MIPS interrupts

The primary interfact into the MIPS interrupt system is through two co-processor registers, `$12` and `$13`. Both of these are on coprocessor 0, and can be accessed by `mfc0` (move from coprocessor 0) and `mtc0` (move to coprocessor 0).

Register `$12` contains the interrupt mask. This consists of a global interrupt enable (bit 0), and enables for the individual interrupts. Please note that when we enable bonk interrupts, we use an interrupt mask of `0x1001` - both the 12th bit (the bonk interrupt level is 12) and the 0th bit (global interrupt enable) are set. The register that we write to `$12` is not related to the interrupt level. The interrupt mask will always be at register `$12`.

Register `$13` is the cause register, and indicates which interrupts have occurred (and not been handled). The layout is similar to the interrupt mask. Each bit position corresponds to a different level interrupt. If the bit is set, the interrupt needs to be handled.

SPIMbot commands and controls

SPIMbot's I/O devices are *memory-mapped* and *interrupt-driven*.

Memory-mapped I/O:

Move: SPIMbot can be ordered to move by writing to address `0xffff0010`. This will cause the SPIMbot to move forward, assuming that sufficient time has passed since the SPIMbot's last movement.

Location: SPIMbot's current (x, y) co-ordinates are available at memory locations `0xffff0020` and `0xffff0024` respectively. The entire playing field is a 12 by 12 grid. *Note:* SPIMbot's location cannot be *set*. You must *drive* the bot to a desired location – it cannot teleport!

Angle: Spimbot's orientation is set by specifying an angle (between -360 and +360) in memory location `0xffff0014`. This alone has no immediate effect. To actually change the angle, the value 0 or 1 must be written to memory location `0xffff0018`: writing 0 means that the angle specified is *relative* to SPIMbot's current angle; writing 1 means that the angle specified is an *absolute* angle relative to the usual X-axis. Since the playing field consists of discrete locations, the SPIMbot's orientation will be rounded off to the nearest 45°.

Timer: The current value of SPIMbot's timer is obtained by reading memory location `0xffff001c`. If a value v is written to this memory, a *timer interrupt* (see below) is generated when SPIMbot's timer equals v.

Debugging: To help debugging, SPIMbot provides an easy way to output integers: simply write the value to memory location `0xffff0080`.

Interrupts:

Bonk: If SPIMbot runs into a wall, it receives a *bonk* interrupt (interrupt level 12). To receive bonk interrupts, bit 12 of the Status register should be set. To acknowledge bonk interrupts, some (arbitrary) value should be written to address `0xffff0060`.

Timer: When SPIMbot's timer equals the last value written to memory location `0xffff001c`, it receives a *timer* interrupt (interrupt level 15). To receive timer interrupts, bit 15 of the Status register should be set. To acknowledge timer interrupts, some (arbitrary) value should be written to address `0xffff006c`.

Getting Started

We will be modifying `bouncing.s`, a simple spimbot that goes until it hits a wall, at which point it turns 120°. You can find the file on the course web page (under `Section Notes`) or in `~cs232/bouncing.s`.

1. If you are on a SUN machine, you should log into a Linux machine, since SPIMbot is only available on Linux.

   ```
   ssh -X netid@dcllnx##.ews.uiuc.edu
   ```

2. Start up the CS232 pseudo-shell.

   ```
   cs232
   ```

3. Copy `bouncing.s` over to your CS232 directory.

   ```
   cp ~cs232/bouncing.s .
   ```

4. Start up `spimbot`.

   ```
   spimbot -file bouncing.s
   ```

This will bring up the SPIM debugger, as well as a graphical representation of the playing field. If you hit the `Run` button, and then `Ok` on the window that pops up, `bouncing.s` will begin executing, and the SPIMbot will traverse the playing field, bouncing when it hits an edge.

Drawing a Box

Your first task will be to modify `bouncing.s` so that instead of bouncing off the edges, its movements will describe a small box. Make a copy of `bouncing.s` called `box1.s`, and make the following changes:

1. Enable the timer interrupt, instead of the bonk interrupt. The code that enables the bonk interrupt is right after the `main` label.
2. Change the interrupt handler so it detects timer interrupts, instead of bonk interrupts. Detecting of bonk interrupts occurs after the `interrupt_dispatch` label.
3. Change the portion of the interrupt handler that handles bonk interrupts (`bonk_interrupt`), so that it handles timer interrupts instead. You will need to change the code so that it acknowledges timer interrupts instead of bonk interrupts, and set up the next timer interrupt.
4. Change the new timer interrupt handler so that it rotates the SPIMbot 90° instead of 120°.
5. Set up the first timer interrupt, after enabling timer interrupts.

Minimizing the Interrupt Footprint

Your next task will be to offload as much of the work onto the user portion of the code as is possible. First, make a copy of `box1.s` called `box2.s`.

Typically, interrupts should be handled as quickly as possible, since they are a disruption to normal processing. A typical interrupt handler will perform the following tasks:

1. Handle the conditions causing the interrupt.
2. Acknowledge the interrupt.
3. Provide signals indicating that an interrupt has occurred. This can include waking up processes or threads that are interested in the interrupt.

Currently, `box2.s` does all of the work in the interrupt. What you will want to do is:

1. Allocate a word of data in user space so that the interrupt can signal the main program. Initialize it to 0.
2. Change the timer interrupt handler so that it sets the flag to 1, instead of changing the orientation of the SPIMbot.
3. Add code to `wait_loop` to check if the flag has been set to 1. If it has, the flag should be set to 0, and the SPIMbot rotated 90°.

`wait_loop` is chosen to check for the flag because, 1) it is just busy waiting, so we will not be interrupting anything important, and 2) the majority of the time is probably spent in `wait_loop`, and the code in `wait_loop` is executed fairly frequently, so once the interrupt handler signals that an interrupt has occurred, it will be dealt with in a fairly timer manner.

Register Destruction

Your last task is to uncomment the commented out code in `wait_loop` in `box1.s`. This code should clear $a0, check if $a0 is equal to zero, and if not, print out the value stored in $a0. Typically, the code should never print out anything. However, if you run it, you should find that this is not actually the case. The problem is that the interrupt can occur at any point in time. Even if a register is being used directly after it is initialized, an interrupt can still change the value stored in that register.

Modify the interrupt handler so that this does not occur. Do not use the stack. (There are several reason for this: 1) typically the interrupt handler is in kernel space, and the stack is in user space - data that will only be used in the kernel space should be stored there, 2) the interrupt handler has no information about how the user program is using the stack, and 3) the interrupt may have occurred because the stack pointer is corrupt.)