南开大学

Nankai University

南　　开　　大　　学

计　算　机　学　院

计算机系统实验报告

PA2

年级：2022 级

专业：计算机科学与技术

姓名：姚知言

学号：2211290

指导教师：卢冶

2025 年 4 月 6 日

目录

一、 实验目的

1. 了解计算机指令的执行流程，完成计算机指令实现。
2. 完善指令系统，了解 AM 的原理。
3. 学习 CPU 与外设交互的过程，及性能测试相关原理。

二、 实验重要内容

PA2 的实验共分为 3 个阶段，包括以下要点：

- 第 1 阶段：了解计算机如何执行一条指令。

```
while (1) {
  从EIP指示的存储器位置取出指令;
  执行指令;
  更新EIP;
}
```

图 1: 计算机执行一条指令

- 第 1，2 阶段：根据现代指令系统，完成指令设计。
- 第 2 阶段：了解运行时环境与 AM。

AM = TRM + IOE + ASYE + PTE + MPE

- TRM(Turing Machine) – 图灵机，为计算机提供基本的计算能力
- IOE(I/O Extension) – 输入输出扩展，为计算机提供输出输入的能力
- ASYE(Asynchronous Extension) – 异步处理扩展，为计算机提供处理中断异常的能力
- PTE(Protection Extension) – 保护扩展，为计算机提供存储保护的能力
- MPE(Multi-Processor Extension) – 多处理器扩展，为计算机提供多处理器通信的能力(MPE 超出了 ICS 课程的范围，在 PA 中不会涉及)

图 2: AM

- 第 2 阶段：构建 Differential Testing 基础设施。
- 第 3 阶段：构建 IOE，实现计算机与外设的交互。
- 第 3 阶段：使用 benchmark 测试计算机性能。

三、 阶段 1 实验方法与结果

(一) 编码内容

1. 初始化测试

首先在 nexus-am/tests/cputest 中键入 makeARCH=x86-nemuALL=dummy run，编译运行 dummy 程序。结果如图3所示。

```
latesoon@latesoon-virtual-machine:~/ics2017/nexus-am/tests/cputest$ make ARCH=x86-nemu ALL=dummy run
Building dummy [x86-nemu]
Building am [x86-nemu]
make[2]: *** 没有指明目标并且找不到 makefile。 停止。
[src/monitor/monitor.c,65,load_img] The image is /home/latesoon/ics2017/nexus-am/tests/cputest/build/dummy-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 14:14:50, Mar 24 2025
For help, type "help"
(nemu) c
invalid opcode(eip = 0x0010000a): e8 01 00 00 00 90 55 89 ...

There are two cases which will trigger this unexpected exception:
1. The instruction at eip = 0x0010000a is not implemented.
2. Something is implemented incorrectly.
Find this eip(0x0010000a) in the disassembling result to distinguish which case it is.

If it is the first case, see
 _____  ___   __   __ __                          _
(_)___ \ / _ \ / /  |  \/  |                      | |
  _   ) | (_) |/ /_  | \  / | __ _ _ __  _   _  __| |
 | |_ < > _ <| '_ \ | |\/| |/ _` | '_ \| | | |/ _` |
 | |__) | (_) | (_) || |  | | (_| | | | | |_| | (_| |
 |_|____/ \___/ \___/ |_|  |_|\__,_|_| |_|\__,_|\__,_|

for more details.

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

(nemu) q
 dummy
```

图 3: 环境测试

可以发现我们没有实现 0x10000a 处的指令,导致运行错误。接下来进行 call,push,sub,xor,pop,ret 六条新指令的实现。

2. opcode_table 补充

查阅 i386 手册附录 A，可以看到各操作码对应情况如下。单字节操作码如图4所示，双字节操作码如图5所示。

One-Byte Opcode Map

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	ADD						PUSH	POP	OR						PUSH	2-byte
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	ES	ES	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	CS	escape
1	ADC						PUSH	POP	SBB						PUSH	POP
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	SS	SS	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	DS	DS
2	AND						SEG	DAA	SUB						SEG	DAS
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	=ES		Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	=CS	
3	XOR						SEG	AAA	CMP						SEG	AAS
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	=SS		Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	=CS	
4	INC general register								DEC general register							
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
5	PUSH general register								POP into general register							
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
6	PUSHA	POPA	BOUND	ARPL	SEG	SEG	Operand	Address	PUSH	IMUL	PUSH	IMUL	INSB	INSW/D	OUTSB	OUTSW/D
			Gv,Ma	Ew,Rw	=FS	=GS	Size	Size	Ib	GvEvIv	Ib	GvEvIv	Yb,DX	Yb,DX	Dx,Xb	DX,Xv
7	Short displacement jump of condition (Jb)								Short-displacement jump on condition(Jb)							
	JO	JNO	JB	JNB	JZ	JNZ	JBE	JNBE	JS	JNS	JP	JNP	JL	JNL	JLE	JNLE
8	Immediate Grp1			Grp1	TEST		XCNG		MOV				MOV	LEA	MOV	POP
	Eb,Ib	Ev,Iv		Ev,Iv	Eb,Gb	Ev,Gv	Eb,Gb	Ev,Gv	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	Ew,Sw	Gv,M	Sw,Ew	Ev
9	NOP	XCHG word or double-word register with eAX							CBW	CWD	CALL	WAIT	PUSHF	POPF	SAHF	LAHF
		eCX	eDX	eBX	eSP	eBP	eSI	eDI			Ap		Fv	Fv		
A	MOV				MOVSB	MOVSW/D	CMPSB	CMPSW/D	TEST		STOSB	STOSW/D	LODSB	LODSW/D	SCASB	SCASW/D
	AL,Ob	eAX,Ov	Ob,AL	Ov,eAX	Xb,Yb	Xv,Yv	Xb,Yb	Xv,Yv	AL,Ib	eAX,Iv	Yb,AL	Yv,eAX	AL,Xb	eAX,Xv	AL,Xb	eAX,Xv
B	MOV immediate byte into byte register								MOV immediate word or double into word or double register							
	AL	CL	DL	BL	AH	CH	DH	BH	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
C	Shift Grp2		RET near		LES	LDS	MOV		ENTER	LEAVE	RET far		INT	INT	INTO	IRET
	Eb,Ib	Ev,Iv	Iw		Gv,Mp	Gv,Mp	Eb,Ib	Ev,Iv	Iw,Ib		Iw		3	Ib		
D	Shift Grp2				AAM	AAD		XLAT	ESC(Escape to coprocessor instruction set)							
	Eb,1	Ev,1	Eb,CL	Ev,CL												
E	LOOPNE	LOOPE	LOOP	JCXZ	IN		OUT		CALL	JNP			IN		OUT	
	Jb	Jb	Jb	Jb	AL,Ib	eAX,Ib	Ib,AL	Ib,eAX	Av	Jv	Ap	Jb	AL,DX	eAX,DX	DX,AL	DX,eAX
F	LOCK		REPNE	REP REPE	HLT	CMC	Unary Grp3		CLC	STC	CLI	STI	CLD	STD	INC/DEC	Indirct
							Eb	Ev							Grp4	Grp5

图 4: i386 单字节操作码

Two-Byte Opcode Map (first byte is 0FH)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	Grp6	Grp7	LAR Gw,Ew	LSL Gv,Ew			CLTS									
1																
2	MOV Cd,Rd	MOV Dd,Rd	MOV Rd,Cd	MOV Rd,Dd	MOV Td,Rd		MOV Rd,Td									
3																
4																
5																
6																
7																
8	Long-displacement jump on condition (Jv)								Long-displacement jump on condition (Jv)							
	JO	JNO	JB	JNB	JZ	JNZ	JBE	JNBE	JS	JNS	JP	JNP	JL	JNL	JLE	JNLE
9	Byte Set on condition (Eb)															
	SETO	SETNO	SETB	SETNB	SETZ	SETNZ	SETBE	SETNBE	SETS	SETNS	SETP	SETNP	SETL	SETNL	SETLE	SETNLE
A	PUSH FS	POP FS		BT Ev,Gv	SHLD EvGvIb	SHLD EvGvCL			PUSH GS	POP GS		BTS Ev,Gv	SHRD EvGvIb	SHRD EvGvCL		IMUL Gv,Ev
B			LSS Mp	BTR Ev,Gv	LFS Mp	LGS Mp	MOVZX Gv,Eb	MOVZX Gv,Ew			Grp-8 Ev,Ib	BTC Ev,Gv	BSF Gv,Ev	BSR Gv,Ev	MOVSX Gv,Eb	MOVSX Gv,Ew
C																
D																
E																
F																

图 5: i386 双字节操作码

部分指令的操作码（Opcodes）由 modR/M 字节的第 5、4、3 位决定，如图6所示。

Opcodes determined by bits 5,4,3 of modR/M byte:

Group	mod nnn R/M							
	000	001	010	011	100	101	110	111
1	ADD	OR	ADC	SBB	AND	SUB	XOR	CMP
2	ROL	ROR	RCL	RCR	SHL	SHR		SAR
3	TEST Ib/Iv		NOT	NEG	MUL AL/eAX	IMUL AL/eAX	DIV AL/eAX	IDIV AL/eAX
4	INC Eb	DEC Eb						
5	INC Ev	DEC Ev	CALL Ev	CALL eP	JMP Ev	JMP Ep	PUSH Ev	

Opcodes determined by bits 5,4,3 of modR/M byte:

Group	mod nnn R/M							
	000	001	010	011	100	101	110	111
6	SLDT Ew	STR Ew	LLDT Ew	LTR Ew	VERR Ew	VERW Ew		
7	SGDT Ms	SIDT Ms	LGDT Ms	LIDT Ms	SMSW Ew		LMSW Ew	
8					BT	BTS	BTR	BTC

图 6: make group

根据以上信息，对 exec.c 中的 opcode_table 进行补充。
填写的主要原则如下：

1. 首先通过查表获得对应的机器码。
2. 指令分为 ID 阶段和 EX 阶段两个阶段。在 ID 阶段的 helper 函数已经实现，需要根据源操作数和目的操作数的数据类型和位宽进行选择。

3. 常用的数据类型包括 E（寄存器或内存地址，通过 modR/M 指定），G（寄存器），I（立即数），J（地址偏移量）。例如：call 指令的译码阶段就是 J 类型的数据。
4. 框架中的译码方式举例：E2G，就是源操作数按照 E 类型译码，目的操作数按照 G 类型译码。
5. 当操作数只有 1 字节的时候，需要 IDEXW(...,...,1) 来指定位宽，一般情况下直接使用 IDEX 就可以，系统会根据机器环境来匹配位宽。
6. EX 的函数是我们自己实现的，后续需要绑定 Ehelper。

据此，填写如下。

- call

```
/* 0xe8 */      IDEX(J, call), EMPTY, EMPTY, EMPTY,
```

- push/pop

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

- ret

其中 0xc0，0xc1 是框架中已经给出的，填写 0xc3。

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

- sub/xor

类型较多，包括 0x28-0x2B，0x30-0x33，也包括 gp1(0x80,0x81,0x83)。

```
/* 0x28 */      IDEXW(G2E,sub,1), IDEX(G2E,sub), IDEXW(E2G,sub,1), IDEX(E2G,
    sub),
/* 0x30 */      IDEXW(G2E,xor,1), IDEX(G2E,xor), IDEXW(E2G,xor,1), IDEX(E2G,
    xor),

/* 0x80, 0x81, 0x83 */
make_group(gp1,
    EMPTY, EMPTY, EMPTY, EMPTY,
    EMPTY, EX(sub), EX(xor), EMPTY)
```

3. all-instr.h 中执行函数添加

在里面加入新的执行函数。

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

4. call/ret 指令实现

首先需要实现一些用到的 rtl 伪指令（rtl_push，rtl_pop，在 rtl.h 中）。

根据提供的伪代码比较容易实现，push 就是先减小 esp，然后将数据存入，pop 则是先取出数据，然后增大 esp。

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

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

还需要补充 SI 的取指逻辑。通过 instr_fetch 读取 eip 后面的内容。（因为 op->width 只可能是 1 或 4，所以只需要 2 个分支）

```
if(op->width == 4)
  op->simm = instr_fetch(eip,op->width);
else{
  op->simm = (int8_t)(instr_fetch(eip,op->width));
}
```

然后就可以实现 call/ret 指令（在 control.c 中）。

因为目标地址已经在译码阶段计算出来，所以 call 指令只需要把当前的下一条指令压栈，并且指定这条指令需要跳转，这样执行之后就会跳转到 jmp_eip。

对于 ret 指令，需要从栈中取出地址，并且复制给 jmp_eip，然后指定该指令跳转即可。

```
make_EHelper(call) {
  rtl_push(&decoding.seq_eip);
  decoding.is_jmp = 1;
  print_asm("call %x", decoding.jmp_eip);
}

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

5. push/pop 指令实现

因为在上一步已经实现了 push/pop 的 rtl 指令，这里相对简单。push 就是把 dest 的数值通过 rtl_push 压栈，pop 就是弹栈，并且写入 dest。

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

```
}

make_EHelper(pop) {
  //TODO();
  rtl_pop(&t1);
  operand_write(id_dest,&t1);
  print_asm_template1(pop);
}
```

6. eflags 实现

在很多算数指令中都需要更新 eflags 寄存器，表示运算状态。其结构如图7所示。

图 7: eflags 寄存器

要关注的位包括:

- CF：无符号整形运算溢出（进位/借位）时为 1，否则为 0。
- ZF：结果为 0 时为 1，否则为 0。
- SF：有符号整形的最高有效位（0 正 1 负）。
- IF：对可屏蔽中断的响应，1 响应 0 禁止。
- OF：有符号整形的溢出，溢出为 1，否则为 0。

实现的结构体如下。(在 reg.h 中添加)

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

然后，在 restart 函数中增加 eflags 的初始化（查看手册得到初始值为 2)。

```
cpu.eflags = 0x00000002;
```

在 rtl.h 中增加对应的宏定义（更新和获取 EFLAGS 位）和 ZF,SF 更新 rtl 指令。ZF 只需要判断是不是 0，SF 要返回 result 的最高有效位。

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


static inline void rtl_update_ZF(const rtlreg_t* result, int width) {
  cpu.ZF = ((*result) & ((0xFFFFFFFF) >> ((4-width) * 8))) ? 0 : 1 ;
}

static inline void rtl_update_SF(const rtlreg_t* result, int width) {
  cpu.SF = ((*result) >> (width * 8 - 1)) & 0x1 ;
}
```

7. sub/xor 指令实现

首先需要完成 rtl 指令 msb 的编写，功能为取出源操作数的符号位。

```
static inline void rtl_msb(rtlreg_t* dest, const rtlreg_t* src1, int width) {
  (*dest) = ((*src1) >> (width * 8 - 1)) & 0x1;
}
```

然后完成执行函数。不但需要完成算数运算和写回，也要同步对 eflag 中 ZF.SF,CF,OF 进行计算更新。sub 在 arith.c 中，xor 在 logic.c 中。

查阅手册可以知道，xor 的 CF/OF 被指定为 0，不用计算，如图8所示。

```
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 8: XOR

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

```

```
  rtl_sltu(&t3, &id_dest->val, &t2);
  rtl_set_CF(&t3);

  rtl_update_ZFSF(&t2, id_dest->width);

  rtl_xor(&t0, &id_dest->val, &id_src->val);
  rtl_xor(&t1, &id_dest->val, &t2);
  rtl_and(&t0, &t0, &t1);
  rtl_msb(&t0, &t0, id_dest->width);
  rtl_set_OF(&t0);

  print_asm_template2(sub);
}

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

  print_asm_template2(xor);
}
```

8. cmd_info 的更新

为了能够更好的检验 eflags，在 info r 增加打印 eflags 信息。

```
static int cmd_info(char *args) {
  //...
  if(!strcmp(strtok(args," "),"r")){
    printf("Registers info:\n");
    //...
    printf("EFLAGS:\n");
    printf("CF:%d ZF:%d SF:%d IF:%d OF:%d\n",cpu.CF,cpu.ZF,cpu.SF,cpu.IF,cpu.
        OF);
    return 0;
  }
  //...
}
```

(二) 测试结果

1. 整体测试

测试结果如图9所示，通过。

```
latesoon@latesoon-virtual-machine:~/ics2017/nexus-am/tests/cputest$ make ARCH=x86-nemu ALL=dummy run
Building dummy [x86-nemu]
Building am [x86-nemu]
make[2]: *** 没有指明目标并且找不到 makefile。 停止。
[src/monitor/monitor.c,65,load_img] The image is /home/latesoon/ics2017/nexus-am/tests/cputest/build/dummy-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 16:04:40, Apr  1 2025
For help, type "help"
(nemu) c
nemu: HIT GOOD TRAP at eip = 0x0010001b

(nemu) q
 dummy
```

图 9: 阶段 1 测试结果

2. 对于 call/push 的单步测试验证

可以看到，call 后，EIP 跳转到 0x100010，ESP-4（因为分配了空间用于存储 call 之前的下一条指令位置）。

push EBP 后，可以看到 ESP 再次-4，通过扫描内存可以看到 ESP 开始的内存分别保存了 EBP 的值以及 call 指令保存的指令位置。

验证通过。

```
latesoon@latesoon-virtual-machine:~/ics2017/nexus-am/tests/cputest$ make ARCH=x86-nemu ALL=dummy run
Building dummy [x86-nemu]
Building am [x86-nemu]
make[2]: *** 没有指明目标并且找不到 makefile。 停止。
[src/monitor/monitor.c,65,load_img] The image is /home/latesoon/ics2017/nexus-am/tests/cputest/build/dummy-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 16:04:40, Apr  1 2025
For help, type "help"
(nemu) si
  100000:   bd 00 00 00 00                      movl $0x0,%ebp
(nemu) si
  100005:   bc 00 7c 00 00                      movl $0x7c00,%esp
(nemu) info r
Registers info:
EAX:0x76FF0EE0 ECX:0x22EC8D3B
EDX:0x42F6DDAE EBX:0x5CA1CF5E
ESP:0x00007C00 EBP:0x00000000
ESI:0x0AC7CDF7 EDI:0x6C701908
EIP:0x0010000A
EFLAGS:
CF:0 ZF:0 SF:0 IF:0 OF:0
(nemu) si
  10000a:   e8 01 00 00 00                      call 100010
(nemu) info r
Registers info:
EAX:0x76FF0EE0 ECX:0x22EC8D3B
EDX:0x42F6DDAE EBX:0x5CA1CF5E
ESP:0x00007BFC EBP:0x00000000
ESI:0x0AC7CDF7 EDI:0x6C701908
EIP:0x00100010
EFLAGS:
CF:0 ZF:0 SF:0 IF:0 OF:0
(nemu) si
  100010:   55                                  pushl %ebp
(nemu) info r
Registers info:
EAX:0x76FF0EE0 ECX:0x22EC8D3B
EDX:0x42F6DDAE EBX:0x5CA1CF5E
ESP:0x00007BF8 EBP:0x00000000
ESI:0x0AC7CDF7 EDI:0x6C701908
EIP:0x00100011
EFLAGS:
CF:0 ZF:0 SF:0 IF:0 OF:0
(nemu) x 20 $esp
Address     Data
0x00007BF8: 00 00 00 00 0F 00 10 00
0x00007C00: 00 00 00 00 00 00 00 00
0x00007C08: 00 00 00 00
```

图 10: call/push 的单步测试验证

3. 对于 sub 的单步测试验证

可以看到，ESP 从 0x7BF8 减小到了 0x7BF0，正确。虽然 EFLAGS 的数值都是 0，但这是正确的。

```
(nemu) info r
Registers info:
EAX:0x76FF0EE0 ECX:0x22EC8D3B
EDX:0x42F6DDAE EBX:0x5CA1CF5E
ESP:0x00007BF8 EBP:0x00007BF8
ESI:0x0AC7CDF7 EDI:0x6C701908
EIP:0x00100013
EFLAGS:
CF:0 ZF:0 SF:0 IF:0 OF:0
(nemu) si
  100013:   83 ec 08                                subl $0x8,%esp
(nemu) info r
Registers info:
EAX:0x76FF0EE0 ECX:0x22EC8D3B
EDX:0x42F6DDAE EBX:0x5CA1CF5E
ESP:0x00007BF0 EBP:0x00007BF8
ESI:0x0AC7CDF7 EDI:0x6C701908
EIP:0x00100016
EFLAGS:
CF:0 ZF:0 SF:0 IF:0 OF:0
```

图 11: sub 的单步测试验证

4. 对于 xor/pop/ret 的单步测试验证

- xor：EAX=0，ZF=1，正确。
- pop：EBP 恢复到了先前 push 的 0xEBF8，ESP+4，正确。
- ret：恢复到函数调用前的地址，HIT GOOD TRAP，正确。

```
(nemu) info r
Registers info:
EAX:0x76FF0EE0 ECX:0x22EC8D3B
EDX:0x42F6DDAE EBX:0x5CA1CF5E
ESP:0x00007BE8 EBP:0x00007BE8
ESI:0x0AC7CDF7 EDI:0x6C701908
EIP:0x00100023
EFLAGS:
CF:0 ZF:0 SF:0 IF:0 OF:0
(nemu) si
  100023:   31 c0                                   xorl %eax,%eax
(nemu) info r
Registers info:
EAX:0x00000000 ECX:0x22EC8D3B
EDX:0x42F6DDAE EBX:0x5CA1CF5E
ESP:0x00007BE8 EBP:0x00007BE8
ESI:0x0AC7CDF7 EDI:0x6C701908
EIP:0x00100025
EFLAGS:
CF:0 ZF:1 SF:0 IF:0 OF:0
(nemu) si
  100025:   5d                                      popl %ebp
(nemu) info r
Registers info:
EAX:0x00000000 ECX:0x22EC8D3B
EDX:0x42F6DDAE EBX:0x5CA1CF5E
ESP:0x00007BEC EBP:0x00007BF8
ESI:0x0AC7CDF7 EDI:0x6C701908
EIP:0x00100026
EFLAGS:
CF:0 ZF:1 SF:0 IF:0 OF:0
(nemu) si
  100026:   c3                                      ret
(nemu) info r
Registers info:
EAX:0x00000000 ECX:0x22EC8D3B
EDX:0x42F6DDAE EBX:0x5CA1CF5E
ESP:0x00007BF0 EBP:0x00007BF8
ESI:0x0AC7CDF7 EDI:0x6C701908
EIP:0x0010001B
EFLAGS:
CF:0 ZF:1 SF:0 IF:0 OF:0
(nemu) si
nemu: HIT GOOD TRAP at eip = 0x0010001b
```

图 12: xor/pop/ret 的单步测试验证

(三) 问答题：大端架构与小端架构

- Q:

  Motorola 68k 系列的处理器都是大端架构的. 现在问题来了, 考虑以下两种情况:

1. 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)。

2. 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。

在这两种情况下, 你需要注意些什么问题? 为什么会产生这些问题? 怎么解决它们?

- A:

在这两种情况下，原封不动地从内存读入 imm 变量中读取并解释会导致错误。

例如，在第一种情况下，由于在 Motorola 68k 是大端架构，而 NEMU 原本设计为模拟 x86（小端）。当 NEMU 通过读取指令中的立即数时，内存中的小端字节序列（如 34 12 00 00）会被大端主机直接解释为大端数值（0x34120000），而非预期的小端值（0x00001234），导致立即数错误。

第二种情况类似，用大端架构的模拟器运行在 x86 架构中也会遇到类似的问题。

为此，我们需要在内存访问接口中添加显式的字节序转换，将内存中的序列和模拟器所需要的序列之间进行转换，保证数据始终满足内存预期。

四、 阶段 2 实验方法与结果

(一) 编码内容：实现更多的指令

1. lea 指令实现

该指令已经在框架中实现，补充对应的 opcode_table。

```
/* 0x8c */      EMPTY, IDEX(lea_M2G, lea), EMPTY, EMPTY,

make_EHelper(lea);
```

2. cmp/add/adc/sbb/or/and 指令实现及 sub/xor 的 opcode_table 补充

由于 add/adc 指令中需要使用 rtl_not，对此进行补全。

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

然后，填写对应的 opcode_table，并补充 EHelper 定义。

```
make_EHelper(add);
make_EHelper(or);
make_EHelper(adc);
make_EHelper(sbb);
make_EHelper(and);
make_EHelper(cmp);

make_group(gp1,
    EX(add), EX(or), EX(adc), EX(sbb),
    EX(and), EX(sub), EX(xor), EX(cmp))
```

```
11
12  /* 0x00 */  IDEXW(G2E,add,1), IDEX(G2E,add), IDEXW(E2G,add,1), IDEX(E2G,
        add),
13  /* 0x04 */  IDEXW(I2a,add,1), IDEX(I2a,add), EMPTY, EMPTY,
14  /* 0x08 */  IDEXW(G2E,or,1), IDEX(G2E,or), IDEXW(E2G,or,1), IDEX(E2G,or),
15  /* 0x0c */  IDEXW(I2a,or,1), IDEX(I2a,or), EMPTY, EX(2byte_esc),
16  /* 0x10 */  IDEXW(G2E,adc,1), IDEX(G2E,adc), IDEXW(E2G,adc,1), IDEX(E2G,
        adc),
17  /* 0x14 */  IDEXW(I2a,adc,1), IDEX(I2a,adc), EMPTY, EMPTY,
18  /* 0x18 */  IDEXW(G2E,sbb,1), IDEX(G2E,sbb), IDEXW(E2G,sbb,1), IDEX(E2G,
        sbb),
19  /* 0x1c */  IDEXW(I2a,sbb,1), IDEX(I2a,sbb), EMPTY, EMPTY,
20  /* 0x20 */  IDEXW(G2E,and,1), IDEX(G2E,and), IDEXW(E2G,and,1), IDEX(E2G,
        and),
21  /* 0x24 */  IDEXW(I2a,and,1), IDEX(I2a,and), EMPTY, EMPTY,
22  /* 0x28 */  IDEXW(G2E,sub,1), IDEX(G2E,sub), IDEXW(E2G,sub,1), IDEX(E2G,
        sub),
23  /* 0x2c */  IDEXW(I2a,sub,1), IDEX(I2a,sub), EMPTY, EMPTY,
24  /* 0x30 */  IDEXW(G2E,xor,1), IDEX(G2E,xor), IDEXW(E2G,xor,1), IDEX(E2G,
        xor),
25  /* 0x34 */  IDEXW(I2a,xor,1), IDEX(I2a,xor), EMPTY, EMPTY,
26  /* 0x38 */  IDEXW(G2E,cmp,1), IDEX(G2E,cmp), IDEXW(E2G,cmp,1), IDEX(E2G,
        cmp),
27  /* 0x3c */  IDEXW(I2a,cmp,1), IDEX(I2a,cmp), EMPTY, EMPTY,
```

- cmp

  查阅文档可以发现，cmp 与 sub 类似，但是不保存结果。在 sub 基础上去掉保存结果的一行就可以。

```
Operation

LeftSRC - SignExtend(RightSRC);
(* CMP does not store a result; its purpose is to set the flags *)


Description

CMP subtracts the second operand from the first but, unlike the SUB
instruction, does not store the result; only the flags are changed. CMP is
typically used in conjunction with conditional jumps and the SETcc
instruction. (Refer to Appendix D for the list of signed and unsigned flag
tests provided.) If an operand greater than one byte is compared to an
immediate byte, the byte value is first sign-extended.
```

图 13: CMP

```
1  make_EHelper(cmp) {
2    rtl_sub(&t2, &id_dest->val, &id_src->val);
3
4    rtl_sltu(&t3, &id_dest->val, &t2);
5    rtl_set_CF(&t3);
```

```
6
7     rtl_update_ZFSF(&t2, id_dest->width);
8
9     rtl_xor(&t0, &id_dest->val, &id_src->val);
10    rtl_xor(&t1, &id_dest->val, &t2);
11    rtl_and(&t0, &t0, &t1);
12    rtl_msb(&t0, &t0, id_dest->width);
13    rtl_set_OF(&t0);
14
15    print_asm_template2(cmp);
16 }
```

- add

  与 adc 类似（但不进行带溢出的运算），进行加法，并更新 eflags。

```
1  make_EHelper(add) {
2    //TODO();
3    rtl_add(&t2, &id_dest->val, &id_src->val);
4    operand_write(id_dest, &t2);
5
6    rtl_sltu(&t3, &t2, &id_dest->val);
7    rtl_set_CF(&t3);
8
9    rtl_update_ZFSF(&t2, id_dest->width);
10
11   rtl_xor(&t0, &id_dest->val, &id_src->val);
12   rtl_not(&t0);
13   rtl_xor(&t1, &id_dest->val, &t2);
14   rtl_and(&t0, &t0, &t1);
15   rtl_msb(&t0, &t0, id_dest->width);
16   rtl_set_OF(&t0);
17
18   print_asm_template2(add);
19 }
```

- adc/abb

  这两条指令在框架中已经实现，不需要自己实现。

- and/or

  逻辑运算，对于 EFLAGS 的定义与 xor 类似。图14展示了 or 指令的定义。直接把 rtl_xor 替换成 rtl_and 或 rtl_or 就可以了。

```
Operation

DEST ← DEST OR SRC;
CF ← 0;
OF ← 0


Description

OR computes the inclusive OR of its two operands and places the result
in the first operand. Each bit of the result is 0 if both corresponding
bits of the operands are 0; otherwise, each bit is 1.


Flags Affected

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

图 14: OR

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

  print_asm_template2(and);
}

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

  print_asm_template2(or);
}
```


3. inc/dec 指令实现

inc/dec 完成的是自增 1/自减 1，图15显示了 inc 的操作手册说明。需要注意的是，手册指出这两条指令并不会更新 CF，因此可以忽略它。

```
Operation

DEST ← DEST + 1;


Description

INC adds 1 to the operand. It does not change the carry flag. To affect
the carry flag, use the ADD instruction with a second operand of 1.


Flags Affected

OF, SF, ZF, AF, and PF as described in Appendix C
```

图 15: INC

```
make_EHelper(inc);
make_EHelper(dec);

make_group(gp4,
    EX(inc), EX(dec), EMPTY, EMPTY,
    EMPTY, EMPTY, EMPTY, EMPTY)

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

make_EHelper(inc) {
  rtl_addi(&t2, &id_dest->val, 1);
  rtl_msb(&t0, &id_dest->val, id_dest->width);
  operand_write(id_dest, &t2);
  rtl_update_ZFSF(&t2, id_dest->width);

  rtl_msb(&t1, &t2, id_dest->width);
  rtl_xori(&t0, &t0, 1);
  rtl_and(&t1, &t0, &t1);
  rtl_set_OF(&t1);

  print_asm_template1(inc);
}

make_EHelper(dec) {
  rtl_subi(&t2, &id_dest->val, 1);
  rtl_msb(&t0, &id_dest->val, id_dest->width);
  operand_write(id_dest, &t2);
  rtl_update_ZFSF(&t2, id_dest->width);

  rtl_msb(&t1, &t2, id_dest->width);
  rtl_xori(&t1, &t1, 1);
  rtl_and(&t1, &t0, &t1);
```

```
35    rtl_set_OF(&t1);
36
37    print_asm_template1(dec);
38  }
```

4. nop(xchg) 指令实现

结合群内助教讲解和反汇编文件可以知道，add.c 中的 66 90 是将 ax 寄存器自身进行互换，是没有意义的，因此可以直接指定为 nop。（其中 66 表示的是操作数 16bit 前缀）

```
1  /* 0x90 */      EX(nop), EMPTY, EMPTY, EMPTY,
2
3  make_EHelper(nop);
```

5. setcc 指令实现

首先对 opcode_table 进行补充。

```
1  make_EHelper(setcc);
2  //2byte
3    /* 0x90 */    IDEXW(E,setcc,1), IDEXW(E,setcc,1), IDEXW(E,setcc,1), IDEXW(E
       ,setcc,1),
4    /* 0x94 */    IDEXW(E,setcc,1), IDEXW(E,setcc,1), IDEXW(E,setcc,1), IDEXW(E
       ,setcc,1),
5    /* 0x98 */    IDEXW(E,setcc,1), IDEXW(E,setcc,1), IDEXW(E,setcc,1), IDEXW(E
       ,setcc,1),
6    /* 0x9c */    IDEXW(E,setcc,1), IDEXW(E,setcc,1), IDEXW(E,setcc,1), IDEXW(E
       ,setcc,1),
```

根据图16指导书中对于 setcc 的说明，补充 setcc 中逻辑。

Opcode	Instruction	Clocks	Description
0F 97	SETA r/m8	4/5	Set byte if above (CF=0 and ZF=0)
0F 93	SETAE 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	SETNAE 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)

图 16: SETCC

因为有一半情况是另一半的取反，所以这里只需要实现一半的情况。

```
void rtl_setcc(rtlreg_t* dest, uint8_t subcode) {
  //...
  switch (subcode & 0xe) {
    case CC_O: rtl_get_OF(dest);break;
    case CC_B: rtl_get_CF(dest);break;
    case CC_E: rtl_get_ZF(dest);break;
    case CC_BE:
      rtl_get_CF(&t0);
      rtl_get_ZF(&t1);
      rtl_or(dest,&t0,&t1);
      break;
    case CC_S: rtl_get_SF(dest);break;
    case CC_L:
      rtl_get_OF(&t0);
      rtl_get_SF(&t1);
      rtl_xor(dest,&t0,&t1);
      break;
    case CC_LE:
      rtl_get_OF(&t0);
      rtl_get_SF(&t1);
      rtl_get_ZF(&t2);
      rtl_xor(&t0,&t0,&t1);
      rtl_or(dest,&t0,&t2);
      break;
    default: panic("should not reach here");
    case CC_P: panic("n86 does not have PF");
  }
  //...
}
```

6. movzx/movsx 指令实现

首先需要补充对应的 rtl 指令：rtl_sext。

```
static inline void rtl_sext(rtlreg_t* dest, const rtlreg_t* src1, int width)
    {
  *dest = ((int32_t)( *src1 << (32 - width * 8) )) >> ( 32 - width * 8);
}
```

该指令已经实现，补充填表即可。

```
make_EHelper(movzx);
make_EHelper(movsx);
//2byte
/* 0xb4 */      EMPTY, EMPTY, IDEXW(mov_E2G,movzx,1), IDEXW(mov_E2G,movzx,2),
/* 0xbc */      EMPTY, EMPTY, IDEXW(mov_E2G,movsx,1), IDEXW(mov_E2G,movsx,2),
```

7. test 指令实现

test 本质上就是一个不保存结果只更新标记位的与运算，通过去掉 and 运算的结果保存即可实现。

```
make_EHelper(test);

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

/* 0x84 */      IDEXW(G2E,test,1), IDEX(G2E,test), EMPTY, EMPTY,

//此行实际上是在第3阶段测试Hello World才需要添加
/* 0xa8 */      IDEXW(I2a,test,1), IDEX(I2a,test), EMPTY, EMPTY,

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

  print_asm_template2(test);
}
```

8. jmp/jcc 指令实现

jmp/jcc 的执行函数已经给出，需要补充 opcode_table。

```
make_EHelper(jcc);
make_EHelper(jmp);

//1byte
  /* 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),
  /* 0xe8 */    IDEX(J, call), IDEX(J,jmp), EMPTY, IDEXW(J,jmp,1),

//2byte
  /* 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),
```

9. shr/shl/sar 指令实现

逻辑右移，逻辑左移，算数右移指令可以直接调用对应的 rtl 指令来实现。在 nemu 中这些指令不需要更新 CF 和 OF，因此只需要更新 ZF 和 SF 就可以。

需要注意的是，算数右移运算需要先进行符号扩展，以确保能够正确移位。

```
make_EHelper(shr);
make_EHelper(shl);
make_EHelper(sar);

make_group(gp2,
    EMPTY, EMPTY, EMPTY, EMPTY,
    EX(shl), EX(shr), EMPTY, EX(sar))

make_EHelper(sar) {
  rtl_sext(&t2, &id_dest->val, id_dest->width);
  rtl_sar(&t0, &t2, &id_src->val);
  rtl_update_ZFSF(&t0, id_dest->width);
  operand_write(id_dest, &t0);
  print_asm_template2(sar);
}

make_EHelper(shl) {
  rtl_shl(&t0, &id_dest->val, &id_src->val);
  rtl_update_ZFSF(&t0, id_dest->width);
  operand_write(id_dest, &t0);
  print_asm_template2(shl);
}

make_EHelper(shr) {
  rtl_shr(&t0, &id_dest->val, &id_src->val);
  rtl_update_ZFSF(&t0, id_dest->width);
  operand_write(id_dest, &t0);
  print_asm_template2(shr);
}
```

10. not 指令实现

not 指令不影响 eflags 位，直接调用 rtl_not 实现，如图17所示。

```
Operation

r/m ← NOT r/m;

Description

NOT inverts the operand; every 1 becomes a 0, and vice versa.

Flags Affected

None
```

图 17: NOT

```
make_EHelper(not);

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

make_EHelper(not) {
  rtl_not(&id_dest->val);
  operand_write(id_dest, &id_dest->val);

  print_asm_template1(not);
}
```

11. mul/imul/div/idiv 指令实现

对应指令已经在框架中实现，需要填充 opcode_table。

```
make_EHelper(mul);
make_EHelper(imul1);
make_EHelper(imul2);
make_EHelper(div);
make_EHelper(idiv);

make_group(gp3,
    IDEX(test_I,test), EMPTY, EX(not), EMPTY,
    EX(mul), EX(imul1), EX(div), EX(idiv))
/* 0xac */      EMPTY, EMPTY, EMPTY, IDEX(E2G,imul2),
```

12. cltd(cwd/cdq) 指令实现

cltd(cwd/cdq) 指令的说明如图18所示，本质上是把 AX/EAX 进行符号扩展，扩展的位存在 DX/EDX 中。

```
Operation

IF OperandSize = 16 (* CWD instruction *)
THEN
   IF AX < 0 THEN DX ← 0FFFFH; ELSE DX ← 0; FI;
ELSE (* OperandSize = 32, CDQ instruction *)
   IF EAX < 0 THEN EDX ← 0FFFFFFFFH; ELSE EDX ← 0; FI;
FI;
```

图 18: CLTD

```
make_EHelper(cltd);

/* 0x98 */      EMPTY, EX(cltd), EMPTY, EMPTY,

make_EHelper(cltd) {
  if (decoding.is_operand_size_16) {
    rtl_msb(&t0,&cpu.eax,2);
    cpu.gpr[2]._16 = t0 ? 0xFFFF : 0;
  }
  else {
    rtl_msb(&t0,&cpu.eax,4);
    cpu.edx = t0 ? 0xFFFFFFFF : 0;
  }

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

13. leave 指令实现

leave 指令的说明如图19所示。其作用是上层程序退出，具体来说就是把 EBP 赋值给 ESP，然后把 EBP 的值修改为 pop 指令弹栈的值。

LEAVE — High Level Procedure Exit

Opcode	Instruction	Clocks	Description
C9	LEAVE	4	Set SP to BP, then pop BP
C9	LEAVE	4	Set ESP to EBP, then pop EBP

```
Operation

IF StackAddrSize = 16
THEN
   SP ← BP;
ELSE (* StackAddrSize = 32 *)
   ESP ← EBP;
FI;
IF OperandSize = 16
THEN
   BP ← Pop();
ELSE (* OperandSize = 32 *)
   EBP ← Pop();
FI;
```

图 19: LEAVE

为实现这一操作，首先需要实现 rtl_mv。

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

make_EHelper(leave);

/* 0xc8 */      EMPTY, EX(leave), EMPTY, EMPTY,

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

  print_asm("leave");
}
```

14. call_rm/jmp_rm 指令实现

本质上还是 call/jmp 指令，但是这里的执行实现和之前的不同。之前的地址计算方式是通过当前 EIP 位置 + 偏移量获得，而此处的地址计算方式则是直接给出，适用于跳转位置较远的情况，因此需要设计新的执行函数。

其中 jmp_rm 的函数原型已经在框架给出，首先实现 call_rm 函数。

```
make_EHelper(call_rm);
make_EHelper(jmp_rm);

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

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

15. push 指令 opcode_table 补充

在此阶段的测试中，因为发现的指令缺失情况进行的补充。

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

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

16. rtl_eq0/rtl_eqi/rtl_neq0 实现

在 debug 过程中，因为怀疑这些指令缺失可能导致 assert 不能通过所以进行了实现。后期分析这些指令并不会造成影响（部分指令在第 3 阶段使用，截至第 2 阶段这些指令没有在执行函数中使用）。

```
static inline void rtl_eq0(rtlreg_t* dest, const rtlreg_t* src1) {
  rtl_sltui(dest, src1, 1);
}

static inline void rtl_eqi(rtlreg_t* dest, const rtlreg_t* src1, int imm) {
  rtl_xori(dest, src1, imm);
  rtl_eq0(dest, dest);
}

static inline void rtl_neq0(rtlreg_t* dest, const rtlreg_t* src1) {
  rtl_eq0(dest, src1);
  rtl_eq0(dest, dest);
}
```

17. 测试结果

逐一对每一个测试样例进行单独测试，结果如图20所示，测试通过。

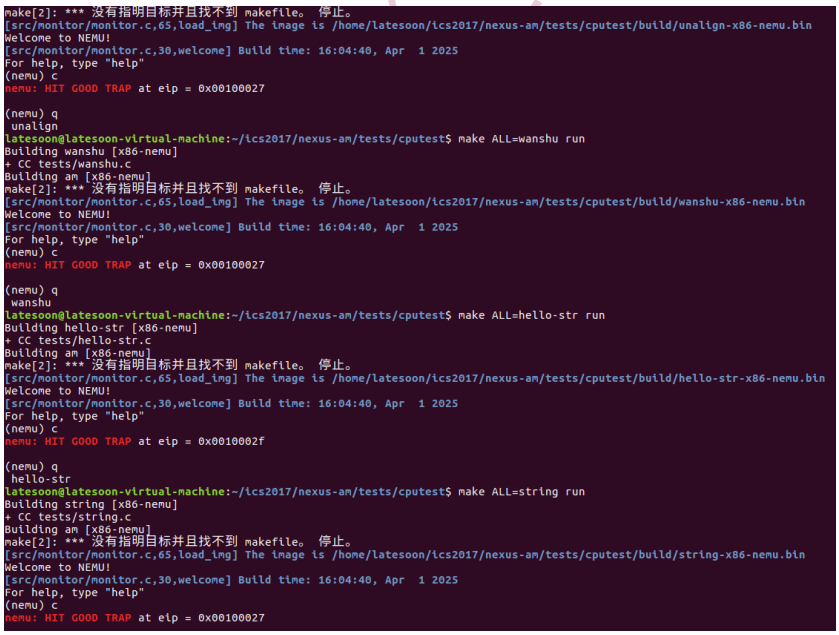

图 20: 实现更多的指令测试结果

(二) 编码内容：实现 differential testing

首先需要打开 DIFF_TEST 宏定义的注释，以启用（在 common.h 中）。

```
#define DEBUG
#define DIFF_TEST
```

在此之后运行 nemu，将会显示 Connect to QEMU successfully。

```
+ CC src/cpu/decode/modrm.c
+ CC src/cpu/decode/decode.c
+ CC src/cpu/reg.c
+ LD build/nemu
[src/monitor/diff-test/diff-test.c,96,init_difftest] Connect to QEMU successfully
[src/monitor/monitor.c,65,load_img] The image is /home/latesoon/ics2017/nexus-am/tests/cputest/build/switch-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 00:50:59, Apr  4 2025
For help, type "help"
```

图 21: differential testing

然后补充 difftest_step，此处要补充的内容主要是将 NEMU 和 QEMU 的 9 个寄存器进行比较，若结果不同则打印结果并且中断程序。

```
void difftest_step(uint32_t eip) {
  //...
  if(r.eax != cpu.eax || r.ebx != cpu.ebx || r.ecx != cpu.ecx || r.edx != cpu.edx ||
    r.esp != cpu.esp || r.ebp != cpu.ebp || r.esi !=cpu.esi || r.edi != cpu.edi || r.eip != cpu.eip){
    diff = true;
    printf("Difftest failed!\n");
    printf("Registers info in NEMU:\n");
    printf("EAX:0x%08X ECX:0x%08X\n",cpu.eax,cpu.ecx);
    printf("EDX:0x%08X EBX:0x%08X\n",cpu.edx,cpu.ebx);
    printf("ESP:0x%08X EBP:0x%08X\n",cpu.esp,cpu.ebp);
    printf("ESI:0x%08X EDI:0x%08X\n",cpu.esi,cpu.edi);
    printf("EIP:0x%08X\n",cpu.eip);
    printf("EFLAGS:0x%08X\n",cpu.eflags);
    printf("CF:%d ZF:%d SF:%d IF:%d OF:%d\n",cpu.CF,cpu.ZF,cpu.SF,cpu.IF,cpu.OF);
    printf("Registers info in QEMU:\n");
    printf("EAX:0x%08X ECX:0x%08X\n",r.eax,r.ecx);
    printf("EDX:0x%08X EBX:0x%08X\n",r.edx,r.ebx);
    printf("ESP:0x%08X EBP:0x%08X\n",r.esp,r.ebp);
    printf("ESI:0x%08X EDI:0x%08X\n",r.esi,r.edi);
    printf("EIP:0x%08X\n",r.eip);
    printf("EFLAGS:0x%08X\n",r.eflags);
  }
  //...
}
```

完成后在 nemu 根目录下通过 bash runall.sh, 对所有样例进行回归测试, 全部通过, 如图22所示。

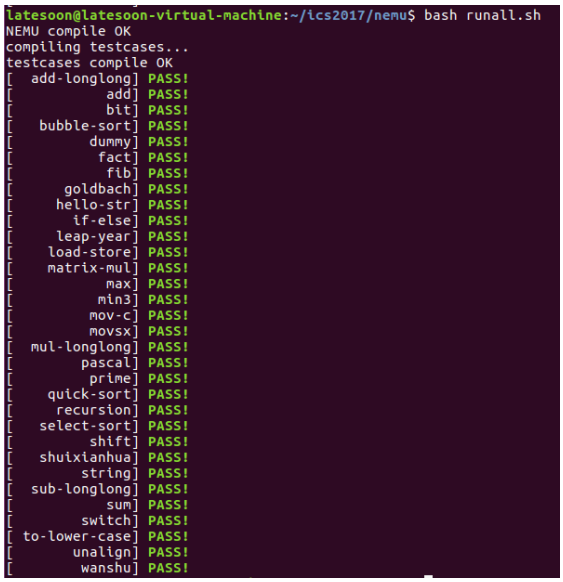

图 22: 回归测试

(三) 问答题

1. 堆和栈在哪里?

- Q:

  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?

- A:

  堆和栈是程序运行时动态需求的内存区域，其内容无法在编译时确定，也为了节省空间及安全考虑，因此不存储在可执行文件中。

  在程序运行时，堆和栈由操作系统动态分配和管理。栈由操作系统在进程启动时自动分配，通常位于虚拟地址空间的高地址端，向低地址增长。堆由运行时环境管理，位于虚拟地址空间的低地址端，向高地址增长。

2. NEMU 的本质

- Q:

  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?

- A:

  并行支持，软件生态等。

3. 捕捉死循环

- Q:

  NEMU 除了作为模拟器之外, 还具有简单的调试功能, 可以设置断点, 查看程序状态. 如果让你为 NEMU 添加如下功能” 当用户程序陷入死循环时, 让用户程序暂停下来, 并输出相应的提示信息” 你觉得应该如何实现?

- A:

  可以记录指令执行历史，当出现循环的重复序列的时候则为死循环。也可以设置每条指令执行次数上限，当某一条指令执行次数超过阈值则认为是死循环。

五、 阶段 3 实验方法与结果

(一) 编码内容及测试结果

1. 运行 Hello World

为运行 Hello World，需要实现 in/out 指令，这些指令完成了 nemu 与设备的交互。该指令实现的功能主要是把某一个端口的数据存入/取出。

```
  /* 0xe4 */     IDEXW(in_I2a,in,1), IDEX(in_I2a,in), IDEXW(out_a2I,out,1),
      IDEX(out_a2I,out),
  /* 0xec */     IDEXW(in_dx2a,in,1), IDEX(in_dx2a,in), IDEXW(out_a2dx,out,1),
      IDEX(out_a2dx,out),

make_EHelper(in);
make_EHelper(out);

make_EHelper(in) {
  t2 = pio_read(id_src->val,id_dest->width);
  operand_write(id_dest, &t2);
  print_asm_template2(in);

#ifdef DIFF_TEST
  diff_test_skip_qemu();
#endif
}

make_EHelper(out) {
  pio_write(id_dest->val,id_src->width,id_src->val);
  print_asm_template2(out);

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

此外，还需要打开以下宏定义

```
//common.h
#define HAS_IOE

//trm.c
#define HAS_SERIAL
```

运行程序，打印 10 次 Hello World，最终 HIT GOOD TRAP，正确。

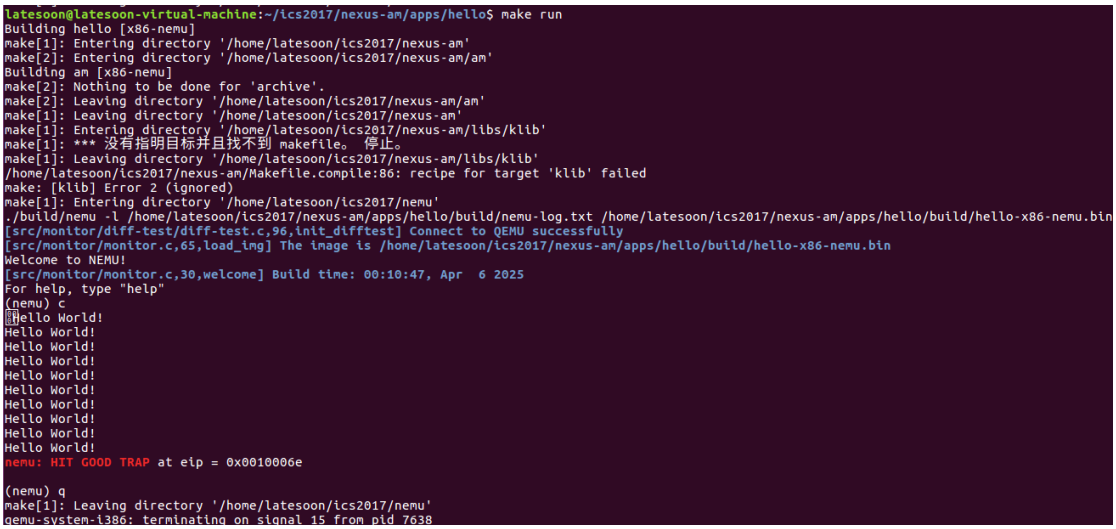

图 23: HELLO WORLD

2. 实现 IOE(1)——时钟

在 ioe.c 中，可以看到定义了全局变量 boot_time，其在 _ioe_init 中通过读取 RTC_PORT 初始化。因此，在 _uptime 的实现中，只需要计算当前时间与其的差值即可。

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

在 timetest 目录下运行 make run 测试，可以看到每秒打印一行，正确。

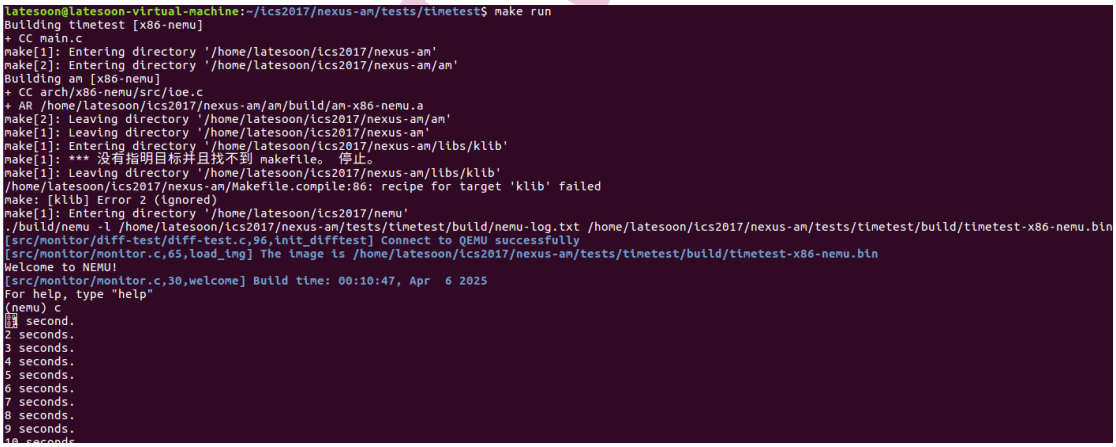

图 24: IOE

3. cwtl 指令实现 (Coremark 指令集依赖)

本质上就是一个 EAX 内部的符号扩展，如图25所示。

CBW/CWDE — Convert Byte to Word/Convert Word to Doubleword

Opcode	Instruction	Clocks	Description
98	CBW	3	AX ← sign-extend of AL
98	CWDE	3	EAX ← sign-extend of AX

Operation

```
IF OperandSize = 16 (* instruction = CBW *)
THEN AX ← SignExtend(AL);
ELSE (* OperandSize = 32, instruction = CWDE *)
   EAX ← SignExtend(AX);
FI;
```

图 25: Enter Caption

```
make_EHelper(cwtl);

  /* 0x98 */     EX(cwtl), EX(cltd), EMPTY, EMPTY,

make_EHelper(cwtl) {
  if (decoding.is_operand_size_16) {
    rtl_lr_b(&t1, R_AL);
    rtl_sext(&t1, &t1, 1);
    rtl_sr_w(R_AX, &t1);
  }
  else {
    rtl_lr_w(&t1, R_AX);
    rtl_sext(&t1, &t1, 2);
    rtl_sr_l(R_EAX, &t1);
  }
  print_asm(decoding.is_operand_size_16 ? "cbtw" : "cwtl");
}
```

4. neg 指令实现 (microbench 指令集依赖)

似乎第 2 部分已经要求实现了该指令，但是第 2 部分的所有样例都没有用到这一指令。原理很简单，只有 CF 的置位比较特殊，如图26所示。

Operation

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

图 26: NEG

实现如下:

```
make_EHelper(neg);

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

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

```
  operand_write(id_dest, &t2);

  rtl_neq0(&t3, &id_dest->val);
  rtl_set_CF(&t3);

  rtl_update_ZFSF(&t2, id_dest->width);

  rtl_xor(&t0, &id_dest->val, &id_src->val);
  rtl_xor(&t1, &id_dest->val, &t2);
  rtl_and(&t0, &t0, &t1);
  rtl_msb(&t0, &t0, id_dest->width);
  rtl_set_OF(&t0);

  print_asm_template1(neg);
}
```

5. rol 指令实现 (microbench 指令集依赖)

是一个循环左移操作，如图27所示。

```
(* ROL - Rotate Left *)
temp ← COUNT;
WHILE (temp ≠ 0)
DO
   tmpcf ← high-order bit of (r/m);
   r/m ← r/m * 2 + (tmpcf);
   temp ← temp - 1;
OD;
IF COUNT = 1
THEN
   IF high-order bit of r/m ≠ CF
   THEN OF ← 1;
   ELSE OF ← 0;
   FI;
ELSE OF ← undefined;
FI;
```

图 27: ROL

```
make_EHelper(rol);

make_group(gp2,
    EX(rol), EMPTY, EMPTY, EMPTY,
    EX(shl), EX(shr), EMPTY, EX(sar))

make_EHelper(rol){
  int shift = id_src->val % (id_dest->width * 8);
  rtl_shri(&t2, &id_dest->val, id_dest->width * 8 - shift);
        rtl_shli(&t1, &id_dest->val, shift);
        rtl_or(&t1, &t2, &t1);

  rtl_msb(&t3,&id_dest->val, id_dest->width);
```

```
  operand_write(id_dest, &t1);

  rtl_msb(&t0, &id_dest->val, id_dest->width);
  rtl_set_CF(&t0);

  rtl_xor(&t0, &t0, &t3);
  rtl_set_OF(&t0);
  print_asm_template1(rol);
}
```

6. 看看 NEMU 跑多快

首先需要关闭 DIFF_TEST 和 DEBUG 宏定义 (在 common.h 中)，以得到真实结果。

```
//#define DEBUG
//#define DIFF_TEST
```

对 Dhrystone，Coremark，microbench 三个测试集分别在 native 和 nemu 下进行测试，结果如下：

- Dhrystone

```
[src/monitor/monitor.c,65,load_img] The image is /home/latesoon/ics2017/nexus-am/apps/dhrystone/build/dhrystone-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 13:52:58, Apr  6 2025
For help, type "help"
(nemu) c
Dhrystone Benchmark, Version C, Version 2.2
Trying 500000 runs through Dhrystone.
Finished in 6904 ms
------------------------------------------------
Dhrystone PASS         149 Marks
                  vs. 100000 Marks (i7-6700 @ 3.40GHz)
nemu: HIT GOOD TRAP at eip = 0x0010006e

(nemu) q
make[1]: Leaving directory '/home/latesoon/ics2017/nemu'
```

图 28: Dhrystone-NEMU

```
/home/latesoon/ics2017/nexus-am/Makefile.compile:86: recipe for target 'klib' failed
make: [klib] Error 2 (ignored)
Dhrystone Benchmark, Version C, Version 2.2
Trying 500000 runs through Dhrystone.
Finished in 24 ms
------------------------------------------------
Dhrystone PASS         42927 Marks
                  vs. 100000 Marks (i7-6700 @ 3.40GHz)
```

图 29: Dhrystone-native

- Coremark

```
[src/monitor/monitor.c,65,load_img] The image is /home/latesoon/ics2017/nexus-am/apps/coremark/build/coremark-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 13:52:58, Apr  6 2025
For help, type "help"
(nemu) c
Running CoreMark for 1000 iterations
2K performance run parameters for coremark.
CoreMark Size    : 666
Total time (ms)  : 8137
Iterations       : 1000
Compiler version : GCC5.4.0 20160609
seedcrc          : 0xe9f5
[0]crclist       : 0xe714
[0]crcmatrix     : 0x1fd7
[0]crcstate      : 0x8e3a
[0]crcfinal      : 0xd340
Finised in 8137 ms.
==================================================
CoreMark PASS       549 Marks
               vs. 100000 Marks (i7-6700 @ 3.40GHz)
nemu: HIT GOOD TRAP at eip = 0x0010006e
```

图 30: Coremark-NEMU

```
Running CoreMark for 1000 iterations
2K performance run parameters for coremark.
CoreMark Size    : 666
Total time (ms)  : 51
Iterations       : 1000
Compiler version : GCC5.4.0 20160609
seedcrc          : 0xe9f5
[0]crclist       : 0xe714
[0]crcmatrix     : 0x1fd7
[0]crcstate      : 0x8e3a
[0]crcfinal      : 0xd340
Finised in 51 ms.
==================================================
CoreMark PASS       87619 Marks
               vs. 100000 Marks (i7-6700 @ 3.40GHz)
```

图 31: Coremark-native

- microbench

  首先通过 make INPUT=TEST run 进行正确性测试。

```
[src/monitor/monitor.c,65,load_img] The image is /home/latesoon/ics2017/nexus-am/apps/microbench/build/microbench-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 15:01:34, Apr  6 2025
For help, type "help"
(nemu) c
[qsort] Quick sort: * Passed.
[queen] Queen placement: * Passed.
[bf] Brainf**k interpreter: * Passed.
[fib] Fibonacci number: * Passed.
[sieve] Eratosthenes sieve: * Passed.
[15pz] A* 15-puzzle search: * Passed.
[dinic] Dinic's maxflow algorithm: * Passed.
[lzip] Lzip compression: * Passed.
[ssort] Suffix sort: * Passed.
[md5] MD5 digest: * Passed.
==============================================
MicroBench PASS
nemu: HIT GOOD TRAP at eip = 0x00100032

(nemu) q
```

图 32: microbench 正确性测试

然后 make clean 清空先前编译文件，再 make run 进行测试。

```
[src/monitor/monitor.c,65,load_img] The image is /home/latesoon/ics2017/nexus-am/apps/microbench/build/microbench-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 15:01:34, Apr  6 2025
For help, type "help"
(nemu) c
[qsort] Quick sort: * Passed.
  min time: 542 ms [1018]
[queen] Queen placement: * Passed.
  min time: 820 ms [629]
[bf] Brainf**k interpreter: * Passed.
  min time: 4364 ms [600]
[fib] Fibonacci number: * Passed.
  min time: 9813 ms [291]
[sieve] Eratosthenes sieve: * Passed.
  min time: 7696 ms [551]
[15pz] A* 15-puzzle search: * Passed.
  min time: 1766 ms [327]
[dinic] Dinic's maxflow algorithm: * Passed.
  min time: 1446 ms [936]
[lzip] Lzip compression: * Passed.
  min time: 4066 ms [650]
[ssort] Suffix sort: * Passed.
  min time: 802 ms [737]
[md5] MD5 digest: * Passed.
  min time: 8170 ms [239]
================================================
MicroBench PASS        597 Marks
                 vs. 100000 Marks (i7-6700 @ 3.40GHz)
nemu: HIT GOOD TRAP at eip = 0x00100032
```

图 33: microbench-NEMU

```
make: [klib] Error 2 (ignored)
[qsort] Quick sort: * Passed.
  min time: 11 ms [50172]
[queen] Queen placement: * Passed.
  min time: 7 ms [73700]
[bf] Brainf**k interpreter: * Passed.
  min time: 24 ms [109204]
[fib] Fibonacci number: * Passed.
  min time: 34 ms [84044]
[sieve] Eratosthenes sieve: * Passed.
  min time: 39 ms [108733]
[15pz] A* 15-puzzle search: * Passed.
  min time: 7 ms [82742]
[dinic] Dinic's maxflow algorithm: * Passed.
  min time: 10 ms [135360]
[lzip] Lzip compression: * Passed.
  min time: 29 ms [91272]
[ssort] Suffix sort: * Passed.
  min time: 6 ms [98583]
[md5] MD5 digest: * Passed.
  min time: 20 ms [97965]
================================================
MicroBench PASS        93177 Marks
                 vs. 100000 Marks (i7-6700 @ 3.40GHz)
Exit (0)
```

图 34: microbench-native

	Dhrystone	Coremark	microbench
i7-6700 3.40 GHz	100000		
NEMU	149	549	597
native	42927	87619	93177

表 1: 性能测试对比 (Marks)

可以看到，相比于现代处理器 i7-6700（虽然也不是很现代）以及 native 环境，NEMU 的差距还是比较大的。

7. 实现 IOE(2)——键盘

阅读手册可以知道，i8042 初始化时会注册 0x60 处的端口作为数据寄存器, 注册 0x64 处的端口作为状态寄存器。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。

因此，当状态寄存器为 1 的时候，返回数据寄存器的读取结果，否则返回 _KEY_NONE 即可。

```
#define KEYBOARD_DATA 0x60
#define KEYBOARD_STATUS 0x64

int __read_key() {
  if(inb(KEYBOARD_STATUS))
    return inl(KEYBOARD_DATA);
  else
    return _KEY_NONE;
}
```

测试 keytest 程序，成功识别键盘操作。

```
[src/monitor/diff-test/diff-test.c,96,init_difftest] Connect to QEMU successfully
[src/monitor/monitor.c,65,load_img] The image is /home/latesoon/ics2017/nexus-am/tests/keytest/build/keytest-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 22:34:08, Apr  6 2025
For help, type "help"
(nemu) c
Get key: 73 UP down
Get key: 73 UP up
Get key: 74 DOWN down
Get key: 74 DOWN up
Get key: 75 LEFT down
Get key: 75 LEFT up
Get key: 76 RIGHT down
Get key: 76 RIGHT up
Get key: 57 X down
Get key: 57 X up
Get key: 45 D down
Get key: 45 D up
Get key: 35 U down
```

图 35: 键盘

8. 添加内存映射 I/O

这里需要对物理内存读写的函数判断是否是 I/O 空间，如果是的话则要访问内存映射 I/O，否则维持原逻辑。

修改 paddr_read 和 paddr_write，注意这里需要添加 mmio 的头文件，否则无法识别函数。

```
#include "device/mmio.h"

```

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

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

测试 videotest 项目，测试结果如图36所示，成功!

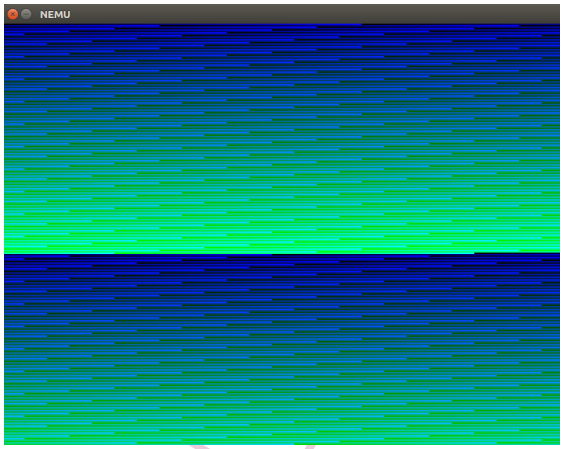

图 36: videotest 测试

9. 实现 IOE(3)——VGA

实现 _draw_rect 函数，以真正实现 VGA 功能。

由于 C 语言是行主存储的，可以按行进行 memcpy。

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

重新测试 videotest，成功!

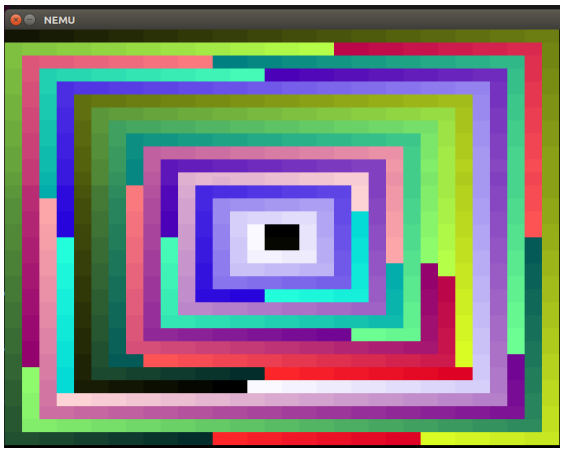

图 37: videotest 重新测试

10. 运行打字小游戏

在 typing 文件夹下运行打字小游戏，成功!

图 38: 运行打字小游戏

(二) 问答题

1. 理解 volatile 关键字

- Q:

  也许你从来都没听说过 C 语言中有 `volatile` 这个关键字，但它从 C 语言诞生开始就一直存在。`volatile` 关键字的作用十分特别，它的作用是避免编译器对相应代码进行优化。你应该动手体会一下 `volatile` 的作用，在 GNU/Linux 下编写以下代码:

```
1 void fun() {
2     volatile unsigned char *p = (void *)0x8049000;
3     *p = 0;
4     while(*p != 0xff);
5     *p = 0x33;
6     *p = 0x34;
7     *p = 0x86;
```

```
8 }
```

然后使用 -O2 编译代码，尝试去掉代码中的 volatile 关键字，重新使用 -O2 编译，并对比去掉 volatile 前后反汇编结果的不同。

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？

- A:

原函数编译结果如图39所示。

```
fun:
.LFB0:
        .cfi_startproc
        endbr64
        movb    $0, 134516736
        .p2align 4,,10
        .p2align 3
.L2:
        movzbl  134516736, %eax
        cmpb    $-1, %al
        jne     .L2
        movb    $51, 134516736
        movb    $52, 134516736
        movb    $-122, 134516736
        ret
        .cfi_endproc
```

图 39: 原函数编译结果

去掉 volatile 后，编译结果如图40所示。

```
fun:
.LFB0:
        .cfi_startproc
        endbr64
        movb    $0, 134516736
.L2:
        jmp     .L2
        .cfi_endproc
```

图 40: 去掉 volatile 编译结果

有 volatile 的情况下，编译器会严格按代码顺序生成指令，每次访问都会从内存中重新读取数据。而无 volatile 的时候则会直接使用旧值进行判断，以提升性能。

在涉及硬件操作、多线程或异步修改的场景中，不使用 volatile 可能会导致严重计算错误。

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 可能会导致严重的硬件交互错误。

2. 如何检测多个键被按下

- Q:

  在游戏中, 很多时候需要判断玩家是否同时按下了多个键, 例如 RPG 游戏中的八方向行走, 格斗游戏中的组合招式等等. 根据键盘码的特性, 你知道这些功能是如何实现的吗?

- A:

  通过建立一个键位状态的数组或者给每个键位分配一个位，在每次有键位被按下或者抬起的时候检索所有按键状态即可。

3. 神奇的调色板

- Q:

  现代的显示器一般都支持 24 位的颜色 (R,G,B 各占 8 个 bit, 共有 $2^8*2^8*2^8$ 约 1600 万种颜色), 为了让屏幕显示不同的颜色成为可能, 在 8 位颜色深度时会使用调色板的概念. 调色板是一个颜色信息的数组, 每一个元素占 4 个字节, 分别代表 R(red),G(green),B(blue),A(alpha) 的值. 引入了调色板的概念之后, 一个像素存储的就不再是颜色的信息, 而是一个调色板的索引: 具体来说, 要得到一个像素的颜色信息, 就要把它的值当作下标, 在调色板这个数组中做下标运算, 取出相应的颜色信息. 因此, 只要使用不同的调色板, 就可以在不同的时刻使用不同的 256 种颜色了.

  在一些 90 年代的游戏中, 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?

- A:

  调色板的结构是一个长度为 256 的数组,每个元素是 4 字节的 RGBA 颜色值(如 palette[256] = R,G,B,A, ...)。屏幕上每个像素存储的是调色板的索引值（0 255），而非实际颜色。

  每帧将所有调色板中的颜色值（R/G/B）逐渐减小（向黑色过渡）或增大（向白色过渡），以实现渐入渐出效果。

六、 必答题

(一) Question 1: 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. 为什么会发生这些错误? 你有办法证明你的想法吗?

1. static

去掉 static，有两种情况。
第一种情况：去掉 rtl_push 中的 static，但是因为里面调用了其他 static 函数导致报错。

```
latesoon@latesoon-virtual-machine:~/ics2017/nemu$ make run
+ CC src/cpu/exec/arith.c
In file included from ./include/cpu/decode.h:6:0,
                 from ./include/cpu/exec.h:9,
                 from src/cpu/exec/arith.c:1:
./include/cpu/rtl.h:150:3: error: ‘rtl_sm’ is static but used in inline function
 ‘rtl_push’ which is not static [-Werror]
   rtl_sm(&cpu.esp, 4, src1);
   ^
./include/cpu/rtl.h:149:3: error: ‘rtl_subi’ is static but used in inline functi
on ‘rtl_push’ which is not static [-Werror]
   rtl_subi(&cpu.esp, &cpu.esp, 4);
   ^
cc1: all warnings being treated as errors
Makefile:23: recipe for target 'build/obj/cpu/exec/arith.o' failed
make: *** [build/obj/cpu/exec/arith.o] Error 1
```

图 41: 去掉 static

第二种情况：去掉一条不调用其他函数的 rtl 指令，如 rtl_not，不报错。

static 的作用是限制函数的作用域为当前文件，避免多文件包含时引发重复定义错误，因为这个设定，非 static 函数不能调用 static 函数。

2. inline

inline 的作用是建议编译器将函数内联展开（直接插入调用处），避免函数调用的开销。去掉后函数不再内联，每次调用都会生成实际的函数调用（跳转指令）。

去掉 inline 后，触发函数未使用的报错。

```
latesoon@latesoon-virtual-machine:~/ics2017/nemu$ make run
+ CC src/cpu/exec/arith.c
+ CC src/cpu/exec/logic.c
+ CC src/cpu/exec/data-mov.c
In file included from ./include/cpu/decode.h:6:0,
                 from ./include/cpu/exec.h:9,
                 from src/cpu/exec/data-mov.c:1:
./include/cpu/rtl.h:133:13: error: ‘rtl_not’ defined but not used [-Werror=unuse
d-function]
 static void rtl_not(rtlreg_t* dest) {
             ^
cc1: all warnings being treated as errors
Makefile:23: recipe for target 'build/obj/cpu/exec/data-mov.o' failed
make: *** [build/obj/cpu/exec/data-mov.o] Error 1
```

图 42: 去掉 inline

原因是 inline 不会为函数生成独立函数调用的代码块，去掉后会生成，然而因为 static 限定，由于在该文件中没有被调用，则会报错未使用警告。

3. 全部去掉

把 static 和 inline 全部去掉后，函数有了独立的代码块，且没有作用域的限制，在多文件包含的时候会引发重复定义的冲突，如图43所示。

```
/home/latesoon/ics2017/nemu/./include/cpu/rtl.h:133: `rtl_not'被多次定义
build/obj/cpu/exec/arith.o:/home/latesoon/ics2017/nemu/./include/cpu/rtl.h:133:
第一次在此定义
build/obj/cpu/exec/system.o: 在函数‘rtl_not’中:
/home/latesoon/ics2017/nemu/./include/cpu/rtl.h:133: `rtl_not'被多次定义
build/obj/cpu/exec/arith.o:/home/latesoon/ics2017/nemu/./include/cpu/rtl.h:133:
第一次在此定义
build/obj/cpu/intr.o: 在函数‘rtl_not’中:
/home/latesoon/ics2017/nemu/./include/cpu/rtl.h:133: `rtl_not'被多次定义
build/obj/cpu/exec/arith.o:/home/latesoon/ics2017/nemu/./include/cpu/rtl.h:133:
第一次在此定义
```

图 43: 全部去掉

(二) Question 2: 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.

主要步骤是:

- 预处理: 对每个.c 文件展开宏和头文件（#include），生成.i 文件（隐式完成）。
- 编译: 将预处理后的代码编译为汇编（.s），再转为目标文件（.o）。
- 所有.o 文件被合并为最终可执行文件.

每次敲入 make 后，可以看到在目录下多出了一个 build 文件夹。

在编译阶段，递归查找 src 中所有的.c 文件，并且在 build 目录下设定其转换为.o 文件的规则，以便链接。

```
SRCS = $(shell find src/ -name "*.c")
OBJS = $(SRCS:src/%.c=$(OBJ_DIR)/%.o)
```

在链接阶段，链接所有.o 文件，以及外部库，生成可执行文件。

```
$(BINARY): $(OBJS)
    @$(LD) -O2 -o $@ $^ -lSDL2 -lreadline
```

(三) Question 3: 编译与链接

1. **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?**

有 29 个。通过以下指令检查生成的二进制文件来获得。

```
nm build/nemu | grep dummy
```

2. **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.**

还是 29 个，因为 common.h 中 include 了 debug.h，且没有任何一个源文件直接 include 了 debug.h。

3. **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.)**

出现重定义错误。因为之前没有对 dummy 赋值，可以视为是对其的声明。而如今两个地方都对其赋值，则会出现定义冲突。

```
latesoon@latesoon-virtual-machine:~/ics2017/nemu$ make run
+ CC src/device/device.c
In file included from src/device/device.c:1:0:
./include/common.h:31:21: error: redefinition of ‘dummy’
 volatile static int dummy = 0;
                     ^
In file included from ./include/common.h:10:0,
                 from src/device/device.c:1:
./include/debug.h:47:21: note: previous definition of ‘dummy’ was here
 volatile static int dummy=0;
                     ^
Makefile:23: recipe for target 'build/obj/device/device.o' failed
make: *** [build/obj/device/device.o] Error 1
```

图 44: 重定义错误

七、 所遇 BUG 及解决思路

在第二阶段新指令实现过程中，遇到了较长时间的 HIT BAD TRAP 的修复过程。

对此问题的解决上，首先先整体回顾检查了一遍代码，部分 bug 在此阶段得到修复，也因此通过了更多样例。在此阶段，我还怀疑过是否由于部分 rtl 指令未能实现而在测试程序中调用导致错误而额外实现了一些 rtl 指令（rtl_eq0/rtl_eqi/rtl_neq0，部分指令在阶段 3 使用，但并没有在阶段 2 使用），不过现在看来大概率不是因为这个。

例如，在 setcc 中，一开始因为忽视了部分 i386 手册中的问题，导致部分计算错误。以及在 rtl_update_ZF 中，将 0xFFFFFFFF 错误写成 0xFFFF 等问题。

然而，完成修复后，仍然有一部分样例未能通过。最终我从中选择了一个样例（load-store.c）进行 nemu 逐行调试与分析，在汇编代码阅读和第一次调试后，我理解了汇编代码的整体逻辑，进而确定的发生错误执行的位置。于是在第二次调试中，我增加了一些 printf 打印信息，并且通过计算先前的执行记录较快的跳转到发生问题代码的位置进行调试。

在 cmp 的执行函数中添加了以下输出信息。

```
1 //printf("%x %x %x",t2,id_dest->val,id_src->val);
```

```
latesoon@latesoon-virtual-machine:~/ics2017/nexus-am/tests/cputest$ make ALL=load-store run
Building load-store [x86-nemu]
Building am [x86-nemu]
make[2]: *** 没有指明目标并且找不到 makefile。 停止。
+ CC src/cpu/exec/arith.c
+ LD build/nemu
[src/monitor/monitor.c,65,load_img] The image is /home/latesoon/ics2017/nexus-am/tests/cputest/build/load-store-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 16:04:40, Apr  1 2025
For help, type "help"
(nemu) si 90
0 0 0fffffff2 2 100 258 258fffffff4 4 100 4abc 4abcfffffff6 6 100 7fff 7fffffffff8 8 10(nemu) info r
Registers info:
EAX:0x00000001 ECX:0x00007BF0
EDX:0x1D490241 EBX:0x00000008
ESP:0x00007BC0 EBP:0x00007BD8
ESI:0x12154784 EDI:0x56E3C720
EIP:0x00100060
EFLAGS:
CF:1 ZF:0 SF:1 IF:0 OF:0
(nemu) si
  100060:   83 ec 0c                            subl $0xc,%esp
(nemu) si
  100063:   0f bf 83 c0 01 10 00                movsxl 0x1001c0(%ebx),%ax
(nemu) info r
Registers info:
EAX:0x00008000 ECX:0x00007BF0
EDX:0x1D490241 EBX:0x00000008
ESP:0x00007BB4 EBP:0x00007BD8
ESI:0x12154784 EDI:0x56E3C720
EIP:0x0010006A
EFLAGS:
CF:0 ZF:0 SF:0 IF:0 OF:0
(nemu) si
10000 8000 ffff8000  10006a:   3b 84 1b a0 01 10 00                cmpl 0x1001a0(%ebx,%ebx,1),%eax
(nemu) x 30 0x1001a0
Address     Data
0x001001A0: 00 00 00 00 58 02 00 00
0x001001A8: BC 4A 00 00 FF 7F 00 00
0x001001B0: 00 80 FF FF 00 81 FF FF
0x001001B8: CD AB FF FF FF FF
(nemu) q
 load-store
```

图 45: 第二次调试

最终，根据分析，是在 movsx 中，rtl_sext 因为没有进行数据类型转换，进行的按位左移右移运算都是无符号运算，未能进行符号扩展导致错误。

最终，对其添加 int32_t 类型转换后，成功解决了该问题。

```
1 static inline void rtl_sext(rtlreg_t* dest, const rtlreg_t* src1, int width)
    {
2   *dest = ((int32_t)( *src1 << (32 - width * 8) )) >> ( 32 - width * 8);
3 }
```