



南开大学
Nankai University

南 开 大 学

计 算 机 学 院

计算机系统设计实验报告

PA2 简单复杂的机器：冯诺依曼计算机系统

蒋薇

年级：2021 级

专业：计算机科学与技术

2024 年 4 月 15 日

目录

一、 概述	1
(一) 实验目的	1
(二) 实验内容	1
二、 stage1	1
(一) 代码: 运行一个 C 程序	1
1. 准备	1
2. CALL	6
3. PUSH	7
4. POP	7
5. SUB	7
6. XOR	9
7. RET	10
8. ENDBR32	11
9. 运行	11
10. 堆和栈	11
三、 stage2 程序运行时环境与 AM	12
(一) 代码: 运行更多的程序	12
1. LEA	13
2. AND	13
3. PUSH	14
4. XCHG	14
5. SETCC	15
6. MOVZX	16
7. TEST	17
8. JCC	18
9. ADD	18
10. CMP	18
11. JMP	19
12. MULIMUL	19
13. DIVIDIV	19
14. ADCSBB	20
15. NEG	21
16. OR	22
17. NOT	22
18. DECINC	22

19.	CLTDCWTL	23
20.	LEAVE	24
21.	CALL 补充	24
22.	SARSALSHLSHR	25
(二)	基础设施	27
1.	代码: Differential Testing	27
2.	一键回归测试	28
四、	stage3	29
(一)	代码: 加入 IOE	29
1.	串口	30
2.	时钟	31
3.	键盘	34
4.	VGA	35
5.	problems	38
(二)	必答题	38
1.	static inline	38
2.	makefile	38
五、	实验总结与感想	39

一、 概述

(一) 实验目的

1. 熟悉取指-译码-执行的指令周期
2. 了解 NEMU 的框架代码是如何执行指令的, 包括其数据结构, 执行流程, 结构化程序设计, RTL(寄存器传输语言)
3. 实现更多新指令
4. 实现 IOE

(二) 实验内容

1. 运行第一个 C 程序, 在 NEMU 中通过 RTL 指令实现必要的指令, 实现成功后, 在 NEMU 中运行客户程序 dummy, 你将会看到 HIT GOOD TRAP 的信息.
2. 运行更多的程序, 实现更多的指令, 并设置一键测试
3. 了解并实现输入输出

二、 stage1

(一) 代码: 运行一个 C 程序

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。

在 nexus-am/tests/cputest 目录下键入

```
makeARCH=x86-nemuALL=dummyrun
```

编译 dummy 程序, 并启动 NEMU 运行它。

1. 准备

路径设置

```
exportNEMU_HOME = /mypath/nemu
```

```
exportAM_HOME = /mypath/nexus-am
```

```
exportNAVY_HOME = /mypath/navy-apps
```

RTL 语言

NEMU 使用 RTL(寄存器传输语言) 来描述 x86 指令的行为。这样做的好处是可以提高代码的复用率, 使得指令模拟的实现更加规整。同时 RTL 也可以作为一种 IR(中间表示) 语言, 将来可以很方便地引入即时编译技术对 NEMU 进行优化。

在 NEMU 中, **RTL 寄存器**有:

- 1.x86 的八个通用寄存器 (在 nemu/include/cpu/reg.h 中定义)
- 2.id_src, id_src2 和 id_dest 中的访存地址 addr 和操作数内容 val(在 nemu/include/cpu/decode.h 中定义)。从概念上看, 它们分别与 MAR 和 MDR 有异曲同工之妙
3. 临时寄存器 t0 t3(在 nemu/src/cpu/decode/decode.c 中定义)
- 4.0 寄存器 tzero(在 nemu/src/cpu/decode/decode.c 中定义), 它只能读出 0, 不能写入

RTL 基本指令包括:

1. 立即数读入 rtl_li

2. 算术运算和逻辑运算, 包括寄存器-寄存器类型 `rtl_(add|sub|and|or|xor|shl|shr|sar|slt|sltu)` 和立即数-寄存器类型 `rtl_(add|sub|and|or|xor|shl|shr|sar|slt|sltu)i`

3. 内存的访存 `rtl_lm` 和 `rtl_sm`

4. 通用寄存器的访问 `rtl_lr_(b|w|l)` 和 `rtl_sr_(b|w|l)`

RTL 伪指令:

1. 带宽度的通用寄存器访问 `rtl_lr` 和 `rtl_sr`

2. EFLAGS 标志位的读写 `rtl_set_(CF|OF|ZF|SF|IF)` 和 `rtl_get_(CF|OF|ZF|SF|IF)`

3. 其它常用功能, 如数据移动 `rtl_mv`, 符号扩展 `rtl_sext` 等

Opcode-table

`exec.c` 中间持有一个最关键的表格, 也就是 `opcode_table`。 `opcode_table` 是一个 `opcode_entry` 类型结构体的数组。

`opcode_entry` 中间持有两个函数指针和设置操作数长度的变量, 而函数指针就是处理一个 `opcode` 所需要执行的解码逻辑和执行逻辑。

`exec.c` 中间 `idex` 函数会调用 `opcode_entry` 中间两个函数指针, 来 `decode` 和 `execute` 指令, 框架已经完成了所有 `mov` 指令的处理, 我们需要完成其他的指令的填写。

Todo 相关函数

```

1 // 执行辅助函数 统一通过宏
2 // def\_EHelper (nemu/include/cpu/exec.h中定义)
3
4 //define make_EHelper(name) void concat(exec_, name) (vaddr_t *eip)
5 typedef void (*EHelper) (vaddr_t *);
6
7 // 执行辅助函数通过 RTL指令来描述指令真正的执行功能。
8
9 // 要在 nemu/src/cpu/exec/all-instr.h 里补全指令
10 make_EHelper(mov);
11 make_EHelper(call);
12 make_EHelper(push);
13 make_EHelper(pop);
14 make_EHelper(sub);
15 make_EHelper(xor);
16 make_EHelper(ret);
17 make_EHelper(operand_size);
18
19 // 在nemu/src/cpu/exec/exec.h 里对 opcode_entry opcode_table [512] 进行相应的
    更改
20 /* 0x30 */ IDEXW(G2E, xor, 1), IDEX(G2E, xor), IDEXW(E2G, xor, 1),
21 IDEX(E2G, xor),
22 /* 0x34 */ IDEXW(I2a, xor, 1), IDEX(I2a, xor), EMPTY, EMPTY,
23 /* 0x50 */ IDEX(r, push), IDEX(r, push), IDEX(r, push), IDEX(r, push),
24 /* 0x54 */ IDEX(r, push), IDEX(r, push), IDEX(r, push), IDEX(r, push),
25 /* 0x58 */ IDEX(r, pop), IDEX(r, pop), IDEX(r, pop), IDEX(r, pop),
26 /* 0x5c */ IDEX(r, pop), IDEX(r, pop), IDEX(r, pop), IDEX(r, pop),
27 /* 0xc0 */ IDEXW(gp2_lb2E, gp2, 1), IDEX(gp2_lb2E, gp2), EMPTY, EX(ret),
28 /* 0xe8 */ IDEX(J, call), IDEX(J, jmp), EMPTY, IDEXW(J, jmp, 1),
29

```

30 //ID 和 EX 分别表示了译码和执行两个阶段，W 表示需要取字。

EFLAGS

需实现 EFLAGS 寄存器, 需要在寄存器结构体中添加 EFLAGS 寄存器。

EFLAGS

```

1 //在nemu/include/cpu/reg.h 中先补充EFlags标志位
2 struct bs {
3 unsigned int CF:1;
4 unsigned int one:1;
5 unsigned int :4;
6 unsigned int ZF:1;
7 unsigned int SF:1; // bit 0 ~ 7
8 unsigned int :1;
9 unsigned int IF:1;
10 unsigned int :1;
11 unsigned int OF:1;
12 unsigned int :20;
13 } eflags;
14
15 // Eflags 将在 nemu/src/monitor/monitor.c 中进行初始化, 并在 nemu/src/cpu/
    arith.c 中进行
16 标志位的设置进行减法计算。
17
18 //补充 restart() 函数和 eflags_modify() 函数
19 //restart 函数使用 memcpy 将eflags设置为 0x00000002H
20 static inline void restart() {
21 /* Set the initial instruction pointer. */
22 cpu.eip = ENTRY_START;
23 unsigned int origin = 2;
24 memcpy(&cpu.eflags, &origin, sizeof(cpu.eflags));
25 #ifdef DIFF_TEST
26 init_qemu_reg();
27 #endif
28 }
29 //eflags_modify 为计算减法并设置 eflags 的值
30 static inline void eflags_modify() {
31 rtl_sub(&t2, &id_dest->val, &id_src->val);
32 rtl_update_ZFSF(&t2, id_dest->width);
33 rtl_sltu(&t0, &id_dest->val, &id_src->val);
34 rtl_set_CF(&t0);
35 rtl_xor(&t0, &id_dest->val, &id_src->val);
36 rtl_xor(&t1, &id_dest->val, &t2);
37 rtl_and(&t0, &t0, &t1);
38 rtl_msb(&t0, &t0, id_dest->width);
39 rtl_set_OF(&t0);
40 }
41
42 //实现Dophelper调用取指进行取字

```

```

43 static inline make_DopHelper(SI) {
44     assert(op->width == 1 || op->width == 4);
45     op->type = OP_TYPE_IMM;
46     /* TODO: Use instr_fetch() to read op->width' bytes of memory* pointed by
        eip'. Interpret the result as a signed immediate,
47     * and assign it to op->simm.
48     *
49     op->simm = ???
50     */
51     /*TODO();
52     op -> simm = instr_fetch(eip, op -> width);
53     if(op -> width == 1) {
54     op -> simm = (int8_t)op -> simm;
55     }
56     rtl_li(&op->val, op->simm);
57 #ifdef DEBUG
58     snprintf(op->str, OP_STR_SIZE, "$0x%x", op->simm);
59 #endif
60 }

```

pushpop

pushpop

```

1 //push主要作用是修改栈顶，将指针src1的内容写入栈
2 static inline void rtl_push(const rtlreg_t* src1) {
3 // esp <- esp - 4
4 // M[esp] <- src1
5 //TODO();
6 rtl_subi(&cpu.esp, &cpu.esp, 4);
7 rtl_sm(&cpu.esp, 4, src1);
8 }
9
10 //pop函数是将rtl_pop读取的数据写入通用寄存器中。
11 static inline void rtl_pop(rtlreg_t* dest) {
12 // dest <- M[esp]
13 // esp <- esp + 4
14 // TODO();
15 rtl_lm(dest, &cpu.esp, 4);
16 rtl_addi(&cpu.esp, &cpu.esp, 4);
17 }

```

更新标识符

更新标识符

```

1 static inline void rtl_eq0(rtlreg_t* dest, const rtlreg_t* src1) {
2 // dest <- (src1 == 0 ? 1 : 0)
3 // TODO();
4 rtl_sltui(dest, src1, 1);
5 }

```

```

6 static inline void rtl_eqi(rtlreg_t* dest, const rtlreg_t* src1, int imm) {
7 // dest <- (src1 == imm ? 1 : 0)
8 // TODO();
9 rtl_xori(dest, src1, imm);
10 rtl_eq0(dest, dest);
11 }
12 static inline void rtl_neq0(rtlreg_t* dest, const rtlreg_t* src1) {
13 // dest <- (src1 != 0 ? 1 : 0)
14 // TODO();
15 rtl_eq0(dest, src1);
16 rtl_eq0(dest, dest);
17 }
18 static inline void rtl_msb(rtlreg_t* dest, const rtlreg_t* src1, int width) {
19 // dest <- src1[width * 8 - 1]
20 //TODO();
21 rtl_shri(dest, src1, width*8-1);
22 rtl_andi(dest, dest, 0x1);
23 }
24
25 //根据提示更新 ZF 位
26 static inline void rtl_update_ZF(const rtlreg_t* result, int width) {
27 // eflags.ZF <- is_zero(result[width * 8 - 1 .. 0])
28 //TODO();
29 rtl_andi(&t0, result, (0xffffffff >> (4-width)*8));
30 rtl_eq0(&t0, &t0);
31 rtl_set_ZF(&t0);
32 }
33
34 //更新 SF 位
35 static inline void rtl_update_SF(const rtlreg_t* result, int width) {
36 // eflags.SF <- is_sign(result[width * 8 - 1 .. 0])
37 // TODO();
38 assert(result != &t0);
39 rtl_msb(&t0, result, width);
40 rtl_set_SF(&t0);
41 }
42 static inline void rtl_update_ZFSF(const rtlreg_t* result, int width) {
43 rtl_update_ZF(result, width);
44 rtl_update_SF(result, width);
45 }
46
47 //EFlags 寄存器标志位的读写函数
48 #define make_rtl_arith_logic(name) \
49 static inline void concat(rtl_, name) (rtlreg_t* dest, const rtlreg_t* src1, \
50 const rtlreg_t* src2) { \
51 *dest = concat(c_, name) (*src1, *src2); \
52 } \
53 static inline void concat3(rtl_, name, i) (rtlreg_t* dest, const rtlreg_t*

```



```

54 | src1, int imm) { \
55 | *dest = concat(c_, name) (*src1, imm); \
56 | }

```

2. CALL

call:call 指令有很多形式, 不过在 PA 中只会用到其中的几种, 现在只需要实现 CALL rel32 的形式就可以了。

%eip 的跳转可以通过将 decoding.is_jump 设为 1, 并将 decoding.jump_eip 设为跳转目标地址来实现, 这时在 update_eip() 函数中会把跳转目标地址作为新的 %eip, 而不是顺序意义下的下一条指令的地址。

在 nexus-am/tests/cputest 目录下键入 make ARCH=x86-nemu ALL=dummy run, 启动 NEMU 运行。

通过/nexus-am/tests/cputest/build 目录下的 dummy-x86-nemu.txt, 可知在 eip=0x0010000a 这个位置有指令没有实现, 操作码 e8 的指令没有实现。

我们寻找 e8 对应指令, A 的含义是地址, v 为单字或双字, 取决于操作空间的大小, 这就是我们调用函数时使用的 call 指令。

Op	Op	Op	Op	Op	Op
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	cd	CALL ptr16:32	17+m, pm=52+m	Call gate, same privilege	

Operation

```

IF rel16 or rel32 type of call
THEN (* near relative call *)
  IF OperandSize = 16
  THEN
    Push(IP);
    EIP ← (EIP + rel16) AND 0000FFFFH;
  ELSE (* OperandSize = 32 *)
    Push(EIP);
    EIP ← EIP + rel32;
  FI;
FI;

```

图 1: CALL

CALL

```

1 | //call指令读取需要压栈的EIP值, 用rtl_push压栈, 最后设置跳转
2 | //all-instr.h 和 opcode_table已修改, 要在 control.c 文件中具体实现
   | make_EHelper(call)
3 | make_EHelper(call) {
4 | // the target address is calculated at the decode stage
5 | //TODO();
6 | rtl_li(&t2, decoding.seq_eip);
7 | rtl_push(&t2);
8 | decoding.is_jump = 1;
9 | print_asm("call_%x", decoding.jump_eip);
10 | }

```

```

11 // call 是一个J形指令，需要编写 decode.c 中的译码函数 make_DHelper(J) ，在里面
    调用
12 decode_op_SI(eip, id_dest, false); 来实现立即数的读取，并更新 jmp_eip 。
13 make_DHelper(J) {
14     decode_op_SI(eip, id_dest, false);
15     // the target address can be computed in the decode stage
16     decoding.jmp_eip = id_dest->siml + *eip;
17 }

```

3. PUSH

需要实现 PUSH r32 的形式, 通过 rtl_push 来实现。

PUSH

```

1 //修改 all-instr.h 和 opcode_table
2 // push 是数据移动指令，需要在 data-mov.c 中实现,调用已实现的 rtl_push 执行函
    数写进栈中
3 make_EHelper(push) {
4     //TODO();
5     rtl_push(&id_dest -> val);
6     print_asm_template1(push);
7 }

```

4. POP

需要实现 POP r32 的形式, 通过 rtl_pop 来实现。

POP

```

1 //调用 rtl_pop 执行函数进入读栈，并把读取的数据写到通用寄存器中
2 //pop 也是数据移动指令，需要在 data-mov.c 中实现
3 make_EHelper(pop) {
4     // TODO();
5     rtl_pop(&t2);
6     operand_write(id_dest, &t2);
7     print_asm_template1(pop);
8 }

```

5. SUB

EFLAGS 是一个 32 位寄存器, 在 NEMU 中, 我们只会用到 EFLAGS 中以下的 5 个位:CF,ZF,SF,IF,OF

添加 EFLAGS 寄存器需要用到结构体的位域 (bit field) 功能

关于 EFLGAS 的初值, 我们遵循 i386 手册中提到的约定, 在 i386 手册的第 10 章中找到这一初值, 然后在 restart() 函数中对 EFLAGS 寄存器进行初始化. 实现了 EFLAGS 寄存器之后, 再实现相关的 RTL 指令, 之后就可以通过这些 RTL 指令来实现 sub 指令

在 sub 指令中, OF/SF/ZF/AF/PF/CF 标志位受到影响. 各标志位的作用如下所示:

CF(bit 0) [进位标志]: 若算术操作产生的结果在最高有效位 (most-significant bit) 发生进位或借位则将其置 1, 反之清零。这个标志指示无符号整型运算的溢出状态。

PF(bit 4) [奇偶标志]: PF 标志位于 EFLAGS 寄存器的第 2 位, 用于表示结果的二进制表示中 1 的个数是否为偶数。对于 sub 指令来说, 当结果的二进制表示中 1 的个数为偶数时, PF 被设置为 1, 否则为 0。

AF(bit 4) [辅助进位标志]: AF 标志位于 EFLAGS 寄存器的第 4 位, 用于表示低 4 位的进位或借位情况。当低 4 位有进位或借位时, AF 被设置为 1, 否则为 0。

ZF(bit 6) [零标志]: 当结果为零时, ZF 被设置为 1, 否则为 0。

SF(bit 7) [符号标志]: 该标志被设置为有符号整型的最高有效位。当结果为负数时, SF 被设置为 1, 否则为 0。

OF(bit 11) [溢出标志]: 如果整型结果是较大的正数或较小的负数, 并且无法匹配目的操作数时将该位置 1, 反之清零。用于表示有符号数的运算结果是否超出了其数据类型所能表示的范围。

INTEL 80386 PROGRAMMER'S REFERENCE MANUAL 1986

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.

Operation

```
IF SRC is a byte and DEST is a word or dword
THEN DEST = DEST - SignExtend(SRC);
ELSE DEST ← DEST - SRC;
FI;
```

图 2: Sub

ib : 表示 8 位立即数

iw : 表示 16 位立即数

id : 表示 32 位立即数

/r : 表示后面有一个 ModR/M 字节, 且其中的 reg/opcode 字段被解释为寄存器编码

/digit : digit 为 0~7 中一个数字(/0), 表示操作码后跟一个 ModR/M 字节, 并且 reg/opcode 字段被解释为扩展 opcode, 取值为 digit。

Sub

```
1 //修改 arith.c 中的函数
2 make_EHelper(sub) {
3     eflags_modify();
4     operand_write(id_dest, &t2);
5     print_asm_template2(sub);
6 }
7 //填写 opcode_table: opcode= 0x2D , 译码函数: //make_DHelper(I2a) , 执行函
   数:
8 //make_EHelper(sub) , 操作数宽度为 2 或 4, 定义为 IDEX(I2a,sub)
9 /* 0x2c */ IDEXW(I2a, sub, 1), IDEX(I2a, sub), EMPTY, EMPTY,
10
```

```

11 //sub 指令具有较多的不同形式，由于其执行阶段相同，译码函数不同。在实现执行函
    数后，只需再根据
12 不同形式设定译码函数和操作数宽度。
13
14 //0x29/r 译码函数：G2E，定义为 IDEX(G2E,sub)
15 //0x2B/r 译码函数：E2G，定义为 IDEX(E2G,sub)
16 /* 0x28 */ EMPTY, IDEX(G23, sub), EMPTY, IDEX(E2G, sub),
17
18 //0x80、0x81 与 0x83 的 sub 需要进行 opcode 的扩展。
19 //查看 make_group 函数，该函数用于处理扩展 opcode 的情况。
20 //在 opcode_table_gp 中存储各 opcode_entry，并在运行 make_EHelper(gp) 时使用
    index(eip,
21 opcode_table_gp1[decoding.ext_opcode]) 作为进一步的译码-执行函数。
22 //因为 sub 的 ext_opcode=5，所以在 opcode_table_gp1[5] 处填写 EX(sub) 执行函
    数
23 /* 0x80, 0x81, 0x83 */
24 make_group(gp1,
25 EX(add), EX(or), EX(adc), EX(sbb),
26 EX(and), EX(sub), EX(xor), EX(cmp))

```

6. XOR

INTEL 80386 PROGRAMMER'S REFERENCE MANUAL 1986

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 iw	XOR r/m32,imm8	2/7	XOR sign-extended immediate word 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

Operation

DEST ← LeftSRC XOR RightSRC
 CF ← 0
 OF ← 0

Description

XOR computes the exclusive OR of the two operands. Each bit of the result is 1 if the corresponding bits of the operands are different; each bit is 0 if the corresponding bits are the same. The answer replaces the first operand.

Flags Affected

CF = 0, OF = 0; SF, ZF, and PF as described in Appendix C; AF is undefined

图 3: Xor

Xor

```

1 //首先修改 all-instr.h
2 make_EHelper(xor);
3 //后修改 opcode_table[]
4 /* 0x30 */ IDEXW(G2E, xor, 1), IDEX(G2E, xor),
5 IDEXW(E2G, xor, 1), IDEX(E2G, xor),

```

```

6  /* 0x34 */ IDEXW(I2a, xor, 1), IDEX(I2a, xor), EMPTY,
7  EMPTY,
8  //xor 是逻辑运算指令, 需要在 logic.c 中实现
9  make_EHelper(xor) {
10 // TODO();
11 rtl_xor(&t2, &id_dest -> val, &id_src -> val);
12 operand_write(id_dest, &t2);
13 rtl_update_ZFSF(&t2, id_dest -> width);
14 rtl_set_CF(&tzero);
15 rtl_set_OF(&tzero);
16 print_asm_template2(xor);
17 }

```

7. RET

RET — Return from Procedure

Opcode	Instruction	Clocks	Description
CB	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

Operation

```

IF instruction = near RET
THEN;
  IF OperandSize = 16
  THEN
    IP ← Pop();
    EIP ← EIP AND 0000FFFFH;
  ELSE (* OperandSize = 32 *)
    EIP ← Pop();
  FI;
IF instruction has immediate operand THEN ESP ← ESP + imm16; FI;
FI;

IF (PE = 0 OR (PE = 1 AND VM = 1))
(* real mode or virtual 8086 mode *)
AND instruction = far RET
THEN;
  IF OperandSize = 16
  THEN
    IP ← Pop();
    EIP ← EIP AND 0000FFFFH;
    CS ← Pop(); (* 16-bit pop *)
  ELSE (* OperandSize = 32 *)
    EIP ← Pop();
    CS ← Pop(); (* 32-bit pop, high-order 16-bits discarded *)
  FI;
IF instruction has immediate operand THEN ESP ← ESP + imm16; FI;
FI;

```

图 4: Ret

Ret

```

1  //修改 all-instr.h
2  make_EHelper(ret);
3  //修改 opcode_table[]
4  /* 0xc0 */ IDEXW(gp2_Ib2E, gp2, 1), IDEX(gp2_Ib2E, gp2), EMPTY, EX(ret),
5  //Ret是逻辑运算指令, 需要在 logic.c 中实现
6  make_EHelper(ret) {
7  // TODO();
8  rtl_pop(&t2);
9  decoding.jmp_eip = t2;
10 decoding.is_jmp = 1;
11 print_asm("ret");
12 }

```

8. ENDBR32

ENDBR32 — Terminate an Indirect Branch in 32-bit and Compatibility Mode

Opcode/Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 1E FB ENDBR32	ZO	V/V	CET_IBT	Terminate indirect branch in 32 bit and compatibility mode.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA	NA

Description

Terminate an indirect branch in 32 bit and compatibility mode.

图 5: ENDBR32

Endbr32

```

1 //该指令由三条指令组成，所以PC = PC + 3
2 //在 all-instr.h 中添加
3 make_EHelper(endbr32)
4 //修改 opcode_table[]
5 /* 0xf0 */ EMPTY, EMPTY, EMPTY, EXW(endbr, 3),
6 //在 special.c 中实现
7 make_EHelper(endbr32)
8 {
9     decinfo.seq_pc += 3;
10    print_asm("endbr32");
11 }

```

9. 运行

```

[src/monitor/monitor.c,65,load_img] The image is /mnt/hgfs/sharing/ICS/nexus-am/
tests/cputest/build/dummy-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 11:01:56, Apr 10 2024
For help, type "help"
(nemu) c
nemu: HIT GOOD TRAP at eip = 0x00100052
(nemu)

```

图 6: 运行结果

10. 堆和栈

堆和栈在哪里？

我们知道代码和数据都在可执行文件里面，但却没有提到堆 (heap) 和栈 (stack). 为什么堆和栈的内容没有放入可执行文件里面？那程序运行时刻用到的堆和栈又是怎么来的？AM 的代码是否能给你带来一些启发？

答：堆和栈都是程序运行时的动态内存分配，它们不会在可执行文件中存储，因为它们的大小和内容在程序运行时才能确定。程序运行时需要用到堆和栈，所以它们被分配在程序的运行时内存中。

栈是一种自动分配和释放的内存，用于存储程序的局部变量、函数参数和函数返回值等。当程序运行到一个函数时，会在栈中为该函数分配一块内存空间，当函数返回时，这块内存会被自动释放。栈的大小由系统自动管理，并根据程序的需要进行动态分配和释放。

堆是另一种动态内存分配的机制，程序可以通过调用库函数（如 `malloc` 和 `free`）来在堆中动态地分配和释放内存。堆的大小由程序员控制，但是由于堆中的内存需要手动释放，所以需要特别注意内存泄漏等问题。

当程序运行时，操作系统会为程序分配一块虚拟内存，其中包括可执行文件中的代码和数据，以及堆和栈。当程序需要访问堆和栈时，操作系统会根据程序的需求在虚拟内存中分配相应的空间，并将其映射到物理内存中。这样，程序就可以访问堆和栈了。

三、 stage2 程序运行时环境与 AM

AM 在概念上定义了一台抽象计算机，它从运行程序的视角刻画了一台计算机应该具备的功能，而真机和 NEMU 都是这台抽象计算机的具体实现，只是真机是通过物理上存在的数字电路来实现，NEMU 是通过程序来实现。

AM 是个抽象类，真机和虚拟机是由 AM 这个抽象类派生出来的两个子类，而 x86 真机和 NEMU 则分别是这两个子类的实例化

（一） 代码：运行更多的程序

为了让 NEMU 支持大部分程序的运行，你还需要实现更多的指令： Data Movement Instructions: `mov`, `push`, `pop`, `leave`, `cld`(在 i386 手册中为 `cdq`), `movsx`, `movzx`

Binary Arithmetic Instructions: `add`, `inc`, `sub`, `dec`, `cmp`, `neg`, `adc`, `sbb`, `mul`, `imul`, `div`, `idiv`

Logical Instructions: `not`, `and`, `or`, `xor`, `sal`(`shl`), `shr`, `sar`, `setcc`, `test`

Control Transfer Instructions: `jmp`, `jcc`, `call`, `ret`

Miscellaneous Instructions:

我们让 AM 项目上的程序默认编译到 x86-nemu 的 AM 中

ARCH ?= x86-nemu

ARCHS = \$(shell ls \$(A_M_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)

添加需要补充的指令

```
1  make_EHelper(add);
2  make_EHelper(inc);
3  make_EHelper(dec);
4  make_EHelper(cmp);
5  make_EHelper(neg);
6  make_EHelper(adc);
7  make_EHelper(sbb);
8  make_EHelper(mul);
9  make_EHelper(imul1);
10 make_EHelper(imul2);
11 make_EHelper(imul3);
12 make_EHelper(div);
```

```

13 make_EHelper(idiv);
14 make_EHelper(not);
15 make_EHelper(and);
16 make_EHelper(or);
17 make_EHelper(xor);
18 make_EHelper(sal);
19 make_EHelper(shl);
20 make_EHelper(shr);
21 make_EHelper(sar);
22 make_EHelper(rol);
23 make_EHelper(setcc);
24 make_EHelper(test);
25 make_EHelper(leave);
26 make_EHelper(cld);
27 make_EHelper(cwtl);
28 make_EHelper(movsx);
29 make_EHelper(movzx);
30 make_EHelper(jmp);
31 make_EHelper(jmp_rm);
32 make_EHelper(jcc);
33 make_EHelper(lea);
34 make_EHelper(nop);
35 make_EHelper(in);
36 make_EHelper(out);
37 make_EHelper(lidt);
38 make_EHelper(int);
39 make_EHelper(pusha);
40 make_EHelper(popa);
41 make_EHelper(iret);
42 make_EHelper(mov_store_cr);

```

1. LEA

Lea

```

1 //根据opcode修改 exec.c
2 /* 0x8c */ EMPTY, IDEX(lea_M2G, lea), EMPTY, IDEX(E, pop),

```

2. AND

AND

```

1 //在i386手册中为CBW和CWD:
2 //1. CBW: AL符号位扩展至AH
3 //2. CWD: AX的符号位扩展至DX
4 // logic.c
5 make_EHelper(and) {
6 // TODO();

```



```

7 | rtl_and(&t2,&id_dest->val,&id_src->val);
8 | operand_write(id_dest,&t2);
9 | rtl_update_ZFSF(&t2,id_dest->width);
10 | rtl_set_CF(&tzero);
11 | rtl_set_OF(&tzero);
12 | print_asm_template2(and);
13 | }
14 | // exec.c
15 | /* 0x80, 0x81, 0x83 */
16 | make_group(gp1,
17 | EX(add), EX(or), EX(adc), EX(sbb),
18 | EX(and), EX(sub), EX(xor), EX(cmp))
19 | /* 0x20 */ IDEXW(G2E, and, 1), IDEX(G2E, and), IDEXW(E2G, and, 1),
20 | IDEX(E2G, and),
21 | /* 0x24 */ IDEXW(I2a, and, 1), IDEX(I2a, and), EMPTY, EMPTY,

```

3. PUSH

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 + /r	PUSH r16	2	Push register word
50 + /r	PUSH r32	2	Push register dword
6A	PUSH imm8	2	Push immediate byte
68	PUSH imm16	2	Push immediate word
68	PUSH imm32	2	Push immediate dword
0E	PUSH CS	2	Push CS
16	PUSH SS	2	Push SS
1E	PUSH DS	2	Push DS
06	PUSH ES	2	Push ES
0F A0	PUSH FS	2	Push FS
0F A8	PUSH GS	2	Push GS

Operation

```

IF StackAddrSize = 16
THEN
  IF OperandSize = 16 THEN
    SP ← SP - 2;
    (SS:SP) ← (SOURCE); (* word assignment *)
  ELSE
    SP ← SP - 4;
    (SS:SP) ← (SOURCE); (* dword assignment *)
  FI;
ELSE (* StackAddrSize = 32 *)

```

图 7: push

pushpushl

```

1 | //push
2 | /* 0x68 */ IDEX(I, push), IDEX(I_E2G, imul3), IDEXW(push_SI, push, 1),
3 | IDEX(SI_E2G, imul3),
4 | /* 0xff */
5 | make_group(gp5,
6 | EX(inc), EX(dec), EX(call_rm), EMPTY,
7 | EX(jmp_rm), EMPTY, EX(push), EMPTY)

```

4. XCHG

Xchg

```
1 //nop实际上什么都不做,其opcode为90
2 /* 0x90 */ EX(nop), EMPTY, EMPTY, EMPTY,
```

5. SETCC

SETcc 指令是一种根据条件设置一个字节数据值的指令，其中 cc 表示条件代码，SETcc 指令会根据特定的条件设置一个字节的数值为 1 或 0

SETcc 指令的操作数是一个寄存器或内存位置，指令会将 1 或 0 写入该位置，其中 cc 表示条件代码，每个条件代码都有特定的作用，可以用于执行基于条件的操作或控制流。SETcc 指令通常与 CMP 或 TEST 等比较或测试指令配合使用，用于根据比较结果设置一个字节的数值。

SETcc — Byte Set on Condition

Opcode	Instruction	Clocks	Description
0F 97	SETA r/m8	4/5	Set byte if above (CF=0 and ZF=0)
0F 93	SETAB r/m8	4/5	Set byte if above or equal (CF=0)
0F 92	SETB r/m8	4/5	Set byte if below (CF=1)
0F 96	SETBE r/m8	4/5	Set byte if below or equal (CF=1 or (ZF=1)
0F 92	SETC r/m8	4/5	Set if carry (CF=1)
0F 94	SETE r/m8	4/5	Set byte if equal (ZF=1)
0F 9F	SETG r/m8	4/5	Set byte if greater (ZF=0 or SF=OF)
0F 9D	SETGE r/m8	4/5	Set byte if greater or equal (SF=OF)
0F 9C	SETL r/m8	4/5	Set byte if less (SF=OF)
0F 9E	SETLE r/m8	4/5	Set byte if less or equal (ZF=1 and SF=OF)
0F 96	SETNA r/m8	4/5	Set byte if not above (CF=1)
0F 92	SETNAB r/m8	4/5	Set byte if not above or equal (CF=1)
0F 93	SETNB r/m8	4/5	Set byte if not below (CF=0)
0F 97	SETNBE r/m8	4/5	Set byte if not below or equal (CF=0 and ZF=0)
0F 93	SETNC r/m8	4/5	Set byte if not carry (CF=0)
0F 95	SETNE r/m8	4/5	Set byte if not equal (ZF=0)
0F 9E	SETNG r/m8	4/5	Set byte if not greater (ZF=1 or SF=OF)
0F 9C	SETNGE r/m8	4/5	Set if not greater or equal (SF=OF)
0F 9D	SETNL r/m8	4/5	Set byte if not less (SF=OF)
0F 9F	SETNLE r/m8	4/5	Set byte if not less or equal (ZF=1 and SF=OF)
0F 91	SETNO r/m8	4/5	Set byte if not overflow (OF=0)
0F 9B	SETNP r/m8	4/5	Set byte if not parity (PF=0)
0F 99	SETNS r/m8	4/5	Set byte if not sign (SF=0)
0F 95	SETNZ r/m8	4/5	Set byte if not zero (ZF=0)
0F 90	SETO r/m8	4/5	Set byte if overflow (OF=1)
0F 9A	SETP r/m8	4/5	Set byte if parity (PF=1)
0F 9A	SETPE r/m8	4/5	Set byte if parity even (PF=1)
0F 9B	SETPO r/m8	4/5	Set byte if parity odd (PF=0)
0F 98	SETS r/m8	4/5	Set byte if sign (SF=1)
0F 94	SETZ r/m8	4/5	Set byte if zero (ZF=1)

Operation

IF condition THEN r/m8 ← 1 ELSE r/m8 ← 0; FI;

图 8: Setcc

Setcc

```
1 /* 0xf6, 0xf7 */
2 make_group(gp3,
3 IDEX(test_I, test), EMPTY, EX(not), EX(neg),
4 EX(mul), EX(imul), EX(div), EX(idiv))
5 //0F时涉及到两字节opcode。程序先在 0x0F 处执行 make_EHelper(2byte_esc)，在该
   函数中再确定其
6 正确的两字节opcode编码，并调用 idex(eip, &opcode_table[opcode]) 进行指令的进
   一步译码与执
7 行。
8 /* 0x90 */ IDEXW(E, setcc, 1), IDEXW(E, setcc, 1), IDEXW(E, setcc, 1),
9 IDEXW(E, setcc, 1),
10 /* 0x94 */ IDEXW(E, setcc, 1), IDEXW(E, setcc, 1), IDEXW(E, setcc, 1),
11 IDEXW(E, setcc, 1),
12 /* 0x98 */ IDEXW(E, setcc, 1), IDEXW(E, setcc, 1), IDEXW(E, setcc, 1),
13 IDEXW(E, setcc, 1),
14 /* 0x9c */ IDEXW(E, setcc, 1), IDEXW(E, setcc, 1), IDEXW(E, setcc, 1),
15 IDEXW(E, setcc, 1),
```

```

16 //还需实现 cc.c 中的 rtl_setcc 函数
17 // TODO: Query EFLAGS to determine whether the condition code is satisfied.
18 // dest <- ( cc is satisfied ? 1 : 0)
19 switch (subcode & 0xe) {
20 case CC_O:
21   rtl_get_OF(dest);
22   break;
23 case CC_B:
24   rtl_get_CF(dest);
25   break;
26 case CC_E:
27   rtl_get_ZF(dest);
28   break;
29 case CC_BE:
30   assert(dest!=&t0);
31   rtl_get_CF(dest);
32   rtl_get_ZF(&t0);
33   rtl_or(dest,dest,&t0);
34   break;
35 case CC_S:
36   rtl_get_SF(dest);
37   break;
38 case CC_L:
39   assert(dest!=&t0);
40   rtl_get_SF(dest);
41   rtl_get_OF(&t0);
42   rtl_xor(dest,dest,&t0);
43   break;
44 case CC_LE:
45   assert(dest!=&t0);
46   rtl_get_SF(dest);
47   rtl_get_OF(&t0);
48   rtl_xor(dest,dest,&t0);
49   rtl_get_ZF(&t0);
50   rtl_or(dest,dest,&t0);
51   break;
52 default:
53   panic("should_not_reach_here");
54 case CC_P:
55   panic("n86_does_not_have_PF");
56 }

```

6. MOVZX

Movzx

```

1 //id_src 和 id_dest 两者宽度不同，在进入函数的时候会重新调整 id_dest 的宽度。
  在

```

```

2 填写译码函数的时候按照 id_src 的宽度填写即可。
3  /* 0xb4 */ EMPTY, EMPTY, IDEXW(mov_E2G, movzx, 1), IDEXW(mov_E2G, movzx,
4  2),
5  /* 0xb8 */ EMPTY, EMPTY, EMPTY, EMPTY,
6  /* 0xbc */ EMPTY, EMPTY, IDEXW(mov_E2G, movsx, 1), IDEXW(mov_E2G, movsx,
7  2),
8  //data-mov.c
9  make_EHelper(movsx) {
10 id_dest->width = decoding.is_operand_size_16 ? 2 : 4;
11 rtl_sext(&t2, &id_src->val, id_src->width);
12 operand_write(id_dest, &t2);
13 print_asm_template2(movsx);
14 }
15 make_EHelper(movzx) {
16 id_dest->width = decoding.is_operand_size_16 ? 2 : 4;
17 operand_write(id_dest, &id_src->val);
18 print_asm_template2(movzx);
19 }

```

7. TEST

TEST 指令是 x86 汇编语言中的一种逻辑操作指令，用于将两个操作数进行按位逻辑与运算，并设置标志位以反映结果。

TEST 指令会设置 CF 和 OF 标志位为 0，SF、ZF 和 PF 标志位根据运算结果而定。如果结果为 0，则 ZF 被设置为 1，否则 ZF 被清零；如果结果中 1 的个数为偶数，则 PF 被设置为 1，否则 PF 被清零；如果结果的最高位为 1，则 SF 被设置为 1，否则 SF 被清零。

Test

```

1  // logic.c
2  make_EHelper(test) {
3  // TODO();
4  rtl_and(&t2, &id_dest->val, &id_src->val);
5  rtl_update_ZFSF(&t2, id_dest->width);
6  rtl_set_CF(&tzero);
7  rtl_set_OF(&tzero);
8  print_asm_template2(test);
9  }
10 //扩展 opcode 时，F6、F7 处使用了 IDEXW(E, gp3, 1) 与 IDEX(E, gp3)。这里的 E 只
    从 ModR/M 中读
11 取了 r/m 的信息（即 LeftSRC），另一 RightSRC 为立即数，需要在 gp3[0] 的译码
    中进行读取。故定
12 义为 IDEX(test_I, test)
13  /* 0xf6, 0xf7 */
14  make_group(gp3,
15  IDEX(test_I, test), EMPTY, EX(not), EX(neg),
16  EX(mul), EX(imul1), EX(div), EX(idiv))

```

8. JCC

有条件跳转指令,Jcc 指令会检查相应的标志位,如果标志位符合指定的条件,则跳转到目标地址;否则,继续执行下一条指令。Jcc 指令的执行跳转范围为有符号的 8 位偏移量,可以用于短跳转和条件循环等操作。

JCC

```

1 //1 byte table
2 /* 0x70 */ IDEXW(J, jcc, 1), IDEXW(J, jcc, 1), IDEXW(J, jcc, 1), IDEXW(J,
3 jcc, 1),
4 /* 0x74 */ IDEXW(J, jcc, 1), IDEXW(J, jcc, 1), IDEXW(J, jcc, 1), IDEXW(J,
5 jcc, 1),
6 /* 0x78 */ IDEXW(J, jcc, 1), IDEXW(J, jcc, 1), IDEXW(J, jcc, 1), IDEXW(J,
7 jcc, 1),
8 /* 0x7c */ IDEXW(J, jcc, 1), IDEXW(J, jcc, 1), IDEXW(J, jcc, 1), IDEXW(J,
9 jcc, 1),
10 //2 byte table
11 /* 0x80 */ IDEX(J, jcc), IDEX(J, jcc), IDEX(J, jcc), IDEX(J, jcc),
12 /* 0x84 */ IDEX(J, jcc), IDEX(J, jcc), IDEX(J, jcc), IDEX(J, jcc),
13 /* 0x88 */ IDEX(J, jcc), IDEX(J, jcc), IDEX(J, jcc), IDEX(J, jcc),
14 /* 0x8c */ IDEX(J, jcc), IDEX(J, jcc), IDEX(J, jcc), IDEX(J, jcc),

```

9. ADD

Add

```

1 //类似Sub, 要修改CF和OF的判别
2 // arith.c
3 make_EHelper(add) {
4 // TODO();
5 rtl_add(&t2, &id_dest->val, &id_src->val);
6 operand_write(id_dest, &t2);
7 rtl_update_ZFSF(&t2, id_dest->width);
8 rtl_sltu(&t0, &t2, &id_dest->val);
9 rtl_set_CF(&t0);
10 rtl_xor(&t0, &id_src->val, &t2);

```

10. CMP

Cmp

```

1 //类似Sub, 不存储结果, 只修改EFLAGS的标志
2 //arith.c
3 make_EHelper(cmp) {
4 // TODO();
5 eflags_modify();
6 print_asm_template2(cmp);
7 }

```

```

8 // exec.c
9 /* 0x38 */ IDEXW(G2E, cmp, 1), IDEX(G2E, cmp), IDEXW(E2G, cmp, 1),
10 IDEX(E2G, cmp),
11 /* 0x3c */ IDEXW(I2a, cmp, 1), IDEX(I2a, cmp), EMPTY, EMPTY,
12 /* 0x80, 0x81, 0x83 */
13 make_group(gp1,
14 EX(add), EX(or), EX(adc), EX(sbb),
15 EX(and), EX(sub), EX(xor), EX(cmp))

```

11. JMP

Jmp

```

1 //make_Ehelper(jmp) 与 make_Ehelper(jmp_rm) 的区别在于 jmp_eip 的计算方法，前
   者根据偏移量
2 计算 eip，后者根据寄存器取值
3 //填写opcode表
4 /* 0x80 */ IDEXW(I2E, gp1, 1), IDEX(I2E, gp1), EMPTY, IDEX(SI2E, gp1),

```

12. MULIMUL

MULIMUL

```

1 /* 0xf6, 0xf7 */
2 make_group(gp3,
3 IDEX(test_I, test), EMPTY, EX(not), EX(neg),
4 EX(mul), EX(imul1), EX(div), EX(idiv))

```

13. DIVIDIV

MULIMUL

```

1 //DIV 用于无符号整数除法，而 IDIV 用于带符号整数除法
2 /* 0xf6, 0xf7 */
3 make_group(gp3,
4 IDEX(test_I, test), EMPTY, EX(not), EX(neg),
5 EX(mul), EX(imul1), EX(div), EX(idiv))

```

14. ADCSBB

ADC — Add with Carry

Opcode	Instruction	Clocks	Description
14 ib	ADC AL,imm8	2	Add with carry immediate byte to AL
15 iw	ADC AX,imm16	2	Add with carry immediate word to AX
15 id	ADC EAX,imm32	2	Add with carry immediate dword to EAX
80 /2 ib	ADC r/m8,imm8	2/7	Add with carry immediate byte to r/m byte
81 /2 iw	ADC r/m16,imm16	2/7	Add with carry immediate word to r/m word
81 /2 id	ADC r/m32,imm32	2/7	Add with CF immediate dword to r/m dword
83 /2 ib	ADC r/m16,imm8	2/7	Add with CF sign-extended immediate byte to r/m word
83 /2 id	ADC r/m32,imm8	2/7	Add with CF sign-extended immediate byte into r/m dword
10 /r	ADC r/m8,r8	2/7	Add with carry byte register to r/m byte
11 /r	ADC r/m16,r16	2/7	Add with carry word register to r/m word
11 /r	ADC r/m32,r32	2/7	Add with CF dword register to r/m dword
12 /r	ADC r8,r/m8	2/6	Add with carry r/m byte to byte register
13 /r	ADC r16,r/m16	2/6	Add with carry r/m word to word register
13 /r	ADC r32,r/m32	2/6	Add with CF r/m dword to dword register

Operation

DEST ← DEST + SRC + CF;

图 9: adc

SBB — Integer Subtraction with Borrow

Opcode	Instruction	Clocks	Description
1C ib	SBB AL,imm8	2	Subtract with borrow immediate byte from AL
1D iw	SBB AX,imm16	2	Subtract with borrow immediate word from AX
1D id	SBB EAX,imm32	2	Subtract with borrow immediate dword from EAX
90 /3 ib	SBB r/m8,imm8	2/7	Subtract with borrow immediate byte from r/m byte
91 /3 iw	SBB r/m16,imm16	2/7	Subtract with borrow immediate from r/m word
91 /3 id	SBB r/m32,imm32	2/7	Subtract with borrow immediate dword from r/m dword
83 /3 ib	SBB r/m16,imm8	2/7	Subtract with borrow sign-extended immediate byte from r/m word
83 /3 id	SBB r/m32,imm8	2/7	Subtract with borrow sign-extended immediate byte from r/m dword
18 /r	SBB r/m8,r8	2/6	Subtract with borrow byte register from r/m byte
19 /r	SBB r/m16,r16	2/6	Subtract with borrow word register from r/m word
19 /r	SBB r/m32,r32	2/6	Subtract with borrow dword from r/m dword
1A /r	SBB r8,r/m8	2/7	Subtract with borrow byte register from r/m byte
1B /r	SBB r16,r/m16	2/7	Subtract with borrow word register from r/m word
1B /r	SBB r32,r/m32	2/7	Subtract with borrow dword register from r/m dword

Operation

IF SRC is a byte and DEST is a word or dword
 THEN DEST = DEST - (SignExtend(SRC) + CF)
 ELSE DEST = DEST - (SRC + CF);

图 10: sbb

ADCSBB

```

1 //带进位加法指令
2 //exec.c
3 /* 0x80, 0x81, 0x83 */
4 make_group(gp1,
5 EX(add), EX(or), EX(adc), EX(sbb),
6 EX(and), EX(sub), EX(xor), EX(cmp))
7 /* 0x10 */ IDEXW(G2E, adc, 1), IDEX(G2E, adc), IDEXW(E2G, adc, 1),
8 IDEX(E2G, adc),
9 /* 0x14 */ IDEXW(I2a, adc, 1), IDEX(I2a, adc), EMPTY, EMPTY,
10
11 //sbb用于带借位的减法运算。在使用 SBB 指令时，需要确保 destination 和 source
   的大小
12 相同
13 make_group(gp1,
14 EX(add), EX(or), EX(adc), EX(sbb),

```

```

15 EX(and), EX(sub), EX(xor), EX(cmp))
16 /* 0x18 */ IDEXW(G2E, sbb, 1), IDEX(G2E, sbb), IDEXW(E2G, sbb, 1),
17 IDEX(E2G, sbb),
18 /* 0x1c */ IDEXW(I2a, sbb, 1), IDEX(I2a, sbb), EMPTY, EMPTY,

```

15. NEG

NEG 用于求一个操作数的负值。NEG 指令会将操作数取反并加 1，得到的结果作为新的值存入操作数中。

CF (进位标志): 如果操作数为零, 则 CF 被清零, 否则 CF 被设置为 1。

OF (溢出标志): 如果操作数为 -231, 则 OF 被设置为 1, 否则 OF 被清零。

PF (奇偶标志): 如果结果的二进制表示中 1 的个数为偶数, 则 PF 被设置为 1, 否则 PF 被清零。

SF (符号标志): 如果结果为负, 则 SF 被设置为 1, 否则 SF 被清零。

ZF (零标志): 如果结果为零, 则 ZF 被设置为 1, 否则 ZF 被清零。

NEG — Two's Complement Negation

Opcode	Instruction	Clocks	Description
F6	/3 NEG r/m8	2/6	Two's complement negate r/m byte
F7	/3 NEG r/m16	2/6	Two's complement negate r/m word
F7	/3 NEG r/m32	2/6	Two's complement negate r/m dword

Operation

```

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

```

图 11: Neg

Neg

```

1 //arith.c
2 make_EHelper(neg) {
3 // TODO();
4 rtl_sub(&t2, &tzero, &id_dest->val);
5 rtl_update_ZFSF(&t2, id_dest->width);
6 rtl_neq0(&t0,&id_dest->val);
7 rtl_set_CF(&t0);
8 rtl_eq1(&t0,&id_dest->val,0x80000000);
9 rtl_set_OF(&t0);
10 operand_write(id_dest,&t2);
11 print_asm_template1(neg);
12 }
13 // exec.c
14 /* 0xf6, 0xf7 */
15 make_group(gp3,
16 IDEX(test_I, test), EMPTY, EX(not), EX(neg),
17 EX(mul), EX(imul1), EX(div), EX(idiv))

```


16. OR

Or

```

1 //logic.c
2 make_EHelper(or) {
3 // TODO();
4 rtl_or(&t2,&id_dest->val,&id_src->val);
5 operand_write(id_dest,&t2);
6 rtl_update_ZFSF(&t2,id_dest->width);
7 rtl_set_CF(&tzero);
8 rtl_set_OF(&tzero);
9 print_asm_template2(or);
10 print_asm_template2(or);
11 }
12 //exec.c
13 /* 0x08 */ IDEXW(G2E, or, 1), IDEX(G2E, or), IDEXW(E2G, or, 1), IDEX(E2G,
14 or),
15 /* 0x0c */ IDEXW(I2a, or, 1), IDEX(I2a, or), EMPTY, EX(2byte_esc),
16 /* 0x80, 0x81, 0x83 */
17 make_group(gp1,
18 EX(add), EX(or), EX(adc), EX(sbb),
19 EX(and), EX(sub), EX(xor), EX(cmp))

```

17. NOT

Not

```

1 //logic.c
2 make_EHelper(not) {
3 // TODO();
4 rtl_not(&id_dest->val);
5 operand_write(id_dest,&id_dest->val);
6 print_asm_template1(not);
7 }
8 //exec.c
9 /* 0xf6, 0xf7 */
10 make_group(gp3,
11 IDEX(test_I, test), EMPTY, EX(not), EX(neg),
12 EX(mul), EX(imul1), EX(div), EX(idiv))

```

18. DECINC

DECINC

```

1 //dec指令：用于将操作数减1，并把新值存储到操作数中。dec指令只影响OF, SF, ZF,
  AF, and PF标志
2 位。
3 //exec.c

```

```

4  /* 0xfe */
5  make_group(gp4,
6  EX(inc), EX(dec), EMPTY, EMPTY,
7  EMPTY, EMPTY, EMPTY, EMPTY)
8  /* 0xff */
9  make_group(gp5,
10 EX(inc), EX(dec), EX(call_rm), EMPTY,
11 EX(jmp_rm), EMPTY, EX(push), EMPTY)
12 //arith.c
13 make_EHelper(dec) {
14 // TODO();
15 rtl_subi(&t2, &id_dest->val, 1);
16 operand_write(id_dest, &t2);
17 rtl_update_ZFSF(&t2, id_dest->width);
18 rtl_eqi(&t0, &t2, 0xffffffff);
19 rtl_set_OF(&t0);
20 print_asm_template1(dec);
21 }
22 //exec.c
23 /* 0x48 */ IDEX(r, dec), IDEX(r, dec), IDEX(r, dec), IDEX(r, dec),
24 /* 0x4c */ IDEX(r, dec), IDEX(r, dec), IDEX(r, dec), IDEX(r, dec),
25
26 //DEST ← DEST + 1;与dec相似, 不影响CF标志位
27 //arith.c
28 make_EHelper(inc) {
29 //TODO();
30 rtl_addi(&t2, &id_dest->val, 1);
31 operand_write(id_dest, &t2);
32 rtl_update_ZFSF(&t2, id_dest->width);
33 rtl_eqi(&t0, &t2, 0x80000000);
34 rtl_set_OF(&t0);
35 print_asm_template1(inc);
36 }
37 //exec.c
38 /* 0xfe */
39 make_group(gp4,
40 EX(inc), EX(dec), EMPTY, EMPTY,
41 EMPTY, EMPTY, EMPTY, EMPTY)
42 /* 0xff */
43 make_group(gp5,
44 EX(inc), EX(dec), EX(call_rm), EMPTY,
45 EX(jmp_rm), EMPTY, EX(push), EMPTY)
46 /* 0x40 */ IDEX(r, inc), IDEX(r, inc), IDEX(r, inc), IDEX(r, inc),
47 /* 0x44 */ IDEX(r, inc), IDEX(r, inc), IDEX(r, inc), IDEX(r, inc),

```

19. CLTDCWTL

cltd: 将带符号的长整数转换为带符号的双长整数。

把 `eax` 的 32 位整数扩展为 64 位, 高 32 位用 `eax` 的符号位填充保存到 `edx`, 或 `ax` 的 16 位整数扩展为 32 位, 高 16 位用 `ax` 的符号位填充保存到 `dx`。

CLTDCWTL

```

1 //data-mov.c
2 make_EHelper(cltd) {
3     if (decoding.is_operand_size_16) {
4         // TODO();
5         rtl_lr_w(&t0, R_AX);
6         rtl_sext(&t0, &t0, 2);
7         rtl_sari(&t0, &t0, 31);
8         rtl_sr_w(R_DX, &t0);
9     }
10    else {
11        // TODO();
12        rtl_sari(&cpu.edx, &cpu.eax, 31);
13    }
14    print_asm(decoding.is_operand_size_16 ? "cwtl" : "cltd");
15 }
16 //exec.c
17 /* 0x98 */ EX(cwtl), EX(cltd), EMPTY, EMPTY,
```

20. LEAVE

LEAVE 指令用于执行栈帧的恢复操作, 其作用相当于执行以下两个操作:

1. 将 `ESP` 寄存器的值设置为 `EBP` 寄存器中保存的值, 以恢复栈指针的位置。
2. 将 `EBP` 寄存器的值设置为栈顶中保存的值, 以恢复调用者的栈帧。

Leave

```

1 //mov esp, ebp
2 //pop ebp
3 //data-mov.c
4 make_EHelper(leave) {
5     // TODO();
6     rtl_mv(&cpu.esp, &cpu.ebp);
7     rtl_pop(&cpu.ebp);
8     print_asm("leave");
9 }
10 // exec.c
11 /* 0xc8 */ EMPTY, EX(leave), EMPTY, EMPTY,
```

21. CALL 补充

`CALL rm` 指令的目标地址是通过寄存器或内存中的值来确定的, 其寻址方式类似于其他的寄存器间接寻址指令。与第一阶段实现的 `call` 的区别在于 `jmp_eip` 的计算方法。

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 ptr16:32	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

NOTE:

Values of ts are given by the following table:

图 12: Call r/m

call rm

```

1 /* 0xff */
2 make_group(gp5,
3 EX(inc), EX(dec), EX(call_rm), EMPTY,
4 EX(jmp_rm), EMPTY, EX(push), EMPTY)

```

22. SARSALSHLSHR

1. SAR (算术右移): SAR 指令是带符号右移指令, 用于将一个有符号数向右移动指定的位数, 移位时保留符号位。SAR 指令的语法为 SAR destination, count, 其中, destination 表示要移位的目标操作数, count 表示要右移的位数。SAR 指令将 destination 操作数的每一位都向右移 count 位, 并用符号位填充空位。

2. SAL (逻辑左移): SAL 指令是逻辑左移指令, 用于将一个无符号数向左移动指定的位数, 移位时低位补零。SAL 指令的语法与 SHL 指令相同, 可以使用 SHL 代替。

3. SHL (逻辑左移): SHL 指令是逻辑左移指令, 用于将一个无符号数向左移动指定的位数, 移位时低位补零。SHL 指令的语法为 SHL destination, count。其中, destination 表示要移位的目标操作数, count 表示要左移的位数。SHL 指令将 destination 操作数的每一位都向左移 count 位, 并用零填充空位。

4. SHR (逻辑右移): SHR 指令是逻辑右移指令, 用于将一个无符号数向右移动指定的位数, 移位时高位补零。SHR 指令的语法为 SHR destination, count, 其中, destination 表示要移位的目标操作数, count 表示要右移的位数。SHR 指令将 destination 操作数的每一位都向右移 count 位, 并用零填充空位。

SARSALSHLSHR

```

1 // logic.c
2 make_EHelper(shl) {
3 // TODO();
4 // unnecessary to update CF and OF in NEMU
5 rtl_shl(&t2, &id_dest->val, &id_src->val);
6 operand_write(id_dest, &t2);
7 rtl_update_ZFSF(&t2, id_dest->width);
8 print_asm_template2(shl);

```

```

9  }
10 make_EHelper(shr) {
11     //TODO();
12     // unnecessary to update CF and OF in NEMU
13     rtl_shr(&t2,&id_dest->val,&id_src->val);
14     operand_write(id_dest,&t2);
15     rtl_update_ZFSF(&t2,id_dest->width);
16     print_asm_template2(shr);
17 }
18 make_EHelper(sar) {
19     // TODO();
20     // unnecessary to update CF and OF in NEMU
21     rtl_sext(&t2, &id_dest->val, id_dest -> width);
22     rtl_sar(&t2, &t2, &id_src->val);
23     operand_write(id_dest, &t2);
24     rtl_update_ZFSF(&t2, id_dest -> width);
25     print_asm_template2(sar);
26 }
27 make_EHelper(shl) {
28     // TODO();
29     // unnecessary to update CF and OF in NEMU
30     rtl_shl(&t2,&id_dest->val,&id_src->val);
31     operand_write(id_dest,&t2);
32     rtl_update_ZFSF(&t2,id_dest->width);
33     print_asm_template2(shl);
34 }
35 // rtl.h
36 static inline void rtl_mv(rtlreg_t* dest, const rtlreg_t *src1) {
37     // dest <- src1
38     // TODO();
39     rtl_addi(dest, src1, 0);
40 }
41 static inline void rtl_not(rtlreg_t* dest) {
42     // dest <- ~dest
43     // TODO();
44     rtl_xori(dest, dest, 0xffffffff);
45 }
46 static inline void rtl_sext(rtlreg_t* dest, const rtlreg_t* src1, int width)
47     {
48     // dest <- signext(src1[(width * 8 - 1) .. 0])
49     // TODO();
50     if(width == 0) {
51         rtl_mv(dest, src1);
52     }
53     else {
54         // assert(width == 1 || width == 2);
55         rtl_shli(dest, src1, (4 - width) * 8);
56         rtl_sari(dest, dest, (4 - width) * 8);

```

```

56 }
57 }
58 //exec.c
59 /* 0xc0, 0xc1, 0xd0, 0xd1, 0xd2, 0xd3 */
60 make_group(gp2,
61 EX(rol), EMPTY, EMPTY, EMPTY,
62 EX(shl), EX(shr), EMPTY, EX(sar))

```

(二) 基础设施

测试与调试, 调试的工具与原理

总结出一些调试的建议: 总是使用-Wall 和-Werror

尽可能多地在代码中插入 assert()

assert() 无法捕捉到 error 时, 通过 printf() 输出可疑的变量, 期望能观测到 error

printf() 不易观测 error 时, 通过 GDB 理解程序的细致行为

1. 代码: Differential Testing

让在 NEMU 中执行的每条指令也在真机中执行一次, 然后对比 NEMU 和真机的状态, 如果 NEMU 和真机的状态不一致, 我们就捕捉到 error, 在软件测试领域称为 differential testing.

为了通过 differential testing 的方法测试 NEMU 实现的正确性, 我们让 NEMU 和 QEMU 逐条指令地执行同一个客户程序. 双方每执行完一条指令, 就检查各自的寄存器和内存的状态, 如果发现状态不一致, 就马上报告错误, 停止客户程序的执行。

在 difftest_step() 中添加相应的代码, 实现 differential testing 的核心功能。

Differential Testing

```

1  if(r.eax!=cpu.eax) {
2  printf("expect:_%d_true:_%d_at:_%x\n", r.eax, cpu.eax, cpu.eip);
3  diff=true;
4  }
5  if(r.ecx!=cpu.ecx) {
6  printf("expect:_%d_true:_%d_at:_%x\n", r.ecx, cpu.ecx, cpu.eip);
7  diff=true;
8  }
9  if(r.edx!=cpu.edx) {
10 printf("expect:_%d_true:_%d_at:_%x\n", r.edx, cpu.edx, cpu.eip);
11 diff=true;
12 }
13 if(r.ebx!=cpu.ebx) {
14 printf("expect:_%d_true:_%d_at:_%x\n", r.ebx, cpu.ebx, cpu.eip);
15 diff=true;
16 }
17 if(r.esp!=cpu.esp) {
18 printf("expect:_%d_true:_%d_at:_%x\n", r.esp, cpu.esp, cpu.eip);
19 diff=true;
20 }
21 if(r.ebp!=cpu.ebp) {

```

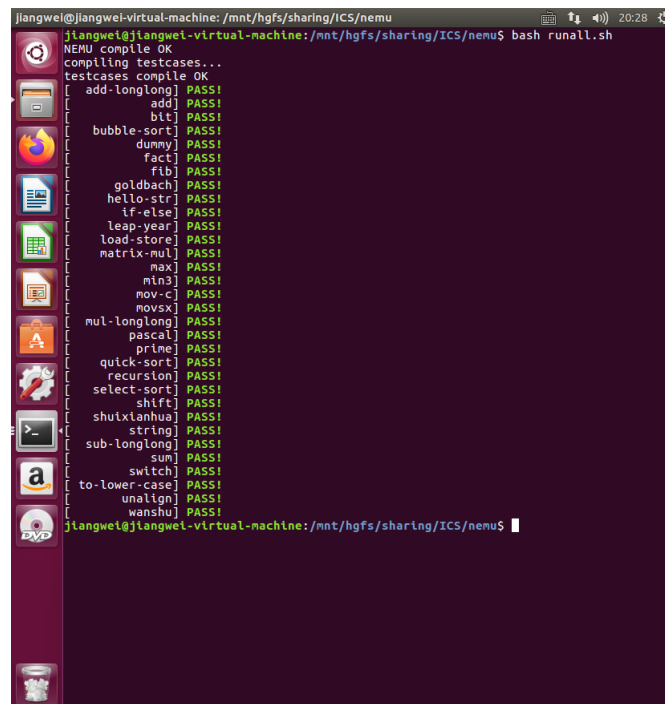
```
22 printf("expect:_%d_true:_%d_at:_%x\n", r.ebp, cpu.ebp, cpu.eip);
23 diff=true;
24 }
25 if(r.esi!=cpu.esi) {
26 printf("expect:_%d_true:_%d_at:_%x\n", r.esi, cpu.esi, cpu.eip);
27 diff=true;
28 }
29 if(r.edi!=cpu.edi) {
30 printf("expect:_%d_true:_%d_at:_%x\n", r.edi, cpu.edi, cpu.eip);
31 diff=true;
32 }
33 if(r.eip!=cpu.eip) {
34 diff=true;
35 Log("different:qemu.eip=0x%x,nemu.eip=0x%x",r.eip,cpu.eip);
36 }
37 if (diff) {
38 nemu_state = NEMU_END;
39 }
40
41 //打开 nemu/include/common.h 中的宏 DIFF_TEST
42
43 #define DIFF_TEST
```

2. 一键回归测试

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。

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。

执行 *bashrunall.sh*



```

jiangwei@jiangwei-virtual-machine: /mnt/hgfs/sharing/ICS/nemu
jiangwei@jiangwei-virtual-machine: /mnt/hgfs/sharing/ICS/nemu$ bash runall.sh
NEMU compile OK
compiling testcases...
testcases compile OK
[ add-longlong] PASS!
[ add] PASS!
[ bit] PASS!
[ bubble-sort] PASS!
[ dunny] 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!
jiangwei@jiangwei-virtual-machine: /mnt/hgfs/sharing/ICS/nemu$

```

图 13: 测试结果

捕捉死循环，如何实现？

对程序设置最大运行时间，如果超过该时间则判断程序进入死循环。

对于一个操作系统来说，操作系统可以通过检测进程是否超过了分配给它的时间片来判断是否出现了死循环。当一个进程的时间片用完时，操作系统会强制将其挂起，并将控制权转移到其他进程。如果一个进程在多次运行时都超过了它的时间片，那么很可能出现了死循环，操作系统可以将其杀死或报告错误。

四、 stage3

输入输出是计算机与外界交互的基本手段，在真实的计算机中，输入输出都是通过 I/O 设备来完成的。

x86 提供了 in 和 out 指令用于访问设备，其中 in 指令用于将设备寄存器中的数据传输到 CPU 寄存器中，out 指令用于将 CPU 寄存器中的数据传送到设备寄存器中。

内存映射 I/O 这种编址方式非常巧妙，它是通过不同的物理内存地址给设备编址的。这种编址方式将一部分物理内存“重定向”到 I/O 地址空间中，CPU 尝试访问这部分物理内存的时候，实际上最终是访问了相应的 I/O 设备，CPU 却浑然不知。这样以后，CPU 就可以通过普通的访存指令来访问设备。这也是内存映射 I/O 得天独厚的好处：物理内存的地址空间和 CPU 的位宽都会不断增长，内存映射 I/O 从来不需要担心 I/O 地址空间耗尽的问题。

(一) 代码：加入 IOE

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

1. 串口

串口是最简单的输出设备, . 由于 NEMU 串行模拟计算机系统的工作, 串口的状态寄存器可以一直处于空闲状态; 每当 CPU 往数据寄存器中写入数据时, 串口会将数据传送到主机的标准输出。

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.

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。

Hello World

```

1  //修改 system.c , in 指令用于将设备寄存器中的数据传输到 CPU 寄存器中
2  // system.c
3  make_EHelper(in) {
4      rtl_li(&t0, pio_read(id_src->val, id_dest->width));
5      operand_write(id_dest, &t0);
6      print_asm_template2(in);
7      #ifdef DIFF_TEST
8      diff_test_skip_qemu();
9      #endif
10 }
11 //out指令用于将数据从CPU寄存器到设备寄存器中
12 make_EHelper(out) {
13     pio_write(id_dest->val, id_src->width, id_src->val);
14     print_asm_template2(out);
15     #ifdef DIFF_TEST
16     diff_test_skip_qemu();
17     #endif
18 }
19
20 //exec.c
21 /* 0xe4 */ IDEXW(in_I2a, in, 1), IDEXW(in_I2a, in, 1), IDEXW(out_a2I, out,
22 1), IDEXW(out_a2I, out, 1),
23 /* 0xe8 */ IDEX(J, call), IDEX(J, jmp), EMPTY, IDEXW(J, jmp, 1),
24 /* 0xec */ IDEXW(in_dx2a, in, 1), IDEX(in_dx2a, in), IDEXW(out_a2dx, out,
25 1), IDEX(out_a2dx, out),
26
27 //in 和 out 指令在函数中分别调用 pio_read() 和 pio_write() 函数。
28 //之后需要在 nexus-am/am/arch/x86-nemu/src/trm.c 中定义宏 HAS_SERIAL 。

```

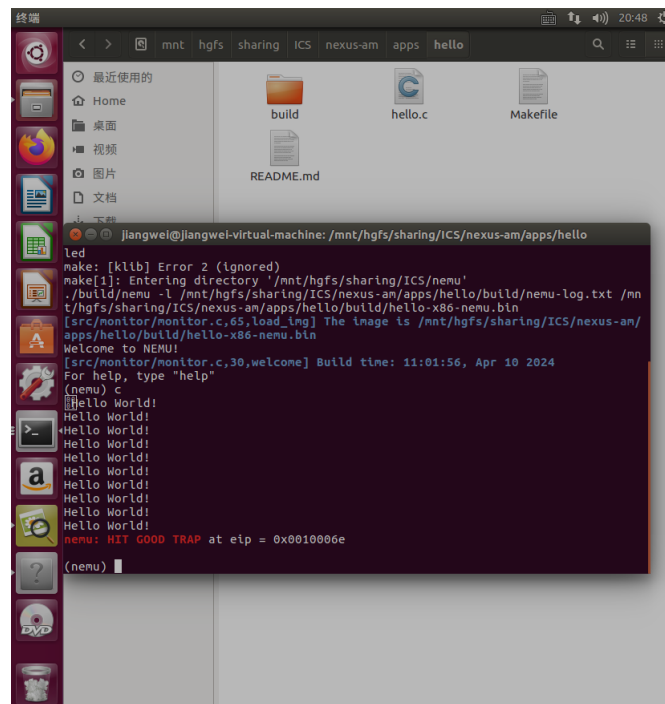


图 14: Hello World

2. 时钟

有了时钟, 程序才可以提供时间相关的体验. 实现 `_uptime()` 后, 在 NEMU 中运行 `timetest`, 程序每隔 1 秒输出一句话.

时钟

```

1 //完善 nexus-am/am/arch/x86-nemu/src/ioe.c 中的代码
2 //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
3 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
   即可。
4 unsigned long _uptime() {
5     unsigned long ms = inl(RTC_PORT) - boot_time;
6     return ms;
7 }

```

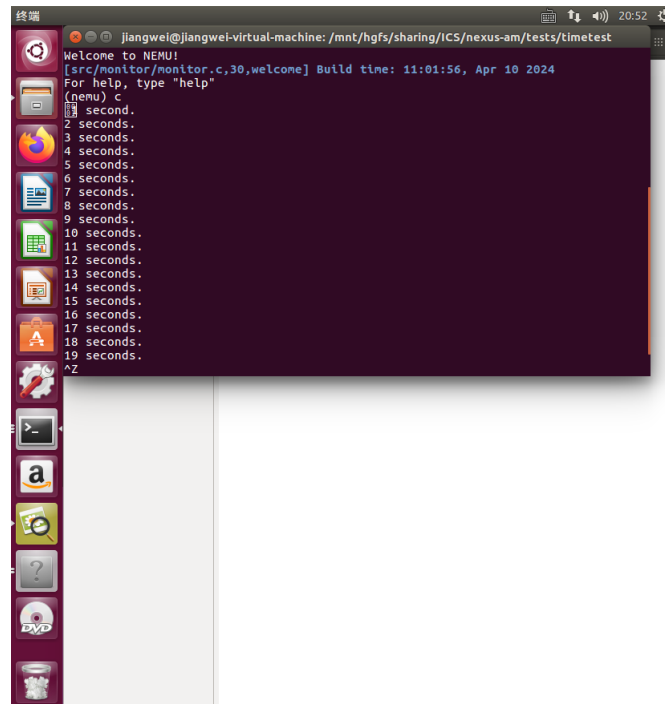


图 15: 时钟

看看 NEMU 跑多快
Dhrystone

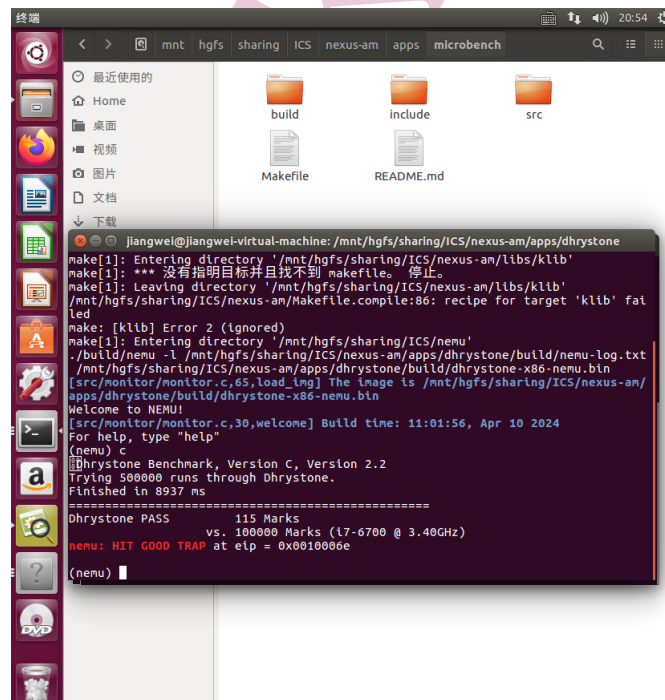
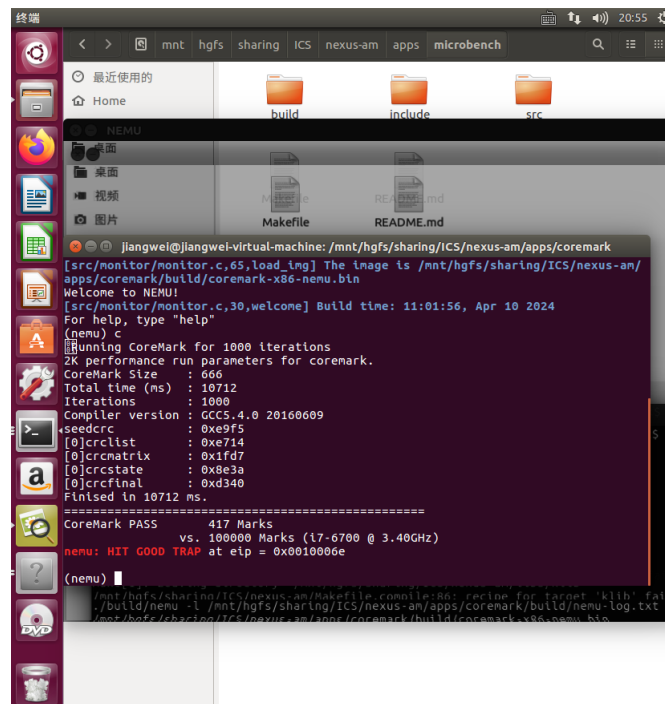


图 16: Dhrystone

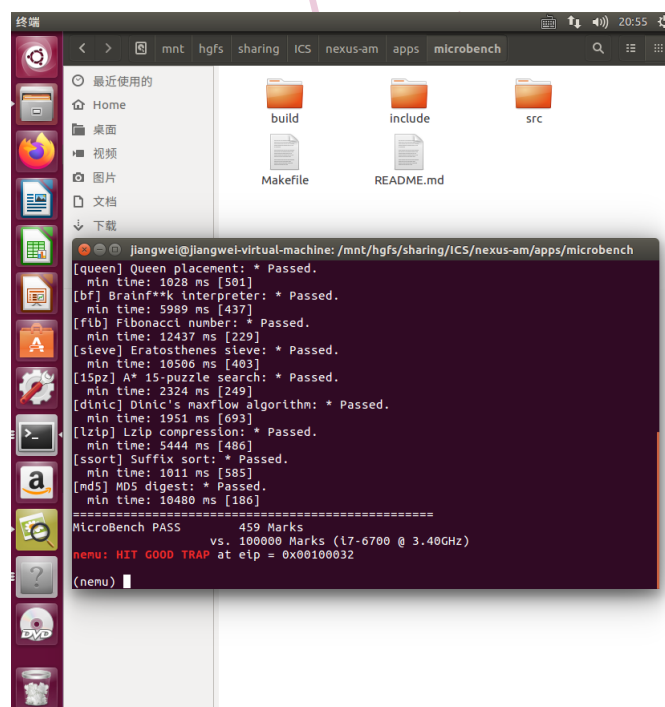
Coremark



```
jiangwei@jiangwei-virtual-machine: /mnt/hgfs/sharing/ICS/nexus-am/apps/coremark
[src/monitor/monitor.c,65,load_img] The image is /mnt/hgfs/sharing/ICS/nexus-am/
apps/coremark/build/coremark-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 11:01:56, Apr 10 2024
For help, type "help"
(nemu) c
Running CoreMark for 1000 iterations.
2K performance run parameters for coremark.
CoreMark Size : 666
Total time (ms) : 10712
Iterations : 1000
Compiler version : GCC5.4.0 20160609
xseedcrc : 0xe9f5
[0]crc1st : 0xe714
[0]crcmatrix : 0x1fd7
[0]crcstate : 0x8e3a
[0]crcfinal : 0xd340
Finished in 10712 ms.
=====
CoreMark PASS 417 Marks
vs. 100000 Marks (i7-6700 @ 3.40GHz)
nemu: HIT GOOD TRAP at eip = 0x0010006e
(nemu)
```

图 17: Coremark

microbench



```
jiangwei@jiangwei-virtual-machine: /mnt/hgfs/sharing/ICS/nexus-am/apps/microbench
[queen] Queen placement: * Passed.
min time: 1028 ms [501]
[bf] Brainf**k interpreter: * Passed.
min time: 5989 ms [437]
[trib] Fibonacci number: * Passed.
min time: 12437 ms [229]
[steve] Eratosthenes sieve: * Passed.
min time: 10506 ms [403]
[15pz] A* 15-puzzle search: * Passed.
min time: 2324 ms [249]
[dninc] Dijkstra's maxflow algorithm: * Passed.
min time: 1951 ms [693]
[lzip] Lzip compression: * Passed.
min time: 5444 ms [406]
[ssort] Suffix sort: * Passed.
min time: 1011 ms [585]
[md5] MD5 digest: * Passed.
min time: 10480 ms [186]
=====
MicroBench PASS 459 Marks
vs. 100000 Marks (i7-6700 @ 3.40GHz)
nemu: HIT GOOD TRAP at eip = 0x00100032
(nemu)
```

图 18: 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。

3. 键盘

键盘是最基本的输入设备. 一般键盘的工作方式如下: 当按下一个键的时候, 键盘将会发送该键的通码 (make code); 当释放一个键的时候, 键盘将会发送该键的断码 (break code).

每当用户敲下/释放按键时, 将会把相应的键盘码放入数据寄存器, 同时把状态寄存器的标志设置为 1, 表示有按键事件发生. CPU 可以通过端口 I/O 访问这些寄存器, 获得键盘码。

实现 `_read_key()` 后, 在 NEMU 中运行 `keytest` 程序, 在程序运行时弹出的新窗口中按下按键, 将会看到程序输出相应的按键信息。

0x60: I8042_DATA_PORT, 数据寄存器, 4 字节。

0x64: I8042_STATUS_PORT, 状态寄存器, 1 字节。

`i8042_io_handler()`: IO 读写时的设备回调函数。该函数只支持读操作: 读取数据时, 将状态寄存器置为 0; 读取状态时, 若当前状态为 0 且存在按下/释放按键事件 (`key_queue` 的 `key_f!=key_r`), 则将事件记录在数据寄存器中并将状态寄存器置 1。

键盘

```
1  int _read_key() {
2  if(inb(0x64)) {
3  return inl(0x60);
4  }
5  return _KEY_NONE;
6  }
```

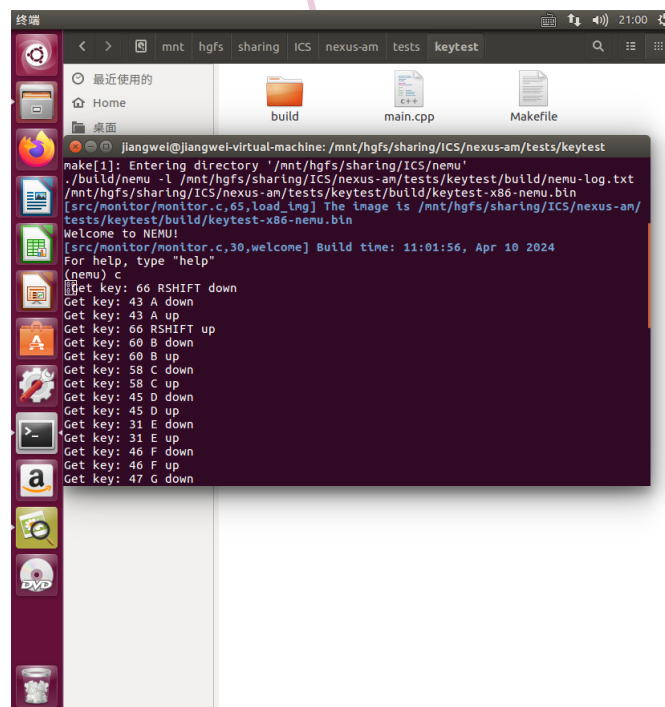


图 19: 键盘

如何检测多个键被同时按下

当发送的数字为键盘码 + 0x8000 时, 意味着键盘被按下, 而当单纯发送键盘码时, 意味着键盘被抬起。每个按键的通码断码都不同, 所以可以识别出不同的按键。

当检测到一个键被按下的时候, 去检测此时其他是否有按键被按下。

4. VGA

VGA 可以用于显示颜色像素, 是最常用的输出设备。

添加内存映射 I/O

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`.

`nemu/src/device/io/mmio.c` 中: `int is_mmio(paddr_t addr)`: 查看该内存地址 `addr`, 若为内存映射 IO, 则返回映射号, 否则返回 -1

`uint32_t mmio_read(paddr_t addr, int len, int map_NO)` 与 `void mmio_write(paddr_t addr, int len, uint32_t data, int map_NO)` 为内存映射 IO 的读写访问函数。

修改 `nemu/src/memory/memory.c`: 需要引用头文件 `device/mmio.h`, 相当于把 `io` 放入内存中。先用 `is_mmio` 函数判断物理地址是否被映射到 I/O 空间, 对于 `paddr_read` 若返回 -1 则访问 `pmem`, 否则使用 `mmio_read` 函数读取 `port` 位置的内存, 对于 `paddr_write`, 若不返回 -1 则调用 `mmio_write` 将数据写入 `port` 位置内存。

VGA

```

1  #include "device/mmio.h"
2  /* Memory accessing interfaces */
3  uint32_t paddr_read(paddr_t addr, int len) {
4      int r = is_mmio(addr);
5      if(r == -1) {
6          return pmem_rw(addr, uint32_t) & (~0u >> ((4 - len) << 3));
7      }
8      else {
9          return mmio_read(addr, len, r);
10     }
11 }
12 void paddr_write(paddr_t addr, int len, uint32_t data) {
13     int r = is_mmio(addr);
14     if(r == -1){
15         memcpy(guest_to_host(addr), &data, len);
16     }
17     else {
18         mmio_write(addr, len, data, r);
19     }
20 }

```

在 NEMU 中运行 `videotest` 程序, 新窗口中输出了一些颜色信息。

实现正确的 `_draw_rect()`, 在 NEMU 中重新运行 `videotest`, 到新窗口中输出了相应的动画效果。

VGA 动画效果

```

1  void _draw_rect(const uint32_t *pixels, int x, int y, int w, int h) {
2      int temp = (w > _screen.width - x) ? _screen.width - x : w;
3      int cp_bytes = temp * sizeof(uint32_t);
4      for (int i = 0; i < h && y + i < _screen.height; i++) {

```

```
5 memcpy(&fb[(y + i) * _screen.width + x], pixels, cp_bytes);  
6 pixels += w;  
7 }  
8 }
```

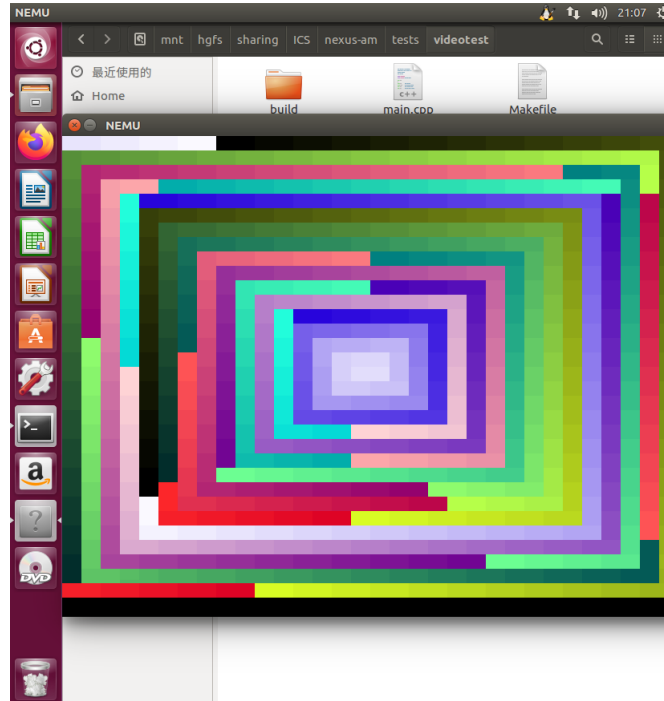


图 20: vga

神奇的调色板

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

答:当游戏场景需要进行渐出渐入效果时,游戏开发者可以通过修改调色板中的颜色序列,使原本的颜色逐渐变暗或变亮,从而实现渐出渐入的效果。例如,当一个游戏场景需要进行淡出效果时,游戏开发者可以将调色板中的颜色序列从明亮的颜色逐渐变为黑色,从而实现淡出的效果。**打字小游戏**



图 21: 打字小游戏

litenes

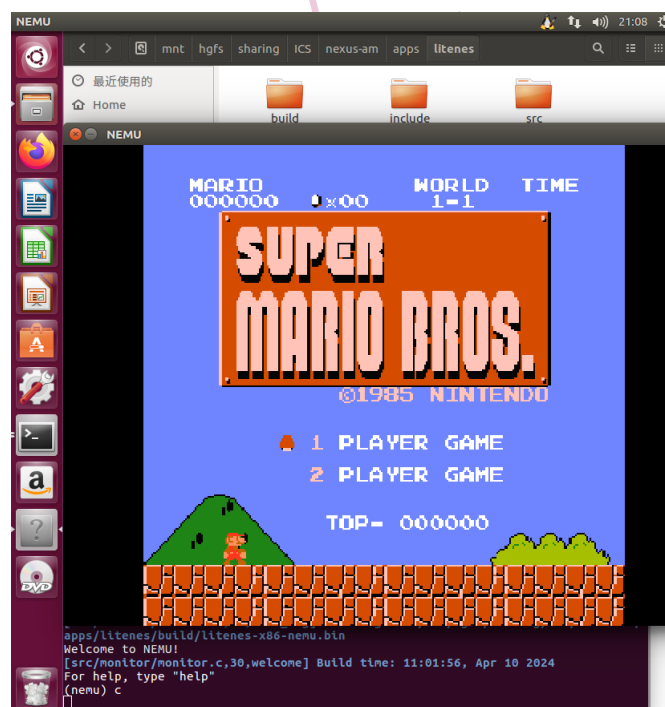


图 22: litenes

理解 volatile 关键字

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?

答：变量如果加了 volatile 修饰，编译器可能会对代码进行优化，则会从内存重新装载内容，而不是直接从寄存器拷贝内容，去掉 volatile 会导致错误发生。会检测不到设备寄存器的变化。
在 TRM 的基础上添加了 IOE, 本质上还是“取指->译码->执行”的工作方式。

5. problems

找不到 `AM_HOME`

.bashrc 中手动添加路径，后 `source ./bashrc`

make run dummy 后提示找不到 `klib-x86-nemu.a`、`dummy-x86-nemu`

从原来环境的框架中拷贝一份，放到 `nexus-am/libs/klib/build/` 目录下

(二) 必答题

1. static inline

在 `nemu/include/cpu/rtl.h` 中，你会看到由 `static inline` 开头定义的各种 RTL 指令函数。选择其中一个函数，分别尝试去掉 `static`，去掉 `inline` 或去掉两者，然后重新进行编译，你会看到发生错误。请分别解释为什么会发生这些错误？你有办法证明你的想法吗？

去掉 `static`

有警告，但可以正常运行。

`inline` 实际上表示建议内联，但并非强制内联，编译器可以忽略。若想要确保内联，在使用 `inline` 的时候要加入 `static`，否则 `inline` 不内联的时候就和普通函数在头文件中的定义是一样的，若多个 `c` 文件都包含时就会产生歧义。如果去掉 `static`，编译器没有强制内联，该函数就相当于一个普通函数。

去掉 `inline`

会出现定义但未使用的问题。

当去掉 `inline` 关键字时，编译器会将该函数生成成为一个独立的代码块，并将其放入对应的目标文件中。如果该函数在其他文件中没有被调用，则编译器会认为该函数是未使用的，并给出相应的警告。因此，如果只去掉 `inline` 关键字，可能会导致编译器给出“unused function”的警告信息。

去掉两者

报错，无法运行，出现重复定义的问题。

当同时去掉 `static` 和 `inline` 关键字时，该函数会变为一个非静态的非内联函数，会被编译器视为一个普通的函数，并且可以在其他文件中被调用。如果在其他文件中也定义了同名的函数，则会出现重复定义的问题，导致编译错误。因为 `exe` 文件夹下的 `.c` 文件中都引用了 `rtl.h` 文件，编译的时候这些文件和 `rtl.h` 文件中都有了 `rtl_mv` 函数的定义，导致了多次定义。

2. makefile

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。)

`make ALL=xxx run` 工作方式：

首先读取 `$(AM_HOME)/Makefile.check` 中的默认参数，根据设置的 `ARCH` 指代架构然后通过命令行指定的 `ALL` 寻找 `tests` 目录下对应的 `.c` 文件

根据 `AM` 中指定 `ARCH` 提供的编译链接规则编译生成可执行文件

将可执行文件作为 NEMU 的镜像启动 NEMU，控制权转交给 NEMU

make 程序会进行以下步骤：1. 读取 Makefile 文件，并解析其中的变量、规则和指令。

2. 根据 Makefile 中的规则和指令，确定需要生成的目标文件，以及生成目标文件所需的依赖文件和编译命令。

3. 检查目标文件和依赖文件的时间戳，确定哪些文件需要重新生成。

如果文件不存在，或是文件所依赖的后面的.o 文件的文件修改时间要比这个文件新，那么，他就会执行后面所定义的命令来生成 h 这个文件，这个也就是重编译

如果文件所依赖的.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。

.c 文件和.h 文件存在，于是 make 会生成.o 文件

4. 根据需要重新生成的文件，执行相应的编译命令，生成目标文件。

5. 重复步骤 3 和步骤 4，直到所有的目标文件都已经生成。

在编译链接的过程中，Makefile 会执行以下操作：

1. 根据源文件的后缀名，使用隐式规则来生成相应的目标文件。例如，对于.c 文件，Makefile 中定义了如下的规则：

```
%o: %.c $(CC) $(CFLAGS) $(CPPFLAGS) -c -o <
```

这个规则表示，对于每个.c 文件，使用 \$(CC) 变量所指定的编译器，加上 \$(CFLAGS) 和 \$(CPPFLAGS) 变量所指定的编译选项，将其编译成一个.o 目标文件，并保存在 \$@ 变量所表示的位置。

2. 根据目标文件之间的依赖关系，生成可执行文件。例如，在 nemu 的 Makefile 中，定义了如下的规则：nemu: \$(OBS) \$(CC) \$(LDFLAGS) -o \$@ \$(LDLIBS)

这个规则表示，对于 nemu 目标文件，它依赖于 \$(OBS) 变量所表示的所有.o 目标文件，使用 \$(CC) 变量所指定的编译器，加上 \$(LDFLAGS) 变量所指定的链接选项，将所有.o 目标文件链接成一个可执行文件，并保存在 \$@ 变量所表示的位置。

五、 实验总结与感想

要实现的指令很多，查手册、补代码、测试很花时间，我觉得很难一点也不好写 (crying.jpg)；希望自己能加深对 nemu 的理解，提升阅读源码的能力和解决 bug 的能力