Queen's University
Department of Electrical and Computer Engineering
ELEC 371 Microprocessor Interfacing and Embedded Systems
Fall 2019

DRAFT Lab 2:
Basic Use of Parallel Input/Output Interfaces, and Introduction to Hardware Interrupts

Objectives

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

- continued application of assembly-language programming concepts for the Nios II, especially for development of modular code,
- 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.

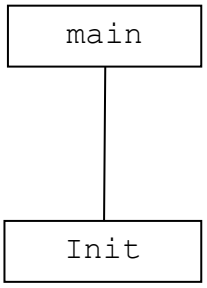
Preparation is specified in this document. The intent is for *each individual student* to make an honest effort to pursue the specified preparation without assistance from others to the extent that is possible. In this manner, a reasonable basis exists for useful discussion with others, and learning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

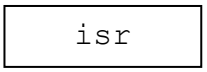
Preparation BEFORE Your Scheduled Laboratory Session

- One of the aims of this exercise is to continue emphasizing proper methodology for software design and implementation. 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. (A template source file is also provided as an aid.)

- In the past and also in this exercise, a module diagram is used to convey the organization of the software for a program, i.e., the partitioning of tasks into modules or subroutines. 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 *sub*modules.
- In general for any program specifications in pseudocode, the nature of the application, the details in the pseudocode, and related comments make 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 switches, LEDs, 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.
- It should be apparent from the features used for the code in this lab description that when using the online *CPUlator* tool, the appropriate system to select is the *DE0 system* for the Nios II.
- There is one difference in how *CPUlator* handles clearing of the pushbutton interrupt source compared to how the vendor documentation indicates how it should be cleared. The vendor documentation says writing *anything* to the edge register clears all of the edge bits (thereby clearing the interrupt source). The author of the simulator, however, has apparently interpreted the behavior differently and requires that the value written to the edge register have a 1 in the position for the bit that is set due to an edge that caused the interrupt. Write your code to satisfy the simulator requirement (a 1 in the relevant bit position) – it will also work on the actual hardware.

- This exercise provides an introduction to hardware interrupts. As 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 a 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 1 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 rightmost button0 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 `ctl4` 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 `lab2.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. Note again the comment on page 2 about the specific requirement of the *CPUlator* tool for clearing the pushbutton interrupt source.
- The *CPUlator* user interface 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.