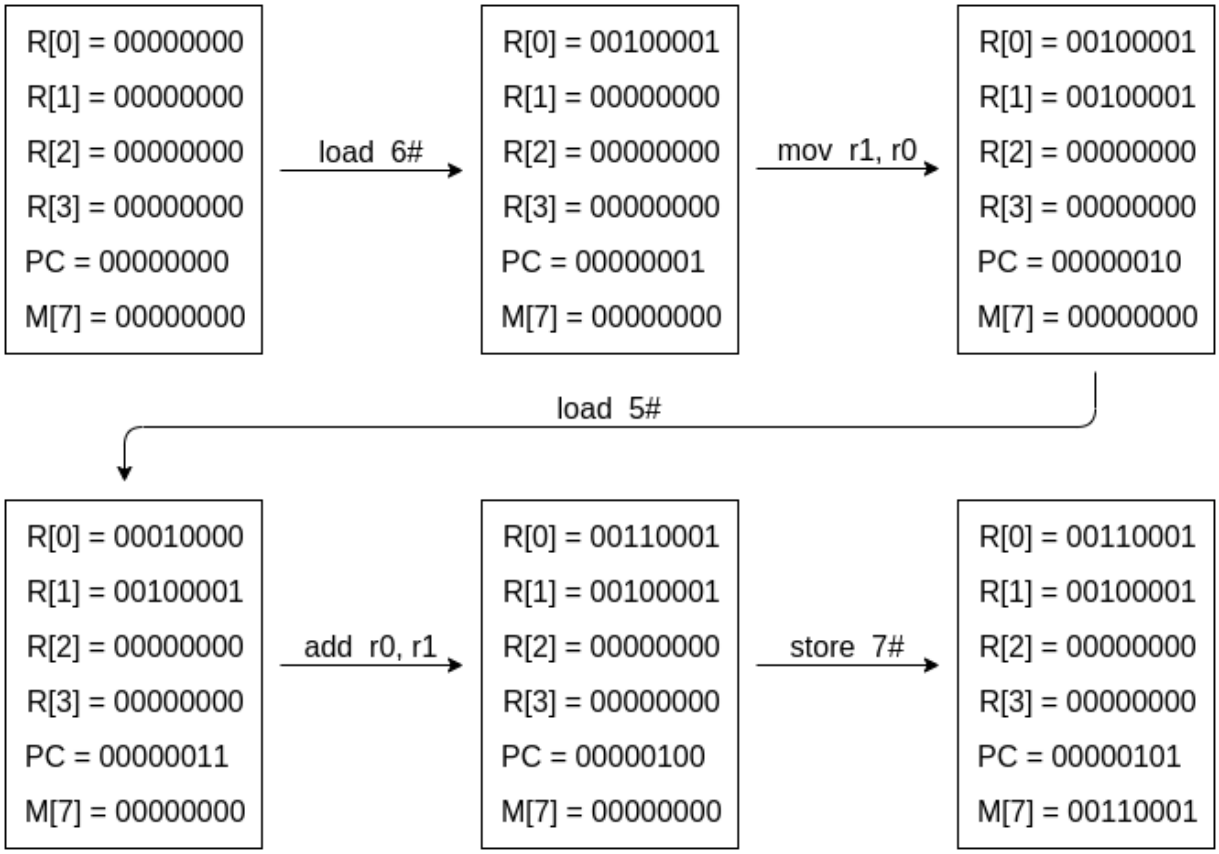
PA 2

201300035 方盛俊

1. 不停计算的机器

1.1 YEMU 上执行加法程序的状态机



1.2 YEMU 如何执行一条指令

1. 从存储器中, 根据 `pc` 从对应位置取出一条指令.
2. 根据指令 `op` 进行操作码译码.
3. 根据指令内容进行操作数译码.
4. 根据指令译码结果执行.
5. 更新 `pc`.

2. 阶段一: RTFSC

2.1 立即数背后的故事

> **Question:** mips32 和 riscv32 的指令长度只有 32 位, 因此它们不能像 x86 那样, 把 C 代码中的 32 位常数直接编码到一条指令中. 思考一下, mips32 和 riscv32应该如何解决这个问题?
>
> **Answer:** 使用 lui 和 addi 两条指令完成. 先用 lui 移动数据的高 20 位, 再用 addi 移动数据的低 12 位.

2.2 为什么不需要 rtl_muls_lo?

> **Question:** 我们没有定义用于获取有符号数乘法结果低 32 位的 RTL 基本指令 rtl_muls_lo, 你知道为什么吗?
>
> **Answer:** 我们可以使用无符号乘法结果低 32 位的 RTL 基本指令 rtl_mulu_lo 实现有符号数乘法结果低 32 位.

2.3 拓展宏

使用 `gcc -E` 等命令, 改造 Makefile 里的内容, 在 `build` 文件夹下生成了预编译过后的代码文件 `*.i`.

2.4 RTFSC

通过 RTFSC, 我逐渐理解了客户端程序每一条指令的执行过程, 分为:

1. 取指(instruction fetch, IF)
2. 译码(instruction decode, ID)
3. 执行(execute, EX)
4. 更新 PC

这些过程中, 使用到了许多的辅助函数, 我们通过编写这些辅助函数, 可以减少代码的耦合性, 可以更方便地加入需要的新指令.

2.5 添加新指令

添加一条新指令, 我们需要更改以下几个文件:

1. `isa-all-instr.h`: 我们需要在这个文件中加入指令的名称, 如 `f(jal)`.
2. `decode.h`: 我们需要在这个文件中加入指令的 `op` 和 `func3` 等标识码.

3. `instr/*.h`: 例如 `compute.h` 文件, 我们要在这个文件中加入 `addi` 指令的话, 就在里面用 RTL 语言来编写指令的功能.

可能还需要关注的几个文件:

1. `rtl-basic.h`: 定义了最为基础和常用的 RTL 语言的实现, 例如 `rtl_addi`.
2. `pseudo.h`: 定义了一些伪指令, 例如 `rtl_li`, 新的伪指令 (ISA 无关) 也在这里实现.

为了运行 `dummy`, 我实现了 `auipc, addi, jal, jalr` 这四条指令.

3. 阶段二: 基础设施与程序

3.1 DiffTest

DiffTest 真是一个非常智慧的实现. 每个人在写代码的时候, 都会有意无意地使用 DiffTest 的思想: 我在实现自己的代码的过程中, 不断与一个正确的实现进行对比, 就能及早发现 Error.

按照软件工程相关的概念, DiffTest 有助于我们在碰到 Error, 如寄存器状态和正确实现不符时, 直接转为 Failure 报错, 以达到尽快发现错误指令实现的 Fault 的效果.

在按文档配置了 DiffTest 之后, 我在 `dut.c` 文件中实现了寄存器对比的函数:

```
bool isa_difftest_checkregs(CPU_state *ref_r, vaddr_t pc) {
    if (cpu.pc != ref_r->pc) return false;
    for (int i = 0; i < 32; ++i) {
        if (cpu.gpr[i]._32 != ref_r->gpr[i]._32) return false;
    }
    return true;
}
```

便能够使用 DiffTest 的强大功能了, 在实现过程中, 我也再次惊叹于 riscv 寄存器实现的简洁, 这大大减少了我的工作量.

在这个过程中, 我还重新对 VSCode 的调试功能进行配置, 以让他能够继续调试添加了 DiffTest 功能的 NEMU 代码.

3.2 指令环形缓冲区 - iringbuf

为了快速定位出错指令位置, 及其对应上下文指令, 我实现了 iringbuf 功能.

大概过程, 就是在 `cpu-exec.c` 文件内, 加入了以下代码:

```
// iringbuf
#define MAX_IRINGBUF_LENGTH 8
typedef char buf[128];
buf iringbuf[MAX_IRINGBUF_LENGTH];
uint32_t iringbuf_count = 0;

static void trace_and_difftest(Decode *_this, vaddr_t dnpc) {
    // iringbuf
    strcpy(iringbuf[iringbuf_count % MAX_IRINGBUF_LENGTH], _this->logbuf);
    iringbuf[iringbuf_count % MAX_IRINGBUF_LENGTH][strlen(_this->logbuf)] = '
    ++iringbuf_count;
}

void cpu_exec(uint64_t n) {
    switch (nemu_state.state) {
        case NEMU_ABORT:
            // iringbuf
            printf("--------------------------\n");
            printf("[iringbuf]:\n");
            for (int i = 0; i < MAX_IRINGBUF_LENGTH - 1; ++i) {
                printf("    %s\n", iringbuf[(iringbuf_count + i) % MAX_IRINGBU
            }
            printf("--> %s\n\n\n", iringbuf[(iringbuf_count + 7) % MAX_IRINGBU
    }
}
```

最后输出了以下格式的内容:

```
--------------------------
[iringbuf]:
    0x80000050: 13 09 89 16 addi        s2, s2, 360
    0x80000054: 93 89 04 02 addi        s3, s1, 32
    0x80000058: 13 04 84 1a addi        s0, s0, 424
    0x8000005c: 03 25 04 00 lw  a0, 0(s0)
    0x80000060: 83 27 09 00 lw  a5, 0(s2)
    0x80000064: 13 04 44 00 addi        s0, s0, 4
    0x80000068: 13 55 75 00 srli        a0, a0, 7
--> 0x8000006c: 33 05 f5 40 sub a0, a0, a5
```

3.3 klib

AM 中需要我们实现的 klib 函数有这些:

1. `malloc`, `free`
2. `strlen`, `strcpy`, `strncpy`, `strcat`, `strcmp`, `strncmp`, `memset`, `memcpy`, `memmove`, `memcmp`
3. `printf`, `vsnprintf`, `snprintf`, `vsprintf`, `sprintf`

后两项已经实现, 可以通过 `string` 和 `hello-str` 样例.

但是第一项 `malloc` 和 `free` 暂未实现, 打算留到之后实现.

3.4 mtrace

实现 mtrace, 有助于让我们找出不正确的访存, 帮助我们进行 bug 的诊断.

我在 `Kconfig` 文件加入了以下选项:

```
config MTRACE
  depends on TRACE && TARGET_NATIVE_ELF && ENGINE_INTERPRETER
  bool "Enable memory tracer"
  default y

config MTRACE_READ
  depends on MTRACE
  bool "Only trace memories when the it is read"
  default n

config MTRACE_WRITE
  depends on MTRACE
  bool "Only trace memories when the it is write"
  default y
```

然后修改 `paddr.c` 的代码为:

```
word_t paddr_read(paddr_t addr, int len) {
#ifdef CONFIG_MTRACE_READ
  log_write("paddr_read(%x, %d) = %u\n", addr, len, pmem_read(addr, len));
#endif
  # ...
}

void paddr_write(paddr_t addr, int len, word_t data) {
#ifdef CONFIG_MTRACE_WRITE
  log_write("paddr_write(%x, %d, %u)\n", addr, len, data);
#endif
  # ...
}
```

最后输出如下:

```
paddr_write(80008fe0, 4, 0)
0x80000044: 23 28 21 01 sw      s2, 16(sp)
paddr_write(80008fd8, 4, 0)
0x80000048: 23 24 41 01 sw      s4, 8(sp)
0x8000004c: 13 09 44 24 addi    s2, s0, 580
paddr_write(80008fec, 4, 2147484132)
0x80000050: 23 2e 11 00 sw      ra, 28(sp)
```

3.5 ftrace

ftrace 实现起来较为麻烦, 主要麻烦在要看许多关于 ELF 文件的手册, 以获取各种信息, 再使用 `elf.h` 头文件里面的各种结构体, 即可读取函数的地址和对应名称. 经过一番查阅手册之后, 总算是完成了 ftrace.

具体来说, 实现了从 `args` 读取 `elf` 文件的功能, 并完成了 ftrace 对应的功能, 具体代码较长, 位于 `ftrace.c` 文件中, 这里就不过多赘述.

然后输出格式类似于:

```
[ftrace] call [_trm_init@0x80000108]
[ftrace]     call [main@0x80000028]
[ftrace]         call [check@0x80000010]
[ftrace]         ret
[ftrace]         call [check@0x80000010]
[ftrace]         ret
[ftrace]         call [check@0x80000010]
[ftrace]         ret
[ftrace]         call [check@0x80000010]
[ftrace]         ret
[ftrace]     ret
[ftrace] ret
```

> **Question:** 你会发现在符号表中找不到和 `add()` 函数对应的表项, 为什么会这样?
>
> **Answer:** `add()` 函数实现较为简单, 在编译过程中被优化掉了.

4. 阶段三: 输入输出

4.1 时钟

经过一番痛苦的 RTFSC, 终于理解了怎么实现时钟的功能.

只需要在 `timer.c` 文件中加入以下代码即可:

```
void __am_timer_uptime(AM_TIMER_UPTIME_T *uptime) {
  uptime->us = inl(RTC_ADDR) + ((uint64_t) inl(RTC_ADDR + 4) << 32);
}
```

4.2 键盘

键盘同理, 但是要注意的是, 我们需要用位操作来处理键盘码 (keycode) 和是否按下 (keydown) 两个数据, 具体实现如下:

```
#define KEYDOWN_MASK 0x8000

void __am_input_keybrd(AM_INPUT_KEYBRD_T *kbd) {
  uint32_t key = inl(KBD_ADDR);
  kbd->keycode = key & ~KEYDOWN_MASK;
  kbd->keydown = key & KEYDOWN_MASK;
}
```

4.3 VGA

VGA 需要非常认真地 RTFSC 才能完成, 花费了较多的时间.

NEMU 硬件方面, 我们要在 `vga.c` 文件中加入以下代码:

```
void vga_update_screen() {
  update_screen();
  vgactl_port_base[1] = false;
}

static void vga_ctl_handler(uint32_t offset, int len, bool is_write) {
  if (is_write) {
    assert(offset == 4);
    assert(vgactl_port_base[1]);
    vga_update_screen();
  } else {
    vgactl_port_base[0] = (screen_width() << 16) | screen_height();
  }
}
```

在 AM 软件方面, 我们要在 `gpu.c` 文件中加入以下代码:

```
void __am_gpu_config(AM_GPU_CONFIG_T *cfg) {
  uint32_t hw = inl(VGACTL_ADDR);
  *cfg = (AM_GPU_CONFIG_T) {
    .present = true, .has_accel = false,
    .width = (hw >> 16), .height = ((hw << 16) >> 16),
    .vmemsz = 0
  };
}

void __am_gpu_fbdraw(AM_GPU_FBDRAW_T *ctl) {
  uint32_t hw = inl(VGACTL_ADDR);
  int i, j;
  int w = (hw >> 16);
  uint32_t *fb = (uint32_t *)(uintptr_t)FB_ADDR;
  for (i = 0; i < ctl->h; ++i) {
    for (j = 0; j < ctl->w; ++j) {
      fb[(ctl->y + i) * w + (ctl->x + j)] = ((uint32_t *)ctl->pixels)[i * ctl
    }
  }
  if (ctl->sync) {
    outl(SYNC_ADDR, 1);
  }
}
```

4.4 游戏是如何运行的?

在 `main()` 函数里, 先是使用 `ioe_init()` 和 `video_init()` 初始化设备, 以实现在之后的代码中读取时钟, 读取键盘输入和更新游戏逻辑并渲染.

紧接着, 就进入一个 `while(1)` 循环, 实现游戏界面更新和信息接收的主体逻辑. 在这个循环里会做以下几件事情:

1. 通过 `io_read(AM_TIMER_UPTIME)` 读取当前时间, 并通过 `FPS` 来获取当前的帧 (frames).
    1. 调用 `io_read(AM_TIMER_UPTIME)`, 就相当于调用这个位于 AM 中的库函数, 相关代码位于 `ioe/timer.c` (通过 `ioe_read()` 分发).
    2. `ioe_read()` 会调用 AM 的 `__am_timer_uptime(uptime)` 函数, 这个函数将会往 `uptime->us` 里写入当前运行的时间.
    3. `__am_timer_uptime()` 会调用 `inl(RTC_ADDR)` 函数, 这个函数是沟通 AM 软件实现和 nemu 硬件实现的桥梁.
    4. `inl(RTC_ADDR)` 会在 nemu 的 `device/timer.c` 中通过分发调用 `rtc_io_handler()` 函数, 进一步获取时间.
    5. `rtc_io_handler()` 函数会通过 `get_time()` 函数获取系统时间, 模拟硬件实现, 并写入 `rtc_port_base[]` 中.

2. 通过当前帧数 (current) 和最新帧数 (frames) 的差, 确定需要更新多少帧, 然后通过 `game_logic_update(current)` 一帧一帧地更新 (逻辑上的, 并不会马上显示).
3. 再次进入一个循环, 以接收键盘输入.
    1. 通过 `io_read(AM_INPUT_KEYBRD)` 读取键盘输入.
    2. `ioe_read()` 调用 `__am_input_keybrd(kbd)`, 将获取到的键盘输入写入到 `kbd->keycode` 和 `kbd->keydown` 中.
    3. 进而在 nemu 中调用 `i8042_data_io_handler()` 写入按键信息.
    4. 最后在 `game.c` 中调用 `check_hit(lut[ev.keycode])` 检测按键是否命中, 命中就更新命中的游戏逻辑.
    5. 不断执行这个循环, 直到没有堆积的按键信息要处理, 就直接跳出循环.
4. 最后, 通过 `render()` 函数, 对游戏画面进行渲染.
    1. 在 `render()` 函数中, 通过 `io_write(AM_GPU_FBDRAW, ...)` 写入画面信息.
    2. `ioe_write()` 调用 `gpu.c` 目录下的 `__am_gpu_fbdraw(cfg)` 函数, 进行相应画面的写入.
    3. 最后在 `vga.c` 文件下执行 `vga_ctl_handler()` 函数, 然后在其中调用 `update_screen()` 进行画面的更新.

整个运行过程中, 涉及到了程序, AM, NEMU 的协调运行, 通过层层的封装, 保证实现接口的统一与便捷.

4.5 编译与链接 (1)

(a) 在 `static inline def_rtl(setrelop, ...)` 中去掉 `static`.

`rtl_setrelop()` 函数调用了 `interpret_relop()` 函数, 后者是一个静态函数, 不能在内联函数内部使用.

(b) 在 `static inline def_rtl(jr, ...)` 中去掉 `static`.

这种情况没有报错, 程序也能正常运行.

(c) 在 `static inline def_rtl(jr, ...)` 中去掉 `inline`.

显示 `'rtl_jr' defined but not used [-Werror=unused-function]`.

(d) 在 `static inline def_rtl(jr, ...)` 中去掉 `static` 和 `inline`.

显示函数在三处地方重复定义了. 分别是 `/src/cpu/cpu-exec.o`, `/src/isa/riscv32/instr/decode.o` 和 `/src/engine/interpreter/hostcall.o`.

综合起来推测, 因为这个 `static inline def_rtl(jr, ...)` 函数的定义和实现是放在 `rtl-basic.h` 头文件内部的 (与常规的头文件只放函数定义不同), 所以引用该头文件的代码文件都会有一份该函数的定义与实现副本.

首先, `inline` 关键字是 "建议内联" 关键字, 它会向编译器提出将这个函数内联到调用函数体内的建议, 是否采纳取决于编译器.

如果内联建议被采纳, `inline` 就会发挥作用, 被内嵌到调用函数体中, 作用域是文件内部, 此时即使不加入 `static` 也不会报错, 这就是 **(b)** 去掉 `static` 正常运行的原因.

如果内联建议未被采纳, `inline` 可以视为不存在, 这时候就需要 `static` 保证该函数是静态的, 作用域仅限文件内, 否则就会在链接时, 报函数重复定义的错误, 这个错误可以等同于 **(d)** 的情况.

如果去掉 `inline` 但不去掉 `static` 的话, 虽然不会报重复定义的错误, 但是此时函数就不是内联函数, 只要在文件中被定义了, 就一定要使用, 否则会被 `-Werror` 选项识别, 然后报函数未被使用的错误, 这就是 **(c)** 报错的原因.

4.6 编译与链接 (2)

1. 在 `common.h` 加入 `volatile static int dummy` 之后, 重新编译后的 NEMU 含有 27 个 dummy 变量的实体. 我使用命令 `find . | grep '\.i' | xargs grep "volatile static int dummy;" | wc -l` 统计经过预编译之后的文件. 最后结果显示是 27.
2. 在 `debug.h` 加入 `volatile static int dummy` 之后, 重新编译后的 NEMU 含有 54 个 dummy 变量的实体. 我使用命令 `find . | grep '\.i' | xargs grep "volatile static int dummy;" | wc -l` 统计经过预编译之后的文件. 最后结果显示是 54. 原因是这 27 个文件都引入了 `common.h` 和 `debug.h` 头文件, 最后就成了 27 * 2 = 54 个重复的 `dummy` 声明.
3. 加入了初始化之后, 编译时会重复定义的错误. 因为, 如果不加初始化, 编译器会将 `volatile static int dummy` 视作声明, 而不是定义, 声明是可以重复的, 而定义是不可重复的, 所以在编译环节就报出了错误. 如果我将两处初始化删除掉一处, 就能正确运行了.

4.7 了解 Makefile

通过分析 Makefile 代码, 我们得到了以下引用关系:

1. `am-kernels/kernels/hello/Makefile`
2. `abstract-machine/Makefile`
3. `abstract-machine/scripts/riscv32-nemu.mk`
4. `abstract-machine/scripts/platform/nemu.mk`

最后我们在 `abstract-machine/scripts/platform/nemu.mk` 找到了第一个目标 (也是一个伪目标): `image`. 而 `image` 的前置目标为 `$(IMAGE).elf`.

对于 `$(IMAGE).elf` 来说, 它的前置目标有三个: `$(OBJS) am $(LIBS)`, 我们分别来分析.

对于 `$(OBJS)` 目标, 这是一个变量, 我们在 Makefile 中加入一句 `$(info Objects: $(OBJS))` 并 `make -n` 便可知, 它是 `am-kernels/kernels/hello/build/riscv32-nemu/hello.o`. 同理可以分析 `$(LIBS)`.

三个目标 `$(OBJS) am $(LIBS)` 统一起来, 可以知道

- `am-kernels/kernels/hello/build/riscv32-nemu/hello.o`
  - 通过 `$(DST_DIR)/%.o: %.c` 使用 `hello.c` 生成 `hello.o` 文件.
- `am`
  - 即执行了 `make abstract-machine/am/Makefile`
- `$(LIBS)`
  - 执行 `make -s -C abstract-machine/am archive`
  - 执行 `make -s -C abstract-machine/klib archive`
  - 这两者又会循环调用 `abstract-machine/Makefile`, 对 `am` 和 `klib` 进行编译.

统一起来, 我们知道有以下步骤:

1. 根据 `hello.c` 生成 `hello.o`, 各种头文件, 是通过 `CFLAGS` 变量设定的.
2. 执行 `make -s -C abstract-machine/am archive`, 使用 `riscv64-linux-gnu-gcc` 编译生成 `trm.o` 和各种其他 `am` 设备文件.
3. 执行 `make -s -C /home/orangex4/ics2021/abstract-machine/klib archive`, 使用 `riscv64-linux-gnu-gcc` 编译生成各种 `klib` 相关文件, 如 `int64.o`, `string.o` 和 `stdio.o` 等.
4. 使用链接器 `riscv64-linux-gnu-ld` 将对应文件链接起来, 生成 `hello-riscv32-meu.elf` 文件.
5. 使用 `riscv64-linux-gnu-objdump` 反汇编可执行文件 `hello-riscv32-meu.elf`, 生成反汇编结果 `hello-riscv32-nemu.txt`.
6. 使用 `riscv64-linux-gnu-objcopy` 将可执行镜像部分拆分出来, 生成 `hello-riscv32-nemu.bin` 文件, 供 NEMU 运行.

以上就是执行 `make ARCH=riscv32-nemu` 后, `make` 程序如何组织 `.c` 和 `.h` 文件, 最终生成可执行文件 `hello-$ISA-nemu.elf` 的过程.