华中科技大学

2024

系统能力培养 课程实验报告

题　　目：	指令模拟器
专　　业：	计算机科学与技术
班　　级：	CS2010 班
学　　号：	U202015596
姓　　名：	杨晨
电　　话：	18602709277
邮　　件：	U202015596@hust.edu.cn
完成日期：	2024-1-15

计算机科学与技术学院

目 录

1 课程实验概述 1

1.1 课设目的 1
1.2 课设任务 1
1.3 实验环境 2

2 实验方案设计 3

2.1 PA1 3
2.1.1 设计 3
2.2 PA2 6
2.2.1 设计 6
2.3 PA3 9
2.3.1 设计 9

3 实验结果与结果分析 11

3.1 PA1 11
3.1.1 实验结果 11
3.1.2 问题回答 13
3.2 PA2 15
3.2.1 实验结果 15
3.2.2 问题回答 18
3.3 PA3 20
3.3.1 实验结果 20
3.3.2 问题回答 24

参考文献 25

1　课程实验概述

1.1 课设目的

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。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/mips32/riscv32 架构相应的教学版子集, 指导学生实现一个经过简化但功能完备的 x86/mips32/riscv32 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.。NEMU 受到了 QEMU 的启发, 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:

- 图灵机与简易调试器
- 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

1.2 课设任务

1）世界诞生前夜：开发环境配置

安装虚拟机，阅读实验手册，设置好初始选项

2）开天辟地的篇章：最简单的计算机

- PA1.1 简易调试器
- PA1.2 表达式求值
- PA1.3 监视点与断点

3）简单复杂的机器：冯诺依曼计算机系统

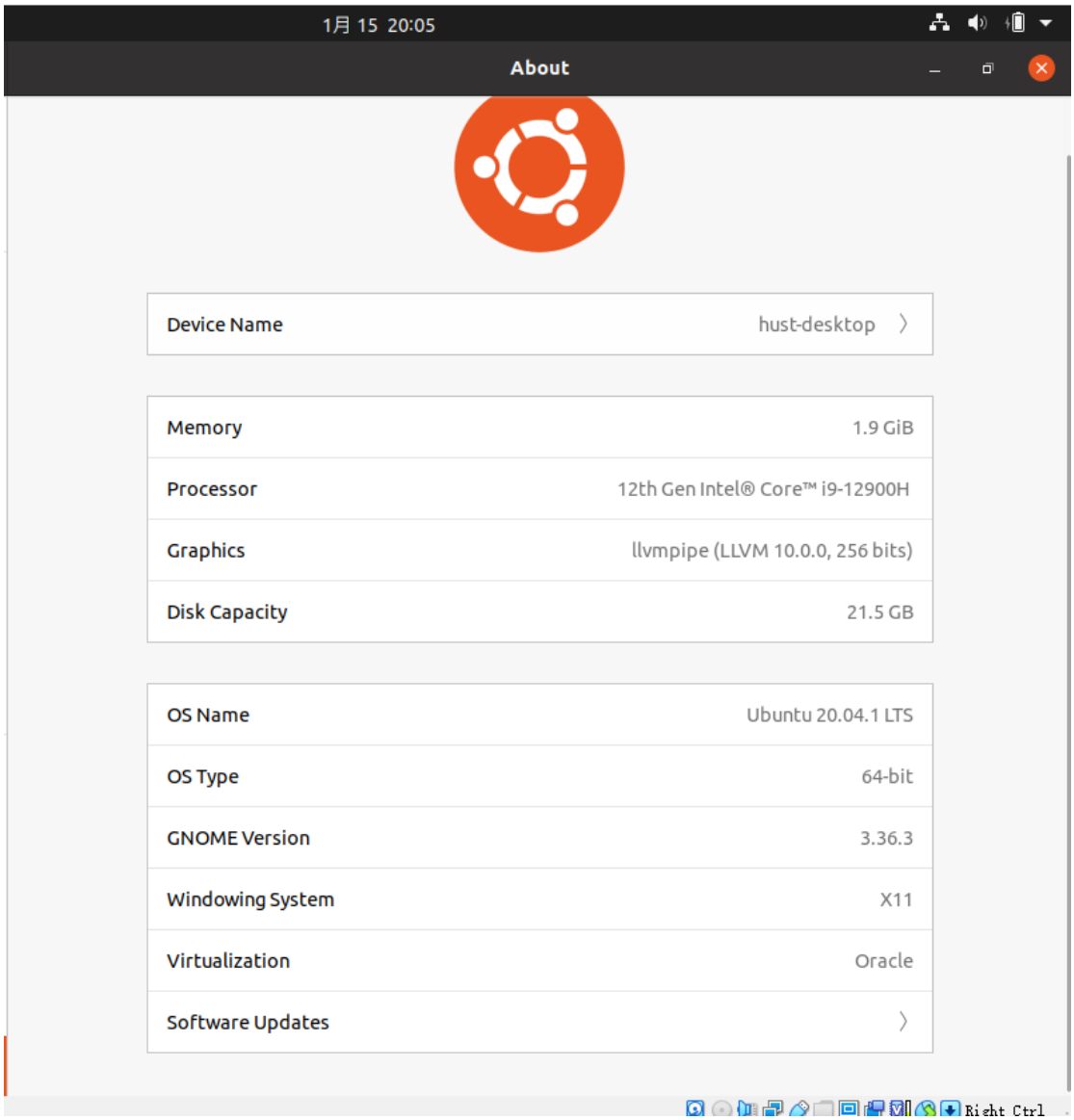
- PA2.1 运行第一个 C 程序
- PA2.2 丰富指令集，测试所有程序
- PA2.3 实现 I/O 指令，测试打字游戏

4）穿越时空的旅程：批处理系统

- PA3.1 实现系统调用
- PA3.2 实现文件系统

- PA3.3 运行仙剑奇侠传

1.3 实验环境



2　实验方案设计

2.1 PA1

2.1.1 设计

PA1 目标是开发一个基础调试器，它类似于一个简版的 gdb。此调试器应具备以下功能：

1)单步执行

2)打印寄存器状态

3)扫描内存

4)表达式求值

5)监视点设置

PA1.1 需要设计并实现以下功能：单步执行、打印寄存器状态、扫描内存。这些功能的相关函数应在 ui.c 文件中声明和定义。

首先, 需要定义三个关键功能：单步执行、打印寄存器状态和扫描内存。在 ui.c 文件中，这些功能分别由 cmd_si、cmd_info 和 cmd_x 函数实现。具体来说：

cmd_si：通过调用 cpu_exec 函数执行指定数量的步骤。

cmd_info：利用 isa_reg_display 函数输出寄存器的名称和值。在 isa_reg_display 内部，使用 reg_name 函数和 reg_l 宏来打印寄存器的名称和值。

cmd_x：解析输入参数，并通过 vaddr_read 函数读取相应地址的值，随后进行打印。

PA1.2 实现表达式求值功能是整个 PA1 的核心任务，该功能的完整实现应位于 expr.c 文件。具体实现步骤包括：补充正则表达式处理、实现括号匹配功能、

编写并优化递归求值函数 eval。最后，将这些组件协调整合，形成完整的 expr 函数。

要实现表达式求值，首先需在 expr.c 中定义规则及其对应的正则表达式。以十六进制数为例，其正则表达式与 token 类型定义为 {"0[Xx][0-9a-fA-F]+", TK_HEX}。然后，实现 make_token 函数，为算术表达式的各类 token 类型添加规则。在成功识别 token 后，按顺序将其信息记录至 tokens 数组。

进一步，开发括号匹配函数 check_parentheses 和运算符优先级函数 op_priority。后者应根据 C 语言的优先级规则确定操作符的优先级。

完成上述辅助功能后，着手编写递归求值函数 eval。此函数接收参数 p 和 q，处理逻辑如下：

若 p > q，直接返回 0，并将 success 置为 false；

若 p == q，则检查该 token 是否为整数、十六进制整数或寄存器。如果是，返回相应值；如果不是，将 success 置为 false 并返回 0；

进行括号匹配检查。若匹配，返回 eval(p+1, q-1)；若不匹配，执行寻找主操作符的逻辑；

若找到主操作符，根据该操作符返回相应的值；若未找到，将 success 置为 false 并返回 0。

最终，将 eval 函数与 make_token 函数整合入 expr 函数中，完成整个表达式求值框架的开发。

PA1.3 实现监视点功能首先要补充 watchpoint 结构，并开发相关的操作函数。随后，在 ui.c 文件中编写处理监视点命令的函数。最终，确保能够在监视点状态发生变化时实现暂停逻辑。

为实现监视点功能，需首先完善 watchpoint 结构，增加必要的变量。由于

监视点需要追踪表达式的值变化，因此：

引入一个 char 数组来存储表达式并设定一个变量以保存表达式的旧值。

在 watchpoint.c 文件中，实现以下函数：

new_wp：创建新的监视点，并将 free_ 链表中的节点移至 head 链表。

free_wp：释放指定的监视点，即从 head 链表中找到对应节点并将其移至 free_ 链表。这两个函数都通过链表的插入和删除操作实现，且为了操作便捷，都采用头插法。

print_wp：打印监视点信息。

在 ui.c 文件中，利用上述函数实现 cmd_x 和 cmd_d 命令，并完善 cmd_info 函数。

最后，在 cpu_exec 中加入监视点状态变化导致程序暂停的逻辑。具体来说，程序每执行一步就对所有监视点的表达式进行求值，并将新值与旧值进行比较。若发现值有变化，则将 nemu 的状态设置为 NEMU_STOP，从而实现监视点功能。

2.2 PA2

2.2.1 设计

PA2.1 首先，运行 make 命令以执行 dummy 程序。该命令执行后可能会导致程序中断，但会在 build 文件夹生成相应的反汇编文件。检查这些反汇编文件，识别出尚未实现的指令。参考相关手册，确认所有需要实现的伪指令。经确认，dummy 程序需要实现的新指令包括 auipc, addi, jal, 和 jalr。

按照以下步骤实现这些指令：

1.在 all-instr.h 中定义这些新指令的执行函数。

2.在 exec.c 的 opcode_table 中添加这些新指令。

3.在 decode.c 中编写这些指令的译码辅助函数。

4.在 rtl.h 中更新和完善 rtl 指令集。

5.在 compute.c、control.c 等文件中利用 rtl 指令实现正确的执行辅助函数。

完成上述步骤后，重新编译并运行程序。

PA2.2 该过程与 PA2.1 相似，但涉及更多指令的实现，随之而来的是更高的出错可能性。因此，优先完成 diff-test 是明智的选择。通过将自实现的 nemu 与 qemu 在执行指令过程中进行对比，即每执行一步就比较两者的 32 个寄存器值是否相同，不一致时报错，这样可以根据出错时的 PC 值在反汇编代码中定位错误行，进而分析解决问题。

完成 diff-test 后，开始实现指令。为便于测试和错误定位，可按文件大小从小到大排序进行编译运行，逐一实现未完成的指令。以 sum.c 为例，其反汇编代码显示需实现 beq、bne、add 和 sltiu 指令。注意到 beq 和 bne 为 B 类型，add 为 R 类型，这些类型在 dummy 中未涉及，因此需编写对应的译码辅助函数。在完成译码函数后，进入执行辅助函数的编写阶段。由于 beq 和 bne 涉及

跳转，应在 control.c 中编写其执行函数；而 add 和 sltiu 作为计算指令，在 compute.c 中编写。紧接着，根据指令行为编写执行辅助函数。

如果新函数尚未声明，需在 decode.h 中声明译码辅助函数，在 all-instr.h 中声明执行辅助函数。在 exec.c 的 opcode_table 中指定指令对应的译码和执行函数。完成后，使用 make 命令运行测试文件，若出错则检查译码和执行函数；若通过，则继续实现下一指令，如此反复直至所有测试文件通过。

在测试文件 hello-str 和 string 中，需要使用如 sprintf、strcmp、strcat、strcpy、memset 等库函数。这些函数应在 nexus-am/libs/klib/src/stdio.c 和 string.c 中实现。string.c 主要涉及常用字符串处理函数，如 strcmp、strlen、strcat 等。stdio.c 中需要实现的是 sprintf、printf 等，它们都是通过 vsprintf 函数间接实现的。vsprintf 需要处理输入字符串 fmt 和可变参数列表 va_list，当在 fmt 中遇到%d、%x、%s 等子串时，调用 va_arg 函数取出 va_list 中相应类型的参数，再转换为字符串输出。sprintf 和 printf 函数通过调用 vsprintf 来实现各自功能。

至此，完成了 PA2.2 的所有内容。使用一键回归测试可验证所有程序的正确性。

PA2.3 需要实现串口、时钟、键盘和 VGA 四个输入输出设备的程序。串口功能已在 trm.c 中实现，并且 printf 函数也已在 PA2.2 中完成，因此不需要重复编写。

对于时钟功能，在 nemu-timer.c 中进行完善。初始化时，利用 RTC_ADDR 地址获取启动时间。在运行时钟功能时，也通过 RTC_ADDR 地址获取当前时间，设置 uptime->hi 为 0，并将 uptime->lo 设置为当前时间减去启动时间，以实现时钟功能。

键盘功能的实现位于 nemu-input.c。通过 KBD_ADDR 地址读取键盘按键信

息，存放到 kbd->keycode 中。将按键信息与 KEYDOWN_MASK 进行位与操作，可以判断按键是被按下（值为 1）还是松开（值为 0）。

最后，实现 VGA 设备的功能，需要在 nemu-video.c 中完善。VGA 设备的实现包括将 pixels 数组中的像素信息写入 VGA 对应的内存地址空间，以便正确显示图像。

2.3 PA3

2.3.1 设计

PA3.1 为实现自陷(trap)机制，首先需实现相关的自陷指令。包括 csrrs、csrrw、ecall 和 sret。其中，ecall 指令触发自陷，csrrs 和 csrrw 用于修改控制状态寄存器，sret 用于从自陷状态返回。

进一步，在 intr.c 文件中编写 raise_intr 函数，模拟自陷过程。该函数执行以下步骤：保存当前 PC 值到 sepc 寄存器，设置异常号到 scause 寄存器，从 stvec 寄存器取出异常入口地址并跳转。

自陷触发后，需要保存当前上下文。依据 trap.S 中的压栈顺序，重构 _Context 结构体。接着，实现正确的事件分发机制。在 __am_irq_handle 函数中，利用异常号识别自陷异常。在 do_event 函数中，识别出_EVENT_YIELD 自陷事件。最后，通过 sret 指令返回并恢复上下文，完成自陷操作的实现。

PA3.2 为支持 TRM 程序的运行，需实现用户程序的加载及系统调用。首先，由于文件系统尚未实现，需在 loader 函数中直接使用 ramdisk_read 函数读取可执行文件中的代码和数据，并将它们放置到内存的适当位置。然后，跳转到程序的入口点以开始执行。

接着，实现系统调用。这一过程类似于 3.1 节中识别自陷事件的机制。需要让 nemu 能够识别系统调用事件。在 do_event 函数中，添加对 do_syscall 函数的调用，处理系统调用。按照 nanos.c 中的 ARGS_ARRAY，实现 riscv32-nemu 中相应的 GPR 宏。

进一步，添加并实现 SYS_yield、SYS_read、SYS_write 和 SYS_brk 等系统调用，以支持程序的基本输入输出及内存管理。特别地，实现 _sbrk 函数以管理

堆区，从而为用户程序提供动态内存分配的能力。

PA3.3 为实现文件系统和设备支持，进而运行《仙剑奇侠传》，需按照以下步骤进行：

实现文件系统基础函数 fs_open、fs_read、fs_close。随着 ramdisk 中文件数量的增加，不再适宜在 loader 函数中直接使用 ramdisk_read。因此，完成这些文件系统函数后，需替换 loader 函数中的 ramdisk_read，修改逻辑以允许通过文件名指定加载的程序。

接着，实现 fs_write 和 fs_lseek 函数，以支持文件的写入和位置定位操作。

补充相关的系统调用以支持文件操作。

实现虚拟文件系统(VFS)，将 IOE 设备抽象为文件。这涉及在 VFS 中添加对多种特殊文件类型的支持。

实现 serial_write 函数完成串口写入操作，以及 events_read 函数支持读操作。

完成 init_fs、fb_write、fbsync_write、init_device、dispinfo_read 函数，以实现对 VGA 设备的支持。

重要的是，由于 Finfo 结构中新增了读和写的函数指针，需要相应地更新 fs_write 和 fs_read 函数的逻辑。

完成上述所有步骤后，如果没有错误，就应能够顺利运行《仙剑奇侠传》。

3　实验结果与结果分析

3.1 PA1

3.1.1 实验结果

进入简易调试器后，输入 help 显示各个指令对应的信息，如图 3.1 所示。

```
make[1]: 离开目录“/media/sf_PA/ics2019/nemu/tools/qemu-diff”
./build/riscv32-nemu -l ./build/nemu-log.txt -d /media/sf_PA/ics2019/nemu/tools/
qemu-diff/build/riscv32-qemu-so
[src/monitor/monitor.c,36,load_img] No image is given. Use the default build-in
image.
[src/memory/memory.c,16,register_pmem] Add 'pmem' at [0x80000000, 0x87ffffff]
[src/monitor/monitor.c,20,welcome] Debug: ON
[src/monitor/monitor.c,21,welcome] If debug mode is on, A log file will be gener
ated to record every instruction NEMU executes. This may lead to a large log fil
e. If it is not necessary, you can turn it off in include/common.h.
[src/monitor/monitor.c,28,welcome] Build time: 15:48:39, Jan 14 2021
Welcome to riscv32-NEMU!
For help, type "help"
(nemu) help
help - Display informations about all supported commands
c - Continue the execution of the program
q - Exit NEMU
si - Single Step Execute
info - Print details of register || watchpoint
x - Scan memory
p - Expression Evaluation
w - Set a New Watchpoint
d - Delete Watchpoint
(nemu)
```

图3. 1

设置四个监视点后，使用 info 打印监视点信息，如图 3.2 所示。

```
help - Display informations about all supported commands
c - Continue the execution of the program
q - Exit NEMU
si - Single Step Execute
info - Print details of register || watchpoint
x - Scan memory
p - Expression Evaluation
w - Set a New Watchpoint
d - Delete Watchpoint
(nemu) w $t0
Set Watchpoint Succeed
(nemu) w $t1
Set Watchpoint Succeed
(nemu) w $t2
Set Watchpoint Succeed
(nemu) w $t3
Set Watchpoint Succeed
(nemu) info w
NO          EXPR                VALUE
3           $t3                 0
2           $t2                 0
1           $t1                 0
0           $t0                 0
(nemu)
```

图3. 2

删除 3 号监视点后打印监视点信息，如图 3.3 所示。

```
(nemu) d 3
Delete No.3 Watchpoint~
(nemu) info w
NO          EXPR                VALUE
2           $t2                 0
1           $t1                 0
0           $t0                 0
(nemu)
```

图 3.3

使用命令 si 执行两步，由于寄存器 t0 的值发生改变，程序暂停，使用 info 命令显示监视点的值，发现 t0 寄存器的值变为一个负数，实际上是十六进制 0x80000000，继续使用 si 单步执行，程序最终会 HIT GOOD TRAP，运行结果如图 3.4 所示。

```
(nemu) d 3
Delete No.3 Watchpoint~
(nemu) info w
NO          EXPR                VALUE
2           $t2                 0
1           $t1                 0
0           $t0                 0
(nemu) si 2
80100000:   b7 02 00 80                             lui  0x80000,t0
watchpoint:Status Changed
(nemu) info w
NO          EXPR                VALUE
2           $t2                 0
1           $t1                 0
0           $t0                 -2147483648
(nemu) si 2
80100004:   23 a0 02 00                             sw  0(t0),$0
80100008:   03 a5 02 00                             lw  0(t0),a0
(nemu) si 2
8010000c:   6b 00 00 00                             nemu trap
nemu: HIT GOOD TRAP at pc = 0x8010000c

[src/monitor/cpu-exec.c,32,monitor_statistic] total guest instructions = 4
```

图 3.4

使用 p 命令进行表达式求值，对于正确的表达式会求出其值，对于错误的表达式会给出相应的提示，如图 3.5 所示。

```
(nemu) p (1+2)*(4/3)
3
(nemu) p --1
1
(nemu) p (3/3)+)(123*4
wrong expression
```

图 3.5

3.1.2 问题回答

必答题

你需要在实验报告中回答下列问题:

- 送分题 我选择的ISA是 _____ .
- 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?

  事实上, 这些数字也许还是有点乐观, 例如就算使用GDB来直接调试客户程序, 这些数字假设你能通过10分钟的时间排除一个bug. 如果实际上你需要在调试过程中获取并分析更多的信息, 简易调试器这一基础设施能带来的好处就更大.
- 查阅手册 理解了科学查阅手册的方法之后, 请你尝试在你选择的ISA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:
  - x86
    - 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?
  - mips32
    - mips32有哪几种指令格式?
    - CP0寄存器是什么?
    - 若除法指令的除数为0, 结果会怎样?
  - riscv32
    - riscv32有哪几种指令格式?
    - LUI指令的行为是什么?
    - mstatus寄存器的结构是怎么样的?
- 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: 目前 `pa1` 分支中记录的正好是做PA1之前的状态, 思考一下应该如何回到"过去"?) 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` ?

ISA：riscv32。

调试需要花费的时间是 500*90%*30*20=270000s=4500min

简易调试器可以节约的时间为

500*90%*20*20=180000s=3000min riscv32 有 R、I、S、

B、U、J 6 种指令格式。

LUI 指令是将 20 位常量加载到寄存器的高 20 位。

mstatus 寄存器的结构如图 2.6 所示。

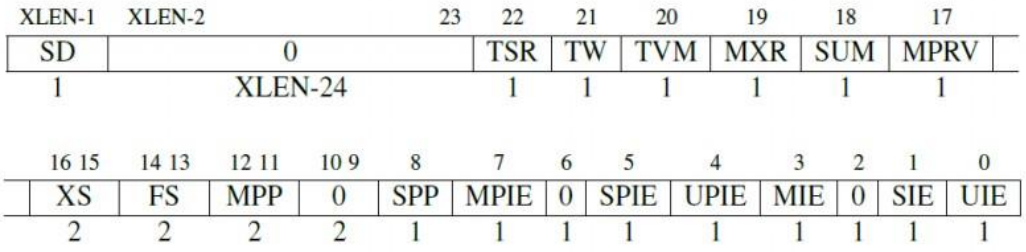


图 2.6 mstatus 寄存器结构

使用命令 find . -name "*[.h|.c]" |xargs cat|wc –l 得出来的行数是 5877。

使用命令 find . -name "*[.h|.c]" |xargs cat|grep -v ^$|wc –l 过滤空行后，得出的行数是 4822。

-Wall 的作用是打开 gcc 所有警告。

-Werror 的作用是要求 gcc 将所有警告当成错误处理。

3.2 PA2

3.2.1 实验结果

在 nemu 目录下，使用一键回归测试，运行结果如图 3.7 所示。

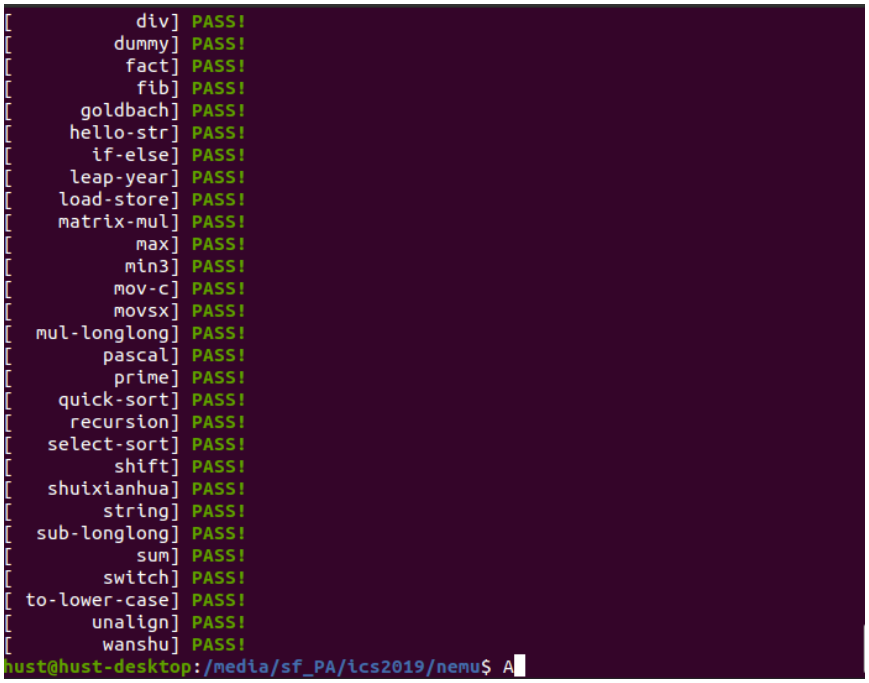


图 3.7

microbench 测试的运行结果如图 3.8 所示。总分为 609 分。

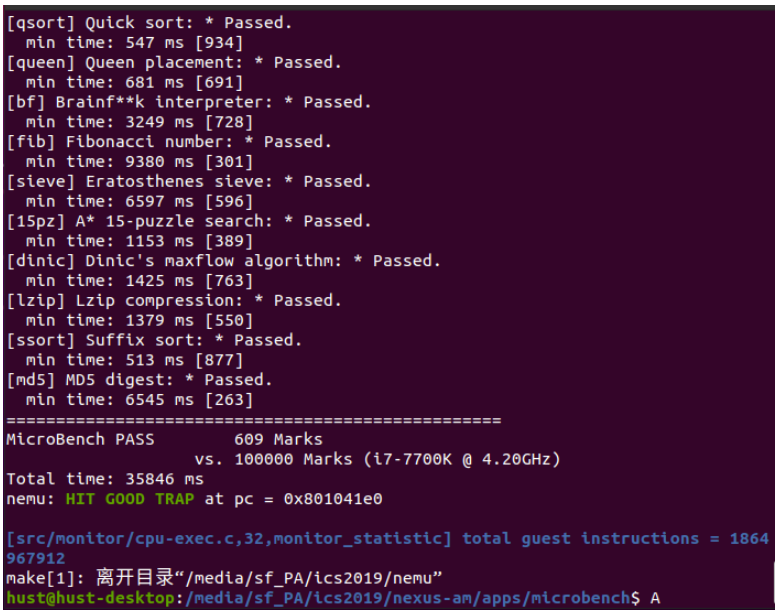


图 3.8

slider 测试的运行结果如图 3.9 所示。

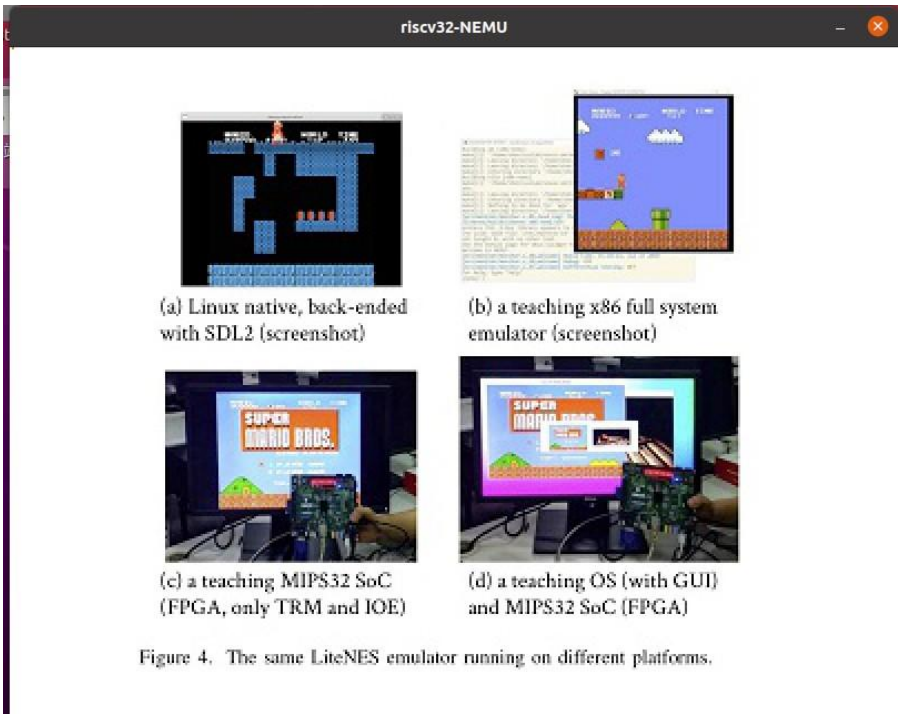


图 3.9

打字小游戏 typing 的测试结果如图 3.10 所示。

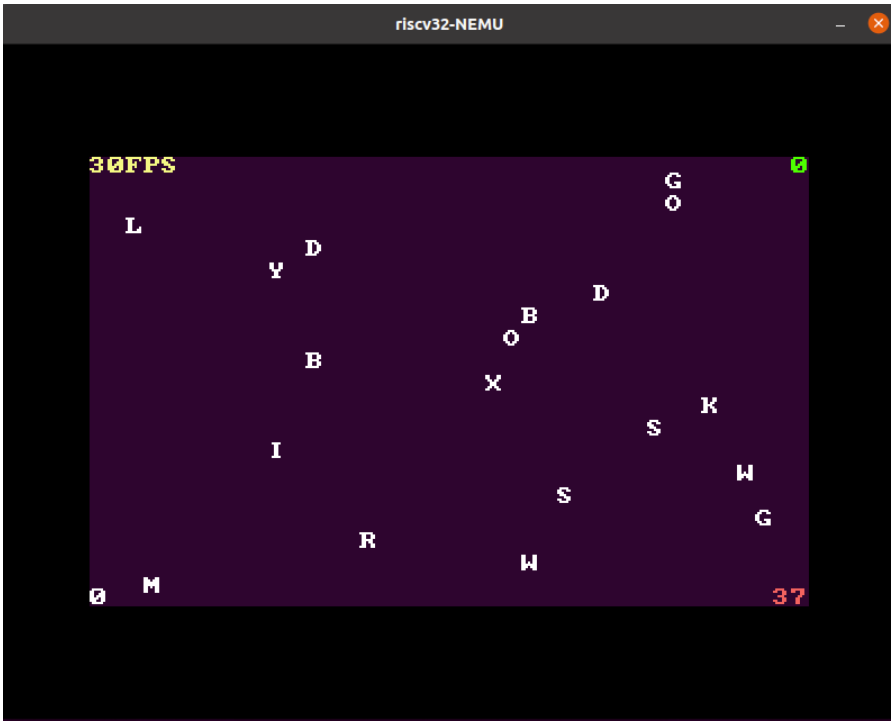


图 3.10

Litenes 测试结果如图 3.11 所示。

图 3.11

3.2.2 问题回答

必答题

你需要在实验报告中用自己的语言, 尽可能详细地回答下列问题.

- RTFSC 请整理一条指令在NEMU中的执行过程. (我们其实已经在PA2.1阶段提到过这道题了)
- 编译与链接 在 `nemu/include/rtl/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?
- 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?
  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.
  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? (回答完本题后可以删除添加的代码.)
- 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/$ISA-nemu` . (这个问题包括两个方面: `Makefile` 的工作方式和编译链接的过程.) 关于 `Makefile` 工作方式的提示:
  - `Makefile` 中使用了变量, 包含文件等特性
  - `Makefile` 运用并重写了一些implicit rules
  - 在 `man make` 中搜索 `-n` 选项, 也许会对你有帮助
  - RTFM

（1）一条指令在 nemu 中执行的过程：

指令执行过程首先从当前的 PC 值开始，使用 instr_fetch 函数获取指令。提取该指令的 opcode，并在 opcode_table 中查找相应的译码和执行辅助函数。使用译码辅助函数对指令进行解码，并将解码结果保存在 decinfo 结构中。然后，执行辅助函数利用 rtl 指令对解码信息执行相应的操作，如计算、读取、保存等。最后，调用 update_pc 函数更新 PC 值。

（2）关于 rtl_li 函数：单独移除 static 或 inline 关键字并重新编译不会引起错误。但同时移除这两个关键字时，会因在另一个文件中存在 rtl_li 的定义而导致重复定义错误。当函数被声明为 static 时，它的作用域被限制在其定义的文件中，避免了重复定义的问题。inline 关键字使得函数在预编译阶段被展开，也不会导致重复定义的错误。若同时移除这两个关键字，就会出现重复定义的错误。

（3）关于 dummy 实体：添加特定代码后，使用 grep 命令检查，发现共有 81 个 dummy 实体。继续添加代码后，重新编译并运行，再次使用 grep 命令检测，现在共有 82 个 dummy 实体。但在代码修改后，重新编译时报错，因为两个实体都被初始化，产生了两个强符号，导致编译错误。

（4）关于 make 过程：执行 make 命令时，它会以 Makefile 文件中的第一个目标文件作为最终目标。如果该目标文件不存在，或其依赖的 .o 文件的修改时间晚于目标文件，make 会重新编译。如果目标文件依赖的 .o 文件也不存在，make 会根据这个 .o 文件的规则来生成它，然后继续生成上层的 .o 文件。如果在这一过程中的任何一步出现错误，make 会立即报错并停止执行。

3.3 PA3

3.3.1 实验结果

运行 hello 测试的运行结果如图 3.12 所示。

```
Hello World from Navy-apps for the 20840th time!
Hello World from Navy-apps for the 20841th time!
Hello World from Navy-apps for the 20842th time!
Hello World from Navy-apps for the 20843th time!
Hello World from Navy-apps for the 20844th time!
Hello World from Navy-apps for the 20845th time!
Hello World from Navy-apps for the 20846th time!
Hello World from Navy-apps for the 20847th time!
Hello World from Navy-apps for the 20848th time!
Hello World from Navy-apps for the 20849th time!
Hello World from Navy-apps for the 20850th time!
Hello World from Navy-apps for the 20851th time!
Hello World from Navy-apps for the 20852th time!
Hello World from Navy-apps for the 20853th time!
Hello World from Navy-apps for the 20854th time!
Hello World from Navy-apps for the 20855th time!
Hello World from Navy-apps for the 20856th time!
Hello World from Navy-apps for the 20857th time!
Hello World from Navy-apps for the 20858th time!
Hello World from Navy-apps for the 20859th time!
Hello World from Navy-apps for the 20860th time!
Hello World from Navy-apps for the 20861th time!
Hello World from Navy-apps for the 20862th time!
Hello World from Navy-apps for the 20863th time!
Hello World from Navy-apps for the 20864th time!
Hello World from Navy-apps for the 20865th time!
Hello World from Navy-apps for the 20866th time!
Hello World from Navy-apps for the 20867th time!
Hello World from Navy-apps for the 20868th time!
```

图 3.12

运行 text 测试的运行结果如图 3.13 所示。

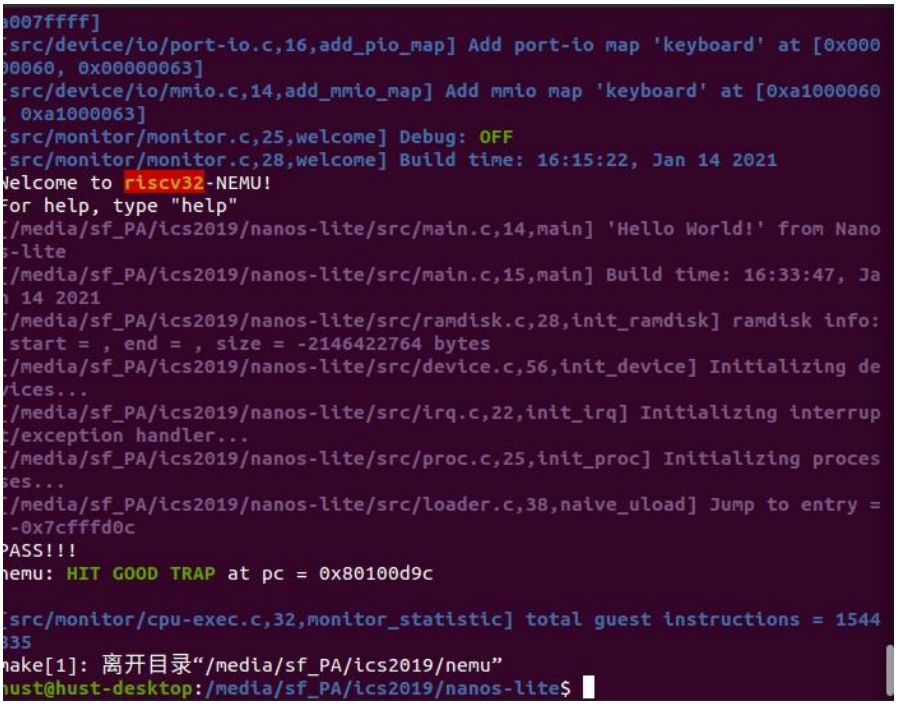


图 3.13

运行 events 测试的结果如图 3.14 所示。

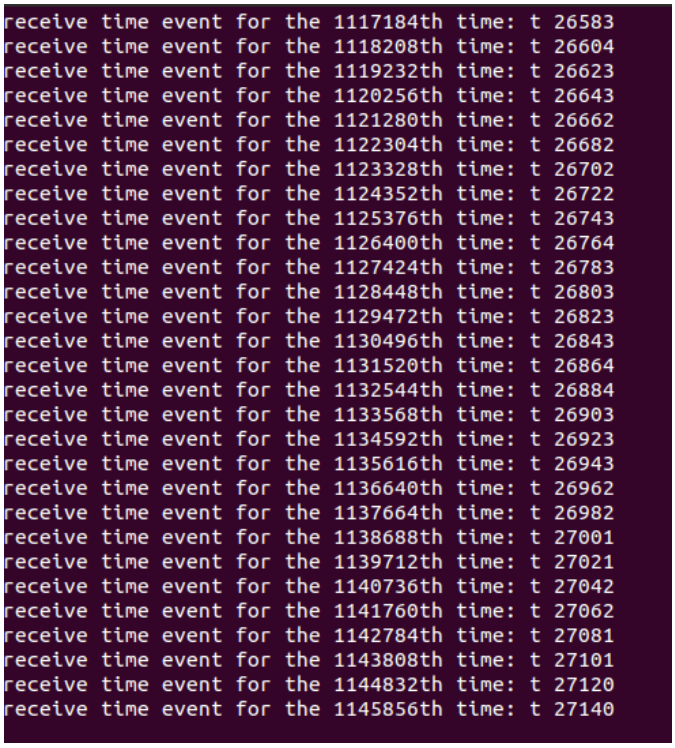


图 3.14

运行 bmptest 的运行结果如图 3.15 所示。

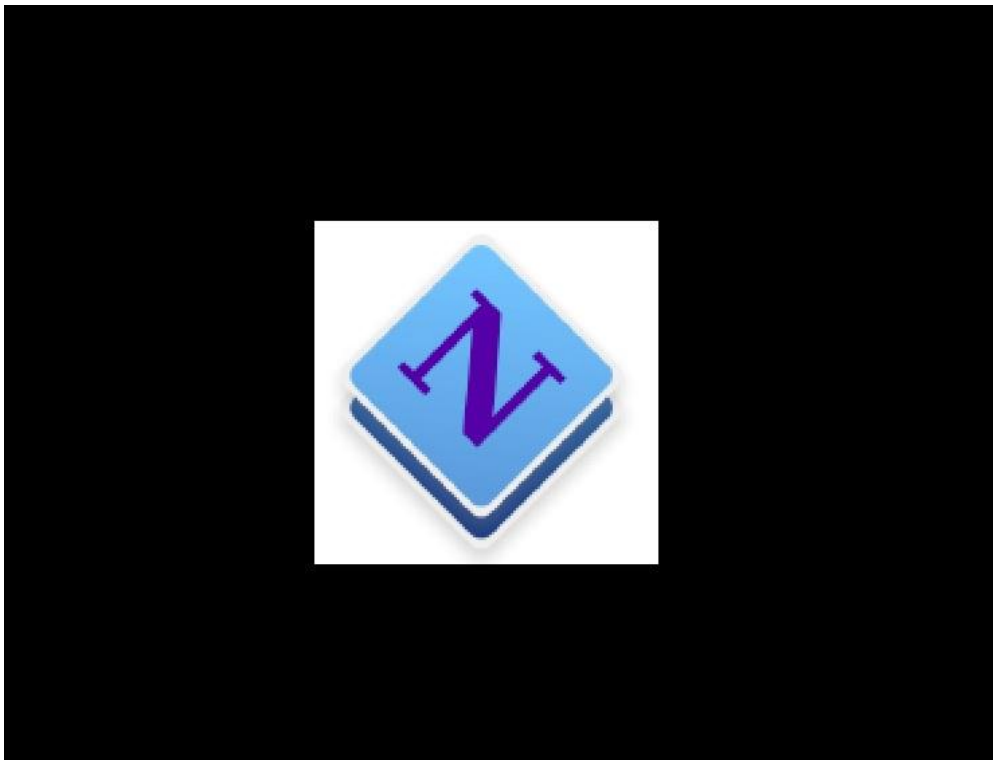


图3.15

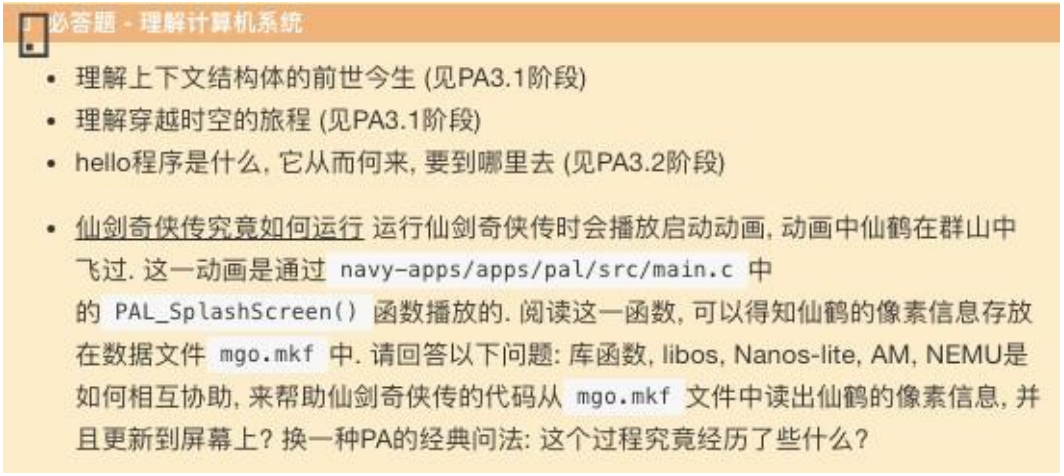
仙剑奇侠传的运行结果如图 3.16 所示。

图 3.16-1


李大娘：
李逍遥！你皮痒啊？
敢说老娘是什么鬼婆！
金钱 0
状态
仙术
物品
系统
储存进度
读取进度
音乐
音效
战斗速度
结束游戏
进度一 0
进度二 0
进度三 0
进度四 0
进度五 0

图 3.16-2

3.3.2 问题回答


必答题 - 理解计算机系统

- 理解上下文结构体的前世今生 (见PA3.1阶段)
- 理解穿越时空的旅程 (见PA3.1阶段)
- hello程序是什么, 它从而何来, 要到哪里去 (见PA3.2阶段)
- 仙剑奇侠传究竟如何运行 运行仙剑奇侠传时会播放启动动画, 动画中仙鹤在群山中飞过. 这一动画是通过 `navy-apps/apps/pal/src/main.c` 中的 `PAL_SplashScreen()` 函数播放的. 阅读这一函数, 可以得知仙鹤的像素信息存放在数据文件 `mgo.mkf` 中. 请回答以下问题: 库函数, libos, Nanos-lite, AM, NEMU是如何相互协助, 来帮助仙剑奇侠传的代码从 `mgo.mkf` 文件中读出仙鹤的像素信息, 并且更新到屏幕上? 换一种PA的经典问法: 这个过程究竟经历了些什么?

（1）在执行自陷操作时，c 指向的上下文结构体 _Context 的赋值是由 trap.S 中的汇编程序完成的。riscv-nemu.h 文件中定义了该结构体的相关结构，而 trap.S 则负责对该结构体进行具体的赋值操作。讲义详细阐述了这一过程的流程，并且实现了相关指令，确保自陷操作能够顺利进行。

（2）Nanos-lite 触发中断时，通过操纵 AM 来发起自陷指令的汇编代码，随即保存当前的上下文。然后，它转移到 CPU 自陷指令的内存区域执行相关操作。执行完毕后，Nanos-lite 恢复先前的上下文，并返回到原先的运行环境。

（3）经编译成 ELF 文件的 hello.c 位于 ramdisk 中。该文件通过 naive_uload 函数被读取到内存并放置在适当位置，之后由操作系统调用执行。程序在运行过程中使用 SYS_write 系统调用来输出字符。执行完毕后，操作系统负责回收该程序占用的内存空间。

（4）操作系统利用库函数读取画面的像素信息，随后通过 VGA 设备进行输出。VGA 被视为一个设备文件，其内部通过 fs_write 函数逐步调用 draw_rect 函数。draw_rect 函数负责将像素信息写入 VGA 对应的内存地址。最终，画面通过调用 update_screen 函数被渲染到屏幕上。

参考文献

[1] (N.d.). https://course.cunok.cn:52443/pa/doc2019/

[2] (N.d.). https://course.cunok.cn:52443/projects/pa/wiki