FIT1047 - Week 4

Part 2:
I/O and Interrupts

MONASH University

Recap

We've now seen

- CPUs
- Memory

There's one component missing to complete the picture:

Input and Output

Overview

Computers are useless without input/output.

Today:

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

Early I/O

The first computers had very limited I/O.

- Punched paper tape or cards
- Teleprinters

Image source: Wikipedia

Modern I/O

- Keyboard, mouse, touch pad, touch screen, voice control, gestures, accelerometers, barometers, GPS, …
- Screens, printers, audio, smart home appliances, robots, …

Also classed as I/O:

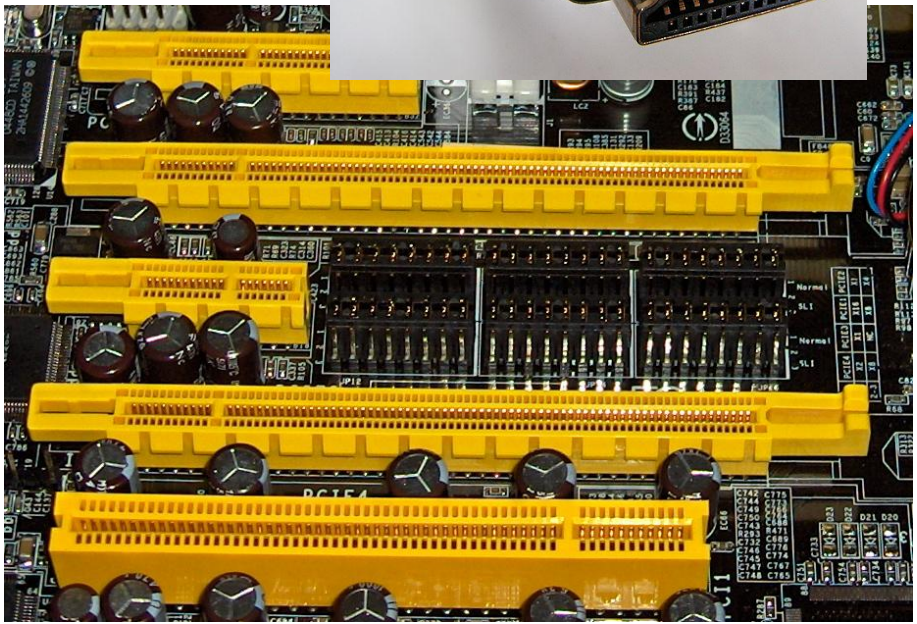
- Storage (hard disks, SSDs, SD cards)
- Network devices (WiFi, 4G, Ethernet)

Modern I/O interfaces

I/O devices are now usually connected via *interfaces:*

- Standardised connectors and protocols
- Can be internal or external

Examples: USB, PCIe, HDMI, ...



I/O and the CPU

The CPU needs to be able to

- Receive data from an I/O device
- Send data to an I/O device

Why send to input devices?

- E.g. set sensitivity, calibrate, switch on/off, …

Why receive from output devices?

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

I/O and the CPU

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

E.g. keyboard device: register holds the currently pressed key

How can we access these registers in machine code?

Two methods: memory-mapped and instruction-based I/O.

I/O Access Method 1: **Memory-mapped**

Each I/O register is **“mapped” to a special memory address**. We can then use Load/Store.

Example: keyboard register is mapped to address A000.

- `Load A000` loads currently pressed key into AC register.

Example: printer register is mapped to address A100.

- `Store A100` stores current AC value in printer register (which then gets printed out).

I/O Access Method 1: **Memory-mapped**

Advantages:

- No need for new instructions
- Simple

Disadvantages:

- We cannot use “mapped” addresses for memory any more
- The overall amount of usable memory is reduced (because some addresses are now unavailable)
- Programs may accidentally access I/O (when a program has a bug)

I/O Access Method 2: **Instruction-based**

Add special I/O instructions to CPU ISA. For example, **`Input`** and **`Output`**.

Each I/O register still has an address (as in memory-mapped I/O).

But now these addresses are *separate* from the memory.

Example:

`Load A000` loads value from memory address A000

`Input A000` loads value from I/O register A000 (e.g. the keyboard)

When to perform 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 with the device?

- How does it know that a key has been pressed?
- How does it know a printer is ready for receiving the next character?
- …

Two methods: programmed and interrupt-based I/O.

Programmed I/O

Programmer adds checks into code to periodically check I/O registers.

Also called **polling** I/O.

Pseudocode:

```
while (true) {
  if (keyboardRegister.canRead()) {
    processKeyboardRegister();
  } else if (printerRegister.canWrite()) {
    processPrinterRegister();
  }
}
```

Programmed I/O

Advantages:

- Very simple (no extra hardware needed)
- Programmer can decide how often to poll
  (e.g. 10000 times/second for network, 10 times/second for keyboard)

Disadvantages:

- Programmer must be careful (poll registers regularly)
- Program is “I/O driven” (constructed around I/O)
- CPU is constantly in “busy loop”, causing high power usage

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

Interrupts

Opposite of polling:

- CPU is **notified** when it should communicate with I/O device
- CPU interrupts current program, executes special **interrupt handler** code, then continues normal program
- Programmer can write separate code for main program and interrupt handler (better separation of concerns)

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? Call interrupt handler.
- Otherwise: normal fetch-decode-execute.

Let’s look at the RTL for this.

RTL for Fetch with Interrupts

An interrupt handler is like a subroutine:

- Use `JnS` to
  - Store the **return address**
  - Jump to the code implementing the handler
- When the handler finishes, return to caller using `JumpI`

RTL for `JnS X`:

6. MBR ← PC
7. M[MAR] ← MBR

Save current PC at X

8. PC ← MAR
9. PC ← PC + 1

Jump to X+1

RTL for Fetch with Interrupts

1. If InterruptBit is 1:
   - 1a. Clear InterruptBit
   - 1b. MAR ← InterruptHandler
   - 1c. MBR ← PC
   - 1d. M[MAR] ← MBR

   Save current PC at **`InterruptHandler`**

   - 1e. PC ← MAR
   - 1f. PC ← PC + 1

   Jump to **`InterruptHandler`**+1

2. MAR ← PC
3. MBR ← M[MAR]
4. IR ← MBR
5. PC ← PC+1
6. ...

Normal fetch cycle

Interrupt Handler

Must leave the CPU in the same state as before the interrupt!

- For MARIE: Contents of AC must be the same as before

This is called a **context switch.** It can be achieved by

- *Shadow registers*:
  When an interrupt happens, the CPU switches to a separate register file

Or

- Programming the interrupt handler to save registers to memory
  (see next example)

Interrupt Vectors

How can the interrupt handler distinguish interrupts from different devices?

- Each device is assigned an **identification number**
- When raising an interrupt, the device **stores its number** in a special register
- The interrupt handler uses that number to jump to different subroutines
- The list of subroutines (one per device) is called an **interrupt vector**

Example Interrupt Handler (MARIE syntax)

```
InterruptHandler, HEX 0
                  Store SaveAC
                  Load InterruptVecAddr
                  Add InterruptDeviceID
                  Store Destination
                  JumpI Destination

SaveAC,           HEX 0                  / Temporary storage for AC register
Destination,      HEX 0                  / Computed destination handler
InterruptVecAddr, ADR InterruptVec
InterruptVec,     ADR KeyboardHandler    / device 0
                  ADR MouseHandler       / device 1
                  ADR PrinterHandler     / device 2
                  ADR SoundHandler       / device 3

KeyboardHandler,  ... / do some work
                  Load SaveAC
                  JumpI InterruptHandler
```

Save AC register to memory

Add device number to address of interrupt vector

Jump to device handler

Restore AC register from memory

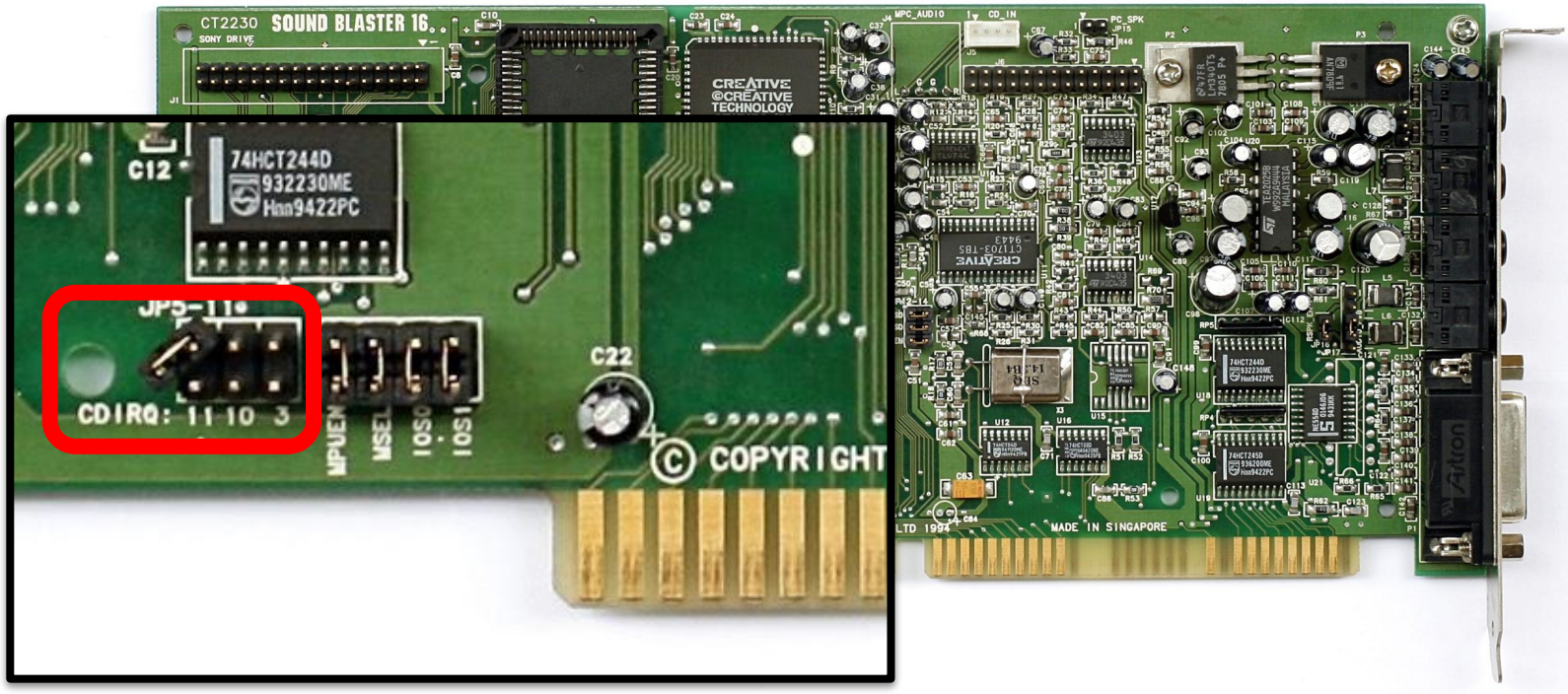
Return to normal program

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



Modern Interrupts

Use *Advanced Programmable Interrupt Controllers* (APICs).

- Integrated into CPUs
- Support more IRQs (results in fewer conflicts)
- Include high-resolution timers
  - E.g. cause an interrupt every millisecond
  - We will see later why that's useful!

Disadvantages of interrupt-based I/O

- Different devices may need different priorities
  - E.g. keyboard (very low) vs graphics card (very high priority)
  - Can be achieved using different interrupt signals
- All memory transfers run through the CPU
  - E.g. read byte from disk, store into memory; or
  - Read byte from memory, transfer to graphics card
- I/O devices are fully controlled by CPU

Solution: DMA

Direct Memory Access (DMA)

Modern CPUs can **delegate** memory transfer to dedicated controller.

- Hard disk controller can transfer data directly to RAM
- Graphics card can fetch image directly from RAM
- CPU is free to perform other tasks

CPU and DMA controller **share the data and address bus:**

- Only one can perform memory transfer at the same time

Summary

I/O

- Memory-mapped vs. instruction based
- Programmed (polling) vs interrupt-driven

Interrupts

- Require *context switch* (call subroutine to handle interrupt, save registers)
- Jump into *interrupt vector* (one handler per device)

DMA

- Off-load responsibility for data transfer to special controller
- Keeps CPU free to do useful tasks

Outlook

Next lecture:

- Booting a computer
- BIOS / UEFI