Interrupt-Driven I/O

Administrative details

Submit a solution to this project by noon on November 18.

The project should be done in a partnership. Everything you turn in should include the names and lab section times of everyone in your partnership. Put two files of program source code, one named iout.s and the other named iecho.s, in a directory named mips2 and submit it as you did for earlier assignments.

Background reading

P&H, pages 679–682 and section A.7 ; G&M, chapter 12. The frameworks for the iout.s and iecho.s programs were described in lab+homework assignment 10.

Project exercise 1

Fill in the body of the interrupt handler in iout.s. Do not modify either the print procedure or the main program. Here is a list of things the interrupt handler must do:

a. If the transmitter is not ready, then the interrupt handler should not do anything. (You shouldn't have gotten an interrupt in the first place if the transmitter isn't ready, but it's a good idea to check anyway; you'll need the check in exercise 2 below.)

b. If the output buffer isn't empty, the interrupt handler should copy the next character from the output buffer to the Transmitter Data Register and advance nextOut circularly.

c. If the output buffer is empty, the interrupt handler should turn off the "interrupt enable" bit in the transmitter control register. Otherwise continuous interrupts will occur. (The interrupt handler in G&M deals with this situation differently.)

d. The interrupt handler must save and restore any registers that it uses, even temporary registers like $8 and $9. This is necessary because interrupts can occur at any time and those registers could have been in use at the time of the interrupt. These registers are to be saved on the stack. The only exceptions to this rule are registers $26 and $27, which are reserved for use by interrupt routines; these registers need not be saved and restored. One of these registers, $26, is used to return from the interrupt routine back to the code that was interrupted.

Your code should output lines continuously, with each line containing the characters "Just wasting time". Debugging interrupt-driven software is very tricky. You can't always single-step to get to the point of a problem, because it may take a large number of instructions before an interrupt occurs. In these cases you'll have to set breakpoints at key places (like the beginning of the interrupt routine) and then single-step from there.

Project exercise 2

Extend the code from exercise 1 to handle interrupt-driven input. The file iecho.s described in lab+homework assignment 10 contains a skeleton for the program. This program does output in the same way as iout.s: there is a copy of the print procedure in iecho.s and you should copy your interrupt routine from iout.s to iecho.s. However, you'll need to add buffered interrupt-driven input to the program. It should work in the same fashion as the buffering in iout.s except that the roles of the interrupt and background routines are reversed: the interrupt routine will add characters to the input buffer and a background routine getchar will remove characters from the input buffer. You should do the following:

a. Define variables for an input buffer that are analogous to the buffer, nextIn, and nextOut variables used for the output buffer. The input buffer should only contain 8 characters worth of space in contrast to the output buffer's 32 characters. This will make it easier to test the "buffer-full" condition below.

b. Extend your interrupt routine to also check the receiver. If the receiver is ready, the interrupt routine should read the input character from the Receiver Data Register, place it in the input buffer, and advance the appropriate index variable circularly. If the input buffer is already full, then the interrupt routine should discard the character read from the receiver: don't add it to the input buffer.

c. Fill in the body of the procedure getchar. This procedure takes no arguments and returns the next character from the input buffer. If the input buffer is empty then getchar should just check the input buffer indices over and over again until eventually a character appears in the buffer. (In a real system like Unix the operating system will run a different user's process while waiting for a character to arrive.)

iecho.s already contains a main program to test both the input and output. The main program is an infinite loop: it calls getchar to wait for a character to be typed, then it places the character in the middle of a string, then it calls print to output the string. The result should be one line of output for each character you type. For example, if you type the character "z", the following output line should appear:

```
Received character 'z'
```

If you type some other character, the same line should appear with the "z" replaced by the character you typed. The program will not stop until you type control-C.

Try typing characters rapidly to make sure your program can handle the case where either the output buffer or the input buffer fills up. For example, if you type two or three characters rapidly the output buffer should fill up. However, no output should be lost: the print procedure will simply have to spin for a bit, during which time additional input characters will be buffered in the input buffer. If you type eight or ten characters very rapidly then the input buffer will fill up. When this happens your interrupt routine will have to discard characters: the program should continue to function but there won't be any printout for the discarded input characters you typed. Once the output catches up with the input your program should accept input again just as if the input buffer had never filled up. We'd suggest setting a stop at an instruction in your interrupt routine that is only executed when an input character is about to be discarded; this way you can be sure that the code is being exercised. In a real system like Unix the input buffer is much larger than 8 characters (256 characters is common in Unix systems) so that it virtually never overflows.

Modify the driver program so that some work is done between getting characters. For example, set up a register initialized to zero and do an (unsigned) add to it in a tight loop. Print out the value in decimal in that register along with the input character, and then set that register to zero. SPIM is quite slow compared to a "real" machine, so this is not a very good measure of how much can be done between key clicks.