Laboratory Exercise 4

Using Interrupts with Assembly Language Code

The purpose of this exercise is to investigate the use of interrupts for the ARM* processor, using assembly-language code. To do this exercise you need to be familiar with the exceptions processing mechanisms for the ARM processor, and with the operation of the ARM Generic Interrupt Controller (GIC). These concepts are discussed in the tutorials *Introduction to the ARM Processor*, and *Using the ARM Generic Interrupt Controller*. We assume that you are using the DE1-SoC Computer to implement the solutions to this exercise. It may be useful to read the parts of the documentation for this computer system that pertains to the use of exceptions and interrupts, and to carefully review the lectures on this topic. We recognize that there is significant complexity to the set up and running of Interrupts and Exceptions, and so this exercise does provide some of the needed code.

Part I

In this part you will write a program that shows the numbers 0 to 3 on the *HEX*0 to *HEX*3 displays, respectively, when a corresponding pushbutton KEY is pressed. Unlike Lab Exercise 3, where you used polled-input to handle the KEYs port, in this exercise you will use interrupt-driven input.

Consider the main program shown in Figure 1. The code first sets the exceptions vector table for the ARM processor using a code section called *.vectors*. Then, in the *.text* section the main program needs to set up the stack pointers (for both interrupt mode and supervisor mode), initialize the generic interrupt controller (GIC), configure the pushbutton KEY port to generate interrupts, and finally enable interrupts in the processor. You are to fill in the code that is not shown in the figure.

After setting up interrupts for the KEYs port, the main program in Figure 1 simply "idles" in an endless loop. Thus, to control the *HEX*3-0 displays you have to use an interrupt service routine for the pushbutton KEY port.

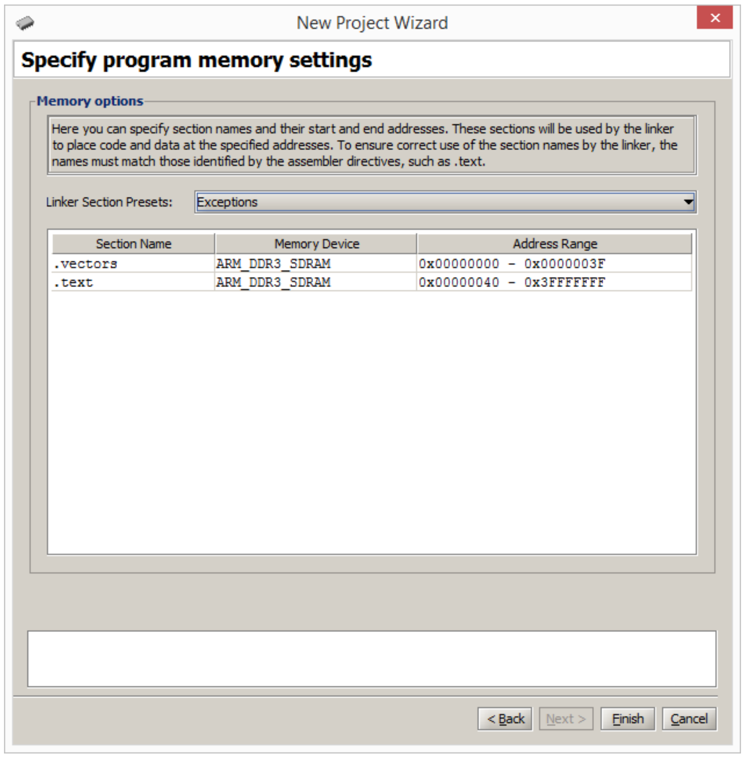
Perform the following:

1. Create a new folder to hold your solution for this part. Make a file called *part1.s*, and type your assembly language code into this file. You may want to begin by copying the file named *part1.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, exception vector table, 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 *part1.s*. You will need to include the code for the subroutine *CONFIG_GIC* that initializes the GIC. This code is included in the file *part1.s* that is provided with the design files for this exercise. It sets up the GIC to send interrupts to the ARM processor from the KEY pushbuttons port.
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 2. 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 bottom part of Figure 1 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.

```
/*********************************************************************************
 * Initialize the exception vector table
 ********************************************************************************/
                .section .vectors, "ax"

                B        _start              // reset vector
                .word    0                   // undefined instruction vector
                .word    0                   // software interrrupt vector
                .word    0                   // aborted prefetch vector
                .word    0                   // aborted data vector
                .word    0                   // unused vector
                B        IRQ_HANDLER         // IRQ interrupt vector
                .word    0                   // FIQ interrupt vector

/*********************************************************************************
 * Main program
 ********************************************************************************/
                .text
                .global  _start
_start:
                /* Set up stack pointers for IRQ and SVC processor modes */
                ... Code not shown

                BL       CONFIG_GIC     // configure generic interrupt controller

                // Configure the KEY pushbutton port to generate interrupts
                ... Code not shown

                // enable IRQ interrupts in the processor
                ... Code not shown
IDLE:
                B        IDLE           // main program simply idles

/********************************************************************************/
IRQ_HANDLER:    PUSH     {R0-R7, LR}

                /* Read the ICCIAR in the CPU interface */
                LDR      R4, =0xFFFEC100
                LDR      R5, [R4, #0x0C]          // read the interrupt ID

CHECK_KEYS:     CMP      R5, #73
UNEXPECTED:     BNE      UNEXPECTED               // if not recognized, stop here

                BL       KEY_ISR
EXIT_IRQ:
                /* Write to the End of Interrupt Register (ICCEOIR) */
                STR      R5, [R4, #0x10]

                POP      {R0-R7, LR}
                SUBS     PC, LR, #4
```

Figure 1: Main program and interrupt service routine.



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	Address Range
.vectors	ARM_DDR3_SDRAM	0x00000000 - 0x0000003F
.text	ARM_DDR3_SDRAM	0x00000040 - 0x3FFFFFFF

Figure 2: Selecting the Exceptions linker section.

Part II

In this part you'll build an interrupt-based program that controls the LEDR lights. 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 3. The code has to set up the ARM stack pointers for interrupt and supervisor modes, and then enable interrupts. A subroutine *CONFIG_GIC* is provided for your use in the design file called *part2.s*. It enables two types of interrupts in the GIC: the A9 Private Timer and the KEY pushbuttons. The main program calls the subroutines *CONFIG_PRIV_TIMER* and *CONFIG_KEYS* to set up the two ports. You are to write each of these subroutines. In *CONFIG_PRIV_TIMER* set up the *A9 Private Timer*, which you learned about in Lab Exercise 5, to generate one interrupt every 0.25 seconds. To see how to enable interrupts from this timer, you may want to read Section 2.4.1 in the *DE1-SoC Computer System* documentation.

In Figure 3 the main program executes an endless loop writing the value of the global variable *COUNT* to the red lights LEDR. 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 I) for the KEYs. Also, you will 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. 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. 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 (*part2*) and file (*part2.s*) for this part; you may want to begin by copying the file named *part2.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 *part2.s*.

```
                  .section .vectors, "ax"
                  ... code not shown

                  .text
                  .global  _start
_start:
/* Set up stack pointers for IRQ and SVC processor modes */
                  ... code not shown

                  BL       CONFIG_GIC          // configure the GIC
                  BL       CONFIG_PRIV_TIMER   // configure A9 Private Timer
                  BL       CONFIG_KEYS         // configure the KEYs port

/* Enable IRQ interrupts in the ARM processor */
                  ... code not shown
                  LDR      R5, =0xFF200000  // LEDR base address
LOOP:
                  LDR      R3, COUNT        // global variable
                  STR      R3, [R5]         // write to the LEDR lights
                  B        LOOP

/* Configure the A9 Private Timer to create interrupts at 0.25 second intervals */
CONFIG_PRIV_TIMER:
                  ... code not shown
                  MOV      PC, LR

/* Configure the pushbutton KEYS to generate interrupts */
CONFIG_KEYS:
                  ... code not shown
                  MOV      PC, LR

/* Global variables */
                  .global  COUNT
COUNT:            .word    0x0              // used by timer
                  .global  RUN              // used by pushbutton KEYs
RUN:              .word    0x1              // initial value to increment
                                            // COUNT
                  .end
```

Figure 3: Main program for Part II.

Part III

Modify your program from Part II (in a file called *part3.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 I. 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 A9 Private Timer within the KEYs interrupt service routine, modifying the load value used in the timer, and then restarting the timer.

Part Part IV: Optional

This part of the lab is entirely optional. You should do this part only if it is of *interest* to you. For this part you are to add a third source of interrupts to your program, using the Interval Timer, which is implemented within the FPGA in the DE1-SoC Computer. To learn how to use this timer, you can refer to Section 2.11 of the *DE1-SoC Computer* documentation. Set up the timer to provide an interrupt every 1/100 of a second. Use this timer to increment a global variable called *TIME*. You should use the *TIME* variable as a real-time clock that is shown on the seven-segment displays $HEX3-0$. Use the format SS:DD, where *SS* are seconds and *DD* are hundredths of a second. You should be able to stop/run the clock by pressing pushbutton KEY_3. When the clock reaches 59:99, it should wrap around to 00:00.

Make a new file, *part4.s*, to hold your solution for this part; you may wish to start by copying the *part4.s* file that is provided as part of the design files for this exercise. Modify the main program from Part III to call a new subroutine, named *CONFIG_TIMER*, which sets up the Interval Timer to generate the required 1/100 second interrupts. To show the *TIME* variable in the real-time clock format SS:DD, you can use the same approach that was followed for Part IV of Lab Exercise 3. In that previous exercise you used polling I/O (with the A9 Private Timer), whereas now you are using interrupts (from the Interval Timer). One possible way to structure your code is illustrated in Figure 4. In this version of the code, the endless loop in the main program writes the value of a variable named *HEX_code* to the $HEX3-0$ displays.

Using the scheme in Figure 4, the interrupt service routine for the Interval Timer has to increment the *TIME* variable, and also update the *HEX_code* variable that is being written to the 7-segment displays by the main program.

Compile, load, test and debug your program.

```
                    .text
                    .global _start
_start:
/* Set up stack pointers for IRQ and SVC processor modes */
                    ... code not shown
                    BL      CONFIG_GIC          // configure the ARM generic
                                                // interrupt controller
                    BL      CONFIG_PRIV_TIMER   // configure the private timer
                    BL      CONFIG_TIMER        // configure the Interval Timer
                    BL      CONFIG_KEYS         // configure the pushbutton
                                                // KEYs port
/* Enable IRQ interrupts in the ARM processor */
                    ... code not shown
                    LDR     R5, =0xFF200000     // LEDR base address
                    LDR     R6, =0xFF200020     // HEX3-0 base address
LOOP:
                    LDR     R4, COUNT           // global variable
                    STR     R4, [R5]            // light up the red lights
                    LDR     R4, HEX_code        // global variable
                    STR     R4, [R6]            // show the time in format
                                                // SS:DD
                    B       LOOP

/* Configure the A9 Private Timer to create interrupts every 0.25 seconds */
CONFIG_PRIV_TIMER:
                    ... code not shown
                    MOV     PC, LR

/* Configure the FPGA Interval Timer to create interrupts every 0.01 seconds */
CONFIG_TIMER:
                    ... code not shown
                    MOV     PC, LR

/* Configure the pushbutton KEYS to generate interrupts */
CONFIG_KEYS:
                    ... code not shown
                    MOV     PC, LR

/* Global variables */
                    .global COUNT
COUNT:              .word   0x0       // used by timer
                    .global RUN       // used by pushbutton KEYs
RUN:                .word   0x1       // initial value to increment COUNT
                    .global TIME
TIME:               .word   0x0       // used for real-time clock
                    .global HEX_code
HEX_code:           .word   0x0       // used for 7-segment displays

                    .end
```

Figure 4: Main program for Part IV.