EECS 373 Winter 2004

Lab 6: Basic Interrupts

Requirements

- **Pre-lab:** Prelabs due in your lab section the week of **February 23, 2004**. Your group must also have an initial software design completed and entered into SingleStep.
- In Lab: This is a two week lab. In-labs are due by Friday at 5:30PM the week of March 1, 2004.

Post Lab: In your lab section the week of March 7, 2004.

Value: This lab is worth 7% of your total grade.

Objectives

The purpose of this lab is to:

- 1. Reinforce your understanding of interrupts.
- 2. Give you experience writing and debugging simple interrupt service routines.
- 3. Show you how interrupts allow a machine to accomplish several tasks at once.

Overview

Interrupts allow a device to request service from the CPU when needed without requiring the CPU to waste time polling the device. In this lab, you will use interrupts generated by the push buttons and the MPC823's built-in real-time clock to update the seven-segment display while the CPU continues to run your calculator application from lab 5.

Specifically, you will combine your calculator program with a stopwatch controlled by the pushbuttons. Your program will maintain a timer, incremented every second, and continuously display its value on the seven-segment display. The timer should start at 0 when the program begins execution. Pushbutton S1 will alternately stop and start the timer. Pushbutton S2 will reload the timer value (and update the display) using the binary value on the DIP switches.

Because the CPU will be occupied running your lab 5 calculator application, you cannot poll the switches or real-time clock to determine how and when to update the display. Instead, you will configure the MPC823's real-time clock to generate an interrupt every second, and you will build hardware to cause the pushbuttons to generate interrupts every time they are pressed. Your interrupt service routine (and, outside of initialization, *only* your interrupt service routine) will control the seven-segment display.

Design Specification

Hardware

Your hardware for this lab will combine your LED and switch registers from lab 3, your memory module from lab 4, and some new hardware that uses the pushbuttons to generate CPU interrupts.

You will be using level-sensitive interrupt request inputs, so you must build hardware that remembers when a pushbutton has been pressed and continues to assert the interrupt request line until the CPU handles the interrupt, even if the pushbutton is released. The core of the new hardware will be two flip-flops, one for each pushbutton. Each flip-flop should be set when the corresponding button is pressed, but should not clear when the but-

ton is released. Pushbutton S1 should assert the $\overline{IRQ1}$ interrupt request input, and pushbutton S2 should assert the $\overline{IRQ7}$ input.

Once the CPU handles a pushbutton interrupt request, it must reset the appropriate flipflop to deassert the interrupt request signal. The flip-flops should be cleared by writing to a 32-bit device register at address 0x03100008. The two least-significant bits (bits 30 and 31) of that register should correspond to switches S1 and S2, respectively. Unlike a memory location, the value written to that address is not stored in the flip-flops. Instead, a 1 bit in the data value written indicates that the corresponding flip-flop should be cleared, while a 0 bit indicates that the corresponding flip-flop should be left unchanged. This protocol allows the CPU to clear individual bits without introducing any race conditions.

For debugging purposes, a read to the same address (0x03100008) should return the state of the two interrupt request flip-flops in bits 30 and 31.

Finally, you will use your 256-byte memory module from lab 4 (still located at 0x3500000) to store your program stack. This module should be large enough for a reasonable stack, but if your stack starts growing out of control (a common occurrence in buggy interrupt code) it is less likely to overwrite your program than if it were in the main memory module.

Software

Your program will have several separate components. They include the main external interrupt service routine, the individual device-specific interrupt handlers, and the main loop of your program.

The External Interrupt Service Routine

Because the PowerPC architecture provides a single external interrupt vector, all of your hardware interrupts will begin in the same ISR. The main part of the ISR is responsible for determining which device (the real-time clock, pushbutton S1, or pushbutton S2) caused the interrupt and calling the appropriate handler. These device-specific interrupt handler functions should be ABI compliant. The main part of the ISR is thus also responsible for saving and restoring any registers that may be overwritten by the device-specific handlers.

You should not use nested interrupts in this lab.

The Real-Time Clock (RTC) Handler

The RTC handler is the function that will be responsible for handling the once-per-second interrupts from the real-time clock. It should increment the timer value and update the seven-segment display.

The Pushbutton Handlers

These functions (one for each pushbutton) will be responsible for handling the interrupts generated by pressing either of the push buttons. See the program specification above for the appropriate actions. The handler must also acknowledge (clear) the interrupt by clearing the flip-flop associated with the button that was processed.

The Main Program

The purpose of this portion of the lab is to show that interrupts allow the processor to efficiently accomplish a number of functions simultaneously. The main portion of the program should be your unmodified code from lab 5. All changes for this lab should be limited to the initialization phase; if your interrupt handlers are written properly, their operation should be completely transparent to the code you wrote for lab 5.

Hardware Details

This section summarizes the key PowerPC and MPC823 registers you will need to use in this lab. Following the name and acronym for each register are the section or page numbers in the MPC823 User's Manual that discuss that register.

Many of the device registers you will use in this lab require only a few specific bits to be set to particular values. If you don't know exactly how other bits in the same register are supposed to be set, leave them unchanged. That is, you should read the register value, change only the bits you are concerned with, and then write the value back.

Machine Status Register (MSR) (p. 6-20)

In the MSR, you will need to manipulate the External Interrupt Enable (EE) bit and the Interrupt Prefix (IP) bit. Once all the appropriate device registers are set up and you are ready to begin handling interrupts, you should set the IP bit to 0 and the EE bit to 1. The MSR is accessed using the mfmsr and mtmsr instructions (see Appendix F of the green book).

Although the MPC823 User's Manual provides the layout of the MSR on p. 6-20, the descriptions it gives are useless. The PowerPC Programming Environments book (the "green book") provides much better descriptions of the MSR bits. See Table 6-5 in the online version on the course web page, which is Table 6-4 in the hard copies in the lab.

Internal Memory Map Register (IMMR) (p. 12-34)

Most of the MPC823's internal device registers are memory mapped. The value in the IMMR determines the base address of the memory-mapped MPC823 registers. By writing to the IMMR, you can relocate the MPC823's internal registers to a different portion of the system memory map. In this lab, you should not change the IMMR; it will be initialized by the MPC823FADS firmware and/or SingleStep. Instead, you should read the IMMR to determine the addresses of the memory-mapped registers. The IMMR itself is not memory mapped. It is a PowerPC special purpose register (SPR), number 638, and can be read into a general-purpose register using the mf spr instruction.

MPC823 System Interface Unit (SIU) Interrupt Controller (Section 12.3)

SIPEND – SIU Interrupt Pending Register (p. 12-7)

This register indicates the interrupts that are currently pending.

SIMASK – SIU Interrupt Mask Register (p. 12-8)

This register is used to enable the interrupts that you are interested in handling. In this lab, you are only interested in handling interrupts for the RTC (Level 0) and the pushbuttons ($\overline{IRQ1}$ and $\overline{IRQ7}$). You should set the bits corresponding to these interrupts, and clear the rest.

SIEL - SIU Interrupt Edge/Level Register (p. 12-9)

Set both to $\overline{\text{IRQ1}}$ and $\overline{\text{IRQ7}}$ to be level-sensitive. Since we do not use the MPC823's low-power mode in lab, the settings of the WMx bits do not matter.

SIVEC – SIU Interrupt Vector Register (p. 12-10)

This register indicates the highest priority unmasked pending interrupt (i.e., the highest priority interrupt whose SIPEND bit and SIMASK bit are both 1). You can use the value in the INTC field to vector into your own jump table, as illustrated in Figure 12-3, or you can load and test the SIVEC value directly to determine which device-specific handler to invoke. You can also ignore the SIVEC register entirely and read the SIPEND register directly to determine which lapproach is recommended for this lab.

MPC823 Real-Time Clock (RTC) (Section 12.7)

System Clock and Reset Control Register (SCCR) (p. 5-3)

This register configures several options related to various MPC823 clock signals. The two fields related to the real-time counter are RTSEL and RTDIV. Set these fields so that the RTC uses the main clock oscillator divided by four.

RTCSC - Real-Time Clock Status and Control Register (p. 12-18)

Several fields in this register have to be set up properly to generate a Level 0 interrupt every second. The RTCIRQ field controls the interrupt request level as follows: set bit 0 to cause a Level 0 interrupt, bit 1 to cause a Level 1 interrupt, etc. The main clock oscillator on the 373 lab boards runs at 32 KHz. The SEC bit in this register indicates that a onceper-second interrupt is being requested, much like your flip-flops indicate that a pushbutton interrupt is being requested. It must be cleared at startup and each time you handle the interrupt.

System Integration Timer Keys (Section 5.4.2.2, p. 3-4 and p. 5-27)

To prevent buggy software from accidentally writing certain critical registers that could seriously mess up the system, the MPC823 protects these registers using "key" registers. If you write to a protected register without first writing a special "key value" to the corresponding "key register", then the write will fail. The protected device registers you will use in this lab are RTCSC and SCCR. The corresponding key registers are RTCSCK and SCCRK, respectively. The key value for all of the registers is the same, 0x55CCAA33.

Design Notes and Hints

- The notes above summarize the key MPC823 registers you will need to use, but they
 do not tell you everything you need to know. You will have to look up additional information in the MPC823 User's Manual to learn how the RTC, the SIU interrupt controller, and other components work, and how to configure and control them using the
 registers listed above. A large part of this lab is learning how to read the data book.
- The linker is in charge of assigning specific addresses to your instructions and data. It does this on the basis of *sections* (a.k.a. *segments*), which are contiguous memory regions that you define in the assembler. The linker command file (e.g., lab2-lnk.txt) tells the linker where to put each section that you define. To set up your interrupt handlers, put each handler in a new section using the ".section" directive. For example,

.section ext_int_vec, text

will put the following code in a section named "ext_int_vec" of type text. Then edit the linker command file, adding a line like:

ext_int_vec ADDRESS 0xnnnn:

which will place the section "ext_int_vec" at address 0x*nnnn*. Note that the ".data" and ".text" directives you've used already are just shorthand for ".section .data, data" and ".section .text, text", respectively.

- Although the RTC maintains its own elapsed time in the RTC register (p. 12-19), you should maintain your own timer value in software. This will make it easier to start, stop, and reset your timer.
- There are several ways to "stop" your timer when S1 is pressed. You may either disable the interrupt in the RTCSC, mask it in SIMASK, or continue to handle the interrupt but use a software flag to keep from incrementing your timer value. The choice is yours.

- Debugging interrupt software is fairly complex for a couple of reasons. First, the point in your main program where an interrupt occurs is typically not repeatable. Second, the debugger itself uses the PowerPC exception mechanism (as discussed in lecture), and will interact with your ISR if you're not careful. The best place to start is to place a breakpoint inside the main part of the External Interrupt ISR to determine whether you are receiving interrupts at all. Remember that you must save and restore SRR0 and SRR1 before you can use the debugger in your ISR code. Be sure not to place a breakpoint *inside* the code that manipulates the SRRs.
- All the device registers are observable in the register window. You can use this feature to determine if you have set up device registers properly and interrupt status. The registers are grouped according to device category and are located just below the general purpose registers. Device registers are processor specific, so the MPC823 must be selected when downloading the ELF file, under the 'processor' tab in the download window.

Pre-lab Assignment

- 1. Draw a block diagram (like Figure 12-2 in the white book) showing the hardware components (both internal and external to the MPC823) and interrupt signal paths used in this lab. This block diagram should include the processor, the devices and the interrupt controllers that you are working with.
- 2. Write assembly code that reads the IMMR value and uses it as the base address to access the SIPEND register.
- 3. Determine the values will you need to write to the following registers to initialize interrupts correctly: SIMASK, SCCR, RTCSC, SCCRK, RTCSCK
- 4. Why is it important that the MSR[EE] bit is cleared automatically by the processor when an interrupt is handled?
- 5. Why does your software need to set the MSR[EE] only once at initialization, even though the MSR[EE] bit is cleared by the processor each time an interrupt occurs?
- 6. What is the minimum set of registers that needs to be saved in an interrupt handler when:
 - a. The interrupt handler doesn't call any functions, but uses r0, r3, r4, r30, and r31
 - b. The interrupt handler calls ABI-compliant functions, but doesn't use any registers itself
- 7. If you initialize the MSR[IP] bit to 0, at what memory location does the processor begin executing the external interrupt ISR?
- 8. How many instructions can you put at this address? What would happen if you put too many instructions there?
- 9. Consider what would happen if you simplified your pushbutton interrupt hardware by always clearing *both* flip-flops on a write to your device register (ignoring the data bus value). Describe a scenario where a button press could be completely missed by your program.

In-Lab Procedure

This procedure is designed to help you work your way through what can be a difficult design/debug process. It intentionally adds the smallest number of unknowns at each step,

and forces you to debug a single interrupt source at a time. Make sure that you have completed the current step before moving on to the next!

- 1. Add your new pushbutton interrupt hardware to your lab 4 schematic, but do not connect it to the IRQ1_BAR or IRQ7_BAR signals on the PROC macro. Verify in the simulator that the individual flip-flops can be read by the CPU without interfering with your other hardware, are set by pressing the pushbutton, and remain set until they are cleared by writing a '1' bit in the appropriate location. Be certain that if both bits are set but the CPU only writes a single '1' then only the appropriate bit is cleared.
- 2. Download your circuit and test and see if basic operations such as reads and writes still work.
- 3. Simulate your interrupt initialization code and ISR. You can test basic code functionality with SingleStep software, however, you can not simulate interrupts. Do not add in your lab 5 code; instead, the main part of your program should be a simple infinite loop.
- 4. SingleStep is configured by default to handle all exceptions that happen on the CPU. Before you run any of your interrupt code on the actual hardware, you need to tell SingleStep to ignore external interrupts and let the board handle them. To do this, enter the following command in the SDS command window: @ der = 0xFDE7400F. Note the space between @ and der.
- 5. Download this simplified interrupt code and place a breakpoint in your main ISR *after* the instructions that save SRR0 and SRR1. Verify that you are getting RTC interrupts. You will not be getting pushbutton interrupts since you have not connected your pushbuttons to the interrupt request inputs. What values do you see in SRR0 and SRR1? Do these values make sense?
- 6. Set a breakpoint on the very first instruction of your ISR, before you save SRR0 and SRR1. Run the program until you hit the breakpoint, then single-step through the part that saves SRR0/1. What values do SRR0 and SRR1 have now? What will happen if you continue executing your program at this point?
- 7. Debug your RTC interrupts completely before moving on.
- 8. Add the connection from S2 to IRQ7_BAR to your circuit and download it.
- 9. Modify your program to disable RTC interrupts and download it. You can now debug your pushbutton S2 interrupt without the RTC interrupts getting in your way. Debug the pushbutton interrupt as completely as possible before moving on.
- 10. Disconnect IRQ7_BAR from your hardware (tying it high) and connect S1 to IRQ1_BAR. Repeat steps 7 and 8. You may want to use one of the bar-graph LED segments to indicate whether the RTC is "on" or "off".
- 11. Reconnect IRQ7_BAR, but don't enable RTC interrupts. Debug your hardware and software with just the two button interrupts.
- 12. Re-enable RTC interrupts and test your program with all three interrupts together. At this point, your system should handle the seven-segment display completely as specified above. Note that your main program is still an infinite loop.
- 13. Replace the infinite loop with your code from lab 5. Download and debug.

Demonstration 1.1: Demonstrate your thoroughly tested system to the lab instructor.

Post-lab

- 1. Summarize the operation of your circuit and describe how you arrived at your design. Include a printout of your final schematic.
- 2. Include a well-commented listing of your program. Comments should include register usage, descriptions of all symbols, and explanations of all derived expressions.
- 3. Discuss any difficulties you may have had in getting your circuit to work correctly: what parts of the program were hard to write initially, what types of bugs did you have to fix, etc. Be sure to discuss any problems or potential problems in debugging caused by using interrupts.
- 4. One way to avoid adding flip-flops to the pushbuttons is to use an edge-triggered (rather than level-sensitive) interrupt request input. The IRQ*n* lines on the MPC823 can be individually configured as level- or edge-sensitive via the edge/level register (SIEL; see section 12.3.3.3). Read the description of the SIEL carefully, then read the discussion of the pending register (section 12.3.3.1).
 - a. How would your system change if you configured IRQ1 as an edge-triggered interrupt? Specifically consider the possible race conditions where one button is pushed, then the other is pushed before or while the ISR for the first is being executed.
 - b. Focusing on section 12.3.3.1, how do you think the logic inside the MPC823 that implements edge-triggered interrupts compares to the logic you built to drive the level-sensitive interrupt request line?
- 5. Another approach to this lab would be to connect both pushbuttons to a single interrupt request line (e.g., IRQ1). How would this change affect both your hardware and your software? Be detailed enough to convince us (and yourself) that you could make it work. Discuss the advantages and disadvantages of this approach compared to having the pushbuttons connected to separate interrupts.

Lab 6 Demonstration Sheet

Print this page and present it to your lab instructor when demonstrating the various lab sections. Turn this sheet in with your post lab or when your in lab demonstration is due. You are required to turn in only one demonstration sheet per group.

List Partners Names

D1.1 Demonstrate operational system.

Lab instructors initials: