

更新于PA2的前半阶段：

I. 实验进度：

A. 我完成了前半阶段的全部部分：

1. 编写了sub , call , push , test , je , cmp 六条指令
2. 在nemu中运行了第一个C程序

```
[hn@ubuntu:/media/psf/Home/ics2015$ make run
+ cc nemu/src/cpu/exec/arith/sub.c
+ ld obj/nemu/nemu
objcopy -S -O binary obj/testcase/mov-c entry
obj/nemu/nemu obj/testcase/mov-c
Welcome to NEMU!
The executable is obj/testcase/mov-c.
For help, type "help"
[(nemu) c
nemu: HIT GOOD TRAP at eip = 0x001000cf
```

```
Welcome to NEMU!
The executable is obj/testcase/add.
For help, type "help"
[(nemu) c
nemu: HIT GOOD TRAP at eip = 0x001000b5
```

3. 实现了add中还没有实现的功能

```
Welcome to NEMU!
The executable is obj/testcase/add.
For help, type "help"
[(nemu) c

Hit breakpoint at eip = 0x001000a4
```

4. 触发断点

```
Welcome to NEMU!
The executable is obj/testcase/add.
For help, type "help"
[(nemu) p test_data
0x101120:      1052960
[(nemu) p *test_data
0x0:          0
[(nemu) p (test_data+1)
0x101121:      1052961
[(nemu) p *(test_data+4)
0x1:          1
```

5. 为表达式求值添加变量支持

## 6. 打印栈帧链

```
[(nemu) c  
  
Hit breakpoint at eip = 0x00100023  
[(nemu) bt  
#0  0x00100024 in add  ( 0 , 0 , 0 , 0 )  
#1  0x00100069 in main ( )
```

## II.选答题:

## A. 立即数背后的故事:

Motorola 68k系列的处理器都是大端架构的, 现在问题来了, 考虑以下两种情况:

- 假设我们需要将NEMU运行在Motorola 68k的机器上(把NEMU的源代码编译成Motorola 68k的机器码)
- 假设我们需要编写一个新的模拟器NEMU-Motorola-68k, 模拟器本身运行在x86架构中, 但它模拟的是Motorola 68k程序的执行

在这两种情况下, 你需要注意些什么问题? 为什么会产生这些问题? 怎么解决它们?

当 NEMU 模拟的机器和运行 NEMU 本身的机器的字节序不同时, 在模拟访问内存时必须注意字节序的问题。NEMU 在读写模拟出的内存时, 由于模拟出的内存中内容 是以相异的字节序存储的, 直接读写会导致字节序错乱。因此在读写内存时需要专门 函数转换读取到的数值, 才能保证模拟出的内存与被模拟机器的字节序一致。

## B. 不能返回的main函数:

### 不能返回的main函数

为什么目前让用户程序从 main 函数返回就会发生错误? 这个错误具体是怎么发生的?

main 函数不能返回的根本原因是在调用 main 函数的指令之后没有其他指令了。浏览代码可以发现, 在 start.S 文件中包含了程序初始化相关的代码。该初始化代码在设置好栈指针后, 便使用 call 指令调用了 main 函数。然而该 call 指令之后再无其他指令。当 main 函数返回后, 会从 call 指令之后的指令处继续执行。事实上, 一般来说这条 call 指令之后会紧接着 C 语言代码中的第一个函数。因此, 在 main 函数返回后, 程序会以混乱的状态继续执行, 最终以访问越界或非法指令终止执行。

## C. 消失的符号:

我们在 add.c 中定义了宏 NR\_DATA, 同时也在 add() 函数中定义了局部变量 c 和形参 a, b, 但你会发现在符号表中找不到和它们对应的表项, 为什么会这样? 思考一下, 什么才算是一个符号(symbol)?

局部变量不能算作符号, 因为它们会在栈上分配空间, 地址并不固定, 不能也无需为它们分配符号。只有全局变量、静态变量以及函数才能作为符号。

## D. 寻找Hello world:

在GNU/Linux下编写一个Hello World程序, 编译后通过上述方法找到ELF文件的字符串表, 你发现"Hello World!"字符串在字符串表中的什么位置? 为什么会这样?

```

28: 0000000000000000      0 FILE      LOCAL  DEFAULT  ABS crtstuff.c
29: 00000000000600e20      0 OBJECT    LOCAL  DEFAULT  21 __JCR_LIST__
30: 00000000000400460      0 FUNC      LOCAL  DEFAULT  14 deregister_tm_clones
31: 000000000004004a0      0 FUNC      LOCAL  DEFAULT  14 register_tm_clones
32: 000000000004004e0      0 FUNC      LOCAL  DEFAULT  14 __do_global_ctors_aux
33: 00000000000601038      1 OBJECT    LOCAL  DEFAULT  26 completed.7585
34: 00000000000600e18      0 OBJECT    LOCAL  DEFAULT  20 __do_global_ctors_aux_fin
35: 00000000000400500      0 FUNC      LOCAL  DEFAULT  14 frame_dummy
36: 00000000000600e10      0 OBJECT    LOCAL  DEFAULT  19 __frame_dummy_init_array_
37: 0000000000000000      0 FILE      LOCAL  DEFAULT  ABS hello.c
38: 0000000000000000      0 FILE      LOCAL  DEFAULT  ABS crtstuff.c
39: 000000000004006f8      0 OBJECT    LOCAL  DEFAULT  18 __FRAME_END__
40: 00000000000600e20      0 OBJECT    LOCAL  DEFAULT  21 __JCR_END__
41: 0000000000000000      0 FILE      LOCAL  DEFAULT  ABS
42: 00000000000600e18      0 NOTYPE    LOCAL  DEFAULT  19 __init_array_end
43: 00000000000600e28      0 OBJECT    LOCAL  DEFAULT  22 __DYNAMIC
44: 00000000000600e10      0 NOTYPE    LOCAL  DEFAULT  19 __init_array_start
45: 000000000004005d4      0 NOTYPE    LOCAL  DEFAULT  17 __GNU_EH_FRAME_HDR
46: 00000000000601000      0 OBJECT    LOCAL  DEFAULT  24 __GLOBAL_OFFSET_TABLE__
47: 000000000004005b0      2 FUNC      GLOBAL  DEFAULT  14 __libc_csu_fini
48: 0000000000000000      0 NOTYPE    WEAK   DEFAULT  UND __ITM_deregisterTMCloneTab
49: 00000000000601028      0 NOTYPE    WEAK   DEFAULT  25 data_start
50: 00000000000601038      0 NOTYPE    GLOBAL  DEFAULT  25 _edata
51: 000000000004005b4      0 FUNC      GLOBAL  DEFAULT  15 _fini
52: 0000000000000000      0 FUNC      GLOBAL  DEFAULT  UND printf@@GLIBC_2.2.5
53: 0000000000000000      0 FUNC      GLOBAL  DEFAULT  UND __libc_start_main@@GLIBC_
54: 00000000000601028      0 NOTYPE    GLOBAL  DEFAULT  25 __data_start
55: 0000000000000000      0 NOTYPE    WEAK   DEFAULT  UND __gmon_start__
56: 00000000000601030      0 OBJECT    GLOBAL  HIDDEN  25 __dso_handle
57: 000000000004005c0      4 OBJECT    GLOBAL  DEFAULT  16 __IO_stdin_used
58: 00000000000400540      101 FUNC     GLOBAL  DEFAULT  14 __libc_csu_init
59: 00000000000601040      0 NOTYPE    GLOBAL  DEFAULT  26 _end
60: 00000000000400430      42 FUNC     GLOBAL  DEFAULT  14 _start
61: 00000000000601038      0 NOTYPE    GLOBAL  DEFAULT  26 __bss_start
62: 00000000000400526      22 FUNC     GLOBAL  DEFAULT  14 main
63: 0000000000000000      0 NOTYPE    WEAK   DEFAULT  UND _Jv_RegisterClasses
64: 00000000000601038      0 OBJECT    GLOBAL  HIDDEN  25 __TMC_END__
65: 0000000000000000      0 NOTYPE    WEAK   DEFAULT  UND __ITM_registerTMCloneTable
66: 000000000004003c8      0 FUNC      GLOBAL  DEFAULT  11 __init

```

```

[28] .shstrtab      STRTAB      0000000000000000  000018ce
      000000000000010c 0000000000000000          0      0      1
[29] .symtab         SYMTAB      0000000000000000  00001070
      0000000000000648 0000000000000018          30     47     8
[30] .strtab         STRTAB      0000000000000000  000016b8
      0000000000000216 0000000000000000          0      0      1

```

```
#include <stdio.h>

void main(){
    printf("hello, world!");
}
```

该字符串字面值不是符号，而是程序当中的数据。字符串表是保存符号名字字符串的地方，因此它不存在于字符串表中。事实上它应该存在于 `.rodata` 节中。

E. 丢失的信息：

在用户程序中定义以下字符数组：

```
char str[] = "abcdefg";
```

尝试通过上述方式输出 `str[1]` 的值，你发现有什么问题？运用现有的信息，你能够解决这个问题吗？如果能，请描述解决方法，并尝试实现；如果不能，请解释为什么，并尝试总结这背后反映的事实。

```
7f451e893000-7f451e895000 rw-p 00000000 00:00 0
7f451e895000-7f451e896000 r--p 00025000 08:01 1835103 /lib/x86_64-linux-gnu/ld-2.23.so
7f451e896000-7f451e897000 rw-p 00026000 08:01 1835103 /lib/x86_64-linux-gnu/ld-2.23.so
7f451e897000-7f451e898000 rw-p 00000000 00:00 0
7fff52220000-7fff52241000 rw-p 00000000 00:00 0 [stack]
7fff52364000-7fff52366000 r--p 00000000 00:00 0 [vvar]
7fff52366000-7fff52368000 r-xp 00000000 00:00 0 [vdso]
ffffffff600000-ffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
Makefile:62: recipe for target 'run' failed
make: *** [run] Aborted (core dumped)
hn@ubuntu:/media/psf/Home/ics2015$ p *str
p: command not found
```

符号表中存有符号的名字和地址，还有符号的大小、对齐等信息，但没有符号的类型信息。因此 NEMU 并不能知道这个符号是数值是什么类型，因此只能假定它是 `uint32_t` 而进行输出。



## F. 冗余的符号表:

在GNU/Linux下编写一个Hello World程序, 然后使用 `strip` 命令丢弃可执行文件中的符号表:

```
gcc -o hello hello.c
strip -s hello
```

用 `readelf` 查看hello的信息, 你会发现符号表被丢弃了, 此时的hello程序能成功运行吗?

目标文件中也有符号表, 我们同样可以丢弃它:

```
gcc -c hello.c
strip -s hello.o
```

用 `readelf` 查看hello.o的信息, 你会发现符号表被丢弃了. 尝试对hello.o进行链接:

## ❧器(2)

```
gcc -o hello hello.o
```

你发现了什么问题? 尝试对比上述两种情况, 并分析其中的原因.

```
hn@ubuntu:/media/psf/Home$ gcc -o hello hello.o
/usr/bin/ld: error in hello.o(.eh_frame); no .eh_frame_hdr table will be created
.
/usr/lib/gcc/x86_64-linux-gnu/5/../../../../x86_64-linux-gnu/crt1.o: In function `__start':
(.text+0x20): undefined reference to `main'
collect2: error: ld returned 1 exit status
```

可重定位目标文件之所以可重定位是因为有重定位表, 一旦删去了符号表, 重定位信息就丢失了, 自然就无法链接了。可执行文件可执行是因为有程序头表, 它存储的是程序如何装入内存的数据, 删除符号表并不会影响它, 因此删去可执行文件的符号表一般没有什么影响。

G. %ebp是必须的吗：

使用优化选项编译代码的时候, gcc会对代码进行优化, 会将 %ebp 当作普通的寄存器来使用, 不再让其作为指示当前的栈帧, 更多的信息可以查阅 man gcc 中的 -fomit-frame-pointer 选项. 我们使用 -O2

2)

来编译NEMU, 你可以对NEMU进行反汇编, 查看一些函数的代码. 在这种情况下, 代码要怎么找到函数调用的参数和局部变量?

另外优化 %ebp 寄存器之后, 就不能使用上述方法来打印栈帧链了. 如果你使用GDB对NEMU进行调试, 你会发现仍然可以使用bt命令来打印栈帧链. 你知道这是怎么做到的吗? 在优化 %ebp 寄存器之后, 为了打印栈帧链, 还需要哪些信息?

当没有 EBP 后, 程序必须通过 ESP 来访问局部变量。编译器会维护函数使用栈的情况, 并准确地计算出局部变量相对于 ESP 的偏移。我认为调试器在缺少 EBP 的情况下, 打印栈帧链是通过编译器附加的调试信息做到的。打印栈帧链的核心是找到函数的返回地址, 编译器只要将使用栈的信息或者返回地址的位置保存好, 调试器就可以找到返回地址, 进而分析出整个栈帧链。

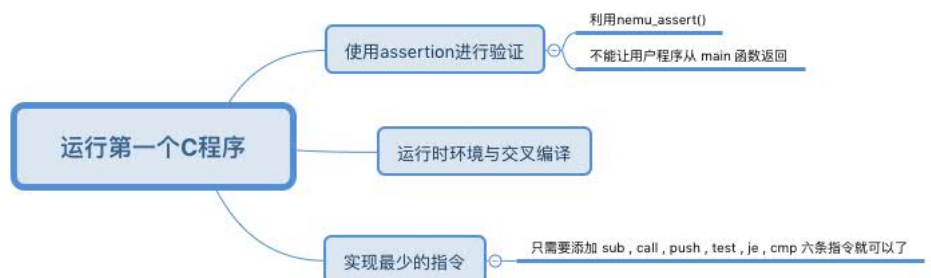
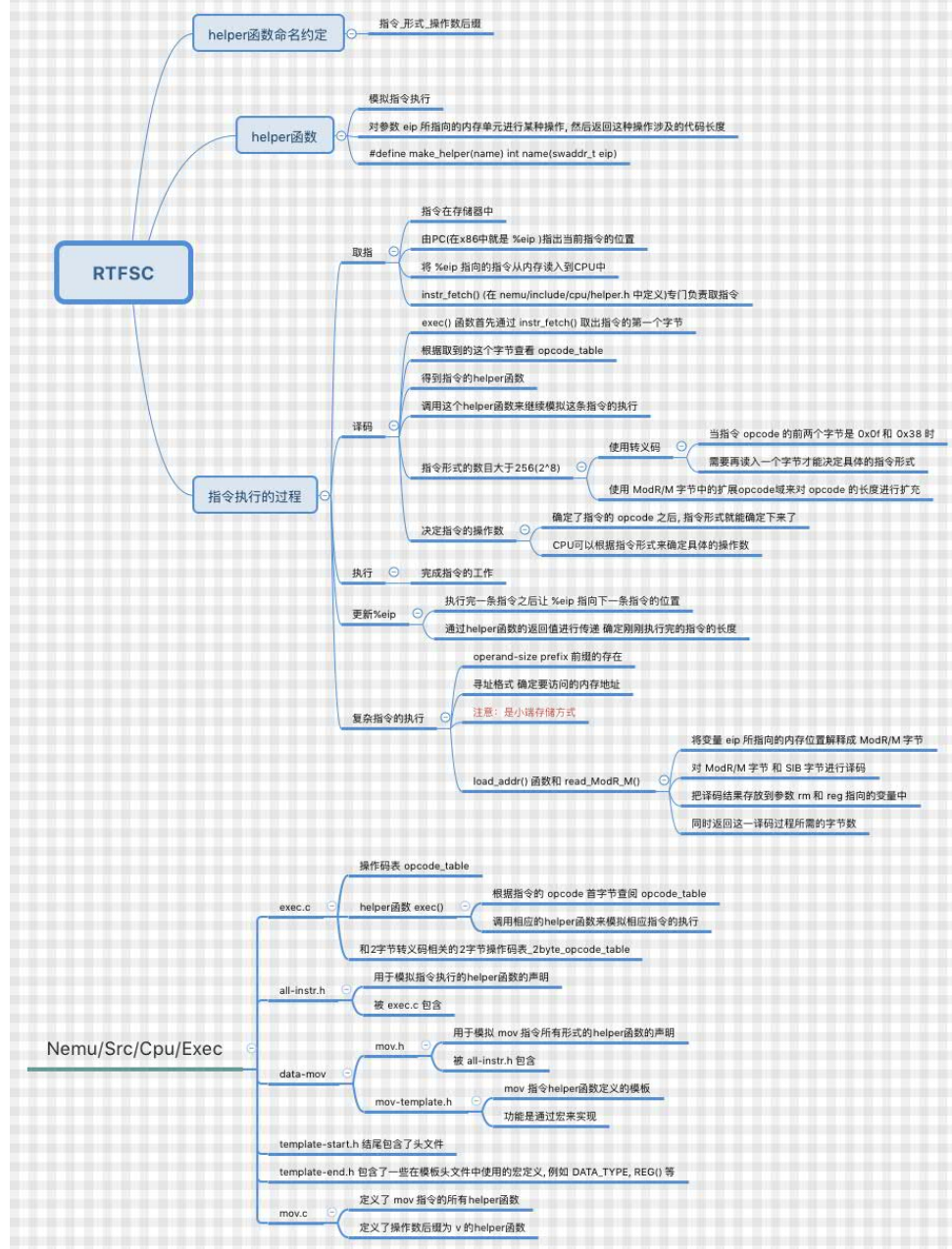
## III. 实验心得:

A. 首先按照惯例用思维导图理清思路。

B. 这一次的实验一开始让我很懵, 不知道应该写什么东西, 感觉要写的很多。光是列了思维导图都还感觉思路不是特别清晰。后来重新看了一次 assignment 之后发现其实需要实现的就是 sub.c/ sub.h/ sub.template 类似的这三个文件, 然后修改 exec.c 中的指令集。

C. 之后再进一步的开始阅读源码, 在源码的基础上进行修改。


D. 中途被一个玄学bug卡了超级久的时间一直不让我跑出来, 编译过了但是后面的执行就不会报错, 所以我一开始真的不知道是哪一句写错了。





```

(nemu) si
100000: bd 00 00 00 00      movl $0x0,%ebp
(nemu) si
100005: bc 00 00 00 08      movl $0x8000000,%esp
(nemu) si
10000a: 83 ec 10             sub $0x10,%esp
(nemu) c
invalid opcode(eip = 0x0010000d): e8 00 00 00 00 55 89 e5 ...
[
There are two cases which will trigger this unexpected exception:
1. The instruction at eip = 0x0010000d is not implemented.
2. Something is implemented incorrectly.
Find this eip value(0x0010000d) in the disassembling result to distinguish which case it is.

If it is the first case, see

for more details.

If it is the second case, remember:
* The machine is always right!
* Every line of untested code is always wrong!

nemu: nemu/src/cpu/exec/special/special.c:24: inv: Assertion `0' failed.
Makefile:62: recipe for target 'run' failed
make: *** [run] Aborted (core dumped)

```

E. 但是！单步执行大法好啊，所以单步执行之后发现我在e8那里卡住了。去对比mov-c.txt中发现是call语句的问题，再加上对比了exec.c中的指令集，发现原来是我指令那里没有声明。进行了修改之后就顺利的实现了。

### Disassembly of section .text:

```

00100000 <start>:
100000: bd 00 00 00 00      mov     $0x0,%ebp
100005: bc 00 00 00 08      mov     $0x8000000,%esp
10000a: 83 ec 10             sub     $0x10,%esp
10000d: e8 00 00 00 00      call    100012 <main>

```

```
155 /* 0xe8 */ call_i_v, inv, inv, inv,
```

更新于PA2的后半阶段:

I. 实验进度:

A. 我完成了PA2的后半阶段的全部部分。

1. 添加了指令。

```
[obj/testcase/sub-longlong](0.01 s): PASS!  
[obj/testcase/wanshu](4.27 s): PASS!  
[obj/testcase/struct](0.02 s): PASS!  
[obj/testcase/bubble-sort](0.48 s): PASS!  
[obj/testcase/mov-c](0.14 s): PASS!  
[obj/testcase/sum](0.00 s): PASS!  
[obj/testcase/add-longlong](0.01 s): PASS!  
[obj/testcase/leap-year](0.01 s): PASS!  
[obj/testcase/pascal](0.04 s): PASS!  
[obj/testcase/integral](0.01 s): PASS!  
[obj/testcase/mul-longlong](0.01 s): PASS!  
[obj/testcase/hello-str](Command terminated by signal 6  
0.02 s): FAIL! see hello-str-log.txt for more information  
[obj/testcase/min3](0.07 s): PASS!  
[obj/testcase/matrix-mul-small](0.12 s): PASS!  
[obj/testcase/fact](0.03 s): PASS!  
[obj/testcase/max](0.02 s): PASS!  
[obj/testcase/quadratic-eq](0.01 s): PASS!  
[obj/testcase/to-lower-case](0.01 s): PASS!  
[obj/testcase/prime](0.10 s): PASS!  
[obj/testcase/hello](Command terminated by signal 6  
0.02 s): FAIL! see hello-log.txt for more information  
[obj/testcase/bit](0.00 s): PASS!  
[obj/testcase/fib](0.00 s): PASS!  
[obj/testcase/string](0.01 s): PASS!  
[obj/testcase/shuixianhua](0.26 s): PASS!  
[obj/testcase/add](0.06 s): PASS!  
[obj/testcase/movsx](0.05 s): PASS!  
[obj/testcase/if-else](0.02 s): PASS!  
[obj/testcase/select-sort](0.33 s): PASS!  
[obj/testcase/quick-sort](0.36 s): PASS!  
[obj/testcase/gotbaha](0.19 s): PASS!  
[obj/testcase/matrix-mul](102.85 s): PASS!  
[obj/testcase/switch](0.36 s): PASS!  
[obj/testcase/hello-inline-asm](Command terminated by signal 6  
0.01 s): FAIL! see hello-inline-asm-log.txt for more information
```

B. 修改了lib.a。

```
[obj/testcase/hello-str](0.23 s): PASS!  
[obj/testcase/hello](Command terminated by signal 6
```

## II. 必答题:

- **编译与链接** 在 `nemu/include/cpu/helper.h` 中, 你会看到由 `static inline` 开头定义的 `instr_fetch()` 函数和 `idex()` 函数. 选择其中一个函数, 分别尝试去掉 `static`, 去掉 `inline` 或去掉两者, 然后重新进行编译, 你会看到发生错误. 请分别解释为什么会发生这些错误? 你有办法证明你的想法吗?

A. 去掉 `static`

链接时发生了错误。这是因为每一个包含 `helper.h` 的 `.c` 文件中都定义了一个名字叫做 `instr_fetch` 的函数实体, 且没有 `static` 说明符。因此每一个这样的 `.c` 对应的 `.o` 文件中都会有强符号 `instr_fetch` 出现。

链接时多个强符号冲突导致出错。

B. 去掉 `inline`

编译时发生了错误。这是因为在某一些包含了 `helper.h` 的 `.c` 文件中并没有用到 `instr_fetch` 这个函数。由于该函数带有 `static` 说明符, 且不带 `inline` 说明

• **编译与链接**

1. 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?
2. 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.
3. 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? (回答完本题后可以删除添加的代码.)

- A. 通过 `readelf -s obj/nemu/nemu` 可以看到最终生成的 `nemu` 的可执行文件的符号表, 只要统计这张表中 `dummy` 出现的次数即可知道生成的 `dummy` 实体的数量。

```
hn@ubuntu:/media/psf/Home/ics2015-Nemu$ readelf -s obj/nemu/nemu | grep ' dummy$'
' | wc -l
92
```

- B. 通过相同的方法得到实体的数量仍然为 92。这是因为所有引用了 `common.h` 的 `.c` 文件都引用了 `debug.h`, 并且所有引用了 `debug.h` 的 `.c` 文件都引



用了 common.h 文件。因此在同一个.c 文件中,出现了两句 volatile static int dummy, 在 C 语言规范中,这是一种“临时定义”,这样的“临时定义”允许出现多次,但是真正定义出来的实体只有一个,所以实体的数目并没有发生变化。

C. 这是因为出现两句 volatile static int dummy = 0 导致的重定义错误。

在 C 语言规范中,这是一种“定义”,在一个.c 文件中只能出现一次,因此是编译错误。

```
hn@ubuntu:/media/psf/Home/ics2015-Nemu$ make run
lib-common/Makefile.part:12: warning: overriding recipe for target 'obj/lib-common/FLOAT.a'
lib-common/Makefile.part:6: warning: ignoring old recipe for target 'obj/lib-common/FLOAT.a'
+ cc nemu/src/cpu/decode/decode.c
In file included from nemu/src/cpu/decode/decode.c:1:0:
nemu/include/common.h:35:21: error: redefinition of 'dummy'
In file included from nemu/include/common.h:12:0,
                 from nemu/src/cpu/decode/decode.c:1:
nemu/include/debug.h:35:21: note: previous definition of 'dummy' was here
nemu/Makefile.part:2: recipe for target 'obj/nemu/cpu/decode/decode.o' failed
make: *** [obj/nemu/cpu/decode/decode.o] Error 1
```

- 了解Makefile 请描述你在工程目录下敲入 make 后,make 程序如何组织.c和.h文件,最终生成可执行文件 obj/nemu/nemu.(这个问题包括两个方面: Makefile 的工作方式和编译链接的过程.) 关于 Makefile 工作方式的提示:
  - Makefile 中使用了变量,函数,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,也许对你有帮助
  - RTFM

通过阅读Makefile的内容可以发现,主Makefile引用了config目录下的 Makefile.build 文件。其中定义了名为 make\_common\_rules 的多行变量。在 nemu/Makefile.part 文件的第二行,利用 eval 函数和 call 函数,生成了对\$(nemu\_BIN)即 obj/nemu/nemu 的一系列生成规则。

仔细观察 make\_common\_rules 的内容可以发现,它首先通过调用 shell 命令执行 find -name "\*.c", 定义了变量\$(1)\_CFILES。即在这个变量里 存储了所有相关的.c 文件的文件名。接着又通过 patsubst 函数定义了 \$(1)\_COBJS, 它存储了\$(1)\_CFILES 变量中.c 对应的.o 的文件名。最后通过重写隐式规则,确定了由.c 生成.o 的命令。在 Makefile.build 的末尾处, 还用-include \$\$(\$(1)\_OBJS:.o=.d)解决了.c 与.h 文件之间的依赖关系。

由.c 生成.o 文件的规则是由 make\_common\_rules 生成的,而由.o 生成最终的可执行文件的规则,定义在 nemu/Makefile.part 中的第六行。可以看出,在依赖了所有.o 文件后,调用\$(CC)命令进行了链接,生成了最终的可执行文件。



## III. 选答题:

## A. Nemu的本质:

用x86指令来描述, 就是 `inc`, `dec` 和 `jnz` 三条指令. 假设除了输入变量之外, 其它变量的初值都是0, 并且假设程序执行到最后一条指令就结束, 你可以仅用这三种指令写一个计算两个正整数相加的程序吗?

```
while (x!=0) x--; a++;
while (y!=0) y--; a++;
a = x+y;

-----

L:
loop1:
JNZ x, loop 2
DEC x
INC a
JNZ b, loop 1 //令b恒为0 JNZ永远跳转

loop2:
JNZ y, done
DEC y
INC a
JNZ b, loop 2
```

## B. 捕捉死循环:

NEMU除了作为模拟器之外, 还具有简单的调试功能, 可以设置断点, 查看程序状态. 如果让你为NEMU添加如下功能

当用户程序陷入死循环时, 让用户程序暂停下来, 并输出相应的提示信息

你觉得应该如何实现? 如果你感到疑惑, 在互联网上搜索相关信息.

这是著名的停机问题, 数学家已经从理论上证明了不可能存在这样的程序. 但是, 对于内存、寄存器有限的机器(即总状态数有限), 从理论上是可以判定是否停机的. 只要记住程序执行中的所有状态, 一旦发现当前状态与历史某状态相同, 则可判定停机. 实际上, 这种方法不可能实现出来. 假设内存只有 1MB, 状态数就高达  $2^{8M}$ , 这个状态数大约是  $10^{2525222}$ , 已经远超宇宙中所有原子的数量  $10^{81}$  了.

## IV.实验心得:

A. 终于写完了终于写完了终于写完了!!!

B. 愿天堂没有NJU。

C. 好了下面是正经的:

1. 除了一大堆玄学bug以外中间碰到好几次坑

a) 比如d9指令需要gcc降级到4.7.4才能运行

```
hn@ubuntu:~$ gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/5/lto-wrapper
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 5.4.0-6ubuntu1~16.04.4' --with-bugurl=file:///usr/share/doc/gcc-5/README.Bugs --enable-languages=c,ada,c++,java,go,d,fortran,objc,obj-c++ --prefix=/usr --program-suffix=-5 --enable-shared --enable-linker-build-id --libexecdir=/usr/lib --without-included-gettext --enable-threads=posix --libdir=/usr/lib --enable-nls --with-sysroot=/ --enable-clocale=gnu --enable-libstdcxx-debug --enable-libstdcxx-time=yes --with-default-libstdcxx-abi=new --enable-gnu-unique-object --disable-vtable-verify --enable-libmpx --enable-plugin --with-system-zlib --disable-browser-plugin --enable-java-awt=gtk --enable-gtk-cairo --with-java-home=/usr/lib/jvm/java-1.5.0-gcj-5-amd64/jre --enable-java-home --with-jvm-root-dir=/usr/lib/jvm/java-1.5.0-gcj-5-amd64 --with-jvm-jar-dir=/usr/lib/jvm-exports/java-1.5.0-gcj-5-amd64 --with-arch-directory=amd64 --with-ecj-jar=/usr/share/java/eclipse-ecj.jar --enable-objc-gc --enable-multiarch --disable-werror --with-arch-32=i686 --with-abi=m64 --with-multilib-list=m32,m64,mx32 --enable-multilib --with-tune=generic --enable-checking=release --build=x86_64-linux-gnu --host=x86_64-linux-gnu --target=x86_64-linux-gnu
Thread model: posix
gcc version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.4)
```

```
gcc version 4.7.4 (Ubuntu/Linaro 4.7.4-3ubuntu12)
```

b) 比如同样的代码在别人的电脑下跑几秒钟 在我电脑上跑几分钟

c) 比如第一天晚上跑了半个小时的矩阵乘法函数...

d) 比如在修改lib的时候的.text的偏移量是40 所以之后的代码的地址都要加上40才能修改

Section Headers:										
[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[ 0]		NULL	00000000	000000	000000	00		0	0	0
[ 1]	.text	PROGBITS	00000000	000040	001ef9	00	AX	0	0	16