从逻辑门到电子计算机

蒋炎岩

南京大学计算机科学与技术系

2014 年 10 月 8 日

Outline

影响了一个时代的两个产品

- 让计算机走进千家万户的先驱
  - 它们都有一颗 25 美元的心

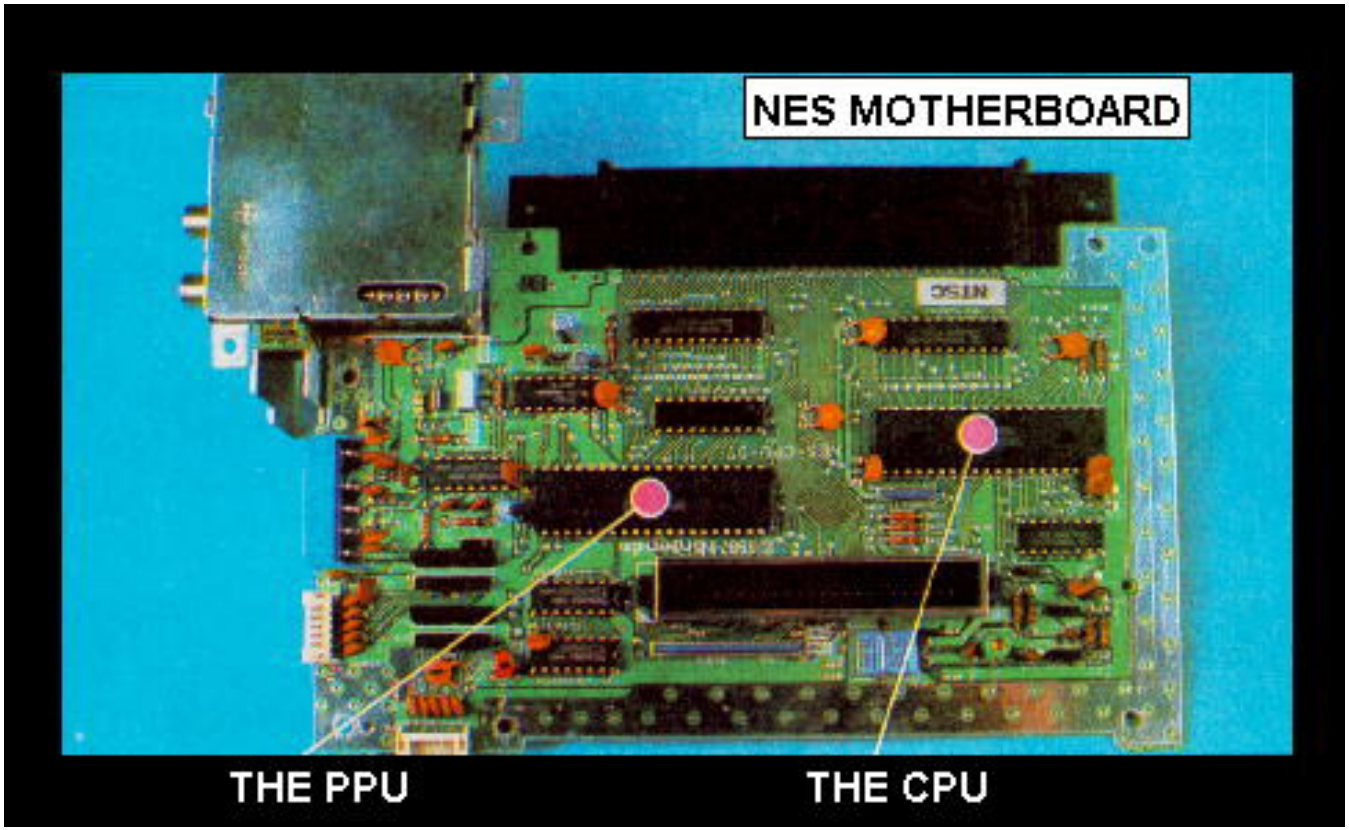
Apple II, 1977

NES (FC), 1983

今天的主角：NES/Famicom/小霸王/学习机

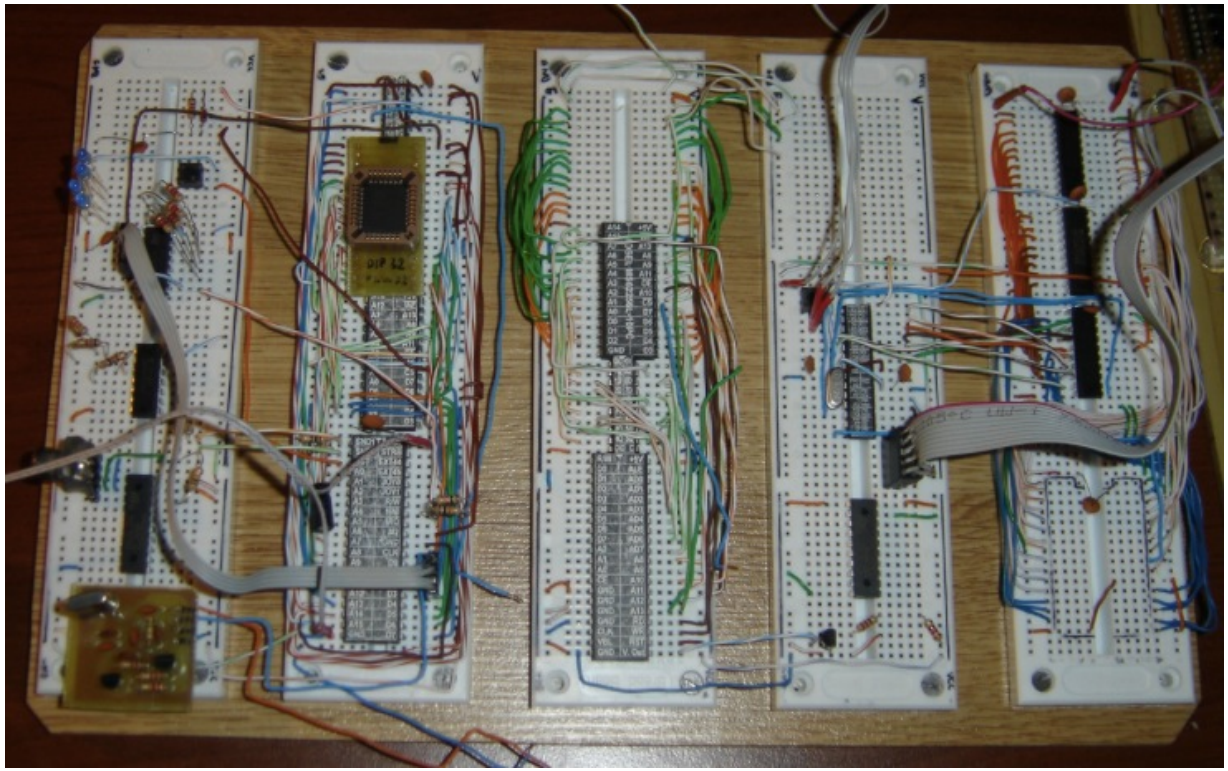
NES 究竟是什么？

- 拆开 NES
  - 主要部分是导线连接的逻辑电路和内存
  - 以及一些 A/D 转换、时钟等模拟电路
  - 给你一个面包板，你也能搭！



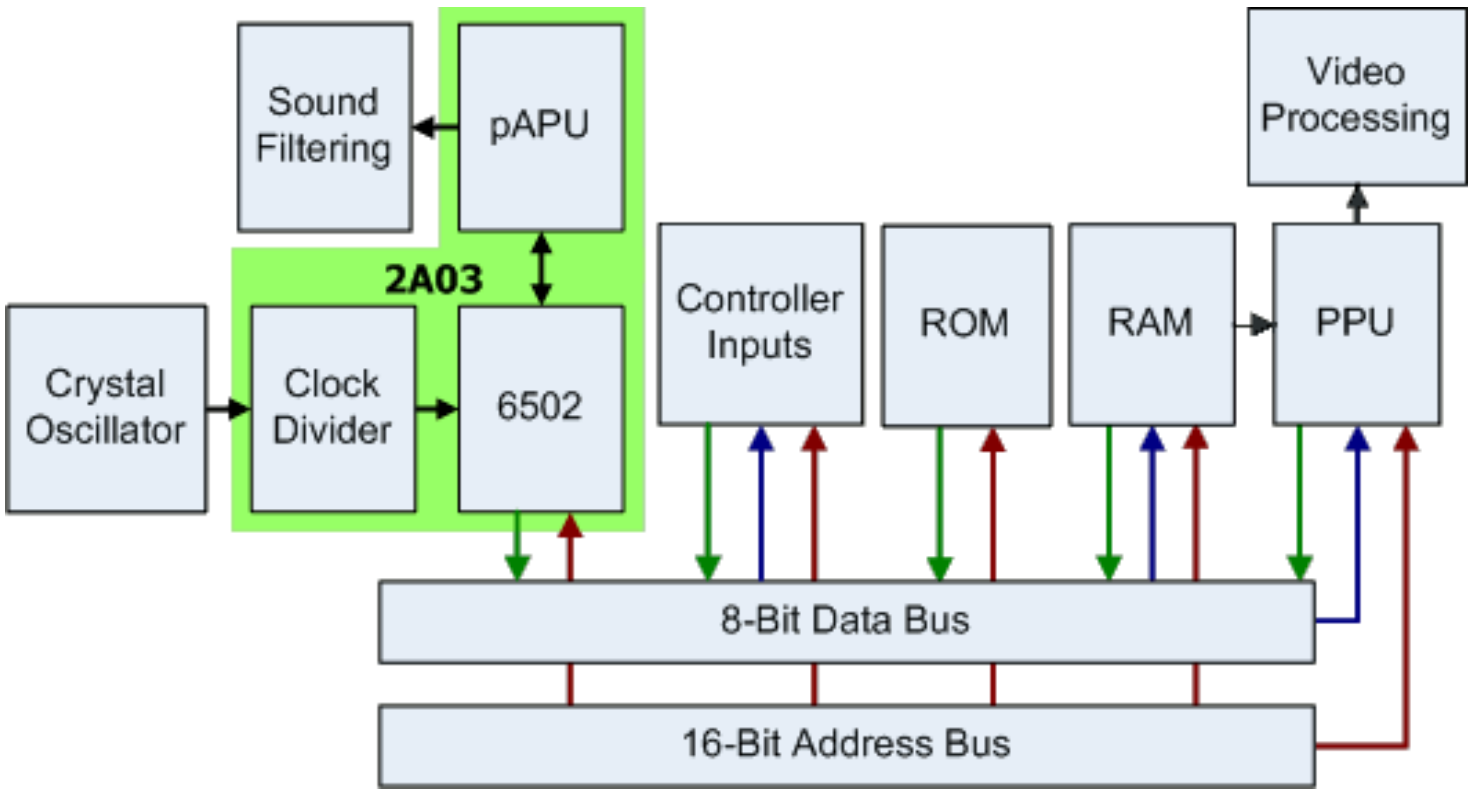
外国友人自制 NES

- 集成电路 ≈ 大号时序逻辑电路
  - 自制 NES 基本就是将集成电路用导线正确地连接起来
  - 所以 Jobs 的团队能在车库里组装出 Apple 电脑



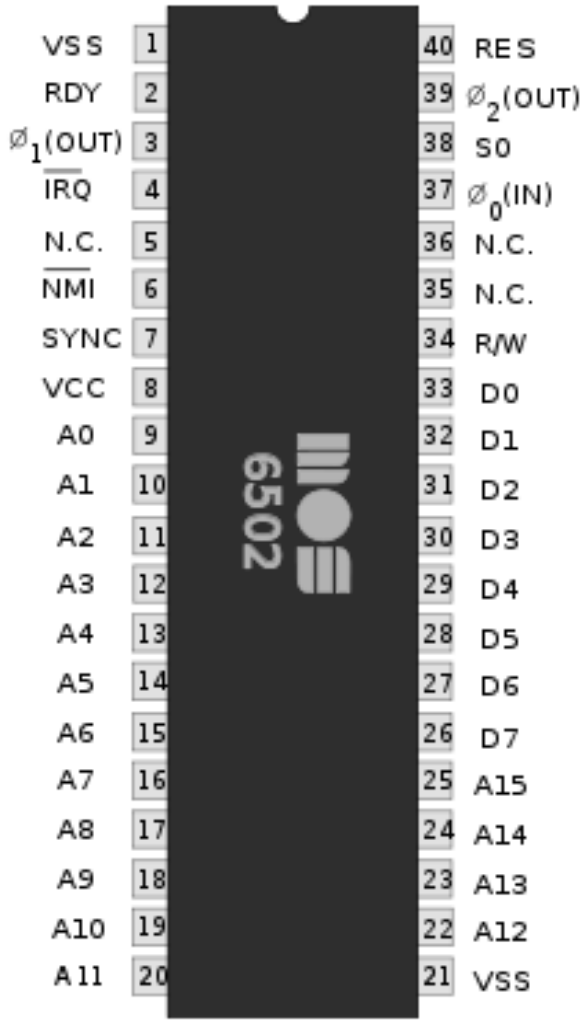
NES 是个完整的计算机系统

- 中央处理器、存储器、图形处理器、内存控制器、总线
  - 今天就给大家讲一个计算机系统的故事



Outline

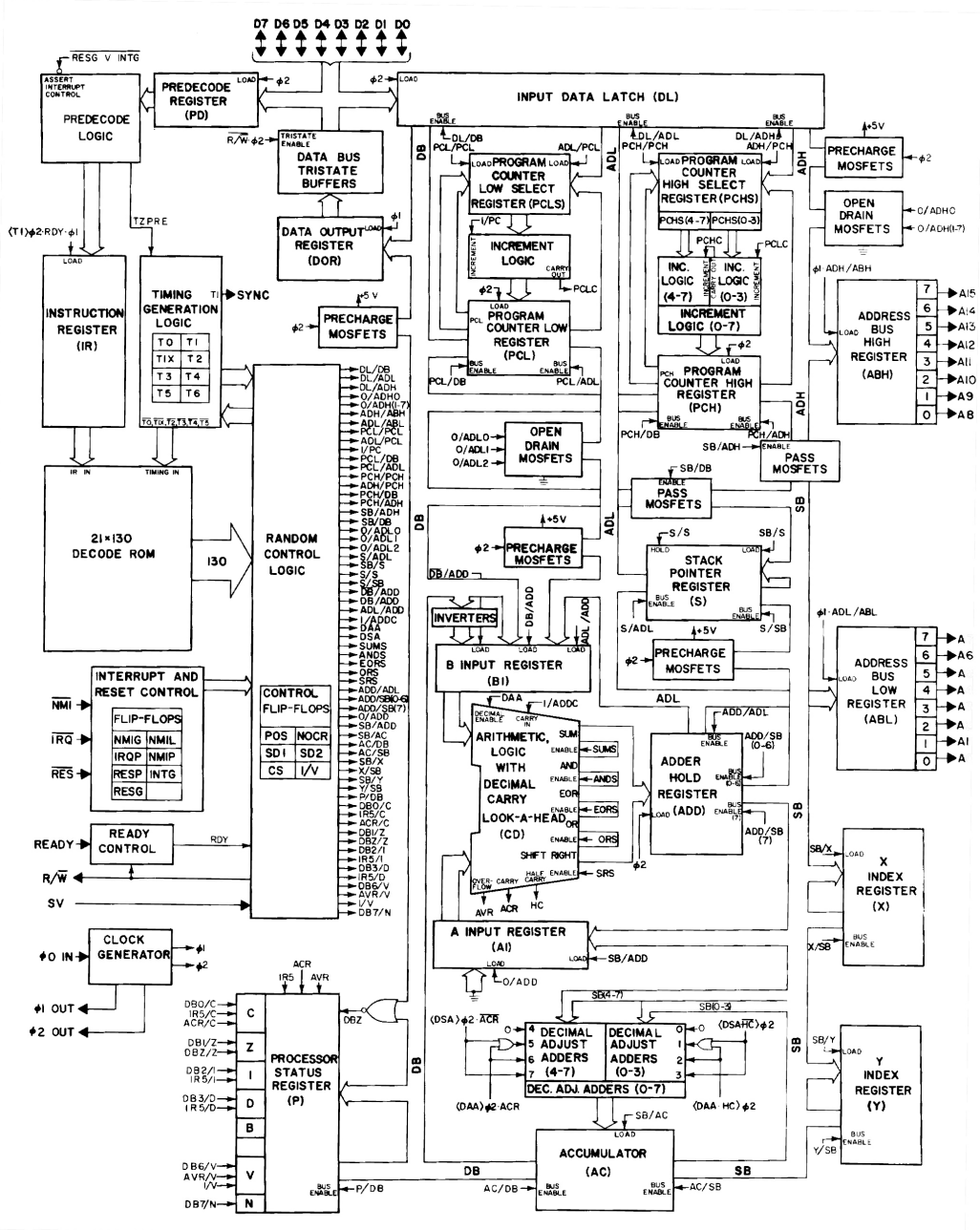
远观 6502


VSS 1
RDY 2
Ø1(OUT) 3
IRQ 4
N.C. 5
NMI 6
SYNC 7
VCC 8
A0 9
A1 10
A2 11
A3 12
A4 13
A5 14
A6 15
A7 16
A8 17
A9 18
A10 19
A11 20
40 RES
39 Ø2(OUT)
38 S0
37 Ø0(IN)
36 N.C.
35 N.C.
34 R/W
33 D0
32 D1
31 D2
30 D3
29 D4
28 D5
27 D6
26 D7
25 A15
24 A14
23 A13
22 A12
21 VSS
6502

时序逻辑观点下的 6502 中央处理器

- 处理器 $\approx$ 寄存器 $+$ 主存接口 $+$ 巨大的组合逻辑电路
  - 程序员可见的状态由内部的 8 位寄存器 $A,\ X,\ Y,\ SP,\ PSR$ 和 16 位寄存器 PC 确定
- 输入
  - 时钟 CLK
  - 中断信号 $\overline{IRQ}$ 和 $\overline{NMI},$
  - 总线输入 $D_{I0} \sim D_{I7}$
- 输出
  - 地址信号 $A_0 \sim A_{15}$，读写 $R/\overline{W}$
  - 总线输出 $D_{O0} \sim D_{O7}$

6502 中央处理器：一大堆逻辑门



6502 中央处理器的行为

- 在时钟 CLK 的控制下，反复执行指令
  - 输出正确的 $A_0 \sim A_{15}$ 和 R/W 信号，就能读写内存
  - 从内存地址 PC 取出指令，指令第一字节为指令类型
  - 根据指令内容实施计算，结果保存到寄存器或内存中
  - 指令会多次访问内存，因此会占用多个时钟周期
- 例子
  - INA: $A \leftarrow A + 1$;
  - STY: $M \leftarrow Y$; (M 根据寻址方式确定)
  - PHX: $Stack \leftarrow A$; ($Stack$ 由 SP 确定)
  - RTS: $PC \leftarrow Stack$;
  - BRK: $Stack \leftarrow PC$; $PC \leftarrow \$FFFE$

简化版的 PA

- 给定已经实现好的函数
  - `read_byte(`a`)` - 从地址 a 读一个字节
  - `write_byte(`a`,` d`)` - 将 d 写入地址 a
  - `read_irq()` - 读取 $\overline{IRQ}$ 状态
  - `read_nmi()` - 读取 $\overline{NMI}$ 状态
- 编写一个 6502 处理器的模拟器
  - 只需编写 `execute()`，功能是执行一条指令
  - 提问：有多少同学感到困难？
  - 这几乎等价于我们的 PA：编程实现处理器的逻辑功能，并且能用控制台操纵处理器的执行

是时候 RTFM 了！

- 又是 RTFFFFFFFFFFFFFFFFM！有完没完？
- RTFM 的技巧
  - 在开始看之前，首先要明确你的精确目标是什么 (做 PA 不是目的)
- 为了实现 `execute()`，整理我们的工作流程
  - 为寄存器分别定义变量 X, Y, A, SP, P, PC
  - 根据 PC 取出指令 (使用 `read_byte`)
  - switch-case 指令的类型，模拟指令执行
  - 重点就是弄清楚每条指令的精确行为
    - 处理器的基础知识、从何处读取操作数、执行何种运算、计算后保存到何处、对哪些标志位产生影响……

理解 6502 指令集: RTFM

- 理解 6502 指令集的阅读顺序
  - 简介 (搜索 6502 → Wikipedia)
    - 6502 处理器的历史、影响、设计和例子
  - 教程 (搜索“NES 模拟器开发”→ 网络爱好者博客)
    - 指令列表、每条指令的大致语义、寻址方式介绍等
  - 手册 (搜索“6502 manual”→ MCS6500 Family Programming Manual)
    - 对每条指令处理器行为的精确定义，以及丰富的示例
- 实现 `execute()`: “照着手册实现”
  - 根据操作码，确定指令的类型和寻址方式
  - 根据寻址方式取出操作数
  - 根据指令类型写入计算结果、更新符号位

PA 为什么这么可怕？

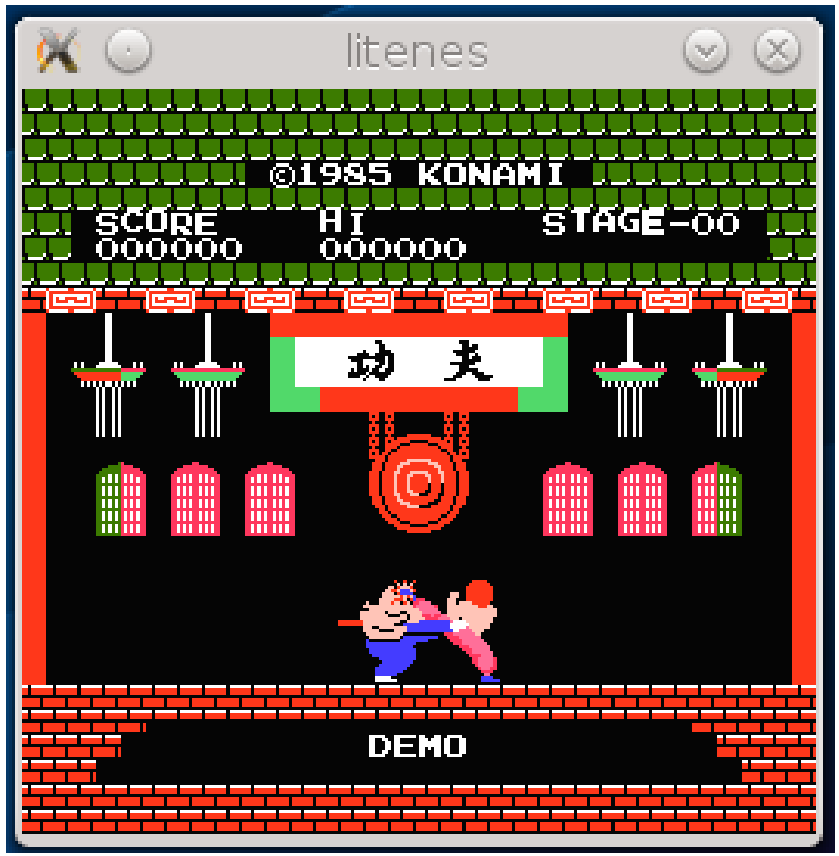
- 如果只是和程序设计课一样，让你在熟悉的环境里写一个 `execute()`，看起来就没那么可怕了
  - 放松，PA2 的任务就是编写 `execute()`
  - 如果你感到这个任务很可怕，你需要考虑重新学习编程
    - 上 coursera 上课吧！
- PA 可怕在一切都是陌生的
  - 不习惯的工作方式 (Linux 命令行)
  - 不习惯的编程语言 (纯 C、指针、宏)
  - 完全陌生的辅助工具 (vim, ctags, gprof, ...)
  - PA0∼1 恰恰是用心良苦的训练啊，拖到最后你就输了

克服恐惧感

- 基本原理：一切合理的要求都会被满足
  - 坚信你所想要完成的事情，一定有很多人也想完成，因此有一个工具能方便地完成它
  - 计算正是有这种不可思议的力量
- 例子
  - 查看二进制文件的各类信息 (`binutils`)
  - 知道 binutils 的工具集 (`info`)
  - 对程序性能调优，查找程序运行时间的瓶颈 (`gprof`)
  - 从网络上下载文件 (`wget`)
  - 统计 git 记录的编译次数 (`git log | grep | wc`)
  - vim 中替换选中区域的字符：`'<,'>s/pattern/substitute/g`
  - 在 vim 中浏览源代码 (ctags)
  - ……

通往计算机系统的大门已经向你敞开

- 大约 500 行代码 (实现指令的一个子集) 的 6502 模拟器就能驱动起 NES 系统
  - LiteNES: `https://github.com/NJUOS/LiteNES`



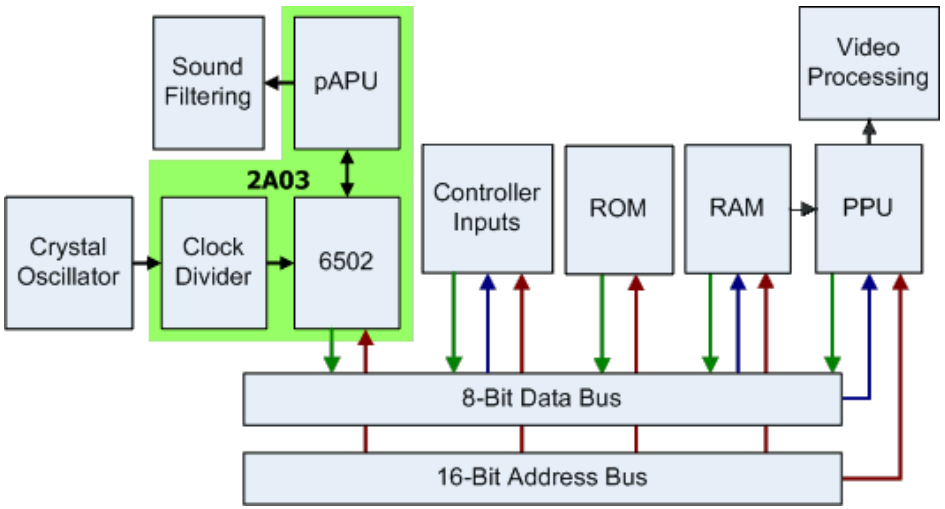
Outline

CPU $\neq$ 计算机系统

- 6502 只是一个 40 接口的执行部件
  - 内存在 6502 外部，执行的指令是从 6502 外部输入的
- 只有 6502 还无法解决的问题
  - 如何确定手柄上按键的状态？
  - 如何在屏幕上绘制图形？
  - 如何按时执行屏幕刷新代码？
  - 如果你是 NES 的设计师，你会如何解决这些问题？

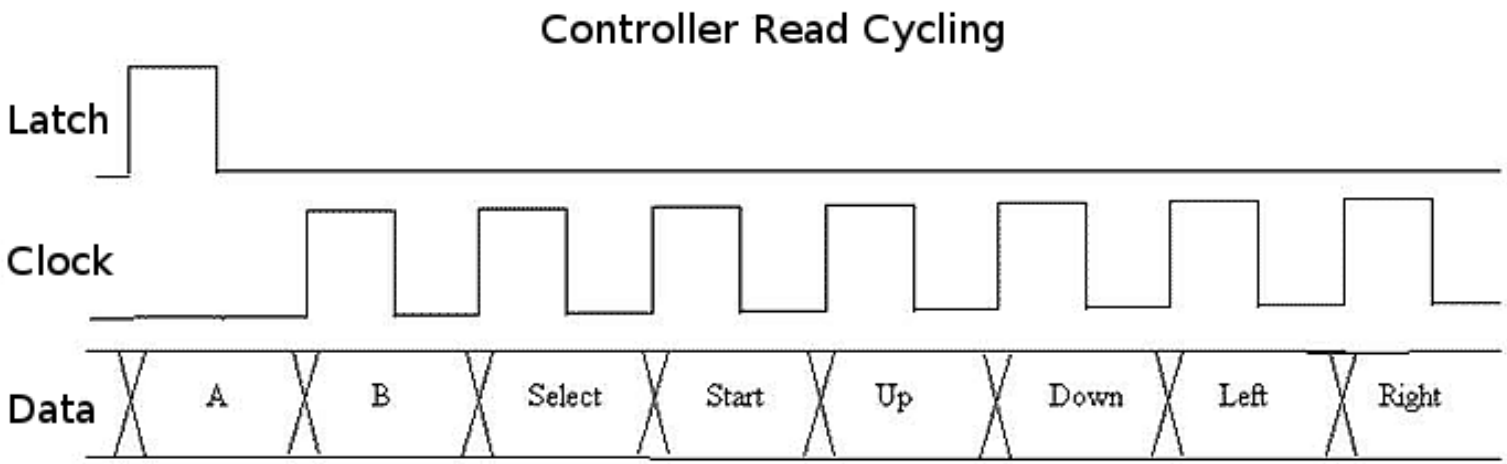
将外部设备映射到内存

- 如何解读“内存地址”取决于系统的配置！
  - 内存包含了主存、外存和所有外部设备
- 共享同一个 16 位地址空间
  - 根据地址总线的信号选择主存或外设



内存映射的 I/O：手柄

- Player 1 手柄，映射在内存 $4016
  - 通过选择器，地址 $4016 由物理导线连接到手柄
- 向 $4016 依次写入 1、0 复位阅读状态——边沿复位
  - 复位后读 8 次 $4016 获取 8 个按键的状态——边沿计数
  - 原来用数字电路课学过的知识就能造呀

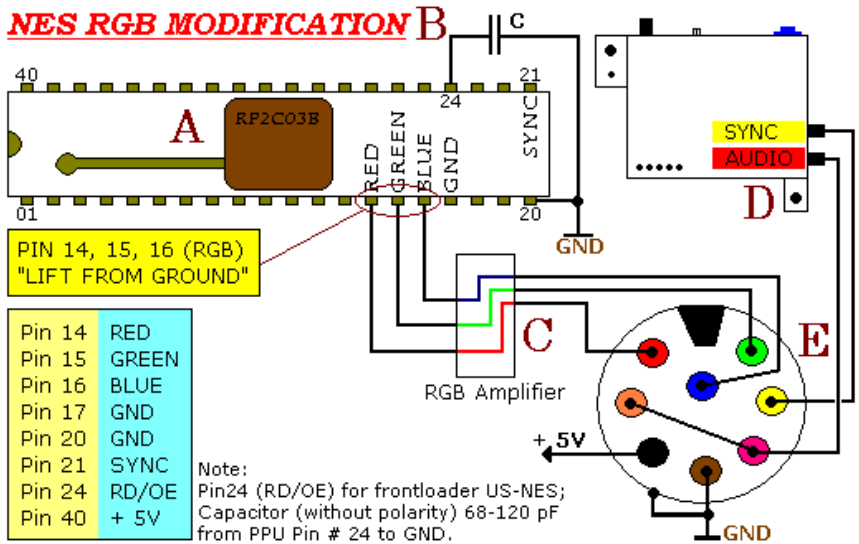


另一种手柄：光线枪

- 只在高富帅家才见到过的电子射击枪，和神奇的游戏
  - 其实它也是一种手柄！
  - 猜一猜它是如何工作的？

解析 I/O 设备：视频加速器

- CPU 无法承受视频处理的负担
  - 每秒渲染 $256 \times 240 \times 60$ 象素并按时序输出
- PPU RP2C07 应运而生
  - 根据 CPU(由内存映射 I/O) 发出的指令渲染图片
    - CPU 只需描述“如何渲染”，不必亲自上阵
  - 根据时序将图像信号输出



NES 系统的显示加速 (cont'd)

- 构成图像的基本单位是 8×8 的 tile
  - 可自由变化的部分 (sprite) 共 64 个 tile
  - 外加一个不可变化的静态背景
- PPU 如何与 CPU 协作输出图形？
  - CPU 将背景写入 nametable (随时可以更换)
  - 运行时，CPU 指定背景的偏移 (卷屏)，并描述每个 sprite 的信息 (位置、颜色、tile 类型)

NES 系统的中断处理

- 中断：当 $\overline{IRQ}$ 信号为 0 时，处理器执行完当前指令后，自动在后续的时钟周期执行以下操作：
  - 把中断返回地址保存到堆栈上 ($Stack \leftarrow PC$)
  - 把 CPU 状态保存到堆栈上 ($Stack \leftarrow P$)
  - 设置 P 的 $IF = 1$ (不再响应中断)
  - 跳转到中断处理程序 ($PC \leftarrow M[\$FFFE]$)
- 中断用来通知处理器外界事件的发生
  - 定时器事件、I/O 设备数据到达等
  - 中断的意义将在操作系统课程中展示

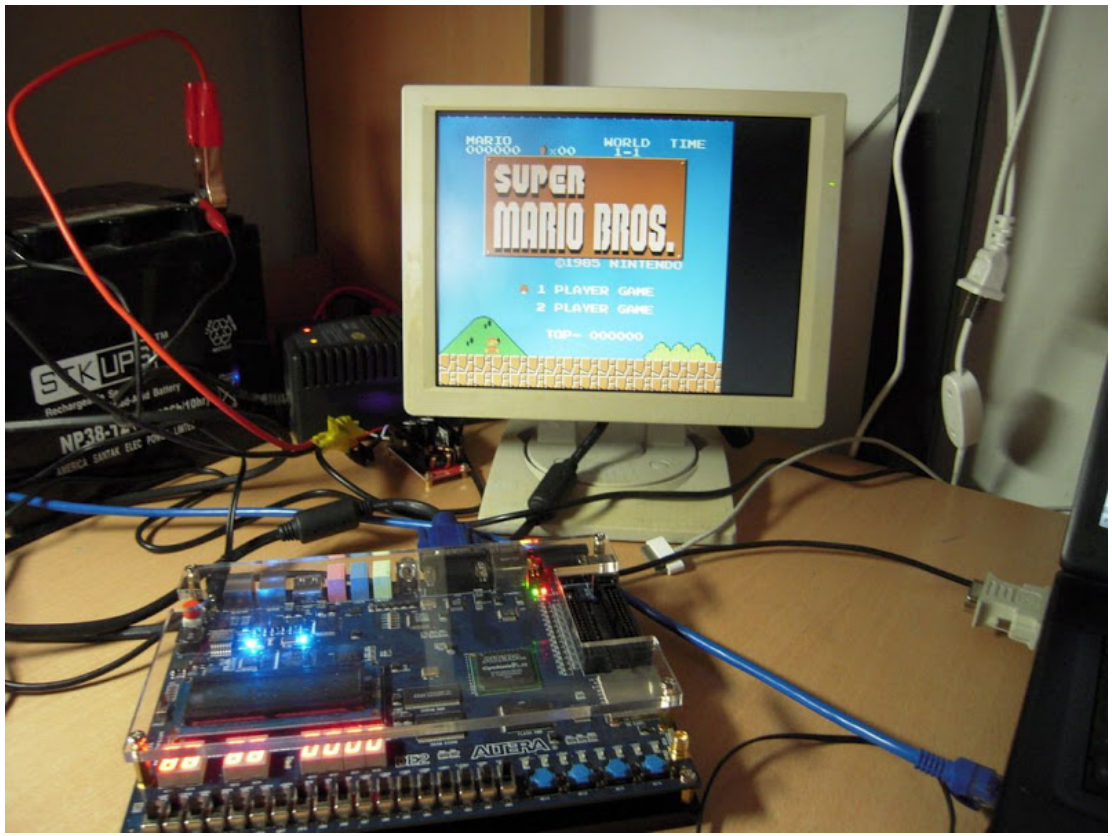
小结：从逻辑门到电子计算机

- 6502 处理器：一个庞大的时序逻辑电路
  - 一个能够根据内存数据执行指令的核心
- 6502 与外界的交互：中断与 I/O
  - 设备以内存映射的形式被 CPU 访问
  - 设备能够向 CPU 发送中断
- 不再神秘，游戏就是在这样的计算机上运行起来的
  - 你想写一个模拟器吗？如果你还不太理解，可以开始阅读 LiteNES 的代码

Outline

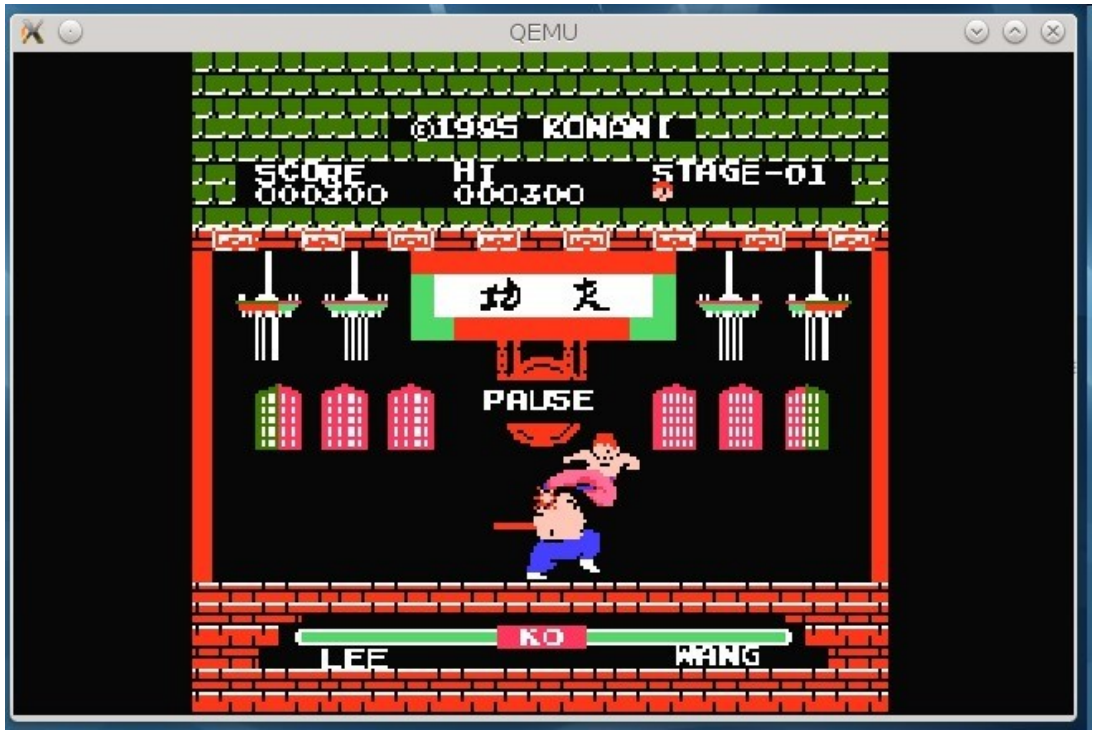
创造一个计算机系统比想象中容易

- 网友用 DE2-70 自制的 NES



创造一个计算机系统比想象中容易

- 我们的 LiteNES 也是为操作系统实验准备的
  - 在下个学期 (OS 实验)，大家将会让这个模拟器直接运行在没有操作系统的计算机上！



完成 PA，就意味着创造了计算机系统

- NEMU 的核心，就是模拟处理器和设备的行为
  - 取指令 $\rightarrow$ 执行指令
- NEMU 模拟了一个真实部署的现代体系结构
  - 存储保护、虚存管理、向量中断……
  - 这些概念与老式处理器一脉相承，但经过时代的净化
- 完成 PA $=$ 对硬件的行为有全新的认识
  - 不要再怀疑 PA 了，你们在完成一件前所未有的事情，能将你的认识水准提升到全新的高度

Outline

我还是感到困难，我到底缺少了什么？

- 相信在座都理解计算机系统的工作原理
  - 仍然不能下手，只是缺少随心所欲编程的能力
- 以一个问题为例
  - 写一个函数，对一个字符串表达式，计算它的值
    - 3+(4*10-6)/7*8
    - (6/2)*(8-7)
  - 用计算机解决问题的思维
    - 用指导小学生的方法，写出“做法说明”

表达式求值：另一种算法

- 回想你的小学老师是怎么教会你们表达式求值的？
  - 每次选一个优先级最高的运算符
  - 运算符左右两边一定是数字
  - 计算这个运算的结果，替换表达式的值
- 用计算机解决这个问题
  - 如何在字符串中找一个优先级最高的运算符？
  - 找到运算符后，如何找到两边的数字？
  - 如何将计算结果替换到字符串中？
  - 我们已经将一个不太容易的问题，分解成程序设计课程教学目标中的内容了

当然了，新鲜出炉的代码几乎不可能是对的

- 对“机器永远是对的”的解读：在没有更多先验知识的情况下，假设程序输出了非预期的结果
  - Pr[是 Linux/GCC/模拟器的错] $<$ Pr[我出门被车撞死]
  - Pr[是 Intel 的错] $<$ Pr[天上掉下来东西把我砸死]
- 当然了，不是说这些系统没有 bug
  - Linux, GCC 是有很多 bug 的[1]
  - Intel 处理器新功能的实现也有 bug (HTM)
- 随着你们能力的增长，你们也可能会探索这些未知的领域

[1] Lu et al. Compiler validation via equivalence modulo inputs. In *Proc. of PLDI*, 2014.

为什么我的程序总是不对？

- 三种类型的错误
  - fault (bug) - 你不能正确实现功能的那部分代码
  - error - 程序执行时不符合预期的状态
  - failure - 能够观测到的错误 (非预期的输出/段错误)
- 发生的过程
  - fault 导致 error, error 导致 failure
- 一个例子
  - fault: `if (head = NULL) {...}`
  - error: `head` 被错置为 `NULL`
  - failure: segmentation fault

难以捉摸的错误

- Fault (bug) $\not\rightarrow$ error, error $\not\rightarrow$ failure
  - 即便你的程序有 bug，你的代码也未必会触发
  - 即便触发了 bug，你也未必能观察到异常
  - 等程序出错的时候，已经离 bug 十万八千里了
- 两个黄金准则
  - 尽可能早地触发 failure，指示 fault 存在 $\rightarrow$ 测试
    - 知道为什么 PA 里有 test 了吧！
  - 尽可能早地检查程序中出现的 error $\rightarrow$ 断言
    - 知道为什么 PA 里有那么多 assert 了吧！

例子：循环链表 (list_head)

- 任何时候访问链表结点 p，检查
  - assert($p \neq \mathrm{NULL}$);
  - assert($p \mapsto prev \mapsto next = p$);
  - assert($p \mapsto next \mapsto prev = p$);
- 看起来非常容易，但是
  - insert/delete 时设错任何一个指针，都会违背断言！

例子：排序

- 将 $a[0] \ldots a[n-1]$ 排序后，检查
  - for $(i = 1;\ i < n;\ i = i + 1)$ assert$(a[i-1] \leq a[i])$;
- 对于只使用交换 (且没有越界交换) 的排序，这个断言能够保证排序的正确性
  - 试图交换 i, j 时，assert$(i \neq j \wedge 0 \leq i, j < n)$

例子：表达式求值

- 表达式求值中有许多陷阱
  - $check\_parentheses(p, q) \equiv s[p] = '(' \wedge s[q] = ')'$
  - $------1,\ 1+-2$
  - $2**(\$eax),\ **(\$esp)$
  - ……
- 由于难以判定求解的正确性 (缺少 test oracle 和容易验证的 specification)，只能设计有限的断言
  - 在算法执行过程中，表达式总是合法 (例如括号总是配对)
- 需要丰富的测试用例

回到基本原理：一切合理的要求都会被满足

- 你所想要完成的事情，一定有很多人也想完成，因此有一个工具能方便地完成它
- 正确的工具能帮助你迅速找出许多程序的 bug
  - 编译时检查：-Wall -Werror
  - 运行时检查：Valgrind, gcov
  - 性能检查：gprof
  - ……
- 文档阅读的三个步骤
  - 简介 $\rightarrow$ 教程 $\rightarrow$ 手册
  - 试试这些工具，它们会陪伴你们很多年的！

Happy Hacking!

- 从逻辑门到游戏主机
  - 原来完整理解一个计算机系统并不那么难
- Fault, error 和 failure
  - 可是实现中难免遇到会遇到障碍
- 测试、断言和工具
  - 不用怕，用正确的方式消灭 bug