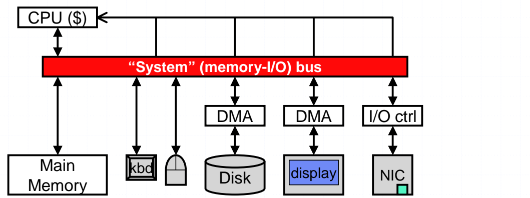
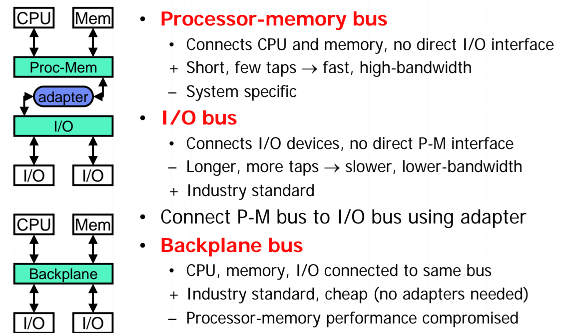
The System Bus

- **System bus**: connects system components together
  - Important: insufficient bandwidth can bottleneck entire system
  - Performance factors
    - Physical length
    - Number and type of connected devices (taps)



Three Buses



- **Processor-memory bus**
  - Connects CPU and memory, no direct I/O interface
  - \+ Short, few taps → fast, high-bandwidth
  - – System specific
- **I/O bus**
  - Connects I/O devices, no direct P-M interface
  - – Longer, more taps → slower, lower-bandwidth
  - \+ Industry standard
- Connect P-M bus to I/O bus using adapter
- **Backplane bus**
  - CPU, memory, I/O connected to same bus
  - \+ Industry standard, cheap (no adapters needed)
  - – Processor-memory performance compromised

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?

(1) Bus Width and Multiplexing

- **Wider**
  - \+ More bandwidth
  - – More expensive and more susceptible to **skew**
- **Multiplexed**: address and data share same lines
  - \+ Cheaper
  - – Less bandwidth
- Burst transfers
  - Multiple sequential data transactions for single address
  - \+ Increase bandwidth at relatively little cost

(2) Bus Clocking

- **Synchronous**: clocked
  - \+ Fast
  - – Bus must be short to minimize clock skew
- **Asynchronous**: un-clocked
  - \+ Can be longer: no clock skew, deals with devices of different speeds
  - – Slower: requires "hand-shaking" protocol
    - For example, asynchronous read
      - Multiplexed data/address lines, 3 control lines
      1. Processor drives address onto bus, asserts `Request` line
      2. Memory asserts `Ack` line, processor stops driving
      3. Memory drives data on bus, asserts `DataReady` line
      4. Processor asserts `Ack` line, memory stops driving
  - P-M buses are synchronous
  - I/O and backplane buses asynchronous or slow-clock synchronous

(3) Bus Switching

- **Atomic**: bus "busy" between request and reply
  - \+ Simple
  - – Low utilization
- **Split-transaction**: requests/replies can be interleaved
  - \+ Higher utilization → higher throughput
  - – Complex, requires sending IDs to match replies to request

(4) Bus Arbitration

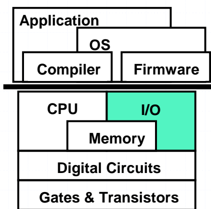
- **Bus master**: component that can initiate a bus request
  - Bus typically has several masters, including processor
  - I/O devices can also be masters (Why? See in a bit)
- **Arbitration**: choosing a master among multiple requests
  - Try to implement **priority** and **fairness** (no device "starves")
  - **Daisy-chain**: devices connect to bus in priority order
    - High-priority devices intercept/deny requests by low-priority ones
    - ± Simple, but slow and can't ensure fairness
  - **Centralized**: special arbiter chip collects requests, decides
    - ± Ensures fairness, but arbiter chip may itself become bottleneck
  - **Distributed**: everyone sees all requests simultaneously
    - Back off and retry if not the highest priority request
    - ± No bottlenecks and fair, but needs a lot of control lines

Standard Bus Examples

	PCI	SCSI	USB
Type	Backplane	I/O	I/O
Width	32–64 bits	8–32 bits	1
Multiplexed?	Yes	Yes	Yes
Clocking	33 (66) MHz	5 (10) MHz	Asynchronous
Data rate	133 (266) MB/s	10 (20) MB/s	0.2, 1.5, 80 MB/s
Arbitration	Distributed	Distributed	Daisy-chain
Maximum masters	1024	7–31	127
Maximum length	0.5 m	2.5 m	–

- **USB (universal serial bus)**
  - Popular for low/moderate bandwidth external peripherals
  - \+ Packetized interface (like TCP), extremely flexible
  - \+ Also supplies power to the peripheral

This Unit: I/O



- I/O system structure
  - Devices, controllers, and buses
- Device characteristics
  - Disks
- Bus characteristics
- 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??)

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

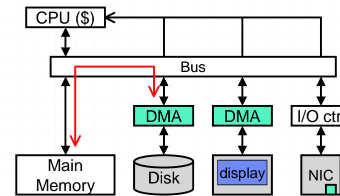
- Percent of time processor spends transferring data
  - **0.05** * (4 MB/s)/(16 B/xfer)*[(100 c/xfer)/(500M c/s)] = 0.25%
- Overhead for polling?
  - (4 MB/s)/(16 B/poll) * [(400 c/poll)/(500M c/s)] = 20%
- Overhead for interrupts?
  - + **0.05** * (4 MB/s)/(16 B/int) * [(400 c/int)/(500M c/s)] = 1%

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)

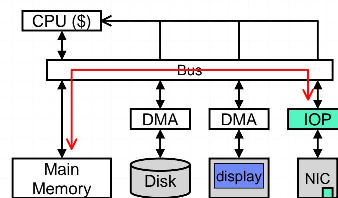
DMA Controllers

- 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!



I/O Processors

- A DMA controller is a very simple component
  - May be as simple as a FSM with some local memory
- Some I/O requires complicated sequences of transfers
  - **I/O processor**: heavier DMA controller that executes instructions
    - Can be programmed to do complex transfers
    - E.g., programmable network card



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 Address Translation

- Which addresses does processor specify to DMA controller?
- **Virtual DMA**
  - + Can specify large cross-page transfers
  - – DMA controller has to do address translation internally
    - DMA contains small translation buffer (TB)
    - OS initializes buffer contents when it requests an I/O transfer
- **Physical DMA**
  - + DMA controller is simple
  - – Can only do short page-size transfers
    - OS breaks large transfers into page-size chunks

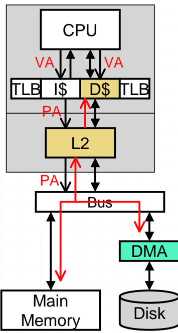
DMA and Caching

- 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 (see ECE 252, 259)



- 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)
    - Reverse translation? No
    - Remember: page size vs. D$ size

Designing an I/O System for Bandwidth

- Approach
  - Find bandwidths of individual components
  - Configure components you can change…
  - To match bandwidth of bottleneck component you can't
- Example (from P&H textbook)
  - Parameters
    - 300 MIPS CPU, 100 MB/s backplane bus
    - 50K OS insns + 100K user insns per I/O operation
    - SCSI-2 controllers (20 MB/s): each accommodates up to 7 disks
    - 5 MB/s disks with $t_{seek} + t_{rotation}$ = 10 ms, 64 KB reads
  - Determine
    - What is the maximum sustainable I/O rate?
    - How many SCSI-2 controllers and disks does it require?

Designing an I/O System for Bandwidth

- First: determine I/O rates of components we can't change
  - CPU: (300M insn/s) / (150K Insns/IO) = 2000 IO/s
  - Backplane: (100M B/s) / (64K B/IO) = 1562 IO/s
  - Peak I/O rate determined by bus: **1562 IO/s**
- Second: configure remaining components to match rate
  - Disk: 1 / [10 ms/IO + (64K B/IO) / (5M B/s)] = 43.9 IO/s
  - How many disks?
    - (1562 IO/s) / (43.9 IO/s) = **36 disks**
  - How many controllers?
    - (43.9 IO/s) * (64K B/IO) = 2.74M B/s per disk
    - (20M B/s) / (2.74M B/s) = 7.2 disks per SCSI controller
    - (36 disks) / (7 disks/SCSI-2) = **6 SCSI-2 controllers**
- Caveat: real I/O systems modeled with simulation

Designing an I/O System for Latency

- Previous system designed for bandwidth
- Some systems have latency requirements as well
  - E.g., database system may require maximum or average latency
- Latencies are actually harder to deal with than bandwidths
  - **Unloaded system**: few concurrent IO transactions
    - Latency is easy to calculate
  - **Loaded system**: many concurrent IO transactions
    - Contention can lead to queuing
    - Latencies can rise dramatically
    - Queuing theory can help if transactions obey fixed distribution
    - Otherwise simulation is needed

Summary

- Role of the OS
- Device characteristics
  - Data bandwidth
  - Disks
    - Structure and latency: seek, rotation, transfer, controller delays
- Bus characteristics
  - Processor-memory, I/O, and backplane buses
  - Width, multiplexing, clocking, switching, arbitration
- I/O control
  - I/O instructions vs. memory mapped I/O
  - Polling vs. interrupts
  - Processor controlled data transfer vs. DMA
    - Interaction of DMA with memory system