实验报告



PA1-简易调试器

学号 6319000359

姓名 张明君-留学生

|  |
| --- |
| 实验进度（任务自查表） |
| **必做任务1：实现正确的寄存器结构体**  在现阶段的NEMU 中通用寄存器为：  32位寄存器：EAX , EDX , ECX , EBX , EBP , ESI , EDI , ESP  16位寄存器：AX , DX , CX , BX , BP , SI , DI , SP  8 位寄存器：AL , DL , CL , BL , AH , DH , CH , BH  但它们在物理上并不是相互独立的, 例如 EAX 的低 16 位是 AX , 而 AX 又分成 AH 和 AL。因此EAX寄存器结构图如下（图中没有标出AH）：    在reg.h文件中的源代码里，用struct结构定义寄存器。查阅资料可以知道struct和union的区别：   * struct和union都是由多个不同的数据类型成员组成, 但在任何同一时刻, union中只存放了一个被选中的成员, 而struct的所有成员都存在。在struct中，各成员都占有自己的内存空间，它们是同时存在的。一个struct变量的总长度等于所有成员长度之和。在Union中，所有成员不能同时占用它的内存空间，它们不能同时存在。Union变量的长度等于最长的成员的长度。 * 对于union的不同成员赋值, 将会对其它成员重写, 原来成员的值就不存在了, 而对于struct的不同成员赋值是互不影响的。   由此可以看出寄存器的特征符合联合体，修改后的代码为：   |  | | --- | | **typedef** **struct** {  **union** {  **union** {  **uint32\_t** \_32;  **uint16\_t** \_16;  **uint8\_t** \_8[**2**];  } gpr[**8**];  /\* Do NOT change the order of the GPRs' definitions. \*/  **struct** {  **uint32\_t** eax, ecx, edx, ebx, esp, ebp, esi, edi;  };  };  **swaddr\_t** eip;  }CPU\_state; |   Make run 运行出来的结果：    **必做任务2：实现单步执行、打印寄存器、扫描内存**  **实现单步执行**  单步执行的格式为si [N]，程序单步执行N条指令后暂停, 当N没有给出时, 缺省为默认为1。根据单步执行的说明得出解题步骤：   1. 传入cmd\_si()函数的参数为字符串，现在需要利用一些方法将其分解为两部分，分别为“si （空格）”和“N”(N是字符串类型的数字），N的部分存到字符串arg中，此过程中需要用到strtok()库函数。  |  | | --- | | C 库函数 **char** \*strtok(**char** \*str, **const** **char** \*delim) 分解字符串 str 为一组字符串，delim 为分隔符。 |  1. 根据字符串arg来判断需要执行的指令数 i，需要使用sscanf()库函数，将字符串arg改为int型的数字 i。  |  | | --- | | C 库函数 **int** sscanf(**const** **char** \*str, **const** **char** \*format, ...) 从字符串读取格式化输入 |   **参数**  str -- 这是 C 字符串，是函数检索数据的源。  format -- 这是 C 字符串，包含了以下各项中的一个或多个：空格字符、非空格字符 和 format 说明符。  若arg为NULL，默认cpu\_exec(1)  若 i < -1, 提示Parameter error  若 i = -1, 调用次 cpu\_exec(-1)  若 i > -1, 调用次 cpu\_exec(steps)  进入~/NEMU2021/nemu/src/monitor/debug/ui.c 附上代码，代码如下：   |  | | --- | | **static** **int** **cmd\_si**(**char** \*args){  **char** \*arg = strtok(NULL," ");  **int** i=**0**;  **int** j;  **if**(arg == NULL){  cpu\_exec(**1**);  **return** **0**;  }  sscanf(arg,"%d",&i);  **if**(i<-**1**){  printf("Parameter error**\n**");  **return** **0**;  }  **if**(i==-**1**){  cpu\_exec(-**1**);  }  **for**(j=**0**;j<i;j++){  cpu\_exec(**1**);  }  **return** **0**;  } |   在static struct 添加命令(cmd\_si [N])，代码如下：   |  | | --- | | **static** **struct** {  **char** \*name;  **char** \*description;  **int** (\*handler) (**char** \*);  } cmd\_table [] = {  { "help", "Display informations about all supported commands", cmd\_help },  { "c", "Continue the execution of the program", cmd\_c },  { "q", "Exit NEMU", cmd\_q },  { "si","Args:[N]; Eg:si 10; Execute [N] instructions step by step",cmd\_si},  }; |   实验运行出来的结果：    **打印寄存器**  打印程序状态的命令格式为info SUBCMD ，当SUBCMD为 r 时info r打印印寄存器状态，只需要printf每一个寄存器的状态。  设计代码如下(ui.c)：   |  | | --- | | **static** **int** **cmd\_info**(**char** \*args){  **if**(args[**0**]=='r'){  **int** i;  **for**(i = **0**; i <= R\_EDI; i++) {  printf( "$%s**\t**0x%08x**\t**%d**\n**", regsl[i], reg\_l(i), reg\_l(i));  }  printf( "$eip**\t**0x%08x**\t**%d**\n**", cpu.eip, cpu.eip );  }  **else** **if**(args[**0**]=='w'){  // info\_wp();  }**else** {printf("the argument must be r or w");}  **return** **0**;  } |   // info wp() ; 先标记 后面还要用 info w  在static struct 添加命令(cmd\_info\_r)，代码如下：   |  | | --- | | **static** **struct** {  **char** \*name;  **char** \*description;  **int** (\*handler) (**char** \*);  } cmd\_table [] = {  { "help", "Display informations about all supported commands", cmd\_help },  { "c", "Continue the execution of the program", cmd\_c },  { "q", "Exit NEMU", cmd\_q },  { "si","Args:[N]; Eg:si 10; Execute [N] instructions step by step",cmd\_si},  { "info","Args:r/w; Eg:info r; Print information about registers or watchpoint", cmd\_info},  }; |   实验运行出来的结果：    **扫描内存**  查阅实验手册知道，访问内存的接口函数相关的源代码存在memory.c文件中，其中lnaddr\_read和lnaddr\_write两个函数用来对内存进行读写，lnaddr\_read函数需要传入两个参数，分别为起始地址和扫描长度。  Memory.c 中参考的代码：   |  | | --- | | **uint32\_t** **lnaddr\_read**(**lnaddr\_t** addr, **size\_t** len) {  **return** hwaddr\_read(addr, len);  }  **void** **lnaddr\_write**(**lnaddr\_t** addr, **size\_t** len, **uint32\_t** data) {  hwaddr\_write(addr, len, data);  } |   内存扫描命令的格式为x N EXPR，N表示扫描长度，EXPR为起始内存。因此得出解题步骤：  1）传入cmd\_x()函数的参数为字符串，需要利用strtok()函数分别得到 N 和 EXPR 部分的字符串，再利用sscanf()函数将字符串 N 转化为十进制整型数 len，把字符串EXPR转化为十六进制的数address。  2）任务中要求以16进制 形式输出连续的N个4字节，因此，将address和4传入lnaddr\_read函数就可以得到，再用for循环循环len次，每次循环时起始地址加4,就可以实现内存的扫描。  代码如下（ui.c）:   |  | | --- | | **static** **int** **cmd\_x**(**char** \*args){  **int** num;  **char** \*arg1=strtok(NULL," ");  sscanf(arg1,"%d",&num);  **swaddr\_t** address;  **char** \*arg2=strtok(NULL," ");  **bool** success;  address=expr(arg2,&success);  **if**(!success) {printf("expression is wrong**\n**");**return** **0**;}  printf("start address is 0x%08x**\n**",address);  **int** i;  **for**(i=**0**;i<num;i++){  printf("address is 0x%08x,value is 0x%08x**\n**",address,swaddr\_read(address,**4**));  address+=**4**;  }  **return** **0**;  } |   在static struct 添加命令(cmd\_x)，代码如下：   |  | | --- | | { "x","scan memory",cmd\_x}, |   实验运行出来的结果：    与mov.txt文件中的内容比较，结果一致的：    **必做任务3：实现算术表达式的词法分析**  想要求出表达式的值，第一步要解决的问题是识别字符串中的数字、符号、括号等等，解决方法是利用正则表达式刻画字符的组合规律，将字符串切割成一个个的有确定类型的token。  表达式中可能出现的类型：   1. 数字：十进制 ，十六进制 … 2. 运算符：+，-，\*，/，（… 3. 符号：test\_case，… 4. 寄存器：$ eax，$ edx，…   利用正则表达式的规则补充rules[]，其中要特别注意，如果识别的符号为正则表达式的元符号则需要加上\符号,代码如下：   |  | | --- | | **static** **struct** rule {  **char** \*regex;  **int** token\_type;  **int** precedence;  } rules[] = {  /\* TODO: Add more rules.  \* Pay attention to the precedence level of different rules.  \*/  {"**\\**b[0-9]+**\\**b",DIGIT,**0**}, // degit  {"**\\**b0x[a-fA-F0-9]+**\\**b",HEX,**0**}, // hex  {"**\\**$[a-zA-Z]+",REGISTER,**0**}, // register  {" +",NOTYPE,**0**}, // tab  {" +", NOTYPE,**0**}, // spaces  {"**\\**+", '+',**4**}, // plus  {"-",'-',**4**}, // minus  {"**\\**\*",'\*',**5**}, // multiple  {"/",'/',**5**}, // divide  {"!",'!',**6**}, // not  {"&&",AND,**2**}, // and  {"**\\**|**\\**|",OR,**1**}, // or  {"==", EQ,**3**}, // equal  {"!=",NEQ,**3**}, // not equal  {"**\\**(",'(',**7**}, // left (  {"**\\**)",')',**7**}, // right )  }; |   扩充完正则表达式规则以后，需要做的就是对输入的字符串进行分析，对每一个符号进行分类，再将各个类型存储在tokens[]数组中，完成此操作的函数为make\_token()函数。已给出代码的部分可以成功识别得到该字符或者字符串的对应规则，而我们需要补充的部分是switch语句，switch语句将表达式中每一个部分用对应的类型及具体值存储到tokens[nr\_token].str中（如NUM类型里存具体的数字，REGISTER类型里存具体的寄存器的名字等等）。  补充代码如下：   |  | | --- | | **switch**(rules[i].token\_type) {  **case** NOTYPE: **break**;  **case** REGISTER:  tokens[nr\_token].type=rules[i].token\_type;  tokens[nr\_token].precedence=rules[i].precedence;  strncpy(tokens[nr\_token].str,for\_register,substr\_len-**1**);  tokens[nr\_token].str[substr\_len-**1**]='\0';  nr\_token++;  **break**;  **default:**  tokens[nr\_token].type=rules[i].token\_type;  tokens[nr\_token].precedence=rules[i].precedence;  strncpy(tokens[nr\_token].str,substr\_start,substr\_len);  tokens[nr\_token].str[substr\_len]='\0';  nr\_token++;  } |   **必做任务4：实现算术表达式的递归求值**  通过任务3，我们已将token存入到了tokens[]数组中，接下来需要用递归的方法求出表达式的值,此功能在eval()函数中实现。实验手册中给出了eval()函数的代码框架，任务4和5中，我们需要做的就是补充eval()函数，在实现eval()函数的过程中我们还会需要其它的函数，例如：括号匹配函数check\_parentheses() 和 寻找dominant operator的函数。  check\_parentheses() 函数设计代码如下：   |  | | --- | | **bool** **check\_parentheses**(**int** l,**int** r){  **if**(tokens[l].type=='('&&tokens[r].type==')'){  **int** index;  **int** lcnt=**0**,rcnt=**0**;  **for**(index=l;index<=r;index++){  **if**(tokens[index].type=='('){lcnt++;}  **if**(tokens[index].type==')'){rcnt++;}  **if**(rcnt > lcnt){**return** false;}  }  **if**(lcnt==rcnt){**return** true;}  }  **return** false;  } |   Dominant\_operator()函数设计代码如下：   |  | | --- | | **int** **dominant\_operator**(**int** l,**int** r){  **int** **operator**=l;  **int** m,n;  **int** min\_precedence=**8**;  **for**(m=l;m<=r;m++){  **if**(tokens[m].type==DIGIT||tokens[m].type==HEX||tokens[m].type==REGISTER){**continue**;}  **int** count=**0**;  **bool** flag = true;  **for**(n=m-**1**;n>=l;n--){  **if**(tokens[n].type=='('&&count==**0**){flag=false;**break**;}  **if**(tokens[n].type=='('){count--;}  **if**(tokens[n].type==')'){count++;}  }  **if**(!flag){**continue**;};  **if**(tokens[m].precedence<min\_precedence){min\_precedence=tokens[m].precedence;**operator**=m;}  }  **return** **operator**;  } |   **必做任务5：实现更复杂的表达式求值**  通过任务4，我们已经做好了表达式运算的基本准备工作。在此任务中我们需要实现完整的表达式求值功能更，因此需要完成 expr.c 中的eval（），expr（）函数，和 ui.c 中的cmd\_p()函数。  现在我们可以继续设计eval()函数，函数代码如下：   |  | | --- | | **uint32\_t** **eval**(**int** l,**int** r){  **if**(l>r){assert(**0**);}  **else** **if**(l==r){  **int** num;  **if**(tokens[l].type==DIGIT){sscanf(tokens[l].str,"%d",&num);}  **if**(tokens[l].type==HEX){sscanf(tokens[l].str,"%x",&num);}  **if**(tokens[l].type==REGISTER){  **if**(strlen(tokens[l].str)==**3**){  **int** i;  **for**(i=R\_EAX;i<=R\_EDI;i++){  **if**(strcmp(tokens[l].str,regsl[i])==**0**){**break**;}  **if**(i>R\_EDI){  **if**(strcmp(tokens[l].str,"eip")==**0**){num=cpu.eip;}  **else** assert(**0**);  }**else** {num=cpu.gpr[i].\_32;}  }  }**else** **if**(strlen(tokens[l].str)==**2**){  **int** i;  **for**(i=R\_AX;i<=R\_DI;i++){  **if**(strcmp(tokens[l].str,regsw[i])==**0**){**break**;}  num=cpu.gpr[i].\_16;  }  }**else**{  **int** i;  **for**(i=R\_AL;i<=R\_BH;i++){  **if**(strcmp(tokens[l].str,regsb[i])==**0**){**break**;}  num=cpu.gpr[i&**0x3**].\_8[i>>**2**];  }  }  }  **return** num;  }**else**{  **if**(check\_parentheses(l,r)){**return** eval(l+**1**,r-**1**);}  **else**{  **int** **operator**=dominant\_operator(l,r);  **if**(l==**operator**||tokens[**operator**].type==POINTER||tokens[**operator**].type==NEG||tokens[**operator**].type=='!'){  **uint32\_t** value=eval(l+**1**,r);  **switch**(tokens[l].type){  **case** '!':**return** !value;  **case** NEG:**return** -value;  **case** POINTER:**return** swaddr\_read(value,**4**);  **default:**assert(**0**);  }  }  **uint32\_t** value1=eval(l,**operator**-**1**);  **uint32\_t** value2=eval(**operator**+**1**,r);  **switch**(tokens[**operator**].type){  **case** '+':**return** value1+value2;  **case** '\*':**return** value1\*value2;  **case** '-':**return** value1-value2;  **case** '/':**return** value1/value2;  **case** AND:**return** value1&&value2;  **case** OR:**return** value1||value2;  **case** EQ:**return** value1==value2;  **case** NEQ:**return** value1!=value2;  **default**:assert(**0**);  **break**;  }  }  }  } |   在ui.c 设计 cmd\_p 函数：   |  | | --- | | **static** **int** **cmd\_p**(**char** \*args){  **bool** success;  **uint32\_t** result;  result=expr(args,&success);  **if**(success){printf("result is 0x%08x**\n**",result);}  **else** assert(**0**);  **return** **0**;  } |   在static struct 添加命令：   |  | | --- | | { "p","calculate the expression",cmd\_p}, |   expr () 函数的实现：是实现表达式求值的函数。该函数中完成了两个选做任务，判断了 “-” 和 “ \* ”的具体意义，然后再调用eval（）函数对表达式进行递归求值。  **选做任务1：实现带有负数的算术表达式的求值**  想要判断-为减号还是负号，只需要判断该符号前面的负号是否为数字或者寄存器，若-的前一个符号为数字或者寄存器说明是减号，若不是则说明是负号。（\*的判断方法也一样）。   |  | | --- | | **if**(tokens[i].type=='\*'&&(i==**0**||(tokens[i-**1**].type!=DIGIT&&tokens[i-**1**].type!=HEX&&tokens[i-**1**].type!=REGISTER&&tokens[i-**1**].type!=')'))){  tokens[i].type=POINTER;  tokens[i].precedence=**6**;  }  **if**(tokens[i].type=='-'&&(i==**0**||(tokens[i-**1**].type!=DIGIT&&tokens[i-**1**].type!=HEX&&tokens[i-**1**].type!=REGISTER&&tokens[i-**1**].type!=')'))){  tokens[i].type=NEG;  tokens[i].precedence=**6**; |   **选做任务2：实现指针解引用**  在expr()函数中已经实现了- 和 \*的具体含义的判断，在eval（）函数只需要加入对应的运算规则即可：   |  | | --- | | **switch**(tokens[**operator**].type){  **case** '+':**return** value1+value2;  **case** '\*':**return** value1\*value2;  **case** '-':**return** value1-value2;  **case** '/':**return** value1/value2;  **case** AND:**return** value1&&value2;  **case** OR:**return** value1||value2;  **case** EQ:**return** value1==value2;  **case** NEQ:**return** value1!=value2;  **default**:assert(**0**);  **break**; |   最后附上实验结果：    **必做任务6：实现监视点池的管理**  首先，我们需要增加监视点结构体的成员。在watchpoint.h文件中有watchpoint结构体的定义。我在结构体中增加了两个成员：   1. char类型的数组 exp[32] , 用来存储算数表达式的内容 2. unit32\_t类型的 value，用来存储算数表达式的结果   接下来需要为了使用监视点池, 我们需要编写以下两个函数WP\* new\_wp() 和 void free\_wp(WP \*wp)。  init\_wp\_pool()函数会对两个链表 free\_ 和 head 进行了初始化：   |  | | --- | | **void** **init\_wp\_pool**() {  **int** i;  **for**(i = **0**; i < NR\_WP; i ++) {  wp\_pool[i].NO = i;  wp\_pool[i].next = &wp\_pool[i + **1**];  }  wp\_pool[NR\_WP - **1**].next = NULL;  head = NULL;  free\_ = wp\_pool;  } |   **new\_wp()函数的实现**  new\_wp()从 free\_链表中返回一个空闲的监视点结构给head链表，且将表达式，表达式的值赋给该监视点结构，具体代码如下：   |  | | --- | | WP\* **new\_wp**(){  WP \*result,\*tmp;  result=free\_;  free\_=free\_->next;  result->next=NULL;  tmp=head;  **if**(tmp==NULL){  head=result;  tmp=head;  }**else**{  **while**(tmp->next!=NULL){tmp=tmp->next;}  tmp->next=result;  }  **return** result;  } |   **free\_wp( )函数的实现**  free\_wp() 函数的参数为WP 类型的指针wp，free\_wp() 的作用是将wp所指的结点归还到free\_链表中。具体步骤如下：   1. 若 wp = NULL ，则说明输入有误 2. 若 wp = head ，说明wp指向head链表的头结点，只需让head指针指向下一个结点，再将wp所指的结点连到free\_链表的第一个位置，并让free\_指针指向该节点 3. 若 wp 为其它结点，则需要对head链表进行遍历找出wp所指的结点，再根据2）中的步骤，将该结点归还到free\_链表中是遍历head链表直到找出对应NO的结点，从head中删除该节点，添加到free\_链表中。   free\_wp() 函数代码设计如下：   |  | | --- | | **void** **free\_wp**(WP \*wp){  WP \*p,\*q;  q=free\_;  **if**(q==NULL){free\_=wp;q=free\_;}  **else**{  **while**(q->next!=NULL){q=q->next;}  q->next=wp;  }  p=head;  **if**(head==NULL){assert(**0**);}  **else**{  **while**(p->next!=NULL&&p->next->NO!=wp->NO){p=p->next;}  **if**(p->next->NO==wp->NO){p->next=p->next->next;}  **else** {assert(**0**);}  }  wp->next=NULL;  wp->value=**0**;  wp->expression[**0**]='\0';  } |   **必做任务7：实现监视点**  **添加监视点**  添加监视点的任务由cmd\_w()函数来完成，该函数中调用new\_wp()函数来存储新的监视点。   |  | | --- | | **static** **int** **cmd\_w**(**char** \*args){  WP \*point;  **bool** success;  point=new\_wp();  point->value=expr(args,&success);  strcpy(point->expression,args);  **if**(!success){printf("expression is wrong**\n**");**return** **0**;}  printf("watchpoint id %d,expression is %s,value is %d**\n**",point->NO,args,point->value);  **return** **0**;  } |   **删除监视点**  删除监视点需要由cmd\_d()函数来实现，该函数中会调用delete\_wp()函数，delete\_wp() 函数的参数为int类型的监视点序号，在该函数中需要遍历head链表，从中找出对应序号的监视点，再调用free\_wp()函数来把此监视点归还到free\_链表中，实现了监视点的删除。   |  | | --- | | **static** **int** **cmd\_d**(**char** \*args){  **int** delete\_num;  sscanf(args,"%d",&delete\_num);  delete\_wp(delete\_num);  printf("delete watchpoint%d successfully**\n**",delete\_num);  **return** **0**;  } |   delete\_wp 函数设计如下（watchpoint.c）：   |  | | --- | | **void** **delete\_wp**(**int** num){  WP \*target;  target=&wp\_pool[num];  free\_wp(target);  } |   **打印监视点**  打印监视点由cmd\_info()函数来实现，该函数中会调用info\_wp()函数，info\_wp()函数所要做的就是对head链表进行遍历  ，然后输出每个监视点的NO, exp，value。  Info\_wp 设计函数如下（watchpoint.c）:   |  | | --- | | **void** **info\_wp**(){  WP \*tmp;  tmp=head;  **if**(!tmp){printf("no watchpoint yet**\n**");}  **else**{  **while**(tmp!=NULL){  printf("watchpoint:%d,expression:%s,value(digit):%d**\n**",tmp->NO,tmp->expression,tmp->value);  tmp=tmp->next;  } |   Cmd\_info\_w 主要在cmd\_info\_r 添加 就行了：   |  | | --- | | **static** **int** **cmd\_info**(**char** \*args){  **if**(args[**0**]=='r'){  **int** i;  **for**(i = **0**; i <= R\_EDI; i++) {  printf( "$%s**\t**0x%08x**\t**%d**\n**", regsl[i], reg\_l(i), reg\_l(i));  }  printf( "$eip**\t**0x%08x**\t**%d**\n**", cpu.eip, cpu.eip );  }  **else** **if**(args[**0**]=='w'){  info\_wp();  }**else** {printf("the argument must be r or w");}  **return** **0**;  } |   最后实验结果 为截图如下： |
| 思考题（请注明题号，如思考题1，思考题2，. . .） |
| **思考题 1：**  opcode\_table 到底是一个什么类型的数组？  答案：在 nemu/src/cpu/exec/exec.c 目录下中看到了opcode\_table数组：   |  | | --- | | helper\_fun opcode\_table [**256**] = {  /\* 0x00 \*/ inv, inv, inv, inv,  /\* 0x04 \*/ inv, inv, inv, inv,  /\* 0x08 \*/ inv, or\_r2rm\_v, or\_rm2r\_b, inv,  /\* 0x0c \*/ or\_i2a\_b, inv, inv, \_2byte\_esc,  /\* 0x10 \*/ inv, adc\_r2rm\_v, inv, inv,  /\* 0x14 \*/ inv, inv, inv, inv,  /\* 0x18 \*/ inv, sbb\_r2rm\_v, inv, inv, |   该数组为 helper\_fun 类型，在同一个文件中看到 helper\_fun 的定义为：   |  | | --- | | **typedef** **int** (\*helper\_fun)(**swaddr\_t**); |   查阅资料后知道了typedef 返回类型(\*新类型)(参数表)的这种使用方式，此处 typedef 的功能是定义新的 helper\_fun 类型，并定义这种类型为指向某种函数的指针，这种函数以一个 swaddr\_t 为参数并返回 int 类型。因此，opcode\_table 数组是一个函数指针数组。  **思考题 2：**  在 cmd\_c()函数中, 调用 cpu\_exec()的时候传入了参数-1 , 你知道为什么吗?  答案：在cpu\_exec.c文件中找到了函数 cpu\_exec()   |  | | --- | | **for**(; n > **0**; n --) {  #ifdef DEBUG  **swaddr\_t** eip\_temp = cpu.eip;  **if**((n & **0xffff**) == **0**) {  /\* Output some dots while executing the program. \*/  fputc('.', stderr);  } |   n是无符号整型，所以-1就是无符号最大的数字，那么函数里的for循环可以执行最大次数的循环，从而让cpu处理之后的指令。  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?  答案：框架代码中定义wp\_pool等变量时使用了关键字static，在此处的含义是静态全局变量，该变量只能被本文件中的函数调用，并且是全局变量，而不能被同一程序其他文件中的函数调用。在此处使用static是为了避免它被误修改。  **思考题 3：**   1. **查阅 i386 手册**   **EFLAGS 寄存器中的 CF 位是什么意思?**  答案：P34页中提到参阅附录c，CF是进位标志   |  | | --- | | 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. |   P419页中写到CF位：在最高位发生进位或者借位的时候将其置1，否则清零。   |  | | --- | | Status Flags' Functions  Bit Name Function  0 CF Carry Flag ── Set on high-order bit carry or borrow; cleared otherwise. |   **ModR/M 字节是什么?**  答案：阅读P241-243页。ModR/M 由 Mod，Reg/Opcode，R/M 三部分组成。  Mod 是前两位，提供寄存器寻址和内存寻址。  Reg/Opcode为3-5位，如果是Reg表示使用哪个寄存器，Opcode表示对group属性的Opcode进行补充。  R/M为6-8位，与mod结合起来查图得8个寄存器和24个内存寻址。    **mov 指令的具体格式是怎么样的?**  答案：在P347页，格式是DEST ← SRC   1. **shell 命令** 2. **Make 文件**   -Wall 和-Werror 有什么作用? 为 什么要使用-Wall 和-Werror？  答案：-Wall 使GCC产生尽可能多的警告信息，取消编译操作，打印出编译时所有错误或警告信息。-Werror 要求GCC将所有的警告当成错误进行处理，取消编译操作。使用 -Wall 和 -Werror就是为了找出存在的错误，尽可能地避免程序运行出错，优化程序。 |
| 实验遇到的问题、思考、解决办法（可以不填写） |
|  |
| 实验心得（可以不填写） |
|  |