LAB5

lab5-1

理解简单 RISC-V 程序 10%

请阅读 `src/lab5-1/acc_plain.s` 回答以下问题：

```
    .file   "acc.c"
    .option pic
    .text
    .align  1
    .globl  acc
    .type   acc, @function
acc:
    addi    sp,sp,-48
    sd  s0,40(sp)
    addi    s0,sp,48
    sd  a0,-40(s0)
    sd  a1,-48(s0)
    sd  zero,-32(s0)
    ld  a5,-40(s0)
    sd  a5,-24(s0)
    j   .L2
.L3:
    ld  a4,-32(s0)
```

```
    ld  a5,-24(s0)
    add a5,a4,a5
    sd  a5,-32(s0)
    ld  a5,-24(s0)
    addi    a5,a5,1
    sd  a5,-24(s0)
.L2:
    ld  a4,-24(s0)
    ld  a5,-48(s0)
    ble a4,a5,.L3
    ld  a5,-32(s0)
    mv  a0,a5
    ld  s0,40(sp)
    addi    sp,sp,48
    jr  ra
    .size   acc, .-acc
    .ident  "GCC: (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0"
    .section    .note.GNU-stack,"",@progbits
```

`acc` 是如何获得函数参数的，又是如何返回函数返回值的？2%

前两个参数a,b分别通过 `a0` 和 `a1` 寄存器传入。
返回值通过`a0`寄存器来返回。

`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是帧指针，在栈指针在这个程序中起到了指示栈顶的作用。
- 在执行 `sd s0, 40(sp)` 中是将S0的数据暂时存到40(sp)中，防止acc的函数将S0的数值改动
- 在执行 `addi s0, sp, 48` 的时候是初始化当前的帧指针，使得S0的帧指针更新为新的指针

`acc` 函数的栈帧 (stack frame) 的大小是多少？1%

- 栈帧的大小是48字节

`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%

- S0旧的s0是存放在0（sp)中
- a0是存放在8(sp)中
- a1是存放在0(sp)中
- 循环控制变量i是存放在24(sp)中

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%

- 初始化：在进入循环前，将零值写入 `-32(s0)`（初始化 `res = 0`），将参数 `a` 的值写入 `-24(s0)`。随后执行 `j .L2` 无条件跳转到条件检查部分。
- 条件检查 (`.L2`)：
  - 取出循环变量 `i`（`-24(s0)`）放入 `a4`，取出结束边界参数 `b`（`-48(s0)`）放入 `a5`。
  - 执行 `ble a4, a5, .L3`（意为：如果 $i \leq b$，则跳转到 `.L3` 执行循环体)。
  - 如果 $i > b$，则不跳转，直接向下执行，退出循环。
- 循环体与自增 (`.L3`)：
  - **累加**：读取当前的 `res` 和 `i` 进行加法，结果写回 `res`（`-32(s0)`)。
  - **自增**：读取 `i` 的值，执行 `addi a5, a5, 1` 将其加 1，然后写回 `i`（`-24(s0)`)。
  - 执行完 `.L3` 的代码后，程序会自然顺序向下再次进入 `.L2` 进行条件检查，从而形成循环。

`src/lab5-1/acc_opt.s` 也是 `src/lab5-1/acc.c` 对应的一种汇编程序，它由另一种命令生成：

| `riscv64-linux-gnu-gcc -S acc.c -O2 -o acc_opt.s` |

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 是指的是optimization level 0 无优化：编译器不进行代码转换，会频繁地访问内存。每条 C 语言语句通常都能对应到几条明确的汇编指令。

-
    - O2 指的是optimization level 2高级优化
      特点：
- 寄存器分配优化：尽可能将变量留在寄存器中，避免昂贵的内存（栈）访问。
- 控制流优化：重排代码块，减少分支跳转次数。
- 指令调度：调整指令顺序以利用处理器的流水线性能。
- 代码简化：消除死代码或进行常量折叠。

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

- acc_plain.s 的优点：便于调试
- acc_plains的缺点：性能极低，代码冗余
- acc_opts.s的优点 ：性能极高，栈的开销小，代码十分紧凑
- acc_opts.s的缺点：调试困难，编译略慢

理解递归汇编程序 15%

```
        .text
        .globl  factor
factor:
        addi    sp,sp,-32
        sd      ra,24(sp)
        sd      s0,16(sp)
        addi    s0,sp,32
        sd      a0,-24(s0)
        ld      a5,-24(s0)
        bne     a5,zero,.L2
        li      a5,1
        j       .L3
.L2:
        ld      a5,-24(s0)
        addi    a5,a5,-1
        mv      a0,a5
        call    factor
        mv      a4,a0
        ld      a5,-24(s0)
        mul     a5,a4,a5
.L3:
        mv      a0,a5
        ld      ra,24(sp)
        ld      s0,16(sp)
        addi    sp,sp,32
        jr      ra
```

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%

- 因为`factor`函数是一个递归函数，`call factor`指令会调用自身，如果没有`ra`函数，则会覆盖返回地址寄存器`ra`为了能够返回原来的`factor`函数的地址，所以要用`ra`。

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

- 在`riscv`语言中函数中传入的整数通过a0传入函数。`a5`中存的是`n-1`的值，使得`call factor`等价于`factor(n-1)`。

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%

- 调用时，栈指针sp减少了32字节，用来分配栈帧，经过调用之后，每一次函数的调用都会使栈上分配一个新的32字节的栈帧，最大的占用内存大小为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%
   $n_{max} \leq \frac{4096}{32}$
   解得：$n \leq 127$

`src/lab5-1/factor_opt.s` 也是 `src/lab5-1/factor.c` 对应的一种汇编程序，它由另一种命令生成：

```
riscv64-linux-gnu-gcc -S factor.c -O2 -o factor_opt.s
```

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%
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%
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%

理解 switch 语句产生的跳转表 5%¶

在 `src/lab5-1/switch.c` 中实现了一个简单的基于 switch 语句的函数`switch_eq`：

```
int switch_eg(long long x, long long y) {
long long result = y;
switch (x) {
case 20:
result = result - 5;
```

```
 case 21:
 result = result + 19;
 break;
  case 22:
  result += 11;
  break;
  case 24:
  case 26:
   result -= 20;
   break;
   default:
   result = 0;     }
   return result; }
```

`src/lab5-1/switch.s` 为该函数的汇编版本，由以下命令生成：

```
riscv64-linux-gnu-gcc -S switch.c -O2 -o switch.s
```

```
    .text
    .globl  switch_eg
switch_eg:
    addi    a5,a0,-20
    li  a4,6
    bgtu    a5,a4,.L8
    lla a4,.L4
    slli    a5,a5,2
    add a5,a5,a4
    lw  a5,0(a5)
    add a5,a5,a4
    jr  a5
    .section    .rodata
    .align  2
    .align  2
.L4:
    .word   .L7-.L4
    .word   .L6-.L4
    .word   .L5-.L4
    .word   .L8-.L4
    .word   .L3-.L4
    .word   .L8-.L4
    .word   .L3-.L4
    .text
.L3:
    addiw   a0,a1,-20
```

```
    ret
.L7:
    addi    a1,a1,-5
.L6:
    addiw   a0,a1,19
    ret
.L5:
    addiw   a0,a1,11
    ret
.L8:
    li  a0,0
    ret
```

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%
   - 先是进行一个范围的检查 `addi a5,a0,-20`， `bgtu a5,a4,.L8` 如果说大于6则返回0
   - 再使进行一个跳转表的遍历，即switch。先加载出跳转表的基地址 `a4` ,然后再计算出偏移量 `slli a5,a5,2` ,再相加得到目标的地址，最后再通过 `jr a5` 跳转到对应的case。
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%
   - 优势：执行的效率高，跳转时间基本上恒定在O(1)，而且代码简洁。
   - 劣势，占用的内存大，需要大量的空间来储存跳转表，使用的范围有限，只能存储离散的取值。

bubble_sort.s 和 fibonacci.s 函数解释

---

1. bubble_sort.s

功能

对 long long 数组进行**冒泡排序**（从小到大）。

C 代码对照

```c
void bubble_sort(long long *arr, long long len) {
    for (long long i = 0; i < len - 1; i++) {
        for (long long j = 0; j < len - 1 - i; j++) {
            if (arr[j] > arr[j+1]) {
                long long temp = arr[j];
                arr[j] = arr[j+1];
```

```
                arr[j+1] = temp;
            }
        }
    }
}
```

汇编逐段解释

函数头与保存寄存器

```
bubble_sort:
    addi sp,sp,-32      # sp 往下挪 32 字节，空出 4 个格子
    sd ra,24(sp)        # 保存 ra（返回地址）
    sd s0,16(sp)        # 保存 s0
    sd s1,8(sp)         # 保存 s1
    sd s2,0(sp)         # 保存 s2
```

> 因为要用 **s0/s1/s2** 来存重要数据，这些寄存器的原始值必须先存到栈上，函数结束再恢复。

把参数转移到安全的地方

```
    mv s0,a0            # s0 = arr（数组首地址）
    mv s1,a1            # s1 = len（数组长度）
    li s2,0             # s2 = i = 0（外层循环计数器）
```

> a0 和 a1 是传进来的参数，但后面计算时要反复用到它们，转移到 s 寄存器更安全。

外层循环

```
.L1:
    addi t0,s1,-1       # t0 = len - 1
    bge s2,t0,.L_done   # 如果 i >= len-1，排序完成
    li t1,0             # j = 0（每次外层循环，内层 j 都从 0 开始）
```

> 外层循环只需要跑 **len-1 轮**，每轮把当前最大的数冒泡到最后。

内层循环：取 arr[j] 和 arr[j+1]

```
.L2:
    sub t2,s1,s2        # t2 = len - i
    addi t2,t2,-1       # t2 = len - i - 1
    bge t1,t2,.L1_next  # 如果 j >= len-1-i，内层结束
```

```
    slli t3,t1,3        # t3 = j * 8（long long = 8 字节）
    add t3,s0,t3        # t3 = &arr[j]
    ld t4,0(t3)         # t4 = arr[j]
    ld t5,8(t3)         # t5 = arr[j+1]（+8 就是下一个元素）
```

> 内层只需要跑到 **len-1-i**，因为最后 i 个元素已经是排好序的了。
> j * 8 是因为每个 long long 占 8 字节。

比较并决定是否交换

```
    ble t4,t5,.L3       # 如果 arr[j] <= arr[j+1]，不交换
    sd t5,0(t3)         # arr[j]   = arr[j+1]（把大的往后放）
    sd t4,8(t3)         # arr[j+1] = arr[j]（把小的往前放）
.L3:
    addi t1,t1,1        # j++
    j .L2
```

> 如果前面的数 <= 后面的数，说明顺序对了，跳过交换。
> 否则交换两个数，把大的往后冒。

函数结尾

```
.L_done:
    ld ra,24(sp)        # 恢复 ra
    ld s0,16(sp)        # 恢复 s0
    ld s1,8(sp)         # 恢复 s1
    ld s2,0(sp)         # 恢复 s2
    addi sp,sp,32       # 归还栈空间
    ret                 # 返回
```

寄存器用途总结

寄存器	存什么
s0	数组首地址 arr
s1	数组长度 len
s2	外层循环变量 i
t0	len-1（外层循环终值）
t1	内层循环变量 j
t2	len-1-i（内层循环终值）
t3	&arr[j]（当前元素的地址）

寄存器	存什么
t4	arr[j]（当前元素的值）
t5	arr[j+1]（下一个元素的值）

---

2. fibonacci.s（递归版）

功能

计算第 n 个斐波那契数：

- fib(0) = 1
- fib(1) = 1
- fib(n) = fib(n-1) + fib(n-2)

C 代码对照

```
long long fibonacci(long long n) {
    if (n == 0) return 1;          // base case
    if (n == 1) return 1;          // base case

    long long fib_n1 = fibonacci(n - 1);  // 递归算 fib(n-1)
    long long fib_n2 = fibonacci(n - 2);  // 递归算 fib(n-2)

    return fib_n1 + fib_n2;               // 加起来的 fib(n)
}
```

汇编逐段解释

函数头与保存寄存器

```
fibonacci:
    addi sp,sp,-32          # 栈上申请 32 字节
    sd ra,24(sp)            # 保存 ra（返回地址）
    sd s0,16(sp)            # 保存 s0（用来存 n）
    sd s1,8(sp)             # 保存 s1（用来存 fib(n-1) 的结果）
    sd s2,0(sp)             # 保存 s2（用来存 fib(n-2) 的结果）
```

> 因为要**递归调用自己**，下一层的 `call` 会把 ra 覆盖掉。
> 同时 a0（参数）也会被覆盖，所以先把 n 转到 s0。
> s1 和 s2 用来存两次递归的结果。

参数转移和 base case 判断

```
    mv s0,a0                # s0 = n（把 n 保存起来）

    li t0,0
    beq s0,t0,.L_base       # 如果 n == 0，直接返回 1
    li t0,1
    beq s0,t0,.L_base       # 如果 n == 1，直接返回 1
```

> beq 只能比较两个**寄存器**，所以先把 0 和 1 加载到 t0 里，再和 s0 比较。

递归调用 fib(n-1)

```
    addi a0,s0,-1           # a0 = n - 1（准备参数）
    call fibonacci          # 调用自己，算 fib(n-1)
    mv s1,a0                # 把结果存到 s1（免得被下一次 call 覆盖）
```

> call 之后，a0 里就是 fib(n-1) 的结果了。
> 马上还要调 fib(n-2)，那一次 call 会覆盖 a0，所以赶紧把结果转移到 s1。

递归调用 fib(n-2)

```
    addi a0,s0,-2           # a0 = n - 2（准备参数）
    call fibonacci          # 调用自己，算 fib(n-2)
    mv s2,a0                # 把结果存到 s2
```

相加得到最终结果

```
    add a0,s1,s2            # a0 = fib(n-1) + fib(n-2) = fib(n)
    j .L_return             # 跳到统一收尾
```

> s1 和 s2 都到手了，加起来就是最终答案。

base case

```
.L_base:
    li a0,1                 # fib(0)=1, fib(1)=1
```

> n=0 或 n=1 时直接返回 1。

收尾

```
.L_return:
    ld ra,24(sp)          # 恢复返回地址
    ld s0,16(sp)          # 恢复 s0
    ld s1,8(sp)           # 恢复 s1
    ld s2,0(sp)           # 恢复 s2
    addi sp,sp,32         # 归还栈空间
    ret                   # 返回
```

> 不管是从 base case 还是从递归计算过来，都要走这里恢复寄存器。

lab5-2

qemu的方法破解

phase1

- 只要是字符串中含8即可

phase2

- 要满足是两个数字`a,b`
- 还要满足 4^a^b=0

phase3

- 输入`5e53fd7d`即可

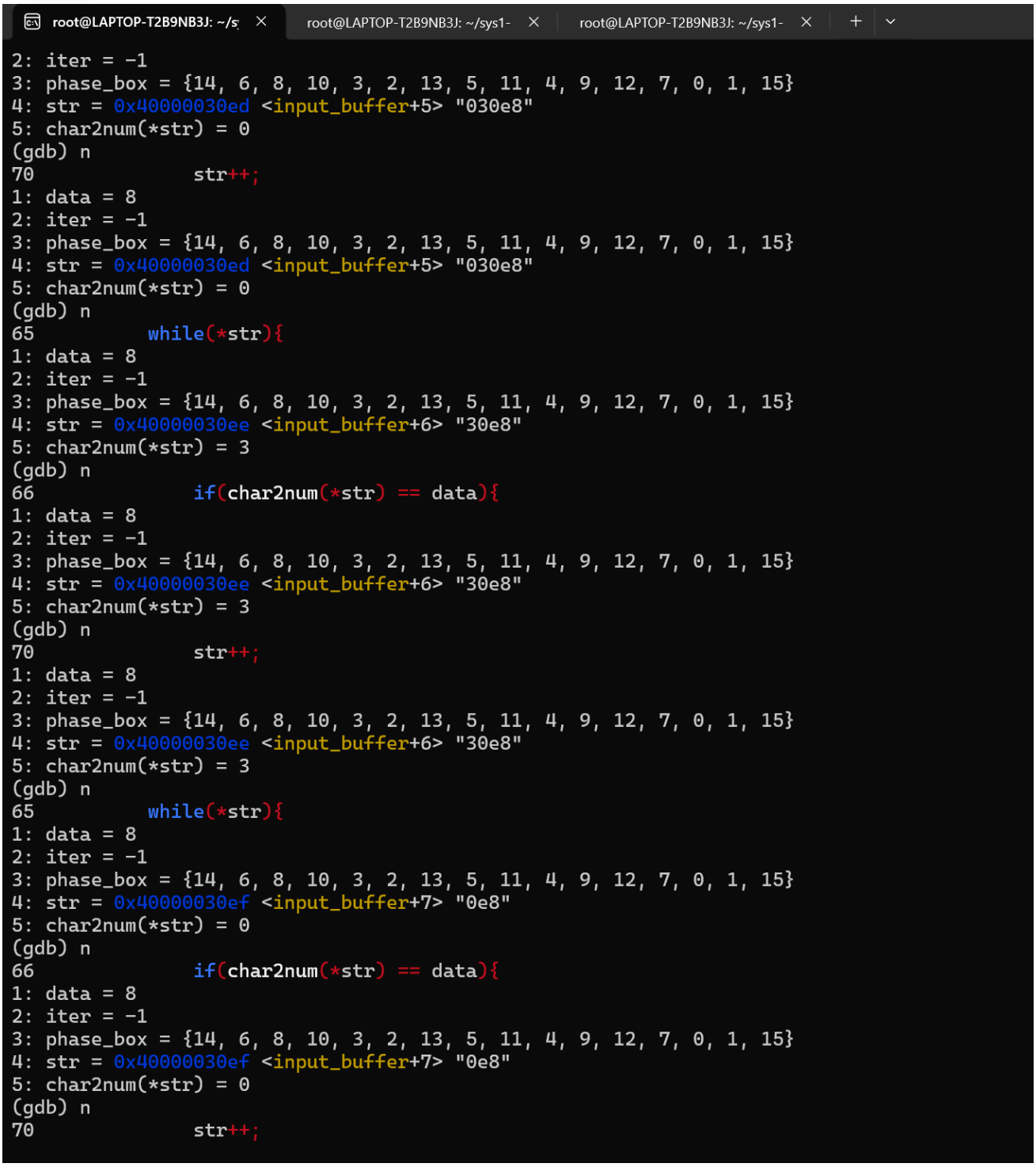
```
root@LAPTOP-T2B9NB3J:~/sys1-sp26/src/lab5-2# qemu-riscv64 challenge
please input your student ID:3250104303
Welcome to my fiendish little bomb. You have 3 phases with
8888888888888888
Phase 1 defused. How about the next one?
40
Phase 2 defused. How about the next one?
5e53fb7d
Phase 3 defused.
root@LAPTOP-T2B9NB3J:~/sys1-sp26/src/lab5-2#
```

下面展示推导的过程

phase1

- 先是通过`b phase1`跳转到phase1的位置设置断点，再`c`跳转到断点。通过gdb的调试发现在代码的66行有一句`if(char2num(*str)==data)`的判断语句，而且通过`list data`的语句发

现 `data` 的值为8，分析可知是字符串中只要存在8就可以通过



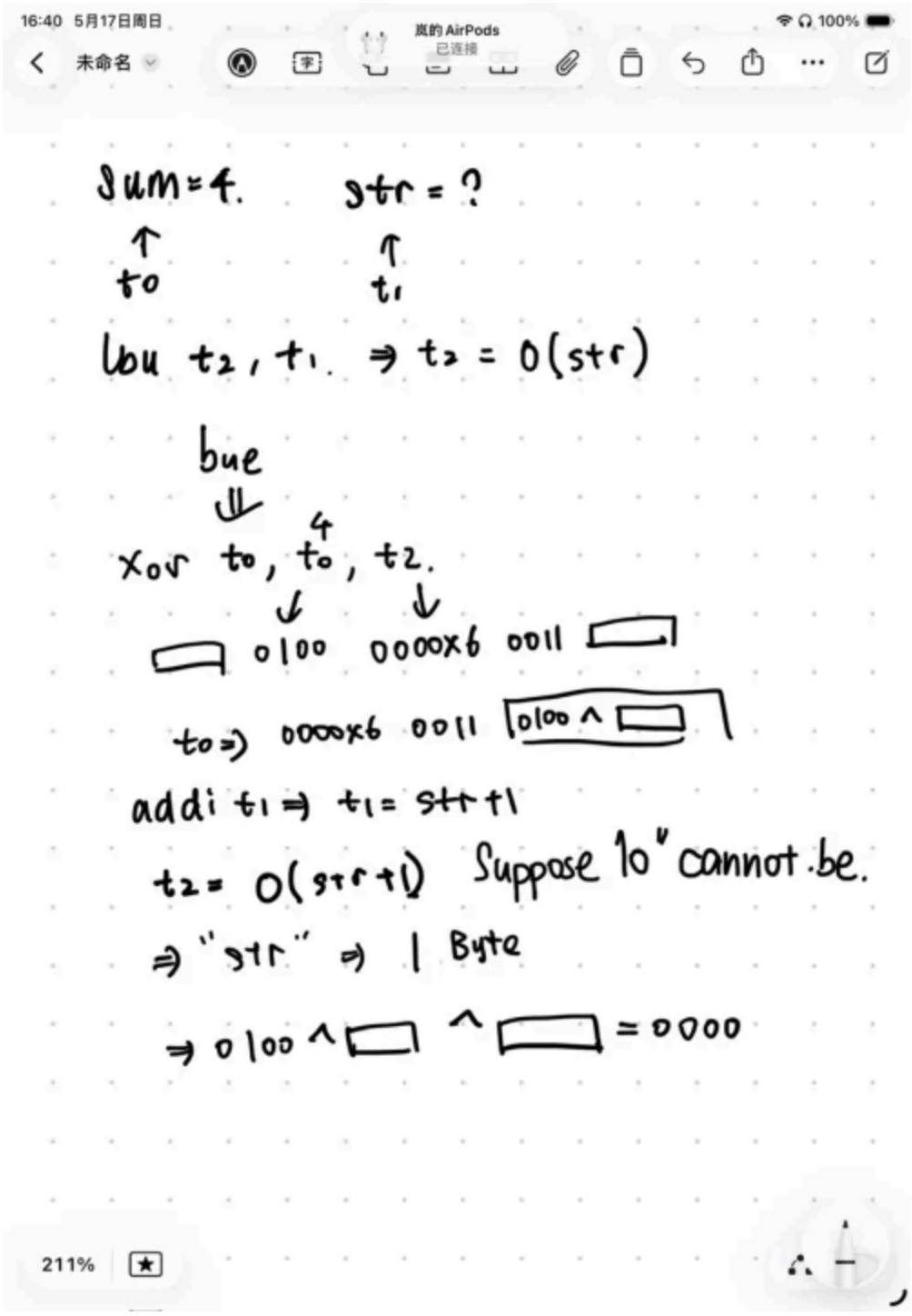
phase2

- 我先通过 `b phase2` `c` 来定位到phase2的位置
- 然后通过尝试发现 `list 75,100` 能显示全部的phase2代码

- 通过 `list sum` 可以调出来sum的值是4

```
(gdb) list 75,100
75      int phase_2(const char* str){
76          int sum = char2num(name_buffer[6])^char2num(name_buffer[7])^char2num(name_buffer[8])^char2num(name_buffer[9]);
77          int ret = 1;
78          asm volatile(
79              "mv t0, %[sum]\n"
80              "mv t1, %[str]\n"
81              "j phase_2_L1\n"
82              "phase_2_L2:\n"
83              "xor t0, t0, t2\n"
84              "addi t1, t1, 1\n"
85              "phase_2_L1:\n"
86              "lbu t2, 0(t1)\n"
87              "bne t2, zero, phase_2_L2\n"
88              "bne t0, zero, phase_2_L3\n"
89              "mv %[ret], zero\n"
90              "phase_2_L3:\n"
91              "nop\n"
92              :[ret] "=r" (ret)
93              :[sum] "r" (sum),[str] "r" (str)
94              :"memory","t0","t1","t2"
95          );
96          return ret;
97      }
98
99      int phase_3(const char* str){
100         if(strlen(str) != 8){
(gdb) n
96          return ret;
3: phase_box = {14, 6, 8, 10, 3, 2, 13, 5, 11, 4, 9, 12, 7, 0, 1, 15}
6: input_buffer = "111\000\060\060\063\060e8", '\000' <repeats 89 times>
7: sum = 4
```

- 然后汇编的代码进行阅读，做出了如下的推理

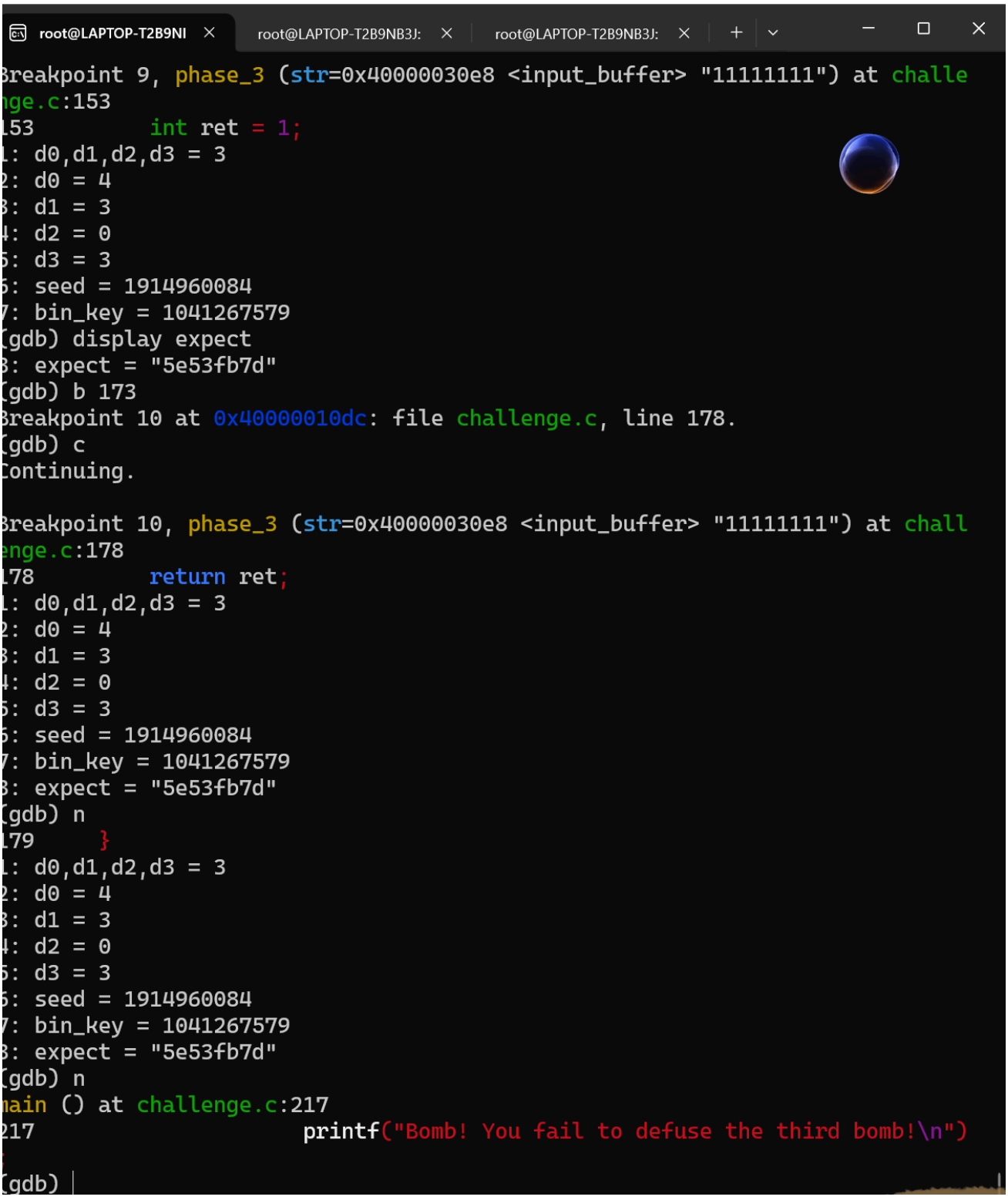


- 发现要满足 4^a^b=0
- 经过实验发现40满足上面的答案

phase3

- 用了phase2里面的方法之后找到phase3的代码部分，发现最终的input是和 `expect` 的函数进行比对的，查看了input的断点运行之后发现input的值并没有发现改变。

- 所以我只需要调用出来最终expect的值我就可以知道我应该输入什么值了



```
(gdb) n
179     }
1: d0,d1,d2,d3 = 3
2: d0 = 4
3: d1 = 3
4: d2 = 0
5: d3 = 3
6: seed = 1914960084
7: bin_key = 1041267579
8: expect = "5e53fb7d"
(gdb) n
main () at challenge.c:217
217                     printf("Bomb! You fail to defuse the third bomb!\n");
(gdb)
```

- 得到expect的值是 `5e53fd7d`