

Lecture 14:

Device Input and Output

CS343 – Operating Systems
Branden Ghena – Fall 2020

Some slides borrowed from:

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

Today's Goals

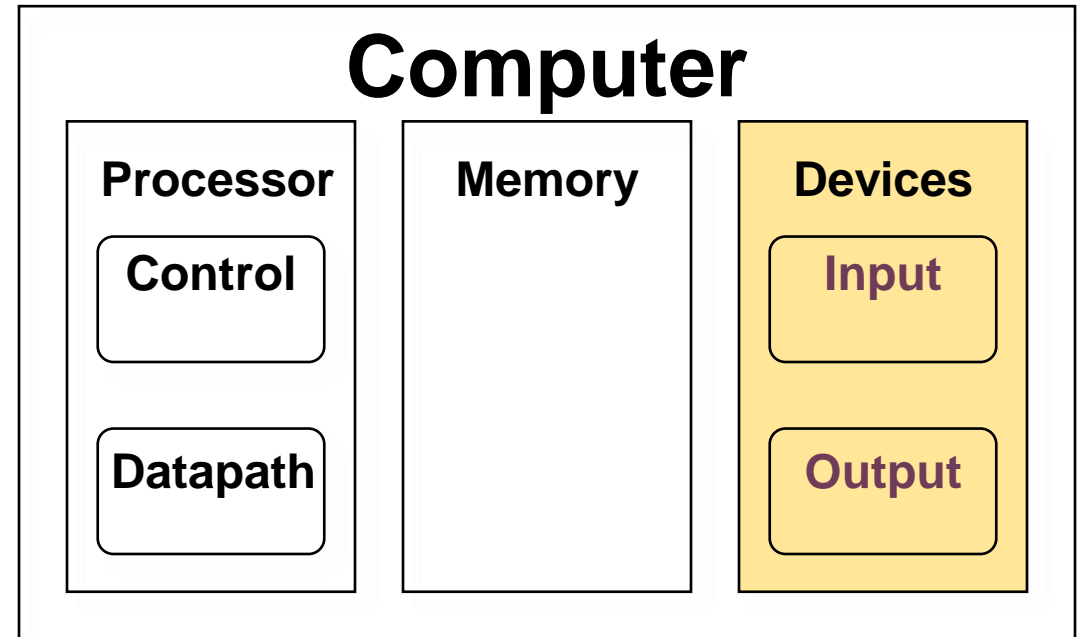
- Discuss I/O devices and how a computer connects to them.
- Understand two different methods of reading/writing device data.
- Explore patterns for device interaction:
 - Synchronous versus Asynchronous
 - Programmed I/O versus Direct Memory Access

Outline

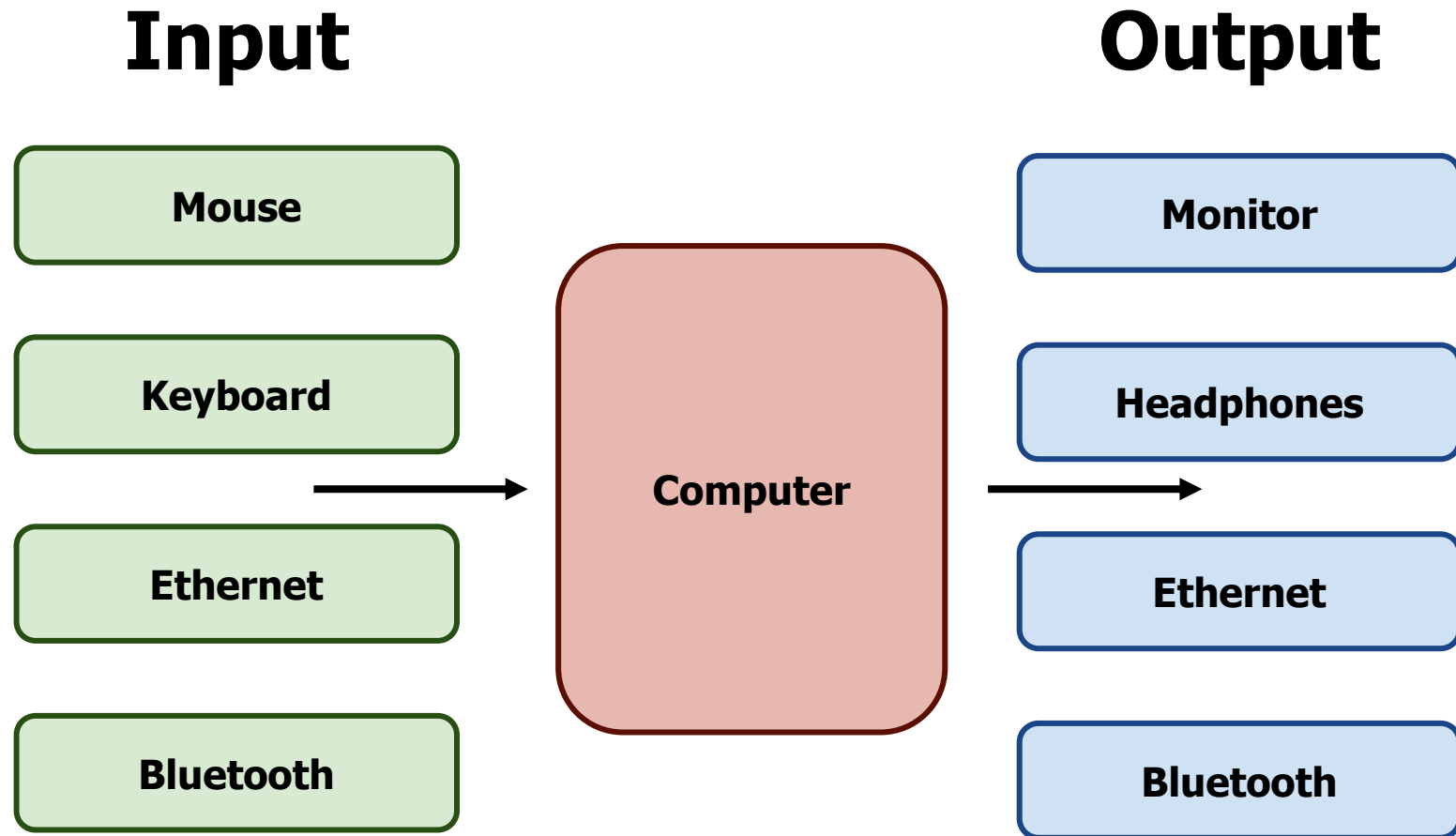
- **Overview of Device I/O**
- Connecting to devices
 - Buses on a computer
- Talking to devices
 - Port-Mapped I/O and Memory-Mapped I/O
- Device interactions
 - Synchronous versus Asynchronous Events
 - Programmed I/O versus Direct Memory Access

Devices are the point of modern computers

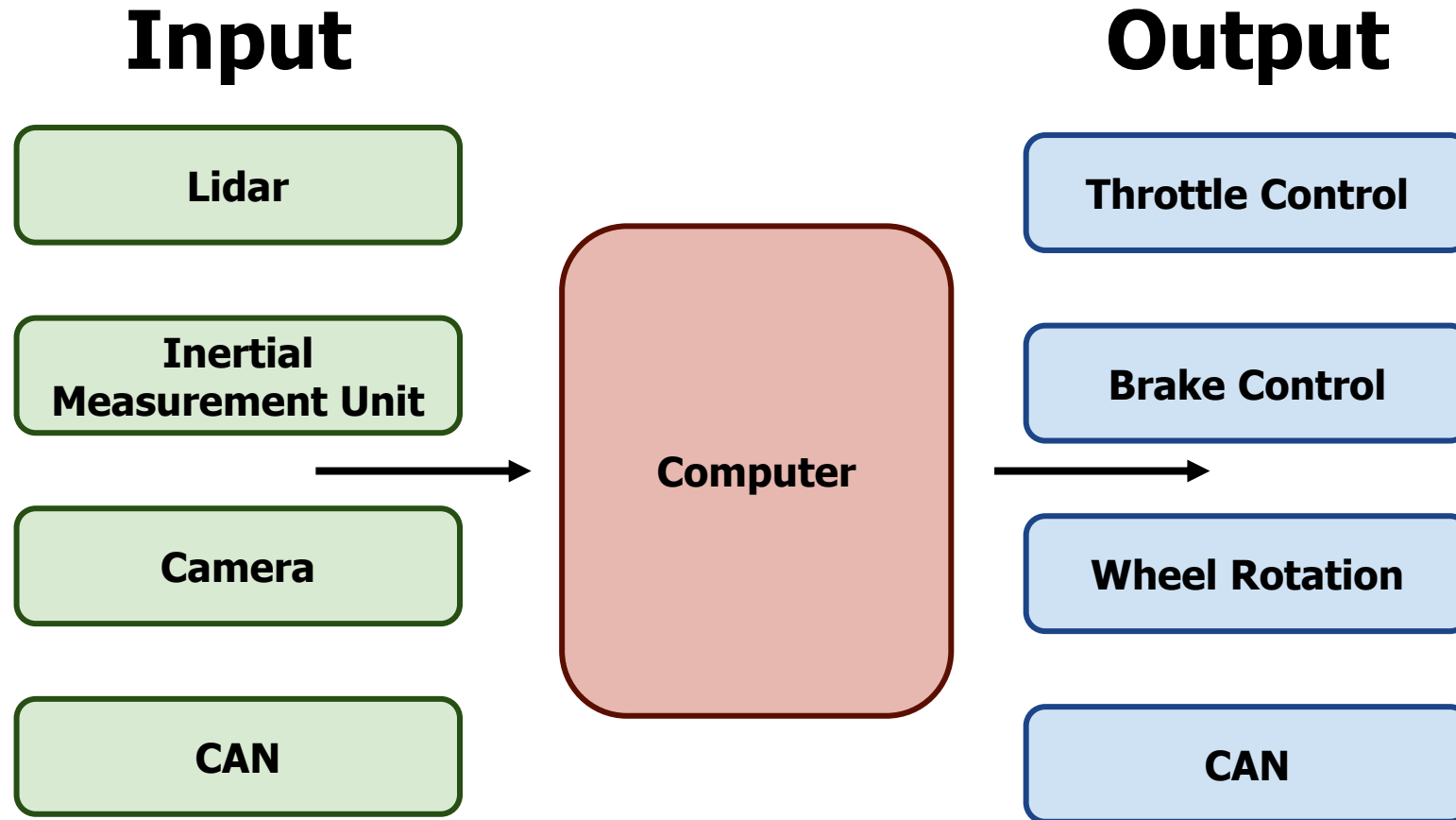
- Computation was sufficient for batch systems
 - Even then, tape was the input and output mechanism
- But interactive systems need to receive input from users and output responses
 - Keyboard/mouse
 - Disk
 - Network
 - Graphics
 - Audio
 - Various USB devices



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

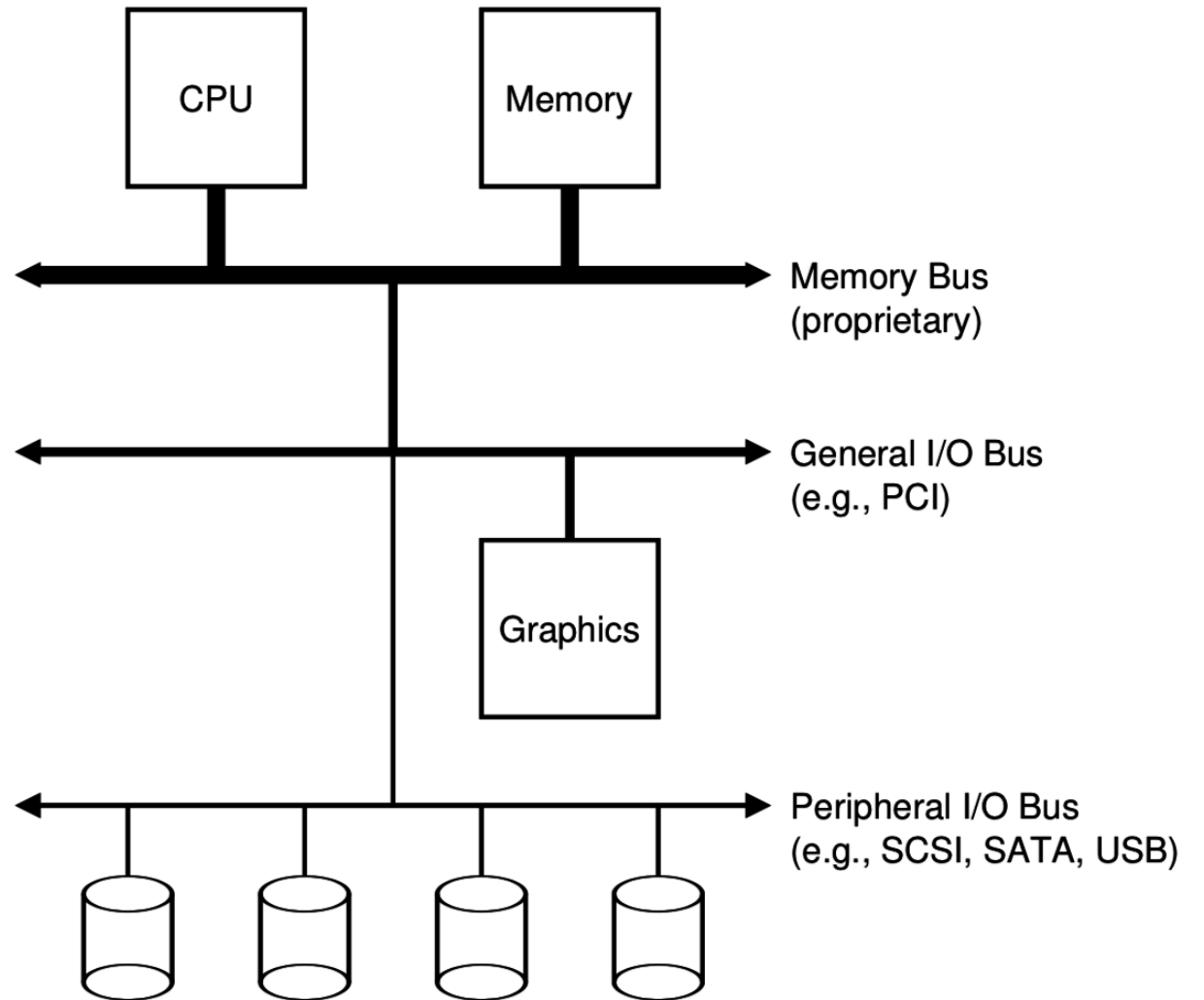
Outline

- 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

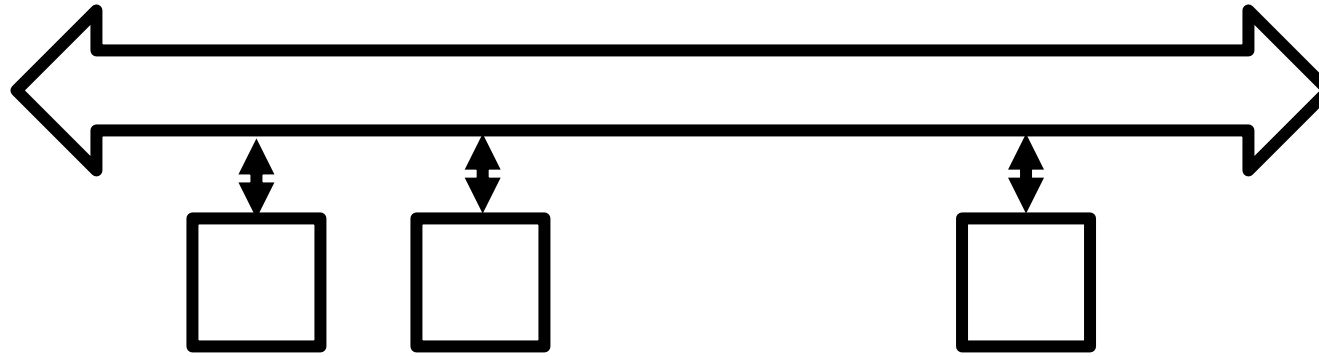
Devices connect to buses on the computer

- 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?



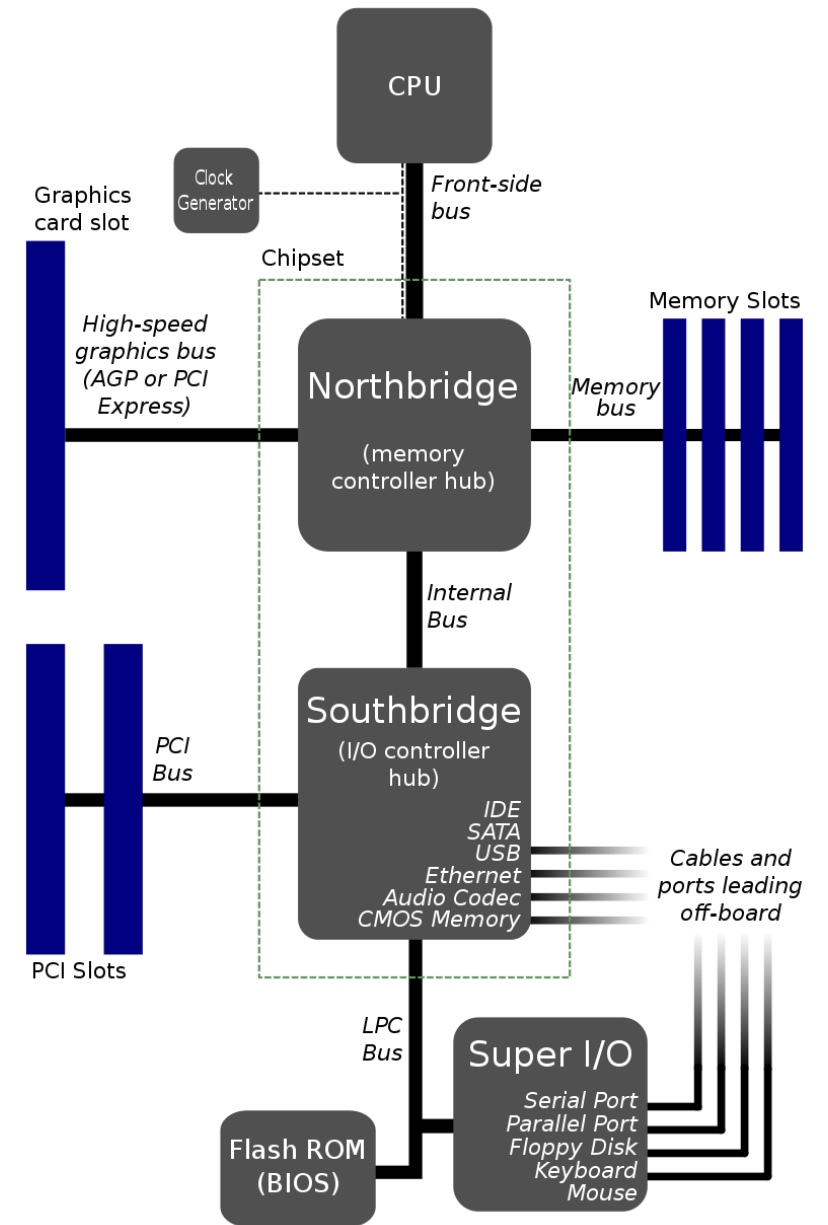
- Common set of wires for communication between two or more components
 - Lets one set of wires connect to N devices
- Standardized buses have a specific set of wires and protocol for communicating over them (DMI, PCI, SATA, USB)
 - Example wires: 64 address wires, 64 data wires, ~10 control wires
- 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
- Internal computer buses differences from general networks
 - Higher speed
 - Very high reliability
 - Accessed cooperatively (often with only one controller: CPU)

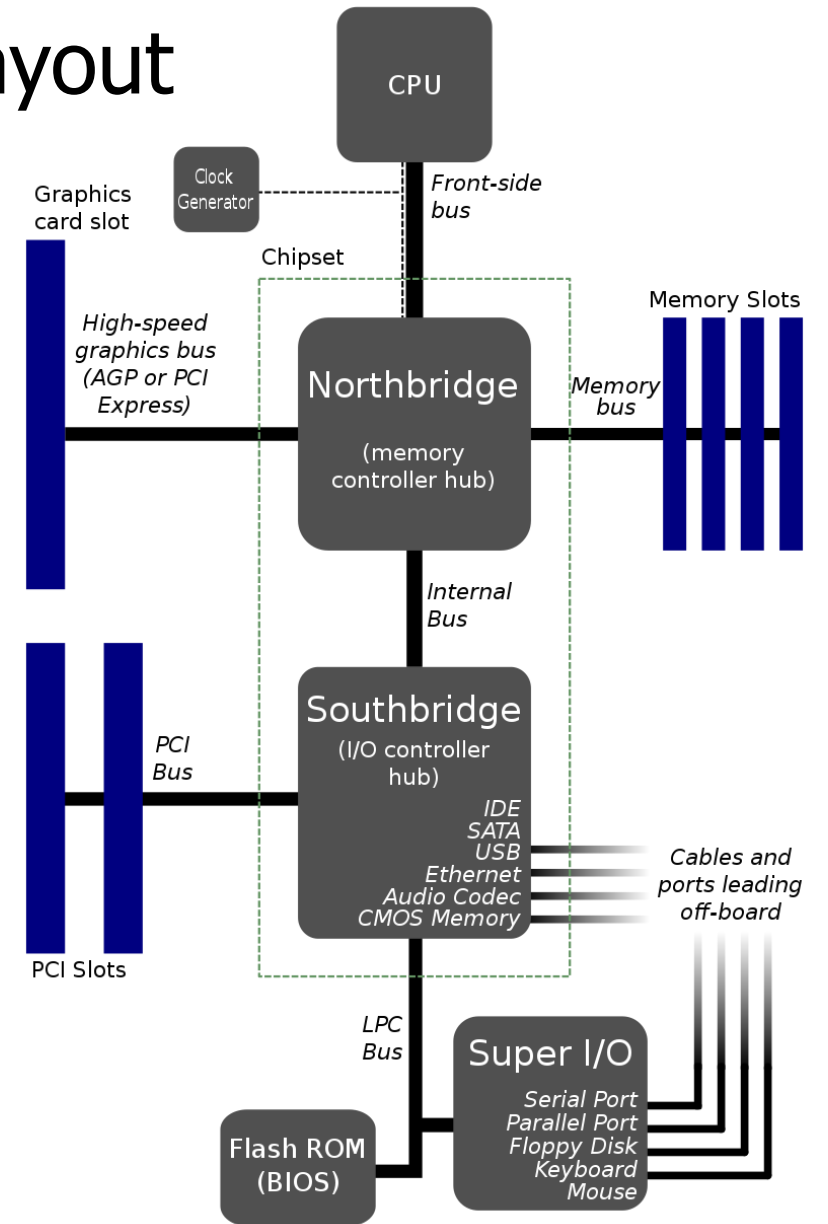
Old version of I/O hierarchy

- Hub architecture
- Northbridge
 - High speed memory control
 - RAM + Graphics
- Southbridge
 - Low speed peripherals
 - Disk, Network, USB



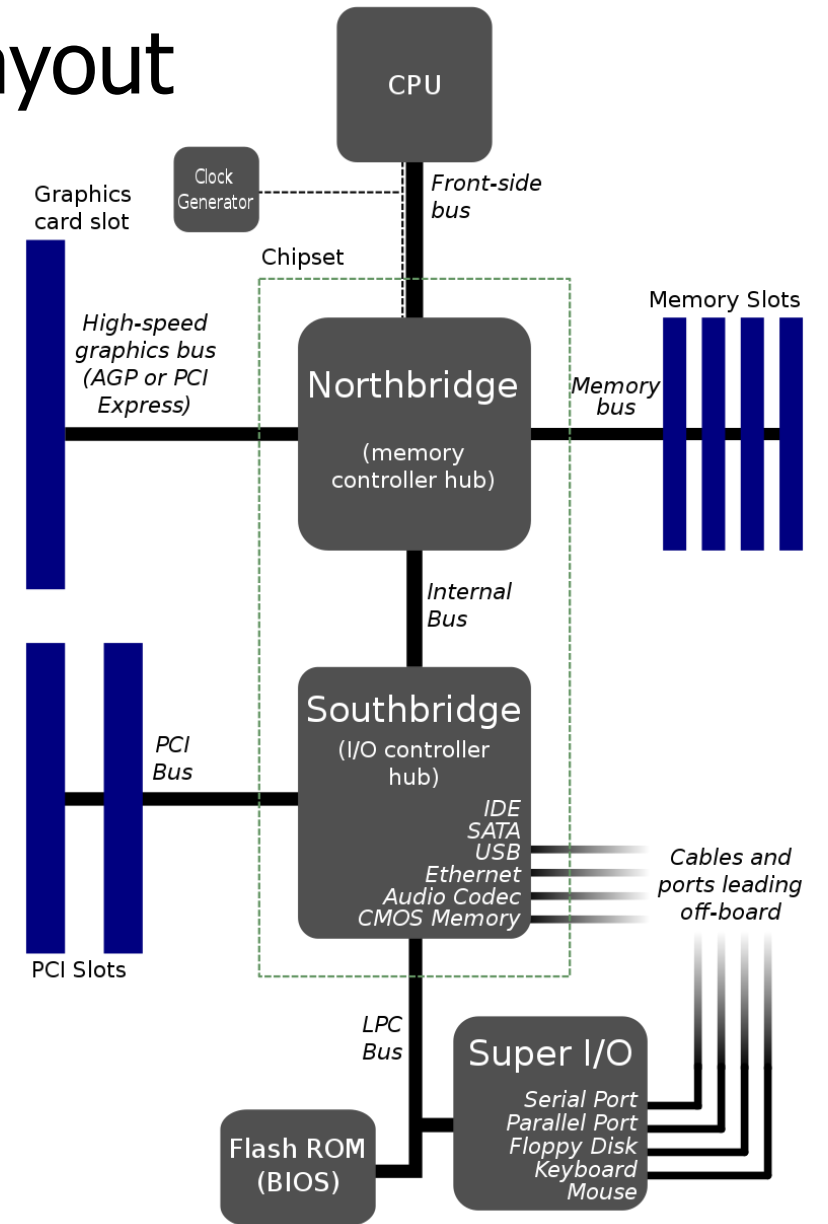
Check your understanding – physical layout

- Which is going to be physically closer to the CPU?
 - Northbridge or Southbridge
- Why?

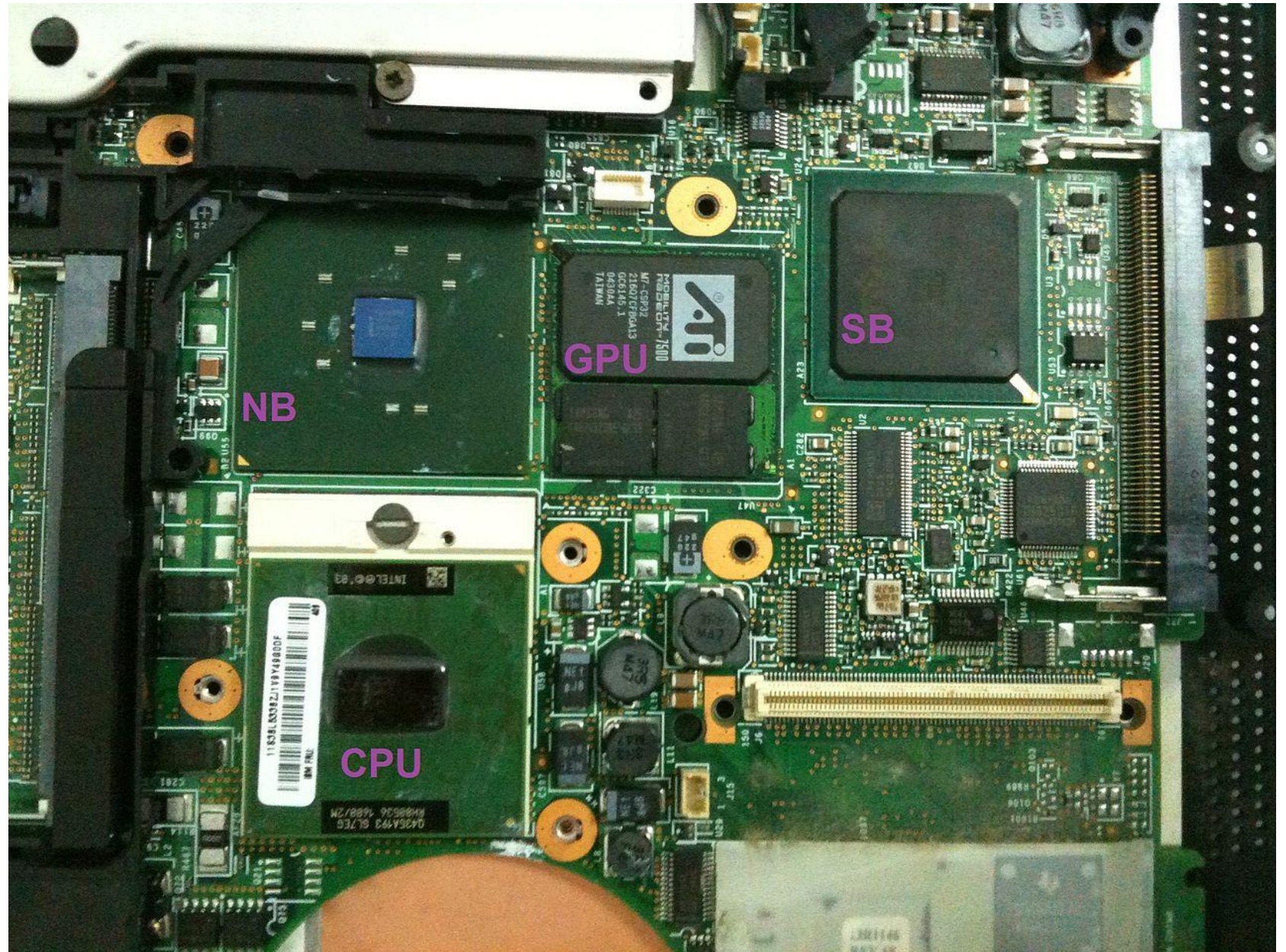


Check your understanding – physical layout

- Which is going to be physically closer to the CPU?
 - **Northbridge** or Southbridge
- Why?
 - Need accesses to be fast
 - Electrical signals are limited by speed of light
 - 1 nanosecond \approx 1 foot

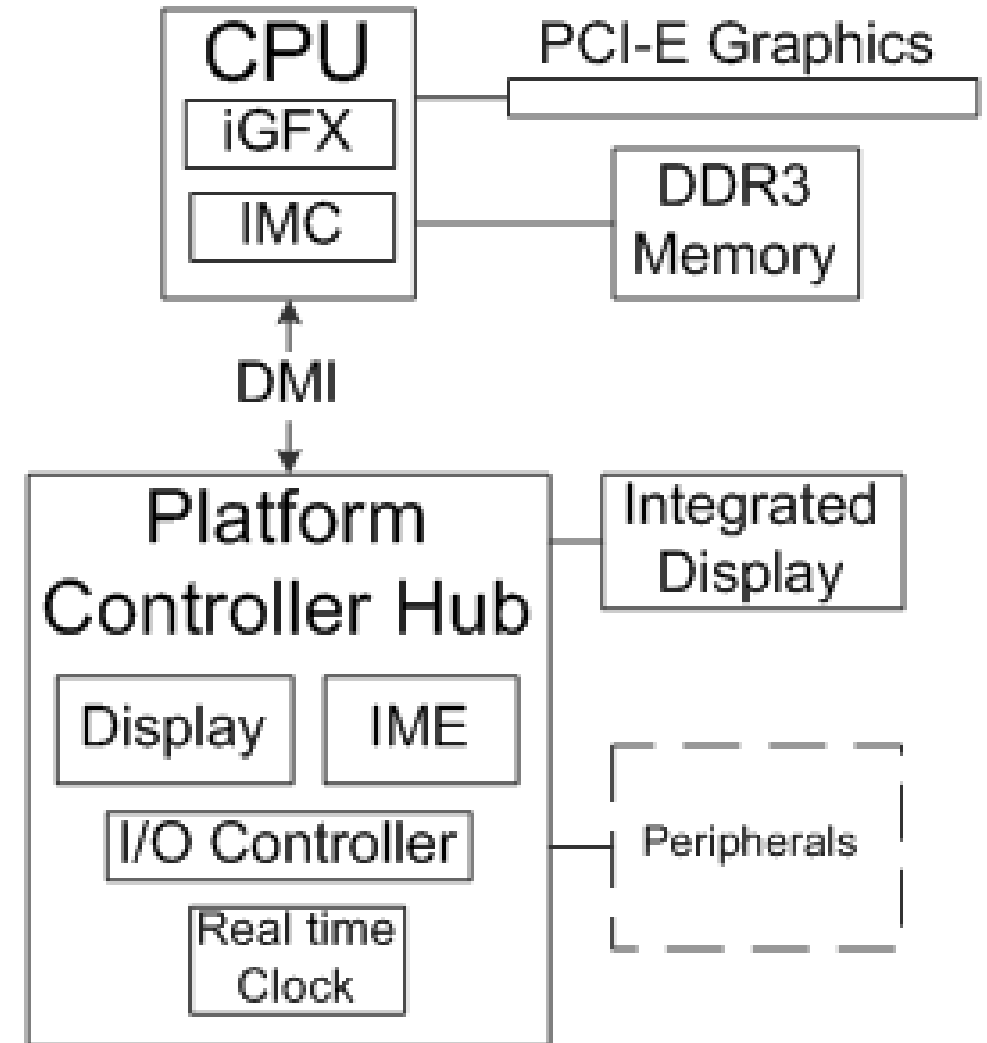


- Physical layout of an old motherboard
- Note physical distance from CPU



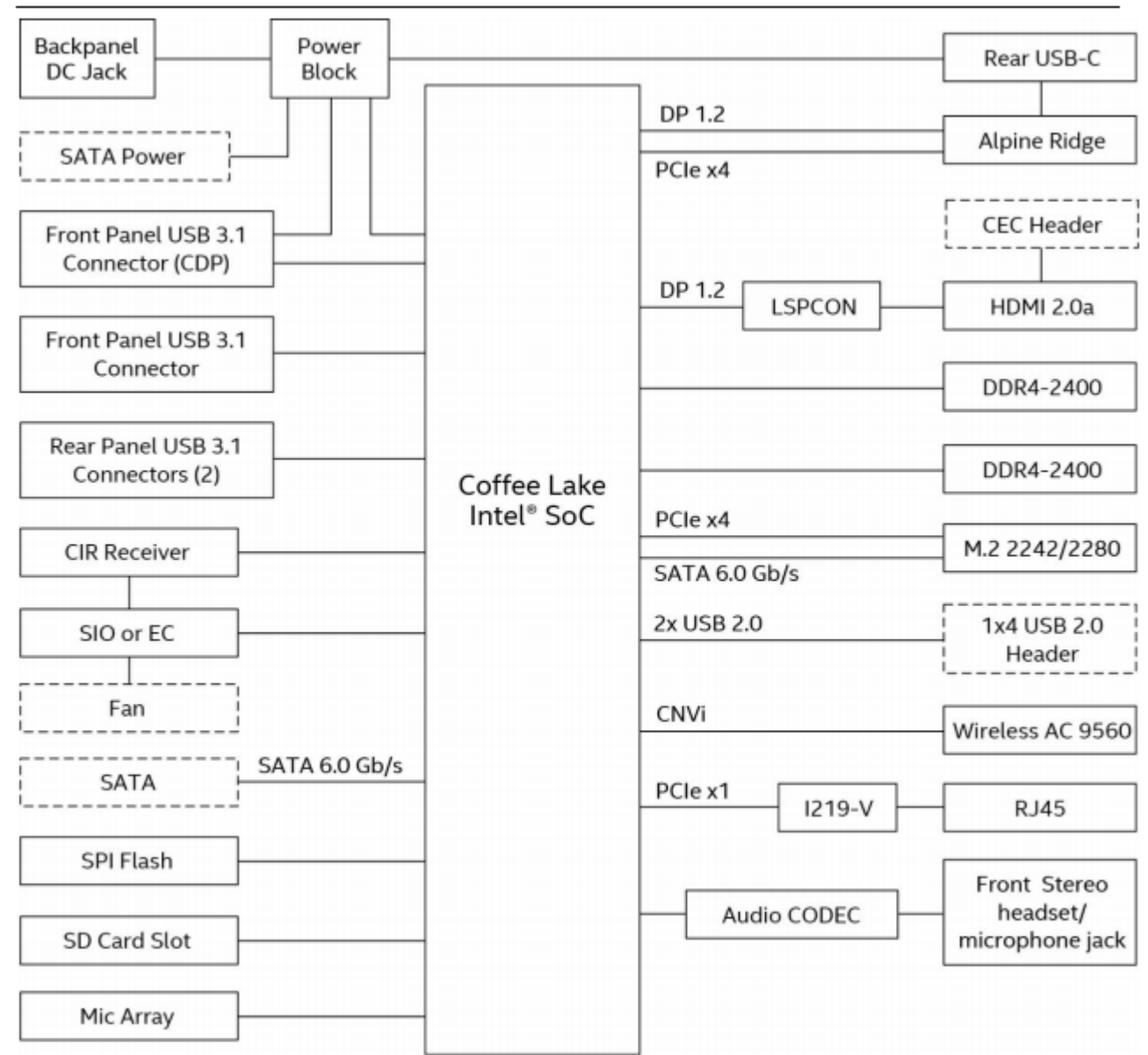
Platform Controller Hub (PCH) architecture

- Enables faster communication high-throughput peripherals
 - Memory and Graphics



Intel NUC 8

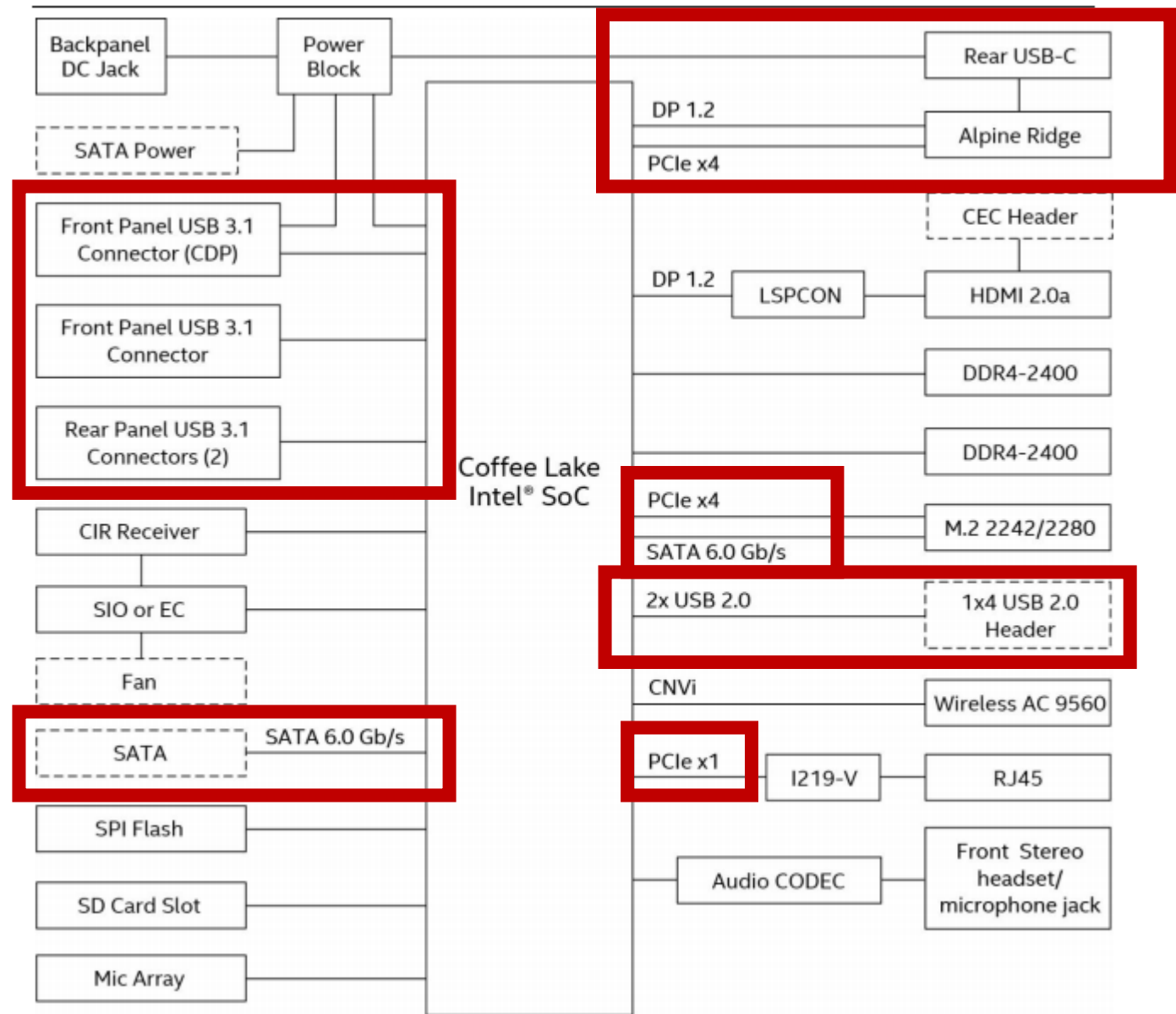
- Small form factor desktop
- CPU includes PCH and GPU in single package
- CPU implements many connections directly
 - USB
 - SATA
 - Some have hardware controllers
 - Ethernet (RJ45)
 - Thunderbolt (Rear USB-C)
 - 3.5mm Audio (Front stereo headset...)



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



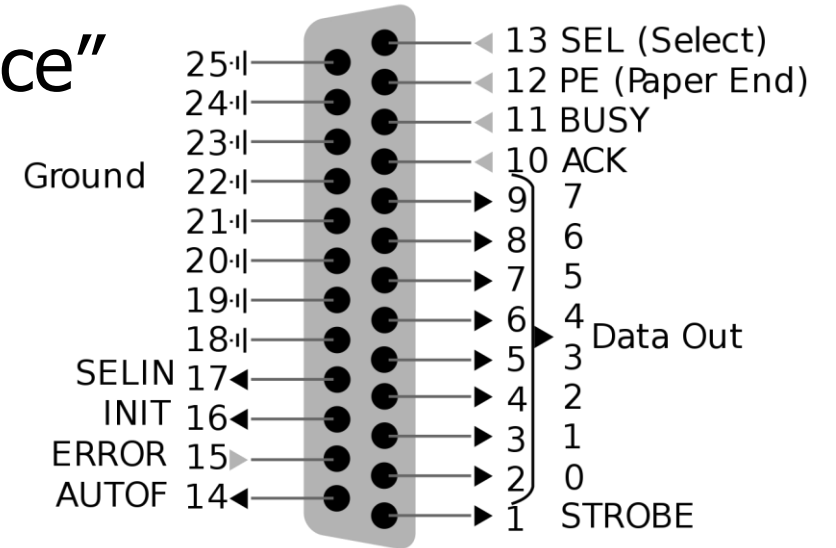
24390

Parallel Port – “Printer Port” or Centronics Port

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

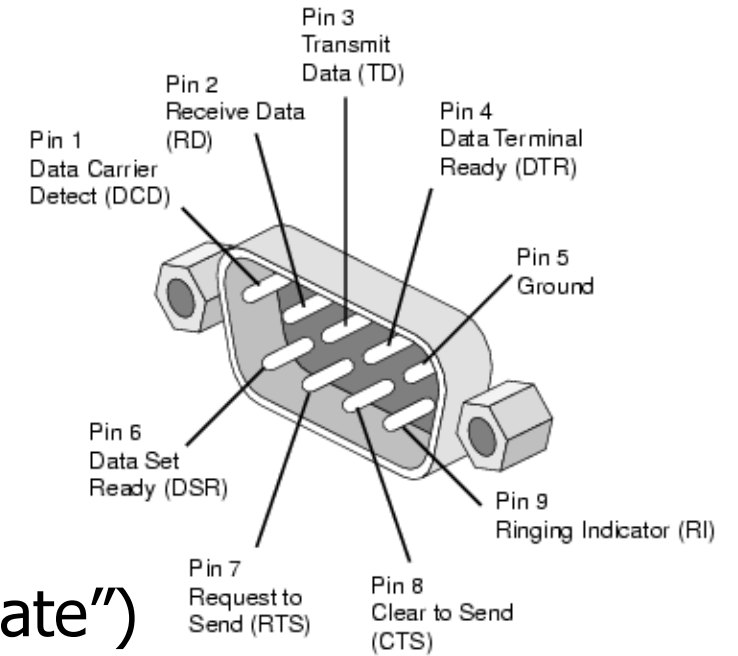
- 8 data bits plus control signals
 - Up to 2.5 Mbps!!

- 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



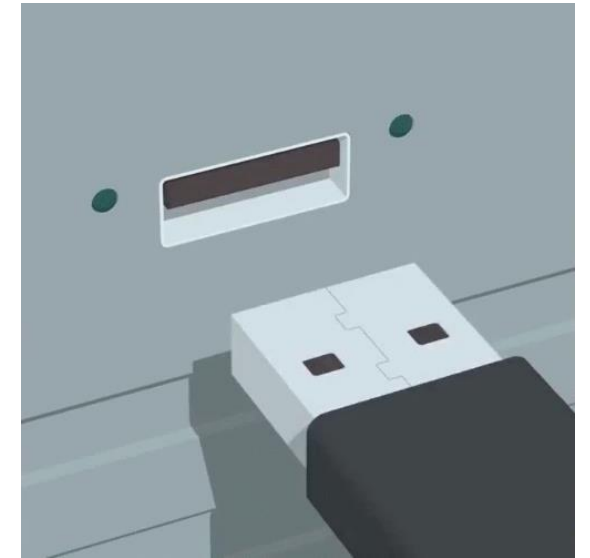
Serial Port – RS-232

- “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”
- 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 a drive 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

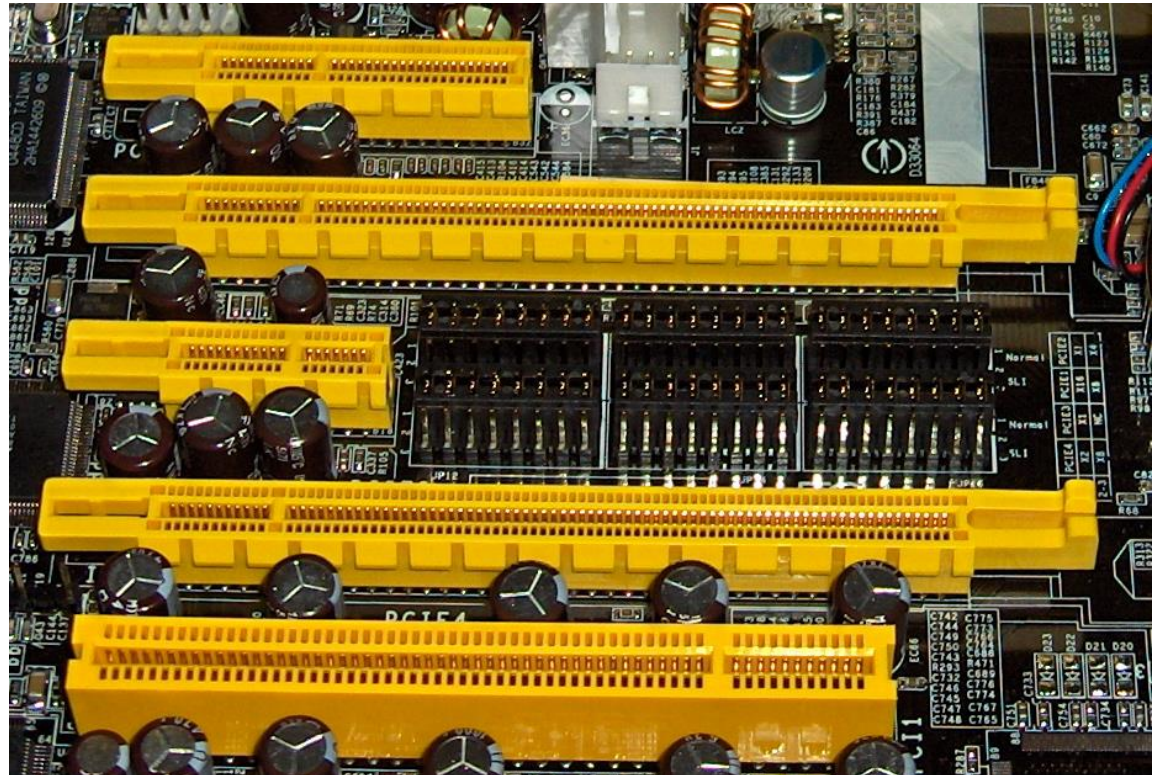


SATA

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

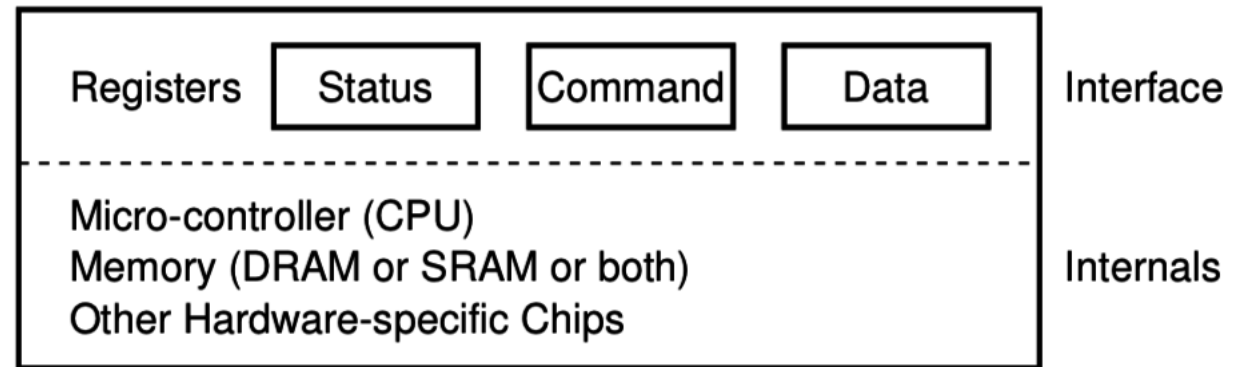


Outline

- 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



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
- Example: read current value from real-time clock

```
mov $0, %al
out %al, $0x70 /* Select output of seconds */
in $0x71, %al  /* AL register contains 0-59 */
```

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

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

Check your understanding – PMIO in C

- How would you access PMIO with C instructions?

Check your understanding – PMIO in C

- How would you access PMIO with C instructions?
 - Can't! Need to use assembly (hopefully with C function wrapper)

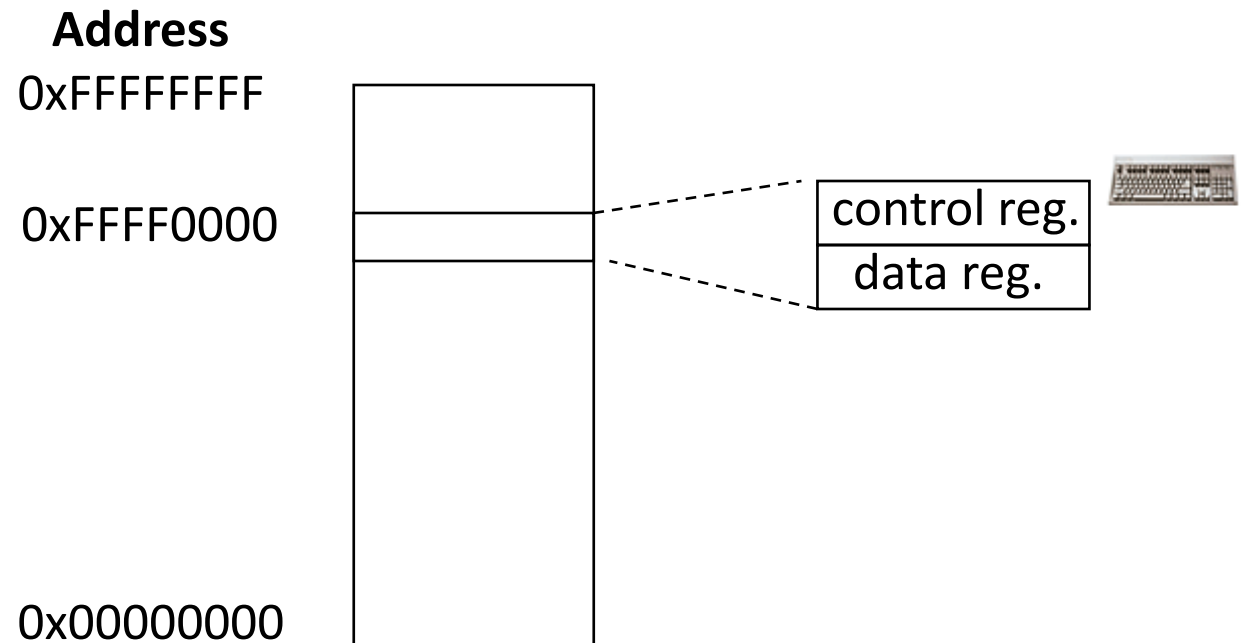
Annoying parts of Port-Mapped I/O

- Special assembly instructions are hard to write in C
 - Need some wrapper function that actually calls them
- Feels sort of like memory read/write, but isn't
 - Why not?
 - Can we just put the "port address space" somewhere in memory?
 - Used to be a problem for 16-bit computers
 - 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 either PMIO or MMIO depending on the device



LED blinking code in C using MMIO (on 32-bit nRF52832)

```
void main(void) {
    const uint32_t LED_NUM = 1;

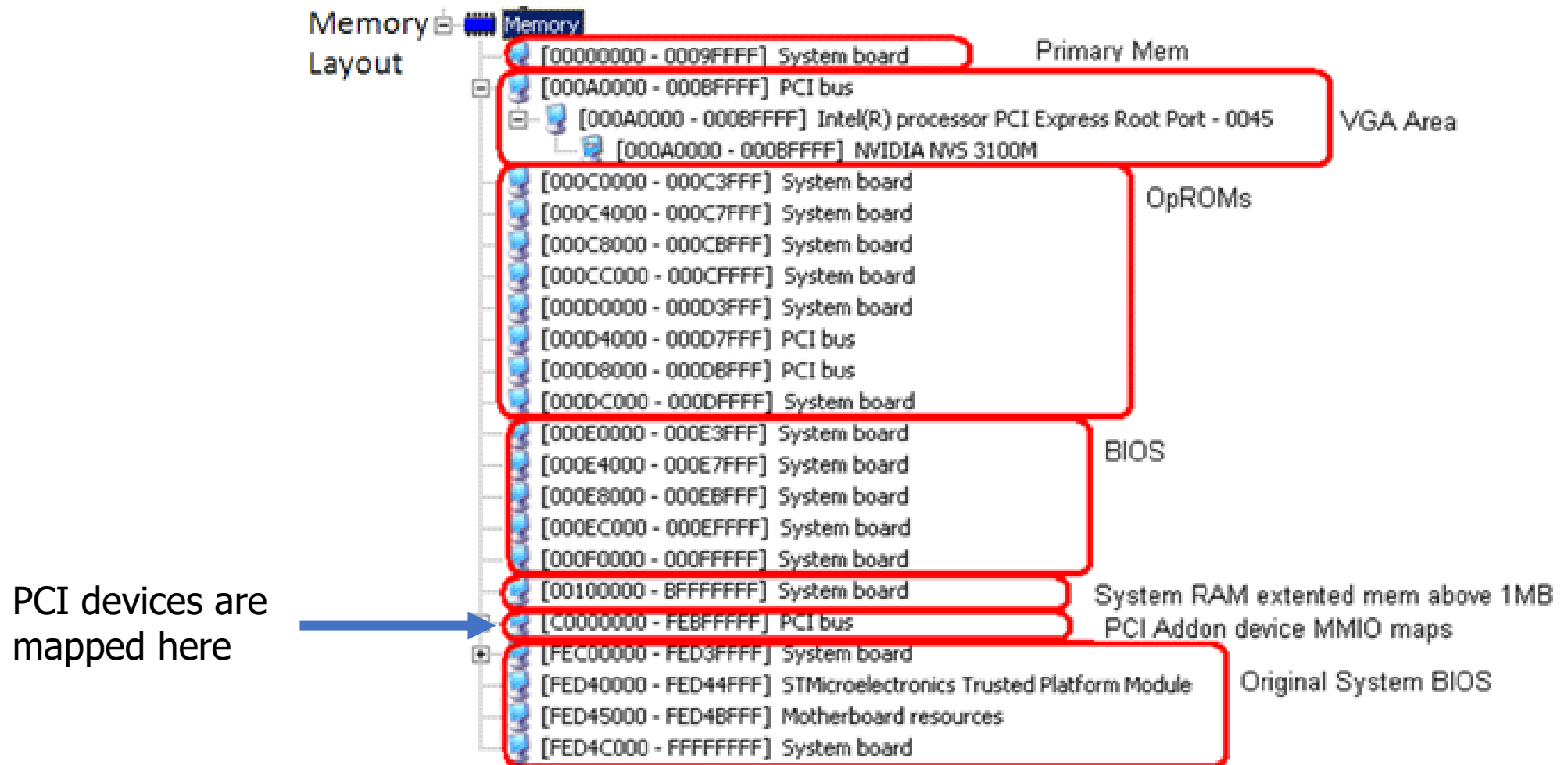
    // set LED pin to be an output
    *((volatile uint32_t*)0x50000700 + 4*LED_NUM) = 0x00000003;

    while (true) {
        volatile int i;

        // LED on
        *((volatile uint32_t*)0x50000504) |= (1 << LED_NUM);
        for (i=0; i<400000; i++); // wait a second

        // LED off
        *((volatile uint32_t*)0x50000504) &= ~(1 << LED_NUM);
        for (i=0; i<400000; i++); // wait a second
    }
}
```

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

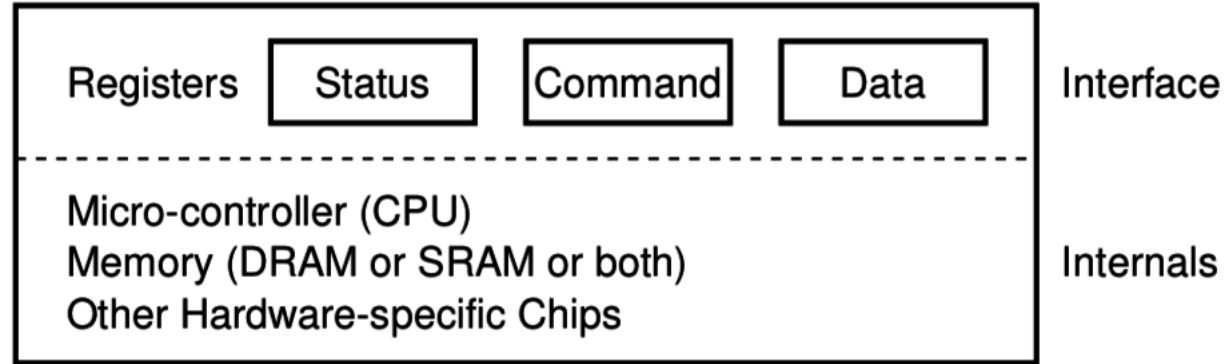
Other details about MMIO

- 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
 - 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

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

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!

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. ¹
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. ²
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. ³
18	#MC	Machine Check	Error codes (if any) and source are model dependent. ⁴
19	#XM	SIMD Floating-Point Exception	SIMD Floating-Point Instruction ⁵
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 (GPU)
 - 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...

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
 - (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 disk

- Let's say that disk 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?

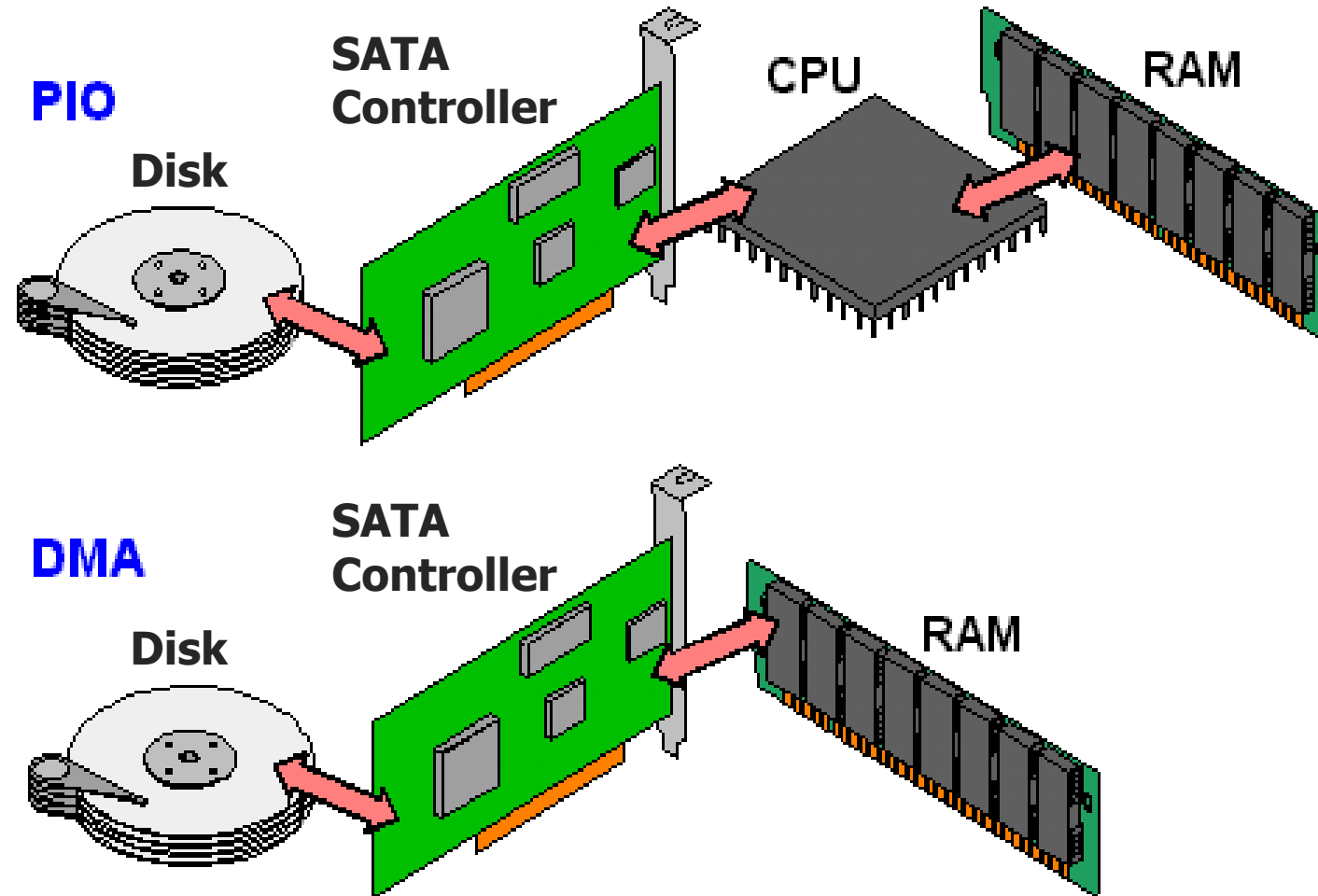
Check your understanding – writing to disk

- Let's say that disk 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 150,000 cycles)

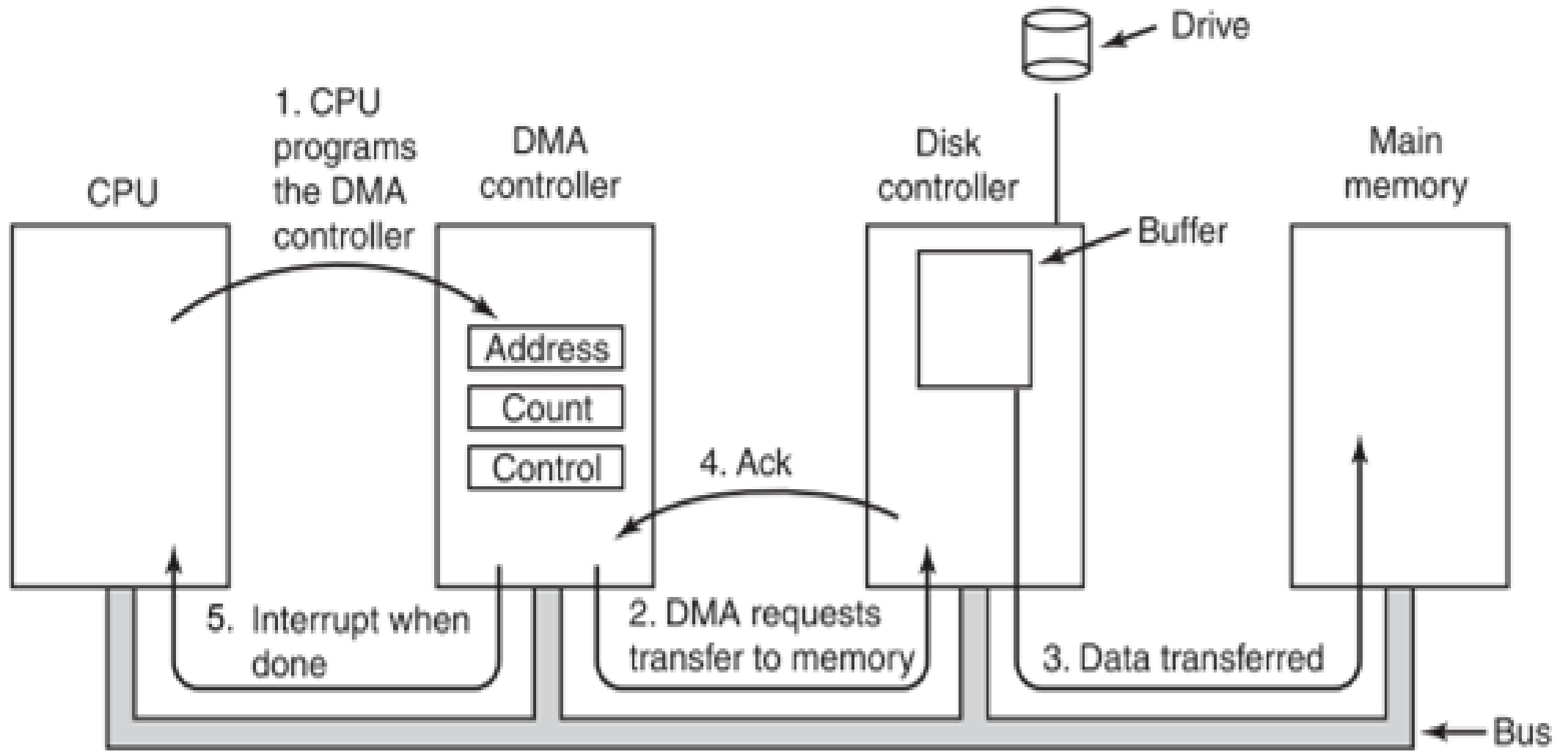
Direct Memory Access (DMA)

- Alternative method that uses hardware to do memory transfers
 - OS writes address of the data and the size
 - Device reads data directly from memory
 - CPU can go do other things while read/write is occurring
- Notes
 - Need to be careful about letting devices access arbitrary memory
 - Are devices part of the trusted compute base?
 - Often a hardware “DMA controller” does the transfer for the device
 - IOMMU can even set up virtual memory spaces for devices

Programmed I/O versus Direct Memory Access



Disk access with DMA



Interaction pattern with Interrupts and DMA

- Interrupt from disk: IDLE (!BUSY)
 - Set up DMA controller
 - Device is disk
 - Select Read or Write
 - Specify memory address and size
 - Start DMA transfer
 - Continue until interrupt again
- OS ends up keeping a queue of transfers to be sent to disk that needs to be accessed asynchronously from several places
 - Producer/Consumer Problem!

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