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

A modern computer (and this includes things like smart phones and washing machines) features many different I/O devices:

- 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, ...

So to the CPU, most devices look like both input and output devices at the same time. And of course things like storage devices and network adapters need to perform both input and output operations.

The next step is to look at how data are transferred between the CPU and the I/O devices.

I/O devices have their own **registers**.

E.g. a keyboard could have a register that holds the last character that was pressed. A network interface could have a register that holds the next byte to be transmitted.

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.

In memory-mapped I/O, the mapping means that certain memory addresses don't refer to actual RAM, but to the registers of an I/O device. Programmers can then use the normal Load and Store instructions to read from and write to the I/O device, which has the advantage of being a very simple programming model. It also means that the instruction set does not need to be modified.

A disadvantage can be that I/O devices now take up *address space*, i.e., their memory addresses cannot be used for regular RAM. For example, MARIE addresses are limited to 12 bits, which

means we can address $2^{12} = 4096$ memory locations. An efficient implementation of memory-mapped I/O would use one bit (e.g. bit 11) to distinguish between RAM and I/O devices, effectively halving the amount of RAM we could use!

An example for a real-world architecture that uses memory-mapped I/O is the family of ARM processors.

Instruction-based I/O is also called *port-mapped* I/O, and the addresses used for a particular I/O device are called *ports*. The instructions look very much like Load and Store instructions, also taking an address as their argument, but the address is interpreted as an I/O port instead of a memory address.

An advantage of this approach is that, since we typically require far fewer I/O addresses than memory addresses, we can use a reduced address width for I/O. E.g., this approach is used in the x86 processor family, where memory addresses are 32 bits wide, but only 16 bits are used for I/O ports. This simplifies the hardware implementation of the logic that decodes I/O port addresses.

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?

There are two fundamental solutions to this problem.

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();
```

```
        }
    }
```

The CPU runs in an infinte loop, asking each I/O device in turn whether there is work to do.

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.

Now the programmer does not have to be aware of I/O at all. The main program will be interrupted when necessary, the CPU jumps into an *interrupt handler* code, and when the handler is finished, continues executing the main program.

Before we look at interrupt handlers in more detail, we need to understand how interrupts are implemented inside the CPU.

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.

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*.

If an interrupt is detectd, the CPU stores the current PC at a special memory location *SavePC*, which the interrupt handler uses as the return address. It then loads the start address of the interrupt handler into the PC.

The remaining RTL is the regular fetch cycle, loading the next instruction from memory and incrementing the PC. If an interrupt occurred, this instruction will be the first instruction of the interrupt handler; otherwise, it will be the next regular instruction of the program code.

The interrupt handler therefore is basically a special subroutine.

Interrupt handler

Must leave the CPU in the exact same state!

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

This concept of saving as much of the CPU state as necessary so that you can return to it at a later point is called a **context switch**. It is an important concept, not only for implementing interrupts, but also for implementing multi-tasking operating systems (where several user programs need to run simultaneously), as we will see later.

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

In the architecture introduced above, the CPU is notified of an interrupt using a single bit.

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**

```
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
```

This is an example of how an interrupt vector could be implemented in MARIE. The handler itself starts at address B00 (that's where the CPU jumps when it detects an interrupt).

The first step is to store the AC register so it can be restored when interrupt processing has finished.

Then it loads the base address of the interrupt vector (IVec, C03), and adds the ID of the device that caused the interrupt. It stores the result in Dest.

E.g., if the keyboard has ID 1, the interrupt handler would add 1 to C03, resulting in C04 being stored in Dest. It then jumps to the address stored at Dest, which in this example is 0FA. That's where the keyboard processing code would then start.

When the interrupt subroutine (e.g. processing the keyboard input) has finished, it needs to do two things:

```
Load  ACInt   / Restore AC register
JumpI SavePC  / Return to interrupted code
```

In these examples we simplified the system a bit by ignoring the fact that another interrupt could happen while processing an interrupt. Typically, systems are designed so that further interrupts are disabled during this time.

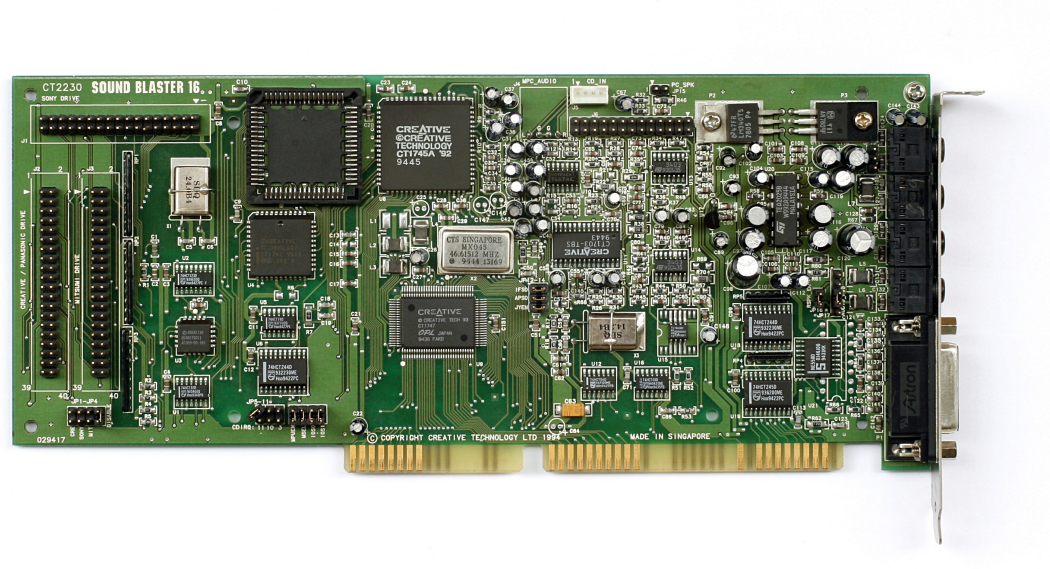
Interrupts in x86 PCs

Original design:

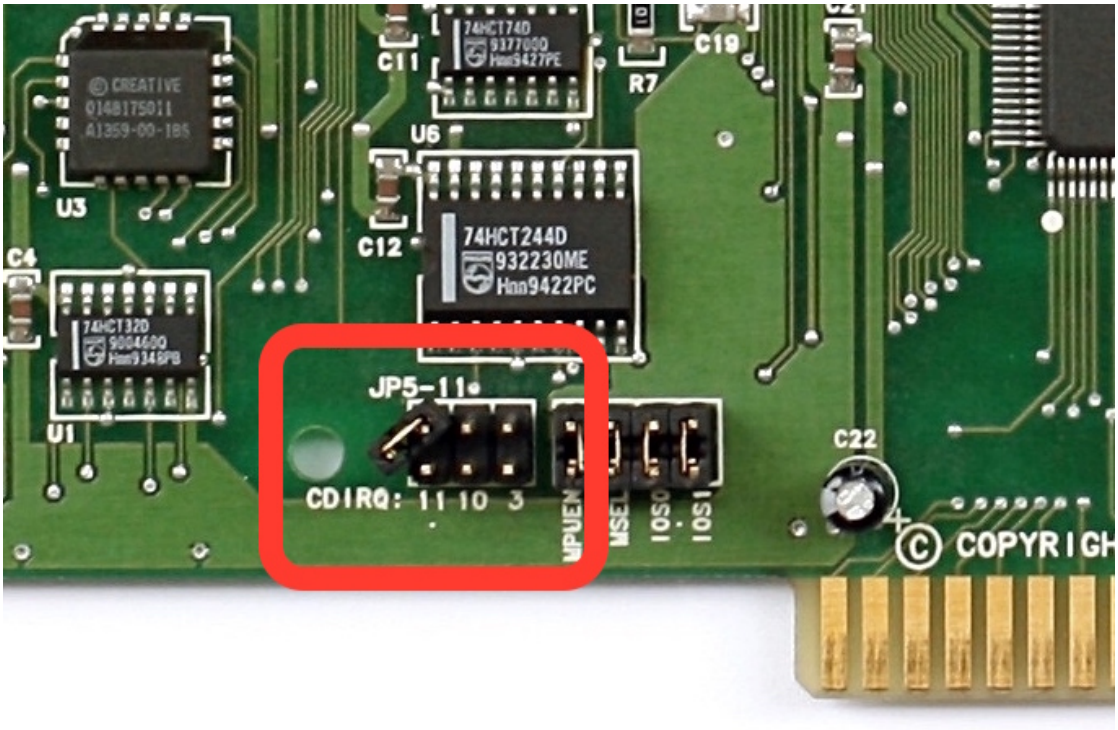
- 15 interrupt request (IRQ) signals
- hardware must be configured to use correct IRQ

- e.g. setting jumper on a network or sound card
- devices have to *share* IRQs

Interrupts in x86 PCs



This is an old *SoundBlaster* sound card for PCs as they were popular in the 1990s. Wen can zoom in on one part of the circuit board:



Here you see a set of *jumpers*, small connectors that can be closed or left open. In this particular case, the jumpers are used to configure the IRQ number for the sound card. It can be set to 11, 10 or 3 (and if you had more than one extension card in your computer, you better made sure the IRQ settings didn't clash!).

Modern Interrupts

Use *Advanced Programmable Interrupt Controllers* (APICs).

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

Timers are another important source of interrupts. They enable the CPU to e.g. execute a particular subroutine every few milliseconds, or switch between different programs.

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

In order to relieve the CPU of the task of just copying data to and from memory, modern architectures implement a feature called *Direct Memory Access* (DMA).

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