EN164: Design of Computing Systems
Lecture 32-33: Misc – I/O

Professor Sherief Reda
http://scale.engin.brown.edu
Electrical Sciences and Computer Engineering
School of Engineering
Brown University
Spring 2011

[ material from Patterson & Hennessy, 4th ed, Harris 1st ed and Parhami ]
**Typical PC I/O organization**

**Busses:**
- USB
- Firewire
- PCI express
- Serial ATA

**Bus:** shared communication channel  
Parallel set of wires for data and synchronization of data transfer
I/O in embedded systems

- **CPU and memory**
- **Network interface**

**Digital output interface**
- Signal conversion
- Digital actuators: stepper motors, relays, alarms, ...

**D/A output interface**
- Signal conversion
- Analog actuators: valves, pumps, speed regulators, ...

**Digital input interface**
- Digital signal conditioning
- Digital sensors: detectors, counters, on/off switches, ...

**A/D input interface**
- Analog signal conditioning
- Analog sensors: thermocouples, pressure sensors, ...

Intelligent devices, other computers, archival storage, ...
Performance of I/O system

- Latency (response time)
- Throughput (bandwidth)
- Latency and throughput is often a trade-off

*Desktops & embedded systems:*
  - Mainly interested in response time & diversity of devices

*Servers:*
  - Interested in throughput (e.g., supercomputing) & expandability of devices
  - Latency and throughput: Transactional workloads
Busses and their appeal

- Point-to-point connections require quadratic number of wires between $n$ units require $n(n - 1)$ channels, or $n(n - 1)/2$ bidirectional links
- Bus connectivity requires only one input and one output port per unit, linear number of wires
- Busses need arbitration
- Bus can be bottleneck; latency and throughput can degrade
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
Example: LCD display in DE2 board

LCD has two rows of characters (16 character each)

- LCD has two 8-bit registers (command and data)
- Two control signals: RS and R/W

- RS = 0 -> data bus has command
- RS = 1 -> data bus has data value
- R/W = 0 -> write (command or data)
- R/W = 1 -> read (e.g., state)
Example: LCD

To show a character at a desired location:
1. write a *command* to move cursor to location (top table)
2. write the desired character data value (right table)

Examples:
- RS = 0 data = 0x80
- RS = 1 data = 0x41
- RS = 0 data = 0xC0
- RS = 1 data = 0x54
Accessing I/O devices 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
Example: LCD

addi $t0, $0, 0x080
sw $t0, 0xFFFO($0)
// wait for some time
addi $t0, $0, 0x141
sw $t0, 0xFFFO($0)

Recall that the 16-bit immediate is sign-extended to 0xFFFFFFFF4

- WriteData bus is \{RS, data\} of the LCD controller
- LCD address is at 0xFFFF0000
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? Two techniques to figure out:
  1. Polling
  2. Interrupts
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
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

![Diagram showing DMA communication]
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
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 string as input and displays it on LCD 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