

# 计算机系统基础 PA实验介绍

## Introduction to ICS Programming Assignment

- PA概述
- PA详细介绍
- 总结

- coursera NJU ICS

- <https://class.coursera.org/njuics1-001>

## ■ PA概述

# ICS背景

- 国外高校与ICS相关的课程与实验设置
  - UCB - Great Ideas in Computer Architecture
    - MapReduce, MIPS指令集模拟器, 矩阵相乘算法实现与优化...
  - CMU - ICS
    - CSAPP labs(datalab, bomblab, buflab...)
  - MIT - Computation Structures
    - 要求完成对一个RISC处理器在门级的设计
  - 系统性和完整性不如操作系统课程实验(JOS等)
- 国内高校: 推广中

# PA简介

- 理解"程序如何在计算机上运行"的根本途径是实现一个完整的计算机系统
- PA任务: 实现NEMU(NJU EMUlator)
  - 功能完备(但经过简化)的x86全系统模拟器
  - 包括1个准备实验(配置实验环境)以及4部分连贯的实验内容
    - 简易调试器(过渡实验)
    - 指令系统
    - 存储管理
    - 中断与I/O

# PA简介

- 实验平台与工具
  - IA-32 + GNU/Linux + gcc + C
  - 其它工具: gdb, make, git
- 课程网站(发布实验讲义)
  - <http://fdu-ics.gitbooks.io/programming-assignment/content/>
- 框架代码
  - <https://github.com/fdu-ics/programming-assignment>

# PA简介

- 前导课程
  - 程序设计基础
- 与理论课紧密结合
  - 知识点覆盖度广: 约95%
    - 只有动态链接没有涉及
  - 并有部分延伸
- 觉得自己上课听懂了?
  - 做一做PA就知道

# PA目标

- ▶ 教学目标
  - ▶ 彻底颠覆对"程序运行"的认识
  - ▶ 体会ISA层次如何支撑起上层程序的运行
- ▶ 工程目标
  - ▶ 体会KISS(Keep It Simple, Stupid)法则
  - ▶ 体会调试公理
    - ▶ 机器永远是对的
    - ▶ 未测试代码永远是错的
- ▶ 科研目标
  - ▶ 阅读英文材料
  - ▶ 使用搜索引擎
  - ▶ RTFM



# PA实验方案

	持续时间/周	预计耗时/小时	代码量/行
PA0 – 开发环境配置	2	20	无
PA1 – 简易调试器	3	30	400
PA2 – 指令系统	5	50	800
PA3 – 存储管理	4	50	500
PA4 – 中断与I/O	3	30	300
总计	17	180	2000

- 预计耗时以中下水平的学生来估算
  - 保守估计200小时

# NEMU历史

## ■ 2014.08

- 问世版本
- 设计受到QEMU的启发
- 结合了GDB调试器的特性

## ■ 2015.08

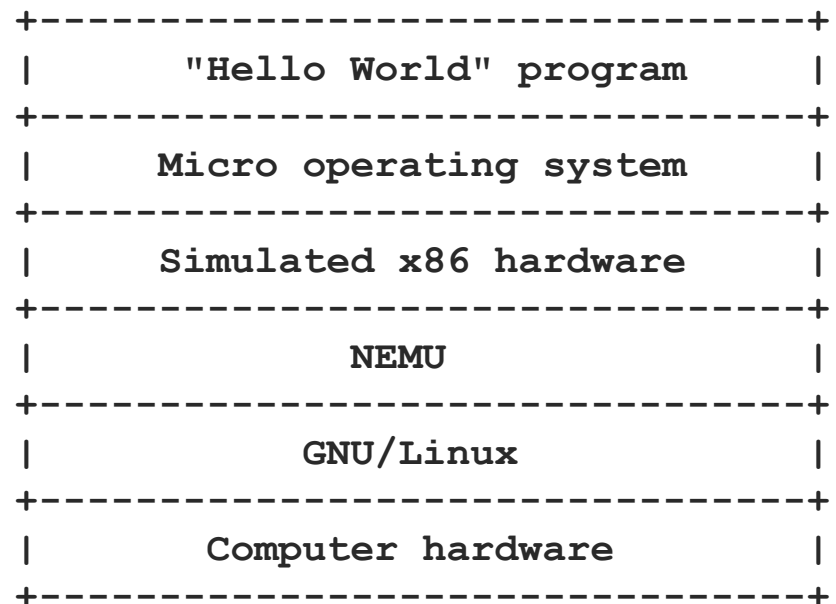
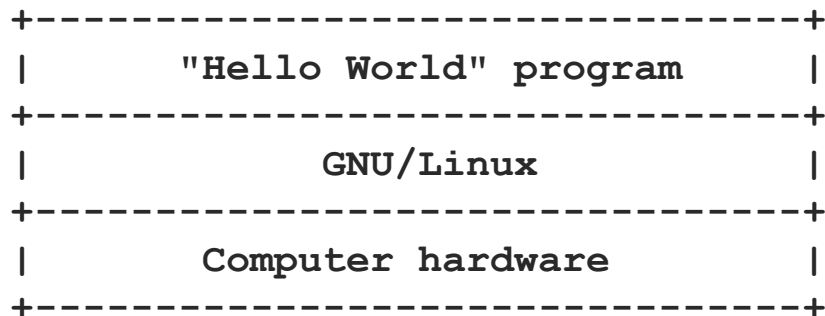
- 引入monitor模块
- 去掉了软件断点
- 重构了CPU模块(译码与执行)
- 加入DMA

# 程序, NEMU和操作系统

## ■ NEMU是一个模拟器

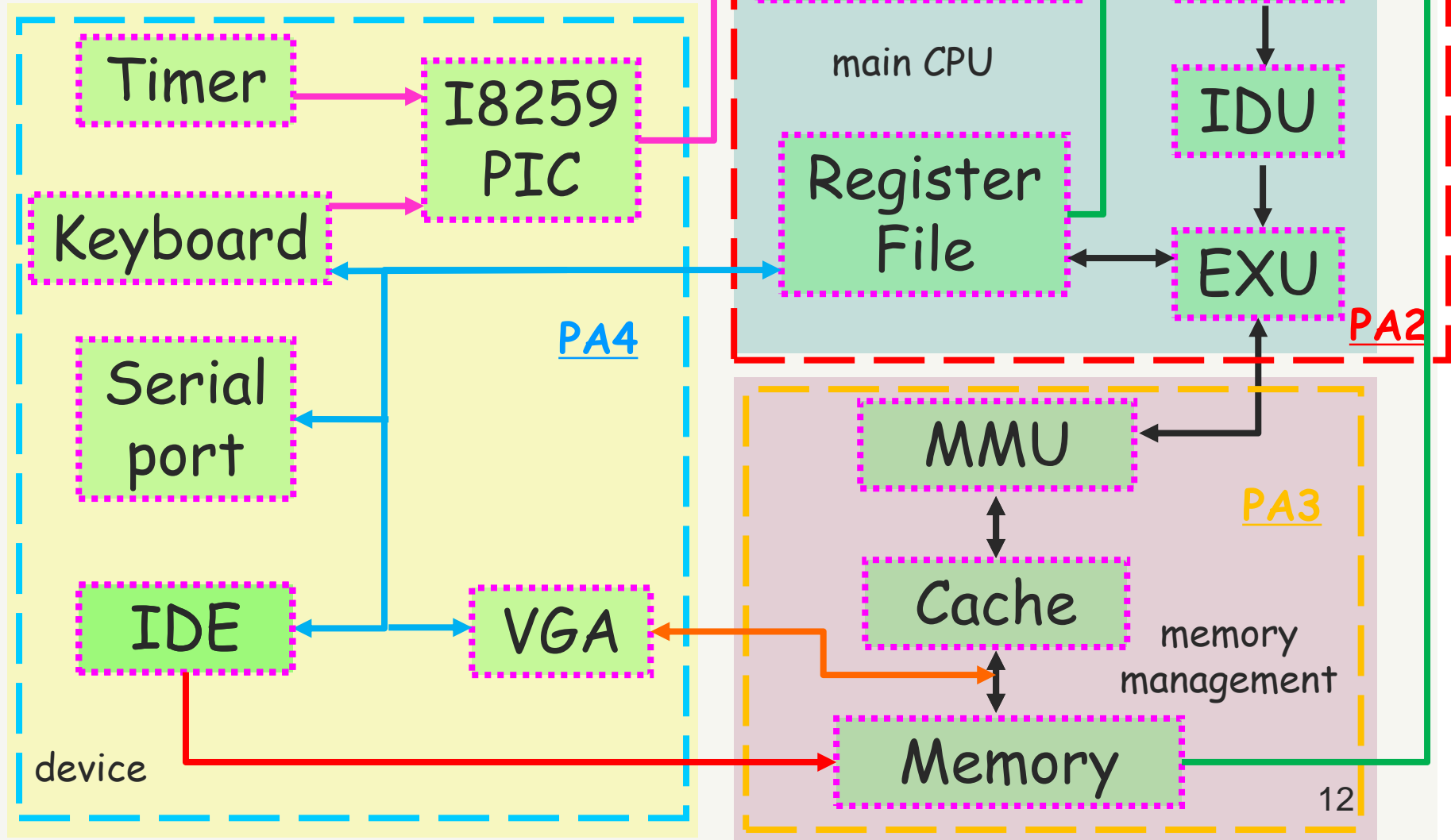
- 是一个普通的程序, 在GNU/Linux操作系统中运行
- 是一个特殊的程序, 虚拟出一个计算机系统, 其它程序可以在其中模拟执行

### ■ 模拟方式: 解释执行





a computer system



# NEMU特性

- ▶ 简易调试器(位于monitor中)
  - ▶ 单步执行, 打印寄存器/内存, 表达式求值, 监视点
- ▶ CPU核心
  - ▶ 完整的指令周期
  - ▶ 支持x86保护模式下的大部分常用指令(不支持实模式)
    - ▶ 不支持x87浮点指令
- ▶ 存储管理
  - ▶ DRAM(包含row buffer和burst的物理特性)
  - ▶ 两级联合cache
  - ▶ MMU
    - ▶ IA-32分段机制, IA-32分页机制(包含TLB)
    - ▶ 不支持保护机制

# NEMU特性

## ■ 中断/异常

- IA-32中断机制
- 不支持保护机制

## ■ 设备

- 时钟, 键盘, VGA, 串口, IDE, I8259 PIC
  - 大部分功能都不可编程
- 端口I/O, 内存映射I/O

# NEMU特性

- 实验中后期会结合kernel进行
  - 一个单核单任务微型操作系统的内核(简化自Nanos)
    - 2个设备驱动
      - Ramdisk, IDE
    - ELF32加载器
    - 分页存储管理
    - 简易文件系统
      - 文件数量, 大小皆固定, 没有目录
    - 6个系统调用
      - open, read, write, lseek, close, brk
- 软(kernel)硬(NEMU)结合理解计算机系统

# PA终极任务

- 在NEMU中运行仙剑奇侠传



- 在计算机看来, hello程序和仙剑奇侠传有什么区别?



## ■ PA详细介绍

# PA0 - 开发环境配置

- 从零开始安装实验平台, 配置开发环境
  - 安装GNU/Linux VM
  - 阅读vim教程, 使用vim编辑文件
  - 配置APT源, 安装实验工具
  - 阅读GNU/Linux入门教程
  - 编辑, 编译, 运行hello程序, 使用GDB
  - 配置ssh, 在host和VM之间传输文件
  - 下载PA框架代码, 配置git
  - 编译, 运行NEMU
- 按讲义内容操作即可, 没有难度

# 开发跟踪与抄袭检测

## ■ 通过git记录实验过程

- git log
- git diff

```
commit 4072d39e5b6c6b6837077f2d673cb0b5014e6ef9
Author: tracer-ics2015 <tracer@njuics.org>
Date: Sun Aug 24 14:30:31 2014 +0800
```

```
> compile NEMU
```

```
141220000
```

```
user
```

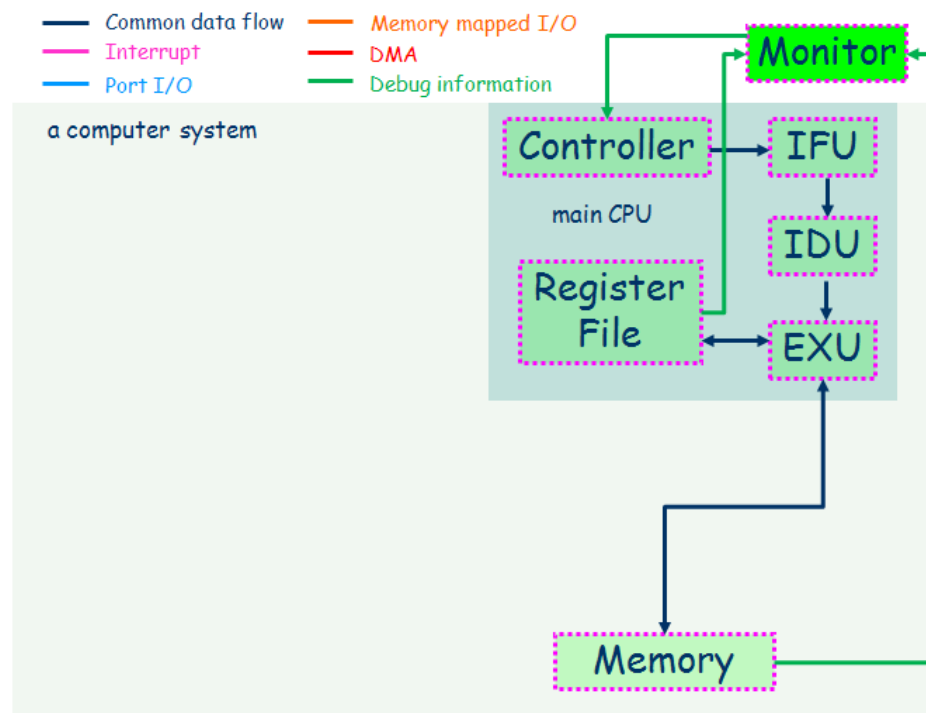
```
Linux debian 3.16.0-4-686-pae #1 SMP Debian 3.16.7-3 i686 GNU/Linux
```

```
14:30:31 up 3:44, 2 users, load average: 0.28, 0.09, 0.07
```

```
3860572d5cc66412bf85332837c381c5c8c1009f
```

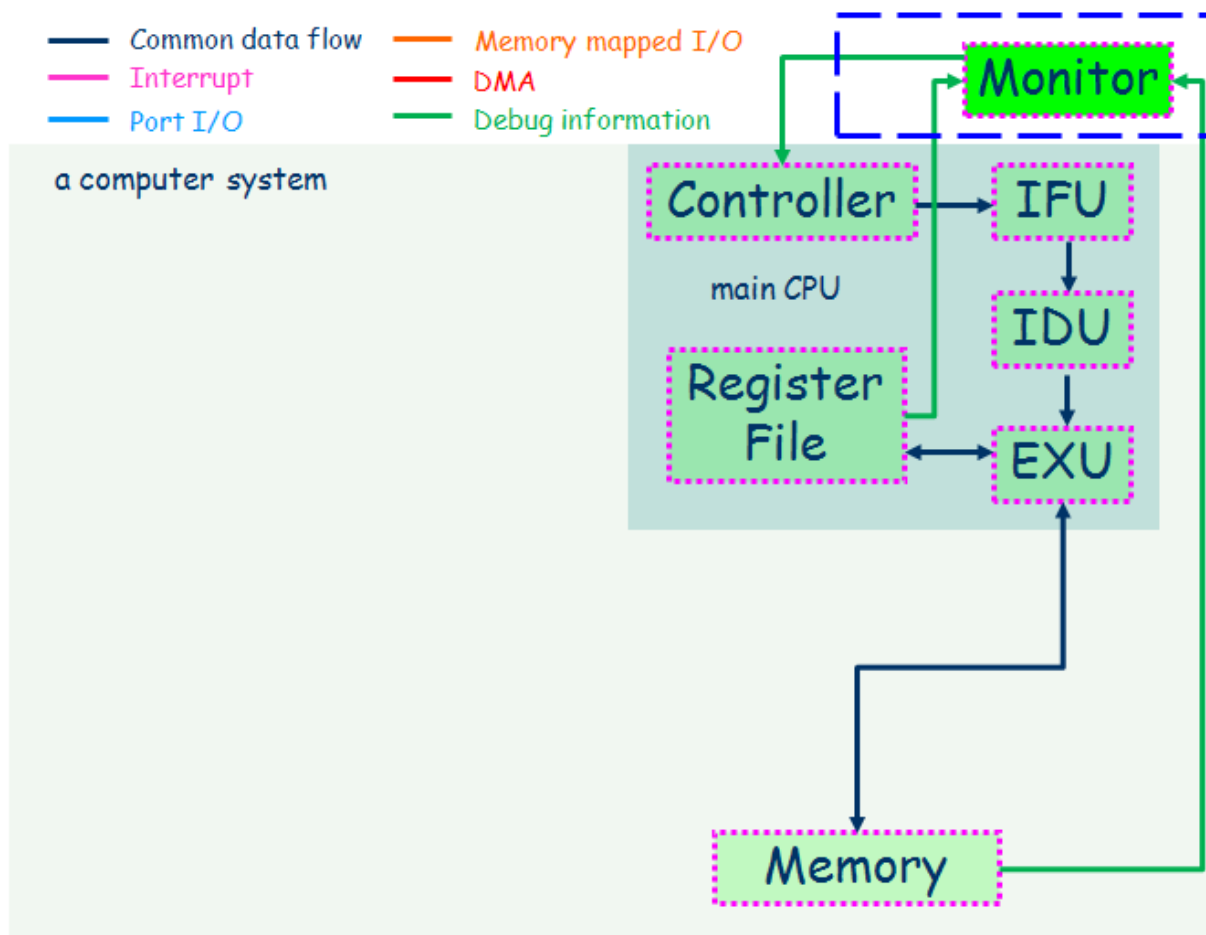
# NEMU框架代码

- 包含如下内容/部件
  - 寄存器堆
    - 未正确实现
  - 内存
    - `swaddr_read()`
    - `swaddr_write()`
  - 完整的指令周期
    - `cpu_exec()`
  - 简易调试器基本框架
    - `ui_mainloop()`
  - 设备相关(在PA4中使用)



# PA1 – 简易调试器

- 实现简易调试器, 为后续阶段的调试提供有用的手段



# PA1 – 简易调试器

命令	格式	使用举例	说明
帮助(1)	<code>help</code>	<code>help</code>	打印命令的帮助信息
继续运行(1)	<code>c</code>	<code>c</code>	继续运行被暂停的程序
退出(1)	<code>q</code>	<code>q</code>	退出NEMU
单步执行	<code>si [N]</code>	<code>si 10</code>	让程序单步执行 <code>N</code> 条指令后暂停执行, 当 <code>N</code> 没有给出时, 缺省为 <code>1</code>
打印程序状态	<code>info SUBCMD</code>	<code>info r</code> <code>info w</code>	打印寄存器状态 打印监视点信息
表达式求值	<code>p EXPR</code>	<code>p \$eax + 1</code>	求出表达式 <code>EXPR</code> 的值, <code>EXPR</code> 支持的运算请见 <a href="#">调试中的表达式求值</a> 小节
扫描内存(2)	<code>x N EXPR</code>	<code>x 10 \$esp</code>	求出表达式 <code>EXPR</code> 的值, 将结果作为起始内存地址, 以十六进制形式输出连续的 <code>N</code> 个4字节
设置监视点	<code>w EXPR</code>	<code>w *0x2000</code>	当表达式 <code>EXPR</code> 的值发生变化时, 暂停程序执行
删除监视点	<code>d N</code>	<code>d 2</code>	删除序号为 <code>N</code> 的监视点
打印栈帧链(3)	<code>bt</code>	<code>bt</code>	打印栈帧链

# PA1 – 简易调试器

- 任务1: 阅读框架代码, 了解
  - `ui_mainloop()` 用户界面主循环
    - 每次循环负责处理一条用户输入的调试命令
  - `cpu_exec()` 指令执行主循环
    - 每次循环负责解释执行一条用户程序的指令

# PA1 – 简易调试器

## ■ 任务2: 重新组织寄存器结构体

- struct/union的使用[2] (注: 数字表示该任务与课本中相应章节的内容相关)

```
/* TODO: Re-organize the 'CPU_state' structure to match the register
 * encoding scheme in i386 instruction format. For example, if we
 * access cpu.gpr[3]._16, we will get the 'bx' register; if we access
 * cpu.gpr[1]._8[1], we will get the 'ch' register. Hint: Use 'union'.
 * For more details about the register encoding scheme, see i386 manual.
 */

typedef struct {
    struct {
        uint32_t _32;
        uint16_t _16;
        uint8_t _8[2];
    } gpr[8];

    /* Do NOT change the order of the GPRs' definitions. */

    uint32_t eax, ecx, edx, ebx, esp, ebp, esi, edi;

    swaddr_t eip;
} CPU_state;
```



# PA1 – 简易调试器

- 任务3: 为简易调试器添加如下功能
  - 单步执行(si)
  - 打印寄存器(info r)
  - 扫描内存(x)
- 编程小练习
  - 使用字符串处理函数(strtok, sscanf, printf...)
  - 使用man查看函数的功能说明

# PA1 – 简易调试器 (学生作业)

```
Terminal
File Edit View Search Terminal Help
eip          0x10000f          0x10000f
(nemu) q
[12:05:47 ~/Templates/ics-homework/PA1/131220001/NEMU]$ make run
./nemu -d testcase/asm/mov 2>&1 | tee log.txt
(nemu) si 5
100000:  b8 00 00 00 00          movl $0x0,%eax
100005:  bb 00 00 00 00          movl $0x0,%ebx
10000a:  b9 00 00 00 00          movl $0x0,%ecx
10000f:  ba 00 00 00 00          movl $0x0,%edx
100014:  b9 00 80 00 00          movl $0x8000,%ecx
(nemu) info r
eax          0x0          0
ecx          0x8000      32768
edx          0x0          0
ebx          0x0          0
esp          0x2a882a0d    0x2a882a0d
ebp          0x4f9cb256    0x4f9cb256
esi          0x7db67b15    2109111061
edi          0x4b512425    1263608869
eip          0x100019      0x100019
(nemu) x 3 0x100000
0x00100000:  0x000000b8 0x0000bb00 0x00b90000
(nemu) █
[0] 0:make* "debian" 12:07 30-Jan-15
```

# PA1 – 简易调试器

- 任务4: 表达式求值
  - 使用正则表达式进行词法分析
    - 不需要理解正则表达式的原理
  - 通过递归进行求值(小学生算法)

```
<expr> ::= <decimal-number>
          | <hexadecimal-number>      # 以"0x"开头
          | <reg_name>                  # 以"$"开头
          | "(" <expr> ")"
          | <expr> "+" <expr>
          | <expr> "-" <expr>
          | <expr> "*" <expr>
          | <expr> "/" <expr>
          | <expr> "==" <expr>
          | <expr> "!=" <expr>
          | <expr> "&&" <expr>
          | <expr> "||" <expr>
          | "!" <expr>
          | "*" <expr>                  # 指针解引用
```

```
"4 + 3 * ( 2 - 1 )"
/*****/
case 1:
    "+"
    /   \
"4"     "3 * ( 2 - 1 )"

case 2:
    "*"
    /   \
"4 + 3"  "( 2 - 1 )"

case 3:
    "-"
    /   \
"4 + 3 * ( 2"  "1 )"

```

# PA1 – 简易调试器

- 任务5: 监视点
  - 链表的使用
  - 反复对待监视表达式进行求值即可
- 任务6: 熟悉i386手册

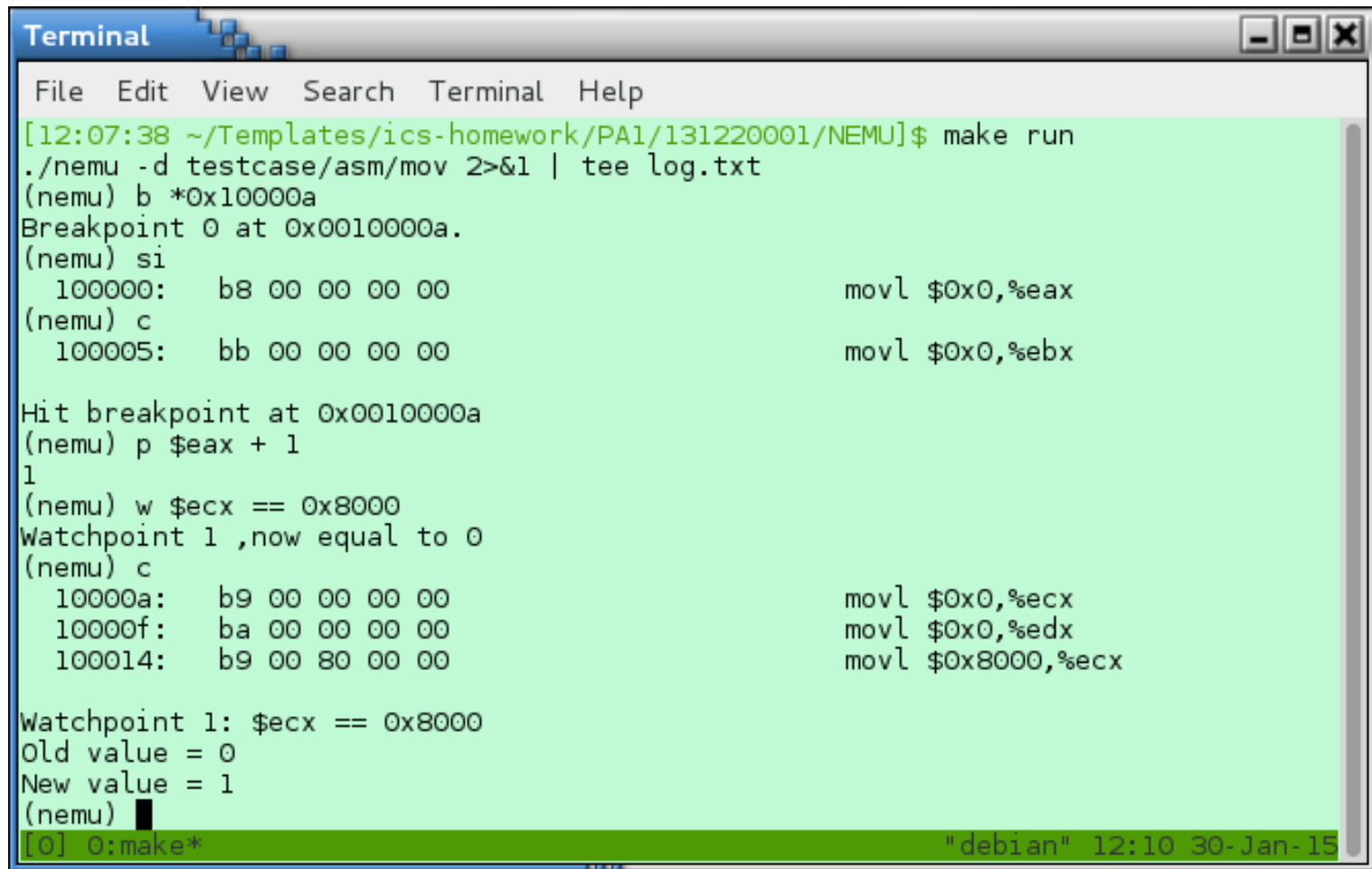
# PA1 – 简易调试器

## 必答题

你需要在实验报告中回答下列问题:

- 查阅i386手册 理解了科学查阅手册的方法之后, 请你尝试在i386手册中查阅以下问题所在的位置, 把需要阅读的范围写到你的实验报告里面:
  - EFLAGS寄存器中的CF位是什么意思?
  - ModR/M字节是什么?
  - mov指令的具体格式是怎么样的?
- shell命令 完成PA1的内容之后, nemu目录下的所有.c和.h和文件总共有多少行代码? 你是使用什么命令得到这个结果的? 和框架代码相比, 你在PA1中编写了多少行代码? (Hint: 使用 `git checkout` 可以回到"过去", 具体使用方法请查阅 `man git-checkout` ) 你可以把这条命令写入 `Makefile` 中, 随着实验进度的推进, 你可以很方便地统计工程的代码行数, 例如敲入 `make count` 就会自动运行统计代码行数的命令. 再来个难一点的, 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?
- 使用man 打开工程目录下的 `Makefile` 文件, 你会在 `CFLAGS` 变量中看到gcc的一些编译选项. 请解释gcc中的 `-Wall` 和 `-Werror` 有什么作用? 为什么要使用 `-Wall` 和 `-Werror` ?

# PA1 – 简易调试器 (学生作业)

A terminal window titled "Terminal" with a menu bar (File, Edit, View, Search, Terminal, Help) and standard window controls. The terminal output shows a user running 'make run' in a directory with path components like ~/Templates/ics-homework/PA1/131220001/NEMU. The user then enters 'b \*0x10000a' to set a breakpoint. The program starts, and the user enters 'si' to step through instructions. The terminal shows assembly instructions: 'movl \$0x0,%eax' at address 100000 and 'movl \$0x0,%ebx' at address 100005. The breakpoint is hit at address 0x0010000a. The user enters 'p \$eax + 1' and sees the value 1. Then, the user enters 'w \$ecx == 0x8000' to set a watchpoint. The terminal shows instructions: 'movl \$0x0,%ecx' at 10000a, 'movl \$0x0,%edx' at 10000f, and 'movl \$0x8000,%ecx' at 100014. The watchpoint is triggered because the new value of \$ecx is 1, not 0. The terminal shows 'Watchpoint 1: \$ecx == 0x8000', 'Old value = 0', and 'New value = 1'. The user enters a single character, and the prompt returns. The status bar at the bottom shows '[0] 0:make\*' on the left and '"debian" 12:10 30-Jan-15' on the right.

```
Terminal
File Edit View Search Terminal Help
[12:07:38 ~/Templates/ics-homework/PA1/131220001/NEMU]$ make run
./nemu -d testcase/asm/mov 2>&1 | tee log.txt
(nemu) b *0x10000a
Breakpoint 0 at 0x0010000a.
(nemu) si
100000:  b8 00 00 00 00          movl $0x0,%eax
(nemu) c
100005:  bb 00 00 00 00          movl $0x0,%ebx

Hit breakpoint at 0x0010000a
(nemu) p $eax + 1
1
(nemu) w $ecx == 0x8000
Watchpoint 1 ,now equal to 0
(nemu) c
10000a:  b9 00 00 00 00          movl $0x0,%ecx
10000f:  ba 00 00 00 00          movl $0x0,%edx
100014:  b9 00 80 00 00          movl $0x8000,%ecx

Watchpoint 1: $ecx == 0x8000
Old value = 0
New value = 1
(nemu) █
[0] 0:make*                                     "debian" 12:10 30-Jan-15
```