Laboratory Exercise 8

Using Interrupts with Assembly Language and C code

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

Part I

The main program in Figure 1 consists of an endless loop that comprises a few logical instructions. The code in this loop does not serve any useful purpose other than to provide a specific set of instructions that are being executed repeatedly: logical AND, Exclusive-OR, logical OR, and branch. When a pushbutton KEY is pressed to cause an interrupt, the main program may be executing any of the instructions in the loop. After this instruction is completed, the processor will branch to the interrupt handler. Figure 2 shows most of the code for the interrupt handler. It passes to the KEYs interrupt service routine, as a parameter in register r4, the machine code of the next instruction to be executed in the main program. You have to write the KEYs interrupt service routine. It should display on the Terminal window of the Monitor Program which instruction will be executed when the processor returns to the main program. You should use many of the concepts and subroutines from Exercise 7. An example of output on the Terminal window after several KEY presses might be:

and
or
br
xor
xor
and
. . .

```
                .text
                .global  _start
_start:
                /* Set up stack pointer*/
                ... code not shown

                /* Configure the pushbutton KEYs port to generate interrupts
                ... code not shown
                /* Enable IRQ interrupts in the Nios II processor */
                ... code not shown
MAIN_LOOP:
                and     r10, r11, r12
                xor     r13, r14, r15
                or      r16, r17, r18
                and     r18, r17, r16
                xor     r15, r14, r13
                or      r12, r11, r10
                br      MAIN_LOOP

                .end
```

Figure 1: Main program for Part I.

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 *EXCEPTION_HANDLER* and the subroutine to handle interrupts from a pushbutton key. Set the exception handler to send interrupts to the Nios II processor from the pushbutton KEYs port.

```
/************************ RESET SECTION ******************************/
        .section    .reset, "ax"
        movia       r2, _start
        jmp         r2                          /* branch to main program */

/********************** EXCEPTIONS SECTION ***************************/
        .section    .exceptions, "ax"
        .global     EXCEPTION_HANDLER
EXCEPTION_HANDLER:
        subi        sp, sp, 20                  /* make room on the stack */
        stw         et, 0(sp)
        rdctl       et, ctl4
        beq         et, r0, SKIP_EA_DEC         /* interrupt is not external */
        subi        ea, ea, 4                   /* must decrement ea by one instruction */
                                                /* for external interrupts, so that the */
                                                /* interrupted instruction will be re-run */
SKIP_EA_DEC:
        stw         ea, 4(sp)                   /* save all used registers on the Stack */
        stw         ra, 8(sp)                   /* needed if call inst is used */
        stw         r4, 12(sp)
        stw         r22, 16(sp)

        rdctl       et, ctl4
        bne         et, r0, CHECK_LEVEL_1       /* interrupt is an external interrupt */
```

Figure 2: Exception handlers (Part a).

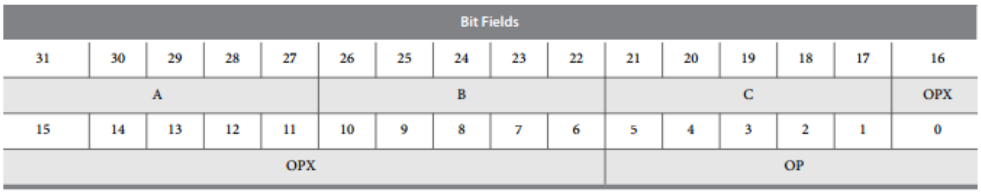
```
NOT_EI:         br      END_ISR                 /* must be unimplemented instruction or TRAP */
                                                /* instruction; ignored in this code */
CHECK_LEVEL_1:                                  /* pushbutton port is interrupt level 1 */
                andi    r22, et, 0b10
                beq     r22, r0, END_ISR        /* other interrupt levels are not handled in this code */

                /*load into r4 the machine code of the instruction that will be executed
                on the return fromt hsi interrupt service routine*/
                call    KEY_ISR

END_ISR:        ldw     et, 0(sp)               /* restore all used register to previous values */
                ldw     ea, 4(sp)
                ldw     ra, 8(sp)               /* needed if call inst is used */
                ldw     r4, 12(sp)
                ldw     r22, 16(sp)
                addi    sp, sp, 20
                eret

                .end
```

Figure 2. The Exception handlers (Part b).

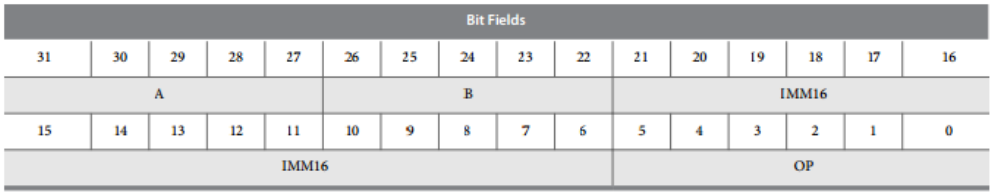
3. To implement the KEYs interrupt service routine, you need to know the Nios II instruction-encodings that are shown in Figure 3. Part a of the figure gives the encoding of R-Type instructions, which include and, xor, and or. Figure 3b gives the encoding of I-Type instructions, which includes the br instruction.

   In part a of Figure 3 the A, B and C fields represent the three registers in the operation. The C field is the destination register, and the A and B fields are the operation registers. For example, an *and* operation would look like: `and rC, rA, rB` . For the R-type instructions `and`, `or`, `xor`, the OP field will always have the value 0x3A. The 11-bit `OPX` field can be used to differentiate between processing instructions: it has values `0x1C0` for **and**, `0x3C0` for **xor** and `0x2C0` for **or**. The `br` instruction is an I-type instruction, which has the format shown in Figure 3b. The IMM16 field specifies a 16-bit immediate value. The OP field holds the operation to be performed. The operation code for the **br** instruction is `0x06`. For the branch instruction, the A and B field can be ignored. For more information on OP codes and Nios II instructions, please see the *Nios II Instruction Set* document.

   Hint: Research the difference between the `ra` *and* `ea` *registers on the Nios II Processor.*



(a) Nios II R-Type Instruction format



(b) Nios II I-Type Instruction format

Figure 3: The machine code format.

To display the name of an instruction mnemonic on the Terminal window you may wish to declare some strings of data as indicated below, and write the characters in these strings to the JTAG UART that is connected to the Terminal window:

```
/* .skip is for padding word alignment*/
AND:        .string    "and\n"
            .skip      3
OR:         .string    "or\n"
XOR:        .string    "xor\n"
            .skip      3
B:          .string    "br\n"
```

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 9, choose Exceptions in the *Linker Section Presets* drop-down menu. In the screen of Figure 5, select JTAG_UART as the *Terminal device*. Refer to Exercise 2, Part IV, for information on using the JTAG UART to communicate with the Monitor Program's Terminal window.

5. Compile, download, and test your program.

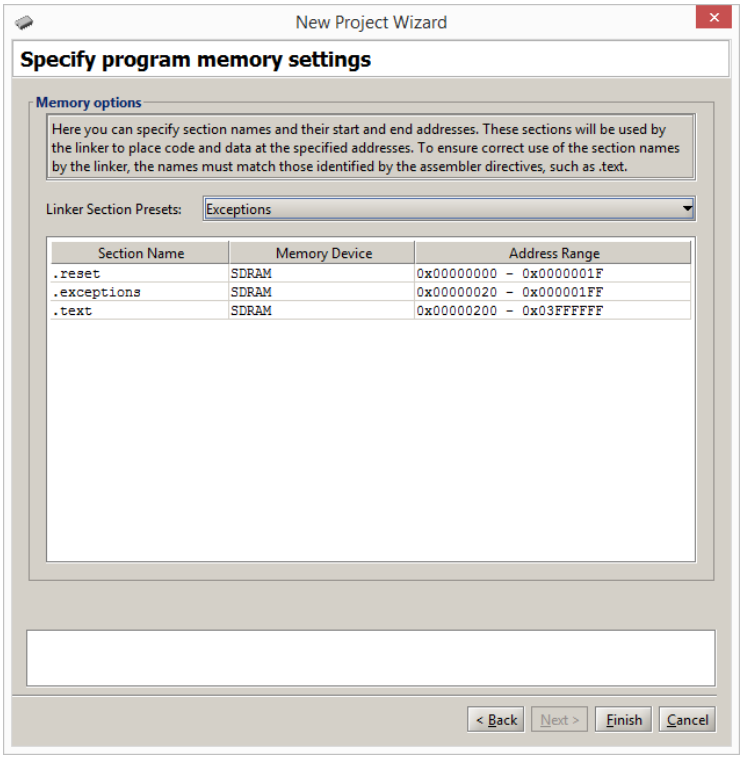


Figure 4: Selecting the Exceptions linker section.

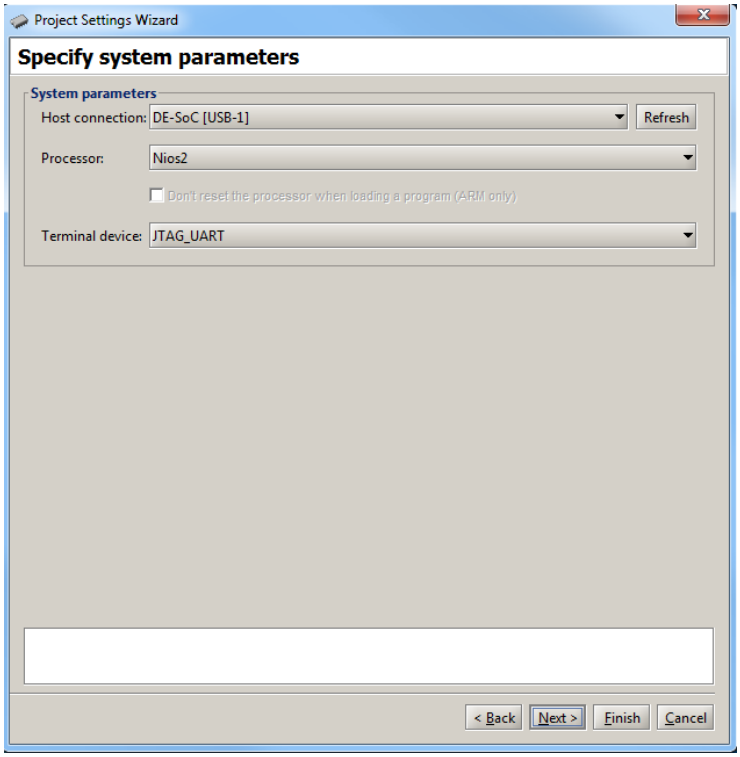


Figure 5: Specifying the *Terminal device*.

Part II

For this part you are to modify your code from Part I so that additional information in displayed on the Terminal window when a pushbutton KEY is pressed. All of your changes for this part should be done in the interrupt service routine for the pushbutton KEYs. You should not have to change any other files. For Part I you were required to display only the mnemonic of the next instruction, such as and or xor, that is to be executed in the main program on return from an interrupt. For this part you are to extend your solution so that it also displays the names of the register arguments of each instruction. Use the machine encodings shown in Figure 3 to find the required information. An example of output produced on the Terminal window after several KEYs have been pressed might be:

```
or      r12, r11, r10
xor     r13, r14, r15
and     r10, r11, r12
or      r16, r17, r18
and     r18, r17, r16
...
```

Part III

Consider the C code shown in Figure 6. It contains an endless loop that repeatedly calls a function named *KEY_pressed* to see if a pushbutton KEY has been pressed. The function *KEY_pressed*, not shown in the figure, checks to see if a KEY is currently being pressed. If so, it waits for the KEY to be released and then returns 1. Otherwise, if no KEY was pressed, 0 is returned. When *KEY_pressed* returns 1, the main program

calls a function named *doit*. This function is supposed to print, on the Terminal window, the mnemonic of the instruction that will be executed on return from the *doit* function. In the main program *inline assembly* commands are used to insert specific instructions following each call to *doit*. For example, the first call to *doit* is followed by the instruction and r10, r11, r12, the second call to *doit* is followed by the instruction xor r13, r14, r15, and so on.

Part of the code for the *doit* function is given in Figure6. It uses two inline assembly commands. The command

```
asm("ldw r10, 0(ra)" : : : "r10")
```

loads the machine code of the instruction pointed to by the return address register into register r10. The argument "r10" in this command informs the C compiler that the contents of register r10 will be changed as a result of executing the instruction. The command

```
asm("mov %0, r10" : "=r" (machine_code) : : )
```

copies the contents of register r10 into the variable *machine_code*. The argument %0 in this command is set to whichever register, for example register r11, is chosen by the C compiler to hold the value of the variable *machine_code*.

Make sure that you understand how the inline assembly commands work. Documentation on these commands can be found by searching on the web for *gnu inline assembly Nios II*, or something similar.

Perform the following:

1. Type the C code for the main program into a file, for example *part3.c*. Also, add your own code for the *KEY_pressed* function.

2. Complete the C code for the *doit* function. You should display on the Terminal window the mnemonic of the instruction corresponding to the *machine_code* variable. Refer to Figure 3 for the machine code formats. Remember from previous exercises that the DE0-Nano and De0-Nano-SoC boards do not have access to enough memory to use library functions like *printf*. You should use the functions you developed in previous exercises (Exercises 6 and 7) to print characters, strings and numbers to the JTAG terminal window.

   Hint: In this example we are not using interrupts, how does this change the use of the ra *and* ea *registers in terms of accessing the next instruction to be executed in the main function?*

3. Make a new Monitor Program project for this part of the exercise. In the Monitor Program screen shown in Figure 7 set the *Terminal device* to JTAG_UART.

4. Compile, download, and test your program.

```
/* Function prototypes */
int KEY_pressed(void);
void doit(void);

/* This program demonstrates the use of inline assembly code in C code */
int main(void)
{
    while (1)                                                    // endless loop
    {
        if (KEY_pressed ()) doit ();
        asm("and r10, r11, r12");
        if (KEY_pressed ()) doit ();
        asm("xor r13, r14, r15");
        if (KEY_pressed ()) doit ();
        asm("or r16, r17, r18");
        if (KEY_pressed ()) doit ();
        asm("and r18, r17, r16");
        if (KEY_pressed ()) doit ();
        asm("xor r15, r14, r13");
        if (KEY_pressed ()) doit ();
        asm("or r12, r11, r10");
        if (KEY_pressed ()) doit ();
    }
}

int KEY_pressed( )
{
    ··· code not shown
}

void doit( )
{
    unsigned int machine_code;
    // get the machine code of the next instruction on return from this subroutine
    asm("ldw r10, 0(ra)" : : : "r10");                           // read machine code into r10
    asm("mov %0, r10" : "=r" (machine_code) : : );               // copy r10 into variable machine_code

    ··· code not shown
}
```

Figure 6: Main program for Part III.

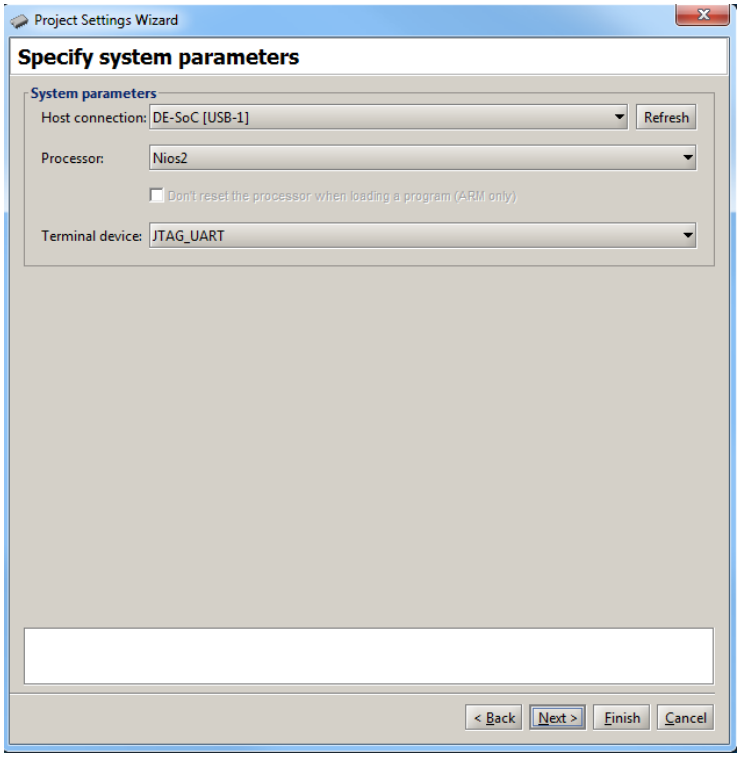


Figure 7: Specifying the *Terminal device*.

Part IV

In this part you are to repeat the tasks given in Parts I and II, but using C code rather than assembly-language code.

Consider the main program shown in Figure 8. The main program calls the subroutine *config_KEYs()* to initialize the pushbutton KEYs port so that it will generate interrupts. Finally, a subroutine *enable_nios2_interrupts()* is called to unmask IRQ interrupts in the Nios II processor.

After completing the initialization steps described above, the main program executes an endless loop that consists of several logical instructions. Inline assembly commands, described in Part III, are used to specify the logic instructions.

Perform the following:

```
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
    {
        asm("and r10, r11, r12");
        asm("xor r13, r14, r15");
        asm("or r16, r17, r18");
        asm("and r18, r17, r16");
        asm("xor r15, r14, r13");
        asm("or r12, r11, r10");
    }
}

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

Figure 8: Main program for Part IV.

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.

2. Use the exception handler code you created for Exercise 7 which called the *interrupt_handler*, and in turn called the *pushbutton_ISR()* function. You should not need to change the exception handler or interrupt handler code you wrote, you should only need to change the functionality of the *pushbutton_ISR()* function.
   Your code should perform the same task as in Part II of this exercise. That is, you are to print on the Terminal window the mnemonic, and register indices, of the instruction that will be executed on return from the interrupt. You can make the same assumptions as in Parts I and II, namely that only the logical instructions, plus the branch instruction, have to be handled by your code

3. Make a new Monitor Program project in the folder where you stored your source-code files. As discussed for Part III, in the Monitor Program screen shown in Figure 7 set the *Terminal device* to JTAG_UART. Also, in the Monitor Program screen illustrated in Figure 9, make sure to choose Exceptions in the *Linker Section Presets* drop-down menu.

4. Compile, download, and test your program.

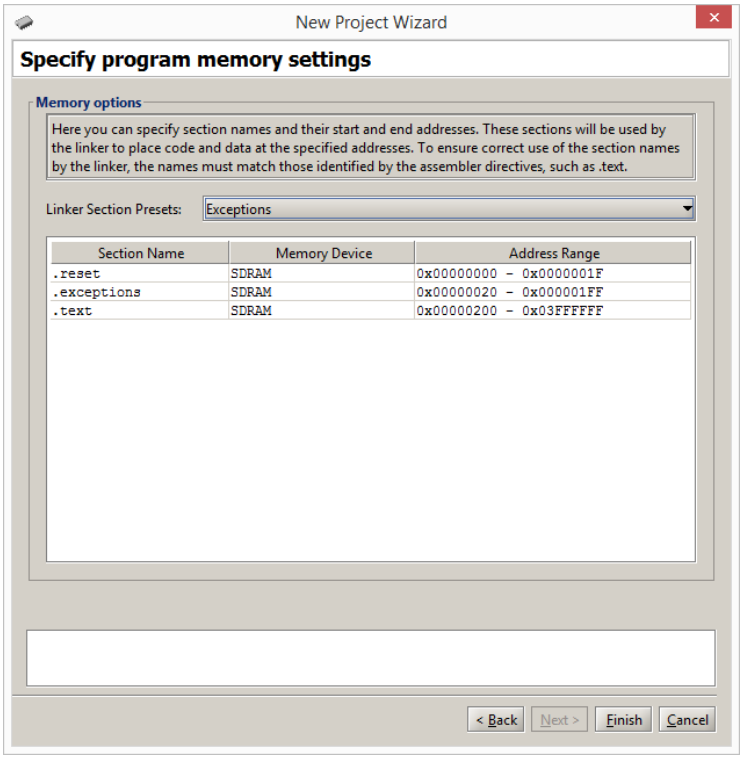


Figure 9: Selecting the Exceptions linker section.