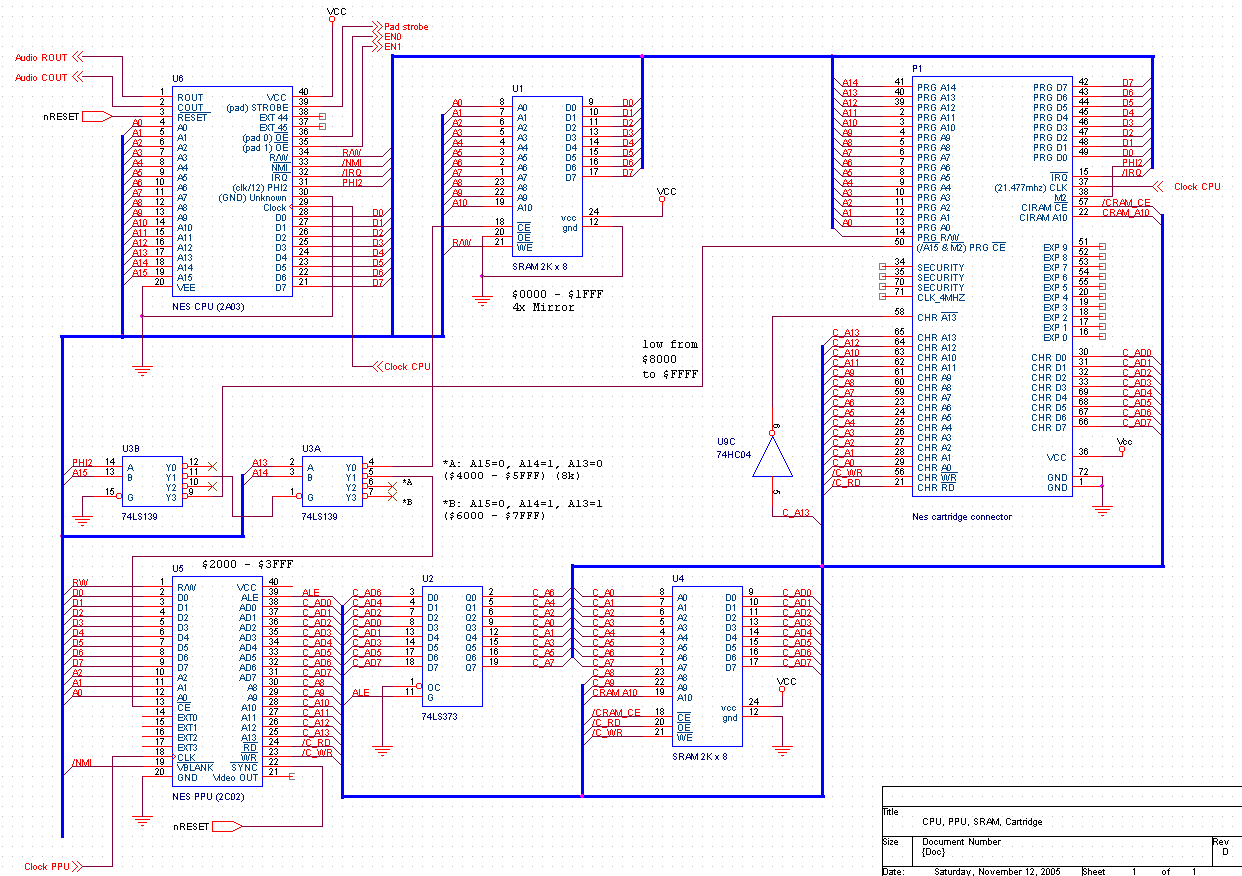
# NES模拟器开发

使用python开发一个nes模拟器

## 1、系统结构

### 1.1 系统原理图

如下图所示，左上角cpu，通过cpu总线连接到ppu、内存、卡带，ppu通过ppu总线连接到vram与卡带。



运行过程如下：

1、Cpu与ppu使用同一个振荡器的时钟信号，分频倍率不同，ppu运行3个周期cpu运行1周期。

2、Cpu不断执行0x8000到0xffff之间的指令，结果写入ram，同时ppu不断执行绘图指令，将vram中的内容显示在屏幕上。

3、ppu画完一帧241条可见扫描线之后触发vblank，使cpu进入nmi中断，约持续20条扫描线，在此期间cpu通过访问ppu寄存器向vram写入数据。

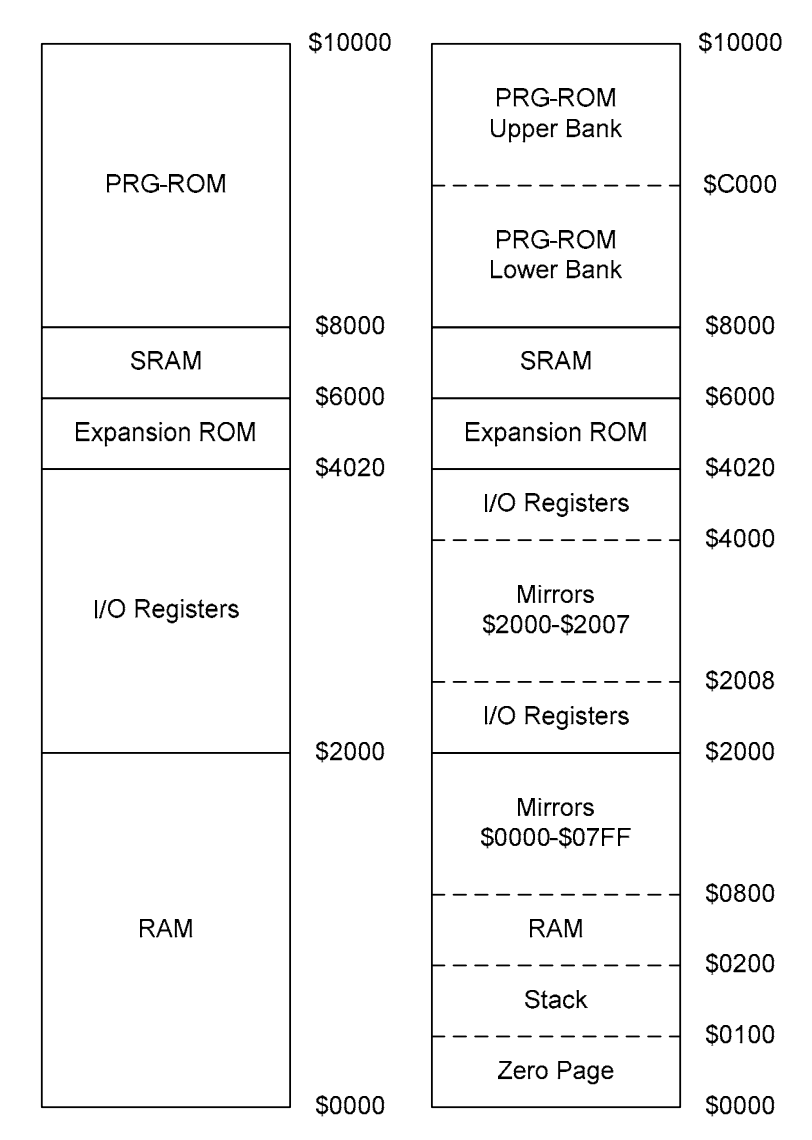
## 2、6502cpu

### 2.1 cpu型号

Nes使用的cpu是 [2A03](https://links.jianshu.com/go?to=http%3A%2F%2Fwiki.nesdev.com%2Fw%2Findex.php%2F2A03)，8bit，工作频率 1.7897725 MHz。它有 16bit 地址总线，所以它的寻址范围为 0x0000 - 0xFFFF，即 64KB。CPU 支持三种中断：RESET（复位），NMI（不可屏蔽中断），IRQ（可屏蔽中断）

### 2.2 cpu内存映射

Cpu与其他设备的交互全部以读写内存的方式实现，内存映射表如下：



 **0x0000 - 0x0800** ( RAM )  
这是主机中 2KB RAM 的数据，分成了 3 块

* **0x0000 - 0x00FF** ( Zero page )  
  前 256 字节划分为 Zero page，这块内存相比其他区域不同点在于能让 CPU 以更快的速度访问，所以需要频繁读写的数据会优先放入此区域
* **0x0100 - 0x01FF** ( Stack )  
  这一块区域用于栈数据的存储，SP（栈指针） 从 0x1FF 处向下增长
* **0x0200 - 0x07FF** ( 剩余 RAM )  
  这是 2KB 被 Zero page 和 Sack 瓜分后剩余的区域

 **0x0800 - 0x2000** ( Mirrors )  
你可能会感觉到奇怪这个 Mirror 到底是干什么的。实际上它是 0x0000 - 0x07FF 数据的镜像，总共重复 3 次  
例如：0x0001, 0x0801, 0x1001, 0x1801 都指向了同样的数据，用程序来解释的话，就是：  
address &= 0x07FF  
对应到硬件上的话，就是 bit11 - 13 的线不接  
至于为什么任天堂要这样设计？我猜可能是考虑到成本原因，2KB RAM 够用了，不需要更大的 RAM，但是地址空间得用完啊，所以才有了 Mirror 效果

 **0x2000 - 0x401F** ( IO Registers )  
这里包含了部分外设的数据，包括 PPU，APU，输入设备的寄存器。比如 CPU 如果想读写 VRAM 的数据，就得靠 PPU 寄存器作为中介

 **0x4020 - 0x5FFF** ( Expansion ROM )  
Nesdev 的论坛上有篇解释这块区域的[帖子](https://links.jianshu.com/go?to=https%3A%2F%2Fforums.nesdev.com%2Fviewtopic.php%3Ff%3D3%26t%3D12936)，简单来讲，该区域用于一些 Mapper 扩展用，大部分情况用不到

 **0x6000 - 0x7FFF** ( SRAM )  
这就是之前说过的带电池的 RAM 了，该区域位于卡带上

 **0x8000 - 0xFFFF** ( Program ROM )  
这里对应了程序的数据，一般 CPU 就在这块区域中执行指令，该区域位于卡带上

#### 2.2.1 mirror效果

什么是mirror效果？mirror效果是模拟器对硬件结构的模拟。

从内存映射表可以看到，cpu给ram预留8k空间，但实际上ram只有2k，在接线时a0-a10就能表示这2k了，a11-a13空接，也就是说无论a11-a13是1还是0，读取或写入的都是内存中的同一个位置，这是由硬件决定的，nes上的游戏有可能利用这一特点。而我们模拟时是用一大块内存，这时就要通过mirror效果来取得与硬件一致的效果。

举例来说，nes游戏如果存取0x700与0x1700，那么实际在硬件上是同一个位置，某个游戏可能写入0x1700，读取0x700，这时0x700的内容是被改变了的，游戏获取到了正确的值。而我们的模拟器中如果不做相应处理，那么写入0x1700就是存到0x1700,0x700的值是不会改变的，那么游戏获取的就是改变前的值，是错误的。

## 3、ppu

PPU 型号为 2C02（好像与6538功能一致），用于产生图像。

NES 中图像分两种：

* Background（背景）

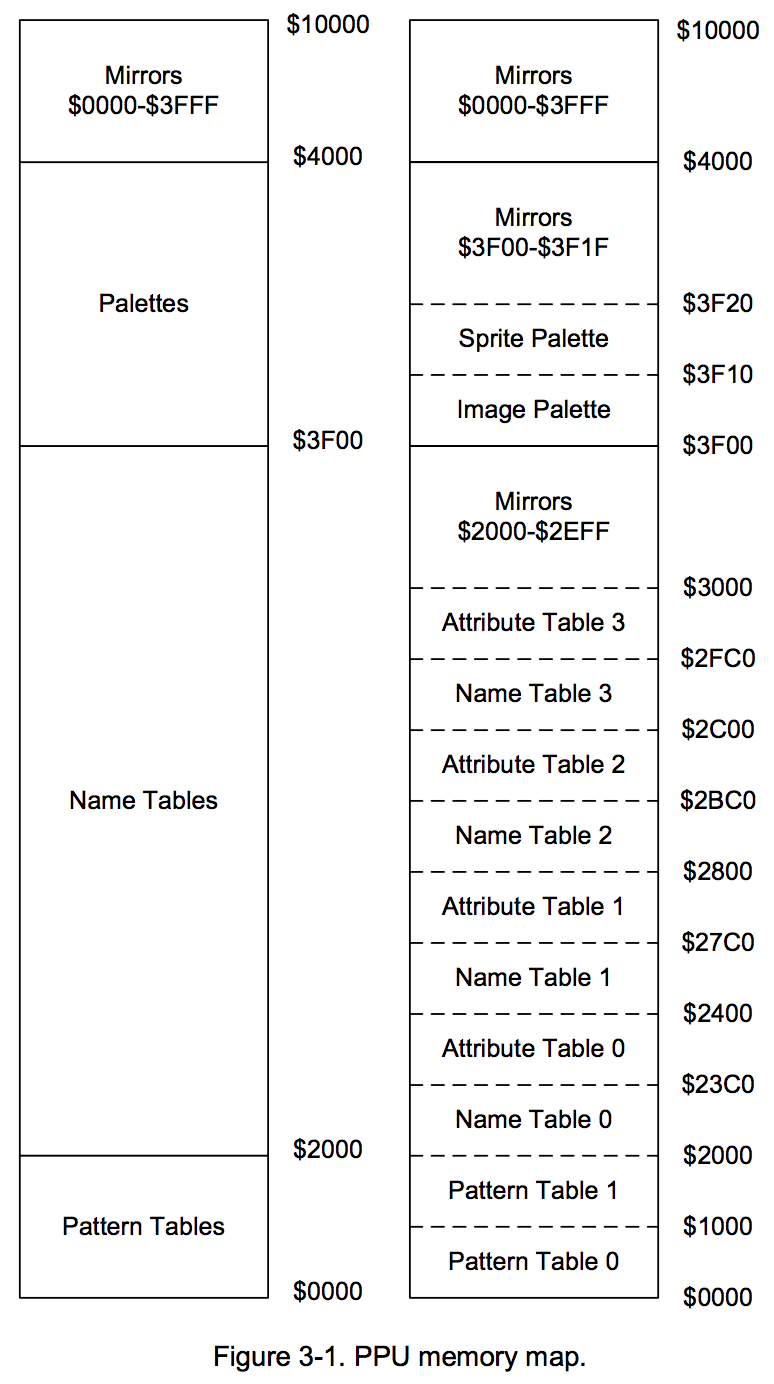
顾名思义，用于背景的显示，比如游戏的天空，草地，建筑

* Sprite（精灵）

用于前景的显示，例如游戏里的人物，子弹等

最终上述两种图像组合后，输出到屏幕

### 3.1 ppu内存映射



### 3.2 图像存储表

Nes的输出分辨率为256\*240。也就是有60k个像素，如果每个像素使用1字节（256色）存储，那么每幅画面就需要60k内存存储，当时没有这么多内存。

那么nes是如何节省内存的呢。

#### 3.2.1 命名表（Name Table）

Nes不存储每个像素，它将256\*240的画面划分为32\*30个8\*8的Tile，每个Tile使用一个字节存储索引，一共需要32\*30 = 960个字节。

每

个

格

子

存

一

个

编

号

0

31

0

29

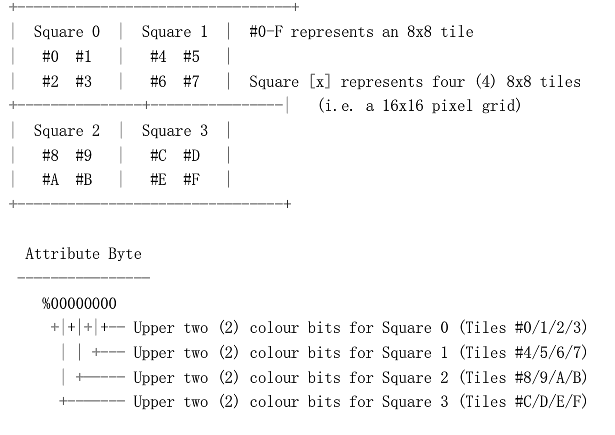
改变命名表的内容就能改变屏幕显示。

#### 3.2.2 属性表（Attribute Table）

每个命名表都有一个属性表与它搭配。

它保存了每个Tile的颜色高两位信息，也就是说一个tile中64个像素点颜色高两位信息都是一样的，每个字节可以存储4\*4个Tile信息，一共需要40个字节，任天堂分配了64个字节给属性表。

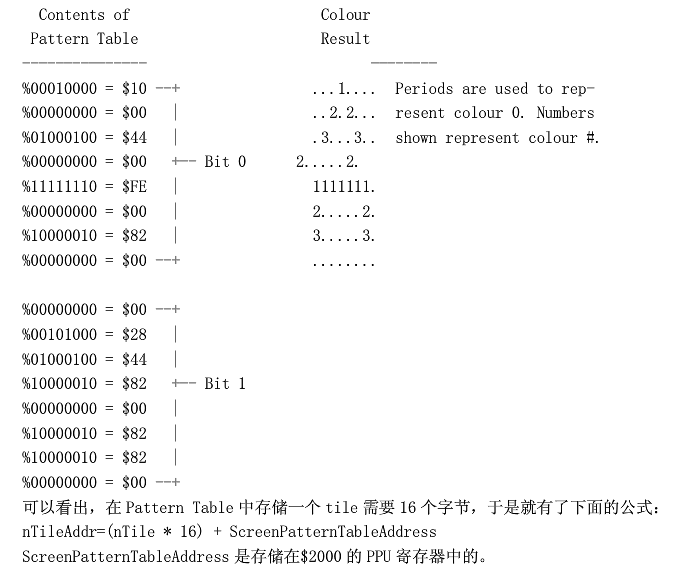
具体存储方法如下图：



这样1个命名表和它搭配的属性表占用1k内存。Nes中共有2k Vram，可以存储两套命名表和属性表。

#### 3.2.3 样式表（Pattern Table）

样式表保存了8\*8 Tile的颜色低两位信息，每个像素占用两个字节有4种状态，每个Tile占用16个字节：



一个pattern table一共4k，可以存放256个tile信息。

通过修改0x2000寄存器可以选择不同的pattern table。