EE108B
Lecture 18
Buses and OS, Review, & What's Next

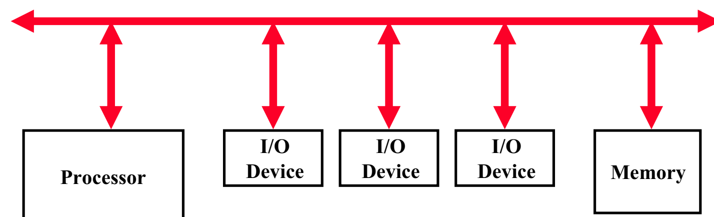
Christos Kozyrakis
Stanford University
http://eeclass.stanford.edu/ee108b

Announcements

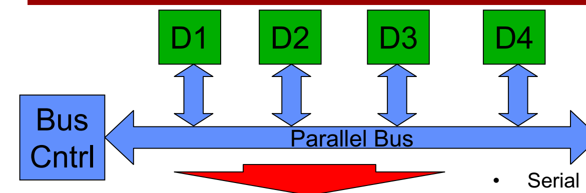
- TBD

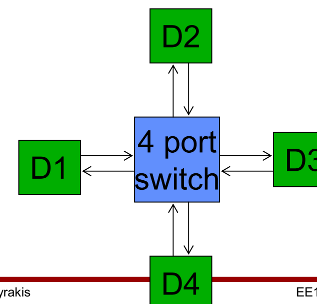
Review: Buses

- A bus is a shared communication link that connects multiple devices
- Single set of wires connects multiple "subsystems" as opposed to a point to point link which only connects two components together
- Wires connect in parallel, so 32 bit bus has 32 wires of data



Review: Trends for Buses Logical Bus and Physical Switch





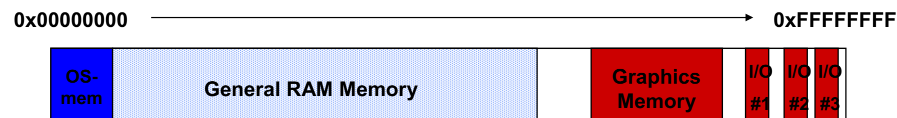
- Serial point-to-point advantages
  - Faster links
  - Fewer chip package pins
  - Higher performance
  - Switch keeps arbitration on chip
- Many bus standards are moving to serial, point to point
- 3GIO, PCI-Express(PCI)
- Serial ATA (IDE hard disk)
- AMD Hypertransport versus Intel Front Side Bus (FSB)

Review: OS Communication

- The operating system should prevent user programs from communicating with I/O device directly
  - Must protect I/O resources to keep sharing fair
  - Protection of shared I/O resources cannot be provided if user programs could perform I/O directly
- Three types of communication are required:
  - OS must be able to give commands to I/O devices
  - I/O device must be able to notify OS when I/O device has completed an operation or has encountered an error
  - Data must be transferred between memory and an I/O device

Reivew: A method for addressing a device

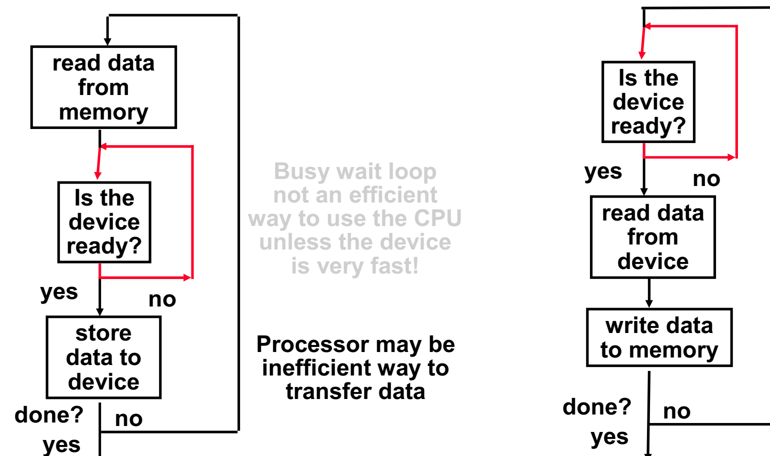
- Memory-mapped I/O:
  - Portions of the address space are assigned to each I/O device
  - I/O addresses correspond to device registers
  - User programs prevented form issuing I/O operations directly since I/O address space is *protected* by the address translation mechanism



Communicating with the CPU

- Method #1: Polling
  - I/O device places information in a status register
  - The OS periodically checks the status register
  - Whether polling is used is often dependent upon whether the device can initiate I/O independently
    - For instance, a mouse works well since it has a fixed I/O rate and initiates its own data (whenever it is moved)
    - For others, such as disk access, I/O only occurs under the control of the OS, so we poll only when the OS knows it is active
  - Advantages
    - Simple to implement
    - Processor is in control and does the work
  - Disadvantage
    - Polling overhead and data transfer consume CPU time

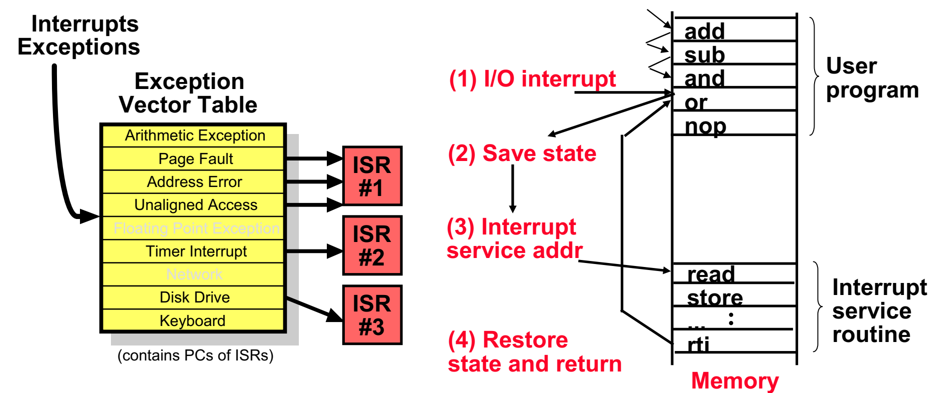
Polling and Programmed I/O



I/O Notification (cont)

- Method #2: I/O Interrupt
  - Whenever an I/O device needs attention from the processor, it *interrupts* the processor
  - Interrupt must tell OS about the event and which device
    - Using “cause” register(s): Kernel “asks” what interrupted
    - Using *vectored* interrupts: A different exception handler for each
    - Example: Intel 80x86 has 256 vectored interrupts
  - I/O interrupts are asynchronous events, and happen anytime
    - Processor waits until current instruction is completed
  - Interrupts may have different priorities
    - Ex.: Network = high priority, keyboard = low priority
  - Advantage: Execution is only halted during actual transfer
  - Disadvantage: Software overhead of interrupt processing

I/O Interrupts



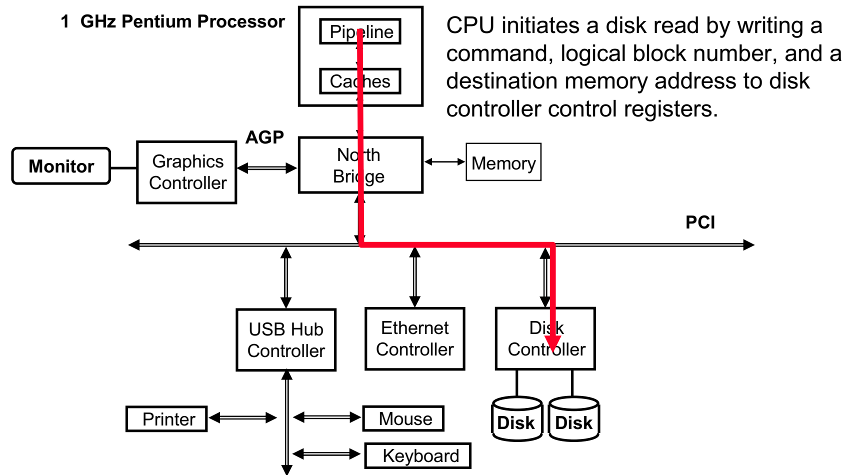
Data Transfer

- The third component to I/O communication is the transfer of data from the I/O device to memory (or vice versa)
- Simple approach: “Programmed” I/O
  - Software on the processor moves *all* data between memory addresses and I/O addresses
  - Simple and flexible, but wastes CPU time
  - Also, lots of excess data movement in modern systems
    - Ex.: Mem --> NB --> CPU --> NB --> graphics
    - When we want: Mem --> NB --> graphics
- So need a solution to allow data transfer to happen *without* the processor’s involvement

Delegating I/O: DMA

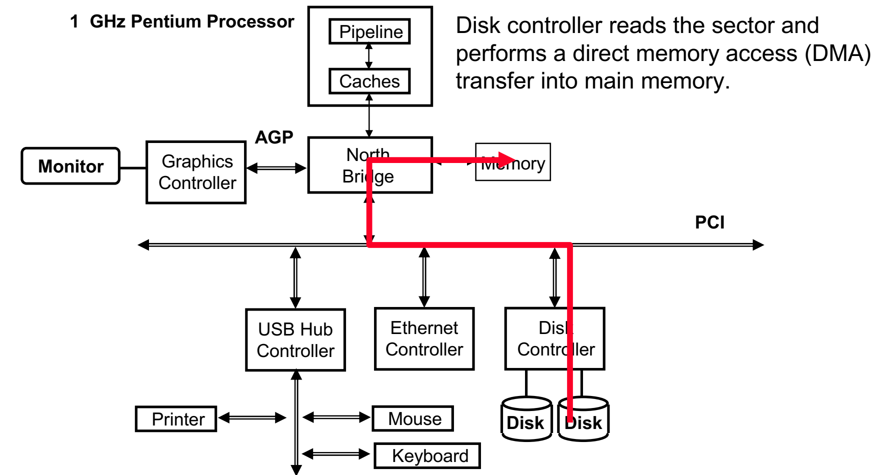
- Direct Memory Access (DMA)
  - Transfer *blocks* of data to or from memory without CPU intervention
  - Communication coordinated by the *DMA controller*
    - DMA controllers are integrated in memory or I/O controller chips
  - DMA controller acts as a bus master, in bus-based systems
- DMA Steps
  - Processor sets up DMA by supplying:
    - Identity of the device and the operation (read/write)
    - The memory address for source/destination
    - The number of bytes to transfer
  - DMA controller starts the operation by arbitrating for the bus and then starting the transfer when the data is ready
  - Notify the processor when the DMA transfer is complete or on error
    - Usually using an interrupt

Reading a Disk Sector (1)



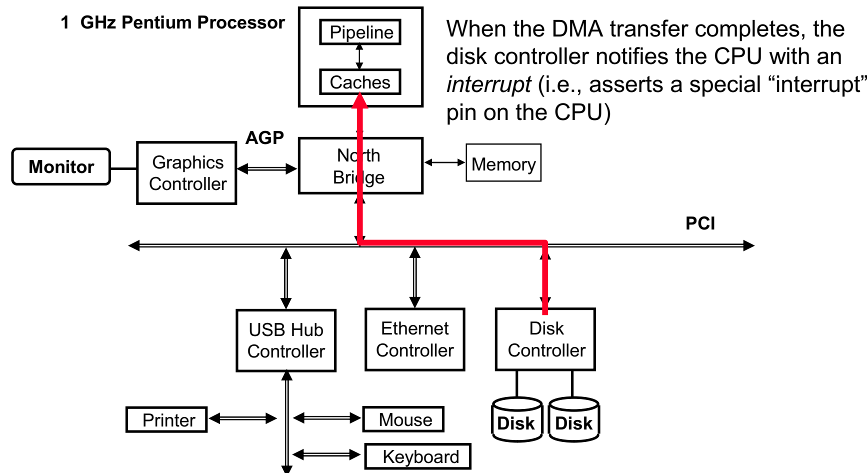
CPU initiates a disk read by writing a command, logical block number, and a destination memory address to disk controller control registers.

Reading a Disk Sector (2)



Disk controller reads the sector and performs a direct memory access (DMA) transfer into main memory.

Reading a Disk Sector (3)



When the DMA transfer completes, the disk controller notifies the CPU with an *interrupt* (i.e., asserts a special "interrupt" pin on the CPU)

DMA Problems: Virtual Vs. Physical Addresses

- If DMA uses physical addresses
  - Memory access across physical page boundaries may not correspond to contiguous virtual pages (or even the same application!)
- Solution 1: ≤ 1 page per DMA transfer
- Solution 1+: chain a series of 1-page requests provided by the OS
  - Single interrupt at the end of the last DMA request in the chain
- Solution 2: DMA engine uses virtual addresses
  - Multi-page DMA requests are now easy
  - A TLB is necessary for the DMA engine
- For DMA with physical addresses: pages must be pinned in DRAM
  - OS should not page to disks pages involved with pending I/O

DMA Problems: Cache Coherence

- A copy of the data involved in a DMA transfer may reside in processor cache
  - If memory is updated: Must update or invalidate "old" cached copy
  - If memory is read: Must read latest value, which may be in the cache
    - Only a problem with write-back caches
- This is called the "cache coherence" problem
  - Same problem in multiprocessor systems

- Solution 1: OS flushes the cache before I/O reads or forces writebacks before I/O writes
  - Flush/write-back may involve selective addresses or whole cache
  - Can be done in software or with hardware (ISA) support
- Solution 2: Route memory accesses for I/O through the cache
  - Search the cache for copies and invalidate or write-back as needed
  - This hardware solution may impact performance negatively
    - While searching cache for I/O requests, it is not available to processor
  - Multi-level, inclusive caches make this easier
    - Processor searches L1 cache mostly (until it misses)
    - I/O requests search L2 cache mostly (until it finds a copy of interest)

I/O Summary

- I/O performance has to take into account many variables
  - Response time and throughput
  - CPU, memory, bus, I/O device
  - Workload e.g. transaction processing
- I/O devices also span a wide spectrum
  - Disks, graphics, and networks
- Buses
  - bandwidth
  - synchronization
  - transactions
- OS and I/O
  - Communication: Polling and Interrupts
  - Handling I/O outside CPU: DMA

EE108B Quick Review (1)

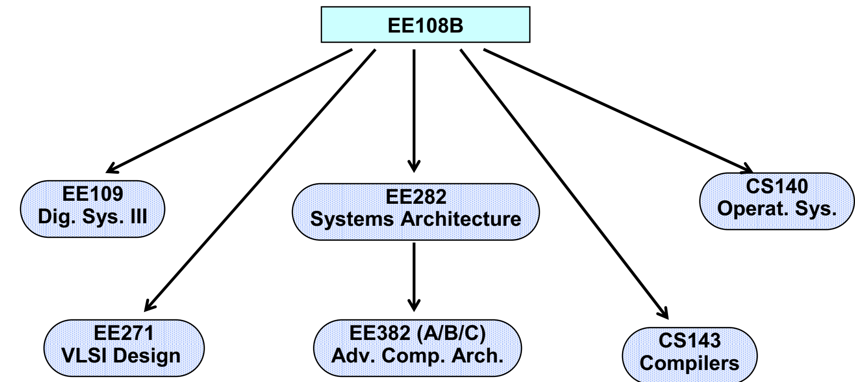
- What we learned in EE108b
  - How to build programmable systems
    - Processor, memory system, IO system, interactions with OS and compiler
  - Processor design
    - ISA, single cycle design, pipelined design, …
  - The basics of computer system design
    - Levels of abstraction, pipelining, caching, address indirection, DMA, …
  - Understanding why your programs sometimes run slowly
    - Pipeline stalls, cache misses, page faults, IO accesses, …

EE108B Quick Review (2)
Major Lessons to Take Away

- Levels of abstraction (e.g. ISA→processor→RTL blocks→gates)
  - Simplifies design process for complex systems
  - Need good specification for each level
- Pipelining
  - Improve throughput of an engine by overlapping tasks
  - Watch out for dependencies between tasks
- Caching
  - Maintain a close copy of frequently accessed data
  - Avoid long accesses or expensive recomputation (memoization)
  - Think about associativity, replacement policy, block size
- Indirection (e.g. virtual→physical address translation)
  - Allows transparent relocation, sharing, and protection
- Overlapping (e.g. CPU work & DMA access)
  - Hide the cost of expensive tasks by executing them in parallel with other useful work
- Other:
  - Amdhal's law: make common case fast
  - Amortization, oarallel processing, memoization, speculation

The End

After EE108B

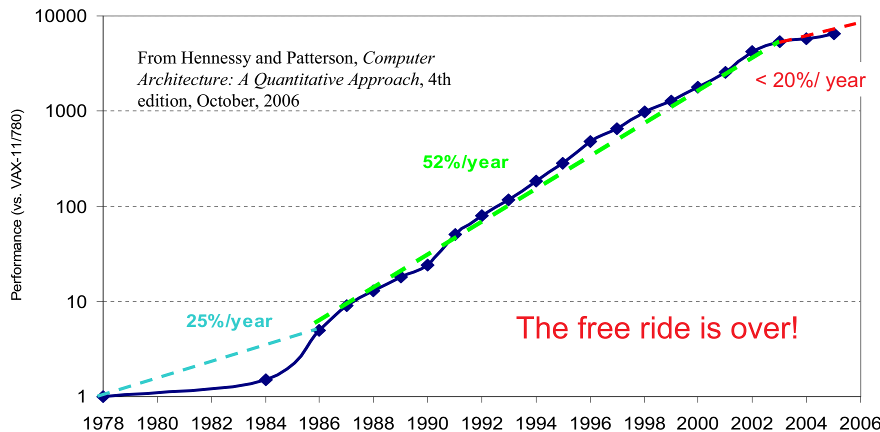


EE282 Spring 07: Computer Systems Architecture

- Goal: learn how to build and use efficient systems
- Topics:
  - Advanced memory hierarchies
  - Advanced I/O and interactions with OS
  - Cluster architecture and programming
  - Virtual machines
  - Reliable systems
  - Energy efficient systems
- Programming assignments
  - Efficient programming for uniprocessors
  - Efficient programming for clusters

Looking Forward

End of Uniprocessor Performance



The World has Changed

- Process Technology Stops Improving
  - Moore's law but …
  - Transistors don't get faster and they leak more (65nm vs. 45nm)
  - Wires are much worse
- Single Thread Performance Plateau
  - Design and verification complexity is overwhelming
  - Power consumption increasing dramatically
  - Instruction-level parallelism (ILP) has been mined out

The Right Hand Turn:
- Move away from frequency as performance
- Multi– everywhere; MT, CMP

Intel Developer FORUM

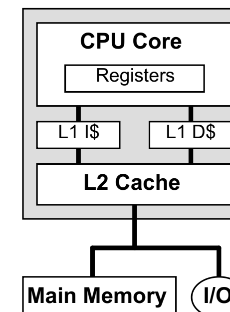
From Intel Developer Forum, September 2004

The Era of Single-Chip Multiprocessors

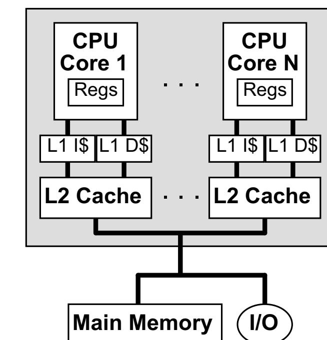
- Single-chip multiprocessors provide a scalable alternative
  - Relies on scalable forms of parallelism
    - Request level parallelism
    - Data level parallelism
  - Modular design with inherent fault-tolerance and match to VLSI technology
- Single-chip multiprocessors systems are here
  - All processor vendors are following this approach
  - In embedded, server, and even desktop systems
- How do we architect CMPs to best exploit thread-level parallelism?
  - Server applications: throughput
  - General purpose and scientific applications: latency

CMP Options

a) Conventional microprocessor



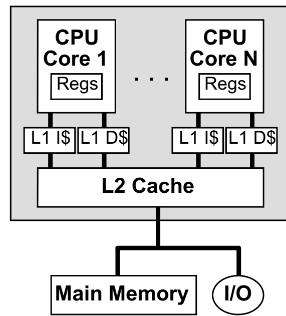
b) Simple chip multiprocessor



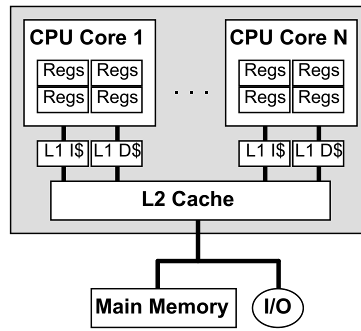
e.g. AMD Dual-core Opteron
Intel Paxville

CMP Options

c) Shared-cache chip multiprocessor



d) Multithreaded, shared-cache chip multiprocess



Multiprocessor Questions

- How do parallel processors share data?
  - single address space (SMP, NUMA)
  - message passing (clusters, massively parallel processors (MPP))
- How do parallel processors coordinate?
  - software synchronization (locks, semaphores)
  - built into send / receive primitives
    OS primitives (sockets)
- How are parallel processors interconnected?
  - connected by a single bus
  - connected by a network