Today’s lecture

- Use **addressing** to get **data** from the outside world
  - Data is moved from peripherals to memory
  - **Addressing** schemes
    - Memory-mapped vs. isolated I/O
  - **Data** movement schemes
    - Programmed I/O vs. Interrupt-driven I/O vs. Direct memory access
Most modern operating systems **pre-emptively** schedule programs

- If a computer is running two programs A and B, the O/S will periodically switch between them
  1. Stop A from running
  2. Copy A’s register values to memory
  3. Copy B’s register values from memory
  4. Start B running

How does the O/S stop program A?
We can treat most devices “as if” they were memory with an “address” for reading/writing

- Many ISAs often make this analogy explicit — to transfer data to/from a particular device, the CPU can access special addresses.

- **Example**: Video card can be accessed via addresses 3B0-3BB, 3C0-3DF and A0000-BFFFF.
Most ISAs one of two protocols for addressing devices: memory-mapped I/O or isolated I/O

Memory-mapped I/O reserves a portion of main memory addresses for I/O

Isolated I/O creates a separate memory address space for devices
Memory-mapped I/O divides main memory addresses into actual memory and devices

- Apple IIe (right) had a 16-bit address bus
  - Addresses C000-CFFF accessed I/O devices.
  - No actual main memory at C000-CFFF
  - All other addresses reference main memory.
- I/O addresses are shared by many peripherals.
  - C010 → keyboard
  - C030 → speaker
- Some devices may need several I/O addresses.
We use control and addressing to determine when data goes to memory or devices

- Each device has to monitor the address bus to see if it is the target. (Apple IIe example)
  - Main memory ignores any transactions with addresses C000-CFFFFFF.
  - The speaker only responds when C030 appears on the address bus.
Isolated I/O creates two separate address spaces and needs two sets of instructions

- **Example** (x86):
  - regular instructions like **MOV** reference RAM
  - special instructions **IN** and **OUT** access a separate I/O address space

- An address could refer to *either* main memory *or* an I/O device, depending on the instruction used
MIPS provides the following instructions for managing memory: load word, load halfword, load byte, store word, store halfword and store byte.

Which I/O addressing method does MIPS use?

a) Memory-mapped I/O
b) Isolated I/O
MIPS/SPIMbot uses memory-mapped I/O

- Examples

  \[
  \begin{align*}
  \text{lw} & \quad $reg, 0xffff0020($0) \quad \# \text{gets SPIMbot x-coord} \\
  \text{sw} & \quad $reg, 0xffff0010($0) \quad \# \text{sets bot speed} = $reg
  \end{align*}
  \]

- Some control commands require a sequence of instructions

  \[
  \begin{align*}
  \text{sw} & \quad $reg, 0xffff0014($0) \\
  \text{li} & \quad $t0, 1 \\
  \text{sw} & \quad $t0, 0xffff0018($0) \quad \# \text{sets bot angle} = $reg
  \end{align*}
  \]
## Example SPIMbot commands

<table>
<thead>
<tr>
<th>What</th>
<th>How</th>
</tr>
</thead>
<tbody>
<tr>
<td>get SPIMbot’s current x/y-coordinate</td>
<td>lw from 0xffff0020 (x)</td>
</tr>
<tr>
<td></td>
<td>lw from 0xffff0024 (y)</td>
</tr>
<tr>
<td>set SPIMbot’s angle (absolute)</td>
<td>sw the angle to 0xfffff0014</td>
</tr>
<tr>
<td></td>
<td>sw 1 to 0xfffff0018</td>
</tr>
<tr>
<td>set SPIMbot’s angle (relative)</td>
<td>sw the angle to 0xfffff0014</td>
</tr>
<tr>
<td></td>
<td>sw 0 to 0xfffff0018</td>
</tr>
<tr>
<td>set SPIMbot’s velocity</td>
<td>sw a number between –10 and 10 to 0xfffff0010</td>
</tr>
<tr>
<td>read the current time</td>
<td>lw from 0xfffff001c</td>
</tr>
<tr>
<td>request a timer interrupt</td>
<td>sw the desired (future) time to 0xfffff001c</td>
</tr>
<tr>
<td>acknowledge a bonk interrupt</td>
<td>sw any value to 0xfffff0060</td>
</tr>
<tr>
<td>acknowledge a timer interrupt</td>
<td>sw any value to 0xfffff006c</td>
</tr>
</tbody>
</table>
SPIMbot coordinate system

(0, 0) (300, 0)
X=0 X=300
Y=0

Y=300 (0, 300) (300, 300)

0° 90° 180° 270°
What will SPIMbot do?

Suppose we want SPIMbot to travel north. Which of the following sequences of instructions will always cause SPIMbot to travel north?

a)  
li $a0, 270  
sw $a0, 0xffff0014 ($zero)  
li $t0, 0  
sw $t0, 0xffff0018 ($zero)

b)  
li $a0, 270  
sw $a0, 0xffff0014 ($zero)  
li $t0, 1  
sw $t0, 0xffff0018 ($zero)  
li $a0, 270  
sw $a0, 0xffff0014 ($zero)  
li $t0, 1  
sw $t0, 0xffff0018 ($zero)
In **programmed I/O**, the program or OS is responsible for transmitting data

- CPU makes a request and then waits (loops) until device is ready (loop 1)
- Buses are typically 32-64 bits wide, so loop 2 is repeated for large transfers

- Also called polling
Programmed I/O is generally bad

- A lot of CPU time is needed for this!
  - most devices are slow compared to CPUs
  - CPU also “wastes time” doing actual data transfer

- CPU must ask repeatedly

- CPU must ask often enough to ensure that it doesn’t miss anything, which means it can’t do much else while waiting
**Interrupt-driven I/O** transfers data when devices interrupt the processor

Interrupt-driven I/O solves the inefficiencies of Programmed I/O

- Instead of waiting, the CPU continues with other calculations
- The device interrupts the processor when the data is ready

- CPU still does the data transfer
Direct memory access (DMA) parallelizes data transfer with a separate controller

- The DMA controller is a simple processor which manages I/O and memory data transfers
  - The CPU asks the DMA controller to transfer data between a device and main memory. After that, the CPU can continue with other tasks
  - The DMA controller issues requests to the right I/O device, waits, and manages the transfers between the device and main memory
  - Once finished, the DMA controller interrupts the CPU
Example of data transfer using DMA

Since both the processor and the DMA controller may need to access main memory, some form of arbitration is required.
Side Note:

MIPS uses a co-processor (a separate datapath with a separate set of registers) to help handle interrupts.
**Interrupts** vs. **Exceptions**

- **Interrupts**: External events that require the processor’s attention
  - Not an error, should be recoverable
  - OS manages and resolves the interrupt

- **Exceptions**: Typically errors that are detected within the processor
  - Always an error, may or may not be recoverable
  - OS must resolve the exception or ask the program to resolve
More details on interrupts

- *Examples*: I/O device needs attention, timer interrupts to mark cycle
- All interrupts are recoverable: interrupted program should resume after the interrupt is handled
- OS responsible to do the right thing, such as:
  - Save the current state and shut down the hardware devices
  - Find and load the correct data from the hard disk
  - Transfer data to/from the I/O device, or install drivers
More details on exceptions

- **Examples**: illegal instruction opcode, arithmetic overflow, or attempts to divide by 0

- There are two possible ways of resolving these errors:
  - If the error is un-recoverable, the operating system kills the program
  - Less serious problems can often be fixed by OS or the program itself
Consider the following scenario.

A program running on MIPS tries to perform a load word from memory at address 0x60000003. The OS immediately stops the program from running and takes control. Is it more likely that an interrupt or exception just happened?

a) An interrupt
b) An exception
Consider the following scenario.

A program sets a 1 second timer. One second later, the OS stops the program from running and takes control. Is it more likely that an interrupt or exception just happened?

a) An interrupt
b) An exception