Lab5-1 Report

4.2 理解简单 RISC-V 程序

1. 在 acc 函数中，它从 a0 寄存器获得其参数（即累加的上限），并在函数结束时，将计算结果（累加的结果）存放回 a0 寄存器以返回。

2. s0 寄存器在 RISC-V 中通常用作帧指针，它用于标记函数栈帧的基址，使得即使栈指针 sp 在函数执行中发生变化，仍然可以稳定地访问局部变量和保存的寄存器值。

3.栈的大小为 48 字节。由 addi sp, sp, -48 可知。

4. s0 值：存储在 40(sp) 的位置

参数 a0：存储在 -40(s0) 的位置，即函数的输入参数（累加上限）。

局部变量（例如，循环的当前值和累加值）：存储在 -32(s0) 和 -24(s0)，分别用于累加器和循环计数器。

5. 初始化：在循环开始前，累加器初始化为 0，循环索引初始化为 0。

循环条件：使用 ble 指令比较当前索引与上限，如果索引小于等于上限，则继续循环。

循环体：

累加操作：将索引值加到累加器上。

索引递增：将循环索引加 1。

循环继续：如果条件满足，使用 j 指令跳转回循环开始的位置。

结束循环：当循环索引超过上限时，循环退出，函数准备返回值并退出。

在使用 GCC 或其他 C/C++ 编译器时，-O 选项用于控制编译器的优化级别。不同的优化级别影响编译速度、执行速度、以及生成代码的大小。

-O0：这是默认的优化级别，不启用优化。编译器的目标是减少编译时间并生成调试友好的代码。这使得生成的代码容易调试，但执行效率可能较低，因为它没有进行任何优化。

-O2：这个优化级别启用了一组旨在不牺牲编译时间的情况下提高程序性能的优化。它会尝试提高代码的执行速度和/或减少代码的大小。-O2 包括几乎所有不涉及空间-时间权衡的优化（即不会显著增加代码大小的优化），如死码删除、循环展开、内联函数等。

两个程序的比较：acc_opt.s 结构较简单（直接使用寄存器，减少了栈操作），更易于理解，同时进行了优化，使得执行效率也较高。acc_plain.s 中更多的局部变量和栈操作使代码更难直接阅读和维护，虽然使得生成的代码容易调试，但执行效率可能较低，因为它没有进行任何优化。

4.3 理解递归汇编程序

1. 这一指令的执行是因为在递归调用中需要保存返回地址 ra。在 factor 函数中，每次调用自身之前，它将返回地址 ra 保存到栈上，以确保在递归调用返回后，能正确回到上一个递归调用的点。而 acc 函数不涉及递归调用，无需执行此类指令。

2. 此指令是为了把当前的 n-1 传递给 factor 函数的下一次递归调用。在递减 a5 之后，通过这条指令，递减后的值（即 n-1）被传入 a0，因为 a0 是调用函数时用于传递第一个参数的标准寄存器。

3. 调用 factor(10)时，每次递归调用都会将当前的栈指针 sp 向下移动 32 字节，以分配局部变量和保存寄存器的空间。

因此，栈的最大内存占用将发生在最深的递归调用层次，即 factor(0)调用时。此时，栈的总占用为 32×11=352 字节（从 factor(10)递归到 factor(0)，共 11 层）。

4. 栈大小为 4096 字节（4KB）。每次递归调用大约占用 32 字节的栈空间，因此最大递归深度大约为 409632=128324096=128 层。这意味着最大安全递归深度为 128 层，即参数 n 最大为 127（因为从 factor(127)调用到 factor(0)为 128 次递归）。

Factor_opt.s

1. factor_plain.s 使用了标准的递归调用，每次调用自身前保存调用栈状态，并在返回后恢复。而 factor_opt.s 则使用循环代替了递归，通过在寄存器中累乘结果来计算阶乘，从而避免了调用栈的频繁使用。
2. factor_plain.s 对于每个递归调用，都会进行栈的分配和寄存器的保存，导致较高的栈内存消耗。而 factor_opt.s 几乎不使用额外的栈空间。它通过循环而非递归来实现计算，因此避免了为每次递归调用分配和释放栈空间。
3. 尾递归优化是一种特别的编译器优化技术，用于优化尾递归函数。尾递归函数是指一个函数的最后一个动作是调用另一个函数（可能是它自己）。在尾递归函数中，因为最后一个动作是函数调用，所以当前函数的栈帧不再需要，可以被新的函数调用重用。

优化原理:

在 factor_opt.s 中，编译器识别到阶乘函数可以通过尾递归优化来实现。它通过将递归结构转换成循环结构，使用循环来迭代减少递归深度，从而避免了每次递归调用都需要新的栈帧。

何时能进行该优化:

尾递归优化可以在函数的最后一个操作是调用函数且不需要使用调用后返回的结果进行其他操作时进行。这通常发生在递归调用中，如果递归调用是函数体中的最后一个动作，并且没有额外的计算依赖于这次调用的返回值。

4.4 理解 switch 语句产生的跳转表

1. 首先，计算 x - 20（通过 addi a5, a0, -20），将结果存入寄存器 a5。这个结果用作跳转表中的索引。然后检查计算得到的索引是否大于 6（bgtu a5, a4, .L8）。如果是，则跳转到处理默认情况的标签.L8。加载跳转表的基地址到 a4，将索引左移 2 位，通过 add 将偏移量加到基地址上，然后通过 lw 加载实际的跳转地址，并通过 jr 指令跳转到相应的代码段。

2.跳转表的优势和劣势：

优势：

通过直接跳转到对应地址，减少了条件判断次数，尤其是在 switch 语句包含许多 case 时更为高效。维护和理解上较为简单，尤其在 case 标签多且分散时。

劣势：如果 **case** 的值分布广泛，跳转表可能会产生大量未使用的条目，造成空间浪费。

主要适用于连续或密集的 **case** 标签。

if-else：

优势：可以根据任何条件进行分支，不受标签值分布的限制，不会产生跳转表可能带来的空间浪费。

在 case 多的情况下，可能需要多次判断，效率低下。大量的 if-else 可能使得代码难以维护和理解。

何时使用跳转表：

当 case 的值较为连续或集中时，使用跳转表可以极大提高效率，当 switch 语句包含大量 case，且执行效率是关键考虑因素时。

何时使用 if-else：

当 case 的值分布广泛，且难以通过计算直接映射到连续的索引时，当需要根据复杂的逻辑条件进行分支，而不仅仅是简单的值比较时。

Bubblesort.s 代码及解释：

```
bubble_sort:
    # 输入: a0 = arr (数组的地址), a1 = len (数组长度
    # 保存寄存器
    addi sp,sp,-32 #腾出栈，修改sp指针
    sd ra,16(sp) #设置返回地址ra
    sd s0,0(sp) #保存帧指针
    addi s0,sp,32 #设置新的帧指针
    li t0,0 #设置t0的值为-1
    j .outer_loop #进入外循环

.outer_loop:  #外循环

    li t1,0 #初始化t1的值为0
    blt t0,a1,.inner_loop #if(t0<len)进入内循环
    j .end
```

```
.inner_loop:  #内循环
    slli t2,t1,3 #每个longlong为8字节
    add t2,a0,t2 #将t2置为&arr[t1]
    ld t3,0(t2) #加载t2中的值至t3,即t3=arr[t1]
    ld t4,8(t2) #加载t2+1中的值至t4,即t4=arr[t1+1]
    ble t3,t4,.no_swap #如果t3<t4,则跳过交换

    sd t3,8(t2) #交换元素
    sd t4,0(t2)

.no_swap: #如果没有交换
    addi t1,t1,1 #t1+=1
    li a2,0
    add a2,a1,-1
    blt t1,a2,.inner_loop #如果t1<a1-1,则继续内循环
    addi t0,t0,1
    bne t0,a1,.outer_loop #如果a1>t0,则继续外循环
.end:  #程序结束，进行返回
    # 恢复寄存器并返回
    ld      ra, 16(sp)          # 恢复返回地址
    ld      s0, 0(sp)          # 恢复帧指针
    addi    sp, sp, 32      # 调整栈指针
```

Fibonacci.s 代码及解释

```
fibonacci:
    # 检查基础情况
    li      t0, 2
    blt     a0, t0, base_case  # 如果 n < 2, 进入基本情况处理

    # 为递归调用保存必要的寄存器
    addi sp, sp, -24 # 为ra, a0, t1腾出空间
    sd ra, 16(sp) # 保存返回地址
    sd a0, 8(sp)  # 保存当前的n值
    sd t1, 0(sp)  # 保存t1

    # 计算 fibonacci(n-1)
    addi a0, a0, -1
    call fibonacci
    mv  t1, a0 # 将结果保存到t1

    # 计算 fibonacci(n-2)
    ld a0, 8(sp) # 从栈恢复原始的n
    addi a0, a0, -2
    call fibonacci
    add  a0, a0, t1 # a0 = fibonacci(n-1) + fibonacci(n-2)
```

```
    # 恢复寄存器并从递归返回
    ld  t1, 0(sp)  # 恢复t1
    ld  ra, 16(sp)  # 恢复ra
    addi sp, sp, 24 # 释放栈空间
    ret

base_case:
    li      a0, 1               # 对于 n = 0 或 n = 1, 返回 1
    ret
```

Lab5-2 Report

Phase_1

三个 phase 均要求 ret=1 才能通过。在 24 行设置断点后，通过 bt full 指令查看 data 的值，发现是 8.阅读 c 代码可知，仅需输入的第一个字符为 8 即可，输入 888，phase1 通过。

```
Breakpoint 9, phase_1 (str=0x40000030c0 <input_buffer> "12345") at challenge.c:24
24          int ret = 1;
(gdb) bt
#0  phase_1 (str=0x40000030c0 <input_buffer> "12345") at challenge.c:24
#1  0x0000004000000d94 in main () at challenge.c:97
(gdb) bt full
#0  phase_1 (str=0x40000030c0 <input_buffer> "12345") at challenge.c:24
        data = 8
        iter = -1
        ret = 0
#1  0x0000004000000d94 in main () at challenge.c:97
No locals.
(gdb)
```

```
root@LAPTOP-78LSF82F:/mnt/d/zju/system/sys1-sp24/src/lab5-2# qemu-riscv64 challenge
please input your student ID:3230104567
Welcome to my fiendish little bomb. You have 3 phases with
88888
Phase 1 defused. How about the next one?
```

Phase_2

阅读 c 代码可知，这个 phase 中的 data 会和自己进行操作，所以无法通过直接观察变量的值获得结果。在 46 行设置断点，可知 data 在和 str 进行操作之前的值。要使得 data 和 str 在运算后相等（data[1] = (data[1] + char2num(str[0])) & 0xf）经过计算之后可知应该输入的三个字符是 fbb，验证如图。

```
Breakpoint 1, phase_2 (str=0x40000030c0 <input_buffer> "fd2") at challenge.c:49
49          int ret = 1;
(gdb) bt full
#0  phase_2 (str=0x40000030c0 <input_buffer> "fd2") at challenge.c:49
        data = {15, 11, 13}
        iter = -1
        ret = 0
```

```
46
(gdb) bt full
#0  phase_2 (str=0x40000030c0 <input_buffer> "1234") at challenge.c:46
        data = {15, 12, 0}
        iter = -1
        ret = 0
#1  0x0000004000000dc4 in main () at challenge.c:105
No locals.
```

```
please input your student ID:3230104567
Welcome to my fiendish little bomb. You have 3 phases with
888
Phase 1 defused. How about the next one?
fbb
Phase 2 defused. How about the next one?
```

Phase_3

首先设置断点，可知 sum 的值为 55，通过代码可知，这个值是不会改变的。通过阅读汇编代码，可知有一步是 t0=t0 xor t2，而 t2 又等于 t1，即为输入 str 的值。而要使得 ret=0，必须要不满足跳转的条件，即 t0=0.而 t0 为 sum 的值，55 和自己异或为 0.所以 t1=55 即可。而 55 显然是个 16 进制数，故输入的数为 7，如图输入 777 即可正确拆除 bomb!

```
(gdb) bt full
#0  phase_3 (str=0x40000030c0 <input_buffer> "12345") at challenge.c:77
        sum = 55
        ret = 12480
#1  0x0000004000000e06 in main () at challenge.c:113
```

```
asm volatile(
    "mv t0, %[sum]\n"//载入sum
    "mv t1, %[str]\n"//载入str的地址
    "j phase_3_L1\n"
    "phase_3_L2:\n"
    "xor t0, t0, t2\n"//t0 = t0 ^ t2
    "addi t1, t1, 1\n"//t1 = t1 + 1
    "phase_3_L1:\n"
    "lbu t2, 0(t1)\n"//t2 = *t1
    "bne t2, zero, phase_3_L2\n"//if t2 != 0, jump to phase_3_L2
    "bne t0, zero, phase_3_L3\n"//if t0 != 0, jump to phase_3_L3
    "mv %[ret], zero\n"//ret = 0
    "phase_3_L3:\n"
    "nop\n"
    :[sum] "=r" (sum),[ret] "=r" (ret)//输出
    :[str] "r" (str)//输入
    :"memory","t0","t1","t2"//使用的寄存器
```

```
root@LAPTOP-78LSF82F:/mnt/d/zju/system/sys1-sp24/src/lab5-2# qemu-riscv64 challenge
please input your student ID:3230104567
Welcome to my fiendish little bomb. You have 3 phases with
888
Phase 1 defused. How about the next one?
fbb
Phase 2 defused. How about the next one?
777
Phase 3 defused.
root@LAPTOP-78LSF82F:/mnt/d/zju/system/sys1-sp24/src/lab5-2#
```