



南開大學  
Nankai University

计算机学院  
计算机系统设计实验报告

PA2-简单复杂的机器：冯诺依曼计算机  
系统

姓名：杨馨仪  
学号：2011440  
专业：计算机科学与技术

2024 年 4 月 15 日

# 目录

|                                      |           |
|--------------------------------------|-----------|
| <b>1 概述</b>                          | <b>2</b>  |
| 1.1 实验目的                             | 2         |
| 1.2 实验内容                             | 2         |
| <b>2 阶段一</b>                         | <b>2</b>  |
| 2.1 代码：运行第一个 C 程序                    | 2         |
| 2.1.1 CALL                           | 4         |
| 2.1.2 PUSH                           | 5         |
| 2.1.3 SUB                            | 5         |
| 2.1.4 XOR                            | 6         |
| 2.1.5 POP                            | 7         |
| 2.1.6 RET                            | 8         |
| <b>3 阶段二程序运行时环境与 AM</b>              | <b>8</b>  |
| 3.1 代码：运行更多的程序                       | 8         |
| 3.1.1 补全 opcode_table                | 8         |
| 3.1.2 Data Movement Instructions     | 8         |
| 3.1.3 Binary Arithmetic Instructions | 10        |
| 3.1.4 Logical Instructions           | 11        |
| 3.1.5 Control Transfer Instructions  | 13        |
| 3.2 基础设施                             | 13        |
| 3.2.1 代码：Differential Testing        | 13        |
| 3.2.2 一键回归测试                         | 14        |
| <b>4 阶段三</b>                         | <b>15</b> |
| 4.1 代码：加入 IOE                        | 15        |
| 4.1.1 串口                             | 15        |
| 4.1.2 时钟                             | 15        |
| 4.1.3 键盘                             | 16        |
| 4.1.4 VGA                            | 17        |
| 4.1.5 遇到的问题                          | 19        |
| 4.2 必答题                              | 20        |
| 4.2.1 static inline                  | 20        |
| 4.2.2 makefile                       | 20        |

# 1 概述

## 1.1 实验目的

了解各指令的实现过程, 对指令的实现有一个整体的把握; 对程序运行过程有一个整体的了解; 对于 IOE 的功能和实现有一个了解。

## 1.2 实验内容

1. 补全 rtl 指令和大部分指令, 能够运行足够多的程序。
2. 补全输入输出、时钟、键盘等部分, 实现与外设的交互。

# 2 阶段一

## 2.1 代码: 运行第一个 C 程序

### 实现标志寄存器

首先查阅 i386 手册。在第 34 页了解到 EFLAGS 的结构:

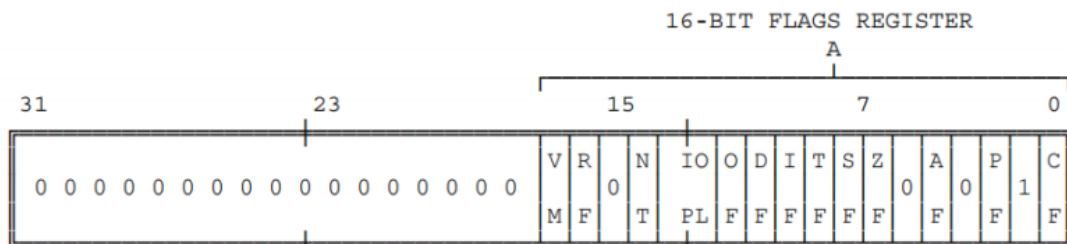


图 2.1: EFLAGS 寄存器

在第 174 页了解到 EFLAGS 初值为 02H。

因此 EFLAGS 结构应设计成一个联合体, 有一个数值来表示其值, 另有一个结构体具体标记出每个标志位。由于在 NEMU 中只用到 CF,ZF,SF,IF,OF, 因此只需指明这五个标志位。

```
union{
    uint32_t EFLAGS;
    struct{
        uint32_t CF:1;
        uint32_t :5;
        uint32_t ZF:1;
        uint32_t SF:1;
        uint32_t :1;
        uint32_t IF:1;
        uint32_t :1;
        uint32_t OF:1;
        uint32_t :20;
    }eflags;
};
```

在 restart() 函数中为 EFLAGS 赋初值做初始化。

```
static inline void restart() {
    /* Set the initial instruction pointer. */
    cpu.eip = ENTRY_START;
    cpu.EFLAGS = 0x000002;
```

### 实现所有 RTL 指令

NEMU 使用 RTL 来描述 x86 指令行为, 其 RTL 指令有 RTL 基本指令和 RTL 伪指令两种。

RTL 基本指令在即时编译技术里面可以只使用一条机器指令来实现相应的功能, 同时也不需要使  
用临时寄存器, 可以看做是最基本的 x86 指令中的最基本的操作。包括立即数读入、算术运算和逻辑  
运算、内存的访存、通用寄存器的访问, 这些指令在代码框架中已经实现。

RTL 伪指令通过 RTL 基本指令或者已经实现的 RTL 伪指令来实现, 包括带宽度的通用寄存器  
访问、EFLAGS 标志位的读写、其它常用功能如数据移动符号扩展等。这部分除带宽度的通用寄存器  
访问 rtl\_lr 和 rtl\_sr 以实现外, 均需要根据指令功能补全:

```
1 static inline void rtl_mv(rtlreg_t* dest, const rtlreg_t *src1) {
2     // dest <- src1
3     *dest = *src1;
4 }
5 static inline void rtl_not(rtlreg_t* dest) {
6     // dest <- ~dest
7     *dest = ~(*dest);
8 }
9 static inline void rtl_sext(rtlreg_t* dest, const rtlreg_t* src1, int width) {
10    // dest <- signext(src1[(width * 8 - 1) .. 0])
11    int32_t t = (int32_t)*src1;
12    t = t << (32 - width * 8);
13    t = t >> (32 - width * 8);
14    *dest = t;
15 }
16 static inline void rtl_push(const rtlreg_t* src1) {
17     // esp <- esp - 4
18     // M[esp] <- src1
19     cpu.esp = cpu.esp - 4;
20     vaddr_write(cpu.esp, 4, *src1);
21 }
22 static inline void rtl_pop(rtlreg_t* dest) {
23     // dest <- M[esp]
24     // esp <- esp + 4
25     rtl_lm(dest, &cpu.esp, 4);
26     cpu.esp = cpu.esp + 4;
27 }
28 static inline void rtl_eq0(rtlreg_t* dest, const rtlreg_t* src1) {
29     // dest <- (src1 == 0 ? 1 : 0)
30     *dest = (*src1==0)?1:0;
31 }
32 static inline void rtl_eqi(rtlreg_t* dest, const rtlreg_t* src1, int imm) {
33     // dest <- (src1 == imm ? 1 : 0)
34     *dest = (*src1==imm)?1:0;
35 }
36 static inline void rtl_neq0(rtlreg_t* dest, const rtlreg_t* src1) {
37     // dest <- (src1 != 0 ? 1 : 0)
38     *dest = (*src1!=0)?1:0;
39 }
40 static inline void rtl_msb(rtlreg_t* dest, const rtlreg_t* src1, int width) {
41     // dest <- src1[width * 8 - 1]
```

```

42  *dest = (*src1 >> (width * 8 - 1))&0x1;
43  }
44  static inline void rtl_update_ZF(const rtlreg_t* result, int width) {
45      // eflags.ZF <- is_zero(result[width * 8 - 1 .. 0])
46      int zf = 0;
47      if(width == 1) { zf = (*result & 0x000000ff) | 0; }
48      else if(width == 2) { zf = (*result & 0x0000ffff) | 0; }
49      else if(width == 4) { zf = (*result & 0xffffffff) | 0; }
50      cpu.eflags.ZF = zf==0 ? 1: 0;
51  }
52  static inline void rtl_update_SF(const rtlreg_t* result, int width) {
53      // eflags.SF <- is_sign(result[width * 8 - 1 .. 0])
54      rtlreg_t t;
55      rtl_msb(&t, result, width);
56      cpu.eflags.SF = t;
57  }
58  static inline void rtl_update_ZFSF(const rtlreg_t* result, int width) {
59      rtl_update_ZF(result, width);
60      rtl_update_SF(result, width);
61  }

```

接下来，我们实现 6 条 x86 指令。

### 2.1.1 CALL

根据最初的 make 提示信息以及反汇编文件，结合当前的 opcode\_table，可以知道首先要实现的是位于 e8 的 call rel32 指令。

查阅 i386 手册 P275 了解到 call rel32 指令的实现过程：eip 入栈，eip 更新为 eip + rel32，同时结合前文代码，得知要设置 decoding.is\_jump 为 1。

```

make_EHelper(call) {
    // the target address is calculated at the decode stage
    rtl_push(eip);
    decoding.is_jump = 1;
    rtl_add(&decoding.jump_eip, eip, &id_dest->val);

    print_asm("call %x", decoding.jump_eip);
}

```

重新 make 编译 dummy 程序，可以看到无效操作码的提示信息发生了变化：

```
invalid opcode(eip = 0x00100010): 55 89 e5 83 ec 08 e8 05 ...

There are two cases which will trigger this unexpected exception:
1. The instruction at eip = 0x00100010 is not implemented.
2. Something is implemented incorrectly.
Find this eip(0x00100010) in the disassembling result to distinguish which case it is.

If it is the first case, see
O3O6MManual
for more details.

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

图 2.2: 增加 call 指令, 编译 dummy 程序

说明 call rel32 指令成功实现。

### 2.1.2 PUSH

结合上面的编译提示信息以及反汇编文件, 得知下一步要实现的是位于 55 的 push r32 指令。查阅 i386 手册 P275 了解到 push r32 指令的实现只需要借助前面实现的 rtl\_push 来实现。

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

    print_asm_template1(push);
}
```

重新 make 编译 dummy 程序, 可以看到无效操作码的提示信息发生了变化:

```
invalid opcode(eip = 0x00100013): 83 ec 08 e8 05 00 00 00 ...

There are two cases which will trigger this unexpected exception:
1. The instruction at eip = 0x00100013 is not implemented.
2. Something is implemented incorrectly.
Find this eip(0x00100013) in the disassembling result to distinguish which case it is.

If it is the first case, see
O3O6MManual
for more details.

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

图 2.3: 增加 push 指令, 编译 dummy 程序

### 2.1.3 SUB

结合上面的编译提示信息以及反汇编文件, 得知下一步要实现的是位于 83 的 sub 指令, 在 exec.c 中看到 sub 指令要补在 gp1 里, 查阅 i386 手册 P145, 知道 sub 指令在 gp1 中的序号是 101, 在对应位置补上指令:

```
make_group(gp1,
    EMPTY, EMPTY, EMPTY, EMPTY,
    EMPTY, EXW(sub,1), EMPTY, EMPTY)
```

查阅 i386 手册 P404 了解到 sub 指令的实现方式，结合 arch.c 中的 make\_EHelper(sbb) 函数实现 sub 指令。

```
make_EHelper(sub) {
    rtl_sub(&t2, &id_dest->val, &id_src->val);
    rtl_sltu(&t3, &id_dest->val, &t2);
    rtl_get_CF(&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(sub);
}
```

重新 make 编译 dummy 程序，可以看到无效操作码的提示信息发生了变化：

```
invalid opcode(eip = 0x00100023): 31 c0 5d c3 00 00 00 00 ...

There are two cases which will trigger this unexpected exception:
1. The instruction at eip = 0x00100023 is not implemented.
2. Something is implemented incorrectly.
Find this eip(0x00100023) in the disassembling result to distinguish which case it is.

If it is the first case, see

[31] [c0] [5d] [c3] [00] [00] [00] [00]

for more details.

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

图 2.4: 增加 sub 指令，编译 dummy 程序

#### 2.1.4 XOR

结合上面的编译提示信息以及反汇编文件，得知下一步要实现的是位于 31 的 xor 指令，在对应位置补上指令：

```
1 /* 0x30 */ ~ ~ IEMPTY, IDEX(G2E, xor), EMPTY, EMPTY,
```

查阅 i386 手册 P411 了解到 xor 指令的实现需要借助 rtl\_xor，并对 ZF、SF 置位，将 OF、CF 置 0。在 logic.c 中补全：

```

make_EHelper(xor) {
    rtl_set_OF(&tzero);
    rtl_set_CF(&tzero);
    rtl_xor(&id_dest->val, &id_dest->val, &id_src->val);
    rtl_update_ZFSF(&id_dest->val, id_dest->width);
    operand_write(id_dest, &id_dest->val);

    print_asm_template2(xor);
}


```

重新 make 编译 dummy 程序，可以看到无效操作码的提示信息发生了变化：

```

invalid opcode(eip = 0x00100025): 5d c3 00 00 00 00 00 00 ...

There are two cases which will trigger this unexpected exception:
1. The instruction at eip = 0x00100025 is not implemented.
2. Something is implemented incorrectly.
Find this eip(0x00100025) 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!

```

图 2.5: 增加 xor 指令，编译 dummy 程序

### 2.1.5 POP

结合上面的编译提示信息以及反汇编文件，得知下一步要实现的是位于 5d 的 pop r32 指令。

```

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

```

查阅 i386 手册 P361 了解到 pop r32 指令的实现只需要借助前面实现的 rtl\_pop 来实现。

```

make_EHelper(pop) {
    rtl_pop(&t0);
    operand_write(id_dest, &t0);

    print_asm_template1(pop);
}

```

重新 make 编译 dummy 程序，可以看到无效操作码的提示信息发生了变化：



```
invalid opcode(eip = 0x00100026): c3 00 00 00 00 00 00 00 ...

There are two cases which will trigger this unexpected exception:
1. The instruction at eip = 0x00100026 is not implemented.
2. Something is implemented incorrectly.
Find this eip(0x00100026) in the disassembling result to distinguish which case it is.

If it is the first case, see
i386 Manual
for more details.

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

图 2.6: 增加 pop 指令, 编译 dummy 程序

### 2.1.6 RET

结合上面的编译提示信息以及反汇编文件, 得知下一步要实现的是位于 c3 的 ret 指令。

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

查阅 i386 手册 P378 了解到 ret 指令的实现需要 pop eip, 然后标记跳转来实现。

```
make_EHelper(ret) {
    rtl_pop(&decoding.jump_eip);
    decoding.is_jump = 1;

    print_asm("ret");
}
```

重新 make 编译 dummy 程序, 可以看到编译成功。

```
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 06:52:40, Apr 12 2024
For help, type "help"
(nemu) c
nemu: HIT GOOD TRAP at eip = 0x0010001b
```

图 2.7: 增加 ret 指令, 编译成功

## 3 阶段二程序运行时环境与 AM

### 3.1 代码: 运行更多的程序

#### 3.1.1 补全 opcode\_table

参考 i386 手册中的附录 A Opcode Map, 在 exec.c 中补全 opcode\_table。

由于涉及到的指令过多, 只补全了比较集中的一部分指令, 余下的在编译后续代码的过程中根据报错提示进行补全。

#### 3.1.2 Data Movement Instructions

需要补全的数据移动指令有 pusha、popa、leave、cld、cwtl。

1. pusha 指令是要将所有寄存器入栈，参考 i386 手册 P369，借助 rtl 指令 rtl\_push 将 eax、ecx、edx、ebx、esp、ebp、esi、edi 入栈。

```
make_EHelper(pusha) {
    t0 = cpu.esp;
    rtl_push(&cpu.eax);
    rtl_push(&cpu.ecx);
    rtl_push(&cpu.edx);
    rtl_push(&cpu.ebx);
    rtl_push(&t0);
    rtl_push(&cpu.ebp);
    rtl_push(&cpu.esi);
    rtl_push(&cpu.edi);

    print_asm("pusha");
}
```

2. popa 指令是要将所有寄存器出栈，参考 i386 手册 P364，借助 rtl 指令 rtl\_pop 将 eax、ecx、edx、ebx、edi、esi、ebp、esp、ebx、edx、ecx、eax 出栈。

```
make_EHelper(popa) {
    rtl_pop(&cpu.edi);
    rtl_pop(&cpu.esi);
    rtl_pop(&cpu.ebp);
    rtl_pop(&t0);
    rtl_pop(&cpu.ebx);
    rtl_pop(&cpu.edx);
    rtl_pop(&cpu.ecx);
    rtl_pop(&cpu.eax);

    print_asm("popa");
}
```

3. leave 指令相当于 mov esp, ebp 和 pop ebp，可以参考 i386 手册 P329。

```
make_EHelper(leave) {
    rtl_mv(&cpu.esp, &cpu.ebp);
    rtl_pop(&cpu.ebp);

    print_asm("leave");
}
```

4. cldt 指令作用是扩展 eax，根据操作数位数有不同的操作。

```
make_EHelper(cldt) {
    if (decoding.is_operand_size_16) {
        rtl_lr_w(&t0, R_AX);
        rtl_sext(&t0, &t0, 2);
        rtl_sari(&t0, &t0, 16);
        rtl_sr_w(R_DX, &t0);
    }
    else {
        rtl_lr_l(&t0, R_EAX);
        rtl_sari(&t0, &t0, 31);
        rtl_sari(&t0, &t0, 1);
        rtl_sr_l(R_EDX, &t0);
    }

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

5. cwtl 指令作用是扩展 eax，根据操作数位数有不同的操作。

```
make_EHelper(cwtl) {
    if (decoding.is_operand_size_16) {
        rtl_lr_b(&t0, R_AX);
        rtl_sext(&t0, &t0, 1);
        rtl_sr_w(R_AX, &t0);
    }
    else {
        rtl_lr_w(&t0, R_EAX);
        rtl_sext(&t0, &t0, 2);
        rtl_sr_l(R_EAX, &t0);
    }

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

### 3.1.3 Binary Arithmetic Instructions

需要补全的运算指令有 add、cmp、inc、dec、neg。

1. add 指令的实现可以参考 adc 指令和 i386 手册 P261，将运算数通过 rtl\_add 相加，置位 OF、SF、ZF、CF。

```
make_EHelper(add) {
    rtl_add(&t2, &id_dest->val, &id_src->val);
    operand_write(id_dest, &t2);

    rtl_update_ZFSF(&t2, id_dest->width);

    rtl_sltu(&t0, &t2, &id_dest->val);
    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(add);
}
```

2. cmp 指令是将两操作数相减，根据结果是否为 0 进行置位，参考 i386 手册 P287，置位 OF、SF、ZF、CF。

```
make_EHelper(cmp) {
    rtl_sub(&t2, &id_dest->val, &id_src->val);
    rtl_update_ZFSF(&t2, id_dest->width);

    rtl_sltu(&t0, &id_dest->val, &t2);
    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(cmp);
}
```

3. inc、dec 指令就是将 add 和 sub 的一个操作数变成了 1，参考 add、sub 的实现，结合 i386 手册 P303、P293 实现。

```
make_EHelper(inc) {
    rtl_addi(&t2, &id_dest->val, 1);
    operand_write(id_dest, &t2);

    rtl_update_ZFSF(&t2, id_dest->width);

    rtl_xori(&t0, &id_dest->val, 1);
    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_template1(inc);
}
```

```
make_EHelper(dec) {
    rtl_subi(&t2, &id_dest->val, 1);
    operand_write(id_dest, &t2);

    rtl_update_ZFSF(&t2, id_dest->width);

    rtl_xori(&t0, &id_dest->val, 1);
    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_template1(dec);
}
```

4. neg 指令是求补指令也就是按位取反再加 1，结合 i386 手册 P354 实现，置位 OF、SF、ZF、CF。

```
make_EHelper(neg) {
    rtl_mv(&t2, &id_dest->val);
    rtl_not(&t2);
    rtl_addi(&t2, &t2, 1);
    operand_write(id_dest, &t2);

    rtl_update_ZFSF(&t2, id_dest->width);

    rtl_eq0(&t1, &id_dest->val);
    rtl_set_CF(&t1);

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

    print_asm_template1(neg);
}
```

### 3.1.4 Logical Instructions

需要补全的逻辑指令有 test、and、or、sar、shl、shr、not、rol。

1. test 指令是做逻辑与，根据与运算结果进行置位，使用到了 rtl 指令 rtl\_and，参考 i386 手册 P405，置位 OF、CF、ZF、SF。

```
make_EHelper(test) {
    rtl_set_OF(&tzero);
    rtl_set_CF(&tzero);
    rtl_and(&t2, &id_dest->val, &id_src->val);
    rtl_update_ZFSF(&t2, id_dest->width);

    print_asm_template2(test);
}
```

2. and 指令是做逻辑与, 使用到了 rtl 指令 rtl\_and, 参考 i386 手册 P262, 置位 OF、CF、ZF、SF, 其中 OF、CF 置为 0。

```
make_EHelper(and) {
    rtl_and(&t2, &id_dest->val, &id_src->val);
    operand_write(id_dest, &t2);
    rtl_update_ZFSF(&t2, id_dest->width);
    rtl_set_OF(&tzero);
    rtl_set_CF(&tzero);

    print_asm_template2(and);
}
```

3. or 指令是做逻辑或, 使用到了 rtl 指令 rtl\_or, 参考 i386 手册 P356, 置位 OF、CF、ZF、SF, 其中 OF、CF 置为 0。

```
make_EHelper(or) {
    rtl_set_OF(&tzero);
    rtl_set_CF(&tzero);
    rtl_or(&t2, &id_dest->val, &id_src->val);
    operand_write(id_dest, &t2);
    rtl_update_ZFSF(&t2, id_dest->width);

    print_asm_template2(or);
}
```

4. sar 指令是算数右移指令, 使用到了 rtl 指令 rtl\_sar, 参考 i386 手册 P383, 在 nemu 中只需置位 ZF、SF。

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

    print_asm_template2(sar);
}
```

5. shl 指令是逻辑左移指令, 使用到了 rtl 指令 rtl\_shl, 参考 i386 手册 P383, 在 nemu 中只需置位 ZF、SF, 函数实现只需要将 sar 实现函数中的 rtl\_sar 替换为 rtl\_shl。

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

    print_asm_template2(shl);
}
```

6. shr 指令是逻辑右移指令, 使用到了 rtl 指令 rtl\_shl, 参考 i386 手册 P383, 在 nemu 中只需置位 ZF、SF, 函数实现只需要将 sar 实现函数中的 rtl\_sar 替换为 rtl\_shr。

```
make_EHelper(shr) {
    rtl_shr(&id_dest->val, &id_dest->val, &id_src->val);
    operand_write(id_dest, &id_dest->val);
    rtl_update_ZFSF(&id_dest->val, id_dest->width);
    // unnecessary to update CF and OF in NEMU
    print_asm_template2(shr);
}
```

### 3.1.5 Control Transfer Instructions

需要补全的控制指令有 jmp\_rm、call\_rm。

1. jmp\_rm 指令

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

2. call\_rm 指令

```
make_EHelper(call_rm) {
    rtl_push(eip);
    rtl_mv(&decoding.jmp_eip, &id_dest->val);
    decoding.is_jump = 1;
    print_asm("call %s", id_dest->str);
}
```

## 3.2 基础设施

### 3.2.1 代码: Differential Testing

diff-test 的实现就是去比较 NEMU 和 QEMU 寄存器的值, 如果有不同, 将 diff 置为 true, 停止程序运行。

```
1  if (r.eax != cpu.eax){
2      diff = true;
3      printf("QEMU.eax:0x%x  NEMU.eax:0x%x \n", r.eax, cpu.eax);
4  }
5  if (r.ecx != cpu.ecx){
6      diff = true;
7      printf("QEMU.ecx:0x%x  NEMU.ecx:0x%x \n", r.ecx, cpu.ecx);
8  }
9  if (r.edx != cpu.edx){
10     diff = true;
11     printf("QEMU.edx:0x%x  NEMU.eax:0x%x \n", r.edx, cpu.edx);
12 }
13 if (r.ebx != cpu.ebx){
14     diff = true;
15     printf("QEMU.ebx:0x%x  NEMU.ebx:0x%x \n", r.ebx, cpu.ebx);
```

```

16 }
17 if (r.esp != cpu.esp){
18     diff = true;
19     printf("QEMU.esp:0x%x  NEMU.esp:0x%x \n", r.esp, cpu.esp);
20 }
21 if (r.ebp != cpu.ebp){
22     diff = true;
23     printf("QEMU.ebp:0x%x  NEMU.ebp:0x%x \n", r.ebp, cpu.ebp);
24 }
25 if (r.esi != cpu.esi){
26     diff = true;
27     printf("QEMU.esi:0x%x  NEMU.esi:0x%x \n", r.esi, cpu.esi);
28 }
29 if (r.edi != cpu.edi){
30     diff = true;
31     printf("QEMU.edi:0x%x  NEMU.edi:0x%x \n", r.edi, cpu.edi);
32 }
33 if (r.eip != cpu.eip){
34     diff = true;
35     printf("QEMU.eip:0x%x  NEMU.eip:0x%x \n", r.eip, cpu.eip);
36 }
37 if (diff) {
38     printf("NEMU.eip:0x%x \n", cpu.eip);
39     nemu_state = NEMU_END;
40 }

```

### 3.2.2 一键回归测试

在 nemu 目录下终端运行命令 `bash runall.sh`

```

yxy@ubuntu:~/ics2017/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!

```

图 3.8: 一键回归测试结果

## 4 阶段三

### 4.1 代码：加入 IOE

#### 4.1.1 串口

在 src/cpu/exec/system.c 里补全 in/out 指令：

```
make_EHelper(in) {
    t1 = pio_read(id_src->val, id_dest->width);
    operand_write(id_dest, &t1);

    print_asm_template2(in);

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

```
make_EHelper(out) {
    pio_write(id_dest->val, id_dest->width, id_src->val);

    print_asm_template2(out);

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

在 opcode\_table 中补全对应的操作码。在 nexus-am/am/arch/x86-nemu/src/trm.c 中定义宏 HAS\_SERIAL。编译 hello 程序：

```
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 00:50:43, Apr 14 2024
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!
Hello World!
nemu: HIT GOOD TRAP at eip = 0x0010006e
```

图 4.9: 输入输出

#### 4.1.2 时钟

##### 实现 IOE

在 nexus-am/am/arch/x86-nemu/src/ioe.c 里补全 \_\_uptime() 函数、

```
1 unsigned long __uptime() {
2     return inl(RTC_PORT) - boot_time;
3 }
```

编译 timetest 程序：



```

Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 00:50:43, Apr 14 2024
For help, type "help"
(nemu) c
1 second.
2 seconds.
3 seconds.
4 seconds.
5 seconds.
6 seconds.
7 seconds.
8 seconds.
9 seconds.
10 seconds.

```

图 4.10: 时钟

### 看看 NEMU 跑多快

在 nemu 和 native 下分别对 Dhrystone、Coremark、microbench 进行跑分测试。

```

Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 00:50:43, Apr 14 2024
For help, type "help"
(nemu) c
Dhrystone Benchmark, Version C, Version 2.2
Trying 500000 runs through Dhrystone.
Finished in 10236 ms
=====
Dhrystone PASS      100 Marks
vs. 100000 Marks (17-6700 @ 3.40GHz)
nemu: HIT GOOD TRAP at etp = 0x0010006e

```

(a) Dhrystone-nemu

```

Dhrystone Benchmark, Version C, Version 2.2
Trying 500000 runs through Dhrystone.
Finished in 17 ms
=====
Dhrystone PASS      60604 Marks
vs. 100000 Marks (17-6700 @ 3.40GHz)

```

(b) Dhrystone-native

```

Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 00:50:43, Apr 14 2024
For help, type "help"
(nemu) c
Running CoreMark for 1000 iterations
2K performance run parameters for coremark.
CoreMark Size : 666
Total time (ms) : 11304
Iterations : 1000
Compiler version : GCCS.4.0 20160609
seedcrc : 0xe9f5
[0]jcrclist : 0xc714
[0]jcrmatrix : 0xf4d7
[0]jcrstate : 0x8e3a
[0]jcrfinal : 0xd340
Finished in 11304 ms.
=====
CoreMark PASS      395 Marks
vs. 100000 Marks (17-6700 @ 3.40GHz)
nemu: HIT GOOD TRAP at etp = 0x0010006e

```

(c) Coremark-nemu

```

Running CoreMark for 1000 iterations
2K performance run parameters for coremark.
CoreMark Size : 666
Total time (ms) : 72
Iterations : 1000
Compiler version : GCCS.4.0 20160609
seedcrc : 0xe9f5
[0]jcrclist : 0xc714
[0]jcrmatrix : 0xf4d7
[0]jcrstate : 0x8e3a
[0]jcrfinal : 0xd340
Finished in 72 ms.
=====
CoreMark PASS      62064 Marks
vs. 100000 Marks (17-6700 @ 3.40GHz)

```

(d) Coremark-native

```

Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 00:50:43, Apr 14 2024
For help, type "help"
(nemu) c
[qsrt] Quick sort: * Passed.
min time: 784 ms [703]
[queen] Queen placement: * Passed.
min time: 1232 ms [418]
[bf] Brainf**k Interpreter: * Passed.
min time: 6827 ms [383]
[fib] Fibonacci number: * Passed.
min time: 13378 ms [140]
[sieve] Eratosthenes sieve: * Passed.
min time: 11309 ms [374]
[ispz] A* 15-puzzle search: * Passed.
min time: 2491 ms [232]
[dinic] Dinic's maxflow algorithm: * Passed.
min time: 2176 ms [622]
[lzip] Lzip compression: * Passed.
min time: 6366 ms [415]
[ssort] Suffix sort: * Passed.
min time: 1140 ms [518]
[md5] MD5 digest: * Passed.
min time: 11268 ms [173]
=====
MicroBench PASS      404 Marks
vs. 100000 Marks (17-6700 @ 3.40GHz)
nemu: HIT GOOD TRAP at etp = 0x00100032

```

(e) microbench-nemu

```

[qsrt] Quick sort: * Passed.
min time: 7 ms [78842]
[queen] Queen placement: * Passed.
min time: 8 ms [64487]
[bf] Brainf**k Interpreter: * Passed.
min time: 24 ms [109204]
[fib] Fibonacci number: * Passed.
min time: 42 ms [68033]
[sieve] Eratosthenes sieve: * Passed.
min time: 47 ms [90225]
[ispz] A* 15-puzzle search: * Passed.
min time: 9 ms [64355]
[dinic] Dinic's maxflow algorithm: * Passed.
min time: 15 ms [90240]
[lzip] Lzip compression: * Passed.
min time: 37 ms [71537]
[ssort] Suffix sort: * Passed.
min time: 2 ms [85722]
[md5] MD5 digest: * Passed.
min time: 27 ms [72566]
=====
MicroBench PASS      77521 Marks
vs. 100000 Marks (17-6700 @ 3.40GHz)

```

(f) microbench-native

### 4.1.3 键盘

在 nexus-am/am/arch/x86-nemu/src/ioe.c 实现 `_read_key()` 函数:

```

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

```

编译 keytest:

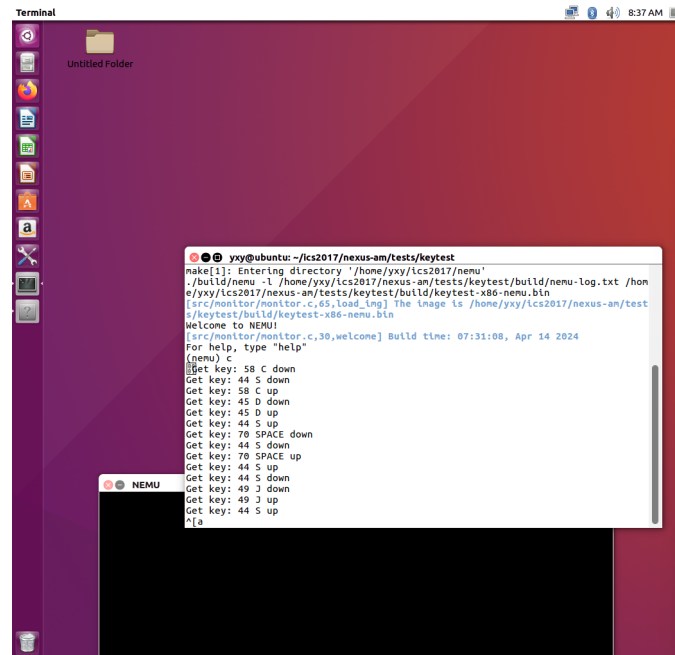


图 4.11: 键盘

#### 4.1.4 VGA

##### 内存 IO 映射

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被映射到 I/O 空间, 如果是, `is_mmio()` 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. 如果不是直接访问 `pmem`.

```
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) {
    if(is_mmio(addr) == -1)
        memcpy(guest_to_host(addr), &data, len);
    else
        mmio_write(addr, len, data, is_mmio(addr));
}
```

编译运行行 `videotest` 程序:

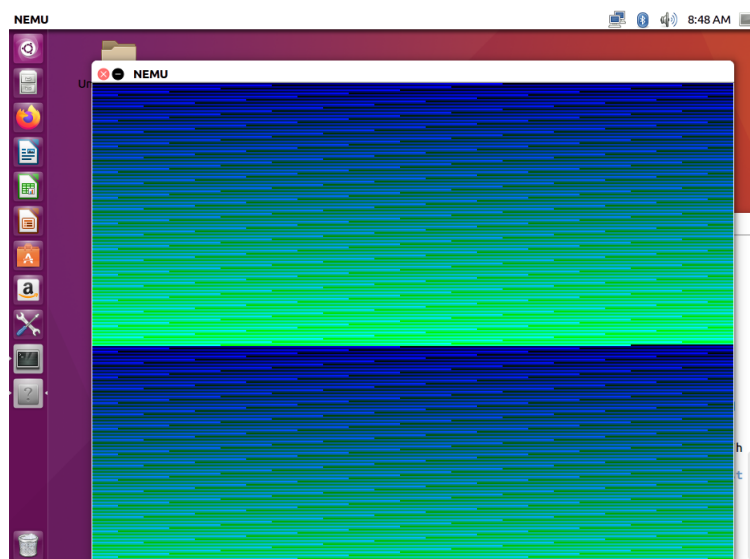


图 4.12: 实现内存 IO 映射

正确输出一些色彩。

### 完整实现 IOE

补全 `_draw_rect` 函数：

```
void _draw_rect(const uint32_t *pixels, int x, int y, int w, int h) {  
    // int i;  
    // for (i = 0; i < _screen.width * _screen.height; i++) {  
    //     fb[i] = i;  
    // }  
  
    int i, j;  
    int k = 0;  
    for (i = y; i < y+h; i++) {  
        for (j = x; j < x+w; j++) {  
            fb[_screen.width*i+j] = pixels[k++];  
        }  
    }  
}
```

实现每个像素点的正确映射，运行 `videotest`：

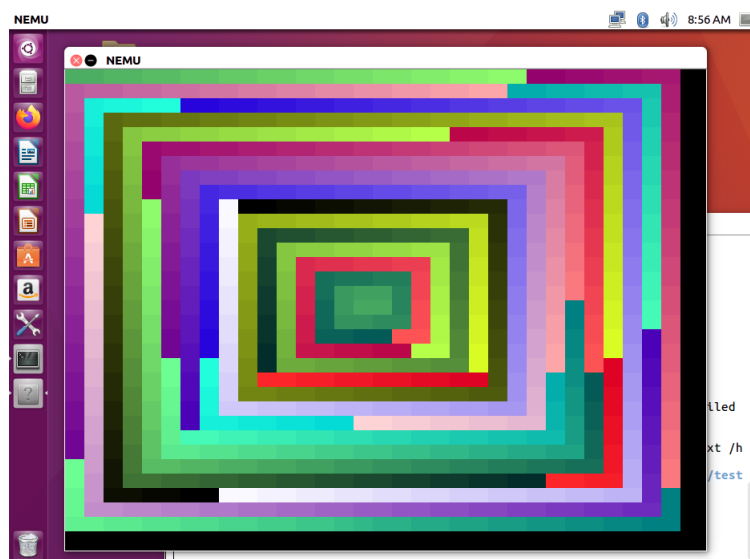


图 4.13: videotest

得到正确的动画效果。

编译运行打字小游戏：

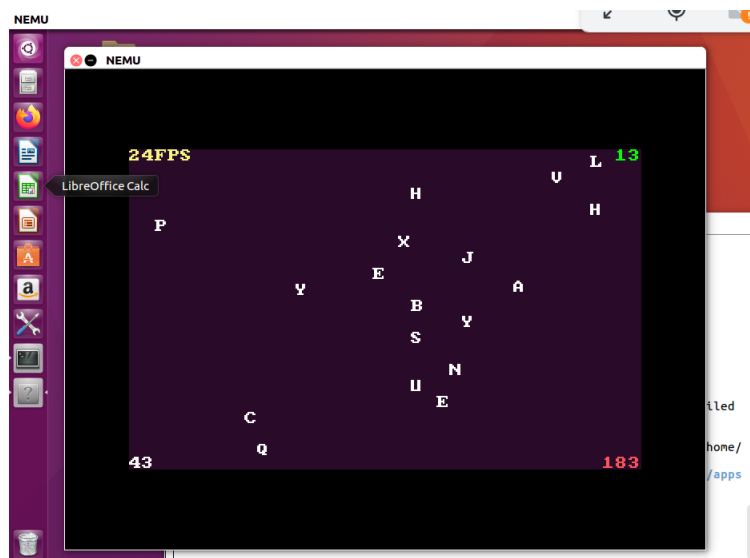


图 4.14: 打字小游戏

运行正确正确输出了可运行的打字小游戏。

#### 4.1.5 遇到的问题

出来窗口之后没点 C, de 了半天 Bug。

## 4.2 必答题

### 4.2.1 static inline

**Q:** 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?

**A:** static inline 结合了 static 和 inline 关键字的特性。static 表示该函数在编译时被静态分配到内存中, 并且只能在当前文件中被访问。这样可以避免与其他文件中同名的函数产生冲突, 同时也有助于提高程序的安全性和可维护性。inline 表示该函数是一个内联函数, 即在 inline 编译时将函数代码直接嵌入到调用该函数的位置。这样可以减少函数调用和返回的开销, 提高程序的运行效率。

### 4.2.2 makefile

**Q:** 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.)

**A:**

Makefile 的基本工作方式:

Makefile 的基本工作方式是将源文件以及它们的依赖关系描述在 Makefile 文件中, 然后由 make 命令来解析 Makefile 文件并执行相应的命令来生成目标文件。Makefile 文件中的每个目标都对应着一个或多个依赖文件和生成目标文件的命令。当某个依赖文件被修改后, make 就会重新生成相应的目标文件。这样可以实现自动化编译和链接的过程, 提高程序开发效率。

Makefile 的编译链接过程:

1. 预处理: 将 include、define 和 ifdef 等命令进行处理, 生成.i 文件;
2. 编译: 将预处理后的.i 文件进行编译, 生成.s 文件, 其中包括汇编指令、常量、变量以及计算机指令等;
3. 汇编: 将编译后的.s 文件进行汇编, 生成.o 文件, 其中包括机器指令、符号表以及其他调试相关信息;
4. 链接: 将多个.o 文件合并成一个目标文件, 同时解析和处理链接时引发的各种问题, 生成可执行文件。

在 Makefile 中, 我们可以通过定义变量和规则来指定编译过程中的命令和文件依赖关系, 以及如何生成最终的可执行文件。

nemu 的 build 过程, 是从.c->.h->.s->.o