University of Puerto Rico – Mayagüez
Department of Electrical and Computer Engineering

INEL4206: Microprocessors[1]
Problem Set #5
Interrupt-Driven Memory-Mapped I/O in SPIM/XSPIM

Due Friday, May 17, 2002

Objectives

The objective of this assignment is for the student to become familiar with the basic principles of interrupts and memory-mapped I/O. Although SPIM/XSPIM is missing a (simulated) hardware timer, and consequently timer interrupts, we will simulate a timer using interrupt-driven keyboard input. Specifically, we will use character inputs from the keyboard (or, equivalently, from stdin via a Unix pipe) to generate pseudo-timer interrupts, to reset the timer, and to quit the program. Interrupts and memory-mapped I/O with real hardware are more complicated than in the SPIM/XSPIM programming environment, but we can still experiment with the basic concepts and mechanisms.

References and Procedures

To complete all parts, you have to use (and understand) features of the SPIM/XSPIM system and MIPS assembly language. An on-line version of Appendix A (Assemblers, Linkers, and the SPIM simulator) is available.

Note that, even with memory-mapped I/O activated within SPIM/XSPIM (i.e., -mapped_io on the command line), the various syscall-based I/O calls continue to be available to your program. Unless told otherwise (read carefully below), your program may use these syscall functions.

You may download the SPIM/XSPIM system and work on any system. However, your final code must work with the version of SPIM installed on the workstations in the Amadeus computing lab. Be aware that there are known differences in how SPIM works on big-endian machines (e.g., Sun SPARC workstations) and little-endian machine (e.g., Intel-based workstations). **Programs that do not work with the SPIM installed on the Amadeus machines will be considered incorrect and they will lose marks.**

As per your course outline, all work in this assignment (and this course) must be **individual work.** Do not work in groups. Do not collaborate.

---

[1] This Problem set is a slight modification of a Problem set used for the Cmput 229: Computer Organization and Architecture I class at the University of Alberta, Canada.

What You Have to Do

Write an exception and interrupt handler (hereafter called the *handler*) that handles interrupts generated by the keyboard component of the terminal device in SPIM/XSPIM. Your handler does not have to handle any other interrupt nor any other exception. Your handler must:

1. Be saved in a file called ps5.trap.handler.s. This file should **not** have a main function.
2. Contain a .ktext 0x80000080 segment, just like the default trap.handler.
3. Be able to handle interrupts from the keyboard device, as per the requirements of memory-mapped I/O under SPIM/XSPIM. The details of how to handle the interrupt are discussed below.
4. Must contain a global label __start which calls the label main, as per the default trap.handler.
5. Within the code at global label __start, the interrupts for the keyboard must be properly enabled. Be sure to properly label this portion of your code with a comment formatted as such:
6. #-------------------------
7. # *** Enable Interrupts ***
8. #-------------------------

Running Main Program testmain.s and Your Handler

Consider a main program, like the one contained in attached file *testmains.s*. If you ran the program with the default trap.handler, you get:

```
% spim -file testmain.s
SPIM Version 6.3a of January 14, 2001
Copyright 1990-2000 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/local/share/spim/trap.handler
Hello, I like to add up numbers.
820 is the sum I get.
820 is the sum I get.
820 is the sum I get.

...stopped with Control-C...
```

The program is simple: It adds up a list of numbers and prints out the sum of the list. After it prints the sum, the program loops back (infinitely) and re-adds the same list of numbers and prints the same output. To complicate things just a little, there is a simple delay loop within the loop that adds up the list of numbers. This simple program represents a compute-intensive program. When an interrupt occurs, the CPU should execute the exception and interrupt handler and then return to this program. We will take **your** ps5.trap.handler.s file and concatenate it with **our** main program, like the sample above. Then, we will run the combined file as follows:

```
% cat lab5.trap.handler.s testmain.s > lab5.trap.s
% spim -mapped_io -notrap -file lab5.trap.s
SPIM Version 6.3a of January 14, 2001
Copyright 1990-2000 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Hello, I like to add up numbers.
820 is the sum I get.
820 is the sum I get.
820 is the sum I get.

...etc...
```

Note that both ways of running the main program result in the same basic output, assuming there is no input from the keyboard or stdin. However, if there is input from the keyboard or stdin, SPIM/XSPIM will generate an interrupt for each character and the inputs have to be handled as follows.

Responding to Interrupts and Input

When your handler receives a keyboard interrupt, it must use memory-mapped I/O techniques to **read** a single ASCII input value from the keyboard. You **cannot** use syscall to read in the character of input from the keyboard.

You must label the code that reads in the character from the keyboard with:

```
#---------------------------------
# *** Read Input via Data Register ***
#---------------------------------
```

You will use input from the keyboard to implement a simple timer. When SPIM/XSPIM starts running, the time will be 0 minutes and 0 seconds. Each time there is input to SPIM/XSPIM from the keyboard, the timer is affected.

In particular, only 3 ASCII characters are valid: t, r, and q. All other characters should be ignored. In response to each character, your handler must do the following:

1. **"t" is for Tick**. In response to receiving a t (i.e., the lower-case letter "t"), the handler increases the timer value by one second. Your handler should also print out the current time. Valid examples of the time output are discussed below. In essence, a keyboard interrupt with the input t is equal to a traditional timer interrupt, with an interval of 1 second.
2. **"r" is for Reset Timer**. In response to receiving a r (i.e., the lower-case letter "r"), the handler resets the timer back to 0 minutes and 0 seconds.
3. **"q" is for Quit**. In response to receiving a q (i.e., the lower-case letter "q"), the handler exits SPIM/XSPIM using syscall with code 10.

NOTE:

- ?? Any samples of input and output in this assignment description are merely examples. Of course, your program should work properly for all valid inputs. We will be testing your program with other inputs, main programs, etc.

The Program theClock

You can find a simple C program called theClock (in file ticktock.c) which generates a single output character t each second. Note the sleep(1) command in the C program; it spaces the output generated by 1 second intervals. By combining the output of this program with your handler, via a Unix pipe, we can use theClock to generate a regular pseudo-timer interrupt. You can run the program as follows:

```
% gcc ticktock.c -o theClock
% ./theClock | spim -mapped_io -notrap -file lab5.trap.s
SPIM Version 6.3a of January 14, 2001
Copyright 1990-2000 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Hello, I like to add up numbers.
Time: 0:01
Time: 0:02
820 is the sum I get.
Time: 0:03
Time: 0:04
Time: 0:05
820 is the sum I get.
Time: 0:06
Time: 0:07
Time: 0:08
820 is the sum I get.
Time: 0:09
Time: 0:10
Time: 0:11
820 is the sum I get.
Time: 0:12
Time: 0:13
820 is the sum I get.
Time: 0:14
Time: 0:15
Time: 0:16
...program stopped...
```

Each second, theClock sends a single character t down the pipe to SPIM. Your handler gets an interrupt and interprets the character t as a clock tick and advances the clock by one second. Your program also prints out the time in the Time: 0:08 format. So, Time: 0:08 represents 8 seconds since the start of the program, or the last reset. And, Time: 1:00 (not shown) would represent 60 seconds since the start of the program, or the last reset.

Note how the output for the timer (e.g., Time: 0:08) and for the main program (i.e., 820 is the sum I get.) are interleaved. Sometimes, 3 seconds pass between the output lines of the main program, and sometimes 2 seconds pass between the output lines. The exact interleaving depends on the speed of your workstation and the activity of other processes on the computer: the Time: outputs are always 1 second apart and and the other outputs can vary depending on the CPU load. Of course, the main program is completely unaware that interrupts are occurring. And, the handler is always careful to return execution properly to the main program, after the interrupt is handled.

Hints

You are free to ignore these hints, but we strongly believe that they are worth your consideration.

We will be using testmain.s and theClock to test your programs. We will also be using other main programs and variations on theClock. Therefore, you should write your own main programs and play with the code for theClock as part of your own testing process.

We recommend that you write the solution in stages. In the long run, it takes more time to try to implement all of the functionality on the first attempt. **Part marks will be given** for significant, but incomplete, functionality according to the following recommend intermediate stages.

1. Either write a new handler from scratch, or modify the default trap.handler, such that interrupts are properly enabled. When a keyboard interrupt occurs, print out a simple string that says, "Hey, I got and recognized an interrupt."
2. Extend your handler to use memory-mapped I/O (i.e., read the data register) to read the input character from the keyboard. Echo this character out as a debugging aid; make sure you read in the correct input.
3. Extend your handler to keep track of time. Support the proper action when the input character is a t. Write the code to output the current time in the proper format.
4. Extend your handler to support reset (i.e., r) and quit (i.e., q).
5. Test with testmain.s and other main programs. Test, test, test.