

循序渐进，学习开发一个 RISC-V 上的操作系统



第 5 章 RISC-V 汇编语言编程

汪辰

- RISC-V 汇编语言入门
- RISC-V 汇编指令总览
- RISC-V 汇编指令进阶
- RISC-V 汇编函数调用约定
- RISC-V 汇编与 C 混合编程

- 【参考 1】： The RISC-V Instruction Set Manual , Volume I: Unprivileged ISA , Document Version 20191213

- **RISC-V 汇编语言入门**
 - 汇编语言概念简介
 - 汇编语言语法介绍
 - 一个典型的例子
- **RISC-V 汇编指令总览**
- **RISC-V 汇编指令进阶**
- **RISC-V 汇编函数调用约定**
- **RISC-V 汇编与 C 混合编程**

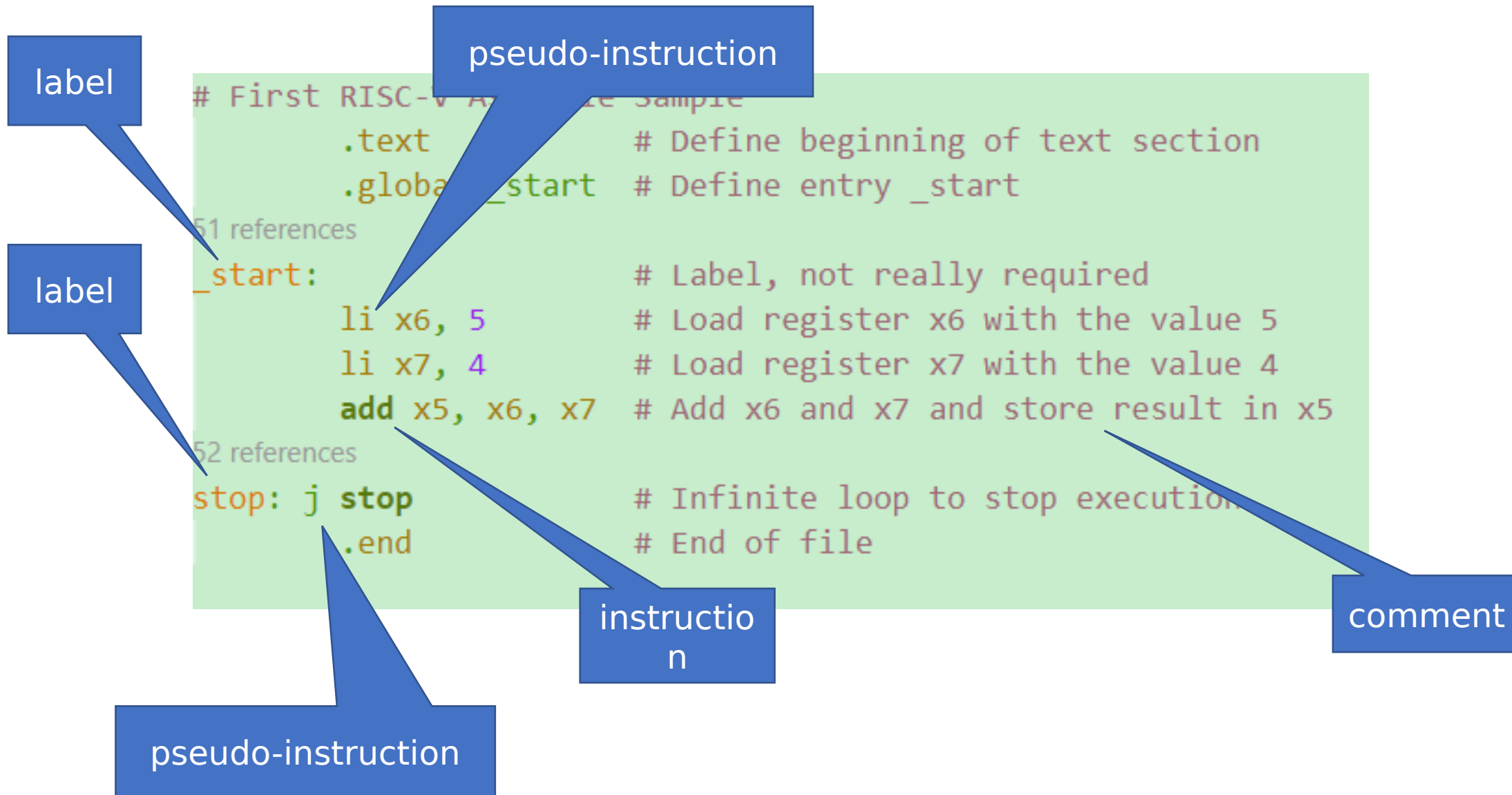
- 汇编语言（Assembly Language）是一种“低级”语言。
- 汇编语言的缺点：
 - 难读
 - 难写
 - 难移植
- 汇编语言的优点
 - 灵活
 - 强大
- 汇编语言的应用场景
 - 需要直接访问底层硬件的地方
 - 需要对性能执行极致优化的地方

- 一个完整的 RISC-V 汇编程序有多条语句组成。
- 一条典型的 RISC-V 汇编语句由 3 部分组成：

`[label:] [operation] [comment]`

- **label:** 标号，GNU 汇编中，任何以冒号结尾的标识符都被认为是一个标号，而不一定非要在一行的开始。
- **operation** 可以有以下几种类型：
 - ✓ **instruction:** 指令，直接对应二进制机器指令的字符串
 - ✓ **pseudo-instruction:** 伪指令，为了提高编写代码的效率，可以用一条伪指令指示汇编器产生多条实际的指令 (instructions)。
 - ✓ **directive:** 伪操作，通过类似指令的形式（以“.”开头），通知汇编器如何控制代码的产生等，不对应具体的指令。
 - ✓ **macro:** 用户自定义的宏
- **comment:** 注释，常用方式：以“#”开始到当前

一个典型的例子



- RISC-V 汇编语言入门
- **RISC-V 汇编指令总览**
 - RISC-V 汇编指令操作对象
 - RISC-V 汇编指令编码格式
 - RISC-V 汇编指令分类
 - RISC-V 汇编伪指令一览
- RISC-V 汇编指令进阶
- RISC-V 汇编函数调用约定
- RISC-V 汇编与 C 混合编程

➤ 寄存器：

- 注意：本章节课程涉及 RV32I 的通用寄存器组
- 32 个通用寄存器，x0 ~ x31
- 在 RISC-V 中，只对寄存器中的数据执行算术运算。

➤ 内存：

- 使用字节（Byte）为基本单位寻址
- RV32 可以访问最多 2^{32} 个字节的内存空间。

| | | | | | | | | | | | | | | |
|-----------------------|----|----|----|-----|----|-----|----|--------|----|-------------|---|--------|---|--------|
| 31 | 27 | 26 | 25 | 24 | 20 | 19 | 15 | 14 | 12 | 11 | 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 10:5] | | | | rs2 | | rs1 | | funct3 | | imm[4:1 11] | | opcode | | B-type |
| imm[31:12] | | | | | | | | | | rd | | opcode | | U-type |
| imm[20 10:1 11 19:12] | | | | | | | | | | rd | | opcode | | J-type |

- 指令长度： 1 word = 4 bytes = 32 bits
- 32 个 bit 划分成不同的 “域（ field ）”
- 6 种指令格式（ format ）

| | | | | | | | | | | | | | | |
|-----------------------|----|----|----|-----|----|-----|----|--------|----|-------------|---|--------|---|--------|
| 31 | 27 | 26 | 25 | 24 | 20 | 19 | 15 | 14 | 12 | 11 | 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 10:5] | | | | rs2 | | rs1 | | funct3 | | imm[4:1 11] | | opcode | | B-type |
| imm[31:12] | | | | | | | | | | rd | | opcode | | U-type |
| imm[20 10:1 11 19:12] | | | | | | | | | | rd | | opcode | | J-type |

➤ 6 种指令格式（format）

- **R-type:**（Register），每条指令中有三个 fields，用于指定 3 个 寄存器参数
- **I-type:**（Immediate），每条指令除了带有两个寄存器参数外，还带有一个立即数参数（宽度为 12 bits）。
- **S-type:**（Store），每条指令除了带有两个寄存器参数外，还带有一个立即数参数（宽度为 12 bits，但 fields 的组织方式不同于 I-type）
- **B-type:**（Branch），每条指令除了带有两个寄存器参数外，还带有一个立即数参数（宽度为 12 bits，但取值为 2 的倍数）。
- **U-type:**（Upper），每条指令含有一个寄存器参数再加上一个立即数参数（宽度为 20 bits，用于表示一个立即数的高 20 位）
- **J-type:**（Jump），每条指令含有一个寄存器参数再加上一个立即数参数（宽度为 20 bits）

RISC-V 汇编指令分类



算术运算指令

| Inst | Name | FMT | Opcode | funct3 | funct7 | Description (C) |
|-------|-------------------------|-----|---------|--------|----------------|------------------------------|
| add | ADD | R | 0110011 | 0x0 | 0x00 | rd = rs1 + rs2 |
| sub | SUB | R | 0110011 | 0x0 | 0x20 | rd = rs1 - rs2 |
| addi | ADD Immediate | I | 0010011 | 0x0 | | rd = rs1 + imm |
| lui | Load Upper Imm | U | 0110111 | | | rd = imm << 12 |
| auipc | Add Upper Imm to PC | U | 0010111 | | | rd = PC + (imm << 12) |
| xor | XOR | R | 0110011 | 0x4 | 0x00 | rd = rs1 ^ rs2 |
| or | OR | R | 0110011 | 0x6 | 0x00 | rd = rs1 rs2 |
| and | AND | R | 0110011 | 0x7 | 0x00 | rd = rs1 & rs2 |
| xori | XOR Immediate | I | 0010011 | 0x4 | | rd = rs1 ^ imm |
| ori | OR Immediate | I | 0010011 | 0x6 | | rd = rs1 imm |
| andi | AND Immediate | I | 0010011 | 0x7 | | rd = rs1 & imm |
| sll | Shift Left Logical | R | 0110011 | 0x1 | 0x00 | rd = rs1 << rs2 |
| srl | Shift Right Logical | R | 0110011 | 0x5 | 0x00 | rd = rs1 >> rs2 |
| sra | Shift Right Arith* | R | 0110011 | 0x5 | 0x20 | rd = rs1 >> rs2 |
| slt | Set Less Than | R | 0110011 | 0x2 | 0x00 | rd = (rs1 < rs2)?1:0 |
| sltu | Set Less Than (U) | R | 0110011 | 0x3 | 0x00 | rd = (rs1 < rs2)?1:0 |
| slli | Shift Left Logical Imm | I | 0010011 | 0x1 | imm[5:11]=0x00 | rd = rs1 << imm[0:4] |
| srlr | Shift Right Logical Imm | I | 0010011 | 0x5 | imm[5:11]=0x00 | rd = rs1 >> imm[0:4] |
| srai | Shift Right Arith Imm | I | 0010011 | 0x5 | imm[5:11]=0x20 | rd = rs1 >> imm[0:4] |
| slti | Set Less Than Imm | I | 0010011 | 0x2 | | rd = (rs1 < imm)?1:0 |
| sltiu | Set Less Than Imm (U) | I | 0010011 | 0x3 | | rd = (rs1 < imm)?1:0 |
| lb | Load Byte | I | 0000011 | 0x0 | | rd = M[rs1+imm][0:7] |
| lh | Load Half | I | 0000011 | 0x1 | | rd = M[rs1+imm][0:15] |
| lw | Load Word | I | 0000011 | 0x2 | | rd = M[rs1+imm][0:31] |
| lbu | Load Byte (U) | I | 0000011 | 0x4 | | rd = M[rs1+imm][0:7] |
| lhu | Load Half (U) | I | 0000011 | 0x5 | | rd = M[rs1+imm][0:15] |
| sb | Store Byte | S | 0100011 | 0x0 | | M[rs1+imm][0:7] = rs2[0:7] |
| sh | Store Half | S | 0100011 | 0x1 | | M[rs1+imm][0:15] = rs2[0:15] |
| sw | Store Word | S | 0100011 | 0x2 | | M[rs1+imm][0:31] = rs2[0:31] |
| beq | Branch == | B | 1100011 | 0x0 | | if(rs1 == rs2) PC += imm |
| bne | Branch != | B | 1100011 | 0x1 | | if(rs1 != rs2) PC += imm |
| blt | Branch < | B | 1100011 | 0x4 | | if(rs1 < rs2) PC += imm |
| bge | Branch ≤ | B | 1100011 | 0x5 | | if(rs1 ≥ rs2) PC += imm |
| bltu | Branch < (U) | B | 1100011 | 0x6 | | if(rs1 < rs2) PC += imm |
| bgeu | Branch ≥ (U) | B | 1100011 | 0x7 | | if(rs1 ≥ rs2) PC += imm |
| jal | Jump And Link | J | 1101111 | | | rd = PC+4; PC += imm |
| jalr | Jump And Link Reg | I | 1100111 | 0x0 | | rd = PC+4; PC = rs1 + imm |

逻辑运算指令

移位运算指令

内存读写指令

分支与跳转指令

注：部分指令因篇幅原因未列出，譬如 Compare/Synch/ Change Level 指令等。具体请参考 RISC-V 规范手册

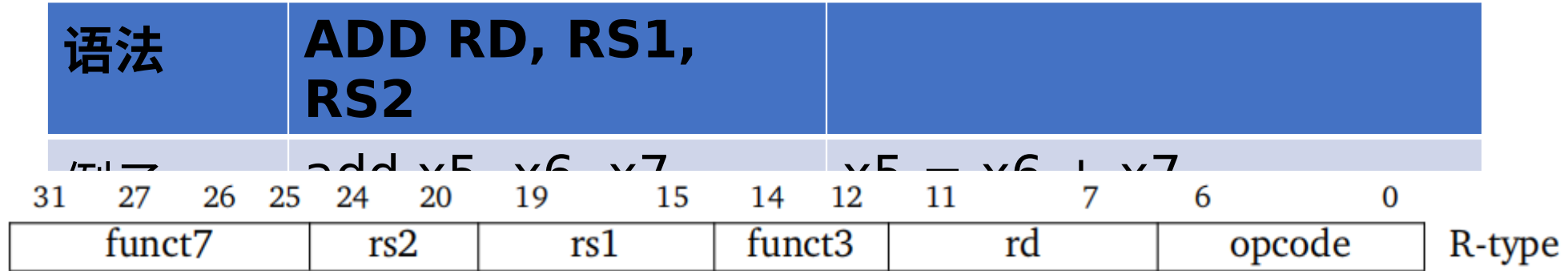
RISC-V 汇编伪指令一览

| Pseudoinstruction | Base Instruction(s) | Meaning |
|---------------------------|---|---------------------------------|
| la rd, symbol | auipc rd, symbol[31:12] addi rd, rd, symbol[11:0] | Load address |
| l{b h w d} rd, symbol | auipc rd, symbol[31:12] l{b h w d} rd, symbol[11:0] (rd) | Load global |
| s{b h w d} rd, symbol, rt | auipc rt, symbol[31:12] s{b h w d} rd, symbol[11:0] (rt) | Store global |
| fl{w d} rd, symbol, rt | auipc rt, symbol[31:12] fl{w d} rd, symbol[11:0] (rt) | Floating-point load global |
| fs{w d} rd, symbol, rt | auipc rt, symbol[31:12] fs{w d} rd, symbol[11:0] (rt) | Floating-point store global |
| nop | addi x0, x0, 0 | No operation |
| li rd, immediate | <i>Myriad sequences</i> | Load immediate |
| mv rd, rs | addi rd, rs, 0 | Copy register |
| not rd, rs | xori rd, rs, -1 | One's complement |
| neg rd, rs | sub rd, x0, rs | Two's complement |
| negw rd, rs | subw rd, x0, rs | Two's complement word |
| sext.w rd, rs | addiw rd, rs, 0 | Sign extend word |
| seqz rd, rs | sltiu rd, rs, 1 | Set if = zero |
| snez rd, rs | sltu rd, x0, rs | Set if \neq zero |
| sltz rd, rs | slt rd, rs, x0 | Set if < zero |
| sgtz rd, rs | slt rd, x0, rs | Set if > zero |
| fmv.s rd, rs | fsgnj.s rd, rs, rs | Copy single-precision register |
| fabs.s rd, rs | fsgnjx.s rd, rs, rs | Single-precision absolute value |
| fneg.s rd, rs | fsgnjn.s rd, rs, rs | Single-precision negate |
| fmv.d rd, rs | fsgnj.d rd, rs, rs | Copy double-precision register |
| fabs.d rd, rs | fsgnjx.d rd, rs, rs | Double-precision absolute value |
| fneg.d rd, rs | fsgnjn.d rd, rs, rs | Double-precision negate |
| beqz rs, offset | beq rs, x0, offset | Branch if = zero |
| bnez rs, offset | bne rs, x0, offset | Branch if \neq zero |
| blez rs, offset | bge x0, rs, offset | Branch if \leq zero |
| bgez rs, offset | bge rs, x0, offset | Branch if \geq zero |
| bltz rs, offset | blt rs, x0, offset | Branch if < zero |
| bgtz rs, offset | blt x0, rs, offset | Branch if > zero |
| bgt rs, rt, offset | blt rt, rs, offset | Branch if > |
| ble rs, rt, offset | bge rt, rs, offset | Branch if \leq |
| bgtu rs, rt, offset | bltu rt, rs, offset | Branch if >, unsigned |
| bleu rs, rt, offset | bgeu rt, rs, offset | Branch if \leq , unsigned |
| j offset | jal x0, offset | Jump |
| jal offset | jal x1, offset | Jump and link |
| jr rs | jalr x0, rs, 0 | Jump register |
| jalr rs | jalr x1, rs, 0 | Jump and link register |
| ret | jalr x0, x1, 0 | Return from subroutine |
| call offset | auipc x1, offset[31:12] jalr x1, x1, offset[11:0] | Call far-away subroutine |
| tail offset | auipc x6, offset[31:12] jalr x0, x6, offset[11:0] | Tail call far-away subroutine |
| fence | fence iorw, iorw | Fence on all memory and I/O |

- RISC-V 汇编语言入门
- RISC-V 汇编指令总览
- **RISC-V 汇编指令详解**
 - 算术运算指令
 - 逻辑运算指令
 - 移位运算指令
 - 内存读写指令
 - 条件分支指令
 - 无条件跳转指令
 - RISC-V 指令寻址模式总结
- RISC-V 汇编函数调用约定
- RISC-V 汇编与 C 混合编程

算术运算指令 (Arithmetic Instructions)

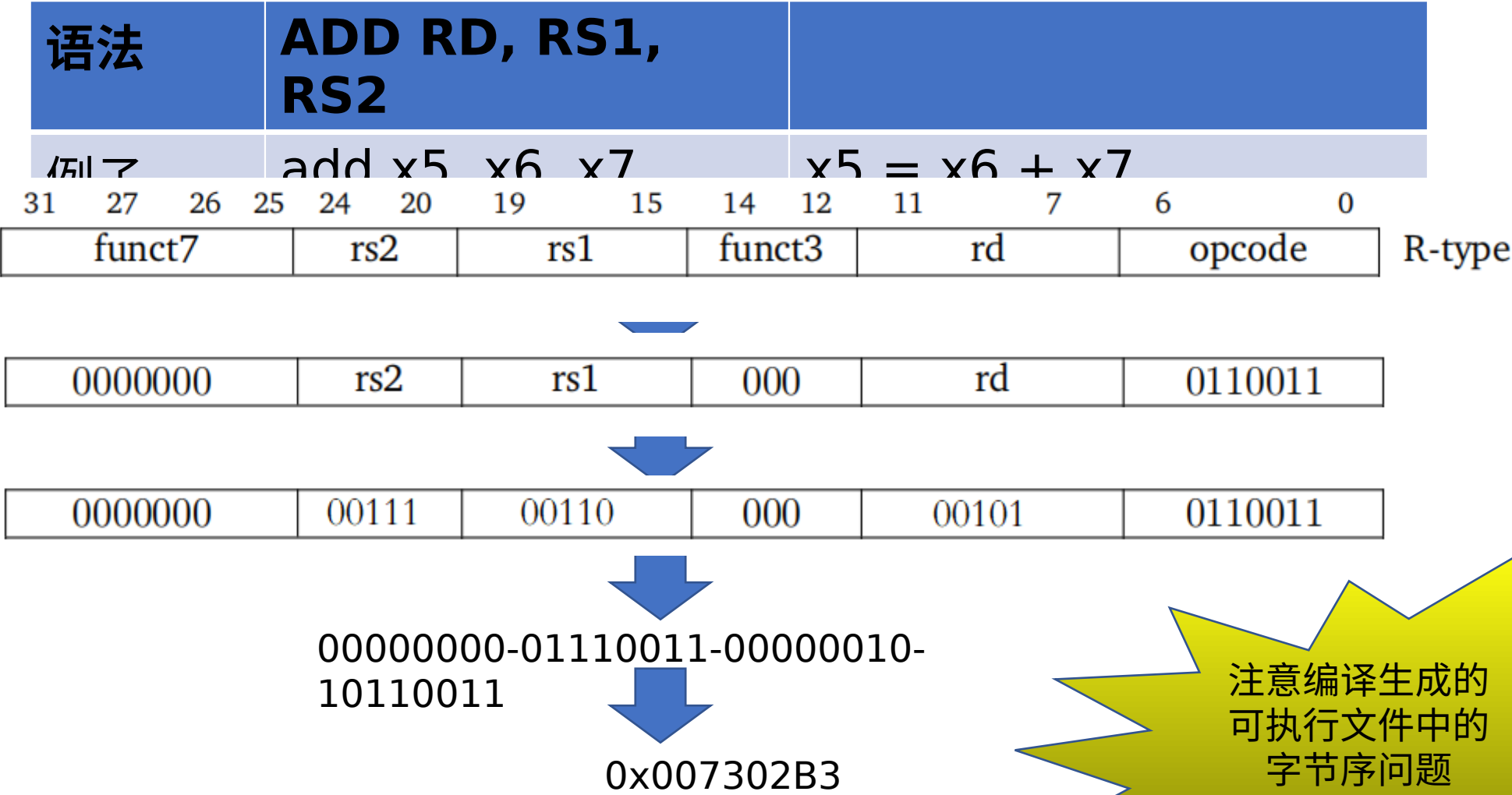
➤ ADD



- **opcode (7):** R-types 的 opcode 是 **0b0110011** ,
- **funct7+funct3 (10):** 和 opcode 一起决定最终的指令类型
- **rs1 (5):** 第一个 operand (“source register 1”)
- **rs2 (5):** 第二个 operand (“source register 2”)
- **rd (5):** “destination register” 用于存放

算术运算指令 (Arithmetic Instructions)

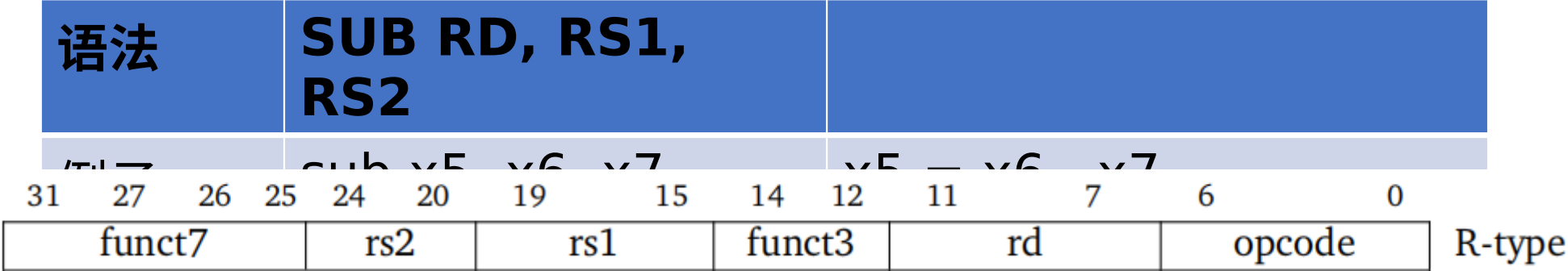
➤ ADD



注意编译生成的可执行文件中的字节序问题

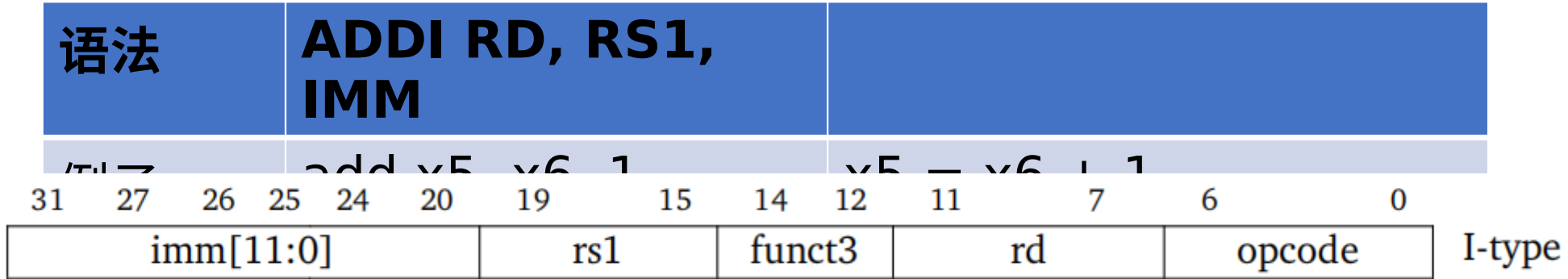
算术运算指令 (Arithmetic Instructions)

➤ SUB



算术运算指令 (Arithmetic Instructions)

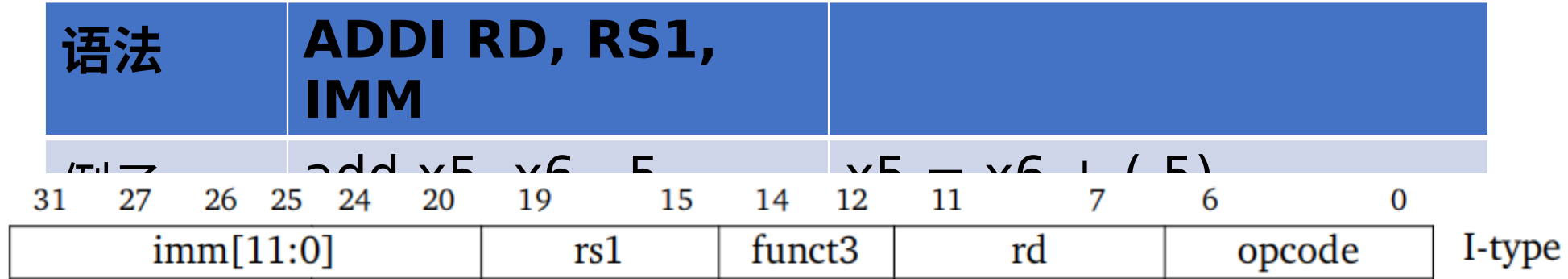
➤ ADDI



- **opcode (7):** I-types 的 opcode 是 **0b0010011** ,
- **funct3 (3):** 和 opcode 一起决定最终的指令类型
- **rs1 (5):** 第一个 operand (“source register 1”)
- **rd (5):** “destination register” 用于存放计算的结果，这里是求和的结果。
- **imm (12):** “immediate” 立即数。代替了

算术运算指令 (Arithmetic Instructions)

➤ ADDI (ADD Immediate)



- **imm (12): “immediate”, 立即数占 12 位**
- **在参与算术运算前该 immediate 会被 “符号扩展” 为一个 32 位的数**
- **这个立即数可以表达的数值范围为: $[-2^{11}, +2^{11})$, 即 $[-2048, 2047)$ 。**

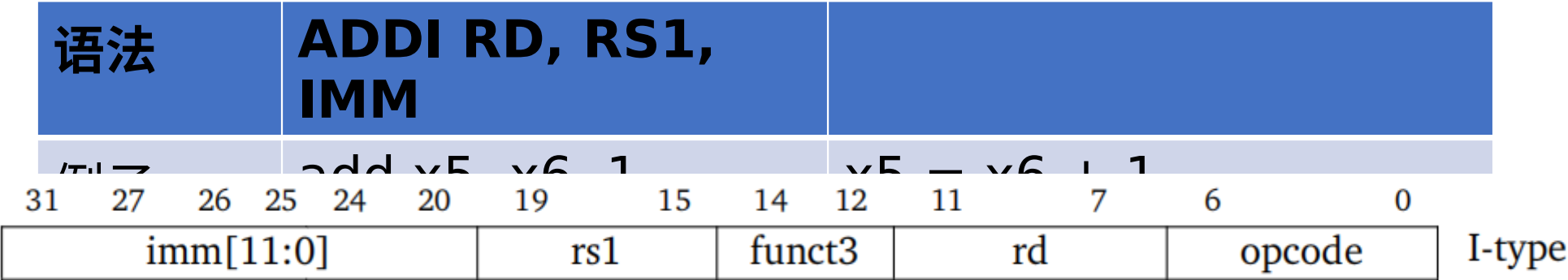


RISC-V ISA 并没有提供 SUBI 指令, why?

➤ 基于算术运算指令实现的其他伪指令

| 伪指令 | 语法 | 等价指令 | 指令描述 | 例子 |
|-----|------------|----------------|------------------------|------------|
| NEG | NEG RD, RS | SUB RD, x0, RS | 对 RS 中的值取反并将结果存放在 RD 中 | neg x5, x6 |
| MV | MV RD, RS | ADDI RD, RS, 0 | 将 RS 中的值拷贝到 RD 中 | mv x5, x6 |
| NOP | NOP | ADDI x0, x0, 0 | 什么也不做 | nop |

➤ ADDI 的局限性



给一个寄存器赋值的数值范围只有： [-2048, 2047)。



如果要赋值一个大数 (RV32) 怎么办？

解决思路：自己构造一个。

具体做法：

引入一个新的命令先设置高 20 位，存放在 rs1
复用现有的 ADDI 命令补上剩余的低 11 位即可

➤ LUI（Load Upper Immediate）

| 语法 | LUI RD, IMM | | | | | | | | | | | | | |
|------------|-----------------|--------------------|----|----|----|----|----|----|----|----|---|--------|---|--------|
| 例子 | lui x5, 0x12345 | x5 = 0x12345 << 12 | | | | | | | | | | | | |
| 31 | 27 | 26 | 25 | 24 | 20 | 19 | 15 | 14 | 12 | 11 | 7 | 6 | 0 | |
| imm[31:12] | | | | | | | | | | rd | | opcode | | U-type |

- LUI 指令采用 U-type :
 - ✓ opcode (7): U-types 的 opcode 是 0b0110111,
 - ✓ rd (5): “destination register” 用于存放结果
 - ✓ imm (20): “immediate”, 立即数
- LUI 指令会构造一个 32 bits 的立即数, 这个立即数的高 20 位对应指令中的 imm, 的低

- 利用 LUI + ADDI 来为寄存器加载一个大数
0x12345678



```
lui    x1, 0x12345    # x1 = 0x12345000  
addi   x1, x1, 0x678  # x1 = 0x12345678
```

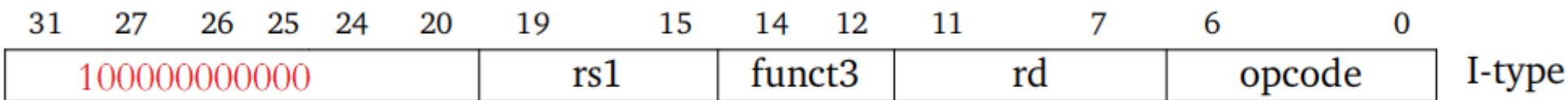
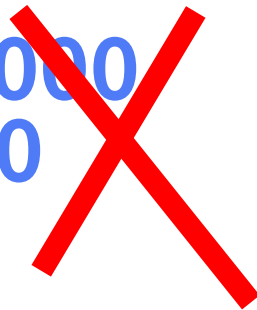
算术运算指令 (Arithmetic Instructions)

➤ 利用 LUI + ADDI 来为寄存器加载一个大数

0x12345800



lui x1, 0x12345 # x1 = 0x12345000
addi x1, x1, 0x800 # x1 = 0x12345800



参与算术运算前该 immediate 会被“符号扩展”为一个 32 位的数

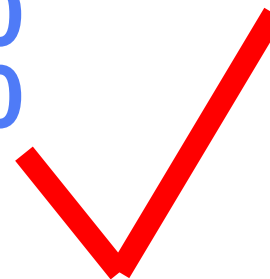
算术运算指令 (Arithmetic Instructions)

- 利用 LUI + ADDI 来为寄存器加载一个大数

0x12345800



```
lui    x1, 0x12346    # x1 = 0x12346000
addi   x1, x1, -0x800  # x1 = 0x12345800
```



算术运算指令 (Arithmetic Instructions)

➤ LI

| 语法 | LI RD, IMM | |
|----|-------------------|-----------------|
| 例子 | li x5, 0x12345678 | x5 = 0x12345678 |

- **LI (Load Immediate) 是一个伪指令 (pseudo-instruction)**
- **汇编器会根据 IMM 的实际情况自动生成正确的真实指令 (instruction)。**

Bingo !

算术运算指令 (Arithmetic Instructions)

➤ AUIPC

| 语法 | AUIPC RD, IMM | | | | | | | | | | | | | |
|------------|-------------------|--------------------|----|----|----|----|----|----|----|----|---|--------|---|--------|
| 例子 | auipc x5, 0x12345 | x5 = 0x12345 << 12 | | | | | | | | | | | | |
| 31 | 27 | 26 | 25 | 24 | 20 | 19 | 15 | 14 | 12 | 11 | 7 | 6 | 0 | |
| imm[31:12] | | | | | | | | | | rd | | opcode | | U-type |

- **AUIPC 指令采用 U-type**
- **和 LUI 指令类似，AUIPC 指令也会构造一个 32 bits 的立即数，这个立即数的高 20 位对应指令中的 imm，低 12 位清零。但和 LUI 不同的是，AUIPC 会先将这个立即数和 PC 值相加，将相加后的结果存放在 RD 中。**

➤ LA (Load Address)

| 语法 | LA RD, LABEL | |
|----|--------------|------|
| 例子 | la x5, foo | x5 = |

- **LA 是一个伪指令 (pseudo-instruction)**
- 具体编程时给出需要加载的 **label** , 编译器会根据实际情况利用 **auipc** 和其他指令自动生成正确的指令序列。
- 常用于加载一个函数或者变量的地址。

算术运算指令（Arithmetic Instructions）

| 指令 | 语法 | 描述 | | 例子 |
|-------|-------------------|--|------------------------|-------------------|
| ADD | ADD RD, RS1, RS2 | RS1 和 RS2 的值相加，结果保存到 RD | | add x5, x6, x7 |
| SUB | SUB RD, RS1, RS2 | RS1 的值减去 RS2 的值，结果保存到 RD | | sub x5, x6, x7 |
| ADDI | ADDI RD, RS1, IMM | RS1 的值和 IMM 相加，结果保存到 RD | | addi x5, x6, 100 |
| LUI | LUI RD, IMM | 构造一个 32 位的数，高 20 位存放 IMM，低 12 位清零。结果保存到 RD | | lui x5, 0x12345 |
| AUIPC | AUIPC RD, IMM | 构造一个 32 位的数，高 20 位存放 IMM，低 12 位清零。结果和 PC 相加后保存到 RD | | auipc x5, 0x12345 |
| | | 的组合 | | |
| NEG | NEG RD, RS | SUB RD, x0, RS | 对 RS 中的值取反并将结果存放在 RD 中 | neg x5, x6 |
| MV | MV RD, RS | ADDI RD, RS, 0 | 将 RS 中的值拷贝到 RD 中 | mv x5, x6 |
| NOP | NOP | ADDI x0, x0, 0 | 什么也不做 | nop |

逻辑运算指令（ Logical Instructions ）

| 指令 | 格式 | 语法 | 描述 | 例子 |
|------|--------|-------------------|-----------------------|-----------------|
| AND | R-type | AND RD, RS1, RS2 | $RD = RS1 \& RS2$ | and x5, x6, x7 |
| OR | R-type | OR RD, RS1, RS2 | $RD = RS1 RS2$ | or x5, x6, x7 |
| XOR | R-type | XOR RD, RS1, RS2 | $RD = RS1 \wedge RS2$ | xor x5, x6, x7 |
| ANDI | I-type | ANDI RD, RS1, IMM | $RD = RS1 \& IMM$ | andi x5, x6, 20 |
| ORI | I-type | ORI RD, RS1, IMM | $RD = RS1 IMM$ | or x5, x6, 20 |
| XORI | I-type | XORI RD, RS1, IMM | $RD = RS1 \wedge IMM$ | xor x5, x6, 20 |

- 所有的逻辑指令都是按位操作
- **XOR（eXclusive OR，“异或”）**：两个 bit 值不同（异）则取值为 1（达到类似取 1 为 OR 的效果）；如果两个 bit 相同则取值为 0。

| 伪指令 | 语法 | 等价指令 | 描述 | 例子 |
|-----|------------|-----------------|------------------------|------------|
| NOT | NOT RD, RS | XORI RD, RS, -1 | 对 RS 的值按位取反，结果存放在 RD 中 | not x5, x6 |
| | | | | |

移位运算指令（Shifting Instructions）

➤ 逻辑移位

| 指令 | 格式 | 语法 | 描述 | 例子 |
|------|--------|-------------------|--|----------------|
| SLL | R-type | SLL RD, RS1, RS2 | 逻辑左移（Shift Left Logical）， $RD = RS1 \ll RS2$ | sll x5, x6, x7 |
| SRL | R-type | SRL RD, RS1, RS2 | 逻辑右移（Shift Right Logical） $RD = RS1 \gg RS2$ | srl x5, x6, x7 |
| SLLI | I-type | SLLI RD, RS1, IMM | 逻辑左移立即数（Shift Left Logical Immediate） $RD = RS1 \ll IMM$ | slli x5, x6, 3 |
| SRLI | I-type | SRLI RD, RS1, IMM | 逻辑右移立即数（Shift Right Logical Immediate） $RD = RS1 \gg IMM$ | srli x5, x6, 3 |

无论是逻辑左移还是逻辑右移，补足的都是 0

移位运算指令（ Shifting Instructions ）

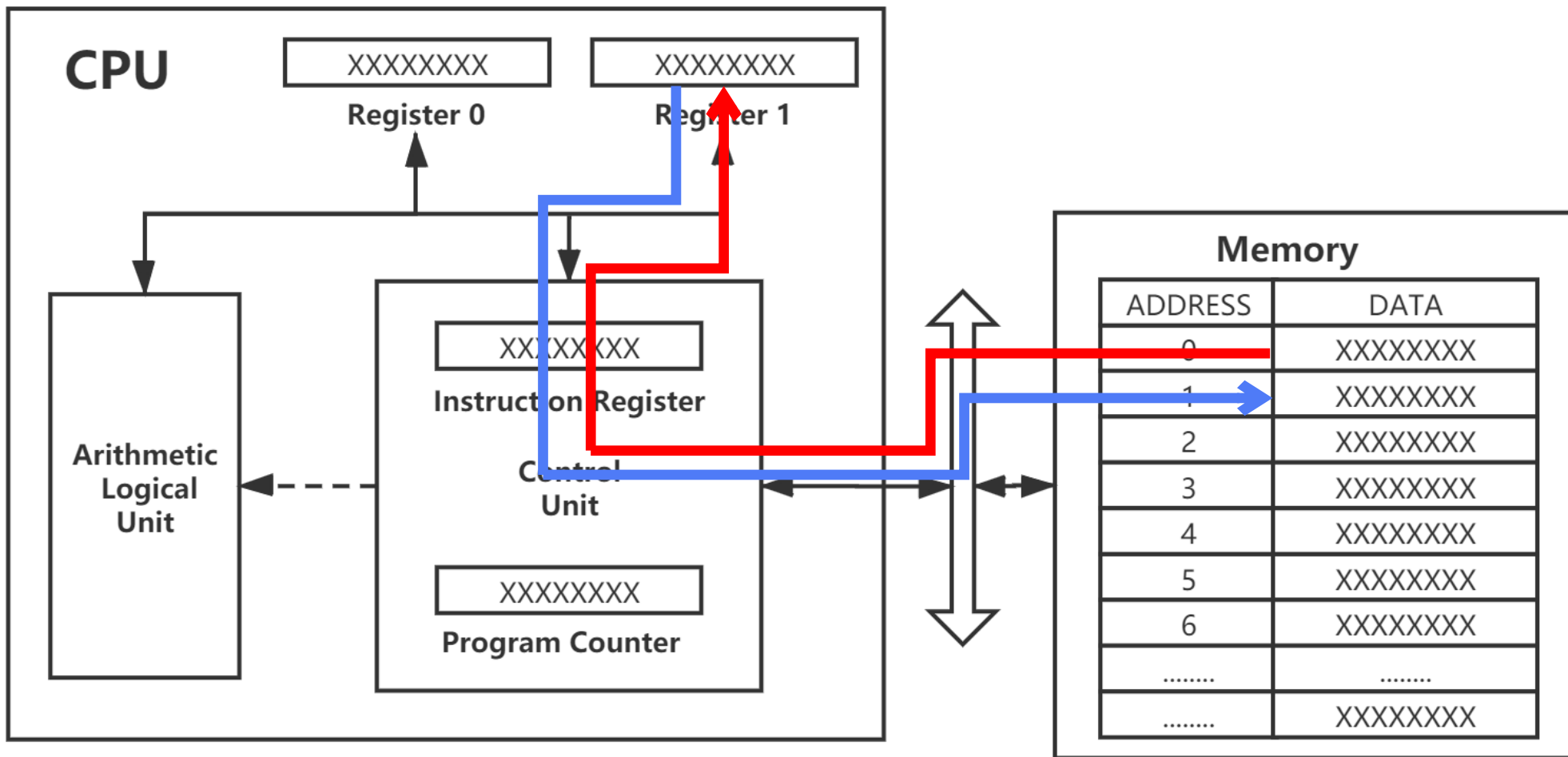
➤ 算术移位

| 指令 | 格式 | 语法 | 描述 | 例子 |
|------|--------|----------------------|--|----------------|
| SRA | R-type | SRA RD,RS1,RS2 | 算术右移（ Shift Right Arithmetic ） RD= RS1 >> RS2 | sra x5, x6, x7 |
| SRAI | I-type | SRAI RD, RS1, IMM | 算术右移立即数（ Shift Right Arithmetic Immediate ） RD= RS1 >> IMM | srai x5, x6, 3 |

- 算术右移时按照符号位值补足。
- 对于算术移位，只有算术右移，没有算术左移，左移会导致 **MSB**（符号位）丢失，失去算术移位的意义。

内存读写指令（Load and Store Instructions）

- 内存读指令：Load，将数据从内存读入寄存器
- 内存写指令：Store，将数据从寄存器写出到内存



内存读写指令（Load and Store Instructions）

➤ 内存读（Load）

| 指令 | 格式 | 语法 | 描述 | 例子 |
|-----|--------|------------------|--|----------------|
| LB | I-type | LB RD, IMM(RS1) | Load Byte，从内存中读取一个 8 bits 的数据到 RD 中，内存地址 = $RS1 + IMM$ ，数据在保存到 RD 之前会执行 sign-extended 。 | lb x5, 40(x6) |
| LBU | I-type | LBU RD, IMM(RS1) | Load Byte Unsigned，从内存中读取一个 8 bits 的数据到 RD 中，内存地址 = $RS1 + IMM$ ，数据在保存到 RD 之前会执行 zero-extended 。 | lbu x5, 40(x6) |
| LH | I-type | LH RD, IMM(RS1) | Load Halfword，从内存中读取一个 16 bits 的数据到 RD 中，内存地址 = $RS1 + IMM$ ，数据在保存到 RD 之前会执行 sign-extended 。 | lh x5, 40(x6) |
| LHU | I-type | LHU RD, IMM(RS1) | Load Halfword Unsigned，从内存中读取一个 16 bits 的数据到 RD 中，内存地址 = $RS1 + IMM$ ，数据在保存到 RD 之前会执行 zero-extended 。 | lhu x5, 40(x6) |
| LW | I-type | LW RD, IMM(RS1) | Load Word，从内存中读取一个 32 bits 的数据到 RD 中，内存地址 = $RS1 + IMM$ | lw x5, 40(x6) |

注意：IMM 给出的偏移量范围是 **$[-2048, 2047]$** 。



为何对 word 的 load 不区分无符号和有符号方式？

➤ 内存写（Store）

| 指令 | 格式 | 语法 | 描述 | 例子 |
|----|--------|------------------|--|---------------|
| SB | S-type | SB RS2, IMM(RS1) | Store Byte，将 RS2 寄存器中低 8 bits 的数据写出到内存中，内存地址 = RS1 + IMM。 | sb x5, 40(x6) |
| SH | S-type | SH RS2, IMM(RS1) | Store Halfword，将 RS2 寄存器中低 16 bits 的数据写出到内存中，内存地址 = RS1 + IMM。 | sh x5, 40(x6) |
| SW | S-type | SW RS2, IMM(RS1) | Store Word，将 RS2 寄存器中的 32 bits 的数据写出到内存中，内存地址 = RS1 + IMM。 | sw x5, 40(x6) |

3127262524201915141211760

imm[11:5]rs2rs1funct3imm[4:0]opcode

S-type

注意：IMM 给出的偏移量范围是 [-2048, 2047]。



为何对于 load 要区分无符号方式和有符号方式，而 store 不区分？

条件分支指令（Conditional Branch Instructions）

| 指令 | 格式 | 语法 | 描述 | 例子 |
|---|--------|------------------|---|------------------|
| BEQ | B-type | BEQ RS1,RS2,IMM | Branch if EQual。比较 RS1 和 RS2 的值，如果相等，则执行路径跳转到一个新的地址。 | beq x5, x6, 100 |
| BNE | B-type | BNE RS1,RS2,IMM | Branch if Not Equal。比较 RS1 和 RS2 的值，如果不相等，则执行路径跳转到一个新的地址。 | bne x5, x6, 100 |
| BLT | B-type | BLT RS1,RS2,IMM | Branch if Less Than。按照 有符号方式 比较 RS1 和 RS2 的值，如果 $RS1 < RS2$ ，则执行路径跳转到一个新的地址。 | blt x5, x6, 100 |
| BLT U | B-type | BLTU RS1,RS2,IMM | Branch if Less Than (Unsigned)。按照 无符号方式 比较 RS1 和 RS2 的值，如果 $RS1 < RS2$ ，则执行路径跳转到一个新的地址。 | bltu x5, x6, 100 |
| BGE | B-type | BGE RS1,RS2,IMM | Branch if Greater than or Equal。按照 有符号方式 比较 RS1 和 RS2 的值，如果 $RS1 \geq RS2$ ，则执行路径跳转到一个新的地址。 | bge x5, x6, 100 |
| BLT U | B-type | BLTU | Branch if Greater than or Equal (Unsigned)。按照 无符号方式 比较 RS1 和 RS2 的值，如果 $RS1 \geq RS2$ ，则执行路径跳转到一个新的地址。 | bgeu x5, x6, 100 |
| <div><div>3127262524201915141211760</div><div>imm[12 10:5]rs2rs1funct3imm[4:1 11]opcode</div></div> <div>B-type</div> | | | | |

- 跳转的目标地址计算方法：先将 IMM x 2，符号扩展后和 PC 值相加得到最终的目标地址，所以跳转范围是以 PC 为基准， +/- 4KB 左右 ([-4096, 4094])。
- 具体编程时，不会直接写 IMM，而是用标号代替，交由链接器来最终决定 IMM 的值。

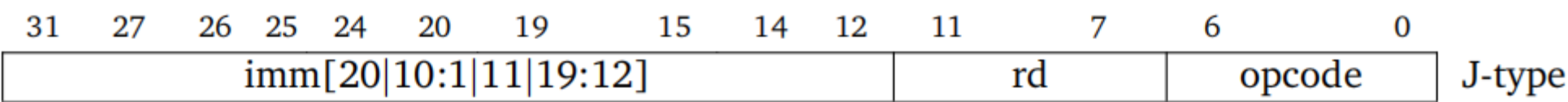
条件分支指令（Conditional Branch Instructions）



| 伪指令 | 语法 | 等价指令 | 描述 |
|------|---------------------|---------------------|---|
| BLE | BLE RS, RT, OFFSET | BGE RT, RS, OFFSET | Branch if Less & Equal ，有符号方式比较，如果 $RS \leq RT$ ，跳转到 OFFSET |
| BLEU | BLEU RS, RT, OFFSET | BGEU RT, RS, OFFSET | Branch if Less or Equal Unsigned ，无符号方式比较，如果 $RS \leq RT$ ，跳转到 OFFSET |
| BGT | BGT RS, RT, OFFSET | BLT RT, RS, OFFSET | Branch if Greater Than ，有符号方式比较，如果 $RS > RT$ ，跳转到 OFFSET |
| BGTU | BGTU RS, RT, OFFSET | BLTU RT, RS, OFFSET | Branch if Greater Than Unsigned ，无符号方式比较，如果 $RS > RT$ ，跳转到 OFFSET |
| BEQZ | BEQZ RS, OFFSET | BEQ RS, x0, OFFSET | Branch if Equal Zero, 如果 $RS == 0$ ，跳转到 OFFSET |
| BNEZ | BNEZ RS, OFFSET | BNE RS, x0, OFFSET | Branch if Not Equal Zero, 如果 $RS \neq 0$ ，跳转到 OFFSET |
| BLTZ | BLTZ RS, OFFSET | BLT RS, x0, OFFSET | Branch if Less Than Zero, 如果 $RS < 0$ ，跳转到 OFFSET |
| BLEZ | BLEZ RS, OFFSET | BGE x0, RS, OFFSET | Branch if Less or Equal Zero, 如果 $RS \leq 0$ ，跳转到 OFFSET |
| BGTZ | BGTZ RS, OFFSET | BLT x0, RS, OFFSET | Branch if Greater Than Zero ，如果 $RS > 0$ ，跳转到 OFFSET |

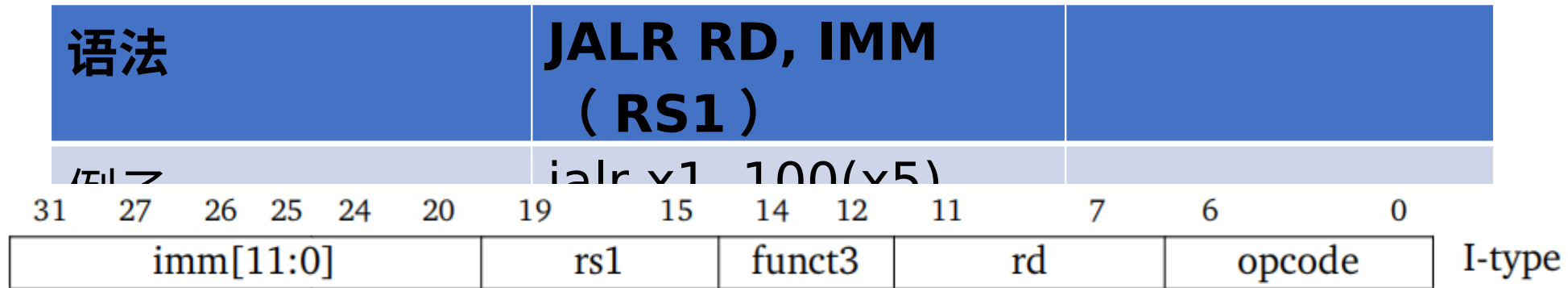
➤ JAL (Jump And Link)

| | | |
|----|---------------|--|
| 语法 | JAL RD, LABEL | |
| 例子 | jal x1, label | |



- JAL 指令使用 J-type 编码格式。
- JAL 指令用于调用子过程 (subroutine/function)。
- 子过程的地址计算方法：首先对 20 bits 宽的 IMM x 2 后进行 sign-extended，然后将符号扩展后的值和 PC 的值相加。因此该函数跳转的范围是以 PC 为基准，上下 ~+/- 1MB。
- JAL 指令的下一条指令的地址写入 RD，保存为返回地址。
- 实际编程时，用 label 给出跳转的目标，具体 IMM

➤ JALR (Jump And Link Register)



- JALR 指令使用 **I-type** 编码格式。
- JALR 指令用于调用子过程 (subroutine/function)。
- 子过程的地址计算方法：首先对 12 bits 宽的 IMM 进行 **sign-extended**，然后将符号扩展后的值和 RS1 的值相加，得到最终的结果后将其最低位设置为 0（确保地址按 2 字节对齐）。因此该函数跳转的范围是以 RS1 为基准，上下 $\sim \pm 2\text{KB}$ 。
- JALR 指令的下一条指令的地址写入 RD，保存为返回地址。

无条件跳转指令（ Unconditional Jump Instructions ）

➤ 如何解决更远距离的跳转？

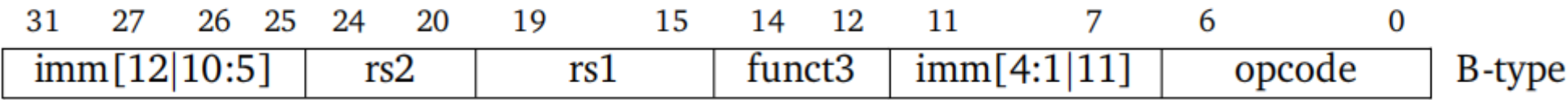
- **AUIPC X6, IMM-20**
- **JALR X1,X6, IMM-12**

如果跳转后不需要返回，可以利用 **x0** 代替 **JAL** 和 **JALR** 中的 **RD**

| 伪指令 | 语法 | 等价指令 | 描述 | 例子 |
|-----|-----------|----------------|----|----|
| J | J OFFSET | JAL X0, OFFSET | | |
| JR | JR OFFSET | JALR X0, 0(RS) | | |

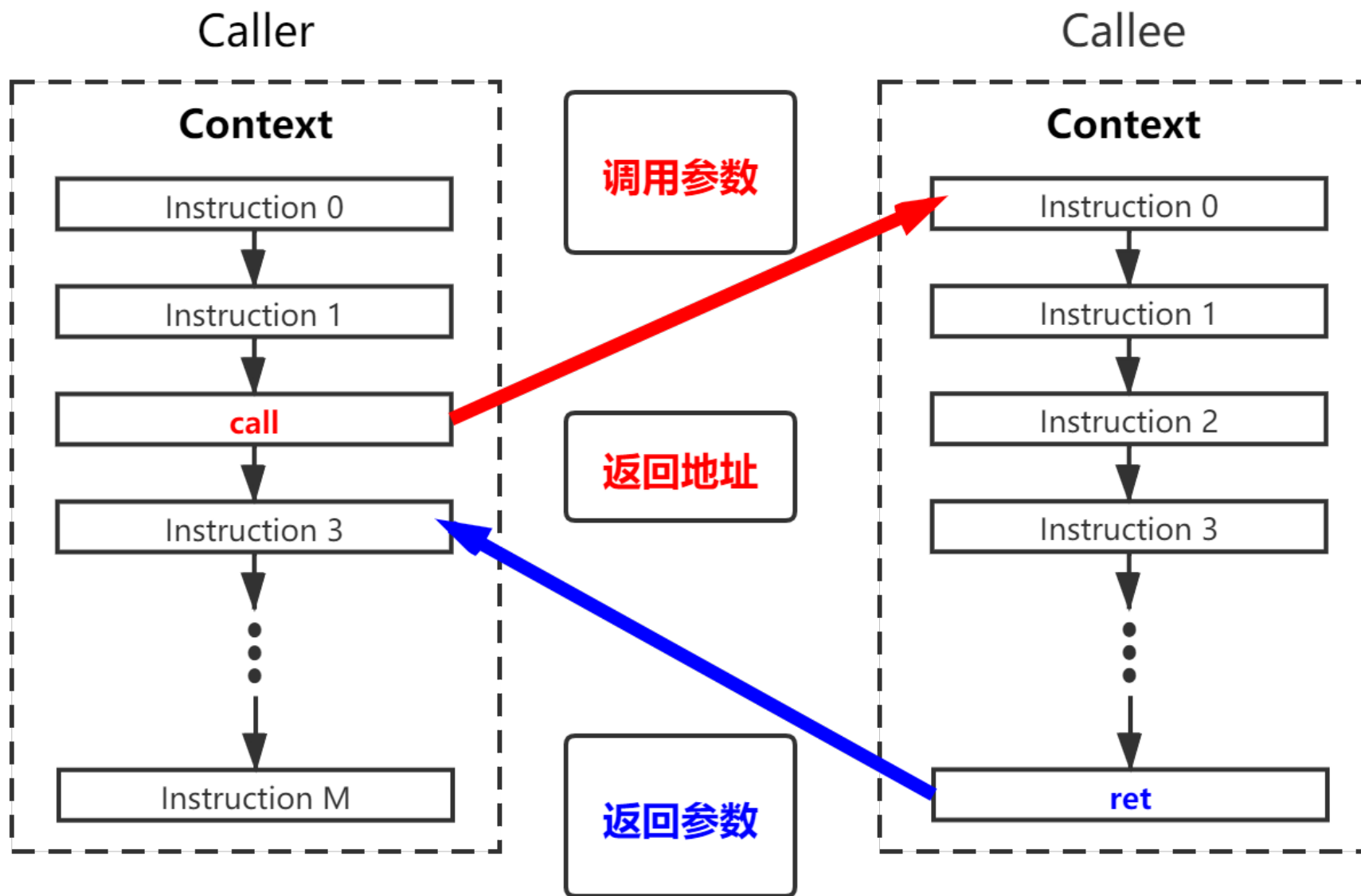
所谓寻址模式指的是指令中定位操作数（operand）或者地址的方式。

| 寻址模式 | 解释 | 例子 |
|---------|--|-----------------|
| 立即数寻址 | 操作数是指令本身的一部分 | addi x5, x6, 20 |
| 寄存器寻址 | 操作数存放在寄存器中，指令中指定访问的寄存器从而获取该操作数。 | addi x5, x6, 20 |
| 基址寻址 | 操作数在内存中，指令中通过指定寄存器（基址 base）和立即数（偏移量 offset），通过 base + offset 的方式获得操作数在内存中的地址从而获取该操作数 | lw x5, 40(x6) |
| PC 相对寻址 | 分支和跳转指令中，通过 PC 和指令中的立即数相加获得目标地址的值 | beq x5, x6, 100 |

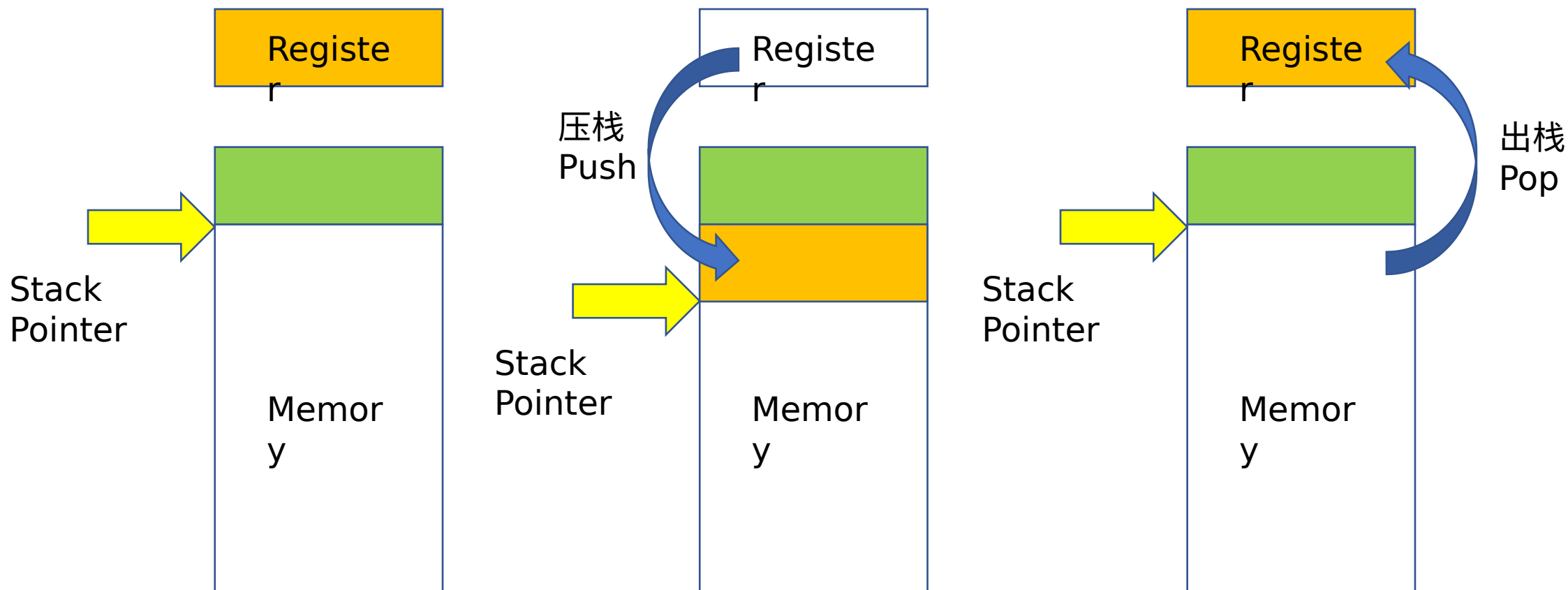


- RISC-V 汇编语言入门
- RISC-V 汇编指令总览
- RISC-V 汇编指令详解
- **RISC-V 汇编函数调用约定**
 - 汇编编程时为何需要制定函数调用约定
 - 函数调用栈的概念
 - 函数调用过程中有关寄存器的编程约定
 - 函数调用过程中函数跳转和返回指令的编程约定
 - 函数调用过程中实现被调用函数的编程约定
- RISC-V 汇编与 C 混合编程

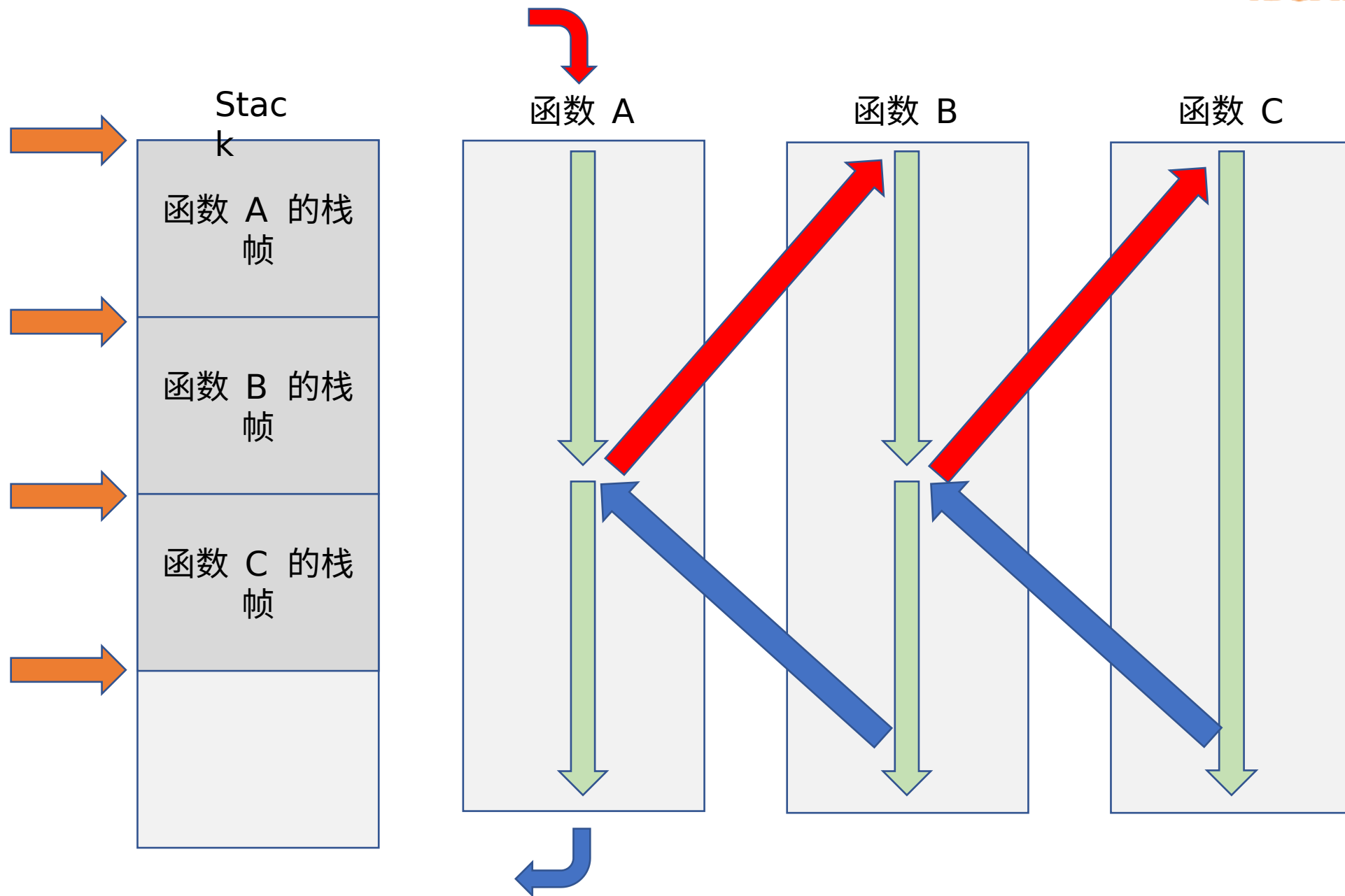
汇编编程时为何需要制定函数调用约定（Calling Conventions）



函数调用栈的概念（1）



函数调用栈的概念（2）



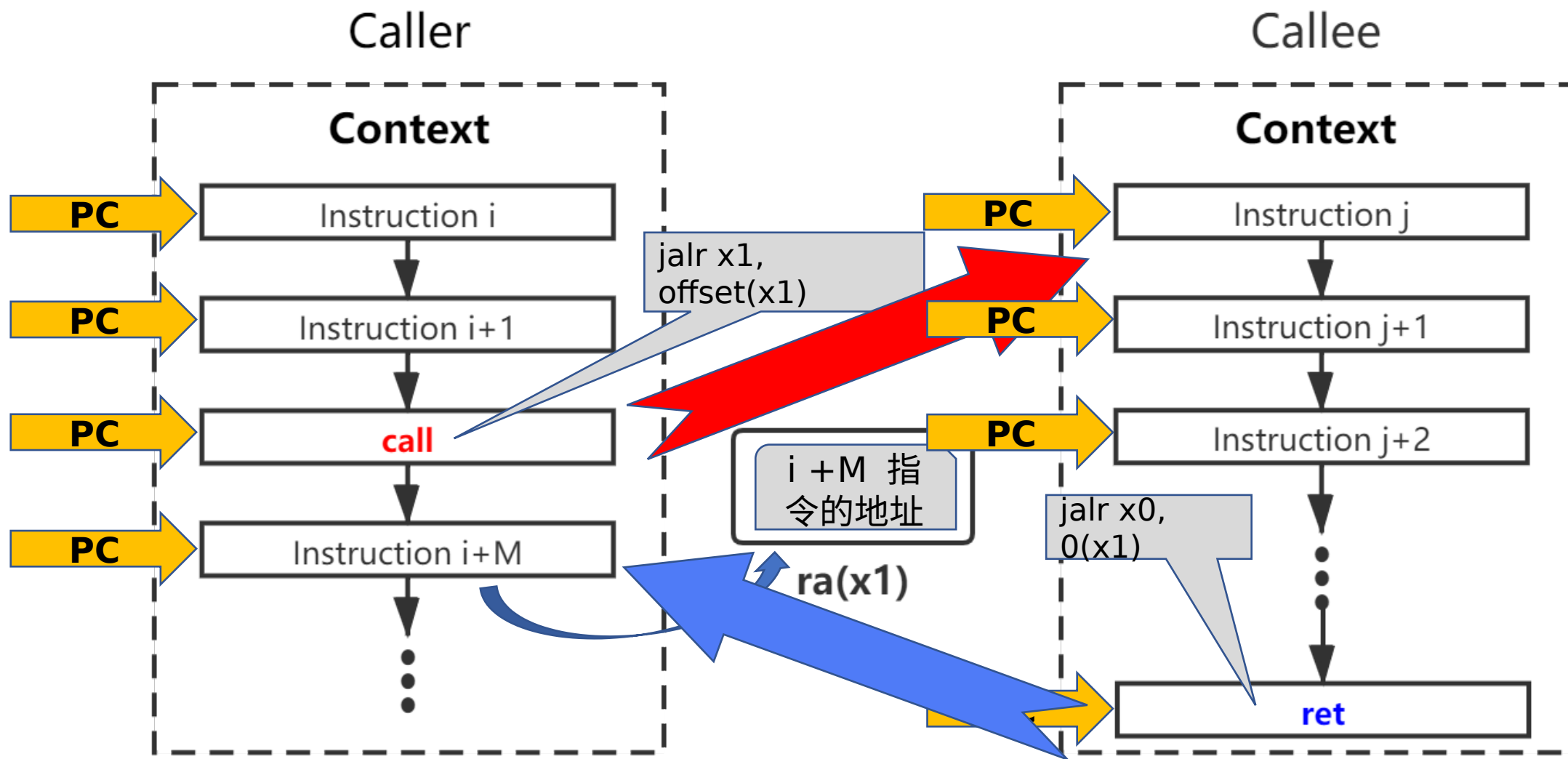
函数调用过程中有关寄存器的编程约定

| 寄存器名 | ABI 名 (编程用名) | 用途约定 | 谁负责保存 |
|--------------------|-------------------|--|--------|
| x0 | zero | 读取时总为 0 ， 写入时不起任何效果 | N/A |
| x1 | ra | 存放函数返回值（ return address ） | Caller |
| x2 | sp | 存放栈指针（ stack pointer ） | Callee |
| x5~x7, x28~x31 | t0~t2, t3~t6 | 临时（ temporaries ）寄存器，相对 Caller 来说 Callee 不保证这些寄存器中的值在函数调用过程中保持不变，这意味着如果需要的话， Caller 需要自己在调用 Callee 之前保存临时寄存器中的值。 | Caller |
| x8, x9, x18~x27 | s0, s1, s2~s11 | 保存（ saved ）寄存器，对于 Callee 来说需要保证这些寄存器的值在函数返回后仍然维持原值，所以一旦 Callee 在自己的函数中会用到这些寄存器则需要在栈中备份并在退出函数时进行恢复。 | Callee |
| x10 , x11 | a0 , a1 | 用于在函数调用过程中保存第一个和第二个参数，以及在函数返回时传递返回值。 | Caller |
| x12 ~ x17 | a2 ~ a7 | 如果函数调用时需要传递更多的参数，则可以用这些寄存器，但注意用于传递参数的寄存器最多只有 8 个（ a0 ~ a7 ），如果还有更多的参数则要利用栈。 | Caller |

函数调用过程中函数跳转和返回指令的编程约定（1）

| 伪指令 | 等价指令 | 描述 | 例子 |
|-------------|---|----------------------------------|----------|
| jal offset | jal x1, offset | 跳转到 offset 制定位置，返回地址保存在 x1 (ra) | jal foo |
| jalr rs | jalr x1, 0(rs) | 跳转到 rs 中值所指定的位置，返回地址保存在 x1 (ra) | jalr s1 |
| j offset | jal x0, offset | 跳转到 offset 制定位置，不保存返回地址 | j loop |
| jr rs | jalr x0, 0(rs) | 跳转到 rs 中值所指定的位置，不保存返回地址 | jr s1 |
| call offset | auipc x1, offset[31 : 12] + offset[11] jalr x1, offset[11:0] (x1) | 长跳转调用函数 | call foo |
| tail offset | auipc x6, offset[31 : 12] + offset[11] jalr x0, offset[11:0](x6) | 长跳转尾调用 | tail foo |

函数调用过程中函数跳转和返回指令的编程约定（2）



def function ()

{

函数起始部分（ Prologue ）

减少 sp 的值，根据本函数中使用 saved 寄存器的情况以及 local 变量的多少开辟栈空间。

将 saved 寄存器的值保存到栈中

如果函数体还会调用其他的函数，则将 ra 寄存器的值保存到栈中

函数退出部分（ Epilogue ）

从栈中恢复 saved 寄存器

如果需要的话，从栈中恢复 ra 寄存器

增加 sp 的值，恢复到进入本函数之前的状态。

调用 ret 返回

}

- RISC-V 汇编语言入门
- RISC-V 汇编指令总览
- RISC-V 汇编指令详解
- RISC-V 汇编函数调用约定
- RISC-V 汇编与 C 混合编程
 - RISC-V 汇编调用 C 函数
 - C 函数中嵌入 RISC-V 汇编

➤ 遵守 ABI （ Abstract Binary Interface ） 的规定

- 数据类型的大小，布局和对齐
- 函数调用约定 （ Calling Convention ）
- 系统调用约定
-

➤ RISC-V 函数调用约定规定：

- 函数参数采用寄存器 a0 ~ a7 传递
- 函数返回值采用寄存器 a0 和 a1 传递

```
asm volatile (  
    汇编指令列表  
    : 输出操作数 (可选)  
    : 输入操作数 (可选)  
    : 可能影响的寄存器或者存储器 (可选)  
);
```

指令列表用双引号括起来，多条指令之间用 ";" 或者 "\n" 分隔

“输出操作数”和“输入操作数”用于将需要操作的 C 变量和汇编指令的操作数对应起来。

“可能影响的寄存器或者存储器”用于告知编译器当前嵌入的汇编语句可能修改的寄存器或者内存，方便编译器执行优化。

谢谢

欢迎交流合作