Computer Architecture
Lab 5

In this lab, we'll begin 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've used `syscall` to print to the console and receive info from the console. In previous classes, you've used `cin`, `cout`, and `fstream` objects to perform similar tasks. Now it's time to see how all of those things actually work. Fundamentally, at the hardware level, ***how does the computer interact with peripherals?***

The theme in architecture is that "everything has an address." This means, in general, anything you want the machine to do must be configured by setting bits in a storage location. This location doesn't have to technically be addressable, as it could be a register, but it does have to physically exist. **Programming means to set bits in physical storage to configure an electronic circuit.**

Just as we have special registers for executing the program, such as the PC and IR, we also have special registers for interacting with peripherals. We attach peripherals to **ports**, and each **port** has associated **status/control** and **data** registers. In this lab, we'll look at the common peripheral known as the keyboard.

The MIPS registers that govern receiving data from the keyboard, the ones that `cin` interacts with, are called the **Receiver Control Register (RCR)** and the **Receiver Data Register (RDR)**. We don't find them amongst our friends `$t1` and `$s4` on the MIPS data sheet, however. The RCR and RDR do not have 5-bit codes that fit nicely into register operand fields. Instead, these registers are **memory-mapped**: that is, they are accessible as if they were a memory location, and this means they have an address.

This choice of computer design is called **Memory-Mapped I/O** (MMIO). It's not required that we do things this way. We could always give these registers space in the register file. But, the cost there is increasing the number of bits required to encode registers, and we've discussed the effects that can have on the overall architecture. So, memory-mapped IO is a common and widespread solution to the problem of what to do with extra system registers that we don't want in the general-purpose file.

The addressess for the RCR and RDR are as follows:

RCR	`0xffff0000`
RDR	`0xffff0004`

The memory addresses beginning at `0xffff0000` are set aside for the memory-mapped I/O area and all the registers in there can be seen in terms of offsets from that base.

When the keyboard sends a keystroke to the computer, the ASCII value of the key is placed in the low 8 bits of the RDR and bit 0 of the RCR is set to 1 to indicate data is available.

To access the keystroke, the programmer must move the data from the RDR to another location every time RCR bit 0 is set. There is only one Data Register! If you don't move the data out of it every time a key is pressed, the data will overwrite and you'll miss keystrokes.

Writing the code to process this is about the lowest-level thing you can do in a computer. These programs are called **device drivers** and are vital for every single peripheral you hook up to your computer. While the specifics of the control and data registers vary from architecture to architecture, this general pattern is consistent across all computers.

It's hard to test all this in an emulator because the real keyboard is running through the real operating system and we can't overwrite the real driver. If we had an actual dev board to work on, we could attach the keyboard to a "naked" machine. But, we don't have that. So, the next best thing is to use the MARS built-in keyboard emulator interface.

Under **Tools**, click on **Keyboard and Display MMIO Simulator**. The lower window represents the keyboard. Type some characters into that and then hit "Connect to MIPS". Now the emulated control and data registers will be engaged via this emulated input. Only the characters you type into this window will interact with the emulated MIPS machine.

Lab Exercise

Write a driver for the keyboard in MIPS. Use a loop to detect when the RCR bit 0 is set, then move the data out of the RDR and into `$a0` so the `print_char` system call can echo the keystroke to the console. Then wait for another keystroke, ad infinitum.

You can store the addresses of the RCR and RDR as `.data` and access the registers themselves through `lw`. Use logic operations to isolate bit 0 of the RCR and a branch to repeat this loop until bit 0 is 1. (It's worth your while to look through the branch pseudo-instructions on pages A-59 through A-63 because there are useful ones in there—I'd recommend `beqz` for this program.)

You only need to get the low byte from RDR (because the keyboard data is an ASCII value.) So you can use `lb` or `lbu` instead of `lw`. You then need to put the character in `$a0` and put an `11` in `$v0` to `syscall` the `print_char` function.

Finally, wrap the whole thing in a loop by writing `j main` at the bottom of the code to return to the top (use a `main:` label for your entry point to the program.)

Open up the **Keyboard and Display MMIO Simulator** and hit **Connect to MIPS**. Then you should see the values you type in that display printing out on the console. Once you do that, congratulations! You have written your first device driver! Understanding how the low level of the computer works gives insight into what things like `cin`, `scanf()`, `scanner.next()`, `input()`, etc., are actually doing.