

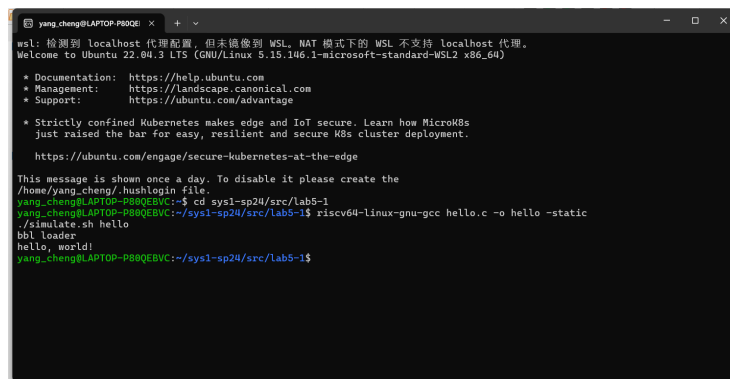
Lab 5

Chenghao Jiang

Date:2024-05-13

Lab 5-1: RISC-V 汇编程序设计

1 实验环境配置



```
yang_cheng@LAPTOP-PR0QE: ~  
wsl: 检测到 localhost 代理配置, 但未检测到 WSL. NAT 模式下的 WSL 不支持 localhost 代理。  
Welcome to Ubuntu 22.04.3 LTS (GNU/Linux 5.15.146.1-microsoft-standard-WSL2 x86_64)  
  
 * Documentation:  https://help.ubuntu.com  
 * Management:   https://landscape.canonical.com  
 * Support:       https://ubuntu.com/advantage  
  
 * Strictly confined Kubernetes makes edge and IoT secure. Learn how MicroK8s  
   just raised the bar for easy, resilient and secure K8s cluster deployment.  
   https://ubuntu.com/engage/secure-kubernetes-at-the-edge  
  
This message is shown once a day. To disable it please create the  
/home/yang_cheng/.hushlogin file.  
yang_cheng@LAPTOP-PR0QE: ~$ cd sys1-sp24/src/lab5-1  
yang_cheng@LAPTOP-PR0QE: ~/sys1-sp24/src/lab5-1$ riscv64-linux-gnu-gcc hello.c -o hello -static  
./simulate.sh hello  
hbl loader  
hello, world!  
yang_cheng@LAPTOP-PR0QE: ~/sys1-sp24/src/lab5-1$
```

Hello World

2 理解简单 RISC-V 程序

Problem 1

acc是如何获得函数参数的, 又是如何返回函数返回值的?

函数的两个参数分别通过寄存器 a0 和 a1 传递进来。在函数的开头, 寄存器 a0 和 a1 的值被分别保存到栈上的偏移为 -40(s0) 和 -48(s0) 的位置。

函数返回值通常是通过 a0 寄存器传递给调用者的。在函数结束前，将要返回的值保存在 a0 寄存器中。

Problem 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？

s0 寄存器被用作帧指针，指向当前函数的栈帧。它在函数的入口被初始化为栈指针的值，然后在函数结束前用来恢复上一个函数的帧指针。

```
1 sd s0,40(sp)
```

这条指令将当前帧指针 s0 的值保存到当前栈帧的偏移为 40 的位置。这样做的目的是在函数结束时，可以从栈中恢复上一个函数的帧指针的值，以保证正确的函数调用关系。

```
1 addi s0, sp, 48
```

这条指令将栈指针 sp 的值加上 48，然后将结果存入 s0 寄存器中。这个操作的目的是设置 s0 寄存器，使其指向当前栈帧的底部，以便在函数执行期间可以通过帧指针访问函数的局部变量和保存的寄存器值。

Problem 3

acc函数的栈帧 (stack frame) 的大小是多少？

addi sp,sp,-48 这条指令用于在栈上分配空间。它将栈指针(sp)减去48，以便为局部变量和保存的寄存器值留出空间。因此，acc函数的栈帧大小为48字节。

Problem 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）？

保存的上一个帧指针 (s0)：存储在栈帧中偏移为 40(sp) 的位置。

参数 a0：存储在栈帧中偏移为 -40(s0) 的位置。

参数 **a1**: 存储在栈帧中偏移为 -48(s0) 的位置。

临时变量 (**res**): 存储在栈帧中偏移为 -32(s0) 的位置。

临时变量 (**i**): 存储在栈帧中偏移为 -24(s0) 的位置。

Problem 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。

循环起始标签: 在汇编代码中, 循环的起始点被标记为.L3, 这是一个标签。在此标签之后是循环的实际代码。

循环条件判断: 循环体内部的代码执行结束后, 会执行一条分支指令 `ble a4, a5, .L3`。该指令的作用是将 `a4` 与 `a5` 进行比较, 如果 `a4` 小于等于 `a5`, 则跳转到.L3标签处, 继续执行循环体内的代码。如果 `a4` 大于 `a5`, 则跳出循环, 执行循环后的代码。

循环体内部的操作: 循环体内部的操作是累加计算的核心。

循环的结束条件: 循环结束的条件是 `a4` 大于 `a5`, 即 `a4` 的值超过了循环计数的上限。

Problem 6

请查阅资料简要描述编译选项 **-O0** 和 **-O2** 的区别。

编译器选项 **-O0** 和 **-O2** 分别代表了不同的优化级别。

-O0 是编译器的最低优化级别, 通常被称为“无优化”或“零级优化”。在这个级别下, 编译器会生成简单直观的机器代码。它不进行任何优化, 包括不执行内联、循环展开、函数调用优化等。

-O2 是编译器的较高优化级别之一, 通常被称为“优化级别 2”。在这个级别下, 编译器会执行一系列优化, 包括但不限于内联函数、循环展开、指令调度、常量折叠等。

Problem 7

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

-O0 适用于调试和理解代码, 编译速度较快, 但生成的代码可能不够高效。

-O2 则适用于生产环境中追求性能的场景，生成的代码更加紧凑和高效，执行速度更快，但编译时间可能会更长。

3 理解简单 RISC-V 程序

Problem 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？

在 factor_plain.s 中的 factor 函数中，由于涉及递归调用，函数需要显式地保存返回地址 (ra) 到栈帧中。这是因为每次递归调用时，返回地址都会被覆盖，所以需要在函数入口处将其保存起来，以便在递归调用返回时能正确找到返回地址，从而顺利返回到上一级调用。

而在 acc_plain.s 中的 acc 函数中，没有涉及递归调用，因此不需要在函数入口处显式地保存返回地址到栈帧中。在非递归函数中，一般情况下，返回地址都会存储在 ra 寄存器中，并在函数调用过程中自动保存和恢复，所以不需要额外的处理。

Problem 2

请解释在call factor前的mv a0, a5这条汇编指令的目的。

在 mv a0, a5 这条指令之前，a5 寄存器中存储的是当前函数 factor 需要计算阶乘的值。而在 call factor 指令中，函数参数是通过 a0 寄存器传递的，因此在调用 factor 函数之前，需要将待计算阶乘的值从 a5 寄存器移动到 a0 寄存器中，以便将其作为参数传递给 factor 函数。

Problem 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。

初始时，调用 factor(10) 函数前，栈上分配32字节的空间。进入 factor 函数后，将返回地址和帧指针保存到栈上，增大栈空间。将参数 10 保存到栈帧中，并加载到寄存器。进入循环后，递归调用 factor 函数，每次调用

都分配新的栈帧，增大栈空间。递归结束后，计算阶乘并返回结果。清理栈并恢复返回地址和帧指针，减小栈空间。

最大内存占用发生在调用递归函数时，每次递归调用都会在栈上分配新的栈帧，直到递归结束。因此，最大内存占用为32字节（栈帧大小）乘以递归调用的次数，即栈帧的最大深度。在本例中，调用 `factor(10)` 时，递归深度为10，因此最大内存占用为320字节，发生在递归调用达到最大深度时。

Problem 4

假设栈的大小为 4KB，请问`factor(n)`的参数`n`最大是多少？

假设栈的大小为4KB，即4096字节。在调用 `factor(n)` 函数时，栈的最大内存占用为栈帧大小乘以递归调用的最大深度。

已知栈帧大小为32字节（根据代码中的分配），因此递归调用的最大深度可以计算如下：

$$\text{最大深度} = \frac{\text{栈大小}}{\text{栈帧大小}} = \frac{4096 \text{ 字节}}{32 \text{ 字节/帧}} = 128$$

因此，最大的参数`n`应该小于等于128，以确保栈的内存占用不超过4KB。

Problem 5

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

factor_opt.s: 采用了更加简洁的汇编代码，减少了不必要的指令和加载/保存操作。使用了 `mul` 指令一次性计算阶乘的乘积，而不是递归调用。没有显式保存和恢复帧指针 (`s0`) 和返回地址 (`ra`)，而是直接使用寄存器。没有额外的循环标签，而是使用条件分支指令 `bne` 来实现循环。整体上更加紧凑和高效。

factor_plain.s: 采用了较为简单直观的汇编代码，容易理解和调试。使用了递归调用的方式来计算阶乘，每次递归调用都会在栈上分配新的栈帧。显式保存和恢复帧指针 (`s0`) 和返回地址 (`ra`)，以及参数的加载和保存。使用了循环标签来实现循环，逻辑较为明确。

Problem 6

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。

factor_opt.s: 由于 factor_opt.s 中采用了迭代计算阶乘，而不是递归调用，因此不需要在栈上分配额外的栈帧来保存递归调用的状态。减少了函数调用和返回时需要保存和恢复的寄存器，例如帧指针 (s0) 和返回地址 (ra)。没有额外的递归深度，因此栈上不需要维护多个递归调用的状态，节省了栈空间。

factor_plain.s: factor_plain.s 中使用了递归调用的方式来计算阶乘，每次递归调用都会在栈上分配新的栈帧。每次递归调用都需要保存和恢复一定数量的寄存器值，包括返回地址 (ra) 和帧指针 (s0)，以及参数的加载和保存。递归调用导致栈的深度增加，从而增加了栈的内存占用。

Problem 7

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。

尾递归优化是一种编译器优化技术，用于优化尾递归函数的执行，以减少函数调用的开销和栈空间的使用。尾递归是指在函数的最后一个操作是递归调用自身的情况。

在生成 factor_opt.s 这样的代码时，编译器可能会进行尾递归优化，具体优化包括将递归调用转换为迭代调用，以减少栈帧的分配和释放，从而减少内存消耗。

尾递归优化的原理是将递归函数转换为迭代函数。这通常通过使用循环来替代递归调用来实现。在每次递归调用时，更新函数参数的值，并将控制权传递给自身，直到满足递归结束条件。这样，就避免了在每次递归调用时分配新的栈帧，从而减少了栈空间的使用。

编译器在进行尾递归优化时，通常会检查函数的调用位置和参数，并确定是否满足尾递归的条件。只有当函数的最后一个操作是对自身的递归调用，并且该调用是函数的返回值时，才能进行尾递归优化。

在生成 factor_opt.s 时，编译器可能会检测到 factor 函数的最后一个操作是对自身的递归调用，并且该调用是函数的返回值，因此可以进行尾递归优化。优化后的代码将使用循环来计算阶乘，避免了递归调用时栈帧的

分配和释放，从而减少了内存消耗。

4 理解 switch 语句产生的跳转表

Problem 1

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

- 1.首先，将输入的参数减去一个偏移量（20），得到一个调整后的值，存储在寄存器 a5 中。
- 2.然后，比较这个调整后的值和一个预设的边界值（6）：
 - 如果调整后的值大于6，则跳转到标签 .L8，执行默认的情况。
 - 如果调整后的值小于等于6，则继续执行后面的代码。
- 3.如果调整后的值小于等于6，将其乘以4，这是因为后面将要对应6个情况，每个情况用一个字（32位）表示，所以每个情况之间的间隔是4字节。
- 4.根据调整后的值计算出对应情况的地址：
 - 加载一个表示 .L4 标签的地址到寄存器 a4 中。
 - 将调整后的值乘以4，得到该情况在表中的偏移量。
 - 将该偏移量加到 .L4 标签地址上，得到对应情况的地址。
- 5.从这个地址中读取一个字（32位），这个字是一个跳转地址，即跳转到对应情况的代码块。
- 6.执行对应情况的代码块。
- 7.在每个情况的代码块中，执行完相应的指令后，会遇到一个 ret 指令，返回到函数的调用者。

Problem 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。

用跳转表实现 switch 的优势：

效率高：跳转表使用了直接跳转，通过索引来访问目标地址，因此在具有多个分支的情况下，可以更快地确定执行的路径，时间复杂度为 $O(1)$ 。

简洁明了：跳转表的实现通常比 if-else 结构更加简洁，易于理解和维护。

可读性好：跳转表将不同情况的处理逻辑封装在一个表中，易于理解和修改。

用跳转表实现 switch 的劣势：

适用性有限：跳转表通常要求 case 值是连续的整数或枚举类型，并且在一定的范围内。对于不连续或者非整数类型的情况，跳转表的使用就不太合适。

内存占用较高：跳转表需要额外的内存空间来存储跳转表，对于分支较多的情况，跳转表可能会占用较多的内存。

用 if-else 实现 switch 的优势：

适用性广泛：if-else 结构适用于任意的条件判断，不受条件的类型和取值范围的限制，适用性更广。

内存占用低：相比于跳转表，if-else 结构不需要额外的内存空间来存储跳转表，因此在内存占用方面可能更加节省。

用 if-else 实现 switch 的劣势：

效率低：if-else 结构需要逐个判断条件，并根据条件的真假来确定执行的路径，时间复杂度为 $O(n)$ ，其中 n 是分支的个数。因此在分支较多的情况下，效率可能较低。

可读性差：if-else 结构嵌套较深或者分支较多时，代码可能变得复杂，可读性较差。

跳转表适用情况：

当 switch 语句中的 case 值是连续的整数或者枚举类型，并且分支较多时，可以考虑使用跳转表。

当对性能要求较高，需要快速确定执行路径时，跳转表是一个较好的选择。

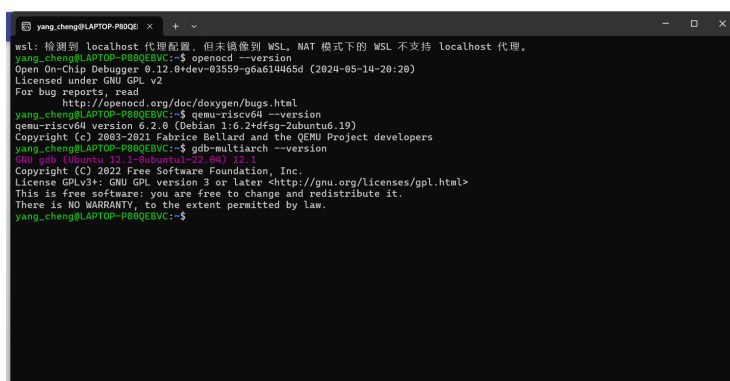
if-else 适用情况：

当 switch 语句中的 case 值不连续或者为非整数类型时，应该使用 if-else 结构。

当 switch 语句中的分支较少，并且不需要过多的内存占用时，if-else 结构更加适合。

Lab 5-2: RISC-V 汇编程序 调试

1 实验环境配置



```
yang_cheng@LAPTOP-P8BQEBVC:~$ wsl: 检测到 localhost 代理配置，但未检测到 WSL，NAT 模式下的 WSL 不支持 localhost 代理。
yang_cheng@LAPTOP-P8BQEBVC:~$ openocd --version
Open On-Chip Debugger 0.12.0+dev-03559-g6a614465d (2024-05-14-20:20)
Licensed under GNU GPL v2
For bug reports, read
    http://openocd.org/doc/doxygen/bugs.html
yang_cheng@LAPTOP-P8BQEBVC:~$ qemu-riscv64 --version
qemu-riscv64 version 6.2.0 (Debian 1:6.2+dfsg-2ubuntu6.19)
Copyright (c) 2003-2021 Fabrice Bellard and the QEMU Project developers
yang_cheng@LAPTOP-P8BQEBVC:~$ gdb-multiarch --version
GNU gdb (Debian 12.1-ubuntu12.08) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
yang_cheng@LAPTOP-P8BQEBVC:~$
```

2 尝试通过调试破解数据

Phase_1

为了使 ret 的值为 0，我们需要一个输入字符串，其中至少包含一个字符 c，使得 char2num(c) 的值与 data 相等。

由图可知，我们知道了 data 的值是 13。根据 char2num 函数的定义，字符 'd' 的 ASCII 值是 100，所以 char2num('d') 应该返回 13。

因此，如果我们构造一个字符串，其中包含字符 'd'，作为 phase_1 函

```

int phase_1(const char* str){
    int data = char2num(name_buffer[6]);
    int iter = 20;
    while(iter--){
        data = (phase_box[data]*phase_box[(data+3)&0xf]*phase_box[phase_box[data]])&0xf;
    }
    int ret = 1;
    while(*str){
        if(char2num(*str) == data){
            ret = 0;
            break;
        }
        str++;
    }
    return ret;
}

```

```

yang_cheng@LAPTOP-P80GEF: X  yang_cheng@LAPTOP-P80GEF: X  +
(gdb) n
20      int iter = 20;
(gdb) n
21      while(iter--){
(gdb) n
22          data = (phase_box[data]*phase_box[(data+3)&0xf]*phase_box[phase_box[data]])&0xf;
(gdb) n
21      while(iter--){
(gdb) b 23
Breakpoint 3 at 0x400000092c: file challenge.c, line 24.
(gdb) c
Continuing.

Breakpoint 3, phase_1 (str=0x40000038c0 <input_buffer> "d") at challenge.c:24
24      int ret = 1;
(gdb) n
25      while(*str){
(gdb)
26          if(char2num(*str) == data){
(gdb) n
27              ret = 0;
(gdb)
28              break;
(gdb)
32      return ret;
(gdb) info locals
data = 13
iter = -1
ret = 0
(gdb)

```

数的参数，那么 ret 的值将会被设置为 0。这样就可以成功破解 phase_1 函数。

Phase_2

为了使 ret 的值为 0，我们需要找到一个长度至少为 3 的字符串，使得经过这个函数处理后，data 数组中的值与输入字符串的前三个字符对应相等。

首先，我们知道 data[0] 的值是 7，根据 char2num 函数的定义，char2num('7') 返回的值是 7。所以 str[0] 应该是 '7'。

接着，我们知道起初 data[1] 的值是 9，运行完 data[1] = (data[1] + char2num(str[0])) & 0xf 之后，data[1] 的值是 0，所以我们需要找到一个字符 c，使得 char2num(c) 的值等于 0。根据 char2num 函数的定义，char2num('0') 返回的值是 0。所以 str[1] 应该是 '0'。

同理可知，data[2] 的值最终是 7。根据 char2num 函数的定义，char2num('7')

```

int phase_2(const char* str){
    if(!str[0] && str[1] && str[2]){
        return 1;
    }
    int data[3] = {char2num(name_buffer[7]),char2num(name_buffer[8]),char2num(name_buffer[9])};
    int iter = 100;
    while(iter--){
        data[0] = (phase_box[phase_box[data[0]]]^phase_box[data[1]]^phase_box[data[2]])&0xf;
        data[1] = (phase_box[data[0]]^phase_box[phase_box[data[1]]]^phase_box[data[2]])&0xf;
        data[2] = (phase_box[data[0]]^phase_box[data[1]]^phase_box[phase_box[data[2]]])&0xf;
    }
    data[1] = (data[1] + char2num(str[0])) & 0xf;
    data[2] = (data[2] + char2num(str[1])) & 0xf;

    int ret = 1;
    if(char2num(str[0])==data[0] && char2num(str[1])==data[1] && char2num(str[2])==data[2]){
        ret = 0;
    }
    return ret;
}

```

返回的值是。所以 str[2] 应该是 '7'。

因此，如果我们构造一个字符串，其中前三个字符分别为为 '7', '0', '7', 作为 phase_2 函数的参数，那么 ret 的值将会被设置为 0。这样就可以成功破解 phase_2 函数。

```

yang_cheng@LAPTOP-PI0QIE: x  yang_cheng@LAPTOP-PI0QIE: x  +  v
Breakpoint 2, phase_2 (str=0x4000003bc0 <input_buffer: "707">) at challenge.c:35
35 int phase_2(const char* str){
(gdb) n
36 if(!str[0] && str[1] && str[2]){
(gdb)
37 return 1;
(gdb)
38 int data[3] = {char2num(name_buffer[7]), char2num(name_buffer[8]), char2num(name_buffer[9])};
(gdb)
39 int iter = 100;
(gdb)
40 while(iter--){
(gdb)
41 data[0] = (phase_box[phase_box[data[0]]]^phase_box[data[1]]^phase_box[data[2]])&0xf;
(gdb)
42 data[1] = (phase_box[data[0]]^phase_box[phase_box[data[1]]]^phase_box[data[2]])&0xf;
(gdb)
43 data[2] = (phase_box[data[0]]^phase_box[data[1]]^phase_box[phase_box[data[2]]])&0xf;
(gdb)
44 while(iter--){
(gdb)
45
(gdb) b 45
Breakpoint 5 at 0x4000000b2a: file challenge.c, line 46.
(gdb) c
Continuing.
Breakpoint 5, phase_2 (str=0x4000003bc0 <input_buffer: "707">) at challenge.c:46
46 data[1] = (data[1] + char2num(str[0])) & 0xf;
(gdb) info locals
data = {7, 9, 7}
iter = -1
ret = 0
(gdb)

```

Phase_3

```

int phase_3(const char* str){
    int sum = char2num(name_buffer[6])*char2num(name_buffer[7])*char2num(name_buffer[8])*char2num(name_buffer[9]);
    int ret = 1;
    asm volatile(
        "mov %0, %1\n"
        "mov %1, %2\n"
        "j phase_3_1\n"
        "phase_3_1:\n"
        "xor %0, %1\n"
        "addi %1, %1, 1\n"
        "phase_3_2:\n"
        "lbu %2, 0(%1)\n"
        "bne %2, zero, phase_3_1\n"
        "bne %0, zero, phase_3_3\n"
        "mov %1, zero\n"
        "phase_3_3:\n"
        "nop\n"
        :sum: "+r"(sum),ret: "+r"(ret)
        :str: "r"(str)
        :memory,"%0","%1","%2"
    );
    return ret;
}

```

```

ra      0x4000000c56      0x4000000c56 <phase_3+114>
fp      0x0001800040      0x0001800040
gp      0x0000003800      0x0000003800
tp      0x00019620e0      0x00019620e0
t0      0x35      53
t1      0x40000038c0      274877919424
t2      0x35      53
fp      0x0001800000      0x0001800000
t1      0x7      7
a0      0x5      5
a1      0xa      10
a2      0x0      0
a3      0x0      0
a4      0x35      53
a5      0x40000038c0      274877919424
a6      0x6c265607207475      7935454842726102133
a7      0x3f      63
a8      0x0      0
a9      0x0      0
a10     0x0      0
a11     0x0      0
a12     0x0      0
a13     0x0      0
a14     0x0      0
a15     0x0      0
a16     0x0      0
a17     0x0      0
a18     0x0      0
a19     0x0      0
--Type <RET> for more, q to quit, c to continue without paging--

```

首先，它计算了 sum，这是对 name_buffer[6] 到 name_buffer[9] 对应字符的值进行异或操作的结果。

然后，它进入了一个循环，遍历输入字符串。在循环的每一次迭代中，它对 t0 执行一次异或操作，并且将指针向后移动一个字符。

如果遇到了字符串的结束（即遇到了空字符 \0），则跳出循环。

最后，它检查 t0 的值是否为零，如果是，则将 ret 设为零。

与之前的函数一样，为了使 ret 的值为 0，我们需要找到一个合适的输入字符串。

Bonus 1

spike 调试部分给出调试过程的关键截图，并且给出 printm 到第一个字符输出的函数调用链、输出字符的关键函数、输出字符那一句汇编。

从 printm 到第一个字符输出的函数调用链：printm → vprintm → vsnprintf → vprintm → uart16550_putchar

输出字符的关键函数：uart16550_putchar

输出字符那一句汇编：sb a0,0(a3)

```
yang_cheng@LAPTOP-P80QE: x + v
wsl: 检测到 localhost 代理配置，但未链接到 WSL。NAT 模式下的 WSL 不支持 localhost 代理。
yang_cheng@LAPTOP-P80QE$ cd sys1-sp24/src/lab5-2
yang_cheng@LAPTOP-P80QE$ ./sys1-sp24/src/lab5-2 spike -H --rbb-port=9824 ../../repo/riscv-pk/build/bbl
Listening for remote bitbang connection on port 9824.
b
```

```
yang_cheng@LAPTOP-P80QE: x + v
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ../../repo/riscv-pk/build/bbl...
(No debugging symbols found in ../../repo/riscv-pk/build/bbl)
(gdb) target remote:3333
Remote debugging using :3333
0x0000000000000000 in ?? ()
(gdb) b printe
Breakpoint 1 at 0x00001c82
(gdb) c
Continuing.

Breakpoint 1, 0x0000000000000000:00001c82 in printe ()
(gdb) b uart
uart16550_done      uart16550_prop      uart_getchar      uart_litex_open    uart_open
uart16550_getchar   uart16550_putchar  uart_litex_done    uart_litex_prop    uart_prop
(gdb) b uart16550_putchar
uart_litex_getchar  uart_litex_putchar  uart_putchar
Breakpoint 2 at 0x0000285a
(gdb)
```

```
yang_cheng@LAPTOP-P80QE: x + v
0x00002852 <uart16550_putchar>      auipc      a5,0xf
0x00002856 <uart16550_putchar+4>      lw         a5,2014(a5)
0x0000285a <uart16550_putchar+8>      li         a4,5
0x0000285c <uart16550_putchar+10>     auipc      a3,0xf
0x00002858 <uart16550_putchar+14>      ld         a3,2012(a3)
0x00002859 <uart16550_putchar+15>     sllw      a4,a4,a3
0x00002859 <uart16550_putchar+22>     add        a4,a4,a3
0x00002859 <uart16550_putchar+24>     lbu        a5,0(a4)
0x00002859 <uart16550_putchar+28>     andi       a5,a5,32
0x00002852 <uart16550_putchar+32>     beqz      a5,0x00002859 <uart16550_putchar+24>
> 0x000028a4 <uart16550_putchar+34>     sb         a0,0(a3)
0x000028a3 <uart16550_putchar+38>     ret
0x000028a4 <uart16550_putchar>      auipc      a3,0xf
0x000028a6 <uart16550_putchar+4>      lw         a3,1974(a3)
0x000028b2 <uart16550_putchar+8>      li         a5,5
0x000028b4 <uart16550_putchar+10>     auipc      a4,0xf
0x000028b8 <uart16550_putchar+14>     ld         a4,1972(a4)

remote Remote target In: uart16550_putchar      L?? PC: 0x000028a4
(gdb) s
0x000000000000285c in uart16550_putchar ()
0x0000000000002858 in uart16550_putchar ()
0x0000000000002859 in uart16550_putchar ()
0x0000000000002859 in uart16550_putchar ()
0x0000000000002859 in uart16550_putchar ()
0x0000000000002852 in uart16550_putchar ()
0x00000000000028a4 in uart16550_putchar ()
(gdb)
```