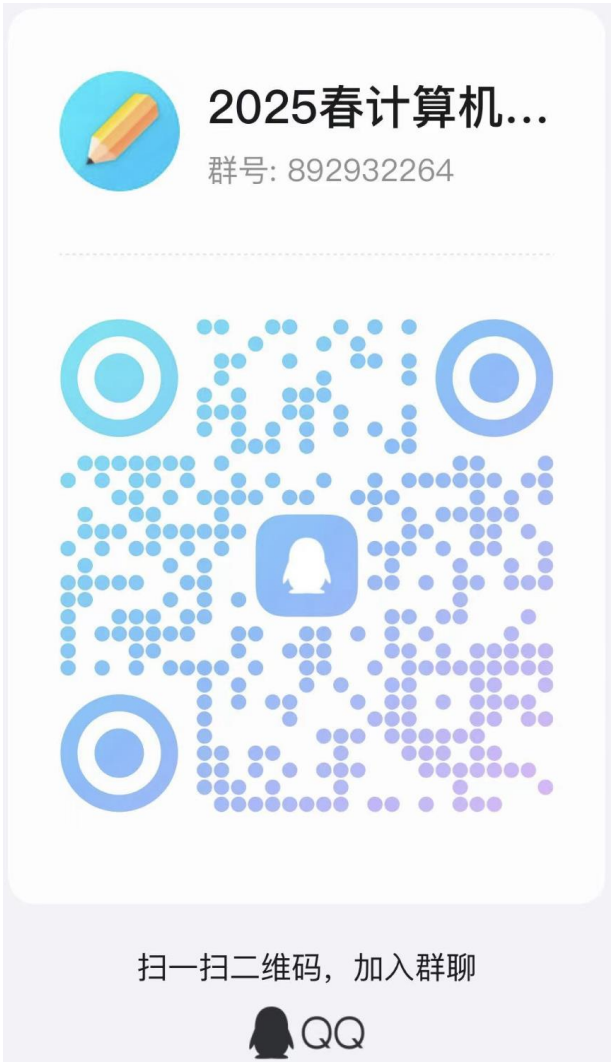
计算机组成原理



花忠云
https://huazhongyun.github.io/
http://faculty.hitsz.edu.cn/huazhongyun

计算机科学与技术学院

第三章 RISC-V汇编及其指令系统

- RISC-V概述
- **RISC-V汇编语言**
- RISC-V指令表示
- 案例分析

第三章 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？

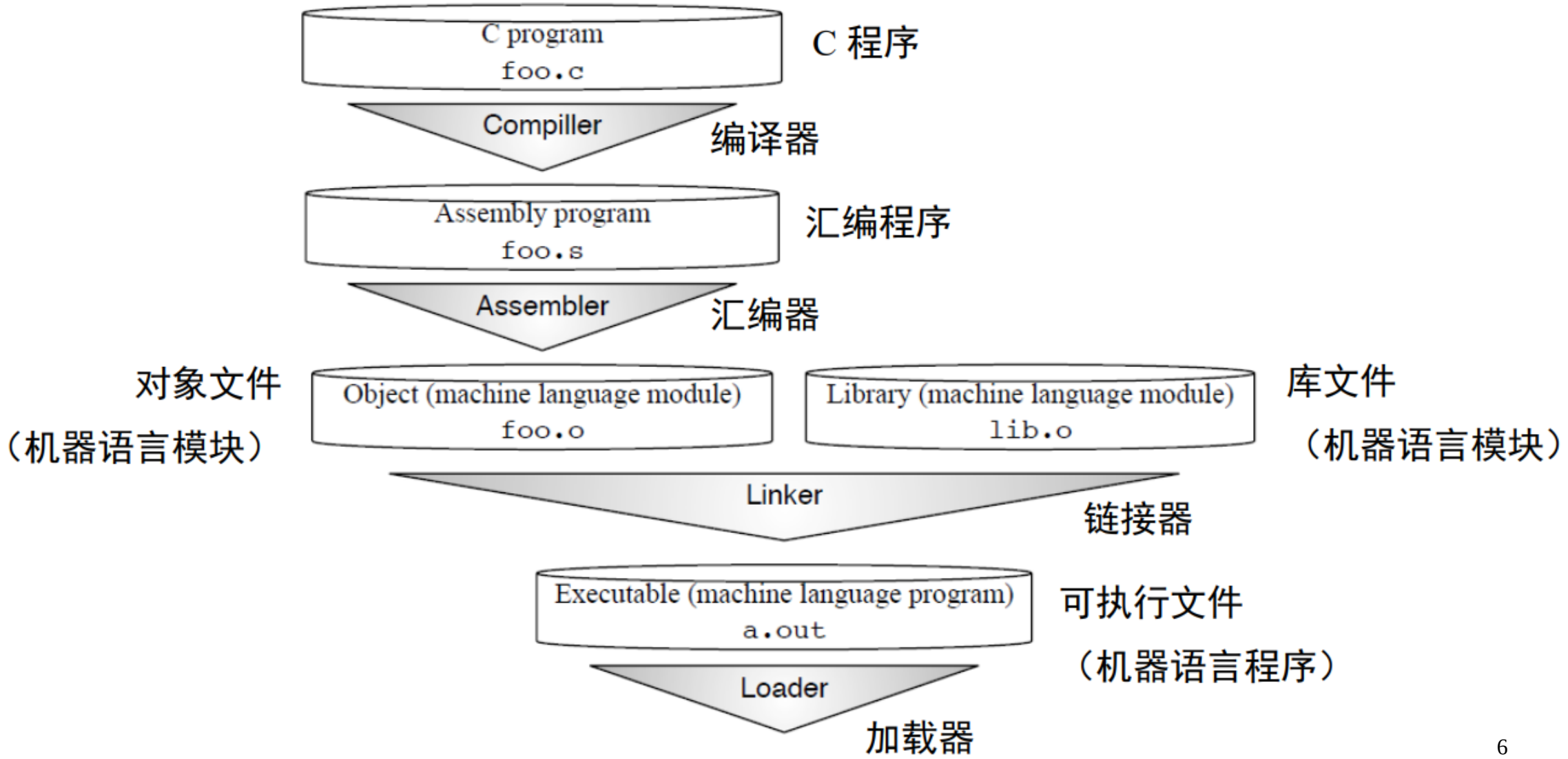
- **汇编语言（Assembly Language）**是一种“低级”语言，直接接触最底层的硬件，需要对底层硬件非常熟悉才能编写出高效的汇编程序。
- 汇编语言属于第二代计算机语言，用一些容易理解和记忆的字母，单词来代替一个特定的指令。**在汇编语言中，用助记符代替机器指令的操作码，用地址符号或标号代替指令或操作数的地址。**
- 例如：用“ADD”代表数字逻辑上的加减，“MOV”代表数据传递等等。

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 program
foo.c
C 程序
Compiller
编译器
Assembly program
foo.s
汇编程序
Assembler
汇编器
对象文件
（机器语言模块）
Object (machine language module)
foo.o
Library (machine language module)
lib.o
库文件
（机器语言模块）
Linker
链接器
Executable (machine language program)
a.out
可执行文件
（机器语言程序）
Loader
加载器

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

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。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。它们在第一次调用例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

第三章 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（**op**eration）
  - 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

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 ~ 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
  - **RV32I指令集通用寄存器是32位整数寄存器，RV64I指令集，通用寄存器是64位寄存器**
- **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）
  - int a;　float b;
  - 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	备注
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（temporary register）
x8	s0/fp	save寄存器/帧指针(Frame Pointer)
x9	s1	save寄存器（Save Register）
x10-x11	a0-a1	函数参数 / 函数返回值(Return Value)
x12-x17	a2-a7	函数参数(Function Argument)
x18-x27	s2-s11	save寄存器
x28-x31	t3-t6	临时寄存器

RISC-V汇编语言指令分类

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

伪指令：方便程序员使用的更加直观的指令，以上指令的组合。

后缀i = immediate

后缀u = unsigned

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 = Memory[x6 + 40]	从存储器取双字到寄存器
	存双字	sd x5, 40(x6)	Memory[x6 + 40] = x5	从寄存器存双字到存储器
	取字	lw x5, 40(x6)	x5 = Memory[x6 + 40]	从存储器取字到寄存器
	取字（无符号数）	lwu x5, 40(x6)	x5 = Memory[x6 + 40]	从存储器取无符号字到寄存器
	存字	sw x5, 40(x6)	Memory[x6 + 40] = x5	从寄存器存字到存储器
	取半字	lh x5, 40(x6)	x5 = 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 = 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)	Memory[x6 + 40] = x5	从寄存器存半字到存储器
	取字节	lb x5, 40(x6)	x5 = 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 = Memory[x6 + 40]	从存储器取无符号字节到寄存器
	存字节	sb x5, 40(x6)	Memory[x6 + 40] = x5	从寄存器存字节到存储器
	取保留字	lr.d x5, (x6)	x5 = Memory[x6]	取；原子交换的前半部分
	存条件字	sc.d x7, x5, (x6)	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 ^ 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 ^ 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	srli 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

第三章 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

算术运算指令

- 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**
  - 语法：add rd, rs1, rs2　　（rs1,rs2 为x0-x31中的某一个寄存器）
  - 功能：把寄存器rs2的**值**与寄存器 rs1的的**值相加，结果写入寄存器rd（忽略算术溢出）**，记为**R[rd]=R[rs1]+R[rs2]。**
- **减法：sub**
  - 语法：sub rd, rs1, rs2
  - 功能：　**R[rd]=R[rs1] - R[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——整数加减法的例子

- 假设：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→x1, b→x2, c→x3, d→x4, e→x5。下面会将一段C语言程序编译成RISC-V汇编指令

a = (b + c) – (d + e);

```
add x6, x4, x5          # tmp1 = d + e
add x7, x2, x3          # tmp2 = b + c
sub x1, x7, x6          # a = (b + c) – (d + e)
```

- 指令中可以看到如何使用临时寄存器（x5-x7的助记符t0-t2）
- 一个简单的C语言表达式变成多条汇编语句

#符号后面是程序的注释

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
```

RISC-V 中的立即数

- 数值常数被称为立即数（immediates）
- 立即数指令的语法： **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 rd, rs1, immediate　　# R[rd]=R[rs1] + sext(immediate) :**
    # 符号扩展立即数

addi x1, x2, 5　# x1=x2+5

addi x3, x3, 1　# x3=x3+1 （自增1）

RISC-V 中的立即数

- 问题：在RISC-V中没有立即数减法, 为什么？
  - 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)

为什么有SUB？

RISC-V 乘法与除法指令

- **乘法：mul**
  - 使用语法：mul rd, rs1, rs2
  - 功能描述：R[rd] = R[rs1] × R[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：R[rd] = R[rs1] ÷ R[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

- 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

算术运算指令

- 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	R[rd] = R[rs1] & R[rs2]	按位“与”操作，结果写入寄存器rd
andi rd, rs1, imm	R[rd] = R[rs1] & sext(imm)	把**符号位扩展的立即数**和寄存器rs1的值进行按位与，结果写入寄存器rd
or rd, rs1, rs2	R[rd] = R[rs1] | R[rs2]	按位“或”操作，结果写入寄存器rd
ori rd, rs1, imm	R[rd] = R[rs1] | sext(imm)	把**符号位扩展的立即数**和寄存器rs1的值进行按位或，结果写入寄存器rd
xor rd, rs1, rs2	R[rd] = R[rs1] ^ R[rs2]	按位“异或”操作，结果写入寄存器rd
xori rd, rs1, imm	R[rd] = R[rs1] ^ sext(imm)	把**符号位扩展的立即数**和寄存器rs1的值进行按位“异或”，结果写入寄存器rd

位操作指令实例

假设：a→x1, b→x2, c→x3

指令	C	RSC-V
And	`a = b & c;`	and x1, x2, x3
And Immediate	`a = b & 8;`	andi x1, x2, 8
Or	`a = b | c;`	or x1, x2, x3
Or Immediate	`a = b | 22;`	ori x1, x2, 22
eXclusive OR	`a = b ^ c;`	xor x1, x2, x3
eXclusive OR Immediate	`a = b ^ -5;`	xori x1, x2, -5

位运算举例

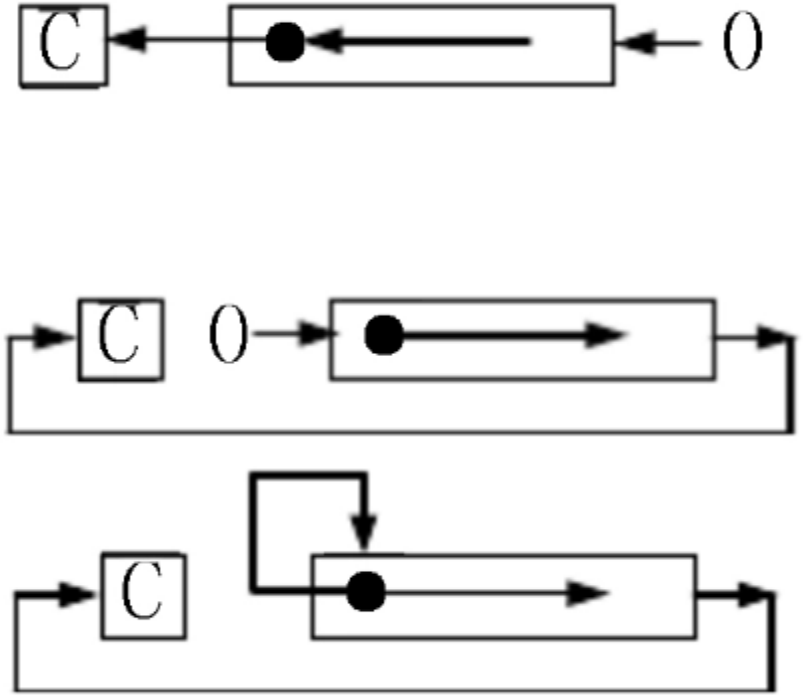
- 逻辑运算
  - 寄存器： and x5, x6, x7　　# x5 = x6 & x7
  - 立即数： andi x5, x6, 3　　# x5 = x6 & 3
- RISC-V中**没有NOT**
  - 对 11111111_2 使用xori
  - 例： xori x5, x6, -1　　# x5= $\overline{x6}$

算术运算指令

- 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



- **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RV64)，只会使用immediate低6位的值 （I类型指令）

imm[11:0]		rs1	111	rd	0010011	andi
000000	**shamt**	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	同逻辑右移
-1	11111111	11111110	11111110	正确	01111111	11111111
.....	.....	.....	.....	正确	0xx.....x	11xxx.....x
-64	11000000	10000000	10000000	正确	01100000	11100000
-65	10111111	01111110	11111110	错误	01011111	11011111
-66	10111110	01111100	11111100	错误	01011111	11011111
-128	10000000	00000000	10000000	错误	01000000	11000000

左移

-61的补码 **1 1 0 0 0 0 1 1**

1 ← **1 0 0 0 0 1 1 0**

$[10000110]_{补}$=-122

+61的补码 **0 0 1 1 1 1 0 1**

0 ← **0 1 1 1 1 0 1 0**

$[01111010]_{补}$=122

-65的补码 **1 0 1 1 1 1 1 1**

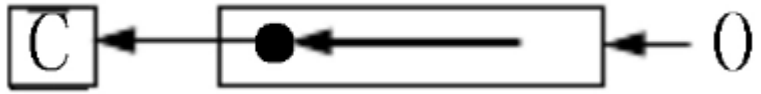
1 ← **0 1 1 1 1 1 1 0**

$[01111110]_{补}$=126

+65的补码 **0 1 0 0 0 0 0 1**

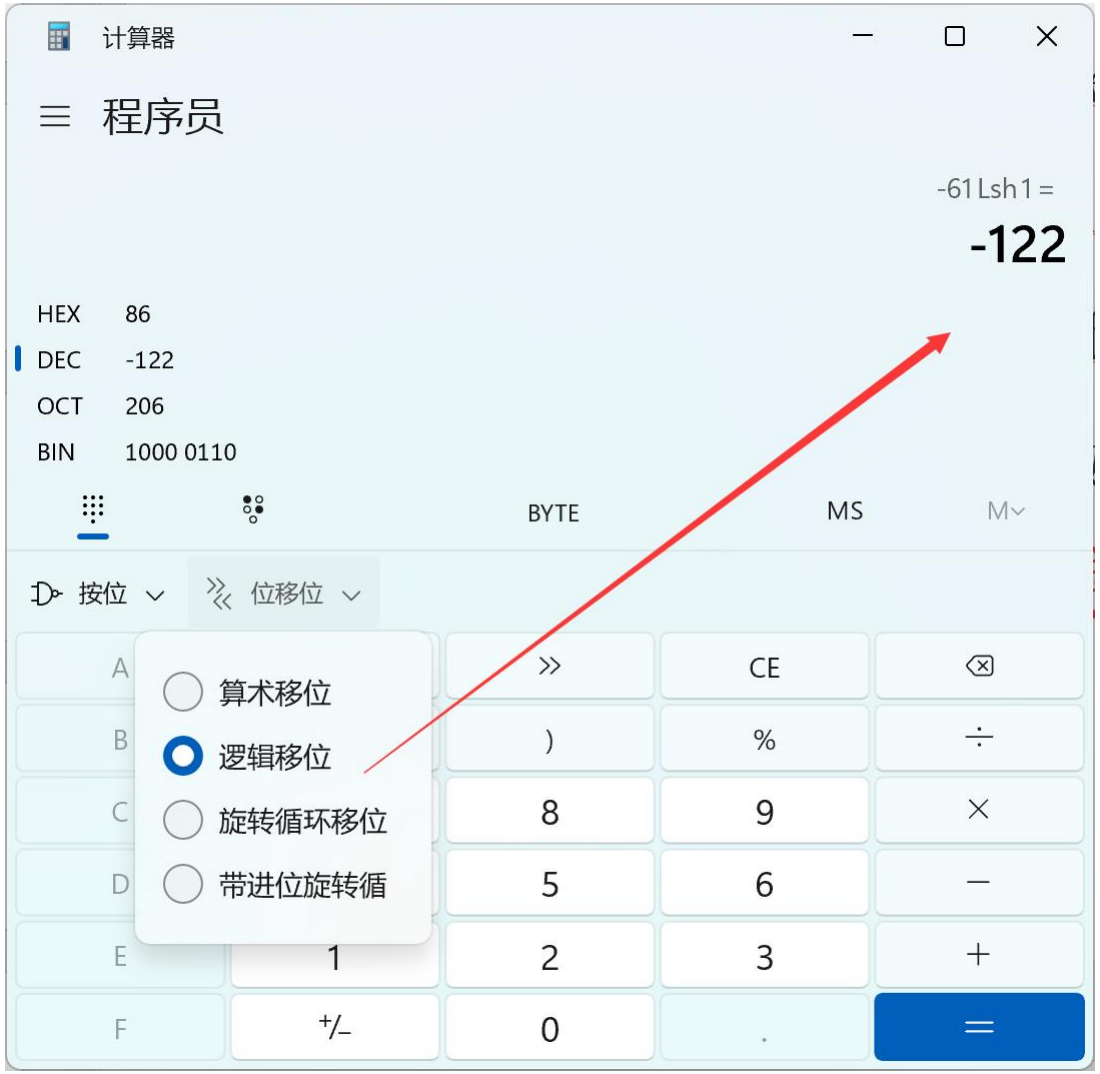
0 ← **1 0 0 0 0 0 1 0**

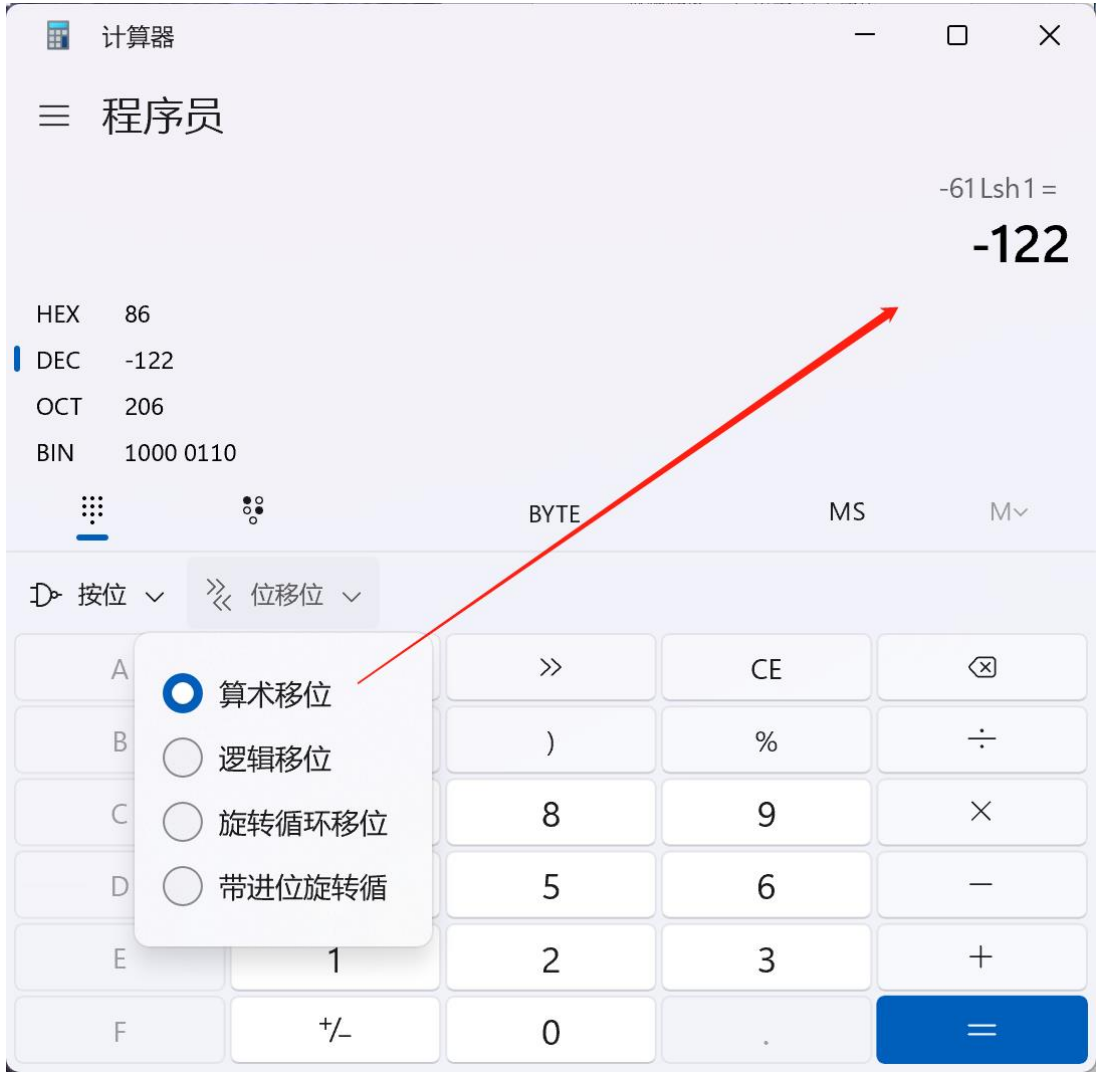
$[10000010]_{补}$=-126

左移 

左移后，“进位”和“首位”一致和不一致?

-61


计算器
程序员
-61 Lsh 1 =
-122
HEX 86
DEC -122
OCT 206
BIN 1000 0110
BYTE
MS
M
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
A
B
C
D
E
F
CE
)
%
8
9
5
6
1
2
3
+/-
0
.


计算器
程序员
-61 Lsh 1 =
-122
HEX 86
DEC -122
OCT 206
BIN 1000 0110
BYTE
MS
M
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
A
B
C
D
E
F
CE
)
%
8
9
5
6
1
2
3
+/-
0
.

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（内存访问指令）

- C语言中的变量会映射到寄存器中；而其它的大量的数据结构，例如数组会映射到内存中
- 内存是一维的数组，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内存之间传输数据
  - **Store指令：数据从寄存器到内存**
  - **Load指令：数据从内存到寄存器**

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的各种变形）
  - reg：源寄存器或者目标寄存器
  - bAddrReg：存数据地址的基地址寄存器 (base address register)
  - offset：地址偏移，字节寻址，为立即数 (immediate)
  - 访问的主存地址为 R[bAddrReg] + offset
  - 必须指向一个合法的地址

内存的字节寻址方式

- 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
  - 内存是按照字节进行编址，不是按照字进行编址的
- 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**

内存与变量的大小（variable size）

- 数据传输指令
  - lw reg, offset(bAddrReg)　# 从存储器 **取** 字到寄存器
  - sw reg, offset(bAddrReg)　# 从寄存器 **存** 字到存储器

注意：R[bAddrReg]+offset 按照字进行对齐，即4的倍数

—例如整数的数字：每个整数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），都不是4个字节的整数倍？

双字：ld/sd；半字：lh/sh..；字符：lb/sb..

主存数据访问指令：字节/半字/字操作

- 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 halfbyte
  - 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
  - lb**u** rd, offset(rs1)
  - lh**u** rd, offset(rs1)
  - lw**u** 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)
  - sh rs2, offset(rs1)
  - sw rs2, offset(rs1)

数据传输指令举例

int A[100];　　(in C)

A[10] = A[3] + a;

（RISC-V中一个字是32位，int类型是32位）

```
# addr of int A[] -> x3,   a -> x2        (in RISC-V)
        lw    x10, 12(x3)      # x10 = A[3]
        add   x10, x2, x10     # x10 = A[3]+a
        sw    x10, 40(x3)      # A[10] = A[3]+a
```

数据传输指令举例

- 读内存指令**(RISC-V中双字是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**64**)

  add x11, x12, x10　　# g = h + A[3]

- 基址寻址
  - 基址寄存器(x15) 的内容+ 偏移量(24)

数据传输指令举例

- 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**64**)

  add x10, x12, x10　　# x10=h + A[3]

  sd x10, 80(x15)　　# A[10] = h + A[3]

- 基址寻址
  - 基址寄存器(x15)的内容 + 偏移量(80)

传输一个字节数据

- 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
```

- 使用专门的字节传输指令

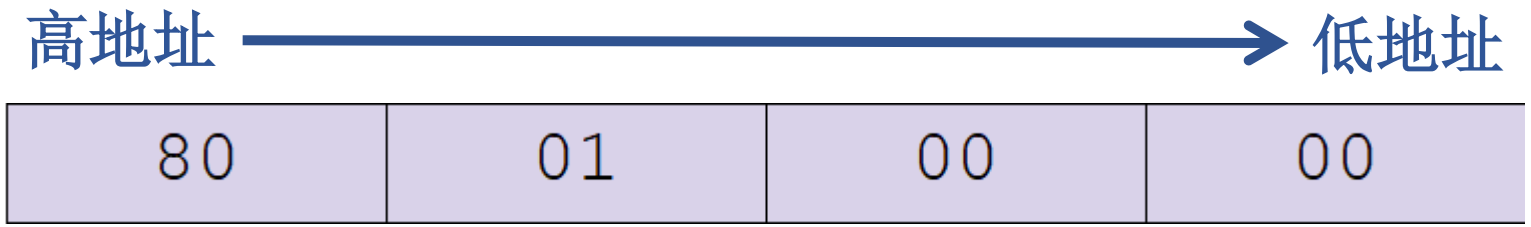
```
lb     x11, 0(x1)
sb     x11, 0(x1)
```

  - 字节传输指令无需字对齐

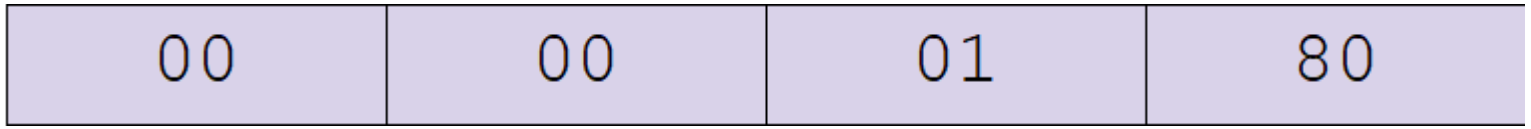
字节排布

- 大端：字地址等于最高字节的地址（字内高字节在最低的地址）
- 小端：字地址等于最低字节的地址（字内低字节在最低的地址）

假设某地址储存数据为 0x00000180



Big Endian

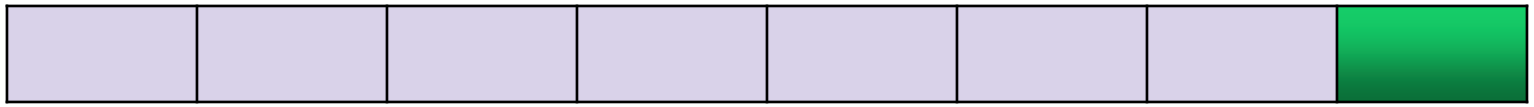


Little Endian

RISC-V 是 小端模式

字节数据传输指令

- lb/sb使用的是低字节
- 如果是sb指令，高56位(RV64)/24位(RV32)被忽略
- 如果是lb指令，高56位(RV64)/24位(RV32)做符号扩展



- 例如：mem[x1]中存储的数据为0x10203040506070 80

(RISC V) lb x11, 1(x1) #x11=0x0000000000000070

lb x12, 0(x1) #x12=0xFFFFFFFFFFFFFF80

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

半字数据传输指令

- l**h** reg, off(bAddr)　#load half word
- s**h** reg, off(bAddr)　#store half word
- lh**u**、sh**u**　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)做**符号扩展**
- lbu、lhu指令, 高位都做0扩展

主存数据访问指令：字节/半字/字操作

- 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 halfbyte
  - 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
  - lb**u** rd, offset(rs1)
  - lh**u** rd, offset(rs1)
  - lw**u** 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)
  - sh rs2, offset(rs1)
  - sw rs2, offset(rs1)

lui指令

- lui指令全称是Load Upper Immediate
- 将20位的立即数加载到寄存器高20位，低12位为0（RV32）

  lui rd, imm

- 例如：lui x1, 0x12345

- 将0x12345000加载到x1寄存器

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, 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, 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,src1,src2: unsigned comparison

  sltiu dst,src,imm: unsigned comparison against constant

- 例:（RISC-V32）

  addi x10,x0,-1

  slti x11,x10,1

  sltiu x12,x10,1

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:)
  - 汇编的控制流就是跳转到标记的位置
  - 在C语言中有类似结构，但被认为是坏的编程风格（goto）

条件分支（跳转）指令

- 条件为真则转到标签所指的语句执行，否则按序执行。
- beq reg1,reg2,label　　#**b**ranch if **eq**ual
  - 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　　#**b**ranch if **n**ot **e**qual
  - 如果R[reg1]**不等于**R[reg2], 程序跳转到label处继续执行
- blt reg1,reg2,label　　#**b**ranch if **l**ess **t**han
  - 如果R[reg1] < R[reg2], 程序跳转到label处继续执行
- bge reg1,reg2,label　　#**b**ranch if **g**reater than or **e**qual
  - 如果R[reg1] >= R[reg2], 程序跳转到label处继续执行

注意:**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: ....
```

条件分支转移指令

C语言中的if-else语句

f→x10, g→x11, h →x12, i →x13, j →x14

```
if (i == j)
    f = g + h;
else
  f = g – h;
```

```
bne x13, x14, Else
add x10, x11, x12
j Exit
Else:  sub x10, x11, x12
Exit:
```

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 **eq**ual
- 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 符号扩展
    - lb,lh vs. lbu,lhu
  - Signed vs. unsigned comparison 比较
    - **slt, slti vs. sltu, sltiu**
  - Signed vs. unsigned branch 比较
    - **blt, bge vs. 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**

无条件跳转指令

- **jal rd, Label（用于函数调用：跳转和链接）**
  - 将下一条指令的地址（ PC+4 ）保存在rd中（通常用**x1**）
  - 跳转到目标地址Label（立即数，有规定的转换规则）
- **jalr rd, offset(rs) （可以用于函数返回）**
  - 类似jal，但跳转到R[rs]+offfset地址处的指令
  - 把PC+4存到rd中，如果rd用x0，相当于只跳转不返回（实际x0不会被改变）

RISC-V汇编中的伪指令

- 伪指令可以给程序员更加直观的指令，但不是直接通过硬件来实现，而是通过汇编器来翻译为实际的硬件指令
- 例子：

  mv dst, src

  并没有实际的数据移动指令，被翻译为下面的指令

  addi dst, src, 0 或者 add dst, src, x0

其它的伪指令

- Load Immediate (li) 装入一个立即数
  - –li dst, imm
  - –装入一个12位的立即数到 dst
  - –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
  - –(思考一下如何翻译)
- 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] = ~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
`nop`	`addi x0, x0, 0`
`not rd, rs`	`xori rd, rs, -1`
`beqz rs, offset`	`beq rs, x0, offset`
`bgt rs, rt, offset`	`blt rt, rs, offset`
`j offset`	`jal x0, offset`
`ret`	`jalr x0, x1, offset`
`call offset` (if too big for just a `jal`)	`auipc x6, offset[31:12]` `jalr x1, x6, offset[11:0]`
`tail offset` (if too far for a `j`)	`auipc x6, offset[31:12]` `jalr x0, x6, offset[11:0]`

伪指令 vs. 硬件指令

- 硬件指令
  - 所有指令都是硬件可以直接执行的指令，在硬件中直接实现了
- 伪指令
  - 汇编语言程序员可以使用的指令（加上了伪指令）
  - 每一条伪指令指令会被翻译为1条或者多条硬件指令

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
	0x00400000	0x10203337	lui x6, 0x00010203	1: lui x6, 0x00010203
	0x00400004	0x000103b7	lui x7, 16	2: li t2, 0x00010203
	0x00400008	0x20338393	addi x7, x7, 0x00000203	
	0x0040000c	0x04330313	addi x6, x6, 0x00000043	3: addi x6, x6, 0x00000043
	0x00400010	0x00331313	slli x6, x6, 3	4: slli x6, x6, 3
	0x00400014	0x01c30313	addi x6, x6, 28	5: addi x6, x6, 28
	0x00400018	0x406003b3	sub x7, x0, x6	6: neg t2, t1
	0x0040001c	0x006002b3	add x5, x0, x6	7: mv x5, x6

Name	Number	Value
zero	0	0x00000000
ra	1	0x00000000
sp	2	0x7fffeffc
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

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，

save的地址在x25中，

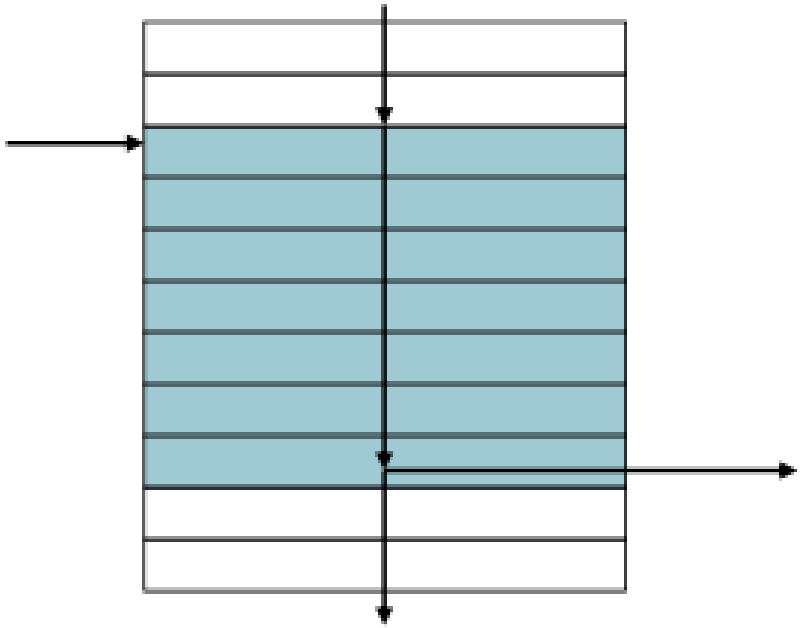
save数组是Long long int 类型。

• RISC-V代码：

```
Loop:
    slli x10, x22, 3
    add x10, x10, x25
    ld x9, 0(x10)
    bne x9, x24, Exit
        addi x22, x22, 1
        beq x0, x0, 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（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

上一堂课回顾

- 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（内存访问指令）

- C语言中的变量会映射到寄存器中；而其它的大量的数据结构，例如数组会映射到内存中
- 内存是一维的数组，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内存之间传输数据
  - **Store指令：数据从寄存器到内存**
  - **Load指令：数据从内存到寄存器**

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的各种变形）
  - reg：源寄存器或者目标寄存器
  - bAddrReg：存数据地址的基地址寄存器 (base address register)
  - offset：地址偏移，字节寻址，为立即数 (immediate)
  - 访问的主存地址为 R[bAddrReg] + offset
  - 必须指向一个合法的地址

内存与变量的大小（variable size）

- 数据传输指令
  - lw reg, offset(bAddrReg)　# 从存储器 **取** 字到寄存器
  - sw reg, offset(bAddrReg)　# 从寄存器 **存** 字到存储器

注意：R[bAddrReg]+offset 按照字进行对齐，即4的倍数

—例如整数的数字：每个整数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），都不是4个字节的整数倍？

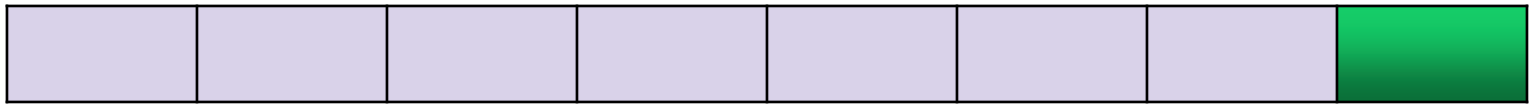
双字：ld/sd；半字：lh/sh..；字符：lb/sb..

主存数据访问指令：字节/半字/字操作

- 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 halfbyte
  - 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
  - lb**u** rd, offset(rs1)
  - lh**u** rd, offset(rs1)
  - lw**u** 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)
  - sh rs2, offset(rs1)
  - sw rs2, offset(rs1)

字节数据传输指令

- lb/sb使用的是低字节
- 如果是sb指令，高56位(RV64)/24位(RV32)被忽略
- 如果是lb指令，高56位(RV64)/24位(RV32)做符号扩展



- 例如：mem[x1]中存储的数据为0x10203040506070 80

(RISC V)　lb x11, 1(x1)　#x11=0x00000000000070

lb x12, 0(x1)　#x12=0xFFFFFFFFFFFFFF80

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

半字数据传输指令

- l**h** reg, off(bAddr)　#load half word
- s**h** reg, off(bAddr)　#store half word
- lh**u**、sh**u**　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)做**符号扩展**
- lbu、lhu指令, 高位都做0扩展

主存数据访问指令：字节/半字/字操作

- 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 halfbyte
  - 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
  - lb**u** rd, offset(rs1)
  - lh**u** rd, offset(rs1)
  - lw**u** 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)
  - sh rs2, offset(rs1)
  - sw rs2, offset(rs1)

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, 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, dst=1, 否则dst=0

条件分支（跳转）指令

- 条件为真则转到标签所指的语句执行，否则按序执行。
- beq reg1,reg2,label　　#**b**ranch if **eq**ual
  - 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　　#**b**ranch if **n**ot **e**qual
  - 如果R[reg1]**不等于**R[reg2], 程序跳转到label处继续执行
- blt reg1,reg2,label　　#**b**ranch if **l**ess **t**han
  - 如果R[reg1] < R[reg2], 程序跳转到label处继续执行
- bge reg1,reg2,label　　#**b**ranch if **g**reater than or **e**qual
  - 如果R[reg1] >= R[reg2], 程序跳转到label处继续执行

注意:**没有依据标志位**的跳转（与x86不同)

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 **eq**ual
- 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 符号扩展
    - lb,lh vs. lbu,lhu
  - Signed vs. unsigned comparison 比较
    - **slt, slti vs. sltu, sltiu**
  - Signed vs. unsigned branch 比较
    - **blt, bge vs. bltu, bgeu**

无条件跳转指令

- **jal rd, Label（用于函数调用：跳转和链接）**
  - 将下一条指令的地址（ PC+4 ）保存在rd中（通常用**x1**）
  - 跳转到目标地址Label（立即数，有规定的转换规则）
- **jalr rd, offset(rs) （可以用于函数返回）**
  - 类似jal，但跳转到R[rs]+offfset地址处的指令
  - 把PC+4存到rd中，如果rd用x0，相当于只跳转不返回（实际x0不会被改变）

第三章 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可以访问到的位置（x10~x17）
  - 将控制转交给函数
  - 获取函数所需的存储资源
  - 执行过程中的操作
  - 将结果值放在调用程序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 rd, ProcedureLabel

  - 将下一条指令的地址（ PC+4 ）保存在rd中
  - 跳转到目标地址

- **过程返回：寄存器跳转和链接**

  jalr rd, offset(rs)

  - 类似jal，但跳转到offset + rs中保存的地址
  - 把PC+4存到rd中，如果rd用x0，相当于只跳转不返回（实际x0不会被改变）

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实现过程调用的机制
  - 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
  - 堆栈指针　　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/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

```
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; jal x0, label
1016 …  # 下个指令
…
2000 sum: add a0, a0, a1 # a0=a0+a1
2004 jr ra   # jump register; jalr x0, 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
```

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

ra=?

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　　#（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)

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？

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)

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

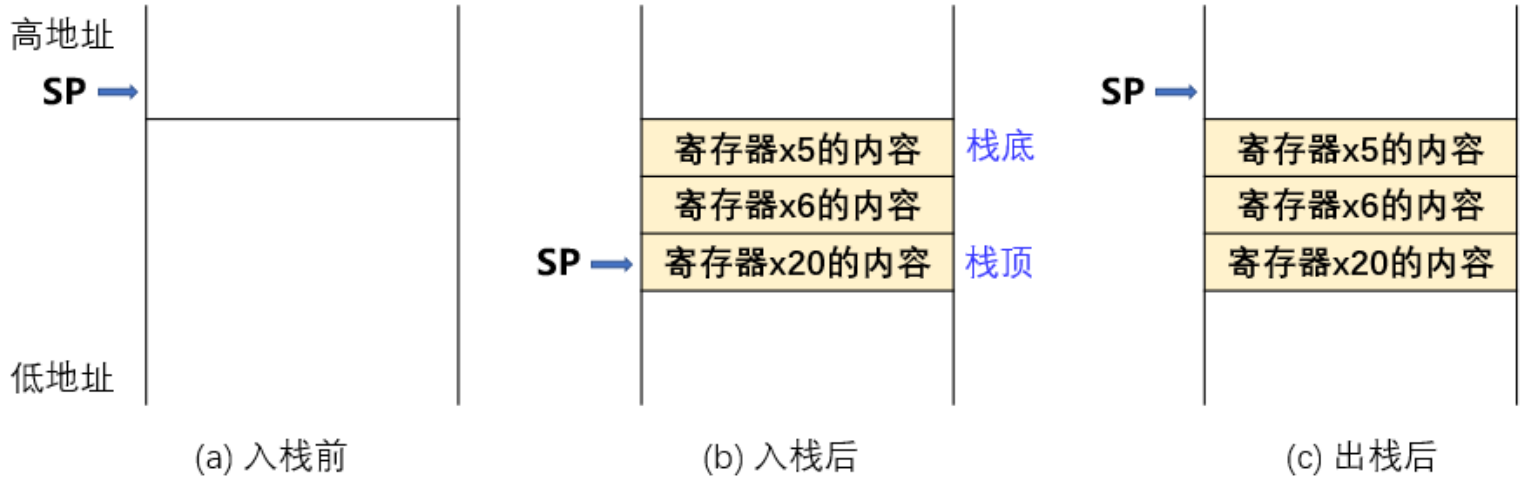
调用mult前需使用**栈（stack）**保存返回地址ad1

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

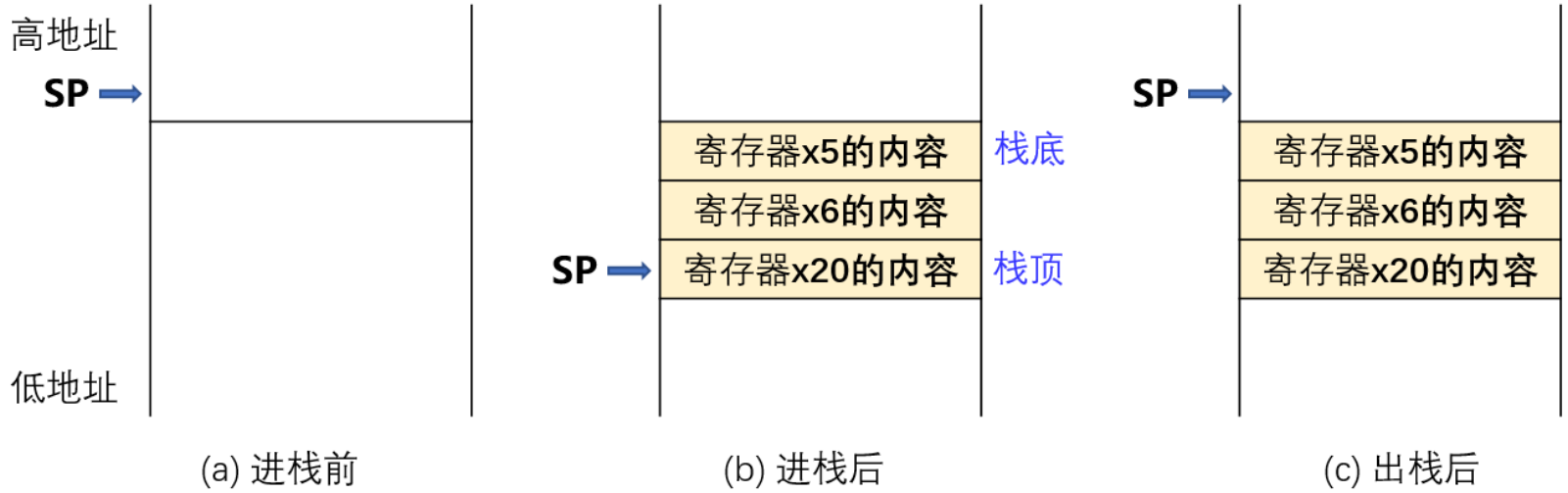


- 按历史惯例，RISC-V中栈按照从高到低的地址顺序增长（即栈底位于高地址，栈顶位于低地址）。
- sp（x2寄存器）是RISC-V中的**栈指针**寄存器，用来保存栈顶的地址。
- 入栈向下移动栈指针(减 sp), 出栈向上移动栈指针(增 sp)。

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

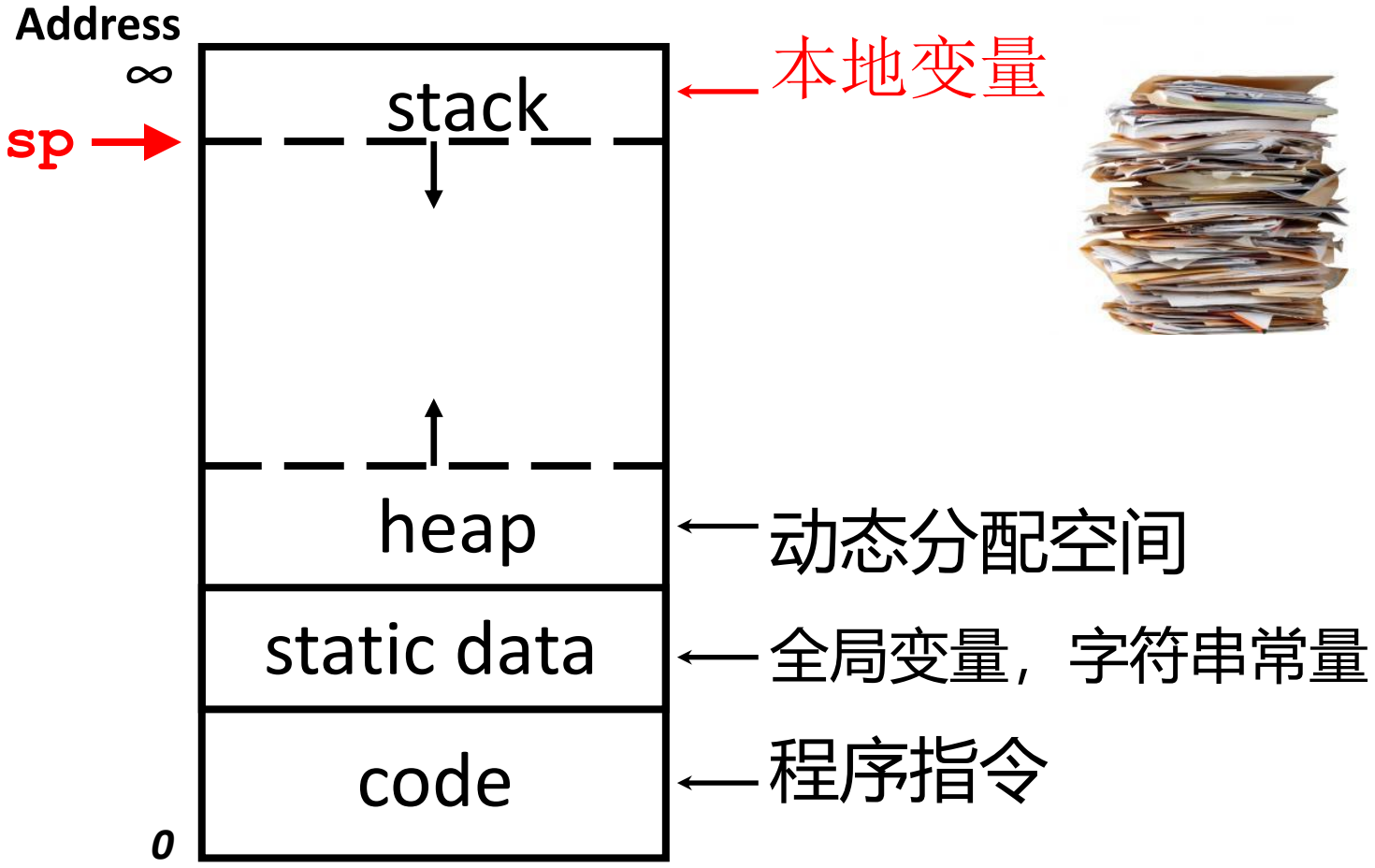
数据“入”栈

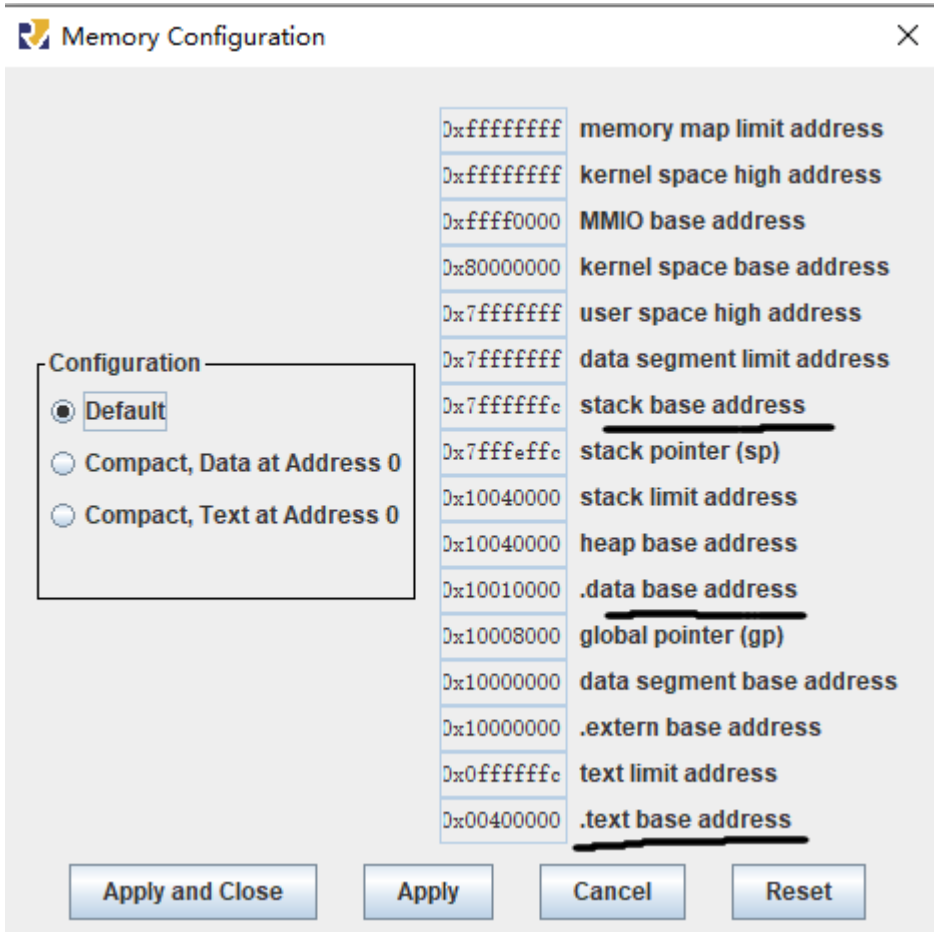
```
addi sp, sp, -24
sd x5, 16(sp)
sd x6, 8(sp)
sd x20,0(sp)
```

数据“出”栈

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

数据的内存排布情况


Address
∞
sp
stack
heap
static data
code
0
本地变量
动态分配空间
全局变量，字符串常量
程序指令


Memory Configuration
Configuration
Default
Compact, Data at Address 0
Compact, Text at Address 0
0xffffffff memory map limit address
0xffffffff kernel space high address
0xffff0000 MMIO base address
0x80000000 kernel space base address
0x7fffffff user space high address
0x7fffffff data segment limit address
0x7ffffffc stack base address
0x7fffeffc stack pointer (sp)
0x10040000 stack limit address
0x10040000 heap base address
0x10010000 .data base address
0x10008000 global pointer (gp)
0x10000000 data segment base address
0x10000000 .extern base address
0x0ffffffc text limit address
0x00400000 .text base address
Apply and Close
Apply
Cancel
Reset

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[ ]，x11保存y[ ]，x19保存i

• 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
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
```

例1：函数嵌套

- sumSquare:

```
        addi sp, sp, -8     # space on stack
        sw ra, 4(sp)        # save return addr
push    sw al, 0(sp)        # save y
        mv al, a0           # mult(x,x)
        jal mult            # call mult
        lw al, 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
  mult:…
```

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

假设 x -> a0, y -> a1

例2：用栈进行递归

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

```
{
    if (n < 1) return f;
    else return n * fact(n - 1);
}
```

  - 参数n在x10中
  - 结果在x10中

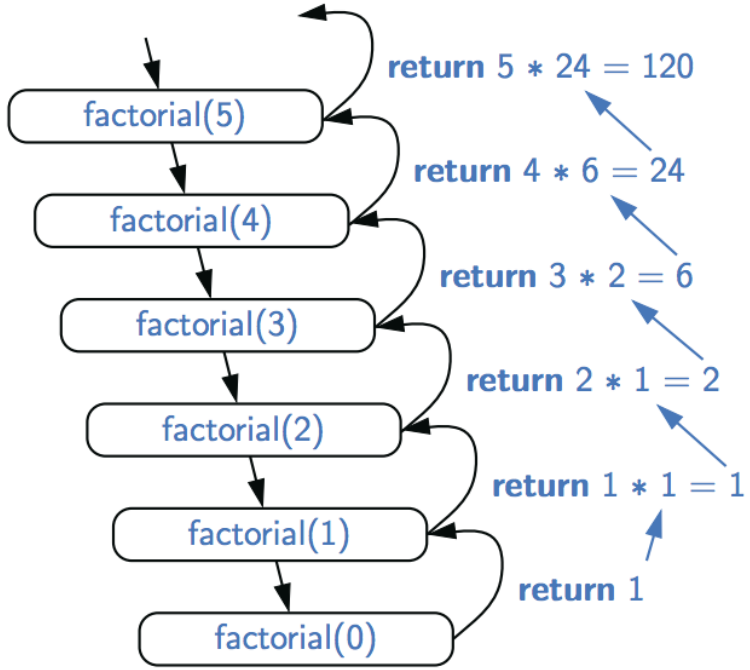
RISC-V code:

```
fact:
      addi sp,sp,-16        #将返回地址和n保存到栈中
      sd x1,8(sp)
      sd x10,0(sp)
      addi x5,x10,-1        # x5 = n - 1
      bge x5,x0,L1          # 若n >= 1，则跳转到L1
      addi x10,x0,1         # 否则，将返值置1
      addi sp,sp,16         # 出栈，此处不需要恢复原值
      jalr x0,0(x1)         # 返回
L1: addi x10,x10,-1         # n = n - 1
      jal x1,fact           # 调用fact(n-1)
     addi x6,x10,0          # 将fact(n - 1)的结果存到x6
     ld x10,0(sp)           # 恢复调用者的n
     ld x1,8(sp)            # 恢复调用者的返回地址
     addi sp,sp,16          # 出栈
     mul x10,x10,x6         # 返回n * fact(n-1)
     jalr x0,0(x1)          # 返回
```

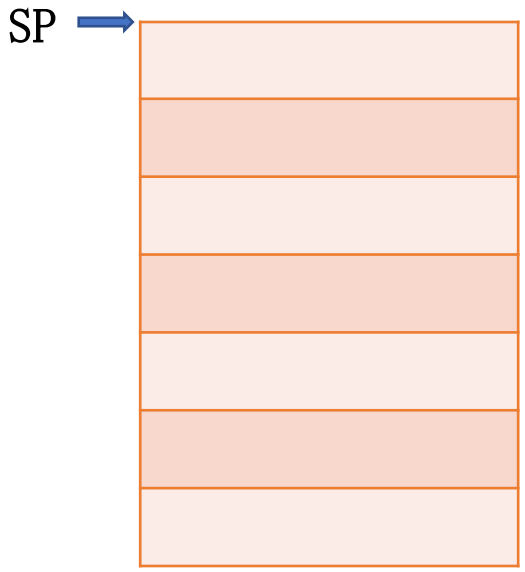
- C代码：long long int fact (long long int n)

```
{
    if (n < 1) return f;
    else return n * fact(n - 1);
}
```

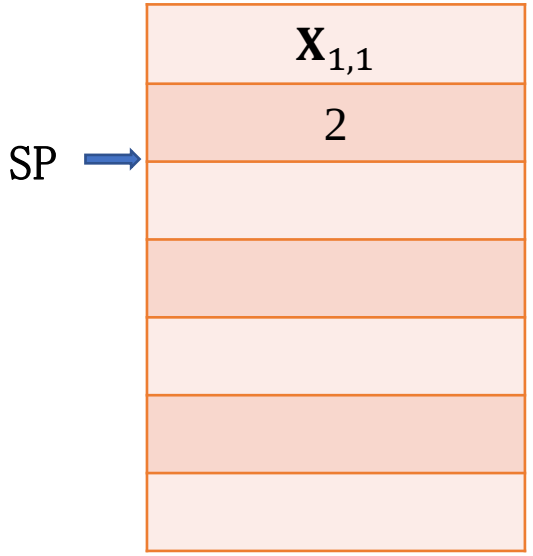
- 参数n在x10中
- 结果在x10中



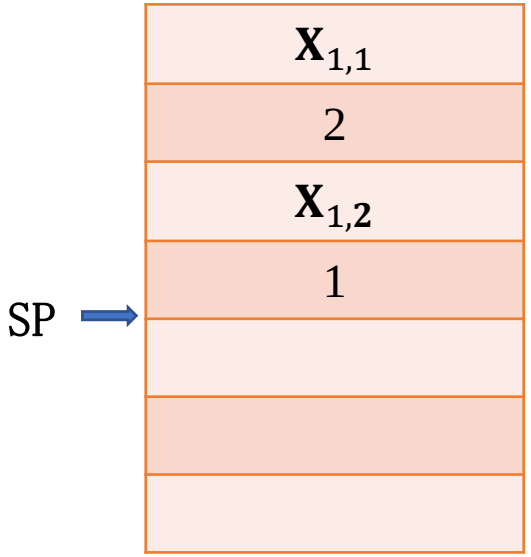
我们以n=2的情况来展示栈中的变化情况



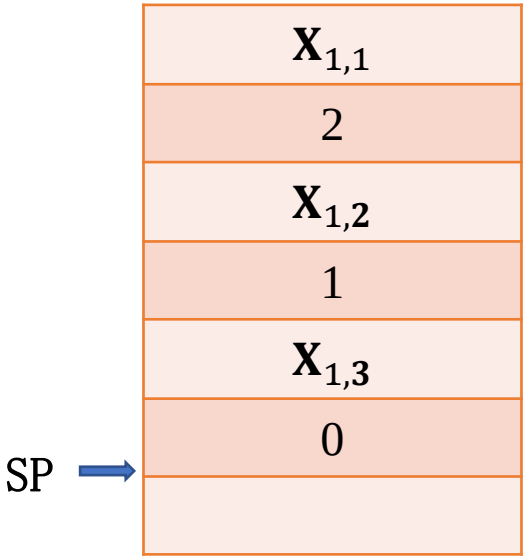
初始状态



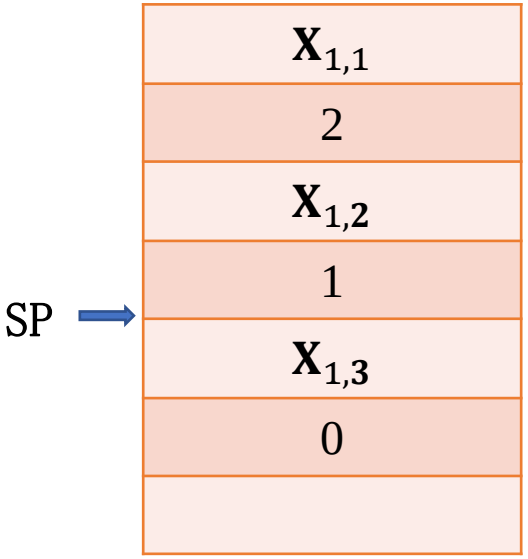
STEP 1



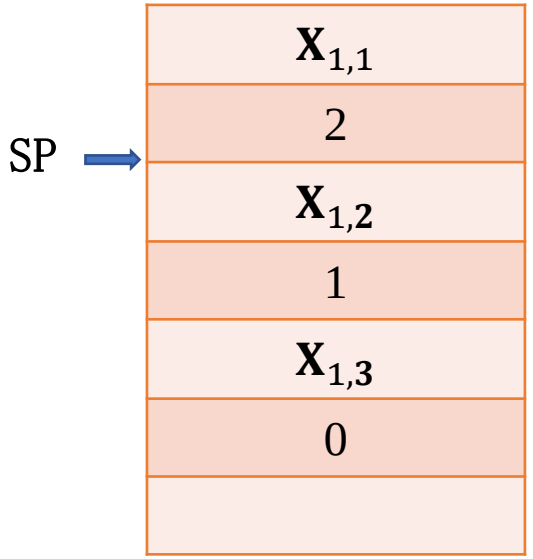
STEP 2



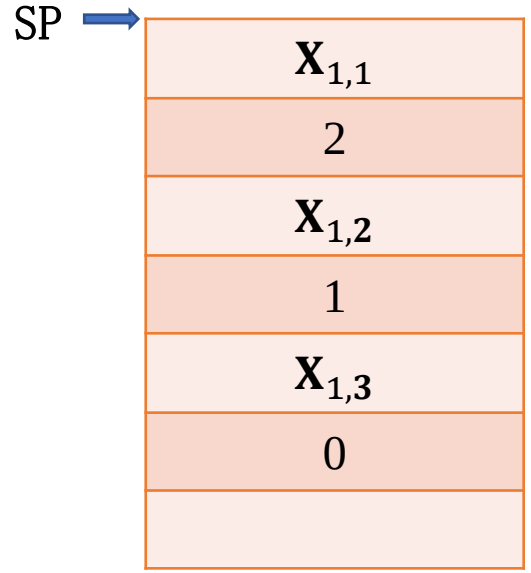
STEP 3



STEP 4



STEP 5



STEP 6最终状态

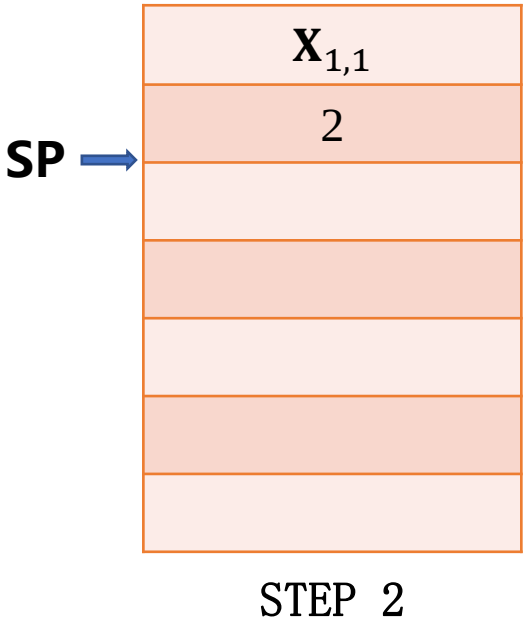
注释

- $X_{1,i}$代表 x1 寄存器存储的不同的返回地址值。
- 2，1，0代表每次递归调用时压入栈中的n的值。
- 在从栈中弹出保存值时，只需要先将需要恢复的值加载到寄存器中，然后向上移动栈指针即可，栈中的数据在下次写入时才会被覆盖。

下面我们按照代码执行的顺序展示

RISC-V code:

```
fact:
    addi sp,sp,-16          #将返回地址和n保存到栈中
    sd x1,8(sp)
    sd x10,0(sp)
    addi x5,x10,-1          # x5 = n - 1
    bge x5,x0,L1            # 若n >= 1，则跳转到L1
    addi x10,x0,1           # 否则，将返值置1
    addi sp,sp,16           # 出栈，此处不需要恢复原值
    jalr x0,0(x1)           # 返回
L1: addi x10,x10,-1         # n = n - 1
    jal x1,fact             # 调用fact(n-1)
   addi x6,x10,0            # 将fact(n - 1)的结果存到x6
   ld x10,0(sp)             # 恢复调用者的n
   ld x1,8(sp)              # 恢复调用者的返回地址
   addi sp,sp,16            # 出栈
   mul x10,x10,x6           # 返回n * fact(n-1)
   jalr x0,0(x1)            # 返回
```



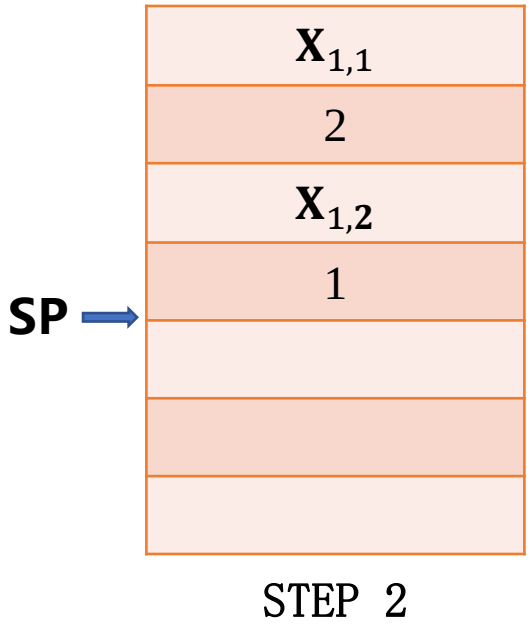
STEP 2

注：红色代表该步中改变栈中内容的代码，蓝色代表栈中数据改变后继续执行的代码。每一步都是从红色代码开始执行。

下面我们按照代码执行的顺序展示

RISC-V code:

```
fact:
    addi sp,sp,-16          #将返回地址和n保存到栈中
    sd x1,8(sp)
    sd x10,0(sp)
    addi x5,x10,-1          # x5 = n - 1
    bge x5,x0,L1            # 若n >= 1，则跳转到L1
    addi x10,x0,1           # 否则，将返值置1
    addi sp,sp,16           # 出栈，此处不需要恢复原值
    jalr x0,0(x1)           # 返回
L1: addi x10,x10,-1         # n = n - 1
    jal x1,fact             # 调用fact(n-1)
   addi x6,x10,0            # 将fact(n - 1)的结果存到x6
   ld x10,0(sp)             # 恢复调用者的n
   ld x1,8(sp)              # 恢复调用者的返回地址
   addi sp,sp,16            # 出栈
   mul x10,x10,x6           # 返回n * fact(n-1)
   jalr x0,0(x1)            # 返回
```



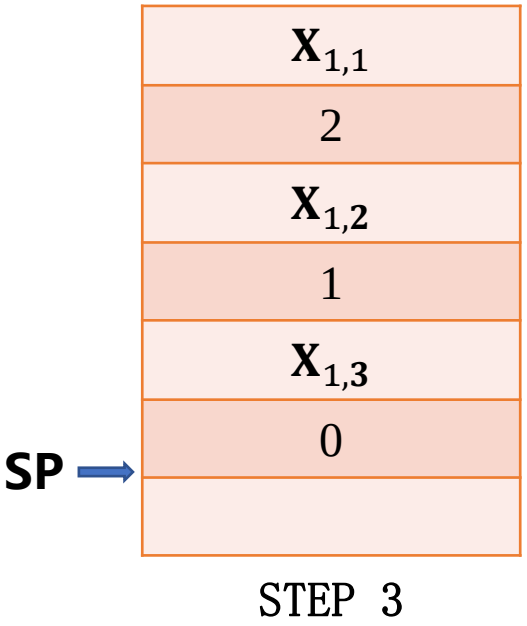
STEP 2

注：红色代表该步中改变栈中内容的代码，蓝色代表栈中数据改变后继续执行的代码。每一步都是从红色代码开始执行。

下面我们按照代码执行的顺序展示

RISC-V code:

```
fact:
    addi sp,sp,-16          #将返回地址和n保存到栈中
    sd x1,8(sp)
    sd x10,0(sp)
    addi x5,x10,-1          # x5 = n - 1
    bge x5,x0,L1            # 若n >= 1，则跳转到L1
    addi x10,x0,1           # 否则，将返值置1
    addi sp,sp,16           # 出栈，此处不需要恢复原值
    jalr x0,0(x1)           # 返回
L1: addi x10,x10,-1         # n = n - 1
    jal x1,fact             # 调用fact(n-1)
    addi x6,x10,0           # 将fact(n - 1)的结果存到x6
    ld x10,0(sp)            # 恢复调用者的n
    ld x1,8(sp)             # 恢复调用者的返回地址
    addi sp,sp,16           # 出栈
    mul x10,x10,x6          # 返回n * fact(n-1)
    jalr x0,0(x1)           # 返回
```



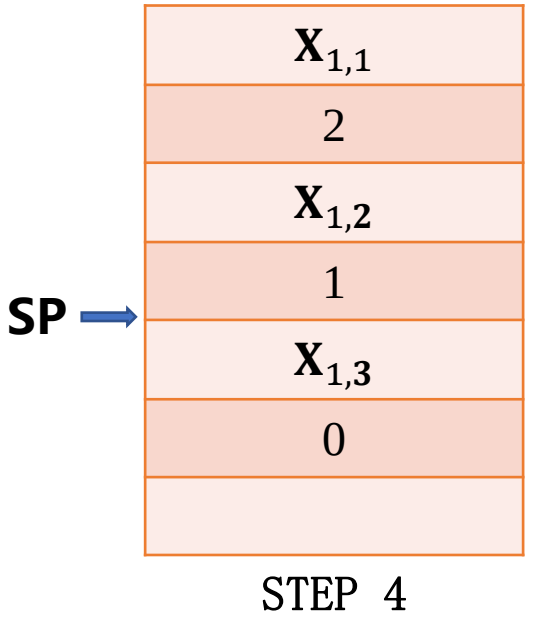
STEP 3

注：红色代表该步中改变栈中内容的代码，蓝色代表栈中数据改变后继续执行的代码。每一步都是从红色代码开始执行。

下面我们按照代码执行的顺序展示

RISC-V code:

```
fact:
    addi sp,sp,-16          #将返回地址和n保存到栈中
    sd x1,8(sp)
    sd x10,0(sp)
    addi x5,x10,-1          # x5 = n - 1
    bge x5,x0,L1            # 若n >= 1，则跳转到L1
    addi x10,x0,1           # 否则，将返值置1
    addi sp,sp,16           # 出栈，此处不需要恢复原值
    jalr x0,0(x1)           # 返回
L1: addi x10,x10,-1         # n = n - 1
    jal x1,fact             # 调用fact(n-1)
    addi x6,x10,0           # 将fact(n - 1)的结果存到x6
    ld x10,0(sp)            # 恢复调用者的n
    ld x1,8(sp)             # 恢复调用者的返回地址
    addi sp,sp,16           # 出栈
    mul x10,x10,x6          # 返回n * fact(n-1)
    jalr x0,0(x1)           # 返回
```



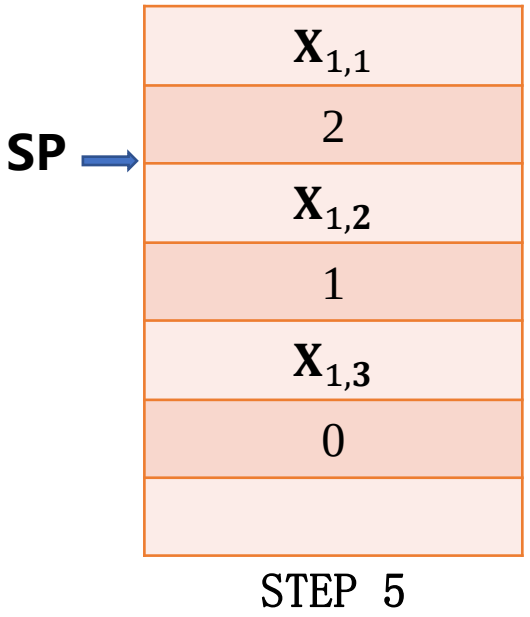
STEP 4

注：红色代表该步中改变栈中内容的代码，蓝色代表栈中数据改变后继续执行的代码。每一步都是从红色代码开始执行。

下面我们按照代码执行的顺序展示

RISC-V code:

```
fact:
    addi sp,sp,-16          #将返回地址和n保存到栈中
    sd x1,8(sp)
    sd x10,0(sp)
    addi x5,x10,-1          # x5 = n - 1
    bge x5,x0,L1            # 若n >= 1，则跳转到L1
    addi x10,x0,1           # 否则，将返值置1
    addi sp,sp,16           # 出栈，此处不需要恢复原值
    jalr x0,0(x1)           # 返回
L1: addi x10,x10,-1         # n = n - 1
    jal x1,fact             # 调用fact(n-1)
    addi x6,x10,0           # 将fact(n - 1)的结果存到x6
    ld x10,0(sp)            # 恢复调用者的n
    ld x1,8(sp)             # 恢复调用者的返回地址
    addi sp,sp,16           # 出栈
    mul x10,x10,x6          # 返回n * fact(n-1)
    jalr x0,0(x1)           # 返回
```



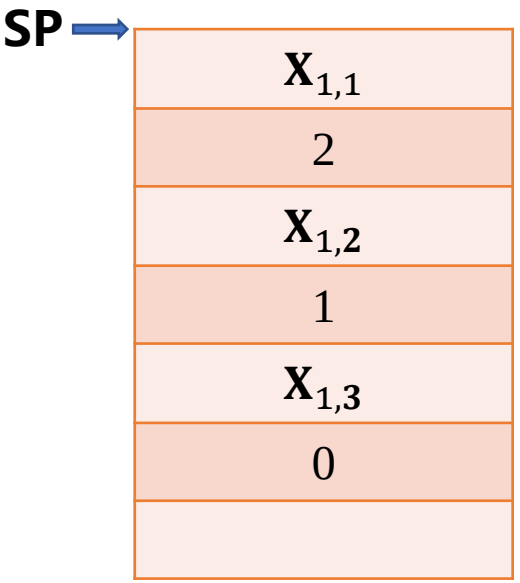
STEP 5

注：红色代表该步中改变栈中内容的代码，蓝色代表栈中数据改变后继续执行的代码。每一步都是从红色代码开始执行。

下面我们按照代码执行的顺序展示

RISC-V code:

```
fact:
    addi sp,sp,-16          #将返回地址和n保存到栈中
    sd x1,8(sp)
    sd x10,0(sp)
    addi x5,x10,-1          # x5 = n - 1
    bge x5,x0,L1            # 若n >= 1，则跳转到L1
    addi x10,x0,1           # 否则，将返值置1
    addi sp,sp,16           # 出栈，此处不需要恢复原值
    jalr x0,0(x1)           # 返回
L1: addi x10,x10,-1         # n = n - 1
    jal x1,fact             # 调用fact(n-1)
    addi x6,x10,0           # 将fact(n - 1)的结果存到x6
    ld x10,0(sp)            # 恢复调用者的n
    ld x1,8(sp)             # 恢复调用者的返回地址
    addi sp,sp,16           # 出栈
    mul x10,x10,x6          # 返回n * fact(n-1)
    jalr x0,0(x1)           # 返回
```



STEP 6最终状态

注：红色代表该步中改变栈中内容的代码，蓝色代表栈中数据改变后继续执行的代码。每一步都是从红色代码开始执行。执行完jalr后，从fact返回主函数。

第三章 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）**：GNU汇编中，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** 可以有以下多种类型：
    - 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：

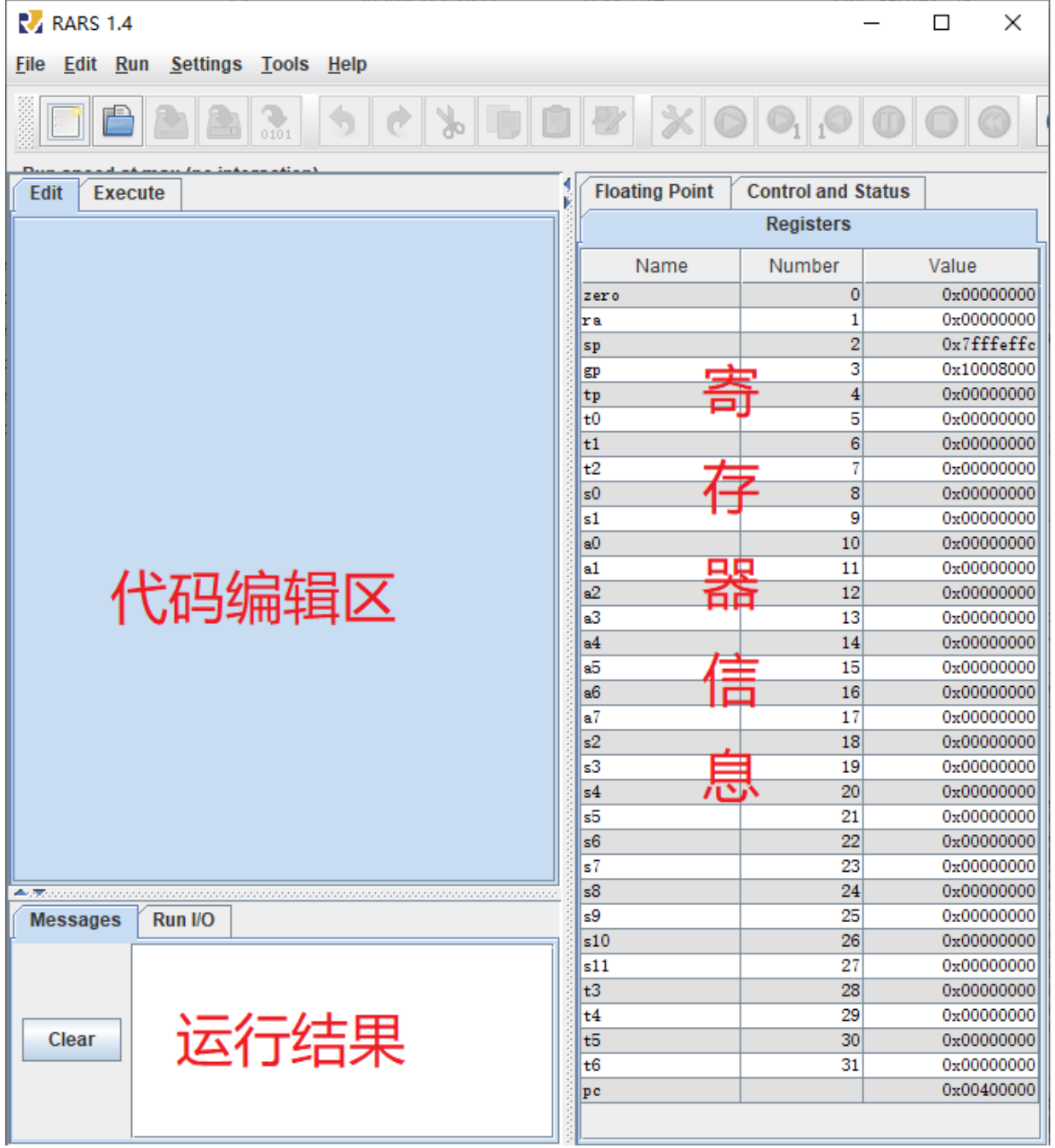

RARS 1.4
File Edit Run Settings Tools Help
Edit Execute
代码编辑区
Messages Run I/O
Clear
运行结果
Floating Point Control and Status
Registers
寄存器信息
Name Number Value
zero 0 0x00000000
ra 1 0x00000000
sp 2 0x7fffeffc
gp 3 0x10008000
tp 4 0x00000000
t0 5 0x00000000
t1 6 0x00000000
t2 7 0x00000000
s0 8 0x00000000
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
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 0x00400000

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：

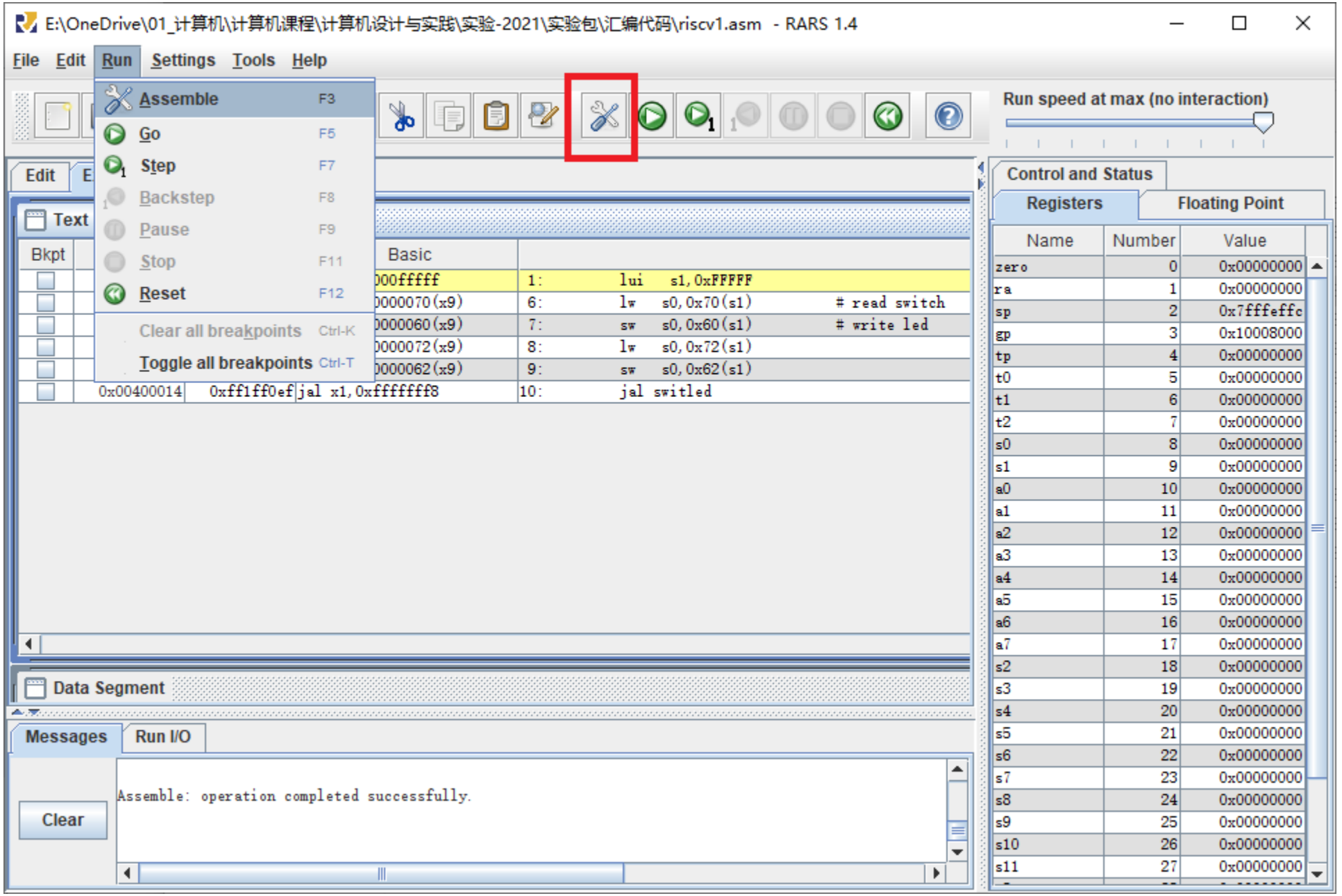

E:\OneDrive\01_计算机\计算机课程\计算机设计与实践\实验-2021\实验包\汇编代码\riscv1.asm - RARS 1.4
File Edit Run Settings Tools Help
Assemble F3
Go F5
Step F7
Backstep F8
Pause F9
Stop F11
Reset F12
Clear all breakpoints Ctrl-K
Toggle all breakpoints Ctrl-T
Run speed at max (no interaction)
Edit
Text
Bkpt
Basic
1: lui s1,0xFFFFF
6: lw s0,0x70(s1) # read switch
7: sw s0,0x60(s1) # write led
8: lw s0,0x72(s1)
9: sw s0,0x62(s1)
10: jal switled
0x00400014 0xff1ff0ef jal x1,0xfffffff8
Data Segment
Messages
Run I/O
Clear
Assemble: operation completed successfully.
Control and Status
Registers
Floating Point
Name Number Value
zero 0 0x00000000
ra 1 0x00000000
sp 2 0x7fffeffc
gp 3 0x10008000
tp 4 0x00000000
t0 5 0x00000000
t1 6 0x00000000
t2 7 0x00000000
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
s7 23 0x00000000
s8 24 0x00000000
s9 25 0x00000000
s10 26 0x00000000
s11 27 0x00000000

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。

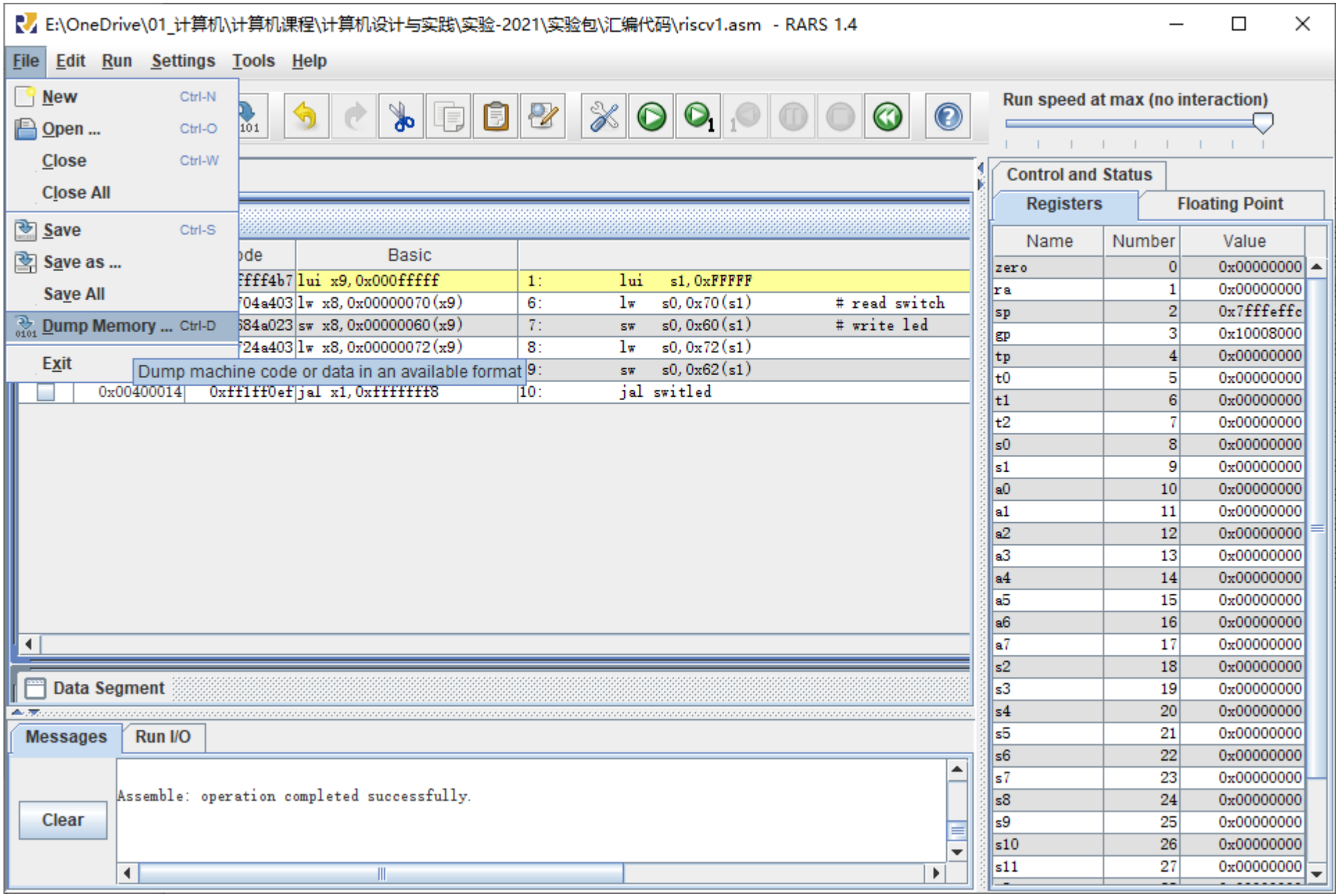

E:\OneDrive\01_计算机\计算机课程\计算机设计与实践\实验-2021\实验包\汇编代码\riscv1.asm - RARS 1.4
File Edit Run Settings Tools Help
New Ctrl-N
Open ... Ctrl-O
Close Ctrl-W
Close All
Save Ctrl-S
Save as ...
Save All
Dump Memory ... Ctrl-D
Exit
Dump machine code or data in an available format
Run speed at max (no interaction)
Basic
lui x9,0x000fffff 1: lui s1,0xFFFFF
lw x8,0x00000070(x9) 6: lw s0,0x70(s1) # read switch
sw x8,0x00000060(x9) 7: sw s0,0x60(s1) # write led
lw x8,0x00000072(x9) 8: lw s0,0x72(s1)
9: sw s0,0x62(s1)
0x00400014 0xfff1ff0ef jal x1,0xfffffff8 10: jal switled
Control and Status
Registers
Floating Point
Name Number Value
zero 0 0x00000000
ra 1 0x00000000
sp 2 0x7fffeffc
gp 3 0x10008000
tp 4 0x00000000
t0 5 0x00000000
t1 6 0x00000000
t2 7 0x00000000
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
s7 23 0x00000000
s8 24 0x00000000
s9 25 0x00000000
s10 26 0x00000000
s11 27 0x00000000
Data Segment
Messages
Run I/O
Clear
Assemble: operation completed successfully.

图3 从菜单导出机器码

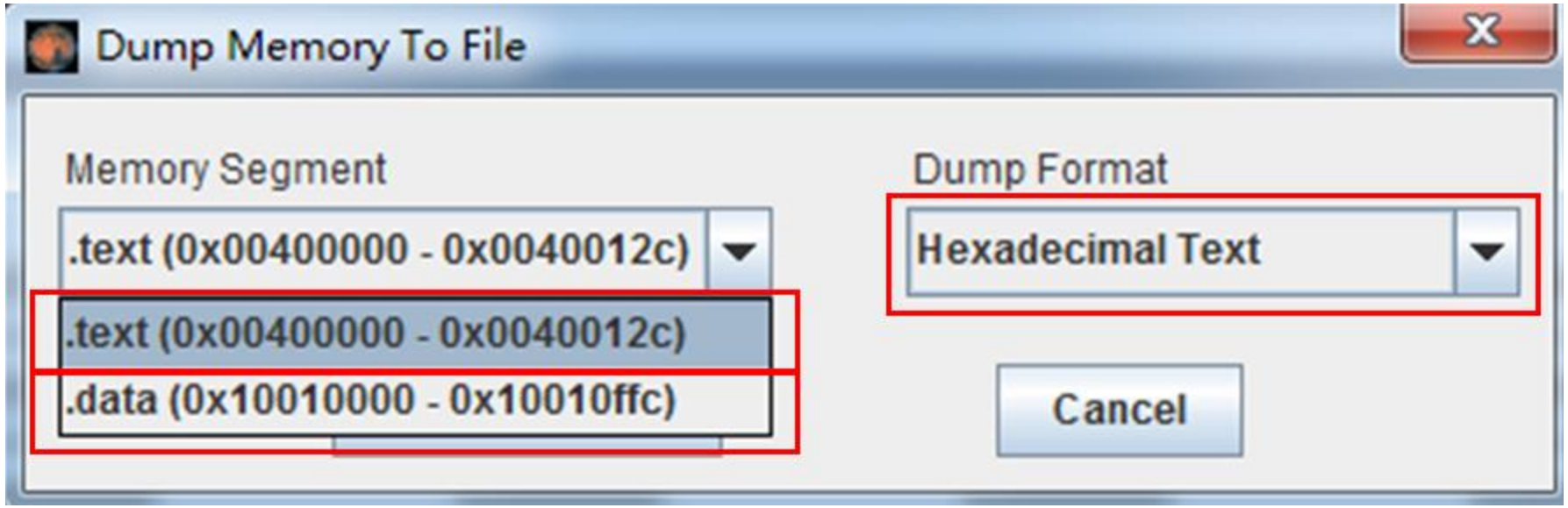


图4 从菜单导出机器码

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

Edit | Execute

riscv1.asm

```
1  addi x10, x0, 21
2  addi x11, x0, 20
3  jal x1, swap
4
5  swap:
6    add x6, x0, x10
7    add x10, x0, x11
8    add x11, x0, x6
9    jalr x0,0(x1)      # 返回调用函数
10
```

Line: 4 Column: 1 ☑ Show Line Numbers

Messages | Run I/O

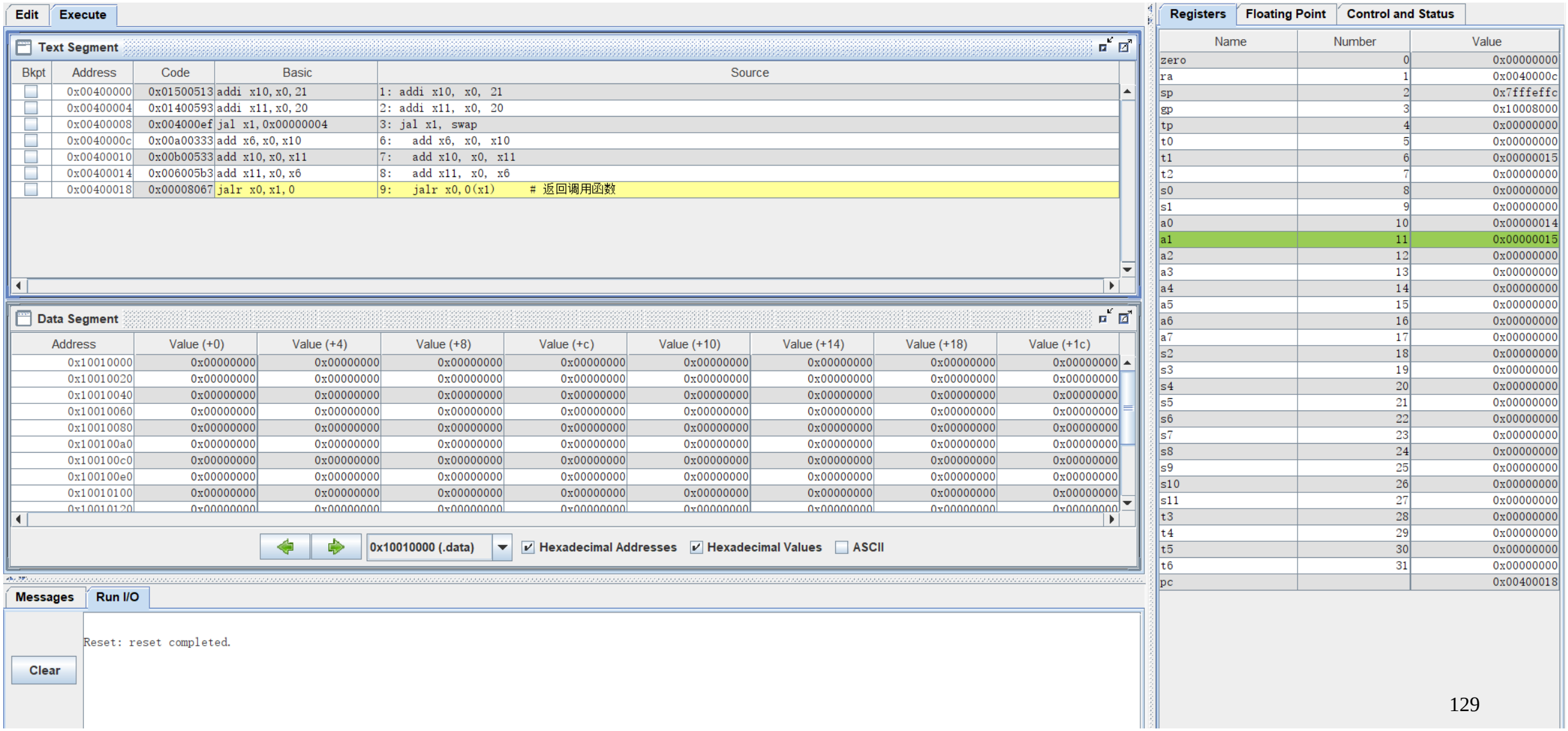
Clear

Reset: reset completed.

Registers | Floating Point | Control and Status

Name	Number	Value
zero	0	0x00000000
ra	1	0x0040000c
sp	2	0x7fffeffc
gp	3	0x10008000
tp	4	0x00000000
t0	5	0x00000000
t1	6	0x00000015
t2	7	0x00000000
s0	8	0x00000000
s1	9	0x00000000
a0	10	0x00000014
a1	11	0x00000015
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
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		0x00400018

运行一个简单的数据交换例子



Edit | Execute

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x01500513	addi x10,x0,21	1: addi x10, x0, 21
	0x00400004	0x01400593	addi x11,x0,20	2: addi x11, x0, 20
	0x00400008	0x004000ef	jal x1,0x00000004	3: jal x1, swap
	0x0040000c	0x00a00333	add x6,x0,x10	6: add x6, x0, x10
	0x00400010	0x00b00533	add x10,x0,x11	7: add x10, x0, x11
	0x00400014	0x006005b3	add x11,x0,x6	8: add x11, x0, x6
	0x00400018	0x00008067	jalr x0,x1,0	9: jalr x0,0(x1) # 返回调用函数

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010100	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010120	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

0x10010000 (.data) ☑ Hexadecimal Addresses ☑ Hexadecimal Values ☐ ASCII

Registers | Floating Point | Control and Status

Name	Number	Value
zero	0	0x00000000
ra	1	0x0040000c
sp	2	0x7fffeffc
gp	3	0x10008000
tp	4	0x00000000
t0	5	0x00000000
t1	6	0x00000015
t2	7	0x00000000
s0	8	0x00000000
s1	9	0x00000000
a0	10	0x00000014
a1	11	0x00000015
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
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		0x00400018

Messages | Run I/O

Clear

Reset: reset completed.

```
fact:
    addi sp,sp,-8
     xxxxx
```

```
fact: addi sp,sp,-8
     xxxxx
```

```
1  addi x10,  x0,0
2  addi x10,  x10,10
3  fact:  addi  sp, sp, -8
4         sw  x1, 4(sp)
5         sw  x10, 0(sp)
6         addi  x5, x10, -1
7         bge  x5, x0, L1
8         addi  x10, x0, 1
9         addi  sp, sp, 8
10        jalr  x0, 0(x1)
11 L1:  addi  x10, x10, -1
12       jal  x1, fact
13       addi  x6, x10, 0
14       lw  x10, 0(sp)
15       lw  x1, 4(sp)
16       addi  sp, sp, 8
17       mul  x10, x10, x6
18       jalr  x0, 0(x1)
```

```
1  addi x10,  x0,0
2  addi x10,  x10,10
3  fact:
4         addi  sp, sp, -8
5         sw  x1, 4(sp)
6         sw  x10, 0(sp)
7         addi  x5, x10, -1
8         bge  x5, x0, L1
9         addi  x10, x0, 1
10        addi  sp, sp, 8
11        jalr  x0, 0(x1)
12 L1:
13       addi  x10, x10, -1
14       jal  x1, fact
15       addi  x6, x10, 0
16       lw  x10, 0(sp)
17       lw  x1, 4(sp)
18       addi  sp, sp, 8
19       mul  x10, x10, x6
20       jalr  x0, 0(x1)
```

Text Segment

Bkpt	Address	Code	Basic	
	0x00400000	0x00000513	addi x10, x0, 0	1: addi x10, x0, 0
	0x00400004	0x00a50513	addi x10, x10, 10	2: addi x10, x10, 10
	0x00400008	0xff810113	addi x2, x2, 0xfffffff8	3: fact: addi sp, sp, -8
	0x0040000c	0x00112223	sw x1, 4(x2)	4: [redacted] sw x1, 4(sp)
	0x00400010	0x00a12023	sw x10, 0(x2)	5: sw x10, 0(sp)
	0x00400014	0xfff50293	addi x5, x10, 0xffffffff	6: addi x5, x10, -1
	0x00400018	0x0002d863	bge x5, x0, 0x00000010	7: bge x5, x0, L1
	0x0040001c	0x00100513	addi x10, x0, 1	8: addi x10, x0, 1
	0x00400020	0x00810113	addi x2, x2, 8	9: addi sp, sp, 8
	0x00400024	0x00008067	jalr x0, x1, 0	10: jalr x0, 0(x1)
	0x00400028	0xfff50513	addi x10, x10, 0xffffffff	11: L1: addi x10, x10, -1
	0x0040002c	0xfddff0ef	jal x1, 0xfffffffdc	12[redacted] jal x1, fact
	0x00400030	0x00050313	addi x6, x10, 0	13: addi x6, x10, 0
	0x00400034	0x00012503	lw x10, 0(x2)	14: lw x10, 0(sp)
	0x00400038	0x00412083	lw x1, 4(x2)	15: lw x1, 4(sp)
	0x0040003c	0x00810113	addi x2, x2, 8	16: addi sp, sp, 8
	0x00400040	0x02650533	mul x10, x10, x6	17: mul x10, x10, x6
	0x00400044	0x00008067	jalr x0, x1, 0	18: jalr x0, 0(x1)

Edit Execute

Text Segment

Bkpt	Address	Code	Basic	
	0x00400000	0x00000513	addi x10, x0, 0	1: addi x10, x0, 0
	0x00400004	0x00a50513	addi x10, x10, 10	2: addi x10, x10, 10
	0x00400008	0xff810113	addi x2, x2, 0xfffffff8	4: addi sp, sp, -8
	0x0040000c	0x00112223	sw x1, 4(x2)	5: sw x1, 4(sp)
	0x00400010	0x00a12023	sw x10, 0(x2)	6: sw x10, 0(sp)
	0x00400014	0xfff50293	addi x5, x10, 0xffffffff	7: addi x5, x10, -1
	0x00400018	0x0002d863	bge x5, x0, 0x00000010	8: bge x5, x0, L1
	0x0040001c	0x00100513	addi x10, x0, 1	9: addi x10, x0, 1
	0x00400020	0x00810113	addi x2, x2, 8	10: addi sp, sp, 8
	0x00400024	0x00008067	jalr x0, x1, 0	11: jalr x0, 0(x1)
	0x00400028	0xfff50513	addi x10, x10, 0xffffffff	13: addi x10, x10, -1
	0x0040002c	0xfddff0ef	jal x1, 0xfffffffdc	14: jal x1, fact
	0x00400030	0x00050313	addi x6, x10, 0	15: addi x6, x10, 0
	0x00400034	0x00012503	lw x10, 0(x2)	16: lw x10, 0(sp)
	0x00400038	0x00412083	lw x1, 4(x2)	17: lw x1, 4(sp)
	0x0040003c	0x00810113	addi x2, x2, 8	18: addi sp, sp, 8
	0x00400040	0x02650533	mul x10, x10, x6	19: mul x10, x10, x6
	0x00400044	0x00008067	jalr x0, x1, 0	20: jalr x0, 0(x1)