



南開大學  
Nankai University

计算机学院  
计算机系统设计实验报告

**PA2**

姓名：高祎珂

学号：2011743

专业：计算机科学与技术

2023 年 4 月 18 日

# 目录

<b>1 实验目的</b>	<b>2</b>
<b>2 实验内容</b>	<b>2</b>
<b>3 阶段一</b>	<b>2</b>
3.1 运行 dummy . . . . .	2
3.2 添加指令 . . . . .	2
3.3 rtl 基本指令 . . . . .	4
3.4 实现对应指令 . . . . .	6
3.5 运行结果 . . . . .	8
<b>4 阶段二</b>	<b>8</b>
4.1 实现新指令 . . . . .	8
4.2 实现 differential testing . . . . .	22
<b>5 阶段三</b>	<b>23</b>
5.1 串口 . . . . .	23
5.2 时钟 . . . . .	25
5.3 键盘 . . . . .	27
5.4 VGA . . . . .	28
<b>6 必答题</b>	<b>30</b>
6.1 问题一 . . . . .	30
6.2 问题二 . . . . .	30
6.3 问题三 . . . . .	30
6.4 问题四 . . . . .	31
6.5 问题五 . . . . .	33

## 1 实验目的

在计算机技术的发展历程中，先驱们为创造计算机世界所做出的努力不可忽视。这些先驱们准备好了各种工具，用以开创计算机领域的先河。为了迈出第一步，他们运用了数字电路的知识，成功地创造出了最小的计算机——图灵机。如今，计算机技术已经发展成为支撑现代社会的重要基石，而图灵机则成为计算机理论的经典代表之一。本次实验基于 PA1 实现的基础设施等内容继续进行指令的编写，完成大部分指令以及加入 IOE。

## 2 实验内容

本次实验总共分为以下几个阶段

- 补充指令，通过 dummy 程序
- 实现 diff-test，便于实验，完善更多指令，通过 cputest，
- 增加 IOE，实现串口、时钟、键盘、VGA

## 3 阶段一

### 3.1 运行 dummy

根据实验指导书的提示，使用指令 `make ARCH=x86-nemu ALL=dummy run`，编译 dummy 程序，并未对代码做任何修改，可以看到输出下面的结果：

```
gyk@ubuntu:~/ics2017/nemu$ cd ../
gyk@ubuntu:~/ics2017$ cd nexus-am/tests/cputest/
gyk@ubuntu:~/ics2017/nexus-am/tests/cputest$ make ARCH=x86-nemu ALL=dummy run
Building dummy [x86-nemu]
Building an [x86-nemu]
make[2]: *** No targets specified and no makefile found. Stop.
[src/monitor/monitor.c,65,load_img] The image is /home/gyk/ics2017/nexus-am/tests/cputest/build/dummy-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 01:09:44, Apr 17 2023
For help, type "help"
(nemu) si
100000: bd 00 00 00 00 movl $0x0,%ebp
(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
03:04 Manual
for more details.

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

(nemu) █
```

图 3.1: 初始结果

### 3.2 添加指令

出现这种结果是因为还有指令未实现，查看反汇编结果，如下：

## dummy 反汇编

```

1 00100000 <_start>:
2   100000:  bd 00 00 00 00      mov     $0x0,%ebp
3   100005:  bc 00 7c 00 00      mov     $0x7c00,%esp
4   10000a:  e8 01 00 00 00      call    100010 <_trm_init>
5   10000f:  90                  nop
6
7 00100010 <_trm_init>:
8   100010:  55                  push    %ebp
9   100011:  89 e5              mov     %esp,%ebp
10  100013:  83 ec 08           sub     $0x8,%esp
11  100016:  e8 05 00 00 00      call    100020 <main>
12  10001b:  d6                (bad)
13  10001c:  eb fe              jmp     10001c <_trm_init+0xc>
14  10001e:  66 90              xchg    %ax,%ax
15
16 00100020 <main>:
17  100020:  55                  push    %ebp
18  100021:  89 e5              mov     %esp,%ebp
19  100023:  31 c0              xor     %eax,%eax
20  100025:  5d                  pop     %ebp
21  100026:  c3                  ret

```

从反汇编文件我们可以看出需要实现以下的一些指令：

指令	编码	Description
call	e8	Call near, displacement relative to next instruction
push	50 + rw /rb	Push register word/dword
sub	83	Subtract sign-extended immediate byte from r/m word
xor	31	Exclusive-OR dword register to r/m word/dword
pop	58+rw/rb	Pop top of stack into word/dword register
ret	c3	Return (near) to caller

根据 i386 手册，我们需要首先补充 opcode\_table 如下：

## 补充 opcode\_table

```

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))
5  /* 0xc0, 0xc1, 0xd0, 0xd1, 0xd2, 0xd3 */
6  make_group(gp2,
7      EX(rol), EMPTY, EMPTY, EMPTY,
8      EX(shl), EX(shr), EMPTY, EX(sar))
9
10 /* 0xe8 */    IDEX(J, call), IDEX(J, jmp), IDEX(I, jmp_rm), IDEXW(J, jmp, 1),
11 /* 0x50 */    IDEX(r, push), IDEX(r, push), IDEX(r, push), IDEX(r, push),
12 /* 0x54 */    IDEX(r, push), IDEX(r, push), IDEX(r, push), IDEX(r, push),
13 /* 0x58 */    IDEX(r, pop), IDEX(r, pop), IDEX(r, pop), IDEX(r, pop),

```

```

14 /* 0x5c */    IDEX(r , pop) , IDEX(r , pop) , IDEX(r , pop) , IDEX(r , pop) ,
15 /* 0x30 */    IDEXW(G2E, xor, 1) , IDEX(G2E, xor) , IDEXW(E2G, xor, 1) , IDEX(E2G, xor) ,
16 /* 0x34 */    IDEXW(I2a, xor, 1) , IDEX(I2a, xor) , EMPTY, EMPTY,
17 /* 0xc0 */    IDEXW(gp2_Ib2E, gp2, 1) , IDEX(gp2_Ib2E, gp2) , IDEXW(I, ret, 2) , EX(ret) ,

```

此外还需要在 all-intr.h 中进行执行函数的声明

#### 声明执行函数

```

1  make_EHelper( call);    // control.c
2  make_EHelper( push);    // data-mov.c
3  make_EHelper( sub);     // arith.c
4  make_EHelper( xor);     // logic.c
5  make_EHelper( pop);     // data-mov.c
6  make_EHelper( ret);     // control.c

```

### 3.3 rtl 基本指令

想要实现上述的指令，我们还要实现一些基础的 rtl 指令，在 rtl.h 文件中，需要补充 pop 和 push 指令，根据代码给的提示 push 和 pop 用于将一个寄存器值存入栈中（压栈）和将一个栈中的值取出并存入寄存器中（弹栈）。对于 push 指令可首先利用 subi 将栈指针寄存器 cpu.esp 减去 4，实现将栈指针向下移动 4 字节，从而为新数据腾出空间。然后利用 store 将 src1 的值存入栈中，pop 则相反，具体代码如下：

#### rtl 基本 pop, push 补充

```

1  static inline void rtl_push(const rtlreg_t* src1) {
2      // esp <- esp - 4
3      // M[esp] <- src1
4      //TODO();
5      rtl_subi(&cpu.esp, &cpu.esp, 4);
6      rtl_sm(&cpu.esp, 4, src1);
7  }
8
9  static inline void rtl_pop(rtlreg_t* dest) {
10     // dest <- M[esp]
11     // esp <- esp + 4
12     // TODO();
13     rtl_lm(dest, &cpu.esp, 4);
14     rtl_addi(&cpu.esp, &cpu.esp, 4);
15 }

```

为实现减法指令，我们还需实现标志位，首先需要对寄存器进行修改，加入标志位，修改如下：

#### 寄存器结构体修改

```

1  typedef struct {
2      union{
3          /* data */
4          union {

```

```

5     uint32_t _32;
6     uint16_t _16;
7     uint8_t _8[2];
8 } gpr[8];
9 struct
10 {
11     rtlreg_t eax, ecx, edx, ebx, esp, ebp, esi, edi;
12 };
13 };
14 /* Do NOT change the order of the GPRs' definitions. */
15
16 /* In NEMU, rtlreg_t is exactly uint32_t. This makes RTL instructions
17  * in PA2 able to directly access these registers.
18  */
19 vaddr_t eip;
20
21 struct bs {
22     unsigned int CF:1;
23
24     unsigned int one:1;
25     unsigned int :4;
26     unsigned int ZF:1;
27     unsigned int SF:1;
28
29     unsigned int :1;
30     unsigned int IF:1;
31     unsigned int :1;
32     unsigned int OF:1;
33     unsigned int :20;
34 } eflags;
35
36 } CPU_state;

```

标志位的实现如下：

#### 实现标志位

```

1 static inline void rtl_msb(rtlreg_t* dest, const rtlreg_t* src1, int width) {
2     // dest <- src1[width * 8 - 1]
3     //TODO();
4     rtl_shri(dest, src1, width*8-1);
5     rtl_andi(dest, dest, 0x1);
6 }
7
8 static inline void rtl_update_ZF(const rtlreg_t* result, int width) {
9     // eflags.ZF <- is_zero(result[width * 8 - 1 .. 0])
10    //TODO();
11    rtl_andi(&t0, result, (0xffffffffu >> (4-width)*8));
12    rtl_eq0(&t0, &t0);
13    rtl_set_ZF(&t0);

```

```

14 }
15
16 static inline void rtl_update_SF(const rtlreg_t* result, int width) {
17     // eflags.SF <- is_sign(result[width * 8 - 1 .. 0])
18     // TODO();
19     assert(result != &t0);
20     rtl_msb(&t0, result, width);
21     rtl_set_SF(&t0);
22 }

```

### 3.4 实现对应指令

#### call

```

1 make_EHelper(call) {
2     // the target address is calculated at the decode stage
3     // TODO();
4     rtl_li(&t2, decoding.seq_eip);
5     rtl_push(&t2);
6
7     decoding.is_jump = 1;
8
9     print_asm("call %x", decoding.jump_eip);
10 }

```

#### ret

```

1 make_EHelper(ret) {
2     // TODO();
3     rtl_pop(&t2);
4     decoding.jump_eip = t2;
5     decoding.is_jump = 1;
6     print_asm("ret");
7 }

```

#### push

```

1 make_EHelper(push) {
2     // TODO();
3     rtl_push(&id_dest -> val);
4     print_asm_template1(push);
5 }

```

#### pop

```

1 make_EHelper(pop) {
2     // TODO();
3     rtl_pop(&t2);

```

```

4 operand_write(id_dest, &t2);
5 print_asm_template1(pop);
6 }

```

## XOR

```

1 make_EHelper(xor) {
2     // TODO();
3     rtl_xor(&t2, &id_dest -> val, &id_src -> val);
4     operand_write(id_dest, &t2);
5
6     rtl_update_ZFSF(&t2, id_dest -> width);
7
8     rtl_set_CF(&tzero);
9     rtl_set_OF(&tzero);
10
11     print_asm_template2(xor);
12 }

```

## sub

```

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 }
12 make_EHelper(sub) {
13     // TODO();
14
15     eflags_modify();
16     operand_write(id_dest, &t2);
17     print_asm_template2(sub);
18 }
19 static inline make_DopHelper(SI) {
20     assert(op -> width == 1 || op -> width == 4);
21
22     op -> type = OP_TYPE_IMM;
23     op -> simm = instr_fetch(eip, op -> width);
24     if(op -> width == 1) {
25         op -> simm = (int8_t)op -> simm;
26     }
27
28     rtl_li(&op -> val, op -> simm);

```



```

29
30 #ifdef DEBUG
31     snprintf(op->str, OP_STR_SIZE, "$0x%x", op->simmm);
32 #endif
33 }

```

### 3.5 运行结果

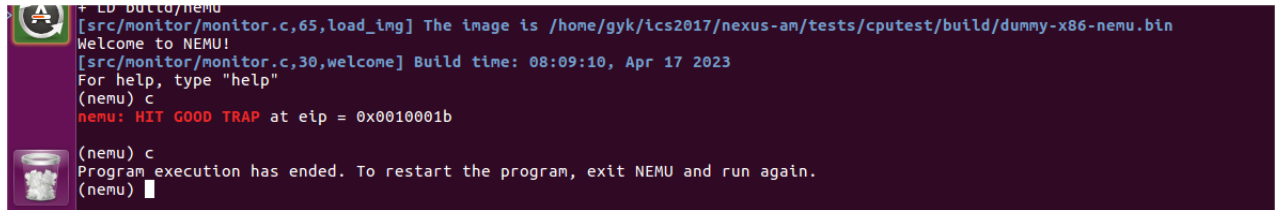


图 3.2: 阶段一运行结果

从图中可以看到，已经完整实现了 dummy 程序所需的指令，实现了功能，阶段一就结束了。

## 4 阶段二

阶段二其实就是阶段一的延续，这里需要实现更多的指令，以通过 cputest 中的所有样例，具体操作就是根据整个 cputest 中的文件 make 一遍，通过查看反汇编文件列出自己需要实现的指令。然后按照阶段一的流程去实现和测试。本阶段需要实现以下所有指令：

### 4.1 实现新指令

- Data Movement Instructions:  
push, pop, leave, cltd (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

其中 xor、push、pop、call、ret、sub 在 PA2.1 已实现，只需实现其余指令即可。  
根据 i386 官方手册给出的指令集表，如下图：

	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, Rv	~FS	~GS	Size	Size	Iv	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, Ib	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		RBT 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	JMP		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

图 4.3: One-Byte Opcode Map

	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)								SETS	SETNS	SETP	SETNP	SETL	SETNL	SETLE	SETNLE
	SETO	SETNO	SETB	SETNB	SETZ	SETNZ	SETBE	SETNBE								
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 Gv, Ew				Grp-8 Ev, Ib	BIC Ev, Gv	BSF Gv, Ev	BSR Gv, Ev	MOVSSX Gv, Eb Gv, Ew	
C																
D																
E																
F																

图 4.4: Two-Byte Opcode Map

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

G r o u p	<table><tr><td colspan="2">mod</td><td colspan="2">nnn</td><td colspan="2">R/M</td><td colspan="2"></td></tr></table>								mod		nnn		R/M			
	mod		nnn		R/M											
000	001	010	011	100	101	110	111									
1	<a href="#">ADD</a>	<a href="#">OR</a>	<a href="#">ADC</a>	<a href="#">SBB</a>	<a href="#">AND</a>	<a href="#">SUB</a>	<a href="#">XOR</a>	<a href="#">CMP</a>								
2	<a href="#">ROL</a>	<a href="#">ROR</a>	<a href="#">RCL</a>	<a href="#">RCR</a>	<a href="#">SHL</a>	<a href="#">SHR</a>		<a href="#">SAR</a>								
3	<a href="#">TEST</a> Ib/Iv		<a href="#">NOT</a>	<a href="#">NEG</a>	<a href="#">MUL</a> AL/eAX	<a href="#">IMUL</a> AL/eAX	<a href="#">DIV</a> AL/eAX	<a href="#">IDIV</a> AL/eAX								
4	<a href="#">INC</a> Eb	<a href="#">DEC</a> Eb														
5	<a href="#">INC</a> Ev	<a href="#">DEC</a> Ev	<a href="#">CALL</a> Ev	<a href="#">CALL</a> eP	<a href="#">JMP</a> Ev	<a href="#">JMP</a> Ep	<a href="#">PUSH</a> Ev									

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

Group	<table><tr><td colspan="2">mod</td><td colspan="2">nnn</td><td colspan="2">R/M</td><td colspan="2"></td></tr></table>								mod		nnn		R/M			
	mod		nnn		R/M											
000	001	010	011	100	101	110	111									
6	<a href="#">SLDT</a> Ew	<a href="#">STR</a> Ew	<a href="#">LLDT</a> Ew	<a href="#">LTR</a> Ew	<a href="#">VERR</a> Ew	<a href="#">VERW</a> Ew										
7	<a href="#">SGDT</a> Ms	<a href="#">SIDT</a> Ms	<a href="#">LGDT</a> Ms	<a href="#">LIDT</a> Ms	<a href="#">SMSW</a> Ew		<a href="#">LMSW</a> Ew									
8					<a href="#">BT</a>	<a href="#">BTS</a>	<a href="#">BTR</a>	<a href="#">BTC</a>								

图 4.5: Opcodes determined by bits 5,4,3 of modR/M byte

因此对 opcode\_table 的修改如下:

```

mov
1
2
3  /* 0xb0 */ IDEXW(mov_I2r, mov, 1), IDEXW(mov_I2r, mov, 1), IDEXW(mov_I2r, mov,
4  1), IDEXW(mov_I2r, mov, 1),
5  /* 0xb4 */ IDEXW(mov_I2r, mov, 1), IDEXW(mov_I2r, mov, 1), IDEXW(mov_I2r, mov,
6  1), IDEXW(mov_I2r, mov, 1),
7  /* 0xb8 */ IDEX(mov_I2r, mov), IDEX(mov_I2r, mov), IDEX(mov_I2r, mov),
8  IDEX(mov_I2r, mov),
9  /* 0xbc */ IDEX(mov_I2r, mov), IDEX(mov_I2r, mov), IDEX(mov_I2r, mov),
10 IDEX(mov_I2r, mov),
11 /* 0xc0 */ IDEXW(gp2_Ib2E, gp2, 1), IDEX(gp2_Ib2E, gp2), EMPTY, EX(ret),
12 /* 0xc4 */ EMPTY, EMPTY, IDEXW(mov_I2E, mov, 1), IDEX(mov_I2E, mov),

```

leave

```

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

```

cld

```
1      /* 0x98 */      EX(cwt1), EX(cltd), EMPTY, EMPTY,
```

movsx

```
1      /* 0xbc */ EMPTY, EMPTY, IDEXW(mov_E2G, movsx, 1), IDEXW(mov_E2G, movsx, 2),
```

movzx

```
1      /* 0xb4 */ EMPTY, EMPTY, IDEXW(mov_E2G, movzx, 1), IDEXW(mov_E2G, movzx, 2),
```

add

```
1      /* 0x00 */ IDEXW(G2E, add, 1), IDEX(G2E, add), IDEXW(E2G, add, 1), IDEX(E2G, add),
```

inc

```
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),
```

sub

```
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,
```

dec

```
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),
```

cmp

```
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,
```

adc

```
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,
```

sbb

```
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,
```

## imul2

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

## and

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

## or

```
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),
```

## setcc

```
1  /* 0x90 */  IDEXW(E, setcc, 1), IDEXW(E, setcc, 1), IDEXW(E, setcc, 1), IDEXW(E,
setcc, 1),
2  /* 0x94 */  IDEXW(E, setcc, 1), IDEXW(E, setcc, 1), IDEXW(E, setcc, 1),
IDEXW(E, setcc, 1),
3  /* 0x98 */  IDEXW(E, setcc, 1), IDEXW(E, setcc, 1), IDEXW(E, setcc, 1),
IDEXW(E, setcc, 1),
4  /* 0x9c */  IDEXW(E, setcc, 1), IDEXW(E, setcc, 1), IDEXW(E, setcc, 1),
IDEXW(E, setcc, 1),
```

## test

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

## jmp

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

## jcc

```
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),
```

## lea

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

## nop

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

## cld

```
1  /* 0x98 */ EX(cwtl), EX(cld), EMPTY, EMPTY,
```

使用 RTL 实现正确的执行函数首先进入 rtl.h，需要实现的框架函数都已经给出，并且内部都有注释，只需要根据注释填充对应内容即可，一些读取和写入操作需要使用 rtl.h 内部已经定义写好的 rtl 函数，具体实现如下：

## rtl 基本函数

```
1  static inline void rtl_mv(rtlreg_t *dest, const rtlreg_t *src1) {
2      // dest <- src1 xfg
3      *dest = *src1;
4  }
5
6  static inline void rtl_not(rtlreg_t *dest) {
7      // dest <- ~dest xfg
8      *dest = ~*dest;
9  }
10
11 static inline void rtl_sext(rtlreg_t *dest, const rtlreg_t *src1, int width) {
12     // dest <- signext(src1[(width * 8 - 1) .. 0]) xfg
13     int32_t dm = (int32_t) *src1;
14     dm <<= 32 - (8 * width);
15     dm >>= 32 - (8 * width);
16     *dest = dm;
17 }
18
19 static inline void rtl_eqi(rtlreg_t *dest, const rtlreg_t *src1, int imm) {
20     // dest <- (src1 == imm ? 1 : 0) xfg
21     *dest = (*src1 == imm);
22 }
23
24 static inline void rtl_neq0(rtlreg_t *dest, const rtlreg_t *src1) {
25     // dest <- (src1 != 0 ? 1 : 0) xfg
26     *dest = (*src1 != 0);
27 }
```

在 all-instr.h 中声明不同指令对应的 make\_EHlper 函数

## make\_EHlper 声明

```
1  make_EHlper(lea);
2  make_EHlper(and);
3  make_EHlper(nop);
4  make_EHlper(add);
5  make_EHlper(cmp);
6  make_EHlper(setcc);
7  make_EHlper(movzx);
8  make_EHlper(test);
9  make_EHlper(jcc);
```

```

10 make_EHelper(adc);
11 make_EHelper(or);
12 make_EHelper(shl);
13 make_EHelper(sar);
14 make_EHelper(shr);
15 make_EHelper(dec);
16 make_EHelper(inc);
17 make_EHelper(not);
18 make_EHelper(jmp);
19 make_EHelper(mul);
20 make_EHelper(imul1);
21 make_EHelper(imul2);
22 make_EHelper(movsx);
23 make_EHelper(leave);
24 make_EHelper(call_rm);
25 make_EHelper(jmp_rm);
26 make_EHelper(sbb);
27 make_EHelper(cwtl);
28 make_EHelper(cltd);
29 make_EHelper(div);
30 make_EHelper(idiv);

```

实现不同指令的 make\_EHelper 函数体

#### make\_EHlper 实现

```

1 //leave
2 //所在位置: nemu/src/cpu/exec/data-mov.c
3 //实现思路: 利用rtl_mv和rtl_pop实现leave指令
4 make_EHelper(leave) {
5     rtl_mv(&cpu.esp,&cpu.ebp);
6     rtl_pop(&cpu.ebp);
7
8     print_asm("leave");
9 }
10
11 //cltd
12 //所在位置: nemu/src/cpu/exec/data-mov.c
13 //实现思路: 首先判断decoding.is_operand_size_16的值, 根据不同的两种情况 (0和1)
14 //使用已经完成的
15 //rtl_lr_w、rtl_sext、rtl_sari、rtl_sr_w、rtl_lr_l、rtl_sr_l实现cltd
16
17 make_EHelper(cltd) {
18     if (decoding.is_operand_size_16) {
19         rtl_lr_w(&t0, R_AX);
20         rtl_sext(&t0, &t0, 2);
21         rtl_sari(&t0, &t0, 16);
22         rtl_sr_w(R_DX, &t0);
23     } else {
24         rtl_lr_l(&t0, R_EAX);

```



```

25         rtl_sari(&t0, &t0, 31);
26         rtl_sari(&t0, &t0, 1);
27         rtl_sr_l(R_EDX, &t0);
28     }
29
30     print_asm(decoding.is_operand_size_16 ? "cwtl" : "cltd");
31 }
32
33 //cwtl
34 //所在位置: nemu/src/cpu/exec/data-mov.c
35 //实现思路: 类似cltd的实现方法, 根据decoding.is_operand_size_16值的不同使用已经完成的
36 //rtl_lr_b、rtl_sext、rtl_sr_w、rtl_sr_l实现cwtl
37 make_EHelper(cwtl) {
38     if (decoding.is_operand_size_16) {
39         rtl_lr_b(&t0, R_AX);
40         rtl_sext(&t0, &t0, 1);
41         rtl_sr_w(R_AX, &t0);
42     }
43     else {
44         rtl_lr_w(&t0, R_AX);
45         rtl_sext(&t0, &t0, 2);
46         rtl_sr_l(R_EAX, &t0);
47     }
48
49     print_asm(decoding.is_operand_size_16 ? "cbtw" : "cwtl");
50 }
51
52 //call_rm
53 //所在位置: /home/dmrf/PA/ics2017/nemu/src/cpu/exec/control.c
54 //实现思路: 首先给is_jump赋值1, 然后给jmp_eip赋值id_dest->val,
55 //最后push&decoding.seq_eip
56 make_EHelper(call_rm) {
57     decoding.is_jump = 1;
58     decoding.jmp_eip = id_dest->val;
59     rtl_push(&decoding.seq_eip);
60
61     print_asm("call *%s", id_dest->str);
62 }
63
64 //test
65 //所在位置: /home/dmrf/PA/ics2017/nemu/src/cpu/exec/logic.c
66 //实现思路: 利用rtl_and、rtl_update_ZFSF、rtl_set_CF、rtl_set_OF实现test
67 make_EHelper(test) {
68     rtl_and(&t0, &id_dest->val, &id_src->val);
69     rtl_update_ZFSF(&t0, id_dest->width);
70     rtl_set_CF(&tzero);
71     rtl_set_OF(&tzero);
72
73     print_asm_template2(test);

```

```

74 }
75
76 //and
77 //所在位置: /home/dmrf/PA/ics2017/nemu/src/cpu/exec/logic.c
78 //实现思路: 使用rtl_and、rtl_update_ZFSF、rtl_set_OF、rtl_set_CF实现and
79 make_EHelper(and) {
80     rtl_and(&t0, &id_dest->val, &id_src->val);
81     operand_write(id_dest, &t0);
82
83     rtl_update_ZFSF(&t0, id_dest->width);
84     rtl_set_OF(&tzero);
85     rtl_set_CF(&tzero);
86
87     print_asm_template2(and);
88 }
89
90 //or
91 //所在位置: /home/dmrf/PA/ics2017/nemu/src/cpu/exec/logic.c
92 //实现思路: 使用rtl_or、operand_write、rtl_update_ZFSF、rtl_set_OF、rtl_set_CF实现or
93 make_EHelper (or) {
94     rtl_or(&t2, &id_dest->val, &id_src->val);
95     printf("id_dest->val:%x\n", id_dest->val);
96     printf("id_src->val:%x,t2:%d\n", id_src->val, t2);
97
98     operand_write(id_dest, &t2);
99     rtl_update_ZFSF(&t2, id_dest->width);
100     rtl_set_OF(&tzero);
101     rtl_set_CF(&tzero);
102
103     print_asm_template2(or);
104 }
105
106
107 //sar
108 //所在位置: /home/dmrf/PA/ics2017/nemu/src/cpu/exec/logic.c
109 //实现思路: 使用rtl_sar、operand_write、rtl_update_ZFSF、print_asm_template2实现sar
110
111 make_EHelper (sar) {
112     rtl_sar(&id_dest->val, &id_dest->val, &id_src->val);
113     operand_write(id_dest, &id_dest->val);
114     rtl_update_ZFSF(&id_dest->val, id_dest->width);
115     // unnecessary to update CF and OF in NEMU
116
117     print_asm_template2(sar);
118 }
119
120 //shl
121 //所在位置: /home/dmrf/PA/ics2017/nemu/src/cpu/exec/logic.c
122 //实现思路: 使用已经写好的rtl_shl和operand_write实现shl, 最后注意update_ZFSF

```

```

123 make_EHelper (shl) {
124     rtl_shl(&id_dest->val, &id_dest->val, &id_src->val);
125     operand_write(id_dest, &id_dest->val);
126     rtl_update_ZFSF(&id_dest->val, id_dest->width);
127     // unnecessary to update CF and OF in NEMU
128
129     print_asm_template2(shl);
130 }
131
132 //shr
133 //所在位置: /home/dmrf/PA/ics2017/nemu/src/cpu/exec/logic.c
134 //实现思路: 使用已经写好的rtl_shr和operand_write实现shr, 最后注意update_ZFSF
135 make_EHelper (shr) {
136     rtl_shr(&id_dest->val, &id_dest->val, &id_src->val);
137     operand_write(id_dest, &id_dest->val);
138     rtl_update_ZFSF(&id_dest->val, id_dest->width);
139     // unnecessary to update CF and OF in NEMU
140
141     print_asm_template2(shr);
142 }
143
144 //not
145 //所在位置: /home/dmrf/PA/ics2017/nemu/src/cpu/exec/logic.c
146 //实现思路: 使用已经写好的rtl_mv、rtl_not和operand_write实现not
147 make_EHelper (not) {
148     rtl_mv(&t0, &id_dest->val);
149     rtl_not(&t0);
150     operand_write(id_dest, &t0);
151
152     print_asm_template1(not);
153 }
154
155 //add
156 //所在位置: /home/dmrf/PA/ics2017/nemu/src/cpu/exec/arith.c
157 //实现思路: 使用rtl_add和operand_write实现add, 注意最后update_ZFSF
158 make_EHelper(add) {
159     rtl_add(&t2, &id_dest->val, &id_src->val);
160     //rtl_get_CF(&t1);
161     operand_write(id_dest, &t2);
162     rtl_update_ZFSF(&t2, id_dest->width);
163
164     rtl_sltu(&t0, &t2, &id_dest->val);
165
166     rtl_set_CF(&t0);
167
168     rtl_xor(&t0, &id_dest->val, &id_src->val);
169     rtl_not(&t0);
170     rtl_xor(&t1, &id_dest->val, &t2);
171     rtl_and(&t0, &t0, &t1);

```

```

172     rtl_msb(&t0, &t0, id_dest->width);
173     rtl_set_OF(&t0);
174
175     print_asm_template2(add);
176 }
177
178 //cmp
179 //所在位置: /home/dmrf/PA/ics2017/nemu/src/cpu/exec/arith.c
180 make_EHelper(cmp) {
181     rtl_sub(&t2, &id_dest->val, &id_src->val);
182     rtl_update_ZFSF(&t2, id_dest->width);
183     rtl_sltu(&t0, &id_dest->val, &t2);
184
185     rtl_set_CF(&t0);
186
187     rtl_xor(&t0, &id_dest->val, &id_src->val);
188     rtl_xor(&t1, &id_dest->val, &t2);
189     rtl_and(&t0, &t0, &t1);
190     rtl_msb(&t0, &t0, id_dest->width);
191     rtl_set_OF(&t0);
192
193     print_asm_template2(cmp);
194 }
195
196 //inc
197 //所在位置: /home/dmrf/PA/ics2017/nemu/src/cpu/exec/arith.c
198 //实现思路: 使用rtl_addi和operand_write实现inc, 最后注意update_ZFSF
199 make_EHelper(inc) {
200     rtl_addi(&t2, &id_dest->val, 1);
201
202     operand_write(id_dest, &t2);
203     rtl_update_ZFSF(&t2, id_dest->width);
204
205
206
207     rtl_xori(&t0, &id_dest->val, 1);
208     rtl_not(&t0);
209     rtl_xor(&t1, &id_dest->val, &t2);
210     rtl_and(&t0, &t0, &t1);
211     rtl_msb(&t0, &t0, id_dest->width);
212     rtl_set_OF(&t0);
213
214
215     print_asm_template1(inc);
216 }
217
218
219 //dec
220 make_EHelper(dec) {

```

```

221     rtl_subi(&t2, &id_dest->val, 1);
222
223     operand_write(id_dest, &t2);
224     rtl_update_ZFSF(&t2, id_dest->width);
225
226
227     rtl_xor(&t0, &id_dest->val, &id_src->val);
228     rtl_xor(&t1, &id_dest->val, &t2);
229     rtl_and(&t0, &t0, &t1);
230     rtl_msb(&t0, &t0, id_dest->width);
231     rtl_set_OF(&t0);
232
233     print_asm_template1(dec);
234 }
235
236 //neg
237 //所在位置: /home/dmrf/PA/ics2017/nemu/src/cpu/exec/arith.c
238 make_EHelper(neg) {
239     if(!id_dest->val){
240         rtl_set_CF(&tzero);
241     }else{
242         rtl_addi(&t0,&tzero,1);
243         rtl_set_CF(&t0);
244     }
245     rtl_add(&t0,&tzero,&id_dest->val);
246     t0=-t0;
247     operand_write(id_dest,&t0);
248     rtl_update_ZFSF(&t2, id_dest->width);
249     rtl_xor(&t0, &id_dest->val, &id_src->val);
250     rtl_xor(&t1, &id_dest->val, &t2);
251     rtl_and(&t0, &t0, &t1);
252     rtl_msb(&t0, &t0, id_dest->width);
253     rtl_set_OF(&t0);
254
255     print_asm_template1(neg);
256 }

```

## EFLAGS 设置

```

1 void rtl_setcc(rtlreg_t* dest, uint8_t subcode) {
2     bool invert = subcode & 0x1;
3     enum {
4         CC_O, CC_NO, CC_B,  CC_NB,
5         CC_E, CC_NE, CC_BE, CC_NBE,
6         CC_S, CC_NS, CC_P,  CC_NP,
7         CC_L, CC_NL, CC_LE, CC_NLE
8     };
9
10    // TODO: Query EFLAGS to determine whether the condition code is satisfied.

```

```

11 // dest <- ( cc is satisfied ? 1 : 0)
12 switch (subcode & 0xe) {
13     case CC_O:
14         rtl_get_OF(dest);
15         break;
16     case CC_B:
17         rtl_get_CF(dest);
18         break;
19     case CC_E:
20         rtl_get_ZF(dest);
21         break;
22     case CC_BE:
23         assert(dest!=&t0);
24         rtl_get_CF(dest);
25         rtl_get_ZF(&t0);
26         rtl_or(dest,dest,&t0);
27         break;
28     case CC_S:
29         rtl_get_SF(dest);
30         break;
31     case CC_L:
32         assert(dest!=&t0);
33         rtl_get_SF(dest);
34         rtl_get_OF(&t0);
35         rtl_xor(dest,dest,&t0);
36         break;
37     case CC_LE:
38         assert(dest!=&t0);
39         rtl_get_SF(dest);
40         rtl_get_OF(&t0);
41         rtl_xor(dest,dest,&t0);
42         rtl_get_ZF(&t0);
43         rtl_or(dest,dest,&t0);
44         break;
45     default:
46         panic("should not reach here");
47     case CC_P:
48         panic("n86 does not have PF");
49 }
50
51 if (invert) {
52     rtl_xori(dest, dest, 0x1);
53 }
54 }

```

## 4.2 实现 differential testing

在上述实现过程中，由于代码量巨大，所以需要利用好调试工具，根据实验报告内容，需要先实现 `difftest_step()` 函数，添加代码如下：

`difftest_step()` 函数

```

1  if(r.eax!=cpu.eax) {
2      printf("expect: %d true: %d at: %x\n", r.eax, cpu.eax, cpu.eip);
3      diff=true;
4  }
5  if(r.ecx!=cpu.ecx) {
6      printf("expect: %d true: %d at: %x\n", r.ecx, cpu.ecx, cpu.eip);
7      diff=true;
8  }
9  if(r.edx!=cpu.edx) {
10     printf("expect: %d true: %d at: %x\n", r.edx, cpu.edx, cpu.eip);
11     diff=true;
12 }
13 if(r.ebx!=cpu.ebx) {
14     printf("expect: %d true: %d at: %x\n", r.ebx, cpu.ebx, cpu.eip);
15     diff=true;
16 }
17 if(r.esp!=cpu.esp) {
18     printf("expect: %d true: %d at: %x\n", r.esp, cpu.esp, cpu.eip);
19     diff=true;
20 }
21 if(r.ebp!=cpu.ebp) {
22     printf("expect: %d true: %d at: %x\n", r.ebp, cpu.ebp, cpu.eip);
23     diff=true;
24 }
25 if(r.esi!=cpu.esi) {
26     printf("expect: %d true: %d at: %x\n", r.esi, cpu.esi, cpu.eip);
27     diff=true;
28 }
29 if(r.edi!=cpu.edi) {
30     printf("expect: %d true: %d at: %x\n", r.edi, cpu.edi, cpu.eip);
31     diff=true;
32 }
33 if(r.eip!=cpu.eip) {
34     diff=true;
35     Log("different: qemu.eip=0x%x,nemu.eip=0x%x",r.eip,cpu.eip);
36 }

```

实现改代码的主要思路就是检测寄存器至，把 NEMU 的 8 个通用寄存器和 `eip` 与从 QEMU 中读出的寄存器的值进行比较，如果发现值不一样，就输出相应的提示信息。定义 `DIFF_TEST`，可以与 QEMU 相连，然后 `make run` 就可以看到提示 `Connect to QEMU successfully`：

```

add
gyk@ubuntu:~/ics2017/nexus-am/tests/cputest$ make ALL=add run
Building add [x86-nemu]
Building am [x86-nemu]
make[2]: *** No targets specified and no makefile found. Stop.
+ CC src/monitor/diff-test/diff-test.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/gyk/ics2017/nexus-am/tests/cputest/build/add-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 08:58:32, Apr 17 2023
For help, type "help"
(nemu) c
nemu: HIT GOOD TRAP at eip = 0x00100027
(nemu) c

```

图 4.6: QEMU 连接成功

利用上述工具实现我的整个代码的编写过程，需要的指令都实现之后，进行整体测试，可以看到结果如下：

```

[3]+ Stopped bash runall.sh
gyk@ubuntu:~/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!
gyk@ubuntu:~/ics2017/nemu$

```

图 4.7: cputest 结果

## 5 阶段三

想要加入 IOE，首先需要定义宏 HAS\_IOE

### 5.1 串口

这里主要是实现 in 和 out 指令  
完成 opcode\_table

opcode\_table

```

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

```



```
2 /* 0xec */ IDEXW(in_dx2a, in, 1), IDEX(in_dx2a, in), IDEXW(out_a2dx, out, 1),  
    IDEX(out_a2dx, out),
```

完成 `make_EHelper` 函数，在 `system.c`

in/out 函数

```
1 make_EHelper(in) {  
2     t1 = pio_read(id_src->val, id_dest->width);  
3     operand_write(id_dest, &t1);  
4  
5     print_asm_template2(in);  
6  
7     #ifdef DIFF_TEST  
8         diff_test_skip_qemu();  
9     #endif  
10 }  
11 make_EHelper(out) {  
12     pio_write(id_dest->val, id_dest->width, id_src->val);  
13  
14     print_asm_template2(out);  
15  
16     print_asm_template2(out);  
17  
18     #ifdef DIFF_TEST  
19         diff_test_skip_qemu();  
20     #endif  
21 }
```

实现之后，运行 `helloworld`，可以看到结果如下

```
macro in PA2 */
```

```
gyk@ubuntu: ~/ics2017/nexus-am/apps/hello
+ CC src/cpu/reg.c
+ CC src/cpu/intr.c
+ LD build/nemu
./build/nemu -l /home/gyk/ics2017/nexus-am/apps/hello/build/nemu-log.txt /home/gyk/ics2017/nexus-am/apps/hello/build/hello-x86-nemu.bin
[src/monitor/monitor.c,65,load_img] The image is /home/gyk/ics2017/nexus-am/apps/hello/build/hello-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 09:27:18, Apr 17 2023
For help, type "help"
(nemu) c
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
nemu: HIT GOOD TRAP at eip = 0x0010006e
(nemu) 
```

图 5.8: 串口实现结果

## 5.2 时钟

实现 `__up_timer()`:

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

运行 `timetest` 可以看到结果如下图所示:

```
sec);
sec) gyk@ubuntu: ~/ics2017/nexus-am/tests/timetest
s/timetest/build/timetest-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 09:27:18, Apr 17 2023
For help, type "help"
(nemu) c
1 second.
2 seconds.
3 seconds.
4 seconds.
5 seconds.
6 seconds.
7 seconds.
8 seconds.
9 seconds.
10 seconds.
11 seconds.
12 seconds.
13 seconds.
14 seconds.
15 seconds.
16 seconds.
17 seconds.
18 seconds.
19 seconds.
```

图 5.9: 时钟实现

看看 NEMU 跑多快，跑分截图如下：

```
gyk@ubuntu: ~/ics2017/nexus-am/apps/dhrystone
+ CC src/cpu/exec/data-mov.c
+ CC src/cpu/exec/logic.c
+ CC src/cpu/exec/prefix.c
+ CC src/cpu/exec/special.c
+ CC src/cpu/reg.c
+ CC src/cpu/intr.c
+ LD build/nemu
./build/nemu -l /home/gyk/ics2017/nexus-am/apps/dhrystone/build/nemu-log.txt /home/gyk/ics2017/nexus-am/apps/dhrystone/build/dhrystone-x86-nemu.bin
[src/monitor/monitor.c,65,load_img] The image is /home/gyk/ics2017/nexus-am/apps/dhrystone/build/dhrystone-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 09:43:28, Apr 17 2023
For help, type "help"
(nemu) c
Dhrystone Benchmark, Version C, Version 2.2
Trying 500000 runs through Dhrystone.
Finished in 16902 ms
=====
Dhrystone PASS      60 Marks
                    vs. 100000 Marks (i7-6700 @ 3.40GHz)
nemu: HIT GOOD TRAP at eip = 0x0010006e
(nemu)
```

图 5.10: Dhrystone 跑分

```

gyk@ubuntu: ~/ics2017/nexus-am/apps/coremark
[src/monitor/monitor.c,65,load_img] The image is /home/gyk/ics2017/nexus-am/apps/coremark/build/coremark-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 09:43:28, Apr 17 2023
For help, type "help"
(nemu) c
Running CoreMark for 1000 iterations
2K performance run parameters for coremark.
CoreMark Size      : 666
Total time (ms)    : 18982
Iterations         : 1000
Compiler version   : GCC5.4.0 20160609
seedcrc           : 0xe9f5
[0]crclist        : 0xe714
[0]crcmatrix      : 0x1fd7
[0]crcstate       : 0x8e3a
[0]crcfinal       : 0xd340
Finised in 18982 ms.
=====
CoreMark PASS      235 Marks
                    vs. 100000 Marks (i7-6700 @ 3.40GHz)
nemu: HIT GOOD TRAP at eip = 0x0010006e
(nemu)

```

图 5.11: Coremark 跑分

```

make[1]: Entering directory '/home/gyk/ics2017/nemu'
./build/nemu -l /home/gyk/ics2017/nexus-am/apps/microbench/build/nemu-log.txt /home/gyk/ics2017/nexus-am/apps/microbench/build/microbench-x86-nemu.bin
[src/monitor/monitor.c,65,load_img] The image is /home/gyk/ics2017/nexus-am/apps/microbench/build/microbench-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 09:43:28, Apr 17 2023
For help, type "help"
(nemu) c
[qsrt] Quick sort: * Passed.
min time: 959 ms [575]
[queen] Queen placements: * Passed.
min time: 1733 ms [297]
[bf] Brainf**k interpreter: * Passed.
min time: 10029 ms [261]
[flb] Fibonacci number: * Passed.
min time: 22312 ms [128]
[sieve] Eratosthenes sieve: * Passed.
min time: 19303 ms [218]
[ispz] A* 15-puzzle search: * Passed.
min time: 3932 ms [147]
[dinic] Dinic's maxflow algorithm: * Passed.
min time: 3301 ms [410]
[lzip] Lzip compression: * Passed.
min time: 10125 ms [261]
[ssort] Suffix sort: * Passed.
min time: 1952 ms [303]
[md5] MD5 digest: * Passed.
min time: 18593 ms [105]
=====
MicroBench PASS    270 Marks
                    vs. 100000 Marks (i7-6700 @ 3.40GHz)
nemu: HIT GOOD TRAP at eip = 0x00100032
(nemu)

```

图 5.12: microbench 跑分

### 5.3 键盘

实现 `_read_key()`:

```

1 int _read_key() {
2     uint32_t key_code = _KEY_NONE;
3
4     if (inb(0x64) & 0x1)
5         key_code = inl(0x60);
6
7     return key_code;
8 }

```

运行 keytest:

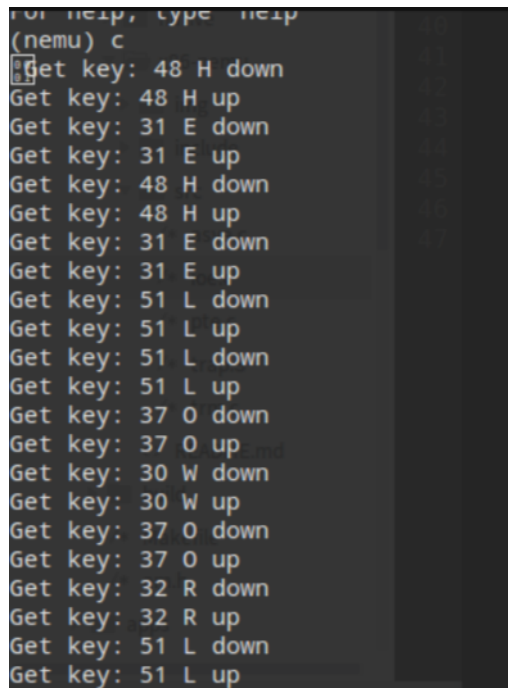


图 5.13: keytest

## 5.4 VGA

实现 `paddr_read()` 和 `paddr_write()`:

```

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

```

运行 `videotest`:

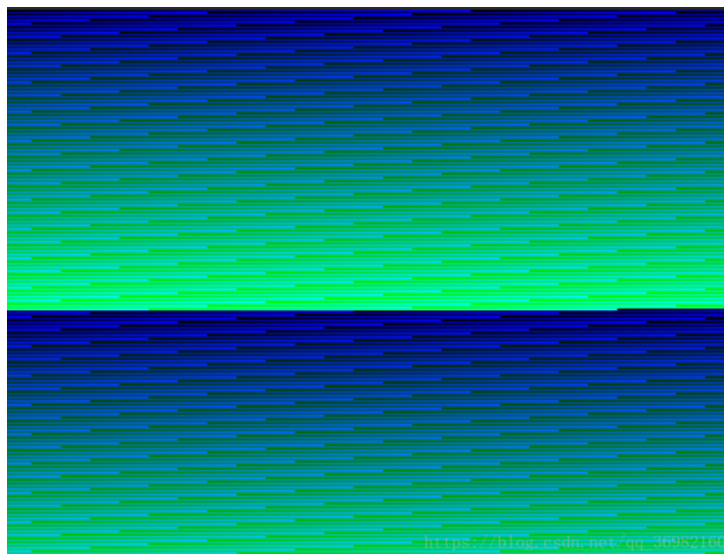


图 5.14: vidoetest 结果

实现 `_draw_rect()`

```
1 void _draw_rect(const uint32_t *pixels, int x, int y, int w, int h) {  
2     int c, r;  
3     for (r = y; r < y + h; r++)  
4         for (c = x; c < x + w; c++)  
5             fb[c+r*_screen.width] = pixels[(r-y)*w+(c-x)];  
6 }
```

运行 vidoetest

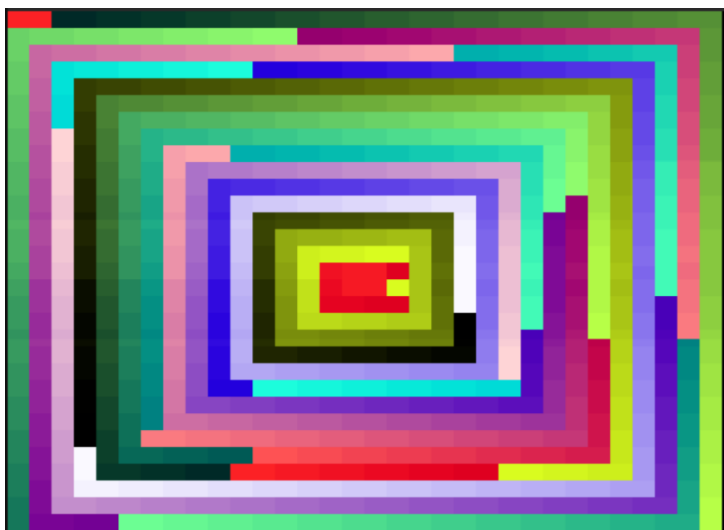


图 5.15: vidoetest 结果

## 6 必答题

### 6.1 问题一

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：

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

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？

问题的根源是不同的 CPU 架构在解释一个内存地址所存储的多字节数据时采用的字节序可能不同。x86 架构采用的是小端序（Little Endian），而 Motorola 68k 等架构采用的是大端序（Big Endian）。所谓大端序和小端序，是指在多字节数据的存储中，高位字节和低位字节的存储顺序。

在 NEMU 运行在 Motorola 68k 的机器上的情况下，如果直接将内存中的多字节数据读取到变量中，由于两种架构的字节序不同，读取到的值会与预期值不同，从而导致程序出现错误。

为了解决这个问题，我们需要在访问多字节数据时进行字节序的转换。对于 x86 架构来说，不需要进行转换；而对于 Motorola 68k 等架构来说，需要将读取到的数据按照大端序转换成小端序才能得到正确的结果。

在 NEMU 的实现中，针对字节序问题，实现了以下两个函数：

`uint32_t instr_fetch(swaddr_t addr, size_t len)`：用于从内存中读取 `len` 字节数据，其中参数 `addr` 表示要读取的数据在内存中的地址。由于 NEMU 的源代码是在 x86 架构上编译的，因此在读取数据时无需进行字节序转换。

`uint32_t instr_fetch_sign(swaddr_t addr, size_t len)`：与 `instr_fetch()` 类似，不同之处在于它读取的是有符号的数据，并且需要进行字节序转换。

在具体实现时，可以通过一些宏定义和条件编译来实现针对不同架构的适配。例如，在 NEMU 的代码中，可以定义宏 `BIG_ENDIAN` 来表示大端序，在读取数据时判断当前架构是否为大端序，如果是，则需要进行字节序的转换。这样，NEMU 就可以在不同的架构上运行，而不会受到字节序的影响。

### 6.2 问题二

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？

和栈的内容没有放入可执行文件里面是因为堆和栈中数据的变化比较频繁，如果放进可执行文件中读取速度会变慢。堆和栈是程序运行时从内存中动态申请的。

### 6.3 问题三

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：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

感, 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?

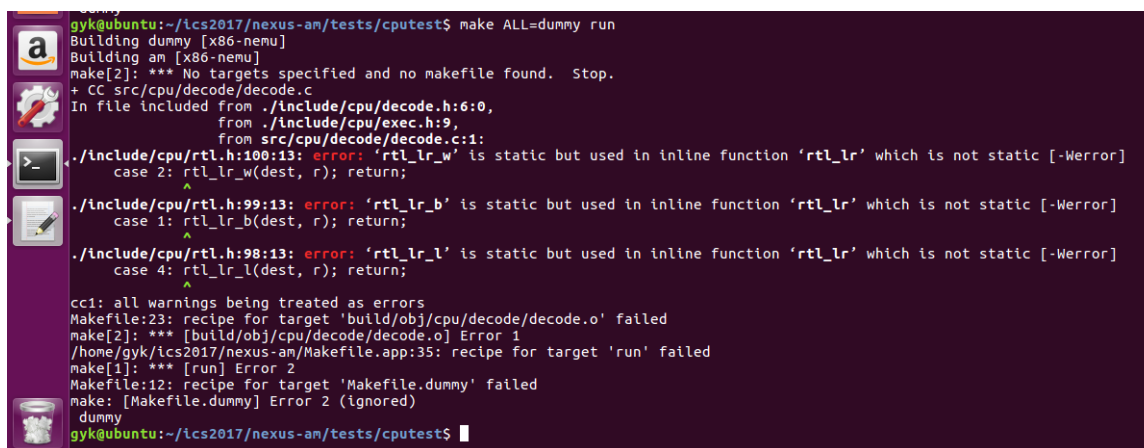
在 C 语言中, `volatile` 关键字用于声明一个变量是“易变”的, 即这个变量的值可能会被程序以外的因素改变, 例如硬件设备的寄存器。因此, 编译器不应该对这个变量的访问进行优化, 以免出现意料之外的结果。使用 `volatile` 关键字可以确保编译器不会对访问设备寄存器的代码进行优化, 因为这些寄存器的值可能会被外部因素改变。如果去掉 `volatile` 关键字, 编译器可能会将读写寄存器的代码进行优化, 导致程序出现错误, 因为它读取的值可能已经过时或不正确。

## 6.4 问题四

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?

选择函数 `rtl_lr`

去掉 `static` 关键字后, 如下图所示,



```

gyk@ubuntu:~/ics2017/nexus-am/tests/cputest$ make ALL=dummy run
Building dummy [x86-nemu]
Building am [x86-nemu]
make[2]: *** No targets specified and no makefile found. Stop.
+ CC src/cpu/decode/decode.c
In file included from ./include/cpu/decode.h:6:0,
                 from ./include/cpu/exec.h:9,
                 from src/cpu/decode/decode.c:1:
./include/cpu/rtl.h:100:13: error: 'rtl_lr_w' is static but used in inline function 'rtl_lr' which is not static [-Werror]
case 2: rtl_lr_w(dest, r); return;
               ^
./include/cpu/rtl.h:99:13: error: 'rtl_lr_b' is static but used in inline function 'rtl_lr' which is not static [-Werror]
case 1: rtl_lr_b(dest, r); return;
               ^
./include/cpu/rtl.h:98:13: error: 'rtl_lr_l' is static but used in inline function 'rtl_lr' which is not static [-Werror]
case 4: rtl_lr_l(dest, r); return;
               ^
cc1: all warnings being treated as errors
Makefile:23: recipe for target 'build/obj/cpu/decode/decode.o' failed
make[2]: *** [build/obj/cpu/decode/decode.o] Error 1
/home/gyk/ics2017/nexus-am/Makefile.app:35: recipe for target 'run' failed
make[1]: *** [run] Error 2
Makefile:12: recipe for target 'Makefile.dummy' failed
make: [Makefile.dummy] Error 2 (ignored)
dummy
gyk@ubuntu:~/ics2017/nexus-am/tests/cputest$

```

图 6.16: 去掉 `static`

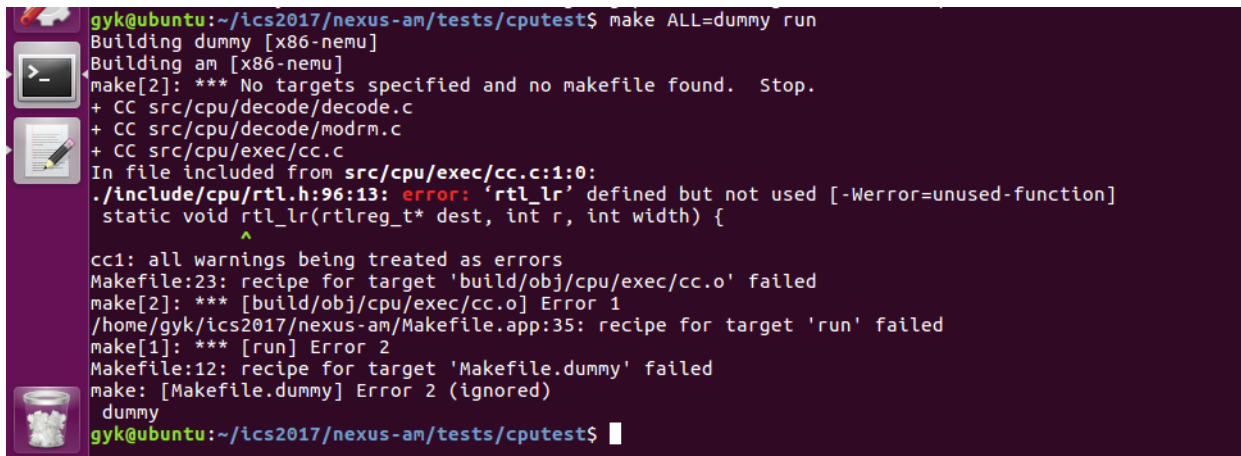
从结果中可以看出涉及到 `rtl_lr` 的指令发生错误

- `'rtl_lr_w' is static but used in inline function 'rtl_lr' which is not static [-Werror]`

这个错误是由于在 `inline` 函数 `rtl_lr()` 中调用了另一个静态函数 `rtl_lr_w()`, 但是 `rtl_lr_w()` 函数被声明为静态函数, 只能在本文件中使用, 不能跨文件调用。由于 `rtl_lr()` 是 `inline` 函数, 编译器会将其内联展开, 从而会将调用 `rtl_lr_w()` 的代码直接插入到 `rtl_lr()` 的代码中, 导致调用了一个不能跨文件访问的静态函数, 从而编译器会报出这个错误。

去掉 `inline` 关键字后, 如下图所示,





```

gyk@ubuntu:~/ics2017/nexus-am/tests/cputest$ make ALL=dummy run
Building dummy [x86-nemu]
Building am [x86-nemu]
make[2]: *** No targets specified and no makefile found. Stop.
+ CC src/cpu/decode/decode.c
+ CC src/cpu/decode/modrm.c
+ CC src/cpu/exec/cc.c
In file included from src/cpu/exec/cc.c:1:0:
./include/cpu/rtl.h:96:13: error: 'rtl_lr' defined but not used [-Werror=unused-function]
static void rtl_lr(rtlreg_t* dest, int r, int width) {
^
cc1: all warnings being treated as errors
Makefile:23: recipe for target 'build/obj/cpu/exec/cc.o' failed
make[2]: *** [build/obj/cpu/exec/cc.o] Error 1
/home/gyk/ics2017/nexus-am/Makefile.app:35: recipe for target 'run' failed
make[1]: *** [run] Error 2
Makefile:12: recipe for target 'Makefile.dummy' failed
make: [Makefile.dummy] Error 2 (ignored)
dummy
gyk@ubuntu:~/ics2017/nexus-am/tests/cputest$

```

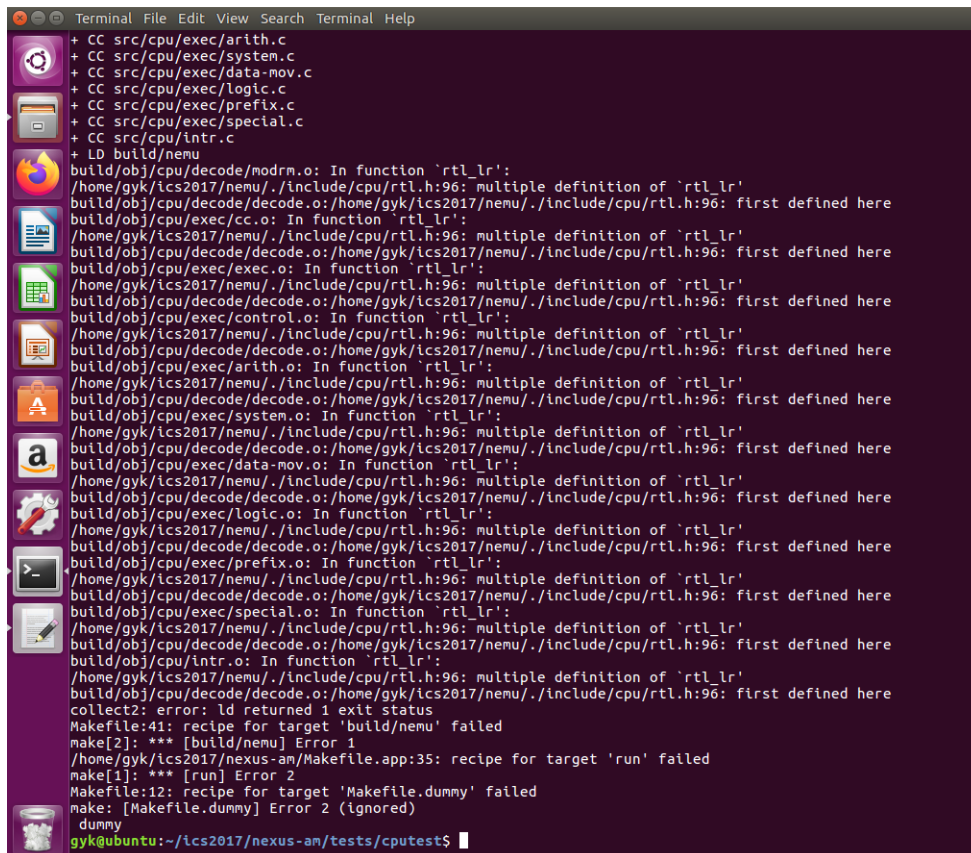
图 6.17: 去掉 inline

从结果中可以看出涉及到 `rtl_lr` 的指令发生错误

- `/include/cpu/rtl.h:96:13: error: 'rtl_lr' defined but not used [-Werror=unused-function]`

`rtl_lr` 函数并没有被使用。编译器在编译代码时启用了 `-Werror=unused-function` 选项，该选项会将未使用的函数视为错误，从而报出这个错误。

去掉 `inline` 和 `static` 关键字后，如下图所示，



```

Terminal File Edit View Search Terminal Help
+ CC src/cpu/exec/arith.c
+ CC src/cpu/exec/system.c
+ CC src/cpu/exec/data-mov.c
+ CC src/cpu/exec/logic.c
+ CC src/cpu/exec/prefix.c
+ CC src/cpu/exec/special.c
+ CC src/cpu/intr.c
+ LD build/nemu
build/obj/cpu/decode/modrm.o: In function 'rtl_lr':
/home/gyk/ics2017/nemu/.include/cpu/rtl.h:96: multiple definition of 'rtl_lr'
build/obj/cpu/decode/decode.o:/home/gyk/ics2017/nemu/.include/cpu/rtl.h:96: first defined here
build/obj/cpu/exec/cc.o: In function 'rtl_lr':
/home/gyk/ics2017/nemu/.include/cpu/rtl.h:96: multiple definition of 'rtl_lr'
build/obj/cpu/decode/decode.o:/home/gyk/ics2017/nemu/.include/cpu/rtl.h:96: first defined here
build/obj/cpu/exec/exec.o: In function 'rtl_lr':
/home/gyk/ics2017/nemu/.include/cpu/rtl.h:96: multiple definition of 'rtl_lr'
build/obj/cpu/decode/decode.o:/home/gyk/ics2017/nemu/.include/cpu/rtl.h:96: first defined here
build/obj/cpu/exec/control.o: In function 'rtl_lr':
/home/gyk/ics2017/nemu/.include/cpu/rtl.h:96: multiple definition of 'rtl_lr'
build/obj/cpu/decode/decode.o:/home/gyk/ics2017/nemu/.include/cpu/rtl.h:96: first defined here
build/obj/cpu/exec/arith.o: In function 'rtl_lr':
/home/gyk/ics2017/nemu/.include/cpu/rtl.h:96: multiple definition of 'rtl_lr'
build/obj/cpu/decode/decode.o:/home/gyk/ics2017/nemu/.include/cpu/rtl.h:96: first defined here
build/obj/cpu/exec/system.o: In function 'rtl_lr':
/home/gyk/ics2017/nemu/.include/cpu/rtl.h:96: multiple definition of 'rtl_lr'
build/obj/cpu/decode/decode.o:/home/gyk/ics2017/nemu/.include/cpu/rtl.h:96: first defined here
build/obj/cpu/exec/data-mov.o: In function 'rtl_lr':
/home/gyk/ics2017/nemu/.include/cpu/rtl.h:96: multiple definition of 'rtl_lr'
build/obj/cpu/decode/decode.o:/home/gyk/ics2017/nemu/.include/cpu/rtl.h:96: first defined here
build/obj/cpu/exec/logic.o: In function 'rtl_lr':
/home/gyk/ics2017/nemu/.include/cpu/rtl.h:96: multiple definition of 'rtl_lr'
build/obj/cpu/decode/decode.o:/home/gyk/ics2017/nemu/.include/cpu/rtl.h:96: first defined here
build/obj/cpu/exec/prefix.o: In function 'rtl_lr':
/home/gyk/ics2017/nemu/.include/cpu/rtl.h:96: multiple definition of 'rtl_lr'
build/obj/cpu/decode/decode.o:/home/gyk/ics2017/nemu/.include/cpu/rtl.h:96: first defined here
build/obj/cpu/exec/special.o: In function 'rtl_lr':
/home/gyk/ics2017/nemu/.include/cpu/rtl.h:96: multiple definition of 'rtl_lr'
build/obj/cpu/decode/decode.o:/home/gyk/ics2017/nemu/.include/cpu/rtl.h:96: first defined here
build/obj/cpu/intr.o: In function 'rtl_lr':
/home/gyk/ics2017/nemu/.include/cpu/rtl.h:96: multiple definition of 'rtl_lr'
build/obj/cpu/decode/decode.o:/home/gyk/ics2017/nemu/.include/cpu/rtl.h:96: first defined here
collect2: error: ld returned 1 exit status
Makefile:41: recipe for target 'build/nemu' failed
make[2]: *** [build/nemu] Error 1
/home/gyk/ics2017/nexus-am/Makefile.app:35: recipe for target 'run' failed
make[1]: *** [run] Error 2
Makefile:12: recipe for target 'Makefile.dummy' failed
make: [Makefile.dummy] Error 2 (ignored)
dummy
gyk@ubuntu:~/ics2017/nexus-am/tests/cputest$

```

图 6.18: 去掉 static 和 inline

从结果中可以看出涉及到 `rtl_lr` 的指令发生错误，会报 `multiple_definition`，错误，这个函数就

变成了一个普通的外部函数，可以被其他文件访问和调用。但是，由于该函数的实现在头文件中，如果多个文件都包含了这个头文件，那么链接器就会报告重复定义的错误。

## 6.5 问题五

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.

该 Makefile 中定义了以下变量和规则：

```
1 NAME: 目标文件名为 nemu。
2 INC_DIR: 头文件目录为 ./include。
3 BUILD_DIR: 生成文件目录为 ./build, 如果没有指定则默认为 ./build。
4 OBJ_DIR: 目标文件目录为 $(BUILD_DIR)/obj。
5 BINARY: 可执行文件名为 $(BUILD_DIR)/$(NAME)。
6 CC: 编译器为 gcc。
7 LD: 链接器为 gcc。
8 INCLUDES: 头文件参数为 -I./include。
9 CFLAGS: 编译选项为 -O2 -MMD -Wall -ggdb 和 $(INCLUDES)。
10 SRCS: 源文件为 src/*.c。
11 OBJS: 目标文件为$(OBJ_DIR)/*.o, 其中 *.o 对应 src/*.c。
12 $(OBJ_DIR)/%.o: src/%.c: 将每个 .c 文件编译成一个 .o 文件, 其中 $(OBJ_DIR)/%.o
    对应目标文件名, src/%.c 对应源文件名。
13 $(OBJS:.o=.d): 生成依赖关系文件, 用于检查头文件的修改。
14 .DEFAULT_GOAL = app: 默认目标为 app。
15 $(BINARY): $(OBJS): 将所有目标文件链接起来生成可执行文件 $(BINARY)。
16 run: 运行可执行文件。
17 gdb: 使用 gdb 调试可执行文件。
18 clean: 删除所有生成的文件。
```

根据该 Makefile 中的规则, make 程序会在 src/ 目录下找到所有后缀名为.c 的文件, 并将它们编译成目标文件(.o 文件)放在 build/obj/ 目录下。然后, 将所有目标文件链接起来生成可执行文件 build/nemu, 并链接必要的库文件 -lSDL2 和 -lreadline。