南开大学
Nankai University

南　开　大　学

计 算 机 学 院

系统综合课程设计实验报告

PA2 实验报告

周辰霏 1712991

年级：2017 级

专业：计算机科学与技术

2020 年 4 月 16 日

目录

一、 概述

(一) 实验目的

- 学习指令周期与指令执行过程, 并简单实现现代指令系统
- 学习运行时环境与 AM 的基本原理, 深入理解 NEMU 的本质
- 了解基础设施测试、调试的基本框架与思想
- 学习 IO 设备的基本实现
- 深入理解冯诺依曼计算机体系结构并尝试在 NEMU 中实现

PA2 承接 PA1 完成的基本部分完成 x86 的主要指令以运行全部的 cputest 实现 CPU 基本的指令执行功能并学习运行时环境、AM 等内容继续完善调试用基础设施。最后根据已有的 IO 设备支持实现简单的 CPU 对 IO 设备的控制。并最终可以实现幻灯片的轮换播放以及打字游戏, 同时也可在 NEMU 中直接运行 Mario 游戏。

(二) 实验内容

PA2 实验涉及现代指令系统的实现、抽象机器 AM 的原理与应用、输入输出设备三部分

1. 了解指令周期与指令执行的原理, 实现几个简单指令, 在 NEMU 中运行第一个 C 程序 dummy
2. 继续完善指令系统, 学习运行时环境和 AM 的基本原理, 更新调试、测试的基础设施——实现 DiffTest
3. 学习 IO 原理, 简单实现 CPU 对输入输出设备的控制

二、 阶段一

(一) dummy.c 运行

一个指令周期可以被抽象为取指 → 译码 → 执行 → 更新 eip, 虽然我们只用实现 jnz、dec、inc 就可以实现全部的可计算任务, 但是 NEMU 做出来是服务于我们而不是让我们更痛苦, 所以我们要实现更多有效指令来帮助我们更好的实现软件运行。

在 $/nexus-am/tests/cputest/$ 中执行 $make\ ARCH=x86-nemu\ ALL=dummy\ run$, 结合生成的 $nexus-am/tests/cputest/build/dummy-x86-nemu.txt$ 反汇编文件以及 monitor 的报错信息可以得出这个什么也没有干的 c 文件需要我们实现 call、push、sub、xor、pop 以及 ret 指令。

而在 RTFSC 和阅读实验指导书之后可以总结出执行流程: ($/nemu/src/cpu/exec/exec.c/$ 中 $exec\_wrapper$ 函数) 将当前 cpu 的 eip 保存到 $decoding.seq\_eip$ 中并将其作为参数传入 $exec\_real$。在 $exec\_real$ 中通过 $instr\_fetch$ 取出 $decoding.seq\_eip$ 对应的指令, 查阅 $opcode\_table$ 得到操作数对应的宽度并通过 set_width 设置宽度, 最后调用 $idex$ 进行译码和执行, 执行主要通过 $make\_DHelper$ 这个宏实现。之后 $exec\_real$ 函数返回到 $exec\_wrapper$, 再通过 $update\_eip$ 更新 eip 的值, 就完成了一个指令周期, 周而复始就构成了一个不停计算的 TRM。

由于操作数有不同类型, 根据指导书指路查看 i386 官方手册的 Appendix A 可以找到以下内容, 以及两个 NEMU 独有的操作数类型代表带符号立即数的 SI 和寄存器 al、ax、eax 的 a

R The mod field of the modR/M byte may refer only to a general register; e.g., MOV (0F20-0F24, 0F26).

G The reg field of the modR/M byte selects a general register; e.g., ADD (00).

I Immediate data. The value of the operand is encoded in subsequent bytes of the instruction.

E A modR/M byte follows the opcode and specifies the operand. The operand is either a general register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, a displacement.

图 1: Appendix A

而我们需要做的部分只有：在 opcode_table 对应位置填写正确的译码函数, 执行函数以及操作数宽度；在 allinstr.h 中声明执行函数 (可省)；在 exec 文件夹中对应文件实现执行函数, 必要时借助 rtl.h 中声明过的 RTL(可省)

根据报错内容在 i386 手册中找到以下指令内容 根据上述 RTFM 结果,call 指令的译码函

CALL — Call Procedure

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

图 2: call-e8-J

PUSH — Push Operand onto the Stack

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

图 3: push-55-r

SUB — Integer Subtraction

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

图 4: sub-83-gp1-6

XOR — Logical Exclusive OR

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

图 5: xor-31-G2E

POP — Pop a Word from the Stack

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

图 6: pop-5d-r

RET — Return from Procedure

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

图 7: ret-c3

数 J($cpu.jmp_eip$=eip+SI), 执行函数 call, 操作数宽度 2/4, 不设置宽度, 所以在 0xe8 的位置填入 IDEX(J,call), 之后根据报错一步步实现符号立即数处理 (SI), 以及 rtl_push 和 rtl_sext 即可。

call 指令实现

```
//nemu/src/cpu/exec/exec.c
/* 0xe8 */      IDEX(J,call),
//nemu/src/cpu/exec/allinstr.h
/*control.c*/
make_EHelper(call);
//nemu/include/cpu/rtl.h
static inline void rtl_push(const rtlreg_t* src1) {
        rtl_subi(&cpu.esp,&cpu.esp,4); //esp-=4
        rtl_sm(&cpu.esp,src1,4); //src1 写入esp
}
static inline void rtl_sext(rtlreg_t* dest, const rtlreg_t* src1, int width) {
  rtl_shli(dest, src1, 32 - width * 8);
  rtl_sari(dest, dest, 32 - width * 8);
}
//nemu/src/cpu/decode/decode.c
static inline make_DopHelper(SI) {
    assert(op->width == 1 || op->width == 4);
    op->type = OP_TYPE_IMM;
    op->simm = 1; //初始化
        t0 = instr_fetch(eip, op->width); //读指令
        rtl_sext(&t0, &t0, op->width); 取SI
        op->simm = t0; //赋值
        //Log("fetch simm: %d", op->simm);
    //TODO();
    rtl_li(&op->val, op->simm);
#ifdef DEBUG
  snprintf(op->str, OP_STR_SIZE, "$0x%x", op->simm);
#endif
}
//nemu/src/cpu/exec/control.c
make_EHelper(call) {
    rtl_push(eip); //压栈eip
    rtl_j(decoding.jmp_eip); //跳转
    print_asm("call %x", decoding.jmp_eip);
}
```

push 指令的译码函数 r(读寄存器的值并存入 op->val), 执行函数 push, 操作数宽度 2/4, 不设置宽度, 所以在 0x50-0x57 的位置填入 IDEX(r,push), 由于 *rtl_push* 在 call 指令中已实现, 之后声明并实现执行函数即可

push 指令实现

```
//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),
//nemu/src/cpu/exec/allinstr.h
/*data-mov.c*/
make_EHelper(push);
```

```
//nemu/src/cpu/exec/data-mov.c
make_EHelper(push) {
    //TODO();
    rtl_push(&id_dest->val);
    print_asm_template1(push);
}
```

sub 指令的译码函数 I2a(读 eax 中数据到 dest, 读立即数写入 src), 执行函数 sub,/5 表示填写在 gp1 的第六个位置即可,EX(sub), 在 reg.h 中补上标志位的相关声明, 在 rtl.h 中实现 *rtl_msb* 以及一系列设置标志位的函数及相关宏再在 arith.c 中实现执行函数 sub 即可

sub 指令实现

```
//nemu/src/cpu/exec/exec.c
/* 0x80, 0x81, 0x83 */
make_group(gp1,
    EX(add), EX(or), EX(adc), EX(sbb),
    EX(and), EX(sub),EX(xor), EX(cmp))
//nemu/include/cpu/reg.h
struct {
                rtlreg_t CF:1;rtlreg_t one1:1;
                rtlreg_t PF:1;rtlreg_t zero1:1;
            rtlreg_t AF:1;rtlreg_t zero2:1;
                rtlreg_t ZF:1;rtlreg_t SF:1;
                rtlreg_t TF:1;rtlreg_t IF:1;
                rtlreg_t DF:1;rtlreg_t OF:1;
                rtlreg_t IOPL:2;rtlreg_t NT:1;
                rtlreg_t zero3:1;rtlreg_t RF:1;
                rtlreg_t VM:1;rtlreg_t zero:16;
    }eflags;
//nemu/include/cpu/rtl.h
static inline void rtl_msb(rtlreg_t* dest, const rtlreg_t* src1, int width) {
    rtl_shri(dest, src1, width*8 - 1);//获取符号位
}
//设置获取标志位
#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; \
  }
static inline void rtl_update_ZF(const rtlreg_t* result, int width) {
        rtl_shli(&at, result, 32 - width * 8);
        if(at == 0) cpu.eflags.ZF = 1;//是0则ZF标记1
        else cpu.eflags.ZF = 0;
}

static inline void rtl_update_SF(const rtlreg_t* result, int width) {
```

```
        rtl_msb(&at, result, width); // 获取符号位
        cpu.eflags.SF = at;
}
//nemu/src/cpu/exec/allinstr.h
/*arith.c*/
make_EHelper(sub);
//nemu/src/cpu/exec/arith.c
make_EHelper(sub) {// 参考sbb代码并删除CF部分
  rtl_sub(&t2, &id_dest->val, &id_src->val);
  rtl_setrelop(RELOP_LTU, &t3, &id_dest->val, &t2);
  operand_write(id_dest, &t2);
  rtl_update_ZFSF(&t2, id_dest->width);
  //CF=1? 进位
  rtl_setrelop(RELOP_LTU, &t0, &id_dest->val, &t2);
  rtl_or(&t0, &t3, &t0);
  rtl_set_CF(&t0);
  //OF 溢出
  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);
//Log("%x-%x=%x",id_dest->val+id_src->val,id_src->val,id_dest->val);

  print_asm_template2(sub);
}
```

xor 指令的译码函数 G2E, 执行函数 xor, 操作数宽度 2/4, 不设置宽度, 所以在 0x31 的位置填入 IDEX(G2E,xor), 之后声明并实现执行函数即可

xor 指令实现

```
//nemu/src/cpu/exec/exec.c
/* 0x30 */      IDEXW(G2E,xor,1), IDEX(G2E,xor),
//nemu/src/cpu/exec/allinstr.h
/*logic.c*/
make_EHelper(xor);
//nemu/src/cpu/exec/logic.c
make_EHelper(xor) {
  //TODO();
  rtl_xor(&id_dest->val, &id_dest->val, &id_src->val);
  operand_write(id_dest,&id_dest->val);
  rtl_li(&t0,0); // t0=0   CF=OF=0
  rtl_set_CF(&t0);
  rtl_set_OF(&t0);
  rtl_update_ZFSF(&id_dest->val, id_dest->width);

  print_asm_template2(xor);
}
```

pop 指令的译码函数 r, 执行函数 pop, 操作数宽度 2/4, 不设置宽度, 所以在 0x5d 的位置填入 IDEX(r,pop), 并在 rtl.h 中实现 *rtl_pop* 之后声明实现执行函数即可

pop 指令实现

```
//nemu/src/cpu/exec/exec.c
/* 0x5c */        IDEX(r,pop), IDEX(r,pop),
//nemu/include/cpu/rtl.h
static inline void rtl_pop(rtlreg_t* dest) {
        rtl_lm(dest,&cpu.esp,4);
        rtl_addi(&cpu.esp,&cpu.esp,4); //esp+=4
}
//nemu/src/cpu/exec/allinstr.h
/*data-mov.c*/
make_EHelper(pop);
//nemu/src/cpu/exec/data-mov.c
make_EHelper(pop) {
  //TODO();
  rtl_pop(&id_src->val);
  operand_write(id_dest,&id_src->val); //写dest

  print_asm_template1(pop);
}
```

ret 指令无译码函数, 执行函数 ret, 所以在 0xc3 的位置填入 EX(ret), 之后声明实现执行函数即可

ret 指令实现

```
//nemu/src/cpu/exec/exec.c
/* 0xc0 */
IDEXW(gp2_Ib2E, gp2,1),IDEX(gp2_Ib2E,gp2),IDEXW(I,ret,2),EX(ret),
//nemu/src/cpu/exec/allinstr.h
/*control.c*/
make_EHelper(ret);
//nemu/src/cpu/exec/control.c
make_EHelper(ret) {
    //TODO();
        rtl_pop(&decoding.jmp_eip);
        rtl_j(decoding.jmp_eip);

    print_asm("ret");
}
```

对以上全部指令正确实现之后, 就可以看到 dummy 的正确运行结果:

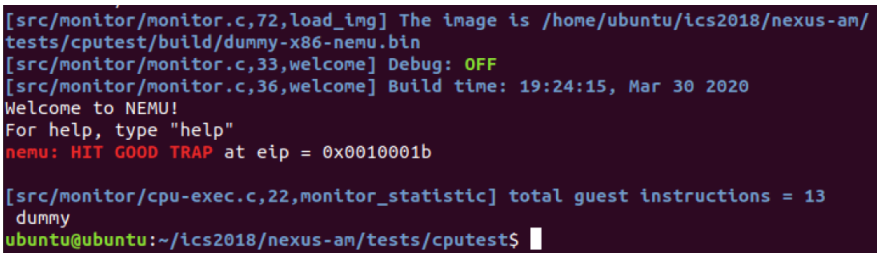

图 8

三、 阶段二

(一) 运行时环境、AM

这一部分内容实际上是在说运行环境以及代码多样性导致的 halt、trap 等指令不同的解决方法——封装成库函数调用各自机器环境下的 API, 就好比 PA1 里面的 *isa_reg_display*, 由 API 在使用运行时环境提供相应功能。2018 代码只有 x86 所以实际上没差。

而 AM 就是使用这种 API 的抽象计算机。它提供运行时的环境,TRM 部分就是 PA1-PA2.2 实现的提供计算能力的部分,IOE 是要在 PA2.3 进行理解和解决的部分。CTE 和 VME 见 PA4。

在了解了这一部分内容后,RTFSC 整个代码结构就会变得容易许多。之后就是按照测试文件的顺序进行逐一实现。(PS: 由于整体的实现过程与 dummy 大同小异, 除特殊情况仅做代码和简单注释)

(二) add

add 中需要实现的指令是 lea、and、nop、add、cmp、sete(setcc)、movzbl、test、je(jcc)、leave。

add

```
//nemu/src/cpu/exec/exec.c
/* 0x8c */      EMPTY, IDEX(lea_M2G,lea),
/* 0x80, 0x81, 0x83 */
make_group(gp1,
    EX(add), EX(or), EX(adc), EX(sbb),
    EX(and), EX(sub),EX(xor), EX(cmp))
/* 0x90 */      EX(nop),
/* 0x00 */
IDEXW(G2E,add,1),IDEX(G2E,add), IDEXW(E2G,add,1), IDEX(E2G,add),
/* 0x38 */
IDEXW(G2E,cmp,1), IDEX(G2E,cmp), IDEXW(E2G,cmp,1), IDEX(E2G,cmp),
//0f对应的是2byte_esc所以要填第二张表查看i386手册90-9f都要填,movzbl同第二张表
/* 0x90 */
IDEXW(setcc_E,setcc,1), IDEXW(setcc_E,setcc,1), IDEXW(setcc_E,setcc,1), IDEXW(setcc_E,setcc,1)
/* 0x94 */
IDEXW(setcc_E,setcc,1), IDEXW(setcc_E,setcc,1), IDEXW(setcc_E,setcc,1), IDEXW(setcc_E,setcc,1)
/* 0x98 */
IDEXW(setcc_E,setcc,1), IDEXW(setcc_E,setcc,1), IDEXW(setcc_E,setcc,1), IDEXW(setcc_E,setcc,1)
/* 0x9c */
IDEXW(setcc_E,setcc,1), IDEXW(setcc_E,setcc,1), IDEXW(setcc_E,setcc,1), IDEXW(setcc_E,setcc,1)
```

```
/* 0xb4 */      EMPTY, EMPTY, IDEXW(mov_E2G,movzx,1),IDEXW(mov_E2G,movzx,2),
/* 0x84 */      IDEXW(G2E,test,1), IDEX(G2E,test),
/* 0x70 */      IDEXW(J,jcc,1), IDEXW(J,jcc,1), IDEXW(J,jcc,1), IDEXW(J,jcc,1),
/* 0x74 */      IDEXW(J,jcc,1), IDEXW(J,jcc,1), IDEXW(J,jcc,1), IDEXW(J,jcc,1),
/* 0x78 */      IDEXW(J,jcc,1), IDEXW(J,jcc,1), IDEXW(J,jcc,1), IDEXW(J,jcc,1),
/* 0x7c */      IDEXW(J,jcc,1), IDEXW(J,jcc,1), IDEXW(J,jcc,1), IDEXW(J,jcc,1),
//根据手册jcc需要将第一张表的70-7f以及0f对应的第二张表的80-8f都填上
/* 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),
/* 0xc8 */      EMPTY, EX(leave),
//nemu/src/cpu/exec/allinstr.h
/*data-mov.c*/
make_EHelper(lea);
make_EHelper(movzx);
make_EHelper(leave);
/*logic.c*/
make_EHelper(test);
make_EHelper(and);
make_EHelper(setcc);
/*special.c*/
make_EHelper(nop);
/*arith.c*/
make_EHelper(add);
make_EHelper(cmp);
/*control.c*/
make_EHelper(jcc);
//nemu/src/cpu/exec/data-mov.c
//lea和movzx已经实现了
make_EHelper(leave) {
  rtl_mv(&cpu.esp, &cpu.ebp);//esp=ebp
  rtl_pop(&cpu.ebp);

  print_asm("leave");
}
//nemu/src/cpu/exec/logic.c
make_EHelper(test) {//and后不返回结果只设置标志位
  rtl_and(&t0, &id_dest->val, &id_src->val);
  rtl_update_ZFSF(&t0, id_dest->width);
  rtl_li(&t0,0);
  rtl_set_CF(&t0);
  rtl_set_OF(&t0);

  print_asm_template2(test);
}
make_EHelper(and) {
  rtl_and(&id_dest->val, &id_dest->val, &id_src->val);
```

```
  operand_write(id_dest, &id_dest->val);
  rtl_li(&t0,0);
  rtl_set_CF(&t0);
  rtl_set_OF(&t0);
  rtl_update_ZFSF(&id_dest->val, id_dest->width);

  print_asm_template2(and);
}
//虽然setcc的声明在logic里但实际需要实现的是cc.c里的rtl_setcc
//nemu/src/cpu/exec/cc.c—rtl_setcc
switch (subcode & 0xe) {//对i386手册内容翻译
    case CC_O:
      rtl_get_OF(&t0);*dest = t0 ? 1 : 0;break;
      //set byte if overflow (OF = 1)
    case CC_B:
      rtl_get_CF(&t0);*dest = t0 ? 1 : 0;break;
      //set byte if below (CF = 1)
    case CC_E:
      rtl_get_ZF(&t0);*dest = t0 ? 1 : 0;break;
      //set byte if equal (ZF = 1)
    case CC_BE:
      rtl_get_ZF(&t0);rtl_get_CF(&t1);*dest = (t0 || t1) ? 1 : 0;break;
      //set byte if below or equal (CF = 1 or ZF = 1)
    case CC_S:
      rtl_get_SF(&t0);*dest = t0 ? 1 : 0;break;
      /set byte if sign (SF = 1)
    case CC_L:
      rtl_get_SF(&t0);rtl_get_OF(&t1);*dest = (t0 != t1) ? 1 : 0;break;
      //set byte if less (SF != OF)
    case CC_LE:
      rtl_get_SF(&t0);rtl_get_OF(&t1);
      rtl_get_ZF(&t2);*dest = ((t0 != t1) || t2) ? 1 : 0;break;
      //set byte if less or equal (ZF = 1 or SF != OF)
      //TODO();
    default: panic("should not reach here");
    case CC_P: panic("n86 does not have PF");
}
//nop不需要写——因为本身就是 no operation的意思
//nemu/include/cpu/rtl.h
static inline void rtl_not(rtlreg_t *dest, const rtlreg_t* src1) {
    *dest=~(*src1);
}
//nemu/src/cpu/exec/arith.c
make_EHelper(add) {//参考adc删去CF两句
  rtl_add(&t2, &id_dest->val, &id_src->val);
  rtl_setrelop(RELOP_LTU, &t3, &t2, &id_dest->val);
  operand_write(id_dest, &t2);
  rtl_update_ZFSF(&t2, id_dest->width);
```

```
  rtl_setrelop(RELOP_LTU, &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, &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);
}
make_EHelper(cmp) {
  rtl_sub(&t2, &id_dest->val, &id_src->val); // 相减
  rtl_setrelop(RELOP_LTU, &t3, &id_dest->val, &t2); // dest<t2?
  rtl_update_ZFSF(&t2, id_dest->width);
  rtl_setrelop(RELOP_LTU, &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); // 符号位看溢出
  rtl_set_OF(&t0);

  print_asm_template2(cmp);
}
//jcc 也已实现执行函数
```

(三) add-longlong

add-longlong 只需实现 adc 和 or 在 opcode_table 中的部分以及 or 的执行函数

add-longlong

```
//nemu/src/cpu/exec/exec.c
/* 0x10 */      IDEXW(G2E,adc,1),IDEX(G2E,adc),IDEXW(E2G,adc,1),IDEX(E2G,adc),
/* 0x08 */  IDEXW(G2E,or,1), IDEX(G2E,or),
//nemu/src/cpu/exec/logic.c
  rtl_or(&id_dest->val, &id_dest->val, &id_src->val);
  operand_write(id_dest, &id_dest->val);
  rtl_li(&t0,0);
  rtl_set_CF(&t0);
  rtl_set_OF(&t0);
  rtl_update_ZFSF(&id_dest->val, id_dest->width);

  print_asm_template2(or);
}
```

（四） bit、bubble-sort

需要实现 sar、shl、dec 指令以及 bubble-sort 里的 inc

bit

```
//nemu/src/cpu/exec/exec.c
/* 0xc0, 0xc1, 0xd0, 0xd1, 0xd2, 0xd3 */
make_group(gp2,
    EX(rol), EMPTY, EMPTY, EMPTY,
    EX(shl), EX(shr), EMPTY, EX(sar))
/* 0xfe */
make_group(gp4,
    EX(inc), EX(dec), EMPTY, EMPTY,
    EMPTY, EMPTY, EMPTY, EMPTY)
/* 0x40 */      IDEX(r,inc), IDEX(r,inc), IDEX(r,inc), IDEX(r,inc),
//nemu/src/cpu/exec/logic.c
make_EHelper(sar) {
  rtl_sext(&id_dest->val, &id_dest->val, id_dest->width); //取符号
  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);

  print_asm_template2(sar);
}
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);

  print_asm_template2(shl);
}
//nemu/src/cpu/exec/arith.c
make_EHelper(dec) {
  rtl_subi(&t2, &id_dest->val, 1);
  rtl_setrelop(RELOP_LTU, &t3, &id_dest->val, &t2);
  operand_write(id_dest, &t2);
  rtl_update_ZFSF(&t2, id_dest->width);
  rtl_setrelop(RELOP_LTU, &t0, &id_dest->val, &t2);
  rtl_or(&t0, &t3, &t0);
  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);
}
make_EHelper(inc) {
  rtl_addi(&t2, &id_dest->val,1);
```

```
  rtl_setrelop(RELOP_LTU, &t3, &t2, &id_dest->val);
  operand_write(id_dest, &t2);
  rtl_update_ZFSF(&t2, id_dest->width);
  rtl_setrelop(RELOP_LTU, &t0, &t2, &id_dest->val);
  rtl_or(&t0, &t3, &t0);
  rtl_xori(&t0, &id_dest->val, 1);
  rtl_not(&t0, &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);
}
```

（五） 其他指令

imul、cltd、idiv、jmp、shr、sbb

余下指令

```
//nemu/src/cpu/exec/exec.c
/* 0xf6, 0xf7 */
make_group(gp3,
    IDEX(test_I,test), EMPTY, EX(not), EX(neg),
    EX(mul), EX(imul1), EX(div), EX(idiv))
/* 0x68 */      IDEX(push_SI,push), IDEX(I_E2G,imul3), IDEXW(push_SI,push,1), IDEXW(I_E2G,imul
/* 0x98 */      EX(cwtl), EX(cltd),
/* 0x18 */      IDEXW(G2E,sbb,1), IDEX(G2E,sbb), IDEXW(E2G,sbb,1), IDEX(E2G,sbb),
//nemu/src/cpu/exec/data-mov.c
make_EHelper(cltd) {// 引用
  if (decoding.is_operand_size_16) {//16b
  rtl_sext(&t0,&reg_l(R_EAX),2); //EAX的符号位
  rtl_mv(&reg_l(R_EDX), (&t0)+2); // 到edx中变为双长
  }
  else {
  rtl_sari(&reg_l(R_EDX), &reg_l(R_EAX), 31); // 符号位扩展到edx
  }
//nemu/src/cpu/exec/logic.c
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);

  print_asm_template2(shr);
}
```

确保全部指令实现正确 (monitor 返回 good trap) 后就可以通过测试样例了, 但是其实好多指令并没有实现 (TODO 遍布在 NEMU 的每一个角落) 但是可以继续往下实现常用库函数

（六） 字符串处理函数

实在是想不到有一天我一个用都用不熟的人还要来实现它本身 (实现了一下发现其实比用起来简单), 那就只能借助 man 命令的 RTFM 和 STFW 实现了。整体不是很难, 基本上只要弄清楚这个函数的用途以及字符串本身特征还有数据类型转换正确等等就可以了。

string.c

```c
//nexus-am/libs/klib/src/string.c
size_t strlen(const char *s) {
  size_t i = 0;//遇到结束符\0之前统计字符个数即可
  while(*(s + i) != '\0')
    i ++;
  return i;
}
char *strcpy(char* dst,const char* src) {
  int i = 0;//逐个复制
  while(*(src + i) != '\0'){
    *(dst + i) = *(src + i);
    i ++;
  }
  return dst;
}
char* strncpy(char* dst, const char* src, size_t n) {
  size_t i;//规定长度加上限制，同时不能忘了在最后加上\0
  for(i = 0; i < n && src[i] != '\0'; ++i)
    dst[i] = src[i];
  for( ; i < n; ++i)
    dst[i] = '\0';
  return dst;
}
char* strcat(char* dst, const char* src) {
  size_t dst_len = strlen(dst);//拼接即可
  size_t i = 0;
  for(i = 0; src[i] != '\0'; ++i)
    dst[dst_len + i] = src[i];
  dst[dst_len + i] = '\0';
  return dst;
}
int strcmp(const char* s1, const char* s2) {
  int len1 = strlen(s1),len2 = strlen(s2),len = 0;
  if(len1 < len2) len = len1;//比对到短的即可
  else len = len2;
  for(int i = 0; i <= len; ++i){
    if(s1[i] < s2[i]) return -1;
    else if(s1[i] > s2[i]) return 1;
  }
  return 0;//一样
}
int strncmp(const char* s1, const char* s2, size_t n) {
```

```
  for(size_t i = 0; i < n; ++i){// 最多比较前n个字符
    if(s1[i] < s2[i]) return -1;
    else if(s1[i] > s2[i]) return 1;
  }
  return 0;
}
void* memset(void* v,int c,size_t n) {
  if(v == NULL || n < 0) return NULL;
  char* temp = (char*) v;
  for(int i = 0; i < n; ++i)
    temp[i] = (char) c;// 按ASCII全体赋值
  return v;
}
void* memcpy(void* out, const void* in, size_t n) {
  if(out == NULL || n < 0) return NULL;
  char* chardest = (char*) out;
  char* charsrc = (char*) in;
  for(int i = 0; i < n; ++i)
    chardest[i] = charsrc[i];//copy
  return out;
}

int memcmp(const void* s1, const void* s2, size_t n){
  char* ch1 = (char*) s1;// 同strcmp理
  char* ch2 = (char*) s2;
  for(int i = 0; i < n; ++i){
    if(ch1[i] < ch2[i]) return -1;
    else if(ch1[i] > ch2[i]) return 1;
  }
  return 0;
}
```


将 string.c 实现之后运行 string 测试样例即可得到 good trap 结果

（七） 标准输入输出

之后需要实现 stdio.c 中的标准输入输出才可以使 hello-str 通过 (既然迟早都得实现为什么不干脆一次性实现完)。就像 PA2.3 里提及的 sprintf 和 printf 的代码会有较大的重叠, 即他们都有和 vsprintf 实现内容重叠的部分, 所以先实现 vsprintf 之后再调用实现 sprintf 和 printf

stdio.c

```
//nexus-am/libs/klib/src/stdio.c
int vsprintf(char *out, const char *fmt, va_list ap) {
  char *outp;
  nt out_len=0;
  int width;// 宽度
  int flags;// 标记
  char nums[1000];
  char *ss=nums;
```

```
for(outp=out;*fmt;fmt++){
  if(*fmt!='%'){//计数
    *outp++=*fmt;
    out_len++;
    continue;
  }
  int temp=1;//处理flags
  flags=0;
  while(temp==1){
    fmt++;
    switch(*fmt){
      case '0':flags|=1;break;//左边置0
      case '+':flags|=4;break;//显示正负号
      case ' ':flags|=8;break;//插入空格
      case '-':flags|=16;break;//左对齐
      case '#':flags|=32;break;//0x.
      default:temp=0;
    }
  }
  width=0;//处理宽度
  if(*fmt>='0'&&*fmt<='9'){
    for(;*fmt>='0'&&*fmt<='9';fmt++){
      width=width*10+*fmt-'0';//编译原理处理方法
    }
  }
  else if(*fmt=='*'){
    fmt++;
    width=va_arg(ap,int);//在ap里找下一个int型
    if(width<0){
      width=-width;
      flags|=16;
    }
  }
  switch(*fmt){
    case 'd':break;
    case 's':{
      char *s=va_arg(ap,char *);
      int len_s=strlen(s);
      if(!(flags&16)){//默认右对齐
        while(len_s<width--){//不足width的填空格
          *outp++=' ';out_len++;
        }
      }
      for(int i=0;i<len_s;i++){
        *outp++=*s++;out_len++;
      }
      while(len_s<width--){//补空格到width
        *outp++=' ';out_len++;
```

```
57            }
58            continue;
59          }
60        }
61
62        int num=va_arg(ap,int);
63        int count=0;
64        if(num==0){
65          *ss++='0';
66          count+=1;
67        }
68        else{
69          if(num<0){// 负号
70            *outp++='-';out_len++;
71            num=-num;
72          }
73          while(num){
74            *ss++=num%10+'0';// 转 str
75            num=num/10;
76            count+=1;
77          }
78        }
79        if(count<width){
80          num=width-count;
81          if(flags&1){// flags0
82            while(num--){
83              *outp++='0';out_len++;
84            }
85          }
86          else if(flags&8){// flags ' '
87            while(num--){
88              *outp++=' ';out_len++;
89            }
90          }
91        }
92        while(count--){
93          *outp++=*--ss;out_len++;
94        }
95      }
96      *outp='\0';
97      return out_len;// 返回写入的字符总数
98  }
99  int sprintf(char *out, const char *fmt, ...) {
100   va_list args;
101   va_start(args,fmt);// 实际上就是调用 vsprintf 把 args 替换到 fmt 里面存到 out
102   int out_len=vsprintf(out,fmt,args);
103   va_end(args);
104   return out_len;
```

```
105 }
106 int printf(const char *fmt, ...) {
107   va_list args;
108   va_start(args,fmt);
109   char out[1000];
110   int out_len=vsprintf(out,fmt,args);
111   va_end(args);
112   for(int i=0;i<out_len;i++){
113     //nexus-am/am/arch/x86-nemu/src/trm.c中
114     _putc(out[i]);//使用_putc输出由于串口未实现所以此时并不可以正确输出
115   }
116   return 0;
117 }
```

实现完 stdio.c 之后再进行 hello-str 的测试就可以正常通过了, 此时使用后文提及的回归测试所用的 bash 脚本 runall.sh 即可测出正常通过测试。

图 9: runall.sh

（八） Differential Testing

这里的 DiffTest 虽然放在了 PA2.2 的末尾但是作为基础设施! 它应该在 PA2.1 至少 2.2 的开头实现, 不然真的难以 de 那些奇奇怪怪的错, 光靠 native 上跑一个没有输出的测试样例是正确的显然不行, 借助 DiffTest 和 Qemu 进行一步一步的对比就可以更快的找出错在哪里, 尤其是标志位出错, 大部分出错都和标志位还有寄存器的写有关, 再者是逐步执行并比对, 所以可以更准确的定位错误, 所以虽然在章节的末尾, 实际上我在 2.2 疯狂报错找不出原因以后就开始写了。

在 common.h 中定义 DiffTest 并实现 difftest_step 中 DiffTest 的核心功能即可

DiffTest

```
//nemu/src/monitor/difftest/difftest.c
//代码框架中载入QEMU并连接逐步执行都已经实现，只需要把reg的检查补充完整就好
bool diff=false;//确有不同
if(ref_r.eip != cpu.eip){
      diff = true;
      printf("Diff: eip  QEMU: 0x%08x\n",ref_r.eip);
      printf("           NEMU: 0x%08x\n",cpu.eip);
  }
  if(ref_r.eax != cpu.eax){
      diff = true;
      printf("Diff: eax  QEMU: 0x%08x\n",ref_r.eax);
      printf("           NEMU: 0x%08x\n",cpu.eax);
  }
  if(ref_r.ecx != cpu.ecx){
      diff = true;
      printf("Diff: ecx  QEMU: 0x%08x\n",ref_r.ecx);
      printf("           NEMU: 0x%08x\n",cpu.ecx);
  }
  if(ref_r.ebx != cpu.ebx){
      diff = true;
      printf("Diff: ebx  QEMU: 0x%08x\n",ref_r.ebx);
      printf("           NEMU: 0x%08x\n",cpu.ebx);
  }
  if(ref_r.edx != cpu.edx){
      diff = true;
      printf("Diff: edx  QEMU: 0x%08x\n",ref_r.edx);
      printf("           NEMU: 0x%08x\n",cpu.edx);
  }
  if(ref_r.ebp != cpu.ebp){
      diff = true;
      printf("Diff: ebp  QEMU: 0x%08x\n",ref_r.ebp);
      printf("           NEMU: 0x%08x\n",cpu.ebp);
  }
  if(ref_r.esp != cpu.esp){
      diff = true;
      printf("Diff: esp  QEMU: 0x%08x\n",ref_r.esp);
      printf("           NEMU: 0x%08x\n",cpu.esp);
  }
  if(ref_r.esi != cpu.esi){
      diff = true;
      printf("Diff: esi  QEMU: 0x%08x\n",ref_r.esi);
      printf("           NEMU: 0x%08x\n",cpu.esi);
  }
  if(ref_r.edi != cpu.edi){
      diff = true;
      printf("Diff: edi  QEMU: 0x%08x\n",ref_r.edi);
      printf("           NEMU: 0x%08x\n",cpu.edi);
```

```
  }
  if (diff) {// 只比对NEMU有的标志位
    printf("ZF---NEMU:%d QEMU:%d\n",cpu.eflags.ZF,ref_r.eflags.ZF);
    printf("SF---NEMU:%d QEMU:%d\n",cpu.eflags.SF,ref_r.eflags.SF);
    printf("OF---NEMU:%d QEMU:%d\n",cpu.eflags.OF,ref_r.eflags.OF);
    printf("CF---NEMU:%d QEMU:%d\n",cpu.eflags.CF,ref_r.eflags.CF);
    printf("IF---NEMU:%d QEMU:%d\n",cpu.eflags.IF,ref_r.eflags.IF);
    nemu_state = NEMU_ABORT;// abort返回
  }
```

四、 阶段三

(一) 串口

串口作为最简单的输入输出设备, 基本的 API 都已经实现, 只需要实现 system.c 中的 in 和 out 指令调用相应函数即可实现简单输入输出。实现参照 port-io.c 的 pio_read_common 和 pio_write_common 对宽度进行区分即可。

串口 in out

```
//nemu/src/cpu/exec/system.c
make_EHelper(in) {// 参照src/device/io/port-io.c
  switch(id_dest->width){
    case 4: rtl_li(&t0, pio_read_l(id_src->val)); break;
    case 2: rtl_li(&t0, pio_read_w(id_src->val)); break;
    case 1: rtl_li(&t0, pio_read_b(id_src->val)); break;
  }
  operand_write(id_dest,&t0);

  print_asm_template2(in);

#if defined(DIFF_TEST)
  difftest_skip_ref();
#endif
}
make_EHelper(out) {
  switch(id_dest->width){
    case 4: pio_write_l(id_dest->val, id_src->val); break;
    case 2: pio_write_w(id_dest->val, id_src->val); break;
    case 1: pio_write_b(id_dest->val, id_src->val); break;
  }

  print_asm_template2(out);

#if defined(DIFF_TEST)
  difftest_skip_ref();
#endif
}
```

完成之后在 nexus-am/apps/hello/下 make run 可以得到正确输出 10 次的 Hello World!

图 10: 串口 io

(二) 时钟

时钟功能在 timer.c 里也被做了简化, 只需要实现 _DEVREG_TIMER_UPTIME 这一表示 AM 系统启动时间的寄存器内容更新即可

timer

```
// nexus-am/am/arch/x86-nemu/src/devices/timer.c
#define RTC_PORT 0x48
uint32_t boot_time;
void timer_init() {
        boot_time = inl(RTC_PORT);
}
case _DEVREG_TIMER_UPTIME: { // case 1 hi和lo拼起来是最后的ms
    _UptimeReg *uptime = (_UptimeReg *)buf;
    uint64_t temptime = inl(RTC_PORT) - boot_time; // inl->uint32_t
    uptime->hi = (uint32_t) (temptime >> 32); // 高32
    uptime->lo = (uint32_t) temptime; // 低32
    return sizeof(_UptimeReg);
}
```

实现之后可以得到 timetest 每隔 1s 的输出:

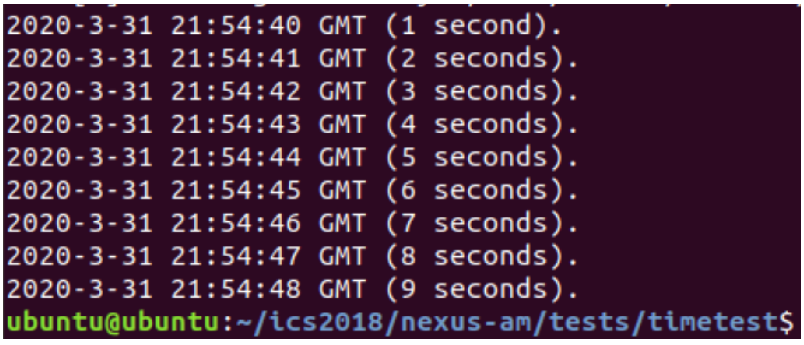

图 11: timer

关闭 DEBUG 和 DIFFTEST 宏之后对三个 benchmark 进行跑分测试:

```
Dhrystone Benchmark, Version C, Version 2.2
Trying 500000 runs through Dhrystone.
Finished in 134 ms
------------------------------------------------
Dhrystone PASS          7688 Marks
                  vs. 100000 Marks (i7-6700 @ 3.40GHz)
```

图 12: dhrystone-nemu

```
Dhrystone Benchmark, Version C, Version 2.2
Trying 500000 runs through Dhrystone.
Finished in 150 ms
------------------------------------------------
Dhrystone PASS          6868 Marks
                  vs. 100000 Marks (i7-6700 @ 3.40GHz)
```

图 13: dhrystone-native

```
Running CoreMark for 1000 iterations
2K performance run parameters for coremark.
CoreMark Size    : 666
Total time (ms)  : 383
Iterations       : 1000
Compiler version : GCC7.5.0
seedcrc          : 0xe9f5
[0]crclist       : 0xe714
[0]crcmatrix     : 0x1fd7
[0]crcstate      : 0x8e3a
[0]crcfinal      : 0xd340
Finised in 383 ms.
------------------------------------------------
CoreMark PASS       11667 Marks
                vs. 100000 Marks (i7-6700 @ 3.40GHz)
```

图 14: coremark-nemu

```
Running CoreMark for 1000 iterations
2K performance run parameters for coremark.
CoreMark Size    : 666
Total time (ms)  : 338
Iterations       : 1000
Compiler version : GCC7.5.0
seedcrc          : 0xe9f5
[0]crclist       : 0xe714
[0]crcmatrix     : 0x1fd7
[0]crcstate      : 0x8e3a
[0]crcfinal      : 0xd340
Finised in 338 ms.
------------------------------------------------
CoreMark PASS       13220 Marks
                vs. 100000 Marks (i7-6700 @ 3.40GHz)
```

图 15: coremark-native

```
[qsort] Quick sort: * Passed.
  min time: 68 ms [8116]
[queen] Queen placement: * Passed.
  min time: 102 ms [5057]
[bf] Brainf**k interpreter: * Passed.
  min time: 237 ms [11058]
[fib] Fibonacci number: * Passed.
  min time: 188 ms [15199]
[sieve] Eratosthenes sieve: * Passed.
  min time: 200 ms [21203]
[15pz] A* 15-puzzle search: * Passed.
  min time: 75 ms [7722]
[dinic] Dinic's maxflow algorithm: * Passed.
  min time: 85 ms [15924]
[lzip] Lzip compression: * Passed.
  min time: 173 ms [15300]
[ssort] Suffix sort: * Passed.
  min time: 53 ms [11160]
[md5] MD5 digest: * Passed.
  min time: 123 ms [15929]
------------------------------------------------
MicroBench PASS         12666 Marks
                  vs. 100000 Marks (i7-6700 @ 3.40GHz)
Exit (0)
```

图 16: microbench-nemu

```
[qsort] Quick sort: * Passed.
  min time: 86 ms [6417]
[queen] Queen placement: * Passed.
  min time: 51 ms [10115]
[bf] Brainf**k interpreter: * Passed.
  min time: 207 ms [12661]
[fib] Fibonacci number: * Passed.
  min time: 149 ms [19177]
[sieve] Eratosthenes sieve: * Passed.
  min time: 185 ms [22922]
[15pz] A* 15-puzzle search: * Passed.
  min time: 83 ms [6978]
[dinic] Dinic's maxflow algorithm: * Passed.
  min time: 72 ms [18800]
[lzip] Lzip compression: * Passed.
  min time: 143 ms [18509]
[ssort] Suffix sort: * Passed.
  min time: 41 ms [14426]
[md5] MD5 digest: * Passed.
  min time: 82 ms [23893]
------------------------------------------------
MicroBench PASS         15389 Marks
                  vs. 100000 Marks (i7-6700 @ 3.40GHz)
Exit (0)
```

图 17: microbench-native

(三) 键盘

键盘的按下检测在 OS 中实际上是一种中断, 在 NEMU 中同样和上述外设一样还是有一个专用寄存器来完成读写的过程。

keyboard

```
1  //nexus-am/am/arch/x86-nemu/src/devices/input.c
2  #define I8042_DATA_PORT 0x60
3  case _DEVREG_INPUT_KBD: {
4      _KbdReg *kbd = (_KbdReg *)buf;
5      int keytemp = inl(I8042_DATA_PORT); // 获取键盘码
6      kbd->keydown = ((keytemp & 0x8000) == 0x8000) ? 1 : 0; // 是否摁下
7      kbd->keycode = keytemp;
8      return sizeof(_KbdReg);
9  }
```

之后的 keytest 测试结果如下:

```
Get key: 60 B down
Get key: 60 B up
Get key: 66 RSHIFT down
Get key: 50 K down
Get key: 50 K up
Get key: 49 J down
Get key: 49 J up
Get key: 49 J down
Get key: 49 J up
Get key: 66 RSHIFT up
Get key: 15 1 down
Get key: 15 1 up
Get key: 16 2 down
Get key: 16 2 up
Get key: 17 3 down
Get key: 17 3 up
ubuntu@ubuntu:~/ics2018/nexus-am/tests/keytest$
```

图 18: keyboard

(四) VGA

VGA 设备需要实现屏幕大小信息给 CPU 的传递以及通过 _FBCtlReg 结构体写入进行图像显示, 同样是根据 amdev.h 中声明的成员变量进行传递赋值即可。

vga

```
// nexus-am/am/arch/x86-nemu/src/devices/video.c
#define SCREEN_PORT 0x100
#define SCREEN_H 300
#define SCREEN_W 400
case _DEVREG_VIDEO_INFO: {
    _VideoInfoReg *info = (_VideoInfoReg *)buf;
    info->width = SCREEN_W;
    info->height = SCREEN_H;
    return sizeof(_VideoInfoReg);
}
case _DEVREG_VIDEO_FBCTL: {
    _FBCtlReg *ctl = (_FBCtlReg *)buf;
        int x = ctl->x, y = ctl->y, w = ctl->w, h = ctl->h;
        uint32_t *pixels = ctl->pixels;
        int cp_bytes = sizeof(uint32_t) * min(w, SCREEN_W - x);
        for(int j = 0; j < h  && y + j < SCREEN_H; ++j){
                memcpy(&fb[(y + j) * SCREEN_W + x], pixels, cp_bytes);
            pixels += w;
        }
    if (ctl->sync) {
        // do nothing, hardware syncs.
    }
      return sizeof(_FBCtlReg);
}
```

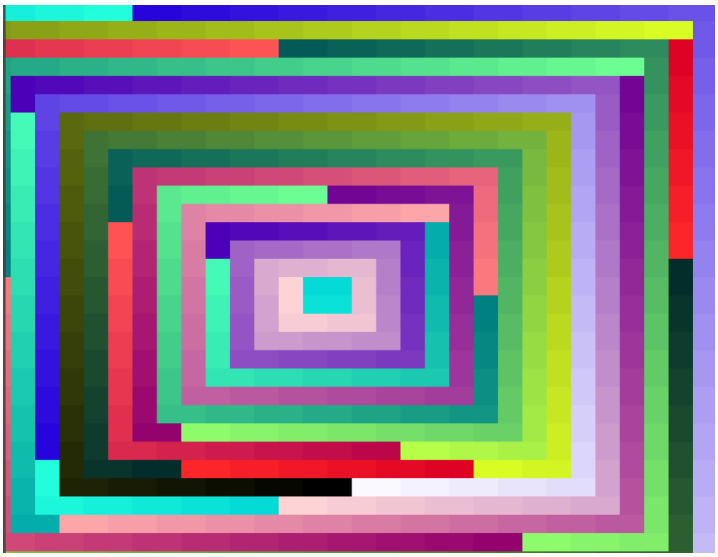

图 19: vga

全部 IOE 实现以后程序打字机和幻灯片就可以运行了,Mario 也可以了

图 20: slider

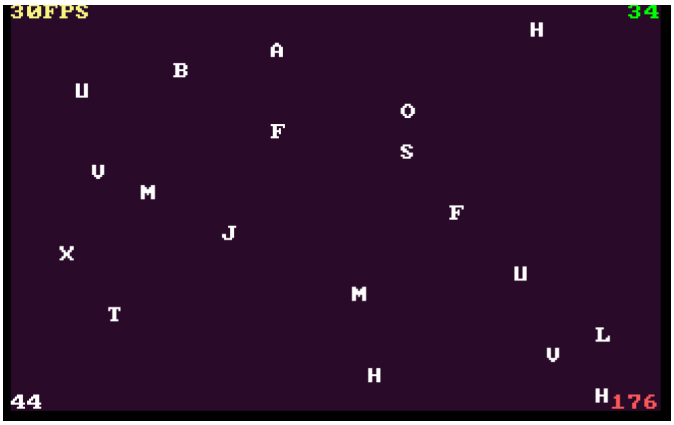

图 21: typing

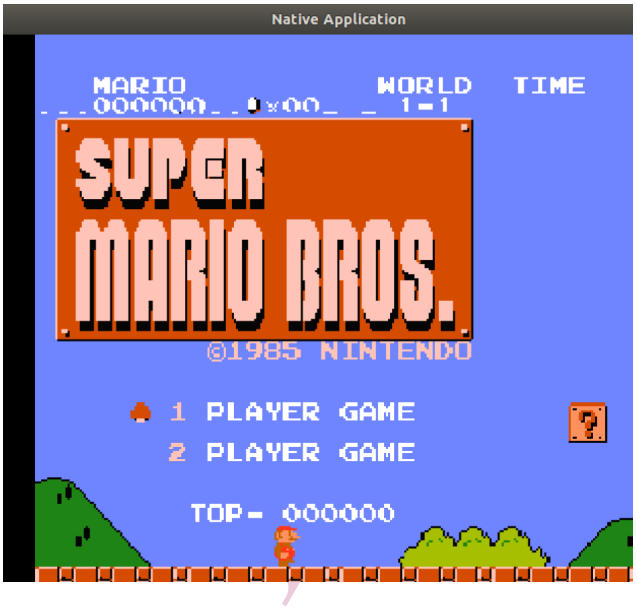

图 22: vga

五、 遇到的 bug 及解决

收回 PA1 的话,PA1 没有习难,PA1 进展太顺利了,PA2 我重写了两次, 不好好看指导书的锅 orz

- ☑ test 指令的 i386 手册表述不是很明确, 本意是不需要写与的结果的, 这个 STFW 了一下才弄清除, 只更新标志位就行了
  ☑ 有的 RTL 指令一定要记得用 operand_write 到 dest 或者需要写的地方, 标志位更新设置的顺序也一定要谨慎, 再就是临时寄存器的使用一定要分清用在哪里, 所以后来有一部分我直接赋给 id_dest 了免得写着写着自己混乱了
  ☑ sar 指令是需要借助 SI 还有 msb 等等进行符号扩展的毕竟是算术移位, 否则会报错
  ☑ PA2 教会了我: 未测试的代码永远是错的, 过了测试的代码也不一定是对的, 暂且不提 runall 的 All Combo, 就光是从 dummy 到 bit 我的代码就撑不住了, 而且尝试了各种方法都找不到原因, 甚至还怀疑 Machine 有问题, 还愚蠢的问到了老师面前, 最后是重写

了一遍 (感谢分支技术和我备份的先见之明) 靠着 DiffTest 找到了是前面 shl 还有 sar 的移位的标志位出了问题, 靠着前面七零八落的 Debug 手段 (Log) 是真的完全找不出来

- ☑ 其实 Copy-Paste 是我以前经常干的事情, 主要是真的很想偷懒, 但是要注意的点太多了还不好 debug 所以这次,STFW 几个 printf 的含义作用以及参考了不少代码, 最终借助 vsprintf 避免了多与的 Copy-Paste 和 Debug
- ☑ 外设的 case 的实现不声明需要的 #define 也可以正常实现指导书要求的功能, 但是回归测试的时候会报错, 因为路径不一样了
- ☑ coremark 跑分的时候有段错误, 也是重新单个跑发现是 imul 的执行函数拼写问题
- ☑ 最早的几个测试样例就是哪里需要填指令就填哪里, 这导致后面重复的指令还要重写 exec.c, 挺浪费时间的, 所以从 fib 开始就一次性写完所有的指令,(虽然不用写完所有的也可以过测试就是了)

六、 思考题 && 必答题

(一) 思考题

1. 立即数背后的故事, 假设我们需要将 NEMU 运行在 Motorola 68k 的机器上 (把 NEMU 的源代码编译成 Motorola 68k 的机器码); 假设我们需要编写一个新的模拟器 NEMU-Motorola-68k, 模拟器本身运行在 x86 架构中, 但它模拟的是 Motorola 68k 程序的执行

   情况一要注意的问题：内存中读取到的指令可能不会被正确的 decode。原因：在大端机上, 数据在内存中以大端方式存储, 而 NEMU 是小端模式处理数据。解决方法：读内存之后对数据进行一定处理使数据符合小端模式的情景。

   情况二要注意的问题: 同上。原因: x86 机器中的数据在内存中是以小端方式存储的,NEMU-Motorola-68k 是以大端方式处理数据的。解决方法：同上以保证 NEMU-Motorola-68k 仍可以使用自己的规则来对数据进行操作。

2. 将运行时环境封装成库函数, 现在不就只有 NEMU 这一个机器吗 (m=1)? 哪里需要抽象?

   看 ics2019 就知道了, 作者想到了多指令集所以就需要了, 全都抽象成 ISA 的 API 调用, 而在运行的时候指定 ISA 就可以了

3. 堆和栈在哪里?

   为什么堆和栈的内容没有放入可执行文件里面?

   auto 类型的局部变量, 他们在运行时被分配在栈中, 所以他们既不在.data 节中体现, 也不在.bss 节中出现。

   那程序运行时刻用到的堆和栈又是怎么来的? AM 的代码是否能给你带来一些启发?

   堆的创建是在程序运行时, 用户栈总是从最大的合法用户地址开始 (低地址方向),NEMU 中的 loader.ld 文件 _stack_pointer = 0x7c00; 规定了栈的起始地址。在入口文件 start.S 中, 调用 mov 指令将栈的开始位置传给寄存器 esp。并在之后调用的 _trm_init 函数中, 对堆的起始地址进行初始化。

4. AT&T 格式反汇编结果中的少量指令, 与 i386 手册中列出的指令名称不符, 如 cltd. 除了 STFW 之外, 你有办法在手册中找到对应的指令吗? 如果有的话, 为什么这个办法是有效的呢?

先 STFW 这个指令的作用, 在手册中查找有类似关键词的。比如 cltd 是 Convert longword to doubleword 的意思, 翻手册的目录能找到意思差不多的 Convert word to double word。或者看 cltd 的源码找到类似指令。

5. 阅读相关 Makefile 和脚本文件, 尝试理解 AM 项目是如何生成 native 的可执行文件的.

   这一环套一环的 Makefile 看的人头晕,Makefile.compile 里规定了 native 的编译参数,Makefile.lib 里规定好路径还有 lib 文件, 在 terminal echo 编译

6. 为什么错误码是 1 呢? 你知道 make 程序是如何得到这个错误码的吗?

   string.c 里面有 nemu_assert 会使得程序中途出错的返回为 false 在 makefile 里面就会返回错误码 1

7. 作为一种基础设施, DiffTest 能帮助你节省多少调试的时间呢?

   这个问题我好有发言权, 就已经不是节省时间了, 有的 bug 不用 DiffTest 比对甚至找不到根源, 一点一点追根溯源的方法不仅不好找还浪费了好多时间,DiffTest 的优点就是他可以最快找到对应错误的位置!

8. 把思绪回归到 PA 中, 通用程序的性质告诉我们, NEMU 的潜力是无穷的. 为了创造出一个缤纷多彩的世界, 你觉得 NEMU 还缺少些什么呢?

   缺少异常和中断, 进程并行

9. 你或许会感到疑惑, 代码优化不是一件好事情吗? 为什么会有 volatile 这种奇葩的存在? 思考一下, 如果代码中 p 指向的地址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?

   编译器是根据输入输出状态进行的代码优化, 但作为人我们会考虑中间过程。如果代码中 p 指向的地址最终被映射到一个设备寄存器, 可能会发生设备状态的跳变, 比如显示器某块地方应该是由蓝变绿再变红, 会直接从蓝色变成红色。

10. 如何检测多个按键同时被按下?

    这个参考 navy-apps/pps/litenes/src/fce.c 里的 fce_run 中判断按键部分, 在外面再套一个 while1 死循环, 当判断为——KEY_NONE 时跳出, 否则就记录寄存器器状态, 并反馈给用户。

(二) 必答题

必答题除了最后一部分, 其余均为代码实现。

1. 编译与链接 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, 你可能会看到发生错误. 请分别解释为什么这些错误会发生/不发生? 你有办法证明你的想法吗?

   去掉 static 不会报错, 去掉之后就是默认的 extern 声明, 由于内存没有改变还是可以照样访问。

   去掉 inline 会报"defined but not used",inline 是把函数里面的代码直接插入到调用这个函数的地方, 而不是用调用函数的形式。如果函数体代码很短的话, 会更有效率。但是 inline 是给编译器的提示, 编译器会根据实际情况自己决定到底要不要进行内联, 如果函数过大、有函数指针指向这个函数或者有递归的情况下编译器都不会进行内联。所以去掉 inline 之后, 可能会因为没有引入该函数, 而报其他地方调用失败的错误。

都去掉会报”multiple definition of”,inline 相当于把一个函数固定在一个固定的内存里, 即使没有 static（不允许外部访问）但是由于内存没有改变所以还是能够访问。但是如果没有 inline 那就不能被外部访问了, 如果都去掉, 就会出现在两个内存中同时定义了这个函数, 所以出现了重定义。

2. 编译与链接

3. (a) 在 nemu/include/common.h 中添加一行 volatile static int dummy; 然后重新编译 NEMU. 请问重新编译后的 NEMU 含有多少个 dummy 变量的实体? 你是如何得到这个结果的?

   应该是一个吧。。

   (b) 添加上题中的代码后, 再在 nemu/include/debug.h 中添加一行 volatile static int dummy; 然后重新编译 NEMU. 请问此时的 NEMU 含有多少个 dummy 变量的实体? 与上题中 dummy 变量实体数目进行比较, 并解释本题的结果.

   两个, 两个文件各一个, 因为 static 只在当前文件起作用

   (c) 修改添加的代码, 为两处 dummy 变量进行初始化:volatile static int dummy = 0; 然后重新编译 NEMU. 你发现了什么问题? 为什么之前没有出现这样的问题? (回答完本题后可以删除添加的代码.)

   重定义报错, 之前没有初始化且定义两个静态变量 dummy, 他们只在各自模块被引用, 两个变量都有各自的分配空间, 并作为两个不同的符号记录到符号表中, 所以不会报错。但是初始化的值一样后就会报重定义的错。

4. 了解 Makefile

   读入 Makefile, 并且查看是否调用其他的 Makefile; 读取 include 文件的 Makefile; 初始化变量; 为目标文件建立依赖关系; 重新生成目标; 执行

七、 总结

PA2 一共多了 264 行代码, 相比于 PA1 的 394 行少了些, 大概原因是费时间的 exec 都不会增加新行吧.

40h 远远不止, 时间跨度大约一个月吧, 虽然中间空了 20 天因为 bug 什么也没干, 但是还是很消耗脑力, 代码能力重要, 更重要的是找对方向, 如果我早一点发现 DiffTest 的作用并且早一些时间, 也许中间空闲的 20 天就不会束手无策了,AM 那一章整体介绍都太长了, 所以 DiffTest 就没有仔细看 (非常后悔)。先看指导书, 多看几遍看懂！然后再去 RTFSC 也许效果更好, 太疲惫了, 希望 PA3 能长点教训。

另外, 并没有实现 PA1 许下的快一点的诺言。希望下次不用快一点, 别在 bug 上撞死就好。