Laboratory Exercise 7

Using Interrupts with C code

The purpose of this exercise is to investigate the use of interrupts for the Nios II processor, using C code. To do this exercise you need to have a good working knowledge of the exceptions processing mechanisms of the Nios II processor. You should also read the information on exceptions and interrupts of the pre-built computer system documentation corresponding to which DE-series board you own.

This exercise involves the same tasks as those given in Exercise 5, except that this exercise uses C code rather than assembly-language code.

Part I

Consider the main program shown in Figure **??**. The program calls a subroutine *config_KEYs()* to initialize the pushbutton KEYs port so that it will generate interrupts, and calls a subroutine *enable_nios2_interrupts()* to enable interrupts in the Nios II processor. You are to fill in the missing code for the subroutines. To enable interrupts the main program includes *macros*, in the file "nios2_ctrl_reg_macros.h", which provide access to the Nios II status and control registers. Examples of useful macros that might be included are provided in Figure **??**.

After completing the initialization steps described above, the main program just "idles" in an endless loop. The purpose of the program is to show the numbers 0 to 3 on the *HEX*0 to *HEX*3 displays, respectively, when a corresponding pushbutton *KEY* is pressed. Since the main program only idles in a loop, the displays have to be controlled by using an interrupt service routine for the pushbutton KEYs port. If you are using a DE10-Lite, you only need to show the numbers 0 and 1, as the board has only two 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.c*, for your main program, and create any other source-code files that you may wish to use. Write the code for the subroutines that are called by the main program. Be sure to enable Nios II interrupts for the pushbutton KEYs port.
2. The reset and exception handlers for the main program are given in Figure **??**. The function called *the_reset* provides a simple reset mechanism by performing a branch to the main program. The function named *the_exception* represents a general exception handler that can be used with any C program. It includes assembly language code to check if the exception is caused by an external interrupt, and, if so, calls a C language routine named *interrupt_handler*. This routine can then perform whatever action is needed for the specific application. In Figure **??**, the *interrupt_handler* code first determines which exception has occurred, by using a macro from Figure **??** that reads the content of the Nios II interrupt pending register.

You have to write the code for the *pushbutton_isr()* interrupt service routine. Your code should show the digit 0 on the $HEX0$ display when KEY_0 is pressed, and then if KEY_0 is pressed again the display should be "blank". You should toggle the $HEX0$ 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 $HEX1$ through $HEX3$ displays each time KEY_1, KEY_2, or KEY_3 is pressed, respectively. If using a DE10-Lite, you do not need to toggle between 2/3 and "blank" on the $HEX2$/$HEX3$ display.

3. Make a new Monitor Program project in the folder where you stored your source-code files. In the Monitor Program screen illustrated in Figure **??**, make sure to choose Exceptions in the *Linker Section Presets* drop-down menu.

4. Compile, download, and test your program.

```
#include "nios2_ctrl_reg_macros.h"

int main(void)
{
    config_KEYs ();                 // configure pushbutton KEYs to generate interrupts
    enable_nios2_interrupts ();     // enable interrupts in the Nios II processor

    while (1)                       // wait for an interrupt
        ;
}

/* Set up the pushbutton KEYs port in the FPGA */
void config_KEYs(void)
{
    . . . code not shown
}

/* Enable interrupts in the Nios II processor */
void enable_nios2_interrupts(void)
{
    . . . code not shown
}
```

Figure 1: Main program for Part I.

```
#ifndef __NIOS2_CTRL_REG_MACROS__
#define __NIOS2_CTRL_REG_MACROS__
/* Macros for accessing the control registers */

#define NIOS2_READ_STATUS(dest) \
    do { dest = __builtin_rdctl(0); } while (0)
#define NIOS2_WRITE_STATUS(src) \
    do { __builtin_wrctl(0, src); } while (0)
#define NIOS2_READ_ESTATUS(dest) \
    do { dest = __builtin_rdctl(1); } while (0)
#define NIOS2_READ_BSTATUS(dest) \
    do { dest = __builtin_rdctl(2); } while (0)
#define NIOS2_READ_IENABLE(dest) \
    do { dest = __builtin_rdctl(3); } while (0)
#define NIOS2_WRITE_IENABLE(src) \
    do { __builtin_wrctl(3, src); } while (0)
#define NIOS2_READ_IPENDING(dest) \
    do { dest = __builtin_rdctl(4); } while (0)
#define NIOS2_READ_CPUID(dest) \
    do { dest = __builtin_rdctl(5); } while (0)

#endif
```

Figure 2: Macros for accessing Nios II status and control registers.

```
#include "nios2_ctrl_reg_macros.h"

/* function prototypes */
void main(void);
void interrupt_handler(void);
void pushbutton_ISR(void);

/* The assembly language code below handles Nios II reset processing */
void the_reset (void) __attribute__ ((section (".reset")));
void the_reset (void)
/*******************************************************************************
 * Reset code; by using the section attribute with the name ".reset" we allow the linker program
 * to locate this code at the proper reset vector address. This code just calls the main program
 ******************************************************************************/
{
    asm (".set      noat");                 // magic, for the C compiler
    asm (".set      nobreak");              // magic, for the C compiler
    asm ("movia  r2, main");                // call the C language main program
    asm ("jmp       r2");
}
```

Figure 3: Reset and exception handler C code (Part *a*).

```
/* The assembly language code below handles Nios II exception processing. This code should not be
 * modified; instead, the C language code in the function interrupt_handler() can be modified as
 * needed for a given application. */
void the_exception (void) __attribute__ ((section (".exceptions")));
void the_exception (void)
/*****************************************************************************
 * Exceptions code; by giving the code a section attribute with the name ".exceptions" we allow
 * the linker to locate this code at the proper exceptions vector address. This code calls the
 * interrupt handler and later returns from the exception.
 *****************************************************************************/
{
    asm (".set      noat");                       // magic, for the C compiler
    asm (".set      nobreak");                    // magic, for the C compiler
    asm ( "subi     sp, sp, 128");
    asm ( "stw      et, 96(sp)");
    asm ( "rdctl    et, ctl4");
    asm ( "beq      et, r0, SKIP_EA_DEC");        // interrupt is not external
    asm ( "subi     ea, ea, 4");                  /* must decrement ea by one instruction for external
                                                   * interrupts, so that the instruction will be run */
    asm ( "SKIP_EA_DEC:" );
    asm ( "stw      r1, 4(sp)" );                 // save all registers
    asm ( "stw      r2, 8(sp)" );
    ...
    ... (save all regs, except for r27 (sp))
    ...
    asm ( "stw      r31, 124(sp)" );              // r31 = ra
    asm ( "addi     fp, sp, 128" );

    asm ( "call     interrupt_handler" );         // call the C language interrupt handler

    asm ( "ldw      r1, 4(sp)" );                 // restore all registers
    asm ( "ldw      r2, 8(sp)" );
    ...
    ... (restore all saved regs) */
    ...
    asm ( "ldw      r31, 124(sp)" );              // r31 = ra

    asm ( "addi     sp, sp, 128" );
    asm ( "eret" );
}
```

Figure **??**. Reset and exception handler C code (Part *b*).

```
/*****************************************************************************
 * Interrupt Service Routine: Determines the interrupt source and calls the appropriate subroutine
 *****************************************************************************/
void interrupt_handler(void)
{
    int ipending;
    NIOS2_READ_IPENDING(ipending);
    if ( ipending & 0x2 )                    // pushbuttons are interrupt level 1
        pushbutton_ISR( );
    // else, ignore the interrupt
    return;
}
```

Figure **??**. Reset and exception handler C code (Part *c*).

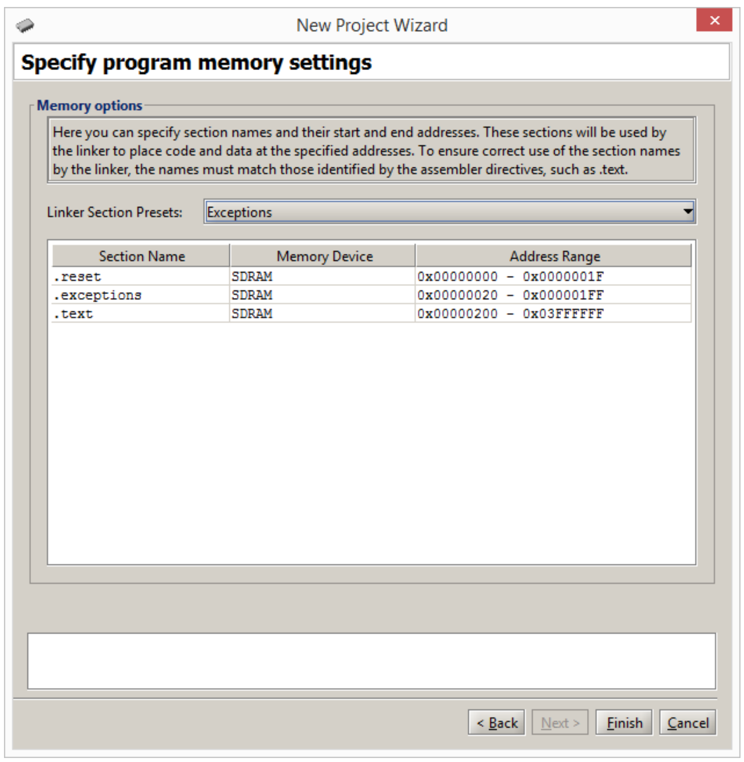

Figure 4: Selecting the Exceptions linker section.

Part II

Consider the main program shown in Figure **??**. The code is required to set up interrupts from two sources: the Interval Timer and the pushbutton KEYs port. The main program calls the subroutines *config_timer( )* and *config_KEYS( )* to set up the two ports. You are to write each of these subroutines. Set up the Interval Timer to generate one interrupt every 0.25 seconds.

In Figure **??** the main program executes an endless loop writing the value of the global variable *count* to the red lights LEDR. In the interrupt service routine for the Interval 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 pushbutton 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 Monitor Program project for this part, and assemble, download, and test your code.

```
int count = 0;                      // global counter for red lights
int run = 1;                        // global, used to increment/not the count variable

int main(void)
{
    volatile int * LEDR_ptr = (int *) 0xFF200000;

    config_timer ();                // configure interval timer
    config_KEYs ();                 // configure pushbutton KEYs to generate interrupts
    enable_nios2_interrupts ();     // enable interrupts in the Nios II processor

    while (1)                       // wait for an interrupt
        *LEDR_ptr = count;
}

/* Set up timer */
void config_timer( )
{
    · · · code not shown
}
/* Set up the pushbutton KEYs port in the FPGA */
void config_KEYs( )
{
    · · · code not shown
}
/* Turn on interrupts in the Nios II processor */
void enable_nios2_interrupts( )
{
    · · · code not shown
}
```

Figure 5: Main program for Part II.

Part III

Modify your program from Part II 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 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 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 either increased or decreased depending on the value of SW_0. If SW_0 is 1, then the rate should be doubled, otherwise the rate should be halved. You should implement this feature by stopping the Interval 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 create a real-time clock that is shown on the seven-segment displays $HEX5-0$. Set up an interval 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 your real time clock. Use the format MM:SS:DD, where *MM* are minutes, *SS* are seconds and *DD* are hundredths of a second. When the clock reaches 59:59:99, it should wrap around to 00:00:00.

Make a new folder to hold your Monitor Program project for this part. Write the program for the real-time clock. To show the *TIME* variable in the real-time clock format *MM*:SS:DD, you can use the same approach that was followed for Part 4 of Lab Exercise 4. In that previous exercise you used polled I/O with the Interval Timer, whereas now you are using interrupts. One possible way to structure your code is illustrated in Figure **??**. The endless loop in this code writes the value of a variable named *HEX_code* to the $HEX3-0$ displays.

Using the scheme in Figure **??**, the interrupt service routine for the second 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.

Make a new Monitor Program project and test your code.

```
int time = 0;                                  // global, used for real-time clock
int HEX_code_lower = 0;                        // global, used for 7-segment displays
int HEX_code_upper = 0;                        // global, used for 7-segment displays

int main(void)
{
    volatile int * HEX_lower_ptr = (int *) // insert HEX3_HEX0 base address here;
    volatile int * HEX_upper_ptr = (int *) // insert HEX7_HEX4 / HEX5_HEX4 base address here;

    config_timer ();                           // configure the Interval Timer

    enable_nios2_interrupts ();                // enable interrupts in the Nios II processor

    while (1)                                  // wait for an interrupt
        *LEDR_ptr = count;
        *HEX_lower_ptr = HEX_code_lower;       // show the time in the format MM:SS:DD
        *HEX_upper_ptr = HEX_code_upper;
}

/* Set up the Interval Timer */
void config_timer( )
{
    · · · code not shown
}
```

Figure 6: Main program for Part IV.