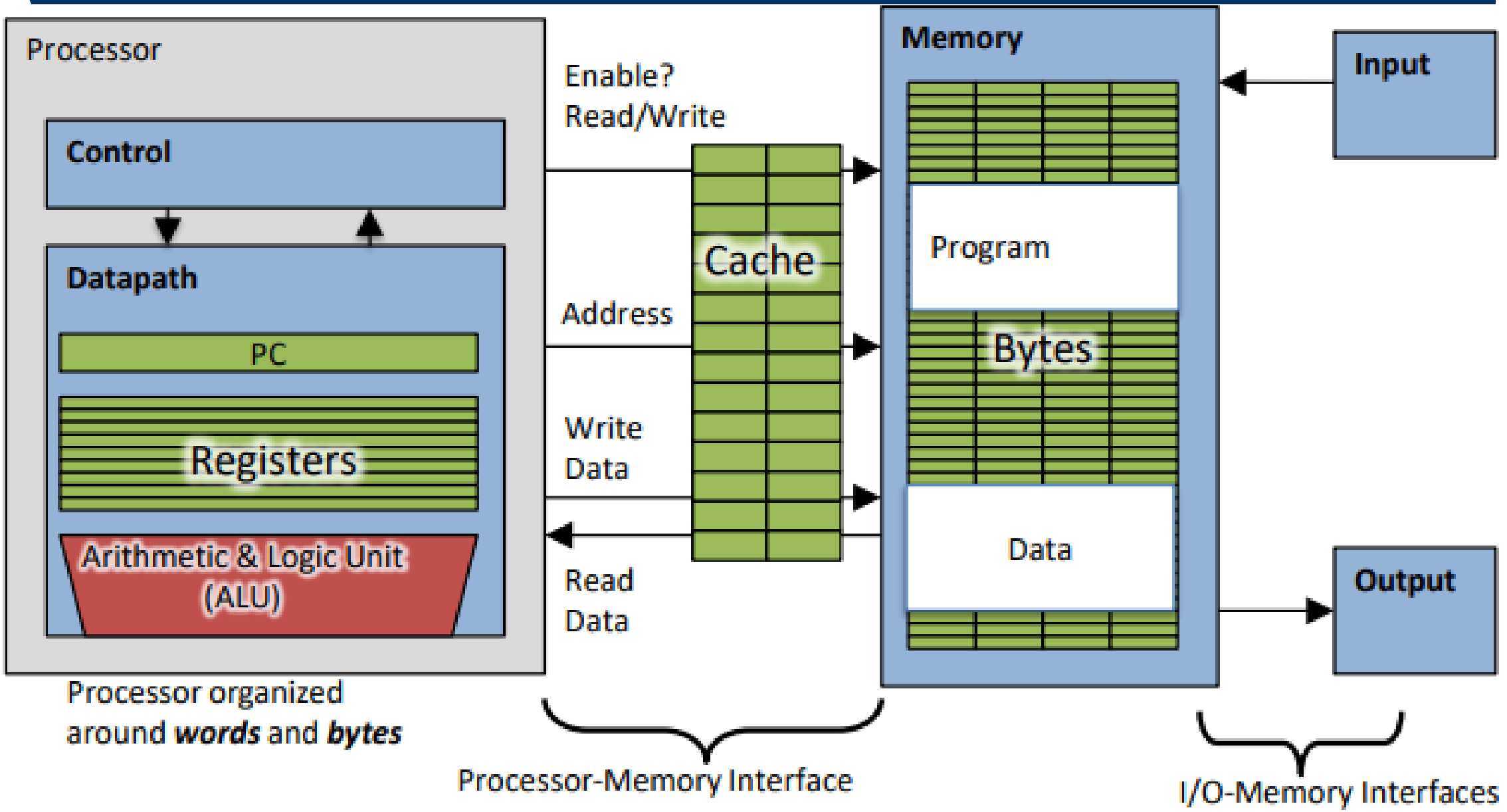
# 第4章 处理器(CPU)设计

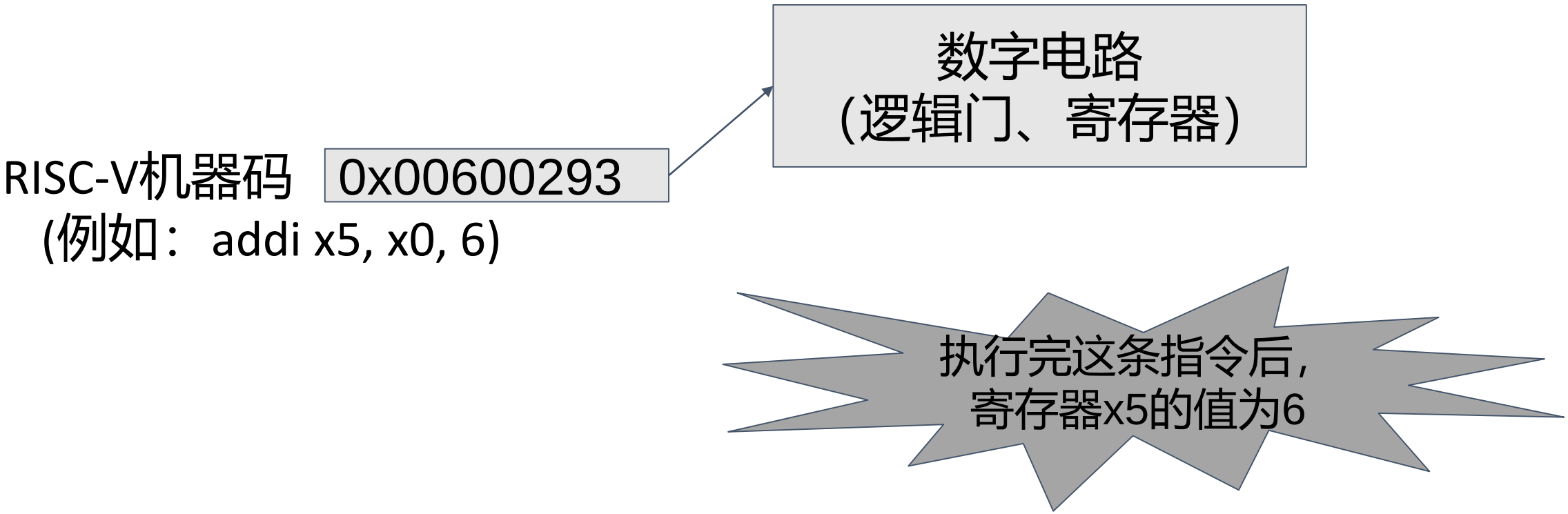
- 处理器设计的需求分析
- RISC-V数据通路的组件选择
- RISC-V部分指令的数据通路设计
- RISC-V控制器

# 单核计算机系统


Processor
Control
Datapath
PC
Registers
Arithmetic & Logic Unit (ALU)
Enable?
Read/Write
Address
Write Data
Read Data
Cache
Memory
Program
Bytes
Data
Input
Output
Processor organized around words and bytes
Processor-Memory Interface
I/O-Memory Interfaces

# 功能层面上CPU的定义

- 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。



# 处理器（CPU）的组成部分

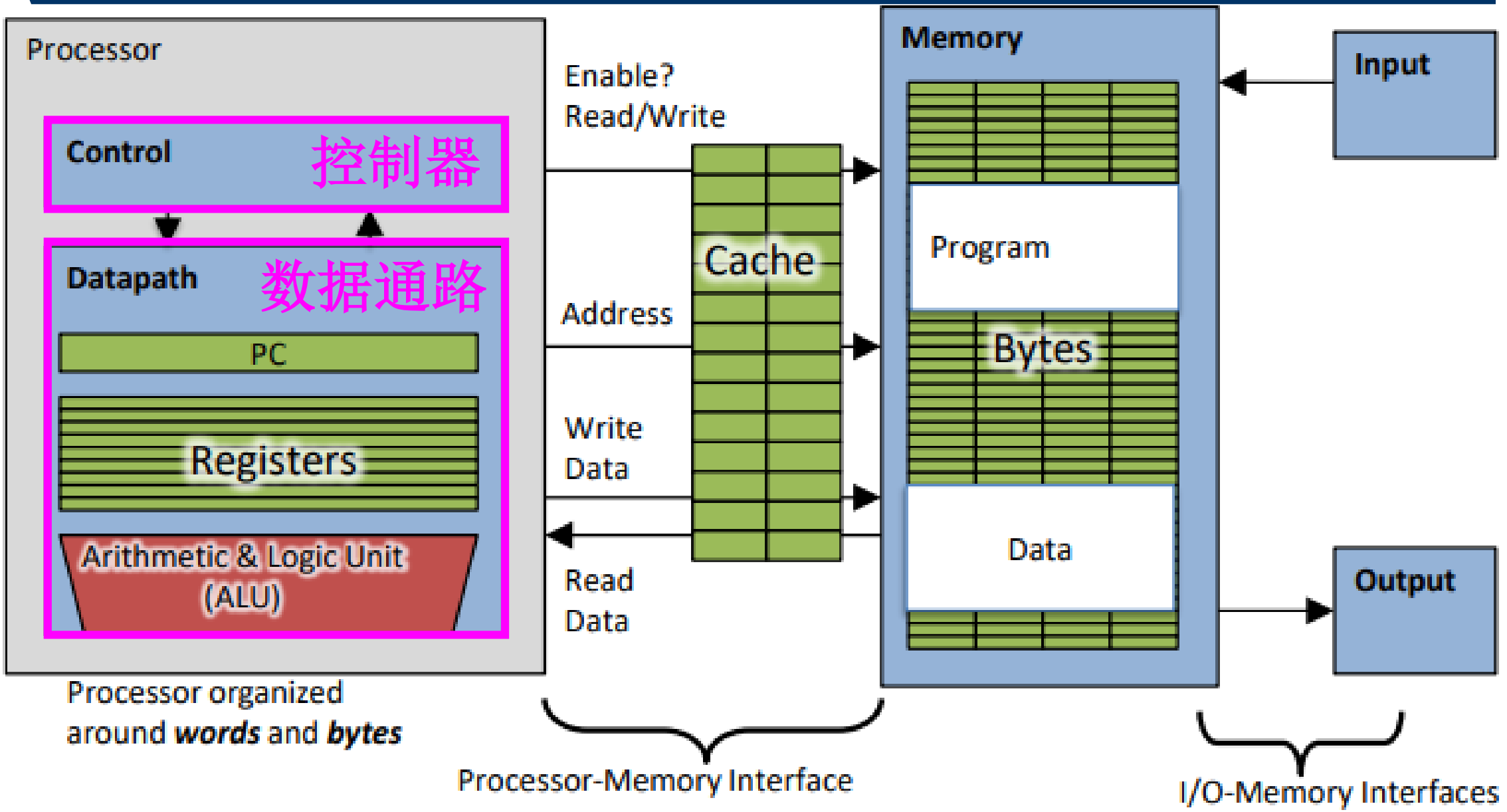
- **数据通路(datapath, ALU...)**是处理器中**执行操作**的硬件
  - 执行控制器的操作（例如:控制器“告诉”数据通路，执行add指令，则数据通路就会将相应操作数传给加法器...）

    ≈≈处理器的**四肢**

- **控制器(CU:Control Unit)**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**

# （再看）单核计算机系统


Processor
Control
控制器
Datapath
数据通路
PC
Registers
Arithmetic & Logic Unit (ALU)
Enable?
Read/Write
Address
Write Data
Read Data
Cache
Memory
Program
Bytes
Data
Input
Output
Processor organized around words and bytes
Processor-Memory Interface
I/O-Memory Interfaces

# CPU的组成部分

addi x5,x0,6 | 00000000011000000000001010010011

## 数据通路

根据收到的控制器信息可知：

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. 加法器结果存目的寄存器x5

## 控制器

**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处理器设计示例

- 本章CPU设计示例是基于以下简单指令进行讲解，包括：
  - 常用算术逻辑指令add、sub、addi等
  - 存储器访问指令ld（load doubleword）和sd（store doubleword）
  - 条件分支指令beq（branch if equal）等
  - 跳转指令jalr，jal
  - lui，auipc等指令
- 这个子集没有包含所有的定点指令（例如移位、乘法、除法等指令均不在集合中），也没有包含任何浮点指令。
- 其余指令的实现与此类似。

# RISC-V指令格式回顾及需求

- 简单源于规整

| R型 | func7 | rs2 | rs1 | func3 | rd | opcode |
|---|---|---|---|---|---|---|
| I型 | imm[11:5] | imm[4:0] | 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 **rd, rs1, rs2**
  - 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 rd, rs1, **imm** # R[rd]=R[rs1] + **s-ext**(imm)
  - load类指令 rd，offset(rs1) #R[rd]=**mem**[R[rs1] + 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类指令 rs2, offset(rs1) #**mem**[R[rs1] + offset]=R[rs2]
  - 新增需求：存储器**可写**、立即数**生成**器（包括扩展等）

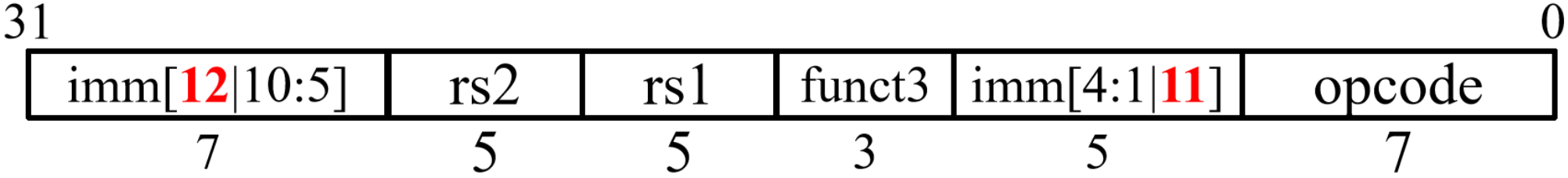
S型

| **imm[11:5]** | rs2 | rs1 | func3 | **imm[4:0]** | opcode |
|---|---|---|---|---|---|

# RISC-V 部分指令回顾——续

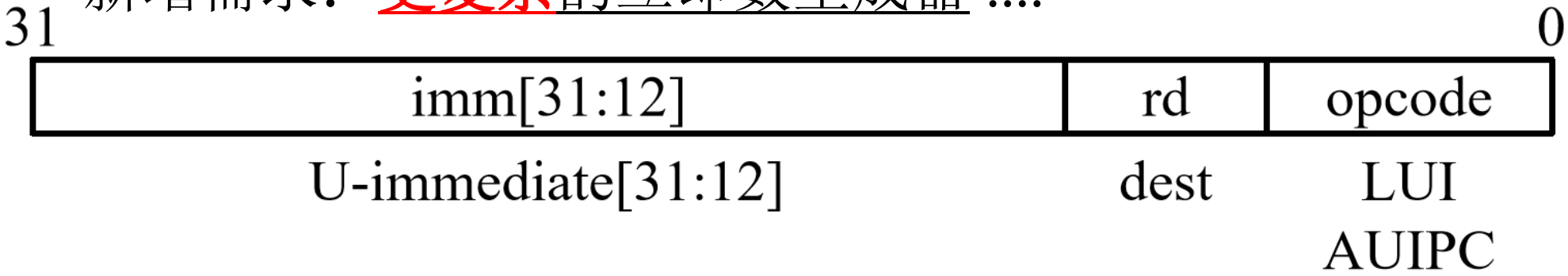
- B型指令： bxx rs1,rs2, **label**
  - 新增需求： 多路选择器、**更复杂**的立即数生成器、比较器

**B型**



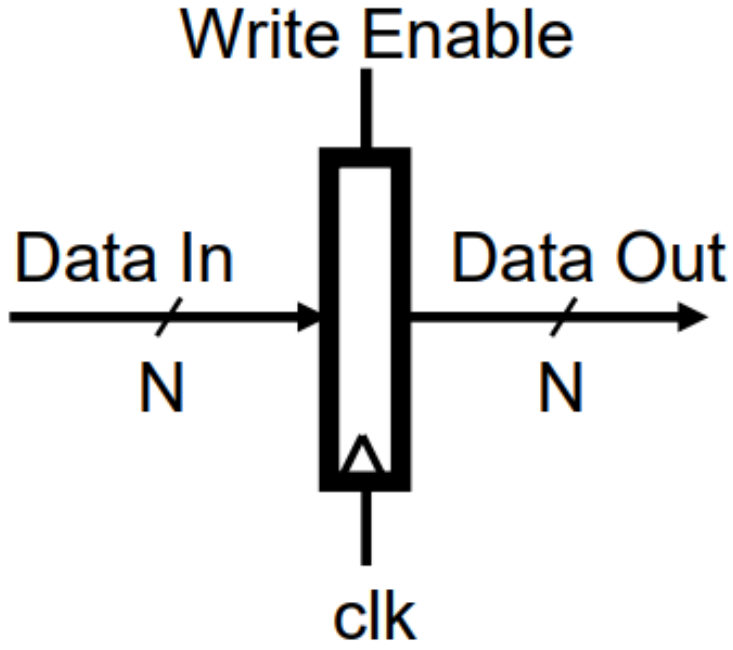

- J型指令： jal rd, offset
- U型指令： 将**长立即数**加到PC并写入目的寄存器
  - 新增需求： **更复杂**的立即数生成器 ....

**U型**



# 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
- 立即数生成器
  - 零扩展、符号扩展、字段拼接等
- 程序计数器（PC）
- 存储组件：寄存器堆(RF:Register File)，存储器(Memory)



# 第4章 处理器设计

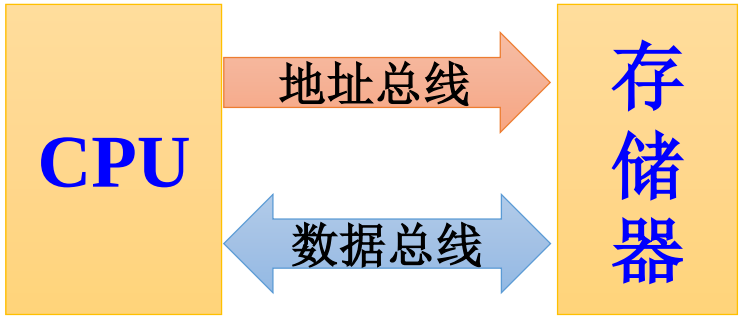
- 处理器设计的需求分析
- **RISC-V数据通路的组件选择**
- RISC-V部分指令的数据通路设计
- RISC-V控制器

# 数据通路组件分类

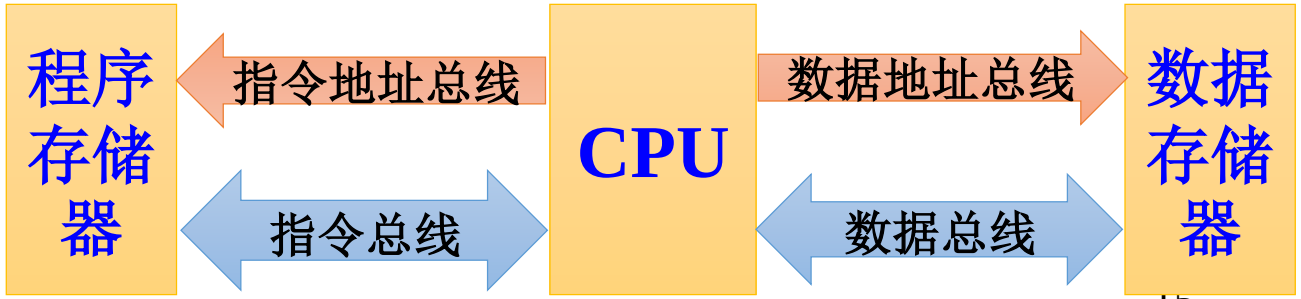
- 组合逻辑单元
  - 处理数据的单元
  - 没有存储功能
  - 输出仅依赖于输入：输入相同，则输出相同
  - 如基本逻辑门、ALU等
- 时序逻辑单元（状态单元）
  - 存储数据的单元
  - 有内部存储功能，即包含“状态”
  - 下一状态取决于输入和当前的状态
  - 如指令存储器、数据存储器以及寄存器（**PC以及寄存器堆**）

# 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和程序存储器（指令存储器），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。

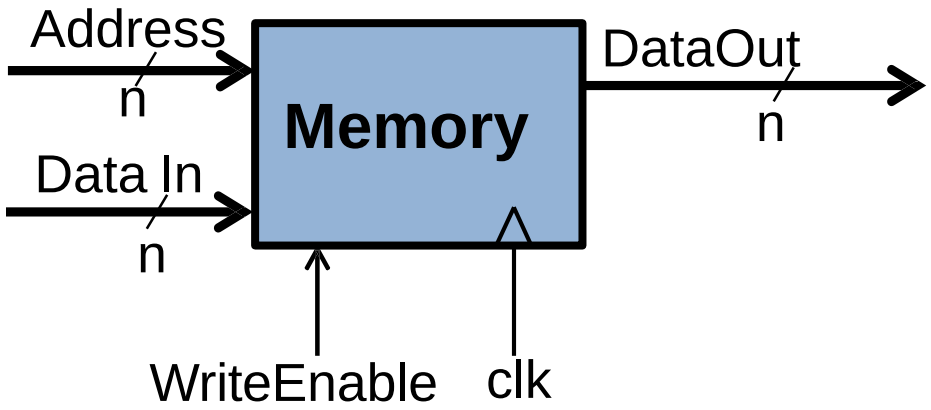


# RISC-V 主要状态单元——存储器

- 将指令和数据**分开**保存在一个64位的**字节寻址**的存储空间中
- 指令存储器 IM（IMEM）： Instruction Memory
- 数据存储器 DM（DMEM）： Data Memory)
- 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

# 数据存储器（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

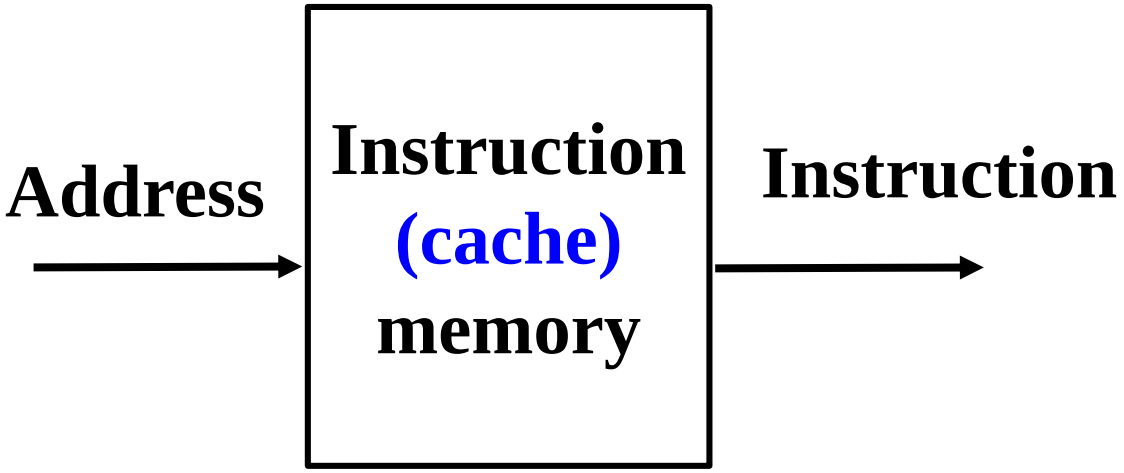



- 读写控制
  - Address：指定一个存储单元，将其内容送到数据输出端
  - WriteEnable：写使能。在时钟信号（clk）的上升沿，如果写使能信号有效，将数据输入信号的内容存入地址信号指定存储单元

**注：存储器的读操作不受时钟控制**

# 指令存储器（IM:Instruction Memory）

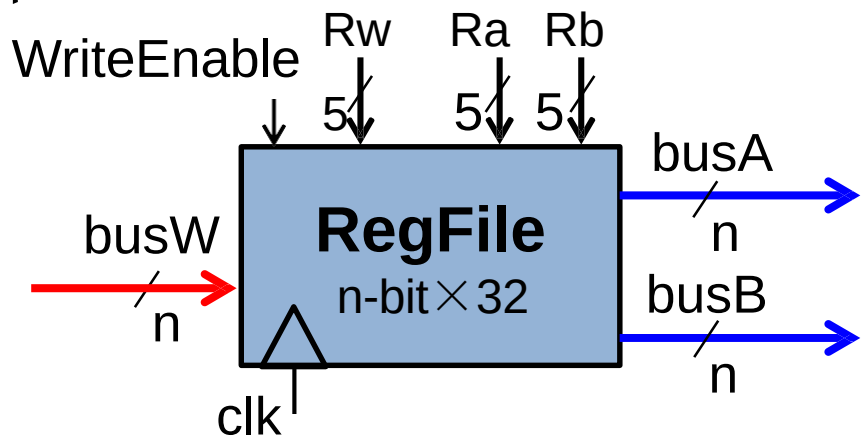
- 输入总线： Instruction Address
- 输出总线： Instruction
- 数据通路中不会写指令存储器




- 读操作时，指令存储器可以看做是组合逻辑电路。

# RISC-V主要状态单元——寄存器堆

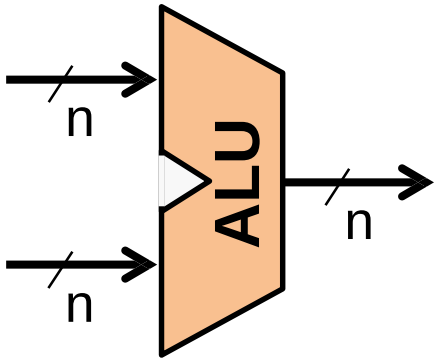
- (RegisterFile)内部构成：32个寄存器
- 数据接口信号
  - busA，busB：两个数据输出
  - busW：一个数据输入
- 读写控制
  - Ra/Rb(5位)：将对应编号的寄存器的内容放到busA/busB(读)
  - Rw(5位)：在时钟信号(clk)的上升沿，如果写使能信号(WriteEnable)有效，将busW的内容存入Rw号寄存器。

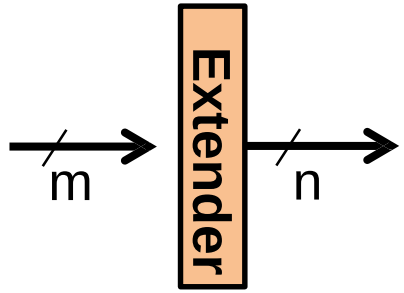



**注：寄存器堆的读操作不受时钟控制，相当于组合逻辑电路。**

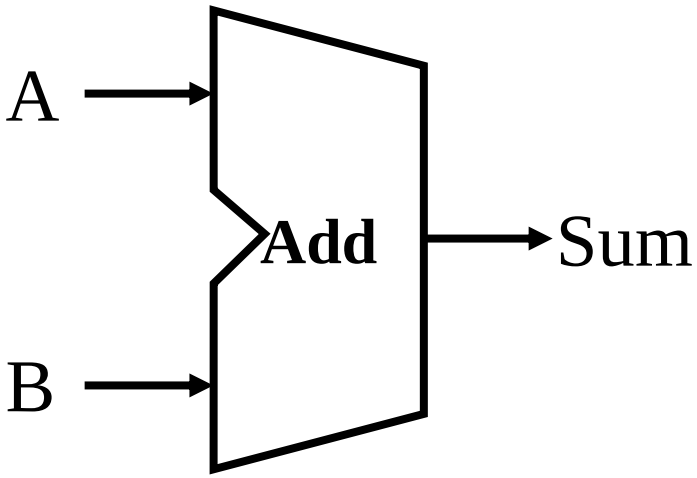
# RISC-V指令系统的数据通路需求

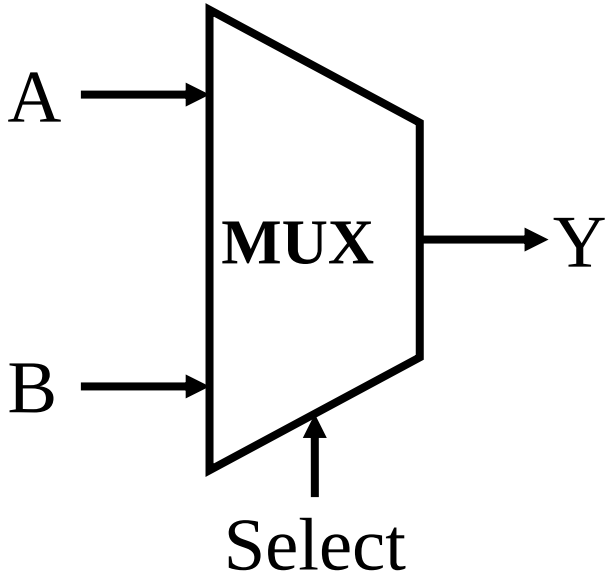
- 算术逻辑单元（ALU）
  - 运算类型：加、减、或、比较等
  - 操作数：来自寄存器或扩展后的立即数
- 立即数生成（伸缩扩展）部件
  - 将一个不足32/64位的立即数扩展为32/64位
  - 扩展方式：零扩展、符号扩展、拼接
- 程序计数器（PC）
  - 一个32/64位的寄存器
  - 支持两种加法：加4 或 加一个立即数

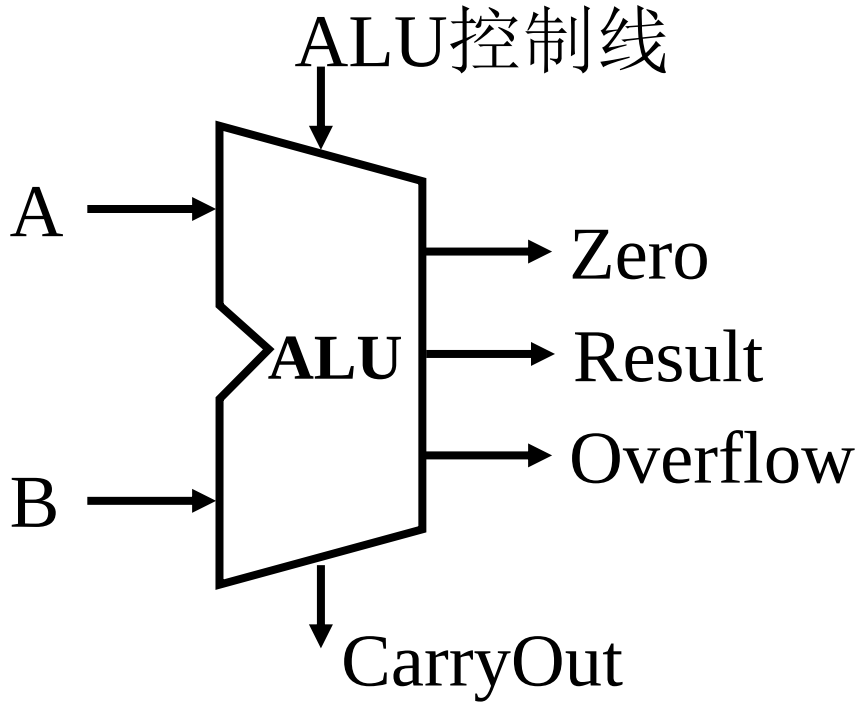



# 数据通路模块——常用组合逻辑单元

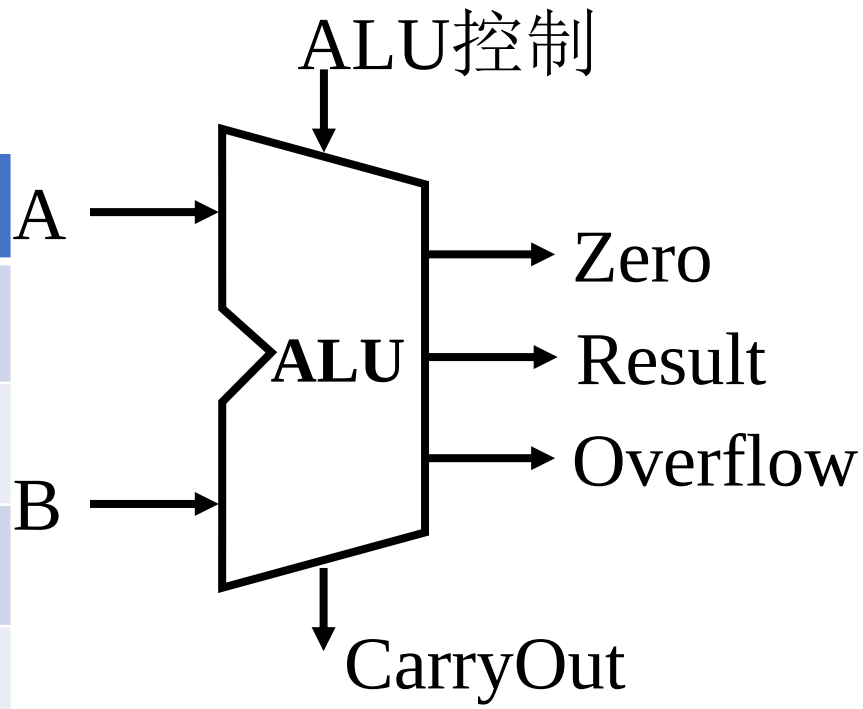


# ALU功能需求

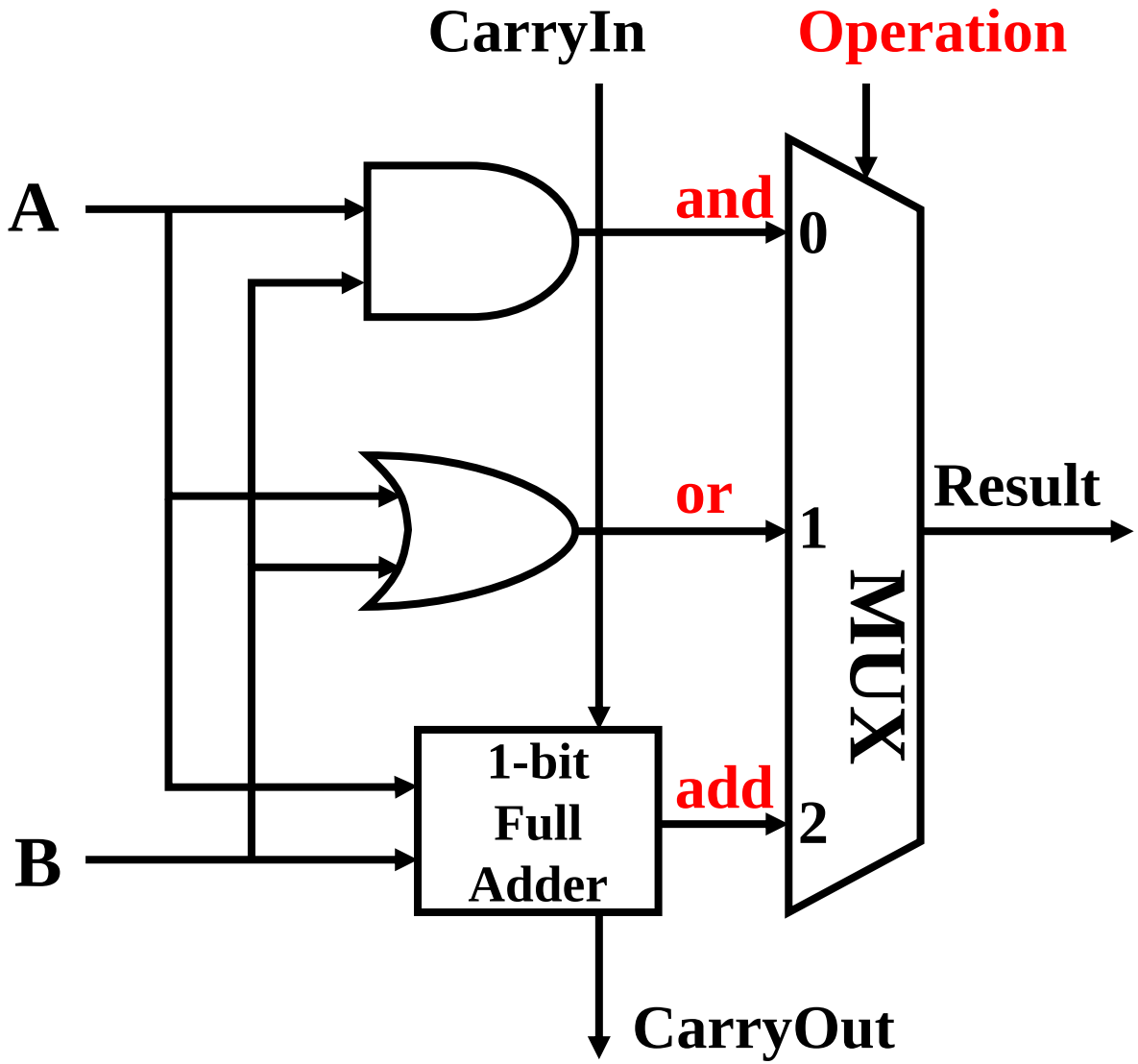
| ALU控制 | 操作 |
| --- | --- |
| 0000 | 与(and) |
| 0001 | 或(or) |
| 0010 | 加(add) |
| 0110 | 减(sub) |
| 0111 | 小于则置位(slt) |
| 1100 | 或非(nor) |



# 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

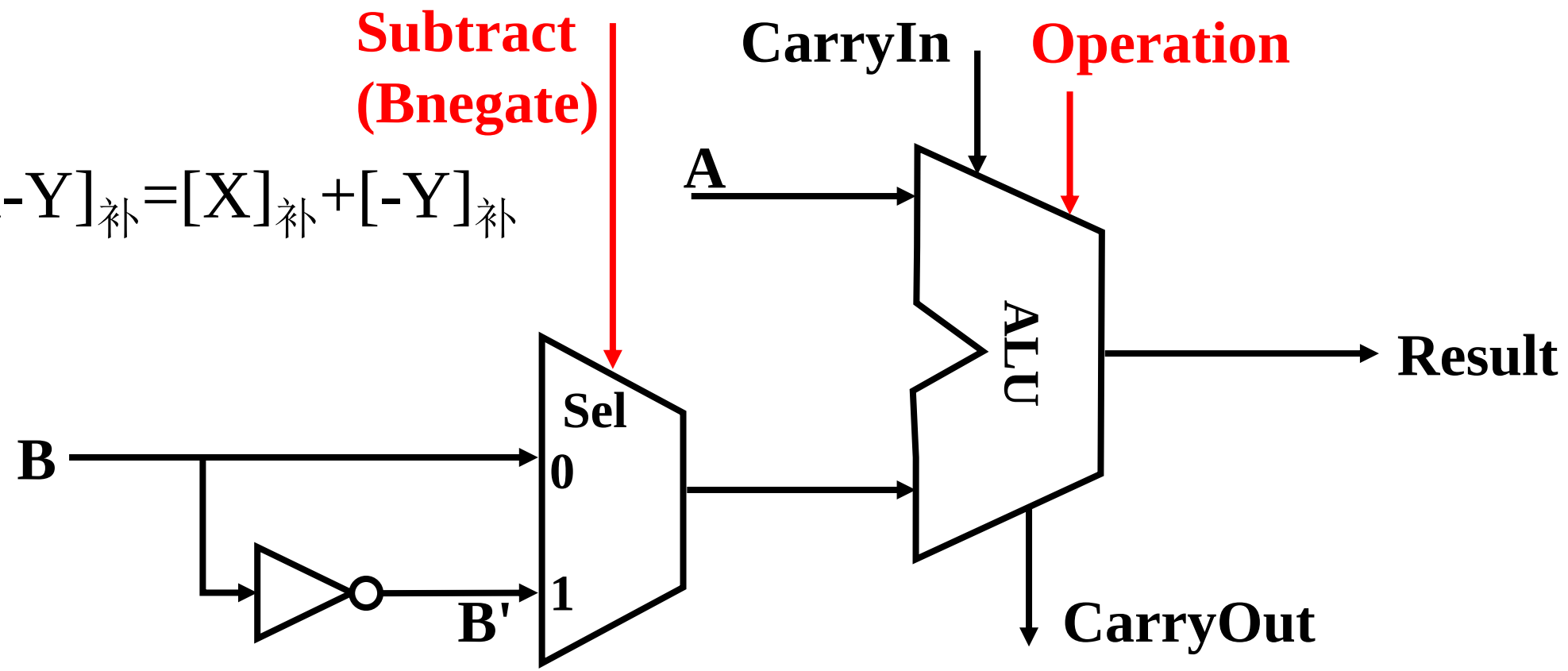
# ALU——简单运算


CarryIn
Operation
A
and
0
or
1
Result
MUX
1-bit
Full
Adder
add
2
B
CarryOut

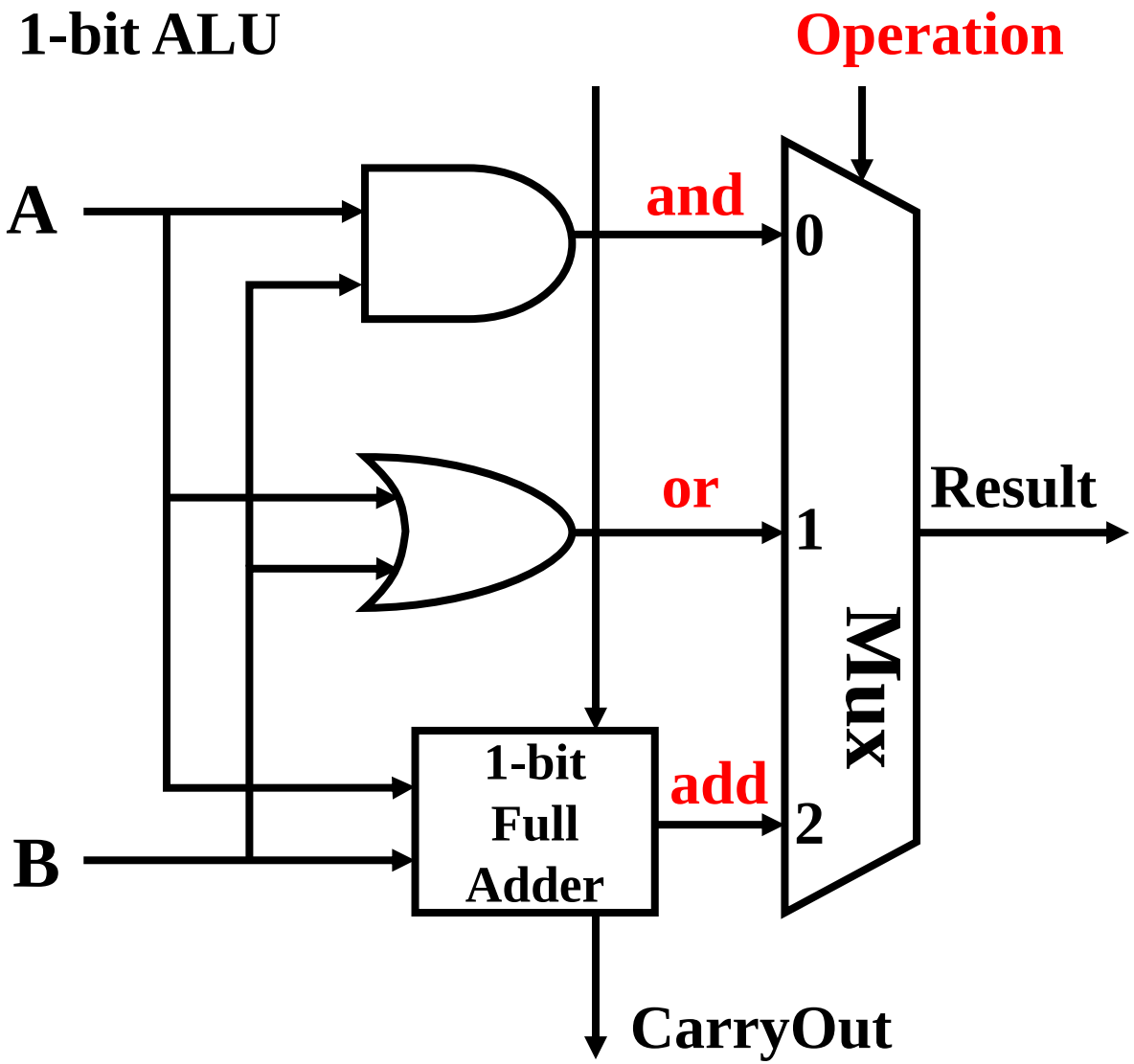
# ALU——如何进行减法运算

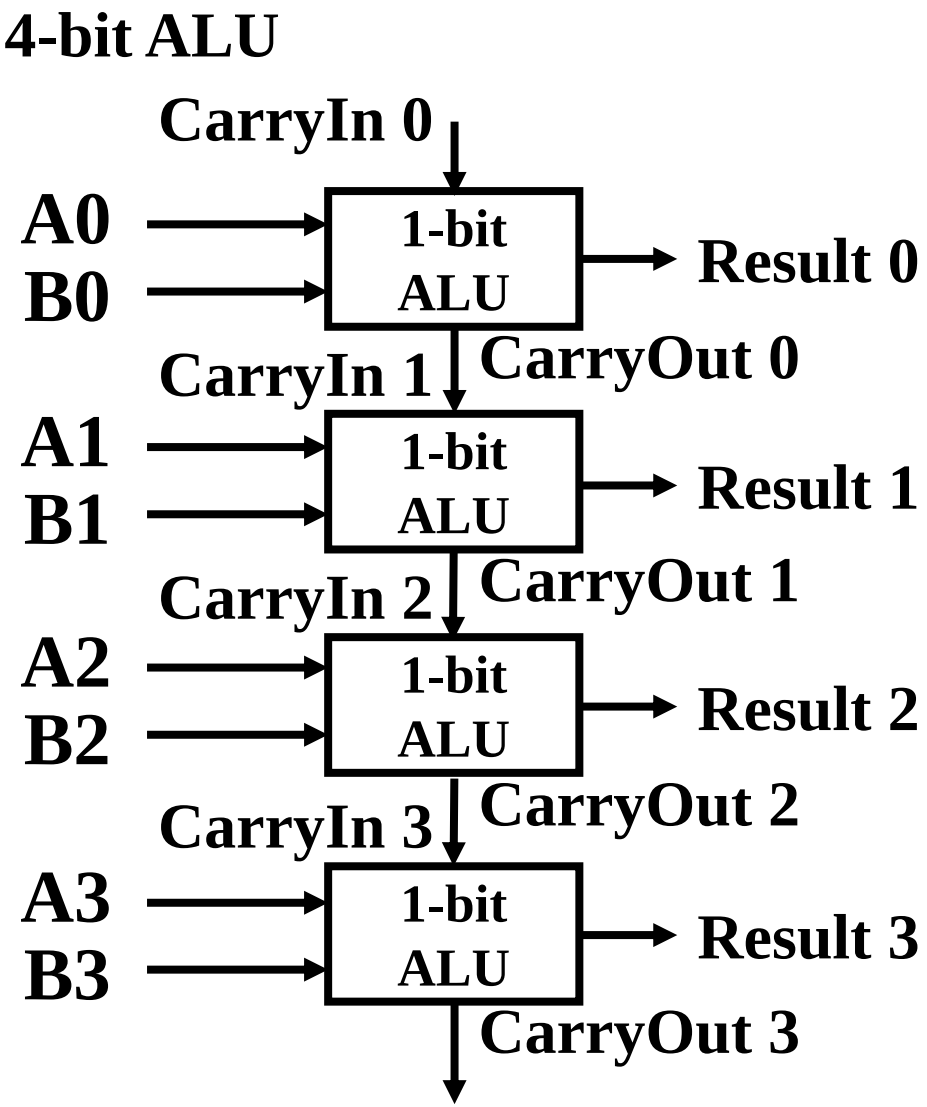
A-B=A+B’+1（**注意：这里A和B都是1位二进制**）

$[X-Y]_{补}=[X]_{补}+[-Y]_{补}$



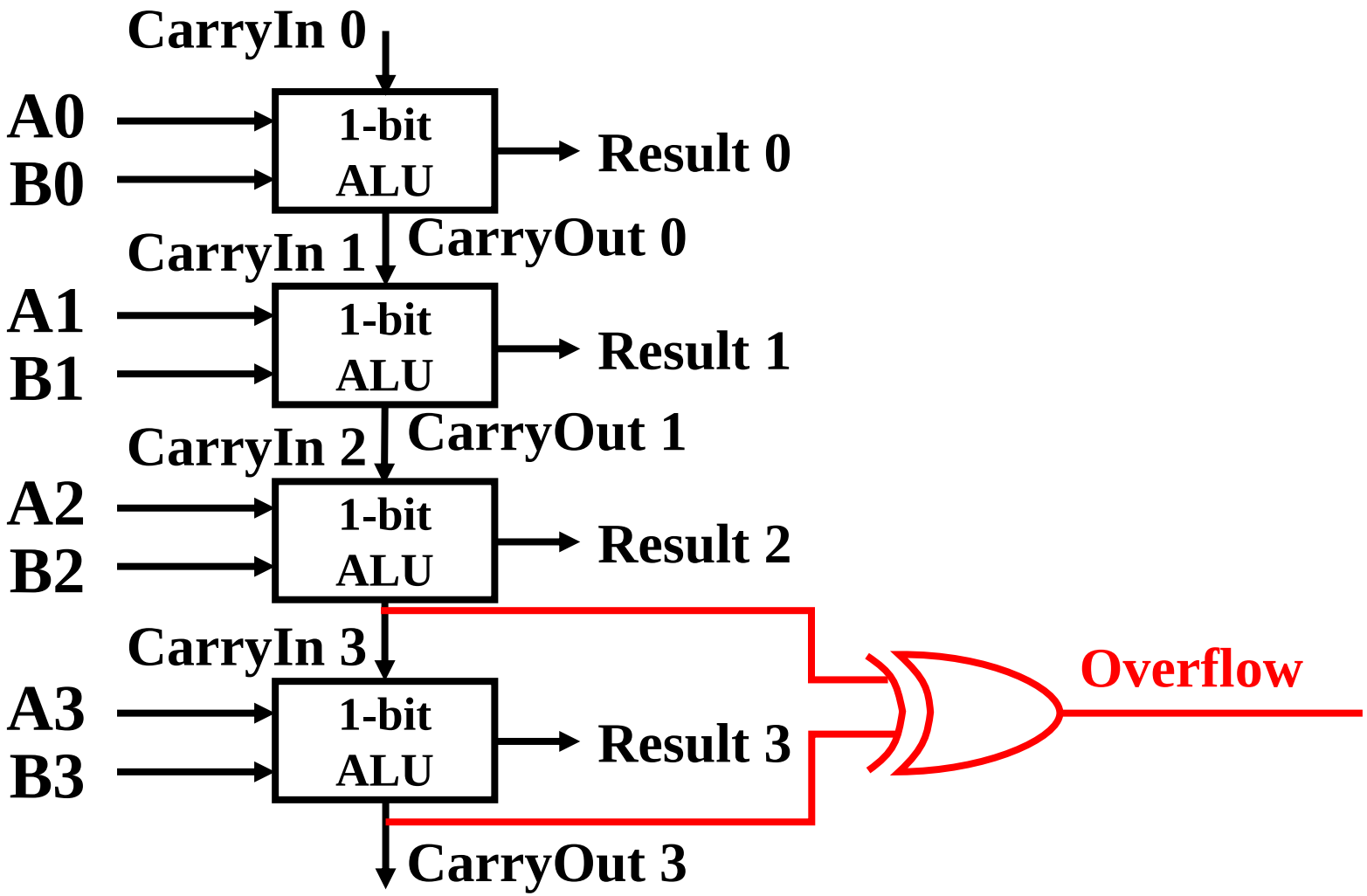
# 1-bit到多位ALU


1-bit ALU
Operation
A
B
and
or
add
0
1
2
Mux
Result
1-bit
Full
Adder
CarryOut

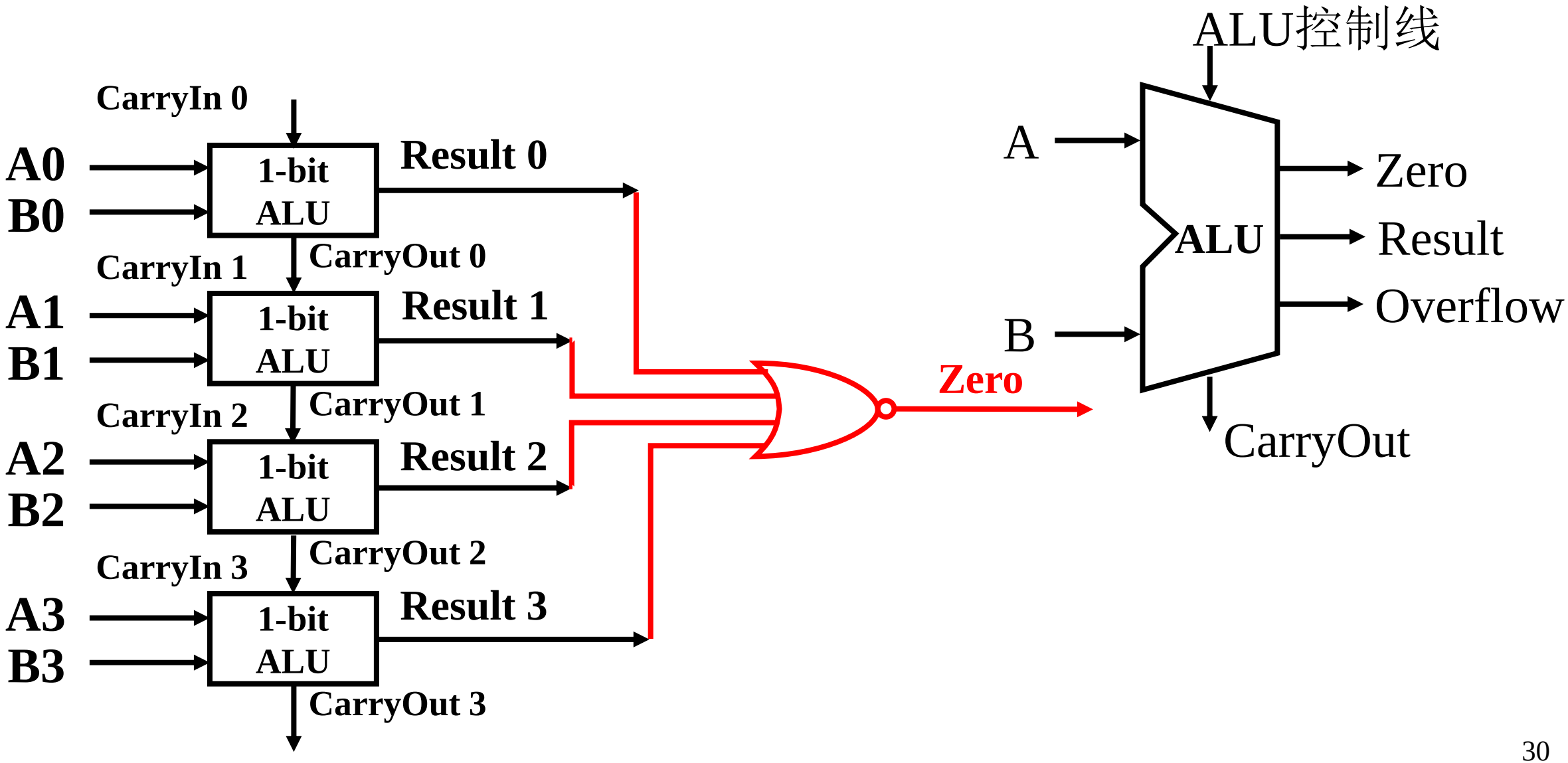

4-bit ALU
CarryIn 0
A0
B0
1-bit
ALU
Result 0
CarryOut 0
CarryIn 1
A1
B1
1-bit
ALU
Result 1
CarryOut 1
CarryIn 2
A2
B2
1-bit
ALU
Result 2
CarryOut 2
CarryIn 3
A3
B3
1-bit
ALU
Result 3
CarryOut 3

# ALU——溢出检测逻辑

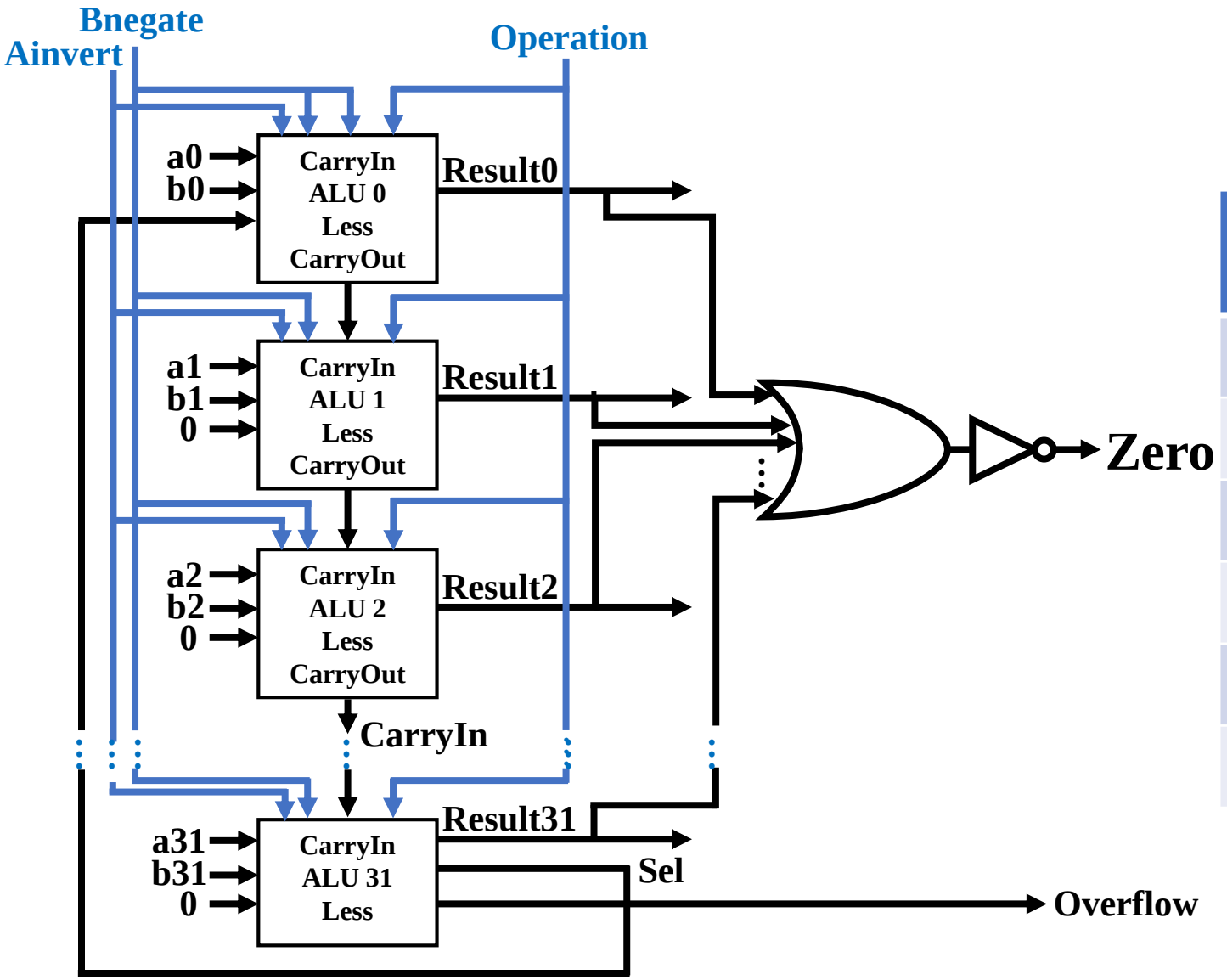
Overflow=CarryIn[N-1] xor CarryOut[N-1]



# ALU——判零逻辑


CarryIn 0
A0
B0
1-bit ALU
Result 0
CarryOut 0
CarryIn 1
A1
B1
1-bit ALU
Result 1
CarryOut 1
CarryIn 2
A2
B2
1-bit ALU
Result 2
CarryOut 2
CarryIn 3
A3
B3
1-bit ALU
Result 3
CarryOut 3
Zero
ALU控制线
A
B
ALU
Zero
Result
Overflow
CarryOut

# 32bit ALU




| ALU控制(ALUControl) | 操作 |
|---|---|
| 0000 | 与(and) |
| 0001 | 或(or) |
| 0010 | 加(add) |
| 0110 | 减(sub) |
| 0111 | 小于则置位(slt) |
| 1100 | 或非(nor) |

# 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

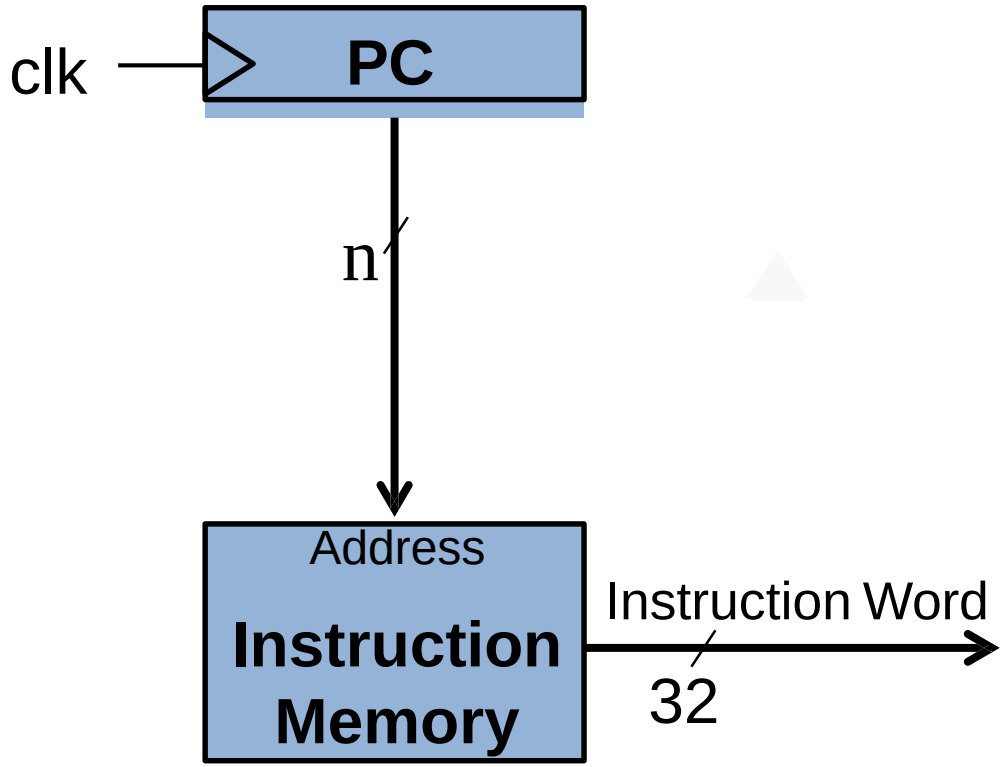
**③ 根据指令需求连接组件建立数据通路**

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

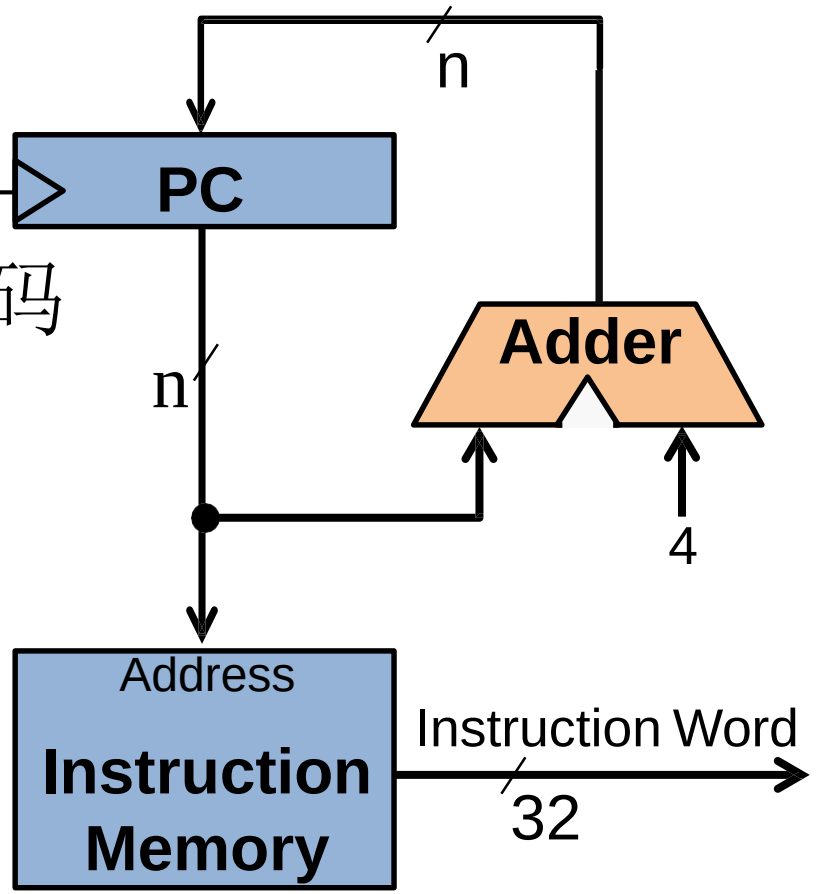
# 所有指令的共同需求——取指令

- 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



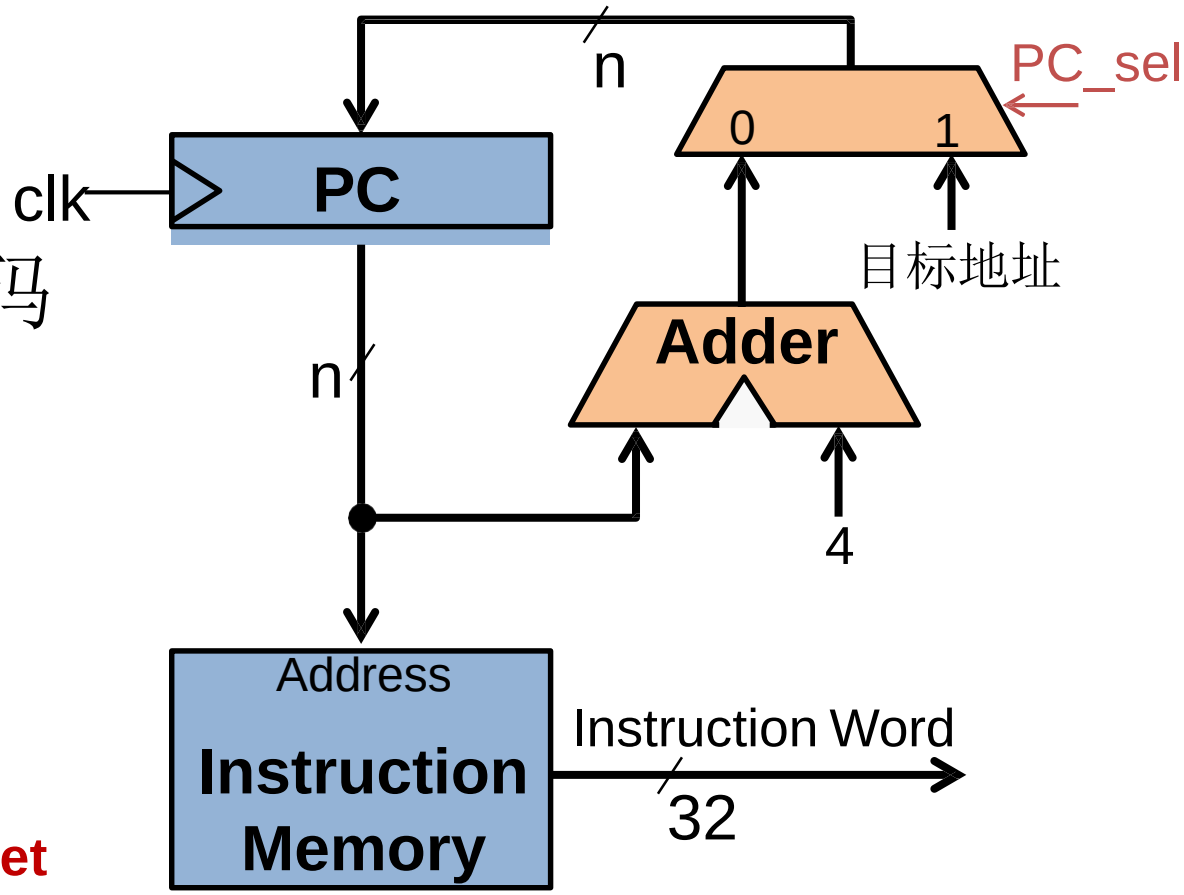
# 所有指令的共同需求——更新PC

- 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 <— PC + 4
- 发生分支及跳转时更新
  - PC <—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 <— PC + 4
- 发生分支及跳转时更新
  - PC <— 目标地址
    - ① 对于B型指令及jal：目标地址是PC + offset
    - ② 对于jalr：目标地址是R[rs1]+offset



# 所有指令的共同需求——更新PC（分支）

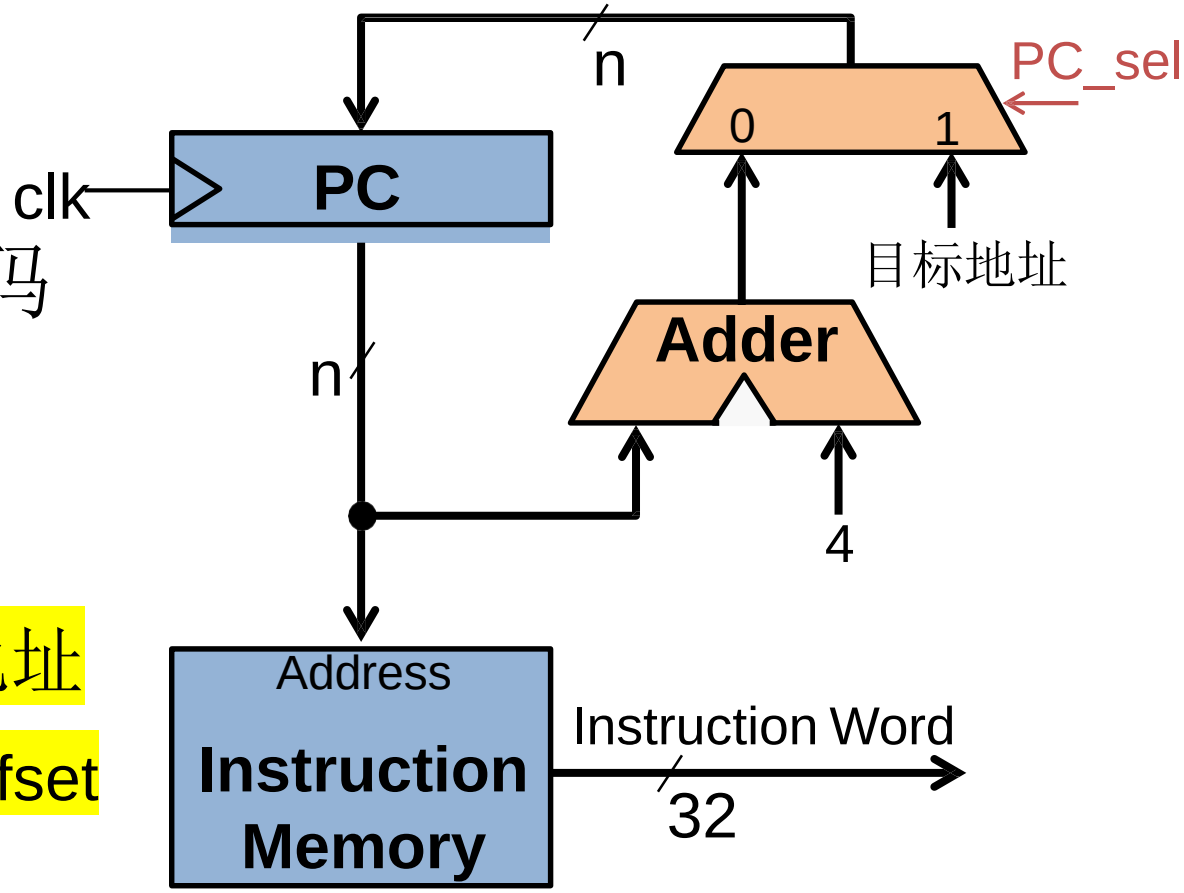
- 取指令
  - 程序计数器（PC）的内容
    是指令的地址
  - 用PC的内容作为地址，
    访问指令存储器获得指令编码
- 更新PC

1）顺序执行时：PC <— PC + 4

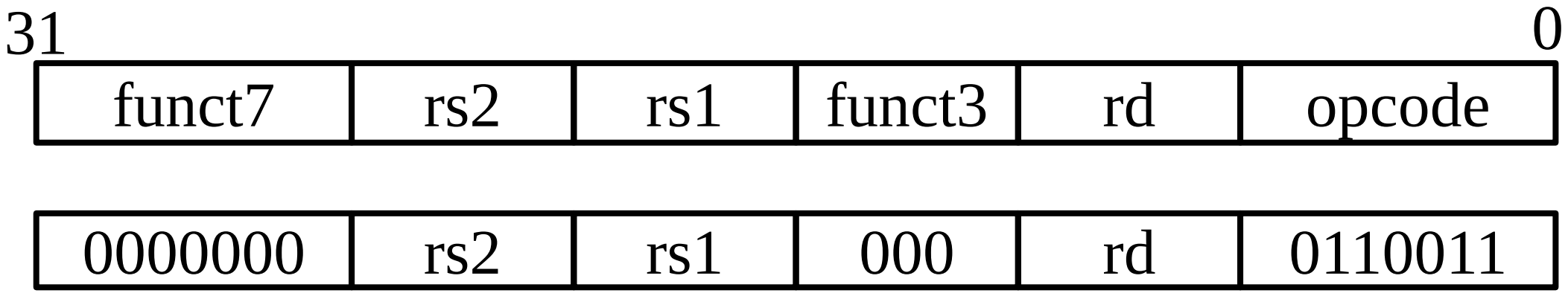
2）发生分支及跳转时：PC <— 目标地址

① 对于B型指令以及jal：目标地址是PC + offset

② 对于jalr：目标地址是R[rs1]+offset



# 实现add指令

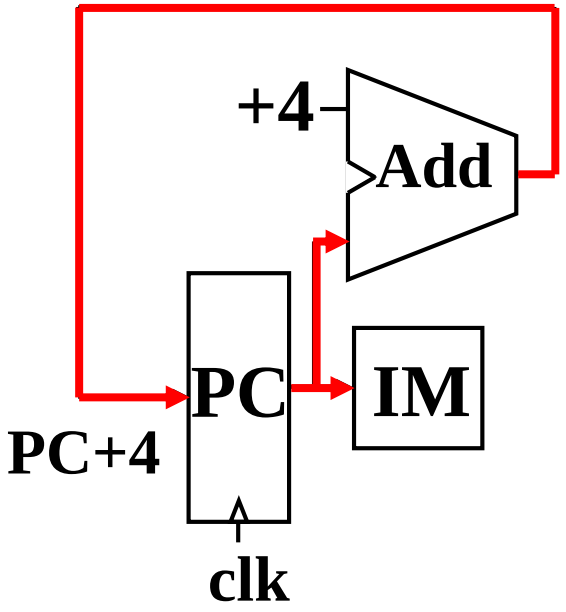



add rd, rs1, rs2

- 指令对机器状态进行两次更改：
  - Reg[rd] = Reg[rs1] + Reg[rs2]
  - PC = PC +4

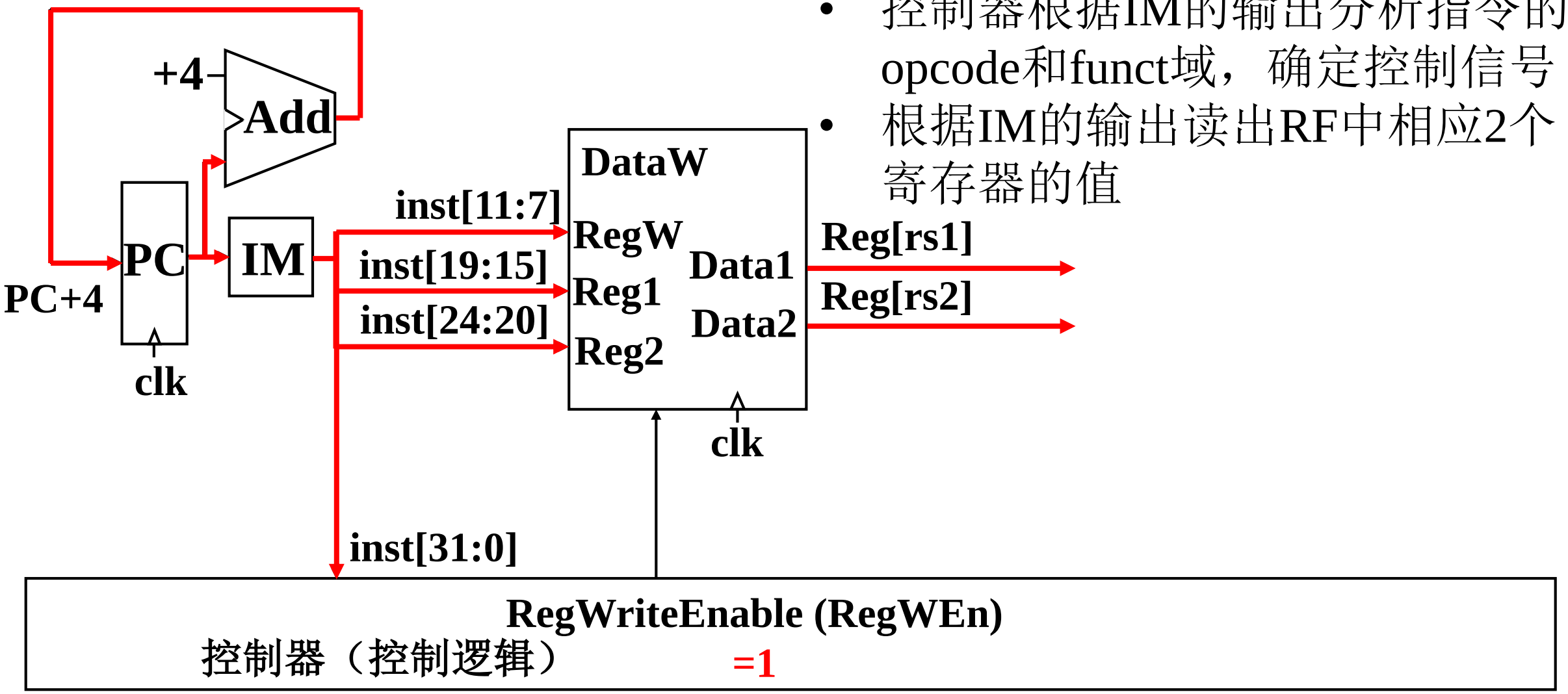
# add指令的数据通路——取指

- 根据PC从IM中读出指令
- 通过加法器计算下一个PC值(PC+4)



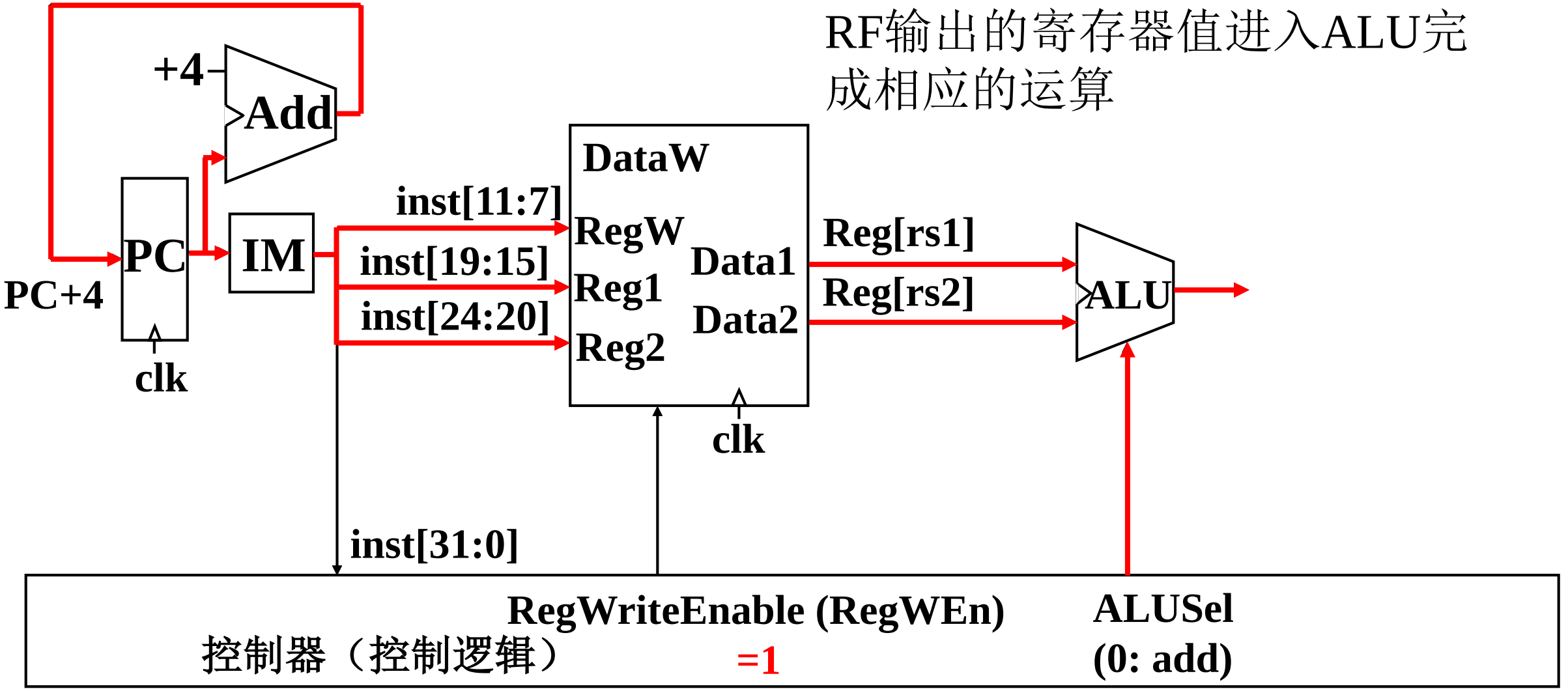
# add指令的数据通路——译码

- 控制器根据IM的输出分析指令的opcode和funct域，确定控制信号
- 根据IM的输出读出RF中相应2个寄存器的值

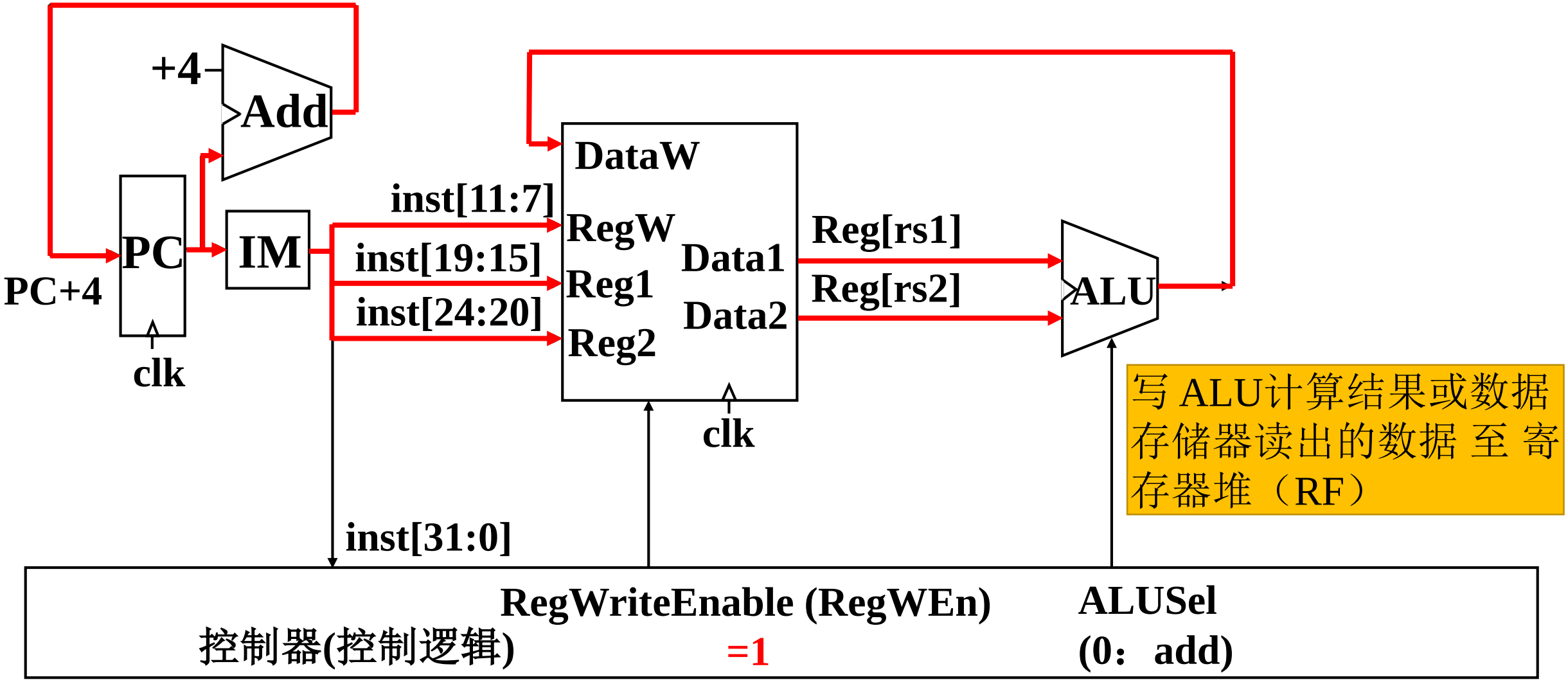


# add指令的数据通路——执行

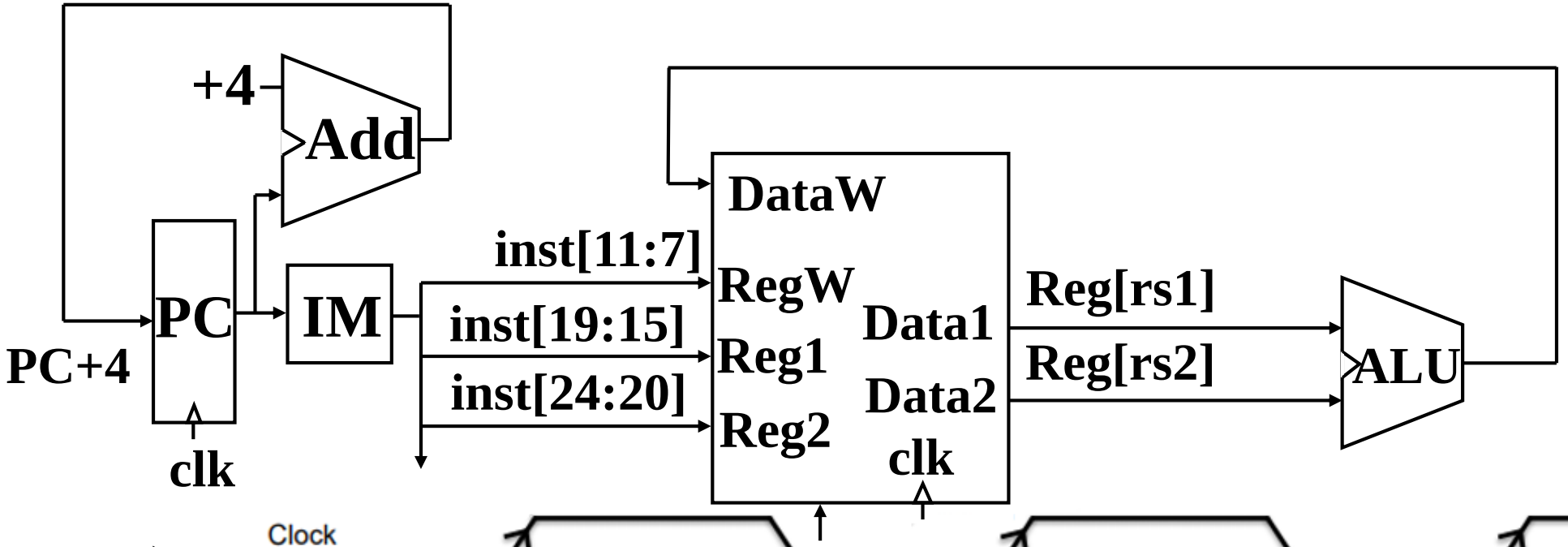
RF输出的寄存器值进入ALU完成相应的运算



# add指令的数据通路——写回寄存器



# add指令的时序图

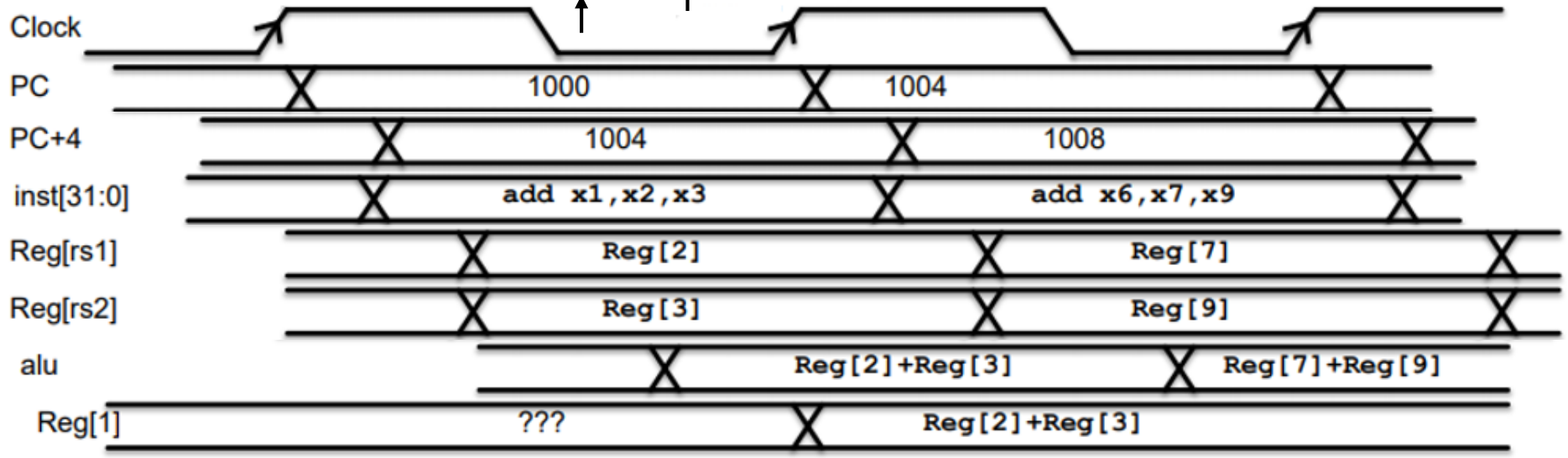



PC时延

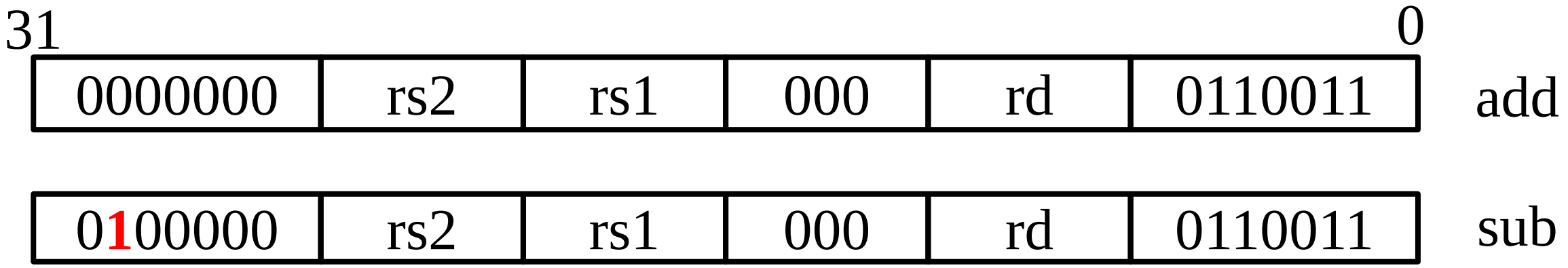
指令存储器时延

寄存器堆时延

ALU时延



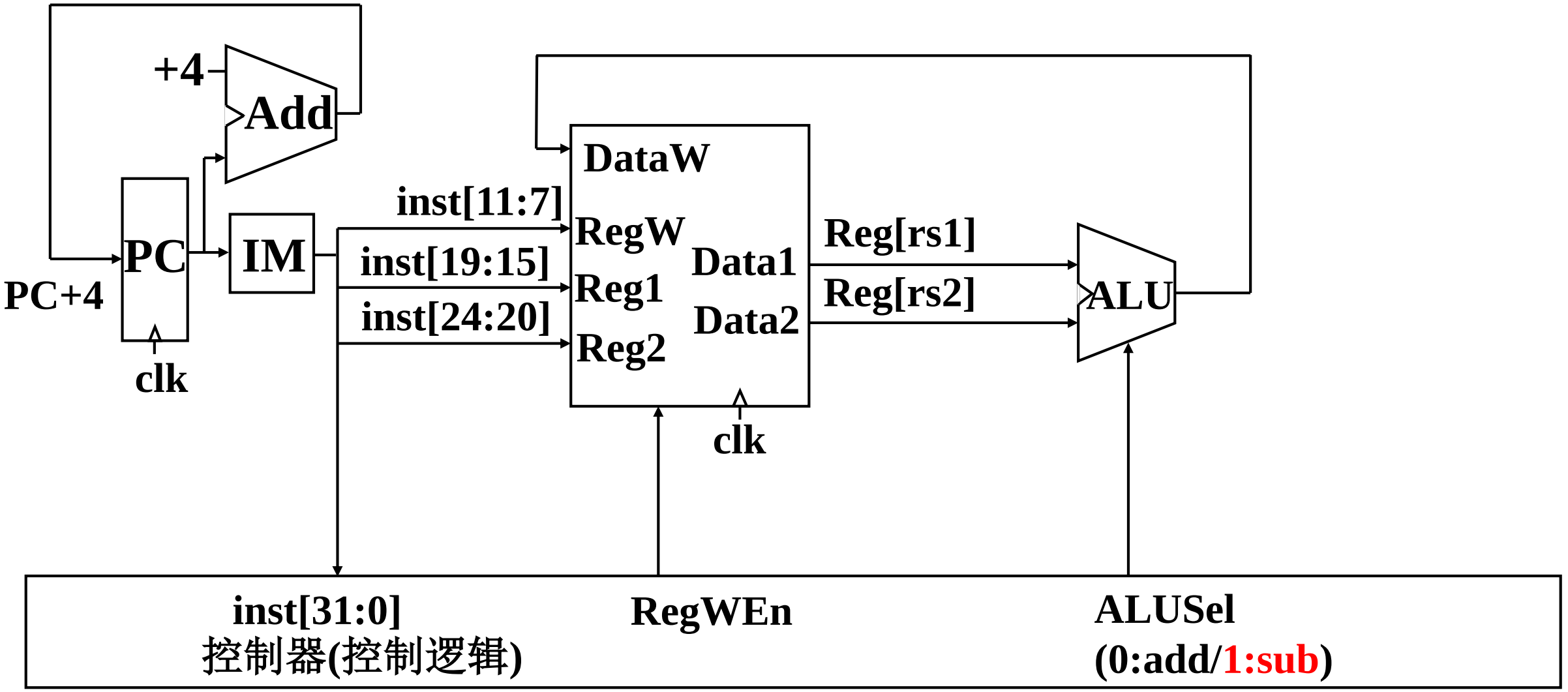
# 实现sub指令




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


+4
Add
PC
IM
PC+4
clk
inst[11:7]
inst[19:15]
inst[24:20]
DataW
RegW
Reg1
Reg2
Data1
Data2
clk
Reg[rs1]
Reg[rs2]
ALU
inst[31:0]
控制器(控制逻辑)
RegWEn
ALUSel
(0:add/1:sub)

# 实现R型部分指令

| | | | | | | |
|---|---|---|---|---|---|---|
| 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 |

opcode、funct3和funct7字段确定ALU的不同功能

# 回顾addi指令

**addi rd, rs1, imm**

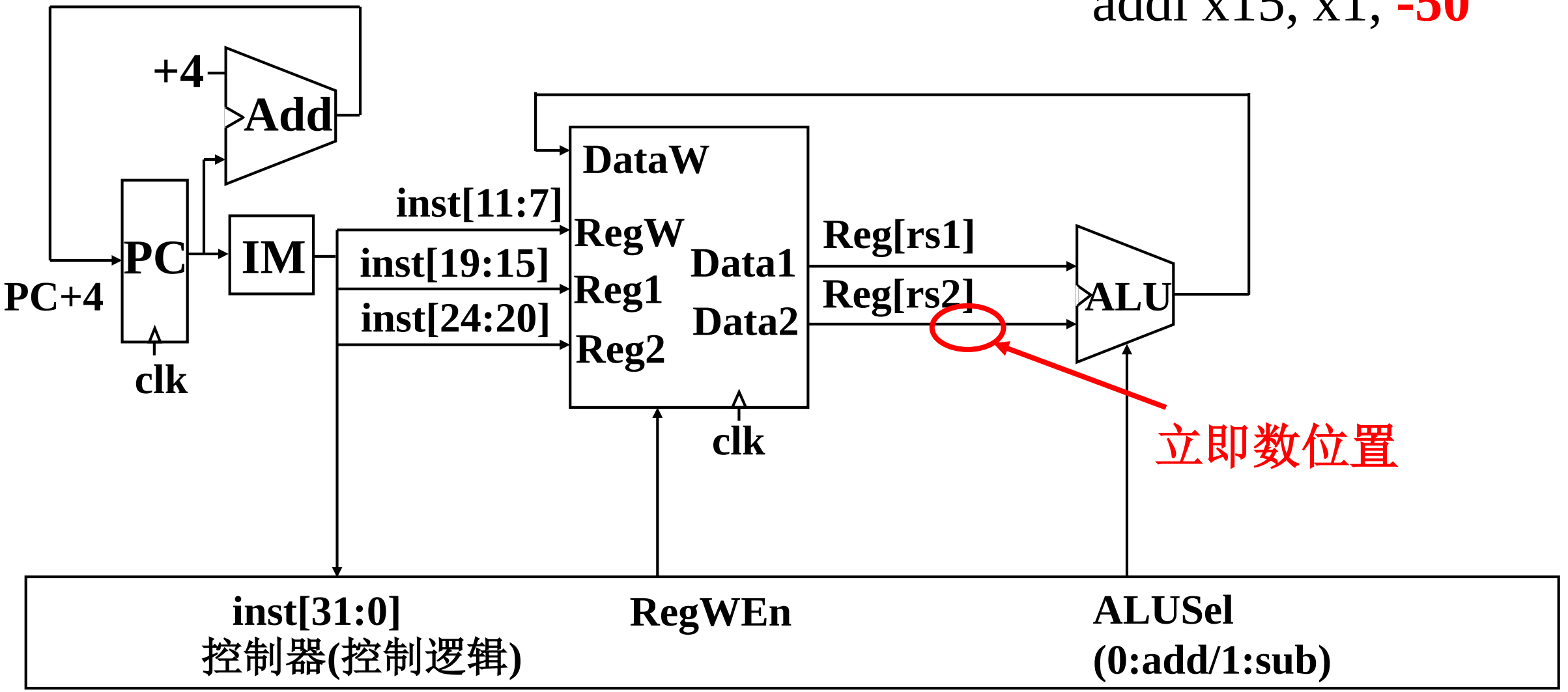
addi x15, x1, -50

31 0

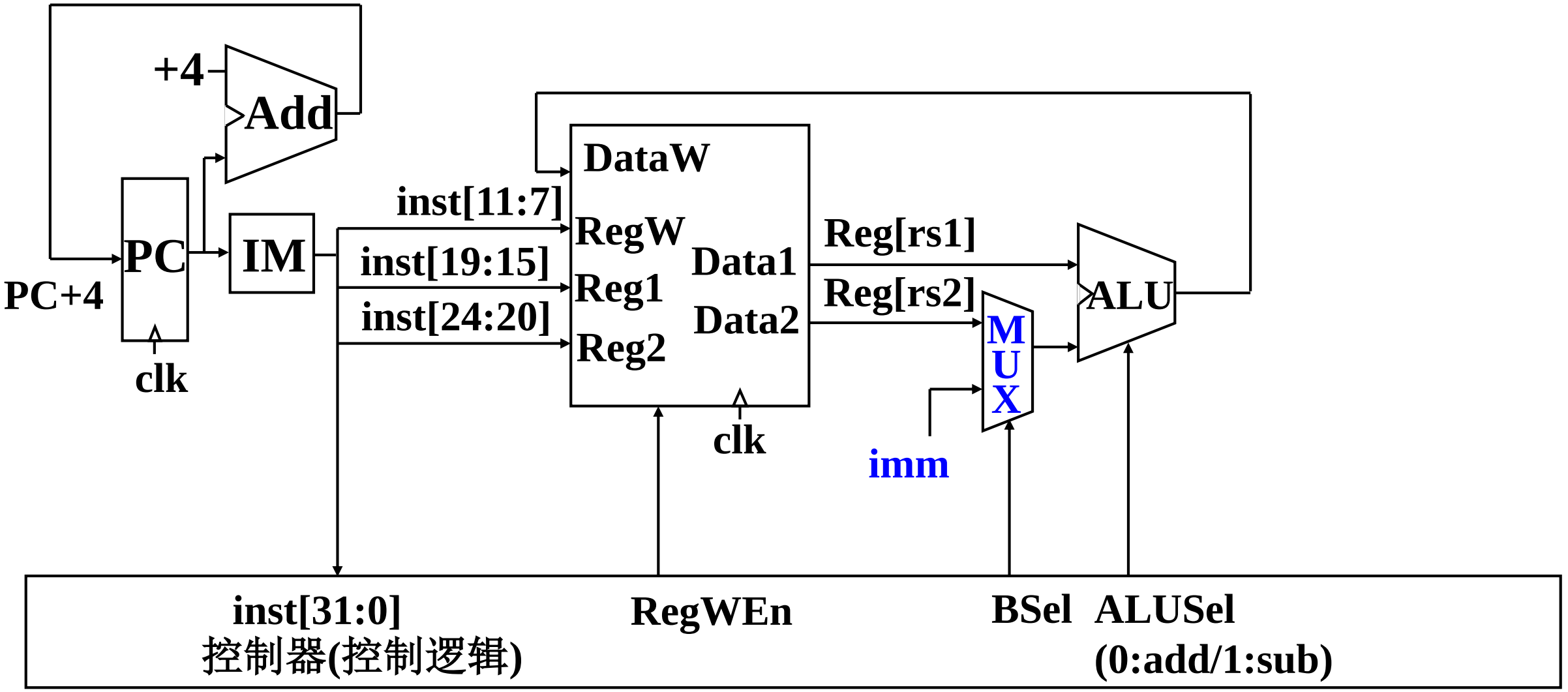
| imm[11:5] | imm[4:0] | rs1 | func3 | rd | opcode |
|---|---|---|---|---|---|
| 1111110 | 01110 | 00001 | 000 | 01111 | 0010011 |
| imm= -50 | | rs1=1 | add | rd=15 | OP-Imm |

# add指令数据通路—>addi?

addi x15, x1, **-50**



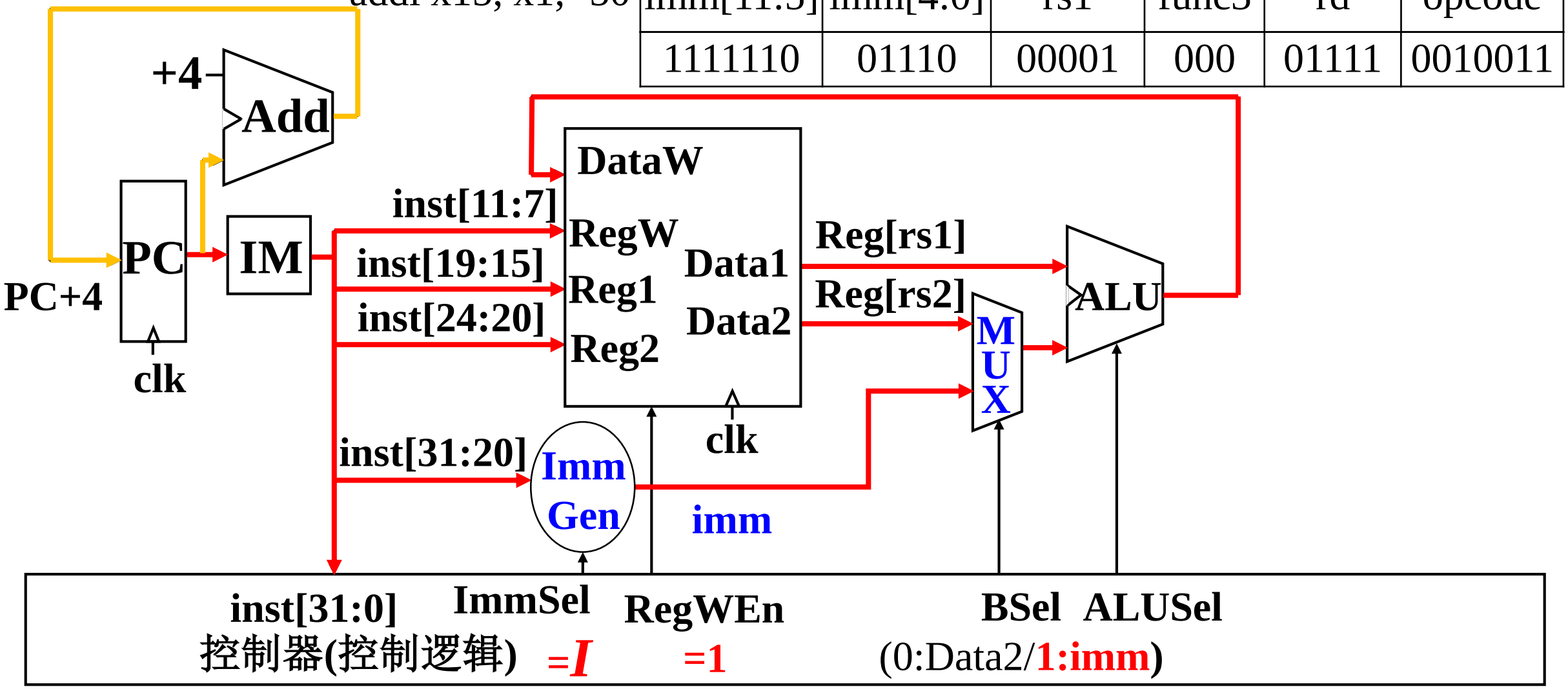
# addi指令的数据通路



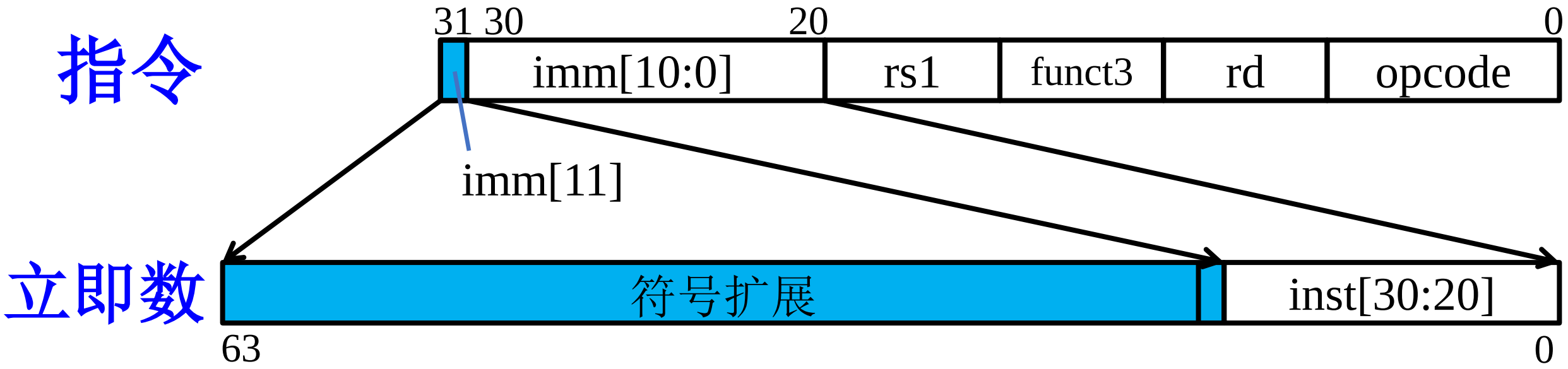
# addi指令的数据通路

addi x15, x1, -50

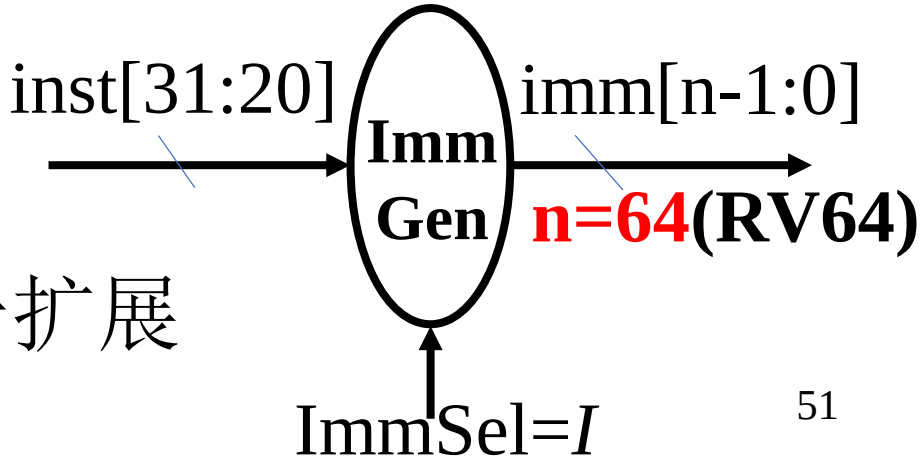
| imm[11:5] | imm[4:0] | rs1 | func3 | rd | opcode |
|---|---|---|---|---|---|
| 1111110 | 01110 | 00001 | 000 | 01111 | 0010011 |



# I型指令的立即数生成

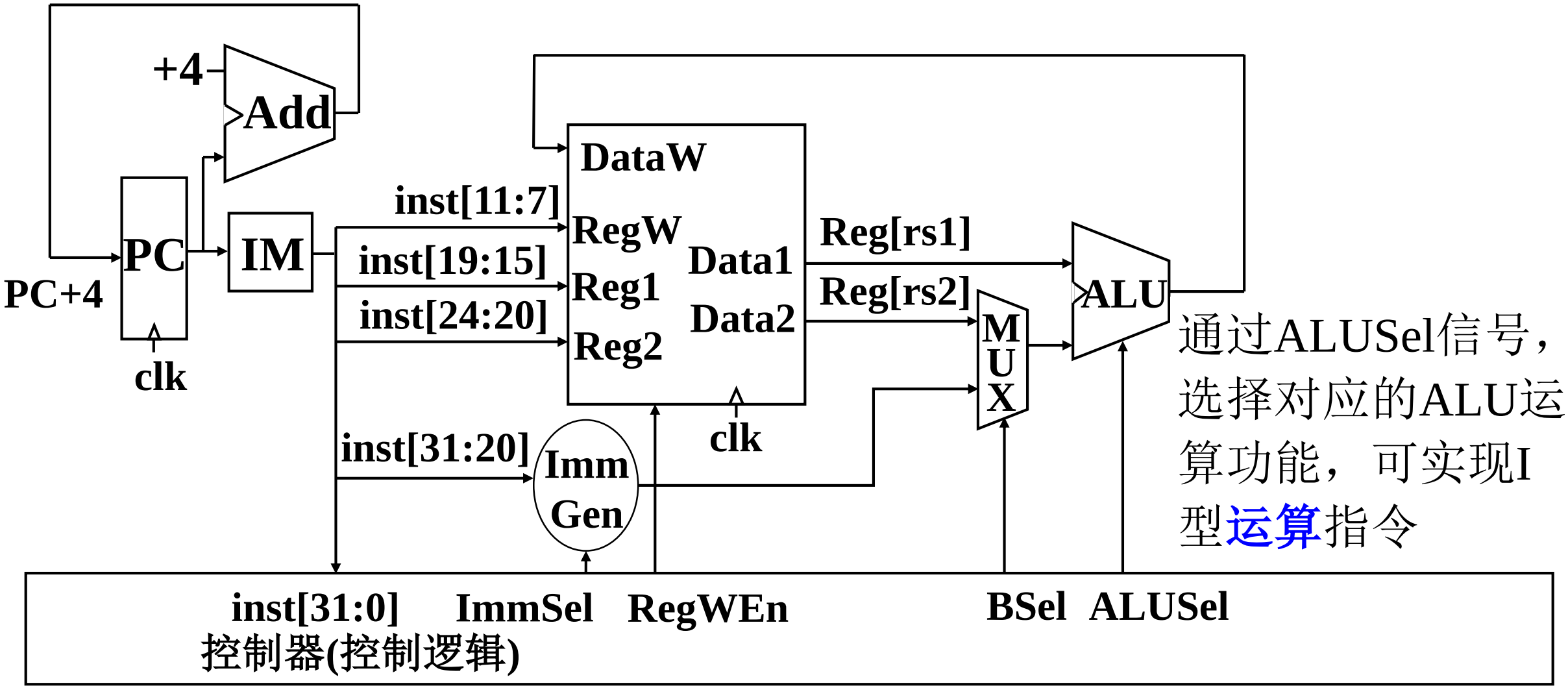



- 指令12位立即数字段复制到立即数的低12位
- 通过将指令的最高比特位复制填充到




立即数除了低12位之外的高位完成符号扩展

# R型和I型共用的数据通路




通过ALUSel信号，选择对应的ALU运算功能，可实现I型运算指令

# 实现ld指令（load dword）

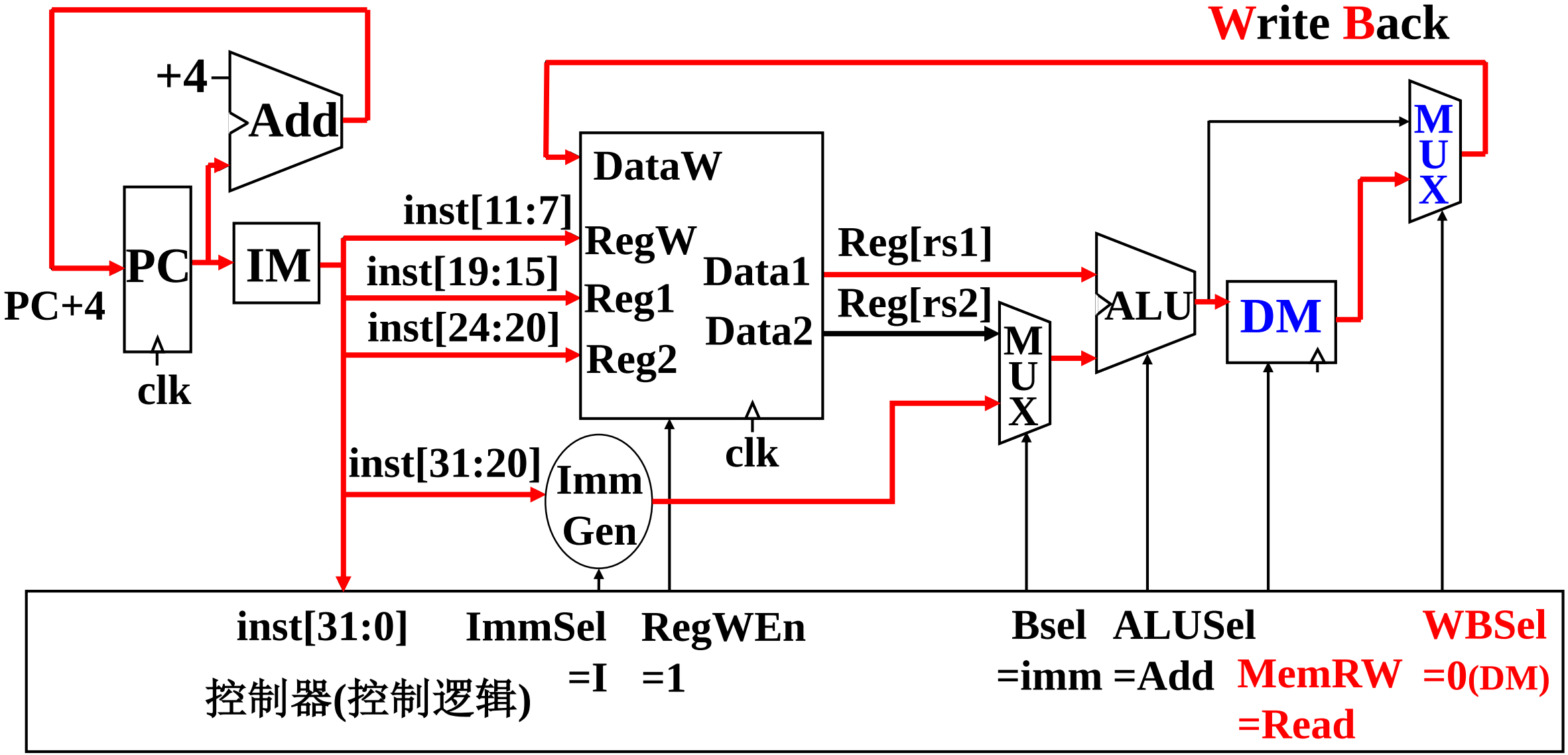
| 31 | | | | | 0 |
|---|---|---|---|---|---|
| imm[11:5] | imm[4:0] | rs1 | func3 | rd | opcode |

ld x14, 8(x2)

| 0000000 | 01000 | 00010 | 011 | 01110 | 0000011 |
|---|---|---|---|---|---|

- R[rs1]为基地址，加上立即数，得到目标访问地址
  - 与addi类似，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


Write Back
+4
Add
PC
IM
PC+4
clk
inst[11:7]
inst[19:15]
inst[24:20]
DataW
RegW
Reg1
Reg2
Data1
Data2
clk
Reg[rs1]
Reg[rs2]
inst[31:20]
Imm
Gen
M
U
X
ALU
DM
M
U
X
inst[31:0]
ImmSel
=I
RegWEn
=1
Bsel
=imm
ALUSel
=Add
MemRW
=Read
WBSel
=0(DM)
控制器(控制逻辑)

# RISC-V 访存装载指令

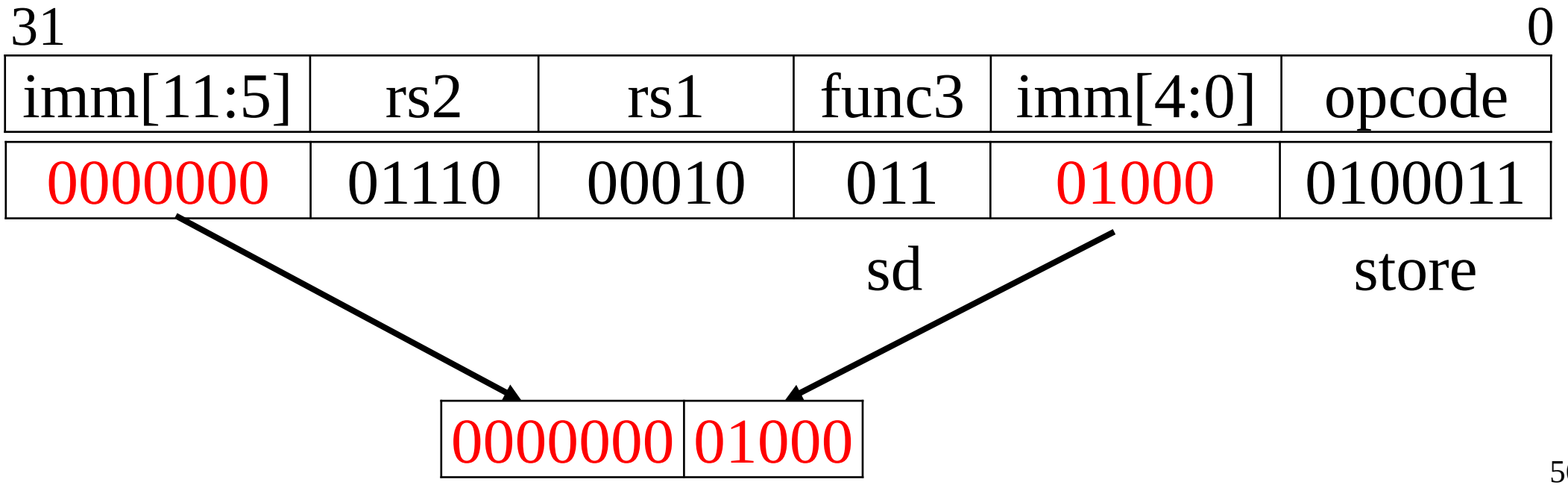
| | | | | | |
|---|---|---|---|---|---|
| imm[11:0] | rs1 | 000 | rd | 0000011 | lb |
| imm[11:0] | rs1 | 001 | rd | 0000011 | lh |
| imm[11:0] | rs1 | 010 | rd | 0000011 | lw |
| imm[11:0] | rs1 | 011 | rd | 0000011 | ld |
| imm[11:0] | rs1 | 100 | rd | 0000011 | lbu |
| imm[11:0] | rs1 | 101 | rd | 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、字等不同大小的数据
  - 写回到寄存器前，进行符号扩展或零扩展

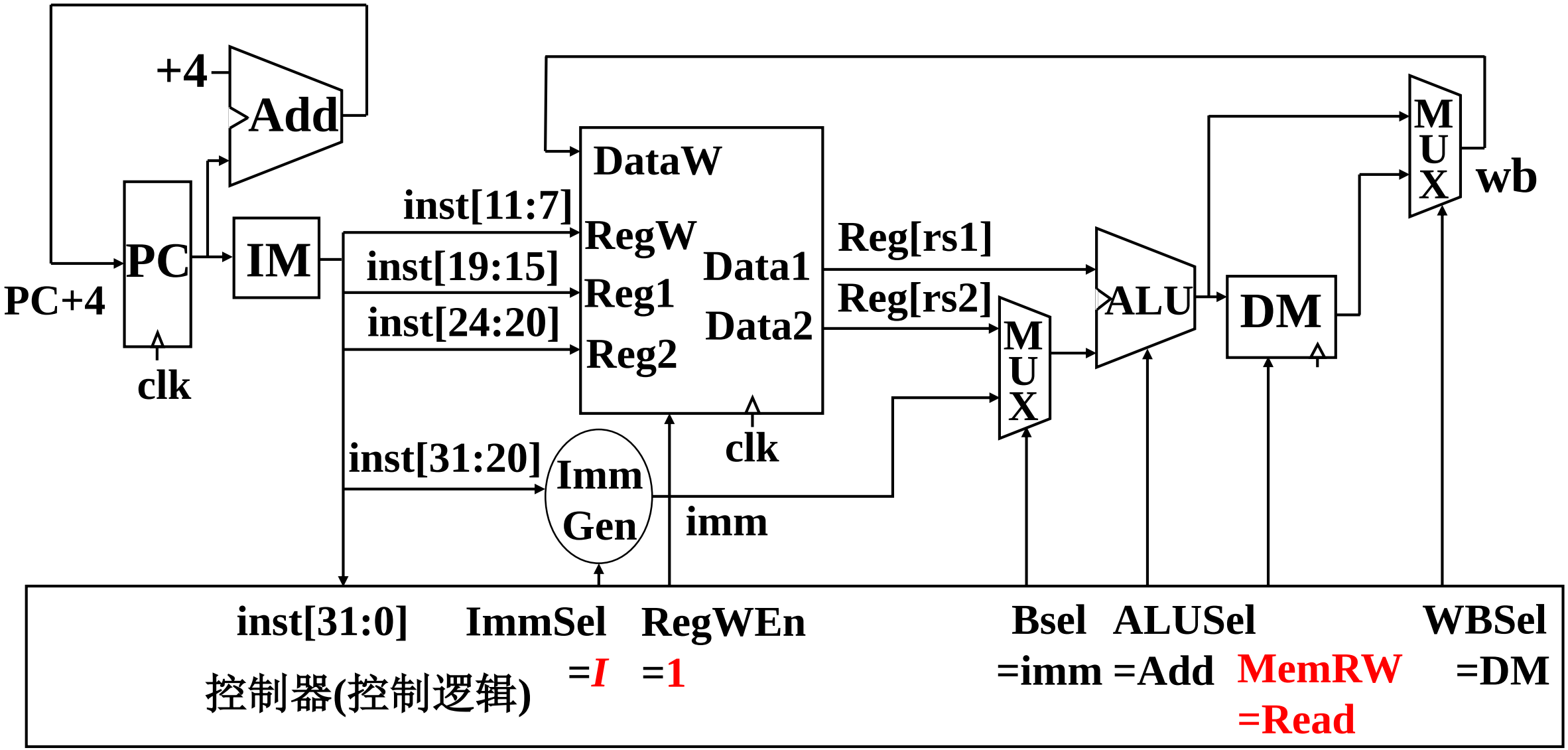
# 实现S型指令——sd

- 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

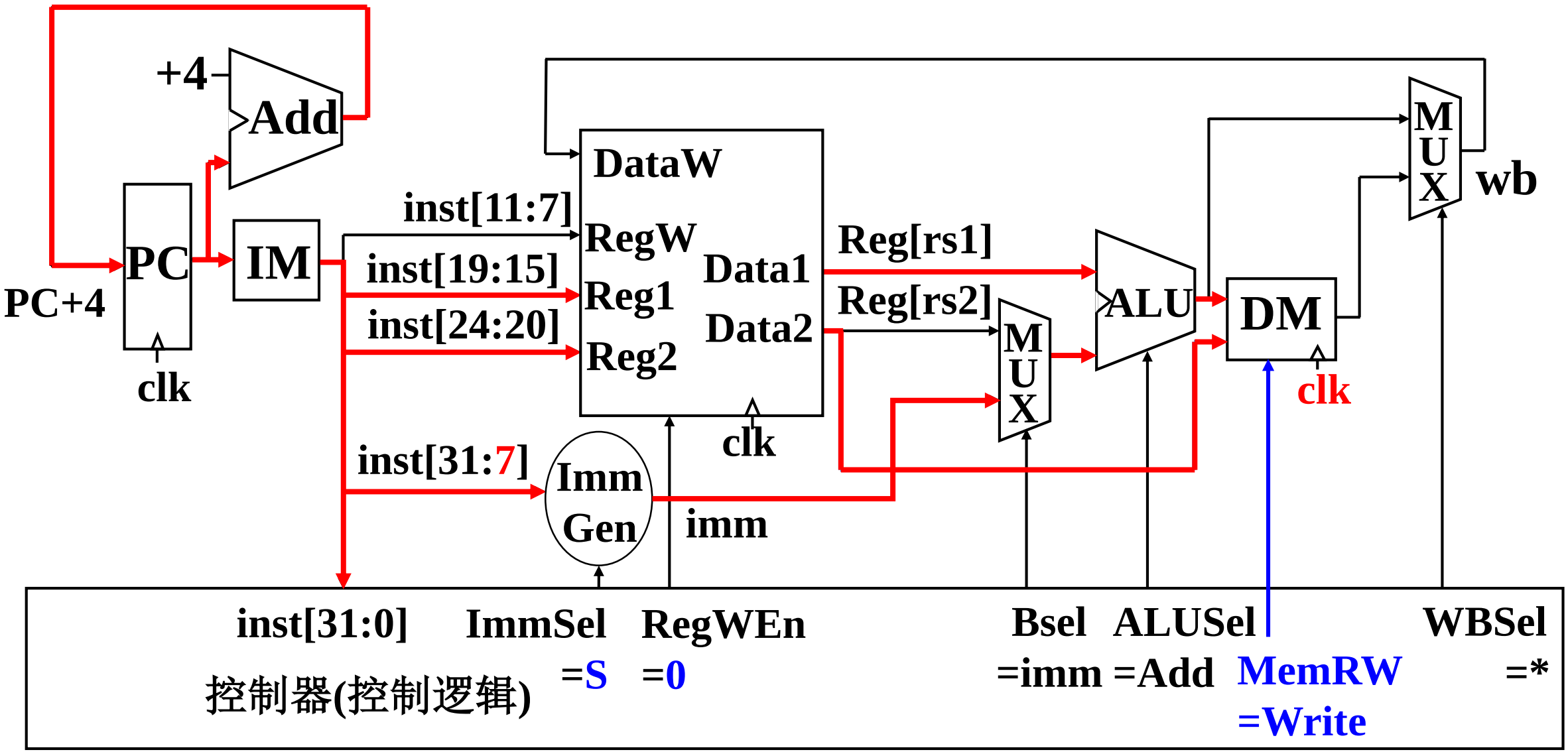
sd x14, 8(x2)



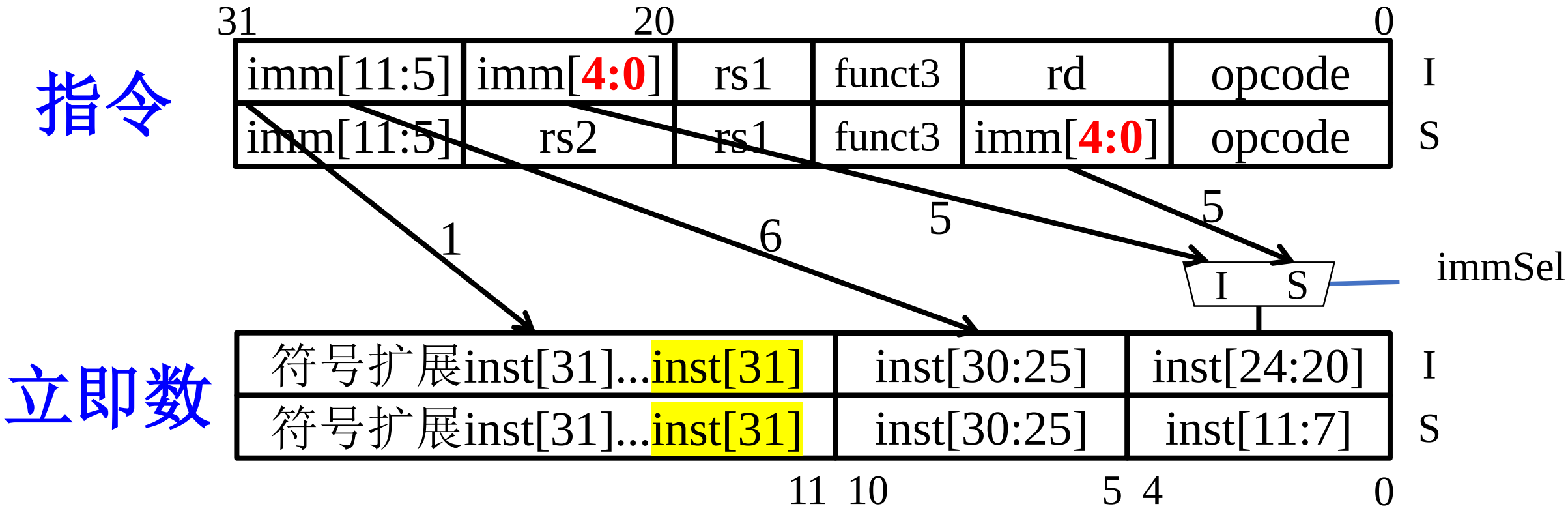
# （回顾对比）ld指令的数据通路



# sd指令的数据通路

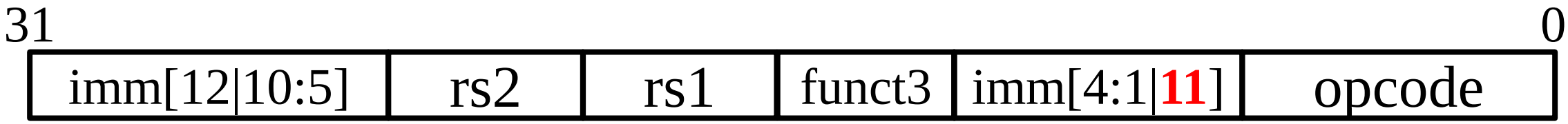

+4
Add
PC
IM
PC+4
clk
inst[11:7]
inst[19:15]
inst[24:20]
DataW
RegW
Reg1
Reg2
Data1
Data2
clk
Reg[rs1]
Reg[rs2]
M
U
X
ALU
DM
clk
M
U
X
wb
inst[31:7]
Imm
Gen
imm
inst[31:0]
ImmSel
=S
RegWEn
=0
控制器(控制逻辑)
Bsel
=imm
ALUSel
=Add
MemRW
=Write
WBSel
=*

# I型指令和S型指令中的立即数生成(ImmGen)




- 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

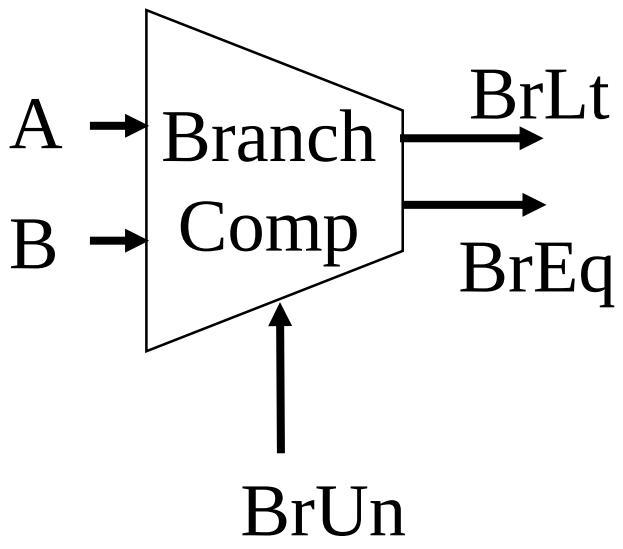
# B型指令的数据通路

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

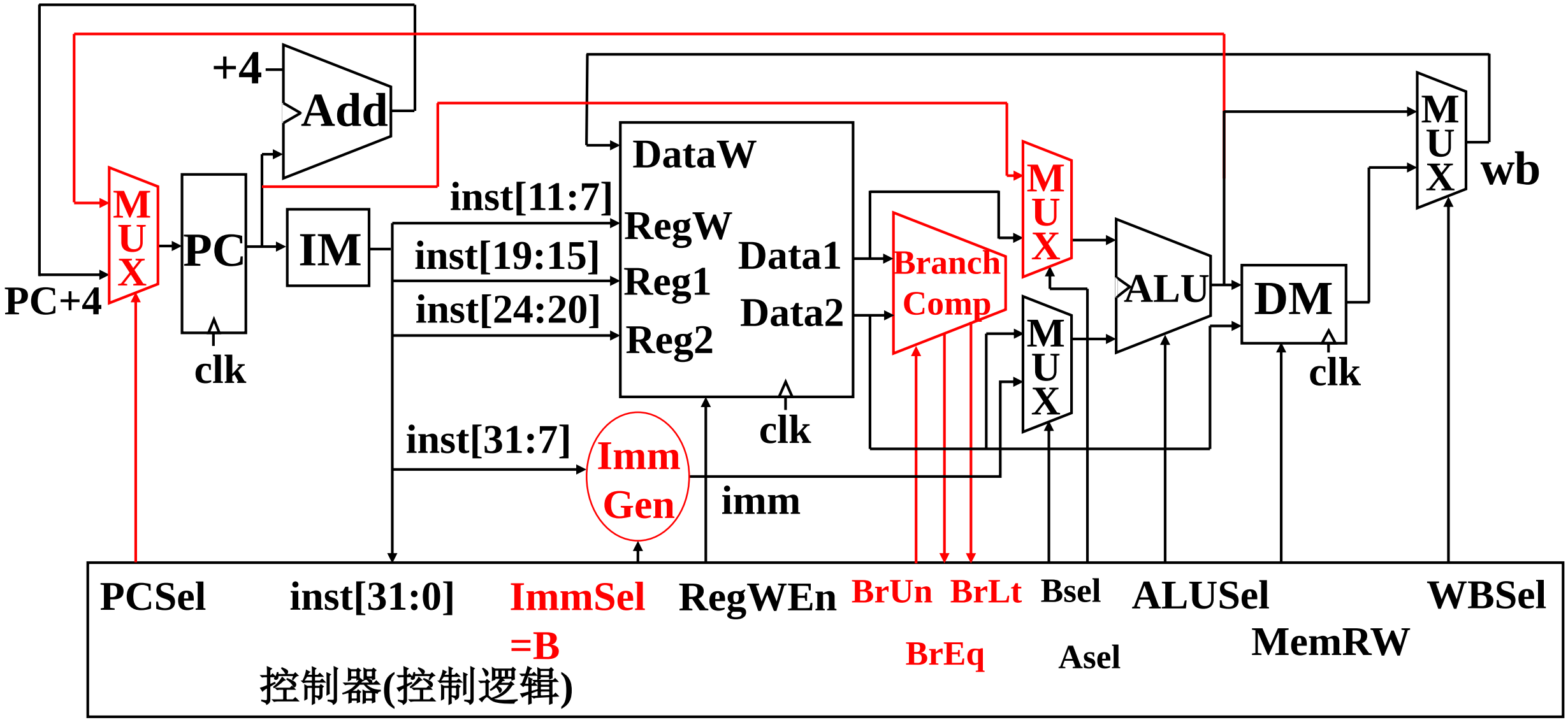
# 分支跳转比较器

- 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

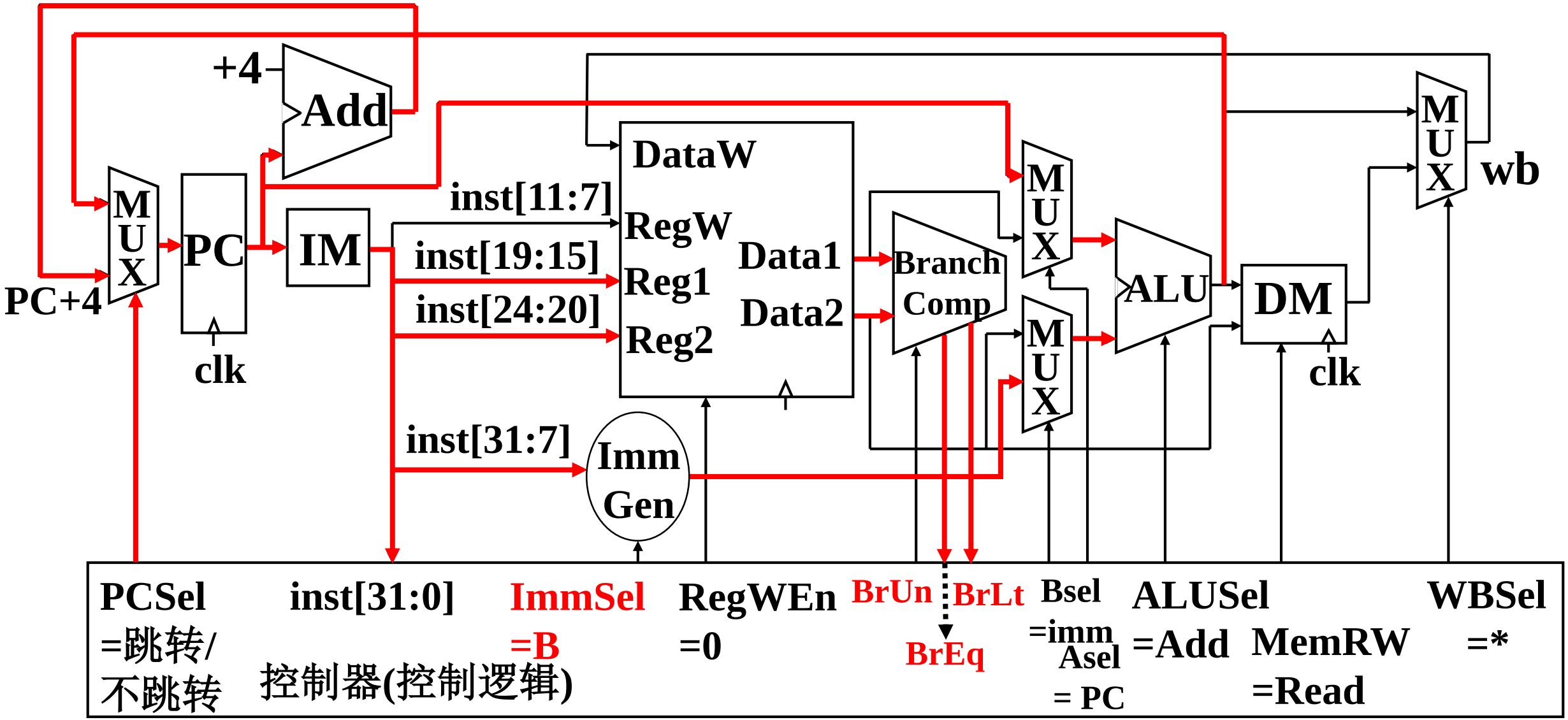
  输入 BrUn = 0时，进行有符号比较
- 对于bge，可以根据BrLt信号取反判断A ≥ B



# B型指令的数据通路


+4
Add
MUX
PC+4
PC
clk
IM
inst[11:7]
inst[19:15]
inst[24:20]
DataW
RegW
Reg1
Reg2
Data1
Data2
clk
inst[31:7]
Imm Gen
imm
Branch Comp
MUX
MUX
ALU
DM
clk
MUX
wb
PCSel
inst[31:0]
ImmSel =B
RegWEn
BrUn
BrLt
BrEq
Bsel
Asel
ALUSel
MemRW
WBSel
控制器(控制逻辑)

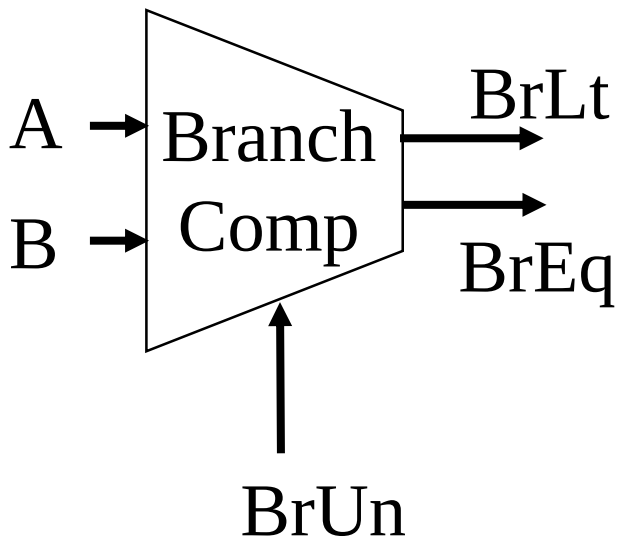
# B型指令的数据通路


+4
Add
M
U
X
PC+4
PC
clk
IM
inst[11:7]
inst[19:15]
inst[24:20]
DataW
RegW
Reg1
Reg2
Data1
Data2
Branch
Comp
M
U
X
ALU
DM
clk
M
U
X
wb
inst[31:7]
Imm
Gen
PCSel
=跳转/
不跳转
inst[31:0]
控制器(控制逻辑)
ImmSel
=B
RegWEn
=0
BrUn
BrLt
BrEq
Bsel
=imm
Asel
= PC
ALUSel
=Add
MemRW
=Read
WBSel
=*

# 分支跳转比较器

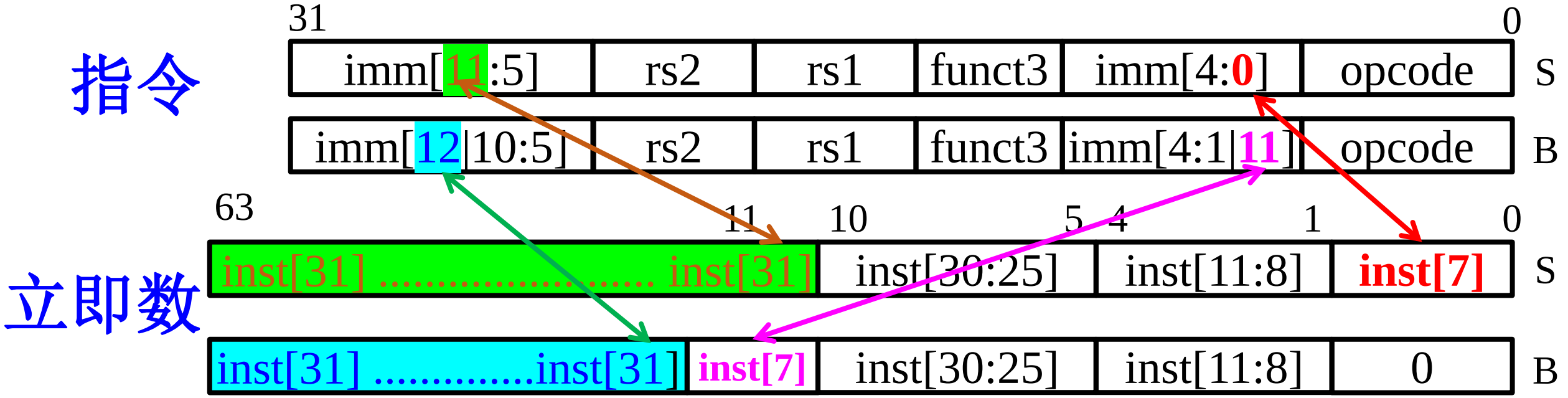
- 当A = B时，输出 BrEq = 1，否则为0
- 当A < B时，输出 BrLt = 1，否则为0
- 输入 BrUn = 1时，进行无符号比较

  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 ≥ B

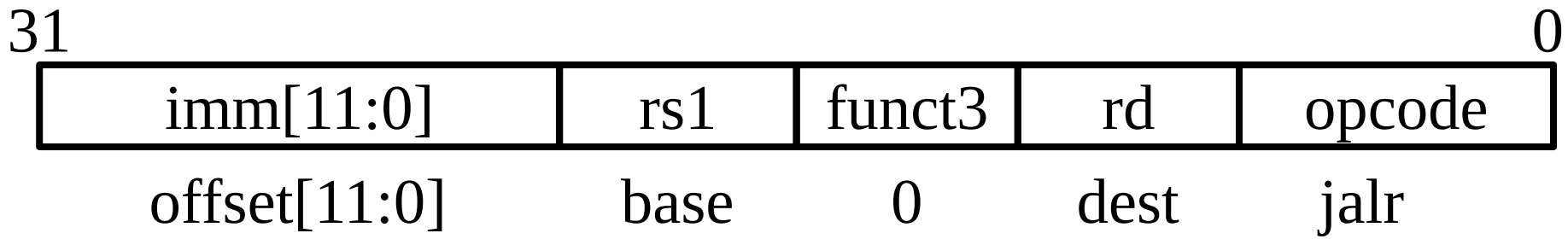


# B型指令立即数生成

- 与S型指令立即数生成方式类似
- 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

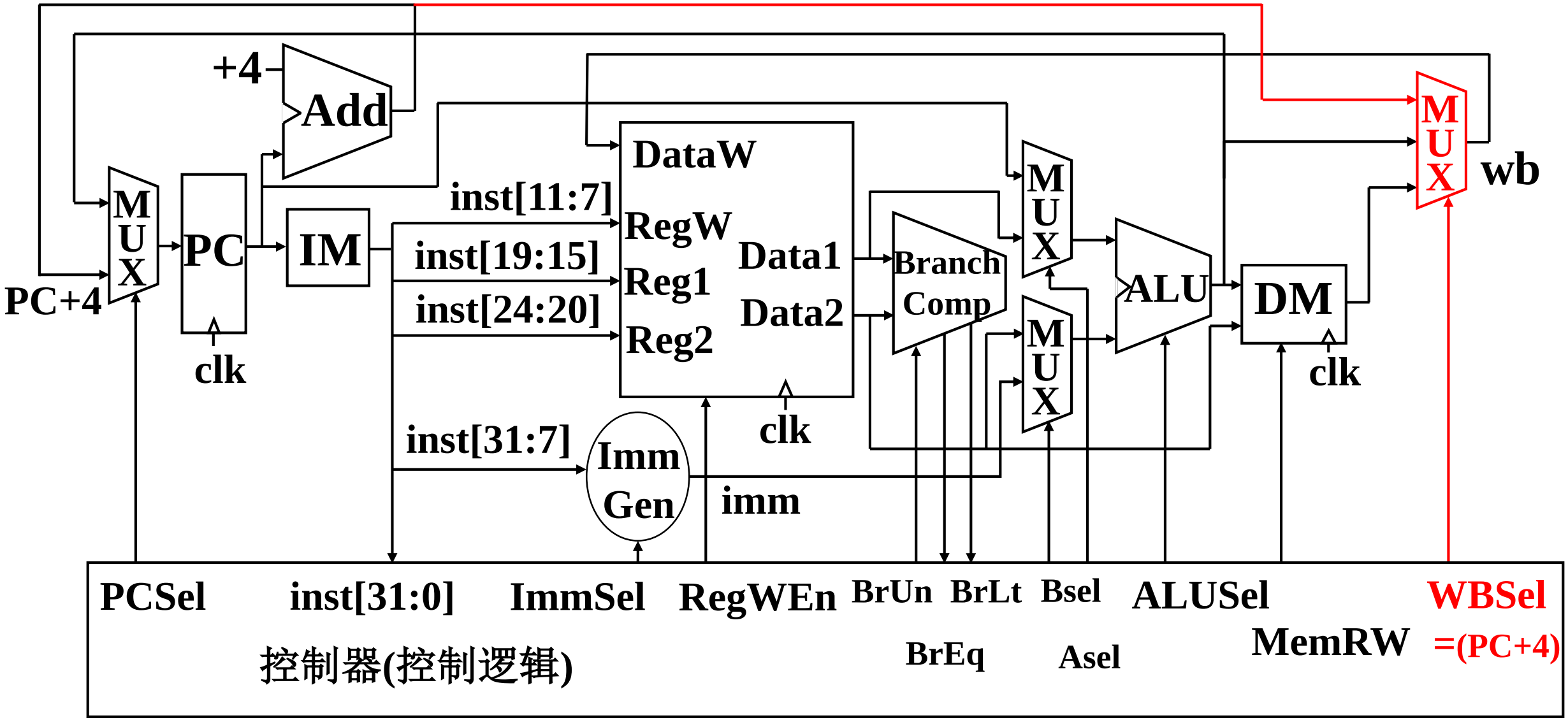


# I型指令——jalr

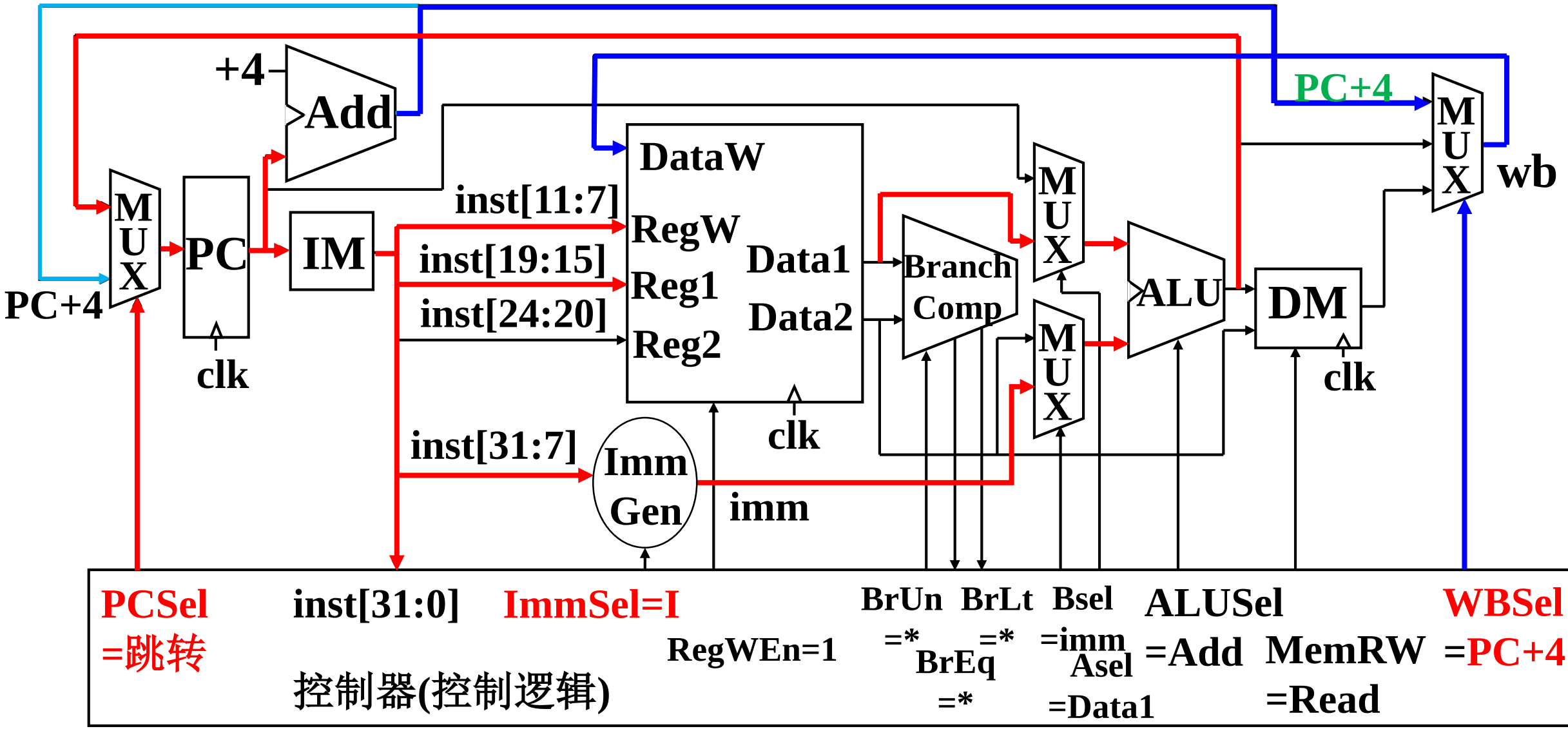



- jalr rd, offset(rs1)　#(jump and link register）
- 将 PC + 4 **写入rd**（返回地址）
- 设置 **PC = R[rs1] + offset**
- 立即数生成与I型算术和装载指令一样

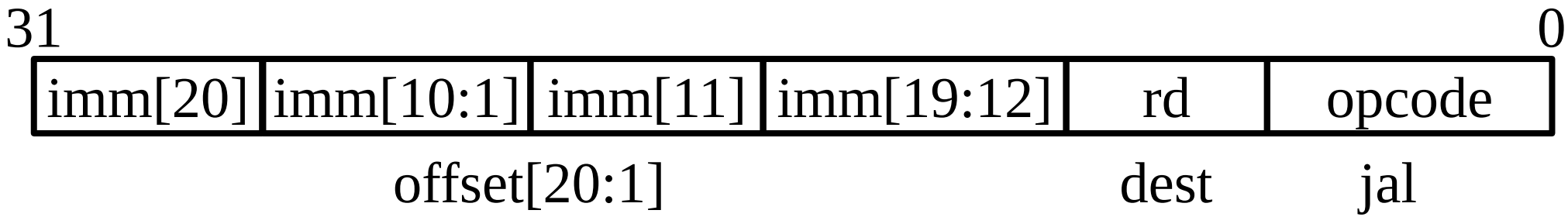
# jalr指令的数据通路



# jalr指令的数据通路

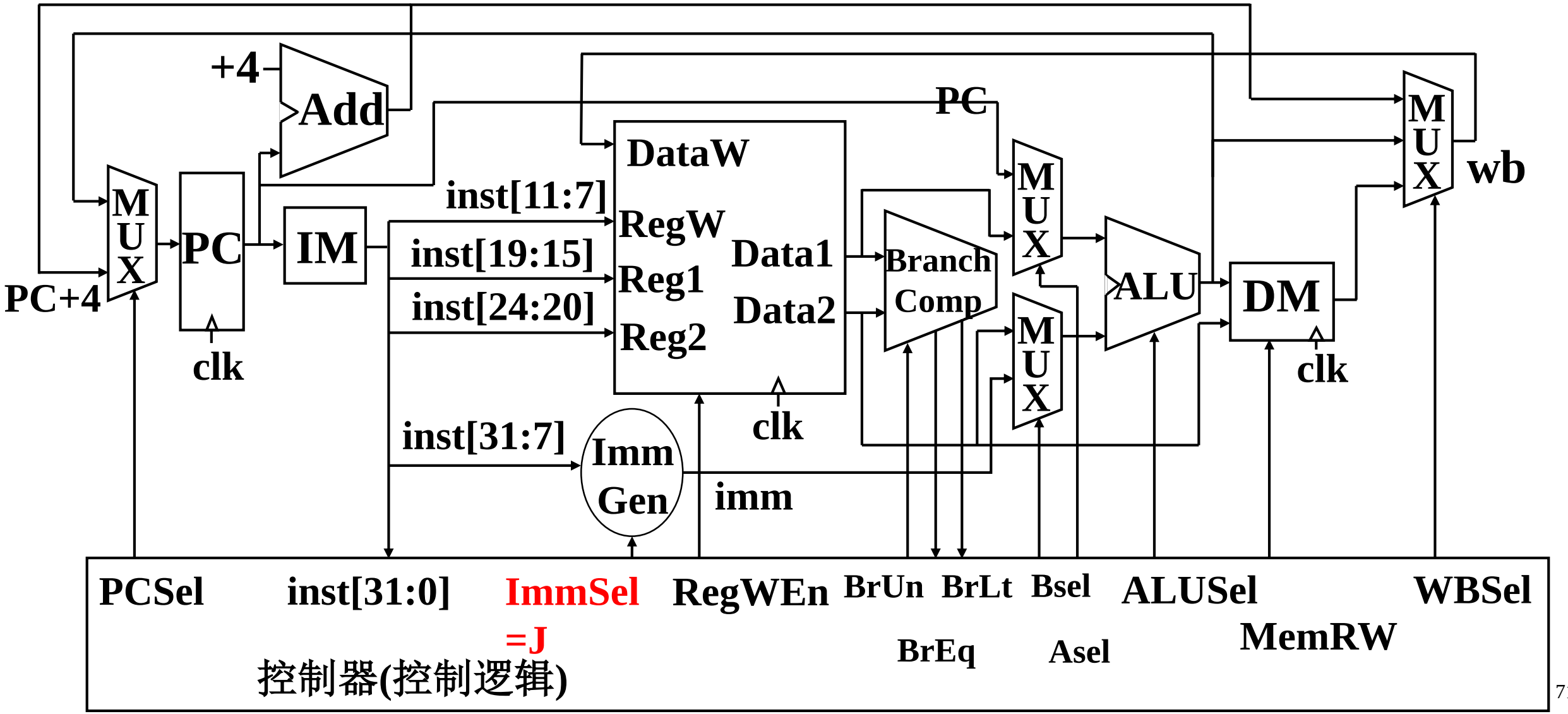


# J型指令——jal

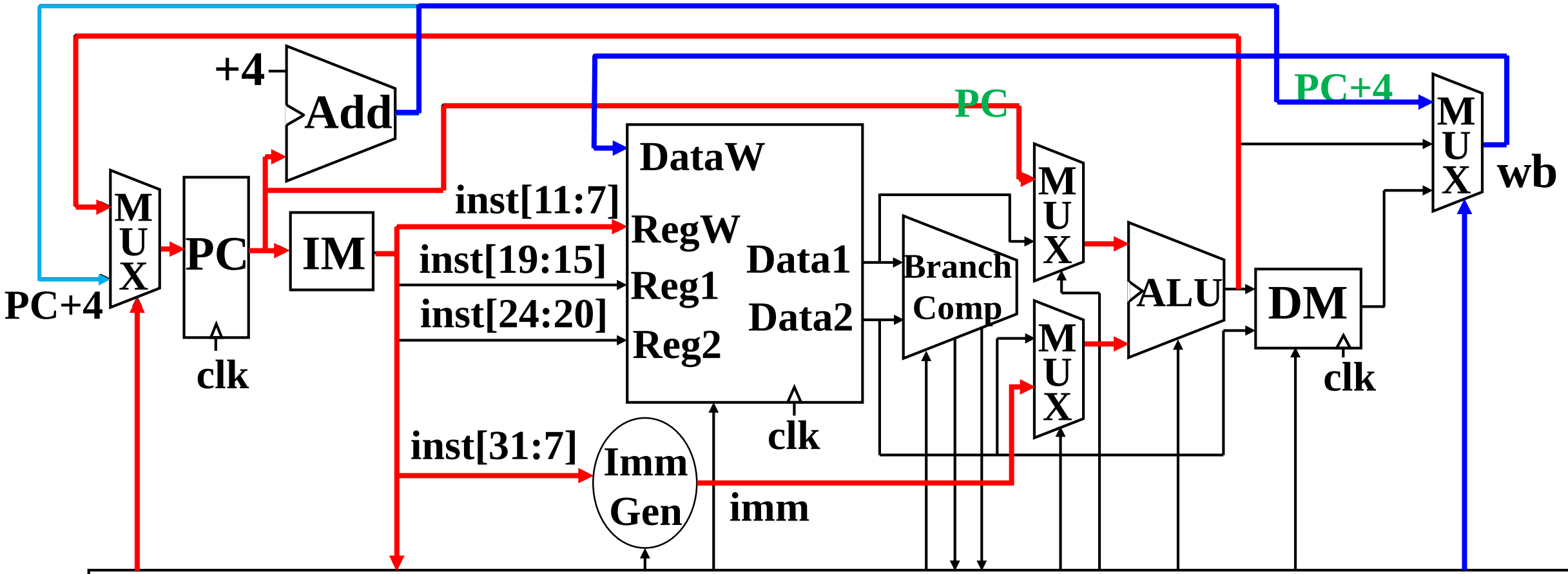



- 语法：jal rd, Label　(Label会被汇编器解析成offset)
- jal将 PC + 4 写入目的寄存器rd中
- 设置 PC = PC + offset
- 立即数字段20位，对应一个$\pm 2^{19}$的偏移量(以**2字节**为单位)

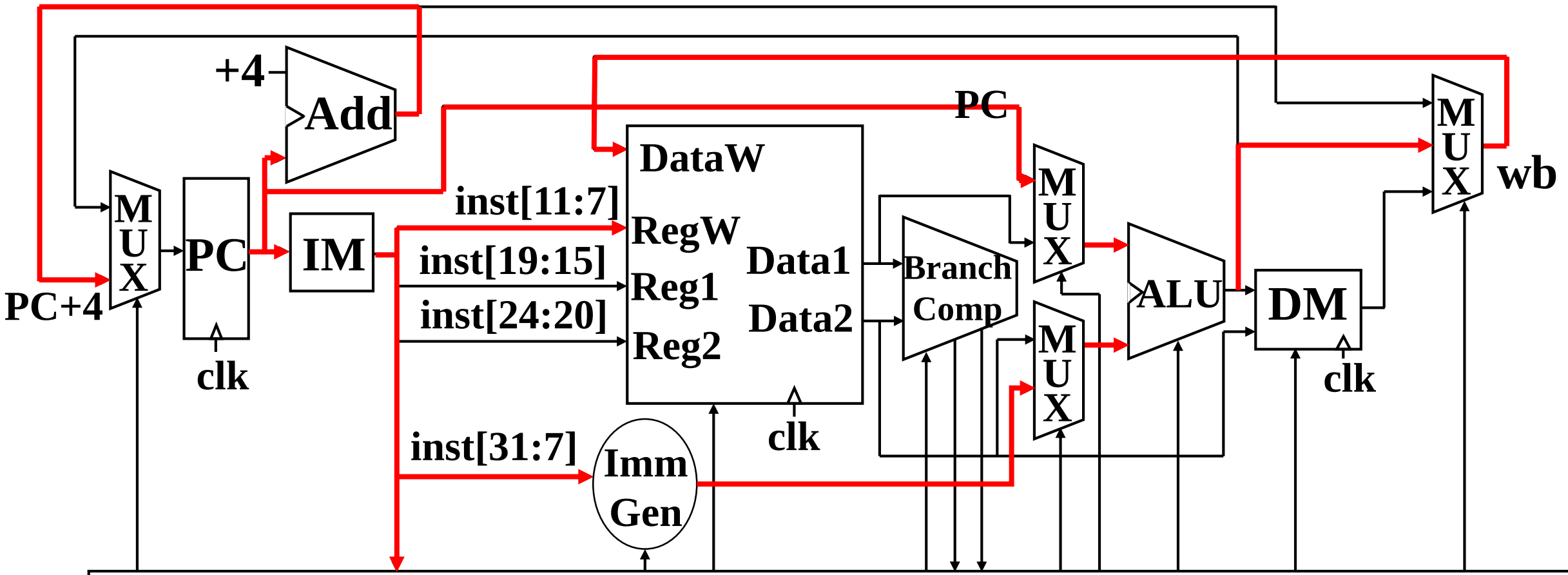
# jal指令的数据通路


+4
Add
PC
DataW
inst[11:7]
RegW
inst[19:15]
Reg1
Data1
inst[24:20]
Reg2
Data2
M
U
X
wb
PC+4
PC
IM
clk
Branch
Comp
ALU
DM
clk
inst[31:7]
Imm
Gen
imm
clk
PCSel
inst[31:0]
ImmSel
=J
RegWEn
BrUn
BrLt
Bsel
ALUSel
WBSel
BrEq
Asel
MemRW
控制器(控制逻辑)

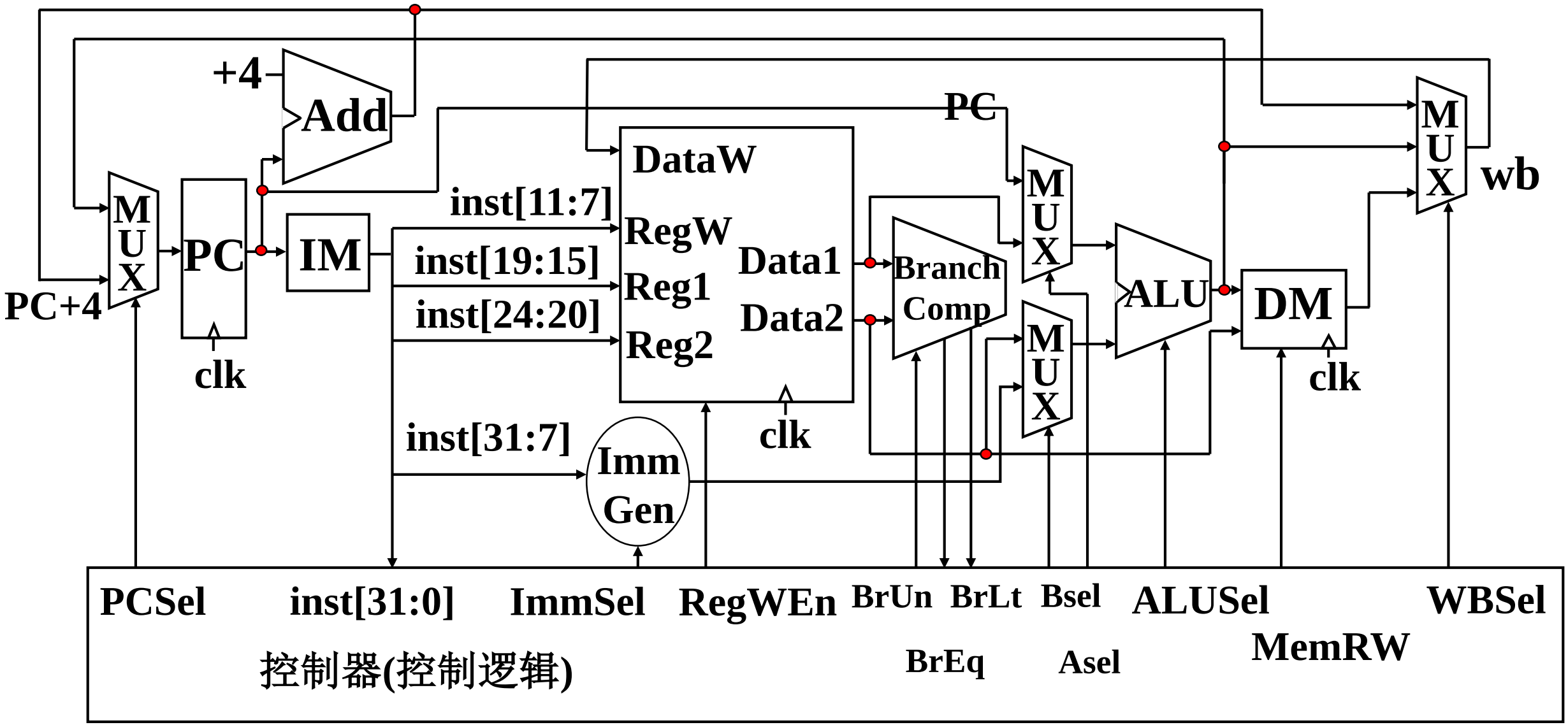
# jal指令的数据通路


+4
Add
PC
PC+4
M
U
X
wb
DataW
RegW
Reg1
Reg2
Data1
Data2
inst[11:7]
inst[19:15]
inst[24:20]
PC+4
PC
clk
IM
Branch
Comp
ALU
DM
clk
clk
inst[31:7]
Imm
Gen
imm
PCSel
=跳转
inst[31:0]
控制器(控制逻辑)
ImmSel
=J
RegWEn
=1
BrUn
=*
BrLt
=*
BrEq
=*
Bsel
=imm
Asel
=PC
ALUSel
=Add
MemRW
=Read
WBSel
=PC+4

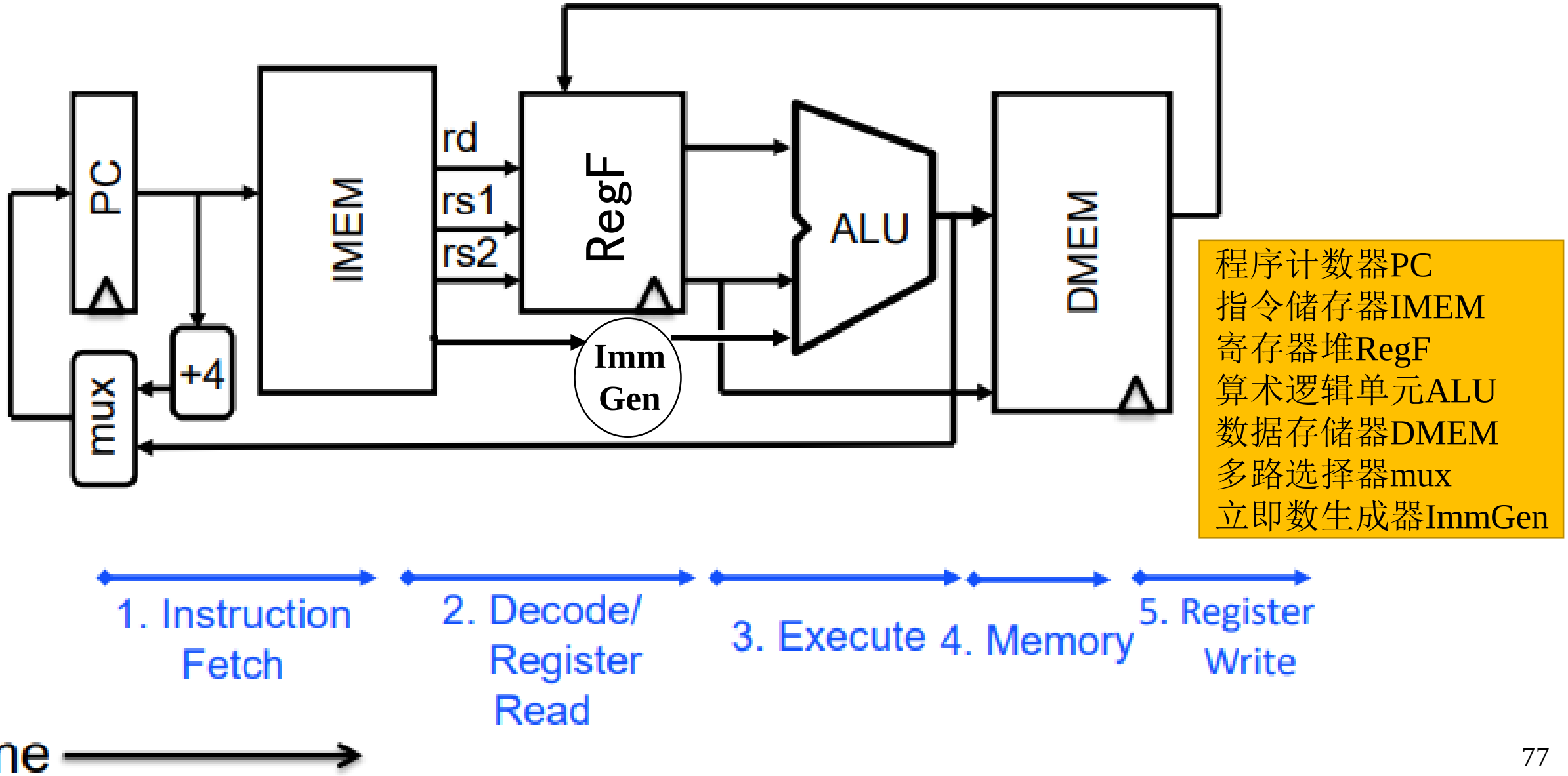
# U型指令auipc的数据通路



# RISC-V 的数据通路基础框架

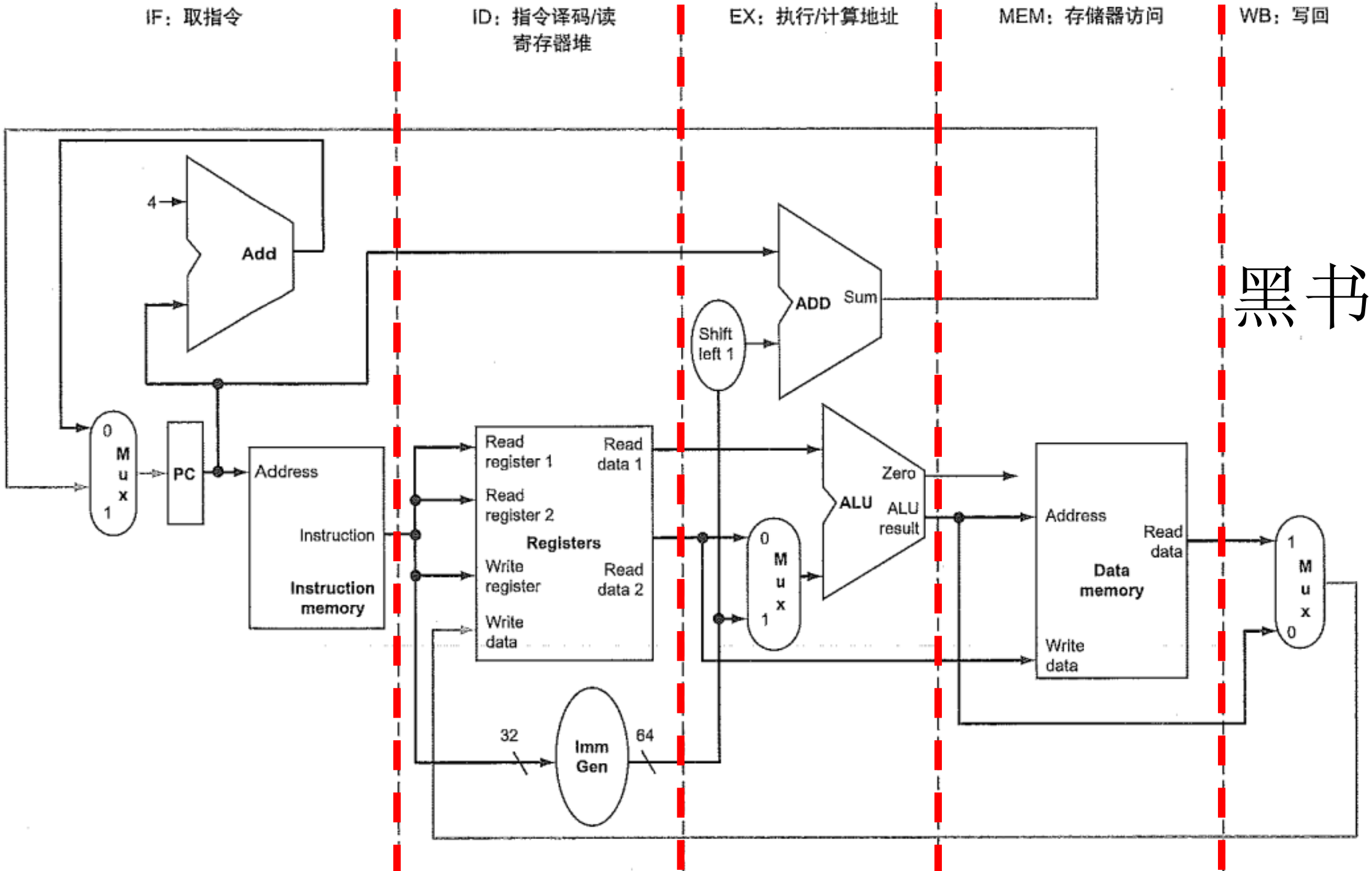


# 指令执行示意图（粗略图）


PC
IMEM
rd
rs1
rs2
RegF
ALU
DMEM
+4
mux
Imm
Gen
程序计数器PC
指令储存器IMEM
寄存器堆RegF
算术逻辑单元ALU
数据存储器DMEM
多路选择器mux
立即数生成器ImmGen
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)


IF：取指令
ID：指令译码/读寄存器堆
EX：执行/计算地址
MEM：存储器访问
WB：写回
4
Add
ADD
Sum
Shift left 1
0
M
u
x
1
PC
Address
Instruction
Instruction memory
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
ALU
Zero
ALU result
Address
Read data
Data memory
Write data
32
Imm Gen
64

黑书P194

# 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

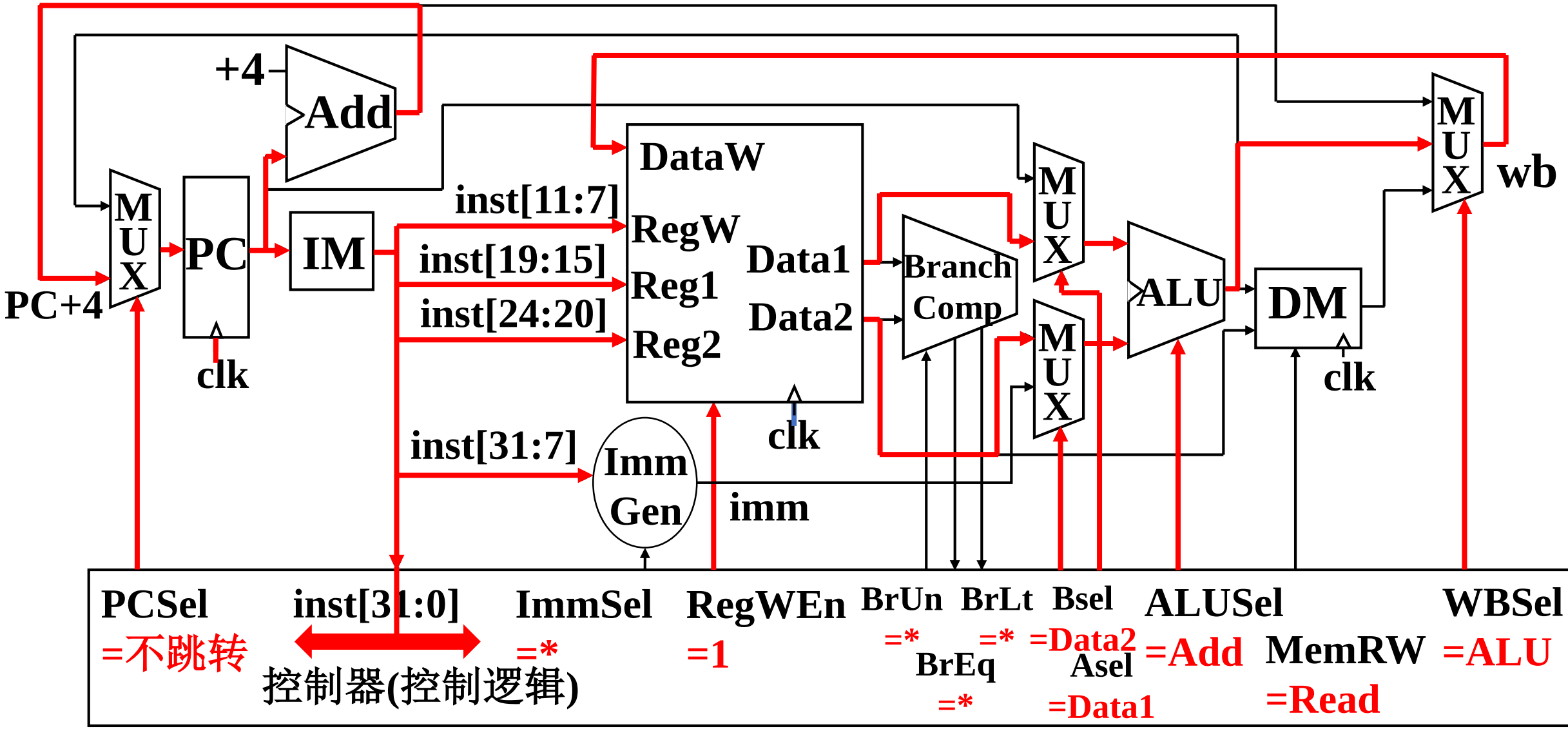
② 为数据通路选择合适的组件

③ 根据指令需求连接组件建立数据通路

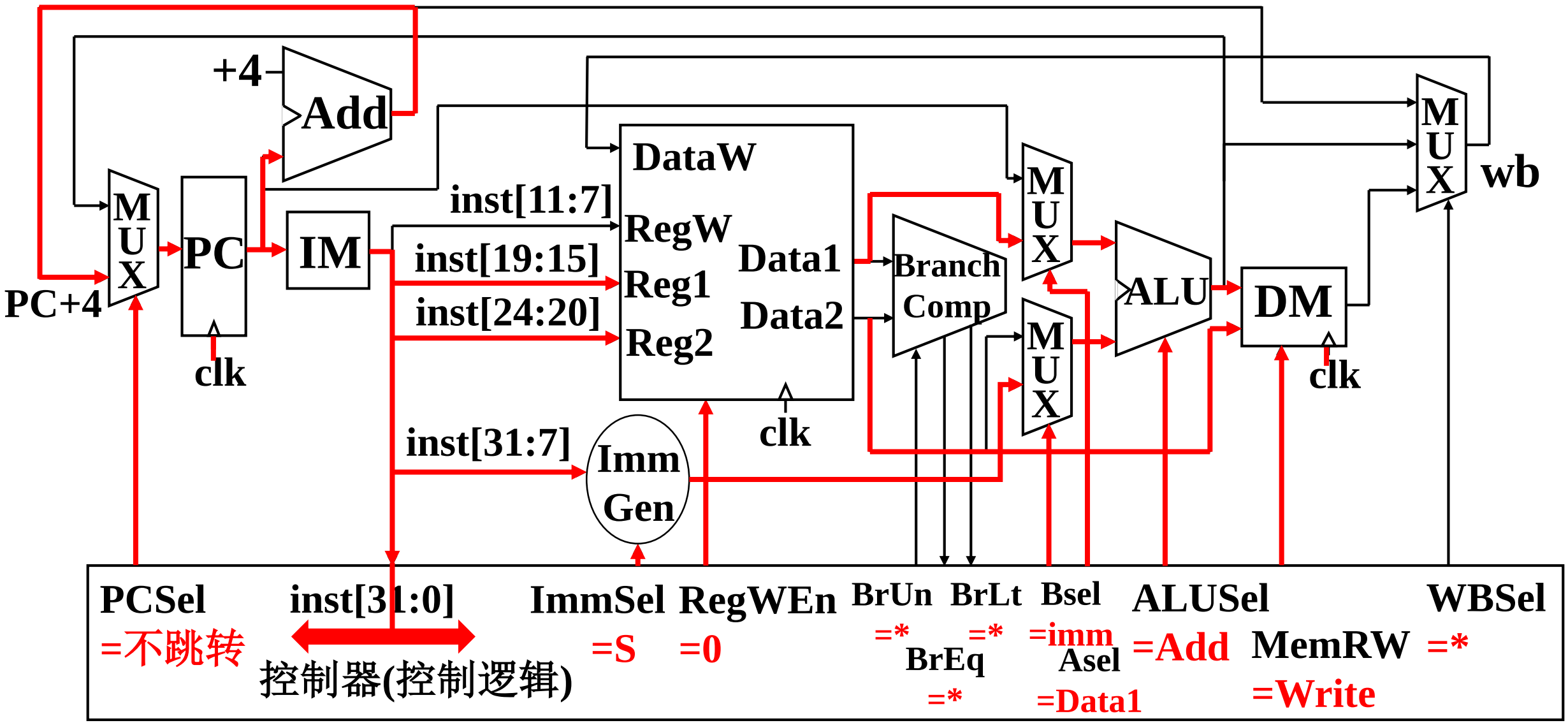
**④ 分析每条指令的实现，以确定控制信号**

⑤ 集成控制信号，形成完整的控制逻辑

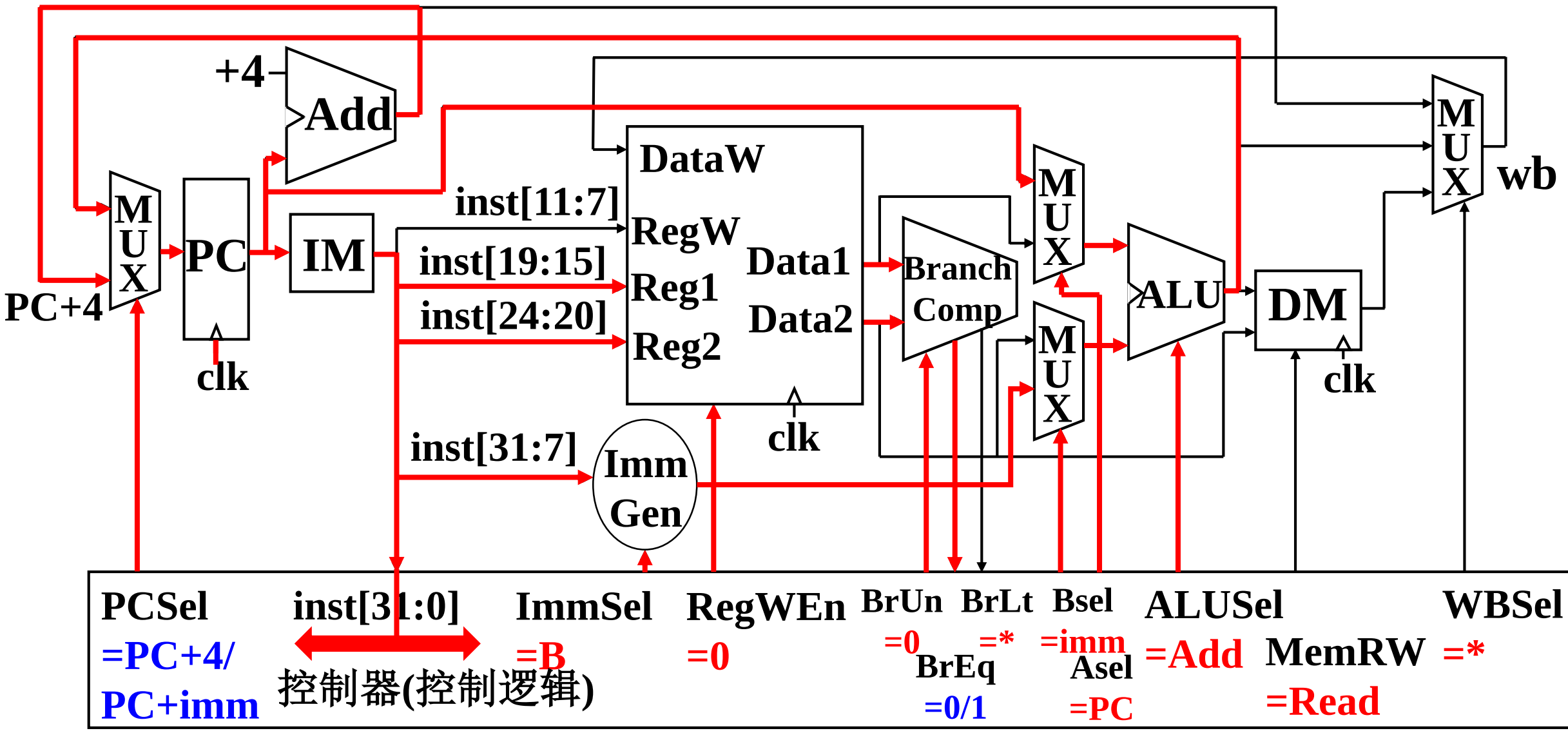
# add指令的执行


+4
Add
MUX
PC+4
PC
clk
IM
inst[11:7]
inst[19:15]
inst[24:20]
DataW
RegW
Reg1
Reg2
Data1
Data2
clk
inst[31:7]
Imm Gen
imm
Branch Comp
MUX
MUX
MUX
ALU
DM
clk
MUX
wb
PCSel
=不跳转
inst[31:0]
控制器(控制逻辑)
ImmSel
=*
RegWEn
=1
BrUn
=*
BrLt
=*
BrEq
=*
Bsel
=Data2
Asel
=Data1
ALUSel
=Add
MemRW
=Read
WBSel
=ALU

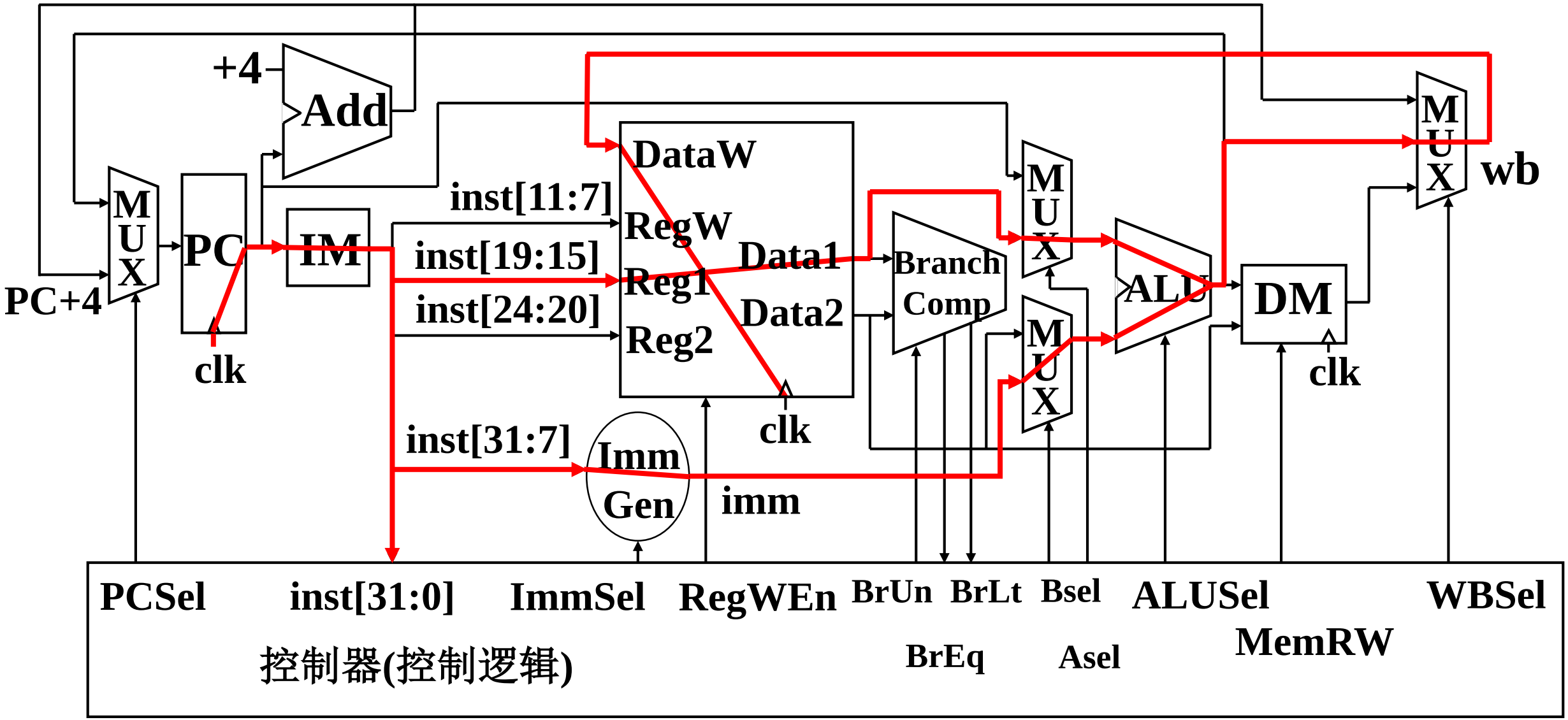
# sd指令的执行


+4
Add
MUX
PC+4
PC
clk
IM
inst[11:7]
inst[19:15]
inst[24:20]
DataW
RegW
Reg1
Reg2
Data1
Data2
clk
inst[31:7]
Imm
Gen
Branch
Comp
MUX
MUX
MUX
ALU
DM
clk
MUX
wb
PCSel
=不跳转
inst[31:0]
控制器(控制逻辑)
ImmSel
=S
RegWEn
=0
BrUn
=*
BrLt
=*
BrEq
=*
Bsel
=imm
Asel
=Data1
ALUSel
=Add
MemRW
=Write
WBSel
=*

# beq指令的执行

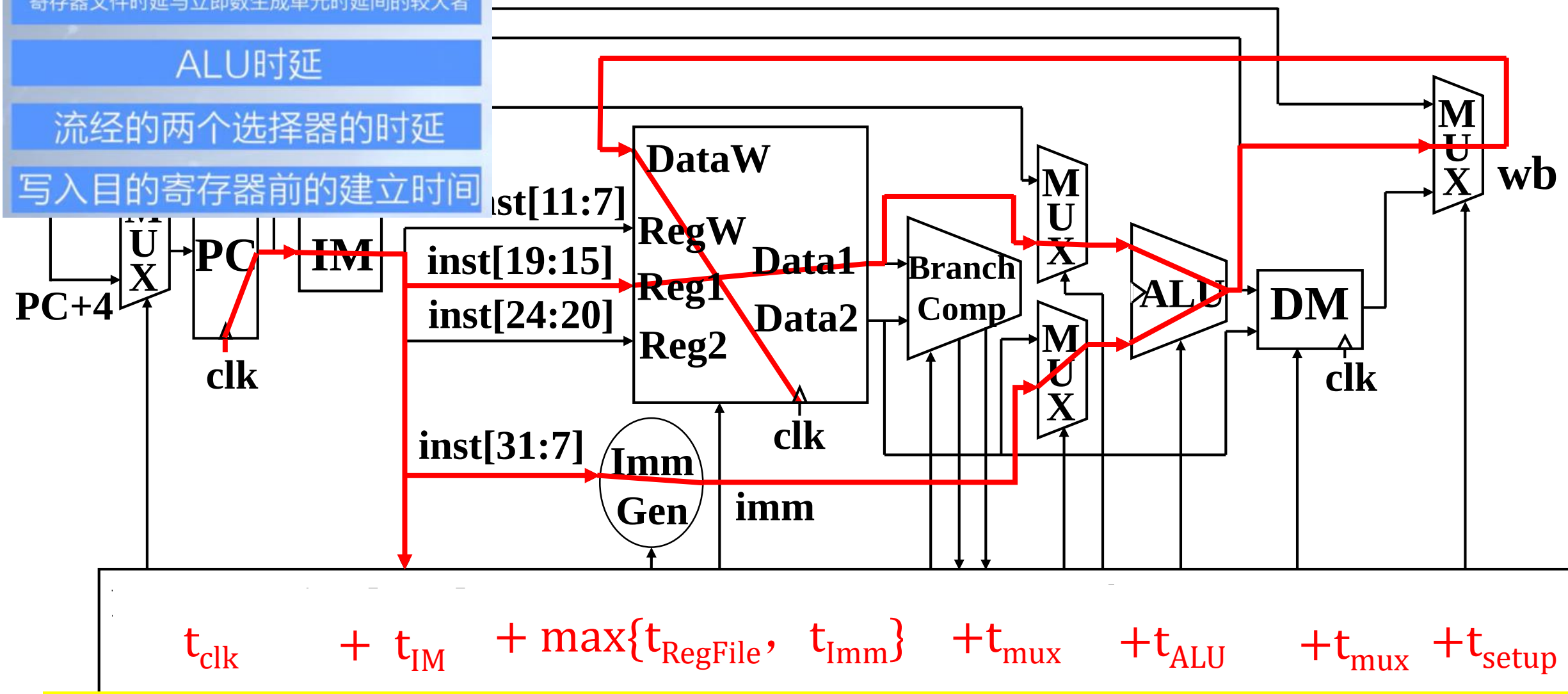

+4
Add
MUX
PC+4
PC
clk
IM
inst[11:7]
inst[19:15]
inst[24:20]
DataW
RegW
Reg1
Reg2
Data1
Data2
clk
inst[31:7]
Imm
Gen
Branch
Comp
MUX
MUX
MUX
ALU
DM
clk
MUX
wb
PCSel
=PC+4/
PC+imm
inst[31:0]
控制器(控制逻辑)
ImmSel
=B
RegWEn
=0
BrUn
=0
BrLt
=*
BrEq
=0/1
Bsel
=imm
Asel
=PC
ALUSel
=Add
MemRW
=Read
WBSel
=*

# 关键路径（这个数据通路执行哪类指令？）



# 关键路径

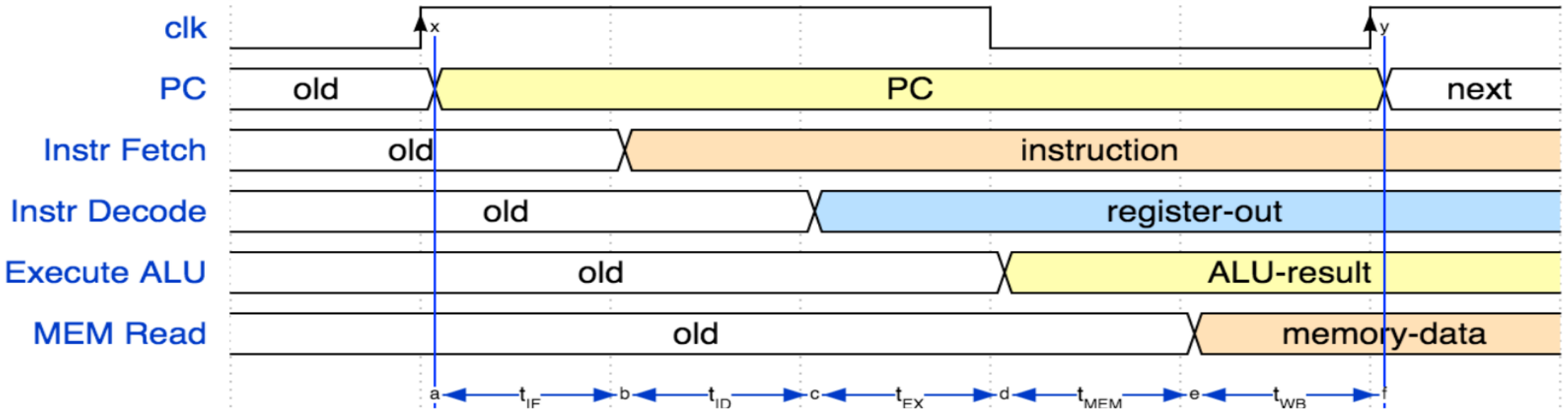
- PC寄存器的时延
- 指令存储器的时延
- 寄存器文件时延与立即数生成单元时延间的较大者
- ALU时延
- 流经的两个选择器的时延
- 写入目的寄存器前的建立时间




$$t_{clk} + t_{IM} + \max\{t_{RegFile}, t_{Imm}\} + t_{mux} + t_{ALU} + t_{mux} + t_{setup}$$

关键路径(Critical Path)是指从输入到输出的信号传输路径中，延迟最长的那条路径，最大延迟决定了单周期处理器的最大工作频率。

# 指令时延




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

问题：该表符合我们学过的那类指令？

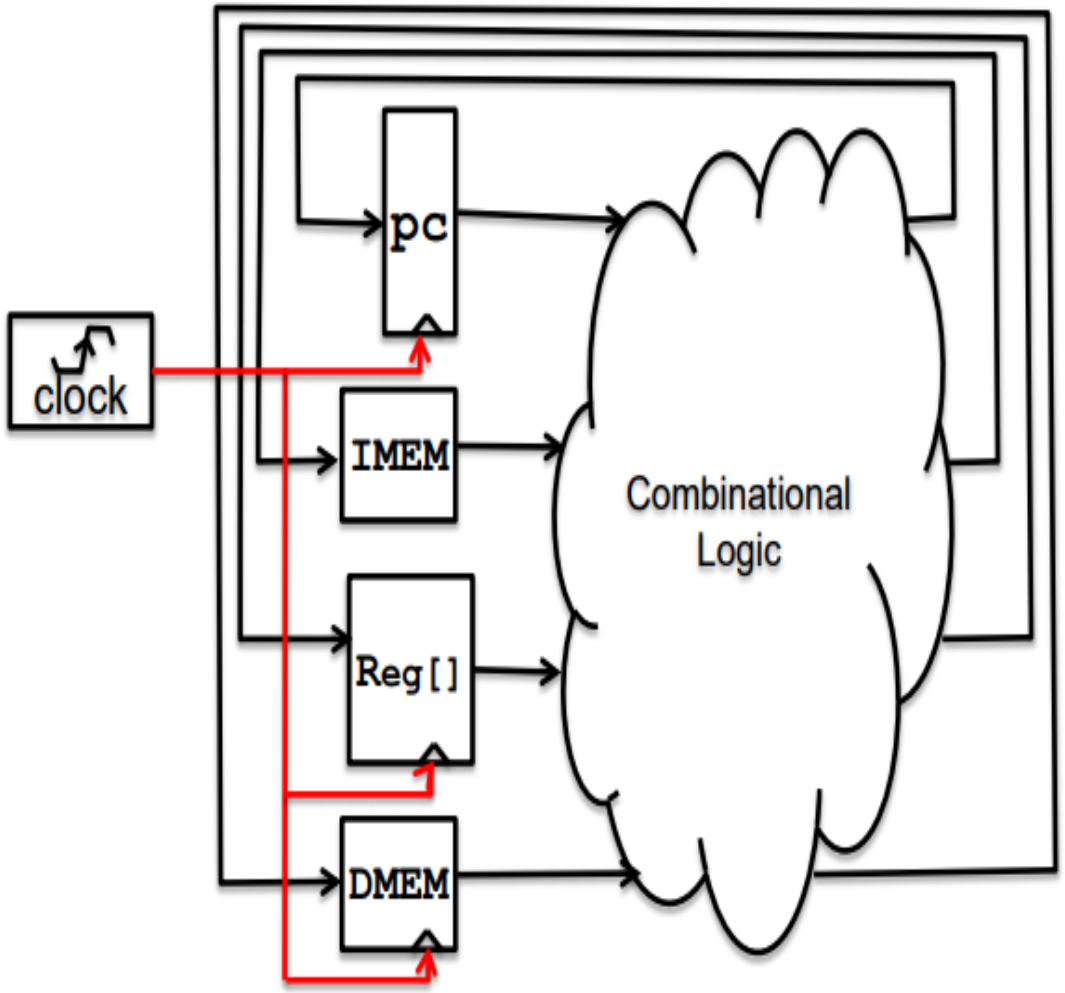
# 指令时延

| 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$f_{max,ALU}$=1/(800ps)=1.25GHz
  （假设一个时钟完成一条指令，也称为单周期CPU）
- 有些指令的某些阶段处于空闲状态
- 单个阶段最大时钟频率$f_{max,ALU}$=1/(200ps)=5GHz

# 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、更新对应寄存器，同时开始下一个时钟周期的执行操作。



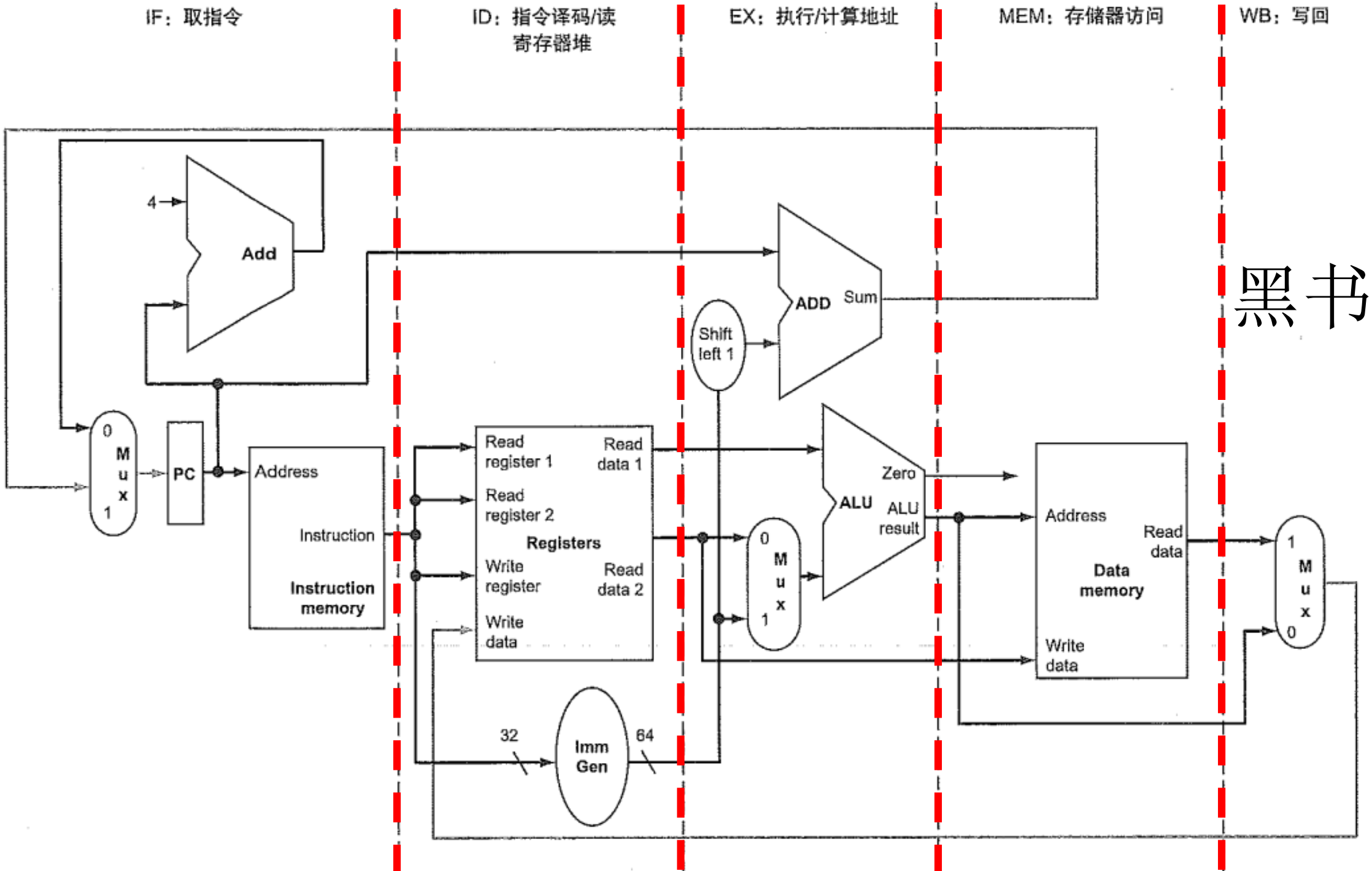
# 分阶段的数据通路——概述

- **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**

- **解决方案：** 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


IF：取指令
ID：指令译码/读寄存器堆
EX：执行/计算地址
MEM：存储器访问
WB：写回
4
Add
ADD
Sum
Shift left 1
0
M
u
x
1
PC
Address
Instruction
Instruction memory
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
ALU
Zero
ALU result
Address
Read data
Data memory
Write data
32
Imm Gen
64

黑书P194

单选题 1分

设置

单周期数据通路必须有独立的指令存储器(IM)和数据存储器(DM)，因为：

A RISC-V中指令与数据格式不同，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。

提交

# 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 | imm[**11**:5] | imm[4:0] | 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 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 | 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 |
| 000000 | shamt | rs1 | 101 | rd | 0010011 | srli |
| 010000 | shamt | rs1 | 101 | rd | 0010011 | srai |

# 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(Opcode)字段，主要用于区分不同类别的指令。

| | | | | | | |
|---|---|---|---|---|---|---|
| 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 |

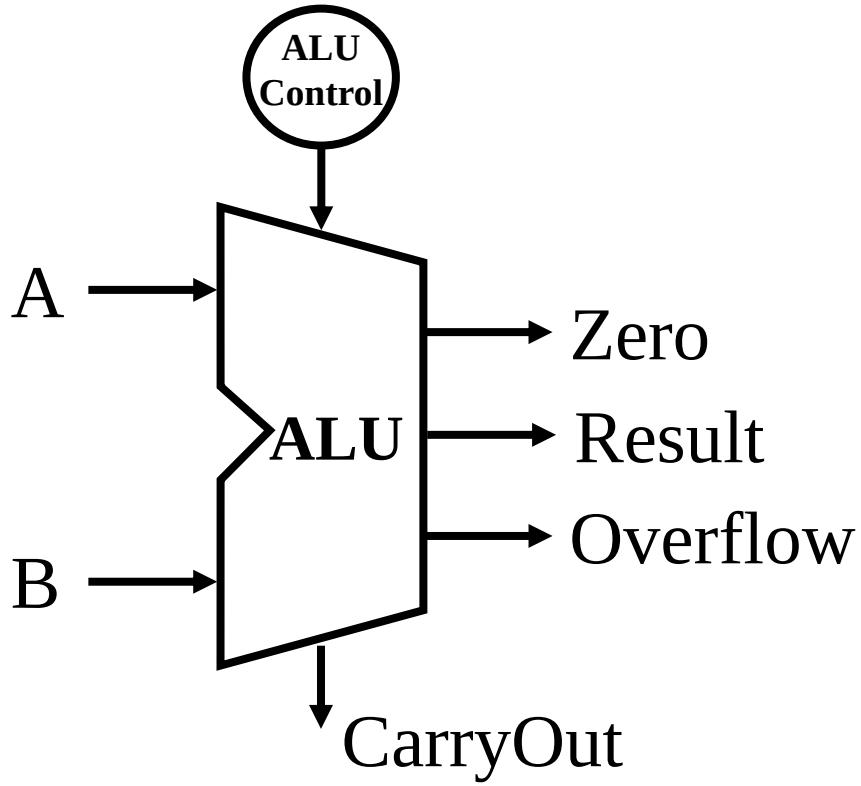
- BrUn = Inst[14] · Inst[13] · Inst[6:2]

# ALU功能需求（关注控制信号生成）

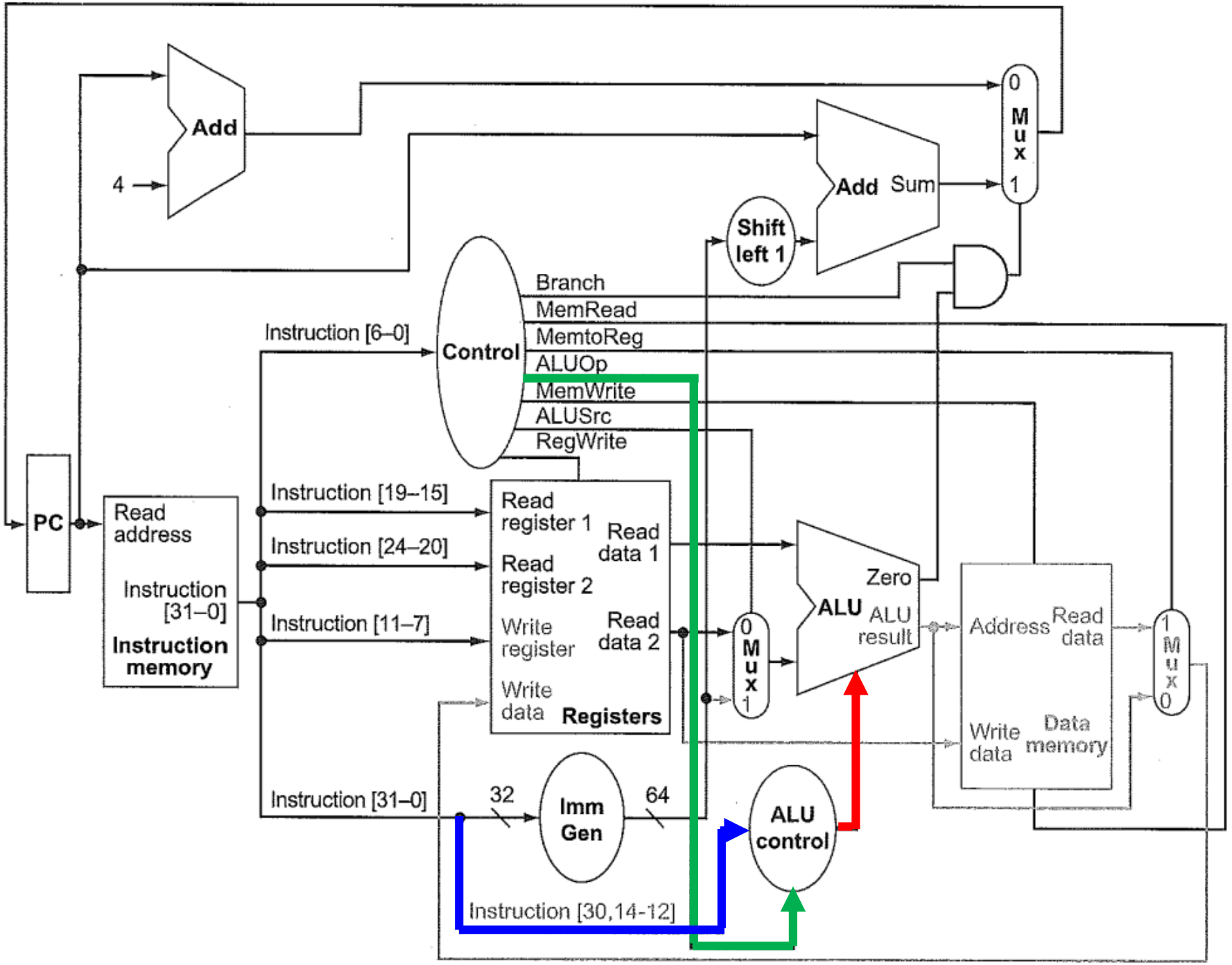
| ALU控制 | 操作 |
| --- | --- |
| 0000 | 与 |
| 0001 | 或 |
| 0010 | 加 |
| 0110 | 减 |
| 0111 | 小于则置位 |
| 1100 | 或非 |

**ALU控制信号如何生成？**

**（最直接的方式：Opcode、funt7、funct3）**



# 黑书P182图


Add
4
Add
Sum
Shift left 1
0
M
u
x
1
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [6–0]
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [19–15]
Instruction [24–20]
Instruction [11–7]
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
Zero
ALU
ALU result
Address
Read data
Write data
Data memory
Instruction [31–0]
32
Imm Gen
64
ALU control
Instruction [30,14-12]

# 根据opcode和ALUOp控制位设置ALU控制（黑书P175）

| opcode | ALUOp | 操作 | func7 | func3 | ALU期望行为 | ALU控制输入 |
|---|---|---|---|---|---|---|
| ld | 00 | load Dword | ******** | *** | add | 0010 |
| sd | 00 | store Dword | ******** | *** | add | 0010 |
| beq | 01 | branch if equal | ******** | *** | subtract | 0110 |
| R-type | 10 | add | 0000000 | 000 | add | 0010 |
| R-type | 10 | sub | 0100000 | 000 | subtract | 0110 |
| R-type | 10 | and | 0000000 | 111 | and | 0000 |
| R-type | 10 | or | 0000000 | 110 | or | 0001 |

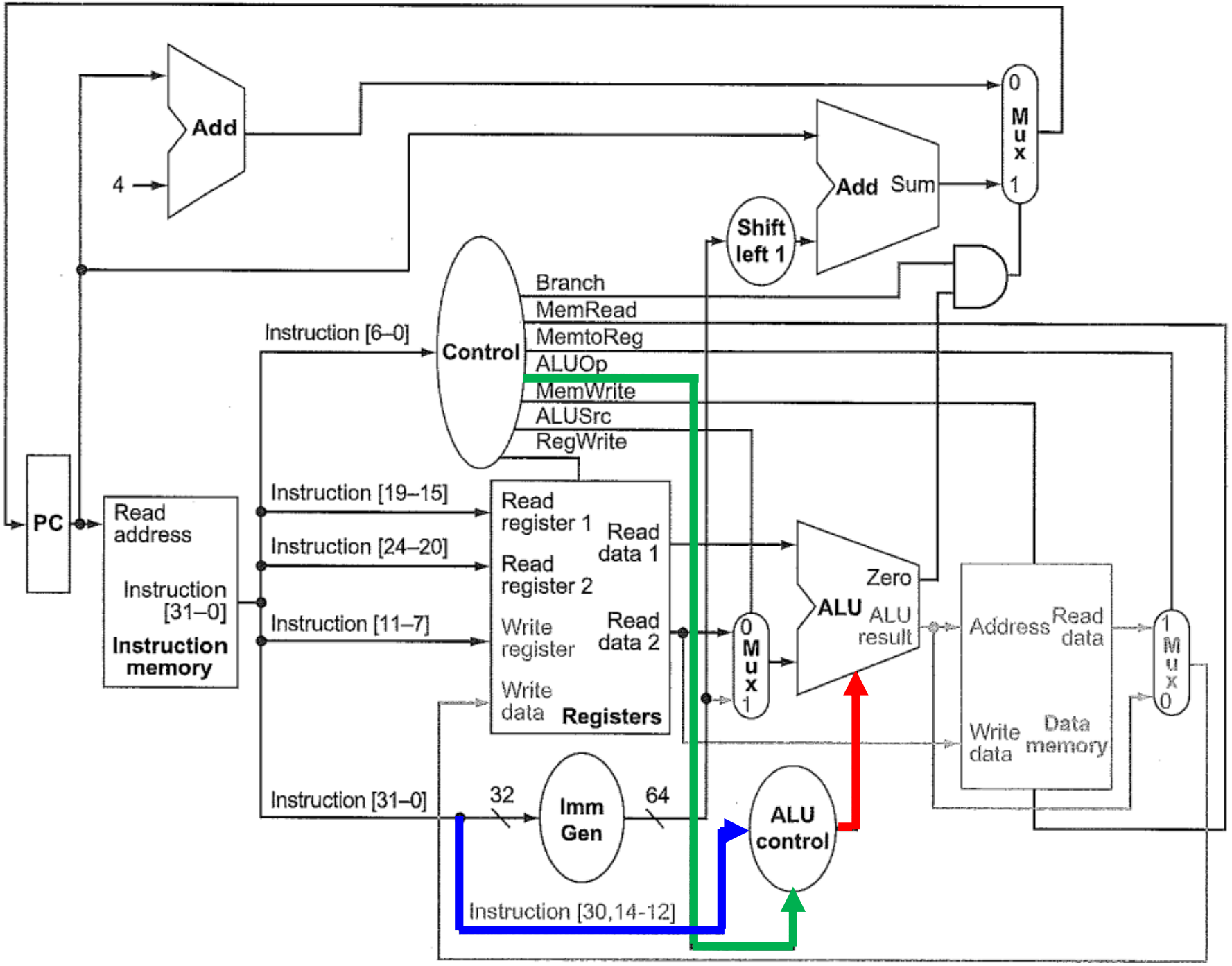
注意：本表对应黑书教材P182的图，教材只考虑七种指令

# ALU控制输入信号真值表（黑书P176）

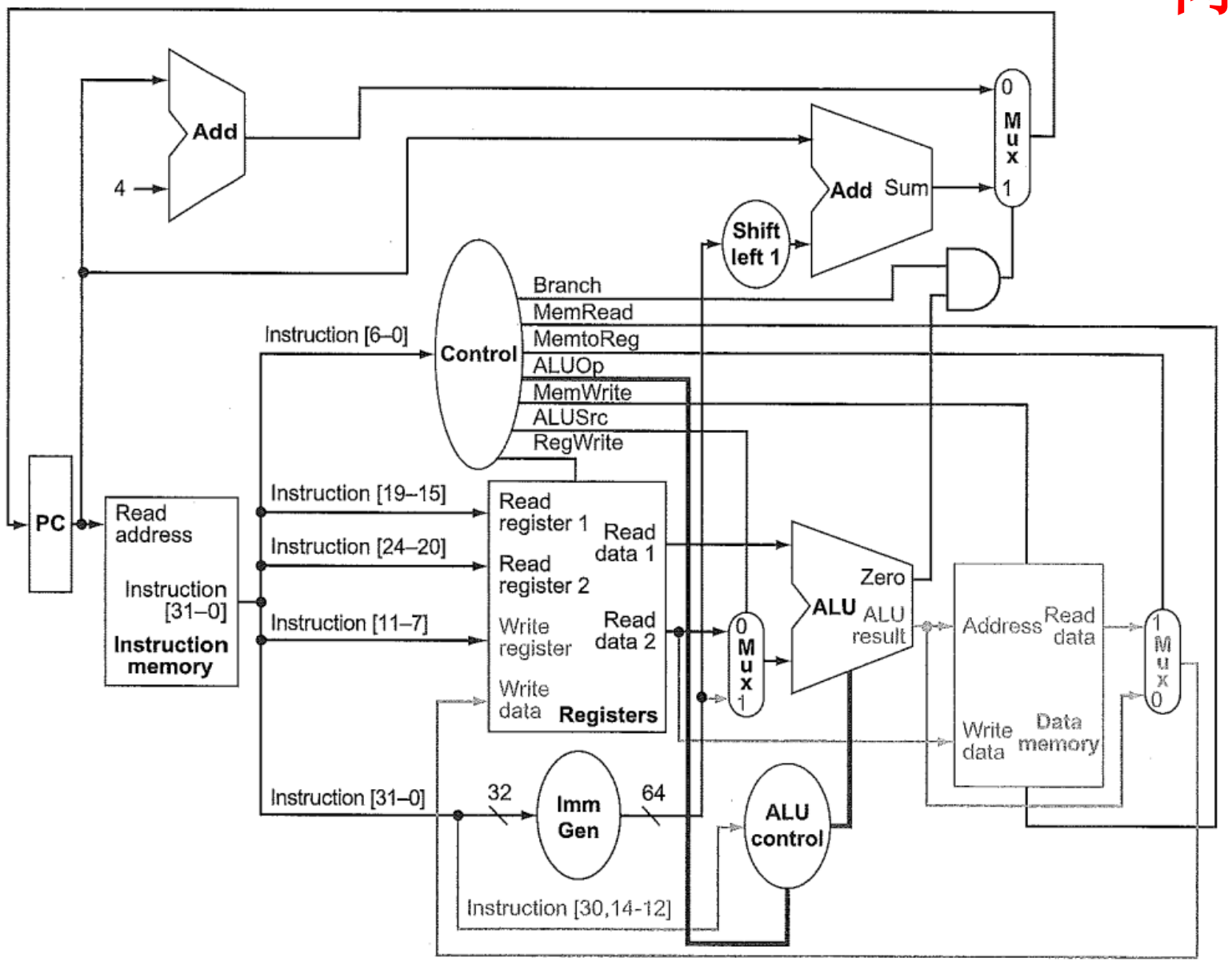
| ALUOp | | funct7字段 | | | | | | | funct3字段 | | | ALU控制输入 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ALUOp1 | ALUOp0 | I[31] | I[30] | I[29] | I[28] | I[27] | I[26] | I[25] | I[14] | I[13] | I[12] | |
| 0 | 0 | X | X | X | X | X | X | X | X | X | X | 0010 |
| 0 | 1 | X | X | X | X | X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

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]

# 黑书P182图


Add
4
0
M
u
x
1
Add Sum
Shift left 1
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [6–0]
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [19–15]
Instruction [24–20]
Instruction [11–7]
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
Zero
ALU
ALU result
Address
Read data
Write data
Data memory
Instruction [31–0]
32
Imm Gen
64
ALU control
Instruction [30,14-12]

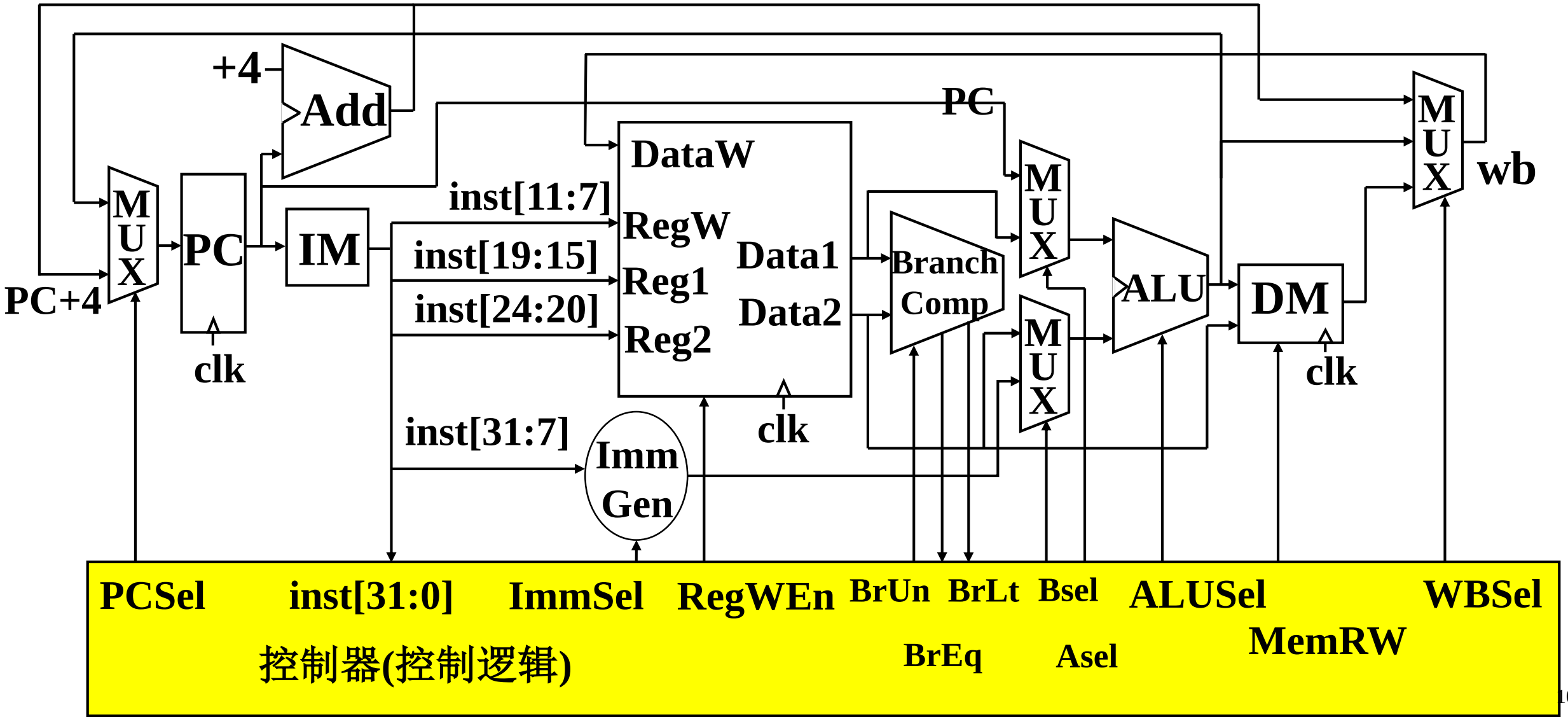
# 黑书P182图




**问题：为什么要有ALUOp？**

答：实现**多级译码**，即主控单元生成ALUOp作用于ALU的输入控制信号，控制ALU执行。多级控制可以减小主控制单元的规模，多个小的控制单元可能潜在地减少控制单元延迟。**这个优化很重要，控制单元的延迟决定了时钟周期。**

# RISC-V 的数据通路的控制器如何实现？

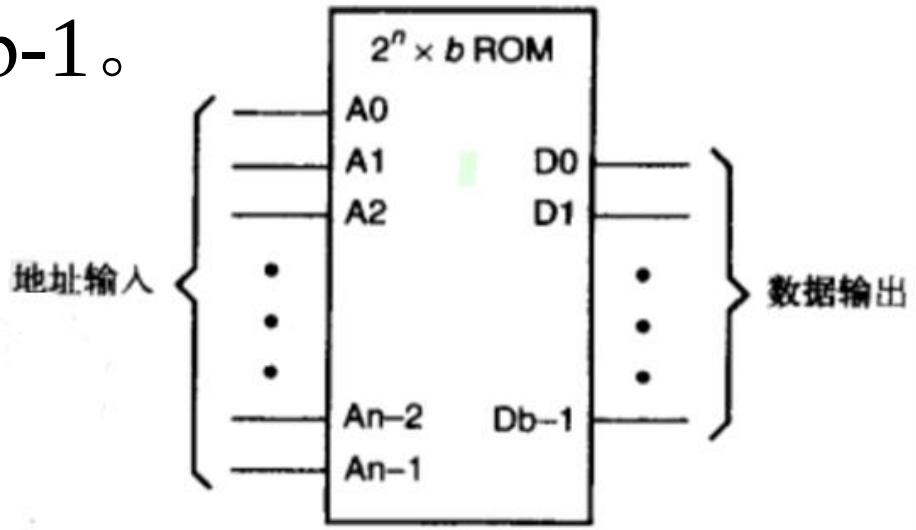


# RISC V控制器两种组合逻辑电路实现方式

- **ROM（组合逻辑）**
  - 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
- **门级网表（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A0, A1, …, An-1。
- 输出被称为数据输出（data output），通常命名为D0, D1, …, Db-1。




一个$2^n$ * 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。

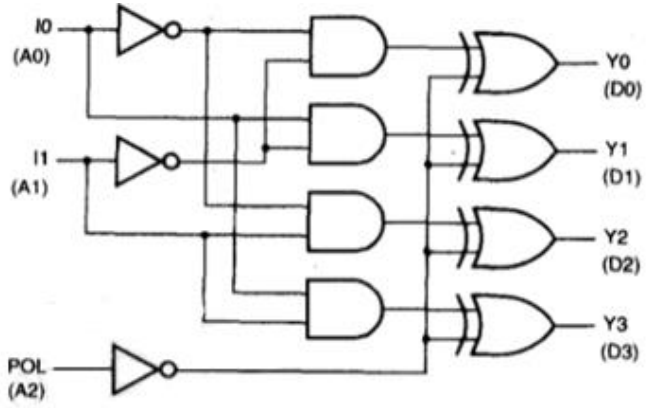
- 一个3输入、4输出的组合功能的真值表，可以被存储在一个$2^3 * 4$ (8 * 4) 的只读存储器中。忽略延迟，ROM的数据输出总是等于真值表中由地址输入所选择的那行输出位。

| 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 |

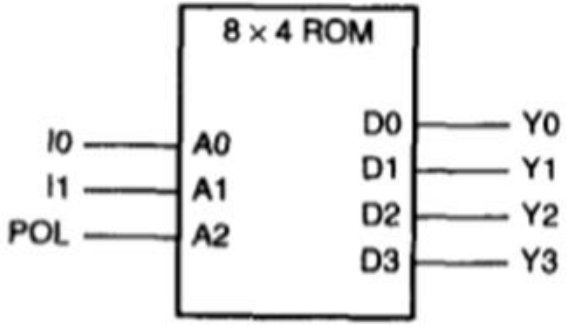
一个3输入4输出组合逻辑函数的真值表
（具有输出极性控制的2-4译码器）

# 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

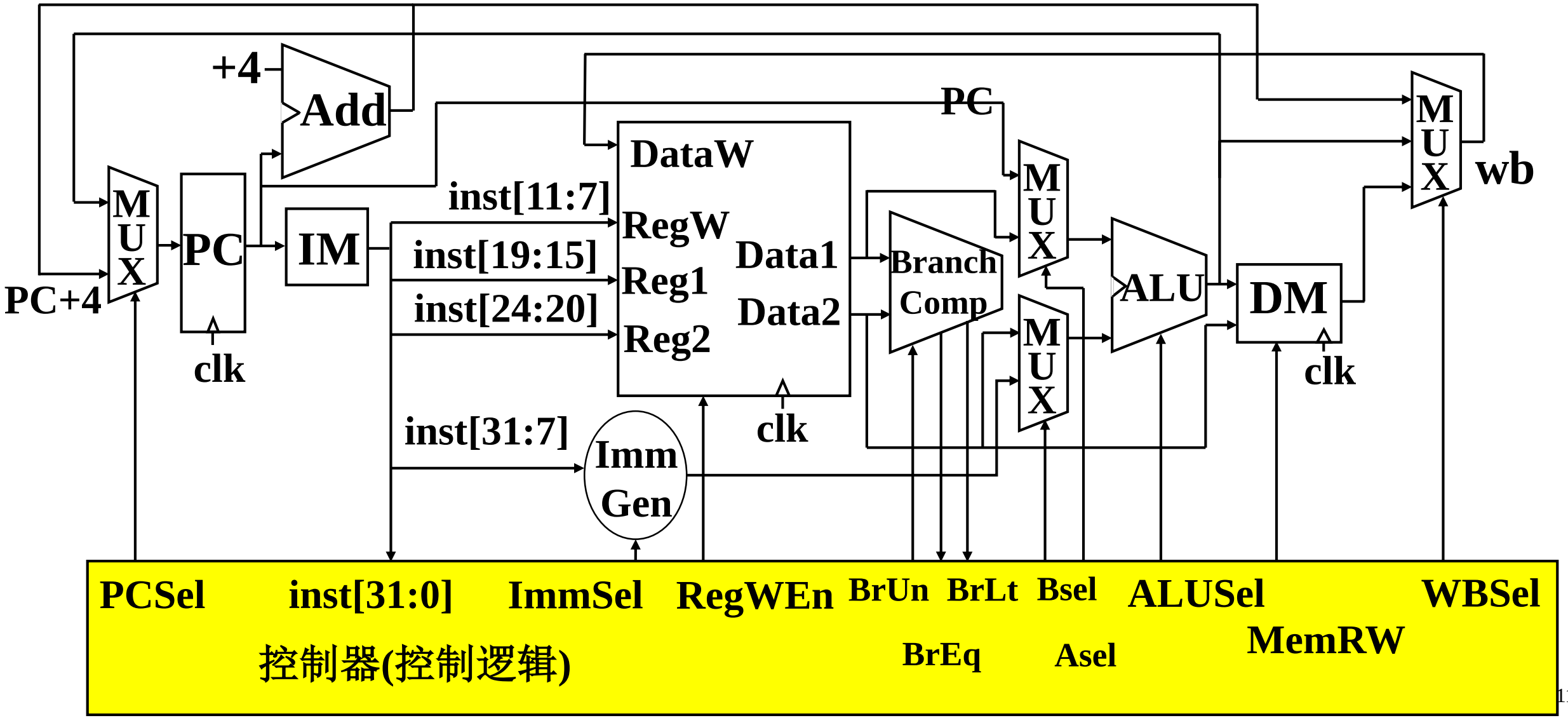



用存储真值表的8 * 4 ROM构建2-4译码器

**实现方式1：门级网表**

**实现方式2：ROM**

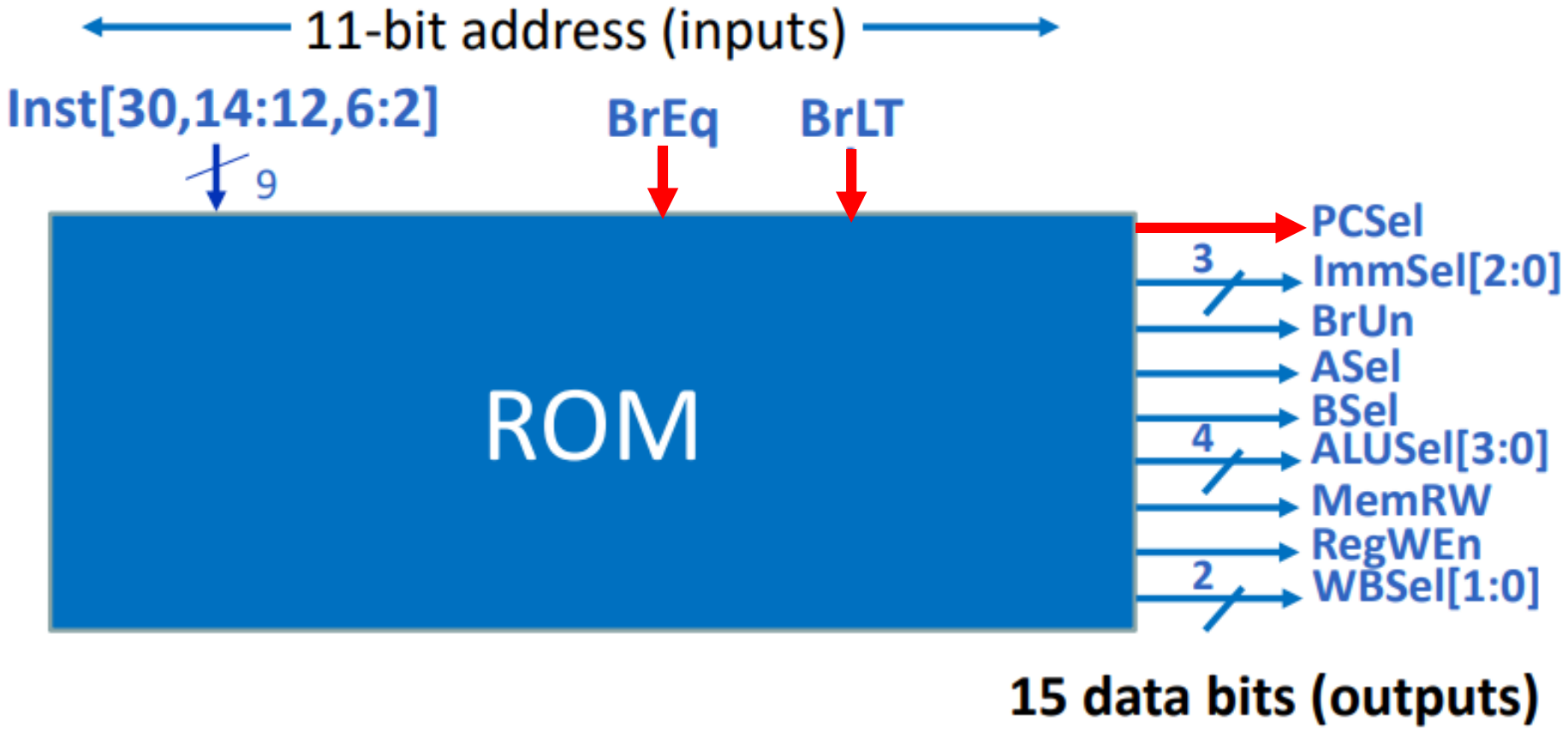
# 将控制器中的控制信息提前存储起来



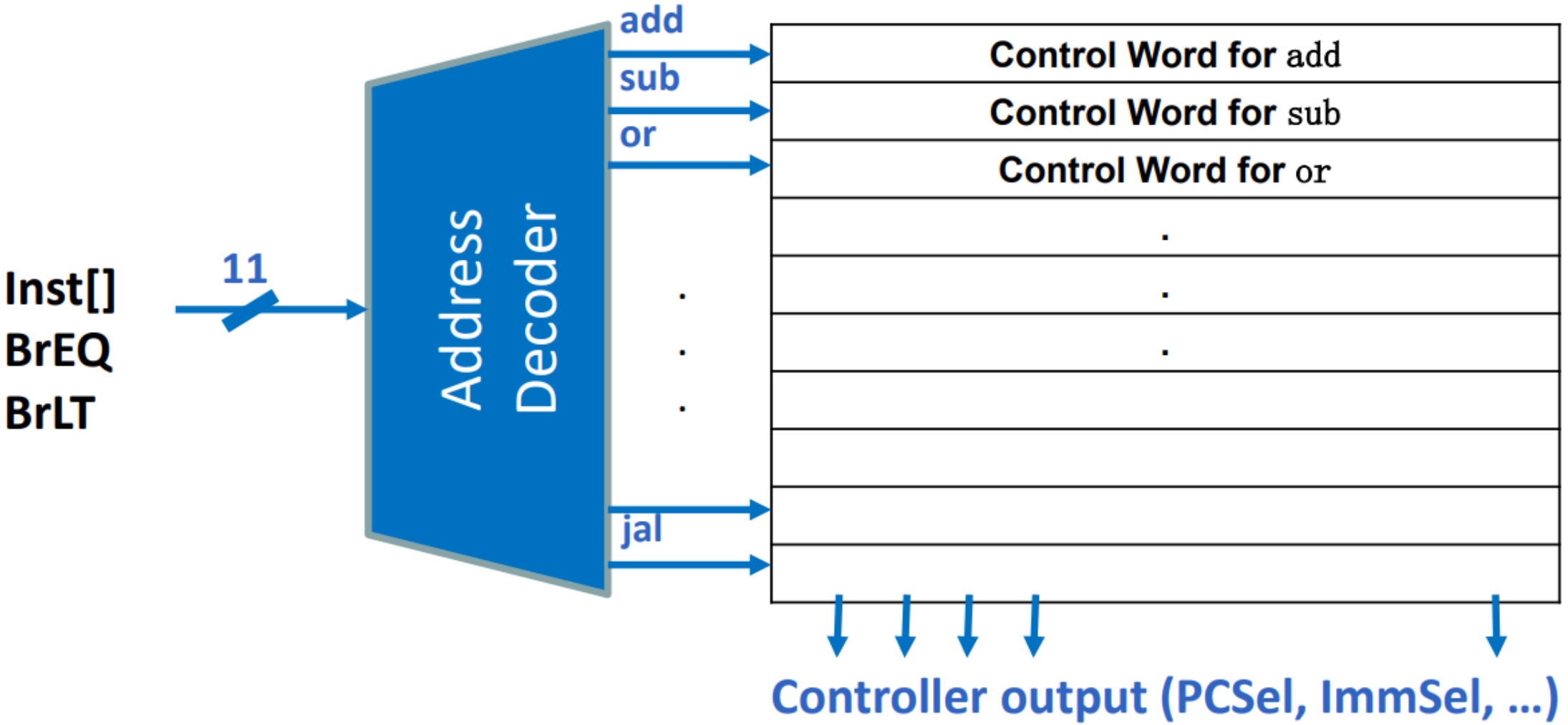
# 利用ROM实现控制器的理论依据

**用来区分某条指令种类的编码只有inst[30]、inst[14:12]、inst[6:2]**

**并且ROM是组合逻辑电路**



# ROM 控制器实现


Inst[]
BrEQ
BrLT
11
Address Decoder
add
sub
or
jal
Control Word for add
Control Word for sub
Control Word for or
Controller output (PCSel, ImmSel, ...)

# 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 | Sub | Read | 1 | ALU |
| addi | * | * | +4 | I | * | Reg | Imm | Add | Read | 1 | ALU |
| ld | * | * | +4 | I | * | Reg | Imm | Add | Read | 1 | Mem |
| sd | * | * | +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 | * |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | | | |
| 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 |

# 小结

- 实现了一个处理器数据通路
  - 能够在一个时钟周期内执行给定RISC-V指令子集
  - 并非所有指令都会用到全部的硬件单元
  - 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（ROM）
  - 基于ROM或组合逻辑实现