ECE/CS 250
Computer Architecture

Fall 2022

I/O

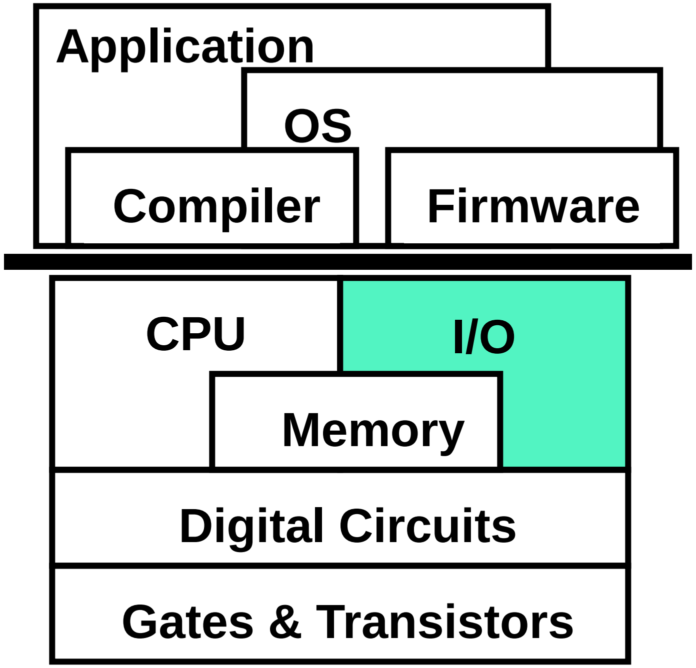
Tyler Bletsch
Duke University

Includes material adapted from Dan Sorin (Duke) and Amir Roth (Penn).
SSD material from Andrew Bondi (Colorado State).

Where We Are in This Course Right Now

- So far:
  - We know how to design a processor that can fetch, decode, and execute the instructions in an ISA
  - We understand how to design caches and memory
- Now:
  - We learn about the lowest level of storage (disks)
  - We learn about input/output in general
- Next:
  - Faster processor cores
  - Multicore processors

This Unit: I/O



- I/O system structure
  - Devices, controllers, and buses
- Device characteristics
  - Disks: HDD and SSD
- I/O control
  - Polling and interrupts
  - DMA

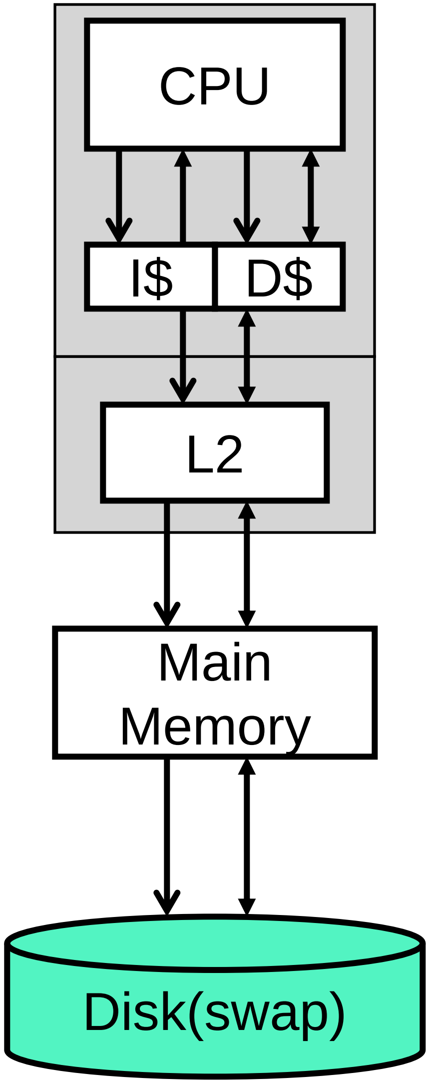
Readings

- Patterson and Hennessy dropped the ball on this topic
- It used to be covered in depth (in previous editions)
  - Now it’s sort of in Appendix A.8

Computers Interact with Outside World

- **Input/output (I/O)**
  - Otherwise, how will we ever tell a computer what to do…
  - …or exploit the results of its work?
- Computers without I/O are not useful
- ICQ: What kinds of I/O do computers have?

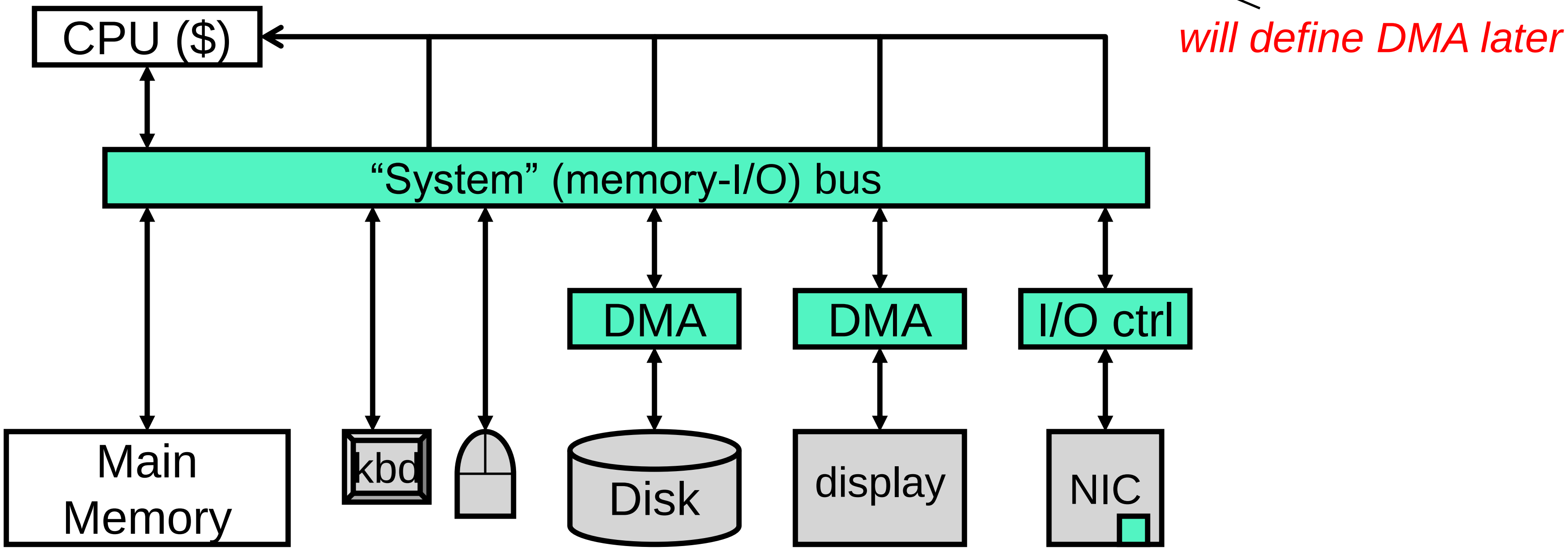
One Instance of I/O



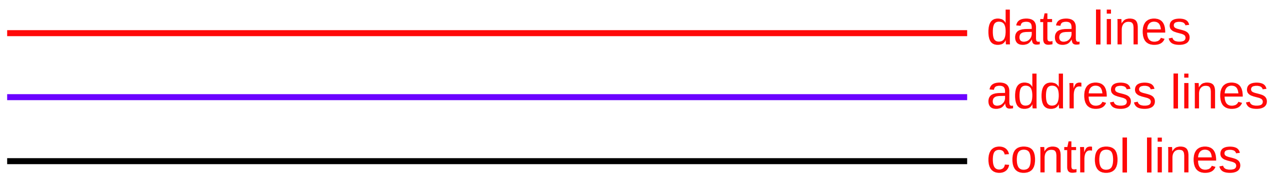
- Have briefly seen one instance of I/O
  - **Disk**: bottom of memory hierarchy
  - Holds whatever can't fit in memory
  - ICQ: What else do disks hold?

A More General/Realistic I/O System

- A computer system
  - CPU, including cache(s)
  - Memory (DRAM)
  - **I/O peripherals**: disks, input devices, displays, network cards, ...
    - With built-in or separate I/O (or DMA) controllers
  - All connected by a **system bus**



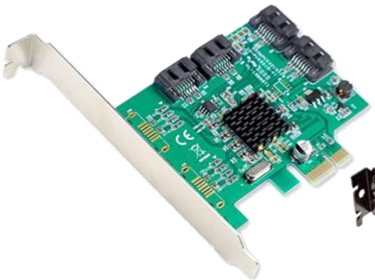
Bus Design



- Goals
  - **High Performance**: low latency and high bandwidth
  - **Standardization**: flexibility in dealing with many devices
  - **Low Cost**
    - Processor-memory bus emphasizes performance, then cost
    - I/O & backplane emphasize standardization, then performance
- Design issues
  1. **Width/multiplexing**: are wires shared or separate?
  2. **Clocking**: is bus clocked or not?
  3. **Switching**: how/when is bus control acquired and released?
  4. **Arbitration**: how do we decide who gets the bus next?

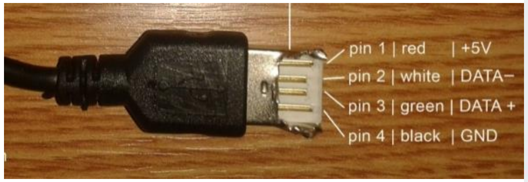
Standard Bus Examples

	PCIe	USB 2.0	USB 3.1
Type	Backplane	I/O	I/O
Width	1 bit per lane (1-16 lanes)	1 bit	4 bit
Multiplexed?	Yes	Yes	Yes
Clocking	2.5 – 8 GHz	Asynchronous	Asynchronous
Data rate	0.250 – 120 GB/s	60 MB/s	10 Gbit/s
Arbitration	Distributed	weird	weird
Maximum masters	255	127	127
Maximum length	~8 inches	–	–

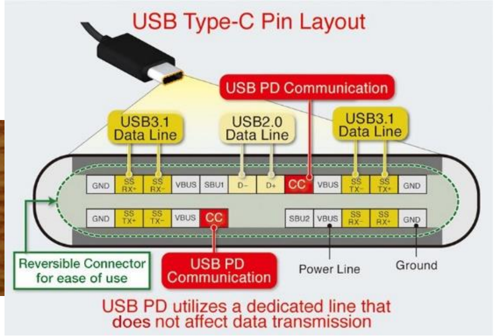


PCIe x1 SATA card

PCIe x16 GPU card

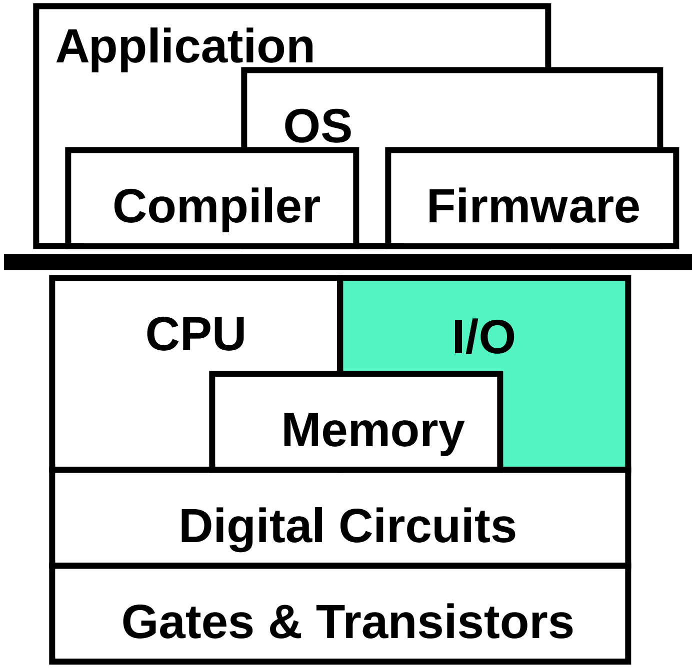


USB 2.0 ‘A’ plug



USB 3.1 ‘C’ plug

This Unit: I/O



- I/O system structure
  - Devices, controllers, and buses
- Device characteristics
  - Disks: HDD and SSD
- I/O control
  - Polling and interrupts
  - DMA

Operating System (OS) Plays a Big Role

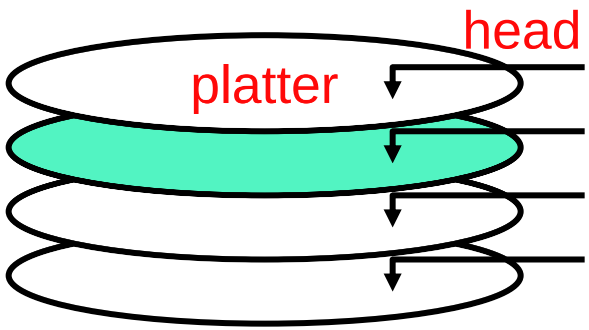
- I/O interface is typically under OS control
  - User applications access I/O devices indirectly (e.g., SYSCALL)
  - Why?
  - Device **drivers** are "programs" that OS uses to manage devices
- **Virtualization**: same argument as for memory
  - Physical devices shared among multiple programs
  - Direct access could lead to conflicts – example?
- **Synchronization**
  - Most have asynchronous interfaces, require unbounded waiting
  - OS handles asynchrony internally, presents synchronous interface
- **Standardization**
  - Devices of a certain type (disks) can/will have different interfaces
  - OS handles differences (via drivers), presents uniform interface

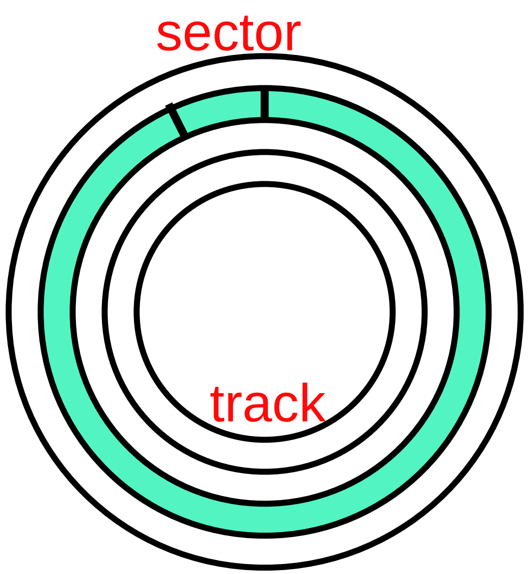
I/O Device Characteristics

- Primary characteristic
  - **Data rate** (aka **bandwidth**)
- Contributing factors
  - **Partner**: humans have slower output data rates than machines
  - **Input or output or both (input/output)**

Device	Partner	I? O?	Data Rate (KB/s)
Keyboard	Human	Input	0.01
Mouse	Human	Input	0.02
Speaker	Human	Output	0.60
Printer	Human	Output	200
Display	Human	Output	240,000
Modem (old)	Machine	I/O	7
Ethernet	Machine	I/O	~1,000,000
Disk	Machine	I/O	~50,000

I/O Device: Disk





- **Disk**: like stack of record players
- Collection of **platters**
  - Each with read/write head
- Platters divided into concentric **tracks**
  - Head seeks (forward/backward) to track
  - All heads move in unison
- Each track divided into **sectors**
  - ZBR (zone bit recording)
    - More sectors on outer tracks
  - Sectors rotate under head
- **Controller**
  - Seeks heads, waits for sectors
  - Turns heads on/off
  - May have its own cache (made w/DRAM)

Understanding disk performance

- One (clock) equals 1 microsecond
- Time to read the “next” 512-byte sector (no seek needed):
  ~2μs
- Time to read a random 512-byte sector (with seek):

~1840μs

Disk Bandwidth

- Disk is bandwidth-inefficient for page-sized transfers
  - Actual data transfer ($t_{transfer}$) a small part of disk access (and cycle)

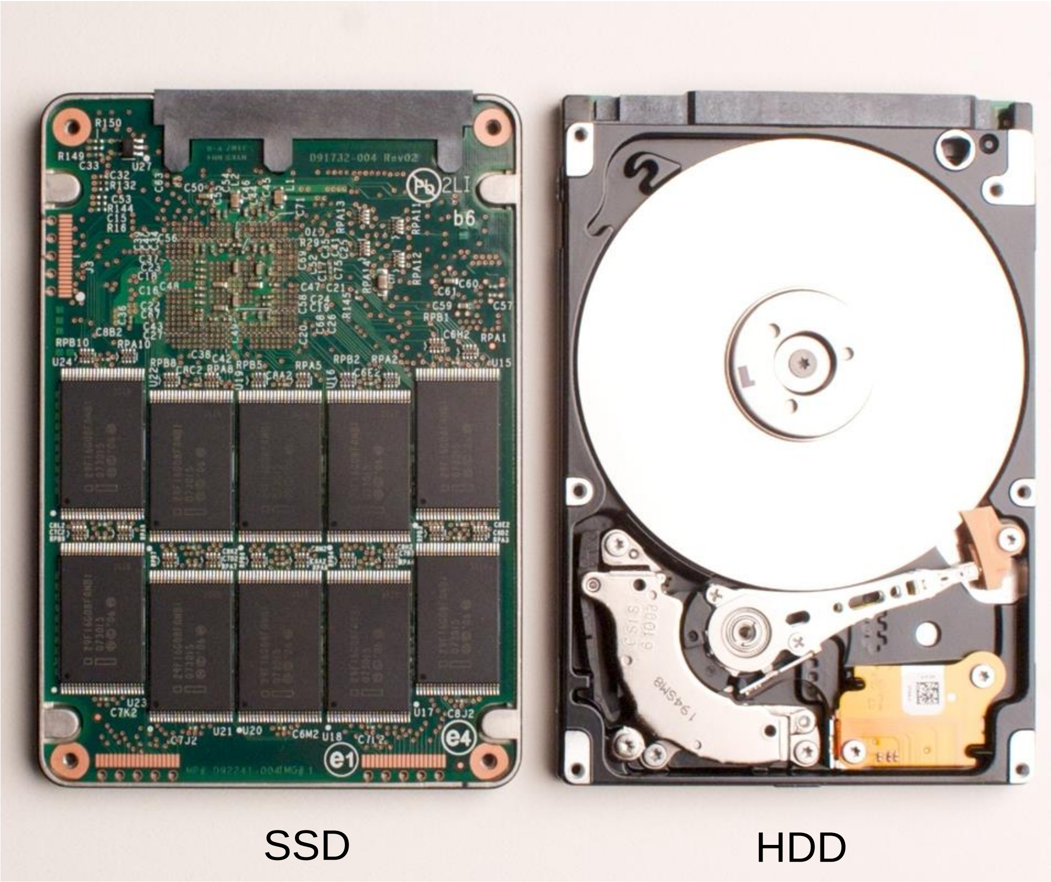
- Increase bandwidth: **stripe data across multiple disks**
  - Striping strategy depends on disk usage model
  - "File System" or "web server": many small files
    - Map entire files to disks
  - "Supercomputer" or "database": several large files
    - Stripe single file across multiple disks

- Both bandwidth and individual transaction latency important

Error Correction: RAID

- **Error correction**: more important for disk than for memory
  - Mechanical disk failures (entire disk lost) is common failure mode
  - Entire file system can be lost if files striped across multiple disks
- **RAID (redundant array of inexpensive disks)**
  - Similar to DRAM error correction, but…
  - Major difference: which disk failed is known
    - Even parity can be used to recover from single failures
    - Parity disk can be used to reconstruct data faulty disk
  - RAID design balances bandwidth and fault-tolerance
  - Many flavors of RAID exist
    - Tradeoff: extra disks (cost) vs. performance vs. reliability
    - Deeper discussion of RAID in ECE 552 and ECE 554; super-duper deep coverage in ECE 566 ("Enterprise Storage Architecture")
  - RAID doesn't solve all problems → can you think of any examples?

What about Solid State Drives (SSDs)?



Adapted from “Solid State Drives” by Andrew Bondi

SSDs

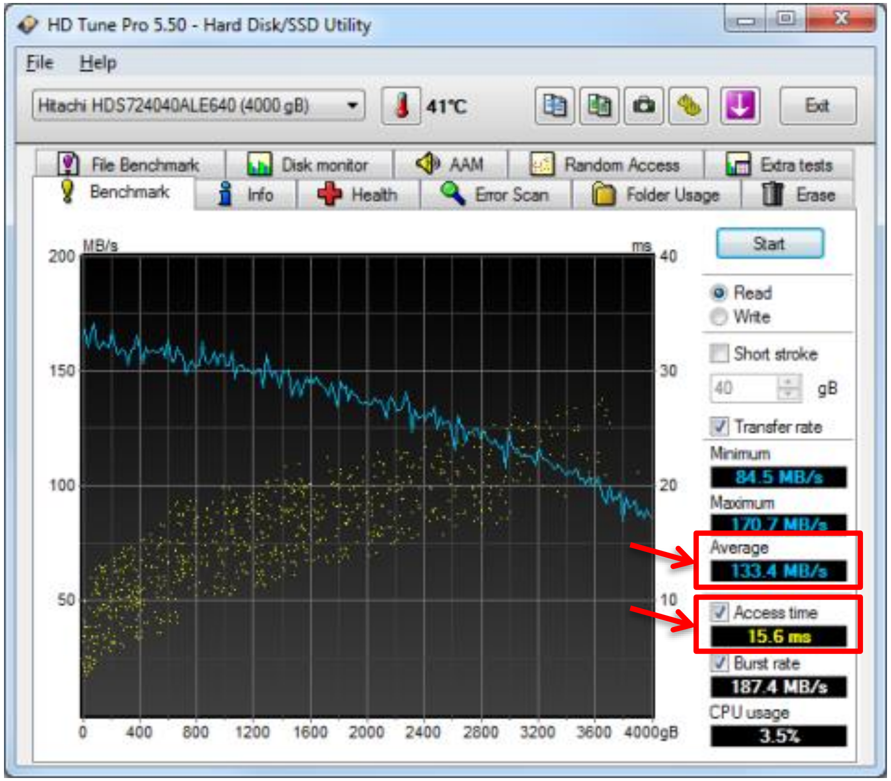
- Multiple NAND flash chips operated in parallel
- Pros:
  - Extremely good “seek” times (since “seek” is no longer a thing)
  - Almost instantaneous read and write times
  - The ability to read or write in multiple locations at once
  - The speed of the drive scales extremely well with the number of NAND ICs on board
  - Way cheaper than disk per IOP (performance)
- Cons:
  - Way more expensive than disk per GB (capacity)
  - Limited number of write cycles possible before it degrades (getting less and less of a problem these days)
  - Fundamental problem: Write amplification
    - You can *set* bits in “pages” (~4kB) fast (microseconds), but you can only *clear* bits in “blocks” (~512kB) slooow (milliseconds)
    - **Solution**: controller that is managing NAND cells tries to hide this

Adapted from “Solid State Drives” by Andrew Bondi

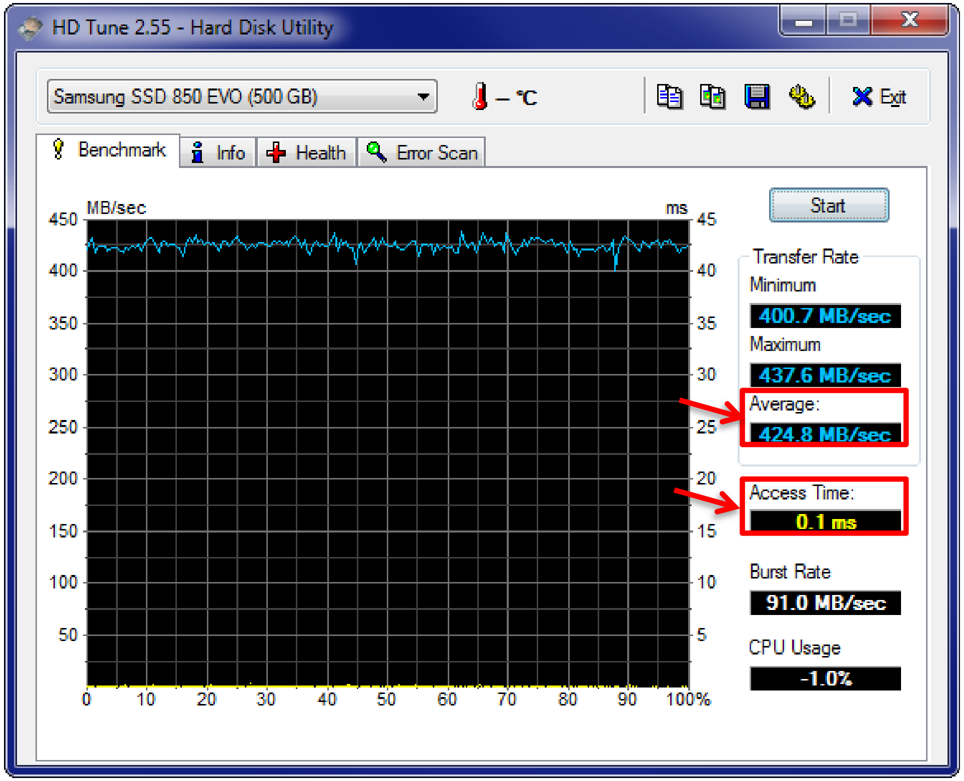
Typical read and write rates: SSD vs HDD

- Benchmark data from HD Tune (Windows benchmark)

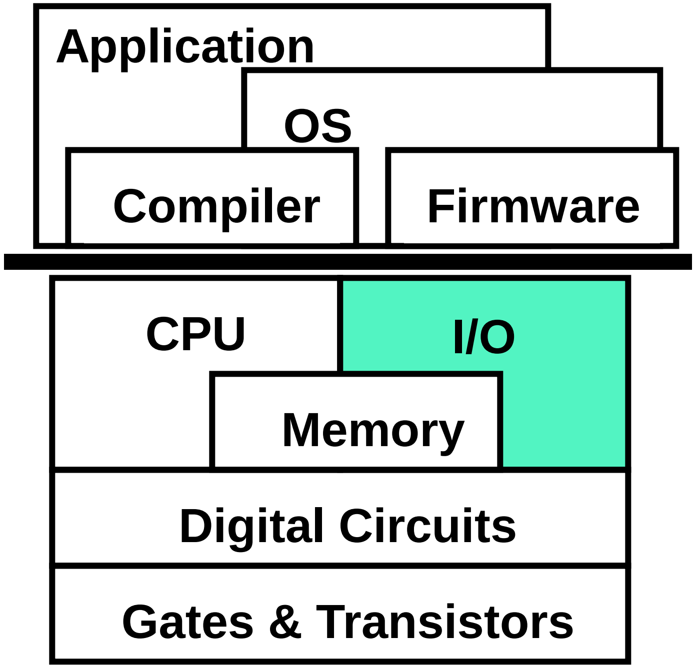
HDD



SSD



This Unit: I/O



- I/O system structure
  - Devices, controllers, and buses
- Device characteristics
  - Disks: HDD and SSD
- I/O control
  - Polling and interrupts
  - DMA

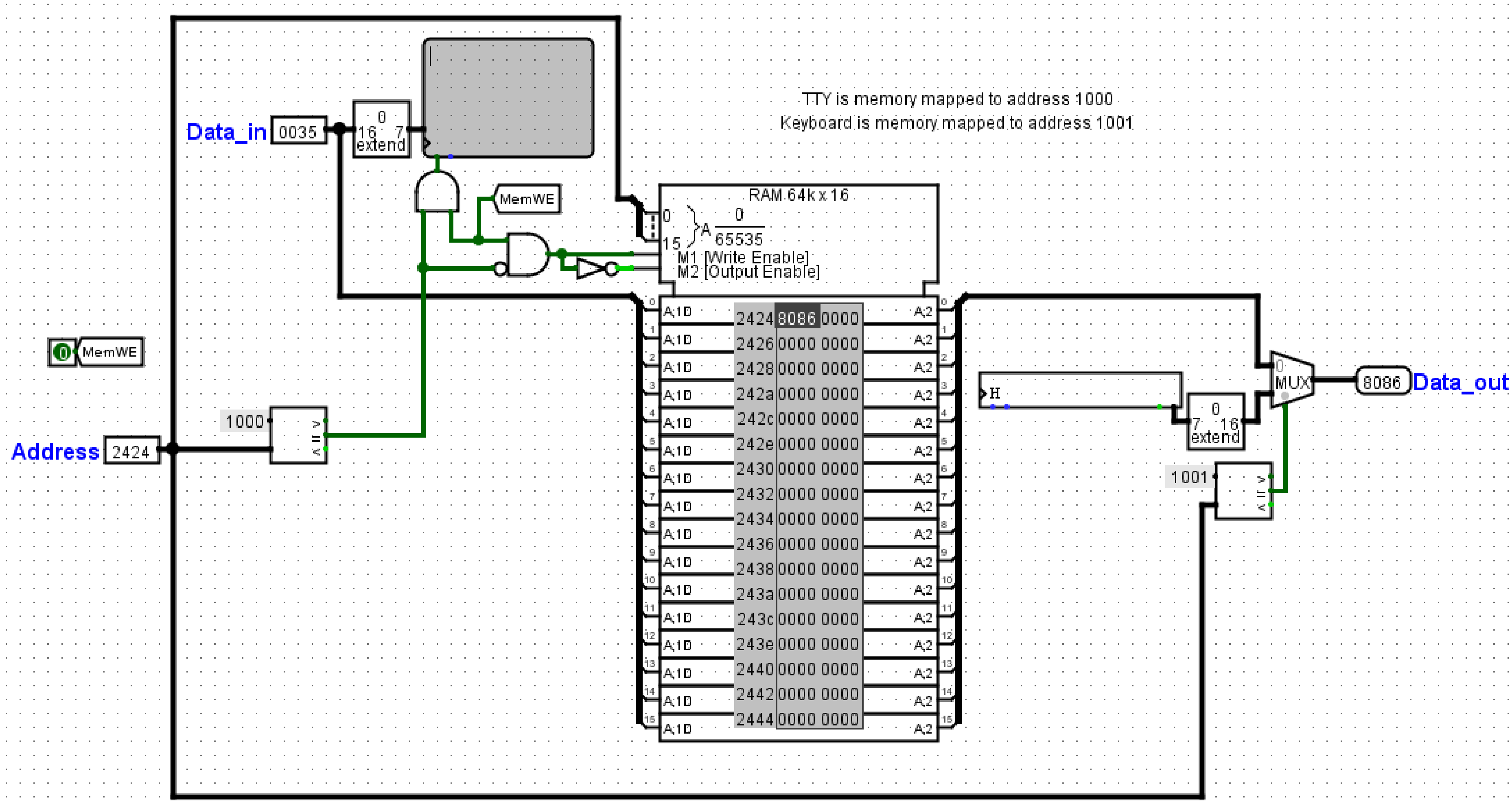
I/O Control and Interfaces

- Now that we know how I/O devices and buses work…
- How does I/O actually happen?
  - How does CPU give commands to I/O devices?
  - How do I/O devices execute data transfers?
  - How does CPU know when I/O devices are done?

Sending Commands to I/O Devices

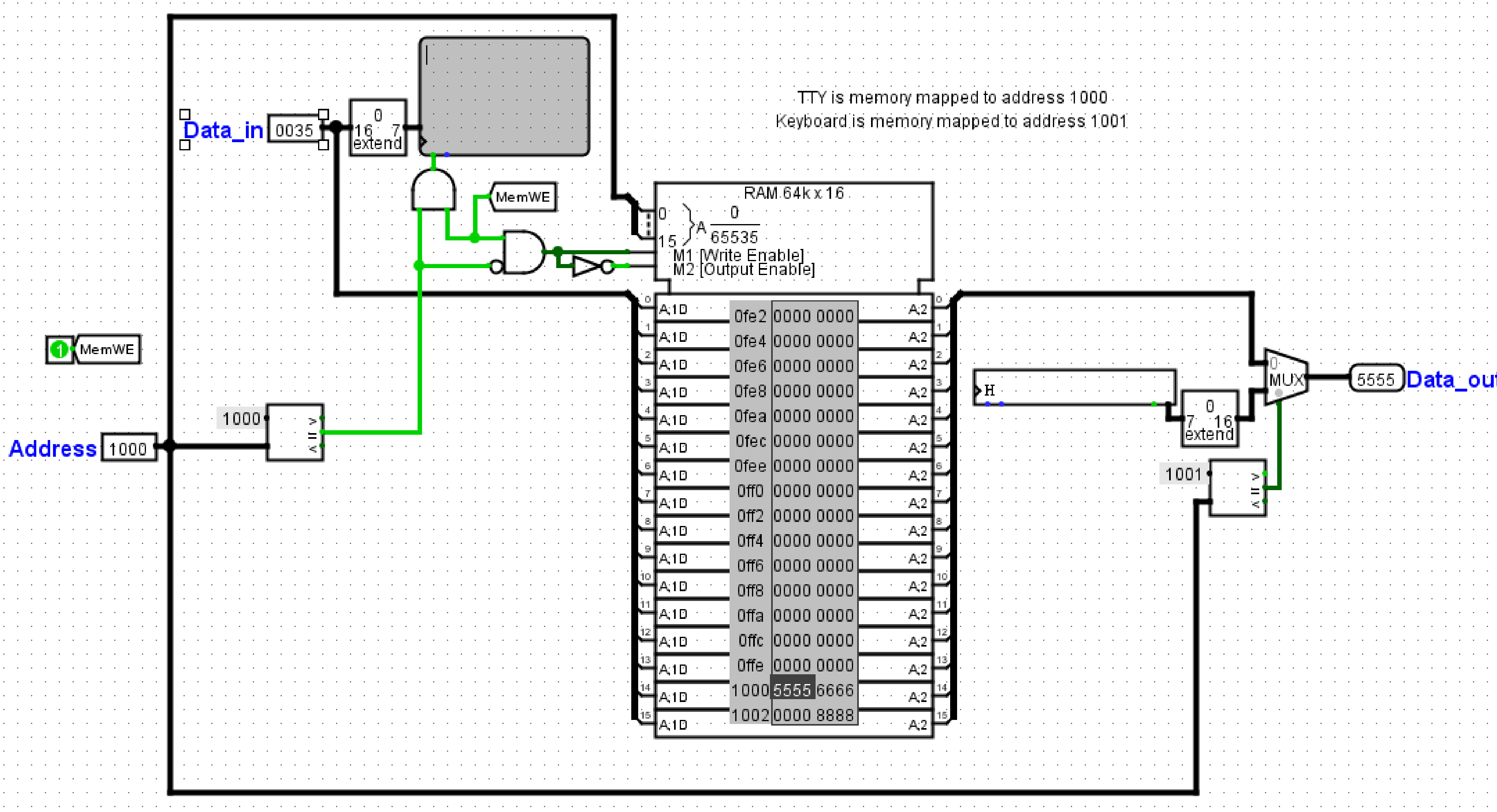
- Remember: only OS can do this! Two options …
- **I/O instructions**
  - OS only? Instructions must be privileged (only OS can execute)
  - E.g., IA-32
- **Memory-mapped I/O**
  - Portion of **physical** address space reserved for I/O
  - OS maps physical addresses to I/O device control registers
  - Stores/loads to these addresses are commands to I/O devices
    - Main memory ignores them, I/O devices recognize and respond
    - Address specifies both I/O device and command
  - These address are not cached – why?
  - OS only? I/O physical addresses only mapped in OS address space
  - E.g., almost every architecture other than IA-32 (see pattern??)

Memory mapped IO example (1)



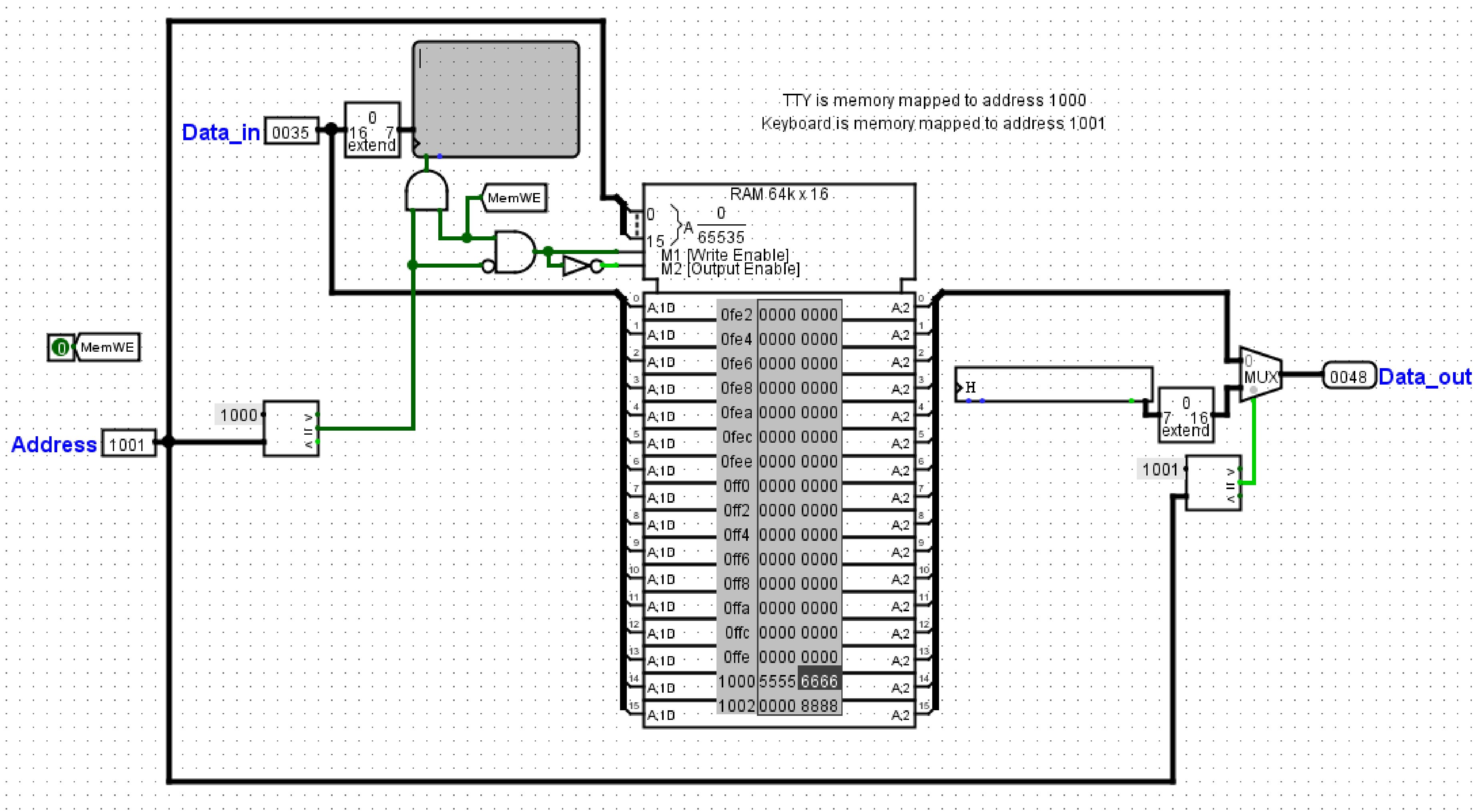
- Non-special read – comes from memory

Memory mapped IO example (2)



- Write to address 1000 – routed to TTY!
  - Mem write disabled, TTY write enabled; signal goes to both

Memory mapped IO example (3)



- Read from address 1001 – data comes from keyboard
  - Mux switches to keyboard for that address

Querying I/O Device Status

- Now that we've sent command to I/O device …
- How do we query I/O device status?
  - So that we know if data we asked for is ready?
  - So that we know if device is ready to receive next command?

- **Polling**: Ready now? How about now? How about now???
  - Processor queries I/O device status register (e.g., with MM load)
    - Loops until it gets status it wants (ready for next command)
    - Or tries again a little later

  \+ Simple

  – Waste of processor's time
    - Processor much faster than I/O device

Polling Overhead: Example #1

- Parameters
  - 500 MHz CPU
  - Polling event takes 400 cycles

- Overhead for polling a mouse 30 times per second?
  - Cycles per second for polling = (30 poll/s)*(400 cycles/poll)
  - → 12000 cycles/second for polling
  - (12000 cycles/second)/(500 M cycles/second) = 0.002% overhead
  - + Not bad

Polling Overhead: Example #2

- Same parameters
  - 500 MHz CPU, polling event takes 400 cycles

- Overhead for polling a 4 MB/s disk with 16 B interface?
  - Must poll often enough not to miss data from disk
  - Polling rate = (4MB/s)/(16 B/poll) >> mouse polling rate
  - Cycles per second for polling=[(4MB/s)/(16 B/poll)]*(400 cyc/poll)
  - → 100 M cycles/second for polling
  - (100 M cycles/second)/(500 M cycles/second) = 20% overhead
  - Bad
  - This is the overhead of polling, not actual data transfer
    - Really bad if disk is not being used (pure overhead!)

Interrupt-Driven I/O

- **Interrupts**: alternative to polling
  - I/O device generates interrupt when status changes, data ready
  - OS handles interrupts just like exceptions (e.g., page faults)
    - Identity of interrupting I/O device recorded in ECR
      - ECR: exception cause register
  - I/O interrupts are **asynchronous**
    - Not associated with any one instruction
    - Don't need to be handled immediately
  - I/O interrupts are **prioritized**
    - Synchronous interrupts (e.g., page faults) have highest priority
    - High-bandwidth I/O devices have higher priority than low-bandwidth ones

Interrupt Overhead

- Parameters
  - 500 MHz CPU
  - Polling event takes 400 cycles
  - Interrupt handler takes 400 cycles
  - Data transfer takes 100 cycles
  - 4 MB/s, 16 B interface disk, transfers data only 5% of time

Note: when disk is transferring data, the interrupt rate is same as polling rate

- Percent of time processor spends transferring data
  - **0.05** * (4 MB/s)/(16 B/xfer)*[(100 c/xfer)/(500M c/s)] = 0.25%
- Overhead for polling?
  - (4 MB/s)/(16 B/poll) * [(400 c/poll)/(500M c/s)] = 20%
- Overhead for interrupts?
  - \+ **0.05** * (4 MB/s)/(16 B/int) * [(400 c/int)/(500M c/s)] = 1%

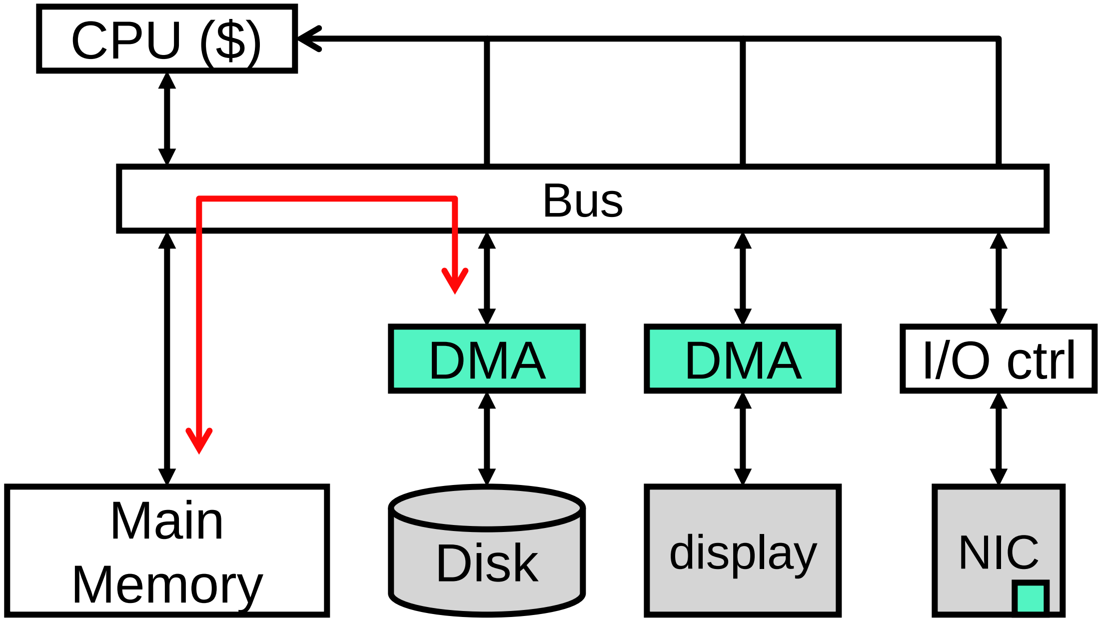
Direct Memory Access (DMA)

- Interrupts remove overhead of polling…
- But still requires OS to transfer data one word at a time
  - OK for low bandwidth I/O devices: mice, microphones, etc.
  - Bad for high bandwidth I/O devices: disks, monitors, etc.

- **Direct Memory Access (DMA)**
  - Transfer data between I/O and memory without processor control
  - Transfers entire blocks (e.g., pages, video frames) at a time
    - Can use bus "burst" transfer mode if available
  - Only interrupts processor when done (or if error occurs)

DMA Controllers

- To do DMA, I/O device attached to **DMA controller**
  - Multiple devices can be connected to one DMA controller
  - Controller itself seen as a memory mapped I/O device
    - Processor initializes start memory address, transfer size, etc.
  - DMA controller takes care of bus arbitration and transfer details
    - So that's why buses support arbitration and multiple masters!



DMA Overhead

- Parameters
  - 500 MHz CPU
  - Interrupt handler takes 400 cycles
  - Data transfer takes 100 cycles
  - 4 MB/s, 16 B interface, disk transfers data 50% of time
  - DMA setup takes 1600 cycles, transfer 1 16KB page at a time

- Processor overhead for interrupt-driven I/O?
  - 0.5 * (4M B/s)/(**16** B/xfer)*[(**500** c/xfer)/(500M c/s)] = 12.5%
- Processor overhead with DMA?
  - Processor only gets involved once per page, not once per 16 B
  - 0.5 * (4M B/s)/(**16K** B/page) * [(**2000** c/page)/(500M c/s)] = 0.05%

DMA and Memory Hierarchy

- DMA is good, but is not without challenges

- Without DMA: processor initiates all data transfers
  - All transfers go through address translation
    - + Transfers can be of any size and cross virtual page boundaries
  - All values seen by cache hierarchy
    - + Caches never contain stale data

- With DMA: DMA controllers initiate data transfers
  - Do they use virtual or physical addresses?
  - What if they write data to a cached memory location?

DMA and Caching

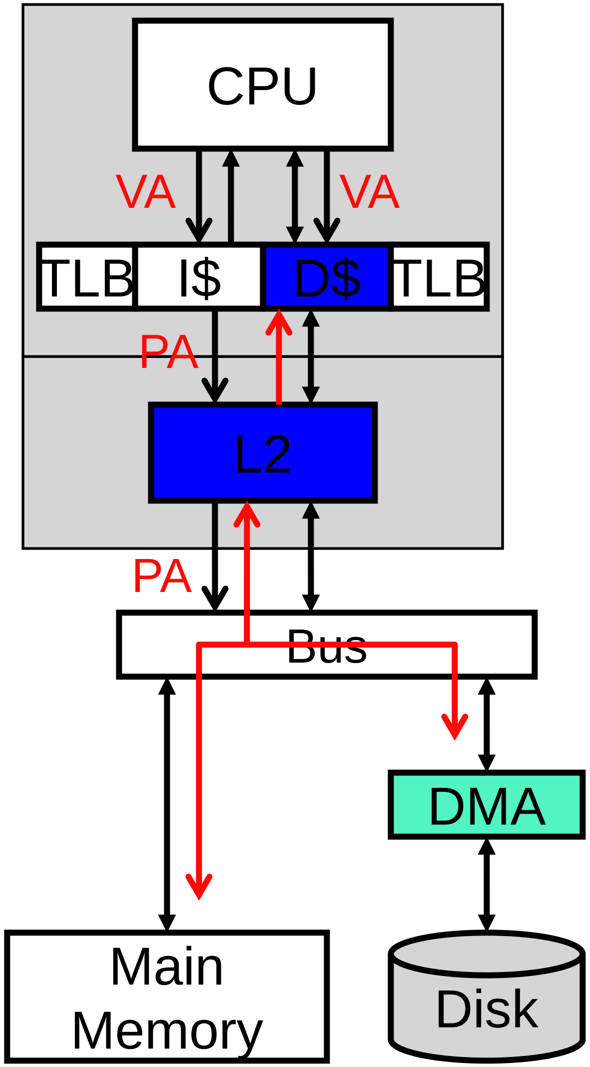
- Caches are good
  - Reduce CPU's observed instruction and data access latency
  + But also, reduce CPU's use of memory…
  + …leaving majority of memory/bus bandwidth for DMA I/O

- But they also introduce a **coherence problem** for DMA
  - Input problem
    - DMA write into memory version of cached location
    - Cached version now stale
  - Output problem: write-back caches only
    - DMA read from memory version of "dirty" cached location
    - Output stale value

Solutions to Coherence Problem

- Route all DMA I/O accesses to cache?
  - + Solves problem
  - – Expensive: CPU must contend for access to caches with DMA
- Disallow caching of I/O data?
  - + Also works
  - – Expensive in a different way: CPU access to those regions slow
- Selective flushing/invalidations of cached data
  - Flush all dirty blocks in "I/O region"
  - Invalidate blocks in "I/O region" as DMA writes those addresses
  - + The high performance solution
    - **Hardware cache coherence** mechanisms for doing this
  - – Expensive in yet a third way: must implement this mechanism

H/W Cache Coherence (more later on this)



- D$ and L2 **"snoop"** bus traffic
  - Observe transactions
  - Check if written addresses are resident
  - **Self-invalidate** those blocks

  \+ Doesn't require access to data part

  – Does require access to tag part
    - May need 2nd copy of tags for this
    - That's OK, tags smaller than data
- Bus addresses are physical
  - L2 is easy (physical index/tag)
  - D$ is harder (**virtual index**/physical tag)

Summary

- Storage devices
  - **HDD**: Mechanical disk. *Seeks are bad*. Cheaper per GB.
  - **SSD**: Flash storage. Cheaper per performance.
  - Can combine drives with **RAID** to get aggregate performance/capacity plus fault tolerance (can survive individual drive failures).
- Connectivity
  - A **bus** is shared between CPU, memory, and/or and multiple IO devices
- How does CPU talk to IO devices?
  - **Special instructions** or **memory-mapped IO** (certain addresses don't lead to RAM, they lead to IO devices)
    - Either requires OS privilege to use
  - Methods of interaction:
    - **Polling** (simple but wastes CPU)
    - **Interrupts** (saves CPU but transfers tiny bit at a time)
    - **DMA**+interrupts (saves CPU+fast, but requires caches to snoop traffic to not become wrong)

EXTRA MATERIAL

Disk Parameters

	Seagate 6TB Enterprise HDD (2016)	Seagate Savvio (~2005)	Toshiba MK1003 (early 2000s)
Diameter	3.5″	2.5″	1.8″
Capacity	6 TB	73 GB	10 GB
RPM	7200 RPM	10000 RPM	4200 RPM
Cache	128 MB	8 MB	512 KB
Platters	~6	2	1
Average Seek	4.16 ms	4.5 ms	7 ms
Sustained Data Rate	216 MB/s	94 MB/s	16 MB/s
Interface	SAS/SATA	SCSI	ATA
Use	Desktop	Laptop	Ancient iPod

Density improving

Caches improving

Seek time not really improving!

Disk Read/Write Latency

- Disk read/write latency has four components
  - **Seek delay (t_{seek})**: head seeks to right track
    - Fixed delay plus proportional to distance
  - **Rotational delay ($t_{rotation}$)**: right sector rotates under head
    - Fixed delay on average (average = half rotation)
  - **Controller delay ($t_{controller}$)**: controller overhead (on either side)
    - Fixed cost
  - **Transfer time ($t_{transfer}$)**: data actually being transferred
    - Proportional to amount of data