Laboratory Exercise 4

Interrupts

Laboratory Exercise 3 illustrated how input/output transfers can be performed using the program-controlled polling approach. Now, we will consider the interrupt-driven approach to perform the same tasks. We will use both the Nios II assembly language and the C programming language to implement the necessary software. As an example of I/O hardware, we will again make use of parallel-port interfaces in the DE-series Media Computer system implemented on an Altera DE-series board.

The application task in this exercise consists of adding together a set of unsigned 8-bit numbers that are entered via the slider switches on the DE-series board. The resulting sum is displayed on the LEDs and 7-segment displays. Use 8 slider switches, SW_{7-0}, as inputs for entering numbers. Use the green lights, $LEDG_{7-0}$, to display the number defined by the slider switches. Use the 16 red lights, $LEDR_{15-0}$, to display the accumulated sum as a binary number. Display this sum also on the 7-segment displays *HEX3-HEX0*. All of these components are connected via parallel ports in the DE-series Media Computer. A new number is to be added to the current sum whenever an interrupt request is raised by pressing the pushbutton switch KEY_1.

Parallel Ports

The parallel port interfaces in the DE-series Media Computer were generated by using Altera's Qsys Tool software (which we will use in Laboratory Exercise 5). A parallel port provides for data transfer in either input or output direction. In the Qsys tool, a parallel port is implemented in the form of a *PIO (Parallel Input/Output)* component. The transfer of data is done in parallel and it may involve from 1 to 32 bits. The number of bits, n, and the type of transfer are specified by the user through the Qsys tool (at the time a Nios II based system is being designed). The PIO interface can contain the four registers shown in Figure 1.

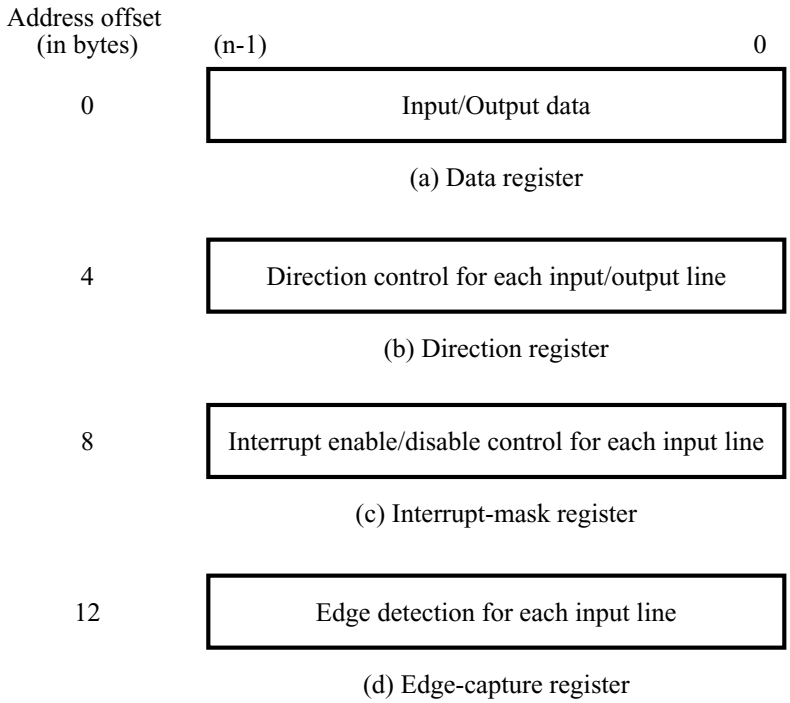

Figure 1. Registers in the PIO interface.

Each register is n bits long. The registers have the following purpose:

- *Data* register holds the n bits of data that are transferred between the PIO interface and the Nios II processor. It can be implemented as an input, output, or a bidirectional register by the Qsys tool.
- *Direction* register defines the direction of transfer for each of the n data bits when a bidirectional interface is generated.
- *Interrupt-mask* register is used to enable interrupts from the input lines connected to the PIO.
- *Edge-capture* register indicates when a change of logic value is detected in the signals on the input lines connected to the PIO.

Not all of these registers are generated in a given PIO interface. For example, the *Direction* register is included only when a bidirectional interface is specified. The *Interrupt-mask* and *Edge-capture* registers must be included if interrupt-driven input/output is used.

The PIO registers are accessible as if they were memory locations. Any base address that has the four least-significant bits equal to 0 can be assigned to a PIO (at the time it is implemented by the Qsys tool). This becomes the address of the *Data* register. The addresses of the other three registers have offsets of 4, 8, or 12 bytes (1, 2, or 3 words) from this base address.

The DE-series Media Computer includes several PIOs that are configured for various uses. The details of these PIOs are described in the *Media Computer System for Altera DE-series Board* tutorial.

Interrupts in the DE-series Media Computer

In the DE-series Media Computer, memory address 0x20 is the interrupt location. This means that when an external interrupt request is received, the Nios II processor will automatically execute the instruction stored at memory address 0x20. The processor performs this action also if an internal exception occurs, such as dividing by zero or encountering a software **trap** instruction.

One of the I/O peripherals that can raise interrupts is the *Pushbutton-switch Parallel Port*. It raises an interrupt request when a pushbutton key is pressed, if the corresponding bit in its *Interrupt-mask* register is set to 1. It also records the occurrence of a change in the signal generated by the pushbutton by setting to 1 the corresponding bit in the *Edge-capture* register.

Note that it is necessary to enable interrupts in three diferrent places: the I/O device interface (*Interrupt-mask* register in a PIO), the control register *ctl3* (*ienable*), and the control register *ctl0* (*status*).

Each I/O peripheral is assigned an IRQ (Interrupt Request) number, which is used to identify the source of an interrupt request. For the Pushbutton Port, the IRQ number is 1, which means that bit b_1 in the Nios II control register 4 will be set to 1 whenever some pushbutton key is pressed and the corresponding interrupts are enabled. The control register 4 can be accessed in an assembly language program as either *ctl4* or *ipending*.

A detailed explanation of Nios II interrupts is given in the tutorials: *Introduction to the Altera Nios II Soft Processor* and *Media Computer System for Altera DE-series Board*.

Reset in the DE-series Media Computer

Pushbutton switch KEY_0 provides the reset capability in the DE-series Media Computer. When this key is pressed, the Nios II processor is forced to execute the instruction stored at memory address 0.

Part I

In this part, you have to write a program that uses interrupts to read the contents of the slider switches, display the corresponding value on the green LEDs, and add this number to a sum that is being accumulated. Display the sum as a binary number on the red LEDs and as a hexadecimal number on the 7-segment displays *HEX3-HEX0*. A new number must be added each time the key KEY_1 is pressed, which has to cause an interrupt request.

Implement this task using the Nios II assembly language, as follows:

1. Write a Nios II assembly-language program that performs the desired task.
2. Create a new directory, *lab4_part1*. Put your program, *lab4_part1.s*, into this directory.
3. Use the *Altera Monitor Program* to create a new project, *part1*, in this directory. Select your program and download the DE-series Media Computer into the FPGA device on the DE-series board. Choose SDRAM as the memory that your program will use. Assemble and download your program.
4. Run your program to demonstrate that it works properly.

Part II

Augment your program to display the sum as a decimal number on the 7-segment displays *HEX3-HEX0*, and verify its correctness.

Interrupts in a C-language Program

When using interrupts in a C-language program, it is necessary to deal with the specific requirements of the Nios II processor and DE-series Media Computer interrupt mechanism, which necessitates using some assembly-language instructions in the C code. Following is a brief explanation of what needs to be done when the Altera Monitor Program is used to run a C-language program.

Accessing the Nios II Control Registers

Nios II processor control registers can be accessed by using the **asm** type statements. For example, the contents of control register 4 can be copied into the general-purpose register **r8** by writing either

```
asm ("rdctl  r8, ctl4");
```

or

```
asm ("rdctl  r8, ipending");
```

To make it easier to incorporate such instructions into a program, there exists a set of macros in a file *nios2_ctrl_reg_macros.h* which is one of the design files provided with this laboratory exercise. Then, the above action can be specified with the statement

```
NIOS2_READ_IPENDING(8);
```

Using Attributes to Specify the Required Locations for Compiled C-Code

Pressing the reset key, KEY_0, forces the instruction at address 0 to be executed. This has to cause a branch to the beggining of the actual program, which can be achieved with the code:

```
void  the_reset(void)__attribute__((section(".reset")));
void  the_reset(void)
{
   asm  ("movia  r2, _start");
   asm  ("jmp  r2");
}
```

An external interrupt request, or an internal exception, forces the instruction at address 0x20 to be executed. The instructions that deal with interrupts and internal exceptions must start at this address, which can be accomplished by preceding the relevant C-code with the statements:

```
void  the_exception(void)__attribute__((section(".exceptions")));
void  the_exception(void)
```

The compiler associates the section attributes *.reset* and *.exceptions* with memory addresses 0x0 and 0x20, respectively, by examining the design parameters of the DE-series Media Computer.

Specifying the Address of the Main Program

The default address into which the main program is loaded is zero. But, the reset and interrupt-handler code has to be loaded into the memory starting at addresses 0x0 and 0x20, respectively. Thus, it is necessary to place the main-program code into higher-address locations to ensure that this code will not overlap with the interrupt-handler code. As shown in Figure 2, we have used the Altera Monitor Program to specify that the main-program code should start at address 0x400.

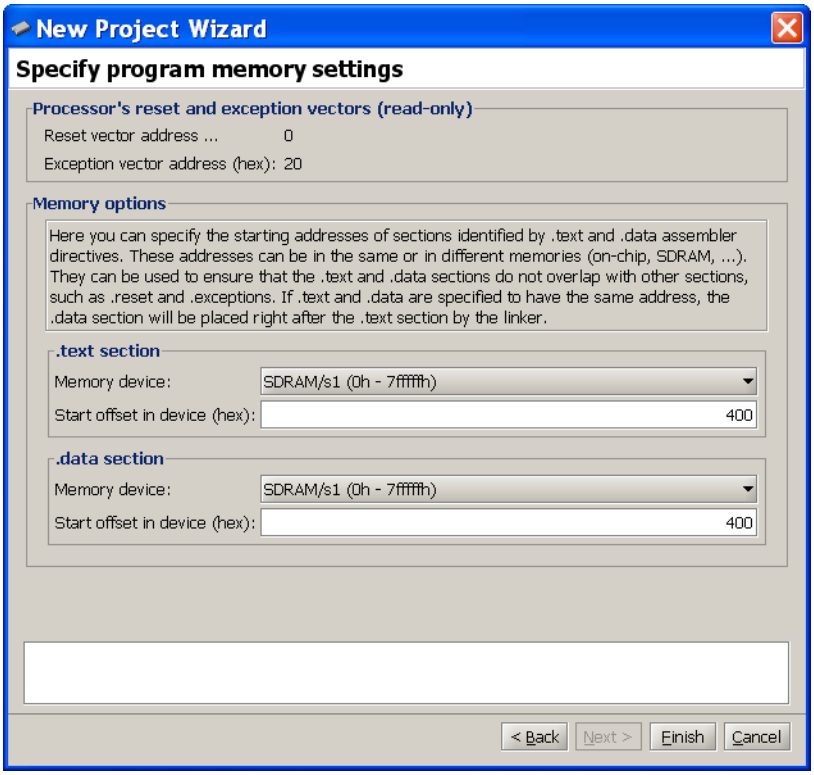

Figure 2. Specifying the memory location of the main-program.

Part III

Repeat Part I by writing the necessary program in the C language. Figure 3 gives a skeleton program that may be used for this purpose. You have to fill in the necessary details.

```
#include "nios2_ctrl_reg_macros.h"

    /* Define here the addresses of switches, lights and 7-segment displays. */

/* Code needed to implement the reset funcionality */
void  the_reset(void)__attribute__((section(".reset")));
void  the_reset(void)
{
    asm  ("movia  r2, _start");
    asm  ("jmp  r2");
}

/* Code needed to deal with exceptions */
void  the_exception(void)__attribute__((section(".exceptions")));
void  the_exception(void)
{
    asm  ("subi  sp, sp, 4");                  /* Save the contents of  */
    asm  ("stw  et, (sp)");                    /*   the et register.    */
    asm  ("rdctl  et, ipending");              /* If external interrupt, */
    asm  ("beq  et, r0, SKIP_EA_DEC");         /*   then decrement et   */
    asm  ("subi  ea, ea, 4");                  /*   by 4.          */
    asm  ("SKIP_EA_DEC:");

    /* Insert here the code needed to save all registers except r0, et and sp. */

    asm  ("call  INTERRUPT_HANDLER");

    /* Insert here the code to restore all registers except r0 and sp. */

    asm  ("eret");                             /* Return from exception. */
}

void  INTERRUPT_HANDLER()
{
    /* Insert here the INTERRUPT_HANDLER code, which has to check if an interrupt */
    /*   from KEY1 has occurred and in response update the displayed sum.    */
}

main()
{
    /* Insert here the code for the main program, which has to enable   */
    /*   the desired interrupts. It also has to display the current sum.     */
}
```

Figure 3. Skeleton program for Part III.

Part IV

Repeat Part II by writing the necessary program in the C language.

Preparation

Your preparation should include the programs for Parts I to IV.