

I/O

**Prof. Hakim Weatherspoon**

**CS 3410, Spring 2015**

Computer Science

Cornell University

See: *Online* P&H Chapter 6.5-6

# Announcements

## Prelim2 Topics

- Lecture: Lectures 10 to 24
- Data and Control Hazards (Chapters 4.7-4.8)
- RISC/CISC (Chapters 2.16-2.18, 2.21)
- Calling conventions and linkers (Chapters 2.8, 2.12, Appendix A.1-6)
- Caching and Virtual Memory (Chapter 5)
- Multicore/parallelism (Chapter 6)
- Synchronization (Chapter 2.11)
- Traps, Exceptions, OS (Chapter 4.9, Appendix A.7, pp 445-452)
  
- HW2, Labs 3/4, C-Labs 2/3, PA2/3
  
- Topics from Prelim1 (not the focus, but some possible questions)

# Announcements

Project3 submit “*souped up*” bot to CMS

**Project3 Cache Race Games night Monday, May 4<sup>th</sup>, 5pm**

- Come, eat, drink, have fun and be merry!
- Location: **B17 Upson Hall**

Prelim2: **Thursday**, April 30<sup>th</sup> in evening

- Time and Location: **7:30pm sharp** in **Statler Auditorium**
- Old prelims are online in CMS
- Prelim Review Session:  
**TODAY**, Tuesday, April 28, 7-9pm in B14 Hollister Hall

Project4:

- Design Doc due May 5<sup>th</sup>, bring design doc to mtg May 4-6
- Demos: May 12 and 13
- **Will not be able to use slip days**

# iClicker Check

# Goals for Today

## Computer System Organization

How does a processor interact with its environment?

- I/O Overview

How to talk to device?

- Programmed I/O or Memory-Mapped I/O

How to get events?

- Polling or Interrupts

How to transfer lots of data?

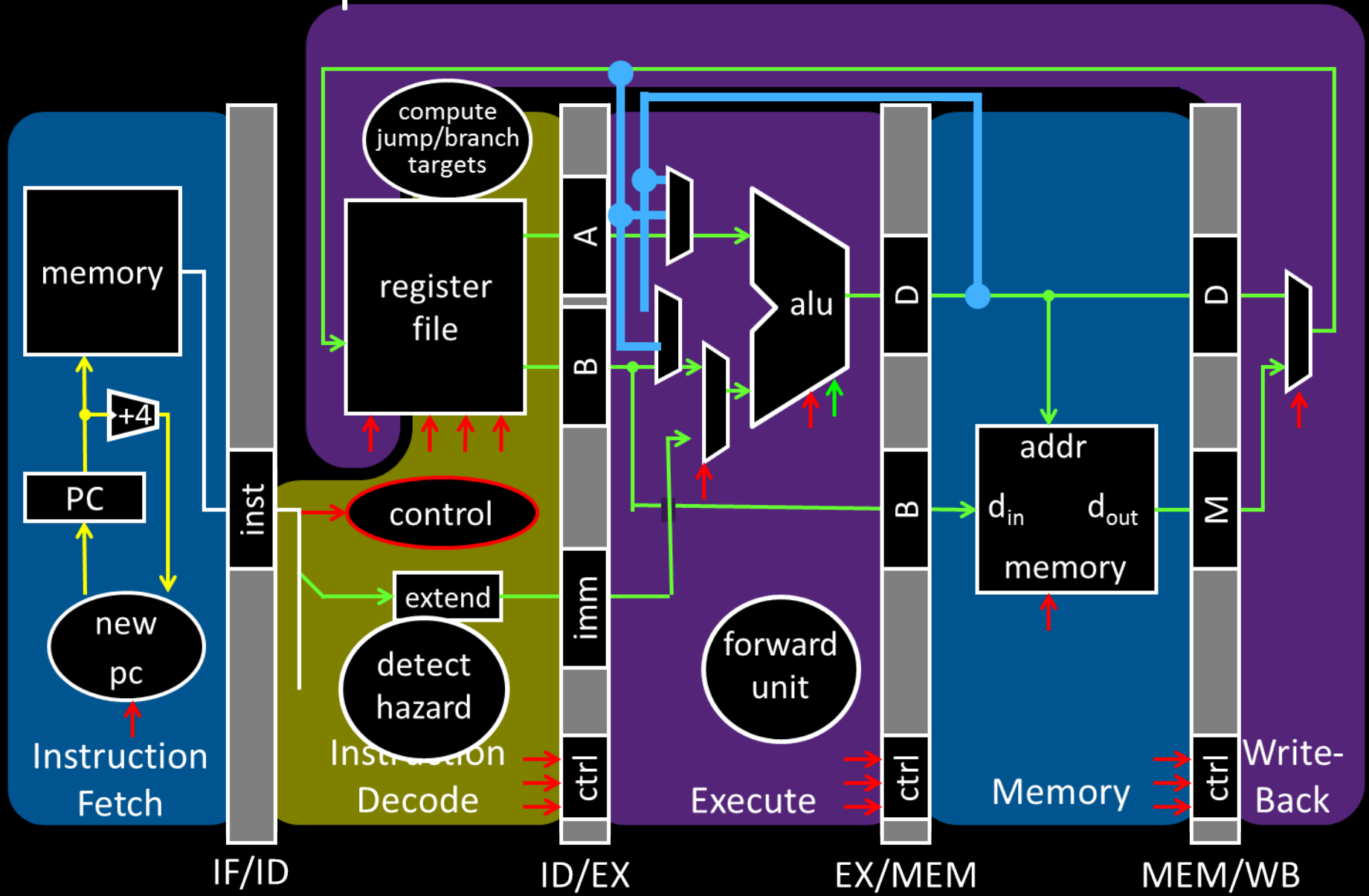
- Direct Memory Access (DMA)

# Next Goal

How does a processor interact with its environment?

# Big Picture: Input/Output (I/O)

How does a processor interact with its environment?

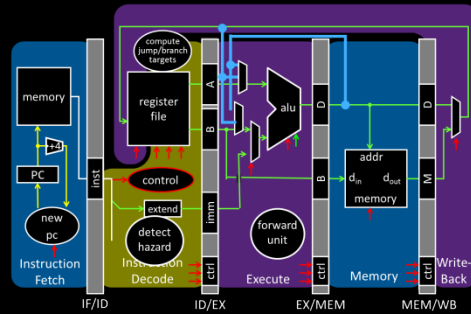


# Big Picture: Input/Output (I/O)

How does a processor interact with its environment?

Computer System Organization =

Memory +  
Datapath +  
Control +  
Input +  
Output



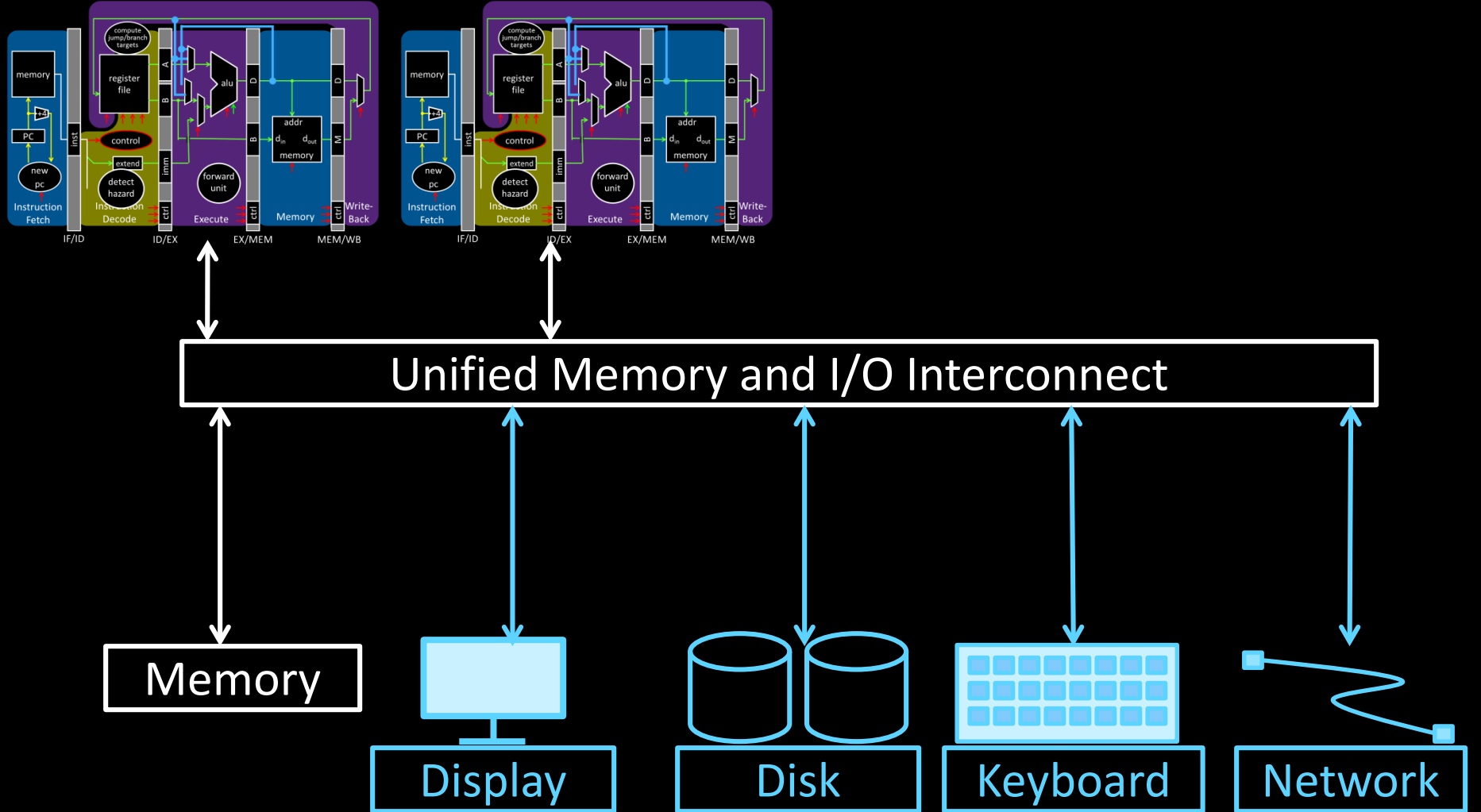


# I/O Devices Enables Interacting with Environment

Device	Behavior	Partner	Data Rate (b/sec)
Keyboard	Input	Human	100
Mouse	Input	Human	3.8k
Sound Input	Input	Machine	3M
Voice Output	Output	Human	264k
Sound Output	Output	Human	8M
Laser Printer	Output	Human	3.2M
Graphics Display	Output	Human	800M – 8G
Network/LAN	Input/Output	Machine	100M – 10G
Network/Wireless LAN	Input/Output	Machine	11 – 54M
Optical Disk	Storage	Machine	5 – 120M
Flash memory	Storage	Machine	32 – 200M
Magnetic Disk	Storage	Machine	800M – 3G

# Attempt#1: All devices on one interconnect

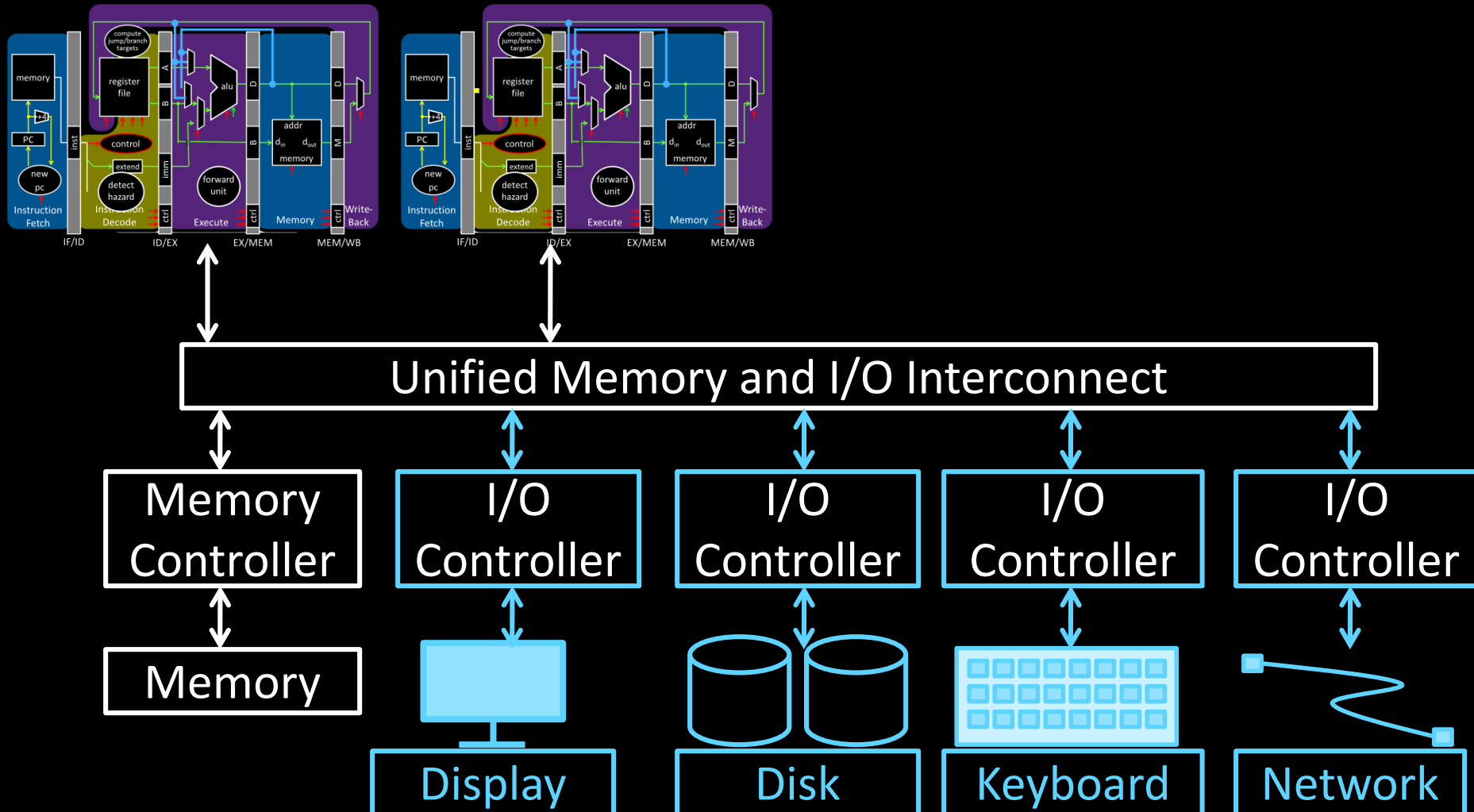
Replace *all* devices as the interconnect changes  
e.g. keyboard speed == main memory speed ?!



# Attempt#2: I/O Controllers

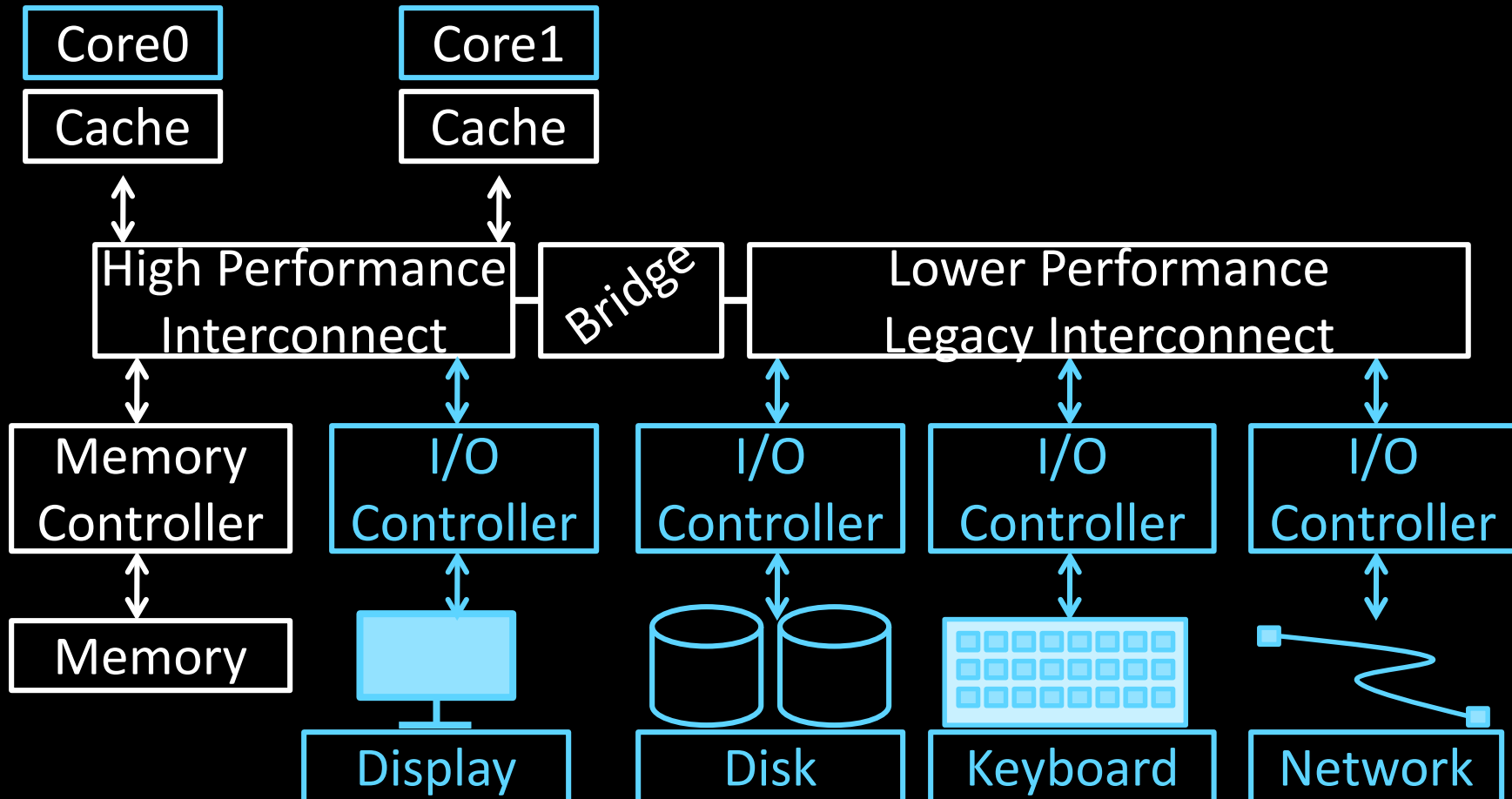
Decouple I/O devices from Interconnect

Enable smarter I/O interfaces



# Attempt#3: I/O Controllers + Bridge

Separate high-performance processor, memory, display interconnect from lower-performance interconnect



# Bus Parameters

Width = number of wires

Transfer size = data words per bus transaction

Synchronous (with a bus clock)

or asynchronous (no bus clock / “self clocking”)

# Bus Types

## Processor – Memory (“Front Side Bus”. Also QPI)

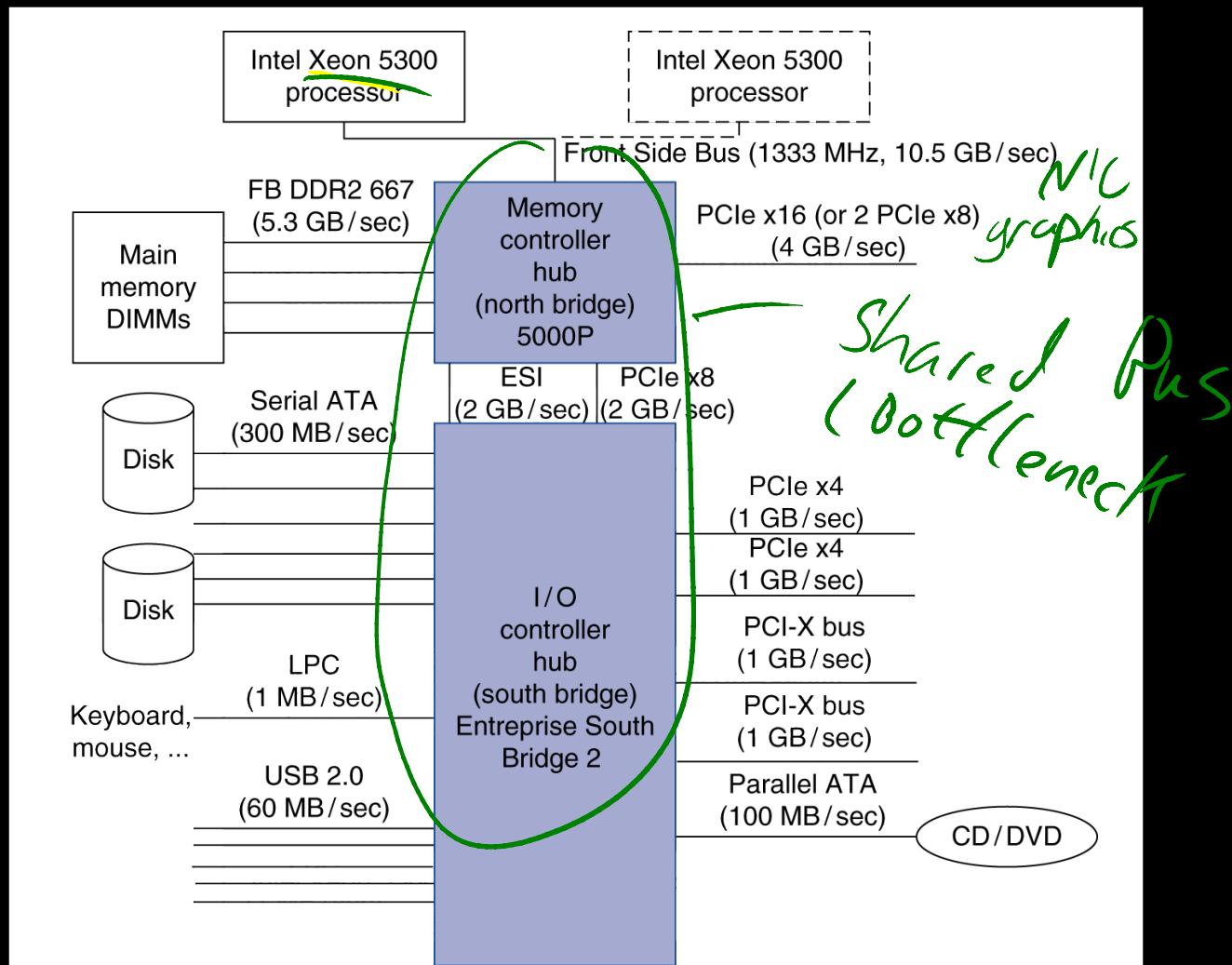
- Short, fast, & wide
- Mostly fixed topology, designed as a “chipset”
  - CPU + Caches + Interconnect + Memory Controller

## I/O and Peripheral busses (PCI, SCSI, USB, LPC, ...)

- Longer, slower, & narrower
- Flexible topology, multiple/varied connections
- Interoperability standards for devices
- Connect to processor-memory bus through a bridge

# Attempt#3: I/O Controllers + Bridge

Separate high-performance processor, memory, display interconnect from lower-performance interconnect



# Example Interconnects

Name	Use	Devics per channel	Channel Width	Data Rate (B/sec)
Firewire 800	External	63	4	100M
USB 2.0	External	127	2	60M
USB 3.0	External	127	2	625M
Parallel ATA	Internal	1	16	133M
Serial ATA (SATA)	Internal	1	4	300M
PCI 66MHz	Internal	1	32-64	533M
PCI Express v2.x	Internal	1	2-64	16G/dir
Hypertransport v2.x	Internal	1	2-64	25G/dir
QuickPath (QPI)	Internal	1	40	12G/dir



# Interconnecting Components

Interconnects are (were?) busses

- parallel set of wires for data and control
- shared channel
  - multiple senders/receivers
  - everyone can see all bus transactions
- bus protocol: rules for using the bus wires

e.g. Intel  
Xeon

Alternative (and increasingly common):

- dedicated point-to-point channels

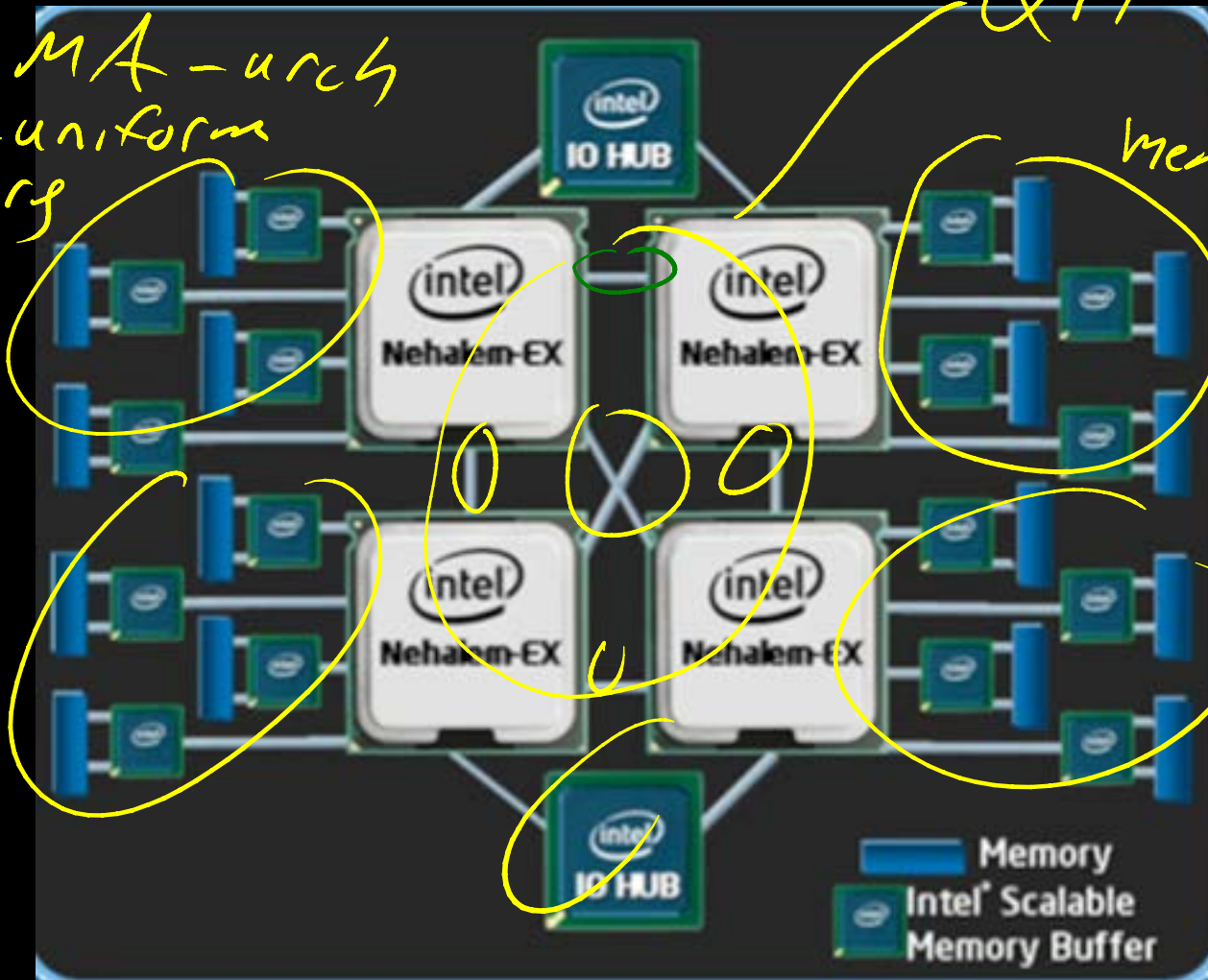
e.g. Intel  
Nehalem

# Attempt#4: I/O Controllers+Bridge+ NUMA

Remove bridge as bottleneck with Point-to-point interconnects

E.g. Non-Uniform Memory Access (NUMA)

*NUMA-arch  
non-uniform  
memory  
Arch  
(No  
Shared  
bus)*



*Point-Point  
Quick  
Path  
interconnect  
Mem*

Memory  
Intel Scalable  
Memory Buffer

# Takeaways

Diverse I/O devices require hierarchical interconnect which is more recently transitioning to point-to-point topologies.

# Next Goal

How does the processor interact with I/O devices?

# I/O Device Driver Software Interface

Set of methods to write/read data to/from device and control device

Example: Linux Character Devices

```
// Open a toy " echo " character device
int fd = open("/dev/echo", O_RDWR);

// Write to the device
char write_buf[] = "Hello World!";
write(fd, write_buf, sizeof(write_buf));

// Read from the device
char read_buf [32];
read(fd, read_buf, sizeof(read_buf));

// Close the device
close(fd);

// Verify the result
assert(strcmp(write_buf, read_buf)==0);
```

# I/O Device API

## Typical I/O Device API

- a set of read-only or read/write registers

### Command registers

- writing causes device to do something

### Status registers

- reading indicates what device is doing, error codes, ...

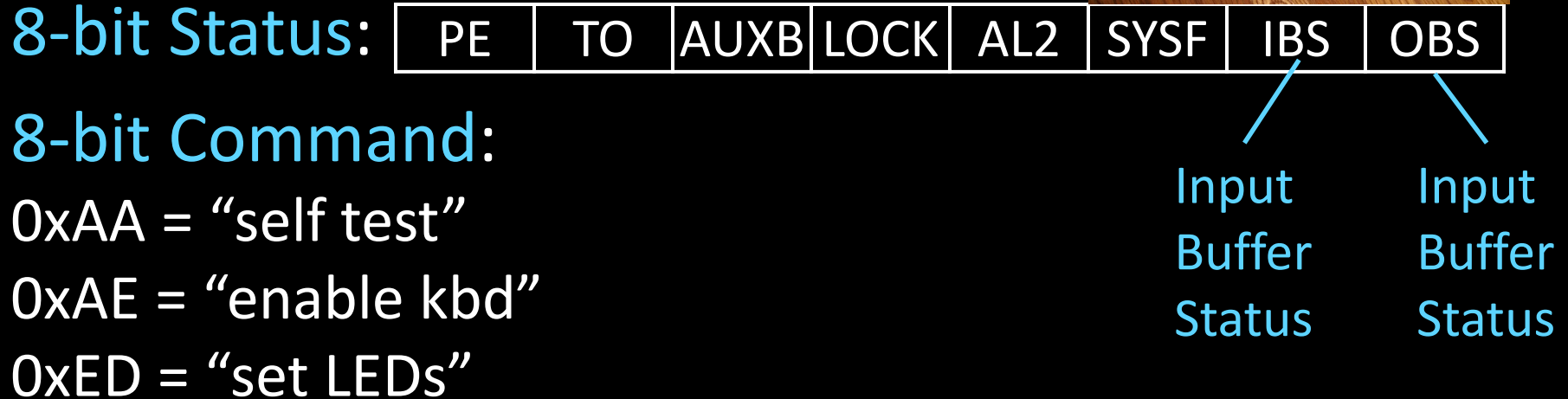
### Data registers

- Write: transfer data to a device
- Read: transfer data from a device

Every device uses this API

# I/O Device API

Simple (old) example: AT Keyboard Device



8-bit Data:

scancode (when reading)

LED state (when writing) or ...

# Communication Interface

Q: How does ~~program~~ ~~OS~~ code talk to device?

A: special instructions to talk over special busses

Programmed I/O ← Interact with cmd, status, and data device registers directly

- `inb $a, 0x64` ← kbd status register

- `outb $a, 0x60` ← kbd data register

- Specifies: device, data, direction

- Protection: only allowed in kernel mode

Kernel boundary crossing is expensive

\*x86: `$a` implicit; also `inw`, `outw`, `inh`, `outh`, ...



# Communication Interface

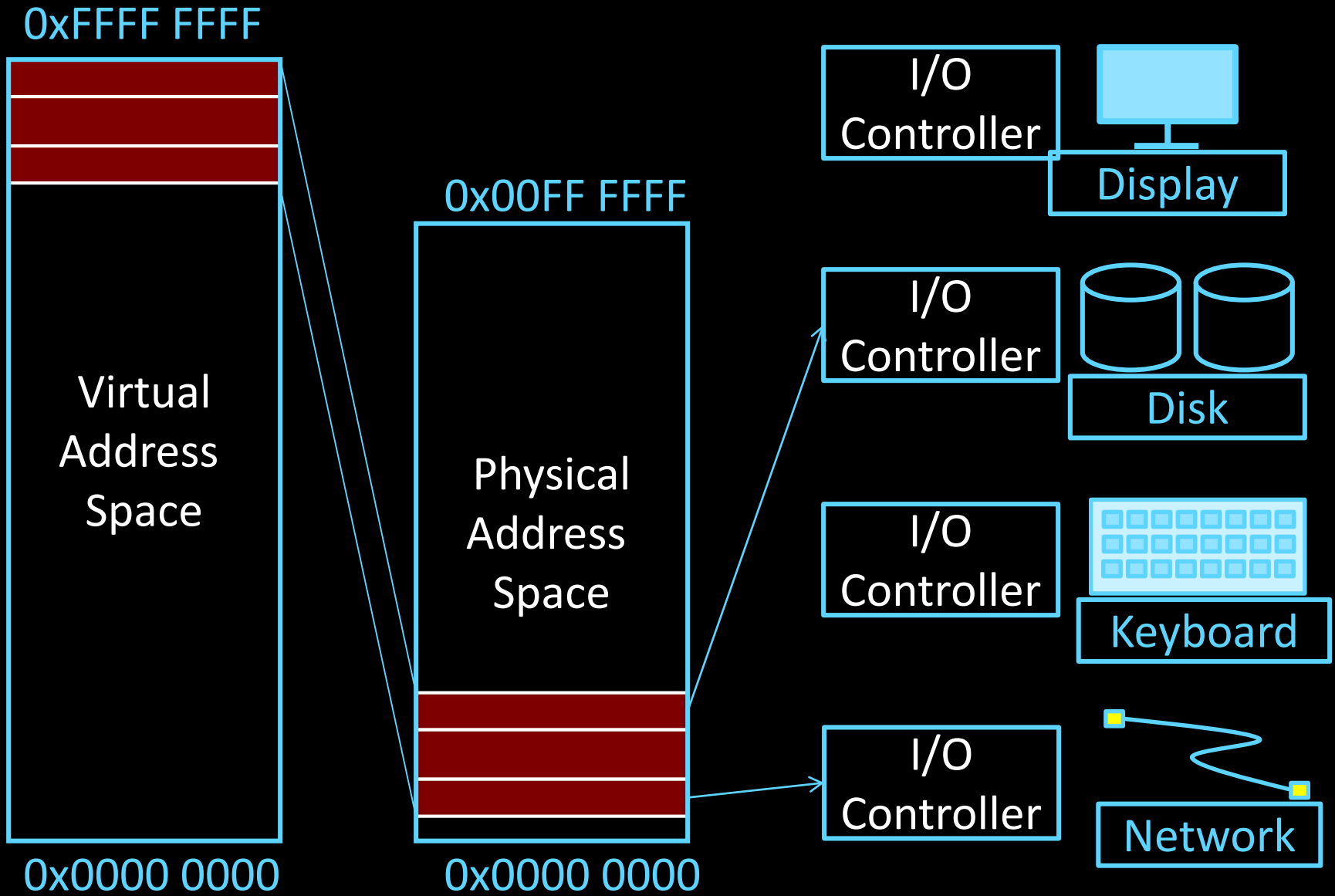
Q: How does ~~program~~ ~~OS~~ code talk to device?

A: Map registers into virtual address space

Memory-mapped I/O ← Faster. Less boundary crossing

- Accesses to certain addresses redirected to I/O devices
- Data goes over the memory bus
- Protection: via bits in pagetable entries
- OS+MMU+devices configure mappings

# Memory-Mapped I/O



# Device Drivers

## Programmed I/O

Polling examples,  
But mmap I/O more  
efficient

```
char read_kbd()
{
do {
    sleep();
    status = inb(0x64);
} while(!(status & 1));
return inb(0x60);
}
```

syscall

**NO**  
syscall

## Memory Mapped I/O

```
struct kbd {
    char status, pad[3];
    char data, pad[3];
};
```

```
kbd *k = mmap(...);
```

syscall

```
char read_kbd()
{
do {
    sleep();
    status = k->status;
} while(!(status & 1));
return k->data;
}
```

# Comparing Programmed I/O vs Memory Mapped I/O

## Programmed I/O

- Requires special instructions
- Can require dedicated hardware interface to devices
- Protection enforced via kernel mode access to instructions
- Virtualization can be difficult

## Memory-Mapped I/O

- Re-uses standard load/store instructions
- Re-uses standard memory hardware interface
- Protection enforced with normal memory protection scheme
- Virtualization enabled with normal memory virtualization scheme

# Takeaways

Diverse I/O devices require hierarchical interconnect which is more recently transitioning to point-to-point topologies.

Memory-mapped I/O is an elegant technique to read/write device registers with standard load/stores.

# Next Goal

How does the processor know device is ready/done?

# Communication Method

Q: How does program learn device is ready/done?

A: **Polling**: Periodically check I/O status register

- If device ready, do operation
- If device done, ...
- If error, take action

```
char read_kbd()
{
do {
    sleep();
    status = inb(0x64);
} while(!(status & 1));

return inb(0x60); }
```

Pro? Con?

- Predictable timing & inexpensive
- But: wastes CPU cycles if nothing to do
- Efficient if there is always work to do (e.g. 10Gbps NIC)

Common in small, cheap, or real-time embedded systems

Sometimes for very active devices too...

# Communication Method

Q: How does program learn device is ready/done?

A: **Interrupts:** Device sends interrupt to CPU

- Cause register identifies the interrupting device
- interrupt handler examines device, decides what to do

Priority interrupts

- Urgent events can interrupt lower-priority interrupt handling
- OS can ~~disable~~ defer interrupts

Pro? Con?

- More efficient: only interrupt when device ready/done
- Less efficient: more expensive since save CPU context
  - CPU context: PC, SP, registers, etc
- Con: unpredictable b/c event arrival depends on other devices' activity



# Takeaways

Diverse I/O devices require hierarchical interconnect which is more recently transitioning to point-to-point topologies.

Memory-mapped I/O is an elegant technique to read/write device registers with standard load/stores.

Interrupt-based I/O avoids the wasted work in polling-based I/O and is usually more efficient

# Next Goal

How do we transfer a *lot* of data *efficiently*?

# I/O Data Transfer

How to talk to device?

- Programmed I/O or Memory-Mapped I/O

How to get events?

- Polling or Interrupts

How to transfer lots of data?

```
disk->cmd = READ_4K_SECTOR;
disk->data = 12;
while (!(disk->status & 1) { }
for (i = 0..4k)
    buf[i] = disk->data;
```

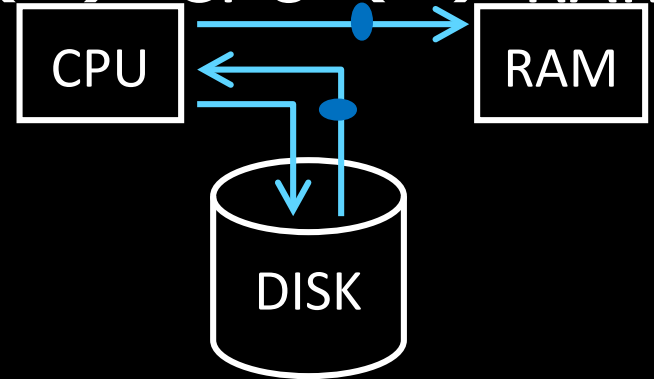
Very,  
Very,  
Expensive

# I/O Data Transfer

Programmed I/O xfer: Device  $\leftrightarrow$  CPU  $\leftrightarrow$  RAM

for  $(i = 1 .. n)$

- CPU issues read request
- Device puts data on bus & CPU reads into registers
- CPU writes data to memory
- **Not** efficient



Read from Disk  
Write to Memory  
**Everything** interrupts CPU  
**Wastes** CPU

# I/O Data Transfer

Q: How to transfer lots of data *efficiently*?

A: Have device access memory directly

## Direct memory access (DMA)

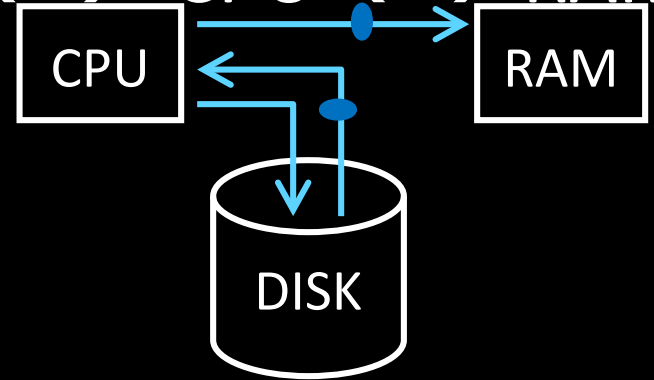
- 1) OS provides starting address, length
- 2) controller (or device) transfers data autonomously
- 3) Interrupt on completion / error

# DMA: Direct Memory Access

Programmed I/O xfer: Device  $\leftrightarrow$  CPU  $\leftrightarrow$  RAM

for (i = 1 .. n)

- CPU issues read request
- Device puts data on bus & CPU reads into registers
- CPU writes data to memory

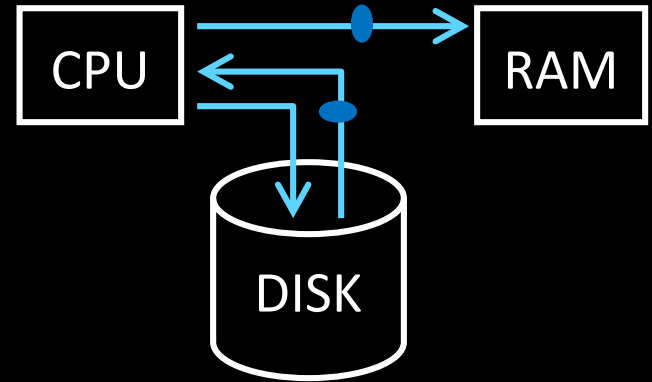


# DMA: Direct Memory Access

Programmed I/O xfer: Device  $\leftrightarrow$  CPU  $\leftrightarrow$  RAM

for (i = 1 .. n)

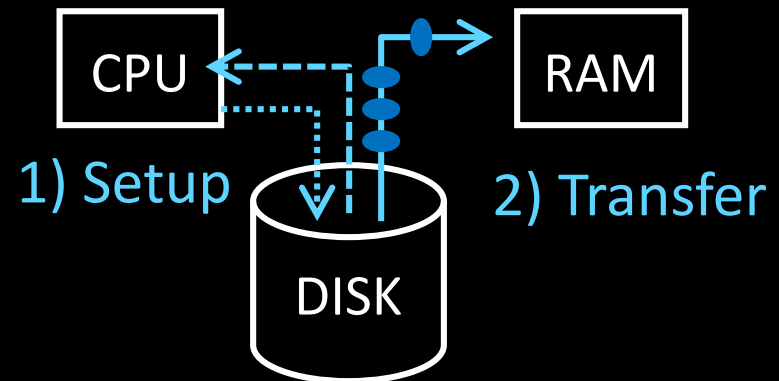
- CPU issues read request
- Device puts data on bus & CPU reads into registers
- CPU writes data to memory



3) Interrupt after done

DMA xfer: Device  $\leftrightarrow$  RAM

- CPU sets up DMA request
- for (i = 1 ... n)
  - Device puts data on bus & RAM accepts it
- Device interrupts CPU after done



# DMA Example

DMA example: reading from audio (mic) input

- DMA engine on audio device... or I/O controller ... or ...

```
int dma_size = 4*PAGE_SIZE;
int *buf = alloc_dma(dma_size);
...
dev->mic_dma_baseaddr = (int)buf;
dev->mic_dma_count = dma_len;
dev->cmd = DEV_MIC_INPUT |
DEV_INTERRUPT_ENABLE | DEV_DMA_ENABLE;
```

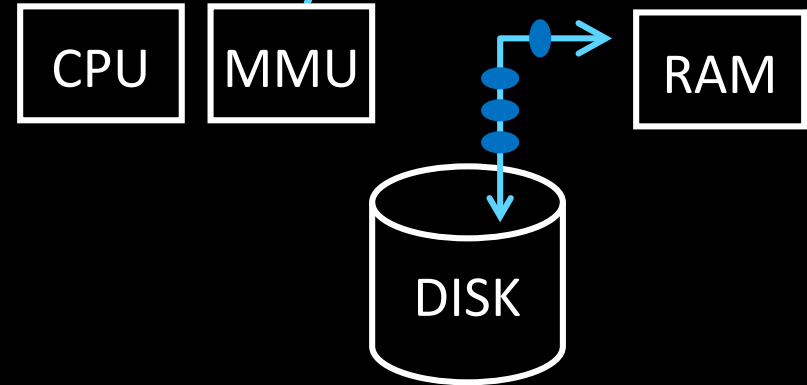


# DMA Issues (1): Addressing

## Issue #1: DMA meets Virtual Memory

RAM: physical addresses

Programs: virtual addresses



Solution: DMA uses physical addresses

- OS uses physical address when setting up DMA
- OS allocates contiguous physical pages for DMA
- Or: OS splits xfer into page-sized chunks  
(many devices support DMA “chains” for this reason)

# DMA Example

DMA example: reading from audio (mic) input

- DMA engine on audio device... or I/O controller ... or ...

```
int dma_size = 4*PAGE_SIZE;
```

```
void *buf = alloc_dma(dma_size);
```

```
...
```

```
dev->mic_dma_baseaddr = virt_to_phys(buf);
```

```
dev->mic_dma_count = dma_len;
```

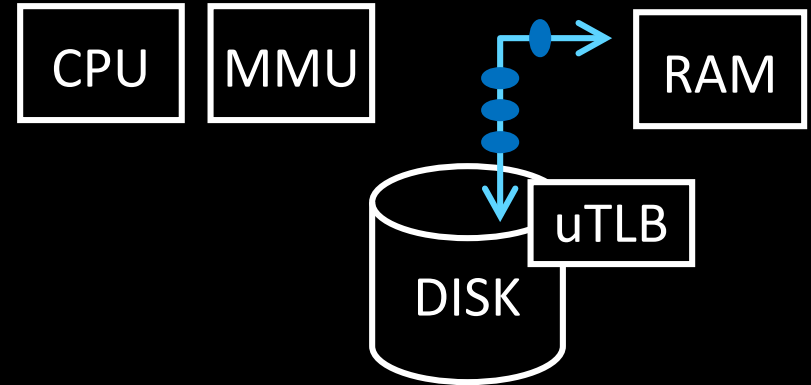
```
dev->cmd = DEV_MIC_INPUT |  
DEV_INTERRUPT_ENABLE | DEV_DMA_ENABLE;
```

# DMA Issues (1): Addressing

## Issue #1: DMA meets Virtual Memory

RAM: physical addresses

Programs: virtual addresses



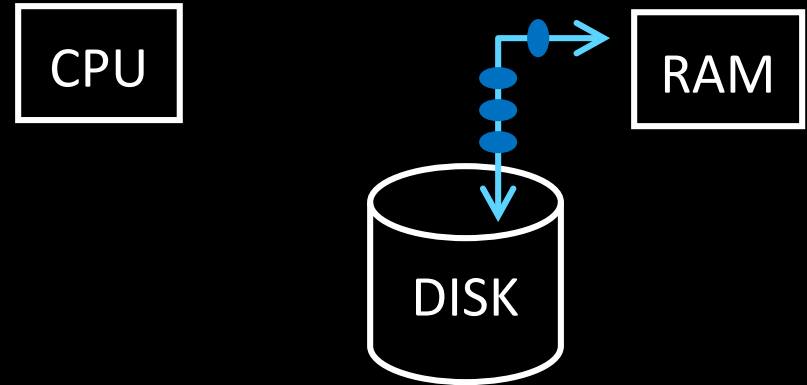
## Solution 2: DMA uses virtual addresses

- OS sets up mappings on a mini-TLB

# DMA Issues (2): Virtual Mem

Issue #2: DMA meets *Paged* Virtual Memory

DMA destination page  
may get swapped out



Solution: **Pin** the page before initiating DMA

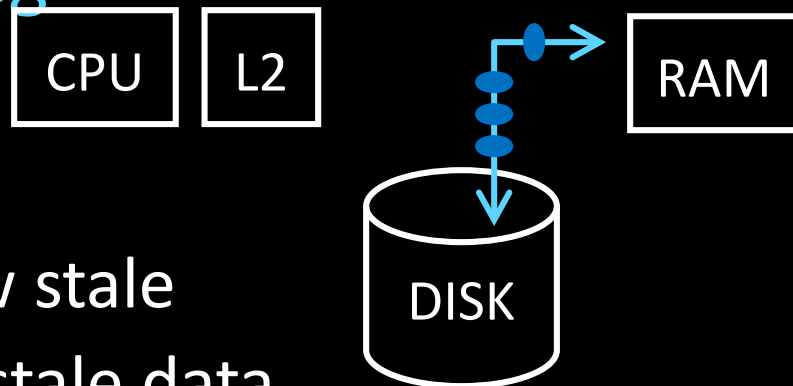
Alternate solution: **Bounce Buffer**

- DMA to a pinned kernel page, then memcpy elsewhere

# DMA Issues (4): Caches

## Issue #4: DMA meets Caching

DMA-related data could be cached in L1/L2



- DMA to Mem: cache is now stale
- DMA from Mem: dev gets stale data

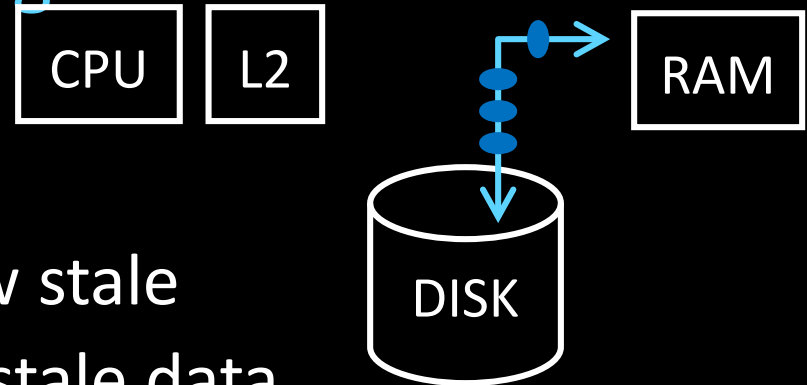
## Solution: (software enforced coherence)

- OS flushes some/all cache before DMA begins
- Or: don't touch pages during DMA
- Or: mark pages as **uncacheable** in page table entries
  - (needed for Memory Mapped I/O too!)

# DMA Issues (4): Caches

## Issue #4: DMA meets Caching

DMA-related data could be cached in L1/L2



- DMA to Mem: cache is now stale
- DMA from Mem: dev gets stale data

## Solution 2: (hardware coherence aka **snooping**)

- cache listens on bus, and conspires with RAM
- DMA to Mem: invalidate/update data seen on bus
- DMA from mem: cache services request if possible, otherwise RAM services

# Takeaways

Diverse I/O devices require hierarchical interconnect which is more recently transitioning to point-to-point topologies.

Memory-mapped I/O is an elegant technique to read/write device registers with standard load/stores.

Interrupt-based I/O avoids the wasted work in polling-based I/O and is usually more efficient.

Modern systems combine memory-mapped I/O, interrupt-based I/O, and direct-memory access to create sophisticated I/O device subsystems.

# I/O Summary

How to talk to device?

Programmed I/O or Memory-Mapped I/O

How to get events?

Polling or Interrupts

How to transfer lots of data?

DMA



# Announcements

Project3 submit “*souped up*” bot to CMS

**Project3 Cache Race Games night Monday, May 4<sup>th</sup>, 5pm**

- Come, eat, drink, have fun and be merry!
- Location: **B17 Upson Hall**

Prelim2: **Thursday**, April 30<sup>th</sup> in evening

- Time and Location: **7:30pm sharp** in **Statler Auditorium**
- Old prelims are online in CMS
- Prelim Review Session:  
**TODAY**, Tuesday, April 28, 7-9pm in B14 Hollister Hall

Project4:

- Design Doc due May 5<sup>th</sup>, bring design doc to mtg May 4-6
- Demos: May 12 and 13
- **Will not be able to use slip days**