

# Lecture 09:

# Device Input and Output

CS343 – Operating Systems  
Branden Ghen a – Spring 2022

Some slides borrowed from:

Stephen Tarzia (Northwestern), Shivaram Venkataraman (Wisconsin), Jaswinder Pal Singh (Princeton), and UC Berkeley CS61C and CS162

# Administrivia

- Exam and PCLab grades will be out this week
  - Soon! By sometime on Wednesday (a few late students)
- Drop deadline is this week as well
  - I'm happy to meet with you if you're concerned
- QueueLab due on Thursday
  - Way more code to write than PCLab, so get going!

# Today's Goals

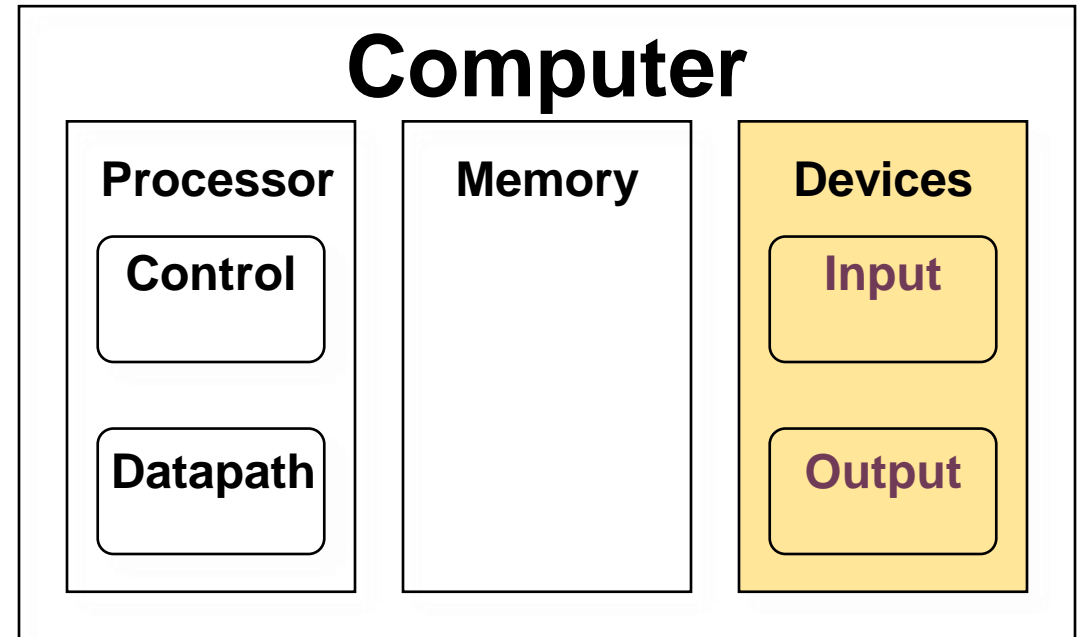
- Discuss I/O devices and how a computer connects to them.
- Understand two different methods of reading/writing device data.
- Explore patterns for device interaction:
  - Synchronous versus Asynchronous
  - Programmed I/O versus Direct Memory Access

# Outline

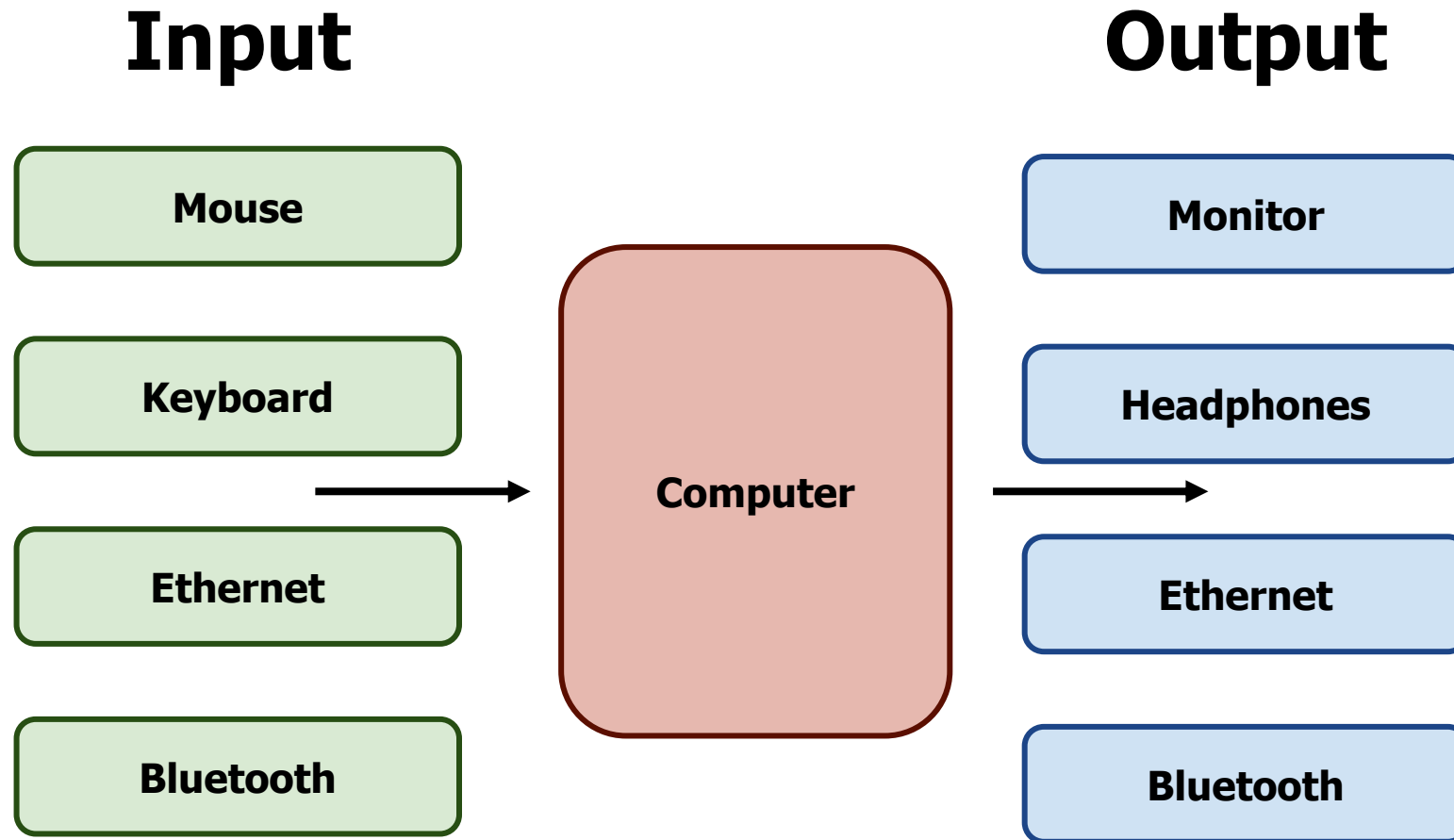
- **Overview of Device I/O**
- Connecting to devices
  - Buses on a computer
- Talking to devices
  - Port-Mapped I/O and Memory-Mapped I/O
- Device interactions
  - Synchronous versus Asynchronous Events
  - Programmed I/O versus Direct Memory Access

# Devices are the point of modern computers

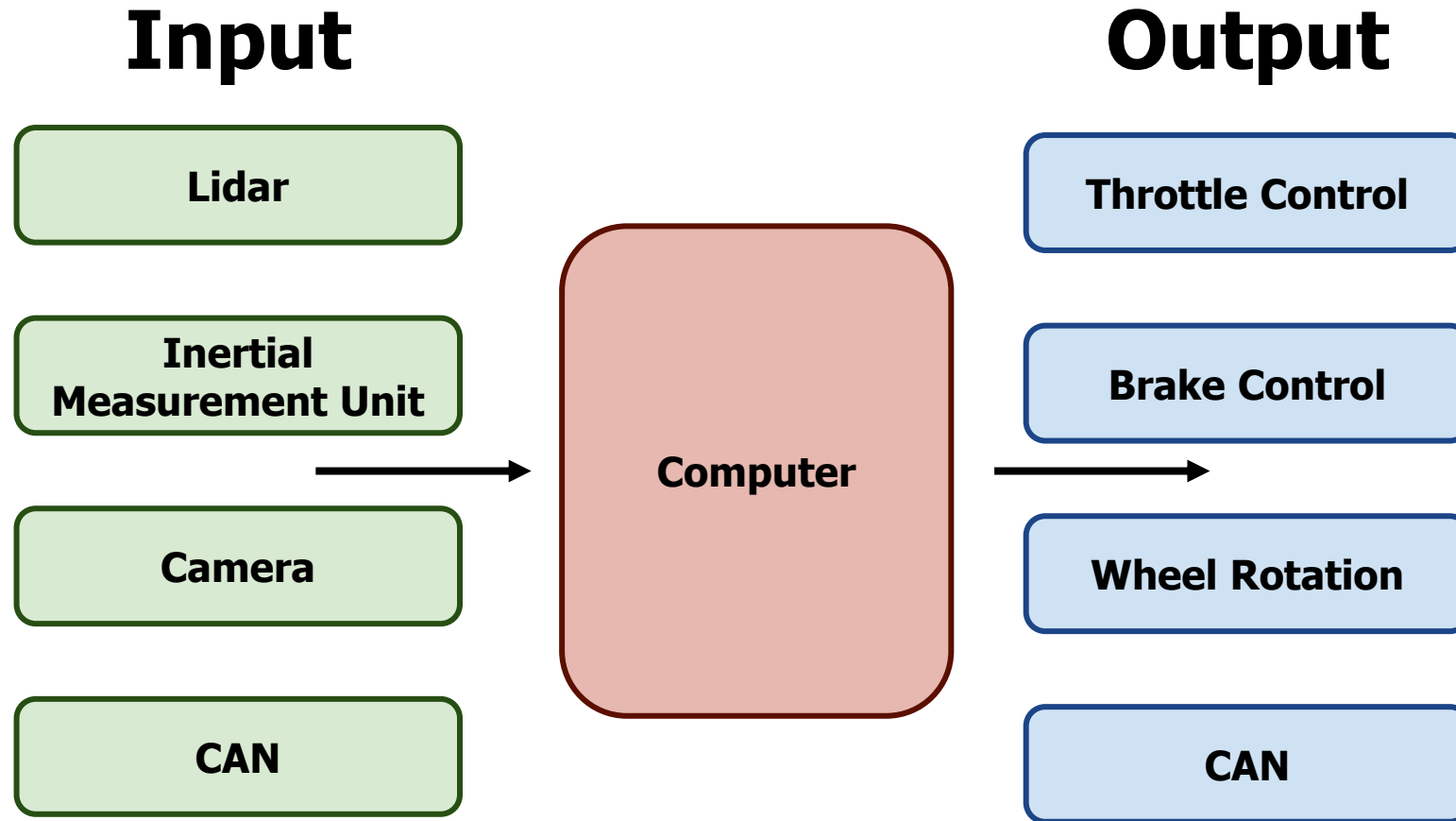
- Computation was sufficient for batch systems
  - Even then, tape or disk was the input and output mechanism
- But interactive systems need to receive input from users and output responses
  - Keyboard/mouse
  - Disk
  - Network
  - Graphics
  - Audio
  - Various USB devices



# Devices are core to useful general-purpose computing



# Devices are essential to cyber-physical systems too



# Device access rates vary by many orders of magnitude

- Rates in bit/sec

- System must be able to handle each of these

- Sometimes needs low overhead
- Sometimes needs to not wait around

Device	Behavior	Partner	Data Rate (Kb/s)
Keyboard	Input	Human	0.2
Mouse	Input	Human	0.4
Microphone	Output	Human	700.0
Bluetooth	Input or Output	Machine	20,000.0
Hard disk drive	Storage	Machine	100,000.0
Wireless network	Input or Output	Machine	300,000.0
Solid state drive	Storage	Machine	500,000.0
Wired LAN network	Input or Output	Machine	1,000,000.0
Graphics display	Output	Human	3,000,000.0



# Handling devices appropriately

- OS concerns
  - Communicating with devices needs to be fast and efficient
  - Devices are shared resources that need to be access controlled and shared
  - Devices are wildly variable and need some common interfaces for application software
- General theme
  - Access I/O devices similarly to memory
  - Read device state and write commands to it
    - With interrupts to inform kernel of events

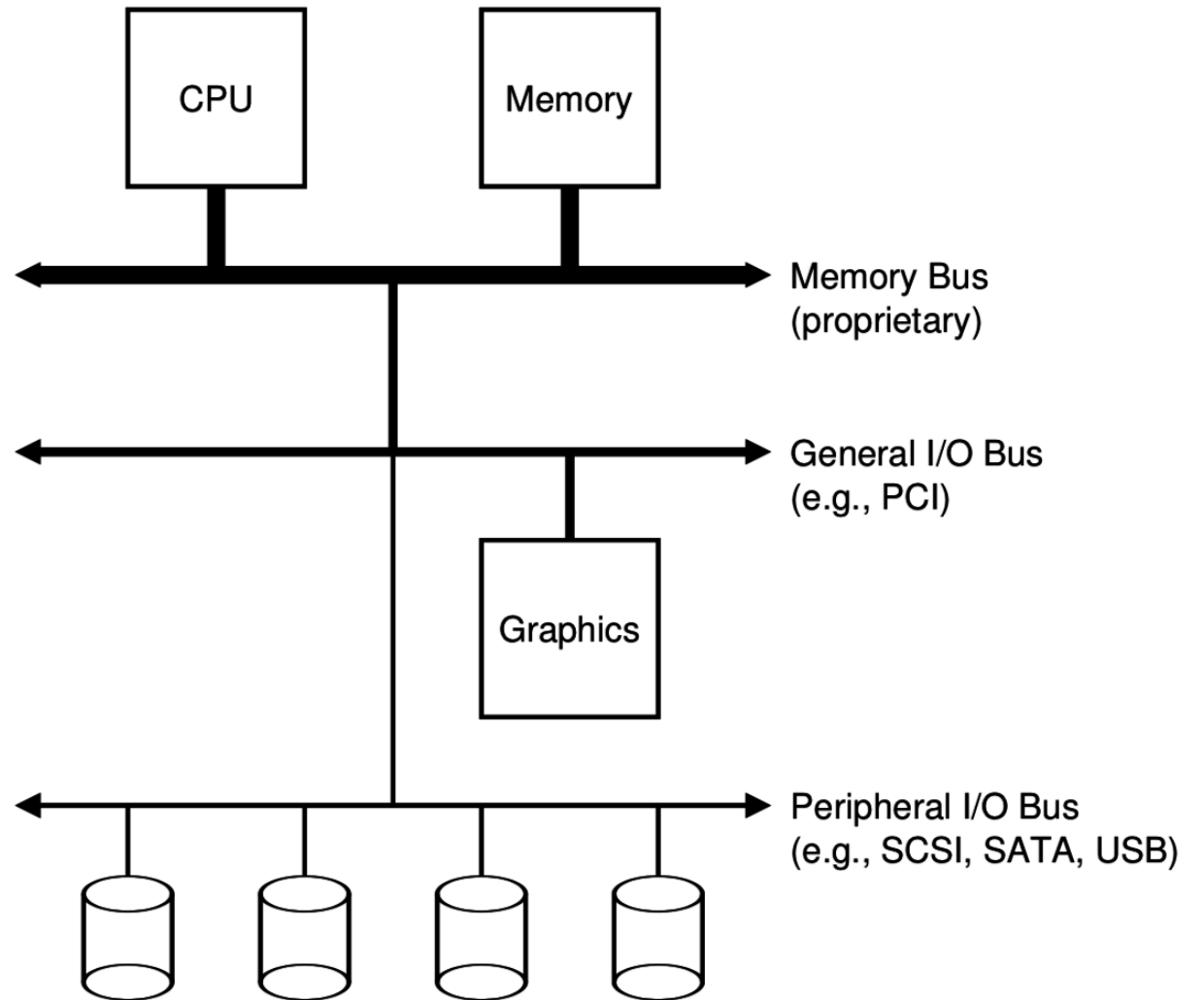
# Outline

- Overview of Device I/O
- **Connecting to devices**
  - **Buses on a computer**
- Talking to devices
  - Port-Mapped I/O and Memory-Mapped I/O
- Device interactions
  - Synchronous versus Asynchronous Events
  - Programmed I/O versus Direct Memory Access

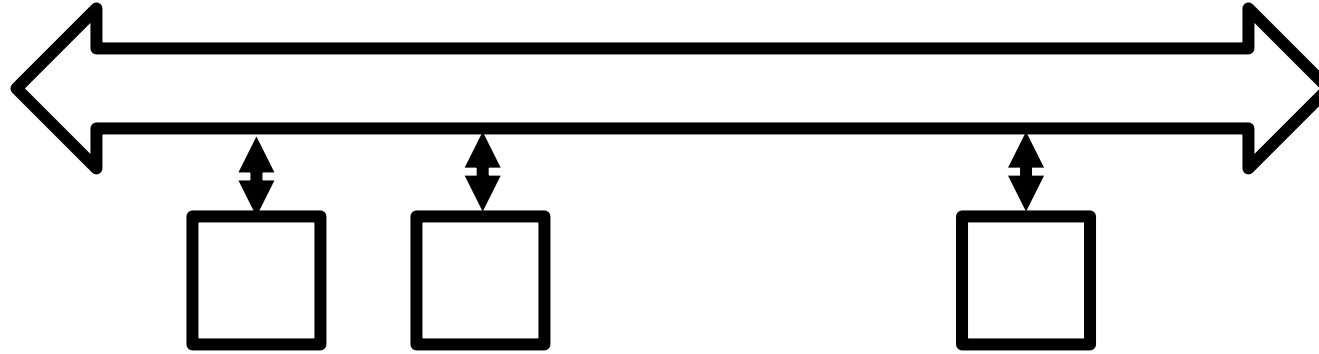
# Devices connect to buses on the computer

- I/O Hierarchy

- Close to the CPU are very fast connections
- Farther from CPU are slower but more flexible protocols



# What is a bus anyways?



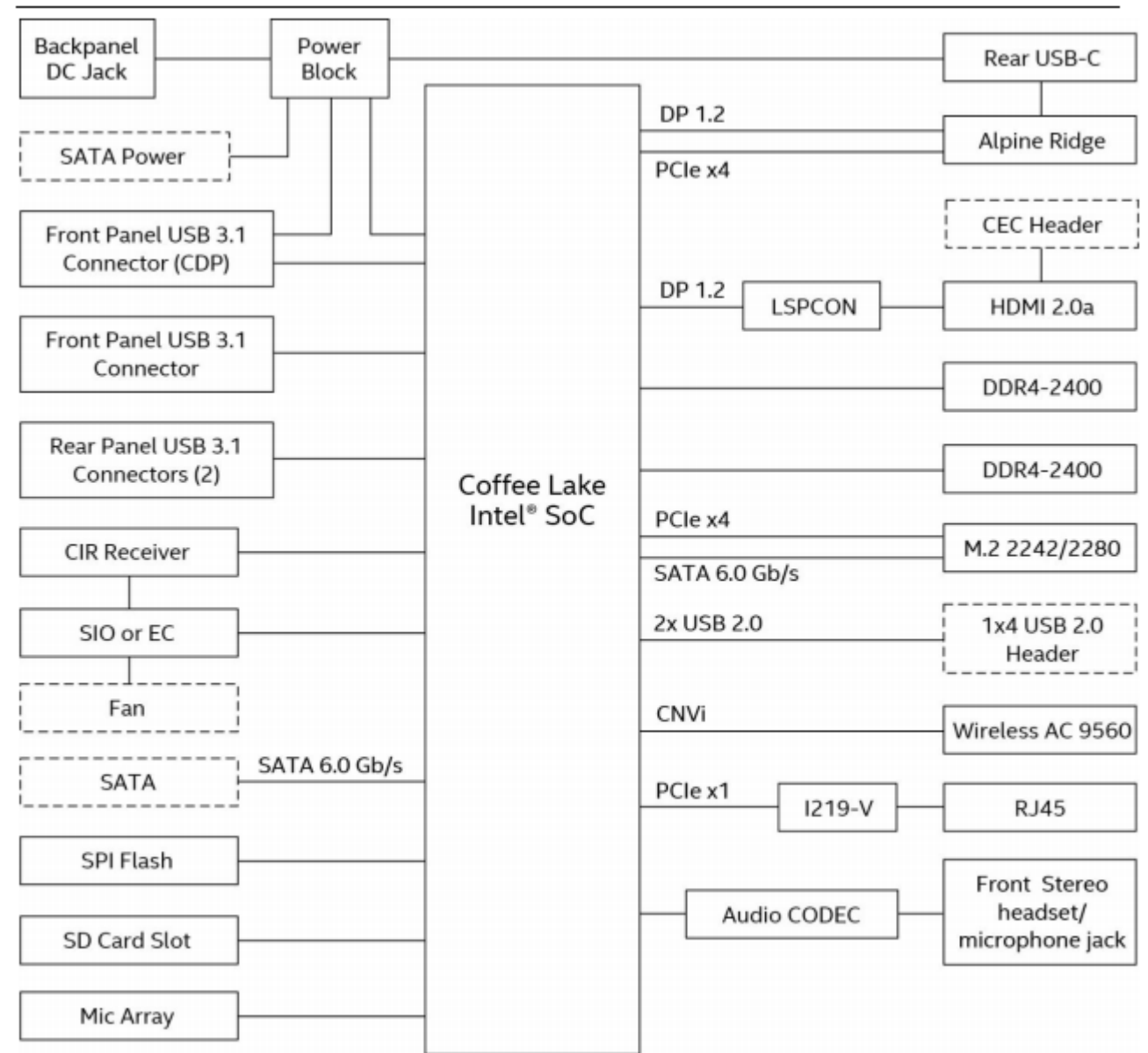
- Common set of wires for communication between two or more components
  - Lets one set of wires connect to N devices
- Standardized buses have a specific set of wires and protocol for communicating over them (DMI, PCI, SATA, USB)
  - Example wires: 64 address wires, 64 data wires, ~10 control wires
- Concerns
  - How many wires in the bus?
  - Single controller or multiple with arbitration?
  - Half-duplex (one direction of communication at a time) or full-duplex?

# Computer networks also run over buses

- Ethernet is a bus too!
- Network protocols specify how two computers communicate, very similarly to these buses
- Compared to networks, internal computer busses are:
  - Higher speed
  - Very high reliability
  - Accessed cooperatively (often with only one controller: the CPU)

# My home “desktop”

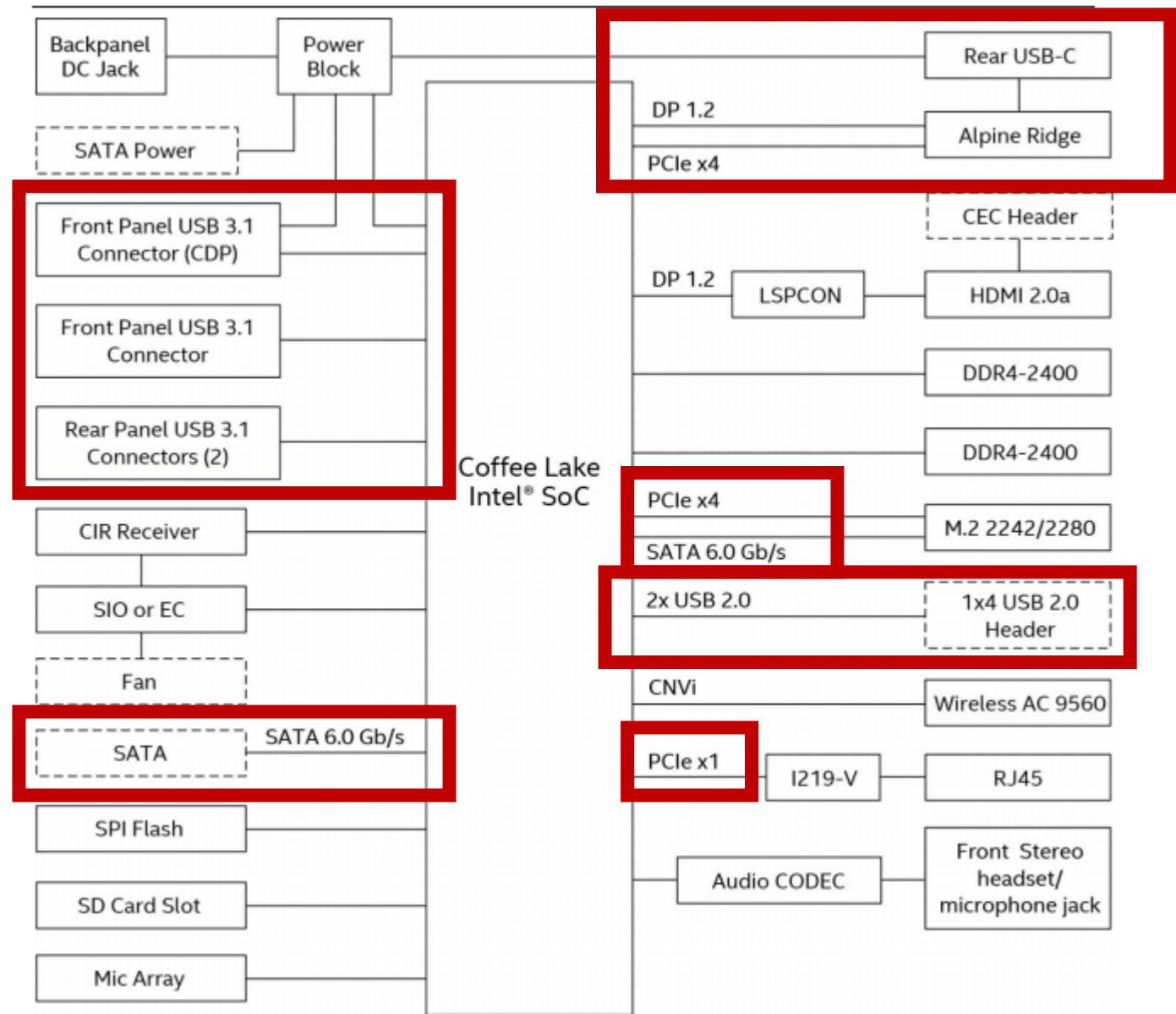
- Small form factor desktop
  - Intel NUC 8
- CPU includes I/O Controller and GPU in single package
- CPU implements many connections directly
  - USB
  - SATA
  - Some have hardware controllers
    - Ethernet (RJ45)
    - Thunderbolt (Rear USB-C)
    - 3.5mm Audio (Front stereo headset...)



24390

# Some important buses

- Legacy
  - Parallel Port
  - Serial Port
- USB
- SATA
- PCIe
- Device driver sometimes talks to bus controller
  - Which sends appropriate signals to device



24390

# Parallel Port – “Printer Port” or Centronics Port

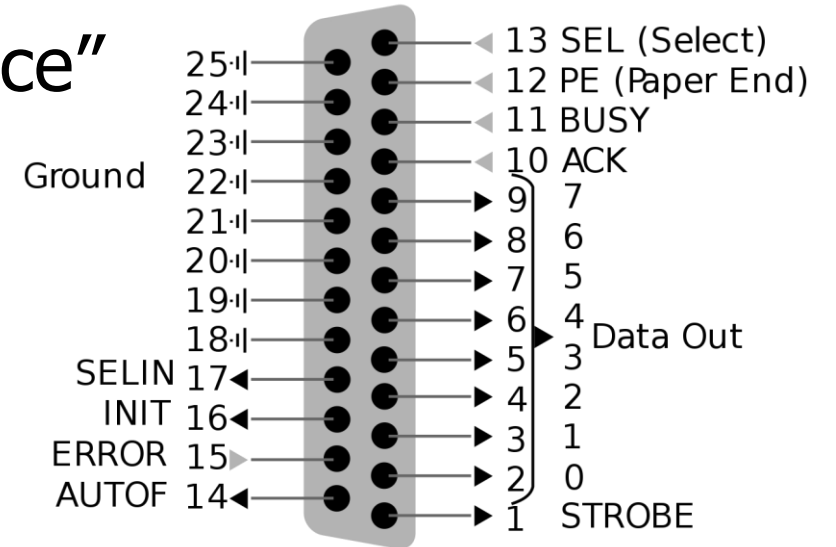
- “That big long one you might have seen once”

- 8 data bits plus control signals

- Up to 2.5 Mbps!!

- Very simple to implement

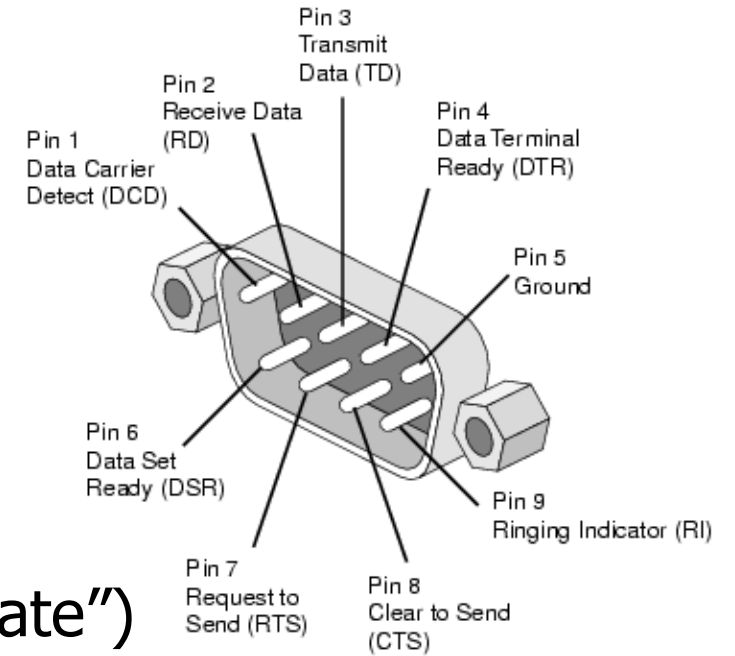
- Write 8 bits of data
  - Set STROBE low
    - BUSY goes low in response
  - When BUSY goes high
    - Set STROBE high
  - Repeat





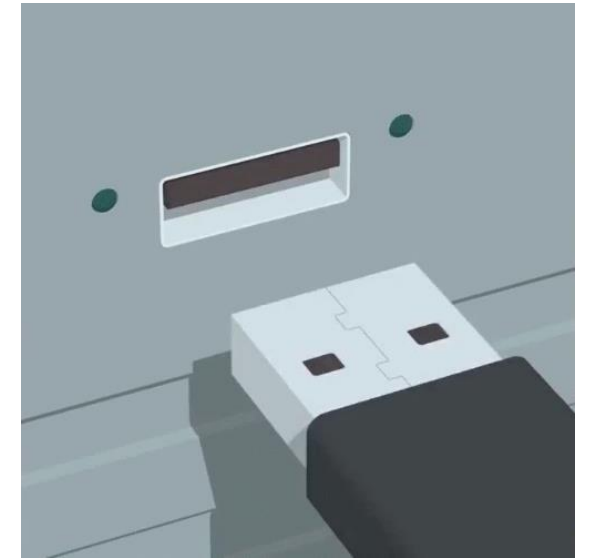
# Serial Port – RS-232

- “That one we used to use before USB”
- RS-232 serial protocol
  - One wire for Transmit, one wire for receive
  - Bits go one after another at a certain speed (“baudrate”)
    - Up to 256 Kbps
- Used to connect modems
  - Still occasionally used for old devices (via a USB to RS-232 converter)



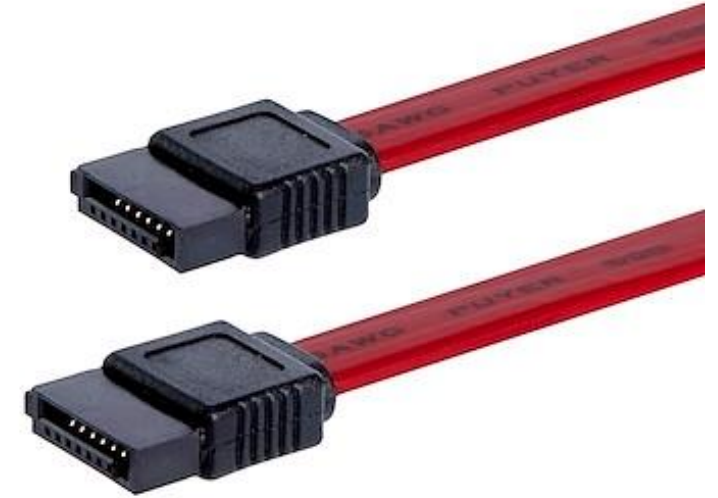
# USB – Universal Serial Bus

- “That one you are familiar with already”
- Connect and power external devices
- Comes in multiple form-factors and versions
  - USB 3.x and type C connectors are the fast ones
- Speed has increased greatly
  - 1.5 Mbps
  - 12 Mbps
  - 480 Mbps
  - 5 Gbps
  - 10 Gbps
  - 40 Gbps



# SATA – Serial ATA – AT Attachment – “Advanced Technology”

- “That cable you plug an SSD/HDD in with”
- Long evolution as you might have guessed from the name
- 6 Gbps
  - One transmit and one receive single
    - Two wires for each in twisted pair
  - Short connection length for high speed

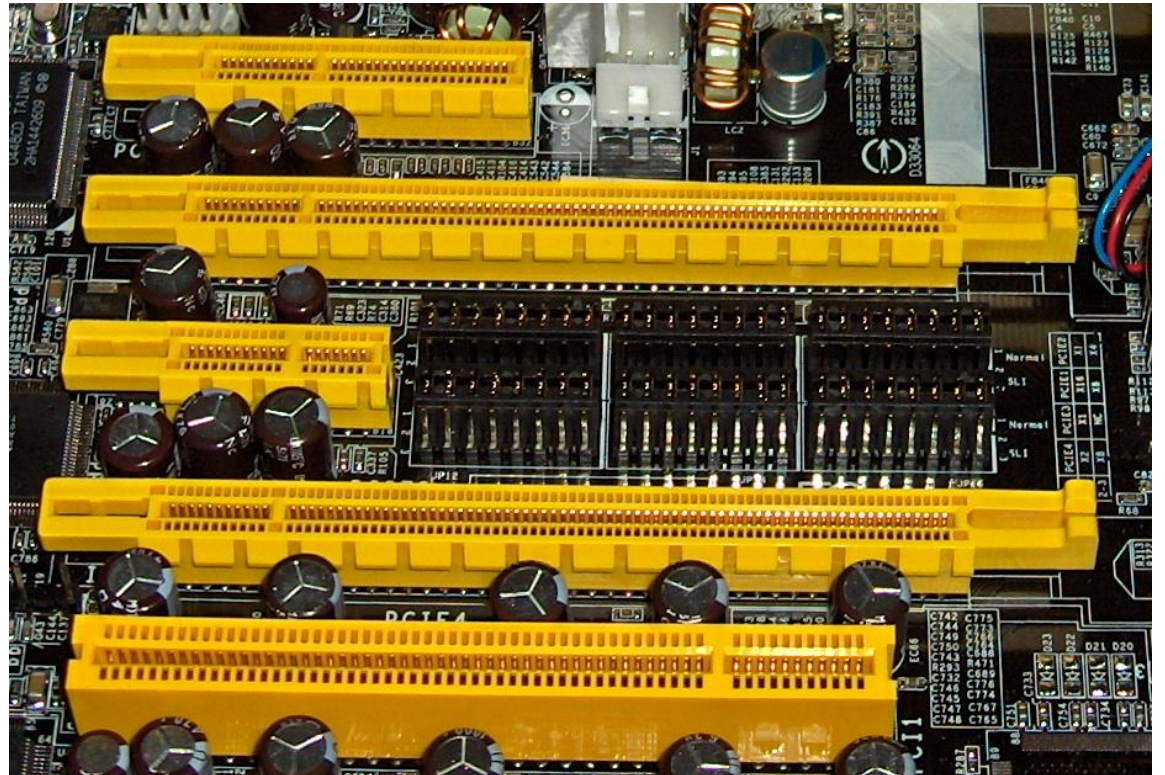


SATA

Power

# PCIe – Peripheral Component Interconnect Express

- “That slot you put a graphics card into”
  - Or WiFi card, or some SSDs
- Collection of point-to-point connections
  - Motherboard and CPU support N “lanes” in various configurations
- Different sizes for different number of bits in parallel
  - Order 10s Gbps



# Live Demo: List devices on a Linux computer

- `lsusb`
  - List USB devices
- `lspci`
  - List PCI devices
- Combine with `-v` flag for verbose mode with more information
- Combine with `-s` flag to select a single device



# Break + Question

- Why do we need all of these busses?
  - Isn't USB enough for everything?

<https://www.reddit.com/r/explainlikeimfive/comments/uf4efj/comment/i6rmv15>

# Break + Question

- Why do we need all of these busses?
  - Isn't USB enough for everything?
- Different tools for different purposes!
  - USB is more general-purpose, short range, powers devices
  - PCIe is for LOTS of data, but very short range and cables would be crazy
  - Ethernet is for long-range, lots of data, no power

<https://www.reddit.com/r/explainlikeimfive/comments/uf4efj/comment/i6rmv15>

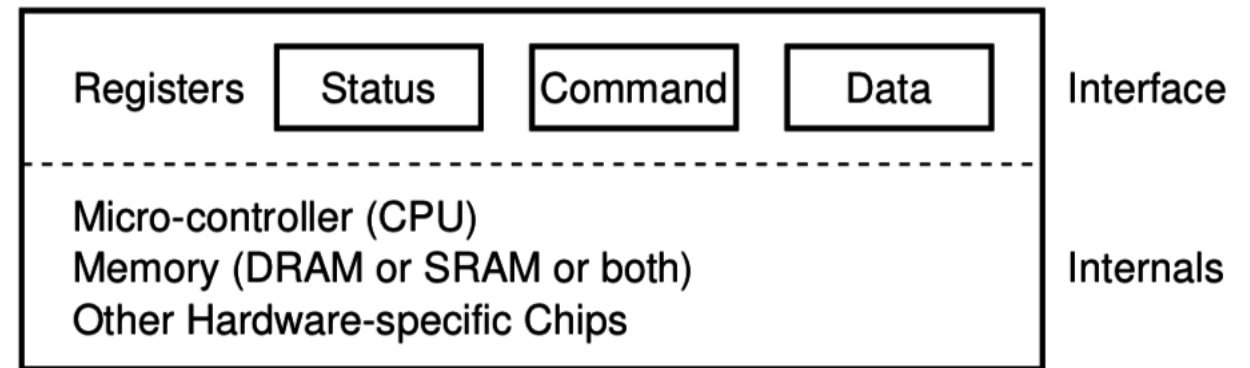
# Outline

- Overview of Device I/O
- Connecting to devices
  - Buses on a computer
- **Talking to devices**
  - **Port-Mapped I/O and Memory-Mapped I/O**
- Device interactions
  - Synchronous versus Asynchronous Events
  - Programmed I/O versus Direct Memory Access



# How does an OS talk with I/O devices?

- A device is really a miniature computer-within-the-computer
  - Has its own processing, memory, software
- We can mostly ignore that and deal with its interface
  - Called registers (actually are from EE perspective, but you can't use them)
  - Read/Write like they're data
- How do we read/write them?
  - Special assembly instructions
  - Treat like normal memory



# Example powered device: Real Time Clock

- Battery-backed up clock on computer motherboard
- Keeps sense of time when computer is off
- Resynchronized when the computer is awake

Index	Contents	Range
0x00	Seconds	0-59
0x02	Minutes	0-59
0x04	Hours	0-23 in 24-hour mode, 1-12 in 12-hour mode, highest bit set if PM
0x06	Weekday	1-7, Sunday =1
0x07	Day of Month	1-31
0x08	Month	1-12
0x09	Year	0-99

<b>Index Register</b>	<b>Data Register</b>
-----------------------	----------------------

# Port-Mapped I/O (PMIO): special assembly instructions

- x86 IN and OUT instructions
  - Privileged instructions (kernel mode only)
  - Two arguments: destination and data register
- Each device is mapped to some port address
  - IN and OUT instructions interact with interface
  - IN <PORT NUMBER>, <REGISTER>
  - OUT <REGISTER>, <PORT NUMBER>

# Example powered device: Real Time Clock

- Example: read current value from real-time clock

```
// read seconds  
mov $0, %al  
out %al, $0x70  
in $0x71, %al
```

Index	Contents	Range
0x00	Seconds	0-59
0x02	Minutes	0-59
0x04	Hours	0-23 in 24-hour mode, 1-12 in 12-hour mode, highest bit set if PM
0x06	Weekday	1-7, Sunday = 1
0x07	Day of Month	1-31
0x08	Month	1-12
0x09	Year	0-99

Index Register	Data Register
0x70 <sup>1</sup>	0x71

← Port Address

## Example I/O port map

This isn't standardized, but these are some typical values.

[https://wiki.osdev.org/Can\\_I\\_have\\_a\\_list\\_of\\_IO\\_Ports](https://wiki.osdev.org/Can_I_have_a_list_of_IO_Ports)

Port range	Summary
0x0000-0x001F	The first legacy <b>DMA controller</b> , often used for transfers to floppies.
0x0020-0x0021	The first <b>Programmable Interrupt Controller</b>
0x0022-0x0023	Access to the Model-Specific Registers of Cyrix processors.
0x0040-0x0047	The <b>PIT</b> (Programmable Interval Timer)
0x0060-0x0064	The " <b>8042</b> " <b>PS/2 Controller</b> or its predecessors, dealing with keyboards and mice.
0x0070-0x0071	The <b>CMOS</b> and <b>RTC</b> registers
0x0080-0x008F	The <b>DMA</b> (Page registers)
0x0092	The location of the fast <b>A20</b> gate register
0x00A0-0x00A1	The second <b>PIC</b>
0x00C0-0x00DF	The second <b>DMA</b> controller, often used for soundblasters
0x00E9	Home of the <b>Port E9 Hack</b> . Used on some emulators to directly send text to the hosts' console.
0x0170-0x0177	The secondary <b>ATA</b> harddisk controller.
0x01F0-0x01F7	The primary <b>ATA</b> harddisk controller.
0x0278-0x027A	Parallel port
0x02F8-0x02FF	Second <b>serial port</b>
0x03B0-0x03DF	The range used for the <b>IBM VGA</b> , its direct predecessors, as well as any modern video card in legacy mode.
0x03F0-0x03F7	<b>Floppy disk controller</b>
0x03F8-0x03FF	First <b>serial port</b>

# Check your understanding – PMIO in C

- How would you access PMIO from a C program?

# Check your understanding – PMIO in C

- How would you access PMIO from a C program?
  - Need to use assembly!
  - Hopefully with C function wrapper, like System Calls

# Annoying parts of Port-Mapped I/O

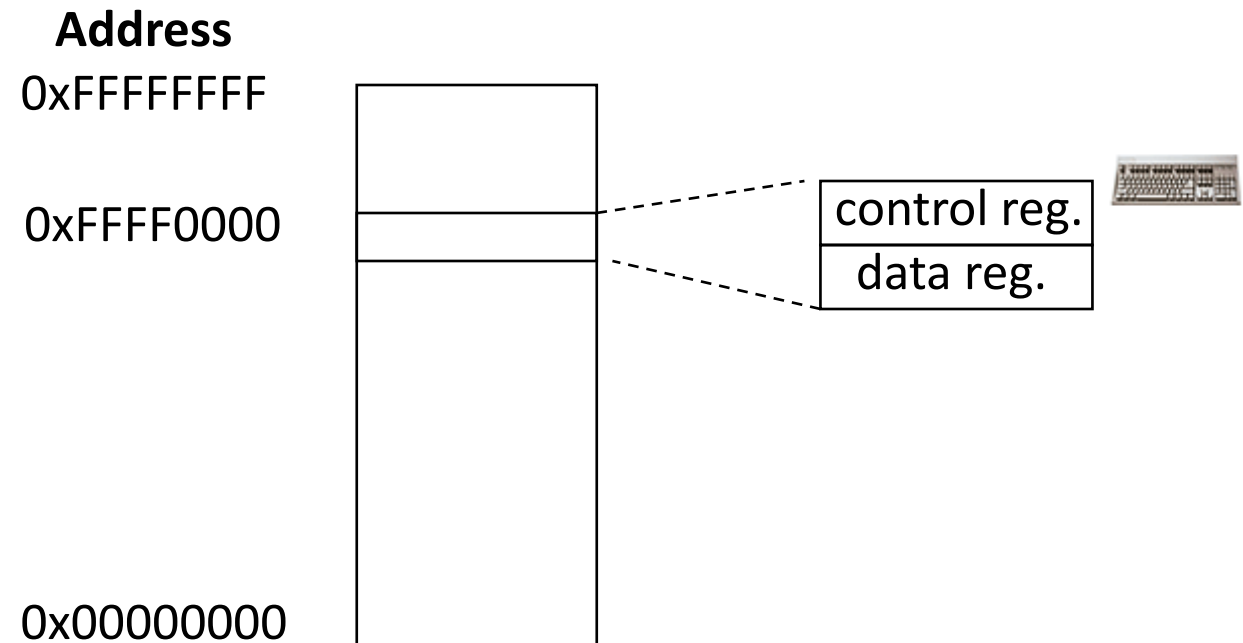
- Special assembly instructions are hard to write in C
  - Need some wrapper function that actually calls them
  - Not really that big of an issue, but a little weird
- Feels sort of like memory read/write, but isn't
  - Why not?
  - Can we just put the "port address space" somewhere in memory?
    - Could be a problem if we don't have enough memory
    - But today we have tons of extra physical address space laying around



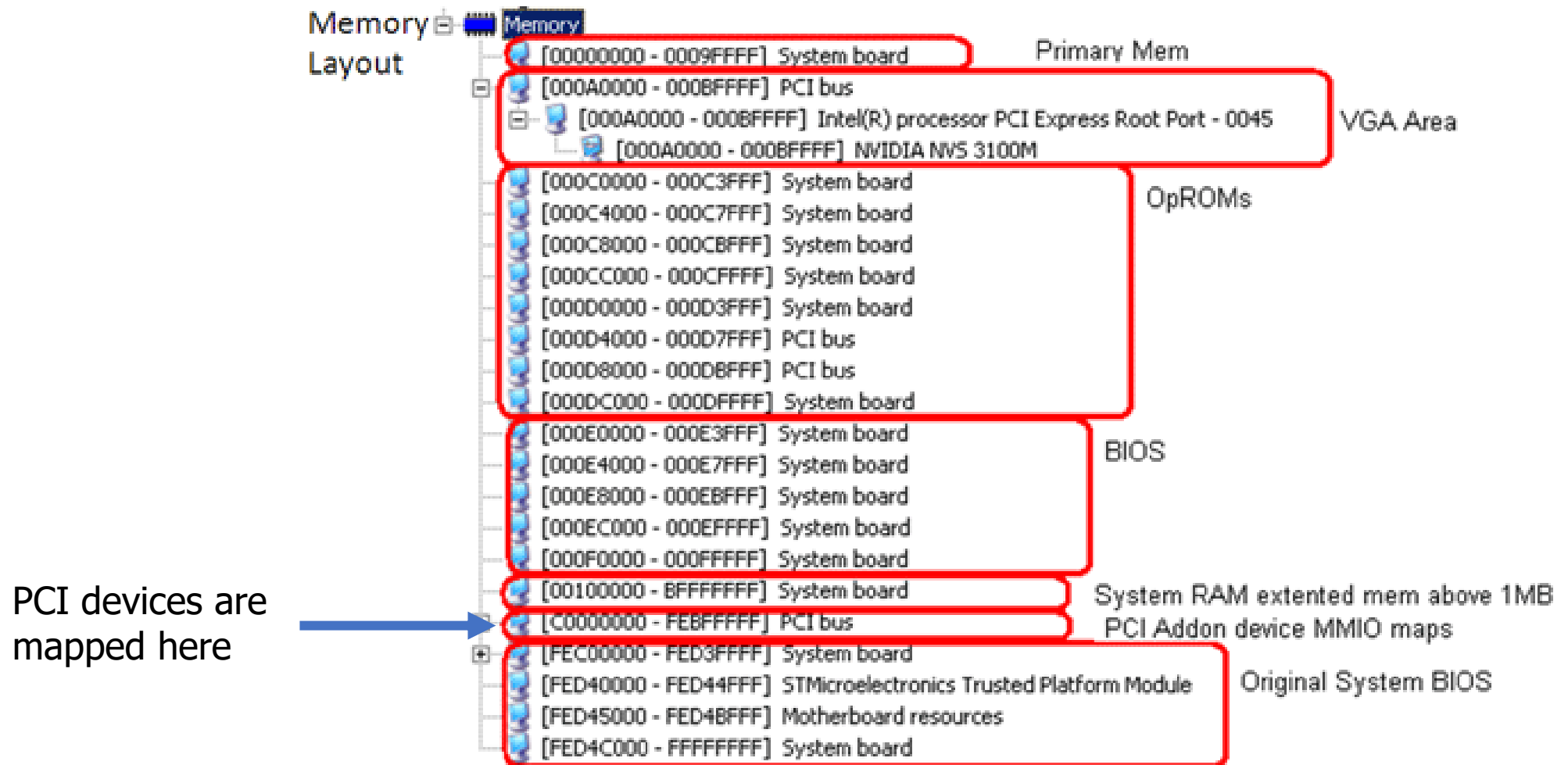
# Memory-mapped I/O (MMIO): treat devices like normal memory

- Certain physical addresses do not actually go to RAM
- Instead, they correspond to I/O devices
  - And any instruction that accesses memory can access them too!

- x86 being the historical amalgamation that it is, uses both PMIO or MMIO depending on the device



# Example memory map (from an old 32-bit computer)





# Example devices on my windows computer

- SATA controller is mapped into memory at two places
- USB controller is mapped into a much higher memory region
- Regions are large because they contain multiple control/data "registers"



Standard SATA AHCI Controller


Resource settings:

Resource type	Setting
 Memory Range	00000000C0A24000 - 00000000C0A25FFF
 Memory Range	00000000C0A27000 - 00000000C0A270FF



Intel(R) USB 3.1 eXtensible Host Controller - 1.10 (Microsoft)

Resource settings:

Resource type	Setting
 Memory Range	000000404AC00000 - 000000404AC0FFFF

# Microcontroller example: reading temperature

- Internal temperature sensor
  - 0.25° C resolution
  - Range equivalent to microcontroller IC (-40° to 105° C)
  - Various configurations for the temperature conversion (ignoring)

Base address	Peripheral	Instance	Description	Configuration
0x4000C000	TEMP	TEMP	Temperature sensor	

*Table 110: Instances*

Register	Offset	Description
TASKS_START	0x000	Start temperature measurement
TASKS_STOP	0x004	Stop temperature measurement
EVENTS_DATARDY	0x100	Temperature measurement complete, data ready
INTENSET	0x304	Enable interrupt
INTENCLR	0x308	Disable interrupt
TEMP	0x508	Temperature in °C (0.25° steps)

nRF52833

# MMIO addresses for TEMP device

- What addresses do we need? (ignore interrupts for now)
  - 0x4000C000 – TASKS\_START
  - 0x4000C100 – EVENTS\_DATARDY
  - 0x4000C508 - TEMP

Base address	Peripheral	Instance	Description	Configuration
0x4000C000	TEMP	TEMP	Temperature sensor	

*Table 110: Instances*

Register	Offset	Description
TASKS_START	0x000	Start temperature measurement
TASKS_STOP	0x004	Stop temperature measurement
EVENTS_DATARDY	0x100	Temperature measurement complete, data ready
INTENSET	0x304	Enable interrupt
INTENCLR	0x308	Disable interrupt
TEMP	0x508	Temperature in °C (0.25° steps)

nRF52833

# Accessing addresses in C

- What does this C code do?

```
*(uint32_t*)(0x4000C000) = 1;
```

# Accessing addresses in C

- What does this C code do?

```
*(uint32_t*)(0x4000C000) = 1;
```

- 0x4000C000 is cast to a uint32\_t\*
- Then dereferenced
- And we write 1 to it
- "There are 32-bits of memory at 0x4000C000. Write a 1 there."

# Other details about MMIO

- Devices are mapped into physical memory
  - Usually only accessible by the kernel
  - But could be directly placed in virtual memory for a process in very special cases
- Devices are NOT memory though
  - Need to be careful not to cache them
    - Values being read could change, or reading could have an effect
  - Cannot let compiler mess with our reads/writes either
    - *volatile* keyword in C
- Conceptually not really very different from PMIO
  - Both just read/write to specific addresses the device is mapped to



# Break + example code to read and print temperature value

```
// loop forever
while (1) {

    // start a measurement
    *(uint32_t*)(0x4000C000) = 1;

    // wait until ready
    volatile uint32_t ready = *(uint32_t*)(0x4000C100);
    while (!ready) {
        ready = *(uint32_t*)(0x4000C100);
    }

    /* WARNING: we can't write the code this way!
     * Without `volatile`, the compiler optimizes out the memory access
     while (!*(uint32_t*)(0x4000C100));
     */

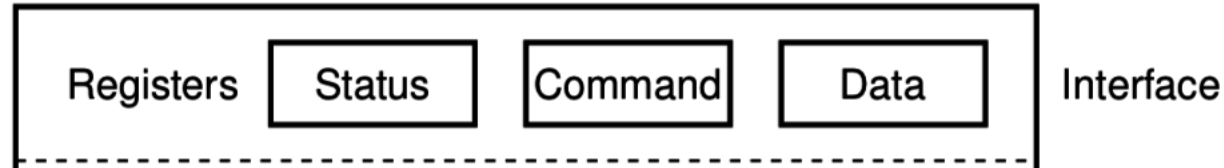
    // read data and print it
    volatile int32_t value = *(int32_t*)(0x4000C508);
    float temperature = ((float)value)/4.0;
    printf("Temperature=%f degrees C\n", temperature);

    nrf_delay_ms(1000);
}
```

# Outline

- Overview of Device I/O
- Connecting to devices
  - Buses on a computer
- Talking to devices
  - Port-Mapped I/O and Memory-Mapped I/O
- **Device interactions**
  - **Synchronous versus Asynchronous Events**
  - Programmed I/O versus Direct Memory Access

# What do interactions with devices look like?



1. `while STATUS==BUSY; Wait`
  - (Need to make sure device is ready for a command)
2. Write value(s) to DATA
3. Write command(s) to COMMAND
4. `while STATUS==BUSY; Wait`
  - (Need to make sure device has completed the request)
5. Read value(s) from Data

This is the “polling” model of I/O.

“Poll” the peripheral in software repeatedly to see if it’s ready yet.

# Waiting can be a waste of CPU time

## **1. while STATUS==BUSY; Wait**

- **(Need to make sure device is ready for a command)**

2. Write value(s) to DATA

3. Write command(s) to COMMAND

## **4. while STATUS==BUSY; Wait**

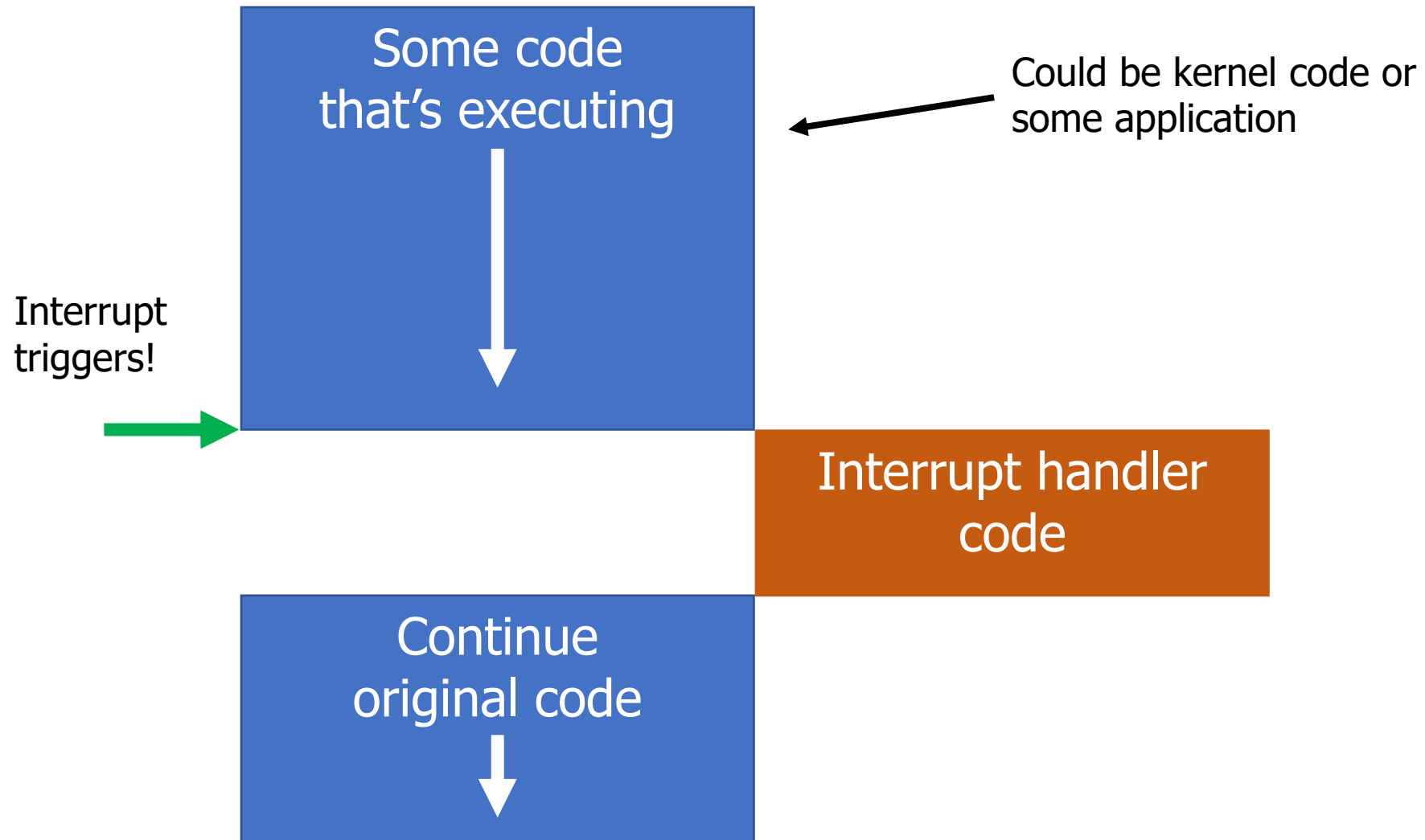
- **(Need to make sure device has completed the request)**

5. Read value(s) from Data

## • Imagine a keyboard device

- CPU could be waiting for minutes before data arrives
- Need a way to notify CPU when an event occurs
  - Interrupts!

# Interrupts, visually



# Hardware devices can generate interrupts

- Each device maps to some number of hardware interrupts
- Done at system boot time
  - Discover devices
  - Map devices into address space
  - Map interrupts for devices

Table 6-1. Exceptions and Interrupts

Vector No.	Mnemonic	Description	Source
0	#DE	Divide Error	DIV and IDIV instructions.
1	#DB	Debug	Any code or data reference.
2		NMI Interrupt	Non-maskable external interrupt.
3	#BP	Breakpoint	INT 3 instruction.
4	#OF	Overflow	INTO instruction.
5	#BR	BOUND Range Exceeded	BOUND instruction.
6	#UD	Invalid Opcode (UnDefined Opcode)	UD2 instruction or reserved opcode. <sup>1</sup>
7	#NM	Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.
9	#MF	CoProcessor Segment Overrun (reserved)	Floating-point instruction. <sup>2</sup>
10	#TS	Invalid TSS	Task switch or TSS access.
11	#NP	Segment Not Present	Loading segment registers or accessing system segments.
12	#SS	Stack Segment Fault	Stack operations and SS register loads.
13	#GP	General Protection	Any memory reference and other protection checks.
14	#PF	Page Fault	Any memory reference.
15		Reserved	
16	#MF	Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Any data reference in memory. <sup>3</sup>
18	#MC	Machine Check	Error codes (if any) and source are model dependent. <sup>4</sup>
19	#XM	SIMD Floating-Point Exception	SIMD Floating-Point Instruction <sup>5</sup>
20-31		Reserved	
32-255		Maskable Interrupts	External interrupt from INTR pin or INT <i>n</i> instruction.

# Interrupts allow waiting to happen asynchronously

- Prior code example was *synchronous*
  - Nothing else continued on the processor until access was complete
  - Good for very fast devices (like the real-time clock, that just returns data)
  - We call this “Polling”
- With interrupts, device handling is now *asynchronous*
  - Access occurs in the background and processor can do something else
  - Good for very slow devices (Disk)
  - Comes with all the downsides of concurrency though...

# Microcontroller TEMP device supports interrupts!

- Can either wait on the EVENTS\_DATARDY register
- Or could enable an interrupt from the device
  - And only both reading data when it is ready

Base address	Peripheral	Instance	Description	Configuration
0x4000C000	TEMP	TEMP	Temperature sensor	

*Table 110: Instances*

Register	Offset	Description
TASKS_START	0x000	Start temperature measurement
TASKS_STOP	0x004	Stop temperature measurement
EVENTS_DATARDY	0x100	Temperature measurement complete, data ready
INTENSET	0x304	Enable interrupt
INTENCLR	0x308	Disable interrupt
TEMP	0x508	Temperature in °C (0.25° steps)



# Outline

- Overview of Device I/O
- Connecting to devices
  - Buses on a computer
- Talking to devices
  - Port-Mapped I/O and Memory-Mapped I/O
- **Device interactions**
  - Synchronous versus Asynchronous Events
  - **Programmed I/O versus Direct Memory Access**

# Programmed I/O (PIO)

1. while STATUS==BUSY; Wait (possibly on interrupt)
  - (Need to make sure device is ready for a command)

- 2. Write value(s) to DATA**

3. Write command(s) to COMMAND

4. while STATUS==BUSY; Wait (possibly on interrupt)
  - (Need to make sure device has completed the request)

- 5. Read value(s) from Data**

- How do we read and write those values? (could be a lot)
  - With normal CPU memory accesses: Programmed I/O
  - Literally: you write a program to do the input and output

# Check your understanding – writing to GPU

- Let's say that a GPU has MMIO registers for an entire 4 KB page
  - Takes 100 ns to write each word (8 bytes) of memory
- Assuming that we're just writing all zeros (ignore reading from memory), how long does it take to write a page to MMIO?

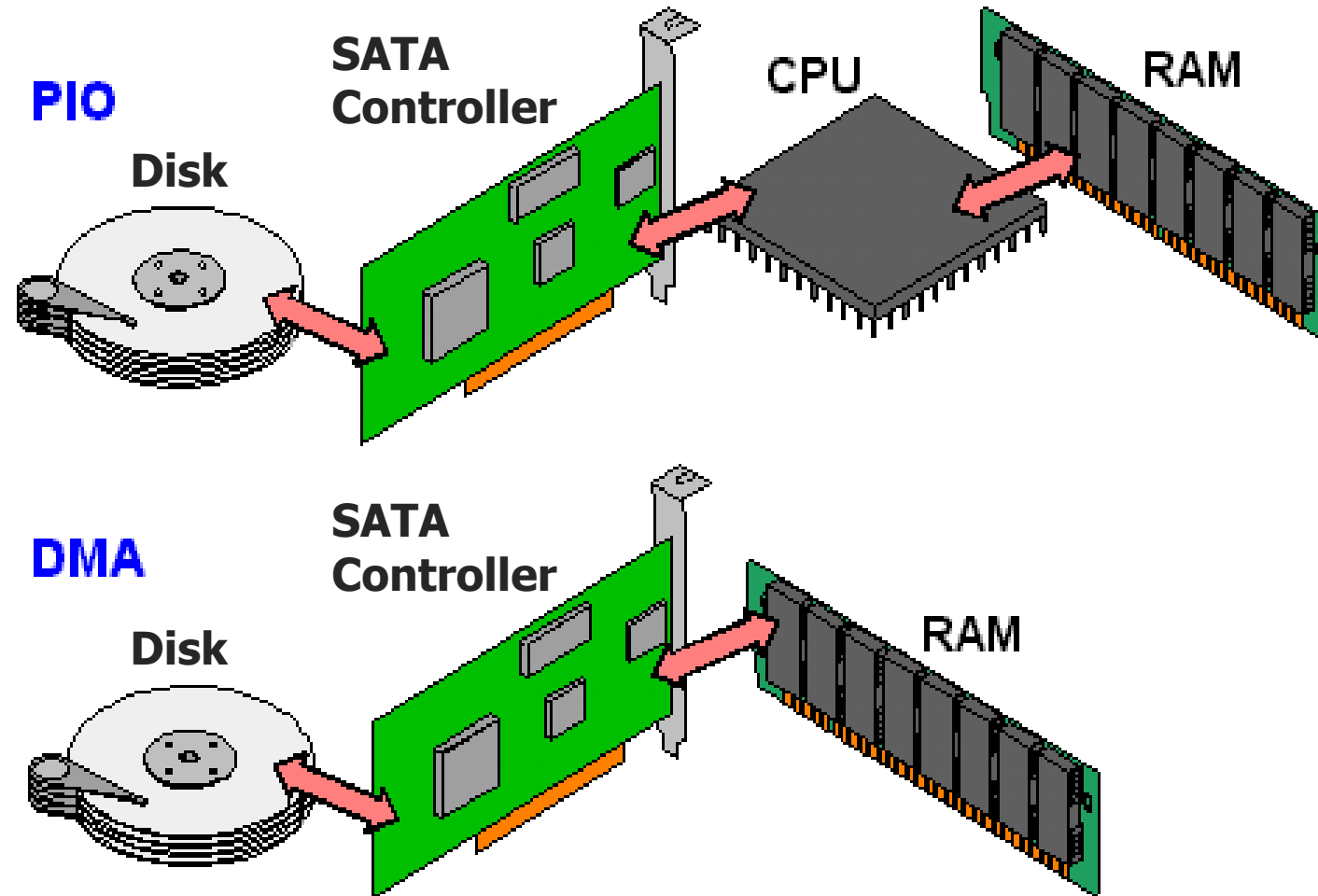
# Check your understanding – writing to GPU

- Let's say that a GPU has MMIO registers for an entire 4 KB page
  - Takes 100 ns to write each word (8 bytes) of memory
- Assuming that we're just writing all zeros (ignore reading from memory), how long does it take to write a page to MMIO?
  - $4 \text{ KB} / 8 \text{ B} = 500 \text{ writes} * 100 \text{ ns / write} = 50 \text{ } \mu\text{s}$ 
    - (For a 3 GHz processor, that's  $\sim 150,000$  cycles)

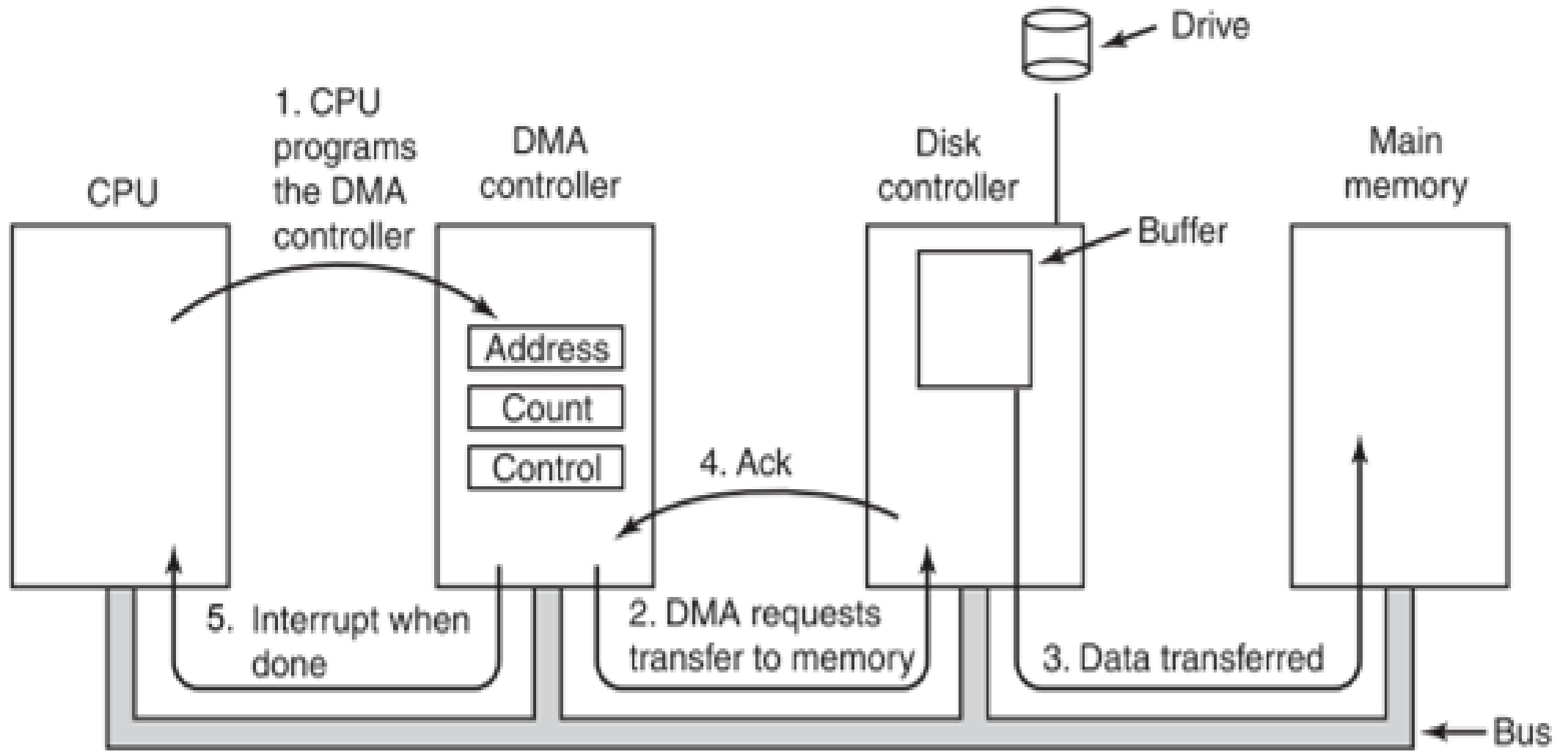
# Direct Memory Access (DMA)

- Even with interrupts, just providing data to the peripheral is time consuming
  - The processor is involved in writing/reading each byte
- DMA is an alternative method that uses hardware to do the memory transfers for the processor
  - Software writes address of the data and the size to the peripheral
  - Device reads data directly from memory
  - Processor can go do other things while read/write is occurring

# Programmed I/O versus Direct Memory Access



# Disk access with DMA



# DMA considerations

- Need to be careful about letting devices access arbitrary memory
  - Are devices part of the “trusted compute base”?
  - This random flash drive that’s plugged in shouldn’t be able to read all of RAM
- Often a hardware “DMA controller” does the transfer for the device
  - IOMMU can even set up virtual memory spaces for devices



# Interaction pattern with Interrupts and DMA

1. Configure the peripheral
2. Enable peripheral interrupts
3. Set up peripheral DMA transfer
4. Start peripheral

Continue on to other code

5. Interrupt occurs, signaling DMA transfer complete
6. Set up next DMA transfer

Continue on to other code, and repeat

- Kernel is in charge of keeping a queue of hardware requests

# Returning to the variety of devices

- Interrupts support high-latency devices and time-sensitive devices

- DMA supports high-throughput devices

Device	Behavior	Partner	Data Rate (Kb/s)
Keyboard	Input	Human	0.2
Mouse	Input	Human	0.4
Microphone	Output	Human	700.0
Bluetooth	Input or Output	Machine	20,000.0
Hard disk drive	Storage	Machine	100,000.0
Wireless network	Input or Output	Machine	300,000.0
Solid state drive	Storage	Machine	500,000.0
Wired LAN network	Input or Output	Machine	1,000,000.0
Graphics display	Output	Human	3,000,000.0

# Outline

- Overview of Device I/O
- Connecting to devices
  - Buses on a computer
- Talking to devices
  - Port-Mapped I/O and Memory-Mapped I/O
- Device interactions
  - Synchronous versus Asynchronous Events
  - Programmed I/O versus Direct Memory Access