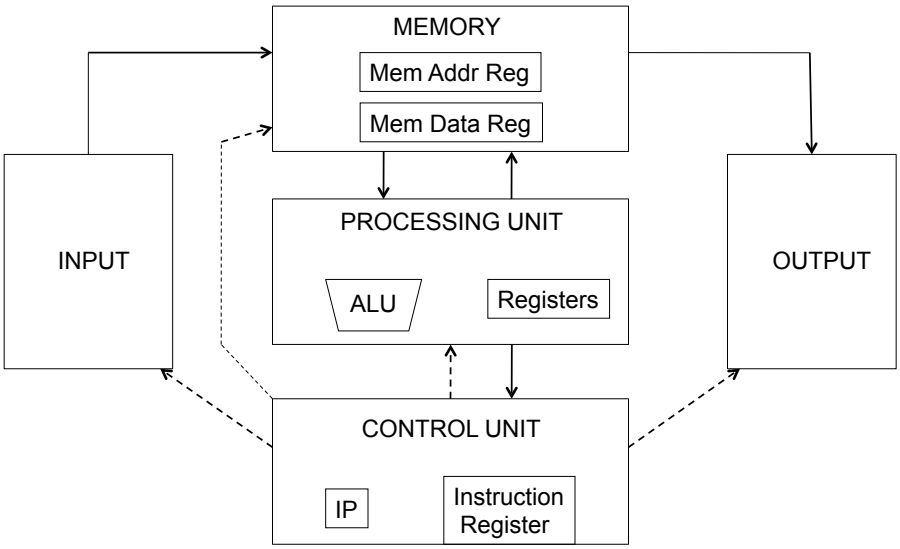
IO

CS 305

Computer Architecture

The Von-Neumann Model (of a Computer)



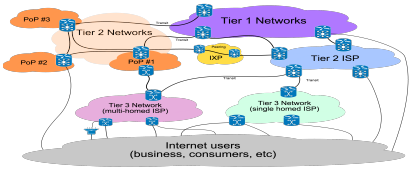
2

Why is I/O needed?

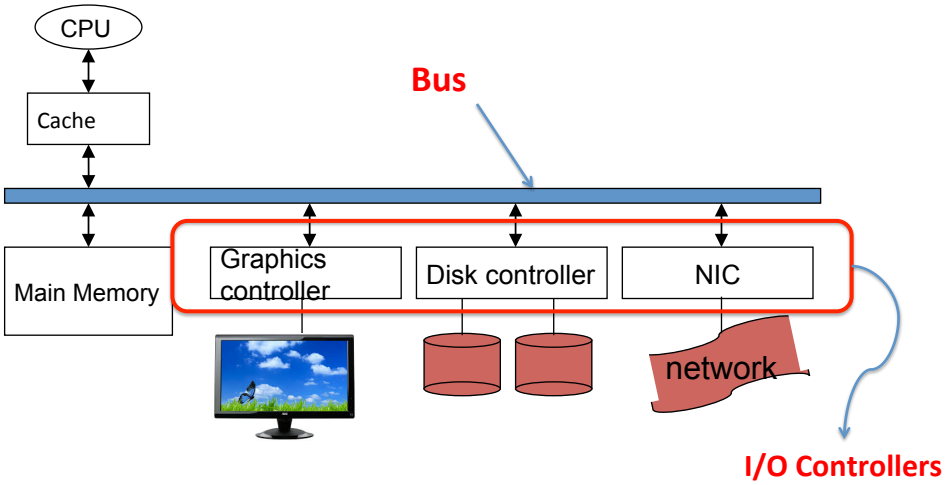
To/from **users**

To/from **non-volatile media**

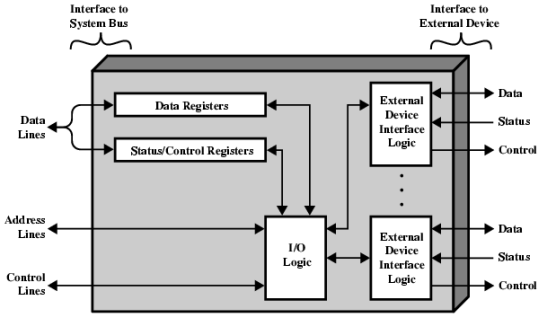
To/from **other computers**



Organization of IO



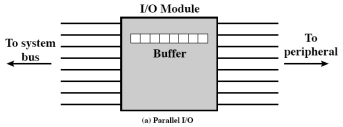
I/O Controller



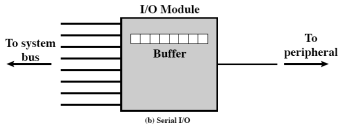
- A **data register**, which is readable (by the processor) for an input device (e.g., a simple keyboard), writable for an output device (e.g., a simple printer), and both readable and writable for input/output devices (e.g., disks).
- **A control register** for giving commands to the device.
- A readable **status register** for reporting errors and announcing when the device is ready for the next action (e.g., for a keyboard telling when the data register is valid, and for a printer telling when the character to be printed has be successfully retrieved from the data register).

Two Types of Interfaces

- ***Serial interface***
  - Single signal wire (also need ground); one bit at a time
  - Less complex hardware with lower cost
  - Most popular serial interface???
  - COM ports on a machine



- ***Parallel interface***
  - Many wires; each wire carries one bit at any time
  - *Width* is number of wires
  - Complex hardware with higher cost
  - Theoretically faster than serial
  - Practical limitation: at high data rates, close parallel wires have potential for interference
  - LPT is the parallel port



Sharing of Interfaces

- Cannot afford to have a separate physical interconnect per device
  - Too many physical wires
  - Not enough pins on a processor chip
  - Interface hardware adds economic cost
- Solution is sharing
  - Allow multiple devices to use a given interconnection
  - Known as a ***bus***

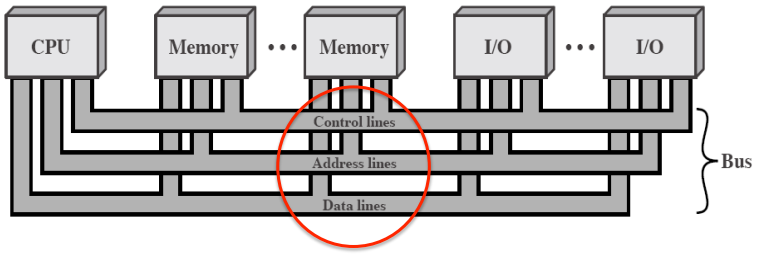
Sharing the bus

- Most buses shared by multiple devices
- Need an *access protocol*
  - Determines which device can use the bus at any time
  - All attached devices follow the protocol
- Note: it is possible and in fact common to have multiple buses in one computer

Types of Buses

- ***Processor-memory bus (usually proprietary)***
  - Short and high speed
  - Matched to the memory system to maximize the memory-processor bandwidth
  - Optimized for cache block transfers
- ***Backplane bus*** *(maybe industry standard)*
  - The backplane is an interconnection structure within the chassis
  - Used as an intermediary bus connecting I/O busses to the processor-memory bus
- ***I/O bus*** *(industry standard, e.g., SCSI, PCI,USB, Firewire)*
  - Usually is lengthy and slower
  - Needs to accommodate a wide range of I/O devices
  - Connects to the processor-memory bus or backplane bus

Conceptual Lines In A Bus



- Early bus designs did indeed use separate wires
- To lower cost, many bus designs now arrange to ***multiplex address and data information*** over the same wires (in a request, use the wires to send an address; in a response, use the same wires to send data)
- *Serial bus* multiplexes all communication over one wire

Bus Operations

- Bus hardware only supports two ***transactions***
  - ***Fetch*** (also called *read*)
    - Place an address on the address lines
    - Use control line to signal *fetch* operation
    - Wait for control line to indicate *operation complete*
    - Extract data item from the data lines
  - ***Store*** (also called *write*)
    - Place an address on the address lines and a data item on the data lines
    - Use control line to signal *store* operation
    - Wait for control line to indicate *operation complete*
- Surprise: all device interaction, including communication with video cameras, speakers, and microphones, must be performed using the fetch-store paradigm

Synchronous and Asynchronous Buses

- **Synchronous bus (e.g., processor-memory buses)**
  - Includes a clock in the control lines and has a fixed protocol for communication that is relative to the clock
  - Advantage: involves very little logic and can run very fast
  - Disadvantages:
    - Every device communicating on the bus must use same clock rate
    - To avoid clock skew, they cannot be long if they are fast
  - Constraint:
    - ALL IO devices must run at the same speed.
- **Asynchronous bus (e.g., I/O buses)**
  - It is not clocked, so requires a handshaking protocol and additional control lines (ReadReq, Ack, DataRdy)
  - Advantages:
    - Can accommodate a wide range of devices and device speeds
    - Can be lengthened without worrying about clock skew or synchronization problems
  - Disadvantage: slow(er)

A Problem

- Memory and bus support two widths of data transfer: 4 words and 16 words at a time
- 64-bit **synchronous** bus; 200MHz; 1 clock cycle for addr; 1 for data.
- Two clocks of ``rest'' between bus accesses
- Memory access times: 4 words in 200ns; additional 4 word blocks in 20ns per block.
- Can overlap transferring data with reading next data.

- Find

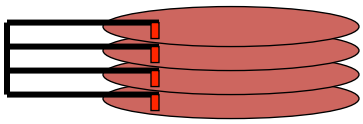
1. Sustained **bandwidth** and **latency** for reading 256 words using both size transfers
2. How many **bus transactions per sec** for each (1 transaction is addr+data).

Magnetic Disks

- Collection of **platters**, 1~15
  - The stack of platters rotates at 3600 to 15000 RPM
- Each platter has two recordable disk surfaces
  - Each surface is divided into concentric circles, or **tracks**
    - 1000~5000 tracks per surface
- Each track is divided into **sectors**
  - 64~200 sectors/track
- Sector is the smallest unit that can be read/written
  - Sector sizes are typically 512 bytes

How to Find Data on Disks?

- Hardware
  - Movable arm
  - Read/write head, electromagnetic coil
- All disk heads for each surface are connected together
  - All disk heads move together



How to Find Data on Disks?

Actions

- Seek. Position the head over proper track
- Rotate. Find the right sector
- Transfer. Send/access the sector

Disk Access Time

Seek
- Seek time varies, depending on which track
- Seek time = 8ms~20ms. Advertised by manufacturers
- Smaller seek time is better
- Question: where to place your frequently-used data?

Rotate
- Rotational delay varies, depending on what sector
- Faster RPM is better
- ***Average rotational delay (e.g. 3600 RPM)***
  = 0.5 rotation / 3600 RPM = 0.5 / (3600/60) = 8.3ms

Example

Advertised seek time = 12ms

512-byte block

rotate at 5400 RPM

Transfer rate = 5MB/sec

Controller overhead = 2ms

What is the average time to read or write one block?

Interfacing with an IO device

Two basic approaches

- ***Programmed I/ O***
  - A terrible name
  - Also called *polled I/ O*
- ***Interrupt-driven I/O***
  - Another poor naming choice
  - Software actually drives I/ O

Polled I/O

1. Processor starts operation on the device.
2. Processor continually checks the device status to see if its complete.
   - To wait for an operation to complete, a processor
     - Executes a loop that repeatedly requests status from device
     - Allows the loop to continue until device indicates "ready"
   - Also called ***busy waiting***

Cost of polling - Example

- Cost of a poll is 400 clock cycles.
- CPU is 500MHz.
- How much of the CPU is needed to poll
  1. A mouse that requires 30 polls per sec?
  2. A CD that sends 2 bytes at a time and achieves 50KB/sec?
  3. A hard disk that sends 16 bytes at a time and achieves 4MB/sec?

Interrupt driven I/O

- Motivation: increase performance by eliminating polling loops
- Technique
  - Add special hardware to processor and devices
  - Allow processor to start operation on a device
  - Arrange for device to *interrupt* the processor when the operation completes

Interrupt Mechanism

- Processor hardware
  - Saves current instruction pointer
  - Jumps to code for the interrupt (interrupt handler – separate for each device)
  - Resumes executing the application when the code executes a *return from interrupt*
- Asynchronous mechanism unlike polling that is synchronous

State management

- Entire state of computation must be saved when interrupt occurs
  - Values in registers
  - Program counter
- Hardware usually saves and restores a few items; interrupt code must save and restore the rest

What if there multiple devices that can interrupt?

- OS maintains, V, an array of pointers to interrupt routines
  - Called an *interrupt vector*
  - Informs bus hardware of the location of V
- Each device is assigned a number from 0 through K-1
- Device specifies its number, i, when interrupting
- Hardware (or in some architectures, the OS) branches to interrupt code at address V[i]
- Interrupt vector IDs are assigned by the OS

During boot

- Processor boots with interrupts *disabled*
- OS
  - Keeps interrupts disabled during initialization
  - Fills in interrupt vector with pointers to interrupt code for each device
- Once all interrupt table entries have been initialized, OS enables interrupts, which allows I/ O to proceed
- When handling an interrupt the OS usually disables other interrupts so that an interrupt handler is not interrupted!

Hierarchical Interrupts

- Simplest processors: only one interrupt at a time
- Advanced processors: devices assigned a priority, and higher priority devices can interrupt lower level interrupt code
- Typically a few priority levels (e.g., 7)
- Rule: at any given time, at most one device can be interrupting at each priority level
- Note: the lowest priority (usually zero) means no interrupt is occurring (i.e., an application program is executing)

Freeing up the CPU with Interrupt driven I/O

- Even with Interrupts, the OS (and the CPU) are involved with transferring the data. Can we free up the CPU to do other useful work?

Direct Memory Access

- Transfer data between I/O and memory
- Processor not involved, so can do useful work when transfer is in progress.
- I/O interrupts only when transfer is completed
- Idea:
  - CPU tells controller where in memory to load data
  - Controller does this work and then informs the CPU.

The details

- Have a DMA engine (a small processor) on the controller.
- The processor initiates the DMA by writing the command into data registers on the controller (e.g., read sector 5, head 4, cylinder 123 into memory location 34500)
- The controller collects data from the device and then sends it on the bus to the memory without bothering the CPU.
- So we have a multimaster bus and need some sort of arbitration.
  - Normally the I/O devices are given higher priority than the CPU.
- Freeing the CPU from this task is good but isn't as wonderful as it seems since the memory is busy (but cache hits can be processed).
- A big gain is that only one bus transaction is needed per bus load. With PIO, two transactions are needed: controller to processor and then processor to memory.
- This was for an input operation (the controller writes to memory). A similar situation occurs for output where the controller reads from the memory). Once again one bus transaction per bus load.
- When the controller detects that the I/O is complete or if an error occurs, it sets the status register accordingly and sends an interrupt to the processor to notify the latter that the I/O is complete.

A complete example

Example: Overhead of Polling I/O

Consider a processor running at 500 MHz

Consider a hard disk
- Transfer 4-word chunks at rate of 4MB/sec

Polling operation includes
- Transferring to the polling routine
- Access the device
- Restart the program
- Total time is 400 cycles

Fraction of CPU time consumed for the I/O?
- Given that no transfer can be missed

Summary

- Devices can use
  - Programmed I/O
  - Interrupt-driven I/O
- Interrupts
  - Allow processor to continue running while waiting for I/O
  - Use vector (usually in memory)
  - Occur "between" instructions in fetch-execute cycle

- Multi-level interrupts handle high-speed and low-speed devices on same bus
- Smart device has some processing power built into the device
- Optimizations for high-speed devices include
  - Direct Memory Access (DMA)
  - Buffer chaining
  - Operation chaining