

# 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汇编语言**

- 汇编语言简介
- RISC-V汇编指令概览
- RISC-V常用汇编指令
- 函数调用及栈的使用
- 编译工具介绍



# 汇编语言是什么？

---

- 汇编语言（**Assembly Language**）是一种“低级”语言，直接接触最底层的硬件，需要对底层硬件非常熟悉才能编写出高效的汇编程序。
- 汇编语言属于第二代计算机语言，用一些容易理解和记忆的字母，单词来代替一个特定的指令。在汇编语言中，用助记符代替机器指令的操作码，用地址符号或标号代替指令或操作数的地址。
- 例如：用“ADD”代表数字逻辑上的加減，“MOV”代表数据传递等等。

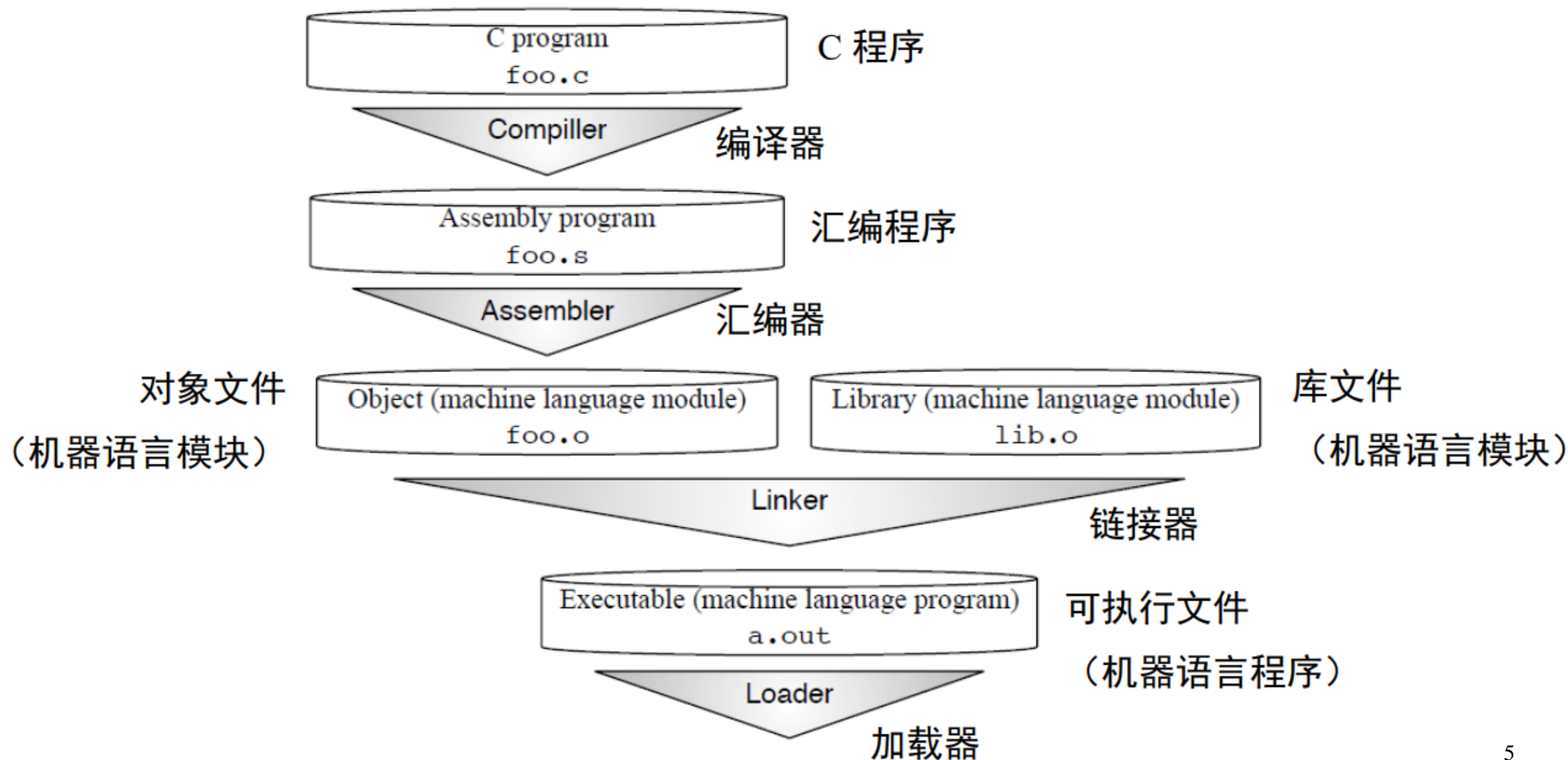
# 汇编语言简介

---

- C语言程序在翻译成可以在计算机上运行的机器语言程序，大致需要四个步骤。
  - 首先将高级语言程序编译成汇编语言程序
  - 然后用机器语言组装成目标模块
  - 链接器将多个模块与库程序组合在一起以解析所有引用
  - 最后加载器将机器代码放入适当的存储器位置以供处理器执行

注：所有程序不一定严格按照这个四个步骤执行。为了加快转换过程，可以跳过或将一些步骤组合到一起。一些编译器直接生成目标模块，一些系统使用链接加载器执行最后两个步骤。

# 翻译并启动程序（C语言的转换层次结构）



# 编译器、汇编器

- **编译器** 将C程序转换为机器能理解的符号形式——汇编语言程序（assembly language program）。
- **汇编器** 将汇编语言转换为目标文件，该目标文件是机器指令、数据和将指令正确放入内存所需信息的组合。
  - 汇编器还可以处理机器指令的常见变体，这类指令称为**伪指令**。

如：li x9, 123          汇编器将其转化为 addi x9, x0, 123

mv x11, x10          汇编器将其转化为 addi x11, x10, 0

and x9, x10, 15      汇编器将其转化为 andi x9, x10, 15

**伪指令为RISC-V提供了比硬件实现更丰富的汇编语言指令系统。**

# 链接器、加载器、动态链接库

- **链接器** 链接器将所有独立汇编的机器语言程序“缝合”在一起，最终生成可执行文件。由于独立编译和汇编每个过程，因此更改一行代码只需要编译和汇编一个过程，链接器有用的原因是修正代码要比重新编译和重新汇编快得多。
- **加载器** 将可执行程序放在主存中以准备执行的系统程序。
- **动态链接库（DLL）** 是在**执行期间**链接到程序的库例程。DLL需要**额外的空间**来存储动态链接所需的信息，但不要求复制或链接整个库。它们在第一次调用历程时会付出大量的开销，但此后只需一个间接跳转。

# 汇编语言的优缺点

---

- 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
  - 需要对性能进行优化



# 第二章 RISC-V汇编及其指令系统

---

- **RISC-V汇编语言**
  - 汇编语言简介
  - **RISC-V汇编指令概览**
  - RISC-V常用汇编指令
  - 函数调用及栈的使用
  - 编译工具介绍

# RISC-V 汇编指令格式

- 通用指令格式

**op dst, src1, src2**

- 1个操作码，3个操作数
- op 操作的名字（**o**peration）
- dst 目标寄存器（destination）
- src1 第一个源操作数寄存器（source）
- src2 第二个源操作数寄存器

- 通过一些限制来保持硬件简单

- 硬件设计基本原则之一：简单源于规整

为什么是3个操作数？4个操作数行不行？



# RISC-V 汇编格式

---

- 每一条指令只有一个操作，每一行最多一条指令
- 汇编指令与C语言的操作相关（=, +, -, \*, /, &, |, 等）
  - C语言中的操作会被分解为一条或者多条汇编指令
  - C语言中的一行程序会被编译为多行RISC-V汇编程序

# RISC-V汇编指令操作对象

- 寄存器:

- 32个通用寄存器,  $x0 \sim x31$  (**注意: 仅涉及 RV32I 的通用寄存器组**)
- 在 RISC-V 中, 算术逻辑运算所操作的数据必须直接来自寄存器
- $x0$ 是一个特殊的寄存器, 只用于全零
- 每一个寄存器都有别名便于软件使用, 实际硬件并没有任何区别

- 内存:

- 可以执行在寄存器和内存之间的数据读写操作
- 读写操作使用字节 (Byte) 为基本单位进行寻址
- RV64可以访问最多  $2^{64}$  个字节的内存空间, 即 $2^{61}$ 个存储单元

# 汇编语言的变量---寄存器

---

- 汇编语言不能使用变量（C、JAVA可以）
  - C语言：int a; float b;
  - 汇编语言：寄存器变量没有数据类型
- 汇编语言的操作对象以寄存器为主
  - 好处：寄存器是最快的数据单元
  - 缺陷：寄存器数量有限，需要充分和高效地使用各寄存器

# 32个RISC-V寄存器

寄存器	助记符	注解
<b>x0</b>	<b>zero</b>	<b>固定值为0</b>
x1	<u>ra</u>	返回地址(Return Address)
x2	<u>sp</u>	栈指针(Stack Pointer)
x3	<u>gp</u>	全局指针(Global Pointer)
x4	<u>tp</u>	线程指针(Thread Pointer)
x5-x7	t0-t2	临时寄存器
x8	s0/ <u>fp</u>	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
x28-x31	t3-6	临时寄存器

# RISC-V汇编语言指令分类

- 算术运算: add、sub、addi、mul、div
- 逻辑运算: and、or、xor、andi、ori、xori
- 移位操作: sll、srl、sra、slli、srli、srai
- 数据传输: ld、sd、lw、sw、lwu、lh、lhu、sh、lb、lbu、sb、lui
- 比较指令: slt、slti、sltu、sltiu
- 条件分支: beq、bne、blt、bge、bltu、bgeu
- 无条件跳转: jal、jalr

助记符后缀

i = “immediate” 是个整型的常量

u = “unsigned” 无符号类型

## RISC-V 汇编语言

类别	指令	示例	含义	注解
算术运算	加	add x5, x6, x7	$x5 = x6 + x7$	三寄存器操作数；加
	减	sub x5, x6, x7	$x5 = x6 - x7$	三寄存器操作数；减
	立即数加	addi x5, x6, 20	$x5 = x6 + 20$	用于加常数
数据传输	取双字	ld x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	从存储器取双字到寄存器
	存双字	sd x5, 40(x6)	$\text{Memory}[x6 + 40] = x5$	从寄存器存双字到存储器
	取字	lw x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	从存储器取字到寄存器
	取字（无符号数）	lwu x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	从存储器取无符号字到寄存器
	存字	sw x5, 40(x6)	$\text{Memory}[x6 + 40] = x5$	从寄存器存字到存储器
	取半字	lh x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	从存储器取半字到寄存器
	取半字（无符号数）	lhu x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	从存储器取无符号半字到寄存器
数据传输	存半字	sh x5, 40(x6)	$\text{Memory}[x6 + 40] = x5$	从寄存器存半字到存储器
	取字节	lb x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	从存储器取字节到寄存器
	取字节（无符号数）	lbu x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	从存储器取无符号字节到寄存器
	存字节	sb x5, 40(x6)	$\text{Memory}[x6 + 40] = x5$	从寄存器存字节到存储器
	取保留字	lr.d x5, (x6)	$x5 = \text{Memory}[x6]$	取；原子交换的前半部分
	存条件字	sc.d x7, x5, (x6)	$\text{Memory}[x6] = x5; x7 = 0/1$	存；原子交换的后半部分
	取立即数高位	lui x5, 0x12345	$x5 = 0x12345000$	取左移12位后的20位立即数
逻辑运算	与	and x5, x6, x7	$x5 = x6 \& x7$	三寄存器操作数；按位与
	或	or x5, x6, x8	$x5 = x6   x8$	三寄存器操作数；按位或
	异或	xor x5, x6, x9	$x5 = x6 \wedge x9$	三寄存器操作数；按位异或
	与立即数	andi x5, x6, 20	$x5 = x6 \& 20$	寄存器与常数按位与
	或立即数	ori x5, x6, 20	$x5 = x6   20$	寄存器与常数按位或
	异或立即数	xori x5, x6, 20	$x5 = x6 \wedge 20$	寄存器与常数按位异或



移位操作	逻辑左移	sll x5, x6, x7	x5 = x6 << x7	按寄存器给定位数左移
	逻辑右移	srl x5, x6, x7	x5 = x6 >> x7	按寄存器给定位数右移
	算术右移	sra x5, x6, x7	x5 = x6 >> x7	按寄存器给定位数算术右移
	逻辑左移立即数	slli x5, x6, 3	x5 = x6 << 3	根据立即数给定位数左移
	逻辑右移立即数	srl i x5, x6, 3	x5 = x6 >> 3	根据立即数给定位数右移
	算术右移立即数	srai x5, x6, 3	x5 = x6 >> 3	根据立即数给定位数算术右移
条件分支	相等即跳转	beq x5, x6, 100	if (x5 == x6) go to PC+100	若寄存器数值相等则跳转到PC相对地址
	不等即跳转	bne x5, x6, 100	if (x5 != x6) go to PC+100	若寄存器数值不等则跳转到PC相对地址
	小于即跳转	blt x5, x6, 100	if (x5 < x6) go to PC+100	若寄存器数值比较结果小于则跳转到PC相对地址
	大于等于即跳转	bge x5, x6, 100	if (x5 >= x6) go to PC+100	若寄存器数值比较结果大于或等于则跳转到PC相对地址
	小于即跳转 (无符号)	bltu x5, x6, 100	if (x5 < x6) go to PC+100	若寄存器数值比较结果小于则跳转到PC相对地址 (无符号)
	大于等于即跳转 (无符号)	bgeu x5, x6, 100	if (x5 >= x6) go to PC+100	若寄存器数值比较结果大于或等于则跳转到PC相对地址 (无符号)
无条件跳转	跳转-链接	jal x1, 100	x1 = PC+4; go to PC+100	用于PC相关的过程调用
	跳转-链接 (寄存器地址)	jalr x1, 100(x5)	x1 = PC+4; go to x5+100	用于过程返回; 非直接调用

# 第二章 RISC-V汇编及其指令系统

---

- **RISC-V汇编语言**
  - 汇编语言简介
  - RISC-V汇编指令概览
  - **RISC-V常用汇编指令**
  - 函数调用及栈的使用
  - 编译工具介绍

# RISC-V指令

---

- 算术运算: add、sub、addi、mul、div
- 逻辑运算: and、or、xor、andi、ori、xori
- 移位操作: sll、srl、sra、slli、srli、srai
- 数据传输: ld、sd、lw、sw、lwu、lh、lhu、sh、lb、lbu、sb、lui
- 比较指令: slt、slti、sltu、sltiu
- 条件分支: beq、bne、blt、bge、bltu、bgeu
- 无条件跳转: jal、jalr

# 算术运算指令——加减法

- **加法: add**

- 使用语法: `add rd, rs1, rs2`
- 功能描述: **`rd=rs1+rs2`** （忽略算术溢出）

- **减法: sub**

- 使用语法: `sub rd, rs1, rs2`
- 功能描述: **`rd=rs1 - rs2`** （忽略算术溢出）

- 溢出是因为计算机中表达数本身是有**范围限制**的

- 计算的结果没有足够多的位数进行表达
- RISC-V 忽略溢出问题，高位被截断，低位写入目标寄存器

# 算术运算指令——加减法的例子

---

- 变量a, b和c被分别放置在寄存器x1, x2,和x3中
- 整数的**加法 (add)**
  - C语言:  $a = b + c$
  - RISC-V: `add x1, x2, x3`
- 整数的**减法 (sub)**
  - C语言:  $a = b - c$
  - RISC-V: `sub x1, x2, x3`

# 算术运算举例

- 假设 $a \rightarrow x1$ ,  $b \rightarrow x2$ ,  $c \rightarrow x3$ ,  $d \rightarrow x4$ ,  $e \rightarrow x5$ 。对于如下C语言程序

$$a = (b + c) - (d + e)$$

可写成如下RISC-V汇编指令

<b>add x6, x4, x5</b>	# tmp1 = d + e
<b>add x7, x2, x3</b>	# tmp2 = b + c
<b>sub x1, x7, x6</b>	# a = (b + c) - (d + e)

- #符号后面是程序的注释
- 指令执行顺序反映了源程序的计算过程
- 指令中可以看到如何使用临时寄存器
- 一个简单的C语言表达式变成多条汇编语句

# 特殊的寄存器x0（zero）

- 0在程序中很常见，拥有一个自己的寄存器（x0）
- x0是一个特殊的寄存器，值恒为0，且不能被改变
  - 注意，在任意指令中，如果使用x0作为目标寄存器，将没有任何效果，仍然保持0不变
- 使用样例

**add x3, x0, x0** # x3=0

**add x1, x2, x0** # x1=x2

**add x0, x0, x0** # nop

0寄存器是“加速经常性事件”的典型实例

# RISC-V 中的立即数

- 数值常数被称为立即数（immediates）
- 立即数有特殊的指令语法：

**opi dst, src, imm**

- 操作码的最后一个字符为**i**的，会将第二个操作数认为是一个立即数（经常用后缀来指明操作数的类型，例如无符号数unsigned的后缀为u）
- 指令举例
  - addi x1, x2, 5 # x1=x2+5**
  - addi x3, x3, 1 # x3=x3+1**



# RISC-V 中的立即数

---

- 问题：在RISC-V中没有立即数减法指令,为什么？
  - 因为如果一个操作可以分解成一个更简单的操作，不要包含它

$f = g - 10$  (in C)

`addi x3, x4, -10` (in RISC-V)

# RISC-V 乘法与除法指令

- **乘法: mul**

- 使用语法: `mul rd, rs1, rs2`
- 功能描述:  $rd = rs1 * rs2$  把寄存器rs2的值乘以寄存器rs1的值, 结果写入寄存器rd, **忽略算术溢出**。

- **除法: div**

- 使用语法: `div rd, rs1, rs2`
- 功能描述:  $rd = rs1 / rs2$  从寄存器rs1的值除以寄存器rs2的值, 向零舍入, 将这些数视为**二进制补码**, 商写入寄存器rd。

# RISC-V 乘法指令

- 积的长度是乘数和被乘数长度的和。两个32位数的积是64位。
- 为了得到有符号或无符号的64位积，RISC-V中有四个指令：
  - 要得到整数32位乘积（64位中的低32位）就用**mul**指令。
  - 要得到高32位：
    - 如果操作数都是有符号数，就用mulh指令；
    - 如果操作数都是无符号数，就用mulhu指令；
    - 如果一个有符号一个无符号，可以用mulhsu指令；

31	25 24	20 19	15 14	12 11	7 6	0	
0000001	rs2	rs1	000	rd	0110011		R mul
0000001	rs2	rs1	001	rd	0110011		R mulh
0000001	rs2	rs1	010	rd	0110011		R mulhsu
0000001	rs2	rs1	011	rd	0110011		R mulhu

# RISC-V 乘法与除法指令——续

---

- 如果需要获得完整的64位值，建议的指令序列为
  - `mulh[[s]u] rdh, rs1, rs2;`
  - `mul rdl, rs1, rs2`
- **注意：**源寄存器必须使用相同的顺序，rdh要注意和rs1和rs2都不相同。这样底层的微体系结构就会把两条指令合并成一次乘法操作，而不是两次乘法操作

# 除法指令举例

# 分别求x6/x7的商和余数。

**div x5, x6, x7    # x5 = x6/x7**

**rem x5, x6, x7    # x5 = x6 mod x7**

0000001	rs2	rs1	100	rd	0110011	R div
0000001	rs2	rs1	101	rd	0110011	R divu
0000001	rs2	rs1	110	rd	0110011	R rem
0000001	rs2	rs1	111	rd	0110011	R remu

# RISC-V指令

---

- 算术运算: add、sub、addi、mul、div
- 逻辑运算: **and**、**or**、**xor**、**andi**、**ori**、**xori**
- 移位操作: sll、srl、sra、slli、srli、srai
- 数据传输: ld、sd、lw、sw、lwu、lh、lhu、sh、lb、lbu、sb、lui
- 比较指令: slt、slti、sltu、sltiu
- 条件分支: beq、bne、blt、bge、bltu、bgeu
- 无条件跳转: jal、jalr

# 逻辑运算指令

语法	功能	描述
and rd, rs1, rs2	$rd = rs1 \ \& \ rs2$	按位“与”操作，结果写入寄存器rd
andi rd, rs1, imm	$rd = rs1 \ \& \ sext(imm)$	把 <b>符号位扩展的立即数</b> 和寄存器rs1的值进行按位与，结果写入寄存器rd
or rd, rs1, rs2	$rd = rs1 \   \ rs2$	按位“或”操作，结果写入寄存器rd
ori rd, rs1, imm	$rd = rs1 \   \ sext(imm)$	把 <b>符号位扩展的立即数</b> 和寄存器rs1的值进行按位或，结果写入寄存器rd
xor rd, rs1, rs2	$rd = rs1 \ \wedge \ rs2$	按位“异或”操作，结果写入寄存器rd
xori rd, rs1, imm	$rd = rs1 \ \wedge \ sext(imm)$	把 <b>符号位扩展的立即数</b> 和寄存器rs1的值进行按位“异或”，结果写入寄存器rd

# 逻辑运算指令实例

**Note:**  $a \rightarrow x1$ ,  $b \rightarrow x2$ ,  $c \rightarrow x3$

Instruction	C	RSC-V
And	$a = b \ \& \ c$	<code>and x1, x2, x3</code>
And Immediate	$a = b \ \& \ 8$	<code>andi x1, x2, 8</code>
Or	$a = b \   \ c$	<code>or x1, x2, x3</code>
Or Immediate	$a = b \   \ 22$	<code>ori x1, x2, 22</code>
Exclusive Or	$a = b \ ^ \ c$	<code>xor x1, x2, x3</code>
Exclusive Or Immediate	$a = b \ ^ \ -5$	<code>xori x1, x2, -5</code>



# 位操作举例

- 逻辑运算

- 寄存器: `and x5, x6, x7`     $\# x5 = x6 \& x7$

- 立即数: `andi x5, x6, 3`     $\# x5 = x6 \& 3$

- RISC-V中**没有NOT**

- 某数的“取反（非）”等价于其和  $1111\dots1111_2$  的xori

- 例: `xori x5, x6, -1`     $\# x5 = \overline{x6}$

# RISC-V指令

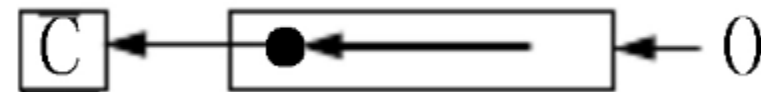
---

- 算术运算: add、sub、addi、mul、div
- 逻辑运算: and、or、xor、andi、ori、xori
- **移位操作: sll、srl、sra、slli、srli、srai**
- 数据传输: ld、sd、lw、sw、lwu、lh、lhu、sh、lb、lbu、sb、lui
- 比较指令: slt、slti、sltu、sltiu
- 条件分支: beq、bne、blt、bge、bltu、bgeu
- 无条件跳转: jal、jalr

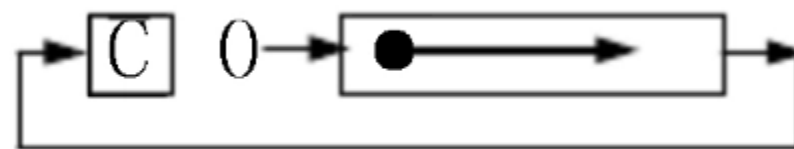
# 移位运算

- 左移相当于乘以2

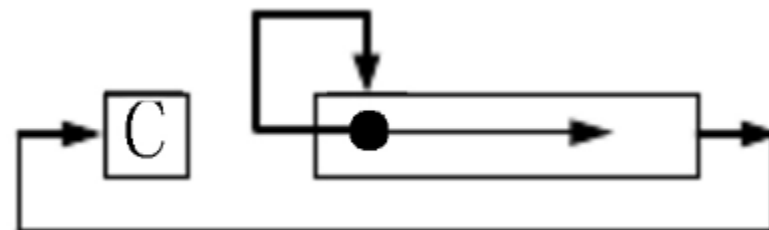
- 左移右边补0
- 左移操作比乘法更快



- 逻辑右移：在最高位添加0




- 算术右移：在最高位添加符号位



➤ **注意：** 移位的位数可以是立即数或者寄存器中的值

# 移位指令

- slli, srli, srai只需要最多移动**63位** (对于RV**64**), 只会使用I类型指令immediate字段低6位的值 

imm[11:0]		rs1	111	rd	0010011	andi
000000	<b>shamt</b>	rs1	001	rd	0010011	slli

Instruction Name	RISC-V
Shift Left Logical	sll rd, rs1, rs2
Shift Left Logical Imm.	slli rd, rs1, shamt
Shift Right Logical	srl rd, rs1, rs2
Shift Right Logical Imm.	srli rd, rs1, shamt
Shift Right Arithmetic	sra rd, rs1, rs2
Shift Right Arithmetic Imm.	srai rd, rs1, shamt

# 移位运算举例

---

- 逻辑移位指令 `sll, slli, srl, srli`

- `slli x11, x12, 2`      # `x11=x12<<2`

- 算术移位指令 `sra, srai`

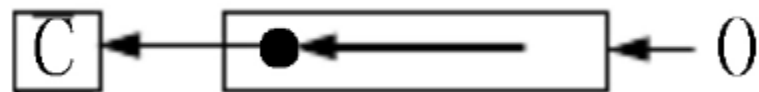
- `srai x10, x10, 4`      # `x10=x10>>4`

- 为什么没有算术左移指令？

# 8位二进制数的补码移位

真值	补码	逻辑左移	算术左移	结果	逻辑右移1位	算术右移1位
127	01111111	11111110	01111110	错误	00111111	同逻辑右移
.....	.....	.....	.....	错误	.....	.....
64	01000000	10000000	00000000	错误	00100000	同逻辑右移
63	00111111	01111110	01111110	正确	00011111	同逻辑右移
.....	.....	.....	.....	正确	.....	.....
1	00000001	00000010	00000010	正确	00000000	同算术右移
-1	11111111	11111110	11111110	正确	01111111	1 <b>1</b> 11111111
.....	.....	.....	.....	正确	0xx.....x	1 <b>1</b> xxx.....x
-64	11000000	10000000	10000000	正确	01100000	1 <b>1</b> 100000
-65	10111111	01111110	11111110	错误	01011111	1 <b>1</b> 011111
-66	10111110	01111100	11111100	错误	01011111	1 <b>1</b> 011111
-128	10000000	00000000	10000000	错误	01000000	1 <b>1</b> 000000

左移



-61的补码

1 1 0 0 0 0 1 1

1 1 0 0 0 0 1 1 0

$[10000110]_{\text{补}} = -122$

+61的补码

0 0 1 1 1 1 0 1

0 1 1 1 1 0 1 0

$[01111010]_{\text{补}} = 122$

-65的补码

1 0 1 1 1 1 1 1

1 0 1 1 1 1 1 1 0

$[01111110]_{\text{补}} = 126$

+65的补码

0 1 0 0 0 0 0 1

1 0 0 0 0 0 1 0

$[10000010]_{\text{补}} = -126$



左移后，“进位”和“首位”一致和不一致？

计算器

程序员

**-61**

0 - 61 =  
**-61**

HEX C3  
DEC -61  
OCT 303  
BIN 1100 0011

BYTE MS

按位 位移位

计算器

程序员

HEX 86  
DEC -122  
OCT 206  
BIN 1000 0110

BYTE MS

按位 位移位

☒ 算术移位  
☐ 逻辑移位  
☐ 旋转循环移位  
☐ 带进位旋转循

>> CE <<

) % ÷

8 9 ×

5 6 -

E 1 2 3 +

F +/- 0 . =

**-61 Lsh 1 =  
-122**

计算器

程序员

HEX 86  
DEC -122  
OCT 206  
BIN 1000 0110

BYTE MS

按位 位移位

☐ 算术移位  
☒ 逻辑移位  
☐ 旋转循环移位  
☐ 带进位旋转循

>> CE <<

) % ÷

8 9 ×

5 6 -

E 1 2 3 +

F +/- 0 . =

**-61 Lsh 1 =  
-122**



# 数据传输指令

---

- 算术运算: add、sub、addi、mul、div
- 逻辑运算: and、or、xor、andi、ori、xori
- 移位操作: sll、srl、sra、slli、srli、srai
- **数据传输: ld、sd、lw、sw、lwu、lh、  
lhu、sh、lb、lbu、sb、lui**
- 比较指令: slt、slti、sltu、sltiu
- 条件分支: beq、bne、blt、bge、bltu、bgeu
- 无条件跳转: jal、jalr

# 数据传输（主存访问指令）

---

- C语言中的变量会映射到寄存器中；而其它大量的数据结构，例如数组会映射到主存中
- 主存可以看作是一维的数组，地址从0开始
- 除数据传输指令外的RISC-V指令只在寄存器中操作
- 数据传输指令专门在寄存器和主存之间传输数据
  - **Store指令：**寄存器的数据存储到主存
  - **Load指令：**主存的数据传输到寄存器

# 数据传输指令的格式

- 数据传输指令的格式

**memop reg, offset(bAddrReg)**

- memop : 操作的名字 (load或者store)
- reg : 寄存器的名字, 源寄存器或者目标寄存器
- bAddrReg : 指向主存的基地址寄存器 (base address register)
- offset : 地址偏移(按字节寻址), 为**立即数** (immediate)

**注意:** 访问的主存地址为 **bAddrReg** + offset  
必须指向一个合法的地址

# 主存的字节寻址方式

- 在现代计算机中操作以一个字节（8bits）为基本单位
  - 不同的体系结构对字（word）的定义不同，RV中1 word = 4 bytes
  - 主存按照**字节（不是字）**进行编址

- 字地址之间有4个字节的距离

- 字的地址为字内最低位字节的地址(**小端模式**)
  - 按字**对齐**，地址最后两位为0（4的倍数）

.....			....	....
高字节	....	....	低字节	12
高字节	.....	....	低字节	8
高字节	....	....	低字节	4
高字节	....	....	低字节	0

- **C语言会自动按照数据类型来计算地址，在汇编中需要程序员自己计算**

# 数据传输指令

- 数据传输指令

- lw reg, offset(bAddrReg) # 从主存 取 字 到寄存器

- sw reg, offset(bAddrReg) # 从寄存器 存 字 到主存

注意：bAddrReg+offset 按字对齐，即4的倍数

—例如一个整型（int）类型的数：32位 = 4字节



如何传输1个双字（8字节）、字符（1字节）或者传输一个short的数据（2字节）？

双字：ld/sd；半字：lh/sh..；字符：lb/sb..

# 数据传输指令举例

- 装入一个字(lw)
- 写出一个字(sw)
- 指令举例 (RISC-V中一个字是32位, int类型是32位)

计算:  $A[10] = A[3] + a$

其中 addr of int A[]  $\rightarrow$  x3, a  $\rightarrow$  x2

lw	<u>x10, 12(x3)</u>	# x10 = A[3]
add	x10, x2, x10	# x10 = A[3] + a
sw	<u>x10, 40(x3)</u>	# A[10] = A[3] + a

# 数据传输指令举例——读

- 读内存指令 (**RISC-V**中一个双字是**64位**, **long long int**类型是**64位**)

`long long int A[100];` (in C)

`g = h + A[3];`

-----  
若 `A[]` -> `x15`, `h` -> `x12`

**`ld x10, 24(x15)`** # `x15`为`A[0]`地址 (in RISC-V ;RV64I)

**`add x11, x12, x10`** # `g = h + A[3]`

■ **基址寻址**: `A[3]`的地址为基址寄存器`x15` + 偏移量24

# 数据传输指令举例——写

- 写内存指令

long long int A[100]; (in C)

A[10] = h + A[3]; 若A[] -> x15, h -> x12

---

ld x10, 24(x15) # x15为A[0]地址 (in RISC-V ;RV64I)

add x10, x12, x10 # h + A[3]赋予寄存器x10

sd x10, 80(x15) # A[10] = h + A[3]



# 主存数据访问指令： 字节/半字/字操作 （ RV64I ）

- 从地址为rs1的主存读字节/半字/字： **符号扩展为64位**存入rd
  - lb rd, offset(rs1) #load byte 对于**RV32**则扩展为**32**位
  - lh rd, offset(rs1) #load half word
  - lw rd, offset(rs1) #load word
- 从主存读一个**无符号**的字节/半字/字： **零扩展为64位**存入rd
  - lbu rd, offset(rs1)
  - lhu rd, offset(rs1)
  - lwu rd, offset(rs1)
- 将一个字节/半字/字写入主存： 保存**最右端的8/16/32位**
  - sb rs2, offset(rs1) # 保存**最右端的8位**（低8位）
  - sh rs2, offset(rs1) # 保存**最右端的16位**（低16位）
  - sw rs2, offset(rs1) # 保存**最右端的32位**（低32位）

# 传输一个字节数据

- ▶ 1) 使用**字类型**指令，配合**位的掩码**来达到目的

```
lw    x11, 0(x1)
```

```
andi  x11, x11, 0xFF    # lowest byte
```

- ▶ 2) 使用**字节**传输指令

```
lb     x11, 1(x1)
```

```
sb     x11, 0(x1)
```

- ◆ **注意：**字节传输指令无需字对齐

# 字节排布

- ▶ 大端：最高的字节在最低的地址，字的地址等于最高字节的地址
- ▶ 小端：最低的字节在最低的地址，字的地址等于最低字节的地址

`mem[x1] = 0x00000180` #地址`mem[x1]` 存的数据为`0x00000180`

高地址  低地址

80	01	00	00
----	----	----	----

大端模式 (Big Endian)

00	00	01	80
----	----	----	----

小端模式 ( Little Endian )

**RISC-V 是小端模式**

使用的寄存器是32位 (RV32I) , 若 $\text{mem}[\text{x1}] = 0\text{x}12000180$ , 执行指令 `lb x11, 1(x1)` , 寄存器x11的数据为:

- ☒ A `0x00000001`
- ☐ B `0x00000008`
- ☐ C `0x00000000`

使用的寄存器是32位 (RV32I) , 若 $\text{mem}[\text{x1}] = 0\text{x}12000180$ , 执行指令 `lb x12, 0(x1)` , 寄存器x12的数据为:

- ☒ A `0xFFFFFFFF80`
- ☐ B `0x00000080`
- ☐ C `0x00000008`
- ☐ D `0x00000000`

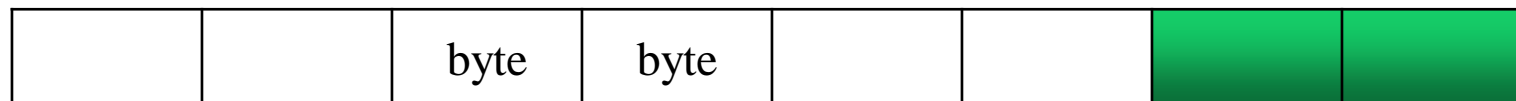
使用的寄存器是 32 位（RV32I），若  $\text{mem}[x1] = 0x12000180$ ，执行如下指令后  $\text{mem}[x1]$  的数据为：

```
lb x12, 0(x1)
sb x12, 2(x1)
```

- ☒ A 0x12800180
- ☐ B 0x12008080
- ☐ C 0x80120080
- ☐ D 0x12000180

# 半字数据传输指令

- **lh** reg, off(bAddr) #load half word
- **sh** reg, off(bAddr) #store half word
- **lhu**、**shu** 无符号半字传输指令
  - off(bAddr)应该是2的倍数（对齐）
  - sh指令中高16位(RV32)/高48(RV64)**忽略**
  - lh指令中高16位(RV32)/高48(RV64)用**符号扩展**
- **lbu**、**lhu**指令：高位都以0扩展

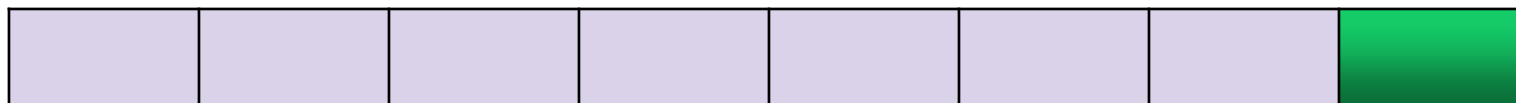


# 字节数据传输指令

- lb/sb 读/写低字节

- sb指令：高56位(RV64)/24位(RV32)被忽略

- lb指令：高56位(RV64)/24位(RV32)用符号扩展



- 例如：mem[x1]中存储的数据为0x1020304050607080

(RISC V) lb x11, 1(x1) #x11=0x000000000000000070

lb x12, 0(x1) #x12=0xFFFFFFFFFFFFFFFF80

sb x12, 2(x1) #mem[x1]=0x1020304050807080



# 比较指令

---

- 算术运算: add、sub、addi、mul、div
- 逻辑运算: and、or、xor、andi、ori、xori
- 移位操作: sll、srl、sra、slli、srli、srai
- 数据传输: ld、sd、lw、sw、lwu、lh、lhu、sh、lb、lbu、sb、lui
- **比较指令: slt、slti、sltu、sltiu**
- 条件分支: beq、bne、blt、bge、bltu、bgeu
- 无条件跳转: jal、jalr

# 有符号数比较指令slt、slti

- **Set Less Than** (slt: 通过判断两个操作数的大小来对目标寄存器进行设置, **有符号比较**)
  - `slt dst, reg1, reg2`
  - 如果reg1中的值 < reg2中的值, 则dst = 1, 否则dst = 0
- **Set Less Than Immediate** (slti)
  - `slti dst, reg1, imm`
  - 如果reg1中的值 < 立即数imm, 则dst=1, 否则dst=0

# 无符号数比较指令sltu、sltiu

- slt和slti的无符号版本 (按照无符号数比较):

**sltu** dst, reg1, reg2 #unsigned comparison

**sltiu** dst, reg1, imm #unsigned comparison against constant

- 例:

```
addi    x10, x0, -1
```

```
slti    x11, x10, 1
```

```
sltiu   x12, x10, 1
```

执行 `addi x10, x0, -1` (in RV32)  
x10=?

- ☒ A 0xFFFFFFFF
- ☐ B 0x80000001
- ☐ C 0x00000001
- ☐ D 0x10000001

提交

执行 `addi x10, x0, -1`  
`slti x11, x10, 1`  
`x11 = ?`

(RV32)

- ☒ A 0x00000001
- ☐ B 0x00000000
- ☐ C 0xFFFFFFFF
- ☐ D 0x10000001

提交

执行 `addi x10,x0,-1`

`sltiu x12,x10,1`

(RV32)

`x12=?`

A

`0x00000000`

B

`0x00000001`

C

`0xFFFFFFFF`

D

`0x10000001`

提交

# 条件分支指令

---

- 算术运算: add、sub、addi、mul、div
- 逻辑运算: and、or、xor、andi、ori、xori
- 移位操作: sll、srl、sra、slli、srli、srai
- 数据传输: ld、sd、lw、sw、lwu、lh、lhu、sh、lb、lbu、sb、lui
- 比较指令: slt、slti、sltu、sltiu
- **条件分支: beq、bne、blt、bge、bltu、bgeu**
- 无条件跳转: jal、jalr

# 条件分支（跳转）指令

---

- C语言中有控制流
  - 比较语句/逻辑语句确定下一步执行的语句块
- RISC-V 汇编无法定义语句块，但是可以通过标记（Label）的方式来定义语句块起始
  - 标记后面加一个冒号（main:）
  - 汇编的控制流：跳转到标记的位置
  - 在C语言中也有类似的结构，但是被认为是坏的编程风格（C语言有goto语句，跳转到标记所在的位置）



# 条件分支（跳转）指令

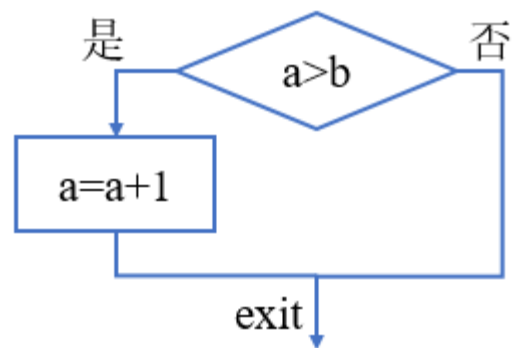
- `beq reg1, reg2, label` #branch if **e**qual
  - 如果`reg1`中的值=`reg2`中的值, 程序跳转到`label`处继续执行
- `bne reg1, reg2, label` #branch if **n**ot **e**qual
  - 如果`reg1`**不等于**`reg2`, 程序跳转到`label`处继续执行
- `blt reg1, reg2, label` #branch if **l**ess **t**han
  - 如果`reg1` < `reg2`, 程序跳转到`label`处继续执行
- `bge reg1, reg2, label` #branch if **g**reater than or **e**qual
  - 如果`reg1` >= `reg2`, 程序跳转到`label`处继续执行

注意: ◆ 条件为真则转到标签（立即数）所指的语句执行，否则按序执行。  
◆ **没有依据标志位**的跳转（与x86不同）

# 条件分支指令的例子

if (a > b) a += 1;

.....



(C语言)

假设：a保存在x22，b保存在x23

(RISC-V)

bge x23, x22, Exit

# 当b >= a时跳转

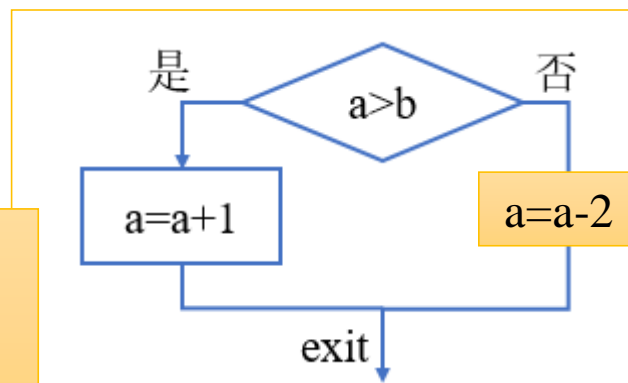
addi x22, x22, 1

# a=a+1

Exit: ....



多一条else a=a-2后，  
汇编代码怎么写？



# 分支转移的条件

---

根据比较结果更改指令执行顺序

- **beq**: **b**ranch if **e**qual
- **bne**: **b**ranch if **n**ot **e**qual
- **blt** : **b**ranch if **l**ess **t**han
- **bge**: **b**ranch if **g**reater than or **e**qual
- 无符号版本命令有: **bltu**、**bgeu**

# RISC-V 中的有符号与无符号

---

- 有符号和无符号在3个上下文环境中
  - Signed vs. unsigned bit extension 位扩展
    - `lb, lh` # 用符号扩展
    - `lbu, lhu` # 用0扩展
  - Signed vs. unsigned comparison 比较
    - `slt, slti`
    - `sltu, sltiu`
  - Signed vs. unsigned branch 条件分支
    - `blt, bge`
    - `bltu, bgeu`

# 无条件跳转指令

---

- 算术运算: add、sub、addi、mul、div
- 逻辑运算: and、or、xor、andi、ori、xori
- 移位操作: sll、srl、sra、slli、srli、srai
- 数据传输: ld、sd、lw、sw、lwu、lh、lhu、sh、lb、lbu、sb、lui
- 比较指令: slt、slti、sltu、sltiu
- 条件分支: beq、bne、blt、bge、bltu、bgeu
- 无条件跳转: **jal、jalr**

# 32个RISC-V寄存器

寄存器	助记符	注解
<b>x0</b>	<b>zero</b>	<b>固定值为0</b>
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

# 无条件跳转指令

- **jal rd, Label** # (**jump and link**)
  - 将下一条指令的地址PC+4保存在寄存器rd(一般用x1)
  - 跳转到目标地址**Label** (立即数偏移量, 有规定的转换规则)
  - 可用于函数调用
  - 伪指令**j Label**, 原型是**jal rd, Label**
- **jalr rd, offset(rs1)** #(**jump and link register**)
  - 类似jal, 但跳转到rs1+offset
  - 把PC+4存到rd中, 如果rd是x1, 则PC+4就是下一条指令的地址
  - 可以用于函数返回

Text Segment				
Bkpt	Address	Code	Basic	Source
<input type="checkbox"/>	0x00400000	0x00000513	addi x10,x0,0	1: addi x10, x0, 0
<input type="checkbox"/>	0x00400004	0x00a50513	addi x10,x10,10	2: addi x10, x10, 10
<input type="checkbox"/>	0x00400008	0xff810113	addi x2,x2,0xffffffff8	4: addi sp, sp, -8
<input type="checkbox"/>	0x0040000c	0x00112223	sw x1,4(x2)	5: sw x1,4(sp)
<input type="checkbox"/>	0x00400010	0x00a12023	sw x10,0(x2)	6: sw x10,0(sp)
<input type="checkbox"/>	0x00400014	0xfff50293	addi x5,x10,0xffffffff	7: addi x5, x10, -1
<input type="checkbox"/>	0x00400018	0x0002d863	bge x5,x0,0x00000010	8: bge x5, x0, L1
<input type="checkbox"/>	0x0040001c	0x00100513	addi x10,x0,1	9: addi x10, x0, 1
<input type="checkbox"/>	0x00400020	0x00810113	addi x2,x2,8	10: addi sp, sp, 8
<input type="checkbox"/>	0x00400024	0x00008067	jalr x0,x1,0	11: jalr x0,0(x1)
<input type="checkbox"/>	0x00400028	0xfff50513	addi x10,x10,0xffffffff	13: addi x10, x10, -1
<input type="checkbox"/>	0x0040002c	0xfddff0ef	jal x1,0xfffffddc	14: jal x1, fact
<input type="checkbox"/>	0x00400030	0x00050313	addi x6,x10,0	15: addi x6, x10, 0
<input type="checkbox"/>	0x00400034	0x00012503	lw x10,0(x2)	16: lw x10,0(sp)
<input type="checkbox"/>	0x00400038	0x00412083	lw x1,4(x2)	17: lw x1,4(sp)
<input type="checkbox"/>	0x0040003c	0x00810113	addi x2,x2,8	18: addi sp, sp, 8
<input type="checkbox"/>	0x00400040	0x02650533	mul x10,x10,x6	19: mul x10, x10, x6
<input type="checkbox"/>	0x00400044	0x00008067	jalr x0,x1,0	20: jalr x0,0(x1)

# RISC-V汇编中的伪指令

- 方便程序员编程
- 非直接硬件实现的指令，需要通过汇编器翻译为实际的硬件指令
- 例： **mv dst, src**  
没有专门的硬件来实现数据移动指令——>  
**addi dst, src, 0 或者 add dst, src, x0**
- li, la, j, jr, neg....



# 其它的伪指令

---

- **Load Immediate (li)** 装入一个立即数
  - li dst, imm
  - 装入一个立即数到 dst寄存器
  - 被翻译为: addi dst, x0, imm
- **Load Address (la)** 装入一个地址
  - la dst, label
  - 装入由Label指定的地址到 dst
- 指令手册

# 伪指令

## PSEUDO INSTRUCTIONS

MNEMONIC	NAME	DESCRIPTION	USES
beqz	Branch = zero	if(R[rs1]==0) PC=PC+{imm,1b'0}	beq
bnez	Branch ≠ zero	if(R[rs1]!=0) PC=PC+{imm,1b'0}	bne
fabs.s, fabs.d	Absolute Value	$F[rd] = (F[rs1] < 0) ? -F[rs1] : F[rs1]$	fsgnx
fmv.s, fmv.d	FP Move	$F[rd] = F[rs1]$	fsgnj
fneg.s, fneg.d	FP negate	$F[rd] = -F[rs1]$	fsgnjn
j	Jump	PC = {imm,1b'0}	jal
jr	Jump register	PC = R[rs1]	jalr
la	Load address	R[rd] = address	auipc
li	Load imm	R[rd] = imm	addi
mv	Move	R[rd] = R[rs1]	addi
neg	Negate	$R[rd] = -R[rs1]$	sub
nop	No operation	R[0] = R[0]	addi
not	Not	$R[rd] = \sim R[rs1]$	xori
ret	Return	PC = R[1]	jalr
seqz	Set = zero	$R[rd] = (R[rs1] == 0) ? 1 : 0$	sltiu
snez	Set ≠ zero	$R[rd] = (R[rs1] != 0) ? 1 : 0$	sltu

# 伪指令实现

Pseudo	Real
<code>nop</code>	<code>addi x0, x0, 0</code>
<code>not rd, rs</code>	<code>xori rd, rs, -1</code>
<code>beqz rs, offset</code>	<code>beq rs, x0, offset</code>
<code>bgt rs, rt, offset</code>	<code>blt rt, rs, offset</code>
<code>j offset</code>	<code>jal x0, offset</code>
<code>ret</code>	<code>jalr x0, x1, offset</code>
<code>call offset</code> (if too big for just a <code>jal</code> )	<code>auipc x6, offset[31:12]</code> <code>jalr x1, x6, offset[11:0]</code>
<code>tail offset</code> (if too far for a <code>j</code> )	<code>auipc x6, offset[31:12]</code> <code>jalr x0, x6, offset[11:0]</code>

# 伪指令便于编程

Basic	Source
lui x6, 0x00010203	10: li t1, 0x1020304050607080 #64位立即数装入寄存器t1(x6)
addiw x6, x6, 0x00000040	
slli x6, x6, 11	
addi x6, x6, 0x00000283	
slli x6, x6, 11	
addi x6, x6, 28	
slli x6, x6, 10	
addi x6, x6, 0x00000080	
sub x7, x0, x6	11: neg t2, t1

Text Segment				
Bkpt	Address	Code	Basic	Source
<input type="checkbox"/>	0x00400000	0x10203337	lui x6, 0x00010203	1: lui x6, 0x00010203
<input type="checkbox"/>	0x00400004	0x000103b7	lui x7, 16	2: li t2, 0x00010203
<input type="checkbox"/>	0x00400008	0x20338393	addi x7, x7, 0x00000203	
<input type="checkbox"/>	0x0040000c	0x04330313	addi x6, x6, 0x00000043	3: addi x6, x6, 0x00000043
<input type="checkbox"/>	0x00400010	0x00331313	slli x6, x6, 3	4: slli x6, x6, 3
<input type="checkbox"/>	0x00400014	0x01c30313	addi x6, x6, 28	5: addi x6, x6, 28
<input type="checkbox"/>	0x00400018	0x406003b3	sub x7, x0, x6	6: neg t2, t1
<input type="checkbox"/>	0x0040001c	0x006002b3	add x5, x0, x6	7: mv x5, x6

Name	Number	Value
zero	0	0x00000000
ra	1	0x00000000
sp	2	0x7fffffc
gp	3	0x10008000
tp	4	0x00000000
t0	5	0x81018234
t1	6	0x81018234
t2	7	0x7efe7dcc
s0	8	0x00000000
s1	9	0x00000000
a0	10	0x00000000
a1	11	0x00000000
a2	12	0x00000000
a3	13	0x00000000
a4	14	0x00000000
a5	15	0x00000000
a6	16	0x00000000
a7	17	0x00000000
s2	18	0x00000000
s3	19	0x00000000
s4	20	0x00000000
s5	21	0x00000000
s6	22	0x00000000

# 伪指令 vs. 硬件指令

---

- 硬件指令
  - 所有指令都是在硬件中直接实现了，可以直接执行。
- 伪指令
  - 方便汇编语言程序员使用的指令
  - 每一条伪指令会被翻译为1条或者多条硬件指令

# 条件判断分支转移指令

## if-else语句

$f \rightarrow x10, g \rightarrow x11, h \rightarrow x12, i \rightarrow x13, j \rightarrow x14$

if ( $i == j$ )

$f = g + h;$

else

$f = g - h;$

end

...



bne x13, x14, Else

add x10, x11, x12

j Exit

Else: sub x10, x11, x12

Exit:



Edit Execute

riscv2.asm riscv3.asm riscv4.asm

```
1 addi x10, x0, 3
2 addi x11, x0, 5
3 addi x12, x0, 7
4 addi x13, x0, 8
5 addi x14, x0, 9
```

```
7 bne x13, x14, Else
8 add x10, x11, x12
9 j Exit
10 Else: sub x10, x11, x12
11 Exit:
```

Assemble F3

Assemble the current file and clear

Step F7

Backstep F8

Pause F9

Stop F11

Reset F12

Clear all breakpoints Ctrl-K

Toggle all breakpoints Ctrl-T

Edit

riscv2.asm

```
1 addi x10, x0, 3
2 addi x11, x0, 5
3 addi x12, x0, 7
4 addi x13, x0, 8
5 addi x14, x0, 9
6
7 bne x13, x14, Else
```

Run speed at max (no interaction)

Text Segment

Bkpt	Address	Code	Basic
<input type="checkbox"/>	0x00400000	0x00300513	addi x10, x0, 3
<input type="checkbox"/>	0x00400004	0x00500593	addi x11, x0, 5
<input type="checkbox"/>	0x00400008	0x00700613	addi x12, x0, 7
<input type="checkbox"/>	0x0040000c	0x00800693	addi x13, x0, 8
<input type="checkbox"/>	0x00400010	0x00900713	addi x14, x0, 9
<input type="checkbox"/>	0x00400014	0x00e69663	bne x13, x14, 0x0000000c
<input type="checkbox"/>	0x00400018	0x00c58533	add x10, x11, x12
<input type="checkbox"/>	0x0040001c	0x0080006f	jal x0, 0x00000008
<input type="checkbox"/>	0x00400020	0x40c58533	sub x10, x11, x12

Floating Point

Control and Status

Registers

Name	Number	Value
zero	0	0x00000000
ra	1	0x00000000
sp	2	0x7ffffefc
gp	3	0x10008000
tp	4	0x00000000
t0	5	0x00000000
t1	6	0x00000000
t2	7	0x00000000
s0	8	0x00000000
s1	9	0x00000000
a0	10	0xffffffff
a1	11	0x00000000
a2	12	0x00000007
a3	13	0x00000008
a4	14	0x00000009
a5	15	0x00000000
a6	16	0x00000000
a7	17	0x00000000
s2	18	0x00000000
s3	19	0x00000000
s4	20	0x00000000
s5	21	0x00000000
s6	22	0x00000000
s7	23	0x00000000
s8	24	0x00000000
s9	25	0x00000000
s10	26	0x00000000
s11	27	0x00000000
t3	28	0x00000000
t4	29	0x00000000
t5	30	0x00000000
t6	31	0x00000000
pc		0x00400028

# 循环结构

```
• long long int A[20];  
  long long int sum = 0;  
  for (long long int i = 0; i < 20; i++)  
      sum += A[i];
```



数组基地址保存在 x8.

```
add x9, x8, x0    # x9=&A[0]  
add x10, x0, x0   # sum=0  
add x11, x0, x0   # i=0  
addi x13, x0, 20  # x13=20
```

Loop:

```
bge x11, x13, Done  
ld x12, 0(x9)     # x12=A[i]  
add x10, x10, x12  
addi x9, x9, 8    # &A[i+1]  
addi x11, x11, 1  # i++  
j Loop
```

Done:



# 循环结构

- C代码:

```
while (save[i] == k) i += 1;
```

i存储在x22中，k存储在x24中，

数组save的地址保存在x25中。

数组save为long long int



- RISC-V代码:

```
Loop: slli x10, x22, 3 # i<<3=8i
```

```
add x10, x10, x25 #x10=&save[i]
```

```
ld x9, 0(x10) #x9=save[i]
```

```
bne x9, x24, Exit # save[i] = k?
```

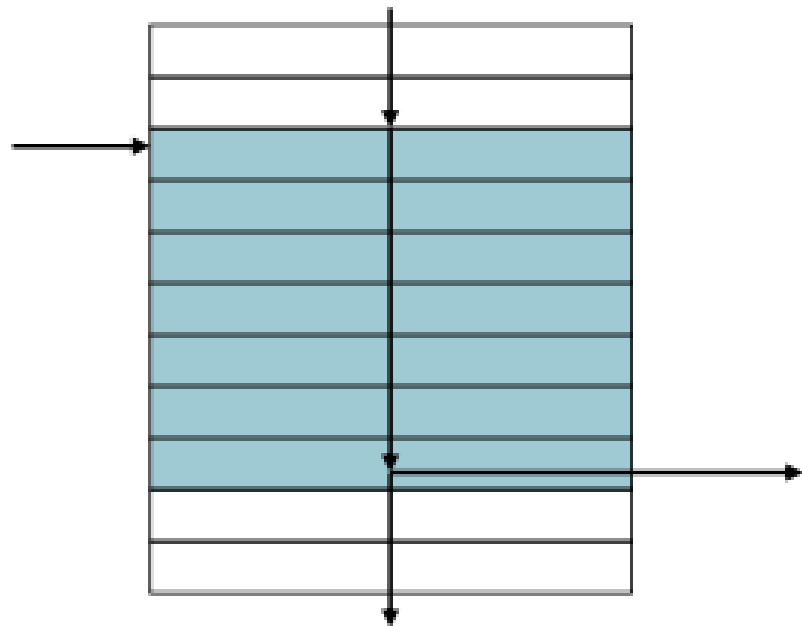
```
addi x22, x22, 1 #i=i+1
```

```
beq x0, x0, Loop # j Loop
```

```
Exit: ...
```

# 基本块

- 基本块是这样的指令序列
  - 没有嵌入分支（除非在末尾）
  - 没有分支目标/标签（除非在开头）



- 编译器可以识别基本块以进行优化
- 先进的处理器能够加速基本块的执行

# 第二章 RISC-V汇编及其指令系统

---

- **RISC-V汇编语言**

- 汇编语言简介
- RISC-V汇编指令概览
- RISC-V常用汇编指令
- 函数调用及栈的使用
- 编译工具介绍

# 函数调用

---

- 函数（过程）调用的6个步骤
  - 将参数放在过程函数(被调用者)可以访问到的位置（x10~x17）
  - 将控制转交给过程函数（被调用者）
  - 获取过程函数（被调用者）所需的存储资源
  - 执行过程函数（被调用者）中的操作
  - 将结果值放在调用程序（调用者）可以访问到的寄存器
  - 返回调用位置（x1中保存的地址）

# 函数调用

---

- 调用子程序包含两个参与者

- 调用者 (caller)

- ◆ 准备函数参数，跳转到被调用者子程序

- 被调用者 (callee)

- ◆ 使用调用者提供的参数，然后运行

- ◆ 运行结束保存返回值

- ◆ 将控制（如跳回）还给调用者。

# 函数调用指令

---

- 过程调用：跳转和链接

## **jal x1, Label**

- 将下一条指令的地址保存在x1中
- 跳转到**Label**（Label对应指令的目标地址）

- 过程返回：寄存器跳转和链接

## **jalr x0, 0(x1)**

- 类似jal，但跳转到 $0 + x1$ 中保存的地址
- 把x0用作目的寄存器（实际x0不会被改变）
- 同样可用于跳转到计算出的位置

# 函数调用

- C语言函数调用

- `int function(int a ,int b)     # a, b: s0, s1`
- `{ return (a+b); }`

- RISC-V过程调用中的常用寄存器

- 返回地址寄存器     x1(ra)
- 参数寄存器     x10-x17
- 返回值寄存器     x10-x11
- 保存寄存器     x8,x9,x18-x27
- 临时寄存器     x5-x7,x28-x31
- 栈指针     x2(sp)

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/ <u>fp</u>	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

```
int function(int a ,int b) # a, b: s0, s1
{ return (a+b); }
```

# 函数调用

## • RISC-V函数调用

1000 mv a0, s0

1004 mv a1, s1

1008 addi ra, zero, 1016 # ra/x1 = 1016

1012 j sum # jump to sum

1016 ... # 下个指令

...

2000 sum: add a0, a0, a1 # a0=a0+a1

2004 jr ra # jump register; jalr zero, 0(ra)

- 调用sum函数必须能够以某种方式返回。
- sum可能会被很多地方调用，所以不能返回到一个固定地点。

为什么要使用jr而不使用j?





# 函数调用

- RISC-V函数调用

1000 mv a0, s0

1004 mv a1, s1

1008 addi ra, zero, 1016 # ra = 1016

1012 j sum # jump to sum

1016 ... # 下个指令

...

2000 sum: add a0, a0, a1

2004 **jr** ra # jump register

ra=?

1008 **jal** ra, sum # ra = 1012, jump to sum

# 函数调用

## • RISC-V函数调用

1000 mv a0, s0

1004 mv a1, s1

1008 addi ra, zero, 1016 # ra = 1016

1012 j sum # jump to sum

1016 ... # 下个指令

...

2000 sum: add a0, a0, a1

2004 jr ra # jump register

jal rd, **Label** # (**j**ump **a**nd **l**ink)

- 将下一条指令的地址PC+4保存在寄存器rd(一般用x1)



为什么ra/x1 = 1012?



1008 jal ra, sum # ra = 1012, jump to sum

# RISC-V中函数调用要考虑的问题

---

- ◆ 高级语言函数体中一般使用局部变量
- ◆ RISC-V汇编子程序使用寄存器（全局变量）
- ◆ 调用者(Caller)和被调用者(Callee)可能使用相同寄存器
  - 造成数据破坏
  - 被调用函数需要保存可能被破坏的寄存器（现场）
  - 哪些寄存器属于现场？

# 例：函数调用嵌套问题

- 函数调用嵌套？递归？

```
main(){  
    ...;  
    numSquare(9,10);  
    ...;}
```

```
int numSquare (int x, int y){  
    return mult (x, x)+y;  
}
```

- main调用函数numSquare，寄存器ra保存返回地址“ad1”
- 函数numSquare调用函数mult，寄存器ra需要保存返回地址“ad2”，“ad1”被覆盖

调用mult前需使用**栈（stack）**保存返回地址ad1

# 函数调用需保存的数据

## □ 函数调用需要保存哪些（寄存器）数据？

- 需要保存的参数/save寄存器的值。
- 函数执行完成后，返回下一条“指令”的地址。
- 其他局部变量（例如函数中使用的数组或结构体）的空间

## □ RISC-V函数调用约定将寄存器分为两类：

- 保留 sp, gp, tp, x1(ra), x8-x9(s0-s1), x18-x27(s2-s11)
- 不保留 x10-x17(a0-a7), x5-x7(t0-t2), x28-x31(t3-t6)

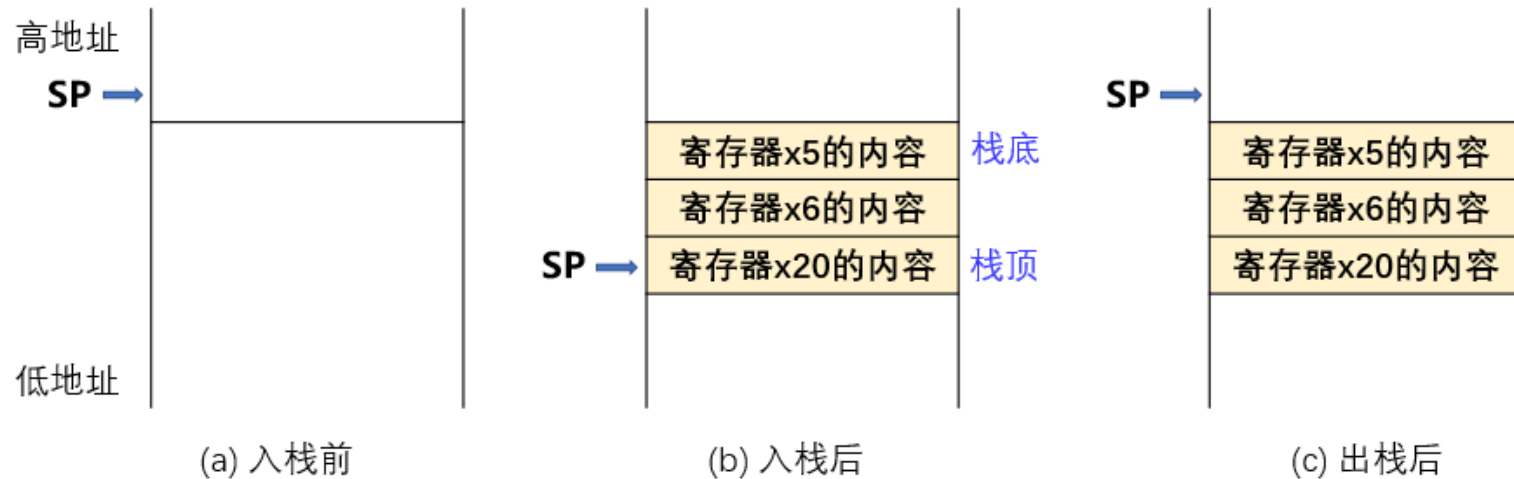
# 栈的使用

---

- 在调用函数之前需要一个保存旧值的地方，在返回时还原它们。在RISC-V中使用栈来完成这个功能。
- 栈：后进先出（LIFO：Last In First Out）队列
  - 入栈：将数据放入栈
  - 出栈：从栈中取出数据

# 函数调用时栈的变化

- 下图展示了在函数调用之前、之中和之后栈的情况



- 按**历史惯例**，RISC-V中栈按照从高到低的地址顺序增长（即栈底位于高地址，栈顶位于低地址）。
- sp（x2寄存器）是RISC-V中的**栈指针**寄存器，用来保存栈顶的地址。
- 入栈**向下移动栈指针(减 sp), **出栈**向上移动栈指针(增 sp)。

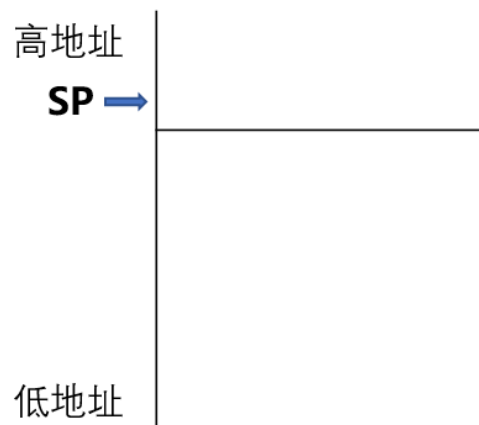
# 栈的使用

---

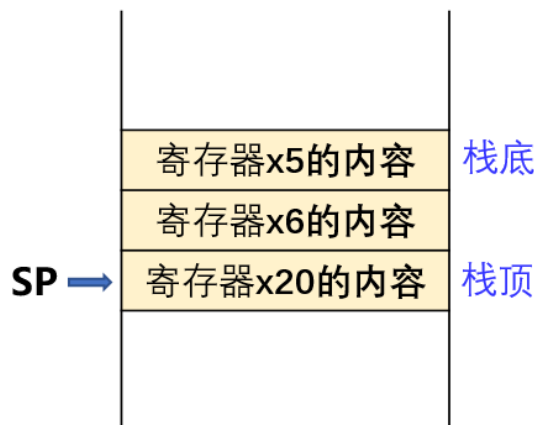
- 寄存器sp总是指向栈中最后使用的空间
- 在使用stack，首先将这个指针减少所需的空间量，然后填入需要保存的信息。
- 例如，将保存寄存器x9(RV-32)的内容入栈：
  - `addi sp, sp, -4` # 将栈指针sp向低地址移动4个字节(RV-32)
  - `sw x9, 0(sp)` # 将寄存器x9中的内容写入栈



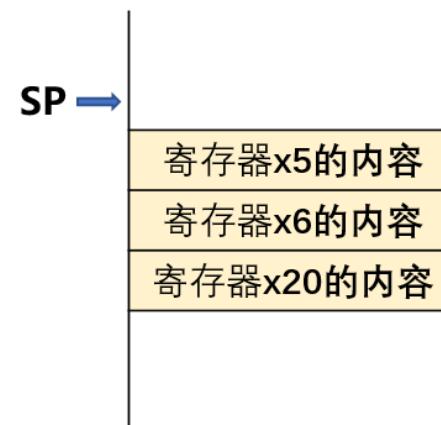
# 栈的使用



(a) 进栈前



(b) 进栈后



(c) 出栈后

In RISC-64

数据“入”栈

```
addi sp, sp, -24
sd x5, 16(sp)
sd x6, 8(sp)
sd x20, 0(sp)
```

数据“出”栈

```
ld x20, 0(sp)
ld x6, 8(sp)
ld x5, 16(sp)
addi sp, sp, 24
```

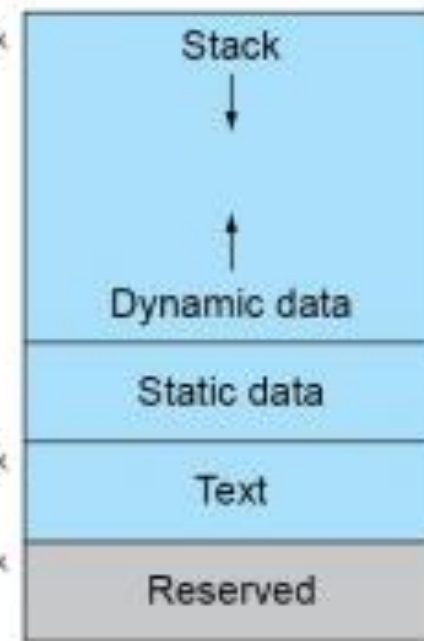
# 内存布局

- 代码区：程序代码
- 静态数据区：全局变量
  - 例如C中的静态变量、常数组和字符串
  - x3（全局指针）初始化为适当的地址，加上正负偏移量可以访问这段内存空间
- 动态数据区：堆
  - 例如C中的malloc、Java中的new
- 栈区：自动存储

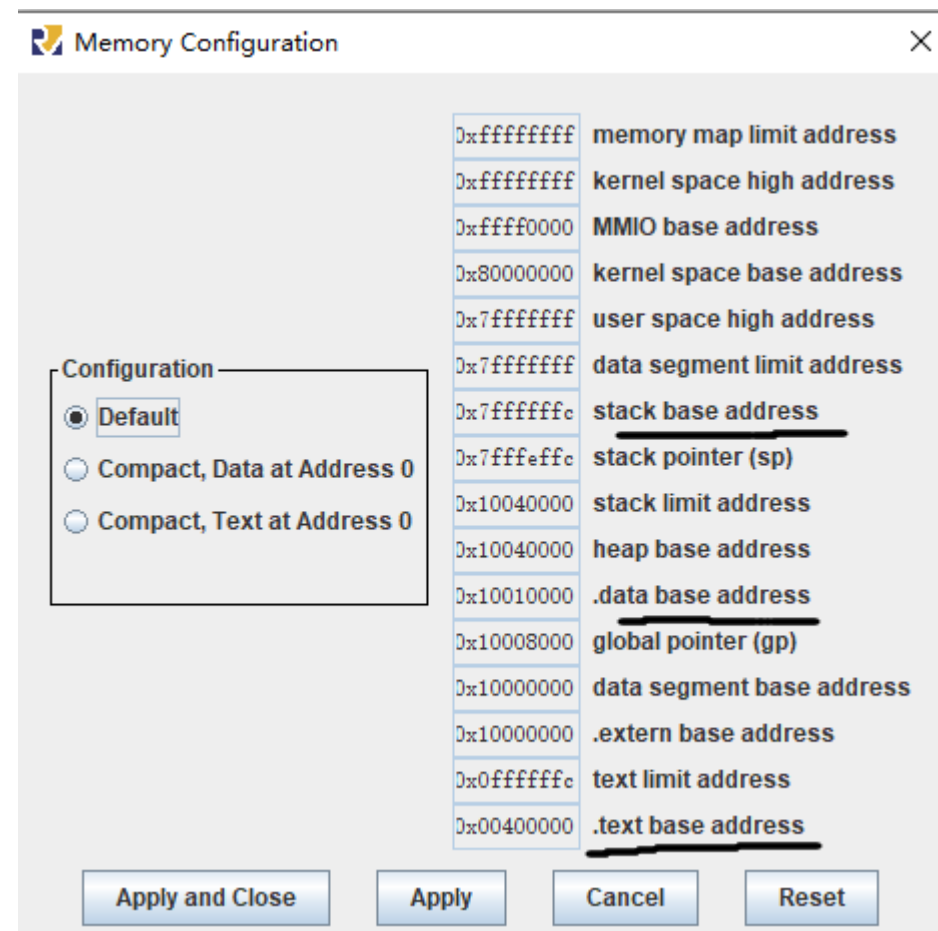
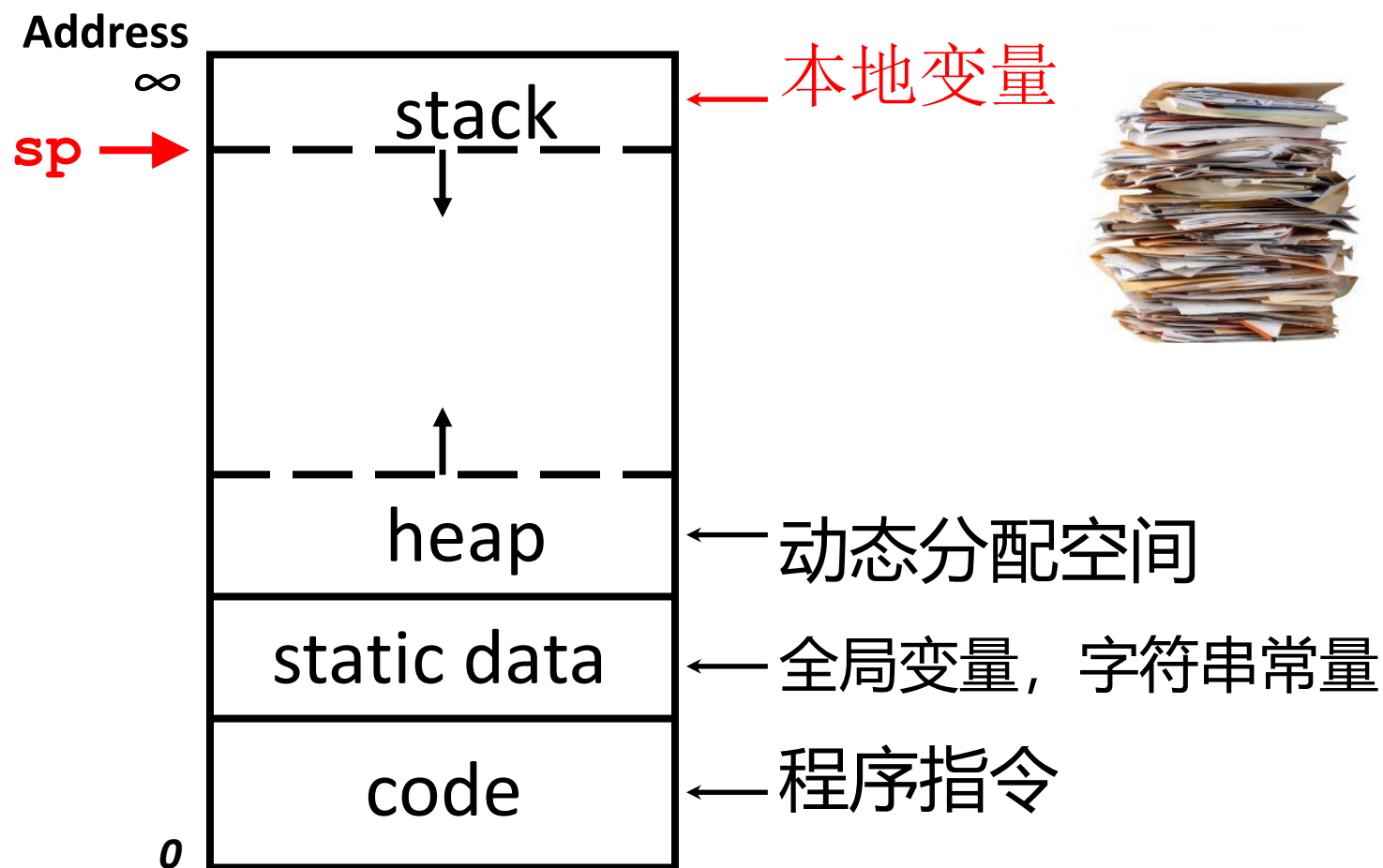
SP → 0000 003f ffff fff0<sub>hex</sub>

0000 0000 1000 0000<sub>hex</sub>

PC → 0000 0000 0040 0000<sub>hex</sub>



# 数据的内存排布情况



# 练习：字符串复制

- C代码：（以null字符结束的字符串）

```
void strcpy (char x[], char y[])  
{  
    int i;  
    i = 0;  
    while ((x[i]=y[i])!='\0')  
        i += 1;  
}
```

- 用x10保存x[ ]的基址， x11保存y[ ]的基址， x19保存i

# RV64

• C代码：（以null字符结束的字符串）

```
void strcpy (char x[], char y[])
```

```
{  
    int i;  
    i = 0;  
    while ((x[i]=y[i])!='\0')  
        i += 1;  
}
```

• 用x10保存x[ ]的基址，x11保存y[ ]的基址，x19保存i

- char: 8位，一个字符用一个字节表示
- x19是保存寄存器，可能含有主程序需要用到的值，需入栈保留

```
strcpy: addi sp, sp, -8    # 调整栈，留出1个双字的空间  
sd x19, 0(sp)            # x19入栈 保护调用函数的数据  
add x19, x0, x0           # i=0  
L1: add x5, x19, x11       # x5 = y[i]的地址  
lbu x6, 0(x5)             # x6 = y[i]  
add x7, x19, x10          # x7 = x[i]的地址  
sb x6, 0(x7)              # x[i] = y[i]  
beq x6, x0, L2            # 若y[i] == 0，则退出  
addi x19, x19, 1          # i = i + 1  
jal x0, L1                # 下一次loop迭代  
L2: ld x19, 0(sp)         # 恢复x19原值  
addi sp, sp, 8            # 从栈中弹出1个双字  
jalr x0, 0(x1)            # 返回主函数
```

# 练习：函数嵌套

```
int numSquare (int x, int y){  
    return mult (x, x)+y; }  
}
```

- x -> a0, y -> a1
- mult函数，假设
  - mult的入口参数在a0和a1两个寄存器中。
  - mult的计算结果，由寄存器a0返回。

• sumSquare:

```
push    addi sp, sp, -8    # space on stack  
        sw ra, 4(sp)      # save return addr  
        sw a1, 0(sp)      # save y  
        mv a1, a0         # a1=a0 (a0=x)  
        jal ra, mult      # call mult  
        lw a1, 0(sp)      # restore y  
        add a0, a0, a1     # mult()+y  
pop     lw ra, 4(sp)      # get ret addr  
        addi sp, sp, 8    # restore stack  
        jr ra             # return to caller function
```

mult:...

# 例：用栈进行递归

---

- C代码： `long long int fact (long long int n)`

{

`if (n < 1) return 1;`

`else return n * fact(n - 1);`

}

- 参数n在x10中； 结果在x10中

# RV汇编用栈实现递归求阶乘fact

```
fact: addi sp, sp, -16 # 留出2个双字的栈空间
      sd x1, 8(sp)      # 返回地址保存到栈中
      sd x10, 0(sp)     # 参数n保存到栈中
      addi x5, x10, -1
      bge x5, x0, L1    # 若参数n-1 >= 0, 则跳转到L1
      addi x10, x0, 1   # 若n-1 < 0, 返回值1并存入x10
      addi sp, sp, 16   # 释放2个字的栈空间, 不用恢复
      jalr x0, 0(x1)    # 返回fact的调用者
L1:   addi x10, x10, -1 # n = n - 1
      jal x1, fact      # 调用fact(n-1), 重复直到fact(0).
Calc: addi t1, x10, 0   # 从fact(0)开始, 结果存到t1
      ld x10, 0(sp)    # 恢复调用者的参数
      ld x1, 8(sp)     # 恢复调用者的返回地址
      addi sp, sp, 16   # 释放2个字的栈空间
      mul x10, x10, t1  # 计算m* fact(m-1) (m从1开始)
      jalr x0, 0(x1)   # 返回Calc
```



# RV汇编用栈实现递归求阶乘fact(2)

```
fact: addi sp, sp, -16 # 留出2个双字的栈空间
      sd ra, 8(sp)     # 返回地址保存到栈中
      sd a0, 0(sp)     # 参数n=2保存到栈中
      addi t0, a0, -1
      bge t0, x0, L1    # 若参数n-1 >= 0, 则跳转到L1
      addi a0, x0, 1
      addi sp, sp, 16
      jalr zero, 0(ra)

L1:   addi a0, a0, -1    # n = n - 1, a0=1
      jal ra, fact      # 调用fact(n-1), 重复直到fact(0).
```

```
Calc: addi t1, a0, 0
      ld a0, 0(sp)
      ld ra, 8(sp)
      addi sp, sp, 16
      mul a0, a0, t1
      jalr x0, 0(ra)
```

SP →



STEP 1

**注：** $x_{1,i}$ 代表x1(a0)寄存器存储的不同的返回地址值。

每一步都是从红色代码开始执行。

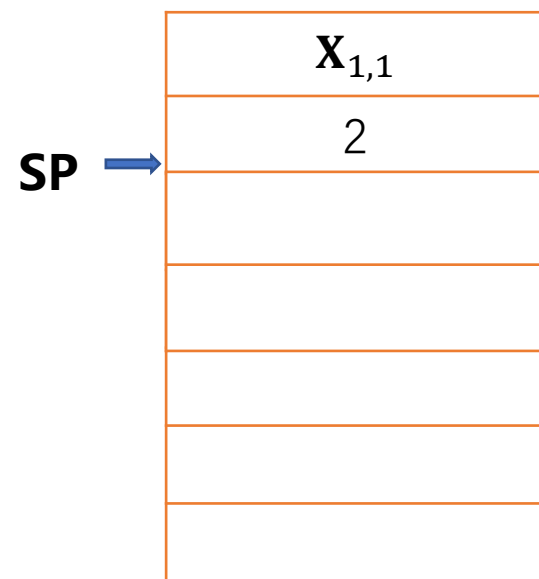
红色代表改变栈指针和栈内容的代码；蓝色代表栈中内容改变后继续执行的代码。

# fact(2)递归调用fact(1)

```
fact: addi sp, sp, -16 #留出2个双字的栈空间
      sd ra, 8(sp)      #返回地址保存到栈中
      sd a0, 0(sp)      #参数n=1保存到栈中
      addi t0, a0, -1
      bge t0, x0, L1     #若参数n-1 >= 0, 则跳转到L1
      addi a0, x0, 1
      addi sp, sp, 16
      jalr zero, 0(ra)

L1: addi a0, a0, -1     # n = n - 1, a0=0
      jal ra, fact      # 调用fact(n-1), 重复直到fact(0).
```

```
Calc: addi t1, a0, 0
      ld a0, 0(sp)
      ld ra, 8(sp)
      addi sp, sp, 16
      mul a0, a0, t1
      jalr x0, 0(ra)
```



STEP 2

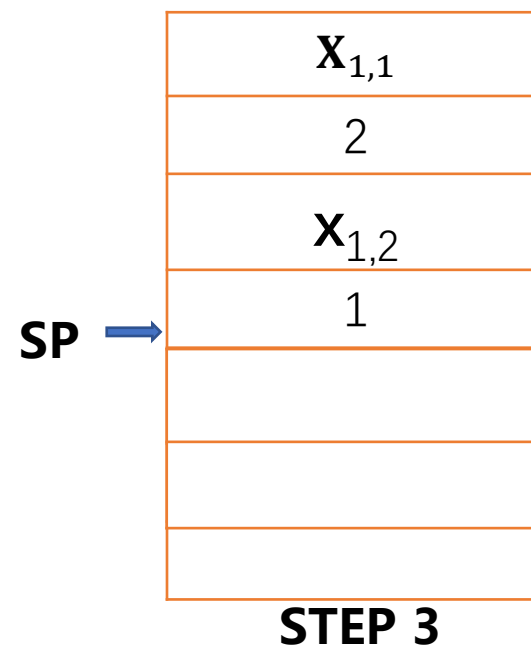
注:  $X_{1,i}$ 代表 $X_1$ 寄存器存储的不同的返回地址值。

每一步都是从红色代码开始执行。

红色代表改变栈指针和栈内容的代码; 蓝色代表栈中内容改变后继续执行的代码。

# fact(1)递归调用fact(0)

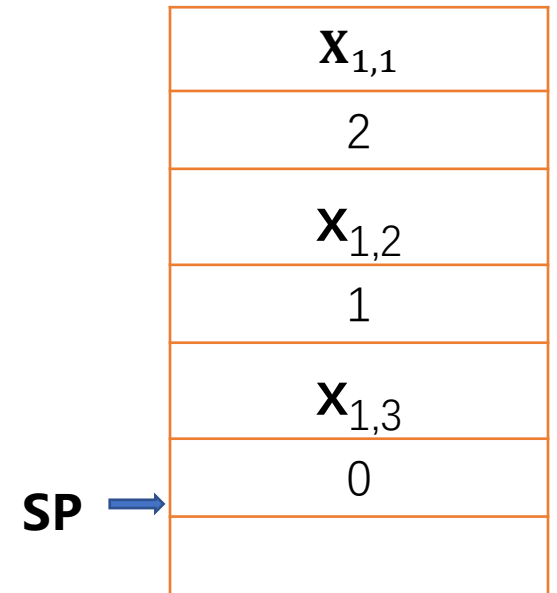
```
fact: addi sp, sp, -16    # 留出2个双字的栈空间
      sd ra, 8(sp)        # 返回地址保存到栈中
      sd a0, 0(sp)        # 参数n=0保存到栈中
      addi t0, a0, -1
      bge t0, x0, L1      # 若参数n-1 >= 0, 则跳转到L1
      addi a0, x0, 1      # 若n-1 < 0, 返回值1并存入a0
      addi sp, sp, 16
      jalr zero, 0(ra)
L1:   addi a0, a0, -1
      jal ra, fact
Calc: addi t1, a0, 0
      ld a0, 0(sp)
      ld ra, 8(sp)
      addi sp, sp, 16
      mul a0, a0, t1
      jalr x0, 0(ra)
```



注：每一步都是从红色代码开始执行。  
红色代表该步中改变**栈指针和栈内容**的代码；  
蓝色代表栈中内容改变后继续执行的代码。

# fact(0)返回结果给fact(1)

```
fact: addi sp, sp, -16 # 留出2个双字的栈空间
      sd ra, 8(sp)    # 返回地址保存到栈中
      sd a0, 0(sp)    # 参数n=0保存到栈中
      addi t0, a0, -1
      bge t0, x0, L1   # 若参数n-1 >= 0, 则跳转到L1
      addi a0, x0, 1    # 若n-1 < 0, 返回值1并存入x10
      addi sp, sp, 16  # 释放2个双字的栈空间
      jalr zero, 0(ra) # 返回fact(0)的调用者: Calc
L1:   addi a0, a0, -1
      jal ra, fact
Calc: addi t1, a0, 0   # t1=1
      ld a0, 0(sp)     # a0=1
      ld ra, 8(sp)     # ra=x1,2
      addi sp, sp, 16
      mul a0, a0, t1
      jalr x0, 0(ra)
```

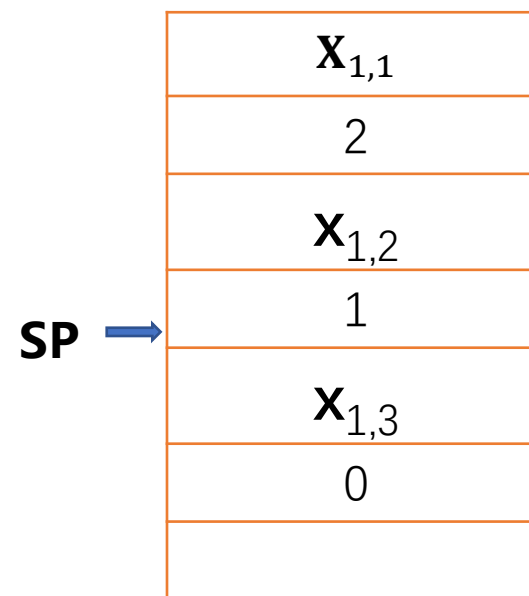


**STEP 4**

注：每一步都是从红色代码开始执行。  
红色代表该步中改变栈指针的代码；蓝色代表栈指针改变后继续执行的代码。

# fact(1)返回结果给fact(2)

```
fact: addi sp, sp, -16 # 留出2个双字的栈空间
      sd ra, 8(sp)      # 返回地址保存到栈中
      sd a0, 0(sp)      # 参数n=0保存到栈中
      addi t0, a0, -1
      bge t0, x0, L1    # 若参数n-1 >= 0, 则跳转到L1
      addi a0, x0, 1    # 若n-1 < 0, 返回值1并存入x10
      addi sp, sp, 16   # 释放2个双字的栈空间, 不用恢复
      jalr zero, 0(ra)  # 返回fact(0)的调用者, Calc
L1:   addi a0, a0, -1   # n = n - 1
      jal ra, fact      # 调用fact(n-1), 重复直到fact(0).
Calc: addi t1, a0, 0    # t1=1
      ld a0, 0(sp)      # 恢复调用者fact(2)的参数a0=2
      ld ra, 8(sp)      # 恢复调用者的返回地址X1,1 (主程序)
      addi sp, sp, 16   # 释放2个双字的栈空间
      mul a0, a0, t1    # a0=a0*t1=1*1=1
      jalr x0, 0(ra)    # 返回Calc ra=X1,2
```

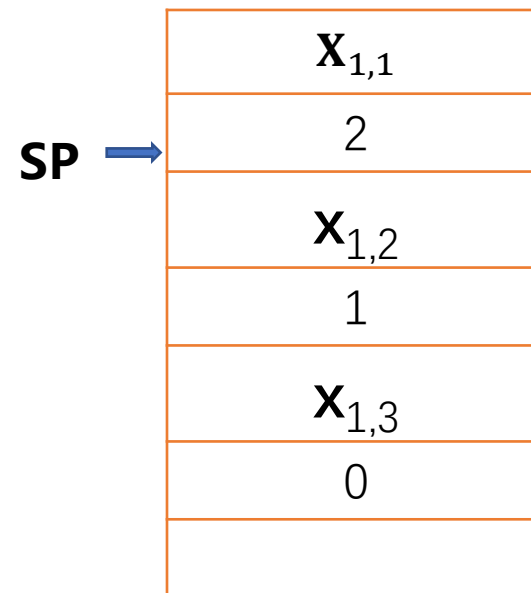


STEP 5

注: 每一步都是从红色代码开始执行。  
红色代表该步中改变栈指针的代码; 蓝色代表栈指针改变后继续执行的代码。

# 计算出fact(2)

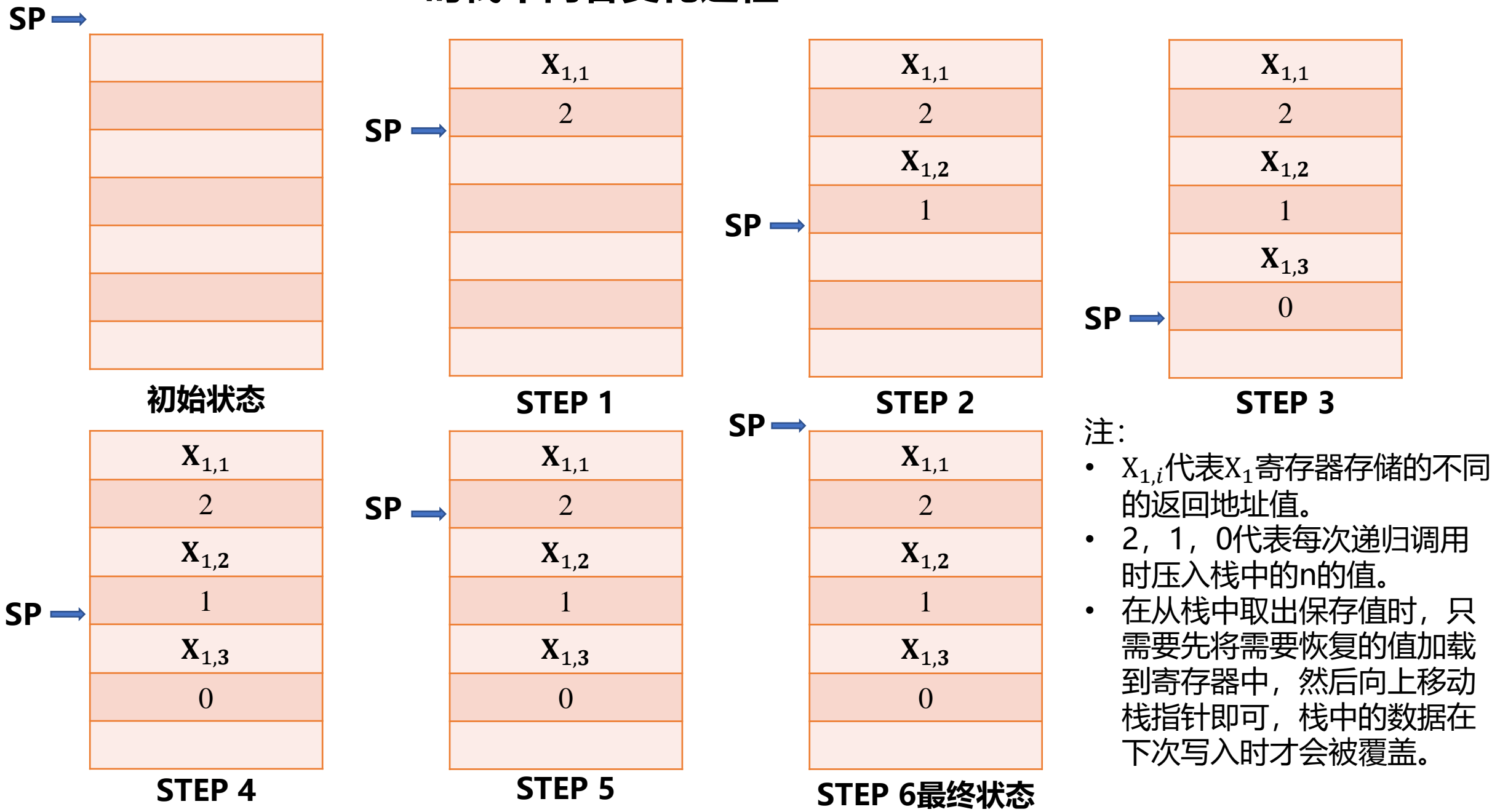
```
fact: addi sp, sp, -16 # 留出2个双字的栈空间
      sd ra, 8(sp)     # 返回地址保存到栈中
      sd a0, 0(sp)     # 参数n=0保存到栈中
      addi t0, a0, -1
      bge t0, x0, L1   # 若参数n-1 >= 0, 则跳转到L1
      addi a0, x0, 1    # 若n-1 < 0, 返回值1并存入x10
      addi sp, sp, 16   # 释放2个双字的栈空间, 不用恢复
      jalr zero, 0(ra)  # 返回fact(0)的调用者, Calc
L1:   addi a0, a0, -1   # n = n - 1
      jal ra, fact     # 调用fact(n-1), 重复直到fact(0).
Calc: addi t1, a0, 0    # t1=1
      ld a0, 0(sp)     # a0=2
      ld ra, 8(sp)     # 恢复调用者的返回地址 X1,1 (主程序)
      addi sp, sp, 16   # 释放2个双字的栈空间
      mul a0, a0, t1    # m * fact(m-1) (m=2)
      jalr x0, 0(ra)    # 返回主程序
```



STEP 6

注: 每一步都是从红色代码开始执行。  
红色代表该步中改变栈指针的代码; 蓝色代表栈指针改变后继续执行的代码。

## n=2时栈中内容变化过程



# 第二章 RISC-V汇编及其指令系统

---

- **RISC-V汇编语言**

- 汇编语言简介
- RISC-V汇编指令概览
- RISC-V常用汇编指令
- 函数调用及栈的使用
- 编译工具介绍



# RISC-V汇编语言规范

- 一个完整的 RISC-V 汇编程序有多条语句（statement）组成。
- 一条典型的 RISC-V 汇编语句由3部分组成：
  - **label**（标号）：任何以冒号结尾的标识符都被认为是一个标号。表示当前指令的位置标记。
  - **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。例如addi指令、lw指令等。
    - 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（指示/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：采用 .macro/.endm 自定义的宏
  - **comment**（注释）：常用方式，“#”开始到当前行结束。

# RISC-V编译工具

---

- RISC-V汇编器有很多种 (<https://riscv.org/exchange/software/>)
- 我们主要使用RARS (RISC-V汇编程序和运行时模拟器) 是一个轻量级的交互式集成开发环境 (IDE), 用于使用RISC-V汇编语言进行编程, 具有代码提示、模拟运行、调试、统计等功能, RARS基本界面如图1所示:

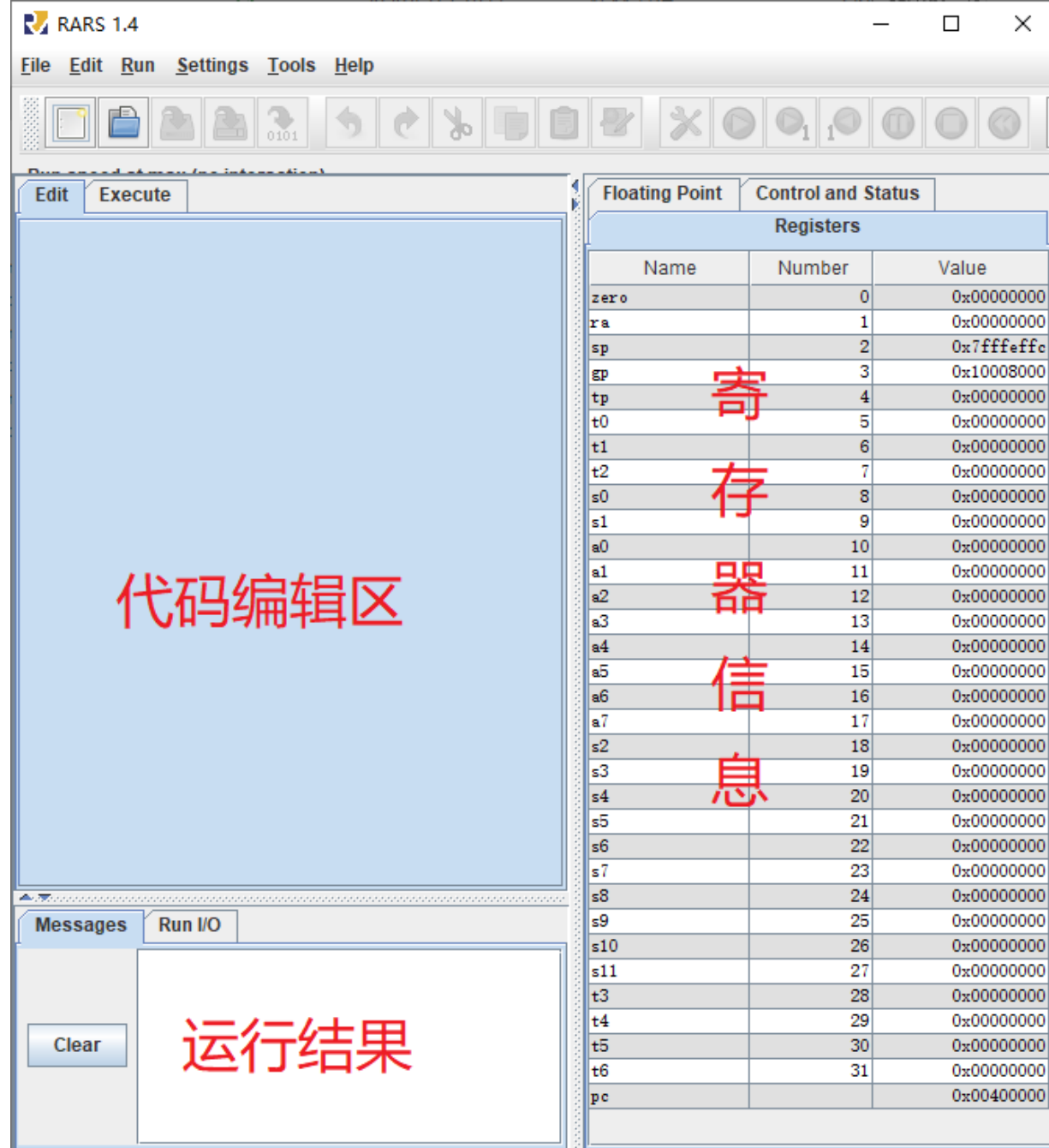


图1 RARS基本界面

# RISC-V编译工具

---

## 1) 运行方法

汇编代码编辑完成后，点击菜单栏的**Run**，选择**Assemble**即可进行汇编。也可以点击下图中用红色框框的图标进行汇编，如图2所示：

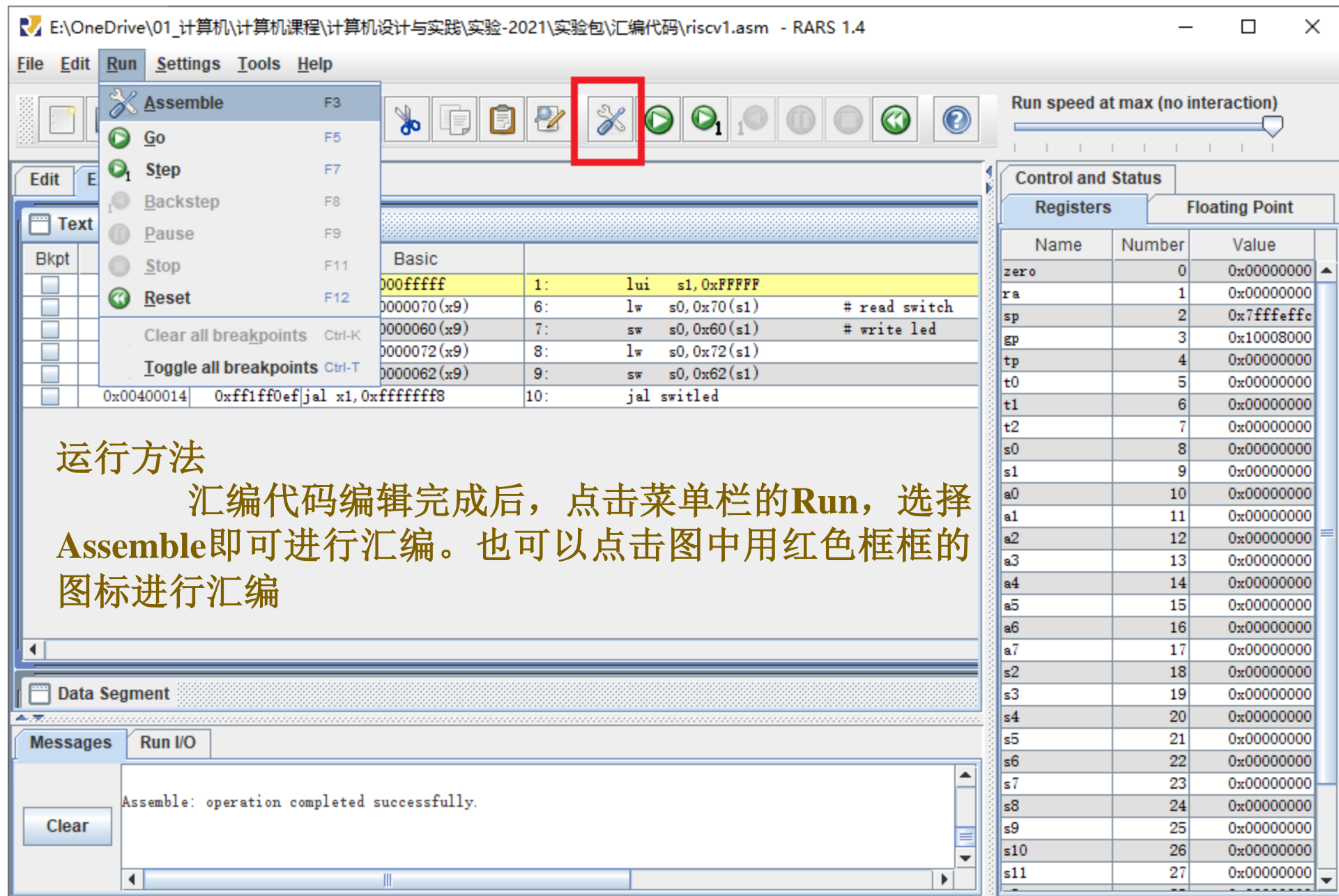


图2 RARS使用方法

# RISC-V编译工具

---

## 2) 导出机器码

程序汇编后可以利用File菜单中的Dump Memory功能将代码段和数据段导出，采用十六进制文本（Hexadecimal Text）的方式导出到 “\*\*.hex”，即可在LOGISIM中加载到RAM或ROM中，如图3（从菜单栏选择导出机器码）和4（选择导出格式）所示。

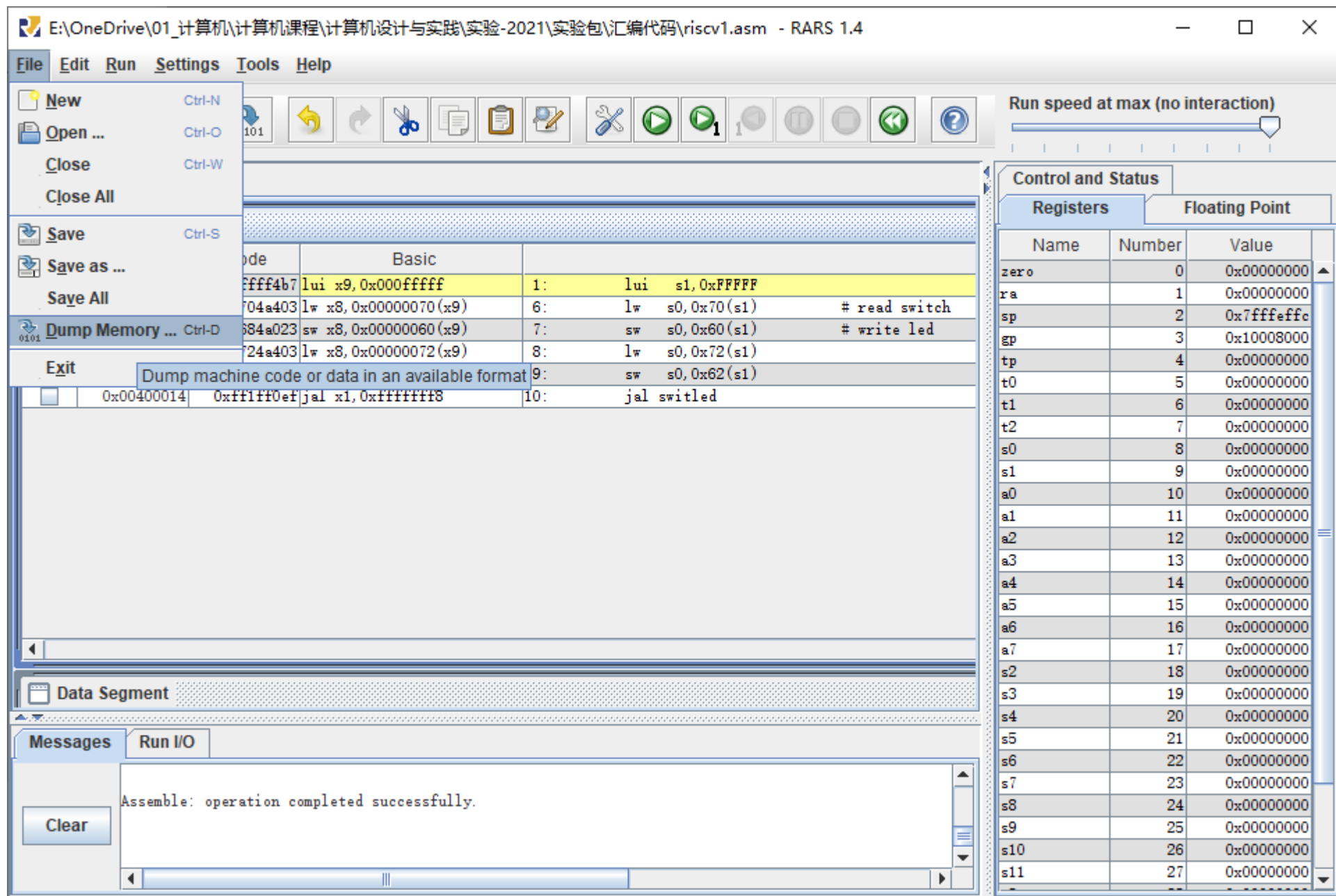


图3 从菜单导出机器码

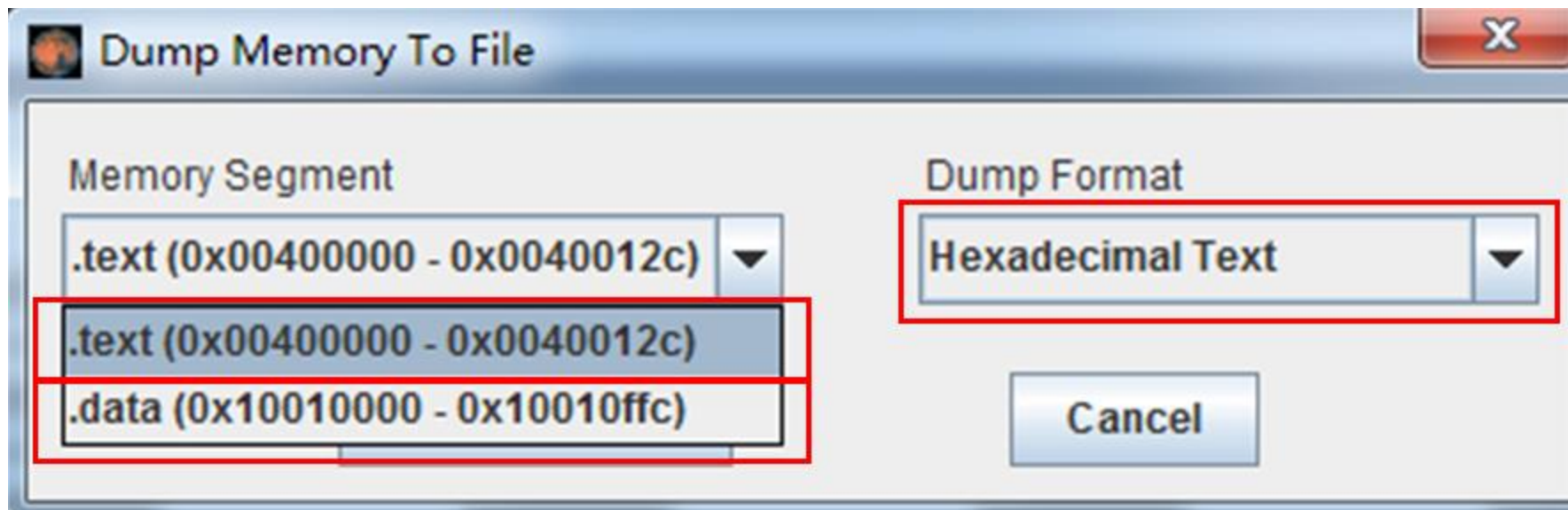


图4 从菜单导出机器码

**注意：** .text是存储在指令存储器中， .data是存储在数据存储器中，需要分别存放到两个 “\*.hex”文件中。（注： 如果汇编代码中没有定义.data， 则不会生成.data段）



# 例子：运行一个简单的数据交换swap

The screenshot displays a RISC-V assembly simulator interface. The main window shows the assembly code for a swap function, with line numbers 1 through 10. The code is as follows:

```
1 addi x10, x0, 21
2 addi x11, x0, 20
3 jal x1, swap
4 j Done
5 swap:
6     add x6, x0, x10
7     add x10, x0, x11
8     add x11, x0, x6
9     jalr x0, 0(x1)
10 Done:
```

The simulator's register window on the right shows the state of the registers. The registers are organized into three columns: Name, Number, and Value. The registers are listed from zero to pc. The values are shown in hexadecimal. The registers t1 and a1 are highlighted with black boxes, indicating they contain the values 0x0000000000000015 and 0x0000000000000014, respectively.

Name	Number	Value
zero	0	0x0000000000000000
ra	1	0x0000000000004000
sp	2	0x000000007fffffc
gp	3	0x0000000010008000
tp	4	0x0000000000000000
t0	5	0x0000000000000000
t1	6	0x0000000000000015
t2	7	0x0000000000000000
s0	8	0x0000000000000000
s1	9	0x0000000000000000
a0	10	0x0000000000000014
a1	11	0x0000000000000015
a2	12	0x0000000000000000
a3	13	0x0000000000000000
a4	14	0x0000000000000000
a5	15	0x0000000000000000
a6	16	0x0000000000000000
a7	17	0x0000000000000000
s2	18	0x0000000000000000
s3	19	0x0000000000000000
s4	20	0x0000000000000000
s5	21	0x0000000000000000
s6	22	0x0000000000000000
s7	23	0x0000000000000000
s8	24	0x0000000000000000
s9	25	0x0000000000000000
s10	26	0x0000000000000000
s11	27	0x0000000000000000
t3	28	0x0000000000000000
t4	29	0x0000000000000000
t5	30	0x0000000000000000
t6	31	0x0000000000000000
pc		0x00000000000040024

The Messages window at the bottom shows the message: "program is finished running (dropped off bottom)".

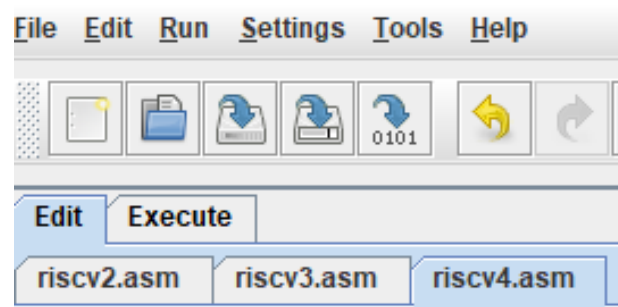
# 运行一个简单的数据交换例子

Text Segment				
Bkpt	Address	Code	Basic	Source
<input type="checkbox"/>	0x00400000	0x01500513	addi x10,x0,21	1: addi x10,x0,21
<input type="checkbox"/>	0x00400004	0x01400593	addi x11,x0,20	2: addi x11,x0,20
<input type="checkbox"/>	0x00400008	0x008000ef	jal x1,0x00000008	3: jal x1,swap
<input type="checkbox"/>	0x0040000c	0x0140006f	jal x0,0x00000014	4: j Done
<input type="checkbox"/>	0x00400010	0x00a00333	add x6,x0,x10	6: add x6,x0,x10
<input type="checkbox"/>	0x00400014	0x00b00533	add x10,x0,x11	7: add x10,x0,x11
<input type="checkbox"/>	0x00400018	0x006005b3	add x11,x0,x6	8: add x11,x0,x6
<input type="checkbox"/>	0x0040001c	0x00008067	jalr x0,x1,0	9: jalr x0,0(x1)

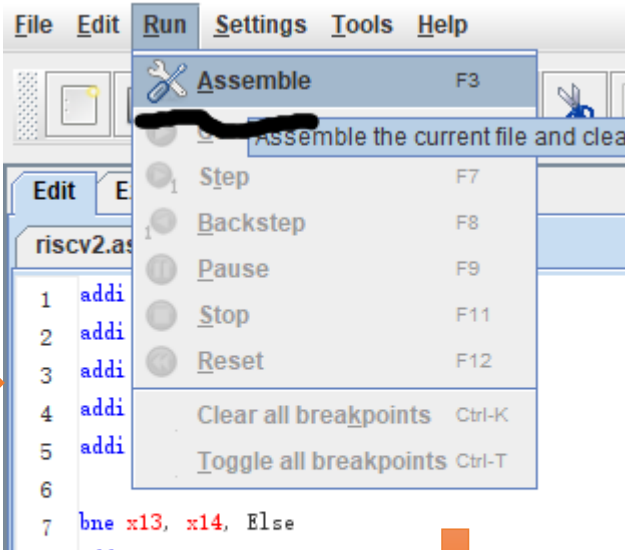
Data Segment						
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)
0x10010000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010100	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010120	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010140	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010160	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010180	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100101a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Messages	
Run I/O	
— program is finished running (dropped off bottom) —	
Clear	

Registers			Floating Point	Control and Status	
Name			Number	Value	
zero			0	0x0000000000000000	
ra			1	0x000000000040000c	
sp			2	0x000000007fffefc	
gp			3	0x0000000010008000	
tp			4	0x0000000000000000	
t0			5	0x0000000000000000	
t1			6	0x0000000000000015	
t2			7	0x0000000000000000	
s0			8	0x0000000000000000	
s1			9	0x0000000000000000	
a0			10	0x0000000000000014	
a1			11	0x0000000000000015	
a2			12	0x0000000000000000	
a3			13	0x0000000000000000	
a4			14	0x0000000000000000	
a5			15	0x0000000000000000	
a6			16	0x0000000000000000	
a7			17	0x0000000000000000	
s2			18	0x0000000000000000	
s3			19	0x0000000000000000	
s4			20	0x0000000000000000	
s5			21	0x0000000000000000	
s6			22	0x0000000000000000	
s7			23	0x0000000000000000	
s8			24	0x0000000000000000	
s9			25	0x0000000000000000	
s10			26	0x0000000000000000	
s11			27	0x0000000000000000	
t3			28	0x0000000000000000	
t4			29	0x0000000000000000	
t5			30	0x0000000000000000	
t6			31	0x0000000000000000	
pc				0x0000000000400024	



```
1 addi x10, x0, 3
2 addi x11, x0, 5
3 addi x12, x0, 7
4 addi x13, x0, 8
5 addi x14, x0, 9
6
7 bne x13, x14, Else
8 add x10, x11, x12
9 j Exit
10 Else: sub x10, x11, x12
11 Exit:
```



Text Segment				
Bkpt	Address	Code	Basic	
<input type="checkbox"/>	0x00400000	0x00300513	addi x10, x0, 3	1: addi x10, x0, 3
<input type="checkbox"/>	0x00400004	0x00500593	addi x11, x0, 5	2: addi x11, x0, 5
<input type="checkbox"/>	0x00400008	0x00700613	addi x12, x0, 7	3: addi x12, x0, 7
<input type="checkbox"/>	0x0040000c	0x00800693	addi x13, x0, 8	4: addi x13, x0, 8
<input type="checkbox"/>	0x00400010	0x00900713	addi x14, x0, 9	5: addi x14, x0, 9
<input type="checkbox"/>	0x00400014	0x00e69663	bne x13, x14, 0x0000000c	7: bne x13, x14, Else
<input type="checkbox"/>	0x00400018	0x00c58533	add x10, x11, x12	8: add x10, x11, x12
<input type="checkbox"/>	0x0040001c	0x0080006f	jal x0, 0x00000008	9: j Exit
<input type="checkbox"/>	0x00400020	0x40c58533	sub x10, x11, x12	10: Else: sub x10, x11, x12

Floating Point		Control and Status	
Registers			
Name	Number	Value	
zero	0	0x00000000	
ra	1	0x00000000	
sp	2	0x7ffffeffc	
gp	3	0x10008000	
tp	4	0x00000000	
t0	5	0x00000000	
t1	6	0x00000000	
t2	7	0x00000000	
s0	8	0x00000000	
s1	9	0x00000000	
a0	10	0xffffffffe	
a1	11	0x00000000	
a2	12	0x00000007	
a3	13	0x00000008	
a4	14	0x00000009	
a5	15	0x00000000	
a6	16	0x00000000	
a7	17	0x00000000	
s2	18	0x00000000	
s3	19	0x00000000	
s4	20	0x00000000	
s5	21	0x00000000	
s6	22	0x00000000	
s7	23	0x00000000	
s8	24	0x00000000	
s9	25	0x00000000	
s10	26	0x00000000	
s11	27	0x00000000	
t3	28	0x00000000	
t4	29	0x00000000	
t5	30	0x00000000	
t6	31	0x00000000	
pc		0x00400028	

执行下列指令序列后，寄存器x10的值是多少？

addi x10, x0, 5

addi x11, x0, 3

beq x11, x0, L1

    add x10, x10, x11

    j Exit

L1:

    sub x10, x10, x11

Exit:

A

8

B

2

C

10

D

0

提交