PA1:开天辟地的篇章： 最简单的计算机

匡亚明学院 181840326 张祎扬

实验进度：我完成了所有内容。

在开始愉快的PA之旅之前

不来玩一下吗?

在按要求执行了`make run`以后，屏幕上出现了一个游戏的图形界面，按照README.md的要求进行操作就可以玩这个超级马里奥了。我试玩了几轮，对于这个游戏的实现原理感到神奇和好奇，很想知道它是怎么实现的（先放在这里作为一个目标）。除了在运行游戏的时候电脑**异常发烫**以外，并没有出现什么别的问题。（吐槽一下macbook pro的散热，真真真小火炉）。

新特性-多主线

经过我的~~深思熟虑的~~决定，我打算选择x86。我很喜欢挑战，~~可是我课业太重了,乱来可能会挂。~~简单的肯定是不会选的，所以还是规规矩矩x86吧。

开天辟地的篇章

计算机可以没有寄存器吗?

我觉得可以没有寄存器，所有的数据都存放在内存中，但是会大大降低存取数据的效率。

remain for compensate

计算机的状态模型

图灵机也有状态的概念。内部状态存储器可以记录图灵机当前的状态。在状态模型中，执行指令就是针对当前的状态作出固定的指令。执行程序的本质是连续自动地执行指令，根据当前的状态作出指令，然后更新当前的状态，再根据新的状态执行下一步指令。

RTFSC

需要多费口舌吗?

一个c语言程序从main函数开始执行。
remain for future consideration

为什么全部都是函数?

将`init_monitor()`函数的功能用不同的子函数实现了分装，**大大提高了代码的可读性和可理解性**。虽然将这些函数的函数体直接展开并不影响代码的正确性，但是代码的可读性却大大下降。在阅读函数`init_monitor()`的代码时，我们**并不需要看到它其中的每一个函数的代码实现细节**，而是可以通过这些子函数的命名方式和注释，直接了解每一个子函数的功能，从而直接对`init_monitor()`函数所做的事有一个**大致而全面**的了解，它分别执行了哪些函数，每个函数又实现了哪些功能。当我们需要了解其中某个函数的**具体代码实现**时，才需要定位到该函数的定义位置。当我们需要修改其中的子函数时，

也不必在冗长的`init_monitor()`函数的代码中进行修改，只要直接在外部修改子函数的内容即可。这样的做法提高了代码的简洁度、美观性、可读性，也是比较规范的做法。有一种将比较低层的实现好的代码进行封装的感觉。

- 从讲义的这里开始，我开始阅读`init_monitor()`函数中各个子函数的代码。我不知道inline型函数的定义，~~（因为我的c语言基础hhhhh 言难尽，我会抓紧学习的！）~~STFW以后了解了一些。
- 通过STFM，我知道了`getopt()`函数的作用是解析命令行选项，如果成功找到选项，则返回该选项，如果已解析所有的命令行选项，则返回-1，同时它的参数里可以提供包含合法选项的字符串。它还可以在选项后跟不同的参数，并把该参数的指针赋给optarg。所以`parse_args()`函数主要就是对命令行的输入命令进行解析。

参数的处理过程

关于这些参数是怎么来的，参数argc和argv都是由`main()`函数传递的参数，即执行`main()`函数时命令行的输入。参数optstring在调用函数`optget()`时给出，是欲处理的选项字符串。

- `init_log()`函数的参数是指向log_file的指针，如果为空，则直接返回，如果不为空，则以写入模式打开日志文件。所以这个函数的功能是初始化log日志文件。
- 其他函数的功能在PA讲义中都给出了比较详细的解释，我也对照着这些解释阅读了部分源代码，并且对一些不懂的地方**STFW**和**RTFM**。

实现x86的寄存器结构体

根据讲义提示，reg.h中的代码并没有正确实现寄存器的结构，并且讲义中给出了正确的x86寄存器结构。首先我定位到reg.h中关于寄存器结构的定义，发现它是一个命名为CPU_state的struct,并且在cpu.c中定义了`CPU_state cpu`,所以reg.h中的struct CPU_state就是对cpu的代码实现。
可以观察到最初的实现代码如下：

```
typedef struct{
 struct{
     uint32_t _32;
     uint16_t _16;
     uint8_t _8[2];
 }gpr[8];
 rtlreg_t eax, ecx, edx, ebx, esp, ebp, esi, edi;
 vaddr_t pc;
}CPU_state;
```

可以看出rtlreg_t定义了8个32位的通用寄存器，并且下面还用vaddr_t形式定义了pc。但观察代码可以发现，原始代码中对于寄存器的定义采用了struct结构，也就是说这些寄存器数组在内存中是顺序排放的，而没有共用的部分，所以无法实现**“这些寄存器在物理上不是独立的”**概念。要解决这个问题，考虑使用**union**，同样的地址可以解释为不同的数据类型。
接下来就是观察讲义中给出的寄存器的结构，因为后面还要用到这里定义的rtlreg_t类型的寄存器，所以把它保留。由于这些寄存器（eax,ecx,……）**在物理上是独立的**，所以考虑把原定义放进一个struct结构里。为了实现其他更低位数的寄存器共享内存的效果，把原定义中的struct改为union（**在这里我通过STFW了解了anonymous union**）
修改后的代码如下：

```
typedef struct{
union{
   union{
       uint32_t _32;
       uint16_t _16;
       uint8_t _8[2];
   }gpr[8];
   struct{
       rtlreg_t eax, ecx, edx, ebx, esp, ebp, esi, edi;
   };
};
vaddr_t pc;
}CPU_state;
```

reg_test()是如何测试你的实现的?

`reg_test()`函数首先定义了一个`sample[8]`数组，数据类型是32位。接着定义了一个`pc_sample`变量，并且把它的值赋给cpu.pc。搜索发现定义reg_l()的宏在头文件reg.h中，它检查了索引index是否越界，并且定义它为**cpu中相应索引相应长度的寄存器**。随后`reg_test()`函数通过for循环，给sample[8]的每一个元素赋上一个随机值，并且令相同索引的32位寄存器也赋上该值。然后用`assert()`函数检查sample元素的低16位是否与对应的16位寄存器中的值相等。
接下来的一组`assert()`函数检查每个8位寄存器的值与sample数组中的对应部分的值是否相等（即32位寄存器的低8位和8-16位）。再接下来的一组`assert()`函数检查每个sample数组的值和对应cpu.寄存器的值是否相等。最后一个`assert()`检查pc的值和pc_sample的值是不是相等。
这一组assert()条件是根据**x86的寄存器结构和各位数寄存器之间的相互对应关系**写出来的，它可以检测我自己用代码实现的寄存器结构以及各寄存器之间的物理（包含）关系是否完全符合x86标准的寄存器结构。
同时，`reg_test()`函数中还调用了`rand()`函数来生成随机值，并把随机值赋给sample数组。**数值的随机性可以在最大程度上避免`assert()`函数的检测失误**，从而保证寄存器的结构实现正确。

> 一点心得
> 通过修改定义cpu的struct(),我对struct和union之间的区别和联系有了更深刻的理解，也对寄存器的结构有了更多了解。当不同位数的寄存器之间有嵌套关系，而不是完全独立的时候，union可以用来实现存储空间的重复利用，比如eax和ax。当要实现互相独立的一组寄存器时，应该使用struct，比如eax和edx。

究竟要执行多久

在cpu_exec.c中查看函数`cpu_exec()`的定义，是`void cpu_exec(uint64_t n)`,所以**函数的参数是64位无符号数。当传入的参数为-1时，它将被解释为64位的无符号整数，它将转化成一个非常大的值**（64位无符号整数的最大值）。继续观察`cpu_exec()`函数的代码可以发现，保持cpu不断执行指令的是一个for循环，`for (; n>0 ;n--)`,参数为-1时，n被解释为一个非常大的无符号数，这可以**保证cpu持续不断地执行指令**，直到执行完所有的指令，而避免指令还没执行完就提前停止执行。

潜在的威胁

属于未定义行为吧.....

— An expression is shifted by a negative number or by an amount greater than or equal to the width of the promoted expression (6.5.7).

— An expression having signed promoted type is left-shifted and either the value of the expression is negative or the result of shifting would be not be representable in the promoted type (6.5.7).

温故而知新

`opcode_table`数组的类型是`OpcodeEntry`,而我查找了头文件exec.h，发现`OpcodeEntry`在其中被定义为一个结构。所以`opcode_table`数组的每一个元素都是一个结构。

谁来指示程序的结束?

remain for compensate

有始有终

remain for conpensate

阅读Makefile

```
3 INC_DIR += ./include ./src/isa/$(ISA)/include
4 BUILD_DIR ?= ./build

34
35 QEMU_DIFF_PATH = $(NEMU_HOME)/tools/qemu-diff
36 QEMU_SO = $(QEMU_DIFF_PATH)/build/$(ISA)-qemu-so
```

可以看到makefile文件用$(ISA)指定你选择的isa的路径，可以根据我们自己选择的isa来改变相应的路径，从而支持多种ISA。

```
41 # Files to be compiled
42 SRCS = $(shell find src/ -name "*.c" | grep -v "isa")
43 SRCS += $(shell find src/isa/$(ISA) -name "*.c")
44 OBJS = $(SRCS:src/%.c=$(OBJ_DIR)/%.o)
```

这里制定了编译的源文件和一些其他的依赖文件。

我觉得我现在对于shell脚本和Makefile之类的有点一知半解，能看懂一些但又不是非常透彻我打算抽出时间来系统地学习一下make和shell脚本语言

~~我打算花一整天（周一下午+晚上）的时间阅读整个框架的源代码。使用tmux来开启多窗口，并使用grep工具来帮助我快速定位各个函数所定义或出现的源文件的位置。首先我定位到程序从main函数开始执行，而main函数在main.c中，所以从main.c开始阅读，并在阅读的过程中添加注释，记录我的理解。~~

基础设施

总有一天会找上门来的bug

> 一点感想：上学期和17级计算机的同学有过一些交流，当时就听说对方debug了很久，最后发现是以前的实现有问题。当别人都在往后进行的时候，他还在debug以前的代码。这件事情其实给我留下了很深的印象。我觉得代码世界其实存在很多侥幸的东西，比如说你的代码在少量的测试中没有表现出问题，但其实它的某个地方的小细节的实现是有问题的，如果**不经过大量测试**，或者说**不充分考虑到一些可能的特殊情况**，你很难发现代码存在的小问题，从而有可能对之后的工作产生一些影响，并为了寻找这个小问题付出成倍的更多的时间。学习程序设计的时候做的OJ其实也是这样，当我以为自己的代码没有问题的时候，总有一些无法通过的样例提醒我还有没有考虑到的特殊情况。所以我**非常感谢**这位同学及早地对我敲响了警钟，让我在开始学习代码的初期就树立了一些**全面**而**周到**的思想。

如何测试字符串处理函数?

在做完表达式求值以后，我收到了一点启发，也许也可以写一个随机生成字符串的函数，生成一定数量的测试用例，再写一个函数读入这些测试用例，并且判断字符串处理函数的结果是否正确。

单步执行

观察源代码，我发现框架代码使用`readline()`函数来读入`(nemu)`后的输入，并且调用`strtok()`函数对读入的字符串进行解析。通过RTFM我了解到`strtok()`函数的功能是将字符串分割成子字符串，当获取其后的子字符串时只需要将第一个参数str换成NULL，这样就可以处理**命令后跟着参数**的情况，比如单步执行时就要用到的`si N`.所以要实现单步执行，必须实现一个可以解读参数的函数，而参数是以字符串的形式给出，所以还需要将字符串转化为int型数据。通过RTFM，我找到函数`atoi()`可以实现把字符串转化为int型数据。所以我打算通过调用这个函数来实现读取单步执行的参数。
当`atoi()`的参数不符合int型数据时，它将返回0，又因为在这里输入的n不能为负数，所以要进行一个条件判断，当n>0时这个n才是有效的，而当n<0时，我需要**输出一些错误信息**。考虑到后面的实现也需要输出错误信息，我打算**实现一个函数**`cmd_err()`,它可以实现对错误信息的输出，并且在这个函数中用`switch case`来实现不同的错误分支。
大概实现代码后我第一次编译，发现switch-case的格式写错了.....~~dbq我c真的有点菜而且有点忘记了我马上去学~~.改完之后第二次编译发现忘记在cmd_table[]里加上新实现的cmd_si指令了。第三次编译发现我的`cmd_err()`函数返回类型写了int但是并没有`return 0`,所以我把返回类型改成了void。**这次！！！！！编译终于通过了！！！！！**
我测试了很多次单步执行的指令，**暂时**还没有找到任何问题，以后遇到bug再说吧.....

```
Welcome to x86-NEMU!
For help, type "help"
(nemu) si -1
Invalid arguments for command 'si'
(nemu) si 0
Invalid arguments for command 'si'
(nemu) si 1
  100000:   b8 34 12 00 00                      movl $0x1234,%eax
(nemu) si 3
  100005:   b9 27 00 10 00                      movl $0x100027,%ecx
  10000a:   89 01                               movl %eax,(%ecx)
  10000c:   66 c7 41 04 01 00                   movw $0x1,0x4(%ecx)
(nemu) si 5
  100012:   bb 02 00 00 00                      movl $0x2,%ebx
  100017:   66 c7 84 99 00 e0 ff ff 01 00       movw $0x1,-0x2000(%ecx,%ebx,4)
  100021:   b8 00 00 00 00                      movl $0x0,%eax
  100026:   d6                                  nemu trap
nemu: HIT GOOD TRAP at pc = 0x00100026
```

打印寄存器

根据讲义提示，只需要调用函数`printf()`即可。因为目前还没有实现eflags寄存器和段寄存器，所以先实现通用寄存器的打印。我参考了gdb的`info r`命令的输出格式。需要注意的是既要输出16进制的机器数又要输出十进制的真值。完善`isa_reg_display()`函数很简单，只需在函数框架中填上`printf()`代码。此外还需要实现的是对于info命令的解析。因为info命令后可以跟两种参数，所以要做好区分。实现命令解析的步骤和单步执行类似。同样地，在完成这一部分时，一些编译错误总是让我认识到自己的不足。

> 一点心得：在代码很多的时候，保持条理性是非常重要的，这就需要我们对框架代码有一个大致而全面的了解，比如说如果要实现一个功能，我该在哪些地方实现代码？在实现这些功能的代码时，我经常出现一些遗漏，比如要调用的函数忘记声明，或者是忘记把新增加的功能添加到table中等等，这些都是对框架代码不够熟悉而造成的，这也提醒我要对框架代码有一个更全面的把握。

编译通过以后，我输入命令`info r`可以打印出所有寄存器的值了，但是我发现了一个问题，用`info r`打印出的所有寄存器的值都是相等的，它们的值在eax被改变后统一改变。我检查代码，果然发现`printf()`的参数忘记改了……~~我知道我太蠢了orz~~改完之后果然正常了，我连续单步执行，观察到打印出的寄存器的值都是正确的。最后我解决了一下显示时候的列对齐问题。
根据我以前的观察，gdb在没有程序运行的时候是不会打印寄存器状态的，而这里我实现的`info r`命令似乎无论程序在不在跑都可以打印寄存器状态，我打算在这里实现首先判断程序在不在运行，如果在的话再执行`info r`。先留在这里我有空再来补

扫描内存

x命令与之前实现的两个命令不同，因为它出现了第二个参数，根据之前的经验，只要再调用一次`strtok()`函数，就可以取得第二个参数值。连续打印n个四字节的操作只需要用一个for循环来实现即可。因为讲义先不要求实现表达式求值，所以我只需要假设读入的是一个十六进制数，所以用`sscanf()`函数来进行格式化字符串。
需要注意的是我让每一行显示四个十六进制数（模仿gdb的格式），所以要对i进行判断，在每一行的开始打印地址，并且每四个就要换行。

> **让我觉得自己很nb的是，这次写完代码竟然一遍通过编译，我make run了一下，各种测试并且和内置的客户程序对比，都没有发现什么问题……那姑且当作我的实现暂时没有问题吧**

表达式求值

实现算术表达式的词法分析

- 在为各种类型的token添加规则的时候并没有遇到太大的困难，对于一些不理解的正则表达式我STFW也得到了答案。
- 在将token信息记录到数组中时，首先要判断有没有溢出问题，因为根据代码框架可以看到，token中的str数组长度只有32，所以要先对字符串的长度进行判断，如果大于32，则用`assert(0)`终止程序。接下来就是一组`switch case`将对应的tokens数组赋值。**需要注意的是**，根据讲义，假如token是一个数字，那么不仅需要存储它的类型，也需要存储这个数字的值，所以在这个case里我用一个for循环来为数组赋值。

> 一点感想：到这里我其实还是一头雾水，只是按照讲义的要求写了肉眼编译**看似**没有问题的代码，并且也通过了make编译，但我完全不知道怎么测试我实现的代码块是否正确。尽管讲义已经指明了用`assert()`,gdb，`printf()`等方法进行调试，还提供了单元测试思想的KISS法则的应用，但是代码框架对我来说还是一个整体一般的存在，我好像还不能透过整体看到其中的一个一个单元，这让我一度无从下手，所以我决定先继续往下实现下面的代码。

为什么printf()的输出要换行?

关于这个问题，我自己的理解是为了使输出的代码美观且易于区分。假如没有换行符，程序的输出会和程序结束执行后显示的信息完全连在一起，这让程序的输出难以区分。同样地，假如我想在一个for循环中用`printf()`打印每一步循环中变量的值，假如我不换行，循环数或者变量数又很多的情况下，这就会让输出的调试信息变得难以区分且不够明晰。
然后我又继续STFW，找到了stackoverflow上的一个回答，C无法在带引号的字符串中嵌入文字换行符，也就是说

```
printf("hello,world
");
```

这样的代码是不符合规范的，会引发编译器报错。所以无论在什么时候，加上'\n'总是对的。生成干净输出的最简单方法是每次调用`printf()`只打印一行，并且在行尾只包含一个换行符。
还有一个回答是这样解释的，通常如果不使用换行符结束格式字符串，在某些情况下（我理解的是输出的信息过长而无法在一行中显示完全），某些输出将保留在stdout缓冲区中，这时候需要调用`fflush`以显示所有的输出。但是在这种情况下你不会得到编译器信息，因为它不是一个错误。如果你的程序只是打印一行然后截止，它可能会刷新缓冲区，但如果你有很多打印行或者你的程序有很长的运行时间，省略的换行符往往会导致输出保留在缓冲区中。而你有时很难注意到这一点。

实现算数表达式的递归求值

一下子实现整个算术表达式求值的函数过于宏大，我目前也没有特别清晰的思路，所以我决定先从小的方面实现，先实现`check_parentheses()`函数。我觉得这个函数的实现有几个难点，首先所有的括号必须是对称的，不能有多出来的括号，否则就fail了，其次如果首尾的括号不能凑成一对（比如它们分别属于两个子表达式）那也会fail。我的想法是，首先检查开头第一个是不是左括号，最后一个是不是右括号。为了保证开头的括号匹配最后的括号，中间的所有括号单独拿出来看也必须是有序的，即一定先出现左括号，再出现右括号，并且保持配对。在这里我用一个int型变量k来表示配对情况，遇到左括号+1，遇到右括号-1，当k是负数则说明不匹配，当最终的k不为0时，也说明括号不匹配。**（这个想法借用了一点上学期的python课的类似的思想）**
按照讲义的递归思想，接下来要做的事情是完成函数`eval()`.按照讲义给定的框架，如果`p == q`，说明是一个数字，而之前已经将数字的值存储在tokens[].str中，只需要利用ascii码的值，
用`tokens[].str[]-48`可以直接求得数字的值。
当情况为一个表达式时，需要确定主运算符。
因为结合的顺序是从左至右，当运算符优先级相同时，主运算符是最右边的运算符，所以在循环时从右往左。然后根据优先级，先判断加减，再判断乘除。假如遇到了括号，则括号内的运算符不算数。
目前实现完之后只是一个非常简单的版本.....**感觉可能会出现非常多的bug**
最后参照之前的实现来实现p命令。

> 我遇到的bug：在执行p命令后，一直输出`Invalid expr!`，由此可以推断出布尔指针success指向的是false。我以为是`make_token()`函数的问题，于是插入了若干个`printf()`和`assert()`来检验，都没有发现异常……最后愚蠢地发现声明`bool success`的时候并没有初始化为true……

实现带有负数的算数表达式的求值

我觉得我会做的但我现在先不做这个

事实证明这个问题在后面还是需要解决的，其实也就是判断一下负号前面的一个符号是什么，如果是表达式的话显然不能解释为负号，如果是其他运算符则根据优先级判断是不是应该解释为负号。

实现表达式生成器

我按照我的理解写了一个递归的表达式生成器。采用两个参数，分别指示表达式在数组中的开始和结束位置，在调用时需要通过参数指定表达式的长度。但是在编译时遇到了越界问题，于是我手动把buf[]数组的长度改小了之后就编译通过了。然后我在实现表达式的switchcase中加入了括号的生成。
生成100个测试样例的过程不算非常困难。但是在测试的时候我遇到了一件令我非常痛苦的事。**segmentation fault.**起初我以为只是token数组不够大的问题，但我转念一想，由于我的函数可以指定表达式的长度，而我设定的长度是11，完全不存在超出的现象，所以我用gdb进行调试，在进行表达式求值前printf一些值，发现表达式的字符串在送入求值函数前是正确的。似乎只是求值函数的问题。
事情真的这么简单吗?
我注释掉表达式求值的函数，把格式化读入的字符串再打印出来，发现后面完全乱掉了。于是我更改了之前使用的`fgets()`函数的方案，重新使用`fscanf()`函数进行格式化读入，这一次输出的结果正确了。但segmentation fault还是完全没有解决，gdb显示在调用regexec函数时出现了问题，然而这是一个库函数……我真的百思不得其解，STFW也没有得到有用的结果。这件事让我的周日晚过的很痛苦，最后我直接提交了实现不完全的代码。

> **致谢：感谢胡俊豪同学用视频聊天的方式对我的帮助，虽然最终也没有解决这个问题，但是我从他与我交流的过程中还是学到了一些他的思路和调试的方法，这也在一定程度上对我平复心情起到了很大的帮助。**

最终这个问题还是解决了，原因在于我在`main()`函数里注释掉了框架代码中原有的两个函数，只需要解除注释即可。知道原因的时候我哭笑不得，这一个小小的问题耗费了我一整个下午+晚上，消磨着我的意志和毅力。但最终我也没有弄明白到底是什么原因。**我非常希望之后的某一天，我可以恍然大悟吧。**

> 一点感想：jyy说过PA对我们来说将是一个噩梦般的存在，虽然早有耳闻，但这的确是我在PA道路上**第一次真真切切地感觉到它对我的身心和意志力的消磨**。但在这一过程中，我也确实感受到自己debug能力的提高，也因为自己基础不够扎实而有很多无力的感觉。整个做PA的感觉就是疯狂地思考疯狂地找错，甚至因为知识储备不够而疯狂地学习。但是我仍要感谢它，是它不断地改变我的心境，让我学会更佛系的看问题，同时也让我对一些困难的事情更加保持平和的心态。再次感谢一些同学对我的帮助，还有一些学长学姐对我的心境上的开导，非常感谢。

解决了segmentation fault以后，我终于得以对随机生成的表达式进行测试，但我发现结果并没有那么顺利，有很多表达式的值计算出来是错误的，这说明我的表达式求值函数实现还有问题，**所以我又回去debug了。**一开始我的实现想法是从右往左检索运算符，检索到的第一个优先级最低的就是主运算符，想起来没有什么问题，但是真的实现起来也不太对。而且我用了多个for循环，假如第一次没有检索出'+'和'-'，就将第二次检索'*'和'/'，而当运算符更多的时候，更要多次检索，效率就更低了。我觉得还是应该检索整个表达式中优先级最低的运算符。那怎么表示出优先级最低呢？我觉得如果直接在代码中

写上`op=='+'||op=='-'`这种代码是很不好的，一个是比较繁琐，另一个就是不容易考虑到所有结果。所以我参考了一下C语言的运算符优先级表，打算给每个符号赋上优先级。然后在实现表达式求值的时候通过比较优先级的方式来返回主运算符，这就直接把问题转化为数值比较了。为了看起来简洁美观，我直接注释掉了之前写的代码，并且在函数外重新写了一个寻找主运算符的函数`get_op()`.
考虑到之后的除零问题，可能会有表达式求值失败的情况，所以我打算增添一个参数success，用来指示表达式求值成功与否，并且做相应的处理。
在这一切都完成以后，我发现还是有很多表达式计算不正确。于是我设置了断点，用`make gdb`命令查看每一步，发现到了某一步读取十进制整数的时候，多读取了一位并不存在的数，然后我突然意识到，**我在复制十进制整数到str的时候并没有在结尾加'\0'！**果然改完之后，整个世界都好了……

> 一点感想：这件事让我对c语言的字符串和数组有了更加深刻的理解，我打算花一点时间再了解一下关于字符串处理的库函数。

除0的确切行为

我发现一部分除0的行为的结果是-1，但还有一部分不符合这个规律。为了过滤一些除0行为，我在判断操作码时的`case '/':`中加入了if语句，如果表达式val2的值为0的话，则不计算除法，返回一个错误信息。

监视点

扩展表达式求值的功能

已经实现的功能有`+,-,*,/`,还需要实现的功能有`0x,$reg,(),==,!=,&&,*`.有了之前的思路铺垫，这些实现起来都没有那么困难了。大体就是把这些类型添加到之前的结构中，再根据操作符优先级对它们的优先级重新赋值。至于读取寄存器的值，我在头文件reg.h中找到了一个函数`reg_name()`，可以返回寄存器的名称，这样就很方便，只需要直接用`strcmp()`函数比较名称是否相等就可以了。关于解除引用，也只需要再次调用之前调用过的`vaddr_read()`函数即可。对于符号和乘号的判断，讲义已经给出了思路，如果它的下标是0，那么说明是取负（取地址），如果它的下标不是0，那么就看它的前一位是什么。根据我的思考，它不能是寄存器或者整数或者右括号。然后在选项中把它排除。

全部完成以后，make run一下，我手动输入了一些表达式测试，发现大体已经比较完整了，但还是有一些小瑕疵，比如还没有实现寄存器的大小写都可以识别之类的**（现在已经实现了）**，虽然问题不大，但是我会慢慢地去修改。以及按照讲义的意思，以后还会有需要添加进去的表达式，这都需要我对之前的代码不断修改。

> 一点感想：对于表达式求值的实现，是一个循序渐进的过程，先实现最简单的版本，让它能够跑起来，再不断考虑更多更复杂的情况，不断往里面添加和优化。回头看整个表达式求值的过程，发现它已经是一个非常庞大的体系了，如果让我直接从脑海中构思这么一个庞大的框架，要考虑到这么多特殊情况，我觉得是非常困难的，而从简单的基本情况开始，一步步精细化，却是可以实现的。这也让我感受到了大体框架的重要性，在考虑比较庞大和复杂的任务时，我们不应该从一开始就追求面面俱到，而是要逐步调试，一步步实现更细节的优化。

实现监视点

命令行需要解析的命令主要有三条：设置监视点`w expr`、删除监视点`d num`和打印监视点信息`info w`。一些具体的分析如下：

- `d num`遍历使用中的链表，删除编号为num的节点。可能会调用函数`void del_wp(WP *wp)`

- `info w`遍历使用中的链表，打印所有的监视点信息（可能会用到for循环打印监视点信息，**另外我参考了gdb的输出格式，发现它还会打印触发该监视点的次数，所以我在考虑添加一个表示次数的值**）
- `w expr`添加监视点信息。它先通过`new_wp()`申请一个空闲的监视点结构，并将表达式读入结构中的表达式。每当执行完`cpu_exec()`,就对表达式求值，检查值有没有发生变化，**如果发生变化，程序暂停（将`nemu_state`设置为`NEMU_STOP`)**,输出一句话提示用户触发了监视点，并返回到`ui_mainloop()`循环中等待用户的命令。

分析一下不难发现，只要实现了基本的对链表进行处理的几个函数，就不难实现监视点了。

在具体的实现过程中，我先实现了几个对链表的基本操作，比如链表的增添、删除之类的基本函数，然后再实现命令行命令的解析。编译的过程还是很反复的，有各种小毛病。后来终于可以run起来了，问题又很多。首先是我添加了一个监视点以后，再用`info w`查看所有的监视点信息，发现一共有32个，并且每个都是一样的，然后我找了半天以后发现，假如`head`指针是空的时候，将`head`指向`free_`的第一个节点，但是并没有把它的`next`设置为NULL，所以在显示`info`的时候就会一直往下循环。然后我还把对链表节点设置的语句放在了一个else里，所以在if语句里并没有对节点进行初始化，所以无论我怎么测试，第一个监视点的值和表达式字符串都是缺失的。

> 一点感想：debug真的是一个漫长而艰巨的过程，很多时候你最终看到的错误也并不就是代码出错的地方，更多的时候我们还是需要借助gdb来细致理解程序的行为。

温故而知新（2）

static在此处的含义是静态变量。静态变量的生存期是程序的整个运行过程，而监视点要在程序的整个运行过程中起到监视作用，所以它的生存期和作用域必须是全局的。此外，静态变量的内存分配是在程序编译时就进行的，而不是在运行时动态分配内存。如果不使用静态变量，那么监视点链表就不能完全存在于程序运行过程中，就不能起到一个全局监控的作用。

一点也不能长?

- 指令的长度为一个字节是必须的。
- 在my-x86中，上述机制不可以正常工作。int 3的单字节形式很有价值，它可以用断点替换任何指令的第一个字节，包括其他一字节指令，而不会覆盖其他代码。这是我们能够插入一个断点而不覆盖其他指令。假如我们要在单字节指令上设置一个断点，如果替换断点的指令的长度超过一个字节，将被迫覆盖下一条指令的一部分，这会使该指令乱码，并且可能产生完全无效的指令。

随心所欲的断点

- 如果把断点设置在指令的非首字节，gdb会停止进程并且意外退出。
- 关于原因的一点想法：断点如果设置在非首字节，那么在指令的首字节就不会检测到断点，就会继续执行。但又因为断点添加在指令中，导致指令乱码（或者指令码不完整）而不能读出完整的指令，而程序又不会因为检测到断点而停止，又不能完整执行一条指令，所以只能意外退出。

NEMU的前世今生

- STFW：The **main difference** between emulator and debugger is that **an emulator is a tool that allows one computer system to behave like another computer system while a debugger is a tool that helps to test and debug computer programs.**
- gdb是如何调试程序的：Ptrace

如何阅读手册

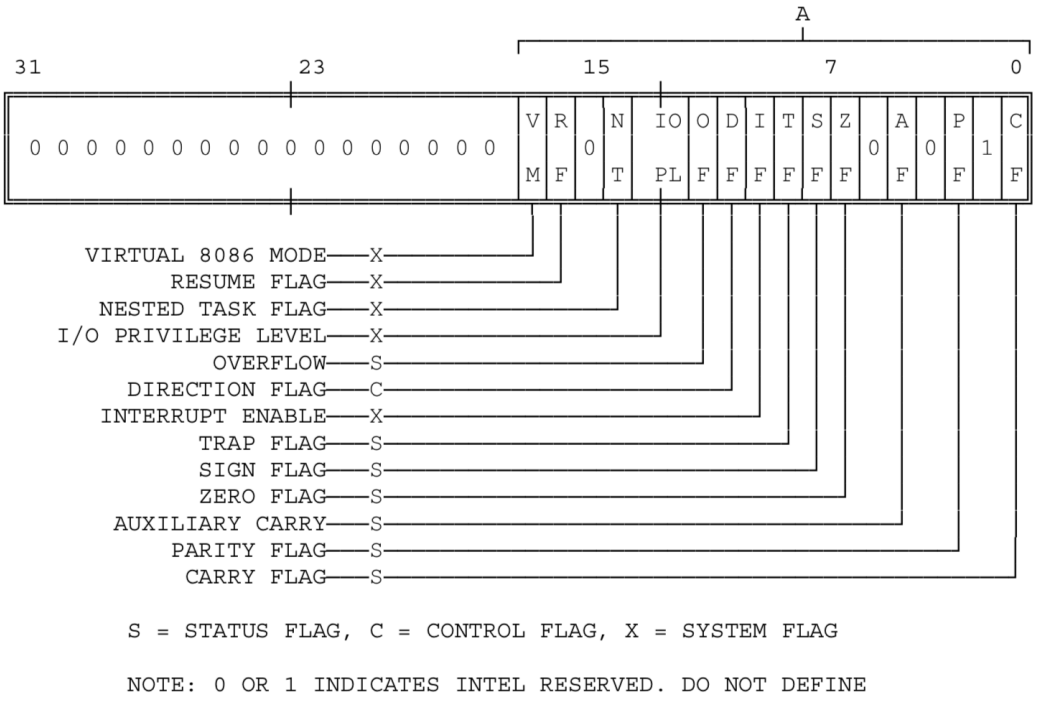
尝试通过目录定位关注的问题

我查阅了目录，定位到它在如下位置

必答题

- **送分题**我选择的ISA是x86
- **理解基础设施**
  - 假设一共500次编译，其中90%用于调试，也就是有450次编译用于调试。假如30s可以从gdb获取并分析一个信息，而排除一个bug需要20个信息，那么排除一个bug就需要$20 * 30 = 600(s)$，也就是10min。假设一次编译排除一个bug，那么就需要$450 * 10 = 4500(min) = 75(hour)$.也就是说，这个学期下来，将会在调试上花费75小时，**这只是非常乐观的估计**。
  - 假设简易调试器只需花费10秒的时间获取相同的信息，那么每获取一个信息可以节省20s，排除一个bug需要20个信息，可以节省$20 * 20 = 400(s)$,总共可以节省$400 * 450 = 180000(s) = 3000(min) = 50(h)$,也就是这个学期下来，简易调试器可以帮助我节省50个小时的时间，**这同样是非常保守的估计**。
- **查阅手册**
  - EFLAGS寄存器中的CF位是什么意思?

    page 33-34　2.3.4 Flags Register



2.3.4.1 Status Flags

The status flags of the EFLAGS register allow the results of one instruction to influence later instructions. The arithmetic instructions use OF, SF, ZF, AF, PF, and CF. The SCAS (Scan String), CMPS (Compare String), and LOOP instructions use ZF to signal that their operations are complete. There are instructions to set, clear, and complement CF before execution of an arithmetic instruction. Refer to Appendix C for definition of each status flag.

从这张图可以看出，CF是进位标志。

3.2 Binary Arithmetic Instructions具体介绍了怎么设置CF位

3.4 Logical Instructions也介绍了对CF位的影响

- ModR/M字节是什么?

  2.5.3 Memory Operands

  2.5.3.2 Effective-Address Computing

  17.2 Instruction Format

  17.2.1 ModR/M and SIB Bytes

- mov指令的具体格式是怎么样的?

  3.1 Data Movement Instructions

- **shell命令**
  - `NEMU/`目录下的所有.c和.h文件总共有**5515**行代码。
  - 使用的命令：wc -l `find -name *.[ch]`
  - 切换回pa0，一共**4970**条代码。相比之下，在pa1中写了**545**行代码。
  - 除去空行外总共有**4535**行代码。
- **使用man**

  查阅手册的结果：

`-Wall` This enables all the warnings about constructions that some users consider questionable, and that are easy to avoid (or modify to prevent the warning), even in conjunction with macros. This also enables some language-specific warnings described in Section 3.5 [C++ Dialect Options], page 39 and Section 3.6 [Objective-C and Objective-C++ Dialect Options], page 51.

`-Werror` Make all warnings into errors.

-Wall的作用是打开所有的警告，-Werror的作用是让所有的warning都以error显示。

使用它们的作用是在编译时尽可能地把潜在的fault直接转变为failure，这样可以最大程度地规避我们代码的不规范行为，更容易帮助我们在较早的阶段发现问题，从而解决bug。这样可使我们尽早发现error，从而降低调试的难度。