Foothill College
Computer Science Department
CS10 Computer Architecture and Organization
Lab # 9
Input/Output; Memory Hierarchy

1 Introduction

In this *optional* lab we are going to look at both polling and interrupt-driven I/O for MARS.

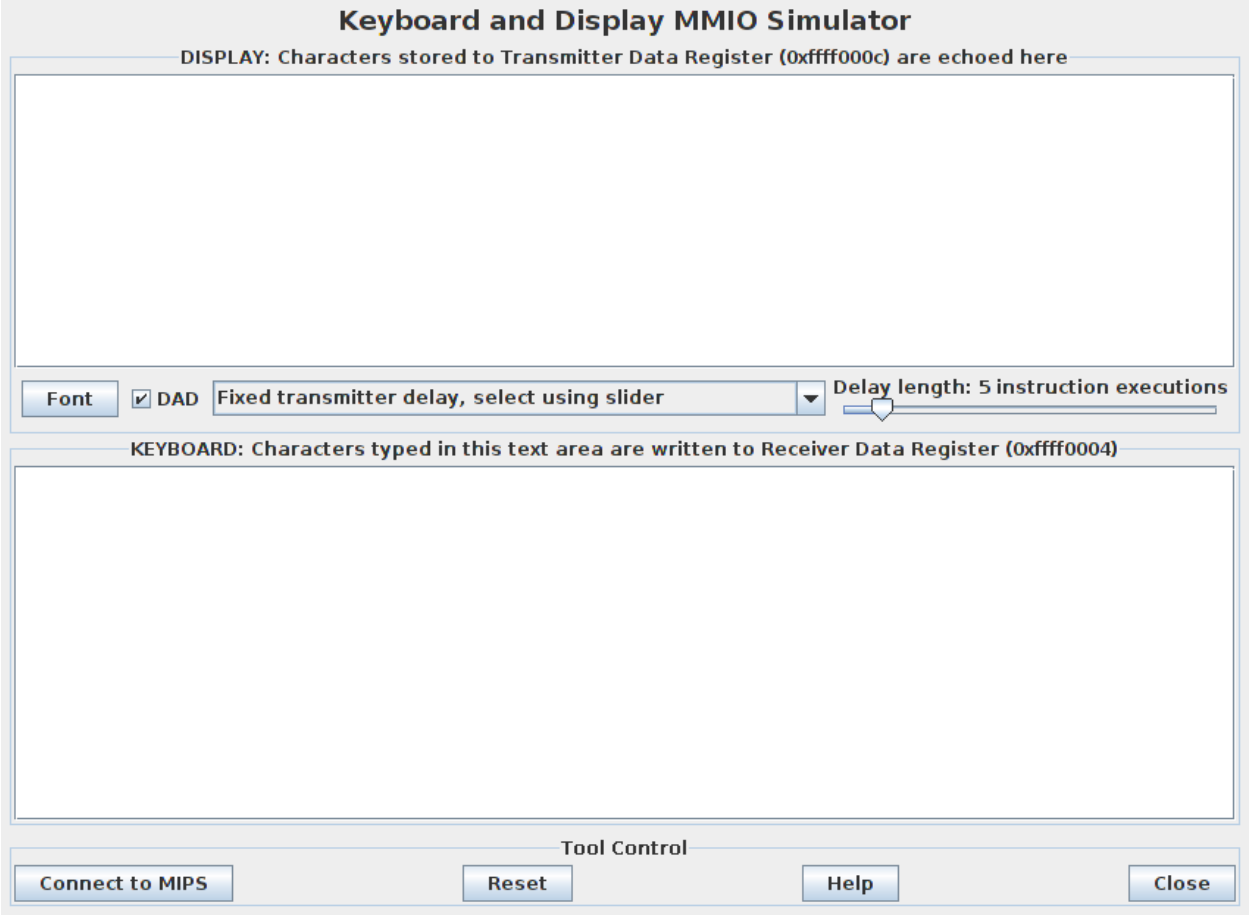
Task 1: Polling I/O

Here is a program, **w10uartio.asm**, that we can use as an example of polling for I/O in MARS.

This is a program which polls a UART-like device, and echoes input from a simulated keyboard back out to a simulated screen. Load it up into MARS and read the short amount of code. The program *polls* a keyboard control register until the register indicates that there is a character ready to read. Then the program reads it.

Similarly on output, the program polls a display control register until the register indicates that a new character can be accepted. When this is true, the program puts the character into the display data register, which has the effect of putting the character on the display.

Assemble the program, but before you run it, turn on Tools -> Keyboard and Display Simulator. This opens up a new window:



Click on Connect to MIPS before you run the program.

When the program is running, type characters into the white text area at the bottom of the above window, and you should see the characters echoed back to you in the display text area at the top of the window.

If things do not seem to be working, click on Reset to set the two devices back to their initial settings.

Something else to try is to run the program at a slower speed (i.e. 15 instructions/second). See if all the characters that you type make it out to the display.

2 A System Call Handler

Now let's work with an example of a system call handler. In task 1, we saw how to poll the Keyboard and Display Simulator device registers so that we could send and receive characters. In this example, we introduce two new system calls:

Syscall	Name	Purpose
104	myprint_string	Like print_string, but using the Display
108	myread_string	Like read_string, but using the Keyboard

Note: A user-mode program is technically not allowed to access the Keyboard and Display Simulator device registers directly, so the proper way to do I/O on this device is to create some new system calls which hide the actual I/O but provide a nice API. Therefore, we now have two new syscalls, 104 and 108, which work exactly like *print_string()* and *read_string()*, but they use the Keyboard and Display Simulator.

To demonstrate this, we need two assembly files:

1. A user-mode program which makes the system calls.
2. A kernel-mode system call handler which does the work.

In the following tasks, we load both files and them run them together.

2.1 Task 1

This task requires a bit of set-up, so make sure you follow the instructions below carefully.

1. Download this file, but for now do **NOT** load it into the MARS simulator: **w10syscallhandler.asm.**
2. Using the instructions from lab8 set the instruction handler to *w10syscallhandler.asm*.
3. Download this program, **w10usenewsyscalls.asm**, load it into MARS and assemble it.
4. Turn on Tools -> Keyboard and Display Simulator. Connect the simulator to MARS and do a Reset, just in case.
5. Now you should be able to run the program. The program prints a prompt string to both the Display Simulator and the MARS console. You get to type in a line of text followed by the Enter key. Finally, the program re-prints your line of text to both the Display Simulator and the MARS console.

If you want to run the program several times, it is usually a good idea to reset the Keyboard and Display Simulator each time.

2.2 Task 2

If you look at *w10usrnewsyscalls.asm*, it is a pretty normal user-mode program using system calls, except that some of them are repeated to show that the new syscalls work just like the old ones.

The complicated file to look at is *w10syscallhandler.asm*, so we will do it in stages.

- There is the syscall handler code, which goes from the `handler:` label to the `eret` instruction.
- There is a function called *myprint_string()*. It has been written just like an everyday function, including doing the stack frames and such. That is because even kernel-mode code should be written just like normal. What it does is poll the send control register until a new character is ready, and then sends a character. This is repeated for all the characters in the input string.
- There is a second function called *myread_string()*. It, too, is a normal function which runs in kernel mode. Again, it polls the read control registers, reads characters when they are there, and fills the input buffer with characters.

The important thing to note here is that all the code so far is normal code, and it could even have been translated from a high-level language like C or Java. When we write the code for an operating system, we write as much as possible in the normal way. The only code which has to be hand-written in assembly code is the actual system call handler, i.e. the code from `handler:` down.

The whole point of this example is that it demonstrates that we can write our own system call handlers in MARS, but we do need to deal with the events surrounding the system call, i.e. the change from user mode to kernel mode and back again.

3 Outlook for the Next Lab

This is our last lab for this quarter in CS10. *Thank you for all your hard work.*