PA2实验报告-1711346-李明旗

目录

1 概述

1.1 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

1.2 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 中运行第一个 C 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。

第三阶段，学习 IO 原理，简单实现 CPU 对输入输出设备的控制

2 阶段一

2.1 代码： 运行第一个 C 程序

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

会报错：

```
Makefile:1: /Makefile.check: No such file or directory
make: *** No rule to make target '/Makefile.check'.  Stop.
```

查看 ics2018/nexus-am/README.md，得知需要对环境变量进行修改。在 terminal 中输入 cd ， 输入 vim.bashrc ， 在文件末尾加入

```
export NEMU_HOME=/home/qiqi/ics2018/nemu
export AM_HOME=/home/qiqi/ics2018/nexus-am
export NAVY_HOME=/home/qiqi/ics2018/navy-apps
```

编译运行：

```
(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)
```

2.11 指令系统的实现要点

然后需要开始在NEMU中添加指令了。要实现哪些指令才 能让dummy在NEMU中运行起来呢? 答案就在其反汇编结果(nexus-am/tests/cputest/build/dummy-x86-nemu.txt)中。查看反汇编结果,发现只需要添加 call,push,sub,xor,pop,ret 六条指令就可以了。每一条指令还有不同的形式，根据 KISS 法则，可以先实现只在 dummy 中出现的指令形式，通过指令的 opcode 可以确定具体的形式。

2.12 代码实现

在文件 nemu/src/cpu/exec/all-instr.h中进行函数声明：

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

由于Call 指令的实现中需要使用压栈操作，这里先实现 push 与pop 指令。

PUSH

make_EHelper(push)需先实现 rtl_push 函数：修改栈顶，并将指针 src1 中的内容 写入栈。

nemu/include/cpu/rtl.h

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

实现 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 opcode_table，根据上文，opcode为 0x50-0x57，译码函数为 make_DHelper(r)，执行函数为 make_EHelper(push)，定义为 IDEX(r,push)。

nemu/src/cpu/exec/exec.c

```
/* 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),
```

POP

和Push相似，实现 rtl_pop 函数：从栈中读取数据并存放在 dest 中。

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

实现 make_EHelper(pop)：将 rtl_pop 读取的数据写入到通用寄存器中。

```
make_EHelper(pop) {
  rtl_pop(&t0);
  operand_write(id_dest,&t0);
  print_asm_template1(pop);
}
```

填写 opcode_table，根据上文，opcode 为 0x58-0x5F （其中不包含 0x5C ），译码函数为 make_DHelper(r)，执行函数为 make_EHelper(pop)，故定义为 IDEX(r,pop)。

```
/* 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),
```

CALL

译码函数：make_DHelper(J) 操作数仅一个，为立即数。CPU 的跳转目标地址=当前 eip+立即数 offset（可 正可负）。 查 看 decode/decode.c， 找 到 对 应 的 译 码 函 数 make_DHelper(J)：make_DHelper(J)调用decode_op_SI 函数实现立即数的读取，并更新jmp_eip。

补充代码： decode/decode.c

```
    t0 = instr_fetch(eip,op->width);
    rtl_sext(&t0,&t0,op->width);
    op->simm = t0;

    rtl_li(&op->val, op->simm);
```

执行函数：make_EHelper(call)，根据 i386 手册知，call 指令需先将 eip 压栈，再进行跳转。

根据 call 功能可知该函数在 exe/control.c 中实现。

```
// eip压栈 设置跳转目标地址 设置跳转标记
rtl_push(eip);
rtl_addi(&decoding.jmp_eip,eip,id_dest->val);/
decoding.is_jmp = 1;
print_asm("call %x", decoding.jmp_eip);
```

填写 opcode_table：

```
/* 0xb8 */    IDEX(mov_I2r, mov), IDEX(mov_I2r, mov), IDEX(mov_I2r, mov), IDEX(mov_I2r, mov),
/* 0xbc */    IDEX(mov_I2r, mov), IDEX(mov_I2r, mov), IDEX(mov_I2r, mov), IDEX(mov_I2r, mov),
```

SUB

eflags 寄存器的定义 :nemu/include/cpu/reg.h

```
struct{
    uint32_t CF:1;
    unsigned:5;
    uint32_t ZF:1;
    uint32_t SF:1;
    unsigned:1;
    uint32_t IF:1;
    unsigned:1;
    uint32_t OF:1;
    unsigned:20;
}; //这个结构占用32bits
}eflags;//32bits寄存器
```

eflags 的初始化 :nemu/src/monitor/monitor.c 修改 restart()函数，

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

  //eflags 设置初值
  cpu.eflags.val = 0x00000002;
  //cs 设置初值
  cpu.cs = 0x00000008;
  //cr0 设置初值
  cpu.cr0.val = 0x60000011;
#ifdef DIFF_TEST
  //设置到和nemu相同
  init_qemu_reg();
#endif
}
```

实现相关的 RTL指令

EFLAGS 寄存器的标志位读写函数

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

EFLAGS 寄存器的标志位更新函数

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

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

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

static inline void rtl_msb(rtlreg_t* dest, const rtlreg_t* src1, int width) {
  // dest <- src1[width * 8 - 1]
  // 返回最高有效位，即标志位
  rtl_shri(dest,src1,width*8-1);
}

static inline void rtl_update_ZF(const rtlreg_t* result, int width) {
  // eflags.ZF <- is_zero(result[width * 8 - 1 .. 0])

  t1=(*result & (~0u >> ((4 - width) << 3)));
  rtl_eq0(&t1,&t1);
  rtl_set_ZF(&t1);
}
```

```
static inline void rtl_update_ZF(const rtlreg_t* result, int width) {
  // eflags.ZF <- is_zero(result[width * 8 - 1 .. 0])

  t1=(*result & (~0u >> ((4 - width) << 3)));
  rtl_eq0(&t1,&t1);
  rtl_set_ZF(&t1);
}

static inline void rtl_update_SF(const rtlreg_t* result, int width) {
  // eflags.SF <- is_sign(result[width * 8 - 1 .. 0])
  rtl_msb(&t1,result,width);
  rtl_set_SF(&t1);
}
```

定义函数 eflags_modify()：计算减法并相应地设置 eflags 寄存器的值。

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

  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);
}
```

XOR

实现 make_Ehelper(xor)： logic.c

```
make_EHelper(xor) {
  rtl_xor(&t2,&id_dest->val,&id_src->val);
  operand_write(id_dest,&t2);

  rtl_update_ZFSF(&t2,id_dest->width);
  rtl_set_CF(&tzero);
  rtl_set_OF(&tzero);

  print_asm_template2(xor);
}
```

填写 opcode_table

```
/* 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,
```

RET

```
make_EHelper(ret) {
  rtl_pop(&decoding.jmp_eip);
  decoding.is_jmp = 1;
  print_asm("ret");
}
```

填写 opcode_table

```
/* 0xc0 */    IDEXW(gp2_Ib2E, gp2, 1), IDEX(gp2_Ib2E, gp2), EMPTY, EX(ret), //ret没有操作数无须译码函数
```

最终运行结果：

```
For help, type "help"
(nemu) c
nemu: HIT GOOD TRAP at eip = 0x00100026

(nemu)
```

3 阶段二 程序运行时环境与AM

3.1 代码：运行更多的程序

与其他文件的测试类似，指令系统的具体实现流程与阶段一类似，先在 exec/all-instr.h 中声明各 make_EHelper：

```
make_EHelper(lea);
make_EHelper(jcc);
make_EHelper(nop);
make_EHelper(cmp);
make_EHelper(setcc);
make_EHelper(imul2);
make_EHelper(movzx);
make_EHelper(movsx);
make_EHelper(test);
make_EHelper(adc);
make_EHelper(or);
make_EHelper(sar);
make_EHelper(shr);
make_EHelper(shl);
make_EHelper(dec);
make_EHelper(mul);
make_EHelper(div);
make_EHelper(idiv);
make_EHelper(imul1);
make_EHelper(neg);
make_EHelper(not);
make_EHelper(inc);
make_EHelper(in);
make_EHelper(out);
make_EHelper(jmp);
make_EHelper(jmp_rm);
make_EHelper(cwtl);
make_EHelper(cltd);
make_EHelper(call_rm);
make_EHelper(leave);
make_EHelper(sbb);
make_EHelper(rol);
make_EHelper(lidt);
make_EHelper(int);
make_EHelper(pusha);
make_EHelper(popa);
```

AND

执行函数：

```
make_EHelper(add) {
  rtl_add(&t2, &id_dest->val, &id_src->val);
  rtl_sltu(&t3, &t2, &id_dest->val);
  |
  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(add);
}
```

填写 opcode：

```
/* 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、PUSHL

查阅相关资料，得知反汇编码 pushl 相当于 pushdword。根据反汇编指令，对应的 opcode 如下：

```
/* 0x68 */    IDEX(push_SI,push), EMPTY, IDEXW(push_SI,push,1), EMPTY,
```

SETcc

SETcc 指令（cc 为 conditioncode 的缩写），不是指一条指令，而是指一系列 形如 SETcc的指令，如：SETNE、SETE、SETNA、SETA、SETNB、SETB 等等。

这里根据需要先实现 exec.c

```
/* 0x90 */    IDEXW(E,setcc,1), IDEXW(E,setcc,1), IDEXW(E,setcc,1), IDEXW(E,setcc,1),
/* 0x94 */    IDEXW(E,setcc,1), IDEXW(E,setcc,1), IDEXW(E,setcc,1), IDEXW(E,setcc,1),
/* 0x98 */    IDEXW(E,setcc,1), IDEXW(E,setcc,1), IDEXW(E,setcc,1), IDEXW(E,setcc,1),
```

此外，还需实现 cc.c 中的 rtl_setcc 函数：

```
// TODO: Query EFLAGS to determine whether the condition code is satisfied.
// dest <- ( cc is satisfied ? 1 : 0)
switch (subcode & 0xe) {
  case CC_O:  //0
    *dest = cpu.eflags.OF;
    break;
  case CC_B:  //2
    *dest = cpu.eflags.CF;
    break;
  case CC_E:  //4
    *dest = cpu.eflags.ZF;
    break;
  case CC_BE: //6
    *dest = ((cpu.eflags.CF) || (cpu.eflags.ZF));
    break;
  case CC_S:  //8
    *dest = cpu.eflags.SF;
    break;
  case CC_L:  //12 c
    *dest = (cpu.eflags.SF != cpu.eflags.OF);
    break;
  case CC_LE: //14 e
    *dest = ((cpu.eflags.ZF) || (cpu.eflags.SF != cpu.eflags.OF));
    break;
  default: panic("should not reach here");
  case CC_P: panic("n86 does not have PF");
```

MOVBX，MOVSX

/2byte_opcode_table/

```
/* 0xa4 */    EMPTY, EMPTY, EMPTY, EMPTY,
/* 0xa8 */    EMPTY, EMPTY, EMPTY, EMPTY,
/* 0xac */    EMPTY, EMPTY, EMPTY, IDEX(E2G,imul2),
```

JCC

jcc 指令的形式非常多，且分为单字节 opcode 与双字节 opcode，

/1byte_opcode_table/

```
/* 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),
```

/2byte_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), IDEX(J,jcc), IDEX(J,jcc),
/* 0x8c */    IDEX(J,jcc), IDEX(J,jcc), IDEX(J,jcc), IDEX(J,jcc),
```

SAR/SAL/SHL/SHR

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）。

logic.c添加代码

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

  print_asm_template2(sar);
}

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

  print_asm_template2(shl);
}

make_EHelper(shr) {
  rtl_shr(&t0,&id_dest->val,&id_src->val);
  operand_write(id_dest,&t0);
  rtl_update_ZFSF(&id_dest->val,id_dest->width);
  // unnecessary to update CF and OF in NEMU

  print_asm_template2(shr);
}
```

rtl.h添加代码

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

static inline void rtl_not(rtlreg_t* dest) {
  // dest <- ~dest
  *dest = ~(*dest);
}


static inline void rtl_sext(rtlreg_t* dest, const rtlreg_t* src1, int width) {
  // dest <- signext(src1[(width * 8 - 1) .. 0])
  // 实现符号扩展，先逻辑左移再算数右移
  rtl_li(&t1,32-width*8);
  rtl_shl(dest,src1,&t1);
  rtl_sar(dest,dest,&t1);
}
```

exec.h添加代码

```
make_group(gp2,
    EX(rol), EMPTY, EMPTY, EMPTY,
    EXW(shl,1), EX(shr), EMPTY, EXW(sar,1))
```

TEST

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

logic.c添加代码

```
  rtl_and(&t0,&id_dest->val,&id_src->val);
  rtl_update_ZFSF(&t0,id_dest->width);
  rtl_set_CF(&tzero);
  rtl_set_OF(&tzero);
  print_asm_template2(test);
```

exec.c添加代码

```
make_group(gp3,
    IDEX(test_I,test), EMPTY, EX(not), EX(neg),
    EX(mul), EX(imul1), EX(div), EX(idiv))
```

ADD

arith.c添加代码

```
make_EHelper(add) {
  rtl_add(&t2, &id_dest->val, &id_src->val);
  rtl_sltu(&t3, &t2, &id_dest->val);

  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(add);
}
```

exec.c添加代码

```
/* 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,
```

CMP

arith.c添加代码

```
make_EHelper(cmp) {
  //rtl_sext(&id_src->val,&id_src->val,id_src->width);

  rtl_sub(&t2, &id_dest->val, &id_src->val);
  rtl_sltu(&t3, &id_dest->val, &t2);
  // ---
  //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(&t3); //before: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);
}
```

exec.c添加代码

```
/* 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,MUL,IMUL,DIV,IDIV,ADC,NEG,OR,DEC,INC,CLTD,LEAVE和上面指令都很相似，均一一实现

CALL补充

exec.c

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

control.c

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

与第一阶段实现的 callorel8 的区别在于 jmp_eip 的计算方法，使用另外的执 行函数，可类别 jmp_rm。

运行结果

通过 cputest 中全部测试样例。

测试用例时，首先让 AM 项目上的程序默认编译到 x86-nemu 的 AM 中:

```
--- nexus-am/Makefile.check
+++ nexus-am/Makefile.check
@@ -7,2 +7,2 @@
-ARCH ?= native
+ARCH ?= x86-nemu
 ARCH = $(shell ls $(AM_HOME)/am/arch/)
```

然后在 nexus-am/tests/cputest/ 目录下执行

```
make ALL=xxx run
```

其中 xxx 为测试用例的名称(不包含.c 后缀).

测试样例部分结果如下:

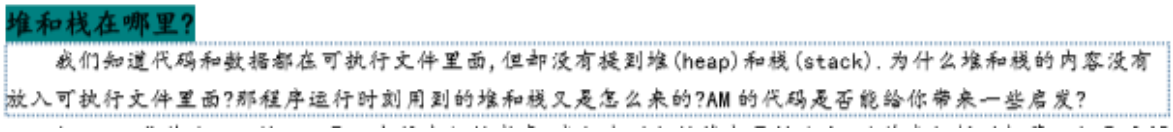
```
(nemu) c
nemu: HIT GOOD TRAP at eip = 0x00100027

(nemu) q
 mul-longlong
qiqi@qiqi-virtual-machine:~/ics2018/nexus-am/tests/cputest$ make ALL=add-longlon
g run
Building add-longlong [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/qiqi/ics2018/nexus-am/tes
ts/cputest/build/add-longlong-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 14:37:31, Apr 27 2020
For help, type "help"
(nemu) c
nemu: HIT GOOD TRAP at eip = 0x00100027

(nemu) q
 add-longlong
qiqi@qiqi-virtual-machine:~/ics2018/nexus-am/tests/cputest$ make ALL=bit run
Building bit [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/qiqi/ics2018/nexus-am/tes
ts/cputest/build/bit-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 14:37:31, Apr 27 2020
For help, type "help"
(nemu) c
nemu: HIT GOOD TRAP at eip = 0x00100027

(nemu) q
 bit
qiqi@qiqi-virtual-machine:~/ics2018/nexus-am/tests/cputest$
```

问题



可执行文件中主要包括：

.text：存可执行文件指令

.rwdata：可读写全局初始值不为 0 的变量

.rodata：只读数据（有些编译器会把只读数据放.text 段）

.bss：可读写全局初始值不为 0 的变量

.init：给 C++等面向对象用的，用于调用全局类变量的构造函数

.dll：动态链接用的地址表 等等。

程序读入到内存后，先创建进程。然后操作系统会给各个段按照链接好的地址分配物理 页面并映射成虚拟页面，之后操作系统会给 .bss 段单独分配一些物理页。

操作系统会把所需的 dll 映射到程序的地址空间，按照空间里的 dll 初始化.dll 所需跳 转地址表。然后操作系统，执行.init 段里面的代码，防止 C++全局变量无法成功构造。以及给程序分配一些其它的必要资源，比如消息队列，用于通信的端口。

之后 OS 给程序初始化堆，这个堆就是 malloc/free，new/delete 对应的内存，操作系统 仅仅是分配了空间段，并没有真正分配几个物理页面，同一个操作系统上每个应用程序的堆 都是一样大的。注意是地址空间，不是实际内存。操作系统本身有一个堆供操作系统自己使 用，这个堆在内核空间，程序看不见也访问不到。程序用到的是用户堆，每个进程有一个， 进程中的每个线程都从这个堆申请内存，这个堆在用户空间。

然后分配栈地址空间，开始创建第一个线程，并把可执行文件头里面指出的第一条指令 地址交给这个线程，开始运行。在创建线程的过程中从栈空间里面分配一块儿空间给这个线 程。

3.2 基础设施

3.2.1 代码：Differential Testing

在nemu/include/common.h 中定义宏 DIFF_TEST 之后，然后完成 difftest_step(uint32_t eip)函数：

```
// TODO: Check the registers state with QEMU.
// Set `diff` as `true` if they are not the same.
if(r.eip != cpu.eip){
    diff = true;
    printf("Diff: eip  QEMU: 0x%08x\n",r.eip);
    printf("           NEMU: 0x%08x\n",cpu.eip);
}
if(r.eax != cpu.eax){
    diff = true;
    printf("Diff: eax  QEMU: 0x%08x\n",r.eax);
    printf("           NEMU: 0x%08x\n",cpu.eax);
}
if(r.ecx != cpu.ecx){
    diff = true;
    printf("Diff: ecx  QEMU: 0x%08x\n",r.ecx);
    printf("           NEMU: 0x%08x\n",cpu.ecx);
}
if(r.ebx != cpu.ebx){
    diff = true;
    printf("Diff: ebx  QEMU: 0x%08x\n",r.ebx);
    printf("           NEMU: 0x%08x\n",cpu.ebx);
}
if(r.edx != cpu.edx){
    diff = true;
    printf("Diff: edx  QEMU: 0x%08x\n",r.edx);
    printf("           NEMU: 0x%08x\n",cpu.edx);
}
if(r.ebp != cpu.ebp){
    diff = true;
    printf("Diff: ebp  QEMU: 0x%08x\n",r.ebp);
    printf("           NEMU: 0x%08x\n",cpu.ebp);
}
if(r.esp != cpu.esp){
    diff = true;
    printf("Diff: esp  QEMU: 0x%08x\n",r.esp);
    printf("           NEMU: 0x%08x\n",cpu.esp);
}
```

重新编译NEMU 后运行,你会发现 NEMU 多输出了 Connect to QEMU successfully 的信息。

```
+ LD build/nemu
c[src/monitor/diff-test/diff-test.c,96,init_difftest] Connect to QEMU successful
ly
[src/monitor/monitor.c,65,load_img] The image is /home/qiqi/ics2018/nexus-am/tes
ts/cputest/build/bit-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 20:44:32, Apr 27 2020
For help, type "help"
(nemu) c
nemu: HIT GOOD TRAP at eip = 0x00100027

(nemu)
```

3.2.2 一键回归测试

在实现指令的过程中,你需要逐个测试用例地运行.但在指令实现正确之后,是不是意味着可以和这些测试 用例说再见呢?显然不是.以后你还需要在 NEMU 中加入新的功能,为了保证加入的新功能没有影响到已有功能的 实现,你还需要重新运行这些测试用例.在软件测试中,这个过程称为回归测试.

将来还要重复运行这些测试用例,手动重新运行每一个测试是一种效率低下的做法.为了提高效率,我们提 供了一个用于一键回归测试的脚本.在 nemu/目录下运行 bash runall.sh来批量运行 nexus-am/tests/cputest/中的所有测试,并报告每个测试用例的运行结果.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;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。

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

4 阶段三 输入输出

4.1 代码：加入IOE

NEMU 框架代码中已经提供了设备的代码，位于 nemu/src/device 目录下.

代码提供了以下模块的模拟:

端口映射 I/O 和内存映射 I/O 两种 I/O 编址方式

串口,时钟,键盘,VGA 四种设备

4.1.1 串口

在nemu/src/cpu/exec/system.c中

```
make_EHelper(in) {
  t2 = pio_read(id_src->val,id_dest->width);
  operand_write(id_dest,&t2);

  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exec.c中

```
/* 0xe4 */    IDEXW(in_I2a,in,1), IDEX(in_I2a,in), IDEXW(out_a2I,out,1), IDEX(out_a2I,out),
/* 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),
```

运行Hello World

运行 Hello World

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

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

打印出10行hello world:

```
qiqi/ics2018/nexus-am/apps/hello/build/hello-x86-nemu.bin
[src/monitor/monitor.c,65,load_img] The image is /home/qiqi/ics2018/nexus-am/a
s/hello/build/hello-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 21:47:21, Apr 27 2020
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
```

4.1.2 时钟

uptime() 返回的是系统（x86-nemu）启动后经过的毫秒数。_ioe_init()中的 boot_time 计算的是系统启动到 IOE 启动时已经经过的毫秒数。故当前 timer 需 要减去初始时间 boot_time。

在nexus-am/am/arch/x86-nemu/src/ioe.c中实现：

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

实现 IOE

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

运行后：

```
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 21:47:21, Apr 27 202
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.
11 seconds.
12 seconds.
13 seconds.
14 seconds.
15 seconds.
16 seconds.
17 seconds.
18 seconds.
19 seconds.
```

有了时钟之后,我们就可以测试一个程序跑多快,从而测试计算机的性能.尝试在 NEMU 中依次运行以下 benchmark(已经按照程序的复杂度排序,均在 nexus-am/apps 目录下;

Dhrystone

Coremark

microbench

成功运行后会输出跑分.跑分以 i7-6700 @ 3.40GHz 的处理器为参照,100000 分表示与参照机器性能相当,100 分表示性能为参照机器的千分之一.

```
Dhrystone Benchmark, Version C, Version 2.2
Trying 500000 runs through Dhrystone.
Finished in 10482 ms
================================================
Dhrystone PASS          98 Marks
                  vs. 100000 Marks (i7-6700 @ 3.40GHz)
nemu: HIT GOOD TRAP at eip = 0x0010006e

(nemu)
```

```
Finised in 11952 ms.
==================================================
CoreMark PASS       373 Marks
                vs. 100000 Marks (i7-6700 @ 3.40GHz)
nemu: HIT GOOD TRAP at eip = 0x0010006e
```

```
==================================================
MicroBench PASS        412 Marks
                vs. 100000 Marks (i7-6700 @ 3.40GHz)
nemu: HIT GOOD TRAP at eip = 0x00100032

(nemu)
```

由分数看出本机的nemu跑的不快。。。。机器配置不行啊。。。

4.1.3 键盘

实现 IOE(2)

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

实现_read_key():

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

运行keytest程序(在nexus-am/tests/keytest目录下), 在程序运行时弹出的新窗口中按下按键, 你将会看到程序输出相应的按键信息。

```
For help, type "help"
(nemu) c
Get key: 45 D down
Get key: 45 D up
Get key: 45 D down
Get key: 45 D up
Get key: 48 H down
Get key: 48 H up
Get key: 34 Y down
Get key: 34 Y up
Get key: 36 I down
Get key: 36 I up
Get key: 49 J down
Get key: 49 J up
Get key: 46 F down
Get key: 46 F up
Get key: 48 H down
Get key: 48 H up
Get key: 49 J down
Get key: 48 H down
Get key: 48 H up
Get key: 49 J up
```

4.1.4 VGA

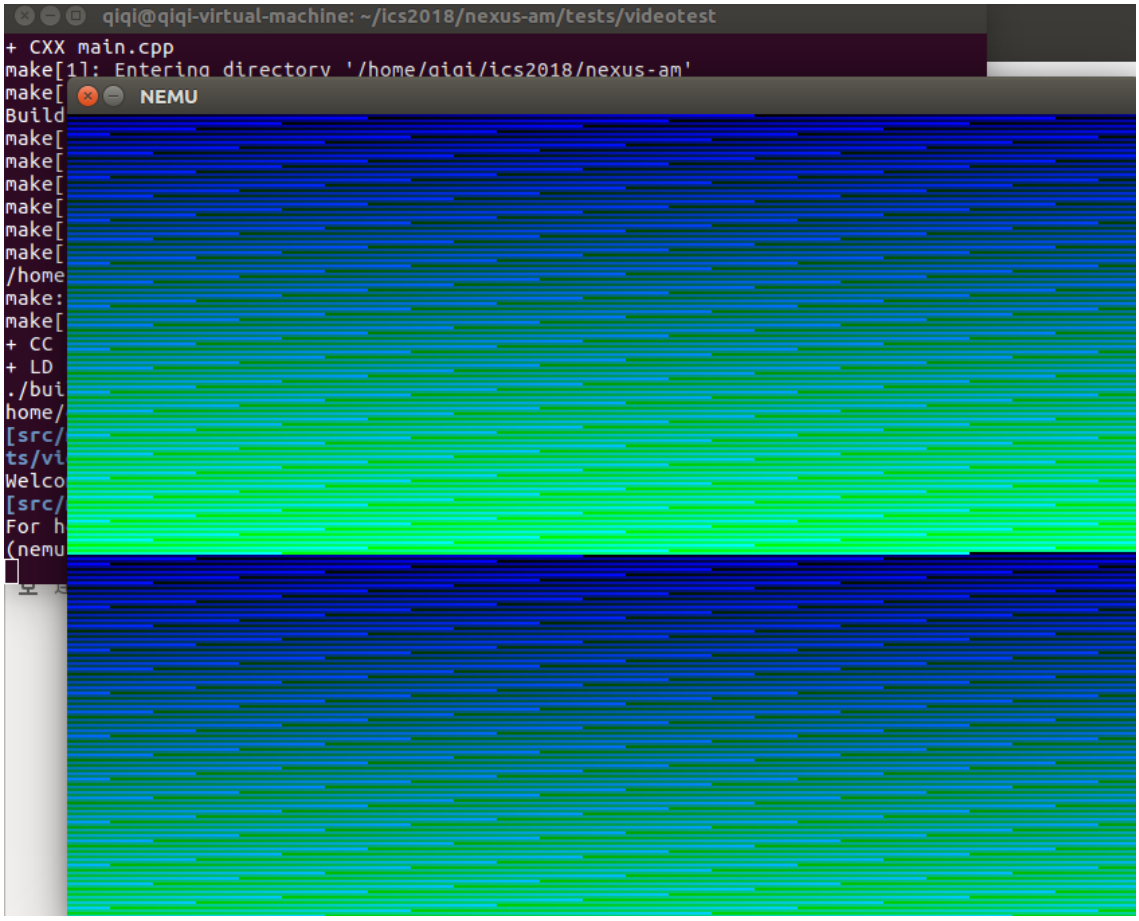
在 paddr_read()和 paddr_write()中加入对内存映射 I/O 的判断。通过 is_mmio()函数判断一个物理地址是 否被映射到 I/O 空间,如果是,is_mmio()会返回映射号,否则返回-1。内存映射 I/O 的访问需要调用 mmio_read() 或 mmio_write(),调用时需要提供映射号.如果不是内存映射 I/O 的访问,就访问 pmem。

在nemu/src/memory/memory.c中:

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

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

运行会看到新窗口中输出了一些颜色信息.



实现 IOE(3)

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

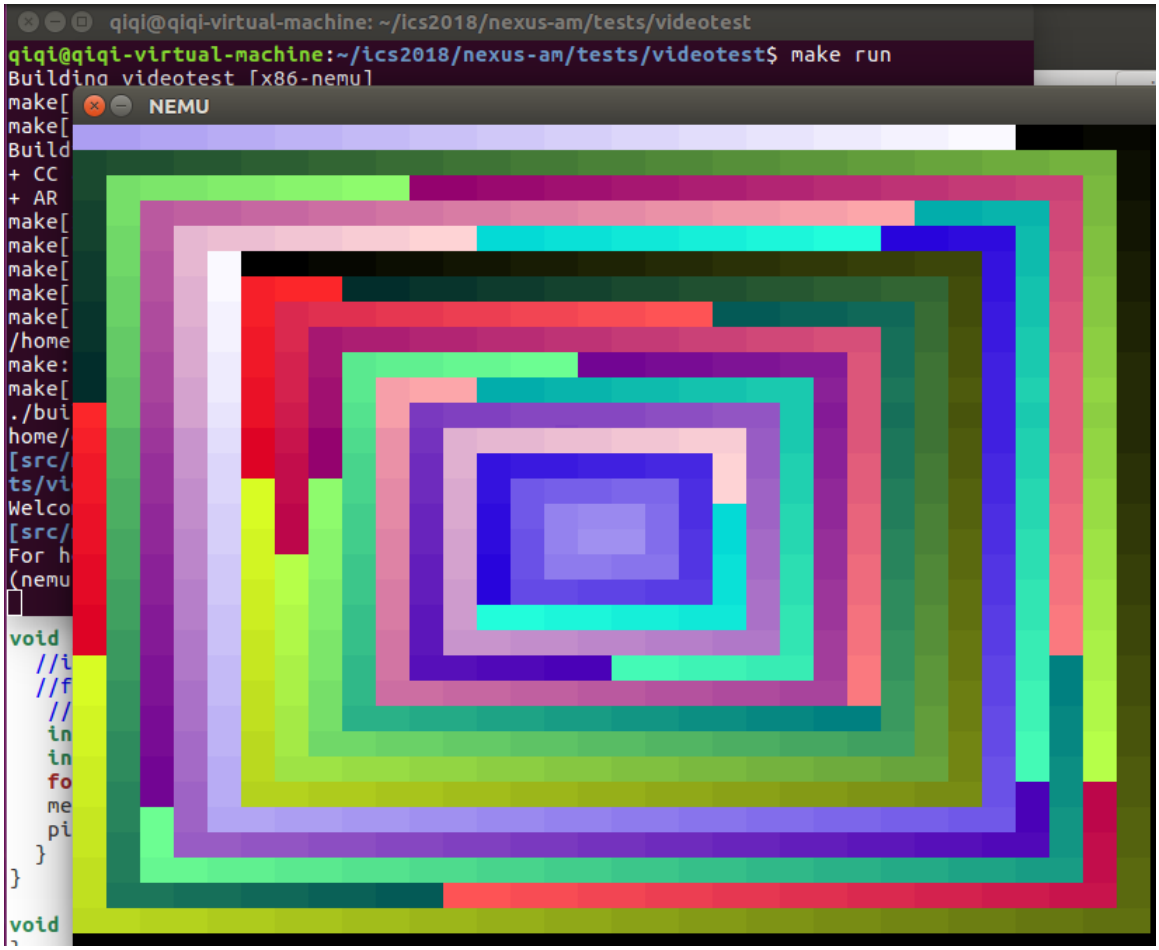
实现draw_rect 函数： 坐标(x,y)对应物理内存 fb 的索引如下图所示，即 index=_screen.width*y+x;

在nexus-am/am/arch/x86-nemu/src/ioe.c中：

类比 nexus-am/am/arch/native/src/gui.c 的_draw_rect()， pixels 是像素数组的首 地址,仿造 native 中的代码实现：

```
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 temp=(w>_screen.width - x)?_screen.width - x: w ;
   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;
  }
}
```

运行结果：具有动画效果



在 NEMU 和 AM 中都完整实现 IOE 后，可以运行打字小游戏(在 nexus-am/apps/typing 目录下).

4.1.5 马里奥

在 x86-nemu 尝试运行马里奥

在 nexus-am/apps/litenes 目录下执行 make run,即可在弹出的新窗口中运行基于 LiteNES 的超级玛丽。

4.1.6 遇到的问题和解决办法

1，在对环境变量修改时，一直不对，运行出现不了Logo，请教学姐后发现在目录下输入pwd后能显示路径。修改完成后发现还是运行不成功，一直报下方的错误：

```
qiqi@qiqi-virtual-machine:~/ics2018/nexus-am/tests/cputest$ pwd
/home/qiqi/ics2018/nexus-am/tests/cputest
qiqi@qiqi-virtual-machine:~/ics2018/nexus-am/tests/cputest$ make ARCH=x86-nemu A
LL=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。
nemu: src/cpu/reg.c:21: reg_test: Assertion `reg_w(i) == (sample[i] & 0xffff)' f
ailed.
Makefile:46: recipe for target 'run' failed
make[2]: *** [run] 已放弃 (core dumped)
/home/qiqi/ics2018/nexus-am/Makefile.app:35: recipe for target 'run' failed
make[1]: *** [run] Error 2
Makefile:12: recipe for target 'Makefile.dummy' failed
make: [Makefile.dummy] Error 2 (ignored)
 dummy
qiqi@qiqi-virtual-machine:~/ics2018/nexus-am/tests/cputest$
```

最后实在不知道错哪了就把pa2删了git branch -d pa2，重新创建pa2，然后就好了。。。。。

2，

准备了一些测试用例. 首先我们让 AM 项目上的程序默认编译到 x86-nemu 的 AM 中：

```
--- nexus-am/Makefile.check
+++ nexus-am/Makefile.check
@@ -7,2 +7,2 @@
-ARCH ?= native
+ARCH ?= x86-nemu
 ARCH = $(shell ls $(AM_HOME)/am/arch/)
```

然后在 nexus-am/tests/cputest/ 目录下执行

```
make ALL=xxx run
```

其中 xxx 为测试用例的名称(不包含.c 后缀).

刚开始没默认编译到x86-nemu，直接用 make ARCH=x86-nemuALL= xxx run也没发现什么问题，直到看到指导上面

(2) 设置默认运行时环境

前两届，有一些同学完成 PA2 实验时，忘记设置默认编译到 x86-nemu 了，导致在实现 IOE 时测试的都是 native 下的运行结果，还天真地以为自己得道成仙了，咋写代码都对orz，所以一定要认真RTFM 啊同志们.....

ps.在实现 IOE 时，可以参考 native 中的 ioe.c 与 gui.c，并查看 native 下的运行效果。

好吧，然后在nexus-am/makefile.check把ARCH ？ =native 改成x86-nemu。

4.2 必答题

必答题

你需要在实验报告中用自己的语言,尽可能详细地回答下列问题.

- 在 nemu/include/cpu/rtl.h 中,你会看到由 static inline 开头定义的各种 RTL 指令函数. 选择其中一个函数,分别尝试去掉 static,去掉 inline 或去掉两者,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?
- 了解 Makefile 请描述你在 nemu 目录下敲入 make 后,make 程序如何组织.c 和.h 文件,最终生成可执行文件 nemu/build/nemu. (这个问题包括两个方面:Makefile 的工作方式和编译链接的过程.) 关于 Makefile 工作方式的提示:
  - Makefile 中使用了变量,包含文件等特性
  - Makefile 运用并重写了一些 implicit rules
  - 在 man make 中搜索-n 选项,也许会对你有帮助
  - RTFM

4.2.1 static inline

Static 函数：在函数的返回类型前加上关键字 static，函数就被定义成为静态函数。普通函数的定义和声明默认情况下是 extern 的，但静态函数只是在声明他的文件当中可见，不能被其他文件所用。因此定义静态函数有以下好处：

<1> 其他文件中可以定义相同名字的函数，不会发生冲突。

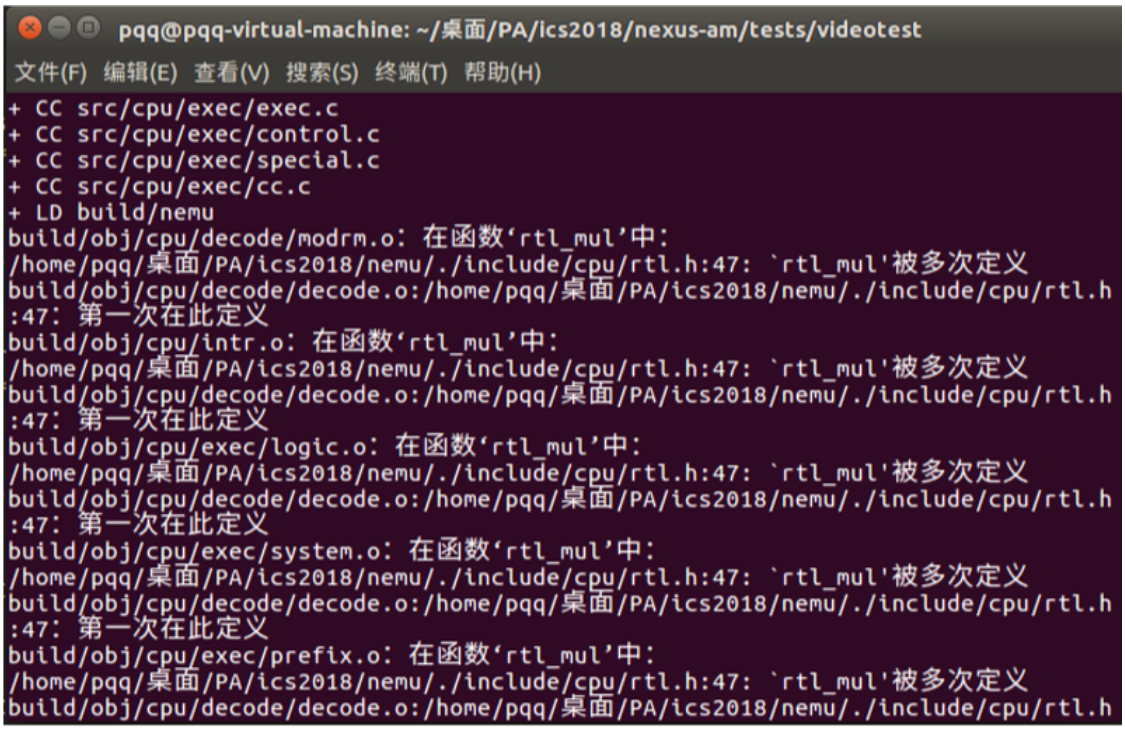
<2> 静态函数不能被其他文件所用。

Inline 函数：内联函数和普通函数最大的区别在于内部的实现方面，而不是表面形式， 普通函数在被调用时，系统首先要跳跃到该函数的入口地址，执行函数体，执行完成后 再返回到函数调用的地方，函数始终只有一个拷贝；而内联函数则不需要进行一个寻址的过程，当执行到内联函数时，此函数展开（很类似宏的使用），如果在 N 处调用了此 内联函数，则此函数就会有 N 个代码段的拷贝。

a) 删掉 static，保留 inline：无报错

b) 删掉 inline，保留 static：报错

c) 删掉 inline 和 static：报错



首先，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 函数其实是跟宏定义类似，不存在所谓的函数 入口。

如果 inline 函数在两个不同的文件 中出现，也就是说一个.h 被两个不同的文件包含，则会出现重名，链接失败。所 以 static inline 的用法就能很好的解决这个问题，使用 static 修饰符，函数仅在文 件内部可见，不会污染命名空间。可以理解为一个 inline 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。

4.2.2 makefile

Makefile：Makefile 文件描述了整个工程的编译、连接等规则。其中包括：工程中的哪些源文件需要编译以及如何编译、需要创建哪些库文件以及如何创建这些库文件、如何最后产生我们想要的可执行文件。尽管看起来可能是很复杂的事情，但是 为工程编写 Makefile 的好处是能够使用一行命令来完成“自动化编译”，一旦提供 一个（通常对于一个工程来说会是多个）正确的 Makefile。编译整个工程你所要

做 的唯一的一件事就是在 shell 提示符下输入 make 命令。整个工程完全自动编译， 极大提高了效率。

make：make 是一个命令工具，它解释 Makefile 中的指令（应该说是规则）。在 Makefile 文件中描述了整个工程所有文件的编译顺序、编译规则。Makefile 有自己 的书写格式、关键字、函数。像 C 语言有自己的格式、关键字和函数一样。而且 在 Makefile 中可以使用系统 shell 所提供的任何命令来完成想要的工作。Makefile （在其它的系统上可能是另外的文件名）在绝大多数的 IDE 开发环境中都在使用， 已经成为一种工程的编译方法。

Makefile 工作方式：

1、make 会在当前目录下找名字叫“Makefile”或“makefile”的文件。

2、如果找到，它会找文件中的第一个目标文件（target），并把这个文件作为最终的目标文件。 3、如果目标文件不存在，或是目标文件所依赖的后面的 .o 文件的文件修改时间 要比目标文件新，那么就会执行后面所定义的命令来生成目标文件。

4、如果目标文件所依赖的.o 文件也不存在，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、C 文件和 H 文件是存在的，于是 make 会生成 .o 文件，然后再用 .o 文件声明 make 的任务，也就是可执行文件。

5 实验结论与感想

1、 学会了一些 Linux 下好用的命令，如在目录下pwd，显示目录的完成路径。

2、 再次深入了解了git

git commit --allow-empty -am”...”
允许在没有任何更改的情况下做代码提交。在切换分支前
对当前暂存区再进行一次提交，确保你可以成功的切换分支。

git checkout master
切换到 master 分支。
git merge pa1
把我们在 pa1 中做到内容合并到 master 中。
git checkout -b pa2
创建 pa2 分支同时切换到 pa2 分支上。

git branch -d pa2

删除分支

3、 知道了看 i386 手册重要性

4、 复习了计算机组成原理，主要是计算机实现指令的流程。

5、 再次学习了一些 C++里不太会用的保留字，比如 static 和 inline，对c++里文件的链接和运行等过程的理解更深刻了。