



南開大學

Nankai University

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

姓名：李颖

学号：2110939

专业：计算机科学与技术

2024 年 4 月 12 日

# 目录

<b>1 实验介绍</b>	<b>3</b>
1.1 实验目的	3
1.2 实验内容	3
<b>2 阶段一</b>	<b>3</b>
2.1 实验流程	3
2.2 eflags 与 RTL 语言	5
2.2.1 实现 eflags 寄存器	5
2.2.2 RTL 指令实现	6
2.3 指令的实现	8
2.3.1 call 指令	8
2.3.2 push 指令	9
2.3.3 sub 指令与 RTL 指令	9
2.3.4 xor 指令	10
2.3.5 pop 指令	11
2.3.6 ret 指令	12
2.4 结果验证	12
<b>3 Differential Testing</b>	<b>12</b>
<b>4 阶段二</b>	<b>13</b>
4.1 mov 指令	13
4.2 leave 指令	14
4.3 lea 指令	14
4.4 group1	14
4.4.1 add 指令	15
4.4.2 or 指令	15
4.4.3 adc 指令	16
4.4.4 sbb 指令	16
4.4.5 and 指令	16
4.4.6 cmp 指令	16
4.5 group5	17
4.5.1 inc 指令	17
4.5.2 dec 指令	17
4.6 test 指令	18
4.7 Jcc 指令	18
4.8 group2	20
4.8.1 rol 指令	20
4.8.2 shl 指令	20
4.8.3 shr 指令	21
4.8.4 sar 指令	21

4.9	grp3	21
4.9.1	not 指令	21
4.9.2	neg 指令	22
4.10	movzx/movsx	22
4.11	cld/cwtl 指令	22
4.12	nop 指令	23
4.13	实验结果	23
<b>5</b>	<b>阶段三</b>	<b>24</b>
5.1	理解 volatile 关键字	24
5.2	串口	25
5.3	时钟	26
5.3.1	实现 IOE	26
5.3.2	看看 NEMU 跑多快	26
5.4	键盘	27
5.4.1	如何检测多个键同时被按下	27
5.4.2	实现 IOE(2)	27
5.5	VGA	28
5.5.1	神奇的调色板	28
5.5.2	添加内存映射 I/O	28
5.5.3	实现 IOE(3)	29
5.5.4	运行打字小游戏	29
5.5.5	运行红白机	30
<b>6</b>	<b>必答题</b>	<b>30</b>
6.1	Q1: static 与 inline	30
6.2	Q2: 了解 makefile	31
<b>7</b>	<b>Bug 汇总</b>	<b>31</b>
7.1	移动文件夹导致环境中路径不一致	31
7.2	2-byte opcode	32
7.3	klib 文件夹找不到目标文件	32

# 1 实验介绍

## 1.1 实验目的

补充实现指令，使 NEMU 能成功在 TRM 上运行大部分程序。

## 1.2 实验内容

实现新指令，具体内容如下：

1. 在 `opcode_table` 中填写正确的译码函数，执行函数以及操作数宽度
2. 完善 `eflags` 寄存器及 RTL 相关函数
3. 用 RTL 实现正确的执行函数

阶段一需实现的指令：call, push, sub, xor, pop, ret 指令。

阶段二需实现的指令：

- ❖ 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: `leq`, `nop`

框架代码已经实现了上述红色标记的指令，但并没有填写 `opcode_table`。此外，某些需要更新 EFLAGS 的指令并没有完全实现好(框架代码中已经插入了 `TODO()` 作为提示)，你还需要编写相应的功能。

# 2 阶段一

在 PA1 中，我们实现了简易的 TRM，能够从 EIP 指示的存储器位置取出指令并执行指令，然后更新 EIP。在本阶段，我们要实现的就是取指-译码-执行的指令周期。

## 2.1 实验流程

实验流程如下所示，以遇到的第一条无效指令为例。

1. 在 NEMU 中运行 dummy 程序，遇到无效操作码导致程序暂停，操作码为 `e8`。

```

Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 01:13:01, Mar 28 2024
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
00000000: 00000000 00000000 00000000 00000000
00000004: 00000000 00000000 00000000 00000000
00000008: 00000000 00000000 00000000 00000000
0000000c: 00000000 00000000 00000000 00000000
00000010: 00000000 00000000 00000000 00000000
00000014: 00000000 00000000 00000000 00000000
00000018: 00000000 00000000 00000000 00000000
0000001c: 00000000 00000000 00000000 00000000
00000020: 00000000 00000000 00000000 00000000
00000024: 00000000 00000000 00000000 00000000
00000028: 00000000 00000000 00000000 00000000
0000002c: 00000000 00000000 00000000 00000000
00000030: 00000000 00000000 00000000 00000000
00000034: 00000000 00000000 00000000 00000000
00000038: 00000000 00000000 00000000 00000000
0000003c: 00000000 00000000 00000000 00000000
00000040: 00000000 00000000 00000000 00000000
00000044: 00000000 00000000 00000000 00000000
00000048: 00000000 00000000 00000000 00000000
0000004c: 00000000 00000000 00000000 00000000
00000050: 00000000 00000000 00000000 00000000
00000054: 00000000 00000000 00000000 00000000
00000058: 00000000 00000000 00000000 00000000
0000005c: 00000000 00000000 00000000 00000000
00000060: 00000000 00000000 00000000 00000000
00000064: 00000000 00000000 00000000 00000000
00000068: 00000000 00000000 00000000 00000000
0000006c: 00000000 00000000 00000000 00000000
00000070: 00000000 00000000 00000000 00000000
00000074: 00000000 00000000 00000000 00000000
00000078: 00000000 00000000 00000000 00000000
0000007c: 00000000 00000000 00000000 00000000
00000080: 00000000 00000000 00000000 00000000
00000084: 00000000 00000000 00000000 00000000
00000088: 00000000 00000000 00000000 00000000
0000008c: 00000000 00000000 00000000 00000000
00000090: 00000000 00000000 00000000 00000000
00000094: 00000000 00000000 00000000 00000000
00000098: 00000000 00000000 00000000 00000000
0000009c: 00000000 00000000 00000000 00000000
000000a0: 00000000 00000000 00000000 00000000
000000a4: 00000000 00000000 00000000 00000000
000000a8: 00000000 00000000 00000000 00000000
000000ac: 00000000 00000000 00000000 00000000
000000b0: 00000000 00000000 00000000 00000000
000000b4: 00000000 00000000 00000000 00000000
000000b8: 00000000 00000000 00000000 00000000
000000bc: 00000000 00000000 00000000 00000000
000000c0: 00000000 00000000 00000000 00000000
000000c4: 00000000 00000000 00000000 00000000
000000c8: 00000000 00000000 00000000 00000000
000000cc: 00000000 00000000 00000000 00000000
000000d0: 00000000 00000000 00000000 00000000
000000d4: 00000000 00000000 00000000 00000000
000000d8: 00000000 00000000 00000000 00000000
000000dc: 00000000 00000000 00000000 00000000
000000e0: 00000000 00000000 00000000 00000000
000000e4: 00000000 00000000 00000000 00000000
000000e8: 00000000 00000000 00000000 00000000
000000ec: 00000000 00000000 00000000 00000000
000000f0: 00000000 00000000 00000000 00000000
000000f4: 00000000 00000000 00000000 00000000
000000f8: 00000000 00000000 00000000 00000000
000000fc: 00000000 00000000 00000000 00000000
00000100: 00000000 00000000 00000000 00000000
00000104: 00000000 00000000 00000000 00000000
00000108: 00000000 00000000 00000000 00000000
0000010c: 00000000 00000000 00000000 00000000
00000110: 00000000 00000000 00000000 00000000
00000114: 00000000 00000000 00000000 00000000
00000118: 00000000 00000000 00000000 00000000
0000011c: 00000000 00000000 00000000 00000000
00000120: 00000000 00000000 00000000 00000000
00000124: 00000000 00000000 00000000 00000000
00000128: 00000000 00000000 00000000 00000000
0000012c: 00000000 00000000 00000000 00000000
00000130: 00000000 00000000 00000000 00000000
00000134: 00000000 00000000 00000000 00000000
00000138: 00000000 00000000 00000000 00000000
0000013c: 00000000 00000000 00000000 00000000
00000140: 00000000 00000000 00000000 00000000
00000144: 00000000 00000000 00000000 00000000
00000148: 00000000 00000000 00000000 00000000
0000014c: 00000000 00000000 00000000 00000000
00000150: 00000000 00000000 00000000 00000000
00000154: 00000000 00000000 00000000 00000000
00000158: 00000000 00000000 00000000 00000000
0000015c: 00000000 00000000 00000000 00000000
00000160: 00000000 00000000 00000000 00000000
00000164: 00000000 00000000 00000000 00000000
00000168: 00000000 00000000 00000000 00000000
0000016c: 00000000 00000000 00000000 00000000
00000170: 00000000 00000000 00000000 00000000
00000174: 00000000 00000000 00000000 00000000
00000178: 00000000 00000000 00000000 00000000
0000017c: 00000000 00000000 00000000 00000000
00000180: 00000000 00000000 00000000 00000000
00000184: 00000000 00000000 00000000 00000000
00000188: 00000000 00000000 00000000 00000000
0000018c: 00000000 00000000 00000000 00000000
00000190: 00000000 00000000 00000000 00000000
00000194: 00000000 00000000 00000000 00000000
00000198: 00000000 00000000 00000000 00000000
0000019c: 00000000 00000000 00000000 00000000
000001a0: 00000000 00000000 00000000 00000000
000001a4: 00000000 00000000 00000000 00000000
000001a8: 00000000 00000000 00000000 00000000
000001ac: 00000000 00000000 00000000 00000000
000001b0: 00000000 00000000 00000000 00000000
000001b4: 00000000 00000000 00000000 00000000
000001b8: 00000000 00000000 00000000 00000000
000001bc: 00000000 00000000 00000000 00000000
000001c0: 00000000 00000000 00000000 00000000
000001c4: 00000000 00000000 00000000 00000000
000001c8: 00000000 00000000 00000000 00000000
000001cc: 00000000 00000000 00000000 00000000
000001d0: 00000000 00000000 00000000 00000000
000001d4: 00000000 00000000 00000000 00000000
000001d8: 00000000 00000000 00000000 00000000
000001dc: 00000000 00000000 00000000 00000000
000001e0: 00000000 00000000 00000000 00000000
000001e4: 00000000 00000000 00000000 00000000
000001e8: 00000000 00000000 00000000 00000000
000001ec: 00000000 00000000 00000000 00000000
000001f0: 00000000 00000000 00000000 00000000
000001f4: 00000000 00000000 00000000 00000000
000001f8: 00000000 00000000 00000000 00000000
000001fc: 00000000 00000000 00000000 00000000
00000200: 00000000 00000000 00000000 00000000
00000204: 00000000 00000000 00000000 00000000
00000208: 00000000 00000000 00000000 00000000
0000020c: 00000000 00000000 00000000 00000000
00000210: 00000000 00000000 00000000 00000000
00000214: 00000000 00000000 00000000 00000000
00000218: 00000000 00000000 00000000 00000000
0000021c: 00000000 00000000 00000000 00000000
00000220: 00000000 00000000 00000000 00000000
00000224: 00000000 00000000 00000000 00000000
00000228: 00000000 00000000 00000000 00000000
0000022c: 00000000 00000000 00000000 00000000
00000230: 00000000 00000000 00000000 00000000
00000234: 00000000 00000000 00000000 00000000
00000238: 00000000 00000000 00000000 00000000
0000023c: 00000000 00000000 00000000 00000000
00000240: 00000000 00000000 00000000 00000000
00000244: 00000000 00000000 00000000 00000000
00000248: 00000000 00000000 00000000 00000000
0000024c: 00000000 00000000 00000000 00000000
00000250: 00000000 00000000 00000000 00000000
00000254: 00000000 00000000 00000000 00000000
00000258: 00000000 00000000 00000000 00000000
0000025c: 00000000 00000000 00000000 00000000
00000260: 00000000 00000000 00000000 00000000
00000264: 00000000 00000000 00000000 00000000
00000268: 00000000 00000000 00000000 00000000
0000026c: 00000000 00000000 00000000 00000000
00000270: 00000000 00000000 00000000 00000000
00000274: 00000000 00000000 00000000 00000000
00000278: 00000000 00000000 00000000 00000000
0000027c: 00000000 00000000 00000000 00000000
00000280: 00000000 00000000 00000000 00000000
00000284: 00000000 00000000 00000000 00000000
00000288: 00000000 00000000 00000000 00000000
0000028c: 00000000 00000000 00000000 00000000
00000290: 00000000 00000000 00000000 00000000
00000294: 00000000 00000000 00000000 00000000
00000298: 00000000 00000000 00000000 00000000
0000029c: 00000000 00000000 00000000 00000000
000002a0: 00000000 00000000 00000000 00000000
000002a4: 00000000 00000000 00000000 00000000
000002a8: 00000000 00000000 00000000 00000000
000002ac: 00000000 00000000 00000000 00000000
000002b0: 00000000 00000000 00000000 00000000
000002b4: 00000000 00000000 00000000 00000000
000002b8: 00000000 00000000 00000000 00000000
000002bc: 00000000 00000000 00000000 00000000
000002c0: 00000000 00000000 00000000 00000000
000002c4: 00000000 00000000 00000000 00000000
000002c8: 00000000 00000000 00000000 00000000
000002cc: 00000000 00000000 00000000 00000000
000002d0: 00000000 00000000 00000000 00000000
000002d4: 00000000 00000000 00000000 00000000
000002d8: 00000000 00000000 00000000 00000000
000002dc: 00000000 00000000 00000000 00000000
000002e0: 00000000 00000000 00000000 00000000
000002e4: 00000000 00000000 00000000 00000000
000002e8: 00000000 00000000 00000000 00000000
000002ec: 00000000 00000000 00000000 00000000
000002f0: 00000000 00000000 00000000 00000000
000002f4: 00000000 00000000 00000000 00000000
000002f8: 00000000 00000000 00000000 00000000
000002fc: 00000000 00000000 00000000 00000000
00000300: 00000000 00000000 00000000 00000000
00000304: 00000000 00000000 00000000 00000000
00000308: 00000000 00000000 00000000 00000000
0000030c: 00000000 00000000 00000000 00000000
00000310: 00000000 00000000 00000000 00000000
00000314: 00000000 00000000 00000000 00000000
00000318: 00000000 00000000 00000000 00000000
0000031c: 00000000 00000000 00000000 00000000
00000320: 00000000 00000000 00000000 00000000
00000324: 00000000 00000000 00000000 00000000
00000328: 00000000 00000000 00000000 00000000
0000032c: 00000000 00000000 00000000 00000000
00000330: 00000000 00000000 00000000 00000000
00000334: 00000000 00000000 00000000 00000000
00000338: 00000000 00000000 00000000 00000000
0000033c: 00000000 00000000 00000000 00000000
00000340: 00000000 00000000 00000000 00000000
00000344: 00000000 00000000 00000000 00000000
00000348: 00000000 00000000 00000000 00000000
0000034c: 00000000 00000000 00000000 00000000
00000350: 00000000 00000000 00000000 00000000
00000354: 00000000 00000000 00000000 00000000
00000358: 00000000 00000000 00000000 00000000
0000035c: 00000000 00000000 00000000 00000000
00000360: 00000000 00000000 00000000 00000000
00000364: 00000000 00000000 00000000 00000000
00000368: 00000000 00000000 00000000 00000000
0000036c: 00000000 00000000 00000000 00000000
00000370: 00000000 00000000 00000000 00000000
00000374: 00000000 00000000 00000000 00000000
00000378: 00000000 00000000 00000000 00000000
0000037c: 00000000 00000000 00000000 00000000
00000380: 00000000 00000000 00000000 00000000
00000384: 00000000 00000000 00000000 00000000
00000388: 00000000 00000000 00000000 00000000
0000038c: 00000000 00000000 00000000 00000000
00000390: 00000000 00000000 00000000 00000000
00000394: 00000000 00000000 00000000 00000000
00000398: 00000000 00000000 00000000 00000000
0000039c: 00000000 00000000 00000000 00000000
000003a0: 00000000 00000000 00000000 00000000
000003a4: 00000000 00000000 00000000 00000000
000003a8: 00000000 00000000 00000000 00000000
000003ac: 00000000 00000000 00000000 00000000
000003b0: 00000000 00000000 00000000 00000000
000003b4: 0
```

	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		XCHG		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	POFF	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	JCKE	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	HLT	CMC	Unary Grp3			CLC	STC	CLI	STI	CLD	STD	INC/DEC	Indirect									
				REPE			Eb	Ev						Grp4	Grp5											

3. 查看 call 指令的编号范围, 将正确的译码函数、执行函数、操作数宽度填写到 exec.c/opcode\_table 中。其中, call 指令的译码函数为 J, 执行函数为 call。

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

4. 补充执行函数定义, 在 exec/all-instr.h 中补充 make\_EHelper() 宏定义。

```
1 //control.c
2 make_EHelper(call);
3 make_EHelper(jmp);
```

5. 在 decode.c 中实现译码函数, 将译码结果保存在 decinfo 中。

```
1 make_DHelper(J) {
2     decode_op_SI(eip, id_dest, false);
3     // the target address can be computed in the decode stage
4     decoding.jump_eip = id_dest->siml + *eip;
5 }
```

6. 在 control.c、data-mov.c、logic.c、arith.c、或 system.c 中补充执行函数。其中, call 指令的执行函数在 control.c 中补充。

```
1 make_EHelper(call) {
2     // the target address is calculated at the decode stage
3     rtl_li(&t2, decoding.seq_eip);
4     rtl_push(&t2);
5
6     decoding.is_jmp = 1;
```

```

7
8   print_asm("call %x", decoding.jump_eip);
9 }
10
11 make_EHelper(jmp) {
12     // the target address is calculated at the decode stage
13     decoding.is_jmp = 1;
14
15     print_asm("jmp %x", decoding.jump_eip);
16 }

```

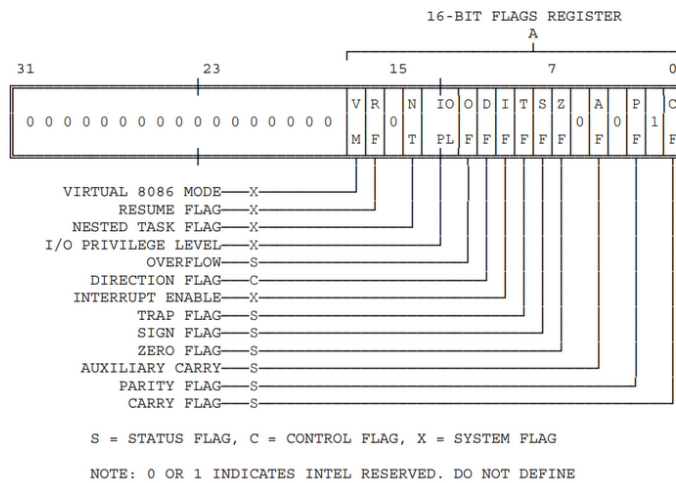
7. 测试查看正确性，如下图所示。若显示在下一条新指令处停止，则说明上一条指令已成功实现。回到第一步，重新进行这几个步骤，直至完成阶段一的几个指令。

## 2.2 eflags 与 RTL 语言

### 2.2.1 实现 eflags 寄存器

EFLAGS 是 x86 架构中的一个寄存器，用于存储处理器状态标志。EFLAGS 寄存器的每一位都代表着不同的含义。查阅 i386 手册，得到 EFLAGS 的每一位的用途如下所示。

Figure 2-8. EFLAGS Register



使用 struct 结构，在 reg.h 中实现 eflags 寄存器的定义 (从低位开始定义)，结构定义如下所示。

```

1 struct bs {
2     uint32_t CF : 1; //carry flag
3     uint32_t   : 1;
4     uint32_t   : 4; //party & auxiliary carry
5     uint32_t ZF : 1; //zero flag
6     uint32_t SF : 1; //sign flag
7     uint32_t TF : 1; //trap flag
8     uint32_t IF : 1; //inerrupt enable
9     uint32_t DF : 1; //direction flag
10    uint32_t OF : 1; //overflow flag
11    uint32_t   : 20;

```

```
12 }eflags;
```

eflags 的标志在 monitor.c 中进行初始化

```
1 static inline void restart() {
2     /* Set the initial instruction pointer. */
3     cpu.eip = ENTRY_START;
4
5     unsigned int origin = 2;
6     memcpy(&cpu.eflags, &origin, sizeof(cpu.eflags));
7
8     #ifdef DIFF_TEST
9         init_qemu_reg();
10    #endif
11 }
```

### 2.2.2 RTL 指令实现

在实现 RTL 指令之前，首先在 arith.c 中编写辅助函数，用于修改 eflags 中的各个标志位，函数中的每个操作都会根据运算结果来更新相应的标志位。

```
1 static inline void eflags_modify() {
2     rtl_sub(&t2, &id_dest -> val, &id_src -> val);
3     rtl_update_ZFSF(&t2, id_dest -> width);
4     rtl_sltu(&t0, &id_dest -> val, &id_src -> val);
5     rtl_set_CF(&t0);
6     rtl_xor(&t0, &id_dest -> val, &id_src -> val);
7     rtl_xor(&t1, &id_dest -> val, &t2);
8     rtl_and(&t0, &t0, &t1);
9     rtl_msb(&t0, &t0, id_dest -> width);
10    rtl_set_OF(&t0);
11 }
```

之后，需要实现用于设置和获取处理器标志寄存器 (cpu.eflags) 中的特定标志位 (f) 的宏，该宏包括两个函数。实现后如下所示。

```
1 #define make_rtl_setget_eflags(f) \
2     static inline void concat(rtl_set_, f) (const rtlreg_t* src) { \
3         cpu.eflags.f=*src;\
4     } \
5     static inline void concat(rtl_get_, f) (rtlreg_t* dest) { \
6         *dest=cpu.eflags.f;\
7     }
```

此外，还需要补充 eflags 寄存器的标志位更新函数，如下所示。

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

```

4 }
5
6 static inline void rtl_eqi(rtlreg_t* dest, const rtlreg_t* src1, int imm) {
7     // dest <- (src1 == imm ? 1 : 0)
8     rtl_xori(dest, src1, imm);
9     rtl_eq0(dest, dest);
10 }
11
12 static inline void rtl_neq0(rtlreg_t* dest, const rtlreg_t* src1) {
13     // dest <- (src1 != 0 ? 1 : 0)
14     rtl_eq0(dest, src1);
15     rtl_eq0(dest, dest);
16 }
17
18 static inline void rtl_msb(rtlreg_t* dest, const rtlreg_t* src1, int width) {
19     // dest <- src1[width * 8 - 1]
20     rtl_shri(dest, src1, width*8-1);
21     rtl_andi(dest, dest, 0x1);
22 }
23
24 static inline void rtl_update_ZF(const rtlreg_t* result, int width) {
25     // eflags.ZF <- is_zero(result[width * 8 - 1 .. 0])
26     rtl_andi(&t0, result, (0xffffffffu >> (4-width)*8));
27     rtl_eq0(&t0, &t0);
28     rtl_set_ZF(&t0);
29 }
30
31 static inline void rtl_update_SF(const rtlreg_t* result, int width) {
32     // eflags.SF <- is_sign(result[width * 8 - 1 .. 0])
33     assert(result != &t0);
34     rtl_msb(&t0, result, width);
35     rtl_set_SF(&t0);
36 }
37
38 static inline void rtl_update_ZFSF(const rtlreg_t* result, int width) {
39     rtl_update_ZF(result, width);
40     rtl_update_SF(result, width);
41 }

```

实现 mov 指令，相当于加上 0:

```

1 static inline void rtl_mv(rtlreg_t* dest, const rtlreg_t *src1) {
2     // dest <- src1
3     rtl_addi(dest, src1, 0);
4 }

```

not 运算:

```

1 static inline void rtl_not(rtlreg_t* dest) {

```



```

2 // dest <- ~dest
3 rtl_xori(dest, dest, 0xffffffff);
4 }

```

符号扩展:

```

1 static inline void rtl_sext(rtlreg_t* dest, const rtlreg_t* src1, int width) {
2 // dest <- signext(src1[(width * 8 - 1) .. 0])
3 if(width == 0) {
4 rtl_mv(dest, src1);
5 }
6 else {
7 rtl_shli(dest, src1, (4 - width) * 8);
8 rtl_sari(dest, dest, (4 - width) * 8);
9 }
10 }

```

其余 RTL 指令的补充会分别在具体指令模块中讲述。

## 2.3 指令的实现

在阶段一，需要实现的指令为 call, push, sub, xor, pop, ret 指令。

### 2.3.1 call 指令

call 指令的发现与完善在【阶段一/实验流程】中已经说明。

call 指令为 J 型指令，操作数仅一个立即数。CPU 的跳转目标地址 = 当前 eip + 立即数 offset。因此，在实现 make\_DHelper(J) 函数时，需要调用 decode\_op\_SI() 函数来实现立即数的读取。该函数的具体实现如下所示。

```

1 static inline make_DopHelper(SI) {
2 assert(op->width == 1 || op->width == 4);
3
4 op->type = OP_TYPE_MM;
5 op->siml = instr_fetch(eip, op->width);
6 if(op->width == 1) {
7 op->siml = (int8_t)op->siml;
8 }
9
10 rtl_li(&op->val, op->siml);
11
12 #ifdef DEBUG
13 snprintf(op->str, OP_STR_SIZE, "$0x%x", op->siml);
14 #endif
15 }

```

之后需要编写的函数如实验流程中所示。

### 2.3.2 push 指令

继续运行程序，输入 c，发现在操作码 55 处停止。查表可知，该指令为 push 指令。

```
[src/monitor/monitor.c,30,welcome] Build time: 01:13:01, Mar 28 2024
For help, type "help"
(nemu) c
invalid opcode(eip = 0x00100010): 55 89 e5 83 ec 08 e8 05 ...
```

在 rtl.h 中补充关于 push 和 pop 的 rtl 函数，如下所示。

```
1 static inline void rtl_push(const rtlreg_t* src1) {
2     // esp <- esp - 4
3     // M[esp] <- src1
4     rtl_subi(&cpu.esp, &cpu.esp, 4);
5     rtl_sm(&cpu.esp, 4, src1);
6 }
7
8 static inline void rtl_pop(rtlreg_t* dest) {
9     // dest <- M[esp]
10    // esp <- esp + 4
11    rtl_lm(dest, &cpu.esp, 4);
12    rtl_addi(&cpu.esp, &cpu.esp, 4);
13 }
```

在 opcode\_table 的对应位置中填写译码函数和执行函数。

```
1 /* 0x50 */ IDEX(r, push), IDEX(r, push), IDEX(r, push), IDEX(r, push),
2 /* 0x54 */ IDEX(r, push), IDEX(r, push), IDEX(r, push), IDEX(r, push),
```

在 all-intsr.h 中注册函数。

```
1 //data-mov.c
2 make_EHelper(push);
```

在 data-mov.c 中完善执行函数。

```
1 make_EHelper(push) {
2     rtl_push(&id_dest -> val);
3     print_asm_template1(push);
4 }
5
6 make_EHelper(pop) {
7     rtl_pop(&t2);
8     operand_write(id_dest, &t2);
9     print_asm_template1(pop);
10 }
```

### 2.3.3 sub 指令与 RTL 指令

继续运行程序，输入 c，发现在操作码 83 处停止。查表可知，该指令为 grp1。

```

Welcome to nemu.
[src/monitor/monitor.c,30,welcome] Build time: 01:13:01, Mar 28 2024
For help, type "help"
(nemu) c
invalid opcode(eip = 0x00100013): 83 ec 08 e8 05 00 00 00 ...

```

0x80、0x81 与 0x83 的 sub 需要进行 opcode 的扩展。查看 make\_group 函数，该函数用于处理扩展 opcode 的情况。以 gp1 为例，在 opcode\_table\_gp1 中存储各 opcode\_entry，并在运行 make\_EHelper(gp1) 时使用 idex(eip, opcode\_table\_gp1[decoding.ext\_opcode]) 作为进一步的译码-执行函数。

因为 sub 的 ext\_opcode=5，所以在 opcode\_table\_gp1[5] 处填写 EX(sub) 执行函数，如下所示。

```

1 make_group(gp1,
2     EMPTY, EMPTY, EMPTY, EMPTY,
3     EMPTY, EX(sub), EMPTY, EMPTY)

```

在 opcode\_table 的对应位置中填写译码函数和执行函数。

```

1 /* 0x28 */ IDExW(G2E, sub, 1), IDEx(G2E, sub), IDExW(E2G, sub, 1), IDEx(E2G, sub),
2 /* 0x2c */ IDExW(I2a, sub, 1), IDEx(I2a, sub), EMPTY, EMPTY,

```

在 all-intsr.h 中注册函数。

```

1 //arith.c
2 make_EHelper(sub);

```

在 arith.c 中完善执行函数。由于 sub 涉及计算，因此需要更新 eflags 的标志位。eflags 标志位更新函数已在【阶段一/eflags 与 RTL 语言】模块中实现。

```

1 make_EHelper(sub) {
2     eflags_modify();
3     operand_write(id_dest, &t2);
4     print_asm_template2(sub);
5 }

```

### 2.3.4 xor 指令

继续运行程序，输入 c，发现在操作码 31 处停止。查表可知，该指令为 xor 指令。

```

For help, type "help"
(nemu) c
invalid opcode(eip = 0x00100023): 31 c0 5d c3 00 00 00 00 ...

```

在 opcode\_table 的对应位置中填写译码函数和执行函数。

```

1 /* 0x30 */ IDExW(G2E, xor, 1), IDEx(G2E, xor), IDExW(E2G, xor, 1), IDEx(E2G, xor),
2 /* 0x34 */ IDExW(I2a, xor, 1), IDEx(I2a, xor), EMPTY, EMPTY,

```

在 all-intsr.h 中注册函数。

```

1 //logic.c
2 make_EHelper(xor);

```

在 logic.c 中完善执行函数。

```

1 make_EHelper(xor) {
2     rtl_xor(&t2, &id_dest -> val, &id_src -> val);
3     operand_write(id_dest, &t2);
4
5     rtl_update_ZFSF(&t2, id_dest -> width);
6
7     rtl_set_CF(&tzero);
8     rtl_set_OF(&tzero);
9
10    print_asm_template2(xor);
11 }

```

### 2.3.5 pop 指令

继续运行程序，输入 c，发现在操作码 5d 处停止。查表可知，该指令为 pop 指令。

```

(nemu) c
invalid opcode(eip = 0x00100025): 5d c3 00 00 00 00 00 00 ...

```

在 rtl.h 中补充关于 pop 的 rtl 函数，如下所示。

```

1 static inline void rtl_pop(rtlreg_t* dest) {
2     // dest <- M[esp]
3     // esp <- esp + 4
4     rtl_lm(dest, &cpu.esp, 4);
5     rtl_addi(&cpu.esp, &cpu.esp, 4);
6 }

```

在 opcode\_table 的对应位置中填写译码函数和执行函数。

```

1 /* 0x58 */ IDEX(r, pop), IDEX(r, pop), IDEX(r, pop), IDEX(r, pop),
2 /* 0x5c */ IDEX(r, pop), IDEX(r, pop), IDEX(r, pop), IDEX(r, pop),

```

在 all-intsr.h 中注册函数。

```

1 //data-mov.c
2 make_EHelper(pop);

```

在 data-mov.c 中完善执行函数。

```

1 make_EHelper(pop) {
2     rtl_pop(&t2);
3     operand_write(id_dest, &t2);
4     print_asm_template1(pop);
5 }

```

### 2.3.6 ret 指令

继续运行程序，输入 c，发现在操作码 c3 处停止。查表可知，该指令为 ret 指令。

```
(nemu) c
invalid opcode(eip = 0x00100026): c3 00 00 00 00 00 00 00 ...
```

在 opcode\_table 的对应位置中填写译码函数和执行函数。

```
1 /* 0xc0 */ IDEXW(gp2_l2E, gp2, 1), IDEX(gp2_l2E, gp2), EMPTY, EX(ret),
```

在 all-intsr.h 中注册函数。

```
1 make_EHelper(ret);
```

在 control.c 中完善执行函数。

```
1 make_EHelper(ret) {
2     rtl_pop(&t2);
3     decoding.jump_eip = t2;
4     decoding.is_jump = 1;
5     print_asm("ret");
6 }
```

## 2.4 结果验证

重新运行程序，发现程序在 good trap 处停止，说明阶段一所需补充的部分已经正确实现。

```
[src/monitor/monitor.c,30,welcome] Build time: 02:54:02, Mar 28 2024
For help, type "help"
(nemu) c
nemu: HIT GOOD TRAP at eip = 0x0010001b
```

## 3 Differential Testing

实现 diff-test，对比 NEMU 和 QEMU 中的寄存器结果是否不同。代码如下所示。

```
1 if(r.eax!=cpu.eax) {
2     printf("eax 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("ecx 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("edx 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("ebx 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("esp 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("ebp 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("esi 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("edi 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 }

```

最后，将 nemu/include/common.h 的注释打开，即可使用 diff-test。但在一键回归测试 bash run-all.sh 中，需要将 diff-test 关闭。

```
1 // #define DIFF_TEST
```

## 4 阶段二

阶段二需要实现更多指令，如下所示。

- ❖ 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: lea, nop

框架代码已经实现了上述红色标记的指令，但并没有填写 opcode\_table。此外，某些需要更新 EFLAGS 的指令并没有完全实现好(框架代码中已经插入了 TODO() 作为提示)，你还需要编写相应的功能。

### 4.1 mov 指令

mov 指令已实现，以下为 mov 指令相关的部分代码。

在 opcode\_table 的对应位置中填写译码函数和执行函数。

```
1 /* 0xc4 */ EMPTY, EMPTY, IDEXW(mov_I2E, mov, 1), IDEX(mov_I2E, mov),
```

执行函数：

```

1 make_EHelper(mov) {
2     operand_write(id_dest, &id_src->val);
3     print_asm_template2(mov);
4 }

```

## 4.2 leave 指令

在 opcode\_table 的对应位置中填写译码函数和执行函数。

```

1 /* 0xc8 */ EMPTY, EX(leave), EMPTY, EMPTY,

```

执行函数：

```

1 make_EHelper(leave) {
2     rtl_mv(&cpu.esp, &cpu.ebp);
3     rtl_pop(&cpu.ebp);
4     print_asm("leave");
5 }

```

## 4.3 lea 指令

继续运行 mov-c.c，输入 c，发现在操作码 8d 处停止。查表可知，该指令为 lea 指令。

```

(nemu) c
invalid opcode(eip = 0x00100048): 8d 4c 24 04 83 e4 f0 ff ...

```

在 opcode\_table 的对应位置中填写译码函数和执行函数。

```

1 /* 0x8c */ EMPTY, IDEX(lea_M2G, lea), EMPTY, IDEX(E, pop),

```

data-mov.c 的执行函数：

```

1 make_EHelper(lea) {
2     rtl_li(&t2, id_src->addr);
3     operand_write(id_dest, &t2);
4     print_asm_template2(lea);
5 }

```

## 4.4 group1

继续运行 mov-c.c，输入 c，发现在操作码 83 处停止。查表可知，该指令为 grp1。

```

(nemu) c
invalid opcode(eip = 0x0010004c): 83 e4 f0 ff 71 fc 55 89 ...

```

补充 grp1 的指令内容如下所示。

```

1  /* 0x80, 0x81, 0x83 */
2  make_group(gp1,
3      EX(add), EX(or), EX(adc), EX(sbb),
4      EX(and), EX(sub), EX(xor), EX(cmp))

```

#### 4.4.1 add 指令

在 opcode\_table 的对应位置中填写译码函数和执行函数。

```

1  /* 0x00 */ IDEXW(G2E, add, 1), IDEX(G2E, add), IDEXW(E2G, add, 1), IDEX(E2G, add),
2  /* 0x04 */ IDEXW(I2a, add, 1), IDEX(I2a, add), EMPTY, EMPTY,

```

arith.c 中的执行函数如下所示。add 指令与 sub 指令相仿，需要修改的地方为 CF 和 OF 的判别。

```

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

```

#### 4.4.2 or 指令

在 opcode\_table 的对应位置中填写译码函数和执行函数。

```

1  /* 0x08 */ IDEXW(G2E, or, 1), IDEX(G2E, or), IDEXW(E2G, or, 1), IDEX(E2G, or),
2  /* 0x0c */ IDEXW(I2a, or, 1), IDEX(I2a, or), EMPTY, EX(2byte_esc),

```

logic.c 中的执行函数如下所示。

```

1  make_EHelper(or) {
2      rtl_or(&t2, &id_dest->val, &id_src->val);
3      operand_write(id_dest, &t2);
4      rtl_update_ZFSF(&t2, id_dest->width);
5      rtl_set_CF(&tzero);
6      rtl_set_OF(&tzero);
7      print_asm_template2(or);
8

```



```

9 | print_asm_template2(or);
10 | }

```

#### 4.4.3 adc 指令

该指令已实现，仅需添加 opcode\_table 并注册。

```

1 | /* 0x10 */ IDEXW(G2E, adc, 1), IDEX(G2E, adc), IDEXW(E2G, adc, 1), IDEX(E2G,
   | adc),
2 | /* 0x14 */ IDEXW(I2a, adc, 1), IDEX(I2a, adc), EMPTY, EMPTY,

```

#### 4.4.4 sbb 指令

该指令已实现，仅需添加 opcode\_table 并注册。

```

1 | /* 0x18 */ IDEXW(G2E, sbb, 1), IDEX(G2E, sbb), IDEXW(E2G, sbb, 1), IDEX(E2G, sbb),
2 | /* 0x1c */ IDEXW(I2a, sbb, 1), IDEX(I2a, sbb), EMPTY, EMPTY,

```

#### 4.4.5 and 指令

在 opcode\_table 的对应位置中填写译码函数和执行函数。

```

1 | /* 0x20 */ IDEXW(G2E, and, 1), IDEX(G2E, and), IDEXW(E2G, and, 1), IDEX(E2G, and),
2 | /* 0x24 */ IDEXW(I2a, and, 1), IDEX(I2a, and), EMPTY, EMPTY,

```

logic.c 中的执行函数如下所示。

```

1 | make_EHelper(and) {
2 |     rtl_and(&t2,&id_dest->val,&id_src->val);
3 |     operand_write(id_dest,&t2);
4 |     rtl_update_ZFSF(&t2,id_dest->width);
5 |     rtl_set_CF(&tzero);
6 |     rtl_set_OF(&tzero);
7 |     print_asm_template2(and);
8 | }

```

#### 4.4.6 cmp 指令

在 opcode\_table 的对应位置中填写译码函数和执行函数。

```

1 | /* 0x38 */ IDEXW(G2E, cmp, 1), IDEX(G2E, cmp), IDEXW(E2G, cmp, 1), IDEX(E2G, cmp),
2 | /* 0x3c */ IDEXW(I2a, cmp, 1), IDEX(I2a, cmp), EMPTY, EMPTY,

```

arith.c 中的执行函数如下所示，只需修改 eflags 寄存器即可。

```

1 | make_EHelper(cmp) {
2 |     eflags_modify();
3 |     print_asm_template2(cmp);

```

```
4 }
```

## 4.5 group5

继续运行 mov-c.c, 输入 c, 发现在操作码 ff 处停止。查表可知, 该指令为 grp5。

```
(nemu) c
invalid opcode(eip = 0x0010004f): ff 71 fc 55 89 e5 51 83 ...
```

补充 grp5 的内容如下所示。其中, call\_rm 指令与 jmp\_rm 指令基本同 call 指令与 jmp 指令, 此后不多赘述。

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

### 4.5.1 inc 指令

在 opcode\_table 的对应位置中填写译码函数和执行函数。

```
1 /* 0x40 */ IDEX(r, inc), IDEX(r, inc), IDEX(r, inc), IDEX(r, inc),
2 /* 0x44 */ IDEX(r, inc), IDEX(r, inc), IDEX(r, inc), IDEX(r, inc),
```

arith.c 中的执行函数如下所示

```
1 make_EHelper(inc) {
2     rtl_addi(&t2, &id_dest->val, 1);
3     operand_write(id_dest, &t2);
4     rtl_update_ZFSF(&t2, id_dest->width);
5     rtl_eqi(&t0, &t2, 0x80000000);
6     rtl_set_OF(&t0);
7     print_asm_template1(inc);
8 }
```

### 4.5.2 dec 指令

opcode\_table 的对应位置中填写译码函数和执行函数。

```
1 /* 0x48 */ IDEX(r, dec), IDEX(r, dec), IDEX(r, dec), IDEX(r, dec),
2 /* 0x4c */ IDEX(r, dec), IDEX(r, dec), IDEX(r, dec), IDEX(r, dec),
```

arith.c 中的执行函数如下所示:

```
1 make_EHelper(dec) {
2     rtl_subi(&t2, &id_dest->val, 1);
3     operand_write(id_dest, &t2);
4     rtl_update_ZFSF(&t2, id_dest->width);
5     rtl_eqi(&t0, &t2, 0x7fffffff);
```

```

6   rtl_set_OF(&t0);
7   print_asm_template1(dec);
8 }

```

继续运行 moc-c.c 可以发现程序不在 ff 处停留，说明 grp5 的指令已完善。

```

(nemu) c
invalid opcode(eip = 0x0010009f): 6a 01 e8 86 ff ff ff 31 ...

```

查询 5A 处指令，为 push。补充 opcode\_table 后可顺利运行。

```

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

```

## 4.6 test 指令

继续运行 mov-c.c，输入 c，发现在操作码 85 处停止。查表可知，该指令为 test。

```

(nemu) c
invalid opcode(eip = 0x00100032): 85 c0 74 02 5d c3 c7 45 ...

```

opcode\_table 的对应位置中填写译码函数和执行函数。

```

1  /* 0x84 */  IDEXW(G2E, test, 1), IDEX(G2E, test), EMPTY, EMPTY,
2  /* 0xa8 */  IDEXW(I2a, test, 1), IDEX(I2a, test), EMPTY, EMPTY,

```

logic.c 中的执行函数如下所示

```

1  make_EHelper(test) {
2      rtl_and(&t2,&id_dest->val,&id_src->val);
3      rtl_update_ZFSF(&t2,id_dest->width);
4      rtl_set_CF(&tzero);
5      rtl_set_OF(&tzero);
6      print_asm_template2(test);
7  }

```

## 4.7 Jcc 指令

继续运行 mov-c.c，输入 c，发现在操作码 74 处停止。查表可知，该指令为 Jb。

```

(nemu) c
invalid opcode(eip = 0x00100034): 74 02 5d c3 c7 45 08 01 ...

```

opcode\_table 的对应位置中填写译码函数和执行函数。

```

1  /* 0x70 */  IDEXW(J, jcc, 1), IDEXW(J, jcc, 1), IDEXW(J, jcc, 1), IDEXW(J, jcc, 1),
2  /* 0x74 */  IDEXW(J, jcc, 1), IDEXW(J, jcc, 1), IDEXW(J, jcc, 1), IDEXW(J, jcc, 1),
3  /* 0x78 */  IDEXW(J, jcc, 1), IDEXW(J, jcc, 1), IDEXW(J, jcc, 1), IDEXW(J, jcc, 1),
4  /* 0x7c */  IDEXW(J, jcc, 1), IDEXW(J, jcc, 1), IDEXW(J, jcc, 1), IDEXW(J, jcc, 1),

```

在 2 byte\_opcode\_table 中完善执行函数和译码函数。

```

1  /* 0x80 */ IDEX(J, jcc), IDEX(J, jcc), IDEX(J, jcc), IDEX(J, jcc),
2  /* 0x84 */ IDEX(J, jcc), IDEX(J, jcc), IDEX(J, jcc), IDEX(J, jcc),
3  /* 0x88 */ IDEX(J, jcc), IDEX(J, jcc), IDEX(J, jcc), IDEX(J, jcc),
4  /* 0x8c */ IDEX(J, jcc), IDEX(J, jcc), IDEX(J, jcc), IDEX(J, jcc),

```

继续运行后报错，查看原因，发现是 cc.c 中需要查询 EFLAGS 寄存器以确定条件码是否满足。补全 nemu/src/cpu/exec/cc.c 中的 rtl\_setcc 函数，代码如下所示。

```

1  switch (subcode & 0xe) {
2      case CC_O:
3          rtl_get_OF(dest);
4          break;
5      case CC_B:
6          rtl_get_CF(dest);
7          break;
8      case CC_E:
9          rtl_get_ZF(dest);
10         break;
11     case CC_BE:
12         assert(dest!=&t0);
13         rtl_get_CF(dest);
14         rtl_get_ZF(&t0);
15         rtl_or(dest,dest,&t0);
16         break;
17     case CC_S:
18         rtl_get_SF(dest);
19         break;
20     case CC_L:
21         assert(dest!=&t0);
22         rtl_get_SF(dest);
23         rtl_get_OF(&t0);
24         rtl_xor(dest,dest,&t0);
25         break;
26     case CC_LE:
27         assert(dest!=&t0);
28         rtl_get_SF(dest);
29         rtl_get_OF(&t0);
30         rtl_xor(dest,dest,&t0);
31         rtl_get_ZF(&t0);
32         rtl_or(dest,dest,&t0);
33         break;
34     default:
35         panic("should not reach here");
36     case CC_P:
37         panic("n86 does not have PF");
38 }

```

## 4.8 group2

运行 bit.c, 输入 c, 发现在操作码 c1 处停止。查表可知, 该指令为 grp2。

```
(nemu) c
invalid opcode(eip = 0x00100050): c1 fa 03 83 e1 07 b8 01 ...
```

指令内容如下所示。

```
1  /* 0xc0, 0xc1, 0xd0, 0xd1, 0xd2, 0xd3 */
2  make_group(gp2,
3      EX(rol), EMPTY, EMPTY, EMPTY,
4      EX(shl), EX(shr), EMPTY, EX(sar))
```

- rol 指令将操作数向左循环移位, 最高位移出后重新进入最低位。
- shl 指令将操作数向左移位, 空出的低位用 0 填充。
- shr 指令将操作数向右移位, 空出的高位用 0 填充。
- sar 指令将操作数向右算术移位, 空出的高位用最高位的值填充, 保持符号不变。

补充完指令后再次运行, 发现可以成功执行 c1 的处的指令, 说明 group2 指令正确实现。具体补充内容如下所示。

```
(nemu) c
please implement me
nemu: ./include/cpu/rtl.h:140: rtl_sext: Assertion `0' failed.
Makefile:46: recipe for target 'run' failed
make[2]: *** [run] Aborted (core dumped)
/home/ly/Desktop/ics2017/nexus-am/Makefile.app:35: recipe for target 'run' failed
make[1]: *** [run] Error 2
Makefile:12: recipe for target 'Makefile.bit' failed
make: [Makefile.bit] Error 2 (ignored)
bit
```

### 4.8.1 rol 指令

logic.c 中的执行函数如下所示。

```
1 make_EHelper(rol) {
2     rtl_shri(&t2, &id_dest->val, id_dest->width * 8 - id_src->val);
3     rtl_shl(&t3, &id_dest->val, &id_src->val);
4     rtl_or(&t1, &t2, &t3);
5     operand_write(id_dest, &t1);
6     print_asm_template2(rol);
7 }
```

### 4.8.2 shl 指令

logic.c 中的执行函数如下所示。

```
1 make_EHelper(shl) {
2     rtl_shl(&t2, &id_dest->val, &id_src->val);
3     operand_write(id_dest, &t2);
```

```

4 rtl_update_ZFSF(&t2, id_dest->width);
5 print_asm_template2(shl);
6 }

```

### 4.8.3 shr 指令

logic.c 中的执行函数如下所示。

```

1 make_EHelper(shr) {
2     rtl_shr(&t2, &id_dest->val, &id_src->val);
3     operand_write(id_dest, &t2);
4     rtl_update_ZFSF(&t2, id_dest->width);
5     print_asm_template2(shr);
6 }

```

### 4.8.4 sar 指令

logic.c 中的执行函数如下所示。

```

1 make_EHelper(sar) {
2     rtl_sext(&t2, &id_dest->val, id_dest->width);
3     rtl_sar(&t2, &t2, &id_src->val);
4     operand_write(id_dest, &t2);
5     rtl_update_ZFSF(&t2, id_dest->width);
6     print_asm_template2(sar);
7 }

```

## 4.9 grp3

继续完善指令，grp3 内容如下所示。

```

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

```

其中，test 指令之前已实现。

mul2 的 2 byte\_opcode\_table 如下所示。

```

1 /* 0xac */ EMPTY, EMPTY, EMPTY, IDEX(E2G, imul2),

```

### 4.9.1 not 指令

logic.c 中的执行函数如下所示。

```

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

```

```

4  print_asm_template1(not);
5  }

```

#### 4.9.2 neg 指令

arith.c 中的执行函数如下所示。

```

1  make_EHelper(neg) {
2      rtl_sub(&t2, &tzero, &id_dest->val);
3      rtl_update_ZFSF(&t2, id_dest->width);
4      rtl_neq0(&t0, &id_dest->val);
5      rtl_set_CF(&t0);
6      rtl_eqi(&t0, &id_dest->val, 0x80000000);
7      rtl_set_OF(&t0);
8      operand_write(id_dest, &t2);
9      print_asm_template1(neg);
10 }

```

#### 4.10 movzx/movsx

因为 id\_src 和 id\_dest 两者的宽度不同，所以在进入函数时会重新调整 id\_dest 的宽度。因此，填写译码函数时，按照 id\_src 的宽度填写。

```

1  make_EHelper(movsx) {
2      id_dest->width = decoding.is_operand_size_16 ? 2 : 4;
3      rtl_sext(&t2, &id_src->val, id_src->width);
4      operand_write(id_dest, &t2);
5      print_asm_template2(movsx);
6  }
7
8  make_EHelper(movzx) {
9      id_dest->width = decoding.is_operand_size_16 ? 2 : 4;
10     operand_write(id_dest, &id_src->val);
11     print_asm_template2(movzx);
12 }

```

#### 4.11 cldt/cwtl 指令

cldt 指令是 x86 汇编语言中的一条指令，用于将 EAX 寄存器中的有符号整数扩展为一个双字 (DWORD) 有符号整数，并将结果存储在 EDX:EAX 寄存器对中。

cwtl 指令是 x86 汇编语言中的一条指令，用于将 AL 寄存器中的有符号整数扩展为一个字 (WORD) 有符号整数，并将结果存储在 AX 寄存器中。

opcode\_table 的对应位置中填写译码函数和执行函数：

```

1  /* 0x98 */ EX(cwtl), EX(cldt), EMPTY, EMPTY,

```

data-mov.c 中的执行函数：

```

1 make_EHelper(cld) {
2     if (decoding.is_operand_size_16) {
3         rtl_lr_w(&t0, R_AX);
4         rtl_sext(&t0, &t0, 2);
5         rtl_sari(&t0, &t0, 31);
6         rtl_sr_w(R_DX, &t0);
7     }
8     else {
9         rtl_sari(&cpu.edx, &cpu.eax, 31);
10    }
11
12    print_asm(decoding.is_operand_size_16 ? "cwtl" : "cld");
13 }
14
15 make_EHelper(cwtl) {
16     if (decoding.is_operand_size_16) {
17         rtl_lr_b(&t0, R_AX);
18         rtl_sext(&t0, &t0, 1);
19         rtl_sr_w(R_AX, &t0);
20     }
21     else {
22         rtl_lr_w(&t0, R_AX);
23         rtl_sext(&t0, &t0, 2);
24         rtl_sr_l(R_EAX, &t0);
25     }
26    print_asm(decoding.is_operand_size_16 ? "cbtw" : "cwtl");
27 }

```

## 4.12 nop 指令

opcode\_table 的对应位置中填写译码函数和执行函数：

```

1 /* 0x90 */ EX(nop), EMPTY, EMPTY, EMPTY,

```

## 4.13 实验结果

经过逐步调整后，在终端输入“bash runall.sh”命令，对每个样例进行测试。由结果可知，所有样例都能通过测试，说明大部分指令能够正确补充，阶段二告一段落。



```

ly@ubuntu:~/Desktop/ics2017/nemu$ bash runall.sh
NEMU compile OK
compiling testcases...
testcases compile OK
[ add-longlong] PASS!
[ add] PASS!
[ bit] PASS!
[ bubble-sort] PASS!
[ dummy] PASS!
[ fact] PASS!
[ fib] PASS!
[ goldbach] PASS!
[ hello-str] PASS!
[ if-else] PASS!
[ leap-year] PASS!
[ load-store] PASS!
[ matrix-mul] PASS!
[ max] PASS!
[ min3] PASS!
[ mov-c] PASS!
[ movsx] PASS!
[ mul-longlong] PASS!
[ pascal] PASS!
[ prime] PASS!
[ quick-sort] PASS!
[ recursion] PASS!
[ select-sort] PASS!
[ shift] PASS!
[ shuixianhua] PASS!
[ string] PASS!
[ sub-longlong] PASS!
[ sum] PASS!
[ switch] PASS!
[ to-lower-case] PASS!
[ unalign] PASS!
[ wanshu] PASS!

```

## 5 阶段三

在 NEMU 中加入 IOE, 只需要在 `nemu/include/common.h` 中定义宏 `HAS_IOE`. 定义后, `init_device()` 函数会对设备进行初始化。重新编译后, 运行 NEMU 时会弹出一个新窗口, 用于显示 VGA 的输出。

### 5.1 理解 volatile 关键字

```

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 }

```

在给定的代码中, `volatile` 关键字告诉编译器不要对指针 `p` 所指向的内存进行优化, 因为该内存地址可能被外部因素改变。在这种情况下, 假设地址 `0x8049000` 最终被映射到一个设备寄存器, 这个设备寄存器的值可能会在没有程序直接操作的情况下被修改。

如果去掉 `volatile` 关键字并使用 `-O2` 优化编译代码, 编译器可能会认为指针 `p` 所指向的内存是普通的内存, 而不是一个设备寄存器。编译器可能会对访问该内存的操作进行优化, 例如缓存该内存的值, 重排指令顺序, 或者进行其他优化, 这可能导致程序行为不符合预期。

在处理与设备寄存器交互的代码时，使用 `volatile` 关键字是很重要的，因为设备寄存器的值可能会被外部事件（例如硬件中断）随时修改，而编译器通常不会意识到这一点。因此，`volatile` 关键字告诉编译器不要对这样的变量进行优化，以确保程序与外部环境的交互正常进行。

## 5.2 串口

为了让程序使用串口进行输出，还需要在 NEMU 中实现端口映射 I/O，实现 `in,out` 指令。  
opcode\_table 的对应位置中填写译码函数和执行函数：

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

在 system.c 中填写执行函数：

```
1  make_EHelper(in) {
2      rtl_li(&t0, pio_read(id_src->val, id_dest->width));
3      operand_write(id_dest, &t0);
4
5      print_asm_template2(in);
6
7  #ifdef DIFF_TEST
8      diff_test_skip_qemu();
9  #endif
10 }
11
12 make_EHelper(out) {
13     pio_write(id_dest->val, id_src->width, id_src->val);
14
15     print_asm_template2(out);
16
17 #ifdef DIFF_TEST
18     diff_test_skip_qemu();
19 #endif
20 }
```

开启宏定义 `HAS_SERIAL`，运行 `hello` 程序，结果如下所示。程序往终端输出了 10 行 `Hello World!`，说明 I/O 端口被正确实现。

```
ly@ubuntu:~/Desktop/ics2017/nexus-am/apps/hello$ make run
Building hello [native]
make[1]: Entering directory '/home/ly/Desktop/ics2017/nexus-am'
make[2]: Entering directory '/home/ly/Desktop/ics2017/nexus-am/am'
Building am [native]
make[2]: Nothing to be done for 'archive'.
make[2]: Leaving directory '/home/ly/Desktop/ics2017/nexus-am/am'
make[1]: Leaving directory '/home/ly/Desktop/ics2017/nexus-am'
make[1]: Entering directory '/home/ly/Desktop/ics2017/nexus-am/libs/klib'
make[1]: *** No targets specified and no makefile found. Stop.
make[1]: Leaving directory '/home/ly/Desktop/ics2017/nexus-am/libs/klib'
/home/ly/Desktop/ics2017/nexus-am/Makefile.compile:86: recipe for target 'klib'
failed
make: [klib] Error 2 (ignored)
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

### 5.3 时钟

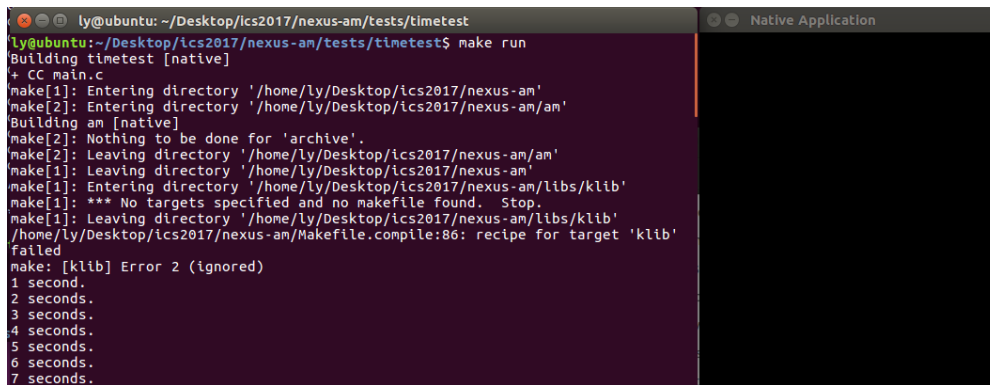
nemu/src/device/timer.c 模拟了 i8253 计时器的功能. 计时器的大部分功能都被简化, 只保留了”发起时钟中断”的功能, 同时添加了一个自定义的 RTC(Real Time Clock), 初始化时将会注册 0x48 处的端口作为 RTC 寄存器,CPU 可以通过 I/O 指令访问这一寄存器, 获得当前时间 (单位是 ms)。

#### 5.3.1 实现 IOE

在 nexus-am/am/arch/x86-nemu/src/ioe.c 中实现 \_\_uptime(), 如下所示。

```
1 unsigned long __uptime() {
2     return inl(RTC_PORT) - boot_time;
3 }
```

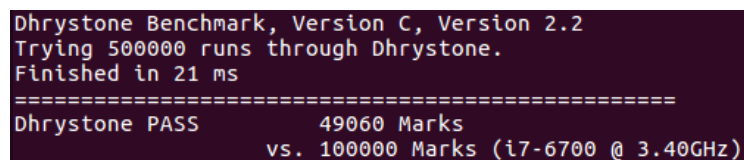
运行 timetest, 发现程序每隔 1 秒输出一句话, 说明时钟功能已被正确实现。



```
ly@ubuntu: ~/Desktop/ics2017/nexus-am/tests/timetest
ly@ubuntu:~/Desktop/ics2017/nexus-am/tests/timetest$ make run
Building timetest [native]
+ CC main.c
make[1]: Entering directory '/home/ly/Desktop/ics2017/nexus-am'
make[2]: Entering directory '/home/ly/Desktop/ics2017/nexus-am/am'
Building am [native]
make[2]: Nothing to be done for 'archive'.
make[2]: Leaving directory '/home/ly/Desktop/ics2017/nexus-am/am'
make[1]: Leaving directory '/home/ly/Desktop/ics2017/nexus-am'
make[1]: Entering directory '/home/ly/Desktop/ics2017/nexus-am/libs/klib'
make[1]: *** No targets specified and no makefile found. Stop.
make[1]: Leaving directory '/home/ly/Desktop/ics2017/nexus-am/libs/klib'
/home/ly/Desktop/ics2017/nexus-am/Makefile.compile:86: recipe for target 'klib'
failed
make: [klib] Error 2 (ignored)
1 second.
2 seconds.
3 seconds.
4 seconds.
5 seconds.
6 seconds.
7 seconds.
```

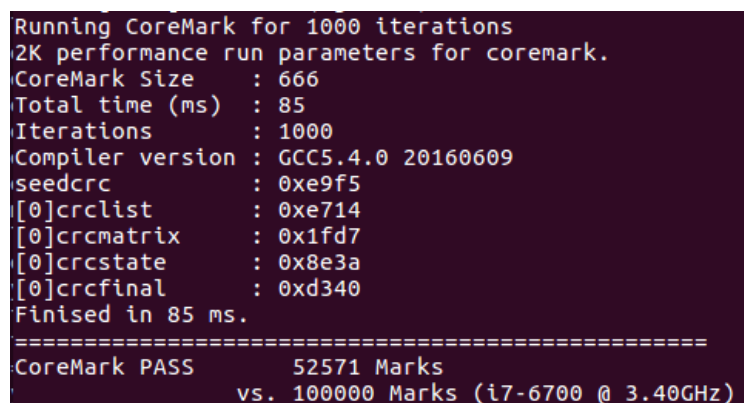
#### 5.3.2 看看 NEMU 跑多快

在 x86 下进行跑分测试。Dhrystone 的分数如下图所示, 49060Marks, 性能还可以。



```
Dhrystone Benchmark, Version C, Version 2.2
Trying 500000 runs through Dhrystone.
Finished in 21 ms
=====
Dhrystone PASS          49060 Marks
                        vs. 100000 Marks (i7-6700 @ 3.40GHz)
```

Coremark 的分数如下所示, 52571 Marks, 性能还不错。



```
Running CoreMark for 1000 iterations
2K performance run parameters for coremark.
CoreMark Size      : 666
Total time (ms)    : 85
Iterations         : 1000
Compiler version   : GCC5.4.0 20160609
SeedCRC            : 0xe9f5
[0]crclist         : 0xe714
[0]crcmatrix       : 0x1fd7
[0]crcstate        : 0x8e3a
[0]crcfinal        : 0xd340
Finished in 85 ms.
=====
CoreMark PASS      52571 Marks
                  vs. 100000 Marks (i7-6700 @ 3.40GHz)
```

Microbench 的分数如下所示，70468 Marks，性能较好。

```
min time: 22 ms [89059]
=====
MicroBench PASS      70468 Marks
                        vs. 100000 Marks (i7-6700 @ 3.40GHz)
Exit (0)
```

## 5.4 键盘

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。

### 5.4.1 如何检测多个键同时被按下

在游戏程序中，可以通过轮询键盘状态来获取当前按下的键，然后将这些键对应的键盘码进行按位运算，从而实现判断玩家是否同时按下了多个键的功能。这种方法可以灵活地应用在各种游戏场景中，如八方向行走、组合招式等。

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，这样来判断按键是否是在一起被按下。

### 5.4.2 实现 IOE(2)

在 ioe.c 中实现 \_\_read\_key()，如下所示。

```
1 int __read_key() {
2     uint32_t key_code = _KEY_NONE;
3     if (inb(0x64) )
4         key_code = inl(0x60);
5     return key_code;
6 }
```

在 NEMU 中运行 keytest 程序如下所示。在程序运行时弹出的新窗口中按下按键，可以看到程序输出相应的按键信息。

```
ly@ubuntu: ~/Desktop/ics2017/nexus-am/tests/keytest
make[1]: Leaving directory '/home/ly/Desktop/ics2017/nexus-am'
make[1]: Entering directory '/home/ly/Desktop/ics2017/nexus-am/libs/klib'
make[1]: *** No targets specified and no makefile found.  Stop.
make[1]: Leaving directory '/home/ly/Desktop/ics2017/nexus-am/libs/klib'
/home/ly/Desktop/ics2017/nexus-am/Makefile.compile:86: recipe for target 'klib'
failed
make: [klib] Error 2 (ignored)
Get key: 48 H down
Get key: 48 H up
Get key: 31 E down
Get key: 31 E up
Get key: 51 L down
Get key: 51 L up
Get key: 51 L down
Get key: 51 L up
Get key: 37 O down
Get key: 37 O up
Get key: 30 W down
Get key: 30 W up
Get key: 67 LCTRL down
Get key: 69 LALT down
Get key: 67 LCTRL up
Get key: 69 LALT up
```

## 5.5 VGA

### 5.5.1 神奇的调色板

具体来说，实现渐出渐入效果的步骤如下：

1. 创建一组不同的调色板，每个调色板包含不同的颜色组合。
2. 在屏幕上显示一个调色板，并逐渐将其颜色切换到下一个调色板，从而实现颜色的渐出效果。
3. 同时，逐渐将下一个调色板的颜色切换到当前调色板，实现颜色的渐入效果。
4. 不断重复上述步骤，就可以创建出流畅的渐出渐入动画效果。

通过这种方法，游戏开发者可以在有限的颜色深度下实现丰富多彩的动画效果，而不需要每一帧都重新绘制整个画面。这种技巧在当时的游戏开发中非常流行，因为可以在资源有限的情况下实现视觉上引人注目的动画效果。

### 5.5.2 添加内存映射 I/O

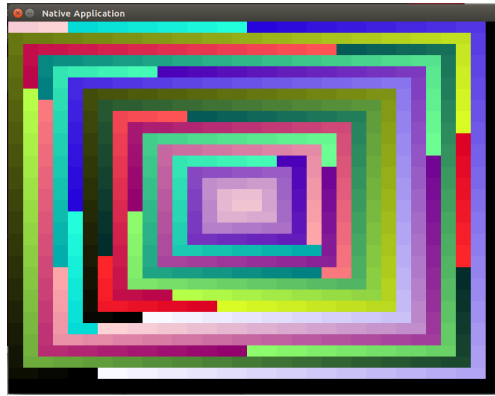
在 `paddr_read()` 和 `paddr_write()` 中加入对内存映射 I/O 的判断. 通过 `is_mmio()` 函数判断一个物理地址是否被映射到 I/O 空间, 如果是, `is_mmio()` 会返回映射号, 否则返回-1. 内存映射 I/O 的访问需要调用 `mmio_read()` 或 `mmio_write()`, 调用时需要提供映射号. 如果不是内存映射 I/O 的访问, 就访问 `pmem`.

```

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

```

`videotest` 的运行结果如下所示，可以看到看到新窗口中输出了一些颜色信息，说明内存映射 I/O 实现正确。



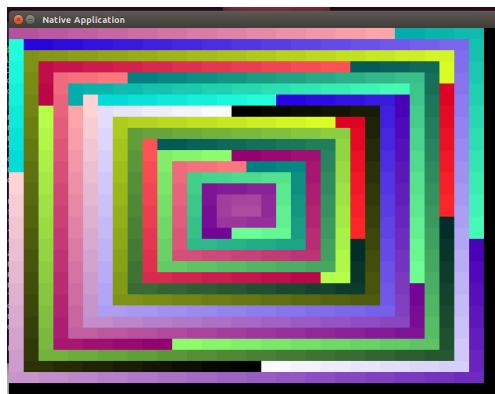
### 5.5.3 实现 IOE(3)

事实上, 刚才输出的颜色信息并不是 videotest 输出的画面, 这是因为框架代码中的 `_draw_rect()` 并未正确实现其功能, 需要实现正确的 `_draw_rect()`。

在 `nexus-am/am/arch/x86-nemu/src/ioe.c` 中实现 `_draw_rect()`, 如下所示。

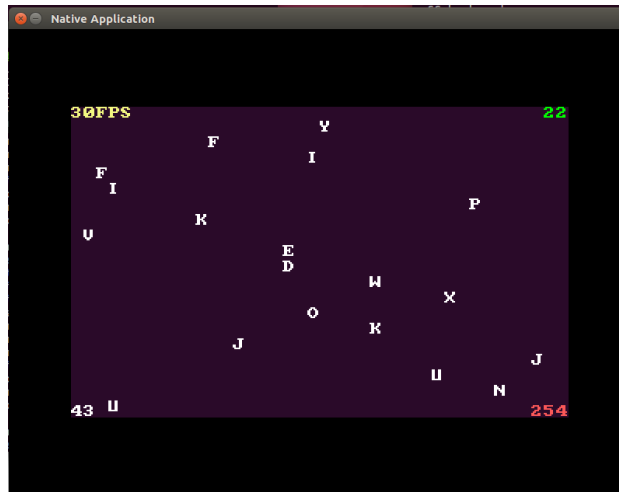
```
1 void _draw_rect(const uint32_t *pixels, int x, int y, int w, int h) {  
2     int temp = (w > _screen.width - x)?_screen.width - x :w;  
3     int cp_bytes = sizeof(uint32_t) *temp;  
4     for(int j = 0; j < h && y + j < _screen.height; j++) {  
5         memcpy(&fb[(y + j) * _screen.width + x], pixels, cp_bytes);  
6         pixels += w;  
7     }  
8 }
```

重新运行 videotest, 得到的结果如下所示。可以看到, 窗口除了输出颜色以外, 还有动画效果。



### 5.5.4 运行打字小游戏

运行 typing 程序, 如下图所示。可以发现能够游玩, 说明上述功能都已实现。



### 5.5.5 运行红白机



## 6 必答题

### 6.1 Q1: static 与 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?

#### 1. 去掉 `static`

选择其中一个 `rtl` 函数, 去掉 `static` 后, 一键回归测试, 发现可以通过测试。

这是因为在 C 语言中, `static inline` 函数在编译时会被强制内联, 即编译器会尝试将函数体直接插入到调用该函数的地方, 而不生成函数调用的指令。这样可以减少函数调用的开销, 提高程序的执行效率。

当去掉 `static` 关键字后, 函数就变成了普通的内联函数, 编译器会根据自身的内联策略决定是否将函数内联。如果编译器认为内联函数较小且频繁调用, 就会选择内联该函数。因此, 即使去掉 `static` 关键字, 编译器仍然有可能选择内联该函数, 从而保证程序的正确运行。

综上所述, static inline 函数是强制内联的, 而去掉 static 后, 编译器有权利选择是否内联该函数, 因此在某些情况下仍然能够正确运行。

## 2. 去掉 inline

去掉 inline 后, 会出现定义未使用的问题。

这是因为在 C 语言中, static inline 函数的 inline 关键字用于告诉编译器将函数内容内联展开, 而 static 关键字用于将函数的作用域限定在当前文件中, 避免与其他文件中的同名函数冲突。

当去掉 inline 关键字后, 原本应该内联展开的函数变成了普通的函数, 但由于函数前面仍然有 static 关键字, 函数的作用域仍然被限定在当前文件中。这就导致了一个问题: 虽然函数被定义了, 但在当前文件中却没有被使用, 编译器会认为这个函数是无用的, 从而产生“定义未使用”的警告。

## 3. 二者皆去

static 和 inline 都去掉后, 会出现重定义问题。

当去掉 static 和 inline 关键字后, 原本应该是内联函数且作用域为当前文件的函数就变成了普通的外部函数。如果这个函数被包含在多个源文件中, 编译器会在链接阶段发现同名函数的多重定义, 从而产生重复定义的问题。

## 6.2 Q2: 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。

Makefile 文件会定义了一系列规则, 包括文件的依赖关系和生成目标的指令。在执行 make 命令时, make 程序会根据这些规则来判断哪些文件需要重新编译, 哪些文件已经是最新的, 从而进行相应的编译和链接操作。makefile 具体工作方式如下所示:

1. 当执行 make 命令时, make 程序会读取当前目录下的 Makefile 文件, 并根据其中定义的规则和指令来决定如何编译和链接源文件。
2. Makefile 中包含了一系列规则, 每个规则由一个目标 (target)、依赖关系 (dependencies) 和命令 (commands) 组成。
3. make 程序会检查所有的.c 文件和.h 文件的依赖关系, 如果某个.c 文件或.h 文件被修改过, 或者它们的依赖文件被修改过, 就会重新编译这个文件。
4. make 程序根据 Makefile 中的指令来编译每个.c 文件, 生成对应的目标文件 (.o 文件)。
5. 链接所有的目标文件, 生成可执行文件 nemu/build/nemu。

## 7 Bug 汇总

### 7.1 移动文件夹导致环境中路径不一致

在正式实验开始之前, 我移动过整个文件夹, 导致当前路径与配置环境变量时的路径不一致, 在运行 dummy.c 程序时, 出现了去错误的路径查找目标文件的问题。

解决办法为: 在终端使用“vim ~/.bashrc”命令, 修改环境变量路径, 然后在需要运行程序的文件夹下执行 source 命令使环境变量生效, 问题解决。

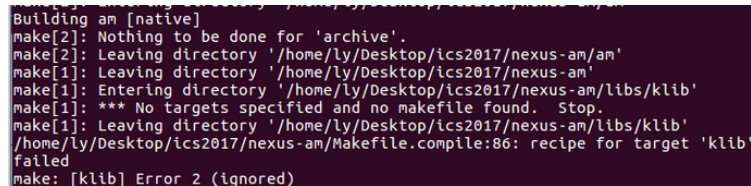


## 7.2 2-byte opcode

有些指令 (例如 jcc 指令) 是两位的, 因此需要去 2 byte\_opcode\_table 中的对应位置填写译码函数和执行函数, 否则在运行程序时, 会发现 0f 的指令未实现。

## 7.3 klib 文件夹找不到目标文件

运行 hello world 程序, 进入 klib 文件夹时显示找不到目标及没有 makefile, 如下图所示。



```
Building am [native]
make[2]: Nothing to be done for 'archive'.
make[2]: Leaving directory '/home/ly/Desktop/ics2017/nexus-am/am'
make[1]: Leaving directory '/home/ly/Desktop/ics2017/nexus-am'
make[1]: Entering directory '/home/ly/Desktop/ics2017/nexus-am/libs/klib'
make[1]: *** No targets specified and no makefile found. Stop.
make[1]: Leaving directory '/home/ly/Desktop/ics2017/nexus-am/libs/klib'
/home/ly/Desktop/ics2017/nexus-am/Makefile.compile:86: recipe for target 'klib'
failed
make: [klib] Error 2 (ignored)
```

这是由于 ics2017 的仓库中只有.a 文件, 没有 src 文件和 makefile。尝试从 ics2019 仓库中将 src 文件夹和 makefile 文件搬过来后, 出现提示没有”amdev.h”文件。再次从 2019 仓库中复制过来, 运行后显示多处地方出现重定义。

最后询问助教得知, 该问题可以不用解决, 只需要 NEMU 能正确运行即可。