

# PA2实验报告

1811\*\*\*\_秦夙

## Part1

### 1.1 实验目的

1. 了解计算机取指、译码、执行等一系列过程
2. 学习结构化程序设计
3. 学习用RTL表示指令行为

### 1.2 实验内容

1. 实现RTL指令
2. 实现 `call`、`push`、`pop`、`sub`、`xor` 和 `ret` 指令
3. 运行dummy.c

### 1.3 实验步骤

#### 1.3.1 实现 EFLAGS 寄存器结构

EFLAGS 寄存器的状态标志允许一个指令的结果影响以后的指令。算术指令使用 OF、SF、ZF、AF、PF 和 CF。SCAS（扫描字符串）、CMPS（比较字符串）和LOOP指令使用ZF发出信号，表示其操作已完成。在执行算术指令之前，有一些需要设置、清除和补充CF的指令。具体的描述可以在i386手册的34页和附录C中找到。

根据实验指导手册的提示，实现sub指令前需要先实现 EFLAGS 寄存器，同时RTL指令中也有设置、获取 EFLAGS 寄存器不同标志位的指令，所以在此先实现 EFLAGS 寄存器。

EFLAGS 寄存器的实现需要在 `nemu/include/cpu/reg.h` 的 `CPU_state` 中添加相应的结构体，具体实现仿照PA1中的寄存器实现：使用union联合体，union中包含一个用于表示整个寄存器的值的成员，以及一个由用于表示寄存器每一位的成员组成的结构体。实验指导书中提到在nemu中只用到 EFLAGS 寄存器中的特定几位，但在此选择将寄存器中的全部位写明。根据实验指导书的提示使用结构体位域表示寄存器的不同位，具体代码如下，其中flagi用于占位：

```
union{ //eflags
uint32_t eflagsVal; //用于赋初值
struct {
    uint32_t CF: 1;
    uint32_t flag0: 1;
    uint32_t PF: 1;
    uint32_t flag1: 1;
    uint32_t AF: 1;
    uint32_t flag2: 1;
    uint32_t ZF: 1;
    uint32_t SF: 1;
}
```

```

uint32_t TF: 1;
uint32_t IF: 1;
uint32_t DF: 1;
uint32_t OF: 1;
uint32_t OLIP: 2;
uint32_t NT: 1;
uint32_t flag3: 1;
uint32_t RF: 1;
uint32_t VM: 1;
uint32_t flag4: 14;
};
} eflags;

```

实现 `EFLAGS` 之后还需要为其赋初值，通过i386手册可知该寄存器的初值是00000002H，在 `nemu/src/monitoe/monitor.c` 的 `restart` 函数中设置 `eflagsVal` 为00000002H：

```
cpu.eflags.eflagsVal = 2;
```

### 1.3.2 实现RTL伪指令

RTL指令的实现需要完善 `nemu/include/rtl.h` 文件，其中部分指令框架中已经完成，我们需要填写其他未完成的部分。下面按照RTL指令的功能分为不同几组，介绍RTL指令的实现。

#### 1.3.2.1 与设置、获取 `EFLAGS` 寄存器不同位相关的RTL伪指令

与设置、获取 `EFLAGS` 寄存器不同位相关的RTL指令分为两部分，第一部分是设置、获取 `CF`、`OF`、`ZF`、`SF` 位的RTL指令，第二部分是更新 `ZF`、`SF` 位的RTL指令。

第一部分使用宏定义批量生成了四个 `EFLAGS` 位的获取和设置函数，只需要将宏定义补充完整即可，在设置函数中，将 `EFLAGS` 的对应位设置为传入的值，在获取函数中将 `EFLAGS` 的对应位赋给引用传参传入的参数。补充后的宏定义如下：

```

#define make_rtl_setget_eflags(f) \
    static inline void concat(rtl_set_, f) (const rtlreg_t* src) { \
        cpu.eflags.f = (*src == 0 ? 0 : 1); \
    } \
    static inline void concat(rtl_get_, f) (rtlreg_t* dest) { \
        *dest = cpu.eflags.f; \
    }

```

第二部分以函数的形式实现，分为更新 `ZF` 的函数、更新 `SF` 的函数，和更新 `ZF`、`SF` 的函数。参数均为运算结果和指令参数。

`ZF` 标志位在结果所有位都为0时置1，即结果为0时置1，否则置0，设置`ZF` 标志位的函数如下：

```
static inline void rtl_update_ZF(const rtlreg_t* result, int width) {
    // 所有位都为0时置1
    cpu.eflags.ZF = *result == 0 ? 1 : 0;
}
```

SF 标志位在结果为负时置1，为正时置0，更新 SF 标志位时只需检查结果最高位是否为1即可知道结果是正数还是负数，函数如下：

```
static inline void rtl_update_SF(const rtlreg_t* result, int width) {
    // 有符号整形，SF 0为正，1为负
    cpu.eflags.SF = (*result)>>(width*8-1);
}
```

更新 ZF、SF 的函数只需要分别调用上述两个函数即可，代码如下：

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

### 1.3.2.2 判断是否相等的RTL伪指令

判断结果是否与某值相等的RTL指令实现较为简单，只需要根据比较结果为 \*dest 赋值即可，代码如下：

```
static inline void rtl_eq0(rtlreg_t* dest, const rtlreg_t* src1) {
    *dest = (*src1 == 0 ? 1 : 0);
}

static inline void rtl_eqi(rtlreg_t* dest, const rtlreg_t* src1, int imm) {
    *dest = (*src1 == imm ? 1 : 0);
}

static inline void rtl_neq0(rtlreg_t* dest, const rtlreg_t* src1) {
    *dest = (*src1 != 0 ? 1 : 0);
}
```

### 1.3.2.3 pop push 相关的RTL伪指令

pop push 指令根据实验指导书中的描述，需要修改栈顶指针的值，即 esp 寄存器的值。具体的，对于 pop 指令，需要先将栈顶指针此时指向的内存位置的值赋给 \*dest，之后将指针值加4；对于 push 指令，需要先将栈顶指针的值减4，之后将传入的 \*src 写入栈顶指针指向的内存。对内存的读写可以使用已经实现的RTL指令 rtl\_lm 和 rtl\_sm，对指针值的加减可以使用已经实现的RTL指令 rtl\_addi 和 rtl\_subi。代码如下：

```
static inline void rtl_pop(rtlreg_t* dest) {
    rtl_lm(dest, &cpu.esp, 4);
    rtl_addi(&cpu.esp, &cpu.esp, 4);
}

static inline void rtl_push(const rtlreg_t* src1) {
    rtl_subi(&cpu.esp, &cpu.esp, 4);
    rtl_sm(&cpu.esp, 4, src1);
}
```

### 1.3.2.4 其他RTL伪指令

除上述RTL伪指令之外，还有一些其他的指令需要实现：

1. 移动数据的 `rtl_mv` 指令：只需要将源操作数赋值给目的操作数即可

```
static inline void rtl_mv(rtlreg_t* dest, const rtlreg_t *src1) {
    *dest = *src1;
}
```

2. 取反的 `rtl_not` 指令：只需要将传入的值取反即可

```
static inline void rtl_not(rtlreg_t* dest) {
    *dest = !(*dest);
}
```

3. 获取最高位的 `rtl_msb` 指令：可以通过位运算获得 `*src` 的最高位

```
static inline void rtl_msb(rtlreg_t* dest, const rtlreg_t* src1, int width)
{
    *dest = (*src1)>>(width*8-1);
}
```

4. 用于符号拓展的 `rtl_sext`：首先根据传入的原始数据宽度 `width` 判断是否需要进行拓展：如果宽度为4则不需要拓展，直接将源操作数赋值给目的操作数即可；先否则计算需要增加的位数，之后通过位运算进行拓展

```
static inline void rtl_sext(rtlreg_t* dest, const rtlreg_t* src1, int width)
{
    if (width == 4){
        rtl_mv(dest, src1);
    }
    else{
        int shiftByBit = (4-width) * 8;

```

```

    assert(width==1 || width==2);
    *dest = ((*src1)<<shiftByBit)>>shiftByBit;
}
}

```

### 1.3.3 运行 dummy.c

此时我们还没有实现 `dummy.c` 中需要的指令如 `call`、`push` 等，故 `dummy.c` 不能正常运行。但在实现这些指令的过程中，跟踪 `dummy.c` 的指令执行过程会很有帮助，我们可以根据程序停下的地方确定是哪一条指令需要实现，以及该条指令的操作数是什么。所以在此先说明运行 `dummy.c` 的步骤。

按照实验指导书所描述的，运行 `dummy.c` 需要在 `nexus-am/tests/cputest` 目录下执行指令：

```
make ARCH=x86-nemu ALL=dummy run
```

该指令会编译 `dummy.c` 并在 `nemu` 中运行它。但直接执行该指令会报错，我们还需要设置一些环境变量，在 `nexus-am/tests/cputest` 目录下执行以下命令：


```
export NEMU_HOME=/home/用户名/Desktop/ics2018/nemu
export AM_HOME=/home/用户名/Desktop/ics2018/nexus-am
export NAVY_HOME=/home/dsr/tests/ics2018/navy-apps
```

之后在运行上一条指令就可以开始运行 `dummy.c`，此时我们可以直接执行 `c` 指令让程序运行到待完成指令处，也可以使用 `si` 指令单步执行程序直到程序停下，程序停止时出现的指令就是待完成指令，我们可以直接在界面上看到该指令的操作码。

```

(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!

(nemu)

```

### 1.3.3 实现 `call`、`push`、`pop`、`sub`、`xor` 和 `ret` 指令

运行 `dummy.c` 之后我们可以很容易地获得需要实现的指令以及指令的操作码，接下来我们开始实现这些指令。

实现指令的方法分为以下几个步骤：

1. 在 `src/cpu/exec/exec.c` 的 `opcode_table` 中对应位置使用对应的宏定义，完成操作数表；
2. 如果有需要，在 `src/cpu/exec/exec.c` 的相应group中填写宏定义；
3. 在 `src/cpu/exec/all-in.h` 中使用 `make_EHelper` 定义对应函数；
4. 在 `src/cpu/exec` 文件夹中找到执行该指令的函数并补充完整。

我们按照上述步骤实现指令：首先，完成操作数表需要通过查阅i386手册获得指令对应的操作数以及其他信息。以 `xor` 指令为例简述实现一条指令的具体过程。运行 `dummy.c` 之后，我们可以在程序停下时的界面获得下一条需要实现的指令的操作码，也可以通过查看 `dummy.c` 编译产生的文件 `dummy-x86-nemu.txt`，直接查找某一条指令的操作码。通过查找，我们得知 `xor` 的操作码为31，之后我们在i386手册附录A的操作码表中找第三行第一列，对应内容为 `Ev, Gv`，再在手册中查找关于 `xor` 指令的描述部分，其中操作码为31的 `xor` 的操作数宽度为16或32位，于是我们在 `opcode_table` 中第 0x31 位置填入 `IDEX(G2E, xor)`（如果操作数宽度为8位则填 `IDEXW(G2E, xor, 1)`）。`IDEX` 是宏定义，`ID` 表明需要解码，解码类型为 `G2E`，`EX` 是定义执行函数，函数名是一个前缀拼接上 `xor`。

因为i386手册中31位置并未涉及group，所以此处我们不需要做第二步操作，直接进行第三步。

在 `src/cpu/exec/all-in.h` 中仿照原有的内容，使用 `make_EHelper` 宏定义定义对应函数，代码如下：

```
make_EHelper(xor);
```

最后补充完整执行该指令的函数，该指令定义在 `src/cpu/exec/logic.c` 中。为了将函数补充完成，我们还需要再次查阅i386手册，找到手册中对 `xor` 指令的具体说明，根据手册说明我们得知 `xor` 指令计算两个数字的按位与结果，运算结果会影响 `EFLAG` 寄存器的 `CF`、`OF`、`ZF`、`SF` 位。实现该函数时，我们首先调用 `rtl_xor` 函数计算按位与结果，之后调用 `operand_write` 函数将计算结果写回相应位置，最后调用对应的rtl指令更新 `EFLAG` 寄存器。代码如下：

```
make_EHelper(xor) {
    rtl_xor(&t0, &id_dest->val, &id_src->val); //计算结果
    operand_write(id_dest, &t0); //写回
    rtl_set_CF(&tzero);
    rtl_set_OF(&tzero);
    rtl_update_ZFSF(&t0, id_dest->width); //更新 EFLAG 寄存器
    print_asm_template2(xor);
}
```

其与几条指令的实现流程与 `xor` 大致相同，在此不再详述其实现流程，只说明实现具体函数的过程。

**call:** `call` 指令实现程序的跳转，根据实验指导书，实现 `call` 指令转可以通过将 `decoding.is_jump` 设为 1，并将 `decoding.jump_eip` 设为跳转目标地址来实现，同时根据i386手册，在跳转前我们还需要记录跳转时的指令地址，具体代码如下：

```
make_EHelper(call) {
    rtl_li(&t2, decoding.seq_eip);
    rtl_push(&t2); //记录跳转前地址
    decoding.is_jump = 1;
    print_asm("call %x", decoding.jump_eip);
}
```

**ret:** `ret` 指令与 `call` 指令类似，首先从栈中获取返回时要跳转到的地址，之后将 `decoding.is_jump` 设为 1，并将 `decoding.jump_eip` 设为返回地址：

```
make_EHelper(ret) {
    rtl_pop(&t0);
    decoding.jump_eip = t0; //返回地址
    decoding.is_jump = 1;
    print_asm("ret");
}
```

**push pop:** `push` 和 `pop` 指令可以通过 `rtl_push` 和 `rtl_pop` 来实现，在实现 `pop` 时要记得将 `pop` 出的结果写回：

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

make_EHelper(pop) {
    rtl_pop(&t2);
    operand_write(id_dest, &t2); //写回结果
    print_asm_template1(pop);
}
```

**sub:** 在实现 `sub` 指令的时候要注意其中操作码为 83 的 `sub` 指令，在 i386 手册中的 0x83 位置标出了 `Grp1`，表示执行函数为指令组，故需要在 `opcode_table` 中第 0x83 位置填入 `IDEX(SI2E, gp1)`，并在 `make_group(gp1, ...)` 中对应位置填入 `sub`；`sub` 实现函数时，首先调用 `rtl_sub` 计算减法结果，并将结果写回，之后还需要处理 `EFLAGS` 寄存器，具体计算结果如何影响该寄存器的值可以在 i386 手册的附录 C 中找到：

```
make_EHelper(sub) {
    rtl_sub(&t2, &id_dest->val, &id_src->val); //计算
    rtl_sltu(&t3, &id_dest->val, &t2);
    operand_write(id_dest, &t2); //写回结果

    rtl_update_ZFSF(&t2, id_dest->width); //设置 ZF SF

    rtl_sltu(&t0, &id_dest->val, &t2);
    rtl_or(&t0, &t3, &t0);
    rtl_set_CF(&t0); //设置 CF
```

```

    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); //设置 OF

    print_asm_template2(sub);
}

```

上述五个指令全部实现后，再次执行 `dummy.c` 会看到 `HIT GOOD TRAP` 结果。

```

qin2mu4@ubuntu:~/Desktop/ics2018/nexus-am/tests/cputest$ make ARCH=x86-nemu ALL=
dummy run
Building dummy [x86-nemu]
Building am [x86-nemu]
make[2]: *** 没有指明目标并且找不到 makefile。 停止。
[src/monitor/monitor.c,65,load_img] The image is /home/qin2mu4/Desktop/ics2018/n
exus-am/tests/cputest/build/dummy-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 09:55:06, Apr 15 2021
For help, type "help"
(nemu) c
nemu: HIT GOOD TRAP at eip = 0x0010001b

```

## Part2

### 2.1 实验目的

1. 学习更多计算机指令
2. 学习测试与调试原理
3. 了解 `am` 相关内容

### 2.2 实验内容

1. 实现其他指令
2. 实现 `diff-test`
3. 进行回归测试

### 2.3 实验步骤

根据试验提示文档，实现 `diff-test` 有助于我们在实现其他指令时更早更便捷地发现错误，因此我们首先实现 `diff-test` 部分，再实现其他指令。

#### 2.3.1 实现 `diff-test`

实现 `diff-test` 较为简单，根据实验指导书，我们首先将 `nemu/include/common.h` 中的宏定义 `DIFF_TEST` 取消注释，之后补全 `nemu/src/monitor/diff-test/diff-test.c` 中定义的 `difftest_step` 函数。



该函数的主要功能是每执行一条指令就对比 NEMU 和 QEMU 部分寄存器中储存的值，当某一个寄存器中的值不同时就输出错误信息并将程序停下。函数中已经事先定义了用于获取 QEMU 寄存器值的变量 `r`，因此我们只需要编写对比寄存器值的内容即可，在寄存器值不同时输出信息并将用于控制程序是否暂停的 `diff` 设为 `true`。需要对比的寄存器有 `eax`, `ecx`, `edx`, `ebx`, `esp`, `ebp`, `esi`, `edi`, `eip`，因 NEMU 和 QEMU 中对 `EFLAGS` 的操作有所不同，故不考虑 `EFLAGS`，部分代码如下：

```
if(r.eax!=cpu.eax){
    printf("EIP: %x\n",cpu.eip);
    printf("EAX in QEMU: %x\n",r.eax);
    printf("EAX in NEMU: %x\n",cpu.eax);
    diff=true;
}

... ..

if(r.eip!=cpu.eip){
    printf("EIP:%x\n",cpu.eip);
    printf("EDI in QEMU:%x\n",r.eip);
    printf("EDI in NEMU:%x\n",cpu.eip);
    diff=true;
}
```

### 2.3.2 实现其他指令

实现part2中的指令时，我们可以使用 `make ARCH=x86-nemu ALL=XXX run` 指令依次运行 `nexus-am/tests/cputest` 下的每一个测试样例，在程序因为指令未实现停下时再去实现对应指令。part2中实现其他指令的具体流程与part1中相同，这里同样只说明实现指令函数的过程。

#### 2.3.2.1 arith.c

**add:** `add` 指令与 `sub` 类似，只需要将 `rtl_sub` 改成 `rtl_add` 并修改计算 `EFLAGS` 的具体值的方法即可：

```
make_EHelper(add) {
    rtl_add(&t2,&id_dest->val,&id_src->val); //计算
    operand_write(id_dest,&t2); //写回结果

    rtl_update_ZFSF(&t2,id_dest->width); //设置 ZF SF

    rtl_sltu(&t0,&t2,&id_dest->val);
    rtl_set_CF(&t0); //设置 CF

    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); //设置 OF
```

```
print_asm_template2(add);
}
```

**cmp:** `cmp` 指令的实现与 `sub` 类似，不同之处在 `cmp` 指令计算出相减结果之后不写回，而是直接根据结果操作 `EFLAGS` 寄存器：

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

    rtl_update_ZFSF(&t2, id_dest->width); //设置 ZF SF

    rtl_sltu(&t0, &id_dest->val, &t2);
    rtl_set_CF(&t0); //设置 CF

    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); //设置 OF

    print_asm_template2(cmp);
}
```

**inc dec:** `inc dec` 指令分别调用 `rtl_addi` `rtl_subi` 计算加一和减一的值，之后写回结果并修改 `EFLAGS` 寄存器：

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

    rtl_update_ZFSF(&t2, id_dest->width); //设置 ZF SF

    rtl_eqi(&t0, &t2, 0x80000000);
    rtl_set_OF(&t0); //设置 OF

    print_asm_template1(inc);
}

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

    rtl_update_ZFSF(&t2, id_dest->width); //设置 ZF SF

    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); //设置 OF
}
```

```
    print_asm_template1(dec);
}
```

**neg:** `neg` 计算一个数的负值，调用 `rtl_subi`，用 0 减去该数并写回，最后修改 `EFLAGS`：

```
make_EHelper(neg) {
    rtl_sub(&t2,&tzero,&id_dest->val); //计算
    operand_write(id_dest,&t2); //写回

    rtl_update_ZFSF(&t2,id_dest->width); //设置 ZF SF

    rtl_neq0(&t0,&id_dest->val);
    rtl_set_CF(&t0); //设置CF

    rtl_eqi(&t0,&id_dest->val,0x80000000);
    rtl_set_OF(&t0); //设置OF

    print_asm_template1(neg);
}
```

### 2.3.2.2 control.c

**call\_rm:** `call_rm` 指令与 `call` 指令类似，只是多了一步将跳转地址赋值的过程：

```
make_EHelper(call_rm) {
    rtl_li(&t2,decoding.seq_eip);
    rtl_push(&t2);
    decoding.jump_eip=id_dest->val;
    decoding.is_jump=1;

    print_asm("call %s", id_dest->str);
}
```

### 2.3.2.3 data-mov.c

**leave:** `leave` 指令反转输入指令的操作。i386手册中描述：通过将帧指针复制到堆栈指针，释放其局部变量过程使用的堆栈空间。旧的帧指针被弹出到 BP 或 EBP 中，以恢复调用者的帧。随后的 RET 指令将删除推送到退出过程的堆栈中的所有参数。实现时需要将栈底指针赋给栈顶指针并恢复上一次的栈底指针值：

```
make_EHelper(leave) {
    cpu.esp=cpu.ebp;
    rtl_pop(&t0);
    cpu.ebp=t0;
    print_asm("leave");
}
```

**cld cwtl**: **cld** 和 **cwtl** 指令类似，都是将有符号数进行拓展，只是拓展的源操作数和拓展后的储存位置以及具体的拓展方法有所不同。实现时首先要判断操作数是 16 位还是 32 位，再根据不同情况进行拓展：

```
make_EHelper(cld) {
    if (decoding.is_operand_size_16) { // 单字节
        rtl_lr_w(&t0, R_AX);
        rtl_sext(&t0, &t0, 2);
        rtl_sari(&t0, &t0, 31);
        rtl_sr_w(R_DX, &t0);
    }
    else { // 双字节
        rtl_sari(&cpu.edx, &cpu.eax, 31);
    }

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

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

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

### 2.3.2.4 logic.c

**test**: **test** 指令调用 **rtl\_and** 按位计算两个操作数，不保留计算结果，仅影响 **EFLAGS**：

```
make_EHelper(test) {
    rtl_and(&t2, &id_dest->val, &id_src->val); // 计算
    rtl_update_ZFSF(&t2, id_dest->width);
    rtl_set_CF(&tzero);
    rtl_set_OF(&tzero); // 设置 EFLAGS
    print_asm_template2(test);
}
```

**and xor or not**: 这四个指令只需要调用相应的 **rtl** 函数计算结果并写回，之后设置 **EFLAGS** 即可 (**not** 不需设置)：

```

make_EHelper(and) {
    rtl_and(&t2, &id_dest->val, &id_src->val); //计算
    operand_write(id_dest, &t2); //写回
    rtl_update_ZFSF(&t2, id_dest->width);
    rtl_set_CF(&tzero);
    rtl_set_OF(&tzero); //设置EFLAGS
    print_asm_template2(and);
}

make_EHelper(xor) {
    rtl_xor(&t2, &id_dest->val, &id_src->val); //计算
    operand_write(id_dest, &t2); //写回
    rtl_set_CF(&tzero);
    rtl_set_OF(&tzero); //设置EFLAGS
    rtl_update_ZFSF(&t2, id_dest->width);

    print_asm_template2(xor);
}

make_EHelper(or) {
    rtl_or(&t2, &id_dest->val, &id_src->val); //计算
    operand_write(id_dest, &t2); //写回
    rtl_update_ZFSF(&t2, id_dest->width);
    rtl_set_CF(&tzero);
    rtl_set_OF(&tzero); //设置EFLAGS
    print_asm_template2(or);
}

make_EHelper(not) {
    rtl_mv(&t0, &id_dest->val);
    rtl_not(&t0); //计算
    operand_write(id_dest, &t0); //写回
    print_asm_template1(not);
}

```

**sar shl shr**: 这三个指令与位运算有关，同样只需要调用相应的 rtl 函数计算结果并写回，之后设置 EFLAGS 即可，但在 NUME 中只需要设置 ZF 和 SF，可以不考虑其他 EFLAGS 位：

```

make_EHelper(sar) {
    rtl_sext(&t2, &id_dest->val, id_dest->width);
    rtl_sar(&t2, &t2, &id_src->val); //计算
    operand_write(id_dest, &t2); //写回
    rtl_update_ZFSF(&t2, id_dest->width); //设置EFLAGS

    print_asm_template2(sar);
}

make_EHelper(shl) {
    rtl_shl(&id_dest->val, &id_dest->val, &id_src->val); //计算
    operand_write(id_dest, &id_dest->val); //写回
    rtl_update_ZFSF(&id_dest->val, id_dest->width); //设置EFLAGS
}

```

```

    print_asm_template2(shl);
}

make_EHelper(shr) {
    rtl_shr(&id_dest->val,&id_dest->val,&id_src->val); //计算
    operand_write(id_dest,&id_dest->val); //写回
    rtl_update_ZFSF(&id_dest->val,id_dest->width); //设置EFLAGS

    print_asm_template2(shr);
}

```

### 2.3.2.5 注意

除了上述指令的实现之外，part2中还有一些需要注意的地方：

- 部分指令的操作码以 0F 开头，表示该指令的操作码占两个字节，对于这类指令需要在 0F 之后再向后读一个字节，之后在 `opcode_table` 第二部分的相应位置填写宏定义。
- 在实现 `jcc` 类的指令时，还需要补全 `nemu/src/cpu/exec/cc.c` 中的 `rtl_setcc` 函数，该函数查询 `EFLAGS` 来确定跳转条件是否满足，实现该函数时需要查看 i386 手册 `jcc` 部分，补充的代码如下：

```

switch (subcode & 0xe) {
    case CC_0:
        rtl_get_OF(dest);
        break;
    case CC_B:
        rtl_get_CF(dest);
        break;
    case CC_E:
        rtl_get_ZF(dest);
        break;
    case CC_BE:
        rtl_get_CF(&t1);
        rtl_get_ZF(&t2);
        rtl_or(dest, &t1, &t2);
        break;
    case CC_S:
        rtl_get_SF(dest);
        break;
    case CC_L:
        rtl_get_SF(&t0);
        rtl_get_OF(&t1);
        *dest = t0 != t1;
        break;
    case CC_LE:
        rtl_get_ZF(&t0);
        rtl_get_SF(&t1);
        rtl_get_OF(&t2);
        *dest = (t1 != t2) || (t0 == 1);
        break;
    // TODO();
}

```

```

default: panic("should not reach here");
case CC_P: panic("n86 does not have PF");
}

```

### 2.3.3 进行回归测试

回归测试是为了保证加入的新功能没有影响到已有功能的实现，因此在part2的最后我们还需要重新运行之前的测试用例，框架中提供了一个可以一键测试的脚本，按照实验指导手册在 `nemu/` 下执行 `bash runall.sh` 可以批量运行测试样例，并获得输出样式是否成功通过。如果一个测试用例运行失败，脚本将会保留相应的日志文件，我们可以通过查看该文件来确定出错的位置；当使用脚本通过这个测试用例的时候，日志文件将会被移除。测试结果如下：

```

qin2mu4@ubuntu: /mnt/hgfs/VMwareShare/ics2018/nemu
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!
qin2mu4@ubuntu: /mnt/hgfs/VMwareShare/ics2018/nemu$

```

## 2.4 问答

2.4.1 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?

**栈(stack)** 是自动分配变量，以及函数调用所使用的一些空间(所谓的局部变量)，地址由高向低减少。

**堆(heap)** 由malloc、new等分配的空间的地址，地址由低向高增长。

堆和栈中数据的变化比较频繁，如果放进可执行文件中读取速度会变慢，因此堆栈是在程序运行时从内存中动态申请的。

2.4.2 用这三种指令写一个计算两个正整数相加的程序

使用三种指令为：

```

V = V + 1
V = V - 1
IF V != 0 GOTO LABEL

```

假设要相加的两个数分别为a、b，计算结果储存在a中，则计算两数相加的程序为：

```
LOOP:  b = b - 1
        a = a + 1
        IF b != 0 GOTO LOOP
```

## Part3

### 3.1 实验目的

1. 学习计算机输入输出原理
2. 学习CPU向设备发送命令的方式
3. 了解 `am` 相关内容

### 3.2 实验内容

1. 实现文字输出
2. 实现时钟
3. 检测键盘
4. 实现图像输出

### 3.3 实验步骤

按照实验指导书，要实现part3的内容，我们需要首先定义两个宏定义。第一个在 `nemu/include/common.h` 中定义宏 `HAS_IOE`，定义后 `init_device()` 函数会对设备进行初始化，NEMU 框架中已经定义好了该宏，只需要取消注释即可。

#### 3.3.1 运行 Hello World

为运行 Hello World 我们需要实现之前为实现的 `in`、`out` 指令，并将 `nexus-am/am/arch/x86-nemu/src/trm.c` 中定义的宏 `HAS_SERIAL` 取消注释即可。

首先查阅i386手册，在 `nemu/src/cpu/exec/exec.c` 的指令译码表中对应位置填写对应的宏指定 `in`、`out` 的译码函数和操作函数，之后在 `nemu/src/cpu/exec/all-instr.h` 中声明 `in`、`out` 指令的操作函数，具体如下：

```
// nemu/src/cpu/exec/exec.c
/* 0xe4 */ IDEXW(in_I2a,in,1), IDEX(in_I2a,in), IDEXW(out_a2I,out,1),
IDEX(out_a2I,out),
... ..

/* 0xec */ IDEXW(in_dx2a,in,1), IDEX(in_dx2a,in), IDEXW(out_a2dx,out,1),
IDEX(out_a2dx, out),
```



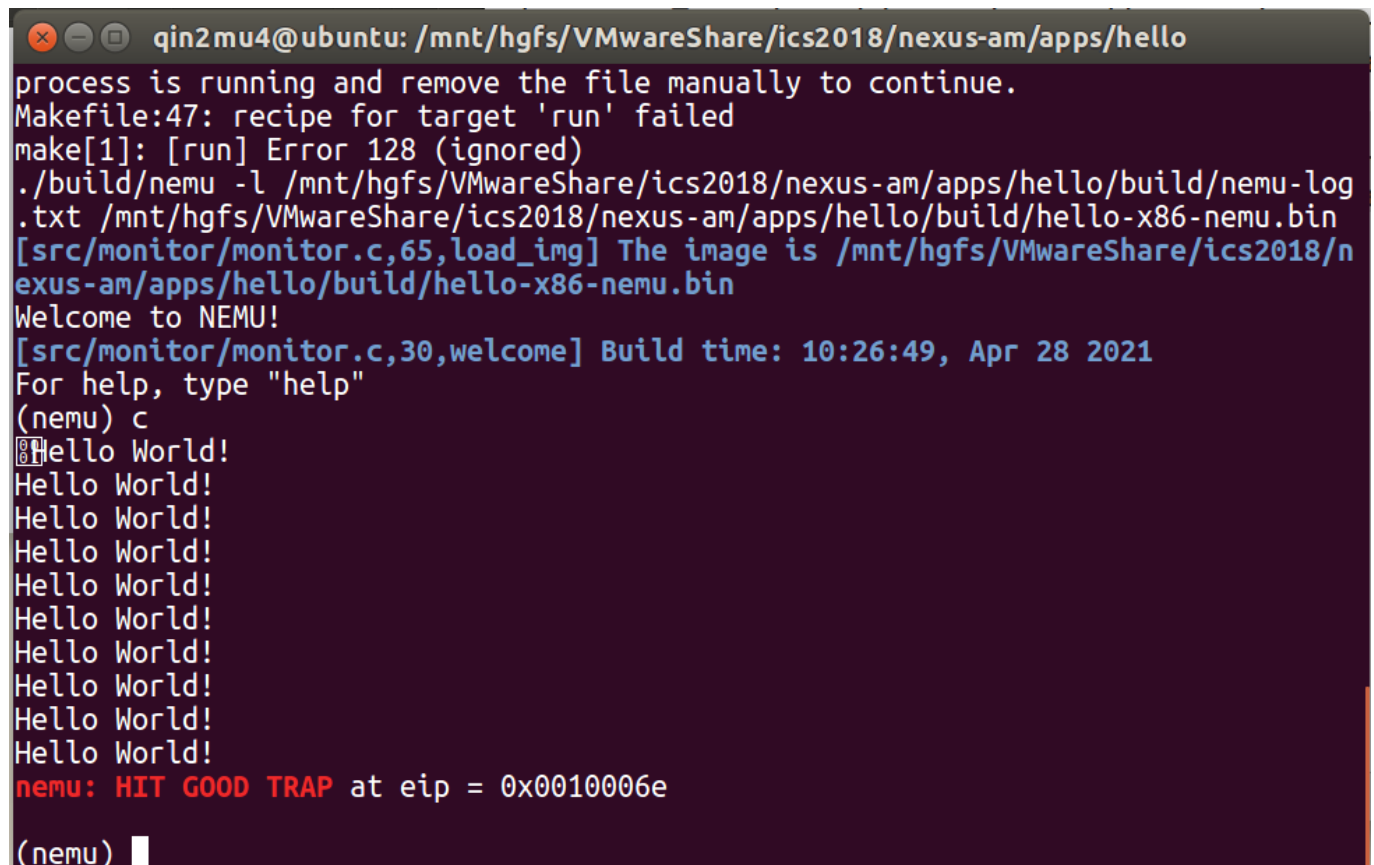
```
// nemu\src\cpu\exec\all-instr.h
make_EHelper(out); make_EHelper(in);
```

最后在 `nemu\src\cpu\exec\system.c` 中补全 `in`、`out` 的操作函数。根据实验指导书，我们可以使用 `pio_read()` 和 `pio_write()` 函数来对串口进行读写，需要注意的是在 `in` 中读完之后还要将结果写回。代码如下：

```
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
}
```

完成代码后在 `nexus-am/apps/hello` 执行 `make run`，弹出窗口后在原窗口中执行 `c` 指令，可以看到原窗口输出了十条 `Hello World` 语句。



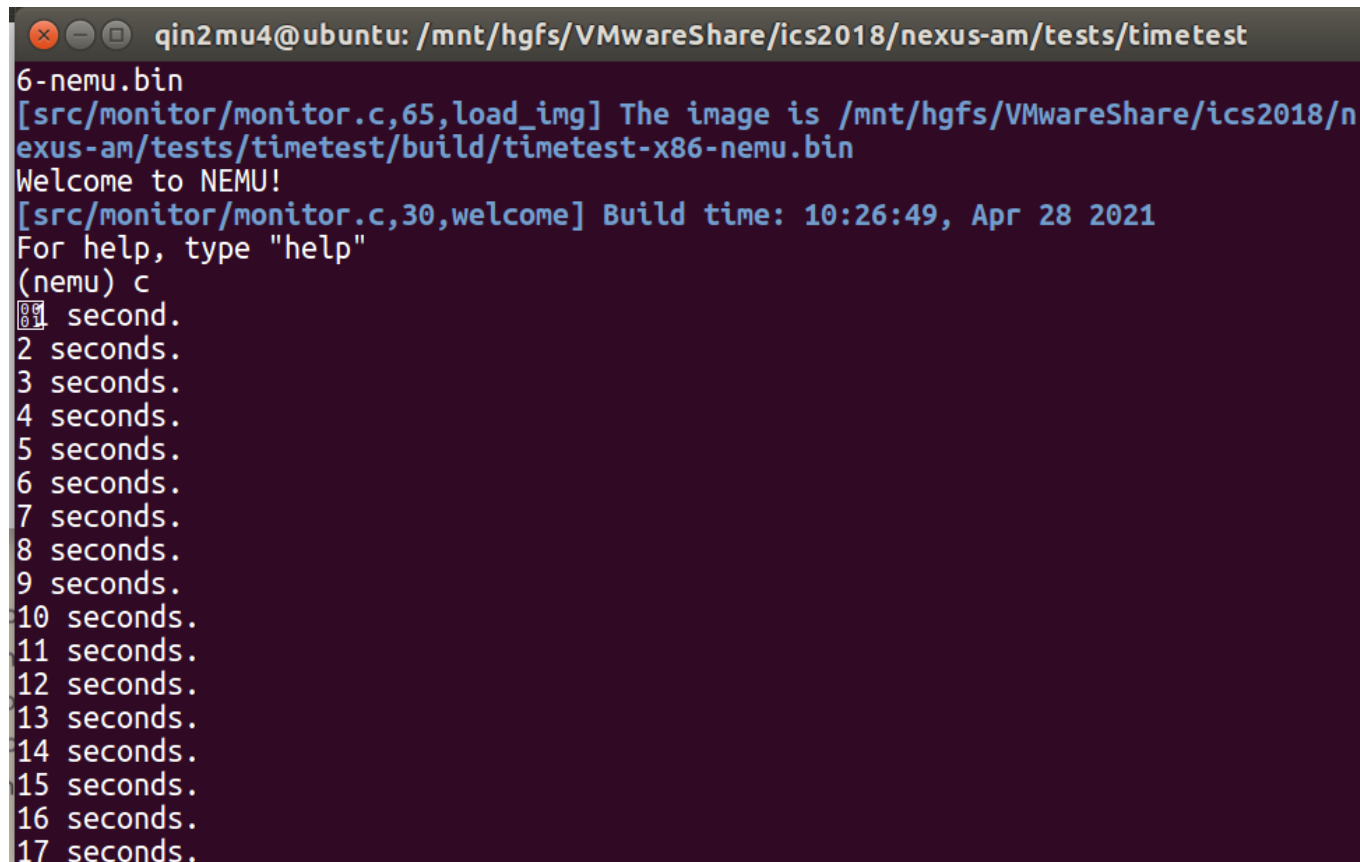
```
qin2mu4@ubuntu: /mnt/hgfs/VMwareShare/ics2018/nexus-am/apps/hello
process is running and remove the file manually to continue.
Makefile:47: recipe for target 'run' failed
make[1]: [run] Error 128 (ignored)
./build/nemu -l /mnt/hgfs/VMwareShare/ics2018/nexus-am/apps/hello/build/nemu-log.txt /mnt/hgfs/VMwareShare/ics2018/nexus-am/apps/hello/build/hello-x86-nemu.bin
[src/monitor/monitor.c,65,load_img] The image is /mnt/hgfs/VMwareShare/ics2018/nexus-am/apps/hello/build/hello-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 10:26:49, Apr 28 2021
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) █
```

### 3.3.2 实现时钟

实现时钟只需要修改 `nexus-am\am\arch\x86-nemu\src\ioe.c` 中的 `_uptime()` 函数即可，该函数用于返回系统启动后经过的毫秒数。实现该函数时，我们可以调用 `inl` 函数从 `RTC` 寄存器中获取当前时间，用获取的时间减去 `boot_time` 即可获得系统启动后经过过的毫秒数，代码如下：

```
unsigned long _uptime() {
    return inl(RTC_PORT)-boot_time;//RTC寄存器中的当前时间 - boot_time
}
```

修改函数后，在 `nexus-am/tests/timetest` 下执行 `make run` 指令，可以看到 NEMU 每隔一秒输出一段文字：



```
qin2mu4@ubuntu: /mnt/hgfs/VMwareShare/ics2018/nexus-am/tests/timetest
6-nemu.bin
[src/monitor/monitor.c,65,load_img] The image is /mnt/hgfs/VMwareShare/ics2018/nexus-am/tests/timetest/build/timetest-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 10:26:49, Apr 28 2021
For help, type "help"
(nemu) c
0.1 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.
```

实现时钟后，我们可以通过在 NEMU 中运行 `nexus-am/apps` 下的 `benchmark` 来测试程序的运行速度，测试前需要先将 `nemu/include/common.h` 中的 `DEBUG` 和 `DIFF_TEST` 宏定义注释掉，之后运行对应的 `benchmark`，结果如下，其中图3是 `microbench` 的跑分结果，图4是 `microbench` 的test测试集跑分结果：

```
[src/monitor/monitor.c,65,load_img] The image is /mnt/hgfs/VMwareShare/ics2018/nexus-am/apps/dhrystone/build/dhrystone-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 10:26:49, Apr 28 2021
For help, type "help"
(nemu) c
Dhrystone Benchmark, Version C, Version 2.2
Trying 500000 runs through Dhrystone.
Finished in 11792 ms
=====
Dhrystone PASS          87 Marks
                        vs. 100000 Marks (i7-6700 @ 3.40GHz)
nemu: HIT GOOD TRAP at eip = 0x0010006e
```

```
qin2mu4@ubuntu: /mnt/hgfs/VMwareShare/ics2018/nexus-am/apps/coremark
[src/monitor/monitor.c,65,load_img] The image is /mnt/hgfs/VMwareShare/ics2018/nexus-am/apps/coremark/build/coremark-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 10:26:49, Apr 28 2021
For help, type "help"
(nemu) c
Running CoreMark for 1000 iterations
2K performance run parameters for coremark.
CoreMark Size      : 666
Total time (ms)    : 13918
Iterations         : 1000
Compiler version   : GCC5.4.0 20160609
seedcrc           : 0xe9f5
[0]crclist        : 0xe714
[0]crcmatrix      : 0x1fd7
[0]crcstate       : 0x8e3a
[0]crcfinal       : 0xd340
Finised in 13918 ms.
=====
CoreMark PASS      321 Marks
                  vs. 100000 Marks (i7-6700 @ 3.40GHz)
nemu: HIT GOOD TRAP at eip = 0x0010006e

(nemu) █
```

```
qin2mu4@ubuntu: /mnt/hgfs/VMwareShare/ics2018/nexus-am/apps/microbench
[src/monitor/monitor.c,65,load_img] The image is /mnt/hgfs/VMwareShare/ics2018/nexus-am/apps/microbench/build/microbench-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 10:26:49, Apr 28 2021
For help, type "help"
(nemu) c
[qs] Quick sort: * Passed.
min time: 1440 ms [383]
[queen] Queen placement: * Passed.
min time: 1202 ms [429]
[bf] Brainf**k interpreter: * Passed.
min time: 7089 ms [369]
[fib] Fibonacci number: * Passed.
min time: 14427 ms [198]
[sieve] Eratosthenes sieve: * Passed.
min time: 12922 ms [328]
[15pz] A* 15-puzzle search: * Passed.
min time: 2753 ms [210]
[dinic] Dinic's maxflow algorithm: * Passed.
min time: 2139 ms [632]
[lzip] Lzip compression: * Passed.
min time: 6429 ms [411]
[ssort] Suffix sort: * Passed.
min time: 1450 ms [407]
[md5] MD5 digest: * Passed.
min time: 13588 ms [144]
=====
MicroBench PASS      351 Marks
                    vs. 100000 Marks (i7-6700 @ 3.40GHz)
nemu: HIT GOOD TRAP at eip = 0x00100032
(nemu) █
```

```

qin2mu4@ubuntu: /mnt/hgfs/VMwareShare/ics2018/nexus-am/apps/microbench
[src/monitor/monitor.c,65,load_img] The image is /mnt/hgfs/VMwareShare/ics2018/nexus-am/apps/microbench/build/microbench-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 10:26:49, Apr 28 2021
For help, type "help"
(nemu) c
[qsort] Quick sort: * Passed.
      min time: 852 ms [647]
[queen] Queen placement: * Passed.
      min time: 1292 ms [399]
[bf] Brainf**k interpreter: * Passed.
      min time: 8355 ms [313]
[fib] Fibonacci number: * Passed.
      min time: 15561 ms [183]
[sieve] Eratosthenes sieve: * Passed.
      min time: 12623 ms [335]
[15pz] A* 15-puzzle search: * Passed.
      min time: 2755 ms [210]
[dinic] Dinic's maxflow algorithm: * Passed.
      min time: 2259 ms [599]
[lzip] Lzip compression: * Passed.
      min time: 6513 ms [406]
[ssort] Suffix sort: * Passed.
      min time: 1206 ms [490]
[md5] MD5 digest: * Passed.
      min time: 12712 ms [154]
=====
MicroBench PASS          373 Marks
                        vs. 100000 Marks (i7-6700 @ 3.40GHz)
nemu: HIT GOOD TRAP at eip = 0x00100032

```

### 3.3.3 检测键盘

一般键盘的工作方式是当按下一个键时，键盘发送该键的通码；当释放一个键，时键盘发送该键的断码。为实现检测键盘的功能，我们需要实现 `nexus-am\am\arch\x86-nemu\src\ioe.c` 中的 `_read_key` 函数，首先调用 `inb` 函数检测是否有按键被按下，没有返回 `_KEY_NONE`，否则调用 `inl` 函数通过端口 I/O 访问寄存器，获得键盘码，代码如下：

```

int _read_key() {
    if(inb(0x64)&0x1)//判断是否有键按下
        return inl(0x60);//获取键盘码
    return _KEY_NONE;
}

```

实现后，运行 `nexus-am/tests/keytest` 下的 `keytest` 程序，当键盘按下或松开时会有响应信息输出。

```

qin2mu4@ubuntu: /mnt/hgfs/VMwareShare/ics2018/nexus-am/tests/keytest
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 10:26:49, Apr 28 2021
For help, type "help"
(nemu) c
Get key: 43 A down
Get key: 43 A up
Get key: 16 2 down
Get key: 16 2 up
Get key: 54 RETURN down
Get key: 54 RETURN up
Get key: 66 RSHIFT down
Get key: 66 RSHIFT up
Get key: 55 LSHIFT down
Get key: 55 LSHIFT up
Get key: 33 T down
Get key: 33 T up
Get key: 21 7 down
Get key: 21 7 up
Get key: 18 4 down
Get key: 18 4 up
Get key: 67 LCTRL down
Get key: 58 C down
Get key: 58 C up
Get key: 67 LCTRL up

```

### 3.3.4 实现图像输出

实现图像输出分为两步，首先需要修改 `nemu/src/memory/memory.c` 中的 `paddr_read` 和 `paddr_write` 函数，加入对内存映射 I/O 的判断。之后需要修改 `nexus-am/am/arch/x86-nemu/src/ioe.c` 中的 `_draw_rect` 函数，是窗口能够输出正确输出相应的动画效果。

在 `paddr_read` 和 `paddr_write` 中，我们首先调用 `is_mmio` 函数来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。当返回的是映射号时，我们调用 `mmio_read` 或 `mmio_write` 访问内存映射 I/O，否则 `pmem`。代码如下：

```

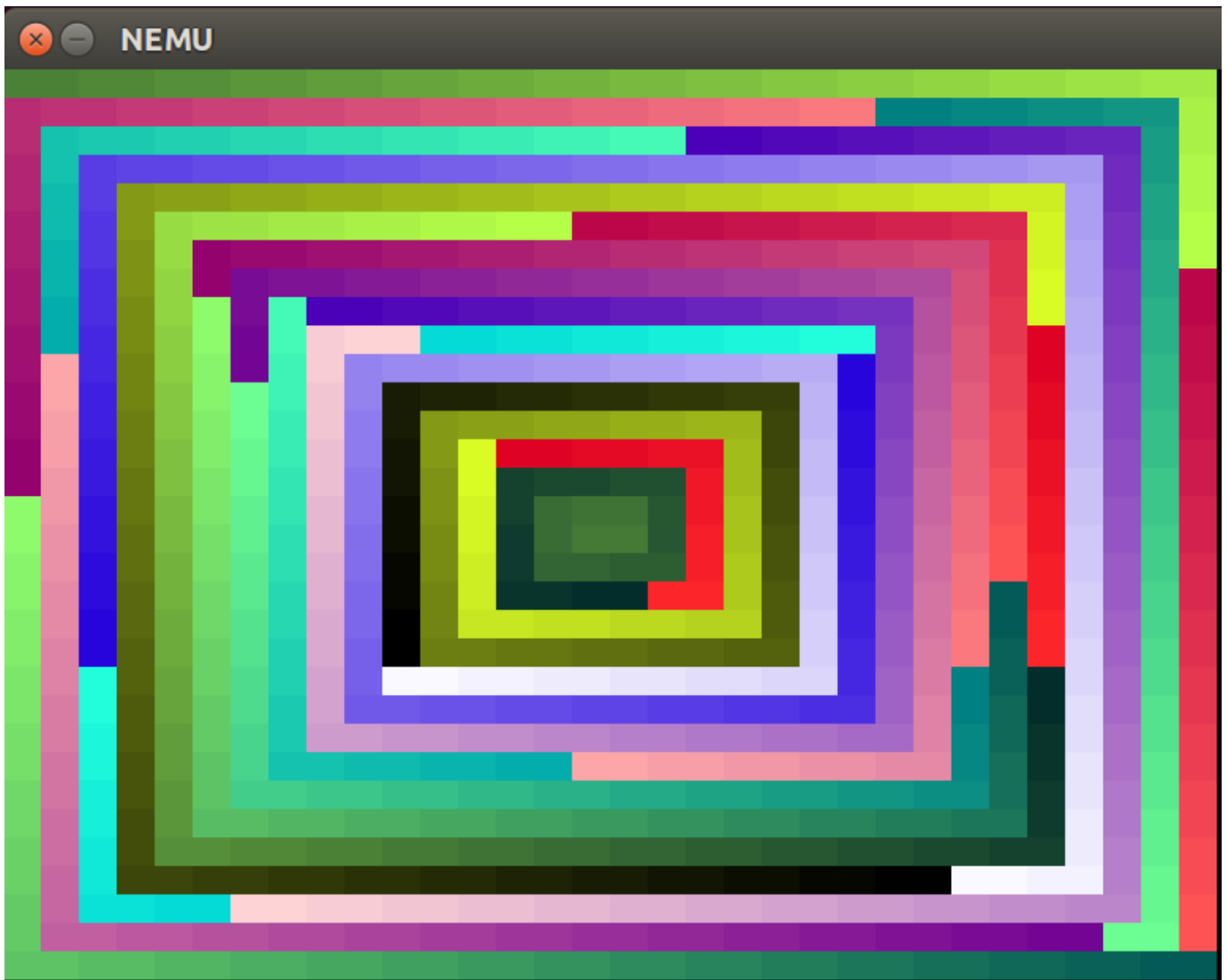
uint32_t paddr_read(paddr_t addr, int len) {
    int isPhyAdd = is_mmio(addr); // 获取内存映射号
    if (isPhyAdd != -1) {
        return mmio_read(addr, len, isPhyAdd); // 被映射到 I/O 空间
    }
    else {
        return pmem_rw(addr, uint32_t) & (~0u >> ((4 - len) << 3));
    }
}

void paddr_write(paddr_t addr, int len, uint32_t data) {}
int isPhyAdd = is_mmio(addr); // 获取内存映射号
if (isPhyAdd != -1) {
    mmio_write(addr, len, data, isPhyAdd); // 被映射到 I/O 空间
}
else {
    memcpy(guest_to_host(addr), &data, len);
}

```

```
}
}
```

添加内存映射 I/O 之后，NEMU 可以输出颜色信息：

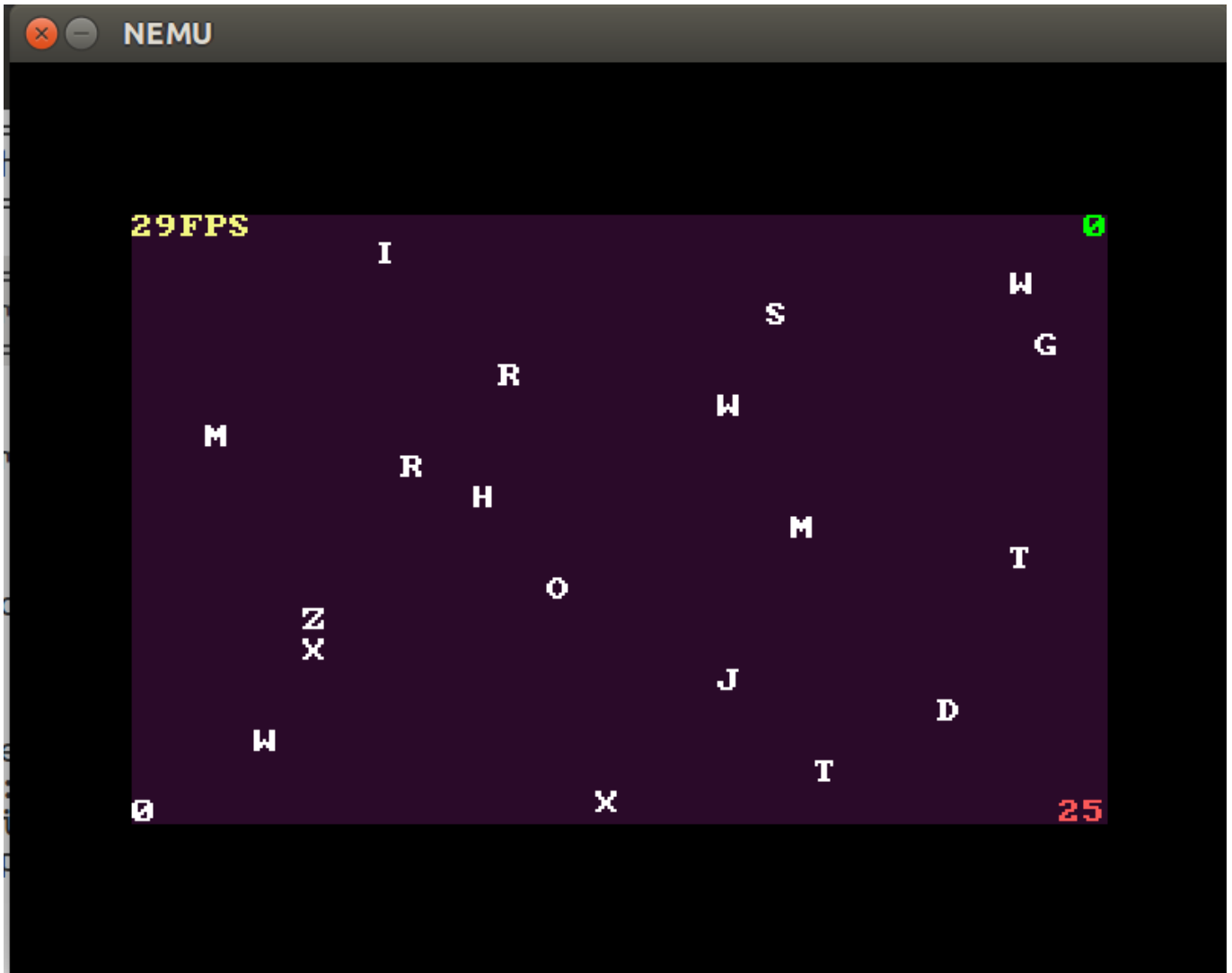


但上面的息并不是 videotest 输出的画面，因为框架代码中的 `_draw_rect` 函数并未正确实现，对于 `_draw_rect` 函数，我们应首先计算出窗口每一行的信息占用多少字节，之后在循环中按行将图像信息拷贝到从 0x40000 开始的一段用于映射到 video memory 的物理内存，代码如下：

```
void _draw_rect(const uint32_t *pixels, int x, int y, int w, int h) {
    int cp_bytes=sizeof(uint32_t) * (w<_screen.width-x?w:_screen.width-x); //一行占的空间
    for(int j=0;j<h && y+j<_screen.height;j++) { //按行拷贝
        memcpy(&fb[(y+j)*_screen.width+x],pixels,cp_bytes);
        pixels+=w;
    }
}
```

修改代码后我们可以在 `nexus-am/apps/typing` 下运行打字小游戏，还可以运行 `nexus-am/apps/litenes` 下的红白机小游戏。





## 3.4 问答

### 3.4.1 volatile 关键字

#### 3.4.1.1 代码优化不是一件好事情吗？为什么会有 volatile？

volatile 告诉编译器变量 *i* 是随时可能发生变化的，每次使用 *i* 必须从 *i* 的地址中读取，因而编译器生成的可执行码会重新从 *i* 的地址读取数据放在 *k* 中。

而优化的做法是，由于编译器发现两次从 *i* 读数据的代码之间的代码没有对 *i* 进行过操作，它会自动把上次读的数据放在 *k* 中。而不是重新从 *i* 里面读。这样以来，如果 *i* 是一个寄存器变量或者表示一个端口数据就容易出错，所以 volatile 可以保证对特殊地址的稳定访问，不会出错。

#### 3.4.1.2 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？

如果代码中的地址被映射到设备寄存器中，且对应于该地址的变量 *i* 未使用 volatile 修饰，当两次从 *i* 读数据的代码之间的代码没有对 *i* 进行过操作，编译器会直接读缓存或寄存器中的数据，而不是重新从磁盘读取，如果在两次读取之间，某设备修改了 *i* 的值，那么寄存器继续从缓存中读取就会获得错误的 *i*，造成程序出现 bug。

### 3.4.2 如何检测多个键同时被按下



猜测可以通过以下方法：每当用户按下一个方向键，就把它的键盘码存进数组（数组里不可以有重复）；每当一个方向键弹起，则把键盘码从数组里清除。如果数组中储存的两个键有相反的效果，则只响应先按的那个。

### 3.4.3 渐出渐入效果如何通过调色板实现

调色板实际上是一个有256个表项的RGB颜色表，颜色表的每项是一个24位的RGB颜色值。使用调色板时，在内存中存储的不是的24位颜色值，而是调色板的4位或8位的索引。这样一来，显示器可同时显示的颜色被限制在256色以内，对系统资源的耗费大大降低。要实现渐入渐出效果，猜测可以将将要显示的内容对应的颜色信息与背景颜色信息在加上不同深度的白色进行特定的运算，计算出三种颜色叠加后的颜色在调色版中的索引号，并显示叠加的颜色，每当要显示的内容的透明度改变就改变白色深度重新计算一次，在渐入渐出过程中进行多次计算以实现效果。

## 4 必答题

4.1 在 `nemu/include/cpu/rtl.h` 中，你会看到由 `static inline` 开头的各种 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？

**inline**：函数被调用时，需要出栈入栈。当函数频繁被调用时，则不断地有函数出栈入栈，会造成栈空间或栈内存的大量消耗。所以引入了 `inline`。在函数声明或定义中函数返回类型前加上关键字 `inline`，即可以把函数指定为内联函数。内联函数会建议编译器对一些特殊函数进行内联扩展，也就是建议编译器将指定的函数体插入并取代每一处调用该函数的地方，从而节省了每次调用函数带来的额外时间开支，有些类似于宏。但 `inline` 函数仅仅是一个对编译器的建议，最后能否真正内联，还要看编译器，并不是说声明了内联就会内联，声明内联仅仅只是一个建议而已。

**static**：在函数的返回类型前加上 `static`，函数会成为静态函数，静态函数只能在声明它的文件中可见，其他文件不能引用该函数，同时不同的文件可以使用相同名字的静态函数，互不影响。

删除 `inline`：

```
qin2mu4@ubuntu:/mnt/hgfs/VMwareShare/ics2018/nemu$ make
+ CC src/cpu/decode/decode.c
In file included from ./include/cpu/decode.h:6:0,
                  from ./include/cpu/exec.h:9,
                  from src/cpu/decode/decode.c:1:
./include/cpu/rtl.h:207:13: warning: 'rtl_update_ZFSF' defined but not used [-Wunused-function]
  static void rtl_update_ZFSF(const rtlreg_t* result, int width) {
                  ^
+ CC src/cpu/decode/modrm.c
In file included from ./include/cpu/decode.h:6:0,
                  from ./include/cpu/exec.h:9,
                  from src/cpu/decode/modrm.c:1:
./include/cpu/rtl.h:207:13: warning: 'rtl_update_ZFSF' defined but not used [-Wunused-function]
  static void rtl_update_ZFSF(const rtlreg_t* result, int width) {
                  ^
+ CC src/cpu/exec/arith.c
+ CC src/cpu/exec/cc.c
In file included from src/cpu/exec/cc.c:1:0:
```

删除 `inline` 后会报未调用的警告，猜测可能是因为每一个调用该函数的文件中都包含了 `rtl.h`，故相当于每一个调用该函数的文件中都重新定义了该函数，同时因为由 `static` 修饰的函数对于其他文件不可见，故不会出现重定义的错误，但每一个文件中调用的都是在本文件中重新定义的该函数，则最以开始在 `rtl.h` 中定义的函数就变成了未被调用的。

**删除 `static`**： `inline` 函数是不能像传统的函数那样放在 `.c` 中然后在 `.h` 中给出接口在其余文件中调用的，因为 `inline` 函数跟宏定义类似，不存在所谓的函数入口。因此会出现一个问题，就是说如果 `inline` 函数在两个不同的文件中出现，即一个 `.h` 被两个不同的文件包含，则会出现重名，链接失败的问题。而使用 `static` 修饰符会使函数仅在文件内部可见，不会污染命名空间。可以理解为 `static` 使一个 `inline` 函数在不同的 `.C` 里面生成了不同的实例，而且名字是完全相同的。

`inline` 关键字实际上仅是建议内联并不强制内联，没有 `static`，编译器认他是全局的，因此会像普通函数一样编译，所以在头文件中用 `inline` 时务必加入 `static`，否则当 `inline` 不内联时就和普通函数在头文件中定义一样，当多个 `.c` 文件包含时就会重定义。

但在实际 NEMU 实验中仅删除 `static`，编译器并未出现重定义报错，仅在同时删除 `static` 和 `inline` 时出现重定义报错（如下图），推测可能与不同的编译器特性不同有关：

```
qin2mu4@ubuntu: /mnt/hgfs/VMwareShare/ics2018/nemu
make: [build/nemu] Error 128 (ignored)
+ LD build/nemu
build/obj/cpu/decode/modrm.o: 在函数‘rtl_update_ZF’中:
modrm.c:(.text+0x0): `rtl_update_ZF'被多次定义
build/obj/cpu/decode/decode.o:decode.c:(.text+0x0): 第一次在此定义
build/obj/cpu/exec/arith.o: 在函数‘rtl_update_ZF’中:
arith.c:(.text+0x0): `rtl_update_ZF'被多次定义
build/obj/cpu/decode/decode.o:decode.c:(.text+0x0): 第一次在此定义
build/obj/cpu/exec/cc.o: 在函数‘rtl_update_ZF’中:
cc.c:(.text+0x0): `rtl_update_ZF'被多次定义
build/obj/cpu/decode/decode.o:decode.c:(.text+0x0): 第一次在此定义
build/obj/cpu/exec/control.o: 在函数‘rtl_update_ZF’中:
control.c:(.text+0x0): `rtl_update_ZF'被多次定义
build/obj/cpu/decode/decode.o:decode.c:(.text+0x0): 第一次在此定义
build/obj/cpu/exec/data-mov.o: 在函数‘rtl_update_ZF’中:
data-mov.c:(.text+0x0): `rtl_update_ZF'被多次定义
build/obj/cpu/decode/decode.o:decode.c:(.text+0x0): 第一次在此定义
build/obj/cpu/exec/exec.o: 在函数‘rtl_update_ZF’中:
exec.c:(.text+0x250): `rtl_update_ZF'被多次定义
build/obj/cpu/decode/decode.o:decode.c:(.text+0x0): 第一次在此定义
build/obj/cpu/exec/logic.o: 在函数‘rtl_update_ZF’中:
logic.c:(.text+0x0): `rtl_update_ZF'被多次定义
build/obj/cpu/decode/decode.o:decode.c:(.text+0x0): 第一次在此定义
build/obj/cpu/exec/prefix.o: 在函数‘rtl_update_ZF’中:
prefix.c:(.text+0x0): `rtl_update_ZF'被多次定义
build/obj/cpu/decode/decode.o:decode.c:(.text+0x0): 第一次在此定义
build/obj/cpu/exec/special.o: 在函数‘rtl_update_ZF’中:
special.c:(.text+0x0): `rtl_update_ZF'被多次定义
build/obj/cpu/decode/decode.o:decode.c:(.text+0x0): 第一次在此定义
```

## 4.2 了解 Makefile

### 4.2.1 Makefile 简介

Makefile 可以简单的认为是一个工程文件的编译规则，描述了整个工程的编译和链接等规则。其中包含了那些文件需要编译，那些文件不需要编译，那些文件需要先编译，那些文件需要后编译，那些文件需要重建等等。编译整个工程需要涉及到的，在 Makefile 中都可以进行描述。换句话说，Makefile 可以使得我们的项目工程的编译变得自动化，不需要每次都手动输入一堆源文件和参数。Makefile 长与 make 指令一同使用。

## 4.2.2 make 指令

make 会一层又一层地去找文件的依赖关系，直到最终编译出第一个目标文件。在找寻的过程中，如果出现错误，比如最后被依赖的文件找不到，那么make就会直接退出，并报错，而对于所定义的命令的错误，或是编译不成功，make 不会特别处理。make 只管文件的依赖性，即如果在找了依赖关系之后，冒号后面的文件还是不在，那么 make 会直接退出。

## 4.2.3 make 程序如何组织.c 和.h 文件，最终生成可执行文件

### 4.2.3.1 Makefile 的工作方式

**Makefile 中的变量：**在 Makefile 中的定义的变量，就像是C/C++语言中的宏一样，他代表了一个文本字符串，在 Makefile 中执行的时候其会自动原模原样地展开在所使用的地方。其与 C/C++ 所不同的是，可以在 Makefile 中改变其值。在 Makefile 中，变量可以使用在“目标”，“依赖目标”，“命令”或是 Makefile 的其它部分中。用户将编译器及编译选项定义为变量，这样做增强了makefile文件的灵活性。

**Makefile 中的 implicit rules：**implicit rules 即隐含规则，它能够告诉 make 怎样使用传统的技术完成任务，这样，当用户使用时就不必详细指定编译的具体细节，而只需把目标文件列出即可。Make 会自动搜索隐式规则目录来确定如何生成目标文件。

**Makefile 的工作方式：**当只输入 make 指令时

1. 它会在当前目录下找名字叫 Makefile 或 makefile 的文件
2. 如果找到，会找文件中的第一个目标文件（target），如目标文件叫 example，它会找到该文件，并把这个文件作为最终的目标文件。
3. 如果 example 文件不存在，或是 example 所依赖的后面的 .o 文件的文件修改时间要比 example 这个文件新，那么，他就会执行后面所定义的命令来生成 example 这个文件。
4. 如果 example 所依赖的 .o 文件也存在，那么 make 会在当前文件中找目标为 .o 文件的依赖性，如果找到则再根据那一个规则生成 .o 文件。
5. 当C文件和H文件都存在时，make 会生成 .o 文件，然后再用 .o 文件声明 make 的终极任务，即生成文件 example。

Makefile 是一个文本文件，用于描述程序源代码之间以及程序可执行代码与源代码之间的依赖关系。例如：最终编译生成的可执行文件 ab.out 是由 a.c 和 b.c 共同编译生成的，那么 Makefile 文件就要写：

```
ab.out: a.c b.c
    gcc a.c b.c -o ab.out
```

其中第一行描述了依赖关系，第二行描述了依赖关系是如何达成的。如果最终编译生成的可执行文件 ab.out 是由 a.c 和 b.o 功能编译生成的，而 b.o 是由 b1.c 和 b2.c 编译而成，那么 Makefile 文件就要写：

```
ab.out: a.c b.o
        gcc a.c b.o -o ab.out

b.o:    b1.c b2.c
        gcc -c b1.c b2.c -o b.o
```

#### 4.2.3.2 编译链接的过程

程序要运行起来，必须要经过四个步骤：预处理、编译、汇编和链接。

**编译：**编译过程是整个程序构建的核心部分，编译成功，会将源代码由文本形式转换成机器语言，编译过程就是把预处理完的文件进行一系列词法分析、语法分析、语义分析以及优化后生成相应的汇编代码文件。

其中**词法分析**是按照一定的词法规则将输入的字符串分割成一个个记号，然后他们放到对应的表中。产生的记号一般分为：关键字、标识符、字面量（包含数字、字符串等）和特殊符号（运算符、等号等）。**语法分析**会根据给定的语法规则，将词法分析产生的记号序列进行解析，然后将它们构成一棵语法树。对于不同的语言，其语法规则不同。对于**语义分析**，编译器能分析的语义只有静态语义，通常包括声明与类型的匹配、类型的转换等。最后编译器会**生成中间代码**，并对中间代码进行**优化**，优化后生成**目标代码**。

**链接：**链接的主要内容就是将各个模块之间相互引用的部分正确的衔接起来。链接过程主要是因为由汇编程序生成的目标文件并不能立即就被执行，其中可能还有许多没有解决的问题。例如，某个源文件中的函数可能引用了另一个源文件中定义的某个符号（如变量或者函数调用等）；在程序中可能调用了某个库文件中的函数，等等。所有的这些问题，都需要经链接程序的处理方能得以解决。链接程序的主要工作就是将有关的目标文件彼此相连接，也即将在一个文件中引用的符号同该符号在另外一个文件中的定义连接起来，使得所有的这些目标文件成为一个能够按操作系统装入执行的统一整体。链接过程主要包括了地址和空间分配、符号决议和重定向。最基本的链接叫做静态链接，就是将每个模块的源代码文件编译成目标文件，然后将目标文件和库一起链接形成最后的可执行文件。库其实就是一组目标文件的包，就是一些最常用的代码变异成目标文件后打包存放。最常见的库就是运行时库，它是支持程序运行的基本函数的集合。

链接中**符号决议**有时候也被叫做符号绑定，是指用符号来去标识一个地址，如 `int a = 6;` 中，用 `a` 来标识一个块4个字节大小的空间，空间里边存放的内容就是4。**重定位**是指重新计算各个目标的地址过程。

#### 4.3.3 make 程序如何组织 .c 和 .h 文件，最终生成可执行文件

根据以上的知识，我们可以推导一下 make 程序组织 .c 和 .h 文件并生成可执行文件的过程。以上面提到的一个简单 Makefile 文件为例：

```
# 可执行文件 ab.out 由 a.c 和 b.o 功能编译生成的， b.o 由 b1.c 和 b2.c 编译而成
ab.out: a.c b.o
        gcc a.c b.o -o ab.out

b.o:    b1.c b1.h
        gcc -c b1.c b1.h -o b.o
```

执行 make 命令后，它首先在当前目录下寻找名字叫 Makefile 或 makefile 的文件，找到后在文件中查找第一个目标文件，即 `ab.out`，并将该文件当做最终目标文件。如果查找第一个目标文件 `ab.out` 未找到，或该文件之后的 `b.o` 文件的文件修改时间比 `ab.out` 新，那么 make 就需要执行 `gcc a.c b.o -o ab.out` 命令生成 `ab.out`



，但在执行前，make 还需要检查 b.o 的依赖性是否存在，在本例中 b.o 依赖 b1.c 和 b1.h，因此 make 会检查 b1.c 和 b1.h 是否存在以及他们的修改时间是否新于 b.o，如果是则需要执行 `gcc -c b1.c b1.h -o b.o` 命令生成 b.o，在 b.o 生成之后再回去执行生成 ab.out 的命令。同样，如果在一开始查找到 ab.out 后查找其依赖的 b.o 不存在，也需要执行 b.o 的依赖。除此之外，如果 make 在执行过程中，发现 a.c 或 b1.c b1.h 不存在，则 make 会直接停止。

上述就是 make 程序组织 .c 和 .h 文件生成可执行文件的过程，过程中执行的 gcc 指令会涉及到编译链接等一些列操作，这些操作已在 4.2.3.2 中阐述。

---

## 5 遇到的问题和解决方法

---

**问题 1：**pa2 刚开始时完全没有思路，不知道怎么实现一条指令。

**解决方法：**在网上找到一篇大神写的[博客](#)，详细介绍了实现流程，看完才明白该怎么做，感谢大神。

**问题 2：**part3 开始时实现输出 Hello World，一直不成功。

**解决方法：**起初又读了几次文档，还是没有进展，后来把这部分的代码删了又写了一次，发现可能是在调用函数的时候参数传的有问题。