

# lab 5-1: RISC-V 汇编程序设计

## 1. 实验环境

1. 操作系统: Ubuntu 24.04

## 2. 实验目的

1. 理解 RISC-V 汇编程序
2. 能够用 RISC-V 汇编指令编写简单的汇编程序

## 3. 实验报告 70%

### 3.1 理解简单 RISC-V 程序 10%

请阅读 `src/lab5-1/acc_plain.s` 回答以下问题：

1. `acc` 是如何获得函数参数的，又是如何返回函数返回值的？ 2%

通过 `a0` 和 `a1` 寄存器传参，返回值通过 `a0` 传回。

2. `acc` 函数中 `s0` 寄存器的作用是什么，为什么在函数入口处需要执行 `sd s0, 40(sp)` 这条指令，而在这条指令之后的 `addi s0, sp, 48` 这条指令的目的是什么？ 2%

`s0`：帧指针 (frame pointer)。`sd s0, 40(sp)` 是先保存调用者的 `s0` (因为 `s0` 是被调用者保存寄存器)；`addi s0, sp, 48` 是把 `s0` 设为当前栈帧基准地址，后续用 `-24(s0)`、`-32(s0)` 固定偏移访问局部变量。作用与 8086 中 `bp` 类似。

3. `acc` 函数的栈帧 (stack frame) 的大小是多少？ 1%

栈帧大小是 48 字节 (`addi sp, sp, -48`)。

4. `acc` 函数栈帧中存储的值有哪些，它们分别存储在哪 (相对于 `sp` 或 `s0` 来说)？ 2%

主要有：旧 `s0` (`40(sp)`，即 `-8(s0)`)、参数 `a0` (`8(sp)`，即 `-40(s0)`)、参数 `a1` (`0(sp)`，即 `-48(s0)`)、累加和 `sum` (`16(sp)`，即 `-32(s0)`)、循环变量 `i` (`24(sp)`，即 `-24(s0)`)。

| 内容                   | 相对于 <code>sp</code> 的偏移 | 相对于 <code>s0</code> 的偏移 |
|----------------------|-------------------------|-------------------------|
| 参数 <code>a1</code>   | 0 (sp)                  | -48 (s0)                |
| 参数 <code>a0</code>   | 8 (sp)                  | -40 (s0)                |
| 累加和 <code>sum</code> | 16 (sp)                 | -32 (s0)                |
| 循环变量 <code>i</code>  | 24 (sp)                 | -24 (s0)                |
| 旧 <code>s0</code>    | 40 (sp)                 | -8 (s0)                 |

Table 1

5. 请简要解释 `acc` 函数中的 for 循环是如何在汇编代码中实现的。1%

先初始化 `sum=0`、`i=a0`，再跳到条件检查标签 `.L2`；若 `i<=a1` 就到 `.L3` 执行循环体（`sum+=i; i++`），之后回到 `.L2` 继续判断；条件不满足时跳出并返回 `sum`。

请阅读 `src/lab5-1/acc_opt.s` 回答以下问题：

1. 请查阅资料简要描述编译选项 `-O0` 和 `-O2` 的区别。1%

`-O0` 基本不做优化，代码更接近源码、便于调试但运行慢；`-O2` 会做较充分优化（如循环优化、寄存器分配改进等），通常更快更小，但调试可读性会下降。

2. 请简要讨论 `src/lab5-1/acc_opt.s` 与 `src/lab5-1/acc_plain.s` 的优劣。1%

`acc_opt.s` 基本用寄存器完成计算、无显式栈帧，指令更少、性能更好；`acc_plain.s` 保留了完整栈帧和中间变量落栈，结构更直观、便于教学和单步调试。

### 3.2 理解递归汇编程序 15%

请阅读 `src/lab5-1/factor_plain.s` 回答以下问题：

1. 为什么 `src/lab5-1/factor_plain.s` 中 `factor` 函数的入口处需要执行 `sd ra, 24(sp)` 指令，而 `src/lab5-1/acc_plain.s` 中的 `acc` 函数并没有执行该指令？2%

`factor` 是递归函数，会执行 `call factor`，`ra` 会被新的调用覆盖，所以必须先保存；`acc` 是叶子函数（不再调用其他函数），不需要保存 `ra`。

2. 请解释在 `call factor` 前的 `mv a0, a5` 这条汇编指令的目的。2%

RISC-V 调用约定要求第一个参数放在 `a0`，而前一步算出的 `n-1` 在 `a5`，所以要先 `mv a0, a5` 再调用，实现 `factor(n-1)` 传参。

3. 请简要描述调用 `factor(10)` 时栈的变化情况；并回答栈最大内存占用是多少，发生在什么时候。3%

每次进入 `factor` 都 `sp-=32` 建栈帧，递归从 `factor(10)` 一直压到 `factor(0)`；随后逐层返回、每层 `sp+=32` 回收。最大占用发生在刚进入最深层 `factor(0)` 时，共 11 层栈帧，即  $11 \times 32 = 352$  字节。

4. 假设栈的大小为 4KB，请问 `factor(n)` 的参数 `n` 最大是多少？2%

每层 32 字节，4KB=4096 字节，最多  $4096/32=128$  层；递归层数是 `n+1`（含 `factor(0)`），所以 `n` 最大为 127。

请阅读 `src/lab5-1/factor_opt.s` 回答以下问题：

1. 请简要描述 `src/lab5-1/factor_opt.s` 和 `src/lab5-1/factor_plain.s` 的区别。2%

`factor_plain.s` 用递归实现，含函数调用与栈帧保存/恢复；`factor_opt.s` 被优化为循环，直接在寄存器里累计结果，不再递归调用。

2. 请从栈内存占用的角度比较 `src/lab5-1/factor_opt.s` 和 `src/lab5-1/factor_plain.s` 的优劣。2%

`factor_plain.s` 的栈占用随 `n` 线性增长（约  $32 \times (n+1)$  字节）；  
`factor_opt.s` 栈占用为常数，不会像前者一样溢出。

3. 请查阅尾递归优化的相关资料，解释编译器在生成 `src/lab5-1/factor_opt.s` 时做了什么优化，该优化的原理，以及什么时候能进行该优化。2%

编译器把“可等价转成尾部跳转/循环”的递归形式改写成循环（在 `factor_opt.s` 中体现为 `.L2` 循环与寄存器累乘），避免重复 `call/ret` 和新栈帧。原理是复用当前栈帧，把下一次调用参数更新后直接跳回循环入口。一般要求递归调用位于函数尾位置（调用后不再有依赖其返回值的额外计算），且语义可等价变换。

### 3.3 理解 switch 语句产生的跳转表 5%

请阅读这两个文件并回答以下问题：

1. 请简述在 `src/lab5-1/switch.s` 中是如何实现 switch 语句的。2%

先把 `case` 值变为索引（`a5 = a0 - 20`），再做无符号越界判断（`bgtu`）；若在范围内，就用索引在 `.rodata` 的跳转表 `.L4` 取偏移地址并 `jr` 间接跳转到对应分支，否则跳默认分支 `.L8`。

2. 请简述用跳转表实现 switch 和用 if-else 实现 switch 的优劣，在什么时候应该采用跳转表，在什么时候应该采用 if-else。3%

跳转表分支时间近似常数、case 多且连续时效率高，但会占用额外表空间，且 case 稀疏时浪费；

if-else 链实现简单、适合 case 少或离散分布，但分支多时比较开销增大。一般“case 值范围紧凑且数量较多”选跳转表，“case 少/稀疏/条件复杂”选 if-else。

### 3.4 设计冒泡排序的汇编代码 20%

```
1  .text
2  .globl bubble_sort
3
4  bubble_sort:
5      # void bubble_sort(long long *arr, long long len);
6      # a0: address of the array
7      # a1: length of the array
8      li a2, 1
9      ble a1, a2, .exit      # 如果数组长度 <= 1, 退出
10     addi t4, a1, -1
11     li a3, 0               # a3: 外层循环
12 .outer_loop:
13     li a4, 0               # a4: 内层循环
14 .inner_loop:
15     # 比较 arr[a4] 和 arr[a4 + 1], 从小到大排序
16     slli t0, a4, 3         # t0 = a4 * 8 (每个long long元素占8字节)
17     add t1, a0, t0         # t1 = &arr[a4], a0基地址加上t0偏移
18     ld t2, 0(t1)          # t2 = arr[a4]
19     ld t3, 8(t1)          # t3 = arr[a4 + 1]
20     blt t2, t3, .no_swap
21     sd t3, 0(t1)
22     sd t2, 8(t1)
23 .no_swap:
24     addi a4, a4, 1
25     blt a4, t4, .inner_loop # 没有优化
26     addi a3, a3, 1
27     blt a3, t4, .outer_loop
28 .exit:
29     ret
30
```

代码解释：a0 是数组首地址，a1 是长度；a3 / a4 分别做外层与内层循环计数。内层通过 slli 把下标乘 8 定位 long long 元素地址，读取相邻两个数比较，必要时交换。当前分支条件 blt t2, t3, .no\_swap 的实际效果是“前小后大时不交换”，因此整体会把较大值往前推（即降序）。

```
1  root@zju-os:/zju-os/code/src/lab5-1# riscv64-linux-gnu-gcc
   bubble_sort_test.c bubble_sort.s -o bubble_sort_test -static
2  ./simulate.sh bubble_sort_test
3  Sort Successfully
```

```

● root@zju-os:/zju-os/code/src/lab5-1# riscv64-linux-gnu-gcc bubble_
sort_test.c bubble_sort.s -o bubble_sort_test -static
./simulate.sh bubble_sort_test
Sort Successfully
● root@zju-os:/zju-os/code/src/lab5-1# riscv64-linux-gnu-gcc fibonac
ci_test.c fibonacci.s -o fibonacci_test -static
./simulate.sh fibonacci_test
Pass

```

Image 1

### 3.5 设计斐波那契数列的汇编代码 20%

```

1  .text
2  .globl fibonacci
3
4  fibonacci:
5      # long long fibonacci(long long n);
6      # a0: n
7      li a1, 2
8      blt a0, a1, .case_0_1    # 如果 n < 2, 返回 1(Fibonacci以1,1开头)
9      addi sp, sp, -24
10     sd ra, 8(sp)              # 返回地址
11     sd a0, 0(sp)              # 参数
12                                # sp+16: fib(n-1) 的结果(中间值)
13                                # sp+8: 返回地址 ra
14                                # sp+0: 参数 n
15                                # sp: 栈顶(低地址)
16     # fibonacci(n - 1)
17     addi a0, a0, -1
18     call fibonacci
19     sd a0, 16(sp)              # !!!保存 fibonacci(n - 1)
20     ld a0, 0(sp)              # 恢复参数 n
21     # fibonacci(n - 2)
22     addi a0, a0, -2
23     call fibonacci
24     mv t2, a0                  # t2 = fibonacci(n - 2)
25     ld t1, 16(sp)              # !!!恢复 fibonacci(n - 1)
26     add a0, t1, t2              # t3 = fibonacci(n - 1) + fibonacci(n - 2)
27     ld ra, 8(sp)              # 恢复返回地址
28     addi sp, sp, 24            # 恢复栈指针
29     ret
30 .case_0_1:
31     li a0, 1
32     ret
33

```

代码解释：当  $n < 2$  直接返回 1。否则先开 24 字节栈帧，保存 ra 和原始参数 n；先递归算  $\text{fibonacci}(n-1)$  并把结果暂存到  $16(sp)$ ，再恢复 n 去算  $\text{fibonacci}(n-2)$ ，最后把两次结果相加作为返回值。

```
1 root@zju-os:/zju-os/code/src/lab5-1# riscv64-linux-gnu-gcc
  fibonacci_test.c fibonacci.s -o fibonacci_test -static
2 ./simulate.sh fibonacci_test
3 Pass
```

```
● root@zju-os:/zju-os/code/src/lab5-1# riscv64-linux-gnu-gcc bubble_
  sort_test.c bubble_sort.s -o bubble_sort_test -static
  ./simulate.sh bubble_sort_test
  Sort Successfully
● root@zju-os:/zju-os/code/src/lab5-1# riscv64-linux-gnu-gcc fibonac
  ci_test.c fibonacci.s -o fibonacci_test -static
  ./simulate.sh fibonacci_test
  Pass
```

Image 2

## lab 5-2: RISC-V 汇编程序调试

### 1. 实验环境

1. 操作系统: Ubuntu 24.04

### 2. 实验目的

1. 掌握 gdb 调试工具的基本命令和使用方法
2. 掌握 qemu、spike 模拟器的调试方法
3. 学习 openocd 等其他软件

### 3. 实验报告 30%

qemu 调试部分给出调试过程的关键截图，并且给出自己推断得到字符串的调试过程和推断依据。

phase\_1:

根据gdb调试，`data = 5`，即输入字符串含有5且满足输入要求即可。

```
1 while(*str){
2     if(char2num(*str) == data){
3         ret = 0;
4         break;
5     }
6     str++;                //逐位检查
7 }
```

```

root@zju-os:/zju-os# gdb-multiarch challenge
(gdb) b main
Breakpoint 1 at 0x55555556f6e: file challenge.c, line 182.
(gdb) c
Continuing.
Reading /lib/libc.so.6 from remote target...

Breakpoint 1, main () at challenge.c:182
182     printf("please input your student ID:");
(gdb) b phase_1
Breakpoint 2 at 0x555555556ab2: file challenge.c, line 59.
(gdb) c
Continuing.

Breakpoint 2, phase_1 (str=0x5555555590a0 <input_buffer> "123") at challenge.c:59
59     int data = char2num(name_buffer[6]);
(gdb) b 64
Breakpoint 3 at 0x555555556b3e: file challenge.c, line 64.
(gdb) c
Continuing.

Breakpoint 3, phase_1 (str=0x5555555590a0 <input_buffer> "123") at challenge.c:64
64     int ret = 1;
(gdb) p data
$1 = 5
(gdb) c
Continuing.
[Inferior 1 (process 286460) exited with code 0377]
(gdb)

```

Image 3

phase\_2:

根据gdb调试, `sum = 9`, 即输入字符串的每位数字异或后等于9且满足输入要求即可。

如: "09", 即  $48 \oplus 57 \rightarrow 00110000(48) \oplus 00111001(57) = 00001001(9)$  满足要求。

```

1      "j phase_2_L1\n"
2      "phase_2_L2:\n"
3      "xor t0, t0, t2\n" //输入字符串的每位数字异或后等于9, 再和sum(t0) 异或
4      "addi t1, t1, 1\n" //向后移动位指针
5      "phase_2_L1:\n"
6      "lbu t2, 0(t1)\n"
7      "bne t2, zero, phase_2_L2\n"
8      "bne t0, zero, phase_2_L3\n" //结果为0, 成功
9      "mv %[ret], zero\n"
10     "phase_2_L3:\n"

```

```

root@zju-os:/zju-os/code/src/lab5-2#
qemu-riscv64 -g 1234 challenge
please input your student ID:3250101
697
Welcome to my fiendish little bomb.
You have 3 phases with
5
Phase 1 defused. How about the next
one?
1
Bomb! You fail to defuse the second
bomb!
root@zju-os:/zju-os/code/src/lab5-2#

root@zju-os:/zju-os# gdb-multiarch challenge

Breakpoint 2, phase_1 (str=0x5555555590a0 <input_buffer> "5") at challenge.c:59
59     int data = char2num(name_buffer[6]);
(gdb) delete
Delete all breakpoints, watchpoints, tracepoints, and catchpoints? (y or n) y
(gdb) b phase_2
Breakpoint 8 at 0x555555556b9a: file challenge.c, line 76.
(gdb) c
Continuing.

Breakpoint 8, phase_2 (str=0x5555555590a0 <input_buffer> "1") at challenge.c:76
76     int sum = char2num(name_buffer[6])^char2num(name_buffer[7])^char2num(n
ame_buffer[8])^char2num(name_buffer[9]);
(gdb) b 77
Breakpoint 9 at 0x555555556c06: file challenge.c, line 77.
(gdb) c
Continuing.

Breakpoint 9, phase_2 (str=0x5555555590a0 <input_buffer> "1") at challenge.c:77
77     int ret = 1;
(gdb) p sum
$2 = 9
(gdb) b phase_3
Breakpoint 10 at 0x555555556c54: file challenge.c, line 100.
(gdb) c
Continuing.
[Inferior 1 (process 292060) exited with code 0377]
(gdb)

```

Image 4

phase\_3:

根据调试, `char expect[9] = "99245cb8"`, 即输入等于 `expect` 即可。

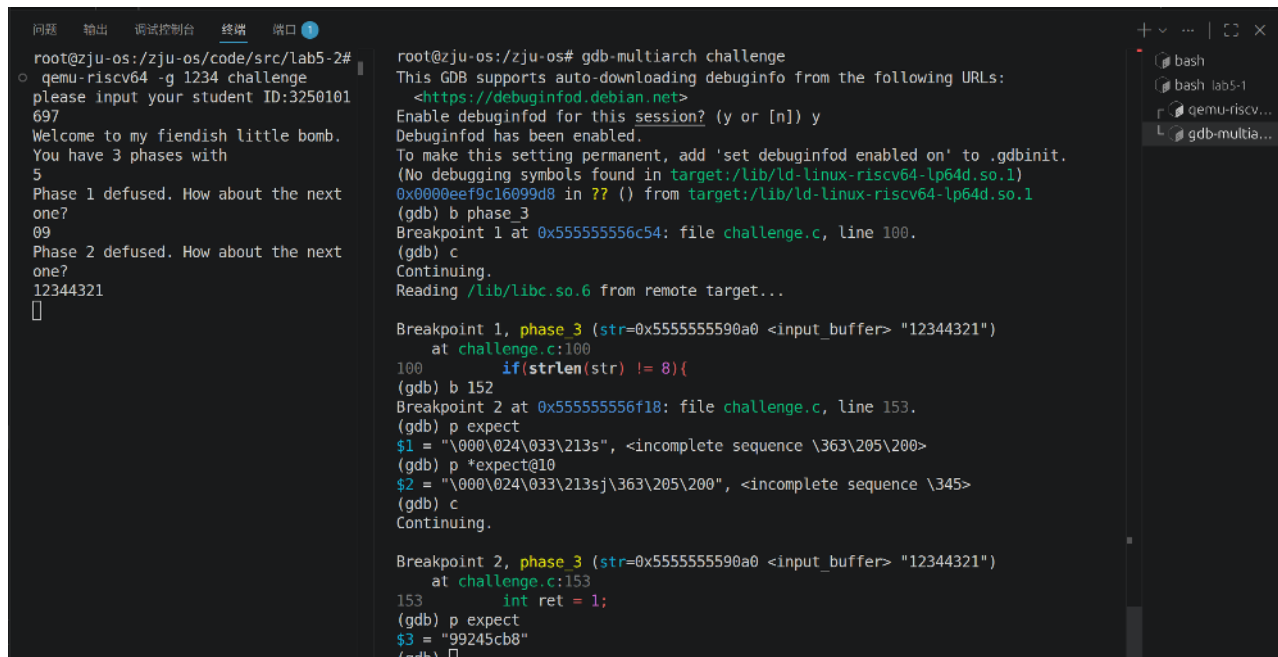


Image 5

final:

见下图左侧的终端。分别输入 `5`, `09`, `99245cb8` 即可通过测试。(phase1, 2 的答案不唯一)

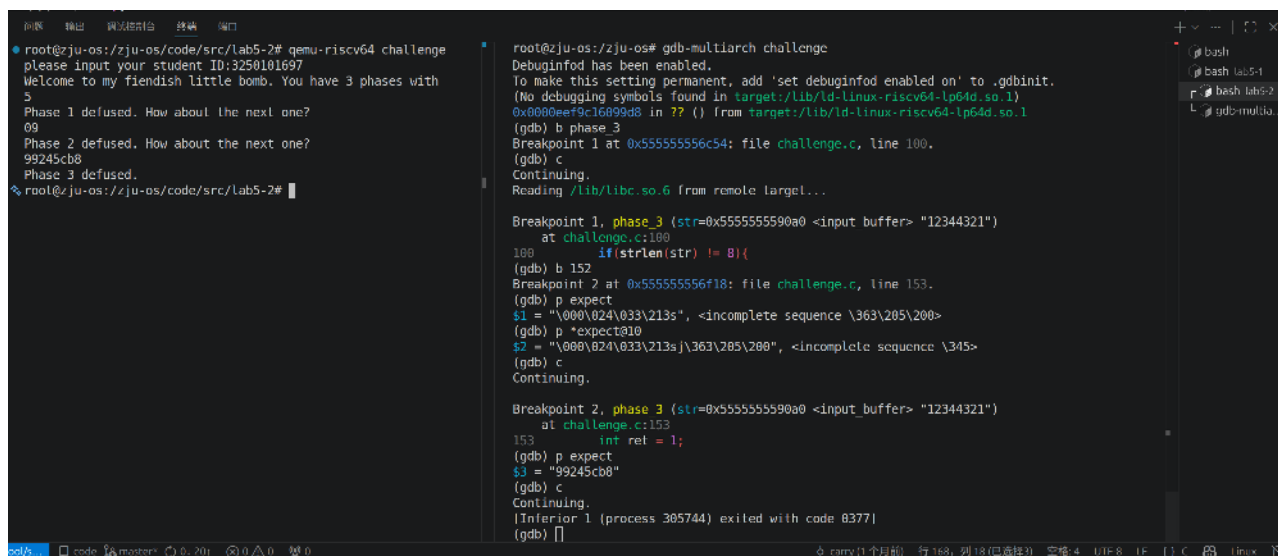


Image 6