

华中科技大学

2018

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: CS1506

学 号: U201514639

姓 名: 刘科翰

电 话: 13018052550

邮 件: 524792973@qq.com

完成日期: 2018-12-28 周五上午



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	设计任务	2
2	PA1 简易调试器.....	3
2.1	功能实现的要点.....	3
2.2	必答题	3
2.3	主要故障与调试.....	5
3	PA2 冯诺依曼计算机系统.....	6
3.1	功能实现的要点.....	6
3.2	必答题	6
3.3	主要故障与调试.....	9
4	PA3 批处理系统.....	10

1 课程设计概述

1.1 设计任务

Programming Assignment:

PA0: 世界诞生的前夜: 开发环境的配置。

PA1: 开天辟地的篇章: 最简单的计算机。

PA2: 简单复杂的机器: 冯诺依曼计算机系统。

PA3: 穿越时空的旅程: 批处理系统。

PA4: 虚实交错的魔法: 分时多任务。

PA5: 天下武功唯快不破: 程序与性能。

2 PA1 简易调试器

2.1 功能实现的要点

task PA1.1: 实现单步执行, 打印寄存器状态, 扫描内存等基本指令。

task PA1.2: 实现简单的算术表达式求值。

task PA1.3: 实现扩展的表达式求值和监视点功能, 完善指令系统。

2.2 必答题

1.理解基础设施。我们通过一些简单的计算来体会简易调试器的作用。首先作以下假设: 假设你需要编译 500 次 NEMU 才能完成 PA。假设这 500 次编译当中, 有 90% 的次数是用于调试。假设你没有实现简易调试器, 只能通过 GDB 对运行在 NEMU 上的客户程序进行调试。在每一次调试中, 由于 GDB 不能直接观测客户程序, 你需要花费 30 秒的时间来从 GDB 中获取并分析一个信息。假设你需要获取并分析 20 个信息才能排除一个 bug。那么这个学期下来, 你将会在调试上花费多少时间? 由于简易调试器可以直接观测客户程序, 假设通过简易调试器只需要花费 10 秒的时间 从中获取并分析相同的信息。那么这个学期下来, 简易调试器可以帮助你节省多少调试的时间?

答: (1) 没有简易调试器下的情况:

一共要进行 450 次调试, 600s 排除一个 bug, 假设一次调试解决一个 bug, 那么在完成 nemu 的过程中, 需要花费 4500 分钟来调试, 也即 75 个小时用于调试。

(2) 有简易调试器下的情况:

解决一个 bug 需要 200s, 则完成 nemu 过程中需要 25 个小时用于调试。

2.查阅 i386 手册。

(1) EFLAGS 寄存器中的 CF 位是什么意思?

(2) ModR/M 字节是什么?

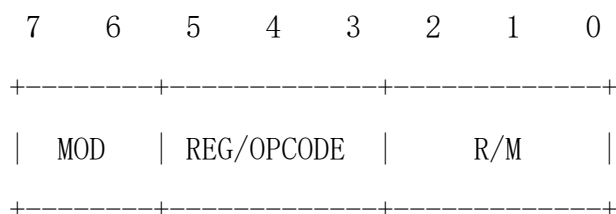
(3) mov 指令的具体格式是怎么样的?

答: (1) 阅读第二章 Basic Programming Model 中的 2.3 章节 Register 其中的 2.3.4 小节 Flag Register, 在 2.3.4.1 中有一个给 Appendix C 的跳转链接, 里面详细介绍了每一个标志物的定义。CF, 全称 Carry Flag, 是一个 Status Flags——状态标志位, 状态标

华中科技大学课程设计报告

志位使得当前指令的执行能够受到之前指令的结果的影响。CF 是标志高位进位或是借位信息的一个标志位。

(2) i386 手册 17.2.1 ModR/M and SIB Bytes 小节中介绍了 ModR/M and SIB 字节的含义和作用, 其中 ModR/M 字节一般紧跟着指令操作码, 这个字节一般包含着三个方面的信息, 一共分为三个区域: 分别为 mod 区、reg 区, r/m 区。mod 区占 3 个 bits, 与 r/m 区的信息结合起来可以指示 8 个寄存器和 24 个索引模式, 一个 32 种可能的情况。reg 区占 3 个 bits, 可以用来表示一个寄存器也可以作为操作码 opcode 的补充, 而这个由这条指令的第一个字节 opcode 决定。而 r/m 区占最后的 3 个 bits, 可以指向一个作为操作数的寄存器, 或者可以成为寻址模式中的一部分编码与 mod 区一起发挥作用。具体的字节信息如下所示:



(3) i386 手册 17.2.2.11 Instruction Set Details 里面详细地讲解了每一种指令, 具体到 MOV 中, MOV 指令的格式为: MOV DES, SRC, MOV 指令将第二个操作数 SRC 的值拷贝到第一个操作数 DES 处。

3.shell 命令。

- (1) 完成 PA1 的内容之后, nemu/ 目录下的所有 .c 和 .h 文件总共有多少行代码?
- (2) 你是使用什么命令得到这个结果的?
- (3) 和框架代码相比, 你在 PA1 中编写了多少行代码?
- (4) 除去空行之外, nemu/ 目录下的所有 .c 和 .h 文件总共有多少行代码?

答: (1) 完成 PA1 后, nemu/目录下总共有 4534 行代码 (包括空行、注释)。

(2) 使用的命令: `find . -name "*.c" | xargs cat | wc -l`

(3) checkout 到 pa0 之后执行 (2) 中的命令, 得到的行数为 3833 行, 故在 pa1 中总共编写了 701 行代码。

(4) 除去空行之后为 3804/3114 行代码。使用的命令为: `find . -name "*.c" | xargs cat | grep -v "\s$" | wc -l`。可见编写代码中又多加了几行无用的空行。

4.请解释 gcc 中的 -Wall 和 -Werror 有什么作用? 为什么要使用 -Wall 和 -Werror?

答: (1) -Wall 选项是一系列警告编译选项的集合, 本次实验中比较常见的警告有[-Wunused], [-Wunused]是也是一系列选项的集合, 用来警告存在一个定义了却未使用的局部变量或者一个函数的参数在函数的实现中并未被用到或者一个显式计算表达式的结果未被使用等等; 还有[-Wimplicit], 它也是一个选项的集合, 用于警告在声明函数却未指明函数返回类型时给出警告或者是在函数声明前调用该函数时给出警告等等。在-Wall 选项集合中还有很多其他的选项集合, 在编译调试时是很有用的工具。

(2) -Werror 是将所有的 Warning 都当成 Error 来处理, 视警告为错误, 只要有警告存在程序都会终止执行, 避免潜藏的 Fault。

2.3 主要故障与调试

2.3.1 寻找主运算符出错

出错原因: 忽略了运算符之间的优先级。

解决方案: 在编写寻找主运算符的过程中, 除了排除括号内的运算符和非运算符, 剩下的就需要根据优先级进行选择, 具体优先级参考 c 语言各个运算符的优先级。

2.3.2 解引用在简单表达式可以用, 复杂则错误

出错原因: 未将解引用当成一个最高优先级的运算符来看待, 而是把它放在 eval 函数的一个 if 分支中进行分流, 实现思想有误。

解决方案: 把解引用当成一个具有最高优先级的运算符, 在寻找主运算符的过程中将它纳入。在进行计算时, 由于它是单目运算符, 需要与双目运算符进行区分。

2.3.3 计算表达式偶尔出错

出错原因: 没有清空 tokens 数组中每一个元素的缓冲区, 以至于还保留着上一次分析的 token 的信息。没有对有效 token 之间的空格 token 进行处理。

解决方案: 每次 make_tokens 的时候, 注意清空缓冲区。时刻注意空格的处理方式。

3 PA2 冯诺依曼计算机系统

3.1 功能实现的要点

task PA2.1: 在 NEMU 中运行第一个 C 程序 dummy。

task PA2.2: 实现更多的指令，在 NEMU 中运行所有 cputest 通过。

task PA2.3: 完善 IOE，运行 slide、打字小游戏、超级玛丽，注意跑分情况。

3.2 必答题

1.编译与链接在 nemu/include/cpu/rtl.h 中，你会看到由 static inline 开头定义的各种 RTL 指令函数。选择其中一个函数，分别尝试去掉 static，去掉 inline 或去掉两者，然后重新进行编译，你可能会看到发生错误。请分别解释为什么这些错误会发生/不发生？你有办法证明你的想法吗？

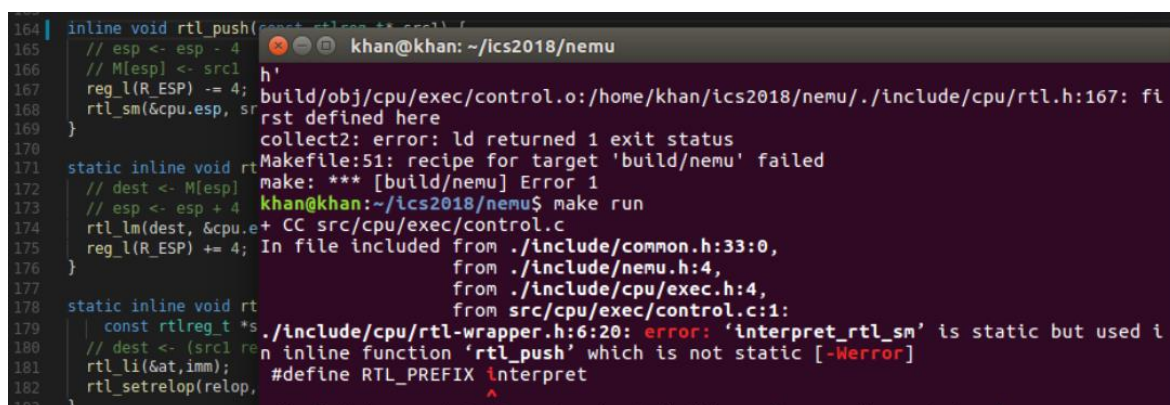
答：本问题选择了 rtl_push 函数进行实验。

(1) 去掉 inline，报的错误是函数定义未使用。

(2) 去掉 static，报的错误是内联的非 static 的 push 函数调用了其他 static 函数。

(3) 去掉 inline 和 static，错误是函数重复定义 multiple definition。

static 函数限制了文件的使用范围，静态函数只是在声明他的文件当中可见，不能被其他文件所用，如果非 static 函数调用了 static 函数，就会破坏这种保护机制。所以去掉 static 会报这个错误。具体报错如图 3-1 所示。



```
164 | inline void rtl_push(const rtlreg_t *src) {
165 |     // esp <- esp - 4
166 |     // M[esp] <- src[0]
167 |     reg_l(R_ESP) -= 4;
168 |     rtl_sm(&cpu.esp, src);
169 | }
170 |
171 | static inline void rtl_push(const rtlreg_t *src) {
172 |     // dest <- M[esp]
173 |     // esp <- esp + 4
174 |     rtl_lm(dest, &cpu.esp);
175 |     reg_l(R_ESP) += 4;
176 | }
177 |
178 | static inline void rtl_push(const rtlreg_t *src) {
179 |     // dest <- (src[0] << 16) | (src[1] << 8) | src[2]
180 |     rtl_li(&dest, imm);
181 |     rtl_setrelop(relop, dest, src);
182 | }
```

```
khan@khan: ~/ics2018/nemu
build/obj/cpu/exec/control.o: /home/khan/ics2018/nemu/./include/cpu/rtl.h:167: fl
rst defined here
collect2: error: ld returned 1 exit status
Makefile:51: recipe for target 'build/nemu' failed
make: *** [build/nemu] Error 1
khan@khan:~/ics2018/nemu$ make run
CC src/cpu/exec/control.c
In file included from ./include/common.h:33:0,
                 from ./include/nemu.h:4,
                 from ./include/cpu/exec.h:4,
                 from src/cpu/exec/control.c:1:
./include/cpu/rtl-wrapper.h:6:20: error: 'interpret_rtl_sm' is static but used i
n inline function 'rtl_push' which is not static [-Werror]
#define RTL_PREFIX interpret
```

图 3-1 去掉 static 关键字报错

inline 函数代替了宏的功能，避免调用函数时对栈空间的反复占用，static 关键字

华中科技大学课程设计报告

避免了重复定义的报错，但是编译器会检查每一个 include rtl.h 的文件中有没有使用到对应的 static 函数，这些 static 函数其实都是对应 include 的文件私有的，彼此没有关系，故在一个文件中没有找到这个函数的话就会认为这个函数定义了但是没有使用。使用 inline 关键字会默认函数已经调用过，需要检查函数是否可以内联。具体报错如图 3-2 所示。

```
164 | static void rtl_push(const rtlreg_t* src1) {
165 |     // esp <- esp - 4
166 |     // M[esp] <- src1
167 |     reg_l(R_ESP) -= 4;
168 |     rtl_sm(&cpu.esp, src1);
169 | }
170 |
171 | static inline void rtl_push(const rtlreg_t* src1) {
172 |     // dest <- M[esp]
173 |     // esp <- esp + 4
174 |     rtl_lm(dest, &cpu.esp);
175 |     reg_l(R_ESP) += 4;
176 | }
177 |
178 | static inline void rtl_push(const rtlreg_t* src1) {
179 |     const rtlreg_t* s = src1;
180 |     // dest <- (src1 << 8)
181 |     rtl_li(&at, imm);
182 |     rtl_setrelop(relop, s, &at);
183 | }
184 | //取出符号位
185 | static inline void rtl_push(const rtlreg_t* src1) {
186 |     // dest <- src1[wid]
187 |     uint32_t offset = 8
188 |     uint32_t temp = (*src1 << offset) >> offset;
189 |     *dest = temp & 0x1;
190 | }
```

```
khan@khan: ~/ics2018/nemu
./build/nemu -l ./build/nemu-log.txt -d /home/khan/ics2018/nemu/tools/qemu-diff/
build/qemu-so
[src/monitor/monitor.c,54,load_default_img] No image is given. Use the default b
uild-in image.
[src/monitor/monitor.c,33,welcome] Debug: OFF
[src/monitor/monitor.c,36,welcome] Build time: 16:24:38, Dec 28 2018
Welcome to NEMU!
For help, type "help"
(nemu) q
khan@khan:~/ics2018/nemu$ make run
+ CC src/cpu/exec/control.c
+ CC src/cpu/exec/data-mov.c
+ CC src/cpu/exec/system.c
In file included from ./include/cpu/decode.h:6:0,
from ./include/cpu/exec.h:9,
from src/cpu/exec/system.c:1:
./include/cpu/rtl.h:164:13: error: 'rtl_push' defined but not used [-Werror=unus
ed-function]
static void rtl_push(const rtlreg_t* src1) {
^
cc1: all warnings being treated as errors
Makefile:31: recipe for target 'build/obj/cpu/exec/system.o' failed
make: *** [build/obj/cpu/exec/system.o] Error 1
```

图 3-2 去掉 inline 关键字报错

去掉两者，则该函数定义变成一个普通函数定义，但是该函数在头文件中，会被很多其他文件 include，会造成普通函数的大量重复定义，所以会报错。具体报错如图 3-3 所示。

```
164 | void rtl_push(const rtlreg_t* src1) {
165 |     // esp <- esp - 4
166 |     // M[esp] <- src1
167 |     reg_l(R_ESP) -= 4;
168 |     rtl_sm(&cpu.esp, src1);
169 | }
170 |
171 | static inline void rtl_push(const rtlreg_t* src1) {
172 |     // dest <- M[esp]
173 |     // esp <- esp + 4
174 |     rtl_lm(dest, &cpu.esp);
175 |     reg_l(R_ESP) += 4;
176 | }
177 |
178 | static inline void rtl_push(const rtlreg_t* src1) {
179 |     const rtlreg_t* s = src1;
180 |     // dest <- (src1 << 8)
181 |     rtl_li(&at, imm);
182 |     rtl_setrelop(relop, s, &at);
183 | }
184 | //取出符号位
185 | static inline void rtl_push(const rtlreg_t* src1) {
186 |     // dest <- src1[wid]
187 |     uint32_t offset = 8
```

```
khan@khan: ~/ics2018/nemu
+ CC src/cpu/exec/cc.c
+ CC src/cpu/exec/special.c
+ CC src/cpu/exec/prefix.c
+ CC src/cpu/exec/exec.c
+ CC src/cpu/exec/arith.c
+ CC src/cpu/exec/logic.c
+ CC src/cpu/decode/decode.c
+ CC src/cpu/decode/modrm.c
+ CC src/cpu/intr.c
+ LD build/nemu
build/obj/cpu/exec/data-mov.o: In function 'rtl_push':
/home/khan/ics2018/nemu/./include/cpu/rtl.h:167: multiple definition of 'rtl_pus
h'
build/obj/cpu/exec/control.o: /home/khan/ics2018/nemu/./include/cpu/rtl.h:167: fi
rst defined here
build/obj/cpu/exec/system.o: In function 'rtl_push':
/home/khan/ics2018/nemu/./include/cpu/rtl.h:167: multiple definition of 'rtl_pus
h'
build/obj/cpu/exec/control.o: /home/khan/ics2018/nemu/./include/cpu/rtl.h:167: fi
rst defined here
```

图 3-3 去掉 static、inline 关键字报错

2. 编译与链接:

(1) 在 nemu/include/common.h 中添加一行 volatile static int dummy; 然后重新编译 NEMU. 请问重新编译后的 NEMU 含有多少个 dummy 变量的实体? 你是如何得到这个结果的?

华中科技大学课程设计报告

答: 重新编译后的 NEMU 含有 30 个 dummy 的实体。

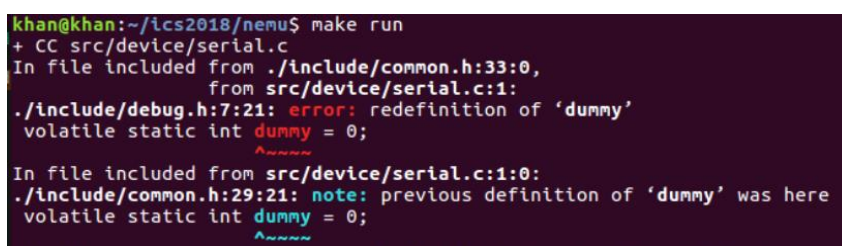
流程: 在未添加任何代码时, 对 nemu 的可执行文件进行反汇编, 反汇编只有 dummy 程序相关的几个 dummy 关键词, 没有变量的实体。修改代码运行 nemu 后, 对 nemu 的可执行文件进行反汇编, 观察到在 bss 段处有 30 个 dummy 变量的实体, 在采用段式内存管理的架构中, BSS 段通常是指用来存放程序中未初始化的全局变量的一块内存区域, 属于静态内存分配。

(2) 添加上题中的代码后, 再在 nemu/include/debug.h 中添加一行 volatile static int dummy; 然后重新编译 NEMU. 请问此时的 NEMU 含有多少个 dummy 变量的实体? 与上题中 dummy 变量实体数目进行比较, 并解释本题的结果。

答: 再添加一行代码编译 NEMU 仍然只有 30 个 dummy 变量的实体。因为 common.h 中同时也 include 了 debug.h, debug.h 中也 include 了 common.h, 所以这两句声明相当于写在同一个文件中。并且 c 语言编译器对一模一样的变量声明不会报错, 只有类型不一样或者重复定义才会报错。

(3) 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. 你发现了什么问题? 为什么之前没有出现这样的问题?

答: 报错: redefinition of 'dummy' 以及 previous definition of 'dummy' 的错误出现, 根据报错很明显可以得知, 无论在什么地方, 同一个项目中变量是不可以重复定义的。之前只是声明变量, 而且类型完全相同, 编译器会忽略这种情况, 当作只有一个变量声明了。报错情况如图 3-4 所示。



```
khan@khan:~/ics2018/nemu$ make run
+ CC src/device/serial.c
In file included from ./include/common.h:33:0,
                 from src/device/serial.c:1:
./include/debug.h:7:21: error: redefinition of 'dummy'
volatile static int dummy = 0;
                    ~~~~~
In file included from src/device/serial.c:1:0:
./include/common.h:29:21: note: previous definition of 'dummy' was here
volatile static int dummy = 0;
                    ~~~~~
```

图 3-4 初始化后报错情况

3. 了解 Makefile 请描述你在 nemu/目录下敲入 make 后, make 程序如何组织.c 和.h 文件, 最终生成可执行文件 nemu/build/nemu。

答: Makefile 里面用.PHONY 明确声明几个用于执行的伪目标, 重写了一些模糊的规则, 使得 make run、make gdb、make clean、make app 执行特定的指令。在 makefile 中生成可执行文件的伪目标是 app, make 默认执行这个目标。run 伪目标中包含了 app 伪目标所需要执行的指令而已。

华中科技大学课程设计报告

设置系统变量 CC 使用 gcc 编译器编译, 设置 LD 使用 gcc 链接, 通过自定义变量 INCLUDE 与 CFLAGS 以及其内部使用到的一些指明路径的变量, 构建了一条链接 c 程序和对应头文件和开启了一系列检查标志具有特定编译格式的 shell 命令。然后 Binary 变量中调用了这条指令, 从而完成了编译链接, 最终生成可执行文件。

3.3 主要故障与调试

3.3.1 初次运行 dummy 系统提示找不到 sys/cdefs.h

出错原因: 系统缺少对应的包

解决方案: 安装 libc6-dev-i384 这个包即可解决问题。

3.3.2 算术右移错误

出错原因: 算术右移未考虑操作数宽度。

解决方案: 对不同宽度的操作数都严格执行算术右移。

3.3.3 跑 coremark 时分数输出错误

出错原因: 检查了 printf 表达式中单个元素都没问题, 只有合起来有问题, 汇编指令采用的执行函数错误。

解决方案: 检查到 imul 指令中使用到三个操作数的情况下错误的使用了 imul2 函数, 使得计算结果错误, 将其改用为 imul3 来执行即解决问题。

3.3.4 Videotest 正常, 之后的超级玛丽、slider、打字游戏不正常

出错原因: 实现 video_write 函数时忽略了所画矩形超出屏幕宽度的情况。

解决方案: 在实现 video_write 函数加入对边界的限制, 即可实现所有测试程序的正常运行。

3.3.5 Difftest 突然找不到 qemu 的句柄 handle

出错原因: make clean 会清除掉编译好的 qemu-so。

解决方案: 进入 nemu/tools/qemu 重新编译 qemu。

4 PA3 批处理系统

• 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：_____