PA2

> 2013551 雷贺奥

PA2

- 实验目的
- 实验内容
- 实验过程
  - 阶段一
    - PUSH
    - POP
    - CALL
    - SUB
    - XOR
    - RET
    - 第一阶段结束
  - 阶段二
    - AND
    - AND
    - PUSH、PUSH
    - XCHG（NOP）
    - SETcc
    - MOVBX、MOVSX
    - JCC
    - SAR/SAL/SHL/SHR
    - TEST
    - CMP
    - JMP\JMP_RM
    - MUL
    - IMUL
    - DIV
    - IDIV
    - ADC
    - SBB
    - NEG
    - OR
    - NOT指令
    - DEC
    - INC
    - LEAVE
    - 阶段二结果
  - 阶段三
    - 串口
    - 时钟
    - 键盘
    - VGA
  - BUG总结
- 手册选答题
  - 问题一
  - 问题二
  - 问题三
  - 问题四
  - 问题五
  - 问题六

手册必答题
问题一
问题二
课堂问题
问题一
问题二

实验目的

①学习指令周期与指令执行过程，并简单实现现代指令系统

②学习运行时环境与 AM 的基本原理，深入理解 NEMU 的本质

③了解基础设施测试、调试的基本框架与思想

④学习 IO 设备的基本实现

⑤深入理解冯诺依曼计算机体系结构并尝试在 NEMU 中实现

实验内容

PA2 实验中主要涉及现代指令系统的实现，抽象机器 AM 的原理与应用，输入输 出设备三大部分。

第一阶段，了解指令周期与指令执行的原理，尝试编写简单的指令，在 nemu 中运行dummy程序。

第二阶段，在阶段一的基础上完善指令系统，学习 AM 的基本原理，更新调试、测试的基础设施。

最终使得`bash runall.sh`样例全部通过。

第三阶段，学习 IO 原理，简单实现 CPU 对输入输出设备的控制。对三个测试跑分文件进行测试`Dhrystone、Coremark、 microbench`，补全所有要求的指令，最终可以运行typing和litenes小游戏。

实验过程

阶段一

> 运行dummy文件，实现call、call_rm、push、pop、sub、xor、ret等指令，最终dummy文件成功运行。nemu: HIT GOOP TRAP at eip = 0x00100026

进入PA2后，在`ics2017/nexus-am/tests/cputest`下运行`make ARCH=x86-nemu ALL=dummy run`，可以得到未完成的部分。

```
[src/monitor/monitor.c,65,load_img] The image is /home/lha/icslha/nexus-am/tests/cputest/build/dummy-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 15:43:19, Apr  1 2023
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
 _ ____   ___    __    __  __                      _
(_)___ \ / _ \  / /   |  \/  |                    | |
 _  __) | (_) |/ /_   | \  / | __ _ _ __  _   _  __ _| |
| ||__ < > _ <| '_ \  | |\/| |/ _` | '_ \| | | |/ _` | |
| |___) | (_) | (_) | | |  | | (_| | | | | |_| | (_| | |
|_|____/ \___/ \___/  |_|  |_|\__,_|_| |_|\__,_|\__,_|_|

for more details.

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

（1）根据Log显示的opcode，定位到未实现的指令。

（2）阅读i386手册，编写对应的`rtl_XXX()`指令行为、`make_DHelper()`译码、`make_EHelper()`执行

（3）在`opcode_table`中填写。通过以上步骤，大抵可以实现一条指令。

PUSH

实现步骤：

- 阅读i386手册，了解指令

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
OF A8	PUSH GS	2	Push GS

- 然后分析指令的译码函数，在`decode.c`中的`make_DHelper(r)`函数，读取 decoding/opcode 中 标志的寄存器，将寄存器的内容方式 op->val 中。
- `Make_EHelper()`在`nemu/src/cpu/exec/all-instr.h`函数声明，这里阶段一的指令一起声明

```
make_EHelper(call);
make_EHelper(call_rm);
make_EHelper(push);
make_EHelper(pop);
make_EHelper(sub);
make_EHelper(xor);
make_EHelper(ret);
```

- `rtl_push`函数：修改栈顶，并将指针 src1 中的内容写入，**M[esp] <- src1**。

```
static inline void rtl_push(const rtlreg_t* src1) {
  // esp <- esp - 4
  // M[esp] <- src1
  //TODO();
  rtl_subi(&cpu.esp,&cpu.esp,4);     //减法 -4
  rtl_sm(&cpu.esp,4,src1);  //写入内存 uint_32
}
```

- `make_EHelper(push)` 函数，调用 `rtl_push` 函数写栈。 `nemu/src/cpu/exec/data-mov.c`

```
make_EHelper(push) {
    rtl_push(&id_dest->val);
    print_asm_template1(push);
}
```

- 填写对应的 `opcode_table`，+/r，表示将通用寄存器编号按数值加到 opcode 中，即0x50-0x57。

```
/* 0x50 */ IDEX(r, push), IDEX(r, push), IDEX(r, push), IDEX(r, push),
/* 0x54 */ IDEX(r, push), IDEX(r, push), IDEX(r, push), IDEX(r, push),
/* 0x68 */  IDEX(I,push), EMPTY, IDEXW(push_SI,push,1),IDEX(I_E2G,imul3),
```

POP

实验步骤：

- 阅读i386手册，了解指令，了解POP指令的格式和opcode: 0x58-0x5F

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

- 译码函数： `make_DHelper(r)` 函数，与push类型，都是使用同一个译码函数。
- 首先需要实现 rtl_pop 函数，从栈中读取数据并保存在dest中。

```
static inline void rtl_pop(rtlreg_t* dest) {
  // dest <- M[esp]
  // esp <- esp + 4
  //TODO();
  rtl_lm(dest,&cpu.esp,4);  //读内存->dest
  rtl_addi(&cpu.esp,&cpu.esp,4);
}
```

- 接着实现 `make_EHelper(pop)` 函数，将 `rtl_pop` 函数读取的数据写入到通用寄存器中。 `data-mov.c` 中。

```
make_EHelper(pop) {
  //TODO();
  rtl_pop(&t2); //临时寄存器保存值
  operand_write(id_dest,&t2);//使用operand_write执行写操作
  print_asm_template1(pop);
}
```

- 填写对应的 opcode_table，知道了该指令的opcode为 0x58-0x5F ， 但是不包含5c。

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

CALL

- 查看i386手册，了解指令的执行过程，指令的结构还有opcode为： 0xE8 ， 其后跟着一个4字节的操作数表示要跳转的地址与当前地址的偏移量。

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

- 译码函数 make_DHelper(J)调用make_DopHelper(SI)

  CPU 的跳转目标地址=当前 eip+立即数 offset（可正可负）, make_DHelper(J) 调用 decode_op_SI 函数实现立即数的读取，并更新 jmp_eip 。 `make_DopHelper(SI)` 与 `make_DopHelper(I)` 类似，的不同之处在于： 要求操作数宽度 width=1 或 4；读取的立即数需转为 signed immediate（有符号的立即数）。

```
static inline make_DopHelper(SI) {
  assert(op->width == 1 || op->width == 4);
  op->type = OP_TYPE_IMM;
  op->simm = instr_fetch(eip, op->width);
```

```
  if(op->width == 1)
  {
    op->simm = (int8_t)op->simm;
  }
  else
  {
    op->simm = (int32_t)op->simm;
  }
  rtl_li(&op->val, op->simm);

#ifdef DEBUG
  snprintf(op->str, OP_STR_SIZE, "$0x%x", op->simm);
#endif
}
```

- 执行函数，`make_EHelper(call)`

  在control.c中实现。通过阅读i386的description，call指令先将eip压栈再进行跳转，以便能够跳转回原函数，再将decoding.is_jmp=1。

```
make_EHelper(call) {
  //eip入栈
  rtl_li(&t2,decoding.seq_eip);
  rtl_push(&t2);
  //is_jump标志位
  decoding.is_jmp=1;
  print_asm("call %x", decoding.jmp_eip);
}
```

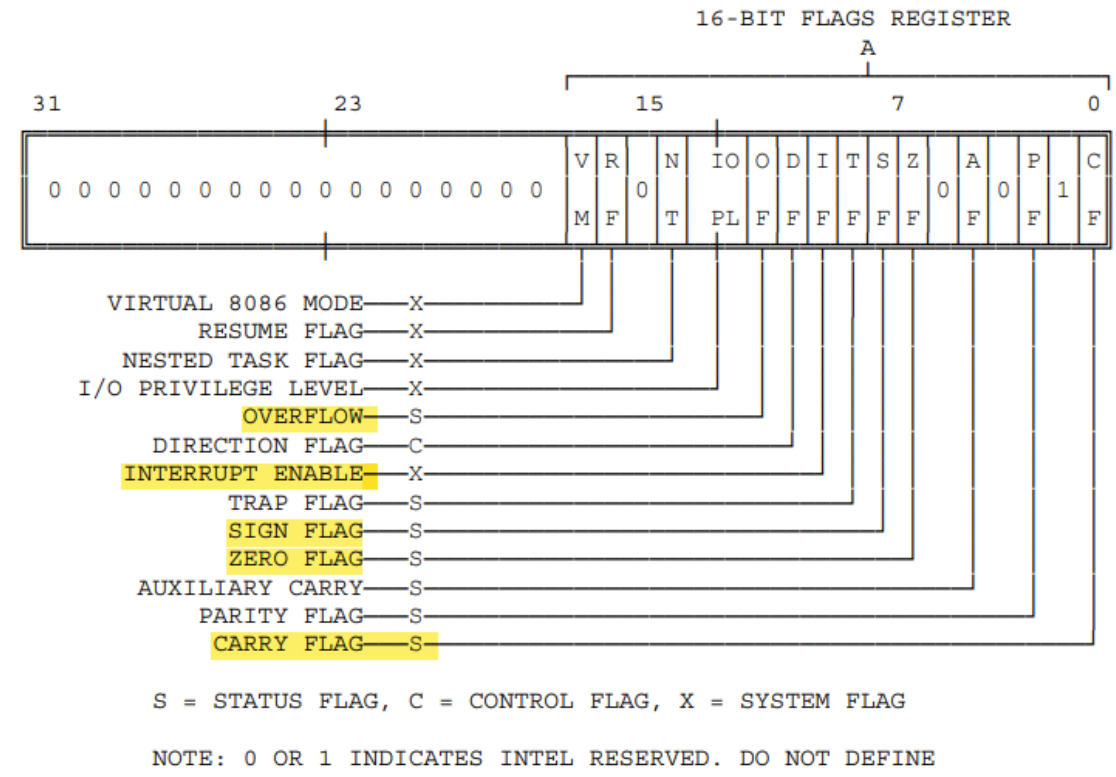
- 填写对应的 opcode_table ， opcode是 0xE8 。在 exec.c 中补充

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

SUB

实验步骤：

在实现该指令之前，需要先了解标志位的概念和功能。



- eflags 寄存器的定义,PA2中只需要CF、ZF、SF、IF、OF

  `nemu/include/cpu/reg.h`

```
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. */
    /* In NEMU, rtlreg_t is exactly uint32_t. This makes RTL instructions
     * in PA2 able to directly access these registers.
     */
     //eax~edi与gpr[8]共享内存
     struct{
       rtlreg_t eax, ecx, edx, ebx, esp, ebp, esi, edi;
     };
  };
  vaddr_t eip;
  //实现eflags寄存器
  struct bs{
    unsigned int CF:1;//[0]
    unsigned int one:1;
    unsigned int PF:1;//PF-PA2没用上
    unsigned int :1;
    unsigned int AF:1;//AF-PA2没用上
    unsigned int :1;
    unsigned int ZF:1;//[6]
    unsigned int SF:1;//[7]
    unsigned int :1;
    unsigned int IF:1;//[9]
    unsigned int :1;
```

```
    unsigned int OF:1;//[11]
    unsigned int :20;
  }eflags;
} CPU_state;
```

- eflags初始化，`nemu/src/monitor/monitor.c`，中修改`restart()`,使用memcpy将eflags设置为`0x0000 0002H`，以此来完成初始化

```
static inline void restart() {
  /* Set the initial instruction pointer. */
  cpu.eip = ENTRY_START;
  //进行eflags的初始化，0x0000 0002H
  unsigned int origin =2;
  memcpy(&cpu.eflags,&origin,sizeof(cpu.eflags));
#ifdef DIFF_TEST
  init_qemu_reg();
#endif
}
```

- 实现相关的 RTL 指令,rtl.h 中实现EFLAGS寄存器的标志位的读写函数。

```
#define make_rtl_setget_eflags(f) \
  static inline void concat(rtl_set_, f) (const rtlreg_t* src) { \
    /*TODO();*/ \
    cpu.eflags.f=*src;\
  } \
  static inline void concat(rtl_get_, f) (rtlreg_t* dest) { \
    /*TODO();*/ \
    *dest=cpu.eflags.f;\
  }
```

实现寄存器标志位的更新函数，`rtl_eq0、rtl_eqi、rtl_neq0、rtl_msb、rtl_update_ZF、rtl_update_SF`，根据TODO提示完成，如下所示：

```
static inline void rtl_eq0(rtlreg_t* dest, const rtlreg_t* src1) {
  // dest <- (src1 == 0 ? 1 : 0)#define c_sltu(a, b) ((a) < (b))
  //TODO();
  rtl_sltui(dest,src1,1);
}

static inline void rtl_eqi(rtlreg_t* dest, const rtlreg_t* src1, int imm) {
  // dest <- (src1 == imm ? 1 : 0)
  //TODO();
  //先异或再使用rtl_eq0
  rtl_xori(dest,src1,imm);
  rtl_eq0(dest,dest);
}

static inline void rtl_neq0(rtlreg_t* dest, const rtlreg_t* src1) {
  // dest <- (src1 != 0 ? 1 : 0)
  //TODO();
  //先等于，再自身取反
  rtl_eq0(dest,src1);
  rtl_eq0(dest,dest);
```

```
}

static inline void rtl_msb(rtlreg_t* dest, const rtlreg_t* src1, int width) {
  // dest <- src1[width * 8 - 1]
  //TODO();
  rtl_shri(dest,src1,width*8-1);//右移
  rtl_andi(dest, dest, 0x1);//保留最后一位
}

static inline void rtl_update_ZF(const rtlreg_t* result, int width) {
  // eflags.ZF <- is_zero(result[width * 8 - 1 .. 0])
  //TODO();
  rtl_andi(&t0, result, ( 0xffffffffu >> (4-width)*8 ) );
  rtl_eq0(&t0, &t0);
  rtl_set_ZF(&t0);
}

static inline void rtl_update_SF(const rtlreg_t* result, int width) {
  // eflags.SF <- is_sign(result[width * 8 - 1 .. 0])
  //TODO();
  rtl_msb(&t0, result, width);
  rtl_set_SF(&t0);
}

static inline void rtl_update_ZFSF(const rtlreg_t* result, int width) {
  rtl_update_ZF(result, width);
  rtl_update_SF(result, width);
}
```

- 查看i386手册，了解SUB指令的结构和字段含义

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

- 0x2D实现，make run时eip报错指令

  译码函数： make_DHelper(I2a) 。该函数调用 decode_op_a 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。

  执行函数：eflags_modify()： 计算减法并相应地设置 eflags 寄存器的值，即使用`rtl_sub`后，再更新eflags中ZF、SF、CF、OF的值，最后将结果值（存储在t2寄存器中）写回id_test。

```
static inline void eflags_modify(){
    rtl_sub(&t2, &id_dest->val, &id_src->val);
    rtl_update_ZFSF(&t2, id_dest->width);
    rtl_sltu(&t0, &id_dest->val, &id_src->val);
    rtl_set_CF(&t0);
    rtl_xor(&t0, &id_dest->val, &id_src->val);
    rtl_xor(&t1, &id_dest->val, &t2);
```

```
    rtl_and(&t0, &t0, &t1);
    rtl_msb(&t0, &t0, id_dest->width);
    rtl_set_OF(&t0);
}
make_EHelper(sub) {
  //TODO();
  eflags_modify();
  operand_write(id_dest,&t2);
  print_asm_template2(sub);
}
```

- 填写opcode_table，指令需要扩展。`exec.c`中。

```
/* 0x28 */ EMPTY, IDEX(G2E, sub), EMPTY, IDEX(E2G, sub),
```

0x80/81/83的sub需要进行opcode拓展,因为`opcode: 80 /5`将其填在gp1中的第六个位置，（之后不再赘述），opcode_table_gp1[5]。

```
/* 0x80, 0x81, 0x83 */
make_group(gp1,
    EX(add), EX(or), EX(adc), EX(sbb),
    EX(and), EX(sub), EX(xor), EX(cmp))
```

扩展后，执行函数处填写为gp1。

```
/* 0x80 */ IDEXW(I2E, gp1, 1), IDEX(I2E, gp1), EMPTY, IDEX(SI2E, gp1),
```

XOR

实验步骤：

- 查看i386手册，了解指令的结构，执行偶成，还需要更新eflags寄存器的标志位。

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 ib	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

得出，扩展指令时填写在gp1[6]的位置，再根据其描述可以得出；

DEST ← LeftSRC XOR RightSRC

CF ← 0

OF ← 0

- 执行函数，先使用`rtl_xor()`计算具体的值，并将其存储在t2寄存器中，随后再更新eflags中的标志位的值即可。

```
make_EHelper(xor) {
  //TODO();
  rtl_xor(&t2,&id_dest->val,&id_src->val);
  //写回寄存器
  operand_write(id_dest,&t2);
  //修改eflags，SF，ZF
  rtl_update_ZFSF(&t2,id_dest->width);
  //CF OF = 0
  rtl_set_CF(&tzero);
  rtl_set_OF(&tzero);
  print_asm_template2(xor);
}
```

- 填写 opcode_table ，根据得到的指令的opcode，填写对应的位置，修改 exec.c

  指令扩展，在gp1中填写xor，ext_opcode=6,则在 gp1[6]处填写 EX(xor)

```
make_group(gp1,
    EX(add), EX(or), EX(adc), EMPTY,
    EX(and), EX(sub), EX(xor), EX(cmp))
```

  根据得到的指令的opcode，填写对应的位置，修改 exec.c，

```
  /* 0x30 */    IDEXW(G2E,xor,1), IDEX(G2E,xor), IDEXW(E2G,xor,1),
IDEX(E2G,xor),
  /* 0x34 */    IDEXW(I2a,xor,1), IDEX(I2a,xor), EMPTY, EMPTY,

  /* 0x80 */    IDEXW(I2E, gp1, 1), IDEX(I2E, gp1), EMPTY, IDEX(SI2E, gp1),
```

RET

实验步骤：

- 查看i386手册，实现较为简单的 0xC3 ret （用栈的数据修改IP的内容，实现近转移）。

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

  得知，只需要使用EX（ret）即可，不需要解码函数。

- 执行函数，make_EHelper(ret)，修改control.c

```
make_EHelper(ret) {
  //TODO();
  rtl_pop(&t2);
  decoding.jmp_eip=t2;//用栈的数据修改EIP
  //is_jmp标志位-》1
  decoding.is_jmp=1;
  print_asm("ret");
}
```

- 填写opcode_table

根据opcode，在对应的位置填写代码，0xc3 处：EX(ret)

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

第一阶段结束

运行`make ARCH=x86-nemu ALL=dummy run`

```
lha@ubuntu:~/icslha/nexus-am/tests/cputest$ make ARCH=x86-nemu ALL=dummy run
Building dummy [x86-nemu]
Building am [x86-nemu]
make[2]: *** No targets specified and no makefile found.  Stop.
[src/monitor/monitor.c,65,load_img] The image is /home/lha/icslha/nexus-am/tests/cputest/build/dummy-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 11:28:28, Apr 18 2023
For help, type "help"
(nemu) c
nemu: HIT GOOD TRAP at eip = 0x00100052
```

阶段二

> 程序运行时环境与AM,完善更多指令。
>
> 使用diff_test debug
>
> 最终使得`bash runall.sh`样例全部通过。

按照文件分类，先将要实现的所有指令在`exec/all-instr.h`中声明各 make_EHelper()，再对各个Make_EHelper()进行填写。

```
#include "cpu/exec.h"
make_EHelper(mov);
make_EHelper(operand_size);
make_EHelper(inv);
make_EHelper(nemu_trap);
make_EHelper(ret);
//arith.c
make_EHelper(add);
make_EHelper(sub);
make_EHelper(cmp);
make_EHelper(inc);
make_EHelper(dec);
make_EHelper(neg);
make_EHelper(adc);
make_EHelper(sbb);
make_EHelper(mul);
make_EHelper(imul1);
make_EHelper(imul2);
make_EHelper(imul3);
make_EHelper(div);
make_EHelper(idiv);
//logic.c
make_EHelper(test);
make_EHelper(and);
make_EHelper(xor);
make_EHelper(or);
make_EHelper(sar);
make_EHelper(shl);
```

```
make_EHelper(shr);
make_EHelper(setcc);
make_EHelper(not);
make_EHelper(rol);
//data-mov.c
make_EHelper(mov);
make_EHelper(push);
make_EHelper(pop);
make_EHelper(pusha);
make_EHelper(popa);
make_EHelper(leave);
make_EHelper(cltd);
make_EHelper(cwtl);
make_EHelper(movsx);
make_EHelper(movzx);
make_EHelper(lea);
//special.c
make_EHelper(nop);
//control.c
make_EHelper(jmp);
make_EHelper(jcc);
make_EHelper(jmp_rm);
make_EHelper(call);
make_EHelper(call_rm);
//system.c
make_EHelper(in);
make_EHelper(out);
```

将ALL先设置为ALL=add，进行makerun，随后对每次eip的报错进行指令完善。这里我现实add测试集。

AND

实验步骤：

- 阅读i386指令，了解add指令

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

DEST ← DEST + SRC;

需要更新ZF、SF，如果进位CF=1，如果溢出OF=1

扩展指令时填写在gp1[0]。

- 执行函数

```
make_EHelper(add) {
  //TODO();
```

```
  rtl_add(&t2,&id_dest->val,&id_src->val);
  operand_write(id_dest,&t2);
  //update ZF SF
  rtl_update_ZFSF(&t2,id_dest->width);
  //进位 CF=1
  rtl_sltu(&t0,&t2,&id_dest->val);
  rtl_set_CF(&t0);
  //溢出 OF
  rtl_xor(&t0,&id_src->val,&t2);//t0 dest t2
  rtl_xor(&t1,&id_dest->val,&t2);//t1 dest t2
  rtl_and(&t0,&t0,&t1);
  rtl_msb(&t0,&t0,id_dest->width);//最高位
  rtl_set_OF(&t0);
  print_asm_template2(add);
}
```

指令扩展，扩展指令时填写在gp1[0]。

```
make_group(gp1,
EX(add), EX(or), EX(adc), EMPTY,
EX(and), EX(sub), EX(xor), EX(cmp))
```

- 填写opcode_table

```
  /* 0x00 */    IDEXW(G2E,add,1), IDEX(G2E,add), IDEXW(E2G,add,1),
IDEX(E2G,add),
  /* 0x04 */    IDEXW(I2a,add,1), IDEX(I2a,add), EMPTY, EMPTY,
```

AND

实验步骤：

- 阅读i386手册，了解指令。

```
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
```

DEST ← DEST AND SRC;

CF ← 0;

OF ← 0;

指令扩展时填写在gp1[4]处，opcodetable也要根据上述opcode填写。

- 执行函数

```
make_EHelper(and) {
  //TODO();
  rtl_and(&t2,&id_dest->val,&id_src->val);
  //写回寄存器
  operand_write(id_dest,&t2);
  //SF ZF
  rtl_update_ZFSF(&t2,id_dest->width);
  //CF OF =0
  rtl_set_CF(&tzero);
  rtl_set_OF(&tzero);
  print_asm_template2(and);
}
```

指令扩展

```
make_group(gp1,
EX(add), EX(or), EX(adc), EMPTY,
EX(and), EX(sub), EX(xor), EX(cmp))
```

- 填写对应opcode_table

```
/* 0x20 */ IDEXW(G2E, and, 1), IDEX(G2E, and), IDEXW(E2G, and, 1),
IDEX(E2G, and),
/* 0x24 */ IDEXW(I2a, and, 1), IDEX(I2a, and), EMPTY, EMPTY,
```

PUSH、PUSH

实验步骤：

- 阅读i386手册，了解指令。

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
OF A8	PUSH GS	2	Push GS

pushl相当于push dword，于此同时其opcode为0xFF，扩展指令时将其填写在gp5[6]。

- 执行函数已经给出

```
make_EHelper(push) {
  //TODO();
  rtl_push(&id_dest->val);
  print_asm_template1(push);
}
```

- 填写opcode_table

  指令扩展

```
make_group(gp5,
    EX(inc), EX(dec), EX(call_rm), EMPTY,
    EMPTY, EMPTY, EX(push), EMPTY)
```

  依据i386手册填写相应opcode对应的解码、执行函数。

```
  /* 0x50 */    IDEX(r,push), IDEX(r,push), IDEX(r,push), IDEX(r,push),
  /* 0x54 */    IDEX(r,push), IDEX(r,push), IDEX(r,push), IDEX(r,push),
  /* 0x68 */    IDEX(I,push), EMPTY, IDEXW(push_SI,push,1),
IDEX(I_E2G,imul3),
```

XCHG（NOP）

在编译运行add.c的程序的时候，在0x0010006a处停止，可以查看反汇编文件，当前的指令为xchg。这里注意到，i386 手册中 xchg 并没有 opcode=66 的这一选项。

```
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
```

根据反汇编代码可以知道，前缀 0x66，opcode=90 时，xchg %ax,%ax 实际上什么都没做，所以可以 直接使用nop 作为执行函数。查看nop 也发现，其opcode=90。

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

make_EHelper(xchg)不存在，所以暂时先不实现opcode=86、87的情况，事实证明，PA2也不需要实现这条指令。

SETcc

实验步骤：

- 阅读i386手册

```
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)
```

SETcc 指令（cc 为 condition code 的缩写），是指一系列形如 SETcc 的指令，如：SETNE、SETE、SETNA、SETA、SETNB、SETB 等等。通过eflags中各个标志位的值，对最终的目的操作数进行赋值（0或1）

0x0F两字节opcode，填写时需要在2byte opcode处。

PA2中只需要实现0F 94；0F 96; 0F 9F即可

- cc.c 中的 rtl_setcc 函数：

```
switch (subcode & 0xe) {
    case CC_O:
        rtl_get_OF(dest);
        break;
    case CC_B:
        rtl_get_CF(dest);
        break;
    case CC_E:
        rtl_get_ZF(dest);
        break;
    case CC_BE://cf==0||zf==0
        assert(dest!=&t0);
        rtl_get_CF(dest);
        rtl_get_ZF(&t0);
        rtl_or(dest,dest,&t0);
        break;
    case CC_S:
        rtl_get_SF(dest);
        break;
    case CC_L://c SF!=OF
        assert(dest!=&t0);
        rtl_get_SF(dest);
        rtl_get_OF(&t0);
```

```
        rtl_xor(dest,dest,&t0);
        break;
    case CC_LE://ZF==1||SF!=OF
        assert(dest!=&t0);
        rtl_get_SF(dest);
        rtl_get_OF(&t0);
        rtl_xor(dest,dest,&t0);
        //ZF
        rtl_get_ZF(&t0);
        rtl_or(dest,dest,&t0);
        break;
      //TODO();
    default: panic("should not reach here");
    case CC_P: panic("n86 does not have PF");
  }
```

- Make_EHelper(setcc)已经给出

```
make_EHelper(setcc) {
  uint8_t subcode = decoding.opcode & 0xf;
  rtl_setcc(&t2, subcode);
  operand_write(id_dest, &t2);
  print_asm("set%s %s", get_cc_name(subcode), id_dest->str);
}
```

- 填写opcode_table

  程序先在 0x0F 处执行 make_EHelper(2byte_esc)，在该函数中再确定其正确的两字节 opcode 编码。

```
/* 0xf6, 0xf7 */
make_group(gp3,
IDEX(test_I, test), EMPTY, EX(not), EMPTY,
EMPTY, EX(imul1), EMPTY, EMPTY)

/* 0x94 */ IDEXW(E, setcc, 1), IDEXW(E, setcc, 1), EMPTY, EMPTY,
/* 0x98 */ EMPTY, EMPTY, EMPTY, EMPTY,
/* 0x9c */ EMPTY, EMPTY, EMPTY, IDEXW(E, setcc, 1),
```

MOVBX、MOVSX

为了实现ret指令需要先实现MOVBX、MOVSX这两条指令。

实验步骤：

- 阅读i386手册了解指令

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

DEST ← ZeroExtend(SRC);

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

DEST ← SignExtend(SRC);

- Make_EHelper()执行函数

```
make_EHelper(movsx) {
  id_dest->width = decoding.is_operand_size_16 ? 2 : 4;
  rtl_sext(&t2, &id_src->val, id_src->width);
  operand_write(id_dest, &t2);
  print_asm_template2(movsx);
}

make_EHelper(movzx) {
  id_dest->width = decoding.is_operand_size_16 ? 2 : 4;
  operand_write(id_dest, &id_src->val);
  print_asm_template2(movzx);
}
```

- 填写opcode_table

```
/* 0xb4 */ EMPTY, EMPTY, IDEXW(mov_E2G, movzx, 1), IDEXW(mov_E2G, movzx,
2),
/* 0xb8 */ EMPTY, EMPTY, EMPTY, EMPTY,
/* 0xbc */ EMPTY, EMPTY, IDEXW(mov_E2G, movsx, 1), IDEXW(mov_E2G, movsx,
2),
```

JCC

Jcc指令为了实现ret

实验步骤：

- 阅读i386手册，了解指令

  Jcc是条件跳转指令，根据前面的指令使得eflags的标志位改变，而执行跳转的语句。阅读i386手册后可以发现。PA2要实现的指令主要在0x70-0x7F、0xe3、以及2 byte_opcode_table的0x80-0x8F。

- Make_EHelper()执行函数

```
make_EHelper(jcc) {
  // the target address is calculated at the decode stage
  uint8_t subcode = decoding.opcode & 0xf;
  rtl_setcc(&t2, subcode);
  decoding.is_jmp = t2;
  print_asm("j%s %x", get_cc_name(subcode), decoding.jmp_eip);
}
```

- 填写opcode_table

```
/*1 byte_opcode_table */
/* 0x70 */ EMPTY, EMPTY, IDEXW(J, jcc, 1), IDEXW(J, jcc, 1),
/* 0x74 */ IDEXW(J, jcc, 1), IDEXW(J, jcc, 1), IDEXW(J, jcc, 1), IDEXW(J,
jcc, 1),
/* 0x78 */ IDEXW(J, jcc, 1), IDEXW(J, jcc, 1), IDEXW(J, jcc, 1), IDEXW(J,
jcc, 1),
/* 0x7c */ IDEXW(J, jcc, 1), IDEXW(J, jcc, 1), IDEXW(J, jcc, 1), IDEXW(J,
jcc, 1),
/* 0xe0 */ EMPTY, EMPTY, EMPTY, IDEXW(J, jcc, 1),

/*2 byte_opcode_table */
/* 0x80 */ IDEX(J, jcc), IDEX(J, jcc), IDEX(J, jcc), IDEX(J, jcc),
/* 0x84 */ IDEX(J, jcc), IDEX(J, jcc), IDEX(J, jcc), IDEX(J, jcc),
/* 0x88 */ IDEX(J, jcc), IDEX(J, jcc), EMPTY, EMPTY,
/* 0x8c */ IDEX(J, jcc), IDEX(J, jcc), IDEX(J, jcc), IDEX(J, jcc),
```

SAR/SAL/SHL/SHR

位运算指令，为了实现ret指令。

- 阅读i386手册

  SHL/SAL: 每位左移, 低位补 0，高位进 CF

  SHR : 每位右移, 低位进 CF, 高位补 0

  SAR: 每位右移, 低位进 CF, 高位保持（原数据的高位）

- Make_EHelper()执行函数

```
make_EHelper(sar) {
  //TODO();
  // unnecessary to update CF and OF in NEMU
  rtl_sext(&t2,&id_dest->val,id_dest->width);//扩展
  rtl_sar(&t2,&t2,&id_src->val);
  operand_write(id_dest,&t2);
  rtl_update_ZFSF(&t2,id_dest->width);
  print_asm_template2(sar);
}

make_EHelper(shl) {
  //TODO();
  // unnecessary to update CF and OF in NEMU
  rtl_shl(&t2,&id_dest->val,&id_src->val);
  operand_write(id_dest,&t2);
  rtl_update_ZFSF(&t2,id_dest->width);
  print_asm_template2(shl);
}

make_EHelper(shr) {
  //TODO();
  // unnecessary to update CF and OF in NEMU
  rtl_shr(&t2,&id_dest->val,&id_src->val);
  //写回寄存器
  operand_write(id_dest,&t2);
  //eflags SF ZF
  rtl_update_ZFSF(&t2,id_dest->width);
  print_asm_template2(shr);
}
```

- rtl_XXX()，指令操作。`rtl.h`

  mv：dest<-src1，可以使用addi，将src1加上立即数0，最后存储在目的操作数中。

  not：dest<- ~dest ，将dest与0xffffffff按位异或，即可得到dest每位取反的结果，最终回写到dest中即可

  sext：dest <- signext(src1[(width * 8 - 1) .. 0])，当width=4时，即直接使用mv进行赋值。当width=1||width=2时，先将每位左移，再将每位右移，（使用sar即src1的高位进行扩展）

```
static inline void rtl_mv(rtlreg_t* dest, const rtlreg_t *src1) {
  // dest <- src1,#define c_sltu(a, b) ((a) < (b))
  //TODO();
  rtl_addi(dest,src1,0);
}

static inline void rtl_not(rtlreg_t* dest) {
  // dest <- ~dest
  //TODO();
  rtl_xori(dest,dest,0xffffffff);
}

static inline void rtl_sext(rtlreg_t* dest, const rtlreg_t* src1, int width)
{
  // dest <- signext(src1[(width * 8 - 1) .. 0])
  //TODO();
  if(width==4)
  {
    rtl_mv(dest,src1);
  }
  else
  {
    rtl_shli(dest,src1,(4-width)*8);
    rtl_sari(dest,dest,(4-width)*8);//符号扩展，scr1的高位扩展
  }
}
```

- opcode_table填写

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

TEST

实验步骤：

- 阅读i386手册

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

DEST : = LeftSRC AND RightSRC;

CF ← 0;

OF ← 0;

运算结果本身不会保存，只是设置根据结果设置标志寄存器的值。

- Make_EHelper()执行函数

```
make_EHelper(test) {
  //TODO();
  //and,不将结果写回
  rtl_and(&t2,&id_dest->val,&id_src->val);
  //更新ZFSF
  rtl_update_ZFSF(&t2,id_dest->width);
  rtl_set_CF(&tzero);
  rtl_set_OF(&tzero);
  print_asm_template2(test);
}
```

- opcode_table填写

  扩展指令时，观察到scr2是imm立即数，所以在gp3[0]处填写IDEX(test_I,test)，而0x84，0x85时不需要扩展，只需要找到相应的解码函数即可`G2E`；0x A8、0xA9解码函数`I2a`，opcode_table的相应位置填写即可。

```
/* 0xf6, 0xf7 */
make_group(gp3,
IDEX(test_I, test), EMPTY, EX(not), EMPTY,
EMPTY, EX(imul1), EMPTY, EMPTY)

/* 0x84 */  IDEXW(G2E,test,1), IDEX(G2E,test), EMPTY, EMPTY,//不确定
/* 0xa8 */ IDEXW(I2a, test, 1), IDEX(I2a, test), EMPTY, EMPTY,
```

CMP

实验步骤：

- 阅读i386手册

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

LeftSRC - SignExtend(RightSRC);

CMP的操作与TEST类似，只是不进行结果的存储，只修改EFLAGS的标 志。

- Make_EHelper()执行函数

```
make_EHelper(cmp) {
// TODO();
    eflags_modify();
    print_asm_template2(cmp);
}
```

- opcode_table填写

  指令扩展，填写在gp1[7]，同时找到相应的解码函数即可，这里不再赘述。

```
/* 0x80, 0x81, 0x83 */
make_group(gp1,
    EX(add), EX(or), EX(adc), EX(sbb),
    EX(and), EX(sub), EX(xor), EX(cmp))

/* 0x38 */  IDEXW(G2E,cmp,1), IDEX(G2E,cmp), IDEXW(E2G,cmp,1), IDEX(E2G,cmp),
/* 0x3c */  IDEXW(I2a,cmp,1), IDEX(I2a,cmp), EMPTY, EMPTY,
```

JMP\JMP_RM

- 阅读i386代码

  PA2只用到了 0xEB 与 0xE9 ， 所以在这里就实现这里两个。

  make_Ehelper(jmp)： jmp_eip由偏移量计算

  make_Ehelper(jmp_rm)： jmp_eip由寄存器取值。

- Make_EHelper()执行函数

```
make_EHelper(jmp) {
  // the target address is calculated at the decode stage
  decoding.is_jmp = 1;
  print_asm("jmp %x", decoding.jmp_eip);
}

make_EHelper(jmp_rm) {
  decoding.jmp_eip = id_dest->val;
  decoding.is_jmp = 1;
  print_asm("jmp *%s", id_dest->str);
}
```

- opcode_table填写

```
make_group(gp5,
    EX(inc), EX(dec), EX(call_rm), EMPTY,
    EX(jmp_rm), EMPTY, EX(push), EMPTY)
```

MUL

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)

指令扩展填写在gp3[4]

```
/* 0xf6, 0xf7 */
make_group(gp3,
    IDEX(test_I, test), EMPTY, EX(not), EMPTY,
    EMPTY, EX(imul1), EMPTY, EMPTY)
```

IMUL

- 阅读i386手册

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
0F AF /r	IMUL r16,r/m16	9-22/12-25	word register ← word register * r/m word
0F 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

  执行函数根据操作数的数目分为 imul1，imul2，imul3。

  可以得出imul1指令扩展gp3[5]

- Make_EHelper()执行函数

```
make_EHelper(mul) {
  rtl_lr(&t0, R_EAX, id_dest->width);
  rtl_mul(&t0, &t1, &id_dest->val, &t0);

  switch (id_dest->width) {
    case 1:
      rtl_sr_w(R_AX, &t1);
      break;
    case 2:
      rtl_sr_w(R_AX, &t1);
      rtl_shri(&t1, &t1, 16);
      rtl_sr_w(R_DX, &t1);
```

```
      break;
    case 4:
      rtl_sr_l(R_EDX, &t0);
      rtl_sr_l(R_EAX, &t1);
      break;
    default: assert(0);
  }

  print_asm_template1(mul);
}

// imul with one operand
make_EHelper(imul1) {
  rtl_lr(&t0, R_EAX, id_dest->width);
  rtl_imul(&t0, &t1, &id_dest->val, &t0);

  switch (id_dest->width) {
    case 1:
      rtl_sr_w(R_AX, &t1);
      break;
    case 2:
      rtl_sr_w(R_AX, &t1);
      rtl_shri(&t1, &t1, 16);
      rtl_sr_w(R_DX, &t1);
      break;
    case 4:
      rtl_sr_l(R_EDX, &t0);
      rtl_sr_l(R_EAX, &t1);
      break;
    default: assert(0);
  }

  print_asm_template1(imul);
}
```

- opcode_table填写

```
/* 0xf6, 0xf7 */
make_group(gp3,
IDEX(test_I, test), EMPTY, EX(not), EMPTY,
EMPTY, EX(imul1), EMPTY, EMPTY)

// 1 byte opcode
/* 0x68 */ EMPTY, EMPTY, IDEXW(push_SI, push, 1), IDEX(I_E2G, imul3),

// 2 byte opcode
/* 0xac */ EMPTY, EMPTY, EMPTY, EMPTY,
```

DIV

与MUL类似，这里不再赘述

- 修改 `exec.c` 文件

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

IDIV

与IMUL类似，这里不再赘述

- 修改 `exec.c` 文件

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

ADC

- 阅读i386手册

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 ib	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

DEST ← DEST + SRC + CF;

adc 是带进位加法指令，它利用了 CF 位上记录的进位值。

- Make_EHelper()执行函数，源代码中已经给出

```
make_EHelper(adc) {
  rtl_add(&t2, &id_dest->val, &id_src->val);
  rtl_sltu(&t3, &t2, &id_dest->val);
  rtl_get_CF(&t1);
  rtl_add(&t2, &t2, &t1);
  operand_write(id_dest, &t2);

  rtl_update_ZFSF(&t2, id_dest->width);

  rtl_sltu(&t0, &t2, &id_dest->val);
  rtl_or(&t0, &t3, &t0);
  rtl_set_CF(&t0);

  rtl_xor(&t0, &id_dest->val, &id_src->val);
  rtl_not(&t0);
  rtl_xor(&t1, &id_dest->val, &t2);
  rtl_and(&t0, &t0, &t1);
  rtl_msb(&t0, &t0, id_dest->width);
  rtl_set_OF(&t0);
```

```
  print_asm_template2(adc);
}
```

- opcode_table填写

```
/* 0x80, 0x81, 0x83 */
make_group(gp1,
    EX(add), EX(or), EX(adc), EX(sbb),
    EX(and), EX(sub), EX(xor), EX(cmp))

/* 0x10 */ IDEXW(G2E, adc, 1), IDEX(G2E, adc), IDEXW(E2G, adc, 1),
IDEX(E2G, adc),
/* 0x14 */ IDEXW(I2a, adc, 1), IDEX(I2a, adc), EMPTY, EMPTY,
```

SBB

- 阅读i386手册

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

IF SRC is a byte and DEST is a word or dword

THEN DEST = DEST - (SignExtend(SRC) + CF)

ELSE DEST ← DEST - (SRC + CF);

- Make_EHelper()执行函数，源代码中已经给出

```
make_EHelper(sbb) {
  rtl_sub(&t2, &id_dest->val, &id_src->val);
  rtl_sltu(&t3, &id_dest->val, &t2);
  rtl_get_CF(&t1);
  rtl_sub(&t2, &t2, &t1);
  operand_write(id_dest, &t2);

  rtl_update_ZFSF(&t2, id_dest->width);

  rtl_sltu(&t0, &id_dest->val, &t2);
  rtl_or(&t0, &t3, &t0);
  rtl_set_CF(&t0);

  rtl_xor(&t0, &id_dest->val, &id_src->val);
  rtl_xor(&t1, &id_dest->val, &t2);
  rtl_and(&t0, &t0, &t1);
  rtl_msb(&t0, &t0, id_dest->width);
```

```
  rtl_set_OF(&t0);

  print_asm_template2(sbb);
}
```

- opcode_table填写

```
/* 0x80, 0x81, 0x83 */
make_group(gp1,
EX(add), EX(or), EX(adc), EMPTY,
EX(and), EX(sub), EX(xor), EX(cmp))

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

NEG

- 阅读i386指令

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

IF r/m = 0 THEN CF ← 0 ELSE CF ← 1; FI; r/m ← - r/m;

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

- Make_EHelper()执行函数

```
make_EHelper(neg) {
  //TODO();
  //用零减去操作数， 然后结果返回操作数。
  rtl_sub(&t2,&tzero,&id_dest->val);
  rtl_update_ZFSF(&t2,id_dest->width);
  //operand=0->CF=0
  rtl_neq0(&t0,&id_dest->val);
  rtl_set_CF(&t0);

  rtl_eqi(&t0,&id_dest->val,0x80000000);//溢出
  rtl_set_OF(&t0);
  operand_write(id_dest,&t2);
  print_asm_template1(neg);
}
```

- opcode_table填写

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

OR

- 阅读i386指令

```
0C  ib      OR AL,imm8          2       OR immediate byte to AL
0D  iw      OR AX,imm16         2       OR immediate word to AX
0D  id      OR EAX,imm32        2       OR immediate dword to EAX
80  /1 ib   OR r/m8,imm8        2/7     OR immediate byte to r/m byte
81  /1 iw   OR r/m16,imm16      2/7     OR immediate word to r/m word
81  /1 id   OR r/m32,imm32      2/7     OR immediate dword to r/m dword
83  /1 ib   OR r/m16,imm8       2/7     OR sign-extended immediate byte with r/m word
83  /1 ib   OR r/m32,imm8       2/7     OR sign-extended immediate byte with r/m dwor
08  /r      OR r/m8,r8          2/6     OR byte register to r/m byte
09  /r      OR r/m16,r16        2/6     OR word register to r/m word
09  /r      OR r/m32,r32        2/6     OR dword register to r/m dword
0A  /r      OR r8,r/m8          2/7     OR byte register to r/m byte
0B  /r      OR r16,r/m16        2/7     OR word register to r/m word
0B  /r      OR r32,r/m32        2/7     OR dword register to r/m dword
```

DEST ← DEST OR SRC;

CF ← 0;

OF ← 0;

- Make_EHelper()执行函数

```
make_EHelper(or) {
  //TODO();
  rtl_or(&t2,&id_dest->val,&id_src->val);
  operand_write(id_dest,&t2);
  //SF ZF
  rtl_update_ZFSF(&t2,id_dest->width);
  //CF OF = 0
  rtl_set_CF(&tzero);
  rtl_set_OF(&tzero);
  print_asm_template2(or);
}
```

- opcode_table填写

```
/* 0x80, 0x81, 0x83 */
make_group(gp1,
    EX(add), EX(or), EX(adc), EX(sbb),
    EX(and), EX(sub), EX(xor), EX(cmp))

/* 0x08 */ IDEXW(G2E, or, 1), IDEX(G2E, or), IDEXW(E2G, or, 1), IDEX(E2G,
or),
/* 0x0c */ IDEXW(I2a, or, 1), IDEX(I2a, or), EMPTY, EX(2byte_esc),
```

NOT指令

- 阅读i386手册

```
Opcode     Instruction    Clocks     Description

F6    /2   NOT r/m8         2/6       Reverse each bit of r/m byte
F7    /2   NOT r/m16        2/6       Reverse each bit of r/m word
F7    /2   NOT r/m32        2/6       Reverse each bit of r/m dword
```

r/m ← NOT r/m;

- Make_EHelper()执行函数

```
make_EHelper(not) {
  //TODO();
  rtl_not(&id_dest->val);
  operand_write(id_dest,&id_dest->val);
  print_asm_template1(not);
}
```

- opcode_table填写

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

DEC

- 阅读i386手册

DEC — Decrement by 1

Opcode	Instruction	Clocks	Description
FE /1	DEC r/m8	2/6	Decrement r/m byte by 1
FF /1	DEC r/m16	2/6	Decrement r/m word by 1
	DEC r/m32	2/6	Decrement r/m dword by 1
48+rw	DEC r16	2	Decrement word register by 1
48+rw	DEC r32	2	Decrement dword register by 1

DEST ← DEST - 1;

- Make_EHelper()执行函数

```
make_EHelper(dec) {
  //TODO();
  rtl_subi(&t2,&id_dest->val,1);
  operand_write(id_dest,&t2);
  //ZF SF
  rtl_update_ZFSF(&t2,id_dest->width);
  //OF
  rtl_eqi(&t0,&t2,0x7fffffff);
  rtl_set_OF(&t0);
  print_asm_template1(dec);
}
```

- opcode_table填写

```
  /* 0xfe */
make_group(gp4,
    EX(inc), EX(dec), EMPTY, EMPTY,
    EMPTY, EMPTY, EMPTY, EMPTY)

  /* 0xff */
  //push->pushl
make_group(gp5,
    EX(inc), EX(dec), EX(call_rm), EMPTY,
    EX(jmp_rm), EMPTY, EX(push), EMPTY)

/* 0x48 */ IDEX(r, dec), IDEX(r, dec), IDEX(r, dec), IDEX(r, dec),
/* 0x4c */ EMPTY, EMPTY, IDEX(r, dec), IDEX(r, dec),
```

INC

与DEC类似这里不再赘述

- Make_EHelper()执行函数

```
make_EHelper(inc) {
  //TODO();
  rtl_addi(&t2,&id_dest->val,1);
  operand_write(id_dest,&t2);
  rtl_update_ZFSF(&t2,id_dest->width);
  //OF
  rtl_eqi(&t0,&t2,0x80000000);
  rtl_set_OF(&t0);
  print_asm_template1(inc);
}
```

- opcode_table填写

```
/* 0xfe */
make_group(gp4,
EX(inc), EX(dec), EMPTY, EMPTY,
EMPTY, EMPTY, EMPTY, EMPTY)
/* 0xff */
make_group(gp5,
EX(inc), EX(dec), EX(call_rm), EMPTY,
EMPTY, EMPTY, EX(push), EMPTY)

/* 0x40 */ IDEX(r, inc), IDEX(r, inc), IDEX(r, inc), IDEX(r, inc),
/* 0x44 */ EMPTY, EMPTY, IDEX(r, inc), IDEX(r, inc),
```

CLTD

AT&T 汇编里的 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。

- Make_EHelper()执行函数

```
make_EHelper(cltd) {
  //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。
  if (decoding.is_operand_size_16) {
```

```
    //TODO();
    rtl_lr_w(&t0,R_AX);
    rtl_sext(&t0,&t0,2);//符号扩展
    rtl_sari(&t0,&t0,31);
    rtl_sr_w(R_DX,&t0);
  }
  else {
    //TODO();
    rtl_sari(&cpu.edx,&cpu.eax,31);
  }

  print_asm(decoding.is_operand_size_16 ? "cwtl" : "cltd");
}
```

- opcode_table填写

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

LEAVE

- 阅读i386手册

```
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;
```

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。

- Make_EHelper()执行函数

```
make_EHelper(leave) {
// TODO();
    rtl_mv(&cpu.esp, &cpu.ebp);
    rtl_pop(&cpu.ebp);
    print_asm("leave");
}
```

- opcode_table填写

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

CALL

- Make_EHelper()执行函数

```
make_EHelper(call_rm) {
  //TODO();
  rtl_li(&t2,decoding.seq_eip);
  rtl_push(&t2);
  decoding.jmp_eip=id_dest->val;
  decoding.is_jmp=1;
  print_asm("call *%s", id_dest->str);
}
```

- opcode_table填写

```
  /* 0xff */
  //push->pushl
make_group(gp5,
    EX(inc), EX(dec), EX(call_rm), EMPTY,
    EX(jmp_rm), EMPTY, EX(push), EMPTY)
```

diff_test

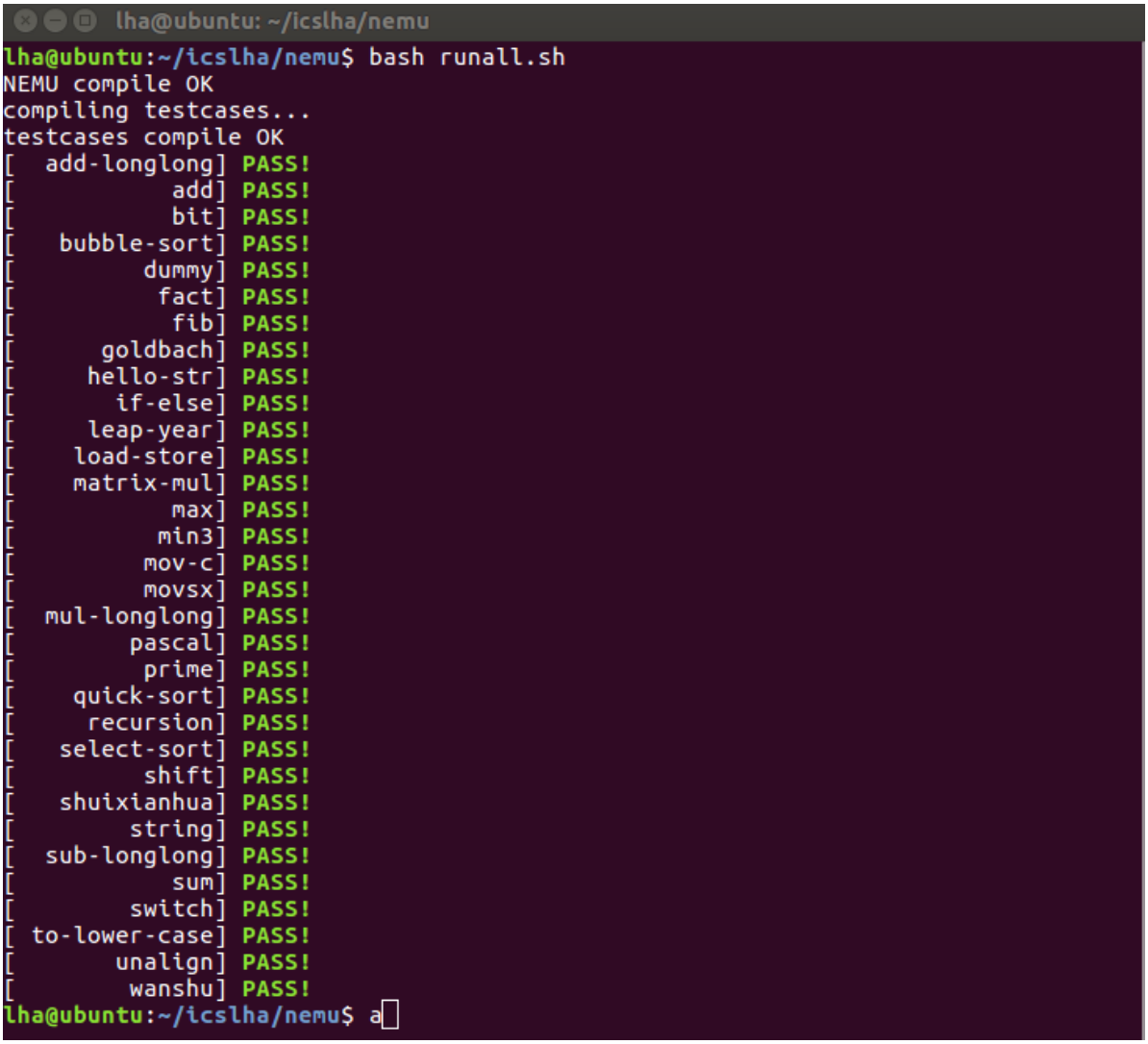
```
void difftest_step(uint32_t eip) {
  union gdb_regs r;
  bool diff = false;

  if (is_skip_nemu) {
    is_skip_nemu = false;
    return;
  }

  if (is_skip_qemu) {
    // to skip the checking of an instruction, just copy the reg state to qemu
    gdb_getregs(&r);
    regcpy_from_nemu(r);
    gdb_setregs(&r);
    is_skip_qemu = false;
    return;
  }

  gdb_si();
  gdb_getregs(&r);

  // TODO: Check the registers state with QEMU.
  // Set `diff` as `true` if they are not the same.
  //TODO();
  if(r.eax!=cpu.eax) {
      printf("eax expect: %d true: %d at: %x\n", r.eax, cpu.eax, cpu.eip);
      diff=true;
    }
    if(r.ecx!=cpu.ecx) {
      printf("ecx expect: %d true: %d at: %x \n", r.ecx, cpu.ecx, cpu.eip);
```

```
      diff=true;
    }
    if(r.edx!=cpu.edx) {
      printf("edx expect: %d true: %d at: %x\n", r.edx, cpu.edx, cpu.eip);
      diff=true;
    }
    if(r.ebx!=cpu.ebx) {
      printf("ebx expect: %d true: %d at: %x\n", r.ebx, cpu.ebx, cpu.eip);
      diff=true;
    }
    if(r.esp!=cpu.esp) {
      printf("esp expect: %d true: %d at: %x\n", r.esp, cpu.esp, cpu.eip);
      diff=true;
    }
    if(r.ebp!=cpu.ebp) {
      printf("ebp expect: %d true: %d at: %x\n", r.ebp, cpu.ebp, cpu.eip);
      diff=true;
    }
    if(r.esi!=cpu.esi) {
      printf("esi expect: %d true: %d at: %x\n", r.esi, cpu.esi, cpu.eip);
      diff=true;
    }
    if(r.edi!=cpu.edi) {
      printf("edi expect: %d true: %d at: %x\n", r.edi, cpu.edi, cpu.eip);
      diff=true;
    }
    if(r.eip!=cpu.eip) {
      diff=true;
      Log("eip different:qemu.eip=0x%x,nemu.eip=0x%x",r.eip,cpu.eip);
    }
    if (diff) {
      nemu_state = NEMU_END;
    }
}
```

我们让在 NEMU 中执行的每条指令也在真机中执行一 次,然后对比 NEMU 和真机的状态,如果 NEMU 和真机的状态不一致,我们就捕捉到 error 了。

阶段二结果

`bash runall.sh`



阶段三

串口

在此阶段继续补充指令，in、out

- 阅读i386手册

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

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

- Make_EHelper()执行函数

```
make_EHelper(in) {
  //TODO();
  rtl_li(&t0,pio_read(id_src->val,id_dest->width));
  operand_write(id_dest,&t0);
  print_asm_template2(in);

#ifdef DIFF_TEST
  diff_test_skip_qemu();
#endif
}

make_EHelper(out) {
  //TODO();
  pio_write(id_dest->val,id_src->width,id_src->val);
  print_asm_template2(out);

#ifdef DIFF_TEST
  diff_test_skip_qemu();
#endif
}
```

- opcode_table填写

```
/* 0xe4 */ IDEXW(in_I2a,in,1), IDEXW(in_I2a,in,1), IDEXW(out_a2I,out,1),
IDEXW(out_a2I,out,1),
/* 0xe8 */ IDEX(J,call), IDEX(J,jmp), EMPTY, IDEXW(J,jmp,1),
/* 0xec */ IDEXW(in_dx2a,in,1), IDEX(in_dx2a,in), IDEXW(out_a2dx,out,1),
IDEX(out_a2dx,out),
```

- 打开宏HAS_SERIAL
- 在hello下make run

```
[src/monitor/monitor.c,65,load_img] The image is /home/lha/icslha/nexus-am/apps/hello/build/hello-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 11:28:28, Apr 18 2023
For help, type "help"
(nemu) c
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
nemu: HIT GOOD TRAP at eip = 0x0010006e

(nemu)
```

时钟

添加了一个自定义的 RTC(Real Time Clock),初始化时将会注册 0x48 处的端口作为 RTC 寄存器,CPU 可以通 过 I/O 指令访问这一寄存器,获得当前时间(单位是 ms).

```
unsigned long _uptime() {
    unsigned long time_ms=inl(RTC_PORT)-boot_time;
    return time_ms;
  //return 0;
}
```

RTC_PORT绑定0x48获取时间，再用其减去bool_time得到运行时间。

- 在 NEMU 中运行 timetest 程序

```
[src/monitor/monitor.c,30,welcome] Build time: 11:28:28, Apr 18 2023
For help, type "help"
(nemu) c
1 second.
2 seconds.
3 seconds.
4 seconds.
5 seconds.
```

- 运行跑分程序，Dhrystone、Coremark、microbench，发现有d3指令没有实现，重新检查代码。
- 补充ROL指令
  - 阅读i386手册
  - Make_EHelper()执行函数

```
//Rotate 32 bits r/m dword left CL times
make_EHelper(rol)
{
    //（循环左移）指令：将操作数所有位都向左移//
    rtl_shri(&t2, &id_dest->val, id_dest->width * 8 - id_src->val);
    rtl_shl(&t3, &id_dest->val, &id_src->val);
    rtl_or(&t1, &t2, &t3);
    operand_write(id_dest, &t1);

    print_asm_template2(rol);
}
```

  - 补充opcode_table

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

- 重新跑分

```
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 11:28:28, Apr 18 2023
For help, type "help"
(nemu) c
Dhrystone Benchmark, Version C, Version 2.2
Trying 500000 runs through Dhrystone.
Finished in 7115 ms
------------------------------------------------
Dhrystone PASS          144 Marks
                    vs. 100000 Marks (i7-6700 @ 3.40GHz)
nemu: HIT GOOD TRAP at eip = 0x0010006e

(nemu)
```

```
Running CoreMark for 1000 iterations
2K performance run parameters for coremark.
CoreMark Size    : 666
Total time (ms)  : 8250
Iterations       : 1000
Compiler version : GCC5.4.0 20160609
seedcrc          : 0xe9f5
[0]crclist       : 0xe714
[0]crcmatrix     : 0x1fd7
[0]crcstate      : 0x8e3a
[0]crcfinal      : 0xd340
Finised in 8250 ms.
------------------------------------------------
CoreMark PASS        541 Marks
                 vs. 100000 Marks (i7-6700 @ 3.40GHz)
nemu: HIT GOOD TRAP at eip = 0x0010006e

(nemu)
```

```
------------------------------------------------------
MicroBench PASS          570 Marks
                     vs. 100000 Marks (i7-6700 @ 3.40GHz)
nemu: HIT GOOD TRAP at eip = 0x00100032
```

键盘

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。

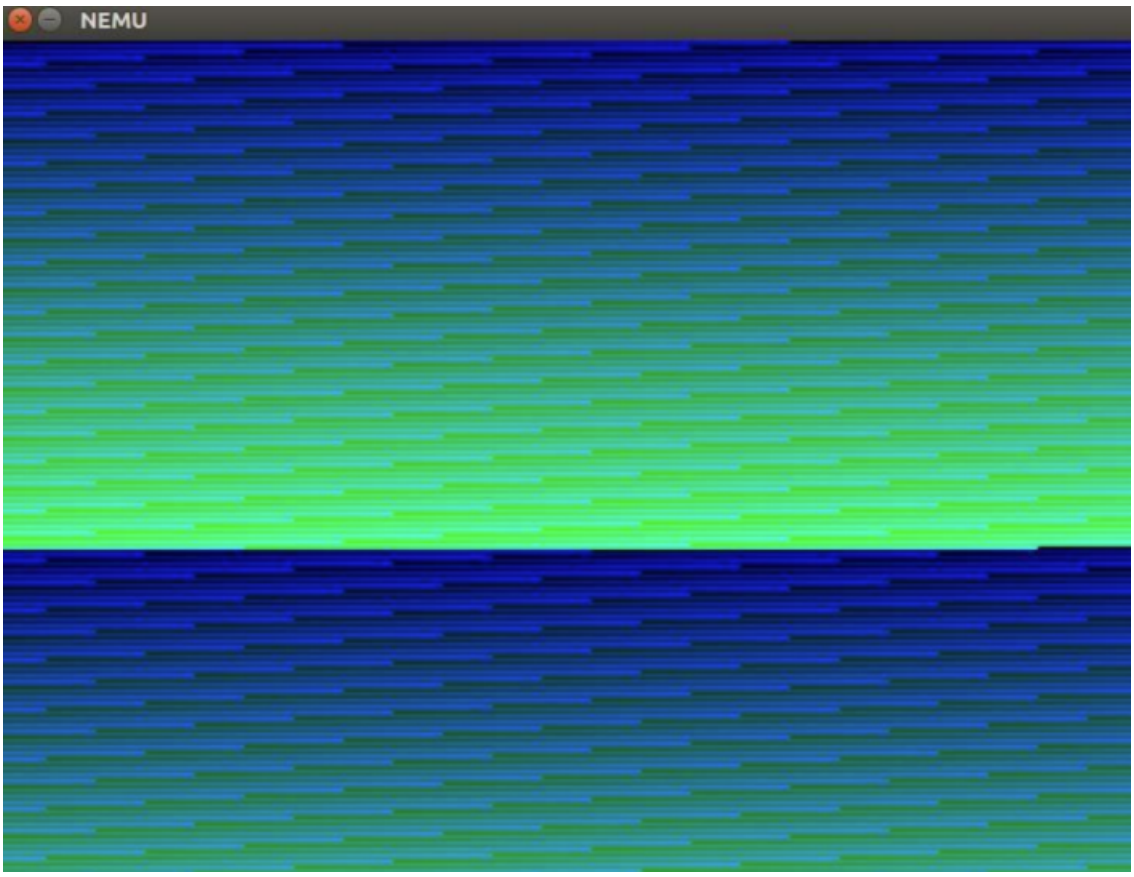
```
int _read_key() {
    if(inb(0x64))
        return inl(0x60);
    else
        return _KEY_NONE;
}
```

```
[src/monitor/monitor.c,65,load_img] The image is /home/lha/icslha/nexus-am/tests
/keytest/build/keytest-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 11:28:28, Apr 18 2023
For help, type "help"
(nemu) c
Get key: 51 L down
Get key: 51 L up
Get key: 48 H down
Get key: 48 H up
Get key: 43 A down
Get key: 43 A up
```

VGA

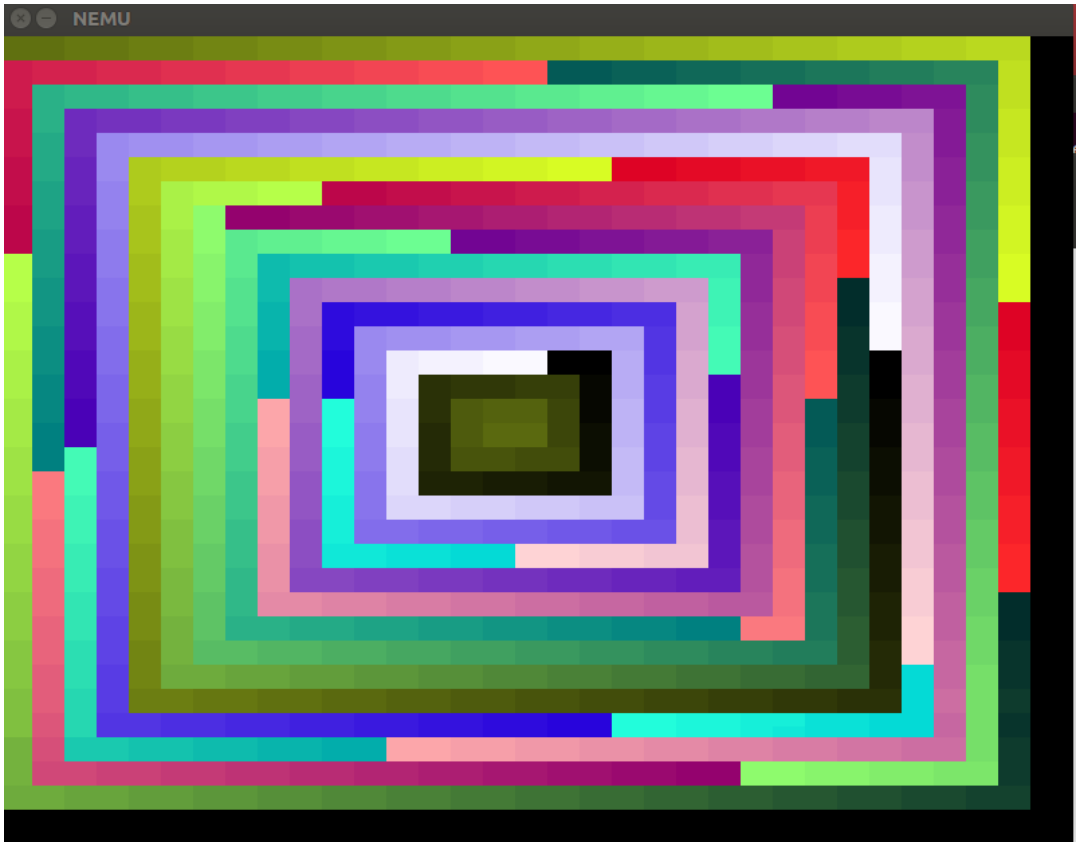
```
uint32_t paddr_read(paddr_t addr, int len) {
    if(is_mmio(addr)==-1)
    {
        return pmem_rw(addr, uint32_t) & (~0u >> ((4 - len) << 3));
    }
    else
    {
        return mmio_read(addr,len,is_mmio(addr));
    }
```

```
}

void paddr_write(paddr_t addr, int len, uint32_t data) {
  //memcpy(guest_to_host(addr), &data, len);
   if(is_mmio(addr)==-1)
   {
        memcpy(guest_to_host(addr), &data, len);
   }
   else
   {
        mmio_write(addr,len,data,is_mmio(addr));
   }
}
```

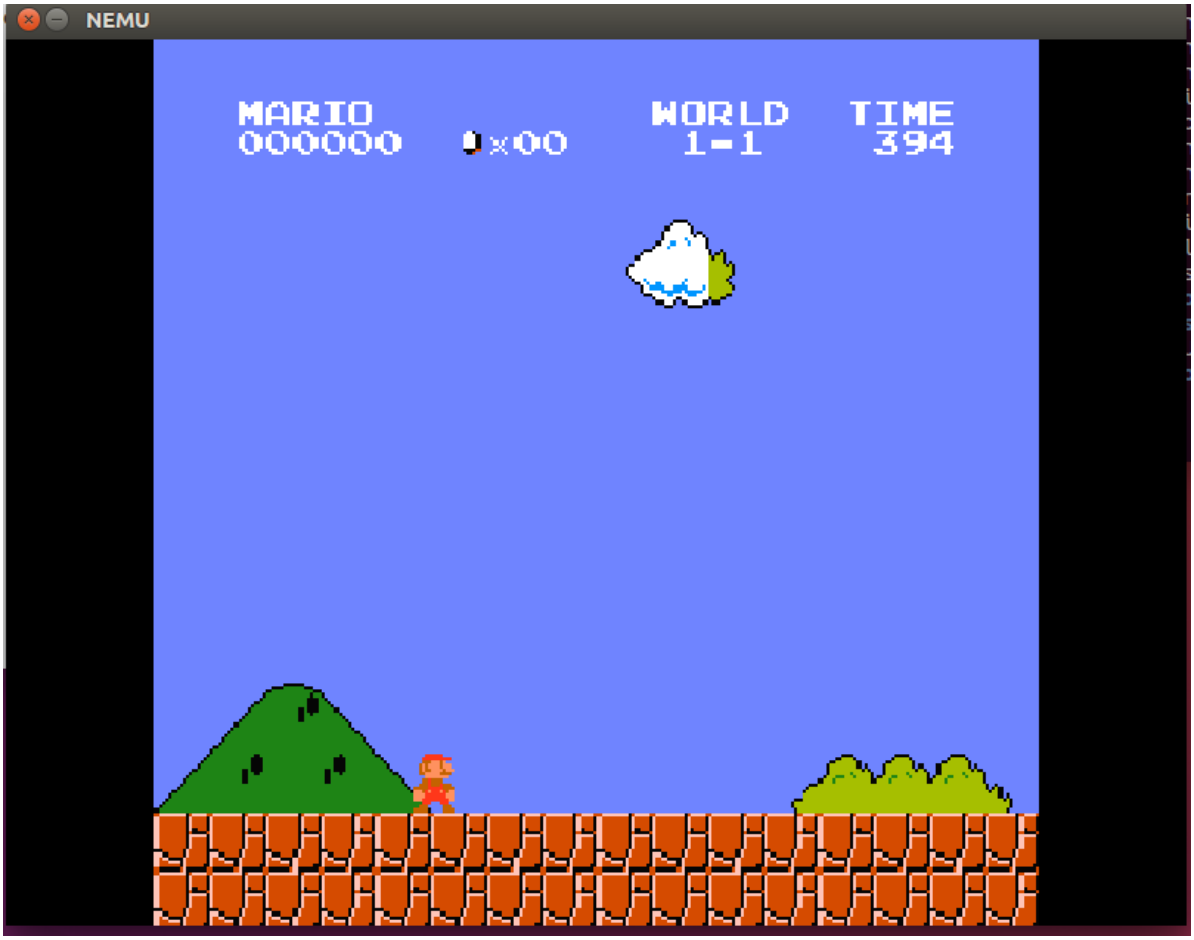


- 实现_draw_rect 函数，修改 nexus-am/am/arch/x86-nemu/src/ioe.c 文件

  可以直接参考native下的ioe.c文件中的draw_rect




```
void _draw_rect(const uint32_t *pixels, int x, int y, int w, int h) {
  //直接cv native
  int temp=0;
  if(w< _screen.width - x)
  {
    temp=w;
  }
  else
  {
    temp=_screen.width - x;
  }
  int cp_bytes = sizeof(uint32_t) * temp;
  for (int j = 0; j < h && y + j < _screen.height; j ++) {
    memcpy(&fb[(y + j) * _screen.width + x], pixels, cp_bytes);
    pixels += w;
  }
}
```



BUG总结

- 测试集add死循环

  cc.c中实现rtl_setcc()中忘记break，最终看到输出的should not reach here 和 Log输出的信息定位。Debug了好久。

- Rol指令问题

  在跑分时，eip执行到d3指令后报错，可是d3的opcode_table已经添上了IDEX(gp2_cl2E, gp2),所以进入汇编代码的txt文本，发现了此指令为d3 c0 `rol`,最后查阅i386手册，补上即可跑风成功。

- jle跳传错误

  并不是jle指令本身出错，而是上面几条的指令出错，从而影响到了EFLAGS寄存器中的标志位，通过打Log解决。

手册选答题

问题一

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

（1）假设我们需要将 NEMU 运行在 Motorola 68k 的机器上 (把 NEMU 的源代码编译成 Motorola 68k 的机器码)。

（2）假设我们需要编写一个新的模拟器 NEMU-Motorola-68k, 模拟器本身运行在 x86 架构 中, 但它模拟的是 Motorola 68k 程序的执行假设我们需要将 NEMU 运行在 Motorola 68k 的机 器上 (把 NEMU 的源代码编译成 Motorola 68k 的机器码)。 需要注意些什么问题? 为什么会产生这些问题? 怎么解决它们?

需要注意的问题是 Motorola68K 是大端运行，而NEMU为小端运行。即向内存输入小端数据0x1234

nemu：0x1234而Motorola68K 0x3412。

解决方式为实现一个栈，先将小端的内存地址从低到高两位两位划 分提取出来存入栈中，随后大端地址按低位从栈的顶部获取数据。

问题二

堆和栈在哪里？我们知道代码和数据都在可执行文件里面, 但却没有提到堆 (heap) 和栈 (stack). 为什么堆和栈的内容没有放入可执行文件里面? 那程序运行时刻用到的堆和栈又是怎么 来的?

堆和栈都在内存里，堆位于程序内存空间的低地址区域，由下向上生长；栈区位于程序 内存高地址区域，由上向下生长。

因为堆栈中的数据变化频繁，如果发在可执行文件中频繁申请注销开销巨大，程序运行时需要的堆栈都是动态申请的。

问题三

如果让你为 NEMU 添加如下功能：当用户程序陷入死循环时, 让用户程序暂停下来, 并 输出相应的提示信息你觉得应该如何实现?

通过查阅资料，发现此为图灵停机问题，应该是无解的。

问题四

使用-O2 编译代码. 尝试去掉代码中的 volatile 关键字, 重新使用-O2 编译, 并对比去掉 volatile 前后反汇编结果的不同. 你或许会感到疑惑, 代码优化不是一件好事情吗? 为什么会有 volatile 这种奇葩的存在? 思考一下, 如果代码中的地址 0x8049000 最终被映射到一个设备寄存 器, 去掉 volatile 可能会带来什么问题?

如果去掉 volatile，编译器会认为 while 循环及其后的代码不可达，即死代码删除优化，func()直接直接为 *p 赋值后返回。

但如果代码中的地址 0x8049000 最终被映射到一个设备寄存器，while 循环事实上是一个忙等待过程，因为在某个时候设备寄存器中的值可能为0xff，从而跳出循环，为*p赋值0x33、0x34、0x86。

这种情况下，若去掉 volatile，后续代码被当作为死代码进行删除，后续的 0x33、0x34、0x86 三个值将不会被写入端口。

问题五

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?

驱动检测按下多个键之间的时间差，如果小于某个特定的值，判定为同时按下。

问题六

在一些 90 年代的游戏中, 很多渐出渐入效果都是通过调色板实现的, 聪明的你知道其中 的玄机吗?

可以将调色板连续设置为由浅入深的同一颜色，随时间索引改变调色板即可实现渐入渐出的效果。

手册必答题

问题一

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 或去掉两者, 然后重新进行编译, 你会看到 发生错误. 请分别解释为什么会发生这些错误? 你有办法证明你的想法吗?

- 去掉static并不会报错

  程序正常运行，因为inline关键字让函数不在栈区内调用，函数执行时将inline的内容直接作为指令来运行，函数不会被视作为一个符号。因此，不会报重定义之类的错误。

- 去掉inline会报"defined but not used"

  static静态关键字能让符号存在静态存储区，函数只能在本文件中调用, 虽然可以和其他文件中同名符号的函数区分开，但是，本文件中并没用调用所有的rtl函数，即有rtl函数没有被调用，即会报错“defined but not used”

- 都去掉会报"multiple definition of..."

  不同文件中含有同名函数，在链接时，链接器在所有文件中寻找函数定义，但如果找到了多个同名函数，就会因为分不清而产生重定义错误。

问题二

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)

- 工作方式

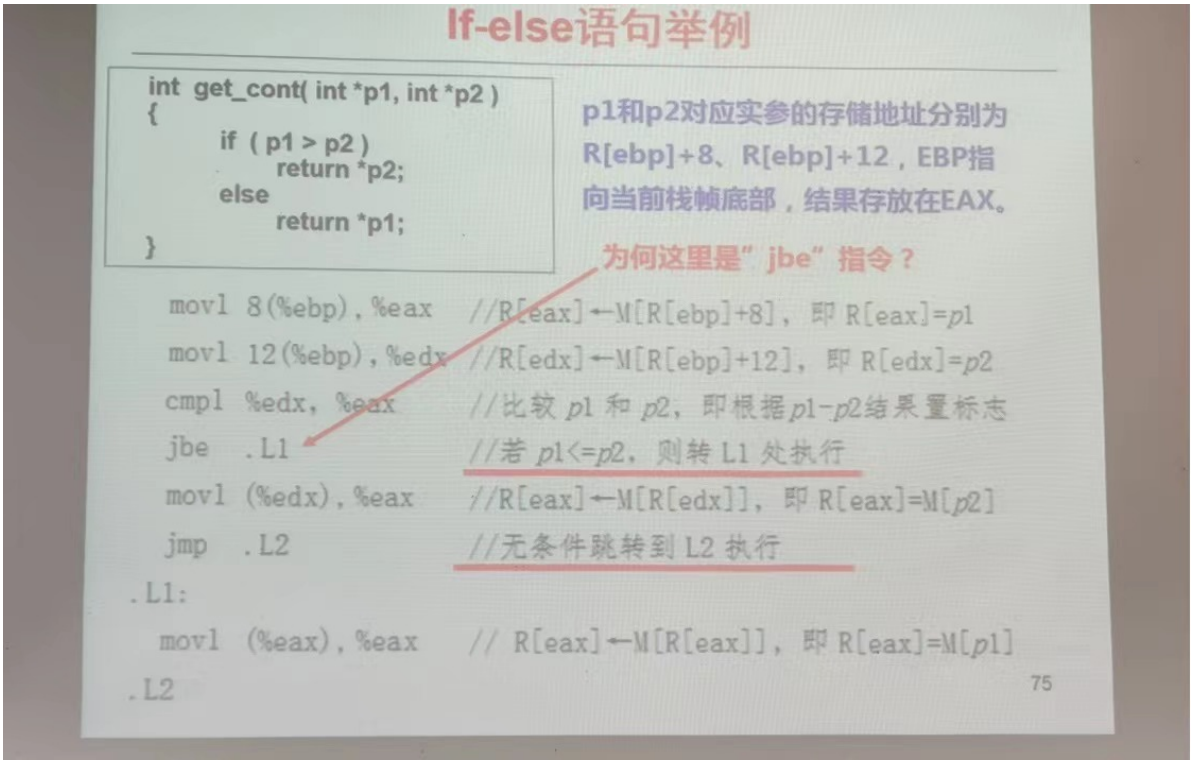
  > make 会在当前目录下找名字叫“Makefile”或“makefile”的文件，从当前 目录下的 Makefile 文件中开始，根据其中 include 的内容，读取上层次的 Makefile 文件。；初始化 Makefile 文件中的变量。；推导隐晦规则, 并为所有文件构建出一条依赖关系链；根据输入的标签，执行相应的指令。

- 编译链接过程

  > 首先为目标文件建立依赖关系；然后按该依赖顺序编译生成目标文件。
  >
  > 如果此工程没有被编译过，则所有c文件都要进行编译和链接工作。
  >
  > 如果此工程被编译过，只需要重新编译几个被修改的c文件，并重新链接目标程序。
  >
  > 如果头文件也修改了，则需要重新编译引用了这些.h文件的c文件，并且重新链接目标程序。

课堂问题

问题一



jbe为小于等于0时JMP的指令。

前一条指令`cmpl %edx,%eax`,将标志位设置为$p_1 - p_2$的结果，

即当$p_1 > p_2$时，进入if语句，jbe不跳转，$R[eax] = M[p_2]$,随后jmp 到.L2 退出此函数。

即当$p_1 <= p_2$时，进入else语句，jbe跳转至.L1,$R[eax] = M[p_1]$,随后执行到.L2，退出此函数

问题二

nemu 输出的 helloword 和程序中输出的 helloworld 有什么区别?

- nemu中的程序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