PA2 – 指令系统

天津大学

智能与计算学部

魏继增

主要内容

- NEMU中的指令执行过程
- NEMU中对浮点数的支持 — 定点化
- 强化简易调试器
- 程序的加载
- 改变程序的行为（选做）

实验目的

- **实验目的**
  - 掌握i386（IA-32）指令格式
  - 掌握NEMU平台的指令周期

回顾：NEMU启动和模拟的运行环境

工程路径/nemu/src/main.c（注：“~”表示“工程路径/nemu/src”）

~/monitor/monitor.c	~/cpu/reg.c	~/monitor/monitor.c	~/monitor/debug/ui.c
init_monitor() →	**reg_test()** →	**restart()** →	**ui_mainloop()**

初始化monitor
1. 打开log文件
2. 加载字符串表和符号表
3. 编译正则表达式
4. 初始化监视点结构池

测试寄存器结构

模拟“计算器系统的启动”

init_ramdisk()

↓

load_entry()

↓

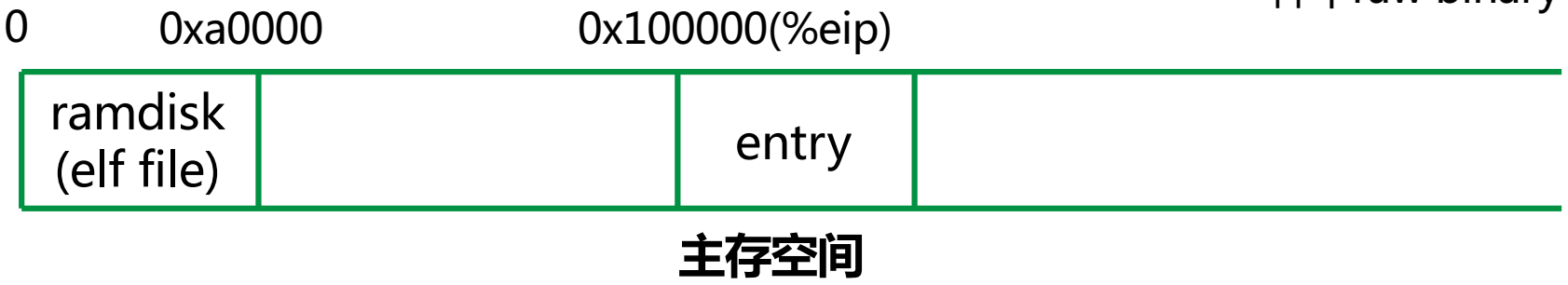
设置%eip初值

↓

init_ddr3()

模拟器主循环，类似GDB，等待用户输入命令。输入“c”，进入指令执行主循环“cpu_exec()”

注：
1. 区别于实际的计算机启动流程
2. entry是可执行文件中raw binary

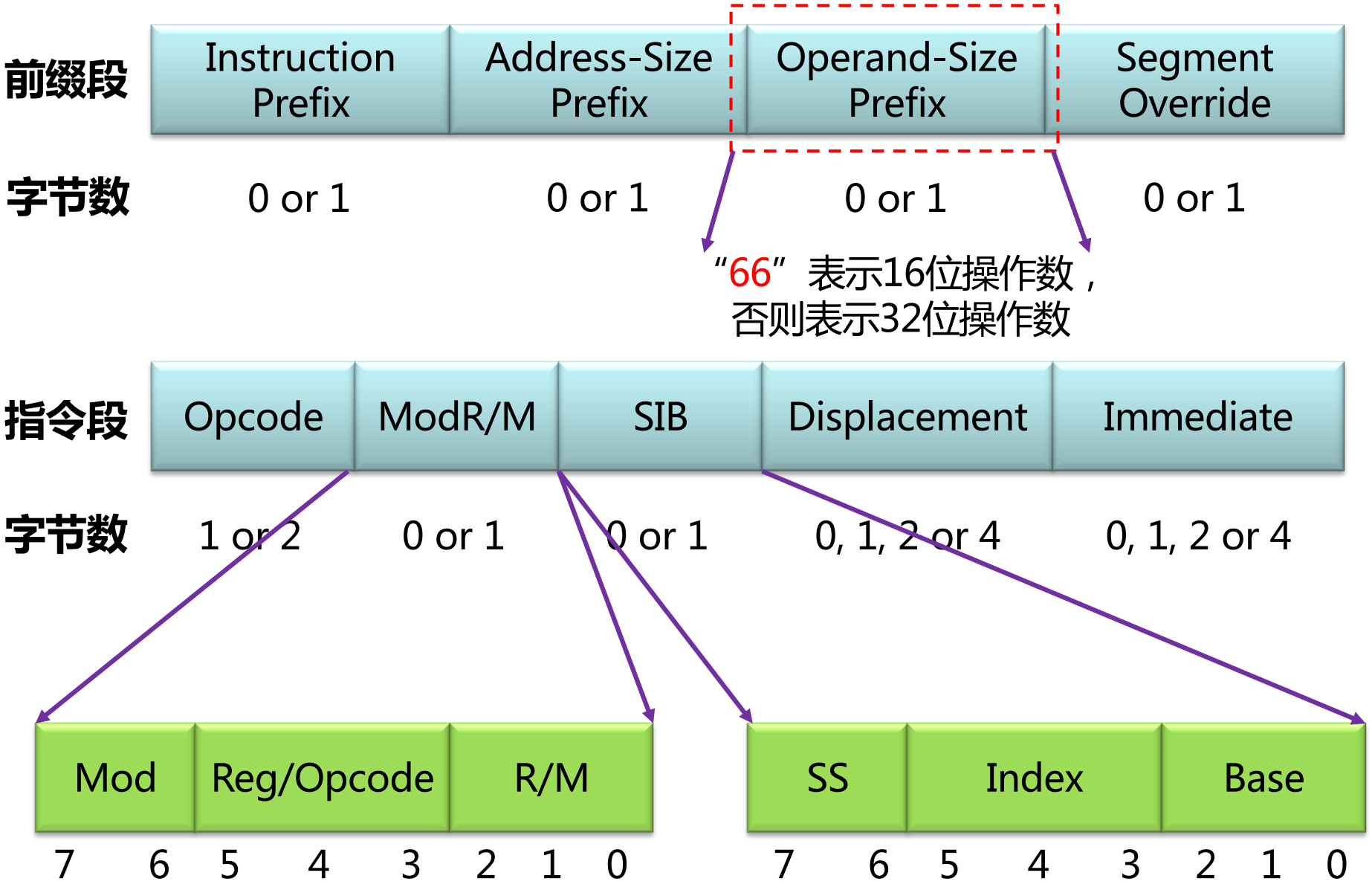


指令执行主循环 — — cpu_exec()

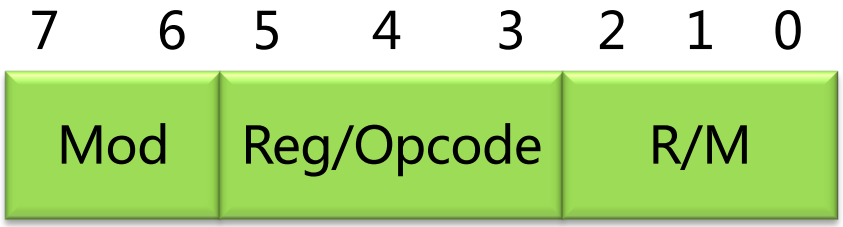
所在文件：~/monitor/cpu-exec.c

```
void cpu_exec(volatile uint32_t n) {
        if(nemu_state == END) {
                printf("Program execution has ended. To restart the .....\n");
              return;
        }
        nemu_state = RUNNING;
        ..........
        setjmp(jbuf);

        for(; n > 0; n --) {
                ..............
                int instr_len = exec(cpu.eip);          //执行当前%eip所指向的指令

                cpu.eip += instr_len;                   //指向下一条指令的%eip
                ................
                if(nemu_state != RUNNING) { return; }
        }

        if(nemu_state == RUNNING) { nemu_state = STOP; }
}
```

i386 (IA-32)的指令格式 - 1



i386 (IA-32)的指令格式 - 2



1. **Mod + R/M:** 32种寻址方式，其中8种寄存器寻址，24中存储器寻址。
2. **Reg/Opcode:** 或表示8个寄存器，或表示3位额外的操作码。

详见Intel 386手册，第244页

1. **SS:** 比例系数。
2. **Index:** 变址寄存器。
3. **Base:** 基址寄存器

详见Intel 386手册，第245页

i386 (IA-32)的指令格式举例

```
1000fe:     66 c7 84 99 00 e0 ff          movw $0x1, -0x2000(%ecx,%ebx,4)
100105:     ff 01 00
```

Instruction Prefix	Address-Size Prefix	Operand-Size Prefix	Segment Override
		66	

Opcode	ModR/M	SIB	Displacement	Immediate
c7	84	99	00 e0 ff ff	01 00

如何阅读i386手册中指令细节

详见Intel 386手册：第17.2节和附录A（opcode map）

Opcode Table

Opcode	Instruction	Clocks	Description
< 1> 88 /r	MOV r/m8,r8	2/2	Move byte register to r/m byte
< 2> 89 /r	MOV r/m16,r16	2/2	Move word register to r/m word
< 3> 89 /r	MOV r/m32,r32	2/2	Move dword register to r/m dword
< 4> 8A /r	MOV r8,r/m8	2/4	Move r/m byte to byte register
< 5> 8B /r	MOV r16,r/m16	2/4	Move r/m word to word register
< 6> 8B /r	MOV r32,r/m32	2/4	Move r/m dword to dword register
< 7> 8C /r	MOV r/m16,Sreg	2/2	Move segment register to r/m word
< 8> 8D /r	MOV Sreg,r/m16	2/5,pm=18/19	Move r/m word to segment register
< 9> A0	MOV AL,moffs8	4	Move byte at (seg:offset) to AL
<10> A1	MOV AX,moffs16	4	Move word at (seg:offset) to AX
<11> A1	MOV EAX,moffs32	4	Move dword at (seg:offset) to EAX
<12> A2	MOV moffs8,AL	2	Move AL to (seg:offset)
<13> A3	MOV moffs16,AX	2	Move AX to (seg:offset)
<14> A3	MOV moffs32,EAX	2	Move EAX to (seg:offset)
<15> B0 + rb ib	MOV r8,imm8	2	Move immediate byte to register
<16> B8 + rw iw	MOV r16,imm16	2	Move immediate word to register
<17> B8 + rd id	MOV r32,imm32	2	Move immediate dword to register
<18> C6 /0 ib (*)	MOV r/m8,imm8	2/2	Move immediate byte to r/m byte
<19> C7 /0 iw (*)	MOV r/m16,imm16	2/2	Move immediate word to r/m word
<20> C7 /0 id (*)	MOV r/m32,imm32	2/2	Move immediate dword to r/m dword

---

NOTES:
moffs8, moffs16, and moffs32 all consist of a simple offset relative to the segment base. The 8, 16, and 32 refer to the size of the data. The address-size attribute of the instruction determines the size of the offset, either 16 or 32 bits.

Opcode Table的阅读 - 1

Tips：i386手册中是Intel格式，objdump默认格式是AT&T格式，两者的源、目的操作数位置不同

	Opcode	Instruction	Clocks	Description
< 1>	88 /r	MOV r/m8, r8	2/2	Move byte register to r/m byte

1. **功能：**将1个8位寄存器中的数据传送到8位的寄存器或者内存中。
2. **“88”**表示opcode，**/r**表示后面跟一个ModR/M字节，并且其中的reg/opcode字段被解释为寄存器的编码。
3. **r8**表示8位寄存器；**r/m8**表示8位寄存器或内存，至于是什么由mod字段决定。

Opcode Table的阅读 - 2

	Opcode	Instruction	Clocks	Description
< 2>	89 /r	MOV r/m16,r16	2/2	Move word register to r/m word
< 3>	89 /r	MOV r/m32,r32	2/2	Move dword register to r/m dword

1. 16位/32位操作的opcode相同，为了避免歧义，通过operand-size前缀进行区分。

2. **operand-size = 66**，表示16位操作数；若该前缀不出现，则操作数默认为32位。

	Opcode	Instruction	Clocks	Description
< 4>	8A /r	MOV r8,r/m8	2/4	Move r/m byte to byte register
< 5>	8B /r	MOV r16,r/m16	2/4	Move r/m word to word register
< 6>	8B /r	MOV r32,r/m32	2/4	Move r/m dword to dword register

Opcode Table的阅读 - 3

	Opcode	Instruction	Clocks	Description
< 7>	8C /r	MOV r/m16,Sreg	2/2	Move segment register to r/m word
< 8>	8D /r	MOV Sreg,r/m16	2/5	Move r/m word to segment register

1. **Sreg**，段寄存器，PA3中需实现这两条指令。

	Opcode	Instruction	Clocks	Description
< 9>	A0	MOV AL,moffs8	4	Move byte at (seg:offset) to AL
<10>	A1	MOV AX,moffs16	4	Move word at (seg:offset) to AX
<11>	A1	MOV EAX,moffs32	4	Move dword at (seg:offset) to EAX
<12>	A2	MOV moffs8,AL	2	Move AL to (seg:offset)
<13>	A3	MOV moffs16,AX	2	Move AX to (seg:offset)
<14>	A3	MOV moffs32,EAX	2	Move EAX to (seg:offset)

1. **moffs**，表示段内偏移，PA2中没有段的概念，可理解为“相对物理地址0的偏移”。

2. 没有ModR/M字节，可以让displacement直接跟在opcode之后，指示内存地址。

Opcode Table的阅读 - 4

Opcode	Instruction	Clocks	Description
<15> B0 + rb ib	MOV r8,imm8	2	Move immediate byte to register
<16> B8 + rw iw	MOV r16,imm16	2	Move immediate word to register
<17> B8 + rd id	MOV r32,imm32	2	Move immediate dword to register

1. **功能：**将1个立即数转送到通用寄存器中。

2. **“+rb, +rw, +rd”**分别表示8位、16位和32位通用寄存器编码。和ModR/M中的reg字段不同，它们表示**直接将寄存器编号按数值加到opcode中**，可通过操作码低3位确定一个寄存器操作数。

3. **“ib”**表示8位立即数；**“iw”**表示16位立即数；**“id”**表示32位立即数。

Opcode Table的阅读 - 5

Opcode	Instruction	Clocks	Description
<18> C6 /0 ib	MOV r/m8,imm8	2/2	Move immediate byte to r/m byte
<19> C7 /0 iw	MOV r/m16,imm16	2/2	Move immediate word to r/m word
<20> C7 /0 id	MOV r/m32,imm32	2/2	Move immediate dword to r/m dword

1. **功能：**将1个立即数转送到通用寄存器或内存中。

2. **“/digit”**，digit为0～7 中一个数字（/0），表示操作码后跟一个ModR/M字节，并且**reg/opcode字段被解释为扩展opcode，取值为digit**。（附录A，416页）

其他信息

Operation

```
DEST ← DEST + SRC;
```

Description

ADD performs an integer addition of the two operands (DEST and SRC). The result of the addition is assigned to the first operand (DEST), and the flags are set accordingly.

When an immediate byte is added to a word or doubleword operand, the immediate value is sign-extended to the size of the word or doubleword operand.

Flags Affected

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

Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

NEMU中指令的执行

重点关注“工程路径/nemu/include/cpu”和“工程路径/nemu/src/cpu”

关键的宏定义：

- #define **make_helper**(name)
  //位于“nemu/include/cpu/helper.h”

- #define **make_helper_v**(name)
  //位于“nemu/include/cpu/exec/helper.h”

- #define **make_instr_helper**(type)
  //位于“nemu/include/cpu/exec/helper.h”

NEMU如何执行指令？——exec()

exec()函数在哪个文件中？ 工程路径/nemu/src/cpu/exec/exec.c

```
make_helper(exec) {
        ops_decoded.opcode = instr_fetch(eip, 1);
        return opcode_table[ ops_decoded.opcode ](eip);
}
```

#define **make_helper**(name) **int name(swaddr_t eip)**

该宏定义一系列“helper”函数，每个“helper”函数对%eip指向内存单元进行操作，并返回操作涉及的代码长度。

```
int exec (swaddr_t eip) {
        ops_decoded.opcode = instr_fetch(eip, 1);
        return opcode_table[ ops_decoded.opcode ](eip);
}
```

保存译码相关的信息，如操作码，源操作数，目的操作数等

NEMU的指令周期 — — 取指IF

函数**instr_fetch( )**负责取指工作，位于“nemu/include/cpu/helper.h”

```
static inline uint32_t instr_fetch(swaddr_t addr, size_t len) {
        return swaddr_read(addr, len);
}
```

NEMU的指令周期 — — 译码ID（1）

1. 确定是哪一条指令的哪一种形式（opcode）
2. 确定操作数（ModR/M, SIB, Displacement以及Immediate）

```
int exec (swaddr_t eip) {
        ops_decoded.opcode = instr_fetch(eip, 1);
        return opcode_table[ ops_decoded.opcode ](eip);
}
```

```
helper_fun opcode_table [256] = {...};
```

```
typedef int (*helper_fun)(swaddr_t);
```

opcode_table数组中每一个元素是一个函数指针（**helper函数**），
对应某条指令的某种形式

NEMU的指令周期——译码ID（2）

```
100014:    b9 00 80 00 00          mov $0x8000,%ecx
......
1000fe:    66 c7 84 99 00 e0 ff    movw $0x1,-0x2000(%ecx,%ebx,4)
100105:    ff 01 00
```

- 首先通过instr_fetch()取得指令的第一个字节0xb9；
- 根据字节0xb9索引opcode_table，找到一个名为mov_i2r_v( )的helper函数
- 指令所对应的helper函数的通用命名形式：指令_形式_操作数后缀
  - 指令：mov；
  - 形式：*i2r*，将立即数移动到寄存器；
  - 操作数后缀：*b*表示操作数长度为8；*v*表示通过操作码无法确定操作数长度，可能是16或32，需要通过ops_decoded.is_data_size_16变量来确定，这个变量就是通过operand-size前缀进行设置。
- 索引opcode_table有两种特殊情况：(1). 两字节转义码0x0f，需要两个字节确定指令形式；(2). ModR/M字节中的reg/opcode字段对opcode进行扩充（指令组）。

NEMU的指令周期 — — 译码ID（3）

1. 确定是哪一条指令的哪一种形式（opcode）
2. 确定操作数（ModR/M, SIB, Displacement以及Immediate）

- 执行mov_i2r_v( )函数，where？
- “nemu/src/cpu/exec/data-mov”目录中和mov指令相关的三个文件：
  - **mov.c**, **mov.h**, **mov-template.c**（大多数指令类型都要实现这三个文件）
- 函数mov_i2r_v( )的定义在文件“**mov.c**”中 — — **make_helper_v**(mov_i2r)

```
#define make_helper_v(name) \
    make_helper(concat(name, _v)) { \
      return (ops_decoded.is_data_size_16 ? concat(name, _w) : concat(name, _l)) (eip); \
}
```

宏展开

```
int mov_i2r_v (swaddr_t eip) {
    return (ops_decoded.is_data_size_16? mov_i2r_w : mov_i2r_l) (eip);
}
```

NEMU的指令周期 — — 译码ID（4）

- 执行mov_i2r_l( ) 函数，where ?
- 函数mov_i2r_l( )定义在文件“**mov-template.h**”中 — — **make_instr_helper**(i2r)

```
#define do_execute concat4(do_, instr, _, SUFFIX)
#define make_instr_helper(type) \
        make_helper(concat5(instr, _, type, _, SUFFIX)) { \
                return idex(eip, concat4(decode_, type, _, SUFFIX), do_execute); \
}
```

宏展开

```
int mov_i2r_l (swaddr_t eip) {
    return idex(eip, decode_i2r_l, do_mov_l);
}
```

idex( )的原型为（ nemu/include/cpu/helper.h ）：

```
static inline int idex(swaddr_t eip, int (*decode)(swaddr_t), void (*execute) (void)) {

        int len = decode(eip + 1);   // decode函数对eip指向的指令进行译码
        execute();                   // execute函数执行这条指令
        return len + 1;              // 返回该条指令的字节长度
}
```

- “mov-template.h”文件会在“mov.c”中被引用3次，分别针对3种操作数长度

NEMU的指令周期 — — 译码ID（5）

- 执行decode_i2r_l( )函数，where？
- “nemu/src/cpu/decode”目录有3个和译码相关的文件：
  - **decode.c**
  - **decode-template.h**

  各种形式译码函数的定义，decode-template.h在decode.c中被引用3次，对应不同长度的操作数
  - **modrm.c**

  对ModR/M和SIB字段的译码
- “nemu/inlcude/cpu/decode”文件夹有1个和译码相关的文件：
  - **decode.h** 各种形式译码函数的声明，采用make_helper宏
- decode_i2r_l( )函数定义在“**decode-template.h**”

```
make_helper(concat(decode_i2r_, SUFFIX)) {
        decode_r_internal(eip, op_dest);
        return decode_i(eip);
}
```

- 译码所得的操作数信息分别存储在**op_src**和**op_dest**两个宏中，这两个宏定义在“nemu/include/cpu/helper.h”文件中

NEMU的指令周期 — — 执行EX

```
int mov_i2r_l (swaddr_t eip) {
    return idex(eip, decode_i2r_l, do_mov_l);
}
```

- 执行do_mov_l( )函数，where ?
- 函数do_mov_l( )定义在文件“**mov-template.h**”中

```
static void do_execute() {
        OPERAND_W(op_dest, op_src->val);
        print_asm_template2();
}
```

宏展开

```
static void do_mov_l() {
        write_operand_l((&ops_decoded.dest), (&ops_decoded.src)->val);
         print_asm_template2();
}
```

mov $0x8000，%ecx

(op_src->val) (op_dest)

规律和体会

- 对于同一指令的不同形式，它们的执行阶段是相同的。如mov_i2rm和mov_rm2r，它们的**执行阶段**都是将源操作数存储目的操作数中。

- 对于不同指令的同一种形式，它们的译码阶段是相同的。如mov_i2rm和sub_i2rm，它们的**译码阶段**都是识别出一个立即数和一个rm操作数。

- 对于同一条指令同一种形式的不同长度，它们的**译码阶段和执行阶段**都是非常类似的。如mov_i2rm_b，mov_i2rm_w和mov_i2rm_l，它们都是识别出一个立即数和一个 rm操作数，然后把立即数存入rm操作数。

实现封装和抽象，尽可能复用，框架代码中使用了大量的宏，见**实验手册第15页**

```
make cpp     //生成 .i 的预处理结果
```

nemu/src/cpu/exec目录

如何在NEMU中添加指令？

以添加ADD指令为例：

- 在nemu/src/cpu/exec/exec.c文件的**opcode_table数组**内与ADD指令操作码相对应的元素处填写ADD指令所有形式的helper函数

若使用转义码，则需要在**_2byte_opcode_table数组**内填写相应指令的helper函数

若使用ModR/M中的reg/opcode字段对opcode进行扩充，则填写相应的**指令组**

- nemu/src/cpu/exec/arith目录下添加三个文件：
  - **add.h**（add指令所有形式helper函数的声明）
  - **add.c**（add指令所有形式helper函数的定义）
  - **add-template.h**（ add指令所有形式helper函数定义的模板）

- 在nemu/src/cpu/exec/all-instr.h文件中**添加对“add.h”的引用**

- 若nemu/src/cpu/decode/decode-template.h中提供的译码helper函数无法满足译码要求（应该没有），则**定义新的译码helper函数**，并在nemu/include/cpu/decode/decode.h中**添加新译码函数的声明**

如何运行测试程序？（1）

- 所有测试程序位于“工程
- 修改工程目录下的Make

```
-USERPROG = obj/testcas
+USERPROG = obj/testcas
ENTRY = $(USERPROG)
```

- 修改后，在终端输入命令

```
invalid opcode(eip = 0x0010000a): e8 06 00 00 00 b8 00 00 ...

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 value(0x0010000a) in the disassembling result to distinguish which case it

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

for more details.

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

nemu: nemu/src/cpu/exec/special/special.c:24: inv: Assertion `0' failed.
```

- 若输出如下信息，则表示NEMU不支持相应指令，需要在NEMU中添加；
- obj/testcase路径下可以找到测试用例的反汇编代码，例如mov-c.txt
- 根据反汇编代码定位不支持的指令，最终通过修改nemu代码实现该指令

如何运行测试程序？（2）

- **如何证明测试程序运行成功呢？**
  - 将运算结果打印到屏幕上
  - 利用简易调试器中扫描内存的功能
  - **自动比对结果？**

× 目前NEMU不支持用户程序的输出！

√ 但人工检查太麻烦！

- 利用断言assertion，但不能使用标准库assertion()，使用自定义的 **nemu_assert()**函数，定义在“工程路径/lib-common/trap.h”

```
#define HIT_GOOD_TRAP \
        asm volatile(".byte 0xd6" : : "a" (0))
#define HIT_BAD_TRAP \
        asm volatile(".byte 0xd6" : : "a" (1))
#define nemu_assert(cond) \
        do { \
                if( !(cond) ) HIT_BAD_TRAP; \
        } while(0)
```

内联汇编

nemu_trap指令

“nemu/src/cpu/exec/special/special.c”

- 运行测试程序，当终端输出“**HIT GOOD TRAP**”，则表示测试通过

添加指令的注意事项 - - 如何调试

- 对照测试程序的反汇编代码，可以在两个地方找到：
  - ✓ obj/testcase/xxx.txt（每次make run后会自动生成）
  - ✓ 工程目录/log.txt（测试程序的执行代码流，对于新增加的指令需要自己添加相应的打印语句）
- 灵活运用PAI中的调试命令，如si、info、x等
  - ✓ 迅速定位到错误附附近，共有3种方法
    - ➢ 在C程序中添加断点，使用set_bp ()函数
    - ➢ 通过“si N”跳过已正确实现指令，迅速定位到错误附近
    - ➢ 通过“w %eip == xxxxx”迅速定位到错误附近
  - ✓ 在错误附近精准定位具体错误
    - ➢ 首先，通过“si”单步调试
    - ➢ 再通过“info r”或“x N Expr”打印寄存器或存储器中的值，判断是否与预期结果一致

其他注意事项

- 仔细阅读实验手册、i386手册以及附件！
- 每次添加新指令，都需要对照一下“附件6 - i386手册堪错表”，确定指令的最终正确表述。
- 仔细阅读代码框架，注释部分也要特别留意！

主要内容

- NEMU中的指令执行过程
- NEMU中对浮点数的支持 — 定点化
- 强化简易调试器
- 程序的加载
- 改变程序的行为（选做）

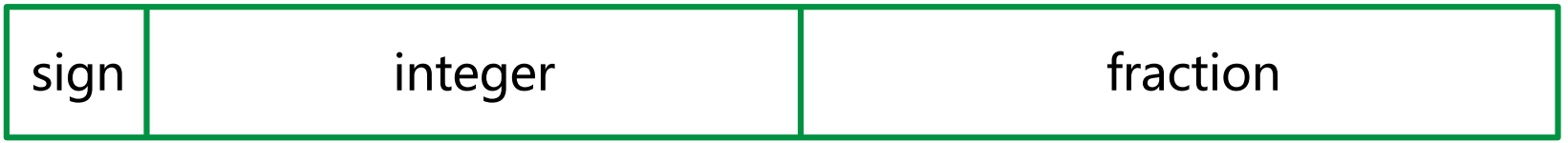
实验目的

- **实验目的**
  - 掌握浮点数的表示和编码
  - 理解机器数和真值的关系

Binary Scaling

- 用**整数**表示实数，该方法表示的实数记为FLOAT；
- 采用32位整数表示的FLOAT类型：
  - 最高位为**符号位**；
  - 接下来的15位表示**整数部分**；
  - 低16位表示**小数部分**；
  - （约定）小数点在15位和16位之间（**定点化**）。

32位FLOAT类型（int）　　牺牲“表数范围和精度”换取“速度”



- 对于实数a，其FLOAT类型A = a × 2^{16}（**仅保留小数点后的16位**）

Binary Scaling（举例-1）

实数1.2和5.6用FLOAT类型近似表示：

$1.2 \times 2^{16} = 78643 = 0x13333$

$5.6 \times 2^{16} = 367001 = 0x59999$

0	1	3 3 3 3

0	5	9 9 9 9

而实际上，这两个FLOAT类型数据表示的数是：

$0x13333 / 2^{16} = 1.19999695$

$0x59999 / 2^{16} = 5.59999084$

Binary Scaling（举例-2）

对于负数，例如-1.2的FLOAT类型表示为：

$-(1.2 \times 2^{16}) = -0x13333 = 0xfffecccd$

1	7 f f e	c c c d

注意：由于整数部分只有15位，因此，要特别小心对于溢出的处理！

FLOAT类型常见的运算

假设实数a，b的FLOAT类型表示分别为A，B

- 由于FLOAT类型采用整数表示，FLOAT类型的加法可以直接采用整数加法

$$A + B = a \times 2^{16} + b \times 2^{16} = (a + b) \times 2^{16}$$

- 由于FLOAT类型也采用补码，FLOAT类型的减法可以直接采用整数减法

$$A - B = a \times 2^{16} - b \times 2^{16} = (a - b) \times 2^{16}$$

- FLOAT类型的乘除法和加减法不同，要特别注意

$$A \times B = a \times 2^{16} \times b \times 2^{16} = (a \times b) \times 2^{32} \; != \; (a \times b) \times 2^{16}$$

乘法的正确结果：

$$FLOAT(a \times b) = (A \times B) / 2^{16}$$

除法的正确结果：

$$FLOAT(a / b) = (A / B) \times 2^{16}$$

NEMU中与FLOAT类型相关的函数

```
/* lib-common/FLOAT.h */
int32_t F2int(FLOAT a);
FLOAT int2F(int a);
FLOAT F_mul_int(FLOAT a, int b);
FLOAT F_div_int(FLOAT a, int b);

/* lib-common/FLOAT.c */
FLOAT f2F(float a);     // a是浮点数，不能直接乘2^16，怎么办？
FLOAT F_mul_F(FLOAT a, FLOAT b);
FLOAT F_div_F(FLOAT a, FLOAT b);
FLOAT Fabs(FLOAT a);
FLOAT sqrt(FLOAT x);
FLOAT pow(FLOAT x, FLOAT y)  //PA4会用到
```

FLOAT类型相关函数的使用

```
float a = 1.2;
float b = 10;
int c = 0;
if(b > 7.9) {
    c = (a + 1) * b / 2.3;
}
```

```
FLOAT a = f2F(1.2);
FLOAT b = int2F(10);
int c = 0;
if(b > f2F(7.9)) {
c = F2int(F_div_F(F_mul_F((a + int2F(1)), b), f2F(2.3)));
}
```

主要内容

- NEMU中的指令执行过程
- NEMU中对浮点数的支持 — 定点化
- 强化简易调试器
- 程序的加载
- 改变程序的行为（选做）

ELF目标文件 (1)

- ELF目标文件：**可重定位**目标文件和**可执行**目标文件

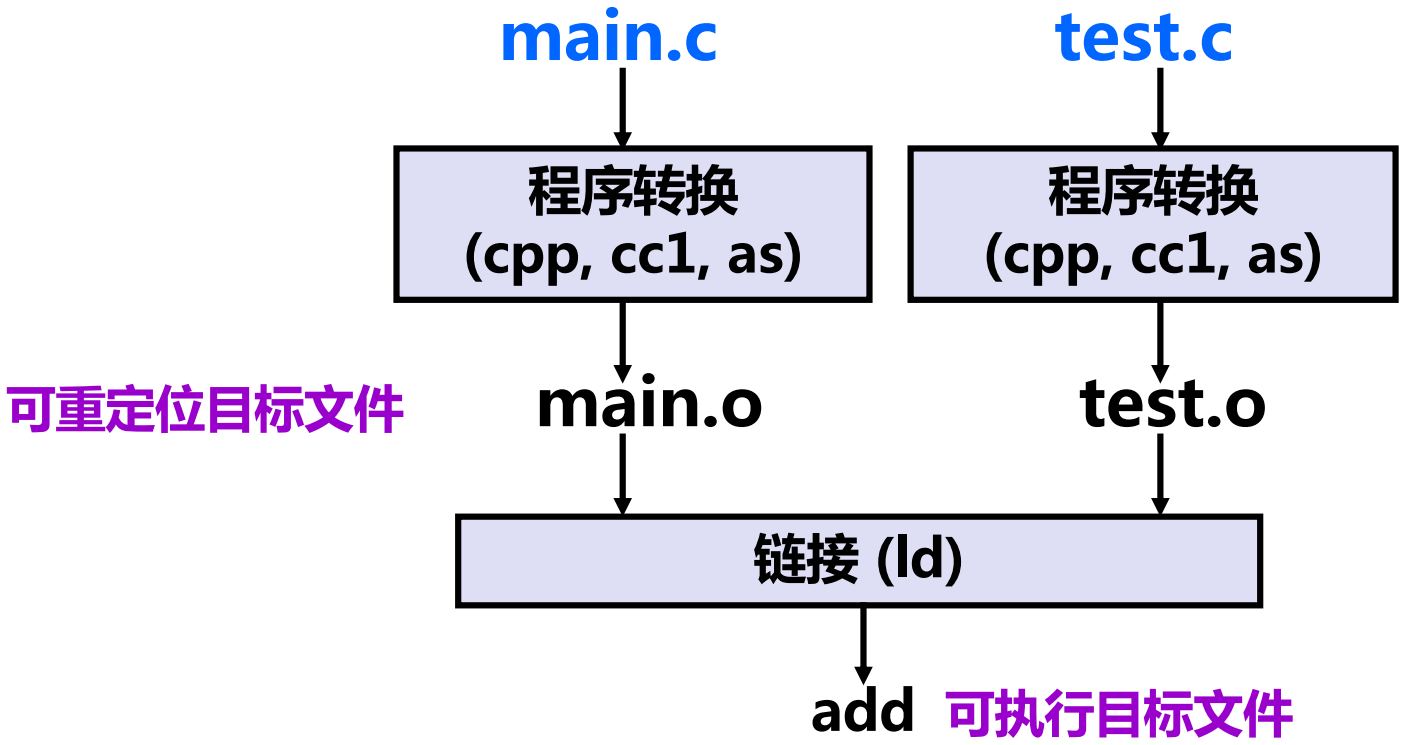
```
/* main.c */
int add(int, int);
int main( )
{
   return add(20, 13);
}
```

```
/* test.c */
int add(int i, int j)
{
    int x = i + j;
    return x;
}
```

使用GCC编译器**编译**并**链接**生成**可执行程序**add

```
-$ gcc -g -o add main.c test.c
-$ ./add
```

ELF目标文件 (2)



- cpp：**预处理**，包括包含头文件，宏展开，条件编译的选择等。
- cc1：**编译**，将C源程序翻译成汇编程序。
- as：**汇编**，将汇编程序转换为机器语言代码。

objdump -d test.o

```
00000000 <test>:
       0:   55              push  %ebp
       1:   89 e5           mov   %esp, %ebp
       3:   83 ec 10        sub   $0x10, %esp
       6:   8b 45 0c        mov    0xc(%ebp), %eax
       9:   8b 55 08        mov    0x8(%ebp), %edx
       c:   8d 04 02        lea    (%edx,%eax,1), %eax
       f:   89 45 fc        mov    %eax, -0x4(%ebp)
      12:   8b 45 fc        mov    -0x4(%ebp), %eax
      15:   c9              leave
      16:   c3              ret
```

objdump -d add

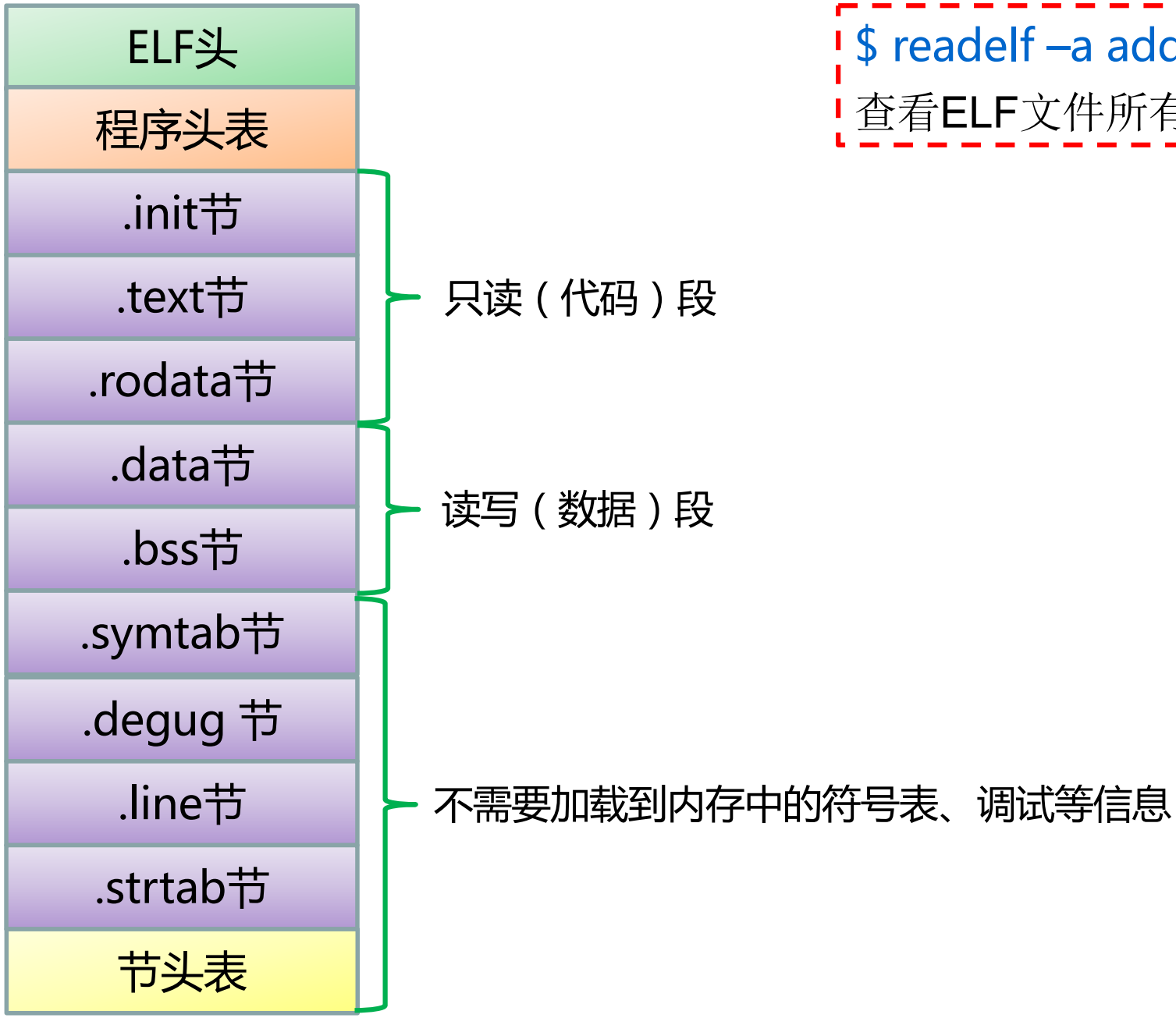
```
080483d4 <add>:
 80483d4:   55              push  %ebp
 80483d5:   89 e5           mov   %esp, %ebp
 80483d7:   83 ec 10        sub   $0x10, %esp
 80483da:   8b 45 0c        mov    0xc(%ebp), %eax
 80483dd:   8b 55 08        mov    0x8(%ebp), %edx
 80483e0:   8d 04 02        lea    (%edx,%eax,1), %eax
 80483e3:   89 45 fc        mov    %eax, -0x4(%ebp)
 80483e6:   8b 45 fc        mov    -0x4(%ebp), %eax
 80483e9:   c9              leave
 80483ea:   c3              ret
```

ELF可重定位目标文件格式

- **节（section）**是 ELF 文件中具有相同特征的最小处理单位
- **.text节**: 代码部分
- .rodata节：只读数据部分，如printf中的格式串等
- **.data节**：已初始化的全局变量
- **.bss节**：未初始化全局变量，但不占用磁盘空间
- **.symtab节**：程序中符号（函数和全局变量）的相关信息
- .rel.text节：和.text节相关的可重定位信息
- .rel.data节：和.data节相关的可重定位信息
- .debug节：调试用符号表（gcc -o）
- .line节：C程序中行号和.text节中机器指令之间的映射
- **.strtab节**：字符串表

- ELF头：ELF文件结构相关说明信息
- 节头表：每个节的节名、偏移和大小

ELF可执行目标文件格式



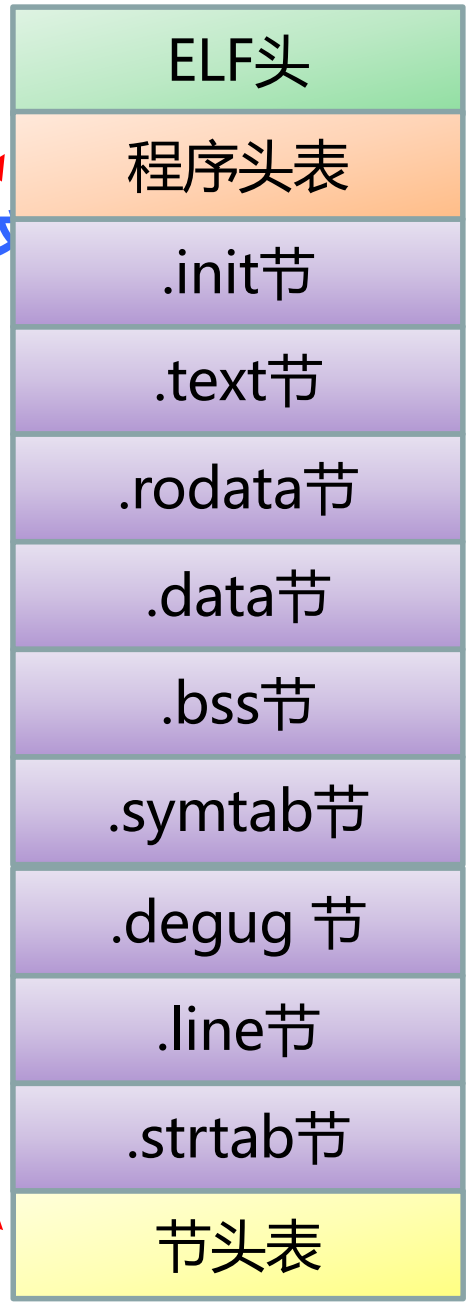
$ readelf –a add

查看ELF文件所有信息

ELF头（1）

```
$ readelf -h main
ELF Header:
  Magic:  7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8048580
  Start of program headers:          52 (bytes into file)
  Start of section headers:          3232 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         8
  Size of section headers:           40 (bytes)
  Number of section headers:         29
  Section header string table index: 26
```

ELF头位于目标文件起始位置，包含文



ELF头（2）

库函数“elf.h”定了与ELF文件相关的数据结构，ELF头对应结构体如下

```
#define EI_NIDENT       16
typedef struct {
        unsigned char   e_ident [EI_NIDENT];
        Elf32_Half      e_type;
        Elf32_Half      e_machine;
        Elf32_Word      e_version;
        Elf32_Addr      e_entry;
        Elf32_Off       e_phoff;
        Elf32_Off       e_shoff;
        Elf32_Word      e_flags;
        Elf32_Half      e_ehsize;
        Elf32_Half      e_phentsize;
        Elf32_Half      e_phnum;
        Elf32_Half      e_shentsize;
        Elf32_Half      e_shnum;
        Elf32_Half      e_shstrndx;
} Elf32_Ehdr;
```

通过命令“man 5 elf”进行查阅

实验目的和要求

- **实验目的**
  - 掌握符号、符号表以及字符串表
  - 理解过程调用的机器级表示，掌握栈帧及其结构

强化调试器（1）——变量支持

- 现在NEMU可以运行C程序，相比汇编C程序多了变量和函数的要素，那如何在PA1实现的调试器的表达式求值中支持变量的输入呢？

```
(nemu) p test_data
```

- 也是说，如何从***test_data***这个字符串找到这个变量运行时的信息？

符号表（Symbol Table）：是可执行ELF文件的一个section，它记录了程序中变量和函数的信息，这些信息都有哪些？

以**add.c**测试程序为例，通过命令“readelf –a add”查看ELF可执行文件信息

符号表（Symbol Table）

什么才算是一个符号（Symbol）？

```
Symbol table '.symtab' contains 10 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00100000     0 SECTION LOCAL  DEFAULT    1
     2: 0010009c     0 SECTION LOCAL  DEFAULT    2
     3: 00100100     0 SECTION LOCAL  DEFAULT    3
     4: 00000000     0 SECTION LOCAL  DEFAULT    4
     5: 00000000     0 FILE    LOCAL  DEFAULT  ABS add.c
     6: 00100084    22 FUNC    GLOBAL DEFAULT    1 add        7
     7: 00100000   129 FUNC    GLOBAL DEFAULT    1 main      11
     8: 00100120   256 OBJECT  GLOBAL DEFAULT    3 ans       16
     9: 00100100    32 OBJECT  GLOBAL DEFAULT    3 test_data 20
```

- 符号表建立了**变量名**和**地址**之间的映射关系；
- 注意，readelf输出的信息是解析过的，实际上“**Name**”属性存放的是字符串在字符串表（string table）中的偏移量。

符号表表项的数据结构

```
typedef struct {
        Elf32_Word      st_name;
        Elf32_Addr      st_value;
        Elf32_Word      st_size;
        unsigned char   st_info;
        unsigned char   st_other;
        Elf32_Half      st_shndx;
} Elf32_Sym;
```

- st_name: 符号在字符串表中的索引（字节偏移量）；
- st_value: 符号所在地址，在可重定位文件中是符号所在位置相对于所在节起始位置的字节偏移量，在可执行文件中则是符号的虚拟地址；
- st_size: 符号所表示对象的字节数；
- st_info：符号的类型和绑定属性；
- st_other：符号的可见性，通常出现在可重定位目标文件中；
- st_shndx：符号所在节在节头表中的索引

字符串表（String Table）

先查看readelf输出中**Section Headers**中的信息：

```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00100000 001000 00009a 00  AX  0   0  4
  [ 2] .eh_frame         PROGBITS        0010009c 00109c 000058 00   A  0   0  4
  [ 3] .data             PROGBITS        00100100 001100 000120 00  WA  0   0 32
  [ 4] .comment          PROGBITS        00000000 001220 00001c 01  MS  0   0  1
  [ 5] .shstrtab         STRTAB          00000000 00123c 00003a 00      0   0  1
  [ 6] .symtab           SYMTAB          00000000 0013b8 0000a0 10      7   6  4
  [ 7] .strtab           STRTAB          00000000 001458 00001e 00      0   0  1
```

- 字符串表在ELF文件偏移为“**0x1458**”的位置开始存放通过命令；
- “**hd add**”输出ELF文件的16进制格式

```
00001450  20 00 00 00 11 00 03 00  00 61 64 64 2e 63 00 61  | ........add.c.a|
00001460  64 64 00 6d 61 69 6e 00  61 6e 73 00 74 65 73 74  |dd.main.ans.test|
00001470  5f 64 61 74 61 00                                |_data.|
```

符号表和字符串表的关系

在表达式求值的过程中，如果发现token的类型为一个变量标识符：

(nemu) p test_data

(nemu) 0x100100

注：因为未实现类型系统，对于基本类型变量只返回地址，对于整形变量x，可通过命令“p *x”输出其取值。

```
Section Headers:
[Nr]      Name      Type      Addr      Off      Size      ES  Flg  Lk  Inf  Al
[ 7]      .strtab   STRTAB    00000000  001458   00001e    00       0    0    1
```

ans test_data

```
00001450   20 00 00 00 11 00 03 00   00 61 64 64 2e 63 00 61 | ........add.c.a|
00001460   64 64 00 6d 61 69 6e 00   61 6e 73 00 74 65 73 74 |dd.main.ans.test|
00001470   5f 64 61 74 61 00                               |_data.|
```

```
Symbol table '.symtab' contains 10 entries:
   Num:    Value      Size  Type    Bind    Vis      Ndx  Name
     5:   00000000     0     FILE    LOCAL   DEFAULT  ABS    1
     6:   00100084     22    FUNC    GLOBAL  DEFAULT  1      7
     7:   00100000     129   FUNC    GLOBAL  DEFAULT  1     11
     8:   00100120     256   OBJECT  GLOBAL  DEFAULT  3     16
     9:   00100100     32    OBJECT  GLOBAL  DEFAULT  3     20
```

如何添加变量支持

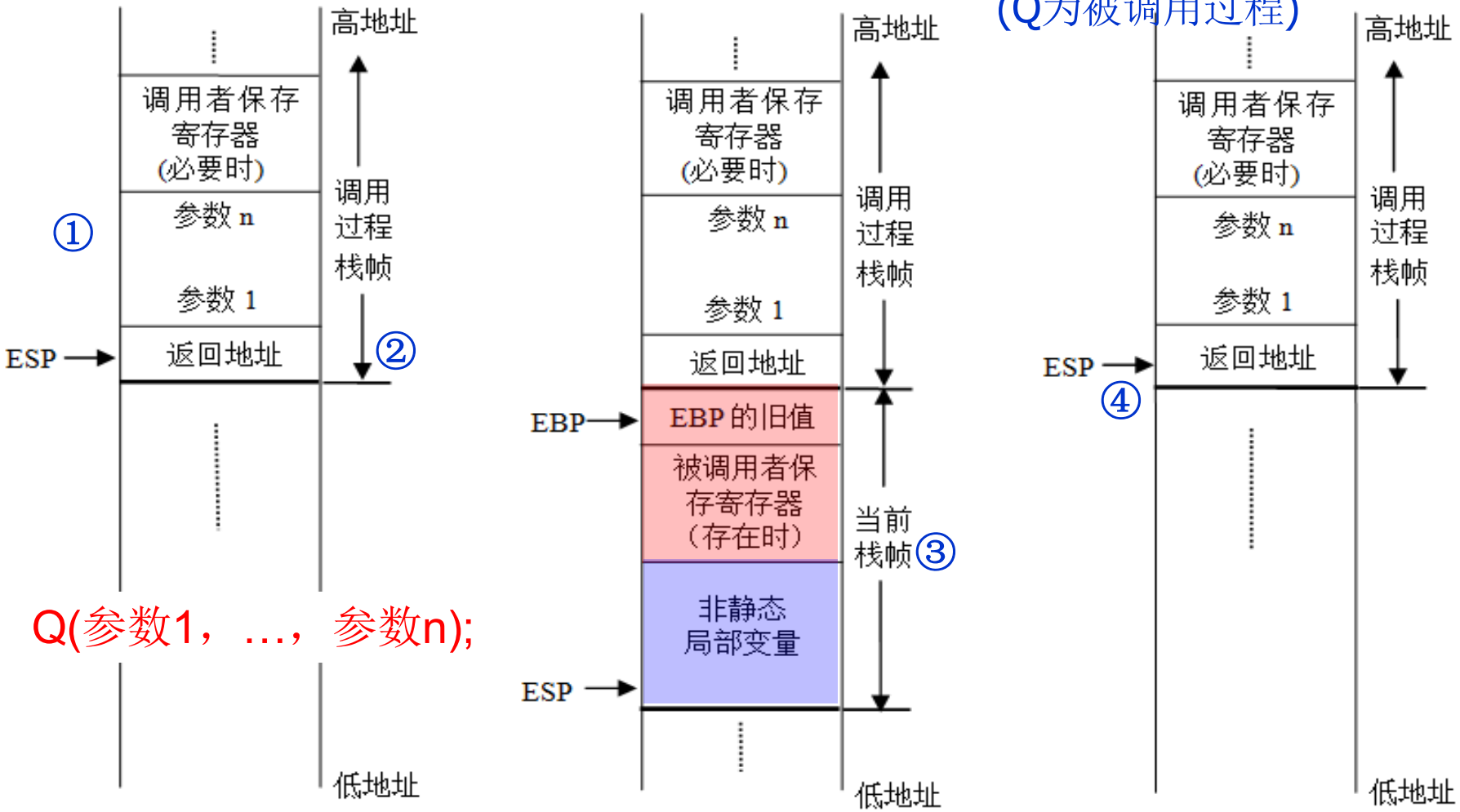
- 在表达式求值的词法分析和递归求值中添加对变量的识别和处理；

- **load_table()**函数（ ~/monitor/debug/elf.c ）已提取符号表和字符串表；
  - strtab：字符串表
  - symtab：符号表
  - nr_symtabl_entry：符号表的表项数目

- 头文件**<elf.h>**定义了与ELF文件相关的数据结构，以便对符号表进行操作，可通过命令“man 5 elf”进行查阅

IA-32过程调用的机器级表示

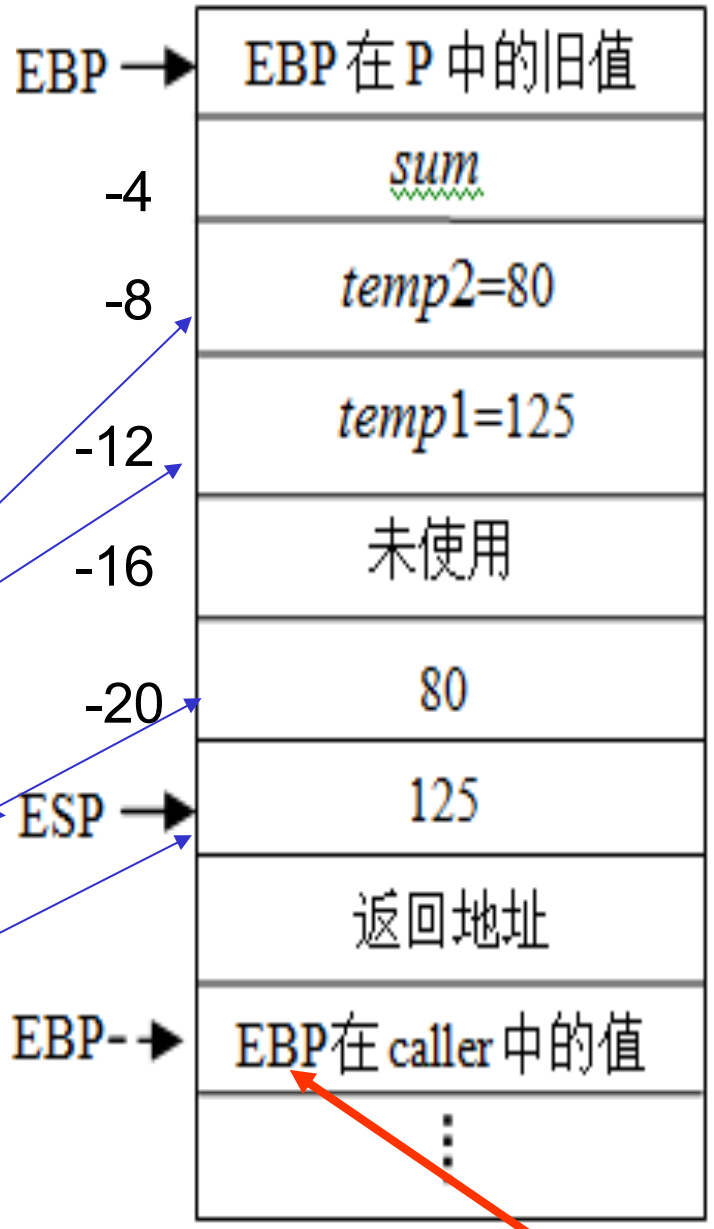
过程调用过程中
栈和栈帧的变化
(Q为被调用过程)



```
int add ( int x, int y ) {
      return x+y;
}
int   caller ( ) {
      int    t1 = 125;
      int    t2 = 80;
      int    sum = add (t1, t2);
      return sum;
}
```

add

caller



caller：

```
pushl    %ebp
movl     %esp, %ebp
subl     $24, %esp
movl     $125, -12(%ebp)
movl     $80, -8(%ebp)
movl     -8(%ebp), %eax
movl     %eax, 4(%esp)
movl     -12(%ebp), %eax
movl     %eax, (%esp)
call     add
movl     %eax, -4(%ebp)
movl     -4(%ebp), %eax
leave
ret
```

准备阶段

分配局部变量

准备入口参数

返回参数总在EAX中

准备返回参数

结束阶段

```
movl   %ebp, %esp
popl   %ebp
```

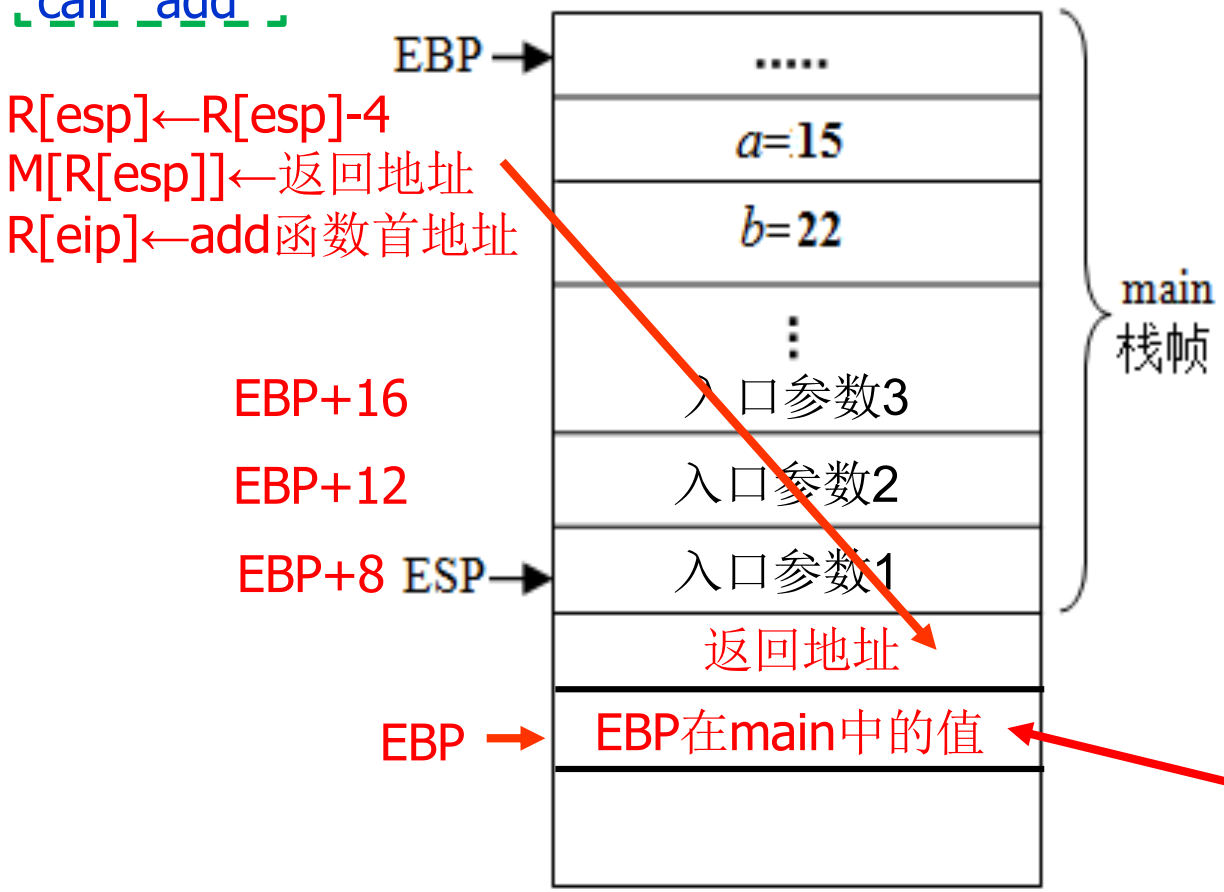
add函数开始是什么？

```
pushl   %ebp
movl    %esp, %ebp
```

入口参数的位置

```
movl  参数3，8(%esp)
.....
movl  参数1, (%esp)
call  add
```

准备入口参数



返回地址是什么？

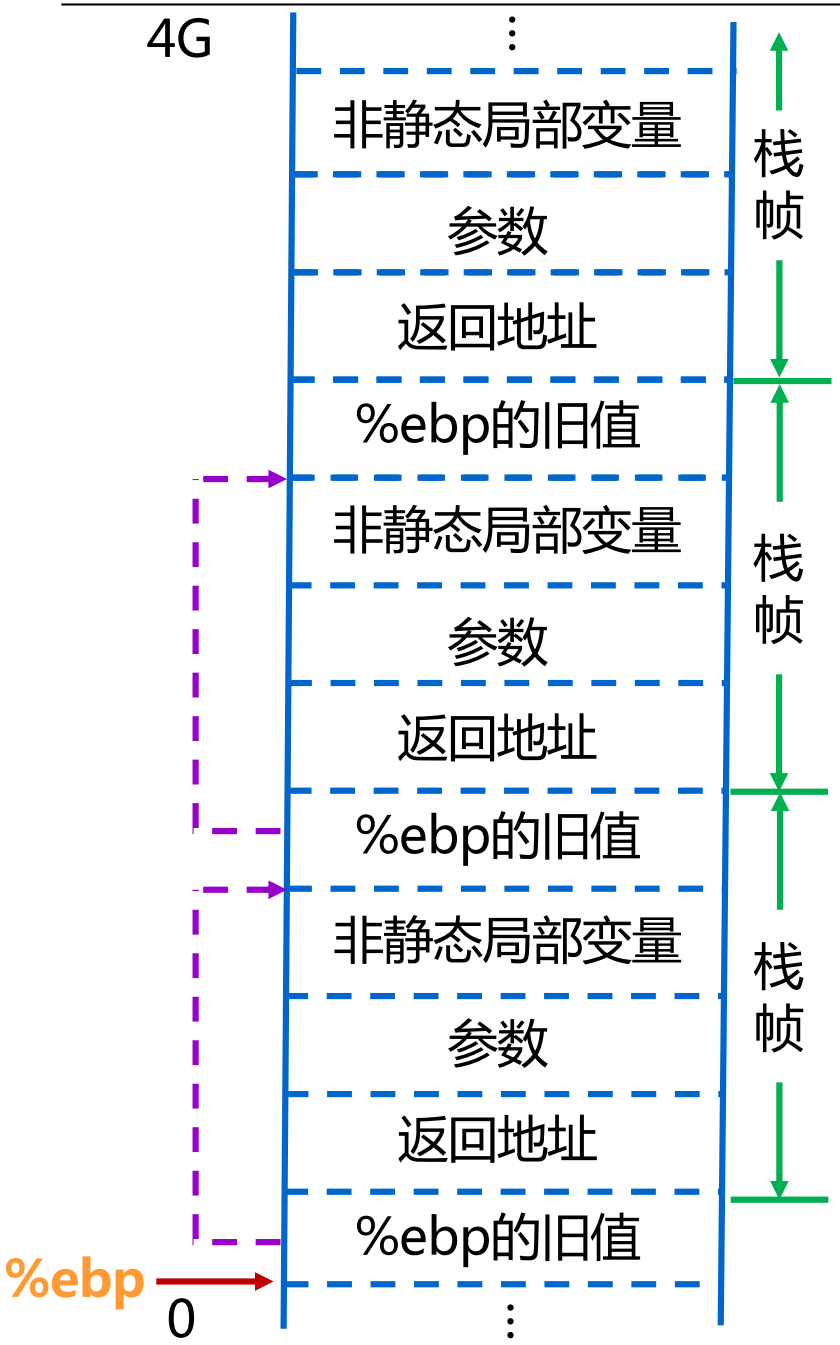
call指令的下一条指令的地址！

- IA-32中，若参数类型是unsigned char、char或unsigned short、short，也都分配4个字节
- 故在被调用函数中，使用R[ebp]+8、R[ebp]+12、R[ebp]+16作为有效地址来访问函数的入口参数
- 每个过程开始两条指令

  pushl %ebp

  movl %esp, %ebp

强化调试器（2）——打印栈帧链 (1)



- 若干次函数调用会在堆栈中形成**栈帧链**；
- 栈帧链保存函数调用相关信息，可供调试使用；
- 在简易调试器中增加“**bt**”命令，打印栈帧链；
- 通过%ebp可以找到每个栈帧的信息。

通过类似链表的结构可以组织栈帧链：

```
typedef struct {
        swaddr_t prev_ebp;  //%ebp的旧值
        swaddr_t ret_addr;   //返回地址
        uint32_t args[4];     //函数的实参
} PartOfStackFrame;
```

链表头存储在%ebp中，当**%ebp = 0**时，表示到达最开始运行的函数

打印栈帧链（2）

- 打印栈帧链时，只需要打印**返回地址**，**函数名**以及**前4个参数**即可；
- 打印格式可参考GDB中的“bt”命令；
- 如何确定某个地址落在哪一个函数中呢？

```
#0  foo (name=0x8048621 "jessie") at test.cpp:5
#1  0x080484f0 in too (name=0x8048621 "jessie") at test.cpp:10
#2  0x08048516 in bar (name=0x8048621 "jessie", myname=0x804861c "jack") at test.cpp:16
#3  0x08048535 in main () at test.cpp:21
```

```
Symbol table '.symtab' contains 10 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00100000     0 SECTION LOCAL  DEFAULT    1
     2: 0010009c     0 SECTION LOCAL  DEFAULT    2
     3: 00100100     0 SECTION LOCAL  DEFAULT    3
     4: 00000000     0 SECTION LOCAL  DEFAULT    4
     5: 00000000     0 FILE    LOCAL  DEFAULT  ABS add.c
     6: 00100084    22 FUNC    GLOBAL DEFAULT    1 add
     7: 00100000   129 FUNC    GLOBAL DEFAULT    1 main
     8: 00100120   256 OBJECT  GLOBAL DEFAULT    3 ans
     9: 00100100    32 OBJECT  GLOBAL DEFAULT    3 test_data
```

为简易调试器添加“bt”命令，实现打印栈帧链的功能。实现后，在add.c中的add()函数中设置断点，触发断点后，在monitor中测试bt命令实现是否正确。

主要内容

- NEMU中的指令执行过程
- NEMU中对浮点数的支持 — 定点化
- 强化简易调试器
- 程序的加载
- 改变程序的行为（选做）

实验目的和要求

- **实验目的**
  - 掌握可执行ELF目标文件格式
  - 实现程序的加载loader功能

- **实验要求**
  - 在NEMU模拟平台的操作系统微内核kernel中，实现程序加载功能，即loader模块。NEMU从kernel开始执行，kernel中的loader模块负责将用户程序记载到正确的内存位置，然后执行用户程序。
  - 实现loader之后，使用test.sh脚本进行测试，在工程目录下运行“make test”即可。

可执行ELF文件的组织

- ELF文件提供**两个**视角组织**可执行文件**：
  - 面向链接的section视角
  - 面向执行的segment视角
  - 一个segment可能包含0个或多个section，但一个section可能不被包含于任何一个segment（readelf可查看section和segment之间的映射关系）
- ELF采用**程序头表**（program header table）来管理segment
- 程序头表中的每一个表项描述了一个segment的所有属性：
  - 类型，虚拟地址，标志，对齐方式，文件内偏移量以及segment大小
- 加载可执行文件时，只需要加载和运行时刻相关的内容（**代码和数据**）

可执行目标文件中的程序头表

```
typedef struct {
        Elf32_Word    p_type;
        Elf32_Off     p_offset;
        Elf32_Addr    p_vaddr;
        Elf32_Addr    p_paddr;
        Elf32_Word    p_filesz;
        Elf32_Word    p_memsz;
        Elf32_Word    p_flags;
        Elf32_Word    p_align;
} Elf32_Phdr;
```

程序头表能够描述可执行文件中的节与虚拟空间中的存储段之间的映射关系

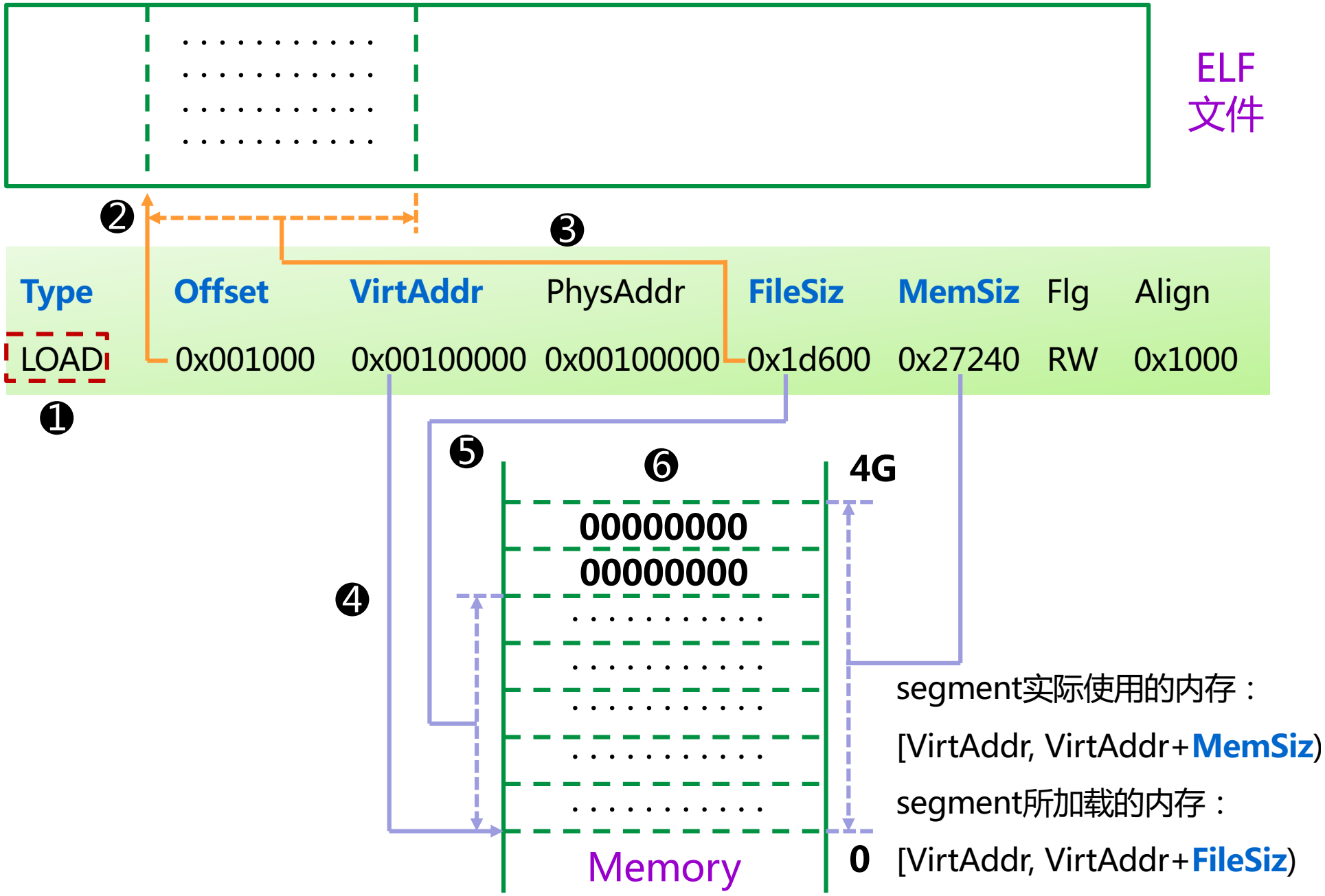
一个表项说明虚拟地址空间中一个连续的片段或一个特殊的节

以下是GNU READELF显示的某可执行目标文件的程序头表信息

$ readelf –l main

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x00100 0x00100 R E 0x4
  INTERP         0x000134 0x08048134 0x08048134 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x004d4 0x004d4 R E 0x1000
  LOAD           0x000f0c 0x08049f0c 0x08049f0c 0x00108 0x00110 RW  0x1000
  DYNAMIC        0x000f20 0x08049f20 0x08049f20 0x000d0 0x000d0 RW  0x4
  NOTE           0x000148 0x08048148 0x08048148 0x00044 0x00044 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4
  GNU_RELRO      0x000f0c 0x08049f0c 0x08049f0c 0x000f4 0x000f4 R   0x1
```

如何根据segment属性进行加载



Kernel简介

- kernel是一个单任务微型操作系统内核，在/工程路径/kernel目录下
- 通过/工程路径/kernel/include/common.h中的宏控制kernel的功能
- 在工程目录下，通过命令“make kernel”来编译kernel

Kernel的源文件组织

```
kernel
├── include
│   ├── common.h
│   ├── debug.h
│   ├── memory.h
│   ├── x86
│   │   ├── cpu.h
│   │   ├── io.h
│   │   └── memory.h
│   └── x86.h
├── Makefile.part
└── src
    ├── driver
    │   ├── ide            # IDE驱动程序
    │   │   └── ...
    │   └── ramdisk.c     # ramdisk驱动程序
    ├── elf             # loader相关
    │   └── elf.c
    ├── fs             # 文件系统
    │   └── fs.c
    ├── irq             # 中断处理相关
    │   ├── do_irq.S        # 中断处理入口代码
    │   ├── i8259.c        # intel 8259中断控制器
    │   ├── idt.c         # IDT相关
    │   └── irq_handle.c     # 中断分发和处理
    ├── lib
    │   ├── misc.c        # 杂项
    │   ├── printk.c
    │   └── serial.c        # 串口
    ├── main.c
    ├── memory            # 存储管理相关
    │   ├── kvm.c        # kernel虚拟内存
    │   ├── mm.c        # 存储管理器MM
    │   ├── mm_malloc.o        # 为用户程序分配内存的接口函数，不要删除它！
    │   └── vmem.c        # video memory
    ├── start.S            # kernel入口代码
    └── syscall            # 系统调用处理相关
        └── do_syscall.c
```

当前kernel的工作流程

#：工程路径/kernel

- 第一条指令从#/src/start.s开始，设置堆栈，跳转到#/src/main.c的**init()**函数执行；
- 由于此时NEMU还不支持分段分页，因此，直接调转到**init_cond()**函数；
- 继续跳过一些初始化工作之后，会通过宏Log()输出一句话；
  - kernel中定义的Log()，并不是NEMU中的Log()，kernel和NEMU相互独立；
  - 在kernel中，宏Log()通过printk()，但目前NEMU还不提供输出功能；
- 调用**loader()**函数加载用户程序，loader()函数会返回程序的入口地址；
- 调转到用户程序的入口地址开始执行。

实现程序加载功能——源代码

- 在~/src/elf/elf.c的loader()函数中定义正确的**ELF文件魔数**；
- **编写加载segment的代码**， 完成加载用户程序的功能；
- 实现程序加载时，使用**ramdisk_read()**函数读出ramdisk中的内容；
  - 位于文件~/src/driver/ramdisk.c中
  - int ramdisk_read(uint8_t *buf, uint32_t offset, unit32_t len)

实现程序加载功能 —— 修改Makefile

- 修改文件工程路径/testcase/Makefile.part中的**链接选项**

```
testcase_START_OBJ := $(testcase_OBJ_DIR)/start.o
-testcase_LDFLAGS := -m elf_i386 -e start -Ttext=0x00100000
+testcase_LDFLAGS := -m elf_i386 -e main -Ttext-segment=0x00800000
```

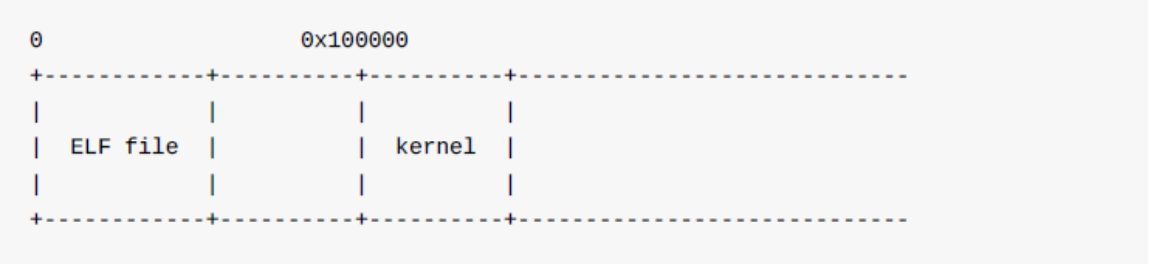
- 修改文件工程路径/Makefile，**把kernel作为entry**

```
USERPROG = obj/testcase/mov-c
-ENTRY = $(USERPROG)
+ENTRY = $(kernel_BIN)
```

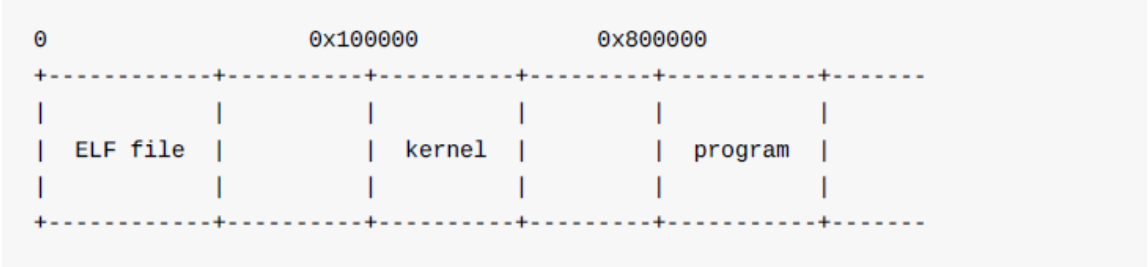
NEMU中物理内存的变化

初始化ramdisk

加载kernel（由nemu中的monitor模块加载）



kernel加载用户程序



主要内容

- NEMU中的指令执行过程
- NEMU中对浮点数的支持 — 定点化
- 强化简易调试器
- 程序的加载
- 改变程序的行为（选做）

实验目的和要求

- **实验目的**
  - 弄清代码的本质
- **实验要求**
  - 在NEMU中成功运行print-FLOAT.c测试程序

print-FLOAT.c测试程序

- print-FLOAT程序的功能是通过 sprintf() 函数的%f 功能格式化 FLOAT类型的变量, 然后使用strcmp()函数来检查；
- 如何实现让**%f**来格式化一个FLOAT类型的变量？
  - ~~修改uclibc的源代码, 然后重新编译；~~
  - **对运行时代码进行修改**；
- 通过对**libc.a**进行攻击，“劫持”相应的执行流，来改变%f的行为！

先在GNU/Linux中实施攻击，成功后再将程序移植到NEMU中运行
在工程路径下运行：

```
make pa2-7
```

“劫持”_vfprintf_internal() 函数（1）

- 运行obj/testcase/print-FLOAT-linux ，发现输出的结果为0.000000？
- 格式化%f时，_vfprintf_internal() 执行如下代码：

```
else if (ppfs->conv_num <= CONV_A) { /* floating point */
        ssize_t nf;
        nf = _fpmaxtostr(stream,
                         (__fpmax_t)
                         (PRINT_INFO_FLAG_VAL(&(ppfs->info),is_long_double)
                          ? *(long double *) *argptr
                          : (long double) (* (double *) *argptr)),
                          &ppfs->info, FP_OUT );
        if (nf < 0) {
            return -1;
        }
        *count += nf;
        return 0;
} else if (ppfs->conv_num <= CONV_S) { /* wide char or string */
```

float类型数据格式化，函数原型：

```
extern ssize_t _fpmaxtostr(FILE * fp, __fpmax_t x, struct
printf_info *info, __fp_outfunc_t fp_outfunc) attribute_hidden;
```

“劫持”_vfprintf_internal() 函数（2）

- 实现lib-common/FLOAT/FLOAT_vfprintf.c 中的**modify_vfprintf()**函数；
- 在运行时刻修改_vfprintf_internal()的二进制代码；
- 使其调用lib-common/FLOAT/FLOAT_vfprintf.c中的**format_FLOAT()**函数，而不是_fpmaxtostr()函数；

劫持后的代码如下：

```
else if (ppfs->conv_num <= CONV_A) { /* floating point */
        ssize_t nf;
        nf = format_FLOAT(stream, *(FLOAT *) *argptr);
        if (nf < 0) {
                return -1;
        }
        *count += nf;
        return 0;
} else if (ppfs->conv_num <= CONV_S) { /* wide char or string */
```

1. 将函数调用目标改为format_FLOAT()
2. 设置好正确的函数调用参数
3. 清理因为调用 _fpmaxtostr() 而留下的浮点指令

修改函数调用目标（1）

对 obj/testcase/print-FLOAT-linux 进行反汇编，找到对应的二进制代码地址范围

- 阅读反汇编代码，e8开头的是call REL32形式的指令，后面跟的是一个相对于当前eip的偏移量；
- 只需修改这一偏移量，就可以达到修改函数调用目标的目的；
  - 首先需要知道这条call指令的地址。假设该指令的地址为p，那么 p+1就是需要修改的偏移量； **如何修改这一偏移量呢?**
  - 只需要让该偏移量加上_fpmaxtostr() 和format_FLOAT()首地址之间的差即可；
  - 在modify_vfprintf()中编写代码，修改函数调用的目标。

新问题：

代码重新编译后，由于代码发生了变化，可能会导致修改的那条call指令在链接重定位阶段后的地址不再是之前的“p”，**如何解决？**

虽然call指令位置会变化，但相对于_vfprintf_internal()函数首地址的偏移量是不变的！

修改函数调用目标（2）

- 运行修改后的代码，程序**触发段错误**！
- 在GNU/Linux下，libc代码是只读的，而我们需要在运行时刻修改程序的代码，进行了违规的写操作，从而触发了段错误；
- 使用系统调用 **mprotect()**，改变相应内存区间的访问权限。

```
#include <sys/mman.h>
mprotect((void *)(((unsigned)_vfprintf_internal + o - 100) & 0xfffff000),
4096*2, PROT_READ | PROT_WRITE | PROT_EXEC);
```

注： “**o**”表示call指令相对于_vfprintf_internal()首地址的偏移量

“**100**”表示从修改的call指令往前100字节处开始添加可写权限

重新编译运行，%f输出的结果是什么？

设置正确的函数调用参数（1）

- 输出了一个十六进制数，但并不是调用printf()时传入的 FLOAT数据；

- 目前，仅仅改变了函数调用目标，_vfprintf_internal()仍然按照调用 _fpmaxtostr()那样压入参数，format_FLOAT() 却以为压入了正确的参数；

- 继续修改modify_vfprintf()代码，传入正确参数。

阅读相应的汇编代码, 并回答以下问题：

1. 调用 _fpmaxtostr()的参数是如何压栈的?
2. 变量 argptr 存放在哪一个寄存器中?
3. 变量 argptr 指向的内容是什么?
4. 变量 stream 的地址在哪里?

设置正确的函数调用参数（2）

- 第一个参数stream是一样的，因此传入stream的代码不需要进行改动；
- 原来的代码中通过一条占用三个字节浮点指令来放置**第二个参数**，我们需要将它换成一条**push指令**，用于将**argptr**指向的内容压栈即可。
  - 使用的push指令**不能超过3个字节**；
  - 如果push指令不足3个字节，剩下的字节要**通过nop来覆盖**，保证原来那条3字节浮点指令“不留任何痕迹”；
  - 由于使用的**push指令改变了栈的深度**，为了正确维护栈的深度，还需要**修改前一条用于分配栈空间大小的指令**；
  - 由于format_FLOAT()只会使用两个参数，传入的其余参数可以暂时忽略。

现在的format_FLOAT()可以正确地获得FLOAT类型的实参。

修改format_FLOAT()的实现，把FLOAT类型数据的十进制小数表示作为格式化的结果。约定格式化结果通过截断的方式保留6位小数。

清除其他浮点指令

- 清除这段代码中的其它浮点指令，让print-FLOAT可以在NEMU中运行；

- 只需要用**nop**指令覆盖它们即可。

注：在NEMU中运行之前，去掉代码中的**mprotect()**系统调用，因为目前NEMU还不能支持系统调用的执行。

新问题：再次遇到浮点指令

- 在NEMU中运行print-FLOAT，仍然遇到浮点指令；
- 遇到的浮点指令位于_ppfs_setargs()中；
- 编写lib-common/FLOAT/FLOAT_vfprintf.c中的modify_ppfs_setargs()函数，对 ppfs_setargs()的运行时二进制进行修改，绕过上述浮点指令。
  - 相关代码是1个switch-case语句, 通过不同格式说明符来获取不同长度的数据；
  - float分支的开头放置一条 jmp指令，跳转到long long分支。

- PA2提交时间：11月6日，晚12:00之前
- 提交内容：
  - 在线报告（写题目）
  - 打包工程（make submit）

PA2到此结束

Q&A ?