

计算机组成原理



群名称:计算机组成原理-文杰老师-4...
群 号:139369553

文杰

计算机科学与技术学院

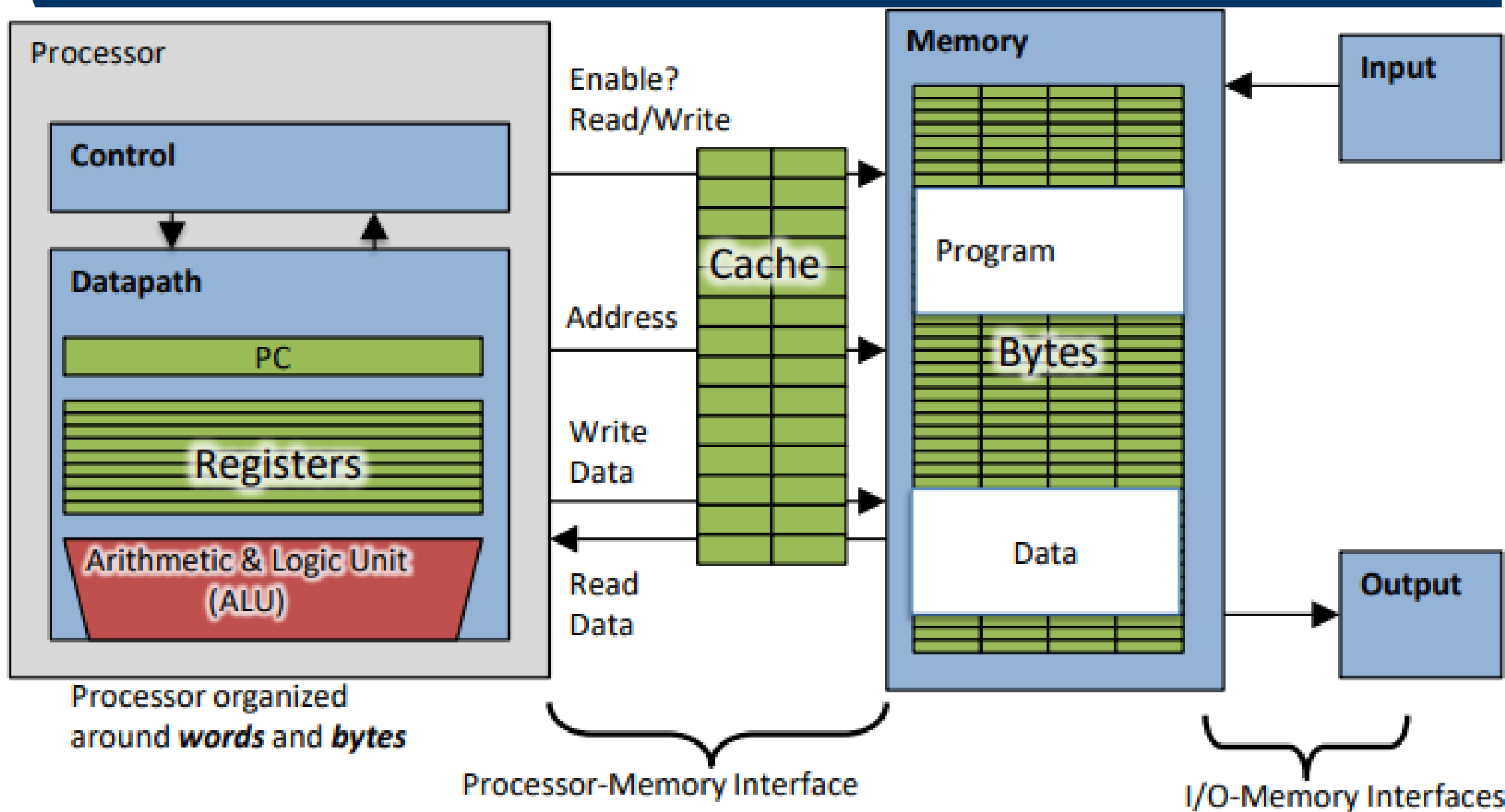
wenjie@hit.edu.cn

个人主页: <http://faculty.hitsz.edu.cn/wenjie>

第四章 处理器设计

- 处理器设计的需求分析
- RISC-V数据通路的组件选择
- RISC-V部分指令的数据通路设计
- RISC-V控制器

单核计算机系统



CPU功能层面的定义

- 构建一个能够通过输入的机器码，执行相应操作、并保持相应状态的**数字电路**。

RISC-V机器码 `0x00600293`
(例如: `addi t0, x0, 6`)

数字电路
(逻辑门、寄存器)

执行完这条指令后,
寄存器t0的值为6

CPU的组成部分

- **数据通路(datapath)**是处理器中执行操作的硬件
 - 执行控制器的操作（例如，控制器告诉数据通路，执行addi指令，则数据通路就会将操作数传递给加法器）

≈≈处理器的四肢

- **控制器**是对数据通路要做什么操作进行调度的硬件结构
 - 告诉数据通路：需要执行什么操作？需要读内存吗？读哪个寄存器？需要写寄存器吗？写哪个寄存器？...

≈≈处理器的大脑

CPU的组成部分

addi t0,x0,6

0x00600293

数据通路

从控制器发送来的信息：

1. 操作数为0号寄存器的值和立即数“6”
2. 执行的运算是加法

故：选择0号寄存器的值和立即数作为操作数，并选择加法器的结果作为最终结果

3. 目的寄存器是t0，所以加法器的结果送往t0

控制器

命令： 要让数据通路执行什么操作？

- 加减乘除/逻辑/比较？
- 源操作数应该如何选择？
- 哪个寄存器？
- 什么立即数？
- 指令执行完后，PC如何改变？是否是分支？

处理器设计步骤

- ① 分析指令系统，得出对数据通路的需求
- ② 为数据通路选择合适的组件
- ③ 根据指令需求连接组件建立数据通路
- ④ 分析每条指令的实现，以确定控制信号
- ⑤ 集成控制信号，形成完整的控制逻辑

RISC-V指令格式回顾及需求

- 需要多个可以进行**读写**的32位寄存器，可以**同时读**2个。

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

RISC-V 部分指令回顾及需求

- R型指令: `op dst, src1, src2`
 - 需求: 常用算术逻辑运算, 可读写的寄存器, 可同时读2个。
- I型指令: `opi dst, src, imm` # $R[dst] = R[src] + s\text{-}ext(imm)$
 - `load dst, offset(bAddrReg)` # $R[dst] = mem[R[bAddrReg] + offset]$
 - `jalr rd, offset(rs1)` # $R[rd] = PC + 4, PC = R[rs1] + offset$
 - 新增需求: 立即数扩展、PC寄存器、存储器可读、加法器...
- S型指令: `store src, offset(bAddrReg)` # $mem[R[bAddrReg] + offset] = R[src]$
 - 新增需求: 存储器可写、立即数生成器 (包括扩展等)

RISC-V 部分指令回顾——续

- B型指令: bxx src1,src2, **label**
 - 新增需求: **多路选择器**、**功能更多**的立即数（扩展）生成器
- J型指令: jal rd, offset
- U型指令: 将**长立即数**加到PC并写入目的寄存器
 - 新增需求: **功能更多**的立即数生成器

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

J型	<u>imm[20]</u>	<u>imm[10:1]</u>	<u>imm[11]</u>	<u>imm[19:12]</u>	<u>rd</u>	opcode
----	----------------	------------------	----------------	-------------------	-----------	--------

U型	imm[31:12]	rd	opcode
----	------------	----	--------

U-immediate[31:12]

dest

LUI
AUIPC

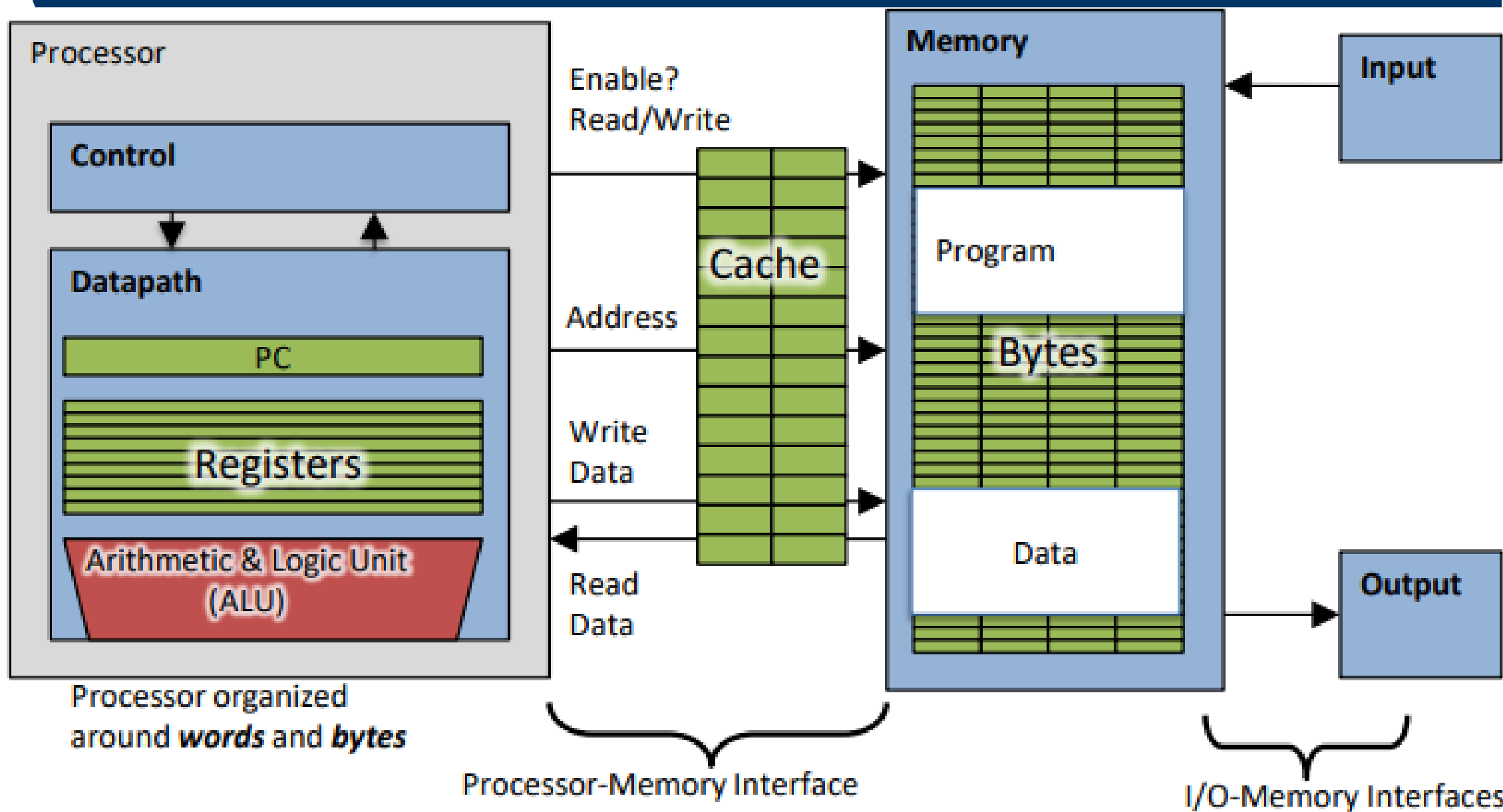
RISC-V指令系统的需求

- 算术逻辑单元（ALU）
 - 运算类型：加、减、或、比较等各种运算
- 立即数扩展（生成）器
 - 零扩展、符号扩展等
- 程序计数器（PC）
- 存储组件：寄存器堆(Register File)，存储器(Memory)

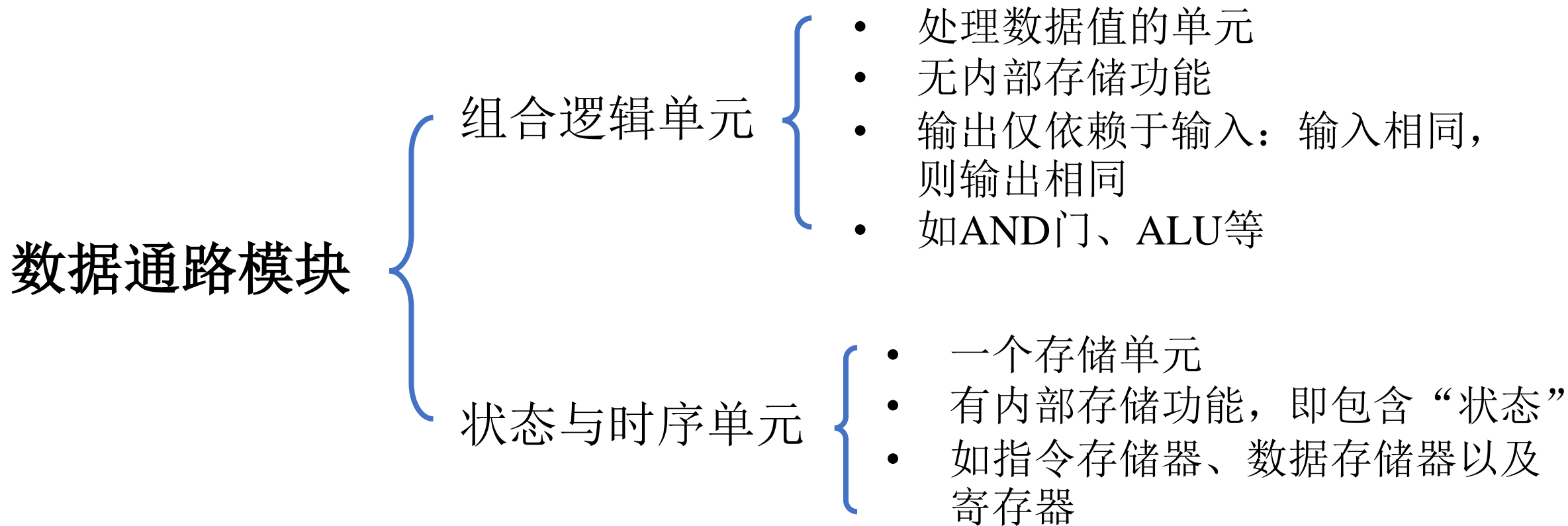
第四章 处理器设计

- 处理器设计的需求分析
- RISC-V数据通路的组件选择
- RISC-V部分指令的数据通路设计
- RISC-V控制器

单核计算机系统

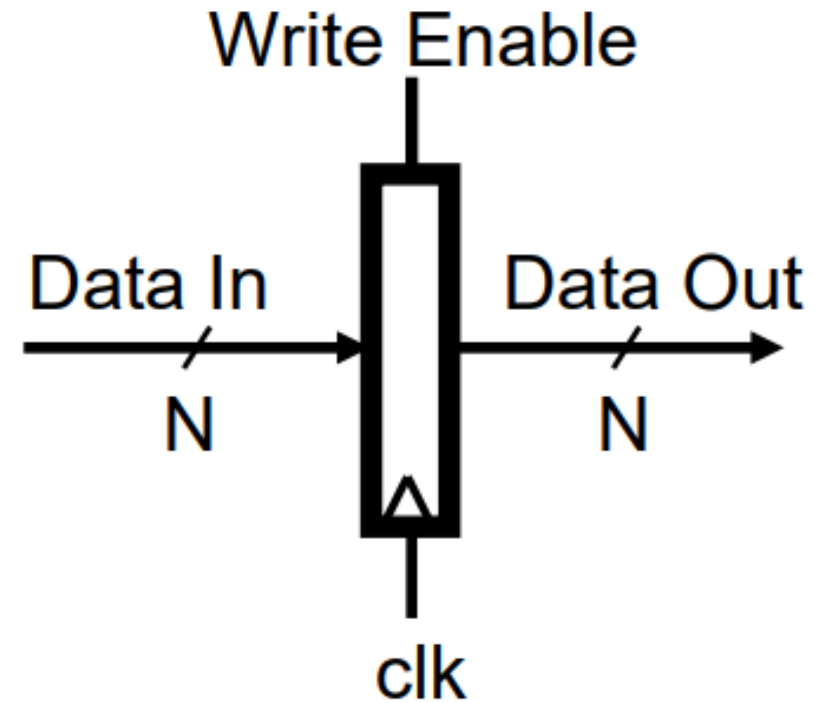


数据通路模块



数据通路模块——状态与时序单元

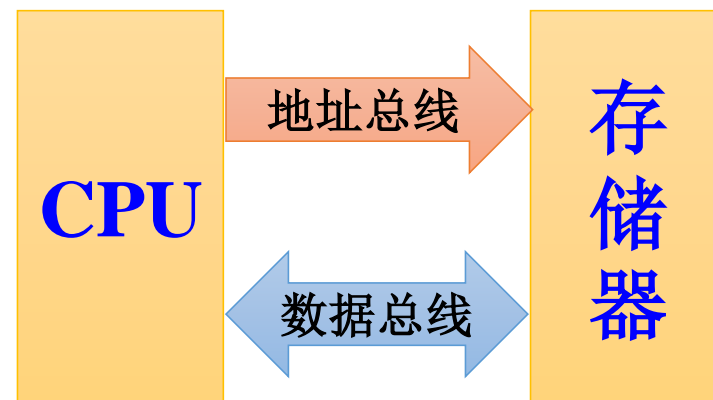
- 寄存器堆和存储器



冯诺依曼架构vs 哈佛架构

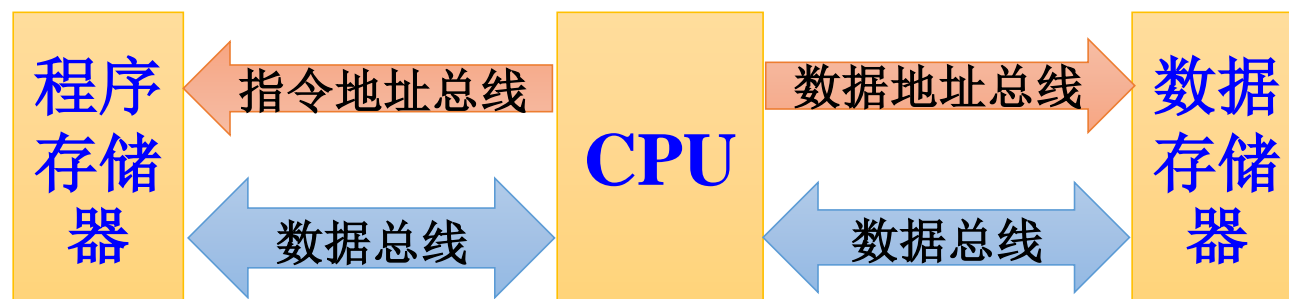
- 冯诺依曼架构（以X86为代表）

- 也叫普林斯顿架构，程序空间和数据空间是一体的，数据和程序采用同一数据总线和地址总线。
- 指令和数据的宽度相同。
- 指令和数据不能同时进行操作，只能顺序执行。



- 哈佛架构（以DSP和ARM为代表）

- 存储器分为数据存储器 and 程序存储器，总线分为程序存储器的数据总线和地址总线以及数据存储器的数据总线和地址总线。
- 可同时对数据和程序进行操作，具有较高的执行效率。
- 指令和数据可以有不同的宽度。



RISC-V 主要状态单元——存储器

- 将指令和数据保存在一个64位的**字节寻址**的存储空间中
- 指令存储器Imem和数据存储器Dmem**分开**（RISC CPU采用的是**哈佛**架构）
- 从指令存储器**读**（取）指令，在数据存储器中**读写**数据

数据存储单元 (DM:Data Memory)

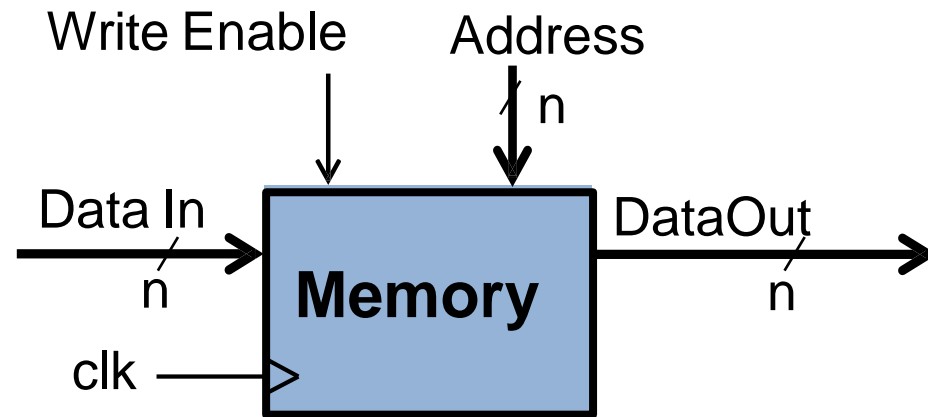
- 组合逻辑 vs 时序逻辑

- 数据接口信号

- Data In: 数据输入
- Data Out: 数据输出

- 读写控制

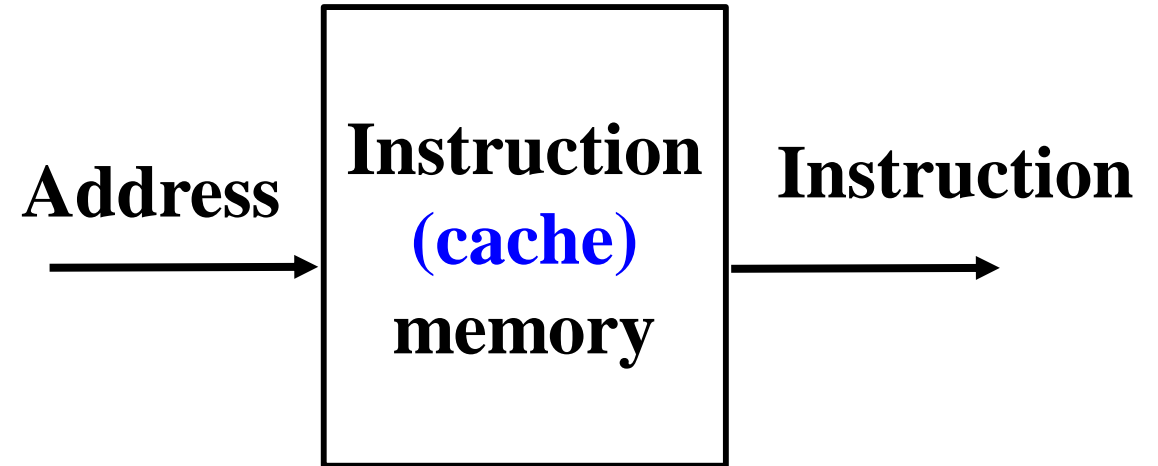
- 读: 通过Address接口, 指定一个存储单元, 将其内容送到数据输出端
- 写: Write Enable (写使能)。在时钟信号 (clk) 的上升沿, 如果写使能信号有效, 将数据输入信号的内容存入地址信号指定存储单元



注: 存储器的读操作不受时钟控制

指令存储器（IM:Instruction Memory）

- 输入总线： Instruction Address
- 输出总线： Instruction
- 数据通路中**不会写**指令存储器



RISC-V主要状态单元——寄存器堆

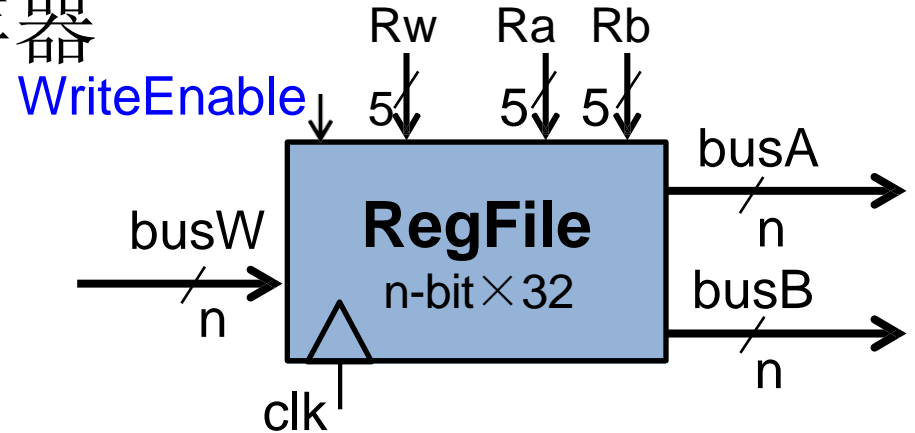
- (Register File)内部构成：32个寄存器

- 数据接口信号

- busW: 一个数据输入
- busA, busB: 两个数据输出

- 读写控制

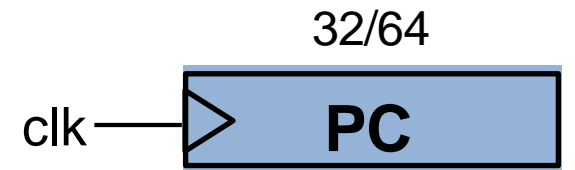
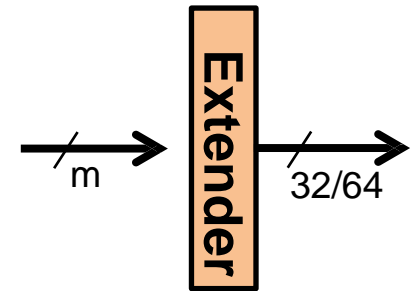
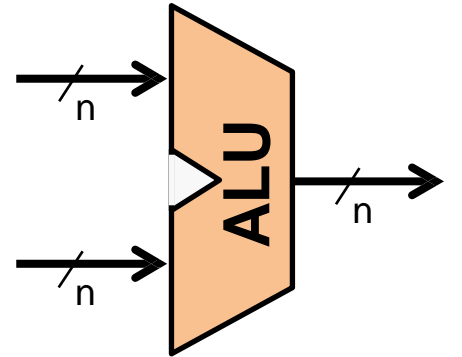
- Ra/Rb(5位): 将对应编号的寄存器的内容放到busA/busB(读)
- Rw(5位): 在时钟信号(**clk**)的上升沿, 如果写使能信号(**WriteEnable**) 有效, 将busW的内容存入**Rw号**寄存器。



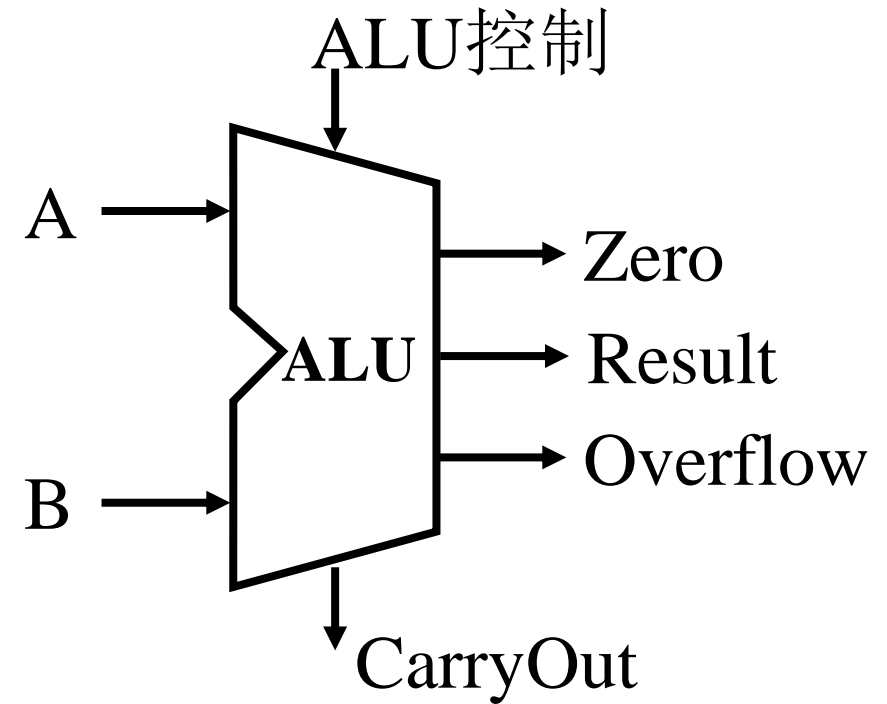
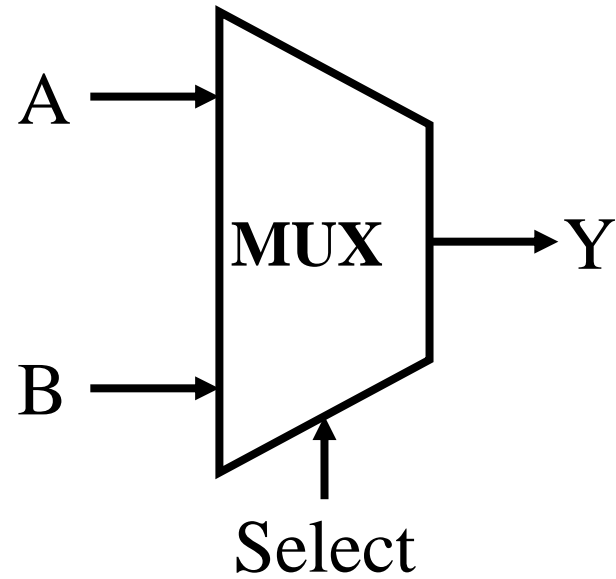
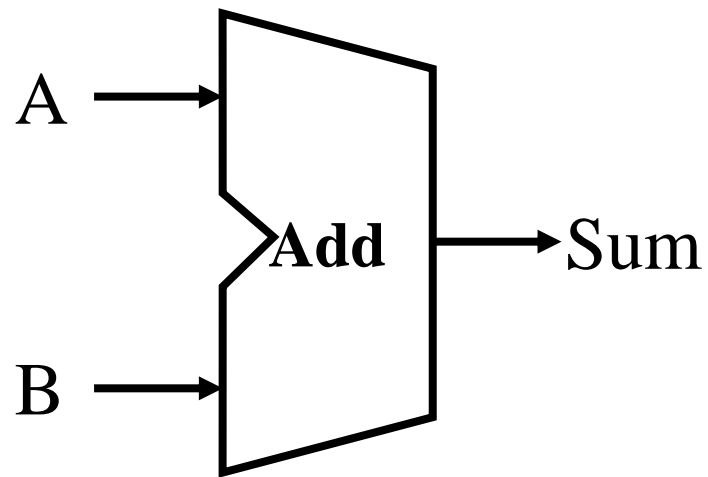
注：寄存器堆的读操作不受时钟控制

RISC-V指令系统的数据通路需求

- 算术逻辑单元（ALU）
 - 运算类型：加、减、或、比较等
 - 操作数：来自寄存器或扩展后的立即数
- 立即数扩展(生成)部件
 - 将一个不足32/64位的立即数扩展为32/64位
 - 扩展方式：零扩展、符号扩展
- 程序计数器（PC）
 - 一个32/64位的寄存器
 - 支持两种加法：加4 或 加一个立即数

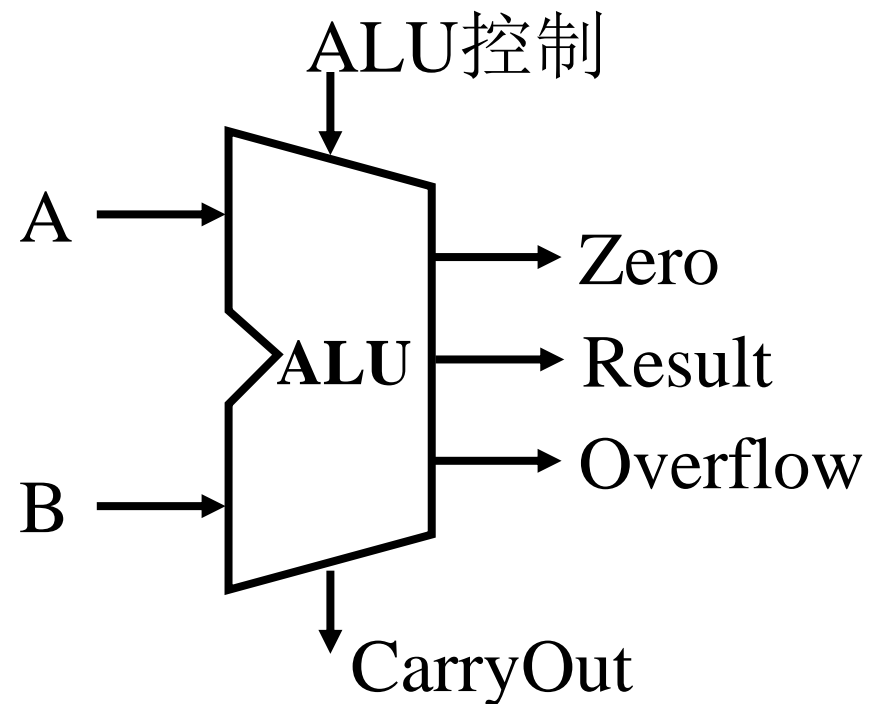


数据通路模块——组合逻辑单元



ALU功能需求

ALU控制	操作
0000	与
0001	或
0010	加
0110	减
0111	小于则置位(slt)
1100	或非(nor)

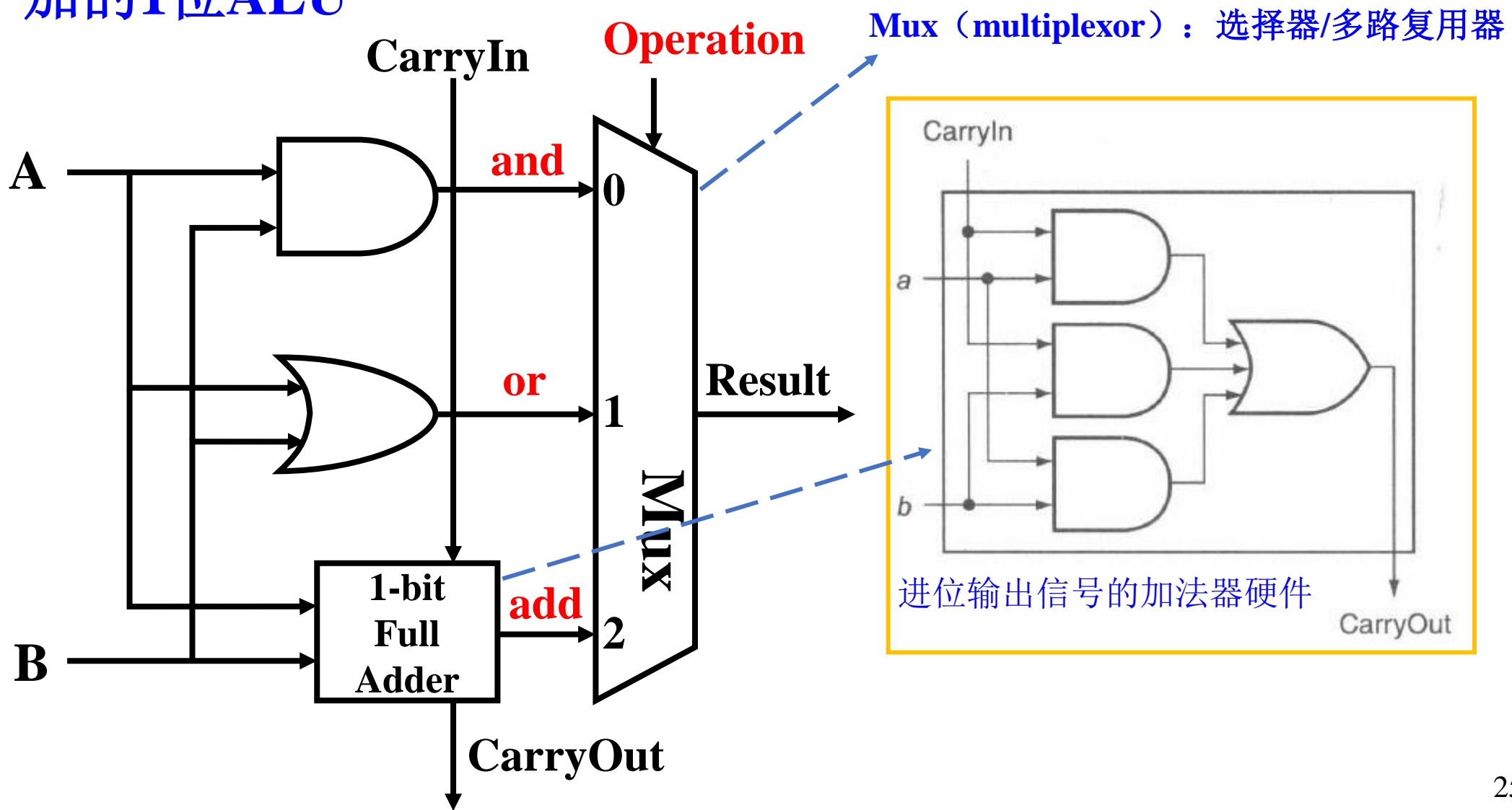


ALU设计技巧

- 从简单开始：然后再进行扩展
- 分而治之：从1位ALU设计开始
- 利用已知的元件或者组件，组合起来实现复杂功能

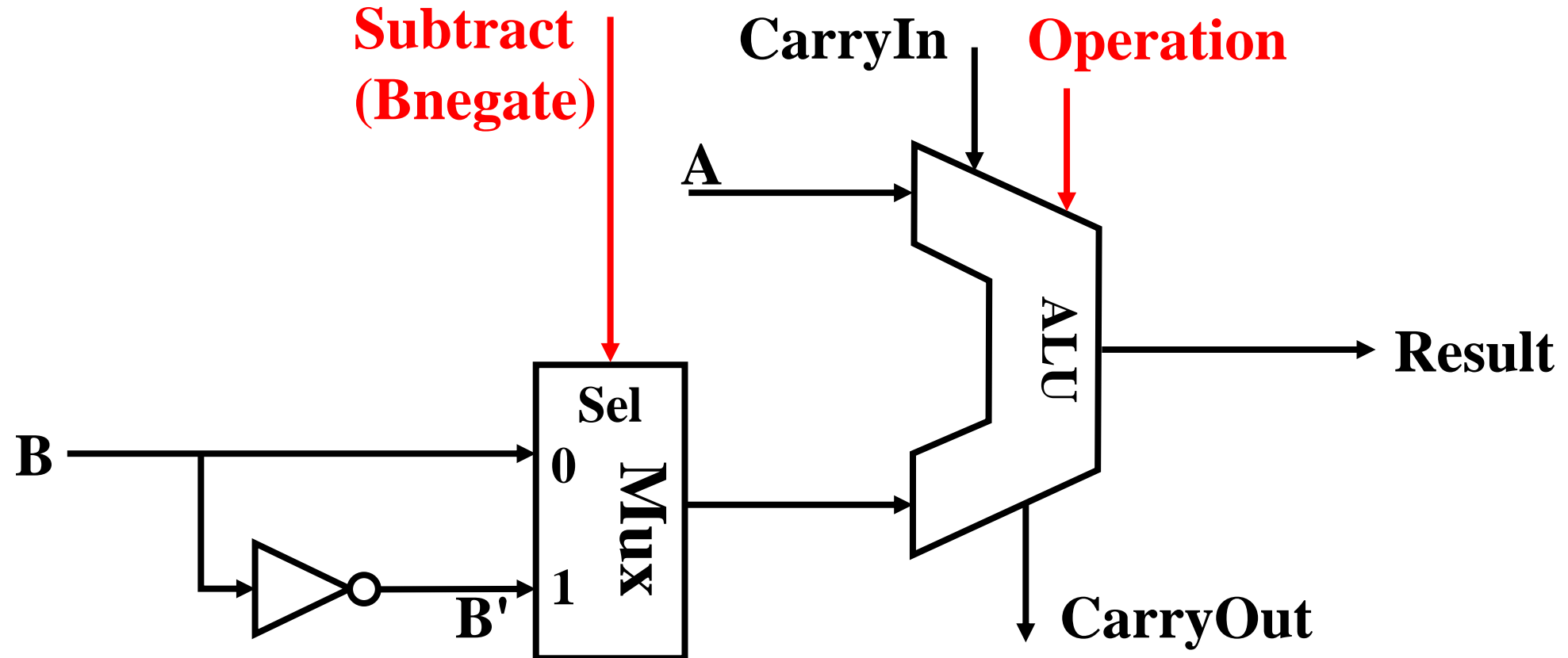
ALU——简单运算

与、或、加的1位ALU



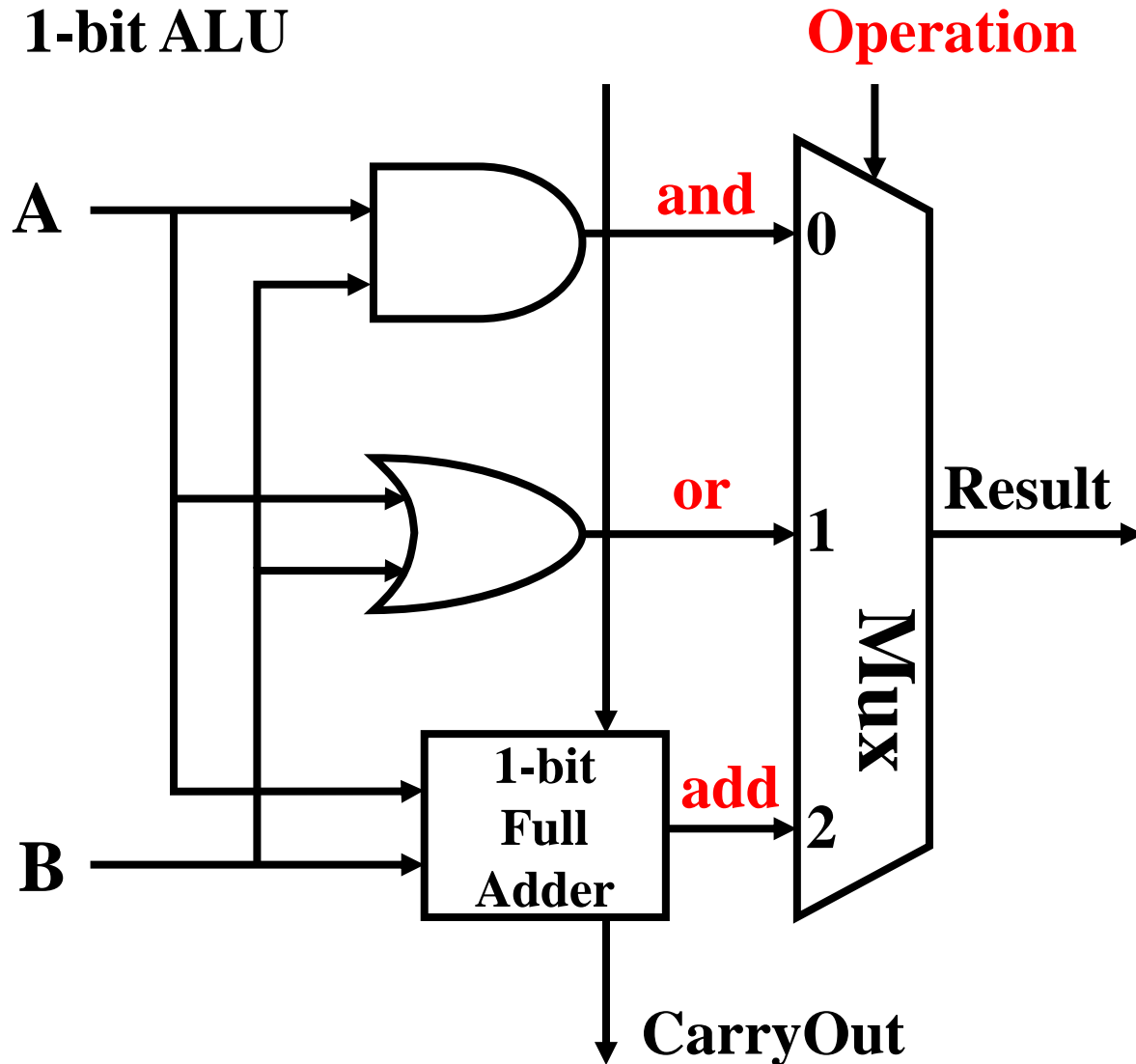
ALU——如何进行减法运算

- $A - B = A + B' + 1$ (补码)

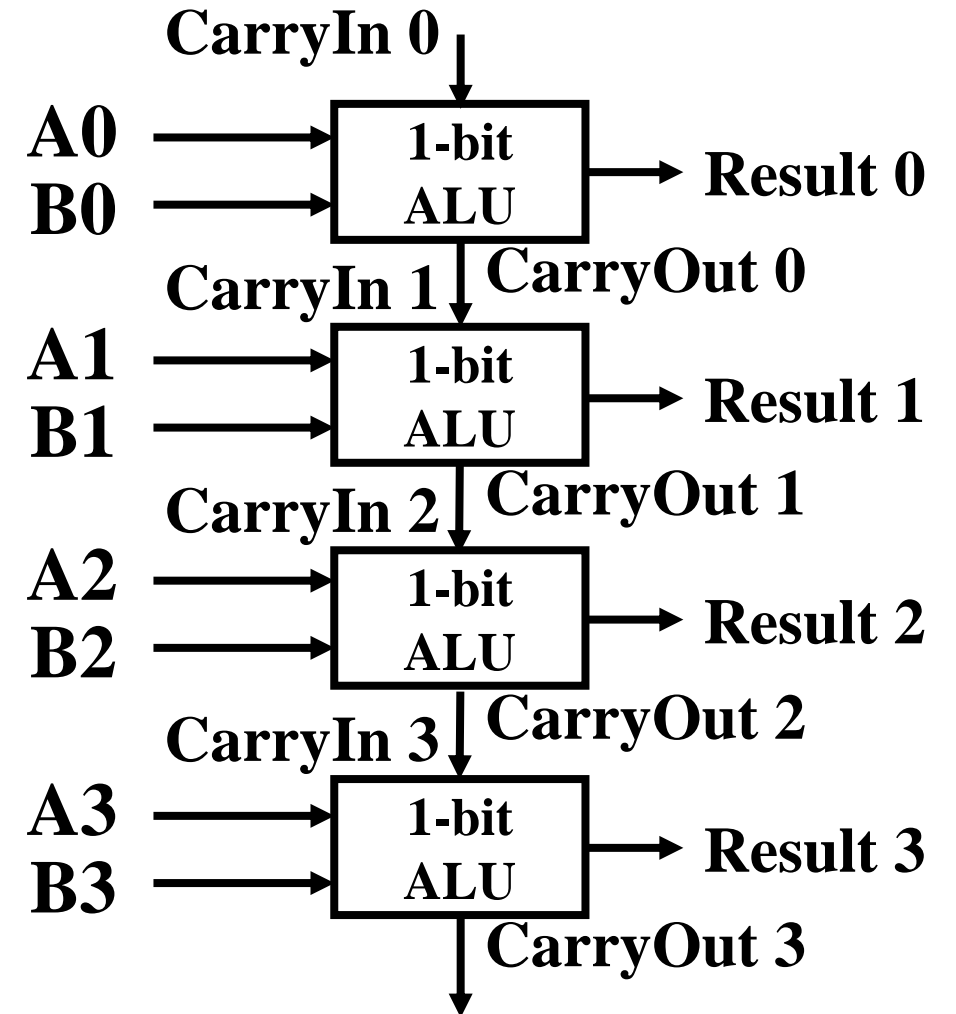


1-bit到多位ALU

1-bit ALU

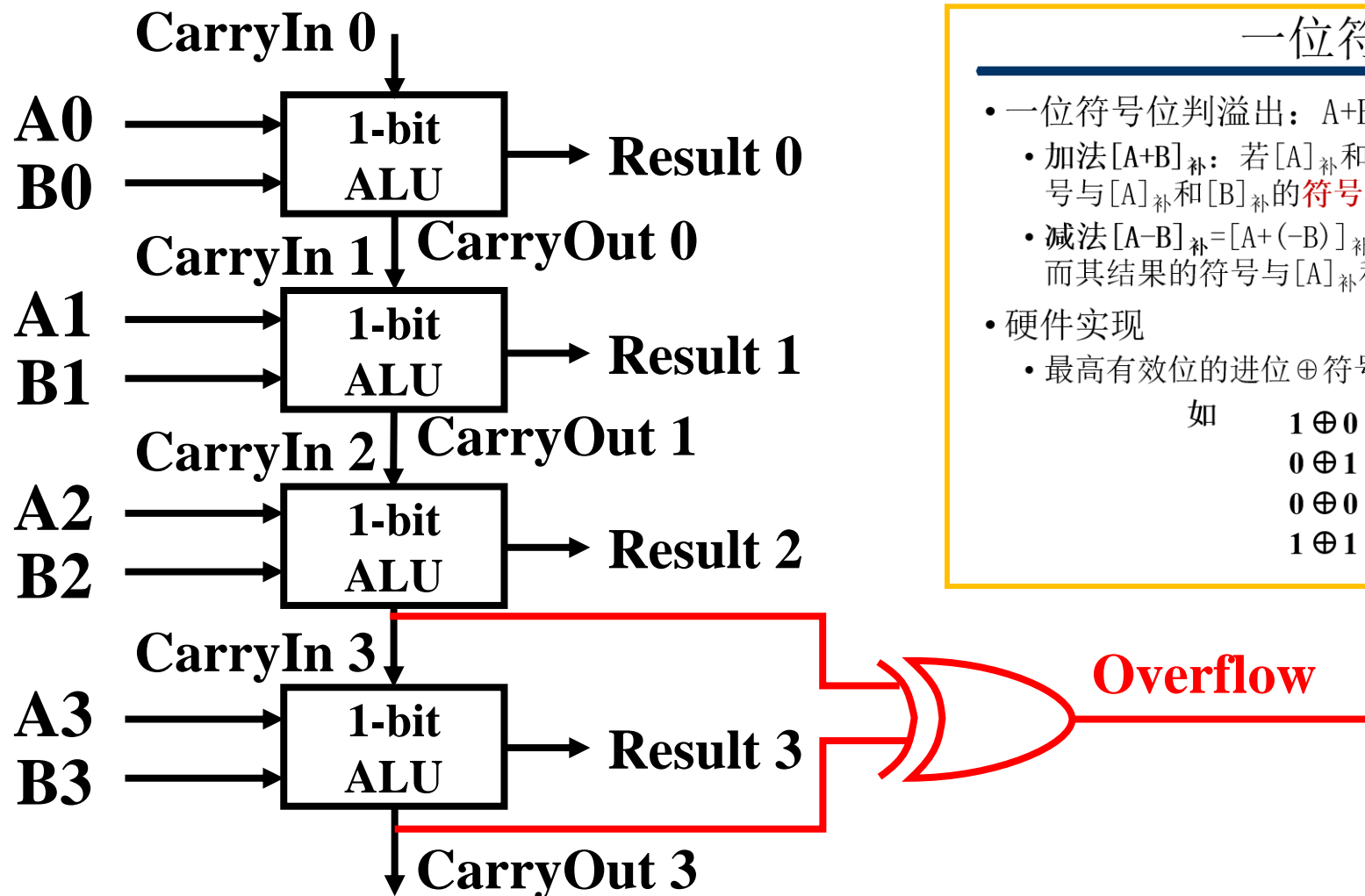


4-bit ALU



ALU——溢出检测逻辑

- $\text{Overflow} = \text{CarryIn}[N-1] \text{ xor } \text{CarryOut}[N-1]$



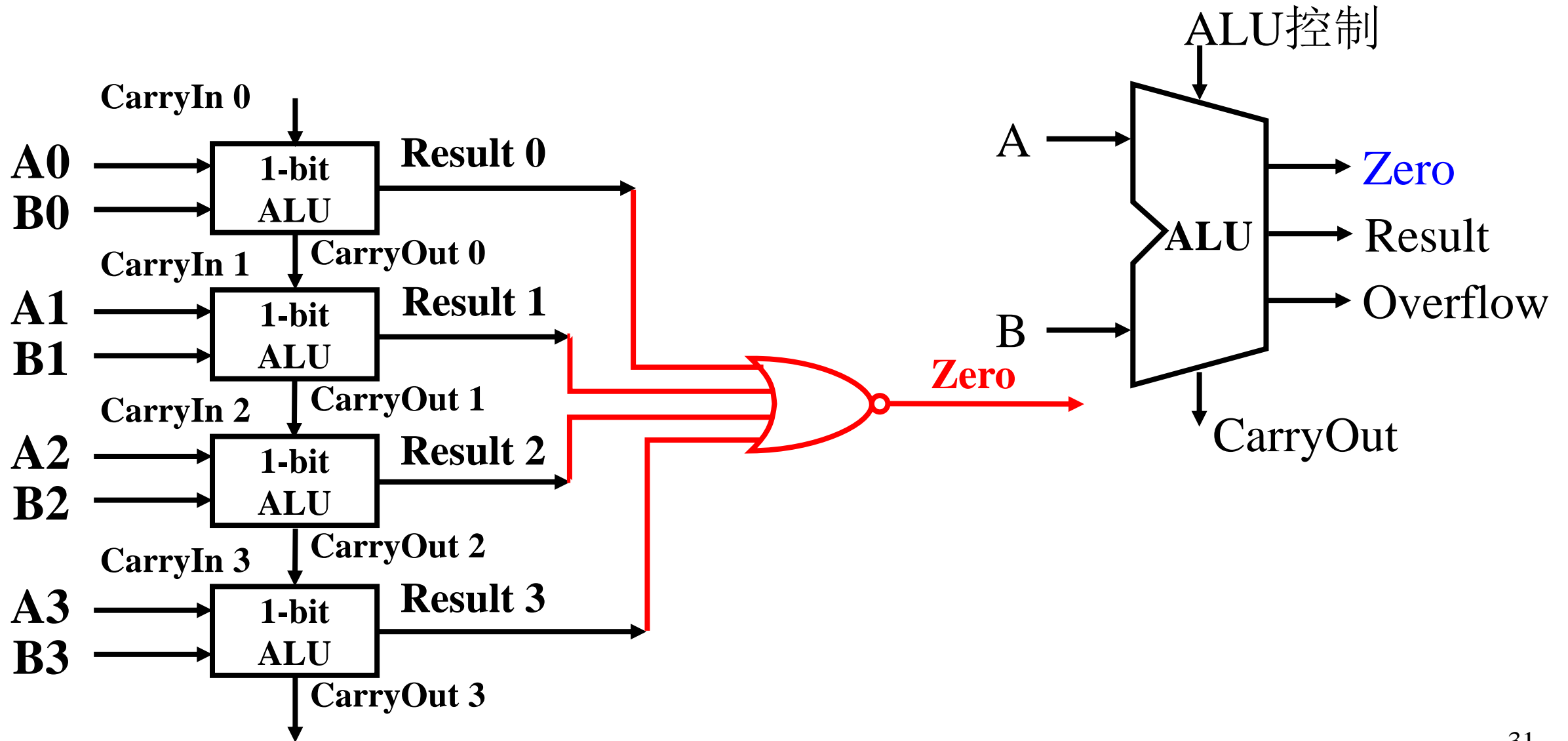
一位符号位判溢出

- 一位符号位判溢出: $A+B$
 - 加法 $[A+B]_{\text{补}}$: 若 $[A]_{\text{补}}$ 和 $[B]_{\text{补}}$ 两个数符号相同, 而其结果的符号与 $[A]_{\text{补}}$ 和 $[B]_{\text{补}}$ 的符号不同, 即为溢出
 - 减法 $[A-B]_{\text{补}} = [A+(-B)]_{\text{补}}$: 若 $[A]_{\text{补}}$ 和 $[-B]_{\text{补}}$ 两个数符号相同, 而其结果的符号与 $[A]_{\text{补}}$ 和 $[-B]_{\text{补}}$ 的符号不同, 即为溢出
- 硬件实现
 - 最高有效位的进位 \oplus 符号位的进位 = 1, 溢出

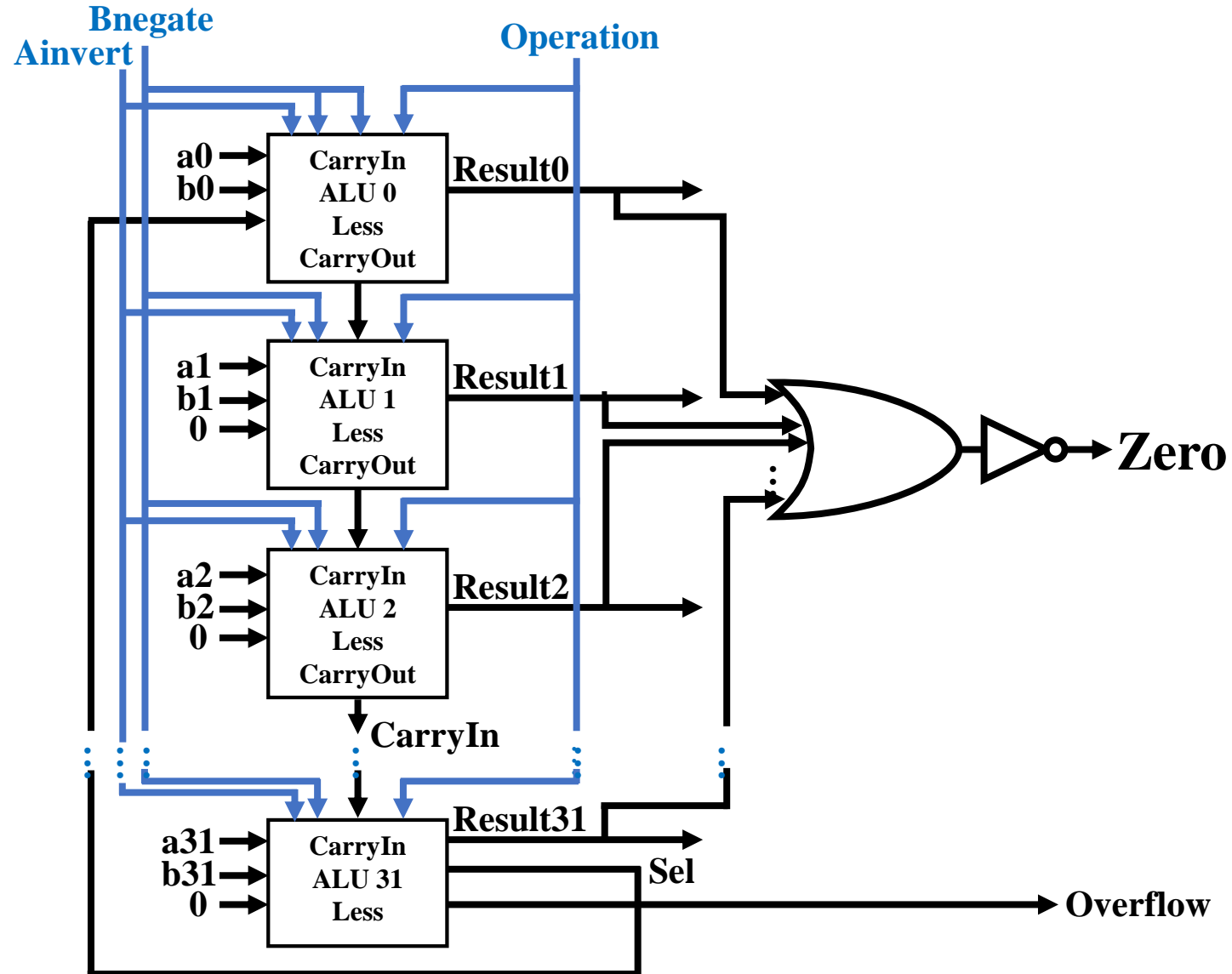
如

$1 \oplus 0 = 1$	} 有 溢出
$0 \oplus 1 = 1$	
$0 \oplus 0 = 0$	} 无 溢出
$1 \oplus 1 = 0$	

ALU——判零逻辑



32bit ALU



ALU控制	Funciton
0000	and
0001	or
0010	add
0110	subtract
0111	set-on-less-than
1100	nor

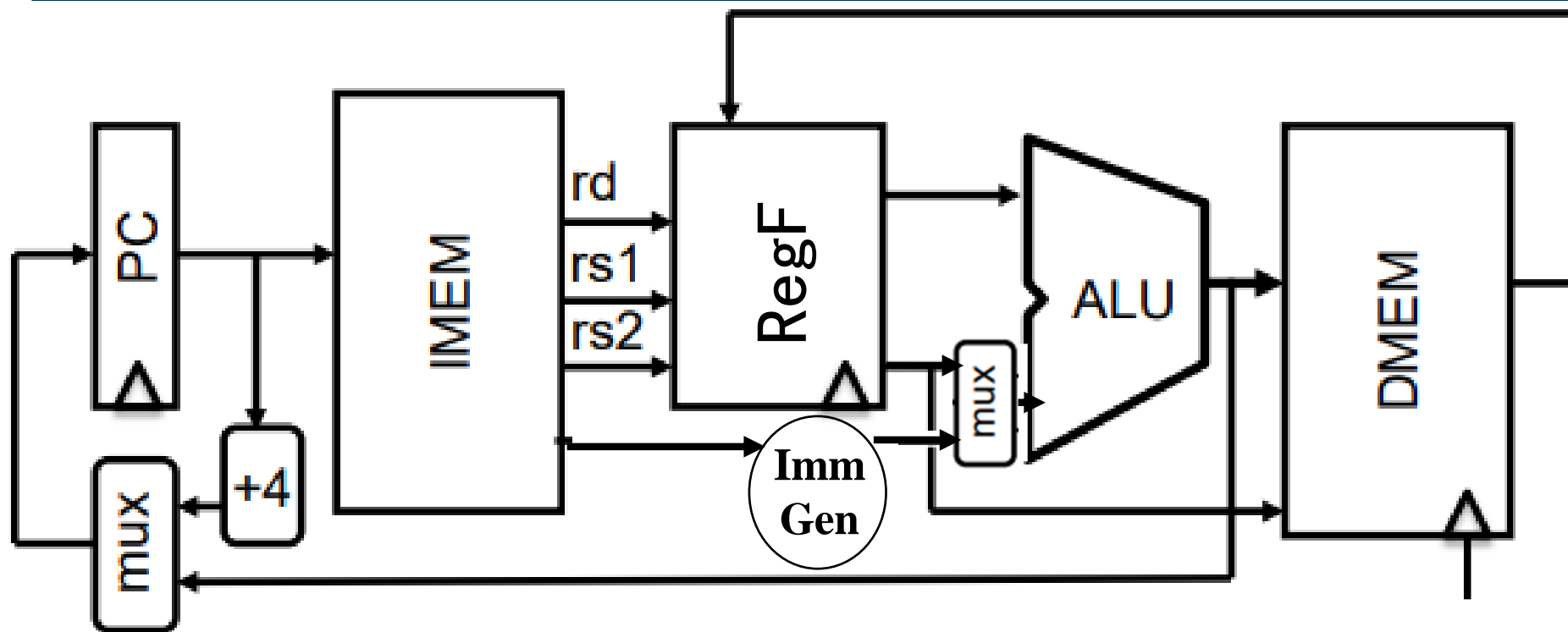
第四章 处理器设计

- 处理器设计的需求分析
- RISC-V数据通路的组件选择
- RISC-V部分指令的数据通路设计
- RISC-V控制器

处理器的设计步骤

- ① 分析指令系统，得出对数据通路的需求
- ② 为数据通路选择合适的组件
- ③ 根据指令需求连接组件建立数据通路
- ④ 分析每条指令的实现，以确定控制信号
- ⑤ 集成控制信号，形成完整的控制逻辑

指令执行示意图



程序计数器PC
指令存储器IMEM
寄存器堆RegF
算术逻辑单元ALU
数据存储器DMEM

1. Instruction
Fetch

2. Decode/
Register
Read

3. Execute

4. Memory

5. Register
Write

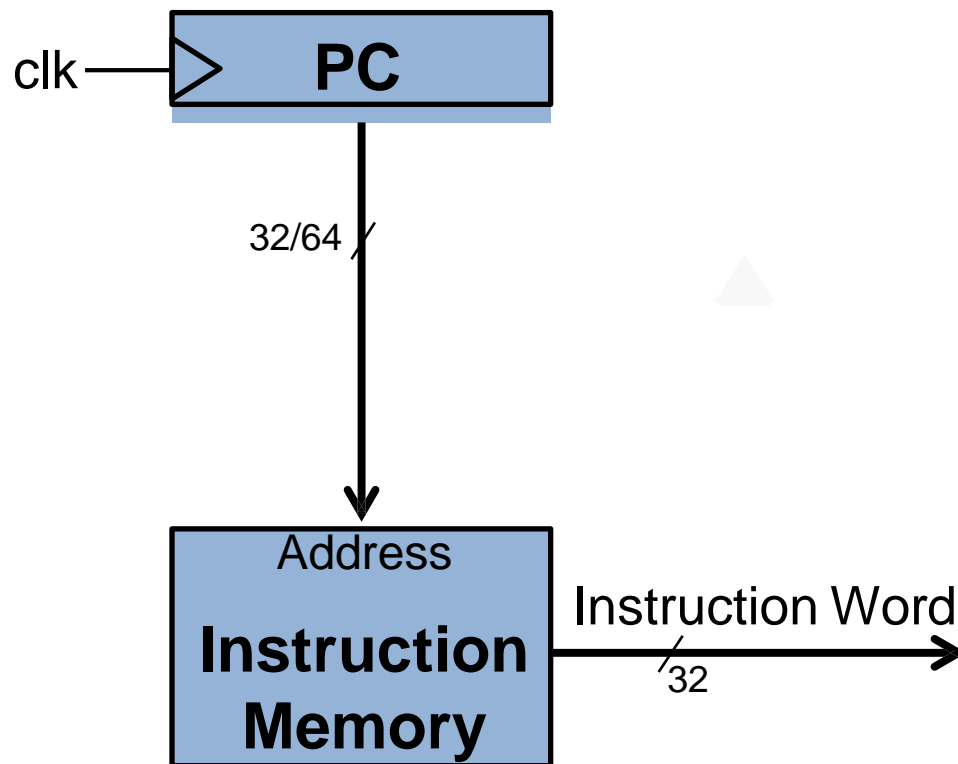
time →

数据通路的五个阶段

- 取指: Instruction Fetch (IF)
- 译码: Instruction Decode (ID)
- 执行: EXecute (EX) - ALU (Arithmetic-Logic Unit)
- 访存: MEMory access (MEM)
- 写回: Write Back to register (WB)

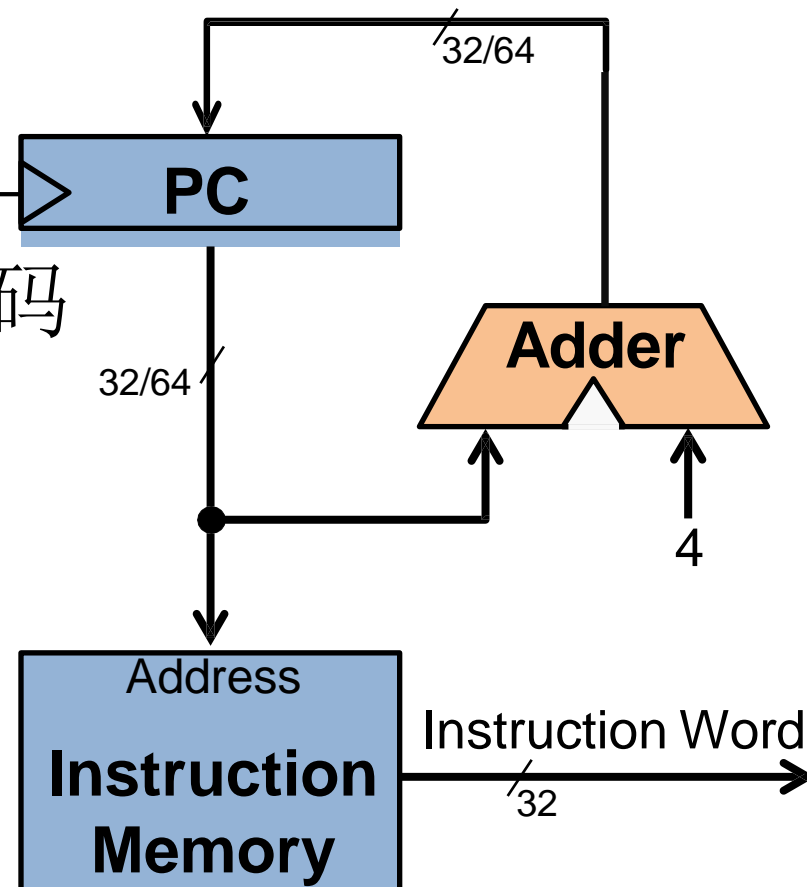
所有指令的共同需求——取指令

- 程序计数器（PC）的内容是指令的地址
- 用PC的内容作为地址，访问指令存储器获得指令编码



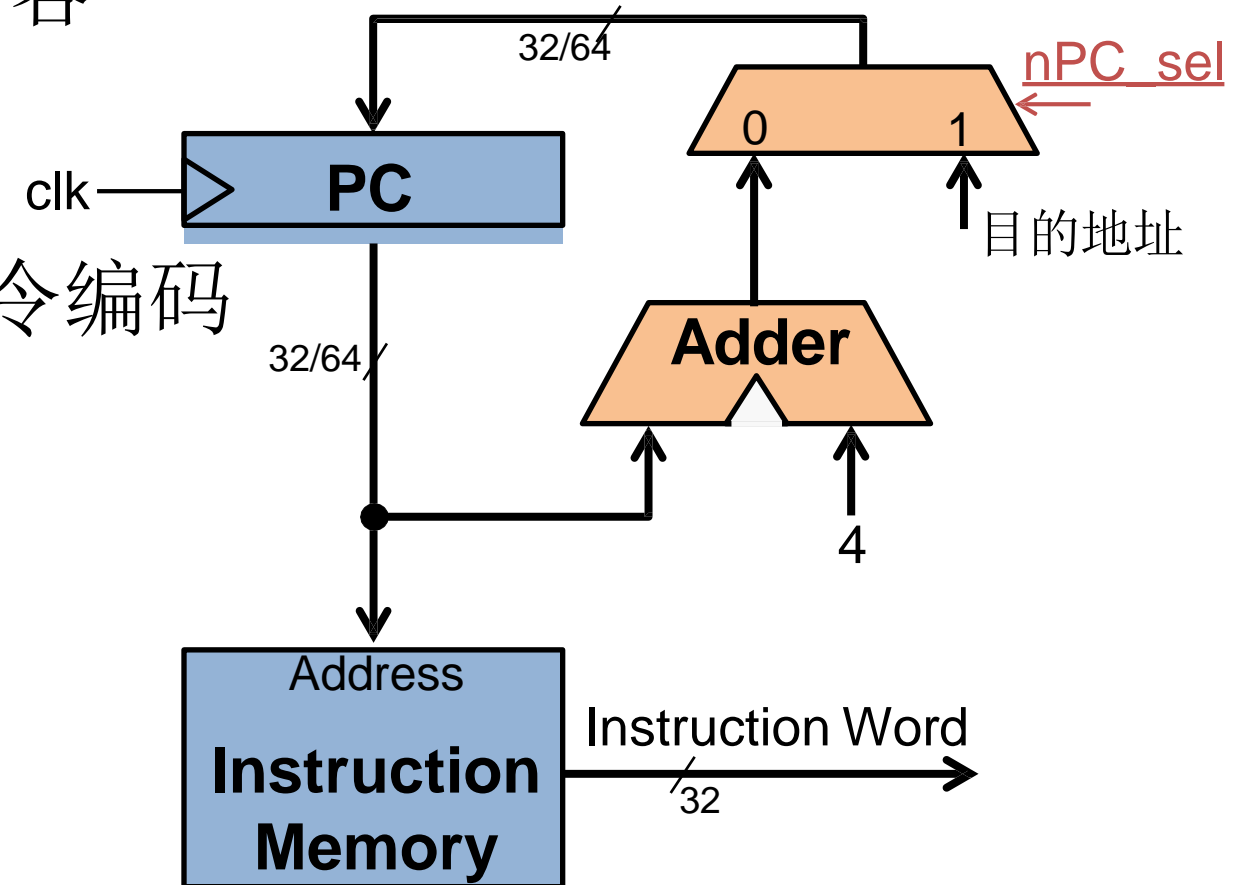
所有指令的共同需求——更新PC

- 取指令
 - 程序计数器（PC）的内容是指令的地址
 - 用PC的内容作为地址，访问指令存储器获得指令编码
- 顺序执行时更新PC
 - $PC \leftarrow PC + 4$
- 发生分支时更新
 - $PC \leftarrow$ 分支目标地址



所有指令的共同需求——更新PC（分支）

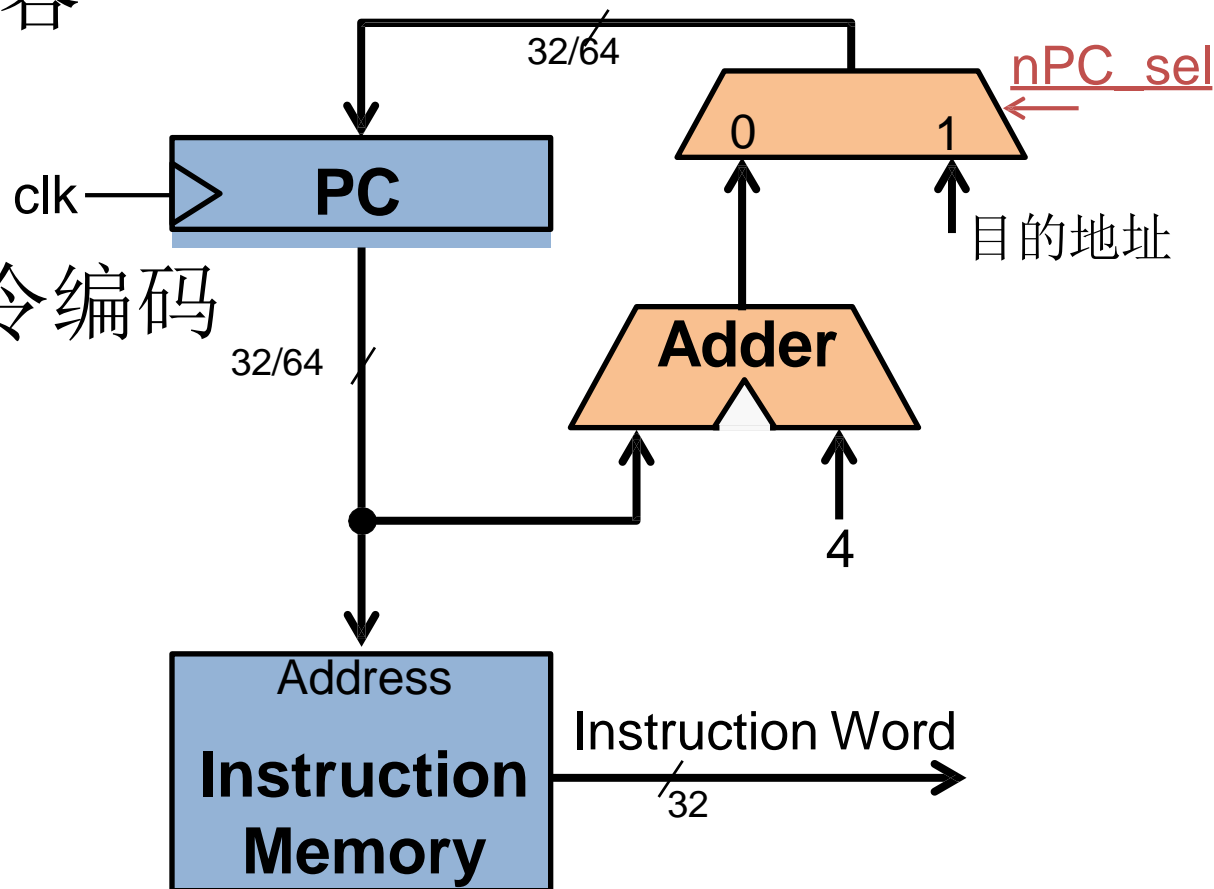
- 取指令
 - 程序计数器（PC）的内容是指令的地址
 - 用PC的内容作为地址，访问指令存储器获得指令编码
- 顺序执行时更新PC
 - $PC \leftarrow PC + 4$
- 发生分支时更新
 - $PC \leftarrow$ 分支目标地址



所有指令的共同需求

- 取指令
 - 程序计数器（PC）的内容是指令的地址
 - 用PC的内容作为地址，访问指令存储器获得指令编码
- 顺序执行时更新PC
 - $PC \leftarrow PC + 4$
- 发生分支时更新
 - $PC \leftarrow$ 分支目标地址

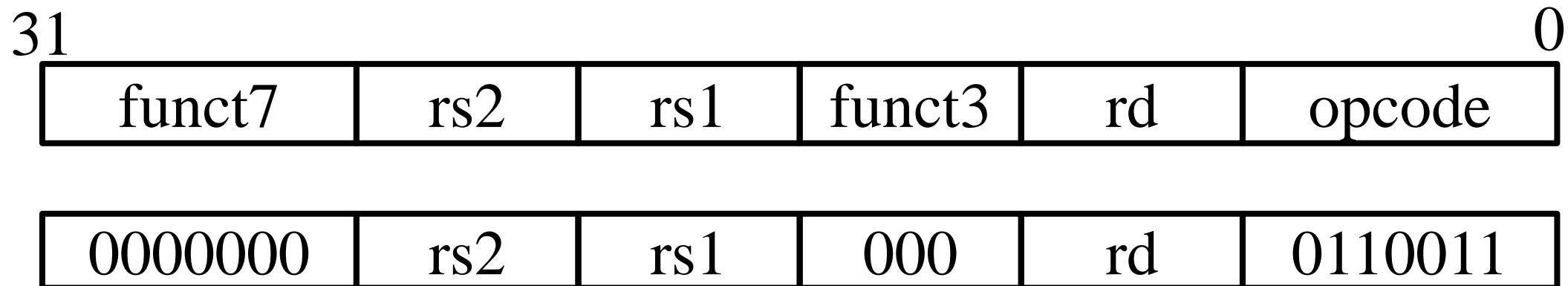
Instruction Fetch Unit(IFU)



处理器的设计步骤

- ① 分析指令系统，得出对数据通路的需求
- ② 为数据通路选择合适的组件
- ③ 根据指令需求连接组件建立数据通路
- ④ 分析每条指令的实现，以确定控制信号
- ⑤ 集成控制信号，形成完整的控制逻辑

实现add指令

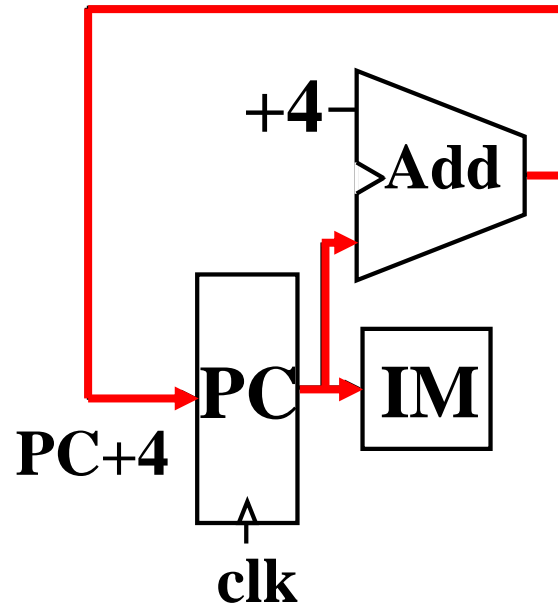


- add rd, rs1, rs2
- 指令对机器状态进行两次更改:
 - $\text{Reg}[\text{rd}] = \text{Reg}[\text{rs1}] + \text{Reg}[\text{rs2}]$
 - $\text{PC} = \text{PC} + 4$

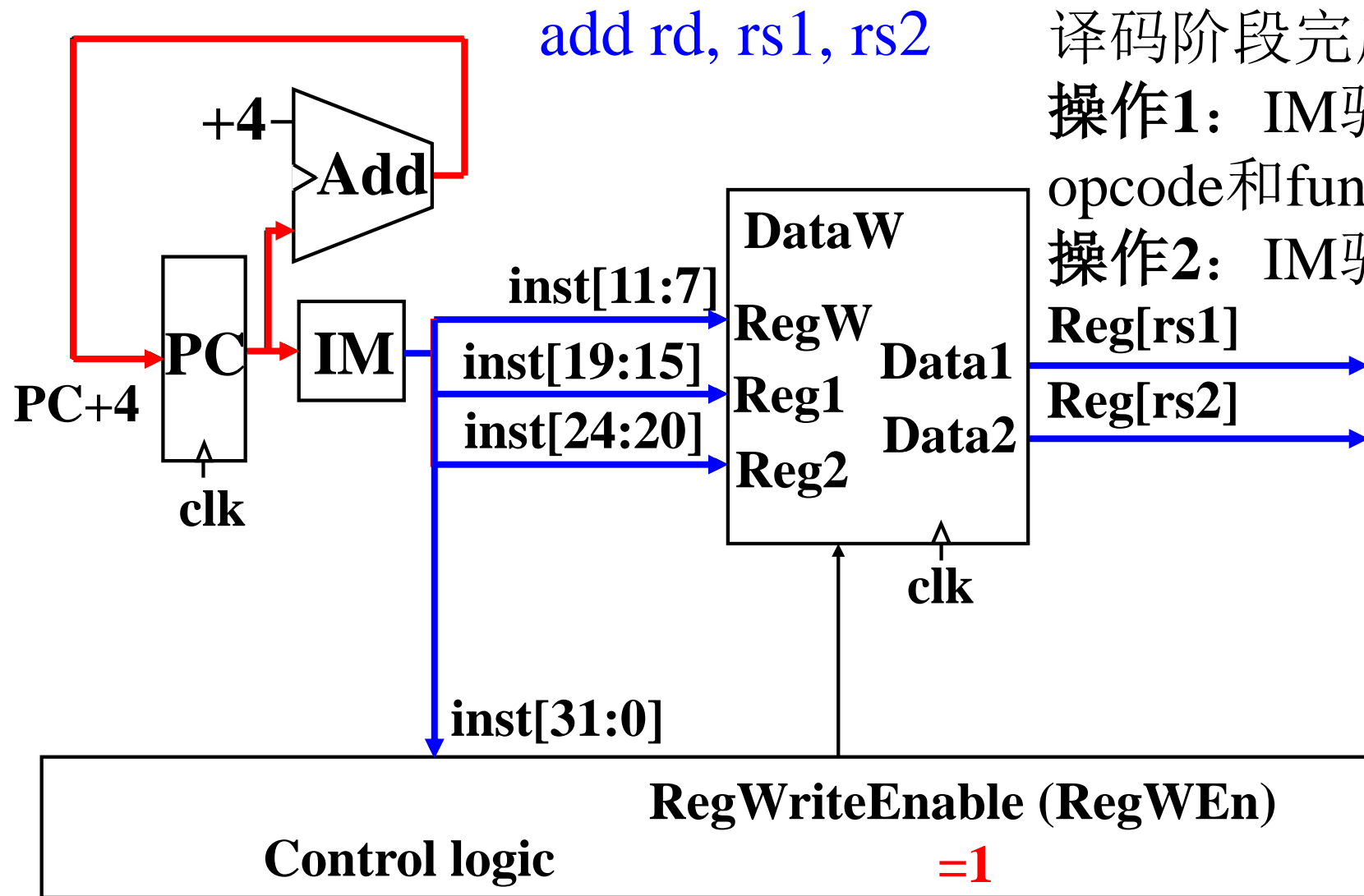
add指令的数据通路——取指

取指阶段：

- 操作1：根据PC，从IM中读出指令
- 操作2：更新PC值，加法器计算下一个PC值



add指令的数据通路——译码



译码阶段完成:

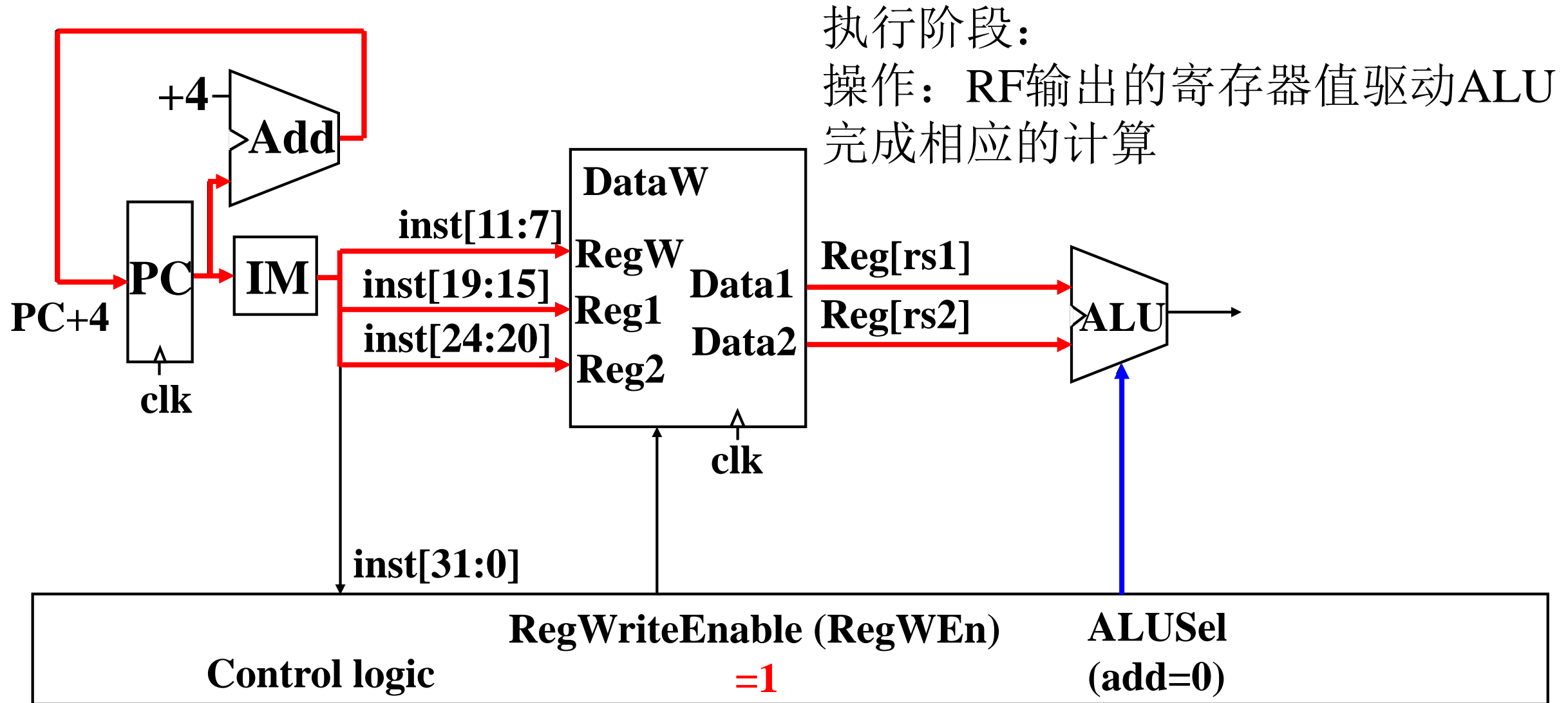
操作1: IM驱动控制器分析指令的 opcode和funct域

操作2: IM驱动RF读出2个寄存器值

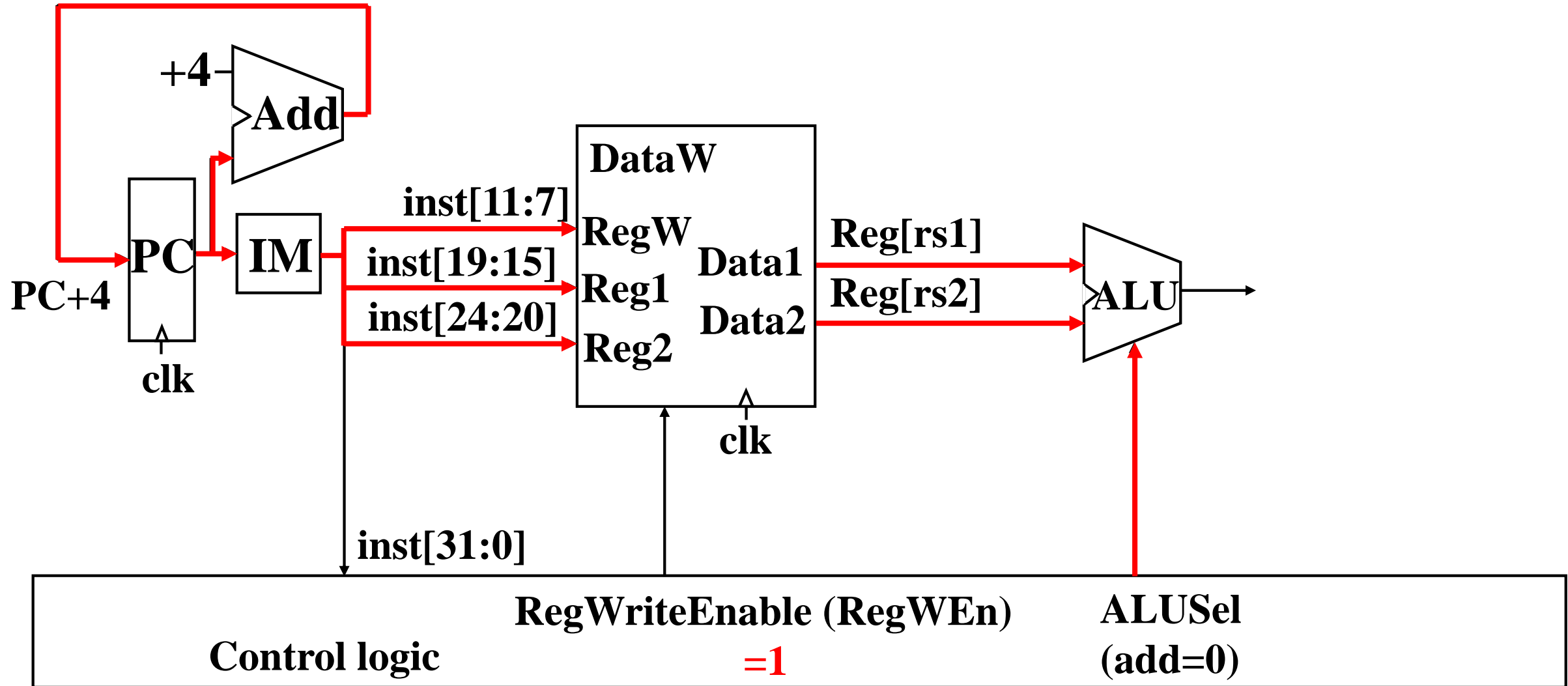
Reg[rs1]

Reg[rs2]

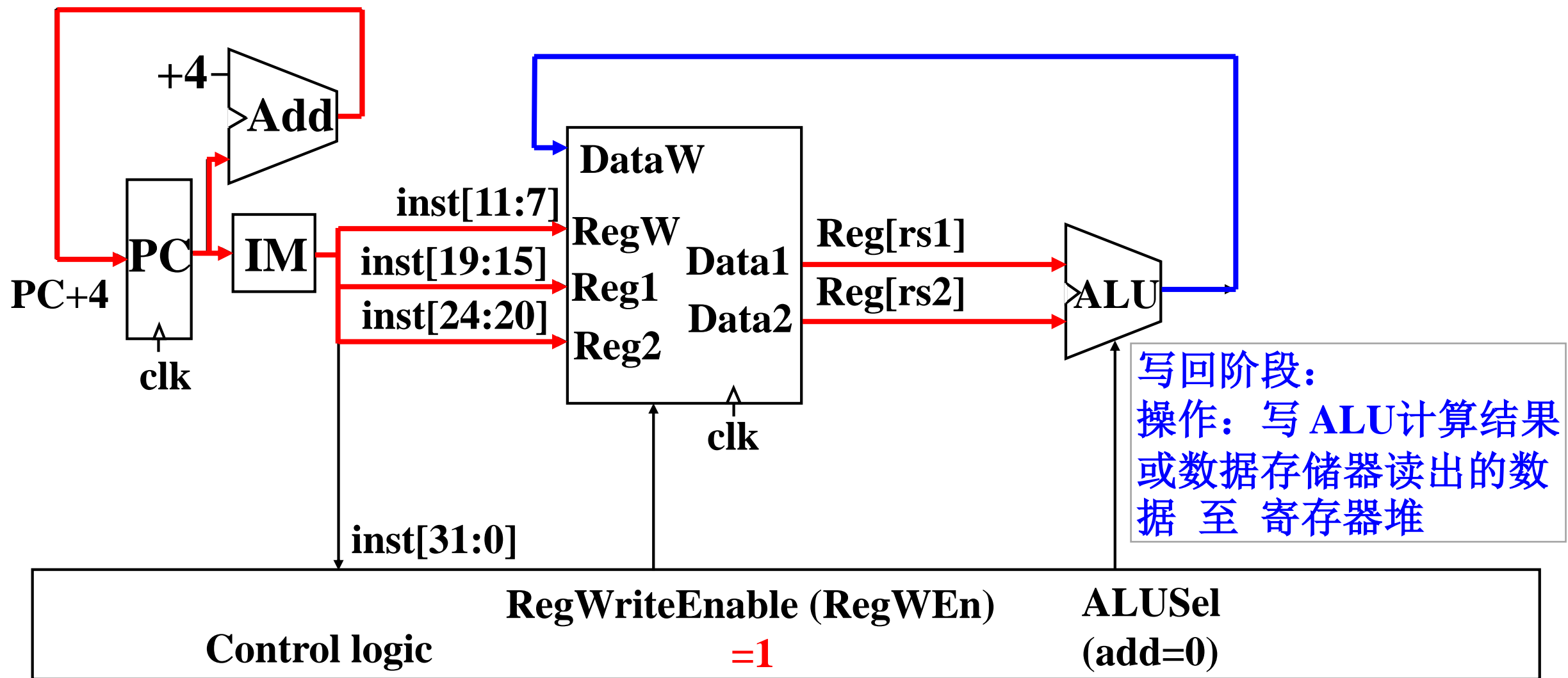
add指令的数据通路——执行



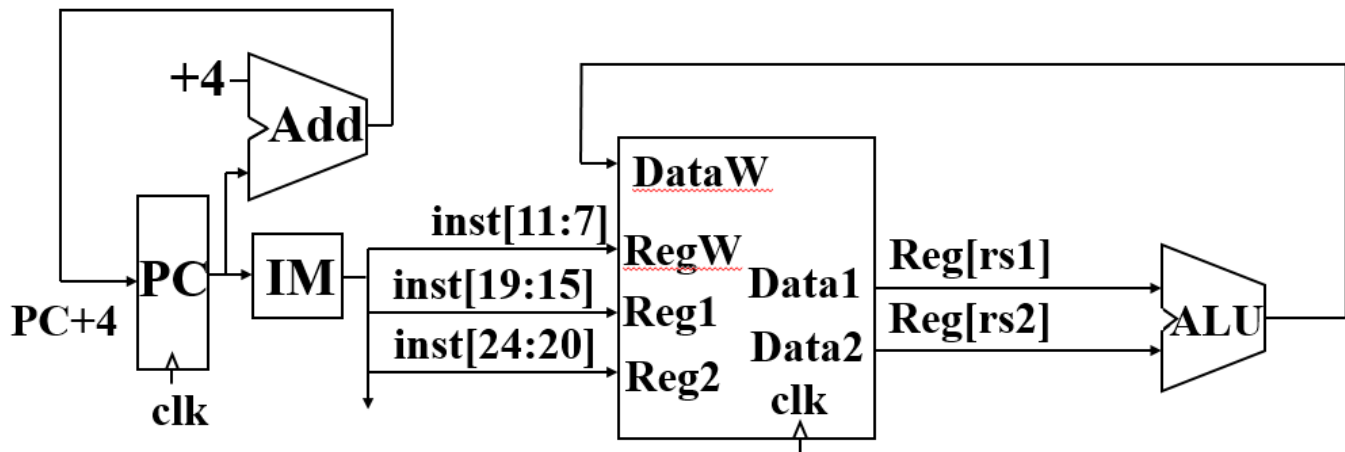
add指令的数据通路



add指令的数据通路——写回寄存器



add指令的时序图

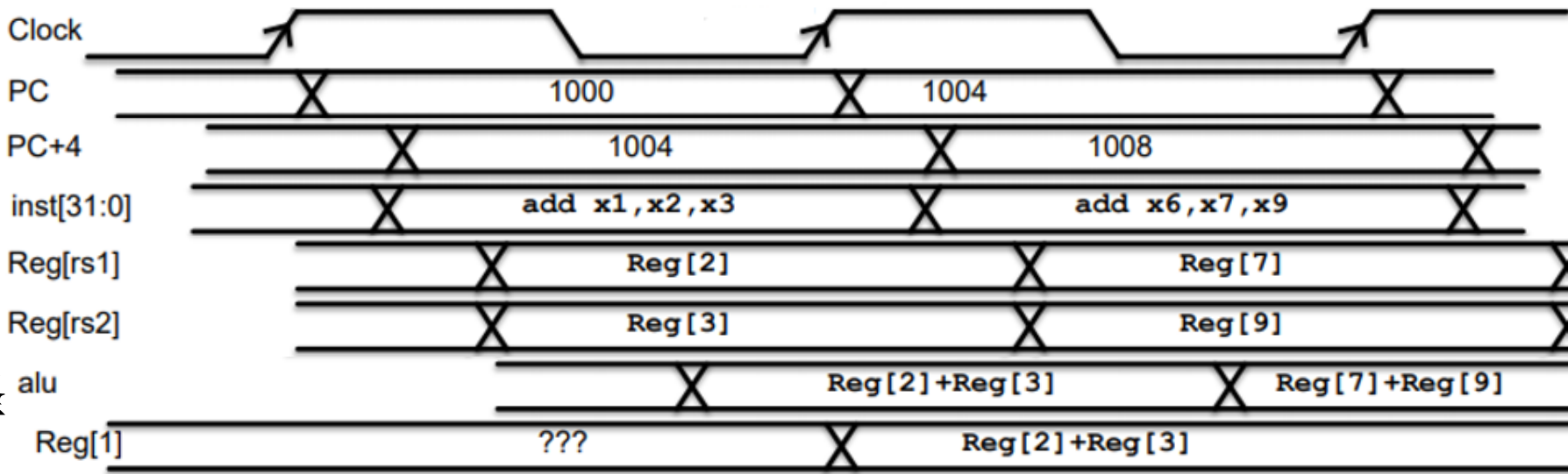


PC寄存器的时延

指令存储器的时延

ALU时延

流经的选择器的时延



max(寄存器文件时延, 立即数生成单元时延)

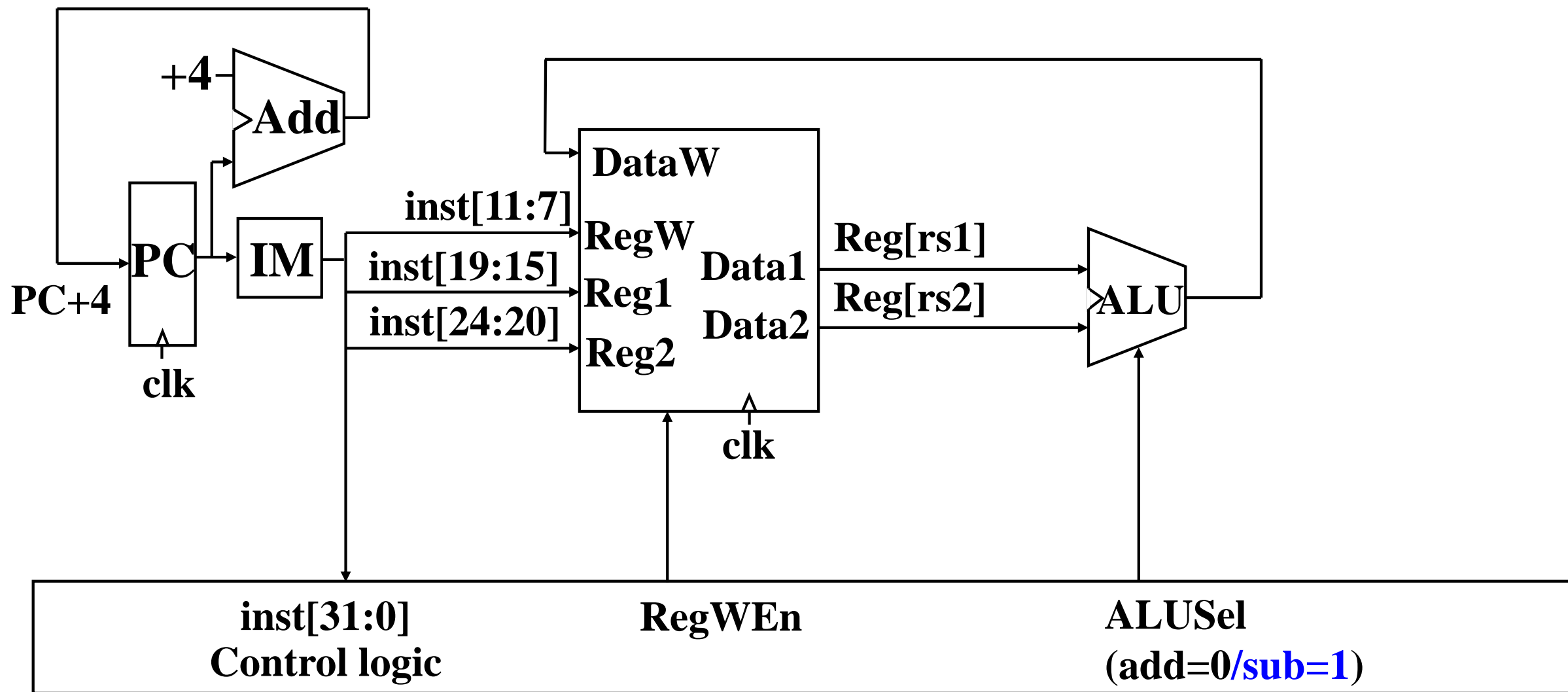
实现sub指令

31						0	
	00000000	rs2	rs1	funct3	rd	0110011	add
	01000000	rs2	rs1	000	rd	0110011	sub

sub rd, rs1, rs2

- 几乎与加法相同
- inst[30]在加法和减法之间进行选择

sub指令的数据通路



R型部分指令

00000000	rs2	rs1	000	rd	0110011	add
01000000	rs2	rs1	000	rd	0110011	sub
00000000	rs2	rs1	001	rd	0110011	sll
00000000	rs2	rs1	010	rd	0110011	slt
00000000	rs2	rs1	011	rd	0110011	sltu
00000000	rs2	rs1	100	rd	0110011	xor
00000000	rs2	rs1	101	rd	0110011	srl
01000000	rs2	rs1	101	rd	0110011	sra
00000000	rs2	rs1	110	rd	0110011	or
00000000	rs2	rs1	111	rd	0110011	and

opcode、funct3和funct7字段确定ALU的不同功能

实现RISC-V addi指令

- addi x15, x1, -50

31

0

imm[11:0]	rs1	funct3	rd	opcode
111111001110	00001	000	01111	0010011

imm= -50

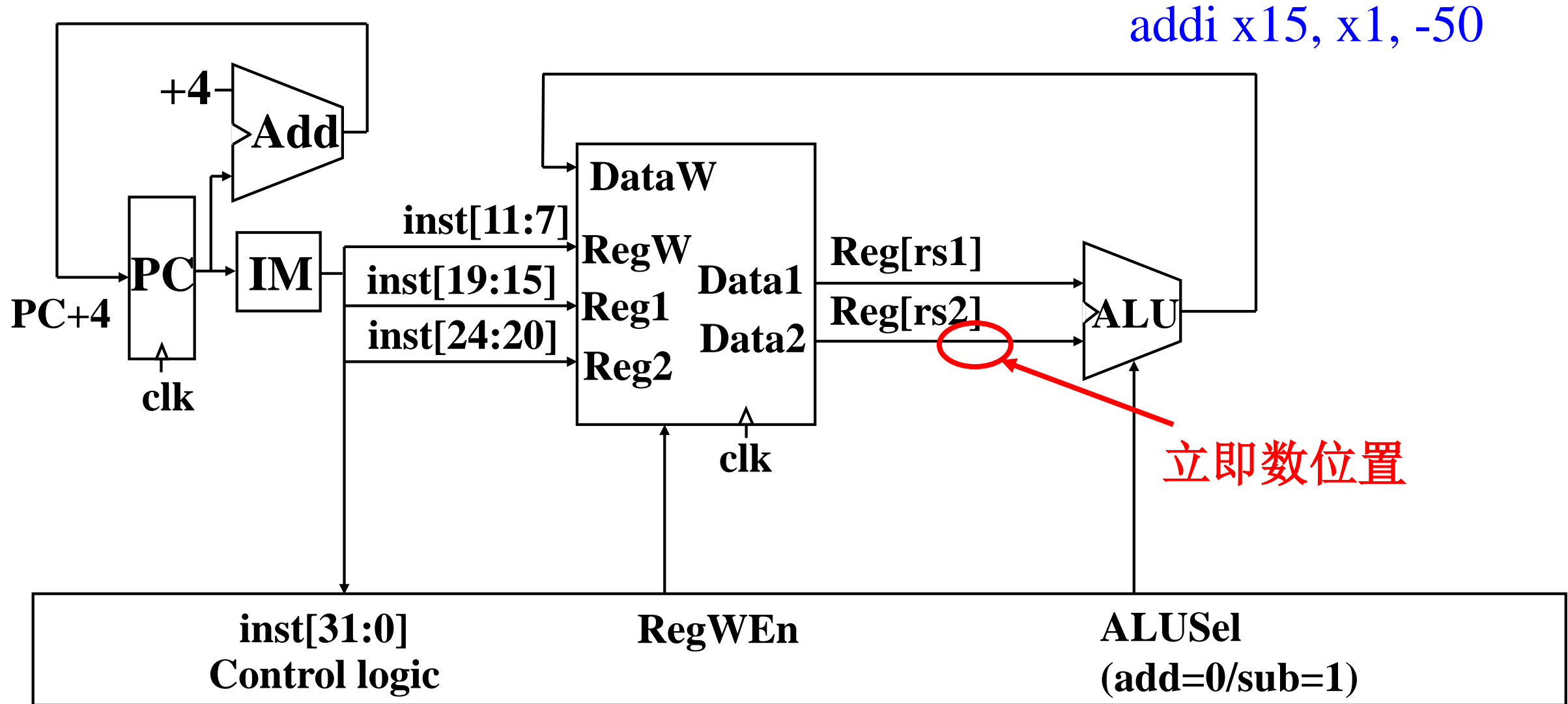
rs1=1

add

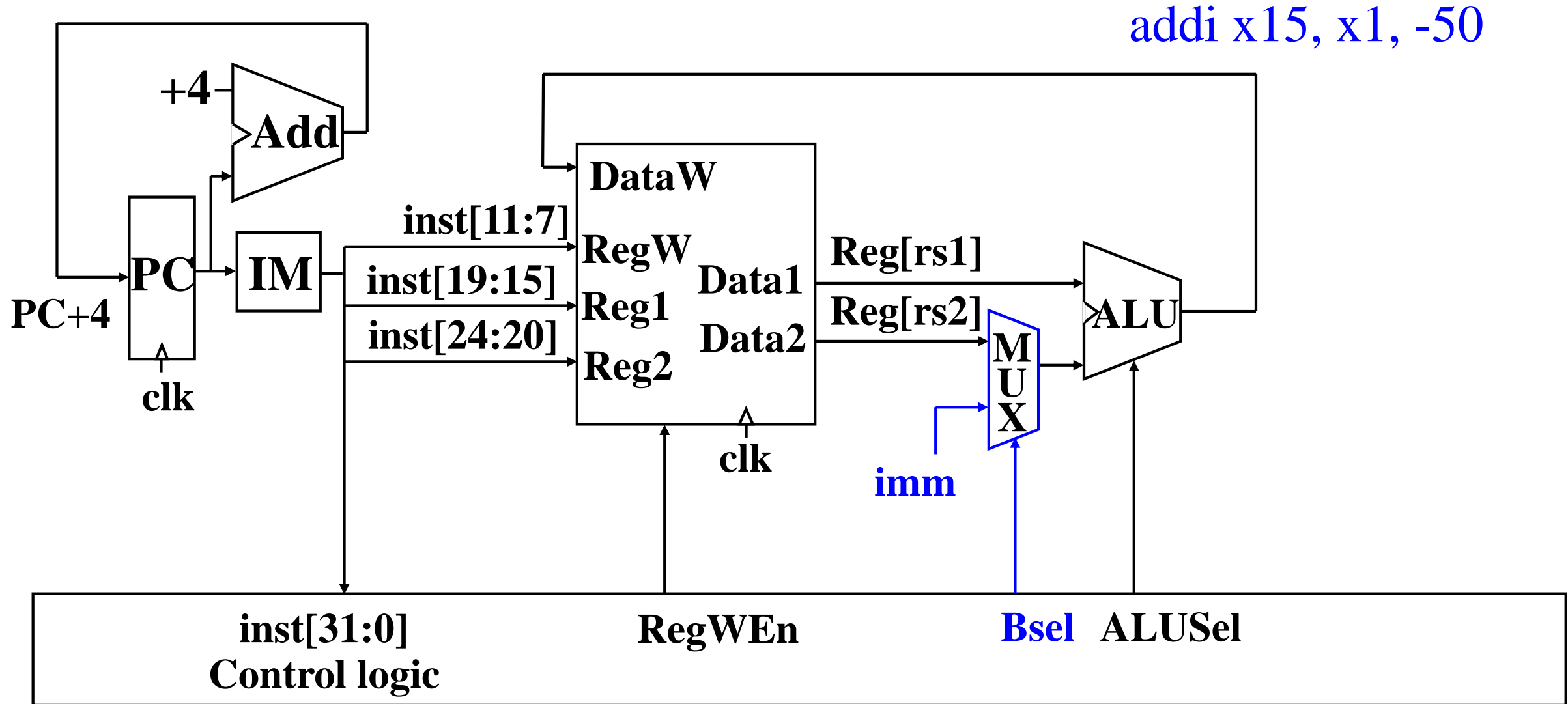
rd=15

OP-Imm

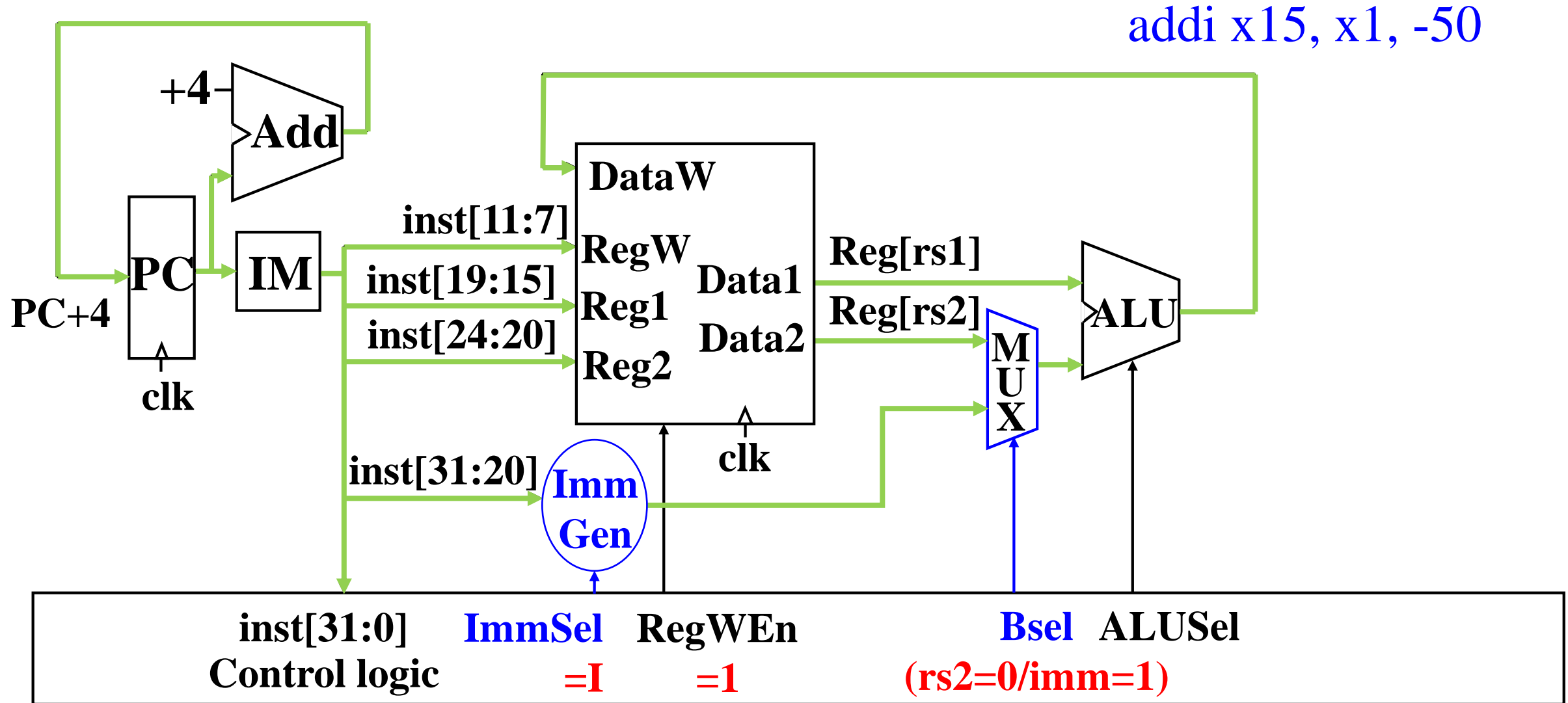
add指令的数据通路



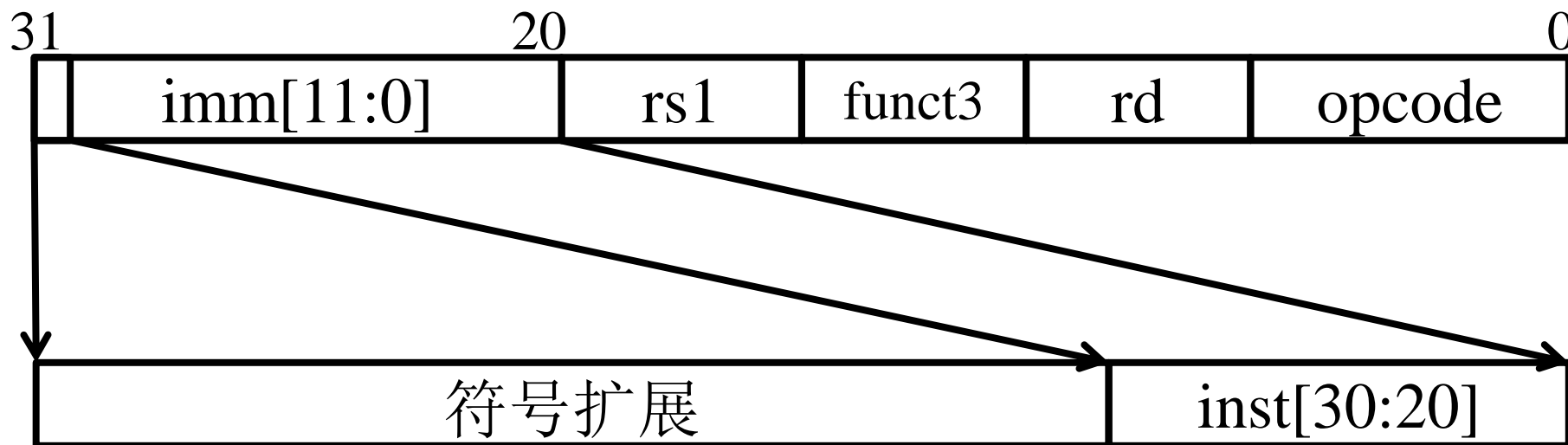
addi指令的数据通路



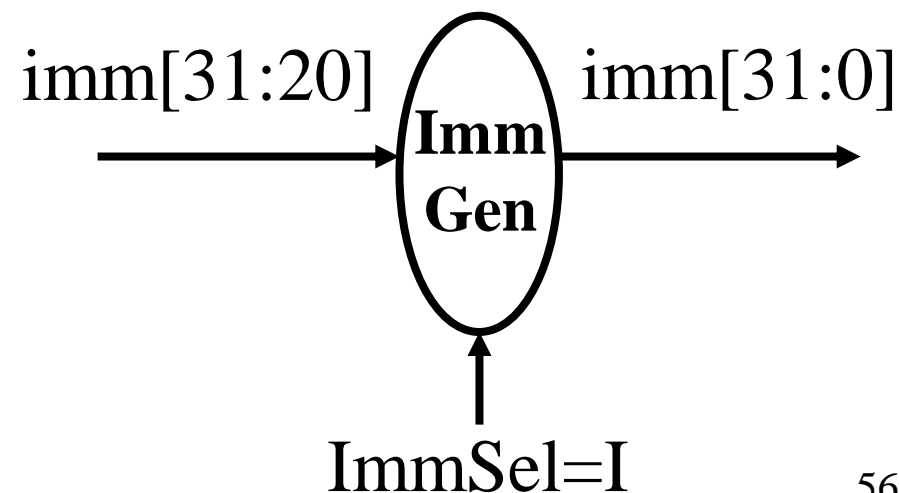
addi指令的数据通路



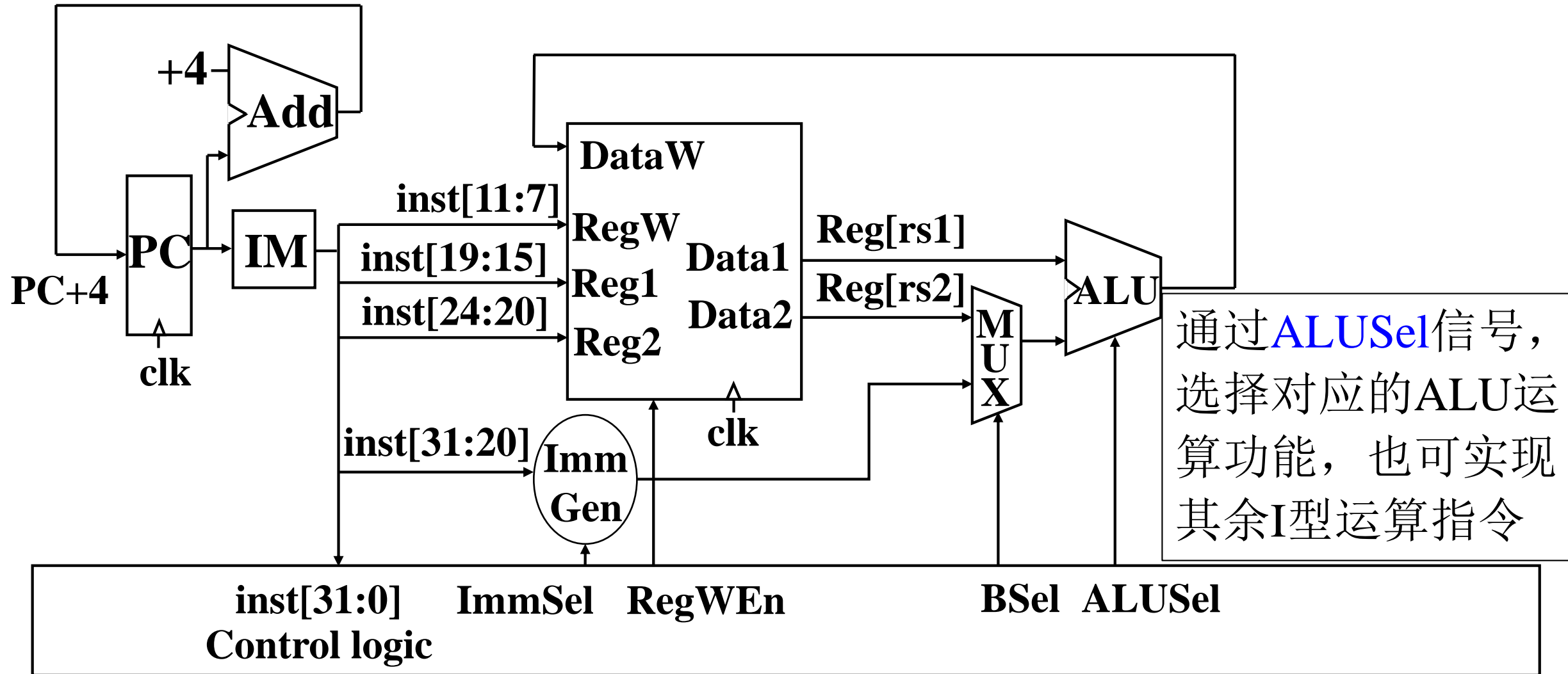
I型指令的立即数生成



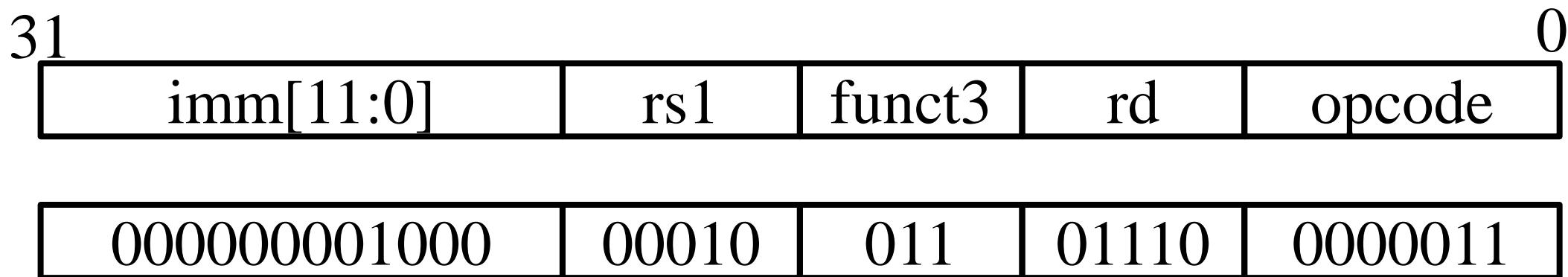
- 高12位复制到立即数的低12位
- 通过将指令的最高比特位复制填充到立即数的高20位完成符号扩展



R型和I型共用的数据通路



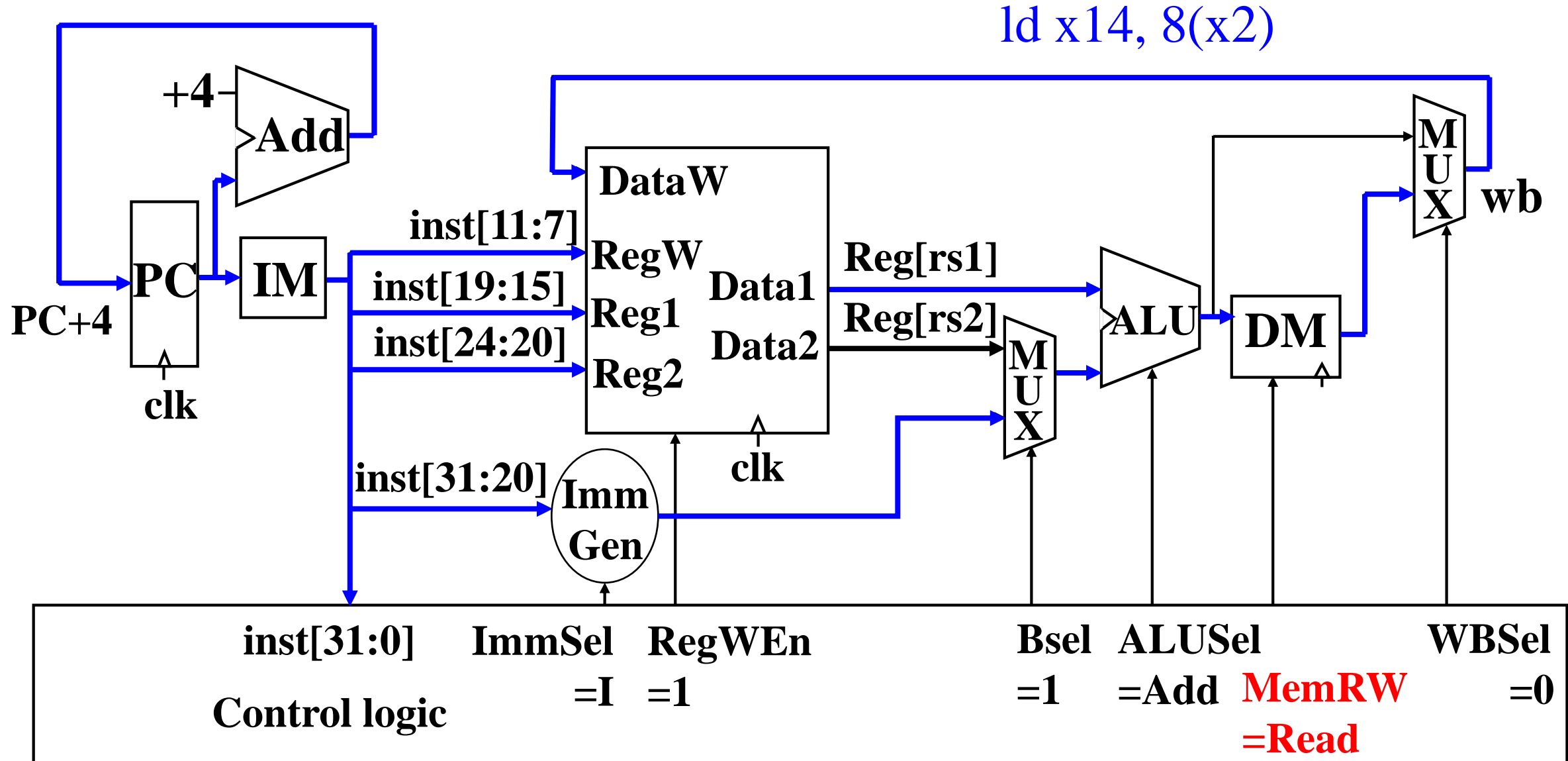
实现ld指令 (load dword)



ld x14, 8(x2)

- 寄存器rs1中的值作为基地址，加上立即数值，得到目标访问地址
 - 与addi操作十分相似，但用于计算地址，而不是获得最终结果
- 从存储器读出的数据装入寄存器rd中

ld指令的数据通路



RISC-V 访存装载指令

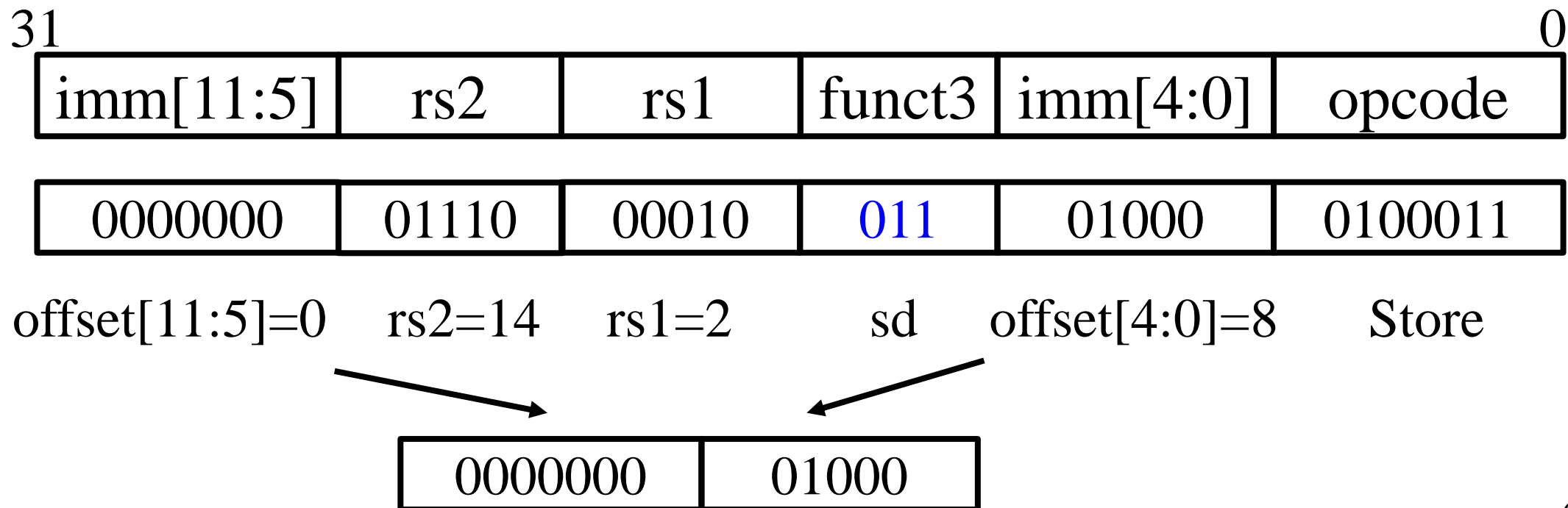
imm[11:0]	rs1	000	rd	0000011	lb
imm[11:0]	rs1	001	rd	0000011	lh
imm[11:0]	rs1	010	rd	0000011	lw
imm[11:0]	rs1	011	rd	0000011	ld
imm[11:0]	rs1	100	rd	0000011	lbu
imm[11:0]	rs1	101	rd	0000011	lhu
imm[11:0]	rs1	110	rd	0000011	lwu

- 为了支持8位、16位、32位等的访存装载指令
 - 增加额外的逻辑电路，用于从存储器取出的数据中提取出字节、半字、字数据等
 - 写回到寄存器前，进行符号扩展或零扩展

实现sd指令

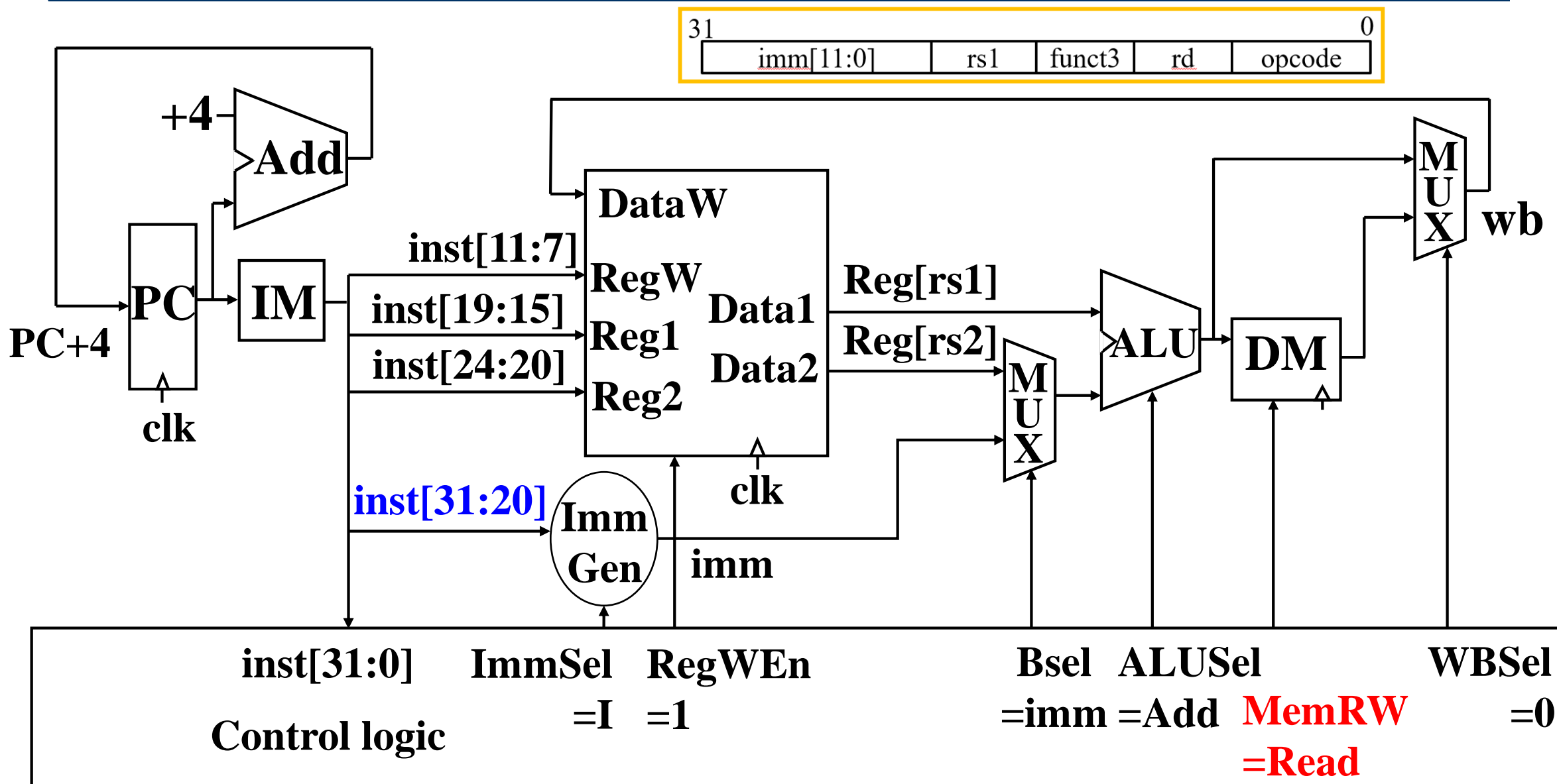
- 读取两个寄存器，rs1作为提供基地址的源寄存器，rs2作为提供待保存数据的源寄存器，以及立即数偏移量

sd x14, 8(x2)



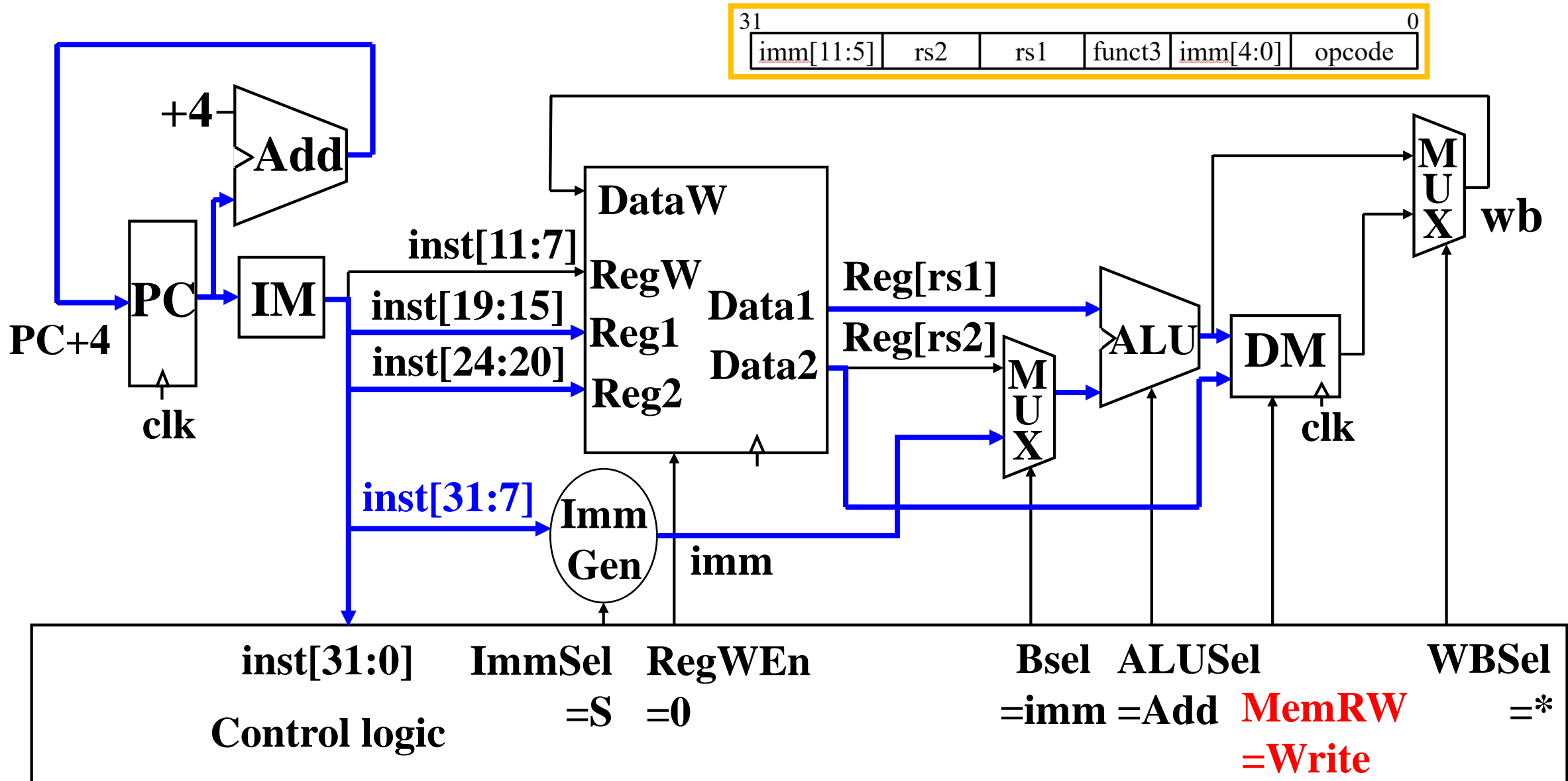
ld指令的数据通路

ld x14, 8(x2)

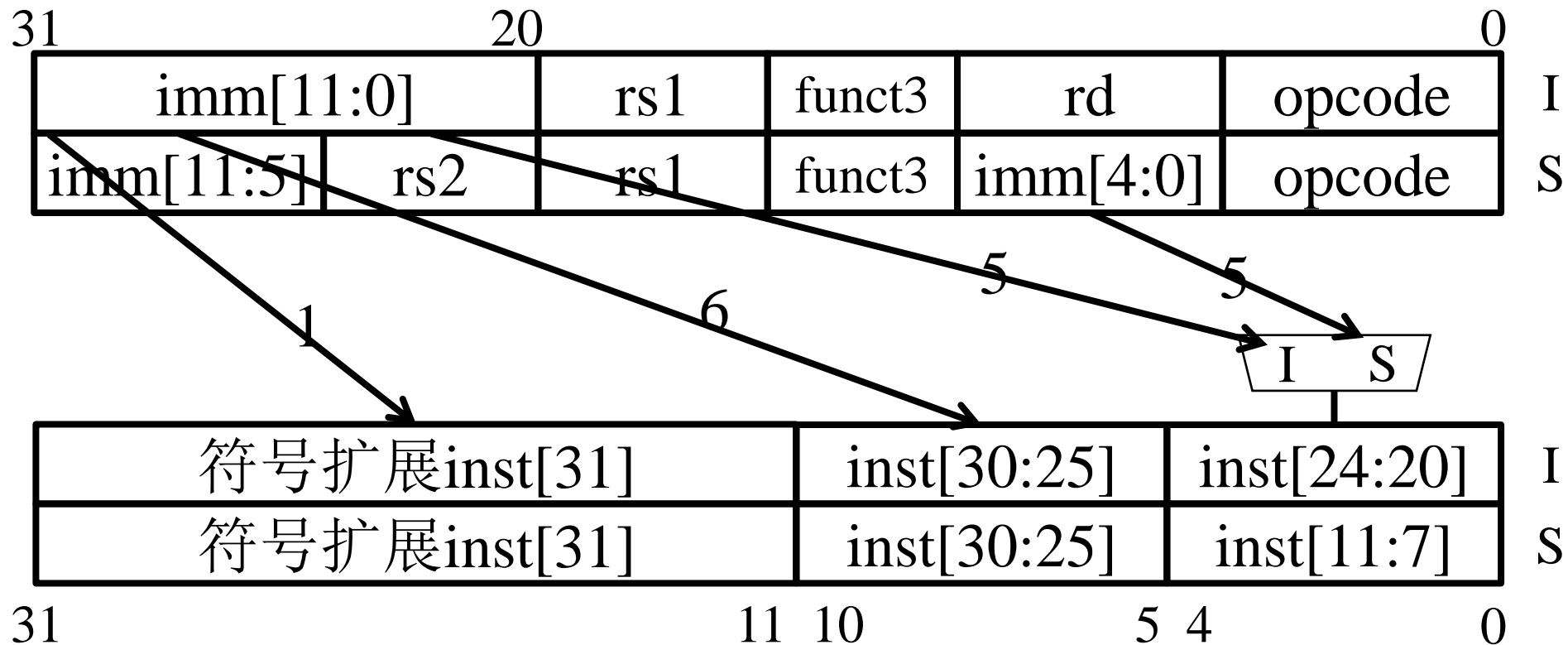


sd指令的数据通路

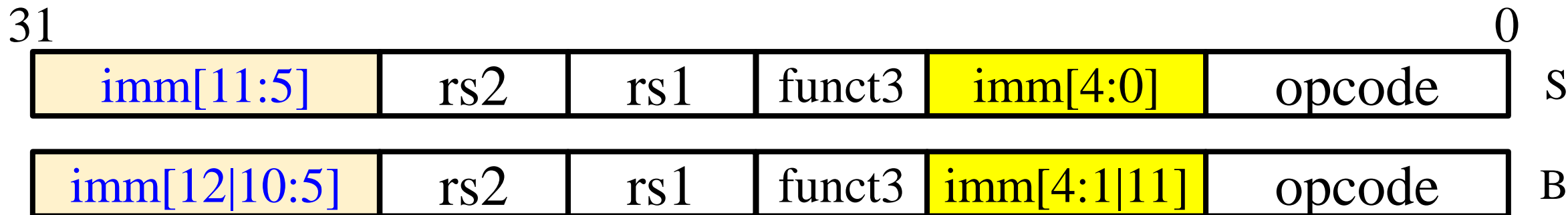
sd x14, 8(x2)



I型指令和S型指令中的立即数生成



- 立即数低5位由I型或S型对应的控制信号选择
- 立即数中的其他位连接到指令中的固定位置



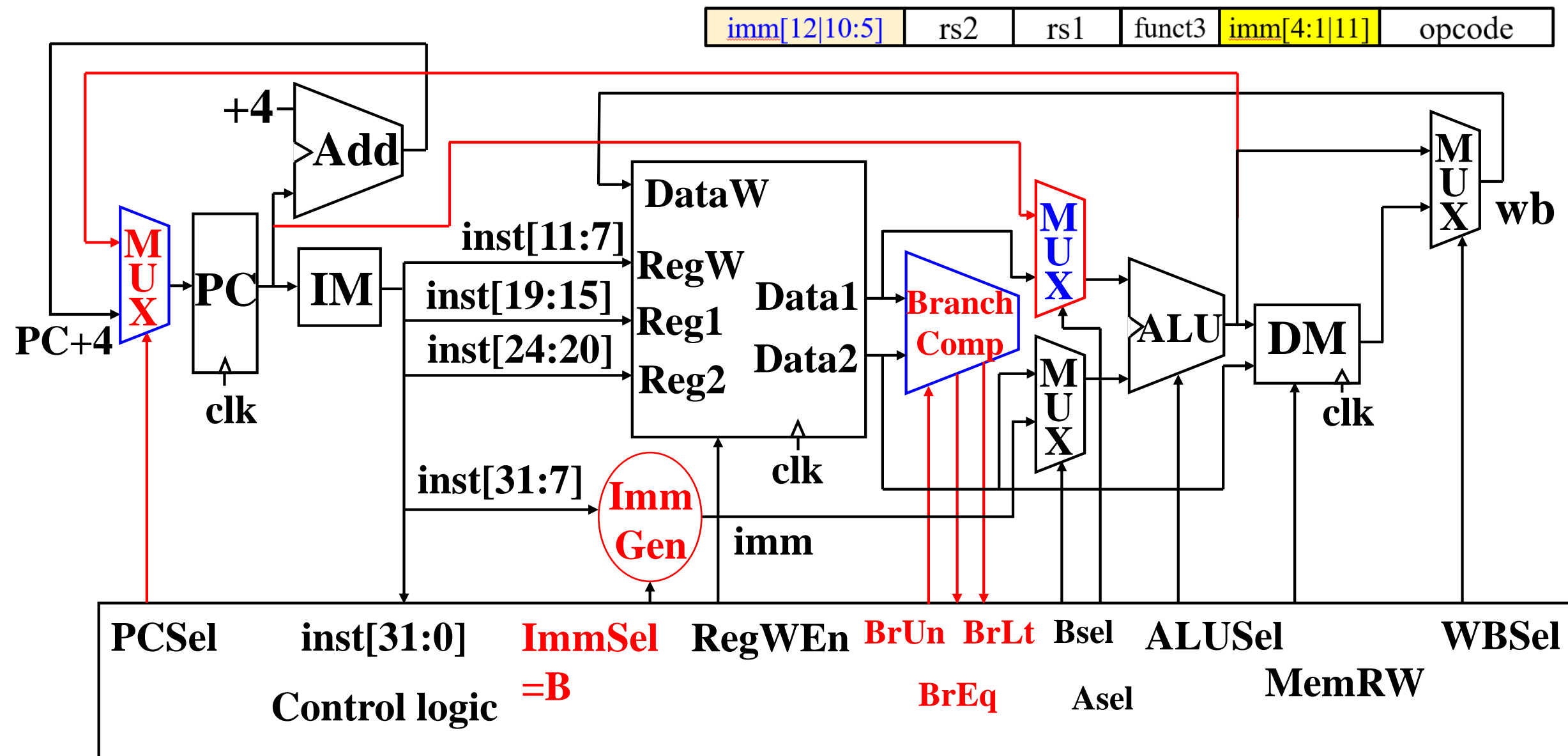
- B型指令格式与S型指令格式基本相同
- 但立即数字段以2字节为增量表示-4096到+4094的偏移量
- 12位立即数字段表示13位有符号字节地址的偏移量
 - 偏移量的最低位始终为零，因此无需存储

B型指令的数据通路

- 六个指令： beq、 bne、 blt、 bge、 bltu、 bgeu
- PC不同的状态变化：
 - $PC + 4$ (不发生分支转移)
 - $PC + \text{immediate}$ (发生分支转移)
- 需要比较rs1和rs2的数值关系，并计算 $PC + \text{立即数}$ 的结果
- 加一个ALU vs 加专门的比较硬件电路

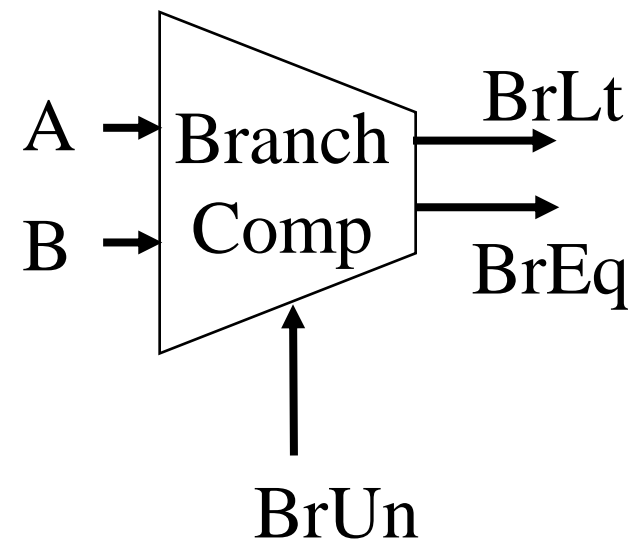
B型指令的数据通路

blt rs1,rs2, label



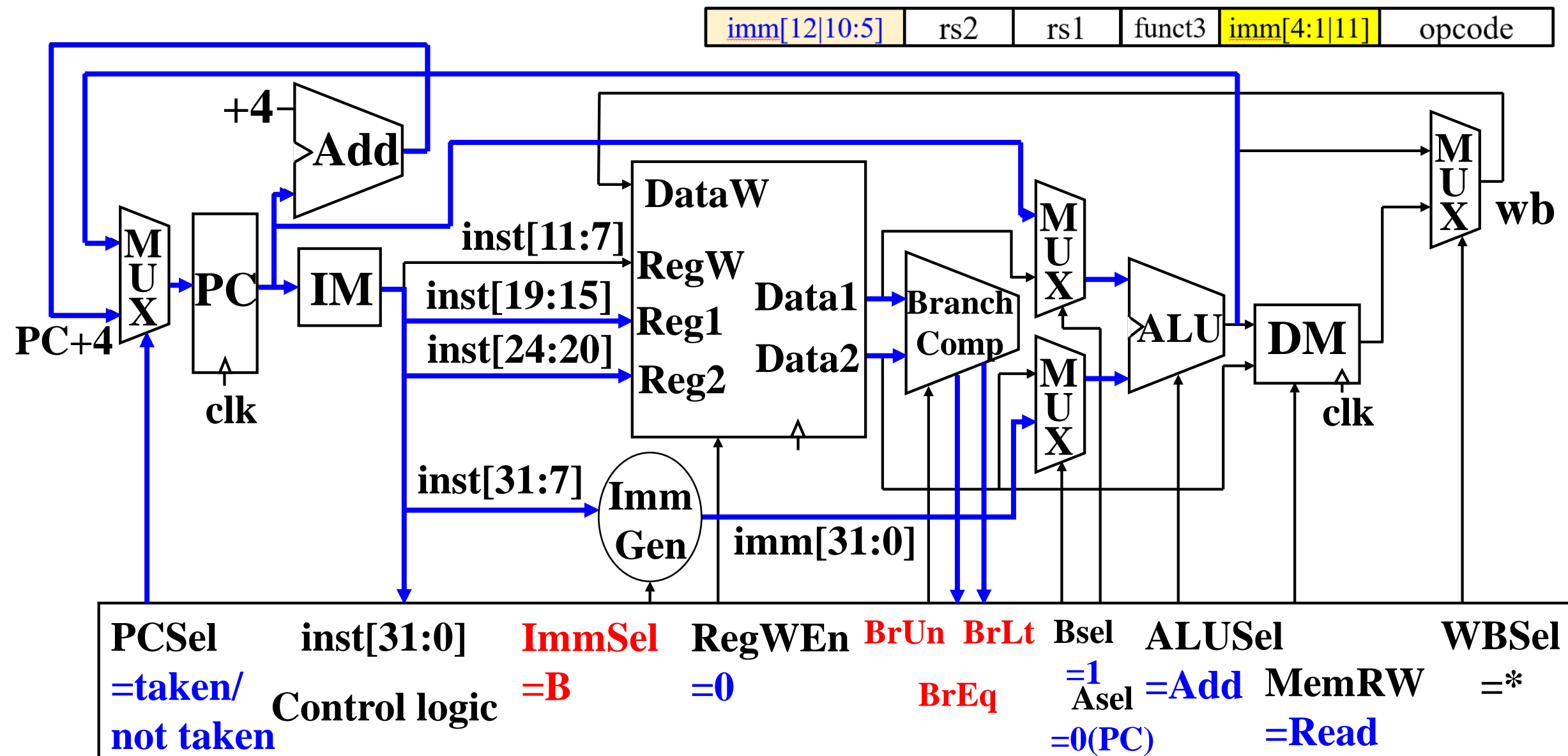
分支跳转比较器

- 当 $A = B$ 时，输出 $BrEq = 1$ ，否则为0
- 当 $A < B$ 时，输出 $BrLt = 1$ ，否则为0
- 输入 $BrUn = 1$ 时，进行无符号比较
输入 $BrUn = 0$ 时，进行有符号比较
- 对于 `bge`，可以根据 $BrLt$ 信号取反判断 $A \geq B$



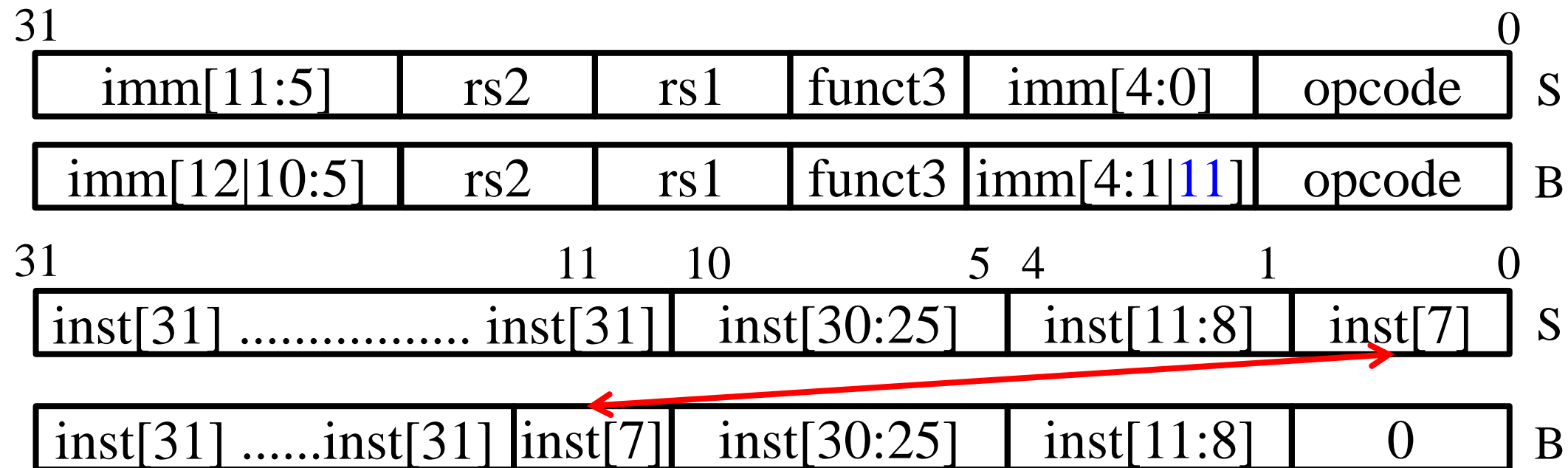
B型指令的数据通路

blt rs1,rs2, label



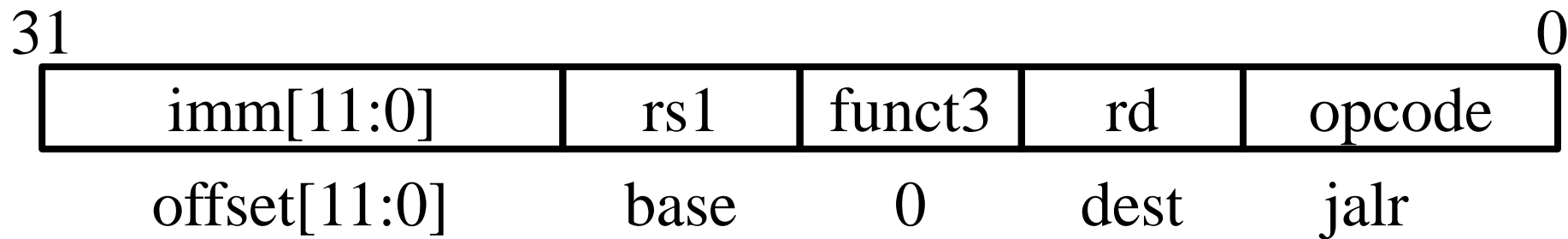
B型指令立即数生成

- 与S型指令立即数生成方式类似，**区别**在于S型立即数最低位在B型立即数中变为第12位
 - 只有一位数据在编码位置上有差异
 - 需要两个1位两路选择器



I型指令——jalr

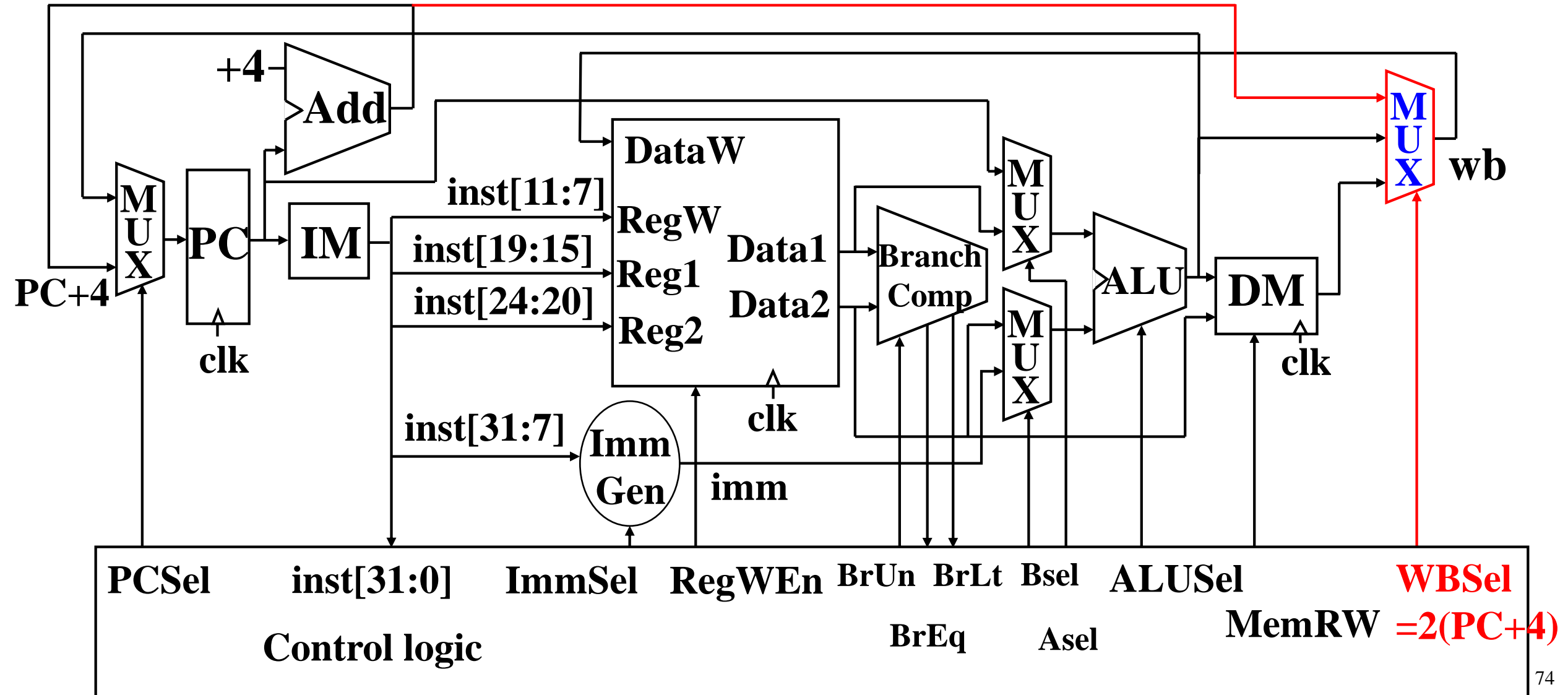
jalr x0, 0(x1)



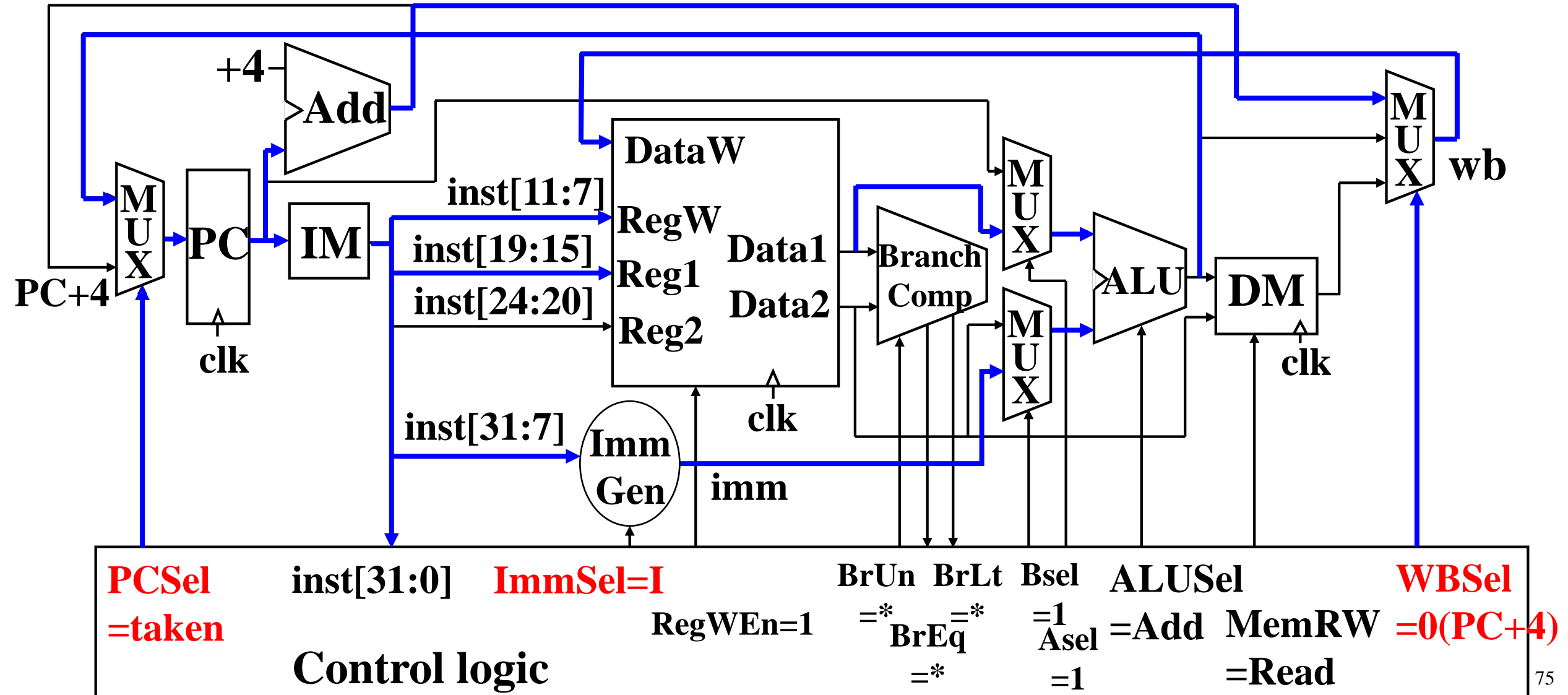
- jalr rd, offset(rs1) #(jump and link register)
- 两个状态变化
 - 将 $PC + 4$ 写入rd（返回地址）
 - 设置 **$PC = R[rs1] + offset$**
- 立即数生成与I型算术和装载指令一样

jalr指令的数据通路

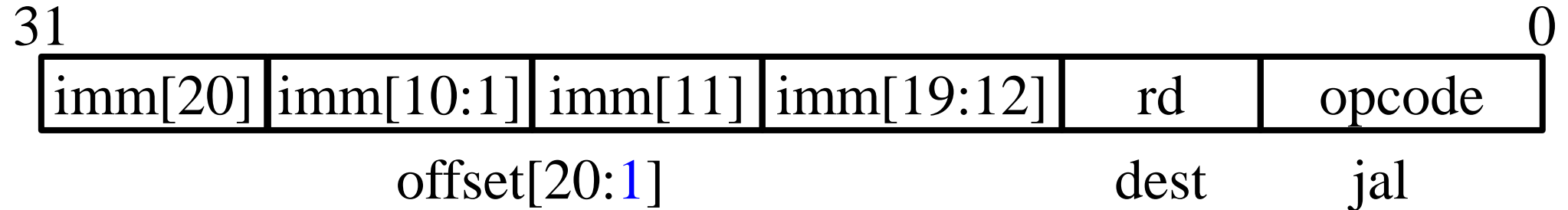
jalr rd, offset(rs1)



jalr指令的数据通路 jalr rd, offset(rs1)



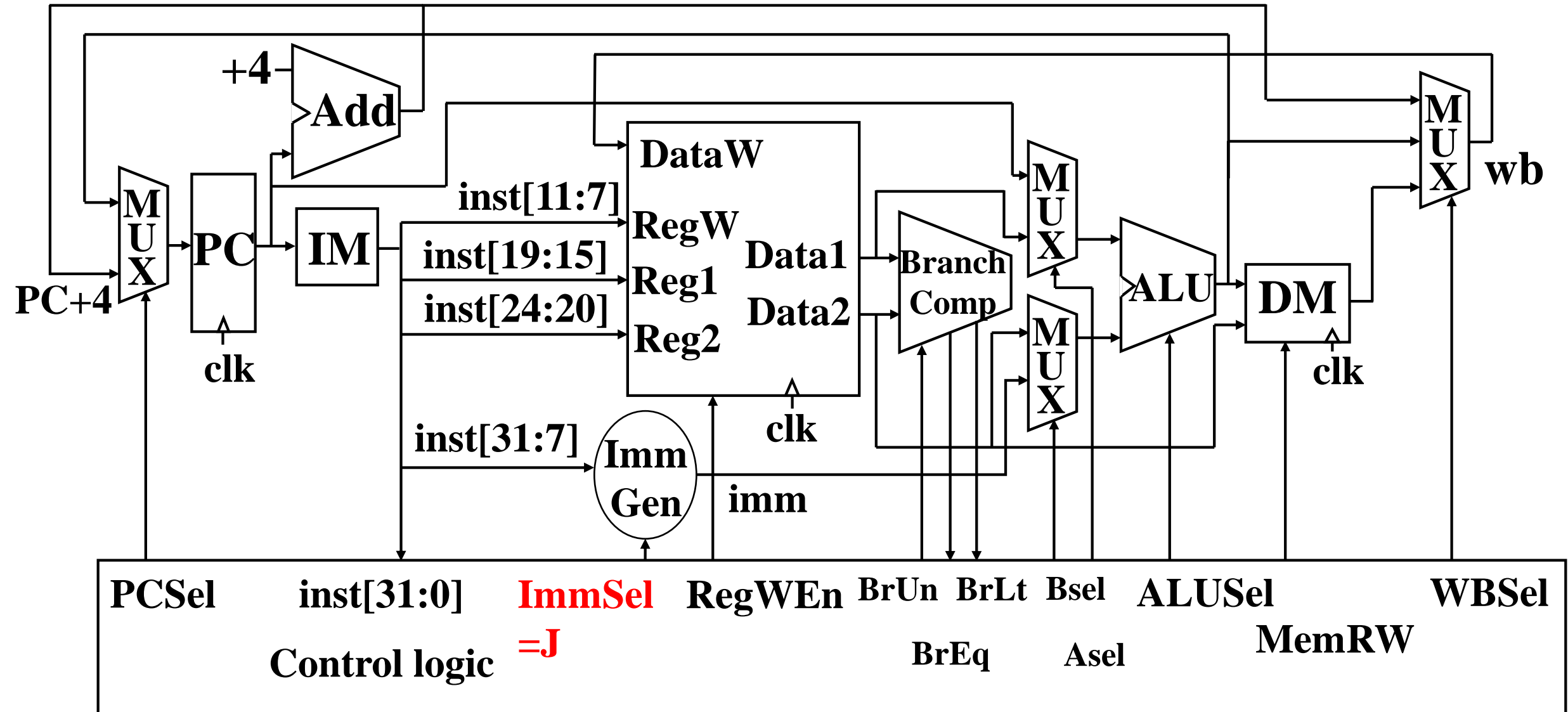
J型指令——jal



jal rd, label

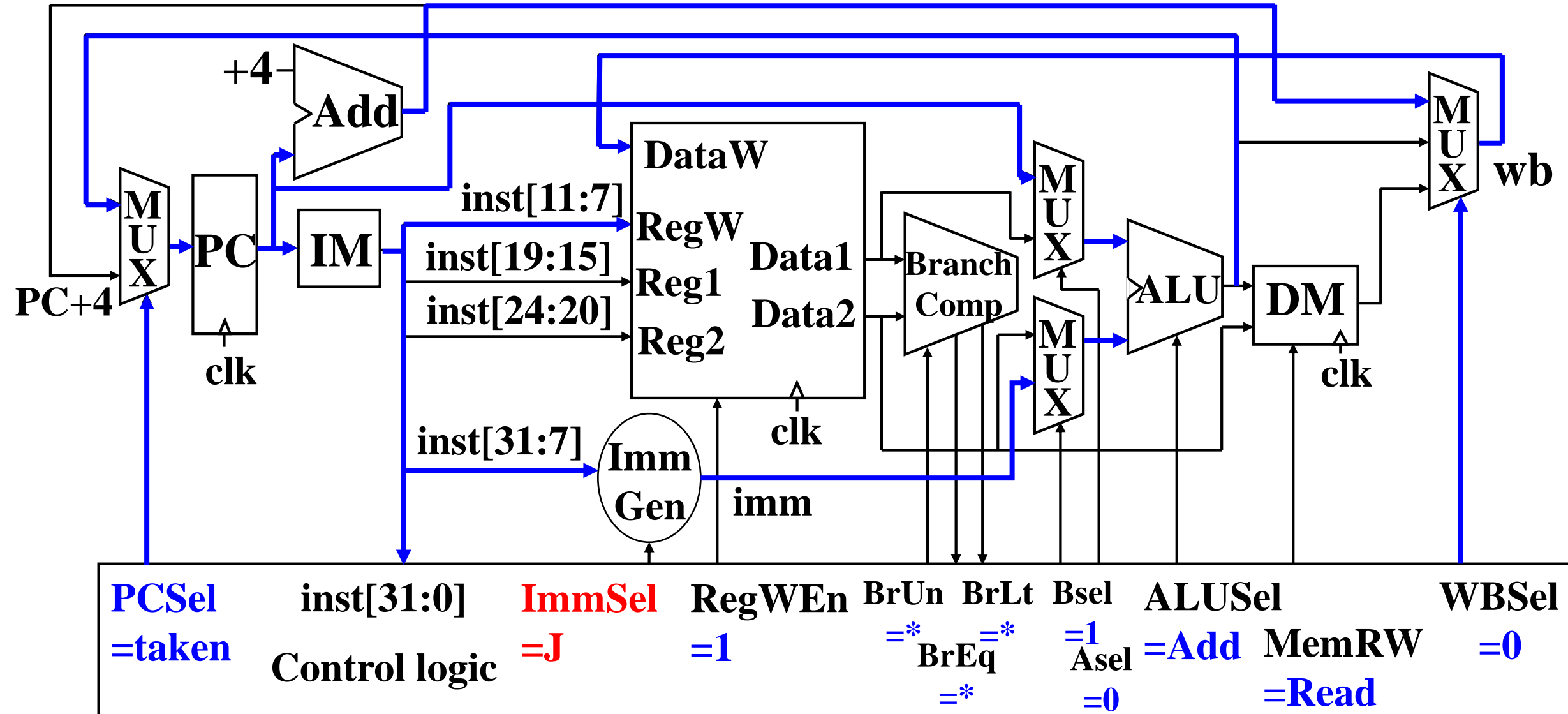
- jal将 $PC + 4$ 写入目的寄存器rd中
- 设置 $PC = PC + offset$
- 立即数字段20位，对应一个约 $\pm 2^{19}$ 的偏移量(以2字节为单位)

jal指令的数据通路



jal指令的数据通路

jal rd, label



U型指令



U-immediate[31:12]

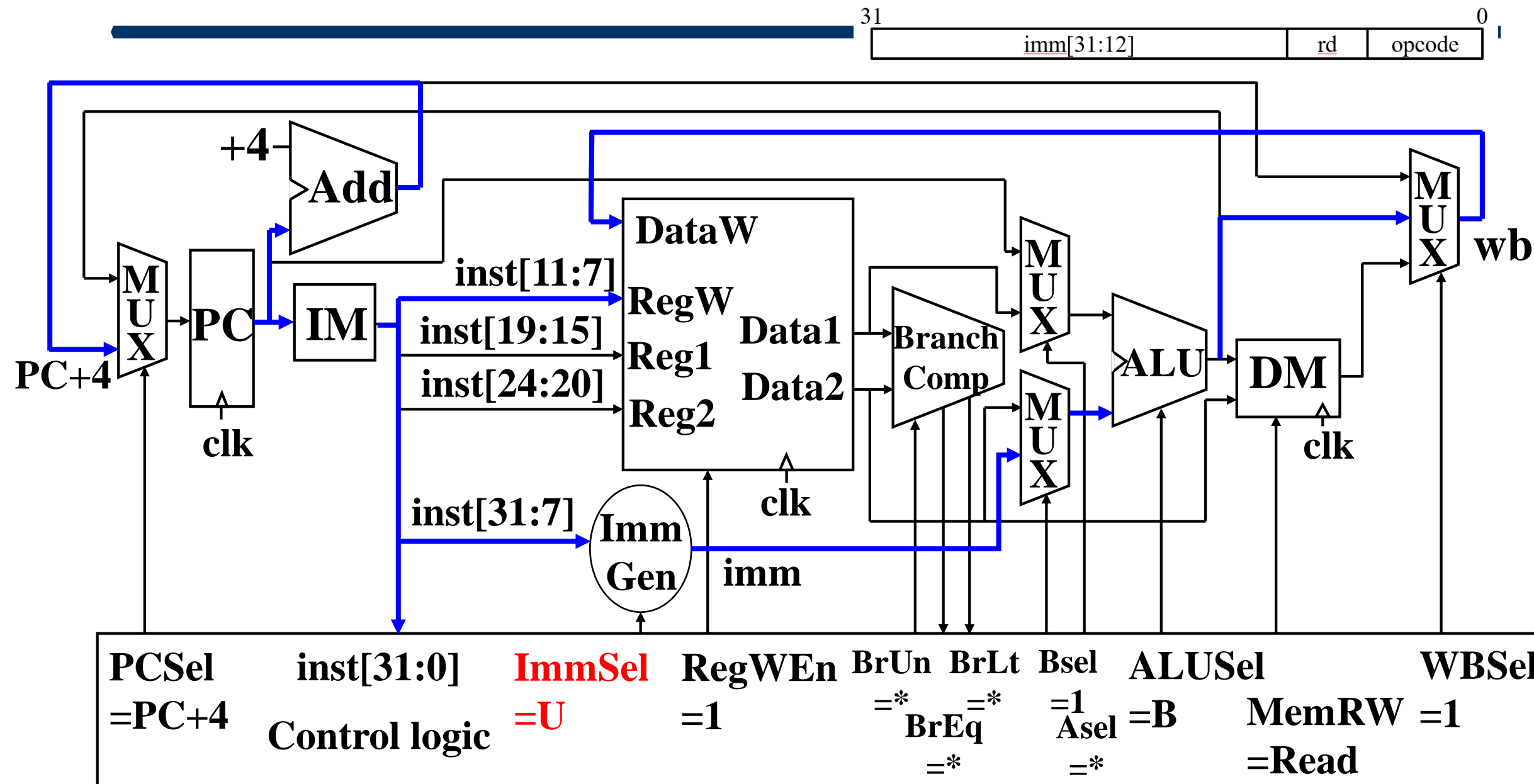
dest

lui

auipc

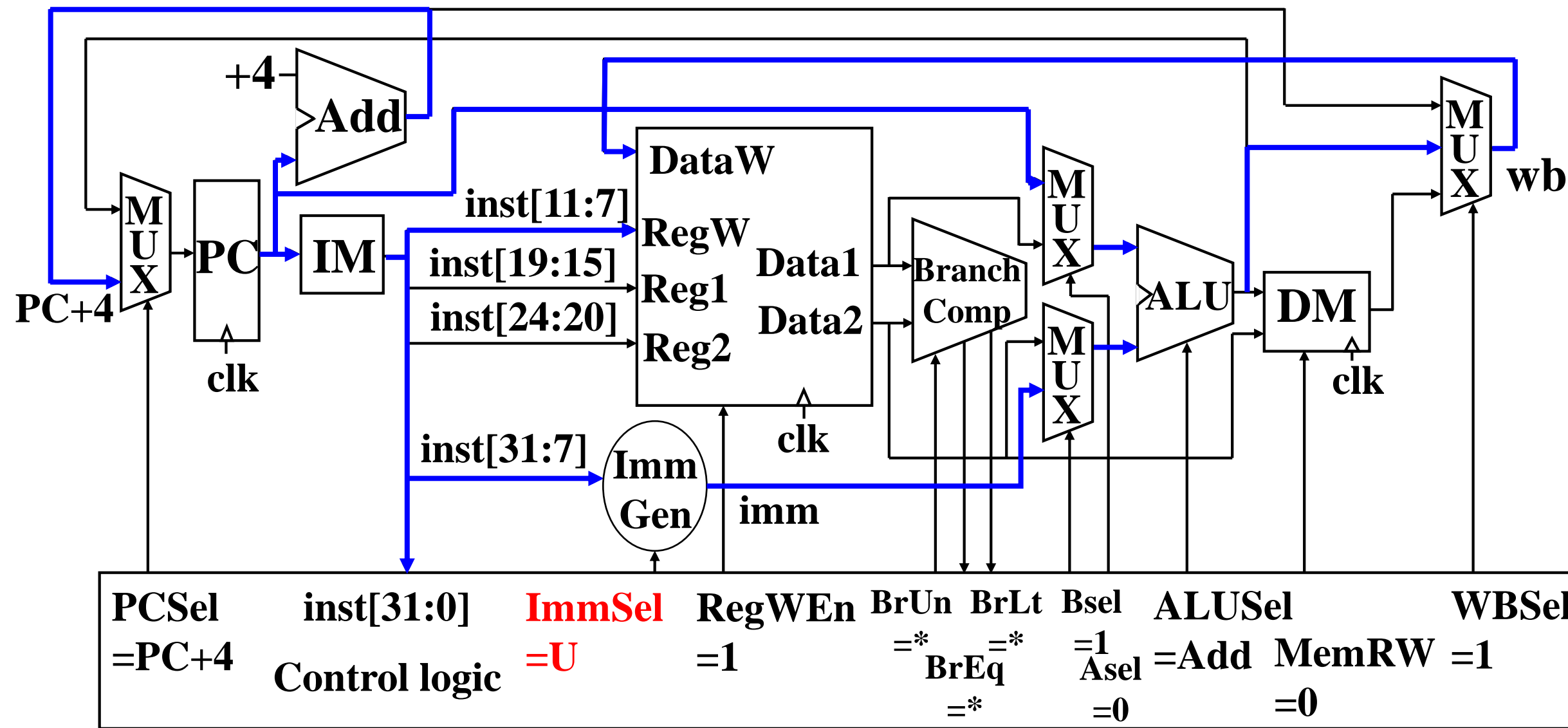
- U型指令进行长立即数操作
 - 高20位为长度为20位的立即数字段
 - 5位目的寄存器字段
- 用于两个指令
 - lui——将长立即数写入目的寄存器
 - auipc——将PC与长立即数相加结果写入目的寄存器

lui指令的数据通路

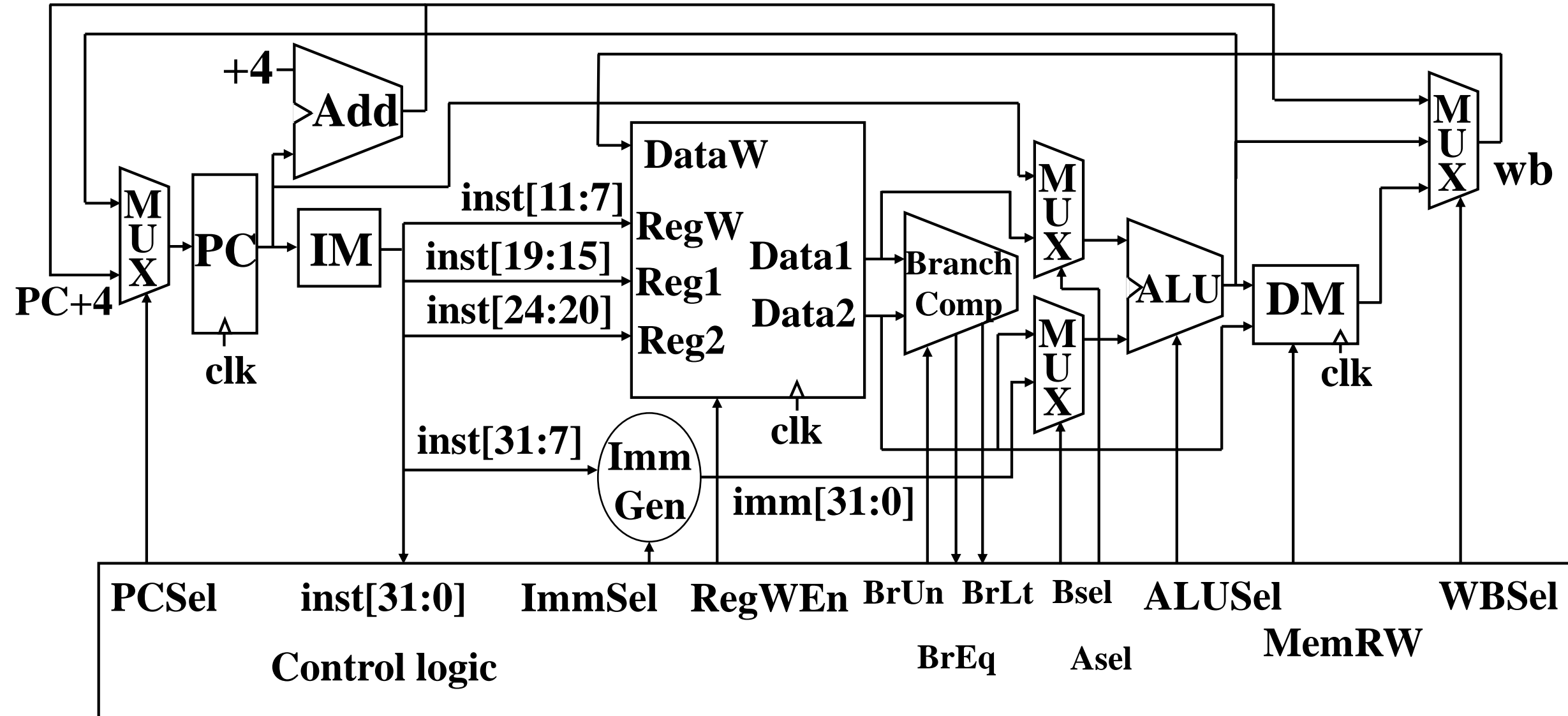


auipc指令的数据通路

将PC与长立即数相加结果写入目的寄存器



RISC-V 的数据通路



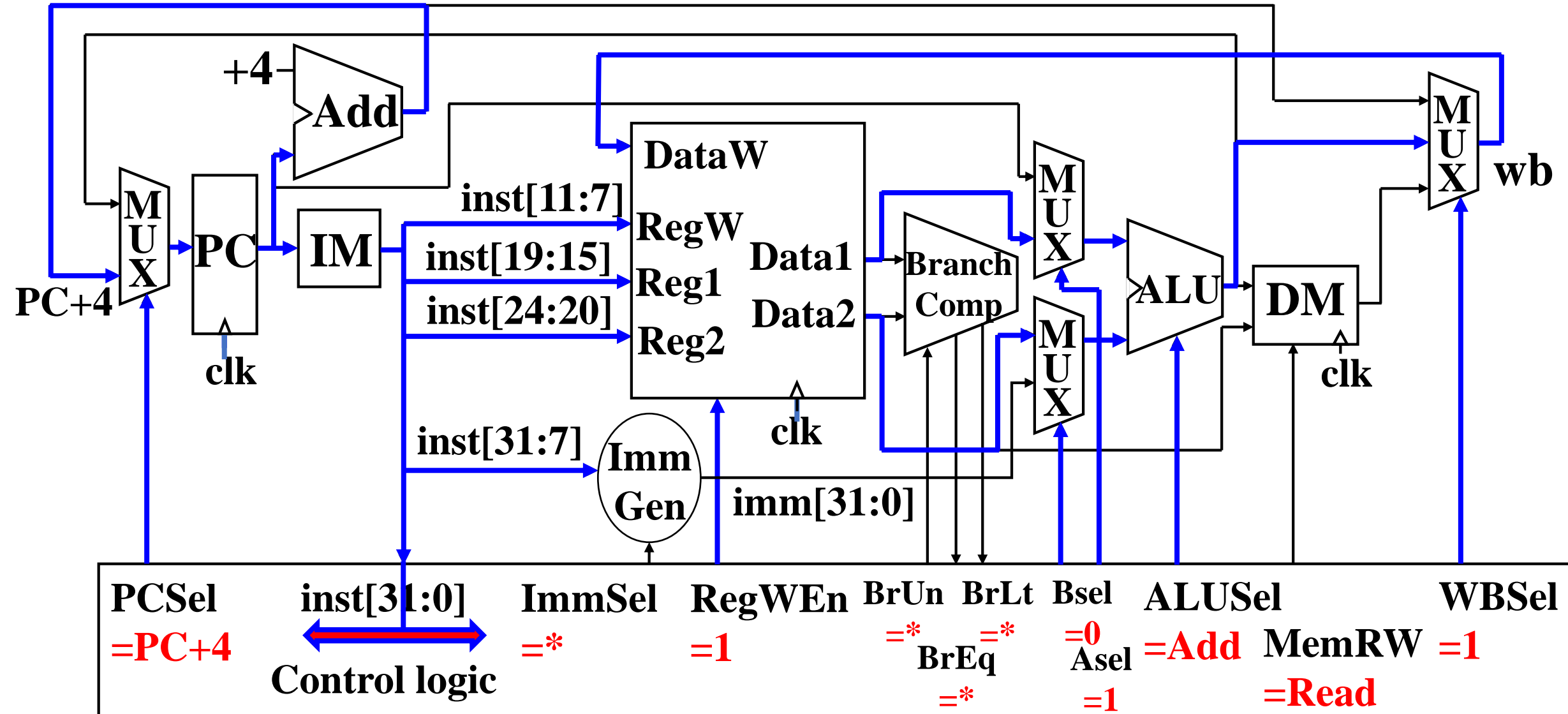
第4章 处理器设计

- 处理器设计的需求分析
- RISC-V数据通路的组件选择
- RISC-V部分指令的数据通路设计
- **RISC-V控制器**

处理器的设计步骤

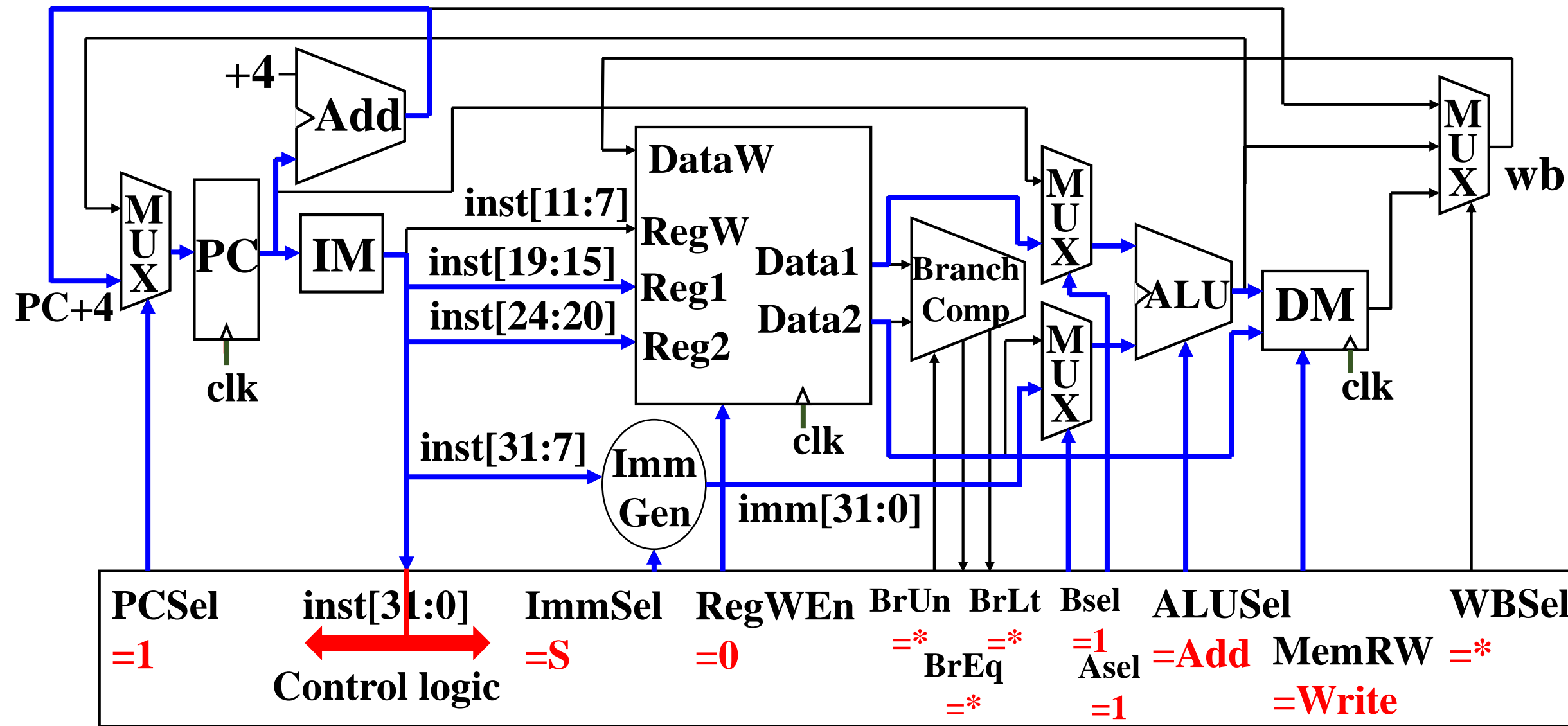
- ① 分析指令系统，得出对数据通路的需求
- ② 为数据通路选择合适的组件
- ③ 根据指令需求连接组件建立数据通路
- ④ 分析每条指令的实现，以确定控制信号
- ⑤ 集成控制信号，形成完整的控制逻辑

add指令的执行



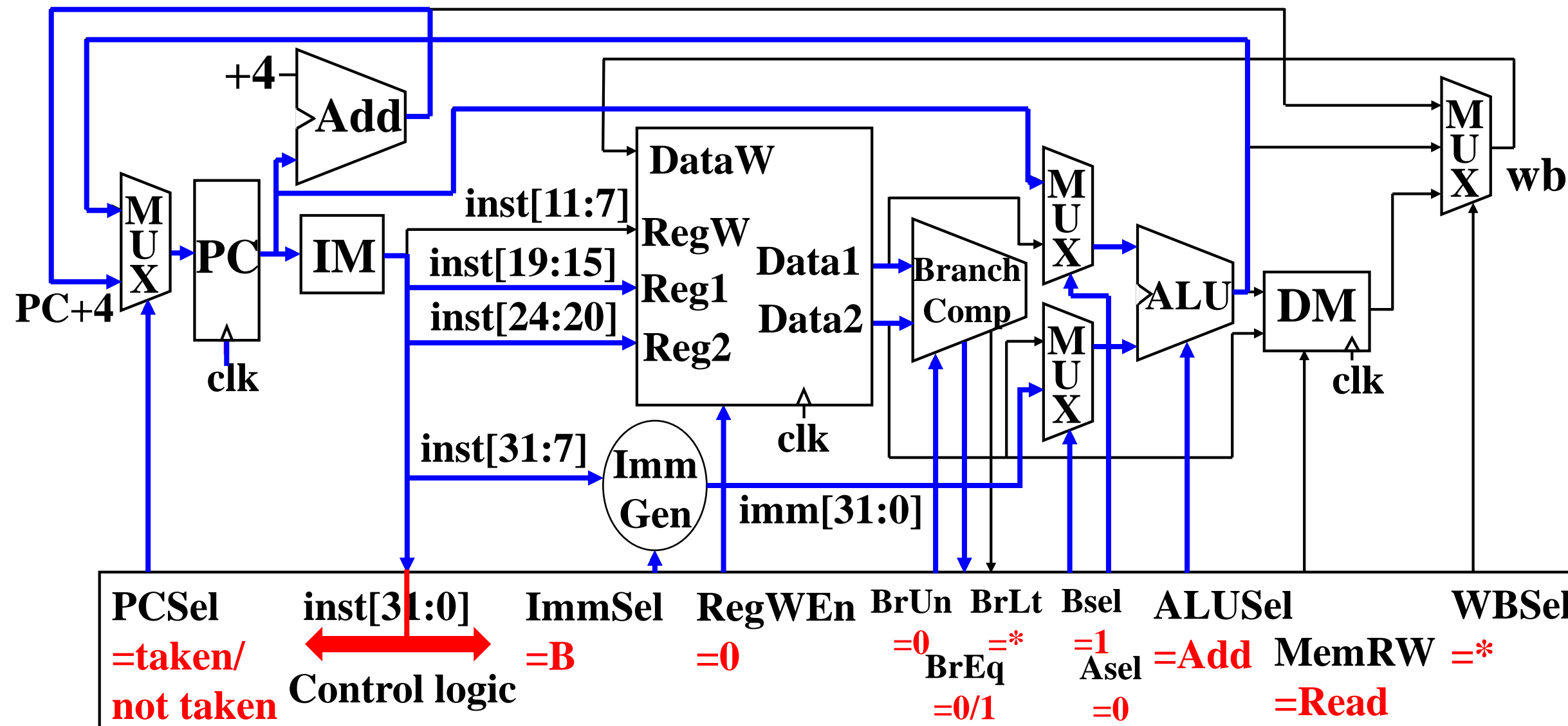
sd指令的执行

sd x14, 8(x2)



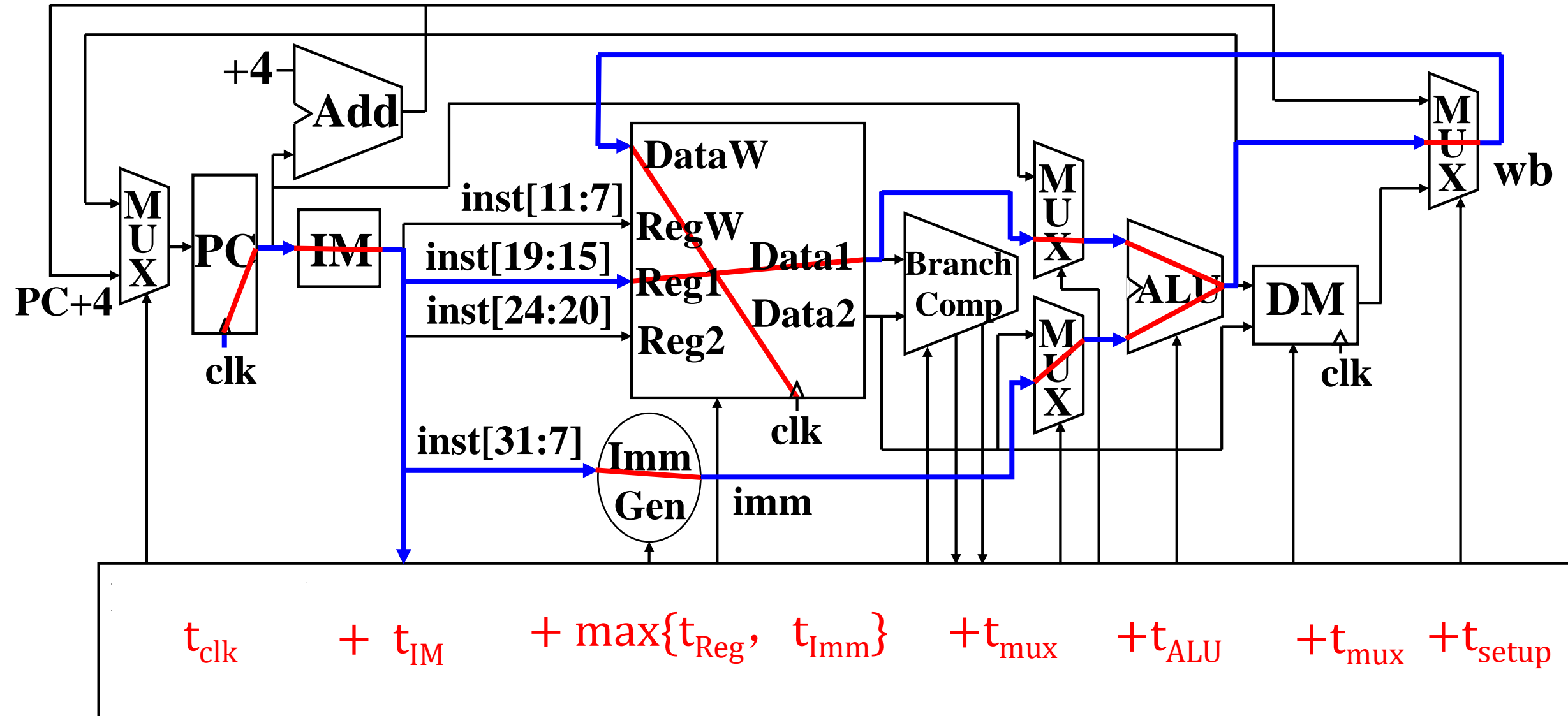
beq指令的执行

beq rs1,rs2, label

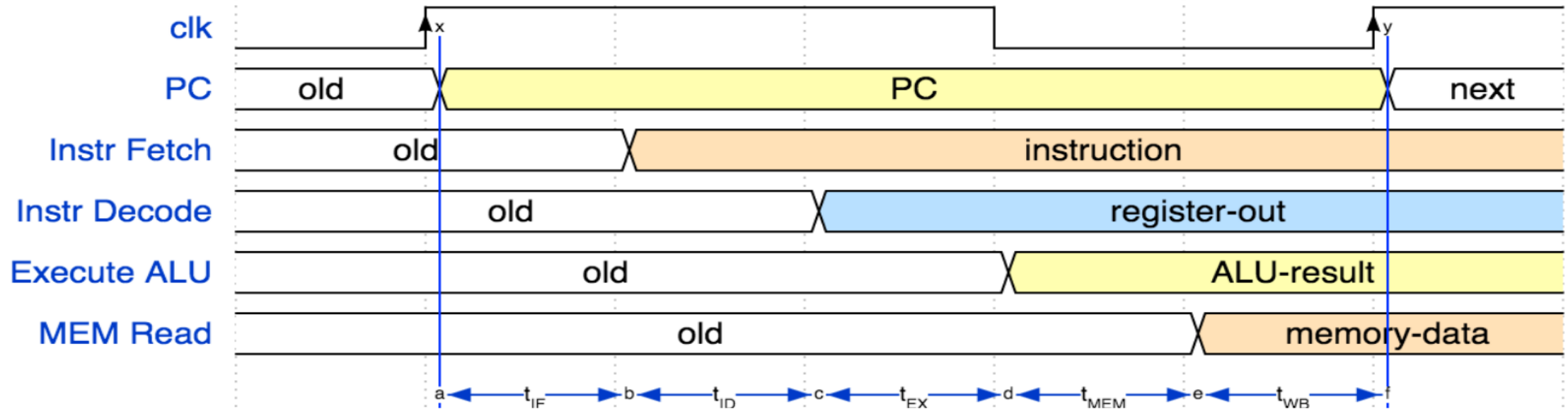


关键路径

addi x15, x1, -50



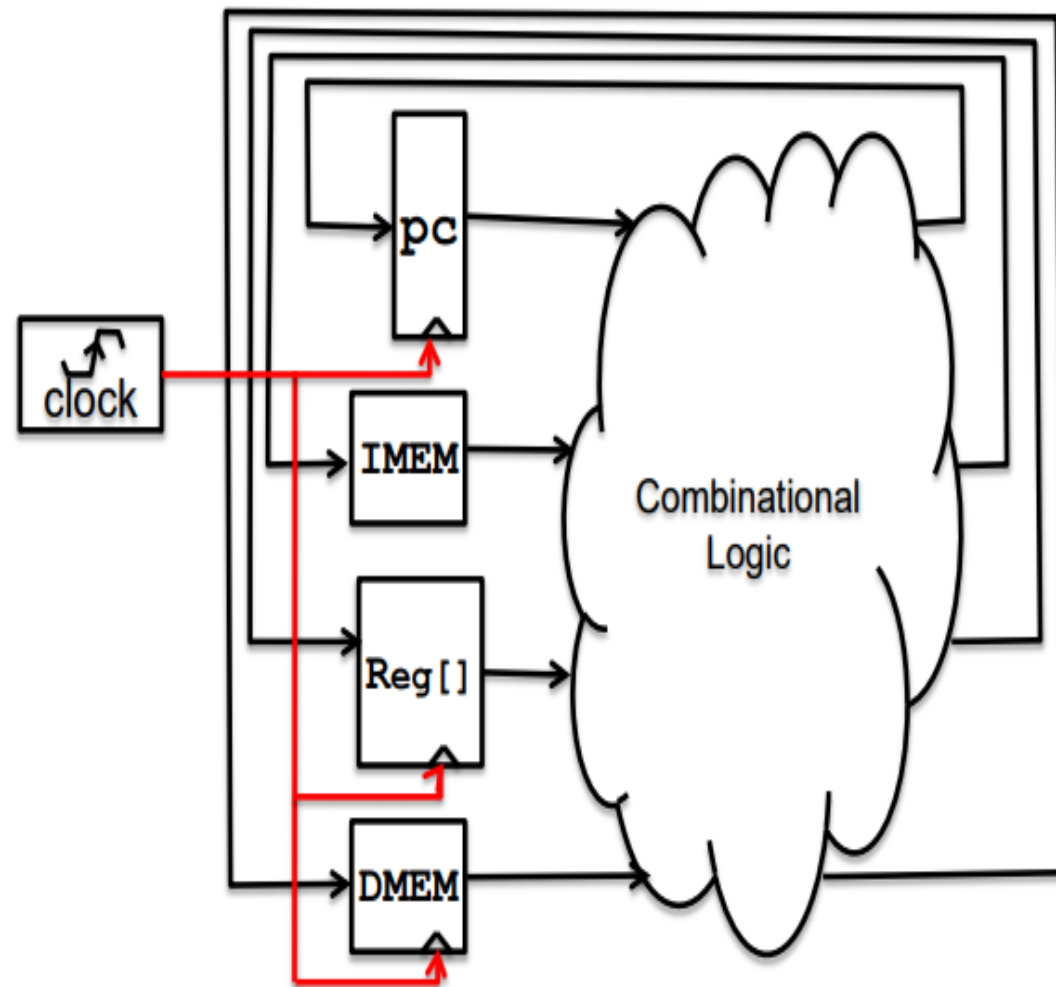
指令时延



IF	ID	EX	MEM	WB	Total
I-MEM	Reg Read	ALU	D-MEM	Reg W	
200ps	100ps	200ps	200ps	100ps	800ps

单周期CPU设计模型

- 每个时钟节拍执行一条指令
- 当前状态值通过一系列组合逻辑得到当前指令机器码，以及对应的操作码类型，操作数，结果。
- 在下一时钟上升沿到来时，所有组合逻辑单元输出的状态值保持或者更新对应寄存器，同时开始下一个时钟周期的执行操作。



指令时延

Instr	IF=200ps	ID=100ps	ALU=200ps	MEM=200ps	WB=100ps	Total
add	√	√	√		√	600ps
beq	√	√	√			500ps
jal	√	√	√		√	600ps
load	√	√	√	√	√	800ps
store	√	√	√	√		700ps

- 最大时钟频率 $f_{\max, \text{ALU}} = 1/(800\text{ps}) = 1.25\text{GHz}$
- 有些段多数时间处于空闲状态
- 单个段最大时钟频率 $f_{\max, \text{ALU}} = 1/(200\text{ps}) = 5\text{GHz}$

分阶段的数据通路——概述

- 问题：单块的逻辑结构完成对所有指令从取指到执行的所有操作，这种设计笨重，效率低下
- 分阶段的数据通路设计：将原来执行一条指令的过程，拆分为几个阶段，然后将这几个阶段对应的电路结构前后串联起来构成完整的数据通路。
 - 电路规模更小，更便于设计实现
 - 便于修改优化一个阶段而不影响其他阶段

RISC-V 编码

- 从前述的RISC-V 指令集编码可以看出，实际上用来区分某条指令种类的编码只有inst[30]、inst[14:12]、inst[6:2]

	int[31:25]	int[24:20]	int[19:15]	int[14:12]	int[11:7]	inst[6:0]
R型	func7	rs2	rs1	func3	rd	opcode
I型	<u>imm</u> [11 :0]	rs2	rs1	func3	rd	opcode
S型	<u>imm</u> [11 :5]	rs2	rs1	func3	<u>imm</u> [4:0]	opcode
B型	<u>imm</u> [12 ,10:5]	rs2	rs1	func3	<u>imm</u> [4:1, 11]	opcode
J型	<u>imm</u> [20 ,10:5]	<u>imm</u> [4:1, 11]	<u>imm</u> [19:15]	<u>imm</u> [14:12]	rd	opcode
U型	<u>imm</u> [31 :25]	<u>imm</u> [24:20]	<u>imm</u> [19:15]	<u>imm</u> [14:12]	rd	opcode

RISC-V 编码

- inst[30]用于区分移位指令是“逻辑右移”和“算术右移”，R型指令的ADD和SUB。

0000000	rs2	rs1	000	<u>rd</u>	0110011	add
0100000	rs2	rs1	000	<u>rd</u>	0110011	sub
0000000	rs2	rs1	001	<u>rd</u>	0110011	<u>sll</u>
0000000	rs2	rs1	010	<u>rd</u>	0110011	<u>slt</u>
0000000	rs2	rs1	011	<u>rd</u>	0110011	<u>sltu</u>
0000000	rs2	rs1	100	<u>rd</u>	0110011	<u>xor</u>
0000000	rs2	rs1	101	<u>rd</u>	0110011	<u>srl</u>
0100000	rs2	rs1	101	<u>rd</u>	0110011	<u>sra</u>
000000	<u>shamt</u>	rs1	101	<u>rd</u>	0010011	<u>srli</u>
010000	<u>shamt</u>	rs1	101	<u>rd</u>	0010011	<u>srai</u>

RISC-V 编码

- inst[14:12]是部分指令32位机器码的funct3字段。它主要用于区分一个大类指令下不同的小功能。例如，B型指令下的各种不同的比较方式。 inst[14:12] inst[6:2]

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

RISC-V 编码

- inst[6:2]是指令的操作码(Opcod)字段，主要用于区分不同类型的指令。

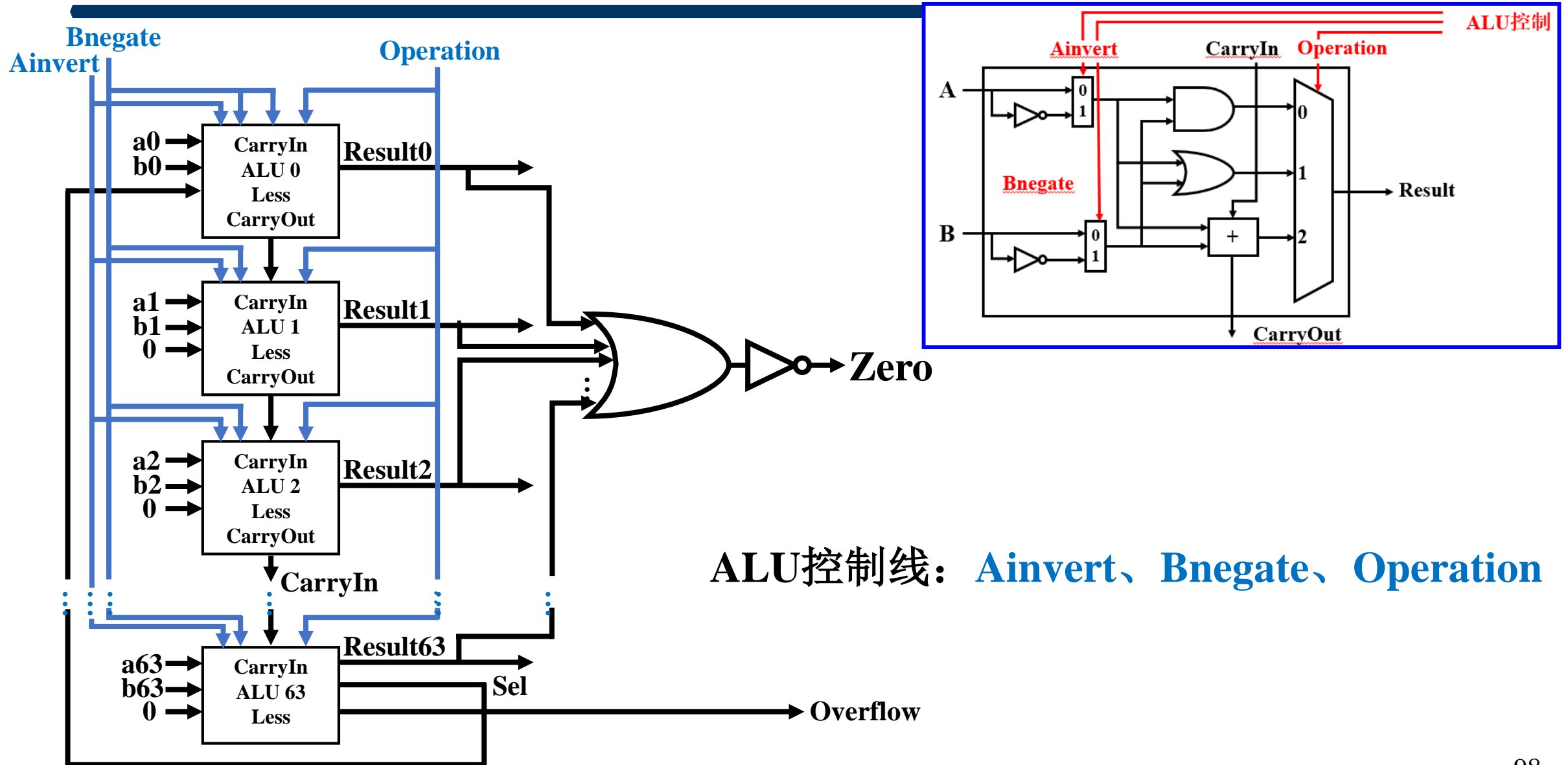
R 型	funct7	rs2	rs1	funct3	rd	0110011
I 型(运算类)	imm[11:0]		rs1	funct3	rd	0010011
I 型(lw)	imm[11:0]		rs1	funct3	rd	0000011
I 型(jalr)	imm[11:0]		rs1	funct3	rd	1100111
S 型	Imm[11:5]	rs2	rs1	funct3	imm[4:0]	0100011
B 型	Imm[12,10:5]	rs2	rs1	funct3	imm[4:1,11]	1100011
J 型(jal)	Imm[20,10:1,11,19:12]				rd	1101111
U 型	Imm[31:12]				rd	0110111

组合逻辑控制——例子

Inst[14:12]				Inst[6:2]		
↓				↓		
imm[12 10:5]	rs2	rs1	000	imm[4:1 11]	1100011	BEQ
imm[12 10:5]	rs2	rs1	001	imm[4:1 11]	1100011	BNE
imm[12 10:5]	rs2	rs1	100	imm[4:1 11]	1100011	BLT
imm[12 10:5]	rs2	rs1	101	imm[4:1 11]	1100011	BGE
imm[12 10:5]	rs2	rs1	110	imm[4:1 11]	1100011	BLTU
imm[12 10:5]	rs2	rs1	111	imm[4:1 11]	1100011	BGEU

• $\text{BrUn} = \text{Inst}[14] \cdot \text{Inst}[13] \cdot \text{Branch}$

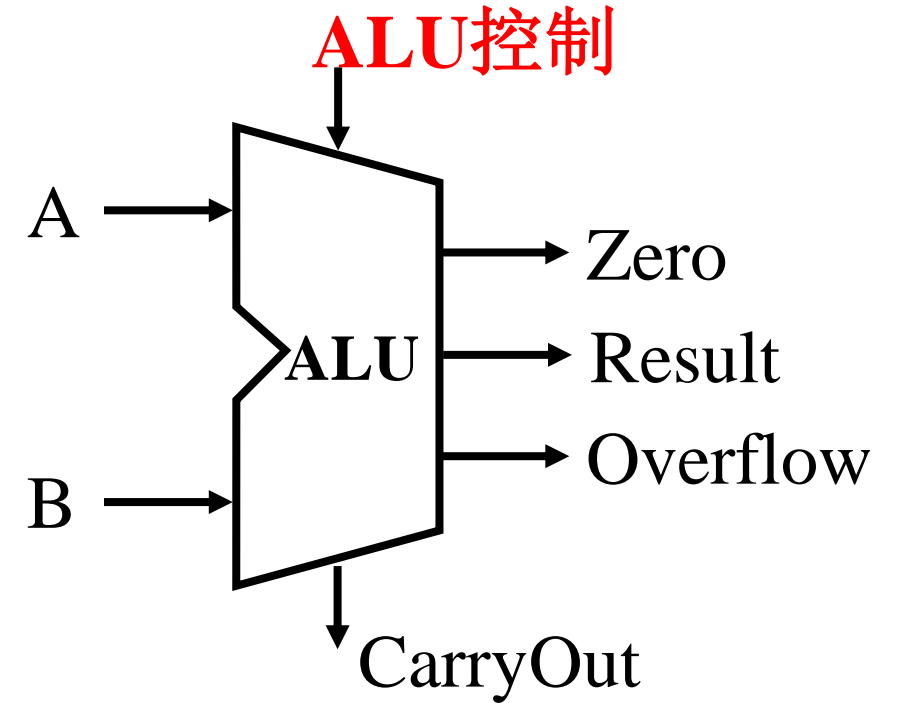
64bit ALU



ALU功能需求

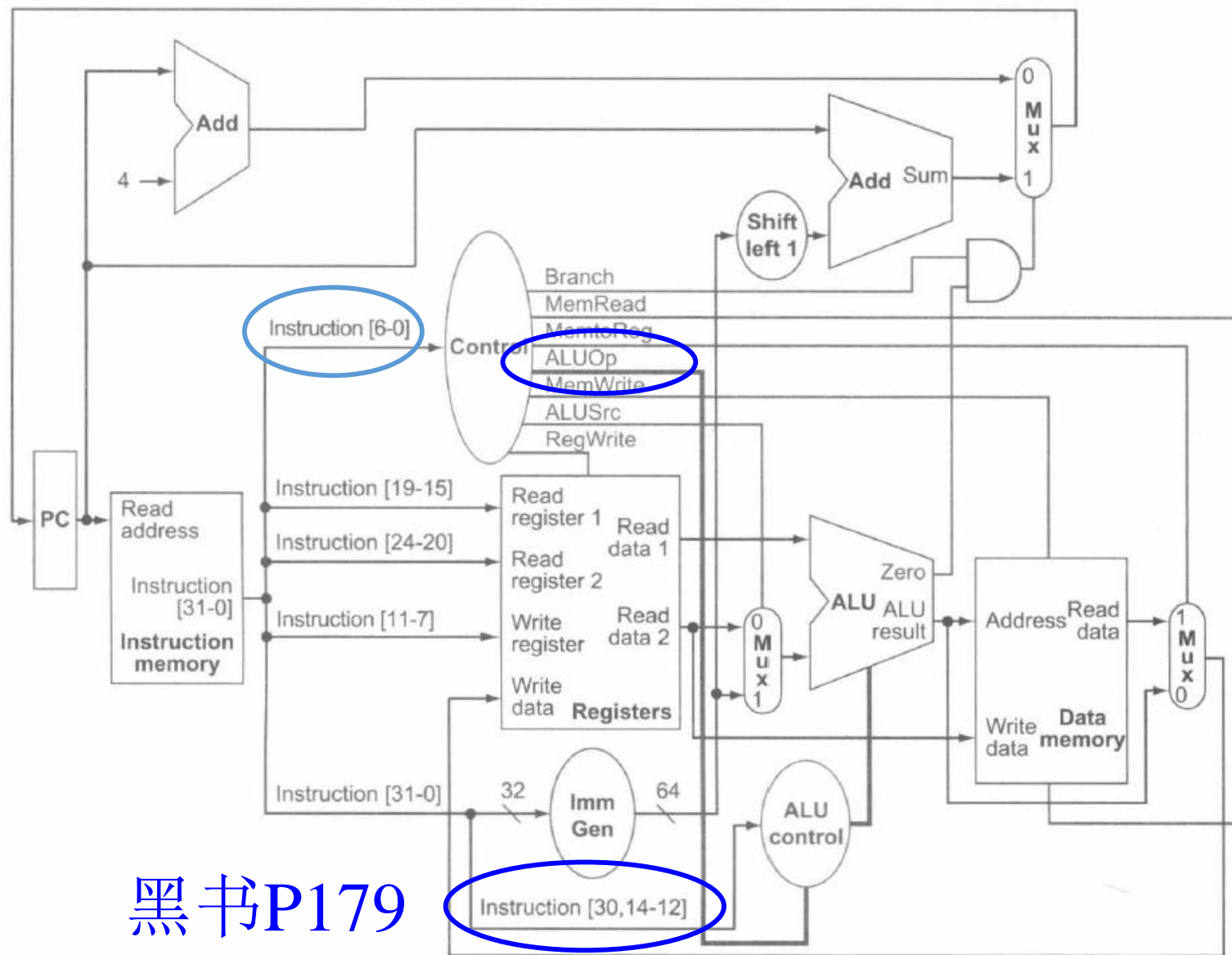
ALU控制线: **Ainvert**、**Bnegate**、**Operation**

ALU控制	操作
0000	与
0001	或
0010	加
0110	减
0111	小于则置位
1100	或非



带有控制单元的简单数据通路

- 控制单元的输入：指令的7位操作码字段
- 4位ALU的输入控制信号：指令func7、func3和ALUOp字段（inst[30]、inst[14:12]、ALUOp）



黑书P179

根据opcode和ALUOp控制位设置ALU控制

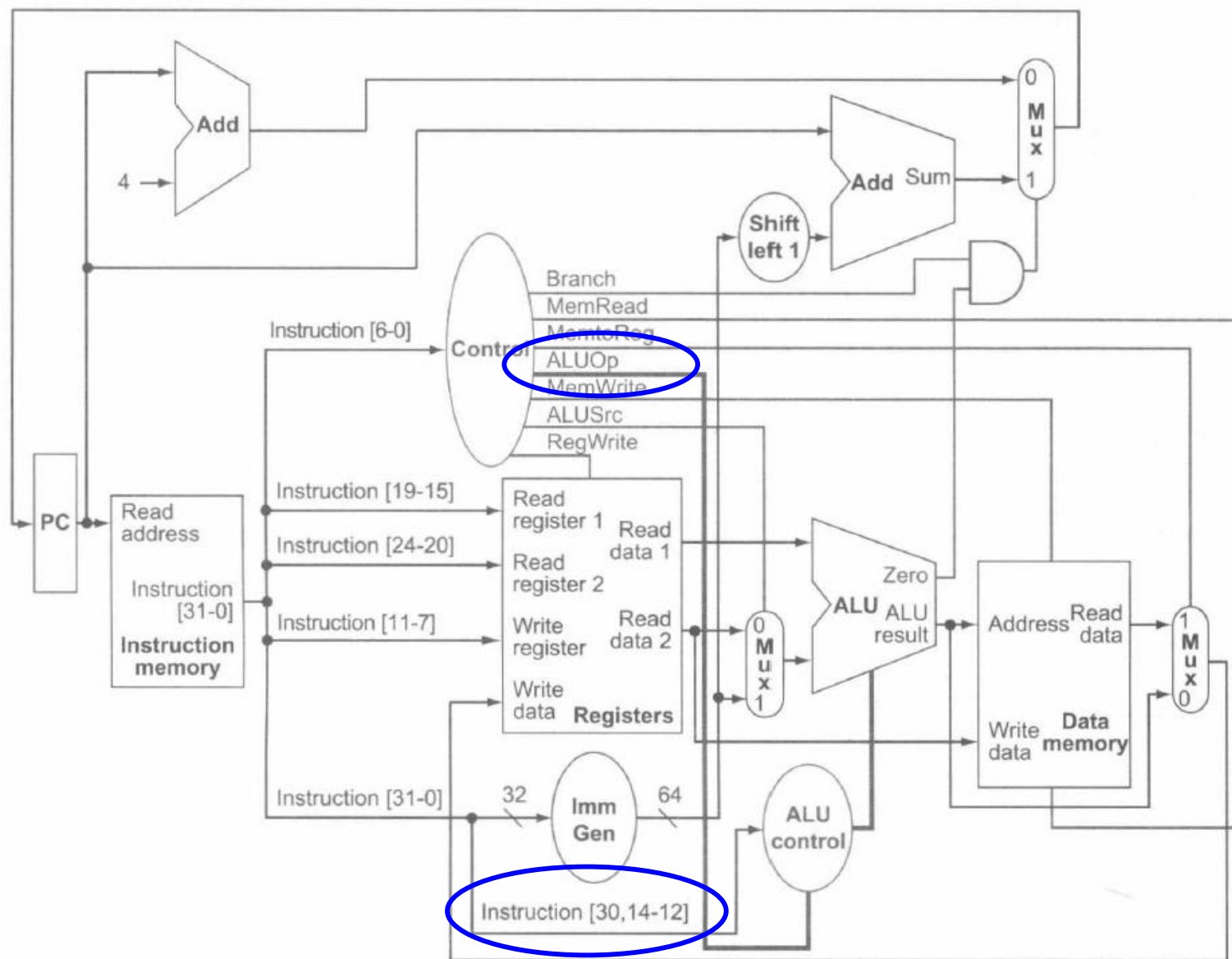
助记符	opcode	ALUOp	操作	func7	func3	ALU期望行为	ALU控制输入
ld	0000011	00	load Dword	*****	***	add	0010
sd	0100011	00	store Dword	*****	***	add	0010
beq	1100011	00	branch if equal	*****	***	add	0010
add	0110011	10	add	0000000	000	add	0010
sub	0110011	10	sub	0100000	000	subtract	0110
and	0110011	10	and	0000000	111	AND	0000
or	0110011	10	or	0000000	110	OR	0001

ALU控制输入信号真值表

ALUOp		funct7字段							funct3字段			ALU控制输入
ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[14]	I[13]	I[12]	
0	0	X	X	X	X	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	X	X	X	X	0110
1	X	0	0	0	0	0	0	0	0	0	0	0010
1	X	0	1	0	0	0	0	0	0	0	0	0110
1	X	0	0	0	0	0	0	0	1	1	1	0000
1	X	0	0	0	0	0	0	0	1	1	0	0001

黑书P176图4-13

带有控制单元的简单数据通路

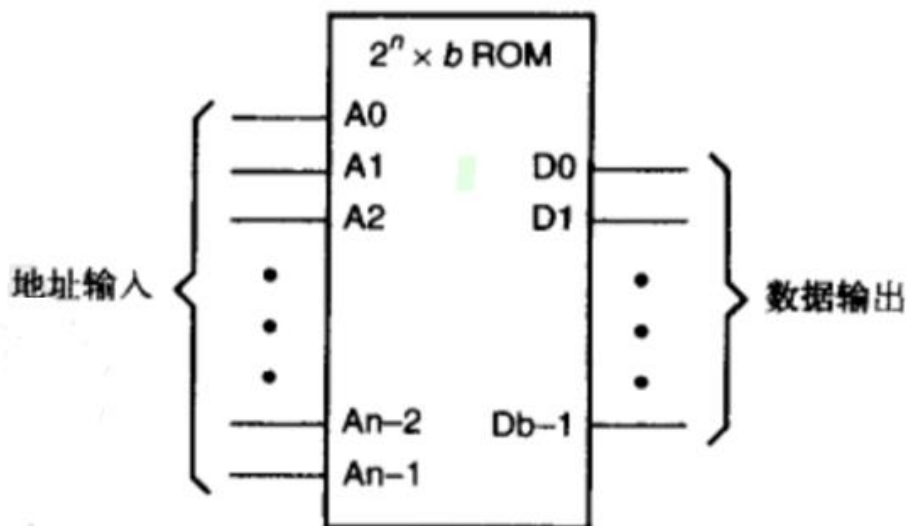


RISC V控制器实现

- ROM （ Read-Only Memory ）
 - 便于重新编程
 - 修正错误
 - 添加新的指令
 - 在人工设计控制逻辑时十分常用
- 组合逻辑
 - 现在很多的芯片设计者会使用逻辑综合工具，将控制器真值表实现为门级网表电路

ROM (Read-Only Memory)

- ROM是一种具有 n 个输入 b 个输出的组合逻辑电路。
- 输入被称为地址输入 (address input)，通常命名为 A_0, A_1, \dots, A_{n-1} 。
- 输出被称为数据输出 (data output)，通常命名为 D_0, D_1, \dots, D_{b-1} 。



一个 $2^n \times b$ ROM的基本结构

ROM和真值表

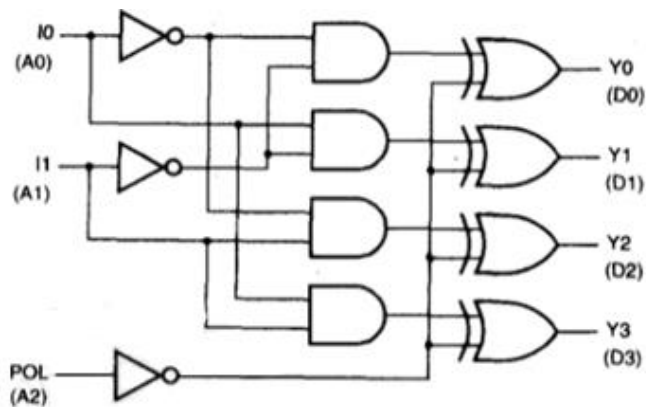
- ROM “存储” 了一个n输入、b输出的组合逻辑功能的真值表。
- 一个3输入、4输出的组合功能的真值表，可以被存储在一个 $2^3 * 4$ ($8 * 4$) 的只读存储器中。忽略延迟，ROM的数据输出总是等于真值表中由地址输入所选择的那行输出位。

输入			输出			
A2	A1	A0	D3	D2	D1	D0
0	0	0	1	1	1	0
0	0	1	1	1	0	1
0	1	0	1	0	1	1
0	1	1	0	1	1	1
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

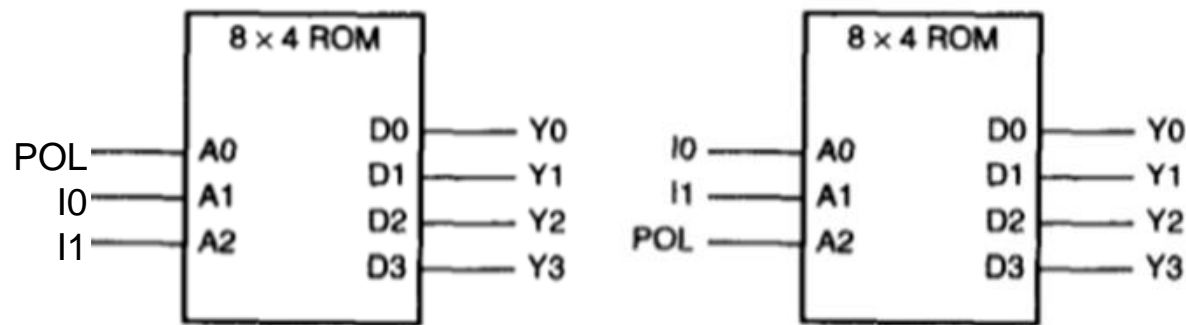
一个3输入4输出组合逻辑函数的真值表
(具有输出极性控制的2-4译码器)

用ROM实现组合逻辑函数

- 两种不同的方式来构建译码器：
 - 使用分立的门
 - 用包含真值表的 8×4 ROM
- 使用ROM的物理实现并不是唯一的。



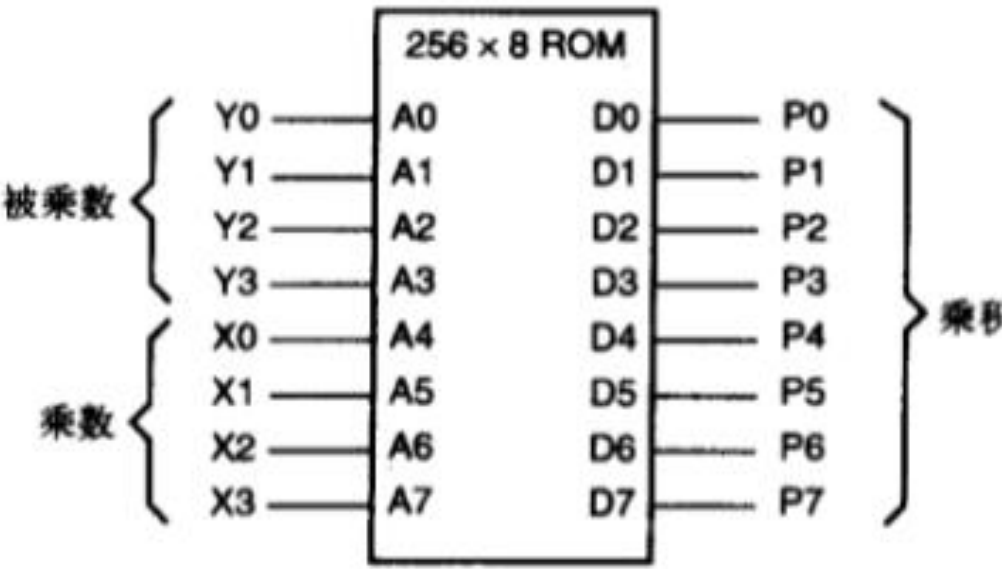
具有输出极性控制的2-4译码器



用存储真值表的 8×4 ROM构建2-4译码器

用ROM实现4位*4位无符号二进制数乘法

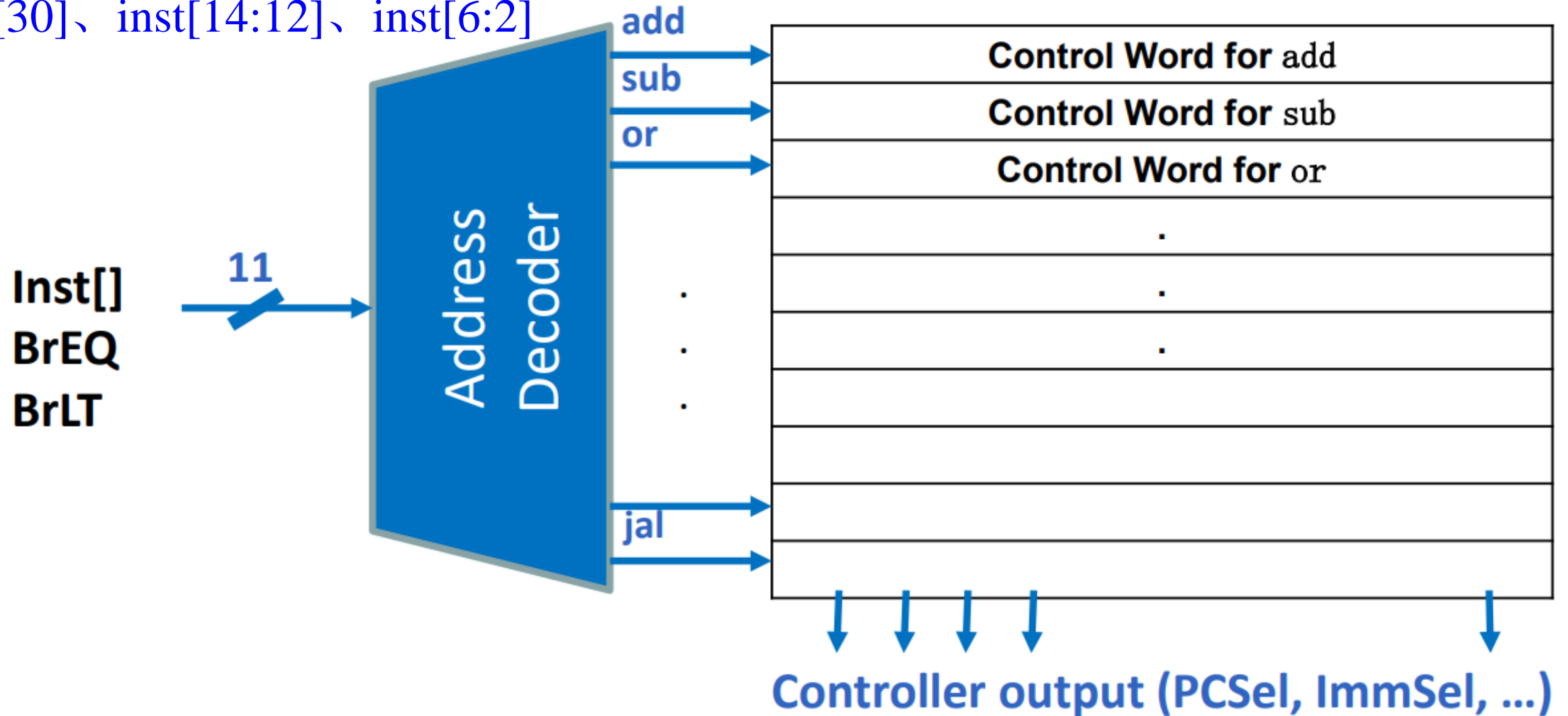
- 多少种组合？
 - $2^4 * 2^4 = 256$
- 乘积最多为几位？
 - $4 + 4 = 8$



地址	Y ₃ Y ₂ Y ₁ Y ₀															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
10	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
20	00	02	04	06	08	0A	0C	0E	10	12	14	16	18	1A	1C	1E
30	00	03	06	09	0C	0F	12	15	18	1B	1E	21	24	27	2A	2D
40	00	04	08	0C	10	14	18	1C	20	24	28	2C	30	34	38	3C
50	00	05	0A	0F	14	19	1E	23	28	2D	32	37	3C	41	46	4B
60	00	06	0C	12	18	1E	24	2A	30	36	3C	42	48	4E	54	5A
70	00	07	0E	15	1C	23	2A	31	38	3F	46	4D	54	5B	62	69
80	00	08	10	18	20	28	30	38	40	48	50	58	60	68	70	78
90	00	09	12	1B	24	2D	36	3F	48	51	5A	63	6C	75	7E	87
A0	00	0A	14	1E	28	32	3C	46	50	5A	64	6E	78	82	8C	96
B0	00	0B	16	21	2C	37	42	4D	58	63	6E	79	84	8F	9A	A5
C0	00	0C	18	24	30	3C	48	54	60	6C	78	84	90	9C	A8	B4
D0	00	0D	1A	27	34	41	4E	5B	68	75	82	8F	9C	A9	B6	C3
E0	00	0E	1C	2A	38	46	54	62	70	7E	8C	9A	A8	B6	C4	D2
F0	00	0F	1E	2D	3C	4B	5A	69	78	87	96	A5	B4	C3	D2	E1

ROM 控制器实现

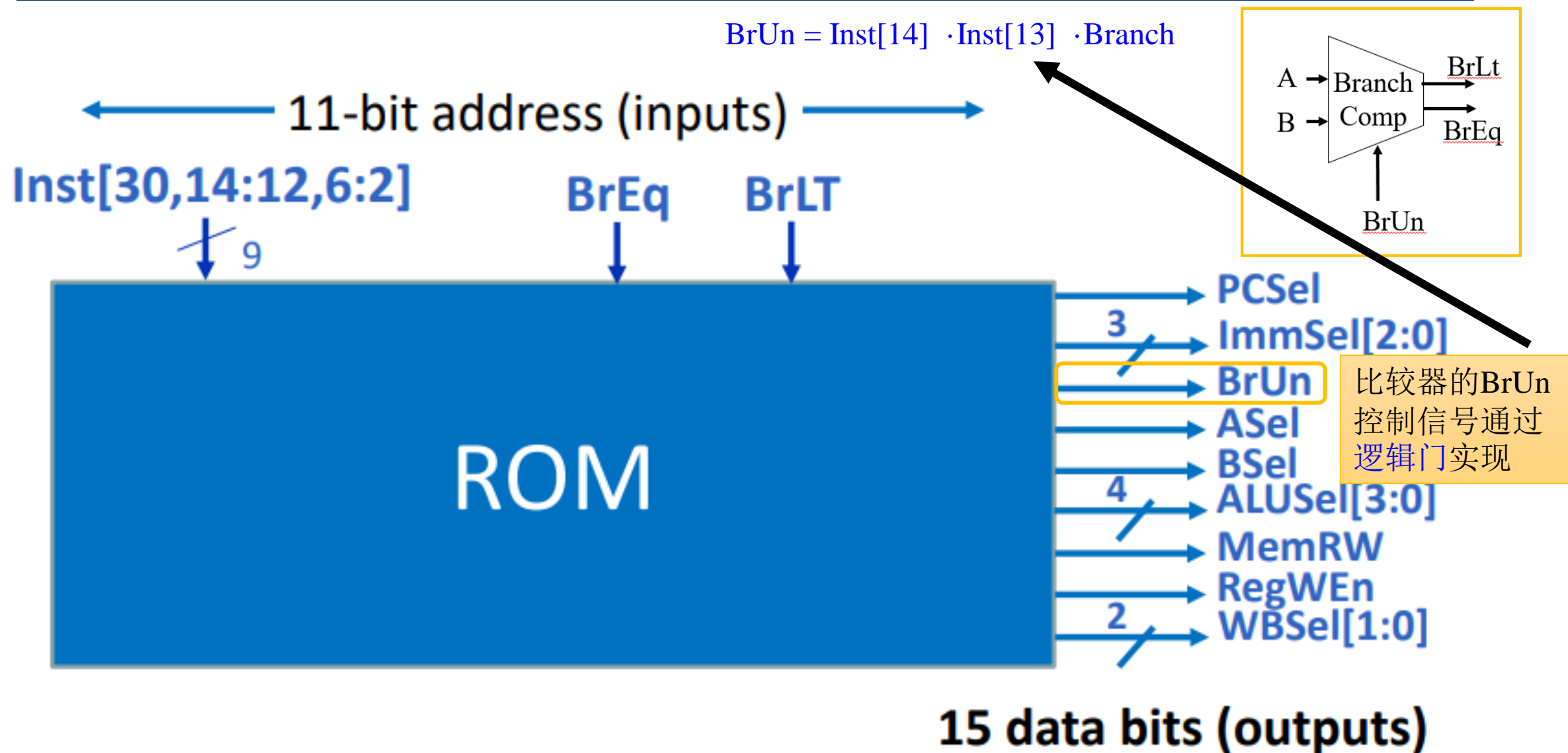
inst[30]、inst[14:12]、inst[6:2]



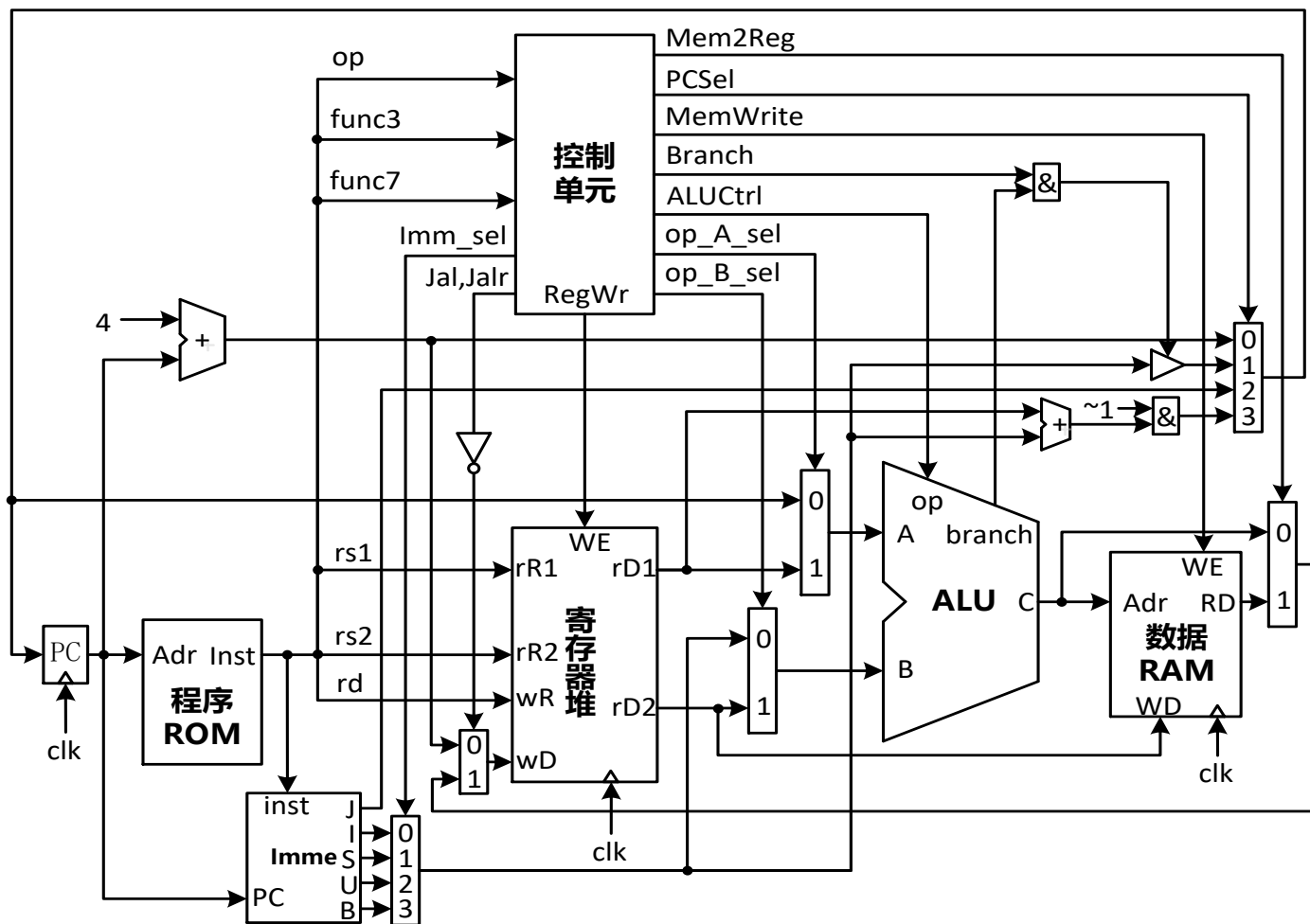
控制信号真值表

Inst[31:0]	BrEq	BrLt	PCSel	ImmSel	BrUn	ASel	BSel	ALUSel	MemRW	RegWen	WBSel
add	*	*	+4	*	*	Reg	Reg	Add	Read	1	ALU
sub	*	*	+4	*	*	Reg	Reg	Add	Read	1	ALU
addi	*	*	+4	I	*	Reg	Imm	Add	Read	1	ALU
lw	*	*	+4	I	*	Reg	Imm	Add	Read	1	Mem
sw	*	*	+4	S	*	Reg	Imm	Add	Write	0	*
beq	0	*	+4	B	*	PC	Imm	Add	Read	0	*
beq	1	*	ALU	B	*	PC	Imm	Add	Read	0	*
bne	0	*	ALU	B	*	PC	Imm	Add	Read	0	*
bne	1	*	+4	B	*	PC	Imm	Add	Read	0	*
blt	*	1	ALU	B	0	PC	Imm	Add	Read	0	*
bltu	*	1	ALU	B	1	PC	Imm	Add	Read	0	*
jalr	*	*	ALU	I	*	Reg	Imm	Add	Read	1	PC+4
jal	*	*	ALU	J	*	PC	Imm	Add	Read	1	PC+4
auipc	*	*	+4	U	*	PC	Imm	Add	Read	1	ALU

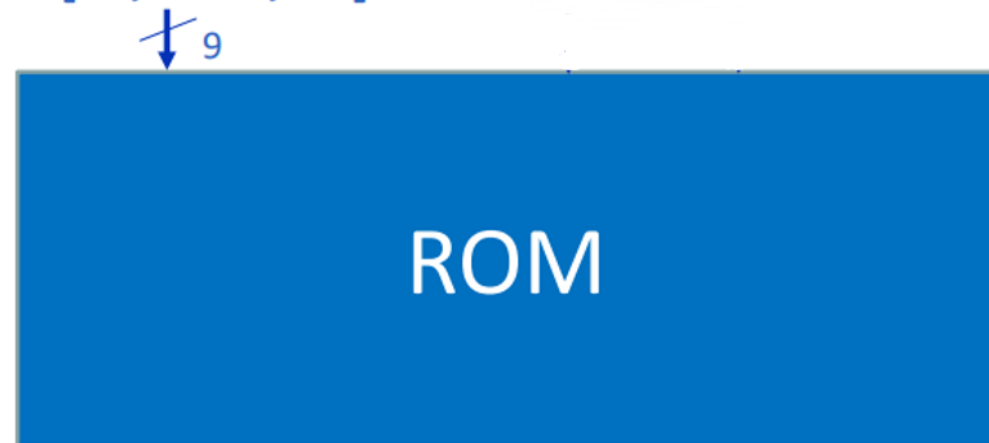
ROM 控制—比较器方案



条件分支比较通过ALU实现的数据通路和ROM控制



输入 `op`、`func3`、`func7`
`Inst[30,14:12,6:2]`



控制信号

回顾

- 实现了一个处理器
 - 能够在—个时钟周期内执行所有RISC-V指令
 - 并非所有指令都会用到所有硬件单元
 - 关键路径
- 5个执行阶段
 - IF、ID、EX、MEM、WB
 - 有的阶段只有部分指令才会用到
- 控制器指定如何执行指令
 - 基于ROM或组合逻辑实现

单周期数据通路必须有可独立访问的指令存储器和数据存储器，因为：

提交

- ☐ A RISC-V指令和数据的格式不同，所以需要不同的存储器
- ☐ B 使用独立的存储器会比较便宜
- ☒ C 处理器在一个周期内只能操作每个部件一次，而在一个周期内不可能对一个（单端口）存储器进行两次存取