2012543-刘沿辰-PA1报告

学号：2012543
姓名：刘沿辰
专业：计算机科学与技术
指令集：RISC-V 32

实验目的

在框架代码中实现小型图灵机的simple debugger，迈出实现计算机的第一步

实验内容

- 第一阶段：实现单步执行，打印寄存器状态，扫描内存
- 第二阶段：实现算术表达式求值
- 第三阶段：实现所有要求，提交完整报告

第零阶段

vim的操作与使用

简洁&特点

vim是从vi衍生而来的一款文本编辑器，代码补全、编译及错误跳转等方便编程的功能特别丰富，在程序员中被广泛使用。不过vim官网将自己定位为程序开发工具而非文本编辑工具。

vim支持高度的自定义和简洁操作。

操作模式

命令模式
此时键入的内容会被当做指令而非字符，常用指令如下表：

按键	功能
i	进入输入模式
x	删除当前光标字符
:	切换到底线命令模式

输入模式

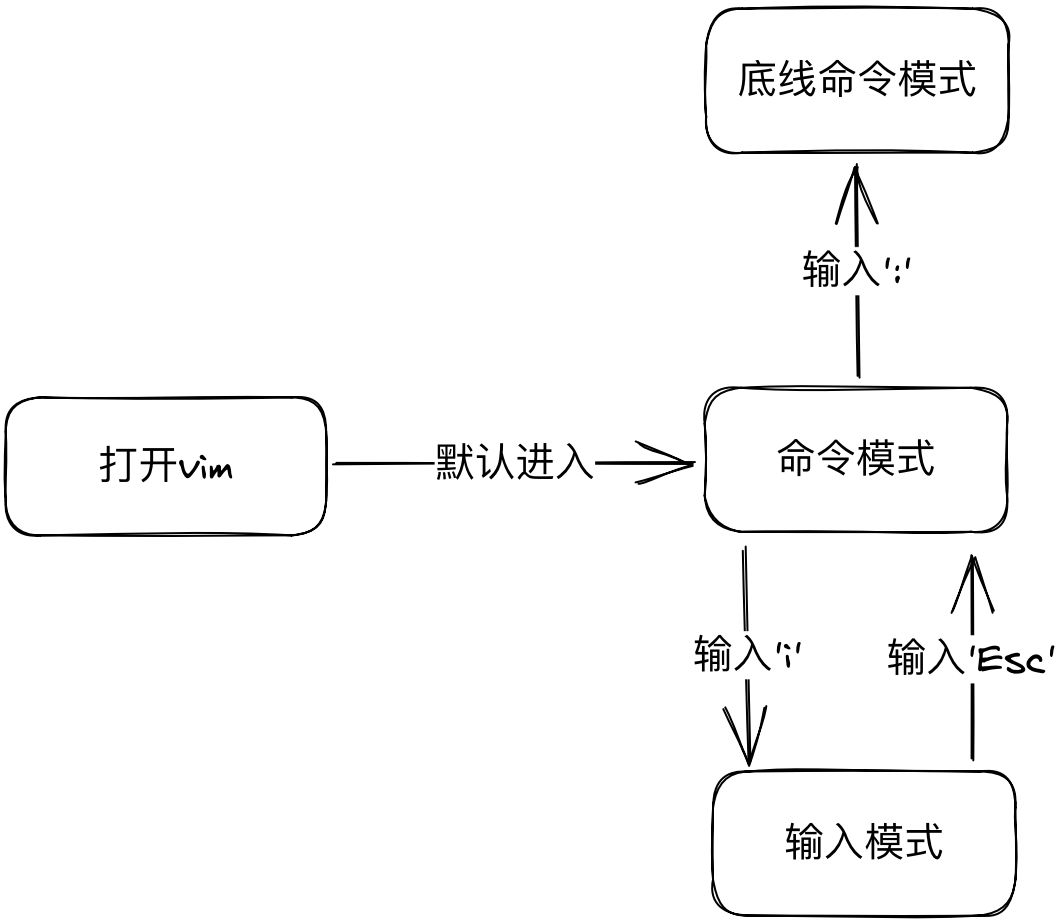
输入模式中大部分按键和常用文本编辑软件一致，特殊按键如下：

按键	功能
HOME	光标移动到行首
END	光标移动到行尾
Page Up/Down	上/下翻页
Insert	切换输入/替换模式

底线命令模式

底线命令是特殊的命令模式，常用的只有两个

按键	功能
q	退出程序
w	保存文件
!	强制执行
(不使用冒号)ZZ	等同于:wq
(不使用冒号)ZQ	不保存强制退出



常用操作

指令`vim textname.txt`：如果textname.txt不存在则创建，然后进入文件的编辑

批量注释：输入`Ctrl+v`进入块选择模式，移动光标选定要注释的行，按下大写的`I`进入行首输入`//`，按两次`Esc`，所有被选中的行都会被注释

批量取消注释：输入`Ctrl+v`进入块选择模式，选中对应行首的注释符号，按下`d`即可删除，`Esc`保存推退出

替换指令

指令格式：`[range]s/{pattern}/{string}/[flags]`

- `[range]`表示搜索范围，允许你指定搜索替换的范围
- `{pattern}`表示搜索的模式，它可以正则表达式，字符串，单词
- `{string}`表示要替换成什么字符串
- `[flags]`特殊标识，改变指令的影响方式
  例如：

```
:%s/foo/bar/g #在整个文件中搜索替换foo
:s/foo//g #将光标当前行的foo替换为空字符串
:10,20s/^/#/ #在10-20行添加#注释
:10,20s@^@#@ #在10-20行添加#注释
```

> 实际上，除了字母和数字之外的任何字符都可以用作分隔符

搜索范围

- 如果要搜索15到30行，那么这部分就是`15,30s`。可以用`%`指代整个文件，如果没有给出`[range]`，将在光标所在行进行搜索替换
- 同时可以输入`.`来表示光标所在行，用`+15`表示光标行开始往后15行，`-15`表示光标行开始往前15行。如果只输入`+`则默认为加一行，减法同理。例如`:.,+5s/foo/bar/g`指令就会从当前行到加五行执行

搜索模式

- 上面例子中后两个式子的`{pattern}`部分的符号`^`与行的开头匹配，因此`^xx`表示以xx开头的行，类似的，`$`与行的结尾匹配。此外这部分还可以用正则表达式作为搜索模式，具体可以自行查看
- 如果输入`abs`，会搜索所有字符串，如果只是想搜索`abs`这个单词，可以用`\<abs\>`来表示

替换成的字符串

- 就是个简单的字符串

特殊标识

标识	意义
g	对所有目标执行操作
c	每次替换都进行询问
i	匹配时不区分大小写

范例
`:5,20s/^/#/`在5到20行前面添加`#`，`5,20s/^#//`删除这些井号
`:%s/apple|orange|mango/fruit/g`搜索整个文档的apple、orange和mango，然后

替换为fruit
`:%s/\s+$//e`删除所有行尾的空格，其中`\s`表示空格

ccache

轻薄本编译速度干翻游戏本的秘诀。

> ccache安装之后需要注意配置一下环境变量

用正确的工具做正确的事

The Unix philosophy is documented by Doug McIlroy in the Bell System Technical Journal from 1978:

1. Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new "features".
2. Expect the output of every program to become the input to another, as yet unknown, program. Don't clutter output with extraneous information. Avoid stringently columnar or binary input formats. Don't insist on interactive input.
3. Design and build software, even operating systems, to be tried early, ideally within weeks. Don't hesitate to throw away the clumsy parts and rebuild them.
4. Use tools in preference to unskilled help to lighten a programming task, even if you have to detour to build the tools and expect to throw some of them out after you've finished using them.

计算机是一件工具，而优秀的工具把它的任务做到极致。

从最开始尝试Linux时离不开图形界面到现在逐渐熟练Linux指令，我明显的收获就是操作效率提升很多，并且找到了更好的看待计算机与程序的视角，虽然我在刚开始使用Ubuntu的时候感觉到的是各种各样的不顺手——“学习是痛苦的，但是不学习就无法享受新技术带来的便捷”。通过STFW和RTFM基本上我们就可以掌握大部分工具的使用方式，通过RTFSC我们就可以理解程序甚至操作系统如何运行。如果我们固执于原有的工具甚至没有工具，处理任何任务的结果必然都是效率断崖式下降，我认为这和我们的认知如何发展有关：掌握基本工具的使用方式可以认为是一种地基的搭建，在拥有了地基之后，我们才能构筑起更高的楼房，所有环节相互辅助提升，在最终的效率上体现出的是指数级别的差异。

同时，作为一名正在Coding的计算机学生，我拥有一个GPU很强大而缓存和内存都很小的大脑，这鼓励我在单一的时间做好单一的事情。比如当我面对一个困难的问题时，我可以选择先寻找/构筑合适的工具，在拥有了工具的辅助之后，以最轻的负担来解决每一个问

题。人类运用能动性改造世界，使世界更适合人类。现实的改造可能不容易，但计算机是一个构筑硬件之上的逻辑世界，在这个世界构筑适宜我的规则，会让我的工作更加高效。

By the way

电脑确实比红白机好操作，我在mario中拿到了有史以来的最高闯关记录

第一阶段

计算机如何计算1+2+...+100

学计算机组成原理时，张金老师的一句话让我印象深刻：“计算机本质上就是从某个地方取出数据，进行某种运算，再放到某处去”。计算机拥有传递信息的管线，拥有计算单元ALU，并拥有暂时存储数据的寄存器。依托这三样东西，计算机便可以实现1+1=2，啊不，是1+1=10；而依托1+1=10，计算机实现了加减乘除，实现了键盘输入，实现了向南开大学的图书馆发出座位预约...其他组成原理的东西在此略去不提。

回到这段代码，代码的前两句是赋值语句，将r1寄存器和r2寄存器赋初值0，之后进入一个循环：

行号	代码	功能
2	add r2, 1	r2自增1
3	add r1, r2	r1加上r2，存储到r1
4	blt r2, 100, 2	如果r2小于100就回到第二行

完成这个循环后代码卡死在第五行，r2寄存器中保存了1到100数字的和。从程序员的视角来看，抽象的代码在计算机中取出几个数字，进行了一定的运算，这个视角难以得出更多的信息。但从机器的视角来看，机器从“代码区”中取得一串操作序列，根据这个序列对“内存”进行一定的处理，之后又去到“代码区”取得一串操作序列，对内存进行一定的处理...这个过程中计算机其实没有宏观的视角，只是在执行规定的任务而修改内存的状态而已。同时，达到相同目的的代码也可以有很多种。从这个角度看，计算机的每次运算都是对“代码”的读取和对“内存”的改变，根据一系列“代码”给计算机的操作指令，计算机就可以把一部分“内存”变成我们想要的样子。

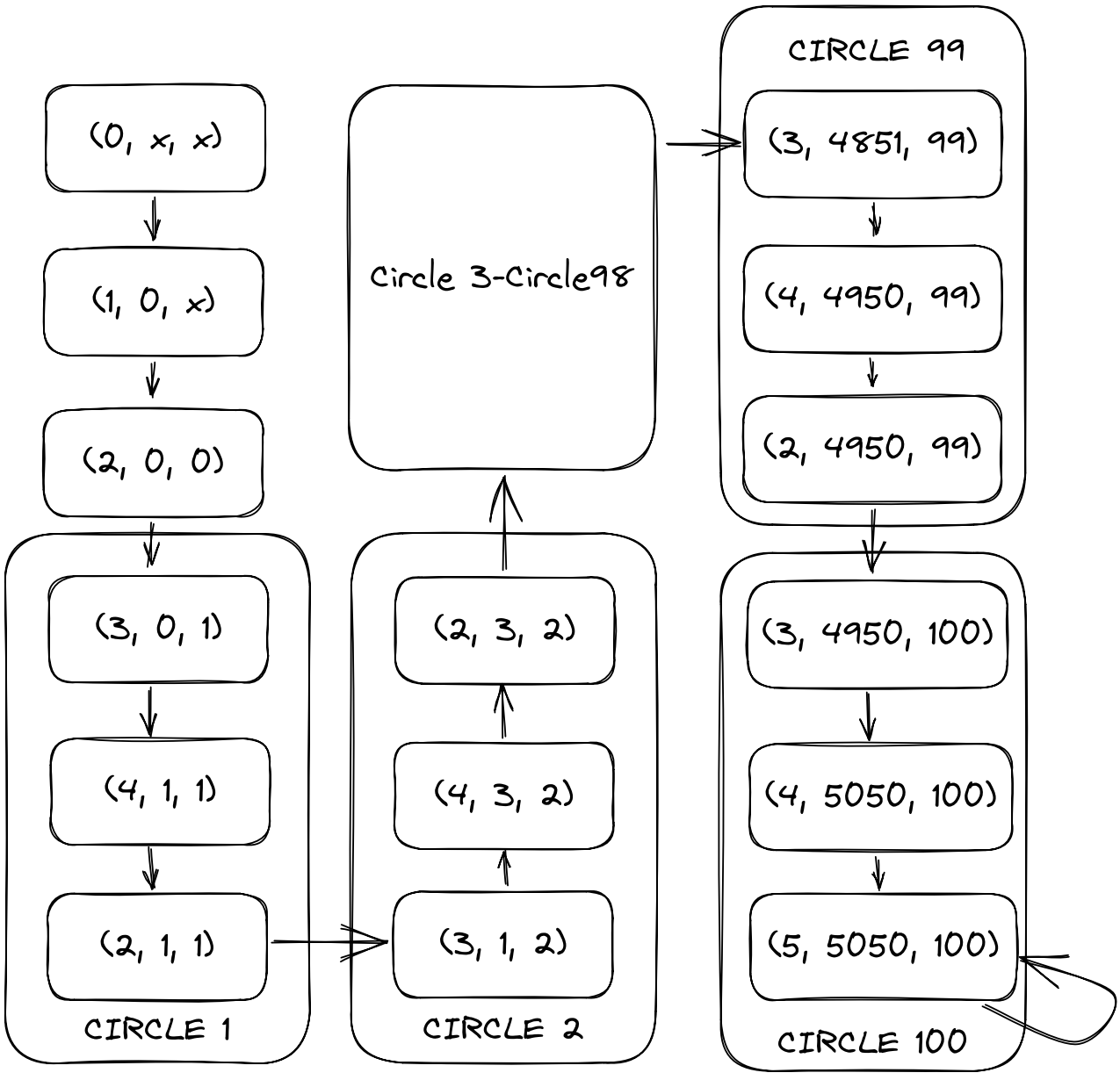
根据后文的介绍，计算机可以被看成一个状态机。这种视角在刚开始接触计算机和状态机时会因计算机或程序的结构过于繁杂而无法得出有效结论，但是当我们理解一部分程序（尤其是汇编程序）的运行逻辑时会起到意想不到的奇妙效果，本程序的状态机会在下一节给出。

1+2+...+100程序的状态机

假定程序状态为(PC, r1, r2)，题目已经绘制好了前三条指令的状态机如下：

```
(0, x, x) -> (1, 0, x) -> (2, 0, 0) -> (3, 0, 1)
```

稍加扩展，即可画出程序的完整状态机：



配置文件kconfig

文件内容很简单，根据实验指导书这是一种基于GNU/Linux中kconfig的简单语言，使用`mconf nemu/Kconfig`指令即可打开设置界面，进行宏定义的设置。

例如我们找一个简单的kconfig条目，其结构如下：

```
config CC_DEBUG
  bool "Enable debug information"
  default y
```

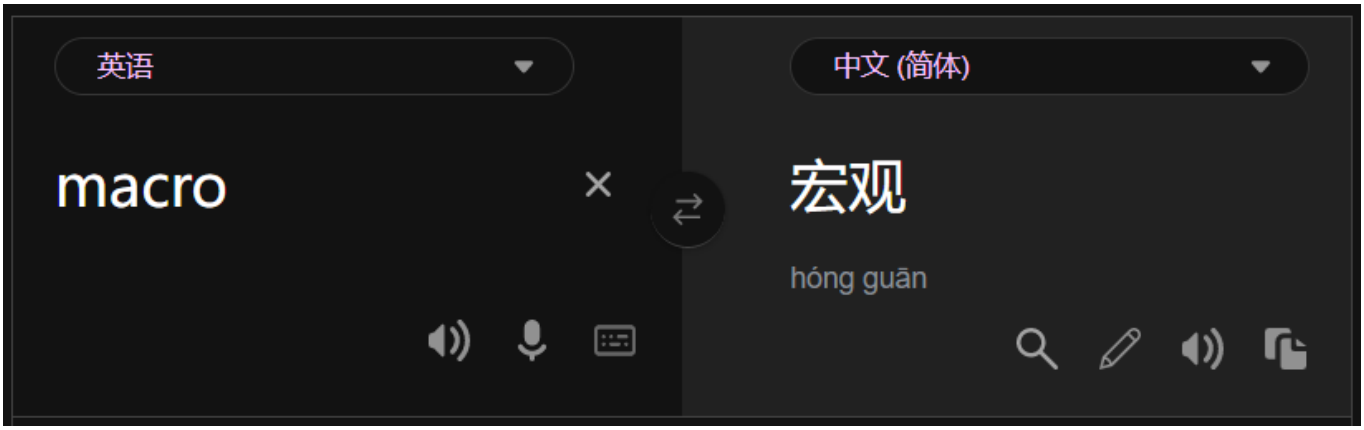
config后跟宏的名字，此处是CC_DEBUG。宏在之后会被编译为CONFIG_CC_DEBUG而在文件中使用。
之后跟一个缩进，bool表示这个宏只有`y`和`n`两个状态，分别表示开启和关闭。default后面跟着默认值，此处的y表示默认开启。

在实现监视点部分时，由于监视点开销很大，一直开启会影响模拟器性能，因此希望能在不用的时候关掉它。所以我在**Testing and Debugging**界面下自定义了宏WATCHPOINT，用于管理监视点是否开启，定义如下：

```
config WATCHPOINT
  bool "Enable watch point"
  default y
```

之后再可视化界面中打开宏设置即可。

神奇的宏定义



宏`IFDEF(macro, ???)`：

- 宏的第一个参数是一个布尔宏（只能是0或者1），第二个参数可以是语句或者代码块。如果布尔宏被定义为1则**在预编译时保留代码**，反之则不然。
- 在定义这个宏时，使用了`MUXDEF`选择器，定义如下：

```
MUXDEF(macro, a, b)
// 如果macro被定义，则选择a，否则选择b
// 可以抽象为两个选择宏
```

- 使用的两个选择宏如下：

```
#define __IGNORE(...)
// 舍弃括号内的输入
#define __KEEP(...) __VA_ARGS__
// 保留括号内的输入
```

那么宏`MUXDEF`是干啥的：

```
#define CHOOSE2nd(a, b, ...) b
#define MUX_WITH_COMMA(contain_comma, a, b)
CHOOSE2nd(contain_comma a, b)
#define MUX_MACRO_PROPERTY(p, macro, a, b)
MUX_WITH_COMMA(concat(p, macro), a, b)
// define placeholders for some property
#define __P_DEF_0  X,
#define __P_DEF_1  X,
#define __P_ONE_1  X,
#define __P_ZERO_0 X,
// define some selection functions based on the properties of
BOOLEAN macro
#define MUXDEF(macro, X, Y)  MUX_MACRO_PROPERTY(__P_DEF_,
macro, X, Y)
```

（注：阅读顺序为自底向上）

- 首先可以看到`MUXDEF`的实现调用了`MUX_MACRO_PROPERTY`，宏`concat`的作用是把`__P_DEF_`作为前缀添加给`macro`，然后调用`MUX_WITH_COMMA`宏。
- `MUX_WITH_COMMA`宏调用`CHOOSE2nd`宏，并将`contain_comma a`作为第一个参数输入。

- `contain_comma`实际上就是上一步骤的`__P_DEF_`+`macro`。如果macro被定义了的话就是`__P_DEF_0`或者`__P_DEF_1`；如果macro没有被定义的话就只是`__P_DEF_`
- `__P_DEF_1`和`__P_DEF_0`都会被换成`X,`，相当于在`CHOOSE2nd`宏中X是第一个参数，那么a就会变成第二个参数从而被保留；如果`macro`没有被定义，那么留下的就是`__P_DEF_`，不会被再次解释，`a`被变成`__P_DEF_a`从而不会被保留。

妙啊。

优美的退出

很不幸，直接输入q退出的时候程序没有被设置为退出状态，因此会产生问题。要解决这个问题，设置语句`nemu_state.state = NEMU_QUIT`即可。

理解基础设施

首先我们作如下假设:

- 假设你需要编译500次NEMU才能完成PA.
- 假设这500次编译当中, 有90%的次数是用于调试.
- 假设你没有实现简易调试器, 只能通过GDB对运行在NEMU上的客户程序进行调试. 在每一次调试中, 由于GDB不能直接观测客户程序, 你需要花费30秒的时间来从GDB中获取并分析一个信息.
- 假设你需要获取并分析20个信息才能排除一个bug.

因此，可以计算出每次调试一个bug耗时约为10min。到目前为止，我的PA0和PA1总git commit次数为493次，由于PA0只有试运行了个位数次，可以粗略认为PA1 commit了约五百次。其中大约2/3的时间程序无法正确运行，也就是大约340次运行是为了调试bug。假设调试一个重要的bug需要运行5次，那么我总共调试了约70个重要的bug，花费时间约700分钟，即便如此就已经超过十一个小时。如果按照题目要求的90%来算的话，这个时间将会达到900分钟，十五个小时。如果之后四次PA使用的时间都是如此的话，我将总计需要花费50-60个小时在调试上面。

假设使用`sdb`时我可以十秒就获得一条信息，时间开销变成原来的1/3，那么调试的时间将会被压缩到20个小时以内，我能节约出大概四十个小时——甚至可以多做一个PA！假如我后续在这个模拟器上进行更多别的修改扩展，那么节省的时间只会越来越多。

在学习tmux的时候指导书提到，不付出学习的代价就无法享受正确的工具带来的便利。这件事情应该是想强调学习掌握正确工具虽然会在初期花费一点时间，但是可以在后期极大的提升效率，而这一点已经在**用正确的工具做正确的事**一节中论证过。回看sdb，虽然开发耗费了一定的时间，但是能极大提升后续的使用效率，很符合gdb基础设施的描述。单一地

来看，gdb使用与否似乎并没有那么大的差别，毕竟gdb本身的开发也耗费了一定的时间；类似的当我们遇到一个合适的工具时，学习使用与否似乎也没有那么大的差别。差别存在于**如果我们一次次如此，最后一定会被拉开一道鸿沟**。

实现单步执行、寄存器打印与扫描内存

单步执行

实现没什么好说的，使用`cpu_exec(1)`即可。值得一提的是`void cpu_exec(uint64_t n)`这个函数本身，由于使用的是uint64_t作为参数，所以当我们想让程序持续执行时，只需要向参数传入`-1`即可，因为`-1`的二进制被解释为无符号整数时，恰好是最大的`uint64_t`，因为其二进制位都为1。

寄存器打印

寄存器打印的实现依托于`info`函数的实现，我参照程序框架代码，使用结构体的方式来定义该函数：

```
static struct {
  const char *arg;
  const char *description;
  int (*handler) (void);
} info_table [] = {
  { "r", "Print all registers", info_r },
  { "w", "Print all watchpointers", info_w },
};
```

使用这样的结构体有两个好处：

- 一是扩展的时候很简单，只需要传入指令和处理函数就可以；
- 二是当用户进行了错误输入时，只需要打印上面的`info_table`数组就可以作为帮助信息提示用户了。

至于底层的实现只需要调用`isa_reg_display()`函数即可，函数内部打印方式（以pc寄存器为例）如下：

```
printf("pc\t0x%x\n", cpu.pc);
```

程序会以16进制输出寄存器的值，便于统一结构，方便阅读

扫描内存

内存的访问依托`vaddr_read`函数实现，阅读代码可知函数的运算符已经经过重载，可以单次阅读长度为1、2、4的内存，并无需手动调整顺序。我进行了一些不同输出的测试，结果如下：

```
// 以下output以地址0x80000000为例

printf("%02x %02x %02x %02x  ",
        vaddr_read(exp_value + 3, 1), vaddr_read(exp_value +
2, 1),
        vaddr_read(exp_value + 1, 1), vaddr_read(exp_value,
1));
// output: 80 00 02 b7

printf("%x\n", vaddr_read(exp_value, 4));
// output: 800002b7

printf("%d\n", vaddr_read(exp_value, 4));
// output: -2147482953
```

为了将内存的输出调整为`?? ?? ?? ??`（`??`是十六进制数）的格式，我选择了单次阅读长度为1，并手动调整输出顺序解决。

第二阶段

表达式词法分析

词法分析的难点在于正则表达式的书写和token的管理，本阶段将会分为这两部分来讲解。

正则表达式

其余正则表达式都比较简单，此处仅列出十进制数字与十六进制数字的正则表达式：

```
{"0[xX][0-9a-fA-F]+", TK_HEX}, // hex number
{"[0-9]+", TK_DEC},            // dec number
```

此外要注意的是`$`, `|`, `+`, `*`, `(`, `)`符号均需要在前面添加`//`来转义

token管理

在存储token时我们使用了一个从256开始的enum来存储各个token。

> 为什么从256开始？我认为是为了避免与ASCII表冲突

假如不进行管理，那么在接下来识别运算符优先级的时候会写出灾难一样的代码。那么如何进行管理呢？考虑把这些token分类进行管理，那么分类的依据是什么呢？答案是token的运算类型。又考虑到这个enum中的数字只是这类token的标识，只需要唯一就可以了，没有其他要求，因此我进行了如下分类：

```
enum {
  TK_NOTYPE = 256,

  // Double
  TK_DOUBLE_BEGIN,  // 无用项，用于标识双目运算符
  TK_EQ,            // equal,  即“==”符号
  TK_NE,            // not equal 即“!=”符号
  ...               // 此处省略剩余运算符
  TK_DOUBLE_END,    // 也是无用项，用于标识双目运算符

  // Single
  TK_SINGLE_BEGIN,  // 无用项，用于标识单目运算符
  ...               // 单目运算符
  TK_SINGLE_END,    // 无用项，用于标识单目运算符

  ...               // 其余的表达式也使用类似管理方式
};
```

我向管理系统中添加了很多无用项，用于标识某一类运算符。比如`TK_DOUBLE_BEGIN`和`TK_DOUBLE_END`，这两个enum将所有双目运算符包围起来，之后我想要识别token x是不是双目运算符，只需要判断`TK_DOUBLE_BEGIN <= x && x <= TK_DOUBLE_END`是否成立即可。这样一来既降低了管理难度，也增加了后续代码的可读性。

表达式计算

算术表达式的计算框架已经搭好，只要明确计算符号参与运算的优先级即可写代码，难度不大，故此处略去。此处主要讲解括号检查函数的思路、单目运算符的实现和寄存器访问实现。

括号检查

括号检查的思路是在函数中设计一个int型变量stack，并初始化为0。为了表示左右括号是否匹配，在参数中引入一个bool型变量bp，如果括号不匹配则将其置为false，代码框架如下：

```
bool check_parentheses(int p, int q, bool *bp) {
  // stack to record paren match
  int stack = 0;

  // scan every token, ignore non-paren token
  for (int i = p; i <= q; i++) {
    检查到左括号stack就加一
    检查到右括号stack就减一
    如果stack小于0就说明表达式不匹配，bp = false;
    如果stack在最后一个字符之前变成0，就说明式子不被括号包围
  }

  最后根据for循环的检查返回结果
}
```

单目运算符

本次实验主要实现了`!`, `*`, `-`三个单目运算符，其中`*`, `-`均需要二次扫描。二次扫描思路如下：

- 在第一遍识别token时，`*`, `-`两个运算符会被识别成乘号和减号，此时我们线性扫描整个表达式，如果遇到这两个符号，则进行检查：
  - 这个运算符是表达式第一个字符
  - 这个运算符前面是另一个运算符
  - 这个运算符前面是左括号
    只要满足上述条件的任何一个，该符号就被转为单目运算符参与运算。

运算的时候就简单了，对于eval(p, q)只需要检查表达式第一个字符是不是单目运算符就可以。如果是则进行该运算，然后返回$s eval(p+1, q)即可（$s代表单目运算符）。

> 如果此处是指针解引用，则使用`vaddr_read(addr, 4)`就可以读出对应的值了

寄存器访问

RISC-V的框架代码和x86的不太一样，RISC-V的访问方式是使用字符串，而x86是使用寄存器编号。

因此，我在expr中维护了一个寄存器数组：

```
const char *regsters_on_expr[] = {
  "$0", "ra", "sp", "gp", "tp", "t0", "t1", "t2",
  "s0", "s1", "a0", "a1", "a2", "a3", "a4", "a5",
  "a6", "a7", "s2", "s3", "s4", "s5", "s6", "s7",
  "s8", "s9", "s10", "s11", "t3", "t4", "t5", "t6"
};
```

在存储token的enum中也以相同的顺序存储这些寄存器，这样在访问的时候用当前寄存器的token值减去$0寄存器的token值就可以得到寄存器的数组下标，通过下标即可得到对应字符串，然后放进框架代码`word_t isa_reg_str2val(const char *s, bool *success)`中即可查找。框架代码内部也是遍历寄存器数组，找到后返回`cpu`中对应数值即可。

> 其实这个方案很愚蠢，仔细观察的话可以发现，它实现了一次下标→名字→下标的转换，那么既然如此为什么不直接用下标呢?

我试着修改过框架代码，但是我发现这个框架代码在别处申明过这个函数，如果修改就需要改多处文件，可能会引发奇怪的bug。加之yzh老师的Gitbook中并没有提到希望我们去改，因此我还是选择保留了这种解决方案。

调试公理

Machine is always right

用`assert(???)`拦截可能的错误是一种不错的解决方案，**尽早处理fault是解决问题成本最低的方案**，这和后文[RTFM](### RTFM (nemu/scripters/build.mk))部分的内容有着异曲同工之妙。

表达式生成器

既然已经写完了表达式计算器，那么应该如何测试它呢？手动自然是很慢的，所以我们需要编写一个自动测试机来测试代码。

测试机的实现选择使用递归实现，由于涉及两个函数代码较为冗长，此处使用框架代码说明：

```
{
  // exp用于存储表达式
  char exp[500] = "";
  bool suc = false;
  gen_rand_expr(exp);
}

void gen_rand_expr(char *exp) {
  switch (rand() % 5) {
    case 0:
    case 1:
          生成"expr 某个运算符 expr"
      gen_rand_expr(exp);
      gen_rand_operation(exp);
      gen_rand_expr(exp);
      return;
    case 2:
      生成"( expr )"
      return;
    case 3:
      生成一个16进制随机数
      return;
    case 4:
      生成一个10进制随机数
      return;
  }
}

void gen_rand_operation(char *exp) {
  生成一个随机运算符
  return;
}
```

通过上述伪代码的执行，程序就可以生成一个个随机的表达式。伪代码中省略了边界条件检查和表达式长度检查等操作，具体代码可以在`expr.c`中查看，难度不大，此处不加赘

述。

为了便捷地测试代码，我添加了一个`pt`指令，输入`pt N`即可自动生成并测试N条表达式，N缺省值为100。

> 此处我遇到了一个问题，刚实现了这个功能时，一旦N超过180，nemu就会显示爆栈了。我检查了一遍代码，自认为没什么空间优化，没有改动的情况下又跑了一次，发现这个问题消失了，并且再也没有复现出来。

第三阶段

监视点池的维护

写了一百遍的链表，注意空指针即可。

监视点的实现

当我们有了一些监视点，我们需要在每次执行完指令后都检查一遍。这个功能需要记录下表达式和上次的值，因此需要在WP结构体中新增条目如下：

```
typedef struct watchpoint {
  int NO;
  struct watchpoint *next;

  /* TODO: Add more members if necessary */
  char expr[64];  //记录表达式
  uint32_t value; //记录表达式上次的值

} WP;
```

我又在`watchpoint.c`中写了这个函数：

```
bool difftest_watchpoint() {
  bool triggered = false;
  遍历所有检查点
    计算expr值
          如果改变，则更新val，triggered = true;

  最后根据truggered决定是否停止
}
```

实现很简单

> **assert(上面的difftest_watchpoint()不是static);**

注意，这类要被其他文件调用的函数不能被实现为static，否则将无法被其他文件识别到。一般来说在工程文件中，**接口化**被认为是良好的编程风格，同时为了效率的提升我们会将一些函数申明为static，这样函数就会被保存在一个固定的区域，使用的时候去直接调用。但是给其他文件提供接口的函数不应被实现为static，否则它的实现将会无法被其他文件找到。

RTFM (efficiently)

我选择的指令集为RISC-V，所以我需要寻找如下问题的位置和阅读范围：

- riscv32有哪几种指令格式?
  1.2 RISC-V ISA Overview
  1.5 *Base Instruction-Length Encoding
  * means it's maybe not necessary, read after others
  2.2 Base Instruction Formats
- LUI指令的行为是什么?
  2.4 Interger Computational Instructions
  Find LUI instruction
- mstatus寄存器的结构是怎么样的?
  18 "N" Standard Extension for User-Level Interrupts

shell command

统计代码行数的指令很简单，直接写出来如下：

```
find . | grep '\.c$\|\.h$' | xargs wc -l
```

这个指令由三部分构成：`find`, `grep`, `xargs wc`，它们分别的作用如下：

name	function
find	search for files in a directory hierarchy
grep	print lines that match patterns
xargs	build and execute command lines from standard input
wc	print newline, word, and byte counts for each file

因此三部分指令的功能也一目了然：

- `find .`中的`.`表示当前目录，即为在当前目录下递归查找所有文件
- `grep '\.c$\|\.h$'`表示使用grep匹配任意以`.c`或`.h`结尾的文件（$匹配到文件名的结尾处）
- `xargs wc -l`首先将输入转换为标准输入，然后统计代码行数

而其中最强大的功能莫过于Linux的管道机制，就是连接每个部分指令的`|`符号。它可以将前一个指令的输出作为下一个指令的输入直接使用（输入输出标准不同时应使用转换），这样就可以任意的组合自己想要的指令来得到想要的效果。

如果想要去除空行，依然借助grep来实现。我们知道在正则表达式中`^`匹配行首，`$`匹配行尾，因此`^$`即可匹配出空行，对应的`[^$]`就可以匹配非空行，于是添加`grep`指令如下：

```
grep -n [^$]
```

但是这条指令只能匹配非空文件名，而非文件内的代码，因此还是需要使用args将输入转化为一个个文件，然后统计非空白行数。完整指令如下：

```
find . | grep '\.c$\|\.h$' | xargs grep -n [^$] | wc -l
```

指令前两部分和之前的完全相同，找出了所有的`.c`或`.h`结尾文件。指令`grep -n [^$] file.xxx`可匹配`file.xxx`内的所有非空白行，那么加上`xargs`后就可以对前面找出的所有`.c`或`.h`结尾文件进行这个操作，最后用`wc -l`统计行数即可。

将上述统计指令写入Makefile中即可：

```
# count: count lines include empty lines
count:
    find . | grep '\.c$$\|\.h$$' | xargs wc -l

# countl: count lines except empty lines
countl:
        find . | grep '\.c$$\|\.h$$' | xargs grep -n [^$$] |
wc -l
```

> 注意，写入Makefile时`$`符号需要转义

RTFM (nemu/scripters/build.mk)

build.mk中的CFLAGS如下：

```
CFLAGS  := -O2 -MMD -Wall -Werror $(INCLUDES) $(CFLAGS)
```

其中的`-Wall`选项可以发现程序中一系列的常见错误警告，例如对检查点部分提到的的部分代码Fault在编译时期就检测出来（虽然能检测出来的Fault十分有限），并且支持对各种错误类型的单独开关。我在写程序的时候会发现有些定义了但是没有使用的变量会被检测为`error`而非`warning`，其中的原因就是加入了`-Werror`编译选项，所有的警告都会被当成错误来处理，并放弃编译。

在编译时使用`-Wall`的好处很明显：能提前发现程序的漏洞并及时提示我们，规避常见的Fault。虽然*任何未经测试的代码都是错误的*，但是能够提前发现并避开一些错误总是好事。至于`-Werror`编译选项的使用，我个人的理解是：假如没有这个选项，我们很容易在代码中引入很多无关的变量或者测试后没删干净的变量，且事后不去做处理。这一方面会导致程序变得臃肿，重要的Warning可能被其他无关Warning淹没；另一方面会导致程序之后的维护变得越来越困难，性能降低甚至出现难以理解的bug。之后的PA中代码与代码之间的交互会变得越来越复杂，前期的Warning被混过去可能会导致后期体系性的难以根除的bug，所有问题都应该被尽早地解决，这才是成本最低的处理方式。