***2019***



**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： | CS1601 |
| 学 号： | U201614531 |
| 姓 名： | 刘本嵩 |
| 电 话： | +1 (669) 266-7699 |
| 邮 件： | root@recolic.org |
| 完成日期： | 2020-01-02 |

目录

[1 课程设计概述 1](#_Toc172092055)

[1.1 课设目的 1](#_Toc1141178468)

[1.2 课设任务 1](#_Toc1420241910)

[1.3 实验环境 2](#_Toc1504584016)

[2 实验过程 3](#_Toc1009591457)

[2.1 PA0 3](#_Toc1488601112)

[2.2 PA1 3](#_Toc1906104787)

[2.2.1 总体设计 3](#_Toc352500315)

[2.2.2 详细设计 4](#_Toc1979632440)

[2.2.3 运行结果 6](#_Toc1952126041)

[2.2.4 问题解答 7](#_Toc1631598016)

[2.3 PA2 10](#_Toc682599573)

[2.3.1 总体设计 10](#_Toc1256933234)

[2.3.2 详细设计 11](#_Toc352620285)

[2.3.3 运行结果 13](#_Toc928299354)

[2.3.4 问题解答 19](#_Toc867603571)

[3 设计总结与心得 22](#_Toc1024813162)

[3.1 课设总结 22](#_Toc1462590790)

[3.2 课设心得 22](#_Toc1404166848)

[3.3 框架代码改进建议 23](#_Toc442491966)

[参考文献 24](#_Toc2015066504)

# 课程设计概述

## 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. 南京大学计算机科学与技术系计算机系统基础课程的小型项(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. 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:

## 课设任务

1. 配置开发环境。
2. 实现debugger的基础功能，例如x p w d c n等指令。实现表达式的evaluation，实现watchpoint等功能。
3. 依次实现i386手册规定的大多数指令，实现一个精简版的libc，实现设备管理相关功能，实现时间和GUI绘图，以便运行打字小游戏、性能测试、超级马里奥游戏等x86的app。

## 实验环境

-`

.o+` recolic@RECOLICPC

`ooo/ OS: Arch Linux

`+oooo: Kernel: x86\_64 Linux 5.4.6-arch3-1

`+oooooo: Uptime: 16h 2m

-+oooooo+: Packages: 2053

`/:-:++oooo+: Shell: fish 3.0.2

`/++++/+++++++: Resolution: 4480x1440

`/++++++++++++++: DE: GNOME 3.34.2

`/+++ooooooooooooo/` WM: Mutter

./ooosssso++osssssso+` WM Theme:

.oossssso-````/ossssss+` GTK Theme: Gnome-OSX-II-2-6 [GTK2/3]

-osssssso. :ssssssso. Icon Theme: hicolor

:osssssss/ osssso+++. Font: Cantarell 11

/ossssssss/ +ssssooo/- Disk: 966G / 1.8T (56%)

`/ossssso+/:- -:/+osssso+- CPU: Intel Core i5-4200H @ 4x 3.4GHz

`+sso+:-` `.-/+oso: GPU: GeForce GTX 950M

`++:. `-/+/ RAM: 3955MiB / 15465MiB

.` `/

# 实验过程

## PA0

我有一台在2016年安装好的Arch Linux计算机，其中早已配置好所有必要的开发环境，因此PA0已经完成。

## PA1

### 总体设计

我做了以下几件事：

1. 修改Makefile和提交脚本，允许用户透明化的设置自己使用的反向代理host。包括git(ssh)和https支持。
2. 改造框架代码和Makefile，将nemu的C代码全部转换为C++。
3. 修改Makefile，将他在每次Make后自动生成一个无意义commit的行为改为，可以使用make git-gen M=”Some thing.”这种命令来手动控制commit生成。修改了脚本的其他无意义的举动。
4. 修改其他配套script，因为老师提供的所有shell script在关键位置没有任何错误检查。这会导致学生在反代环境的不稳定网络下体验极差。
5. 搭建校园网反向代理服务，提供反向代理服务文档。校园网内的一个闲置Linux路由器(武汉)，proxy.recolic.net(香港)，git.recolic.net(彰化縣)，storage.recolic.net(法国巴黎)，共同支持了此服务的运行。
6. 增加public-template分支，这允许用户不再面对我已经解决过的问题。同时，这允许imzhwk等用户更方便的二次开发出Docker镜像。
7. 改造框架代码，优化common.h中不合理的命名，引入自己实现的C++库(rlib)，修改不合理的gitignore。
8. 实现PA1要求的Debugger功能。

### 详细设计

PA1的详细功能中，首先修改reg.h，增加需要的9个32bit寄存器。

实现info r和x指令，打印寄存器和内存。寄存器可以直接使用rlib::printfln()和C++ iomanip库进行打印。内存则直接访问vaddr\_read即可。

static int **cmd\_info**(char \*\_args) {

if(\_args == NULL)

throw std::**runtime\_error**("Usage: info <what>");

if(**string**(\_args).**strip**() == "r") {

**println**("Registers:");

**printfln**("%eax={}, %ebx={}, %ecx={}, %edx={}", **dumpReg**(cpu.eax), **dumpReg**(cpu.ebx), **dumpReg**(cpu.ecx), **dumpReg**(cpu.edx));

**printfln**("%esp={}, %ebp={}, %esi={}, %edi={}", **dumpReg**(cpu.esp), **dumpReg**(cpu.ebp), **dumpReg**(cpu.esi), **dumpReg**(cpu.edi));

**printfln**("%eip={}, CF/OF/SF/ZF/DF={}/{}/{}/{}/{}", **dumpReg**(cpu.eip), cpu\_eflags::**get**<cpu\_eflags::**CF**>(),

cpu\_eflags::**get**<cpu\_eflags::**OF**>(), cpu\_eflags::**get**<cpu\_eflags::**SF**>(), cpu\_eflags::**get**<cpu\_eflags::**ZF**>(),

cpu\_eflags::**get**<cpu\_eflags::**DF**>());

}

else if(**string**(\_args).**strip**() == "w") {

**println**("Watchpoints:");

**println**(watchpoints);

}

return 0;

}

实现一个表达式求值工具。这很简单，我使用bison配合flex，并稍微修改Makefile，使主Makefile在拉取cc源文件列表前，先运行这个子目录的generate目标。在bison的帮助下，我在几十分钟内完成了它。一次通过。

%%

entry: {\*result = 0;}

| expr {\*result = $1;}

;

expr: T\_INT { $$ = $1; }

| expr '+' expr { $$ = $1 + $3; }

| expr '-' expr { $$ = $1 - $3; }

| expr '\*' expr { $$ = $1 \* $3; }

| expr '/' expr { $$ = $1 / $3; }

| expr T\_EQUAL expr { $$ = ($1 == $3); }

| expr T\_NEQUAL expr { $$ = ($1 != $3); }

| expr T\_LOGICAL\_AND expr { $$ = ($1 && $3); }

| '(' expr ')' { $$ = $2; }

| '-' expr { $$ = 0 - $2; }

| '\*' expr { $$ = ryy\_read\_memory\_value((uint32\_t)$2); }

| T\_REG { $$ = ryy\_read\_cpu\_reg\_value($1); }

;

%%

然后要实现watchpoints。本来以为文档里说的watchpoint池是一个池，没想到只是一个普通的list。可惜rlib里有一个已经实现好的通用对象池。这样直接实现watchpoint类然后用std::list装起来即可。

struct **WP** {

std::string expr;

uint32\_t curr\_value;

int id;

bool **evalulate**() {

auto new\_value = **evaluate\_expr**(expr);

std::**swap**(new\_value, curr\_value);

return new\_value != curr\_value && curr\_value != 0;

}

**WP**(std::**string** e, int id) : **expr**(e), **id**(id) {

**evalulate**(); *// initial expr value.*

}

};

inline std::list<WP> watchpoints;

*// WARNING: Not thread-safe*

inline int max\_watchpoint\_id;

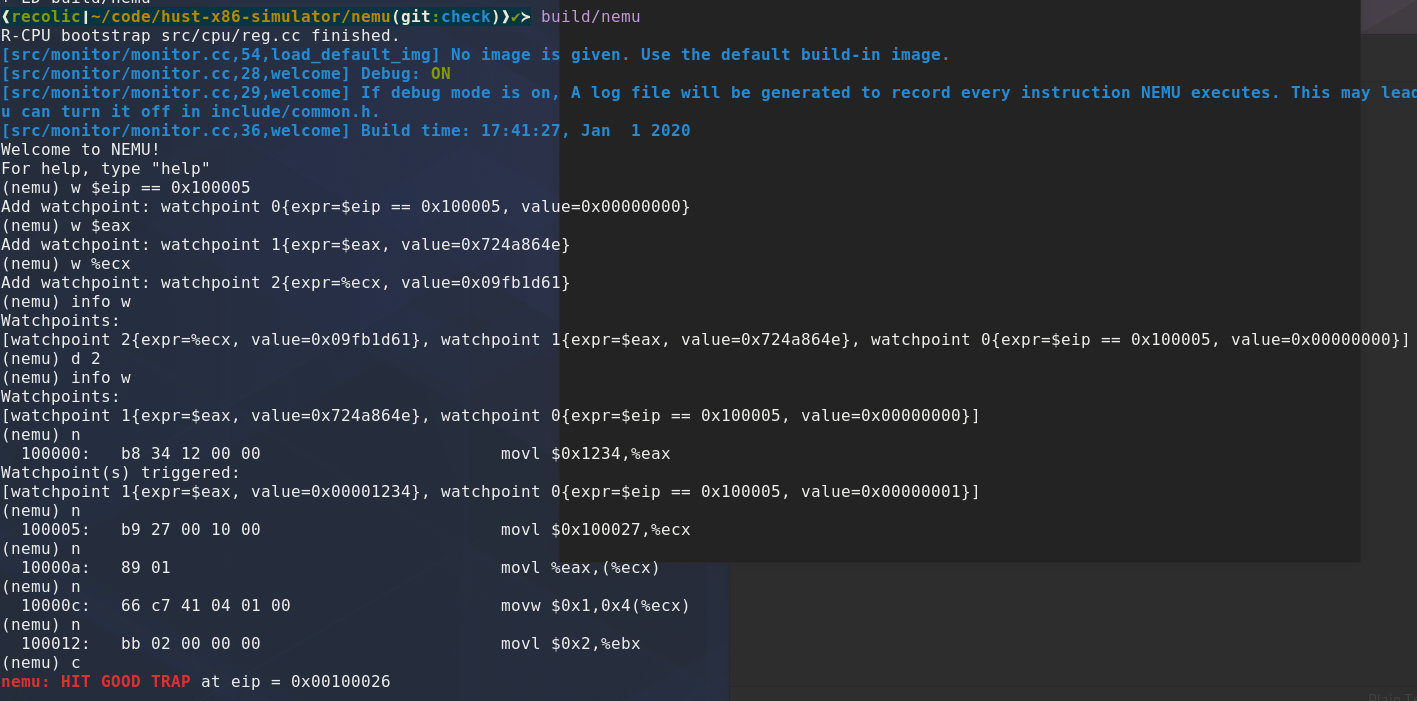
inline std::**ostream** & operator<< (std::**ostream** &os, const **WP** &watchpoint) {

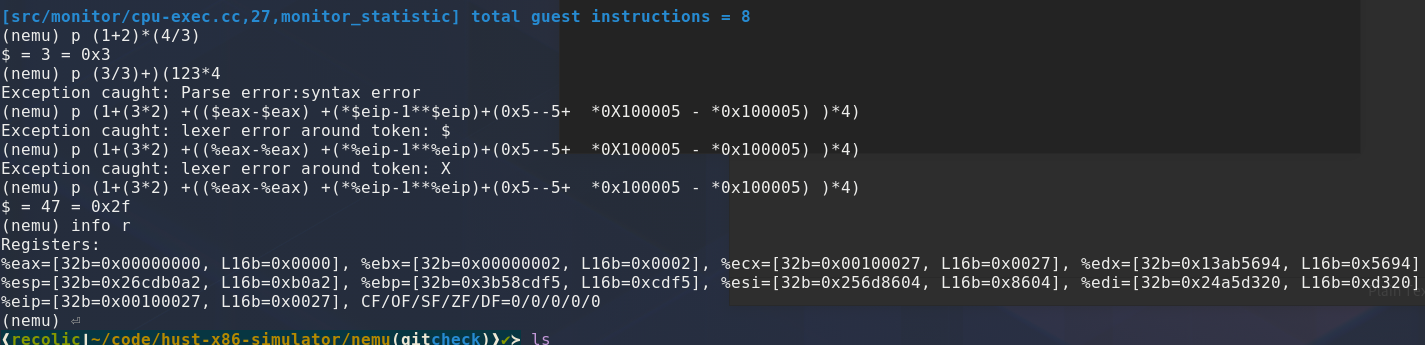
return os << "watchpoint " << watchpoint.id << "{expr=" << watchpoint.expr << ", value=" << **num2hex**(watchpoint.curr\_value) << "}";

}

值得注意的是，这里我有几个与约定不符的地方，例如我用n而不是si表示单步执行，用%eax而不是$eax表示寄存器，用1表示第一个watchpoint而不是0，在下边缘也会触发watchpoint，不支持大写的0X100005。这些修改补丁我都在pa2这个分支(最终检查之前)才增加(同时支持新旧写法)。

### 运行结果





### 问题解答

1. 理解基础设施

由于问题不清晰，我们这里假设：每次”用于调试”的编译中，仅获取并分析一个（而不是0个或多个）信息。

则，你会在调试上花费500\*0.9\*30s = 13500s时间。

简易调试器可以节约500\*0.9\*20s = 9000s时间。

1. 查阅i386手册

Q: EFLAGS寄存器中的CF位是什么意思?

A: Page 419 of 421, Appendix C Status Flag Summary:

0 -- CF -- Carry Flag -- Set on high-order bit carry or borrow; cleared

otherwise.

Q: ModR/M字节是什么?

A: Page 241-242 of 421, Section 17.2.1 ModR/M and SIB Bytes

The ModR/M byte contains three fields of information:

● The mod field, which occupies the two most significant bits of the

byte, combines with the r/m field to form 32 possible values: eight

registers and 24 indexing modes.

● The reg field, which occupies the next three bits following the mod

field, specifies either a register number or three more bits of opcode

information. The meaning of the reg field is determined by the first

(opcode) byte of the instruction.

● The r/m field, which occupies the three least significant bits of the

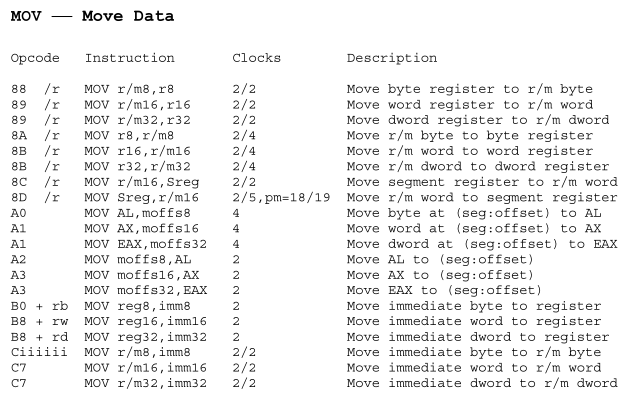
byte, can specify a register as the location of an operand, or can form

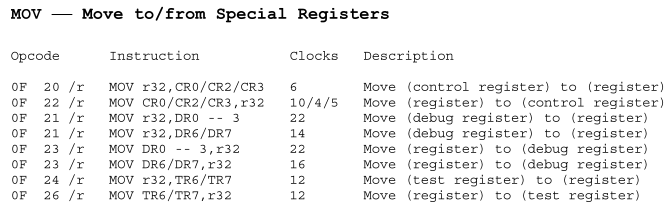
part of the addressing-mode encoding in combination with the field as

described above.

Q: mov指令的具体格式是怎么样的?

A: Page 345-347 of 421





1. shell命令

声明：回答此问题使用的PA1指hash为

99f069f03c4d34824bf1af5bc2eb3fd3a608f5de的commit。未来PA1的HEAD指针的移动不影响此题答案。

1. \*.c的文件的代码数。

由于此工程的nemu部分已经被更换为C++，因此.c文件中**共有0行代码。除去空行后，共有0行代码。**

1. \*.h的文件的代码数。

为了统计.h文件的代码数，先使用命令

find . -name '\*.h' -exec cp '{}' ~/tmp/hust/ ';'

把所有h文件从工程目录中移出，防止.hpp文件和.cc文件的干扰。然后使用sloc工具，结果如下。(**共有950行代码，除去空行后，共有760行代码**)

Language Files Code Comment Blank Total

Total 24 675 90 190 950

C 24 675 90 190 950

1. 真正实用的代码数统计。

直接在nemu目录下使用sloc工具，结果如下。

Language Files Code Comment Blank Total

Total 70 5533 1156 1520 7957

C++ 37 4581 1046 1250 6631

C 25 701 94 200 990

YACC 1 91 2 20 112

Make 4 87 12 35 134

Shell 1 33 1 6 40

Markdown 1 24 0 2 26

Lex 1 16 1 7 24

1. 使用man
2. gcc 9.2.0中-Wall的作用如下：

This enables all the warnings about constructions that some users consider questionable, and that are easy to avoid (or modify to prevent the warning), even in conjunction with macros. This also enables some language-specific warnings

described in C++ Dialect Options and Objective-C and Objective-C++ Dialect Options.

-Wall turns on the following warning flags:

-Waddress -Warray-bounds=1 (only with -O2) -Wbool-compare -Wbool-operation -Wc++11-compat -Wc++14-compat -Wcatch-value (C++ and Objective-C++ only) -Wchar-subscripts -Wcomment -Wduplicate-decl-specifier (C and Objective-C only)

-Wenum-compare (in C/ObjC; this is on by default in C++) -Wformat -Wint-in-bool-context -Wimplicit (C and Objective-C only) -Wimplicit-int (C and Objective-C only) -Wimplicit-function-declaration (C and Objective-C only) -Winit-self

(only for C++) -Wlogical-not-parentheses -Wmain (only for C/ObjC and unless -ffreestanding) -Wmaybe-uninitialized -Wmemset-elt-size -Wmemset-transposed-args -Wmisleading-indentation (only for C/C++) -Wmissing-attributes -Wmissing-braces

(only for C/ObjC) -Wmultistatement-macros -Wnarrowing (only for C++) -Wnonnull -Wnonnull-compare -Wopenmp-simd -Wparentheses -Wpointer-sign -Wreorder -Wrestrict -Wreturn-type -Wsequence-point -Wsign-compare (only in C++)

-Wsizeof-pointer-div -Wsizeof-pointer-memaccess -Wstrict-aliasing -Wstrict-overflow=1 -Wstringop-truncation -Wswitch -Wtautological-compare -Wtrigraphs -Wuninitialized -Wunknown-pragmas -Wunused-function -Wunused-label -Wunused-value

-Wunused-variable -Wvolatile-register-var

Note that some warning flags are not implied by -Wall. Some of them warn about constructions that users generally do not consider questionable, but which occasionally you might wish to check for; others warn about constructions that

are necessary or hard to avoid in some cases, and there is no simple way to modify the code to suppress the warning. Some of them are enabled by -Wextra but many of them must be enabled individually.

1. gcc 9.2.0中-Werror的作用如下：

Make all warnings into errors.

1. 为什么要使用-Wall和-Werror?

-Wall有利于及时发现代码风格的部分不严谨之处，并在当次提交之前得以改正。

我反对-Werror的使用，我在此实验中也没有使用此选项。相比于冗长无趣且浪费纸张的抱怨，我选择直接引用这位开发者的博客。我的观点与他相同。

<https://embeddedartistry.com/blog/2017/05/22/werror-is-not-your-friend/>

## PA2

### 总体设计

我做了以下几件事：

1. 修改runall脚本，允许更好的日志报告机制。
2. 修改Makefile和icc编译校本，允许一键使用icc进行较优化的编译，包括指导icc在编译的过程中尝试运行代码，以便根据benchmark运行结果反过来优化编译过程。仅在编译方法的修改上，这可以减少40%的代码运行时间。
3. 增加gitlab-ci，在每次提交后和Merge Request中，我的Gitlab Runner会自动检查我的代码是否能通过gcc的编译、icc的编译、和runall.sh的自动测试。并直观的显示在gitlab界面上。
4. 改造框架代码，将EHelper、DHelper等不恰当的宏使用，改为更优的namespace方案。增加重载，允许rlib::print家族函数直接打印Operand等结构体。
5. 依次增加实验需要用到的指令，并通过所有的CPU test。
6. 依次实现实验需要用到的设备函数和libc函数，并通过time test，video test，和其他benchmark和超级马里奥。
7. 根据Intel Vtune给出的性能分析结果，改造框架代码，优化热点代码，并采用多线程异步渲染画面。这可以更有效的利用CPU时间，额外减少约30%的代码运行时间。

### 详细设计

值得注意的是，用函数名拼接的方法实现namespace的功能给IDE开发和debug带来了困难，因此我把DHelper和EHelper都装进了C++原生的namespace。

#define **make\_EHelper**(name) void name(vaddr\_t \*eip)

namespace EHelperImpl {

**make\_EHelper**(mov);

**make\_EHelper**(...);

}

依次添加了CF OF SF ZF DF寄存器，实现所有汇编指令。同时实现IO相关函数，在paddr\_read/write函数中加入MMIO相关逻辑。

namespace cpu\_eflags {

enum {CF = 0, PF, AF, ZF, SF, OF, DF, SIZE\_OF\_EFLAGS};

template <size\_t **Which**>

inline bool &**get**() {

static bool actual\_buffer[SIZE\_OF\_EFLAGS];

return actual\_buffer[Which];

}

}

\_\_attribute\_\_((hot)) uint32\_t **paddr\_read**(paddr\_t addr, int len) {

static const uint32\_t niddle[] = {0, 0xff, 0xffff, 0xffffff, 0xffffffff};

#ifndef **DISABLE\_MMIO**

if(const auto mmio\_id = **is\_mmio**(addr); **RLIB\_MACRO\_LIKELY**(-1 == mmio\_id)) {

#endif

return **pmem\_rw**(addr, uint32\_t) & niddle[len];

#ifndef **DISABLE\_MMIO**

}

else {

return **mmio\_read**(addr, len, mmio\_id);

}

#endif

}

修改device\_update，将性能影响较大的device更新操作放入新线程。

void **device\_update**() {

#ifdef **ENABLE\_ASYNC\_RENDER**

#else

if(device\_update\_flag.**exchange**(false)) {

**device\_update\_impl**();

}

#endif

}

[[maybe\_unused]] static void **device\_update\_thread\_daemon**() {

while(true) {

if(device\_update\_flag.**exchange**(false)) {

**device\_update\_impl**();

}

*// At most, 1000FPS*

std::this\_thread::**sleep\_for**(std::chrono::**milliseconds**(1));

}

}

修复SDL多线程问题，将SDL初始化操作也移入更新线程。

void **update\_screen**() {

#ifndef **DISABLE\_MMIO**

#ifdef **ENABLE\_ASYNC\_RENDER**

if(window == nullptr) **init\_vga\_impl**();

#endif

**SDL\_ErrorCheck**(**SDL\_UpdateTexture**(texture, NULL, vmem, SCREEN\_W \* sizeof(vmem[0][0])));

**SDL\_ErrorCheck**(**SDL\_RenderClear**(renderer));

**SDL\_ErrorCheck**(**SDL\_RenderCopy**(renderer, texture, NULL, NULL));

**SDL\_RenderPresent**(renderer);

#endif

}

void **init\_vga**() {

#ifdef **ENABLE\_ASYNC\_RENDER**

*// Because of fucking SDL design, vga\_init should be done in updating thread.*

*// Do nothing in main thread.*

#else

**init\_vga\_impl**();

#endif

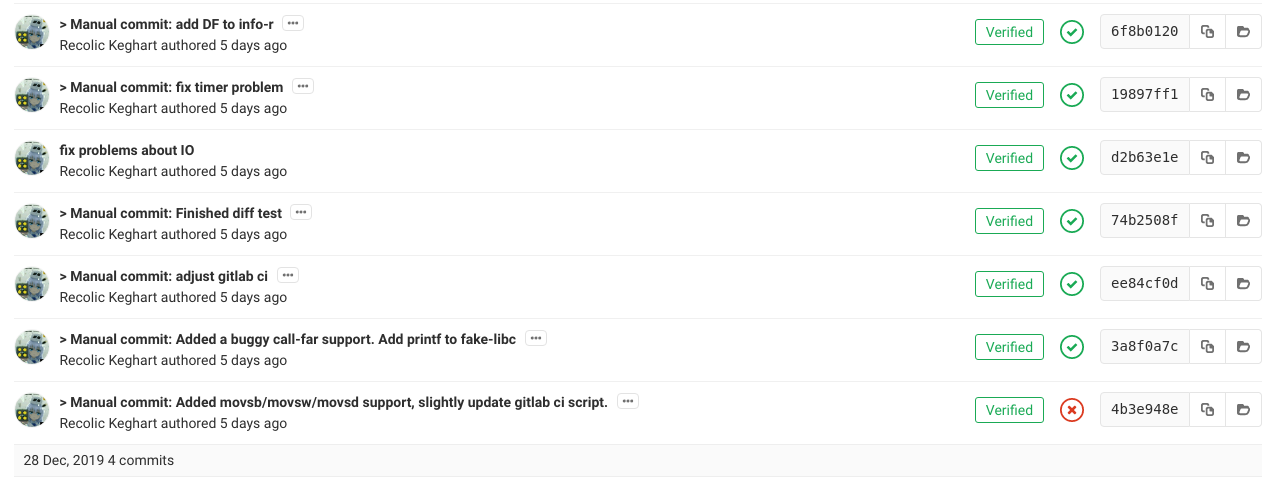
}

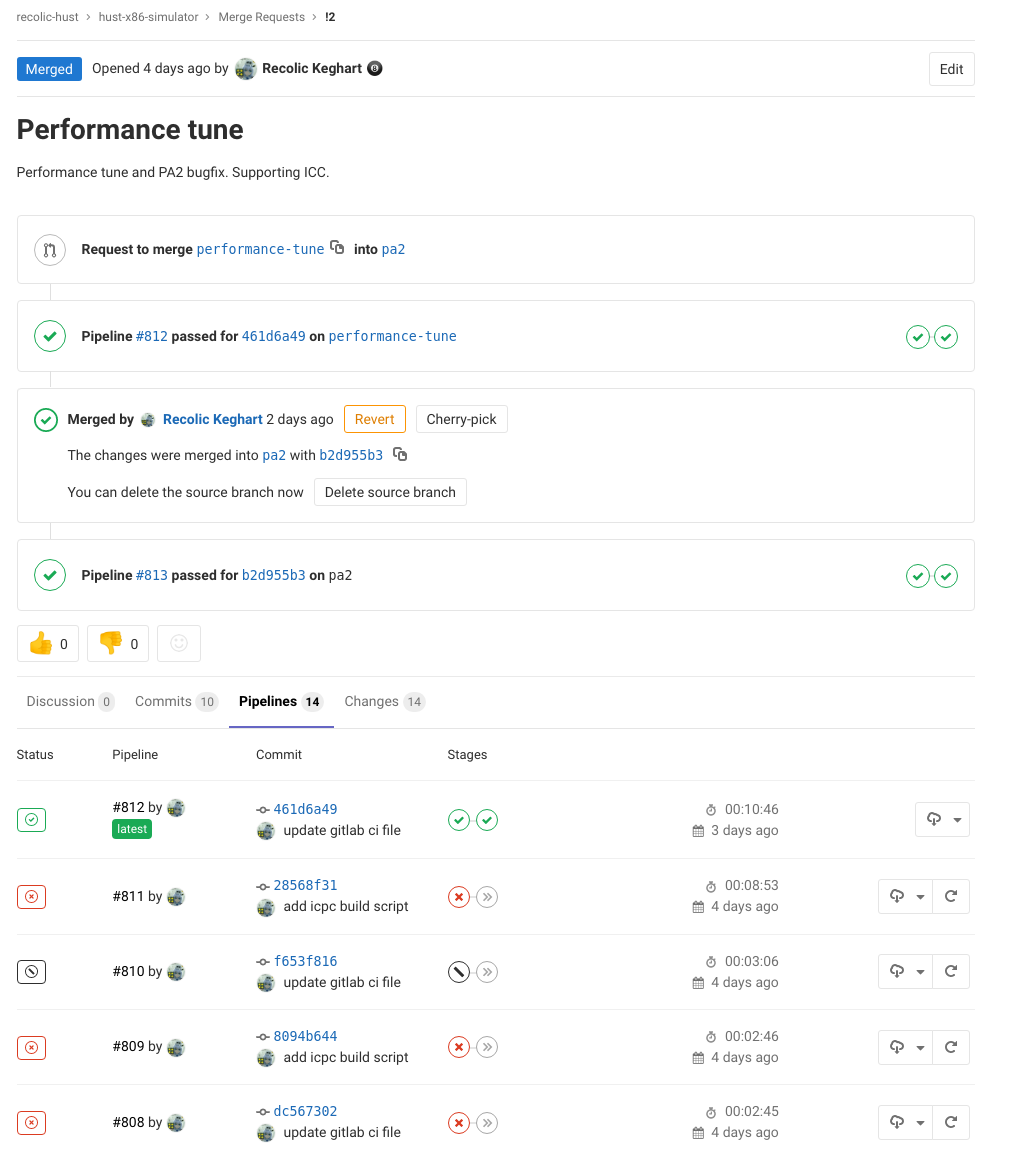
实现了时钟和IO等操作。

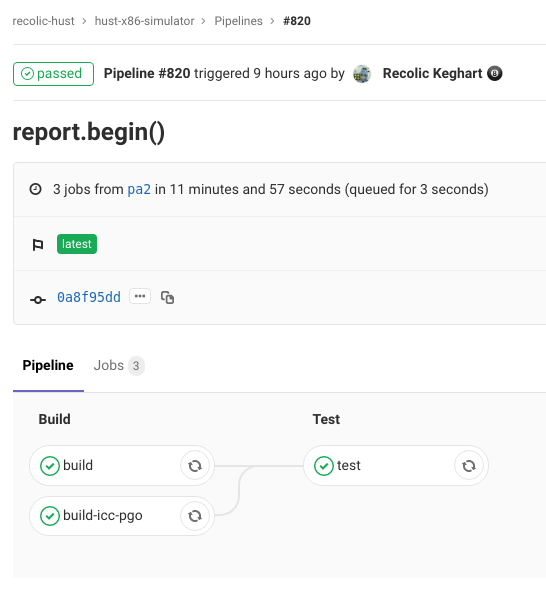
值得注意的是，我的libc的string实现直接移植了glibc，memcpy等操作来自uclibc，printf的实现则来自另一个嵌入式设备的libc实现。这些开源实现在提升性能的同时，也迫使我额外实现了movs等字符串操作汇编指令。

### 运行结果

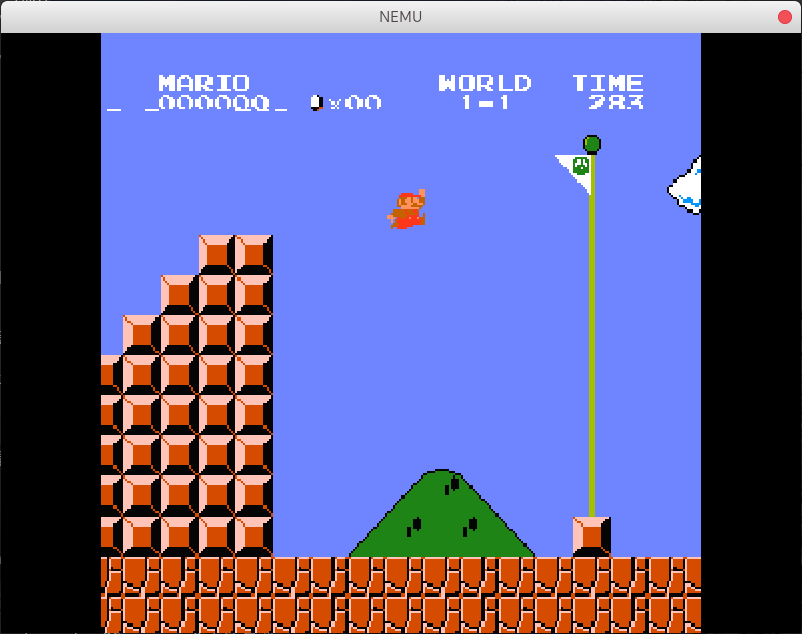
值得注意的是，CI提供的，在另一台全新的机器上的自动测试，对自动发现代码的问题有很大作用，我因此发现了关于icc、开发环境和依赖库的很多问题。

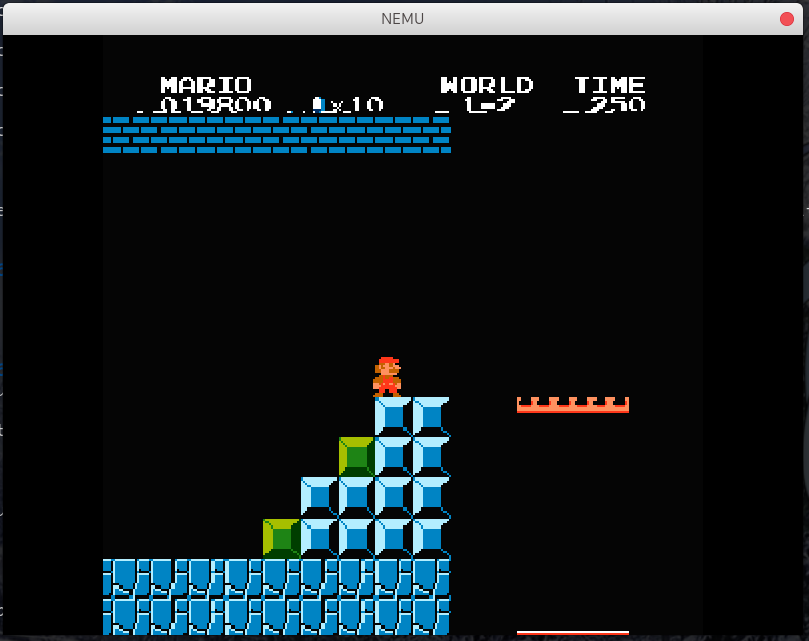






下面是代码的运行结果。图为超级马里奥游戏过程。

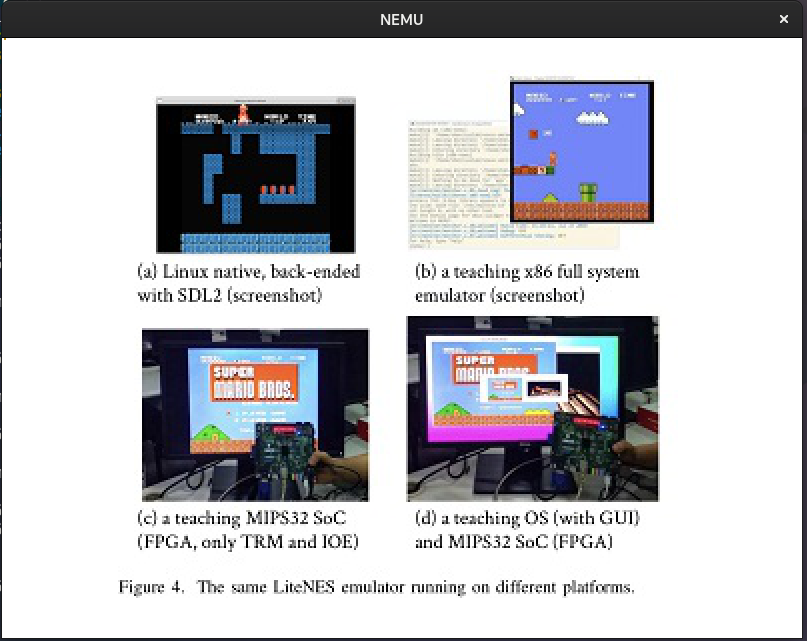


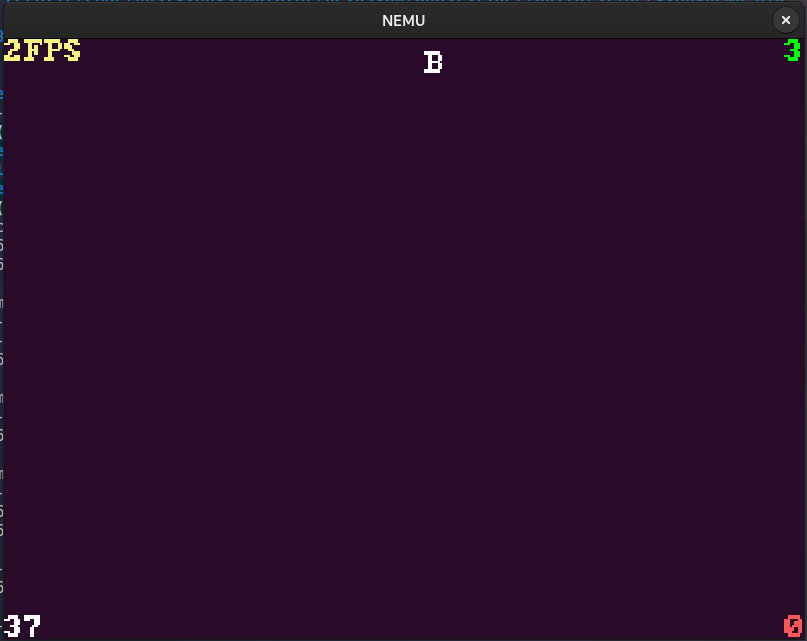


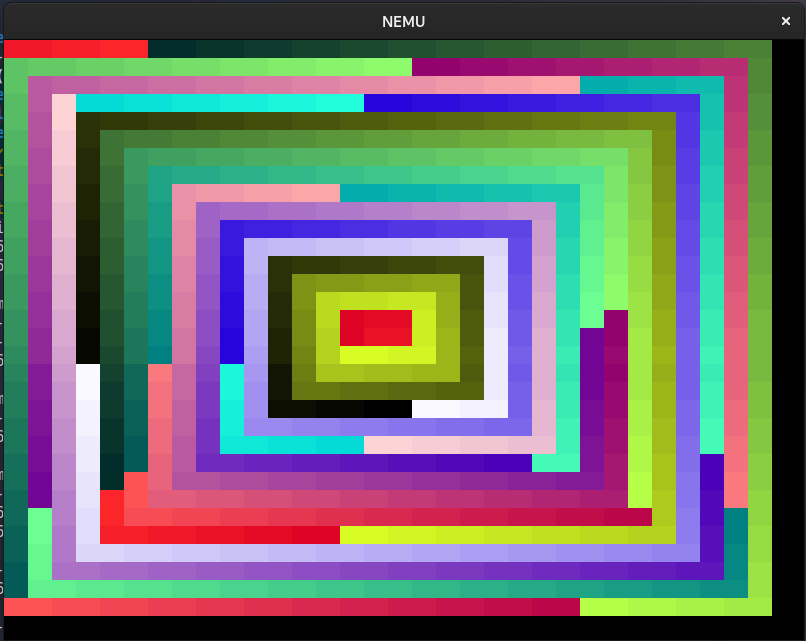
下面是micro benchmark的结果，781分的性能勉强支撑起超级马里奥的游戏体验(稳定约4fps，至少能玩，并且有一定乐趣)。



下面是其他几个test的结果，依次展示。







### 问题解答

1. 编译与链接

去掉static，将不会发生错误。我不了解应当如何解释一个不发生的错误。

去掉inline，编译时不会发生错误，链接时会发生错误。LD在链接不同的object时，会发现同一个符号在不同object中被多次定义，并且我们没有告诉LD在这种情形下应当怎么进行链接，因此报错。inline避免链接器报错的原因是，C++标准就是这么规定的。下面的文字中解释了C++标准这样的规定如何得以实现。

It specifies the scope of the function to be that of a translation unit. So, if an inline function appears in foo.cpp (either because it was written in it, or because it #includes a header in which it was written, in which case the preprocessor basically makes it so). Now you compile foo.cpp, and possibly also some other bar.cpp which also contains an inline function with the same signature (possibly the exact same one; probably due to both #includeing the same header). When the linker links the two object files, it will not be considered a violation of the ODR, as the inline directive made each copy of the file local to its translation unit (the object file created by compiling it, effectively). This is not a suggestion, it is binding.

1. 编译与链接

声明：回答此问题使用的PA2指hash为

0a8f95dd3f89bb7902fa984b01f7be49432417d6的commit。未来PA2的HEAD指针的移动不影响此题答案。

声明：此问题没有指定，新增的代码在头文件中的位置。我在此假定：新增的代码位于header guarder macro的下面一行。即，头文件头部的#ifndef宏的后面。

声明：我的PA2使用g++作为CC和Linker，使用-std=c++17编译选项。原框架代码使用的C语言标准与C++17标准出现不一致的结果请谅解。

1. 29个。

因为按照C++标准，实体数应等同于被链接的包含了common.h的object的个数。然后做一个简单的实验，在common.h添加一行int dummy;，然后看LD报错多少行，就知道有多少个object包含了common.h了。

1. 29个。

用与上一小题同样的方式做实验即可。实验结果的解释：他就是有这么多的object包含了debug.h头文件。

1. 没有出现任何问题。
2. 了解Makefile

声明：我修改了Makefile的工作方式。请参阅

(https://git.recolic.org/recolic-hust/hust-x86-simulator/blob/pa2/nemu/Makefile)

开头的大量变量定义就是代码字面意思，过于冗长略去。值得注意的是，它定义了默认target为app，以及包含了Makefile.git，用于在每次编译时自动生成一个，美观度存在争议的commit。

随后，Makefile会运行SUBDIR的generate这个target，让bison和flex自动生成所需的头文件和.cc源文件。

然后，shell命令统计本目录下所有的.cc源文件，自动生成对应的.o的target。

随后，GNU make执行到app这个default target，并invoke $(BINARY)这个target。此target假装依赖于所有的object，并使得所有object被编译，并在$(BINARY)中进行链接，生成最终的二进制文件，结束。

# 设计总结与心得

## 课设总结

在本次实验中，我基于逐条指令模拟的方式，实现了一个软件x86模拟器和一个简单的libc。模拟器支持大多数i386指令，并且有IO、vga设备和计时器设备可用。模拟器使用自己实现的简易libc，而不是glibc、uclibc-ng等更广泛使用的实现。

它能够运行大多数x86程序，例如打字小游戏、nes模拟器等，但基于逐条指令模拟的方式实现的模拟器是有性能瓶颈的。即使将此实验中的代码经过充分重构和优化，也不可能达到与二进制翻译相似的性能水平。如果要做到像qemu那样，拥有堪比5年前CPU的性能（我们将其称为”真正可用”），就只能用jit的方法，将大多数指令尽可能放到宿主机CPU直接执行。

## 课设心得

在本次课设中，我意识到了基础设施的重要性，了解了分布式版本控制系统的使用，和GNU工具链的使用（包括GNU Make，Bash等全套Unix传统工具链）。遵从Unix哲学而设计出的工具链拥有与NT工具链不同的美感，他们易于组合并实现复杂的自动化功能。

小即是美。

让程序只做好一件事。

尽可能早地建立原型。

可移植性比效率更重要。

数据应该保存为文本文件。

尽可能地榨取软件的全部价值。

使用shell脚本来提高效率和可移植性。

避免使用可定制性低下的用户界面。

所有程序都是数据的过滤器。

同时，在本次课设中，我更加熟悉了计算机的工作方式，熟悉了i386指令集、图形程序和IO操作、文件与分页、保护模式和系统调用等特性的实现方式，对巩固CS基础知识有很大作用。

## 框架代码改进建议

<https://git.recolic.org/recolic-hust/hust-x86-simulator/merge_requests/4/diffs>

# 参考文献

1. DAVID A.PATTERSON(美).计算机组成与设计硬件/软件接口(原书第4版).北京：机械工业出版社.
2. David Money Harris(美).数字设计和计算机体系结构（第二版）. 机械工业出版社
3. 秦磊华，吴非，莫正坤.计算机组成原理. 北京：清华大学出版社，2011年.
4. 袁春风编著. 计算机组成与系统结构. 北京：清华大学出版社，2011年.
5. 张晨曦，王志英. 计算机系统结构. 高等教育出版社，2008年.