Interrupts

- Modern operating systems (OS) schedule programs *pre-emptively*

- Several running programs: OS periodically switches between them

- Specifically, the OS will:
  - Stop “current” program A from running
  - Copy A’s register values to memory
  - Copy another program B’s register values from memory
  - Start B running

- OS stops program A by issuing an *interrupt*
I/O Programming, Interrupts, and Exceptions

- Most I/O requests are made by applications or OS
  - involve moving data between peripheral device and main memory

- Two main ways for programs to communicate with devices:
  - Memory-mapped I/O
  - Isolated I/O

- Several ways to transfer data between devices and main memory:
  - Programmed I/O
  - Interrupt-driven I/O
  - Direct memory access

- We will explore some of these in more detail in the MPs
Communicating with devices

- Most devices can be considered as memories, with an “address” for reading or 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.

- Two ways to access these addresses:
  - Memory-mapped I/O
  - Isolated I/O
Memory-mapped I/O

- With memory-mapped I/O, one address space is divided into two parts
  - Some addresses refer to physical memory locations
  - Other addresses actually reference peripherals

- The I/O address-space is shared by many peripherals
  - Some devices may need several I/O addresses

- CPU writes data to the appropriate I/O address
  - Address and data then transmitted along the bus

- Every device must monitor the address bus to see if it is the target
Isolated I/O

- Here, there are *separate* address spaces for memory and I/O devices
  - special instructions that access the I/O space

- *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
Comparing memory-mapped and isolated I/O

- Memory-mapped I/O with a single address space is nice because the same instructions that access memory can also access I/O devices
  - E.g., in MIPS `sw` instructions to the proper addresses can store data to an external device
  - However, part of the address space is taken by I/O devices, reducing the amount of main memory that’s accessible

- With isolated I/O, special instructions are used to access devices
  - Less flexible for programming
  - However, I/O and memory addresses are kept separate, so the amount of accessible memory isn’t affected by I/O devices

- SPIMbot uses memory-mapped I/O:
  \[
  \text{sw} \quad $\text{reg}, \ 0xffff0080($0) \quad \# \ \text{prints integer} \ \$\text{reg}
  \]
  \[
  \text{sw} \quad $\text{reg}, \ 0xffff0010($0) \quad \# \ \text{sets bot speed} = \ $\text{reg}
  \]
Transferring data with programmed I/O

- How data is transferred?

- With programmed I/O, it’s all up to the program or the OS
  - CPU makes a request and then waits (loops) until device is ready
  - Buses are typically 32-64 bits wide, so above loop is repeated for large transfers

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

- Continually checking to see if a device is ready is called **polling**
- Wastes CPU time:
  - 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
- Solution: **Interrupt-driven 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

- One final method of data transfer is to introduce a direct memory access, or DMA, controller.

- The DMA controller is a simple processor which does most of the functions that the CPU would otherwise have to handle.
  - 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.

- This is yet another form of parallel processing.
Main memory problems

- As you might guess, there are some complications with DMA
  - Since both the processor and the DMA controller may need to access main memory, some form of arbitration is required
  - If the DMA unit writes to a memory location that is also contained in the cache, the cache and memory could become inconsistent

- Having the main processor handle all data transfers is less efficient, but easier from a design standpoint!
Interrupts vs. Exceptions

- **Interrupts** are external events that require the processor’s attention
  - *Examples*: I/O device needs attention, timer interrupts to mark cycle
- These situations are normal, not errors
  - All interrupts are recoverable: interrupted program will need to be resumed 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
- **Exceptions** are typically errors that are detected within the processor
  - *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
User handled exceptions

- Sometimes users want to handle their own exceptions:
  - e.g. numerical applications can scale values to avoid floating point overflow/underflow

- Many operating systems provide a mechanism for applications for handling their exceptions
  - Unix lets you register “signal handler” functions

- Modern languages like Java provide programmers with language features to “catch” exceptions (this is much cleaner)
Instruction Emulation: an exception handling example

- Periodically ISA’s are extended with new instructions
  - e.g., MMX, SSE, SSE2, etc

- If programs are compiled with these new instructions, they will not run on older implementations (e.g., a 486)
  - “Forward compatibility” problem

- Can’t change existing hardware, so we add software to “emulate” these instructions

```
User

Execute Application

Illegal opcode exception

Kernel

Decode inst in software;
Perform it’s functionality

Return from exception

Execute Application
```
How interrupts/exceptions are handled

- For simplicity exceptions and interrupts are handled the same way

- When an exception/interrupt occurs, we stop execution and transfer control to the operating system, which executes an “interrupt handler” to decide how it should be processed

- The interrupt handler needs to know two things:
  - The cause of the interrupt/exception (e.g. overflow, illegal opcode)
  - Which instruction was executing when the interrupt/exception occurred. This helps the operating system report the error or resume the program (where?)

- This is another example of interaction between software and hardware, as the cause and current instruction must be supplied to the operating system by the processor
Receiving Interrupts

- To receive interrupts, the software has to enable them
  - MIPS: done by writing to the Status register (on the co-processor)
  - Enable interrupts by setting bit zero
  - Select which interrupts to receive by setting one or more of bits 8-15

```assembly
li    $t0, 0x8001  # enable interrupts and interrupt 15
mtc0  $t0, $12     # set Status register = $12
```
Handling Interrupts

- When an interrupt occurs, the Cause register indicates which one
  - For an exception, the exception code field holds the exception type
  - For an interrupt, the exception code field is 0000 and bits will be set for pending interrupts

```
mfc0  $t0, $13  # Get Cause register
srl   $t1, $t0, 2
andi  $t1, $t1, 0xF
bne   $t1, $0, non_interrupt
```