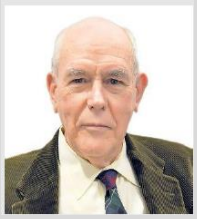
第三章 RISC-V 汇编语言

Ivan Sutherland（1938-），因为在1962年发明出Sketchpad而获得图灵奖，被誉为计算机图形学之父。Sketchpad是现代计算机的图形用户界面的先驱。



给看似困难的问题找到简单的解法往往让人心满意足，而且最好的解法常常是都简单的。

——Ivan Sutherland

3.1 导言

图 3.1[link]表明了从 C 程序翻译成为可以在计算机上执行的机器语言程序的四个经典步骤。这一章的内容包括了后面的三个步骤，不过我们要从汇编语言在 RISC-V 函数调用规范中的作用开始说起。

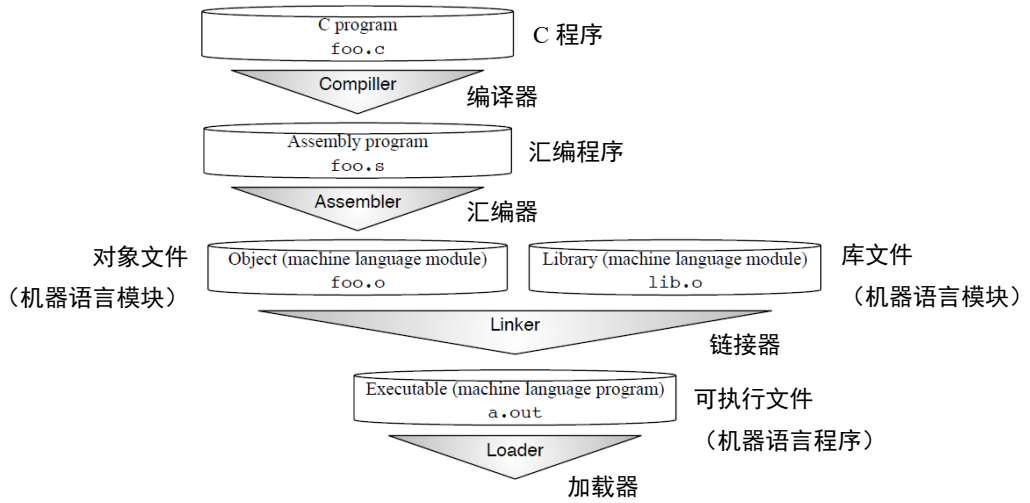


图 3.1 从 C 源代码翻译为可运行程序的步骤。这是从逻辑上进行的划分，实际中一些步骤会被结合起来，加速翻译过程。我们在这里使用了 Unix 的文件后缀命名习惯，分别对应 MS-DOS 中的.C, .ASM, .OBJ, .LIB 和.EXE。

3.2 函数调用规范（Calling convention）

函数调用过程通常分为 6 个阶段[Patterson and Hennessy 2017][link]。

1. 将参数存储到函数能够访问到的位置；
2. 跳转到函数开始位置（使用 RV32I 的 jal 指令）；
3. 获取函数需要的局部存储资源，按需保存寄存器；
4. 执行函数中的指令；
5. 将返回值存储到调用者能够访问到的位置，恢复寄存器，释放局部存储资源；
6. 返回调用函数的位置（使用 ret 指令）。

为了获得良好的性能，变量应该尽量存放在寄存器而不是内存中，但同时也要注意避免频繁地保存和恢复寄存器，因为它们同样会访问内存。

RISC-V 有足够多的寄存器来达到两全其美的结果：既能将操作数存放在寄存器中，同时也能减少保存和恢复寄存器的次数。其中的关键在于，在函数调用的过程中不保留部分寄存器存储的值，称它们为*临时寄存器*；另一些寄存器则对应地称为*保存寄存器*。不再调用其

它函数的函数称为叶函数。当一个叶函数只有少量的参数和局部变量时，它们可以都被存储在寄存器中，而不会“溢出（spilling）”到内存中。但如果函数参数和局部变量很多，程序还是需要把寄存器的值保存在内存中，不过这种情况并不多见。

函数调用中其它的寄存器，要么被当做保存寄存器来使用，在函数调用前后值不变；要么被当做临时寄存器使用，在函数调用中不保留。函数会更改用来保存返回值的寄存器，因此它们和临时寄存器类似；用来给函数传递参数的寄存器也不需要保留，因此它们也类似于临时寄存器。对于其它一些寄存器，调用者需要保证它们在函数调用前后保持不变：比如用于存储返回地址的寄存器和存储栈指针的寄存器。图 3.2[link]列出了寄存器的 RISC-V 应用程序二进制接口（ABI）名称和它们在函数调用中是否保留的规定。

寄存器 接口名称 描述 在调用中是否保留？

Register	ABI Name	Description	Preserved across call?
x0	zero	Hard-wired zero 硬编码 0	—
x1	ra	Return address 返回地址	No
x2	sp	Stack pointer 栈指针	Yes
x3	gp	Global pointer 全局指针	—
x4	tp	Thread pointer 线程指针	—
x5	t0	Temporary/alternate link register 临时寄存器	No /备用链接寄存器
x6–7	t1–2	Temporaries 临时寄存器	No
x8	s0/fp	Saved register/frame pointer 保存寄存器	Yes /帧指针
x9	s1	Saved register 保存寄存器	Yes
x10–11	a0–1	Function arguments/return values 函数参数	No /返回值
x12–17	a2–7	Function arguments 函数参数	No
x18–27	s2–11	Saved registers 保存寄存器	Yes
x28–31	t3–6	Temporaries 临时寄存器	No
f0–7	ft0–7	FP temporaries 浮点临时寄存器	No
f8–9	fs0–1	FP saved registers 浮点保存寄存器	Yes
f10–11	fa0–1	FP arguments/return values 浮点参数/返回值	No
f12–17	fa2–7	FP arguments 浮点参数	No
f18–27	fs2–11	FP saved registers 浮点保存寄存器	Yes
f28–31	ft8–11	FP temporaries 浮点临时寄存器	No

图 3.2 RISC-V 整数和浮点寄存器的汇编助记符。RISC-V 有足够的寄存器，如果过程或方法不产生其它调用，就可以自由使用由 ABI 分配的寄存器，不需要保存和恢复。调用前后不变的寄存器也称为“由调用者保存的寄存器”，反之则称为“由被调用者保存的寄存器”。浮点寄存器将第 5[link]章进行解释。（这张图源于[Waterman and Asanovi´c 2017]|[link]的表 20.1。）

根据 ABI 规范，我们来看看标准的 RV32I 函数入口和出口。下面是函数的开头：

```
entry_label:
  addi sp,sp,-framesize    # Allocate space for stack frame      调整栈指针（sp 寄存器）分配栈帧
                           # by adjusting stack pointer (sp register)
  sw   ra,framesize-4(sp)  # Save return address (ra register) 保存返回地址（ra 寄存器）
  # save other registers to stack if needed 按需保存其它寄存器
  ... # body of the function  函数体
```

如果参数和局部变量太多，在寄存器中存不下，函数的开头会在栈中为函数帧分配空间，来存放。当一个函数的功能完成后，它的结尾部分释放栈帧并返回调用点：

```
# restore registers from stack if needed 按需恢复其它寄存器
lw   ra,framesize-4(sp)  # Restore return address register 恢复返回地址
addi sp,sp, framesize    # De-allocate space for stack frame 释放栈帧空间
ret                      # Return to calling point   返回调用点
```

我们很快将会看到使用这套 ABI 的一个例子，但首先我们需要对汇编的其它部分进行

一些解释。

补充说明：保存寄存器和临时寄存器为什么不是连续编号的？

为了支持 RV32E——一个只有 16 个寄存器的嵌入式版本的 RISC-V（参见第 11[link]章），只使用寄存器 x0 到 x15——一部分保存寄存器和一部分临时寄存器都在这个范围内。其它的保存寄存器和临时寄存器在剩余 16 个寄存器内。RV32E 较小，但由于和 RV32I 不匹配，目前还没有编译器支持。

3.3 汇编器

在 Unix 系统中，这一步的输入是以.s 为后缀的文件，比如 foo.s；在 MS-DOS 中则是.ASM。

图 3.1[link]中的汇编器的作用不仅仅是从处理器能够理解的指令产生目标代码，还能翻译一些扩展指令，这些指令对汇编程序员或者编译器的编写者来说通常很有用。这类指令在巧妙配置常规指令的基础上实现，称为伪指令。图 3.3[link]和 3.4[link]列出了 RISC-V 伪指令，前者中要求 x0 寄存器始终为 0，后者中则没有这种要求。例如，之前提到的 ret 实际上是一个伪指令，汇编器会用 jalr x0, x1, 0 来替换它（见图 3.3[link]）。大多数的 RISC-V 伪指令依赖于 x0。因此，把一个寄存器硬编码为 0 便于将许多常用指令——如跳转（jump）、返回（return）、等于 0 时转移（branch on equal to zero）——作为伪指令，进而简化 RISC-V 指令集。

图 3.5[link]为经典的 C 程序 Hello World，编译器产生的汇编指令如图 3.6[link]，其中使用了图 3.2[link]的调用规范和图 3.3[link]、3.4[link]的伪指令。

Hello World 程序通常是一个新设计处理器上运行的第一个程序。设计者通常把能运行操作系统并成功打印出“Hello World”作为新的芯片能工作的标志。他们会马上发邮件给领导和同事，告诉他们这个结果，然后出去搓一顿。

汇编程序的开头是一些汇编指示符（assemble directives）。它们是汇编器的命令，具有告诉汇编器代码和数据的位置、指定程序中使用的特定代码和数据常量等作用。图 3.9[link]是 RISC-V 的汇编指示符。其中图 3.6[link]中用到的指示符有：

- .text：进入代码段。
- .align 2：后续代码按 22 字节对齐。
- .globl main：声明全局符号“main”。
- .section .rodata：进入只读数据段
- .balign 4：数据段按 4 字节对齐。
- .string “Hello, %s!\n”：创建空字符结尾的字符串。
- .string “world”：创建空字符结尾的字符串。

汇编器产生如图 3.7[link]的目标文件，格式为标准的可执行可链接文件（ELF）格式[TIS Committee 1995][link]。

伪指令	基础指令	含义	
Pseudoinstruction	Base Instruction(s)	Meaning	
`nop`	`addi x0, x0, 0`	No operation	无操作
`neg rd, rs`	`sub rd, x0, rs`	Two's complement	补码
`negw rd, rs`	`subw rd, x0, rs`	Two's complement word	字的补码
`snez rd, rs`	`sltu rd, x0, rs`	Set if $\neq$ zero	非 0 则置位
`sltz rd, rs`	`slt rd, rs, x0`	Set if $<$ zero	小于 0 则置位
`sgtz rd, rs`	`slt rd, x0, rs`	Set if $>$ zero	大于 0 则置位
`beqz rs, offset`	`beq rs, x0, offset`	Branch if $=$ zero	为 0 则转移
`bnez rs, offset`	`bne rs, x0, offset`	Branch if $\neq$ zero	非 0 则转移
`blez rs, offset`	`bge x0, rs, offset`	Branch if $\leq$ zero	小于等于 0 则转移
`bgez rs, offset`	`bge rs, x0, offset`	Branch if $\geq$ zero	大于等于 0 则转移
`bltz rs, offset`	`blt rs, x0, offset`	Branch if $<$ zero	小于 0 则转移
`bgtz rs, offset`	`blt x0, rs, offset`	Branch if $>$ zero	大于 0 则转移
`j offset`	`jal x0, offset`	Jump	跳转
`jr rs`	`jalr x0, rs, 0`	Jump register	寄存器跳转
`ret`	`jalr x0, x1, 0`	Return from subroutine	从子过程返回
`tail offset`	`auipc x6, offset[31:12]` `jalr x0, x6, offset[11:0]`	Tail call far-away subroutine	尾调用远程子过程
`rdinstret[h] rd`	`csrrs rd, instret[h], x0`	Read instructions-retired counter	读取过时指令计数器
`rdcycle[h] rd`	`csrrs rd, cycle[h], x0`	Read cycle counter	读取周期计数器
`rdtime[h] rd`	`csrrs rd, time[h], x0`	Read real-time clock	读取实时时钟
`csrr rd, csr`	`csrrs rd, csr, x0`	Read CSR	读 CSR 寄存器
`csrw csr, rs`	`csrrw x0, csr, rs`	Write CSR	写 CSR 寄存器
`csrs csr, rs`	`csrrs x0, csr, rs`	Set bits in CSR	CSR 寄存器置零位
`csrc csr, rs`	`csrrc x0, csr, rs`	Clear bits in CSR	CSR 寄存器清
`csrwi csr, imm`	`csrrwi x0, csr, imm`	Write CSR, immediate	立即数写入 CSR
`csrsi csr, imm`	`csrrsi x0, csr, imm`	Set bits in CSR, immediate	立即数置位 CSR
`csrci csr, imm`	`csrrci x0, csr, imm`	Clear bits in CSR, immediate	立即数清除 CSR
`frcsr rd`	`csrrs rd, fcsr, x0`	Read FP control/status register	读取 FP 控制/状态寄存器
`fscsr rs`	`csrrw x0, fcsr, rs`	Write FP control/status register	写入 FP 控制/状态寄存器
`frrm rd`	`csrrs rd, frm, x0`	Read FP rounding mode	读取 FP 舍入模式
`fsrm rs`	`csrrw x0, frm, rs`	Write FP rounding mode	写入 FP 舍入模式
`frflags rd`	`csrrs rd, fflags, x0`	Read FP exception flags	读取 FP 例外标志
`fsflags rs`	`csrrw x0, fflags, rs`	Write FP exception flags	写入 FP 例外标志

图 3.3 依赖于 x0 的 RISC-V 伪指令。附录 A[link]包含了这些 RISC-V 的伪指令和真实指令。在 RV32I 中，那些读取 64 位计数器的指令默认读取低 32 位，增加“h”时读取高 32 位。（这张图源于[Waterman and Asanovi´c 2017][link]的表 20.2 和表 20.3。）

伪指令	基础指令	含义	
Pseudoinstruction	Base Instruction(s)	Meaning	
`lla rd, symbol`	`auipc rd, symbol[31:12]` `addi rd, rd, symbol[11:0]`	Load local address	取局部地址
`la rd, symbol`	*PIC*: `auipc rd, GOT[symbol][31:12]` `l{w|d} rd, rd, GOT[symbol][11:0]` *Non-PIC*: Same as `lla rd, symbol`	Load address	取地址
`l{b|h|w|d} rd, symbol`	`auipc rd, symbol[31:12]` `l{b|h|w|d} rd, symbol[11:0](rd)`	Load global	读取全局量
`s{b|h|w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `s{b|h|w|d} rd, symbol[11:0](rt)`	Store global	存储全局量
`fl{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fl{w|d} rd, symbol[11:0](rt)`	Floating-point load global	读取浮点全局量
`fs{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fs{w|d} rd, symbol[11:0](rt)`	Floating-point store global	存储浮点全局量
`li rd, immediate`	*Myriad sequences*	Load immediate	读取立即数
`mv rd, rs`	`addi rd, rs, 0`	Copy register	复制寄存器
`not rd, rs`	`xori rd, rs, -1`	One's complement	反码
`sext.w rd, rs`	`addiw rd, rs, 0`	Sign extend word	有符号扩展字
`seqz rd, rs`	`sltiu rd, rs, 1`	Set if = zero	为 0 时置位
`fmv.s rd, rs`	`fsgnj.s rd, rs, rs`	Copy single-precision register	复制单精度寄存器
`fabs.s rd, rs`	`fsgnjx.s rd, rs, rs`	Single-precision absolute value	单精度绝对值
`fneg.s rd, rs`	`fsgnjn.s rd, rs, rs`	Single-precision negate	单精度相反数
`fmv.d rd, rs`	`fsgnj.d rd, rs, rs`	Copy double-precision register	复制双精度寄存器
`fabs.d rd, rs`	`fsgnjx.d rd, rs, rs`	Double-precision absolute value	双精度绝对值
`fneg.d rd, rs`	`fsgnjn.d rd, rs, rs`	Double-precision negate	双精度相反数
`bgt rs, rt, offset`	`blt rt, rs, offset`	Branch if >	大于时转移
`ble rs, rt, offset`	`bge rt, rs, offset`	Branch if ≤	小于等于时转移
`bgtu rs, rt, offset`	`bltu rt, rs, offset`	Branch if >, unsigned	无符号大于时转移
`bleu rs, rt, offset`	`bgeu rt, rs, offset`	Branch if ≤, unsigned	无符号小于等于时转移
`jal offset`	`jal x1, offset`	Jump and link	跳转并链接
`jalr rs`	`jalr x1, rs, 0`	Jump and link register	跳转并链接寄存器
`call offset`	`auipc x1, offset[31:12]` `jalr x1, x1, offset[11:0]`	Call far-away subroutine	远程调用子过程
`fence`	`fence iorw, iorw`	Fence on all memory and I/O	内存和 I/O 屏障
`fscsr rd, rs`	`csrrw rd, fcsr, rs`	Swap FP control/status register	交换 FP 控制/状态寄
`fsrm rd, rs`	`csrrw rd, frm, rs`	Swap FP rounding mode	交换 FP 舍入模式
`fsflags rd, rs`	`csrrw rd, fflags, rs`	Swap FP exception flags	交换 FP 例外标志

图 3.4 不依赖于 x0 寄存器的 RISC-V 伪指令。在 la 指令一栏，GOT 代表全局偏移表（Global Offset Table），记录动态链接库中的符号的运行时地址。附录 A[link]包含了这些 RISC-V 的伪指令和真实指令。（这张图源于[Waterman and Asanovi´c 2017][link]的表 20.2 和表 20.3。）

```
#include <stdio.h>
int main()
{
   printf("Hello, %s\n", "world");
   return 0;
}
```

图 3.5 C 语言的 Hello World 程序（hello.c）。

```
  .text                   # Directive: enter text section          指示符：进入代码段
  .align 2                # Directive: align code to 2^2 bytes     指示符：按 2^2 字节对齐代码
  .globl main             # Directive: declare global symbol main  指示符：声明全局符号 main
main:                     # label for start of main                main 开始标记
  addi sp,sp,-16          # allocate stack frame                   分配栈帧
  sw   ra,12(sp)          # save return address                    存储返回地址
  lui  a0,%hi(string1)    # compute address of                     计算 string1 的地址
  addi a0,a0,%lo(string1) #   string1
  lui  a1,%hi(string2)    # compute address of                     计算 string2 的地址
  addi a1,a1,%lo(string2) #   string2
  call printf             # call function printf                   调用 printf 函数
  lw   ra,12(sp)          # restore return address                 恢复返回地址
  addi sp,sp,16           # deallocate stack frame                 释放栈帧
  li   a0,0               # load return value 0                    读取返回值
  ret                     # return                                 返回
  .section .rodata        # Directive: enter read-only data section  指示符：进入只读数据段
  .balign 4               # Directive: align data section to 4 bytes 指示符：按 4 字节对齐数据
string1:                  # label for first string                   第一个字符串标记
  .string "Hello, %s!\n"  # Directive: null-terminated string        指示符：空字符结尾的字符串
string2:                  # label for second string                  第二个字符串标记
  .string "world"         # Directive: null-terminated string        指示符：空字符结尾的字符串
```

图 3.6 RISC-V 汇编语言的 Hello World 程序（hello.s）。

```
00000000 <main>:
 0: ff010113  addi  sp,sp,-16
 4: 00112623  sw    ra,12(sp)
 8: 00000537  lui   a0,0x0
 c: 00050513  mv    a0,a0
10: 000005b7  lui   a1,0x0
14: 00058593  mv    a1,a1
18: 00000097  auipc ra,0x0
1c: 000080e7  jalr  ra
20: 00c12083  lw    ra,12(sp)
24: 01010113  addi  sp,sp,16
28: 00000513  li    a0,0
2c: 00008067  ret
```

图 3.7 RISC-V 机器语言的 Hello World 程序（hello.o）。位置 8 到 1c 这六条指令的地址字段为 0，将在后面由链接器填充。目标文件的符号表记录了链接器所需的标签和地址。

3.4 链接器

链接器允许各个文件独立地进行编译和汇编，这样在改动部分文件时，不需要重新编译全部源代码。链接器把新的目标代码和已经存在的机器语言模块（如函数库）等“拼接”起来。链接器这个名字源于它的功能之一，即编辑所有对象文件的跳转并链接指令（jump and link）中的链接部分。它其实是链接编辑器（link editor）的简称，图 3.1[link]中的这一步骤过去就被称为链接编辑。在 Unix 系统中，链接器的输入文件有.o 后缀，输出 a.out 文件；在 MS-DOS 中输入文件后缀为.OBJ 或.LIB，输出.EXE 文件。

图 3.10[link]展示了一个典型的 RISC-V 程序分配给代码和数据的内存区域，链接器需要调整对象文件的指令中程序和数据的地址，使之与图中地址相符。如果输入文件中的是与位置无关的代码（PIC），链接器的工作量会有所降低。PIC 中所有的指令转移和文件内的数据访问都不受代码位置的影响。如第 2[link]章所言，RV32I 的相对转移（PC-relative branch）

特性使得程序更易于实现 PIC。

除了指令，每个目标文件还包含一个符号表，存储了程序中标签，由链接过程确定地址。其中包括了数据标签和代码标签。图 3.6[link]中有两个数据标签（string1 和 string2）和两个代码标签（main 和 printf）需要确定。由于在单个 32 位指令中很难指定一个 32 位的地址，RV32I 的链接器通常需要为每个标签调整两条指令。如图 3.6[link]所示：数据标签需要调整 lui 和 addi，代码标签需要调整 quipc 和 jalr。图 3.8[link]显示了图 3.7[link]中的目标文件链接后产生的 a.out 文件。

```
000101b0 <main>:
   101b0: ff010113 addi sp,sp,-16
   101b4: 00112623 sw   ra,12(sp)
   101b8: 00021537 lui  a0,0x21
   101bc: a1050513 addi a0,a0,-1520 # 20a10 <string1>
   101c0: 000215b7 lui  a1,0x21
   101c4: a1c58593 addi a1,a1,-1508 # 20a1c <string2>
   101c8: 288000ef jal  ra,10450 <printf>
   101cc: 00c12083 lw   ra,12(sp)
   101d0: 01010113 addi sp,sp,16
   101d4: 00000513 li   a0,0
   101d8: 00008067 ret
```

图 3.8 链接后的 RISC-V 机器语言 Hello World 程序。在 Unix 系统中，它的文件名是 a.out。

RISC-V 编译器支持多个 ABI，具体取决于 F 和 D 扩展是否存在。RV32 的 ABI 分别名为 ilp32，ilp32f 和 ilp32d。ilp32 表示 C 语言的整型（int），长整型（long）和指针（pointer）都是 32 位，可选后缀表示如何传递浮点参数。在 lip32 中，浮点参数在整数寄存器中传递；在 ilp32f 中，单精度浮点参数在浮点寄存器中传递；在 ilp32d 中，双精度浮点参数也在浮点寄存器中传递。

自然，如果想在浮点寄存中传递浮点参数，需要相应的浮点 ISA 添加 F 或 D 扩展（见第 5[link]章）。因此要编译 RV32I 的代码（GCC 选项-march=rv32i），必须使用 ilp32 ABI（GCC 选项-mabi=lib32）。反过来，调用约定并不要求浮点指令一定要使用浮点寄存器，因此 RV32IFD 与 ilp32，ilp32f 和 ilp32d 都兼容。

链接器检查程序的 ABI 是否和库匹配。尽管编译器本身可能支持多种 ABI 和 ISA 扩展的组合，但机器上可能只安装了特定的几种库。因此，一种常见的错误是在缺少合适的库的情况下链接程序。在这种情况下，链接器不会直接产生有用的诊断信息，它会尝试进行链接，然后提示不兼容。这种错误常常在从一台计算机上编译另一台计算机上运行的程序（交叉编译）时发生。

补充说明：链接器松弛（linker relaxation）

跳转并链接指令（jump and link）中有 20 位的相对地址域，因此一条指令就足够跳到很远的位置。尽管编译器为每个外部函数的跳转都生成了两条指令，很多时候其实一条就已经足够了。从两条指令到一条的优化同时节省了时间和空间开销，因此链接器会扫描几遍代码，尽可能地把两条指令替换为一条。每次替换会导致函数和调用它的位置之间的距离缩短，所以链接器会多次扫描替换，直到代码不再改变。这个过程称为链接器松弛，名字来源于求解方程组的松弛技术。除了过程调用之外，对于 gp 指针±2KiB 范围内的数据访问，RISC-V 链接器也会使用一个全局指针替换掉 lui 和 auipc 两条指令。对 tp 指针±2KiB 范围内的线程局部变量访问也有类似的处理。

指示符	描述	
Directive	Description	
.text	Subsequent items are stored in the text section (machine code).	代码段（机器语言代码）。
.data	Subsequent items are stored in the data section (global variables).	数据段（全局变量）。
.bss	Subsequent items are stored in the bss section (global variables initialized to 0).	bss 段（初始化为 0 的全局变量）。
.section .foo	Subsequent items are stored in the section named .foo.	命名为.foo 的段。
.align n	Align the next datum on a 2^n-byte boundary. For example, .align 2 aligns the next value on a word boundary.	按 2n 字节对齐。如.align 2 是按字对齐。
.balign n	Align the next datum on a n-byte boundary. For example, .balign 4 aligns the next value on a word boundary.	按 n 字节对齐。如.balign 4 是按字对齐。
.globl sym	Declare that label sym is global and may be referenced from other files.	声明 sym 标签为全局的，可从其它文件访问。
.string "str"	Store the string str in memory and null-terminate it.	将字符串 str 存入内存，空字符结尾。
.byte b1,..., bn	Store the n 8-bit quantities in successive bytes of memory.	在内存中连续存储 n 个 8 位的量。
.half w1,...,wn	Store the n 16-bit quantities in successive memory halfwords.	在内存中连续存储 n 个 16 位的量。
.word w1,...,wn	Store the n 32-bit quantities in successive memory words.	在内存中连续存储 n 个 32 位的量。
.dword w1,...,wn	Store the n 64-bit quantities in successive memory doublewords.	在内存中连续存储 n 个 64 位的量。
.float f1,..., fn	Store the n single-precision floating-point numbers in successive memory words.	在内存中连续存储 n 个单精度浮点数。
.double d1,..., dn	Store the n double-precision floating-point numbers in successive memory doublewords.	在内存中连续存储 n 个双精度浮点数。
.option rvc	Compress subsequent instructions (see Chapter 7).	压缩指令（见第 7[link]章）。
.option norvc	Don't compress subsequent instructions.	不压缩指令。
.option relax	Allow linker relaxations for subsequent instructions.	允许链接器松弛（linker relaxation）。
.option norelax	Don't allow linker relaxations for subsequent instructions.	不允许链接器松弛。
.option pic	Subsequent instructions are position-independent code.	与位置无关代码段。
.option nopic	Subsequent instructions are position-dependent code.	与位置有关代码段。
.option push	Push the current setting of all .options to a stack, so that a subsequent .option pop will restore their value.	将所有的.option 设置存入栈。
.option pop	Pop the option stack, restoring all .options to their setting at the time of the last .option push.	从栈中弹出上次存入的.option 设置。

图 3.9 常见 RISC-V 汇编指示符。

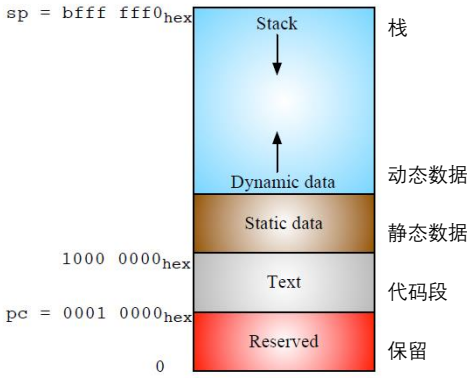


图 3.10 RV32I 为程序和数据分配内存。图中的顶部是高地址，底部是低地址。在 RISC-V 软件规范中，栈指针（sp）从 0xbfffff0 开始向下增长；程序代码段从 0x00010000 开始，包括静态链接库；程序代码段结束后是静态数据区，在这个例子中假设从 0x10000000 开始；然后是动态数据区，由 C 语言中的 malloc()函数分配，向上增长，其中包含动态链接库。

3.5 静态链接和动态链接

体系结构研究者常用静态链接的基准程序来测试处理器，尽管大多数实际的程序都有动态链接。他们说，关心性能的用户应该只使用静态链接，但其实这并不合理，因为加速实际的程序显然比加速基准程序更有意义。

上一节对静态链接（static linking）进行了说明，在程序运行前所有的库都进行了链接和加载。如果这样的库很大，链接一个库到多个程序中会十分占用内存。另外，链接时库是绑定的，即使它们后来的更新修复了 bug，强制的静态链接的代码仍然会使用旧的、有 bug 的版本。

为了解决这两个问题，现在的许多系统使用动态链接（dynamic linking），外部的函数在第一次被调用时才会加载和链接。后续所有调用都使用快速链接（fast linking），因此只会产生一次动态开销。每次程序开始运行，它都会按照需要链接最新版本的库函数。另外，如果多个程序使用了同一个动态链接库，库代码在内存中只会加载一次。

编译器产生的代码和静态链接的代码很相似。其不同之处在于，跳转的目标不是实际的函数，而是一个只有三条指令的存根函数（stub function）。存根函数会从内存中的一个表中加载实际的函数的地址并跳转。不过，在第一次调用时，表中还没有实际的函数的地址，只有一个动态链接的过程的地址。当这个动态链接过程被调用时，动态链接器通过符号表找到实际要调用的函数，复制到内存中，更新记录实际的函数地址的表。后续的每次调用的开销就是存根函数的三条指令的开销。

3.6 加载器

类似图 3.8[link]的程序以一个可执行文件的形式存储在计算机的存储设备上。运行时，加载器的作用是把这个程序加载到内存中，并跳转到它开始的地址。如今的“加载器”就是操作系统。换句话说，加载 a.out 是操作系统众多的任务之一。

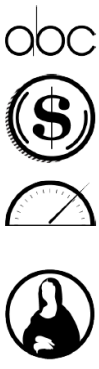
动态链接程序的加载稍微有些复杂。操作系统不直接运行程序，而是运行一个动态链接器，再由动态链接器开始运行程序，并负责处理所有外部函数的第一次调用，把它们加载到内存中，并且修改程序，填入正确的调用地址。

3.7 结语

保持简洁，保持功能单一。

——Kelly Johnson，提出“KISS 原则”的航空工程师，1960

汇编器向 RISC-V ISA 中增加了 60 条伪指令，使得 RISC-V 代码更易于读写，并且不增加硬件开销。将一个寄存器硬编码为 0 使得其中许多伪指令更容易实现。使用加载高位立即数（lui）和程序计数器与高位立即数相加（auipc）两条指令，简化了编译器和链接器寻找外部数据/函数的地址的过程。使用相对地址转移的代码与位置无关，减少了链接器的工作。大量的寄存器减少了寄存器保存和恢复的次数，加速函数调用和返回。



RISC-V 提供了一系列简单又有影响力的机制，降低成本，提高性能，并且使得编写程序更加容易。

3.8 扩展阅读

D. A. Patterson and J. L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann, 2017.

TIS Committee. Tool interface standard (TIS) executable and linking format (ELF) specification version 1.2. *TIS Committee*, 1995.

A. Waterman and K. Asanovi´c, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. May 2017. URL https://riscv.org/specifications/.

Notes: 1. http://parlab.eecs.berkeley.edu (I have no idea about what this is. It's in p43.)