# Lecture 16: Input/Output and Storage Devices

(Interface, Bus, Interrupt, DMA, Heterogeneous Computing)

Xin Liu

Florida State University

xl24j@fsu.edu

COP 4610 Operating Systems

https://xinliulab.github.io/cop4610.html

October 30, 2024

# Outline

Today's Key Question:

**What exactly is a "Device" to an OS?**

**Main Topics for Today:**

- **Interface Between Computer and Peripheral Devices**
  - **Character Devices**
  - **Block Device: Storage Drives**
- **Bus, Interrupt Controller, and DMA (Direct Memory Access)**
- **Heterogeneous Computing and GPU**

# Interface Between Computers and the World

# The Lonely CPU

**CPU: Just an "Instruction-Executing Machine"**

- The CPU operates without emotion, executing instructions: **Fetch, Decode, Execute**
- Not User-Friendly


Altair-8800 (1975), featuring the Intel 8080A CPU with 256B RAM
(Manual input of the execution start address was required on the front panel switches.)

# From a Need to an Implementation

How can we use a computer to launch a nuclear missile?

- **Key Question**: How can a computer sense external states and perform actions in the real world?

The nuclear football

- What makes computers interesting are the I/O devices!

# Device Management

- The OS component that manages hardware devices.
- Provides a uniform interface to access devices with different physical characteristics.
- Optimizes the performance of individual devices.

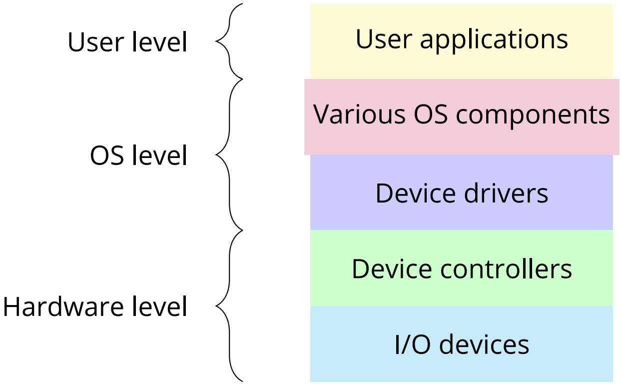
# Device Controller

- Converts between serial bit stream and a block of bytes
- Performs error correction if necessary
- Components
  - Device registers to communicate with the CPU
  - Data buffer that an OS can read or write

# Device Driver

- An OS component that is responsible for hiding the complexity of an I/O device
- So that the OS can access various devices in a uniform manner

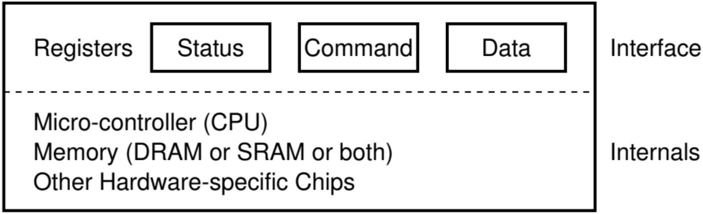
# Device Driver Illustrated



# Difference between Driver and Controller

- **Device Driver:**
  - A software component in the OS.
  - Interfaces between the OS and the device, providing a standardized way for applications to use hardware without needing to know its specifics.
  - Translates high-level OS requests (e.g., read, write) into specific commands understood by the device controller.
- **Device Controller:**
  - A hardware component, usually located on or within the device itself.
  - Manages the data transfer between the device and the computer system.
  - Handles protocol details, signal conversion, and low-level operations, acting as an intermediary between the device driver and the actual hardware.

**From CPU's perspective:** "An interface/controller that exchanges data with the CPU"



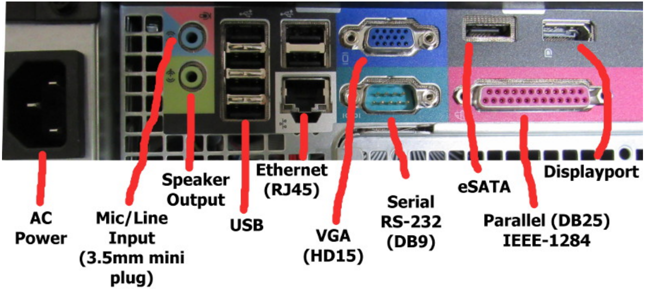

In simple terms:

- Just "a set of wires with agreed-upon functions" (RTFM)
  - Data is read/written from the wires via handshake signals
- Each set of wires has its own address
  - The CPU can directly use instructions (in/out/Memory-mapped I/O) to exchange data with the device
- The CPU doesn't care how the device is implemented
  - Using Ctrl (Valid/Ready), Rd, Wr, Addr, and Dat to model all devices!

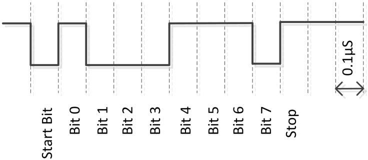
# More on I/O Devices

- **Main categories**
  - **A character device** delivers or accepts a stream of characters, and individual characters are not addressable
    - e.g., serial port, keyboards, printers
  - **A block device** stores information in fixed-size blocks, each one with its own address
    - e.g., disks
  - **A network device** transmits data packets

# Example (1): Serial Port (UART)




0x71, 8N1 ( 8 Data bits, No Parity, 1 Stop)



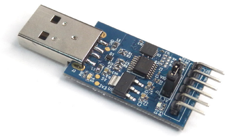

A UART Waveform

## "COM1"; Implementation of `putch()`

```
// Define the base address of COM1 (standard for serial port communication)
#define COM1 0x3f8

// Initialize the UART serial port at COM1
static int uart_init() {
  outb(COM1 + 2, 0);
  outb(COM1 + 3, 0x80);
  outb(COM1 + 0, 115200 / 9600);
  ...
}

// Transmit function for UART - send data through COM1
static void uart_tx(AM_UART_TX_T *send) {
  outb(COM1, send->data);
}

// Receive function for UART - read data from COM1
static void uart_rx(AM_UART_RX_T *recv) {
   recv->data = (inb(COM1 + 5) & 0x1) ? inb(COM1) : -1;
}
```

# The Role and Evolution of UART in Modern Systems

- The OS doesn't really care about the protocol details.
  - It only configures the UART's mode and the data to send.
  - A UART chip helps the OS abstract the external UART device.
    - It handles the protocol, translating CPU instructions into electrical signals that comply with the UART standard.
- Nowadays, even the chip has been removed:
  - Laptops no longer have a built-in UART port, and even most desktop PCs don't include it anymore.
- However, the UART is still widely used.
  - Engineers use a device with a chip that converts USB signals into UART signals to connect the computer to other UART devices.



USB to UART

# Example (2): Keyboard Controller

**IBM PC/AT 8042 PS/2 (Keyboard) Controller**

- ”Hardcoded” to two I/O ports: 0x60 (data), 0x64 (status/command)

| Command Byte | Use | Description |
|---|---|---|
| 0xED | LED Control | ScrollLock, NumLock, CapsLock LEDs |
| 0xF3 | Set Repeat Rate | 30Hz - 2Hz; Delay: 250ms - 1000ms |
| 0xF4 / 0xF5 | Enable / Disable | N/A |
| 0xFE | Resend | N/A |
| 0xFF | RESET | N/A |

# Example (3): Disk Controller

**ATA (Advanced Technology Attachment)**

- IDE (Integrated Drive Electronics) interface disks
- Primary port range: 0x1f0 - 0x1f7; Secondary port range: 0x170 - 0x177

```
// Function to read a sector from the disk
void readsect(void *dst, int sect) {
  waitdisk();
  out_byte(0x1f2, 1); // sector count (1)
  out_byte(0x1f3, sect); // sector number (low byte)
  out_byte(0x1f4, sect >> 8); // cylinder number (low byte)
  out_byte(0x1f5, sect >> 16); // cylinder number (high byte)
  out_byte(0x1f6, (sect >> 24) | 0xe0); // drive selection
  out_byte(0x1f7, 0x20); // command to read sector
  waitdisk();
  for (int i = 0; i < SECTSIZE / 4; i ++)
    ((uint32_t *)dst)[i] = in_long(0x1f0); // read data from data register
}
```

# Example (4): Printer

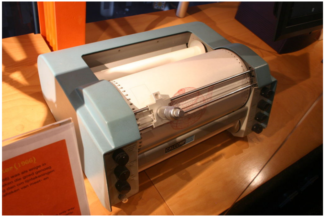
Translates a stream of bytes into printed text or graphics on paper.

- **Simple Use:** Basic text output (like a typewriter).
- **Complex Use:** Graphics described by programming languages.
- **High-Resolution Images:** Transmitting full-page, high-resolution images poses a significant challenge.

**Example: PostScript (1984)**

- A domain-specific language (DSL) for describing page layouts.
  - Similar to assembly language.
  - Can create high-quality documents using a “compiler” (e.g., LaTeX).
    - The slides you’re viewing now were generated with LaTeX.

- PDF is a superset of PostScript (e.g., page.ps).
  - Printers are devices equipped with CPUs, functioning as standalone processors.



Calcomp 565 drum plotter

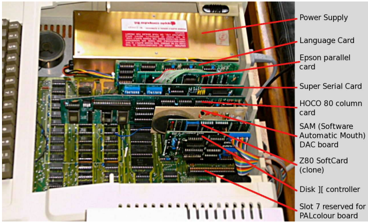
# Bus, Interrupt Controller, and DMA

**If you're only building "a computer"**

- Simply assign a port/address to each device and connect them to the CPU using a multiplexer

**But what if you want room for expansion?**

- Consider the "mainframes" sold at a high price
  - IBM, DEC, ...
- Or the "microcomputers" built in a garage
  - Visionaries with ambitious dreams
- All want to connect more I/O devices
  - Even unknown devices in the future, but they don't want to change the CPU




Apple II Inside

# Device Addressing

- **Two approaches**
  - **Dedicated range of device addresses in the physical memory**
    - Requires special hardware instructions associated with individual devices
  - **Memory-mapped I/O:** makes no distinction between device addresses and memory addresses
    - Devices can be accessed the same way as normal memory, with the same set of hardware instructions

# Bus: A Special I/O Device

Provides registration and address-based forwarding for devices.

- Forward addresses (bus addresses) and data to the corresponding device.
- Example: I/O ports are addresses on the bus.
  - The CPU of an IBM PC only sees this one I/O device on the bus.

Thus, the CPU only needs to connect to one bus!

- Today, the PCI bus handles this role.
- The bus can bridge to other buses (e.g., PCI to USB).
- Commands like `lspci -tv` and `lsusb -tv` allow you to see devices on the bus.
- Conceptually simple, but actually very complex...
  - Electrical characteristics, burst transfers, interrupts, and Plug and Play.

# Example: PCI Device Probe

- QEMU (Virtual Machine Emulator, supports x86-64/i386)
- Try adding the option `-soundhw ac97` to test.

```
// Scan all buses and slots for PCI devices
for (int bus = 0; bus < 256; bus++)
 for (int slot = 0; slot < 32; slot++) {
  uint32_t info = pciconf_read(bus, slot, 0, 0);
  uint16_t id = info >> 16, vendor = info & 0xffff;
  if (vendor != 0xffff) {
   printf("%02d:%02d␣device␣%x␣by␣vendor␣%x", bus, slot, id, vendor);
   if (id == 0x100e && vendor == 0x8086) {
    printf("␣<--␣This␣is␣an␣Intel␣e1000␣NIC␣card!");
   }
   printf("\n");
  }
 }
```

# Ways to Access a Device (1)

- **Polling:** a CPU repeatedly checks the status of a device for exchanging data
  - **+** Simple
  - **-** Busy-waiting

# Ways to Access a Device (2)

- **Interrupt-driven I/Os:** A device controller notifies the corresponding device driver when the device is available
  - **+** More efficient use of CPU cycles
  - **-** Data copying and movements
  - **-** Slow for character devices (i.e., one interrupt per keyboard input)

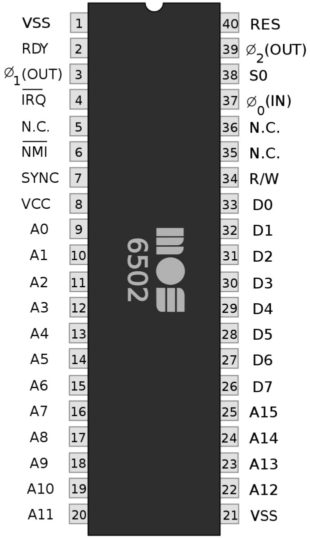
# Interrupt Controller

The CPU has an interrupt pin.

- Receiving a specific electrical signal triggers an interrupt.
  - Saves 5 registers (cs, rip, rflags, ss, rsp)
  - Jumps to the corresponding entry in the interrupt vector table

Other devices in the system can connect to the interrupt controller.

- **Intel 8259 PIC**
  - Programmable Interrupt Controller
  - Can configure interrupt masking, interrupt triggering, etc.
- **APIC (Advanced PIC)**
  - **Local APIC**: Interrupt vector table, IPI, timer, etc.
  - **I/O APIC**: Other I/O devices




MOS Technology 6502 Pinout

# The Problem That Interrupts Cannot Solve

Suppose a program wants to write 1 GB of data to the disk.

- Even if the disk is ready, the loop is still very slow and wastes CPU cycles.
- The `out` command writes to the device buffer, but data needs to go through the bus.
  - Cache is disabled; stores are actually very slow.

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

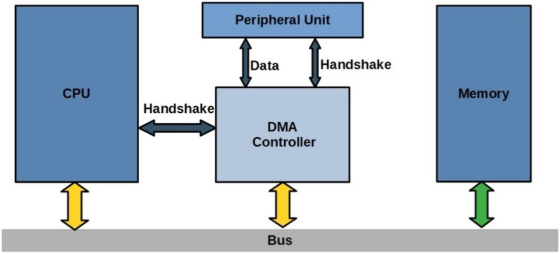
Can we free the CPU from executing the loop?

- For example, by using a small CPU in the system dedicated to copying data?
- Something like `memcpy_to_port(ATA0, buf, length);`

# Ways to Access a Device (3)

**DMA (Direct Memory Access):** A dedicated CPU for executing "memcpy" operations

- Adding a general-purpose processor is too costly
- A simple controller is a better solution
- Supported types of memcpy:
  - memory $\rightarrow$ memory
  - memory $\rightarrow$ device (register)
  - device (register) $\rightarrow$ memory
    - Practical implementation: Directly connect the DMA controller to the bus and memory
    - Intel 8237A



# More on DMA

- CPU is not involved in copying data
- A process cannot access in-transit data
- PCI bus supports DMA
  - Handles a large number of complex tasks

# Ways to Access a Device (4)

- **Double buffering:** uses two buffers
  - While one is being used, the other is being filled
  - Analogy: pipelining
  - Extensively used for graphics and animation
    - So a viewer does not see the line-by-line scanning

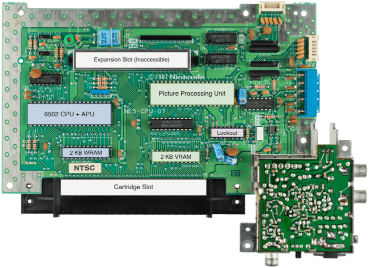
# Heterogeneous Computing and GPU

**DMA is essentially a CPU for "special tasks"**

- Then, could we have CPUs for various tasks?

**Example: Displaying Patterns**

```
#include <stdio.h>

int main() {
   int H = 10;
   int W = 10;

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

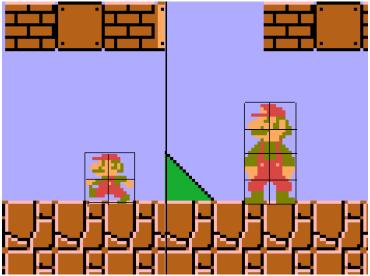


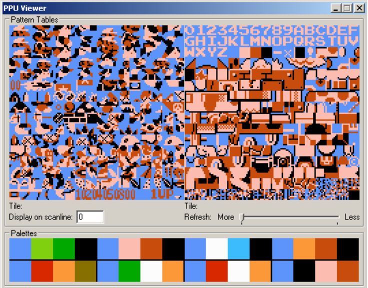

Nintendo Entertainment System (NES) Motherboard

**The Challenge of Performance:** NES: 6502 @ 1.79MHz; IPC = 0.43

- Screen resolution: 256 x 240 = 61K pixels (256 colors)
- 60FPS ⇒ Each frame must complete within  10K instructions
  - How to achieve 60Hz with limited CPU computing power?

# NES Picture Processing Unit (PPU)






The **CPU** only describes the arrangement of 8x8 tiles

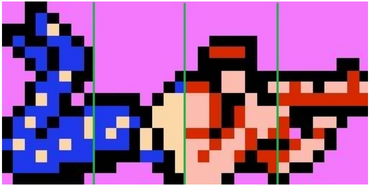
- The background is part of a larger image
  - No more than 8 foreground tiles per line
- The PPU completes the rendering
  - A simpler type of ”CPU”
- Enjoy!

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|
| \| | \| | \| | \| | \| | \| | \| | \| | |
| \| | \| | \| | \| | \| | \| | + | + | Palette |
| \| | \| | \| | \| | + | + | - | - | Unimplemented |
| \| | \| | + | - | - | - | - | - | Priority |
| \| | + | - | - | - | - | - | - | Flip horizontally |
| + | - | - | - | - | - | - | - | Flip vertically |

# Providing Rich Graphics with Limited Capability

Why do the characters in KONAMI's Contra adopt a prone position with their legs raised?

- Video



# Better 2D Game Engine

**What if we have more powerful processors?**

- The NES PPU is essentially a ”tile-based” system aligned with the coordinate axes.
  - It only requires addition and bitwise operations to work.
- Greater computational power = More complex graphics rendering.

**2D Graphics Accelerator: Image ”Clipping” + ”Pasting”**

- Supports rotation, material mapping (scaling), post-processing, etc.

**Achieving 3D**

- Polygons in 3D space are also polygons in the visual plane.
  - Thm. Any polygon with $n$ sides can be divided into $n - 2$ triangles.

# Simulated 3D with Clipping and Pasting

**GameBoy Advance**

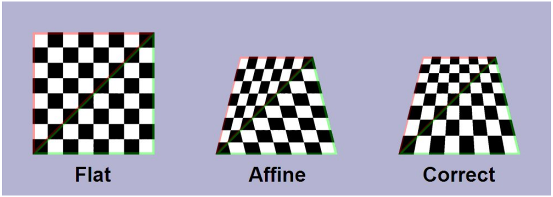
- 4 background layers; 128 clipping objects; 32 affine objects
    - CPU provides the description; GPU performs the rendering (acting as a ”program-executing” CPU)


V-Rally; Game Boy Advance, 2002

# But We Still Need True 3D

**Triangles in 3D space require correct rendering**

- Modeling at this stage includes:
  - Geometry, materials, textures, lighting, etc.
- Most operations in the rendering pipeline are massively parallel




*"Perspective correct" texture mapping (Wikipedia)*

# Solution: Full PS (Post-Processing)

**Example: GLSL (Shading Language)**

- Enables ”shader programs” to execute on the GPU
  - Can be applied at various rendering stages: vertex, fragment, pixel shaders
  - Functions as a ”PS” program to calculate lighting changes for each part
    - Global illumination, reflections, shadows, ambient occlusion, etc.

# Modern GPU: A General-Purpose Computing Device

A complete multi-core processing system

- Focuses on massively parallel similar tasks
  - Programs are written in languages like OpenGL, CUDA, OpenCL, etc.
- Programs are stored in memory (video memory)
  - `nvcc` (LLVM) compiles in two parts
    - Main: Compiles/links to a locally executable ELF
    - Kernel: Compiles to GPU instructions (sent to drivers)
- Data is also stored in memory (video memory)
  - Can output to video interfaces (DP, HDMI, ...)
  - Can also use DMA to transfer to system memory

# Example: PyTorch and Deep Learning

What is a ”Deep Neural Network”?
How do we ”train”?

- Requires computationally intensive tasks

```
class NeuralNetwork(nn.Module):
   def __init__(self):
      super(NeuralNetwork, self).__init__()
      self.flatten = nn.Flatten()
      self.linear_relu_stack = nn.Sequential(
         nn.Linear(28*28, 512), nn.ReLU(),
         nn.Linear(512, 512), nn.ReLU(),
         nn.Linear(512, 10), nn.ReLU(),
      )
...
model = NeuralNetwork().to('cuda')
```

Many components can perform the ”same task”

- The key is to choose the component with the most suitable power/performance/time trade-off!

**Examples of Components:**

- CPU, GPU, NPU, DSP, DSM/RDMA

# Takeaways

- What exactly are input/output devices?
- I/O Devices (Controllers): A set of interfaces and protocols for data exchange