ICS PA2 Report

集成电路 211870293 李居奇

实验进度

- [x] 完成指令集的补充和测试
- [x] 完成库函数的实现和测试
- [x] 完成各种踪迹的记录(mtrace, ftrace, itrace, ring)，优化调试工具 (diff)
- [x] 完成所有外设实现，包括声卡
- [x] 通过 OJ 所有测试 ✌

实验必答题 & 思考题

Question 1

> YEMU 可以看成是一个简化版的 NEMU, 它们的原理是相通的, 因此你需要理解YEMU是如何执行程序的. 具体地, 你需要:
>
> - 画出在 YEMU 上执行的加法程序的状态机
> - 通过 RTFSC 理解 YEMU 如何执行一条指令
>
> 思考一下, 以上两者有什么联系?

我们定义一个状态为 (PC, R[0], R[1], M[dst]) 的状态机，则状态转移图为：

```
(0, x, x, 0) -> (0, 33, x, 0) -> (1, 33, x, 0)
-> (1, 33, 33, 0) -> (2, 33, 33, 0) -> (2, 16, 33, 0)
-> (3, 16, 33, 0) -> (3, 49, 33, 0) -> (4, 49, 33, 0)
-> (4, 49, 33, 49) -> (5, 49, 33, 49) -> (5, 49, 33, 49)
```

YEMU 执行指令的流程在本质上与 NEMU 完全一致，满足如下循环：

- 取指：将 PC 指向的指令读出
- 译码：利用 switch 结构逐条译码匹配
- 执行：执行指令对应的操作，读写寄存器/内存，或者数值计算
- 更新：使 PC 指向下一条将要执行的指令

Question 2

> 为了避免你长时间对代码的理解没有任何进展(~~那必不可能~~😎), 我们就增加一道必答题吧:
> 请整理一条指令在 NEMU 中的执行过程.

总体而言，NEMU 的指令执行过程与上一题基本一致，可以分成以下几个部分：

- ***取指***：NEMU 中 指令的读取由以下函数完成，该函数调用 `paddr_read`，从 `snpc` 指向的内存中读取指令，读取指令结束后 `snpc = snpc + 4`，函数返回后读取的指令保存在 `s->isa.inst.val`，注意这里并**不是** PC 真正更新的地方.

```
static inline uint32_t inst_fetch(vaddr_t *pc, int len) {
  uint32_t inst = vaddr_ifetch(*pc, len);
  (*pc) += len;
  return inst;
}
```

- ***译码***：由于需要翻译的指令基本包含了整个 RISCV-32 的基础指令集，这部分的封装比前文提到的 YEMU 复杂，精妙很多. 顶层函数为：`decode_exec(Decode *s)`，这里不得不提到 `Decode s` 结构体，它的定义如下：

```
typedef struct Decode {
  vaddr_t pc;
  vaddr_t snpc; // static next pc
  vaddr_t dnpc; // dynamic next pc
  ISADecodeInfo isa;
  IFDEF(CONFIG_ITRACE, char logbuf[128]);
} Decode;
```

结构体的三个指针：`pc, snpc, dnpc`和 CPU 中的 PC 指针在初读代码时会感到相当头疼😇，简单来说可以分成四个阶段：

- 取指前，`snpc = PC, pc = PC`
- 刚取指之后，`snpc = snpc + 4`
- 指令刚执行之后，`dnpc = 下一条指令`
- 最后更新 PC 时，`PC = dnpc`

而 `isa` 是用来存放取到的指令的，前文已经提过. 在顶层函数中，最核心的是宏 `INSTPAT`，以识别指令 `addi` 为例：

```
do {
    uint64_t key, mask, shift;
    pattern_decode("??????? ????? ????? 000 ????? 0010011",
    (sizeof("??????? ????? ????? 000 ????? 0010011") - 1), &key, &mask, &shift);
    if ((((uint64_t)((s)->isa.inst.val) >> shift) & mask) == key) {
        { decode_operand(s, &rd, &src1, &src2, &imm, TYPE_I);
        (cpu.gpr[check_reg_idx(rd)]) = src1 + imm ; };
        goto *(__instpat_end); }
}
while (0)
```

在 `pattern_decode` 中当指令部分出现问号时，跳过识别该部分，如果指令的 0 1 部分完全匹配，则进入下一步执行操作，执行完成后通过 `goto` 语句直接跳出函数，不再遍历后续指令是否匹配，如果不匹配，则继续执行下一个宏. 当整个表都被遍历完成还没有匹配成功时，则通过 `ebreak` 向 NEMU 抛出 hit bad trap 异常.

- ***执行***: 执行可以大致分为两个部分，一是对指令操作数的解析，提取立即数，内存地址，源寄存器和目的寄存器编号等，并按照指令要求进行无符号拓展或者有符号拓展，二是对寄存器和内存进行读写. 在操作数解析环节，主要由函数 `decode_operand` 完成，查阅 RISCV-32 的手册可知，整个基本指令集由 6 种指令组成，每种指令共享一个"模版"，即只要是相同的种类，就可以通过同样的方式提取到操作数. 具体实现过程这里不赘述，思路反正是很简单的. 在读写环节，直接按照指令手册的要求进行读写计算即可
- ***更新***: 更新也可以大致分成两个部分，第一部分为更新 PC 指针，由于考虑的跳转指令的出现，故不能简单的进行 `PC = PC + 4`，而是需要动态，静态指针辅助完成更新，具体过程前文已经提到；第二部分主要在函数`trace_and_difftest` 和 `device_update`中，实现 NEMU 运行状态的更新，执行 trace 相关的 debug 工具，以及更新外设如 VGA 相关的信息，完成这些后 NEMU 进入下一条指令的执行.

Question 3

> 请你以打字小游戏为例, 结合"程序在计算机上运行"的两个视角, 来剖析打字小游戏究竟是如何在计算机上运行的. 具体地, 当你按下一个字母并命中的时候, 整个计算机系统(NEMU, ISA, AM, 运行时环境, 程序) 是如何协同工作, 从而让打字小游戏实现出"命中"的游戏效果?

`video_init()` 在程序开始运行前清理了整个屏幕，确定了 texture 中字母的颜色. `game_logic_update()` 负责生成字母，并赋予随机的字符，随机的速度. 同时更新每个字母在屏幕上的位置，并将已经达到屏幕底部的字母判定为 Miss，程序在 while 循环中检查按键情况, 当按键按下并且定义了按键符号, 程序开始执行 `check_hit()`, 根据键盘读取的字母查找是否在屏幕上的字母中，如果在，则字母速度向上，同时 hit++, 如果不在则 wrong++，最后 `render()` 显示结果, 如果速度为正数, 字母颜色为白色, 为负数, 字母颜色为绿色, 速度等于零, 为红色，同时将当时的结果实时打印在终端上.

当我按下一个字母时，程序通过 AM 的 IOE 访问 NEMU 的键盘外设，判断是否按下正确字母，得到结果后再通过 IOE 向 NUMU 的 GPU 写入数据，来更新整个屏幕画面.

Question 4

> 在 nemu/include/cpu/ifetch.h 中, 你会看到由 static inline 开头定义的 `inst_fetch()` 函数. 分别尝试去掉 static, 去掉 inline 或去掉两者, 然后重新进行编译, 你可能会看到发生错误. 请分别解释为什么这些错误会发生/不发生? 你有办法证明你的想法吗?

只删除 static 或者 inline 编译不会报错😊，如果将两者都删除则编译器报错如下：

```
hostcall.o: in function `inst_fetch':
cpu/ifetch.h:20: multiple definition of `inst_fetch';
cpu/ifetch.h:20: first defined here.
```

这是一个重定义错误，原因是 `inst_fetch()` 函数直接在 h 文件中定义. 我们知道 `include` 的本质是原封不动的复制粘贴，如果有两个 C 文件都写了 `include <ifetch.h>`，则在两个 C 文件中都会出现 `inst_fetch()` 函数的定义，而此时如果对两个 C 文件编译后的 .o 文件进行链接，就会出现重定义的情况.

我们知道重定义的本质是因为两个目标文件的符号表中出现了相同的强符号，下面来解释为什么加上两者中的任意一个都便不再报错，加上 static 之后该函数变成静态函数，静态变量仅在该 C 文件内可见，故链接时不会出现重定义；

而当一个函数被声明为 inline 时，编译器会尝试将函数的代码在编译时直接复制到每个调用它的地方，这样类似宏的操作可以避免函数调用的开销. 所以在加上 inline 之后，函数可能根本不会被"定义"，其函数名甚至不会出现在符号表中，这样也自然也不会被认为重定义. 验证方法：读取可执行文件的符号表，显示加上 inline 和不加 inline 时变量的出现次数：

```
dracacys@dracacys-virtual-machine:~/ics2023/am-kernels/kernels/typing-game$ read
elf --symbols $NEMU_HOME/build/riscv32-nemu-interpreter | grep -c inst_fetch
0
```

加上 inline

```
dracacys@dracacys-virtual-machine:~/ics2023/am-kernels/kernels/typing-game$ read
elf --symbols $NEMU_HOME/build/riscv32-nemu-interpreter | grep -c inst_fetch
2
dracacys@dracacys-virtual-machine:~/ics2023/am-kernels/kernels/typing-game$ read
elf --symbols $NEMU_HOME/build/riscv32-nemu-interpreter | grep inst_fetch
    74: 000000000000d942   104 FUNC    LOCAL  DEFAULT   16 inst_fetch
   242: 0000000000019c2d   104 FUNC    LOCAL  DEFAULT   16 inst_fetch
```

不加 inline

Question 5

> 在 nemu/include/common.h 中添加一行 volatile static int dummy; 然后重新编译 NEMU. 请问重新编译后的 NEMU 含有多少个 dummy 变量的实体? 你是如何得到这个结果的?

由上一题的分析可知，实体变量可以查询链接后的符号表，用类似的命令即可显示 dummy 出现的次数，排除前两个不是变量，可知总共有 34 个实体：

```
dracacys@dracacys-virtual-machine:~/ics2023/am-kernels/kernels/typing-game$ readelf --symbols $NEMU_HOME/build/riscv32-nemu-interpreter | grep -c dummy
36
dracacys@dracacys-virtual-machine:~/ics2023/am-kernels/kernels/typing-game$ readelf --symbols $NEMU_HOME/build/riscv32-nemu-interpreter | grep dummy
     9: 000000000000cae0     0 FUNC    LOCAL  DEFAULT   16 frame_dummy
    10: 0000000000025778     0 OBJECT  LOCAL  DEFAULT   22 __frame_dummy_in[...]
    14: 000000000002d080     4 OBJECT  LOCAL  DEFAULT   28 dummy
    23: 000000000002d160     4 OBJECT  LOCAL  DEFAULT   28 dummy
    27: 000000000002d1a0     4 OBJECT  LOCAL  DEFAULT   28 dummy
    35: 000000000002d220     4 OBJECT  LOCAL  DEFAULT   28 dummy
    43: 000000000002d2a0     4 OBJECT  LOCAL  DEFAULT   28 dummy
    58: 000000000002e7e0     4 OBJECT  LOCAL  DEFAULT   28 dummy
    70: 000000000002e920     4 OBJECT  LOCAL  DEFAULT   28 dummy
    77: 000000000002e9e0     4 OBJECT  LOCAL  DEFAULT   28 dummy
    81: 000000000002ea20     4 OBJECT  LOCAL  DEFAULT   28 dummy
    89: 000000000002ea60     4 OBJECT  LOCAL  DEFAULT   28 dummy
    95: 000000000002eaa0     4 OBJECT  LOCAL  DEFAULT   28 dummy
   100: 000000000002eae0     4 OBJECT  LOCAL  DEFAULT   28 dummy
   104: 000000000002eb20     4 OBJECT  LOCAL  DEFAULT   28 dummy
   108: 000000000002eb60     4 OBJECT  LOCAL  DEFAULT   28 dummy
   113: 000000000002eba0     4 OBJECT  LOCAL  DEFAULT   28 dummy
   131: 000000000002ec80     4 OBJECT  LOCAL  DEFAULT   28 dummy
   141: 000000000002ef20     4 OBJECT  LOCAL  DEFAULT   28 dummy
   152: 000000000002f1c0     4 OBJECT  LOCAL  DEFAULT   28 dummy
   156: 000000000002f240     4 OBJECT  LOCAL  DEFAULT   28 dummy
   160: 0000000000089180     4 OBJECT  LOCAL  DEFAULT   28 dummy
   171: 0000000000089360     4 OBJECT  LOCAL  DEFAULT   28 dummy
   179: 00000000000893c0     4 OBJECT  LOCAL  DEFAULT   28 dummy
   183: 0000000000089500     4 OBJECT  LOCAL  DEFAULT   28 dummy
   198: 0000000000089660     4 OBJECT  LOCAL  DEFAULT   28 dummy
   220: 0000000000089720     4 OBJECT  LOCAL  DEFAULT   28 dummy
   230: 000000000008ab80     4 OBJECT  LOCAL  DEFAULT   28 dummy
   242: 000000000108ef80     4 OBJECT  LOCAL  DEFAULT   28 dummy
   248: 000000000108f000     4 OBJECT  LOCAL  DEFAULT   28 dummy
   252: 000000000108f040     4 OBJECT  LOCAL  DEFAULT   28 dummy
   258: 000000000108f0e0     4 OBJECT  LOCAL  DEFAULT   28 dummy
   262: 000000000108f140     4 OBJECT  LOCAL  DEFAULT   28 dummy
   272: 0000000001090000     4 OBJECT  LOCAL  DEFAULT   28 dummy
   276: 0000000009091000     4 OBJECT  LOCAL  DEFAULT   28 dummy
   280: 0000000009091040     4 OBJECT  LOCAL  DEFAULT   28 dummy
```

Question 6

> 添加上题中的代码后, 再在 nemu/include/debug.h 中添加一行 volatile static int dummy; 然后重新编译 NEMU. 请问此时的NEMU含有多少个 dummy 变量的实体? 与上题中 dummy 变量实体数目进行比较, 并解释本题的结果.

得到的结果和之前完全一致，因为 `debug.h` 其实包含了 `common.h`，而对于包含了 `debug.h` 的文件而言，相当于在文件中写入了两次 `volatile static int dummy;`，对于没有赋值的弱定义变量，编译器会认为第一次为定义，第二次为申明，故不会增加变量实体的个数.

Question 7

> 修改添加的代码, 为两处 dummy 变量进行初始化: volatile static int dummy = 0; 然后重新编译 NEMU. 你发现了什么问题? 为什么之前没有出现这样的问题?

编译器报错 `redefinition of 'dummy'`，因为初始化之后的 dummy 是强定义，对于包含了 `debug.h` 的文件， dummy 相当于出现了两次强定义，会被判定为重定义. 而之前是两次弱定义，故不会报错.

Question 8

> 请描述你在 am-kernels/kernels/hello/ 目录下敲入 `make ARCH=$ISA-nemu` 后, make 程序如何组织 .c 和 .h 文件, 最终生成可执行文件 `am-kernels/kernels/hello/build/hello-$ISA-nemu.elf`
>
> - Makefile 中使用了变量, 包含文件等特性
> - Makefile 运用并重写了一些 implicit rules
> - 在 man make 中搜索 -n 选项, 也许会对你有帮助
> - RTFM

我们知道整体框架可以分成两个部分：

1. AM 端编译生成镜像文件 IMG，作为系统软件部分
2. NEMU 端编译生成 Interpreter，然后模仿硬件执行 IMG 文件

首先 hello 中的 Makefile 将 NAME 和 SRC 设置好，然后将 AM_HOME 中的 Makefile 贴进来.

```
NAME = hello
SRCS = hello.c
include $(AM_HOME)/Makefile
```

考虑到 native 和 NEMU 或者其他 ISA 的通用性，AM_HOME 下的 Makefile 主要定义了一些通用的编译选项，编译规则. 如 C 文件到目标文件的编译规则如下，它会将所有的目标文件放到 DST_DIR, 即 AM 下的 build 目录. 而镜像文件则是使用自己的 Klib 库和其他目标文件链接产生.

```
$(DST_DIR)/%.o: %.c
    @mkdir -p $(dir $@) && echo + CC $<
    @$(CC) -std=gnu11 $(CFLAGS) -c -o $@ $(realpath $<)
$(IMAGE).elf: $(OBJS) am $(LIBS)
    @echo + LD "->" $(IMAGE_REL).elf
    @$(LD) $(LDFLAGS) -o $(IMAGE).elf --start-group $(LINKAGE) --end-group
```

上述文件只是定义了规则，具体的编译执行过程是在该 Makefile 包含的 `-include $(AM_HOME)/scripts/$(ARCH).mk` 中完成，当我们键入`make ARCH=$ISA-nemu run` 后，就相当于前往对应的 $(ARCH).mk 中执行操作，以 riscv32-nemu 为例，其包含了 `nemu.mk`，该 Makefile 定义的目标如下：

```
run: image
	$(MAKE) -C $(NEMU_HOME) ISA=$(ISA) run ARGS="$(NEMUFLAGS)" IMG=$(IMAGE).bin
```

完成的操作主要有在 AM 端编译生成 image 文件，随后前往 NEMU 所在的目录，传入 ISA, ARGS, IMG 等参数后，在目录下执行 make run，与上述类似，该命令负责先编译出 NEMU 的 Interpreter，然后传入并执行 IMG 的内容.

实验关键步骤

顺利理解宏

> 为了理解一个宏的语义, 你可能会尝试手动对它进行宏展开, 但你可能会碰到如下困难:
> 宏嵌套的次数越多, 理解越困难一些拼接宏会影响编辑器的代码跳转功能.

我才用的方法是修改 Makefile 的编译规则，在生成目标文件的同时生成预处理文件，这样宏会被自动展开如下，方便 RTFSC：

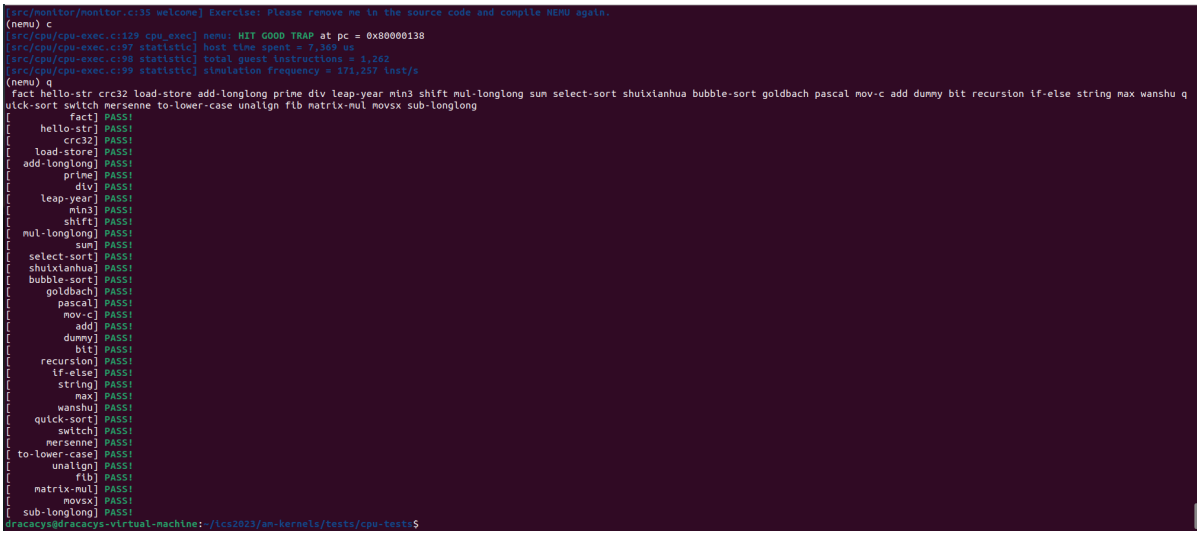
```
static int decode_exec(Decode *s) {
  int rd = 0;
  word_t src1 = 0, src2 = 0, imm = 0;
  s->dnpc = s->snpc;
#define INSTPAT_INST(s) ((s)->isa.inst.val)
#define INSTPAT_MATCH(s,name,type,...) { decode_operand(s, &rd, &src1, &src2, &imm, concat(TYPE_, type)); __VA_ARGS__ ; }
  { const void ** __instpat_end = &&__instpat_end_;
  do { uint64_t key, mask, shift; pattern_decode("??????? ????? ????? ??? ????? 00101 11", (sizeof("??????? ????? ????? ??? ????? 00101 1
  do { uint64_t key, mask, shift; pattern_decode("??????? ????? ????? 100 ????? 00000 11", (sizeof("??????? ????? ????? 100 ????? 00000 1
  do { uint64_t key, mask, shift; pattern_decode("??????? ????? ????? 000 ????? 01000 11", (sizeof("??????? ????? ????? 000 ????? 01000 1
  do { uint64_t key, mask, shift; pattern_decode("0000000 00001 00000 000 00000 11100 11", (sizeof("0000000 00001 00000 000 00000 11100 1
  do { uint64_t key, mask, shift; pattern_decode("??????? ????? ????? ??? ????? ????? ??", (sizeof("??????? ????? ????? ??? ????? ????? ?
  __instpat_end_: ; };
  (cpu.gpr[check_reg_idx(0)]) = 0;
  return 0;
}
int isa_exec_once(Decode *s) {
  s->isa.inst.val = inst_fetch(&s->snpc, 4);
  return decode_exec(s);
}
```

实现指令集以及自动测试

完成指令集后，需要在 cpu-tests 下运行十几个测试文件，也就是每次运行 NUMU 时都需要敲 c 运行再敲 q 退出，为了偷懒🤯，可以修改 Makefile 如下，执行 make test 之后自动在每次运行时传入 cq.

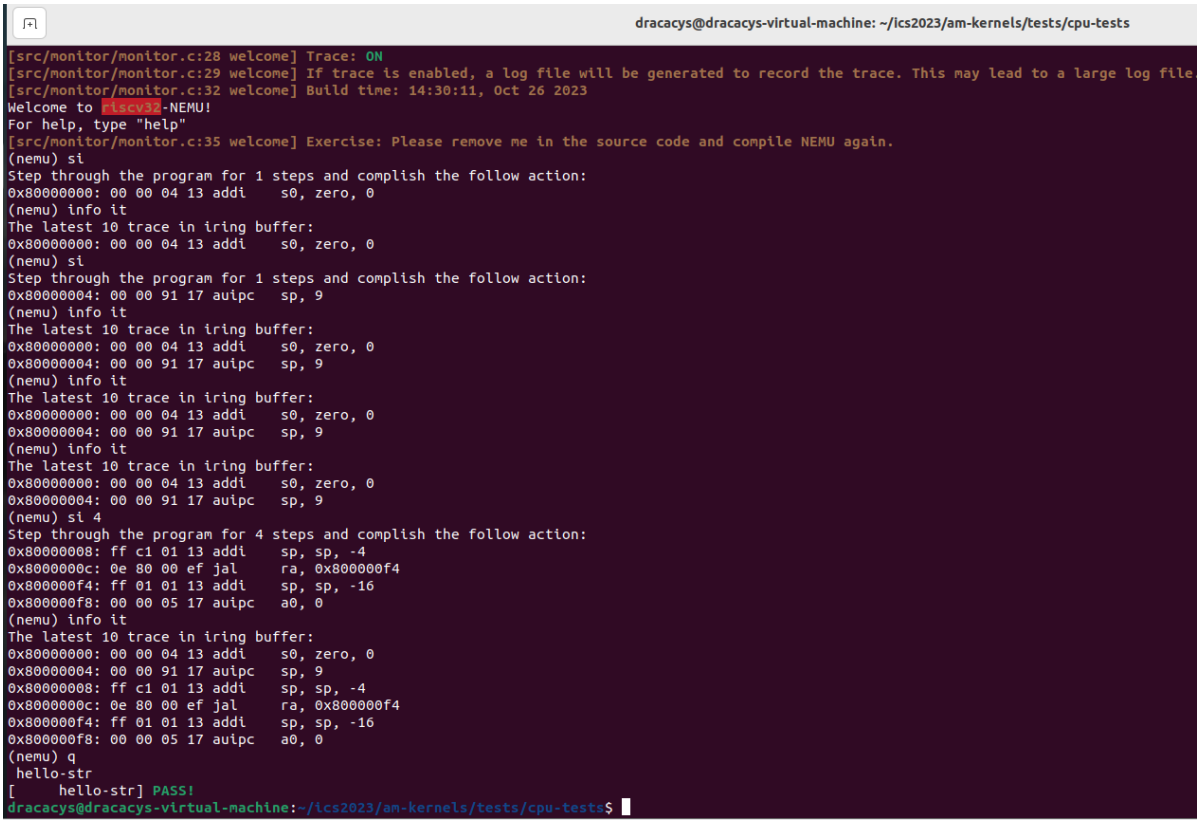
```
test: run-env
	$(call git_commit, "run NEMU")
	@echo "c\nq" | $(NEMU_EXEC)
```

测试通过截图：



Ring buffer

为了调试的便捷
这里数据结构采用环形链表的方式，方便在打印的时候先打印旧的，再打印新的，以及在 buffer 满了之后先覆盖旧值，完成后效果如下：




```
#define buffer_size 10
typedef struct buffer
{
  char log_buf[80];
  bool use_state;
  struct buffer *next;
}
ring_buffer;

ring_buffer* ring_head = NULL;

// 初始化环形链表
```

```
ring_buffer *init_ring_buffer(void) {
  ring_buffer *ptr = (ring_buffer *)malloc(sizeof(ring_buffer));
  ptr->log_buf[0] = '\0';
  ptr-> use_state = false;
  ring_buffer *head = ptr;
  for (int i = 1; i < buffer_size; i++) {
    ptr->next = (ring_buffer *)malloc(sizeof(ring_buffer));
    ptr = ptr->next;
    ptr->log_buf[0] = '\0';
    ptr-> use_state = false;
  }
  ptr->next = head;
  return head;
}

// 写入buffer
ring_buffer *write_ring_buffer(ring_buffer *head, char *log_str) {
  strcpy(head->log_buf, log_str);
  head->use_state = true;
  return head->next;
}

// 打印buffer的全部内容
void print_ring_buffer(ring_buffer *head) {
  ring_buffer *ptr = head;
  puts("The latest 10 ITRACE in ring buffer:");
  if (ptr->use_state == true) {
    while(1) {
      puts(ptr->log_buf);
      if (ptr->next == head) {
        break;
      }
      else {
        ptr = ptr->next;
      }
    }
  }
  //buf未填满
  else {
    while(ptr->use_state != true) {
      if(ptr->next == head) {
        return;
      }
      ptr = ptr->next;
    }
    while(1) {
      puts(ptr->log_buf);
      if (ptr->next == head) {
        break;
      }
      else {
        ptr = ptr->next;
      }
    }
  }
}
```

```
// 销毁buffer空间
void destroy_ring_buffer(ring_buffer *head) {
  ring_buffer *ptr = head;
  ring_buffer *next_;
  while(1) {
    next_ = ptr->next;
    free(ptr);
    if(next_ == head) {
      break;
    }
    else {
      ptr = next_;
    }
  }
}
```

Ftreace

Ftrace 的符号表和解析和函数名存储都有一定难度，符号表解析的函数名我用数组的形式存放，但似乎这样过于粗糙，因为如果程序是像 Mario 之类的游戏，我的数组很容易溢出. 目前我的 Ftrace 只能用调试较小的程序，这个问题我还尚未着手解决.

函数名的存储我采用栈的形式，指令跳转时判断是 ret 还是 call 对于我的实现而言十分简单. 如果是 `ret(0x00008067)` 则栈顶元素出栈，判断其执行了返回; 如果是跳转但不是 ret，则在符号表中查询该地址是否落在栈顶函数的作用域中，不在则新函数入栈, 判断其执行了跳转.

实现后效果如下:

```
[src/cpu/cpu-exec.c:124 statistic] host time spent = 23,089,539 us
[src/cpu/cpu-exec.c:125 statistic] total guest instructions = 8,256,468
[src/cpu/cpu-exec.c:126 statistic] simulation frequency = 357,584 inst/s
(nemu) info f-table
FTRACE table:
name:__am_timer_config          addr:0x80000878                size:16
name:__am_input_config          addr:0x80000888                size:12
name:__am_uart_config           addr:0x80000894                size:8
name:__am_net_config            addr:0x8000089c                size:8
name:fail                       addr:0x800008a4                size:104
name:printf                     addr:0x80000e9c                size:152
name:vsprintf                   addr:0x80000d50                size:332
name:_trm_init                  addr:0x80000858                size:32
name:__am_gpu_status            addr:0x80000b28                size:12
name:__am_audio_status          addr:0x80000ba4                size:16
name:__am_audio_init            addr:0x80000b34                size:4
name:ioe_write                  addr:0x80000984                size:28
name:ioe_read                   addr:0x80000968                size:28
name:__udivmoddi4               addr:0x80000f78                size:804
name:__am_gpu_config            addr:0x80000a08                size:52
name:__am_input_keybrd          addr:0x800009dc                size:40
name:__udivdi3                  addr:0x8000129c                size:8
name:putch                      addr:0x80000840                size:12
name:__am_disk_blkio            addr:0x80000c88                size:4
name:__am_gpu_fbdraw            addr:0x80000a3c                size:236
name:__am_timer_rtc             addr:0x800009bc                size:32
name:__clzsi2                   addr:0x800012a4                size:152
name:__am_audio_config          addr:0x80000b38                size:20
name:_start                     addr:0x80000000                size:0
name:__am_disk_status           addr:0x80000c84                size:4
name:rand                       addr:0x80000f34                size:52
name:__am_timer_uptime          addr:0x800009a4                size:24
name:render                     addr:0x800001a8                size:504
name:__am_gpu_init              addr:0x80000a04                size:4
name:__am_timer_init            addr:0x800009a0                size:4
name:__am_audio_ctrl            addr:0x80000b4c                size:88
name:memset                     addr:0x800013b0                size:32
name:main                       addr:0x80000688                size:440
name:game_logic_update          addr:0x800000dc                size:204
name:ioe_init                   addr:0x8000090c                size:92
name:__am_audio_play            addr:0x80000bb4                size:200
name:__ctzsi2                   addr:0x8000133c                size:116
name:check_hit                  addr:0x800003a0                size:208
name:halt                       addr:0x8000084c                size:12
name:int2str                    addr:0x80000c8c                size:196
```

打印函数表

```
[src/cpu/cpu-exec.c:124 statistic] host time spent = 23,089,539 us
[src/cpu/cpu-exec.c:125 statistic] total guest instructions = 8,256,468
[src/cpu/cpu-exec.c:126 statistic] simulation frequency = 357,584 inst/s
(nemu) info f-table
FTRACE table:
name:__am_timer_config          addr:0x80000878                size:16
name:__am_input_config          addr:0x80000888                size:12
name:__am_uart_config           addr:0x80000894                size:8
name:__am_net_config            addr:0x8000089c                size:8
name:fail                       addr:0x800008a4                size:104
name:printf                     addr:0x80000e9c                size:152
name:vsprintf                   addr:0x80000d50                size:332
name:_trm_init                  addr:0x80000858                size:32
name:__am_gpu_status            addr:0x80000b28                size:12
name:__am_audio_status          addr:0x80000ba4                size:16
name:__am_audio_init            addr:0x80000b34                size:4
name:ioe_write                  addr:0x80000984                size:28
name:ioe_read                   addr:0x80000968                size:28
name:__udivmoddi4               addr:0x80000f78                size:804
name:__am_gpu_config            addr:0x80000a08                size:52
name:__am_input_keybrd          addr:0x800009dc                size:40
name:__udivdi3                  addr:0x8000129c                size:8
name:putch                      addr:0x80000840                size:12
name:__am_disk_blkio            addr:0x80000c88                size:4
name:__am_gpu_fbdraw            addr:0x80000a3c                size:236
name:__am_timer_rtc             addr:0x800009bc                size:32
name:__clzsi2                   addr:0x800012a4                size:152
name:__am_audio_config          addr:0x80000b38                size:20
name:_start                     addr:0x80000000                size:0
name:__am_disk_status           addr:0x80000c84                size:4
name:rand                       addr:0x80000f34                size:52
name:__am_timer_uptime          addr:0x800009a4                size:24
name:render                     addr:0x800001a8                size:504
name:__am_gpu_init              addr:0x80000a04                size:4
name:__am_timer_init            addr:0x800009a0                size:4
name:__am_audio_ctrl            addr:0x80000b4c                size:88
name:memset                     addr:0x800013b0                size:32
name:main                       addr:0x80000688                size:440
name:game_logic_update          addr:0x800000dc                size:204
name:ioe_init                   addr:0x8000090c                size:92
name:__am_audio_play            addr:0x80000bb4                size:200
name:__ctzsi2                   addr:0x8000133c                size:116
name:check_hit                  addr:0x800003a0                size:208
name:halt                       addr:0x8000084c                size:12
name:int2str                    addr:0x80000c8c                size:196
```

打印函数调用记录

主要代码如下:

```c
typedef struct ftrace_f {
    uint32_t addr;    // function address
    char name [50];   // function name
    uint32_t size;    // function size
} ftrace_fun;

typedef struct ftrace_s {
    uint32_t fun_table_index;
    uint32_t fun_stack_index;
} ftrace_stack;

FILE *elf_fp = NULL;
ftrace_fun ftrace_table [FTRACE_TABLE_SIZE] = {0};  // record fun
uint32_t ftrace_table_size = 0;
ftrace_stack fun_stack [FUN_STACK_SIZE] = {0};      // fun stack
int32_t stack_top = -1;                             // 栈顶指针
uint32_t stack_cum = 0;                             // 累计入栈函数个数
char flog [FUN_LOG_SIZE][150] = {0};                // 调用返回记录
uint32_t flog_indx = 0;

#define FTRACE_COND (strcmp(CONFIG_FTRACE_COND, "true") == 0)

// Success return 1, else return 0
```

```c
uint8_t init_ftrace(char *elf_addr) {

    Elf32_Ehdr ehdr = {0};           // ELF头
    Elf32_Shdr shdr [1000] = {0};    // 节头表
    Elf32_Sym sym_table [1000]= {0}; // 符号表
    size_t read_size;
    if ((elf_fp = fopen(elf_addr, "rb")) == NULL) {
        puts("读取ELF文件失败, ftrace 未启动.");
        return 0;
    }
    else {
        char magic_buf[6] = {0};
        fseek(elf_fp, 0, SEEK_SET);
        read_size = fread(magic_buf, 1, 5, elf_fp);
        //魔数检查
        if (magic_buf[0] != 0x7f || magic_buf[1] != 'E'
        || magic_buf[2] != 'L' || magic_buf[3] != 'F') {
            puts("文件类型错误, ftrace 未启动.");
            return 0;
        }
        if (magic_buf[4] != 0x01) {
            puts("警告: ELF文件非32位系统生成.");
        }
    }
    fseek(elf_fp, 0, SEEK_SET);
    // 读取ELF头
    read_size = fread(&ehdr, sizeof(Elf32_Ehdr), 1, elf_fp);
    // 跳转到节头表, 并全部读取
    fseek(elf_fp, ehdr.e_shoff, SEEK_SET);
    read_size = fread(&shdr, ehdr.e_shentsize, ehdr.e_shnum, elf_fp);
    // 跳转到shstrtab, 并全部读取
    char shstr_table [10000] = {0};
    fseek(elf_fp, shdr[ehdr.e_shstrndx].sh_offset, SEEK_SET);
    read_size = fread(&shstr_table, 1, shdr[ehdr.e_shstrndx].sh_size, elf_fp);
    // 查找 strtab, 并跳转读取
    uint32_t index;
    for(index = ehdr.e_shnum - 1; index >= 0; index--) {
        if(strcmp(".strtab", &shstr_table[shdr[index].sh_name]) == 0) {
            break;
        }
    }
    char str_table [10000] = {0};
    fseek(elf_fp, shdr[index].sh_offset, SEEK_SET);
    read_size = fread(&str_table, 1, shdr[index].sh_size, elf_fp);
    // 查找 symtab, 并跳转读取
    for(index = ehdr.e_shnum - 1; index >= 0; index--) {
        if(strcmp(".symtab", &shstr_table[shdr[index].sh_name]) == 0) {
            break;
        }
    }
    fseek(elf_fp, shdr[index].sh_offset, SEEK_SET);
    int sym_num = shdr[index].sh_size / sizeof(Elf32_Sym);
    read_size = fread(&sym_table, sizeof(Elf32_Sym), sym_num, elf_fp);
    index = 0;
    // Parse symbol table
    for(int i = 0; i < sym_num; i++) {
```

```
        if(ELF32_ST_TYPE(sym_table[i].st_info) == STT_FUNC) {
            ftrace_table[index].addr = sym_table[i].st_value;
            ftrace_table[index].size = sym_table[i].st_size;
            strcpy(ftrace_table[index].name, &str_table[sym_table[i].st_name]);
            index++;
        }
    }
    ftrace_table_size = index;
    if(read_size == 0) return 0;
    else return 1;
}

void stack_push(uint32_t fun_index) {
    if(fun_index >= ftrace_table_size || stack_top >= 499) {
        return ;
    }
    stack_top++;
    fun_stack[stack_top].fun_table_index = fun_index;
    fun_stack[stack_top].fun_stack_index = stack_cum;
    stack_cum++;
    return;
}

void stack_get(ftrace_stack* temp) {
    if(stack_top <= -1) {
        return;
    }
    temp->fun_stack_index = fun_stack[stack_top].fun_stack_index;
    temp->fun_table_index = fun_stack[stack_top].fun_table_index;
    return;
}

void stack_pull(ftrace_stack* temp) {
    stack_get(temp);
    stack_top--;
    return;
}

// 获取当前函数在ftrace_table中的编号
int32_t get_fun_index(vaddr_t pc) {
    for(int i = 0; i < ftrace_table_size; i++) {
        if(pc >= ftrace_table[i].addr && pc <
        ftrace_table[i].addr + ftrace_table[i].size) {
            return i;
        }
    }
    //未找到相关函数
    return -1;
}

void ftrace_process(Decode *ptr) {
    // 下一条指令的所在函数序号
    int32_t ftab_index = get_fun_index(ptr->dnpc);
    // 不在函数中
    if(ftab_index == -1) {
        return;
```

```
    }
    else {
        ftrace_stack temp;
        // 非空栈
        if(stack_top > -1) {
            stack_get(&temp);
            // 与栈顶函数一致，非跳转
            if(temp.fun_table_index == ftab_index) {
            #ifdef CONFIG_FTRACE_COND
                if(FTRACE_COND){log_write("FTRACE: 0x%08x\t in
                (stack_idx = %03u)[%s@0x%08x]\n", ptr->pc, temp.
                fun_stack_index, ftrace_table[temp.fun_table_index].name,
                 ftrace_table[temp.fun_table_index].addr);}
            #endif
                return;
            }
            else {
                if(ptr->isa.inst.val == 0x00008067) {
                    stack_pull(&temp);
                #ifdef CONFIG_FTRACE_COND
                    if(FTRACE_COND){log_write("FTRACE: 0x%08x\t ret
                      (stack_idx = %03u)[%s@0x%08x]\n", ptr->pc, temp.
                      fun_stack_index, ftrace_table[temp.fun_table_index].
                      name, ftrace_table[temp.fun_table_index].addr);}
                #endif
                    if(flog_indx < FUN_LOG_SIZE) {sprintf(flog[flog_indx++],
                     "FTRACE: 0x%08x\t ret  (stack_idx = %03u)
                    [%s@0x%08x]\n", ptr->pc, temp.fun_stack_index,
                  ftrace_table[temp.fun_table_index].name, ftrace_table
                  [temp.fun_table_index].addr);}
                    return;
                }
                else {
                    stack_push(ftab_index);
                    stack_get(&temp);
                #ifdef CONFIG_FTRACE_COND
                    if(FTRACE_COND){log_write("FTRACE: 0x%08x\t call
                     (stack_idx = %03u)[%s@0x%08x]\n", ptr->pc, temp.
                     fun_stack_index, ftrace_table[temp.fun_table_index].
                     name, ftrace_table[temp.fun_table_index].addr);}
                #endif
                    if(flog_indx < FUN_LOG_SIZE) {sprintf(flog[flog_indx
                    ++], "FTRACE: 0x%08x\t call (stack_idx = %03u)
                    [%s@0x%08x]\n", ptr->pc, temp.fun_stack_index,
                  ftrace_table[temp.fun_table_index].name, ftrace_table
                  [temp.fun_table_index].addr);}
                    return;
                }
            }
        }
        // 空栈
        else {
            stack_push(ftab_index);
            stack_get(&temp);
        #ifdef CONFIG_FTRACE_COND
            if(FTRACE_COND){log_write("FTRACE: 0x%08x\t call
```

```
            (stack_idx = %03u)[%s@0x%08x]\n", ptr->pc, temp.
            fun_stack_index, ftrace_table[temp.fun_table_index].name,
            ftrace_table[temp.fun_table_index].addr);}
        #endif
            if(flog_indx < FUN_LOG_SIZE) {sprintf(flog[flog_indx++],
            "FTRACE: 0x%08x\t call (stack_idx = %03u)[%s@0x%08x]\n",
            ptr->pc, temp.fun_stack_index, ftrace_table[temp.
            fun_table_index].name, ftrace_table[temp.fun_table_index].addr);}
            return;
        }

    }
}

void ftrace_table_d(void) {
    puts("FTRACE table:");
    for(int i = 0; i < ftrace_table_size; i++) {
        printf("name:%-20s\taddr:0x%08x\t\t\tsize:%-5d\n",
        ftrace_table[i].name, ftrace_table[i].addr, ftrace_table[i].size);
    }
    return;
}

void ftrace_log_d(void) {
    puts("FTRACE log:");
    for(int i = 0; i < flog_indx; i++) {
        printf("%s", flog[i]);
    }
    return;
}
```

库函数和跑分结果

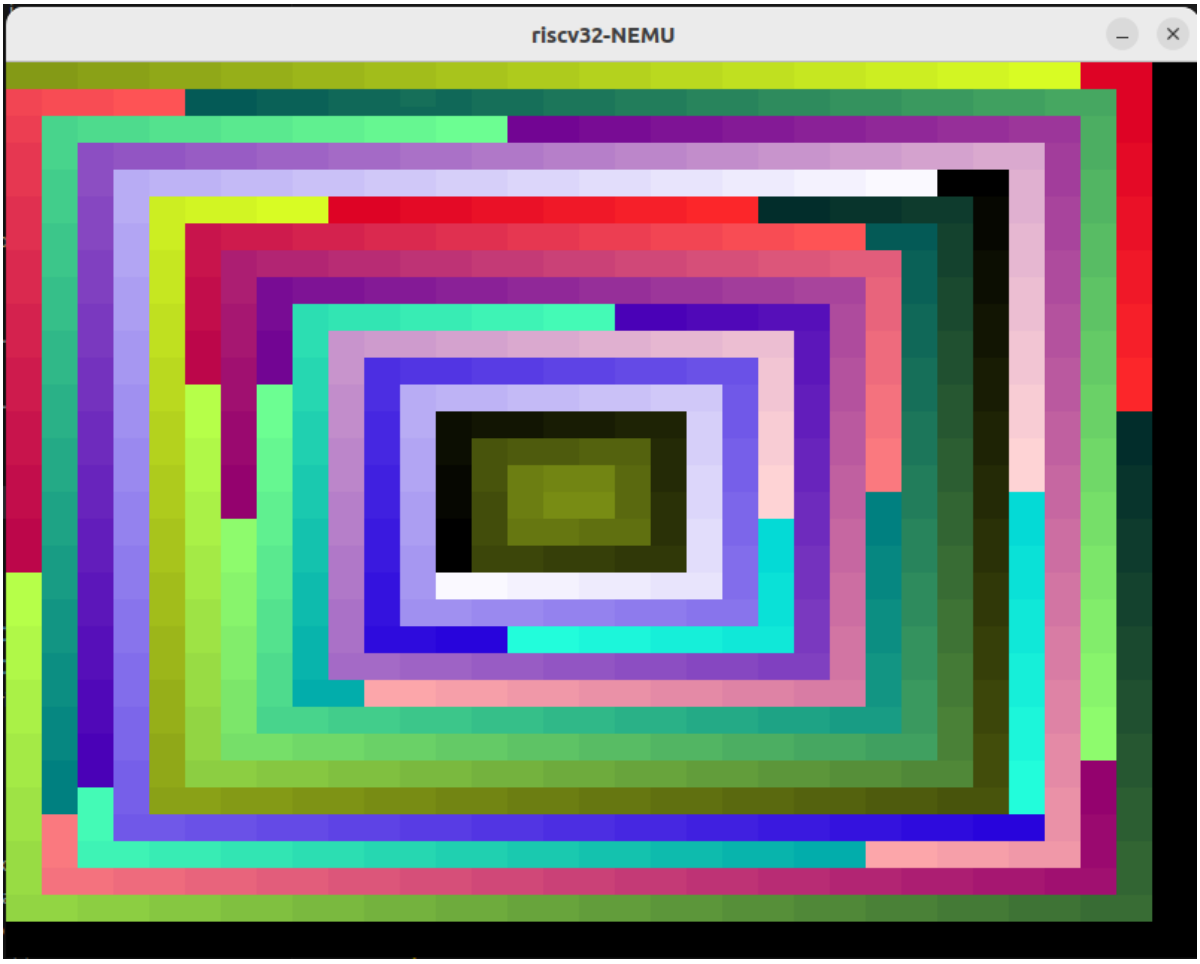
库函数部分太难了，其他函数还好，`printf` `sprintf`只实现了 %d 和 %s, 幸运的是已经可以通过 OJ 测试了，就懒得完善了 😛.　这里记录一个有 Linux C 源码参考的网站：LIBC
之后实现定时器外设之后，进行跑分的结果如下：

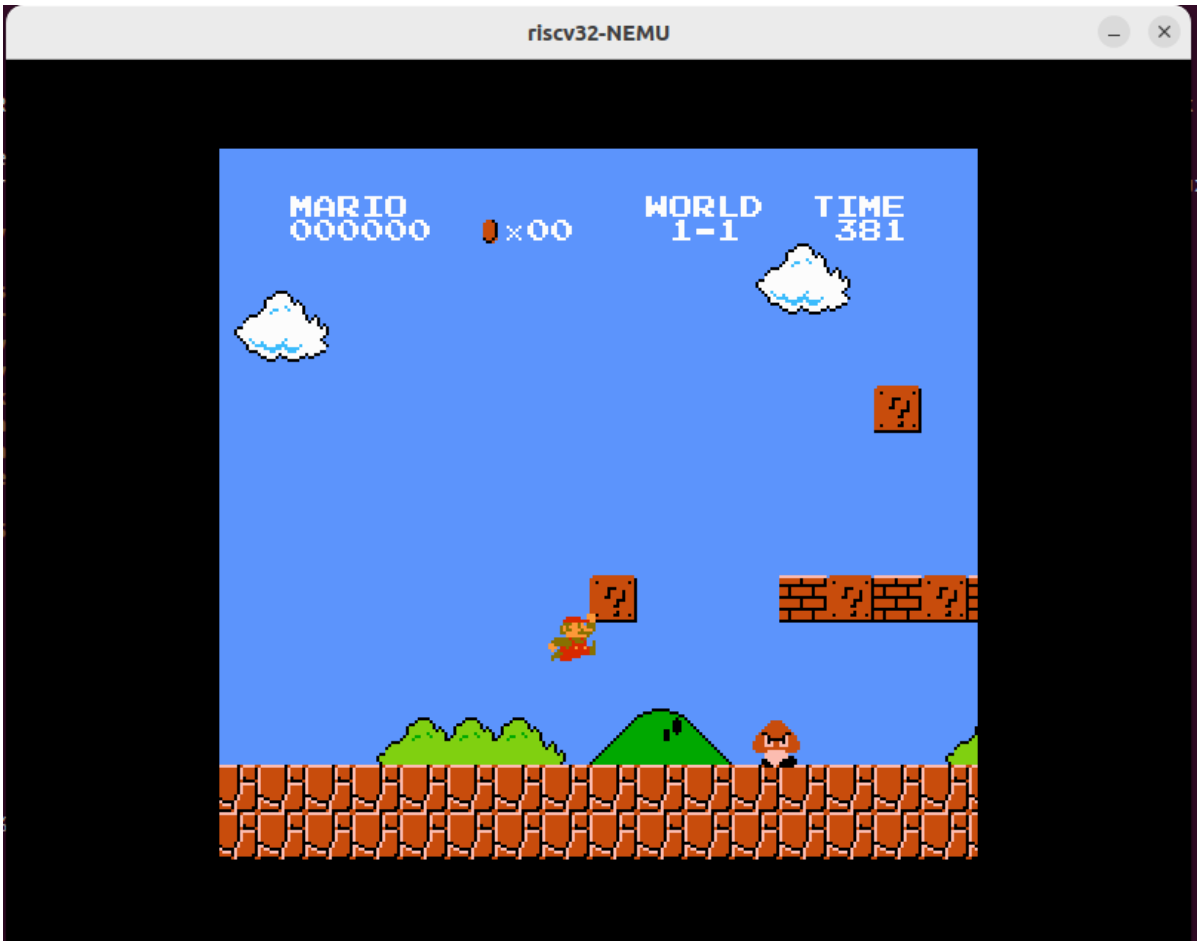
```
dracacys@dracacys-virtual-machine: ~/ics2023/am-kernels/benchmarks/microbench
[src/device/io/mmio.c:52 add_mmio_map] Add mmio map 'vgactl' at [0xa0000100, 0xa0000107]
[src/device/io/mmio.c:52 add_mmio_map] Add mmio map 'vmem' at [0xa1000000, 0xa10752ff]
[src/device/io/mmio.c:52 add_mmio_map] Add mmio map 'keyboard' at [0xa0000060, 0xa0000063]
[src/device/io/mmio.c:52 add_mmio_map] Add mmio map 'audio' at [0xa0000200, 0xa0000217]
[src/device/io/mmio.c:52 add_mmio_map] Add mmio map 'audio-sbuf' at [0xa1200000, 0xa120ffff]
[src/monitor/monitor.c:63 load_img] The image is /home/dracacys/ics2023/am-kernels/benchmarks/microbench/build/microbench-riscv32-nemu.bin, size = 26664
[src/monitor/monitor.c:31 welcome] Trace: OFF
[src/monitor/monitor.c:35 welcome] Build time: 14:36:51, Nov  4 2023
Welcome to riscv32-NEMU!
For help, type "help"
(nemu) c
Empty mainargs. Use "ref" by default
======= Running MicroBench [input *ref*] =======
[qsort] Quick sort: * Passed.
  min time: 914.150 ms [481]
[queen] Queen placement: * Passed.
  min time: 1803.982 ms [225]
[bf] Brainf**k interpreter: * Passed.
  min time: 10810.694 ms [155]
[fib] Fibonacci number: * Passed.
  min time: 19521.962 ms [103]
[sieve] Eratosthenes sieve: * Passed.
  min time: 21515.022 ms [161]
[15pz] A* 15-puzzle search: * Passed.
  min time: 2545.936 ms [210]
[dinic] Dinic's maxflow algorithm: * Passed.
  min time: 3070.665 ms [266]
[lzip] Lzip compression: * Passed.
  min time: 3175.710 ms [213]
[ssort] Suffix sort: * Passed.
  min time: 1191.945 ms [335]
[md5] MD5 digest: * Passed.
  min time: 18353.353 ms [82]
================================================
MicroBench PASS        223 Marks
                 vs. 100000 Marks (i9-9900K @ 3.60GHz)
Scored time: 82903.419 ms
Total  time: 95700.796 ms
[src/cpu/cpu-exec.c:157 cpu_exec] nemu: HIT GOOD TRAP at pc = 0x80004e24
[src/cpu/cpu-exec.c:124 statistic] host time spent = 95,701,634 us
[src/cpu/cpu-exec.c:125 statistic] total guest instructions = 1,866,996,741
[src/cpu/cpu-exec.c:126 statistic] simulation frequency = 19,508,514 inst/s
(nemu) q
make[1]: Leaving directory '/home/dracacys/ics2023/nemu'
dracacys@dracacys-virtual-machine:~/ics2023/am-kernels/benchmarks/microbench$
```

略低于指导手册的参考分数，不过电脑配置不高，还是在虚拟机里跑的，勉强能算正常吧...

键盘，显卡和声卡

外设部分都比较简单，键盘只需从指定空间读取键盘按键的状态，显卡的写函数重点关注坐标，大小，颜色即可，实现显卡后的一些展示如下：





运行超级马里奥(FPS=9, 卡成PPT了...)

声卡部分的数据结构采用队列，实现思路大致如下：

AM 端

1. 进入一个循环，该循环退出的条件是 `heap->start == heap->end`，即完成将 heap 中的数据拷贝到 stream_buf 中
2. 如果 `head != (tail + 1) % size`，则说明队列没满，一个一个拷贝， `tail = (tail + 1) % size`，start++;
3. 如果触发条件，说明队列已满，不再执行拷贝，同将 buf_overflow 标志位置为 1，之后在循环中每次检查该标志位，如果为 0 则继续拷贝操作，直至再次触发或退出循环

NEMU 端

1. 在回调函数中，先判断 AM 是否在写数据，如果不在写数据，则将数据从 stream_buf 中一个一个写入 stream，每写一次 `head = (head + 1) % size`
2. 如果 `head == tail` 说明队列已经全空，此时如果 len != 0，则接下来需要向 stream 中写入 0，防止产生噪声
3. 全部拷贝完成后，将 buf_overflow 置为 0

实现声卡后，超级马里奥的运行会接近卡死，但可以流畅地播放我下载的一首测试音乐，相关展示视频的链接如下：chicken

相关代码如下：

```
#include <am.h>
#include <nemu.h>
```

```
// 这里手动设置 audio 是否使用
// #define CONFIG_HAS_AUDIO 1

// Register in memory
#define AUDIO_FREQ_ADDR      (AUDIO_ADDR + 0x00)
#define AUDIO_CHANNELS_ADDR  (AUDIO_ADDR + 0x04)
#define AUDIO_SAMPLES_ADDR   (AUDIO_ADDR + 0x08)
#define AUDIO_SBUF_SIZE_ADDR (AUDIO_ADDR + 0x0c)
#define AUDIO_INIT_ADDR      (AUDIO_ADDR + 0x10)
#define AUDIO_COUNT_ADDR     (AUDIO_ADDR + 0x14)
#define AUDIO_HEAD_ADDR      (AUDIO_ADDR + 0x18) // head of the quene
#define AUDIO_TAIL_ADDR      (AUDIO_ADDR + 0x1C) // tail of the quene
#define AUDIO_OF_ADDR        (AUDIO_ADDR + 0x20) // 1 is buf_overflow
#define AUDIO_STATE_ADDR     (AUDIO_ADDR + 0x24) // 0 is playing, 1 is writing
#define SB_SIZE 0x10000

#ifdef CONFIG_HAS_AUDIO
static bool present = false;
#endif

void __am_audio_init() {

}

void __am_audio_config(AM_AUDIO_CONFIG_T *cfg) {
#ifdef CONFIG_HAS_AUDIO
  cfg->present = 1;
  cfg->bufsize = 0x10000;
#else
  cfg->present = 0;
#endif
}

// 声音参数设置，同时初始化物理寄存器
void __am_audio_ctrl(AM_AUDIO_CTRL_T *ctrl) {
#ifdef CONFIG_HAS_AUDIO
  outl(AUDIO_FREQ_ADDR, (uint32_t)ctrl->freq);
  outl(AUDIO_CHANNELS_ADDR, (uint32_t)ctrl->channels);
  outl(AUDIO_SAMPLES_ADDR, (uint32_t)ctrl->samples);
  outl(AUDIO_INIT_ADDR, (uint32_t)true);
  if(present == false) {
    present = true;
    outl(AUDIO_SBUF_SIZE_ADDR, SB_SIZE);
    outl(AUDIO_COUNT_ADDR, 0);
    outl(AUDIO_HEAD_ADDR, 0);
    outl(AUDIO_TAIL_ADDR, 0);
    outl(AUDIO_OF_ADDR, false);
    outl(AUDIO_STATE_ADDR, false);
  }
#endif
}

void __am_audio_status(AM_AUDIO_STATUS_T *stat) {
#ifdef CONFIG_HAS_AUDIO
  stat->count = inl(AUDIO_COUNT_ADDR);
#endif
```

```
}

void __am_audio_play(AM_AUDIO_PLAY_T *ctl) {
#ifdef CONFIG_HAS_AUDIO
  uint32_t head = 0;
  uint32_t tail = 0;
  uint32_t count = 0;
  uint8_t *start = (uint8_t *)((ctl->buf).start);
  uint8_t *end = (uint8_t *)((ctl->buf).end);
  while(1) {
    outl(AUDIO_STATE_ADDR, true);
    head = inl(AUDIO_HEAD_ADDR);
    tail = inl(AUDIO_TAIL_ADDR);
    count = inl(AUDIO_COUNT_ADDR);
    while(start != end) {
      if((tail + 1) % SB_SIZE != head) {
        outb(AUDIO_SBUF_ADDR + tail, *start);
        tail = (tail + 1) % SB_SIZE;
        start++;
        count++;
      }
      else {
        outl(AUDIO_OF_ADDR, true);
        break;
      }
    }
    outl(AUDIO_HEAD_ADDR, head);
    outl(AUDIO_TAIL_ADDR, tail);
    outl(AUDIO_COUNT_ADDR, count);
    outl(AUDIO_STATE_ADDR, false);
    if(inl(AUDIO_OF_ADDR) == 0)
      break;
    else {
      // 等待回调函数将一些数据出队列
      while(inl(AUDIO_OF_ADDR) == true);
    }
  }
  return;
#endif
}
```

```
enum {
  reg_freq,
  reg_channels,
  reg_samples,
  reg_sbuf_size,
  reg_init,
  reg_count,
  reg_head,     // head of the quene
  reg_tail,     // tail of the quene(index of the next element)
  reg_overflow, // 1 is buf_overflow
  reg_state,    // 0 is playing, 1 is writing
  nr_reg
};
```

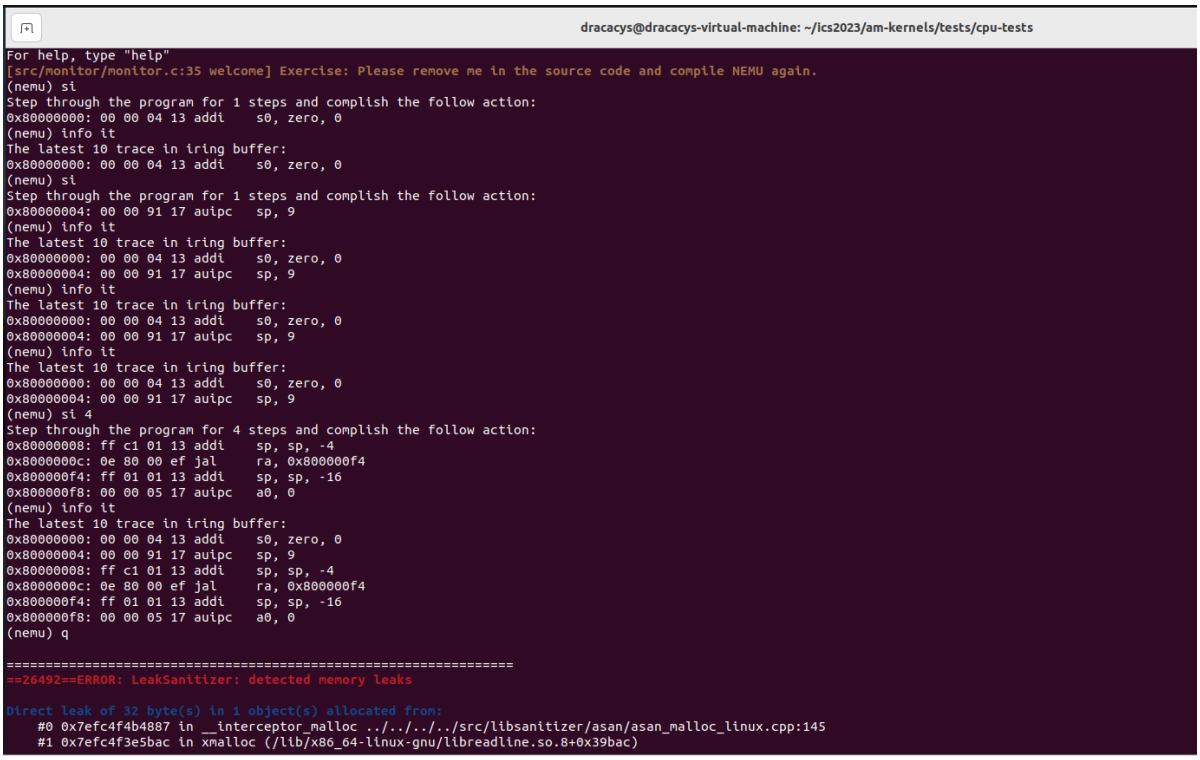
```c
static uint8_t *sbuf = NULL;
static uint32_t *audio_base = NULL;

// Audio callback fucntion
void audio_callback(void *userdata, Uint8 *stream, int len) {
  while (audio_base[reg_state] == true);
  uint32_t count = audio_base[reg_count];
  uint32_t head = audio_base[reg_head];
  uint32_t tail = audio_base[reg_tail];
  while (len > 0) {
    if (head != tail) {
      *stream = sbuf[head];
      head = (head + 1) % CONFIG_SB_SIZE;
      count--;
    }
    else {
      *stream = 0;
    }
    stream++;
    len--;
  }
  audio_base[reg_count] = count;
  audio_base[reg_head] = head;
  audio_base[reg_tail] = tail;
  audio_base[reg_overflow] = false;
  return;
}

static void audio_io_handler(uint32_t offset, int len, bool is_write) {
  if(offset != 0x10 || is_write == 0)
    return;
  else {
    // 当 init 寄存器被写时，即开始初始化
    SDL_AudioSpec s = {};
    s.format = AUDIO_S16SYS;
    s.userdata = NULL;
    s.freq = audio_base[reg_freq];
    s.channels = audio_base[reg_channels];
    s.samples = audio_base[reg_samples];
    s.callback = audio_callback;
    int ret = SDL_InitSubSystem(SDL_INIT_AUDIO);
    if (ret == 0) {
      SDL_OpenAudio(&s, NULL);
      SDL_PauseAudio(0);
      printf("Audio init success.\n");
    }
  }
}
```

心得和踩坑经历

随着实验深入，已经出现一些匪夷所思的 bug 了... 比如说同一个程序在 NEMU 上运行大约几十次会有 1 次内存泄漏报错，由于出现频率实在太低，这个问题至今未解决.



还有**更玄学更逆天**的问题，比如这一段代码：

```
void panic_test(const char *fmt, ...) {
  va_list ap;
  va_start(ap, fmt);
  if(va_arg(ap, int) != -2147483648) {
    halt(1);
  }
  va_end(ap);
}
```

分别执行下面四条语句，只有最后一条会出现 hit bad trap. 经过汇编代码的分析，发现问题出现在汇编代码上，但是 spike 不报错，说明指令执行没问题. 换成 X86 GCC 编译，不会出错，说明在交叉编译时 RISCV64 的 GCC 产生了**错误的汇编代码**. 和老师讨论初步猜测是由于 GCC 优化时对 0x80000000 做了一些特别的操作，该问题也尚未得到解决.

```
/*OK*/
int x = -2147483648;
panic_test("%d\n", x);

/*OK*/
panic_test("%d\n", (int)-2147483648);

/*OK*/
panic_test("%d\n", -2147483647);

/*Error*/
panic_test("%d\n", -2147483648);
```