Computer Architecture
Lab 6

In this lab, we'll continue to understand how the computer interacts with peripherals via control and data registers.

You should turn in the source file for the program you're asked to write. It should be submitted via the tool available on Sakai for this lab. It is due by the start of the next class period.

Interacting with Peripherals

We know how to write polling loops for the keyboard and monitor. This is great and all, but are we really going to use all our precious clock cycles staring at a single bit? That could be considered a waste of computing power.

What we need is *concurrency*. We need to be able to run two concurrent processes: **(1)** run the polling loop for the keyboard while simultaneously **(2)** playing Fallout 4 or whatever else needs the full CPU resources.

In the lecture, we're studying digital logic. Using digital logic, we can implement algorithms directly in circuits. So, what we can do here is take our polling loop and implement it directly in the hardware. We can then let that hardware detect the keyboard activity without consuming instruction cycles in the CPU and disrupting our hiking simulator.

This is how most computers are built: in addition to the main CPU that runs applications, they include specialty single-purpose circuits to do helpful things like stare at the single bit in the input control register and alert us when it goes high.

We can enable this circuit to run and we can write code to execute when the bit is logic-1. However, there is a bit of a catch here: since the hardware is looking at the bit, how do we know where to write our code? Normally we could write a funtion and use `jal` to call it. But, if the hardware is controlling the program flow instead of us, how do we call the function?

The answer is that the hardware is configured to always call the function located at a specific memory address. So, we have to write our function and place it at a specific address.

In MIPS, that address is `0x80000180`, which happens to be in the kernel (the area reserved for core OS operations.) We place code in the kernel using the `.ktext` directive, and we can follow the directive with an address to put the code in a specific location. So, we write `.ktext 0x80000180` before our function that processes the keyboard input.

The code to echo the keyboard input to the console itself is straightforward: we move the byte from the `RDR`, place it in `$a0`, and call `print_char`. This is exactly what we did in the last lab.

We end the function with the instruction `eret`. (Since the function was called by the hardware, it has a special register where it puts the return address and our normal `jr $ra` won't work, so we need to use a specialized instruction that is hardwired to the hardware function return register.)

So, there we go! We've written the hardware-called function to process keyboard input.

Note that this function can be called at any time. It will interrupt normal operations and the programmer has no control over when the hardware might choose to call it, as the user can press a key on the keyboard at any time. Therefore, any register you use ought to be saved and restored, else you can encounter really impossible to track down (even impossible to replicate!) errors.

Because the hardware is able to interrupt any instruction anywhere, we call these **interrupts**. The function you wrote is called an **interrupt handler**. Working with interrupts and writing interrupt handlers is pretty much a defining characteristic of programming at the low level. In an Operating Systems course, you get into much more detail on how to do all this, because managing interrupts is a core OS duty.

Enabling Interrupts

We have to do one more thing. We have to **enable** the interrupts. By default, this functionality is disabled. We have to specificially enable it in order for our interrupt handler to ever be called. (This is so that the computer isn't trying to call nonexistent interrupt handlers if the programmer is not using interrupts.)

Enabling interrupts in MIPS is a two-step process. We need to set bits in both the peripheral control register and in the global interrupt control register.

Bit 1 in the `RCR` will enable the keyboard interrupt. So, in your `main` function in the `.text` area of your code, write code that sets this bit. In the polling loop, you wrote code that looked at bit 0 of the `RCR`. For this code, we are going to set bit 1 of the `RCR`.

Bit 11 of the global interrupt control register must also be set to enable the keyboard interrupt. This register is a bit hidden. Because the interrupts are detected via specialty hardware separate from the main instruction processor, the MARS simulator puts them in a tab called CoProc 0. You can see on the right where it lists all the MIPS registers there are actually three tabs: Registers, CoProc 1, and CoProc 0. Click the tab for CoProc 0. (For those who want a glimpse of Christmas Future, look at CoProc 1 and I bet you can figure out what we're in store for there.)

On the CoProc 0 tab you see four registers: `vaddr`, `status`, `cause`, and `epc`. For this lab, we only care about register `$12`, the `status` register. We can't access this register using normal instructions, not even via memory-mapping. Instead we must use the instruction pair `mfc0` and `mtc0` to access it (*move from coprocessor 0* and *move to coprocessor 0*, respectively.) Further, we can't even use the special name for the register, so we must use the register number directly in the code. So, we have to write `mfc0 $t0, $12` to load the value from `status` to `$t0` and then `mtc0 $t0, $12` to move the value back from `$t0` to the `status` register (note this instruction reverses our normal `dest` and `src` operand order, which just adds to the joy inherent in assembly programming.)

In between the move instructions, we write the code to set bit 11 to enable the keyboad interrupts. We call this a **read-modify-write** design pattern because we can't access `status` directly and must first *read* it into a register, *modify* that register, and then *write* the new result back to `status`. This is very common when directly working with hardware.

Put the code to enable the interrupts in your `main` function. Now we're set.

Testing the Interrupt

To make sure things are working, our `main` function needs to do something that we may interrupt. There are two options. The first option is to compute pi! That's fun. Write the code to compute digits of pi endlessly.

Or, if you'd rather not do that then you can try the second option. Let's just do absolutely nothing. End your `main` function with the following glorious line of code:

```
wait:    j    wait
```

This will branch….all the way back to itself. But, it will keep the CPU running and while it's busy "branching" you can test the interrupt code to see your keyboard input echoed to the console.

Enable the Keyboard and Display MMIO Simulator as we did before, make sure it's connected to MIPS, and you should see your key presses appear on the console.

This is how real computers process inputs of most every kind. They have interrupts enabled and the interrupt handlers are set up to drive the peripheral. If you ever work in the embedded domain, maybe tinkering with an Arduino or building a robot, you will do a lot of interrupt programming.