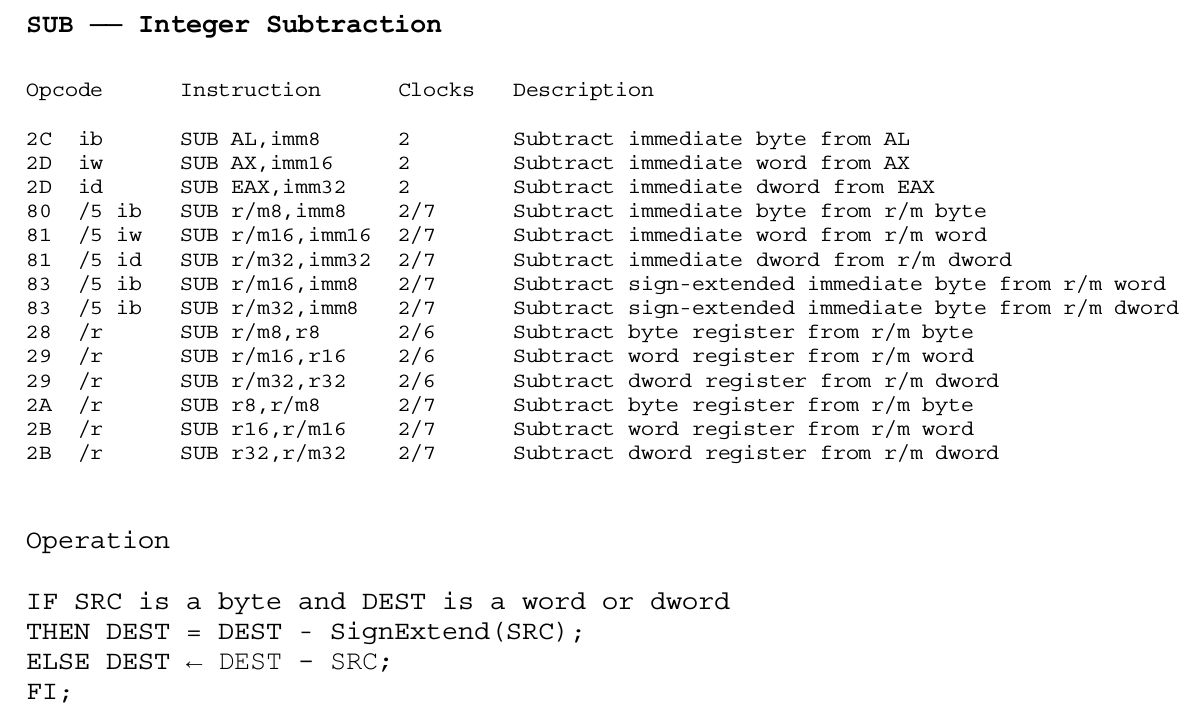
PA2前半阶段实验报告

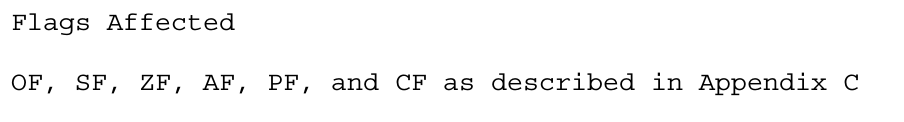
　　　　　　　　　　　　　　　　　　 　姓名：熊倩　　学号：15307130228

PA2阶段1: 运行用户程序mov-c

需要实现sub , call , push ,test , je , cmp这六条指令，下面依次对这六条指令进行分析：

sub指令：先在i386手册中找到sub指令如下：





从手册中可以看到，sub的dest操作数有eax，r/m8,16,32，r8,16,32,三大类，src操作数有立即数imm8,16,32，r/m8,16,32，r8,16,32三大类，其中8,16,32分别表示操作数有8位，16位，32位。根据讲义中的命名规则：指令\_形式\_操作数后缀，把sub指令分为sub\_i2a，

sub\_i2rm，sub\_r2rm，sub\_rm2r，sub\_si2rm这五种。这五种指令中一个比较特别的是sub\_si2rm，该形式需要对立即数进行符号扩展。

在手册中找到信息后，需要在exec.c文件中找到每种形式对应的opcode，并填入，从i386手册中可以看到：

sub\_i2a\_b的opcode为2C，sub\_i2a\_w的opcode为2D，sub\_i2rm\_b的opcode为80，sub\_i2rm\_v的opcode为81（后缀为v表示操作数长度还不能确定,可能是16位或者32位, 需要通过 ops\_decoded.is\_data\_size\_16 成员变量来决定），sub\_si2rm\_v的opcode为83，sub\_r2rm\_b的opcode为28，sub\_r2rm\_v的opcode为29，sub\_rm2r\_b的opcode为2A，sub\_rm2r\_v的opcode为2B。

在exec.c文件的opcode\_table中填入指令对应的opcode后，需要继续写出其对应的helper函数来具体执行该指令，所以在exec文件夹中的arith文件夹中新建sub.c，sub.h，sub-template.h三个文件，sub.c文件中填写make\_helper\_v函数的定义：make\_helper\_v(sub\_i2a)，make\_helper\_v(sub\_i2rm)，make\_helper\_v(sub\_r2rm)，make\_helper\_v(sub\_rm2r)，make\_helper\_v(sub\_si2rm)；sub.h文件中填写make\_helper函数的定义如下：

make\_helper(sub\_i2a\_b);make\_helper(sub\_i2rm\_b);make\_helper(sub\_r2rm\_b);make\_helper(sub\_rm2r\_b);make\_helper(sub\_si2rm\_b);make\_helper(sub\_i2a\_v);make\_helper(sub\_i2rm\_v);make\_helper(sub\_r2rm\_v);make\_helper(sub\_rm2r\_v);make\_helper(sub\_si2rm\_v);

该定义覆盖了上面sub指令的五种形式，每种形式包含\_b,\_w,\_l三种不同长度的指令后缀。

而具体的指令执行内容需要在sub-template.h文件中写：

#define instr sub

make\_instr\_helper(i2a)

make\_instr\_helper(i2rm)

make\_instr\_helper(r2rm)

make\_instr\_helper(rm2r)

#if DATA\_BYTE == 2 || DATA\_BYTE == 4

make\_instr\_helper(si2rm)

#endif

上面的定义与下面的宏有关：

#define make\_instr\_helper(type) \

make\_helper(concat5(instr, \_, type, \_, SUFFIX)) { \

return idex(eip, concat4(decode\_, type, \_, SUFFIX), do\_execute); \

}

15 static inline int idex(swaddr\_t eip, int (\*decode)(swaddr\_t), void (\*execute) (void)) {

16 /\* eip is pointing to the opcode \*/

17 int len = decode(eip + 1);

18 execute();

19 return len + 1; // "1" for opcode

20 }

其表示make\_instr\_helper函数被拆分为eip，concat4(decode\_, type, \_, SUFFIX)，do\_execute三个部分，然后用写好的各种形式的decode函数从eip + 1处（跳过opcode）开始解码，解码后执行do\_execute函数，最后返回包括opcode的指令的总长度。

一个需要注意的地方是，如果形式是si2rm，那么就有符号扩展，rm中的目标操作数就只能是16位或者32位，如果是8位就谈不上扩展，所以有

#if DATA\_BYTE == 2 || DATA\_BYTE == 4

make\_instr\_helper(si2rm)

#endif

例如r2rm形式的解码函数如下，最终可以得到dest操作数op\_dest和src操作数op\_src，得到之后，就可以在do\_execute函数中具体对操作数进行操作，根据386手册中的描述执行该指令。

static int concat3(decode\_rm\_, SUFFIX, \_internal) (swaddr\_t eip, Operand \*rm, Operand \*reg) {

rm->size = DATA\_BYTE; //DATA\_BYTE means rm's size

int len = read\_ModR\_M(eip, rm, reg);

reg->val = REG(reg->reg);

#ifdef DEBUG

snprintf(reg->str, OP\_STR\_SIZE, "%%%s", REG\_NAME(reg->reg));

#endif

return len;

}

/\* Eb <- Gb

\* Ev <- Gv

\*/

make\_helper(concat(decode\_r2rm\_, SUFFIX)) {

return decode\_rm\_internal(eip, op\_dest, op\_src);

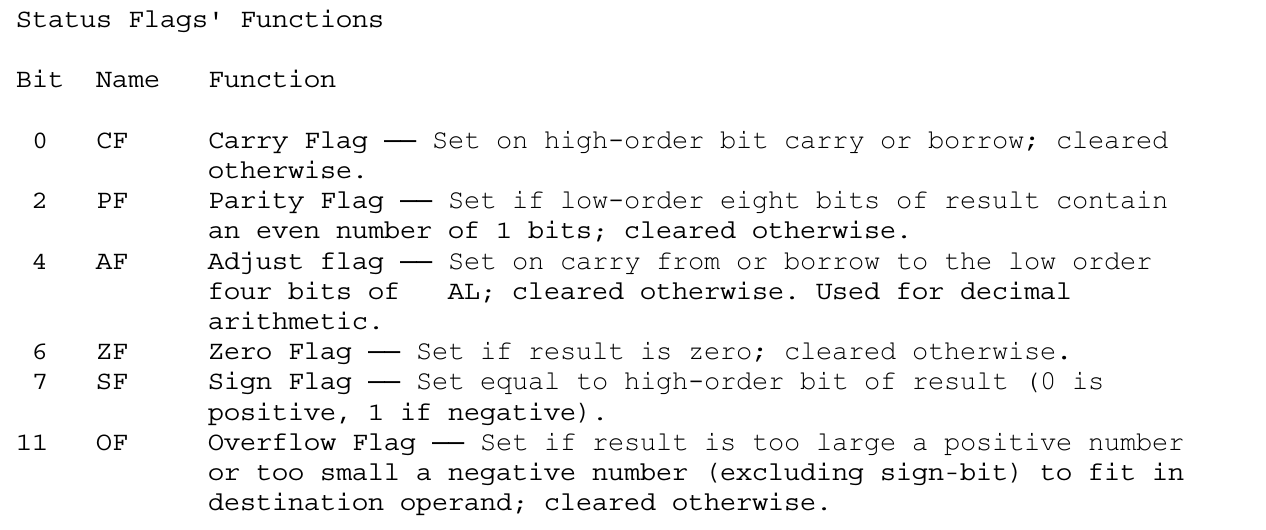
}

接着，需要填写do\_execute函数，这需要做两件事，一个是对操作数进行操作，一个是对flags进行处理，对sub指令而言，需要把op\_dest操作数-op\_src操作数的结果写回到op\_dest操作数中，通过下面两语句来实现：

DATA\_TYPE result = (op\_dest->val)-(op\_src->val);

OPERAND\_W(op\_dest, result);

对flags的处理包括SF,ZF,OF,CF,PF五种flag，在386手册中的描述如下：



对SF，即把运算结果的最高位得到即可

int len = (DATA\_BYTE << 3) - 1;

cpu.SF=result >> len;

对ZF, 若result为0，ZF为1，否则为0，通过非运算实现: cpu.ZF=!result;

对OF, 若两操作数符号不相同，且op\_dest操作数减后变号，则溢出：

s1=op\_dest->val>>len;

s2=op\_src->val>>len;

cpu.OF=(s1 != s2 && s2 == cpu.SF) ;

对CF，若op\_dest->val < op\_src->val，则发生借位：

cpu.CF = op\_dest->val < op\_src->val;

对PF，需要对运算结果进行递归运算，从而得出PF：

result ^= result >>4;

result ^= result >>2;

result ^= result >>1;

cpu.PF=!(result & 1);

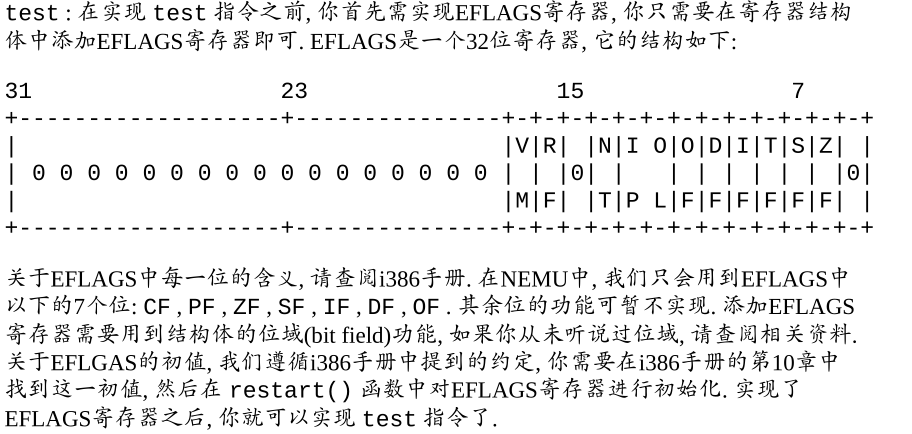
最后需要在all-instr.h中包含sub.h头文件，sub指令就实现好了。

call , push ,test , je , cmp指令也类似，唯一不同的是opcode可能在group中，group表示多个指令形式共用一个opcode，并通过ModR/M 字节中的reg/opcode 域来确定是哪个指令。下面就只描述每个指令的do\_execute函数和对flags的更新。

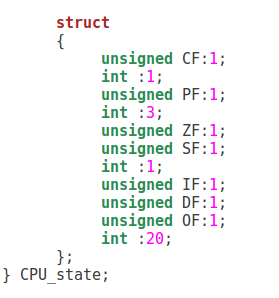
call指令：call指令在contran（表示Control Transfer Instructions）文件夹中，首先用concat(decode\_i\_, SUFFIX) (eip + 1)对指令解码，并找出指令长度，然后ESP寄存器减去DATA\_BYTE个长度，来填写call后面一个指令的eip地址，swaddr\_write (reg\_l (R\_ESP) , 4 , cpu.eip + len)，跳转长度displacement = op\_src->val，最后cpu.eip +=displacement，call指令执行完毕后，eip还会增加整个call指令的总长度len+1。call指令并不影响flags。

push指令：call指令在data-mov文件夹中，ESP寄存器减去DATA\_BYTE个长度，来填写源操作数：MEM\_W(cpu.esp, op\_src->val)，然后执行print\_asm\_template1()进行输出提示。push指令并不影响flags。

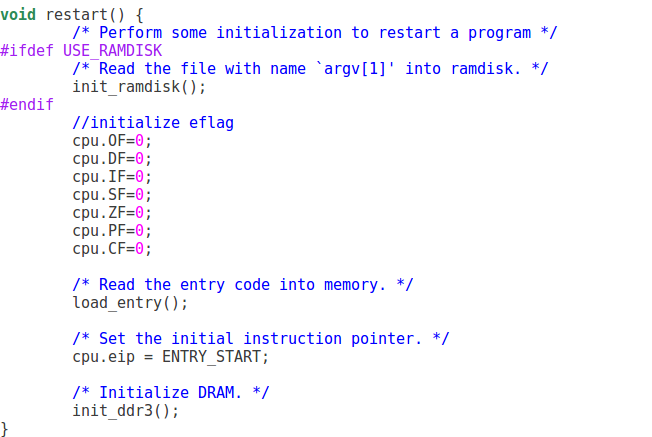
test指令:



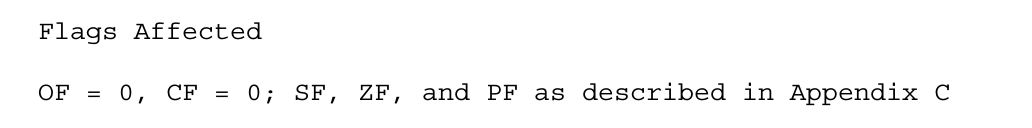
先实现EFLAGS寄存器：



并初始化如下：



test指令在logic文件夹中表示逻辑运算，其结果没有影响，只用于更新flags标志。 先算出两操作数相与的结果DATA\_TYPE result = (op\_dest->val)&(op\_src->val)，然后根据386手册中：



所以各标志如下：

cpu.CF=0;

cpu.OF=0;

int len = (DATA\_BYTE << 3) - 1;

cpu.SF=result >> len;

cpu.ZF=!result;

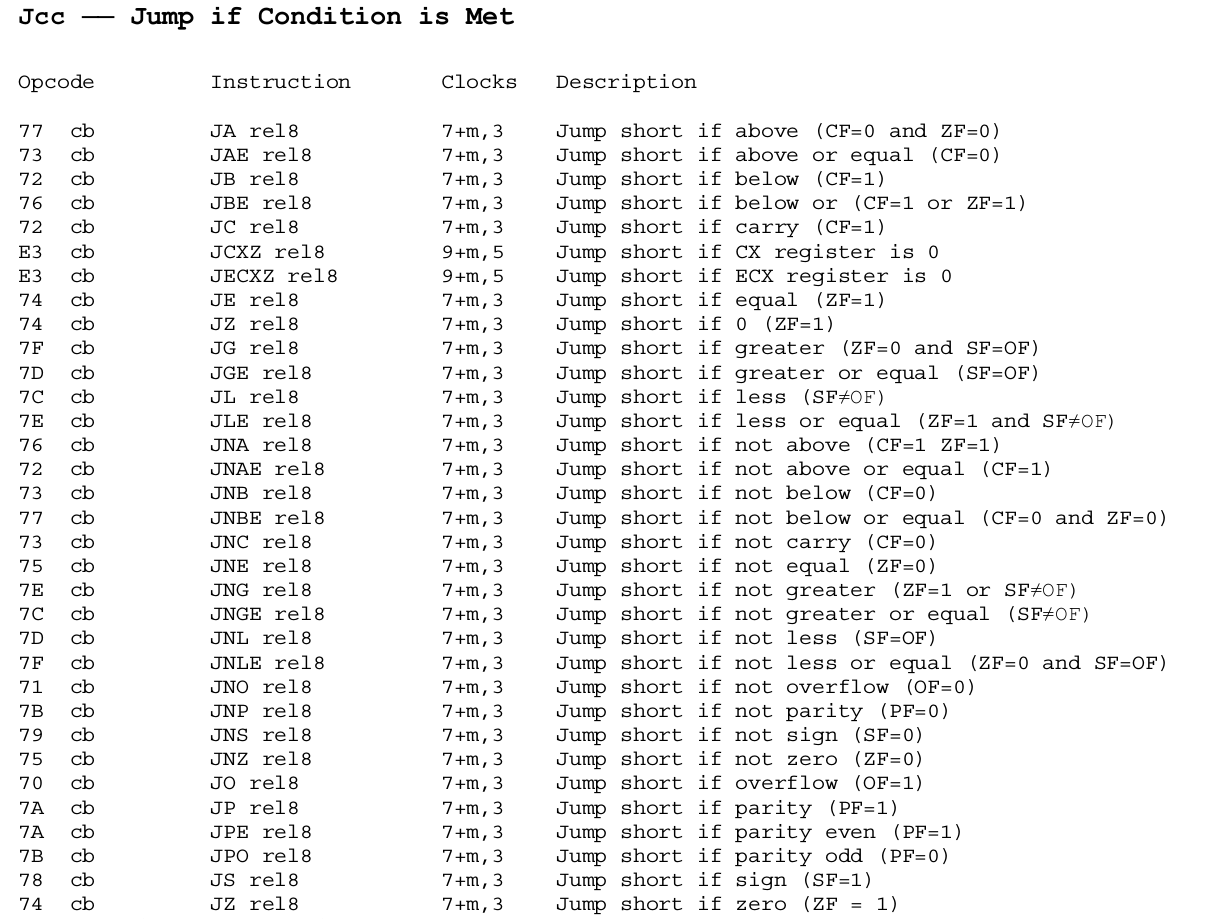
result ^= result >>4;

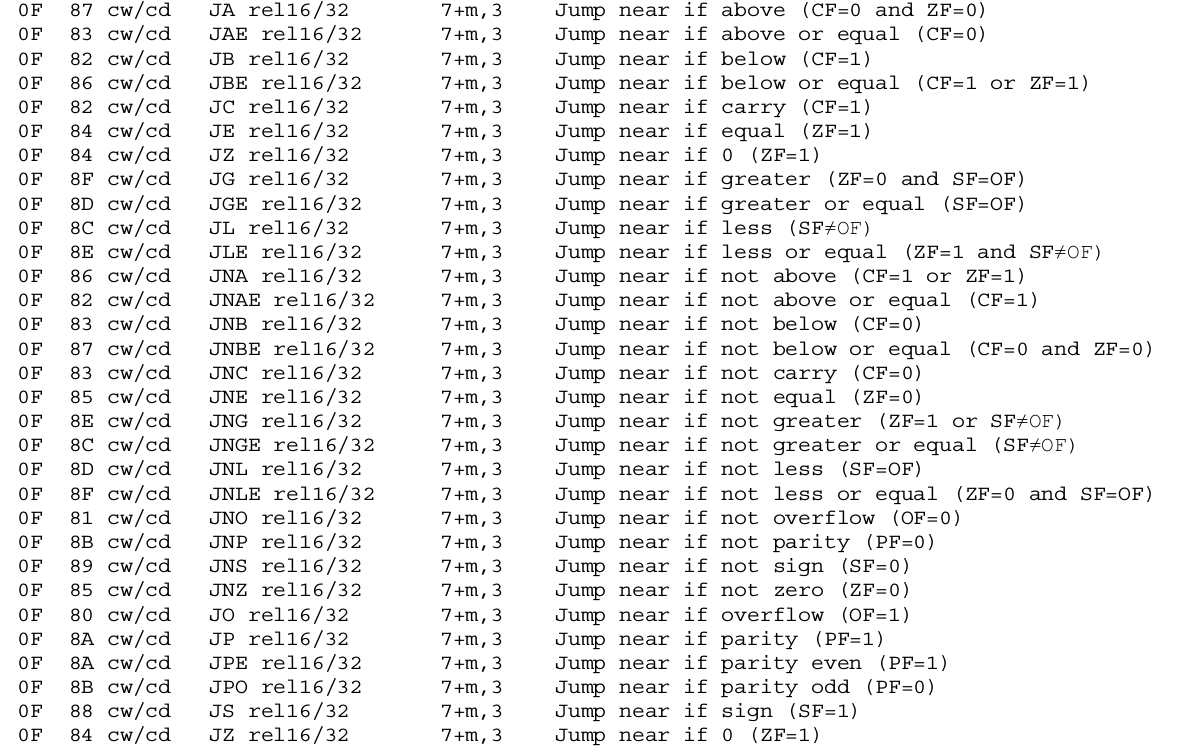
result ^= result >>2;

result ^= result >>1;

cpu.PF=!(result & 1);

je指令：je指令在contran文件夹中，je是jcc的一种形式，在386手册中如下：





这里用jcc.h对jcc的各种形式打包，包含ja.h等头文件，只要在all-instr.h中包含jcc.h即可。实现jcc的各种指令只要根据其对应的条件依次实现即可，这里不全部一一写出，只举例如下：

jbe指令:如果(cpu.ZF == 1)或者(cpu.CF == 1)的话，就eip就增加op\_src操作数，op\_src操作数是立即数， cpu.eip= cpu.eip + (DATA\_TYPE\_S)op\_src->val，否则eip不变。

cmp指令：cmp指令在data-mov文件夹中，该指令用op\_dest操作数减去op\_src操作数，并通过结果来影响flags：

cpu.CF = op\_dest->val < op\_src->val;

int len = (DATA\_BYTE << 3) - 1;

cpu.SF=result >> len;

int s1,s2;

s1=op\_dest->val>>len;

s2=op\_src->val>>len;

cpu.OF=(s1 != s2 && s2 == cpu.SF) ;

cpu.ZF=!result;

result ^= result >>4;

result ^= result >>2;

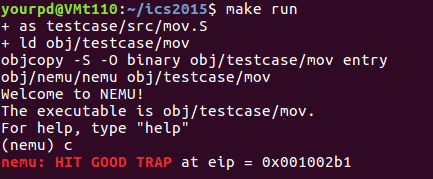
result ^= result >>1;

cpu.PF=!(result & 1);

print\_asm\_template2();

上面就是mov程序所要用到的指令。

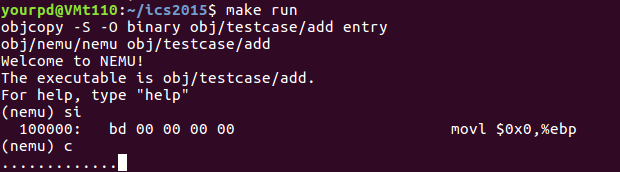
最后附上运行结果：



PA2阶段2简易调试器(2)：

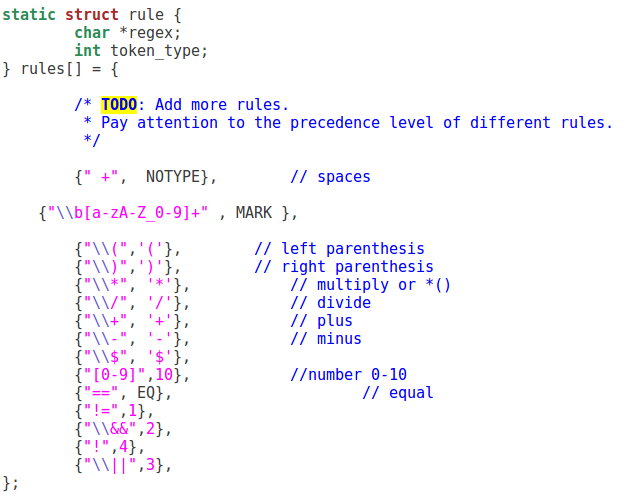
运行用户程序add

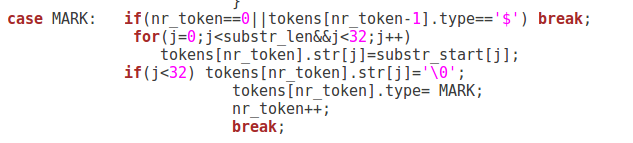
如上所述，根据386手册继续实行指令，使add程序能够运行：



为表达式求值添加变量的支持

需要在表达式求值的词法分析和递归求值中添加对变量的识别和处理：

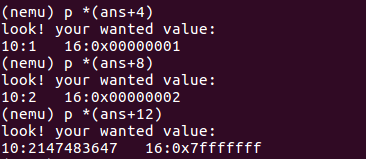




最后在eval函数中求出变量名对应的地址：在表达式递归求值的过程中, 如果发现token的类型是一个标识符, 就通过这个标识符在符号表中找到一项符合要求的表项(表项的 Type 属性是 OBJECT , 并且将 Name 属性的值作为字符串表中的偏移所找到的字符串和标识符的命名一致), 找到标识符的地址, 并将这个地址作为结果返回。



运行结果如图：



打印栈帧链：

bt实现如下：



运行结果如下：



前半阶段结束。

必答题：

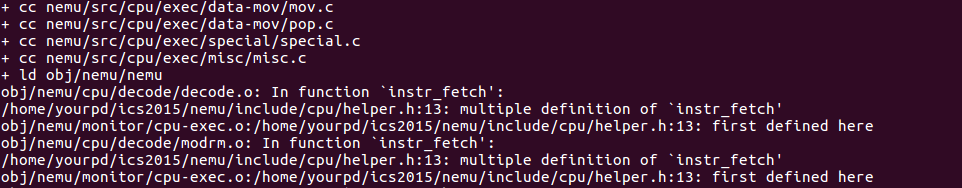
编译与链接 在 nemu/include/cpu/helper.h 中, 你会看到由 static

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?

把instr\_fetch() 函数的static和inline修饰去掉，结果如下：



提示是instr\_fetch() 函数多重定义，因为nemu/include/cpu/helper.h 是头文件，然而在头文件中却放入了instr\_fetch() 函数和 idex() 函数的定义，所以在其它文件引用nemu/include/cpu/helper.h时，函数的定义又被放在其它文件中，所以产生了多重定义的问题，而且可以看到，报错的都是包含nemu/include/cpu/helper.h头文件的文件，而如果加入inline修饰，则使用函数地方就直接替换成函数定义，这样函数定义如果不在头文件中，就无法替换成函数定义，所以内联函数可以在程序中定义多次；如果加入static修饰，那么各个文件中的相应函数都是static的，只在单个文件中起作用，就不会产生冲突。

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?

使用命令readelf –s 0bj/nemu/nemu可以得到可执行文件的符号表，再对符号表进行正则操作|grep “dummy$”|wc可以得到总数为83



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.



总数没有改变，由于common.h文件中包含debug.h文件，而nemu中的文件只包含common.h文件，不包含debug.h文件，所以在debug.h文件中加不加上述语句，包含common.h文件中都有volatile static int dummy;语句，所以结果不变。

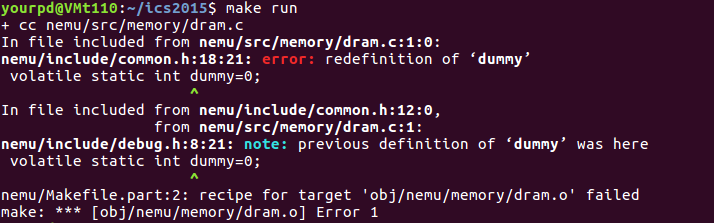


3. 修改添加的代码, 为两处 dummy 变量进行初始化: volatile static

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?

运行结果出现重定义错误：



因为在debug.h文件中已经通过volatile static int dummy = 0;对dummy进行了定义，如果在common.h文件中在对dummy进行初始化，就会产生重定义的问题；而之前没有报重定义的错是因为dummy没有进行初始化，static变量可以在多个文件中定义。

Makefile 的工作方式：

Makefile可以包含变量, 函数,和文件，如：

CC := gcc

include config/Makefile.git //包含的Makefile文件也是整个工程的Makefile的一部分

nemu: $(nemu\_BIN)

一般的命令包含目标文件，依赖文件和命令，例如：

entry: $(ENTRY)

objcopy -S -O binary $(ENTRY) entry

其中，entry是目标文件，$(ENTRY)是依赖文件，objcopy -S -O binary $(ENTRY) entry是命令。

Makefile 的编译链接：

make会找文件中的第一个目标文件（target），并把这个文件作为最终的目标文件，在nemu的Makefile中，执行make，该目标文件是nemu（nemu: $(nemu\_BIN)）。

如果nemu文件不存在，或是nemu所依赖的后面的 文件的文件修改时间要比nemu这个文件新，那么，他就会执行后面所定义的命令来生成nemu这个文件。这样，make又将去先去生成nemu所依赖的后面的 文件，就依次推导下去，直至生成nemu可执行文件。