Laboratory 7: Prelab

Date ______________________ Section ______________________

Name ______________________

Making Sense Of An Exception Handler

During the prelab you will study an existing exception handler, the one that comes with the SPIM simulator. You will then be able to write one on your own (this will be the inlab task).

Introduction

When an exception occurs, the processor enters the kernel mode and control is transferred at address `0x80000080`. Part of the invocation process is disabling interrupts: the hardware takes care of it. This is needed since the exception handler has to save the state of the processor before doing any processing for that interrupt. At the very least, the processor must be able to execute uninterrupted long enough to

- find out what caused the exception and save the relevant information (EPC, the Cause register, the Status register)
- save the state of the processor (actually save all registers the trap handler may use)

Disabling interrupts does not mean they are eliminated. It only means their effect is deferred. Since interrupts inform the processor about significant events, one may not indefinitely disable them. But it is safe to ignore interrupts for short periods of time. At the time interrupts are enabled again, the hardware will treat any pending interrupt as if it just occurred and immediately invoke the exception handler.

As noted before, the exception handler is invoked with the interrupts being disabled. However, there is nothing to prevent traps from occurring within the handler. It is the programmer's responsibility to make sure the exception handler does not execute any instruction that cause an exception.

If the exception handler returns control to the interrupted program after a short period of time, then it may run without enabling interrupts. In this case there is no need to save the exception information (EPC, Cause, Status). The registers the handler uses must be saved. Of course the two dedicated registers, **`$k0`** and **`$k1`**, need not be saved and can be used right away.

If the exception handler contains code that may run for long times, then it would be unsafe to execute that code with the interrupts disabled since some events may be lost. Imagine for instance the exception handler executing code for a communication interface and ignoring an interrupt that indicates a power failure. In general, a correctly written exception handler should be itself interruptible. Such a handler is said to be **reentrant**.

If interrupts are enabled within the exception handler, then the relevant information (EPC, Cause, Status) must be saved before enabling interrupts exception. All registers the handler may use must be saved anyway.

© 1996, Virgil Bistriceanu

Two things must be properly set to enable interrupts:

- IEc (bit 0 in the Status register) must be 1: this indicates that interrupts are allowed in the current state
- the interrupt mask (IM) which indicates what specific interrupt(s) to enable

The IEc bit enables or disables all interrupts while the individual bits (IMi) in the interrupt mask enable or disable individual interrupts. Either can disable an interrupt but both (IEc and IMi must be set for interrupt i to be enabled).

The Interrupt Pending bits (IPi) in the Cause register indicate whether or not an interrupt is pending. Note here that an IPi bit in this register will be set, thus indicating a pending interrupt, even if the interrupts are disabled. This way the exception handler can determine whether there are pending interrupts which would cause an exception had they not been disabled.

Step 1

Begin by making a copy of the file called *trap.handler*: call the copy *trap.handler.original*. You will work on the file *trap.handler* which the SPIM simulator loads by default. Look at the file and, based on what you see, try to answer a few questions. The '.set noat' assembler directive instructs the assembler not to complain about using the **`$at`** register (which is reserved for the assembler's usage).

Q 1:

An exception handler needs to save the registers it may change. Can this be done on the stack, using **`$sp`**? Explain.

Q 2:

Is the exception handler you find in the file *trap.handler* re-entrant or not? Explain. A possible comment in the code that says the handler is not re-entrant is not a good argument! There are several possible explanations, but you need provide only one.

Q 3:

Since a `syscall` creates an exception, do you think it is appropriate to have `syscalls` inside an exception

© 1996, Virgil Bistriceanu

handler? Explain.

Q 4:

Is there any code in the exception handler that actually does print an integer (print_int)? True, there are calls to print_int, but where are they handled?

Q 5:

Why is there a need to check whether the EPC is word aligned?

Step 2

As you can see the exception handler does not worry about interrupts. We ignore them for now. What the handler does is that it only prints out a message indicating what happened and then returns. The message SPIM prints has two parts, the first part comes from the simulator itself and the second is printed within the exception handler.

Write a program called *lab7.1.asm* which:

- declares a variable called MAX_INT with the initial value the maximum value of a signed integer
- loads the value of MAX_INT in register **`$t0`** and adds it to itself (use signed addition)

Q 6:

Will your program, when run, generate an exception? If you answer yes, then indicate whether it is a trap or an interrupt.

© 1996, Virgil Bistriceanu

Run *lab7.1.asm* and write down the error message you get on screen. For each line in the error message clearly indicate the source in the 'Source' column: use a S to indicate the message is generated inside the simulator, and a T to indicate it is generated inside the trap handler.

Error message generated by *lab7.1.asm*

Error message	Source

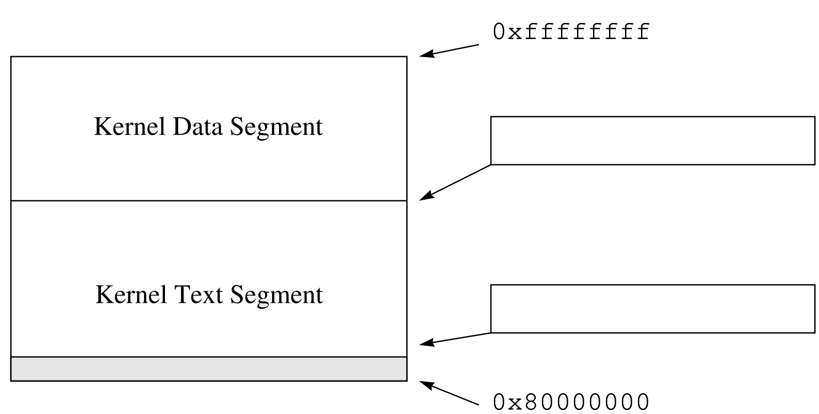
If you don't like having SPIM print a message every time an exception is generated, then you can run the simulator in quiet mode: use the `-quiet` flag in the command line to do so. In this case the only error messages will be the ones printed from within the exception handler.

Step 3

Step through *lab7.1.asm* until you enter the exception handler. Some instructions load data from the kernel data segment. Based on what the simulator prints when you step, find out at what address the data segment starts. Show your work.

Step 4

In the Laboratory #2 inlab exercise, the model of MIPS systems memory has been presented. However, there are no details about the kernel space. A more detailed image is presented below. Fill out the missing information on this figure.



© 1996, Virgil Bistriceanu

Step 5

The trap handler prints the appropriate error message for each exception by using the exception code from the Cause register as an index in a table containing message addresses. At this step you are required to identify the table and show a few entries in it. In the 'Label' column, the first row will be the label where the table begins. In the 'Comment' column indicate what is the content stored at each of the addresses in the column 'Address'.

Label	Address	Contents	Comment

Step 6

Printing inside the exception handler is done by using `syscall`. However, the code that actually prints a character on the console is not there. SPIM uses the services of the machine it is running on to do the actual character output or input.

We now want to see how it is like to do it, using the bare devices the machine offers. SPIM simulates a memory-mapped terminal connected to the processor. Memory mapped means that accessing some memory locations accesses the I/O devices in reality. Writing to a specific memory location will actually write to an output device, while reading from some specific memory location actually reads from an I/O device. The terminal device SPIM simulates consists of two independent units: the *receiver* and the *transmitter*. The receiver reads characters from the keyboard as they are typed. The transmitter unit writes characters to the terminal's display. Both the receiver and the transmitter can work in interrupt mode. At this step, however, you will do something simpler: you will modify the exception handler as to print using the transmitter unit. Modify the exception handler as follows:

- write a procedure called *PrintString* which receives in **`$a0`** the address of the string to print, and returns no value. The string to print is null-terminated
- the procedure uses busy-waiting for printing
- call 'PrintString' in the exception handler instead of print_str

A few hints are probably in place:

- you will probably need to save more registers when you enter the exception handler than it does now; save them in kernel data segment
- busy waiting means that you have a loop where you test the 'Ready' bit in the Transmitter Control Register until it becomes 1, thus signaling that the output device is ready to accept a new character
- start the spim simulator with the `-mapped_io` flag in the command line
- the null symbol is the byte 0x00

© 1996, Virgil Bistriceanu

Use the program *lab7.1.asm* for testing. If you attempt to run the program and it hangs or produces exceptions other than what *lab7.1.asm* does when running with the unmodified exception handler, then you may want to go step by step and watch the content of relevant registers and memory.

© 1996, Virgil Bistriceanu