Laboratory Exercise 5

Using Interrupts with Assembly Code

The purpose of this exercise is to investigate the use of interrupts for the ARM A9 processor, using assembly-language code. To do this exercise you need to be familiar with the exceptions processing mechanisms for the ARM A9 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*. You should also read the parts of the DE0-Nano-SoC Computer documentation that pertain to the use of exceptions and interrupts.

Part I

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 KEYs 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.

The function of your program is to turn on/off the green lights LED_1 and LED_0 when a corresponding pushbutton KEY_1 or KEY_0 is pressed. Since the main program simply "idles" in an endless loop, as shown in Figure 1, you have to control the LEDs by using an interrupt service routine for the pushbutton KEYs.

Perform the following:

1. Create a new folder to hold your Monitor Program project for this part. Create a file, such as *part1.s*, and type the assembly language code for the main program into this file.
2. Create any other source code files you may want, and write the code for the *CONFIG_GIC* subroutine that initializes the GIC. Set up the GIC to send interrupts to the ARM processor from the pushbutton KEYs port.
3. The bottom part of Figure 1 gives the code required for the interrupt handler, *SERVICE_IRQ*. You have to write the code for the *KEY_ISR* interrupt service routine. Your code should turn *on* LED_0 when KEY_0 is pressed, and then if KEY_0 is pressed again you should turn LED_0 *off*. The state of LED_0 should toggle between *on* and *off* in this manner each consecutive time KEY_0 is pressed. Similarly, you should control LED_1 each time KEY_1 is pressed.

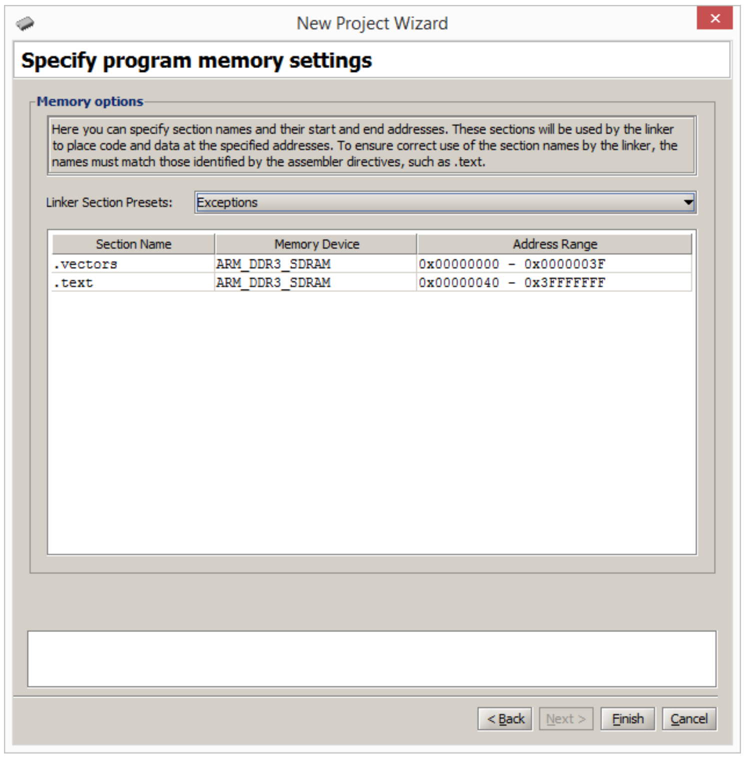
   Figure 2 provides code, using just simple loops, which can be used for the other ARM exception handlers.
4. Make a new Monitor Program project in the folder where you stored your source-code files. In the Monitor Program screen illustrated in Figure 3, make sure to choose Exceptions in the *Linker Section Presets* drop-down menu. Compile, download, and test your program.

```
                .section   .vectors, "ax"
                B          _start               // reset vector
                B          SERVICE_UND          // undefined instruction vector
                B          SERVICE_SVC          // software interrupt vector
                B          SERVICE_ABT_INST     // aborted prefetch vector
                B          SERVICE_ABT_DATA     // aborted data vector
                .word      0                    // unused vector
                B          SERVICE_IRQ          // IRQ interrupt vector
                B          SERVICE_FIQ          // FIQ interrupt vector

                .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

/* Configure the pushbutton KEYs port to generate interrupts
                ··· code not shown

/* Enable IRQ interrupts in the ARM processor */
                ··· code not shown
IDLE:
                B          IDLE                 // main program simply idles

/* Define the exception service routines */

SERVICE_IRQ:    PUSH       {R0-R7, LR}

                LDR        R4, =0xFFFEC100      // GIC CPU interface base address
                LDR        R5, [R4, #0x0C]      // read the ICCIAR in the CPU interface

FPGA_IRQ1_HANDLER:
                CMP        R5, #73              // check the interrupt ID
UNEXPECTED:     BNE        UNEXPECTED           // if not recognized, stop here

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

                POP        {R0-R7, LR}
                SUBS       PC, LR, #4           // return from exception
```

Figure 1: Main program and interrupt service routine.

```
/* Undefined instructions */
SERVICE_UND:
              B          SERVICE_UND
/* Software interrupts */
SERVICE_SVC:
              B          SERVICE_SVC
/* Aborted data reads */
SERVICE_ABT_DATA:
              B          SERVICE_ABT_DATA
/* Aborted instruction fetch */
SERVICE_ABT_INST:
              B          SERVICE_ABT_INST
SERVICE_FIQ:
              B          SERVICE_FIQ

              .end
```

Figure 2: Exception handlers.



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 3: Selecting the Exceptions linker section.

Part II

Consider the main program shown in Figure 4. The code is required to set up the ARM A9 stack pointers for interrupt and supervisor modes, and then enable interrupts. The subroutine *CONFIG_GIC* configures the GIC to send interrupts to the ARM processor from two sources: HPS Timer 0, and the pushbutton KEYs port. The main program calls the subroutines *CONFIG_HPS_TIMER* and *CONFIG_KEYS* to set up the two ports. You are to write each of these subroutines. Set up HPS Timer 0 to generate one interrupt every 0.25 seconds.

In Figure 4 the main program executes an endless loop writing the value of the global variable *COUNT* to the green lights LED. In the interrupt service routine for HPS Timer 0 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 pushbutton KEYs, each time a KEY is pressed. When *RUN* = 0, the main program will display a static count on the green lights, and when *RUN* = 1, the count shown on the green lights will increment every 0.25 seconds.

Make a new Monitor Program project for this part, and assemble, download, and test your code.

Part III

Modify your program from Part II so that you can vary the speed at which the counter displayed on the green lights is incremented. All of your changes for this part should be made in the interrupt service routine for the pushbutton 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 II. Hence, pressing KEY_0 stops/runs the incrementing of the *COUNT* variable. When SW_0 is high and KEY_1 is pressed, the rate at which *COUNT* is incremented should be doubled, and when SW_0 is low and KEY_1 is pressed the rate should be halved. You should implement this feature by stopping the HPS Timer within the pushbutton KEYs interrupt service routine, modifying the load value used in the timer, and then restarting the timer.

Part IV

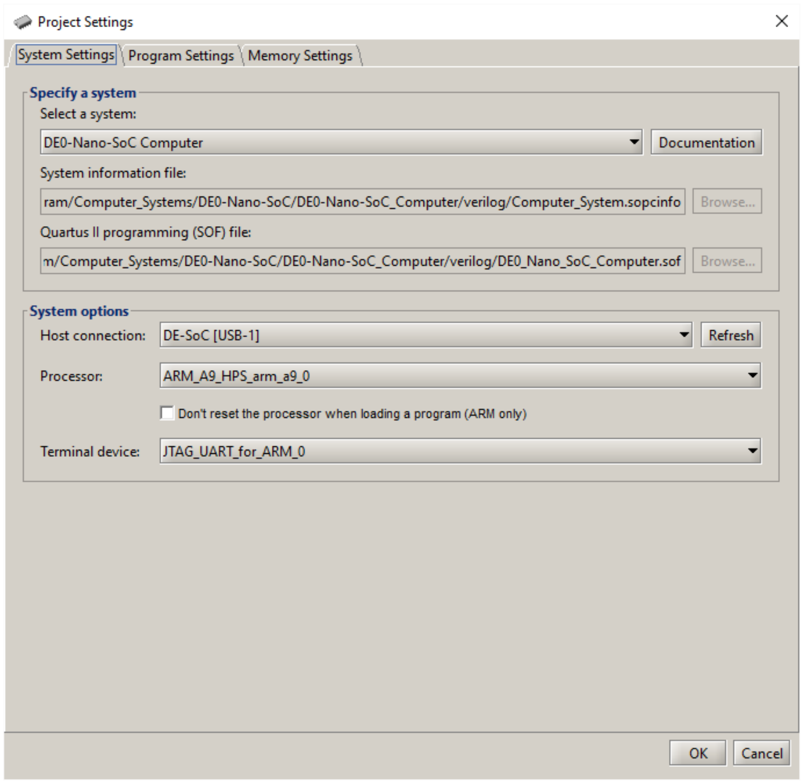
For this part you are to add a third source of interrupts to your program, using the A9 Private Timer. Set up the timer to provide one interrupt each second. Use this timer to increment a global variable called *TIME*. You should use the *TIME* variable as a real-time clock that is shown on the Monitor Program Terminal window. Use the format MM:SS, where *MM* are minutes and *SS* are seconds. You should be able to stop/run the clock by pressing KEY_0. When the clock reaches 59:99, it should wrap around to 00:00.

Make a new folder to hold your Monitor Program project for this part. Modify the main program from Part III to call a new subroutine, named *CONFIG_PRIV_TIMER*, which sets up the A9 Private Timer to generate the required interrupts. To show the *TIME* variable in the real-time clock format MM:SS, you can use the approach that was followed for Part IV of Lab Exercise 4. In Lab Exercise 4 you used polled I/O with the private timer, whereas now you are using interrupts. The interrupt service routine for the private timer should display the real-time clock on the Terminal window.

```
                .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 ARM generic interrupt controller
                BL          CONFIG_HPS_TIMER    // configure HPS Timer 0
                BL          CONFIG_KEYS         // configure the pushbutton KEYs port

// Enable IRQ interrupts in the ARM processor
                · · · code not shown
                LDR         R5, =0xFF200000     // LED base address
LOOP:
                LDR         R3, COUNT           // global variable
                STR         R3, [R5]            // light up the green lights
                B           LOOP

/* Configure the HPS timer to create interrupts at 0.25 second intervals */
CONFIG_HPS_TIMER:
                · · · code not shown
                BX          LR

/* Configure the pushbutton KEYS to generate interrupts */
CONFIG_KEYS:
                · · · code not shown
                BX          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 4: Main program for Part II.

Make a new Monitor Program project and test your program. In the screen shown in Figure 5, make sure to select JTAG_UART_for_ARM_0 as the *Terminal device*. Without this setting no character output will appear on the Terminal window when your code writes to the JTAG UART.



Specify a system
Select a system: DE0-Nano-SoC Computer
System information file: ram/Computer_Systems/DE0-Nano-SoC/DE0-Nano-SoC_Computer/verilog/Computer_System.sopcinfo
Quartus II programming (SOF) file: m/Computer_Systems/DE0-Nano-SoC/DE0-Nano-SoC_Computer/verilog/DE0_Nano_SoC_Computer.sof

System options
Host connection: DE-SoC [USB-1]
Processor: ARM_A9_HPS_arm_a9_0
[ ] Don't reset the processor when loading a program (ARM only)
Terminal device: JTAG_UART_for_ARM_0

Figure 5: Specifying the *Terminal device*.

As a final exercise, add to your progam the ability to slow down/speed up the A9 private timer, in the same way that you implemented this capability for the HPS Timer in Part III of this exercise. Observe the behavior of the Terminal window as it displays the real-time clock value at various timer rates. Discuss any anomolous behavior that you observe.