Laboratory Exercise 5

Using Input/Output Devices

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 ARM* assembly language. Your programs will be executed on the ARM processor in the DE1-SoC Computer system. 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.

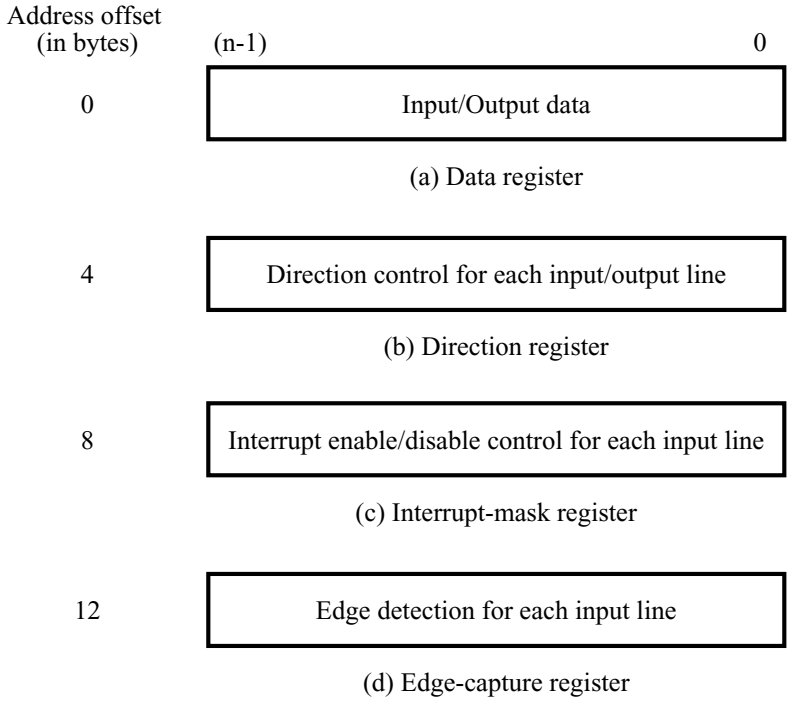

Figure 1: Registers in the parallel port interface.

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 ARM 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 ARM 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 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. In the DE1-SoC Computer parallel ports are used to connect to SW slide switches, KEY pushbuttons, LEDs, and seven-segment displays.

Part I

Write an ARM assembly language program that displays a decimal digit on the seven-segment display *HEX*0. The other seven-segment displays *HEX*$5-1$ should be blank.

The DE1-SoC Computer contains a parallel port connected to the seven-segment displays *HEX*$3-0$. The port is memory mapped at the base address 0xFF200020. A second parallel port is connected to *HEX*$5-4$, at the base address 0xFF200030. Figure 2 shows how the display segments are connected to the parallel ports.

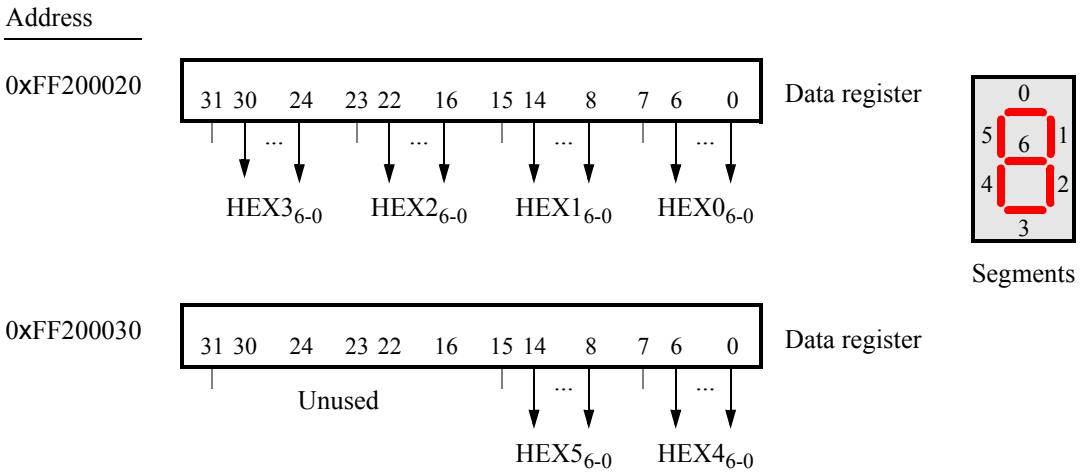

Figure 2: The parallel ports connected to the seven-segment displays *HEX*$5-0$.

Perform the following:

1. If KEY_0 is pressed on the board, you should set the number displayed on *HEX*0 to 0. If KEY_1 is pressed then increment the displayed number, and if KEY_2 is pressed then decrement the number. Pressing KEY_3 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* has the base address 0xFF200050, as illustrated in Figure 3. 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.
2. Create a new folder to hold your solution for this part. Create a file called *part1.s* and type your assembly language code into this file. You may want to refer to discussions, and examples of assembly-language code, about displaying numbers on seven-segment displays in previous lab exercises.

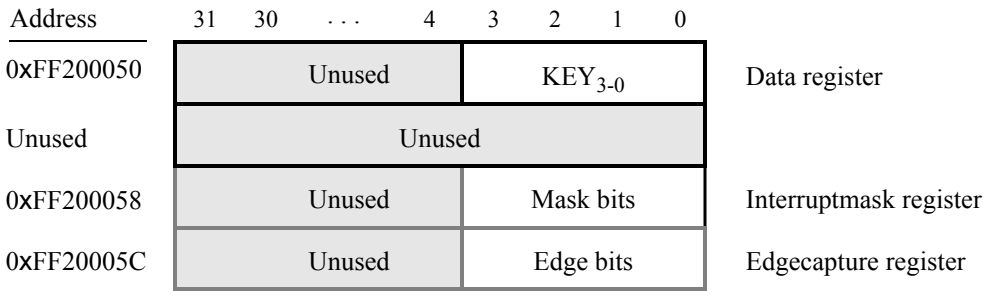

Figure 3: The parallel port connected to the pushbutton *KEYs*.

3. Make a new Monitor Program project in the folder where you stored the *part1.s* file.
4. Compile, download, and test your program.

Part II

Write an ARM assembly language program that displays a two-digit decimal counter on the seven-segment displays *HEX*$1-0$. 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:  LDR   R7, =200000000   // delay counter
SUB_LOOP:  SUBS  R7, R7, #1
           BNE   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; it remains set until your program resets it back to 0. You reset an *Edgecapture* bit by writing a 1 into the corresponding position of the register.

Perform the following:

1. Create a new folder to hold your solution 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.
3. Compile, download, and test your program.

Part III

In Part II you used a delay loop to cause the ARM 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 ARM processor to wait for the timer.

The DE1-SoC Computer includes a number of hardware timers. For this exercise use the timer called the ARM A9 *Private Timer*. As shown in Figure 4 this timer has four registers, starting at the base address 0xFFFEC600. To use the timer you need to write a suitable value into the *Load* register. Then, you need to set the enable bit E in

the *Control* register to 1, to start the timer. The timer starts counting from the initial value in the *Load* register and counts down to 0 at a frequency of 200 MHz. The counter will automatically reload the value in the *Load* register and continue counting if the A bit in the *Control* register is set to 1. When it reaches 0, the timer sets the F bit in the *Interrupt status* register to 1. You should poll this bit in your program to cause the ARM processor to wait for the timer. To reset the F bit to 0 you have to write the value `1` into this bit-position.

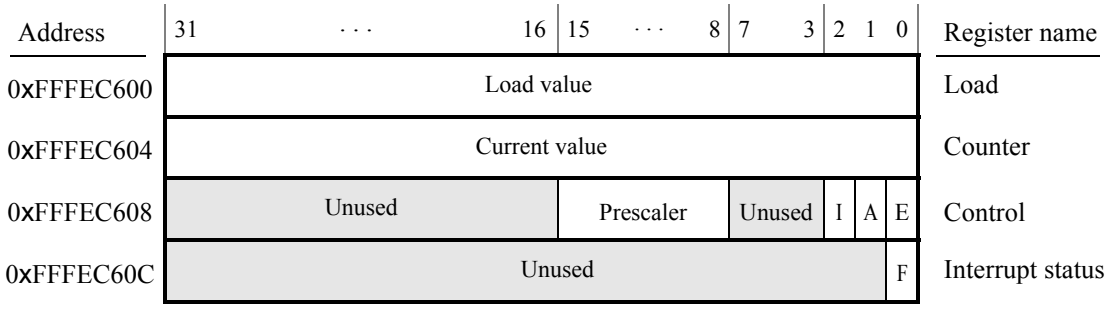

Figure 4: The ARM A9 Private Timer registers.

Perform the following:

1. Make a new folder to hold your solution for this part. Create a file called *part3.s* and type your assembly language code into this file.
2. 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 ARM A9 Private 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.

Perform the following:

1. Make a new folder to hold your solution for this part. Create a file called *part4.s* and type your code into this file.
2. Make a new Monitor Program project for this part of the exercise, and then compile, download, and test your program.