

计算机组成原理

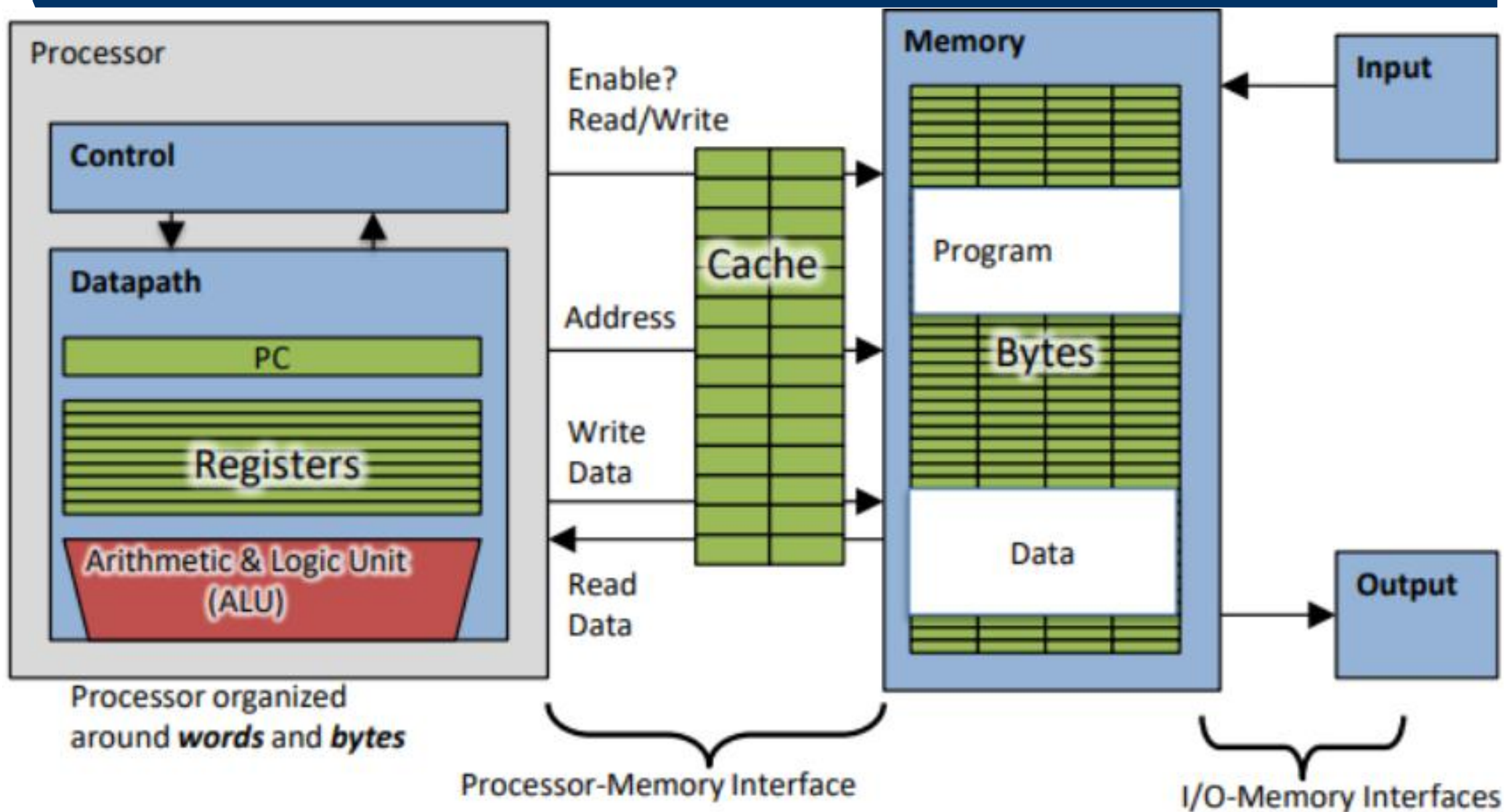
王鸿鹏 wanghp@hit.edu.cn

计算机科学与技术学院

第七章 处理器设计

- RISC-V数据通路
 - 数据通路概念
 - RISC-V部分指令的数据通路
- 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
(例如: `addi t0 x0 6`)

010111010100010101..1

数字电路
(逻辑门、
寄存器)

执行完毕这条指令后，
寄存器t0的值为6

CPU的组成部分

- **数据通路**是处理器中执行所需操作的硬件

- 执行控制器的操作（例如，控制器告诉数据通路，执行add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？需要写寄存器吗？
写哪个寄存器？读哪个寄存器？

≈≈处理器的大脑

CPU的组成部分

addi t0 x0 6

010111010100010101..1

数据通路

从控制器来的信息：

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

控制通路

决定：要让数据通路执行什么操作？
（加减乘除/逻辑/比较？）

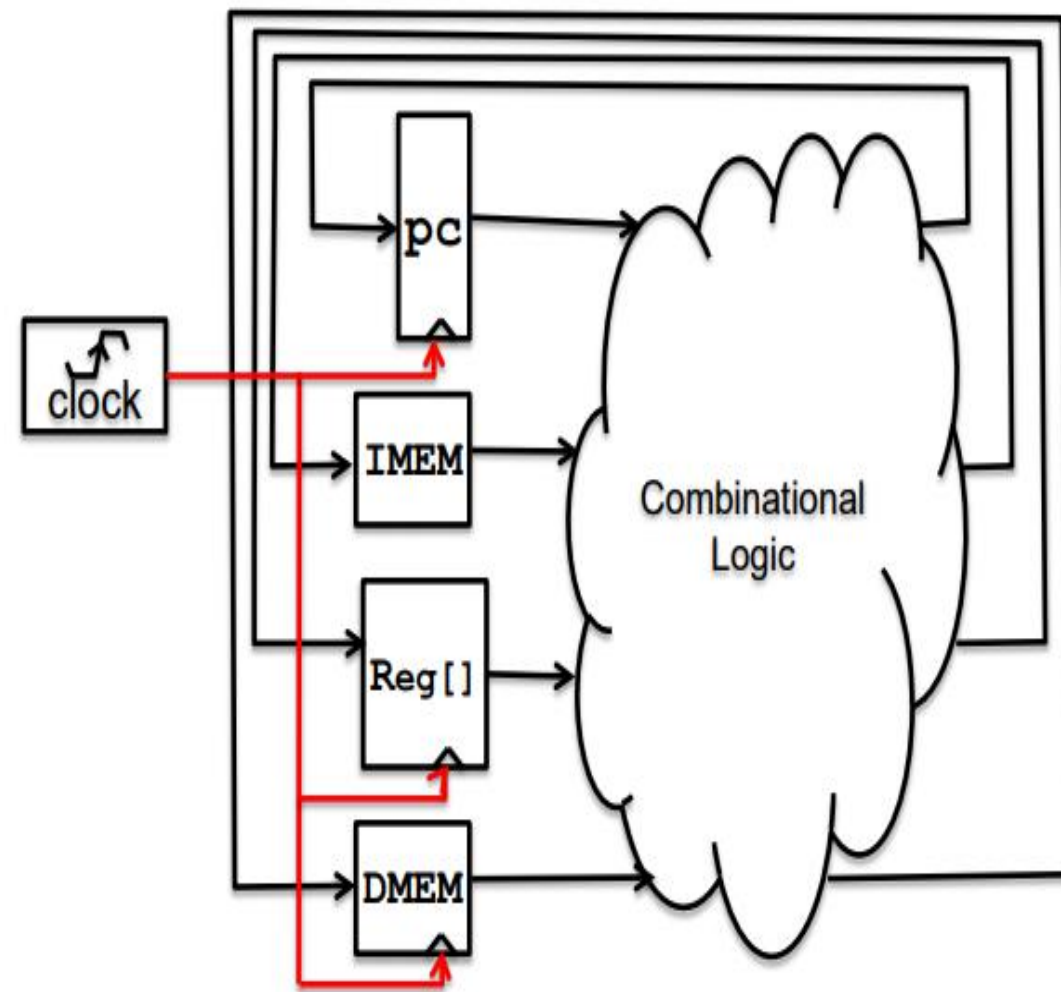
源操作数应该如何选择？

（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？）

单周期CPU设计模型

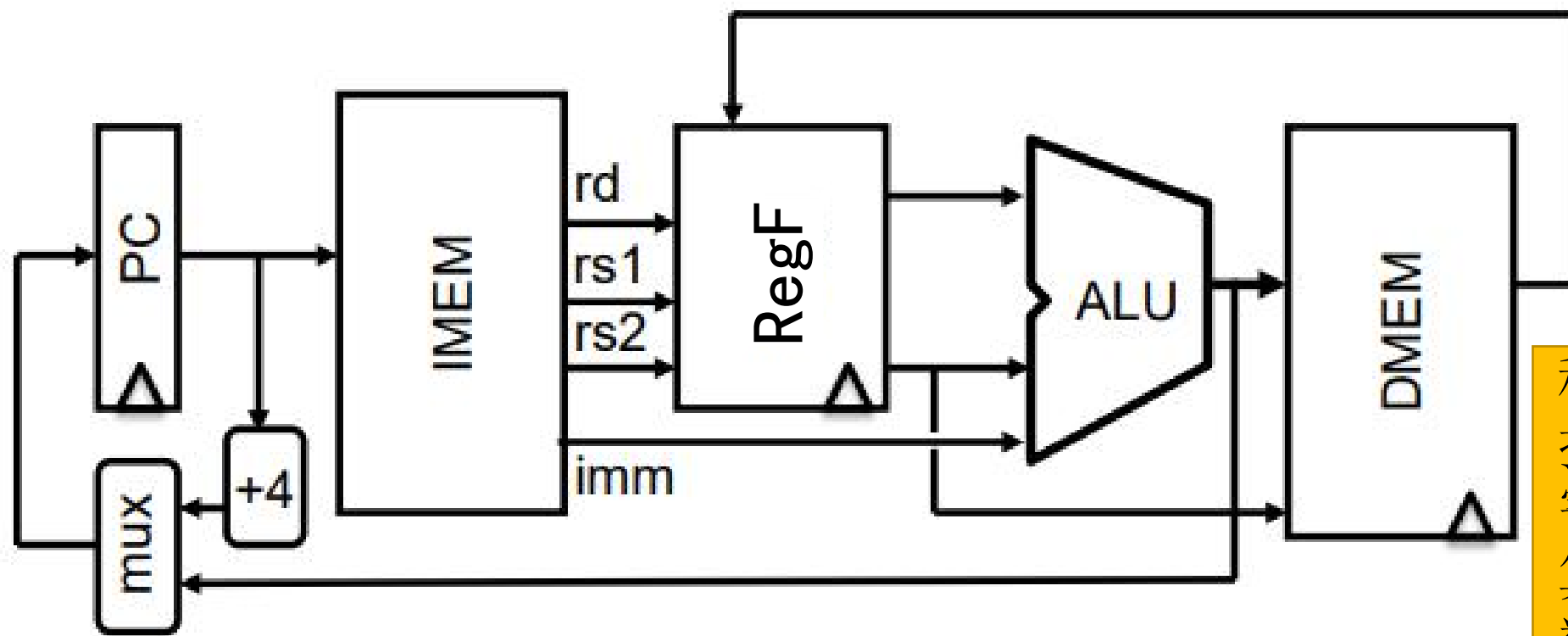
- 每个时钟节拍执行一条指令
- 当前状态值通过一系列组合逻辑得到当前指令机器码，以及对应的操作码类型，操作数，结果。
- 在下一时钟上升沿到来时，所有组合逻辑单元输出的状态值保持或者更新对应寄存器，同时开始下一个时钟周期的执行操作。



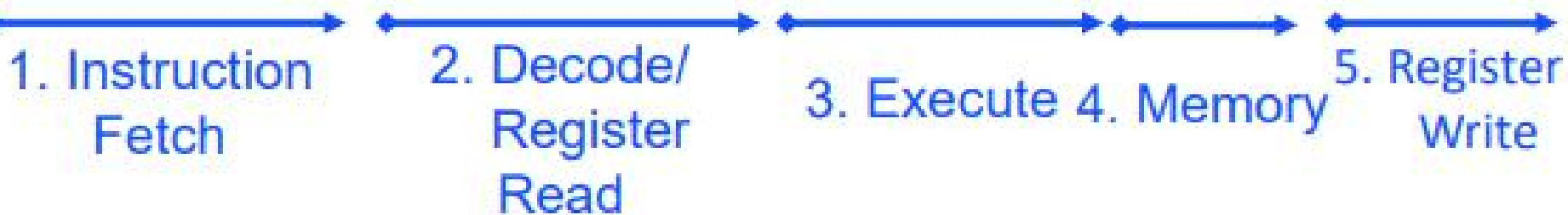
分阶段的数据通路——概述

- 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

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

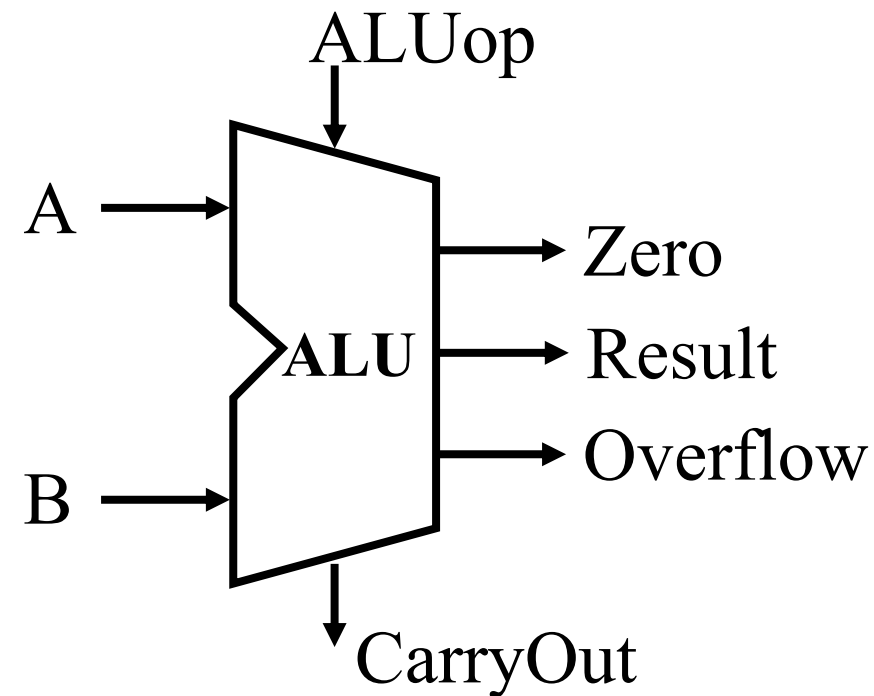
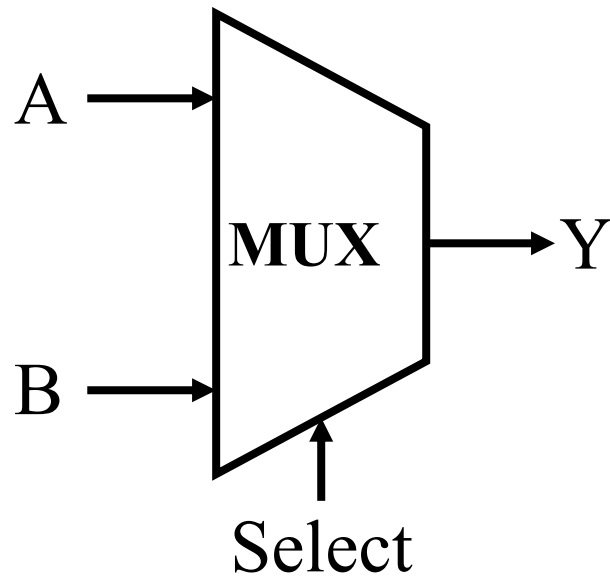
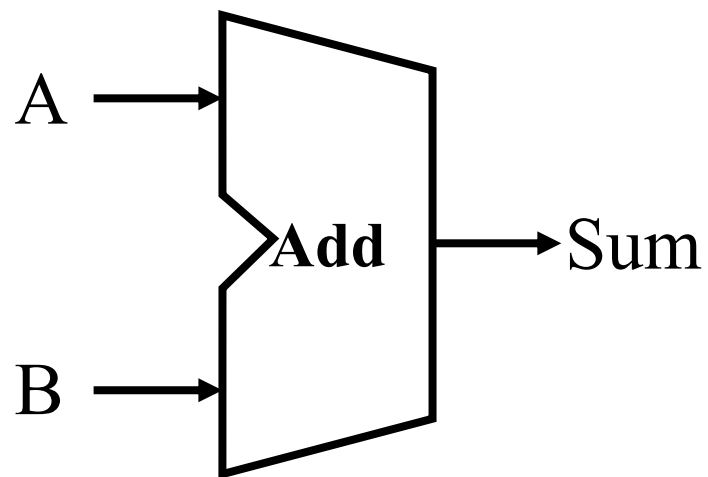


time →

数据通路的五个阶段

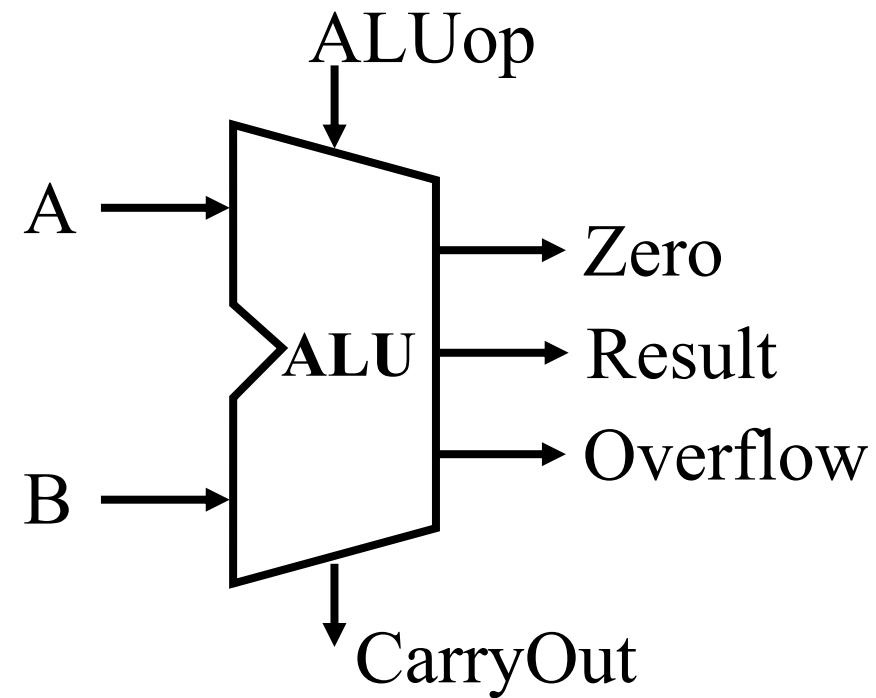
- 取指: Instruction Fetch (IF)
- 译码: Instruction Decode (ID)
- 执行: EXecute (EX) - ALU (Arithmetic-Logic Unit)
- 访存: MEMory access (MEM)
- 写回: Write Back to register (WB)

数据通路模块——组合逻辑单元



ALU功能需求

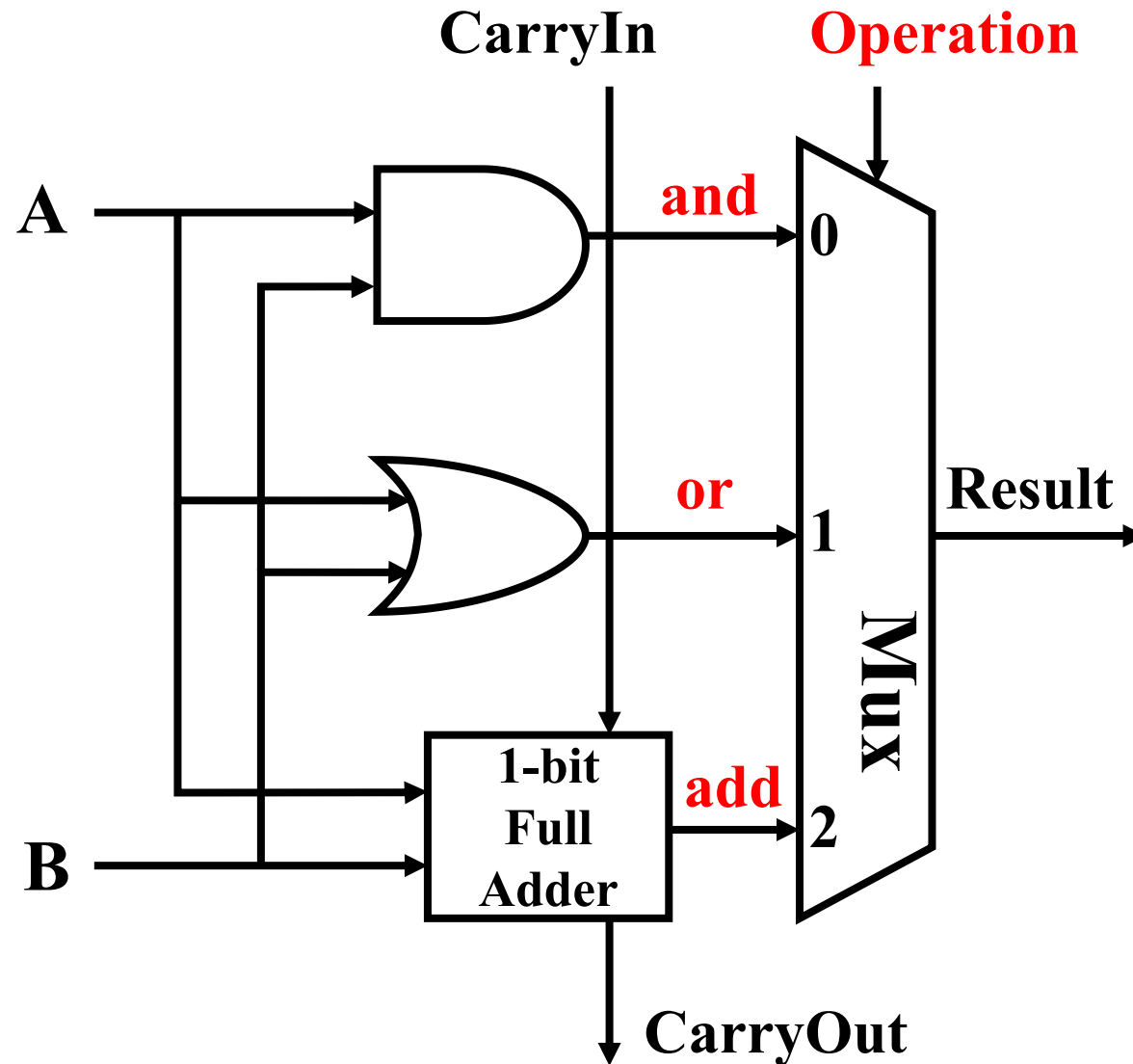
ALU Control(ALUOp)	操作
0000	and
0001	or
0010	add
0110	subtract
0111	set on less than
1100	nor



ALU设计技巧

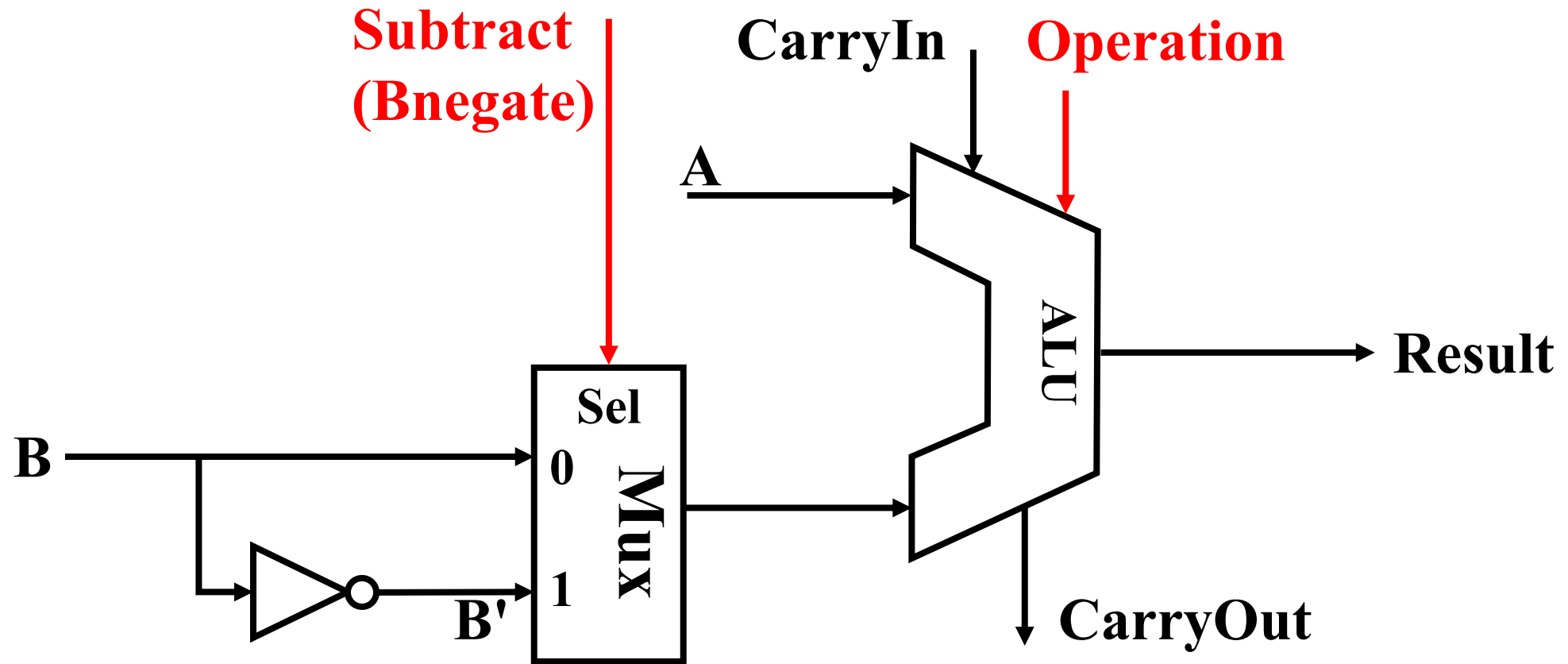
- 从简单开始：然后再进行扩展
- 分而治之：从1位ALU设计开始
- 利用已知的元件或者组件，组合起来实现复杂功能

ALU——简单运算



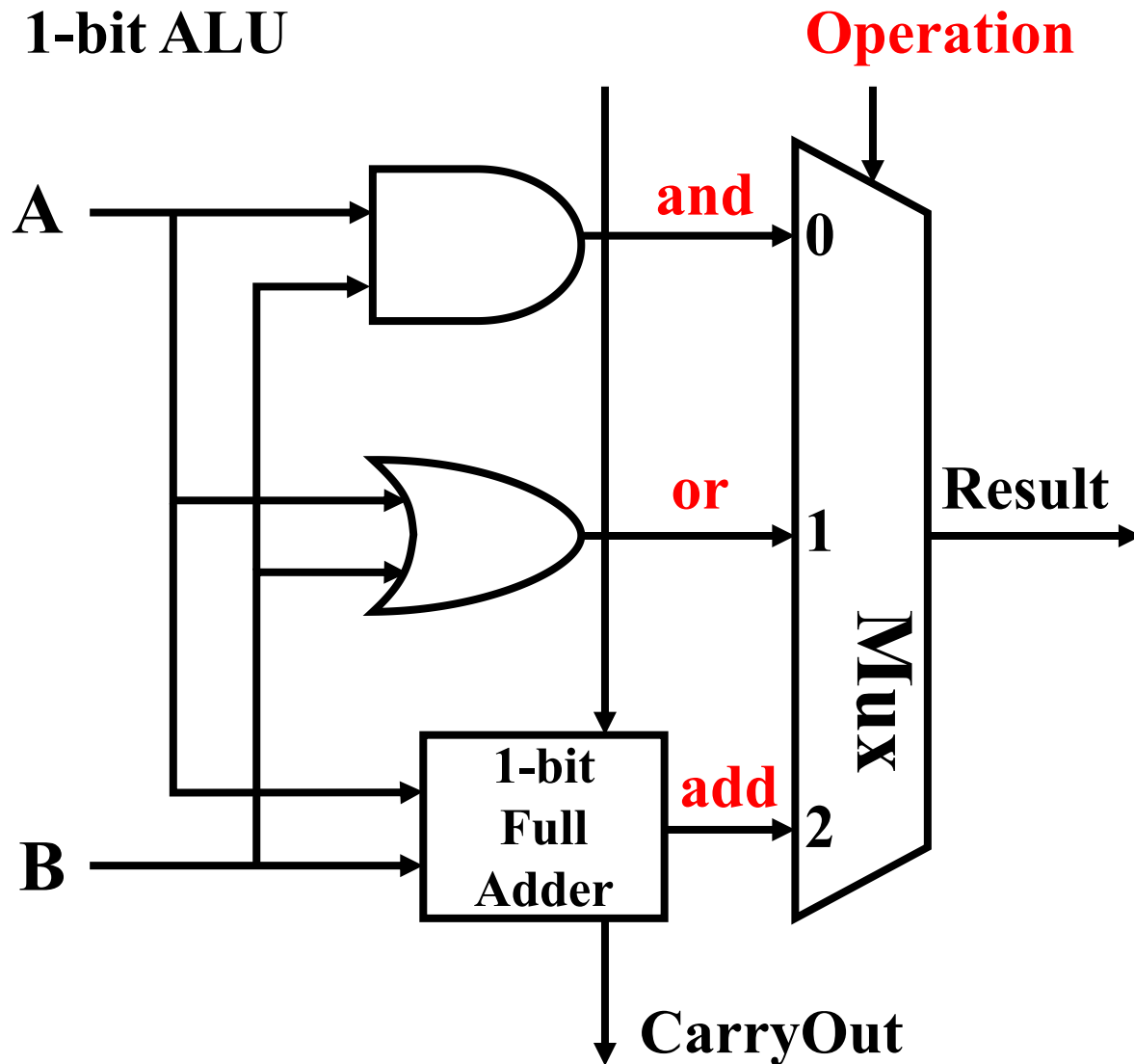
ALU——如何进行减法运算

- $A - B = A + B' + 1$

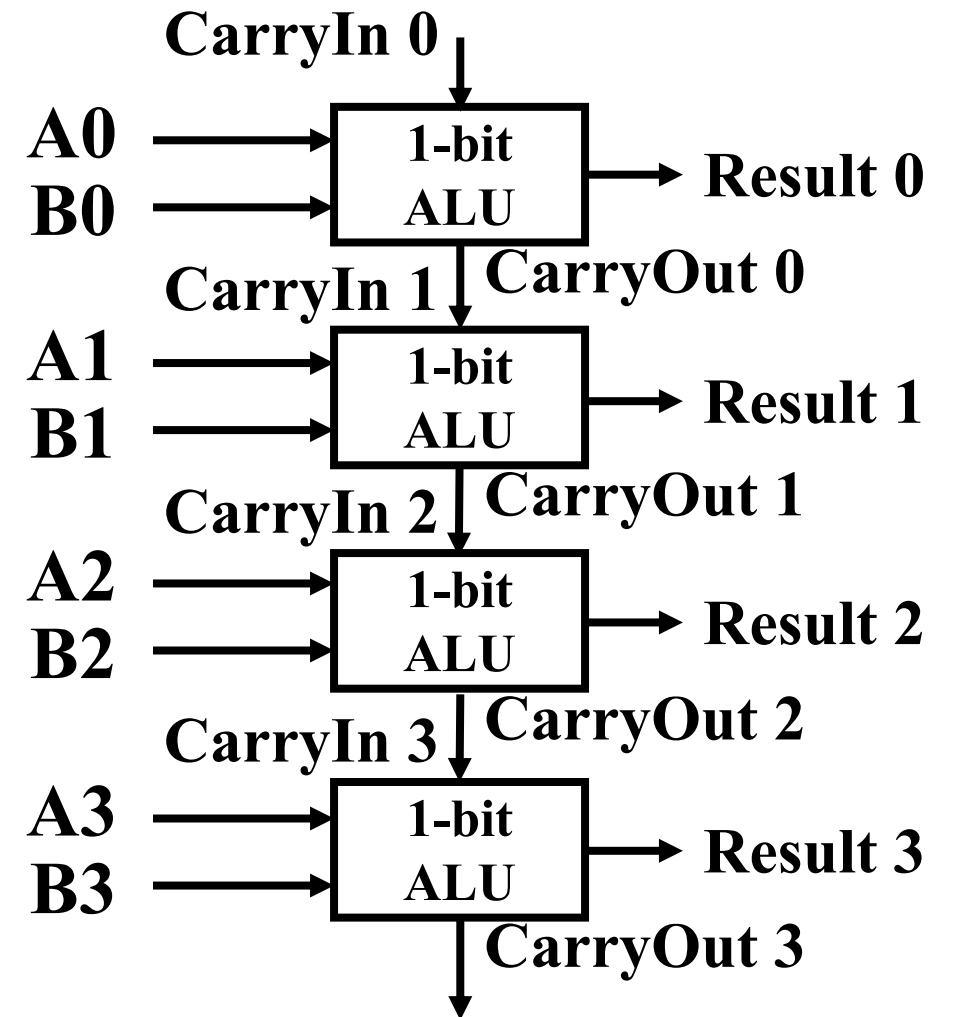


1-bit到多位ALU

1-bit ALU

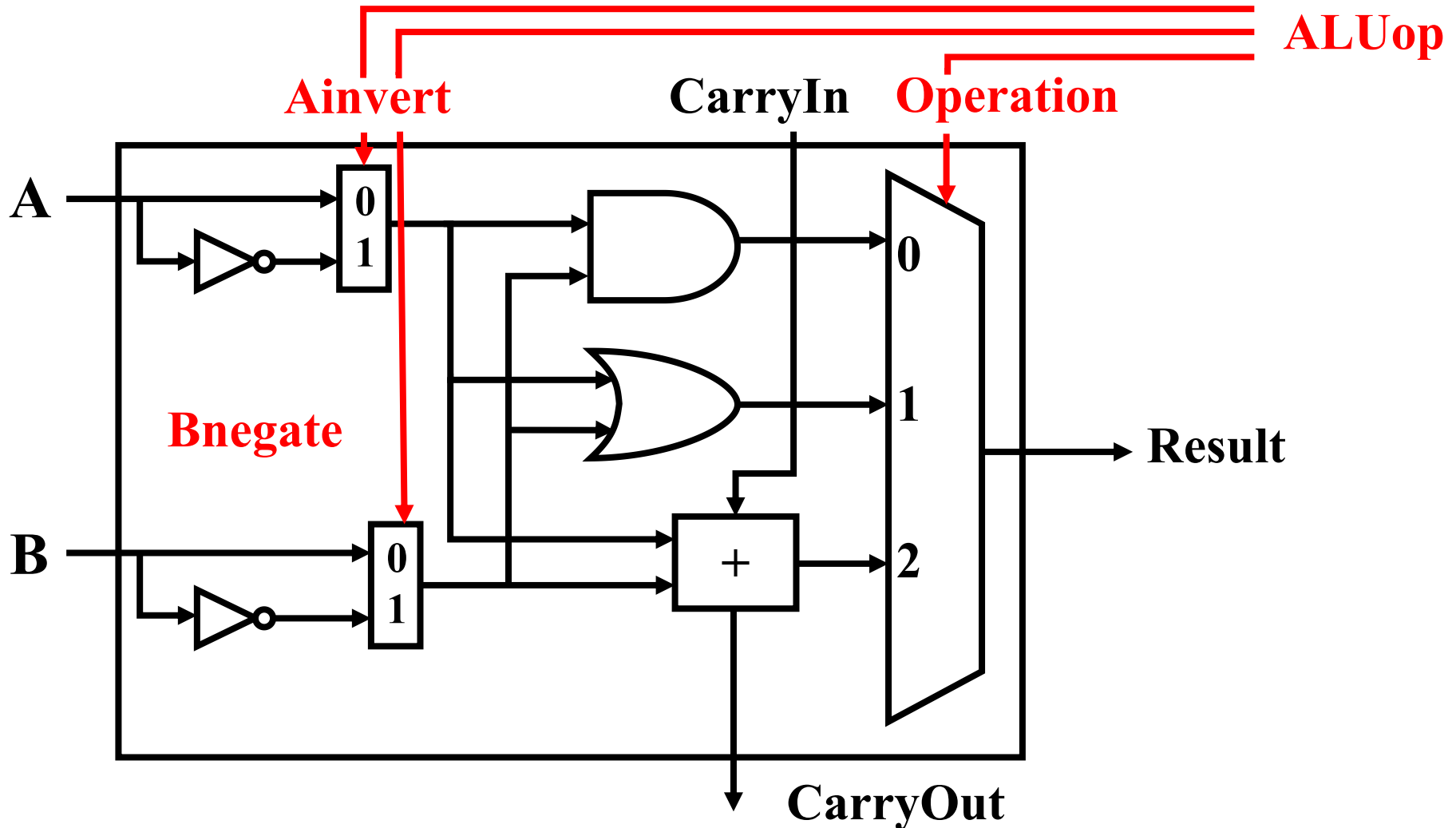


4-bit ALU



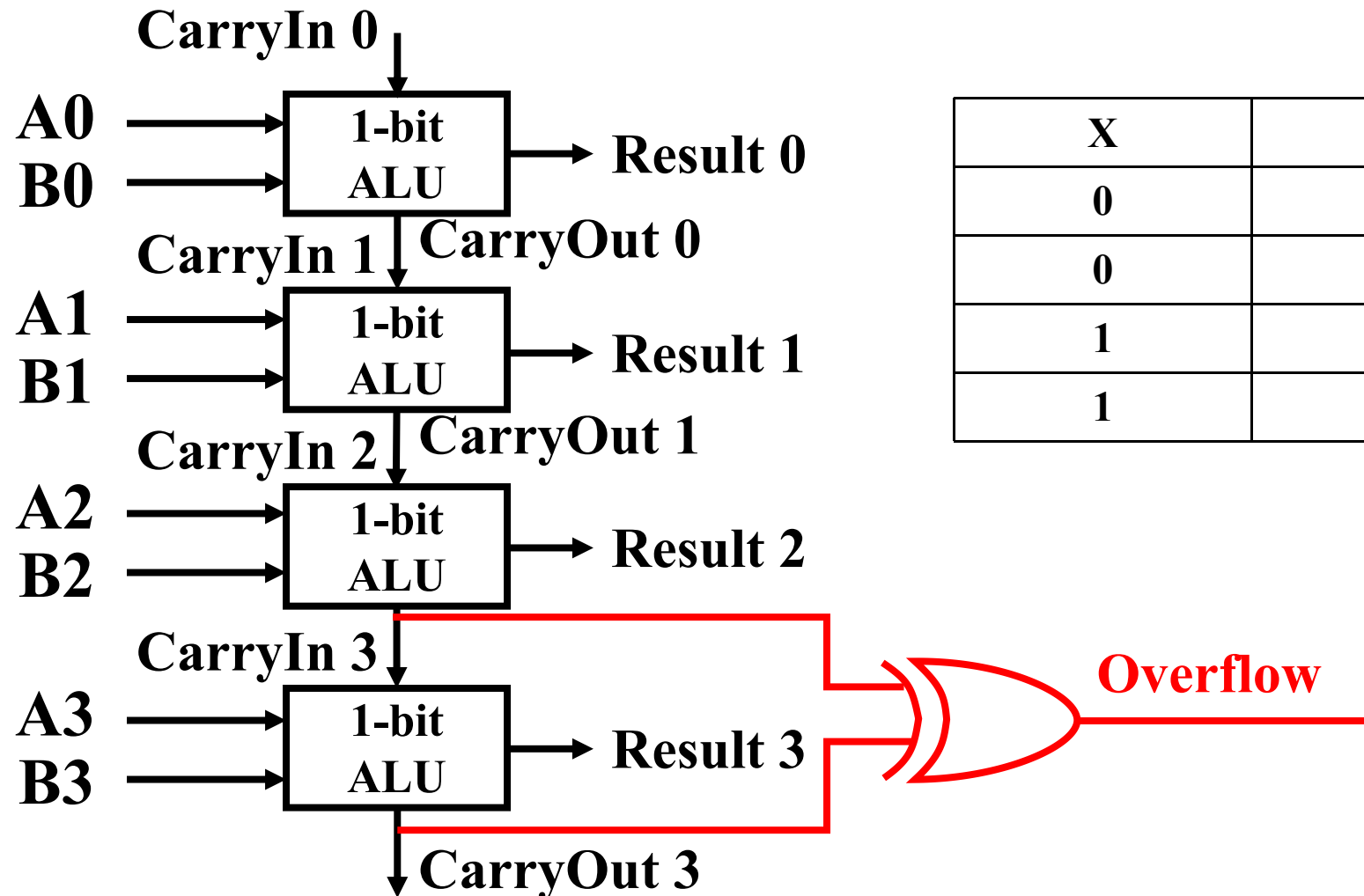
ALU——nor运算

$$A \text{ nor } B = (\text{not } A) \text{ and } (\text{not } B)$$



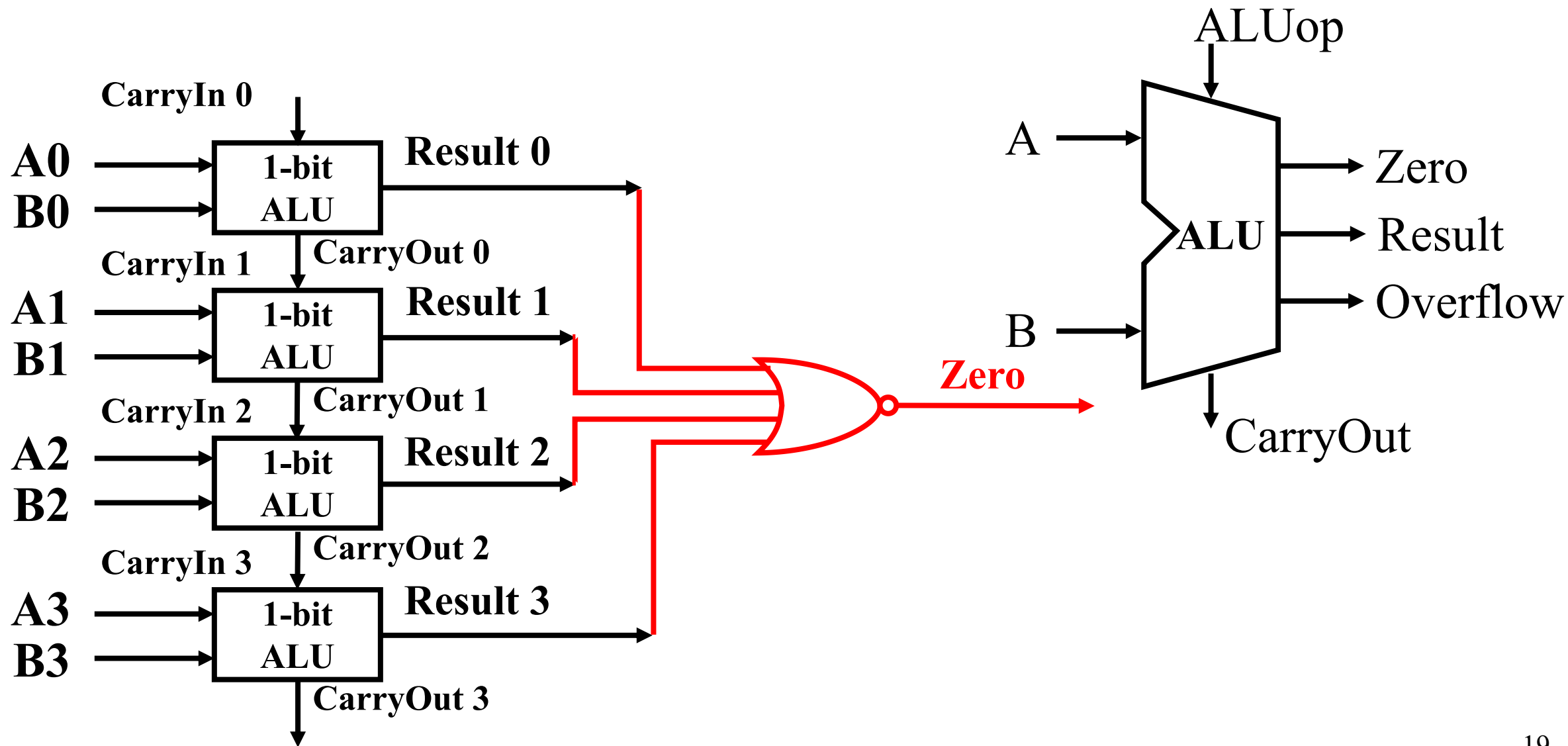
ALU——溢出检测逻辑

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

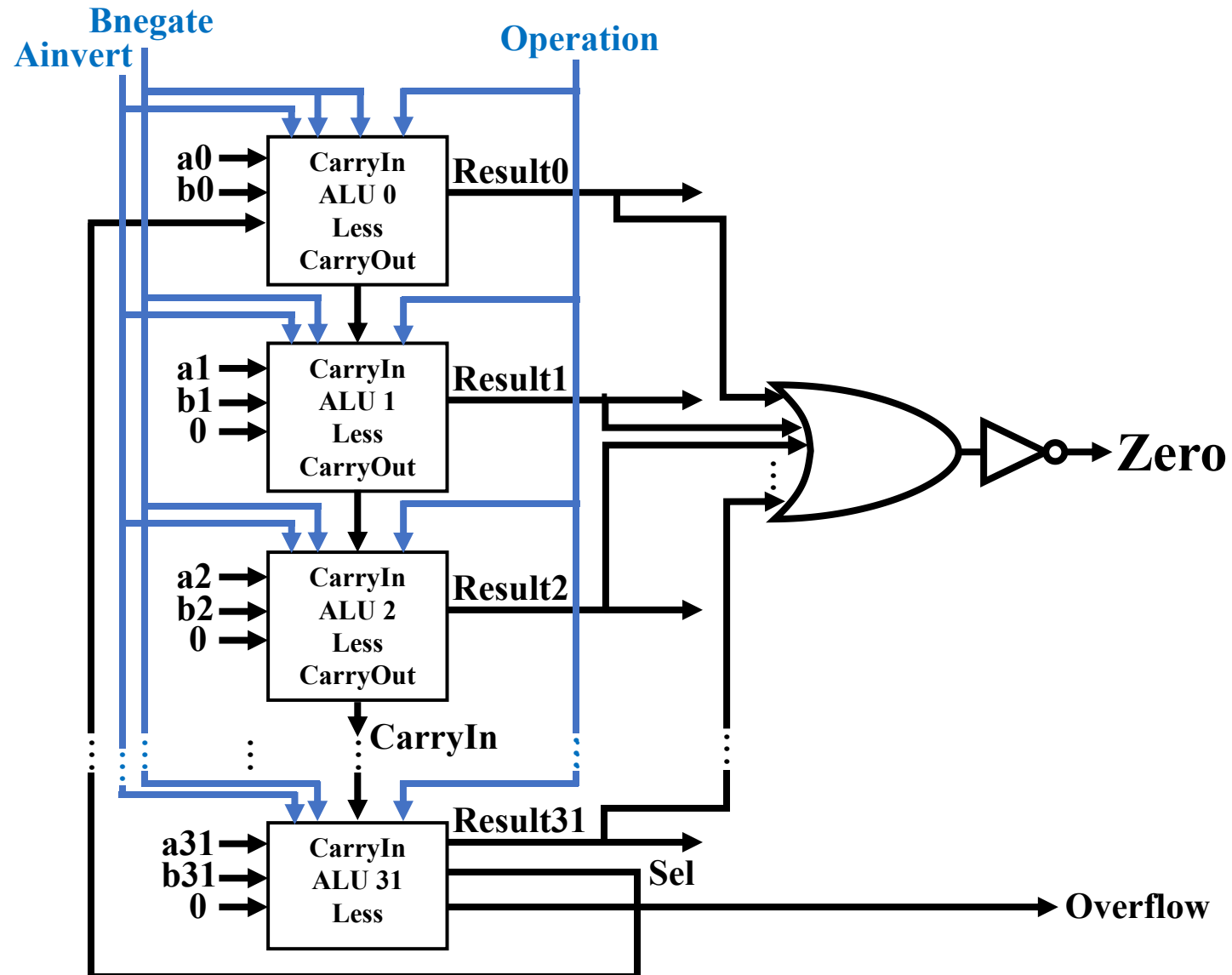


X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

ALU——判零逻辑



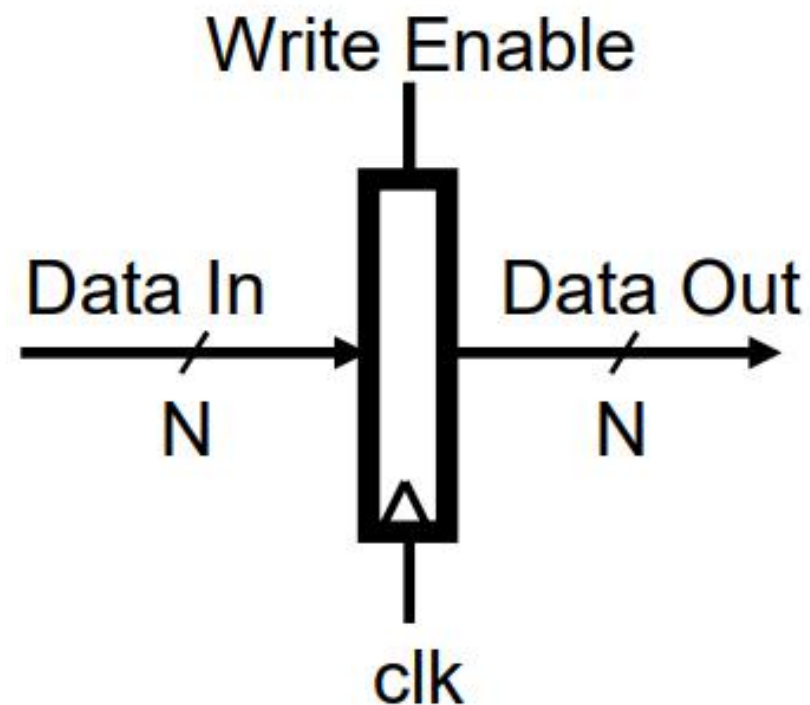
32bit ALU



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

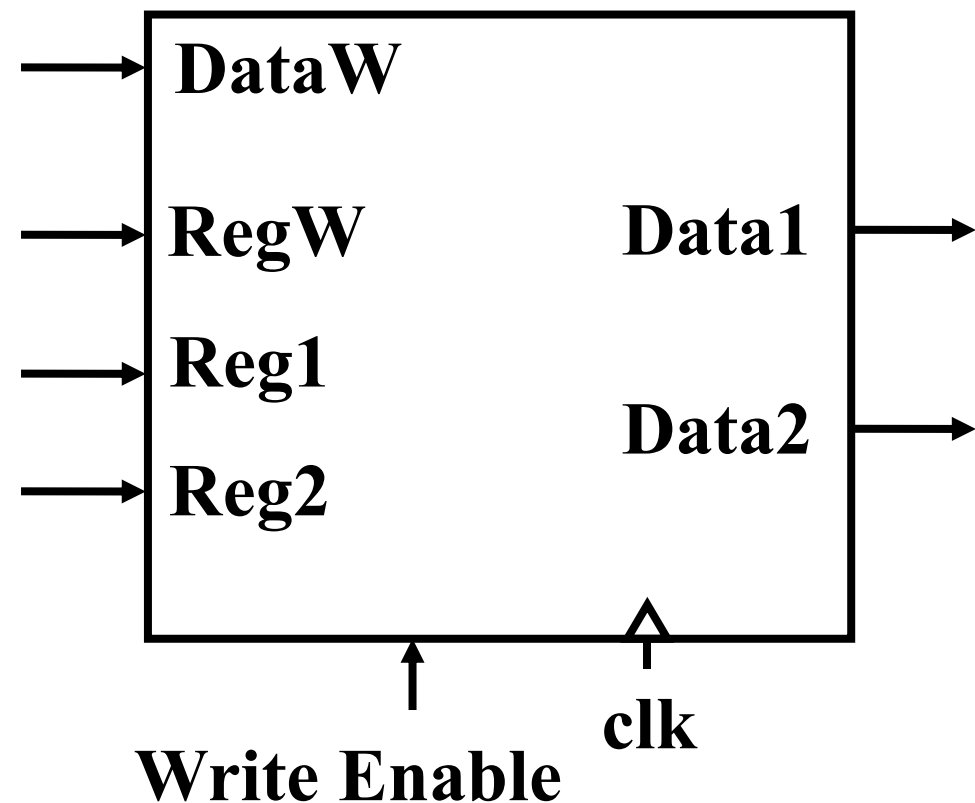
数据通路模块——状态与时序单元

- 寄存器堆和存储器



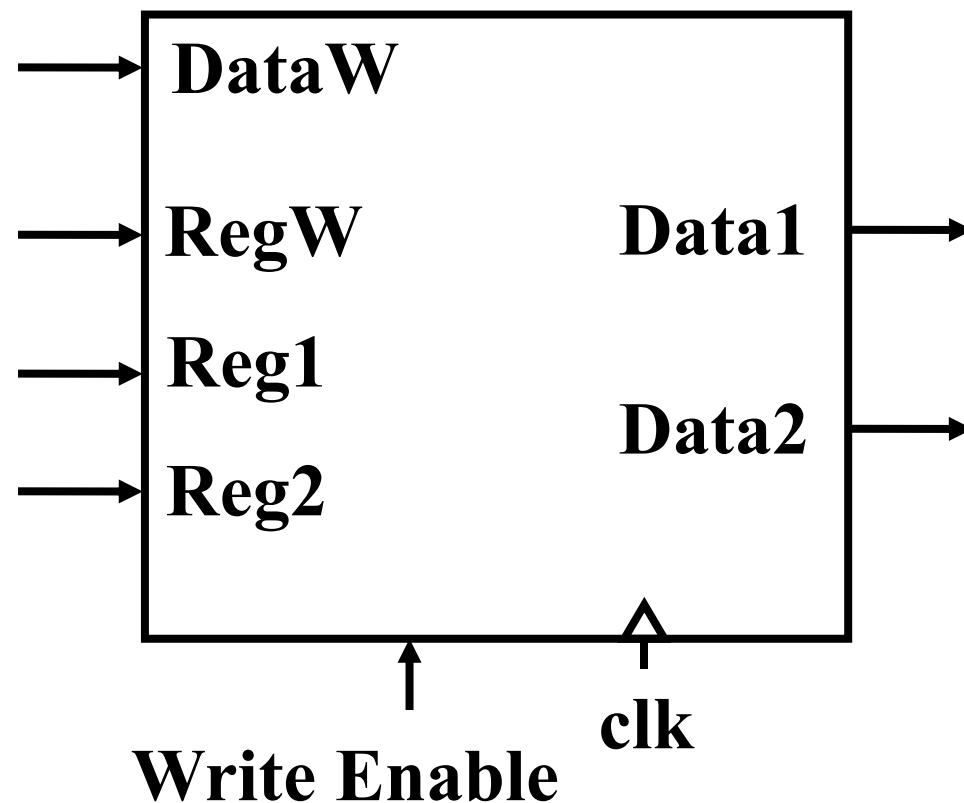
数据通路模块——寄存器堆(reg file)

- 寄存器文件由32个寄存器组成：
 - 两个输出总线：Data1和Data2
 - 一个输入总线：DataW
 - Reg1选择寄存器输出到Data1上
 - Reg2选择寄存器输出到Data2上
 - RegW选择寄存器，当写使能为1时，将DataW上的数据写入



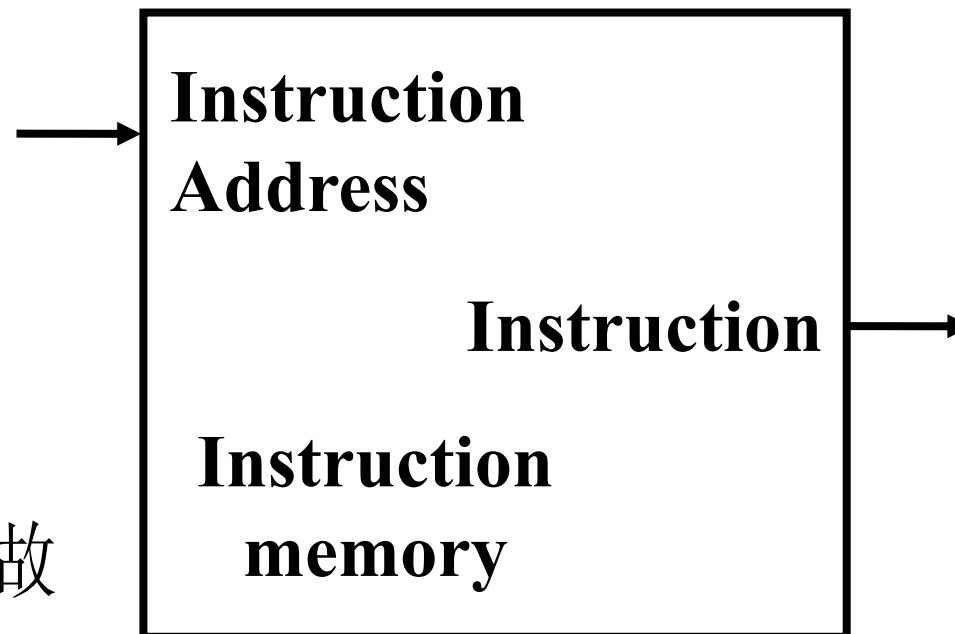
数据通路模块——寄存器堆(reg file)

- Clk输入仅在**写**寄存器时起作用
- 在读操作时，寄存器堆的行为类似于普通的**组合逻辑**功能
 - 当Reg1或Reg2有效时，经过一定访问响应时间，得到有效的Data1或Data2输出的数据



数据通路模块——存储器

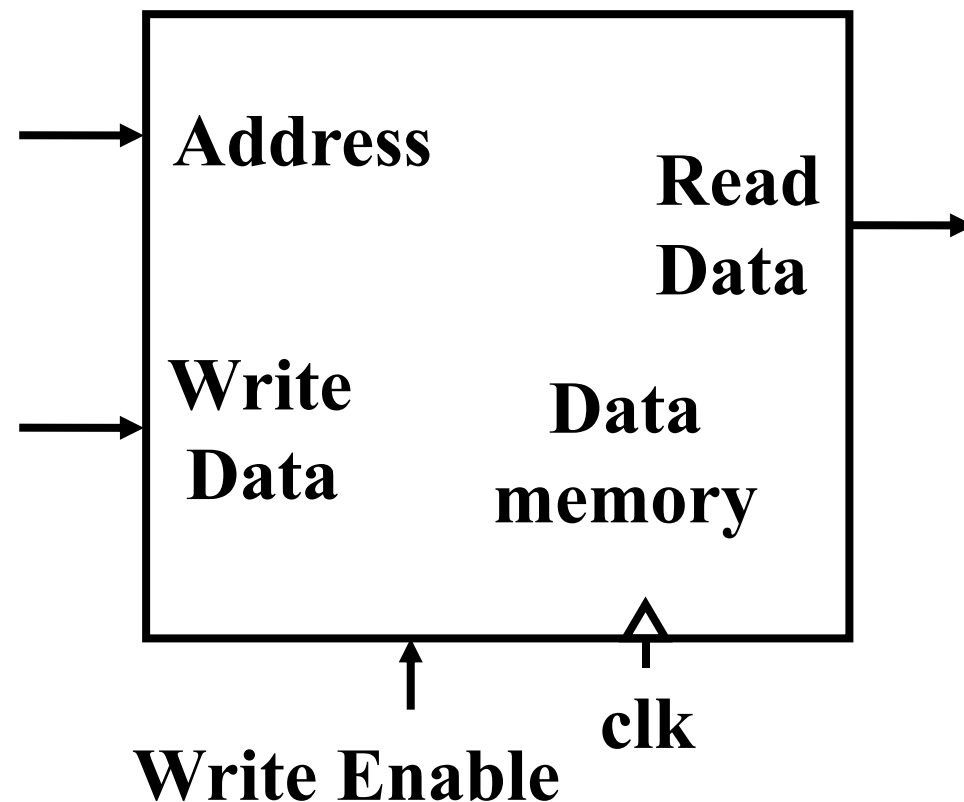
- 指令存储器
 - 输入总线: Instruction Address
 - 输出总线: Instruction
- 数据通路中**不会写**指令存储器
- 读操作时, 指令存储器可以看做是**组合逻辑**电路



数据通路模块——存储器

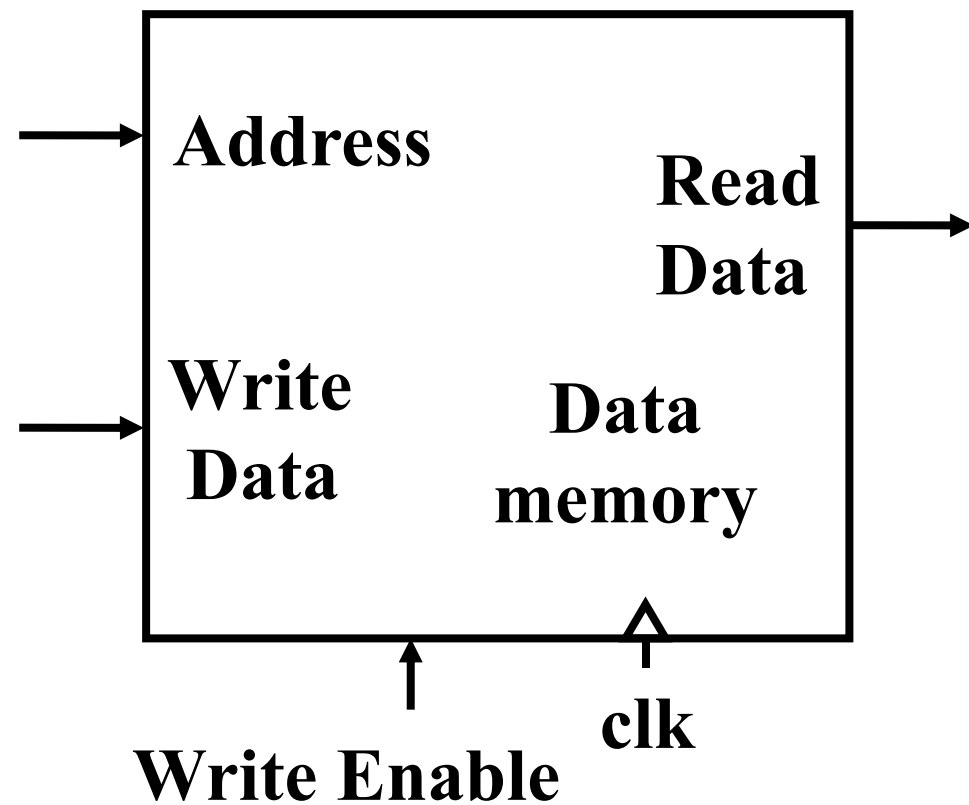
- 数据存储器

- 输入总线: Write Data
- 输出总线: Read Data
- 写使能端: Write Enable
- 地址端: Address
- 时钟信号端: clk



数据通路模块——存储器之读写

- 读存储器时，由地址端输入的地址，选择存储器中对应地址上的数据，输出到Read Data
- 写存储器时，将写使能置1，表示有效，将Write Data输入的数据写到Address指定的存储单元



RISC-V 主要状态单元——寄存器

- 由32个寄存器(0-31)构成的寄存器文件
- 由指令中的rs1字段指定读取的源寄存器1，由rs2字段指定读取的源寄存器2，由rd字段指定写入的目的寄存器
- 另外x0寄存器中恒为0值，对它的写入操作可以忽略
- 程序计数器（PC寄存器），保存着当前执行指令的地址

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
- 使用**分立**的两块存储器分别作为指令存储器Imem和数据存储器Dmem（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

普林斯顿架构 vs 哈佛架构

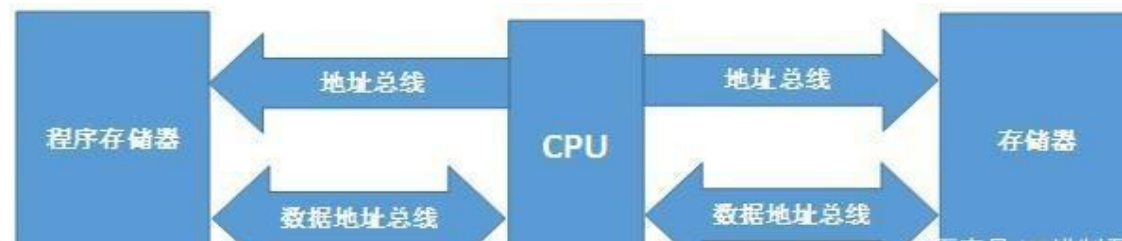
- 冯诺依曼架构（以X86为代表）

- 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 and 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）

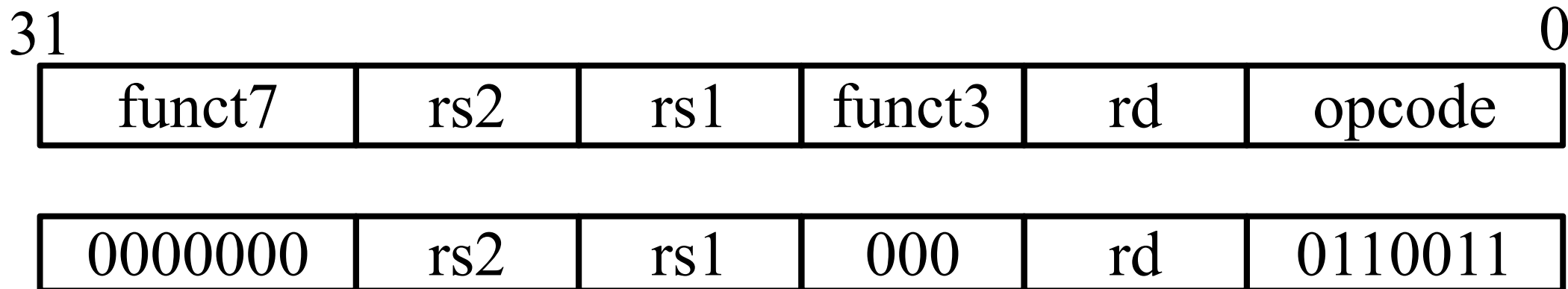
- 存储器分为数据存储器 and 程序存储器，总线分为程序存储器的数据总线 and 地址总线以及数据存储器的数据总线 and 地址总线。
- 可同时对数据和程序进行操作，具有较高的执行效率。
- 指令和数据可以有不同的宽度。



第七章 处理器设计

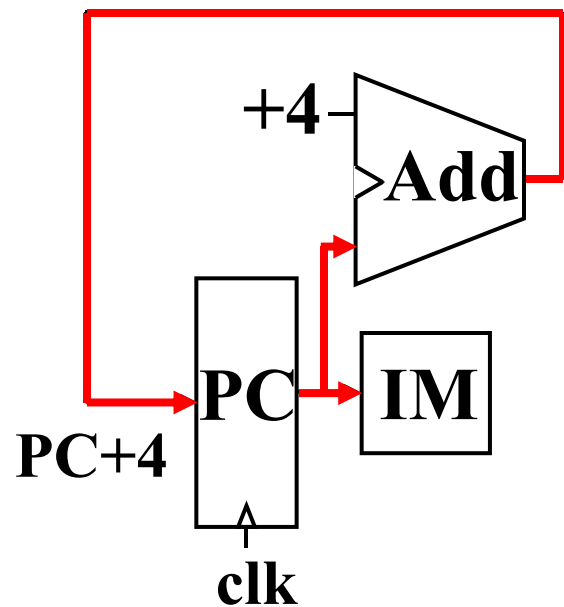
- RISC-V数据通路
 - 数据通路概念
 - RISC-V部分指令的数据通路
- RISC-V控制器

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

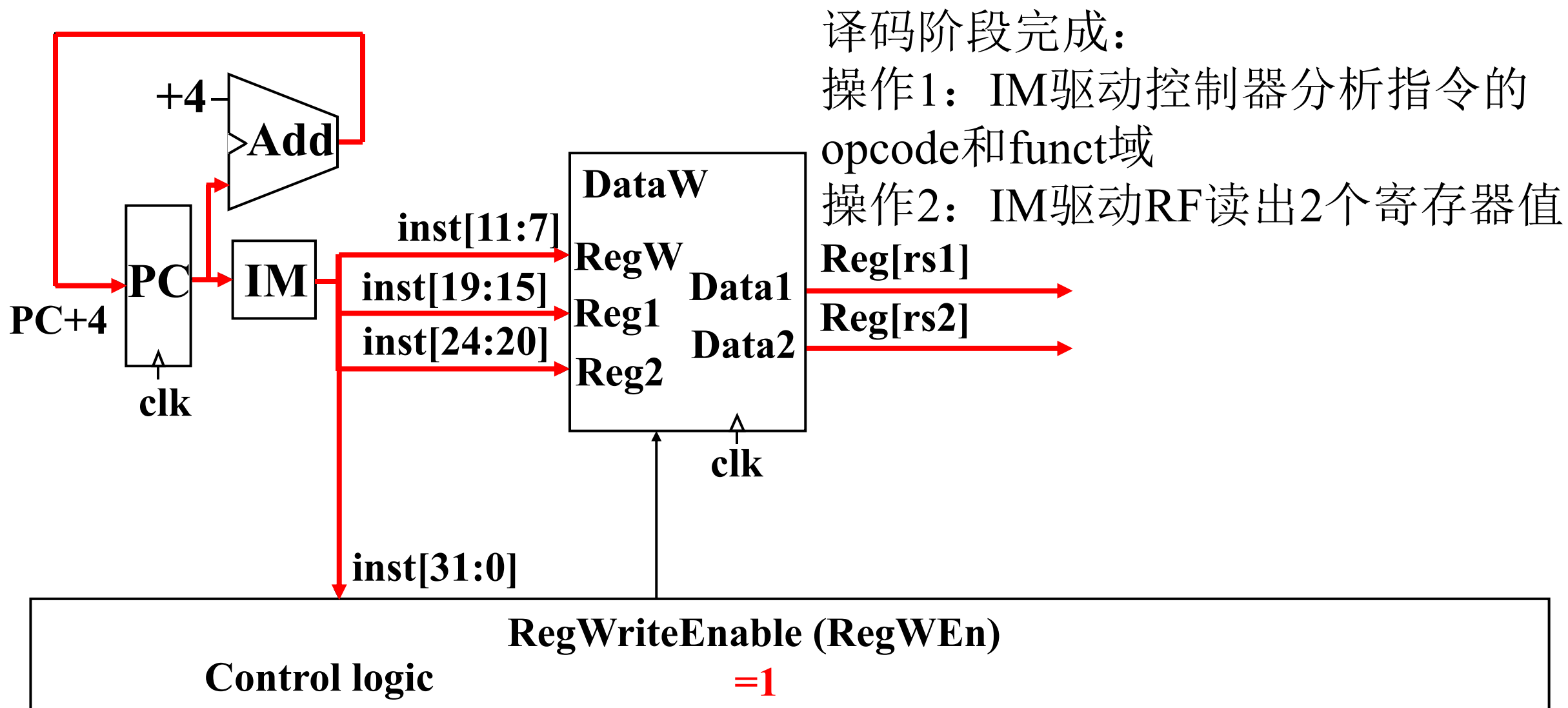


取指阶段完成:

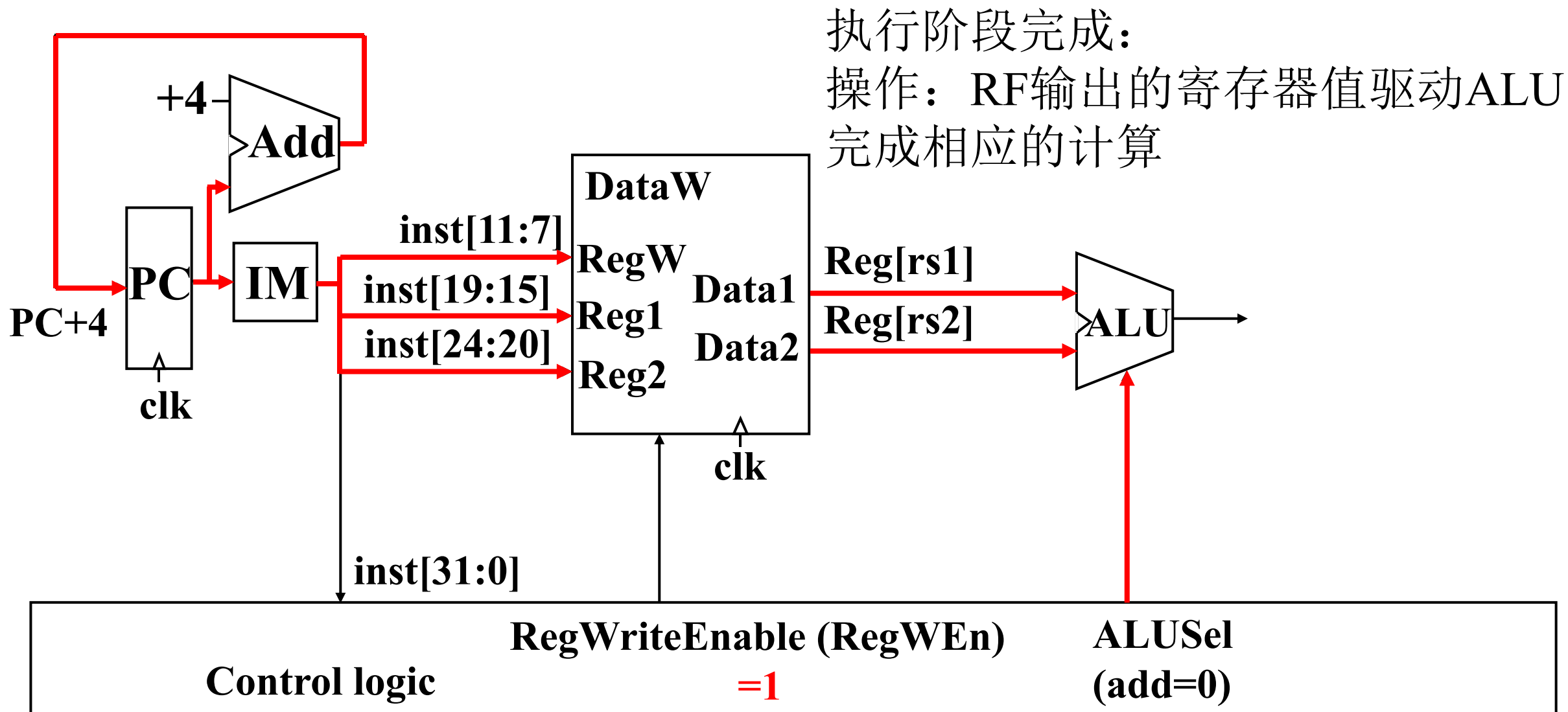
操作1: PC驱动IM输出指令

操作2: PC驱动加法器计算下一个PC值

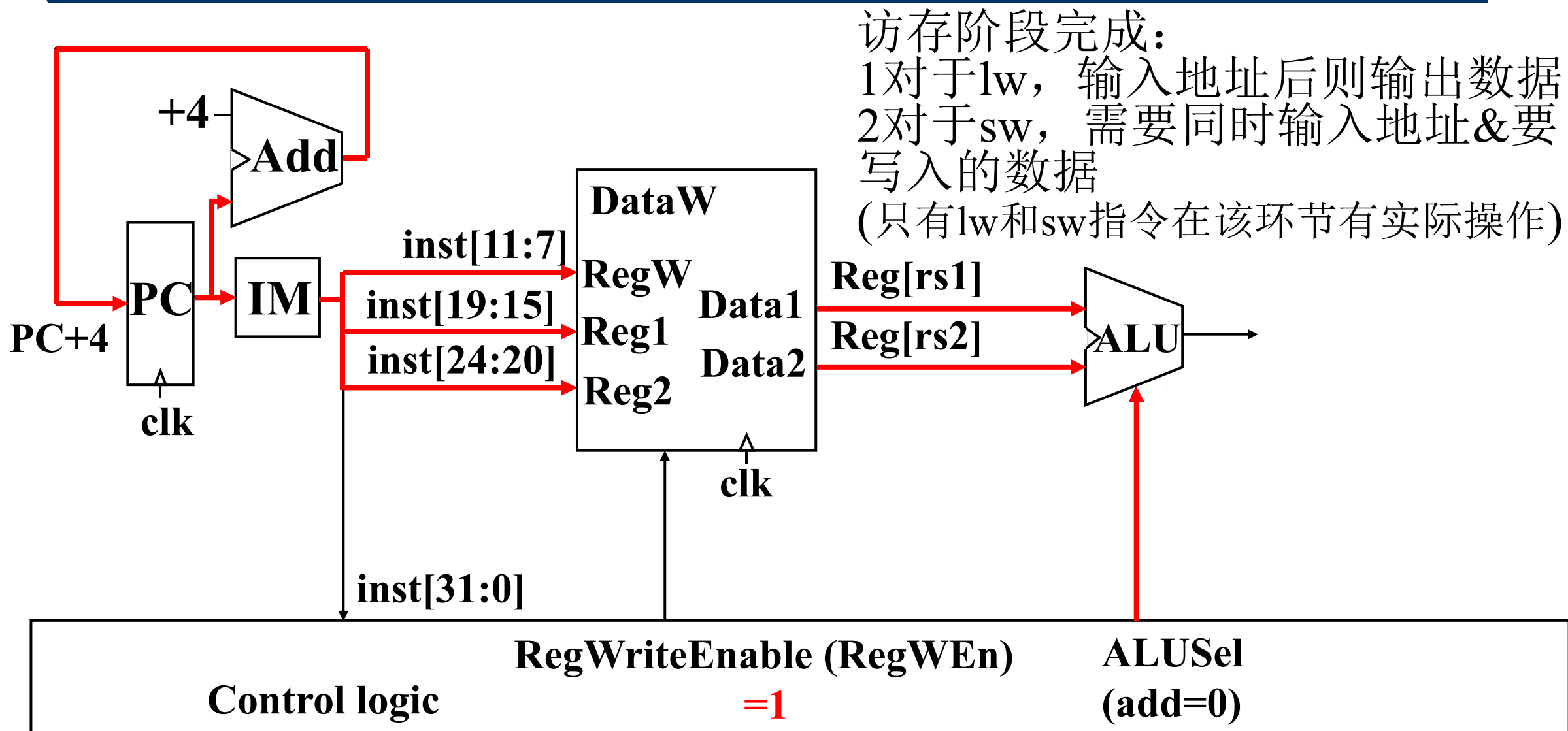
add指令的数据通路



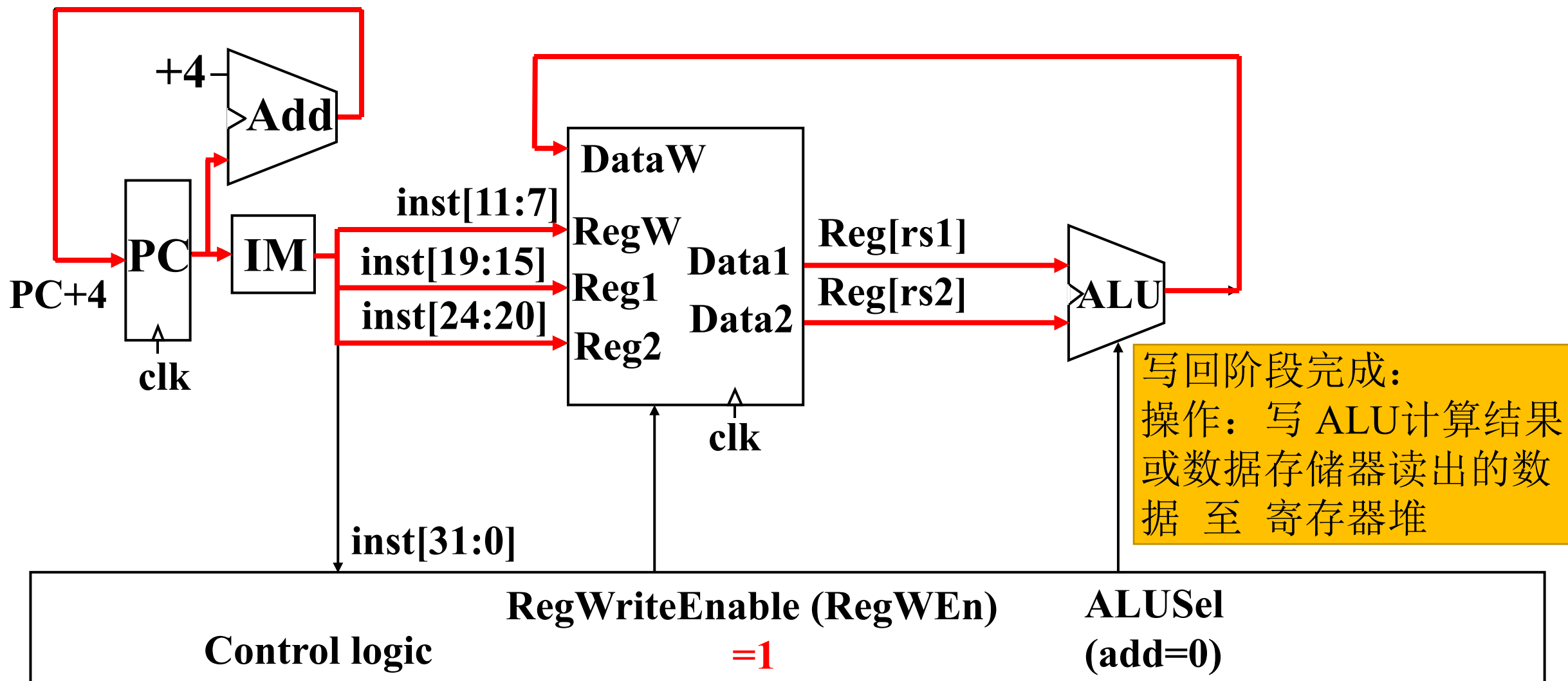
add指令的数据通路



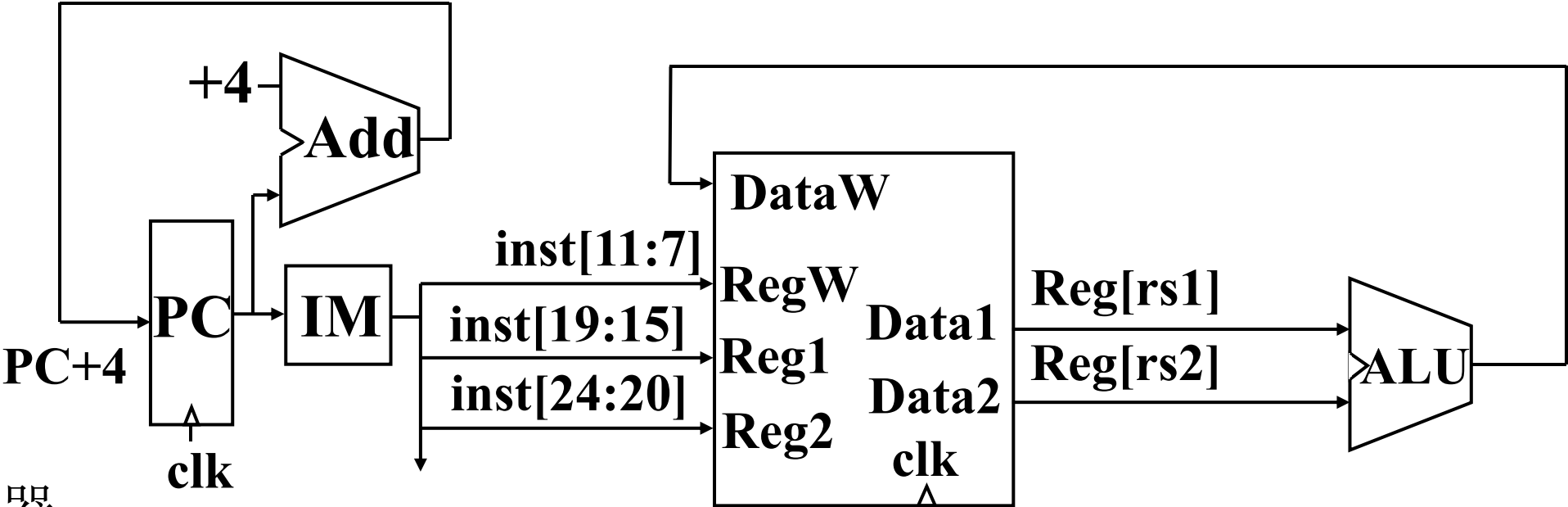
add指令的数据通路



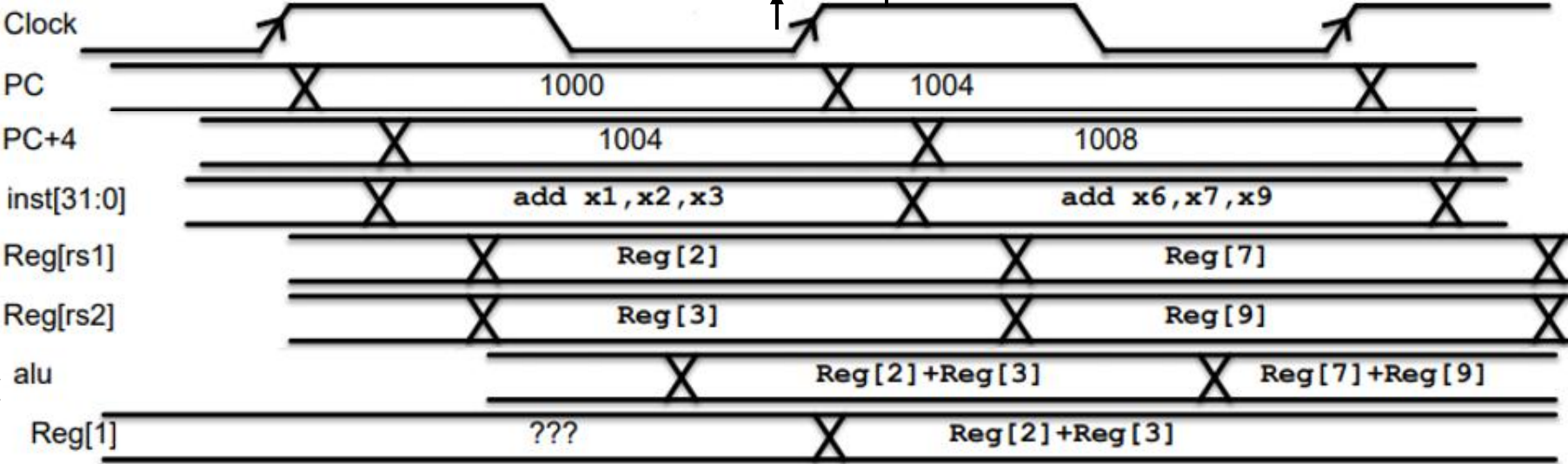
add指令的数据通路



add指令的时序图

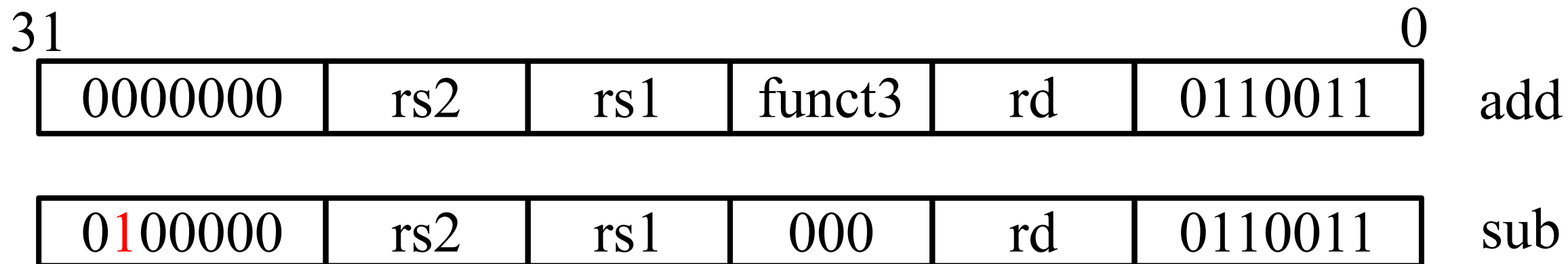


PC寄存器的时延
指令存储器的时延
ALU时延
流经的选择器的时延



max(寄存器文件时延, 立即数生成单元时延)

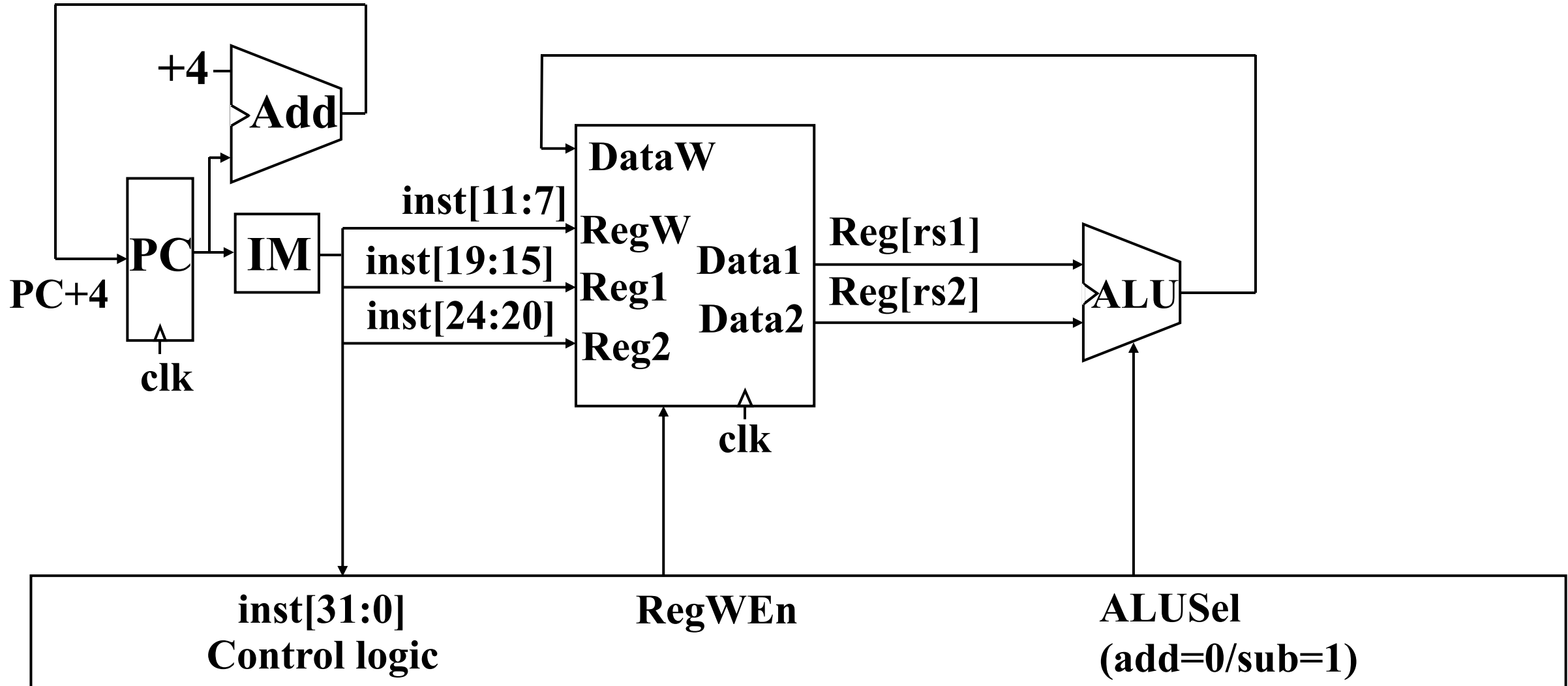
实现sub指令



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



实现R型指令

0000000	rs2	rs1	000	rd	0110011	add
0100000	rs2	rs1	000	rd	0110011	sub
0000000	rs2	rs1	001	rd	0110011	sll
0000000	rs2	rs1	010	rd	0110011	slt
0000000	rs2	rs1	011	rd	0110011	sltu
0000000	rs2	rs1	100	rd	0110011	xor
0000000	rs2	rs1	101	rd	0110011	srl
0100000	rs2	rs1	101	rd	0110011	sra
0000000	rs2	rs1	110	rd	0110011	or
0000000	rs2	rs1	111	rd	0110011	and

funct3和funct7字段不同，选择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

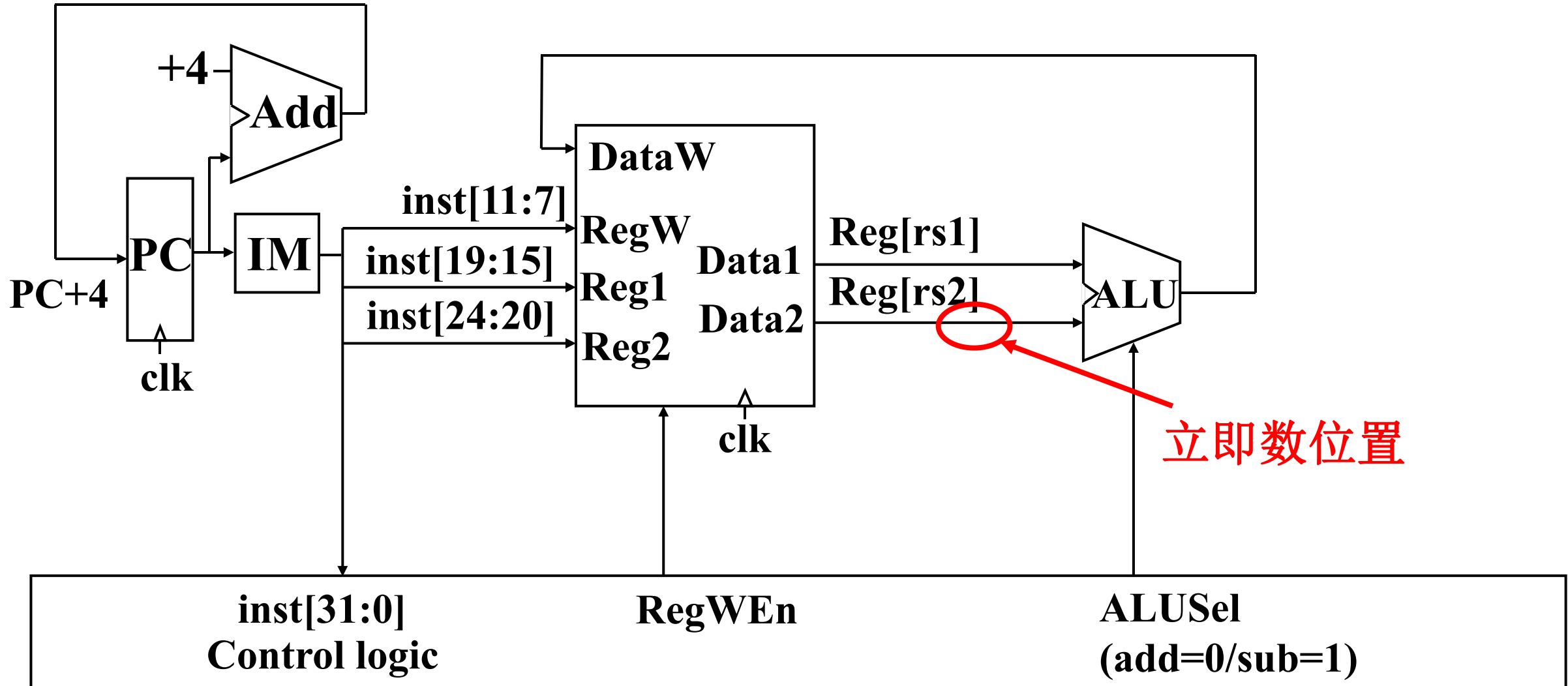
rs1=1

add

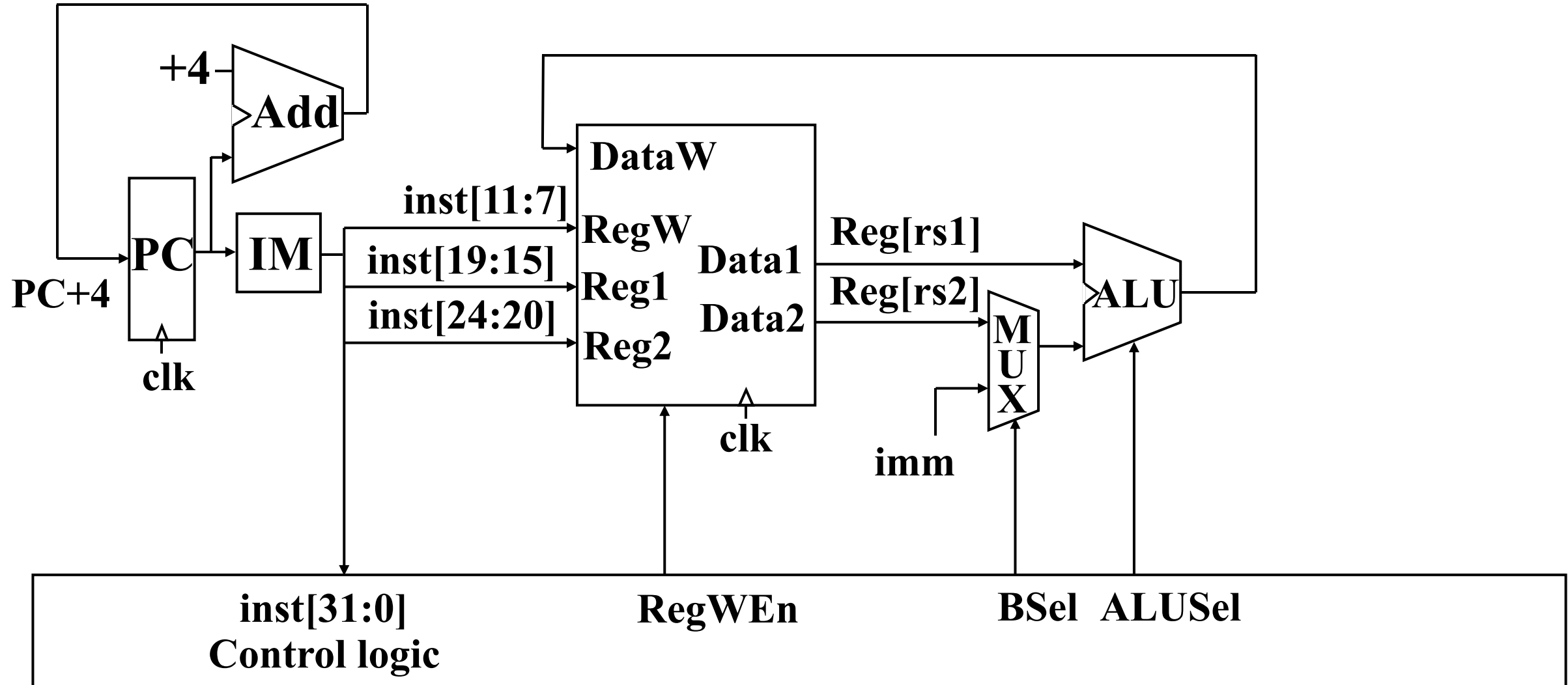
rd=15

OP-Imm

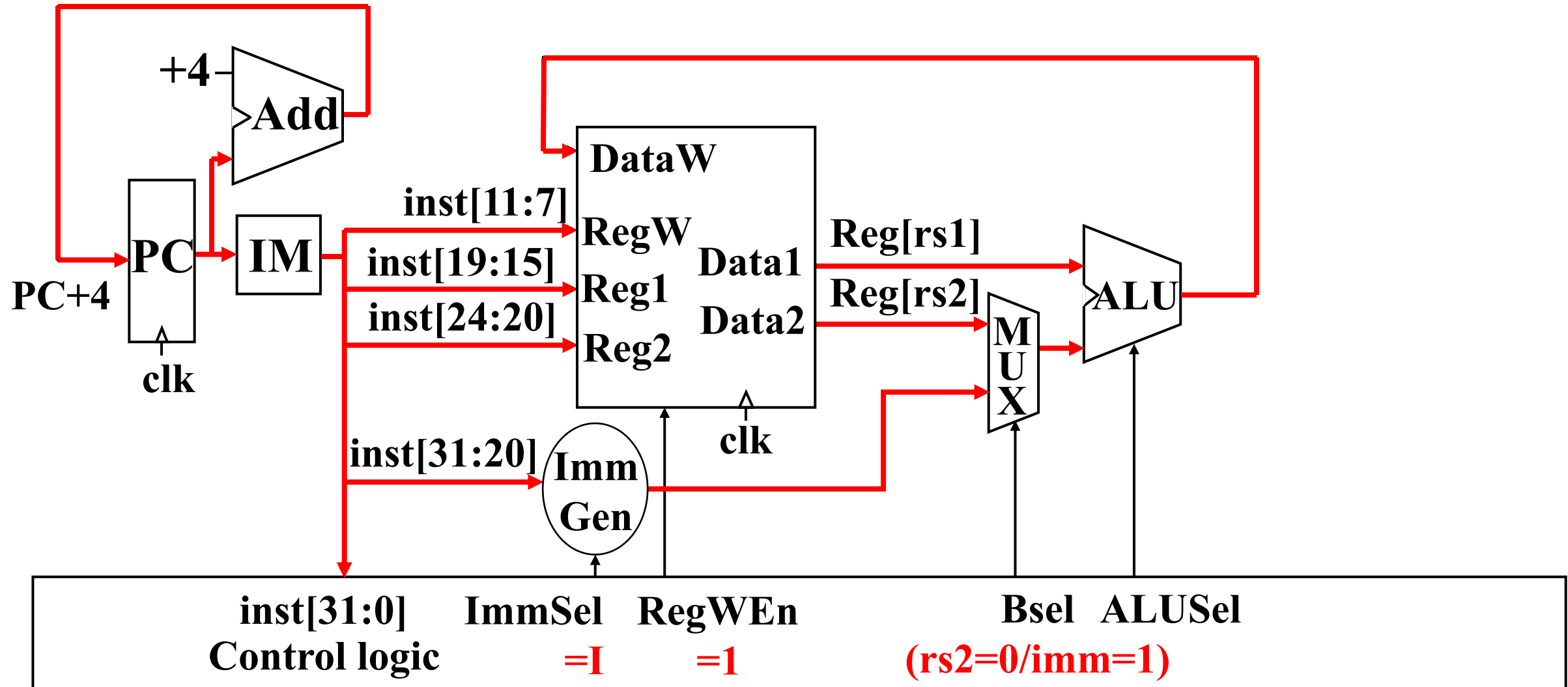
add指令的数据通路



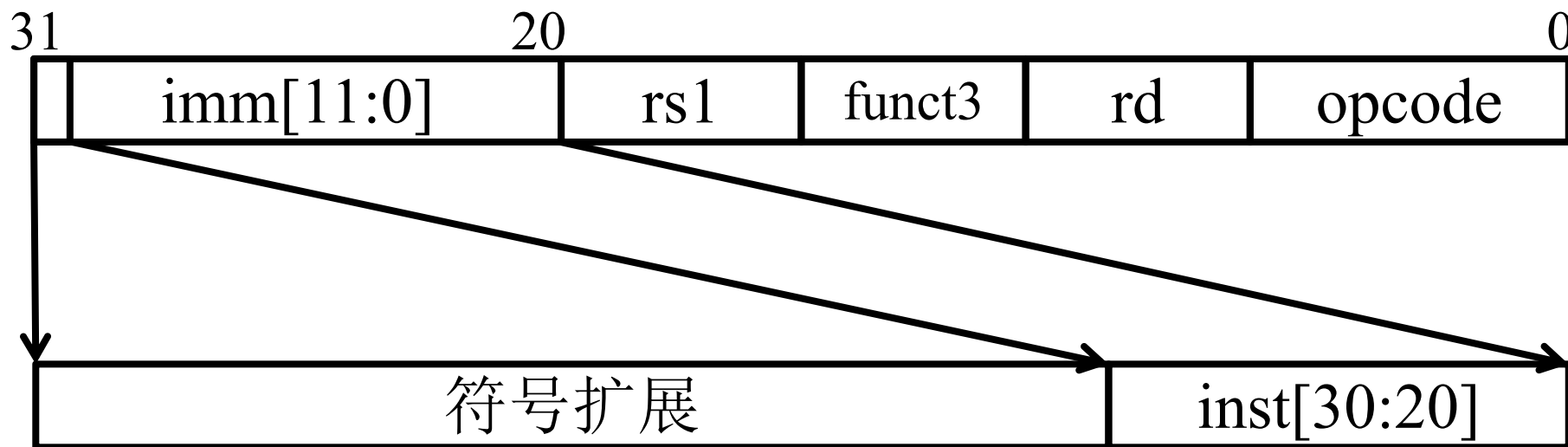
addi指令的数据通路



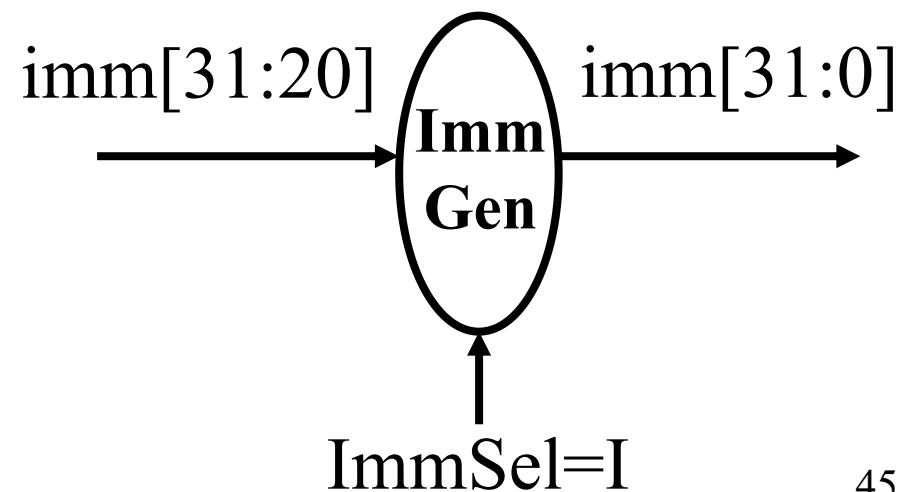
addi指令的数据通路



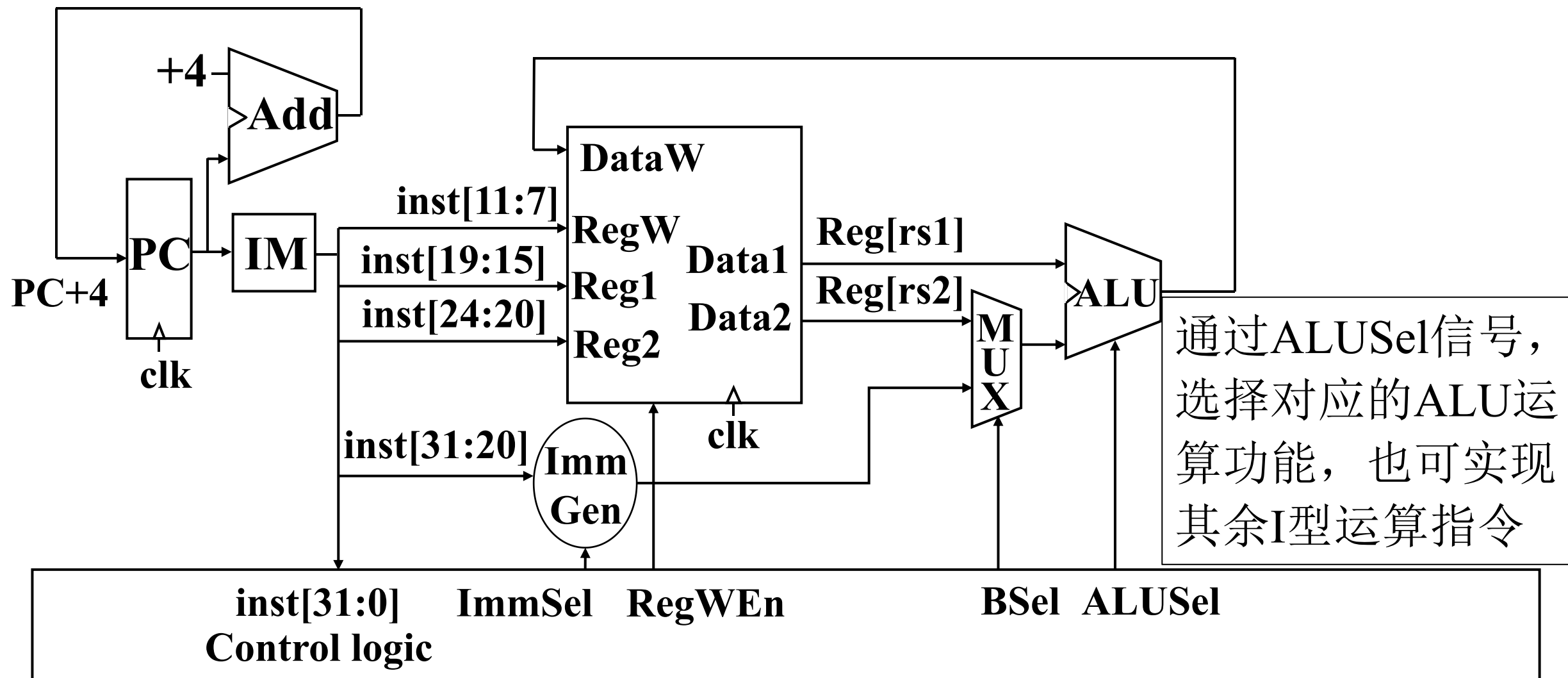
I型指令的立即数生成



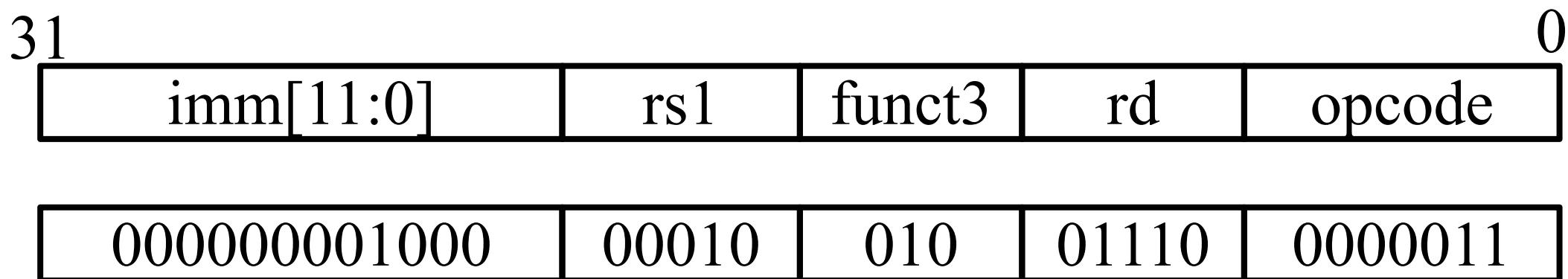
- 高12位复制到立即数的低12位
- 通过将指令的最高比特位复制填充到立即数的高20位完成符号扩展



R型和I型共用的数据通路



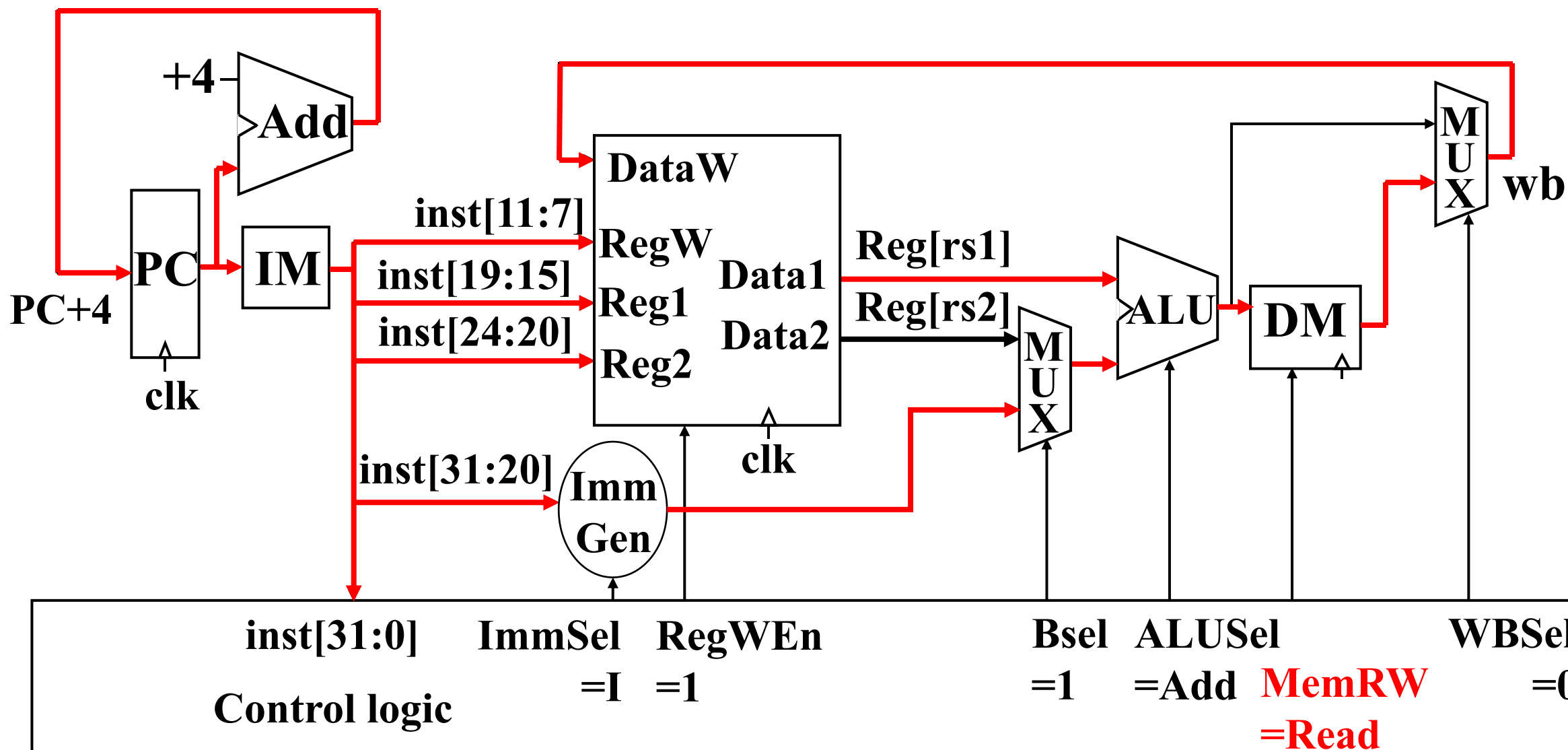
实现lw指令 (load word)



lw 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

lw指令的数据通路



RISC-V 访存装载指令

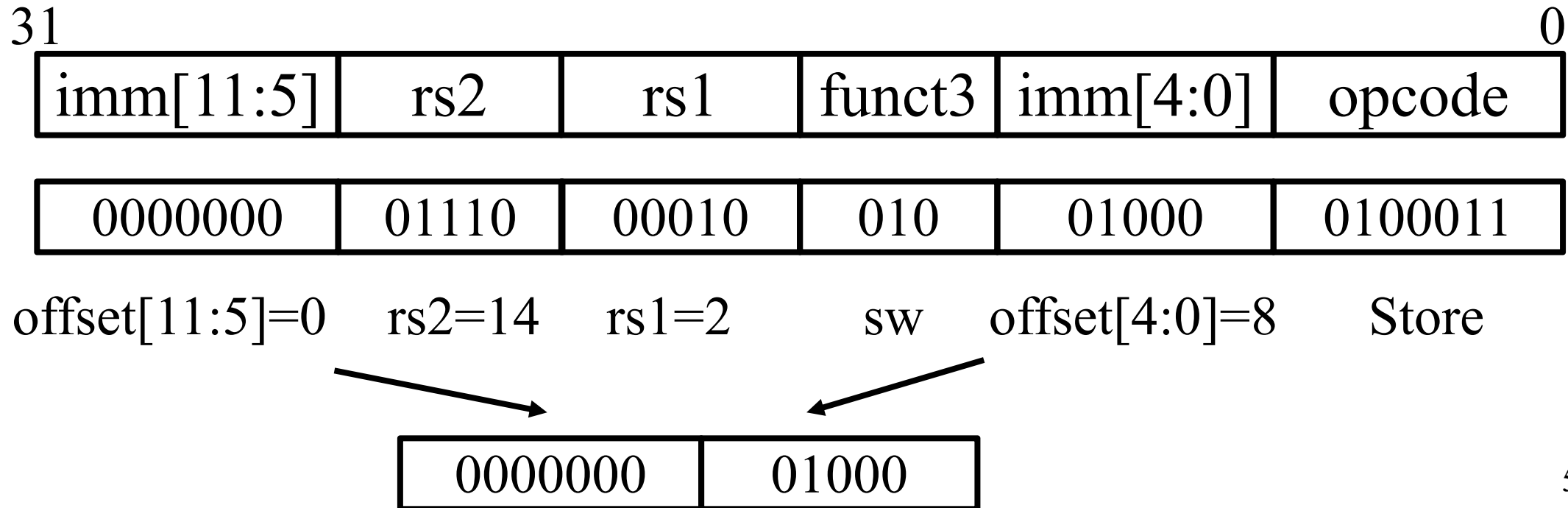
imm[11:0]	rs1	000	rd	0000011	lb
imm[11:0]	rs1	010	rd	0000011	lh
imm[11:0]	rs1	011	rd	0000011	lw
imm[11:0]	rs1	100	rd	0000011	lbu
imm[11:0]	rs1	110	rd	0000011	lhu

- 为了支持8位和16位的访存装载指令
 - 增加额外的逻辑电路，用于从存储器取出的字数据中提取出半字或者字节数据
 - 写回到寄存器前，进行符号扩展或零扩展

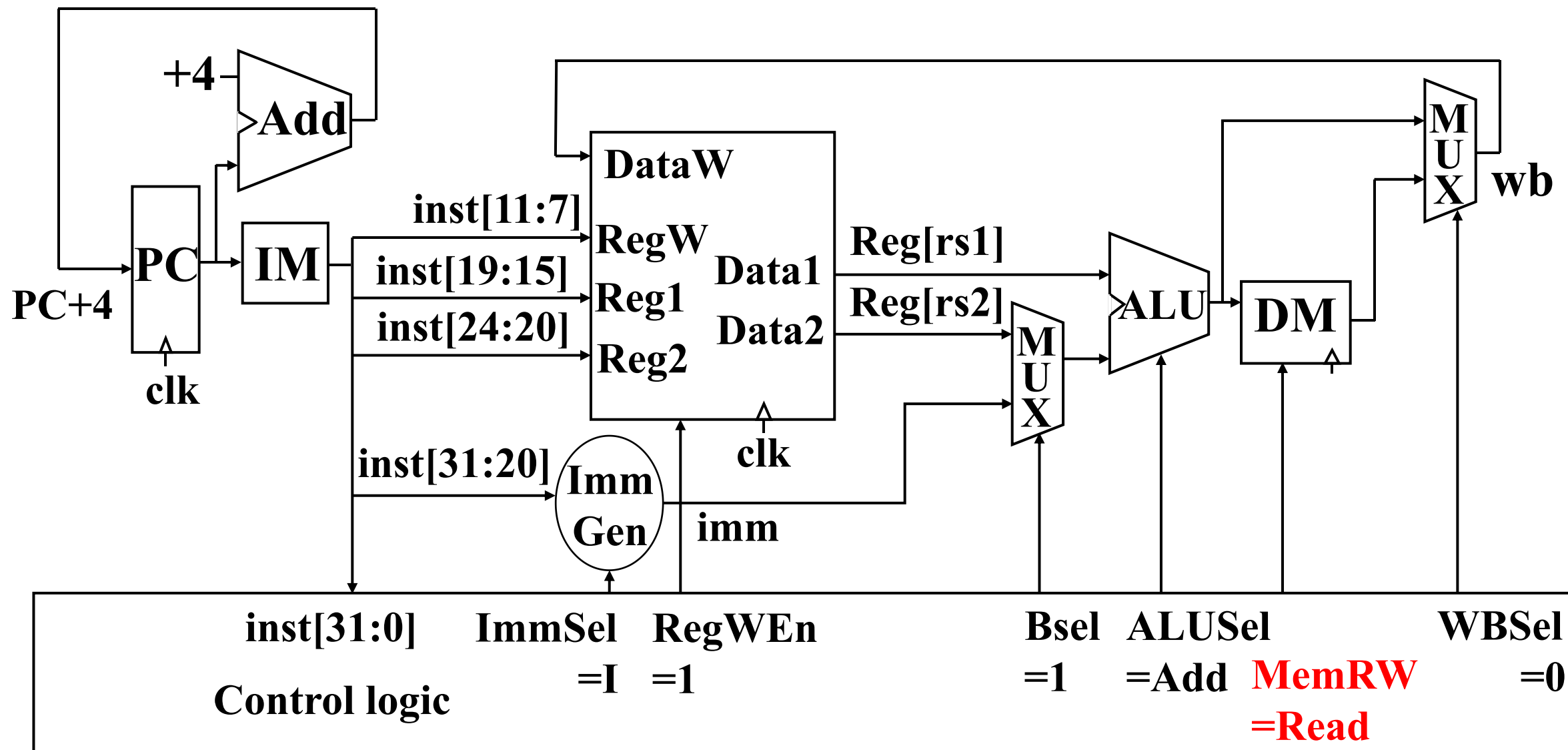
实现sw指令

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

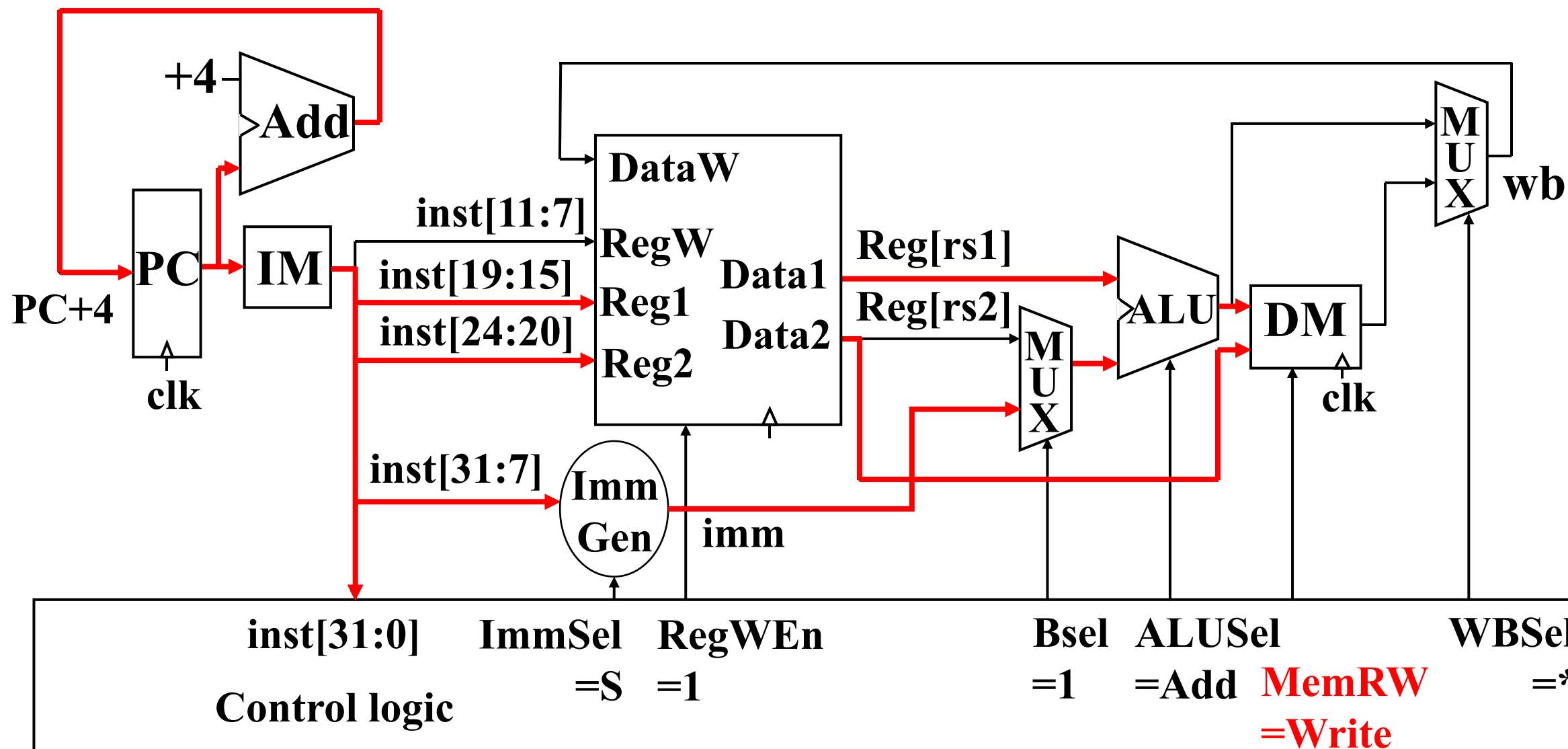
sw x14, 8(x2)



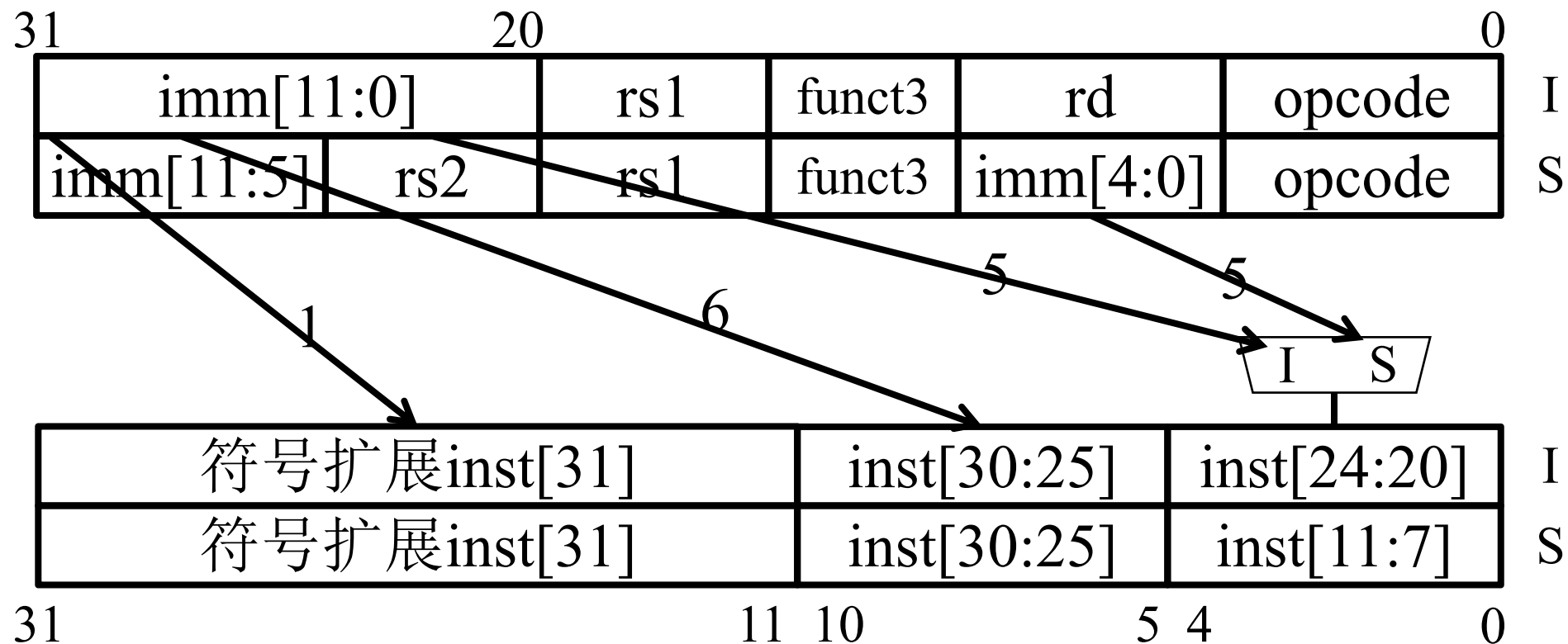
lw指令的数据通路



sw指令的数据通路

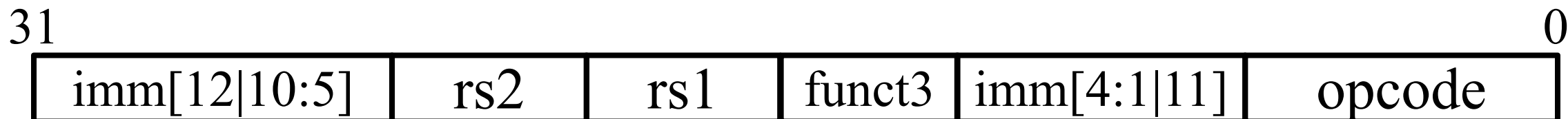


I型指令和S型指令中的立即数生成



- 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

实现B型指令

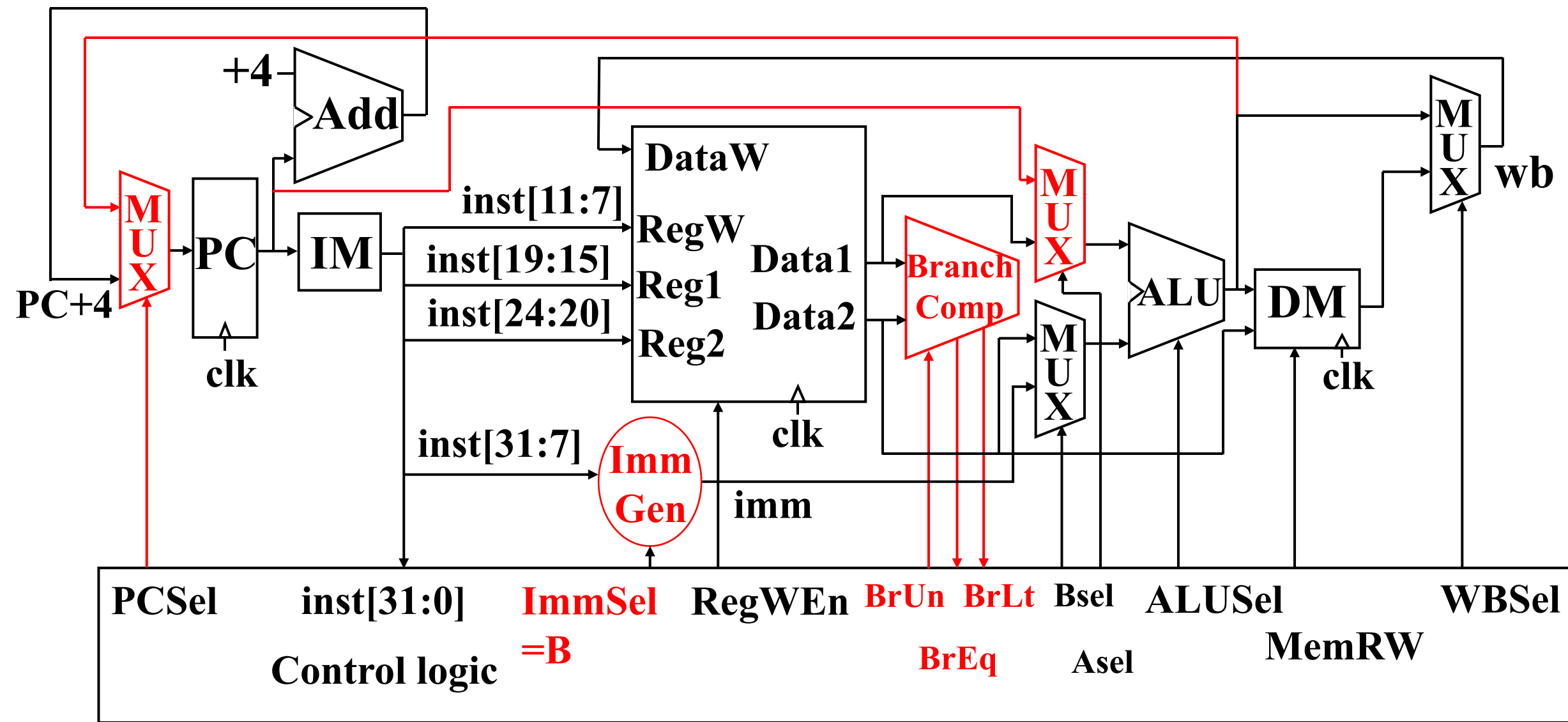


- B型指令格式与S型指令格式基本相同
- 但立即数字段以2字节为增量表示-4096到+4094的偏移量
- 12位立即数字段表示13位有符号字节地址的偏移量
 - 偏移量的最低位始终为零，因此无需存储

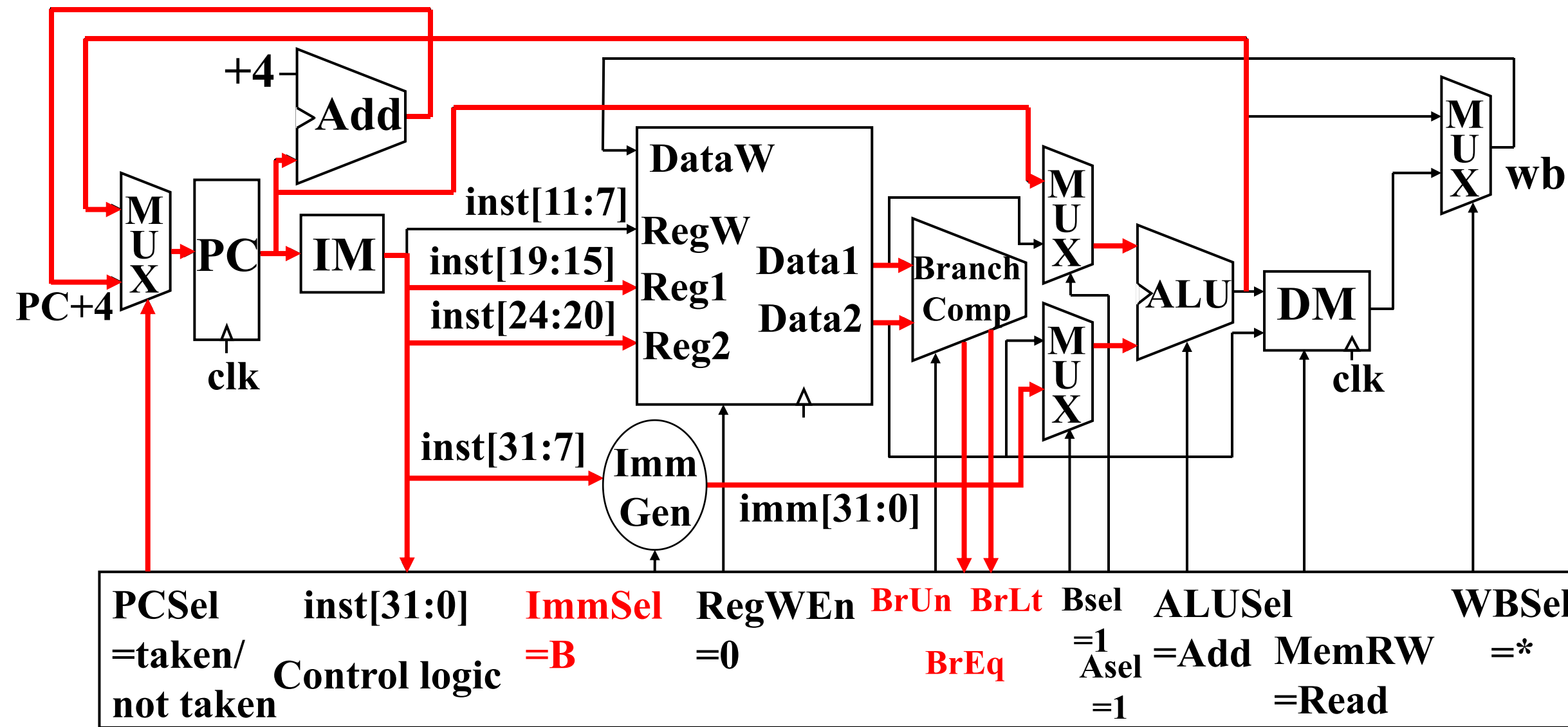
B型指令的数据通路

- 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
- 只有一个ALU
- 需要更多硬件

B型指令的数据通路

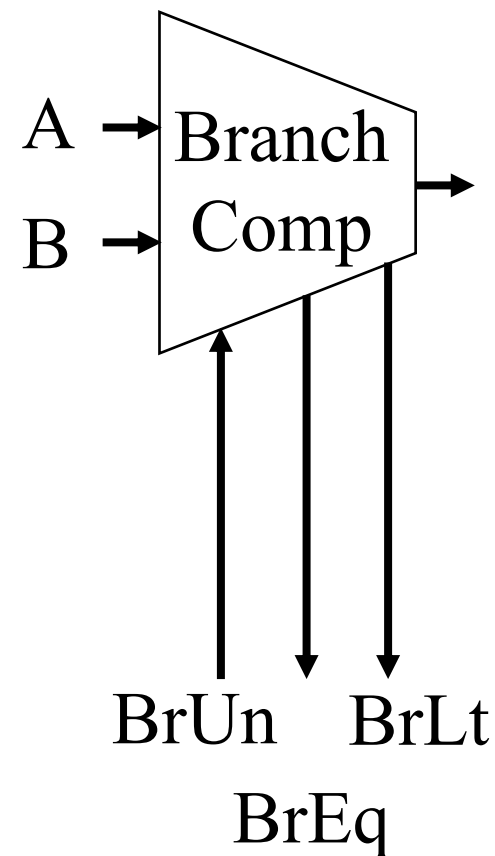


B型指令的数据通路



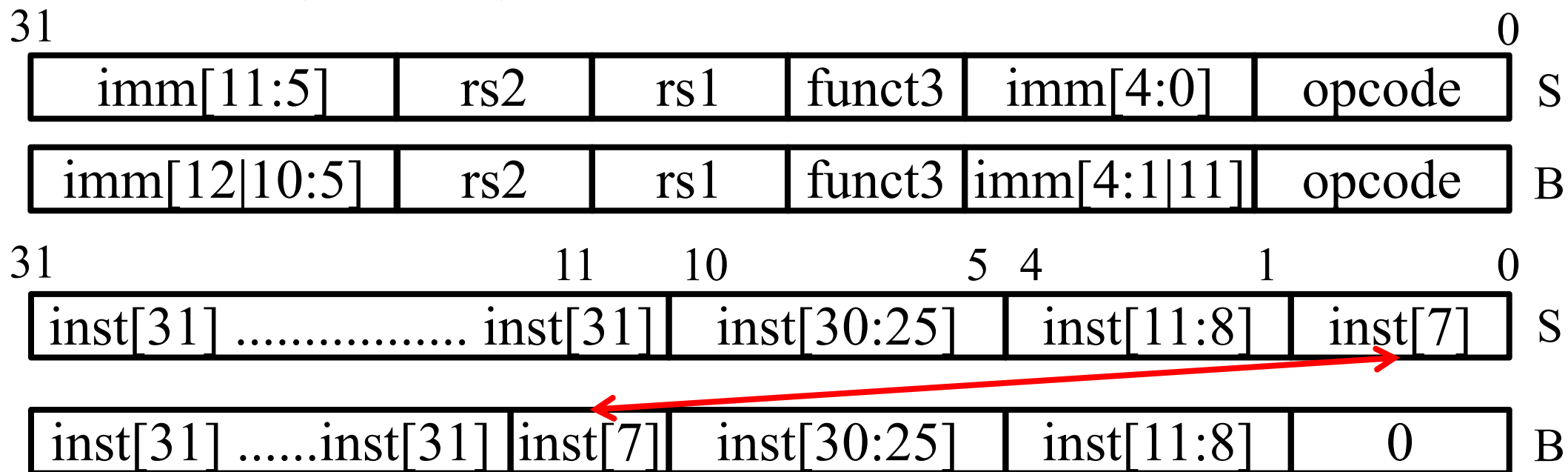
分支跳转比较器

- 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，选择无符号比较结果
输入 $BrUn = 0$ 时，选择有符号比较结果
- 对于bge，可以根据 $BrLt$ 信号取反判断 $A \geq B$

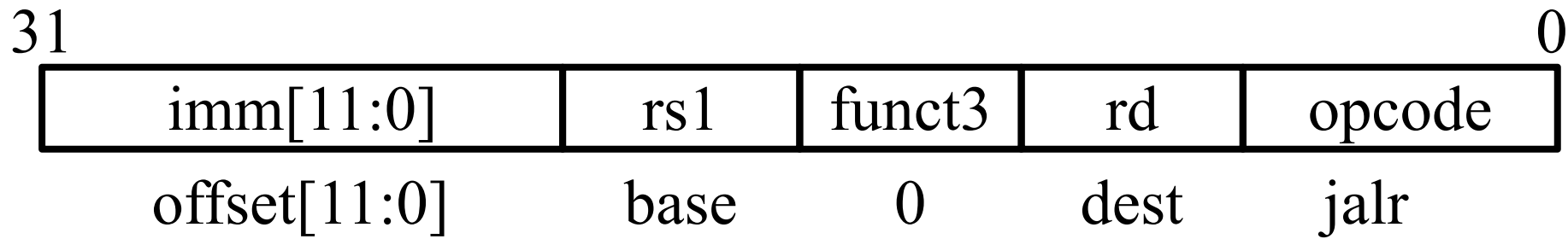


B型指令立即数生成

- 与S型指令立即数生成方式类似，除了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



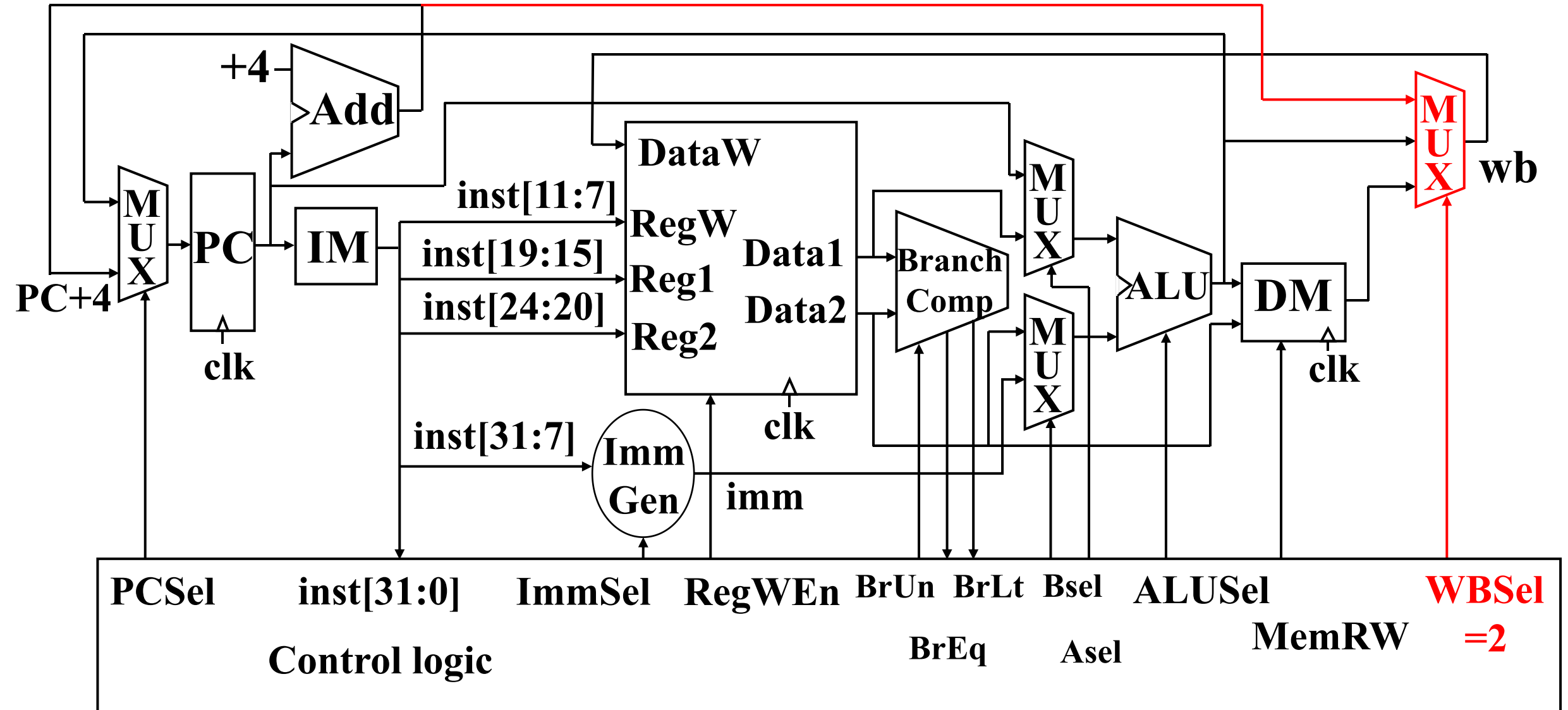
实现jalr指令



jalr rd, rs, immediate

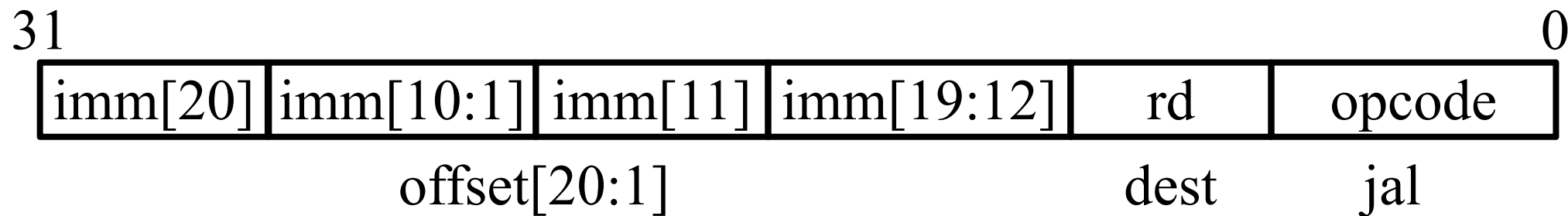
- 两个状态变化
 - 将 $PC + 4$ 写入rd（返回地址）
 - 设置 $PC = rs + immediate$
 - 立即数生成与I型算术和装载指令一样

jalr指令的数据通路



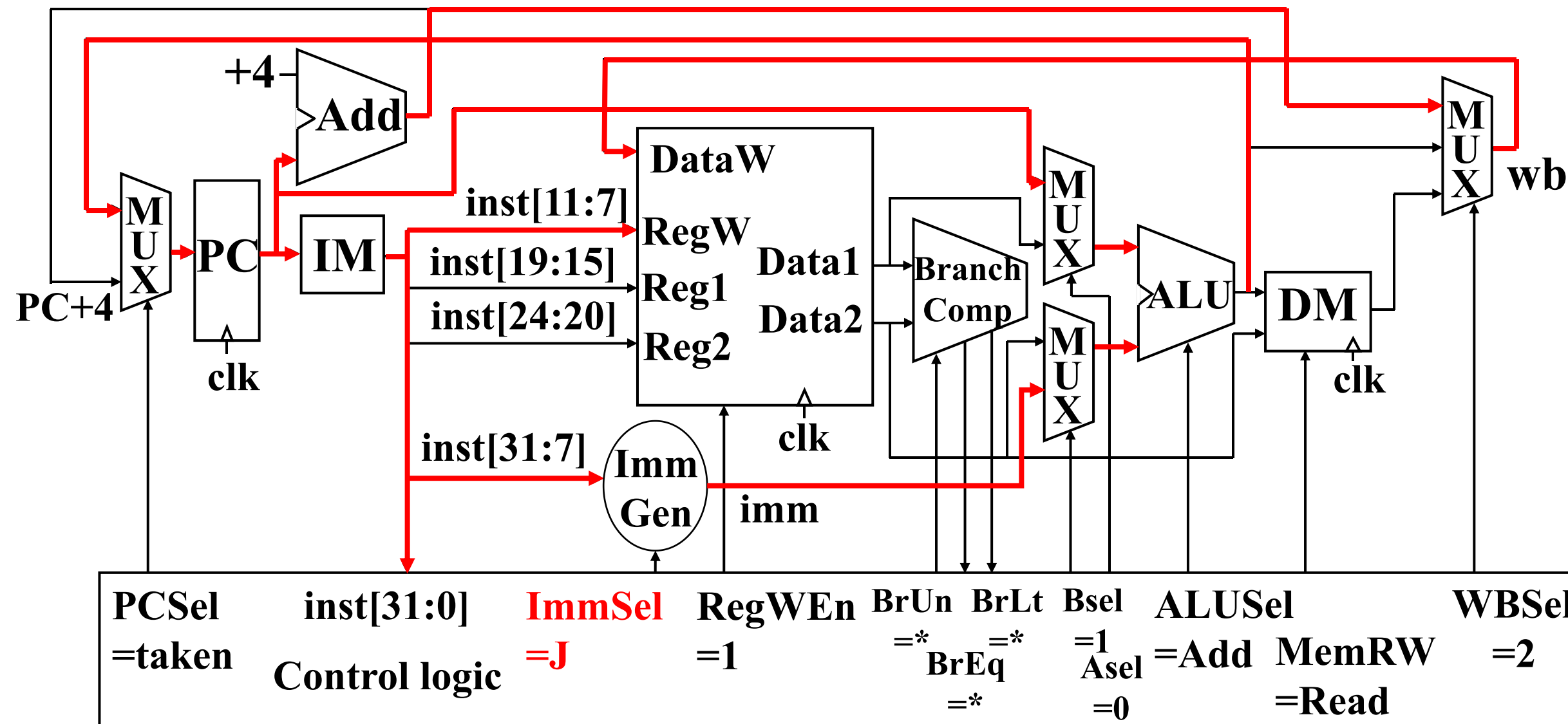


实现jal指令

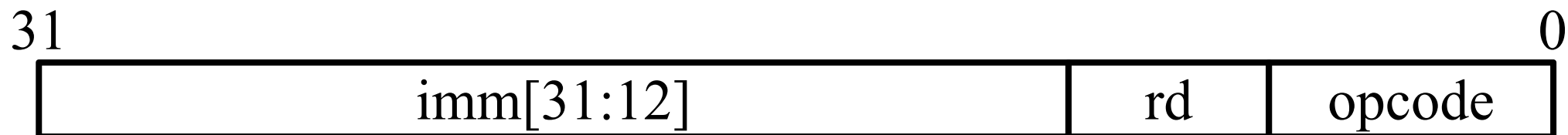


- jal将 $PC + 4$ 写入目的寄存器rd中
- 设置 $PC = PC + offset$
- 立即数字段20位，对应一个 $\pm 2^{19}$ 的偏移量(以2字节为单位)

jal指令的数据通路



U型指令



U-immediate[31:12]

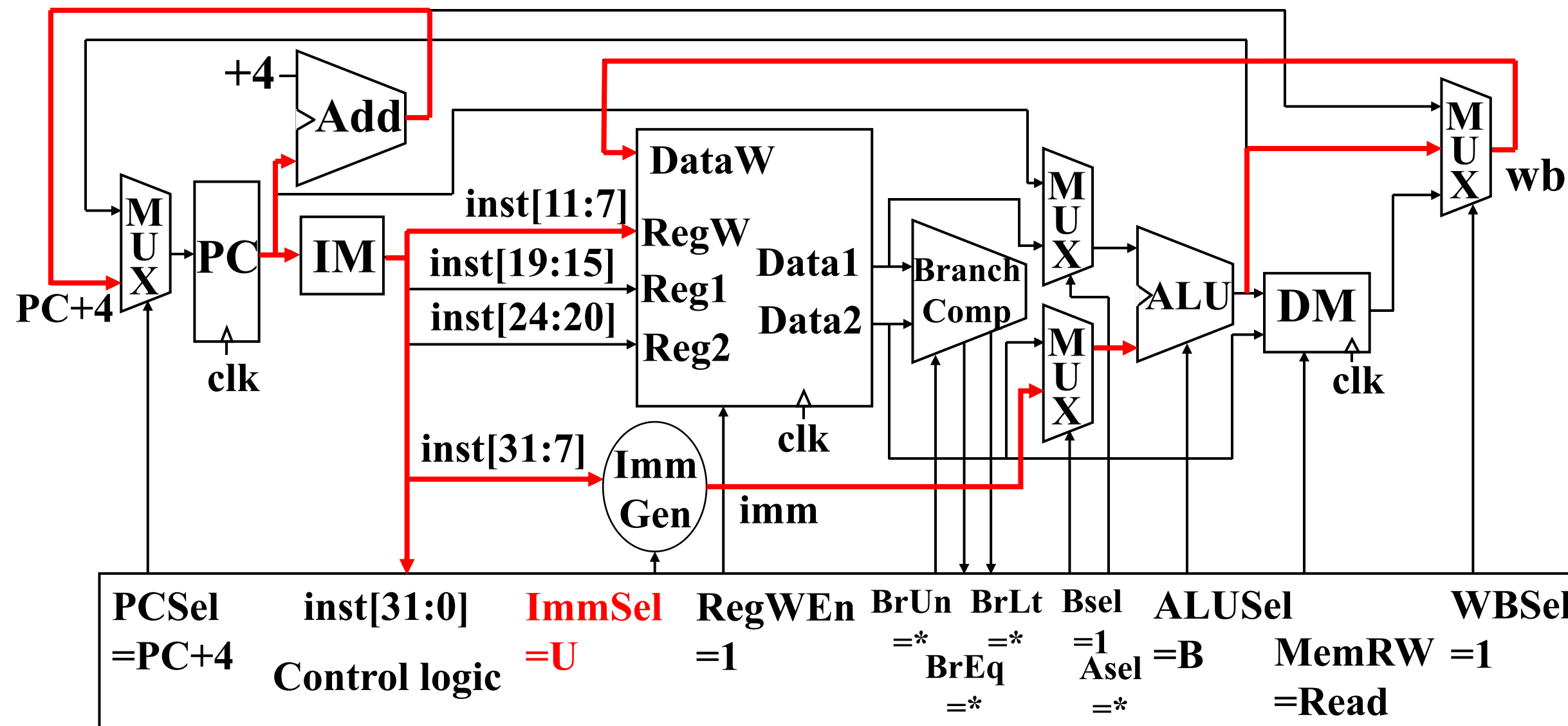
dest

lui

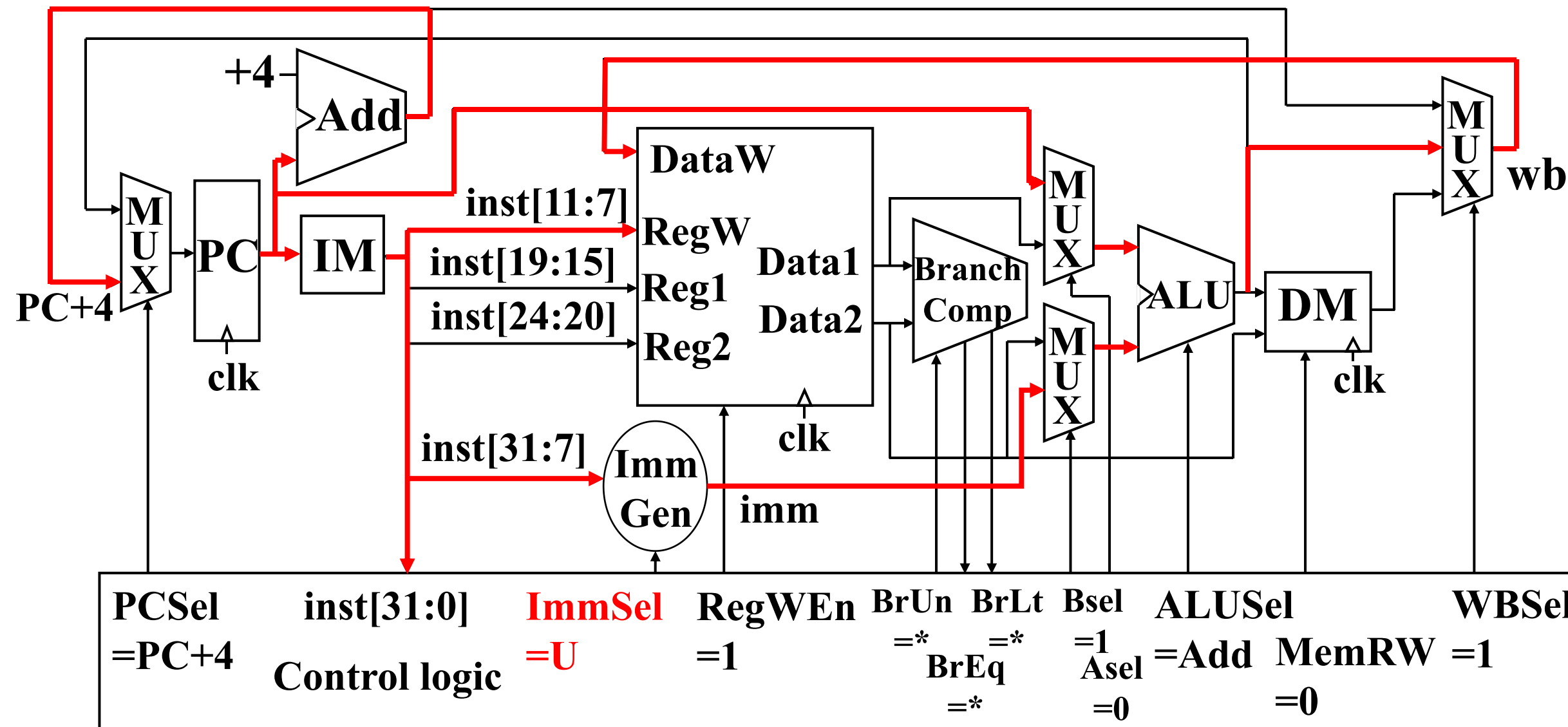
auipc

- 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

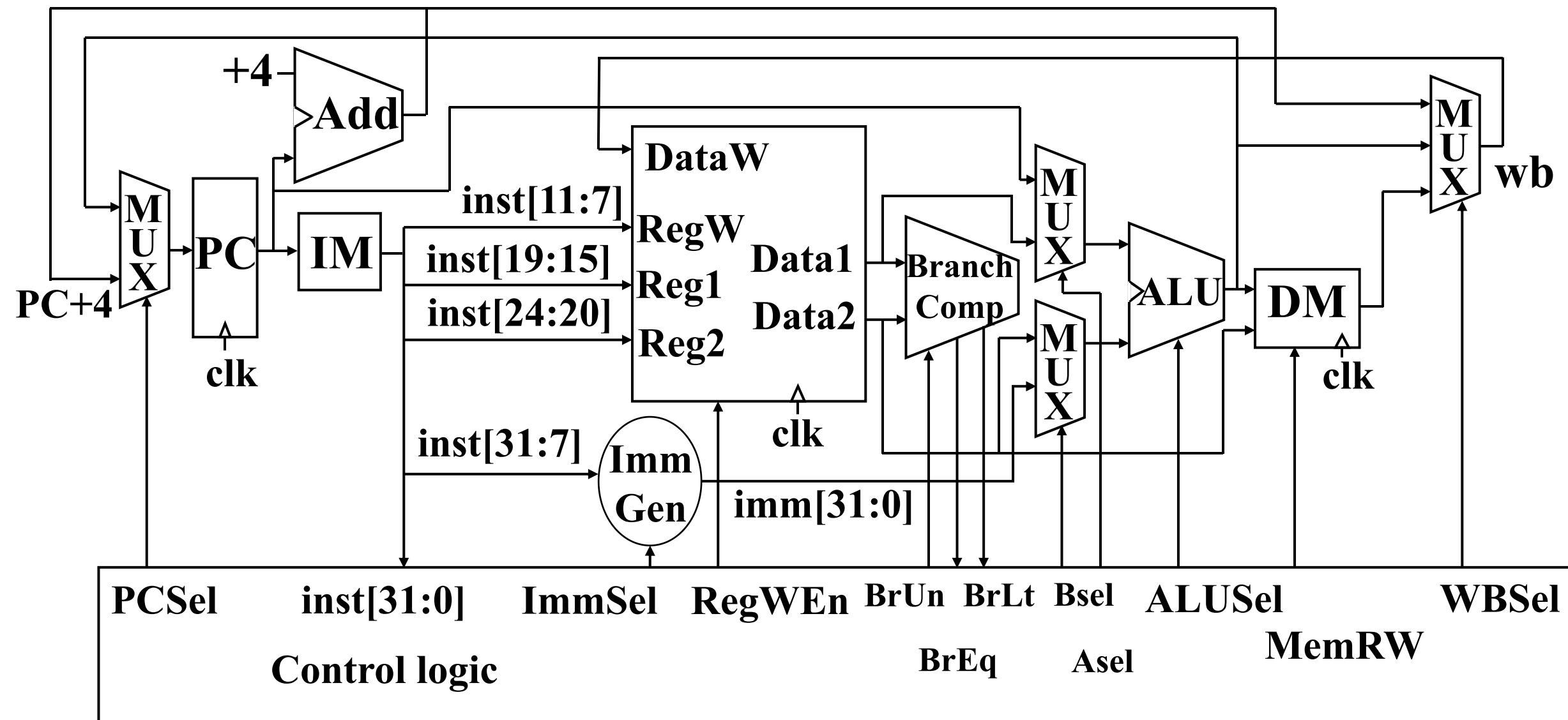
lui指令的数据通路



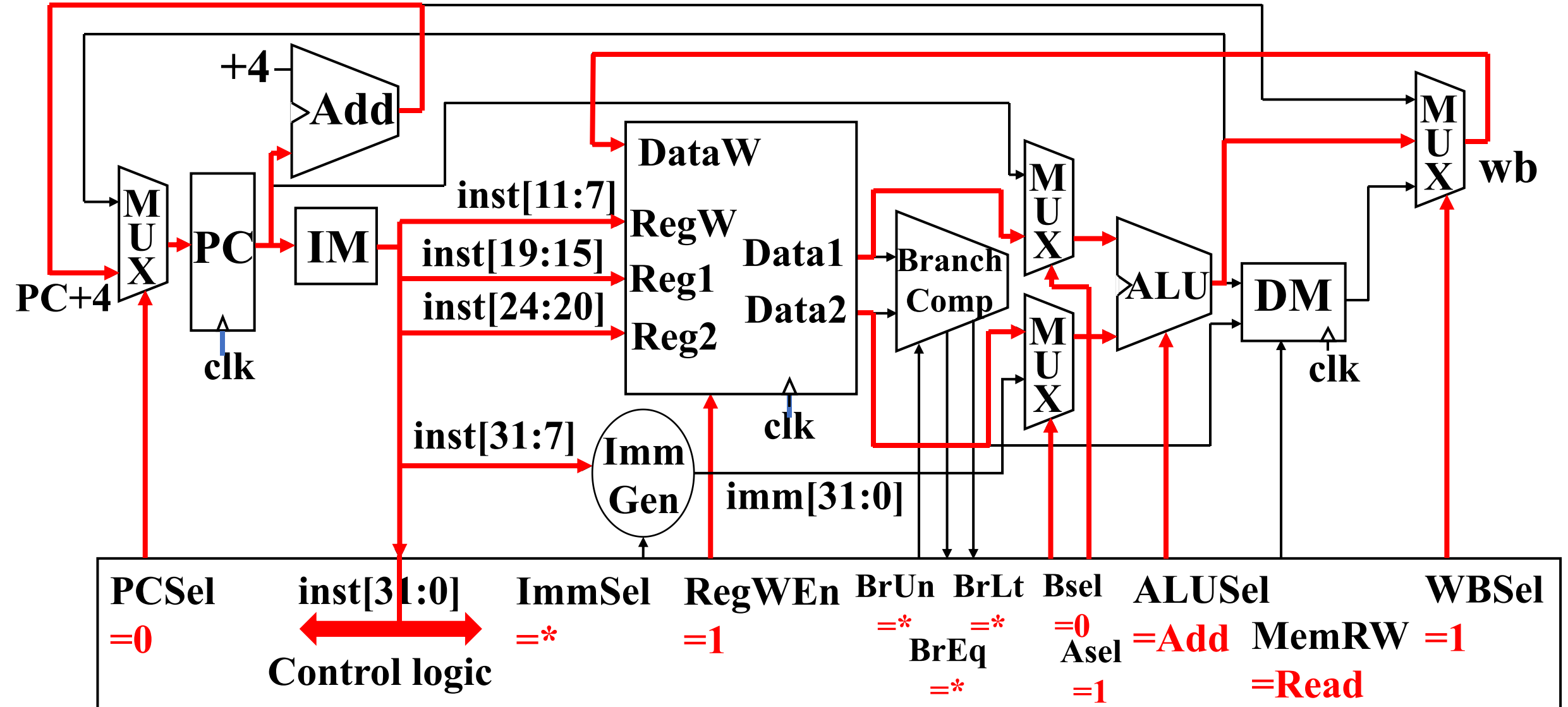
auipc指令的数据通路



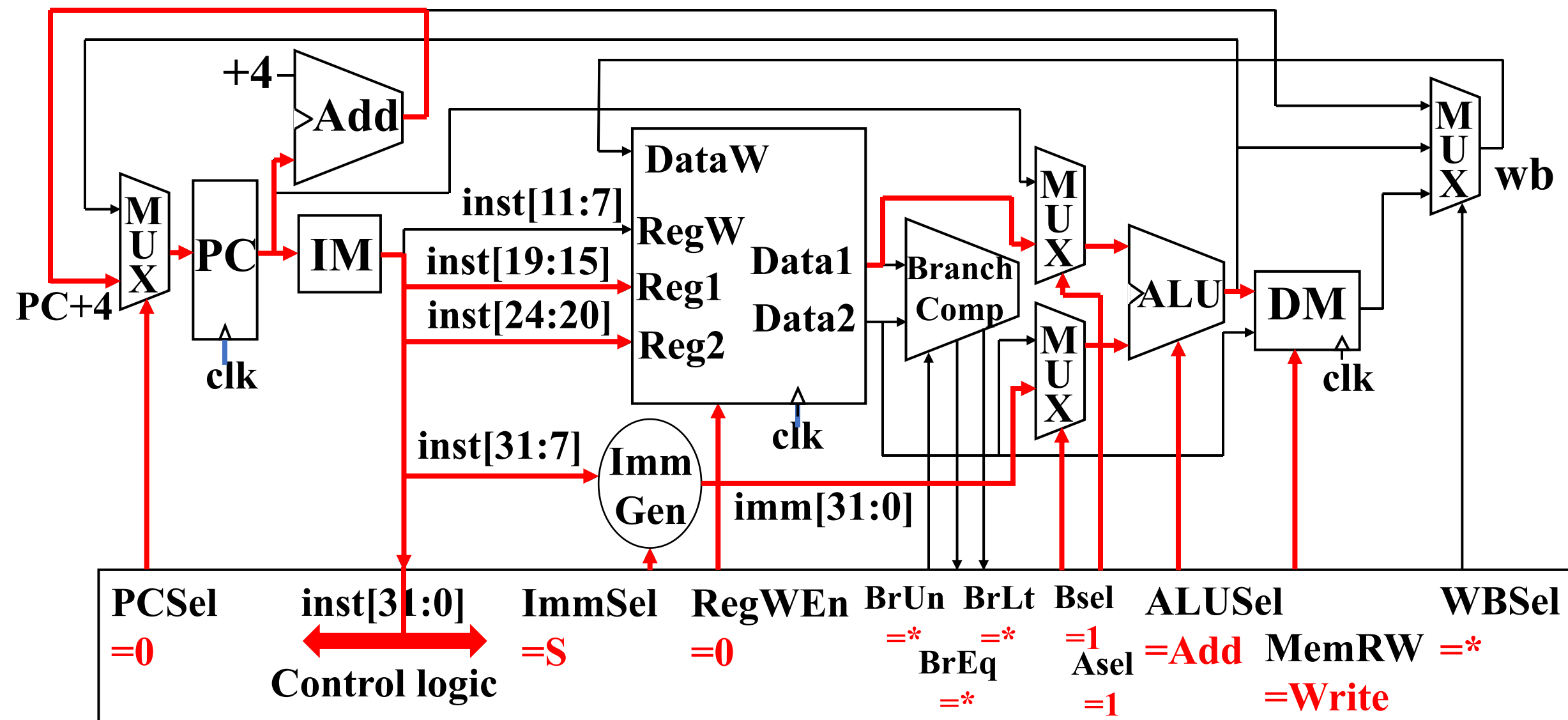
RISC-V 的数据通路



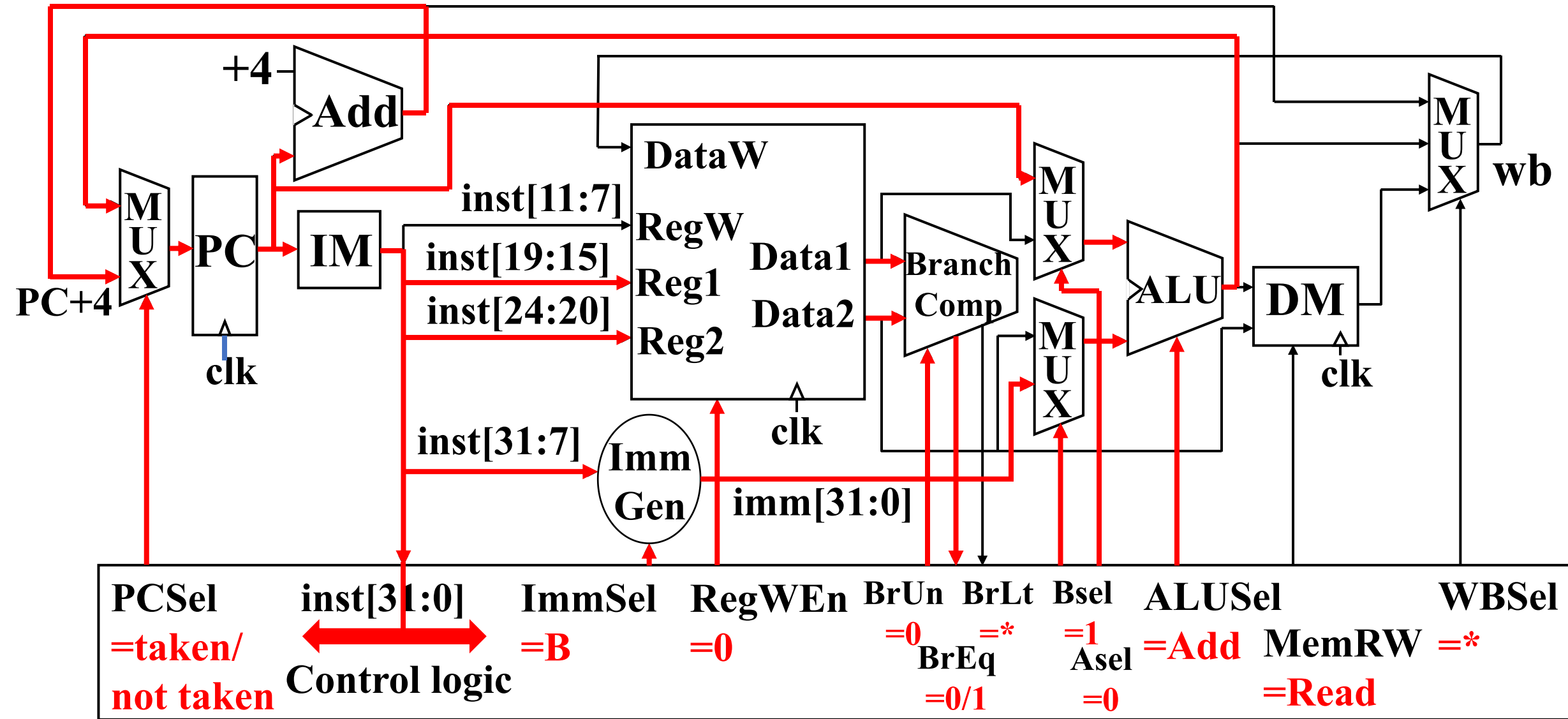
add指令的执行



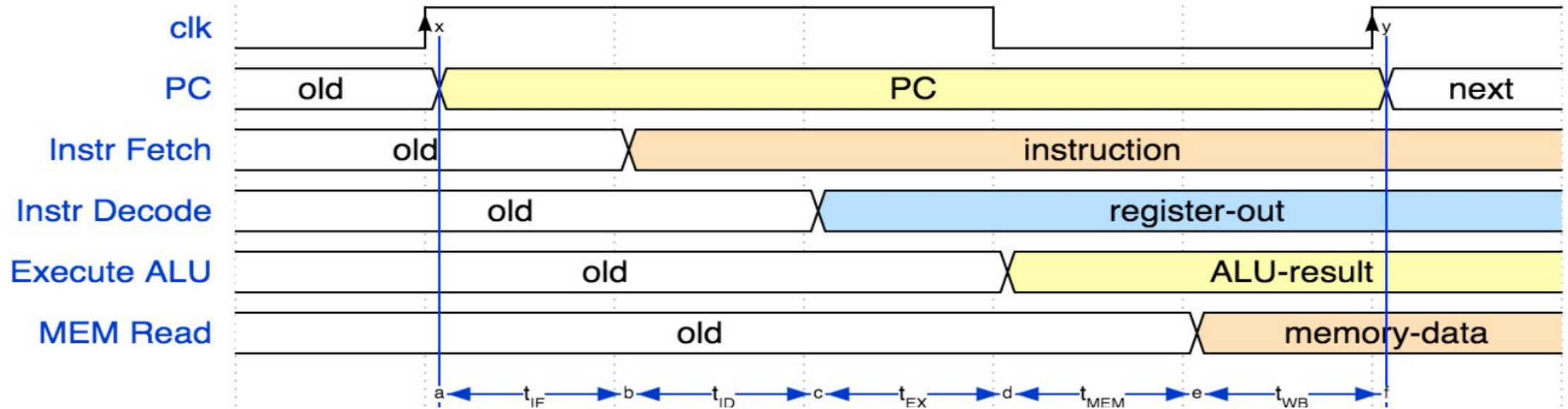
sw指令的执行



beq指令的执行



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

指令时延

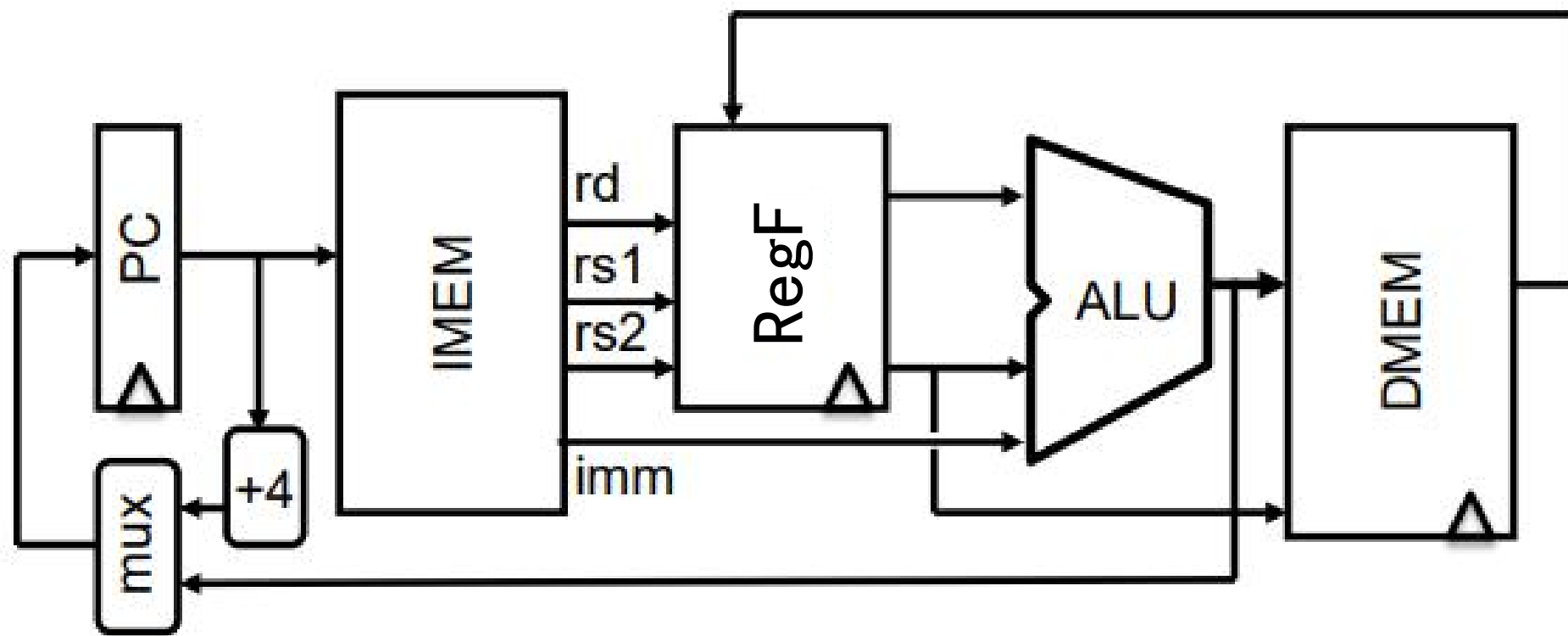
Instr	IF=200ps	ID=100ps	ALU=200ps	MEM=200ps	WB=100ps	Total
add	X	X	X		X	600ps
beq	X	X	X			500ps
jal	X	X	X			500ps
lw	X	X	X	X	X	800ps
sw	X	X	X	X		700ps

最大时钟频率 $f_{\max, \text{ALU}} = 1/(800\text{ps}) = 1.25\text{GHz}$

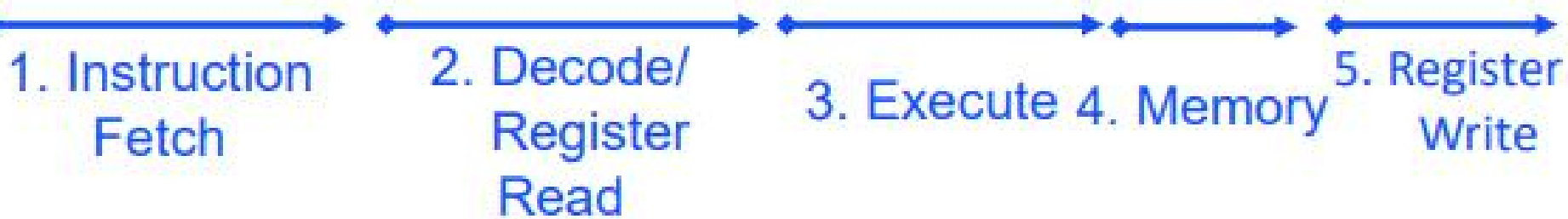
大多数区块大部分时间处于空闲状态

$f_{\max, \text{ALU}} = 1/(200\text{ps}) = 5\text{GHz}$

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



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)

第六章 处理器设计

- RISC-V数据通路
 - 数据通路概念
 - RISC-V部分指令的数据通路
- RISC-V控制器

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]

R 型	funct7	rs2	rs1	funct3	rd	opcode
I 型	imm[11:0]		rs1	funct3	rd	opcode
S 型	Imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
SB / B 型	Imm[12,10:5]	rs2	rs1	funct3	imm[4:1,11]	opcode
UJ / J 型	Imm[20,10:1,11,19:12]				rd	opcode
U 型	Imm[31:12]				rd	opcode

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

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

RISC V控制器实现

- ROM

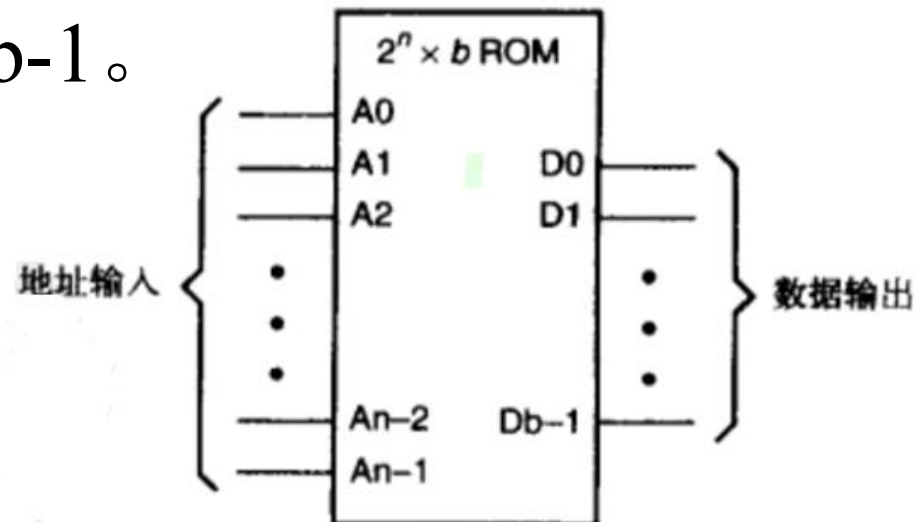
- 便于重新编程
 - 修正错误
 - 添加新的指令
- 在人工设计控制逻辑时十分常用

- 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

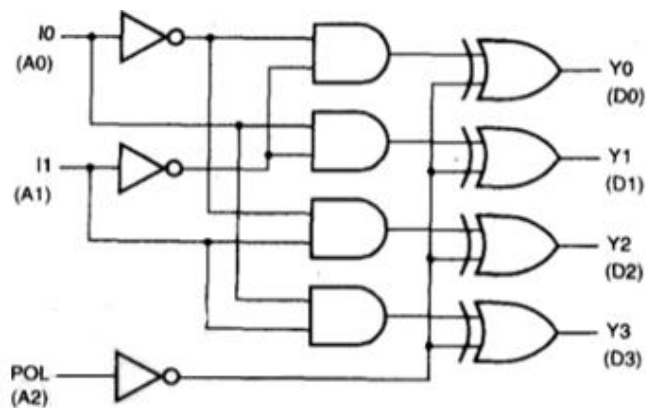
- ROM “存储” 了一个n输入、b输出的组合逻辑功能的真值表。
- 一个3输入、4输出的组合功能的真值表，可以被存储在一个 $2^3 * 4$ ($8 * 4$) 的只读存储器中。忽略延迟，ROM的数据输出总是等于真值表中由地址输入所选择的那行输出位。

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

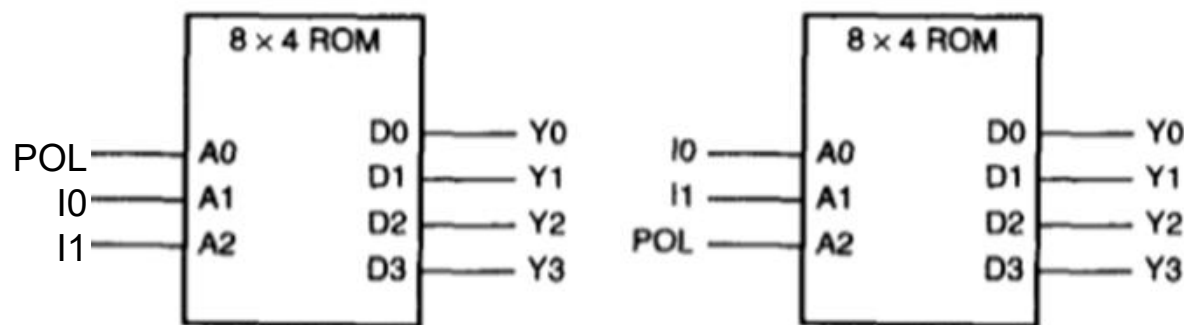
一个3输入4输出组合逻辑函数的真值表
(具有输出极性控制的2-4译码器)

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。



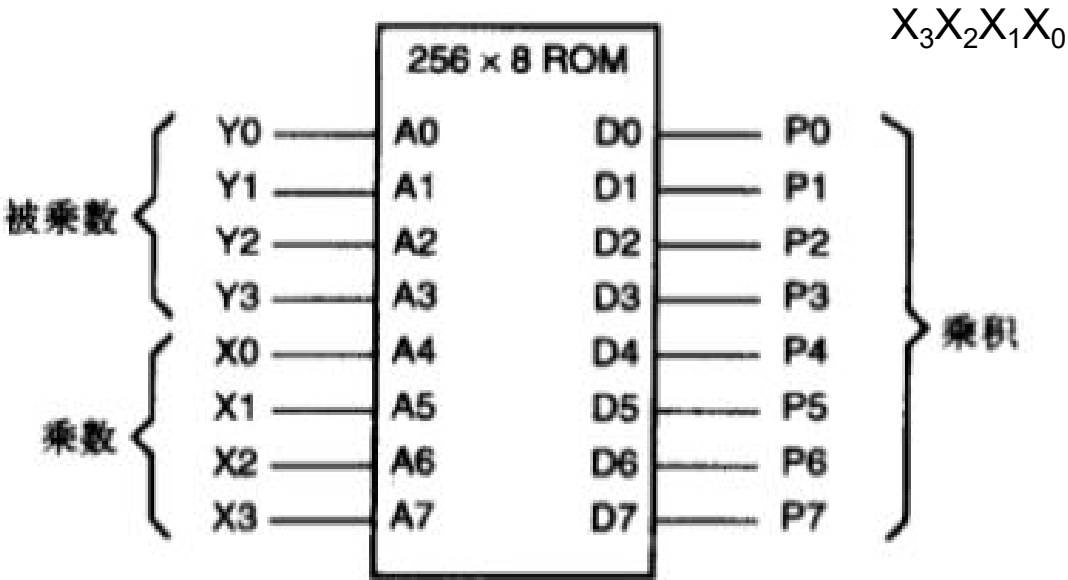
具有输出极性控制的2-4译码器



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

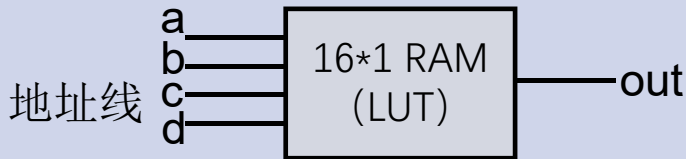
- 多少种组合？
- 乘积最多为几位？



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

FPGA中的查找表（LUT）

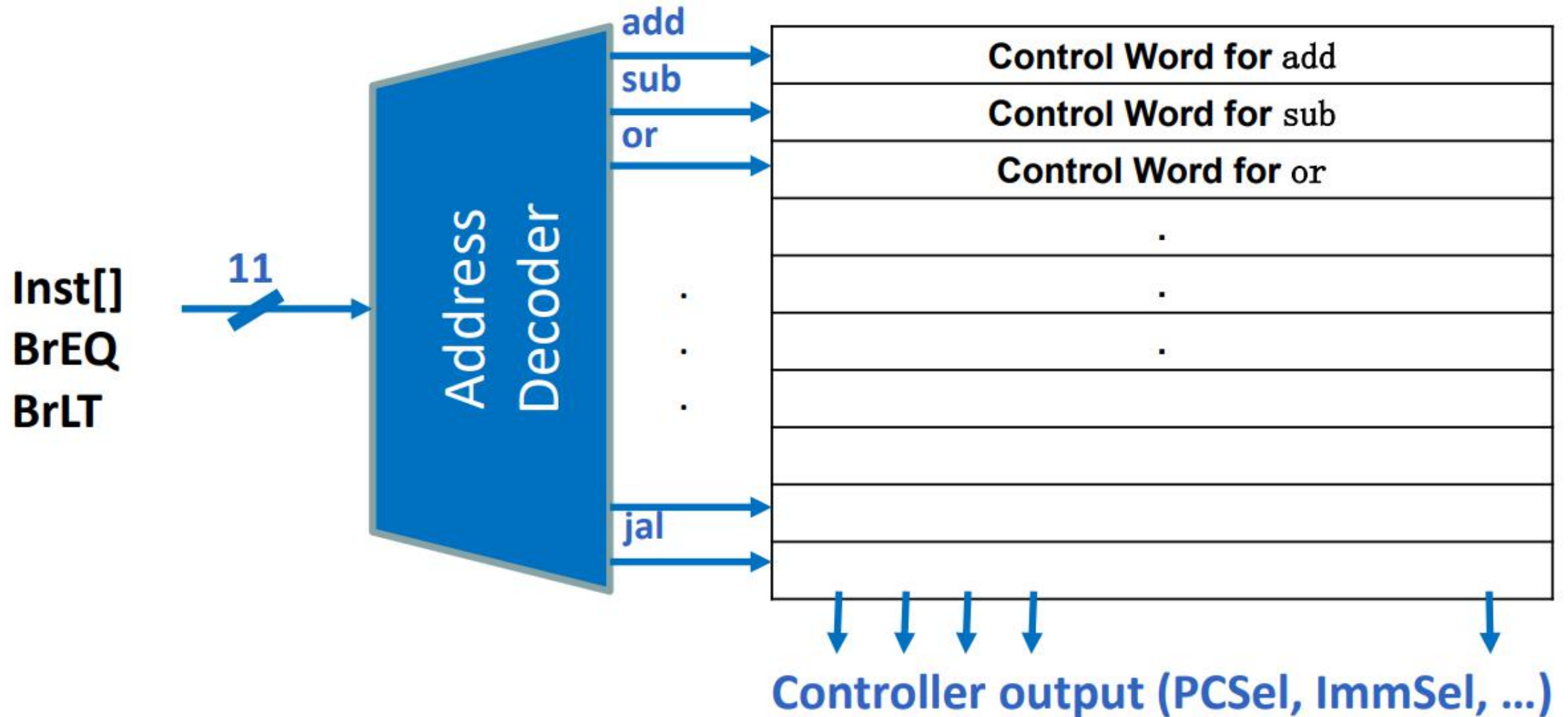
- 例：使用LUT实现一个4与门电路逻辑功能

实际逻辑电路		LUT的实现方式	
			
a、b、c、d	逻辑输出	地址	RAM中存储的内容
0000	0	0000	0
0001	0	0001	0
.....	0	0
1111	1	1111	1

LUT本质就是RAM，主流的FPGA是5输入或6输入LUT

A,B,C,D由FPGA芯片的管脚输入后进入可编程连线，然后作为地址线连到LUT，LUT中已经事先写入了所有可能的逻辑结果，通过地址查找到相应的数据然后输出，这样组合逻辑就实现了。

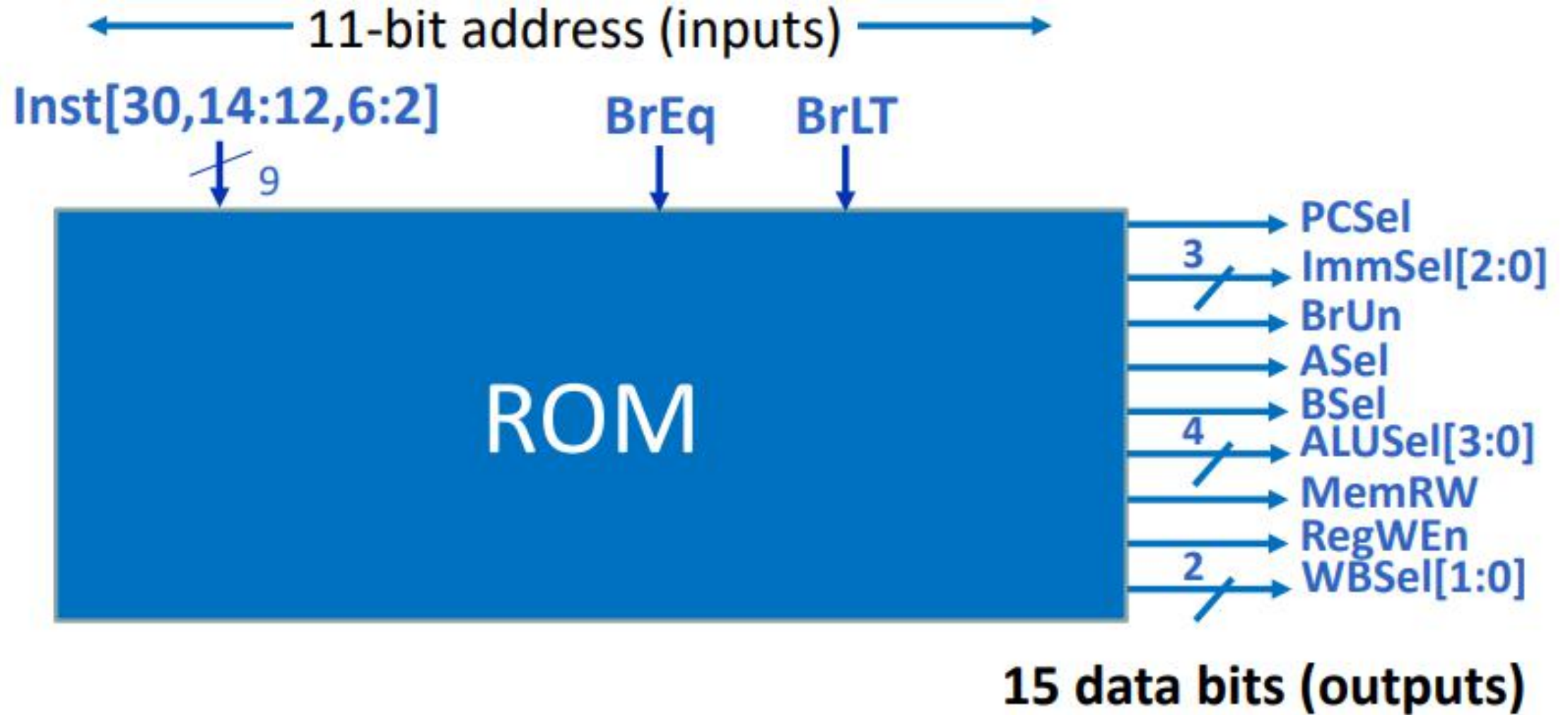
ROM 控制器实现



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
(R-R Op)	*	*	+4	*	*	Reg	Reg	(Op)	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



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