FIT1047 - Week 4

I/O and Interrupts

Overview

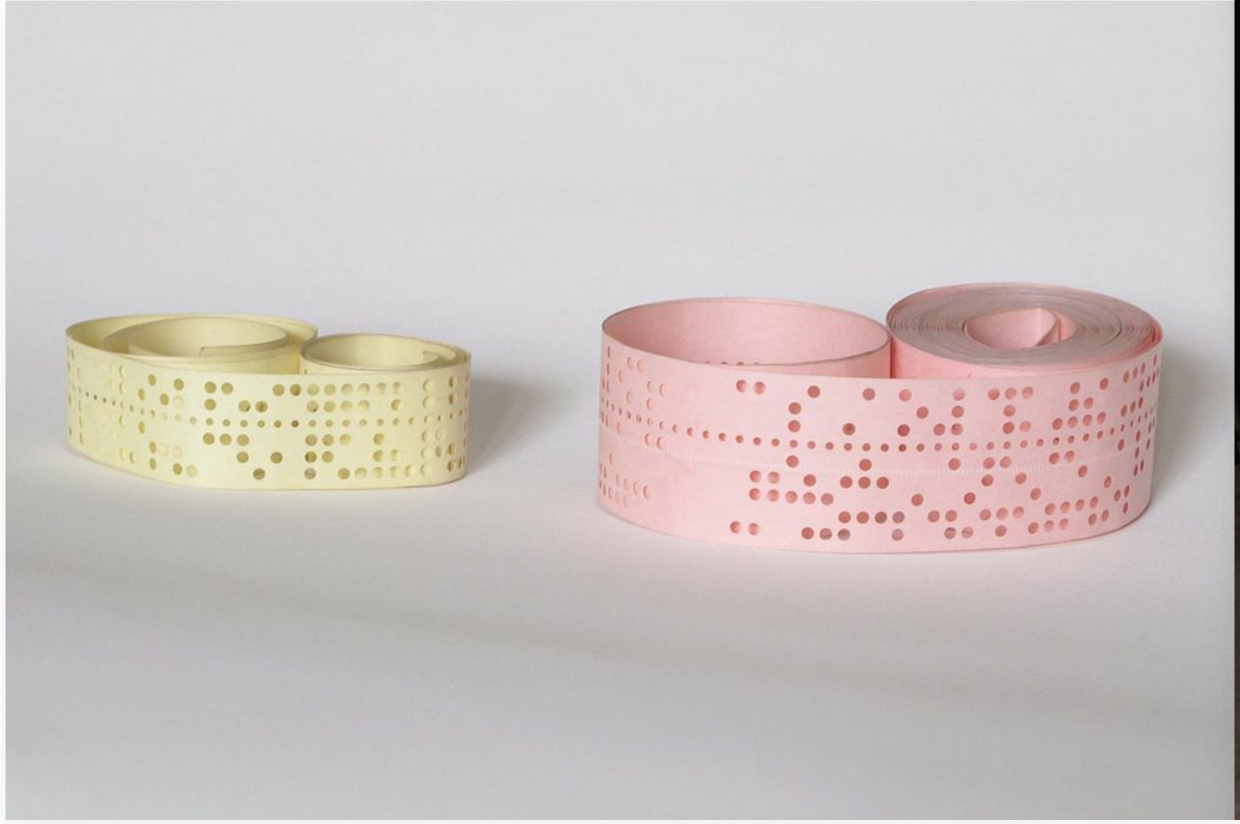
Computers are almost useless without input/output.

- How does the CPU communicate with I/O devices?
- How does it handle time critical I/O?

Early I/O

The first computers had limited I/O:

- Punched paper tape or cards
- Teleprinters



Modern I/O

- Keyboard, mouse, touch pad, touch screen, voice control, gestures, accelerometers, barometers, GPS
- Screens, printers, audio, robots, ...

Also classed as I/O:

- External storage, network devices (WiFi, 4G, Ethernet)

Modern I/O interfaces

I/O devices are now usually connected via interfaces:

- Standardised connectors and protocol
- Can be internal or external
- E.g. USB, SATA, HDMI, PCI Express

I/O and the CPU

The CPU needs to

- read data from an I/O device
- write data to an I/O device

Why write to input devices?

- E.g. set sensitivity, calibrate, ...

Why read from output devices?

- E.g. check if ready for output, check if successful, ...

I/O and the CPU

I/O devices have their own registers.

Two ways to communicate:

- Memory-mapped:
  - I/O registers are mapped into CPU address space.
  - Use Load, Store etc to communicate with I/O.
- Instruction-based:
  - CPU has special I/O instructions.
  - Similar to Load, Store etc but with separate address space.

When to do I/O

Now we know how to communicate with I/O devices.

But most I/O devices are much, much slower than the CPU. So when should the CPU communicate?

- How does it know that a new character is available from the keyboard?
- How does it know a network device is ready for sending?

Programmed I/O

Program checks registers periodically.

Also called polling I/O.

Pseudocode:

```
while (true) {
  if (IORegister1.canRead()) {
    processRegister1();
  } else if (IORegister2.canRead()) {
    processRegister2();
  } else if (IORegister3.canWrite()) {
    processRegister3();
  }
}
```

Programmed I/O

Advantage:

- We can decide how often to poll a device (prioritisation!)

Disadvantage:

- Program is I/O-driven
- CPU is constantly in "busy loop"

Programmed I/O is mostly used in embedded special-purpose systems.

Interrupts

Opposite of polling:

- CPU is notified when I/O is available.
- CPU interrupts what it's doing, processes I/O, then continues normal program.

Interrupt signals

Device notifies CPU of pending interrupt by setting a bit in a special register.

CPU checks before each fetch-decode-execute cycle:

- Is interrupt bit set? Process interrupt.
- Otherwise: fetch-decode-execute.

Interrupt signals

RTL for fetch cycle with interrupts:

```
if InterruptBit is set:
  Clear InterruptBit
  MAR    <- SavePC
  M[MAR] <- PC
  PC     <- InterruptHandler

MAR    <- PC
IR     <- M[MAR]
PC     <- PC+1
```

The interrupt handler code is always stored at a fixed address InterruptHandler.

Interrupt handler

Must leave the CPU in the exact same state!

- For MARIE, contents of AC must be the same as before the interrupt.

Can be achieved by

- shadow registers, a separate register file that the CPU uses while processing interrupts; or
- saving all registers to memory

Interrupt vectors

So how can we process interrupts from different devices?

- Each device is assigned an identification number
- When raising an interrupt, stores that number (in special register or in memory)
- Interrupt handler jumps into an interrupt vector

Interrupt vectors

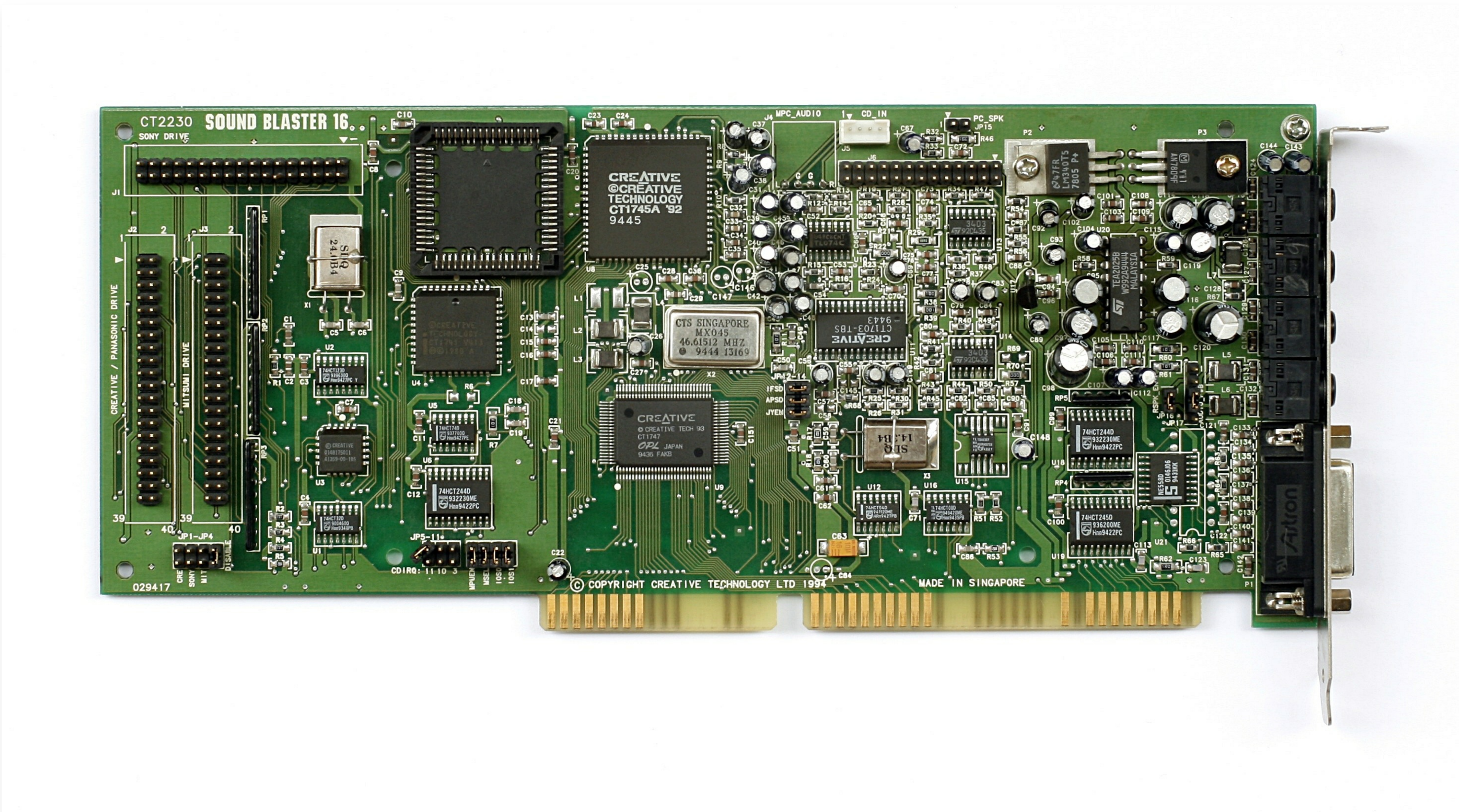
```
B00       Store ACInt
B01       Load  IVec
B02       Add   DeviceID
B03       Store Dest
B04       JumpI Dest

C00 ACInt, HEX 0    / Temporary storage for AC
C01 Dest,  HEX 0    / Destination of jump
C02 IVec,  HEX C03
C03       HEX 0F0  / Address of first handler
C04       HEX 0FA  / Address of second handler
C05       HEX 1B0  / Address of third handler
```

Interrupts in x86 PCs

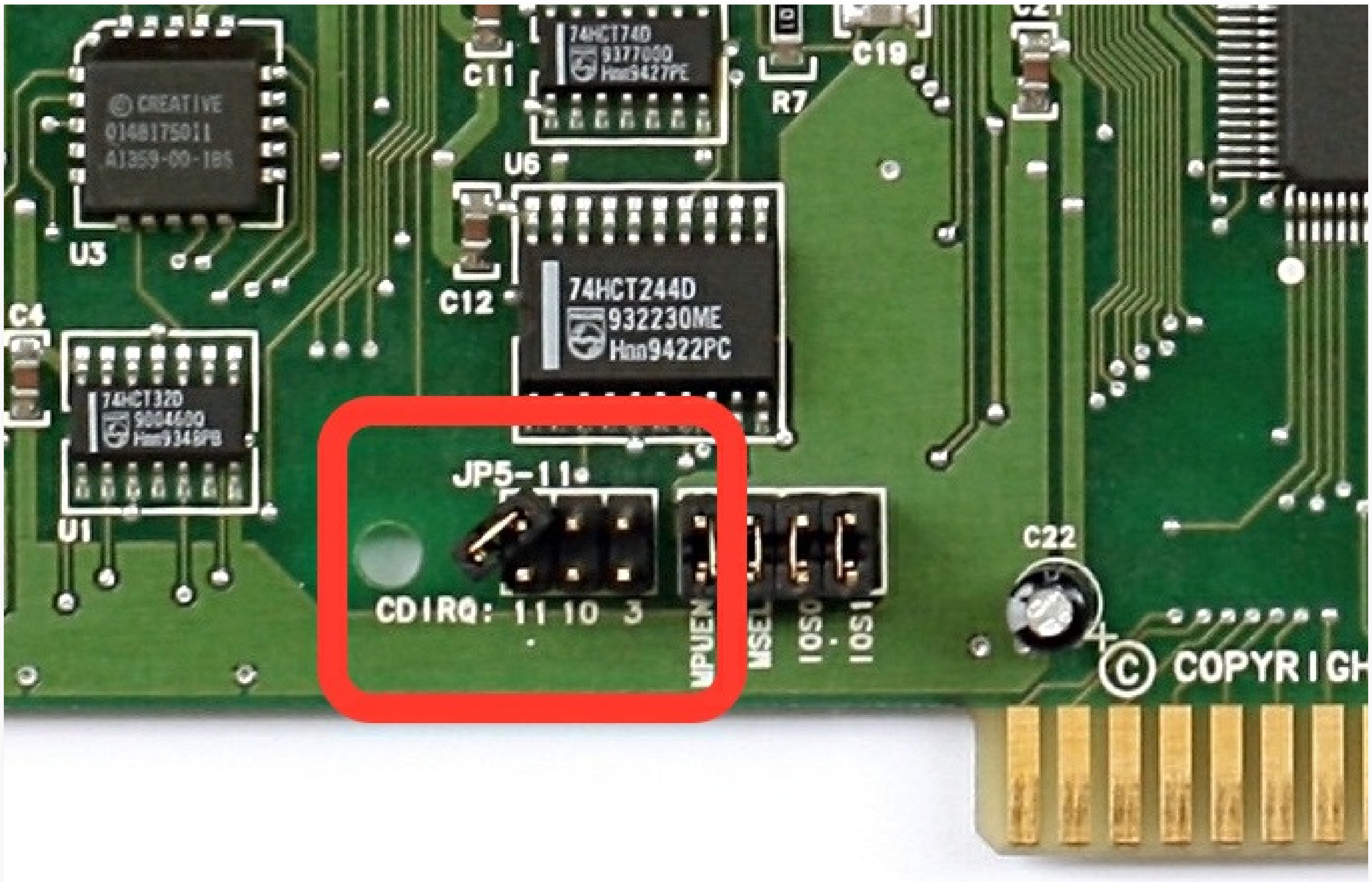
Original design:

- 15 interrupt request (IRQ) signals
- hardware must be configured to use correct IRQ
- e.g. setting jumper on a network or sound card
- devices have to share IRQs

Interrupts in x86 PCs



Interrupts in x86 PCs



Modern Interrupts

Use Advanced Programmable Interrupt Controllers (APICs).

- nowadays integrated into CPUs
- allow more IRQs (fewer conflicts)
- include high-resolution timers

Disadvantages

- Different I/O devices need different priorities
  - can be achieved using different interrupt signals
- All memory transfers run through the CPU:
  - e.g. reads word from disk storage, writes word to memory
  - reads word from memory, writes word to graphics card
- I/O devices are fully controlled by CPU

DMA

(Direct Memory Access)
CPU can delegate memory transfer operations to dedicated controller.

- hard disk controller can transfer data to memory
- graphics card can fetch image from memory
- while CPU can perform other tasks

CPU and DMA controller share the data bus:

- only one can do memory transfers at the same time

Summary

I/O

- memory-mapped vs. instruction-based
- programmed vs. interrupt-driven

Interrupts

- require context switch
- jump into interrupt vector

DMA

- off-load responsibility for data transfers to special controller
- keeps CPU free to do more interesting tasks

Next lecture

- Booting up a computer
- BIOS, UEFI