



## 指令格式与数据通路设计

2021年秋

# RISC-V 的指令格式

- 每一条指令是32位，4个字节（1个字）



- 指令格式将32位分为不同的“域”，就是不同的部分
- 定义了六种指令的格式
- R-格式 I-格式 S-格式 B格式 U格式 J-格式

# RISC-V 的指令格式 (I)

- ❑ R-格式：寄存器指令，指定指令中的3个寄存器
- ❑ I-格式：指令中包含立即数，用于带一个常数的算术指令以及加载指令
- ❑ S-格式：store指令

|            |    |           |    |     |         |     |            |        |        |        |          |          |         |        |        |        |
|------------|----|-----------|----|-----|---------|-----|------------|--------|--------|--------|----------|----------|---------|--------|--------|--------|
| 31         | 30 | 25        | 24 | 21  | 20      | 19  | 15         | 14     | 12     | 11     | 8        | 7        | 6       | 0      |        |        |
| funct7     |    |           |    | rs2 |         |     | rs1        |        | funct3 |        | rd       |          |         | opcode |        | R-type |
| imm[11:0]  |    |           |    |     |         | rs1 |            | funct3 |        | rd     |          |          | opcode  |        | I-type |        |
| imm[11:5]  |    |           |    | rs2 |         |     | rs1        |        | funct3 |        | imm[4:0] |          |         | opcode |        | S-type |
| imm[12]    |    | imm[10:5] |    |     | rs2     |     |            | rs1    |        | funct3 |          | imm[4:1] | imm[11] | opcode |        | B-type |
| imm[31:12] |    |           |    |     |         |     |            |        |        | rd     |          |          | opcode  |        | U-type |        |
| imm[20]    |    | imm[10:1] |    |     | imm[11] |     | imm[19:12] |        |        | rd     |          |          | opcode  |        | J-type |        |

# RISC-V 的指令格式 (II)

□ B-格式：分支指令

□ U-格式：大立即数

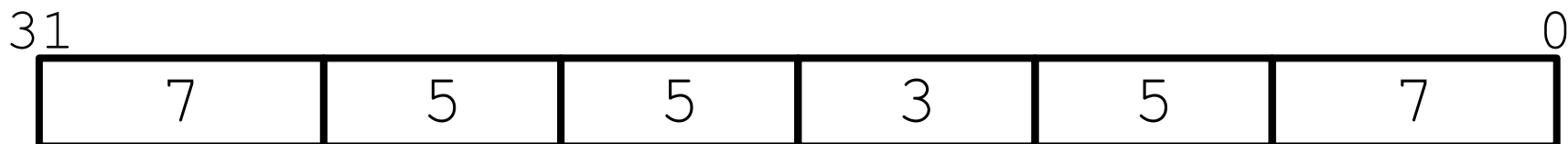
- 两个指令lui (load upper imm), auipc (add upper imm to PC)

□ J-格式：唯一指令jal

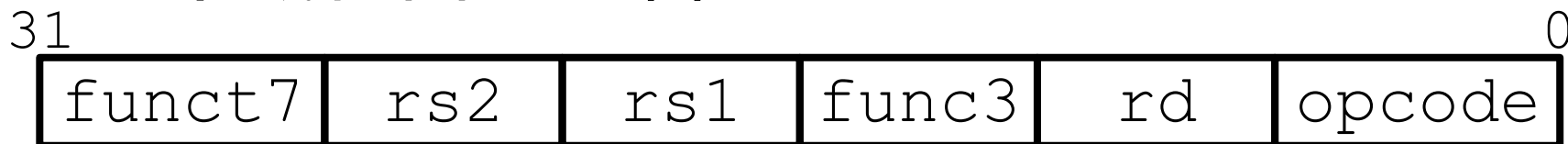
|            |    |           |    |     |         |     |            |        |        |        |          |          |         |        |        |        |
|------------|----|-----------|----|-----|---------|-----|------------|--------|--------|--------|----------|----------|---------|--------|--------|--------|
| 31         | 30 | 25        | 24 | 21  | 20      | 19  | 15         | 14     | 12     | 11     | 8        | 7        | 6       | 0      |        |        |
| funct7     |    |           |    | rs2 |         |     | rs1        |        | funct3 |        | rd       |          |         | opcode |        | R-type |
| imm[11:0]  |    |           |    |     |         | rs1 |            | funct3 |        | rd     |          |          | opcode  |        | I-type |        |
| imm[11:5]  |    |           |    | rs2 |         |     | rs1        |        | funct3 |        | imm[4:0] |          |         | opcode |        | S-type |
| imm[12]    |    | imm[10:5] |    |     | rs2     |     |            | rs1    |        | funct3 |          | imm[4:1] | imm[11] | opcode |        | B-type |
| imm[31:12] |    |           |    |     |         |     |            |        | rd     |        |          | opcode   |         |        | U-type |        |
| imm[20]    |    | imm[10:1] |    |     | imm[11] |     | imm[19:12] |        |        | rd     |          |          | opcode  |        | J-type |        |

# R类型指令

- R型指令每个指令字分成6个域：

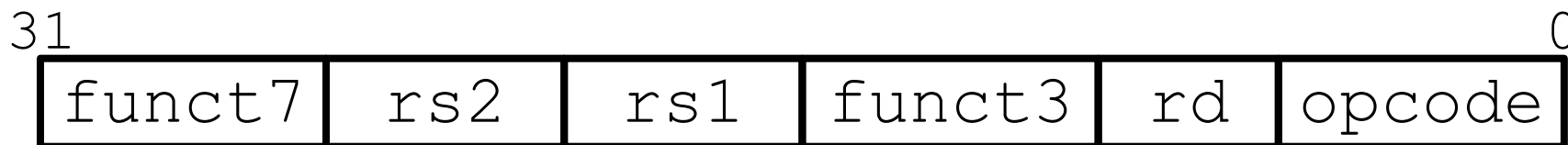


- 每个域有自己的名字:



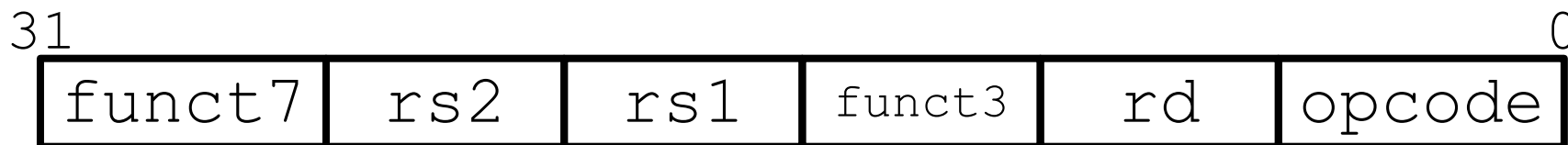
- 每个域有不同含义

# R类型指令



- **opcode** (7): 操作码，但未指明确切操作
  - R类型指令操作码都为 0110011
- **funct3** (3), **funct7** (7): 与操作码 **opcode** 一起指定了指令的具体功能
- 计算一下能够编码多少R类型指令功能?
  - **opcode** 是固定的, 依据 **funct**:  $2^{10} = 1024$

# R类型指令



- **rs1** (5): 第一个操作数, (“source register”)
- **rs2** (5): 第二个操作数, (“second source register”)
- **rd** (5): 目标寄存器, (“destination register”)
- 每一个寄存器5位进行编码, 可以指定32个寄存器, 编码的是寄存器编号

# R类型指令举例

add x18 x19 x10

funct7

funct3

opcode

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

31

0

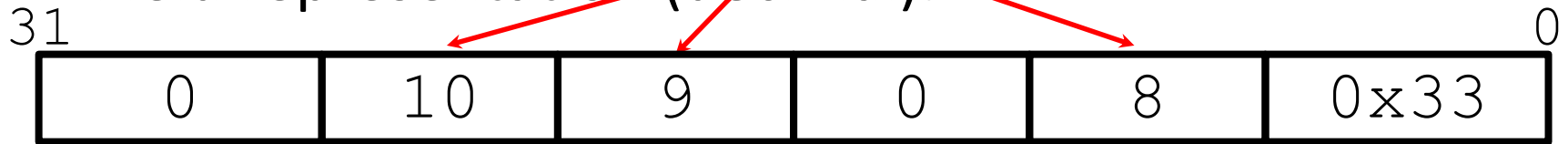
|        |     |     |        |    |        |
|--------|-----|-----|--------|----|--------|
| 0      | 10  | 19  | 0      | 18 | R-OP   |
| funct7 | rs2 | rs1 | funct3 | rd | opcode |



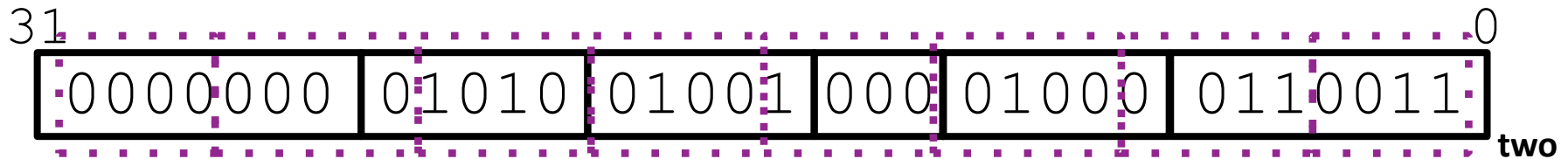
# R类型机器码

- Instruction: `add x8, x9, x10`

Field representation (decimal):



Field representation (binary):



hex representation: `0x 00A4 8433`

decimal representation: `10,781,747`

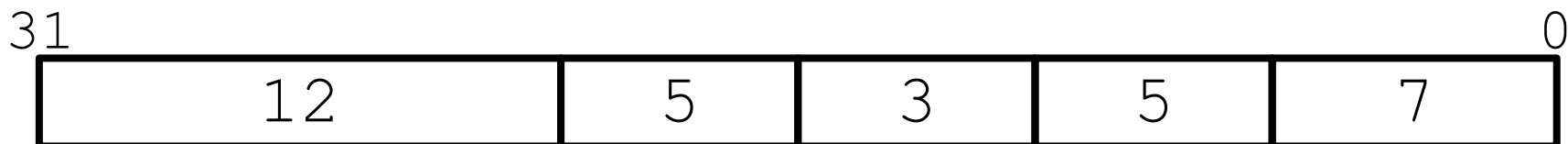
机器码

# I类型指令

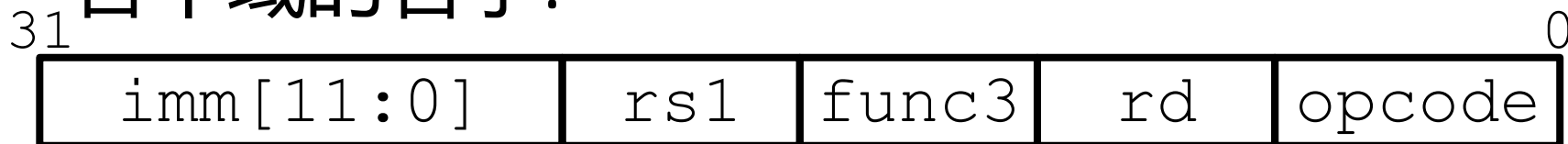
- ❑ 对于含有立即数的指令来说，5位的域能够表达的范围太小
- ❑ 理想来说，RISC-V 指令最好只有一种格式。但是实际情况下需要折衷
- ❑ 在实际中可以依据类似于R指令格式的方式定义新的指令格式
  - 如果一条指令使用了立即数，这条指令最多使用两个寄存器

# I类型指令

- 下面是I类型指令的格式:

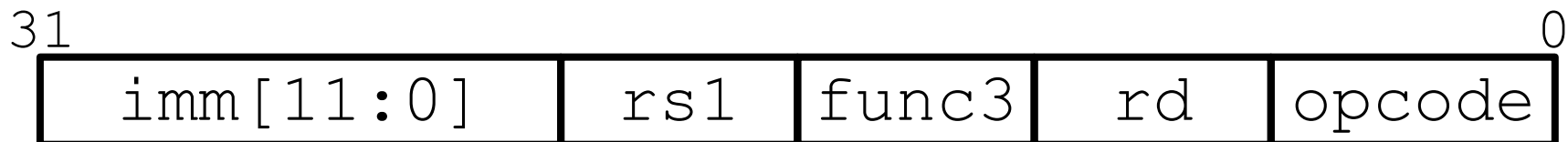


- 各个域的名字:



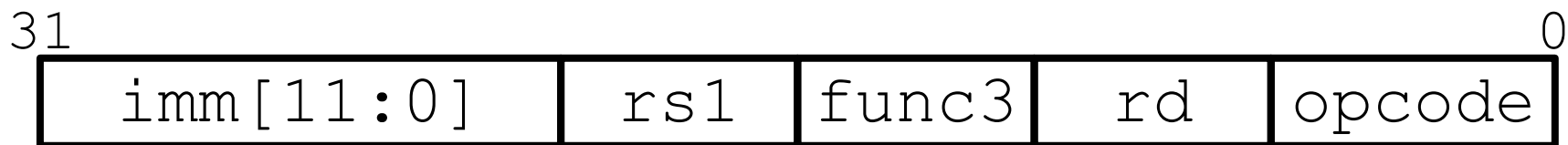
- 四个域与R类型指令是一样的
  - opcode 仍然放置在原来的位置

# I类型指令



- `opcode` (7): 指定操作类型
- `rs1` (5): 指定一个寄存器操作数
- `rd` (5): 指定目标寄存器 (“destination register”)

# I类型指令



- **immediate (12):** 12位立即数
  - 所有的计算是用字进行计算，即32位，必须要对立即数进行扩展
  - 采用符号扩展方式
- 12位表示范围 [-2048, +2047]
  - 如果立即数超过12位，怎么办？

# I类型举例

addi x15, x1, -50

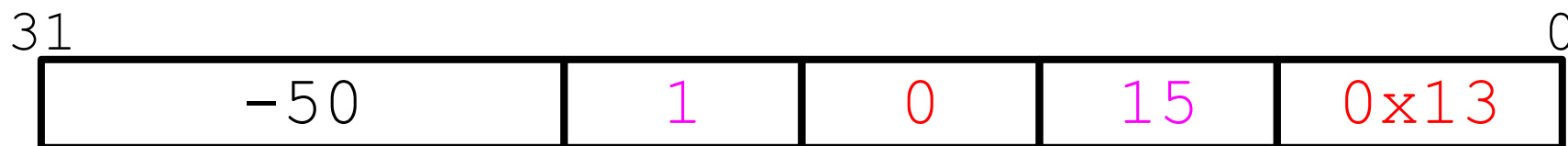
|           |       |     |     |    |         |       |
|-----------|-------|-----|-----|----|---------|-------|
| imm[11:0] |       | rs1 | 000 | rd | 0010011 | ADDI  |
| imm[11:0] |       | rs1 | 010 | rd | 0010011 | SLTI  |
| imm[11:0] |       | rs1 | 011 | rd | 0010011 | SLTIU |
| imm[11:0] |       | rs1 | 100 | rd | 0010011 | XORI  |
| imm[11:0] |       | rs1 | 110 | rd | 0010011 | ORI   |
| imm[11:0] |       | rs1 | 111 | rd | 0010011 | ANDI  |
| 0000000   | shamt | rs1 | 001 | rd | 0010011 | SLLI  |
| 0000000   | shamt | rs1 | 101 | rd | 0010011 | SRLI  |
| 0100000   | shamt | rs1 | 101 | rd | 0010011 | SRAI  |

|           |  |     |       |    |        |
|-----------|--|-----|-------|----|--------|
| 31        |  |     |       |    | 0      |
| -50       |  | 1   | 0     | 15 | 0x13   |
| imm[11:0] |  | rs1 | func3 | rd | opcode |

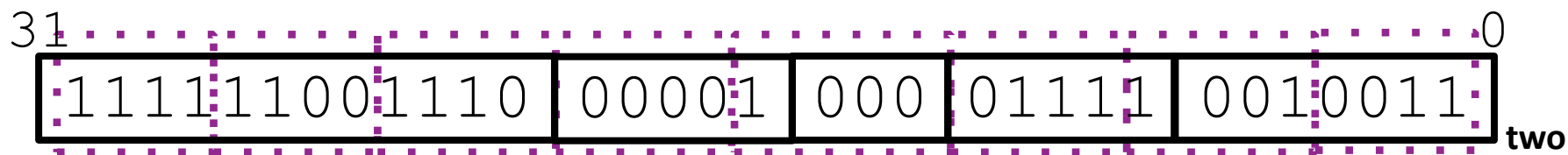
# I类型举例

- 指令: `addi x15, x1, -50`

各个域的十进制表示:



各个域的二进制表示:

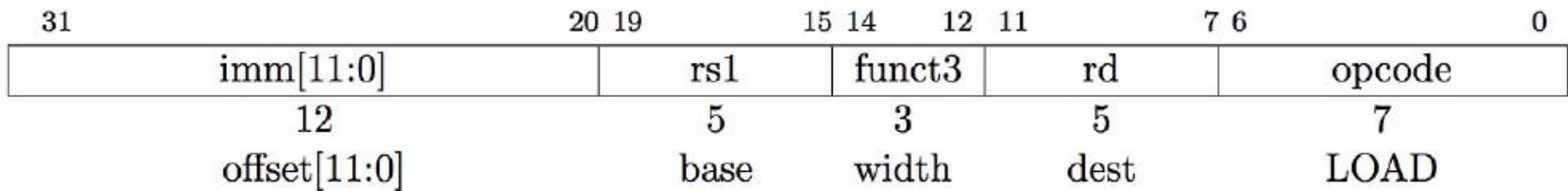


十六进制表示: `0x FCE08793`

十进制表示: `4,242,573,203`

# I 类型举例 - load

□ 指令: `lw x14, 8(x2)`



□ `rd` (5): 结果放置的寄存器

□ `rs1` (5): 基地址寄存器

□ `immediate` (12)

■ 基地址寄存器的值 + 立即数 → load内存地址



# I 类型举例 - load

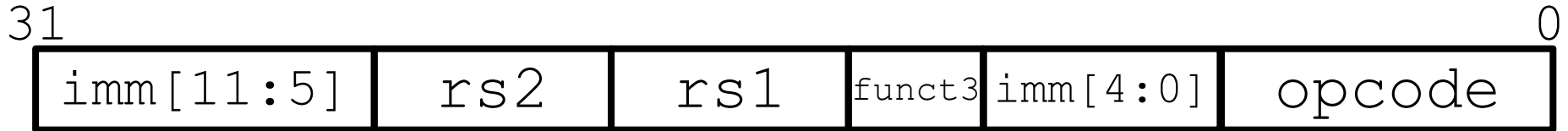
□ 指令: `lw x14, 8(x2)`

|              |    |    |    |      |    |        |      |   |        |  |
|--------------|----|----|----|------|----|--------|------|---|--------|--|
| 31           | 20 | 19 | 15 | 14   | 12 | 11     | 7    | 6 | 0      |  |
| imm[11:0]    |    |    |    | rs1  |    | funct3 | rd   |   | opcode |  |
| 12           |    |    |    | 5    |    | 3      | 5    |   | 7      |  |
| offset[11:0] |    |    |    | base |    | width  | dest |   | LOAD   |  |

|           |     |     |    |         |     |
|-----------|-----|-----|----|---------|-----|
| 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 | 100 | rd | 0000011 | LBU |
| imm[11:0] | rs1 | 101 | rd | 0000011 | LHU |

|              |       |     |       |         |
|--------------|-------|-----|-------|---------|
| 000000001000 | 00010 | 010 | 01110 | 0000011 |
| imm=+8       | rs1=2 | LW  | rd=14 | LOAD    |

# S 类型指令



## □ Store指令

- `rs1` (5): 基地址寄存器
- `rs2` (5): 数据寄存器
- `immediate` (11:5) + `immediate` (4:0) : 地址偏移

## □ 为什么不把rs2用作imm的低位?

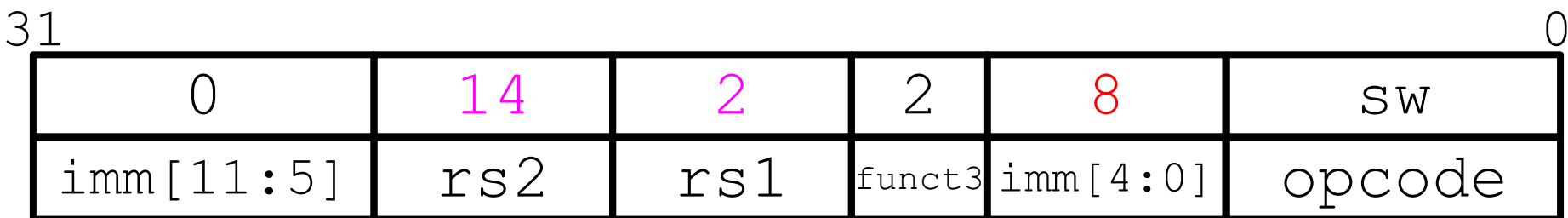
- RISC-V设计让`rs1`和`rs2`的位置保持固定

# RISC-V 设计策略：让寄存器位置保持不变

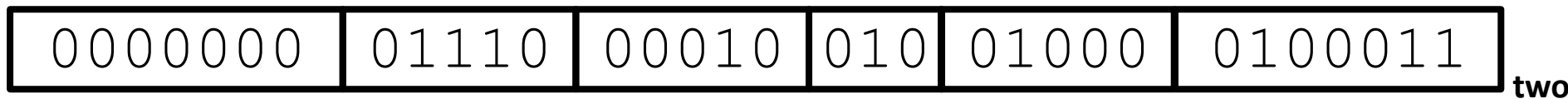
- 对于所有的操作来说，关键路径包括获得寄存器的值
- 把读的寄存器都放到相同的位置，那么每次都可以立即去读寄存器文件，不需要做判断
  - 如果读出来的数据其实是不必要的（例如I类型），后续丢弃即可
- 其它的RISC体系结构的设计有些许的不同
  - 在译码阶段需要通过指令来判断需要读哪些寄存器
- 寄存器位置保持不变是RISC-V的体系结构上一系列微小的调整之一，可以让整个处理器工作流程更好

# S 类型举例

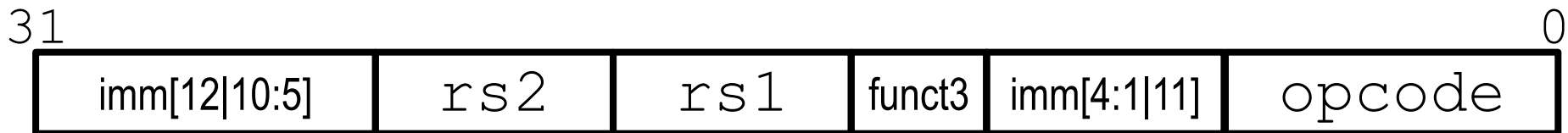
□ 指令: `sw x14, 8(x2)`



|           |     |     |     |          |         |    |
|-----------|-----|-----|-----|----------|---------|----|
| imm[11:5] | rs2 | rs1 | 000 | imm[4:0] | 0100011 | SB |
| imm[11:5] | rs2 | rs1 | 001 | imm[4:0] | 0100011 | SH |
| imm[11:5] | rs2 | rs1 | 010 | imm[4:0] | 0100011 | SW |



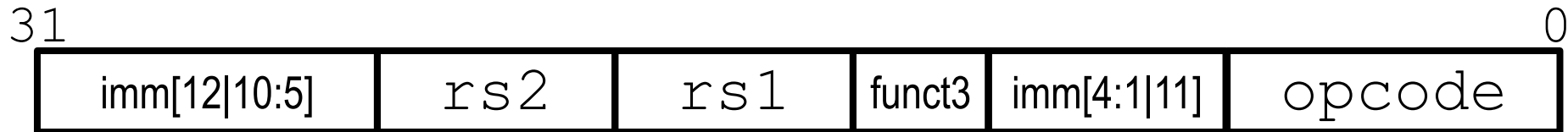
# B 类型指令



## □ 分支指令

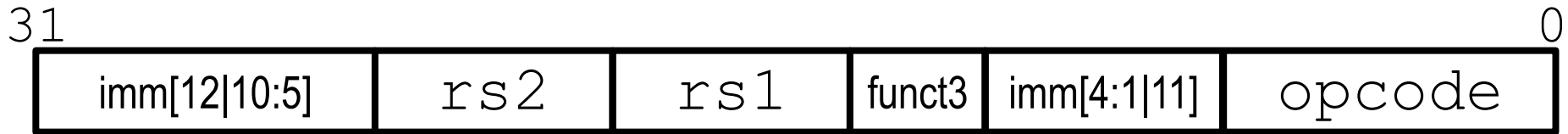
# 分支指令

- beq and bne
  - 需要指定目标地址
  - 也需要两个寄存器间比较



- opcode 指定是 beq (4) 还是 bne (5)
  - rs1 , rs2 用来指定两个寄存器
  - 使用 immediate 指定地址? (PC相对跳转, 循环的跳转一般在一个比较小的范围)

# PC相对跳转寻址



- **PC-Relative Addressing:** 使用立即数作为补码，用作在PC上的偏移
- 类似于 lw/sw 的基址偏移寻址，源寄存器 rs1 为PC
- 这样可以指定从PC开始的  $\pm 2^{11}$  的地址范围

# 实际地址范围的计算

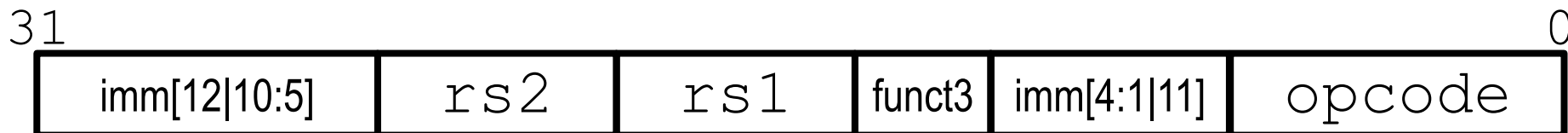
- ❑ 实际上 RISC-V 使用的是固定长度的指令字长，内存是按字节地址寻址的
- ❑ 指令是按照字对齐的，指令的地址总是2的倍数（末1位为0）（与Compact的指令格式兼容，需要最后一位为0即可，但是在RV32I中实际是4字节对齐的）
- ❑ PC总是指向一个指令地址，就可以做类型指针的操作
- ❑ 在指令地址范围计算的时候，可以指定  $\pm 2^{12}$  范围的地址



# 分支的计算

- 不需要分支
  - $PC = PC + 4$
- 需要进行分支
  - $PC = PC + \text{immediate}$
- **Observations:**
  - `immediate` 是跳转字节数 (注意`imm[12:1]`需要加上最末尾隐含的0位)
  - 为向前 (+), 或者向后 (-) 的偏移地址 (范围为 $\pm 4\text{KB}$ )

# B 类型指令



□ 与 S 类型指令非常相似

- 两个寄存器 (rs1, rs2), 一个立即数 (12位)

□ Imm 表示了 [-4096, +4094] 范围的 2字节对齐 的偏移

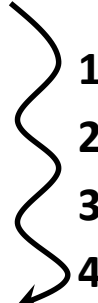
□ Immediate 用 12比特 表示了 13比特的范围

- 因为2字节对齐, 所以最低位总是为0, 因而省略了
- B类型中imm[12:1] 与 S类型中imm[11:0] 不同

# 分支举例

- RISC-V Code:

```
Loop: beq x9, x0, End
      add x8, x8, x10
      addi x9, x9, -1
      j Loop
End: <some instr>
```



1 Count  
2 instructions  
3 from branch  
4

- B-Format fields:

opcode = 0x67

rs1 = 9 (first operand)

rs2 = 0 (second operand)

immediate = ???  $4*4=16$

# 分支举例

- RISC-V Code:

Loop: **beq** **x9, x0, End**

add x8, x8, x10

addi x9, x9, -1

j Loop

End: <some instr>

imm = 16

0 0000 0001 0000

|    |              |     |     |        |             |        |
|----|--------------|-----|-----|--------|-------------|--------|
| 31 |              | 0   | 9   | beq    |             | 0      |
|    | imm[12 10:5] | rs2 | rs1 | funct3 | imm[4:1 11] | opcode |

|          |       |       |     |       |         |     |
|----------|-------|-------|-----|-------|---------|-----|
| 00000000 | 01110 | 00010 | 010 | 10000 | 1100111 | two |
|----------|-------|-------|-----|-------|---------|-----|

# 所有B型指令

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

# PC相对寻址的特性

□ PC相对寻址的情况下，如果整块代码进行移动，则代码中的相对寻址值无需更改，如果只移动其中的几行代码，则相对寻址值可能需要更改

□ 如果需要移动的范围超过表达的指令范围  $> 2^{10}$

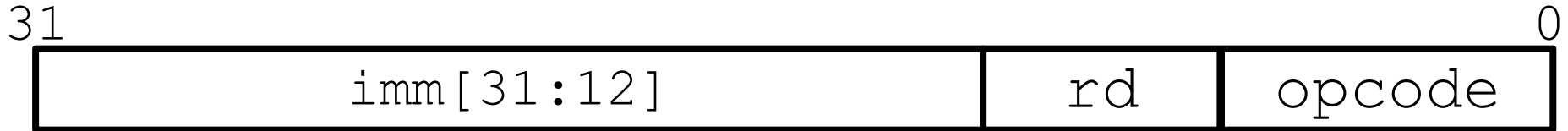
- 可以使用其它的指令

- ```
beq x10,x0,far
# next instr
```

 --> 

```
bne x10,x0,next
j far
next: # next instr
```

# U 类型指令



- ❑ 如何处理一个字长的立即数？指令字长总共32位
- ❑ lui: Load Upper Immediate
  - lui reg, imm
  - 将20位的立即数装入到寄存器reg的高20位
  - 将低12位清零
- ❑ auipc: Add Upper Immediate to PC

# 使用lui的例子

□ lui设置了高20位，通过addi设置低12位

□ 例1：设置0x87654321

```
lui    x10, 0x87654          # x10=0x87654000
```

```
addi   x10, x10, 0x321       # x10=0x87654321
```

□ 例2：设置0xDEADBEEF

```
lui    x10, 0xDEADB         # x10=0xDEADB000
```

```
addi   x10, x10, 0xEEF      # x10=0xDEADAEEF
```

解决办法：lui 高20位 加1

伪指令li x10, 0xDEADBEEF：自动生成两条指令



# J 类型指令



## □ Jal 指令

- Rd: 返回地址 (PC+4)
- Imm: 偏移offset, 设置跳转  $PC = PC + offset$

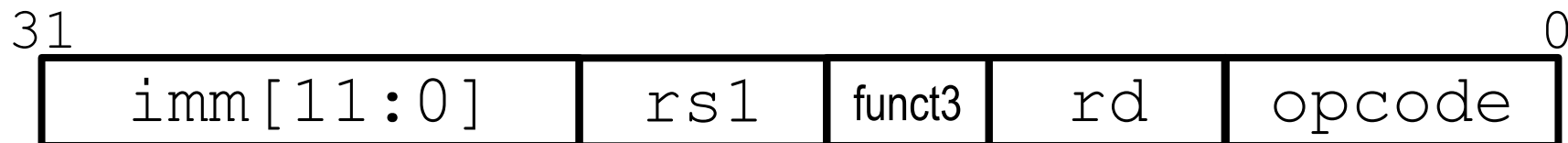
## □ 跳转范围: $[-2^{19}, +2^{19}]$ 个位置, 2字节为单位

- 实际跳转范围  $[-2^{18}, +2^{18}]$ 个32-bit指令

## □ J 伪指令

- # j pseudo-instruction  
j Label = jal x0, Label # Discard return address
- # Call function within  $2^{18}$  instructions of PC  
jal ra, FuncName

# Jalr 指令（属于 I 类型，不是 J 类型）



## □ Jalr rd, rs, immediate

- Rd: 返回地址 (PC+4)
- Imm: 偏移offset, 设置跳转  $PC = rs1 + offset$
- Immediate与 I 类型指令中的算术和加载指令 一样
  - 注意, 不需要 \* 2

# Jalr 用法例子

---

# ret and jr psuedo-instructions

ret = jr ra = jalr ra, x0, 0

# Call function at any 32-bit absolute address

lui x1, <hi20bits>

jalr ra, x1, <lo12bits>

# Jump PC-relative with 32-bit offset

auipc x1, <hi20bits> # Adds upper immediate value to

# and places result in x1

jalr x0, x1, <lo12bits> # Same sign extension trick needed

# as LUI

# RISC-V 指令格式总结

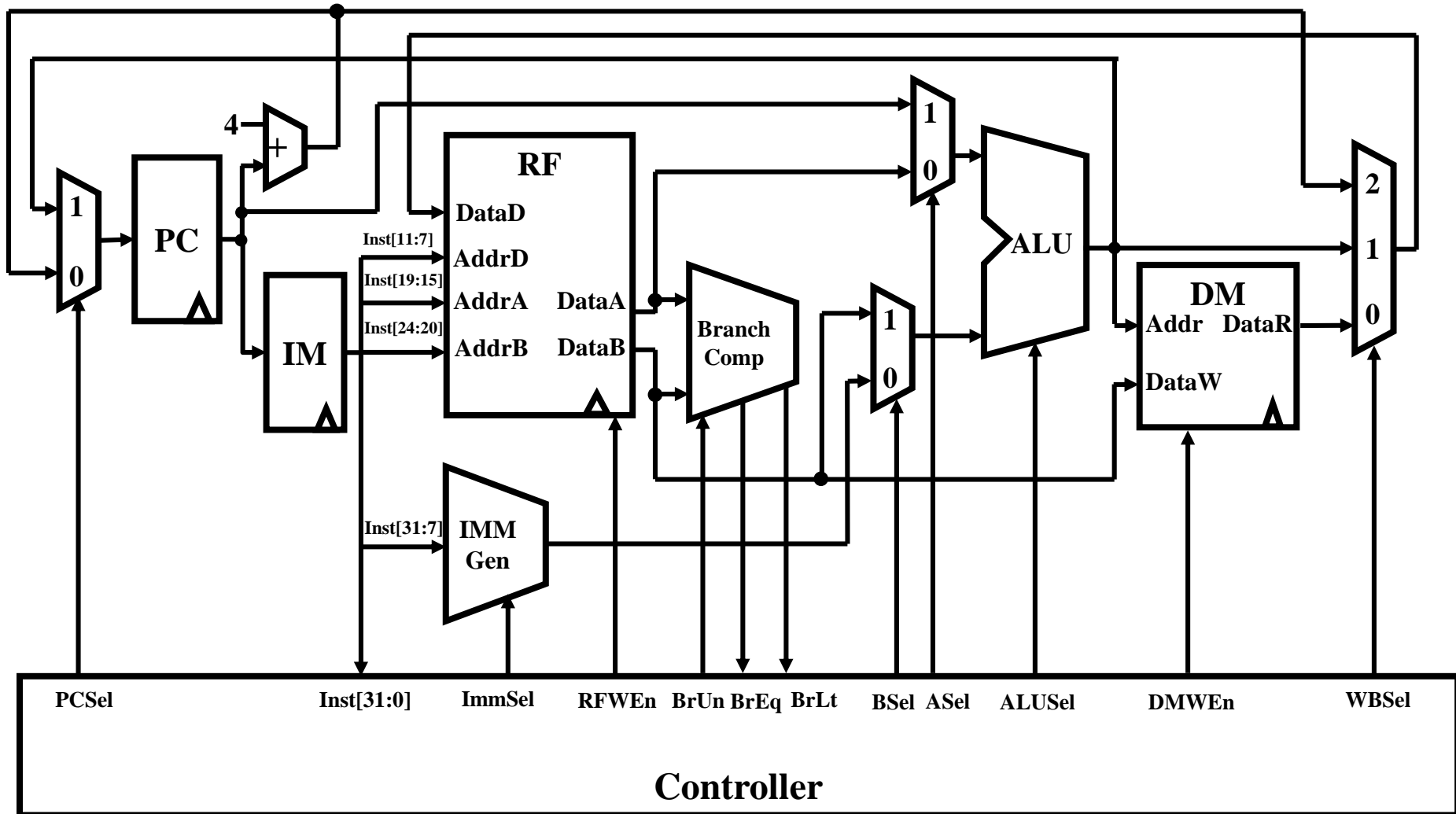
- 代码与数据一样保存在程序地址空间非常重要，硬件和软件在一定程度上同等处理
- RISC-V 机器语言指令:

|            |    |           |    |     |     |         |     |            |        |        |          |          |        |         |        |        |        |
|------------|----|-----------|----|-----|-----|---------|-----|------------|--------|--------|----------|----------|--------|---------|--------|--------|--------|
| 31         | 30 | 25        | 24 | 21  | 20  | 19      | 15  | 14         | 12     | 11     | 8        | 7        | 6      | 0       |        |        |        |
| funct7     |    |           |    | rs2 |     |         | rs1 |            | funct3 |        | rd       |          |        | opcode  |        | R-type |        |
| imm[11:0]  |    |           |    |     |     | rs1     |     | funct3     |        | rd     |          |          | opcode |         | I-type |        |        |
| imm[11:5]  |    |           |    | rs2 |     |         | rs1 |            | funct3 |        | imm[4:0] |          |        | opcode  |        | S-type |        |
| imm[12]    |    | imm[10:5] |    |     | rs2 |         |     | rs1        |        | funct3 |          | imm[4:1] |        | imm[11] |        | opcode | B-type |
| imm[31:12] |    |           |    |     |     |         |     |            |        | rd     |          |          | opcode |         | U-type |        |        |
| imm[20]    |    | imm[10:1] |    |     |     | imm[11] |     | imm[19:12] |        |        |          | rd       |        |         | opcode |        | J-type |

---

# 控制器数据通路设计

# RISC-V RV32I 单周期数据通路



# R型指令—— add, sub, etc.

add \ sub

add rd rs1 rs2

sub rd rs1 rs2

指令功能

$R[rd] = R[rs1] + R[rs2]$

$R[rd] = R[rs1] - R[rs2]$

执行过程

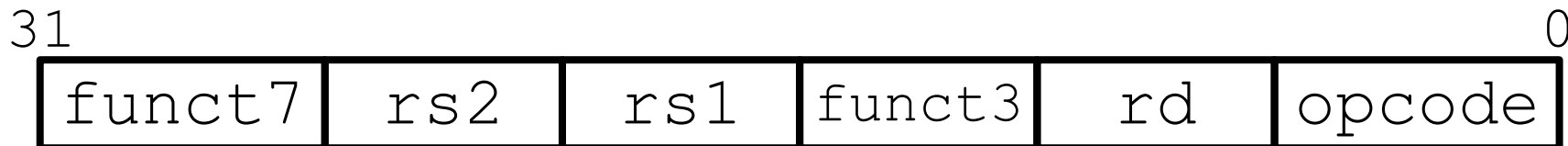
取指令

分析指令

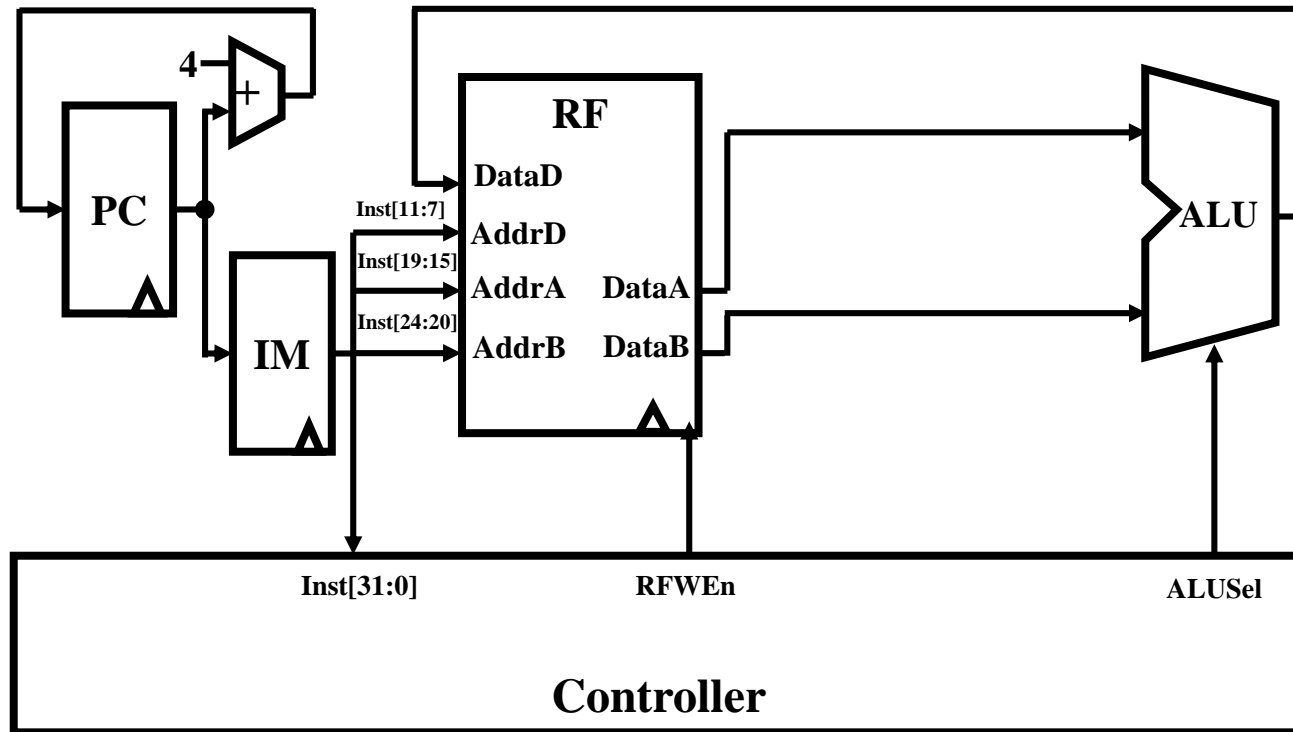
执行指令

- 取操作数
- 运算
- 结果写回

计算下一条指令的地址



# 数据通路设计1



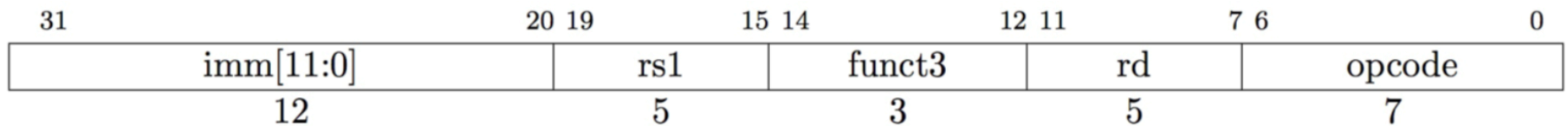


# 算术I型指令—— addi, etc.

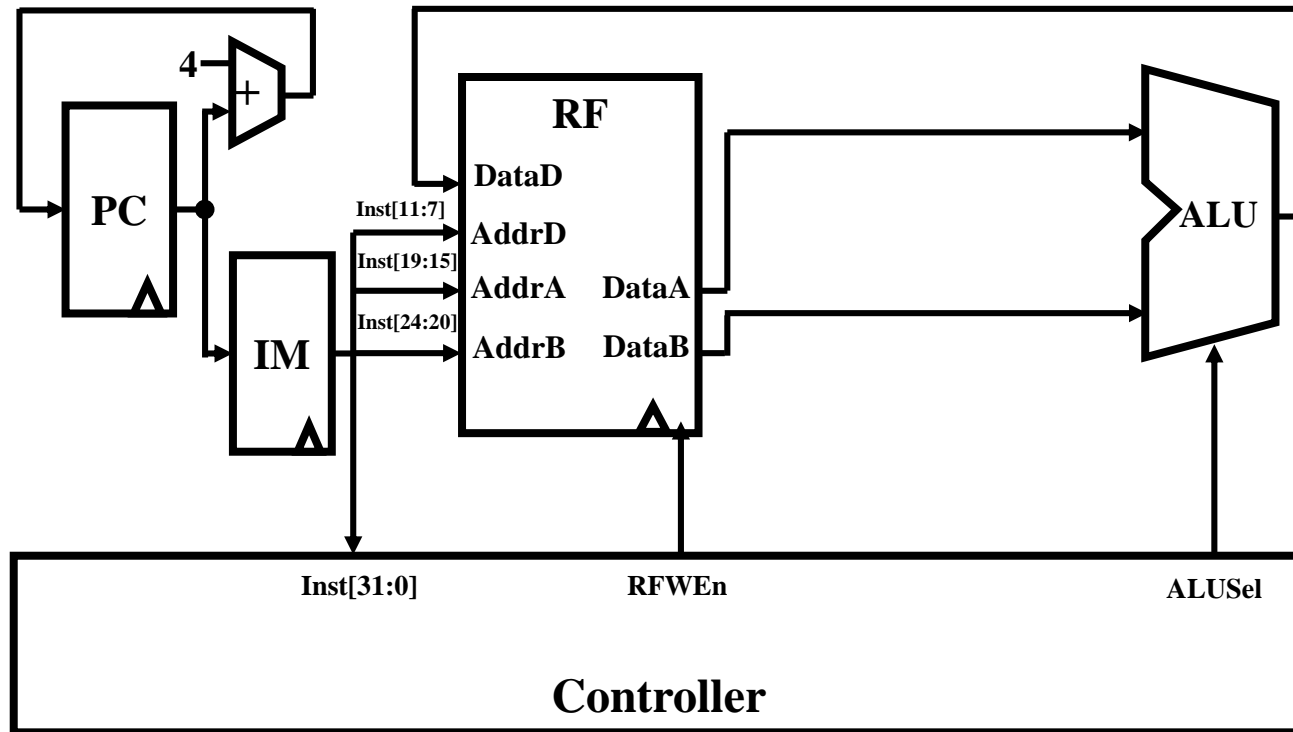
□ addi rd,rs1,imm

□ 指令功能

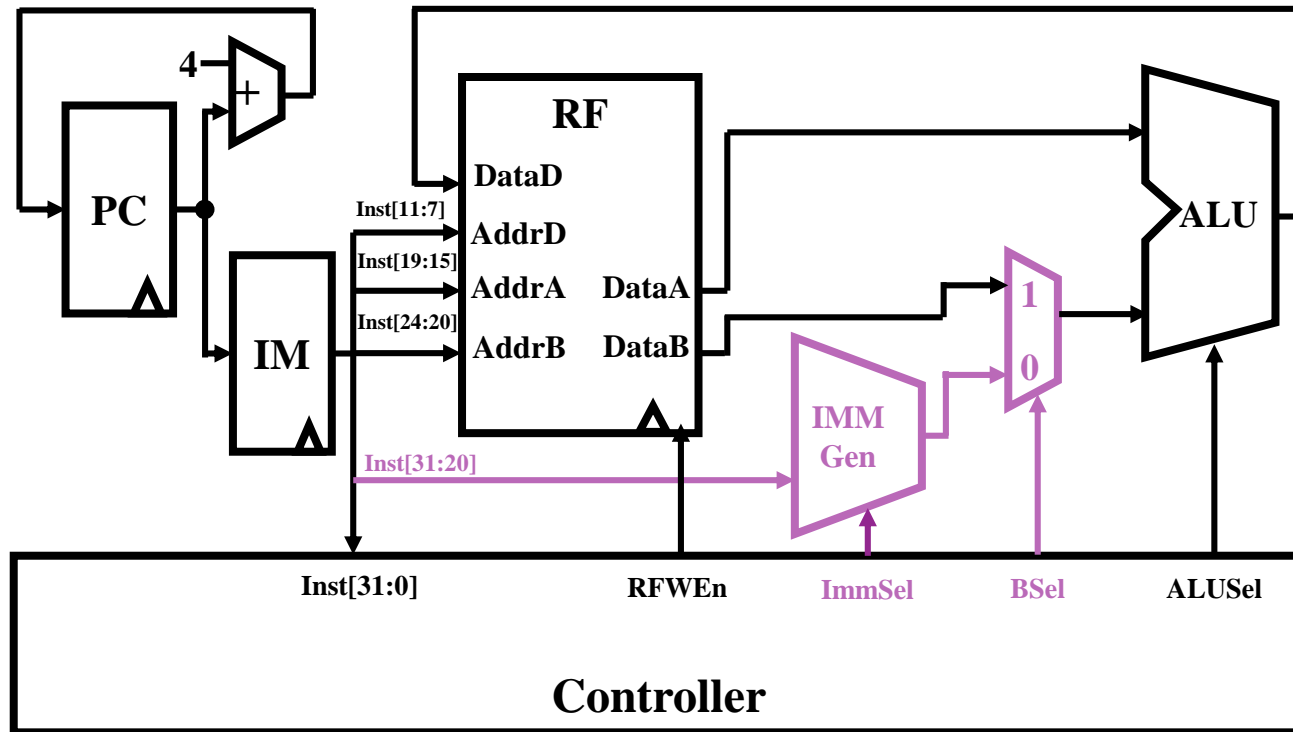
■  $R[rd] = R[rs1] + \text{SignExt}(imm)$



# 数据通路设计1



# 数据通路设计2

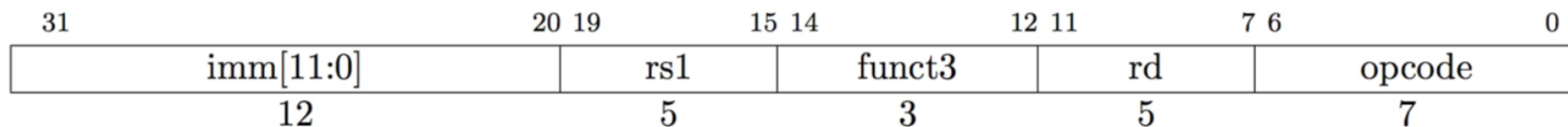


# 访存I型指令——Load系列

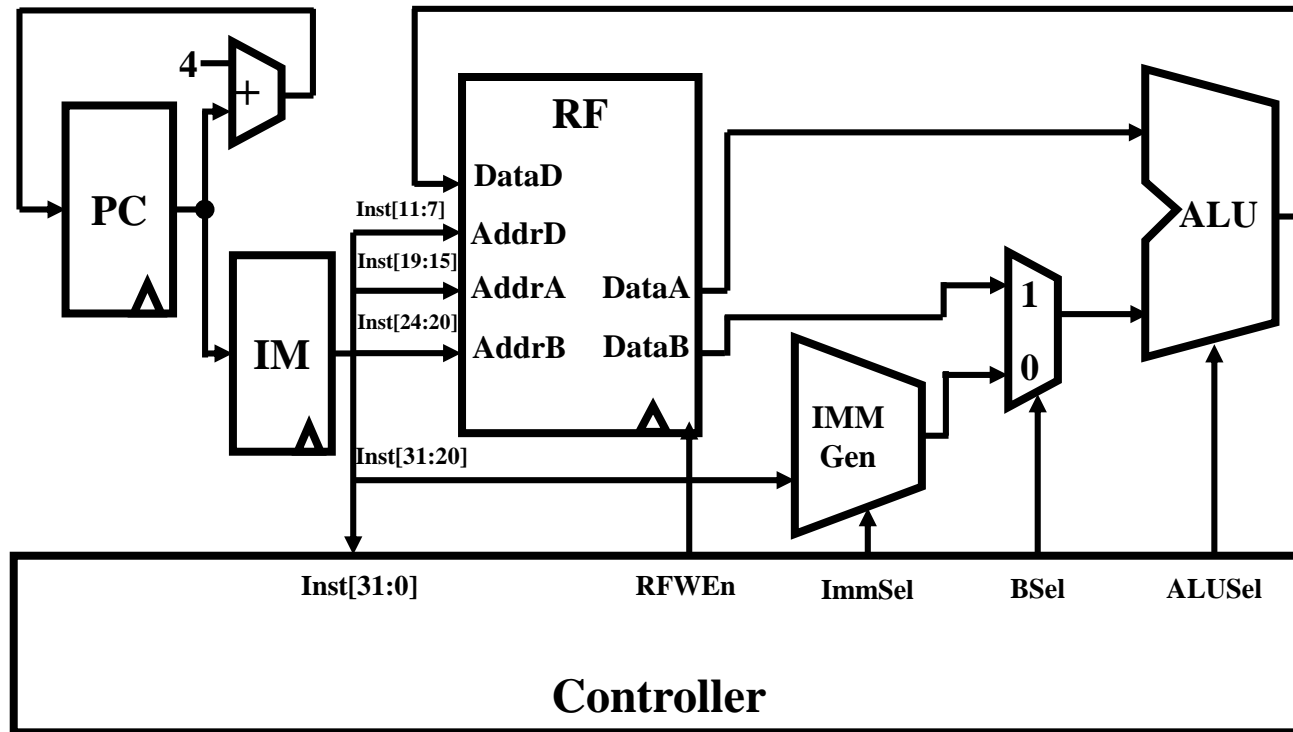
□ Load: lw rd rs1 imm

□ 指令功能

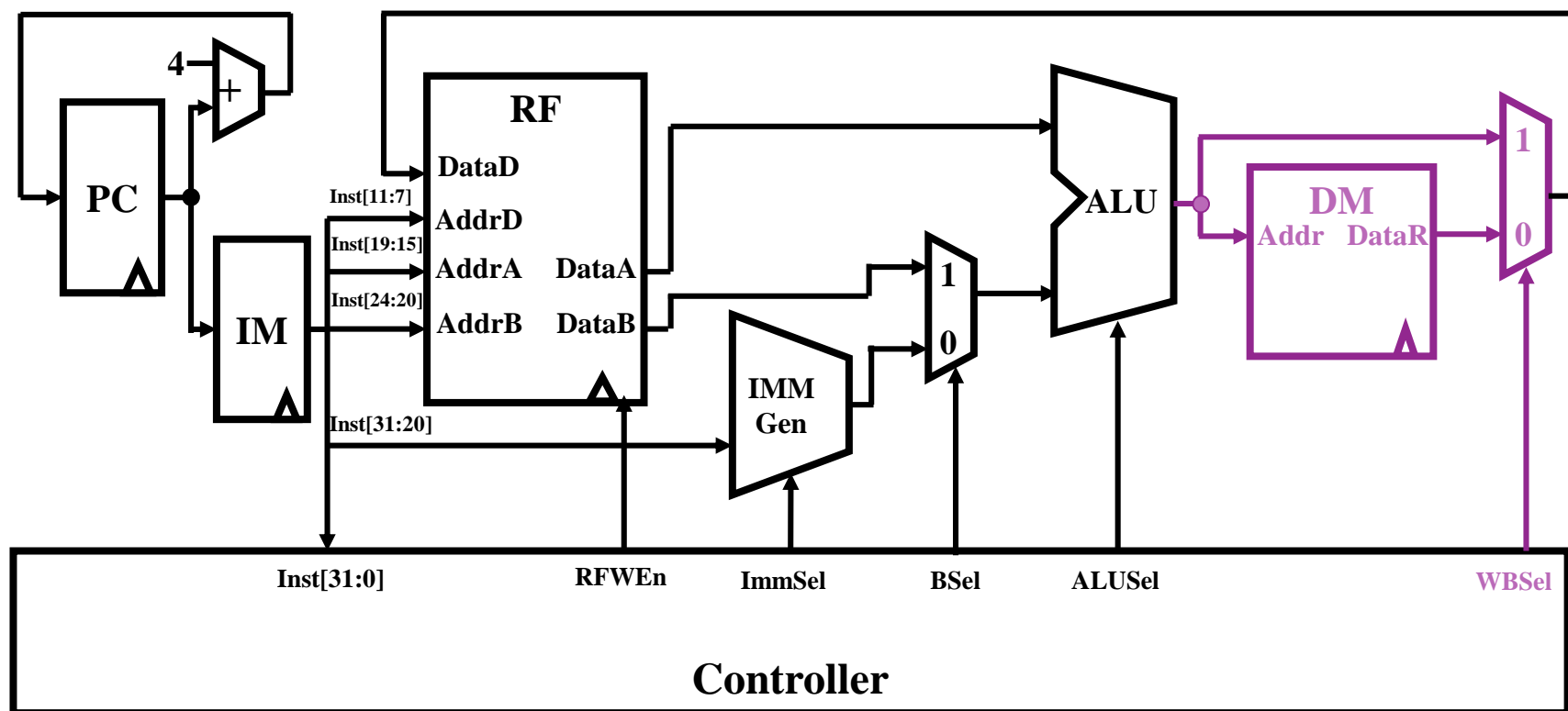
- $\text{Addr} = \text{R}[\text{rs1}] + \text{SignExt}(\text{imm})$
- $\text{R}[\text{rd}] = \text{MEM}[\text{Addr}]$



# 数据通路设计2



# 数据通路设计3

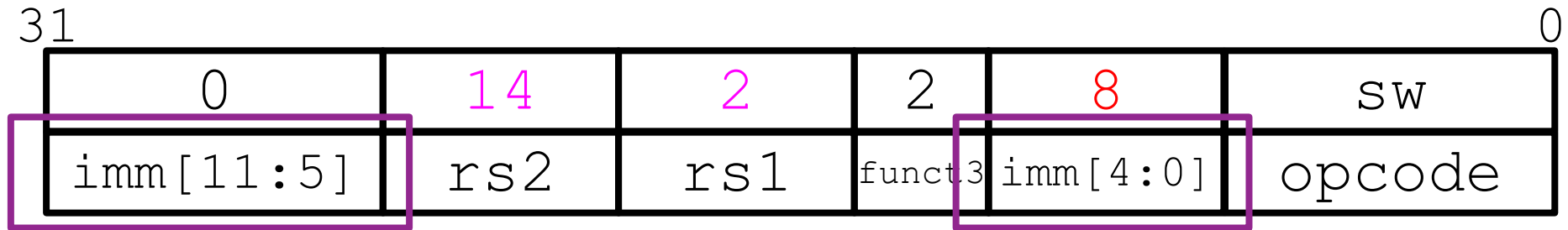


# S型指令——Store

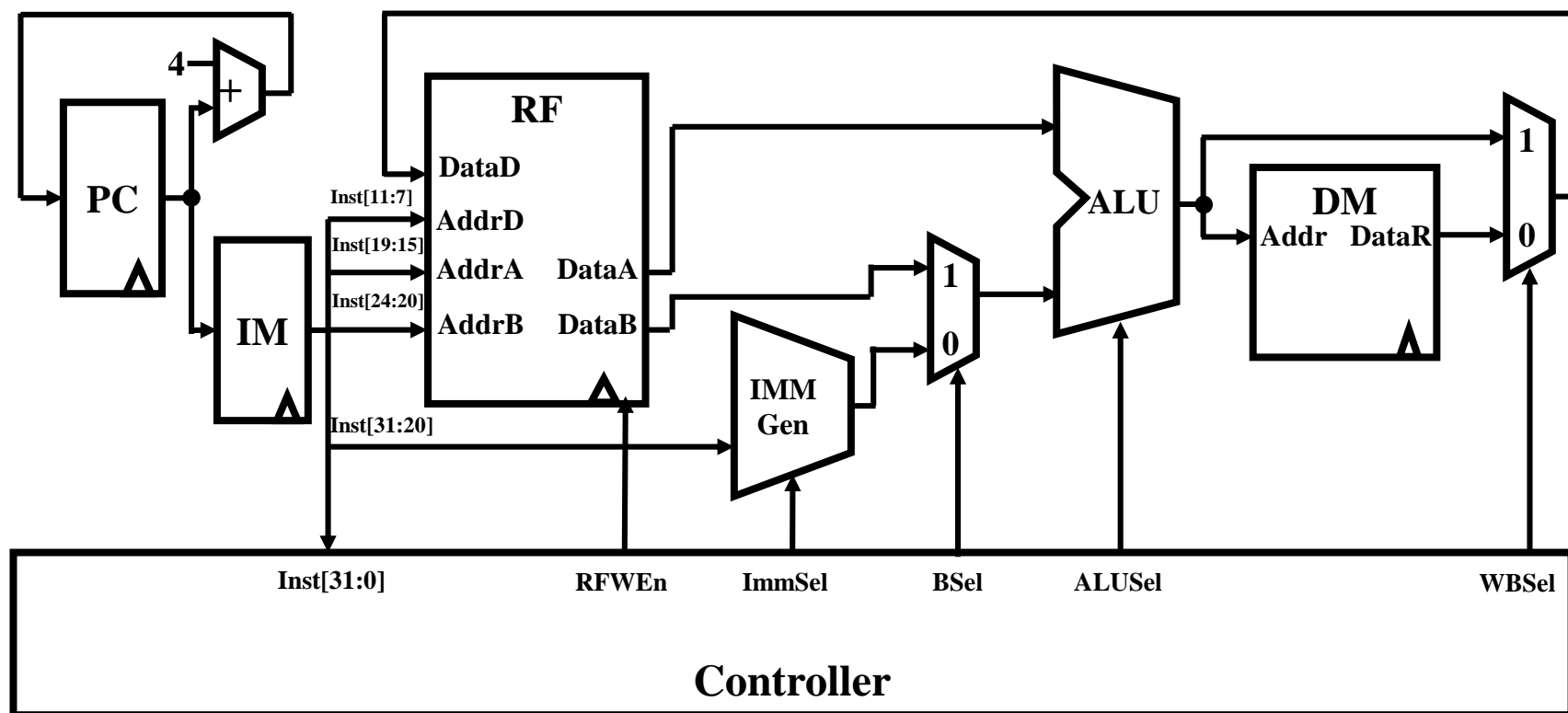
□ Store: sw rs2 rs1 imm

□ 指令功能

- $\text{Addr} = \text{R}[\text{rs1}] + \text{SignExt}(\text{imm})$
- $\text{MEM}[\text{Addr}] = \text{R}[\text{rs2}]$

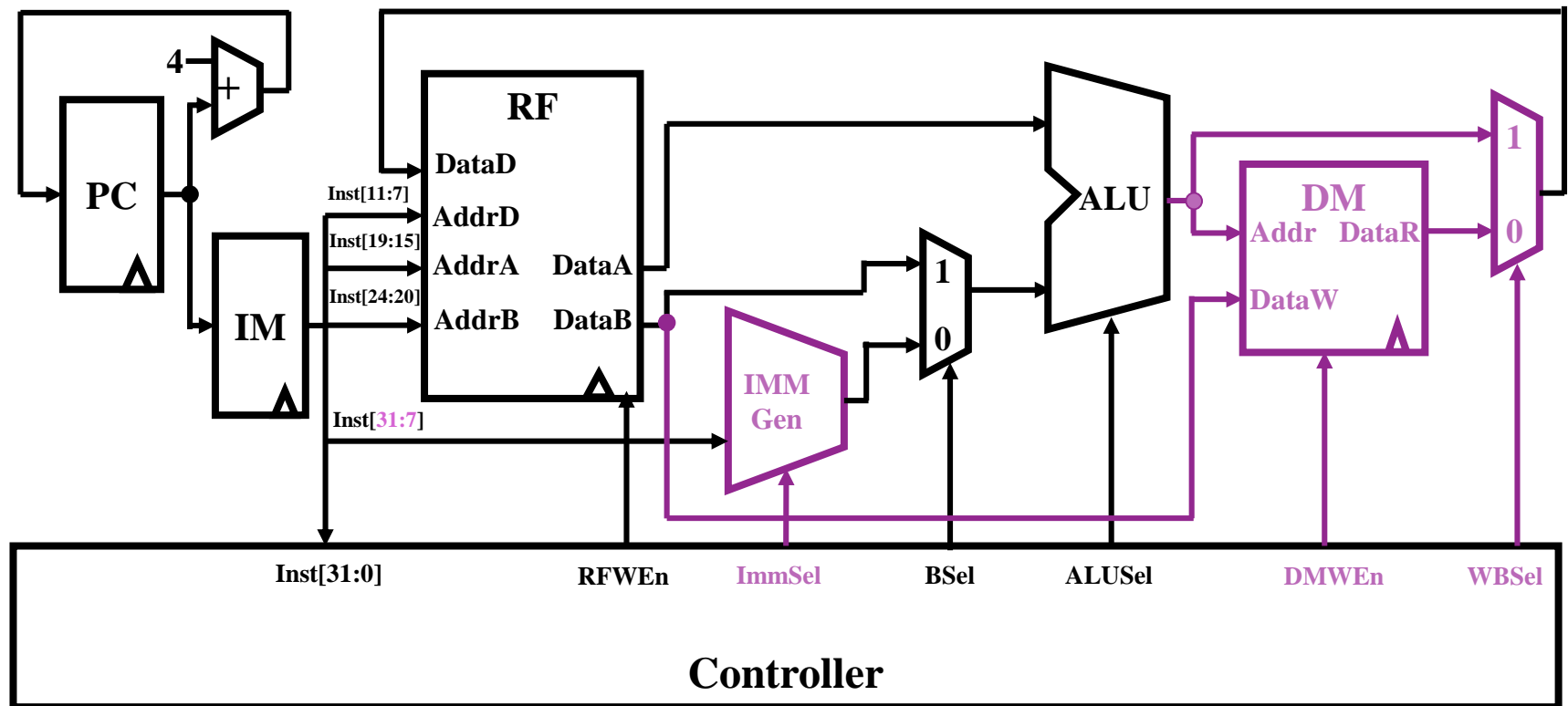


# 数据通路设计3





# 数据通路设计4

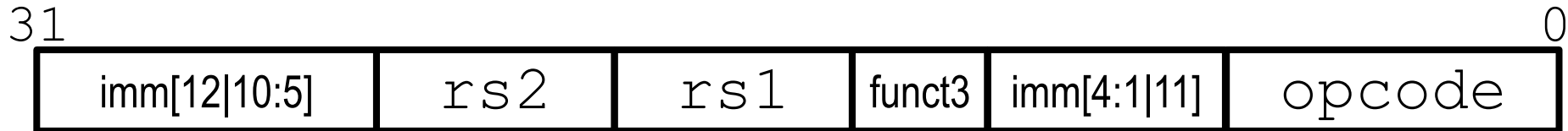


# B型指令——BEQ, etc.

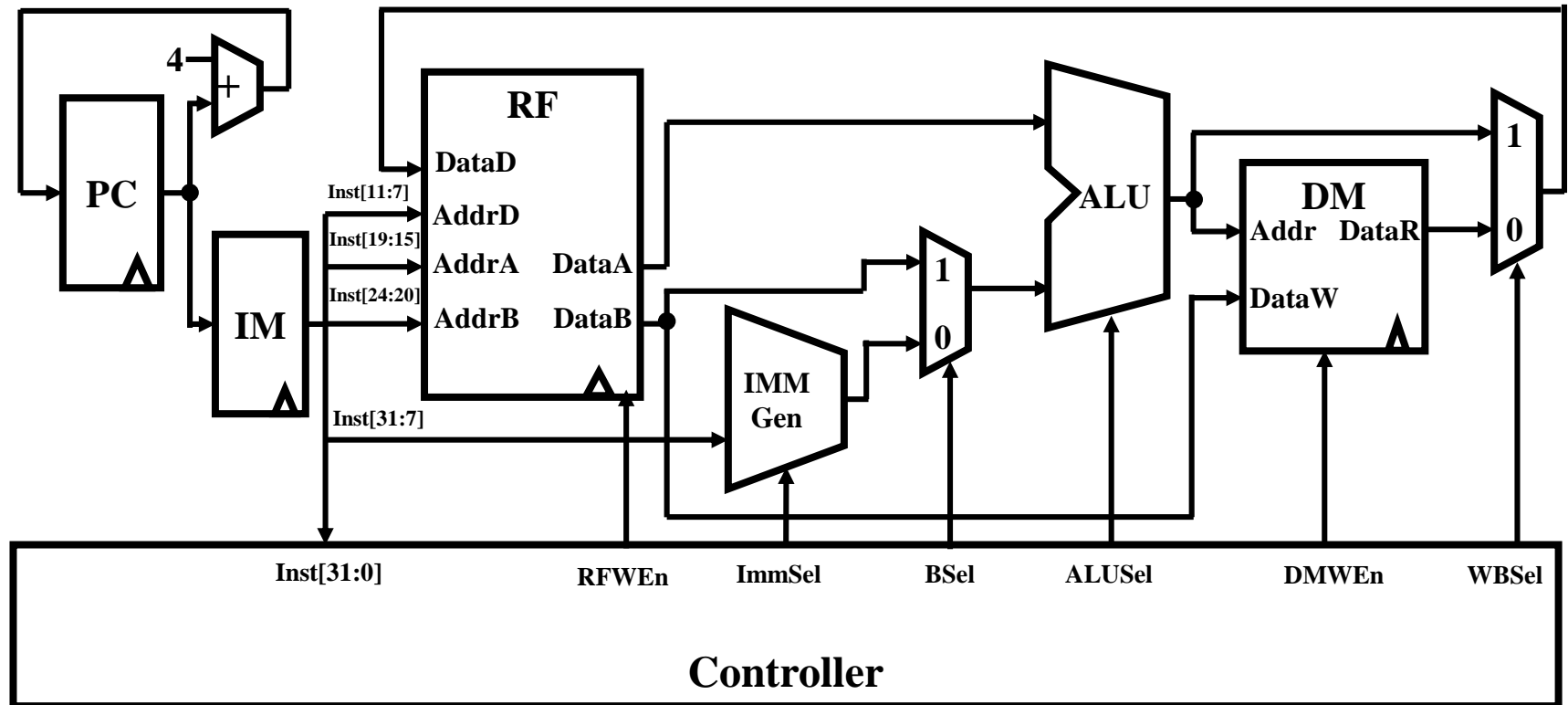
□ BEQ: beq rs1 rs2 label

□ 指令功能

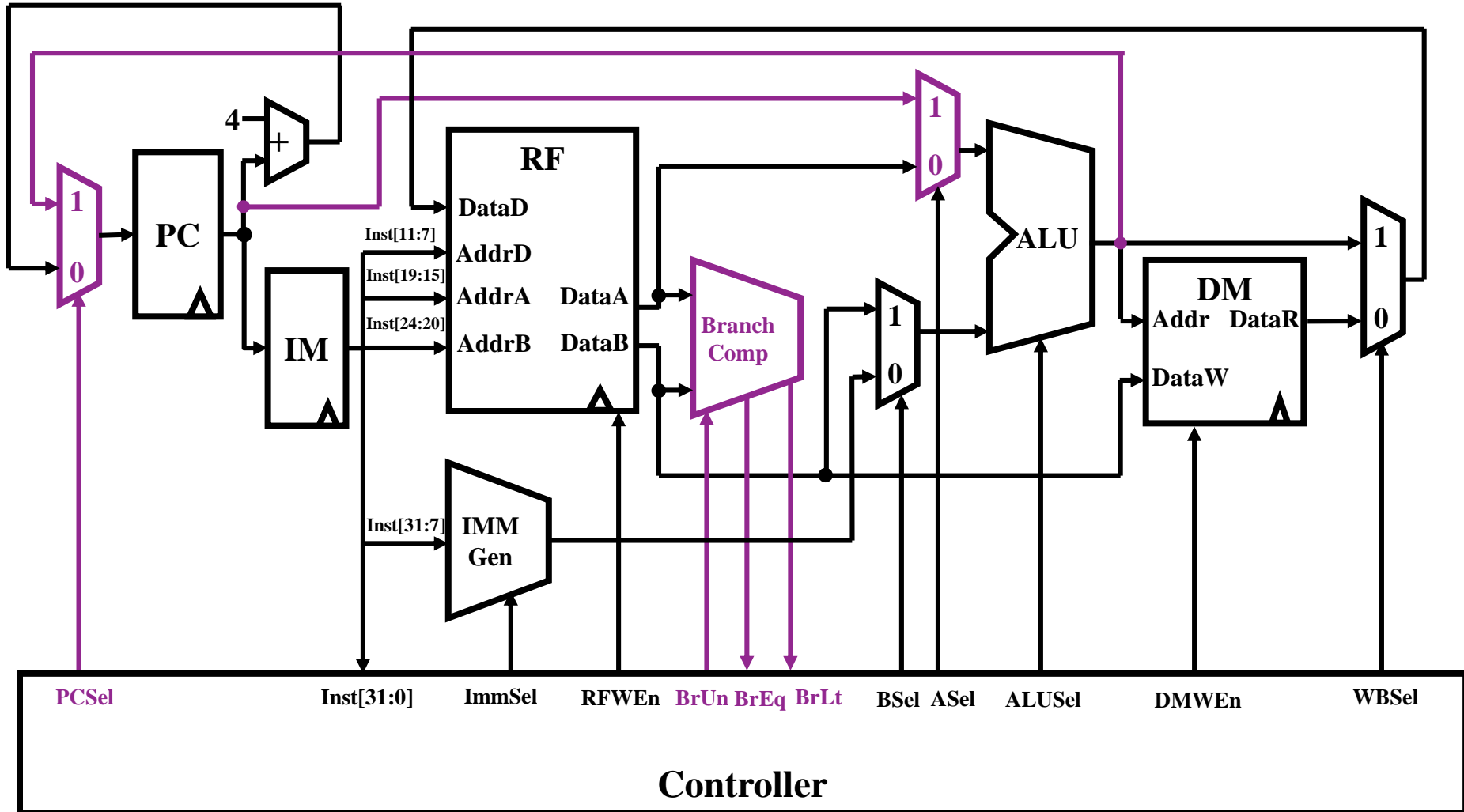
- if  $R[rs1] = R[rs2]$
- then  $PC \leftarrow PC + \text{SignExt}(\text{imm})$
- Else  $PC \leftarrow PC + 4$



# 数据通路设计4



# 数据通路设计5



# U型指令——lui, etc.

□ lui reg, imm

□ 指令功能

- 将20位的立即数装入到寄存器reg的高20位
- 将低12位清零

□ 对数据通路没有提出新的需求



# J型指令——jal, etc.

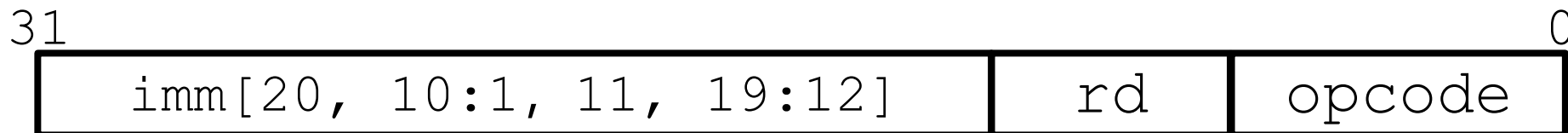
□ Jump: jal rd, imm

□ 指令功能:

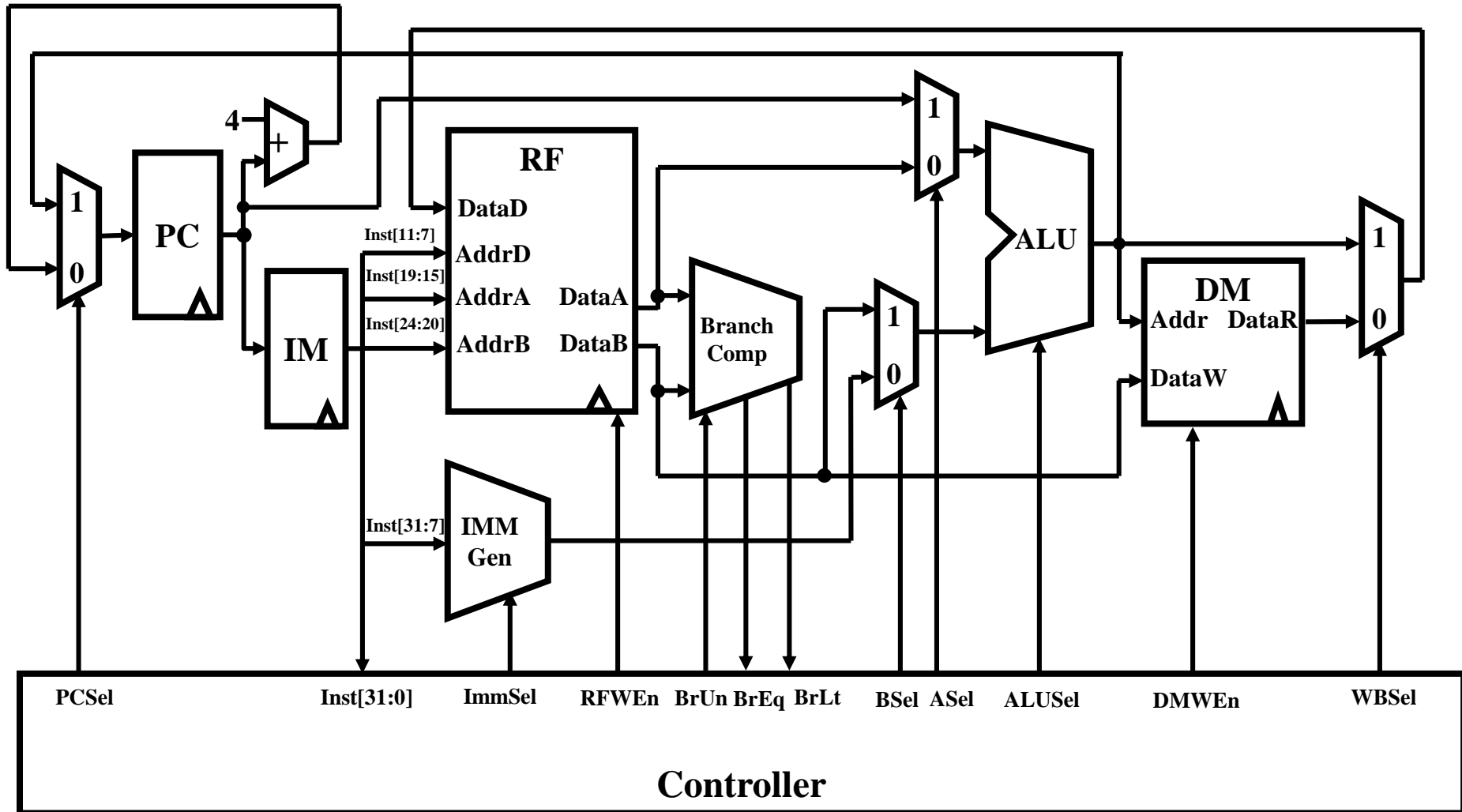
- 置rd值为返回地址 (PC+4)
- 设置跳转  $PC = PC + \text{offset}$

□ 思考: 是否缺少PC+4到RF的数据通路?

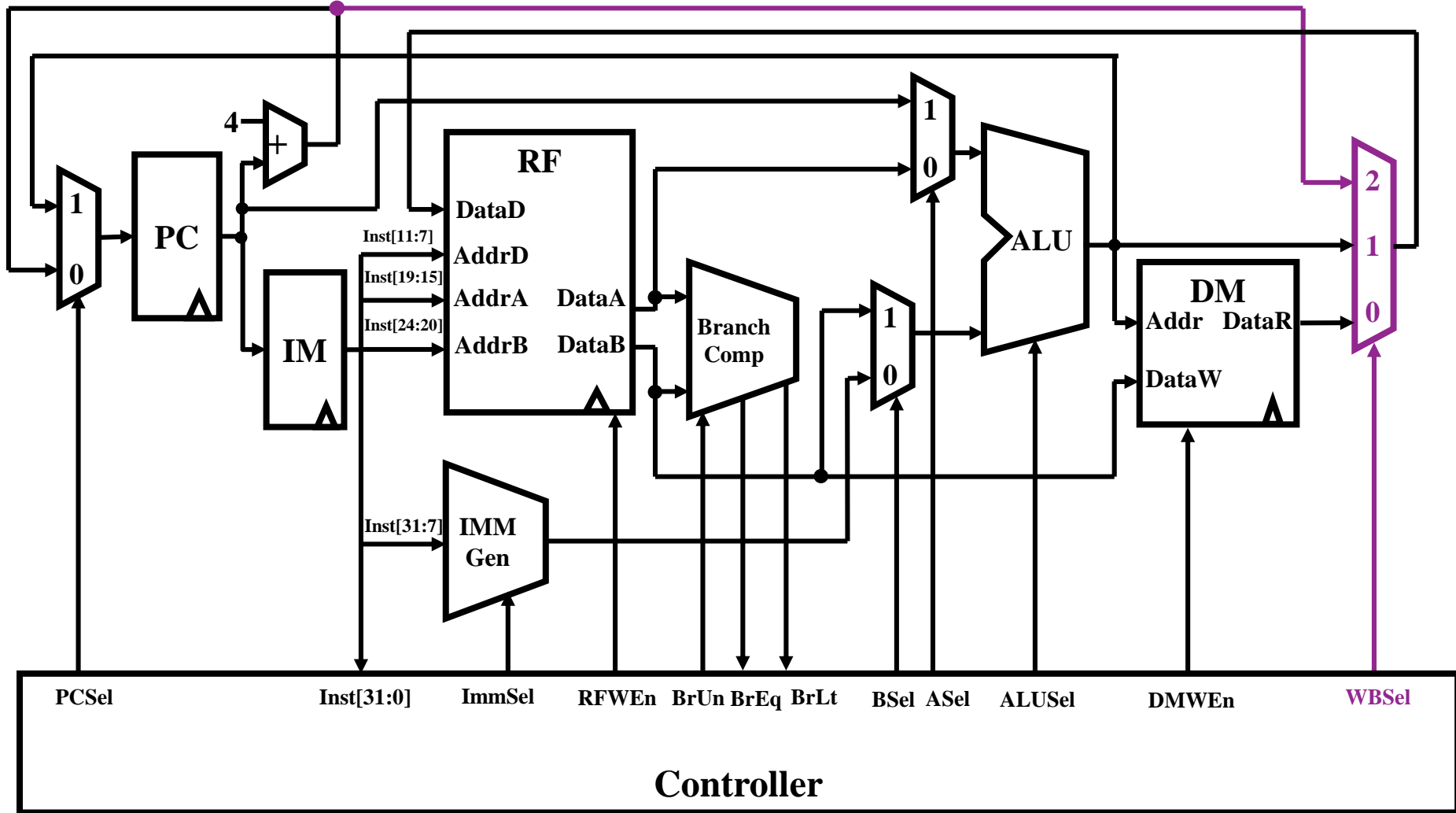
- 修改ALU?
- 新增数据通路?



# 数据通路设计5



# 数据通路设计6



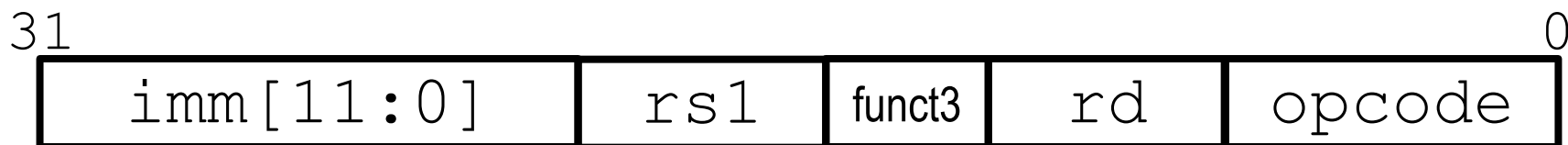


# 跳转I型指令——jalr, etc.

## □ Jalr rd, rs, immediate

- Rd: 返回地址 (PC+4)
- Imm: 偏移offset, 设置跳转  $PC = rs1 + offset$
- Immediate与 I类型指令中的算术和加载指令 一样

## □ 思考：现有数据通路是否可以满足该指令功能？



# 指令执行过程

---

- 取指令
- 分析指令
- 读取源操作数
  - 寄存器组
  - 立即数
- 计算
- 访存
- 写回寄存器组

# 指令功能如何实现？

---

## □ 硬件组成

- ALU
- MEM
- Register File
- ADDer
- PC
- Mux

## □ 数据通路

- 如何实现指令的功能？

# 指令的执行过程与控制

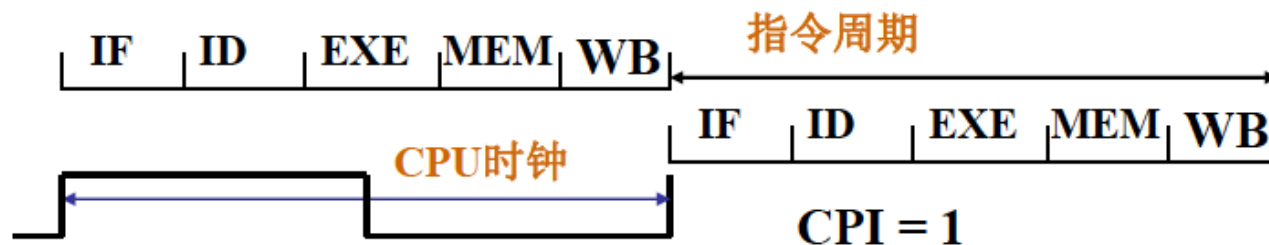
- 冯·诺依曼结构的计算机，即存储程序计算机，设置内存，存放程序和数据，在程序运行之前存入。
- 执行程序：
  - 正确从程序首地址开始；
  - 正确分步地执行每一条指令，
  - 还要形成下条待执行指令的地址；
  - 并保证自动地连续执行指令，
  - 直到结束程序的最后一条指令。
- 从主存储器读来一条指令，分析指令，按指令的功能要求完成执行过程，本条指令完成后自动开始下一条指令的执行过程，由硬件本身完成。

# 指令的执行步骤

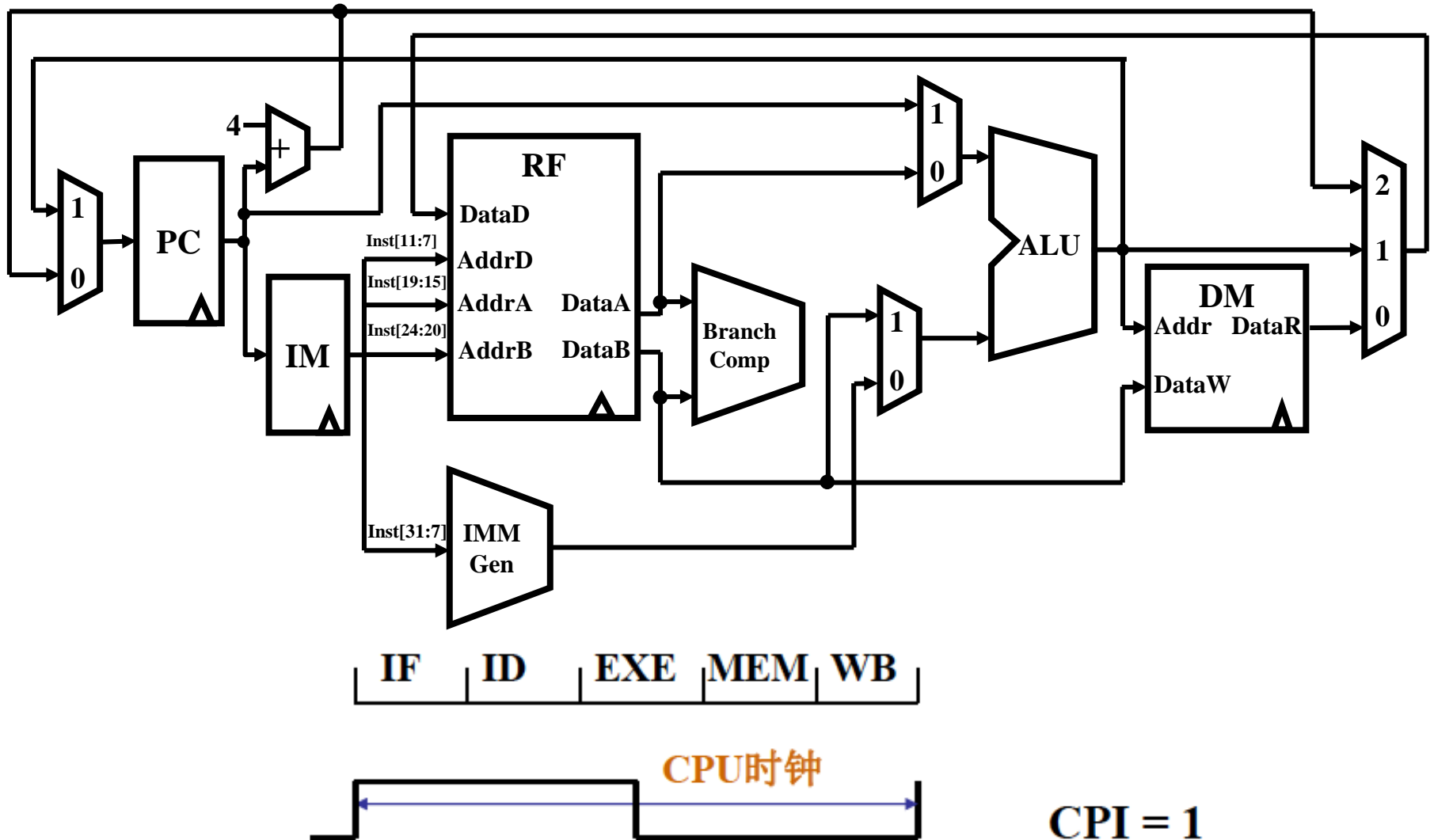
- 当前的计算机系统中，流行的做法是把一条指令的执行过程划分为如下的5 个阶段，是由指令的功能和计算机硬件结构共同决定的，有内在的逻辑关系。
  - 读取指令(IF), 从存储器读来指令并形成下条指令地址
  - 指令译码(ID), 指令译码，读寄存器堆为ALU准备数据
  - 执行运算(EXE), ALU 执行数据运算或计算存储器地址
  - 存储器读写(MEM), 完成存储器的读操作或者写操作
  - 写回(WB), 写ALU的结果或存储器读出数据到寄存器堆
- 从如何为不同指令安排这几个阶段，几个阶段如何衔接考虑，大体有3 种可行方案，各自都对计算机的内部结构、部件设置及其控制方式有着不同要求。

# 单周期CPU

- 计算机一条指令的执行时间被称为**指令周期**，一个CPU时钟时间被称为**CPU周期**(在某些计算机中，还可再把一个CPU周期区分为几个更小的步骤，称其为节拍)。执行**每条指令平均使用的CPU周期个数**被称为**CPI**
- 全部指令都选用**一个CPU周期**完成的系统被称为**单周期CPU**，指令串行执行，前一条指令结束后才启动下条指令。每条指令都用5个步骤的时间完成，控制各部件运行的信号在整个指令周期不变化。**单周期CPU**用于早期计算机，系统性能和资源利用率很低，相对当前技术变得**不再实用**。

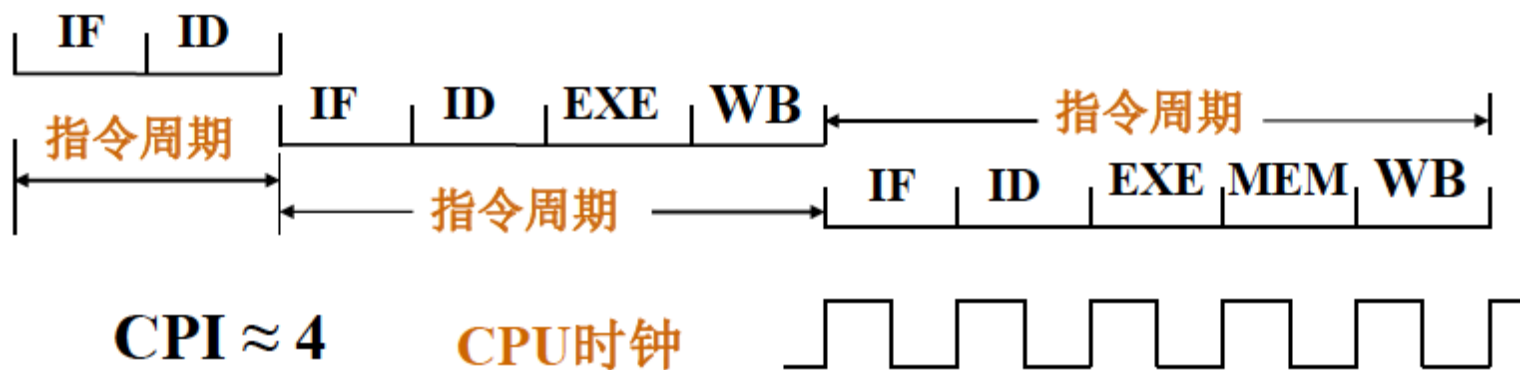


# 单周期CPU



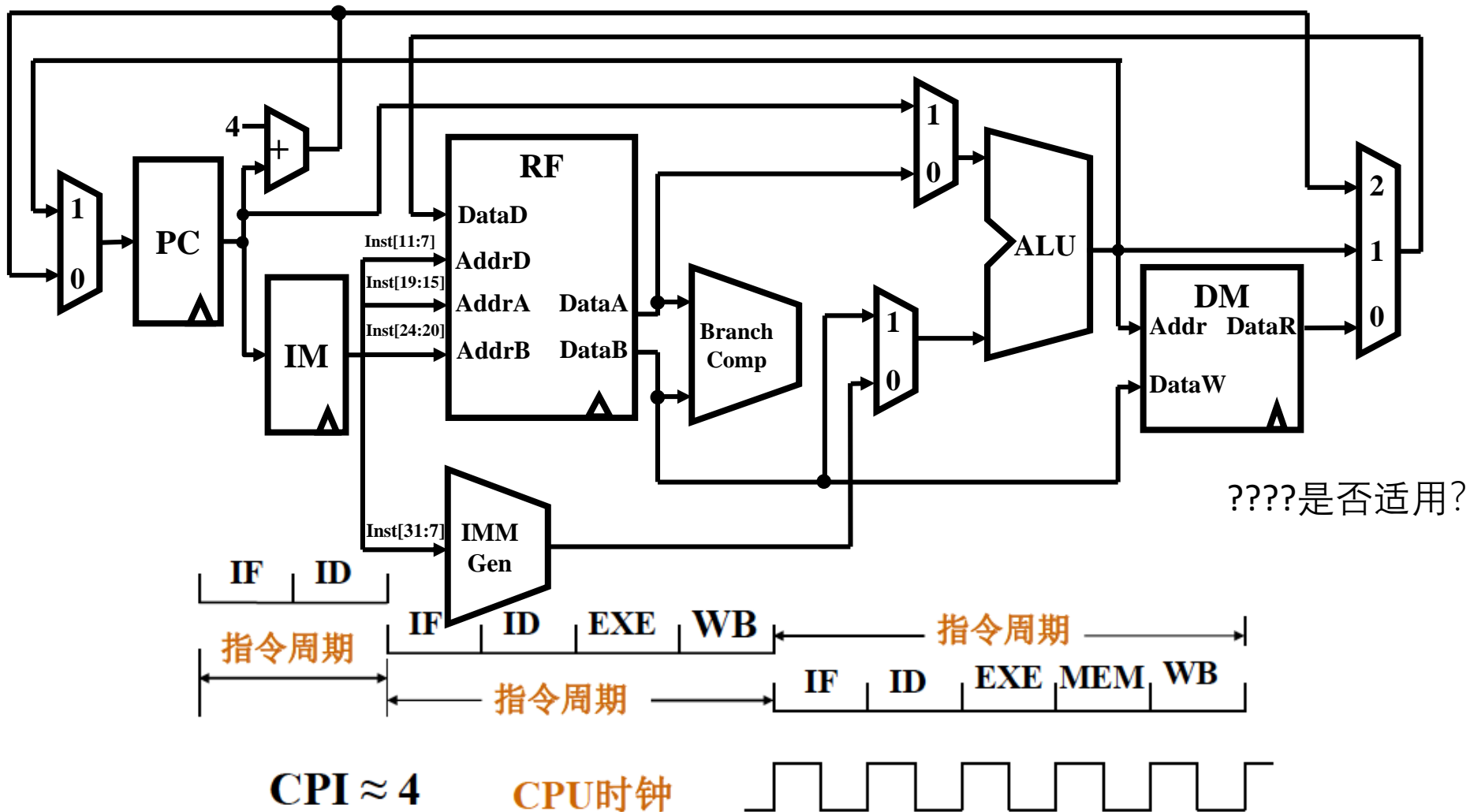
# 多周期CPU

- ❑ 计算机一条指令的执行时间被称为指令周期，一个CPU时钟时间被称为CPU周期。
- ❑ 依据不同指令各自的功能需求为其选择不等的执行步骤的系统被称为多周期CPU，控制各部件运行的控制信号随着指令执行步骤改变，系统性能和资源利用率更高。相邻指令可以完全串行执行，也可能部分时间重叠，多周期CPU（相比单周期CPU）更实用。



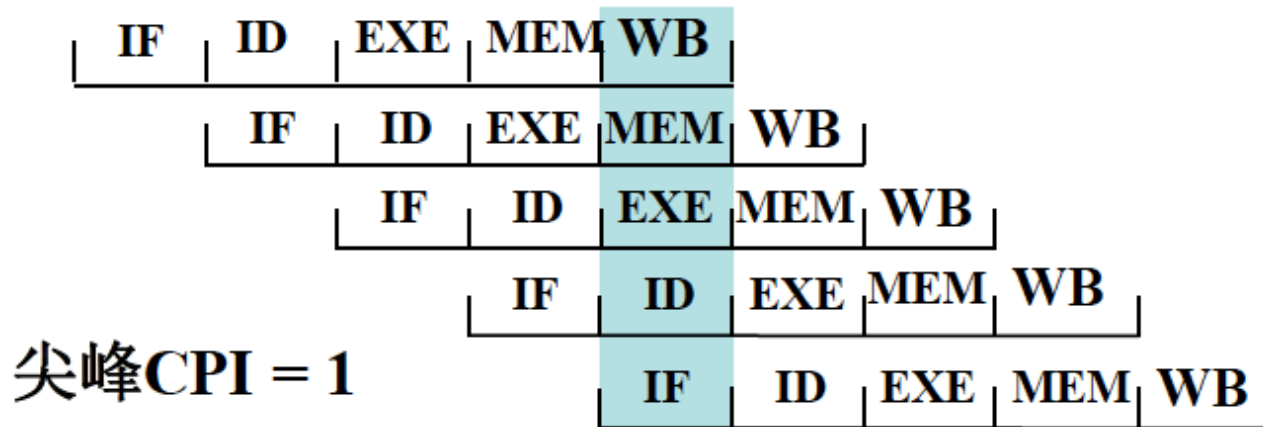


# 多周期CPU



# 指令流水线CPU

- 全部指令都是选用5个步骤完成，执行时间相同,但相邻指令的执行并不是完全串行的，执行时间有所重叠，例如每结束指令的一个执行步骤就启动下条指令，这被称为**指令流水线技术**，所有部件都高速运行，尖峰速度每个CPU时钟执行一条指令，系统性能和资源利用率更高，显著地提高系统的性能价格比，但**计算机结构和控制器的设计、实现略显复杂**。**当前计算机中普遍使用这种方案**。



# 小结

---

- 指令系统受到技术条件的制约
- 兼容性是指令系统的重要要求
- RISC，以简洁换取性能提高
- CISC，以丰富换取编程方便
- 指令执行
  - 单步骤串行
  - 多步骤串行
  - 流水

# 阅读和思考

---

□ 阅读

□ 思考

□ 实践

- 分析ThinPAD RV指令系统的特点，并对其指令格式进行分类
- 设计能实现ThinPAD RV指令系统的数据通路，尤其要注意专用寄存器的实现方法

---

谢谢