
系统综合课程设计-实验报告

PA2-简单复杂的机器：冯诺依曼计算机系统

姓名：孙铭

学号：1711377

学院：计算机学院

专业：计算机科学与技术

时间：2020年4月20日

目录

系统综合课程设计-实验报告

PA2-简单复杂的机器：冯诺依曼计算机系统

目录

PA2-阶段1

1. 代码：运行第一个C程序

 1.1 试运行dummy

 1.2 push指令实现

 1.3 POP指令的实现

 1.4 CALL指令的实现

 1.5 SUB指令的实现

 1.6 XOR的实现

 1.7 RET的实现

 1.8 运行结果

PA2-阶段2

2. 代码：运行更多程序

 2.1 AND的实现

 2.2 PUSH、PUSHL的实现

 2.3 XCHG(NOP)的实现

 2.4 SETcc的实现

 2.5 MOVZX, MOVSX的实现

 2.6 JCC的实现

 2.7 SAR、SAL、SHL、SHR的实现

 2.8 TEST的实现

 2.9 ADD的实现

 2.10 CMP的实现

 2.11 JMP的实现

 2.12 MUL的实现

 2.13 IMUL的实现

 2.14 DIV的实现

 2.15 IDIV的实现

 2.16 ADC的实现

 2.17 SBB的实现

 2.18 NEG的实现

 2.19 OR的实现

 2.20 NOT的实现

 2.21 DEC的实现

 2.22 INC的实现

 2.23 CLTD的实现

 2.24 LEAVE的实现

 2.25 CALL补充

 2.26 运行与结果

3. 基础设施

 3.1 代码：实现differential testing

PA2-阶段3

4 代码：加入IOE

 4.1 串口

 4.2 时钟

 4.3 键盘

 4.4 VGA

5. 一个非常严重的问题

PA2-必答题

必答题1

- (1) 去掉 `static`
- (2) 去掉 `inline`
- (3) 去掉 `static inline`

必答题2

来个小小的总结吧

下面进入本次实验内容的分析。

PA2-阶段1

1. 代码：运行第一个C程序

问题描述：

你在PA2的第一个任务，就是实现若干条指令，是的第一个简单的C程序可以在NEMU中运行起来。这个简单的C程序的代码是 `nexus-am/tests/cputest/tests/dummy.c`，它什么都不做就直接返回，在 `nexus-am/tests/cputest` 下键入 `make ARCH=x86-nemu ALL=dummy run`，编译dummy程序，并启动NEMU运行它。

1.1 试运行dummy

那么首先，根据题目描述，我们在 `nexus-am/tests/cputest` 下输入 `make ARCH=x86-nemu ALL=dummy run`，出现错误：

```
sun@ests:~/Desktop/ics2018/nexus-am/tests/cputest$ make ARCH=x86-nemu ALL=dummy run
Makefile:1: /Makefile.check: 没有那个文件或目录
make: *** No rule to make target '/Makefile.check'。停止。
```

报错有些熟悉，问题可能出在环境变量的设置上。查阅 `ics2018/nexus-am/README.MD`，看到了如下信息。

```
28 一些注意事项：
29
30 * `NAME` 定义了应用的名字。编译后会在 `build/` 目录里出现以此命名的应用程序。
31 * `SRCS` 指定了编译应用所需的源文件。可以放在应用目录中的任意位置。
32
33
34 * 应用目录下的 `include/` 目录会被添加到编译的 `-I` 选项中。
35 * 环境变量 `AM_HOME` 需要包含 **nexus-am项目的根目录的绝对路径**。
36
37 编译时，首先确保 `AM_HOME` 正确设置，然后执行 `make ARCH=体系结构名` 编译。例如 `make ARCH=native` 将会编译成本地可运行的项目，`make ARCH=mips32-minimal` 生成用于仿真的MIPS32程序>。
`ARCH` 缺省时默认编译到本地。
```

注意到一点是“环境变量 `AM_HOME` 需要包含 **nexus-am** 项目的根目录的绝对路径”，因此需要在系统中添加环境变量。为此，首先需要将目录切至家目录，之后执行命令 `vim .bashrc` 来添加环境变量，在该文件末尾处加入如下代码。

```
119 export NEMU_HOME=/home/sun/Desktop/ics2018/nemu
120 export AM_HOME=/home/sun/Desktop/ics2018/nexus-am
121 export NAVY_HOME=/home/sun/Desktop/ics2018/navy-apps
```

关闭vim编辑器，切换到之前的目录，执行上述指令，发现依然报相同的错误。可能是由于对换进变量的修改没有生效，于是我杀死进程关闭终端，重新执行上述指令，这一次得到了运行结果如下。

```
[src/monitor/monitor.c,65,load_img] The image is /home/sun/Desktop/ics2018/nexus-am/tests/cputest/build/dummy-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 16:39:27, Apr 22 2020
For help, type "help"
(nemu) c
invalid opcode(eip = 0x0010000a): e8 01 00 00 00 90 55 89 ...
There are two cases which will trigger this unexpected exception:
1. The instruction at eip = 0x0010000a is not implemented.
2. Something is implemented incorrectly.
Find this eip(0x0010000a) in the disassembling result to distinguish which case it is.

If it is the first case, see
[REDACTED]
for more details.

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

(nemu) █
```

查看反汇编结果 `nexus-am/tests/cputest/build/dummy-x86-nemu.txt` 可知，如下图所示，根据题目报错要求，还有call、push、sub、xor、pop、ret六条指令没有实现。

```
5 Disassembly of section .text:
6
7 00100000 <_start>:
8 100000: bd 00 00 00 00      mov    $0x0,%ebp
9 100005: bc 00 7c 00 00      mov    $0x7c00,%esp
10 10000a: e8 01 00 00 00     call   100010 <_trm_init>
11 10000f: 90                 nop
12
13 00100010 <_trm_init>:
14 100010: 55                 push   %ebp
15 100011: 89 e5              mov    %esp,%ebp
16 100013: 83 ec 08          sub    $0x8,%esp
17 100016: e8 05 00 00 00     call   100020 <main>
18 10001b: d6                 (bad)
19 10001c: eb fe              jmp   10001c <_trm_init+0xc>
20 10001e: 66 90              xchg   %ax,%ax
21
22 00100020 <main>:
23 100020: 55                 push   %ebp
24 100021: 89 e5              mov    %esp,%ebp
25 100023: 31 c0              xor    %eax,%eax
26 100025: 5d                 pop    %ebp
27 100026: c3                 ret
```

1.2 push指令实现

接下来，分别实现各指令。由于实现call指令时需要进行压栈操作，因此这里首先对push和pop指令进行实现。

根据实验指导，现阶段只需实现 `PUSH r32` 即可。

查询i386手册。手册中对push指令的描述为：如果指令的操作数大小属性为16位，push将堆栈指针递减2；否则，将堆栈指针递减4。之后，push将操作数放置在堆栈新的顶部，堆栈指针指向该顶部。而80386 PUSH的eSP指令会将eSP的值压至指令之前的状态。不同于8086，PUSH SP会将新值压入（递减2）。

PUSH — Push Operand onto the Stack

Opcode		Instruction	Clocks	Description
FF	/6	PUSH m16	5	Push memory word
FF	/6	PUSH m32	5	Push memory dword
50	+ /r	PUSH r16	2	Push register word
50	+ /r	PUSH r32	2	Push register dword
6A		PUSH imm8	2	Push immediate byte
68		PUSH imm16	2	Push immediate word
68		PUSH imm32	2	Push immediate dword
0E		PUSH CS	2	Push CS
16		PUSH SS	2	Push SS
1E		PUSH DS	2	Push DS
06		PUSH ES	2	Push ES
0F	A0	PUSH FS	2	Push FS
0F	A8	PUSH GS	2	Push GS

由 `PUSH r32` 对应的 opcode 内容 `50 +/r` 可以看出，push 指令的 opcode 为 `0x50-0x57`。

因此，首先，我们需要在 `nemu/src/cpu/exec/all-instr.h` 中进行函数声明，由于之后实现的命令都需要声明，因此这里给出全部指令声明的代码如下。

```
10 make_EHelper(call);
11 make_EHelper(push);
12 make_EHelper(pop);
13 make_EHelper(sub);
14 make_EHelper(xor);
15 make_EHelper(ret);
```

之后，根据指令功能可以知道，`make_EHelper(push)` 函数的实现在于 `data-mov.c` 文件中，而实现 `make_EHelper(push)` 函数之前，需要先实现 `rtl_push` 函数：即修改栈顶，并将指针 `src1` 中的内容写入栈。在 `nemu/include/cpu/rtl.h` 中，添加如下代码：

```
142 static inline void rtl_push(const rtlreg_t* src1) {
143     // esp <- esp - 4
144     // M[esp] <- src1
145     // TODO();
146     // cpu.esp中存放栈顶地址
147     rtl_subi(&cpu.esp, &cpu.esp, 4);    //减法：寄存器与立即数
148     rtl_sm(&cpu.esp, 4, src1); //写入内存：32bit
149 }
```

在实现了 `rtl_push` 函数的基础上，下一步就需要实现 `make_EHelper(push)` 函数了，该函数主要功能即调用 `rtl_push` 函数写栈，这样在调用译码函数 `make_DHelper(r)` 时，会读取 `decoding.opcode` 中标志的寄存器，将寄存器内容放入 `op->val` 中。在 `nemu/src/cpu/exec/data-mov.c` 中添加如下代码：

```
8 make_EHelper(push) {
9     // TODO();
10    rtl_push(&id_dest->val);
11
12    print_asm_template1(push);
13 }
```

最后需要填写`opcode_table`，根据前文`push r32`对应的opcode为0x50-0x57，译码函数为`make_DHelper(r)`，执行函数为`make_EHelper(push)`，定义为`IDEX(r, push)`。在`nemu/src/cpu/exec/exec.c`中添加如下代码。这里需要注意，`push r32`添加在1 byte_opcode_table下。

```
95  /* 0x50 */    IDEX(r, push), IDEX(r, push), IDEX(r, push), IDEX(r, push),
96  /* 0x54 */    IDEX(r, push), IDEX(r, push), IDEX(r, push), IDEX(r, push),
```

1.3 POP指令的实现

POP将当前堆栈顶部（由ESP指示）中的单字或双字传送到目标操作数，然后递增ESP以指向堆栈的新顶部。首先查看i386手册中关于POP指令的讲解。

POP — Pop a Word from the Stack

Opcode	Instruction	Clocks	Description
8F /0	POP m16	5	Pop top of stack into memory word
8F /0	POP m32	5	Pop top of stack into memory dword
58 + rw	POP r16	4	Pop top of stack into word register
58 + rd	POP r32	4	Pop top of stack into dword register
1F	POP DS	7, pm=21	Pop top of stack into DS
07	POP ES	7, pm=21	Pop top of stack into ES
17	POP SS	7, pm=21	Pop top of stack into SS
0F A1	POP FS	7, pm=21	Pop top of stack into FS
0F A9	POP GS	7, pm=21	Pop top of stack into GS

根据实验指导，现阶段需要实现的POP指令为`POP r32`，观察上图，`POP r32`对应的opcode为58 + rd。rd表示32位通用寄存器编号，+rd将32位通用寄存器编号按数值加到opcode中。这里注意，POP通用寄存器时，不包含ESP（用于栈顶指针）。故opcode为0x58-0x5F（不包含0x5C）。

与PUSH类似，POP指令使用`make_DHelper(r)`作为译码函数，执行函数为`make_EHelper(pop)`，在此之前须先实现`rtl_pop()`函数，该函数作用是从栈中读出数据并放在`dest`中。

在`nemu/include/cpu/rtl.h`中，添加如下代码：

```
151 static inline void rtl_pop(rtlreg_t* dest) {
152     // dest <- M[esp]
153     // esp <- esp + 4
154     // TODO();
155     rtl_lm(dest, &cpu.esp, 4); //读内存
156     rtl_addi(&cpu.esp, &cpu.esp, 4);
157 }
```

下面需要实现`make_EHelper(pop)`，即将`rtl_pop`读取的数据写入到通用寄存器中。这里使用`operand_write()`函数（位于`decode.c`）执行读写寄存器操作。在`nemu/src/cpu/exec/data-mov.c`中添加如下代码：

```
15 make_EHelper(pop) {
16     // TODO();
17     rtl_pop(&t2); //使用临时寄存器保存值
18     operand_write(id_dest, &t2); //使用operand_write执行写操作
19
20     print_asm_template1(pop);
21 }
```

最后，填写opcode_table，由前文有，指令opcode为0x58-0x5F（不包含0x5C），使用make_DHelper(r)作为译码函数，执行函数为make_EHelper(pop)，故可以定义为IDEX(r, pop)。同样在1byte_opcode_table下，添加如下代码：

```
97  /* 0x58 */    IDEX(r, pop), IDEX(r, pop), IDEX(r, pop), IDEX(r, pop),
98  /* 0x5c */    EMPTY, IDEX(r, pop), IDEX(r, pop), IDEX(r, pop),
```

1.4 CALL指令的实现

根据实验指导中的要求，现阶段需要实现的call指令为CALL rel32形式。
%eip的跳转可以通过将decoding.is_jmp设为1，并将decoding.jmp_eip设为跳转目标地址来实现，这时在update_eip()函数中会把跳转目标地址作为新的%eip，而不是顺序意义下的下一条指令的地址。

查阅i386手册关于CALL指令的定义如下。

CALL — Call Procedure

Opcode	Instruction	Clocks	Description
E8 cw	CALL rel16	7+m	Call near, displacement relative to next instruction
FF /2	CALL r/m16	7+m/10+m	Call near, register indirect/memory indirect
9A cd	CALL ptr16:16	17+m, pm=34+m	Call intersegment, to full pointer given
9A cd	CALL ptr16:16	pm=52+m	Call gate, same privilege
9A cd	CALL ptr16:16	pm=86+m	Call gate, more privilege, no parameters
9A cd	CALL ptr16:16	pm=94+4x+m	Call gate, more privilege, x parameters
9A cd	CALL ptr16:16	ts	Call to task
FF /3	CALL m16:16	22+m, pm=38+m	Call intersegment, address at r/m dword
FF /3	CALL m16:16	pm=56+m	Call gate, same privilege
FF /3	CALL m16:16	pm=90+m	Call gate, more privilege, no parameters
FF /3	CALL m16:16	pm=98+4x+m	Call gate, more privilege, x parameters
FF /3	CALL m16:16	5 + ts	Call to task
E8 cd	CALL rel32	7+m	Call near, displacement relative to next instruction
FF /2	CALL r/m32	7+m/10+m	Call near, indirect
9A cp	CALL ptr16:32	17+m, pm=34+m	Call intersegment, to pointer given
9A cp	CALL ptr16:32	pm=52+m	Call gate, same privilege
9A cp	CALL ptr16:32	pm=86+m	Call gate, more privilege, no parameters
9A cp	CALL ptr32:32	pm=94+4x+m	Call gate, more privilege, x parameters
9A cp	CALL ptr16:32	ts	Call to task
FF /3	CALL m16:32	22+m, pm=38+m	Call intersegment, address at r/m dword
FF /3	CALL m16:32	pm=56+m	Call gate, same privilege
FF /3	CALL m16:32	pm=90+m	Call gate, more privilege, no parameters
FF /3	CALL m16:32	pm=98+4x+m	Call gate, more privilege, x parameters
FF /3	CALL m16:32	5 + ts	Call to task

CALL指令不能实现短转移，在指令执行时，先将当前的IP或CS与IP压入栈中，再转移eip。CALL rel32的opcode为0xE8，E8 cd表示字节0xE8后面跟着一个4字节的操作数，用于表示要跳转的地址与当前地址的偏移量。

该指令的译码函数为make_DHelper(J)，涉及到的操作数只有一个，即立即数。CPU的跳转目标地址为当前eip加上立即数offset（正负均可）。在decode/decode.c中可以找到对应的译码函数make_DHelper(J)，该函数通过调用decode_op_SI函数来实现立即数的读取，并更新decoding.jmp_eip。make_DHelper(J)函数代码如下。

```
263 make_DHelper(J) {
264     decode_op_SI(eip, id_dest, false);
265     // the target address can be computed in the decode stage
266     // 目标函数可在译码阶段进行计算
267     decoding.jmp_eip = id_dest->simm + *eip;
268 }
```

其中，decode_op_SI函数即为make_DopHelper(SI)，同make_DopHelper(I)相比，make_DopHelper(SI)的操作数宽度width=1或width=4；此外，后者读取的立即数需转为signed immediate。实现代码如下：

```

30 static inline make_DopHelper(SI) {
31     assert(op->width == 1 || op->width == 4);
32
33     op->type = OP_TYPE_IMM;
34
35     /* TODO: Use instr_fetch() to read `op->width` bytes of memory
36      * pointed by `eip`. Interpret the result as a signed immediate,
37      * and assign it to op->simm.
38      *
39      op->simm = ??? */
40
41     // TODO();
42     op->simm = instr_fetch(eip, op->width); //获取立即数
43     if(op->width == 1)
44         op->simm = (int8_t)op->simm;
45
46     rtl_li(&op->val, op->simm);
47
48 #ifdef DEBUG
49     sprintf(op->str, OP_STR_SIZE, "$0x%0x", op->simm);
50 #endif
51 }

```

接下来是对执行函数`make_EHelper(call)`函数的实现，该函数位于
`nemu/src/cpu/exec/control.c`文件中，call指令需先将eip压栈，再实现跳
转，因此执行函数实现代码如下。

```

26 make_EHelper(call) {
27     // the target address is calculated at the decode stage
28     // TODO();
29     // eip入栈：因为实现入栈的是push寄存器，故这里使用临时寄存器t2
30     rtl_li(&t2, decoding.seq_eip);
31     rtl_push(&t2);
32     // 设置is_jmp为1
33     decoding.is_jmp = 1;
34
35     print_asm("call %x", decoding.jmp_eip);
36 }

```

最后是对`opcode_table`的填写，前文已经指明，opcode为0xE8，译码函数
`make_DHelper(J)`，执行函数为`make_EHelper(call)`，操作数宽度2/4，定
义为`IDEX(J, call)`。同样在1 byte_opcode_table下，添加如下代码：

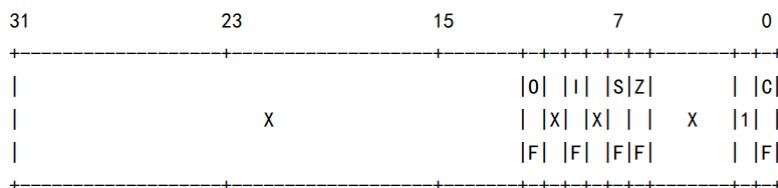
```

133    /* 0xe8 */    IDEX(J,call), EMPTY, EMPTY, EMPTY,

```

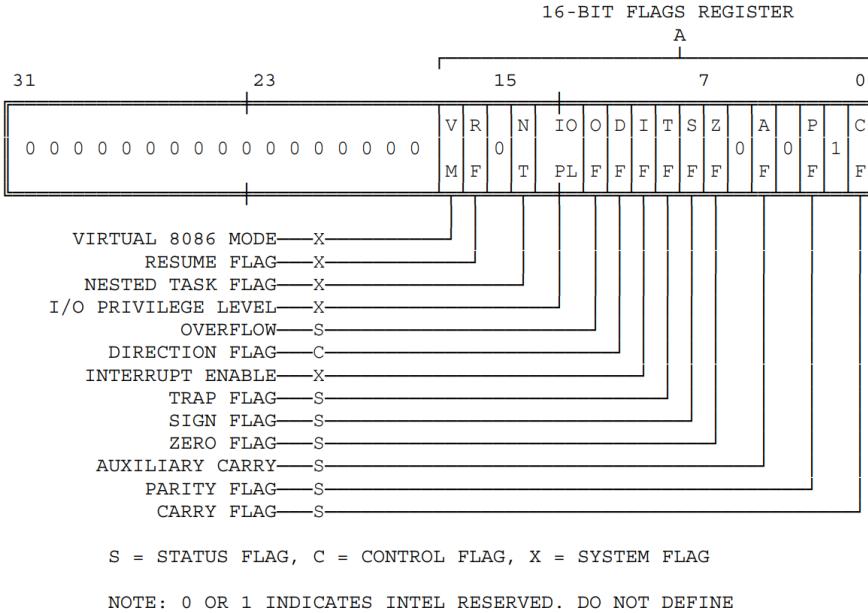
1.5 SUB指令的实现

在实现SUB指令之前，需要先实现EFLAGS寄存器，需要做的是在寄存器结
构体中添加EFLAGS寄存器即可，EFLAGS是个32位寄存器，结构如下。



根据实验指导，我们只会用到以下五个位：即CF，ZF，SF，IF，OF，标记
成X的位不用关心，其功能可以暂不实现。关于EFLAGS寄存器中每一位的含
义，查阅i386手册如下。

Figure 2-8. EFLAGS Register



各标志位详细功能如下。

CF(bit 0)[Carry Flag]

若算术操作产生的结果在最高有效位(*most-significant bit*)发生进位或错位则将其置为1，反之清零。这个标志指示无符号整形运算的溢出状态，这个标志同样在多倍精度运算中使用。

ZF(bit 6)[Zero Flag]

若结果为0则将其置1，反之清零。

SF(bit 7)[Sign Flag]

该标志被设置为有符号整型的最高有效位(0指示结果为正，反之则为负)。

IF(bit 9)[Interrupt Enable Flag]

该标志用于控制处理器对可屏蔽中断请求的响应。置1以响应可屏蔽中断，反之则禁止可屏蔽中断。

OF(bit 11)[Overflow Flag]

如果整型结果是较大的正数或较小的负数，并且无法匹配目的操作数时将该位置1，反之清零。这个标志为带符号整型运算指示溢出状态。

因此，首先对EFLAGS寄存器进行定义，代码在 `nemu/include/cpu/reg.h` 下。

```
34 // 实现eflags寄存器
35 struct bs{
36     unsigned int CF:1;
37     unsigned int one:1; //i386中为1
38     unsigned int :4;
39     unsigned int ZF:1;
40     unsigned int SF:1; //bit 0-7
41
42     unsigned int :1;
43     unsigned int IF:1;
44     unsigned int OF:1;
45     unsigned int :20; //bit 8-31
46 }eflags;
```

接下来，对EFLAGS进行初始化，初始化部分的代码在 `nemu/src/monitor/monitor.c` 下。同时修改 `restart()` 函数，使用 `memcpy()` 将 EFLAGS 设置为 `0x0000 0002H`（i386手册第十章）。

```
82 static inline void restart() {
83     /* Set the initial instruction pointer. */
84     cpu.eip = ENTRY_START;
85     // 进行eflags的初始化
86     unsigned int origin = 2;
87     memcpy(&cpu.eflags,&origin, sizeof(cpu.eflags));
88
89 #ifdef DIFF_TEST
90     init_qemu_reg();
91 #endif
92 }
```

下一步需要实现相关的RTL指令，代码在 `nemu/include/cpu/rtl.h` 目录下。

关于EFLAGS寄存器的标志位读写函数如下。

```
114 #define make_rtl_setget_eflags(f) \
115     static inline void concat(rtl_set_, f) (const rtlreg_t* src) { \
116         TODO(); \
117         cpu.eflags.f = *src; \
118     } \
119     static inline void concat(rtl_get_, f) (rtlreg_t* dest) { \
120         TODO(); \
121         *dest = cpu.eflags.f; \
122     }
```

关于EFLAGS寄存器的标志位更新函数。

```

161 static inline void rtl_eq0(rtlreg_t* dest, const rtlreg_t* src1) {
162     // dest <- (src1 == 0 ? 1 : 0)
163     TODO();
164     rtl_sltui(dest, src1, 1);
165     // 使用RTL的比较指令（寄存器-立即数）进行实现：src<1, dest=1;else dest=0
166     // 注意区别unsigned int 与 int
167     // 可以使用rtl_eq0(dest, dest)进行dest为1/0的翻转
168 }
169
170 static inline void rtl_eqi(rtlreg_t* dest, const rtlreg_t* src1, int imm) {
171     // dest <- (src1 == imm ? 1 : 0)
172     TODO();
173     // 使用RTL的异或指令进行判断（寄存器-立即数）
174     rtl_xori(dest, src1, imm); //src1==imm时dest=0,否则dest=1
175     rtl_eq0(dest, dest); //使用eq0进行校准。src1==imm,dest=1;else dest=0
176 }
177
178 static inline void rtl_neq0(rtlreg_t* dest, const rtlreg_t* src1) {
179     // dest <- (src1 != 0 ? 1 : 0)
180     TODO();
181     rtl_eq0(dest, src1); //src==0时dest=1, src==1时dest=0
182     rtl_eq0(dest, dest); //使用eq0进行校准, dest==0时dest=1,dest==1时dest=0
183     // 也可使用rtl_xori(dest, dest, 0x1);与1异或得（非x）
184 }

```

```

186 static inline void rtl_msb(rtlreg_t* dest, const rtlreg_t* src1, int width)
187     // dest <- src1[width * 8 - 1]
188     TODO();
189     rtl_shri(dest, src1, width * 8 - 1); // 右移
190     rtl_andi(dest, dest, 0x1); // 保留最后一位，即src1[width * 8 - 1]
191 }
192
193 static inline void rtl_update_ZF(const rtlreg_t* result, int width) {
194     // eflags.ZF <- is_zero(result[width * 8 - 1 .. 0])
195     TODO();
196     assert(result != &t0);
197     // 取result的后width个字节，可类比vaddr_read的实现
198     rtl_andi(&t0, result, (0xfffffffffu >> (4 - width) * 8));
199     rtl_eq0(&t0, &t0); // 判断是否为0
200     rtl_set_ZF(&t0);
201 }
202
203 static inline void rtl_update_SF(const rtlreg_t* result, int width) {
204     // eflags.SF <- is_sign(result[width * 8 - 1 .. 0])
205     TODO();
206     assert(result != &t0);
207     rtl_msb(t0, result, width); // 使用result[width*8-1]为0/1判断该数为正/负
208     rtl_set_SF(&t0);
209 }

```

在实现了EFLAG寄存器及其基本操作之后，接下来就可以实现SUB指令了，查阅i386手册中对SUB指令的定义如下。

SUB — Integer Subtraction

Opcode	Instruction	Clocks	Description
2C ib	SUB AL,imm8	2	Subtract immediate byte from AL
2D iw	SUB AX,imm16	2	Subtract immediate word from AX
2D id	SUB EAX,imm32	2	Subtract immediate dword from EAX
80 /5 ib	SUB r/m8,imm8	2/7	Subtract immediate byte from r/m byte
81 /5 iw	SUB r/m16,imm16	2/7	Subtract immediate word from r/m word
81 /5 id	SUB r/m32,imm32	2/7	Subtract immediate dword from r/m dword
83 /5 ib	SUB r/m16,imm8	2/7	Subtract sign-extended immediate byte from r/m word
83 /5 ib	SUB r/m32,imm8	2/7	Subtract sign-extended immediate byte from r/m dword
28 /r	SUB r/m8,r8	2/6	Subtract byte register from r/m byte
29 /r	SUB r/m16,r16	2/6	Subtract word register from r/m word
29 /r	SUB r/m32,r32	2/6	Subtract dword register from r/m dword
2A /r	SUB r8,r/m8	2/7	Subtract byte register from r/m byte
2B /r	SUB r16,r/m16	2/7	Subtract word register from r/m word
2B /r	SUB r32,r/m32	2/7	Subtract dword register from r/m dword

对于表中opcode字段，“ib”表示8位立即数，“iw”表示16位立即数，“id”表示32位立即数。“/r”表示后面跟一个ModR/M字节，并且其中的reg/opcode字段被解释为寄存器的编码。“/digit”中，digit为0~7中的一个数字（/0），表示操作码后跟一个ModR/M字节，并且reg/opcode字段被解释为扩展opcode，取值为digit。

接下来需要实现0x2D，译码函数是 `make_DHelper(I2a)`，该函数调用 `decode_op_e` 读取AX/EAX中的数据写入id_dest，调用 `decode_op_I` 读取立即数并存入id_src。首先需要实现执行函数 `eFlags_modify()`，计算减法并相应地设置eFlags寄存器的值。在 `nemu/src/cpu/exec` 中，代码实现如下。

```
3 static inline void eFlags_modify(){
4     rtl_sub(&t2, &id_dest->val, &id_src->val);
5     // ZF SF
6     rtl_update_ZFSF(&t2, id_dest->width);
7     // CF
8     // CF=1的判断：作为无符号数的被减数小于同样作为无符号数的减数
9     rtl_sltu(&t0, &id_dest->val, &id_src->val);
10    rtl_set_CF(&t0);
11    // OF
12    // OF的判断：正-负=负 或 负-正=正 时发生溢出。（使用最高位来判断正负）
13    // 即被减数同时与 减数/差 异号
14    rtl_xor(&t0, &id_dest->val, &id_src->val);
15    rtl_xor(&t1, &id_dest->val, &t2);
16    rtl_and(&t0, &t0, &t1);
17    rtl_msb(&t0, &t0, id_dest->width); // 取表示正负的最高位
18    rtl_set_OF(&t0);
19 }
```

接下来需要实现 `make_EHelper(sub)`，调用 `eFlags_modify()` 计算减法之后，将值写回寄存器。还是在 `arith.c` 文件下。

```
27 make_EHelper(sub) {
28     // TODO();
29     eFlags_modify();
30     operand_write(id_dest, &t2);
31     print_asm_template2(sub);
32 }
```

下面需要填写 `opcode_table`，前文已经知道，SUB指令 `opcode` 为0x2D，译码函数 `make_DHelper(I2a)`，执行函数 `make_EHelper(sub)`，操作数宽度为2或4，定义为 `IDEX(I2a, sub)`。在 `exec.c` 文件中写入如下代码。

```
86 /* 0x2c */ EMPTY, IDEX(I2a,sub), EMPTY, EMPTY,
```

`sub` 指令具有很多不同形式，由于其执行阶段相同，译码函数不同，在实现执行函数后，只需再根据不同形式设定译码函数和操作数宽度。对于 `0x29/r`，译码函数 G2E，定义为 `IDEX(G2E, sub)`；对 `0x2B/r`，译码函数 E2G，定义为 `IDEX(E2G, sub)`。在 `exec.c` 文件下修改如下代码。

```
85 /* 0x28 */ EMPTY, IDEX(G2E,sub), EMPTY, IDEX(E2G,sub),
86 /* 0x2c */ EMPTY, IDEX(I2a,sub), EMPTY, EMPTY,
```

对于 `sub` 指令中 `0x80, 0x81, 0x83` 的 `sub` 指令需要进行 `opcode` 扩展。扩展函数位于 `make_group()` 函数下，由于 `sub` 的 `ext_opcode=5`，因此在 `opcode_table_gp1[5]` 处填写 `EX(sub)` 执行函数如下。

```
42 /* 0x80, 0x81, 0x83 */
43 make_group(gp1,
44     EMPTY, EMPTY, EMPTY, EMPTY,
45     EMPTY, EX(sub), EMPTY, EMPTY)
```

1.6 XOR的实现

首先查看 i386 手册中关于 XOR 指令的介绍。

XOR — Logical Exclusive OR

Opcode	Instruction	Clocks	Description
34 ib	XOR AL,imm8	2	Exclusive-OR immediate byte to AL
35 iw	XOR AX,imm16	2	Exclusive-OR immediate word to AX
35 id	XOR EAX,imm32	2	Exclusive-OR immediate dword to EAX
80 /6 ib	XOR r/m8,imm8	2/7	Exclusive-OR immediate byte to r/m byte
81 /6 iw	XOR r/m16,imm16	2/7	Exclusive-OR immediate word to r/m word
81 /6 id	XOR r/m32,imm32	2/7	Exclusive-OR immediate dword to r/m dword
83 /6 ib	XOR r/m16,imm8	2/7	XOR sign-extended immediate byte with r/m word
83 /6 id	XOR r/m32,imm8	2/7	XOR sign-extended immediate byte with r/m dword
30 /r	XOR r/m8,r8	2/6	Exclusive-OR byte register to r/m byte
31 /r	XOR r/m16,r16	2/6	Exclusive-OR word register to r/m word
31 /r	XOR r/m32,r32	2/6	Exclusive-OR dword register to r/m dword
32 /r	XOR r8,r/m8	2/7	Exclusive-OR byte register to r/m byte
33 /r	XOR r16,r/m16	2/7	Exclusive-OR word register to r/m word
33 /r	XOR r32,r/m32	2/7	Exclusive-OR dword register to r/m dword

因此对于执行函数 `make_EHelper(xor)`，在 `logic.c` 中实现代码如下。

```
15 make_EHelper(xor) {
16     // TODO();
17     rtl_xor(&t2, &id_dest->val, &id_src->val);
18     operand_write(id_dest, &t2); // 写回寄存器
19     // 修改eflags
20     // SF ZF
21     rtl_update_ZFSF(&t2, id_test->width);
22     // CF OF <-0
23     rtl_set_CF(&tzero);
24     rtl_set_OF(&tzero);
25
26     print_asm_template2(xor);
27 }
```

接下来填充gp1，和sub相似，xor也具有扩展opcode形式即 `0x80`、`0x81`、`0x83`，因此填写gp1表项，ext_opcode=6，则在gp1[6]处填写EX(xor)。在 `exec.c` 中添加如下代码。

```
42 /* 0x80, 0x81, 0x83 */
43 make_group(gp1,
44     EMPTY, EMPTY, EMPTY, EMPTY,
45     EMPTY, EX(sub), EX(xor), EMPTY)
```

```
87 /* 0x30 */    INDEXW(G2E,xor,1), INDEX(G2E,xor), INDEXW(E2G,xor,1), INDEX(E2G,xor),
88 /* 0x34 */    EMPTY, INDEX(I2a,xor), EMPTY, EMPTY,
```

1.7 RET的实现

首先查看i386手册中关于RET指令的说明。

RET — Return from Procedure

Opcode	Instruction	Clocks	Description
C3	RET	10+m	Return (near) to caller
CB	RET	18+m,pm=32+m	Return (far) to caller, same privilege
CB	RET	pm=68	Return (far), lesser privilege, switch stacks
C2 iw	RET imm16	10+m	Return (near), pop imm16 bytes of parameters
CA iw	RET imm16	18+m,pm=32+m	Return (far), same privilege, pop imm16 bytes
CA iw	RET imm16	pm=68	Return (far), lesser privilege, pop imm16 bytes

这里实现 `0xC3` 的RET指令，原理即用栈的数据修改IP内容，从而实现近转移。该指令无译码函数，执行函数为 `make_EHelper(ret)`。在 `control.c` 中添加如下代码。

```
38 make_EHelper(ret) {
39     // TODO();
40     rtl_pop(&t2);
41     decoding.jmp_eip = t2; //用栈的数据修改EIP
42     decoding.is_jmp = 1; // 设置is_jmp标志
43
44     print_asm("ret");
45 }
```

接下来填写opcode_table，在0xc3处写入EX(ret)如下。

```
123 /* 0xc0 */    IDEWX(gp2_Ib2E, gp2, 1), IDEX(gp2_Ib2E, gp2), EMPTY, EX(ret),
```

1.8 运行结果

执行指令make run，发现报错，经过查找，发现eflags寄存器定义有问题，于是修改如下。

```
// 实现eflags寄存器
struct bs{
    unsigned int CF:1;
    unsigned int one:1; //i386中为1
    unsigned int :4;
    unsigned int ZF:1;
    unsigned int SF:1;      //bit 0-7

    unsigned int :1;
    unsigned int IF:1;
    unsigned int :1;
    unsigned int OF:1;
    unsigned int :20;      //bit 8-31
}eflags;
```

继续编译nemu，发现了一些其他细小的bug，这里就不再继续说明了。最后调试成功时结果如下。

```
[src/monitor/monitor.c,47,load_default_img] No image is given. Use the default b
uild-in image.
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 22:29:01, Apr 30 2020
For help, type "help"
(nemu) c
nemu: HIT GOOD TRAP at eip = 0x00100026
```

以上是本次实验阶段一。

PA2-阶段2

2. 代码：运行更多程序

在nexus-am/tests/cputest/目录下，执行指令`make ARCH=x86-nemu ALL=add run`命令，并输入指令`c`，结果如下。

```
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 22:29:01, Apr 30 2020
For help, type "help"
(nemu) c
invalid opcode(eip = 0x00100054): 8d 4c 24 04 83 e4 f0 ff ...

There are two cases which will trigger this unexpected exception:
1. The instruction at eip = 0x00100054 is not implemented.
2. Something is implemented incorrectly.
Find this eip(0x00100054) 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!
```

在文件 `nexus-am/tests/cputest/build/add-x86-nemu.txt` 文件下查看反汇编源码，其中 `invalid opcode(eip=0x00100054)` 处对应的指令为 `lea`，于是继续实现 `lea` 指令，直至运行 `add.c` 时不再报错为止。

```
00100054 <main>:
100054:   8d 4c 24 04      lea    0x4(%esp),%ecx
100058:   83 e4 f0      and    $0xffffffff,%esp
10005b:   ff 71 fc      pushl -0x4(%ecx)
10005e:   55             push   %ebp
10005f:   89 e5          mov    %esp,%ebp
100061:   57             push   %edi
100062:   56             push   %esi
100063:   53             push   %ebx
100064:   51             push   %ecx
100065:   83 ec 08      sub    $0x8,%esp
100068:   31 ff          xor    %edi,%edi
10006a:   66 90          xchg   %ax,%ax
```

首先在 `exec/all-instr.h` 中声明各指令执行函数 `make_EHelper()` 如下。

```
17 make_EHelper(leave);
18 make_EHelper(cltd);
19 make_EHelper(movsx);
20 make_EHelper(movzx);
21
22 make_EHelper(add);
23 make_EHelper(inc);
24 make_EHelper(dec);
25 make_EHelper(cmp);
26 make_EHelper(neg);
27 make_EHelperadc());
28 make_EHelper(sbb());
29 make_EHelper(mul);
30 make_EHelper(imul1);
31 make_EHelper(imul2);
32 make_EHelper(imul3);
33 make_EHelper(div);
34 make_EHelper(idiv);
```

```

36 make_EHelper(not);
37 make_EHelper(and);
38 make_EHelper(or);
39 make_EHelper(xor);
40 make_EHelper(sal);
41 make_EHelper(shl);
42 make_EHelper(shr);
43 make_EHelper(sar);
44 make_EHelper(setcc);
45 make_EHelper(test);
46
47 make_EHelper(jmp);
48 make_EHelper(jmp_rm);
49 make_EHelper(jcc);
50 make_EHelper(lea);
51 make_EHelper(nop);
52 make_EHelper(in);
53 make_EHelper(out);

```

关于`lea`指令的执行函数已在框架代码`data-mov.c`中实现。

```

76 make_EHelper(lea) {
77     rtl_li(&t2, id_src->addr);
78     operand_write(id_dest, &t2);
79     print_asm_template2(lea);
80 }

```

接下来是按照顺序对其他指令的实现。

2.1 AND的实现

对AND指令，查阅i386手册如下，

AND — Logical AND

Opcode	Instruction	Clocks	Description
24 ib	AND AL,imm8	2	AND immediate byte to AL
25 iw	AND AX,imm16	2	AND immediate word to AX
25 id	AND EAX,imm32	2	AND immediate dword to EAX
80 /4 ib	AND r/m8,imm8	2/7	AND immediate byte to r/m byte
81 /4 iw	AND r/m16,imm16	2/7	AND immediate word to r/m word
81 /4 id	AND r/m32,imm32	2/7	AND immediate dword to r/m dword
83 /4 ib	AND r/m16,imm8	2/7	AND sign-extended immediate byte with r/m word
83 /4 ib	AND r/m32,imm8	2/7	AND sign-extended immediate byte with r/m dword
20 /r	AND r/m8,r8	2/7	AND byte register to r/m byte
21 /r	AND r/m16,r16	2/7	AND word register to r/m word
21 /r	AND r/m32,r32	2/7	AND dword register to r/m dword
22 /r	AND r8,r/m8	2/6	AND r/m byte to byte register
23 /r	AND r16,r/m16	2/6	AND r/m word to word register
23 /r	AND r32,r/m32	2/6	AND r/m dword to dword register

依照原理，首先在`logic.c`文件中添加执行函数`make_EHelper(and)`，代码如下。

```

9 make_EHelper(and) {
10    // TODO();
11    rtl_and(&t2, &id_dest->val, &id_src->val);
12    operand_write(id_dest, &t2); // 写回寄存器
13    // 修改eflags
14    // SF,ZF
15    rtl_update_ZFSF(&t2, id_dest->width);
16    // CF OF <- 0
17    rtl_set_CF(&tzero);
18    rtl_set_OF(&tzero);
19
20    print_asm_template2(and);
21 }

```

接下来按照手册中给出的指令及其地址，在`exec.c`文件中修改如下代码：

```

42 /* 0x80, 0x81, 0x83 */
43 make_group(gp1,
44     EMPTY, EMPTY, EMPTY, EMPTY,
45     EX(and), EX(sub), EX(xor), EMPTY)

83 /* 0x20 */    INDEXW(G2E, and, 1), INDEX(G2E, and), INDEX(E2G, and, 1), INDEX(E2G, and),
84 /* 0x24 */    INDEXW(I2a, and, 1), INDEX(I2a, and), EMPTY, EMPTY,

```

2.2 PUSH、PUSHL的实现

反汇编编码`PUSHL`相当于`PUSH DWORD`，对应opcode见如下i386手册内容。

PUSH — Push Operand onto the Stack

Opcode		Instruction	Clocks	Description
FF	/6	PUSH m16	5	Push memory word
FF	/6	PUSH m32	5	Push memory dword
50	+ /3	PUSH r16	2	Push register word
50	+ /3	PUSH r32	2	Push register dword
6A		PUSH imm8	2	Push immediate byte
68		PUSH imm16	2	Push immediate word
68		PUSH imm32	2	Push immediate dword
0E		PUSH CS	2	Push CS
16		PUSH SS	2	Push SS
1E		PUSH DS	2	Push DS
06		PUSH ES	2	Push ES
0F	A0	PUSH FS	2	Push FS
0F	A8	PUSH GS	2	Push GS

可以看到，opcode为`FF /6`及`68`，因此，扩展opcode如下。

```

62 /* 0xff */
63 make_group(gp5,
64     EMPTY, EMPTY, EMPTY, EMPTY,
65     EMPTY, EMPTY, EX(push), EMPTY)

101 /* 0x68 */    INDEX(I,push), EMPTY, INDEXW(push_SI,push,1), INDEX(I_E2G,imul3),

```

2.3 XCHG(NOP)的实现

继续运行 `add.c`, 程序在0x0010006a处停止如下。

```
invalid opcode(eip = 0x0010006a): 66 90 8d 34 3f 31 db 8d ...
There are two cases which will trigger this unexpected exception:
1. The instruction at eip = 0x0010006a is not implemented.
2. Something is implemented incorrectly.
Find this eip(0x0010006a) in the disassembling result to distinguish which case it is.
```

反汇编文件中对应指令为xchg, 查阅i386手册如下。

XCHG — Exchange Register/Memory with Register

Opcode	Instruction	Clocks	Description
90 + r	XCHG AX,r16	3	Exchange word register with AX
90 + r	XCHG r16,AX	3	Exchange word register with AX
90 + r	XCHG EAX,r32	3	Exchange dword register with EAX
90 + r	XCHG r32,EAX	3	Exchange dword register with EAX
86 /r	XCHG r/m8,r8	3	Exchange byte register with EA byte
86 /r	XCHG r8,r/m8	3/5	Exchange byte register with EA byte
87 /r	XCHG r/m16,r16	3	Exchange word register with EA word
87 /r	XCHG r16,r/m16	3/5	Exchange word register with EA word
87 /r	XCHG r/m32,r32	3	Exchange dword register with EA dword
87 /r	XCHG r32,r/m32	3/5	Exchange dword register with EA dword

手册中并没有给出opcode=66的指令, 根据反汇编码, 前缀0x66, opcode=90时, `xchg %ax %ax`什么都没做, 因此可以直接使用nop作为执行函数。添加opcode指令如下。

```
111 /* 0x90 */ EX(nop), EMPTY, EMPTY, EMPTY,
```

2.4 SETcc的实现

SETcc指令, 不是指一条指令, 而是指一系列形如SETcc的指令。查阅i386手册如下。

SETcc — Byte Set on Condition

Opcode	Instruction	Clocks	Description
0F 97	SETA r/m8	4/5	Set byte if above (CF=0 and ZF=0)
0F 93	SETAE r/m8	4/5	Set byte if above or equal (CF=0)
0F 92	SETB r/m8	4/5	Set byte if below (CF=1)
0F 96	SETBE r/m8	4/5	Set byte if below or equal (CF=1 or (ZF=1))
0F 92	SETC r/m8	4/5	Set if carry (CF=1)
0F 94	SETE r/m8	4/5	Set byte if equal (ZF=1)
0F 9F	SETG r/m8	4/5	Set byte if greater (ZF=0 or SF=OF)
0F 9D	SETGE r/m8	4/5	Set byte if greater or equal (SF=OF)
0F 9C	SETL r/m8	4/5	Set byte if less (SF≠OF)
0F 9E	SETLE r/m8	4/5	Set byte if less or equal (ZF=1 and SF≠OF)
0F 96	SETNA r/m8	4/5	Set byte if not above (CF=1)
0F 92	SETNAE r/m8	4/5	Set byte if not above or equal (CF=1)
0F 93	SETNB r/m8	4/5	Set byte if not below (CF=0)
0F 97	SETNBE r/m8	4/5	Set byte if not below or equal (CF=0 and ZF=0)
0F 93	SETNC r/m8	4/5	Set byte if not carry (CF=0)
0F 95	SETNE r/m8	4/5	Set byte if not equal (ZF=0)
0F 9E	SETNG r/m8	4/5	Set byte if not greater (ZF=1 or SF≠OF)
0F 9C	SETNGE r/m8	4/5	Set if not greater or equal (SF≠OF)
0F 9D	SETNL r/m8	4/5	Set byte if not less (SF=OF)
0F 9F	SETNLE r/m8	4/5	Set byte if not less or equal (ZF=1 and SF≠OF)
0F 91	SETNO r/m8	4/5	Set byte if not overflow (OF=0)
0F 9B	SETNP r/m8	4/5	Set byte if not parity (PF=0)
0F 99	SETNS r/m8	4/5	Set byte if not sign (SF=0)
0F 95	SETNZ r/m8	4/5	Set byte if not zero (ZF=0)
0F 90	SETO r/m8	4/5	Set byte if overflow (OF=1)
0F 9A	SETP r/m8	4/5	Set byte if parity (PF=1)
0F 9A	SETPE r/m8	4/5	Set byte if parity even (PF=1)
0F 9B	SETPO r/m8	4/5	Set byte if parity odd (PF=0)
0F 98	SETS r/m8	4/5	Set byte if sign (SF=1)
0F 94	SETZ r/m8	4/5	Set byte if zero (ZF=1)

这里需要根据条件设置字节，根据EFLAGS寄存器中的状态标志（CF、SF、OF、ZF、PF）将目标操作数设置成0或1，这里的目标操作数指向一个字节寄存器，也就是8位寄存器（AL、BL、CL）或内存中的一个字节。状态码后缀（cc）指明了将要测试的条件。

在拓展opcode修改代码如下。

```
52  /* 0xf6, 0xf7 */
53 make_group(gp3,
54     EMPTY, EMPTY, EMPTY, EMPTY,
55     EMPTY, EX(imul1), EMPTY, EMPTY)
```

根据需要实现0x94、0x95、0x9F，在2byte opcode中添加opcode_table如下。

```
112  /* 0x94 */     IDEWX(E, setcc, 1), IDEWX(E, setcc, 1), EMPTY, EMPTY,
113  /* 0x98 */     EMPTY, EMPTY, EMPTY, EMPTY,
114  /* 0x9c */     EMPTY, EMPTY, EMPTY, IDEWX(E, setcc, 1),
```

还需要实现cc.c中的rtl_setcc函数如下。

```
16  switch (subcode & 0xe) {
17      case CC_0:
18          rtl_get_OF(dest); //0
19          break;
20      case CC_B:
21          rtl_get_CF(dest); //2
22          break;
23      case CC_E:
24          rtl_get_ZF(dest); //4
25          break;
26      case CC_BE: //6 CF==0||ZF==0
27          assert(dest != &t0);
28          rtl_get_CF(dest);
29          rtl_get_ZF(&t0);
30          rtl_or(dest, dest, &t0);
31          break;
32      case CC_S: //8
33          rtl_get_SF(dest);
34          break;
35      case CC_L: //C SF!=OF
36          assert(dest != &t0);
37          rtl_get_SF(dest);
38          rtl_get_OF(&t0);
39          rtl_xor(dest, dest, &t0);
40          break;
41      case CC_LE: //E ZF==1||SF!=OF
42          assert(dest != &t0);
43          rtl_get_SF(dest);
44          rtl_get_OF(&t0);
45          rtl_xor(dest, dest, &t0); //CC_L
46          rtl_get_ZF(&t0);
47          rtl_or(dest, dest, &t0);
```

2.5 MOVZX, MOVSX的实现

查阅i386手册，MOVSX指令结构如下。

MOVSX — Move with Sign-Extend

Opcode	Instruction	Clocks	Description
0F BE /r	MOVsx r16,r/m8	3/6	Move byte to word with sign-extend
0F BE /r	MOVsx r32,r/m8	3/6	Move byte to dword, sign-extend
0F BF /r	MOVsx r32,r/m16	3/6	Move word to dword, sign-extend

Operation

```
DEST ← SignExtend(SRC);
```

MOVZX指令结构如下。

MOVZX — Move with Zero-Extend

Opcode	Instruction	Clocks	Description
0F B6 /r	MOVzx r16,r/m8	3/6	Move byte to word with zero-extend
0F B6 /r	MOVzx r32,r/m8	3/6	Move byte to dword, zero-extend
0F B7 /r	MOVzx r32,r/m16	3/6	Move word to dword, zero-extend

Operation

```
DEST ← ZeroExtend(SRC);
```

在2byte_opcode_table中添加如下代码。

```
187 /* 0xb4 */    EMPTY, EMPTY, IDEWX(mov_E2G,movzx,1), IDEWX(mov_E2G,movzx,2),
188 /* 0xb8 */    EMPTY, EMPTY, EMPTY, EMPTY,
189 /* 0xbc */    EMPTY, EMPTY, IDEWX(mov_E2G,movsx,1), IDEWX(mov_E2G,movsx,2),
```

2.6 JCC的实现

查阅i386手册中关于jcc指令的部分，由于jcc指令数量太多，这里就不放截图了，根据手册中jcc指令的结构，在1byte_opcode_table中添加opcode如下。

```
103 /* 0x70 */    EMPTY, EMPTY, IDEWX(J,jcc,1), IDEWX(J,jcc,1),
104 /* 0x74 */    IDEWX(J,jcc,1), IDEWX(J,jcc,1), IDEWX(J,jcc,1), IDEWX(J,jcc,1),
105 /* 0x78 */    IDEWX(J,jcc,1), IDEWX(J,jcc,1), IDEWX(J,jcc,1), IDEWX(J,jcc,1),
106 /* 0x7c */    IDEWX(J,jcc,1), IDEWX(J,jcc,1), IDEWX(J,jcc,1), IDEWX(J,jcc,1),
```

```
131 /* 0xe0 */    EMPTY, EMPTY, EMPTY, IDEWX(J,jcc,1),
```

在2byte_opcode_table中添加代码如下。

```
174 /* 0x80 */    IDEX(J,jcc), IDEX(J,jcc), IDEX(J,jcc), IDEX(J,jcc),
175 /* 0x84 */    IDEX(J,jcc), IDEX(J,jcc), IDEX(J,jcc), IDEX(J,jcc),
176 /* 0x88 */    IDEX(J,jcc), IDEX(J,jcc), IDEX(J,jcc), EMPTY, EMPTY,
177 /* 0x8c */    IDEX(J,jcc), IDEX(J,jcc), IDEX(J,jcc), IDEX(J,jcc),
```

2.7 SAR、SAL、SHL、SHR的实现

首先，查阅i386手册如下。

SAL/SAR/SHL/SHR — Shift Instructions

Opcode	Instruction	Clocks	Description
D0 /4	SAL r/m8,1	3/7	Multiply r/m byte by 2, once
D2 /4	SAL r/m8,CL	3/7	Multiply r/m byte by 2, CL times
C0 /4 ib	SAL r/m8,imm8	3/7	Multiply r/m byte by 2, imm8 times
D1 /4	SAL r/m16,1	3/7	Multiply r/m word by 2, once
D3 /4	SAL r/m16,CL	3/7	Multiply r/m word by 2, CL times
C1 /4 ib	SAL r/m16,imm8	3/7	Multiply r/m word by 2, imm8 times
D1 /4	SAL r/m32,1	3/7	Multiply r/m dword by 2, once
D3 /4	SAL r/m32,CL	3/7	Multiply r/m dword by 2, CL times
C1 /4 ib	SAL r/m32,imm8	3/7	Multiply r/m dword by 2, imm8 times
D0 /7	SAR r/m8,1	3/7	Signed divide^(1) r/m byte by 2, once
D2 /7	SAR r/m8,CL	3/7	Signed divide^(1) r/m byte by 2, CL times
C0 /7 ib	SAR r/m8,imm8	3/7	Signed divide^(1) r/m byte by 2, imm8 times
D1 /7	SAR r/m16,1	3/7	Signed divide^(1) r/m word by 2, once
D3 /7	SAR r/m16,CL	3/7	Signed divide^(1) r/m word by 2, CL times
C1 /7 ib	SAR r/m16,imm8	3/7	Signed divide^(1) r/m word by 2, imm8 times
D1 /7	SAR r/m32,1	3/7	Signed divide^(1) r/m dword by 2, once
D3 /7	SAR r/m32,CL	3/7	Signed divide^(1) r/m dword by 2, CL times
C1 /7 ib	SAR r/m32,imm8	3/7	Signed divide^(1) r/m dword by 2, imm8 times
D0 /4	SHL r/m8,1	3/7	Multiply r/m byte by 2, once
D2 /4	SHL r/m8,CL	3/7	Multiply r/m byte by 2, CL times
C0 /4 ib	SHL r/m8,imm8	3/7	Multiply r/m byte by 2, imm8 times
D1 /4	SHL r/m16,1	3/7	Multiply r/m word by 2, once
D3 /4	SHL r/m16,CL	3/7	Multiply r/m word by 2, CL times
C1 /4 ib	SHL r/m16,imm8	3/7	Multiply r/m word by 2, imm8 times
D1 /4	SHL r/m32,1	3/7	Multiply r/m dword by 2, once
D3 /4	SHL r/m32,CL	3/7	Multiply r/m dword by 2, CL times
C1 /4 ib	SHL r/m32,imm8	3/7	Multiply r/m dword by 2, imm8 times
D0 /5	SHR r/m8,1	3/7	Unsigned divide r/m byte by 2, once
D2 /5	SHR r/m8,CL	3/7	Unsigned divide r/m byte by 2, CL times
C0 /5 ib	SHR r/m8,imm8	3/7	Unsigned divide r/m byte by 2, imm8 times
D1 /5	SHR r/m16,1	3/7	Unsigned divide r/m word by 2, once
D3 /5	SHR r/m16,CL	3/7	Unsigned divide r/m word by 2, CL times
C1 /5 ib	SHR r/m16,imm8	3/7	Unsigned divide r/m word by 2, imm8 times
D1 /5	SHR r/m32,1	3/7	Unsigned divide r/m dword by 2, once
D3 /5	SHR r/m32,CL	3/7	Unsigned divide r/m dword by 2, CL times
C1 /5 ib	SHR r/m32,imm8	3/7	Unsigned divide r/m dword by 2, imm8 times

Not the same division as IDIV; rounding is toward negative infinity.

SHL、SAL: 每位左移, 低位补0, 高位进CF

SAR: 每位右移, 低位进CF, 高位补0

SHR: 每位右移, 低位进CF, 高位保持 (这里指原数据高位)

该指令的执行函数没有在框架中给出, 需要自己去添加, 首先在 `logic.c` 中添加SHL、SHR指令如下。

```

50 make_EHelper(shl) {
51     // TODO();
52     // unnecessary to update CF and OF in NEMU
53     rtl_shl(&t2, &id_dest->val, &id_src->val);
54     operand_write(id_dest, &t2); // 写回寄存器
55     // 修改eflags
56     // SF ZF
57     rtl_update_ZFSF(&t2, id_dest->width);
58
59     print_asm_template2(shl);
60 }

```

```

62 make_EHelper(shr) {
63     // TODO();
64     // unnecessary to update CF and OF in NEMU
65     rtl_shr(&t2, &id_dest->val, &id_src->val);
66     operand_write(id_dest, &t2); // 写回寄存器
67     // 修改eflags:SF ZF
68     rtl_update_ZFSF(&t2, id_dest->width);
69     print_asm_template2(shr);
70 }

```

在 `rtl.h` 中添加实现SAR需要的函数如下。

```

129 static inline void rtl_mv(rtlreg_t* dest, const rtlreg_t *src1) {
130     // dest <- src1
131     // TODO();
132     rtl_addi(dest, src1, 0);
133 } // 数据移动
134
135 static inline void rtl_not(rtlreg_t* dest) {
136     // dest <- ~dest
137     // TODO();
138     rtl_xori(dest, dest, 0xffffffff);
139 }
140
141 static inline void rtl_sext(rtlreg_t* dest, const rtlreg_t* src1, int width)
{
142     // dest <- signext(src1[(width * 8 - 1) .. 0])
143     // TODO();
144     if(width == 4){
145         rtl_mv(dest, src1);
146     }
147     else{
148         assert(width == 1 || width == 2);
149         rtl_shli(dest, src1, (4 - width) * 8);
150         rtl_sari(dest, dest, (4 - width) * 8);
151     }
152 }

```

在 `logic.c` 中添加执行函数如下。

```

43 make_EHelper(sar) {
44     // TODO();
45     // unnecessary to update CF and OF in NEMU
46     rtl_sext(&t2, &id_dest->val, id_dest->width);
47     rtl_sar(&t2, &t2, &id_src->val);
48     operand_write(id_dest, &t2);
49     rtl_update_ZFSF(&t2, id_dest->width);
50
51     print_asm_template2(sar);
52 }

```

在 `exec.c` 中拓展 `opcode_table` 如下。

```

47 /* 0xc0, 0xc1, 0xd0, 0xd1, 0xd2, 0xd3 */
48 make_group(gp2,
49     EMPTY, EMPTY, EMPTY, EMPTY,
50     EX(shl), EX(shr), EMPTY, EX(sar))

```

2.8 TEST的实现

查阅i386手册，TEST指令的功能如下。

TEST — Logical Compare

Opcode		Instruction	Clocks	Description
A8	ib	TEST AL,imm8	2	AND immediate byte with AL
A9	iw	TEST AX,imm16	2	AND immediate word with AX
A9	id	TEST EAX,imm32	2	AND immediate dword with EAX
F6	/0 ib	TEST r/m8,imm8	2/5	AND immediate byte with r/m byte
F7	/0 iw	TEST r/m16,imm16	2/5	AND immediate word with r/m word
F7	/0 id	TEST r/m32,imm32	2/5	AND immediate dword with r/m dword
84	/r	TEST r/m8,r8	2/5	AND byte register with r/m byte
85	/r	TEST r/m16,r16	2/5	AND word register with r/m word
85	/r	TEST r/m32,r32	2/5	AND dword register with r/m dword

TEST指令功能为执行BIT与BIT之间的逻辑运算，Test对两个参数执行AND逻辑操作，并根据结果设置标志寄存器，结果本身不会保存。

在 `logic.c` 中添加该指令执行函数如下。

```

3 make_EHelper(test) {
4     // TODO();
5     rtl_and(&t2, &id_dest->val, &id_src->val);
6     rtl_update_ZFSF(&t2, id_dest->width);
7     rtl_set_CF(&tzero);
8     rtl_set_OF(&tzero);
9
10    print_asm_template2(test);
11 }

```

在 `exec.c` 中拓展 `opcode_table` 如下。

```

52     /* 0xf6, 0xf7 */
53 make_group(gp3,
54     IDEX(test_I,test), EMPTY, EMPTY, EMPTY,
55     EMPTY, EX(imul1), EMPTY, EMPTY)

108    /* 0x84 */    IDEXW(G2E,test,1), IDEX(G2E,test), EMPTY, EMPTY,
117    /* 0xa8 */    IDEXW(I2a,test,1), IDEX(I2a,test), EMPTY, EMPTY,

```

2.9 ADD的实现

查阅 i386 手册中关于 ADD 指令结构的介绍。

ADD — Add

Opcode	Instruction	Clocks	Description
04 ib	ADD AL,imm8	2	Add immediate byte to AL
05 iw	ADD AX,imm16	2	Add immediate word to AX
05 id	ADD EAX,imm32	2	Add immediate dword to EAX
80 /0 ib	ADD r/m8,imm8	2/7	Add immediate byte to r/m byte
81 /0 iw	ADD r/m16,imm16	2/7	Add immediate word to r/m word
81 /0 id	ADD r/m32,imm32	2/7	Add immediate dword to r/m dword
83 /0 ib	ADD r/m16,imm8	2/7	Add sign-extended immediate byte to r/m word
83 /0 ib	ADD r/m32,imm8	2/7	Add sign-extended immediate byte to r/m dword
00 /r	ADD r/m8,r8	2/7	Add byte register to r/m byte
01 /r	ADD r/m16,r16	2/7	Add word register to r/m word
01 /r	ADD r/m32,r32	2/7	Add dword register to r/m dword
02 /r	ADD r8,r/m8	2/6	Add r/m byte to byte register
03 /r	ADD r16,r/m16	2/6	Add r/m word to word register
03 /r	ADD r32,r/m32	2/6	Add r/m dword to dword register

Operation

`DEST ← DEST + SRC;`

首先在 `arith.c` 文件下实现执行函数如下。

```

21 make_EHelper(add) {
22     // TODO();
23     rtl_and(&t2, &id_dest->val, &id_src->val);
24     operand_write(id_dest, &t2);
25     //ZF SF
26     rtl_update_ZFSF(&t2, id_dest->width);
27     //CF=1判断: 1+1=0 (进位), 即当无符号比较, 结果<任意加数时, CF=1
28     rtl_sltu(&t0, &t2, &id_dest->val);
29     rtl_set_CF(&t0);
30     //OF的判断: 正+正=负或负+负=正时发生溢出
31     rtl_xor(&t0, &id_src->val, &t2);
32     rtl_xor(&t1, &id_dest->val, &t2);
33     rtl_and(&t0, &t0, &t1);
34     rtl_msb(&t0, &t0, id_dest->width);
35     rtl_set_OF(&t0);
36
37     print_asm_template2(add);
38 }

```

在 `exec.c` 下修改 `opcode_table` 如下。

```

42 /* 0x80, 0x81, 0x83 */
43 make_group(gp1,
44     EX(add), EMPTY, EMPTY, EMPTY,
45     EX(and), EX(sub), EX(xor), EMPTY)

```

```

75 /* 0x00 */    IDEWX(G2E,add,1), IDEWX(G2E,add), IDEWX(E2G,add,1), IDEX(E2G,add),
76 /* 0x04 */    IDEWX(I2a,add,1), IDEX(I2a,add), EMPTY, EMPTY,

```

2.10 CMP的实现

查阅i386手册中关于CMP指令的部分。

CMP — Compare Two Operands

Opcode	Instruction	Clocks	Description
3C ib	CMP AL,imm8	2	Compare immediate byte to AL
3D iw	CMP AX,imm16	2	Compare immediate word to AX
3D id	CMP EAX,imm32	2	Compare immediate dword to EAX
80 /7 ib	CMP r/m8,imm8	2/5	Compare immediate byte to r/m byte
81 /7 iw	CMP r/m16,imm16	2/5	Compare immediate word to r/m word
81 /7 id	CMP r/m32,imm32	2/5	Compare immediate dword to r/m dword
83 /7 ib	CMP r/m16,imm8	2/5	Compare sign extended immediate byte to r/m word
83 /7 ib	CMP r/m32,imm8	2/5	Compare sign extended immediate byte to r/m dword
38 /r	CMP r/m8,r8	2/5	Compare byte register to r/m byte
39 /r	CMP r/m16,r16	2/5	Compare word register to r/m word
39 /r	CMP r/m32,r32	2/5	Compare dword register to r/m dword
3A /r	CMP r8,r/m8	2/6	Compare r/m byte to byte register
3B /r	CMP r16,r/m16	2/6	Compare r/m word to word register
3B /r	CMP r32,r/m32	2/6	Compare r/m dword to dword register

CMP指令的操作与SUB类似，只是不进行结果的存储，而只修改EFLAGS的标志。首先在 `arith.c` 中添加执行函数如下。

```

47 make_EHelper(cmp) {
48     // TODO();
49     eflags_modify();
50
51     print_asm_template2(cmp);
52 }

```

接下来在 `exec.c` 文件中修改opcode如下。

```

89 /* 0x38 */    IDEWX(G2E,cmp,1), IDEX(G2E,cmp), IDEWX(E2G,cmp,1), IDEX(E2G,cmp),
90 /* 0x3c */    IDEWX(I2a,cmp,1), IDEX(I2a,cmp), EMPTY, EMPTY,

```

```

42 /* 0x80, 0x81, 0x83 */
43 make_group(gp1,
44     EX(add), EMPTY, EMPTY, EMPTY,
45     EX(and), EX(sub), EX(xor), EX(cmp))

```

2.11 JMP的实现

i386手册中关于JMP指令的叙述如下。

JMP —— Jump

Opcode	Instruction	Clocks	Description
EB cb	JMP rel8	7+m	Jump short
E9 cw	JMP rel16	7+m	Jump near, displacement relative to next instruction
FF /4	JMP r/m16	7+m/10+m	Jump near indirect
EA cd	JMP ptr16:16	12+m, pm=27+m	Jump intersegment, 4-byte immediate address
EA cd	JMP ptr16:16	pm=45+m	Jump to call gate, same privilege
EA cd	JMP ptr16:16	ts	Jump via task state segment
EA cd	JMP ptr16:16	ts	Jump via task gate
FF /5	JMP m16:16	43+m, pm=31+m	Jump r/m16:16 indirect and intersegment
FF /5	JMP m16:16	pm=49+m	Jump to call gate, same privilege
FF /5	JMP m16:16	5 + ts	Jump via task state segment
FF /5	JMP m16:16	5 + ts	Jump via task gate
E9 cd	JMP rel32	7+m	Jump near, displacement relative to next instruction
FF /4	JMP r/m32	7+m, 10+m	Jump near, indirect
EA cp	JMP ptr16:32	12+m, pm=27+m	Jump intersegment, 6-byte immediate address
EA cp	JMP ptr16:32	pm=45+m	Jump to call gate, same privilege
EA cp	JMP ptr16:32	ts	Jump via task state segment
EA cp	JMP ptr16:32	ts	Jump via task gate
FF /5	JMP m16:32	43+m, pm=31+m	Jump intersegment, address at r/m dword
FF /5	JMP m16:32	pm=49+m	Jump to call gate, same privilege
FF /5	JMP m16:32	5 + ts	Jump via task state segment
FF /5	JMP m16:32	5 + ts	Jump via task gate

该指令框架代码中已经实现，只需填写opcode表即可。测试样例中用了0xEB和0xE9，这里实现这两个指令。修改opcode表如下。

```
62 /* 0xff */
63 make_group(gp5,
64     EX(inc), EX(dec), EX(call_rm), EMPTY,
65     EX(jmp_rm), EMPTY, EX(push), EMPTY)

133 /* 0xe8 */    IDEX(J,call), IDEX(J,jmp), EMPTY, IDEWX(J,jmp,1),
```

2.12 MUL的实现

查阅i386手册如下。

MUL —— Unsigned Multiplication of AL or AX

Opcode	Instruction	Clocks	Description
F6 /4	MUL AL,r/m8	9-14/12-17	Unsigned multiply (AX ← AL * r/m byte)
F7 /4	MUL AX,r/m16	9-22/12-25	Unsigned multiply (DX:AX ← AX * r/m word)
F7 /4	MUL EAX,r/m32	9-38/12-41	Unsigned multiply (EDX:EAX ← EAX * r/m dword)

在exec.c中添加指令如下。

```
52 /* 0xf6, 0xf7 */
53 make_group(gp3,
54     IDEX(test_I,test), EMPTY, EX(not), EMPTY,
55     EX(mul), EX(imul1), EMPTY, EMPTY)
```

2.13 IMUL的实现

查阅i386手册如下。

IMUL — Signed Multiply

Opcode	Instruction	Clocks	Description
F6 /5	IMUL r/m8	9-14/12-17	AX ← AL * r/m byte
F7 /5	IMUL r/m16	9-22/12-25	DX:AX ← AX * r/m word
F7 /5	IMUL r/m32	9-38/12-41	EDX:EAX ← EAX * r/m dword
OF AF /r	IMUL r16,r/m16	9-22/12-25	word register ← word register * r/m word
OF AF /r	IMUL r32,r/m32	9-38/12-41	dword register ← dword register * r/m dword
6B /r ib	IMUL r16,r/m16,imm8	9-14/12-17	word register ← r/m16 * sign-extended immediate byte
6B /r ib	IMUL r32,r/m32,imm8	9-14/12-17	dword register ← r/m32 * sign-extended immediate byte
6B /r ib	IMUL r16,imm8	9-14/12-17	word register ← word register * sign-extended immediate byte
6B /r ib	IMUL r32,imm8	9-14/12-17	dword register ← dword register * sign-extended immediate byte
69 /r iw	IMUL r16,r/m16,imm16	9-22/12-25	word register ← r/m16 * immediate word
69 /r id	IMUL r32,r/m32,imm32	9-38/12-41	dword register ← r/m32 * immediate dword
69 /r iw	IMUL r16,imm16	9-22/12-25	word register ← r/m16 * immediate word
69 /r id	IMUL r32,imm32	9-38/12-41	dword register ← r/m32 * immediate dword

在 `exec.c` 文件中添加代码如下。

```
52 /* 0xf6, 0xf7 */
53 make_group(gp3,
54     IDEX(test_I,test), EMPTY, EX(not), EMPTY,
55     EX(mul), EX(imul1), EMPTY, EMPTY)
```

```
185 /* 0xac */
    EMPTY, EMPTY, EMPTY, IDEX(E2G,imul2),
```

2.14 DIV的实现

查阅i386手册如下。

DIV — Unsigned Divide

Opcode	Instruction	Clocks	Description
F6 /6	DIV AL,r/m8	14/17	Unsigned divide AX by r/m byte (AL=Quo, AH=Rem)
F7 /6	DIV AX,r/m16	22/25	Unsigned divide DX:AX by r/m word (AX=Quo, DX=Rem)
F7 /6	DIV EAX,r/m32	38/41	Unsigned divide EDX:EAX by r/m dword (EAX=Quo, EDX=Rem)

在 `exec.c` 中添加如下代码。

```
52 /* 0xf6, 0xf7 */
53 make_group(gp3,
54     IDEX(test_I,test), EMPTY, EX(not), EMPTY,
55     EX(mul), EX(div), EMPTY)
```

2.15 IDIV的实现

查阅i386手册如下。

IDIV — Signed Divide

Opcode	Instruction	Clocks	Description
F6 /7	IDIV r/m8	19	Signed divide AX by r/m byte (AL=Quo, AH=Rem)
F7 /7	IDIV AX,r/m16	27	Signed divide DX:AX by EA word (AX=Quo, DX=Rem)
F7 /7	IDIV EAX,r/m32	43	Signed divide EDX:EAX by DWORD byte (EAX=Quo, EDX=Rem)

在 `exec.c` 中添加如下代码。

```

52 /* 0xf6, 0xf7 */
53 make_group(gp3,
54     INDEX(test_I,test), EMPTY, EX(not), EMPTY,
55     EX(mul), EX(imul1), EX(div), EX(idiv))

```

2.16 ADC的实现

查阅i386手册如下。

ADC — Add with Carry

Opcode	Instruction	Clocks	Description
14 ib	ADC AL,imm8	2	Add with carry immediate byte to AL
15 iw	ADC AX,imm16	2	Add with carry immediate word to AX
15 id	ADC EAX,imm32	2	Add with carry immediate dword to EAX
80 /2 ib	ADC r/m8,imm8	2/7	Add with carry immediate byte to r/m byte
81 /2 iw	ADC r/m16,imm16	2/7	Add with carry immediate word to r/m word
81 /2 id	ADC r/m32,imm32	2/7	Add with CF immediate dword to r/m dword
83 /2 ib	ADC r/m16,imm8	2/7	Add with CF sign-extended immediate byte to r/m word
83 /2 id	ADC r/m32,imm8	2/7	Add with CF sign-extended immediate byte into r/m dword
10 /r	ADC r/m8,r8	2/7	Add with carry byte register to r/m byte
11 /r	ADC r/m16,r16	2/7	Add with carry word register to r/m word
11 /r	ADC r/m32,r32	2/7	Add with CF dword register to r/m dword
12 /r	ADC r8,r/m8	2/6	Add with carry r/m byte to byte register
13 /r	ADC r16,r/m16	2/6	Add with carry r/m word to word register
13 /r	ADC r32,r/m32	2/6	Add with CF r/m dword to dword register

在 `exec.c` 中添加如下代码。

```

42 /* 0x80, 0x81, 0x83 */
43 make_group(gp1,
44     EX(add), EMPTY, EXadc), EMPTY,
45     EXand), EXsub), EXxor), EXcmp))

79 /* 0x10 */ INDEX(G2E,adc,1), INDEX(G2E,adc), INDEXW(E2G,adc,1), INDEX(E2G,adc),
80 /* 0x14 */ INDEXW(I2a,adc,1), INDEX(I2a,adc), EMPTY, EMPTY,

```

2.17 SBB的实现

查阅i386手册如下。

SBB — Integer Subtraction with Borrow

Opcode	Instruction	Clocks	Description
1C ib	SBB AL,imm8	2	Subtract with borrow immediate byte from AL
1D iw	SBB AX,imm16	2	Subtract with borrow immediate word from AX
1D id	SBB EAX,imm32	2	Subtract with borrow immediate dword from EAX
80 /3 ib	SBB r/m8,imm8	2/7	Subtract with borrow immediate byte from r/m byte
81 /3 iw	SBB r/m16,imm16	2/7	Subtract with borrow immediate from r/m word
81 /3 id	SBB r/m32,imm32	2/7	Subtract with borrow immediate dword from r/m dword
83 /3 ib	SBB r/m16,imm8	2/7	Subtract with borrow sign-extended immediate byte from r/m word
83 /3 ib	SBB r/m32,imm8	2/7	Subtract with borrow sign-extended immediate byte from r/m dword
18 /r	SBB r/m8,r8	2/6	Subtract with borrow byte register from r/m byte
19 /r	SBB r/m16,r16	2/6	Subtract with borrow word register from r/m word
19 /r	SBB r/m32,r32	2/6	Subtract with borrow dword from r/m dword
1A /r	SBB r8,r/m8	2/7	Subtract with borrow byte register from r/m byte
1B /r	SBB r16,r/m16	2/7	Subtract with borrow word register from r/m word
1B /r	SBB r32,r/m32	2/7	Subtract with borrow dword register from r/m dword

在 `exec.c` 中添加如下代码。

```

42 /* 0x80, 0x81, 0x83 */
43 make_group(gp1,
44     EXadd), EMPTY, EXadc), EXsbb),
45     EXand), EXsub), EXxor), EXcmp))

```

```
81  /* 0x18 */    IDEWX(G2E,sbb,1), IDEX(G2E,sbb), IDEWX(E2G,sbb,1), IDEX(E2G,sbb),
82  /* 0x1c */    IDEWX(I2a,sbb,1), IDEX(I2a,sbb), EMPTY, EMPTY,
```

2.18 NEG的实现

查阅i386手册如下。

NEG — Two's Complement Negation

Opcode	Instruction	Clocks	Description
F6 /3	NEG r/m8	2/6	Two's complement negate r/m byte
F7 /3	NEG r/m16	2/6	Two's complement negate r/m word
F7 /3	NEG r/m32	2/6	Two's complement negate r/m dword

NEG是汇编指令中的求补指令，对操作数执行求补运算：用零减去操作数，然后结果返回操作数。求补运算也可以表达成：将操作数按位取反后加1.

在 `arith.c` 中添加执行函数如下。

```
66 make_EHelper(neg) {
67     // TODO();
68     rtl_sub(&t2, &tzero, &id_dest->val);
69     rtl_update_ZFSF(&t2, id_dest->width);
70     rtl_neq0(&t0, &id_dest->val);
71     rtl_set_CF(&t0);
72     rtl_eqi(&t0, &id_dest->val, 0x80000000);
73     rtl_set_OF(&t0);
74     operand_write(id_dest, &t2);
75
76     print_asm_template1(neg);
77 }
```

在 `exec.c` 中补充opcode如下。

```
52  /* 0xf6, 0xf7 */
53 make_group(gp3,
54     IDEX(test_I,test), EMPTY, EX(not), EX(neg),
55     EX(mul), EX(imul1), EX(div), EX(idiv))
```

2.19 OR的实现

首先修改eflags寄存器。在 `logic.c` 中添加执行函数。

```

41 make_EHelper(or) {
42     // TODO();
43     rtl_or(&t2, &id_dest->val, &id_src->val);
44     operand_write(id_dest, &t2);
45     // SF ZF
46     rtl_update_ZFSF(&t2, id_dest->width);
47     // CF OF <- 0
48     rtl_set_CF(&tzero);
49     rtl_set_OF(&tzero);
50
51     print_asm_template2(or);
52 }

```

在 `exec.c` 中修改 opcode 表如下。

```

77 /* 0x08 */    IDEWX(G2E,or,1), IDEWX(G2E,or), IDEWX(E2G,or,1), IDEX(E2G,or),
78 /* 0x0c */    IDEWX(I2a,or,1), IDEX(I2a,or), EMPTY, EX(2byte_esc),

```

```

42 /* 0x80, 0x81, 0x83 */
43 make_group(gp1,
44     EX(add), EX(or), EXadc), EX(sbb),
45     EX(and), EX(sub), EX(xor), EX(cmp))

```

2.20 NOT的实现

在 `logic.c` 中添加执行函数如下。

```

95 make_EHelper(not) {
96     // TODO();
97     rtl_not(&id_dest->val);
98     operand_write(id_dest, &id_dest->val);
99
100    print_asm_template1(not);
101 }

```

在 `exec.c` 中修改 opcode、表如下。

```

52 /* 0xf6, 0xf7 */
53 make_group(gp3,
54     IDEX(test_I,test), EMPTY, EX(not), EX(neg),
55     EX(mul), EX(imul1), EX(div), EX(idiv))

```

2.21 DEC的实现

根据 i386 手册，DEC 指令不修改 CF 的标志位，在 `arith.c` 中添加执行函数如下。

```
60 make_EHelper(dec) {
61     // TODO();
62     rtl_subi(&t2, &id_dest->val, 1);
63     operand_write(id_dest, &t2);
64     // ZF SF
65     rtl_update_ZFSF(&t2, id_dest->width);
66     // OF
67     rtl_eqi(&t0, &t2, 0xffffffff);
68     rt_set_OF(&t0);
69
70     print_asm_template1(dec);
71 }
```

在 `exec.c` 中修改 opcode 表如下。

```
57 /* 0xfe */
58 make_group(gp4,
59             EMPTY, EX(dec), EMPTY, EMPTY,
60             EMPTY, EMPTY, EMPTY, EMPTY)
61
62 /* 0xff */
63 make_group(gp5,
64             EX(inc), EX(dec), EX(call_rm), EMPTY,
65             EX(jmp_rm), EMPTY, EX(push), EMPTY)
93 /* 0x48 */    INDEX(r,dec), INDEX(r,dec), INDEX(r,dec), INDEX(r,dec),
94 /* 0x4c */    EMPTY, EMPTY, INDEX(r,dec), INDEX(r,dec),
```

2.22 INC的实现

在 `arith.c` 中添加执行函数如下。

```
54 make_EHelper(inc) {
55     // TODO();
56     rtl_addi(&t2, &id_dest->val, 1);
57     operand_write(id_dest, &t2);
58     // ZF SF
59     rtl_update_ZFSF(&t2, id_dest->width);
60     // OF
61     rtl_eqi(&t0, &t2, 0x80000000);
62     rt_set_OF(&t0);
63
64     print_asm_template1(inc);
65 }
```

在 `exec.c` 中添加如下代码。

```
57 /* 0xfe */
58 make_group(gp4,
59             EX(inc), EX(dec), EMPTY, EMPTY,
60             EMPTY, EMPTY, EMPTY, EMPTY)
61
62 /* 0xff */
63 make_group(gp5,
64             EX(inc), EX(dec), EX(call_rm), EMPTY,
65             EX(jmp_rm), EMPTY, EX(push), EMPTY)
```

```
91  /* 0x40 */      INDEX(r,inc), INDEX(r,inc), INDEX(r,inc), INDEX(r,inc),
92  /* 0x44 */      EMPTY, EMPTY, INDEX(r,inc), INDEX(r,inc),
```

2.23 CLTD的实现

AT&T汇编的cltd指令相当于cdq指令，作用是把eax的32位整数扩展为64位，高32位用eax的符号位填充保存到edx，或ax的16位整数扩展为32位，高16位用ax的符号位填充保存到dx。

在 `data-mov.c` 中添加执行函数如下。

```
41 make_EHelper(cltd) {
42     if (decoding.is_operand_size_16) {
43         // TODO();
44         rtl_lr_w(&t0, R_AX);
45         rtl_sext(&t0, &t0, 2); // 符号扩展
46         rtl_sari(&t0, &t0, 31);
47         rtl_sr_w(R_DX, &t0);
48     }
49     else {
50         // TODO();
51         rtl_sari(&cpu.edx, &cpu.eax, 31);
52     }
53     print_asm(decoding.is_operand_size_16 ? "cwtl" : "cltd");
55 }
```

在 `exec.c` 中补充opcode表如下。

```
113  /* 0x98 */      EMPTY, EX(cltd), EMPTY, EMPTY,
```

2.24 LEAVE的实现

`LEAVE` 指令是将栈指针指向帧指针，然后POP备份的原帧指针到%EBP。
i386手册中介绍如下。

LEAVE — High Level Procedure Exit

Opcode	Instruction	Clocks	Description
C9	LEAVE	4	Set SP to BP, then pop BP
C9	LEAVE	4	Set ESP to EBP, then pop EBP

Operation

```
IF StackAddrSize = 16
THEN
    SP ← BP;
ELSE (* StackAddrSize = 32 *)
    ESP ← EBP;
FI;
IF OperandSize = 16
THEN
    BP ← Pop();
ELSE (* OperandSize = 32 *)
    EBP ← Pop();
FI;
```

在 `data-mov.c` 中添加执行函数如下。

```
35 make_EHelper(leave) {
36     // TODO();
37     rtl_mv(&cpu.esp, &cpu.ebp);
38     rtl_pop(&cpu.ebp);
39
40     print_asm("leave");
41 }
```

在 `exec.c` 中补充 opcode 表如下。

```
125 /* 0xc8 */    EMPTY, EX(leave), EMPTY, EMPTY,
```

2.25 CALL 补充

对 `FF /2` 的 `CALL` 指令进行补充。

在 `control.c` 中添加执行函数如下。

```
47 make_EHelper(call_rm) {
48     // TODO();
49     rtl_li(&t2, decoding.seq_eip);
50     rtl_push(&t2);
51     decoding.jmp_eip = id_dest->val;
52     decoding.is_jmp = 1;
53
54     print_asm("call *%s", id_dest->str);
55 }
```

在 `exec.c` 中添加 opcode 扩展表如下。

```
62 /* 0xff */
63 make_group(gp5,
64             EX(inc), EX(dec), EX(call_rm), EMPTY,
65             EX(jmp_rm), EMPTY, EX(push), EMPTY)
```

2.26 运行与结果

运行代码，发现在 `0x0010032` 处出现错误如下。

```
[src/monitor/monitor.c,30,welcome] Build time: 22:29:01, Apr 30 2020
For help, type "help"
(nemu) c
invalid opcode(eip = 0x00100032): 85 c0 74 02 5d c3 c7 45 ...
There are two cases which will trigger this unexpected exception:
1. The instruction at eip = 0x00100032 is not implemented.
2. Something is implemented incorrectly.
Find this eip(0x00100032) in the disassembling result to distinguish which case
it is.
```

检查反汇编文件，发现代码在 `0x0010032` 处的指令为 `85 c0 test %eax`，而此处的 `test` 指令未添加至 `opcode` 表中，因此加入如下代码。

```
108 /* 0x84 */    IDEWX(G2E,test,1), IDEX(G2E,test), EMPTY, EMPTY,
```

再次执行程序，成功运行。

```

sun@ests:~/Desktop/ics2018/nexus-am/tests/cputest$ make ARCH=x86-nemu ALL=add run
Building add [x86-nemu]
Building am [x86-nemu]
make[2]: *** 没有指明目标并且找不到 makefile。 停止。
+ CC src/cpu/exec/exec.c
+ CC src/cpu/exec/logic.c
+ LD build/nemu
[src/monitor/monitor.c,65,load_img] The image is /home/sun/Desktop/ics2018/nexus
-am/tests/cputest/build/add-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 22:29:01, Apr 30 2020
For help, type "help"
(nemu) c
nemu: HIT GOOD TRAP at eip = 0x00100027

```

3. 基础设施

3.1 代码：实现differential testing

实现 differential testing

在 `difftest_step()` 中添加相应的代码，实现 differential testing 的核心功能。实现正确后，你将会得到一款无比强大的测试工具。

根据实验指导，nemu中已经准备好利用differential testing测试功能实现正确性的相应环境。首先我们先在 `nemu/include/common.h` 中定义宏 `DIFF_TEST`，之后重新编译nemu如下。

```

[src/monitor/diff-test/diff-test.c,96,init_difftest] Connect to QEMU successfully
[src/monitor/monitor.c,47,load_default_img] No image is given. Use the default built-in image.
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 10:25:27, May 5 2020

```

代码中出现 `Connet to QEMU successfully` 信息，说明初始化工作完成，QEMU和NEMU处于相同状态。接下来需要进行逐条指令执行后的状态对比，实现这一功能的是 `difftest_step()` 函数（位于 `nemu/src/monitor/diff-test/diff-test.c` 中），它会在 `exec_wrapper()` 的最后被调用，在NEMU中执行完一条指令之后，就在 `difftest_step()` 中让QEMU执行相同的指令，然后读出QEMU中的寄存器。在实验指导中给出的要求是，需要将NEMU的8个通用寄存器和eip与从QEMU中读出的寄存器的值进行比较，如果发现值不同，就输出相应的提示信息，并将diff标志设置为true，在 `difftest_step()` 的最后，如果检测到diff标志为true，就停止客户程序的运行。

因此，在 `nemu/src/monitor/diff-test/diff-test.c` 中实现 `difftest_step()` 代码如下。

```

150 // TODO: Check the registers state with QEMU.
151 // Set `diff` as `true` if they are not the same.
152 // TODO();
153 if(r.eax != cpu.eax) diff = true;
154 if(r.ecx != cpu.ecx) diff = true;
155 if(r.edx != cpu.edx) diff = true;
156 if(r.ebx != cpu.ebx) diff = true;
157 if(r.esp != cpu.esp) diff = true;
158 if(rebp != cpu.ebp) diff = true;
159 if(r.esi != cpu.esi) diff = true;
160 if(r.edi != cpu.edi) diff = true;
161
162 if (diff) {
163     //nemu_state = NEMU_END;
164     Log("different in general registers: when nemu.eip=0x%x", cpu.eip);
165 }
166 if(r.eip != cpu.eip){
167     diff = true;
168     Log("different: qemu.eip=0x%x, and nemu.eip=0x%x", r.eip, cpu.eip);
169 }
170

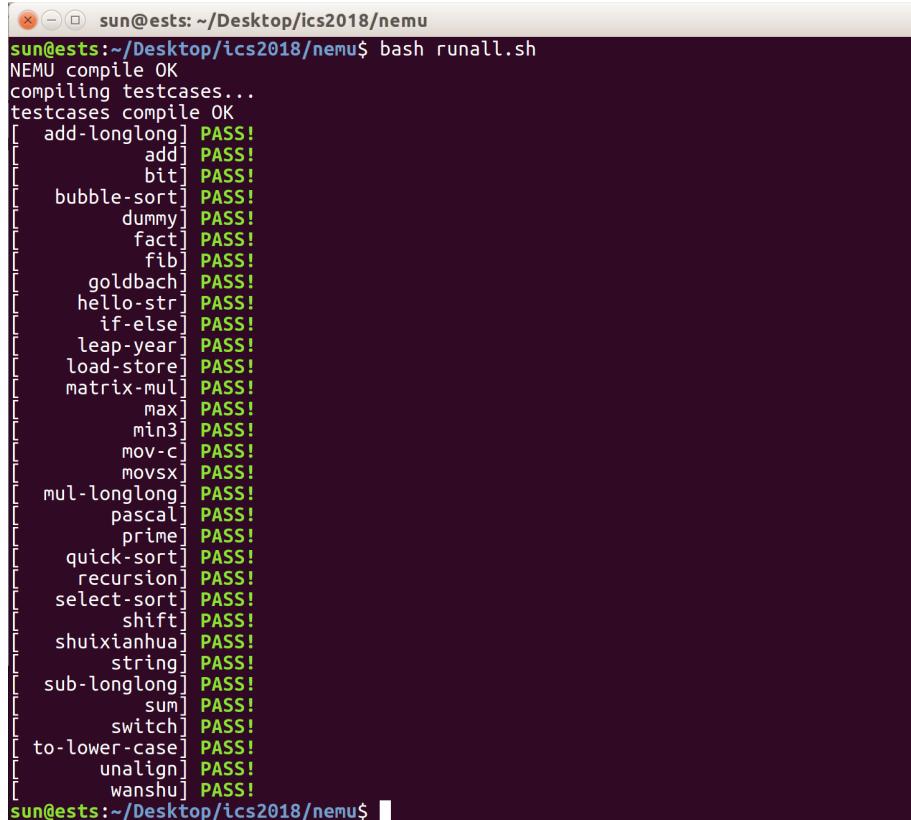
```

再次编译NEMU，执行结果如下。

```
sun@ests:~/Desktop/ics2018/nemu$ make run
+ CC src/monitor/diff-test/diff-test.c
+ LD build/nemu
./build/nemu -l ./build/nemu-log.txt
[src/monitor/diff-test/diff-test.c,96,init_difftest] Connect to QEMU successfully
[src/monitor/monitor.c,47,load_default_img] No image is given. Use the default built-in image.
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 10:25:27, May 5 2020
For help, type "help"
(nemu) c
nemu: HIT GOOD TRAP at eip = 0x00100026
```

可以看到，出现 `Connet to QEMU successfully` 信息，成功运行。由于 different testing 测试会降低 NEMU 运行速度，因此这里先关闭此功能，在 `common.h` 中注释掉 `#define DIFF_TEST` 即可。

然后进行一键回归测试如下。执行代码时出现报错，发现有 8 个文件没有成功运行，经过排查，发现在 `cc.c` 文件中的代码出现错误，即一个 case 下未加 `break` 语句引起的报错，添加语句后再次执行代码，结果如下。



```
sun@ests: ~/Desktop/ics2018/nemu
sun@ests:~/Desktop/ics2018/nemu$ bash runall.sh
NEMU compile OK
compiling testcases...
testcases compile OK
[ add-longlong] PASS!
[ add] PASS!
[ bit] PASS!
[ bubble-sort] PASS!
[ dummy] PASS!
[ fact] PASS!
[ fib] PASS!
[ goldbach] PASS!
[ hello-str] PASS!
[ if-else] PASS!
[ leap-year] PASS!
[ load-store] PASS!
[ matrix-mul] PASS!
[ max] PASS!
[ min3] PASS!
[ mov-c] PASS!
[ movsx] PASS!
[ mul-longlong] PASS!
[ pascal] PASS!
[ prime] PASS!
[ quick-sort] PASS!
[ recursion] PASS!
[ select-sort] PASS!
[ shift] PASS!
[ shuixianhua] PASS!
[ string] PASS!
[ sub-longlong] PASS!
[ sum] PASS!
[ switch] PASS!
[ to-lower-case] PASS!
[ unalign] PASS!
[ wanshu] PASS!
```

可以看到一键回归测试已经完整通过。

以上是本次实验阶段二部分。

PA2-阶段3

4 代码：加入 IOE

4.1 串口

串口是最简单的输出设备，在 `nemu/src/device/serial.c` 中模拟了串口的功能，但其大部分功能被简化，只保留了数据寄存器和状态寄存器。根据实验指导，该部分的实验要求如下。

运行 Hello World

实现 `in, out` 指令，在它们的 `helper` 函数中分别调用 `pio_read()` 和 `pio_write()` 函数。由于 NEMU 中有一些设备的行为是我们自定义的，与 QEMU 中的标准设备的行为不完全一样（例如 NEMU 中的串口总是就绪的，但 QEMU 中的串口并不是这样），这导致在 NEMU 中执行 `in` 和 `out` 指令的结果与 QEMU 可能会存在不可调整的偏差。为了使得 `differential testing` 可以正常工作，我们在这两条指令中调用了相应的函数来设置 `is_skip_qemu` 标志，来跳过与 QEMU 的检查。

实现后，在 `nexus-am/am/arch/x86-nemu/src/trm.c` 中定义宏 `HAS_SERIAL`，然后在 `nexus-am/apps/hello` 目录下键入 `make run`，在 NEMU 中运行基于 AM 的 `hello` 程序。如果你的实现正确，你将会看到程序往终端输出了 10 行 `Hello World!`

需要注意的是，这个 `hello` 程序和我们在程序设计课上写的第一个 `hello` 程序所处的层次是不一样的：这个 `hello` 程序可以说是直接运行在裸机上，可以在 AM 的抽象下直接输出到设备（串口）；而我们在程序设计课上写的 `hello` 程序位于操作系统之上，不能直接操作设备，只能通过操作系统提供的服务进行输出，输出的数据要经过很多层抽象才能到达设备层。我们会在 PA3 中进一步体会操作系统的作用。

首先，查阅 i386 手册，手册中关于 `in` 和 `out` 指令基本结构的内容如下。

IN — Input from Port

Opcode	Instruction	Clocks	Description
E4 ib	IN AL,imm8	12,pm=6*/26**	Input byte from immediate port into AL
E5 ib	IN AX,imm8	12,pm=6*/26**	Input word from immediate port into AX
E5 ib	IN EAX,imm8	12,pm=6*/26**	Input dword from immediate port into EAX
EC	IN AL,DX	13,pm=7*/27**	Input byte from port DX into AL
ED	IN AX,DX	13,pm=7*/27**	Input word from port DX into AX
ED	IN EAX,DX	13,pm=7*/27**	Input dword from port DX into EAX

NOTES:

- * If CPL ≤ IOPL
 - ** If CPL > IOPL or if in virtual 8086 mode
-

OUT — Output to Port

Opcode	Instruction	Clocks	Description
E6 ib	OUT imm8,AL	10,pm=4*/24**	Output byte AL to immediate port number
E7 ib	OUT imm8,AX	10,pm=4*/24**	Output word AL to immediate port number
E7 ib	OUT imm8,EAX	10,pm=4*/24**	Output dword AL to immediate port number
EE	OUT DX,AL	11,pm=5*/25**	Output byte AL to port number in DX
EF	OUT DX,AX	11,pm=5*/25**	Output word AL to port number in DX
EF	OUT DX,EAX	11,pm=5*/25**	Output dword AL to port number in DX

NOTES:

- * If CPL ≤ IOPL
 - ** If CPL > IOPL or if in virtual 8086 mode
-

根据手册，首先，需要在 `nemu/src/cpu/exec/system.c` 中添加执行函数如下。

```

47 make_EHelper(in) {
48     // TODO();
49     rtl_li(&t0, pio_read(id_src->val, id_dest->width));
50     operand_write(id_dest, &t0);
51
52     print_asm_template2(in);
53
54 #ifdef DIFF_TEST
55     diff_test_skip_qemu();
56 #endif
57 }
58
59 make_EHelper(out) {
60     // TODO();
61     pio_write(id_dest->val, id_src->width, id_src->val);
62
63     print_asm_template2(out);
64
65 #ifdef DIFF_TEST
66     diff_test_skip_qemu();
67 #endif
68 }

```

接下来，在opcode表中补全in和out的opcode码如下。

```

132    /* 0xe4 */   IDEWX(in_I2a,in,1), IDEWX(in_I2a,in,1), IDEWX(out_a2I,out,1)
133 , IDEWX(out_a2I,out,1),
134 /* 0xe8 */   IDEX(J,call), IDEX(J,jmp), EMPTY, IDEWX(J,jmp,1),
135 /* 0xec */   IDEWX(in_dx2a,in,1), IDEX(in_dx2a,in), IDEWX(out_a2dx,out,1)
136 , IDEX(out_a2dx,out),

```

同时在 `nexus-am/am/arch/x86-nemu/src/trm.c` 中定义宏 `HAS_SERIAL` 如下。

```

4 // Define this macro after serial has been implemented
5 #define HAS_SERIAL
6
7 #define SERIAL_PORT 0x3f8

```

接下来运行代码，结果如下。

```

sun@ests:~/Desktop/ics2018/nexus-am/apps/Hello$ make run
Building hello [native]
+ CC hello.c
make[1]: Entering directory '/home/sun/Desktop/ics2018/nexus-am'
make[2]: Entering directory '/home/sun/Desktop/ics2018/nexus-am/am'
Building am [native]
+ CC arch/native/src/ioe.c
+ CC arch/native/src/trm.c
+ CC arch/native/src/gui.c
+ AR /home/sun/Desktop/ics2018/nexus-am/am/build/am-native.a
make[2]: Leaving directory '/home/sun/Desktop/ics2018/nexus-am'
make[1]: Leaving directory '/home/sun/Desktop/ics2018/nexus-am'
make[1]: Entering directory '/home/sun/Desktop/ics2018/nexus-am/libs/klib'
make[1]: *** 没有指定目标并且找不到 makefile。停止。
make[1]: Leaving directory '/home/sun/Desktop/ics2018/nexus-am/libs/klib'
/home/sun/Desktop/ics2018/nexus-am/Makefile.compile:86: recipe for target 'klib' failed
make: [klib] Error 2 (ignored)
Hello World!

```

可以看到代码成功运行。

4.2 时钟

实现 |OE

实现 `uptime()` 后，在 NEMU 中运行 `timetest` 程序（在 `nexus-am/tests/timetest` 目录下，编译和运行方式请参考上文，此后不再额外说明）。如果你的实现正确，你将会看到程序每隔 1 秒输出一句话。

ative 作为 AM

"native"是指操作系统默认的运行时环境,例如我们通过 `gcc hello.c` 编译程序时,就会编译到 GNU/Linux 提供的运行时环境。事实上, `native` 也可以看做一个简单的 AM, 目前只支持 TRM 和 IOE。但很快你就会看到, `native` 也已经可以支撑很多程序的运行了。

`_uptime()` 返回的是系统（x86-nemu）启动后经过的毫秒数，
`_ioe_init()` 中的 `boot_time` 计算的是系统启动到 IOE 启动时已经经过的毫秒数，故当前 timer 需要减去初始时间 `boot_time`。在 `nexus-am/am/arch/x86-nemu/src/ioe.c` 下添加 `_uptime()` 函数如下。

```
11 unsigned long _uptime() {
12     unsigned long ms = inl(RTC_PORT) - boot_time;
13     // return 0;
14     return ms;
15 }
```

在 `nexus-am/tests/timetest` 目录下运行程序如下。可以看到程序每隔一秒输出一句话。

```
1 second.
2 seconds.
3 seconds.
4 seconds.
5 seconds.
6 seconds.
7 seconds.
8 seconds.
9 seconds.
10 seconds.
11 seconds.
12 seconds.
13 seconds.
14 seconds.
15 seconds.
16 seconds.
17 seconds.
18 seconds.
19 seconds.
20 seconds.
```

接下来就是要看NEMU跑多快了，在NEMU中依次运行以下benchmark，即 Dhrystone, Coremark, microbench。结果如下。

```
[sun@ests:~/Desktop/ics2018/nexus-am/apps/dhrystone$ make run
Building dhrystone [native]
+ CC dry.c
make[1]: Entering directory '/home/sun/Desktop/ics2018/nexus-am'
make[2]: Entering directory '/home/sun/Desktop/ics2018/nexus-am/am'
Building am [native]
make[2]: Nothing to be done for 'archive'.
make[2]: Leaving directory '/home/sun/Desktop/ics2018/nexus-am/am'
make[1]: Leaving directory '/home/sun/Desktop/ics2018/nexus-am'
make[1]: Entering directory '/home/sun/Desktop/ics2018/nexus-am/libs/klib'
make[1]: *** 没有指明目标并且找不到 makefile。 停止。
make[1]: Leaving directory '/home/sun/Desktop/ics2018/nexus-am/libs/klib'
/home/sun/Desktop/ics2018/nexus-am/Makefile.compile:86: recipe for target 'klib'
    failed
make: [klib] Error 2 (ignored)
Dhrystone Benchmark, Version C, Version 2.2
Trying 500000 runs through Dhrystone.
Finished in 21 ms
=====
Dhrystone PASS      49060 Marks
                  vs. 100000 Marks (i7-6700 @ 3.40GHz)
```

```
Running CoreMark for 1000 iterations
2K performance run parameters for coremark.
CoreMark Size      : 666
Total time (ms)   : 85
Iterations        : 1000
Compiler version  : GCC5.4.0 20160609
seedcrc           : 0xe9f5
[0]crclist        : 0xe714
[0]crcmatrix      : 0x1fd7
[0]crcstate       : 0x8e3a
[0]crcfinal       : 0xd340
Finised in 85 ms.
=====
CoreMark PASS      52571 Marks
                  vs. 100000 Marks (i7-6700 @ 3.40GHz)
```

```
=====
MicroBench PASS    51239 Marks
                  vs. 100000 Marks (i7-6700 @ 3.40GHz)
Exit (0)
```

4.3 键盘

如何检测多个键同时被按下

在游戏中，很多时候需要判断玩家是否同时按下了多个键，例如 RPG 游戏中的八方向行走，格斗游戏中的组合招式等等。根据键盘码的特性，你知道这些功能是如何实现的吗？

实现 IOE (2)

实现 `_read_key()` 后，在 NEMU 中运行 `keytest` 程序（在 `nexus-am/tests/keytest` 目录下）。如果你的实现正确，在程序运行时弹出的新窗口中按下按键，你将会看到程序输出相应的按键信息。

根据实验指导，键盘是最基本的输入设备，一般键盘的工作方式是，当按下一个键的时候，键盘将会发送该键的通码(make code)；当释放一个键的时候，键盘将会发送该键的断码(break code)。`nemu/src/device/keyboard.c` 模拟 i8042 通用设备接口芯片的功能。其大部分功能也被简化，只保留了键盘接口。i8042 初始化时会注册 0x60 处的端口作为数据寄存器，注册 0x64 处的端口作为状态寄存器。每当用户敲下/释放按键时，将会把相应的键盘码放入数据寄存器，同时把状态寄存器的标志设置为 1，表示有按键事件发生。CPU 可以通过端口 I/O 访问这些寄存器，获得键盘码。在 AM 中，约定通码的值为断码 +0x8000。

在 `nexus-am/am/arch/x86-nemu/src/ioe.c` 中实现 `_read_key()` 函数如下。

```
36 int _read_key() {
37     if(inb(0x64))
38         return inl(0x60);
39     else
40         return _KEY_NONE;
41 }
```

运行结果如下。

```

Get key: 44 S down
Get key: 44 S up
Get key: 45 D down
Get key: 45 D up
Get key: 47 G down
Get key: 47 G up
Get key: 70 SPACE down
Get key: 70 SPACE up
Get key: 54 RETURN down
Get key: 54 RETURN up
Get key: 1 ESCAPE down
Get key: 1 ESCAPE up
Get key: 3 F2 down
Get key: 3 F2 up
Get key: 11 F10 down
Get key: 11 F10 up
Get key: 27 BACKSPACE down
Get key: 27 BACKSPACE up
Get key: 28 TAB down
Get key: 28 TAB up
Get key: 46 F down
Get key: 46 F up
Get key: 22 8 down
Get key: 22 8 up

```

4.4 VGA

添加内存映射 I/O

在 `paddr_read()` 和 `paddr_write()` 中加入对内存映射 I/O 的判断。通过 `is_mmio()` 函数判断一个物理地址是否被映射到 I/O 空间，如果是，`is_mmio()` 会返回映射号，否则返回 -1。内存映射 I/O 的访问需要调用 `mmio_read()` 或 `mmio_write()`，调用时需要提供映射号。如果不是内存映射 I/O 的访问，就访问 `pmem`。

实现后，在 NEMU 中运行 `videotest` 程序（在 `nexus-am/tests/videotest` 目录下）。如果内存映射 I/O 实现正确，你会看到新窗口中输出了一些颜色信息。

实现 IOE(3)

事实上，刚才输出的颜色信息并不是 `videotest` 输出的画面，这是因为框架代码中的 `_draw_rect()` 并未正确实现其功能。你需要实现正确的 `_draw_rect()`。实现后，在 NEMU 中重新运行 `videotest`。如果你的实现正确，你将会看到新窗口中输出了相应的动画效果。

运行打字小游戏

根据实验指导，VGA 可以用于显示颜色像素，是最常用的输出设备。原理部分这里不再赘述，直接展示实现代码如下（`nexus-am/am/arch/x86-nemu/src/ioe.c`）

```

13 /* Memory accessing interfaces */
14
15 uint32_t paddr_read(paddr_t addr, int len) {
16     int r = is_mmio(addr);
17     if(r == -1)
18         return pmem_rw(addr, uint32_t) & (~0u >> ((4 - len) << 3));
19     else
20         return mmio_read(addr, len, r);
21 }
22 //~为按位取反，u后缀为unsigned，即将32位的0按位取反得111...111后右移(4-len)
23 //<<3位
24 //len取1 2 3 4,得pmem_rw(addr, uint32_t)的最后8 16 24 32位
25 void paddr_write(paddr_t addr, int len, uint32_t data) {
26     int r = is_mmio(addr);
27     if(r == -1)
28         memcpy(guest_to_host(addr), &data, len);
29     else
30         mmio_write(addr, len, data, r)
31 }

```

这里注意要先声明头文件 `#include "device/mmio.h"`。

在 `nexus-am/tests/videotest` 目录下运行代码，结果如下。

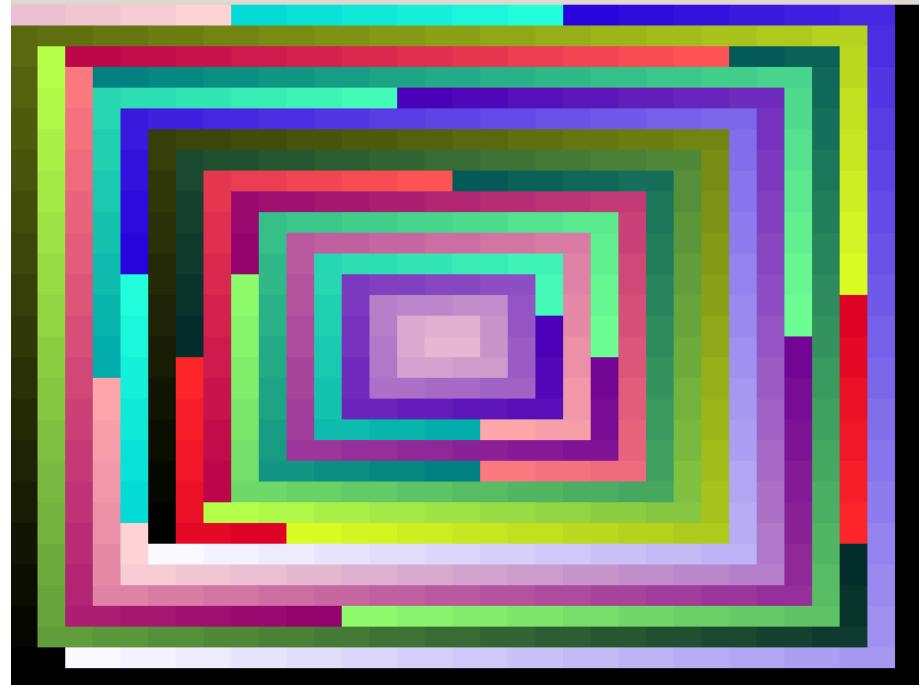
```
x - sun@ests: ~/Desktop/ics2018/nexus-am/tests/videotest
jet key: 28 TAB down
jet key: 28 TAB up
jet key: 46 F down
jet key: 46 F up
jet key: 22 8 down
jet key: 22 8 up
sun@ests:~/Desktop/i
sun@ests:~/Desktop/i
videotest coutest
sun@ests:~/Desktop/i
sun@ests:~/Desktop/i
building videotest [
+ CXX main.cpp
make[1]: Entering di
make[2]: Entering di
building am [native]
make[2]: Nothing to
make[2]: Leaving di
make[1]: Leaving di
make[1]: Entering di
make[1]: *** 没有指明目标并且找不到 makefile。 停止。
make[1]: Leaving directory '/home/sun/Desktop/ics2018/nexus-am/libs/klib'
/home/sun/Desktop/ics2018/nexus-am/Makefile.compile:86: recipe for target 'klib'
failed
make: [klib] Error 2 (ignored)
]

Native Application
videotest
am'
m'
libs/klib'
```

之后在 `ioe.c` 中实现 `_draw_rect()` 函数。

```
26 void _draw_rect(const uint32_t *pixels, int x, int y, int w, int h) {
27     /*
28     int i;
29     for (i = 0; i < _screen.width * _screen.height; i++) {
30         fb[i] = i;
31     */
32     /*
33     int temp = (w > _screen.width - x) ? _screen.width - x: w;
34     int cp_bytes = sizeof(uint32_t) * temp;
35     for(int j = 0; j < h && y + j < _screen.height; j ++){
36         memcpy(&fb[(y + j) * _screen.width + x], pixels, cp_bytes);
37         pixels += w;
38     }
39 }
```

再次执行 `videotest`，得到具有动画效果的实验结果。



在 `nexus-am/apps/typinig` 目录下执行 `make run`，可以运行打字小游戏如下。

```

sun@ests:~/Desktop/ics2018/nexus-am/apps/typing$ make run
Building typing [native]
+ CC game.c
+ CC draw.c
+ CC font.c
+ CC keyboard.c
make[1]: Entering direct
make[2]: Entering direct
Building am [native]
make[2]: Nothing to be d
make[2]: Leaving directo
make[1]: Leaving directo
make[1]: Entering direct
make[1]: *** 没有指明目标
make[1]: Leaving directo
/home/sun/Desktop/ics201
    failed
make: [klib] Error 2 (ig
[] sun@ests:~/Desktop/ics2

```

运行马里奥程序如下。

```

sun@ests:~/Desktop/ics2018/nexus-am/apps/litenes
make[2]: Entering directory '/home/sun/Desktop/ics2018/nexus-am/am'
Building am [native]
make[2]: Nothing to be d
make[2]: Leaving directo
make[1]: Leaving directo
make[1]: Entering direct
make[1]: *** 没有指明目标
make[1]: Leaving directo
/home/sun/Desktop/ics201
    failed
make: [klib] Error 2 (ig
sun@ests:~/Desktop/ics20
Building litenes [native
make[1]: Entering direct
make[2]: Entering direct
Building am [native]
make[2]: Nothing to be d
make[2]: Leaving directo
make[1]: Leaving directo
make[1]: Entering direct
make[1]: *** 没有指明目标并且找不到 makefile。停止。
make[1]: Leaving directory '/home/sun/Desktop/ics2018/nexus-am/libs/klib'
/home/sun/Desktop/ics2018/nexus-am/Makefile.compile:86: recipe for target 'klib'
    failed
make: [klib] Error 2 (ignored)
[]
```

关于`rol`指令实现代码如下。在`logic.c`中添加执行函数。

```

103 make_EHelper(rol){
104     rtl_shri(&t2, &id_dest->val, id_dest->width * 8 - id_src->width);
105     rtl_shl(&t3, &id_dest->val, &id_src->val);
106     rtl_or(&t1, &t2, &t3);
107     operand_write(id_dest, &t1);
108     print_asm_template2(rol);
109 }
```

在`exec.c`中添加opcode表。

```

47     /* 0xc0, 0xc1, 0xd0, 0xd1, 0xd2, 0xd3 */
48     make_group(gp2,
49             EX(rol), EMPTY, EMPTY, EMPTY,
50             EX(shl), EX(shr), EMPTY, EX(sar))
```

编译NEMU，结果如下。

```

sun@ests:~/Desktop/ics2018/nemu$ make run
./build/nemu -l ./build/nemu-log.txt
[src/monitor/monitor.c,47,load_default_img] No image is given. Use the default b
uild-in image.
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 10:50:07, May  5 2020
For help, type "help"
(nemu) c
nemu: HIT GOOD TRAP at eip = 0x00100026
```

再次执行一键回归测试确保程序正常运行。

```
sun@ests:~/Desktop/ics2018/nemu$ bash runall.sh
NEMU compile OK
compiling testcases...
testcases compile OK
[ add-longlong] PASS!
[ add] PASS!
[ bit] PASS!
[ bubble-sort] PASS!
[ dummy] PASS!
[ fact] PASS!
[ fib] PASS!
[ goldbach] PASS!
[ hello-str] PASS!
[ if-else] PASS!
[ leap-year] PASS!
[ load-store] PASS!
[ matrix-mul] PASS!
[ max] PASS!
[ min3] PASS!
[ mov-c] PASS!
[ movsx] PASS!
[ mul-longlong] PASS!
[ pascal] PASS!
[ prime] PASS!
[ quick-sort] PASS!
[ recursion] PASS!
[ select-sort] PASS!
[ shift] PASS!
[ shuixianhua] PASS!
[ string] PASS!
[ sub-longlong] PASS!
[ sum] PASS!
[ switch] PASS!
[ to-lower-case] PASS!
[ unalign] PASS!
[ wanshu] PASS!
```

5. 一个非常严重的问题

在实现完全部代码后，发现一个严重的问题，就是没有修改 `nexus-am/Makefile.check` 文件下 AM 程序默认编辑的目录。

于是修改文件 `nexus-am/Makefile.check` 如下。

```
1ifeq ($(MAKECMDGOALS),clean) # ignore check for make clean
2
3ifeq ($(AM_HOME),) # AM_HOME must exist
4$(error Environment variable AM_HOME must be defined.)
5endif
6
7ARCH ?= x86-nemu
8ARCHS = $(shell ls $(AM_HOME)/am/arch/)
9
10ifeq ($(filter $(ARCHS), $(ARCH)), ) # ARCH must be valid
11$(error Invalid ARCH. Supported: $(ARCHS))
12endif
13
14endif
```

再次回到 `nemu` 目录下进行一键回归测试，结果如下。

```
sun@ests:~/Desktop/ics2018/nemu$ bash runall.sh
NEMU compile OK
compiling testcases...
testcases compile OK
[ add-longlong] PASS!
[      add] PASS!
[      bit] PASS!
bubble-sort] PASS!
[ dummy] PASS!
[ fact] PASS!
[ fib] PASS!
goldbach] PASS!
hello-str] PASS!
if-else] PASS!
leap-year] PASS!
load-store] PASS!
matrix-mul]
[      max] PASS!
[      min3] PASS!
[ mov-c] PASS!
[ movsx] PASS!
mul-longlong] PASS!
[ pascal] PASS!
[ prime] PASS!
quick-sort] PASS!
recursion] PASS!
select-sort] PASS!
shift] PASS!
shuixianhua] PASS!
string] PASS!
sub-longlong] PASS!
[      sum] PASS!
switch] PASS!
to-lower-case] PASS!
unalign] PASS!
wanshu] PASS!
sun@ests:~/Desktop/ics2018/nemu$
```

然后我又重新运行了一遍IOE的各个测试，发现功能正常，由于截图比较多，这里就不放运行截图了。这也警醒自己以后写PA的时候一定要RTFM，还好这次没有出现太大的问题。

以上是本次实验阶段三部分。

PA2-必答题

必答题

你需要在实验报告中用自己的语言，尽可能详细地回答下列问题。

- ❖ 在 `nemu/include/cpu/rtl.h` 中，你会看到由 `static inline` 开头定义的各种 RTL 指令函数。选择其中一个函数，分别尝试去掉 `static`，去掉 `inline` 或者两者都去掉，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？
- ❖ 了解 `Makefile` 请描述你在 `nemu` 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 `nemu/build/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，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，也许会对你有帮助
 - ◆ RTFM

必答题1

首先，打开 `nemu/include/cpu/rtl.h` 文件。

```

sun@ests: ~/Desktop/ics2018/nemu/include/cpu
1 #ifndef __RTL_H__
2 #define __RTL_H__
3
4 #include "nemu.h"
5
6 extern rtlreg_t t0, t1, t2, t3;
7 extern const rtlreg_t tzero;
8
9 /* RTL basic instructions */
10
11 static inline void rtl_li(rtlreg_t* dest, uint32_t imm) {
12     *dest = imm;
13 }
14
15 #define c_add(a, b) ((a) + (b))
16 #define c_sub(a, b) ((a) - (b))
17 #define c_and(a, b) ((a) & (b))
18 #define c_or(a, b) ((a) | (b))
19 #define c_xor(a, b) ((a) ^ (b))
20 #define c_shl(a, b) ((a) << (b))
21 #define c_shr(a, b) ((a) >> (b))
22 #define c_sar(a, b) ((int32_t)(a) >> (b))
23 #define c_slt(a, b) ((int32_t)(a) < (int32_t)(b))
24 #define c_sltu(a, b) ((a) < (b))
25
26 #define make_rtl_arith_logic(name) \
27     static inline void concat(rtl_, name) (rtlreg_t* dest, const rtlreg_t* src \
28     1, const rtlreg_t* src2) { \
29         *dest = concat(c_, name) (*src1, *src2); \
30     } \
31     static inline void concat3(rtl_, name, i) (rtlreg_t* dest, const rtlreg_t* \
32     src1, int imm) { \
33         *dest = concat(c_, name) (*src1, imm); \
34     }
35

```

(1) 去掉 `static`

以 `rtl_li()` 为例，先去掉 `static`，执行 `nemu`，结果如下。

```

sun@ests:~/Desktop/ics2018/nemu$ make run
./build/nemu -l ./build/nemu-log.txt
[src/monitor/diff-test/diff-test.c,96,init_difftest] Connect to QEMU successfully
[src/monitor/monitor.c,47,load_default_img] No image is given. Use the default build-in image.
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 21:51:28, May 5 2020
For help, type "help"
(nemu) c
nemu: HIT GOOD TRAP at eip = 0x00100026
(nemu) q
qemu-system-i386: terminating on signal 15 from pid 18296

```

可以看到程序正常编译，这是因为在本实验环境和操作下执行 `inline` 内联。

(2) 去掉 `inline`

```

sun@ests:~/Desktop/ics2018/nemu$ make run
+ CC src/cpu/intr.c
In file included from ./include/cpu/decode.h:6:0,
                  from ./include/cpu/exec.h:9,
                  from src/cpu/intr.c:1:
./include/cpu/rtl.h:11:13: error: 'rtl_li' defined but not used [-Werror=unused-function]
    static void rtl_li(rtlreg_t* dest, uint32_t imm) {
                           ^
cc1: all warnings being treated as errors
Makefile:23: recipe for target 'build/obj/cpu/intr.o' failed
make: *** [build/obj/cpu/intr.o] Error 1

```

这个报错的发生是由于 `makefile` 文件中的 `-Wall -Werror` 所致，将其去掉会发现编译正常，`static inline` 定义的是内联函数，编译时直接将函数体嵌入每个调用处，与普通函数处理不同。

(3) 去掉 `static inline`

```
sun@est: ~/Desktop/ics2018/nemu
build/obj/cpu/intr.o:/home/sun/Desktop/ics2018/nemu./include/cpu/rtl.h:12: 第一
次在此定义
build/obj/cpu/exec/arith.o: 在函数‘rtl_li’中:
/home/sun/Desktop/ics2018/nemu./include/cpu/rtl.h:12: `rtl_li`被多次定义
build/obj/cpu/intr.o:/home/sun/Desktop/ics2018/nemu./include/cpu/rtl.h:12: 第一
次在此定义
build/obj/cpu/exec/exec.o: 在函数‘rtl_li’中:
/home/sun/Desktop/ics2018/nemu./include/cpu/rtl.h:12: `rtl_li`被多次定义
build/obj/cpu/intr.o:/home/sun/Desktop/ics2018/nemu./include/cpu/rtl.h:12: 第一
次在此定义
build/obj/cpu/exec/system.o: 在函数‘rtl_li’中:
/home/sun/Desktop/ics2018/nemu./include/cpu/rtl.h:12: `rtl_li`被多次定义
build/obj/cpu/intr.o:/home/sun/Desktop/ics2018/nemu./include/cpu/rtl.h:12: 第一
次在此定义
build/obj/cpu/exec/logic.o: 在函数‘rtl_li’中:
/home/sun/Desktop/ics2018/nemu./include/cpu/rtl.h:12: `rtl_li`被多次定义
build/obj/cpu/intr.o:/home/sun/Desktop/ics2018/nemu./include/cpu/rtl.h:12: 第一
次在此定义
build/obj/cpu/decode/decode.o: 在函数‘rtl_li’中:
/home/sun/Desktop/ics2018/nemu./include/cpu/rtl.h:12: `rtl_li`被多次定义
build/obj/cpu/intr.o:/home/sun/Desktop/ics2018/nemu./include/cpu/rtl.h:12: 第一
次在此定义
build/obj/cpu/decode/modrm.o: 在函数‘rtl_li’中:
/home/sun/Desktop/ics2018/nemu./include/cpu/rtl.h:12: `rtl_li`被多次定义
build/obj/cpu/intr.o:/home/sun/Desktop/ics2018/nemu./include/cpu/rtl.h:12: 第一
次在此定义
collect2: error: ld returned 1 exit status
Makefile:41: recipe for target 'build/nemu' failed
make: *** [build/nemu] Error 1
```

去掉 `static inline` 时会发现出现了重定义的问题。在 `rtl.h` 中的多个函数中符号表只包含了 `rtl_li`，因其不是内联函数。出现这种报错的原因是编译器在编译文件时，对于每个c或cpp文件，都包含了函数的声明和实现，即重定义，而在链接时，连接器会在所有的项目文件中寻找函数的实现方法，而此时函数在多个链接文件中出现，不清楚到底是链接了哪个同名函数，因此会出现错误。

必答题2

- ❖ 了解 `Makefile` 请描述你在 `nemu` 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 `nemu/build/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, 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, 也许会对你有帮助
 - ◊ `RTFM`

参考助教给的[博客](#)，当`make`命令执行时，需要一个`Makefile`文件，以告诉`make`命令需要如何去编译和链接程序。`Makefile`的规则如下。

```
target...:prerequisites...(预备知识, 先决条件)
```

```
command(指令)
```

```
...
```

```
...
```

```
-----
```

`target`是一个目标文件，可以是Object File，也可以是执行文件，还可以是一个标签（label）；`prerequisites`即要生成那个`target`所需要的文件或是目标；`command`就是`make`需要执行的命令（任意shell命令）。这是一个文件的依赖关系，`target`这一个或多个的目标文件依赖于`prerequisites`中的文件，其生成规则

定义在command中。即prerequisites中如果有一个以上的文件比target文件要新的话，command所定义的命令就会被执行。这就是Makefile中最核心的内容。

Makefile基本工作方式如下。

1. 读入主Makefile，主Makefile中可以引用其他Makefile
2. 读入被include的其他Makefile
3. 初始化文件中的变量
4. 推导隐晦规则，并分析所有规则
5. 为所有的目标文件创建依赖关系链
6. 根据依赖关系，决定哪些目标要重新生成
7. 执行生成命令

在文件nemu/Makefile中，进行逐步分析如下。

首先，makefile中定义了一些基本变量，这些是关于实现各文件的路径或名称声明。

```
1 NAME = nemu
2 INC_DIR += ./include
3 BUILD_DIR ?= ./build
4 OBJ_DIR ?= $(BUILD_DIR)/obj
5 BINARY ?= $(BUILD_DIR)/$(NAME)
```

下面代码定义了编译的源文件与中间目标文件。

```
17 # Files to be compiled
18 SRCS = $(shell find src/ -name "*.c")
19 OBJS = $(SRCS:src/%.c=$(OBJ_DIR)/%.o)
```

下面代码定义了高级变量，用以将所有.c文件编译生成对应的.o文件。

```
21 # Compilation patterns
22 $(OBJ_DIR)/%.o: src/%.c
23         @echo + CC $<
24         @mkdir -p $(dir $@)
25         @$< $(CFLAGS) -c -o $@ $<
```

此外，还定义了实现自动推导功能的隐含规则如下。

```
11 # Compilation flags
12 CC = gcc
13 LD = gcc
14 INCLUDES = $(addprefix -I, $(INC_DIR))
15 CFLAGS += -O2 -MMD -Wall -Werror -ggdb $(INCLUDES)
```

此文件中还包含了Makefile.git，用到了makefile包含文件的特性，实现了在Makefile中对git_commit(msg)函数的调用。

```
40 $(BINARY): $(OBJS)
41     $(call git_commit, "compile")
42     @echo + LD $@
43     @$< -O2 -o $@ $^ -lSDL2 -lreadline
```

因此，梳理整个Makefile执行的流程如下。

1. 读入主Makefile和被包含的其他Makefile文件；

2. 初始化文件中的变量，隐含变量，文件路径及名称等等；
3. 对所有隐晦规则的推导及分析，在 `nemu/Makefile` 中给出的一些规则为：`app` 依赖于 `BINARY` (`nemu` 可执行文件)，`BINARY` 依赖于 `OBJS` (`build` 下所有 `.o` 文件)，`build` 的 `.o` 文件依赖于 `src` 中的所有 `.c` 文件，各 `.c` 文件依赖于其中定义的 `.h` 文件；
4. 为目标文件创建关系链，生成相应的依赖文件，并根据依赖文件决定哪些目标要重新生成；
5. 最后是执行生成命令，将所有 `.c` 文件编译生成所有 `.o` 中间文件，再将各 `.o` 文件链接生成 `nemu` 可执行文件。

以上是本次实验必答题部分。

来个小小的总结吧

在本次实验中，自己遇到了很多问题。中间关于指令的部分很多代码都是参考助教的代码实现的，即便如此，自己还是有一些问题没有想的足够清楚，归结原因，终究是自己的能力有限，知识的欠缺所致。这也启发我在今后学习PA的路上需要加倍努力，拓展自己的知识范围，遇到困难勇敢地去钻研和解决。再次回顾本次PA，真的感觉受益匪浅。虽然自己实现过程中问题不断，参照了很多助教的代码，实验报告也交的很晚，但每一个模块，每一句话，每一行代码，也都是自己认真思考之后的结果，我希望在PA3中自己能够独立完成更多的代码，进一步提升自己的能力。感谢PA这门课，也感谢辛苦付出的卢老师和助教们。

以上是本次实验内容，感谢批阅！
