

计算机系统基础
Programming Assignment

PA 2 程序的执行（第一课） ——PA 2-1 指令解码与执行

2018年9月26日

调课通知

课 程 节 次	星 期 一	二	三	四	五
1--2 节	数字电路与数字系统 实验 (一) 基础实验楼乙 125	高级程序设计 仙 I -106	计算机系统基础 (单) (一) 仙 II -214 (二) 仙 II -215 数字电路与数字系统 (双) 仙 II -214	离散数学 仙 II -406	中国近现代史纲要 (一) 仙 II -112
3--4 节	计算机系统基础 (一) 仙 II -214 (二) 仙 II -215 数字电路与数字系统 仙 II -211	线性代数 (一) 仙 II -303 (二) 仙 II -505	数据结构 (一) 仙 II -214 (二) 仙 II -215	线性代数 (一) 仙 II -303 (二) 仙 II -505	中国近现代史纲要 (二) 仙 II -112 高级程序设计 双周, 基础实验楼乙 124 单周, 仙 I -106
5--6 节	数据结构 (一) 仙 II -214 (二) 仙 II -215	离散数学 仙 II -406 数字电路与数字系统 实验 (二) 基础 实验楼乙 125	程序设计基础实验 基础实验楼乙 126 计算机系统基础 (一) 仙 II -214	大学物理实验 实验楼乙 307、309、 314、316 丙 301-303、 305、307、308、317、 319、322	计算机系统基础 (二) 仙 II -215
7--8 节	10月10日		形势与政策 (上) (单) 仙 I -107	数据结构 (实验), 基 础实验楼乙 124、125	
9--10 节	网络攻防实验 基础实验楼乙 125		网络安全实验, 基础实验楼乙 206		

10月12日

二班：唐杰老师班

10月17日

一班：苏丰老师班

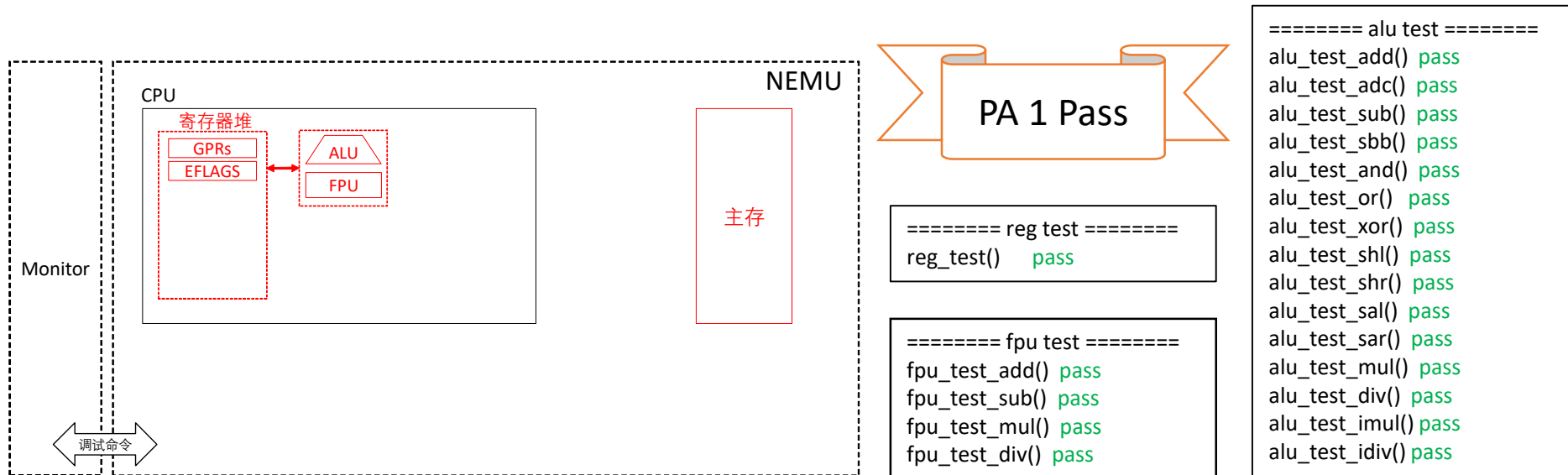
10月15日

课 程 节 次	星 期 一	二	三	四	五
1--2 节	数字电路与数字系统 实验 (一) 基础实验楼乙 125	高级程序设计 仙 I -106	计算机系统基础 (单) (一) 仙 II -214 (二) 仙 II -215 数字电路与数字系统 (双) 仙 II -214	离散数学 仙 II -406	中国近现代史纲要 (一) 仙 II -112
3--4 节	计算机系统基础 (一) 仙 II -214 (二) 仙 II -215 数字电路与数字系统 仙 II -211	线性代数 (一) 仙 II -303 (二) 仙 II -505	数据结构 (一) 仙 II -214 (二) 仙 II -215	线性代数 (一) 仙 II -303 (二) 仙 II -505	中国近现代史纲要 (二) 仙 II -112 高级程序设计 双周, 基础实验楼乙 124 单周, 仙 I -106
5--6 节	数据结构 (一) 仙 II -214 (二) 仙 II -215	离散数学 仙 II -406 数字电路与数字系统 实验 (二) 基础 实验楼乙 125	程序设计基础实验 基础实验楼乙 126 计算机系统基础 (一) 仙 II -214	大学物理实验 实验楼乙 307、309、 314、316 丙 301-303、 305、307、308、317、 319、322	计算机系统基础 (二) 仙 II -215
7--8 节			形势与政策 (上) (单) 仙 I -107		
9--10 节	网络攻防实验 基础实验楼乙 125		网络安全实验, 基础实验楼乙 206	数据结构 (实验), 基 础实验楼乙 124、125	

另, 9月30日 (周日上周五课) 的安排:

- 1. 有关PA 2-1 中宏的说明
 - 2. Makefile的说明
 - 3. 有关debugging的杂谈
- 进度无关, 只讲一遍, 自由选择来听课

前情提要



- 数据的表示和运算
 - 整数：带符号、无符号 -> 各种运算 -> 标志位寄存器
 - 带小数的实数：IEEE 754浮点数 -> 各种运算和规格化

汇编知识提要

• 一条汇编语句的格式

(gcc接受的格式，也是我们写程序的格式)

AT&T格式： 指令长度后缀 源操作数, 目的操作数

`movl $0x7, %eax`

`MOV EAX, 0x7`

INTEL格式： 指令 目的操作数, 源操作数

(i386手册上采用的格式)

汇编知识提要

- 汇编语言是对应机器语言的助记符
- 汇编语言的一条语句对应一条机器指令

hello_world.o 反汇编其内容

相对于起始位置的偏移量

hello_world.o: file format elf32-i386

Di 实际存储在.o文件中的机器指令（和数据）

汇编助记符

00000000 <main>:

0:	8d 4c 24 04
4:	83 e4 f0
7:	ff 71 fc
a:	55
b:	89 e5
d:	53
e:	51
f:	e8 fc ff ff ff

lea	0x4(%esp),%ecx
and	\$0xfffffffff0,%esp
pushl	-0x4(%ecx)
push	%ebp
mov	%esp,%ebp
push	%ebx
push	%ecx
call	10 <main+0x10>

指令序列的执行流程 & 指令的解码

首先！待执行的指令和数据都放到内存里！

NEMU的初始化

NEMU的初始化

- 执行make run或者make test之后

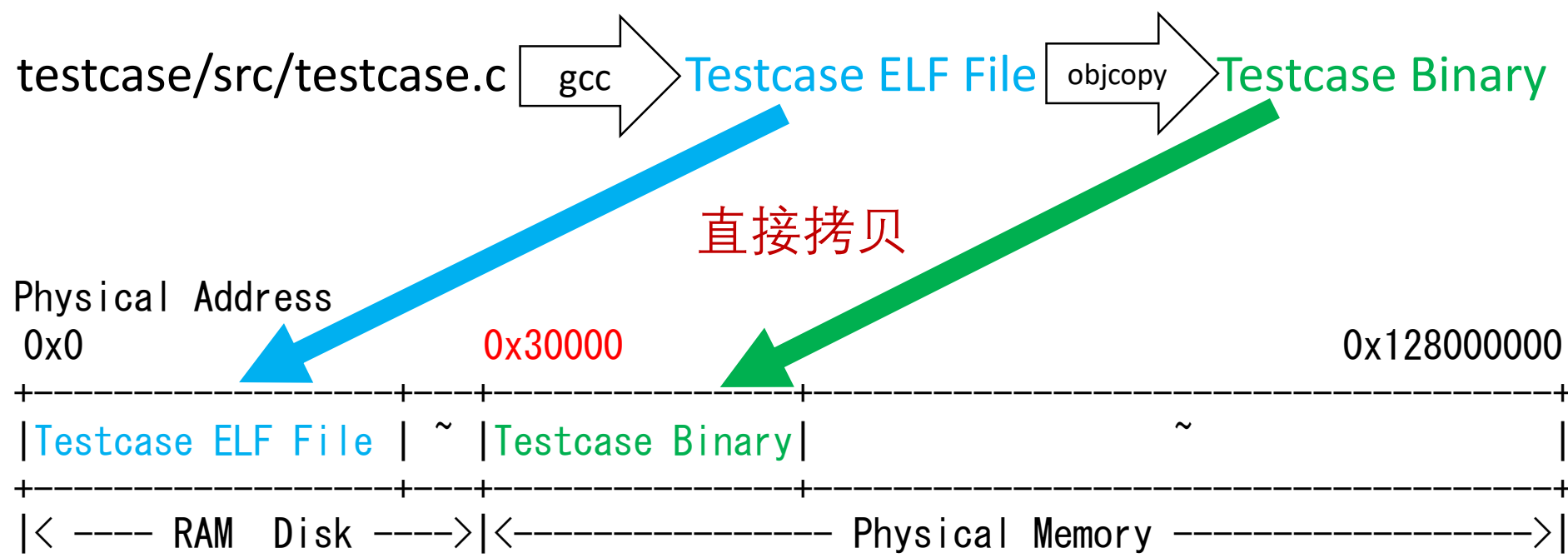


第一步，编译得到测试用例的可执行文件和二进制镜像文件

NEMU的初始化

- 执行make run或者make test之后

第二步，NEMU初始化模拟内存

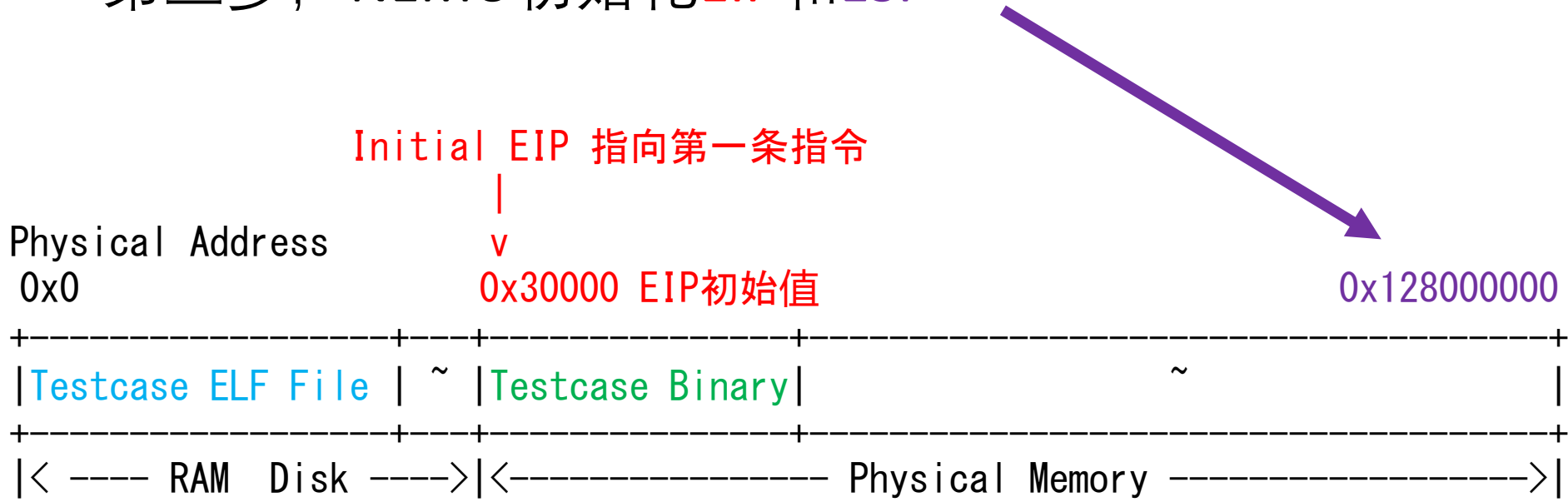


带有RAM Disk时的NEMU模拟内存划分方式

NEMU的初始化

- 执行make run或者make test之后

第三步，NEMU初始化EIP和ESP



带有RAM Disk时的NEMU模拟内存划分方式

NEMU的初始化

- 此时，NEMU中的模拟内存

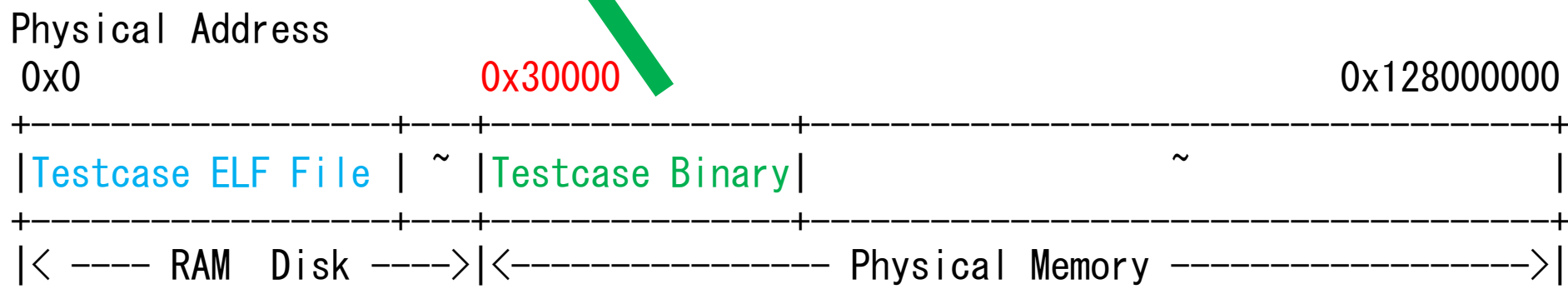
反汇编

8d 4c 24 04 83 e4 f0 ff 71
fc 55 89 e5 53 51 e8 fc ff
.....

```
hello_world.o:      file format elf32-i386

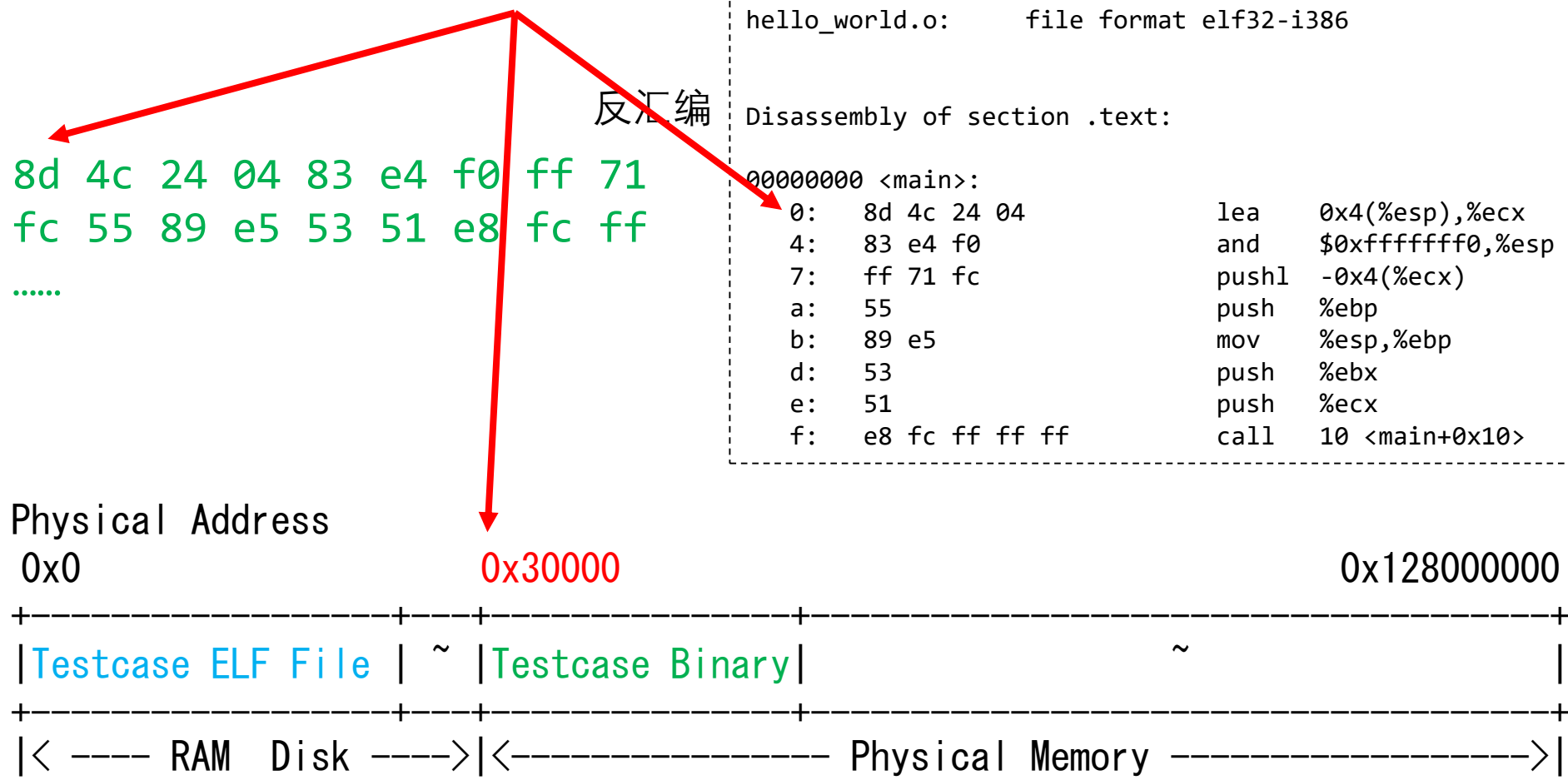
Disassembly of section .text:

00000000 <main>:
  0:  8d 4c 24 04          lea    0x4(%esp),%ecx
  4:  83 e4 f0             and    $0xffffffff0,%esp
  7:  ff 71 fc             pushl  -0x4(%ecx)
 a:  55                  push   %ebp
 b:  89 e5               mov    %esp,%ebp
 d:  53                  push   %ebx
 e:  51                  push   %ecx
 f:  e8 fc ff ff ff      call   10 <main+0x10>
```



NEMU的初始化

- 此时，NEMU中的EIP = 0x30000



第二！ 循环往复地执行EIP所指向的指令！

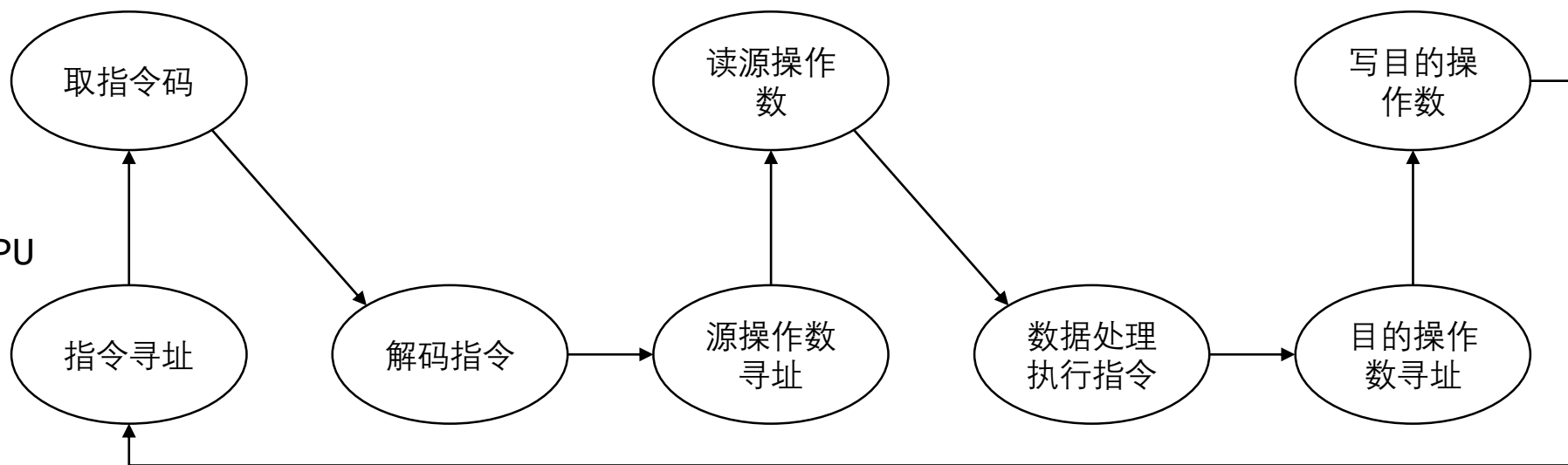
指令序列的执行

指令序列的执行

指令的顺序执行流程

Memory

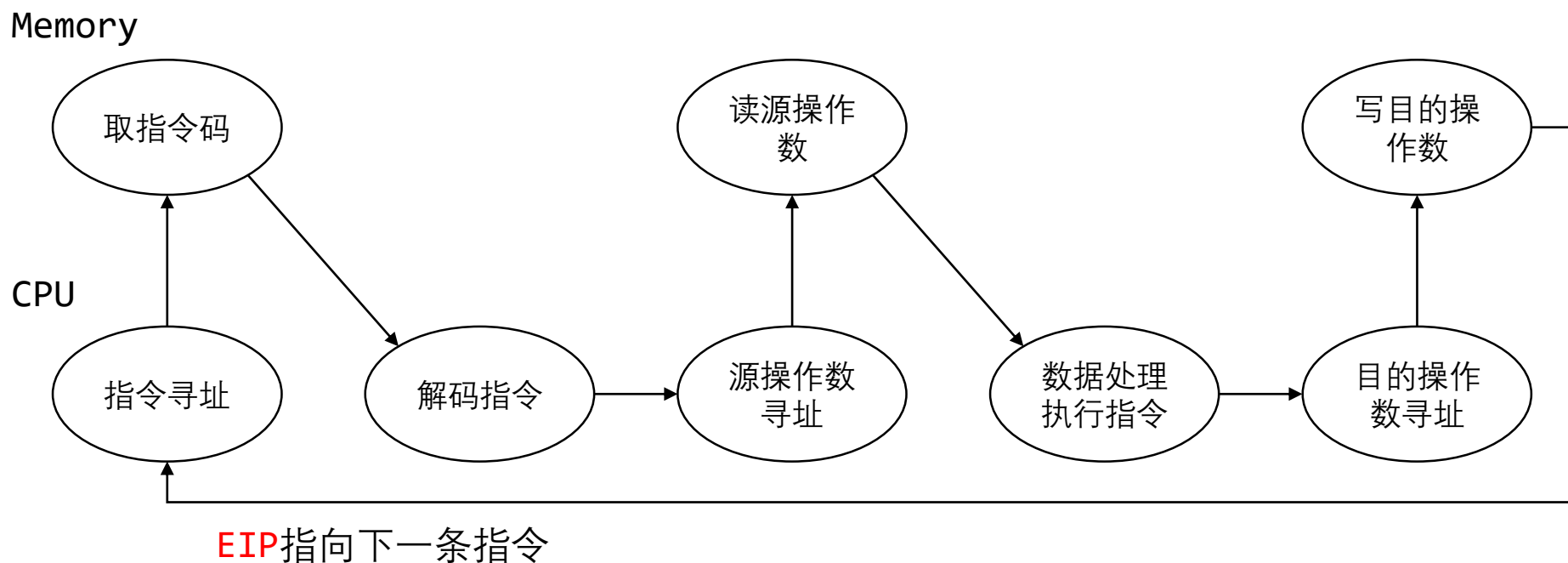
CPU



EIP指向下一条指令

指令序列的执行

指令的顺序执行流程



此循环往复的过程，用C语言如何模拟？

NEMU模拟指令执行

假设此时指令已经在内存中放好了，
EIP初始化为第一条指令的地址

- 指令循环：一条接一条的执行指令

nemu/src/cpu/cpu.c

```
void exec(uint32_t n) {  
    ...  
    while( n > 0 && nemu_state == NEMU_RUN) {  
        ...  
        instr_len = exec_inst();  
        cpu.eip += instr_len;  
        n--;  
        ...  
    }  
    ...  
}  
  
int exec_inst() {  
    uint8_t opcode = 0;  
    // get the opcode, 取操作数  
    opcode = instr_fetch(cpu.eip, 1);  
    // instruction decode and execution, 执行这条指令  
    int len = opcode_entry[opcode](cpu.eip, opcode);  
    return len; // 返回指令长度  
}
```

```
uint32_t instr_fetch(vaddr_t vaddr, size_t len) {  
    assert(len == 1 || len == 2 || len == 4);  
    return vaddr_read(vaddr, SREG_CS, len);  
}
```

怎么从main走到这个函数的？我们最后讲

循环执行指令

执行一条指令

3. 根据指令长度更新EIP，指向下一条指令

1. 取指令

2. 模拟执行

NEMU模拟指令执行

假设此时指令已经在内存中放好了，
EIP初始化为第一条指令的地址

- 指令循环：一条接一条的执行指令 [nemu/src/cpu/cpu.c](#)

```
void exec(uint32_t n) {
    ...
    while( n > 0 && nemu_state == NEMU_RUN) {
        ...
        instr_len = exec_inst();
        cpu.eip += instr_len;
        n--;
        ...
    }
    ...
}

int exec_inst() {
    uint8_t opcode = 0;
    // get the opcode, 取操作数
    opcode = instr_fetch(cpu.eip, 1);
    // instruction decode and execution, 执行这条指令
    int len = opcode_entry[opcode](cpu.eip, opcode);
    return len; // 返回指令长度
}
```

循环执行指令

执行一条指令

3. 根据指令长度
更新EIP，指向
下一条指令

1. 取指令

2. 模拟执行

NEMU模拟指令执行

假设此时指令已经在内存中放好了,
EIP初始化为第一条指令的地址

- 指令循环：一条接一条的执行指令

nemu/src/cpu/cpu.c

```
void exec(uint32_t n) {  
    ...  
    while( n > 0 && nemu_state == NEMU_RUN) {  
        ...  
        instr_len = exec_inst();  
        cpu.eip += instr_len;  
        n--;  
        ...  
    }  
    ...  
}  
  
int exec_inst() {  
    uint8_t opcode = 0;  
    // get the opcode, 取操作数  
    opcode = instr_fetch(cpu.eip, 1);  
    // instruction decode and execution 执行这条指令  
    int len = opcode_entry[opcode](cpu.eip, opcode);  
    return len; // 返回指令长度  
}
```

循环执行指令

执行一条指令

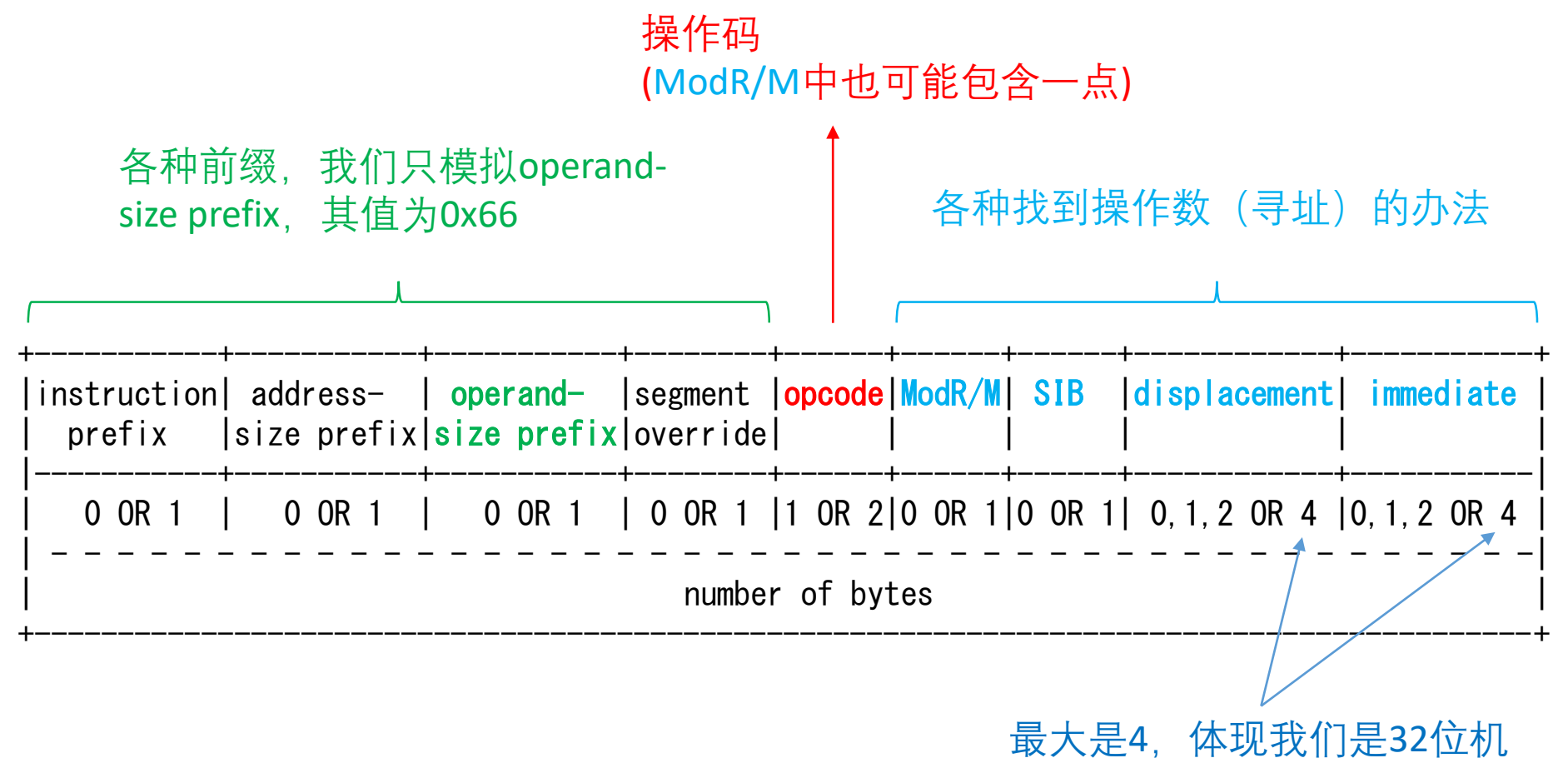
核心：指令的译码与执行

反复循环！指令的解码与执行

指令的解码与执行

指令的编码方式

内存中的指令数据: 8b 94 83 00 11 00 00 8b 45 f4



指令的编码方式

- 理解指令译码的过程
(假设此时EIP取初始值, 为某程序第一条指令)

EIP



8b 94 83 00 11 00 00 8b 45 f4

- 不是0x66, 操作数32位, 0x8b为操作码
- 查i386手册

指令的编码方式

B

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

8

i386手册, pg. 414, Appendix A, One-Byte Opcode Map

指令的编码方式

B

One-Byte Opcode Map

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

8

i386手册, pg. 414, Appendix A, One-Byte Opcode Map

指令的编码方式

MOV Gv, Ev

- E A modR/M byte follows the opcode and specifies the operand. The operand is either a general register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, a displacement.
- G The reg field of the modR/M byte selects a general register; e.g., ADD (00).
- v Word or double word, depending on operand size attribute.
- | | |
|-----|-----|
| 16位 | 32位 |
|-----|-----|

i386手册, pg. 412, Appendix A

指令的编码方式

- 理解指令译码的过程
(假设此时EIP取初始值, 为某程序第一条指令)

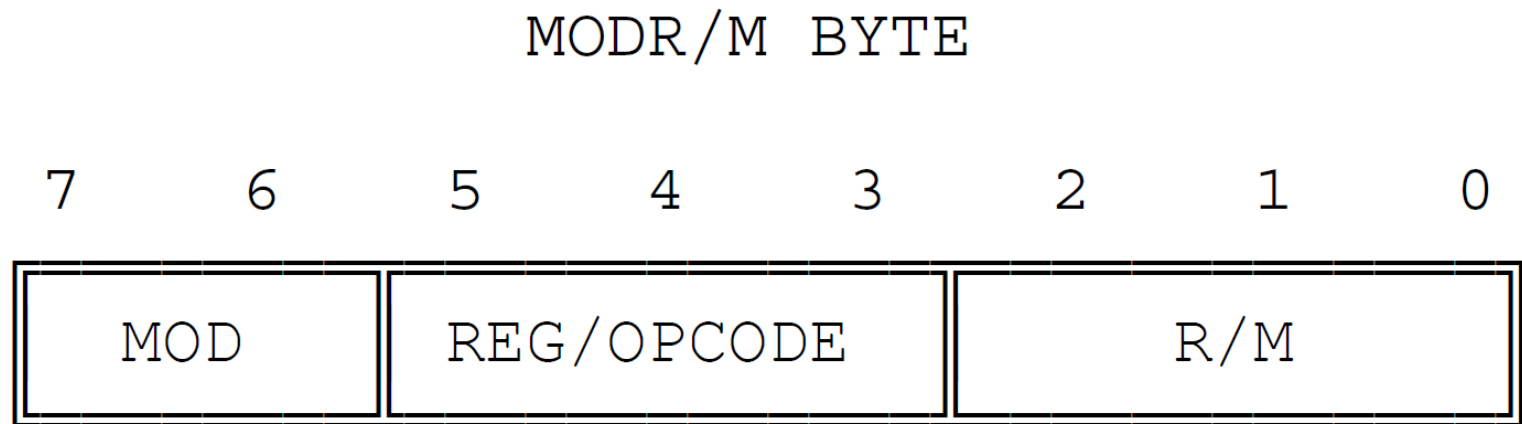
opcode



8b 94 83 00 11 00 00 8b 45 f4

- 不是0x66, 操作数32位, 0x8b为操作码
- 查i386手册 – Appendix A – 0x8b对应MOV Gv, Ev
 - Intel格式, 表示把一个Ev类型操作数MOV到Gv类型操作数里
 - objdump和gdb中采用的AT&T格式指令操作数顺序正好相反
 - 若有需要, 到i386手册Chapter 17细查
- Ev和Gv都说明后面跟ModR/M字节

指令的编码方式



i386手册, pg. 242, Section 17.2.1

指令的编码方式

- 理解指令译码的过程
(假设此时EIP取初始值, 为某程序第一条指令)

opcode modr/m



8b 94 83 00 11 00 00 8b 45 f4

-
- 不是0x66, 操作数32位, 0x8b为操作码
 - 查i386手册 – Appendix A – 0x8b对应 **MOV Gv, Ev**
 - Ev和Gv都说明后面跟ModR/M字节

0x94 =	10	010	100
	MOD	REG/OPCODE	R/M

指令的编码方式

- 理解指令译码的过程
(假设此时EIP取初始值, 为某程序第一条指令)

opcode modr/m



8b 94 83 00 11 00 00 8b 45 f4

-
- 不是0x66, 操作数32位, 0x8b为操作码
 - 查i386手册 – Appendix A – 0x8b对应 **MOV Gv, Ev**
 - Ev和Gv都说明后面跟ModR/M字节

0x94 =

根据Gv和没有0x66前缀, 查i386手册表17.2, 010表示edx

10

010

100

MOD

REG/OPCODE

R/M

指令的编码方式

- 理解指令译码的过程
(假设此时EIP取初始值, 为某程序第一条指令)

opcode modr/m



8b 94 83 00 11 00 00 8b 45 f4

-
- 不是0x66, 操作数32位, 0x8b为操作码
 - 查i386手册 – Appendix A – 0x8b对应MOV Gv, Ev

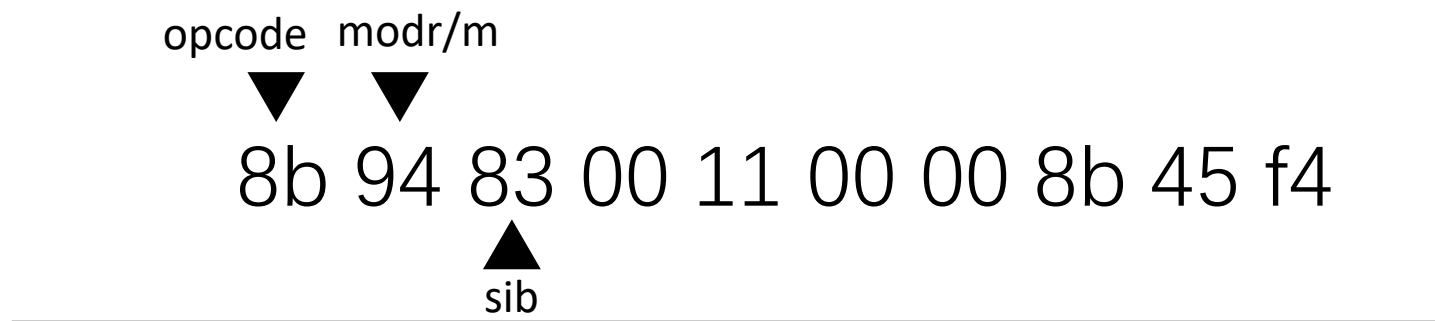
3. Ev和Gv都说明后面跟ModR/M字节

查i386手册表17-3, 发现是内存地址disp32[--][--], 还有SIB字节

0x94 =	10	010	100
	MOD	REG/OPCODE	R/M

指令的编码方式

- 理解指令译码的过程
(假设此时EIP取初始值, 为某程序第一条指令)



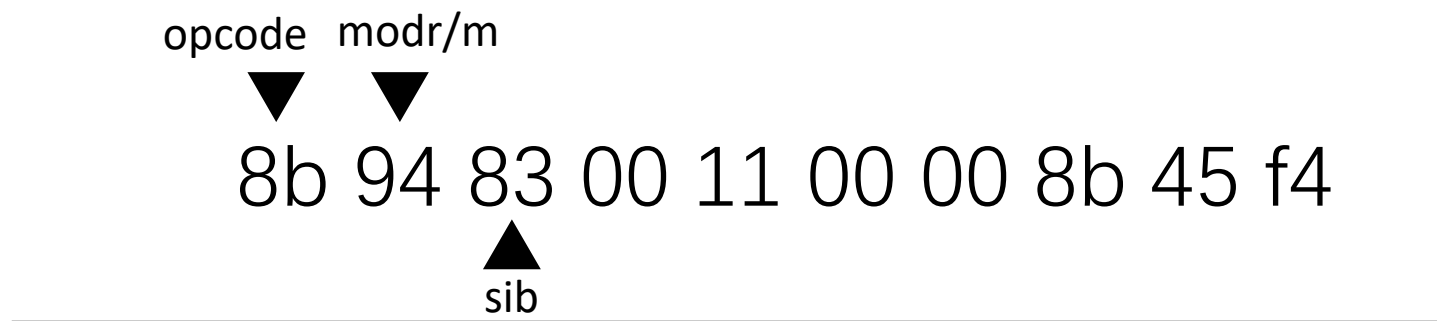
1. 不是0x66, 操作数32位, 0x8b为操作码
2. 查i386手册 – Appendix A – 0x8b对应MOV Gv, Ev
3. Ev和Gv都说明后面跟ModR/M字节
4. 根据Mod + R/M域决定有SIB字节 (内存地址disp32[--][--])

0x83 = 10 000 011

 SS INDEX BASE

指令的编码方式

- 理解指令译码的过程
(假设此时EIP取初始值, 为某程序第一条指令)



- 不是0x66, 操作数32位, 0x8b为操作码
- 查i386手册 – Appendix A – 0x8b对应MOV Gv, Ev
- Ev和Gv都说明后面跟ModR/M字节
- 根据Mod + R/M域决定有SIB字节 (内存地址disp32[--][--])

内存地址 = $\text{disp32} + \text{ebx} + \text{eax} * 4$

0x83 = 10 000 011

查i386手册表17-4

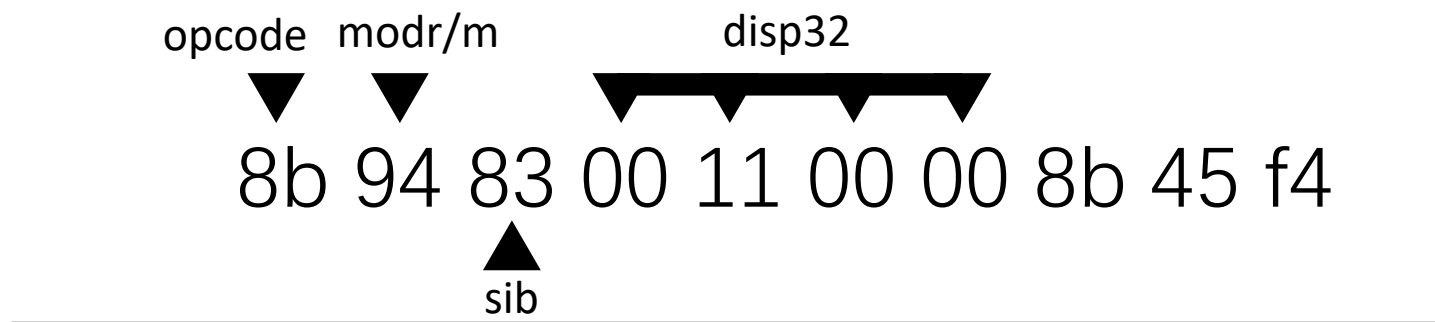
SS
*4

INDEX
eax

BASE
ebx

指令的编码方式

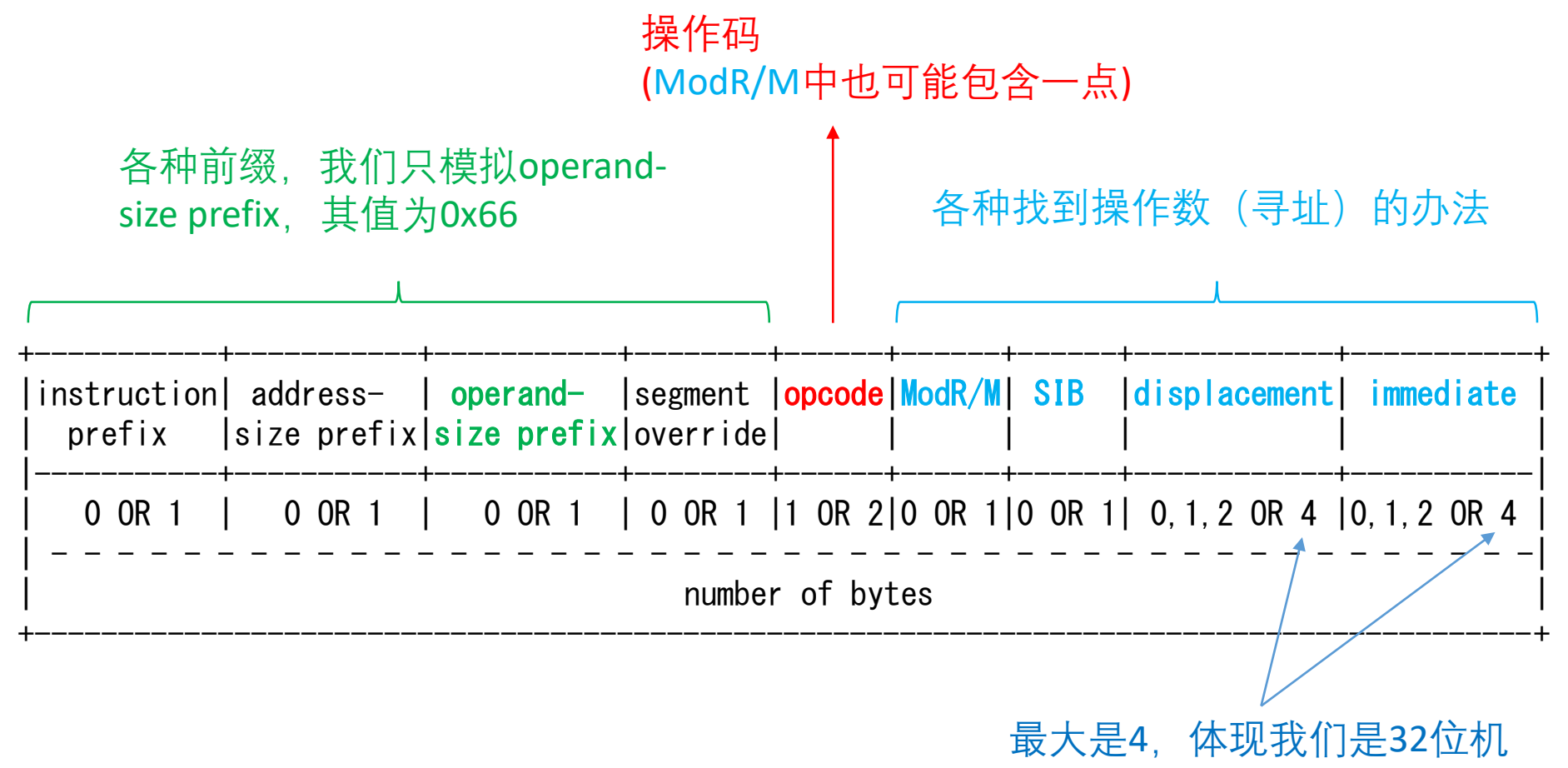
- 理解指令译码的过程
(假设此时EIP取初始值, 为某程序第一条指令)



1. 不是0x66, 操作数32位, 0x8b为操作码
2. 查i386手册 – Appendix A – 0x8b对应MOV Gv, Ev
3. Ev和Gv都说明后面跟ModR/M字节
4. 根据Mod + R/M域决定有SIB字节 (内存地址disp32[--][--])
5. SIB字节后面自然还有disp32 – 32位的偏移量 (小端方式)

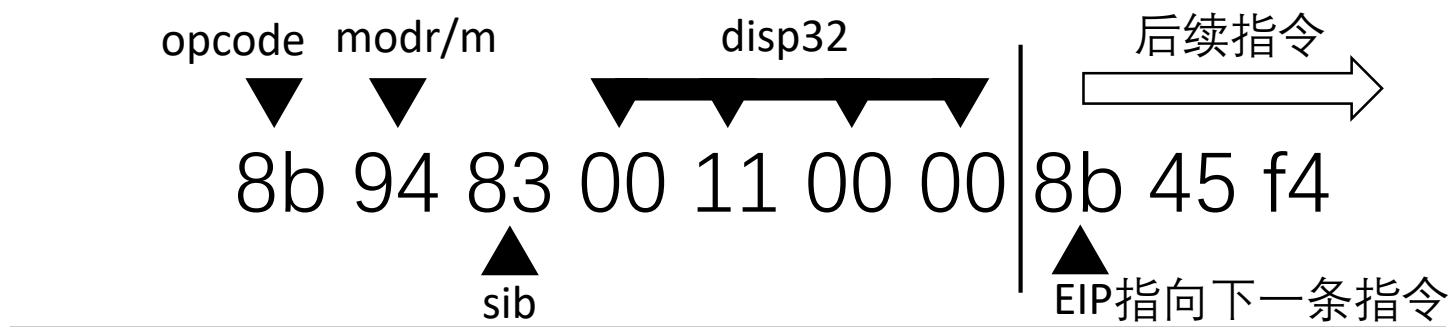
指令的编码方式

内存中的指令数据: 8b 94 83 00 11 00 00 8b 45 f4



指令的编码方式

- 理解指令译码的过程
(假设此时EIP取初始值, 为某程序第一条指令)



1. 不是0x66, 操作数32位, 0x8b为操作码
2. 查i386手册 – Appendix A – 0x8b对应MOV Gv, Ev
3. Ev和Gv都说明后面跟ModR/M字节
4. 根据Mod + R/M域决定有SIB字节 (内存地址disp32[--][--])
5. SIB字节后面自然还有disp32 – 32位的偏移量 (小端方式)
6. 该指令所有需要的信息已经获得, 对应AT&T格式汇编:

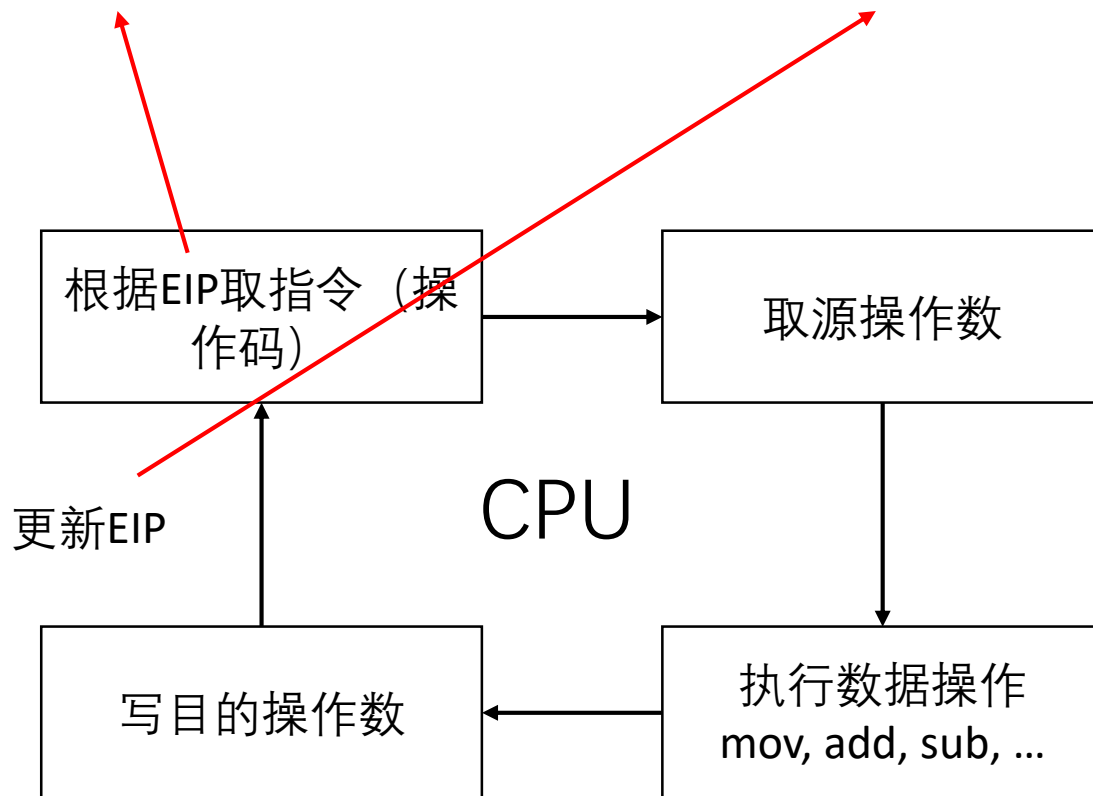
movl 0x1100(%ebx, %eax, 4), %edx

指令的编码方式

- X86有多少条指令？
 - I386手册Chapter 17
 - 手册上没有的：<http://www.felixcloutier.com/x86/>
- 指令怎么那么多？结构怎么这么复杂？
 - 复杂指令集计算机（CISC）：Intel，AMD为代表
 - 程序员选择多
 - 优化方案多
 - 可扩展性强，向下兼容性好
- 有没有简单点的？
 - 精简指令集计算机（RISC）：MIPS为代表
 - CPU实现简单，嵌入式系统欢迎
 - 一条指令集计算机（One Instruction Set Computers, OISC）
 - https://en.wikipedia.org/wiki/One_instruction_set_computer

指令序列的执行流程

内存: 8b 94 83 00 11 00 00 8b 45 f4



先把指令依次在内存中排好, 给EIP赋一个初始值, 指向第一条指令, CPU就可以循环执行每一条指令了

NEMU模拟指令执行

假设此时指令已经在内存中放好了,
EIP初始化为第一条指令的地址

- 指令循环：一条接一条的执行指令

nemu/src/cpu/cpu.c

```
void exec(uint32_t n) {  
    ...  
    while( n > 0 && nemu_state == NEMU_RUN) {  
        ...  
        instr_len = exec_inst();  
        cpu.eip += instr_len;  
        n--;  
        ...  
    }  
    ...  
}
```

循环执行指令

```
int exec_inst() {  执行一条指令  
    uint8_t opcode = 0;  
    // get the opcode, 取操作数  
    opcode = instr_fetch(cpu.eip, 1);  
    // instruction decode and execution 执行这条指令  
    int len = opcode_entry[opcode](cpu.eip, opcode);  
    return len; // 返回指令长度  
}
```

核心：指令的译码与执行

NEMU模拟指令译码和执行

NEMU模拟指令译码和执行

- `opcode_entry`是一个函数指针数组
 - 其中每一个元素指向一条指令的模拟函数

访问 `opcode_entry[opcode]` == 调用对应位置指向的函数
实现某一条指令的功能
nemu/src/cpu/decode/opcode.c

```
#include "cpu/instr.h"  
instr_func opcode_entry[256] = { ... }
```

nemu/include/cpu/instr_helper.h

```
// the type of an instruction entry  
typedef int (*instr_func)(uint32_t eip, uint8_t opcode);
```

内存: C7 05 48 11 10 00 02 00 00 00

▲
当前EIP

mov_i2rm_v
是模拟C7指
令的函数

nemu/src/cpu/instr/mov.c

```
make_instr_func(mov_i2rm_v) {  
    OPERAND rm, imm;  
    rm.data_size = data_size;  
    int len = 1;  
    len += modrm_rm(eip + 1, &rm);  
  
    imm.type = OPR_IMM;  
    imm.addr = eip + len;  
    imm.data_size = data_size;  
  
    operand_read(&imm);  
    rm.val = imm.val;  
    operand_write(&rm);  
  
    return len + data_size / 8;  
}
```

5. 循环开启下一
条指令

```
void exec(uint32_t n) {  
    ...  
    while( n > 0 && nemu_state == NEMU_RUN) {  
        ...  
        instr_len = exec_inst();  
        cpu.eip += instr_len;  
        n--;  
        ...  
    }  
    ...  
}
```

nemu/src/cpu/cpu.c

```
int exec_inst() {  
    uint8_t opcode = 0;  
    opcode = instr_fetch(cpu.eip, 1);  
    int len = opcode_entry[opcode](cpu.eip, opcode);  
    return len; // 返回指令长度  
}
```

1. cpu.eip指向C7

2. Opcode取出为C7

3. 访问数组即函数调用

4. 返回指令长度

```
#include "cpu/instr.h"  
instr_func opcode_entry[256] = {  
    ...  
    /* 0xc4 - 0xc7 */  
    inv, inv, mov_i2rm_b, mov_i2rm_v,  
    ...  
}
```

nemu/src/cpu/decode/opcode.c

NEMU模拟指令译码和执行

- 怎么写某操作码对应的instr_func?

```
// 宏展开后这一行即为 int mov_i2rm_v(uint32_t eip, uint8_t opcode) {
make_instr_func(mov_i2rm_v) {
    OPERAND rm, imm;      // OPERAND定义在nemu/include/cpu/operand.h
                           // 看教程§2-1.2.3

    rm.data = imm.val;    // 立即数放入寄存器，表示操作数的地址长度
    int len = 0;          // 立即数长度
    len += 16;             // 立即数长度

    imm.val = 0;           // 立即数清零
    imm.data = 0;          // 立即数清零
    imm.data = 0;          // 立即数清零

    operand = 0;           // 立即数清零
    rm.val = imm.val;
    operand_write(&rm);

    return len + data_size / 8; // opcode长度 + ModR/M字节扫描长度 + 立即数长度
}
```

这是一条把一个立即数mov到R/M中的指令，操作数长度为16或32位

推荐命名规则：

指令名_源操作数类型2目的操作数类型_长度后缀

NEMU模拟指令译码和执行

- 于是在`nemu/include/cpu/instr_helper.h`中我们给出了用于精简指令实现的宏，一些实用信息（详细用法参阅教程，比较详尽）

`#define make_instr_impl_2op(inst_name, src_type, dest_type, suffix) ...`

- `inst_name`就是指令的名称：mov, add, sub, ...
- `src_type`和`dest_type`是源和目的操作数类型，与`decode_operand`系列宏一致：
 - rm – 寄存器或内存地址 – 对应手册E类型
 - r – 寄存器地址 – 对应手册G类型
 - i – 立即数 – 对应手册I类型
 - m – 内存地址 – 差不多对应手册M类型
 - a – 根据操作数长度对应al, ax, eax – 手册里没有
 - c – 根据操作数长度对应cl, cx, ecx – 手册里没有
 - o – 偏移量 – 对应手册里的O类型
- `suffix`是操作数长度后缀，与`decode_data_size`系列宏一致：
 - b, w, l, v – 8, 16, 32, 16/32位
 - bv – 源操作数为8位，目的操作数为16/32位，特殊指令用到
 - short, near – jmp指令用到，分别指代8位和32位

你可以根据实际需要添加其他的宏或改写已有的宏

NEMU模拟指令译码和执行

- 怎么写某操作码对应的instr_func?

```
// 宏展开后这一行即为 int mov_i2rm_v(uint32_t eip, uint8_t opcode) {  
make_instr_func(mov_i2rm_v) {  
    OPERAND rm, imm;          // OPERAND定义在nemu/include/cpu/operand.h  
                                // 看教程§2-1.2.3  
    rm.data_size = data_size; // data_size是个全局变量, 表示操作数的比特长度  
    int len = 1;              // opcode 长度1字节  
    len += modrm_rm(eip + 1, &rm); // 读ModR/M字节, rm的type和addr会被填写  
  
    imm.type = OPR_IMM;       // 填入立即数类型  
    imm.addr = eip + len;     // 找到立即数的地址  
    imm.data_size = data_size;  
  
    operand_read(&imm);       // 执行 mov 操作  
    rm.val = imm.val;  
    operand_write(&rm);  
  
    return len + data_size / 8; // opcode长度 + ModR/M字节扫描长度 + 立即数长度  
}
```

NEMU模拟指令译码和执行

- 怎么写某操作码对应的instr_func?

```
// 宏展开后这一行即为 int mov_i2rm_v(uint32_t eip, uint8_t opcode) {  
make_instr_func(mov_i2rm_v) {  
    OPERAND rm, imm;      // OPERAND定义在nemu/include/cpu/operand.h  
                           // 看教程§2-1.2.3
```

```
    rm.d  
    int le  
    len +
```

[nemu/include/cpu/instr_helper.h](#)

```
#define make_instr_func(name) int name(uint32_t eip, uint8_t opcode)
```

```
    imm.type = OPR_IMM;    // 填入立即数类型  
    imm.addr = eip + len;  // 找到立即数的地址  
    imm.data_size = data_size;
```

```
#include "cpu/instr.h"
```

```
    opcode_entry[256] = { ... }
```

对比一下
opcode_entry的类型

```
    // the type of an instruction entry  
    typedef int (*instr_func)(uint32_t eip, uint8_t opcode);  
    return  
}
```

NEMU模拟指令译码和执行

- 怎么写某操作码对应的instr_func?

```
// 宏展开后这一行即为 int mov_i2rm_v(uint32_t eip, uint8_t opcode) {  
make_instr_func(mov_i2rm_v) {  
    OPERAND rm, imm;          // OPERAND定义在nemu/include/cpu/operand.h  
                                // 看教程§2-1.2.3  
    rm.data_size = data_size; // data_size是个全局变量, 表示操作数的比特长度  
    int len = 1;              // opcode 长度1字节  
    len += modrm_rm(eip + 1, &rm); // 读ModR/M字节, rm的type和addr会被填写  
  
    imm.type = OPR_IMM;       // 填入立即数类型  
    imm.addr = eip + len;     // 找到立即数的地址  
    imm.data_size = data_size;  
  
    operand_read(&imm);       // 执行 mov 操作  
    rm.val = imm.val;  
    operand_write(&rm);  
  
    return len + data_size / 8; // opcode长度 + ModR/M字节扫描长度 + 立即数长度  
}
```

NEMU模拟指令译码和执行

- 怎么写某操作码对应的instr_func?

```
// 宏展开后这一行即为 int mov_i2rm_v(
make_instr_func(mov_i2rm_v) {
    OPERAND rm, imm;           // OPERAND定
                                // 看教程§2-1
    rm.data_size = data_size; // data_size是
    int len = 1;                // opcode 长度
    len += modrm_rm(eip + 1, &rm); // 读M

    imm.type = OPR_IMM;         // 填入立即
    imm.addr = eip + len;       // 找到立即
    imm.data_size = data_size;

    operand_read(&imm);         // 执行 mov
    rm.val = imm.val;
    operand_write(&rm);

    return len + data_size / 8; // opcode长
}
```

nemu/include/cpu/operand.h

```
enum {OPR_IMM, OPR_REG, OPR_MEM,
      OPR_CREG, OPR_SREG};
```

```
typedef struct {
```

```
    int type;
```

// addr地址, 随type不同解释也不同

```
    uint32_t addr;
```

```
    uint8_t sreg; // 现在不管
```

```
    uint32_t val;
```

// data_size = 8, 16, 32

```
    size_t data_size;
```

```
#ifdef DEBUG
```

```
    MEM_ADDR mem_addr;
```

```
#endif
```

```
} OPERAND;
```

NEMU模拟指令译码和执行

- 怎么写某操作码对应的instr_func?

```
// 宏展开后这一行即为 int mov_i2rm_v(uint32_t eip, uint8_t opcode) {  
make_instr_func(mov_i2rm_v) {  
    OPERAND rm, imm;          // OPERAND定义在nemu/include/cpu/operand.h  
                                // 看教程§2-1.2.3  
    rm.data_size = data_size; // data_size是个全局变量, 表示操作数的比特长度  
    int len = 1;              // opcode 长度1字节  
    len += modrm_rm(eip + 1, &rm); // 读ModR/M字节, rm的type和addr会被填写  
  
    imm.type = OPR_IMM;       // 填入立即数类型  
    imm.addr = eip + len;     // 找到立即数的地址  
    imm.data_size = data_size;  
  
    operand_read(&imm);       // 执行 mov 操作  
    rm.val = imm.val;  
    operand_write(&rm);  
  
    return len + data_size / 8; // opcode长度 + ModR/M字节扫描长度 + 立即数长度  
}
```

NEMU模拟指令译码和执行

- 怎么写某操作码对应的instr_func?

// 宏展开后这一行即为 `int mov_i2rm_v(uint32_t eip, uint8_t opcode) {`

`make_instr_func(mov_i2rm_v) {`

`OPERAND rm, imm;`

`rm.data_size = data_size;`

`int len = 1;`

`len += modrm_rm(eip + 1,`

`imm.type = OPR_IMM;`

`imm.addr = eip + len;`

`imm.data_size = data_size;`

`operand_read(&imm);`

`rm.val = imm.val;`

`operand_write(&rm);`

`return len + data_size / 8;`

`}`

`nemu/src/cpu/instr/data_size.c`

`uint8_t data_size = 32;`

`make_instr_func(data_size_16) {`

`uint8_t op_code = 0;`

`int len = 0;`

`data_size = 16;`

`op_code = instr_fetch(eip + 1, 1);`

`len = opcode_entry[op_code](eip + 1, op_code);`

`data_size = 32;`

`return 1 + len;`

`}`

// opcode长度 + ModR/M字节扫描长度 + 立即数长度

NEMU模拟指令译码和执行

- 怎么写某操作码对应的instr_func?

```
// 宏展开后这一行即为 int mov_i2rm_v(uint32_t eip, uint8_t opcode) {  
make_instr_func(mov_i2rm_v) {  
    OPERAND rm, imm;          // OPERAND定义在nemu/include/cpu/operand.h  
                                // 看教程§2-1.2.3  
    rm.data_size = data_size; // data_size是个全局变量, 表示操作数的比特长度  
    int len = 1;              // opcode 长度1字节  
    len += modrm_rm(eip + 1, &rm); // 读ModR/M字节, rm的type和addr会被填写  
  
    imm.type = OPR_IMM;       // 填入立即数类型  
    imm.addr = eip + len;     // 找到立即数的地址  
    imm.data_size = data_size;  
  
    operand_read(&imm);       // 执行 mov 操作  
    rm.val = imm.val;  
    operand_write(&rm);  
  
    return len + data_size / 8; // opcode长度 + ModR/M字节扫描长度 + 立即数长度  
}
```

NEMU模拟指令译码和执行

- 怎么写某操作码对应的instr_func?

```
// 宏展开后这一行即为 int mov_i2rm_v(uint32_t eip, uint8_t opcode) {  
make_instr_func(mov_i2rm_v) {  
    OPERAND rm, imm;          // OPERAND定义在nemu/include/cpu/operand.h  
                                // 看教程§2-1.2.3  
    rm.data_size = data_size; // data_size是个全局变量, 表示操作数的比特长度  
    int len = 1;              // opcode 长度1字节  
    len += modrm_rm(eip + 1, &rm); // 读ModR/M字节, rm的type和addr会被填写
```

```
    imm.type = OPR_IMM;  
    imm.addr = eip + len;  
    imm.data_size = data_size;
```

```
    operand_read(&imm, rm);  
    rm.val = imm.val;  
    operand_write(&rm, imm.val);
```

[nemu/src/cpu/decode/modrm.c](#)

```
int modrm_rm(uint32_t eip, OPERAND * rm) ;
```

就是查表过程变成代码

会将传入的rm变量的type和addr（包括sreg）填好

返回解析modr/m所扫描过的字节数（包括可能的SIB和disp）

```
    return len + data_size / 8; // opcode长度 + ModR/M字节扫描长度 + 立即数长度
```

```
}
```

NEMU模拟指令译码和执行

- 怎么写某操作码对应的instr_func?

```
// 宏展开后这一行即为 int mov_i2rm_v(uint32_t eip, uint8_t opcode) {  
make_instr_func(mov_i2rm_v) {  
    OPERAND rm, imm;      // OPERAND定义在nemu/include/cpu/operand.h  
                           // 看教程§2-1.2.3  
    rm.data_size = data_size; // data_size是个全局变量, 表示操作数的比特长度  
    int len = 1;             // opcode 长度1字节  
    len += modrm_rm(eip + 1, &rm); // 读ModR/M字节, rm的type和addr会被填写  
  
    imm.type = OPR_IMM;      // 填入立即数类型  
    imm.addr = eip + len;    // 找到立即数的地址  
    imm.data_size = data_size;  
  
    operand_read(&imm);      // 执行 mov 操作  
    rm.val = imm.val;  
    operand_write(&rm);  
  
    return len + data_size / 8; // opcode长度 + ModR/M字节扫描长度 + 立即数长度  
}
```

NEMU模拟指令译码和执行

nemu/src/cpu/decode/operand.c

```
void operand_read(OPERAND * opr) {
    switch(opr->type) {
        case OPR_MEM: ...
        case OPR_IMM:
            opr->val = vaddr_read(opr->addr, SREG_CS, 4);
            break;
        case OPR_REG:
            if(opr->data_size == 8) {
                opr->val = cpu.gpr[opr->addr % 4]._8[opr->addr / 4];
            } else {
                opr->val = cpu.gpr[opr->addr]._32;
            }
            break;
        case OPR_CREG: ...
        case OPR_SREG: ...

        // deal with data size
        switch(opr->data_size) {
            case 8: opr->val = opr->val & 0xff; break;
            case 16: opr->val = opr->val & 0xffff; break;
            case 32: break;
            default: ...
        }
    }
}
```

NEMU模拟指令译码和执行

- 怎么写某操作码对应的instr_func?

```
// 宏展开后这一行即为 int mov_i2rm_v(uint32_t eip, uint8_t opcode) {  
make_instr_func(mov_i2rm_v) {  
    OPERAND rm, imm;          // OPERAND定义在nemu/include/cpu/operand.h  
                                // 看教程§2-1.2.3  
    rm.data_size = data_size; // data_size是个全局变量, 表示操作数的比特长度  
    int len = 1;              // opcode 长度1字节  
    len += modrm_rm(eip + 1, &rm); // 读ModR/M字节, rm的type和addr会被填写  
  
    imm.type = OPR_IMM;        // 填入立即数类型  
    imm.addr = eip + len;      // 找到立即数的地址  
    imm.data_size = data_size;  
  
    operand_read(&imm);        // 执行 mov 操作  
    rm.val = imm.val;  
    operand_write(&rm);  
  
    return len + data_size / 8; // opcode长度 + ModR/M字节扫描长度 + 立即数长度  
}
```

执行mov操作并且
写目的操作数

NEMU模拟指令译码和执行

- 怎么写某操作码对应的instr_func?

```
// 宏展开后这一行即为 int mov_i2rm_v(uint32_t eip, uint8_t opcode) {  
make_instr_func(mov_i2rm_v) {  
    OPERAND rm, imm;          // OPERAND定义在nemu/include/cpu/operand.h  
                                // 看教程§2-1.2.3  
    rm.data_size = data_size; // data_size是个全局变量, 表示操作数的比特长度  
    int len = 1;              // opcode 长度1字节  
    len += modrm_rm(eip + 1, &rm); // 读ModR/M字节, rm的type和addr会被填写  
  
    imm.type = OPR_IMM;        // 填入立即数类型  
    imm.addr = eip + len;      // 找到立即数的地址  
    imm.data_size = data_size;  
  
    operand_read(&imm);        // 执行 mov 操作  
    rm.val = imm.val;  
    operand_write(&rm);  
  
    return len + data_size / 8; // opcode长度 + ModR/M字节扫描长度 + 立即数长度  
}
```

返回指令长度

内存: C7 05 48 11 10 00 02 00 00 00

▲
当前EIP

mov_i2rm_v
是模拟C7指
令的函数

nemu/src/cpu/instr/mov.c

```
make_instr_func(mov_i2rm_v) {  
    OPERAND rm, imm;  
    rm.data_size = data_size;  
    int len = 1;  
    len += modrm_rm(eip + 1, &rm);  
  
    imm.type = OPR_IMM;  
    imm.addr = eip + len;  
    imm.data_size = data_size;  
  
    operand_read(&imm);  
    rm.val = imm.val;  
    operand_write(&rm);  
  
    return len + data_size / 8;  
}
```

5. 循环开启下一
条指令

```
void exec(uint32_t n) {  
    ...  
    while( n > 0 && nemu_state == NEMU_RUN) {  
        ...  
        instr_len = exec_inst();  
        cpu.eip += instr_len;  
        n--;  
        ...  
    }  
    ...  
}  
  
int exec_inst() {  
    uint8_t opcode = 0;  
    opcode = instr_fetch(cpu.eip, 1);  
    int len = opcode_entry[opcode](cpu.eip, opcode);  
    return len; // 返回指令长度  
}
```

1. cpu.eip指向C7

2. Opcode取出为C7

3. 访问数组即函数调用

4. 返回指令长度

```
#include "cpu/instr.h"  
instr_func opcode_entry[256] = {  
    ...  
    /* 0xc4 - 0xc7 */    inv, inv, mov_i2rm_b, mov_i2rm_v,  
    ...  
}
```

nemu/src/cpu/decode/opcode.c

NEMU模拟指令译码和执行

`./nemu/nemu --testcase add`

也可以 `./nemu/nemu --autorun --testcase add`

- 所以PA 2-1要做的任务： 执行 `make run` 或 `make test_pa-2-1`

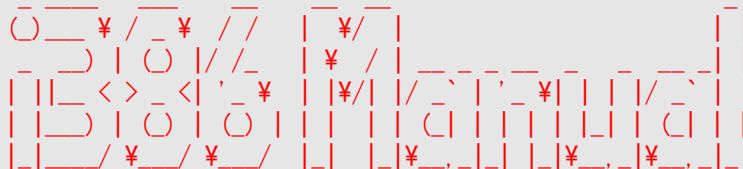
invalid opcode(eip = 0x00030033): 83 f8 01 66 c7 05 34 12 ...

There are two cases which will trigger this unexpected exception:

1. The instruction at eip = 0x00030033 is not implemented.
2. Something is implemented incorrectly.

Find this eip value(0x00030033) in the disassembling result to distinguish which case it is.

If it is the first case, see



for more details.

If it is the second case, remember:

- * The machine is always right!
- * Every line of untested code is always wrong!

不加--autorun的话
会进入交互调试界面，
相应的命令看
guide pg. 28-29的表
格

1. 查i386手册得知这是一条什么指令
 - a) 先查appendix A得知指令的类型和格式
 - b) 必要的话查section 17.2.1译码ModR/M和SIB字节
 - c) 必要的话查section 17.2.2.11查看指令的具体含义和细节

NEMU模拟指令译码和执行

- 所以PA 2-1要做的任务： 执行make run或make test_pa-2-1

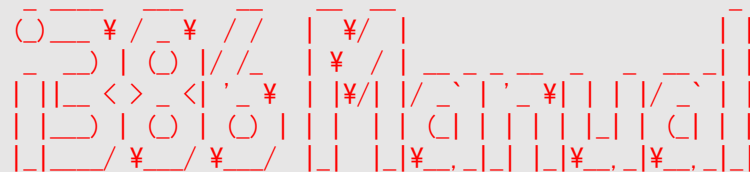
invalid opcode(eip = 0x00030033): 83 f8 01 66 c7 05 34 12 ...

There are two cases which will trigger this unexpected exception:

1. The instruction at eip = 0x00030033 is not implemented.
2. Something is implemented incorrectly.

Find this eip value(0x00030033) in the disassembling result to distinguish which case it is.

If it is the first case, see



for more details.

If it is the second case, remember:

- * The machine is always right!
- * Every line of untested code is always wrong!

1. 查i386手册得知这是一条什么指令
2. 写该操作码对应的instr_func
 - a) 例如: make_instr_func(mov_i2rm_v)

NEMU模拟指令译码和执行

- 所以PA 2-1要做的任务： 执行make run或make test_pa-2-1

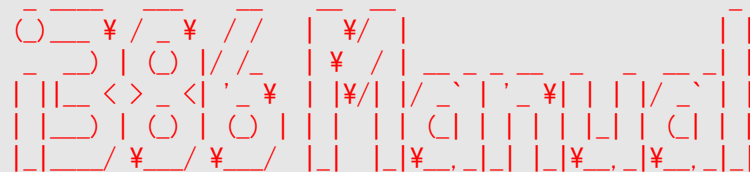
invalid opcode(eip = 0x00030033): 83 f8 01 66 c7 05 34 12 ...

There are two cases which will trigger this unexpected exception:

1. The instruction at eip = 0x00030033 is not implemented.
2. Something is implemented incorrectly.

Find this eip value(0x00030033) in the disassembling result to distinguish which case it is.

If it is the first case, see



for more details.

If it is the second case, remember:

- * The machine is always right!
- * Every line of untested code is always wrong!

1. 查i386手册得知这是一条什么指令
2. 写该操作码对应的instr_func
3. 把这个函数在nemu/include/cpu/instr.h中声明一下
4. 在opcode_entry对应该操作码的地方把这个函数的函数名填进去替代原来的inv

NEMU模拟指令译码和执行

- 所以PA 2-1要做的任务： 执行make run或make test_pa-2-1

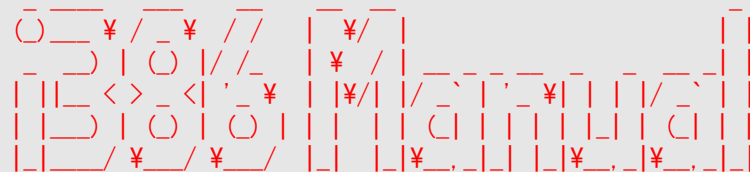
invalid opcode(eip = 0x00030033): 83 f8 01 66 c7 05 34 12 ...

There are two cases which will trigger this unexpected exception:

1. The instruction at eip = 0x00030033 is not implemented.
2. Something is implemented incorrectly.

Find this eip value(0x00030033) in the disassembling result to distinguish which case it is.

If it is the first case, see



for more details.

If it is the second case, remember:

- * The machine is always right!
- * Every line of untested code is always wrong!

1. 查i386手册得知这是一条什么指令
2. 写该操作码对应的instr_func
3. 把这个函数在nemu/include/cpu/instr.h中声明一下
4. 在opcode_entry对应该操作码的地方把这个函数的函数名填进去替代原来的inv
5. 重复上述过程直至完成所有需要模拟的指令

NEMU模拟指令译码和执行

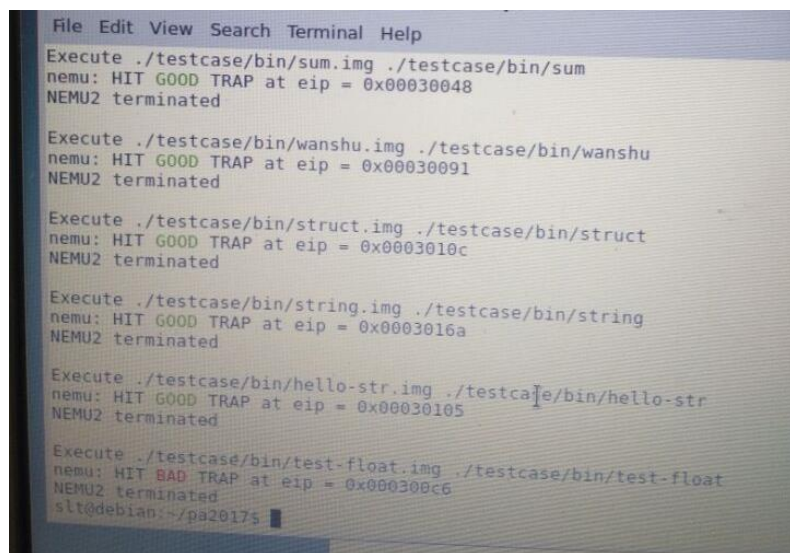
- 针对这个框架有一些要特别注意的地方

nemu/src/cpu/cpu.c

```
void exec(uint32_t n) {  
    ...  
    while( n > 0 && nemu_state == NEMU_RUN) {  
        ...  
        instr_len = exec_inst();  
        cpu.eip += instr_len;  
        n--;  
        ...  
    }  
    ...  
}
```

这一步非常机械，对于某些指令，如特殊的jmp、ret中涉及到跳转到某一个绝对的地址（而非相对下一条指令起始地址的偏移量）时，要在实现时灵活指定指令长度为0，来规避
`cpu.eip += instr_len`

实验目标



```
File Edit View Search Terminal Help
Execute ./testcase/bin/sum.img ./testcase/bin/sum
nemu: HIT GOOD TRAP at eip = 0x00030048
NEMU2 terminated

Execute ./testcase/bin/wanshu.img ./testcase/bin/wanshu
nemu: HIT GOOD TRAP at eip = 0x00030091
NEMU2 terminated

Execute ./testcase/bin/struct.img ./testcase/bin/struct
nemu: HIT GOOD TRAP at eip = 0x0003010c
NEMU2 terminated

Execute ./testcase/bin/string.img ./testcase/bin/string
nemu: HIT GOOD TRAP at eip = 0x0003016a
NEMU2 terminated

Execute ./testcase/bin/hello-str.img ./testcase/bin/hello-str
nemu: HIT GOOD TRAP at eip = 0x00030105
NEMU2 terminated

Execute ./testcase/bin/test-float.img ./testcase/bin/test-float
nemu: HIT BAD TRAP at eip = 0x000300c6
NEMU2 terminated
slt@debian:~/pa2017s
```

- PA 2-1提交方式
 - make submit_pa-2-1
 - 服务器还没修好的话就上传cms备用窗口
- PA 2-1提交截止时间
 - 待定

PA 2-1到此结束

祝大家学习快乐，身心健康！

欢迎大家踊跃参加问卷调查

用于精简指令实现的宏

NEMU模拟指令译码和执行

- 精简指令实现的宏

	0	1	2	3	4	5
0	ADD					
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	eAX, Iv
1	ADC					
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	eAX, Iv
2	AND					
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	eAX, Iv
3	XOR					
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	eAX, Iv

大量指令操作相同，只是操作数的类型和长度不同。

NEMU模拟指令译码和执行

- 精简指令实现的宏
 - 采用前面所示的方法自然能够写出所有指令的实现
 - 但会涉及到大量重复的代码
 - 于是不断使用CP（复制-粘贴）大法来进行编码
- 但是代码复制是很糟糕的！
 - `alu_test.c` – bad example!

20170911有关div测试代码的修正说明

在原框架代码中的nemu/src/cpu/test/alu_test.c中针对div的测试用例，在随机测试部分误用了针对idiv的测试代码。修复方案为改为针对div的测试代码。具体请参见群文件：20170911有关div测试代码的修正说明.txt

讲师 汪亮 发表于 09-11 20:50 86人已读

你猜我是怎么写错的？

NEMU模拟指令译码和执行

- 精简指令实现的宏：许多指令的实现流程固定

1. 声明操作数OPERAND
2. 设置操作数长度data_size
3. 根据操作数类型进行解码 decode
4. 进行数据操作
5. 返回指令长度

```
make_instr_func(mov_i2rm_v) {  
    OPERAND rm, imm;  
    rm.data_size = data_size;  
    int len = 1;  
    len += modrm_rm(eip + 1, &rm);  
  
    imm.type = OPR_IMM;  
    imm.addr = eip + len;  
    imm.data_size = data_size;  
  
    operand_read(&imm);  
    rm.val = imm.val;  
    operand_write(&rm);  
  
    return len + data_size / 8;  
}
```

NEMU模拟指令译码和执行

- 于是在[nemu/include/cpu/instr_helper.h](#)中我们给出了用于精简指令实现的宏

```
#define make_instr_impl_1op(inst_name, src_type, suffix) ...  
#define make_instr_impl_1op_cc(inst_name, src_type, suffix, cc) ...  
#define make_instr_impl_2op(inst_name, src_type, dest_type, suffix) ...  
#define make_instr_impl_2op_cc(inst_name, src_type, dest_type, suffix, cc) ...
```

还有

decode_data_size系列

decode_operand系列

condition系列

宏在预处理阶段被gcc处理，本质就是字符串替换，拿右边的替换左边的，换行必须打上\

NEMU模拟指令译码和执行

- 于是在 `nemu/include/cpu/instr_helper.h` 中我们给出了用于精简指令实现的宏：举个例子

源操作数类型 长度后缀
指令名称 目的操作数类型

```
#define make_instr_impl_2op(inst_name, src_type, dest_type, suffix) \  
    make_instr_func(concat7(inst_name, _, src_type, 2, dest_type, _, suffix)) { // 拼接得到 mov_i2rm_v  
        int len = 1; // opcode 占一字节  
        concat(decode_data_size_, suffix) \  
        concat3(decode_operand, _, concat3(src_type, 2, dest_type)) \  
        print_asm_2(...); // 打印调试信息  
        instr_execute_2op(); \  
        return len; \  
    }
```

`make_instr_impl_2op(mov, i, rm, v)`

NEMU模拟指令译码和执行

- 于是在 `nemu/include/cpu/instr_helper.h` 中我们给出了用于精简指令实现的宏：举个例子

源操作数类型 长度后缀

指令名称 目的操作数类型

```
#define make_instr_impl_2op(inst_name, src_type, dest_type, suffix) \  
    make_instr_func(concat7(inst_name, _, src_type, 2, dest_type, _, suffix)) { //拼接得到mov_i2rm_v  
        int len = 1; //opcode占一字节  
        concat(decode_data_size_, suffix) \  
        concat3(decode_operand, _, concat3(src_type, 2, dest_type)) \  
        print_asm_2(...); //打印调试信息  
        instr_execute_2op(); \  
        return len; \  
    }
```

```
#define decode_data_size_v opr_src.data_size = opr_dest.data_size = data_size;
```

两个全局OPERAND类型的变量，免去创建局部变量的开销

`make_instr_impl_2op(mov, i, rm, v)`

NEMU模拟指令译码和执行

- 于是在 `nemu/include/cpu/instr_helper.h` 中我们给出了用于精简指令实现的宏：举个例子

源操作数类型 长度后缀

指令名称 目的操作数类型

```
#define make_instr_impl_2op(inst_name, src_type, dest_type, suffix) \
    make_instr_func(concat7(inst_name, _, src_type, 2, dest_type, _, suffix)) { //拼接得到mov_i2rm_v
        int len = 1; //opcode占一字节
        concat(decode_data_size_, suffix) \
        concat3(decode_operand, _, concat3(src_type, 2, dest_type)) \
        print_asm_2(...); // 打印调试信息
        instr_execute_2op(); \
        return len; \
    }
```

解码两个操作数并增加len

```
#define decode_operand_i2rm \
    len += modrm_rm(eip + 1, &opr_dest); \
    opr_src.type = OPR_IMM; \
    opr_src.sreg = SREG_CS; \
    opr_src.addr = eip + len; \
    len += opr_src.data_size / 8;
```

`make_instr_impl_2op(mov, i, rm, v)`

NEMU模拟指令译码和执行

- 于是在 `nemu/include/cpu/instr_helper.h` 中我们给出了用于精简指令实现的宏：举个例子

源操作数类型 长度后缀

指令名称 目的操作数类型

```
#define make_instr_impl_2op(inst_name, src_type, dest_type, suffix) \
    make_instr_func(concat7(inst_name, _, src_type, 2, dest_type, _, suffix)) {\ //拼接得到mov_i2rm_v
        int len = 1; \ //opcode占一字节
        concat(decode_data_size_, suffix) \
        concat3(decode_operand, _, concat3(src_type, 2, dest_type)) \
        print_asm_2(...); \ // 打印调试信息
        instr_execute_2op(); \
        return len; \
    }
```

```
static void instr_execute_2op() {
    operand_read(&opr_src);
    opr_dest.val = opr_src.val;
    operand_write(&opr_dest);
}
```

执行函数写在 `mov.c` 中，根据指令的具体操作来实现，`static` 不可少！

`make_instr_impl_2op(mov, i, rm, v)`

NEMU模拟指令译码和执行

- 于是在[nemu/include/cpu/instr_helper.h](#)中我们给出了用于精简指令实现的宏：举个例子

```
static void instr_execute_2op() {  
    operand_read(&opr_src);  
    opr_dest.val = opr_src.val;  
    operand_write(&opr_dest);  
}
```

```
make_instr_impl_2op(mov, r, rm, b)  
make_instr_impl_2op(mov, r, rm, v)  
make_instr_impl_2op(mov, rm, r, b)  
make_instr_impl_2op(mov, rm, r, v)  
make_instr_impl_2op(mov, i, rm, b)  
make_instr_impl_2op(mov, i, rm, v)  
make_instr_impl_2op(mov, i, r, b)  
make_instr_impl_2op(mov, i, r, v)  
make_instr_impl_2op(mov, a, o, b)  
make_instr_impl_2op(mov, a, o, v)  
make_instr_impl_2op(mov, o, a, b)  
make_instr_impl_2op(mov, o, a, v)
```

[nemu/src/cpu/instr/mov.c](#)

NEMU模拟指令译码和执行

- 于是在`nemu/include/cpu/instr_helper.h`中我们给出了用于精简指令实现的宏，一些实用信息（详细用法参阅教程，比较详尽）

`#define make_instr_impl_2op(inst_name, src_type, dest_type, suffix) ...`

- `inst_name`就是指令的名称：mov, add, sub, ...
- `src_type`和`dest_type`是源和目的操作数类型，与`decode_operand`系列宏一致：
 - rm – 寄存器或内存地址 – 对应手册E类型
 - r – 寄存器地址 – 对应手册G类型
 - i – 立即数 – 对应手册I类型
 - m – 内存地址 – 差不多对应手册M类型
 - a – 根据操作数长度对应al, ax, eax – 手册里没有
 - c – 根据操作数长度对应cl, cx, ecx – 手册里没有
 - o – 偏移量 – 对应手册里的O类型
- `suffix`是操作数长度后缀，与`decode_data_size`系列宏一致：
 - b, w, l, v – 8, 16, 32, 16/32位
 - bv – 源操作数为8位，目的操作数为16/32位，特殊指令用到
 - short, near – jmp指令用到，分别指代8位和32位

你可以根据实际需要添加其他的宏或改写已有的宏