

计算机系统 I 实验报告 5

实验 5-1: RISC-V 汇编程序设计

实验 5-2: RISC-V 汇编程序调试

姓名: 洪奕迅

学号: 3230102930

班号: 工信 2319

计算机系统 I

(春夏, 2024)

浙江大学

计算机学院

2024 年 5 月 26 日

目录

1 实验 5-1: RISC-V 汇编程序设计	2
1.1 理解简单 RISC-V 程序	2
1.1.1 参数传递	2
1.1.2 s0 寄存器	2
1.1.3 栈帧	3
1.1.4 栈存储的值	3
1.1.5 for-循环的实现	3
1.1.6 编译器优化	4
1.2 理解递归汇编程序	5
1.2.1 返回值保存	5
1.2.2 递归参数传递	6
1.2.3 递归函数调用实例	6
1.2.4 编译器对尾递归的优化	6
1.3 理解跳转表汇编程序	8
1.3.1 跳转表实现 switch	8
1.3.2 if-else 和 switch 的优劣	8
1.4 冒泡排序汇编设计	9
1.5 斐波那契数列汇编设计	10
2 实验 5-2: RISC-V 汇编程序调试	11
2.1 QEMU 调试	11
2.1.1 Phase1	11
2.1.2 Phase2	13
2.1.3 Phase3	15
2.2 BONUS: SPIKE 调试	16
2.3 BONUS: 理解串口输出	17
A vsnprintf	20

实验 5-1：RISC-V 汇编程序设计

1.1 理解简单 RISC-V 程序

1.1.1 参数传递

acc 是如何获得函数参数的，又是如何返回函数返回值的？

在 RISC-V 中参数的传递通过函数寄存器实现。

Theorem 1.1.1 ▶ RISC-V 中的参数寄存器

参数寄存器是 RISC-V 中的一类寄存器，这些寄存器用于传递函数参数，并且属于 *caller-saved type*，因此不同于汇编器生成时使用的类似 *s0* 等寄存器需要存储并且保证前后其值不变，参数寄存器并不一定要将其存入栈空间。参数寄存器共有 *a0 ~ a7*，在函数调用时其参数会按顺序传入这些寄存器，在参数过多的情况下则会调用栈空间。而其中的 *a0 ~ a1* 还将作为返回参数寄存器，在位宽足够时只使用 *a0*，否则则会调用 *a1* 加以扩展。

因此考虑 *acc* 的参数类型可知，函数的两个参数将分别传递给 *a0* 和 *a1*（因为一个寄存器足以存储 64 位的 *long long* 类型）。同样的，在返回的时候返回值也会被存在 *a0* 之中并传递回主程序。

1.1.2 *s0* 寄存器

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？

Theorem 1.1.2 ▶ RISC-V 中的保存寄存器

保存寄存器包括 *s0 ~ s11*，这类寄存器属于 *callee-saved*，因此在函数调用时需要先进行保存并确保寄存器保存值前后没有发生改变。而其中 *s0* 寄存器还被作为帧指针（Frame Pointer）使用。

而对于这段汇编器生成的代码，*s0* 的作用正是作为帧指针使用。因此 *sd s0, 40(sp)* 的作用正是为了存储 *s0* 的原值，这是自身寄存器属性决定的，因为保存寄存器在函数调用前后需要保证值不变。而 *addi s0, sp, 48* 则体现了其为**帧指针**的作用。众所周知，函数的存在仅限于函数运行本身，因此我们需要保证栈指针的值并不受到函数调用影响，而 *s0* 记录了这个值，建立了函数栈与原程序的联系，起到了**释放内存**的作用。

1.1.3 栈帧

acc 函数的栈帧 (stack frame) 的大小是多少？

在 1.1.2 中以及提及了帧指针，而其也标志了栈顶位置，栈帧大小即 48 Bytes。

1.1.4 栈存储的值

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）？

栈帧中存储了以下寄存器：

Offset by <i>s0</i>	变量
- 8	Saved register <i>s0</i>
- 48	<i>a1</i>
- 40	<i>a0</i>
- 32	<i>zero</i>
- 24	<i>a5</i>

这主要是标签 *acc* 初始化传递参数之后的结果，在执行接下来几个标签之后 *zero* 寄存器将被 *a5* 取代，最终实现循环的作用：

Offset by <i>s0</i>	变量
- 8	Saved register <i>s0</i>
- 48	传入参数 <i>a1</i>
- 40	返回值 <i>a0</i>
- 32	临时求和结果
- 24	循环计数器

1.1.5 for-循环的实现

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。

由于 RISC-V 汇编代码本质仍然是按照顺序逐条执行的，而对于结构的控制依赖于跳转语句，接下

来通过各个标签在跳转语句的控制下的结合来解释汇编怎么实现循环：

- **acc**: 初始标签完成的工作主要是参数的传递和进行保存寄存器的原值保存，其中在 $-40(sp)$, $-48(sp)$ 中分别存储了两个传入参数 $a0 = a, a1 = b$ ，而在 $-32(sp), -24(sp)$ 处则分别存储了辅助寄存器：零寄存器和用于计数的 $a5$ 寄存器，其初值等于传入参数 a 。
- **.L2**: L2 阶段相当于实现了循环的判断语句，而汇编器生成的整个代码结构可能同 **while** 更为相似。在这个阶段，首先先是利用 $a4$ 暂存原来 $a5$ 的值，再用 $a5$ 存储传入参数 b 进行循环退出条件判断，若小于等于则跳入循环体 **.L3**。我们先跳过对于循环体的描述，只需要知道在循环体内完成了求和，临时结果被存储在 $-32(sp)$ 之中。因此 **.L2** 的最后阶段就是将这一结果存入返回参数寄存器 $s0$ ，并进行栈帧释放。
- **.L3**: L3 阶段是循环体阶段，可能是出于节省寄存器的原因，汇编器生成的代码总是将较少的几个寄存器不断修改值作为不同阶段不同作用的变量，这导致代码非常难懂。但简而言之，这一阶段就是取出临时求和结果与循环计数器相加并通过 $a5$ 存回临时求和结果所在的位置。最后再还原 $a5$ 的值并为其加一。而循环判断的实现则依靠汇编本身逐行执行的特性保证，L3 阶段以后会直接进入 L2 进行循环判断。

1.1.6 编译器优化

请查阅资料简要描述编译选项 $-O0$ 和 $-O2$ 的区别。

Theorem 1.1.3 ▶ 编译器优化

编译器优化选项是在执行编译操作时可选的参数，编译器会根据规则对源代码进行优化，这在汇编代码上体现于代码结构的变化，一方面能够简化汇编代码的结构，另一方面在一些情况下可以降低时间复杂度（可以去任意 OJ 网站打开 $O2$ 优化测试 (x)）。而这些编译器优化选项大致包括以下几类（基于 gcc 文档）：

- $-O0$: 默认编译操作，不进行优化
- $-O1$: 最初等的优化，编译时会消耗稍多的内存和时间，主要是对分支、表达式、常量的优化，这涉及到了相当多的编译语句，大致包括：
 1. 常量优化：合并常量
 2. 分支优化：将条件跳转优化为无分支跳转，根据跳转目的地调整分支语句顺序，删除无意义的寄存器复制，引入随机化技术进行分支猜测
 3. 表达式优化：从循环中移除常量表达式
 4. 函数优化：减少无意义参数弹出，而是调整为多个函数执行后一起弹出
- $-O2$: 在 $-O1$ 已有优化基础上进行了进一步的优化，相应会消耗更多时间，优化范围进一步拓展到寄存器和指令，由于涉及的优化过多这里不作详细说明，包括对循环和递归的展开，对指令顺序的调整，对相同指令的整合等等
- $-O3$: 在 $-O2$ 基础上加入了伪寄存器网络和普通函数的内联等

此外还有 Os , $O-fast$ 等各种优化选项，参照：GCC 文档

请简要讨论 `src/lab5 - 1/accopt.s` 与 `src/lab5 - 1/accplain.s` 的优劣。

上面已经讨论了编译器优化的问题，再来讨论其导致的优劣问题：

- `-O0`:

1. 由于编译器进行的优化大多数时候相当抽象，因此未经优化的代码本身具有更强可读性，也更为符合逻辑，但是私以为在这里不成立，由于累加功能极为简单导致主要优化是移除了栈的使用，我反而认为优化后有更强可读性。
2. 由于不需要编译器进行优化，编译本身消耗的时间和内存也更小
3. 由于未经优化，函数本身会消耗更多的栈空间和时间，在累加函数中体现为摒弃了栈空间

- `-O2`

1. 同前，优化会使用函数内联等操作，从而降低程序可读性，当然这个函数由于这些都没有，更短的汇编代码反而增强了可读性，减少了无意义的寄存器复制操作
2. 优化会消耗较多的内存和时间
3. 经过优化的函数会减少无意义指令，提升运行速度，减少无意义空间使用

1.2 理解递归汇编程序

1.2.1 返回值保存

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

Theorem 1.2.1 ▶ 叶函数和非叶函数

叶函数和非叶函数是从内部调用方式对函数的区分。**叶函数**在函数内部不会发生对其他函数的调用。而**非叶函数**则不同，其内部会调用其他函数。天然的，这要求非叶函数调用过程中需要保存更多的寄存器，最典型的需要保存返回地址，在调用另一个函数之前提前将返回地址压入栈空间，从而能够正确返回。

这是因为在阶乘函数中我们使用了递归的实现方式，而在原来的求和中使用的迭代的方式。递归函数很显然是一类特殊的非叶函数，即调用自身，但他依然需要满足非叶函数的基本约定，在进行函数跳转之前需要先保存返回地址，即 `sd ra, 24(sp)`。而迭代实现的累加函数，所有指令局限于函数内部，不存在不同函数间的跳转，自然不需要保存返回值。

1.2.2 递归参数传递

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

此前已经提过 RISC-V 的函数参数传递规则，而在我们递归调用前，自然也需要先进行参数的传递，因此要先把计算好的下次函数的存储在 `a5` 寄存器的输入值 ($n = n - 2$) 传递给参数寄存器 `a0`，从而使下一次函数调用有正确的参数输入。

1.2.3 递归函数调用实例

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。

对于单次调用函数栈的使用大概是这样的：

- 分配 32-Bytes 的栈空间
- 保存返回值 `ra` 和帧指针 `s0` 原值
- 保存参数 `a0`
- 递归调用函数

对于每次函数调用都会新建栈空间，且在不能进入 `L3` 阶段前这些栈不会被清空，因此在 `factor(0)` 处达到内存占用最大值 $MAX = 11 \times 32\text{Bytes} = 352\text{Bytes}$ ，此后每次跳转返回都会不断释放栈空间。

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

$$\text{同上问, } n_{\max} = \frac{4096\text{B}}{32\text{B}} - 1 = 127$$

1.2.4 编译器对尾递归的优化

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

肉眼可见的，在优化后的汇编程序中栈指针直接消失了，说明在优化后的函数中栈空间的调用已经不再存在了。代码变得更为简短。这也是两者主要的区别。

很显然，在优化后的程序中根本没有使用栈空间，也不存在递归调用，从内存角度优化后的函数远强于原始函数。

接下来说明为什么编译器会做出这样的优化：

Theorem 1.2.2 ▶ 尾递归

尾递归是一种特别的尾调用。尾调用即对于其他函数的调用只在函数尾部发生，而尾递归则在自身尾部调用了自己，因此可以通过优化只占用常量空间。

递归对于内存的占用体现在每次都需要保存栈帧上。因此我们会有一个相当长的调用链条。例如对于没有经过优化的阶乘函数：

Code Snippet 1.2.3 ▶ *factor_plain.c*

```
1 long long factor(long long n) {
2     if (n == 0) {
3         return 1;
4     }
5     return n * factor(n - 1);
6 }
7 //在计算 factor(6) 时会产生这样一个递归链
8 /*
9 6 * fact(5)
10 6 * (5 * fact(4))
11 6 * (5 * (4 * fact(3)))
12 ...
13 6 * (5 * (4 * (3 * (2 * 1))))
14 */
```

而产生这个递归链的原因就在于每次计算总是依赖于下一级的递归函数的结果，而如果我们将其改写成尾递归的形式就不再会出现这样的问题：

Code Snippet 1.2.4 ▶ *factor_plain.c*

```
1 long long factor(long long n, long long result) {
2     if (n == 0) {
3         return 1*result;
4     }
5     return factor(n-1, n*r);
6 }
7 //现在展开后会发现每次递归调用都是单独的过程
8 fact(6, 1)
9 fact(5, 6)
10 fact(4, 30)
```



```
11 fact(3, 120)
12 fact(2, 360)
13 fact(1, 720)
```

即然每次递归调用都只要继续向下计算而不需要将结果返回给前级，那我们也不再需要保存返回地址，甚至栈空间可以被完全舍弃，因为我们只需要保证参数被计算传递，而前级的局部变量地址对后级运算没有任何作用。而这样优化后的尾递归就会被编译器直接优化成迭代的形式，因为如果不需要保存返回入口的信息，这样的递归其实和循环是完全等价的，而在汇编层面上，开启 `-O2` 优化以后，能够被优化成尾递归的函数也会被优化成类似迭代的形式。

编译器在实现这种优化的时候主要使用了以下的命令：

- `-foptimize-sibling-calls`: 在处理能够尾递归化的函数时，这条指令会把递归函数展开成一系列分立语句
- `-fcaller-saves`: 这个选项使函数能够访问寄存器值，而不必恢复和保存他们

1.3 理解跳转表汇编程序

1.3.1 跳转表实现 switch

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

`Switch` 语句在 RISC-V 中是通过一张跳转表实现的，通过一系列的运算我们将传递参数 `a0` 计算为对应的表项地址，然后跳转至跳转表的对应表项来进行对应计算。

1.3.2 if-else 和 switch 的优劣

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`。

不难发现，我们的跳转表是一张线性表，其所有表项在内存中有一个连续地址。因此如果我们要实现的跳转不是一个线性的流程，采用 `if-else` 能够减少空间的浪费。但是 `if-else` 在时间上对 `switch` 语句没有优势，因为需要逐步检查是否满足目标条件而不能直接跳转，这会带来最大 $O(N)$ 的时间复杂度，因此在大规模连续分支的情况下使用跳转表会有更好的时间表现。

总的来说，逻辑相对简单且互相不连续的情况下使用 `if-else`，对性能有更高要求且分支多而连续的情况下优先考虑跳转表。

1.4 冒泡排序汇编设计

在回答了这么多的问题之后，总算可以开始进行汇编代码的设计了，对于代码的说明会在注释中写明：

Code Snippet 1.4.1 ▶ src/lab5-1/bubble_sort.s

```

1  bubble_sort:
2      addi    sp, sp, -32          # 分配栈空间
3      sd      ra, 24(sp)          # 保存 ra
4      sd      s0, 16(sp)          # 保存 s0
5      addi    s0, sp, 32          # 设置帧指针
6      sd      a0, -32(s0)         # 保存 arr
7      sd      a1, -24(s0)         # 保存 len
8      addi    t0, a1, -1          # 外循环计数器
9  loop1:
10     mv      t2, a0              # 数组起始地址
11     li      t3, 0               # 内循环计数器
12  loop2:
13     bge     t3, t0, judge        # 如果 t3 >= t0, 跳出外循环
14     ld      t4, 0(t2)            # t4 = arr[j]
15     ld      t5, 8(t2)           # t5 = arr[j+1]
16     ble     t4, t5, no_swap      # 如果 arr[j+1] <= arr[j], 不交换
17     sd      t5, 0(t2)           # arr[j] = t5
18     sd      t4, 8(t2)           # arr[j+1] = t4
19  no_swap:
20     addi    t2, t2, 8            # 读出下一个数组元素
21     addi    t3, t3, 1            # t3 = t3 + 1
22     j       loop2               # 进入内循环
23  judge:
24     addi    t0, t0, -1           # 外循环计数器-1
25     ble     zero, t0, loop1      # 若还需进行外循环则跳回
26                                     # 不然开始收尾
27  exit:
28     ld      ra, 24(sp)
29     ld      s0, 16(sp)
30     addi    sp, sp, 32
31     jr      ra

```

1.5 斐波那契数列汇编设计

同理，具体解释详见注释

Code Snippet 1.5.1 ▶ src/lab5-1/bubble_sort.s

```

1  fibonacci:
2      li      t0, 2
3      bge     a0, t0, recursive    # 若没到递归边界就开始递归
4      li      t1, 0
5      li      t2, 1
6      beq     a0, t0, return       # 两个递归边界判定
7      beq     a0, t1, return
8      ret                                #return (C 代码意义上的)
9  return:
10     li      a0, 1                 # 边界返回值设置为 1
11     ret
12  recursive:
13     addi     sp, sp, -24
14     sd       ra, 0(sp)
15     sd       s0, 8(sp)
16     sd       s1, 16(sp)
17     mv       s0, a0
18     addi     a0, s0, -1           #fibo(n-1)
19     call     fibonacci
20     mv       s1, a0              # 暂存返回值
21     addi     a0, s0, -2           #fibo(n-2)
22     call     fibonacci
23     add      a0, s1, a0           # 两个返回值相加
24     ld       ra, 0(sp)           # 清理阶段
25     ld       s0, 8(sp)
26     ld       s1, 16(sp)
27     addi     sp, sp, 24
28     ret

```

实验 5-2: RISC-V 汇编程序调试

2.1 QEMU 调试

三部分调试整体思路相同，都通过分析语句找到重要变量再借助

2.1.1 Phase1

Code Snippet 2.1.1 ▶ Phase1

```
1  int phase_1(const char* str){//phase01=0
2      int data = char2num(name_buffer[6]);
3      int iter = 20;
4      while(iter--){
5          data = (phase_box[data]^phase_box[(data+3)&0xf]
6              ^phase_box[phase_box[data]])&0xf;
7      }
8      //printf("%d\n",data);
9      int ret = 1;
10     while(*str){
11         if(char2num(*str) == data){
12             ret = 0;
13             break;
14         }
15         str++;
16     }
17     return ret;
```

阅读 Phase-1 的代码部分，不难发现最后判断是否破译成功的条件仅仅是 `data == char2num(*str)`。那么就非常好办了，我们并不关心这个复杂的迭代过程到底在干什么，只需要知道 `data` 的值即可。

因此不妨使用 `gdb` 运行至迭代完成然后利用 `gdb` 指令查看 `data` 的值，执行以下指令：

Code Snippet 2.1.2 ▶ 调试指令

```
1  qemu-riscv64 -g 1234 challenge (terminal1)
2  gdb-multiarch challenge
3  target remote localhost:1234 # 连接远程端口
4  b phase_1 # 在 phase_1 函数处断点
5  c # 运行至断点处
6  p data # 输出 data 值
7  b phase_2 # 跳至 phase_2 处
8  c # 这样能看到第一次的输出结果
```

可以看到终端直接打印了 *data* 的值，然后重新打开输入这个值并跳转至下一个函数时，可以看到程序输出我们破译成功：

```
Breakpoint 1, phase_1 (str=0x40000030c0 <input_buffer> "0") at challenge.c:19
19      int data = char2num(name_buffer[6]);
(gdb) p data
$1 = 0
```

图 2.1: 调试指令

```
Welcome to my fiendish little bomb. You have 3 phases with
0

Phase 1 defused. How about the next one?
```

图 2.2: 程序输出

2.1.2 Phase2

Code Snippet 2.1.3 ▶ Phase_2

```

1  int phase_2(const char* str){//phase02=ca0
2      if(!(str[0] && str[1] && str[2])){
3          return 1;
4      }
5      int data[3] =
6      {char2num(name_buffer[7]),char2num(name_buffer[8]),
7      char2num(name_buffer[9])};
8      int iter = 100;
9      while(iter--){
10         data[0] =
11         (phase_box[phase_box[data[0]]]^phase_box[data[1]]
12         ^phase_box[data[2]])&0xf;
13         data[1] =
14         (phase_box[data[0]]^phase_box[phase_box[data[1]]]
15         ^phase_box[data[2]])&0xf;
16         data[2] =
17         (phase_box[data[0]]^phase_box[data[1]]
18         ^phase_box[phase_box[data[2]]])&0xf;
19     }
20     data[1] = (data[1] + char2num(str[0])) & 0xf;
21     data[2] = (data[2] + char2num(str[1])) & 0xf;
22     int ret = 1;
23     if(char2num(str[0])==data[0] && char2num(str[1])==data[1] &&
24     ↵ char2num(str[2])==data[2]){
25         ret = 0;
26     }
27     return ret;
28 }

```

在最后的判断条件中我们注意到了三个重要的迭代结果值：*data[0]*, *data[1]*, *data[2]*，因此我们首先要通过 *gdb* 来获得这些关键值，再结合最后的判断语句得出正确的输入语句。

在 *gdb* 中输入以下语句：

Code Snippet 2.1.4 ▶ 调试命令

```
1 b 46    # 跳至中间值计算完成前
2 c       # 跳转至断点
3 p data  # 打印数组所有元素值
```

于是我们就知道了 `data[0]`, `data[1]`, `data[2]` 的结果:

```
Breakpoint 4, phase_2 (str=0x40000030c0 <input_buffer> "ca0") at challenge.c:47
47      data[1] = (data[1] + char2num(str[0])) & 0xf;
(gdb) p data
$4 = {12, 14, 6}
```

图 2.3: 中间值输出值

接下来我们来考虑输入字符串, 可以看到 `data[1]`, `data[2]` 后续还与 `str` 的值有关, 而 `data[0]` 的值不再变动, 由此我们再根据字符转数字函数可以推测出 `str[0] = 'c'`, 将其带入 `data[1]` 表达式, 计算出实际的 `data[1]` 值为 10, 因此 `str[1] = 'a'`, 如法炮制得到 `data[2] = 0`, `str[2] = '0'`, 现在输入 `str = "ca0"`, 可以看到破译成功:

```
ca0
Phase 2 defused. How about the next one?
```

图 2.4: Enter Caption

2.1.3 Phase3

Code Snippet 2.1.5 ▶ Phase_3

```

1  int phase_3(const char* str){
2      int sum =
3      char2num(name_buffer[6])^char2num(name_buffer[7])
4      ^char2num(name_buffer[8])^char2num(name_buffer[9]);
5      int ret = 1;
6      asm volatile(
7          "mv t0, %[sum]\n"
8          "mv t1, %[str]\n"
9          "j phase_3_L1\n"
10         "phase_3_L2:\n"
11         "xor t0, t0, t2\n"
12         "addi t1, t1, 1\n"
13         "phase_3_L1:\n"
14         "lbu t2, 0(t1)\n"
15         "bne t2, zero, phase_3_L2\n"
16         "bne t0, zero, phase_3_L3\n"
17         "mv %[ret], zero\n"
18         "phase_3_L3:\n"
19         "nop\n"
20         :[ret] "=r" (ret)
21         :[sum] "r" (sum),[str] "r" (str)
22         : "memory", "t0", "t1", "t2"
23     );
24     return ret;
25 }

```

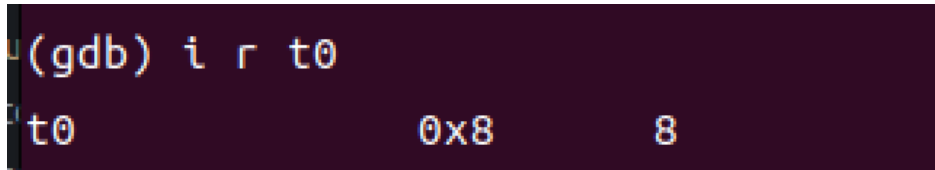
我们首先要理解这段内联的汇编代码，可以看到其实现了一个循环的功能，每次都会将字符串当前的字符的 ASCII 码值与 *sum* 进行异或，将全部字符异或的结果与 0 作对比，如果结果相同则认为破译成功，因此我们只需要知道 *sum* 的值即可，当然最简单的破译字符串就是这个 *sum* 对应的字符值。

输入以下指令来获得 *sum* 的值，这里采用的方式是查看其对应的寄存器：

Code Snippet 2.1.6 ▶ 调试指令

```
1 b 58
2 si 4 # 沿函数向下进行四条汇编命令
3 i r t0 # 查看寄存器的值
```

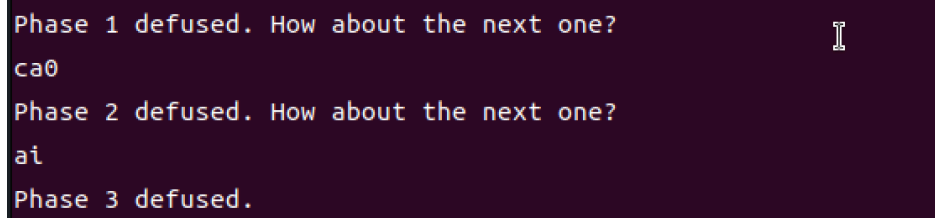
能看到以下内容：



```
(gdb) i r t0
t0                                0x8      8
```

图 2.5: 中间值结果

由于 8 并不是对应一个真正的字符，我们输入一个两位字符串其 ASCII 码异或结果为 8，能够看到破译完成：



```
Phase 1 defused. How about the next one?
ca0
Phase 2 defused. How about the next one?
a!
Phase 3 defused.
```

图 2.6: 破译完成

2.2 BONUS: SPIKE 调试

在这一节我们将通过 gdb 和 spike 模拟器来找到 *printm* 语句背后的函数调用链，主要会使用到 *disassemble* 反汇编语句和 *si* 进入到汇编代码内部逐行运行。

首先对 *printm* 进行反汇编可以看到内部调用了 *vprintm*，然后我们会发现其调用了很多函数，其中 *uart*, *uart16550*, *uart_litex* 应当是在串口上实现了输出的功能，我们后续继续检查：

```
Listening for remote breakpoint 4, 0x000000008000287e in uart16550_putchar ()
(gdb)
```

图 2.9: 执行后的字符输出

```
Dump of assembler code for function printf:
0x000000000001c08 <+0>: addi sp,sp,-96
0x000000000001c0a <+2>: addi t1,sp,40
0x000000000001c0c <+6>: sd a1,40(sp)
0x000000000001c0e <+8>: mv a1,t1
0x000000000001c10 <+10>: sd ra,24(sp)
0x000000000001c12 <+12>: sd a2,48(sp)
0x000000000001c14 <+14>: sd a3,56(sp)
0x000000000001c16 <+16>: sd a4,64(sp)
0x000000000001c18 <+18>: sd a5,72(sp)
0x000000000001c1a <+20>: sd a6,80(sp)
0x000000000001c1c <+22>: sd a7,88(sp)
0x000000000001c1e <+24>: sd t1,8(sp)
0x000000000001c20 <+26>: jal ra,0x00001bda <vprintf>
0x000000000001c22 <+28>: ld ra,24(sp)
0x000000000001c24 <+30>: addi sp,sp,96
0x000000000001c26 <+32>: ret
```

图 2.7: *printf* 的反汇编代码

```
Dump of assembler code for function vprintf:
0x000000000001b0a <+0>: addi sp,sp,-304
0x000000000001b0c <+2>: mv a2,a0
0x000000000001b0e <+4>: mv a3,a1
0x000000000001b10 <+6>: mv a0,sp
0x000000000001b12 <+8>: li a1,256
0x000000000001b14 <+10>: sd ra,296(sp)
0x000000000001b16 <+12>: sd s0,288(sp)
0x000000000001b18 <+14>: sd s1,280(sp)
0x000000000001b1a <+16>: sd s2,272(sp)
0x000000000001b1c <+18>: sd s3,264(sp)
0x000000000001b1e <+20>: sd s4,256(sp)
0x000000000001b20 <+22>: jal ra,0x00001bda <vsnprintf>
0x000000000001b22 <+24>: lbu a0,0(sp)
0x000000000001b24 <+26>: beqz a0,0x00001c40 <vprintf+102>
0x000000000001b26 <+28>: mv s0,sp
0x000000000001b28 <+30>: auipc s1,0x10
0x000000000001b2a <+32>: addi s1,s1,1114 # 0x00012950 <uart>
0x000000000001b2c <+34>: auipc s2,0x10
0x000000000001b2e <+36>: addi s2,s2,1122 # 0x00012950 <uart16550>
0x000000000001b30 <+38>: auipc s3,0x10
0x000000000001b32 <+40>: addi s3,s3,1122 # 0x00012970 <uart_litter>
0x000000000001b34 <+42>: auipc s4,0x10
```

图 2.8: *vprintf* 的汇编代码

在继续查看调用后发现这所有的串口底下都带有对应的 *putchar* 函数,但是测试后只有 *uart16550_putchar* 在执行后会在终端输出字符:

于是我们跳到 *uart16550_putchar* 并开始逐行执行内部代码,发现在执行完 *sb a0,0(a3)* 后输出了字符,因此在 *printf* 中真正输出字符的是这条将值保存到 *a0* 的命令,结合此前对于 *a0* 作用的说明,推测 *printf* 的输出字符函数会输出其返回值,而该语句相当于保存了返回值,因此能够实现输出的功能:

```
forever@forever:~$ c
forever@forever:~/sy
d/riscv-pk/bbl
Listening for remote
ha
B+ 0x8000287e <uart16550_putchar+8> li a4,5
0x80002880 <uart16550_putchar+10> auipc a3,0xf
0x80002884 <uart16550_putchar+14> ld a3,2024(a3)
0x80002888 <uart16550_putchar+18> sllw a4,a4,a5
0x8000288c <uart16550_putchar+22> add a4,a4,a3
0x80002890 <uart16550_putchar+26> lbu a5,0(a4)
0x80002894 <uart16550_putchar+30> andi a5,a5,32
0x80002898 <uart16550_putchar+34> beqz a5,0x8000288e <uart16550_putchar+30>
0x8000289c <uart16550_putchar+38> sb a0,0(a3)
> 0x8000289e <uart16550_putchar+42> ret
0x800028a0 <uart16550_putchar+46> auipc a3,0xf
0x800028a4 <uart16550_putchar+50> lw a3,1986(a3)
0x800028a8 <uart16550_putchar+54> li a5,5

remote Remote target In: uart16550_putchar L?? PC: 0x8000289c
0x0000000080002892 in uart16550_putchar ()
(gdb) si
0x0000000080002896 in uart16550_putchar ()
(gdb) si
```

图 2.10: 执行后输出字符

2.3 BONUS: 理解串口输出

上一节已经找到了真正对应字符输出的函数 *uart16550_putchar*, 现在来观察一下其源码:

Code Snippet 2.3.1 ▶ `repo/riscv-pk/machine/uart16550.c`

```
1 void uart16550_putchar(uint8_t ch)
2 {
3     while ((uart16550[UART_REG_LSR << uart16550_reg_shift] & UART_REG_STATUS_TX) ==
4             0);
5     uart16550[UART_REG_QUEUE << uart16550_reg_shift] = ch;
6 }
```

这可以说是和我们在 lab4-1 和 lab4-2 中实现的东西完全一样了，串口的输出首先是维护了一个移位寄存器，每次在信号可用的时候就进行移位并将字符送入目标移位寄存器。因此在图 2.10 中展示的就是其中的一次移位和入队过程。

同时在浏览了一遍源码之后，找到了调用串口的上层输出函数所在文件 *mtrap.c*;

Code Snippet 2.3.2 ▶ `repo/riscv-pk/machine/mtrap.c`

```
1 void vprintm(const char* s, va_list vl)
2 {
3     char buf[256];
4     vsnprintf(buf, sizeof buf, s, vl);
5     putstring(buf);
6 }
7
8 void printm(const char* s, ...)
9 {
10     va_list vl;
11
12     va_start(vl, s);
13     vprintm(s, vl);
14     va_end(vl);
15 }
```

至此我们不仅可以搞明白这个输出的流程，还可以回答上一节的遗留问题了，即为什么那么多个输出函数里面只有 *uart16550* 类型发挥了应有的作用：在阅读三者源码后发现在实现上并没有太多区别，这些不同的串口通信主要的差异是协议不同即他们遵循不同的握手信号，因此应该是我们的模拟器采用了 *uart_16550* 而导致了 *mtrap.c* 调用了该模块的字符输出。

具体输出流程大概是这样：

1. *printm* 调用 *vprintm*

2. *vprintm* 调用 *vsnprintf* 进行字符串的格式化
 3. 字符串通过 *putstring* 来逐个输出
 4. 根据硬件类型选择合适的串口输出函数，逐字符输出
- 这也是为什么此前保存返回值之后就会产生输出的原因。

vsnprintf

关于此前提到的的 *vsnprintf* 是 riscv-pk 实现的一个字符串格式化函数：

Code Snippet A.0.1 ► repo/riscv-pk/util/snprintf.c

```
1  int vsnprintf(char* out, size_t n, const char* s, va_list vl)
2  {
3      bool format = false;
4      bool longarg = false;
5      bool longlongarg = false;
6      size_t pos = 0;
7      for( ; *s; s++)
8      {
9          if(format)
10             {
11                 switch(*s)
12                 {
13                     case 'l':
14                         if (s[1] == 'l') {
15                             longlongarg = true;
16                             s++;
17                         }
18                         else
19                             longarg = true;
20                         break;
21                     case 'p':
22                         longarg = true;
23                         if (++pos < n) out[pos-1] = '0';
24                         if (++pos < n) out[pos-1] = 'x';
25                     case 'x':
26                         {
27                             long num = longarg ? va_arg(vl, long) : va_arg(vl, int);
28                             for(int i = 2*(longarg ? sizeof(long) : sizeof(int))-1; i >= 0; i--) {
29                                 int d = (num >> (4*i)) & 0xF;
```

```
30         if (++pos < n) out[pos-1] = (d < 10 ? '0'+d : 'a'+d-10);
31     }
32     longarg = false;
33     format = false;
34     break;
35 }
36 case 'd':
37 {
38     long long num;
39     if (longarg)
40         num = va_arg(vl, long);
41     else if (longlongarg)
42         num = va_arg(vl, long long);
43     else
44         num = va_arg(vl, int);
45     if (num < 0) {
46         num = -num;
47         if (++pos < n) out[pos-1] = '-';
48     }
49     long digits = 1;
50     for (long long nn = num; nn /= 10; digits++)
51         ;
52     for (int i = digits-1; i >= 0; i--) {
53         if (pos + i + 1 < n) out[pos + i] = '0' + (num % 10);
54         num /= 10;
55     }
56     pos += digits;
57     longarg = false;
58     longlongarg = false;
59     format = false;
60     break;
61 }
62 case 's':
63 {
64     const char* s2 = va_arg(vl, const char*);
65     while (*s2) {
66         if (++pos < n)
67             out[pos-1] = *s2;
```

```
68         s2++;
69     }
70     longarg = false;
71     format = false;
72     break;
73 }
74 case 'c':
75 {
76     if (++pos < n) out[pos-1] = (char)va_arg(vl,int);
77     longarg = false;
78     format = false;
79     break;
80 }
81 default:
82     break;
83 }
84 }
85 else if(*s == '%')
86     format = true;
87 else
88     if (++pos < n) out[pos-1] = *s;
89 }
90 if (pos < n)
91     out[pos] = 0;
92 else if (n)
93     out[n-1] = 0;
94 return pos;
95 }
```

我并没有进行逐行的阅读，因此放在附录中，大致来说，这个函数会根据给出的格式符来处理对应类型的数据。