

华中科技大学

2017

系统能力综合训练 课程设计报告

题 目:	X86 模拟器设计
专 业:	计算机科学与技术
班 级:	CS1406
学 号:	U20151465
姓 名:	梅纽因
电 话:	15158575337
邮 件:	jeff.mei@outlook.com
完成日期:	2017-10-31 周二上午



计算机科学与技术学院

华中科技大学课程设计报告

目 录

1	课程设计概述.....	2
1.1	课设背景	2
1.2	设计任务	2
2	简易调试器.....	3
2.1	功能实现要点.....	3
2.2	必答题	10
2.3	主要故障与调试.....	11

1 课程设计概述

1.1 课设背景

理解"程序如何在计算机上运行"的根本途径是从"零"开始实现一个完整的计算机系统。华中科技大学计算机科学与技术系计算机系统基础课程的小型项目 (Programming Assignment, PA) 将提出 x86 架构的一个教学版子集 n86, 指导学生实现一个功能完备的 n86 模拟器 NEMU(NJU EMUlator), 最终在 NEMU 上运行游戏"仙剑奇侠传", 来让学生探究"程序在计算机上运行"的基本原理。

1.2 设计任务

NEMU 是一个基于 QEMU(Quick Emulator)的项目, 并去除了大量与课程内容差异较大的部分。PA 包括一个准备实验(配置实验环境)以及 5 部分连贯的实验内容, 包括:

- 简易调试器
- 冯诺依曼计算机系统
- 批处理系统
- 分时多任务
- 程序性能优化

实验环境为:

- CPU 架构: x64
- 操作系统: GNU/Linux
- 编译器: GCC
- 编程语言: C 语言

2 简易调试器

2.1 功能实现要点

PA1 的主要任务是实现一个简易调试器，为后续实验提供一个 debug 工具。调试器的主要功能包括：

- 单步执行 N 条指令 (si)
- 表达式求值 (p)
- 创建、删除监视点 (w/d)
- 打印寄存器和监视点状态 (info)

上述的命令均在 `nemu/src/monitor.debug/ui.c` 中的 `cmd_table` 结构体中有定义。对应每一种命令在该文件下均有一个专门的入口函数来执行相应的操作。

2.1.1 实现寄存器结构体

根据 x86 手册, CPU 内共有 8 个通用寄存器。我们即可以通过其位次访问 `gpr[i]`, 亦可以使用其别名访问。因此我们需要正确修改相关结构, 保证两种访问方式等价。

由于计算机中寄存器公用一片内存, 因此我们使用 `union` 实现内存公用, 定义如下结构体和枚举。(在 `nemu/include/cpu/reg.h` 中定义)

```
const char *regsl[] = {"eax", "ecx", "edx", "ebx", "esp", "ebp",  
"esi", "edi"};  
  
union {  
    union{  
        uint32_t _32;  
        uint16_t _16;  
        uint8_t _8[2];  
    } gpr[8];  
    struct{  
        rtlreg_t eax, ecx, edx, ebx, esp, ebp, esi, edi;  
    };  
};
```

华中科技大学课程设计报告

实现 gpr 结构体后，运行 NEMU。当通过 `nemu/src/cpu/reg.c` 中的 `reg_test` 的断言测试时，说明该结构体的实现是正确的。

2.1.2 基础设施

1. 单步执行 N 条指令

当调试器读取到“si”字符串时，则会调用 `cmd_si` 函数进行相应的处理。若参数缺省，则默认为单步执行；若有参数则将参数转换为 `int` 类型后再调用 `cpu_exec()` 函数，参数则为执行的步数即可。测试结果如图 2.1 所示。

```
static int cmd_si(char *args) {
    char *arg = strtok(NULL, ""); //extract the first argument
    if (arg == NULL) {
        cpu_exec(1); //no argument given
    }
    else {
        int count = atoi(arg);
        cpu_exec(count);
    }
    return 0;
}
```



The screenshot shows the NEMU terminal interface. It starts with a welcome message and a prompt for help. The user enters 'si', which triggers a single-step execution. The terminal displays the current instruction address and its assembly code, along with the state of the CPU registers. The first instruction is at address 100000, which is a 'movl \$0x1234,%eax' instruction. The second instruction is at address 100005, which is a 'movl \$0x100027,%ecx' instruction. The third instruction is at address 10000a, which is a 'movl %eax,(%ecx)' instruction. The terminal prompt '(nemu)' is visible at the bottom.

图 2.1 单步调试测试结果

2. 打印寄存器和监视点状态

当调试器读取到“info”字符串时，则会调用 `cmd_info` 函数进行相应的处理。若参数为“r”，则根据寄存器的别名顺序依次打印出每一个寄存器内当前的状态；若参数

华中科技大学课程设计报告

为“w”，则会调用 `print_wp` 打印监视点当前的状态。这部分内容在下一小节之后会实现。测试效果如图 2.2 所示。

```
static int cmd_info(char *args) {
    char *arg = strtok(NULL, ""); //extract the first argument
    if (arg == NULL) {
        printf("Wrong argument!\n");
        return 0;
    }
    else if (!strcmp(arg, "r")) {
        printf("Register Status:\n");
        for(int i = R_EAX; i <= R_EDI; i++){
            printf("%s\t\t0x%x\n", reg_name(i, 4), reg_l(i));
        }
        printf("eip\t\t0x%x\n", cpu.eip);
        return 0;
    }
    else if (!strcmp(arg, "w")) {
        print_wp();
        return 0;
    }
    else {
        printf("Wrong argument!\n");
        return 0;
    }
}
```

```
Welcome to NEMU!
For help, type "help"
(nemu) info r
Register Status:
eax          0x51eea056
ecx          0x4ece53a3
edx          0x60f77b88
ebx          0x34b0b2f2
esp          0xbff01ec
ebp          0x7c2c8e2e
esi          0x6d6d2a1
edi          0x125c8f60
eip          0x100000
(nemu) █
```

图 2.2 打印寄存器状态测试

3. 扫描内存

当调试器读取到“x”字符时，则会调用 `cmd_x` 函数做相应的处理。参数内容则为一个整形 `N` 和一个表达式 `EXPR`。函数功能为：首先求出表达式 `EXPR` 的值，将结果作为起始内存地址，随后以十六进制形式输出连续的 `N` 个 4 字节。

进入 `cmd_x` 函数后，首先提取第一个和第二个参数并转换成可处理的整形和表达式结果。若转换失败，则输出错误信息；若转换成功，以表达式 `EXPR` 的值作为内存地址 `addr`，随后以十六进制形式输出连续的 `N` 个 4 字节。

```
static int cmd_x(char *args){
    char *arg1 = strtok(NULL, " ");
    char *arg2 = arg1 + strlen(arg1) + 1;
    int num = atoi(arg1);
    bool success;
    vaddr_t addr = expr(arg2, &success);
    if(!success){
        printf("Invalid input!\n");
        return 0;
    }
    for(int i = 0; i < num; i++){
        vaddr_t addr_temp = addr + 4 * i;
        uint32_t data = vaddr_read(addr_temp, 4);
        if(i % 4 == 0){
```

```
printf("0x%x:\t", addr_temp);
}

printf("0x%x\t", data);

if((i + 1) % 4 == 0){

    printf("\n");

}

}

printf("\n");

return 0;

}
```

2.1.3 表达式求值

这一部分是实现简易调试器中最为重要的一环。在 `nemu/src/debug/expr.c` 中的 `rule` 结构中，我们给出了以下 `token`，我们需要将他们的正则表达式按如下识别（而非运算）的优先级存放在该结构中。

表 2-1 token 及识别顺序表

优先级	运算符	正则表达形式
1	空格	" "
2	+ - * /	"\\+" "\\-" "*" "/"
3	()	"\\(" "\\)"
4	十六进制数字	"0[xX][0-9a-fA-F]+"
5	十进制数字	"[0-9][0-9]*"
6	寄存器	"\\\$[a-zA-Z][a-zA-Z]+"
7	== !=	"==" "!="
8	&&	"&&" "\\ \\ \\ "
9	!	"!"

当每一个 `token` 均被成功识别之后，他们会被存放在 `tokens` 数组，进入 `expr` 函数继续进行处理。大致过程可以分为 3 步：

华中科技大学课程设计报告

1.对减与负(-), 乘与指针(*)进行区分。若‘-’或‘*’出现在表达式的开头, 或他们之前的一个 token 并非数值类型, 此时他们为负或指针; 否则他们为减或乘。

2.在 eval 函数中进行递归求值。

3.若在之前有任何的不匹配发生, 输出错误信息; 否则返回运算结果。

递归求值的核心思想为, 先在表达式中找到主运算符, 根据主运算符的位置将表达式拆分为左右两个子表达式; 随后再将左右两个子表达式继续做如下拆分, 直至表达式中仅剩下一个 token 时进行求值, 随后一步步向上回溯得到整个表达式的值。在实验指导中已经给出了框架逻辑, 这里不再赘述。

找到主运算符的 dominant_operator 函数的逻辑为, 我们首先遍历整个输入的表达式。若该 token 不是运算符或者在一个括号内时, 指针向下一个 token 移动; 若找到了符合要求的 token 时, 暂且存放当前的位置 i, 令取一个新的指针向后遍历。该函数的伪代码如下。

```
dominant_operator(p, q) {  
    i = p;  
    //go on traverse  
    while(i <= q && (!is_operator(tokens[i].type) || inside_par(i, p, q))){}  
    //invalid  
    if(i > q) {}  
    dominant_pos = i;  
    for(++i; i <= q; ++i){  
        if(!is_operator(tokens[i].type) || inside_par(i, p, q))  
            continue;  
        cur_type = tokens[i].type;  
        dominant_type = tokens[dominant_pos].type;  
        //if the pointer hit a more dominant operator  
        if(check_priority(cur_type, dominant_type) > 0)    dominant_pos = i;  
    }  
    return dominant_pos;  
}
```

华中科技大学课程设计报告

若新的指针指向的 token 满足以下条件时,则会取代当前存放的主运算符的位置:

1. 是一个运算符且不在括号内
2. 运算优先级比当前的运算符高

找到主运算符后,我们便可以递归地在 `eval` 函数中求值得到结果。

在上述过程中会使用到以下辅助函数:

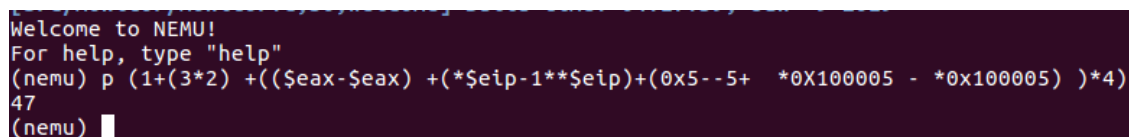
`check_parentheses`: 检查表达式的左右括号是否匹配,并返回相应的 bool 值。

`associate`: 检查该运算符的结合性,返回 0 代表左结合,返回 1 代表右结合。

`check_priority`: 根据输入的运算符返回其运算的优先级。数值越小代表运算优先级越高。

`read_reg`: 若当前的 token 类型为寄存器类型,则需要根据字符串的内容,与 `regsl`, `regsw`, `regsb` 三个数组内的寄存器别名字符串进行比较后找到相应的寄存器并读取结果。

表达式求值测试结果如图 2.3 所示。



```
Welcome to NEMU!
For help, type "help"
(nemu) p (1+(3*2) + (($eax-$eax) + (*$eip-1**$eip)+(0x5--5+ *0X100005 - *0x100005) ) *4)
47
(nemu) 
```

图 2.3 表达式求值测试

2.1.4 监视点

在框架代码中,监视点是作为两个链表(分别代表空闲监视点和被占用监视点)进行存放的。

首先对 `nemu/include/monitor/watchpoint.h` 中的 `WP` 监视点数组进行修改。两个成员,分别为表达式结果 `value` 和表达式字符串本身 `expr`。

```
typedef struct watchpoint {
    int NO;
    struct watchpoint *next;
    /* TODO: Add more members if necessary */
    uint32_t value;                // original value of the expr
}
```

华中科技大学课程设计报告

```
char expr[MAX_LENGTH_OF_EXPR + 1];    // the expression
} WP;
```

随后在 `nemu/src/monitor/debug/watchpoint.c` 实现以下辅助函数：

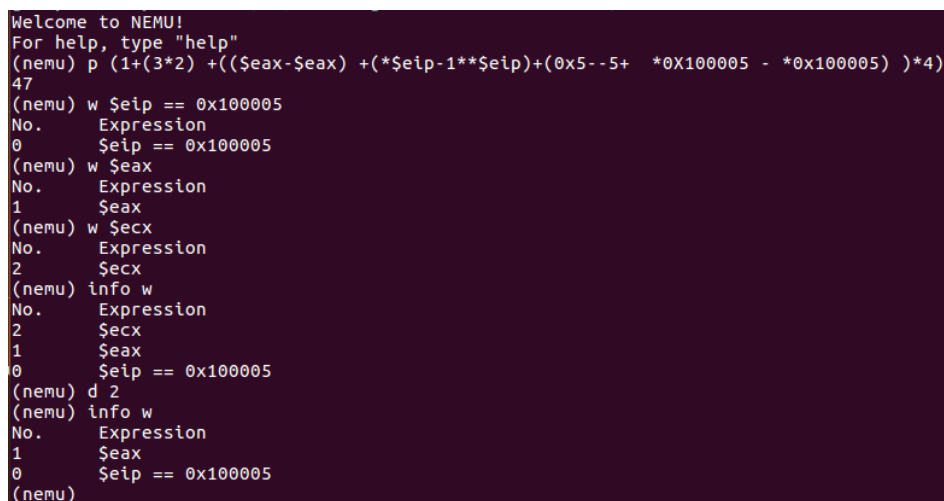
new_wp: 从空闲监视点链表中取出一个节点分配给新的监视点，返回新分配监视点的指针。

free_wp: 当一个监视点使用完毕时，将其从被占用监视点链表中转移到空闲监视点链表中。

check_wp: 被 `nemu/src/cpu/exec.c` 所调用。遍历被占用监视点链表，并重新计算表达式内容本身和最初的值 `value` 进行比较。如出现变化，则将表达式旧值和新值打印供调试用。

print_wp: 遍历被占用监视点链表，打印每一个监视点的状态。

监视点测试结果如图 2.4 所示。



```
Welcome to NEMU!
For help, type "help"
(nemu) p (1+(3*2) +(($eax-$eax) +(*$eip-1*$eip)+(0x5--5+  *0x100005 - *0x100005) )*4)
47
(nemu) w $eip == 0x100005
No.      Expression
0        $eip == 0x100005
(nemu) w $eax
No.      Expression
1        $eax
(nemu) w $ecx
No.      Expression
2        $ecx
(nemu) info w
No.      Expression
2        $ecx
1        $eax
0        $eip == 0x100005
(nemu) d 2
(nemu) info w
No.      Expression
1        $eax
0        $eip == 0x100005
(nemu)
```

图 2.4 监视点测试

2.2 必答题

1.理解基础设施

如果没有简易调试器，则花费的调试时间为 $500 \times 0.9 \times 30 \times 20 = 270,000s = 75h$ 。使用简易调试器可以节约 $1/3$ 的时间。

2.查阅 i386 手册

EFLAGS 中的 CF 位

位于 Appendix C – Status Flag Summary

华中科技大学课程设计报告

ModR/M

位于 17.2 Instruction Format

mov

位于 17 80836 Instruction Set

3.shell 命令

使用 `find . -name "*.c|h" |xargs cat|wc -l` 命令统计代码行数。PA1 部分共有 4336 行代码，PA0 共有 3824 行代码，共有 512 行的差别/

使用 `find . -name "*.c|h" |xargs cat|grep -v ^$|wc -l` 命令统计非空行代码行数。PA1 分支共有 3594 行。

4.使用 man

-wall 表示输出 warning 信息。

-werror 表示将所有的 warning 变为 error 使程序停止编译。

这种方法可以更好的帮我们调试代码，查看错误信息。

2.3 主要故障与调试

1.表达式 token 识别顺序错误

最开始在 `rule` 结构中加入每一个 token 时，我没有注意到每一个 token 存放的顺序其实和识别过程的优先级有关。因此我没有在意，将识别十进制的正则表达式放置于识别十六进制的 token 之前。这样当出现十六进制数时，对于其中的“0x”部分，表达式引擎会先识别 0 作为一个十进制数，随后将“x”作为一个非法 token。我在 `make_token` 函数中将每一个 token 识别的过程单步打印出来后才发现问题，注意到了框架代码注释中对于 precedence 的提示。

2.表达式长度问题

在进行最终的表达式测试时，对于教师给出的测试案例

$(1+(3*2)+((\$eax-\$eax)+(*\$eip-1**\$eip)+(0x5--5+ *0X100005 - *0x100005))*4)$

如果将表达式拆开进行求解结果合法，但是输入完整的表达式时无法输出正确的答案。经过单步调试后我发现在 `nemu/src/monitor/debug/expr.c` 中有如下定义：

```
Token tokens[64];
```

```
int nr_token;
```

这个定义说明了 tokens 数组最多只能存放 32 个 token，而该表达式由于过长，之后的 token 无法被正确识别，才导致了没有任何结果输出。将数组长度从 32 改为 64 即可修复该问题。

3 冯诺依曼计算机系统

3.1 功能实现要点

在 PA2 中，我们要实现 i386 手册中的绝大部分指令，能够运行一个最为基本的冯-诺伊曼计算机系统。这也是整个系统能力培养中工作量最大的一个部分。

在 `nemu/src/decode/` 目录下的文件均与指令译码有关。其中 `decode.c` 实现了 `include/cpu/decode.h` 中的译码函数族，函数 `operand_write` 以及译码信息变量 `decoding`，`include/cpu/rtl.h` 中的临时寄存器 `t0,t1,t2,t3,at` 和函数 `decoding_set_jump`。

在 `nemu/src/exec/` 目录下的文件均与指令执行有关。其中，`all-instr.h` 定义了已经实现的指令执行函数；`exec.c` 位指令执行过程核心实现。大部分工作填充数组 `opcode_table`，也就是译码表。译码表分两段：单字节指令码和双字节指令码。

在 `nemu/include/cpu/rtl.h` 中，定义和实现了一些 RTL 指令，用于提供对指令执行的底层建模。可使用这些操作将复杂指令分解成更简单的操作。

3.1.1 dummy 程序实现

观察 dummy 的反汇编代码文件 `dummy-x86-nemu.txt`，并与框架代码中已经实现的指令作比较，我们需要实现 `call`, `sub`, `push`, `pop`, `xor`, `ret` 六条指令即可，并在 `exec.c` 中添加相应调用，以及 `all-instr.h` 中添加声明即可。

1.rtl 基本指令与伪指令

对于 rtl 基本指令而言，不需要使用临时寄存器，可以看做是最基本的 x86 指令中的最基本的操作。实现时添加了 `interpret_` 前缀，但在 `include/cpu/rtl-wrapper.h` 作用下，其它代码中使用到这些 rtl 基本指令时会自动添加 `interpret_` 前缀。

2.call 指令

该指令将下一个 `eip` 压入栈中，随后跳转至指定地址继续工作。结合之前实现的 `rtl_push` 和 `rtl_j` 指令，我们可以轻松地实现 `call` 指令。

```
make_EHelper(call) {  
    rtl_push(&decoding.seq_eip);  
    rtl_j(decoding.jump_eip);  
    print_asm("call %x", decoding.jump_eip);  
}
```

```
}
```

3.sub 指令

该指令为减法指令，需要实现减法并写回。在实现该指令前，我们需要回到寄存器结构，添加 EFLAGS 寄存器，新添加后的结构如下：

```
union {  
    struct {  
        uint8_t CF : 1;  
        uint8_t DEF1: 1;  
        uint8_t DEF2: 4;  
        uint8_t ZF : 1;  
        uint8_t SF : 1;  
        uint8_t DEF3: 1;  
        uint8_t IF : 1;  
        uint8_t DEF4: 1;  
        uint8_t OF : 1;  
        uint32_t DEF5: 20;  
    } eflags;  
    uint32_t flags;
```

其中 CF, OF 根据操作结果判断并更新。CF 将数字当作无符号数，OF 将数字当作有符号数。

对于 sub 指令，只有当源操作数与目的操作数符号不同时才会溢出，而 CF 标志位则判断最高位是否产生借位或进位。实现方式如下。

```
make_EHelper(sub) {  
    rtl_sext(&t1, &id_dest->val, id_dest->width);  
    rtl_sext(&t2, &id_src->val, id_src->width);  
  
    rtl_sub(&t0, &t1, &t2);  
    t3 = (t0 > t1);  
    rtl_set_CF(&t3);
```

华中科技大学课程设计报告

```
t3 = (((int32_t)(t1) < 0) == (((int32_t)(t2) >> 31) == 0))
&& (((int32_t)(t0) < 0) != (((int32_t)(t1) < 0)));

rtl_set_OF(&t3);

rtl_update_ZFSF(&t0, 4);

operand_write(id_dest, &t0);

print_asm_template2(sub);
}
```

4.push 指令

该指令为压栈，与 call 类似，通过 rtl_push 实现。rtl_push 将 val 写入时总是 4 字节，所以对不足 4 字节的数，我们要对其进行扩展再压栈。

```
make_EHelper(push) {
    if(id_dest->width == 1){
        uint8_t utmp = id_dest->val;
        int8_t temp = utmp;
        id_dest->val = temp;
    }

    rtl_push(&id_dest->val);
    print_asm_template1(push);
}
```

5.pop 指令

该指令为 push 指令的逆操作。pop 操作通过 rtl_pop 实现，出栈之后需要将 pop 出来的结果写回。

```
make_EHelper(pop) {
    rtl_pop(&t0);

    if(id_dest->width == 1){
        uint8_t utemp = t0;
        int8_t temp = utemp;
        id_dest->val = temp;
    }
}
```

```
    }  
  
    else  
  
        id_dest->val = t0;  
  
        operand_write(id_dest, &id_dest->val);  
  
        print_asm_template1(pop);  
}
```

6.xor 指令

该指令为异或指令，实现很简单，但是要注意需要将 CF，OF 置为 0。

```
make_EHelper(xor) {  
  
    rtl_xor(&t1, &id_dest->val, &id_src->val);  
  
    t0 = 0;  
  
    rtl_set_OF(&t0);  
    rtl_set_CF(&t0);  
  
    rtl_update_ZFSF(&t1, id_dest->width);  
  
    operand_write(id_dest, &t1);  
  
    print_asm_template2(xor);  
}
```

7.ret 指令

该指令为调用函数返回，为 call 指令的逆操作，通过 rtl_pop 和 rtl_j 实现。

```
make_EHelper(ret) {  
  
    rtl_pop(&decoding.jump_eip);  
  
    rtl_j(decoding.jump_eip);  
  
    print_asm("ret");  
}
```

8.在 exec.c 中添加调用

我们在这里的主要工作是参阅 i386 手册中的 Appendix A -- Opcode Map，完成 `opcode_table` 的填写。在此之前，我们需要理解 exec.c 中一些宏定义的含义，如表 3-1 所示。

华中科技大学课程设计报告

表 3-1 exec.c 中相关宏定义

宏	描述
IDEXW(id, ex, w)	根据译码函数名, 执行函数名, 宽度生成 opcode_entry
IDEX(id, ex)	根据译码函数名, 执行函数名, 以宽度 0 生成 opcode_entry
EXW(ex, w)	根据执行函数名, 宽度, 生成无译码函数的 opcode_entry
EX(ex)	根据执行函数名, 生成宽度 0, 无译码函数的 opcode_entry
EMPTY	未实现的命令, 使用 exec_inv (定义在 special.c 中) 构造 opcode_entry
make_group(name, item0, item1, item2, item3, item4, item5, item6, item7)	用于实现 sub /5 这种根据第二个指令码 /5 区分不同指令的情况。 会自动生成一个 exec_name 的统一执行函数, 并分配到指定执行函数。

随后将已经实现的指令根据其操作数填入 opcode_table, 并在 all-instr.h 中添加声明后, 即可编译运行, 加载测试程序 dummy。当输出 “HIT GOOD TRAP” 时, 说明指令实现正确, 如图 3.1 所示。

```
[src/monitor/monitor.c,35,Welcome] Debug: 0x1
[src/monitor/monitor.c,36,Welcome] Build time: 00:59:43, Jan 10 2019
Welcome to NEMU!
For help, type "help"
nemu: HIT GOOD TRAP at eip = 0x0010001b
```

图 4.1 dummy 运行测试

3.1.2 实现更多指令

在通过 dummy 测试后, 发现其他指令的实现都是类似的。遇到未实现的指令, 根据指令查阅 i386 手册即可, 或通过反汇编代码直接确定未实现指令码所代表的指令。随后在 opcode_table 添加相应的译码函数与执行函数, 在 all-instr.h 中声明新实现的执行函数。有的函数可能没有给出函数体, 我们可以在对应的文件中进行添加即可。其中有几类指令, 需要特别强调:

1. 0f 类指令

依照 i386 手册约定, 为解决 8 位最多包含 256 中指令的局限性, 应用了 0f 类型的指令表。当发现首字符位 0f 时, 则需要再读入一个字节, 并通过 2 号指令表, 来确定实际的指令。

对于本实验, 查看 exec.c 中 opcode_table 可知, 下标 0f 处值位 EX(2byte_esc),

华中科技大学课程设计报告

即执行函数 `exec_2byte_esc`。该函数定义可在 `exec.c` 中 2 号译码表后找到。该函数通过 `instr_fetch` 读入指令中的下个字节,将该字节与 `0x100` 取或,从而索引 2 号译码表,再进行执行。

2. group 类指令:

`group` 类指令特点在于具有相同的译码方式,但执行函数却不同。这种指令的译码使用 `ModR/M` 字节,使用指令码中的 3 位进行 8 种执行指令的索引。`make_group` 宏定义了一个 `group` 数组,包含 8 个函数指针以及 1 个 `group` 相关执行函数,该函数专门负责该 `group` 函数的索引。

对于索引 `ext_opcode`,经阅读源代码,可以注意到大部分译码函数均会调用函数 `decode_op_rm`,该函数则会调用 `read_ModR_M`,用于 `ModR/M` 字节的译码。

3.1.3 实现库函数

我们需要实现在 `nexus-am/libs/klib/src/string.c` 和 `nexus-am/libs/klib/src/stdio.c` 中列出了将来可能会用到的库函数。

其中,`string.c` 包括与字符串、内存拷贝相关的基本函数,`stdio.c` 为一些基本标准输入输出函数。根据 KISS 法则,在调试过程中,为了区分指令实现的错误和库函数实现的错误,我们可以先在机器上运行我们自行实现的库函数,随后移植到 `nemu` 上进行运行。

完成以上工作后,我们便可以进行一键回归测试。通过 `bash runall.sh` 命令进行,如图 3.2 所示。通过一键回归测试后,我们可以继续下一部分的工作。

```
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!  
jeffnet@ubuntu:~/lcs2018/nemu$
```

图 4.2 一键回归测试

3.1.4 输入输出

1.in, out 指令

这两条指令的实现均与读写设备相关。其中, `in` 指令用于将设备寄存器中的数据

华中科技大学课程设计报告

传输到 CPU 寄存器中, out 指令用于将 CPU 寄存器中的数据传送到设备寄存器中。由于设备采用统一译码的方式, 因此只需要对指定区间的内存地址进行访问。这部分实现使用接口 `pio_read_[l|w|b]()` 和 `pio_write_[l|w|b]()` 实现, 其中的 `[l|w|b]` 分别代表传送数据的不同位宽。

为支持外设方式的内存访问，需要增加 `mmio` 类型的内存访问，即修改原来的内存访问函数，使其增加与内存访问相关的处理。首先通过函数 `is_mmio` 获取地址对应的设备号，若返回值为-1，则说明该内存地址为通常地址，按原来的内存访问处理；否则说明为设备地址，需要通过 `mmio_read` 携带设备号作为参数进行访问，并返回其值。

前往 `nexus-am/app/hello` 下，输入命令 `make ARCH=x86-nemu run` 命令运行 hello 程序，如图 3.3 所示，实现正确。

```
Welcome to NEMU!  
For help, type "help"  
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 = 0x00100037
```

图 3.3 hello 程序运行

2.时钟

我们针对 `nexus-am/am/amdev.h` 中的抽象寄存器 `_DEVREG_TIMER_UPTIME` 进行处理。在 `nexus-am/am/arch/x86-nemu/src/devices/timer.c` 的 `timer_read` 函数中，我们使用 `inl` 函数读取时钟端口，并将读取值的高位和地位分别传送至 `_UptimeReg` 结构体的高位和低位成员，实现获得 AM 启动时间的功能。测试如图 3.4 所示。

```
call_rm 0x100415
call_rm 0x10043c
2018-0-0 00:00:00 GMT (9218 seconds).
call_rm 0x100415
call_rm 0x10043c
2018-0-0 00:00:00 GMT (9219 seconds).
call_rm 0x100415
call_rm 0x10043c
2018-0-0 00:00:00 GMT (9220 seconds).
call_rm 0x100415
call_rm 0x10043c
2018-0-0 00:00:00 GMT (9221 seconds).
call_rm 0x100415
call_rm 0x10043c
2018-0-0 00:00:00 GMT (9222 seconds).
call_rm 0x100415
call_rm 0x10043c
```

图 3.4 运行 timetest 测试

3. 键盘

华中科技大学课程设计报告

同时钟一样，在 `nexus-am/am/arch/x86-nemu/src/devices/input.c` 通过 `inl` 函数读取键盘接口。当我们获得键盘是否按下的标志 `press` (断码) 后，若 `press` 并非 `KEY_NONE` 信号，`key_down` 只需取反即可。

4.vga

同样根据指导手册填充框架代码即可。

完成以上工作后，我们已经实现了一个冯诺依曼计算机系统。我们可以运行一些有趣的程序来体现我们的成果。

如图 3.5 所示，加载幻灯片。（程序位于 `nexus-am/apps/slider` 目录下）

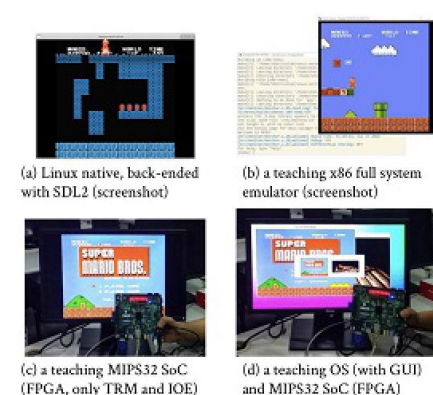


Figure 4. The same LiteNES emulator running on different platforms.

图 3.5 加载幻灯片

如图 3.6 所示，加载打字小游戏。（程序位于 `nexus-am/apps/typing` 目录下）



图 3.6 运行打字小游戏

如图 3.7 所示，运行超级马里奥游戏。（程序位于 `nexus-am/apps/litenes` 目录下）

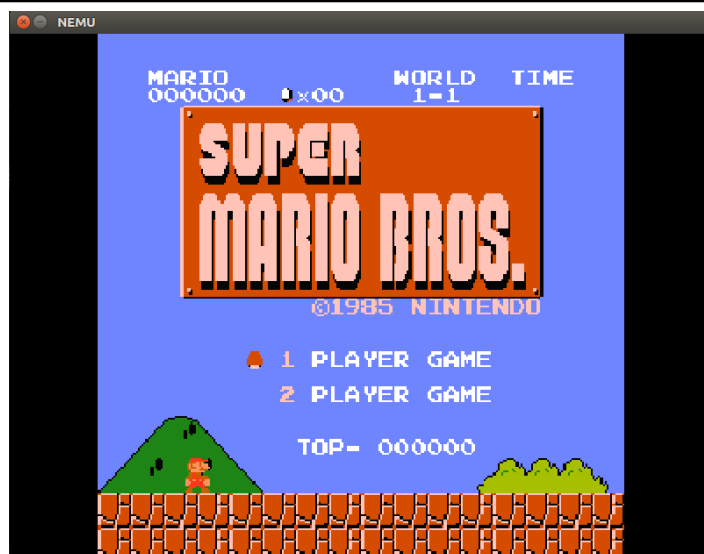


图 3.7 运行超级马里奥游戏

3.2 必答题

1.static 与 inline

去除 inline 会出现如图 3.8 所示的错误。

```
In file included from ./include/cpu/decode.h:6:0,
                 from ./include/cpu/exec.h:10,
                 from src/cpu/intr.c:1:
./include/cpu/rtl.h:170:13: error: 'rtl_push' defined but not used [-Werror=unused-function]
static void rtl_push(const rtlreg_t* src1) {
^
cc1: all warnings being treated as errors
```

图 3.8 去除 inline 后出现的错误

去除 static 不会出现报错。

去除 static 和 inline 两者后会出现如图 3.9 所示的错误。

```
defined here
build/obj/cpu/decode/decode.o: In function 'rtl_push':
/home/jeffmei/ics2018/nemu/./include/cpu/rtl.h:174: multiple definition of 'rtl_push'
build/obj/cpu/intr.o:/home/jeffmei/ics2018/nemu/./include/cpu/rtl.h:174: first defined here
build/obj/cpu/decode/modrm.o: In function 'rtl_push':
/home/jeffmei/ics2018/nemu/./include/cpu/rtl.h:174: multiple definition of 'rtl_push'
build/obj/cpu/intr.o:/home/jeffmei/ics2018/nemu/./include/cpu/rtl.h:174: first defined here
collect2: error: ld returned 1 exit status
Makefile:51: recipe for target 'build/nemu' failed
make[2]: *** [build/nemu] Error 1
/home/jeffmei/ics2018/nexus-am/Makefile.app:31: recipe for target 'run' failed
make[1]: *** [run] Error 2
Makefile:12: recipe for target 'Makefile.dummy' failed
make: [Makefile.dummy] Error 2 (ignored)
```

图 3.9 去除 static 后出现的错误

当函数被声明 static 后，它只在定义它的源文件内有效，其他源文件无法访问，所以用来解决不同文件函数重名问题，如果去掉进行编译的话，若不同文件有相同

华中科技大学课程设计报告

函数名则会报错。

`inline` 修饰的函数变为内联函数,同时和 `static` 类似, 只有本地文件可见, 允许多个文件内重复定义相同名的函数, 错误与 `static` 类似, 可能会报重复定义的错误。

2.编译与链接

在 `nemu/include/common.h` 中添加 `volatile static int dummy;`后重新编译 NEMU。重新编译后的 NEMU 有 1 个 `dummy` 变量的实体, 因为在这里用 `volatile` 定义了一个 `dummy`。

上一问题条件下在 `nemu/include/debug.h` 中添加 `volatile static int dummy;`。重新编译后的 NEMU 有 2 个 `dummy` 变量的实体。因为两个文件中都使用了 `volatile` 进行 `dummy` 的定义, 所以不会发生冲突。

修改添加的代码, 为两处 `dummy` 变量进行初始化 `volatile static int dummy = 0;`。然后重新编译 NEMU, 会报错。因为当 `volatile` 修饰的 `dummy` 被赋予了确定的值之后, 两个 `dummy` 就指向了同一个内存地址, 会发生重复定义的错误。

3.了解 MakeFile

makefile 的工作方式和编译链接的过程如下:

- 在当前目录下找名字叫 Makefile 或 makefile 的文件
- 如果找到, 它会找文件中的第一个目标文件, 并把这个文件作为最终的目标文件
- 如果文件不存在, 或是文件所依赖的后面的.o 文件的文件修改时间要比这个文件新, 那么, 他就会执行后面所定义的命令来生成 h 这个文件, 这个也就是重编译
- 如果文件所依赖的.o 文件也存在, 那么 make 会在当前文件中找目标为.o 文件的依赖性, 如果找到则再根据那一个规则生成.o 文件。
- .c 文件和.h 文件存在, 于是 make 会生成 .o 文件
- make 会一层一层去找文件的依赖关系, 直到最终编译出第一个目标文件, 若是过程中出现了错误, make 会直接退出并报错, 直到最后生成可执行文件 `nemu`

3.3 主要故障与调试

1.虚拟机版本不适配

在之前的 PA1 实验中，我都是在 Ubuntu14.04-amd64 版本的虚拟机下完成实验的。随后进行 PA2，完成 dummy 程序所需的指令实现后运行 dummy 指令，一直无法通过编译，并且会出现“\$ARCHIVE”字样。教师指出在之前的实验中亦有使用同样版本学生出现同样的错误，

2.与 IOE 相关的宏

在完成有关输入输出的工作后进行测试，但是无论如何均无法通过。在调试过程中我在每个相关函数中添加 Log 输出调试信息均没有输出，证明程序都没有进入这些相关函数。随后仔细阅读指导手册后我发现需要定义宏 HAS_IOE 后才能进入 init_device 函数，完成设备的初始化和输入输出功能的运行。这个问题只能说明在写代码的过程不能只顾着实现需要的函数，需要认真阅读指导手册，理解整个实验过程才能提高自身知识水平。

• 指导教师评定意见 •

一、原创性声明

本人郑重声明本报告内容，是由作者本人独立完成的。有关观点、方法、数据和文献等的引用已在文中指出。除文中已注明引用的内容外，本报告不包含任何其他个人或集体已经公开发表的作品成果，不存在剽窃、抄袭行为。

特此声明！

作者签字：

二、对课程设计的学术评语（教师填写）

三、对课程设计的评分（教师填写）

评分项目 (分值)	报告撰写 (30 分)	课设过程 (70 分)	最终评定 (100 分)
得分			

指导教师签字：_____