Laboratory Exercise 4

Input/Output in an Embedded System

The purpose of this exercise is to investigate the use of devices that provide input and output capabilities for a processor. There are two basic techniques for dealing with I/O devices: program-controlled polling and interrupt-driven approaches. You will use the polling approach in this exercise, writing programs in the Nios II assembly language. Your programs will be executed on a Nios II processor in one of the DE-series computer systems. Parallel port interfaces, as well as a timer module, will be used as examples of I/O hardware.

A parallel port provides for data transfer in either the input or output direction. 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 depend on the specifications of the specific parallel port being used. The parallel port interface can contain the four registers shown in Figure 1.

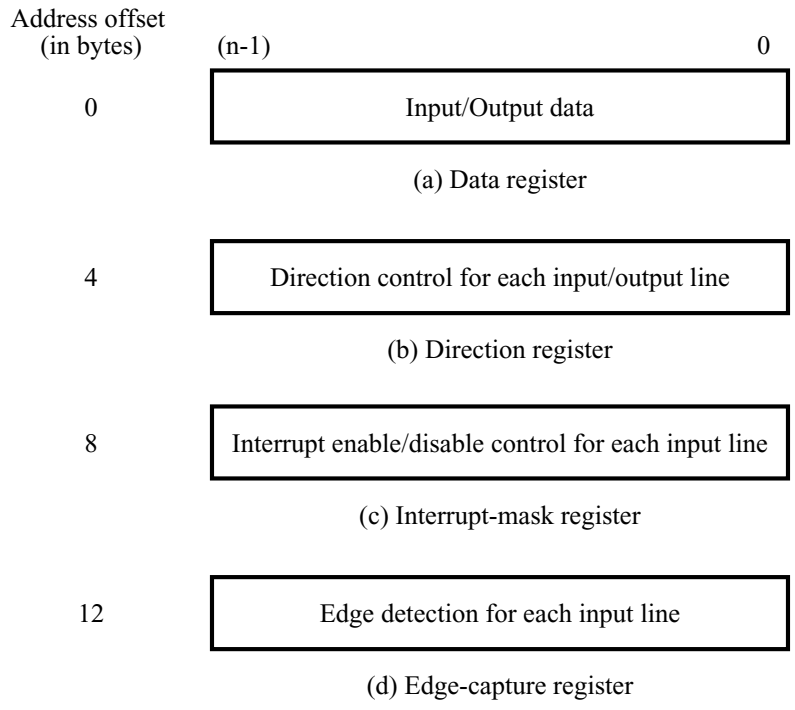

Figure 1: Registers in the parallel port interface.

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 parallel port and the Nios II processor. It can be implemented as an input, output, or a bidirectional register.

- *Direction* register defines the direction of transfer for each of the n data bits when a bidirectional interface is generated.
- *Interrupt-mask* register: used to enable interrupts from the input lines connected to the parallel port.
- *Edge-capture* register: indicates when a change of logic value is detected in the signals on the input lines connected to the parallel port. Once a bit in the edge capture register becomes asserted, it will remain asserted. An edge-capture bit can be de-asserted by writing to it using the Nios II processor.

Not all of these registers are present in some parallel ports. 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 parallel port registers are memory mapped, starting at a specific *base* address. The base address has to be a multiple of four if the parallel port is to be accessed using word accesses from the Nios II processor. The base address becomes the address of the *Data* register in the parallel port. 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 pre-built computer system all have parallel ports connected to slider switches, pushbutton KEYs, LEDs, and seven-segment displays (when those peripherals exist).

Part I

Write a Nios II assembly-language program that displays a decimal digit on the seven-segment display *HEX*0. The other seven-segment displays on your DE-series board should be blank.

The parallel port connected to the seven-segment displays $HEX3-0$ is memory mapped at the address 0xFF200020, and the port connected to $HEX5-4$ is at the address 0xFF200030. Figure 2 shows how the display segments are connected to the parallel ports.

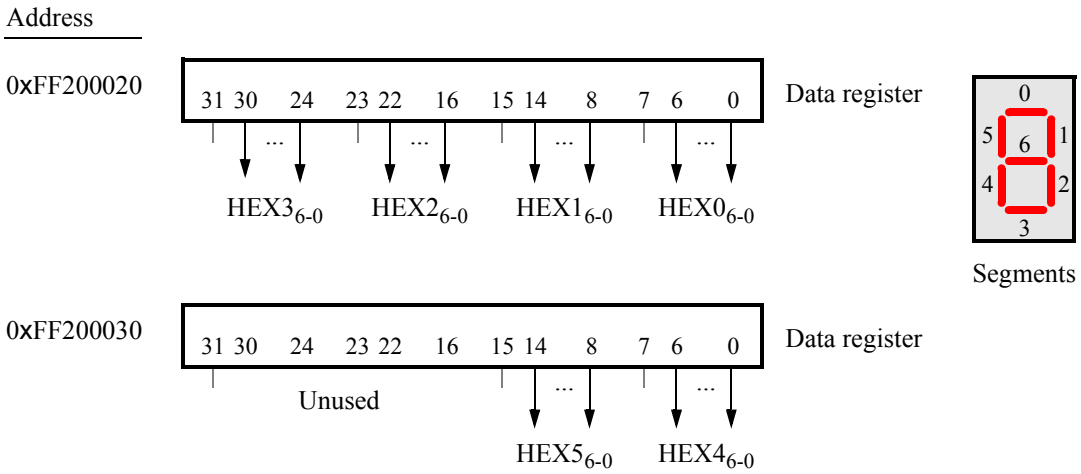

Figure 2: The parallel ports connected to the seven-segment displays, $HEX5-0$, in the DE-series copmuter systems.

Initially the number displayed on *HEX*0 should be 0. If KEY_1 is pressed then SW_0 should be checked. If SW_0 is high increment the displayed number, and if it is low then decrement the number. Pressing KEY_0

should blank the display, and pressing any other KEY after that should return the display to 0. The parallel port connected to the pushbutton *KEYs* is illustrated in Figure 3. The parallel port connected to the slider switches *SWs* is illustrated in Figure 4. In your program, use polled I/O to read the *Data* register to see when a button is being pressed. When you are not pressing any *KEY* the *Data* register provides 0, and when you press KEY_i the *Data* register provides the value 1 in bit position i. Once a button-press is detected, be sure that your program waits until the button is released. You should not use the *Interruptmask* or *Edgecapture* registers for this part of the exercise.

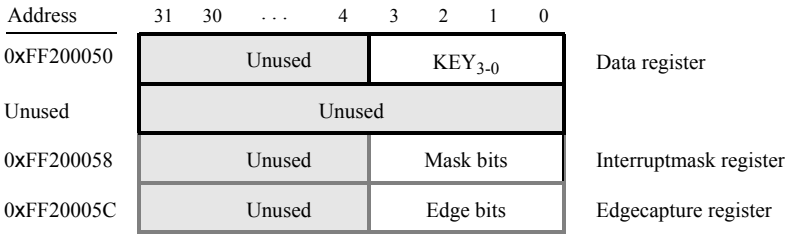

Figure 3: The parallel port connected to the pushbutton *KEYs*.

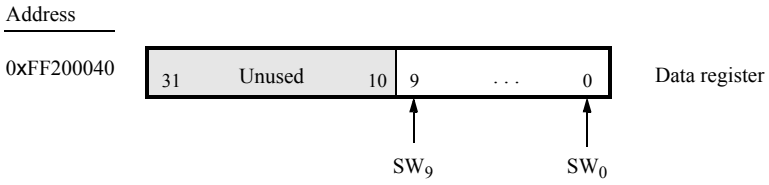

Figure 4: The parallel port connected to the slider switches *SWs*.

Perform the following:

1. Create a new folder to hold your Monitor Program project for this part. Create a file called *part1.s* and type your assembly language code into this file. You may want to refer to a discussion, and examples of assembly-language code, in Part IV of Lab Exercise 2 about displaying numbers on seven-segment displays.
2. Make a new Monitor Program project in the folder where you stored the *part1.s* file. Select Nios II as the target processor architecture and use the appropriate pre-built computer system for your DE-series board.
3. Compile, download, and test your program.

Part II

Write a Nios II assembly-language program that displays a two-digit decimal counter on the seven-segment displays $HEX1 - 0$. The counter should be incremented approximately every 0.25 seconds. When the counter reaches the value 99, it should start again at 0. The counter should stop/start when any pushbutton *KEY* is pressed.

To achieve a delay of approximately 0.25 seconds, use a delay-loop in your assembly language code. A suitable example of such a loop is shown below.

```
DO_DELAY:   movia   r7, 8000000          # delay counter
SUB_LOOP:   subi    r7, r7, 1
            bne     r7, zero, SUB_LOOP
```

To avoid "missing" any button presses while the processor is executing the delay loop, you should use the *Edgecapture* register in the *KEY* port, shown in Figure 3. When a pushbutton is pressed, the corresponding bit in the *Edgecapture* register is set to 1, and it remains set until reset to 0 by writing into the register.
Perform the following:

1. Create a new folder to hold your Monitor Program project for this part. Create a file called *part2.s* and type your assembly language code into this file.
2. Make a new Monitor Program project in the folder where you stored the *part2.s* file. Select Nios II as the target processor architecture and use the appropriate pre-built computer system for your DE-series board.
3. Compile, download, and test your program.

Part III

In Part II you used a delay loop to cause the Nios II processor to wait for approximately 0.25 seconds. The processor loaded a large value into a register before the loop, and then decremented that value until it reached 0. In this part you are to modify your code so that a hardware timer is used to measure an exact delay of 0.25 seconds. You should use polled I/O to cause the Nios II processor to wait for the timer.

The pre-built computer systems include an *Interval Timer* implemented in the FPGA that can be used by

the Nios II processor. This timer can be loaded with a preset value, and then counts down to zero using the 100-MHz clock signal provided as the system clock in the DE-series computer systems (or 50-Mhz for the DE2-115 Computer System). The programming interface for the timer includes six 16-bit registers, as illustrated in Figure 5.

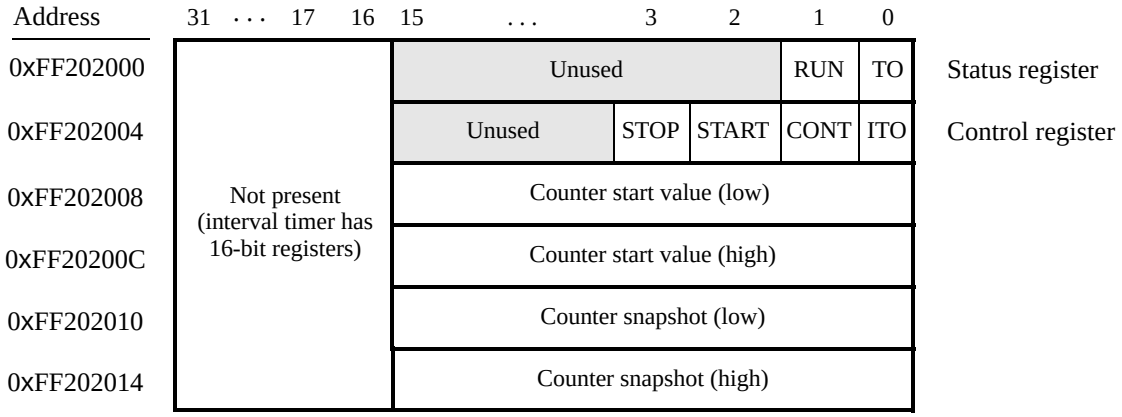

Figure 5: The Interval Timer registers.

The *TO* bit in the *Status* register provides a timeout signal which is set to 1 by the timer when it has reached a count value of zero. You should poll this bit in your program to cause the Nios II processor to wait for the timer. The *TO* bit can be reset by writing a 0 into it.

The *CONT* bit affects the continuous operation of the timer. When the timer reaches a count value of zero it automatically reloads the specified starting count value. If *CONT* is set to 1, then the timer will continue counting down automatically. But if $CONT = 0$, then the timer will stop after it has reached a count value of 0. The (*START*/*STOP*) bits can be used to commence/suspend the operation of the timer by writing a 1 into the respective bit.

The two 16-bit registers for the Counter start value allow the period of the timer to be changed. The default setting provided gives a timer period of 125 msec. To achieve this period, the starting value of the count is $100 \text{ MHz} \times 125 \text{ msec} = 12.5 \times 10^6$.

Make a new folder to hold your Monitor Program project for this part. Create a file called *part3.s* and type your assembly language code into this file. Make a new Monitor Program project for this part of the exercise, and then compile, download, and test your program.

Part IV

In this part you are to write an assembly language program that implements a real-time clock. Display the time on the seven-segment displays $HEX3 - 0$ in the format SS:DD, where *SS* are seconds and *DD* are hundredths of a second. Measure time intervals of 0.01 seconds in your program by using polled I/O with the Interval Timer. You should be able to stop/run the clock by pressing any pushbutton *KEY*. 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. Create a file called *part4.s* and type

your code into this file. Make a new Monitor Program project for this part of the exercise, and then compile, download, and test your program.