Counter

X=0: L/S:Counter<= Counter+1; Den : Disp<= Counter

X=1: $\overline{\text{L/S}}$:Counter<= Counter; $\overline{\text{Den}}$: Disp<= Disp

Hardware Programmable: Detection consecutive “0/1" number





Reset： L/S:Counter<=0； Den :Disp<= Counter

X=0/1： L/S:Counter<= Counter+1; Den :Disp<= Counter

X=1/0： $\overline{\text{L/S}}$:Counter<= Counter； $\overline{\text{Den}}$ Disp<= Disp

Specific and general system

◎ **Programmable System : General system**

- a portion of the input consists of a sequence of *instructions* called a *program*,
- typically stored in a memory and addressed by a *program counter*.
- The Control Unit is responsible for fetching and executing these instructions.

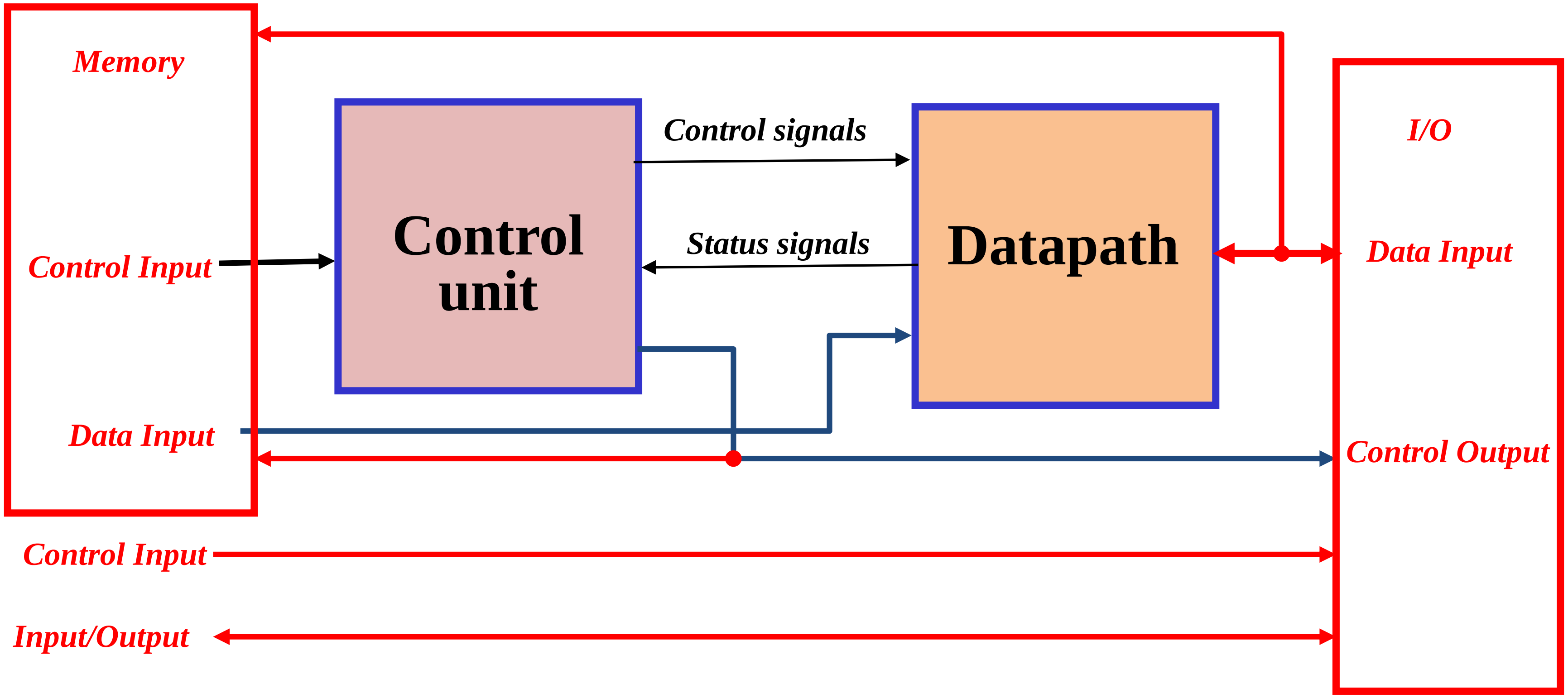
◎ **Non-programmable System: specific System**

- the control unit does not deal with fetching and executing instructions
- but contains all of the information for sequencing register transfers based on inputs and on status bits from the datapath

Design methodology 3: *Programmable* Control of Register Transfers

◎ General system

- Program state machine （PSM）



Is there a better way?

--Universal Turing machine thinking

考察图灵机

◎ **基本思想**

- 用机器来模拟人们用纸笔进行数学运算的过程
- 两种简单的动作
  - 纸上写上或擦除某个符号
  - 把注意力从纸的一个位置移动到另一个位置
- 下个动作依赖于
  - （a）此人当前所关注的纸上某个位置的符号
  - （b）此人当前思维的状态
- 图灵构造出一台假想的机器

图灵机的组成部分

1. **一条无限长的纸带TAPE**

   纸带被划分为连续的小格子，每个格子上包含一个来自有限集合（字母表）的符号，字母表中有一个特殊的符号□表示空白。纸带上的格子从左到右依次被编号为0，1，2，...，纸带的右端可以无限伸展

2. **一个读写头HEAD**

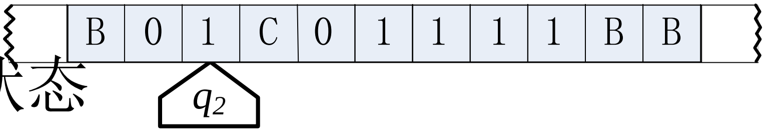
   该读写头可以在纸带上左右移动，它能读出当前所指的格子上的符号，并能改变当前格子上的符号

3. **一套控制规则TABLE**

   它根据当前机器所处的状态以及当前读写头所指的格子上的符号来确定读写头下一步的动作，并改变状态寄存器的值，令机器进入一个新的状态

4. **一个状态寄存器**

   它用来保存图灵机当前所处的状态



图灵机模型：马文•闵斯基 （1967）

图灵机指令结构

◎ **典型七元组（有多种形式的变体）**

- 典型的是一个七元组[3]，M={Q, Σ, Γ, δ, q0, qaccept, qreject}，其中 Q, Σ, Γ 都是有限集合

1. Q 是状态集合；
2. Σ 是输入字母表，其中不包含特殊的空白符“□”；
3. Γ 是字母表，其中 b∈Γ为空白符，且Σ∈Γ；
4. δ 是转移函数：Q × Σ→Q ×Γ{L,R}，其中L、R表示读写头是向左移还是向右移；
5. q0∈Q是起始状态；
6. qaccept是接受状态；
7. qreject是拒绝状态，且qreject≠qaccept。

图灵机程序

◎ 状态转移函数

- 程序

  δ={ qi Sj Sk R(L或N) ql }

  qi表示机器目前所处的状态;

  sj表示机器从方格中读入的符号;

  sk：表示机器用来代替Sj写入方格中的符号;

  R、L、N表示向右移一格、向左移一格或不移动;

  ql：表示下一步转移的机器状态。

- 例如：δ={q0 0 0 R q0；q0 1 0 R q0；}
  - 右移并抹去所有经过的纸带上的内容
  - 永不停止永不后退

图灵机程序δ:连续“1”检测

◎ 除起始状态q0外有7个内部状态

```
q0 B B R q1           //起始状态,读写头指向左边缘,次态为q1,如图 3-14 (a)。
q1 1 C L q2           //状态1,检测到“1”,标记C,开始计数,   如图 3-14 (b)。
q1 0 C L q7           //状态1,检测到“0”,标记C,开始清零,   如图 3-14 (c)。
q1 B B N q1(STOP)    //状态1,检测到右边缘停机,左边是检测结果, 如图 3-14 (d)。
q2 0 1 L q4           //状态2,计数。从右（低位）向左（高位）计数。
q2 1 0 L q2           //状态2,计数。从右（低位）向左（高位）计数。
q2 B 1 L q4           //状态2,到左边缘计数结束，次态是q4。
q4 1 1 L q4           //状态4,读写头移到左边缘。
q4 0 0 L q4           //状态4,读写头移到左边缘。
q4 B B R q3           //读写头到左边缘,开始计数值右移1位。
q3 1 0 R q5           //计数值将右移1位，左边填“0”。
q3 0 0 R q6           //计数值将右移1位，左边填“0”。
q3 C 0 R q1           //右移结束，清计数标志。
q5 1 1 R q5           //状态5，右移“1”。
q5 0 1 R q6           //状态5，右移“1”。
q5 C 1 R q1           //右移“1”并结束右移，清计数标志。
q6 1 0 R q5           //状态6，右移“0”。
q6 0 0 R q6           //状态6，右移“0”。
q6 C 0 R q1           //右移“0”并结束右移，清计数标志。
q7 1 0 L q7           //状态7，清零。
q7 0 0 L q7           //状态7，清零。
q7 B B R q3           //检测到左边缘，清零结束。转到状态3，计数值右移一位。
```

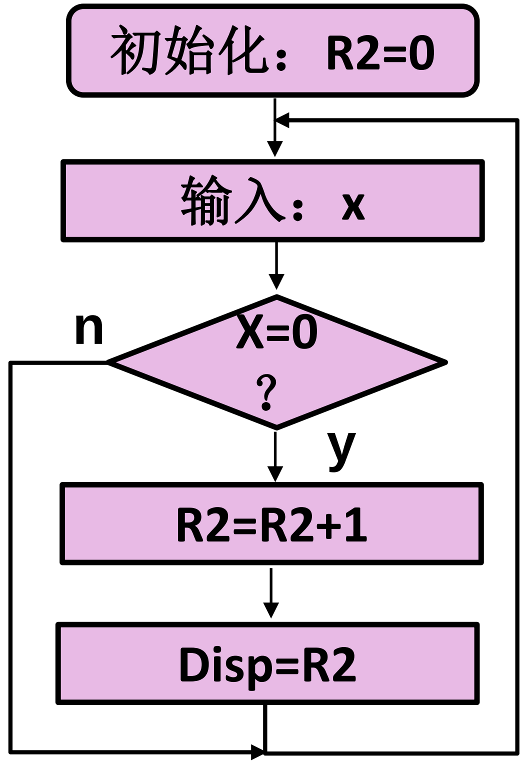
普适图灵机实现

- **理想图灵机是无法实现的**
  - 无法找到无限长的纸带
    - 也不是通用的计算系统
    - 只是固定内容机械的计算系统
- **普适(通用)图灵机**
  - 足够长的磁带并将其首尾相连来代替无限长纸带
  - 任意一台图灵机指令表编码
    - 写在**纸带/磁带**的开始部分
  - 作用于输入的余下部分
  - 效率依然很低
- **存在2个问题**
  - 载体
  - 效率

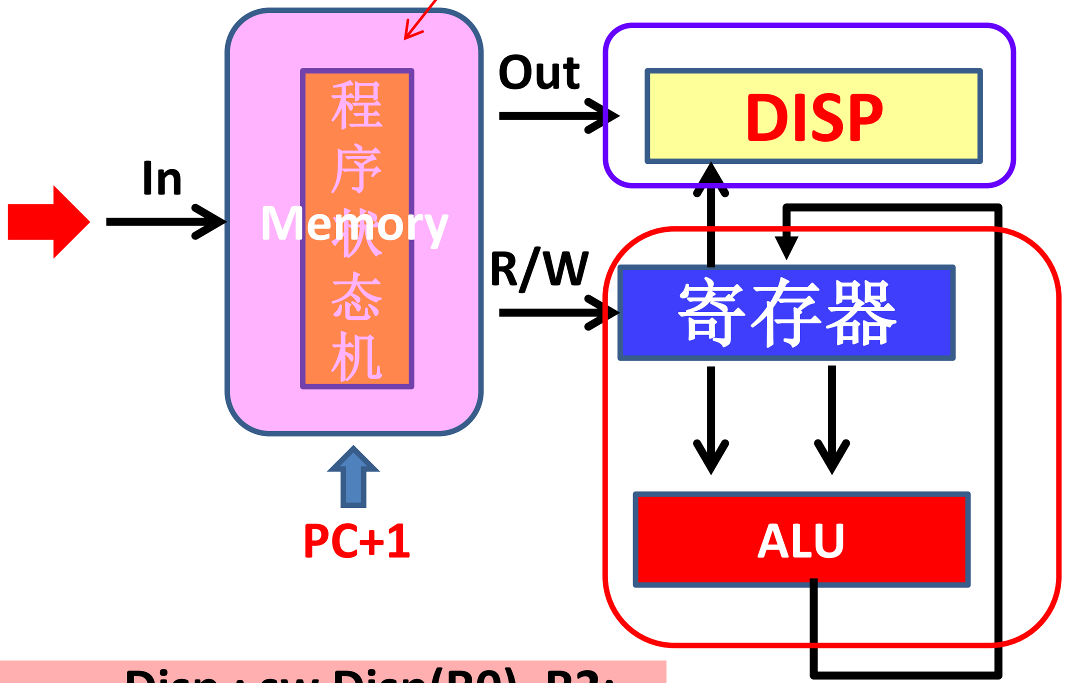
可实用的图灵机--真正意义上的计算机

- ◎ **一个足够大的存储器来代替磁带**
  - 磁芯存储器代替磁带
  - 半导体存储器代替磁带
- ◎ **保存内态**
  - 内态不保存计算的中间值，使得效率非常低下
  - 寄存器来存放中间值（相当于磁带的子集）
  - 可以极大地简化计算结构并提高计算效率
- ◎ **增强读写“头”处理能力--CPU**
  - 需要一个更强的读写头
  - 寄存器传输控制技术可以实现：CPU
  - 令人相当满意的通用图灵机

可编程寄存器传输技术实现通用普适图灵机



问题改变，程序状态机改变



```
Reset:   add R2, R0,R0;            Disp : sw Disp(R0), R2;
Input:   lw R3, InPort(X);
X=0:     addi R2, R0,1;            Disp :Sw Disp(R0), R2;
X=1:     Nop;
```

Conversion of thinking:
from digital systems turn to computer systems

- **CPU实现图灵机的核心思想**
  - 通用数字系统 图灵机的一种很好的实现方式
    - 存储器控制序列：解决无限长纸带
    - 三指令能做什么：读写头HEAD基本操作
    - 控制流指令：更丰富的控制规则TABLE
    - 寄存器：更丰富的状态存储和中间存储
  - 计算思维：程序解题
    - 设计算法：计算思维
    - 设计程序：软件思维

    } 转向CPU实现数字信息处理
    - 编程规则：指令→规则工作流程→指令的控制流程
      - 定义指令格式：设计DataPath和控制器
      - 定义数据结构：分配存储空间（Memory、I/O）
  - 软硬协同处理思想

CPU we will be doing

- **We'll look at an implementation of the RISC-V**
- **Simplified to contain only:**
  - memory-reference instructions: **lw, sw**
  - arithmetic-logical instructions: **add, sub, and, or, slt**
  - control flow instructions: **beq, j(jar)**
- **An Overview of the implementation**
  - For every insruction, the first two step are identical
    1. Fetch the instruction from the memory
    2. Decode and read the registers
  - Next steps depend on the instruction class
    - **Memory-refrernce Arithmetic-logical branches**

What are the steps?
How many functions

Computer Organization

- Computer
  - CPU
    - Control unit
    - Datapath
      - Path:
        multiplexors
      - ALU
      - Registers
      - ……
  - Memory
  - I/O interface

计算机系统设计— 想象和自由的空间

——给学生一个创新目标：渐进的计算机硬件系统

大型综合设计与实践

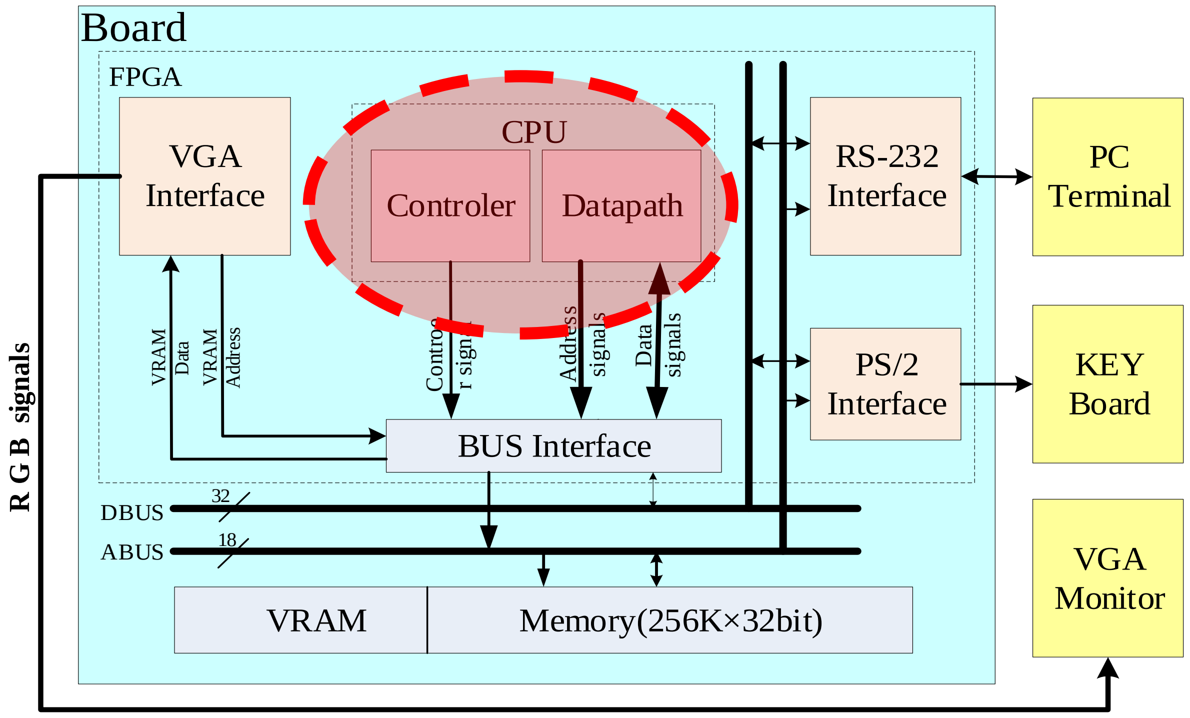
CACHE

MMU

动态流水

ISA扩展

中断处理



数字逻辑→计算机组成(CPU) →体系结构(流水CPU) →简易计算机系统

高级数字系统设计→**计算机接口** →SOC设计→计算机系统设计→**多核系统设计**

→嵌入式系统设计

Simple but rich

编译、**OS**

Teaching Prototype: RISC-V

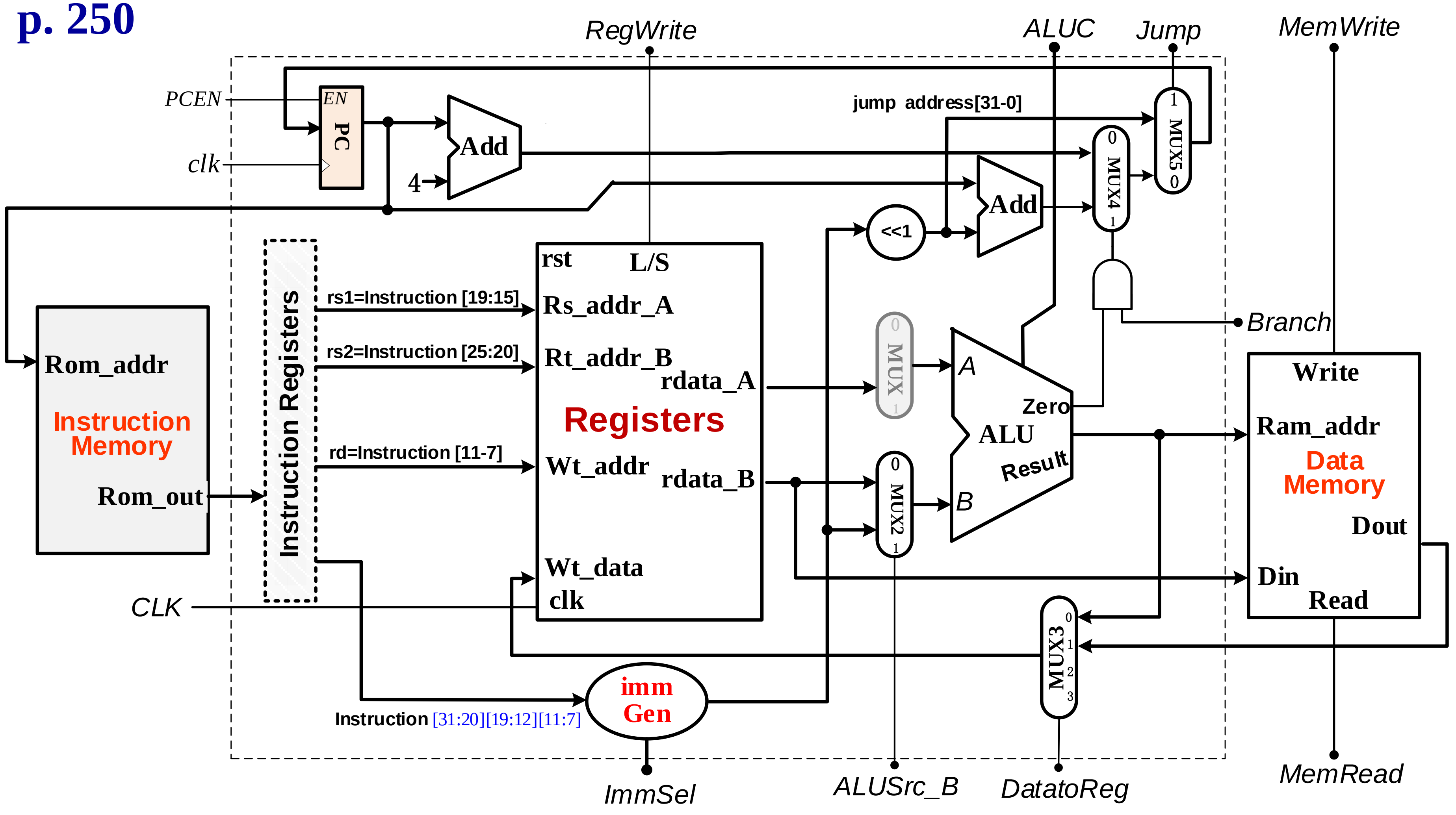
What is the MIPS his ancestor?

Microprocessor
without
Interlocked
Pipeline
Stages

What does ARM mean ?

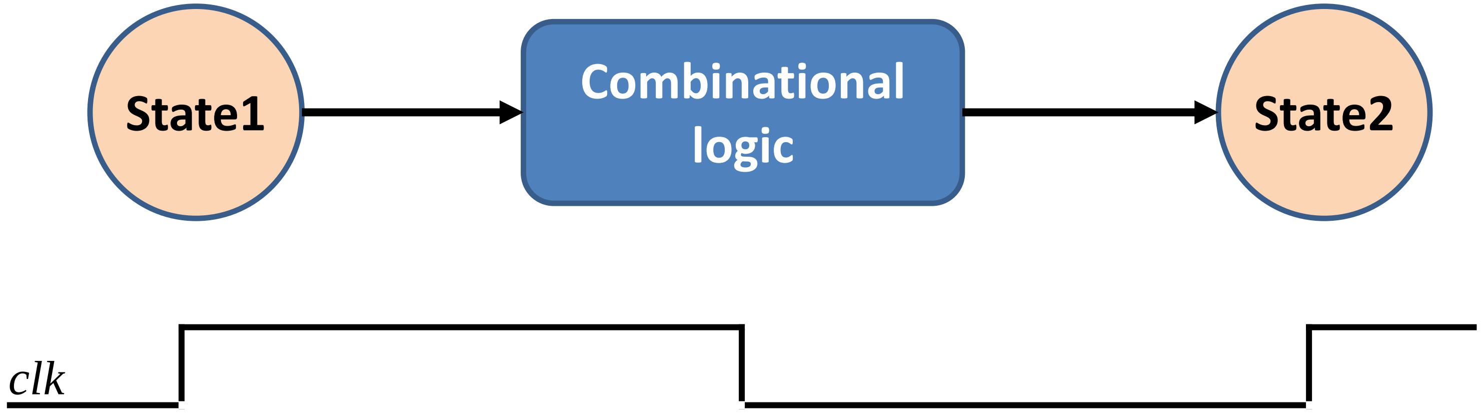
An abstract view of Simple implementation of RISC-V

p. 250



Simple Implementation

- **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



Logic circuit review

- ……

Contents of Chapter 4

Building a Datapath ……

Building With Common Elements

Including ALU & Registers

Watch the video：第3讲-处理器设计-单周期数据通路分析.mp4

RISC-V fields (format)

imm Region: ±2^{12}

Field size	31 7bits 25	24 5bit 20	19 5bit 15	14 3bits 12	11 5bits 7	6 7bits 0	All RISC-V instruction 32 bits
R	funct7	rs2	rs1	funct3	rd	**opcode**	Arithmetic instruction format
I	immediate[11:0]		rs1	funct3	rd	**opcode**	Loads & immediate arithmetic
S	immed[11:5]	rs2	rs1	funct3	immed[4:0]	**opcode**	Stores
SB	*imm[12,10:5]*	rs2	rs1	funct3	*imm[4:1,11]*	**opcode**	Conditional branch format
UJ	*i[20]* ④ *i[10:1],*	*i[11]*	② *imm[19:12]*		rd	**opcode**	Unconditional jump format
U	immediate[31:12]				rd	**opcode**	Upper immediate format

① ③

- **op:** *basic operation of the instruction, traditionally called the opcode.*
- **rd:** *destination register number.*
- **funct3:** *3-bit function code (additional opcode).*
- **rs1:** *the first register source operand.*
- **rs2:** *the second register source operand.*
- **funct7** *7-bit function code (additional opcode).*

Must bear in mind !

The datapath there are ……

Name	Example	Comments
32 registers	*x*0-*x*31	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.
*x*1(**ra**)	1	Return address(link register)	yes
*x*2(**sp**)	2	Stack pointer	yes
*x*3(**gp**)	3	Global pointer	yes
*x*4(**tp**)	4	Thread pointer	yes
*x*5-*x*7(**t0-t2**)	5-7	Temporaries	no
*x*8(**s0/fp**)	8	Saved/frame point	Yes
*x*9(**s1**)	9	Saved	Yes
*x*10-*x*17(**a0-a7**)	10-17	Arguments/results	no
*x*18-*x*27(**s2-s11**)	18-27	Saved	yes
*x*28-*x*31(**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.

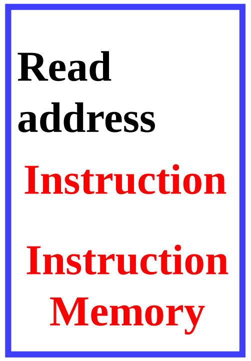
Instruction execution in RISC-V

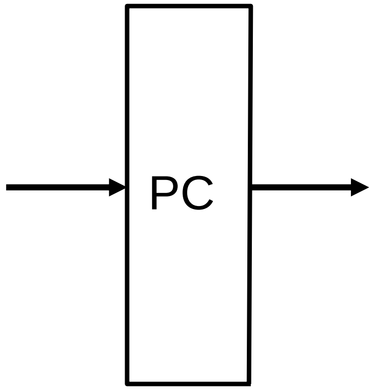
- **Fetch：**
  - Take instructions from the instruction memory
  - Modify PC to point the next instruction
- **Instruction decoding & Read Operand:**
  - Will be translated into machine control command
  - Reading Register Operands(whether or not to use)
  - Read Immediate and perform 32 bit extension(whether or not to use)
    - Bus transfer operation only
- **Executive Control:**
  - Control the implementation of the corresponding ALU operation
- **Memory access:**
  - Write or Read data from memory (Only L/S：ld/lw、sd/sw)
- **Write results to register:**
  - All results are written to rd
  - Modify PC： for all instructions(PC+4、branch and jal)

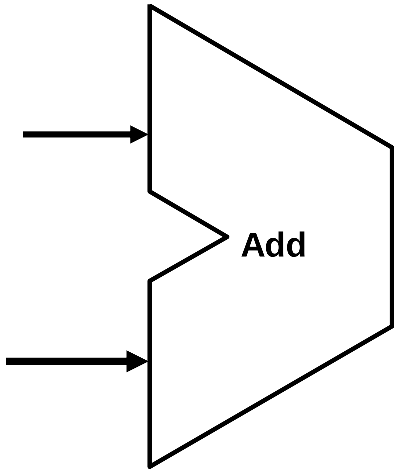
Instruction fetching three elements

Data Stream of Instruction fetching

How to connect? Who?



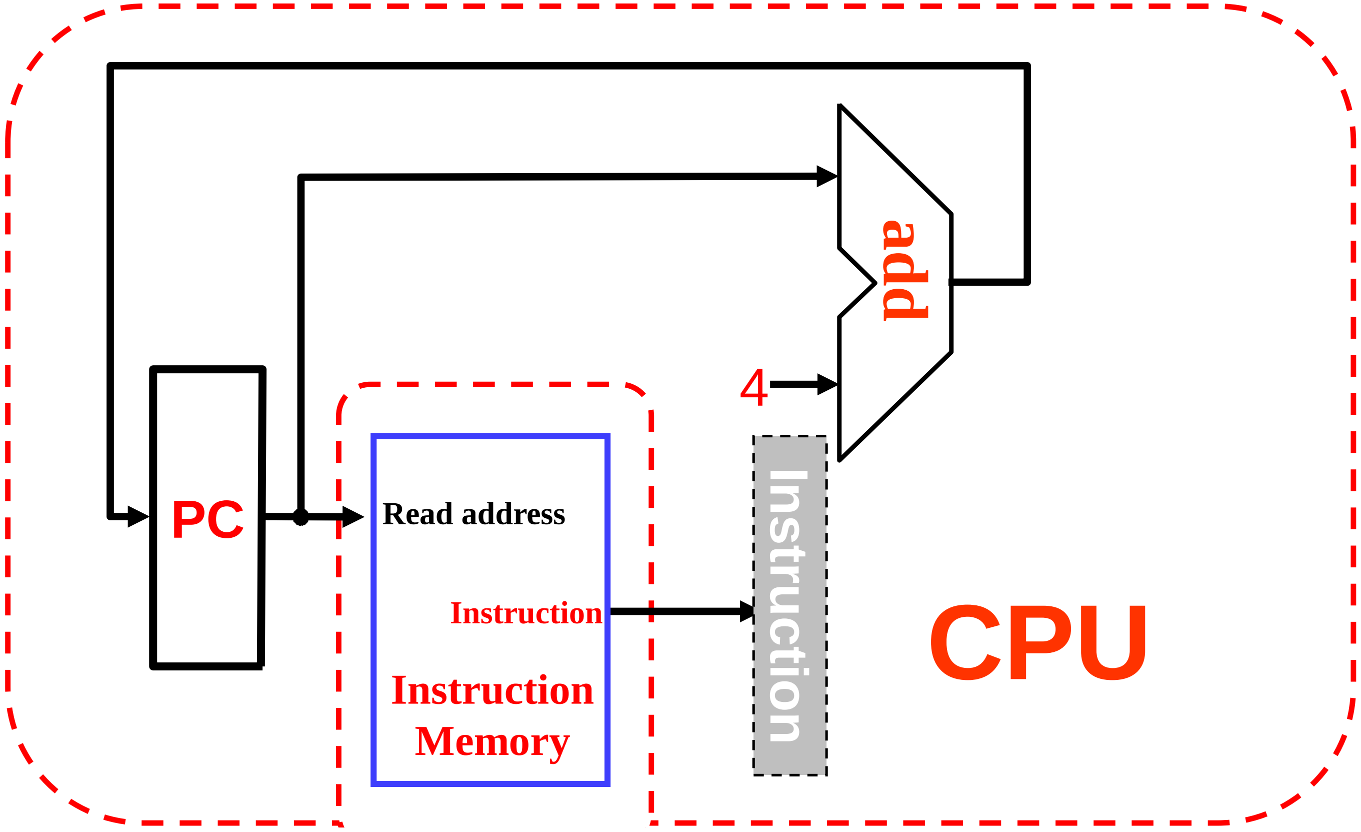




Instruction memory **Program counter** **Adder**

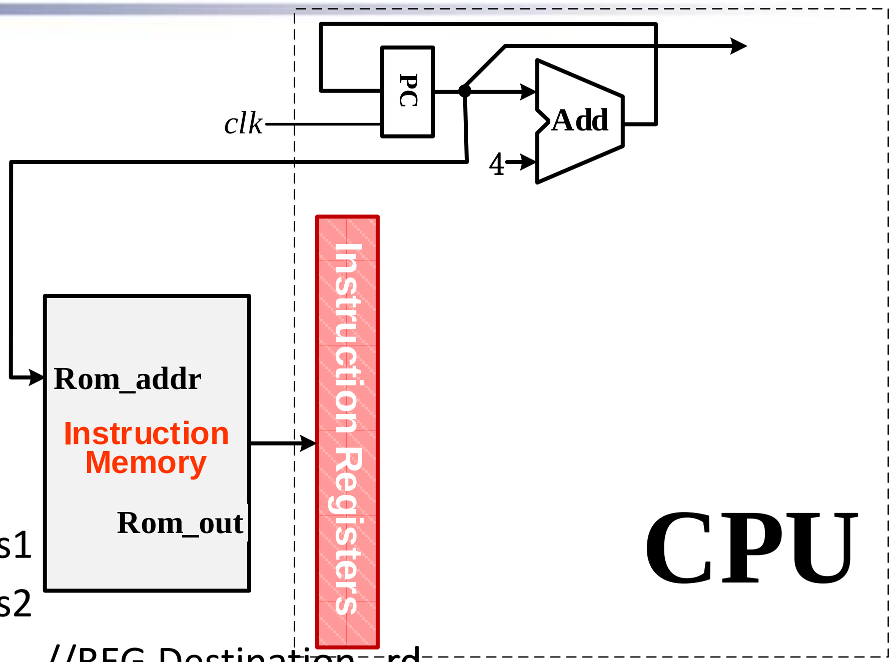
Instruction fetching unit

- Instruction Register
  - Can you omit it?



How simple is!

Why PC+4?

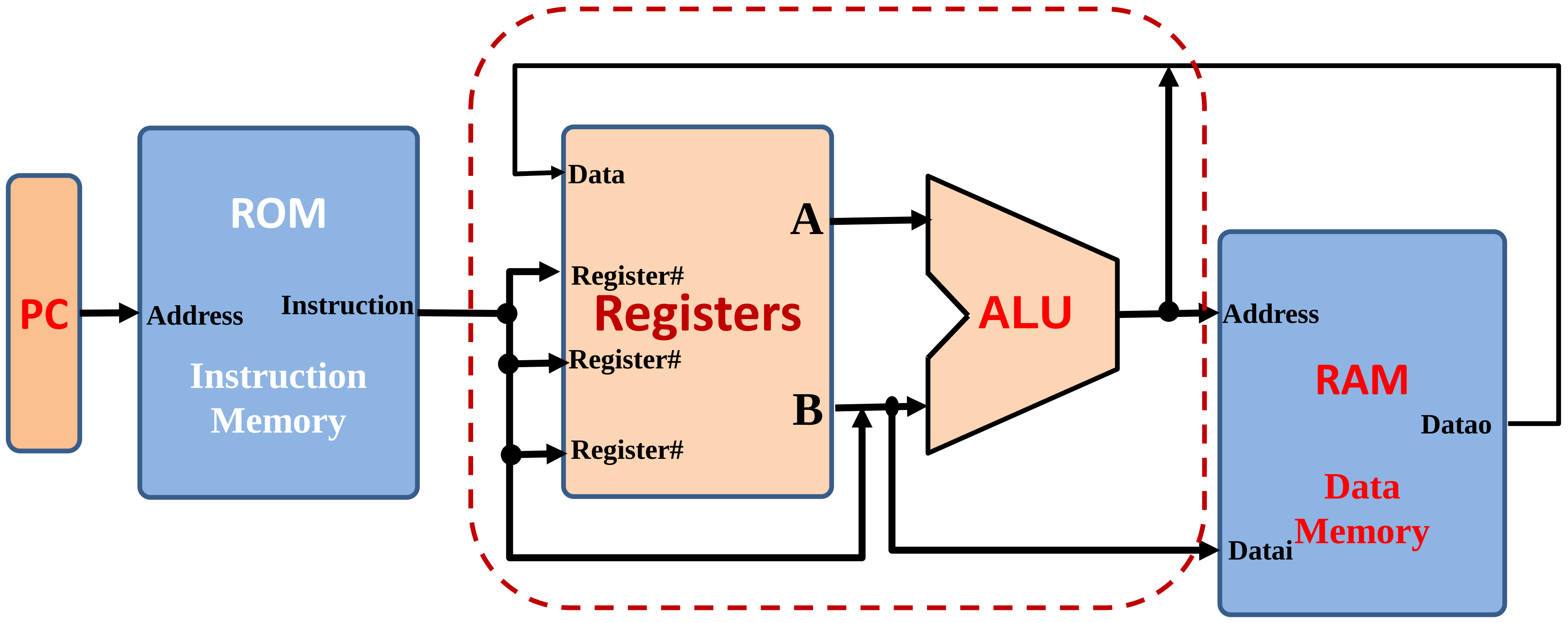


```
wire[31:0] Rom_Addr = PC;
                                  //指令存储器地址通路
wire[31:0] inst_field = Rom_out;
                                  //指令存储器数据通路

wire[31:0] PC_4 = PC + 4;           //修改PC指针
wire[4:0] rs1 = inst_field[19:15];  //REG Source 1 rs1
wire[4:0] rs2 = inst_field[24:20];  //REG Source 2 rs2
wire[4:0] rd =  inst_field[11:7];                         //REG Destination  rd
wire[12:0] imm12 = inst_field[31:20];                     //12位Immediate
wire[12:0] ims12 = inst_field[31:25],inst_field[11:7]     //12位store偏移立即数(地址)
wire[13:0] imsb12 = inst_field[31],inst_field[7],inst_field[30:25],inst_field[11:8],1'b0
                                                          //13位branch偏移立即数(地址)
wire[13:0] imj20 =inst_field[31],inst_field[19:12],inst_field[20],inst_field[30:21],1'b0
                                                          //21位Jump长偏移立即数(地址)
wire[13:0] imu20 = inst [31:12], 12'h0                    //20位高位Immediate
```

More Implementation Details

◻Abstract / Simplified View:



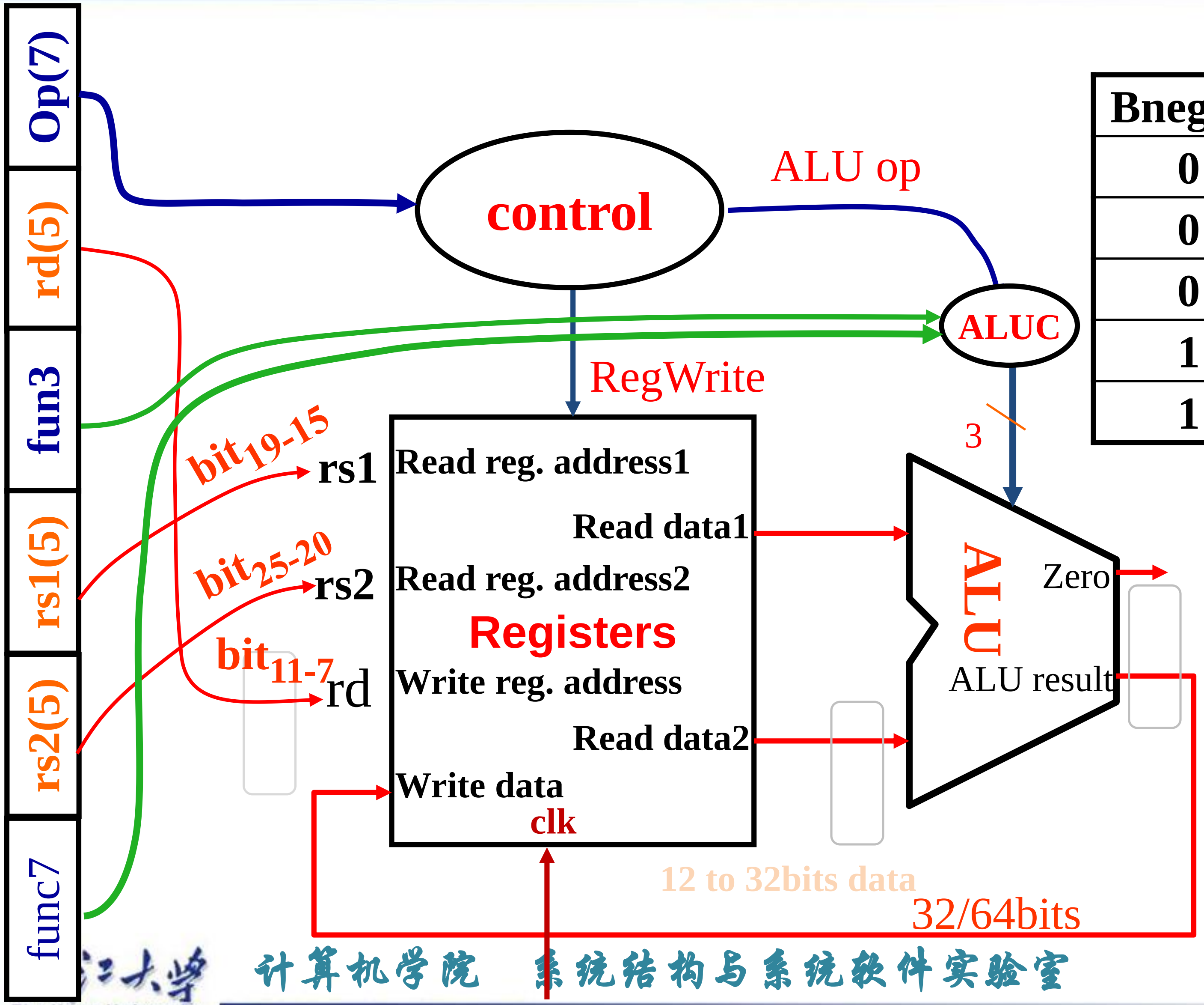
Data Stream of Instruction executing

- **R-type instruction Datapath**
- **I-type instruction Datapath**
  - For ALU、For Load memory
- **S-type (store) instruction Datapath**
- **SB-type instruction Datapath**
  - For branch
- **UJ-type instruction Datapath**
  - For Jal(Jump)
- **First:**

  Look at the data flow within instruction execution

R type Instruction & Data stream

add s1, s4, s5



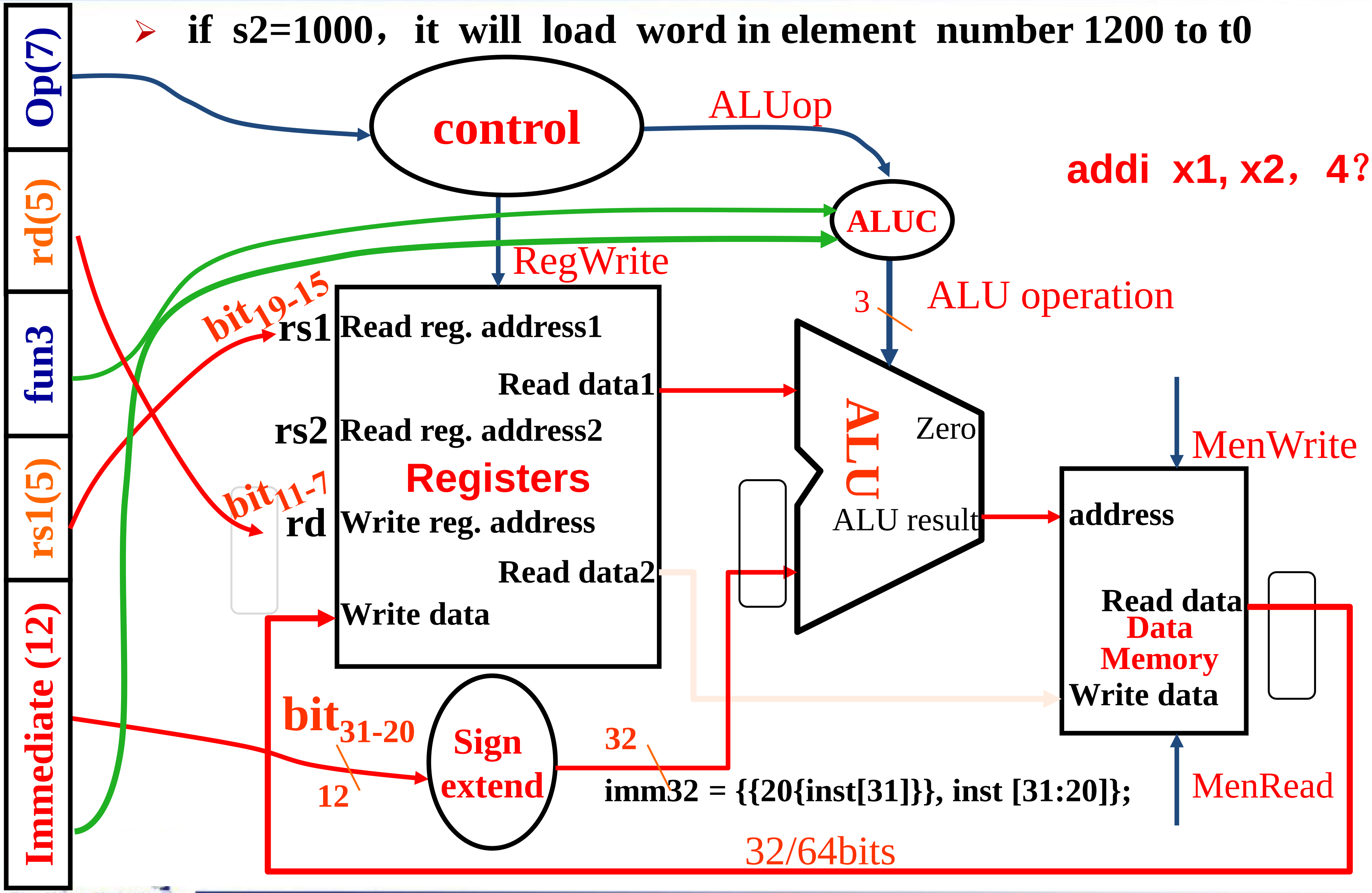
Bnegate	op	function
0	00	and
0	01	Or
0	10	Add
1	10	Sub
1	11	Slt

I type Instruction & Data stream

lw t0, 200(s2)

- if s2=1000， it will load word in element number 1200 to t0

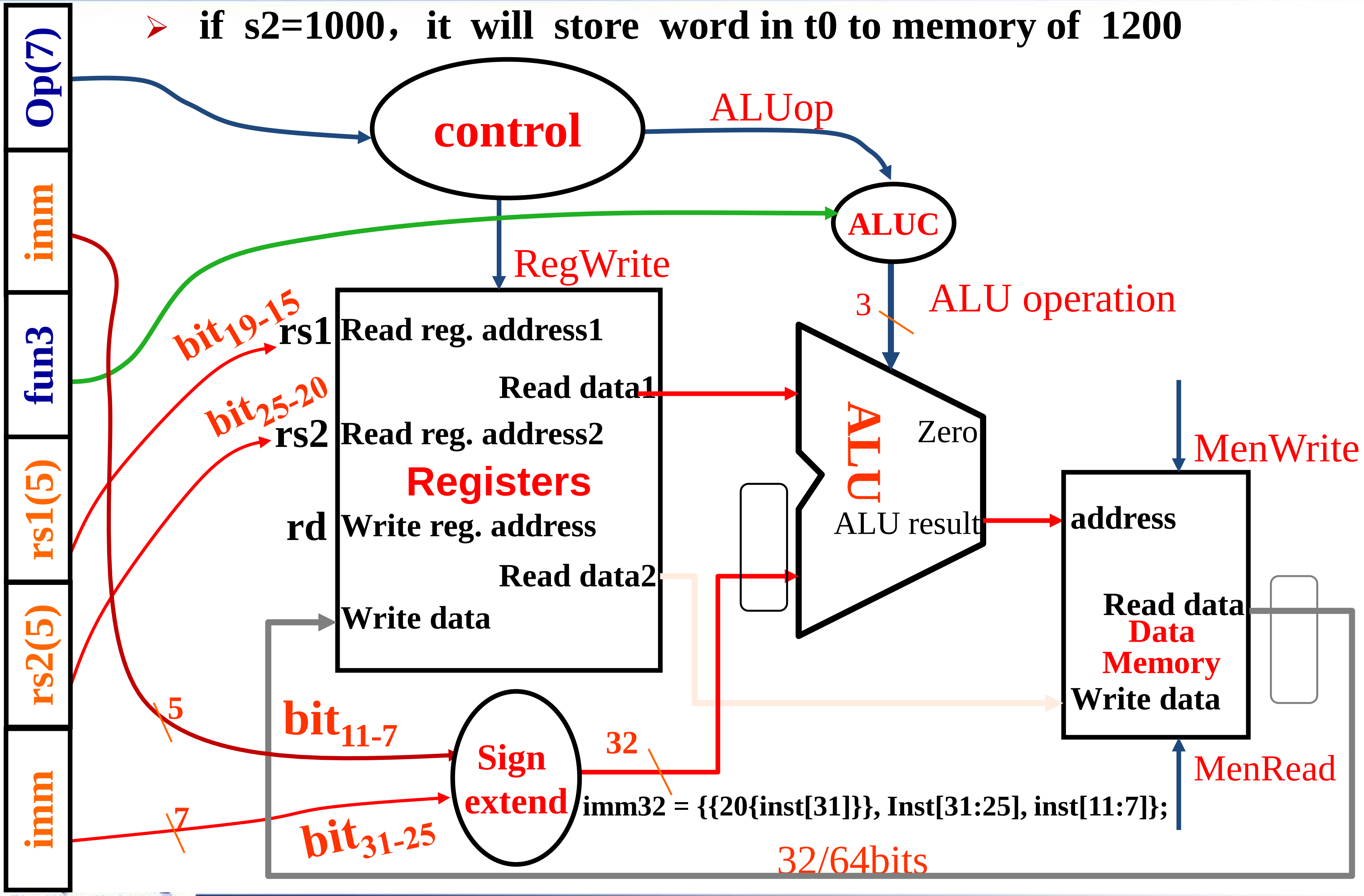
addi x1, x2， 4?



S type Instruction & Data stream

sw t0, 200(s2)

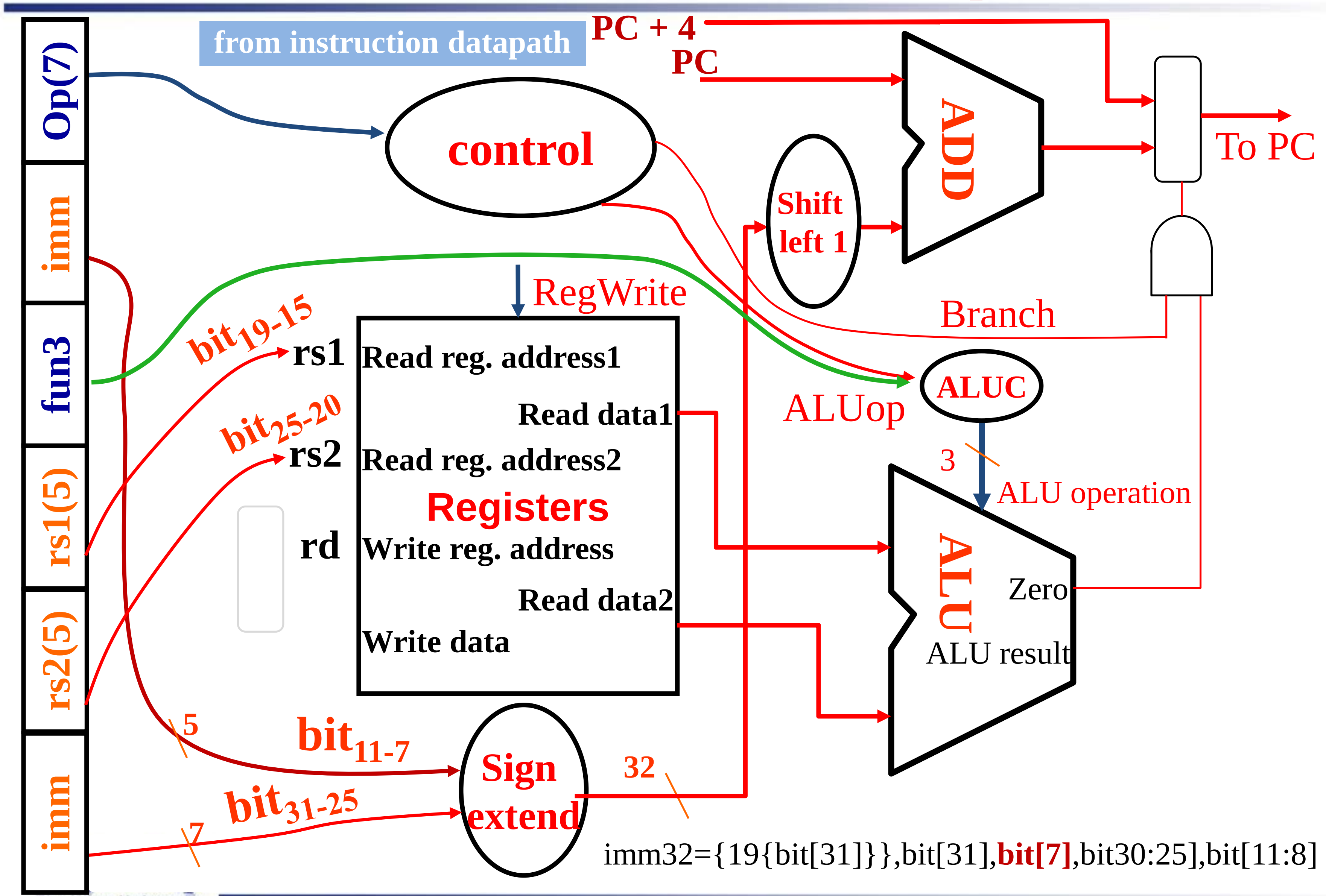
- if s2=1000，it will store word in t0 to memory of 1200



imm32 = {{20{inst[31]}}, Inst[31:25], inst[11:7]};

SB type Instruction & Data stream

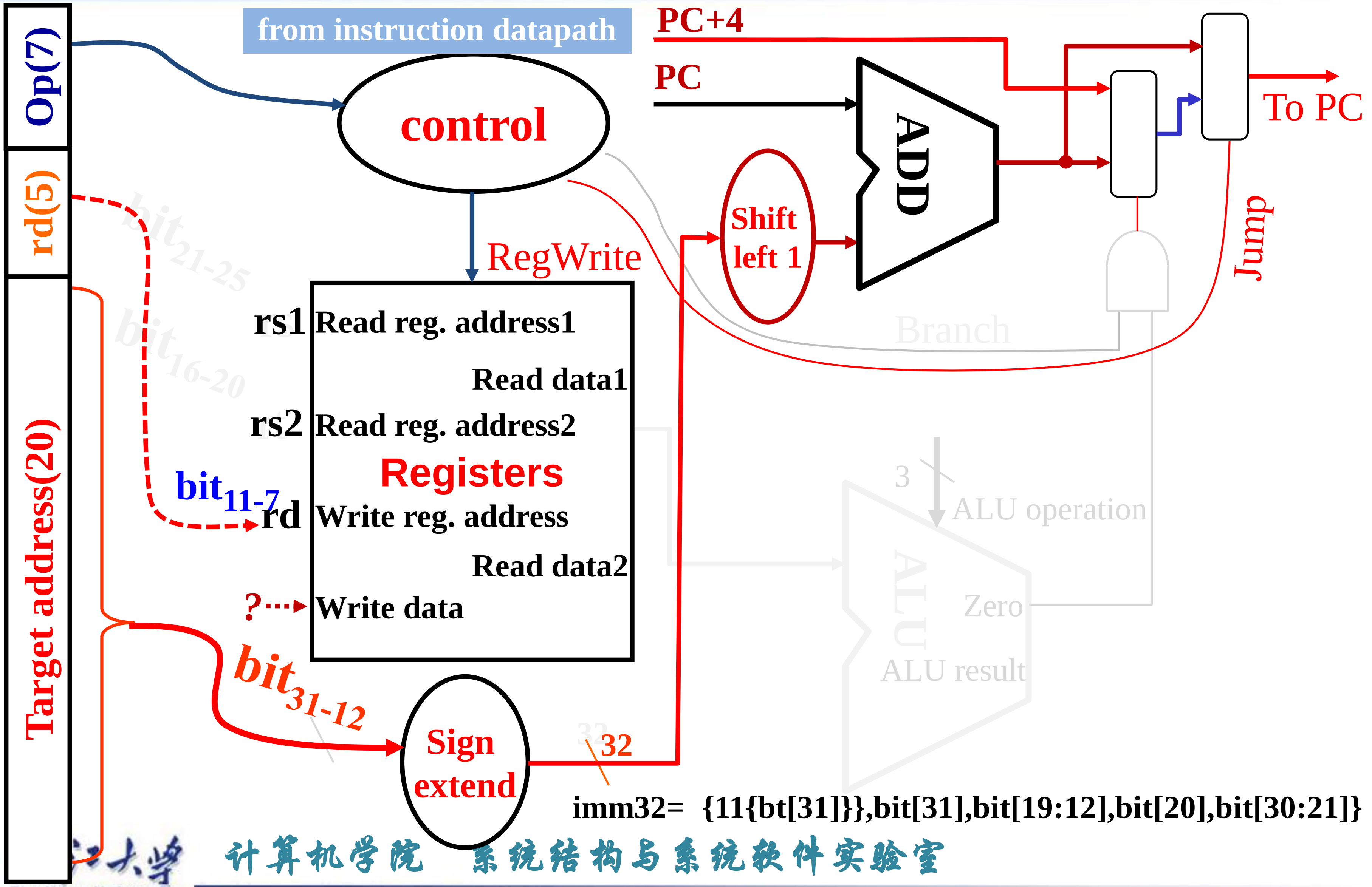
beq t0, t2， 200



imm32={19{bit[31]}},bit[31],**bit[7]**,bit30:25],bit[11:8]

J/Jal type Instruction & Data stream

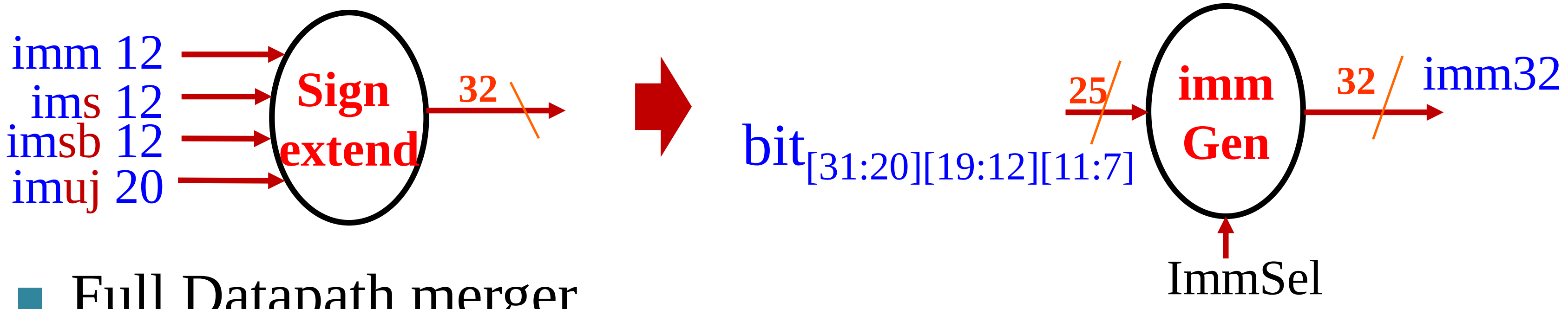
jal x0, Loop



imm32= {11{bt[31]}},bit[31],bit[19:12],bit[20],bit[30:21]}

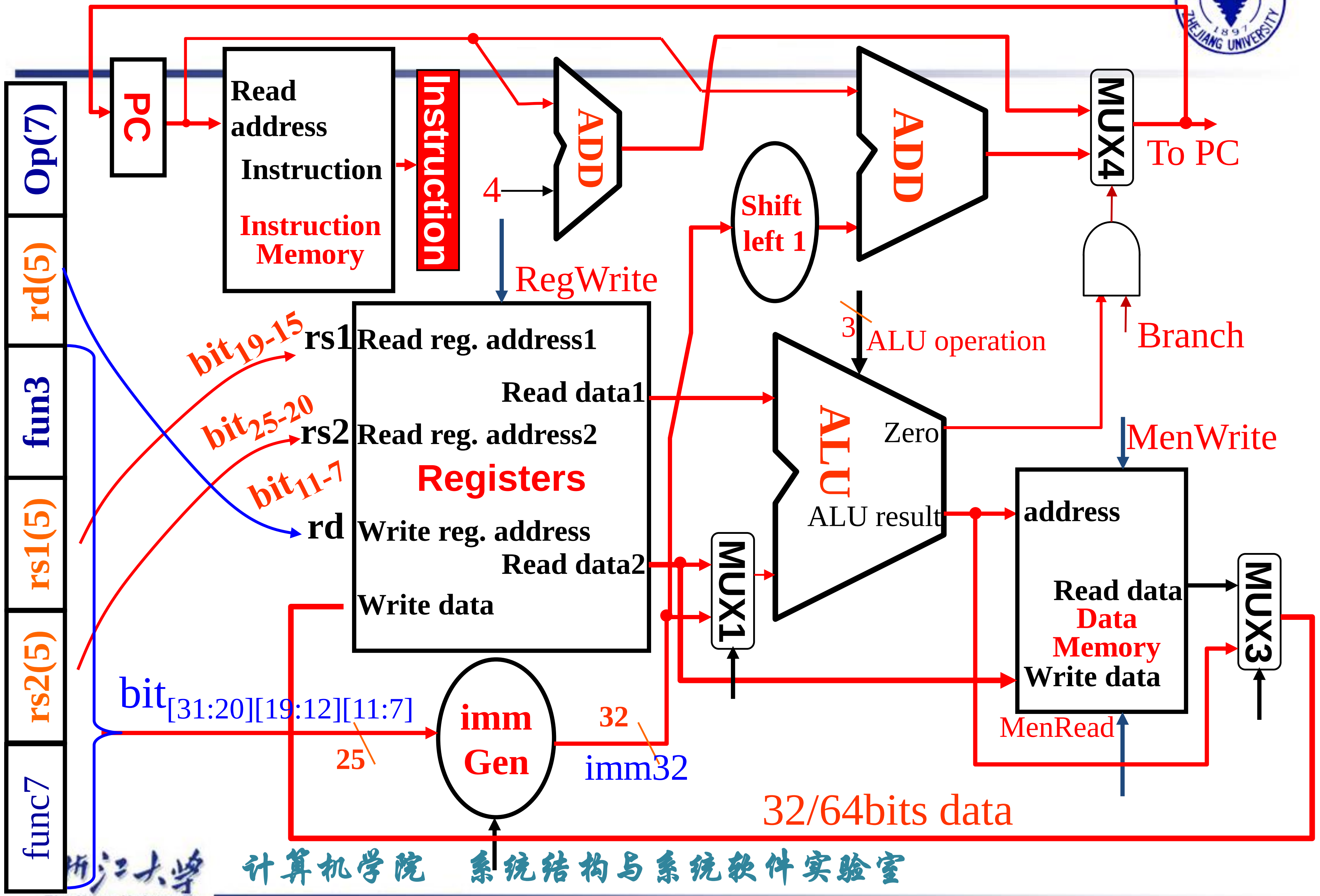
Composing the Elements

- **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**
  - Immediate datapath merger：I、B、S、SB、UJ

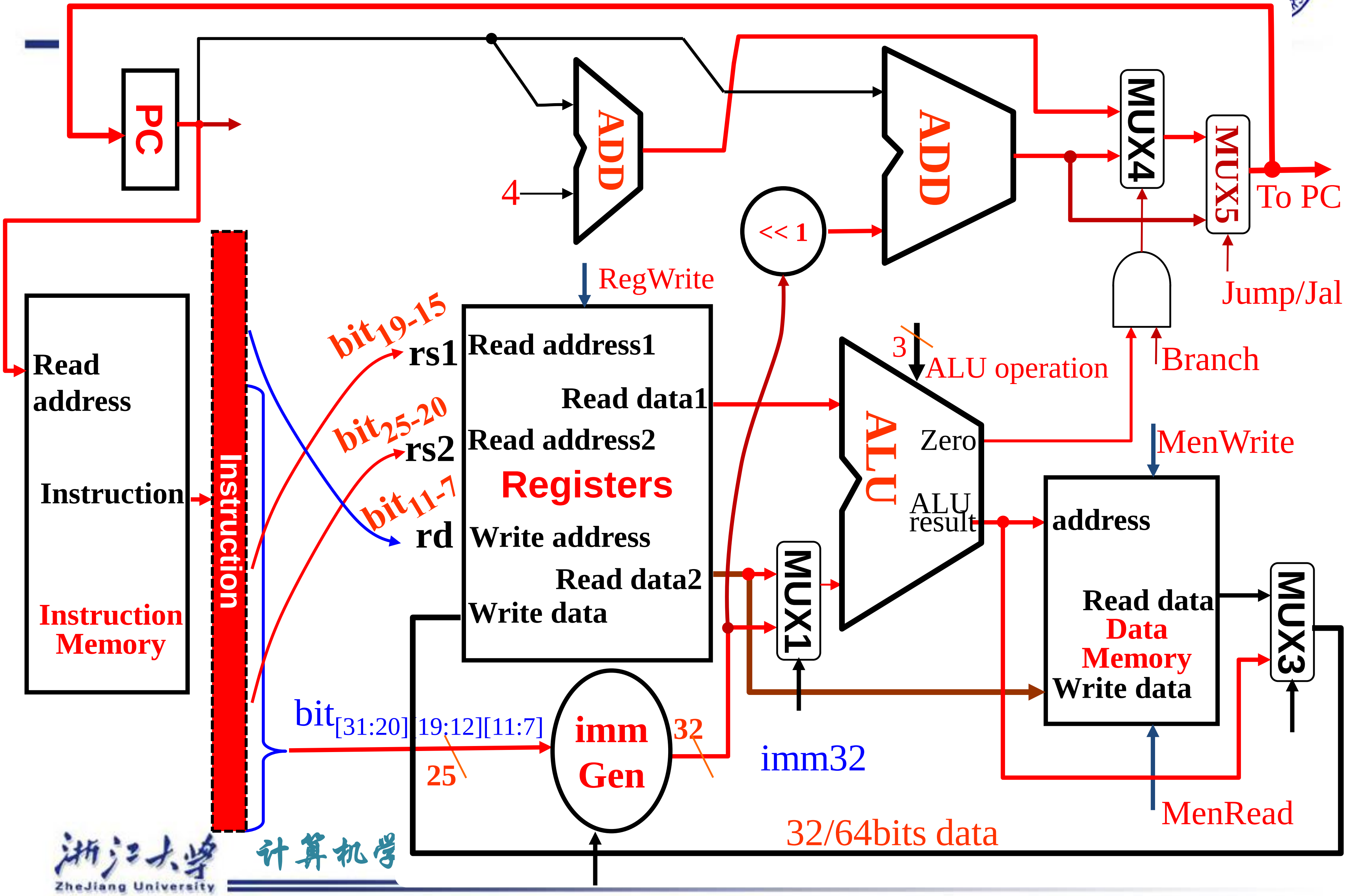


  - Full Datapath merger

Integrating Datapath：R-Type/Load/Store/SB



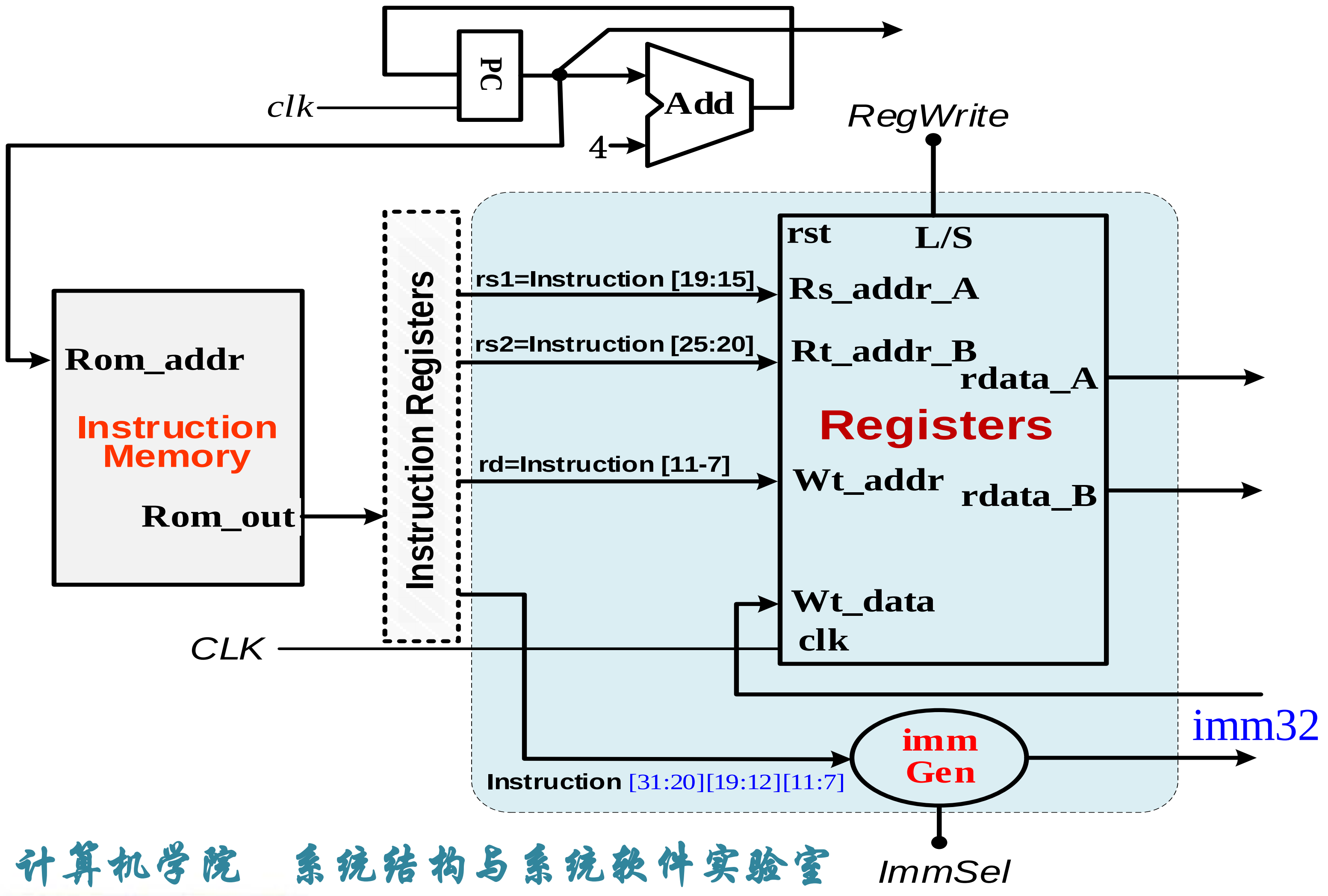
Full Datapath: Control & Data Steam Perspective



Analysis from the Path Perspective

Path Built Using Multiplexer

- Path： Read & Write Regs

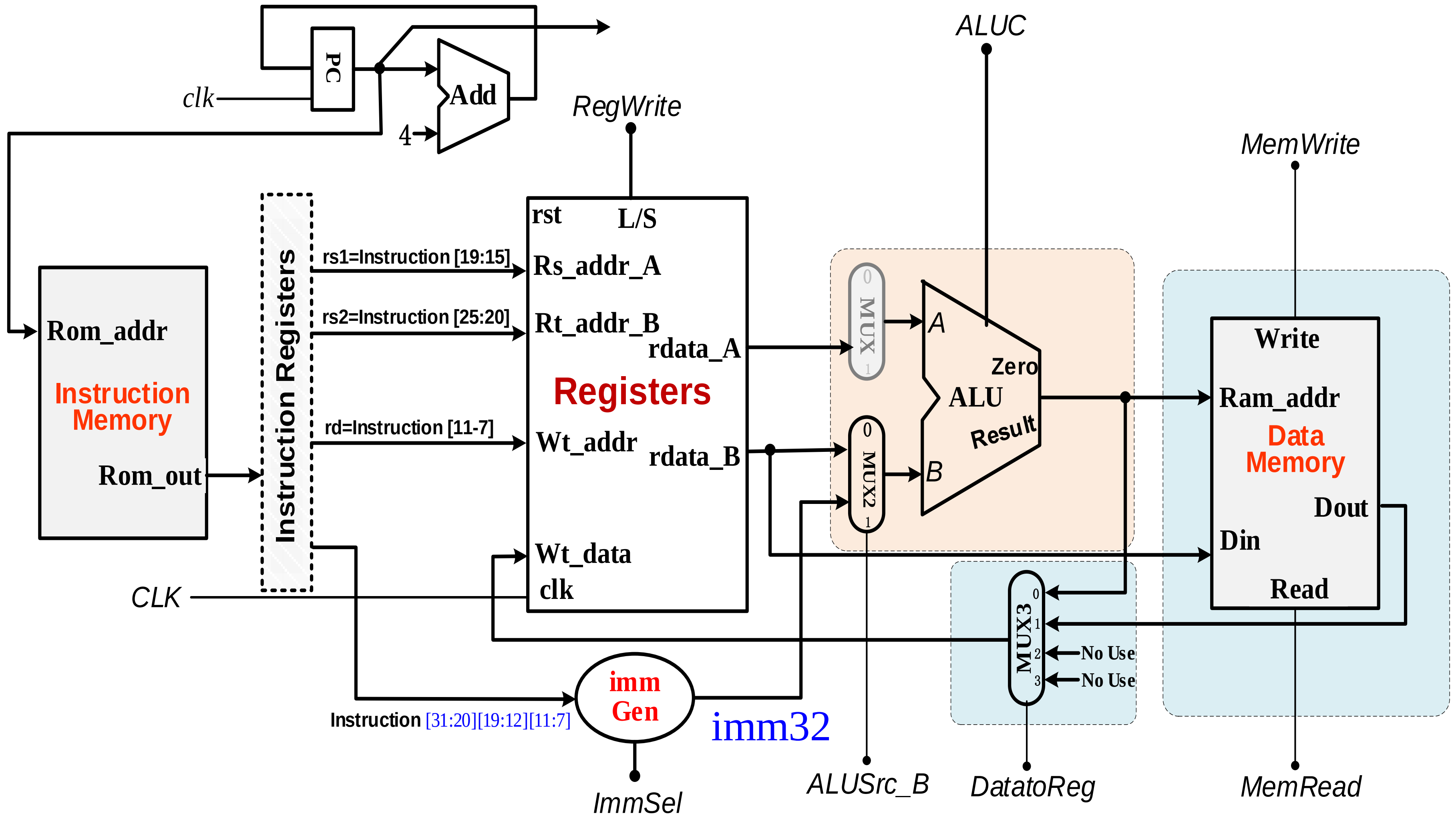


HDL description：Read & Write Regs

```
//寄存器读通路
    wire[4:0] Reg_addr_A = rs1;                    wire[4:0] Reg_addr_B = rs2;
    wire[31:0]rdata_A, rdata_B;
//寄存器写通路
wire[4:0] Wt_addr = rd;
//寄存器写数据通路(也是存储器读和PC保护通道。注意：不要重复，后继要扩展)

 wire[31:0] Wt_data = MemtoReg ? Data_in : ALU_out;
//立即数通道选择通路
     MUX4T1_32   ImmGem(.s(ImmSel),        //ImmGen
                        .I0({{20{inst_field[31]}},inst_field[31:20]}),          //addi\lw(I):    I-Type
                        .I1({{20{inst_field[31]}},inst_field[31:25],inst_field[11:7]}), //sw(s): S-Type
                        .I2({{19{inst_field[31]}},inst_field[31],inst_field[7],
                           inst_field[30:25],inst_field[11:8],1‘b0}),          //beq(b):       B-Type
                        .I3({{11{inst_field[31]}},inst_field[31],inst_field[19:12],
                           inst_field[20],inst_field[30:21],1'b0}),            //jal(j) :      BJ-Type
                        .o(Imm32));        //another imm：U_imm = {inst [31:12], 12’h0}   U-Type
//寄存器调用
    Regs                reg_files(.clk(clk), .rst(rst),
                             .R_addr_A(Reg_addr_A), .R_addr_B(Reg_addr_B),
                             .Wt_addr(Wt_addr),       .Wt_data(Wt_data),
                             .L_S(RegWrite),
                             .rdata_A(rdata_A),       .rdata_B(rdata_B));
```

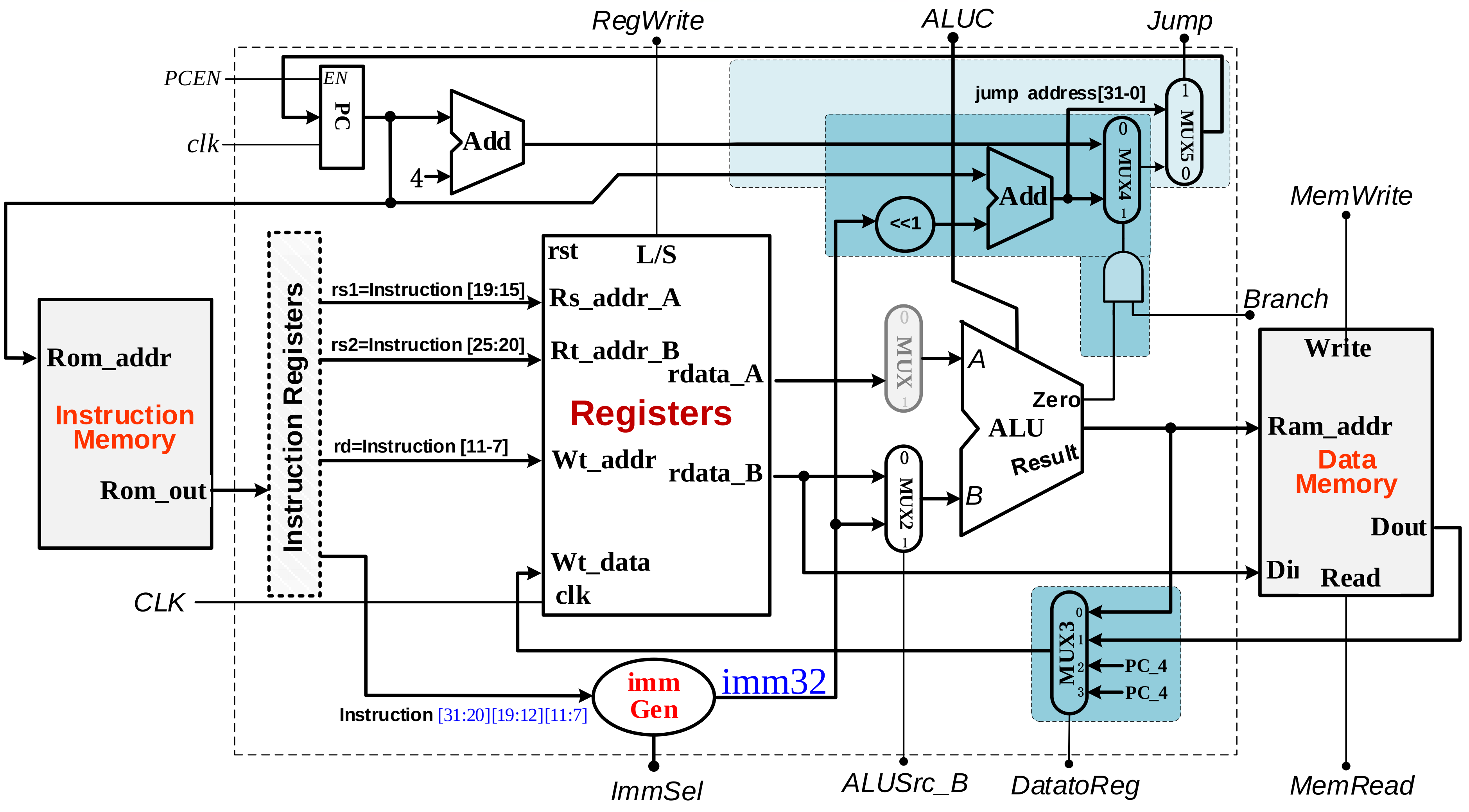
Path: ALU Operating & Memory



HDL description：ALU Operating & Memory

```
//ALU输入通路
wire [31:0]ALU_B;
  MUX2to1_32        MUX2 (.a(imm32),                              //ALU操作数B选择通路
                          .b(rdata_B),
                          .sel(ALUSrc_B),        wire[4:0] wire[31:0] ALU_B = ALUSrc_B ? imm32 : rdata_B;
                          .o(ALU_B));
//ALU输出通路
wire [31:0]Result;
  wire[31:0] ALU_out = Result;
  alu             ALU(.A(rdata_A),          .B(ALU_B),            //ALU调用
                      .ALUC(ALUC),
                      .res(Result),         .zero(zero),
                      .overflow(overflow) );
//存储器通路
  wire[31:0] Data_out = rdata_B;                                  //Datapath(CPU)输出
  wire[31:0] Din = Data_out;                                      //存储器数据输入
  wire[31:0] Ram_addr = ALU_out;                                  //存储器地址输入
//存储器读通路(也是寄存器写数据通路，不必重复)
  MUX4T1_32         MUX3(.sel(DatatoReg),
                         .I0(ALU_out),                            //寄存器写数据选择：ALU
                         .I1Data_in),                             //寄存器写数据选择：存储器
     扩展到四路          .I2(PC_4),                               //寄存器写数据选择：ra：PC+4
                         .I2(PC_4),                               //寄存器写数据选择：ra：PC+4
                         .o(Wt_data));                            //写入rd寄存器
```

Path： Control Stream-SB & UJ



HDL description： PC Path

```
//Branch通路
    wire[31:0] Branch_PC = PC + {imm32[30:0], 1'b0};
//Jump通路
    wire[31:0] Jump_PC   = {imm32[30:0], 1'b0};
//PC通路合成
wire[31:0] PC_next;
    mux4to1_32          MUX4(.a(PC_4),                          //PC通路，也可用case语句
                             .b(Branch_PC),                     //MUX4、MUX5合并
                             .c(Jump_PC),
                             .d(),
                             .sel({Jump,zero&Branch}),
                             .o(PC_next));
//PC寄存器写描述
reg[31:0]PC;
    always @(posedge clk or posedge rst ) begin
                    if (rst==1) begin                            //reset
                        PC <= 32'h00000000;
                    end
                    else if(PCEN)PC <= PC_next;  else PC <= PC;
    end
```

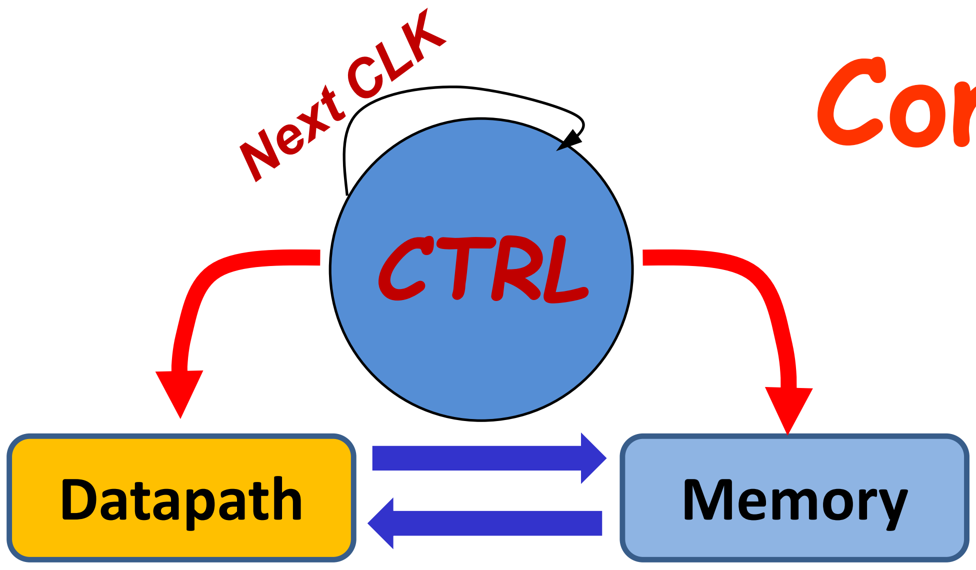
Signals for datapath

Defined 8+2 control (p. 256)

Signal name	Effect when deasserted(=0)	Effect when asserted(=1)
ImmSel	=00： imm32 =01： imm32=ims32	=10： imm32=imsb32 =11： imm32=imuj32
RegWrite	None	The register on the Write register input is written with the value on the Write data input.
ALUScrB	The second ALU operand come from the second register file output (Read data 2)	The second ALU operand is the sign-extended, 12 bits of the instruction.
PCSrc (Branch)	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.
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 DatatoReg	The value fed to register Write data input comes from the Alu	The value fed to the register Write data input comes from the data memory.
Jump	None	The PC is replaced by Jump(Jal) target

Contents of Chapter 4

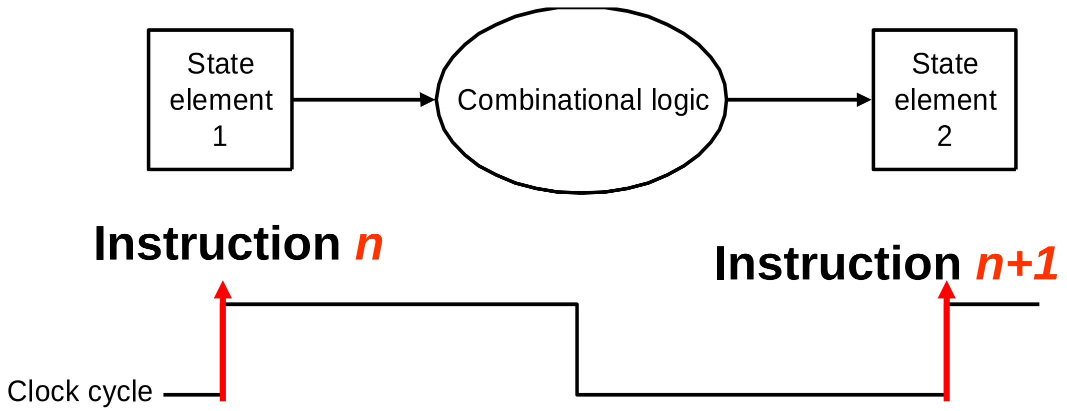
Processor Design……



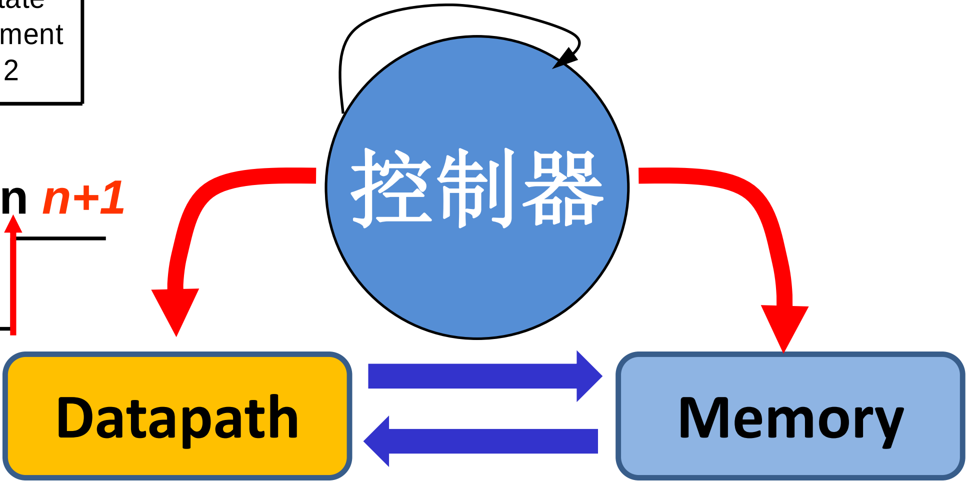
Controller Design

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



时钟下触发下一条指令

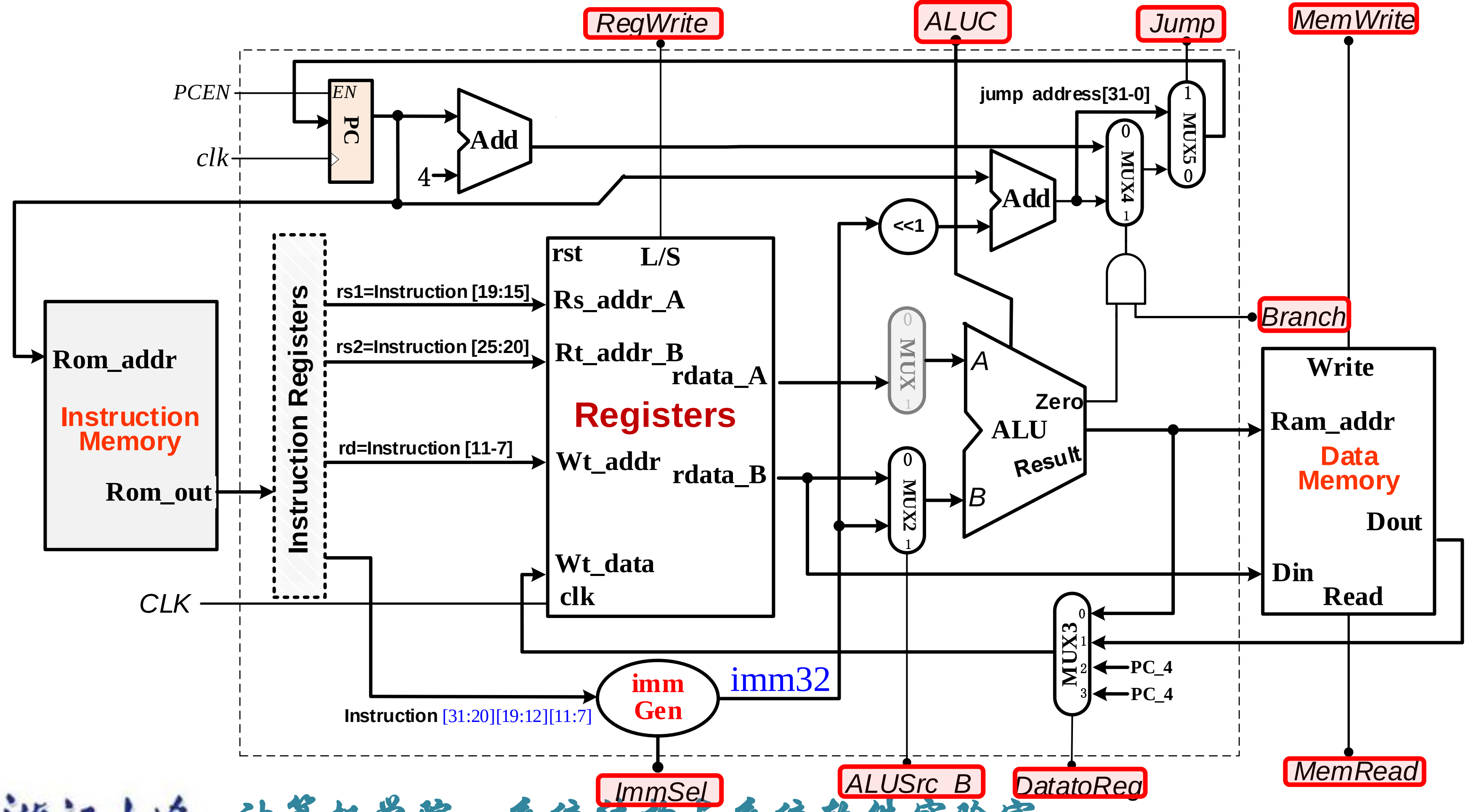


We are ignoring some details like setup and hold times

Building the Datapath & Controller

- **Use multiplexors to stitch them together**
  - There are 9(13) signals

Note：Page 257 F4.15



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

31 25	24 20	19 15	14 12	11 7	6 0
R-format instruction (add, sub, and, or, slt)					
Funct7	**rs2**	**rs1**	**funct3**	**rd**	**Op**
7 bits	5bits	5bits	3bits	5bits	7bits
I-format instruction (lw)					
Immediate[11:0]		**rs1**	**funct3**	**rd**	**Op**
S-format instruction (sw)					
Imm[11:5]	**rs2**	**rs1**	**funct3**	I[4:0]	**Op**
7 bits	5bits	5bits	3bits	5bits	7bits
sb-format instruction (beq)					
Imm[12,10:5]	**rs2**	**rs1**	**funct3**	I[4:1,11]	**Op**
7 bits	5bits	5bits	3bits	5bits	7bits
UJ-format instruction (jal)					
Immediate[20,10:1,11]		Imm[19:12]		**rd**	**Op**
12bits		8bits		5bits	7bits

Instruction Code: Input P103

- **Operations** comes from the OPcode of the instruction

Instruction Code

		31 Fun7 25	24 20	19 15	Fun3 14 12	11 7	6 OP 0	
add	R	0000000	rs2	rs1	000	rd	0110011	
sub	R	0100000	rs2	rs1	000	rd	0110011	
and	R	0000000	rs2	rs1	111	rd	0110011	
or	R	0000000	rs2	rs1	110	rd	0110011	
slt	R	0000000	rs2	rs1	010	rd	0110011	
lw	I	Imm[11:0]		rs1	010	rd	0000011	
sw	S	Imm[11:5]	rs2	rs1	010	Imm[4:0]	0100011	
beq	SB	ofst[12,10:5]	rs2	rs1	000	osft[4:0,11]	1100111	
jal	UJ	Target[20,10:1,11]		Target[19:12]		rd	1101111	

What should ALU do ?

Why is the code for subtract 110 and not 011?

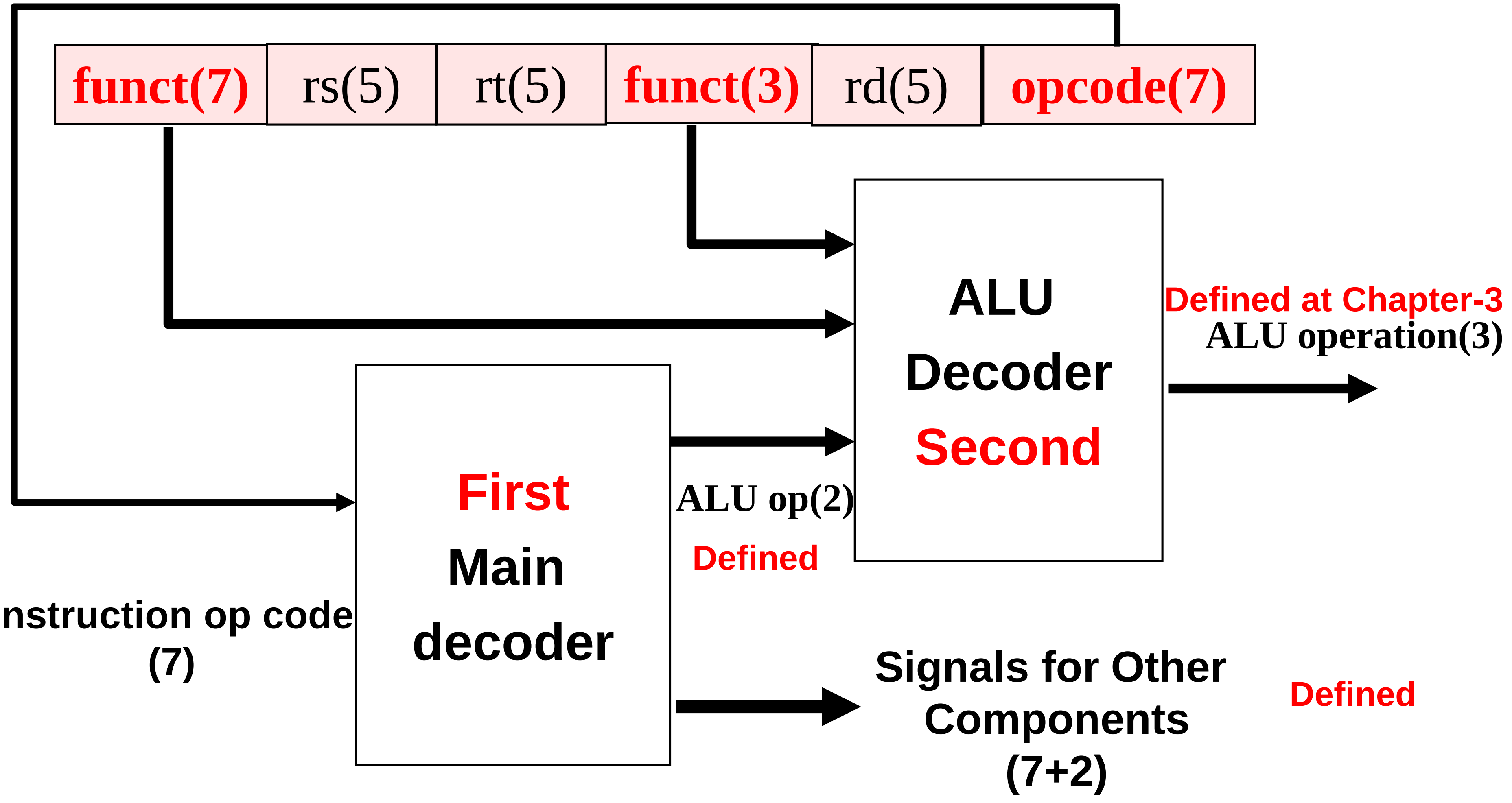
B negate	op	function
0	00	and
0	01	or
0	10	add
1	10	sub
1	11	slt

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

opcode	ALUOp	Operation	Funct7	Fun3	Desired ALU action	ALU Control input
lw/ld	**00**	load register	XXXXXXX	010	add	0 010
sw/sd	**00**	store register	XXXXXXX	010	add	0 010
beq	**01**	branch on equal	XXXXXXX	000	subtract	0 110
R-type	**10**	add	0000000	000	add	0 010
		subtract	0100000	000	subtract	0 110
		and	0000000	111	AND	0 000
		or	0000000	110	OR	0 001
		slt	0000000	010	subtract	0 111

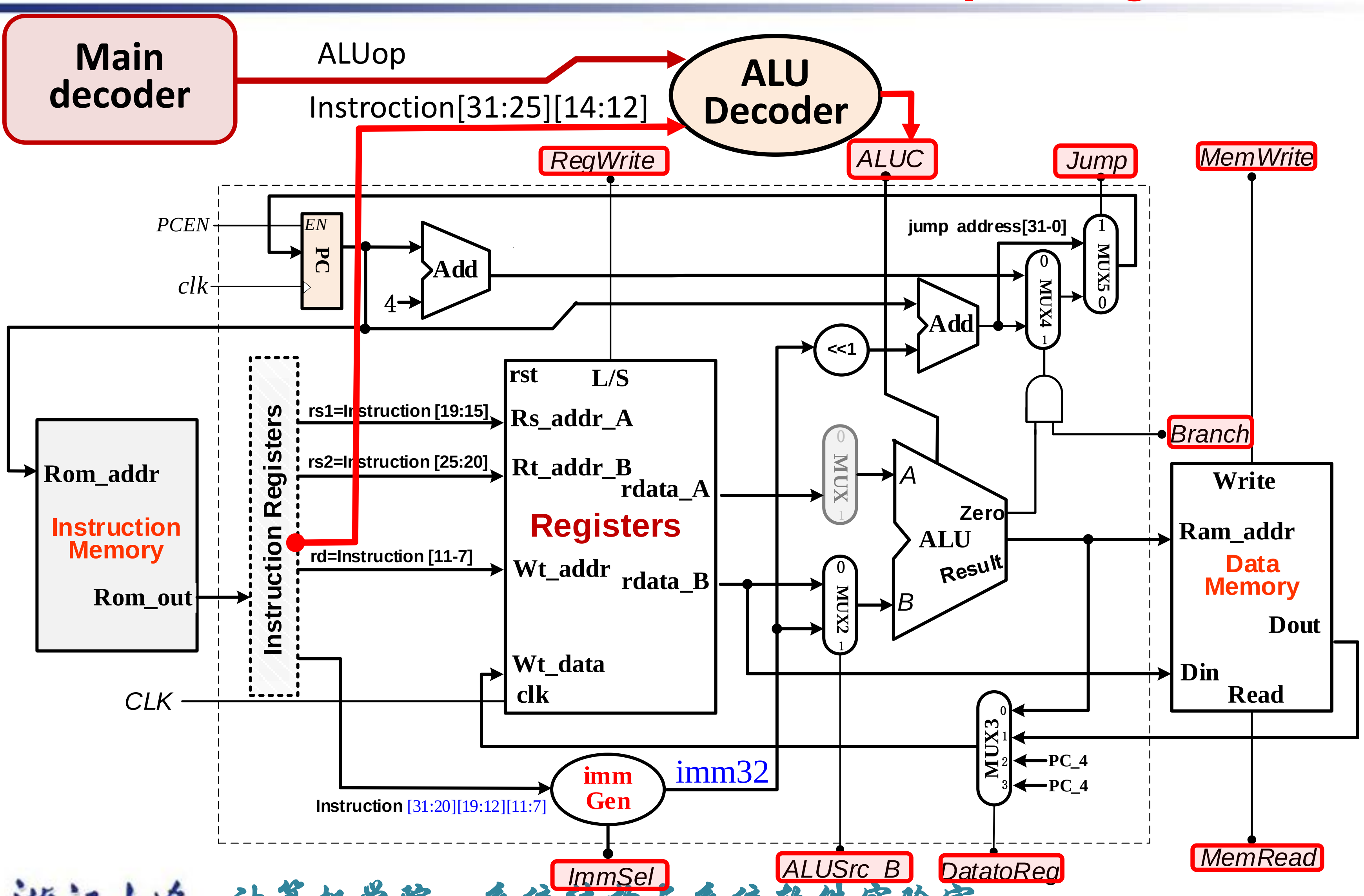
Scheme of Controller

- 2-level decoder



The ALU control where is & Other signals(8)

Output signals



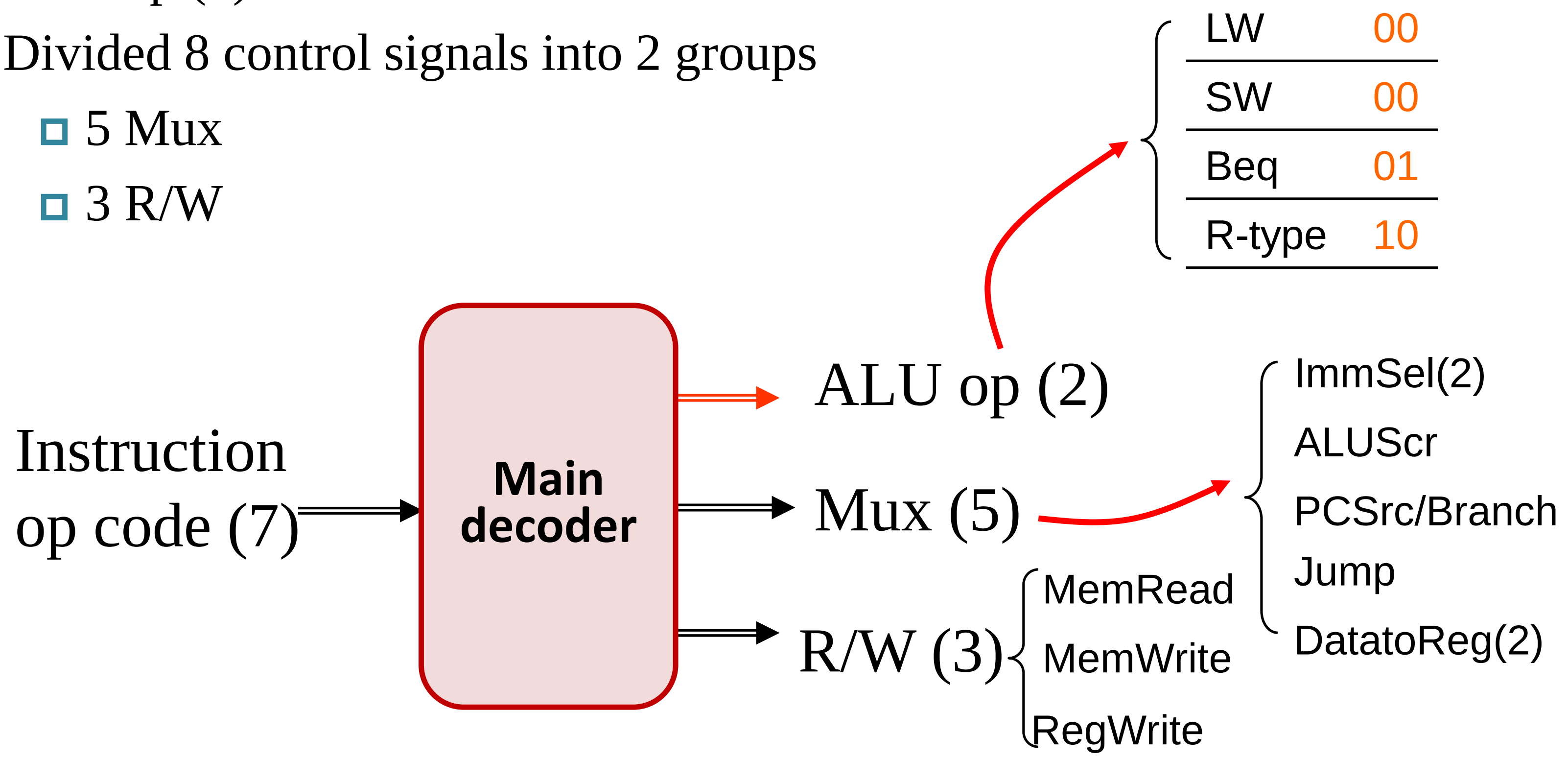
Signals for datapath

Defined 8+2 control (p. 256)

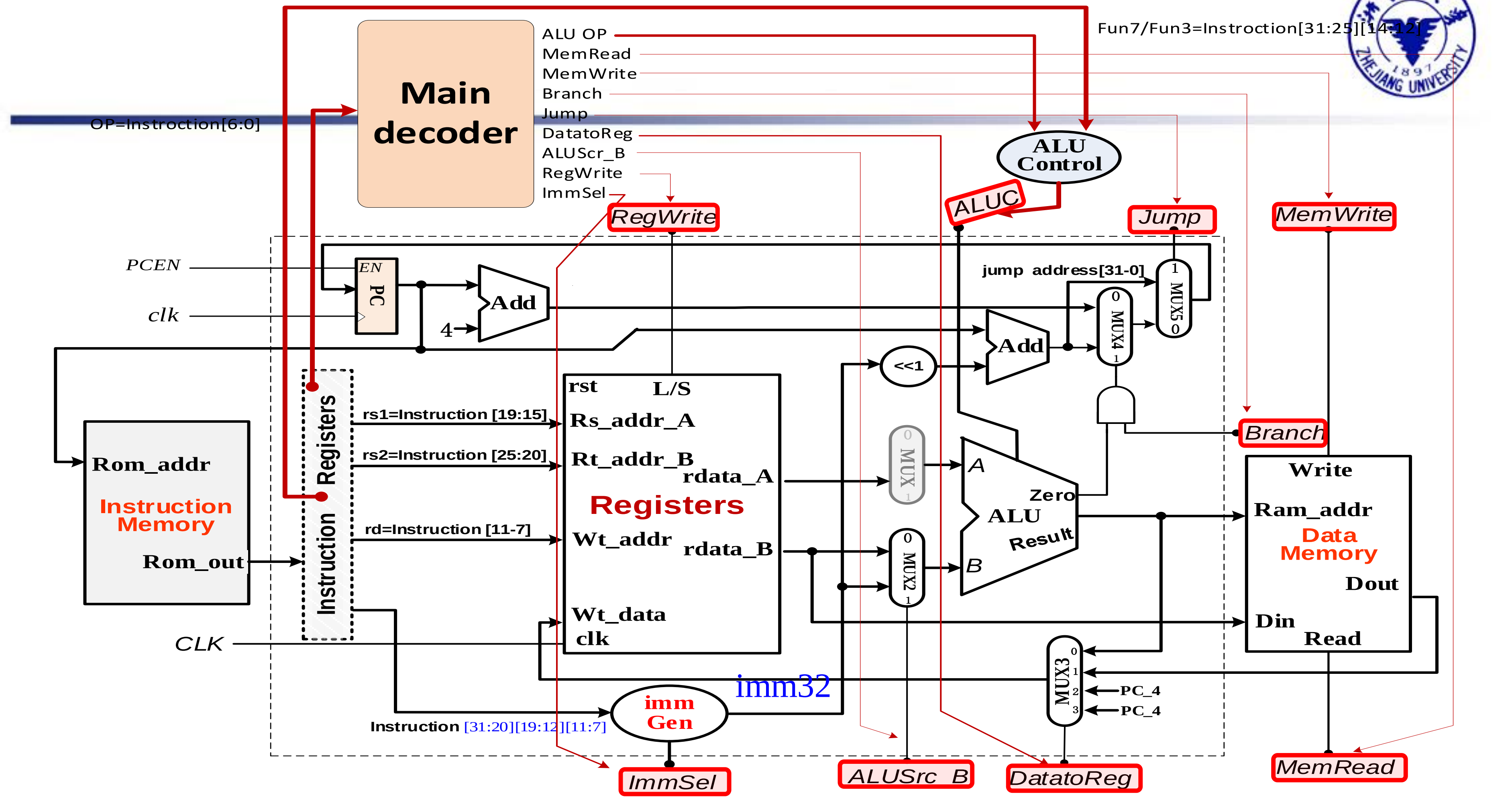
Signal name	Effect when deasserted(=0)	Effect when asserted(=1)
ImmSel	=00：imm32 =01：imm32=ims32	=10：imm32=imsb32 =11：imm32=imuj32
RegWrite	None	The register on the Write register input is written with the value on the Write data input
ALUScrB	The second ALU operand come from the second register file output (Read data 2)	The second ALU operand is the sign-extended lower 16 bits of the instruction..
PCSrc (Branch)	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.
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 DatatoReg	The value fed to register Write data input comes from the Alu	The value fed to the register Write data input comes from the data memory.
Jump	None	The PC is replaced by Jump(Jal) target
ALUop	Main decoder definition	
ALUC	Refer to Chapter 3 ALU Function Table	

Designing the Main Control Unit
First level

- **Main Control Unit function**
  - ALU op (2)
  - Divided 8 control signals into 2 groups
    - 5 Mux
    - 3 R/W



Truth Table for Main decoder



Instruction	ImmSel	ALUSrc_B	DatatoReg	RegWrite	MemRead	MemWrite	Branch	Jump	ALU$_{op}$
R-Type	xx	0	00	1	0	0	0	0	10
I-Type/lw	00	1	01	1	1	0	0	0	00
S-Type/sw	01	1	xx	0	0	1	0	0	00
Uj-Type/jal	xx	x	***10***	***1***	0	0	0	1	xx

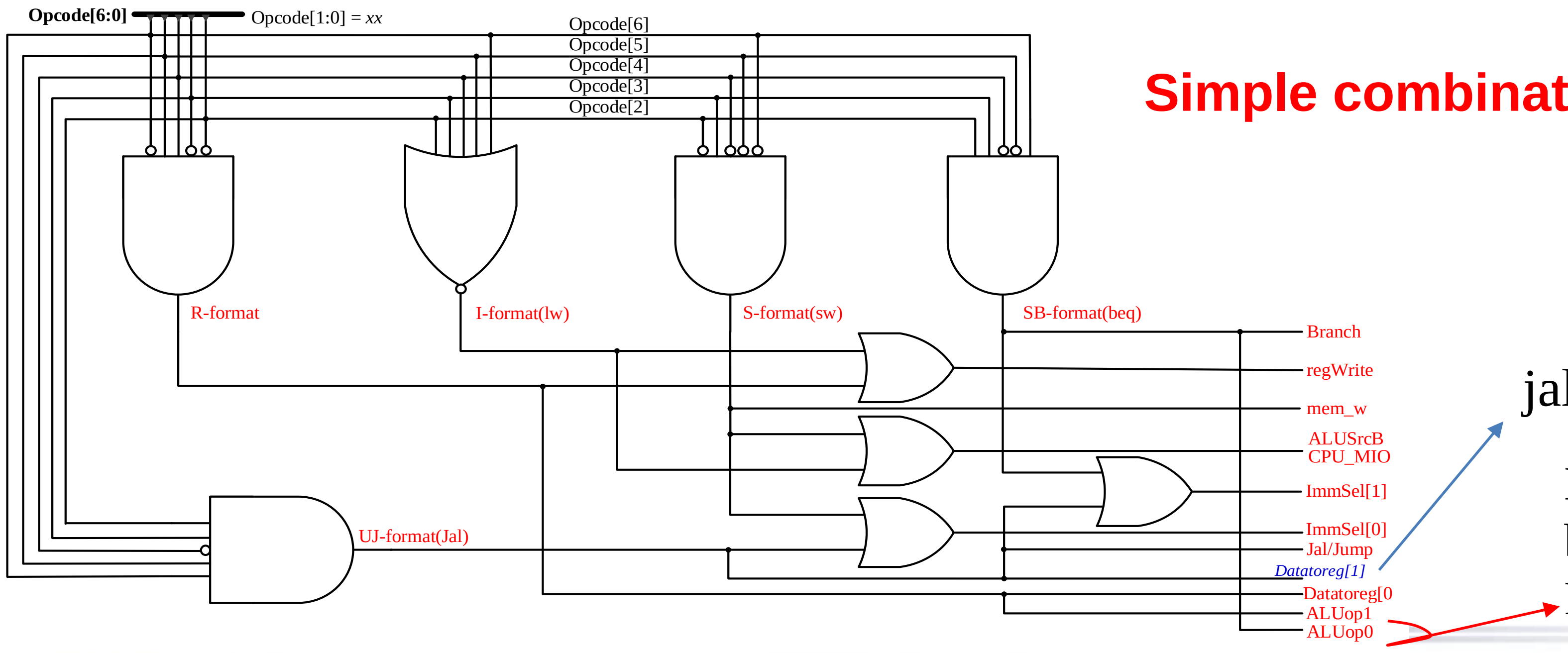
For 9 Instruction Controler Truth

Instruction	format	Imm Sel	ALU Src_B	Data toReg	Reg Write	Mem Read	Mem Write	Branch	Jump	ALU op1	ALU op0
add	**R**	xx	0	00	1	0	0	0	0	1	0
sub	**R**	xx	0	00	1	0	0	0	0	1	0
and	**R**	xx	0	00	1	0	0	0	0	1	0
or	**R**	xx	0	00	1	0	0	0	0	1	0
slt	**R**	xx	0	00	1	0	0	0	0	1	0
nor	**R**	1	0	0	1	0	0	0	0	1	0
lw	**I**	00	1	01	1	1	0	0	0	0	0
sw	**I**	01	1	***xx***	0	0	1	0	0	0	0
beq	**I**	***10***	0	***xx***	0	0	0	1	0	0	1
jal	**J**	***11***	***x***	10	**1/0**	0	0	***x***	1	***x***	***x***

Truth tables & Circuitry of main Controller

Input		Output								
Instruction	OPCode	Imm Sel	ALU SrcB	Datato reg	reg Write	Mem Read	Mem Write	Branch	Jump	ALU_{Op}
R-format	0110011	xx	0	00	1	0	0	0	0	10
I-Type/lw	0000011	00	1	01	1	1	0	0	0	00
S-Type/sw	0100011	01	1	xx	0	0	1	0	0	00
SB-Type/beq	1100111	10	0	xx	0	0	0	1	0	01
UJ/Jump	1101111	11	x	10	1	0	0	0	1	xx

Simple combinational logic



jal

L/S	00
beq	01
R-type	10

Main Controller Code

```
always @* begin
    ALUSrc_A  =0;
    ALUSrc_B  =0;
    DatatoReg =0;
    RegWrite  =0;
    Branch    =0;
    Jump      =0;
    WR        =0;
    CPU_MIO   =0;
    ALUop     =2'b10;
    Error     =6'b00000;

  case(OPcode)
    5'b01100: begin ALUop=2'b10;RegWrite=1;          ALUSrc_B=0;Branch=0;Jump=0;DatatoReg=2'b00;  end //ALU(R)
    5'b00000: begin ALUop=2'b00;RegWrite=1;ImmSel=00;ALUSrc_B=1;Branch=0;Jump=0;DatatoReg=2'b01;
                                                                  WR=0;CPU_MIO = 1; end //load
    5'b01000: begin ALUop=2'b00;RegWrite=0;ImmSel=01;ALUSrc_B=1;Branch=0;Jump=0;WR=1;CPU_MIO = 1; end //store
    5'b11000: begin ALUop=2'b01;RegWrite=0;ImmSel=10;ALUSrc_B=0;Branch=1;Jump=0;                  end //beq
    5'b11011: begin             RegWrite=1;ImmSel=11;                        Jump=1;DatatoReg=2'b10;  end //jump
//增加立即数ALU运算指令，需要增加ALUop编码，但不用修改数据通路
    5'b00100: begin ALUop=2'b11;RegWrite=1;ImmSel=00;ALUSrc_B=1;Branch=0;Jump=0;DatatoReg=2'b00;  end //ALU(I)
    default:        Error   =6'b101010;                                          //illegal instruction
  endcase
end
```

Designing the ALU decoder

Second level

- Must describe hardware to Compute 3-bit ALU conrol input

opcode	ALUop	Operation	Funct7	Funct3	Desired ALU action	ALU control Input
LW/LD	00	load register	XXXXXXX	xxx	Load word	0010
SW/SD	00	store register	XXXXXXX	xxx	Store word	0010
Beq	01	branch on equal	XXXXXXX	xxx	branch equal	0110
R-type	10	add	0000000	000	add	0010
		subtract	0100000	000	subtract	0110
		AND	0000000	111	AND	0000
		OR	0000000	110	OR	0001
		Set on less than	0000000	010	slt	0111

Truth Table for ALU decoder

- Describe it using a truth table (can turn into gates):

opcode	ALUop		Funct7			Funct3			ALUC
	ALU_{op1}	ALU_{op0}	$F7_6$	$F7_5$	$F7_{4\text{-}0}$	$F3_2$	$F3_1$	$F3_0$	*210*
L/S	0	0	x	x	xxxxx	x	x	x	010
beq	x	1	x	x	xxxxx	x	x	x	110
R-type	1	x	x	0	xxxxx	0	0	0	010
			x	1	xxxxx	0	0	0	110
			x	0	xxxxx	1	1	1	000
			x	0	xxxxx	1	1	0	001
			x	0	xxxxx	0	1	0	111

x： don’t care

$$\mathrm{ALUC_2} = ALUop_0 + ALU_{op1}(\mathrm{F7_5}\,\overline{\mathrm{F3}}_2\overline{\mathrm{F3}}_1\overline{\mathrm{F3}}_0 + \overline{\mathrm{F7}}_5\,\overline{\mathrm{F3}}_2\mathrm{F3}_1\overline{\mathrm{F3}}_0)$$

$$\mathrm{ALUC_1} = \overline{ALU_{op1}(\overline{\mathrm{F7}}_5\,\mathrm{F32}F3_1F3_0 + \overline{\mathrm{F7}}_5\,F\mathrm{32F3_1}\overline{\mathrm{F3}}_0)}$$

$$\mathrm{ALUC_0} = ALU_{op1}(\overline{\mathrm{F7}}_5\,F\mathrm{32F3_1}\overline{\mathrm{F3}}_0 + \overline{\mathrm{F7}}_5\,\overline{\mathrm{F3}}_2\mathrm{F3}_1\overline{\mathrm{F3}}_0)$$

The ALU control signals – Combination circuit

$$\mathbf{ALUC_0} = ALU_{op1}\, F3_1 \overline{F3_0}$$

$$\mathbf{ALUC_1} = \overline{ALU_{op1}\ F32} = \overline{ALU_{op1}} + \overline{F32}$$

$$\mathbf{ALUC_2} = ALUop_0 + ALU_{op1}F7_5 + ALU_{op1}\overline{F3_2}F3_1\overline{F3_0}$$

Fun = {Inst[30], Inst[14:12]}

$F7_5F3_2$ \ $F3_1F3_0$				
	0	x	x	1
	x	x	0	0
	x	x	x	x
	1	x	x	x

$F7_5$

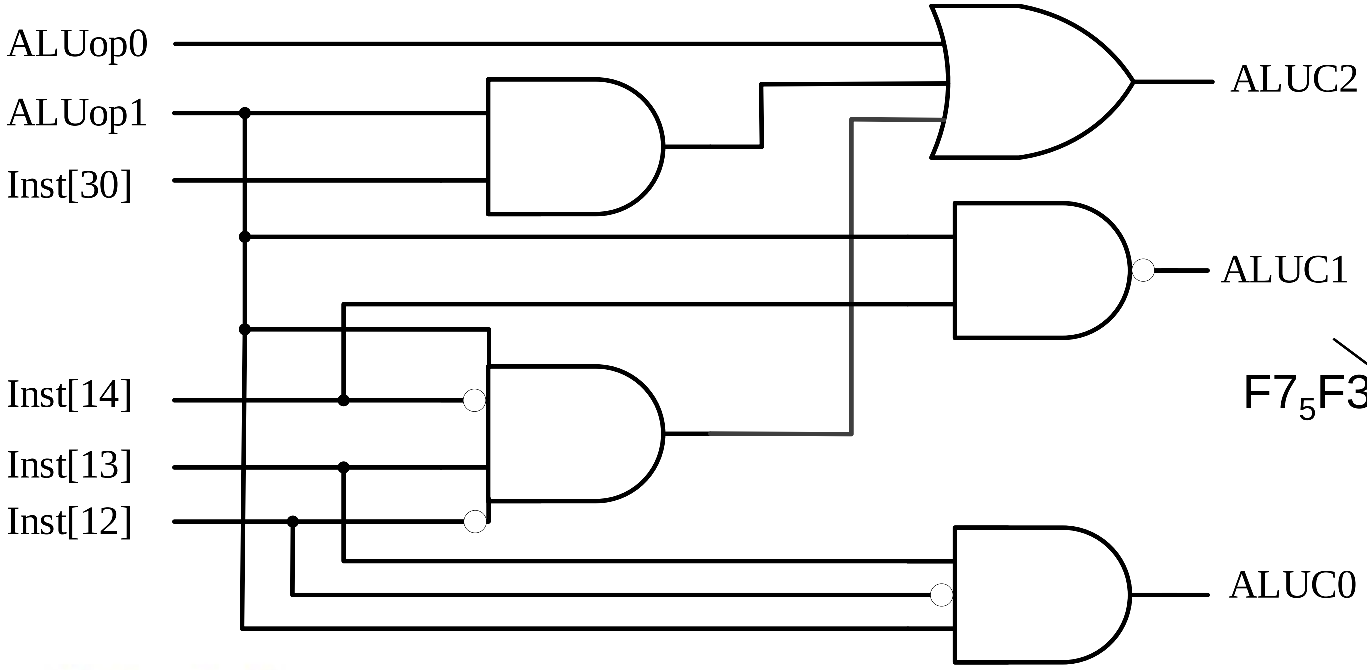
$\overline{F3_2}F3_1\overline{F3_0}$

$F7_5F3_2$ \ $F3_1F3_0$				
	1	x	x	1
	x	x	0	0
	x	x	x	x
	1	x	x	x

$F3_2$

$F7_5F3_2$ \ $F3_1F3_0$				
	0	x	x	1
	x	x	0	1
	x	x	x	x
	0	x	x	x

$F3_1\overline{F3_0}$



ALU Controller Code

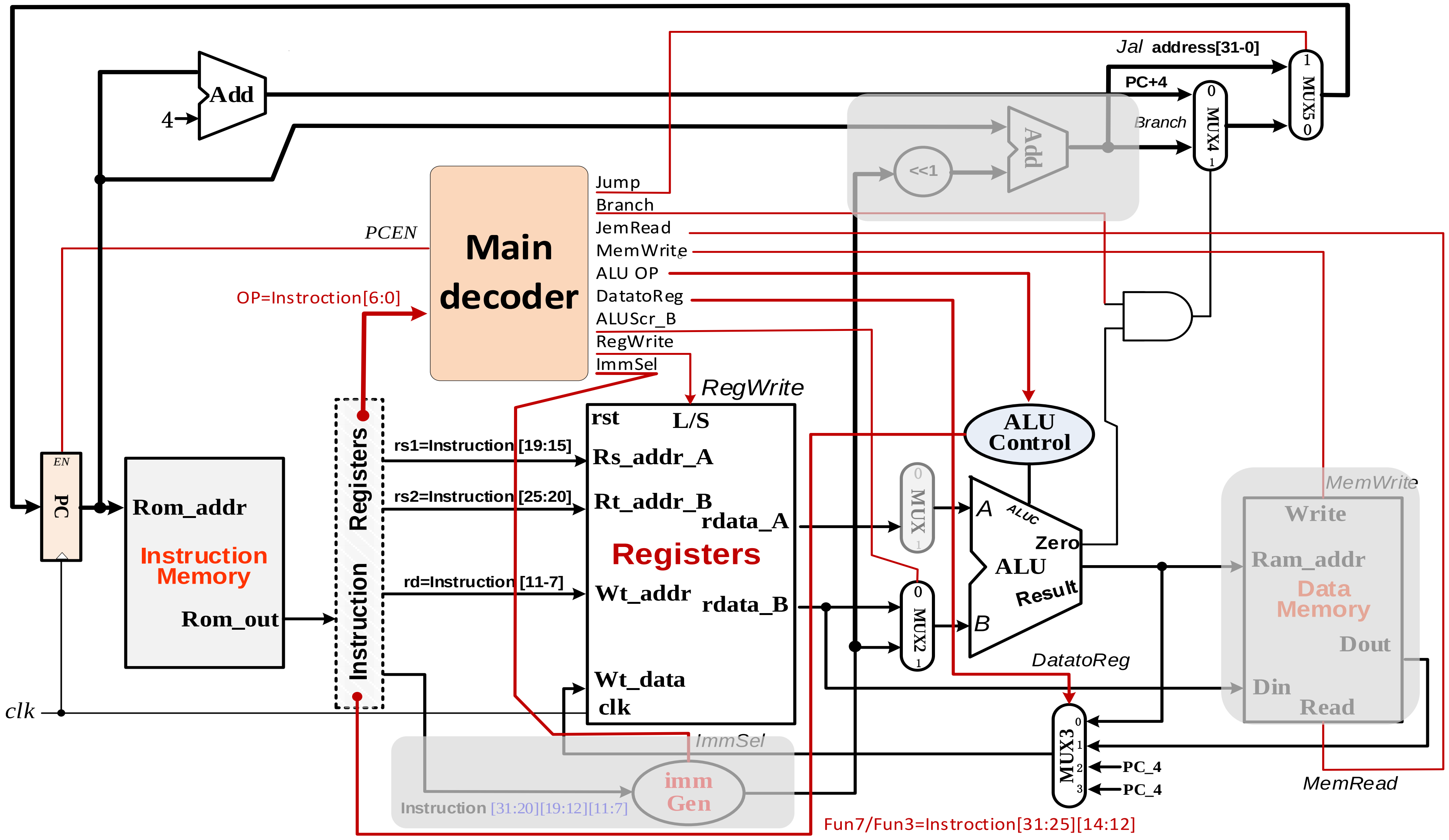
```
assign Fun = {Fun3,Fun7};
always @* begin
    case(ALUop)
        2'b00: ALUC=3'b010;                                 //load/store
        2'b01: ALUC =3'b110;                                //sub: beq
        2'b10:
            case(Fun)
                4'b0000: ALUC =3'b010;                      //add
                4'b0001: ALUC =3'b110;                      //sub
                4'b1110: ALUC =3'b000;                      //and
                4'b1100: ALUC =3'b001;                      //or
                4'b0100: ALUC =3'b111;                      //slt
                4'b1010: ALUC =3'b101;                      //srl
                4'b1000: ALUC =3'b011;                      //xor
                default: ALUC =3'bx;
            endcase
        2'b11:
                case(Fun3)
                    3'b000: ALUC =3'b010;                   //addi
                    3'b111: ALUC =3'b000;                   //andi
                    3'b110: ALUC =3'b001;                   //ori
                    3'b010: ALUC =3'b111;                   //slti
                    3'b101: ALUC =3'b101;                   //srli
                    3'b100: ALUC =3'b011;                   //xori
                    default:    ALUC =3'bx;
                endcase
        default:    ALUC =3'bx;
    endcase
end
```

立即数扩展

R-Type Instruction

add x9, x20, x21

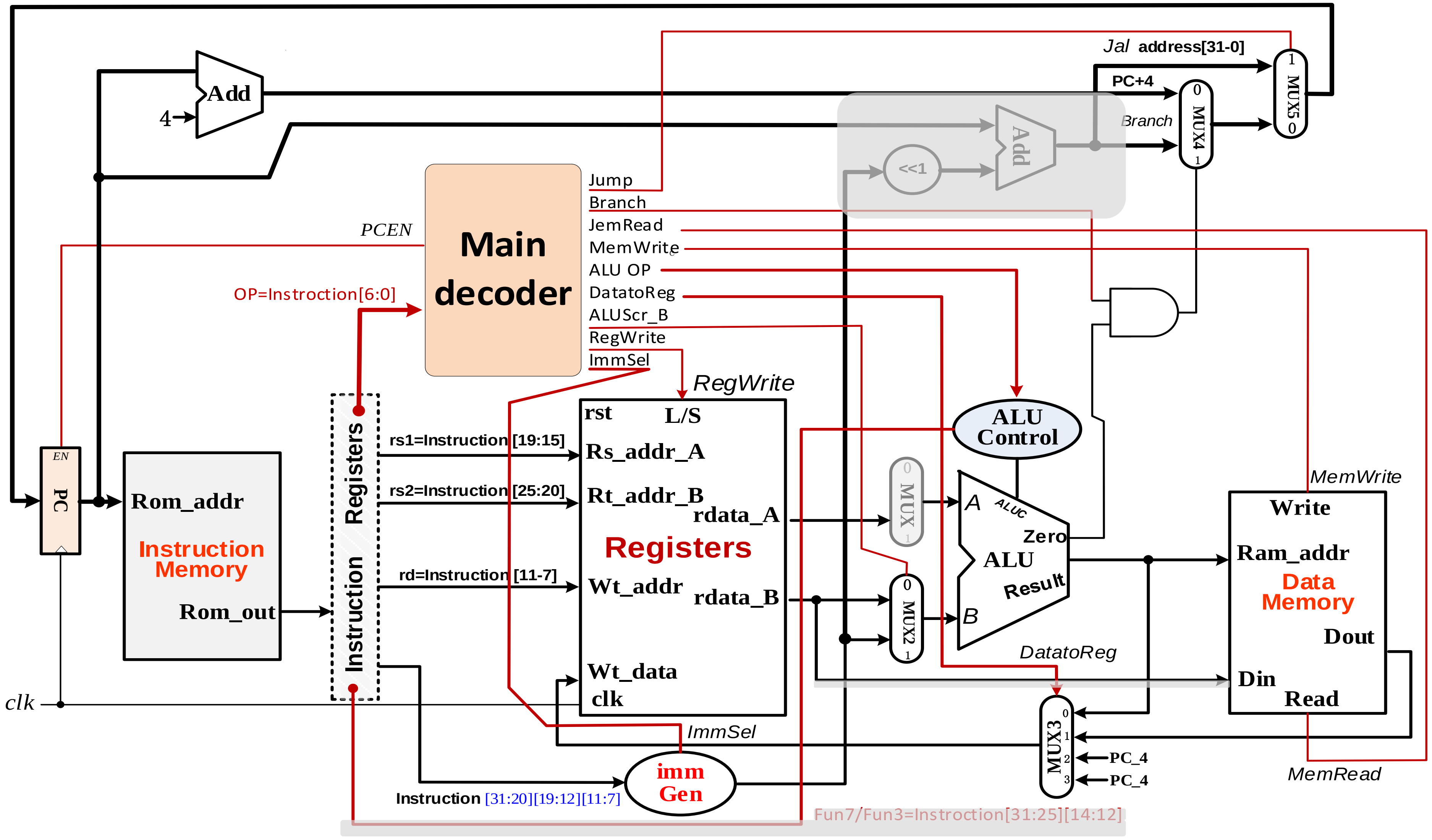
funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits



Load Instruction

lw x1, 200(x2)

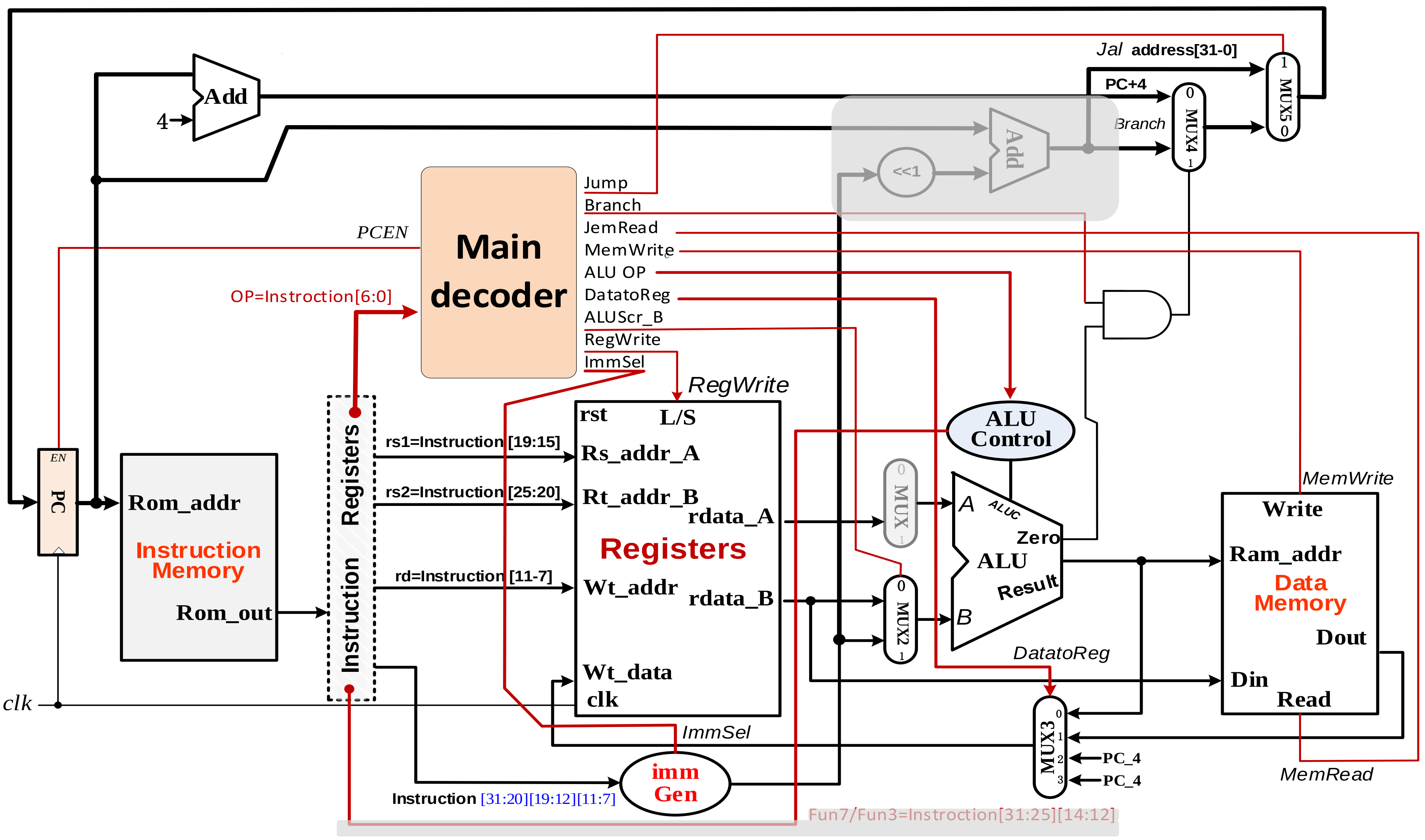
immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits



Store Instruction

sw x1, 200(x2)

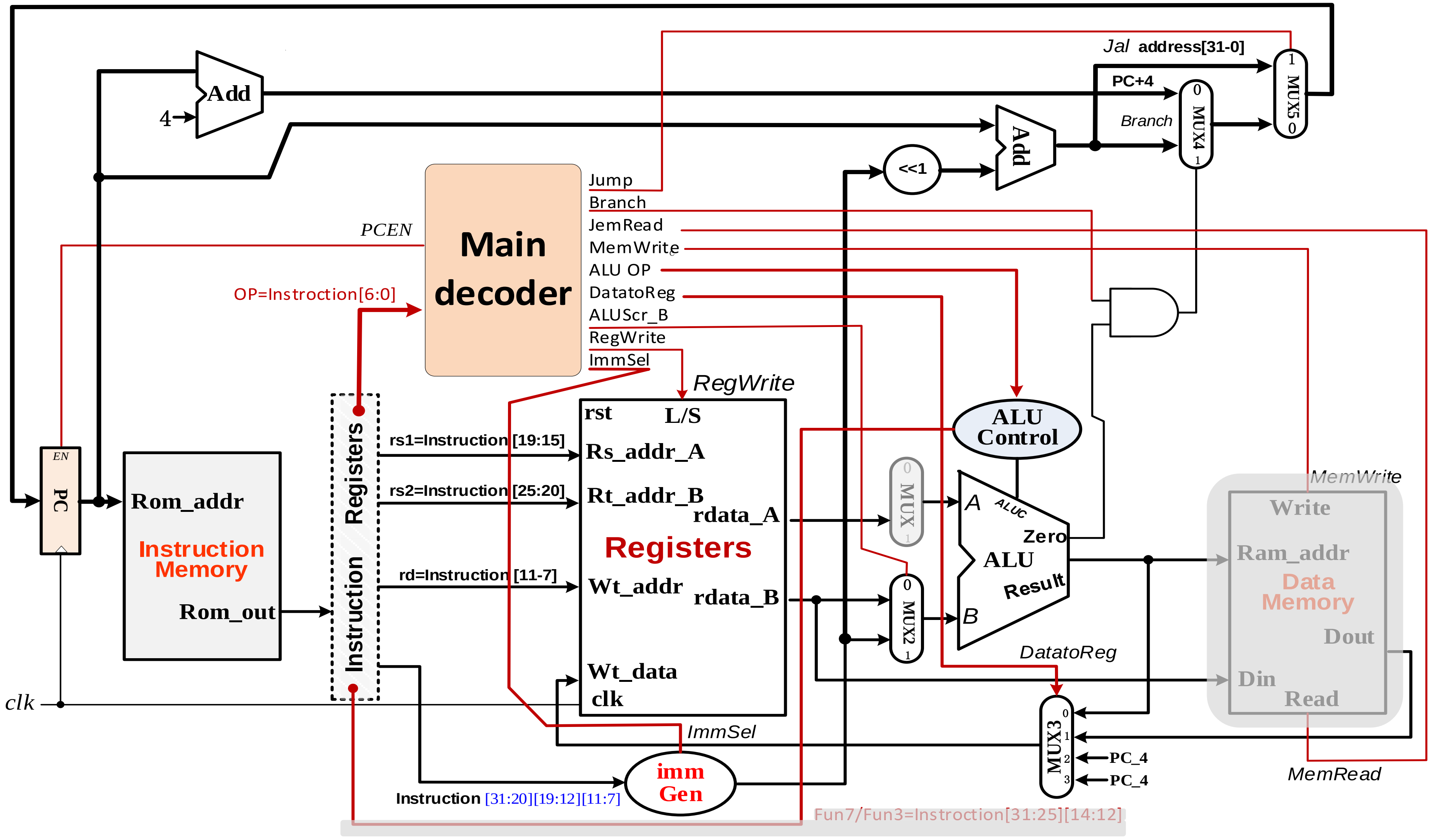
immediate[11:5]	rs2	rs1	funct3	immediate[4:0]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits



beq Instruction

beq x1, x2， 200

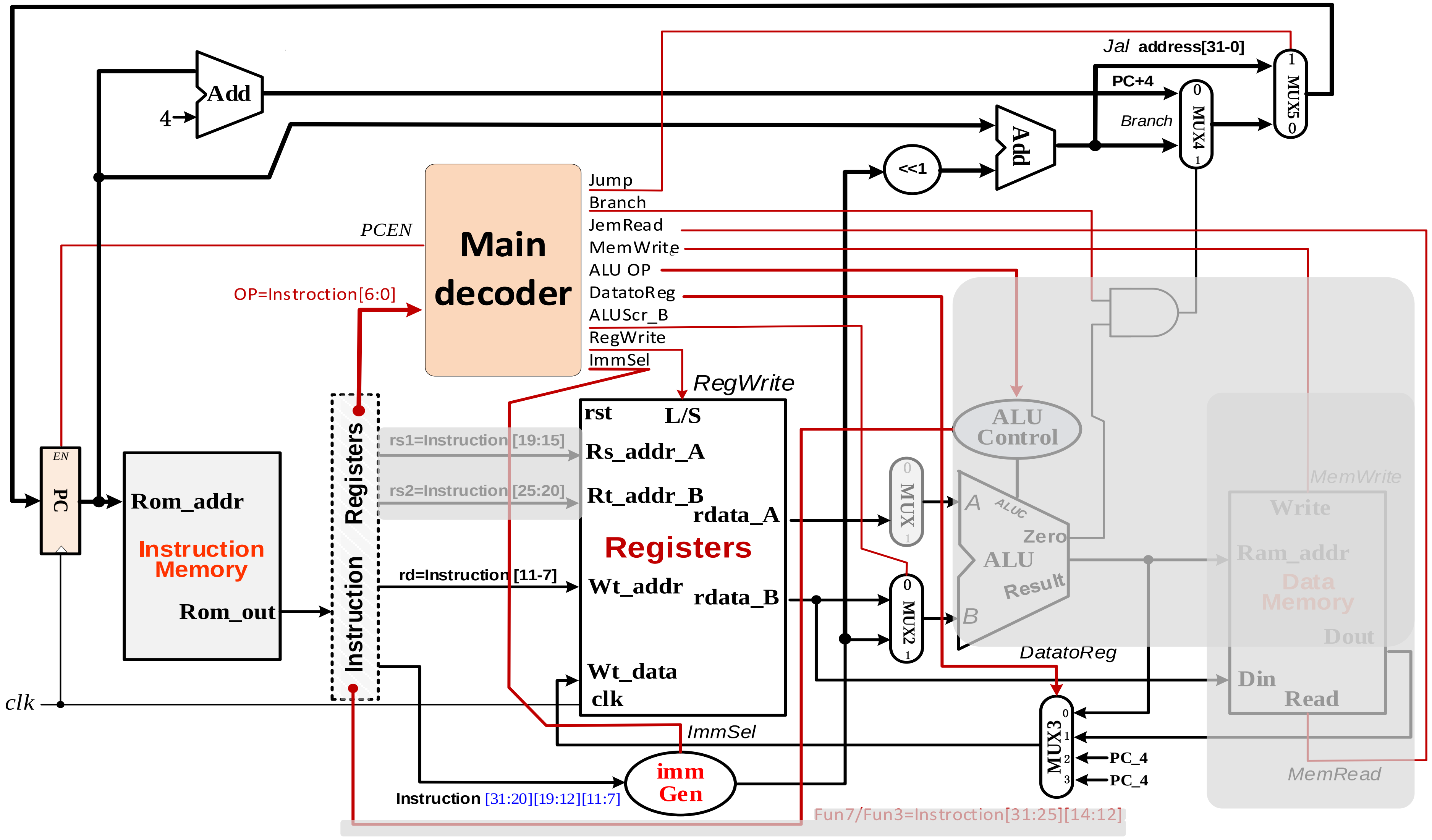
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode



Jal Instruction

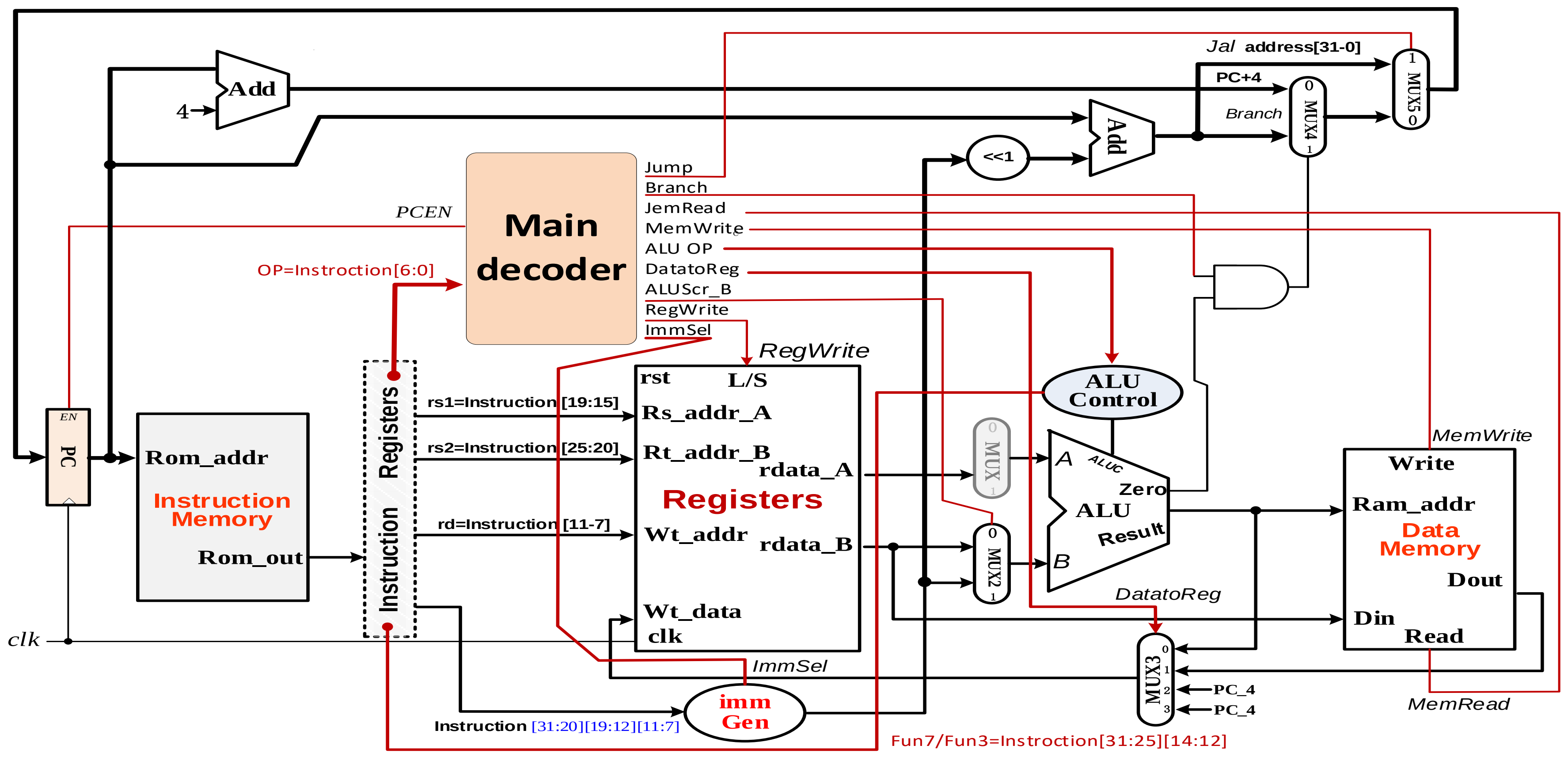
jal x1, procedure

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



Single Cycle Implementation performance for lw

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



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

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/lw	200ps	100 ps	200ps	200ps	100 ps	800ps
Sd/sw	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.

- **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** ⏎

A Multicycle Implementation

The CPU Performance Equation

$$\text{CPU time} = \text{CPU clock cycles for a program} \times \text{Clock cycle time}$$

$$\text{CPU time} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate}}$$

$$\mathbf{CPU}_{\mathbf{time}} \mathbf{= I\ CPI\ \tau}$$

$$\text{CPI} = \frac{\text{CPU clock cycles for a program}}{\text{Instruction count}}$$

$$\text{CPU time} = \text{Instruction count} \times \text{Clock cycle time} \times \text{Cycles per instruction}$$

$$\text{CPU time} = \frac{\text{Instruction count} \times \text{Clock cycle time}}{\text{Clock rate}}$$

$$\frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}} = \frac{\text{Seconds}}{\text{Program}} = \text{CPU time}$$

- CPU performance is dependent upon three characteristics:
  - clock cycle (or rate)
  - clock cycles per instruction
  - and instruction count.
- It is difficult to change one parameter in complete isolation from others because the basic technologies involved in changing each characteristic are interdependent:
  - Clock cycle time—Hardware technology and organization
  - Clock Per Instruction—Hardware technology and organization
  - Instruction count—Instruction set architecture and compiler technology

MIPS *(million instruction per second)*

$$\text{MIPS} = \frac{\text{Instruction count}}{\text{Execution time} \times 10^6} = \frac{\text{Clock rate}}{\text{CPI} \times 10^6}$$

$$\text{Execution time} = \frac{\text{Instruction count}}{\text{MIPS} \times 10^6}$$

× ***The bigger the MIPS, the faster the machine.***

- **Three problems with MIPS:**
  - MIPS is dependent on the instruction set, making it difficult to compare MIPS of computers with different instruction sets.
- MIPS varies between programs on the same computer.
- Most importantly, MIPS can vary inversely to performance!

⦿END