

I/O设备选讲

王慧妍

why@nju.edu.cn

南京大学



计算机科学与技术系



计算机软件研究所



提醒

- PA

PA2: Deadline: 2023年11月19日 23:59:59

- Lab

Lab2: Deadline: 2023年11月19日 23:59:59

本讲概述

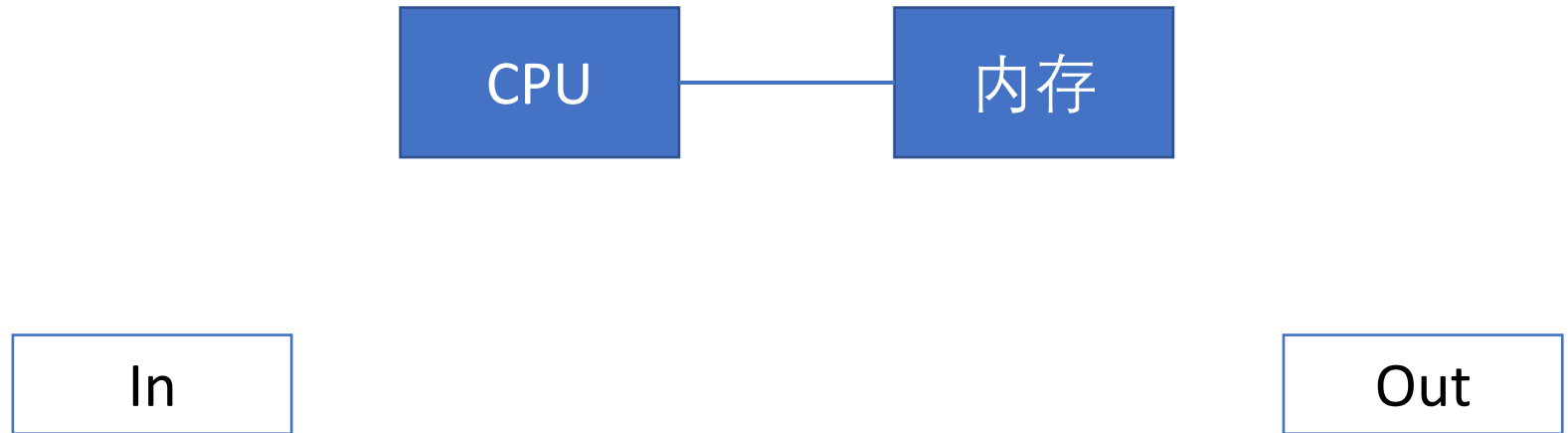
对所有学计算机的同学来说，“设备”才是真正触摸到的，但设备到底是什么？

按下键盘之后，计算机硬件/软件系统到底发生了什么？

- 本讲内容
 - 处理器-设备接口
 - I/O设备选讲

处理器-设备接口

Turing Machine

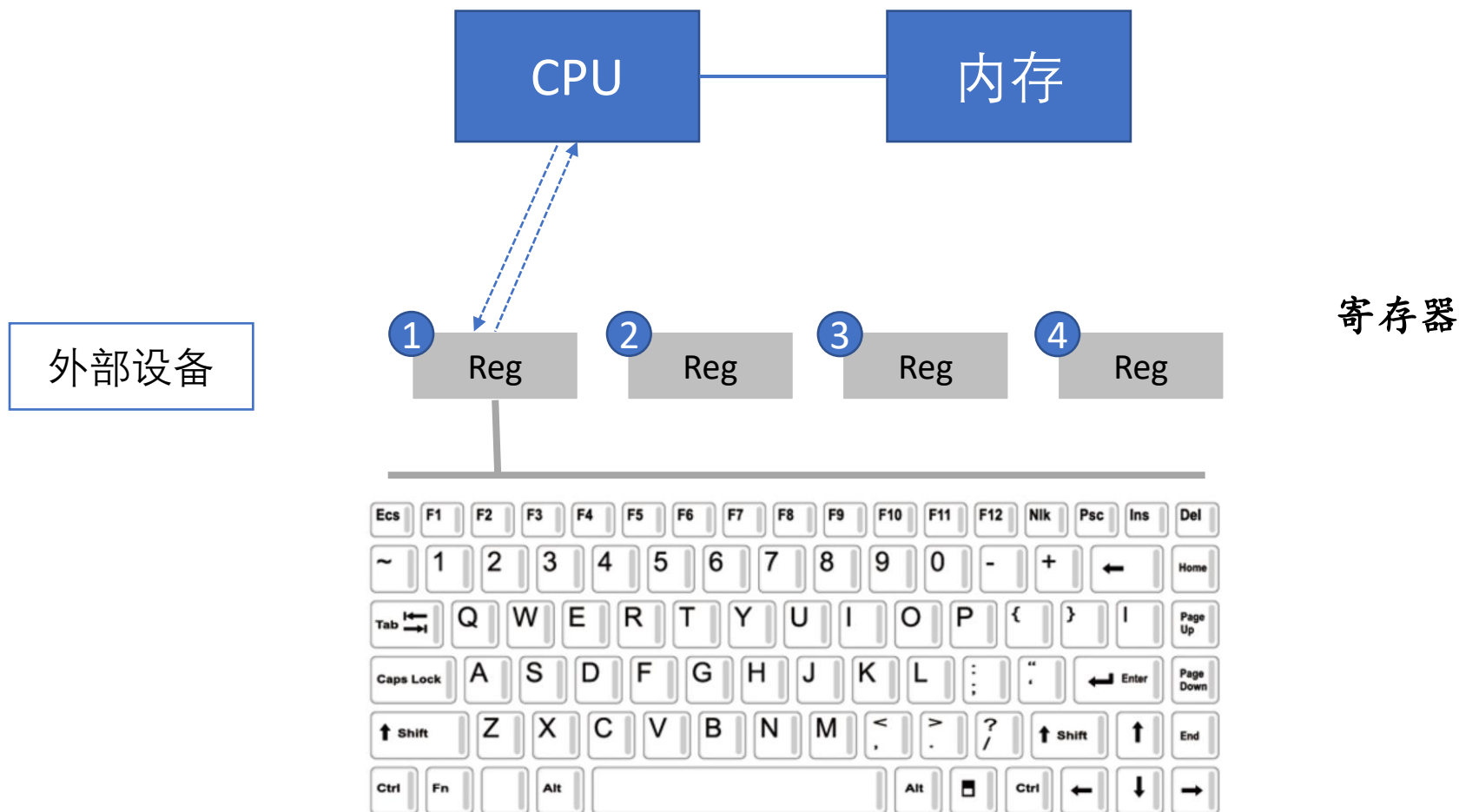


回顾：为什么会有I/O设备

1950s: 随着计算机运算速度和存储器的发展, 计算机已经快到可以“计算” 人类日常任务了。

- 计算机 “Turing Machine” 中的一切都是数据
 - 因此计算机一定会提供一个机制
 - 从设备中读取数据 (input data)
 - 键盘按键的代码、鼠标移动的偏移量、.....
 - 向设备写入数据 (output data)
 - 输出到打印机的字符串、屏幕上显示的像素.....
 - 设备可以向处理器发送中断 (后话)

Turing Machine



设备-处理器接口

设备 = 一组寄存器，每次可以交换一定数量的数据。

- 每个寄存器有特定的含义和格式
 - 主要功能：读取数据/写入数据/配置设备

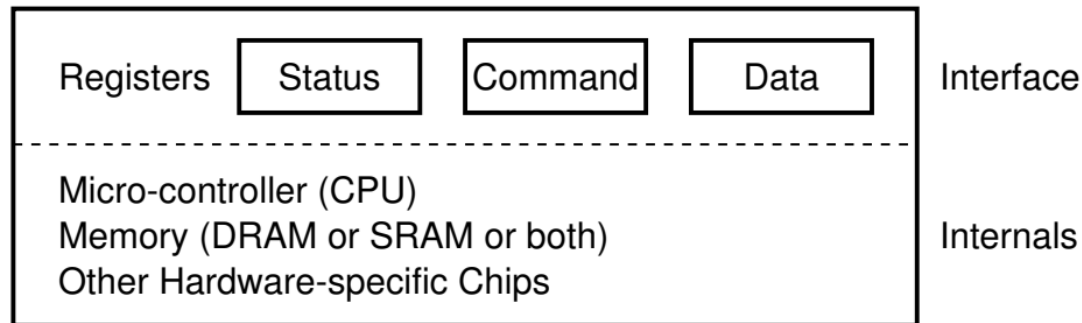
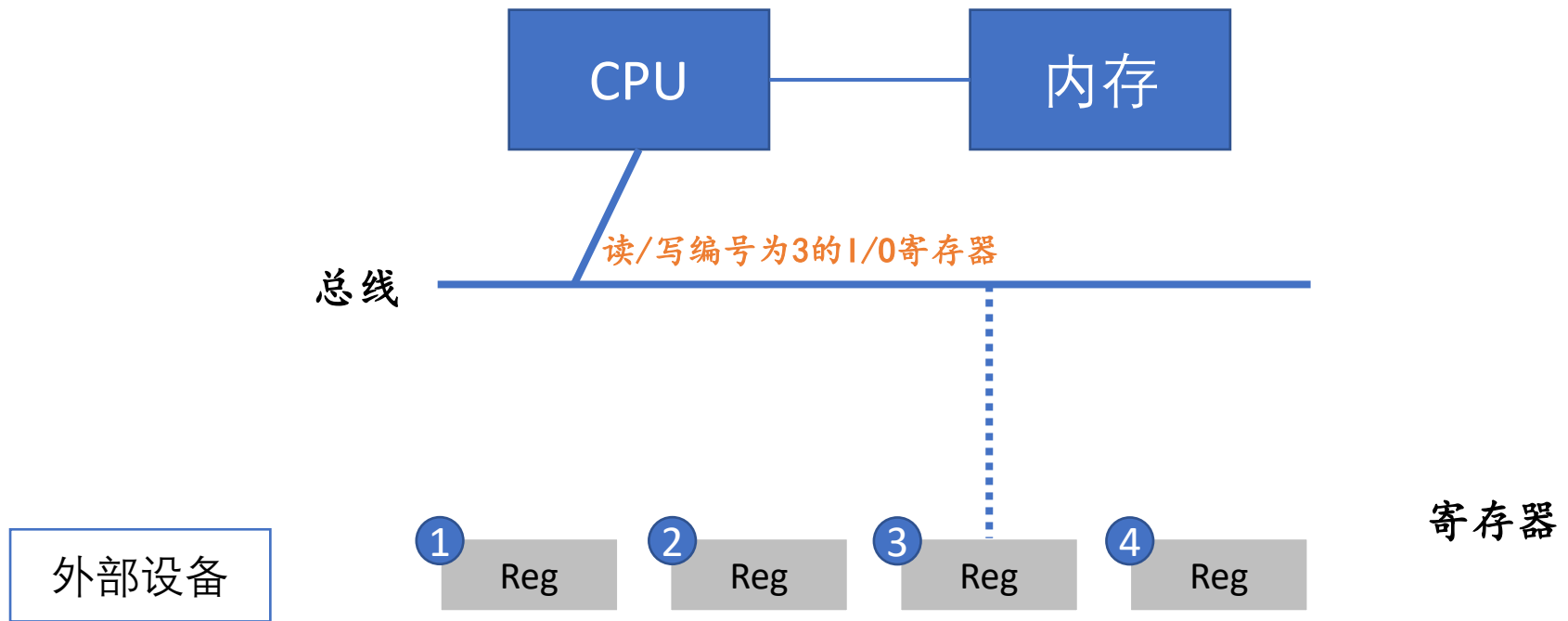
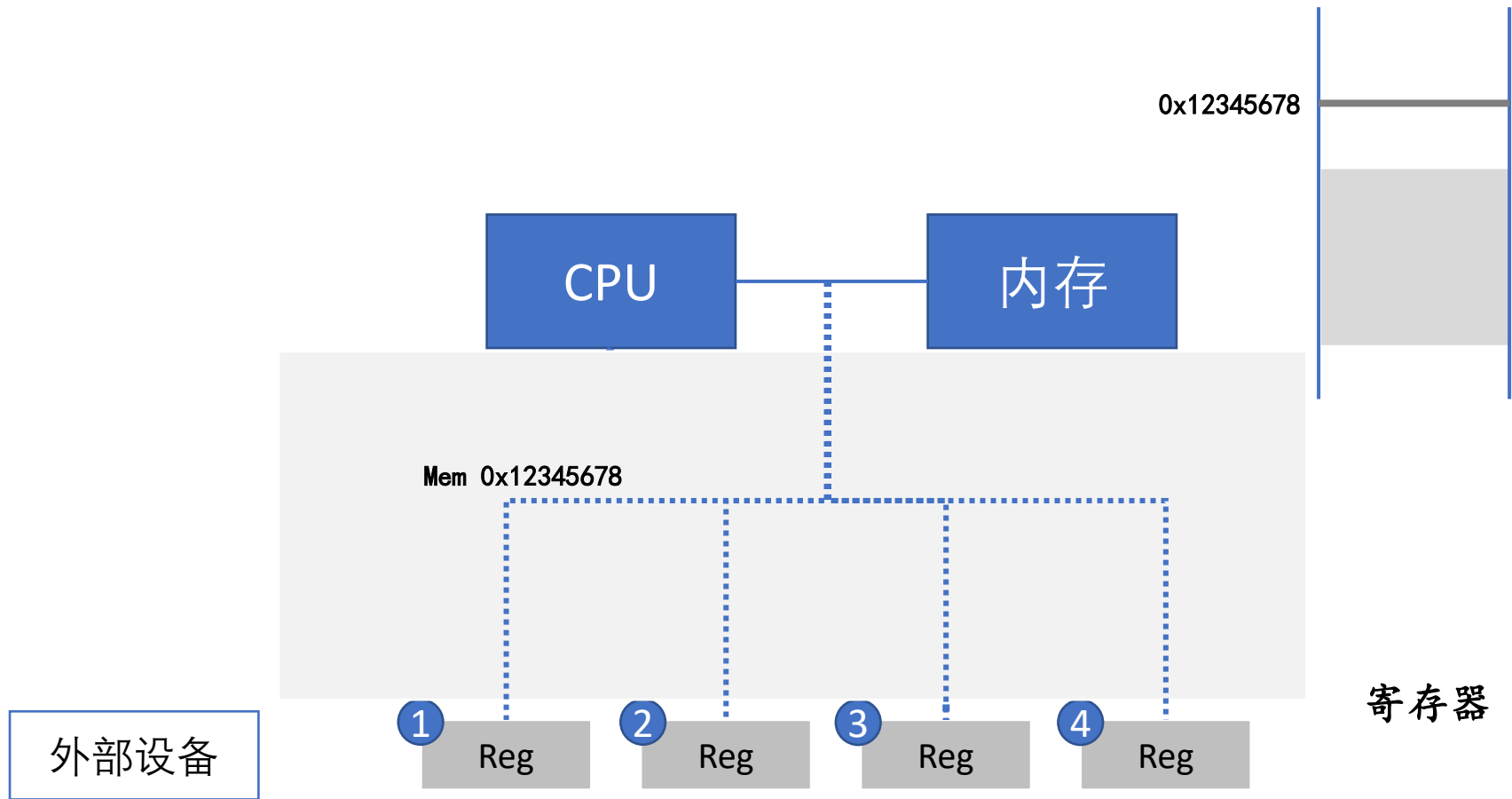


Image source: [OSTEP](#)

I/O交换方式1



I/O交换方式2

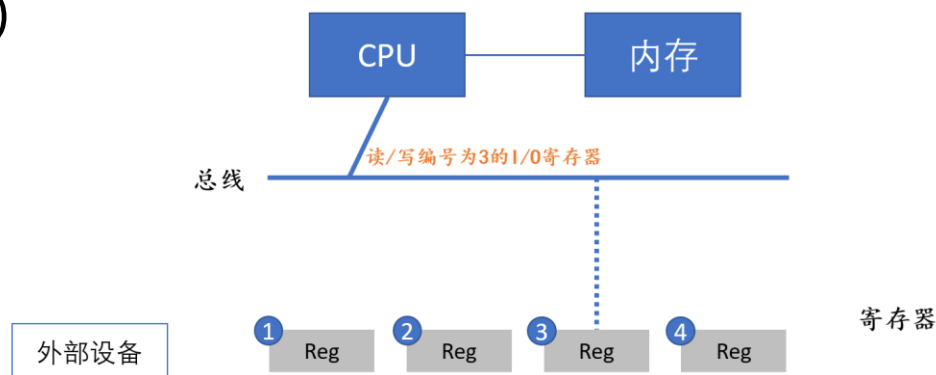


设备-处理器接口 (cont'd)

- CPU可以直接通过指令读写这些寄存器

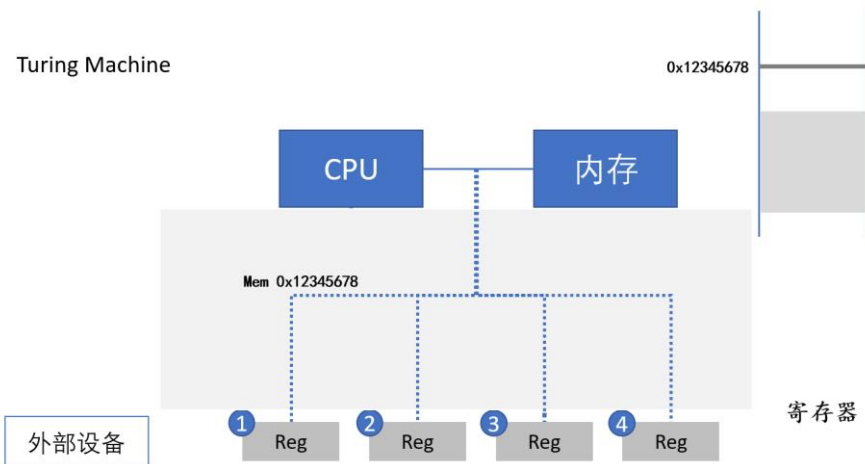
- Port-mapped I/O (PMIO)

- I/O地址空间 (port)
- CPU直连I/O总线



- Memory-mapped I/O (MMIO)

- 直观：使用普通内存读写指令就能访问
- 带来了一些设计和实现的麻烦：编译器优化、缓存、乱序执行



管理 控制 视图 热键 设备 帮助

```
1 #define ADDR 0X12345678
2
3 void foo(){
4     for (int i = 0; i<1024; i++){
5         //out(ADDR, 0);
6         (*(char *)ADDR) = 0;
7     }
8
9 }
```

~/Documents/ICS2021/teach/I0/a.c[+1] [c] unix utf-8 Ln 6 S 英 简 拼

 Right Shift + Right Alt

处理器眼中的I/O设备：x86-qemu UART

- “COM1”; putch()的实现

```
#define COM1 0x3f8

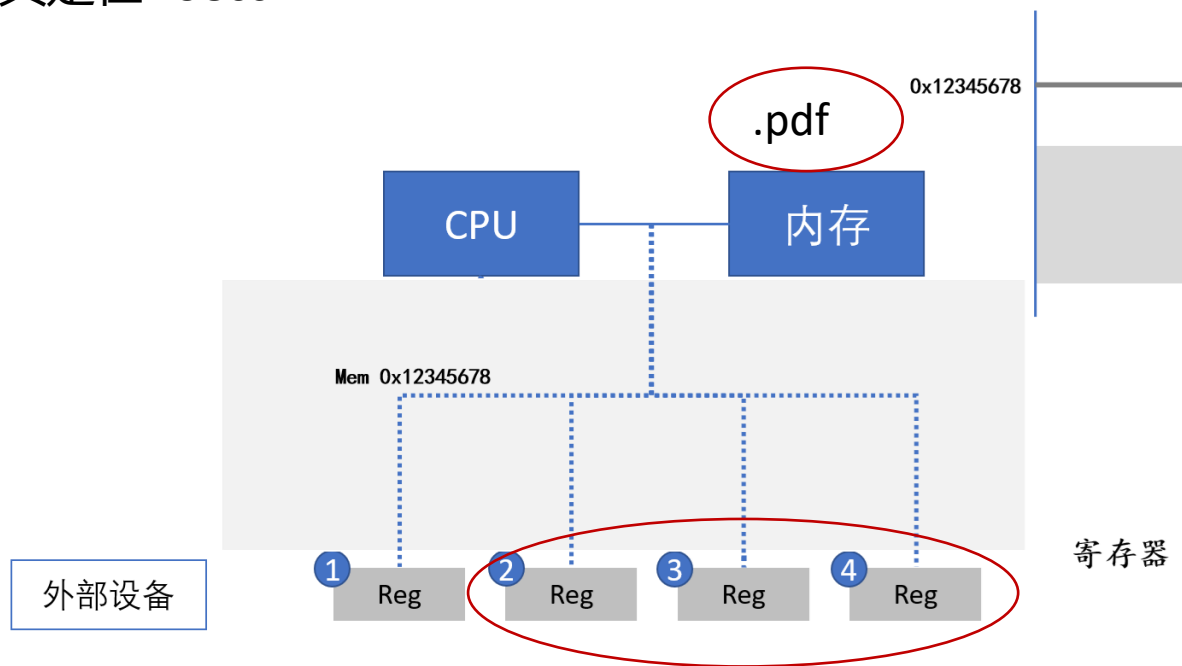
static int uart_init() {
    outb(COM1 + 2, 0); // 控制器相关细节
    outb(COM1 + 3, 0x80);
    outb(COM1 + 0, 115200 / 9600);
    ...
}

static void uart_tx(AM_UART_TX_T *send) {
    outb(COM1, send->data);
}

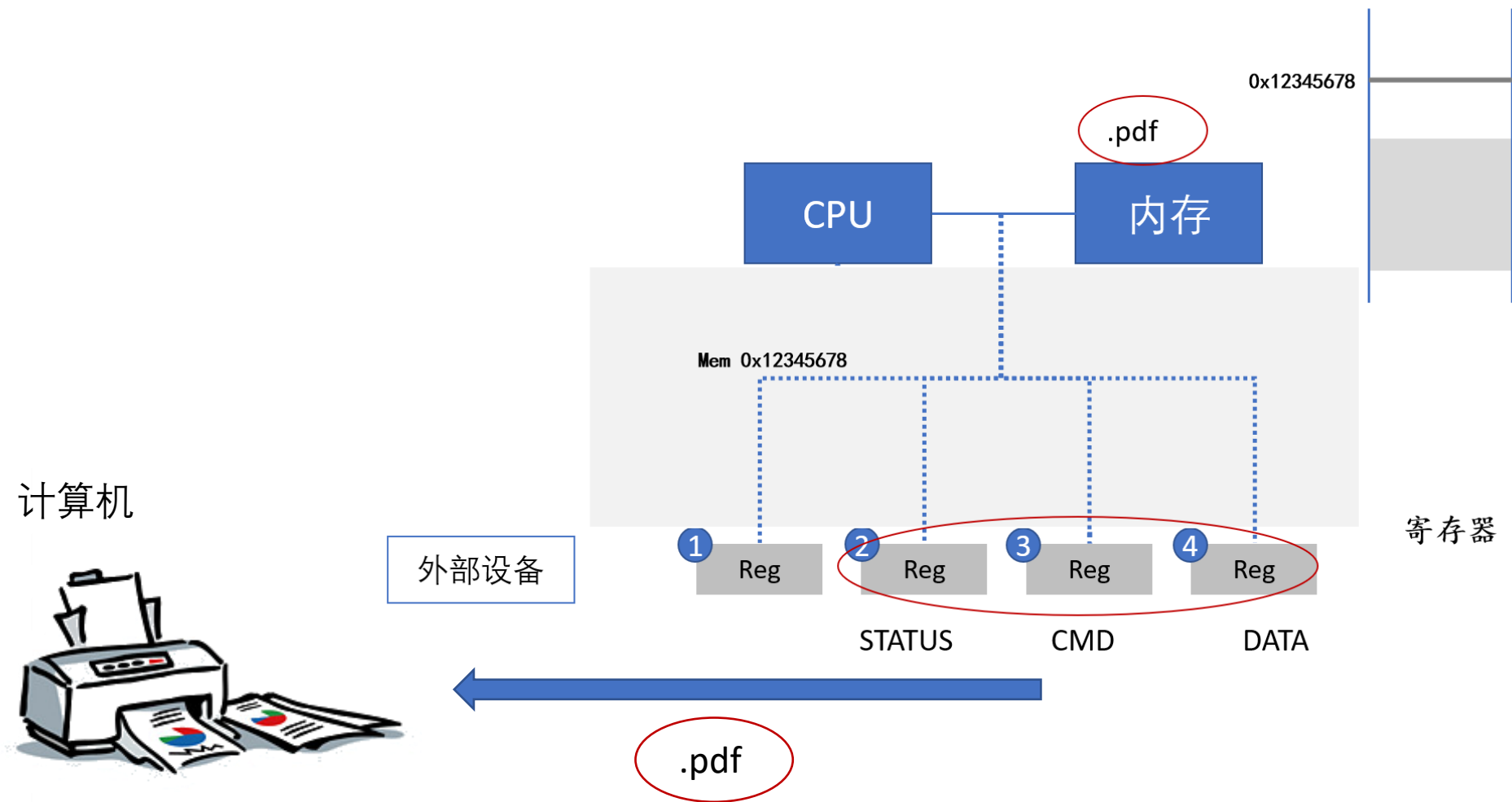
static void uart_rx(AM_UART_RX_T *recv) {
    recv->data = (inb(COM1 + 5) & 0x1) ? inb(COM1) : -1;
}
```

更复杂的设备：打印机与PostScript

- 打印机：将字节流描述的文字/图形打印到纸张上
 - 可简单 (ASCII 文本序列，像打字机一样打印)
 - 可复杂 (编程语言描述的图形)
 - 高清全页图片的传输是很大的挑战
 - 尤其是在 1980s



更复杂的设备：打印机与PostScript



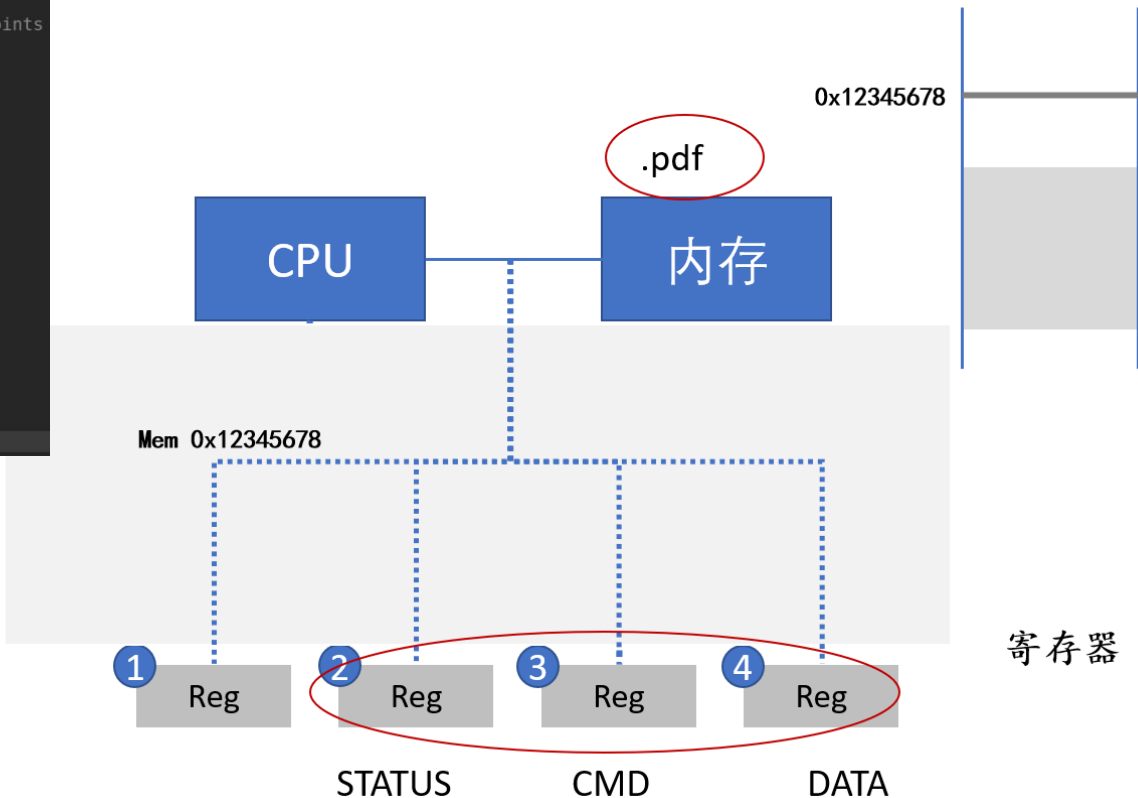
更复杂的设备：打印机与PostScript

- 打印机：将字节流描述的文字/图形打印到纸张上
 - 可简单 (ASCII 文本序列, 像打字机一样打印)
 - 可复杂 (编程语言描述的图形)
 - 高清全页图片的传输是很大的挑战
 - 尤其是在 1980s
- 例子：PostScript (1984)
 - 一种描述页面布局的 domain-specific language
 - 类似于汇编语言
 - 可以在命令行中创建高质量的文稿
 - PDF 是它的 superset
 - 例子： [page.ps](#)

\$

更复杂的设备：打印机与PostScript

```
3
4 72 72 scale % scale coordinate system so units are inches, not points
5 2 2 translate % put origin 2 inches from lower left of page
6
7 /Symbol findfont 4 scalefont setfont
8 % current font is now Symbol about 4 inches high
9 gsave % save graphics state (coordinate system & stuff)
10 .5 setgray
11 60 rotate
12 0 0 moveto (abcde) show
13 grestore % restore previous graphics state
14
15 /Helvetica-Bold findfont .2 scalefont setfont
16 % current font is now slanted Helvetica about .2 inches high
17 0 4 moveto (Postscript is good at this stuff) show
18
19 /Times-Italic findfont 2 scalefont setfont
20 % current font is now italic Times about 2 inches high
21 1 0 0 setrgbcolor
22 0 6 moveto (yowza!) show
23
24 showpage
```



计算机

.pdf

外部设备

寄存器



更复杂的设备：打印机与PostScript

- 打印机：将字节流描述的文字/图形打印到纸张上
 - 可简单 (ASCII 文本序列, 像打字机一样打印)
 - 可复杂 (编程语言描述的图形)
 - 高清全页图片的传输是很大的挑战
 - 尤其是在 1980s
- 例子：PostScript (1984)
 - 一种描述页面布局的 domain-specific language
 - 类似于汇编语言
 - 可以在命令行中创建高质量的文稿
 - PDF 是它的 superset
 - 例子： [page.ps](#)

两个特殊的I/O设备

- 总线

- 系统里可能有很多 (甚至是可变的) I/O 设备
 - 总线实现了设备的查找、映射、和命令/数据的转发
- CPU 可以只直接连接到总线
 - 总线可以连接其他总线
 - 例子: `lspci -t`, `lsusb -t`

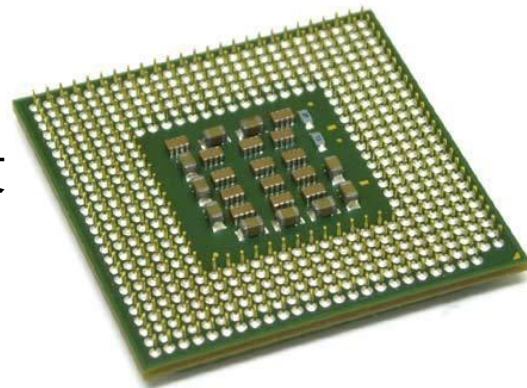
- 中断控制器

- 管理多个产生中断的设备
 - 汇总成一个中断信号给 CPU
 - 支持中断的屏蔽、优先级管理等

两个特殊的I/O设备

- 总线

- 系统里可能有很多 (甚至是可变的) I/O 设备
 - 总线实现了设备的查找、映射、和命令/数据的转发
- CPU 可以只直接连接到总线
 - 总线可以连接其他总线
 - 例子: `lspci -t`, `lsusb -t`



```
#define COM1 0x3f8

static int uart_init() {
    outb(COM1 + 2, 0); // 控制器相关细节
    outb(COM1 + 3, 0x80);
    outb(COM1 + 0, 115200 / 9600);
    ...
}

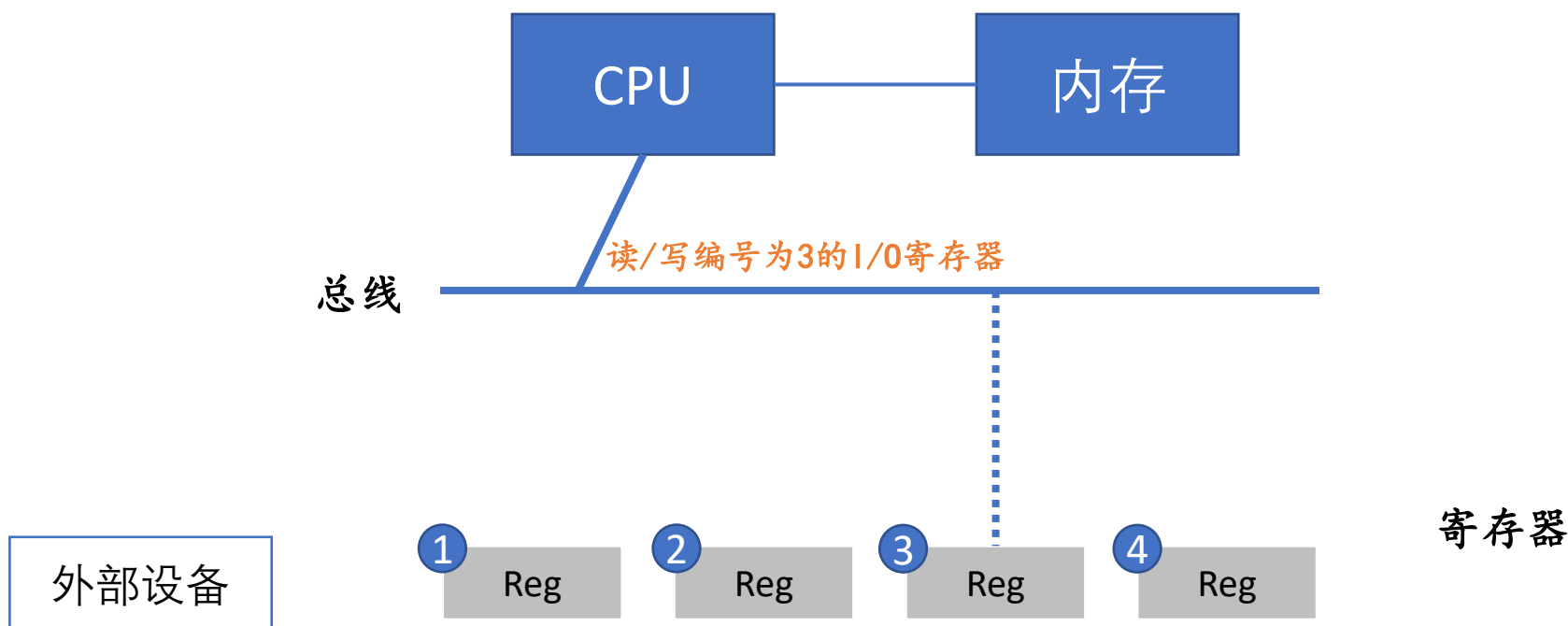
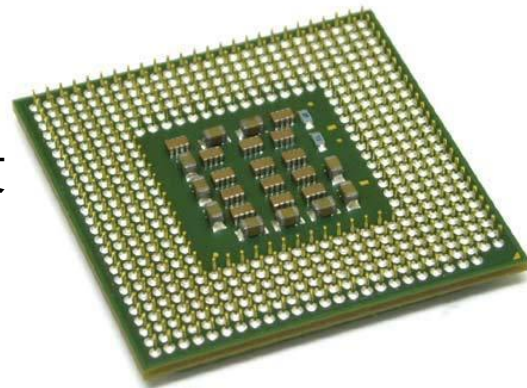
static void uart_tx(AM_UART_TX_T *send) {
    outb(COM1, send->data);
}

static void uart_rx(AM_UART_RX_T *recv) {
    recv->data = (inb(COM1 + 5) & 0x1) ? inb(COM1) : -1;
}
```

两个特殊的I/O设备

- 总线

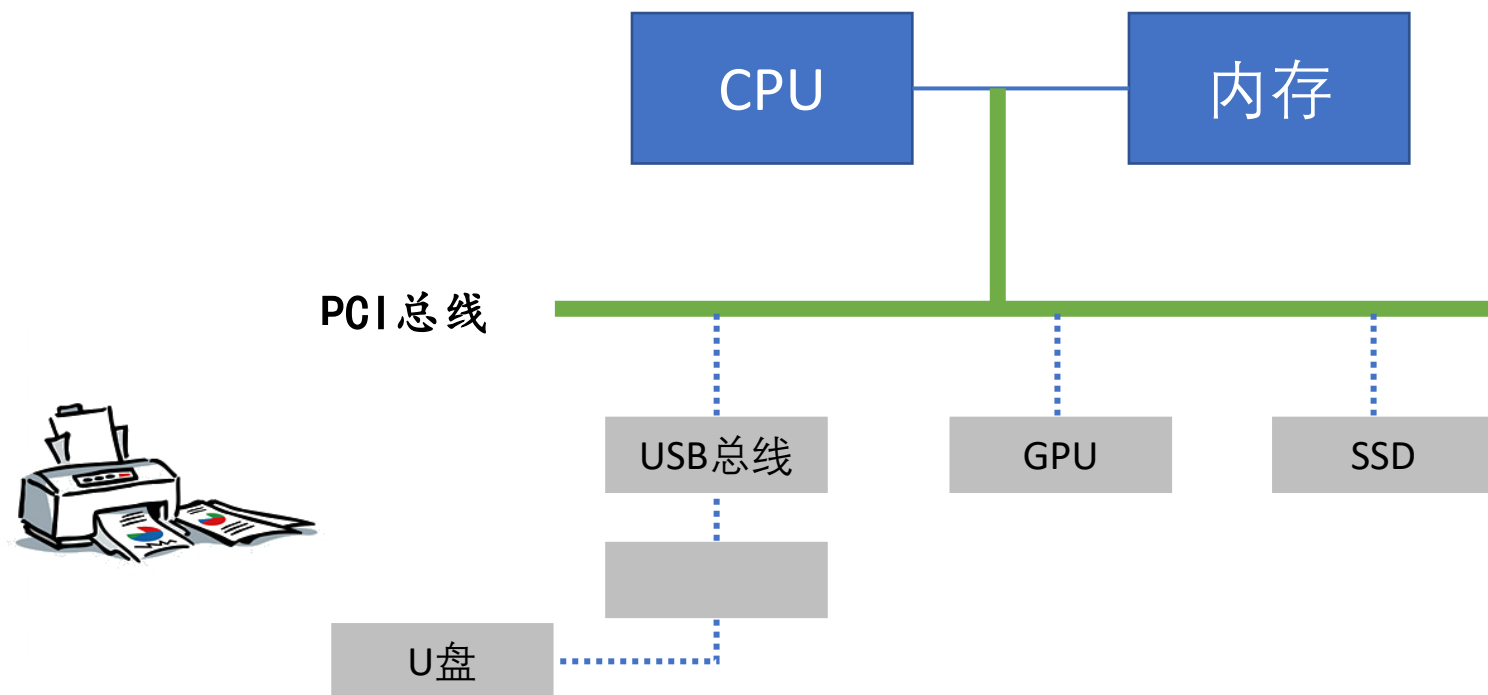
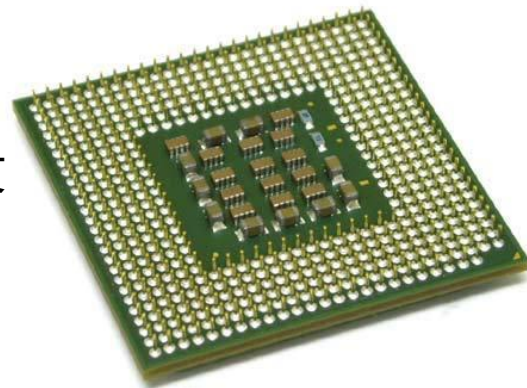
- 系统里可能有很多 (甚至是可变的) I/O 设备
 - 总线实现了设备的查找、映射、和命令/数据的转发
- CPU 可以只直接连接到总线
 - 总线可以连接其他总线
 - 例子: `lspci -t`, `lsusb -t`



两个特殊的I/O设备

- 总线

- 系统里可能有很多 (甚至是可变的) I/O 设备
 - 总线实现了设备的查找、映射、和命令/数据的转发
- CPU 可以只直接连接到总线
 - 总线可以连接其他总线
 - 例子: `lspci -t`, `lsusb -t`

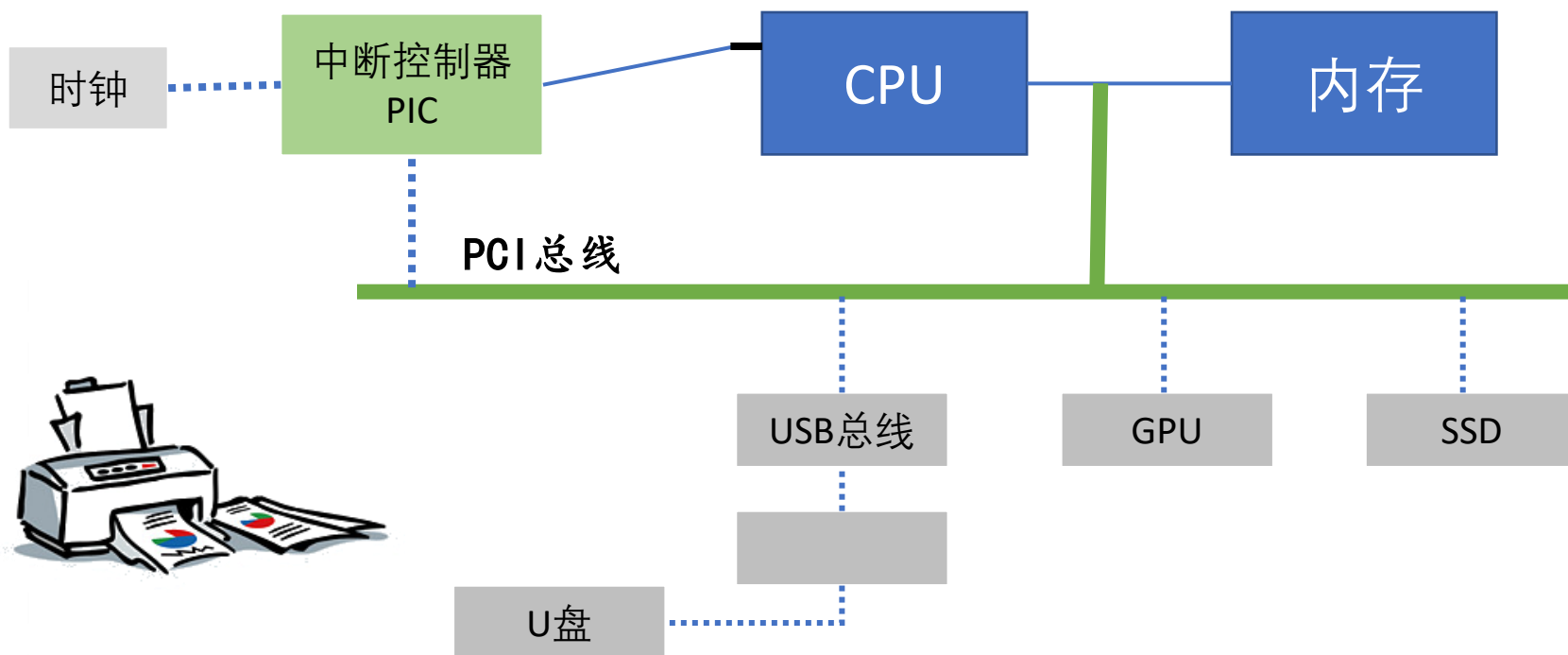


```
$ lspci
00:00.0 Host bridge: Intel Corporation 440FX - 82441FX PMC [Natoma] (rev 02)
00:01.0 ISA bridge: Intel Corporation 82371SB PIIX3 ISA [Natoma/Triton II]
00:01.1 IDE interface: Intel Corporation 82371AB/EB/MB PIIX4 IDE (rev 01)
00:02.0 VGA compatible controller: VMware SVGA II Adapter
00:03.0 Ethernet controller: Intel Corporation 82540EM Gigabit Ethernet Controller (rev 02)
00:04.0 System peripheral: InnoTek Systemberatung GmbH VirtualBox Guest Service
00:05.0 Multimedia audio controller: Intel Corporation 82801AA AC'97 Audio Controller (rev 01)
00:06.0 USB controller: Apple Inc. KeyLargo/Intrepid USB
00:07.0 Bridge: Intel Corporation 82371AB/EB/MB PIIX4 ACPI (rev 08)
00:0d.0 SATA controller: Intel Corporation 82801HM/HEM (ICH8M/ICH8M-E) SATA Controller [AHCI mode] (rev 02)
```

```
$ lsusb
Bus 001 Device 002: ID 80ee:0021 VirtualBox USB Tablet
Bus 001 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
```

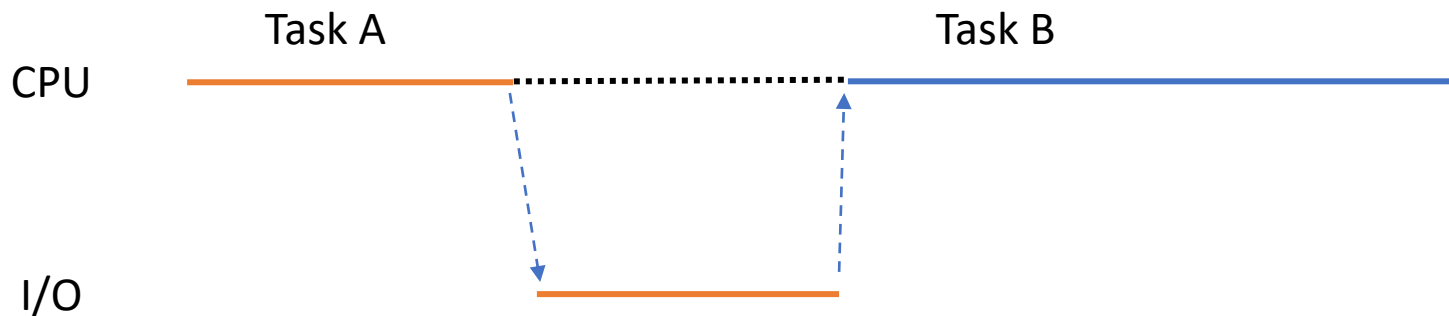

两个特殊的I/O设备

- 中断控制器
 - 管理多个产生中断的设备
 - 汇总成一个中断信号给 CPU
 - 支持中断的屏蔽、优先级管理等



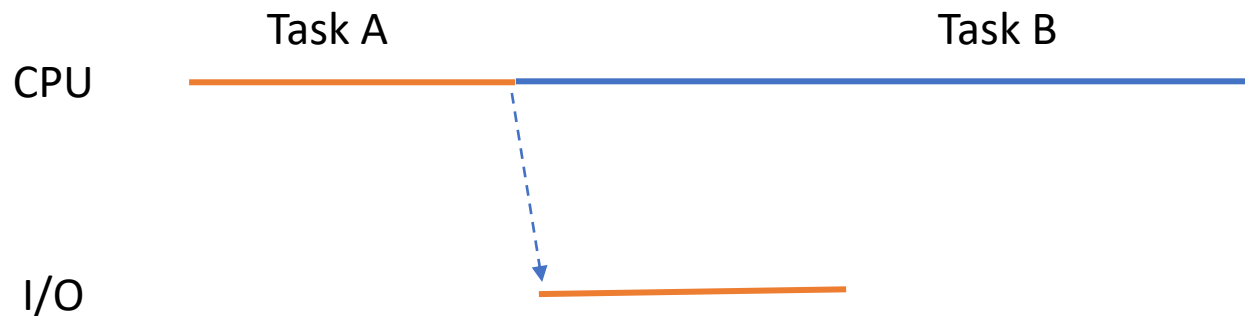
中断：弥补I/O设备的速度缺陷

- CPU cycles 实在太珍贵了
 - 不能用来浪费在等 I/O 设备完成上
 - 拥有机械部件的 I/O 设备相比于 CPU 来说实在太慢了



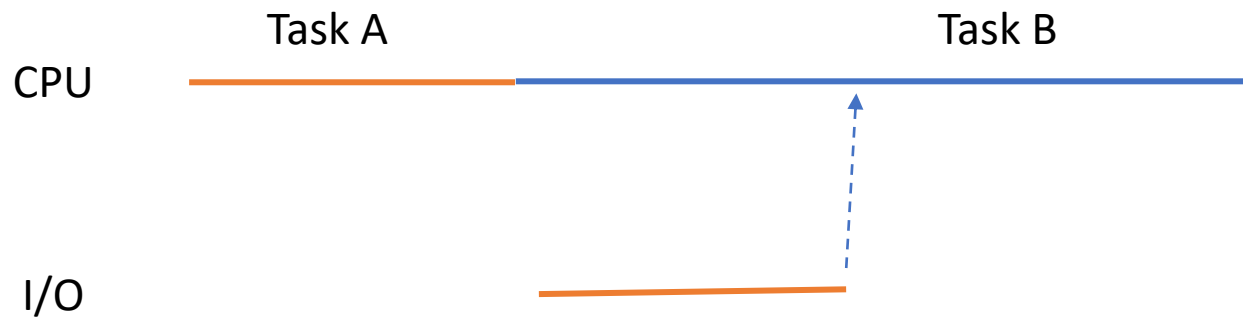
中断：弥补I/O设备的速度缺陷

- CPU cycles 实在太珍贵了
 - 不能用来浪费在等 I/O 设备完成上
 - 拥有机械部件的 I/O 设备相比于 CPU 来说实在太慢了



中断：弥补I/O设备的速度缺陷

- CPU cycles 实在太珍贵了
 - 不能用来浪费在等 I/O 设备完成上
 - 拥有机械部件的 I/O 设备相比于 CPU 来说实在太慢了



中断：弥补I/O设备的速度缺陷

- CPU cycles 实在太珍贵了
 - 不能用来浪费在等 I/O 设备完成上
 - 拥有机械部件的 I/O 设备相比于 CPU 来说实在太慢了
- 中断 = 硬件驱动的函数调用
 - 相当于在每条语句后都插入

```
if (pending_io && int_enabled) {  
    interrupt_handler();  
    pending_io = 0;  
}
```

- (硬件上好像不太难实现)
 - 于是就有了最早的操作系统：管理I/O设备的库代码

处理器-设备接口：NES实现 (1983)

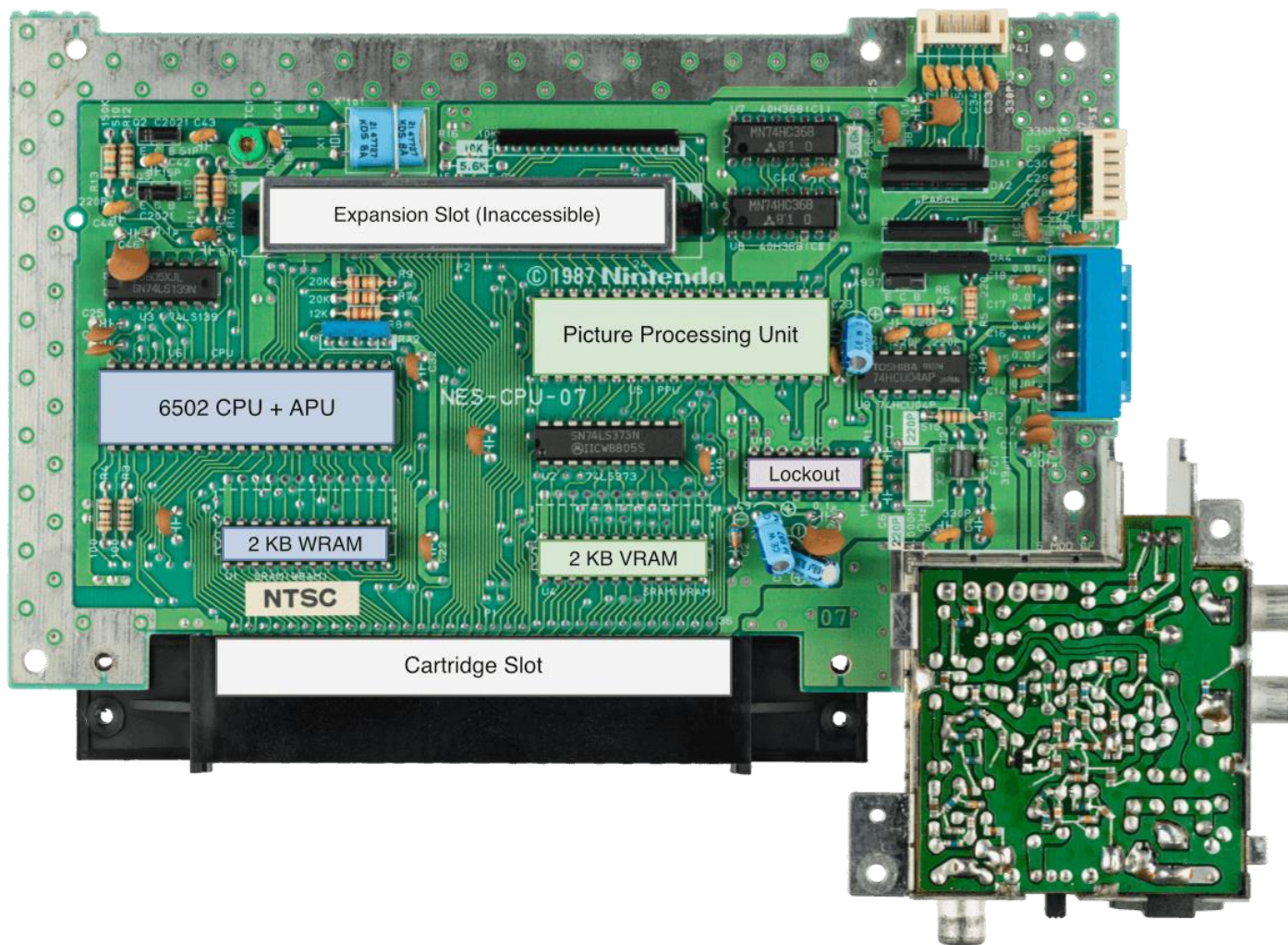
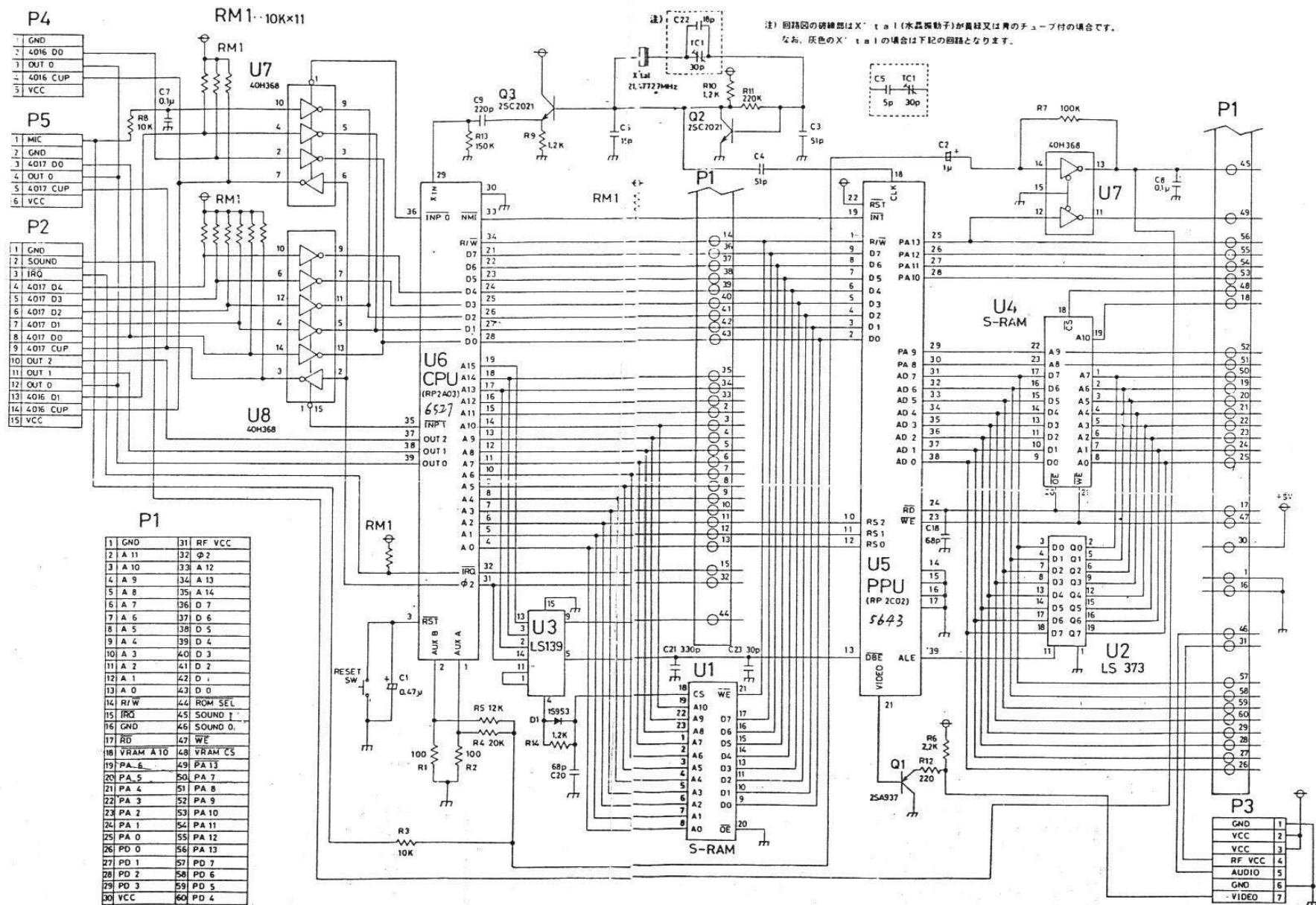


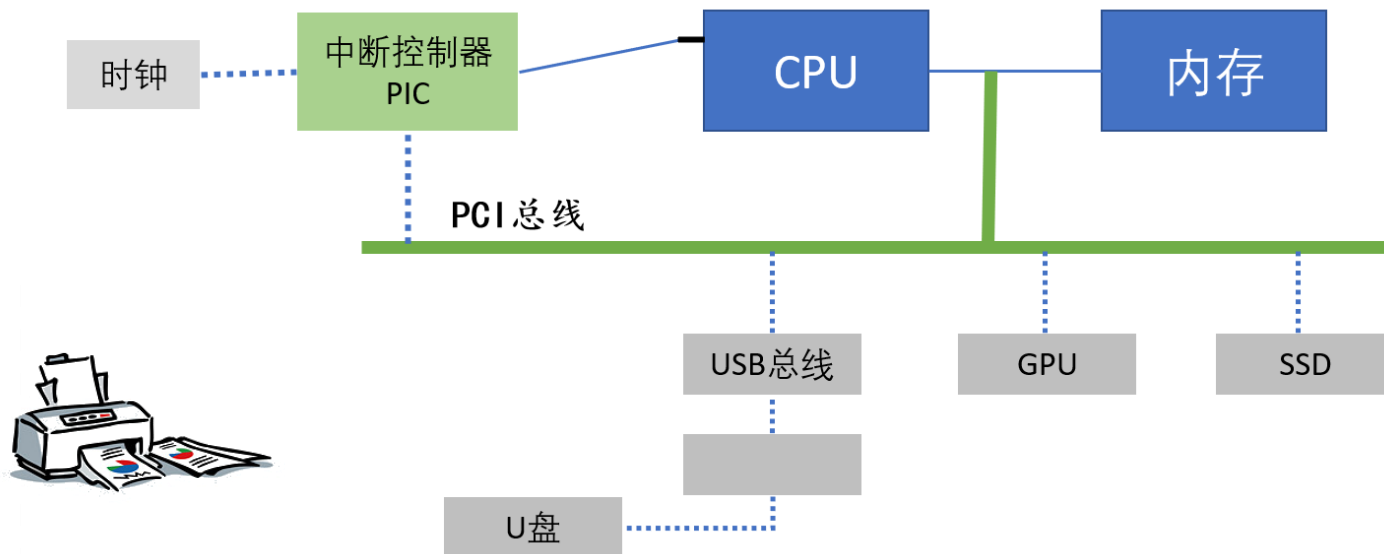
Image by [Rodrigo Copetti](#); 以及[电路图](#)

CPU基板回路図

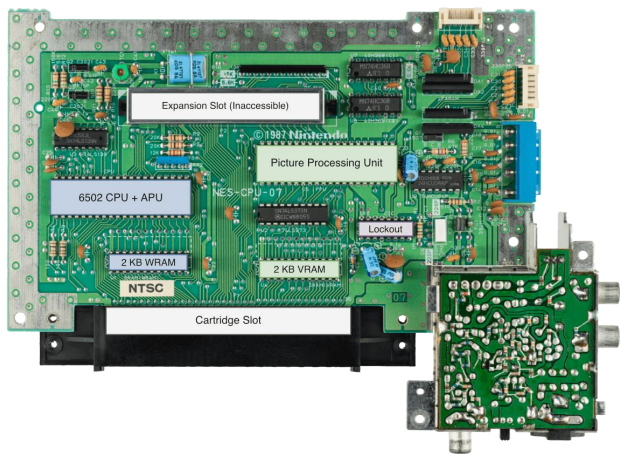


小结

- I/O 设备：“小计算机”
 - 完成和物理世界的交互功能
 - 连接到总线/中断控制器
 - CPU 通过 PIO/MMIO 访问
- 今天很多 I/O 设备都带有或简单或复杂的 CPU
 - 带跑马灯/编程功能的键盘和鼠标
 - 显示加速器和 “显卡计算”



2D图形绘制硬件



理论：一切皆可“计算”

```
for (int i = 1; i <= H; i++) {  
    for (int j = 1; j <= W; j++)  
        putchar(j <= i ? '*' : ' ');  
    putchar('\n');  
}
```

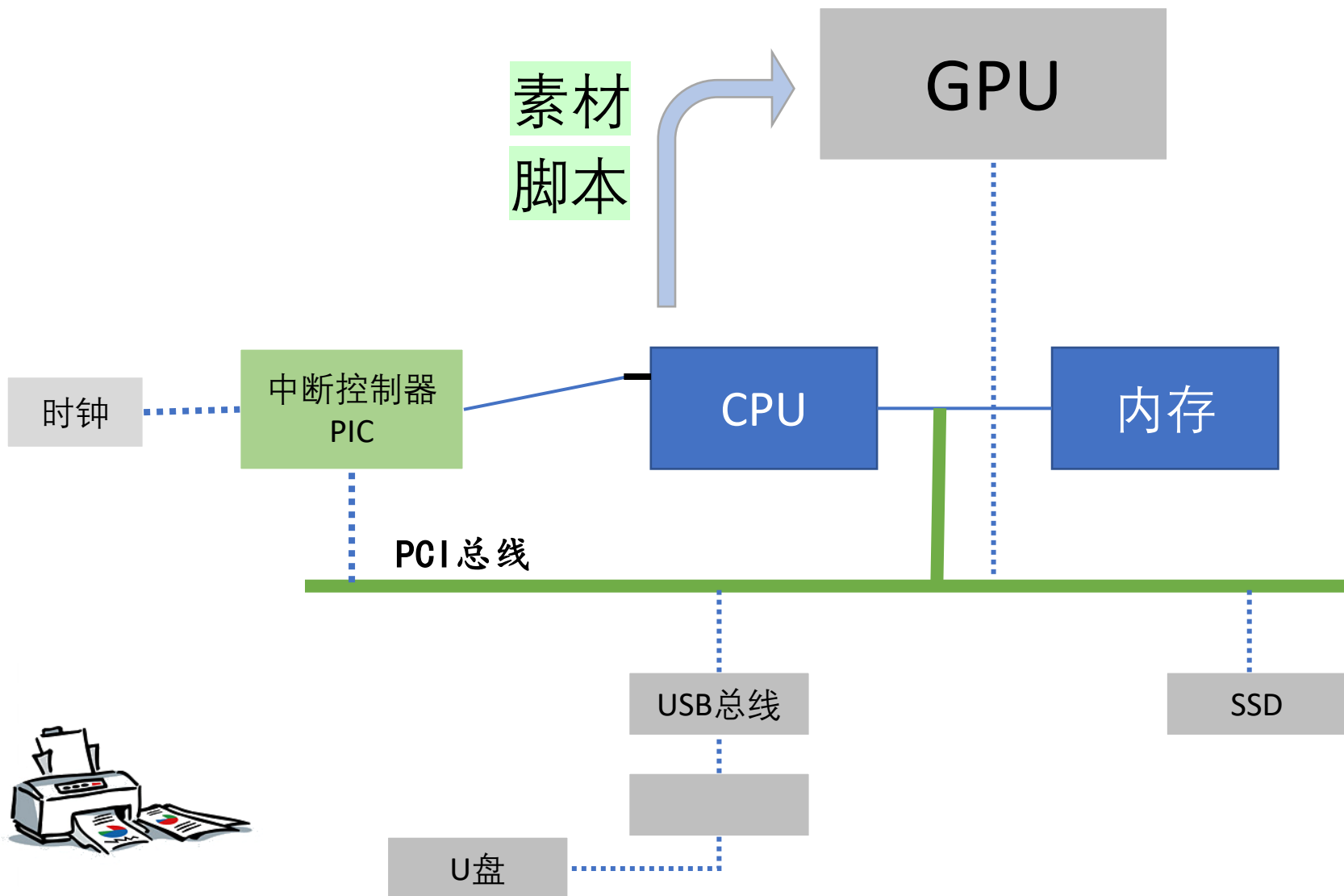


```
*  
* *  
* * *
```

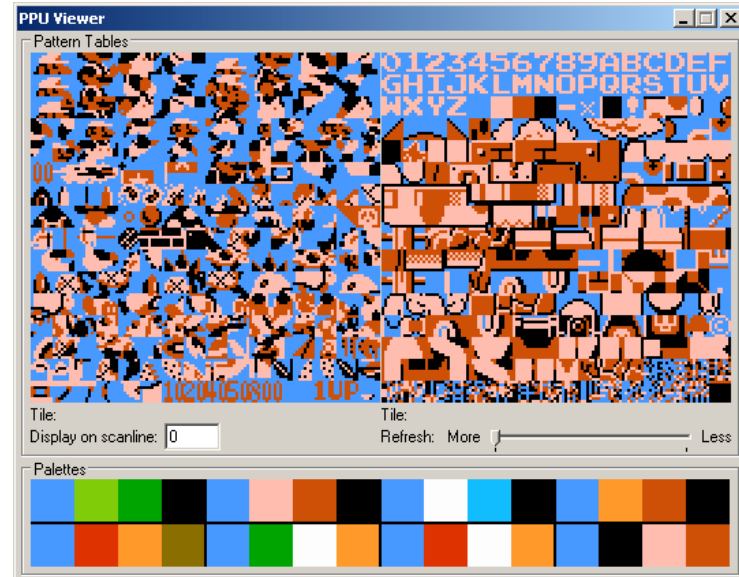
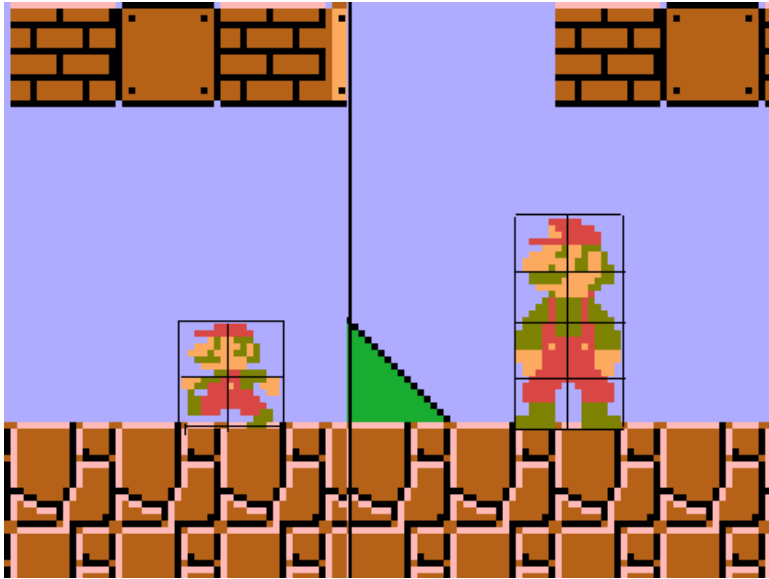
- 难办的是性能

- NES: 6502 @ 1.79Mhz; IPC = 0.43
 - 屏幕共有 $256 \times 240 = 61\text{K}$ 像素 (256 色)
 - 60FPS $\rightarrow$ 每一帧必须在 $\sim 10\text{K}$ 条指令内完成
 - 如何在有限的 CPU 运算力下实现 60Hz?

画什么？怎么画？



PR2C02 Picture Processing Unit (PPU)



- CPU 只描述 8x8 “贴块” 的摆放方法
- 类似于 PostScript 脚本
 - 背景是 “大图” 的一部分
 - 每行的前景块不超过 8 个
- PPU 完成图形的绘制

76543210

|||||||

|||||++- Palette

|||+++-- Unimplemented

||+----- Priority

|+----- Flip horizontally

+----- Flip vertically

PPU Viewer



Pattern Tables



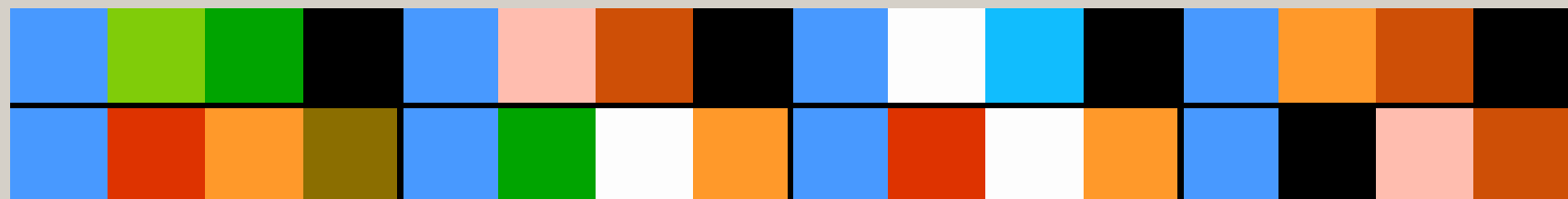
Tile:

Display on scanline:

Tile:

Refresh: More Less

Palettes

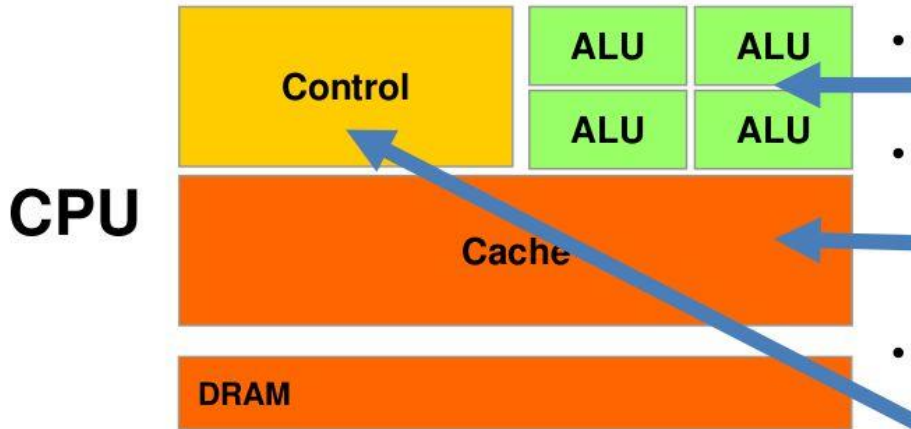


PPU: 只执行一个程序的CPU

```
for (int x = 0, pos = 0; x < HEIGHT; x++) { // 行扫描
    for (int y = 0; y < WIDTH; y++, pos++) {
        vbuf[pos] = draw(x, y); // 算出 (x,y) 的贴块 (和颜色)
    }
}
```

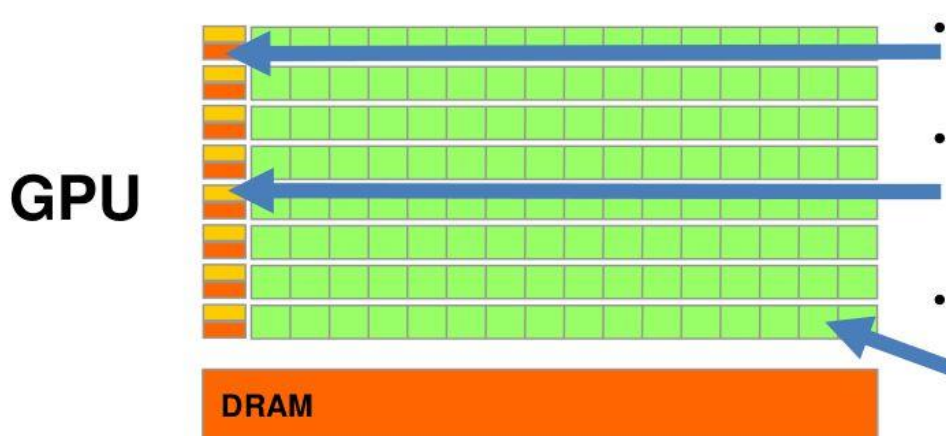
地址空间	大小	功能
\$0000-\$1FFF	8 KB	Pattern tables
\$2000-\$2FFF	4 KB	Nametables
\$3000-\$3EFF	3.75KB	Mirrors of \$2000-\$2EFF
\$3F00-\$3F1F	32B	Palette RAM indexes
\$3F20-\$3FFF	224B	Mirrors of \$3F00-\$3F1F
OAM (DMA 访问)	256B	Sprite Y, #, attribute, X

CPU: Latency Oriented Design



- Powerful ALU
 - Reduced operation latency
- Large caches
 - Convert long latency memory accesses to short latency cache accesses
- Sophisticated control
 - Branch prediction for reduced branch latency
 - Data forwarding for reduced data latency

GPUs: Throughput Oriented Design

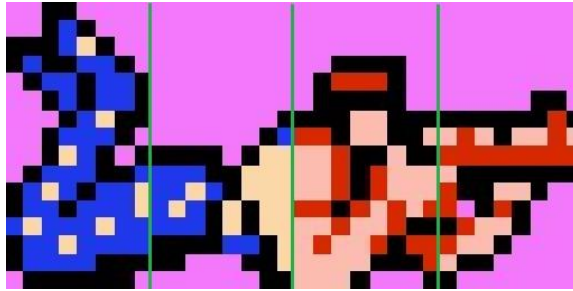


- Small caches
 - To boost memory throughput
- Simple control
 - No branch prediction
 - No data forwarding
- Energy efficient ALUs
 - Many, long latency but heavily pipelined for high throughput
- Require massive number of threads to tolerate latencies

在受限的机能下提供丰富的图片

- 前景：

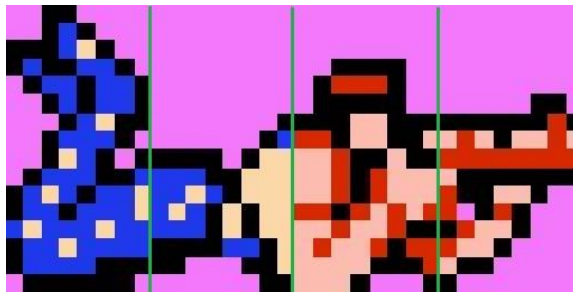
- 《魂斗罗》（Contra）中角色为什么要「萝莉式屈腿俯卧」？



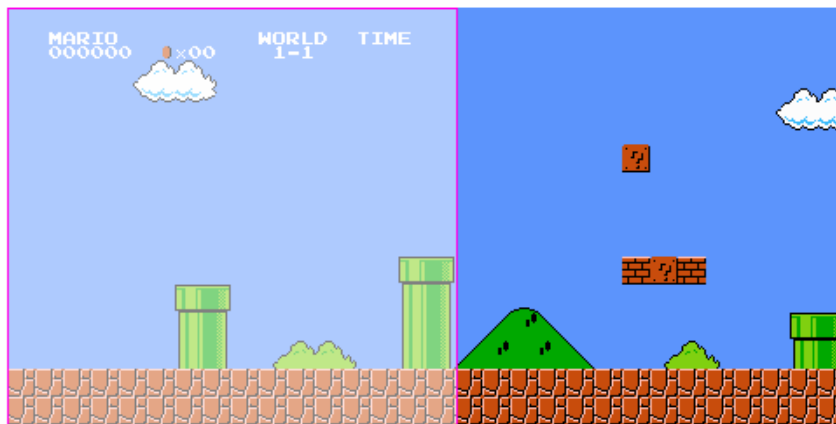
在受限的机能下提供丰富的图片

- 前景：

- 《魂斗罗》（Contra）中角色为什么要「萝莉式屈腿俯卧」？

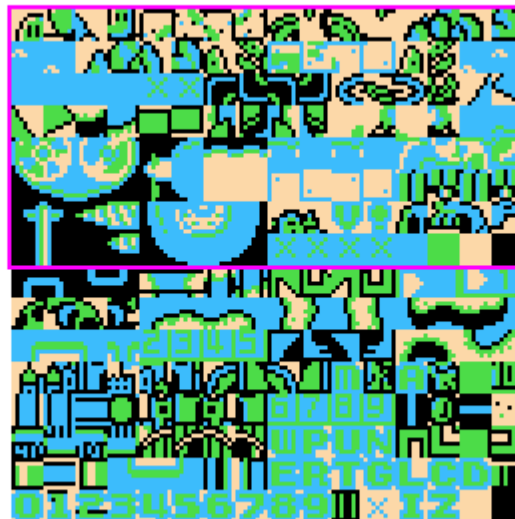
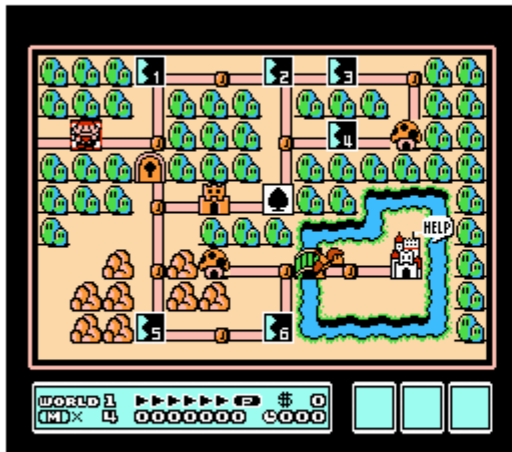


- 背景：“卷轴”操作

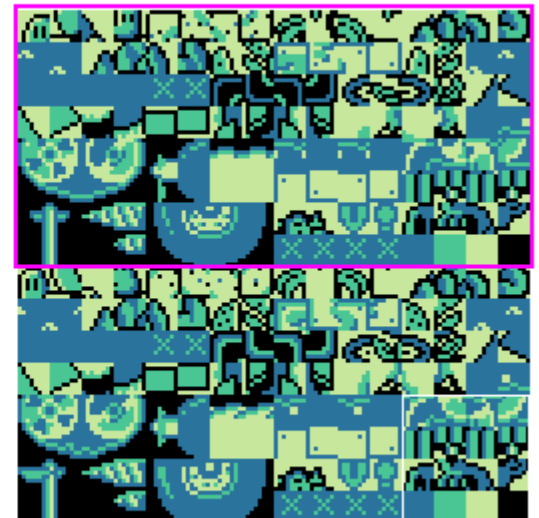


在受限的硬件下做游戏：背景动画

- 通过切换 tile pattern table 实现背景动画
 - 同时更新屏幕上的所有 tiles
 - 难怪为什么有些“次世代”的游戏画面那么精良
 - 更大的存储
 - 专有的硬件 (图形甚至整个计算系统)
 - 例子: [GUN-NAC](#) (1990)



\$05C813



NEXOFT

~GunNac~

PLAYED BY Valis77

WWW.LONGPLAYS.ORG

处理器-设备接口：NES实现 (1983)

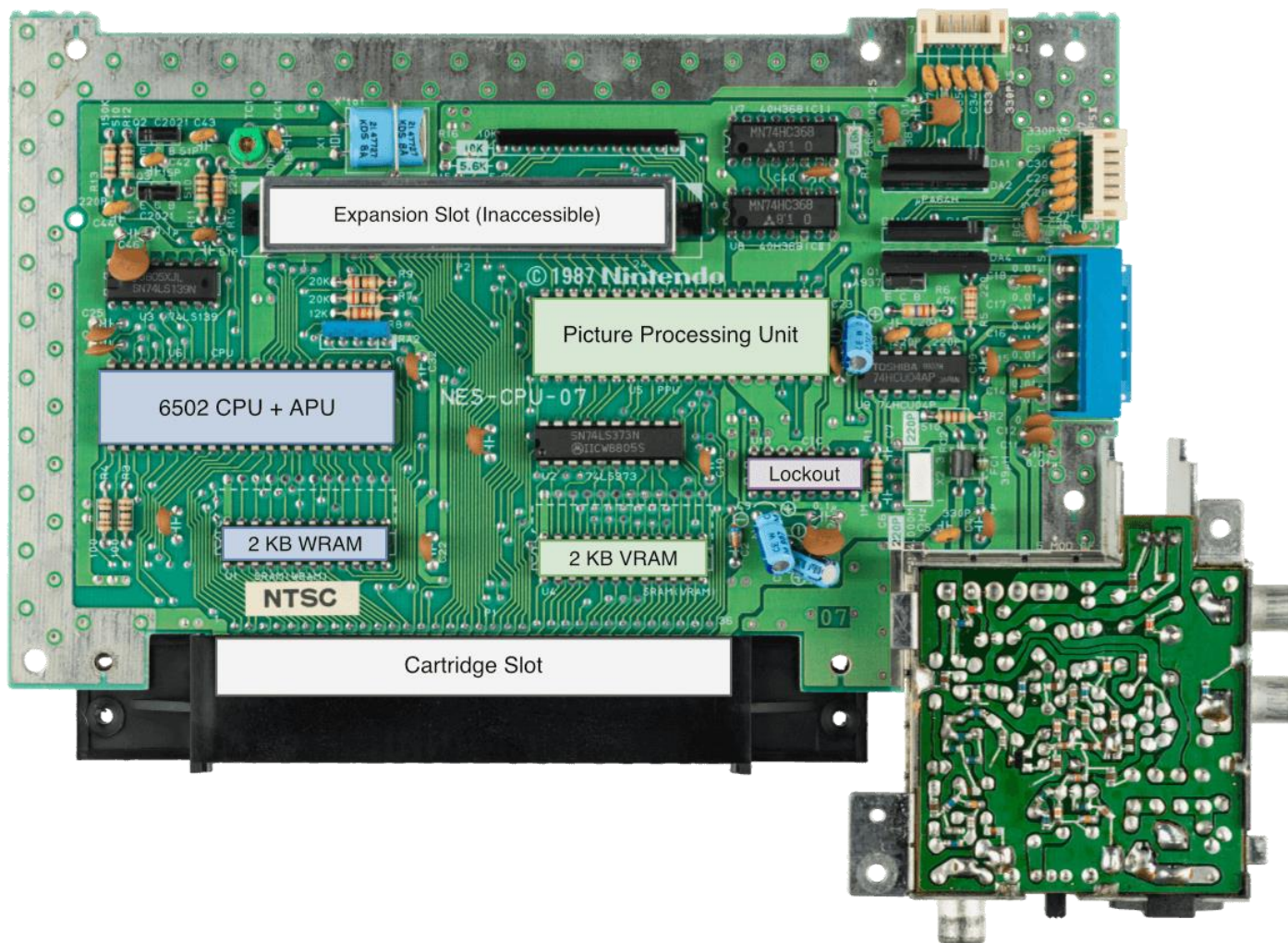
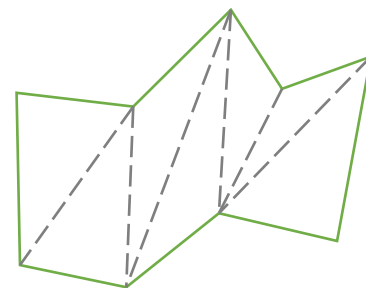
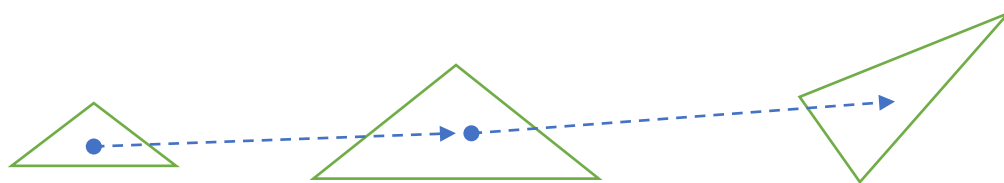


Image by [Rodrigo Copetti](#); 以及[电路图](#)

3D图形绘制硬件

更好的 2D 游戏引擎

- 如果我们有更多的晶体管？
 - NES PPU 的本质是和坐标轴平行的“贴块块”
 - 实现上只需要加法和位运算
 - 更强大的计算能力 = 更复杂的图形绘制
- 2D 图形加速硬件：图片的“裁剪” + “拼贴”
 - 支持旋转、材质映射 (缩放)、后处理、.....



- 实现 3D
 - 三维空间中的多边形，在视平面上也是多边形
 - 任何 n 边形都可以分解成 $n-2$ 个三角形

以假乱真的剪贴3D

- GameBoy Advance
 - 4 层背景; 128 个剪贴 objects; 32 个 affine objects
 - CPU 给出描述; GPU 绘制 (执行 “一个程序” 的 CPU)



([V-Rally](#); Game Boy Advance, 2002)



Licensed by
NINTENDO



00:05 / 06:09



自动

倍速



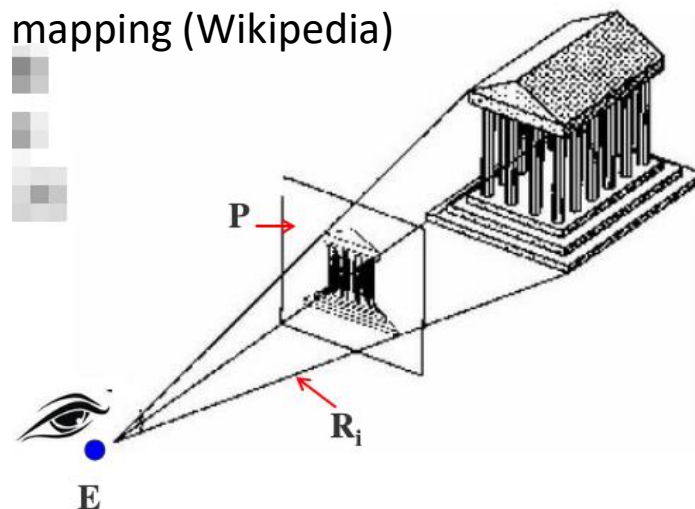
但我们还是需要真正的3D

- 三维空间中的三角形需要正确渲染



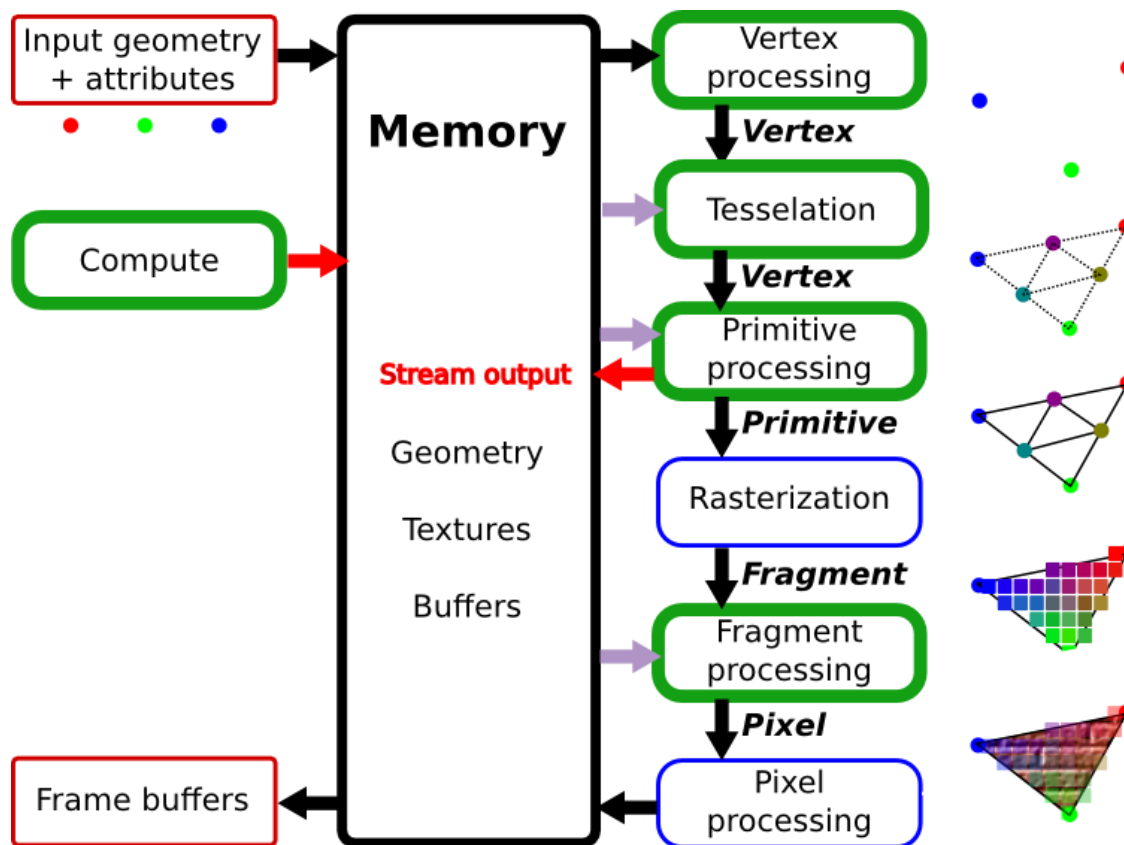
“Perspective correct” texture mapping (Wikipedia)

- 我们不要纸片人
 - 我们要真正的 3D
 - 要数百万个多边形构造的精细场景



现代 3D 图形

- CPU 负责描述, GPU 负责渲染
 - GPU 收到代码 + 数据 → 运算 → 写回结果到内存 → 发送中断
 - GPU 中有数千个运算单元实现并行的计算



理论：一切皆可“计算”

```
for (int i = 1; i <= H; i++) {  
    for (int j = 1; j <= W; j++)  
        putchar(j <= i ? '*' : ' ');  
    putchar('\n');  
}
```



- 难办的是性能

- NES: 6502 @ 1.79Mhz; IPC = 0.43
 - 屏幕共有 $256 \times 240 = 61\text{K}$ 像素 (256 色)
 - 60FPS → 每一帧必须在 $\sim 10\text{K}$ 条指令内完成
 - 如何在有限的 CPU 运算力下实现 60Hz?

现代 3D 图形：硬件上的实现

```
__global__  
void hello(char *a, char *b) {  
    a[threadIdx.x] += b[threadIdx.x];  
}  
  
char a[N] = "Hello ";  
char b[N] = {15, 10, 6, 0, -11, 1};  
cudaMalloc( (void**)&ad, N );  
cudaMalloc( (void**)&bd, N );  
cudaMemcpy( ad, a, N, cudaMemcpyHostToDevice );  
cudaMemcpy( bd, b, N, cudaMemcpyHostToDevice );  
printf("%s", a); // Hello  
dim3 dimBlock( blocksize, 1 );  
dim3 dimGrid( 1, 1 );  
hello<<<dimGrid, dimBlock>>>(ad, bd); // run on GPU  
cudaMemcpy( a, ad, N, cudaMemcpyDeviceToHost );  
printf("%s\n", a); // World!  
...
```

题外话：如此丰富的图形是怎么来的？



答案：全靠 PS (后处理)

- 例子：GLSL (Shading Language)
 - 使 “shader program” 可以在 GPU 上执行
 - 可以作用在各个渲染级别上：vertex, fragment, pixel shader
 - 相当于一个 “PS” 程序，算出每个部分的光照变化
 - 全局光照、反射、阴影、环境光遮罩.....



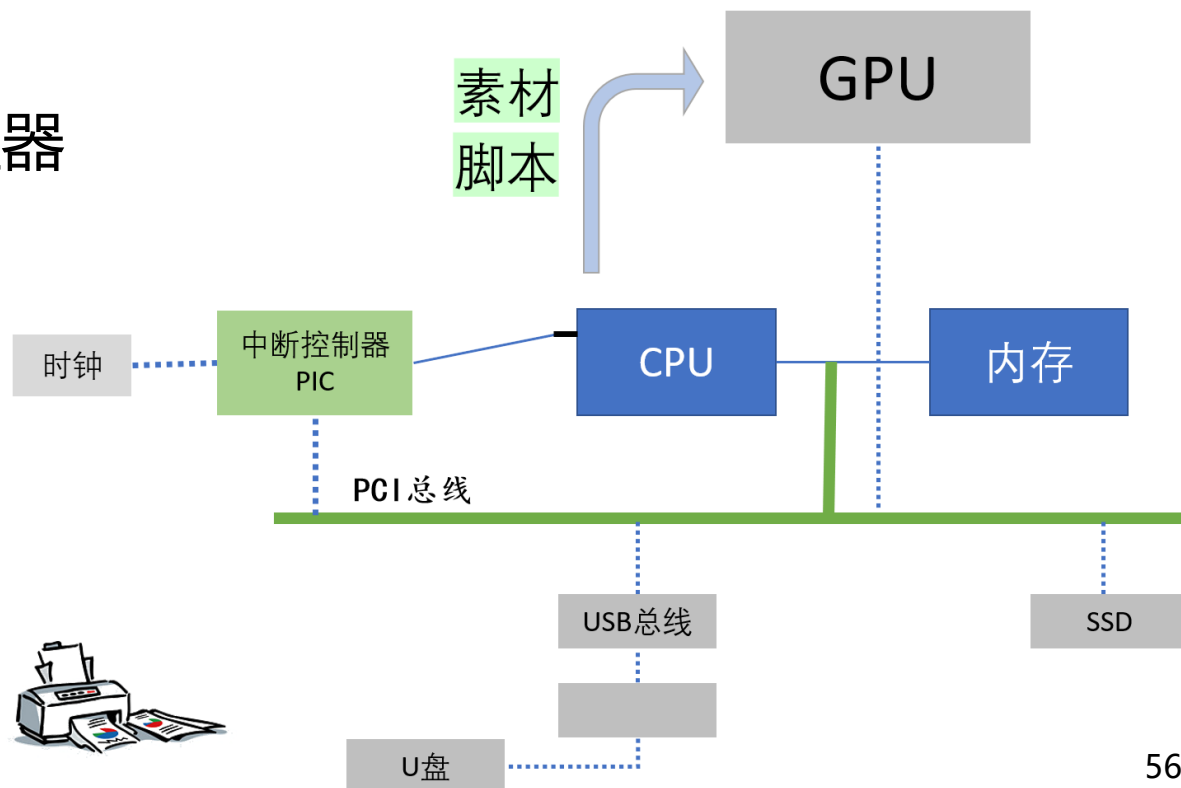
Photo by [Rob Lim](#)

总结

I/O 设备

任何能与 CPU 交换数据的“东西”。

- 完成与物理世界的交互
 - 键盘、鼠标、打印机.....
- 效率更高的专用处理器
 - GPU, NPU, FPGA, ...



复杂系统的构造和解释

《计算机系统基础》到底学什么？

一句话的 take-away message: “计算机系统是一个状态机”。

- 更具体一点？
 - 状态机的状态是什么？
 - 内存 + 寄存器 + (外部设备状态)
 - 状态机的行为是什么？
 - 取指令 + 译码 + 执行
 - 输入/输出设备访问
 - 中断 + 异常控制流
 - 地址转换
- (是否感到有点驾驭不了？)

复杂中隐藏的秩序

- 理解/构造一个复杂系统 (操作系统/处理器/航母)?
 - USS Midway (CV-41); 1945 年首航, “沙漠风暴” 行动后退役
 - 舰船配置; 资源管理/调度; 容错.....



(picture: www.seaforces.org; familyvacationhub.com)

复杂中隐藏的秩序 (cont'd)

航母不是一天造成的。



(picture: history.navy.mil)

复杂中隐藏的秩序 (cont'd)

采矿船继承了航海时代的设计。



(picture: history.navy.mil)

复杂中隐藏的秩序 (cont'd)

复杂系统的演化通常是*evolutionary*的而不是*revolutionary*的。

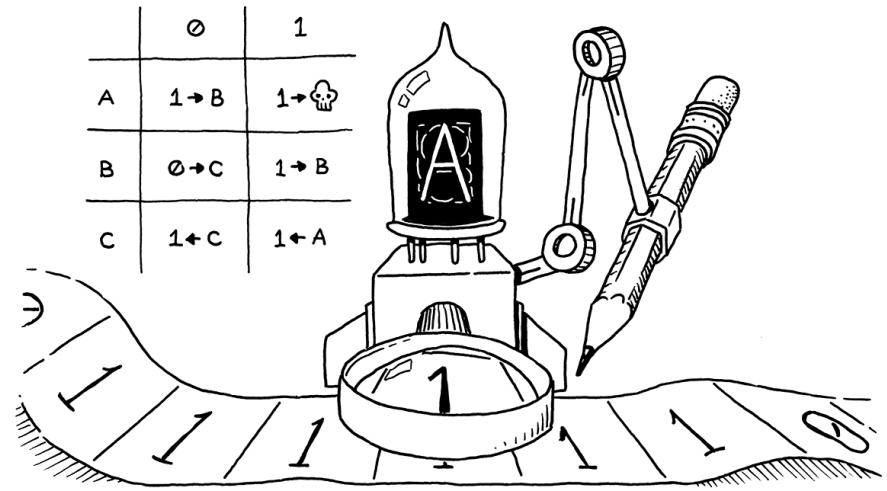
- 无法第一次就设计出 “绝对完美” 的复杂系统
 - (因为环境一直在变)
- 实际情况：从 minimal, simple, and usable 的系统不断经过 local modifications (trial and errors)
 - 计算机硬件
 - 操作系统
 - 编译器/程序设计语言
 - 需求和系统设计/实现螺旋式迭代

理解计算机系统

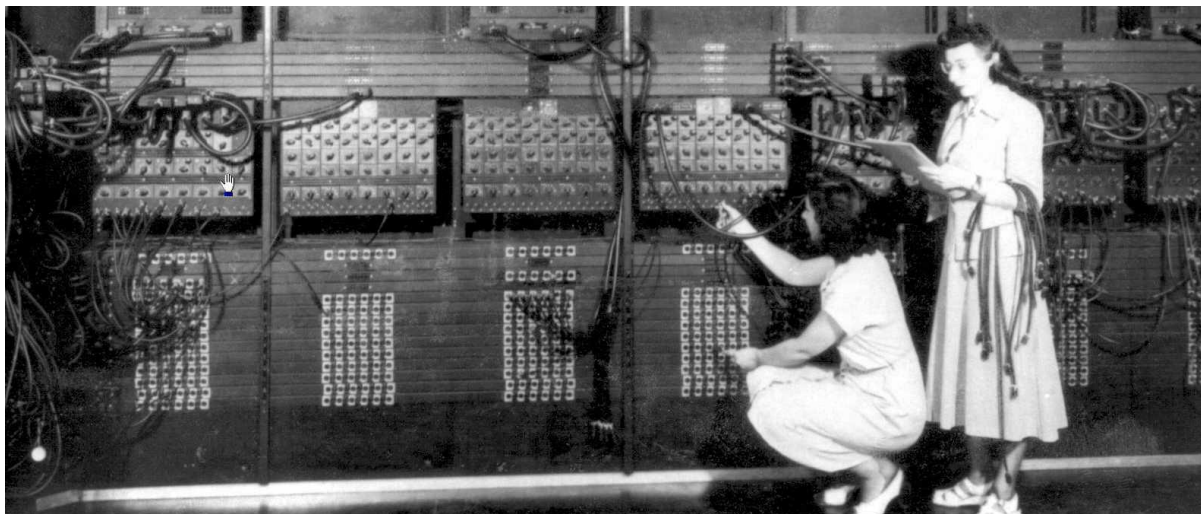
在计算机诞生之前

- Alan Turing's “machine” (1936)

- 并没有真正 “造出来”
- 纸带+自动机
 - 纸带 `map<int, int> mem`
 - 读写头 `pos`
 - 自动机 (程序)
 - (移动读写头) `pos++`, `pos--`
 - (写内存) `mem[pos] = 0`, `mem[pos] = 1`
 - (读内存) `if(mem[pos]) { } else { }`
 - (跳转) `goto label`
 - (终止) `halt()`



ENIAC: 人类可用的Turing Machine



- ENIAC Simulator by Brian L. Stuart

- 20 word (not bit) memory
- 自带“寄存”的状态 (寄存器)
- 支持算数运算
- 支持纸带 (串行) 输入输出



von Neumann Machine: 存储程序控制

可以把状态机的状态保存在存储器里，而不要每次重新设置。

- 计算机的设计受到数字电路实现的制约
 - 执行一个动作 (指令) 只能访问有限数量的存储器
 - 常用的临时存储 (寄存器; 包括 PC)
 - 更大的、编址的内存
 - (是不是想起了 YEMU?)

von Neumann Machine: 更多的I/O设备

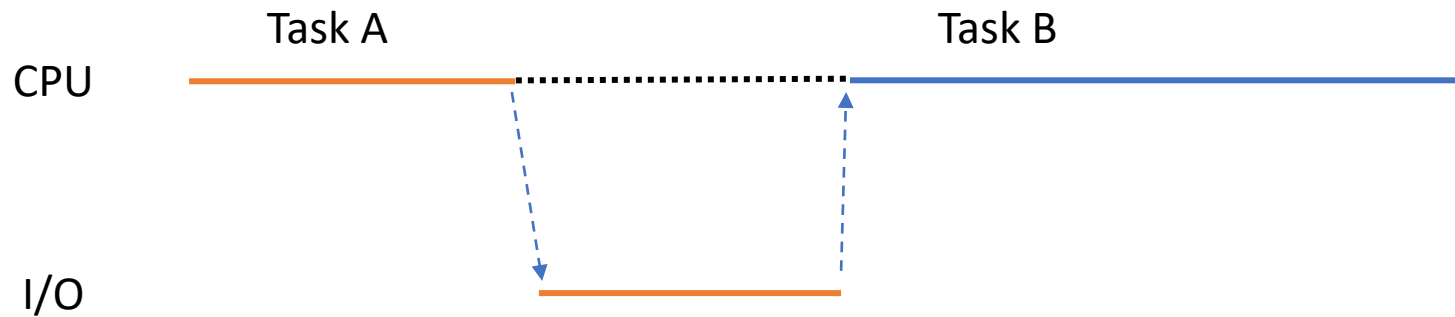
存储程序的通用性真正掀起了计算机走向全领域的革命。



- 只要增加 in 和 out 指令，就可以和物理世界建立无限的联系
 - 持久存储 (磁带、磁盘.....)
 - 读卡器、打印机.....

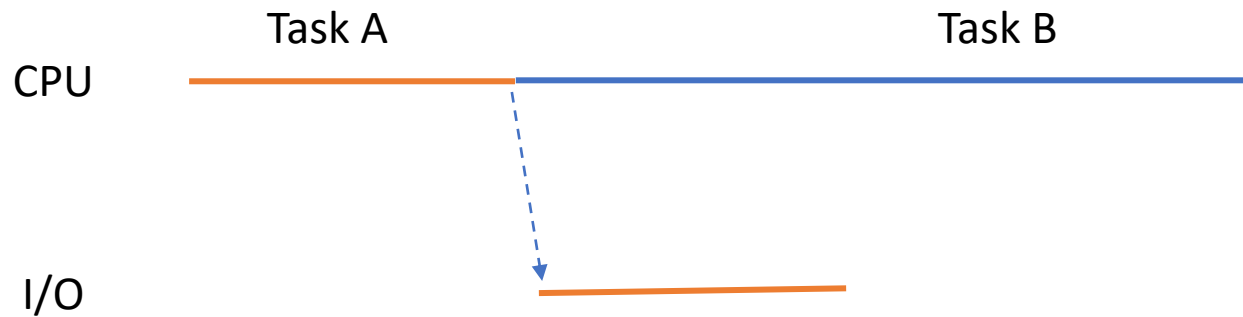
中断：弥补I/O设备的速度缺陷

- CPU cycles 实在太珍贵了
 - 不能用来浪费在等 I/O 设备完成上
 - 拥有机械部件的 I/O 设备相比于 CPU 来说实在太慢了



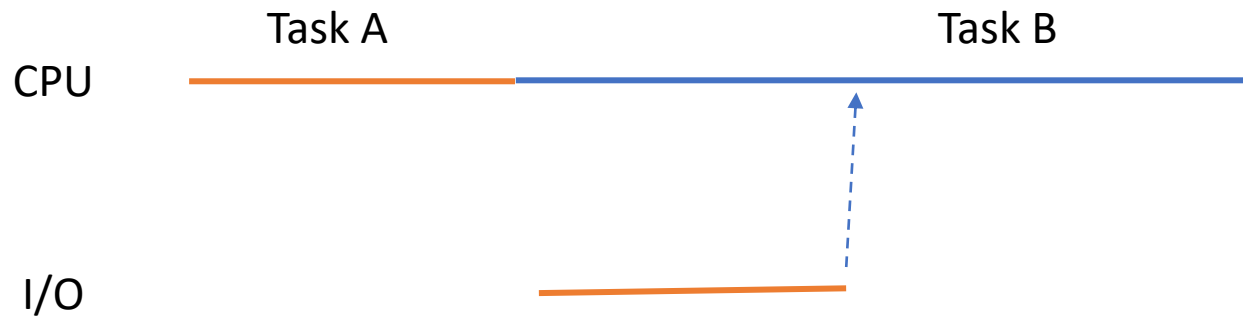
中断：弥补I/O设备的速度缺陷

- CPU cycles 实在太珍贵了
 - 不能用来浪费在等 I/O 设备完成上
 - 拥有机械部件的 I/O 设备相比于 CPU 来说实在太慢了



中断：弥补I/O设备的速度缺陷

- CPU cycles 实在太珍贵了
 - 不能用来浪费在等 I/O 设备完成上
 - 拥有机械部件的 I/O 设备相比于 CPU 来说实在太慢了



中断：弥补I/O设备的速度缺陷

- CPU cycles 实在太珍贵了
 - 不能用来浪费在等 I/O 设备完成上
 - 拥有机械部件的 I/O 设备相比于 CPU 来说实在太慢了
- 中断 = 硬件驱动调用的函数调用
 - 相当于在每条语句后都插入

```
if (pending_io && int_enabled) {  
    interrupt_handler();  
    pending_io = 0;  
}
```

- (硬件上好像不太难实现)
 - 于是就有了最早的操作系统：管理I/O设备的库代码

中断 + 更大的内存 = 分时多线程

```
void foo() { while (1) printf("a"); }  
void bar() { while (1) printf("b"); }
```

- 能够让foo()和bar() “同时” 在处理器上执行?
 - 借助每条语句后被动插入的interrupt_handler()调用

```
void interrupt_handler() {  
    dump_regs(current->regs);  
    current = (current->func == foo) ? bar : foo;  
    restore_regs(current->regs);  
}
```

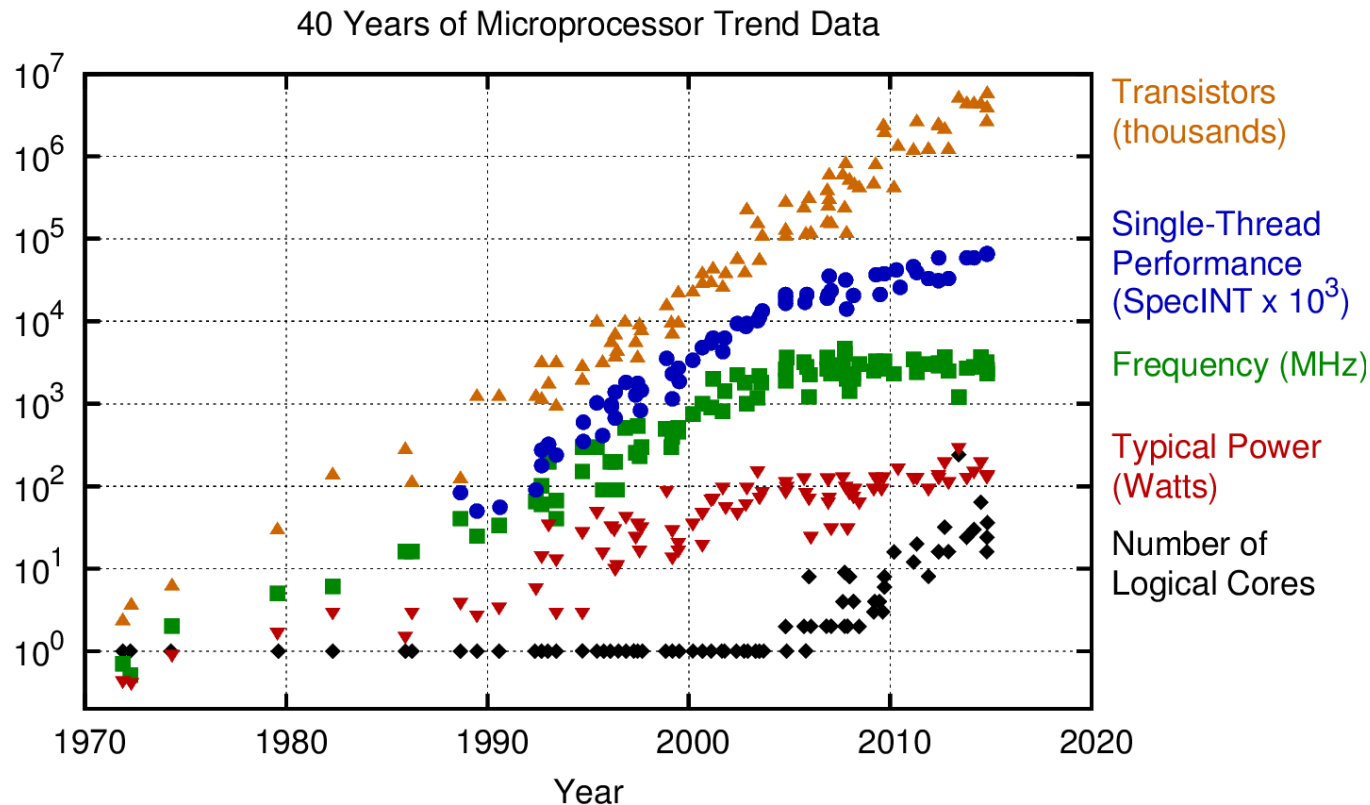
- 操作系统背负了 “调度” 的职责

分时多线程 + 虚拟存储 = 进程

- 让foo()和bar()的执行互相不受影响
 - 在计算机系统里增加映射函数 $f_{\text{foo}}, f_{\text{bar}}$
 - foo访问内存地址 m 时, 将被重定位到 $f_{\text{foo}}(m)$
 - bar访问内存地址 m 时, 将被重定位到 $f_{\text{bar}}(m)$
- foo和bar本身无权管理 f
- 操作系统需要完成进程、存储、文件的管理
- UNIX诞生 (就是我们今天的进程; Android app; ...)
 - `gcc a.c`
 - `readelf -a a.out` → 二进制文件的全部信息

故事其实没有停止.

- 面对有限的功耗、难以改进的制程、无法提升的频率、应用需求
 - 多处理器、big.LITTLE、异构处理器 (GPU、NPU、...)
 - 单指令多数据(MMX, SSE, AVX, ...), 虚拟化(VT; EL0/1/2/3), ... , 安全执行环节 (TrustZone; SGX), ...



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

End.

(今天应该解开了很多同学对计算机专业的疑惑?)

GitHub is a development platform inspired by the way you work. From open source to business, you can host and review code, manage projects, and build software alongside 50 million developers.

```
git clone -b 2021 https://github.com/NJU-ProjectN/ics-pa ics2021  
git clone https://github.com/NJU-ProjectN/ics-workbench
```