ECE 243S - Computer Organization
February 2026
Lab 4

Memory Mapped I/O, Polling and Timers

Due date/time: During your scheduled lab period in the week of February 2, 2026. [Due date is not when Quercus indicates, as usual.]

"Everything should be made as simple as possible, but not simpler." - Albert Einstein

Learning Objectives

The goal of this lab is to explore the use of devices that provide input and output capabilities for a processor, and to see how a processor connects to those inputs and outputs, through the Memory Mapped Input/Output method. You'll learn how to synchronize between device signals and the computer using the *polling* method. You'll also be introduced to a device that has a special purpose in computer systems, the Timer, which is used for precise measurement and allocation of time within a computer.

In addition, we will now expect you to make appropriate use of subroutines.

What To Submit

You should hand in the following files prior to the end of your lab period. However, note that the grading takes place in-person, with your TA:

- The assembly code for Parts I,II, III and IV in the files `part1.s`, `part2.s`, `part3.s` and `part4.s`.

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 lab, 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 handles data transfer to and 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. We call it 'serial' data transmission if just one bit goes at a time . The number of bits, n, and the type of transfer depend on how a paralel port is set up.

The parallel port interface we will use contains the four registers shown in Figure 1. These registers are *distinct* from the 32 registers in the Processor. They reside in the input/output unit, and as taught in class, are accessed through memory mapping, and so have a specific addressed assigned to them. Each register is n bits long, where n is mostly 32. 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 value is detected in the input wires connected to the parallel port. Once a bit in the edge capture register is turned on, it will remain that way, until something explicitly resets it, which is typical a store operation from the processor.

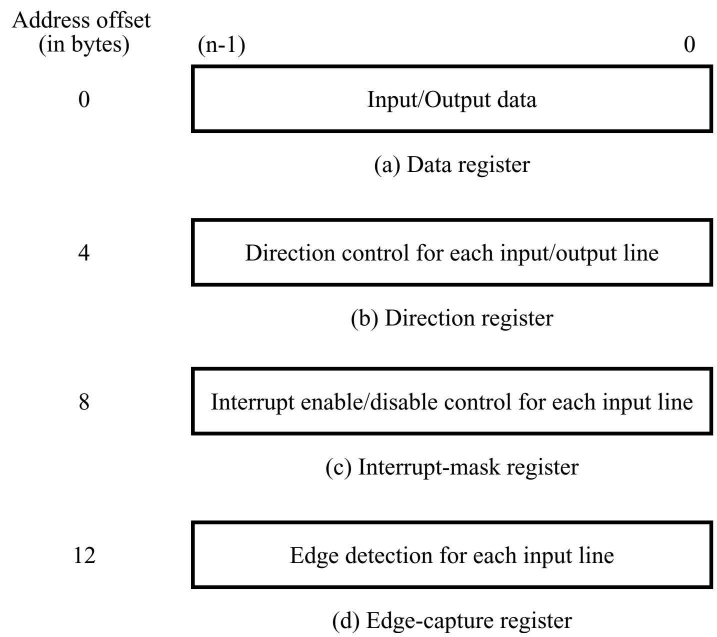

Figure 1: Registers in the parallel port interface.

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.

The Nios V/RISC V Subroutine Calling Convention

Note that in this lab you are expected to use subroutines to make your code appropriately modular.
As discussed in class, the NIOS V Subroutine Calling Convention provides guidelines ("rules") for how registers should be used in a program, especially when subroutines are involved. They use the nomenclature that the code calling the subroutine is the 'caller' and the code being called is the 'callee.' The full rules are as follows:

Registers `t0` to `t6` are *caller saved* meaning that it is the job of the caller code to save these registers on the stack to preserve because they are allowed to be used by a subroutine. The caller would restore the registers once the subroutine returns.

Registers `s0` to `s11` are callee saved registers meaning that it is the job of the subroutine itself, to save these registers on the stack at the beginning of the subroutine if they are to be used in the subroutine.

The proper procedure for passing parameters to a subroutine is to use registers `a0` to `a7`. If there are more than 8 parameters that have to be passed to a subroutine they have to be saved in the stack in reverse order meaning that if we have 10 parameters as an example, we would save parameter 10 first, and then parameter 9. Inside the subroutine we would pop parameter 9 first and then parameter 10.

To return a value from a subroutine, the `a0` and `a1` registers are used. If more results are to be returned, then the stack is used. The callee pushes items on the stack and the caller pops them off.

(Important Note: when you are using CPUlator you will see an error stating that you have "clobbered" one or more of the registers `s0` to `s11` if you over-write these registers in a subroutine without saving/restoring their values as required by the calling convention.)

You should start using this calling convention for all NIOS V code that you write in this lab, in all subsequent lab exercises, and on all test/exam questions in this course.

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 as described below.

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. You may wish to read that document more now that we're getting deeper into the labs in this course.

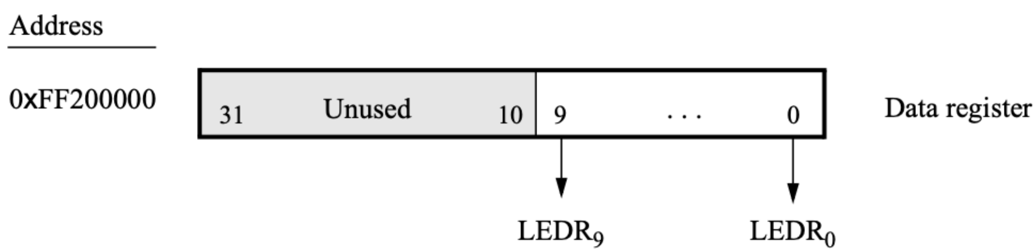

Figure 2: The parallel ports connected to the 10 red LEDs

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 then 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 then 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.

Address	31 30 … 4	3 2 1 0	
0xFF200050	Unused	$KEY_{3\text{-}0}$	Data register
Unused	Unused		
0xFF200058	Unused	Mask bits	Interruptmask register
0xFF20005C	Unused	Edge bits	Edgecapture register

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

- 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. 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 associate hardware are useful.

Create a new folder to hold your solution for this part and put your code into a file called `part1.s`. You can debug your program at home using CPUlator, and you will test it on the DE1-SoC board in the lab and demonstrate it to your TA for grading. Submit `part1.s` to Quercus before the end of your lab period.

Part II

Write a NIOS V assembly language program that displays a 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 Edgecapture register and this causes that specific bit to be reset to 0. Assuming you only wanted to reset that specific bit, you would make the other 3 bits stored in the Edgecapture register equal to 0. This is interpreted as 'don't change' those other bits.

This functionality is achieved through digital logic hardware that receives the stored value from the processor, and is designed to have this (apparently 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. Submit `part2.s` to Quercus before the end of your lab period.

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 (also at 0xFF202020 for the second timer). To use the timer you need to write a suitable value into the *Counter start* registers. Then, you must 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.

When it counts down to 0, 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.

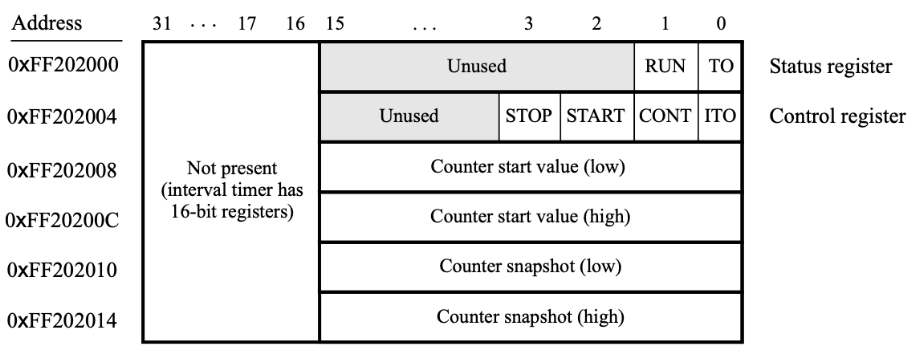

Figure 4: The Interval Timer registers.

Put your code into a folder called part3 and a file called `part3.s`, and test and debug your program and show it working to your TA for grading. Submit `part3.s` to Quercus before the end of your lab period.

Part IV

In this part you are to write an assembly language program that implements a real-time binary clock. Display a binary version of time on the 10 LEDs on the DE1-SoC board. We want to display both seconds and hundredths of a second. The seconds should be displayed on the high-order 3 bits of the red LEDS (i.e. LEDR9:7) and the hundredths of seconds should be displayed on the low-order 7 bits (i.e. LEDR6:0). Your code should use a single timer to count both seconds and hundredths of a second, storing these values in separate registers. This requires careful consideration of how to track elapsed hundredths and seconds with just one timer..

Measure time intervals of 0.01 seconds in your program by using polled I/O with the Timer. You should be able to stop/run the clock by pressing any pushbutton *KEY*. When the clock reaches 8_{10} seconds 99_{10} hundreths, it should wrap around to 000:0000000.

Put your code into a folder called part4 and a file called `part4.s`, and test and debug your program and show it working to your TA for grading. Submit `part4.s` to Quercus before the end of your lab period.