第26讲 输入输出设备

背景回顾：我们已经了解了底层存储 1-Bit 数据的原理：磁、坑、电。是时候把它们 “接到” 电脑上了。那么，输入输出设备又是如何和处理器协作完成任务的？

本讲内容：输入输出设备原理

- 计算机与外设的接口
- 总线、中断控制器和 DMA
- GPU 和加速器

1. CSAPP: recall from ICS

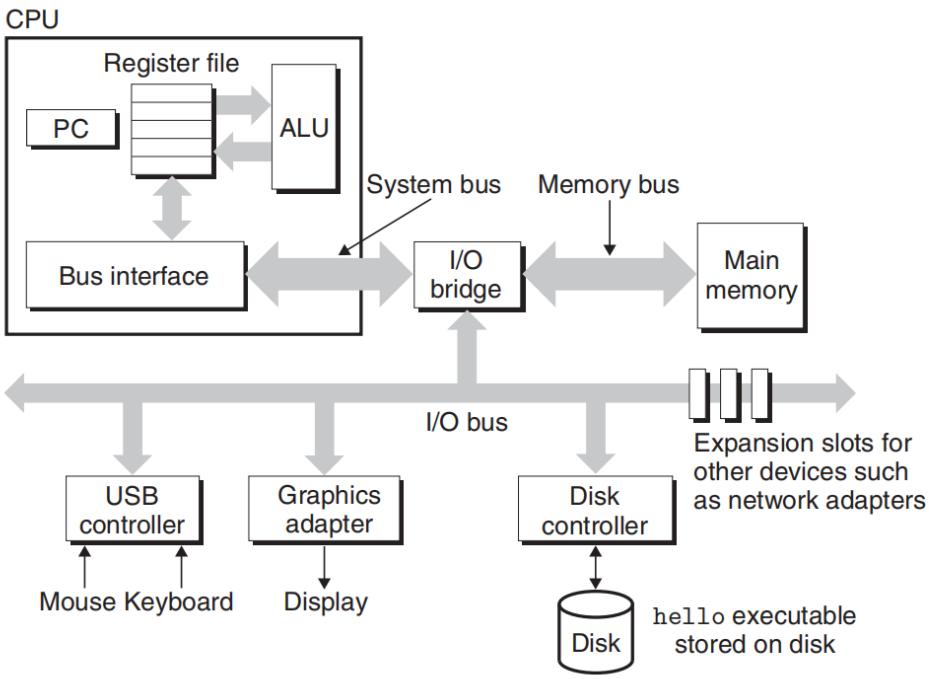


Figure 1.4
Hardware organization of a typical system. CPU: central processing unit, ALU: arithmetic/logic unit, PC: program counter, USB: Universal Serial Bus.

1.4.1 Hardware Organization of a System

To understand what happens to our `hello` program when we run it, we need to understand the hardware organization of a typical system, which is shown in Figure 1.4. This particular picture is modeled after the family of recent Intel systems, but all systems have a similar look and feel. Don't worry about the complexity of this figure just now. We will get to its various details in stages throughout the course of the book.

Buses

Running throughout the system is a collection of electrical conduits called *buses* that carry bytes of information back and forth between the components. Buses are typically designed to transfer fixed-size chunks of bytes known as *words*. The number of bytes in a word (the *word size*) is a fundamental system parameter that varies across systems. Most machines today have word sizes of either 4 bytes (32 bits) or 8 bytes (64 bits). In this book, we do not assume any fixed definition of word size. Instead, we will specify what we mean by a "word" in any context that requires this to be defined.

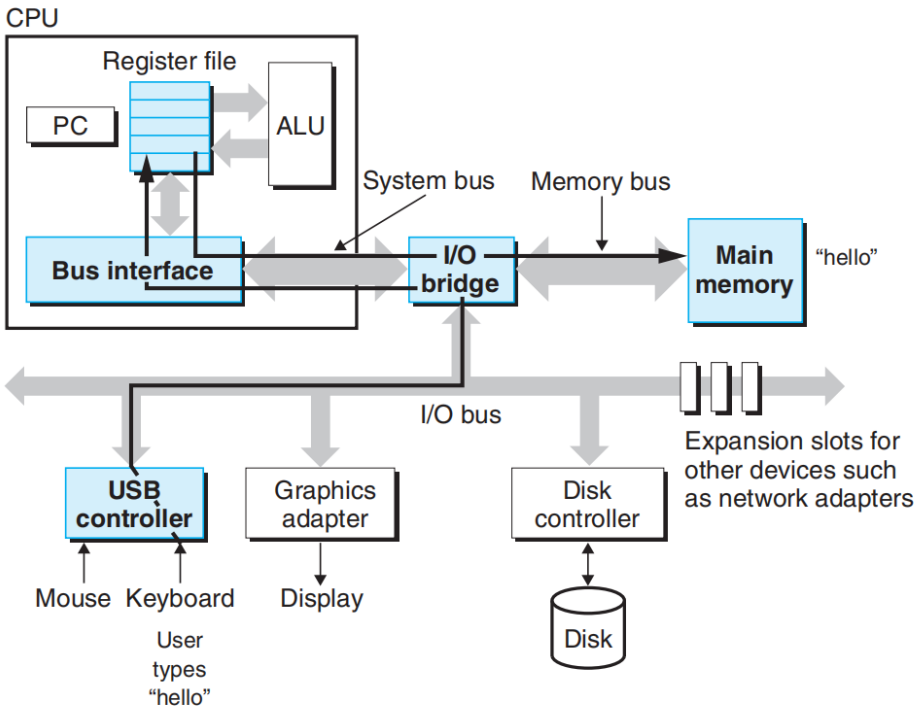
I/O Devices

Input/output (I/O) devices are the system's connection to the external world. Our example system has four I/O devices: a keyboard and mouse for user input, a display for user output, and a disk drive (or simply disk) for long-term storage of data and programs. Initially, the executable `hello` program resides on the disk.

Each I/O device is connected to the I/O bus by either a *controller* or an *adapter*. The distinction between the two is mainly one of packaging. Controllers are chip sets in the device itself or on the system's main printed circuit board (often called the *motherboard*). An adapter is a card that plugs into a slot on the motherboard. Regardless, the purpose of each is to transfer information back and forth between the I/O bus and an I/O device.

Chapter 6 has more to say about how I/O devices such as disks work. In Chapter 10, you will learn how to use the Unix I/O interface to access devices from your application programs. We focus on the especially interesting class of devices known as networks, but the techniques generalize to other kinds of devices as well.

Figure 1.5
Reading the `hello` command from the keyboard.



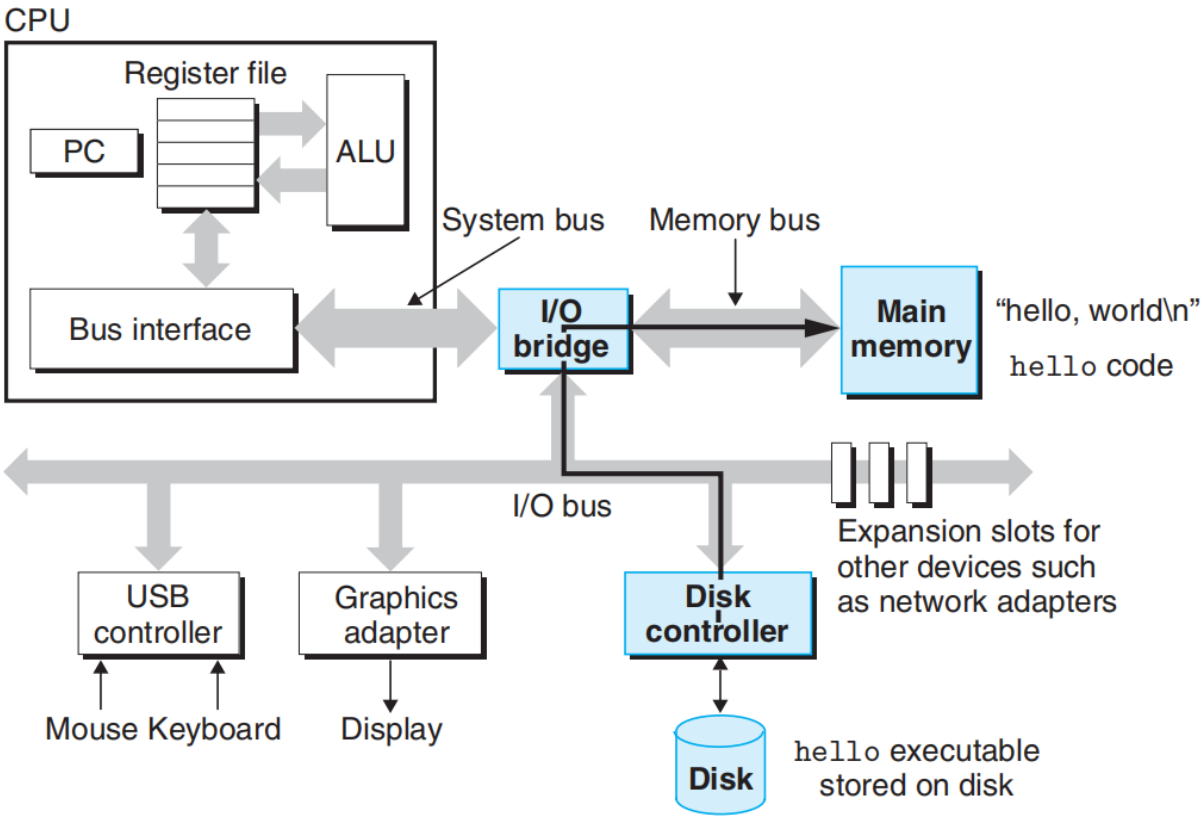


Figure 1.6 Loading the executable from disk into main memory.

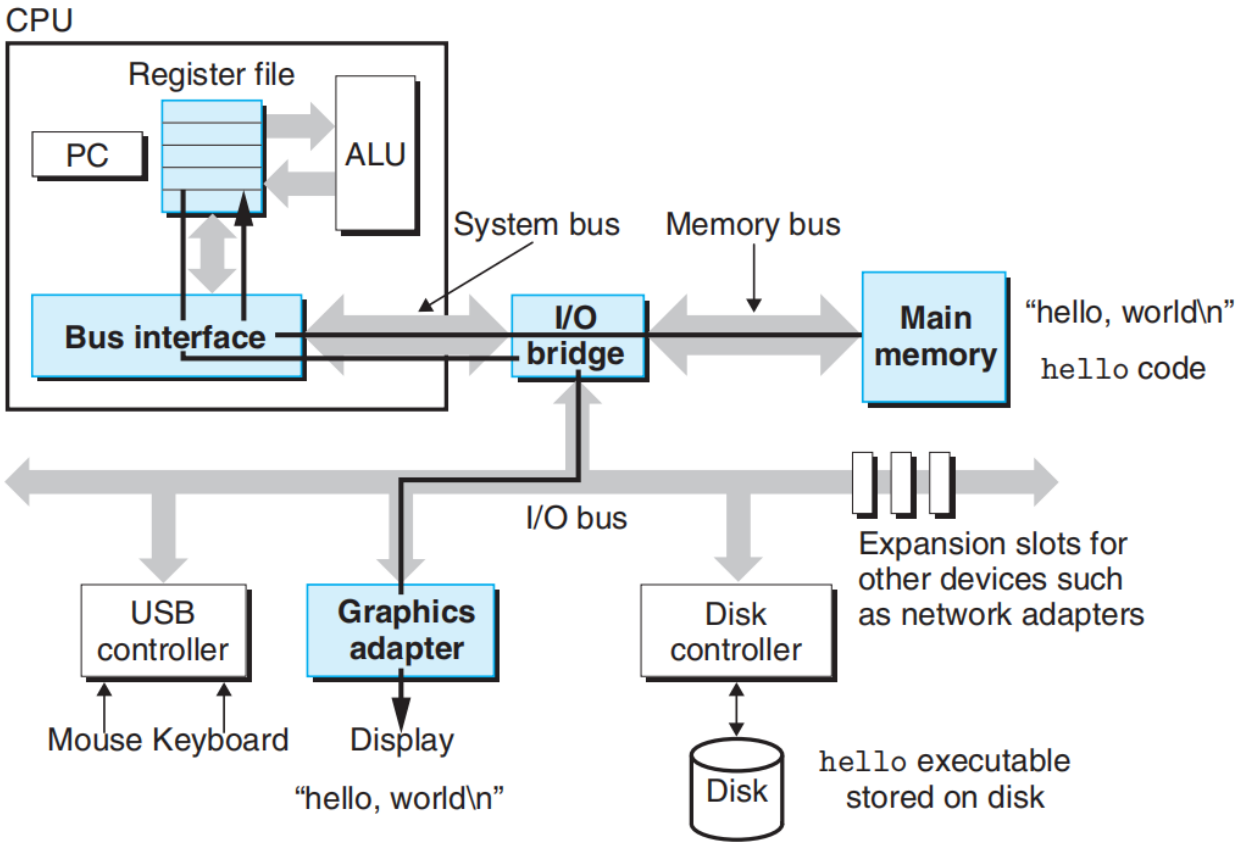


Figure 1.7 **Writing the output string from memory to the display.**

1.4.2 Running the `hello` Program

Given this simple view of a system's hardware organization and operation, we can begin to understand what happens when we run our example program. We must omit a lot of details here that will be filled in later, but for now we will be content with the big picture.

Initially, the shell program is executing its instructions, waiting for us to type a command. As we type the characters `./hello` at the keyboard, the shell program reads each one into a register and then stores it in memory, as shown in Figure 1.5.

When we hit the `enter` key on the keyboard, the shell knows that we have finished typing the command. The shell then loads the executable `hello` file by executing a sequence of instructions that copies the code and data in the `hello`

object file from disk to main memory. The data includes the string of characters `hello, world\n` that will eventually be printed out.

Using a technique known as *direct memory access* (DMA, discussed in Chapter 6), the data travel directly from disk to main memory, without passing through the processor. This step is shown in Figure 1.6.

Once the code and data in the `hello` object file are loaded into memory, the processor begins executing the machine-language instructions in the `hello` program's `main` routine. These instructions copy the bytes in the `hello, world\n` string from memory to the register file, and from there to the display device, where they are displayed on the screen. This step is shown in Figure 1.7.

2. 输入输出设备原理

孤独的CPU——无情的执行指令的机器。

- Fetch, decode, execute

如何让CPU只会外部设备？

比如——如何实现核弹发射箱？如何使得计算机感知外部状态（耳朵、眼睛），对外部实施动作（手）？

答案：一根线

核弹发射箱

- 把钥匙的状态写入 flip/flop (0/1)，并配指示灯
- 把按钮的状态写入 flip/flop (0/1)
- CPU 提供一条指令
  - reg_read(x)
  - 检测到按钮和钥匙的装填，reg_write(launch, 1)
- 检测到 launch 寄存器，即发射核弹

(这个可以做数字电路课的选做实验)

下面这张图就是IO设备的抽象：

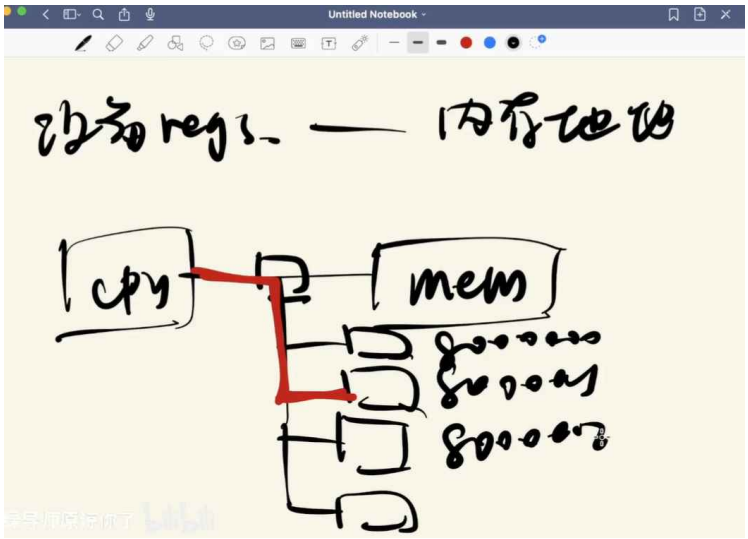
- 你可以认为它有三种类型的寄存器
  - status：这个设备的状态。这个磁盘它是不是正在写数据啊？他忙不忙？我能不能给他发指令？因为通常而言一个设备它一个时间它只能处理一个工作，比如说这个打印机正在打印的时候，可能我就不能再让他再打印第二个文件。比如我发射核弹，你就可以读出来我现在的核弹是不是处于就绪的位置？然后我的这个按键，按钮是不是属于按下的位置？钥匙是不是属于转过去的位置？

- Data：当我们谈输入输出的时候，我们通常还是会希望在 CPU 和其他的内存之间要传送数据。

对于输入输出设备而言，有一个很重要的特点，就是我们系统里面的输出设备是很多的。但是我们的 CPU 只有一个，而且你们也知道 CPU 是不可能有太多的引脚的。那么为了管理很多的输入输出设备，那每一个输出设备它都是若干个寄存器，我们就可以给每一个寄存器编一个地址。当他有地址的时候，你突然就发现，我们可以像访问内存一样访问这些寄存器了。

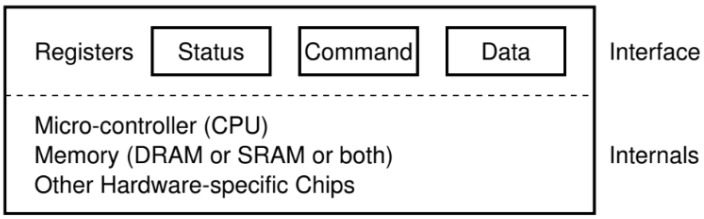
【设备寄存器到内存地址的映射】实现：

- cpu接到内存控制器上，如果你给内存相应的地址，它就可以把相应的数据写进去或者读出来。而同样的我们可以在这里加一个多路选择器——相当于这里有一个if-else，这就是数字电路的东西了。
- 我们 CPU 读写某一个地址的时候，我们就不再是直接从内存里读数据，而是从设备寄存器里面将其读出来。
- 那这样你就可以用 load 和 store 指令，而不需要为你的 CPU 单独增加额外的新的指令。



I/O 设备：“计算”和“物理世界”之间的桥梁

I/O 设备 (CPU 视角)：“一个能与 CPU 交换数据的接口/控制器”



说人话

- 就是“几组约定好功能的线” (RTFM)
  - 通过握手信号从线上读出/写入数据
- 每一组线有自己的地址
  - CPU 可以直接使用指令 (in/out/MMIO) 和设备交换数据
  - (CPU 完全不管设备具体是如何实现的)

我们的计算机系统里有什么设备？

厂商必须告诉操作系统：哪个设备在什么地址

- 0x1f2 - ATA 控制器
- 0x3f8 - COM1

Linux 有一个 Device Tree 机制

- qemu-system-aarch64 -machine virt -cpu cortex-a57 -machine dumpdtb=dtb.dtb
- 可以通过 dtc 查看厂商给出的配置

3. 总线、中断控制器、DMA

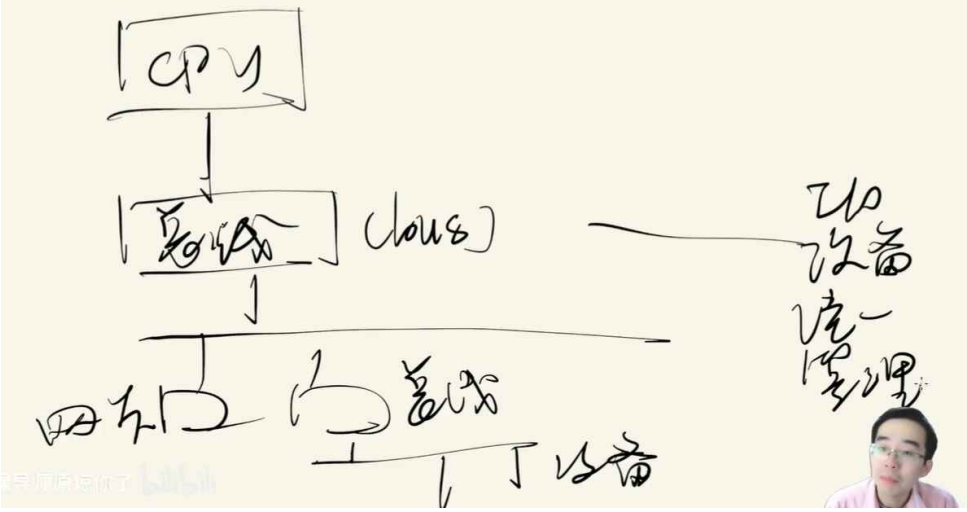
对我们自己做实验来说，或者你们自己做一个自制的CPU，那你肯定会想，OK，我就打算支持这 5 个设备，那我就把这几个设备支持了。

但是如果你希望给未来留一点空间，比如你希望不断升级你的设备，你希望计算机不断接入新的IO设备，而不仅仅是现在能够看到的IO设备，而且你甚至也不知道未来能够出现什么新的IO设备。

如果我们希望在未来能够接入更多 IO 设备、有更多的扩展的空间的时候，这个时候，我们聪明的计算机的设计者他就想到了做一个特殊的 IO 设备，然后这个 IO 设备它会把其他所有的 IO 设备都管起来——总线。

CPU从此不需要直接看到所有的设备（IBM-PC就需要规定每一个设备在一个固定的位置），现在CPU只需要看到总线，总线上面可以连接很多的设备，甚至别的总线控制器。总线就成为了把所有的IO设

备管理起来的设备了。



- 举例：
  - 有很多设备都想给CPU发中断，那我们的总线控制器也会协调这些中断，比如说哪一个中断的优先级更高，它会把这个中断送给处理器。
  - 包括我们的 CPU 也可以问总线控制器是哪一个设备发生中断。
- 所以你可以想象成总线就是一个特殊的 IO 设备，它可以提供设备的注册。其实很重要的， IBM PC 它就有一个很重要的缺点，就是它那些设备不支持热插拔。

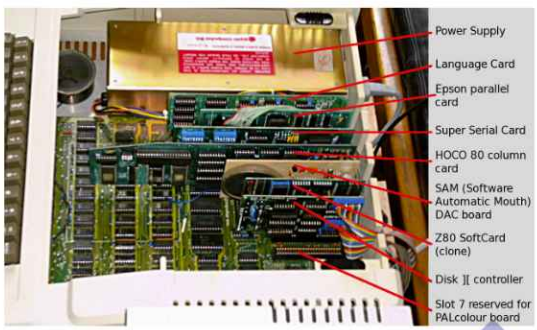
越来越多的 I/O 设备

如果你只造“一台计算机”

- 随便给每个设备定一个端口/地址，用 mux 连接到 CPU 就行
  - 你们的实验 (AbstractMachine) 和自制 CPU 就是这么做的

但如果你希望给未来留点空间？

- 想卖大价钱的“大型机”
  - IBM, DEC, ...
- 车库里造出来的“微型机”
  - 名垂青史的梦想家
- 都希望接入更多 I/O 设备
  - 甚至是未知的设备，但不希望改变 CPU？



总线：一个特殊的 I/O 设备

提供设备的注册和地址到设备的转发

- 把收到的地址 (总线地址) 和数据转发到相应的设备上
- 例子: port I/O 的端口就是总线上的地址
  - IBM PC 的 CPU 其实只看到这一个 I/O 设备

这样 CPU 只需要直连一个总线 就行了！

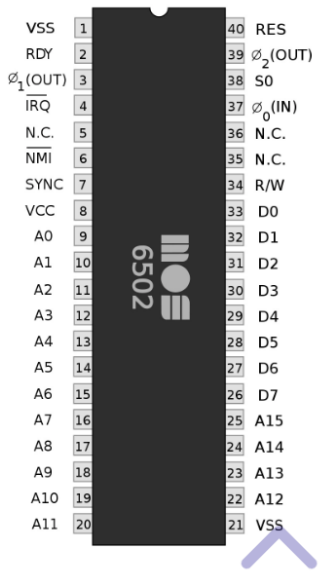
- 今天 PCI 总线肩负了这个任务
  - 总线可以桥接其他总线 (例如 PCI ➡ USB)
- `lspci -tv` 和 `lsusb -tv`: 查看系统中总线上的设备
  - 概念简单，实际非常复杂……
    - 电气特性、burst 传输、中断、Plug and Play……

- 还有一种非常特别的设备，就是中断控制器。中断是一个很有意思的机制，这个机制奠定了操作系统的霸主地位。为什么死循环不会使计算机失去响应？因为我们除了执行指令之外，我们还有一种可能性，就是发生中断。，而我们的应用程序是没有办法控制这个中断的。而到底什么是中断？中断就是我们的 CPU 上的一根线。

中断控制器

CPU 有一个中断引脚

- 收到某个特定的电信号会触发中断
  - 保存 5 个寄存器 (cs, rip, rflags, ss, rsp)
  - 跳转到中断向量表对应项执行



系统中的其他设备可以向中断控制器连线

- Intel 8259 PIC
  - programmable interrupt controller
  - 可以设置中断屏蔽、中断触发等……
- APIC (Advanced PIC)
  - local APIC: 中断向量表, IPI, 时钟, ……
  - I/O APIC: 其他 I/O 设备

写数据，那这个其实也是一个非常耗时的操作。如果有一个 CPU 或者有一个线程要做这件事情的话，那这个 CPU 基本上就赔在里面了。既然这个操作很特殊，就是把我们内存里面的某一块搬到某一个 IO 设备里，那么我们有没有可能把 CPU 从这个执行代码的循环里面给解放出来？

- 解决方案：专门有一个小的CPU，然后这个 CPU 它就只能执行一个功能，就是 mem-copy。它不需要像一个我们通用的 CPU 一样支持很多很多的指令。它甚至可以直接把 memory copy 的逻辑硬编码到一个电路里。
- 所以你就想象成是如果你在系统里面征用一个小的CPU，专门用来执行 memory copy。
- 我们就可以把在设备和内存之间，或者内存和内存之间做这个很大的数据传输，这样的事情从我们的 CPU 里面解脱出来，我们就不要用一个全功能的 CPU 去执行这样的命令——太浪费。
- 这就是传说中的DMA（direct memory access）。虽然我们叫它一个执行memcpy的cpu，这玩意其实还是可以看成是一个IO设备。
- 它既可以支持 memory 到memory，也可以支持 memory 到device（当然反过来也行）。

中断没能解的问题

假设程序希望写入 1 GB 的数据到磁盘

- 即便磁盘已经准备好，依然需要非常浪费缓慢的循环
- out 指令写入的是设备缓冲区，需要去总线上绕一圈
  - cache disable; store 其实很慢的

```
for (int i = 0; i < 1 GB / 4; i++) {
  outl(PORT, ((u32 *)buf)[i]);
}
```

能否把 CPU 从执行循环中解放出来？

- 比如，在系统里征用一个小 CPU，专门复制数据?
- 好像 `memcpy_to_port(ATA0, buf, length);`

Direct Memory Access (DMA)

DMA: 一个专门执行“memcpy”程序的 CPU

- 加一个通用处理器太浪费，不如加一个简单的

支持的几种 memcpy

- memory ➞ memory
- memory ➞ device (register)
- device (register) ➞ memory
  - 实际实现：直接把 DMA 控制器连接在总线和内存上
  - Intel 8237A

PCI 总线支持 DMA

- sudo cat /proc/iomem

所以我们刚才讲了三种很特别的 IO 设备，看到为了支撑我们现代的高性能的计算机系统，我们的 IO 设备会比一个最简化的模型——每一个设备就是几个寄存器，要做的更聪明一些，也解决的更漂亮一些。

4. GPU和加速器

刚才我们说到，IO 设备和计算机之间的边界其实是很模糊的。比如说我们的DMA，是一种IO设备，某种程度上也可以解释为一个专门执行memcpy的cpu。

所以按照相同的思路——我们的 X86 的CPU，它是一个集成度很高、成本很高的CPU，那么它他拿来做我们通用的计算当然是很好的，但是他用来做 DMA 就有点浪费了。同样，渲染图形也是如此。如果你有一件事情很重要的事情要做，我们 CPU 如果想做起来有点捉襟见肘，那我们自然就可以在系统里面添一个专用的 CPU 来做这件事，那就自然就有了显卡。

I/O 设备和计算机之间的边界逐渐模糊

DMA 不就是一个 “做一件特别事情” 的 CPU 吗

- 那么我们还可以有做各种事情的 “CPU” 啊

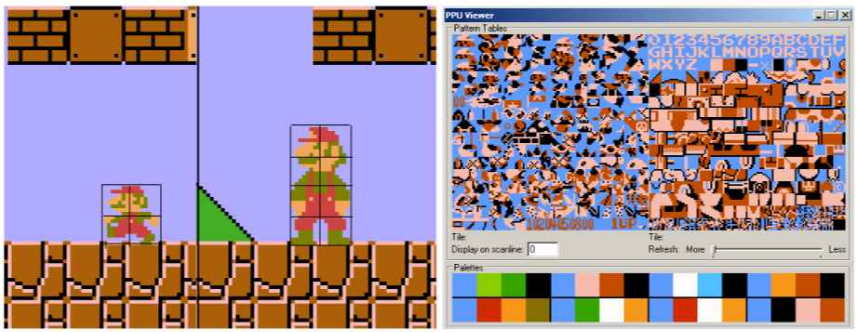
例如，显示图形

```
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 x 240 = 61K 像素 (256 色)
- 60FPS ➜ 每一帧必须在 ~10K 条指令内完成
  - 如何在有限的 CPU 运算力下实现 60Hz?

NES Picture Processing Unit (PPU)



CPU 只描述 8x8 “贴块” 的摆放方法

- 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
  - 一个并行度更高的 “CPU”

```
76543210
||||||||
||||||++- Palette
|||+++--- Unimplemented
||+------ Priority
|+------- Flip horizontally
+-------- Flip vertically
```

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
  - Thm. 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)

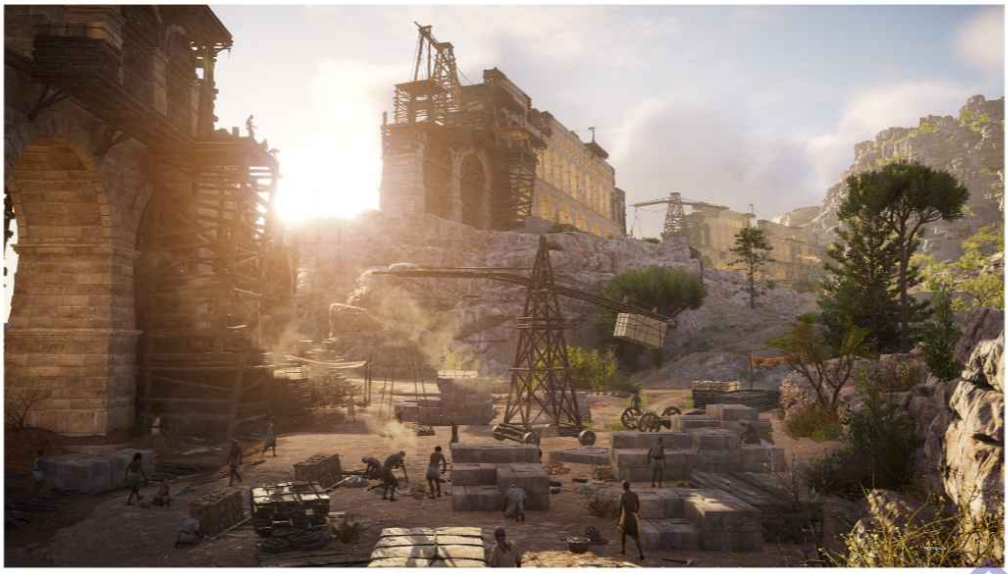
但我们还是需要真正的 3D

三维空间中的三角形需要正确渲染

- 这时候建模的东西就多了
  - 几何、材质、贴图、光源、……
  - Rendering pipeline 里大部分操作都是 massive parallel 的

“Perspective correct” texture mapping (Wikipedia)

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……

现代 GPU: 一个通用计算设备

一个完整的众核多处理器系统

- 注重大量并行相似的任务
  - 程序使用例如 OpenGL, CUDA, OpenCL, ... 书写
- 程序保存在内存 (显存) 中
  - nvcc (LLVM) 分两个部分
    - main 编译/链接成本地可执行的 ELF
    - kernel 编译成 GPU 指令 (PTX; 送给驱动)
    - 支持矩阵运算 (Tensor Cores)
- 数据也保存在内存 (显存) 中
  - 可以输出到视频接口 (DP, HDMI, ...)
  - 也可以通过 DMA 传回系统内存

5. Take away messages

输入/输出设备是 “与处理器交换数据” 接口——因此，我们的设备可以实现得任意复杂，甚至是一个完整的计算机系统。从我们今天的打印机、SSD、GPU，都遵循了这个模式，在 CPU 的统一管理和调度下各自完成各自的功能。