

# Lecture 23

---

- I/O, Interrupts, exceptions

# A Timely Question.

---

- Most modern operating systems **pre-emptively** schedule programs.
  - If you are simultaneously running two programs A and B, the O/S will periodically switch between them, as it sees fit.
  - Specifically, the O/S will:
    - Stop A from running
    - Copy A's register values to memory
    - Copy B's register values from memory
    - Start B running
- How does the O/S stop program A?

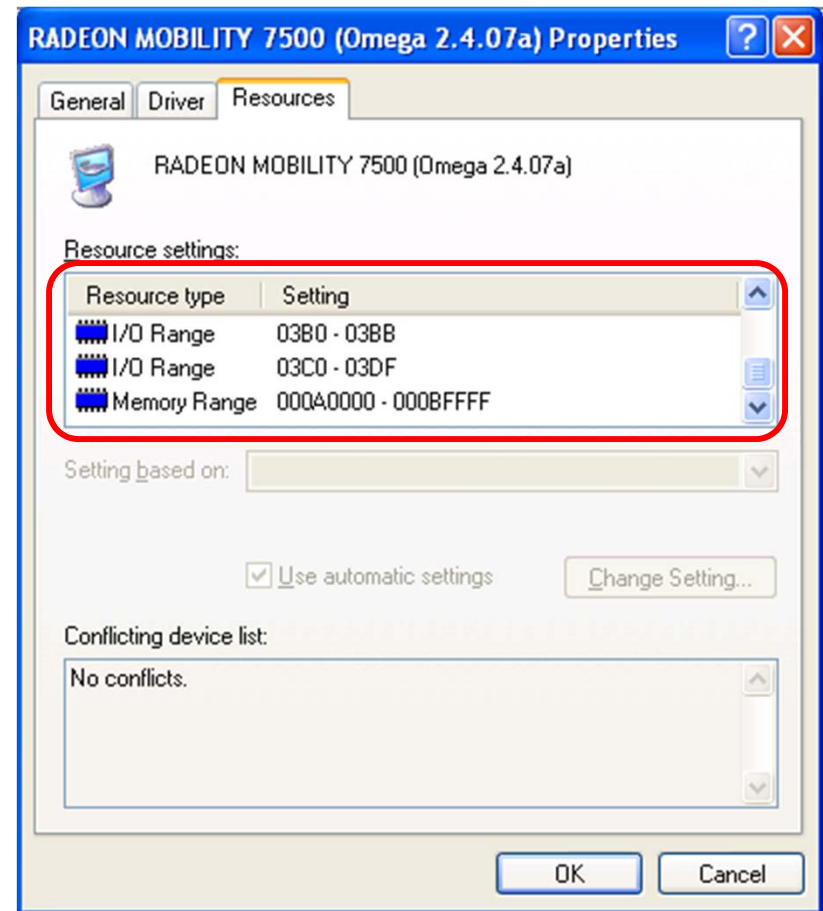
# I/O Programming, Interrupts, and Exceptions

---

- Most I/O requests are made by applications or the operating system, and involve moving data between a peripheral device and main memory.
- There are two main ways that programs communicate with devices.
  - Memory-mapped I/O
  - Isolated I/O
- There are also several ways of managing data transfers between devices and main memory.
  - Programmed I/O
  - Interrupt-driven I/O
  - Direct memory access
- Interrupt-driven I/O motivates a discussion about:
  - Interrupts
  - Exceptions
  - and how to program them...

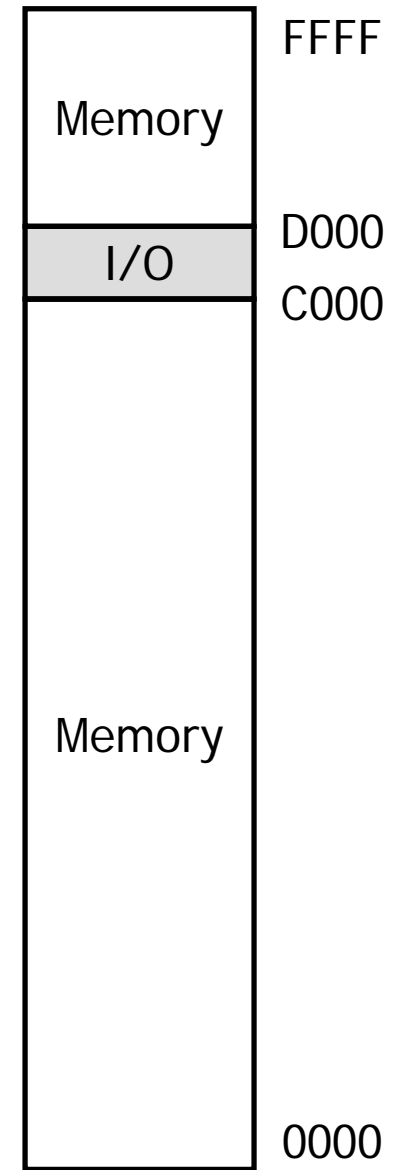
# Communicating with devices

- Most devices can be considered as memories, with an “address” for reading or writing.
- Many instruction sets often make this analogy explicit. To transfer data to or from a particular device, the CPU can access special addresses.
- Here you can see a video card can be accessed via addresses 3B0-3BB, 3C0-3DF and A0000-BFFFF.
- There are two ways these addresses can be accessed.

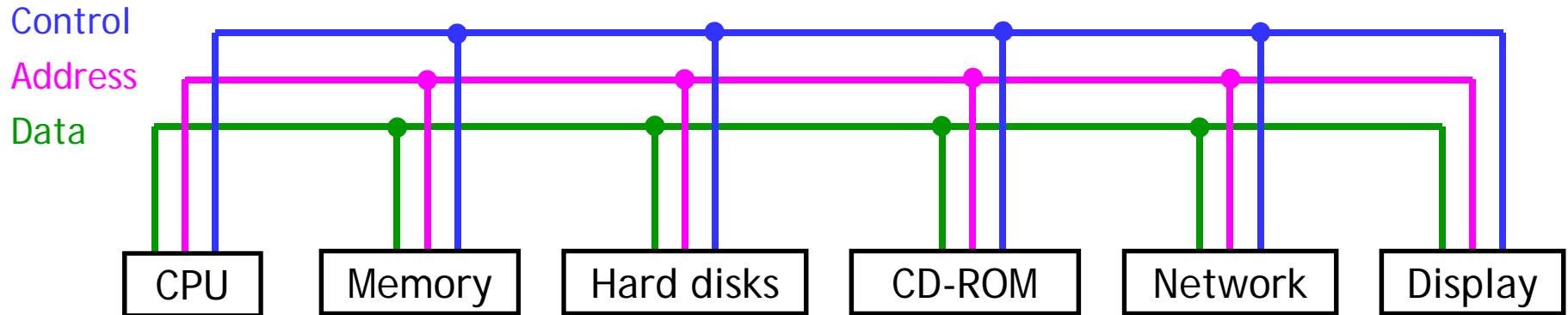


# Memory-mapped I/O

- With **memory-mapped I/O**, one address space is divided into two parts.
  - Some addresses refer to physical memory locations.
  - Other addresses actually reference peripherals.
- For example, an Apple IIe had a 16-bit address bus which could access a whole 64KB of memory.
  - Addresses **C000-CFFF** in hexadecimal were not part of memory, but were used to access I/O devices.
  - All the other addresses did reference main memory.
- The I/O addresses are shared by many peripherals. In the Apple IIe, for instance, C010 is attached to the keyboard while C030 goes to the speaker.
- Some devices may need several I/O addresses.



# Programming memory-mapped I/O

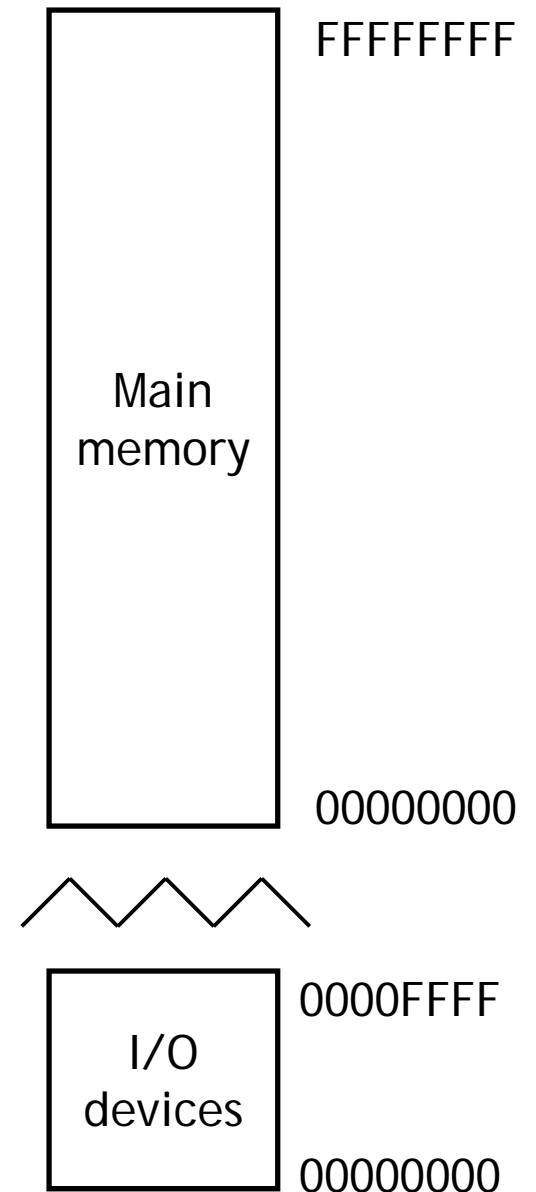


- To send data to a device, the CPU writes to the appropriate I/O address. The address and data are then transmitted along the bus.
- Each device has to monitor the address bus to see if it is the target.
  - The Apple IIe main memory ignores any transactions whose address begins with bits 1100 (addresses C000-CFFF).
  - The speaker only responds when C030 appears on the address bus.



# Isolated I/O

- Another approach is to support *separate* address spaces for memory and I/O devices, with special instructions that access the I/O space.
- For instance, 8086 machines have a 32-bit address space.
  - Regular instructions like **MOV** reference RAM.
  - The special instructions **IN** and **OUT** access a separate 64KB I/O address space.



# Comparing memory-mapped and isolated I/O

---

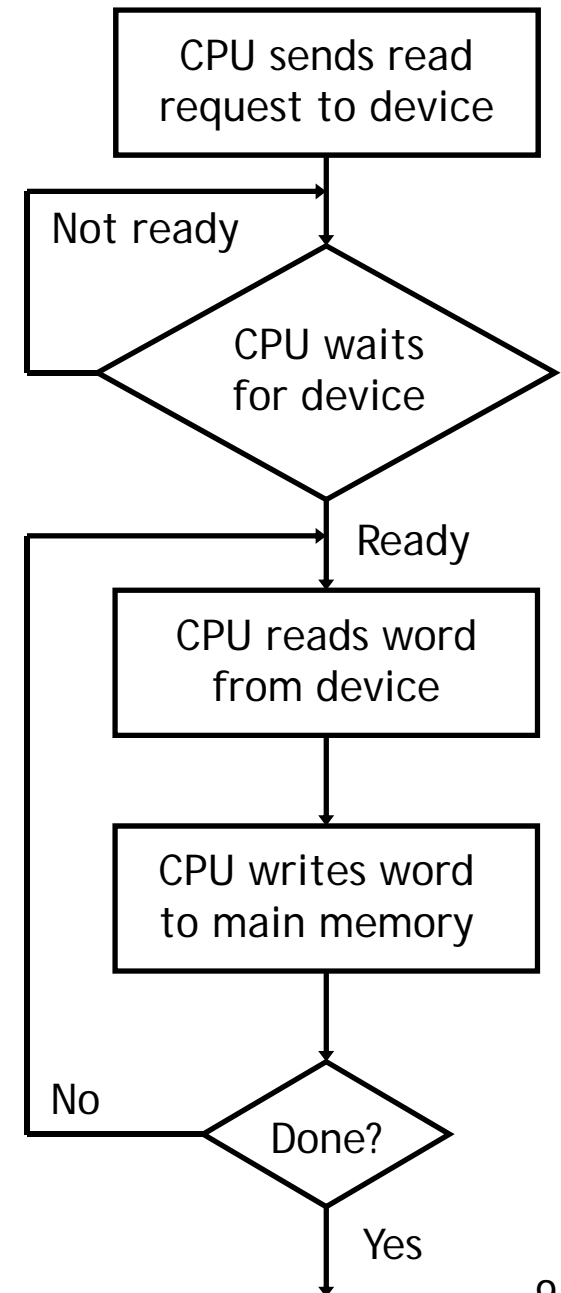
- Memory-mapped I/O with a single address space is nice because the same instructions that access memory can also access I/O devices.
  - For example, issuing MIPS sw instructions to the proper addresses can store data to an external device.
- With isolated I/O, special instructions are used to access devices.
  - This is less flexible for programming.
  - May have advantages for bandwidth/throughput (memory and I/O busses separate)



# Transferring data with programmed I/O

- The second important question is how data is transferred between a device and memory.
- Under **programmed I/O**, it's all up to a user program or the operating system.
  - The CPU makes a request and then waits for the device to become ready (*e.g.*, to move the disk head).
  - Buses are only 32-64 bits wide, so the last few steps are repeated for large transfers.
- A lot of CPU time is needed for this!
  - If the device is slow the CPU might have to wait a long time—as we will see, most devices *are* slow compared to modern CPUs.
  - The CPU is also involved as a middleman for the actual data transfer.

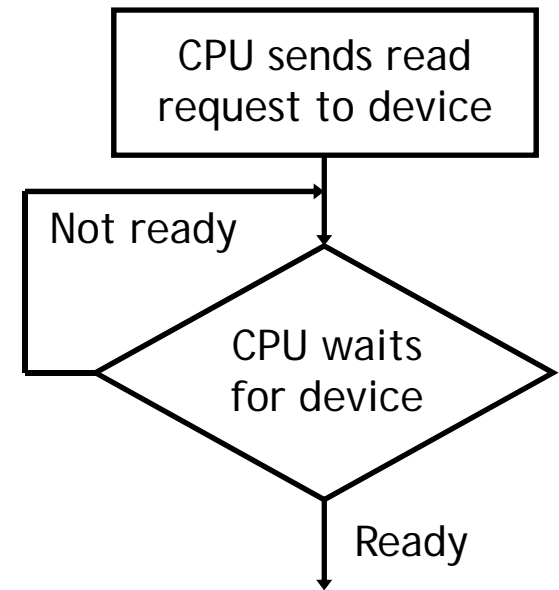
(This CPU flowchart is based on one from *Computer Organization and Architecture* by William Stallings.)



# Can you hear me now? Can you hear me now?

---

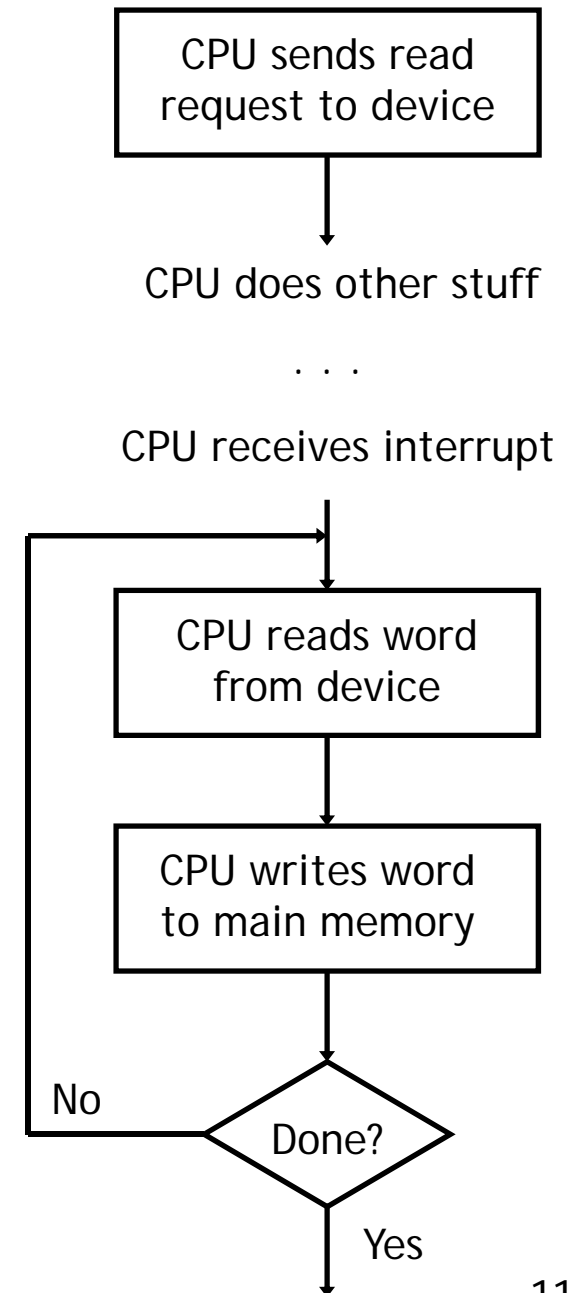
- Continually checking to see if a device is ready is called **polling**.
- It's not a particularly efficient use of the CPU.
  - The CPU repeatedly asks the device if it's ready or not.
  - The processor has to ask often enough to ensure that it doesn't miss anything, which means it can't do much else while waiting.
- An analogy is waiting for your car to be fixed.
  - You could call the mechanic every minute, but that takes up all your time.
  - A better idea is to wait for the mechanic to call *you*.



# Interrupt-driven I/O

- **Interrupt-driven I/O** attacks the problem of the processor having to wait for a slow device.
- Instead of waiting, the CPU continues with other calculations. The device **interrupts** the processor when the data is ready.
- The data transfer steps are still the same as with programmed I/O, and still occupy the CPU.

(Flowchart based on Stallings again.)



# Interrupts

---

- **Interrupts** are external events that require the processor's attention.
  - Peripherals and other I/O devices may need attention.
  - Timer interrupts to mark the passage of time.
- These situations are not errors.
  - They happen normally.
  - All interrupts are recoverable:
    - The interrupted program will need to be resumed after the interrupt is handled.
- It is the operating system's responsibility to do the right thing, such as:
  - Save the current state.
  - Find and load the correct data from the hard disk
  - Transfer data to/from the I/O device.

# Exception handling

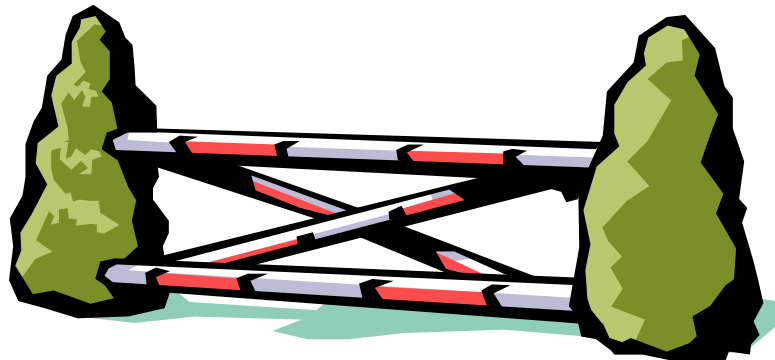
---

- **Exceptions** are typically errors that are detected within the processor.
  - The CPU tries to execute an illegal instruction opcode.
  - An arithmetic instruction overflows, or attempts to divide by 0.
  - The a load or store cannot complete because it is accessing a virtual address currently on disk
    - Virtual Memory!
- There are two possible ways of resolving these errors.
  - If the error is **un-recoverable**, the operating system kills the program.
  - Less serious problems can often be fixed by the O/S or the program itself.

# How interrupts/exceptions are handled

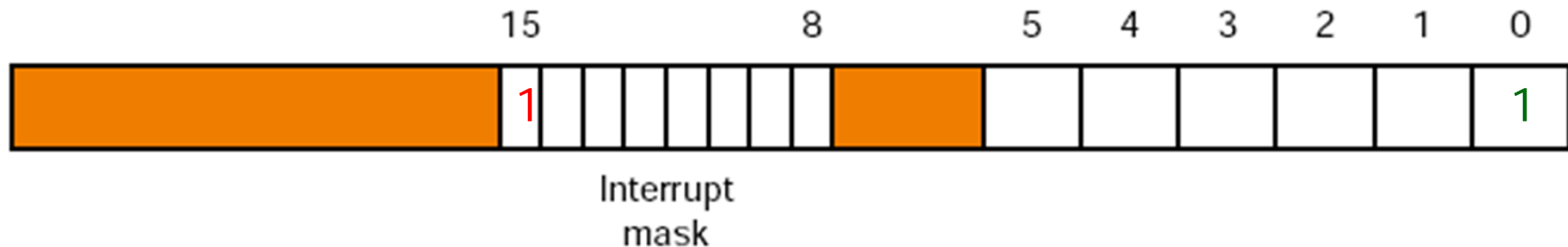
---

- For simplicity exceptions and interrupts are handled the same way.
- When an exception/interrupt occurs, we stop execution and transfer control to the operating system, which executes an “**exception handler**” to decide how it should be processed.
- The exception handler needs to know two things.
  - The cause of the exception (e.g., overflow or illegal opcode).
  - What instruction was executing when the exception occurred. This helps the operating system report the error or resume the program.
- This is another example of interaction between software and hardware, as the cause and current instruction must be supplied to the operating system by the processor.



# MIPS Interrupt Programming

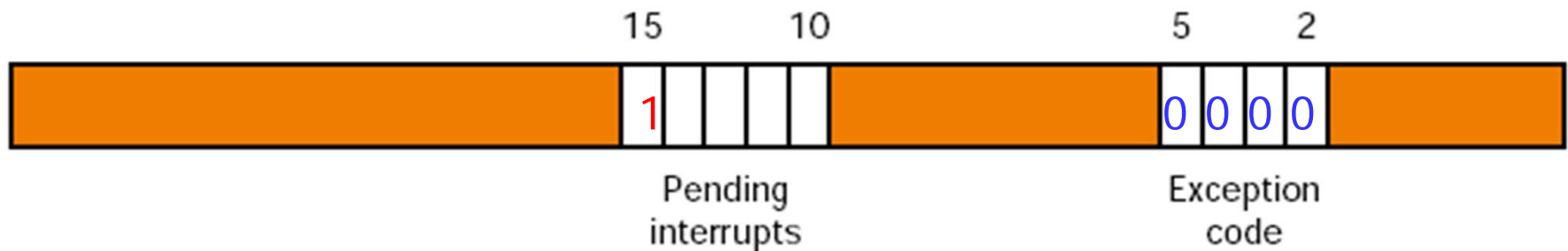
- In order to receive interrupts, the software has to enable them.
  - On a MIPS processor, this is done by writing to the [Status register](#).
    - Interrupts are enabled by setting [bit zero](#).



- MIPS has multiple interrupt levels
  - Interrupts for different levels can be selectively enabled.
  - To receive an interrupt, its bit in the [interrupt mask](#) (bits 8-15 of the Status register) must be set.
    - In the Figure, interrupt level 15 is enabled.

# MIPS Interrupt Programming

- When an interrupt occurs, the Cause register indicates which one.
  - For an **exception**, the **exception code field** holds the exception type.
  - For an **interrupt**, the **exception code field** is **0000** and bits will be set for pending interrupts.
    - The register below shows a pending interrupt at level 15



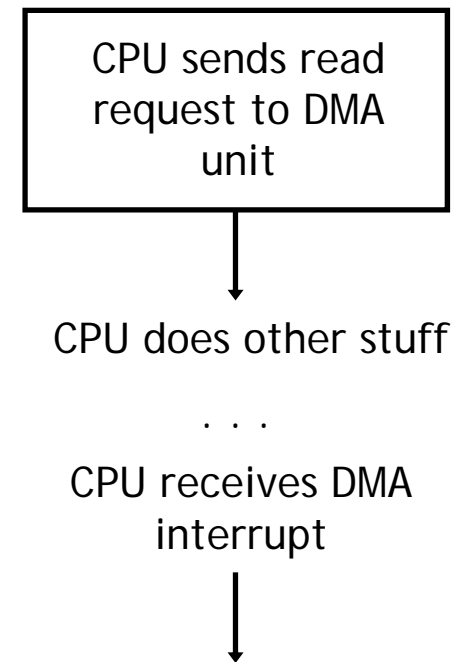
- The exception handler is generally part of the operating system.



# Direct memory access

---

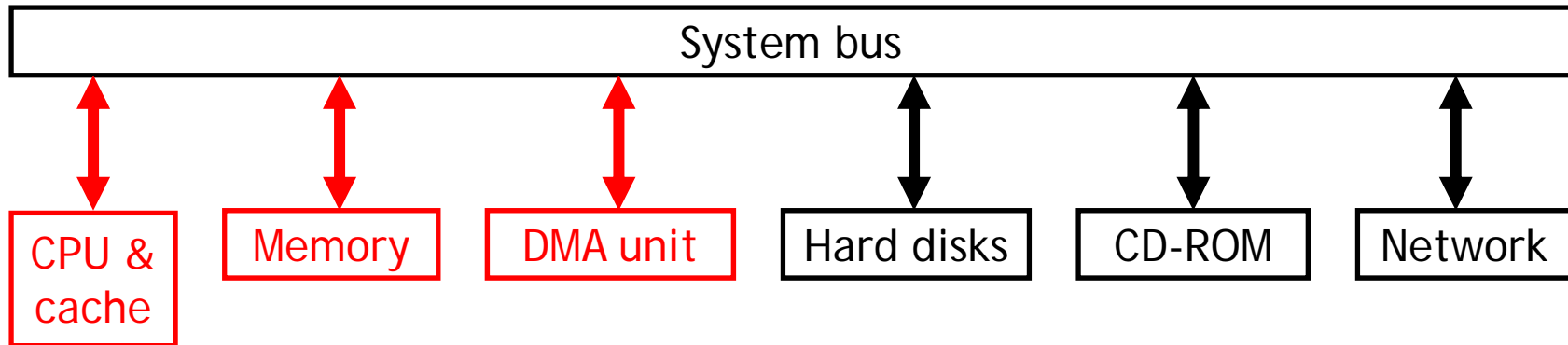
- One final method of data transfer is to introduce a **direct memory access**, or **DMA**, controller.
- The DMA controller is a simple processor which does most of the functions that the CPU would otherwise have to handle.
  - The CPU asks the DMA controller to transfer data between a device and main memory. After that, the CPU can continue with other tasks.
  - The DMA controller issues requests to the right I/O device, waits, and manages the transfers between the device and main memory.
  - Once finished, the DMA controller interrupts the CPU.



(Flowchart again.)

# Main memory problems

---



- As you might guess, there are some complications with DMA.
  - Since both the processor and the DMA controller may need to access main memory, some form of arbitration is required.
    - Who is the “bus master” at any given time?
  - If the DMA unit writes to a memory location that is also contained in the cache, the cache and memory could become inconsistent.
    - Can be done automatically by the cache controller, or by explicit CPU instructions invalidating part of the cache, or ...