

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

Lab 1:
Review of Assembly-Language Programming,
Use of Parallel Input/Output Interfaces,
and Introduction to Hardware Interrupts

Copyright © 2018 by Dr. Naraig Manjikian, P.Eng.
All rights reserved.

*Any direct or derivative use of this material
beyond the course and term stated above
requires explicit written consent from the author,
with the exception of future private study and review
by students registered in the course and term stated above.*

Objectives

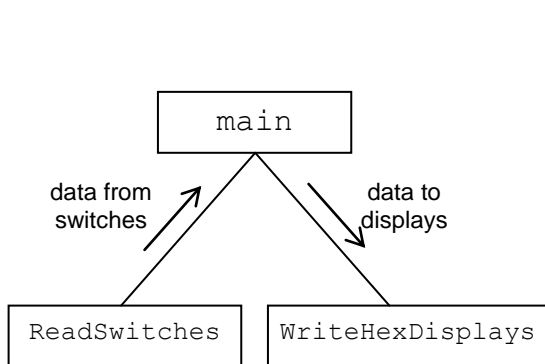
This laboratory activity for *ELEC371* provides the opportunity for students to:

- review assembly-language programming for the Nios II through development of modular code, as part of preparation before attending a laboratory session and also as in-lab activity,
- consider code for accessing basic parallel input/output interfaces,
- and acquire initial exposure to the details of interrupt-based code that must perform special initialization to configure input/output interfaces and the processor for interrupt processing.

Preparation is specified in this document. The intent is for each individual student 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

- One of the aims of this first laboratory exercise is to describe a methodology for software design and implementation, and to encourage the adoption and application of this methodology. Use of a diagram and associated pseudocode to specify a program is highlighted. From that pseudocode, it is expected that appropriately modular and well-structured assembly-language code is developed.
- For Part 2, the following diagram shows the call/return and parameter-passing relationships between the modules or subroutines, and the pseudocode provides the details of each module.



```

main():: /* variable data is local */
loop
    data = ReadSwitches()
    WriteHexDisplays (data)
end loop
  
```

```

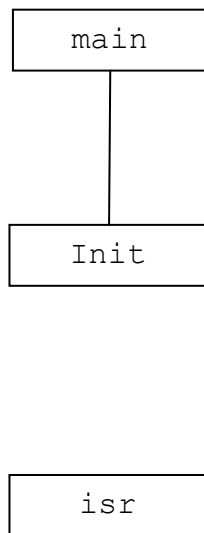
ReadSwitches()::
    data = read from switches data register
    return data
  
```

```

WriteHexDisplays(data)::
    write data to hex displays data register
  
```

- Note that the module diagram does not reflect the number of times or the order for the calls to the subroutines. It simply shows that there is *at least one call at some point to each subroutine*. For a more complex program, there may be many calls in different orders to modules and submodules.
- Implement a complete assembly-language program** in a single source file `lab1_part2.s` and use the on-line *CPUlator* tool to generate executable code and verify correct program behavior. Transformation of the pseudocode above into Nios II assembly language can be done in less than 20 instructions, including stack pointer initialization and saving/restoring registers in subroutines.
- In general for any program specifications in pseudocode, the nature of the application combined with the pseudocode and related comments makes it possible to identify the variables that are parameters, the variables that are local to each module or subroutine (including variables that are used for preparing and holding return values), and variables that are global to the entire program.
- Parameters to a subroutine are normally in registers. For assembly-language programming in this course on the Nios II processor, the expected convention for code is to use registers `r2`, `r3`, `r4`, etc. Register `r0` is always 0, and register `r1` is avoided because it is reserved for use by the assembler. A subroutine return value is returned in register `r2`; calling code should be written accordingly.
- A local variable can normally be register-allocated, provided that the register value is first saved. The one exception is the main routine which does not need to save/restore register values because the main program does not return to other calling code that expects unmodified registers.
- On the ELEC 371 Webpage with course material, consult the *Technical Documentation* section for reference documentation on the Nios II instruction set. There is also the document entitled *Basic Computer for the Altera DE0 Board* that provides information in Section 2 on the input/output interfaces for the switches and hex displays found on the circuit board used in the laboratory.
- In the *Course Material* section of the ELEC 371 Webpage, consult the two documents prepared by the instructor that provide guidelines on assembly-language programming (particularly for code with subroutines) and a tutorial on using the on-line *CPUlator* tool.

- Part 3 is an introduction to hardware interrupts. As the relevant concepts and processor-specific details are pursued in the lecture portion of the course, this initial exposure to the principles of interrupts and the preparation of interrupt-based programs will provide a useful foundation and reference. The remaining laboratory exercises will all have aspects related to interrupts.
- The diagram and pseudocode below provide the specifications for another program in assembly language that captures three essential elements of interrupt-based software: (a) initialization of relevant hardware including the processor to properly recognize and respond to interrupt requests from hardware sources, (b) dedicated code that is invoked by the processor to service interrupt requests, and (c) the code of the main routine and any normal subroutine that it calls. In the context of interrupt-based software, the *main program* is the main routine and its subroutines.



(The interrupt-service routine is NOT called directly by any software. It is invoked by the processor in response to a hardware event, often related to input/output aspects. Hence, there is no line connecting this module to any other module in the diagram.)

```

main()::
  Init()
  local_var = 0      /* allocate in a reg. */
  loop
    local_var = local_var + 1
  end loop


Init():: /* this code uses special regs. */
  enable interrupts on pshbtn 0
  enable procr to recog. pshbtn interrupt
  enable procr to respond to all interrupts




/* this is NOT a normal subroutine, but
   modified regs. must be saved/restored
   (except special ones) */
isr()::
  perform processor-specific reg. update
  read special reg. with pending interrupts
  if (pshbtn interrupt is pending) then
    clear pshbtn interrupt source
    toggle LED0 using XOR operation
  end if
  return from interrupt
  
```

- Most of the time, the code in the main program is being executed. When a hardware source requests interrupt service, the processor saves state information (including the program counter value) related to the main program, then invokes the code in the interrupt-service routine to respond to the hardware interrupt request. Once the relevant processing is complete, a special return instruction is executed to recover the saved state information, which enables the processor to continue with the next instruction in the main program. With registers properly saved/restored by the interrupt-service routine, the fact that some other code was invoked at some arbitrary point during the execution of the main program is not apparent to the main program, *which is exactly the desired behavior so that main program code can easily be implemented in the usual manner.*
- For this exercise, the hardware source is pressing then releasing pushbutton 0 on the circuit board used in laboratory activity. A parallel input/output port associated with the pushbuttons must be configured for assertion of a hardware signal to the processor when this pushbutton event occurs.



- In addition to configuring the port interface, the processor itself must be configured to recognize the pushbutton port interrupt signal and the processor must also be configured to actually respond to *any* interrupt. The interrupt-service routine will not be invoked by the processor unless all three items (enabling interface to assert request, enabling processor to recognize source, and enabling processor to respond to any interrupt request) have been performed as part of initialization.
- Guidance on how the pushbutton parallel port interface found in the computer system used in laboratory activity can be configured for interrupt capability is found in Section 3.1.1 on page 16 of *Basic Computer System for the Altera DE0 Board*. Note that there is an error in Figure 16: although KEY2-1 is properly indicating two user pushbuttons are available, the surrounding box mistakenly covers three bits, but that box should only cover two bits. (On the DE0 board, the leftmost button is used as the system reset input for the computer system.)
- Guidance on configuration of the special `ienable` and `status` registers in the processor is found in Section 3.5 of *Basic Computer System for the Altera DE0 Board*. Special `wrctl` instructions must be used with these special processor registers to modify them. The global interrupt-enable bit in the `status` register is bit 0. As shown in Table 1 on page 15 in Section 3 of *Basic Computer System for the Altera DE0 Board*, the bit for the pushbutton port in the `ienable` register is bit 1. The text on page 15 does not explicitly state (although it should) that these IRQ identification bits corresponding bit positions in the `ienable` and `ipending` registers in the processor.
- The interrupt-service routine must check the contents of the special `ipending` register in the processor, which must be accessed with a special `rdctl` instruction. Section 3.5 of *Basic Computer System for the Altera DE0 Board* provides guidance on these aspects. As indicated in part *b* of Figure 19, `andi` instructions are used for bit masking to determine if a particular bit of the pattern read from `ipending` register is set.
- The implementation of the Nios II processor architecture requires an adjustment to the `ea` register, as shown in part *a* of Figure 19, when responding to hardware interrupt sources.
- The register identified as `ctl14` in part *a* of Figure 19 is actually the `ipending` register. Special registers are labelled `ctl0`, `ctl1`, ..., but the labels of `status`, `ienable`, and `ipending` are accepted by the assembler and are certainly more informative choices when writing code.
- Throughout the first half of this course, all assembly-language programs that involve interrupts will be prepared in a single source file with unconditional branch instructions at address 0x0 and 0x20 which will direct execution to the main program and the interrupt-service routine, respectively.
- **DO NOT USE** the approach with `.section` directives that is reflected in Figures 18 and 19 of *Basic Computer System for the Altera DE0 Board*; use the instructor's far simpler approach.
- **Prepare a complete interrupt-based assembly-language program** in a file `lab1_part3.s`. Using the given specifications, using the guidance from the technical documentation cited above, and using additional information provided by the instructor, a modular implementation is possible with as few as 5 instructions for the main routine, approximately 15 instructions for the initialization subroutine, and approximately 20 instructions for the interrupt-service routine.
- Use the on-line *CPUlator* tool to generate executable code and verify correct program behavior. The user interface of this tool allows the state of the simulated pushbutton to be changed between pressed and released. In response, the simulated state of LED0 should change (toggle with each release of the pushbutton). Furthermore, the code that is in the loop of the main routine should be executed whenever the pushbutton interrupt is not asserted, and the value in the register used by the main routine should be incremented.

In-Lab Part 1: Development of Loop-Based Program from Specifications

- On your network storage, create a folder `Z:\My Files\ELEC371\LAB1_PART1`.
- Use any text editor on the computer to create a file `lab1_part1.s` in the folder named above.
- Working from the specifications provided to you at the start of your laboratory session, prepare the code of a loop-based program in proper modular style.
- Use paper/pencil initially, then enter code into your `lab1_part1.s` file.
- Include a main program that prepares input parameters, calls the subroutine of interest, and make appropriate use of the return value, if any. Use appropriate code at the end of the main program to complete its execution.
- Use `.org 0x1000` followed by data directives as appropriate based on the specifications.
- Save the completed `lab1_part1.s` file.
- 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\LAB1_PART1`.
- For the project name, enter `lab1_part1`, then press *Next*.
- Select <Custom System> 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 your assembly-language source file `lab1_part1.s` in the list. Once you have added your file to the list, press *Next*.
- Make sure that the *Host connection* shows *USB-Blaster*. If it is blank, first verify that the DE0-CV board is properly connected to the computer with the USB cable and the power is turned. Then, press the *Refresh* button. Press *Next* when the Host connection is fully established.
- In the final setup window, simply press *Finish* to complete the project creation task.
- Press *Yes* when you are prompted to download the project’s system to the board. A green LED below the power button on the DE0-CV board should blink, and soon the system should be downloaded. A prompt on the screen should indicate success. Press *OK* to continue.
- Press  or select *Actions → Compile & Load* to generate and download the machine code.
- The *Disassembly* view of the Monitor Program should display your program. The code that you typed is shown in a brown color, and actual machine code in memory is shown in a green color. Each actual machine instruction has an address on the left where it is located in memory, and the encoding of the instruction is shown beside the address. Both numbers are shown in hexadecimal.
- Note the register display at the top right of the window. This display allows you to observe how the contents of the registers are modified as the program executes one instruction after another.
- Click on the *Memory* tab and observe how the Monitor program displays the contents of memory beginning at address 0 in a more direct format, without the annotations of the disassembly view.
- In the text box beside the *Go* button, type 1000 (which is a hexadecimal value) and press *Go*.
- The *Memory* view should now display the contents of the words in memory for the program data.






- Click on the *Disassembly* view again to display your program code.
- To initiate execution of your program at full speed, press  or select *Actions* → *Continue*.
- Because the computation of the program is quite modest, it will be completed almost instantly relative to human speed. Press  or select *Actions* → *Stop*, then confirm in the *Disassembly* view that the end of the program has been reached.
- Click on the *Memory* tab and view the memory contents beginning at address 0x1000 to verify that the program has functioned correctly. If the outcome is not correct, carefully review the code of the program to determine the reason for the incorrect outcome, and then use the debugging features of the Monitor Program to set a breakpoint and single-step to observe the program behavior.
- After making any changes, press  or select *Actions* → *Compile & Load* to test the code again.
- *Once correct behavior has been achieved, demonstrate the working program for credit.*

In-Lab Part 2: Hardware Execution of Parallel Input/Output Program

- On your network storage, create a folder Z:\My Files\ELEC371\LAB1_PART2.
- Place your prepared lab1_part2.s in the folder named above.
- Once the Monitor Program opens, choose *File* → *New Project*.
- In the dialog box, use *Browse...* to navigate to Z:\My Files\ELEC371\LAB1_PART2.
- For the project name, enter lab1_part2, then press *Next*.
- Select <Custom System>, set the .qsys system information file, set the .sof programming file, then press *Next*.
- Select *Assembly Program* as the program type, then press *Next*.
- Use *Add...* to include the lab1_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 proper functionality by confirming that changing the switch settings affects the hex displays.

In-Lab Part 3: Hardware Execution of Interrupt-Based Program

- On your network storage, create a folder Z:\My Files\ELEC371\LAB1_PART3.
- Place your prepared lab1_part3.s in the folder named above.
- Once the Monitor Program opens, choose *File* → *New Project*.
- In the dialog box, use *Browse...* to navigate to Z:\My Files\ELEC371\LAB1_PART3.
- For the project name, enter lab1_part3, then press *Next*.
- Select <Custom System>, set the .qsys system information file, set the .sof programming file, then press *Next*.
- Select *Assembly Program* as the program type, then press *Next*.

- Use *Add...* to include the `lab1_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.
- Verify proper functionality by confirming that pressing then releasing the first pushbutton causes the toggling of LED 0. On the DE0-CV board with the computer system that is provided by the instructor, the first pushbutton is the rightmost larger-sized button (NOT the small FPGA reset button).
- Also, press  or select *Actions* → *Stop* to observe the contents of the register that is incremented in the body of the main loop. Then, press  to allow the execution to proceed for a few seconds, then press  again to suspend execution. Examine the contents of the aforementioned register again to compare the new value with the previous value.