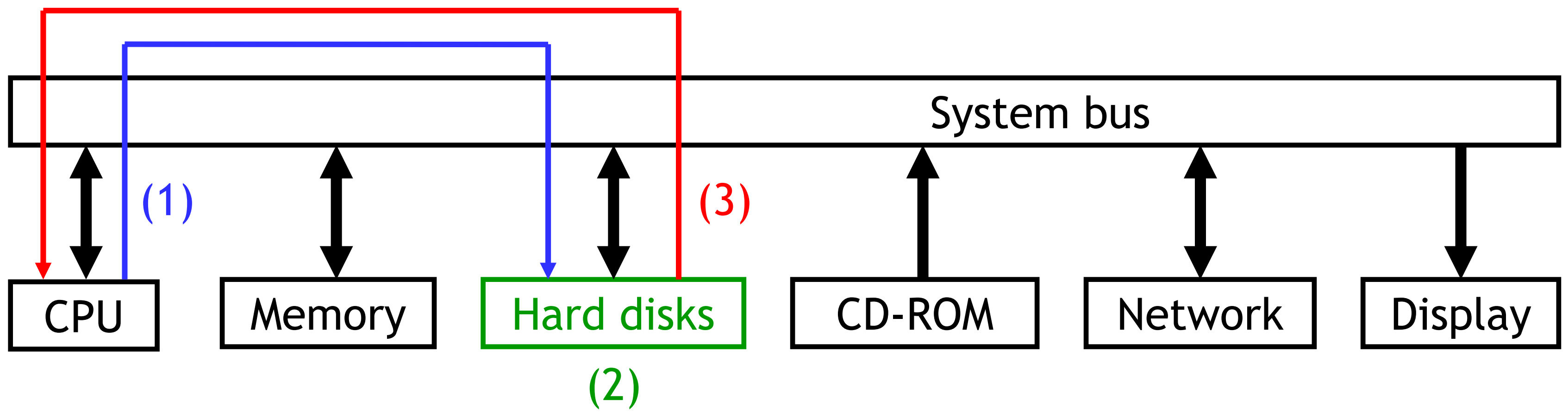
Buses

- There are two main ingredients to I/O systems.
  - Devices like hard drives and networks provide input and output.
  - Buses connect devices to each other and the processor.
- Today we’ll focus on buses and address several issues.
  - What signals are sent and received over a bus?
  - How can multiple devices transmit data across a single bus?
  - Who controls data transfers across the bus?

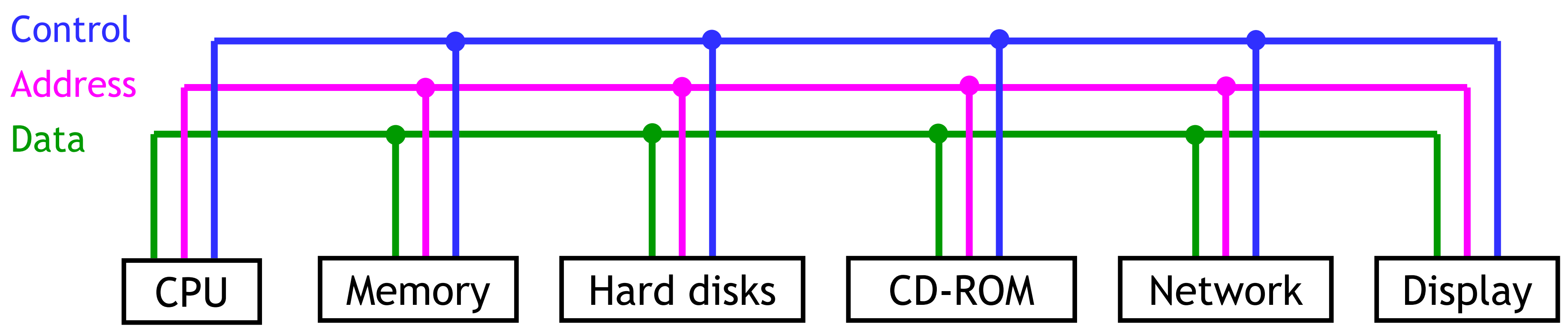
Basic bus protocols

- A bus is a shared path that connects several devices together. Devices can communicate with one another over the bus.
- It's convenient to think of I/O operations as memory operations, in which case two devices might interact as follows.
  1. An initiator sends an address and data over the bus to a target.
  2. The target processes the request by "reading" or "writing" data.
  3. The target sends a reply over the bus back to the initiator.
- The bus width limits the number of bits transferred per cycle.



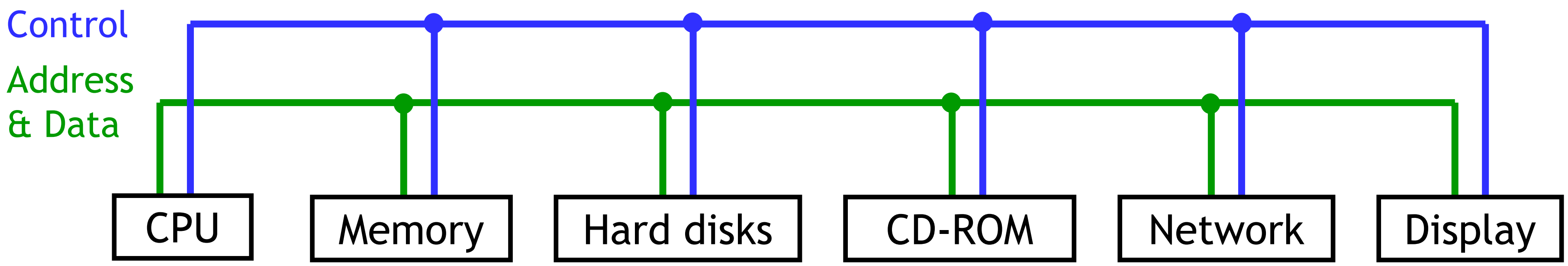
What is the bus anyway?

- A bus is just a bunch of wires which transmits three kinds of information.
  - Control signals specify commands like "read" or "write."
  - The location on the device to read or write is the address.
  - Finally, there is also the actual data being transferred.
- Some buses include separate control, address and data lines, so all of this information can be sent in one clock cycle.

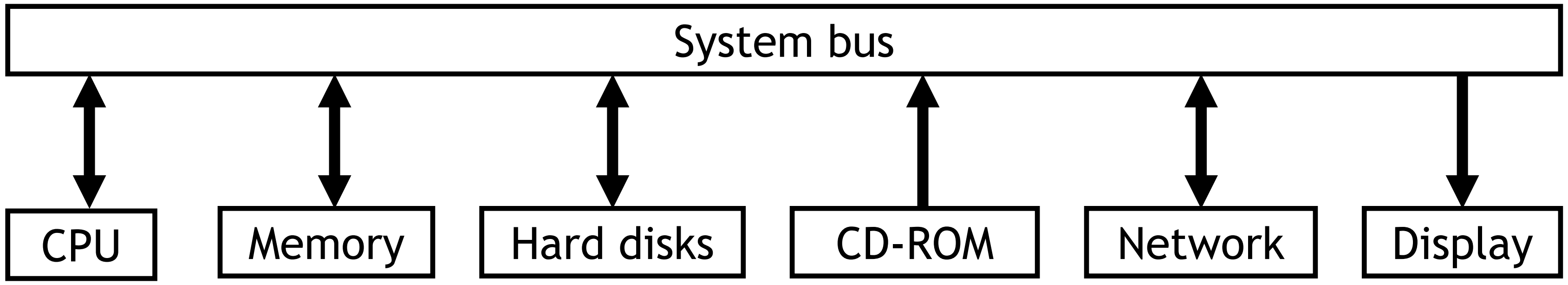


Multiplexed bus lines

- Unfortunately, this could lead to many wires, occupying a lot of physical space and electrically interfering with each other.
  - Many buses transfer 32 to 64 bits of data at a time.
  - Addresses are usually at least 32-bits long.
- Another common approach is to multiplex some lines.
  - For example, we can use the same lines to send both the address and the data, one after the other.
  - The drawback is that now it takes *two* cycles to transmit both pieces of information.



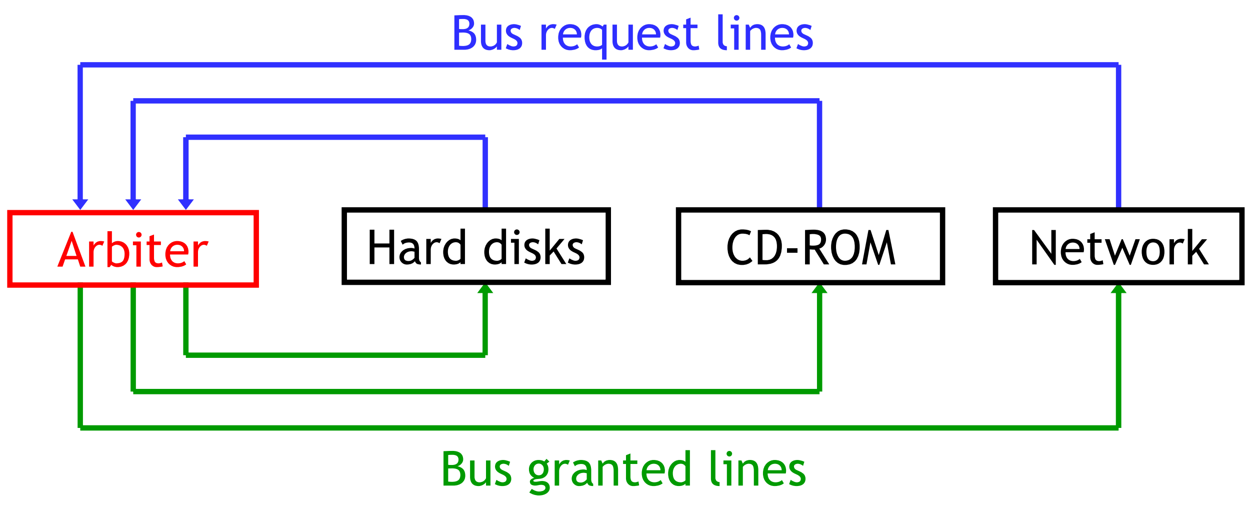
Arbitration



- Only one device may transfer data at a time on the bus.
- Arbitration is the process of deciding which device has priority for using the bus, when many devices all need to transmit data. There are three main schemes.
  - Centralized arbitration
  - Daisy-chained arbitration
  - Distributed arbitration

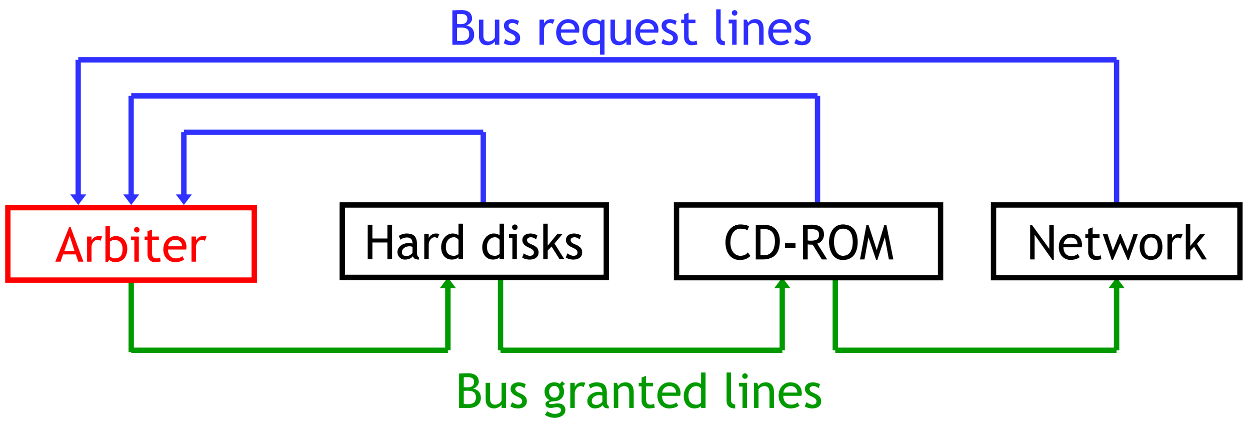
Centralized arbitration

- One idea is to have all devices ask a central arbiter for permission before using the bus. If several devices need the bus in the same cycle, then the arbiter determines who goes first.
  - Each device has a request line and a granted line to the arbiter.
  - Devices that need to use the bus will set their request lines.
  - The arbiter will set *one* of the granted lines in response.
- The PCI bus in most desktop computers uses central arbitration.



Daisy-chained arbitration

- If devices have fixed priorities, a daisy-chained method can work well.
  - A grant line goes from the arbiter to the highest-priority device, then the device with the next-highest priority, and so forth.
  - In the picture below, if the hard disk needs the bus, it will intercept the granted signal. The CD-ROM only gets access to the bus if the disk doesn't want it.
- If we're not careful, this arbitration method may not be fair. Low-priority devices may *never* get to access the bus.



Distributed arbitration

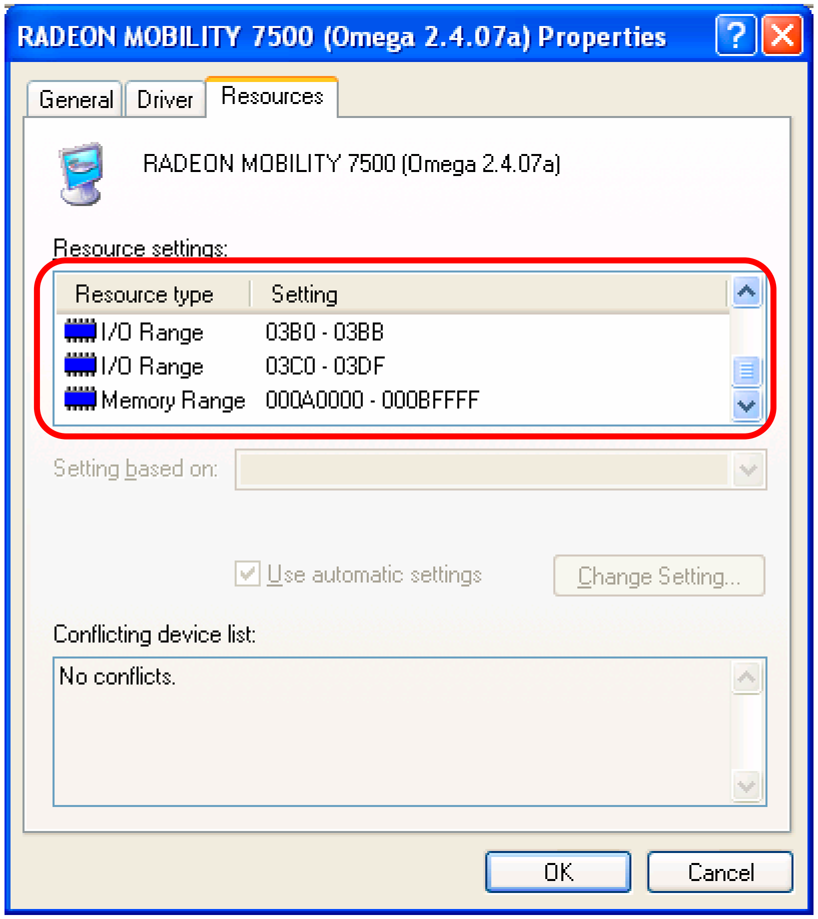
- With distributed arbitration schemes, the devices themselves negotiate with each other to determine who gets the bus.
- One example is Ethernet, which uses collision detection.
  - Any device can start sending data at any time.
  - Devices must be able to detect collisions when other nodes send data at the same time. If this happens, then the devices all stop and wait a random amount of time before retrying.
- People do this in real life too, such as when many people start talking at once or when cars arrive at an intersection simultaneously.

I/O requests

- 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

Communicating with devices

- We already mentioned that 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.



RADEON MOBILITY 7500 (Omega 2.4.07a) Properties

RADEON MOBILITY 7500 (Omega 2.4.07a)

Resource settings:

Resource type	Setting
I/O Range	03B0 - 03BB
I/O Range	03C0 - 03DF
Memory Range	000A0000 - 000BFFFF

Setting based on:

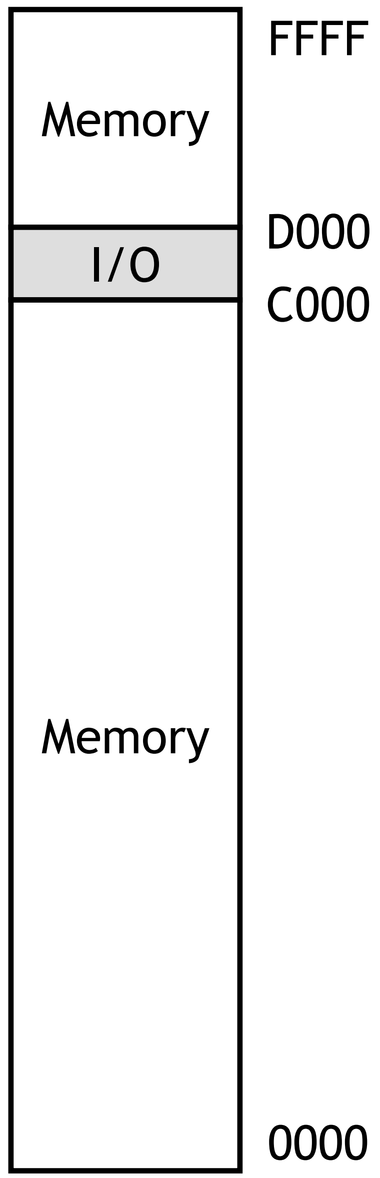
[✓] Use automatic settings

Conflicting device list:

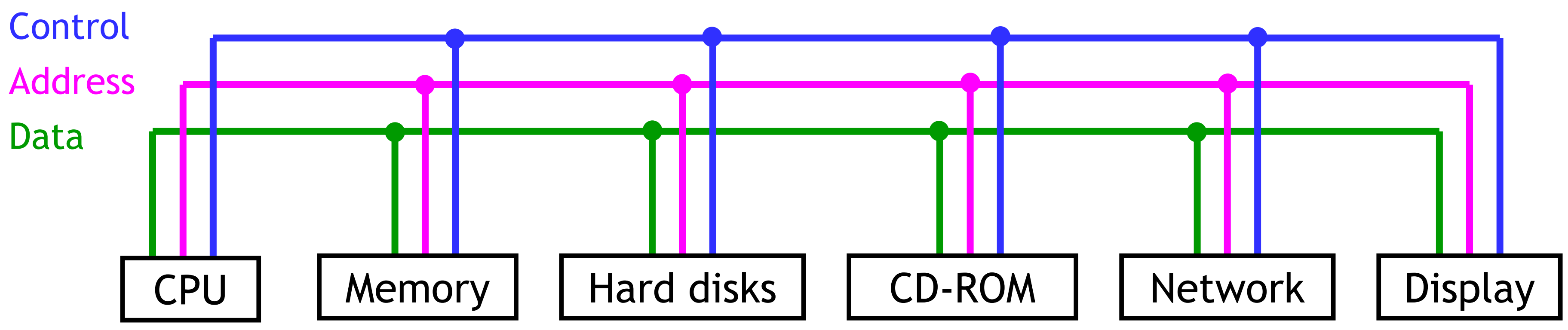
No conflicts.

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, my old 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.



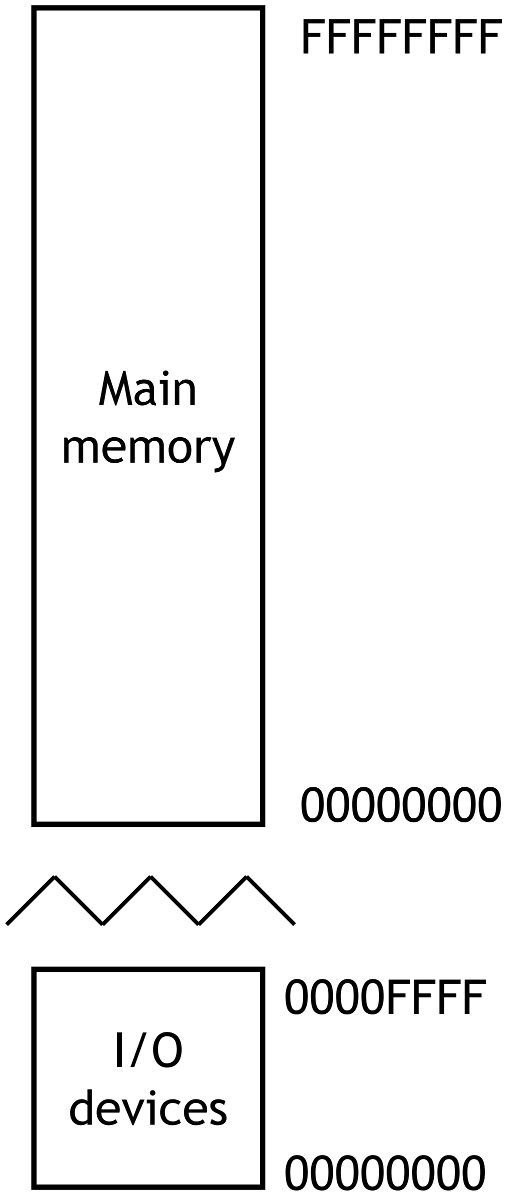
Programming memory-mapped I/O



- 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.
  - Address 0000FFFF could refer to *either* main memory *or* an I/O device, depending on what instruction was used.



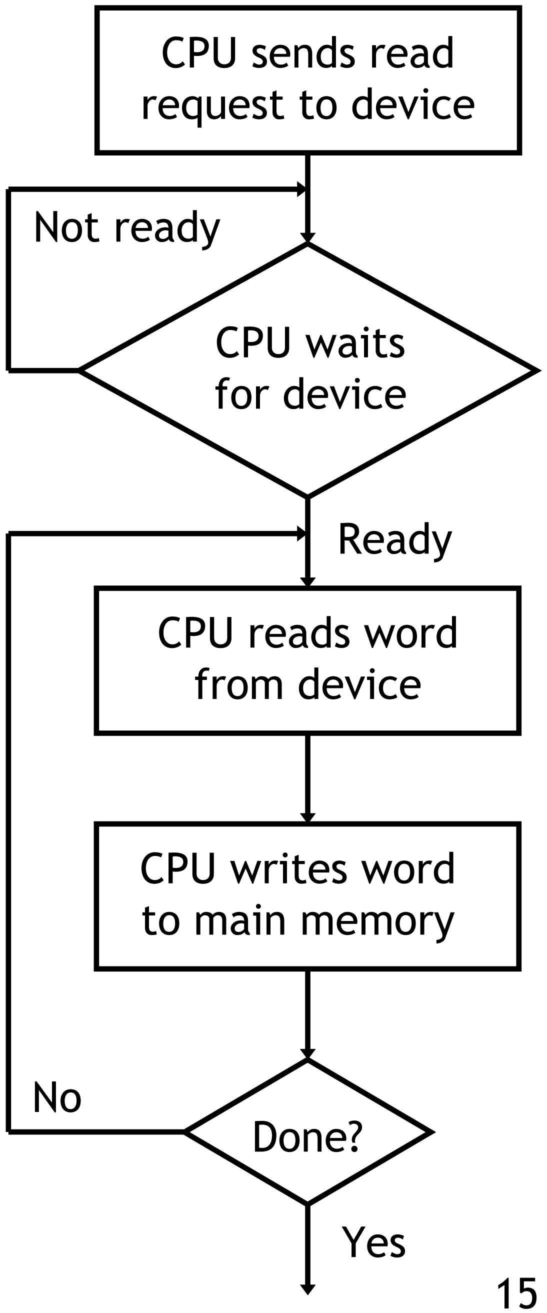
Comparing memory-mapped and isolated I/O

- 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.
  - However, part of the address space is taken by I/O devices, reducing the amount of main memory that's accessible.
- With isolated I/O, special instructions are used to access devices.
  - This is less flexible for programming.
  - On the other hand, I/O and memory addresses are kept separate, so the amount of accessible memory isn't affected by I/O devices.

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 already saw, most devices *are* slow compared to modern CPUs.
  - The CPU is also involved as a middleman for the actual data transfer.

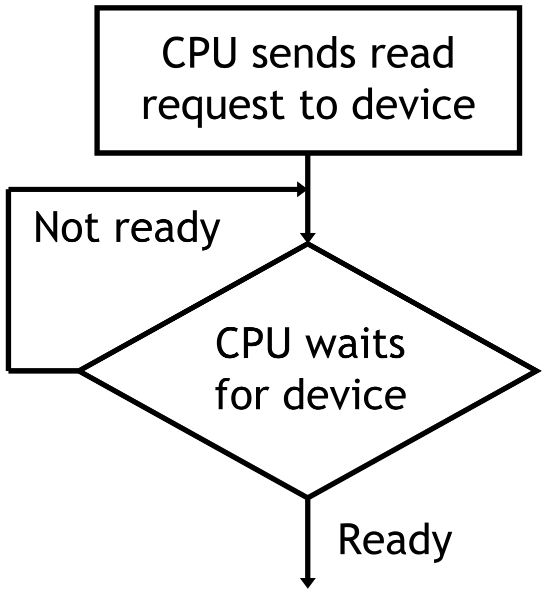
(This CPU flowchart is based on one from *Computer Organization and Architecture* by William Stallings.)



CPU sends read request to device
Not ready
CPU waits for device
Ready
CPU reads word from device
CPU writes word to main memory
No
Done?
Yes

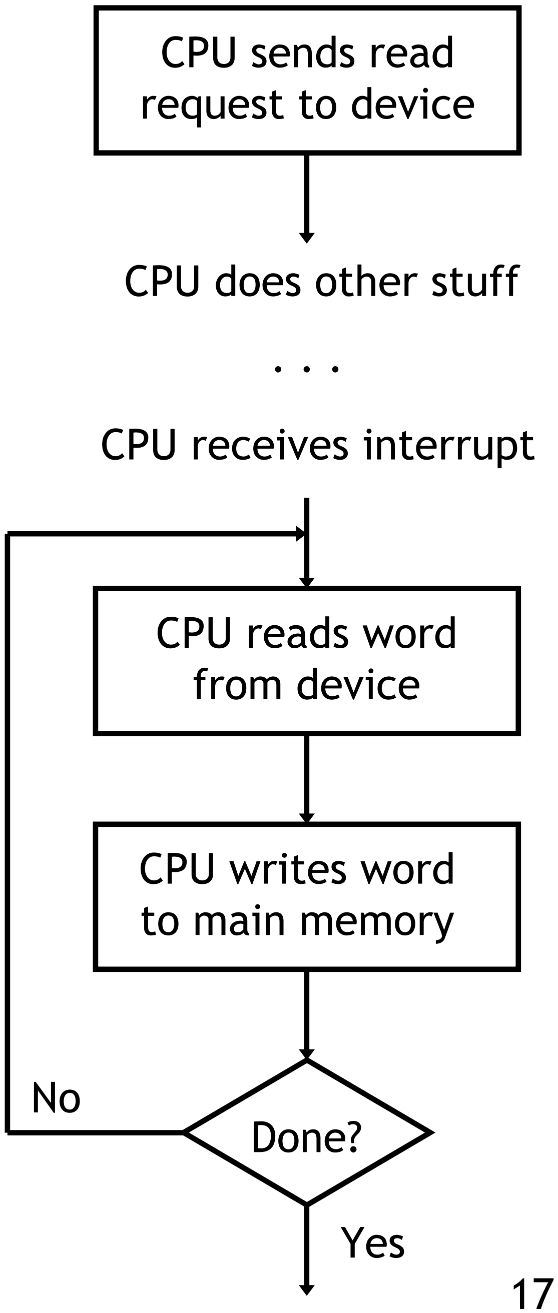
Can you hear me now? Can you hear me now?

- 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.
- Device interrupts are similar to exceptions.
  - They are typically handled by the operating system, which performs the data transfer.
  - See Lecture 19 for a little more information about exceptions.



CPU sends read request to device
CPU does other stuff
. . .
CPU receives interrupt
CPU reads word from device
CPU writes word to main memory
Done?
No
Yes

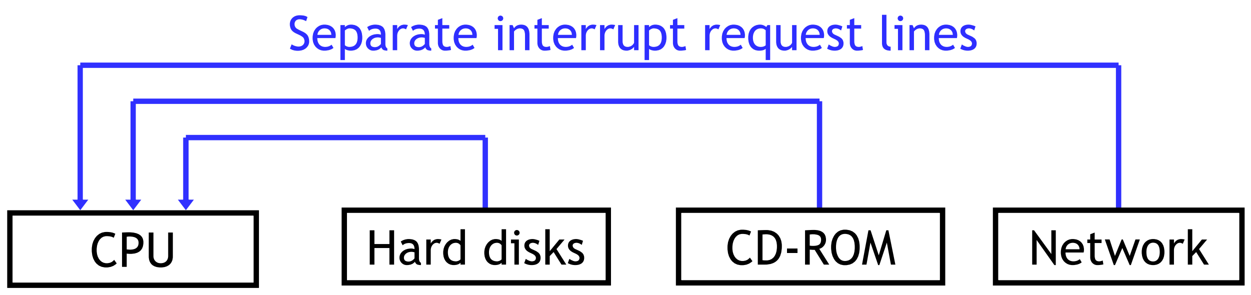
(Flowchart based on Stallings again.)

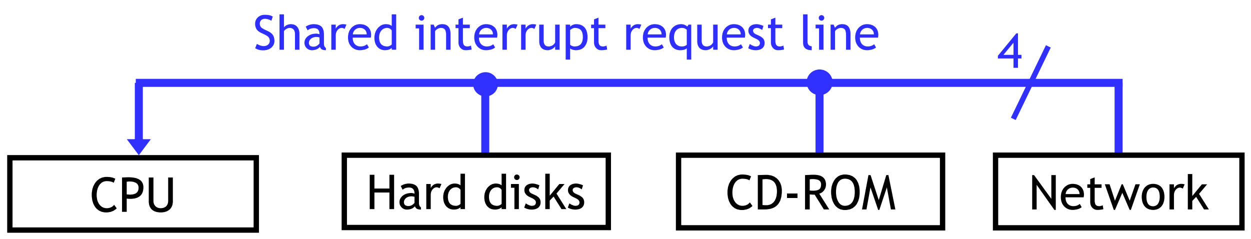
Multiple interrupts

- If the CPU continues running while waiting for a device, it’s possible that another process will make another I/O request.
- There could be several outstanding I/O requests at any time, which also means that several devices could interrupt the CPU at the same time.
- In this case, the processor must determine two things.
  - Which devices sent interrupts.
  - Which device has priority and should be handled first.
- There are several possible solutions.
  - Use multiple interrupt signals.
  - Perform software polling.
  - Enable hardware daisy-chaining.

Multiple interrupt signals

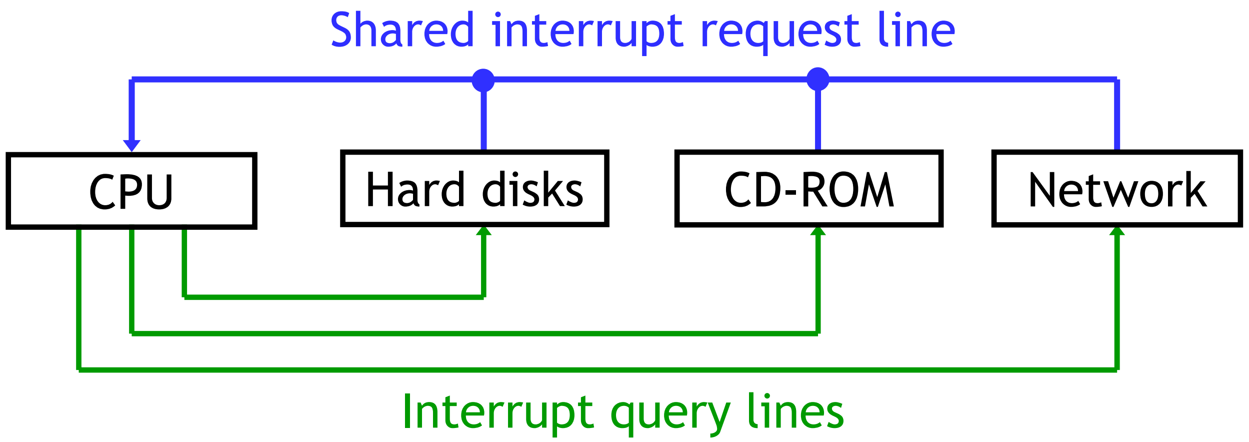
- With multiple interrupt signals, the CPU can easily tell which devices sent interrupts.
  - Each device might have its own interrupt signal line.
  - Devices could also send an identification code along a single shared interrupt line; the hard disk might be device 1110, for example.
- The processor then determines which devices have priority, and handles those first.





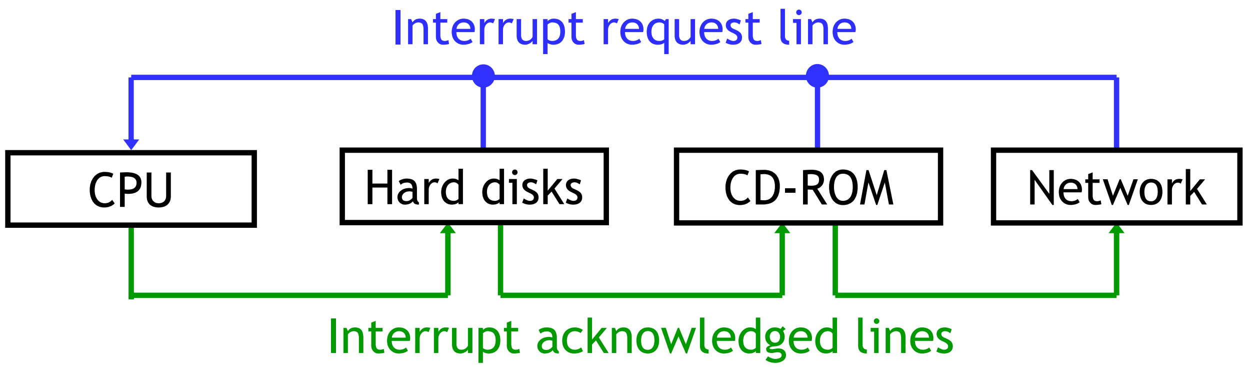
Software polling

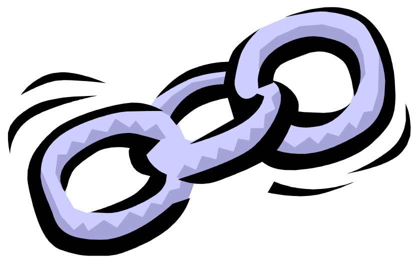
- If there is only one interrupt request line, software polling can be used.
  - The operating system asks each device if it sent an interrupt.
  - The order in which devices are queried reflects the priorities.
- This is an inefficient use of CPU time, especially with many devices.



Daisy-chaining

- This is the same basic idea as before, but we use it here to prioritize CPU interrupts instead of bus requests.
  - Each device can send an interrupt signal to the processor.
  - Only one of them will be "acknowledged" by the CPU. Priorities are determined by position in the daisy chain.

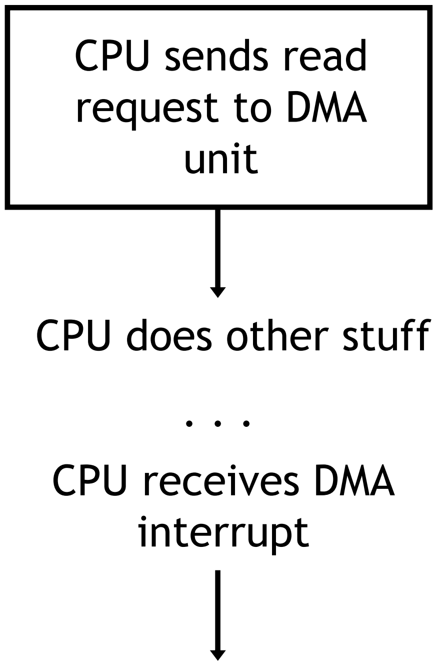




- Other methods are also possible, such as having an arbiter decide which device gets to interrupt the CPU first.

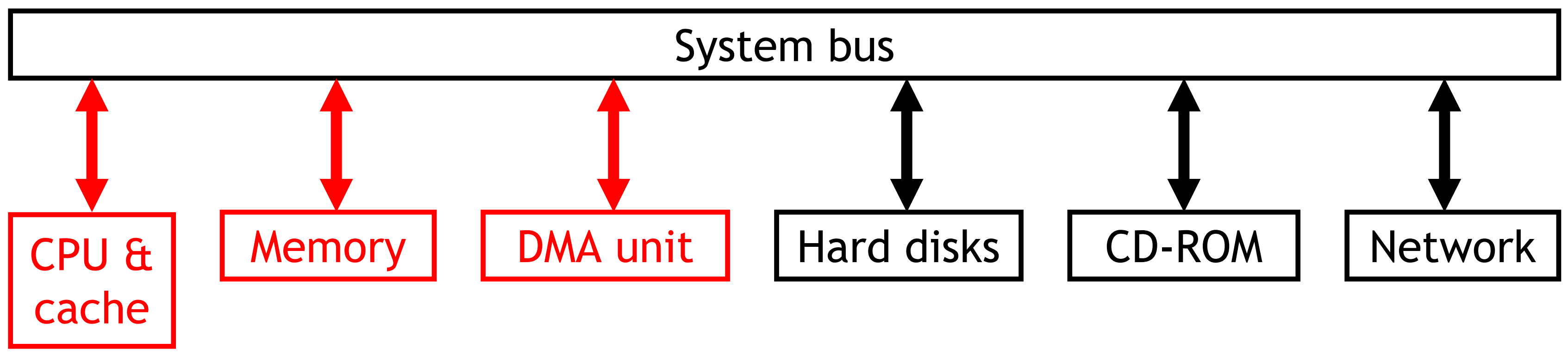
Direct memory access

- 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.
- This is yet another form of parallel processing.


CPU sends read request to DMA unit
CPU does other stuff
. . .
CPU receives DMA interrupt

(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, there could be a lot of bus contention.
  - If the DMA unit writes to a memory location that is also contained in the cache, the cache and memory could become inconsistent.
- Having the main processor handle all data transfers is less efficient, but easier from a design standpoint!

Summary

- Buses connect multiple devices together.
- Only one device may transmit data on the bus at any time, so arbitration schemes are needed to decide which devices have priority.
- Processors communicate with I/O devices in two main ways.
  - Memory-mapped I/O uses standard data transfer instructions.
  - Isolated I/O requires special instructions in the ISA.
- Actual data transfers can be done by the CPU using either programmed or interrupt-driven I/O, or they can be offloaded to a DMA controller.
- Next time we'll see examples of PC buses and get ready for the final!