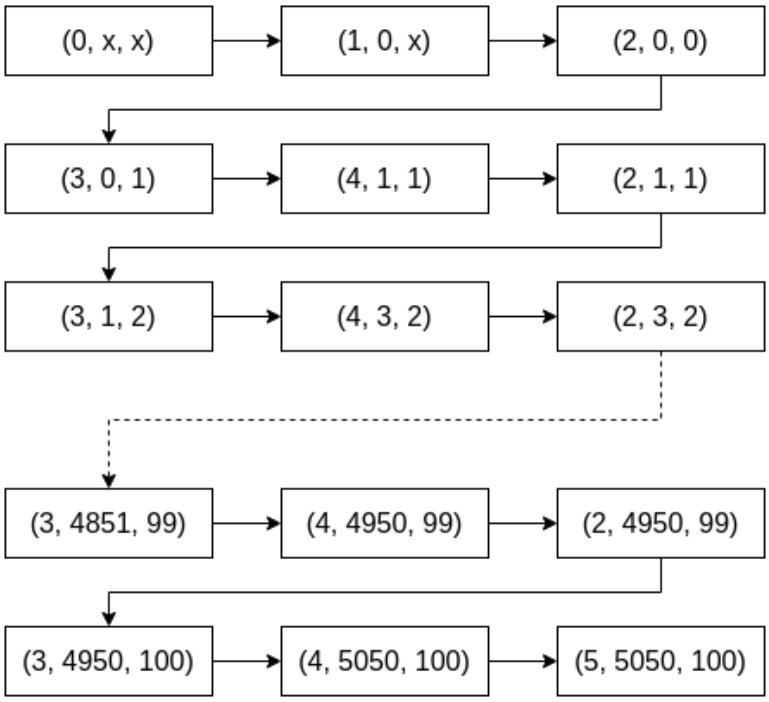
PA 1

201300035 方盛俊

1. RTFSC

1.1 从状态机视角理解程序运行



1.2 实现 x86 的寄存器结构体

使用匿名 union, 修改后的代码如下:

```
typedef struct {
    union {
        struct {
            union {
                uint32_t _32;
                uint16_t _16;
                uint8_t _8[2];
            };
        } gpr[8];

        /* Do NOT change the order of the GPRs' definitions. */

        /* In NEMU, rtlreg_t is exactly uint32_t. This makes RTL instructions
         * in PA2 able to directly access these registers.
         */
        struct {
            rtlreg_t eax, ecx, edx, ebx, esp, ebp, esi, edi;
        };
    };

    vaddr_t pc;
} x86_CPU_state;
```

1.3 尝试使用 vscode + gdb 进行调试并完善相关配置

直接使用 gdb 还是感觉太麻烦了, 执行需要各种各样的命令, 而且代码查看和跳转也不方便.

这种时候还是要有像正常 IDE 那样的调试界面才快乐啊. 所以我把目光投向了 VSCode 的调试功能. VSCode 既有 IDE 的强大, 有着可以媲美 Vim 的强大功能, 又有着对命令行功能的良好适配. 其中 VSCode 对 C 和 C++ 语言的调试功能正是通过 gdb 实现的, 所以我们可以很容易地进行一些配置, 使得 VSCode 的调试功能适配 NEMU 项目.

我们只需要添加两个文件, 第一个是 `tasks.json`:

```
{
    "version": "2.0.0",
    "options": {
        "cwd": "${workspaceRoot}/nemu" // $ nemu 路径
    },
    "tasks": [
        {
            "label": "make", // 任务名称, 与 launch.json 的 preLaunchTask 相对应
            "command": "make",
            "args": [
                "vscode" // 对应 `make vscode`, 类似于 `make gdb`, 但实际上用的是 vscode 的调试
            ],
            "type":"shell"
        },
        {
            "label": "kill",
            "command": "killall", // 即执行 killall -9 x-terminal-emulator 杀死进程
            "args": [
                "-9",
                "x-terminal-emulator"
            ],
            "type":"shell"
        }
    ]
}
```

其对应着命令 `make vscode`, 即在执行前, 重新生成可执行程序, 然后使用 VSCode 调试, 结束调试之后执行 `killall -9 x-terminal-emulator` 杀死进程.

这里是我修改过的 Makefile (修改 `native.mk` 文件), 以便能正常地跟踪我的 VSCode 调试, 其中 `script/native.mk` 的修改为 (即去除默认的调试, 使用 VSCode 的调试功能进行调试运行):

```
--- .PHONY: run gdb run-env clean-tools clean-all $(clean-tools)
+++ .PHONY: run gdb vscode run-env clean-tools clean-all $(clean-tools)

+++ vscode: run-env
+++     $(call git_commit, "gdb")
```

第二个是 `launch.json`:

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Debug nemu", // 配置名称, 将会在启动配置的下拉菜单中显示
            "type": "cppdbg", // 配置类型, 这里只能为cppdbg
            "request": "launch", // 请求配置类型, 可以为 launch (启动) 或 attach (附加)
            "program": "${workspaceRoot}/nemu/build/riscv32-nemu-interpreter", // 将要进行调
            "stopAtEntry": false, // 设为 true 时程序将暂停在程序入口处
            "cwd": "${workspaceRoot}/nemu", // 调试程序时的工作目录
            "environment": [], // 环境变量
            "externalConsole": true, // 调试时是否显示控制台窗口, 一般设置为 true 显示控制台
            "MIMode": "gdb", // 指定连接的调试器, 可以为 gdb 或 lldb
            "preLaunchTask": "make", // 调试会话开始前执行的任务, 一般为编译程序. 与 tasks.json 的
            "postDebugTask": "kill", // 在退出之后, 执行 "killall -9 x-terminal-emulator" 杀死
        }
    ]
}
```

这个是使用 VSCode 进行调试的关键, 它先使用 `preLaunchTask` 执行了可执行程序的生成指令 `make vscode`, 然后再使用 gdb 运行并附加到 `/nemu/build/riscv32-nemu-interpreter` 这个编译生成的文件.

相当于命令 `gdb ./nemu/build/riscv32-nemu-interpreter`.

最后, 我们要注意, 使用 `make nemuconfig`, 然后开启选项:

```
Build Options
  [*] Enable debug information
```

这样就配置完成了, 我们可以很简单地在 VSCode 中配置断点, 并按下 F5 运行程序开始调试. 这不比 gdb 香? (不是

1.4 一个程序从哪里开始执行呢?

> **Question:** 一个程序从哪里开始执行呢?
>
> **Answer:** 如果只是按照我们在程序设计中学过的课程来说, 似乎就是从 `main()` 函数开始执行. 但实际上, 就算只是用我们已经学过的知识, 都可以知道不可能是从 `main()` 函数开始执行. 例如, **全局变量的初始化**应该放在什么地方呢? 一个程序从哪里开始执行, 我也许得等到学完了编译原理才能知道.

1.5 为什么全都是函数?

> **Question:** 阅读 `init_monitor()` 函数的代码, 你会发现里面全部都是函数调用. 按道理, 把相应的函数体在 `init_monitor()` 中展开也不影响代码的正确性. 相比之下, 在这里使用函数有什么好处呢?
>
> **Answer:** 好处是可读性更强, 并且可以将不同功能的代码分散到不同的文件中, 这样便更有条理, 不至于导致一个文件上千上万行的惨状.

1.6 参数的处理过程

> **Question:** `parse_args()` 的参数是从哪里来的呢?

> **Answer:** 从启动该程序时附带的参数中来. 例如我们常见的命令 `man xxx`, 其中 `man` 其实是一个程序, 我们要求其中这个叫 `man` 的程序, 而后面紧跟的 `xxx` 便是参数. 类似的, `parse_args()` 的参数就是从这里来的.

1.7 "reg_test()" 是如何测试你的实现的?

> **Question:** 阅读 `reg_test()` 的代码, 思考代码中的 `assert()` 条件是根据什么写出来的.
>
> **Answer:** `sample[R_EAX] & 0xff` 这种写法, 是取出 EAX 寄存器中的低 8 位, 于是 `assert(reg_b(R_AL) == (sample[R_EAX] & 0xff));` 就是判断两者是否一致, 其他的也类似.

1.8 究竟要执行多久?

> **Question:** 在 `cmd_c()` 函数中, 调用 `cpu_exec()` 的时候传入了参数 `-1`, 你知道这是什么意思吗?
>
> **Answer:** 仔细观察代码不难发现, `cpu_exec()` 定义为 `void cpu_exec(uint64_t)`, 即参数是无符号的. 于是我们传入的 `-1`, 实际上是变成了一个最大的数, 这样就能让程序一直运行, 直到退出为止.

1.9 潜在的威胁

> **Question:** "调用 `cpu_exec()` 的时候传入了参数-1", 这一做法属于未定义行为吗? 请查阅 C99 手册确认你的想法.
>
> **Answer:** 并不属于. `-1` 是有符号数, 有符号数转为无符号数的时候, 均为直接转换, `-1` 便会转为一个很大的无符号数, 这并不属于未定义行为.

1.10 谁来指示程序的结束?

> **Question:** 在程序设计课上老师告诉你, 当程序执行到 `main()` 函数返回处的时候, 程序就退出了, 你对此深信不疑. 但你是否怀疑过, 凭什么程序执行到 `main()` 函数的返回处就结束了? 如果有人告诉你, 程序设计课上老师的说法是错的, 你有办法来证明/反驳吗? 如果你对此感兴趣, 请在互联网上搜索相关内容.
>
> **Answer:** 并没有直接退出, 还需要释放各种资源, 如全局变量, 打开的文件, 设备等.

1.11 优美地退出

> **Question:** 之间键入 `q` 后退出, 会发现终端出现了错误信息 `make: *** [/home/orangex4/ics2021/nemu/scripts/native.mk:23: run] 错误 1`, 该怎么解决这个问题?
>
> **Answer:** 经过调试发现, 是最后 `main()` 调用 `is_exit_status_bad()` 的时候, `utils/state.c` 中的代码 `NEMUState nemu_state = { .state = NEMU_STOP };` 出现了错误, 这里应该是 "退出" 而不是 "停止", 因此, 修改为 `NEMUState nemu_state = { .state = NEMU_QUIT };` 之后, 代码就恢复了正常, 退出时也就不会报错了.

2. 阶段一: 基础设施

2.1 实现基本命令

基本命令实现起来没有太大的困难, 文档描述得十分详细, 只要先初步了解大致的代码架构, 就能往里面填充代码.

单步执行命令 `si [N]` 比较简单. 其最核心的代码, 就是调用函数 `cpu_exec(n)`. 理解了这一层, 就没有什么困难的地方了. 其他细节的地方, 例如 `atoi` 函数将字符串转为整数, 还有一些错误处理.

打印寄存器状态的命令 `info r` 稍微麻烦一点. 主要麻烦的地方在于, 不同的 ISA 有着不同的寄存器结构. 但是这也麻烦不到哪里去, 特别是对于 riscv32 架构, 打印起来寄存器信息相对比较简单.

在实现打印寄存器命令的时候, 我遇到的主要困难是不太清楚寄存器的数据的存储位置. 在认真阅读了代码之后, 我逐渐了解到, 只需要在 `isa_reg_display()` 函数中调用全局变量 `cpu`, 就能获取到寄存器信息. 对于 riscv32 架构来说, 就是 `cpu.gpr[i]._32`.

扫描内存的指令 `x N EXPR` 实现起来也不困难, 并且文档中也详细地写出了应该如何读取内存数据:

> 内存通过在 `nemu/src/memory/paddr.c` 中定义的大数组 `pmem` 来模拟. 在客户程序运行的过程中, 总是使用 `vaddr_read()` 和 `vaddr_write()` (在 `nemu/include/memory/vaddr.h` 中定义) 来访问模拟的内存. `vaddr`, `paddr` 分别代表虚拟地址和物理地址.

要注意的就是, 需要引入头文件 `include <memory/vaddr.h>` 才能使用 `vaddr_read()` 函数.

2.2 如何测试字符串处理函数?

> **Question:** 你可能会抑制不住编码的冲动: 与其RTFM, 还不如自己写. 如果真是这样, 你可以考虑一下, 你会如何测试自己编写的字符串处理函数?
>
> **Answer:** 使用简单的单元测试, 手动或自动构造测试样例, 并使用类似 `Assert()` 之类的方法进行测试.

2.3 实现单步执行, 打印寄存器, 扫描内存

> **Question:** 为了查看单步执行的效果, 你可以把nemu/include/common.h中的DEBUG宏打开, 这样以后NEMU会把单步执行的指令打印出来(这里面埋了一些坑, 建议你RTFSC看看指令是在哪里被打印的).
>
> **Answer:** DEBUG 宏已经是默认打开了的, 所以会有相应的显示. 其中的坑是操作数或寄存器与目标寄存器的顺序不对, 如本应显示为 "lui t0,0x80000000" 的指令显示成了 "lui 0x80000000,t0". 经过调试, 发现具体的输出指令在文件 `nemu/include/cpu/decode.h` 对应的宏中. 具体的修改方式如下:

```
--- #define print_asm_template2(instr) \
---   print_asm(str(instr) "%c %s,%s", suffix_char(id_dest->width), id_src1->str, id_dest->str)
--- #define print_asm_template3(instr) \
---   print_asm(str(instr) "%c %s,%s,%s", suffix_char(id_dest->width), id_src1->str, id_src2->str,

+++ #define print_asm_template2(instr) \
+++   print_asm(str(instr) "%c %s,%s", suffix_char(id_dest->width), id_dest->str, id_src1->str)
+++ #define print_asm_template3(instr) \
+++   print_asm(str(instr) "%c %s,%s,%s", suffix_char(id_dest->width), id_dest->str, id_src1->str,
```

3. 阶段二: 表达式求值

表达式求值部分相对麻烦一些, 特别是考虑到各种粗枝末节之后.

首先要完成表达式求值最基本的功能, 基础数学表达式的**词法分析**和**语法分析**.

3.1 词法分析

最简单的词法分析实现起来并不算太困难. 特别是允许使用正则表达式来进行词法, 让整个过程轻松了许多. 注意好转译, 加减乘除括号这些符号识别起来几乎没有任何难度；真正麻烦的是**数字**的识别.

最简单的识别数字的表达式应该是 `[0-9]+`, 匹配形如 `12345`, `01`, `100` 这样的**十进制数字**. (感谢 RegExr 让我可以很方便地书写正则表达式).

想要判断一个 "-" 是减号还是负号是一件麻烦的事, 古老的 `regcomp` 既不支持 `\d`, 也不支持 `negative lookbehind`. 那对于负数就没什么办法了, 只能用 `[0-9]+` 匹配出数字, 然后再根据前面有无数字, 从而判断是负号还是减号. 负号对应的是**单目操作符**, 减号对应的是**双目操作符**. 就这样, 解决了负数的问题.

还有一个血的教训: 使用 `strncpy(dest, str, n)` 进行字符串截取的时候, 一定要在后面加上一句 `dest[n] = '\0'`, 不然字符串没有结束符, 就会导致各种各样的奇怪 bug.

3.2 语法分析

实现数学表达式的求值, 我们可以用递归求值的方法. 文档中已经解释了大部分的代码和思路, 只需要简单地填入代码即可. 相当于我们只需要写 `check_parentheses(p, q)` 和 `get_op(p, q)` 这两个函数即可.

并且由于负数的引进, 增加了**单目运算符**, 因而代码也需要相应的更改.

3.3 单元测试

在实现表达式求值算法时, 很容易在细节的地方出 bug, 这时候, **单元测试**的重要性以及必要性就体现了出来. 于是我又加入了一个简单的命令 `test`, 并优化了 `Log` 的输出与开关, 就可以进行简单的单元测试了.

并且, 做好了随机测试算法, 生成了数千个样本, 在改了几个 bugs 之后, 全部样本都通过了.

```
Welcome to riscv32-NEMU!
For help, type "help"
(nemu) test r
Random Test [eval]
[##################################################][842 / 842]
All tests pass.
(nemu)
```

(样本保存在 `nemu/tools/gen-expr/input.txt` 中.)

3.4 为什么 "printf()" 的输出要换行?

> **Question:** 为什么 `printf()` 的输出要换行? 如果不换行, 可能会发生什么?
>
> **Answer:** 不换行的话, 输出内容会挤成一堆, 难以观看.

3.5 表达式生成器如何获得C程序的打印结果?

> **Question:** 表达式生成器如何获得C程序的打印结果?

> **Answer:** 使用 `popen()` 打开一个可执行文件之后, 可以使用 `fscanf()` 获取程序的输出内容. 其实这也恰好吻合了 Linux 的哲学: 一切皆文件. 我们可以用类似于操作文件读写的方式, 对可执行程序的输入输出进行读写.

3.6 为什么要使用无符号类型?

> **Question:** 我们在表达式求值中约定, 所有运算都是无符号运算. 你知道为什么要这样约定吗? 如果进行有符号运算, 有可能会发生什么问题?
>
> **Answer:** 如果进行有符号数的运算, 就可能会出现 "减号 + 负数" 的情况, 类似于 "1 - -1", 这时候 C 语言的编译器会认为 "--1" 是自减操作, 便会报错.

3.7 除 0 的确切行为

> **Question:** 如果生成的表达式有除0行为, 你编写的表达式生成器的行为又会怎么样呢?
>
> **Answer:** 如果有明显的除零行为, 例如 `1 / 0`, 编译器编译的时候就会报错, 即 `ret = system("gcc /tmp/.code.c -o /tmp/.expr")` 执行后 `ret` 就不等于零了. 但是对于复杂一点的除零行为, 例如 `1 / ( 0 / 1 )`, 编译器是检测不出来的, 这时候只能等到运行时才能报错, 即使用 `ret = pclose(fp)` 获取退出码, `ret` 不等于零, 代表未正常退出, 就说明表达式除零了, 得舍去.

4. 阶段三: 监视点

4.1 扩展表达式求值的功能: 用栈 (Stack) 计算表达式

如果我还是沿用阶段二的那种递归式求表达式的值的方法的话, `get_op()` 操作符优先级的问题会弄得我焦头烂额, 根本无从下手.

所以我决定用栈 (Stack) 重构表达式求值的部分, 这样在后续添加新的操作符会非常方便.

用栈计算表达式部分的思想参考了这篇文章: Infix to Postfix/Prefix converter.

大概思路如下:

1. Scan input string from left to right character by character.
2. If the character is an operand, put it into output stack.
   If the character is an operator and operator's stack is empty, push operator into operators' stack.
3. If the operator's stack is not empty, there may be following possibilities.
4. If the precedence of scanned operator is greater than the top most operator of operator's stack, push this operator into operand's stack.
5. If the precedence of scanned operator is less than or equal to the top most operator of operator's stack, pop the operators from operand's stack until we find a low precedence operator than the scanned character. Never pop out ('(') or (')') whatever may be the precedence level of scanned character.
6. If the character is opening round bracket ('('), push it into operator's stack.
7. If the character is closing round bracket (')'), pop out operators from operator's stack until we find an opening bracket ('(').
8. Now pop out all the remaining operators from the operator's stack and push into output stack.

我实现了两个数据结构: `Stack` 和 `Map`, 存放在 `struct.c` 中. 使用这种方法重构之后, 我就可以很方便地添加各种操作符了.

最终, 我实现的表达式求值的功能如下:

```
static struct rule {
    char *regex;
    int token_type;
} rules[] = {

    {" +", TK_NOTYPE},              // spaces
    {"\\$[a-z]*[0-9]*", TK_REG},    // $reg
    {"0[xX][0-9a-fA-F]+", TK_HEX},  // hex number
    {"[0-9]+", TK_NUMBER},          // number
    {"\\!", '!'},                   // not
    {"\\~", '~'},                   // bitwise not
    {"\\+", '+'},                   // plus
    {"-", '-'},                     // minus or negative
    {"\\*", '*'},                   // multiply
    {"\\/", '/'},                   // divide
    {"\\\%", '%'},                  // mod
    {"\\(", '('},                   // left bracket
    {"\\)", ')'},                   // right bracket
    {"<<", TK_LS},                  // left shift
    {">>", TK_RS},                  // right shift
    {">=", TK_GTE},                 // greater than or equal to
    {"<=", TK_LTE},                 // less than or equal to
    {">", TK_GT},                   // greater than
    {"<", TK_LT},                   // less than
    {"==", TK_EQ},                  // equal
    {"!=", TK_NEQ},                 // not equal
    {"&&", TK_AND},                 // and
    {"\\|\\|", TK_OR},              // or
    {"&", TK_BAND},                 // bitwise and
    {"\\|", TK_BOR},                // bitwise or
    {"\\^", TK_XOR},                // bitwise xor
};
```

4.2 实现监视点

为了更好地操作链表, 不容易出错, 我给每个链表设定了一个 `sentinel`, 即一个不存储值的空头部, 有了它, 就不用担心删除完所有元素之后, `head` 变为空指针, 或者在删除头部或更新头部的时候, 就不用担心 `head` 的指针操作和各种变化.

我还给 `watchpoint` 添加了 `enable` 和 `disable` 的功能, 以防有时候想暂时停止这个监视点的检测, 就需要将其整个删除的情况.

接下来要做的就是监视点功能的具体实现. 我在 `watchpoint.c` 中又实现了一个 `is_stop()` 函数, 每次 `debug_hook()` 末尾都会调用它. 实现了前面的表达式求值和监视点的链表操作之后, 这部分就较为简单.

4.3 温故而知新: static 的含义

> **Question:** 框架代码中定义 `wp_pool` 等变量的时候使用了关键字 `static`, `static` 在此处的含义是什么? 为什么要在此处使用它?

> **Answer:** `static` 是 "静态" 的意思, 在此处指静态全局变量. 静态全局变量与全局变量的区别在于, 如果程序包含多个文件的话, 它作用于定义它的文件里, 不能作用到其它文件里, 即被 `static` 关键字修饰过的变量具有文件作用域. 这样即使两个不同的源文件都定义了相同名字的静态全局变量, 它们也是不同的变量.

4.4 如何提高断点的效率

> **Question:** 如果你在运行稍大一些的程序 (如 microbench) 的时候使用断点, 你会发现设置断点之后会明显地降低NEMU执行程序的效率. 思考一下这是为什么? 有什么方法解决这个问题吗?
>
> **Answer:** 用监视点的方法设定了断点之后, 每执行一步都要中断, 然后遍历所有的监视点, 观察监视的值是否有变化, 这样是非常耗时间的. 解决这个问题的方法有两种, 第一种是用硬件断点, 但是我们做的是模拟器, 用不了硬件断点, 所以这个方法搁置; 第二种方法是用类 `int3` 指令断点进行 "偷天换日". `int3` 断点是 x86 系统的一种中断指令, 每当执行到这一条指令的时候, 程序就会终止, 寻找附加到这个程序的调试器, 然后将控制权交给调试器. 于是, 在 x86 中下软件断点, 就是就是将对应的指令换成 x86 指令, 在调试器让程序继续执行的时候, 再换回来, 继续运行. 使用这种 "让程序自主报告" 的方式, 速度肯定是优于 "调试器不断进行检查" 的.

4.5 一点也不能长?

> **Question:** x86 的 `int3` 指令不带任何操作数, 操作码为 1 个字节, 因此指令的长度是1个字节. 这是必须的吗? 假设有一种 x86 体系结构的变种 my-x86, 除了 `int3` 指令的长度变成了2个字节之外, 其余指令和 x86 相同. 在 my-x86 中, 上述文章中的断点机制还可以正常工作吗? 为什么?
>
> **Answer:** 不能. x86 中除了 `int3` 指令外, 也还有很多其他长度为 1 字节的指令, 例如 "空操作" `nop` 指令. 如果我将 1 字节的 `nop` 转为 2 字节的 `int3` 指令的话, 就会将 `nop` 指令的下一条指令 (姑且称为 `next` 指令) 给覆盖掉, 使之不能正常运行. 如果此时有一个跳转指令跳转到 `next` 指令处的话, 既会出现逻辑错误, 也不会触发断点停止, 继续执行, 最后导致程序崩溃.

4.6 随心所欲的断点

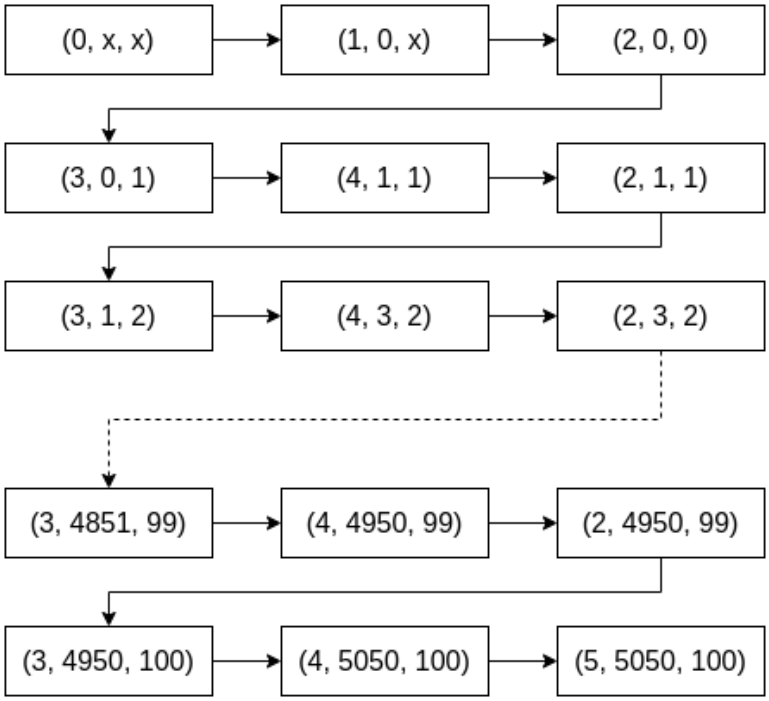
> **Question:** 如果把断点设置在指令的非首字节 (中间或末尾), 会发生什么? 你可以在 GDB 中尝试一下, 然后思考并解释其中的缘由.
>
> **Answer:** 一般不会停止, 会报错或者继续错误地执行. 因为对于多字节的指令, 在中间或末尾替换成 `int3` 指令, 也只是修改了该指令的一小部分, 这个指令不会被识别成 `int3` 指令, 而是会被识别成一个错误的多字节指令.

4.7 NEMU的前世今生

> **Question:** 模拟器 (Emulator) 和调试器 (Debugger) 有什么不同? 更具体地, 和NEMU相比, GDB到底是如何调试程序的?
>
> **Answer:** 模拟器 (Emulator) 是在操作系统上多加了一层抽象层, 然后要调试的程序跑在这个抽象层上, 所以要调试的程序所用的指令集可以和操作系统的指令集不一样; 而调试器 (Debugger, 这里特指 C 和 C++ 这类将程序编译成系统对应机器码的语言的调试器) 是依托于系统提供的功能, 在操作系统中运行这个要调试的程序, 然后用操作系统自带的调试器 API 进行调试, 如 `int3` 断点, 硬件断点和中断 flag 这类接口实现的调试器, 一个显著特点就是, 要调试的程序所用的指令集和操作系统的指令集一定要一致.

5. 必答题

5.1 程序是个状态机



5.2 理解基础设施

∵ 编译 500 次 NEMU 才能完成 PA, 其中 90% 的次数是用来调试

∴ 调试的次数为 $500 \times 90\% = 400$ 次

∵ 假设每一次调试是为了解决一个 bug, 每个 bug 要分析 20 个信息, 分析一个信息要 30 秒, 即 0.5 分钟

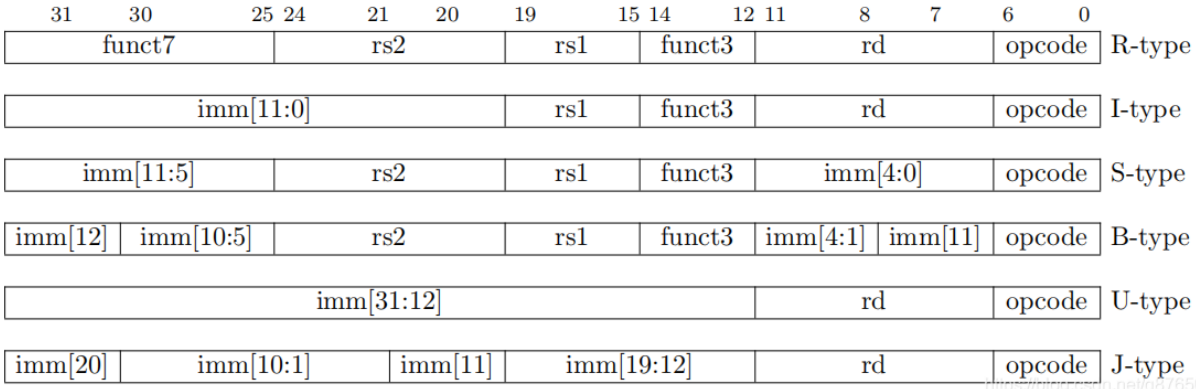
∴ 一共要在调试上花费 $400 \times 20 \times 0.5 = 4000$ 分钟, 即 $4000/60 = 66.7$ 个小时

如果实现了简易调试器, 所用调试时间就降为了原来的 $\frac{1}{3}$, 即节省了 $(1 - \frac{1}{3}) \times 66.7 = 44.4$ 个小时

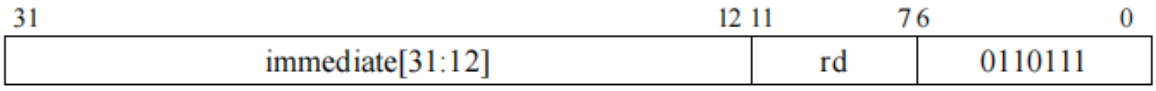
5.3 RTFM

(a) riscv32 有哪几种指令格式?

基本上有这六种指令格式.

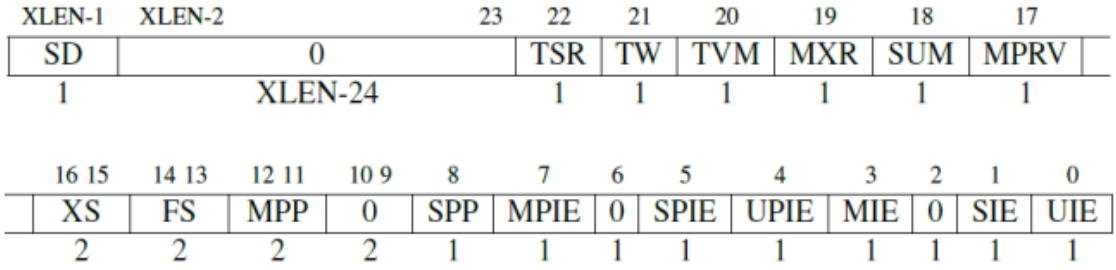


(b) LUI 指令的行为是什么?



将符号位扩展的 20 位立即数 `immediate` 左移 12 位, 并将低 12 位置零, 写入 `x[rd]` 中.

(c) mstatus 寄存器的结构是怎么样的?



5.4 shell 命令

`nemu/` 目录下总共有 24167 行代码. 使用的命令是 `find ./nemu | grep '\.c$\|\.h$' | xargs wc -l | grep 'total' | awk '{print substr($1, 1)}'` .

如果要实现与 `pa0` 进行对比, 我们需要用 `git checkout pa0` 切换到 `pa0` 中, 然后在用上述命令统计出行数, 最后进行一个减法, 再切换回 `pa1` 即可.

对应的 sh 命令文件如下:

```
lines_count_pa1=`find . | grep '\.c$\|\.h$' | xargs wc -l | grep 'total' | awk '{print sub
pa=`git branch | grep '*' |  awk '{print substr($2, 1)}'`
git checkout pa0 >/dev/null 2>/dev/null
lines_count_pa0=`find . | grep '\.c$\|\.h$' | xargs wc -l | grep 'total' | awk '{print sub
git checkout $pa >/dev/null 2>/dev/null
expr="$lines_count_pa1 - $lines_count_pa0"
echo New lines: `expr $expr`
```

最后, 我在 `native.mk` 文件中加入了命令 `sh ../count.sh` , 就可以成功地统计新增代码行数了.

输出格式为: `New lines: 1008`

这说明, 我在 pa1 中共写了 1008 行代码.

5.5 RTFM

> -Wall: enable a set of warning, actually not all.
> -Werror: every warning is treated as an error.

`-Wall` 编译选项是指, 开启一系列的警告 (绝大部分的警告).

`-Werror` 编译选项是指, 将每个警告都视作错误, 在编译时就使其无法编译通过.