

# 第二章 RISC-V汇编及其指令系统

---

- 计算机中数的表示
- RISC-V概述
- RISC-V汇编语言
- **RISC-V指令表示**
- 案例分析



# 第二章 RISC-V汇编及其指令系统

---

- **RISC-V指令表示**
  - RISC-V六种指令格式
  - RISC-V寻址模式介绍



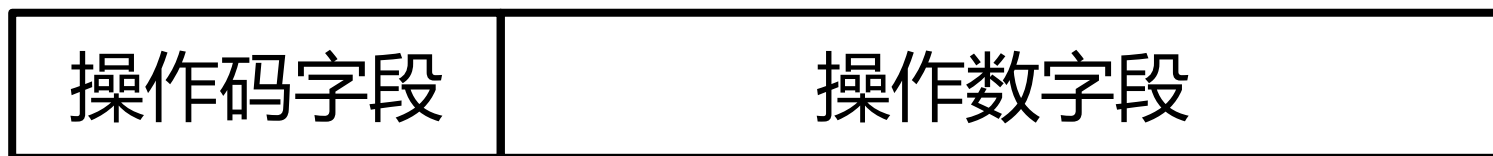
# RISC-V指令表示

- 指令字长、机器字长、存储字长 (二进制位数)
  - **指令字长**: 指令中二进制代码的总位数
  - **机器字长**: 直接处理的二进制位数, 一般等于寄存器总位数
  - **存储字长**: 一个存储单元存储的二进制代码的长度
- 如何表示指令?
  - 计算机只懂1和0 (汇编语言只是助记符)
  - 表示一个操作的一串二进制数, 称为指令字, 简称**指令**
  - RISC-V追求简单性 (指令与常用数据字长相同)
    - 用于RV32、RV64、RV128的32位指令相同

Bkpt	Address	Code	Basic	Source
<input type="checkbox"/>	0x00400000	0x00000513	addi x10, x0, 0	1: addi x10, x0, 0
<input type="checkbox"/>	0x00400004	0x06450513	addi x10, x10, 0x00000064	2: addi x10, x10, 100
<input type="checkbox"/>	0x00400008	0x00000293	addi x5, x0, 0	4: addi x5, x0, 0
<input type="checkbox"/>	0x0040000c	0x06428293	addi x5, x5, 0x00000064	5: addi x5, x5, 100
<input type="checkbox"/>	0x00400010	0x00500333	add x6, x0, x5	7: mv x6, x5

# 指令格式

- 指令格式：用二进制代码表示指令的结构形式
  - 要求计算机处理**什么数据**？
  - 要求计算机对数据做**什么处理**？
  - 计算机**怎样**才能**得到**要处理的**数据**？
- RISC-V指令包含两类字段：操作码字段和操作数字段
  - 操作码字段长度决定指令系统规模
  - 每条指令对应一个操作码



# RISC-V指令

- 指令中操作码和操作数可分成多个“字段”（一部分位）
  - 每个字段有特定的含义和作用
  - 理论上可以为每条指令定义不同的字段
- RISC-V定义了六种基本类型的指令格式：
  - R型指令                      用于寄存器 - 寄存器操作
  - I型指令                      用于短立即数和访存 load 操作
  - S型指令                      用于访存 store 操作
  - B型指令                      用于条件跳转/分支/转移操作
  - U型指令                      用于长立即数操作
  - J型指令                      用于无条件跳转操作

# RISC-V指令格式

- 会查表

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
B 型	Imm[12,10:5]	rs2	rs1	funct3	imm[4:1,11]	opcode
J 型	Imm[20,10:1,11,19:12]				rd	opcode
U 型	Imm[31:12]				rd	opcode

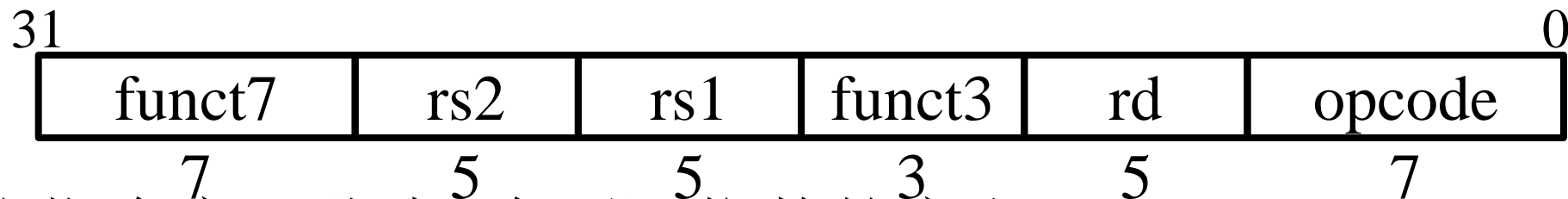
# RISC-V指令类型

---

- **R-type**: **R**egister-register arithmetic/logical operations
- **I-type**: register-**I**mmEDIATE arith/logical operations & loads
- **S-type** (S型) : for **S**tore
- **B-type** (B型) : for **B**ranches(**SB**型?)
- **U-type** (U型) : for 20-bit **U**pper immediate instructions
- **J-type** (J型) : for **J**umps (**UJ**型?)
- 其他: Used for OS & Synchronization

# R型指令

**op rd, rs1, rs2**



- 32位指令字，分为6个不同位数的字段
- opcode(操作码): 指定它是哪类指令
  - 对于所有R型指令，此字段等于**0110011**
- funct7+funct3: 这两个字段结合操作码描述要执行的操作

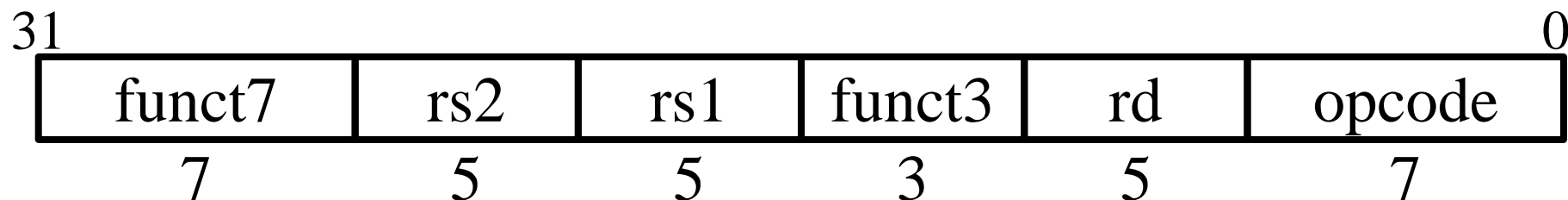
inst[4:2]	000	001	010	011	100	101	110	111
inst[6:5]								(> 32b)
00	LOAD	LOAD-FP	custom-0	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	custom-1	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	reserved	custom-2/rv128	48b
11	BRANCH	JALR	reserved	JAL	SYSTEM	reserved	custom-3/rv128	≥ 80b

Table 24.1: RISC-V base opcode map, inst[1:0]=11



# R型指令

**op rd, rs1, rs2**



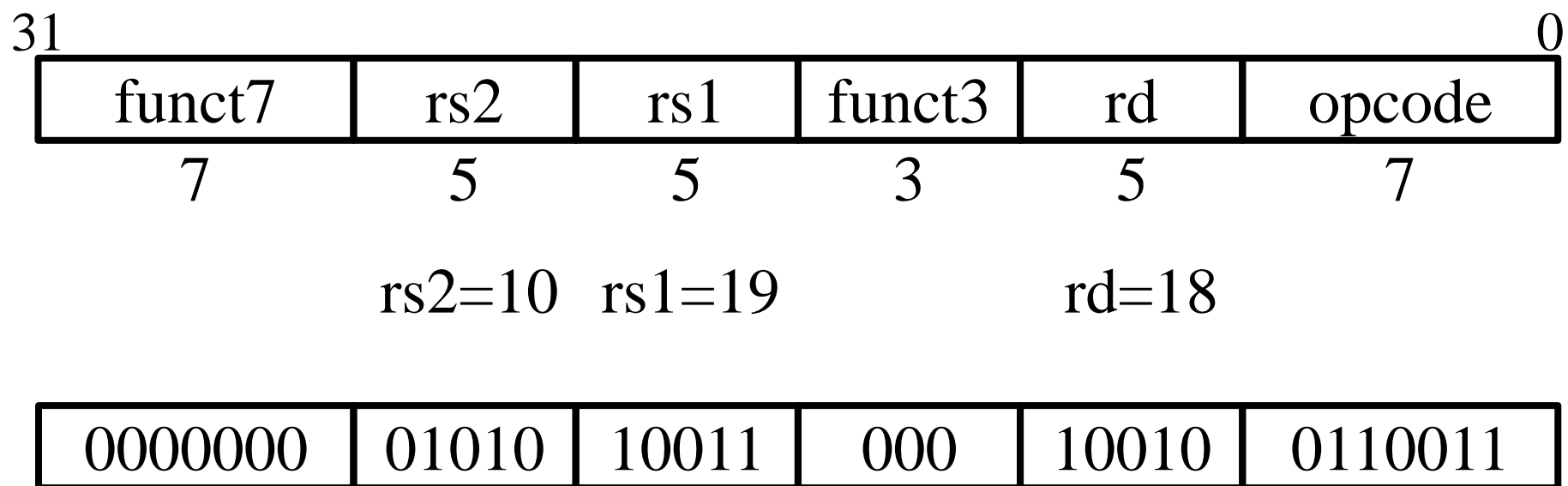
- rs1（源寄存器1）：指定第一个寄存器操作数
- rs2：指定第二个寄存器操作数
- rd（目的寄存器）：指定接收计算结果的寄存器
- 每个字段保存一个5位无符号整数（0-31），对应于一个寄存器号（x0-x31）（寄存器约定详见P75图2-14）

# R型指令

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

# R型指令格式示例

- RISC-V汇编指令 `add x18, x19, x10`



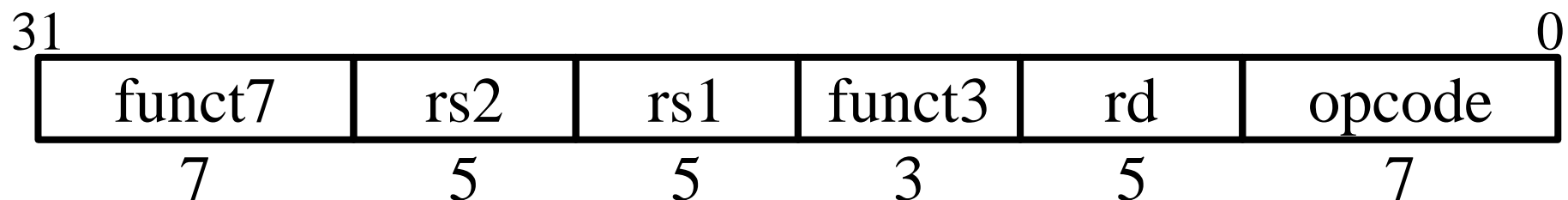
add x8, sp, zero 对应的机器指令码为\_\_\_\_\_。

提交

- ☐ A 0000 0000 1000 0001 0000 0000 0011 0011
- ☒ B 0000 0000 0000 0001 0000 0100 0011 0011
- ☐ C 0000 0000 1000 0001 0000 0000 0011 0011
- ☐ D 0000 0000 1000 0010 0000 0000 0011 0011

# R型指令格式示例

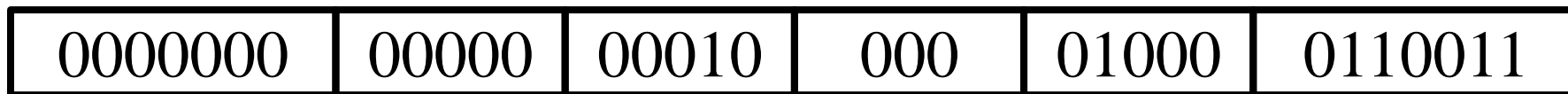
- RISC-V汇编指令 `add x8, sp, zero`  $\longleftrightarrow$  `add x8, x2, x0`



rs2=0

rs1=2

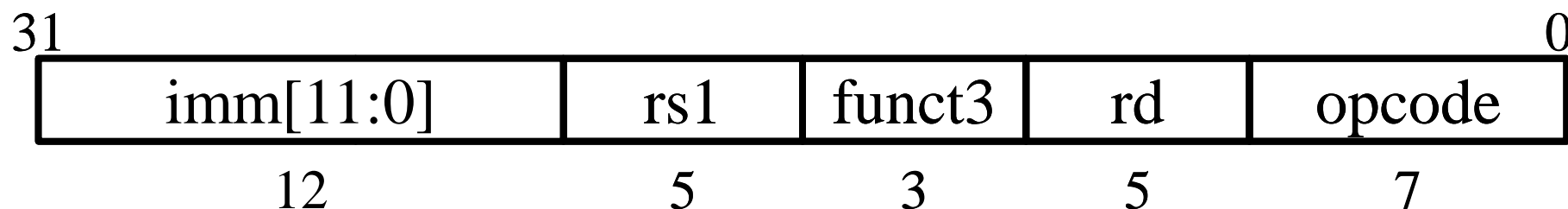
rd=8



答案为B: 0000 0000 0000 0001 0000 0100 0011 0011

# I型指令

**opi** rd, rs1, imm



- rs1、funct3、rd、opcode字段与R型相同
- rs2和funct7被12位有符号立即数imm[11:0]替换
- 在算术运算前，立即数总是进行符号扩展



- 立即数的表示范围？
- 如何处理超过12位的立即数？

# I型指令

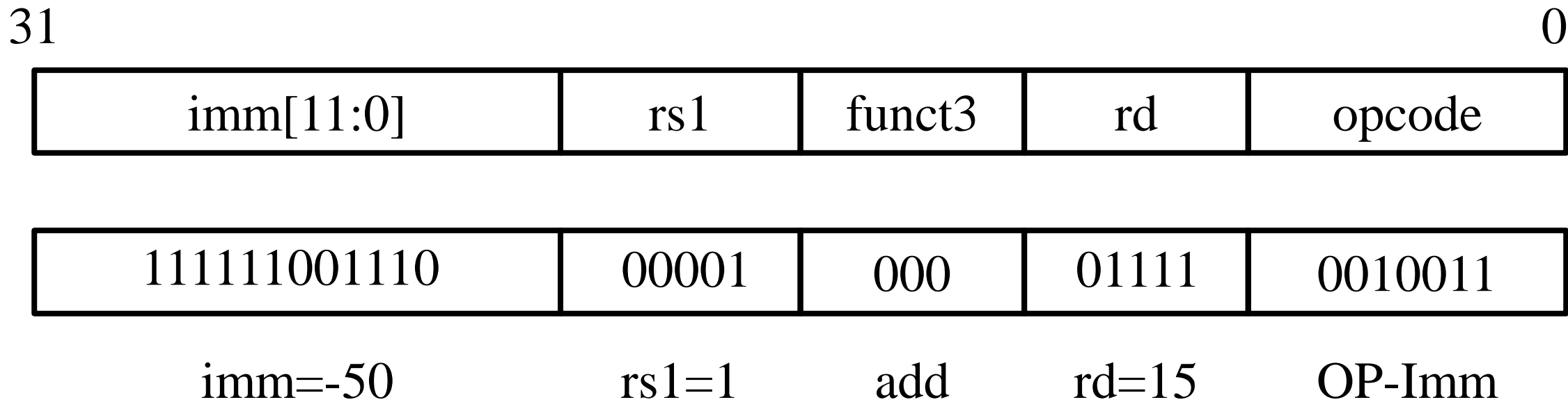
imm[11:0]		rs1	000	rd	0010011	addi
imm[11:0]		rs1	010	rd	0010011	slti
imm[11:0]		rs1	011	rd	0010011	sltiu
imm[11:0]		rs1	100	rd	0010011	xori
imm[11:0]		rs1	110	rd	0010011	ori
imm[11:0]		rs1	111	rd	0010011	andi
000000	shamt	rs1	001	rd	0010011	slli
000000	shamt	rs1	101	rd	0010011	srli
010000	shamt	rs1	101	rd	0010011	srai

其中一个高阶立即数位用于区分  
“逻辑右移”（SRLI）和“算术  
右移”（SRAI）

“按立即数移位”指令仅将立  
即数的低位6位用作移位量（最  
多只能移位63次）

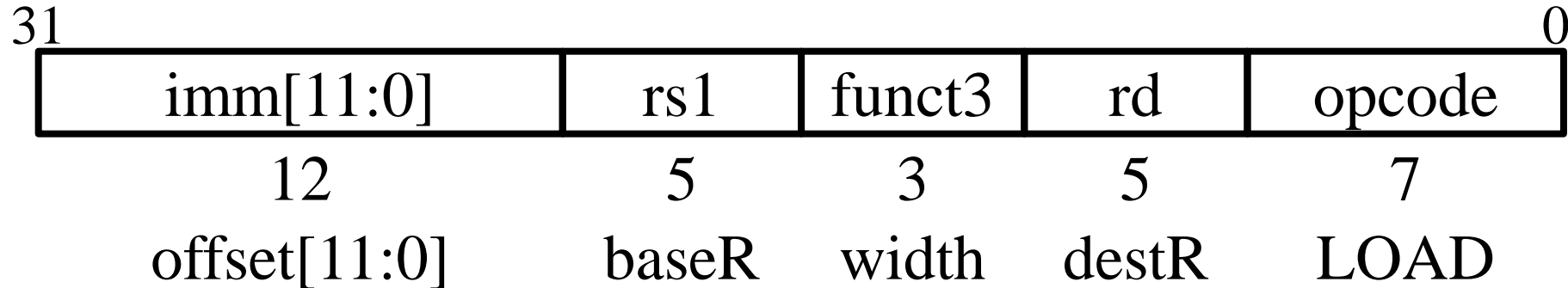
# I型指令示例

addi x15, x1, -50





# I型——Load类指令 **memop rd, off(rs1)**



- Load类指令是I型指令
- 12位有符号立即数加寄存器rs1的基址，形成内存地址
  - 这与add immediate操作非常相似，但用于创建地址
- 从内存读取到的值存储在寄存器rd中
- width表示装载的数据位数宽度和是否考虑符号

# I型Load类指令

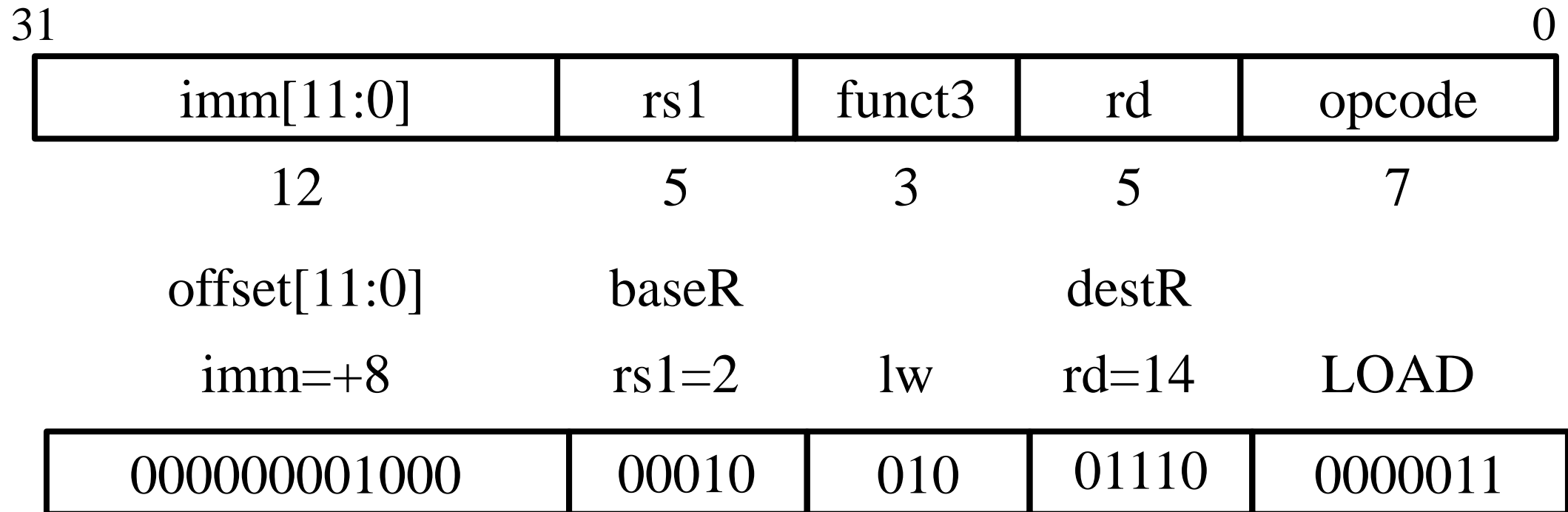
imm[11:0]	rs1	000	rd	0000011	lb
imm[11:0]	rs1	001	rd	0000011	lh
imm[11:0]	rs1	010	rd	0000011	lw
imm[11:0]	rs1	011	rd	0000011	ld
imm[11:0]	rs1	100	rd	0000011	lbu
imm[11:0]	rs1	101	rd	0000011	lhu
imm[11:0]	rs1	110	rd	0000011	lwu

funct3字段对读取数据的大小  
和“有符号性”进行编码

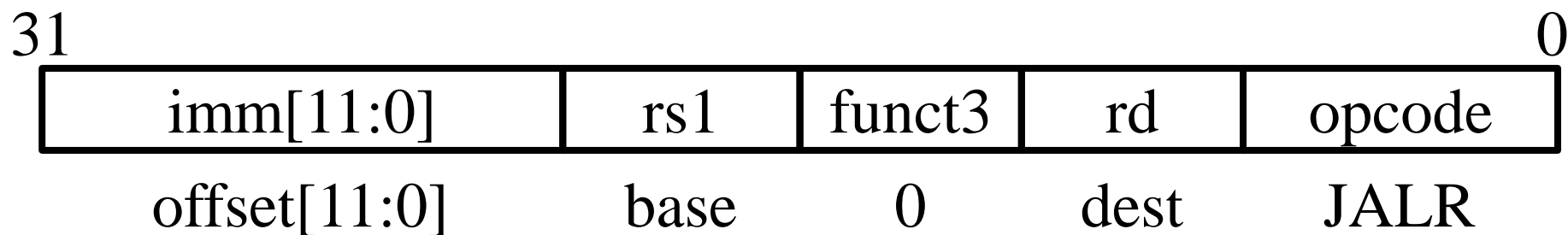
- LBU是“读取无符号字节”
- LH是“读取半字”，符号扩展以填充32/64位寄存器
- LHU是“读取无符号半字”，零扩展以填充32/64位寄存器

# I型Load类指令示例

lw x14, 8(x2)

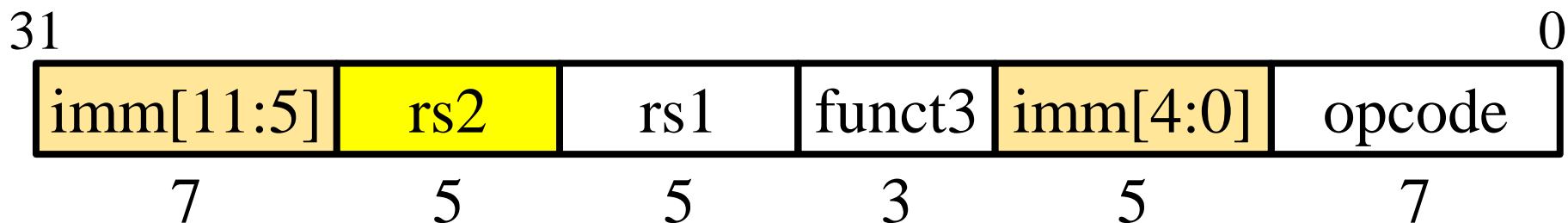


# I型指令——jalr指令



- JALR属于I型指令： `jalr rd, offset(rs1)` #jump and link register
- 将PC + 4保存在rd中
- 设置  $PC = rs1 + immediate$
- 与load指令得到地址的方式类似

# S型指令



- **memop** **rs2**, **offset(rs1)**
- 存储指令需要读取两个寄存器，**rs1**为内存地址，rs2为要写入的数据，以及立即偏移量offset
- **无rd**：S型指令不会将值写入寄存器
- RISC-V将**立即数的低5位**放置到其他类型指令的rd字段所在的位置——保持rs1/rs2字段在同一位置

# S型指令

- RISC-V中S型指令： sb、 sh、 sw、 sd

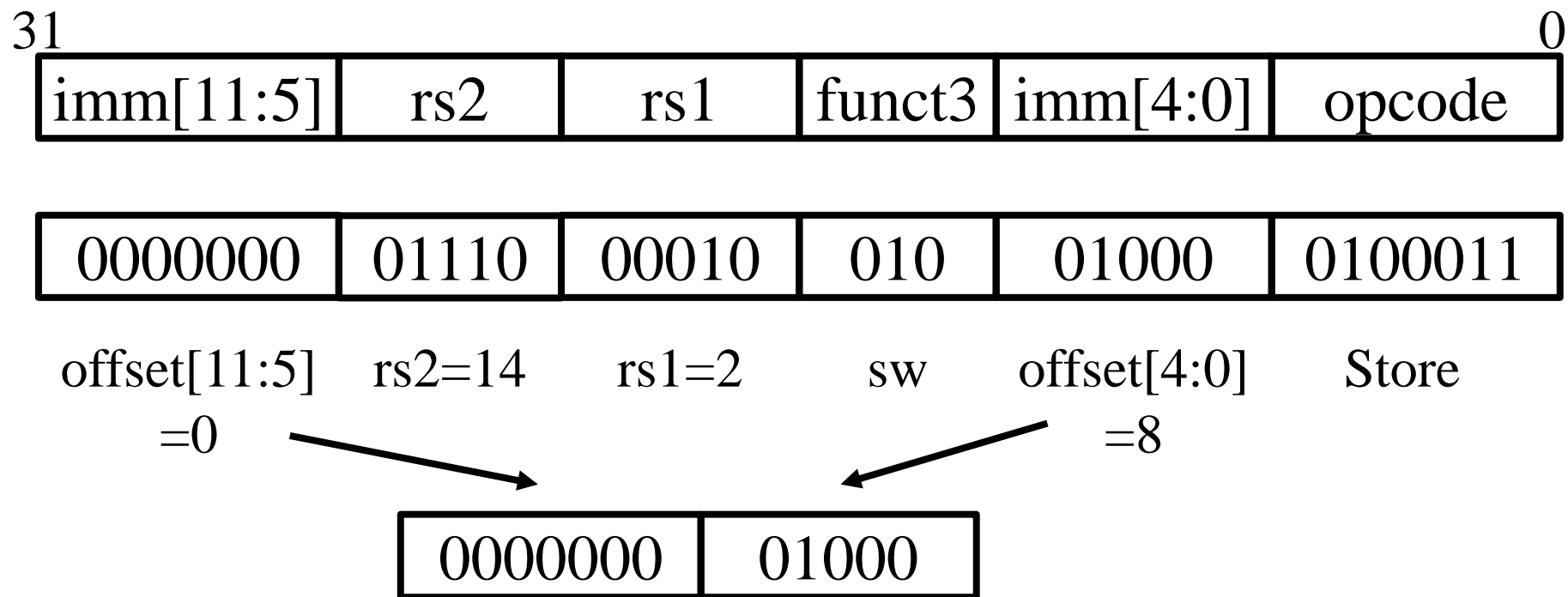
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	sb
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	sh
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	sw
imm[11:5]	rs2	rs1	011	imm[4:0]	0100011	sd

“黑书”图2-18（P83）中funct3错误，封底sd的类型“标错”。  
请参考“RV手册2019版P131”

# S型指令示例

- RISC-V汇编S型指令

sw x14, 8(x2)



# B型指令

- 条件分支指令：beq、bne、blt、bge、bltu、bgeu
  - beq rs1, rs2, Label
- B型指令读取两个寄存器，但不写回寄存器
  - 类似于S型指令



如何对Label进行编码（用哪些字段）？转移的目标位置如何得到？



# B型指令

---

- 用于条件语句或循环语句（if-else、while、for）
  - 条件或循环体通常很小（<50条指令）：SPEC基准测试中约有一半条件分支跳转距离小于16条指令
  - 最大分支转移距离与代码量有关
  - 当前指令的地址存储在程序计数器（PC）中

# B型指令



- B型指令格式与S型指令格式基本相同 (**imm[12:1]**)
- PC相对寻址：将立即数字段(12位)作为相对PC的补码偏移量
  - 若偏移量以**字节**为单位：范围**约**为 $\pm 2^{11}$ 字节
  - 若偏移量以“**字**”为单位：范围**约**为 $\pm 2^{11} \times 4$ 字节
- 基于RISC-V的扩展指令集支持16位压缩编码指令
- **折中**：B型指令偏移量以**半字**为单位(2字节增量)
  - 偏移量范围约为 $\pm 2^{11} \times 2$ 字节(**-4096到+4094字节**)
  - 用12位立即数字段表示13位有符号字节地址的偏移量，而偏移量的最低位始终为零，因此无需存储

# B型指令

- RISC-V汇编指令:

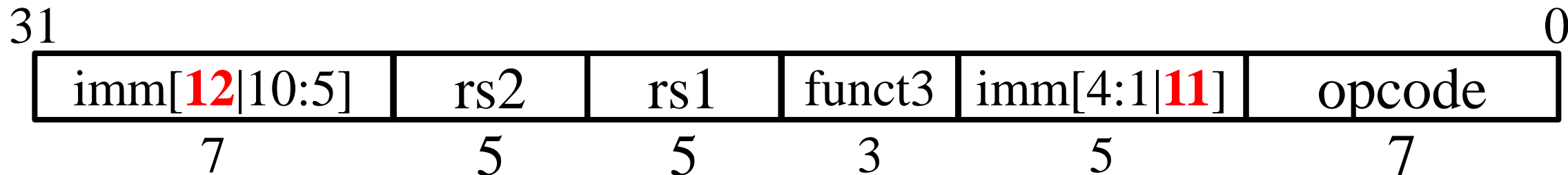
- Loop: `beq x19, x10, End`

- `add x18, x18, x10`

- `addi x19, x19, -1`

- `j Loop`

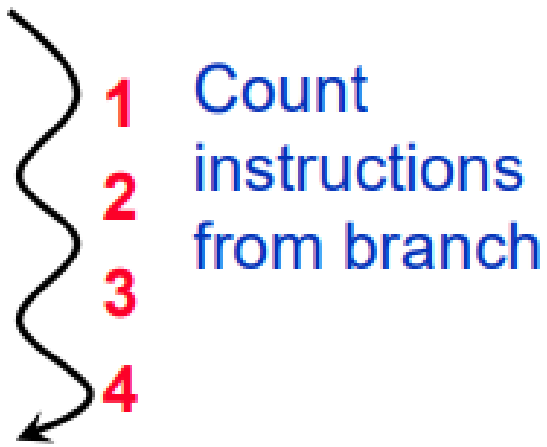
- `End:   # 目标指令`



# B型指令

- RISC-V汇编指令:

- Loop:   beq x19, x10, End  
          add x18, x18, x10  
          addi x19, x19, -1  
          j Loop

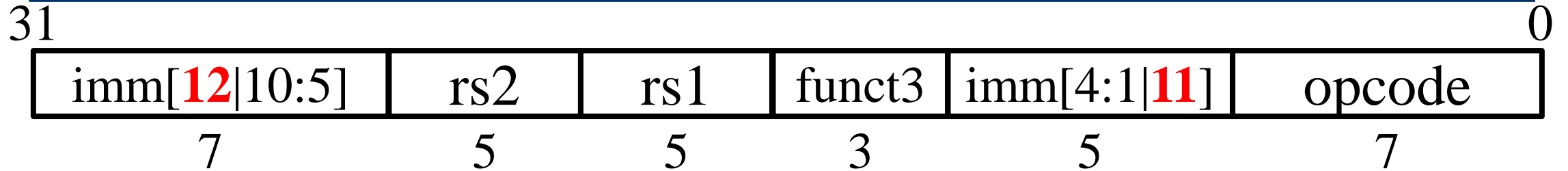


Count instructions from branch

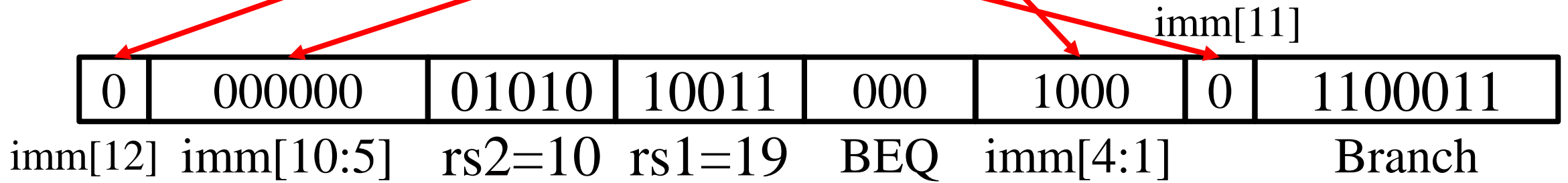
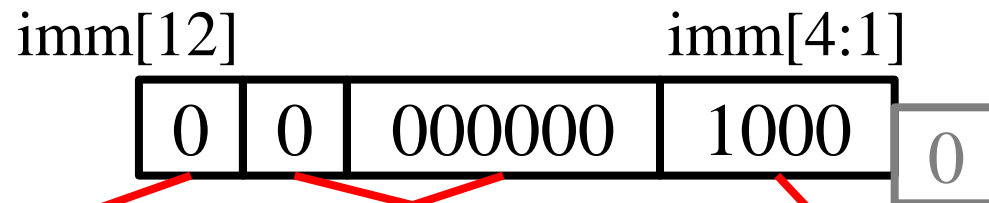
End:   # 目标指令

- $\text{offset} = 4 \times 32 \text{ bit/instruction} = 16 \text{ bytes} = 8 \times 2 \text{ bytes}$

# 例中beq x19, x10, ...的立即数表示



beq x19, x10, offset = 8 (× 2 bytes)



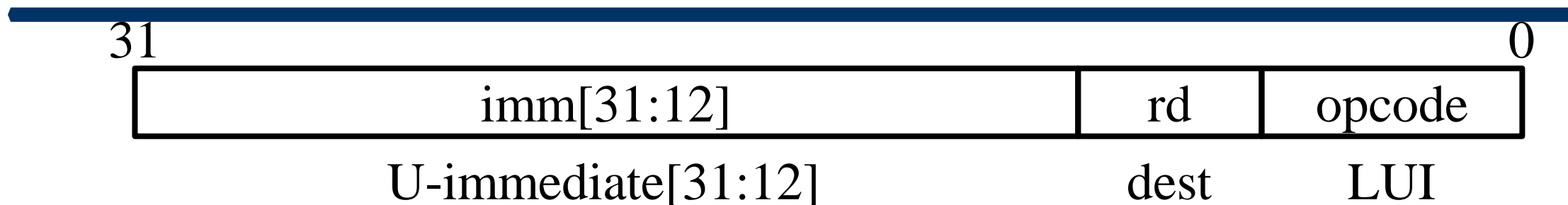
# RISC-V所有B型指令


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

$-2^{11} \sim 2^{11}-1$

# U型指令

**opi** rd, imm



- ADDI等I型指令中的立即数范围是多少？ 
- U型指令进行**长**立即数操作
  - 高20位（ imm[31:12] ）：为长度为20位的立即数字段
  - rd（ 5位 ）：目的寄存器字段
- LUI(**L**oad **U**pper **I**mmEDIATE)—将长立即数写入目的寄存器
- AUIPC(**A**dd **U**pper **I**mmEDIATE to **PC**)——将PC与长立即数相加结果写入目的寄存器

# U型指令LUI的应用

- LUI将长立即数写入目的寄存器的高20位，并清除低12位。
- LUI与ADDI一起可在寄存器中创建任何32位立即数值
  - 如给寄存器x10赋值0x87654321

LUI     x10, 0x87654     # x10 = 0x87654000

ADDI   x10, x10, 0x321    # x10 = 0x87654321

Code	Basic	Source	
0x876542b7	LUI x5, 0x00087654	2: LUI	t0, 0x87654    #t0 = 0x87654000
0x32128293	ADDI x5, x5, 0x00000321	3: ADDI	t0, t0, 0x321    # t0 = 0x87654321



# U型指令LUI的应用

- 如何创建 0xDEADBEEF?

LUI t2, 0xDEADB **B** # t2 = 0xDEADB **000**

#ADDI t2, t2, 0xEEF



#**会出错**? too large to fit in ...

ADDI t2, t2, 0xFFFFFEEF # t2 = 0xDEAD **A**EEF?

#ADDI 立即数总是进行**符号扩展**, 如果高位为1, 将从高位的20位中减去1

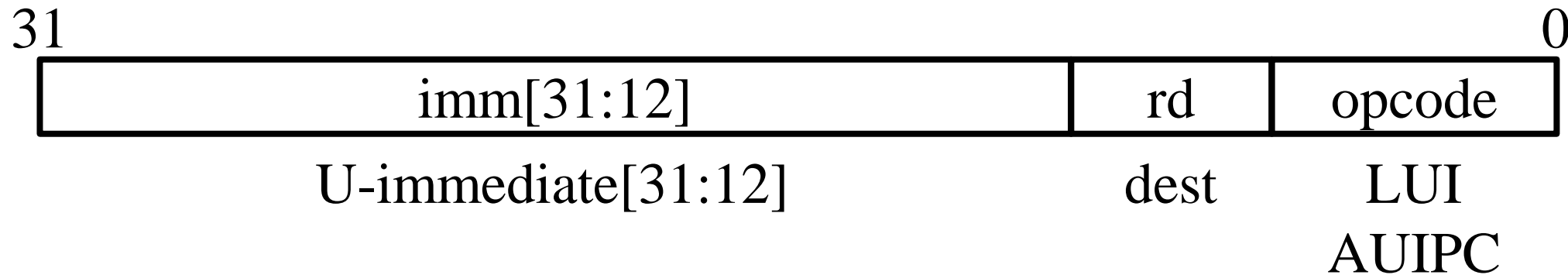
- 伪指令 li x10, 0xDEADB **B**EEF # 创建两条指令

LUI t1, 0xDEADC **C** # t1 = 0xDEADC **000**

ADDI t1, t1, 0xFFFFFEEF # t1 = 0xDEADBEEF

- **不用伪指令** 如何创建64位立即数?

# U型指令—AUIPC(Add Upper Immediate to PC)



- 用于PC相对寻址

- 将长立即数值加到PC并写入目的寄存器

Label: AUIPC rd, im #将Label地址+立即数imm存rd

# lui,auipc,jalr指令的应用

---

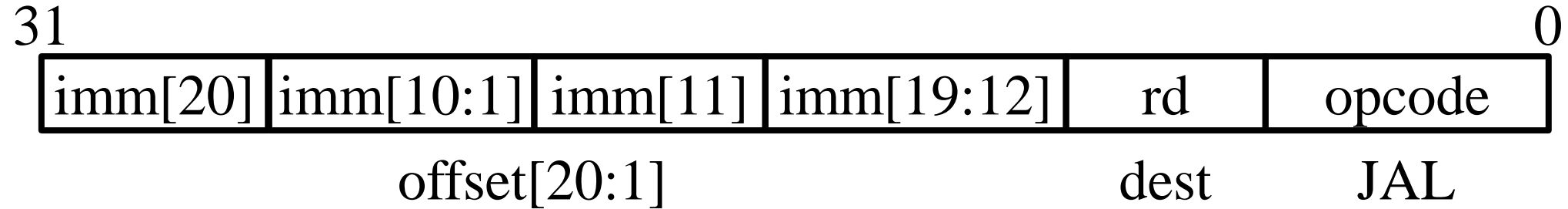
- ret 和 jr 伪指令
  - $\text{ret} = \text{jr ra} = \text{jalr x0}, 0(\text{ra})$
- 对任意32位绝对地址处的函数进行调用
  - $\text{lui Reg1}, \langle \text{hi20bits} \rangle \quad \# \langle \dots \rangle$  示意地址的高20位,%hi(label)
  - $\text{jalr rd}, \langle \text{lo12bits} \rangle(\text{Reg1}) \quad \# \langle \text{lo12bits} \rangle$  地址的低12位
- 相对PC地址32位偏移量的相对寻址
  - $\text{auipc x1}, \langle \text{hi20bits} \rangle$
  - $\text{jalr rd}, \text{x1}, \langle \text{lo12bits} \rangle$

# 练习-U型指令

- 写出产生64位常量0x1122334455667B88的RISC-V汇编代码，并将该值存储到寄存器x10中。

- lui x10, 0x11223                      #x10=0x0000 0000 1122 3000
- addi x10, x10, 0x344                #x10=0x0000 0000 1122 3344
- slli x10, x10, 32                    #x10=0x1122 3344 0000 0000
- lui x5, 0x5566**8**                    #x5 = 0x0000 0000 5566 **8**000
- addi x5, x5, 0x**FFFFFF**B88        #x5 = 0x0000 0000 5566 **7B**88
- add x10, x10, x5                    #x10=0x1122 3344 5566 7B88

# J型指令



- JAL将PC+4写入目的寄存器rd中
  - J是伪指令，等同于JAL,但设置rd = x0不保存返回地址
- $PC = PC + offset$ （PC相对寻址）
- 访问相对PC,以2字节为单位的 $\pm 2^{19}$ 范围内的地址空间
  - $\pm 2^{18}$  字指令空间，或 $\pm 2^{20}$ **字节（1MiB）**地址范围
- 立即数字段编码的优化类似于B型指令，以降低硬件成本

# J型指令

---

- j 伪指令

- j Label = jal x0, Label      # 设置目的寄存器为x0

- jal 指令

- jal ra, FuncName

# RISC-V指令格式

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
B 型	Imm[12,10:5]	rs2	rs1	funct3	imm[4:1,11]	opcode
J 型	Imm[20,10:1,11,19:12]				rd	opcode
U 型	Imm[31:12]				rd	opcode

# 第二章 RISC-V汇编及其指令系统

---

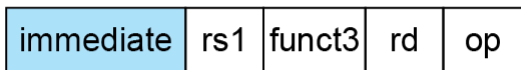
- **RISC-V指令表示**
  - RISC-V六种指令格式
  - **RISC-V寻址模式介绍**



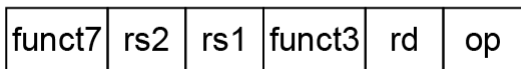


# RISC-V的寻址模式

## 1. Immediate addressing



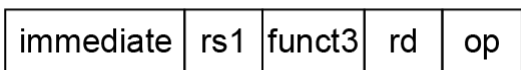
## 2. Register addressing



Registers

Register

## 3. Base addressing



Memory

Register

+

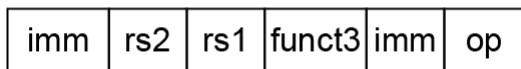
Byte

Halfword

Word

Doubleword

## 4. PC-relative addressing



Memory

PC

+

Word

# 立即数寻址

---

- 操作数是指令本身的常量
  - `addi x3, x4, 10`
  - `andi x5, x6, 3`

immediate	rs1	funct3	rd	opcode
-----------	-----	--------	----	--------

# 寄存器寻址

- 操作数在寄存器中

- add x1, x2, x3

- and x5, x6, x7

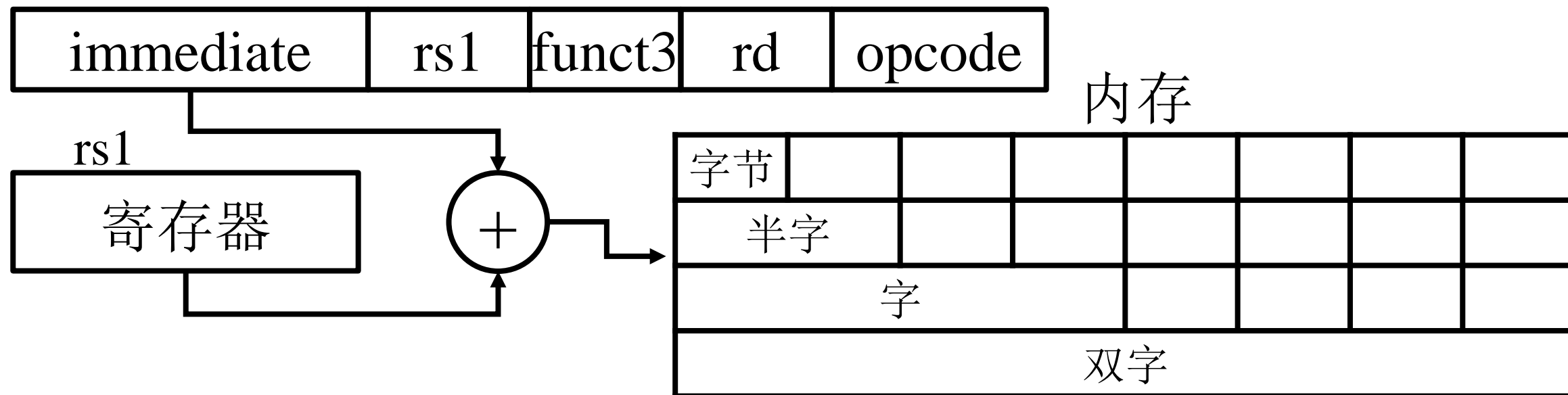


# 基址或偏移寻址

- 操作数于内存中，其地址是寄存器和指令中的常量之和

lw x10, 12(x15)

- 基址寄存器(x15) + 偏移量(12)



# PC相对寻址

- 分支地址是PC与指令中立即数之和

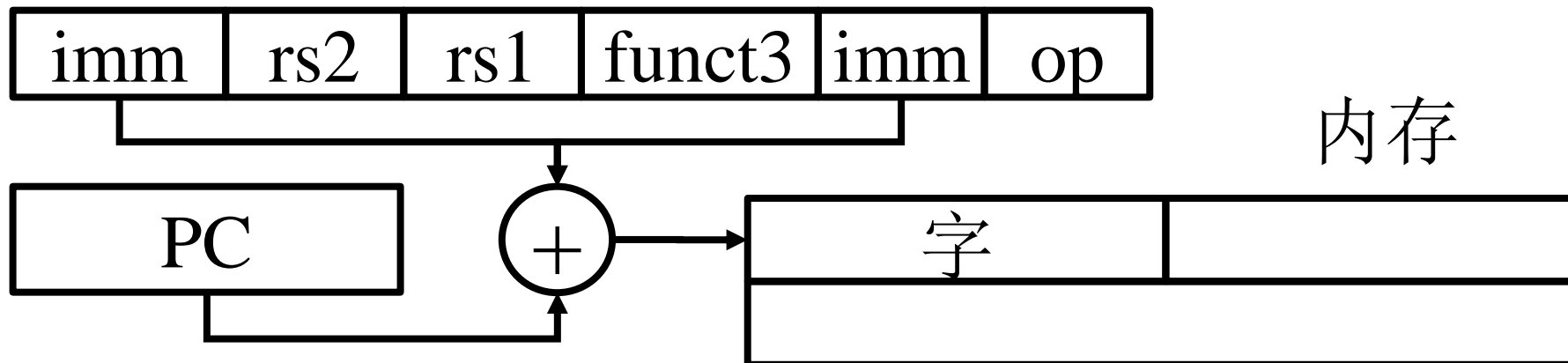
Loop: `beq x19, x10, End`

`add x18, x18, x10`

`addi x19, x19, -1`

`jal x0, Loop`

End:



在RISC V汇编语言中，op Regx, Regy, Regz中，从左到右三个寄存器依次代表：

- ☐ A 源寄存器1，源寄存器2，目的寄存器
- ☐ B 目的寄存器，源寄存器2，源寄存器1
- ☒ C 目的寄存器，源寄存器1，源寄存器2
- ☐ D 源寄存器2，源寄存器1，目的寄存器

提交

R型指令格式中，从右到左，依次为 [填空1]，[填空2]，[填空3]，[填空4]，[填空5]，[填空6]  
从右到左各个字段的长度分别为 [填空7]，[填空8]，  
[填空9]，[填空10]，[填空11]，[填空12]。

--	--	--	--	--	--

作答

# 综合练习——黑书81页例题

- 对于以下汇编代码，假设循环的开始在内存80000处，那么这个循环的RISC-V机器代码是什么？

- Loop: slli x10, x22, 3  
add x10, x10, x25  
ld x9, 0(x10)  
bne x9, x23, Exit  
addi x22, x22, 1  
beq x0, x0, Loop

地址	指令
80000	00000000 00011 10110 001 01010 0010011
800	
800	
800	
800	
800	

Exit:



# 第二章 RISC-V汇编及其指令系统

---

- 计算机中数的表示
- RISC-V概述
- RISC-V汇编语言
- RISC-V指令表示
- 案例分析



# 第二章 RISC-V汇编及其指令系统

---

- 案例分析
  - **RISC-V**冒泡排序
  - 数组与指针



# C排序的例子

- C冒泡排序函数的汇编指令示例

- **swap**过程（叶过程）

```
void swap(long long int v[], long long int k)
{
    long long int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

- v保存在x10, k保存在x11, temp保存在x5

# swap过程

- **swap:**

```
slli x6,x11,3    # x6 = k * 8
add x6,x10,x6    # x6 = v + (k * 8)
ld x5,0(x6)      # x5 (temp) = v[k]
ld x7,8(x6)      # x7 = v[k + 1]
sd x7,0(x6)      # v[k] = x7
sd x5,8(x6)      # v[k+1] = x5 (temp)
jalr x0,0(x1)    # 返回调用函数
```

- **swap过程（叶过程）**

```
void swap(long long int v[], long long int k)
{
    long long int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

- v保存在x10，k保存在x11，temp保存在x5

# C版本的sort过程

- 非叶过程（调用了swap）

```
void sort (long long int v[], long long int n) {  
    long long int i, j;  
    for (i = 0; i < n; i += 1) {  
        for (j = i - 1;  
            j >= 0 && v[j] > v[j + 1];  
            j -= 1) {  
            swap(v,j);  
        }  
    }  
}
```

- v保存在x10, n保存在x11, i保存在x19, j保存在x20

# 外层循环

- 外层循环的框架:

for (i = 0; i < n; i += 1) {

v->x10, n->x11, i->x19, j->x20

- 初始化: li x19, 0 # i = 0
- 循环判断: for1tst:  
bge x19, x11, exit1
- 循环体
- 循环增值:  
addi x19, x19, 1 # i += 1  
j for1tst # 跳转到外层循环

```
li x19,0      # i = 0
for1tst:
    bge x19, x11, exit1  # 如果x19 ≥ x11 (i ≥ n), 跳转到exit1
    ....                # 外层循环的函数体
    addi x19, x19, 1     # i += 1
    j for1tst           # 跳转到外层循环
exit1:
```

# 内层循环

- 内层循环的框架: `for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {`

`v->x10, n->x11, i->x19, j->x20`

- 初始化: `addi x20, x19, -1` #  $j = i - 1$
- 循环for2tst:判断有两部分:  $j \geq 0 \ \&\& \ v[j] > v[j + 1]$ 
  - 第一个判断 $j < 0$ : `blt x20, x0, exit2`
  - 第二个判断 $v[j] \leq v[j + 1]$ , 首先加载 $v[j]$ 和 $v[j + 1]$ 
    - `slli x5, x20, 3` #  $x5 = j * 8$
    - `add x5, x10, x5` #  $x5 = v + (j * 8)$
    - `ld x6, 0(x5)` #  $x6 = v[j]$
    - `ld x7, 8(x5)` #  $x7 = v[j + 1]$
  - 然后判断: `ble x6, x7, exit2`
- 都不成立, 则执行循环体, 即swap
- 自减循环: `addi x20, x20, -1`  
`j for2tst`

```
addi x20, x19, -1    # j = i - 1
```

```
for2tst:
```

```
blt x20, x0, exit2   # 如果x20 < 0 (j < 0), 跳转到exit2
```

```
slli x5, x20, 3      # x5 = j * 8
```

```
add x5, x10, x5      # x5 = v + (j * 8)
```

```
ld x6, 0(x5)         # x6 = v[j]
```

```
ld x7, 8(x5)         # x7 = v[j + 1]
```

```
ble x6, x7, exit2    # 如果x6 ≤ x7, 跳转到exit2
```

```
...                  # 第二个循环体
```

```
addi x20, x20, -1    # j -= 1
```

```
j for2tst            # 跳转到内层循环的判断语句
```

```
exit2:
```

# 第二个for循环的循环体swap

- sort函数:  $v \rightarrow x10$ ,  $n \rightarrow x11$ ,  $i \rightarrow x19$ ,  $j \rightarrow x20$

- swap函数:  $v \rightarrow x10$ ,  $k \rightarrow x11$ , temp保存在x5

问题: sort过程用到x10和x11, 但 swap用 x10和x11来传参

- sort过程的x10和x11用其他寄存器替代

- 较早的地方将x10和x11的值复制出来:

- $mv\ x21,\ x10$       # 将参数x10 (v的基址) 复制到x21

- $mv\ x22,\ x11$       # 将参数x11 (n) 复制到x22

- 调用swap时用x10, x11传参

- $mv\ x10,\ x21$       # swap的第一个参数是v

- $mv\ x11,\ x20$       # swap的第二个参数是j

- $jal\ x1,\ swap$       # 调用swap

- 需保存的寄存器:

- $x1,\ x22,\ x21,\ x20,\ x19$



# 保存寄存器值-入栈

- 需保存的寄存器:

x1, x22, x21, x20, x19

寄存器	助记符	注解
x0	zero	固定值为0
x1	ra	返回地址(Return Address)
x2	sp	栈指针(Stack Pointer)
x3	gp	全局指针(Global Pointer)
x4	tp	线程指针(Thread Pointer)
x5-x7	t0-t2	临时寄存器
x8	s0/fp	save寄存器/帧指针(Frame Pointer)
x9	s1	save寄存器
x10-x11	a0-a1	函数参数 / 函数返回值
x12-x17	a2-a7	函数参数
x18-x27	s2-s11	save寄存器
x28-x31	t3-6	临时寄存器

- 执行循环体之前

```
addi sp,sp,-40 # 在栈中留出5个寄存器  
                (双字) 的空间
```

```
sd x1,32(sp) # x1入栈
```

```
sd x22,24(sp) #sort中n->x22(原先的n-  
                >x11 作为了swap参数)
```

```
sd x21,16(sp) #sort中v[] ->x21
```

```
sd x20,8(sp) #sort中j->x20
```

```
sd x19,0(sp) # sort中i->x19
```

# 恢复寄存器值-出栈

- 恢复的寄存器:

x1, x22, x21, x20, x19

- 执行完循环体返回之前

exit1:

ld x19, 0(sp) # 恢复x19 (出栈)

ld x20, 8(sp) # 恢复x20 (出栈)

ld x21, 16(sp) # 恢复x21 (出栈)

ld x22, 24(sp) # 恢复x22 (出栈)

ld x1, 32(sp) # 恢复x1 (出栈)

addi sp, sp, 40 # 恢复栈指针

最后返回主程序: jalr x0, 0(x1) # 返回调用线程

## 保存寄存器（入栈）

sort:

```
addi sp, sp, -40    # 在栈中留出5个寄存器（双字）的空间
sd x1, 32(sp)       # 保存x1的值（入栈）
sd x22, 24(sp)      # 保存x22的值（入栈）
sd x21, 16(sp)      # 保存x21的值（入栈）
sd x20, 8(sp)       # 保存x20的值（入栈）
sd x19, 0(sp)       # 保存x19的值（入栈）
```

## 过程体

移动参数

```
mv x21, x10          # 复制x10中的值到x21
mv x22, x11          # 复制x11中的值到x22
```

外循环

```
li x19, 0             # i = 0
for1tst:
    bge x19, x22, exit1 # 如果x19 ≥ x22 (i ≥ n), 跳转到exit1
    addi x20, x19, -1   # j = i-1
```

内循环

```
for2tst:
    blt x20, x0, exit2  # 如果x20 < 0 (j < 0), 跳转到exit2
    slli x5, x20, 3      # x5 = j * 8
    add x5, x21, x5      # x5 = v + (j * 8)
    ld x6, 0(x5)         # x6 = v[j]
    ld x7, 8(x5)         # x7 = v[j + 1]
    ble x6, x7, exit2    # 如果x6 ≤ x7, 跳转到exit2
```

参数传递  
和调用

```
mv x10, x21          # swap的第一个参数是v
mv x11, x20          # swap的第二个参数是j
jal x1, swap          # 调用swap
```

内循环

```
addi x20, x20, -1    # j -= 1
j for2tst             # 跳转到内层循环for2tst
```

v->x10, n->x11, i->x19, j->x20

## 外循环

exit2:

```
addi x19, x19, 1      # i += 1
j for1tst              # 跳转到外层循环的判断语句
```

## 恢复寄存器

exit1:

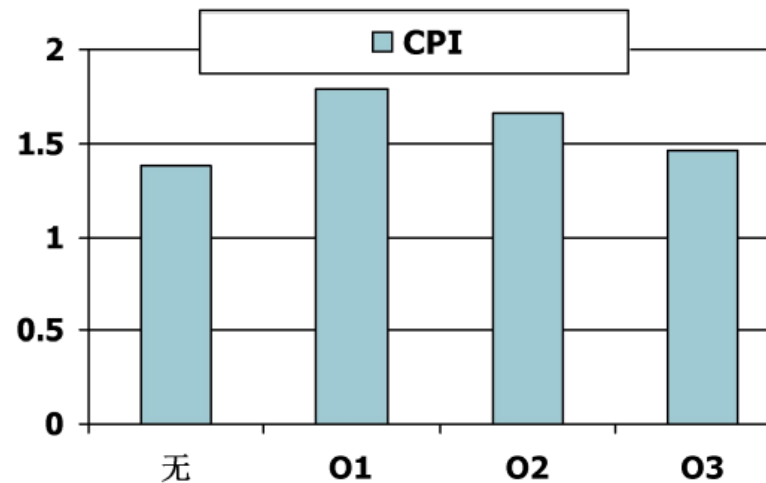
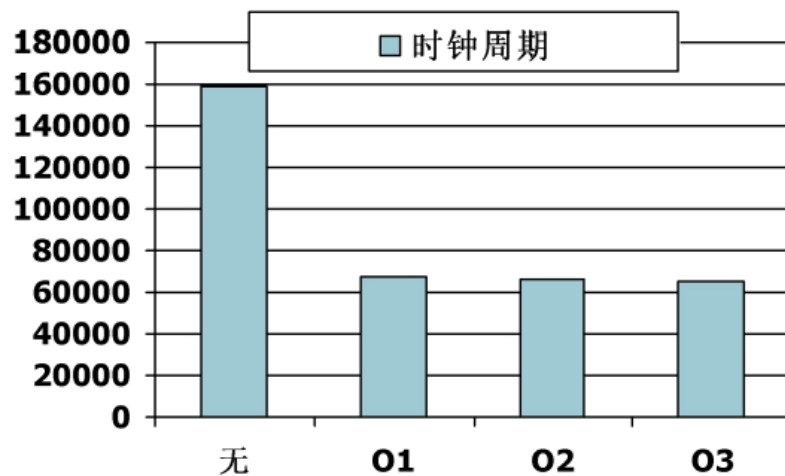
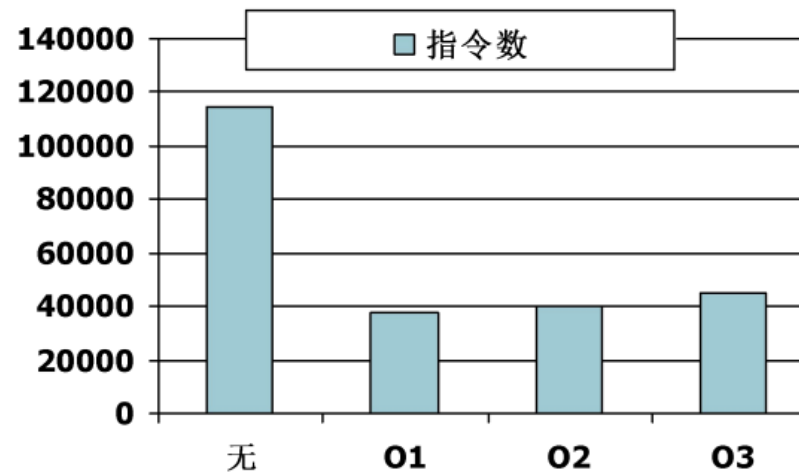
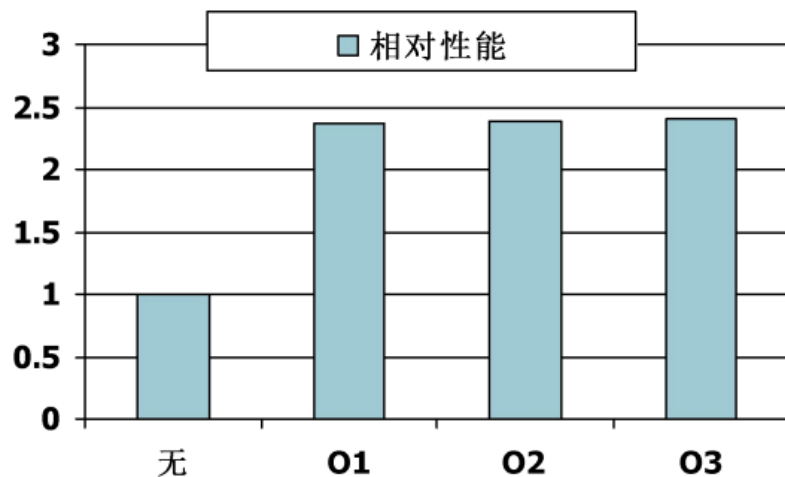
```
ld x19, 0(sp)         # 恢复x19（出栈）
ld x20, 8(sp)         # 恢复x20（出栈）
ld x21, 16(sp)        # 恢复x21（出栈）
ld x22, 24(sp)        # 恢复x22（出栈）
ld x1, 32(sp)         # 恢复x1（出栈）
addi sp, sp, 40       # 恢复栈指针
```

## 过程返回

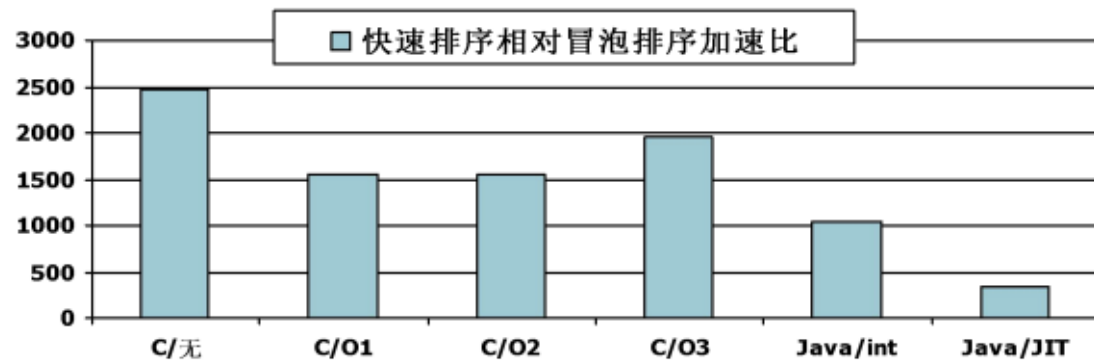
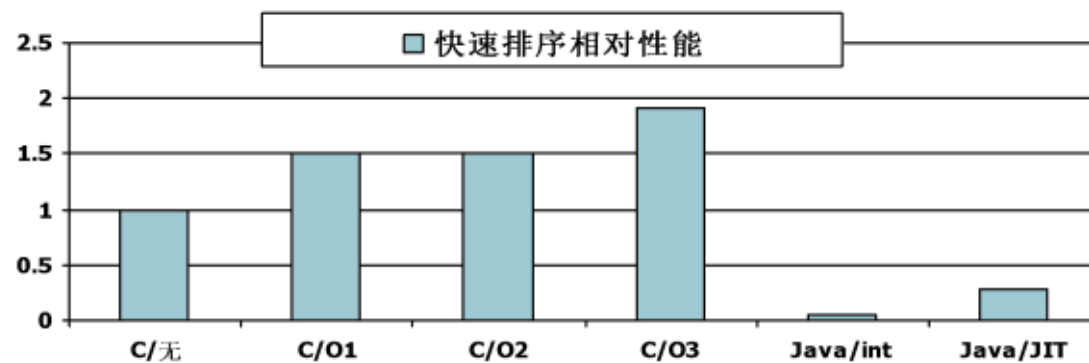
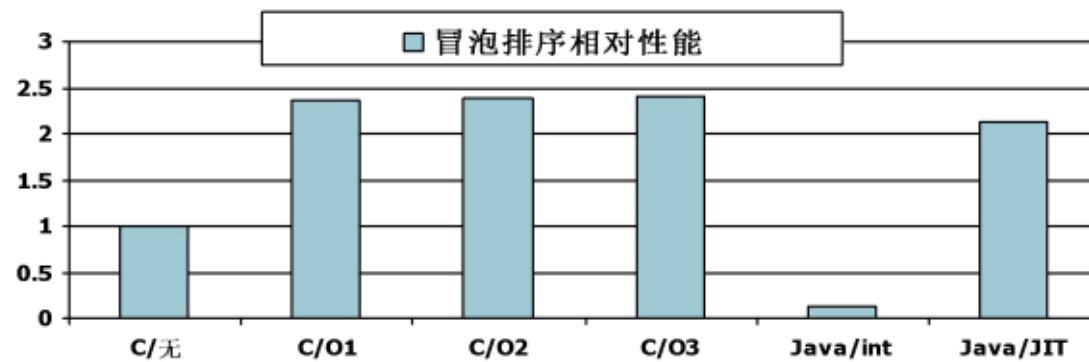
```
jalr x0, 0(x1)        # 返回调用线程
```

# 编译器优化的影响

用gcc for Pentium 4在Linux下编译



# 编程语言和算法的影响



# 总结

---

- 单看指令数或CPI都不是好的性能指标
- 编译器优化对算法敏感
- Java即时编译器生成的代码明显快于用JVM解释的代码
  - 有些情况下可以和优化过的C代码相媲美
- 没有什么可以弥补拙劣的算法！

# 第二章 RISC-V汇编及其指令系统

---

- 案例分析
  - RISC-V冒泡排序
  - 数组与指针



# 数组与指针

---

- 数组索引包括
  - 用元素长度乘以下标
  - 与数组基址相加
- 指针直接对应存储器地址
  - 可以避免索引的复杂性



# 例：将数组清零

- 使用数组下标清零

```
clear1 (long long int array[], int size) {  
    int i;  
    for (i = 0; i < size; i += 1)  
        array[i]=0  
}
```

- 使用指针清零

```
clear2 (long long int *array, int size) {  
    long long int *p;  
    for (p = &array[0]; p < &array[size]; p=p+1)  
        *p=0  
}
```

# 例：使用数组下标清零

- 内层循环的框架：

**array[]**->**x10**, **size**->**x11**, **i**->**x5**

- 初始化： `li x5, 0`    `# i = 0`

- 获取array[i]的字节地址：

`loop1: slli, x6, x5, 3`

`add x7, x10, x6`

- 将0存到该地址： `sd x0, 0(x7)`

- 循环增值：

`addi x5, x5, 1`    `# i += 1`

- 循环测试i是否小于size：

`blt x5, x11, loop1`

```
clear1 (long long int array[], int size) {  
    int i;  
    for (i = 0; i < size; i += 1)  
        array[i]=0  
}
```

```
li x5, 0    # i = 0  
loop1:  
    slli, x6, x5, 3  
    add x7, x10, x6  
    sd x0, 0(x7)  
    addi x5, x5, 1    # i += 1  
    blt x5, x11, loop1
```

# 例：使用指针清零

- 内层循环的框架：

**array[]**->**x10**, **size**->**x11**, **p**->**x5**

- 初始化，如指针p赋值为数组第一个元素地址：

mv **x5**, x10    # p 等于array[0]地址

slli x6, x11, 3    # 最大的地址偏移量

add **x7**, x10, x6    # array[size]的地址

- For循环体，循环赋值0：

loop2: sd, x0, 0(x5)

- 修改p指向下一个双字：

addi x5, x5, 8

- 循环判断：

bltu x5, x7, loop2

```
clear2 (long long int *array, int size) {  
    long long int *p;  
    for (p = &array[0]; p < &array[size]; p=p+1)  
        *p=0  
}
```

```
mv x5, x10  
slli x6, x11, 3  
add x7, x10, x6  
loop2:  
    sd, x0, 0(x5)  
    addi x5, x5, 8  
    bltu x5, x7, loop2
```

# 例：将数组清零

<pre>clear1(long long int array[], int size) {     int i;     for (i = 0; i &lt; size; i += 1)         array[i] = 0; }</pre>	<pre>clear2(long long int *array, int size) {     long long int *p;     for (p = &amp;array[0]; p &lt; &amp;array[size];         p = p + 1)         *p = 0; }</pre>
<pre>li    x5,0          # i = 0 loop1: slli  x6,x5,3        # x6 = i * 8 add   x7,x10,x6      # x7 = array[i]的地址 sd    x0,0(x7)       # array[i] = 0 addi  x5,x5,1        # i = i + 1 blt   x5,x11,loop1   # if (i&lt;size)                         # go to loop1</pre>	<pre>mv x5,x10           # p = array[0]的地址 slli x6,x11,3       # x6 = size * 8 add x7,x10,x6       # x7= array[size]的地址 loop2: sd x0,0(x5)         # Memory[p] = 0 addi x5,x5,8        # p = p + 8 bltu x5,x7,loop2    # if (p&lt;&amp;array[size])                         # go to loop2</pre>

# 数组与指针的比较

- 乘8——>左移3次（编译器优化——强度减弱）
- 指针版本的循环体指令数少于数组版本
  - 数组版本由于i的递增，需要重新计算下一个数组元素的地址
- 编程推荐使用下标方式访问数组元素

<pre>li    x5,0          # i = 0 loop1: slli  x6,x5,3        # x6 = i * 8 add   x7,x10,x6      # x7 = array[i]的地址 sd    x0,0(x7)       # array[i] = 0 addi  x5,x5,1        # i = i + 1 blt   x5,x11,loop1   # if (i&lt;size)                         # go to loop1</pre>	<pre>mv    x5,x10         # p = array[0]的地址 slli  x6,x11,3        # x6 = size * 8 add   x7,x10,x6      # x7= array[size]的地址 loop2: sd    x0,0(x5)       # Memory[p] = 0 addi  x5,x5,8        # p = p + 8 bltu  x5,x7,loop2     # if (p&lt;&amp;array[size])                         # go to loop2</pre>
--	---