数字电路与数字系统实验

2019 秋学期

FC（NES）红白机

181870243 杨超

概述

在数电实验课程的最开始几周，我就在想最后大实验要做什么，目标是难度高趣味性也高的选题，随后有了一个疯狂的想法：实现一个可以玩游戏的 FC（NES）红白机。

NES 的 CPU 基于 MOS 6502 处理器，除 CPU 外还有音频处理器（APU，Audio Processing Unit）、图形处理器（PPU，Picture Processing Unit），以及游戏大小超出 CPU 寻址空间时用到的 memory mappers.

对于 mapper，在查找资料过程中发现了一句这样的评价：
Playing SMB was as far as we took the original project. At the time, we weren't terribly anxious to implement a whole slew of mappers, as ***that could have easily been a far larger task than the whole rest of the project*** *(having now done so, I can say that it absolutely is [more difficult]!).* (https://danstrother.com/fpga-nes/)

因此在本实验中，mapper 自始至终都没有涉及。

在调研的最后，我找到了 NesDev 社区，里面有相关各个方面的详细资料。
感谢 NesDev.com!!!

由于此时仍在学期初，数电实验以及 ICS 课程都还刚起步，缺乏基础便没有继续深入。直到 11 月底做完实验十一后开启大实验时才开始着手阅读资料和实现。

本实验由我单人完成。CPU 利用 NesDev 上的测试程序顺利地完成了 debug，而 PPU 的 debug 耗费了较长时间以至于延迟到 1 月 2 日验收，本不打算实现 APU 但延迟了有时间多就用了两三天写了个不完全体 APU。

这个报告写得不是很好，抱歉。时间比较紧，内容繁多我有点不知道写什么来突出重点。

CPU

NES 的 CPU 为 Ricoh 生产的定制化 MOS 6502 处理器，增加了 DMA，移除了 BCD 十进制加减法模式。NES 的 CPU 与 APU 被放在一块 RP2A03/RP2A07 芯片上。

CPU 部分设计实现参考《计算机组成与系统结构》，（袁春风等），单/多周期处理器设计

MOS 6502 处理器简介

详细内容参考 *MCS6500 Microcomputer Family Programming Manual*

6502 是来自上世纪 70 年代的 8 位变长指令处理器（使用 16 位内存地址）
用 verilog，资源丰富的开发板来实现这一款老处理器并没有那么难（但还是难）

寄存器

Accumulator	8-bit	最主要的寄存器，多数算术指令只用于 Accumulator
Index X / Y	8-bit	这两个寄存器用于带偏移量寻址，通常存循环计数变量
Status Register	8-bit	状态寄存器
Program Counter	16-bit	PC
Stack Pointer	8-bit	栈区为 0100H-01FFH，地址高 8 位固定为 01H，因此只存低 8 位

状态寄存器

Bit	7	6	5	4	3	2	1	0
	N	V	-	B	D	I	Z	C

N：运算结果最高位为 1 时设置
V：溢出，在 ADC/SBC 指令中结果溢出时被设置，BIT 指令也会设置该标志位
B：由 BRK 指令设置
D：十进制模式状态位，NES 中不使用，程序初始化时用 CLD 指令将该位置 0
I： 中断使能，置 1 时屏蔽 IRQ 中断
Z：运算结果等于 0 时被设置
C：Carry Flag，在 ADC，SBC，CMP 以及移位指令中被设置

寻址方式

Name	Abbr	
Immediate	#i	立即数，OP CODE 后一个字节即为操作数
Absolute	a	OP CODE 后的两个字节为 16 位绝对地址
Zero Page	d	地址高 8 位为 0，OP CODE 后的一个字节为地址低 8 位
Indexed	a/d, x/y	相对于绝对寻址和 Zero Page，加上 X/Y 寄存器的值作为偏移量
Indirect	(a)	OP CODE 后两个字节为 16-bit 间接地址，间接地址指向的即是最终有效地址（16 位绝对地址）
Indexed Indirect	(d, x)	间接地址高 8 位为 0，OP CODE 后的一个字节加上 X 作为间接地址低 8 位；间接地址指向有效地址（16 位绝对地址）
Indirect Indexed	(d), y	间接地址高 8 位为 0，OP CODE 后一个字节为间接地址低 8 位，由间接地址指出有效地址的基址（16 位），基址加上 Y 作为最终有效地址
Relative		PC + 偏移量（带符号）=> PC，仅用于分支跳转（Branch）指令

Zero Page：地址为 0000H-00FFH 的内存空间。一个页为 256 字节，地址高 8 位为页号，低 8 位为页内偏移量，页只是地址划分的一个称呼，无其他含义。

wrap-around

在 6502 中，为了节省资源，寻址过程中地址的计算应该是通过 ALU 完成的。地址低 8 位加上偏移量（偏移量来自 X/Y 寄存器，或者 Indirect 寻址中读取第二个字节时所加的 1），有可能产生进位，在 Zero Page 方式下（任何的默认地址高 8 位为 0），忽略进位；而在绝对寻址中，发生进位时需要额外的一个 CPU 周期来将进位加到地址高 8 位上。

实现中，为地址的计算分配了一个 16 位加法器而不是借助 ALU，通过控制信号选择是否传递进位，这样一来就无需用额外的一个 CPU 周期来处理进位了。

指令集

传输指令

指令	简述
LDA/LDX/LDY	Load Accumulator/Index X/Index Y with Memory
STA/STX/STY	Store A/X/Y in Memory
TAX/TXA/TAY/TYA	Transfer between A and X/Y
TXS/TSX	Transfer between Index X and Stack Pointer

算术指令

指令	简述
ADC	Add Memory with Carry to Accumulator
SBC	Substract Memory from Accumulator with Borrow
AND	"AND" Memory with Accumulator
ORA	"OR" Memory with Accumulator
EOR	"Exclusive OR" Memory with Accumulator
LSR	Logic Shift Right(Accumulator/Memory)
ASL	Arithmetuc Shift Right(Accumulator/Memory)
ROL	Rotate Left(Accumulator/Memory)
ROR	Rotate Right(Accumulator/Memory)
INC/INX/INY	Increment Memory/X/Y by One
DEC/DEX/DEY	Decrement Memory/X/Y by One

标志位

指令	简述
SEC	Set Carry Flag
CLC	Clear Carry Flag
SEI	Set Interrupt Diable Flag
CLI	Clear Interrupt Diable Flag
SED	Set Decimal Mode Flag
CLD	Clear Decimal Mode Flag
CLV	Clear Overflow Flag

分支跳转

指令	简述
JMP	Absolute/Indirect Jump
BMI	Branch on Result Minus(Negative Flag Set)
BPL	Branch on Result Plus(Negative Flag Clear)

BCC	Branch on Carry Clear
BCS	Branch on Carry Set
BEQ	Branch on Zero Flag Set
BNE	Branch on Zero Flag Clear
BVS	Branch on Overflow Set
BVC	Branch on Overflow Clear

Test

指令	简述
CMP/CPX/CPY	Compare Memory with A/X/Y
BIT	Test Bits in Memory with Accumulator

过程

指令	简述
JSR	Jump to Subroutine
RTS	Return from Subroutine
RTI	Return from Interrupt
PHA	Push Accumulator on Stack
PLA	Pull Accumulator from Stack
PHP	Push Status Register on Stack
PLP	Pull Status Register from Stack
BRK	Force Break

其中 BRK 指令在 NES 中虽然被保留了下来，但绝大多数游戏都不会用到，因此没有实现该指令。

JSR 指令，OP CODE 后跟着子过程的 16 位入口地址，跳转前 PC 入栈，注意放入栈中的 PC 指向的是 JSR 指令的第 3 个字节，等于后一条指令首地址减 1。

RTS 是 JSR 的逆过程，从栈中取出 PC 并置空一个周期从而让 PC 加 1 指向 JSR 后的指令。

熟悉但又不同的...

加减法

加减法指令只有 ADC（add with carry），SBC（sub with borrow）而没有单纯的 ADD 和 SUB。
带进位加法和带借位减法是方便多字节数的加减法（记住 6502 是个 8 位处理器）。
比如要计算两个 16 位无符号数的和（差）：

- 执行 CLC 将 CF 置 0（SEC 将 CF 置 1）
- ADC 低 8 位相加（SBC 低 8 位相减）
- ADC 高 8 位相加（SBC 高 8 位相减）

其中省略了变量的存取指令

另外注意到，6502 中减法对 CF 的设置与我们所熟知的 IA-32 相反。当无符号减法溢出时，IA-32 的 CF 为 1，而 6502 的 CF 为 0 表示产生借位。
从实现上看，减法是利用加法器实现的，6502 的 CF 就是加法器中最高位的的进位。

加法：$\{oCF, oResult\} = A + B + iCF$

减法：$\{oCF, oResult\} = A + \sim B + iCF$

其中'o'前缀代表 ALU 输出，'i'前缀代表 ALU 输入

栈 Push/Pop

在我们熟悉的 IA-32 中：

Push: $ESP <= ESP - 4;\ (ESP) <= SRC;$

Pop: $DEST <= (ESP);\ ESP <= ESP + 4;$

而在 6502 中：

Push: $(SP) <= SRC;\quad SP <= SP - 1;$

Pop: $SP <= SP + 1;\ DEST <= (SP);$

注意两者的栈指针加减、操作数取存，顺序是相反的

其他

CMP 指令不会设置 VF（overflow）

指令编码的规律

参考 http://wiki.nesdev.com/w/index.php/CPU_unofficial_opcodes（注意网页中字体加粗的指令为 unofficial opcode，不需要实现，official opcode 的范畴参照 Programming Manual Appendix D）

将 OP CODE 的高 3 位作为行号，低 5 位作为列号，并对列进行重排：

	0	4	8	+0C	10	14	18	+1C	1	5	9	+0D	11	15	19	+1D
0			PHP		BPL *+d		CLC		ORA (d,x)	ORA d	ORA #i	ORA a	ORA (d),y	ORA d,x	ORA a,y	ORA a,x
20	JSR a	BIT d	PLP	BIT a	BMI *+d		SEC		AND (d,x)	AND d	AND #i	AND a	AND (d),y	AND d,x	AND a,y	AND a,x
40	RTI		PHA	JMP a	BVC *+d		CLI		EOR (d,x)	EOR d	EOR #i	EOR a	EOR (d),y	EOR d,x	EOR a,y	EOR a,x
60	RTS		PLA	JMP (a)	BVS *+d		SEI		ADC (d,x)	ADC d	ADC #i	ADC a	ADC (d),y	ADC d,x	ADC a,y	ADC a,x
80		STY d	DEY	STY a	BCC *+d	STY d,x	TYA		STA (d,x)	STA d		STA a	STA (d),y	STA d,x	STA a,y	STA a,x
A0	LDY #i	LDY d	TAY	LDY a	BCS *+d	LDY d,x	CLV	LDY a,x	LDA (d,x)	LDA d	LDA #i	LDA a	LDA (d),y	LDA d,x	LDA a,y	LDA a,x
C0	CPY #i	CPY d	INY	CPY a	BNE *+d		CLD		CMP (d,x)	CMP d	CMP #i	CMP a	CMP (d),y	CMP d,x	CMP a,y	CMP a,x
E0	CPX #i	CPX d	INX	CPX a	BEQ *+d		SED		SBC (d,x)	SBC d	SBC #i	SBC a	SBC (d),y	SBC d,x	SBC a,y	SBC a,x

	2	6	+0A	+0E	12	16	+1A	+1E	3	7	+0B	+0F	13	17	+1B	+1F
0		ASL d	ASL	ASL a		ASL d,x		ASL a,x								
20		ROL d	ROL	ROL a		ROL d,x		ROL a,x								
40		LSR d	LSR	LSR a		LSR d,x		LSR a,x								
60		ROR d	ROR	ROR a		ROR d,x		ROR a,x								
80		STX d	TXA	STX a		STX d,y	TXS									
A0	LDX #i	LDX d	TAX	LDX a		LDX d,y	TSX	LDX a,y								
C0		DEC d	DEX	DEC a		DEC d,x		DEC a,x								
E0		INC d		INC a		INC d,x		INC a,x								

由绿色的算术指令块便可明显看出规律，OP CODE 中间 3 位对应寻址模式，其他 5 位确定指令类型。在实现中寻址模式就是依此来编码的。

有兴趣的也可以考虑以 ROM 的方式解析寻址模式。

中断

在 6502 中，中断有 3 种形式：NMI，RESET，IRQ
其中断向量（中断服务程序入口地址）分别位于 FFFA-FFFB，FFFC-FFFD，FFFE-FFFF

进入中断时，将 PC、状态寄存器依次入栈，随后取出入口地址并跳转。
指令 RTI 从中断返回，恢复状态寄存器、PC，继续执行指令。

6502 中，启动/重启（RESET）被当做中断来处理，此时 PC、SP 没有确定的值，入栈行为无意义。RESET 的中断向量也就是整个程序的入口地址（第一条指令的地址）。

CPU 对中断信号/NMI 是边沿敏感的，需要进行同步处理来判断其下降沿。
由于本实验没有实现 APU 的 DMC，更没实现 mapper，IRQ 没有了来源也就置 1 了。

I/O

6502 采用内存映射形式的 I/O

NES 内存空间分配

Address range	**Device**
$0000-$07FF	2KB internal RAM
$0800-$0FFF	Mirrors of $0000-$07FF
$1000-$17FF	
$1800-$1FFF	
$2000-$2007	NES PPU registers
$2008-$3FFF	Mirrors of $2000-2007 (repeats every 8 bytes)
$4000-$4017	NES APU and I/O registers
$4018-$401F	APU and I/O functionality that is normally disabled.
$4020-$FFFF	Cartridge space: PRG ROM, PRG RAM, and mapper registers

实现中为了方便直接使用了一块 64KB 的 RAM

实现

组合逻辑 ALU

输入：
四位控制信号 Ctrl
两个 8 位操作数 A，B
状态寄存器中的 CF，VF

输出：
8 位计算结果 Result
标志位 CF，VF，ZF，NF

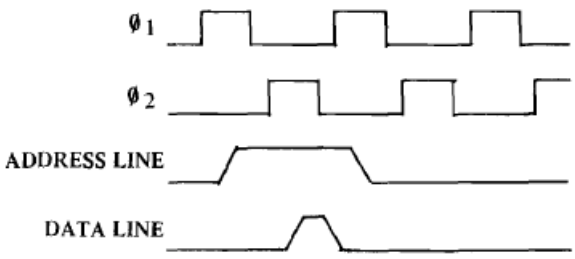
```
module ALU (
    input  [3:0] iCtrl,
    input  [7:0] iA,
    input  [7:0] iB,
    input        iCF,
    input        iVF,

    output [7:0] oResult,
    output  reg  oCF,
    output  reg  oVF,
    output       oNF,
    output       oZF
);
```

指令 ADC，SBC 需要 CF 参与运算，因此输入中有 CF;
此外，只有部分指令会设置 CF/VF，而在 CPU 模块的设计中，凡是会设置标志位的指令，都会从 ALU 输出更新标志寄存器中的 CF、VF 、ZF、NF，当一个算术指令不设置 CF/VF 时，ALU 输出的 CF/VF 等于输入的 CF/VF。

除了算术指令，ALU 还覆盖了传输指令，其中 Result 等于 A 或 B，并设置 ZF、NF，这样就统一了算术运算指令和数据传输指令。

CPU

CPU 有两个频率相同相位不同的时钟，CLK_P2 用于内存读写，CLK_P1 用于其他逻辑。



Example of Timing - MCS650X Family

FIGURE 5.2

输出的内存写入值 `MEM_WR_DATA` 直接来自于 ALU 的输出，这也是 ALU 中添加直接传输功能的原因。读出的 8 位内存值 `MEM_Q` 可能要到下一个周期才会使用（比如读取 16 位地址时），因此设置了寄存器 `MEM_DATA_BUF` 来保存上一个周期的 `MEM_Q`

PC 控制信号

```
1   always @ (posedge CLK_P1) begin
2       if (!RESET_L) //not a must
3           REG_PC <= 16'h8000;
4       else if (pc_wren)
5           REG_PC <= { MEM_Q, JMP_PCL };
6       else if (pc_branch)
7           REG_PC <= REG_PC + { {8{MEM_Q[7]}}, MEM_Q } + 1;
8       else if (state == `STATE_DECODE && (NMI_reg || (!IRQ_L && !REG_STATUS[`IF])))
9           REG_PC <= REG_PC;
10      else if (pc_inc)
11          REG_PC <= REG_PC + 1;
12      else
13          REG_PC <= REG_PC;
14  end
```

其中第 5 行，开始的想法是 `REG_PC <= { MEM_Q, MEM_DATA_BUF };`
后来意识到 JSR 取出跳转地址低 8 位后，先将当前 PC 入栈再读取跳转地址的高 8 位，那么这样写就是错误的，于是另外设置寄存器 `JMP_PCL` 记录跳转地址的低 8 位。

控制信号 `pc_branch` 用于 Branch 类指令，在解析分支指令中要做的就是根据当前状态寄存器选择性地将 `pc_branch` 置 1

`pc_inc` 控制 PC 是否在时钟上升沿时+1.
中断要在一条指令执行结束后处理，也是在下一条指令的 DECODE 阶段中处理，而此时 `pc_inc` 被置 1，PC 将指向还未执行的指令的下一字节。为了解决这一情况有了 8-9 行。

操作数地址计算

寄存器与控制信号

```
reg [7:0] MAR_H, MAR_L; //基址
reg [7:0] MEM_ADDR_OFFSET; //等于REG_X或REG_Y
reg addr_wrap;  //地址低8位计算的进位是否传递给高8位
reg addr_plus1; //用于读取下一个字节
wire [8:0] mem_addr_adder;
wire [15:0] OPERAND_ADDR;

assign mem_addr_adder = MAR_L + MEM_ADDR_OFFSET + addr_plus1;
assign OPERAND_ADDR[15:8] = MAR_H + ((~addr_wrap) & mem_addr_adder[8]);
```

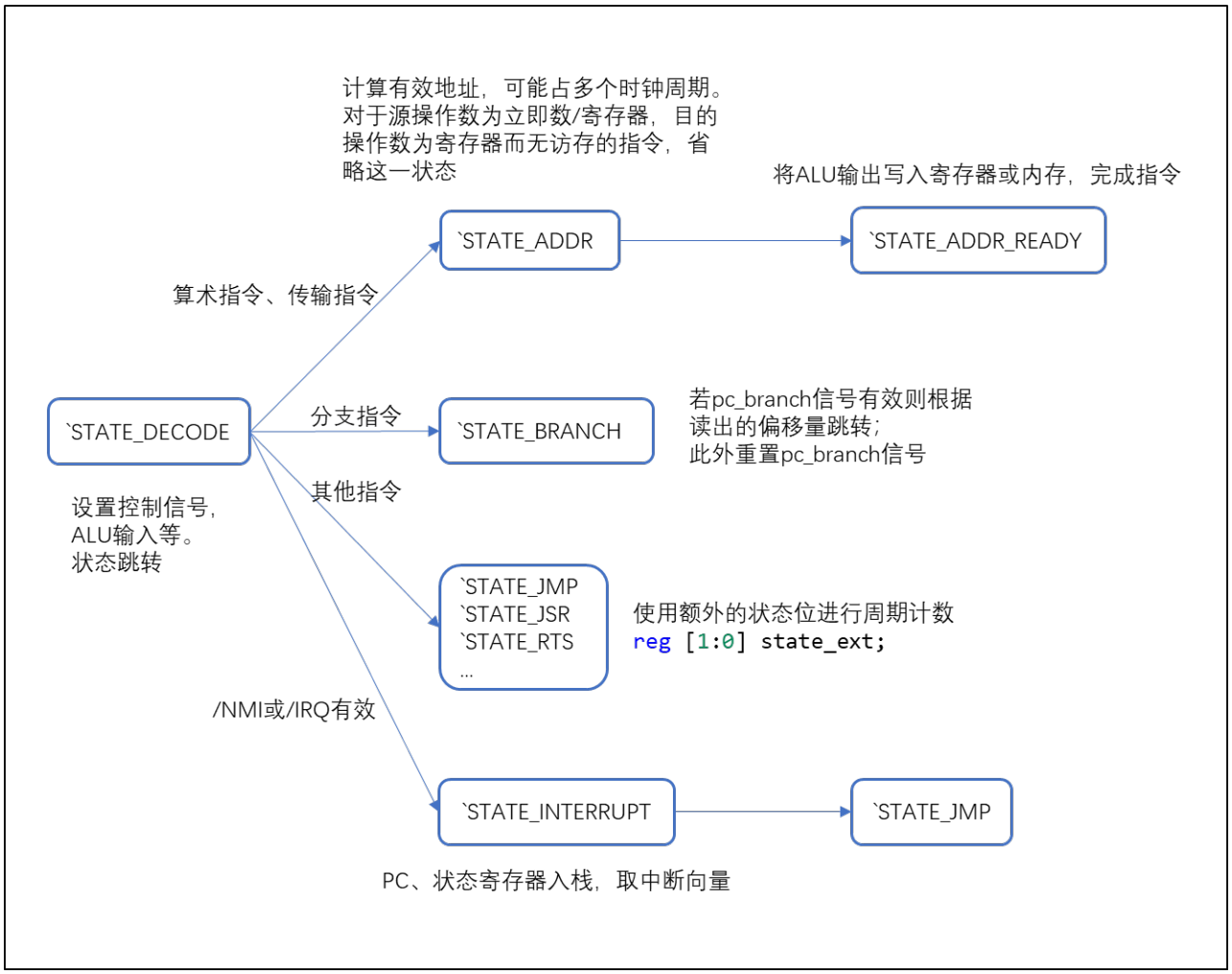
内存地址的选择

`MEM_ADDR` 为输出的访存地址，使用控制信号 `addr_sel`

```
always @ (*) begin
    case(addr_sel)
    2'b00: MEM_ADDR <= REG_PC; //PC
    2'b01: MEM_ADDR <= OPERAND_ADDR; //操作数地址
    2'b10: MEM_ADDR <= { 8'h01, REG_SP }; //栈地址
    2'b11: MEM_ADDR <= { 8'hFF, INTERRPUT_ADDR } + addr_plus1; //中断向量地址
    endcase
end
```

指令执行

通过状态机来实现，主要状态跳转如下：



设置/清除标志位的指令如 CLC、SEC，用一个周期完成，因此未出现在上图

对于算术指令、传输指令，有控制信号如下：
这些信号在`STATE_DECODE 中被设置，在`STATE_ADDR_READY 中被用来完成写回操作

```
/*  which register to write to.
    '00':accumulator,
    '01':x index,
    '10':y index,
    '11':stack pointer
*/  reg [1:0] reg_sel;
    reg write_reg;   //1: write ALU result to register selected by reg_sel
    reg write_mem;   //1: write ALU result to Memory
    reg write_pflag; //1: update CF,VF,ZF,NF from ALU output
    reg read_modify_write;
    reg [3:0] addr_mode;
```

read-modify-write 指令：从内存中读取操作数，运算后写回内存。在计算出操作数地址后进入`STATE_ADDR_READY 的第一个周期将操作数放到 ALU 输入并清除控制信号 `read_modify_write`，第二个周期和其他指令行为一致。

addr_mode：在`STATE_ADDR 中根据在`STATE_DECODE 中设置的寻址模式来计算有效地址。部分寻址模式需要多个周期来完成，而寻址的每个周期都有分配相应编码。

在建立好这样一个框架后，要做的就是根据指令的具体行为对控制信号、各寄存器进行操作，没什么特别的东西。在代码中用了大量的 Ctrl+C/V，虽然不是好的编程风格，但确实粗暴方便。

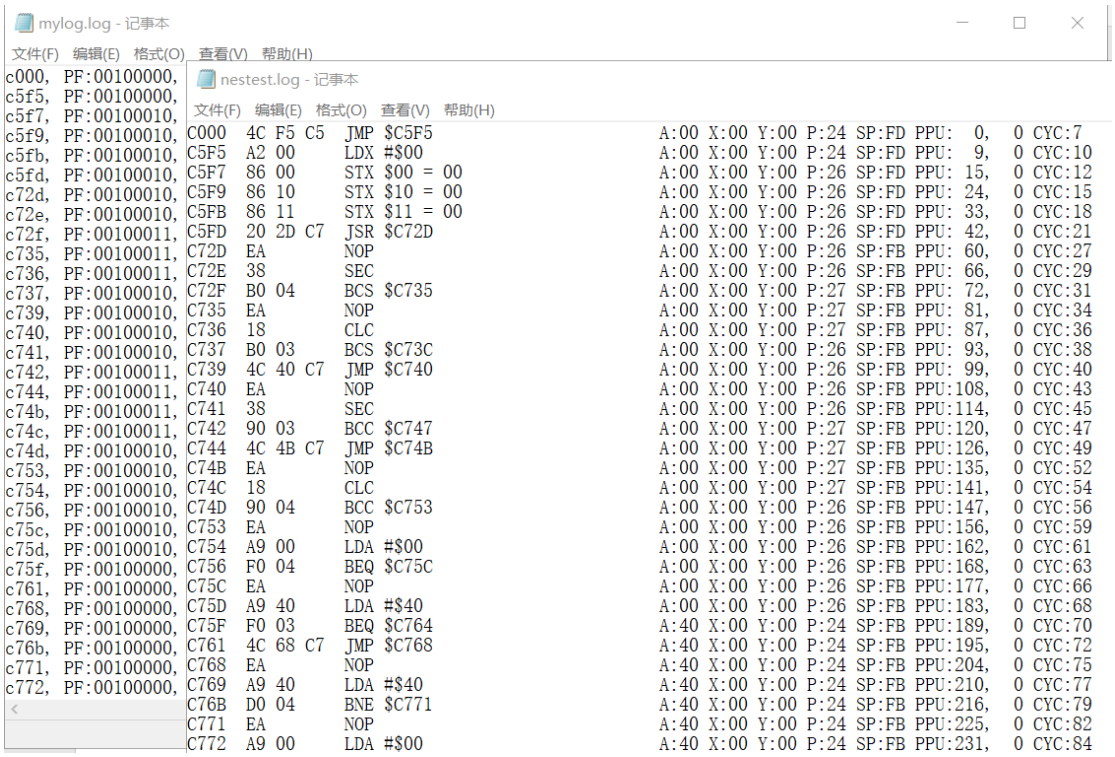
测试 Debug

在 NesDev 上列举出了一些测试程序（http://wiki.nesdev.com/w/index.php/Emulator_tests）实验中选用了 nestest，并根据 Nintendulator 模拟器产生的黄金版本的指令执行记录 log 来进行比对。

由于此时只完成了 CPU，还无法产生中断信号/NMI 或/IRQ，因而在此未对中断进行测试。

程序的入口地址为 0xC000 而不是由 RESET 中断向量（位于 0xFFFC-0xFFFD）指出的地址，在之后有了 PPU 和控制器（键盘模拟的手柄）后便从中断向量进入，在可视化的交互下进行测试。

测序程序中使用了大量的分支指令，所以对比指令的执行记录（指令首地址记录）即可。



仿真程序输出的 mylog.log 与黄金版本 nestest.log 对比

在 Nintendulator 模拟器中状态寄存器未使用的第 5 位始终置 1；而在我的实现中，如果栈顶字节的第 5 位为 0，那么该位会在 PLP 指令的作用下被置 0，由此产生区别

通过仿真测试，找到并修复了几个 bug，最后一共执行了 5000 条指令而保持一致，往后是对 unofficial opcode 的测试，无需理会。（在实现中所有的非 official opcode 都当做 NOP）

在开发板上的测试也是一样的方法，其中遇到的问题是 SW 忘消抖，启动时序安排错误等。

DMA

以上完成了一个缺少十进制模式（Decimal Mode）的 MOS 6502 处理器，而前面提到，NES 在 6502 上加了 DMA 功能，而这点我在 PPU 中看到 DMA 后才发现。

CPU 写$4014 会开启 DMA 数据传输，此时 CPU 的内存被占用，CPU 暂停工作。

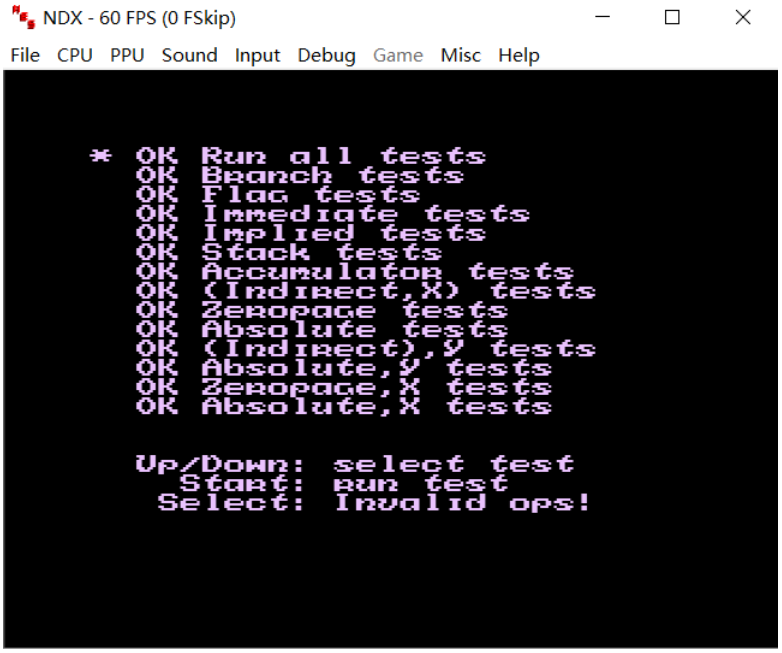
于是 CPU 增加了一个 `PAUSE` 输入信号，为 1 时 CPU 暂停。

实现其他模块后运行测试程序 nestest

可以通过控制器（键盘）来选择执行测试，左侧显示 OK 代表通过测试，卡死或显示错误代码则未通过测试。

按理经过了之前的测试，应该是全 OK 的，但（Indirect, X）tests 却显示了 A2 错误代码，在测试程序的说明文档中也没能找到(Indirect, X)错误码为 A2 的解释。之后通过 Nintendulator 模拟器设置指定 OP CODE 的断点来查找几个游戏中的（Indirect, X）指令，结果是断点从未触发，应该是没有使用（Indirect, X）寻址的指令。

肯定是哪里出了问题，但既然几个游戏都能正常运行也就没有深入找 bug 了。



NintendulatorDX 模拟器上运行 nestest（忘记保存自己的照片了）

PPU

在之前 CPU 的实现中，由于 ICS 课程学习了一些相关的概念和知识，并且找到了非常好的参考资料 *MCS6500 Microcomputer Family Programming Manual*，同时有方便利用的测试程序，总体上是顺利的。而 PPU 光理解其工作机制就花了不少时间，测试 debug 也变得相当不方便，因此花费了更多的时间。

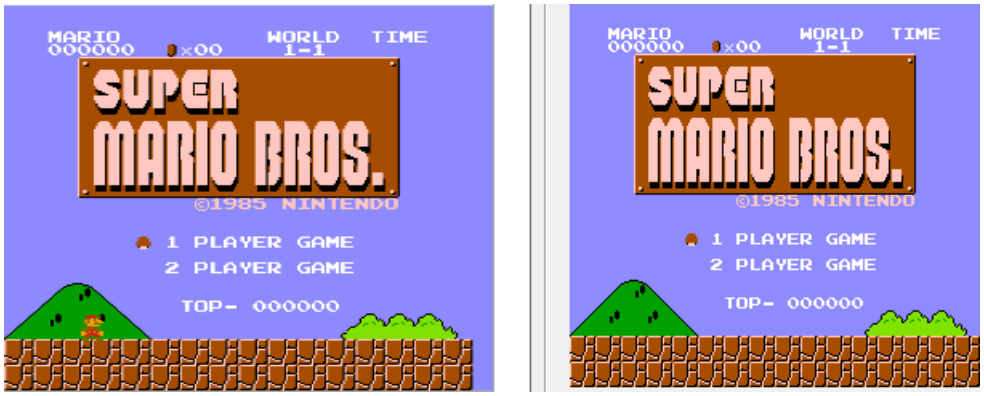
PPU 的功能是提供视频输出（分辨率为 256x240），CPU 通过相关端口可读取 PPU 的状态、写控制寄存器、写显示内容。根据视频输出模式的不同，NES 分为 NTSC NES 和 PAL NES，而本实验可以说是实现了一个 VGA NES.

NMI

在 PPU 中有一个 VBLANK Flag，在视频输出进入 VBLANK 时置 1，在 VBLANK 的结尾清除。同时 PPU 的控制寄存器中有一个 NMI Enable 位，VBLANK Flag 和 Enable 位与非后作为 CPU 的 NMI 输入。CPU 主动读$2002 的 VBLANK Flag 或者被动地通过 NMI，来完成游戏的定时。

Background，Sprite

PPU 的图形分为两种，Background 和 Sprite
下图中左边为游戏画面，右边为其 Background，马里奥由 sprite 组成



Background 和 Sprite 的基本单位是 8x8 像素点，一个像素点对应 4 位的 Palette 索引，其中高 2 位称为 Attribute，低 2 位在 Pattern Table 中，如果低 2 位为 00，则称其是透明的（transparent）。Pattern Table 由 tile 组成，一个 tile 有 16 个字节，对应一个 8x8 的区域，其划分为两个平面（plane）：前 8 个字节为第一个平面，对应的是 8x8 像素点 Palette 索引的第 0 位；后 8 个字节组成另一个平面，对应 Palette 索引的第 1 位。

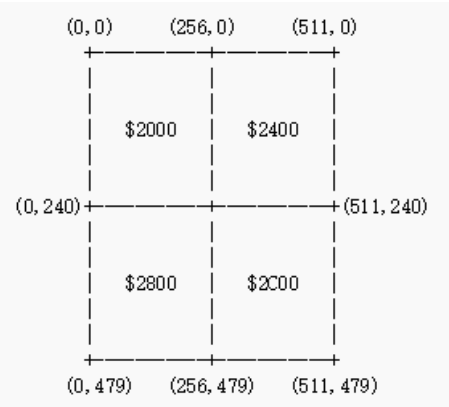
```
Bit Planes           Pixel Pattern
$0xx0=$41  01000001
$0xx1=$C2  11000010
$0xx2=$44  01000100
$0xx3=$48  01001000
$0xx4=$10  00010000
$0xx5=$20  00100000        .1.....3
$0xx6=$40  01000000        11....3.
$0xx7=$80  10000000  ===  .1...3..
                          .1..3...
$0xx8=$01  00000001  ===  ...3.22.
$0xx9=$02  00000010       ..3....2
$0xxA=$04  00000100       .3....2.
$0xxB=$08  00001000       3....222
$0xxC=$16  00010110
$0xxD=$21  00100001
$0xxE=$42  01000010
$0xxF=$87  10000111
```

注意比特位顺序，最左边的像素对应的是 MSB

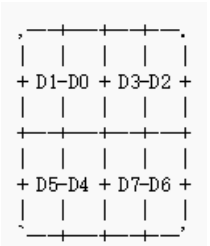
PPU 内存地址 0000H-0FFFH 为 Pattern table 0，1000H-1FFFH 为 Pattern table 1. Background，Sprite（8x8 大小）分别根据 PPU 控制寄存器中相应位选择 Pattern table 0 或 Pattern table 1，随后使用 8 位索引即可定位到 tile.

Nametables

Nametable 为背景内存，一个 Nametable 大小为 1024B，对应一个 256x240 的屏幕，32x30 个 8x8 区块。PPU 总共有 4 个 Nametable，而通常只会用到两个，采用 vertical mirroring 或 horizontal mirroring 进行镜像映射。



其中前 960 个字节，每个字节都对应一个 8x8 区块的 Pattern Table tile 索引。末尾 64 个字节为 Attribute Table，其一个字节分为 4 个 2-bit 区域，每个 2-bit 对应的是一个 16x16 区域的 Attriubute，即 Palette 索引的高 2 位，对应 4 个 tile. 如图所示，Bit 1-0 对应 16x16 区域的左上方，Bit 3-2 对应右上方，Bit 5-4 对应左下方，Bit 7-6 对应右下方。

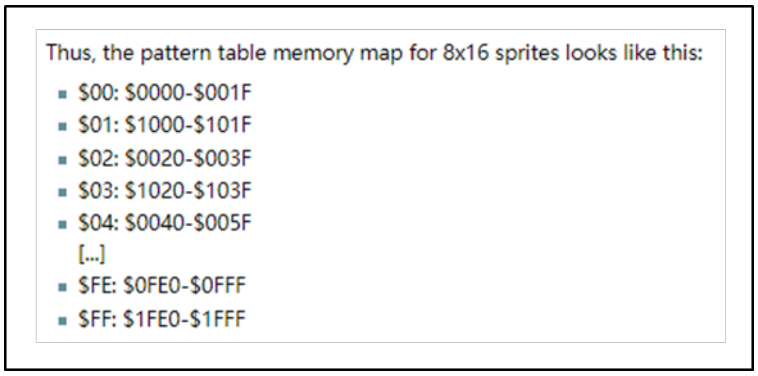


OAM (Object Attribute Memory)

OAM 有 256 个字节，由 64 个 Sprite 组成，一个 Sprite 有 4 个字节的信息，Sprite 的大小为 8x8 或 8x16 像素点，由 PPU 的控制寄存器决定。

Byte 0 - Y position of top of sprite
sprite 的显示会延迟一扫描行，如果这个字节为 0，那么 sprite 会出现在第 1 行而不是第 0 行；如果值大于等于 239，那么 sprite 不会被显示

Byte 1 - Tile index number
如果 sprite 大小为 8x8，那么这 8 位就是由控制寄存器选中的 Pattern Table 的索引
如果 sprite 大小为 8x16，则由这个字节的第 0 位选择 Pattern Table，高 7 位仍然作为索引


Thus, the pattern table memory map for 8x16 sprites looks like this:

- $00: $0000-$001F
- $01: $1000-$101F
- $02: $0020-$003F
- $03: $1020-$103F
- $04: $0040-$005F

[...]

- $FE: $0FE0-$0FFF
- $FF: $1FE0-$1FFF

Byte 2

```
76543210
||||||||
||||||++- Palette (4 to 7) of sprite
|||+++--- Unimplemented
||+------ Priority (0: in front of background; 1: behind background)
|+------- Flip sprite horizontally
+-------- Flip sprite vertically
```

Byte 3 - X position of left side of sprite

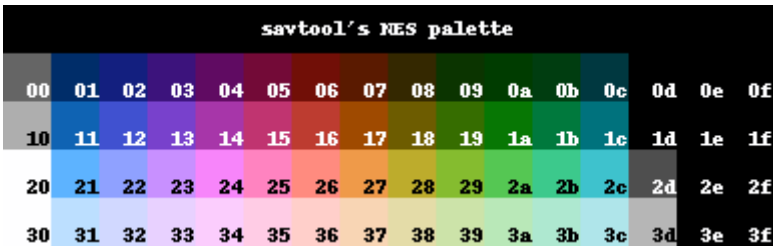
Palette

Palette 位于 PPU 内存地址 3F00H-3F1FH，可由 CPU 通过 PPU 端口写入。

Address	Purpose
$3F00	Universal background color
$3F01-$3F03	Background palette 0
$3F05-$3F07	Background palette 1
$3F09-$3F0B	Background palette 2
$3F0D-$3F0F	Background palette 3
$3F11-$3F13	Sprite palette 0
$3F15-$3F17	Sprite palette 1
$3F19-$3F1B	Sprite palette 2
$3F1D-$3F1F	Sprite palette 3

写时，$3F10/$3F14/$3F18/$3F1C 映射到$3F00/$3F04/$3F08/$3F0C
读时，只要 palette 索引低 2 位为 00，就映射到$3F00
$3F04/$3F08/$3F0C 处的颜色可以通过某种方法显示出来，但并没有实现这一特性。

Palette 中每个字节对应 1 种颜色，而这一字节只会用到低 6 位，6 位颜色转化为 VGA 输出的 24 位颜色，还需要通过一个 Palette 的作用，这个 Palette 不是 PPU 的一部分，是对于模拟器而言要实现的。



PPU Register

PPU 有 8 个“寄存器”映射到 CPU 地址$2000-$2007.

虽然这里称为寄存器，但不要真的认为有这样一个寄存器存在并一一映射关系。建议把 CPU 读、写$2000-$2007 某个地址分别看作一个动作，PPU 需要根据这个动作来改变内部寄存器的值或着将值送出。

各个寄存器的内容参考 http://wiki.nesdev.com/w/index.php/PPU_programmer_reference
这里就不一一介绍了，不过给出几个 Tip：

$2005，$2006 会对同一个 15-bit 寄存器 t 进行写，只是$2005 还会写另一个 3-bit 寄存器 x.（参考 http://wiki.nesdev.com/w/index.php/PPU_scrolling）

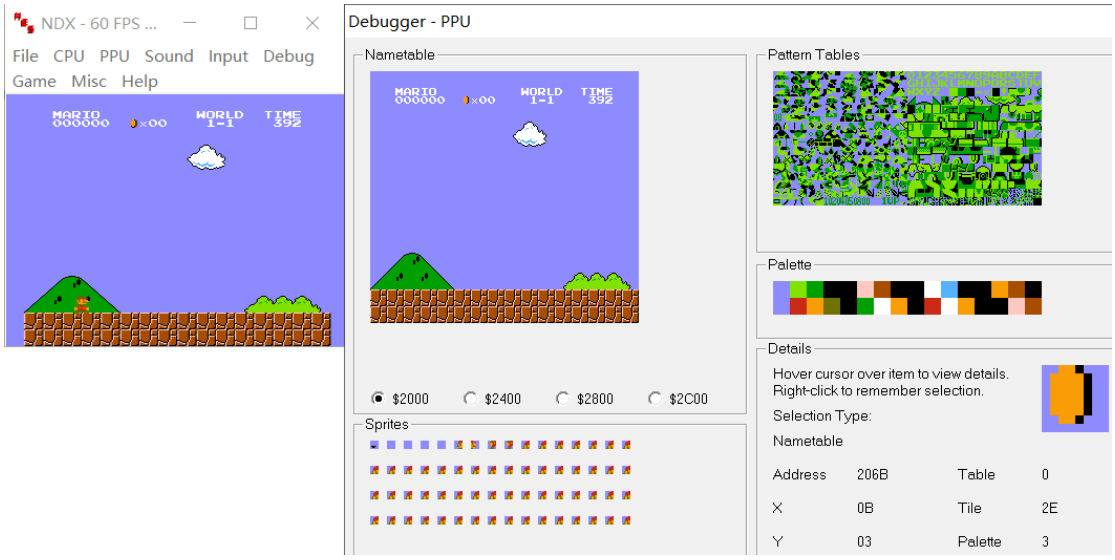
CPU 通过$2007 读会有一次延迟。CPU 把地址写到$2006 后，第一次读$2007 并不是该地址对应的值，第二次才是。简单来说，$2007 的读需要加一个 buf 寄存器。如果没有实现该特性，超级玛丽将不能正确的显示封面。

早期游戏的 PRG ROM 空间不足，可能会把部分只读数据放在 CHR ROM（Pattern Table）中，通过$2006，$2007 来读取这部分数据。（比如超级玛丽）

Sprite Zero Hit

如果 Hit Flag 没有被设立，那么超级玛丽将“卡死”在封面，封面左下方的马里奥也不会出现

超级玛丽的 sprite 0 在状态栏中金币的下方，看起来就像是金币的阴影。
超级玛丽每一帧输出先将 scroll 设为 0，在 hit flag 被设置后再向$2005 写入 scroll，这就是其状态栏的实现方法，超级玛丽的状态栏一直位于第一个 Nametable 而不需要移动。



（使用 NintendulatorDX 模拟器进行查看）

PPU 的视频输出类似于之前的字符交互实验，只不过更复杂而已。Nametable 就好比存 ASCII 码的 RAM，Pattern Table 加上 Attribute 类似于 ASCII 码到像素点阵的 ROM.

PPU 输出的视频分辨率为 256x240，为了放大显示，横向上一个像素点输出占两个像素点时间，纵向上每两行的第二行重复第一行的动作（也不完全重复，比如 sprite evaluation）

也可以设立一个 256x240x6bit 的显存，PPU 写显存，VGA 输出时读显存，这是我在中间想到的另一种实现方式。

以下叙述中像素点时间均指 256x240 格式下的像素点时间

背景输出

以 8 个像素点时间为单位，以下每次访存占用两个像素点时间

- 读取 Nametable，获得 tile 地址
- 读取 Attribute table
- 从 PatternTable 读 tile 的平面 0 的字节（pattern_low）
- 从 PatternTable 读 tile 的平面 1 的字节（pattern_high）

每一帧、每一行的起始 scroll Y，scroll X 可通过$2005 设置
http://wiki.nesdev.com/w/index.php/PPU_scrolling

pattern_low，pattern_high 各有一个 16-bit 移位寄存器，每个像素点时间右移一位，每 8 个像素点时间，高 8 位载入新的数据。3 位的 fine_x_scroll 从低 8 位中选择一位作为 Background 的像素输出。

对于每一行，背景的读取需要提前 16 个像素点时间从而完成载入，一般认为这是在上一行的结尾完成，因为一行的第 0 个像素点时间就处于显示区域。而在 VGA 中，一行的显示区域前还有像素点时间，于是可以在这里来完成需要提前的操作。

Sprite 输出

除了 256 字节的 OAM 外，还有一个 32 字节的 second OAM，可以存放 8 个 sprite，在每一扫描行都会进行 evaluation，遍历主 OAM，如果 Sprite 在下一行会被显示，那么将其 4 个字节放入 second OAM 中。evaluation 结束后，根据 second OAM，从 Pattern Table 读取 sprite 的 tile，sprite 的 X position 会放到一个计数器中，在显示区域时每过一个像素点时间减一直到 0。当计数器不为 0 时，那个 sprite 的像素输出为 0，当计数器为 0 时，每次右移一位，高位 0 填充。

选取 8 个 sprite 中第一个不透明的 sprite 作为像素点输出。

时钟

PPU 的时钟使用 VGA 视频输出的时钟频率，也就是 25MHz.
由于 PPU 的 OAM 会同过 DMA 方式从 CPU 内存载入 256 个字节，而 DMA 必然是时钟同步实现起来方便，因此 PPU、CPU 内存 RAM 的时钟都采用 25MHz.

CPU 的时钟频率没有上限，虽然部分程序在一些地方会用指令周期数来大致计时，但游戏的主要逻辑是通过 PPU 视频输出的 VBLANK Flag 或其产生的 NMI（其频率与视频输出帧率一致，在 VGA/NTSC 中为 60Hz）来定时的。

当然，这是在没有实现 APU 的前提下。和视频输出一样，音频输出的时钟频率也不能乱选，否则音频就乱了。在 NTSC NES 中，CPU 的时钟频率为 1.789773 MHz，是 PPU 时钟频率的三分之一，而 APU 与 CPU 同在 RP2A03 芯片中，时钟频率与 CPU 一致。

APU 有只写端口映射到 CPU 内存地址上，此时，如果 CPU 频率比 APU 还高，那就必须实现一个 I/O 缓冲器，这是不划算的，因此 APU、CPU 的时钟频率选择和 NTSC NES 一致。使用 IP 核中的锁相环来生成 1.789773MHz 时钟。

此时由于 CPU 与 CPU 内存 RAM 不同步，就需要对 CPU 输出的读写信号进行同步处理。

```
1   assign cpu_clk_posedge = (~cpu_clk_sync[2]) & cpu_clk_sync[1];
2
3   always @ (posedge CLK_25) begin
4       if (!RESET_L)
5           cpu_clk_sync <= 0;
6       else if (!cpu_pause)
7           cpu_clk_sync <= {cpu_clk_sync[1:0], CPU_CLK_P1};
8   end
9
10  always @ (posedge CLK_25) begin
11      if (!cpu_pause && cpu_clk_posedge) begin
12          cpu_read_sync <= (CPU_WREN == 1'b0);
13          cpu_write_sync <= (CPU_WREN == 1'b1);
14      end
15      else begin
16          cpu_read_sync <= 1'b0;
17          cpu_write_sync <= 1'b0;
18      end
19  end
```

键盘

实验中使用键盘完成 Standard Controller（游戏手柄）的功能，
参考：　http://wiki.nesdev.com/w/index.php/Controller_reading
　　http://wiki.nesdev.com/w/index.php/Standard_controller

复用之前实验写的键盘模块，这里键盘模块唯一需要做的就是输出相关按键的状态，按下则相应位为 1，否则为 0。

$4016 write

```
7  bit  0
---- ----
xxxx xxxS
       |
       +- Controller shift register strobe
```

当 S 为 1 时，Controller shift register 每个周期都会载入（reload）8 个按键的状态

$4016/$4017 read

```
7  bit  0
---- ----
xxxx xMES
     |||
     ||+- Primary controller status bit
     |+-- Expansion controller status bit (Famicom)
     +--- Microphone status bit (Famicom, $4016 only)
```

controller 1 通过$4016 读取，controller 2 通过$4017 读取
每次读取都会在第 0 位获取一个按键的状态（shift register 最低位），随后 shift register 右移一位。

INES（.nes）文件

参考：http://wiki.nesdev.com/w/index.php/INES

将下载的游戏.nes 文件通过程序转为.mif 即可将游戏放到开发板上。
INSE 文件的前 16 个字节为 header

Byte	
0-3	Constant $4E $45 $53 $1A ("NES" followed by MS-DOS end-of-file)
4	Size of PRG ROM in 16 KB units
5	Size of CHR ROM in 8 KB units (Value 0 means the board uses CHR RAM)
6	Flags 6 - Mapper, mirroring, battery, trainer
...	...

在这里需要关心的是字节 4-6
由于没有实现 mapper，能运行的游戏有限。不需要 mapper 的游戏，PRG ROM 大小为 16/32KB，CHR ROM 大小为 8KB，这一点光看文件大小就能得知。

字节 6 只需要关心其第 0 位和第 3 位，其代表的是 Nametable 的 mirroring 方式

```
76543210
    |  |
    |  +- 0: horizontal mirroring (vertical arrangement)
    |     1: vertical mirroring (horizontal arrangement)
    |
    +---- 1: Ignore above mirroring bit; instead provide four-screen VRAM
```

APU

APU 不是必须实现的，有了 CPU 和 PPU 就可以运行游戏了。

APU 由两个 Pulse 方波频道，一个三角波频道，一个噪声频道，以及一个 DMC 频道组成。

我在最后用了 3 天写了个无 DMC 频道的 APU，效果不怎么好，但已经可以听出基本的游戏声音了。APU 模块写出来后上板有效果我就没 Debug 了。另外，CPU-PPU 中有个未解的 bug：游戏的时间速度是正常的两倍，这也是会影响 APU 的，因此 APU 做得相当粗糙。

APPENDIX

报告附件包括

- *MCS6500 Microcomputer Family Programming Manual*
- 模拟器 nintendulatordx
- OPCODE.xlsx ——official opcode 表格

.nes 游戏文件可在 https://www.emulatorgames.net/roms/nintendo/下载

Nesdev Wiki 左侧提供了离线网页下载