学　　号：2014218760　　密　　级：公开

合肥工業大學

Hefei University of Technology

本科毕业设计（论文）

UNDERGRADUATE THESIS

类　　型：	设计
题　　目：	基于 6502 CPU 的 NES 模拟器设计与实现
专业名称：	计算机科学与技术
入校年份：	2014 级
学生姓名：	罗能
指导教师：	安鑫-副教授
系名称　：	信息工程系
完成时间：	2018 年 6 月

合　肥　工　业　大　学

本科毕业设计（论文）

基于 6502 CPU 的 NES 模拟器设计与实现

学生姓名：罗能

学生学号：2014218760

指导教师：安鑫-副教授

专业名称：计算机科学与技术

系名称　：信息工程系

2018 年 6 月

A Dissertation Submitted for the Degree of Bachelor

Design and Implementation of a NES Emulator Based on 6502 CPU

By

Neng Luo

Hefei University of Technology

Hefei, Anhui, P.R.China

6 Month, 2018 Year

毕业设计（论文）独创性声明

本人郑重声明：所呈交的毕业设计（论文）是本人在指导教师指导下进行独立研究工作所取得的成果。据我所知，除了文中特别加以标注和致谢的内容外，设计（论文）中不包含其他人已经发表或撰写过的研究成果，也不包含为获得 合肥工业大学 或其他教育机构的学位或证书而使用过的材料。对本文成果做出贡献的个人和集体，本人已在设计（论文）中作了明确的说明，并表示谢意。

毕业设计（论文）中表达的观点纯属作者本人观点，与合肥工业大学无关。

毕业设计（论文）作者签名：　　签名日期：　　年　　月　　日

毕业设计（论文）版权使用授权书

本学位论文作者完全了解 合肥工业大学 有关保留、使用毕业设计（论文）的规定，即：除保密期内的涉密设计（论文）外，学校有权保存并向国家有关部门或机构送交设计（论文）的复印件和电子光盘，允许设计（论文）被查阅或借阅。本人授权 合肥工业大学 可以将本毕业设计（论文）的全部或部分内容编入有关数据库，允许采用影印、缩印或扫描等复制手段保存、汇编毕业设计（论文）。

（保密的毕业设计（论文）在解密后适用本授权书）

学位论文作者签名：　　指导教师签名：

签名日期：　　年　　月　　日　　签名日期：　　年　　月　　日

摘　要

NES[1](任天堂娱乐系统) 在 20 世纪 80 年代是世界上使用最广泛的电子游戏终端，其将许多游戏带入了家庭，并为当今电子游戏产业铺平了道路。

随着科技的发展，许多 NES 游戏已经无法在当今系统上游玩，然而归功于模拟器的存在，使得这些经典能够延续下去。

NES 是一个由 8 位 6502 CPU 组成的微型计算机，能够有条不絮地运行游戏程序。本课题设计并用 C++ 实现一个跨平台[2]的 NES 模拟系统，提供一个具体的软、硬件环境，以达到在现代操作系统中能够模拟并运行上个年代的 NES 游戏的目的。

关键词：模拟器；6502；计算机组成原理；NES

[1]全称为 Nintendo Entertainment System

[2]在 Win/Linux/Mac 三大平台运行

ABSTRACT

The NES (Nintendo Entertainment System) was the world's most widely used video game console system in the 1980s, bringing many games to the home and paving the way for today's video game industry.

With the development of science and technology, many NES games can no longer play on modern operating systems, but thanks to the presence of emulators, these classics can continue.

The NES is a minicomputer that consists of an 8-bit 6502 CPU that can run game programs. This paper design and implement a cross-platform NES emulator in C++, providing a specific hardware and software environment, in order to be able to achieve the purpose of emulation and runing NES games of the last decade in modern operating systems.

KEYWORDS: Emulator; 6502; Computer Organization; NES

目　录

插图清单

表格清单

1 引言

1.1 课题背景及意义

NES 是一个 8 位家用电子游戏终端系统，由任天堂公司开发与制作。最初于 1983 年 7 月 15 日发行于日本名叫 Famicom[3]的电子游戏机，后来于 1985 年发行于纽约，1986 到 1987 年遍布整个美国和欧洲，1987 年在澳大利亚发行。

当时在游戏机市场最畅销的时候，NES 在 1983 年的电子游戏行业崩溃[4]之后振兴了美国电子游戏行业。任天堂公司提出了严格的第三方开发者授权的商业模式来确保游戏质量，所有游戏必须通过任天堂的批准，并且第三方厂商每年只能开发一定数量的游戏，后来的 SNES[5]也采用了这种模式。正是因为这款游戏机的先进技术和严格的授权开发商业模式，使其成为电视游戏机的开山鼻祖。

在 2009 年时，NES 被 IGN[6]评为游戏历史上最伟大的电子游戏机[7]。

模拟器软件能够在一台电子设备或一个计算机程序中运行另外一台设备或程序，对于游戏模拟器而言，需要严格精确地模拟硬件，其中模拟 CPU 的重点是“精准”，比如硬件 BUG 一致、寻址正确无误、指令执行周期一致、中断优先级得模拟出来。除此之外还有总线、内存、外设的模拟。对于总线、内存模拟，需要考虑读写是否有效、大块数据正确传输等等。模拟器的关键之处就在于用代码实现了硬件的功能。

为了能够对大学所学知识加以应用，这个课题能够深入理解计算机是如何运行程序的，同时又能将经典游戏继续延续下去；为了能够跨平台流畅运行，还需要写出高兼容性、高性能代码；为了保证开发的效率，需要学习 6502 汇编，编写单元测试。

[3]也叫红白机，美国称 NES

[4]由于市场饱和，同时又充斥着大量粗制滥造的游戏

[5]Super Nintendo Entertainment System, 由任天堂于 1990 年发行的 16 位电子游戏机

[6]最大最权威的电子游戏评测网站

[7]`http://www.ign.com/lists/top-25-consoles/1`

1.2 课题研究现状

由于任天堂未公布相关硬件细节，许多 NES 模拟器开发者通过对硬件逆向工程获得了许多信息，将这些信息整合起来就能够了解内部工作原理，足以实现一个模拟器了。

目前有以下三种方法来实现模拟器：

直接翻译 读取源程序 PC 指针上的指令，并翻译成目标机器指令，更新 PC 指针、内存。由于在执行过程中进行翻译，可能会影响性能问题。

静态编译 将源程序一次性编译到能够在目标系统上运行的程序，然而静态编译无法判断运行时遇到的分支跳转语句。

动态编译 结合以上两种方式，算是一种折中方案。

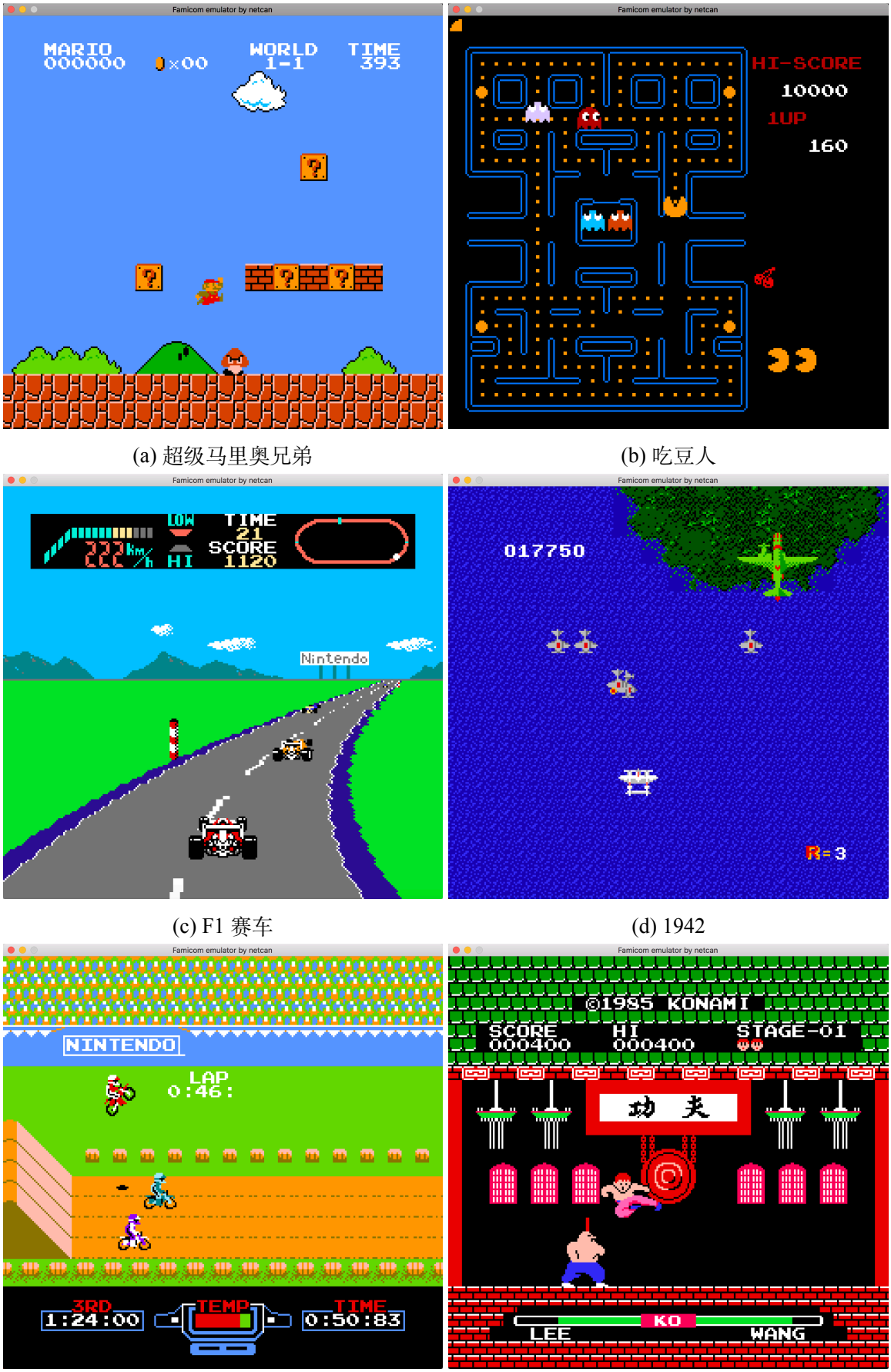
最终本课题采取直接翻译的方法来实现模拟器，考虑性能问题，利用 C++ 来实现，以达到最大性能；为了显示图形、处理键盘输入，利用 SDL 库绘图、响应；为保证模拟器能够正确工作、重构，利用 Google Test 框架来单元测试。为了方便跨平台编译，使用 CMake 自动化构建。

1.3 预期成果

由于写一个兼容目前所有游戏将超出本课题范围，这里最终成果是能够运行超级马里奥兄弟、吃豆人、F1 赛车等经典游戏，如图1.1所示，从技术角度上来讲也极具挑战性，因为它们或多或少依赖一些硬件上的特性，实现起来需要特别处理。

比较遗憾的是，本课题还未实现 Mapper[8]，声音模块，实现它们也将是一件有意思的事情。

[8]能够对换卡带中的程序 ROM 到 CPU 内存中，用来运行大容量游戏



(a) 超级马里奥兄弟　(b) 吃豆人

(c) F1 赛车　(d) 1942

(e) 摩托　(f) 功夫

图 1.1　本模拟器模拟的一些经典游戏

2 任天堂娱乐系统

本章主要介绍一下 NES 的各个硬件模块的相关细节。

2.1 CPU

NES 采用由 Ricoh 公司改造的 8 位 6502 的 MOS 处理器，代号 2A03/2A07[9]。该改造后的 CPU 不同于通用的 6502 的是，它能够处理声音，后果是无法处理 BCD 码[10]，除此之外其余部分例如指令集都是一样的。

6502 CPU 是一个小端 CPU，即高地址存放高字节，低地址存放低字节。举个例子，16 进制数 0x1234 的 0x34 字节的内存地址是 x，那么 0x12 的地址是 $(x+1)$。CPU 主频为 1.79 MHz，基频为 21.48 MHz，即主频对基频 12 分频。

NES 使用内存 I/O 映射技术，使得处理器写入指定内存位置，即可对外设进行通讯（PPU、控制器设备等等）。

图 2.1　基于 6502 改造的 2A03 处理器

2.1.1 系统总线

如图2.2所示，NES 采用三总线结构：

数据总线 8 位双向数据总线，在 CPU 与 RAM、I/O 设备之间双向传输（读、写），在程序卡带 ROM 之间单向传输（只读）。

控制总线 8 位控制线，用于控制目标状态是读还是写。

地址总线 16 位地址总线，用于指定目标的位置。

同时，内存被划分为三个部分：

[9] 2A03 用于 NTSC 版本，而 2A07 用于 PAL 版本，本课题采用 NTSC 制式

[10] 用 4 个比特位来表示数字 0-9

- 卡带中的 ROM 区，只读存储器，由 MMC 组件[11]来访问，扮演内存块兑换的角色
- CPU 的 RAM 区
- I/O 寄存器映射区，用于 CPU 与外部组件 PPU[12]、控制器进行通信

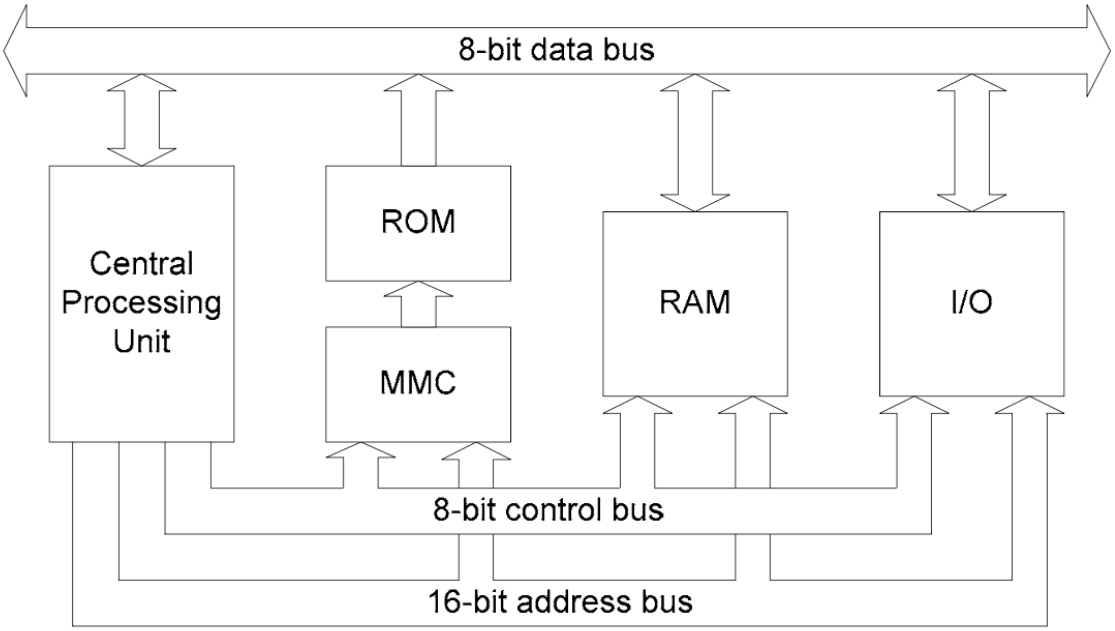


图 2.2　NES 的系统总线

2.1.2 内存

CPU 的 16 位的地址线，能够支持 64KB 大小的内存，寻址范围：0x0000-0xffff，如表2.1所示。

若游戏 ROM 只有一块（16KB 为单位），则加载到内存 0x6000，0x8000 这两部分中；若只有两块，则第一块加载到 0x6000，第二块加载到 0x8000；若游戏 ROM 超过两块 ($16KB \times 2 = 32KB$) 大小，将使用 Mapper 内存块对换来决定将哪块加载进内存，本课题暂未实现 Mapper。

[11]Memory Mapper Chip，也称为 Mapper
[12]图形处理器

表 2.1　NES 的内存区域

地址	大小	描述
0x0000-0x00FF	256B	Zero Page（也称为零页），内存的第一页，用于快速寻址
0x0100-0x01FF	256B	栈区，空递减堆栈
0x0200-0x07FF	1.5KB	RAM 区
0x0800-0x1FFF	6KB	这块区域用于对 Zero Page 镜像 3 次，意味着，写到 0x0000，同时也会写到 0x0800, 0x1000, 0x1800
0x2000-0x401F	16KB	内存映射 IO 寄存器，从 0x2000-0x2007 这 8 个字节镜像填充满 0x2008-0x3FFF 区域
0x4020-0x5FFF		扩展区
0x6000-0x7FFF	8KB	SRAM，用于访问卡带中的 RAM，保存游戏用
0x8000-0xFFFF	32KB	这块区域被用于访问卡带的程序 ROM，程序 ROM 以 16KB 为一个单位块 (bank)，一共两块

2.1.3 寄存器

6502 CPU 有 6 个寄存器，其中 3 个特殊寄存器，程序计数器（PC）、栈指针（SP）、程序状态寄存器（P），3 个通用寄存器，累加器（A）、X、Y 寄存器，表2.2详细描述了各寄存器的作用。

表 2.2　6502 CPU 各寄存器作用

寄存器名称	寄存器位数	描述
程序计数器（PC）	16	存放下一条待执行的指令地址
栈指针（SP）	8	指向栈区（0x0100-0x01ff），从 0x0100 的内存位置作为偏移量，空递减堆栈，也不会检测栈溢出（0x00-0xff）
程序状态寄存器（P）	8	受到指令执行后的影响，标记程序状态
累加器（A）	8	存储算数、逻辑运算的结果
X 寄存器	8	一般做计数器或者用于一些寻址方式的偏移值，或者 SP 的临时值
Y 寄存器	8	和 X 寄存器一样，但是不能用来做 SP 的临时值

8 位状态寄存器（P）受到指令执行后的影响，其中每一位都有特别的含义，这些标志位在寄存器中的顺序如图2.3所示：

- 负数标志位（N），当运算结果最高位第 7 位为 1 的时候置位，表明负数。

- 溢出标志位（V），当两个补码运算产生非法的结果置位，例如正 + 正为负的时候。
- Break 指令标志（B），用于标记当 BRK 指令执行后，产生的 IRQ 中断（软件中断）。
- 十进制模式（D），6502 通过设置该标志位切换到 BCD 模式，由于 2A03 不支持 BCD，所以这位是无效的。SED 指令置位，CLD 指令复位。
- 中断屏蔽标志位（I），通过设置该位可以屏蔽 IRQ 中断。SEI 指令置位，CLI 指令复位。
- 零标志位（Z），当运算结果为 0 的时候置位。
- 进位标志位（C），当运算结果最高位第 7 位符号翻转的时候置位。SEC 置位，CLC 复位。

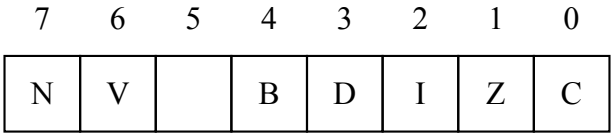


图 2.3　程序状态寄存器 P

2.1.4 中断

中断用于处理硬件、软件触发的信号，表明发生了某个事件需要注意。NES 有三种中断：不可屏蔽中断（NMI）、可屏蔽中断（IRQ）、复位（Reset），具体如表2.3所述。

表 2.3　NES 的中断

中断类型	向量地址	描述
NMI	0xFFFA	当 PPU 中每一帧图像渲染结束时产生 VBlank 信号触发该中断（PPU 的控制寄存器 1 可设置是否发出 VBlank 信号）
Reset	0xFFFC	当用户按下复位按钮的时候产生
IRQ	0xFFFE	可屏蔽中断，受到中断屏蔽标志位 (I) 的影响，也能被 BRK 指令（软件中断）触发

各个中断优先级如下：Reset > NMI > IRQ。在中断产生的时候，执行一个中断一般需要 7 个机器周期，处理步骤如下：

1. 识别中断请求
2. 完成当前指令
3. 将 PC，P 寄存器入栈（保存现场）
4. 设置中断屏蔽标志，以防再次中断（关中断）
5. 将 PC 设置为位于中断向量表的中断程序地址
6. 执行中断程序
7. 执行 RTI 指令（相当于 x86 的 IRET 指令），出栈恢复到 PC，P 寄存器（恢复现场）
8. 程序继续执行

2.1.5 寻址模式

6502 有 13 种寻址模式，介绍如下。

隐式寻址 操作数隐藏在操作码中无需给出，也就是没有操作数

```
CLC ; 清除进位标志位
```

累加器寻址 只有累加器这一个操作数

```
LSR A ; 对累加器 A 进行逻辑右移
```

立即数寻址 操作数为第二个字节指明的常量，在 6502 汇编中用 # 号来表明

```
LDA #10; 将 10 存放到累加器 A 中
```

零页寻址 第二字节为操作数的地址，由于只用一个字节来表示地址，故操作数地址范围在 0x00-0xff，即零页，在 6502 汇编中用 $ 来表明 16 进制地址

```
LDA $00; 将内存地址 0x00 上的存储单元的值作为操作数存放到累加器 A 中
```

零页 X 变址寻址 第二个字节作为基址，加上 X 寄存器的值作为最终操作数地址[13]

```
STY $10,X; 将内存地址 (0x10 + X) 上的存储单元的值存放到寄存器 Y 中
```

零页 Y 变址寻址 和零页 X 变址一样，只不过是换成了 Y 寄存器

```
LDX $10,Y; 将内存地址 (0x10+Y) 上的存储单元的值存放到寄存器 X 中
```

[13]需要注意的是地址高位不进位，地址始终限制在 0x00-0xff 范围内

相对寻址 分支跳转指令专用，第二个字节操作数（-128 到 127）加到 PC 指针上作为跳转目标的地址

```
BEQ $2d; 若结果为 0 则跳转到 PC+0x2d 的地址
```

绝对寻址 操作数地址为第二、三字节组成的 16 位地址

```
LDA $1234; 将内存地址 0x1234 上的存储单元的值存放到累加器 A
```

绝对 X 变址寻址 第二、三字节组成的 16 位地址加上 X 寄存器的值作为操作数地址

```
STA $3000,X; 将内存地址 (0x3000+X) 上的存储单元的值存放到累加器 A
```

绝对 Y 变址寻址 和绝对 X 变址一样，只不过是换成了 Y 寄存器

```
STA $3000,Y; 将内存地址 (0x3000+Y) 上的存储单元的值存放到累加器 A
```

间接寻址 JMP 跳转指令专用，第二、三字节组成的 16 位地址内存单元上的值作为地址

```
JMP $FFFC; 跳转到 Reset 中断向量
```

零页变址间接寻址 第二字节为基址，加上 X 寄存器的值组成的零页内存地址（间址）单元上的值作为操作数地址

```
LDA ($40,X); (0x40+X) 作为间址，作为操作数地址取操作数存放到累加器 A
```

间接寻址变址 第二字节为间址，取 16 位操作数并加上 Y 作为操作数有效地址

```
LDA ($40),Y; 取 0x40, 0x41 组成 16 位地址，加上 Y 作为操作数有效地址，取操作数存放到累加器 A
```

2.1.6 指令集

6502 有 56 条不同的指令，各指令因为不同的寻址方式有不同的变种，总共有 151 个操作码[14]。指令长度在 1 到 3 字节，第一字节为操作码，后面的为操作数。具体的指令集细节可参考文献 [1]，指令可分为下几类：

- Load/Store 指令，读内存数据到寄存器，从寄存器写到内存
- 寄存器转移指令，复制 X 或 Y 寄存器内容到累加器（A）中，或相反
- 栈操作指令，入栈或出栈，根据 X 寄存器的值来读写栈指针

[14]还有 105 个未在 CPU 官方文档注明的操作码，本课题也对它们进行实现

- 逻辑运算指令，对累加器（A）和内存中的值进行逻辑运算
- 算术运算，对寄存器和内存进行算术运算
- 增减指令，对 X,Y 寄存器或内存的值进行增减运算
- 位移指令，对累加器（A）或内存中的值进行位移操作
- 跳转/调用指令，跳到指定地址继续执行
- 分支指令，当条件满足（P 寄存器）的时候跳到指定地址继续执行
- 操作状态寄存器指令，设置状态寄存器的某些标志位
- 系统指令，执行一些系统功能

2.2 PPU

Ricoh 公司也提供了 2C02/2C07[15]芯片作为图形处理器 PPU，PPU 的寄存器映射到 CPU 内存的 0x2000-0x2007 和 0x4014 区，这些特殊的寄存器用来控制图像信息，例如背景滚动、精灵图控制、数据传输等等。

PPU 的频率是基频的 4 分频，即 5.37MHz，正好是 CPU 频率的 3 倍。

2.2.1 显存

同样的，PPU 也有自己的内存，又称作显存（VRAM，Video RAM）。不像 CPU，虽然 PPU 也能寻址 64KB 范围空间，但是它只有 16KB 物理内存，其他区域是物理内存的镜像。表2.4为显存的布局。除了显存，PPU 还有一块 256 字节的 OAM 专门用来存放精灵信息（如坐标，图块号，垂直、水平翻转，颜色等等），每个精灵需要 4 个字节，一共能存放 64 个精灵信息。

NES 的调色板一共有 56 种颜色（用 6 个比特位来表示索引），然而这些颜色不能同时显示在图形上，显存中有 2 个调色板（分别位于 0x3F00-0x3F0F, 0x3F10-0x3F1F）：背景调色板、精灵调色板。每个调色板能够存放 16 种颜色，因为存放的是索引，所以也只需要用 6 个比特位来表达一种颜色，由于这两个调色板某些字节被镜像，最终只能显示 25 种颜色。

[15]2C02 用于 NTSC 版本，2C07 用于 PAL 版本

表 2.4　PPU 显存布局

地址	大小	描述
0x0000-0x0FFF	4KB	图块表 (Pattern Table)0，存放背景、精灵图块的颜色索引的低两位
0x1000-0x1FFF	4KB	图块表 1，同上
0x2000-0x23FF	1KB	名称表 (Nametable)0，存放背景信息
0x2400-0x27FF	1KB	名称表 1，同上
0x2800-0x2BFF	1KB	名称表 2，同上
0x2C00-0x2FFF	1KB	名称表 3，同上
0x3000-0x3EFF		0x2000-0x2EFF 的镜像
0x3F00-0x3F1F	32B	调色板，存放背景/精灵的颜色索引
0x3F20-0x3FFF		0x3F00-0x3F1F 的镜像

显存中的图块表区域，用来存放背景、精灵图块的调色板指针的低 2 位。背景、精灵图块颜色信息需要一共需要 4 个比特位来存放。图块表一共有 2 个，每个 4KB，图块的基本尺寸为 8x8 像素，每行 8 个像素点用一个字节来表示调色板指针的一位，由于图块表只存放颜色索引的低 2 位，所以需要 16 字节大小来存放一个图块，一个图块表能存放 256 块，图2.4按左右顺序排列了这 2 个图块表 (16x16=256 块)。背景图块有 8x8 这一种模式，而精灵图块支持 8x8, 8x16[16]两种模式。

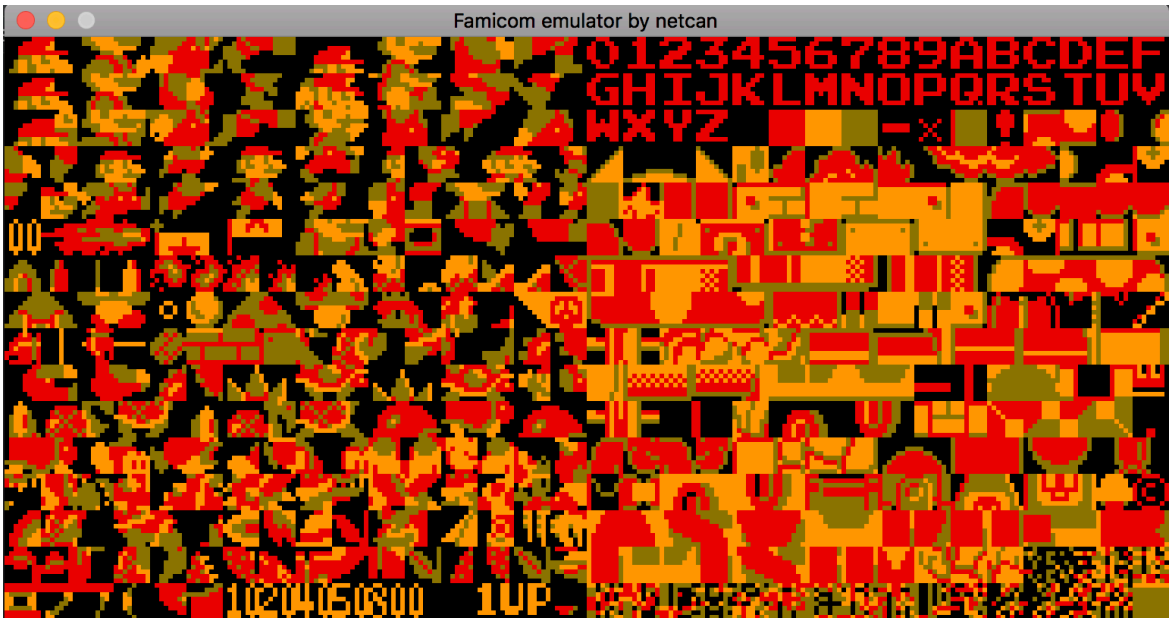


图 2.4　超级马里奥的图块表

[16]由 2 个 8x8 基本图块组成

而背景、精灵图块的调色板指针的高 2 位分别存放于名称表中的属性表、OAM 中。

虽然有 4 块名称表用于存放背景信息，每块 1KB，但实际上能用的只有 2 块，另外 2 块做镜像用，从而形成了垂直镜像、水平镜像等模式。名称表的每个字节表明图块表中的块号，由 PPUCTRL 寄存器来选择哪一个图块表。如图2.5所示，名称表由 32x30=960 块组成，形成 256x240 大小的背景图形，一共占用 960 字节，而剩下的 64 字节区域也叫属性表，用于保存背景图块的调色板指针的高两位。

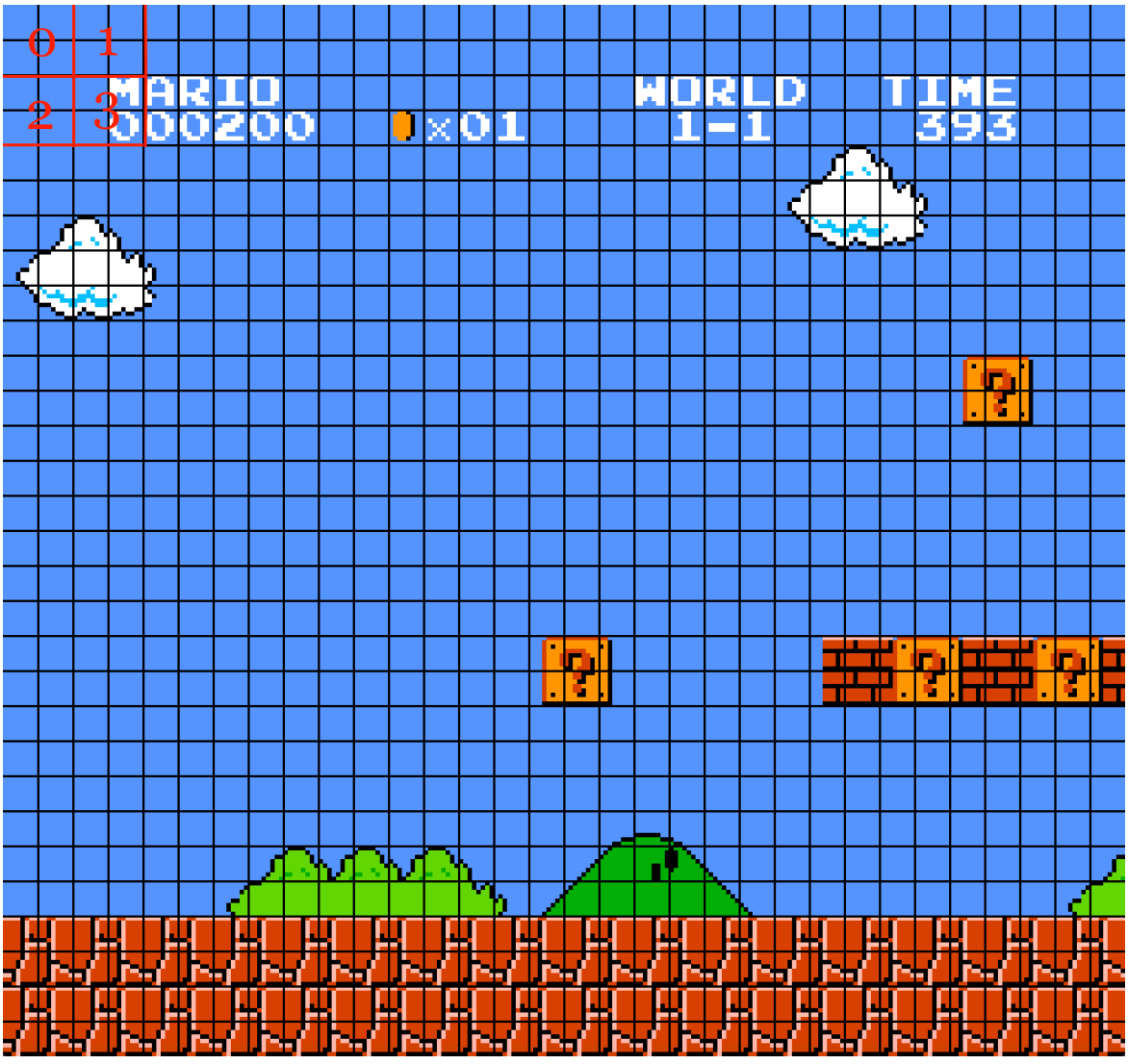


图 2.5　超级马里奥的名称表，背景中的每一个方块由图块表中的 8x8 图块组成，而左上角标注的 0,1,2,3 由属性表中的一个字节来描述背景图块调色板指针的高两位

OAM 用 4 个字节来描述一个精灵信息，第一个字节描述精灵的 Y 坐标，第二个字节描述精灵的图块号，第三个字节描述精灵的调色板指针的高两位、是否水平、

垂直翻转，是否显示，最后一个字节描述精灵的 X 坐标。OAM 也支持 DMA，可高效地将 CPU 内存数据写入 OAM 中。通过写入 OAMDMA 寄存器来触发，写入 N 将会从 CPU 内存 $N \times 0x100$ 起始地址开始连续对 OAM 写入 256 个字节，这期间将会发生周期挪用现象，即 CPU 无法访存，也将无法进一步获取指令信息，直到 DMA 过程完成。

2.2.2 寄存器

PPU 的各个寄存器主要作用见表2.5。

表 2.5　PPU 的各个寄存器主要作用

寄存器名	地址	属性	主要用途
PPUCTRL	0x2000	写	用于控制是否产生 NMI 中断、精灵的高度、背景块的图块表选择、名称表选择
PPUMASK	0x2001	写	是否显示背景、精灵
PPUSTATUS	0x2002	读	描述 PPU 的状态，是否处于 VBlank
OAMADDR	0x2003	写	OAM 读写地址
OAMDATA	0x2004	读、写	OAM 读写数据
PPUSCROLL	0x2005	写两次	背景滚动的位置（用于产生横、竖向滚动效果）
PPUADDR	0x2006	写两次	PPU 读写地址
PPUDATA	0x2007	读、写	PPU 读写数据
OAMDMA	0x4014	写	DMA

PPUSCROLL, PPUADDR 共用内部寄存器 [2]，各需要写两次生效，前者依次写摄像机的 x, y 坐标，后者依次写高、低地址。

2.2.3 渲染

PPU 绘制一帧需要 341x262=89342 个 PPU 时钟周期，可分为三个阶段：渲染、HBlank、VBlank，具体如图2.6所示。

渲染部分大小为 256x240，按行渲染，期间每一个 PPU 时钟周期计算并绘制一个像素点，同时会获取背景块信息、更新当前绘制坐标等等 [3]。

在渲染期间每一行后的 HBlank 阶段，会取出下一行绘制所需要的精灵信息。[4]

当渲染完成后，会经过 VBlank 阶段，根据 PPUCTRL 寄存器来决定是否产生

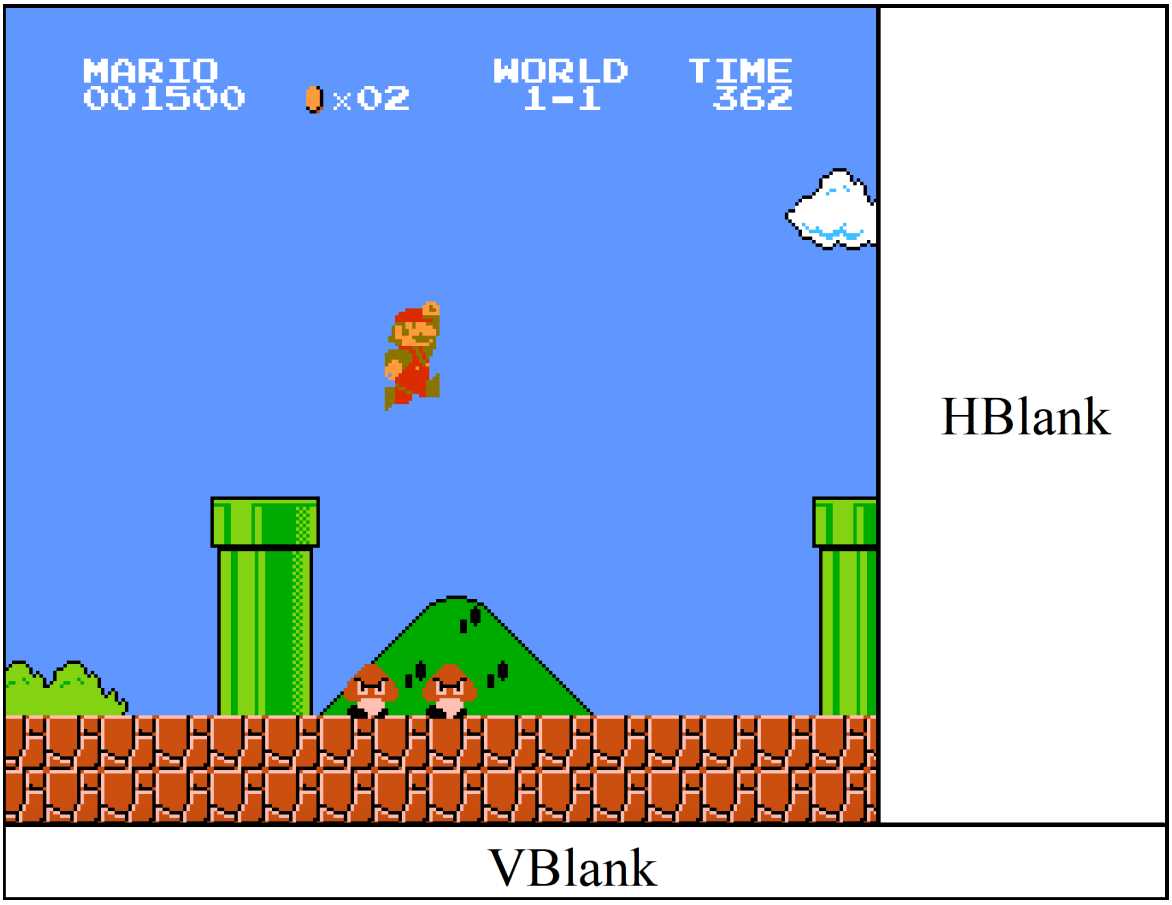


图 2.6　PPU 的渲染

NMI 中断，从而进入中断程序，由中断程序来更新名称表、OAM 内容，从而更新下一帧所需要的背景、精灵信息。

2.3 标准控制器

图2.7为标准控制器（手柄），是 NES 的输入设备，没有它也就没法游玩游戏了。控制器有 8 个键，上、下、左、右、选择、确认、A、B，采用 8 位移位寄存器实现，每一个比特位代表一个键是否按下。默认情况下支持两个控制器，分别映射至内存 0x4016, 0x4017 位置，每次读一位并且移动一位。通过对 0x4016 写入下降沿电压来触发重载这两个移位寄存器。

图 2.7　NES 的标准控制器

2.4 卡带

图2.8展示了 NES 用的游戏卡带，NES 游戏程序 ROM 分发于卡带中，最原始的卡带由 PCB 板和 ROM 芯片组成。

从图2.8中可以看到 ROM 有两块，分别为 CHR ROM 和 PRG ROM，前者用于存放图像数据（PPU 的图块表部分），后者用于存放游戏程序（CPU 内存的 0x8000-0xFFFF 区）。

图 2.8　NES 的卡带

2.4.1 Mapper

由于 16 位地址总线问题，导致程序尺寸被限制在 32KB（而图像数据仅有 8KB），随着科技进步，ROM 存储器越来越便宜，容量越来越大，游戏的需求也越来越高，于是任天堂提出了 MMC 芯片，也就是 Mapper，通过内存块[17]对换技术[18]，使得游戏容量能够突破限制。而其他生产商也研发了自己的 Mapper，形成了多种多样的形式，例如能支持不同的块尺寸、支持 RAM 等等，从而可以为游戏添加存档功能。

2.4.2 iNES 文件格式

通过一些拷贝装置，可将卡带上的数据拷贝到电脑硬盘上，然而仅仅有这些数据还是不行的，需要一种文件格式来描述。最初由 Marat Fayzullin 开发了一款名叫 iNES 的模拟器，他提出的文件格式也在今后的 NES 模拟器中使用最广泛，后缀名为.nes。iNES 文件格式记录了 Mapper 类型、ROM 大小、ROM 数据、NTSC/PAL 制式等信息。

[17]以 16KB 为单位

[18]准确来说是 Bank Switching

3 相关技术介绍

3.1 Simple DirectMedia Layer

SDL(Simple DirectMedia Layer) 是一个通过 OpenGL 和 Direct3D 提供了对声音、键盘、鼠标、手柄、图形硬件访问接口的跨平台开发库。广泛用于视频播放器、模拟器、游戏开发中。

SDL 由 C 语言写成，可在 C/C++ 中使用，同时也支持其他语言，例如 Python。

由 SDL 开发的一些经典作品有：DOTA2、求生之路 2、QEMU 等。

(a) DOTA2　　(b) 求生之路 2　　(c) QEMU

图 3.1　由 SDL 开发的一些经典作品

本课题使用 SDL 来绘图、处理键盘输入事件、定时器。

3.2 Google Test

Google Test 是一个跨平台 C++ 单元测试框架，编写测试样例也相当简单，使得调试过程相当具体，满足了许多开发人员的需求。

使用 Google Test 框架的经典项目有：Chrome、LLVM、OpenCV 等。

需要注意几个术语可能会混淆，由于历史原因，Google Test 将同一组件下相关的测试称为测试用例（Test Case），而目前出版的包括国际软件测试资质认证委员会[19]和许多软件测试书籍在内，将这个称为测试套件（Test Suite）。Google Test 将指定程序输入验证输出这个行为称作测试（Test），而 ISTQB 将这个称为测试用例（Test Case）。

本课题使用 Google Test 来验证各个程序模块（CPU/PPU 等）是否正确工作。

[19]International Software Testing Qualifications Board (ISTQB)

3.3 CMake

CMake 是一个跨平台的自动化构建系统，通过配置文件来控制整个构建过程，和 Linux/Unix 下的 Makefile 相似，配置文件名为 CMakeLists.txt。

CMake 并不直接构建出最终的程序，而是生成构建文件（Linux/Unix 下的 Makefile 或者 Windows VC++ 下的 projects/workspace），再用一般的构建方式生成程序。

使用 CMake 的经典项目有：LLVM/CLang、MySQL、OpenCV、Qt 等。

本课题使用 CMake 来产生跨平台构建文件。

3.4 Valgrind

Valgrind 是一个用于检测内存泄露、性能分析的程序。Valgrind 发行版目前包含了六个工具：一个内存错误检测器、两个线程错误检测器、一个缓存和分支预测分析器、一个调用图分析器、一个堆分析器，能够跨平台运行。

本课题使用 Valgrind 的 cachegrind 和 callgrind 工具来进行性能优化。由于这是一个命令行工具，这里推荐使用 KCachegrind 对 Valgrind 输出日志进行可视化，方便分析。

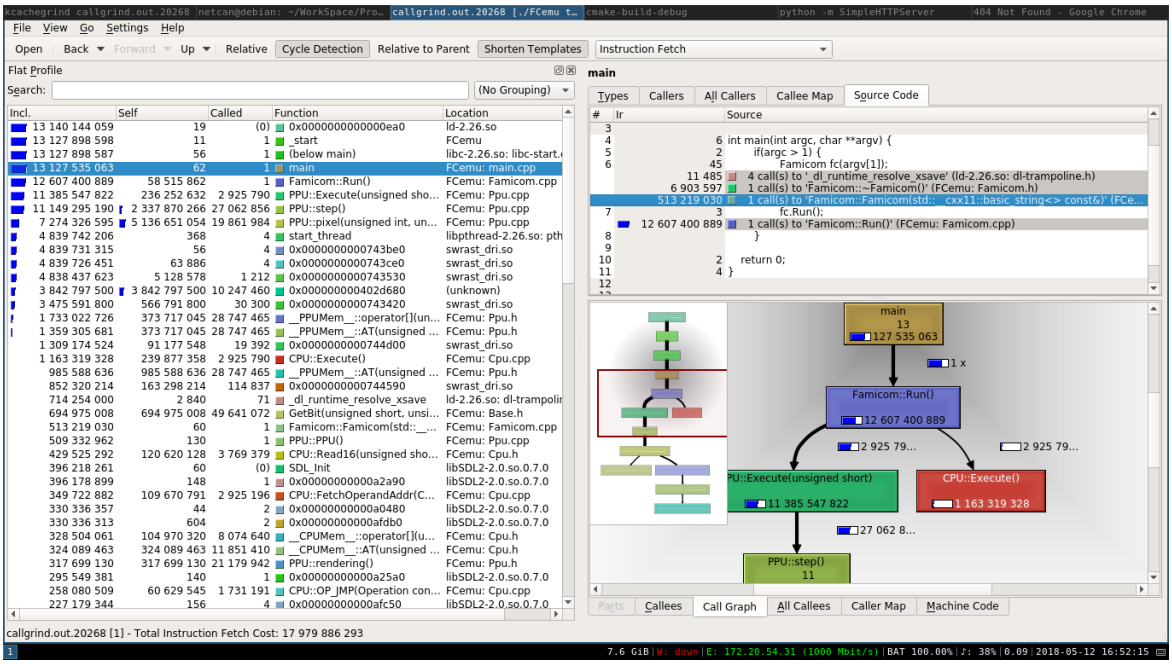


图 3.2　Valgrind/KCachegrind 性能分析工具，可看到程序运行时每个函数需要的指令数、调用次数、位置，代码每一行所需要的指令数，调用图等信息。

4 系统设计

本课题只对 NES 的这几个硬件模块利用面向对象思想进行了实现：CPU，PPU，标准控制器、卡带，整体结构类似图2.2。

由于 CPU 和 PPU 是同时运行，PPU 的时钟频率是 CPU 的 3 倍，在系统设计的时候可以采用 CPU、PPU 分时运行的方法，每当 CPU 执行完一条指令，紧接着 PPU 执行 3 倍于 CPU 指令周期长度，不断交替执行，从宏观上看，这两者是同时运行的。

下面依次对每个模块的设计进行简要说明。

4.1 CPU

CPU 的工作流程由以下几个阶段组成：

1. 取指，这个阶段根据 PC 指针从内存中取出操作码
2. 译码，根据操作码来获取操作数
3. 执行，通过算数逻辑单元进行运算
4. 访存、写回，更新内存、寄存器的值
5. 更新 PC 指针

CPU 每执行一条指令，返回该指令的时钟数，供 PPU 运行对应长度的时钟数。

4.2 PPU

PPU 在渲染期间的每一个时钟周期计算名称表、属性表、OAM 精灵数据地址，接着获取对应数据[20]，再形成一个像素点，最终形成 256x240 大小的图像，当进入 VBlank 阶段时，利用 SDL 将完全的一帧图像绘制出来。

[20]这里取出的信息是下两个图块的，绘制的像素点是前两块的，即提前存好的信息

4.3 标准控制器

采用一个字节变量来模拟移位寄存器，当对 0x4016 写入下降沿数据时，将当前键位信息存入变量中；当对 0x4016 读取数据时，每次移出一个比特位来表示键位是否按下。

4.4 卡带

卡带的功能比较简单，读取当前的 ROM 文件，然后将数据复制到 CPU、PPU 对应的内存区域。

5 系统实现

5.1 CPU

5.1.1 内存类实现

程序 ROM 数据需要加载进内存，供 CPU 运行，那么可对 CPU 内存类定义：

```
class __CPUMem__ {
private:
    uint8_t           Ram[0x800]; // 0x0000-0x07ff
    uint8_t    *PPURegister[0x8]; // 0x2000-0x2007
    uint8_t    *IORegister[0x20]; // 0x4000-0x401f
    uint8_t ExpansionRom[0x1fe0]; // 0x4020-0x5fff
    uint8_t         SRam[0x2000]; // 0x6000-0x7fff
    uint8_t  LowerPRGRom[0x4000]; // 0x8000-0xbfff
    uint8_t  UpperPRGRom[0x4000]; // 0xc000-0xffff

    inline const uint8_t &AT(uint16_t addr) const {
            return  addr < 0x2000 ? Ram[addr & 0x7ff]:
                    addr < 0x4000 ? *PPURegister[addr & 0x7]:
                    addr < 0x4020 ? *IORegister[(addr - 0x4000)]:
                    addr < 0x6000 ? ExpansionRom[addr - 0x4020]:
                    addr < 0x8000 ? SRam[addr - 0x6000]:
                    addr < 0xc000 ? LowerPRGRom[addr - 0x8000]:
                                    UpperPRGRom[addr - 0xc000];
    }
public:
    uint8_t &operator[](uint16_t addr) { return AT(addr); }

    using iterator = MemIterator<__CPUMem__>; // 迭代器，实现见下文
    iterator begin() { return iterator(this, 0); }
    iterator end() { return iterator(this, 0x10000); }
};
```

为方便对内存进行操作，这里通过继承 STL 的迭代器类 iterator 实现 MemIterator，可达到类似 C++ STL 容器中的迭代器，从而可以使用 algorithm 中诸如 copy 等方法。

```
template<class MEMType>
class MemIterator: public std::iterator // 内部用 32 位来表示地址，是因为 16 位不好判断 end()
        <std::random_access_iterator_tag, uint8_t, ptrdiff_t, uint32_t, uint8_t &> {
public:
    MemIterator(): parent(NULL), addr(0) {}
    MemIterator(MEMType *parent, pointer addr): parent(parent), addr(addr) {}
```

```
7
8       reference operator*() { return parent->operator[](addr); }
9       uint8_t * get_raw_pointer() { return &parent->operator[](addr); }
10
11      MemIterator& operator++() { ++addr; return *this; } // ++it
12      MemIterator& operator--() { --addr; return *this; } // --it
13
14      MemIterator& operator+=(const uint16_t value) { this->addr += value; return *this; }
15      friend MemIterator operator+(MemIterator lhs, const uint16_t& rhs) { lhs += rhs; return
        ↪  lhs; }
16
17      friend bool operator==(const MemIterator &lhs, const MemIterator &rhs) { return lhs.addr
        ↪  == rhs.addr; }
18      friend bool operator< (const MemIterator &lhs, const MemIterator &rhs) { return lhs.addr
        ↪  < rhs.addr; }
19
20      friend difference_type operator-(const MemIterator &lhs, const MemIterator &rhs) {
        ↪  return lhs.addr - rhs.addr; }
21
22      uint8_t &operator[](uint16_t value) { return parent->operator[](addr + value); }
23  private:
24      pointer addr;
25      MEMType *parent;
26  };
```

在卡带将程序数据加载进内存 0x8000 位置时，可如下调用：

```
1  __CPUMem__::iterator cpuRamIt = cpu.mem.begin();
2  std::copy(PRGRomData, PRGRomData + sizeof(PRGRomData), cpuRamIt + 0x8000);
```

5.1.2 指令类定义

CPU 主要任务就是解析操作码，并执行相应的动作，可以对指令类定义如下：

```
1  struct Operation {                              // 指令
2      uint8_t code;                               // 8 位操作码
3      CPU::OpAddressingMode addressing_mode;      // 寻址模式
4      uint8_t bytes, cycles;                      // 操作码的长度、周期数
5      uint8_t (*exe)(OpExeFuncArgs);              // 令的具体动作，返回执行的 cycles 数目
6  };
```

从定义中可以看到，Operation 类详细地记录了一条指令所需要的信息，需要一个指令表，存放这些指令数据，定义如下：

```
1  static const Operation op_entity[] = {
```

```
        { 0x69, OpAddressingMode::    Immediate, 2, 2, CPU::OP_ADC },
        { 0x65, OpAddressingMode::     ZeroPage, 2, 3, CPU::OP_ADC },
        { 0x75, OpAddressingMode::    ZeroPageX, 2, 4, CPU::OP_ADC },
        { 0x6D, OpAddressingMode::     Absolute, 3, 4, CPU::OP_ADC },
        { 0x7D, OpAddressingMode::    AbsoluteX, 3, 4, CPU::OP_ADC },
        { 0x79, OpAddressingMode::    AbsoluteY, 3, 4, CPU::OP_ADC },
        { 0x61, OpAddressingMode::IndexIndirect, 2, 6, CPU::OP_ADC },
        { 0x71, OpAddressingMode::IndirectIndex, 2, 5, CPU::OP_ADC },
        { 0x29, OpAddressingMode::    Immediate, 2, 2, CPU::OP_AND },
        { 0x25, OpAddressingMode::     ZeroPage, 2, 3, CPU::OP_AND },
        { 0x35, OpAddressingMode::    ZeroPageX, 2, 4, CPU::OP_AND },
        { 0x2D, OpAddressingMode::     Absolute, 3, 4, CPU::OP_AND },
        { 0x3D, OpAddressingMode::    AbsoluteX, 3, 4, CPU::OP_AND },
        { 0x39, OpAddressingMode::    AbsoluteY, 3, 4, CPU::OP_AND },
        { 0x21, OpAddressingMode::IndexIndirect, 2, 6, CPU::OP_AND },
        { 0x31, OpAddressingMode::IndirectIndex, 2, 5, CPU::OP_AND },
        ... // 省略剩下的指令定义
}
static const Operation *optable[0xff + 1];
for(const auto & op: op_entity) optable[op.code] = &op;
```

将 256 条指令[21]存入到 optable 表中，当 CPU 取出的操作码为 0x69，就通过 optable[0x69] 找到正确的 Operation 对象。

有了指令表，还需要实现对应的 exe 函数，例如实现 ADC，AND 指令：

```
OpExeFuncDefine(OP_ADC) {
    /** 将内存中的数据和累加器 A 与状态寄存器 P 的进位标志位相加，结果存到累加器 A 中，并更新状
    ↪  态寄存器的 Z, C, N 位。**/
    uint8_t operand = cpu->Read8(opd_addr); // 读取内存上的数据
    uint16_t result = uint16_t(cpu->A) + uint16_t(operand) + cpu->P.Carry; // 计算结果
    // 更新各个标志位
    cpu->P.Overflow = GetBit(result, 0x8) ^
                      GetBit( (cpu->A & uint8_t(0x7f)) + (operand & uint8_t(0x7f)) +
                      ↪  cpu->P.Carry, 0x7); // 补码运算
    cpu->P.Carry = GetBit(result, 0x8); // 原码
    cpu->P.Zero = (result & 0xff) == 0;
    cpu->P.Negative = GetBit(uint8_t(result), 7);
    cpu->A = uint8_t(result);
    return self.cycles; // 返回执行的周期数
}
OpExeFuncDefine(OP_AND) {
    /** 将内存中的数据和累加器 A 进行与运算，并更新状态寄存器的 Z, N 位。**/
    uint8_t operand = cpu->Read8(opd_addr); // 读取内存上的数据
    cpu->A &= value;
    cpu->P.Negative = GetBit(cpu->A, 7);
    cpu->P.Zero = (cpu->A == 0);
    return self.cycles; // 返回执行的周期数
```

[21]这里包含了非官方指令集

```
}
```

5.1.3 主类定义

接下来看看 CPU 类的结构：

```
class CPU {
private:
    uint16_t PC; // 程序计数器
    uint8_t SP; // 栈指针，$0100-$01ff
    uint8_t A, X, Y; // 累加器，X，Y 寄存器
    ProcessorStatus P; // 状态寄存器
    __CPUMem__ mem;
    uint32_t cycles; // 累计执行周期
    PPU *ppu; // 控制 PPU
    Joypad *pad; // 外设
    bool nmi, irq, dma; // 是否产生中断、DMA
    enum class InterruptVector: uint16_t { // 中断向量表
        NMI = 0xfffa,
        Reset = 0xfffc,
        IRQ = 0xfffe
    };
    ...
}
```

CPU 类包含了各个寄存器，内存，需要外接的 PPU，标准控制器 Pad，中断向量表，以及产生的中断类型，还有是否发生 DMA。

5.1.4 工作流程实现

接下来就是使得 CPU 工作的最关键实现了，也就是之前阐述的五个部分：取指、译码、执行、写回、更新 PC。若期间发生中断或者 DMA，则在指令执行结束后进行相应的处理。

```
uint16_t CPU::Execute() { // 执行一条指令，返回执行周期数
    // 取指-> 译码-> 执行-> 更新 PC->...
    // 处理中断，DMA
    if(nmi) { nmi = false; return Interrupt(static_cast<uint16_t>(InterruptVector::NMI)); }
    if(dma) { dma = false; return OAMDMA(); }
    // 取指
    uint8_t op_code = mem[PC];
    uint16_t updated_pc = PC + optable[op_code]->bytes;
    // 译码，获取操作数
```

```
    uint16_t opd_addr = 0xFFFF; // 默认值 0xFFFF
    bool crossed_page = false, has_operand = true;
    FetchOperandAddr(optable[op_code]->addressing_mode, opd_addr, crossed_page,
    ↪  has_operand);
    // 执行，写回
    uint8_t cycle = ExeFunc(optable[op_code], this, opd_addr, updated_pc, crossed_page,
    ↪  has_operand);
    // 更新 PC
    PC = updated_pc;
    cycles += cycle;
    return cycle;
}
```

5.1.5 中断实现

当中断发生时，需要保存现场，即入栈 PC 指针、程序状态字 P，接着关中断，最后将 PC 指向中断向量，从而进入中断程序。

```
uint8_t CPU::Interrupt(uint16_t vec_addr) {
    auto PCH = uint8_t( ( (PC) >> 0x8) & 0xff),
         PCL = uint8_t((PC) & 0xff);
    Push(PCH); // 入栈保存 PC 指针
    Push(PCL);
    Push(P); // 入栈保存程序状态字 P
    P.IrqDisabled = true; // 关中断
    PC = Read16(vec_addr); // 获取中断向量
    return 7; // 中断需要 7 个周期
}
```

中断程序结束后，会调用 RTI 结束中断，这个过程会恢复现场，即出栈程序状态字 P、PC 指针，开中断。

```
OpExeFuncDefine(OP_RTI) {
    cpu->P = cpu->Pop(); // 出栈恢复程序状态字
    uint8_t PCL = cpu->Pop(), PCH = cpu->Pop(); // 出栈恢复 PC 指针
    updated_pc = (PCH << 0x8) | PCL;
    return self.cycles;
}
```

5.1.6 子程序调用

6502 CPU 可通过 JSR 指令对子程序调用，在子程序结束处调用 RTS 指令返回。整个过程和中断类似，除了开中断、保存、恢复程序状态字外，其他基本一样，这里不再重复。

5.2 PPU

5.2.1 内存类定义

PPU 的内存类和 CPU 的基本一致，核心代码如下：

```
class __PPUMem__ {
private:
    uint8_t VRAM[0x800];                // 2KB，存放 2 个名称表
    uint8_t PatternTable[2][0x1000];    // 4KB * 2，存放图块表
    uint8_t *NameTable[4][0x400];       // 1KB * 4，实际上只能用 2 块，指向 VRAM，另外两块做镜像用
    uint8_t Palette[0x20];              // 32B，调色板
    inline const uint8_t &AT(uint16_t addr) const {
        return    (addr < 0x2000 ? PatternTable[addr >> 0x0c][addr & 0xfff]:
                   addr < 0x3000 ? *NameTable[(addr >> 0x0a) & 0x03][addr & 0x3ff]:
                   addr < 0x3f00 ? AT(addr & 0x2fff) :
                   addr < 0x4000 ? Palette[addr & 0x1f]:
                   AT(addr & 0x3fff));
    }
public:
    uint8_t &operator[](uint16_t addr) { return AT(addr); }

    using iterator = MemIterator<__PPUMem__>; // 内存类的迭代器
    iterator begin() { return iterator(this, 0); }
    iterator end() { return iterator(this, 0x4000); }
};
```

5.2.2 主类定义

如下代码主要列出了 PPU 类的寄存器、内部寄存器，以及主要工作流程的方法。

```
class PPU {
private:
    uint32_t cycles; // PPU 的时钟周期数，是 cpu 的三倍
    __PPUMem__ mem;
    PPURegister PPUCTRL, PPUMASK, PPUSTATUS, // PPU 通用寄存器
```

```
                OAMADDR, OAMDATA, PPUSCROLL,
                PPUADDR, PPUDATA, OAMDMA;
    uint8_t OAM[0x100]; // 256B 的 OAM 精灵存储区
    const static uint16_t frame_width = 341, frame_height = 262; // NTSC, 60fps
    const static uint16_t screen_width = 256, screen_height = 240; // 每帧图像的大小
    constexpr static double frame_duration = 1000.0 / 60; // 每帧的时间
    double cur_time; // 稳定帧率用的计时器
    uint32_t video_buffer[frame_width * frame_height]; // 当前帧的像素数据
    // 以下均为内部寄存器、存储区
    uint16_t v, t;              // PPU 内部寄存器 v, t
    uint8_t fineX;              // 内部寄存器，8x8 图块中的 X 列
    bool w;                     // 内部寄存器，用于判断第一次、第二次读写
    bool odd_frame;             // 奇偶帧
    uint32_t frames_count = 0; // 统计帧数
    struct {
        uint8_t id;
        uint8_t Y, tileIdx, Attr, X;     // 当前行绘制的精灵的 X, Y 坐标、块号、属性
        uint8_t tileL, tileH;            // 精灵图块数据
    } secOAM[8], sprTile[8];             // 内部寄存器，保存渲染的精灵，每行最多 8 个精灵
    uint16_t bgShiftL, bgShiftH;         // 内部移位寄存器，存放当前行 2 块的背景图块像素数据
    uint8_t bgL, bgH, nt, at;            // 存放当前行背景图块的像素数据，nt 存放的是图块表中
    ↪ 的编号，at 是背景图块的调色板指针高两位
    uint8_t atShiftL, atShiftH;          // 内部移位寄存器，存放当前行 2 块的背景图块的调色板
    ↪ 指针高两位
    bool atL, atH;                       // L, H 组成当前行当前背景图块的调色板指针高两位
    uint16_t internal_load_addr;         // PPU 读取像素数据用的地址
    uint8_t PPUDATA_buffer;              // 读的数据缓冲区
    static const uint32_t palette[0x40]; // 背景、精灵调色板
    CPU *cpu;
    // 工作流程
    void step();                         // 执行一个 ppu 周期
    void pixel(unsigned x, unsigned y);  // 绘制渲染区的一个像素点
    void clear_OAM();                    // 清除 OAM`
    void eval_sprites(unsigned y);       // 获得下一行的 8 个精灵信息
    void load_sprites(unsigned y);       // 存储下一行的 8 个精灵信息
}
```

5.2.3 主要工作流程实现

PPU 绘图的实现主要调用了 step() 函数，每一个 PPU 时钟周期执行一次，根据当前不同的行数、列数进行不同的操作（取数据、计算像素值）：

```
void PPU::step() {
    // 根据当前 PPU 总周期数来计算行数、列数
    uint16_t    scanline = cycles / frame_width,
                dot = cycles % frame_width;
    if( (scanline >= 0 && scanline < 240) || scanline == 261) {
        // 精灵
```

```
        switch (dot) {
            case 1: clear_OAM(); if(scanline == 261) PPUSTATUS.S = PPUSTATUS.O = 0; break;
            case 257: eval_sprites(scanline); break;
            case 321: load_sprites(scanline); break;
        }
        // 背景
        switch (dot) {
            case 2 ... 255:
            case 322 ... 337:
                pixel(dot - 2, scanline);
                ... // 省略一些代码
            case 256: pixel(dot - 2, scanline); bgH = mem[internal_load_addr];
            ↪ v_scroll(); break; // Vertical bump.
            case 257: pixel(dot - 2, scanline); reload_shift(); h_update(); break;
            ↪ // Update horizontal position.
            case 280 ... 304: if (scanline == 261) v_update(); break; // Update
            ↪ vertical position.
            case 1: internal_load_addr = get_nt_addr(); if(scanline == 261)
            ↪ PPUSTATUS.V = 0; break;
            case 321: case 339: internal_load_addr = get_nt_addr(); break;
            case 338: nt = mem[internal_load_addr]; break;
            case 340: nt = mem[internal_load_addr]; if(scanline == 261 &&
            ↪ rendering() && odd_frame) ++cycles; break;
        }
    }
    else if(scanline == 240 && dot == 0) { // 形成完整的一帧图像
        // 绘制一帧
        SDL_UpdateTexture(texture, NULL, video_buffer, sizeof(uint32_t) * screen_width);
        SDL_RenderCopy(renderer, texture, NULL, NULL); SDL_RenderPresent(renderer);
        SDL_PollEvent(&event); // 捕捉事件（按键）
        odd_frame = !odd_frame; ++frames_count;
    } else if(scanline == 241 && dot == 1) { // 发出 VBlank 信号
        PPUSTATUS.V = 1; if(PPUCTRL.V) cpu->nmi = true;
    }
    if(++cycles >= (frame_width * frame_height)) {
        cycles %= (frame_width * frame_height); odd_frame = !odd_frame;
    }
}
```

而在渲染部分，都调用了 pixel() 函数，用于计算当前行、列的像素值：

```
    uint8_t draw_palette = 0, spr_palette = 0; // 调色板指针
    bool spr_priority = 0;
    if(y < 240 && x >= 0 && x < 256) {
        // 背景
        if (PPUMASK.b && !(!PPUMASK.m && x < 8)) {
            draw_palette = GetBit(bgShiftH, 15 - fineX) << 1 |
                           GetBit(bgShiftL, 15 - fineX);
            if (draw_palette)
                draw_palette |= (GetBit(atShiftH, 7 - fineX) << 1 |
```

```
                                GetBit(atShiftL, 7 - fineX)) << 2;
        }
        // 精灵
        if (PPUMASK.s && !(!PPUMASK.M && x < 8))
            for(int i = 7; i >= 0; --i) {
                if(sprTile[i].id == 64) continue; // 空
                unsigned sprX = x - sprTile[i].X;
                if(sprX >= 8) continue;
                if(sprTile[i].Attr & 0x40) sprX ^= 7; // 水平翻转
                uint8_t p = GetBit(sprTile[i].tileH, 7 - sprX) << 1 |
                            GetBit(sprTile[i].tileL, 7 - sprX);
                if(p == 0) continue;
                // sprite 0 的非零像素覆盖背景的非零像素
                if(sprTile[i].id == 0 && draw_palette && x != 255) PPUSTATUS.S = 1;
                p |= (sprTile[i].Attr & 0x03) << 2;
                spr_palette = p + 0x10;                // sprite 的 palette 在 0x3f10
                spr_priority = sprTile[i].Attr & 0x20; // 精灵在背景前面还是后面
            }
        if(spr_palette && (draw_palette == 0 || spr_priority == 0))
            draw_palette = spr_palette;
        video_buffer[screen_width * y + x] = palette[mem[0x3F00 + (rendering() ?
        ↪  draw_palette : 0)]]; // 存储像素点
    }
    // 更新内部移位寄存器的值
    bgShiftL <<= 1; bgShiftH <<= 1;
    atShiftL = (atShiftL << 1) | atL;
    atShiftH = (atShiftH << 1) | atH;
}
```

5.3 卡带

卡带的实现就比较简单了，其类定义如下：

```
class Cartridge {
public:
    bool LoadRomFile(CPU &cpu, PPU &ppu, const std::string & filename);
private:
    struct {                    // ROM 文件的文件头定义
        uint8_t INes[4];        // 应为 0x4e 0x45 0x53 0x1a
        uint8_t PRGRomBankCnt;  // PRG ROM 的大小（块数），单位 16K 一块
        uint8_t CHRRomBankCnt;  // CHR ROM 的大小（块数），单位 8K 一块，0 表示使用 CHR RAM
        uint8_t ROMControl[2];  // ROM 控制位
        uint8_t PRGRamBankCnt;  // PRG RAM 的大小（块数），单位 8K 一块，若为 0 则代表 1 块，8k
        uint8_t Reserved[7];    // 保留位，全零
    } header;
    std::string filename;       // ROM 文件名
};
```

LoadRomFile 方法将 ROM 文件加载进内存，主要代码如下：

```
bool Cartridge::LoadRomFile(CPU &cpu, PPU &ppu, const std::string &filename) {
    this->filename = filename;
    FILE *fp = fopen(filename.c_str(), "rb"); // 打开文件
    if(fp == NULL) {
        printf("open file "); perror(filename.c_str());
        return false;
    }
    fread((void*)&header, sizeof(header), 1, fp); // 读取头
    __CPUMem__::iterator cpuRamIt = cpu.mem.begin(); // 内存迭代器
    __PPUMem__::iterator ppuRamIt = ppu.mem.begin();
    if(! GetBit(header.ROMControl[0], 3))  // 名称表的水平、垂直映射
        GetBit(header.ROMControl[0], 0) ?  ppu.mem.setVerticalMirroring() :
        ↪  ppu.mem.setHorizontalMirroring();
    // Mapper
    switch (JointBits(GetUpperBits(header.ROMControl[1]),
                      GetUpperBits(header.ROMControl[0]))) {
        case 0: case 64: { // Mapper 0
            uint8_t PRGRomData[0x4000];
            fread((void*)PRGRomData, sizeof(PRGRomData), 1, fp);
            std::copy(PRGRomData, PRGRomData + sizeof(PRGRomData), cpuRamIt + 0x8000); // 复
            ↪  制 ROM 数据到 CPU 内存中
            if(header.PRGRomBankCnt > 1)
                fread((void*)PRGRomData, sizeof(PRGRomData), 1, fp);
            std::copy(PRGRomData, PRGRomData + sizeof(PRGRomData), cpuRamIt + 0xc000);
            uint8_t CHRRomData[0x2000];
            fread((void*)CHRRomData, sizeof(CHRRomData), 1, fp); // 复制图像数据到 CPU 内存中
            std::copy(CHRRomData, CHRRomData + sizeof(CHRRomData), ppuRamIt + 0x0000);
            break;
        }
    }
    fclose(fp);
    return true;
}
```

5.4 标准控制器

标准控制器采用移位寄存器实现，类定义如下：

```
class Joypad {
private:
    uint8_t const *keypad;
    // R, L, D, U, St, Sel, B, A
    uint8_t joypad_bits[2];                 // 手柄 1、手柄 2 移位寄存器
    bool strobe; // 控制是否读入键位
public:
    // n 为 0 或者 1，表明操作第几个手柄
```

```
    uint8_t read_joypad_status(int n = 0);
    void write_joypad_status(bool v);
    uint8_t get_joypad_status(int n = 0);  // 读取键盘上的按键状态
};
```

读、写手柄状态寄存器分别为 read_joypad_status() 和 write_joypad_status()，主要代码如下：

```
uint8_t Joypad::read_joypad_status(int n) {
    // 若 S 为高电平，则读取 A 键，由于硬件问题高第二位为 1
    if(strobe) return 0x40 | (get_joypad_status(n) & 1);
    // 每次读取一位并右移一位
    uint8_t key_status = 0x40 | (joypad_bits[n] & 1);
    joypad_bits[n] = 0x80 | (joypad_bits[n] >> 1);
    return key_status;
}
void Joypad::write_joypad_status(bool v) {
    if(strobe && !v) // 下降沿重载
        for(int i = 0; i < 2; ++i) joypad_bits[i] = get_joypad_status(i);
    strobe = v;
}
```

6 单元测试

在开发过程中，需要对某些模块例如 CPU 进行单元测试来保证程序正确性，具体做法是编写测试用的汇编指令，执行完后检查各个寄存器的值是否符合预期，下面是截取某一段单元测试片段：

```
TEST(CPUTest, opTest) {
    CPU cpu; PPU ppu;
    cpu.connectTo(ppu); ppu.connectTo(cpu);
    uint8_t &X = cpu.getX(), &Y = cpu.getY(),
            &SP = cpu.getSP(), &A = cpu.getA();
    uint16_t &PC = cpu.getPC();
    ProcessorStatus &P = cpu.getP();
    EXPECT_TRUE(cart.LoadRomFile(cpu, ppu, "./test.nes"));
    EXPECT_TRUE(cart.PrintHeader());
    cpu.Reset();
    // 测试是否正常读取指令
    EXPECT_EQ(cpu.Read8(PC), 0xf8); // SED
    // 开始执行
    EXPECT_EQ(cpu.Execute(), 2);    // sed
    EXPECT_TRUE(P.Decimal);
    EXPECT_EQ(cpu.Execute(), 2);    // cld
    EXPECT_FALSE(P.Decimal);
    A = 0xff;
    EXPECT_EQ(cpu.Execute(), 2);    // asl
    // 测试各个寄存器的值是否符合预期
    EXPECT_EQ(A, 0xfe); EXPECT_FALSE(P.Zero); EXPECT_TRUE(P.Negative); EXPECT_TRUE(P.Carry);
    cpu.Write(0x0066, 0);
    EXPECT_EQ(cpu.Execute(), 5);    // asl $66
    EXPECT_EQ(cpu.Read8(0x66), 0x0); EXPECT_TRUE(P.Zero); EXPECT_FALSE(P.Negative);
    ↪  EXPECT_FALSE(P.Carry);
    X = 1;
    cpu.Write(0x0000, 0x80);
    EXPECT_EQ(cpu.Execute(), 6);    // asl $ff,X
    EXPECT_EQ(cpu.Read8(0x66), 0x0); EXPECT_TRUE(P.Zero); EXPECT_FALSE(P.Negative);
    ↪  EXPECT_TRUE(P.Carry);
    ... // 省略余下代码
}
```

7 总结

通过实现游戏模拟器来深入理解计算机原理，是一个不错的选择。模拟器的实现，使得我在今后程序开发的道路上越走越深。

本课题完成的主要工作如下：

1. NES 使用的是 2A03 处理器，基于 6502 的小端 CPU。一共有 56 条指令集和 13 种寻址方式总共 151 个有效操作码，6 个寄存器，时钟频率 1.77MHz。需要正确无误实现每一条操作码、不同寻址模式、内存布局/IO 映像、DMA、栈帧、寄存器、中断特性、设计上的 BUG 等等。
2. NES 使用 2C02 图形处理器 PPU，时钟频率是 CPU 的 3 倍，显存 16KB，帧分辨率 341x262，可视部分分辨率 256x240，每一个时钟周期渲染一个像素点，每秒传输 60 帧。需要精确同步 CPU 与 PPU 的时钟频率，计算图形数据在内存中的定位，并高效的渲染每一帧，模拟读写寄存器产生的副作用。
3. 采用直接翻译游戏 ROM 指令的方式，读取 PC 指针的操作码进行译码，运行，写回寄存器/内存，更新 PC。
4. 跨平台开发，考虑代码的兼容性、确保高性能。
5. 进行单元测试保证正确性。

在这几个月的毕业设计过程中，本人查阅了许多资料，接触到许多新东西，对自身技能也提高了许多。自己独立完成了整个系统的设计与实现，提高了自己分析问题和解决问题的能力，对于如何调试程序也有了深刻的理解。

由于时间问题和自身经验不足，本系统有以下几个部分需要完善：

1. 实现常用的 Mapper
2. 实现 APU 音频模块

参考文献

[1] JACOBS A. 6502 Instruction Reference[EB/OL]. 2003 [2018-06-01]. http://obelisk.me.uk/6502/reference.html.

[2] BANSHAKU, TEPPLES, OTHERS. PPU Scrolling[EB/OL]. 2017 [2018-06-01]. http://wiki.nesdev.com/w/index.php/PPU_scrolling.

[3] DRAG, TEPPLES, OTHERS. PPU Rendering[EB/OL]. 2016 [2018-06-01]. http://wiki.nesdev.com/w/index.php/PPU_rendering.

[4] NDWIKI, TEPPLES, OTHERS. PPU sprite evaluation[EB/OL]. 2016 [2018-06-01]. http://wiki.nesdev.com/w/index.php/PPU_sprite_evaluation.

[5] SIEBER J. Implementing the Nintendo Entertainment System on a FPGA[D]. [S.l.]: [s.n.], 2013.

[6] KING A. FPGA NES[J], 2012.

[7] DISKIN P. Nintendo Entertainment System Documentation[J]. Tokyo: Nin-tendo, 2004.

致谢

在论文完善之际，感谢所有帮助过我的人！

感谢我的母校，由于高考志愿填报问题导致我不能如愿以偿地进入本校计算机科学与技术专业进行学习，在食品与科学工程的两年里，依旧不忘初心地自学我所爱。凭借特长在各位老师、领导的支持与厚爱下转入信息工程系，让我得以深入学习，最后从事一份自己所感兴趣的工作。

感谢我的指导老师安鑫老师，安鑫老师严谨的治学态度和精益求精的工作作风，深深地感染着我，使我终生受用。从课题的开题到论文的最终完成，安鑫老师都始终给予我细心的指导，在这期间向我提出了许多宝贵意见和建议。当我遇到困难时，都是安鑫老师给我鼓励与指引，使我能够克服重重困难。在此谨向安鑫老师致以诚挚的谢意。

感谢支持和关心我的同学们，朋友职愈博、曹鑫，在毕业设计遇到问题能够提供建议与帮助，使我顺利地完成毕业设计。

最后，感谢我的家人，感谢你们在我的学习生活中所给予的支持和理解，让我能够不断进取。没有你们，就没有我的今天，你们的支持与鼓励，永远是支撑我前进的最大动力。

作者：罗能

2018 年 6 月 5 日