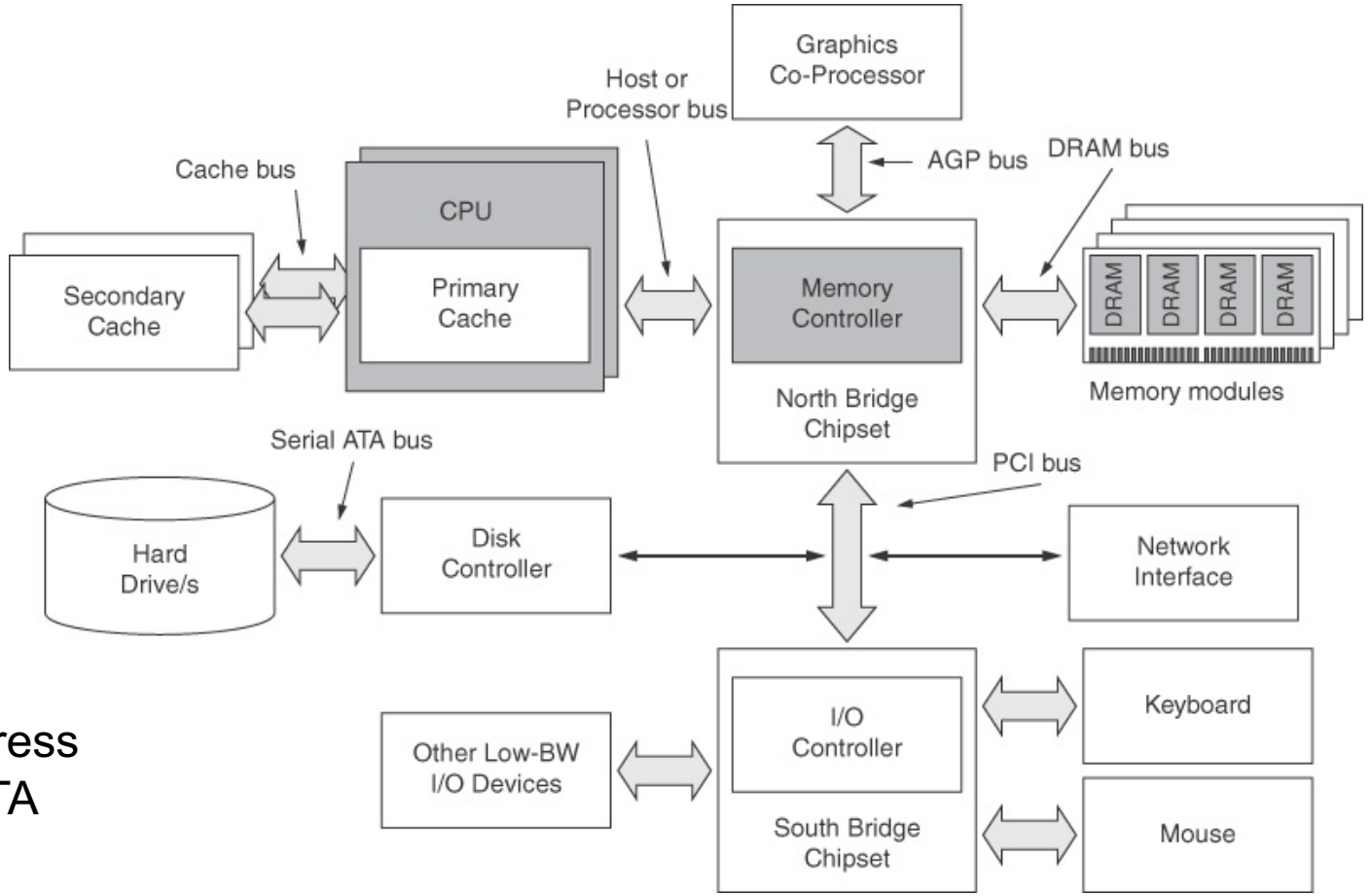
EN1640: Design of Computing Systems
Topic 07: I/O

Professor Sherief Reda
http://scale.engin.brown.edu
Electrical Sciences and Computer Engineering
School of Engineering
Brown University
Spring 2019

[material from Patterson & Hennessy, 4th ed, Harris 1st ed and Parhami]

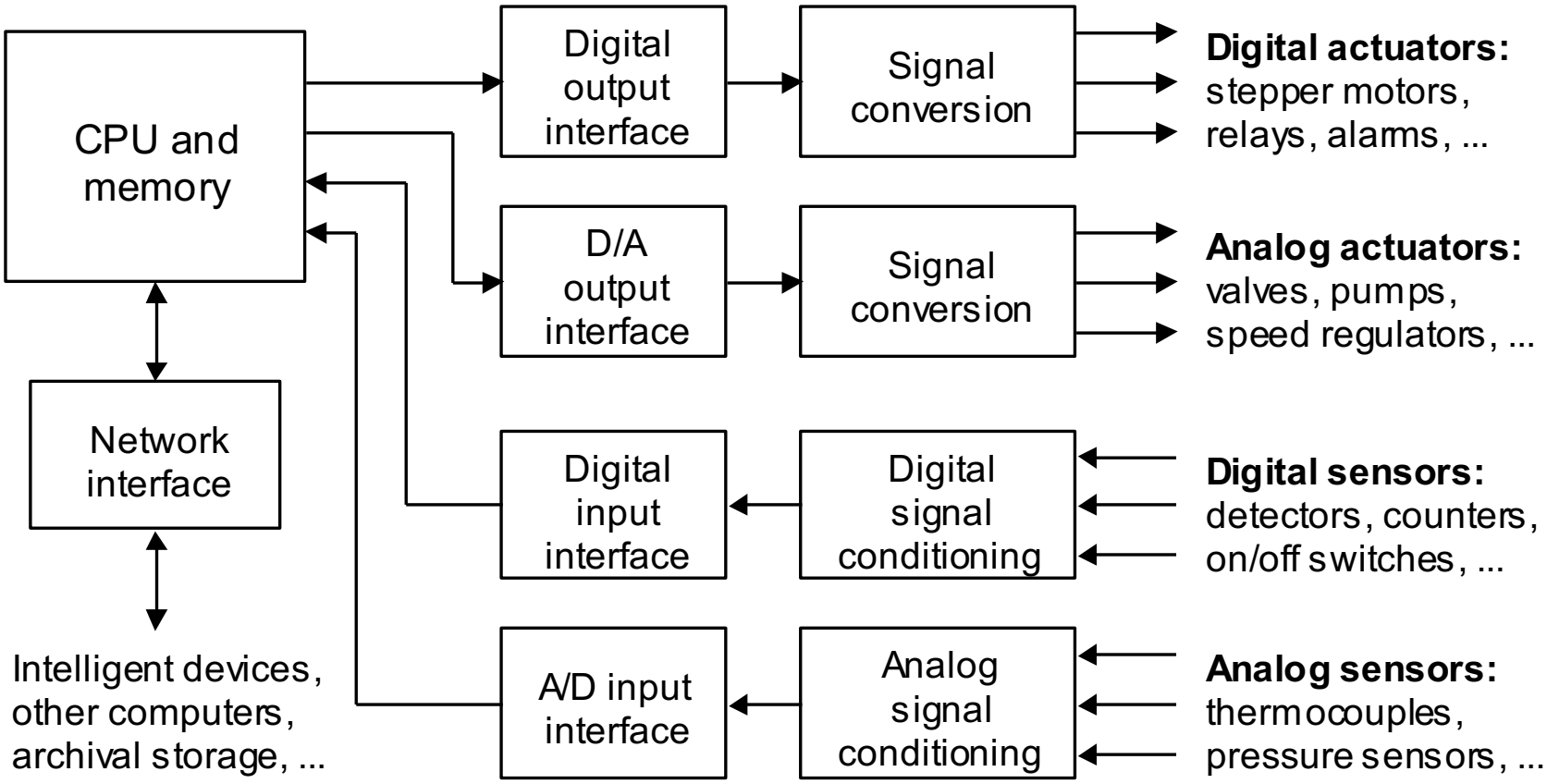
Typical PC I/O organization

Bus: shared communication channel
Parallel set of wires for data and synchronization of data transfer



Busses:
USB
Firewire
PCI express
Serial ATA

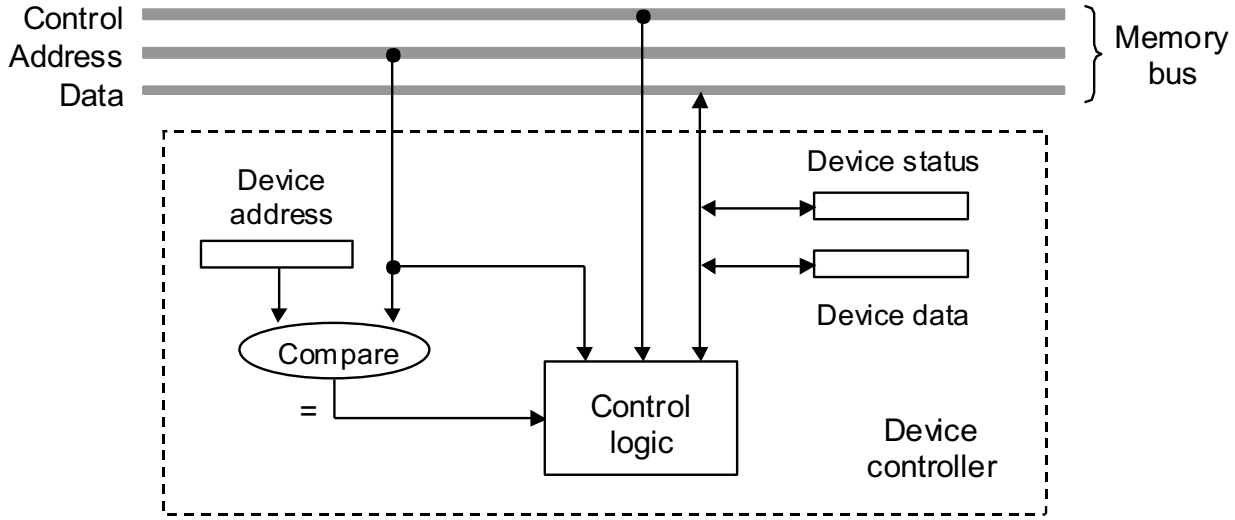
I/O in embedded systems



Performance of I/O system

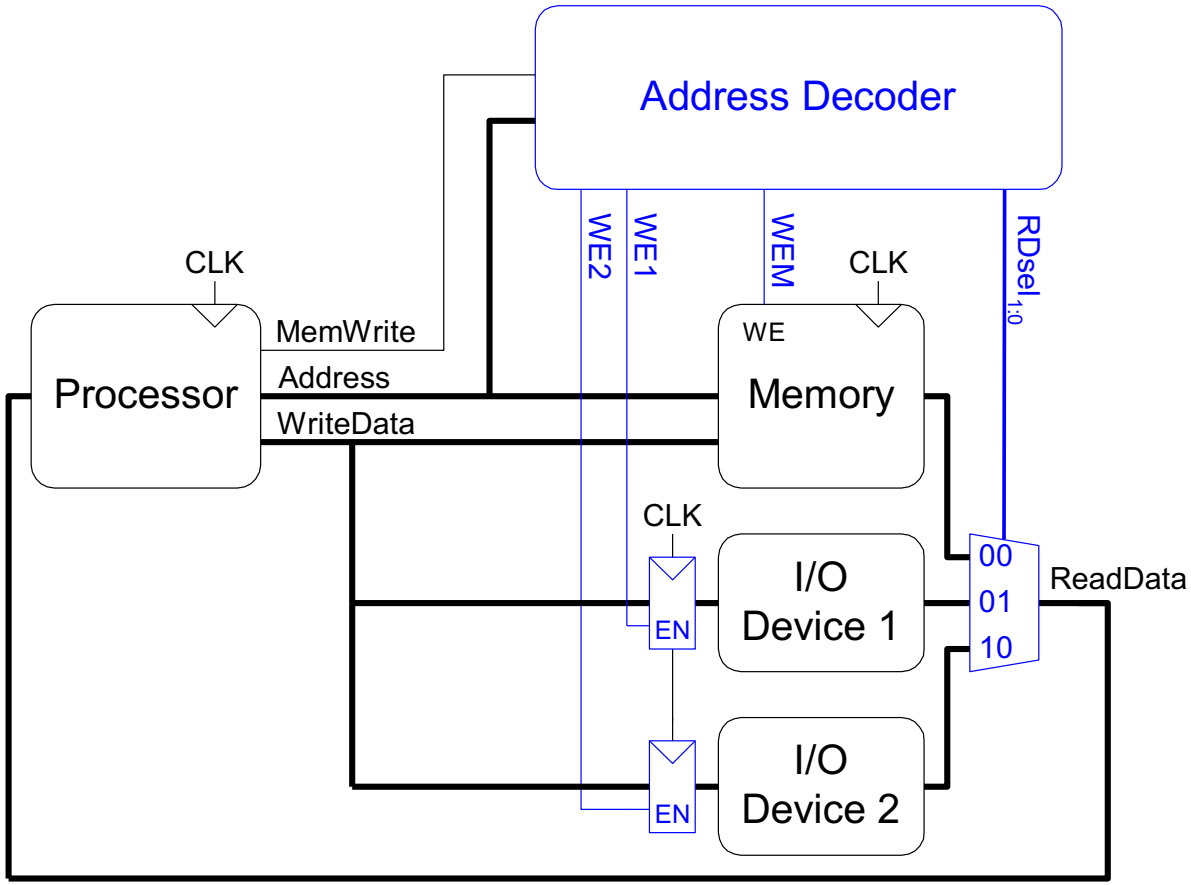
- Latency (response time)
- Throughput (bandwidth)
  - Latency and throughput is sometimes a trade-off
- Reliability
- *Desktops & embedded systems:*
  - response time & diversity of devices
- *Servers:*
  - Interested in throughput (e.g., supercomputing)
  - Latency and throughput: Transactional workloads

I/O commands



- I/O devices are managed by I/O controller hardware interface
- Control lines
  - Read versus write
- Status registers
  - Indicate what the device is doing and occurrence of errors
- Data registers
  - Write: transfer data to a device
  - Read: transfer data from a device

Accessing I/O devices registers using memory mapped I/O interface



- Device registers are addressed in same space as memory
- Example: 0xFFFF0000 to 0xFFFFFFFF reserves the last 64KB in memory for I/O device registers

Why “wait for some time” between I/O memory writes?

- I/O devices are usually (way) slower than the processor.
- It will not be possible to issue commands or read/write data on consecutive instructions because the I/O device is processing the earlier command
- Need to wait for sometime between accesses.
- How much? Three techniques to figure out:
  1. Polling
  2. Interrupts
  3. Direct memory transfer

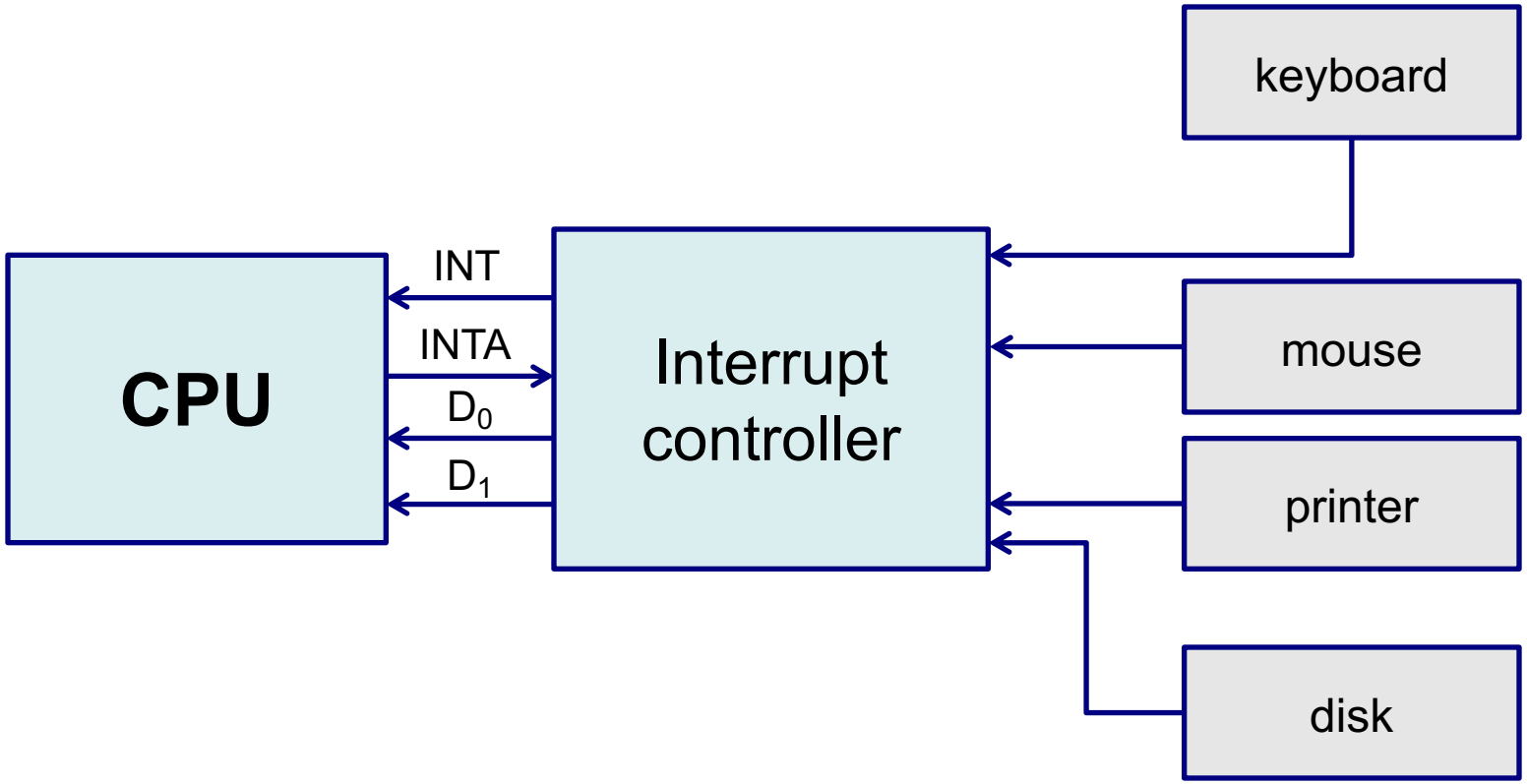
1. Polling

- Periodically check I/O status register
  - If device ready, do operation
  - If error, take action
- Common in small or low-performance real-time embedded systems
  - Predictable timing
  - Low hardware cost
- In other systems, wastes CPU time

2. Interrupts

- When a device is ready or error occurs
  - Controller interrupts CPU
- Interrupt is like an exception
  - But not synchronized to instruction execution
  - Can invoke handler between instructions
  - Cause information often identifies the interrupting device
- Priority interrupts
  - Devices needing more urgent attention get higher priority
  - Can interrupt handler for a lower priority interrupt

Organization of interrupt-driven I/O



When CPU receives interrupt signal:

1. Saves processor state
2. Loads the PC with the appropriate interrupt service route (ISR)
3. Return and restore state

In RISCV, Interrupt controller called Platform-Level IC (PLIC).

Interrupt handling in RISCV

- Interrupts can be synchronous or asynchronous.
- A pending interrupt is flagged in register *mip*.
- Interrupts are identified by reading the *mcause* control & status register (CSR) → register which contain working state of RISCV processor.

Interrupt = 1 (Asynchronous)	
Exception Code	Description
0	User Software Interrupt
1	Supervisor Software Interrupt
2	Reserved
3	Machine Software Interrupt
4	User Timer Interrupt
5	Supervisor Timer Interrupt
6	Reserved
7	Machine Timer Interrupt
8	User External Interrupt
9	Supervisor External Interrupt
10	Reserved
11	Machine External Interrupt
12 - 15	Reserved
≥16	Local Interrupt X

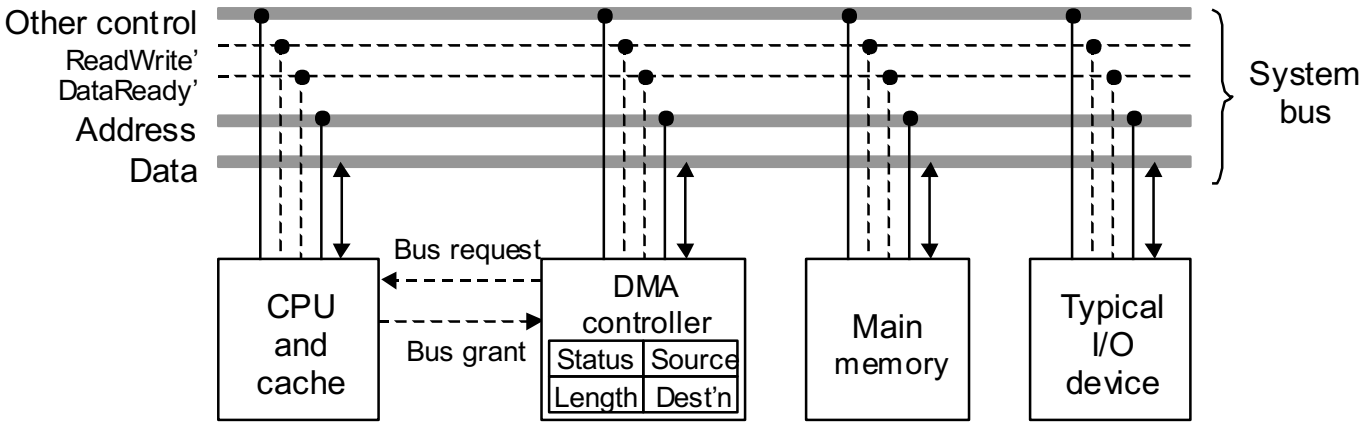
Interrupt = 0 (Synchronous)	
Exception Code	Description
0	Instruction Address Misaligned
1	Instruction Access Fault
2	Illegal Instruction
3	Breakpoint
4	Load Address Misaligned
5	Load Access Fault
6	Store/AMO Address Misaligned
7	Store/AMO Access Fault
8	Environment Call from U-mode
9	Environment Call from S-mode
10	Reserved
11	Environment Call from M-mode
12	Instruction Page Fault
13	Load Page Fault
14	Reserved
15	Store/AMO Page Fault
≥16	Reserved

<u>Steps after receiving an interrupt:</u>

- The current state is saved by CPU by copying the PC into special register *MPEC*
- The PC is forced to a default interrupt handler address
- The interrupt handler code:
  - Pushes all registers onto the stack
  - Branches to a handler code based on the nature of the interrupt e.g., table)
  - Pops back all registers
- Load MPEC back to PC

3. Using DMAs for data transfer

- Polling and interrupt-driven I/O
  - CPU transfers data between memory and I/O data registers
  - Time consuming for high-speed devices
- Direct memory access (DMA)
  - OS provides starting address in memory
  - I/O controller transfers to/from memory autonomously
  - Controller interrupts on completion or error



DMA / cache interaction

- If DMA writes to a memory block that is cached
  - Cached copy becomes stale
- If write-back cache has dirty block, and DMA reads memory block
  - Reads stale data
- Need to ensure cache coherence
  - Flush blocks from cache if they will be used for DMA
  - Or use non-cacheable memory locations for I/O

OS responsibilities to I/O

1. Coordinate sharing of I/O subsystem among multiple programs
2. Handling interrupts
3. Supplying routines to handle low-level control of I/O

Device drivers

- Understand the HW interface of the I/O device and knows how to access it
- Provide the applications with routines (i.e., system calls) that abstract the HW aspects of the I/O and simplifies programmer life
- Part of the OS (no need to re-invent the wheel for every application)
- *Example:* a OS device driver routine could take a number as input and displays it on 7-segment display

I/O summary

- Registers of I/O devices are mapped to the main memory space in reserved segments
- Processor sends commands and data to the I/O device over the bus as if it is writing to a memory
- I/O device controller accepts commands and data and carry out required actions
- Processor can either query I/O device controller to check if it is done or I/O controller can interrupt the processor to tell it that it is done
- DMA controller relieves the processor data transfer