ECE 243S - Computer Organization
February 2026
Lab 5

Hex Displays and Interrupt-Driven Input/Output

Due date/time: During your scheduled lab period in the week of February 9, 2026. [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 V 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 V processor. Interrupts are part of a broader class of events called *traps*, and we have described the concepts in class. An exception is one kind of trap, and an interrupt is another kind. You can find a description of all of the registers used for interrupts on chapter 8 of the document NIOS_V_Intro 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 and associated files.

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 `a0` give the 4-bit value to be displayed (which turns into one of the 16 hex digits, 0->F), and the number in register `a1` says which of the six HEX digits to display that digit on. The display can be blanked by turning on bit 4 of `a0` (that is the fifth bit, counting from 0) and `a1` to the value of the display that will be blanked from 0->5.

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 should read and understand the provided subroutine in the code `HEXdisp_subroutine.s`. Pay particular attention to how it saves and restores several registers on the stack.

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 described above. Notice that this specification is intentionally open-ended, and you should not ask for a more detailed specification (on Piazza for example), 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, you are likely to use someone else's software, and you should always be skeptical of that software until you've convinced yourself that it actually works. (Please quote this often: “Software is *broken* until proven otherwise.")

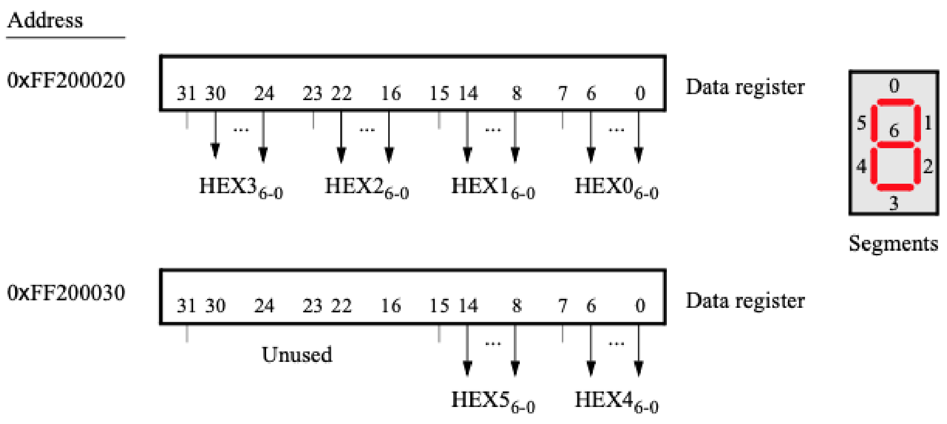


Figure 1: The DE1_SoC Hex Display

Create a new folder to hold your solution for this part and put your code into a file called `part1.s`. Write and debug your program using CPUlator and then make it work as usual on the real hardware in the lab. 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 polling to synchronize with the KEYs port, here you will use interrupt-driven synchronization.

The specific behaviour to implement 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 of KEY_3.

Consider the main program which is provided to you in the file `part2_skeleton.s`. The code has code with the label `interrupt_handler` this should be the address where the NIOS V processor transfers execution after an interrupt is caused. This `interrupt_handler` code checks if the interrupt is from the KEYs port, and if so, it calls the subroutine *KEY_ISR* located at the bottom to handle the interrupt. In the main section of the program (at the top of the skeleton code), the stack pointer is initialized, interrupts are set up properly from both the NIOS V control registers, and the push button's interrupt-mask registers. You are to fill in the code that is not shown.

After setting up interrupts for the KEYs port, the main program of `part2_skeleton.s` simply “idles” in an endless loop. Thus, changing the *HEX*3-0 displays should happen in code that your write 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 put your assembly language code into this file. You may want to begin by copying the file named *part2_skeleton.s* that is provided with the design files for this exercise.

2. Note that the in-lab compilation tool (invoked by GMAKE and described in the Makefile provided in lab 1) 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. The code in `part2_skeleton.s` gives the code required for the interrupt_handler subroutine. You will need to write the code for the *KEY_ISR* interrupt service routine, and the code to set up the NIOS V processor, and the push buttons such that an interrupt is caused when any of the push buttons are pressed. Your code should display 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.

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

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: first, to handle the pushbutton KEY port and second to respond to interrupts from a timer. You will use the timer interrupt to increment the value of the binary counter displayed on the LEDR lights.

Consider the main program included in the file `part3_skeleton.s`. 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 section 3.2 in the file *DE1_SoC_Computer_NiosV* given in Lab 1.
You are to modify the code given in `part3_skeleton.s` to make it work as follows (some of which is already given in the skeleton code):

1. The main program executes an endless loop that continually stores the word at address *COUNT* to the red lights LEDR. It is very important to realize that this loop will be interrupted by the timer and KEYs interrupt request lines, which will cause the value stored at COUNT to change.

2. You must write the code for the 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 *interrupt_handler* subroutine (which starts on line 26 of the skeleton code) 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. (So if RUN=0, the value at COUNT doesn't change, but it does if RUN=1).

5. You are to toggle the value of the *RUN* global variable in the interrupt service routine for the KEYs, each time *any* 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. Your counter should reset to 0 after the count reaches 255. That is, be sure to display 255, wait for 0.25 seconds and then reset.

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 in-lab compiler supports multiple files, if you are going to develop/debug with CPUlator, then you'll need to put all of your source code in *part3.s*.

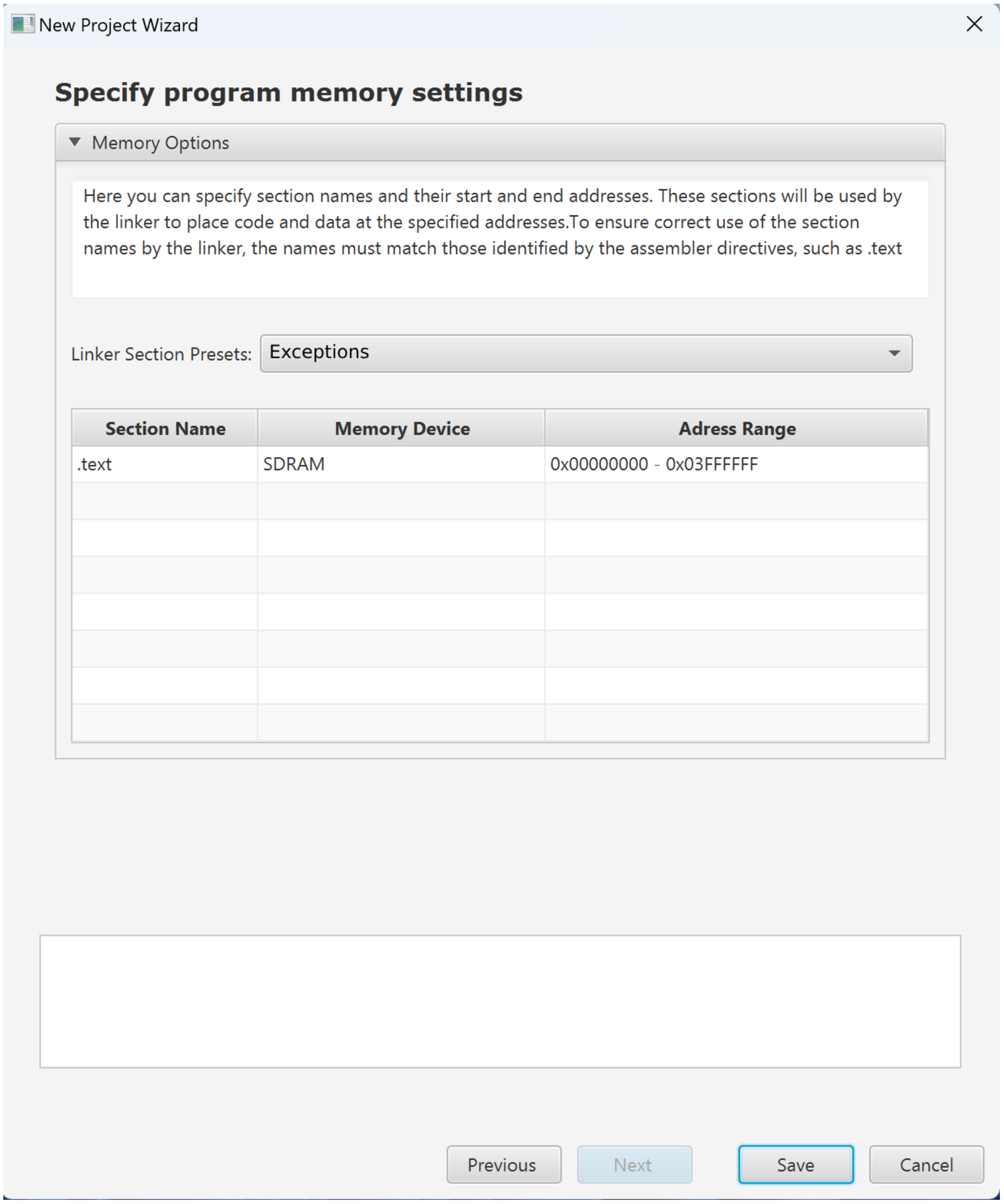
Show your code working to your TA for grading in the lab. Submit `part3.s` to Quercus before the end of your lab period.

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. Additionally, pressing the button to halve the speed might reduce it to zero. If you then try to double the speed to increase the timer, it will not work. Therefore, set both a minimum and maximum speed limit.

Show your code working to your TA for grading in the lab. Submit `part4.s` to Quercus before the end of your lab period.


New Project Wizard

Specify program memory settings

Memory Options

Here you can specify section names and their start and end addresses. These sections will be used by the linker to place code and data at the specified addresses.To ensure correct use of the section names by the linker, the names must match those identified by the assembler directives, such as .text

Linker Section Presets: Exceptions

Section Name	Memory Device	Adress Range
.text	SDRAM	0x00000000 - 0x03FFFFFF

Figure 2: Allowing for exceptions in the Monitor Program