CS201 W'00 Lecture Notes
Lecture 4

I/O and Related Topics

There is a huge variety of **I/O devices** available.

Most important way to characterize is by **data rate**:

Device	**Type**	**Partner**	**Data Rate (KBytes/sec)**
Keyboard	Input	Human	0.01
Mouse	Input	Human	0.02
Scanner	Input	Human	400.00
Line printer	Output	Human	1.00
Laser printer	Output	Human	200.00
Graphics display	Output	Human	60,000.00
Modem	I/O	Machine	2.00 - 8.00
Network	I/O	Machine	1000.00 - 10,000.00
Floppy disk	Storage	Machine	100.00
CD/ROM	Input	Machine	5000.00
Magnetic Disk	Storage	Machine	2000.00 - 10,000.00

Compare: on-chip data rate of a 100MHz 32-bit processor is 400,000 KBytes/sec.

Memory access rate is typically 1/10 of CPU rate.

So CPU's and memory are usually **much** faster than I/O devices! (Graphics are an important exception. Disks, networks also relatively fast.)

Example Device: Mouse

Invented by Doug Engelbart (ca. 1965)

Uses wheels or optical pulses to notice **relative** motion of mouse body.

Typical setup: mouse controller sends a short sequence of bytes to CPU each time:

– mouse moves more than a small minimum distance, or

– button is pressed or released

Transmitted info tells how far mouse has moved in X and in Y and gives current state of buttons.

CPU is completely responsible for interpreting this info to update screen, etc. Can easily keep up!

Example Device: Ethernet LAN

(Often used as synonym for LAN.)

Original Ethernet was 1.25 MByte/sec wire. Messages are sent in **packets** of size 64 to 1518 bytes. Packet **header** indicate intended receiver; all nodes listen to all headers, waiting for theirs.

Any node can start sending at any time, but continues to listen. If it detects that another sender has started before it, it **backs off** and tries again later after a (pseudo-random) period of time.

How is device interfaced? Has own memory buffer to accumulate packet; uses DMA (more later) to transfer entire packet to/from memory.

Example Device: Graphics display

Modern graphic displays are treated as arrays of **pixels** (picture elements), e.g., 640x480 or 1280x960 per screen.

Software can control contents of each pixel independently, allowing arbitrary pictures to be drawn.

Pixel values are stored in high-speed **video RAM**, usually accessible to CPU as special range of memory addresses.

The values in the RAM are accessed by the video controller to control the physical screen appearance; details depend on the display mechanism (e.g., CRT or active matrix LCD).

Need refresh rate of 30 frames/sec for full-motion video.

For reasonable color, 24 bits (8 for each primary color) must be specified for each pixel. Many machines use a restricted subset of colors at any given time, allowing fewer bits per pixel.

Overall data rate required is large! This is why video game applications have often taxed home computers!

Often we're just displaying **text**, which can be encoded **much** more compactly than arbitrary graphics using **characters**, each having a fixed pixel pattern.

Most display controllers have mode that interprets character codes and displays corresponding characters from a pre-defined **font**. (Older output devices all worked (only) like this.)

Buses

I/O devices are connected to computer system via a **bus**, which is just a collection of parallel wires connecting components.

Simplifies interconnection: each component has single bus connection instead of pairwise connections with other components.

Typically carry separate address/data and control signals.

Bus traffic occurs in discrete units called **transfers**.

These are set up by **controllers** at sending and receiving end. (CPU is not necessarily involved.)

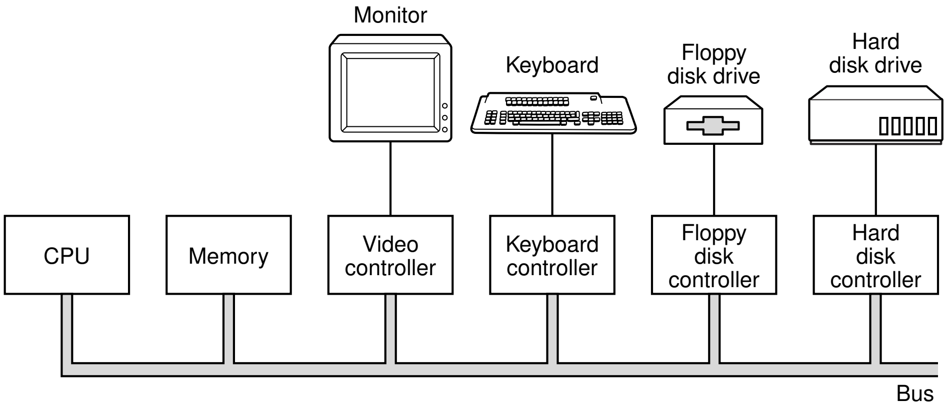

Figure 2-29. Logical structure of a simple personal computer.

Arbitration hardware is used to sequence transfers that want to overlap.

Multiple buses with different performance characteristics are now common.

PC Buses

ISA Bus

- Standardization of IBM PC-AT bus.
- 16-bit, 8.3 MHz, but only 4 MBytes/sec.

PCI Bus

- Intel developed, but not proprietary.
- 32-bit, 33 MHz, 133 MBytes/sec maximum.

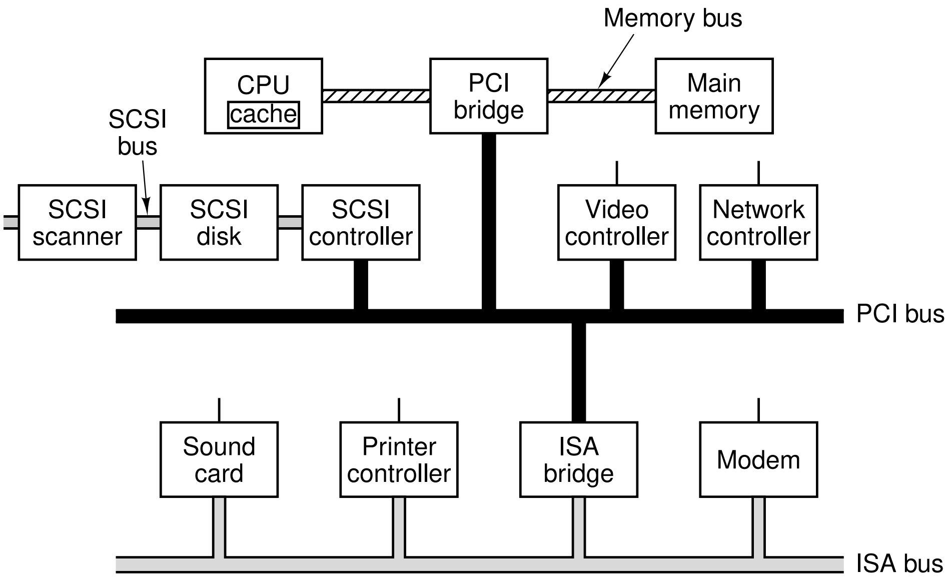

Figure 2-30. A typical modern PC with a PCI bus and an ISA bus. The modem and sound card are ISA devices; the SCSI controller is a PCI device.

I/O Control Instructions

How does CPU access I/O devices to control them or move data to/from them?

Using control registers and data registers (and/or DMA).

Typical example: UART ("universal asynchronous receiver/transmitter").

- Used for PC "serial" interface, e.g. connected to modem.

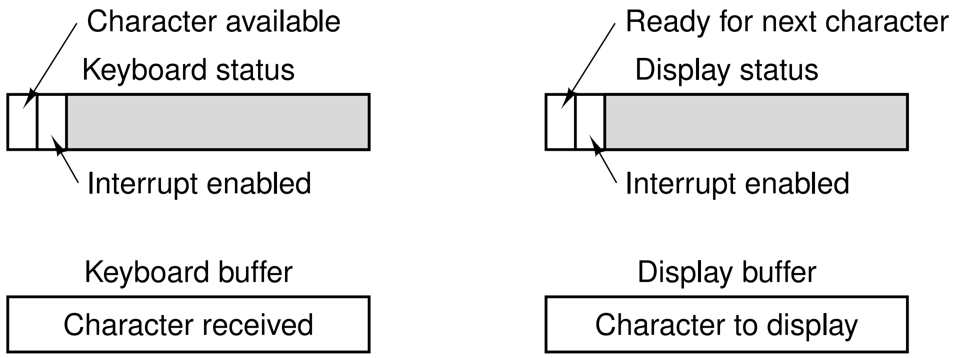

Figure 5-30. Device registers for a simple terminal.

How does program access registers? One of two methods:

Memory-mapped I/O

- I/O device registers are assigned to special memory addresses.

- Reads and writes to these addresses actually send data from/to device.

Dedicated I/O Instructions

- Encode target device and operation in instruction.

- E.g: X86: `IN` *accumulator,port* and `OUT` *port,accumulator*

I/O and O/S and Processor Modes

In most modern computer systems, I/O operations are performed by the operating system rather than directly by user applications.

- If multiple programs are sharing the processor, O/S must protect them from each other by:

– checking all I/O attempts for validity
– preventing two programs from accessing same device at same time

- O/S provides high-level abstractions to encapsulate low-level device operations.

- O/S handles **exceptions** generated by devices.

To prevent user programs from performing I/O directly, the relevant memory-mapped addresses can be marked as invalid.

Most processors have at least two distinct **modes**, e.g., **user** mode and **privileged** mode.

- Ordinary application code runs in user mode.

- O/S code runs in privileged mode.

- Certain instructions (e.g., I/O instructions, `HALT`, etc.) are only valid in privileged mode.

Approaches to I/O Programming

Simplest approach is **programmed I/O**.

- CPU explicitly reads or writes data, one unit at a time.

- Between operations, CPU can **busy wait** (sit in a tight loop), but then it can't do anything else!

```
extern char *output_status_reg, *output_buffer_reg;

void output_buffer(char buff[], int count) {
  int i;
  for (i = 0; i < count; i++) {
    while (*output_status_reg & 0x80 == 0);
        /* Tanenbaum Fig. 5-31 has two bugs */
    *output_buffer_reg = buf[i];
  }
}
```

- Alternatively, CPU can **poll** device periodically to see if a transfer is ready. Makes sense if device is very slow, and there's a convenient place to put polling operation.

Interrupt-driven I/O

To avoid tying up CPU, most devices use **interrupt** mechanism.

• CPU (i.e., O/S) initiates I/O in the usual way, but then goes about its business.

• Device signals when I/O is complete, by using special hardware signals to interrupt normal sequence of CPU instructions.

• When interrupt occurs, CPU (under control of O/S) tends to device and performs next desired operation.

Interrupts are **asynchronous**: they can happen at any time, regardless of what the CPU is doing.

They must be handled **transparently**, so that after the interrupt has been **serviced** the application program can continue as if nothing had happened.

Interrupt Handling Steps

• Device controller asserts an interrupt signal on system bus, and uses other bus signals to identify itself via a small integer id.

• Seeing the interrupt signal, the CPU stops executing instructions and saves the current PC and other processor status information.

• The CPU enters privileged mode and jumps to an interrupt **service routine** (ISR).

– Service routine address may be looked up in a table indexed by interrupting device id; or

– All devices may share single service routine which consults special register to find device id.

• ISR saves program registers.

• ISR performs appropriate action based on type of device and interrupt.

– E.g., output next byte from buffer to device.

• ISR restores saved registers.

• ISR performs a special **return-from-interrupt** instruction that restores the original PC and processor status info.

Interrupt Priorities

If there are **multiple I/O devices**, what happens if one issues an interrupt while another one's interrupt is still being serviced?

Usual solution: assign **priorities** to devices based on their speed. Higher-priority devices can interrupt the ISRs of lower-priority devices, but not vice-versa.

- No danger of losing higher-priority device data.

- Transparency principle applies to ISRs as well as user code.

DMA

For devices with high data rates (e.g., disks, network interfaces), programmed I/O will take too much CPU time to let anything else get done.

Solution: **Direct Memory Access (DMA)** allows fast device controllers to transfer data directly to/from memory.

- CPU initiates transfer by command to controller, specifying

  – starting memory address to read/write
  – number of bytes to transfer
  – direction of transfer

- Controller uses bus to read/write memory directly, bypassing CPU completely.

- When transfer is done, controller interrupts CPU.

DMA controller's use of bus is called **cycle stealing**. If CPU wants bus during DMA transfer it must wait, because device will lose data if denied bus [see homework problem].

DMA works well for large transfers, where setup time per transfer is easily **amortized**. For smaller transfers, interrupt overhead may still be too high.

Traps

Most CPUs support software **traps** (also called **exceptions**).

• Similar to interrupts, but triggered by instruction execution rather than an external device event.

• Normally triggered by uncommon error, such as

– using undefined or protected instruction
– accessing illegal memory
– violating alignment constraint
– arithmetic overflow (if traps enabled)
– hardware malfunctions (e.g., parity violation)

• As with interrupts, current state is saved and control is dispatched to **trap handler** running under O/S control in privileged mode.

• After coping with fault (e.g., by politely terminating user program), CPU returns to user mode using a return-from-interrupt instruction.

Also, many CPU's have `trap` instructions that deliberately cause a trap:

• Can be used to implement privileged system calls.