**Queen's University**
**Department of Electrical and Computer Engineering**
**ELEC 371 Microprocessor Interfacing and Embedded Systems**
**Fall 2018**

# Lab 2:
# Timer Interrupts, Multiple Interrupt Sources, and Character Input/Output Subroutines

## Objectives

This laboratory activity for *ELEC371* provides the opportunity for students to:

- learn about asm.-lang. interrupt programming with a timer interface,
- acquire experience in handlinh multiple interrupt sources in asm.-lang. programs,
- and become (re)acquainted with character input/output support in subroutine form.

Preparation is specified in this document. The intent is for <u>*each individual student*</u> to make an honest effort to pursue the specified preparation without assistance from others to the extent that is possible. In this manner, a reasonable basis exists for useful discussion with others, and learning can thereby be made more effective.

## Preparation BEFORE Your Scheduled Laboratory Session

- This laboratory exercise introduces timer interrupts, reuses pushbutton interrupts from previous activity, and involves character input/output capabilities through the use of modular subroutines.
- The following pseudocode is intended to provide an initial exposure to basic use of the original timer interface in the DE0 Computer (which is also included in the new QUEENS Computer).

```
main()::
    Init()    /* subroutine to perform initialization */
    loop
        /* nothing */
    end loop

Init()::
    turn off LEDs
    clear extraneous pending timer interrupt
    num_cycles = 25000000  /* with 50-MHz clock, this is 0.5 sec */
    write upper/lower 16 bits of num_cycles value
        to timer start hi/lo registers
    write 0x7 to timer control register to start timer
        in continuous mode with interrupts
    set bit for timer interface in ienable register
    set IE bit in Nios II status register to 1

/* the isr is _NOT_ a normal subroutine, but modified regs.
   must be saved/restored (except special ones) */

isr()::
    perform processor-specific register update
    read special register for pending interrupts
    if (interrupt source is the timer) then
        clear the timer interrupt
    toggle LED bit 0 using XOR operation
    end if
    return from interrupt  /* distinct from normal subroutine */
```

- Prepare a complete Nios II assembly-language program by introducing code in the appropriate places of the template source file `lab2_part1.s` that is posted on the course Webpage.
- Consult page 4 of the Lab 1 description for references to appropriate parts of vendor-provided technical documentation where information on interrupt-based programming is available. The sample code in vendor documentation illustrates certain technical details, but you should closely follow pseudocode specifications and develop assembly-language code in this course in the manner described by the instructor, including a strong emphasis on modularity.
- Use the on-line CPUlator simulator tool to test the execution of your completed program and verify that the expected behavior (LED blinking on and off) is achieved. In simulation, the rate of blinking will not be exact, and it may vary from computer to computer. Nonetheless, if it blinks as expected, execution of the same program on hardware will lead to a precise rate of blinking for the LED.

- In the previous laboratory exercise, interrupts were introduced through the pushbutton interface. The preparation for this laboratory exercise has separately explored the timer interface for interrupts. The next step is to handle multiple interrupt sources in the same program.
- Prepare a new file `lab2_part2.s` where both pushbutton and timer interrupts will be combined.
- One important change for the combined program is to use different LEDs for the different interrupt sources. Therefore, when you copy your timer-related code into the new `lab2_part2.s` file, modify the timer interrupt handling to toggle bit **1** of the LEDs instead of bit 0. Let the code that responds to the pushbutton interrupt source toggle LED bit 0 as done originally for Lab 1.
- Another important change for the combined program is to properly check for the different interrupt sources within the interrupt-service routine. For reference purposes, consult slides 24 to 26 in the Chapter 3 material, but also note that the instructor has intentionally included two interrupt sources in the first question of Tutorial 3, hence that tutorial discussion is highly relevant for this laboratory exercise. Finally, there is also the vendor documentation (see page 4 of Lab 1), but once again, you should follow the guidelines of the instructor for properly-structured code.
- The initialization subroutine must now prepare the hardware interfaces for both interrupt sources.
- The `ienable` register in the processor must also be prepared for both interrupt sources.
- It is your choice as to whether to have dedicated subroutines for separate handling of the pushbutton and timer sources (again, see Chapter 3 material and Tutorial 3 for guidance), or whether to have a longer interrupt-service routine containing all of the code for both sources. (The instructor will always recommend the more modular approach with subroutines, and for more complex applications the value of the modular approach only increases.)
- Furthermore, if you choose to use dedicated subroutines as indicated above, it is also your choice on whether to clear the interrupt source in the interrupt-service routine before calling the dedicated subroutine to perform the task(s) for servicing the interrupt, or whether to clear the interrupt source within the dedicated subroutine. Either way, do not forget to clear the interrupt source.
- If you use subroutine calls within the interrupt-service routine, do not forget that you must save additional information on the stack to prevent incorrect execution after the main program resumes.
- Test your `lab2_part2.s` file with the simulator and confirm that the LED controlled by the timer interrupt blinks continuously, whereas the LED controlled by the pushbutton only toggles in response to pressing/releasing the pushbutton (clicking twice on the box for the pushbutton).

- The final part of this laboratory exercise involves adding support for two character input/output subroutines to be used by the main program, while also having multiple interrupt sources.
- Review Section 2.4 of the document entitled *Basic Computer for the Altera DE0 Board*. Because nothing is “controlled” in this activity, the register at 0x10001004 will be used for “status” only.
- Copy the `lab2_part2.s` file into a new `lab2_part3.s` file.
- The first subroutine, *PrintChar(ch)*, sends the character *ch* (in register *r2* using the convention for passing parameters) to the JTAG UART of the DE0/QUEENS Computer systems. The subroutine must first read a word from the “status” register at address 0x10001004 and check whether the upper 16 bits of the word are zero (using the *andhi* instruction). If the bits are zero, then there is no space for a character, hence the subroutine should *poll* until space is available. If space is available, then character *ch* should be written (as a ***word***) to the “data” register at 0x10001000.

```
PrintChar(ch)::
    do
        wspace = read JTAG UART status register
        wspace = wspace AND 0xFFFF0000
    while (wspace is zero)
    write ch to JTAG UART data register
```

- Appropriately modular code for an assembly-language subroutine that implements *PrintChar(ch)* is provided below, along with relevant equate directives to place at the top of the source file.

```
        .equu   JTAG_UART_BASE,    0x10001000  # address of first JTAG UART register
```

```
        .equ    JTAG_UART_BASE,    0x10001000  # address of first JTAG UART register
        .equ    DATA_OFFSET,       0           # offset of JTAG UART data register
        .equ    STATUS_OFFSET,     4           # offset of JTAG UART status register
        .equ    WSPACE_MASK,       0xFFFF      # used in AND operation to check status

        ⋮

PrintChar:
        subi    sp, sp, 8        # adjust stack pointer down to reserve space
        stw     r3, 4(sp)        # save value of register r3 so it can be a temp
        stw     r4, 0(sp)        # save value of register r4 so it can be a temp
        movia   r3, JTAG_UART_BASE    # point to first memory-mapped I/O register
pc_loop:
        ldwio   r4, STATUS_OFFSET(r3) # read bits from status register
        andhi   r4, r4, WSPACE_MASK   # mask off lower bits to isolate upper bits
        beq     r4, r0, pc_loop       # if upper bits are zero, loop again
        stwio   r2, DATA_OFFSET(r3)   # otherwise, write character to data register
        ldw     r3, 4(sp)        # restore value of r3 from stack
        ldw     r4, 0(sp)        # restore value of r4 from stack
        addi    sp, sp, 8        # readjust stack pointer up to deallocate space
        ret                      # return to calling routine with result in r2
```

- Implement the code above for *PrintChar(ch)* in your new `lab2_part3.s` file.
- To complete the support for character-based subroutines, consider the following pseudocode for *GetChar()* that returns a character (in register *r2* as dictated by parameter-passing convention) after polling the JTAG UART interface until it is indicating that a character is available.

```
GetChar(ch)::
    do
        bits = read JTAG UART data register
    while ((bits AND 0x8000) is zero)      /* note: _not_ 'andhi' */
    ch = bits AND 0xFF
    return ch
```

- *Note how the original information read from the data register is preserved*. The first AND operation does not change that information. If bit 15 (RVALID) is set, meaning that a character is available, then *the character is already in the same word that was read from the data register*. The hardware for the interface works in this manner. The character should be extracted from the lowest eight bits with a separate AND operation. You should *not* read the data register a second time.
- For the assembly-language implementation, introduce one more equate directive for the pattern used to check the RVALID bit, then prepare the code for a modular subroutine *GetChar(ch)* in your new `lab2_part3.s` file. Use the code for the printing subroutine as a guide.
- Finally, change the main program in the `lab2_part3.s` file to the pseudocode below.

```
main()::
    Init()
    loop
        ch = GetChar()
        PrintChar(ch)
    end loop
```

## In-Lab Part 1: Testing Timer-Interrupt Program on Hardware

- On your network storage, create a folder `Z:\My Files\ELEC371\LAB2_PART1`.
- Place your prepared `lab2_part1.s` in the folder named above.
- Plug the DE0-CV board into the computer using the USB cable, and turn the power on.
- Start the Altera Monitor Program on the PC computer. An easy method for doing so is to type "monitor" in the Search feature of the MS-Windows Start menu. The icon for the Altera Monitor Program should appear in the results of the search, and you can click on the icon.
- Once the Monitor Program opens, choose *File* → *New Project*.
- In the dialog box, use *Browse…* to navigate to `Z:\My Files\ELEC371\LAB2_PART1`.
- For the project name, enter `lab2_part1`, then press *Next*.
- Ensure that you have downloaded the instructor-provided files for the QUEENS Computer.
- Select <u>&lt;Custom System&gt;</u> from the pulldown menu under the "Select a system" heading.
- Set the system information file to the `.qsys` file that is provided by the instructor.
- Set the programming file to the `.sof` file that is provided by the instructor (it is *not* optional when the aim is to execute code on the hardware; the FPGA chip must be configured with this file).
- Once the two files have been identified properly for a custom system, press *Next*.
- Select *Assembly Program* as the program type, then press *Next*.
- Use *Add…* to include the `lab2_part1.s` file in the list, then press *Next*.
- Make sure that the *Host connection* shows *USB-Blaster*.
- In the final setup window, simply press *Finish* to complete the project creation task.
- Press *Yes* to download the project's system to the board, and press *OK* when that task is done.
- Press or select *Actions* → *Compile & Load* to generate and download the machine code.
- Press or select *Actions* → *Continue* to initiate execution of the program.
- Verify proper functionality by confirming that the LED blinks on or off every 0.5 second.

## In-Lab Part 2: Testing Combined Timer/Pushbutton Interrupt Program

- On your network storage, create a folder `Z:\My Files\ELEC371\LAB2_PART2`.
- Place your prepared `lab2_part2.s` in the folder named above.
- In the Monitor Program opens, choose *File* → *New Project*.
- In the dialog box, use *Browse…* to navigate to `Z:\My Files\ELEC371\LAB2_PART2`.
- For the project name, enter `lab2_part2`, then press *Next*.
- Select <Custom System>, set the `.qsys` file, set the `.sof` file, then press *Next*.
- Select *Assembly Program* as the program type, then press *Next*.
- Use *Add…* to include the `lab2_part2.s` file in the list, then press *Next*.
- Make sure that the *Host connection* shows *USB-Blaster*.
- In the final setup window, simply press *Finish* to complete the project creation task.
- Press *Yes* to download the project's system to the board, and press *OK* when that task is done.
- Press or select *Actions* → *Compile & Load* to generate and download the machine code.
- Press or select *Actions* → *Continue* to initiate execution of the program.
- Verify that the LED blinks every 0.5 sec, and that the other LED toggles with the pushbutton.

## In-Lab Part 3: Testing Combined Program with Character Input/Output

- On your network storage, create a folder `Z:\My Files\ELEC371\LAB2_PART3`.
- Place your prepared `lab2_part3.s` in the folder named above.
- In the Monitor Program opens, choose *File* → *New Project*.
- In the dialog box, use *Browse…* to navigate to `Z:\My Files\ELEC371\LAB2_PART3`.
- For the project name, enter `lab2_part3`, then press *Next*.
- Select <Custom System>, set the `.qsys` file, set the `.sof` file, then press *Next*.
- Select *Assembly Program* as the program type, then press *Next*.
- Use *Add…* to include the `lab2_part3.s` file in the list, then press *Next*.
- Make sure that the *Host connection* shows *USB-Blaster*.
- In the final setup window, simply press *Finish* to complete the project creation task.
- Press *Yes* to download the project's system to the board, and press *OK* when that task is done.
- Press or select *Actions* → *Compile & Load* to generate and download the machine code.
- Press or select *Actions* → *Continue* to initiate execution of the program.
- Click the mouse button with the mouse pointer over the *Terminal* section of the Monitor Program. This step is necessary to ensure that the Monitor accepts keyboard input.
- Verify that the program correctly "echoes" each character that you type on the keyboard.
- Meanwhile, one LED blinks every 0.5 sec, and the other LED toggles with the pushbutton.
- Move the DE0-CV board to be able to view both the Terminal window on the screen and the LEDs on the board, then type characters and press the pushbutton at the same time.
- Even though the system processes instructions in sequence and is either executing code in the main program or in the interrupt-service routine, operation of the system at 50 MHz is such that to a human the apparent behavior is that all of the activity is happening simultaneously.

## In-Lab Part 4: Extension of Combined Program with Character Input/Output

- Using the specifications provided to you in the lab, prepare a new file `lab2_part4.s` for a program that is an extension of the working code from Part 3.
- Create a folder `Z:\My Files\ELEC371\LAB2_PART4` containing your new file.
- Create and properly set up a new project `lab2_part4` in the Monitor Program.
- Compile and load the program to verify that it functions as intended.

***DEMONSTRATE THE WORKING PART 4 PROGRAM TO OBTAIN CREDIT.***