Laboratory Exercise 8

Memory Mapped I/O, Polling and Timers

Revision of November 11, 2025

In this lab, you will learn how to use devices that provide input and output capabilities for a processor and understand how a processor connects to these inputs and outputs using Memory Mapped I/O. You will also be introduced to timers, starting with software delay loops, and then progressing to the use of hardware timers with one of the NIOS-V timers.

Note: This lab will be marked entirely **in-person during your scheduled practical session via a demo to your TA by the end of your assigned practical session. It is STRONGLY recommended that you make good progress on the lab prior to coming to your in-person PRA session. There is no tester for this lab but you are still required to submit your code.**

1 Background

There are two basic techniques for synchronizing with I/O devices: program-controlled *polling* and *interrupt-driven* approaches. We will use the polling approach in this exercise, and interrupts will be covered in the next lab.

In general, an *embedded system* like the one we are using in the lab, has what is called a *parallel port* that provides for data transfer to or from external input/output devices such as the LEDs or Switches or Keys on the DE1-SoC board. The transfer of data may involve from 1 to 32 bits, and we call it 'in parallel' if there is more than 1 bit at a time (and 'serial' if just one bit at a time). The number of bits, n, and the type of transfer depend on the specific parallel port being used. The parallel port interface we will use contains the four registers shown in Figure 1. Although we are calling these registers, they shouldn't be confused with the 32 registers in the processor. These reside in the input/output unit, and are accessed through memory mapping, and so have a specific addressed assigned to them. Each register is n bits long. The registers have the following roles:

- *Data* register: holds the n bits of data that are transferred between the parallel port and the NIOS-V 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 V processor.

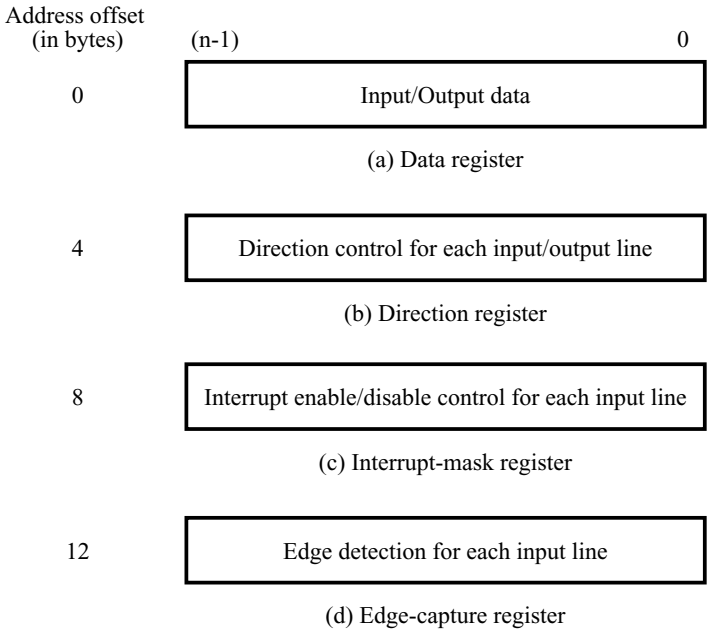

Figure 1: Registers in the parallel port interface.

Not all of these registers are present in some parallel ports. For example, the *Direction* register is included only when a bidirectional interface is permitted by the parallel port. 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.

2 Part I

You are to write a NIOS V assembly language program that displays a binary number on the 10 LEDs (that you've been using in the previous labs) under control of the four pushbuttons (also called KEYs) on the DE1-SoC board.

The DE1-SoC Computer contains a parallel port connected to 10 red LEDs on the board. Figure 2 shows the address we've used for the LEDs and picture of the register, taken from page 6 of the `DE1-SoC_Computer_NiosV.pdf` document that was given out in Lab 1.

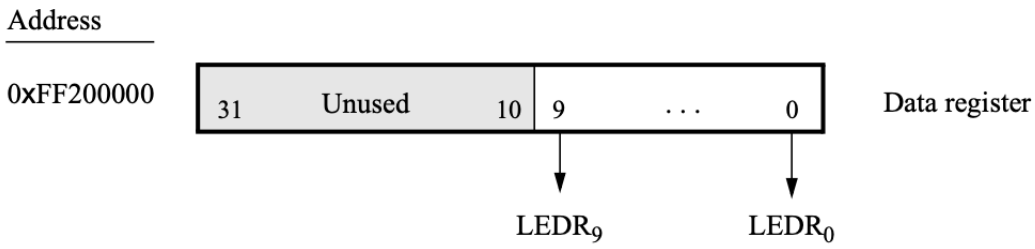

Figure 2: The parallel ports connected to the 10 red LEDs

The following functionality should be included:

- If KEY_0 is pressed on the board, you should set the binary number displayed on the 10 LEDs to be 1 in base 10, which we will annotate from now on as 1_{10} (or in base 2 as 0000000001_2).
- If KEY_1 is pressed, you should increment the displayed number, but don't let the number go above 15_{10} (i.e., pressing the key won't change the value if it is already at 15_{10} or 1111_2).
- If KEY_2 is pressed, you should decrement the number, but don't let the number go below 1 (i.e. pressing the key won't change the value on the LEDs if it is already 1).
- Pressing KEY_3 should blank the display (which is the same as 0), and pressing any other KEY after that should return the display to 1.
- The parallel port connected to the pushbutton *KEYs* has the base address 0xFF200050, as illustrated in Figure 3. In your program, use the *polling* I/O method 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

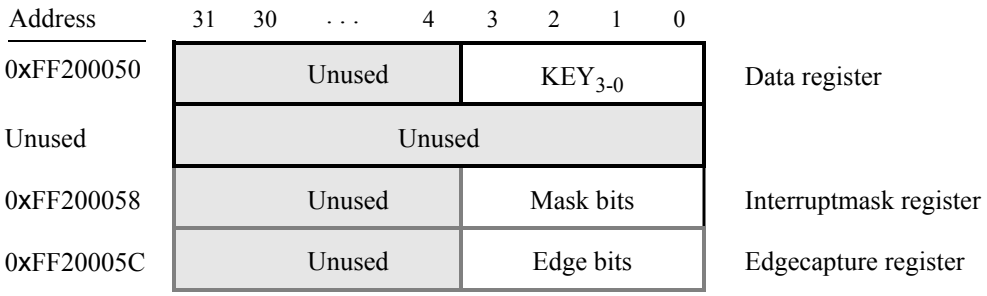

Figure 3: The parallel port connected to the pushbutton *KEYs*.

detected, be sure that your program waits until the button is released. IMPORTANT: You *must not* use the *Interruptmask* or *Edgecapture* registers for this part of the exercise. Doing this way first (a later section changes this) will teach you partly why the edge capture register and associated hardware are useful.

All of the above functionality should be developed prior to coming to lab so you are ready to demo this part as soon as lab starts.

Create a new folder to hold your solution for this part and put your code into a file called `part1.s`. Show your code working to your TA for grading in the lab. You will be graded based on your code running on the board, not the simulator. You can develop this code in the simulator but you should familiar with how to run the Monitor program to demo the code on the boards in lab.

3 Part II

Write a NIOS V assembly language program that displays binary *counter* on the 10 LEDs. The counter should be incremented approximately every 0.25 seconds. When the counter reaches the value 255_{10}, 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, which gives a good value for the delay when using a NIOS V processor on the DE1-SoC board (e.g., 10,000,000). For the CPUlator, a much smaller delay value (e.g., 500,000) has to be used, because the *simulated* NIOS V processor in the CPUlator tool "executes" code *much* more slowly than the real NIOS V processor on the DE1-SoC board.

```
DO_DELAY:   la     s0, COUNTER_DELAY
```

```
SUB_LOOP:   addi   s0, s0, -1
            bnez   s0, 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 at the value of 1 until your program does something specific to set it back to 0, as follows: to reset a specific bit of the *Edgecapture* register your program must 'store' a '1' into it. Yes, that's correct, in our experience many seeing this for the first time find it confusing, to repeat: you store a 1 into a specific bit into the memory mapped register (leaving all other bits that you don't want reset at 0), and this causes that specific bit to be reset to 0. Also, the other bits that are stored as 0 do not change the value of the Edge Capture register. This is achieved through digital logic hardware that receives the stored value from the processor, and is designed to have this (somewhat counter-intuitive, but well-motivated) behaviour.

Put your code into a folder called part2 and a file called `part2.s`, and test and debug your program and show it working to your TA for grading. It is **strongly recommended** that you complete the majority of part 2 prior to coming to lab.

4 Part III

In Part II, you used a delay loop to cause the NIOS V 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 polling I/O to cause the NIOS V processor to wait for the timer.

The DE1-SoC Computer includes a number of hardware timers. For this exercise, you will use one of the two timers available to NIOS V. As shown in Figure 4 this timer has six registers, starting at the base address 0xFF202000 (0xFF202020 for the second timer). To use the timer you need to write a suitable value into the *Counter start* registers. Then, you need to set the enable bit `START` in the *Control* register to 1, to start the timer. The timer starts counting from the initial value in the *Counter start* register and counts down to 0 at the precise rate of 100 MHz, which means that the count goes down by one every 10ns.

The counter will automatically reload the value in the *Load* register and continue counting if the `CONT` bit in the *Control* register is set to 1. When it reaches 0, the timer sets the `TO` bit in the *status* register to 1. You should poll this bit in your program to make the NIOS V processor wait for this event to occur. To reset the `TO` bit to 0 you have to write the value `0` into this bit-position.

Put your code into a folder called part3 and a file called `part3.s`, and test and debug your

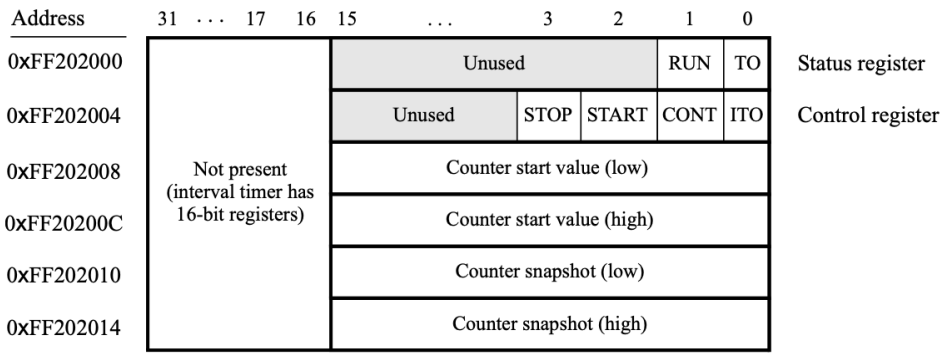

Figure 4: The Interval Timer registers.

program and show it working to your TA for grading. ALL code must be demo'd for your TA prior to leaving lab. There will be no other time to demo working code.

5 Submission and grading

You must submit all 3 parts of lab using the submit command. You will be graded based on a demo of your working code to your TA **during your practical session.** TA will test the functionality and ask you questions about your code. It will take the TAs time to mark everyone's demos to please come to lab with much of the lab work completed already. **Do not leave the lab before having your code marked by the TA.**