

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？

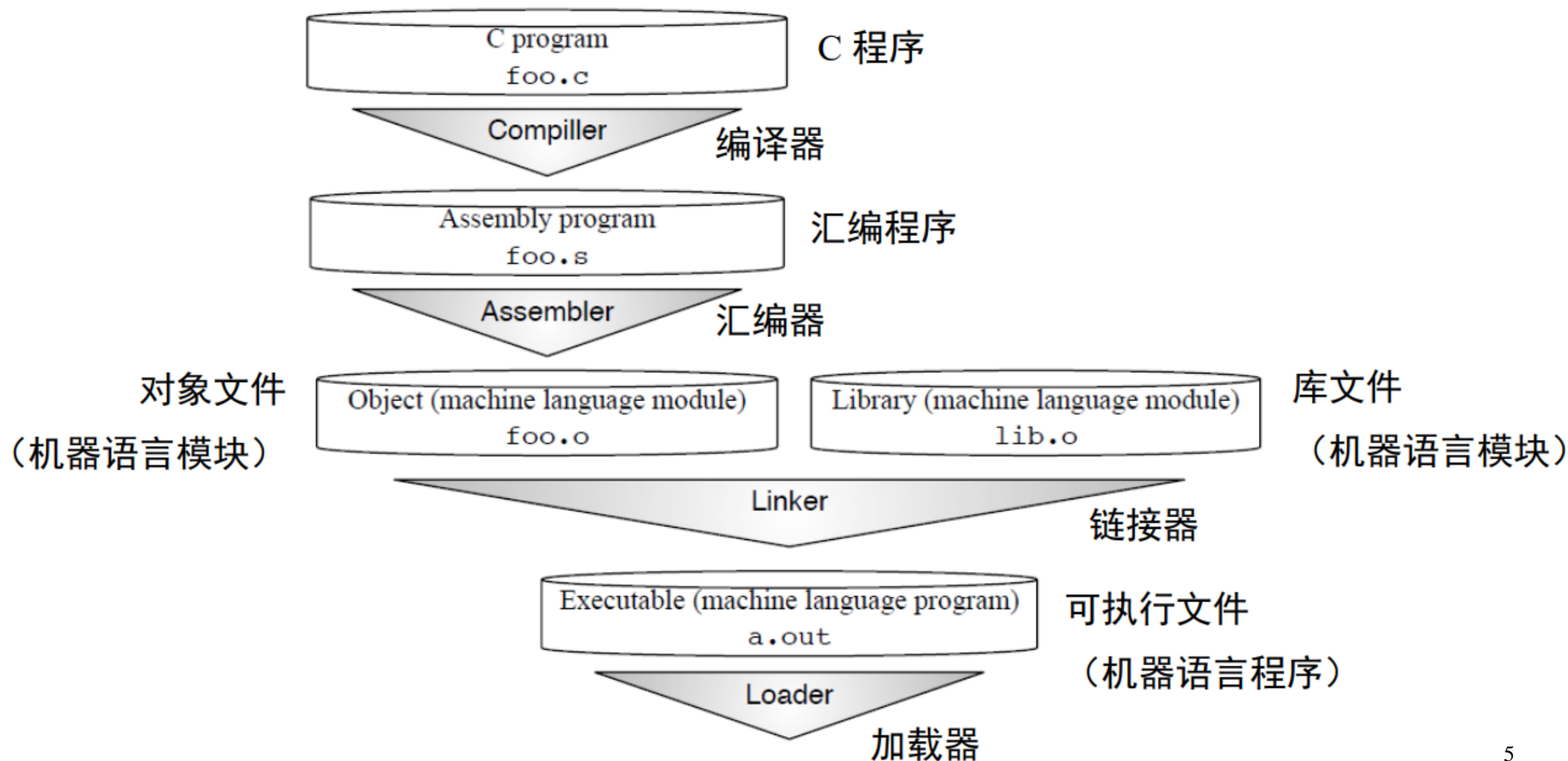
- 汇编语言（**Assembly Language**）是一种“低级”语言，直接接触最底层的硬件，需要对底层硬件非常熟悉才能编写出高效的汇编程序。
- 汇编语言属于第二代计算机语言，用一些容易理解和记忆的字母，单词来代替一个特定的指令。在汇编语言中，用助记符代替机器指令的操作码，用地址符号或标号代替指令或操作数的地址。
- 例如：用“ADD”代表数字逻辑上的加减，“MOV”代表数据传递等等。

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



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 rd, rs1, rs2 1个操作码, 3个操作数

- op 操作的名字 (**o**peration)
- rd 目标寄存器 (**r**egister **d**estination)
- rs1 第一个源操作数寄存器 (**r**egister **S**ource)
- rs2 第二个源操作数寄存器 (**r**egister **S**ource)
- 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, $x0 \sim x31$
- 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
- 读写操作以字节 (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 (8字节)₁₂ 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(Function Argument) 以及 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 (temporary register)

RISC-V汇编语言指令分类

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

伪指令: 方便程序员使用的指令，是以上指令的变化或组合。

RISC-V 汇编语言

类别	指令	示例	含义	注释
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

卡片2
需要记忆

卡片2 需要记忆

移位操作	逻辑左移	sll x5, x6, x7	$x5 = x6 \ll x7$	按寄存器给定位数左移
	逻辑右移	srl x5, x6, x7	$x5 = x6 \gg x7$	按寄存器给定位数右移
	算术右移	sra x5, x6, x7	$x5 = x6 \gg x7$	按寄存器给定位数算术右移
	逻辑左移立即数	slli x5, x6, 3	$x5 = x6 \ll 3$	根据立即数给定位数左移
	逻辑右移立即数	srl i x5, x6, 3	$x5 = x6 \gg 3$	根据立即数给定位数右移
	算术右移立即数	srai x5, x6, 3	$x5 = x6 \gg 3$	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**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 $a \rightarrow x1$, $b \rightarrow x2$, $c \rightarrow x3$, $d \rightarrow x4$, $e \rightarrow x5$ 。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 rd, rs1, imm**
 - 操作码的最后一个字符为i的, 会将第二个源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** # $R[rd] = R[rs1] + \text{sext}(\text{immediate})$:
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**subi**指令，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)

- 问题：尽管没有subi，但**为什么有sub**？

执行 `addi x10, x0, -1`
`x10 = ?`

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

提交

执行 `addi x10,x0,-1`

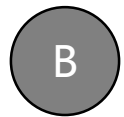
`slti x11,x10,1`

`x11=?`



A

1



B

0

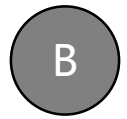
提交

执行 `addi x10,x0,-1`
`sltiu x12,x10,1`

`x12=?`



0



1

提交

RISC-V 乘法与除法指令

- **乘法: mul**

- 使用语法: `mul rd, rs1, rs2`
- 功能描述: $R[rd] = R[rs1] \times R[rs2]$ 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: $R[rd] = R[rs1] \div R[rs2]$ 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64位数的积是128位。
- 为了得到有符号或无符号的128位积，RISC-V中有四个指令。
 - 要得到整数64位乘积（128位中的低64位）就用**mul指令**。
 - 要得到**高**64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

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128位积，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，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

逻辑运算指令

- 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	$R[rd] = R[rs1] \wedge R[rs2]$	按位“异或”操作，结果写入寄存器rd
xori rd, rs1, imm	$R[rd] = R[rs1] \wedge sext(imm)$	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: $a \rightarrow x1$, $b \rightarrow x2$, $c \rightarrow x3$

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 $111\dots11_2$ 使用 `xori`

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

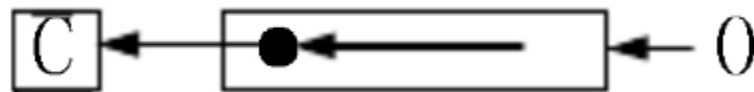
移位运算指令

- 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

- 逻辑左移相当于乘以2

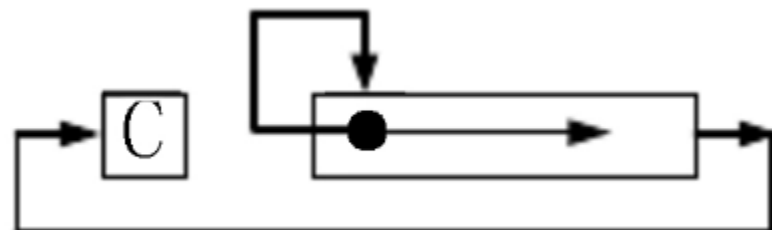
- 左移右边补0
- 左移操作比乘法更快



- 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RV**64**), 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`

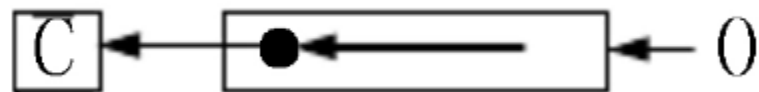
- `srai x10, x10, 4` # `x10=x10>>4`

- 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	1 1 11111111
.....	正确	0xx.....x	1 1 xxx.....x
-64	11000000	10000000	10000000	正确	01100000	1 1 100000
-65	10111111	01111110	11111110	错误	01011111	1 1 011111
-66	10111110	01111100	11111100	错误	01011111	1 1 011111
-128	10000000	00000000	10000000	错误	01000000	1 1 000000 ⁴¹

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 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

0 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

计算器

程序员

HEX 86
DEC -122
OCT 206
BIN 1000 0110

BYTE MS

按位 位移

算术移位
逻辑移位
旋转循环移位
带进位旋转循

>> CE <>
) % ÷
8 9 ×
4 5 6 -
E 1 2 3 +
F +/- 0 . =

计算器

程序员

**-61 Lsh 1 =
-122**

HEX 86
DEC -122
OCT 206
BIN 1000 0110

BYTE MS

按位 位移

算术移位
逻辑移位
旋转循环移位
带进位旋转循

>> CE <>
) % ÷
8 9 ×
4 5 6 -
E 1 2 3 +
F +/- 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等**
- 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；而其它的大量的数据结构，例如数组会映射到内存中
- 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**类指令：数据从寄存器到内存
 - **Load**类指令：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: (存地址的) 基地址寄存器 (base address register)
- offset: 字节偏移量, 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

- l**w** reg, offset(bAddrReg) # 从存储器 **取** 字到寄存器

- s**w** 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
 - lh rd, offset(rs1) #load half word
 - lw rd, offset(rs1) #load word
 - ld rd, offset(rs1) #load double 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/双字写入主存：保存最右端的8/16/32/64位
 - sb rs2, offset(rs1)
 - sh rs2, offset(rs1)
 - sw rs2, offset(rs1)
 - sd rs2, offset(rs1)

注意：对于RV32则扩展为32位，对于RV64则扩展为64位

数据传输指令示例

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

lw x10, 12(x3)

add x10, x2, x10

sw x10, 40(x3)

(in RISC-V)

x10 = A[3]

x10 = A[3]+a

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 $A[] \rightarrow x15$, $h \rightarrow x12$

`ld x10, 24(x15)` # $x15$ 为 $A[0]$ 地址 (in RISC-V64)

`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**来操作字节数据

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）

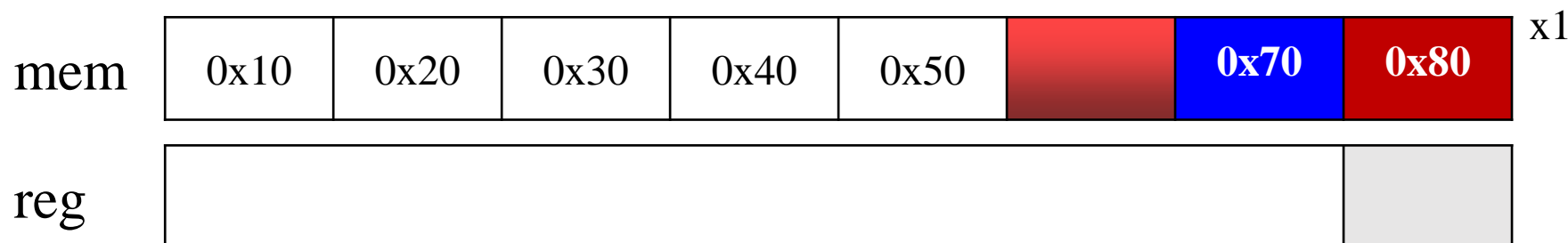
假设某地址储存数据为 0x00000180



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0x1020304050607080
(RV64) lb x11, **1**(x1) #x11=0x0000000000000000**70**
 lb x12, **0**(x1) #x12=0xFFFFFFFFFFFFFFFF**80**
 sb x12, **2**(x1) #mem[x1]=0x1020304050**80**7080



半字（2字节）数据传输指令

- **lh** reg, offset(bAddr) #load half word
- **sh** reg, offset(bAddr) #store half word
- **lhu** 无符号半字传输指令
 - offset(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**

类似地, lbu、lhu、lwu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
 - lh rd, offset(rs1) #load half word
 - lw rd, offset(rs1) #load word
 - ld rd, offset(rs1) #load double 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/双字写入主存：保存最右端的8/16/32/64位
 - sb rs2, offset(rs1)
 - sh rs2, offset(rs1)
 - sw rs2, offset(rs1)
 - sd rs2, offset(rs1)

注意：对于RV32则扩展为32位，对于RV64则扩展为64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 `0x00000080`
- ☒ B `0x00000001`
- ☐ C `0x10000001`
- ☐ D `0x11111111`

提交

(RV32I) 若 $\text{mem}[\text{x1}] = 0\text{x}12000180$ ，执行指令 `lb x12, 0(x1)`，寄存器x12中存储的内容为：

- ☒ A `0xFFFFFFFF80`
- ☐ B `0x00000080`
- ☐ C `0x00000008`
- ☐ D `0x00000012`

提交

(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

提交

(RV32I) , 若 $\text{mem}[\text{x1}] = 0\text{x}12000180$, 执行如下指令后
 $\text{mem}[\text{x1}]$ 的数据为:

```
addi x12, x0, -0x82  
sb x12, 2(x1)
```

- ☐ A 0x12820180
- ☐ B 0x12007E80
- ☐ C 0x12008280
- ☒ D 0x127E0180

提交

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等
- **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 rd, rs1, rs2`
 - 如果rs1中的值 < rs2中的值, $rd = 1$, 否则 $rd=0$
- **Set Less Than Immediate** (slti)
 - `slti rd, rs1, imm`
 - 如果rs1中的值 < 立即数imm, $rd=1$, 否则 $rd=0$

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 rd, rs1, rs2: unsigned comparison

sltiu rd, rs1, imm: unsigned comparison against constant

- 例:

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

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

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


执行 `addi x10,x0,-5`

`sltiu x12,x10,0x1`

`x12`中存储的内容的十六进制表示为_____? (RV64)

- ☒ A 0x0000000000000000
- ☐ B 0x0000000000000001
- ☐ C 0x1111111111111111
- ☐ D 0x1111111111111110

提交

执行下列指令序列后，寄存器x10的值是多少？

☒ A 2

☐ B 5

☐ C 1

☐ D 4

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

```
addi x11, x0, 3
```

```
andi x11, x11, 1
```

```
add x10, x10, x11
```

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（C语言可以用“{”定义语句块），但可通过标记（Label）定义语句块起始
 - 标记后面加一个冒号（例如 **Loop:**）
 - 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（立即数）所指的语句执行，否则按序执行。
- `beq rs1, rs2, label` #branch if **e**qual
 - 如果`rs1`中的值`=``rs2`中的值, 程序跳转到`label`处继续执行
- `bne rs1, rs2, label` #branch if **n**ot **e**qual
 - 如果`R[rs1]` **不等于** `R[rs2]`, 程序跳转到`label`处继续执行
- `blt rs1, rs2, label` #branch if **l**ess **t**han
 - 如果`R[rs1]` **<** `R[rs2]`, 程序跳转到`label`处继续执行
- `bge rs1, rs2, label` #branch if **g**reater than or **e**qual
 - 如果`R[rs1]` **>=** `R[rs2]`, 程序跳转到`label`处继续执行

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

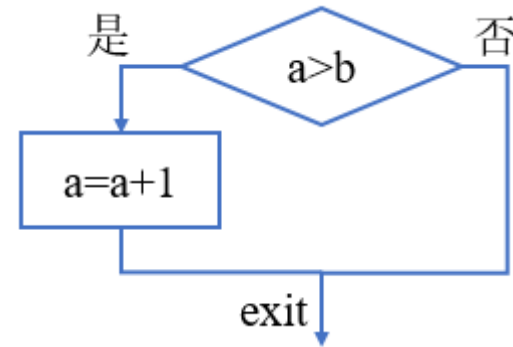
- 有符号和无符号在3个上下文环境中

—Signed	vs.	unsigned bit extension 符号扩展
• lb, lh	vs.	lb <u>u</u> , lh <u>u</u>
—Signed	vs.	unsigned comparison 比较
• slt, slti	vs.	slt <u>u</u> , slti <u>u</u>
—Signed	vs.	unsigned branch 比较
• blt, bge	vs.	blt <u>u</u> , bge <u>u</u>

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:

用RV64I汇编实现C语言while循环结构

C代码:

```
long long int save[100];
```

.....

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

...

假设: i存储在x22中, k
存储在x24中, save数组
元素的地址保存在x25中。

RISC-V64I 代码:

Loop: slli x10, x22, 3 #8*i

add x10, x10, x25 #save[]+8*i

ld x9, 0(x10) #save[i]—>x9

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

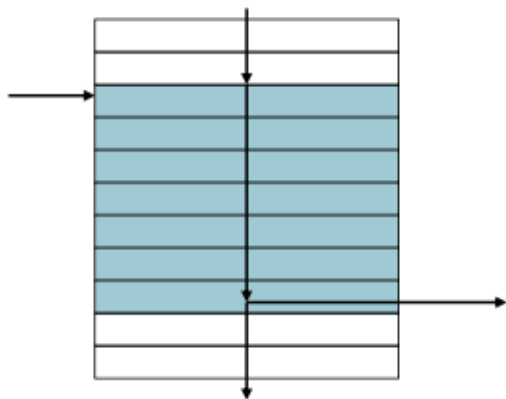
addi x22, x22, 1 #i=i+1

beq x0, x0, **Loop** # 循环

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



基本块通常有很多指令，这些指令总是按顺序执行，并被视为代码的基本块，这些基本块之间没有任何跳转语句，即当执行第一条指令时，同一基本块中所有指令按其出现顺序执行，而不会失去程序的流控制。

程序可以有各种各样的基本块，如 if then else switch-case 条件语句 和 循环如（do while for 等等）

基本块的一个典型特点是：只要基本块中第一条指令被执行了，那么基本块内所有指令都会按照顺序仅执行一次。

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** # (**jump and link**) , **RARS编程语法**
 - 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(PC+立即数偏移量offset, 有规定的转换规则)
 - 可用于函数调用
- **jalr rd, offset(rs1)** #(**jump and link register**)
 - 类似jal, 但跳转到R[rs1]+offset地址处的指令 (更远)
 - 把PC+4存到rd中, 如果rd用x0, 相当于只跳转不返回
 - 可以用于函数返回

原始语法格式: jal rd, offset

卡片2 需要记忆

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

注意：实际编程时，偏移量都用跳转标签替代，（例子中的偏移量offset是100）
RARS自动帮我们计算偏移，从而翻译成机器指令

(回顾) 用RV64I汇编实现C语言while循环结构

C代码:

```
long long int save[100];
```

.....

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

...

假设: i存储在x22中, k
存储在x24中, save数组
元素的地址保存在x25中。



RISC-V64I 代码:

```
Loop: slli x10, x22, 3    #8*i
```

```
add x10, x10, x25    #save[]+8*i
```

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

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

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

```
beq x0, x0, Loop    # 循环
```

```
Exit: ...
```

(对等表示) 用RV64I汇编实现C语言while循环

C代码:

```
long long int save[100];
```

.....

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

...

假设: i存储在x22中, k
存储在x24中, save数组
元素的地址保存在x25中。



RISC-V64I 代码:

```
Loop: slli x10, x22, 3    #8*i
```

```
add x10, x10, x25    #save[]+8*i
```

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

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

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

```
jal x0, Loop         # 或者 j Loop
```

```
Exit: ...
```

RISC-V汇编中的伪指令

- 方便程序员编程
- 通过汇编语言的变化或者组合来实现，不是直接通过硬件来实现

- 例： **mv rd, rs**

没有专门的硬件来实现数据移动指令——>

addi rd, rs1, 0 或者 add rd, rs1, x0

- li, la, j, jr, neg....

其它的伪指令

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

伪指令

PSEUDO INSTRUCTIONS

MNEMONIC	NAME	DESCRIPTION	USES
beqz	Branch = zero	$\text{if}(R[rs1]==0) \text{PC}=\text{PC}+\{\text{imm}, 1\text{b}'0\}$	beq
bnez	Branch \neq zero	$\text{if}(R[rs1]!\neq 0) \text{PC}=\text{PC}+\{\text{imm}, 1\text{b}'0\}$	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	$\text{PC} = \{\text{imm}, 1\text{b}'0\}$	jal
jr	Jump register	$\text{PC} = R[rs1]$	jalr
la	Load address	$R[rd] = \text{address}$	auipc
li	Load imm	$R[rd] = \text{imm}$	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	$\text{PC} = R[1]$	jalr
seqz	Set = zero	$R[rd] = (R[rs1]==0) ? 1 : 0$	sltiu
snez	Set \neq zero	$R[rd] = (R[rs1]!\neq 0) ? 1 : 0$	sltu

伪指令实现

常用

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>

常用

<code>jr ra</code>	<code>jalr x0, 0(ra)</code>
--------------------	-----------------------------

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

Edit Execute

Text Segment

Program Arguments:

Bkpt	Address	Code	Basic	Source
<input type="checkbox"/>	0x00400000	0x00300513	addi x10, x0, 3	1: main: addi a0, zero, 3 #想求fact(3), 把立...
<input type="checkbox"/>	0x00400004	0x00000317	auipc x6, 0	2: call fact #结果存在x10中返回
<input type="checkbox"/>	0x00400008	0x020300e7	jalr x1, x6, 0x00000020	
<input type="checkbox"/>	0x0040000c	0xfffff03b7	lui x7, 0xfffffff0	4: li t2, 0xfffff000c
<input type="checkbox"/>	0x00400010	0x00c3839b	addiw x7, x7, 12	
<input type="checkbox"/>	0x00400014	0x04050313	addi x6, x10, 0x00000040	5: addi t1, x10, 0x40
<input type="checkbox"/>	0x00400018	0x00638023	sb x6, 0(x7)	6: sb t1, 0(t2)

Labels

Label	Address ▲
fact.asm	
main	0x00400000
fact	0x00400024
L1	0x00400044

☒ Data ☒ Text

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x7ffffefe0	0x00000000	0x00000000	0x00000000	0x00000003	0x00000000	0x0040000c	0x00000000	0x00000000
0x7fffff000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7fffff020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7fffff040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7fffff060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7fffff080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7fffff0a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000



current sp ▼

☒ Hexadecimal Addresses

☒ Hexadecimal Values

☐ ASCII

条件分支转移指令示例

C语言

RISC V

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

if (i == j)

$f = g + h;$

else

$f = g - h;$

bne x13, x14, Else

add x10, x11, x12

j Exit #伪指令

Else: sub x10, x11, x12

Exit:

用RV64I汇编实现C语言for循环结构

C语言

long long int **A[20]**;

long long int **sum = 0**;

for (long long int **i = 0**; i < **20**; i++)

 sum += A[i];

//假设数组A的首地址保存在 x8,

//sum保存在x10中

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:

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

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
- 仿真软件介绍

函数调用（调用者caller与被调用者callee）

- 函数（过程）调用的6个步骤
 - (**Caller**)将参数放在过程函数可以访问的寄存器x10~x17/a0-a7
 - (**Caller**)将控制权转交给函数(**Callee**)
 - (**Callee**)获取函数所需的存储资源(包括保存现场)
 - (**Callee**)执行过程中的操作
 - (**Callee**)将结果值放在调用者可以访问到的寄存器x10~x11/a0~a1
 - (**Callee**)返回调用位置(ra中的地址)，控制权交还调用者(**Caller**)

函数调用者与被调用者

- 主调用者（caller）

准备函数参数，跳转到被调用者（子程序）——用命令*jal*

- 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

将控制权还给调用者（如跳回）——用命令*jalr*

函数调用指令（以下指令一般成对使用）

- 函数（过程）调用：

jal x1, ProcedureAddress（跳转和链接）

- 将下一条指令的地址保存在x1中
- 跳转到目标地址**ProcedureAddress**

- 函数（过程）返回：

jalr x0, 0(x1)（寄存器跳转和链接）

- 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

函数调用相关的寄存器

- 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(a0-a7)**
 - 返回值寄存器 **x10-x11(a0-a1)**
 - 保存寄存器 x8,x9,x18-x27(s0,s1,s2-s11)
 - 临时寄存器 x5-x7,x28-x31(t0-t2,t3-t6)
 - 栈指针 x2(sp)

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(Function Argument) 以及 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 (temporary register)

(回顾) 伪指令实现

常用

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>

常用

`jr ra`

`jalr x0, 0(ra)`

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

- **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

- 为什么要使用 **jr** 而不使用 j?
- sum可能会被很多地方调用
- 所以不能返回到一个固定地点
- 调用sum函数必须能够以某种方式返回

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

存在的问题:
手动计算返回地址
过于复杂

更简洁的函数调用与返回 (**jal+jalr**)

• RISC-V函数调用

1000 mv a0, s0

1004 mv a1, s1

1008 **jal** ra, **sum** # ra=1012, 此时不等价j sum

1012 ... # 下个指令

...

1996 **sum**: add a0, a0, a1

2000 **jalr** x0, 0(ra) # jump register, 等价于**jr** ra

• **jal** rd, offset # (**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)

编程要注意：函数调用**jal**和**jalr**要成对出现，前有跳转后有返

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）

◆ 主调函数和被调用函数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？

◆ 为了高效利用有限数量的寄存器，需要借助栈临时保存寄存器内容

调用函数时寄存器的使用

- 当被调用函数从执行中返回时，主调函数需要知道哪些寄存器可能已经更改，哪些寄存器保证不变。
- 在RISC-V函数调用中约定将寄存器分为两类：
 - 由主调用者保存 x10-x17(a0-a7), x1(ra), x5-x7(t0-t2), x28-x31(t3-t6)
 - 由被调用者保存 sp, gp, tp, x8-x9(s0-s1), x18-x27(s2-s11)

REGISTER NAME, USE, CALLING CONVENTION

REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5-x7	t0-t2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Function arguments/Return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller
f0-f7	ft0-ft7	FP Temporaries	Caller
f8-f9	fs0-fs1	FP Saved registers	Callee
f10-f11	fa0-fa1	FP Function arguments/Return values	Caller
f12-f17	fa2-fa7	FP Function arguments	Caller
f18-f27	fs2-fs11	FP Saved registers	Callee
f28-f31	ft8-ft11	$R[rd] = R[rs1] + R[rs2]$	Caller

函数调用嵌套问题

- 什么是函数调用嵌套？
- 函数嵌套过程

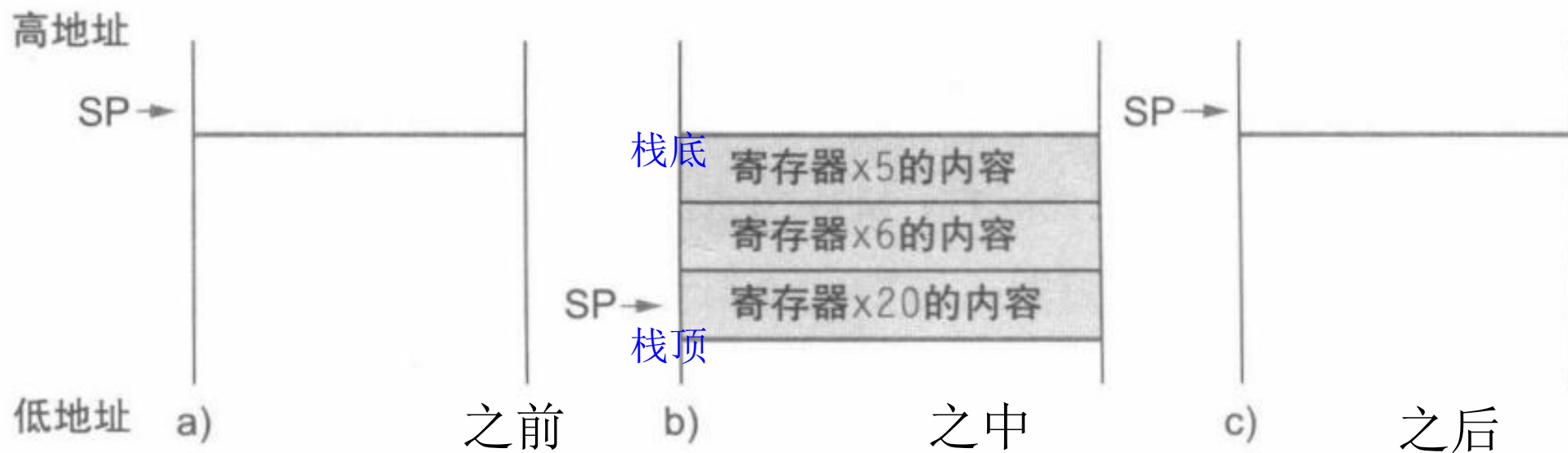
```
int sumSquare (int x, int y){  
    return mult (x, x)+y;  
}
```

- 函数sumSquare调用函数mult，寄存器ra保存的返回地址上存在冲突
- 调用mult前需要使用栈保存主调函数sumSquare的返回地址

栈的使用

- 在调用函数之前需要一个保存旧值的地方，在返回时取出并还原它们，方便寄存器内容的保护和还原。
- 堆栈：后进先出（LIFO: Last In First Out）队列
 - push: 将数据放入堆栈
 - pop: 从堆栈中取出数据

函数调用时栈的变化



- 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。
- 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**）。
- push 向下移动栈指针(减 sp)，pop 向上移动栈指针(增 sp)。

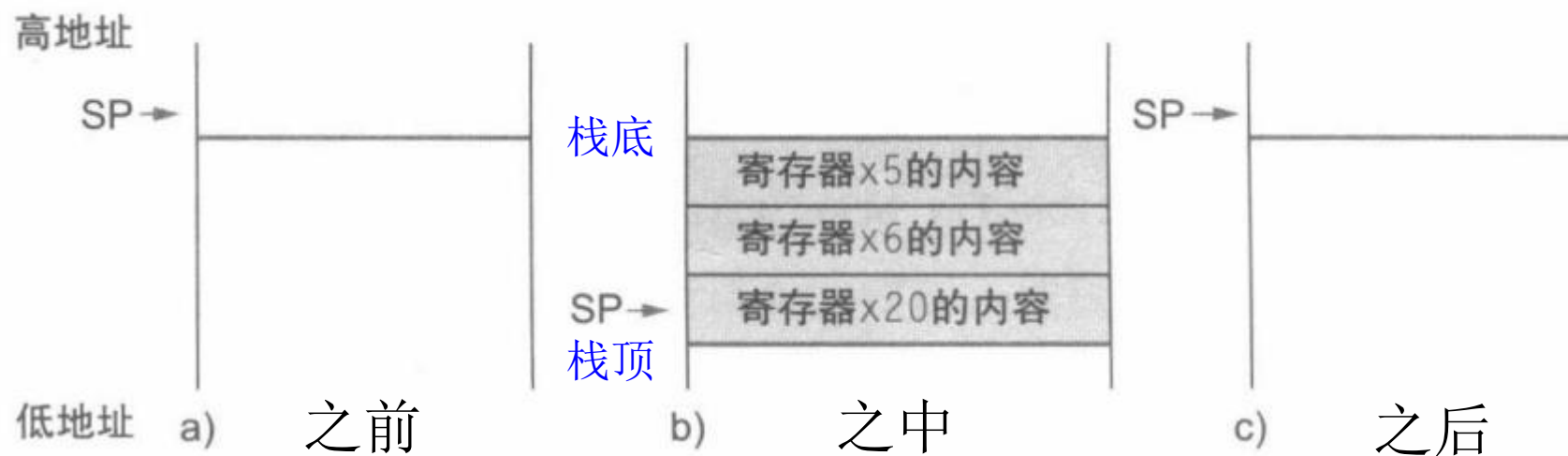
栈中保存数据的使用

- 需要保存在栈中的数据包括：
 - 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
- 栈帧（一次过程调用中产生的数据）占据连续的内存块，栈指针指示栈帧的栈顶位置。（阅读黑书73页）
- 当过程结束时，栈帧从栈中弹出，为将来的栈帧释放内存。

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，首先将这个指针地址减少(所需的空间量)，然后用需要保存信息填充它。
- 例如，将保存寄存器x9 (RV32)的内容**压栈（Push操作）**。
 - `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中的内容写入堆栈

栈的使用



RISC-V64

数据“压”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
```

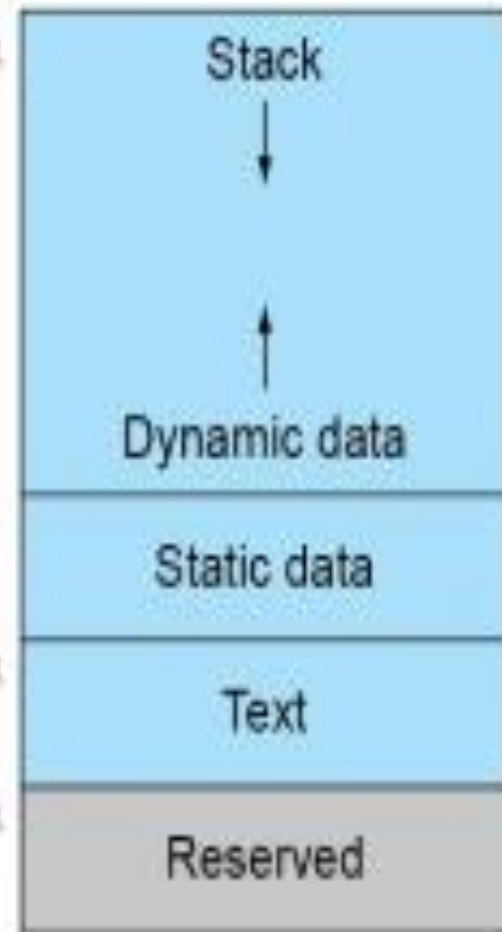
内存布局

- 代码：程序代码
- 静态数据：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，
sp加上正负偏移量可以访问这段内存空间
- 动态数据：堆
 - 例如C中的malloc、Java中的new
- 栈：自动存储

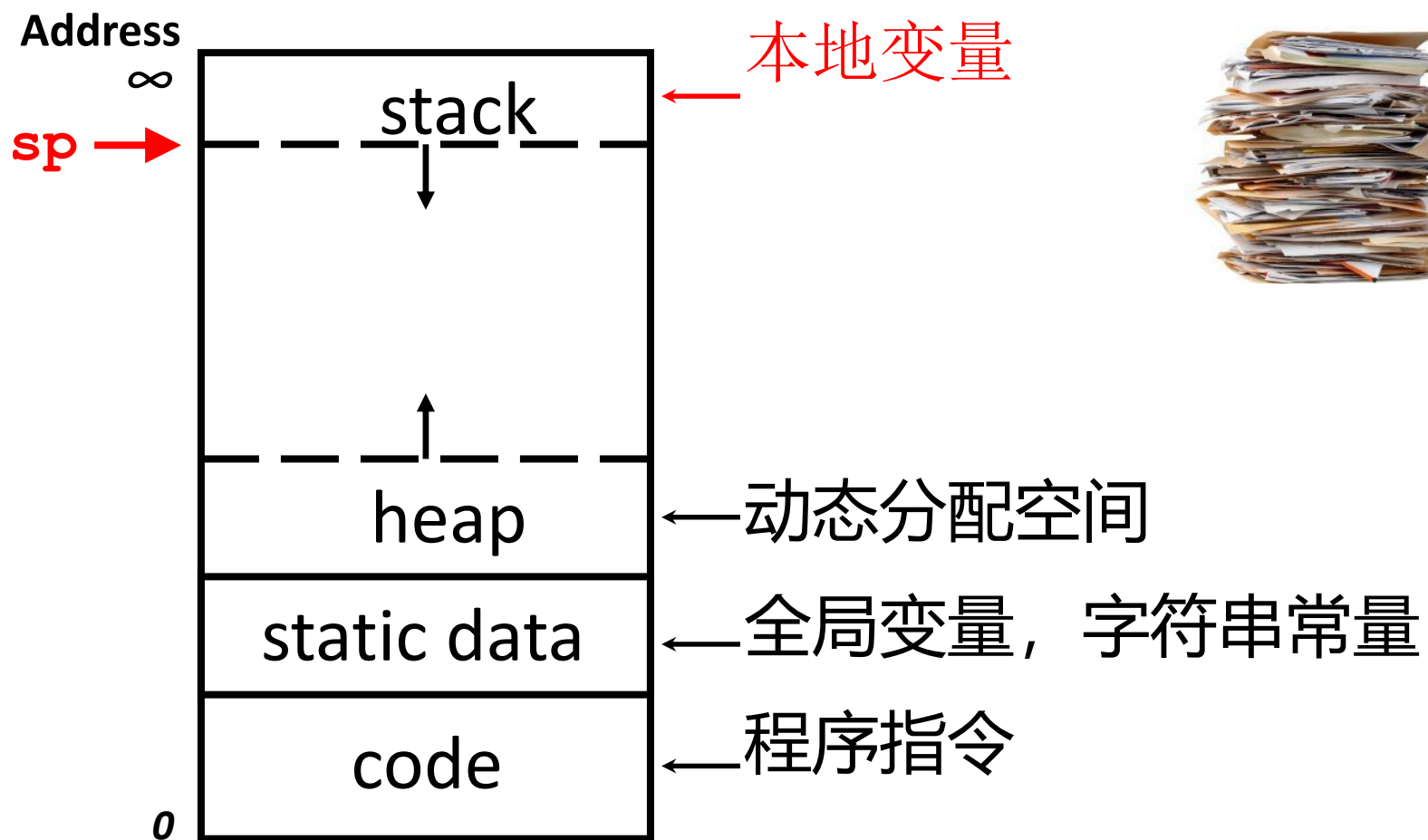
SP → 0000 003f ffff fff0_{hex}

0000 0000 1000 0000_{hex}

PC → 0000 0000 0040 0000_{hex}



数据的内存排布情况



```
int sumSquare (int x, int y){  
    return mult (x, x)+y; }  
已知: x->a0, y->a1
```

例：函数嵌套

- 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 # mult(x,x)
 jal ra, mult # call mult
pop {
 lw a1, 0(sp) # restore y
 add a0, a0, a1 # mult()+y
 lw ra, 4(sp) # get ret addr
 addi sp, sp, 8 # restore stack
 jr ra
mult: mul a0, a0, a1
 jr ra

问题：该程序在保护寄存器的入栈过程中，为什么进行了-8而不是-16操作？

答：因为本程序是32I的编程模式，通用寄存器长度都是32位，所以后面入栈时候的寄存器都是4个字节，使用的是sw和lw指令。

如果是64I下，通用寄存器长度是64位（8字节），此时入栈两个寄存器内容都是64位，源汇编程序应该修改为：addi sp, sp, -16，并且入栈和出栈要分别替换成sd和ld指令

问题：请大家分析一下ra的变化

例：字符串复制

C代码：（以null字符结束的字符串）

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

- 用x10保存数组dest地址， x11保存数组src地址， x19保存i

RV64I下，用x10保存数组dest地址，x11保存数组src地址，x19保存i

strcpy:addi sp,sp,-8

sd x19,0(sp)

add x19,x0,x0

L1: add x5,x19,x11

lbu x6,0(x5)

add x7,x19,x10

sb x6,0(x7)

beq x6,x0,L2

addi x19,x19,1

jal x0,L1

L2: ld x19,0(sp)

addi sp,sp,8

jalr x0,0(x1)

调整栈，留出1个双字的空间

x19入栈，使用前由callee保存到栈里

i=0 #开始使用x19，即初始化i

x11为数组src首地址，x5 = src[i]的地址

x6最低字节为src[i]

#x10保存dest首地址，x7 = dest[i]的地址

dest[i] = src[i]

若src[i] == 0，则退出

i = i + 1

下一次loop迭代

恢复x19原值

从栈中弹出1个双字

返回 #x1 (ra)

C代码：（以null字符结束的字符串）

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


例：字符串复制

C代码：（以null字符结束的字符串）

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

问题：该程序对于程序员来说并不严谨，为什么？

答：没有判断dest字符串数组大小能否存下src的全部内容，并且char数组需要预留一个'\0'作为字符串结束符。课后可以尝试一下。

- 用x10保存数组dest地址， x11保存数组src地址， x19保存i

例：用栈实现递归

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
- 结果也存储在x10中

见黑书72页例题

```
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
- 结果也存储在x10中

入栈

```
fact: addi sp, sp, -16  
sd x1, 8(sp)  
sd x10, 0(sp)
```

```
addi x5, x10, -1  
bge x5, x0, L1
```

```
addi x10, x0, 1  
addi sp, sp, 16  
jalr x0, 0(x1)
```

```
L1: addi x10, x10, -1  
jal x1, fact
```

```
addi t1, x10, 0  
ld x10, 0(sp)  
ld x1, 8(sp)  
addi sp, sp, 16
```

```
mul x10, x10, t1  
jalr x0, 0(x1)
```

出栈

函数
返回

| 答案 | 参数变量 n 对应参数寄存器 x10。编译后的程序从过程的标签开始，然后在栈中保存两个寄存器，返回地址和 x10:

```
fact:  
    addi sp, sp, -16 // adjust stack for 2 items  
    sd x1, 8(sp) // save the return address  
    sd x10, 0(sp) // save the argument n
```

第一次调用 fact 时，sd 保存程序中调用 fact 的地址。下面两条指令测试 n 是否小于 1，如果 $n \geq 1$ 则跳转到 L1。

```
addi x5, x10, -1 // x5 = n - 1  
bge x5, x0, L1 // if (n - 1) >= 0, go to L1
```

如果 n 小于 1，fact 将 1 放入一个值寄存器中以返回 1：它将 1 加 0 并将和存入 x10 中。然后从栈中弹出两个已保存的值并跳转到返回地址：

```
addi x10, x0, 1 // return 1  
addi sp, sp, 16 // pop 2 items off stack  
jalr x0, 0(x1) // return to caller
```

在从栈中弹出两项之前，可以加载 x1 和 x10。因为当 n 小于 1 时 x1 和 x10 不会改变，所以跳过这些指令。

如果 n 不小于 1，则参数 n 递减，然后用递减后的值再次调用 fact：

```
L1: addi x10, x10, -1 // n >= 1: argument gets (n - 1)  
jal x1, fact // call fact with (n - 1)
```

下一条指令是 fact 的返回位置，其结果在 x10 中。现在旧的返回地址和旧的参数与栈指针一起被恢复：

```
addi x6, x10, 0 // return from jal: move result of fact  
// (n - 1) to x6:  
ld x10, 0(sp) // restore argument n  
ld x1, 8(sp) // restore the return address  
addi sp, sp, 16 // adjust stack pointer to pop 2 items
```

接下来，参数寄存器 x10 得到旧参数与 fact(n-1) 结果的乘积，目前在 x6 中。假设有一个乘法指令可用，尽管在第 3 章才会涉及：

```
mul x10, x10, x6 // return n * fact (n - 1)
```

最后，fact 再次跳转到返回地址：

```
jalr x0, 0(x1) // return to the caller
```

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
- 仿真软件介绍

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。

1. `.data`

- **作用：**切换到数据段（data segment）。数据段用于存储全局变量和静态数据。
- **示例：**

```
assembly

.data
x: .word 10
y: .word 20
```

2. `.text`

- **作用：**切换到代码段（text segment）。代码段用于存储程序的指令。
- **示例：**

```
assembly

.text
.globl main
main:
    addi t0, zero, 10
```

3. `.globl`

- **作用：**声明全局符号。全局符号可以在其他文件中引用。
- **示例：**

```
assembly

.globl main
main:
    addi t0, zero, 10
```

4. `.word`

- **作用：**在数据段中分配一个或多个 32 位宽的存储单元，并初始化为指定的值。
- **示例：**

```
assembly

.data
x: .word 10
y: .word 20, 30, 40
```

5. `.byte`

- **作用：**在数据段中分配一个或多个 8 位宽的存储单元，并初始化为指定的值。
- **示例：**

```
assembly

.data
x: .byte 1, 2, 3, 4
```

6. `.half`

- **作用：**在数据段中分配一个或多个 16 位宽的存储单元，并初始化为指定的值。
- **示例：**

```
assembly

.data
x: .half 1, 2, 3, 4
```

7. `.ascii`

- **作用：**在数据段中分配一个以空字符（`\0`）结尾的字符串。
- **示例：**

```
assembly

.data
str: .ascii "Hello, World!"
```

8. `.align`

- **作用：**对齐当前地址到指定的边界。常用于确保数据或代码的对齐，以提高访问效率。
- **示例：**

```
assembly

.data
.align 2 # 对齐到 4 字节边界
x: .word 10
```

9. `.section`

- **作用：**切换到指定的段。可以用于创建自定义的段。
- **示例：**

```
assembly

.section .mysection
x: .word 10
```

示例代码

以下是一个完整的 RISC-V 汇编程序示例，展示了如何使用伪指令：

```
assembly

.data
x: .word 10
y: .word 20
str: .ascii "Hello, World!"

.text
.globl main
main:
    lw t0, x # 加载变量 x 的值到寄存器 t0
    lw t1, y # 加载变量 y 的值到寄存器 t1
    add t2, t0, t1 # 将 t0 和 t1 的值相加，结果存储在 t2
    li t3, 0 # 设置返回值为 0
    ecall # 调用系统调用退出程序
```

RISC-V仿真软件

- 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 (注意: RARS需要在安装了Java的环境下直接运行)。RARS基本界面如图1所示:

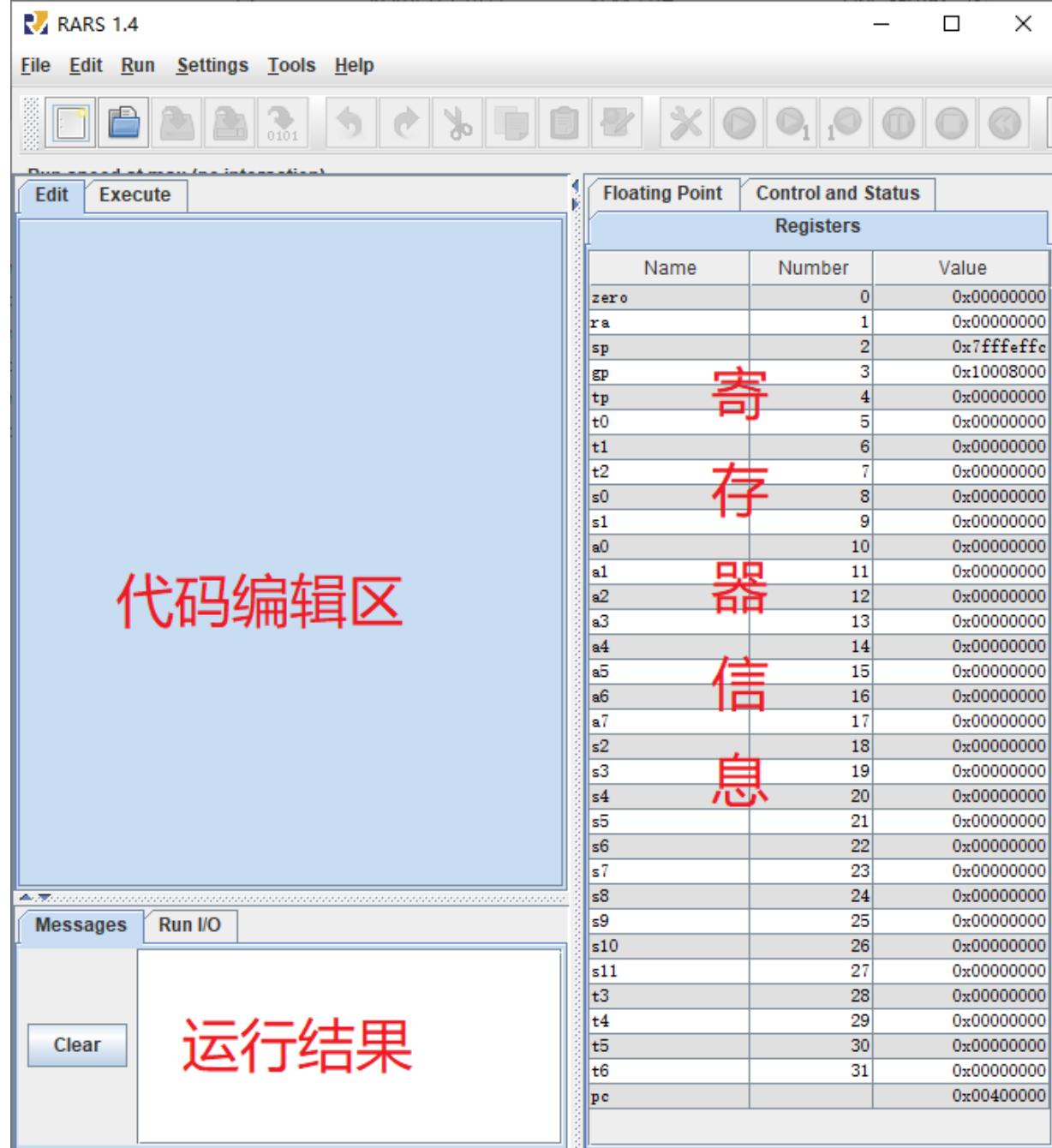


图1 RARS基本界面

RISC-V仿真软件

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：

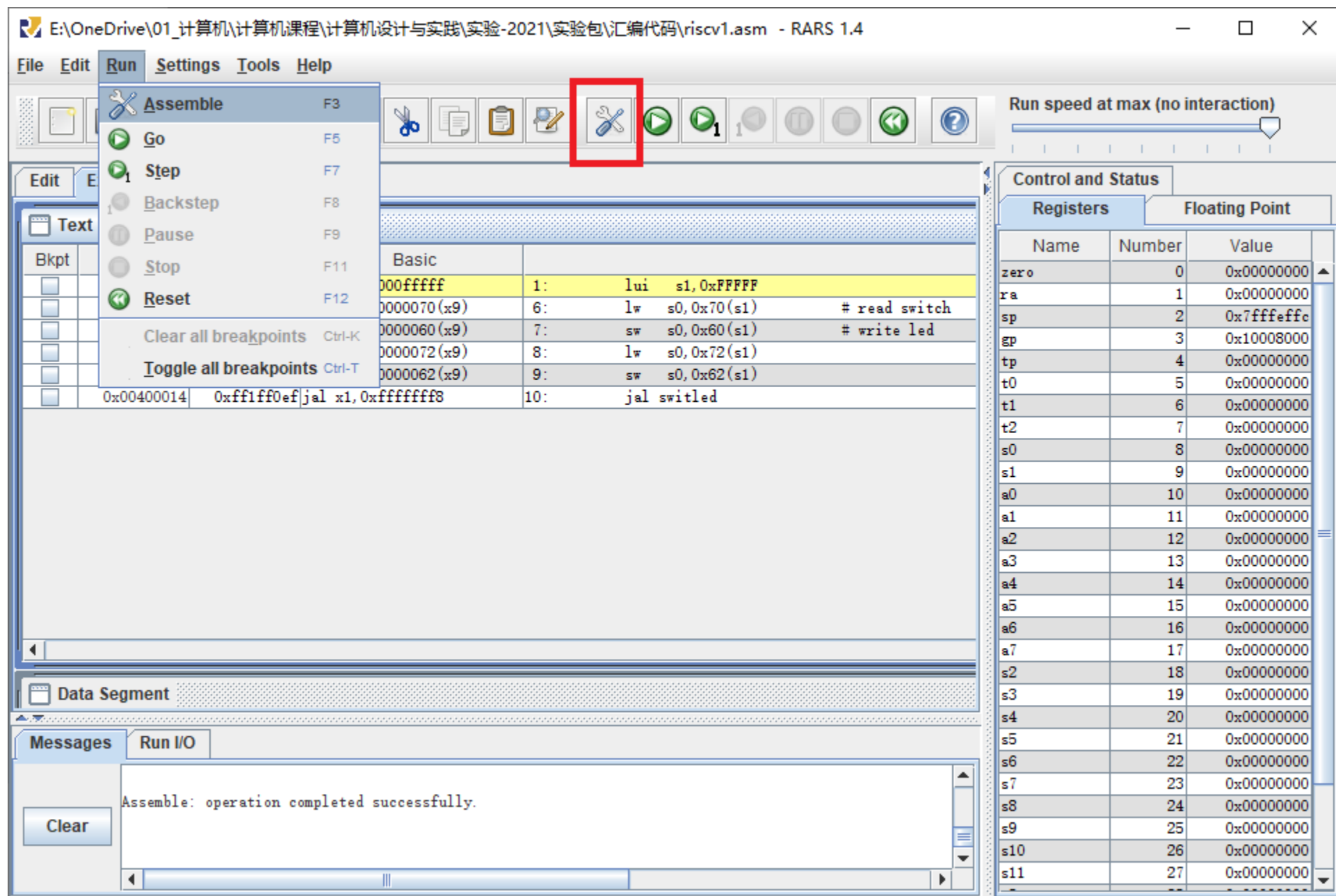


图2 RARS使用方法

RISC-V仿真软件

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。

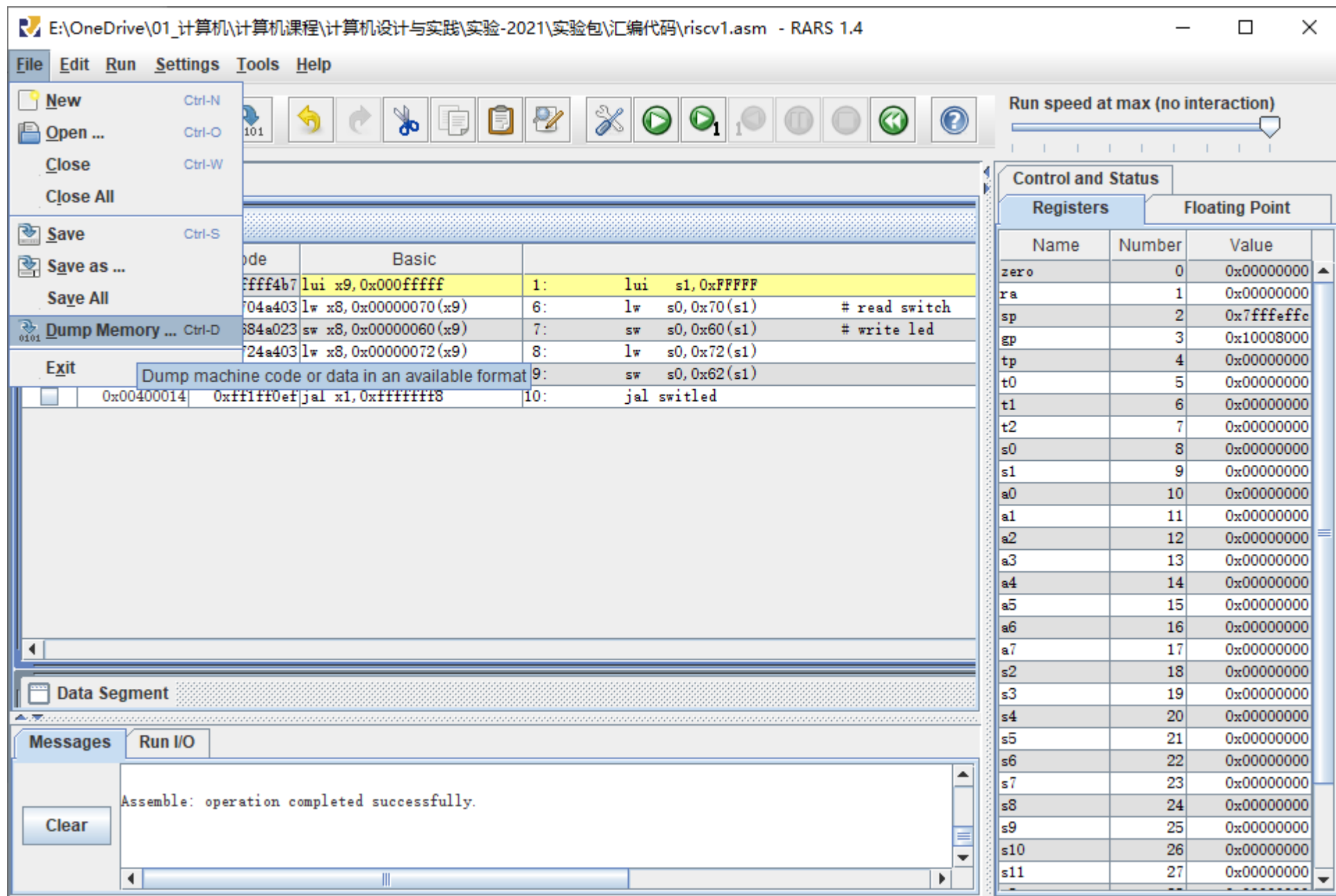


图3 从菜单导出机器码

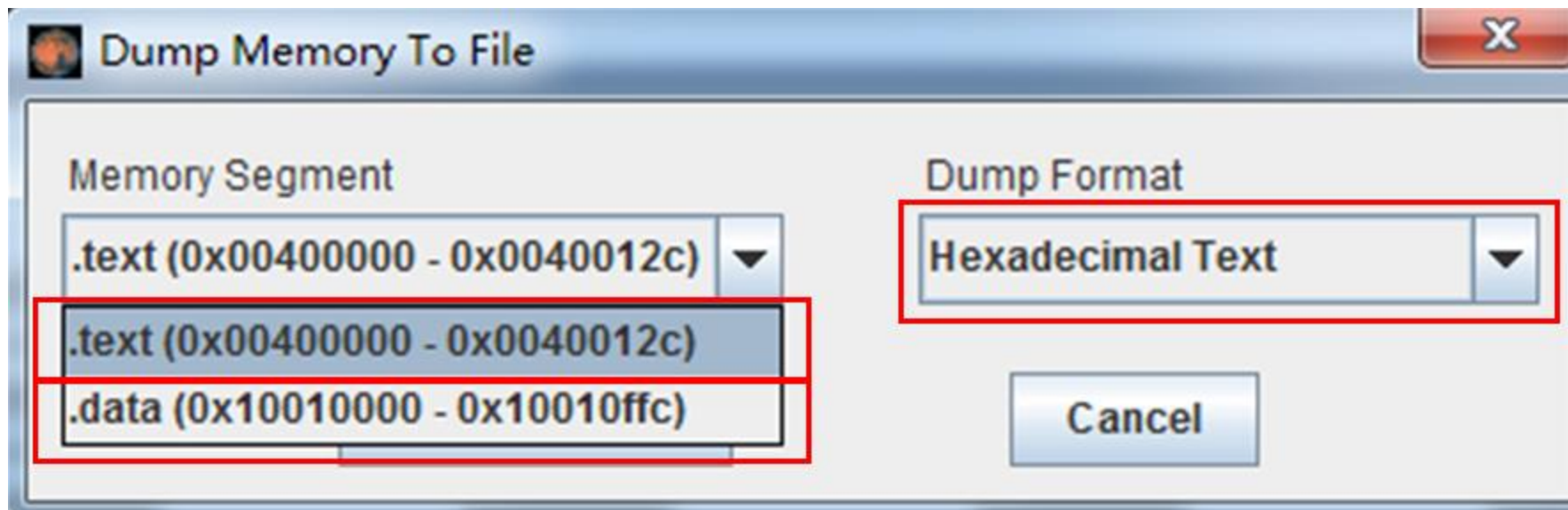


图4 从菜单导出机器码

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 image shows a RISC-V assembly simulator interface. The main window displays the assembly code for a file named `riscv1.asm`. The code defines a `swap` function that takes two arguments, `x10` and `x11`, and swaps their values using a temporary register `x6`. The function returns by jumping to the caller's return address (`jalr x0, 0(x1)`).

```
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
```

The right panel shows the register file with the following registers and values:

Name	Number	Value
zero	0	0x00000000
ra	1	0x0040000c
sp	2	0x7ffefffe
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

The bottom panel shows the Messages window with the message: "Reset: reset completed." and a "Clear" button.

运行一个简单的数据交换例子

EditExecute

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	0x004000ef	jal x1,0x00000004	3: jal x1, swap
<input type="checkbox"/>	0x0040000c	0x00a00333	add x6,x0,x10	6: add x6, x0, x10
<input type="checkbox"/>	0x00400010	0x00b00533	add x10,x0,x11	7: add x10, x0, x11
<input type="checkbox"/>	0x00400014	0x006005b3	add x11,x0,x6	8: add x11, x0, x6
<input type="checkbox"/>	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

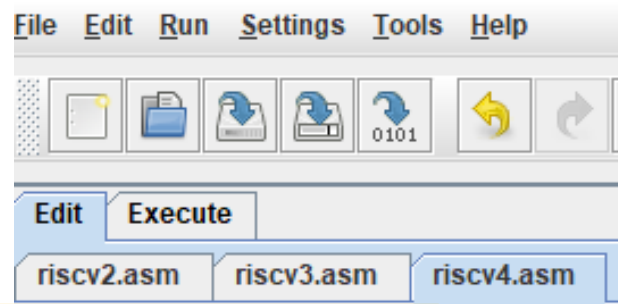
RegistersFloating PointControl and Status

Name	Number	Value
zero	0	0x00000000
ra	1	0x0040000c
sp	2	0x7ffffeffc
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

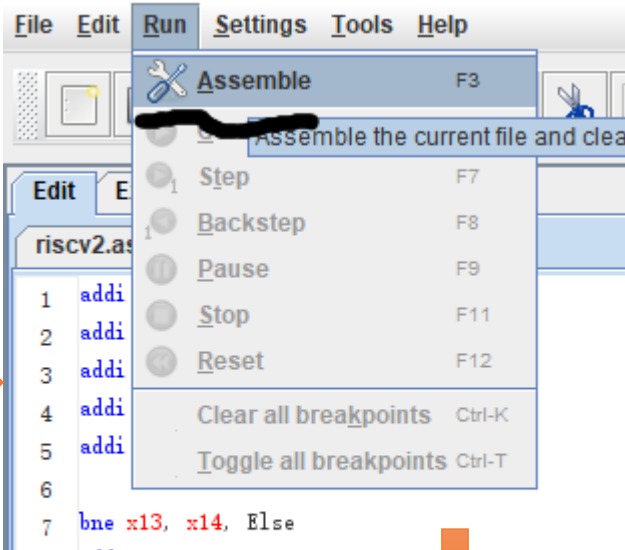
MessagesRun I/O

Reset: reset completed.

Clear

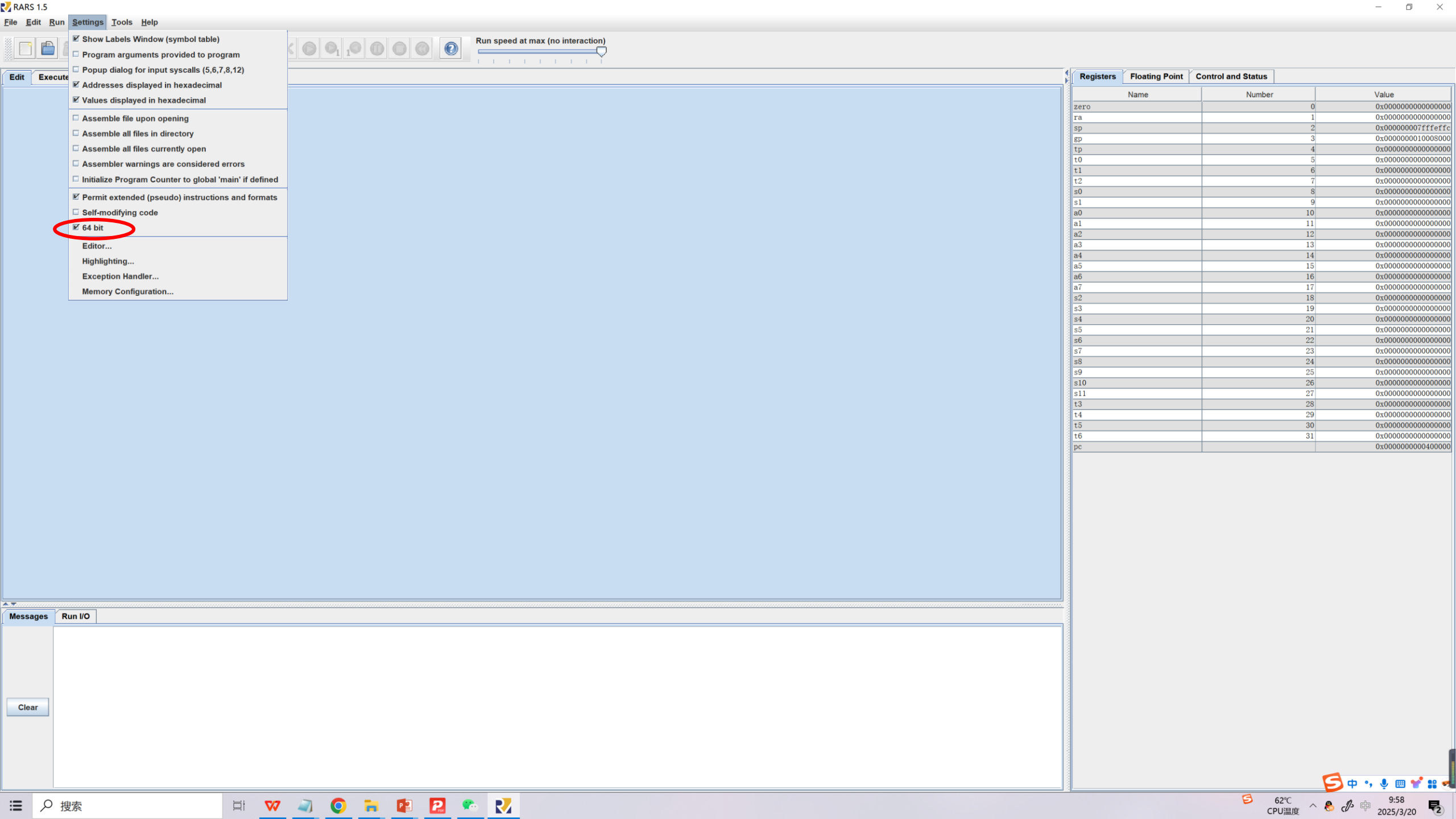


```
1 addi x10, x0, 3
2 addi x11, x0, 5
3 addi x12, x0, 7
4 addi x13, x0, 8
5 addi x14, x0, 9
6
7 bne x13, x14, Else
8 add x10, x11, x12
9 j Exit
10 Else: sub x10, x11, x12
11 Exit:
```



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	0x00000008	
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	



课后实践

任务1：用RARS完整编写并运行swap以及strcpy函数代码。

任务2：分析PPT中的swap和strcpy代码存在的问题。

任务3：用RARS完整编写fact(5)阶乘并运行出正确结果。

注意：RARS需要提前安装配置好java运行环境，见群文件

“RARS使用说明文档.pdf”。