ECE 243S - Computer Organization
February 2024
Lab 4

Memory Mapped I/O, Polling and Timers

Due date/time: During your scheduled lab period in the week of February 5, 2024. [Due date is not when Quercus indicates, as usual.]

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 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 the computer.

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 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 (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 as taught in class, they 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 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.

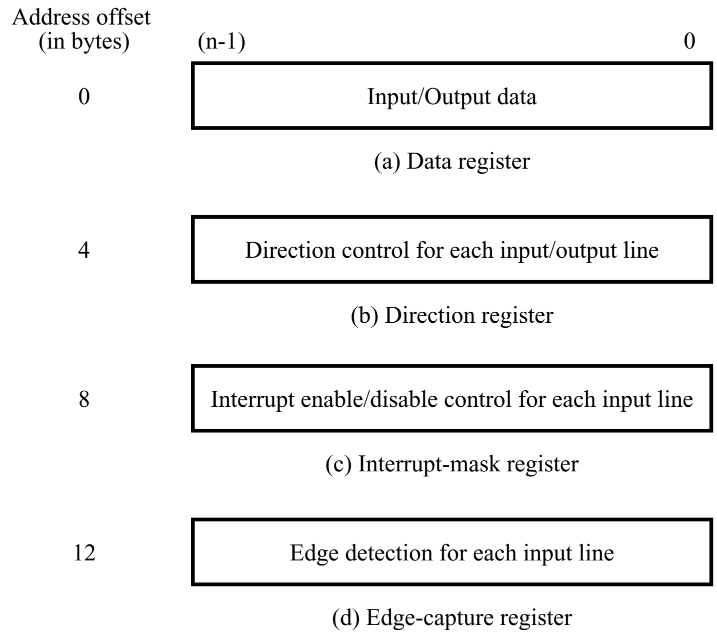

Figure 1: Registers in the parallel port interface.

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 II Subroutine Calling Convention

As discussed in class, the NIOS II *Subroutine Calling Convention* provides guidelines ("rules") for how registers should be used in a program, especially when subroutines are involved. It states that a subroutine is allowed to modify only NIOS II registers `r8` to `r15`, but it is not permitted to have changed the contents of NIOS II registers `r16` to `r23` after execution of the subroutine. If you need to use registers `r16` to `r23` inside a subroutine, then you must save their current values by pushing them onto the stack at the beginning of the subroutine, and then restore these saved values by popping them off the stack before returning from the subroutine. (Note: if you are using CPUlator, then you will see an error stating that you have "clobbered" one ore more of the registers `r16` to `r23` if you over-write these registers in a subroutine without saving/restoring their values as required by the calling convention.)

The proper procedures for passing parameters into a subroutine, and returning results, are also specified by the calling convention. It states that parameters have to be passed to a subroutine using registers `r4` to `r7`. If you need to transmit more than four parameters into a subroutine, then the extra ones should be passed through the stack, where the caller must both push them onto the stack, and remove them upon return from the callee. Similarly, at most two results can be returned from a subroutine using register `r2` and `r3`. If more results need to be returned, then the stack is also used.

You should start using this calling convention for all NIOS II 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 II 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 4 of the `DE1-SoC_Computer_NiosII.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.

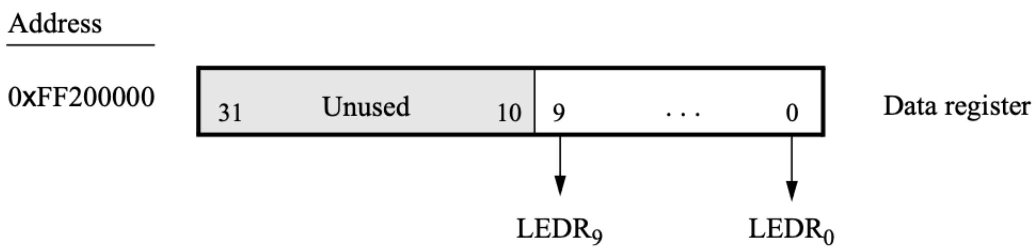

Figure 2: The parallel ports connected to the 10 red LEDs

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.
- 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 will need to make a Monitor Program project for running your code on a DE1-SoC board; but you can also debug your program at home using CPUlator. Show it working to your TA for grading in the lab. Submit `part1.s` to Quercus before the end of your lab period.

Part II

Write a NIOS II 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.

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*.

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 II 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 II processor in the CPUlator tool "executes" code *much* more slowly than the real NIOS II processor on the DE1-SoC board.

```
DO_DELAY:   movia   r8, COUNTER_DELAY
SUB_LOOP:   subi    r8, r8, 1
            bne     r8, r0, 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 (bits 0-3) 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 (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 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 polling I/O to cause the NIOS II 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 II. 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 II 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.

Address	31 . . . 17 16	15 . . .	3	2	1	0	
0xFF202000	Not present (interval timer has 16-bit registers)	Unused			RUN	TO	Status register
0xFF202004		Unused	STOP	START	CONT	ITO	Control register
0xFF202008		Counter start value (low)					
0xFF20200C		Counter start value (high)					
0xFF202010		Counter snapshot (low)					
0xFF202014		Counter snapshot (high)					

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 the time on the 10 LEDs in the following format: SS:DD, where *SS* are seconds and *DD* are hundredths of a second. The *SS* should be displayed on the high-order 5 bits of the red LEDS (i.e. LEDR9:5) and the DD should be displayed on the low-order 5 bits (i.e. LEDR4:0). Your code should keep the seconds and hundreth seconds counting separately, which means you'll need to write think about how to code so that they work correctly together.

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 59_{10}:99_{10}, it should wrap around to 00:00.

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.