ECE 243S - Computer Organization
February 2024
Lab 5

Hex Displays and Interrupt-Driven Input/Output

Due date/time: During your scheduled lab period in the week of February 12, 2024. [Due date is not when Quercus indicates, as usual.]

Learning Objectives

Interrupt-driven I/O is a fundamental way that all processors synchronize with the outside world. The goal of this lab is to understand the use of interrupts for the NIOS II processor, using assembly-language code, and to get some practice with subroutines, modularity, and learning how to use the HEX displays on the DE1-SoC.

To do this exercise, you need to be familiar with the interrupt processing mechanisms for the NIOS II processor. Interrupts are part of a broader class of events called *exceptions*, and we have described the concepts in class. You can also find a detailed description of the exception handling in the Chapter 11 of the document *Introduction to the Intel NIOS II Soft Processor* which was provided as part of Lab 1 of this course.

There is some complexity to the proper arrangement of code to prepare for and handle interrupts, and so we have provided a fair amount of example code, in the lectures and here in this lab.

What To Submit

You should hand in the following files prior to the end of your lab period. However, note that the grading takes place in-person, with your TA:

- The assembly code for Parts I,II, III and IV in the files `part1.s`, `part2.s`, `part3.s` and `part4.s`.

Part I

You used the Hex 7-segment displays as an output in the ECE 241 course, but so far not in this course. In the files provided with this lab, we have given you a subroutine, called `HEX_DISP`, (in file `HEXdisp_subroutine.s`) which will display any 4-bit hexadecimal digit on any one of the six Hex 7-segment displays, HEX5 - HEX0.

The subroutine takes in two parameters: the low-order 4 bits of register `r4` give the 4-bit value to be displayed (which turns into one of the 16 hex digits, 0->F), and the number in register r5 says which of the six HEX digits to display that digit on. The display can be blanked by turning on bit 4 of r4 (that is the fifth bit, counting from 0).

It may be useful to know that storing a bit pattern to the memory-mapped address for the hex displays causes that pattern to be put into a memory-mapped register that then drives the display as you likely did in the hardware you designed in ECE 241. Figure 1 shows those registers and how they connect to the display. You don't need to know those details, though, because the provided subroutine deals with them for you.

The goal in this part is to write an assembly program that demonstrates the full functionality of the provided `HEX_DISP` subroutine. Your program should *not* use interrupts, and it should be the shortest, simplest demonstration that proves that the subroutine works as advertised. Notice that this specification is intentionally open-ended, and you should not ask for a more detailed specification, but take it as part of the task to determine those specifics.

Another motivation for this part is to get you used to testing individual parts of software, including code like this that is given to you. For your whole technical career, someone will be handing you, or you will be using someone

else's software, and you should always be skeptical of that software until you've convinced yourself that it actually works.

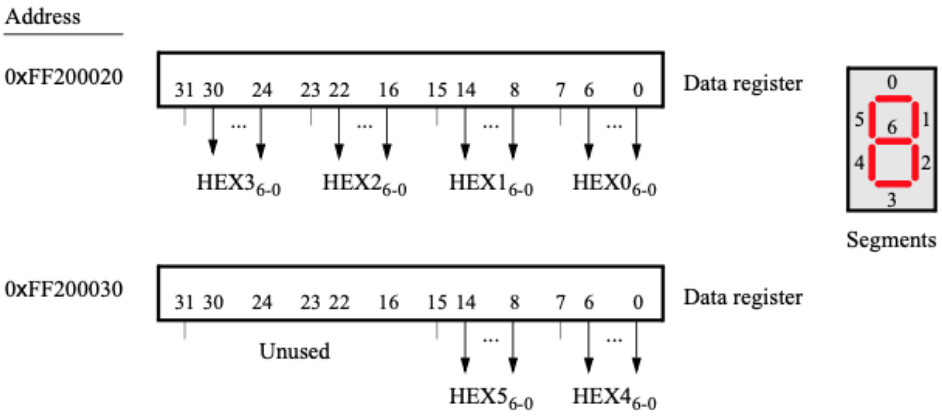

Figure 1: The DE1_SoC Hex Display

Create a new folder to hold your solution for this part and put your code into a file called `part1.s`. You will need to make a Monitor Program project for running your code on a DE1-SoC board; but you can also debug your program at home using CPUlator. Show it working to your TA for grading in the lab. Submit `part1.s` to Quercus before the end of your lab period.

Part II

In this part, you will write a program that will display specific numbers on the *HEX*0 to *HEX*3 displays, in response to the press and release of the four KEY pushbuttons. You must make use of the `HEX_DISP` code given in part 1, and unlike Lab 4, where you used polled-input to handle the KEYs port, in this exercise, you will use interrupt-driven input.

The specific behaviour is as follows: when KEY_i (i = 0 .. 3) is pressed and released, then display HEX_i will be set to display the number i. When KEY_i is again pressed and released, then display HEX_i will become blank. Every subsequent press and release toggles that number i on and then blank, and so on. For example, pushing KEY_3 would turn on the number 3 and then blank on the next press/release.

Consider the main program shown in Figure 2, which is also provided to you in the file `part2_skeleton.s`. The code first defines an Interrupt Service Routine (ISR) called *IRQ_HANDLER* using a code section called *.exceptions*. In this ISR, the code checks if the interrupt is from the KEYs port, and if so, it calls the subroutine *KEY_ISR* to handle the interrupt. Then, the *.text* section the main program initializes the stack pointer, configures the pushbutton KEY port to generate interrupts, and finally enables interrupts in the processor. You are to fill in the code that is not shown in the figure, which is also provide in the files associated with this lab.

After setting up interrupts for the KEYs port, the main program in Figure 2 simply "idles" in an endless loop. Thus, controlling the *HEX*3-0 displays should happen in *KEY_ISR*.

Do the following:

1. Create a new folder to hold your solution for this part. Make a file called *part2.s*, and type your assembly language code into this file. You may want to begin by copying the file named *part2.s* that is provided with the design files for this exercise.

2. Note that the Monitor Program tool supports multiple files in a project, which means that you could organize parts of your code, such as the main program, ISRs, and so on, in different files. However, this approach cannot be used with the CPUlator tool, which supports only one file at a time. For CPUlator you'll need to put all of your source code in *part2.s*.

3. When making your Monitor Program project you have to specify that Exceptions will be used as part of the code. This is done by selecting Exceptions in the *Linker Section Presets* screen illustrated in Figure 3. Be sure to make this selection for all of your Monitor Program projects in this Lab Exercise, or else your programs will not be able to be assembled properly.

4. The top part of Figure 2 gives the code required for the interrupt handler, called *IRQ_HANDLER*. You have to write the code for the *KEY_ISR* interrupt service routine. Your code should show the digit 0 on the *HEX*0 display when KEY_0 is pressed, and then if KEY_0 is pressed again the display should be "blank". You should toggle the *HEX*0 display between 0 and "blank" in this manner each time KEY_0 is pressed. Similarly, toggle between "blank" and 1, 2, or 3 on the *HEX*1 to *HEX*3 displays each time KEY_1, KEY_2, or KEY_3 is pressed, respectively. Compile, test and run your program.

```
/*********************************************************************************
 * Write an interrupt service routine
 ********************************************************************************/
.section .exceptions, "ax"
IRQ_HANDLER:
        # save registers on the stack (et, ra, ea, others as needed)
        subi    sp, sp, 16          # make room on the stack
        stw     et, 0(sp)
        stw     ra, 4(sp)
        stw     r20, 8(sp)

        rdctl   et, ctl4            # read exception type
        beq     et, r0, SKIP_EA_DEC # not external?
        subi    ea, ea, 4           # decrement ea by 4 for external interrupts

SKIP_EA_DEC:
        stw     ea, 12(sp)
        andi    r20, et, 0x2        # check if interrupt is from pushbuttons
        beq     r20, r0, END_ISR    # if not, ignore this interrupt
        call    KEY_ISR             # if yes, call the pushbutton ISR

END_ISR:
        ldw     et, 0(sp)           # restore registers
        ldw     ra, 4(sp)
        ldw     r20, 8(sp)
        ldw     ea, 12(sp)
        addi    sp, sp, 16          # restore stack pointer
        eret                        # return from exception

/*********************************************************************************
 * set where to go upon reset
 ********************************************************************************/
.section .reset, "ax"
        movia   r8, _start
        jmp    r8

/*********************************************************************************
 * Main program
 ********************************************************************************/
.text
.global  _start
_start:
        /*
        1. Initialize the stack pointer
        2. set up keys to generate interrupts
        3. enable interrupts in NIOS II
        */
IDLE:   br  IDLE
```

Figure 2: Main program and interrupt service routine for Part II.

Show your code working to your TA for grading in the lab. Submit `part2.s` to Quercus before the end of your lab period.

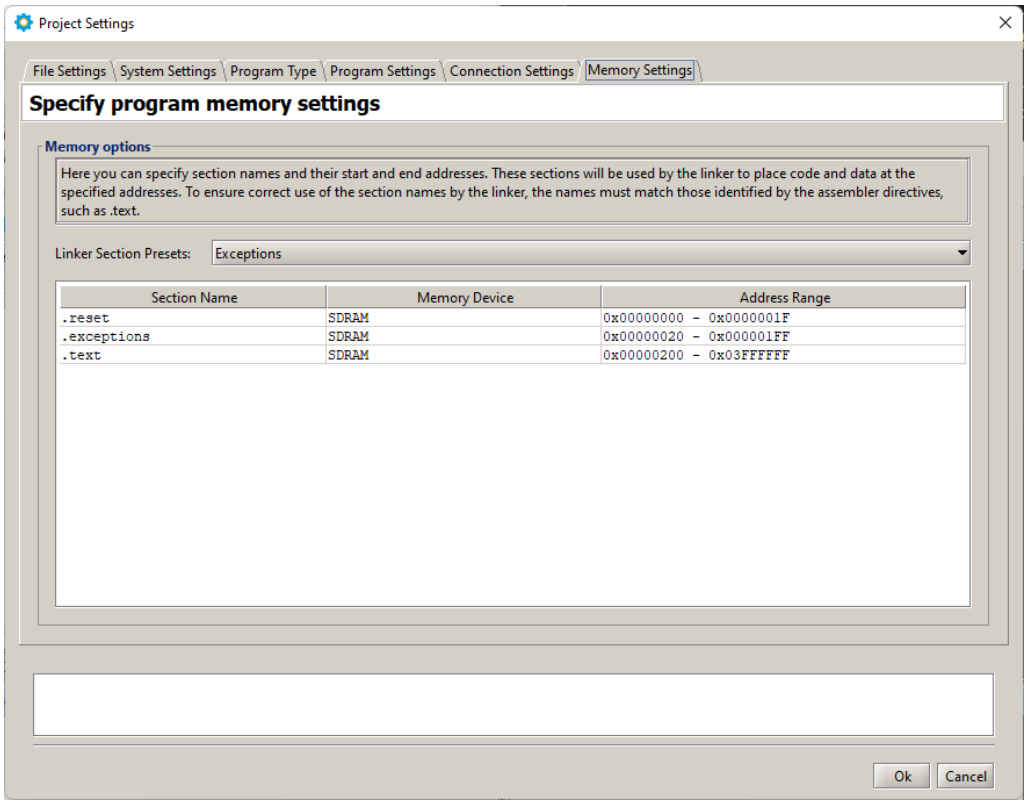

Figure 3: Selecting the Exceptions linker section.

Part III

In this part you'll build an interrupt-based program that controls the LEDR lights, in a manner similar to Lab 4, but using interrupt-driven I/O instead of polling. These lights will display the value of a binary counter, which will be incremented at a certain rate by your program. Your code will use interrupts, in two ways: to handle the pushbutton KEY port and to respond to interrupts from a timer. You will use the timer to increment the value of the binary counter displayed on the LEDR lights.

Consider the main program shown in Figure 4. This time, the main program must enable two types of interrupts: the Timer and the KEY pushbuttons. To do this, the main program calls the subroutines *CONFIG_TIMER* and *CONFIG_KEYS*. You are to write each of these subroutines. In *CONFIG_TIMER*, set up the *Timer*, to generate one interrupt every 0.25 seconds. To see how to enable interrupts from this timer, review the lectures and also look at sections 2.5 and 3.3 in the *DE1-SoC Computer System* document given in Lab 1, in the file `DE1-SoC_Computer_NiosII.pdf`.

You are to modify the code in Figure 4 to make it work as follows (if it doesn't already):

1. The main program executes an endless loop, writing the value of the global variable *COUNT* to the red lights LEDR.
2. You have to write an interrupt service routine for the timer, and a new interrupt service routine (different from the one you wrote in Part II) for the KEYs.
3. You will also need to modify your *IRQ_HANDLER* so that it calls the appropriate interrupt service routine, depending on whether an interrupt is caused by the timer or the KEYs port.
4. In the interrupt service routine for the timer you are to increment the variable *COUNT* by the value of the *RUN* global variable, which should be either 1 or 0.

5. You are to toggle the value of the *RUN* global variable in the interrupt service routine for the KEYs, each time a KEY is pressed. When *RUN* = 0, the main program will display a static count on the red lights, and when *RUN* = 1, the count shown on the red lights will increment every 0.25 seconds.

Make a new folder (*part3*) and file (*part3.s*) for this part; you may want to begin by copying the file named `part3_skeleton.s` that is provided with the design files for this exercise. Write your code for this part, and then assemble, run, test and debug it. As noted above, although the Monitor Program supports multiple files in a project, if you are going to develop/debug with CPUlator, then you'll need to put all of your source code in *part3.s*.

```
/*********************************************************************************
 * Write an interrupt service routine
 ********************************************************************************/
.section .exceptions, "ax"
IRQ_HANDLER:
        # save registers on the stack (et, ra, ea, others as needed)
        subi    sp, sp, 16          # make room on the stack
        stw     et, 0(sp)
        stw     ra, 4(sp)
        stw     r20, 8(sp)

        rdctl   et, ctl4            # read exception type
        beq     et, r0, SKIP_EA_DEC # not external?
        subi    ea, ea, 4           # decrement ea by 4 for external interrupts

SKIP_EA_DEC:
        stw     ea, 12(sp)
        andi    r20, et, 0x2        # check if interrupt is from pushbuttons
        beq     r20, r0, END_ISR    # if not, ignore this interrupt
        call    KEY_ISR             # if yes, call the pushbutton ISR

END_ISR:
        ldw     et, 0(sp)           # restore registers
        ldw     ra, 4(sp)
        ldw     r20, 8(sp)
        ldw     ea, 12(sp)
        addi    sp, sp, 16          # restore stack pointer
        eret                        # return from exception

/*********************************************************************************
 * set where to go upon reset
 ********************************************************************************/
.section .reset, "ax"
        movia   r8, _start
        jmp    r8

/*********************************************************************************
 * Main program
 ********************************************************************************/
.text
.global  _start
_start:
        /*
        1. Initialize the stack pointer
        2. set up keys to generate interrupts
        3. enable interrupts in NIOS II
        */
IDLE:   br  IDLE
```

Figure 4: Main program for Part III.

Show your code working to your TA for grading in the lab. Submit `part3.s` to Quercus before the end of your lab period.

Part IV

Modify your program from Part III (in a file called *part4.s*) so that you can vary the speed at which the counter displayed on the red lights is incremented. All of your changes for this part should be made in the interrupt service routine for the KEYs. The main program and the rest of your code should not be changed.

Implement the following behavior: When KEY_0 is pressed, the value of the *RUN* variable should be toggled, as in Part III. Hence, pressing KEY_0 stops/runs the incrementing of the *COUNT* variable. When KEY_1 is pressed, the rate at which *COUNT* is incremented should be doubled, and when KEY_2 is pressed the rate should be halved. You should implement this feature by stopping the timer within the KEYs interrupt service routine, modifying the load value used in the timer, and then restarting the timer. Show your code working to your TA for grading in the lab. Submit `part4.s` to Quercus before the end of your lab period.