Contents

[Educational Objective 2](#_Toc473478745)

[Technical Objective 2](#_Toc473478746)

[Deliverables 2](#_Toc473478747)

[Reference Materials 3](#_Toc473478748)

[Lab-Procedure 3](#_Toc473478749)

[Part One – VHDL implementation 3](#_Toc473478750)

[Part Two – Nios Implementation Using C code 5](#_Toc473478751)

[Part Three – Nios Implementation Using Assembly Code 5](#_Toc473478752)

[Part Four – Using Interrupts 5](#_Toc473478753)

# Educational Objective

The topics covered in the lab exercise are…

* User I/O
  + Metastability
  + Switch debounce
* Creating logic cicuits using VHDL, and using the Altera Quartus tools to implement this on an Altera Cyclone V FPGA.
* Nios GPIO peripheral – polled and interrupt driven
* C Programming
* Using C pointers to address memory mapped I/O
* Interrupts
* Assembly Language programming

# Technical Objective

In this lab, you will write code to do the following:

* Read an 8 bit number from the switches SW0 through SW7.
* Add the 8 bit number read from the switches to a 16 bit accumulator when push-button KEY1 is pressed.
* Display the 16 bit accumulator in hex on the seven segment displays HEX0 through HEX3.
* Reset the system when KEY0 is pressed. Reset should also zero the accumulator.

You will do this in 4 different ways, as detailed in parts 1, 2, 3 and 4 of the lab:

1. Using only VHDL.
2. Using a Nios processor to implement the accumulator. In this part you will use C code.
3. Using a Nios processor with main code written in assembly language.
4. Using a Nios processor with C code, but this time generating an interrupt when KEY1 is pressed.

# Deliverables

Create two directories named cpet561/lab2/quartus and cpet561/lab2/src. The src directory will contain all of your VHDL source code, and the quartus directory will contain your Quartus project and your Nios system.

For this lab, you will submit the following files in the drop-box on myCourses/.

* All source files including VHDL (\*.vhd), C code (\*.c), and assembly code (\*.s) are submitted to dropbox.
* Within Quartus you will select Project -> Archive Project to create an archive of your design. You will also submit the archive file in the dropbox.
* You must obtain a signature from the instructor or TA for each of the four main parts of the lab. Please print out the sign-off sheet at the end of the document.
* A state diagram of the VHDL debounce module, indicating how you modified it for part 2 of the lab.

# Reference Materials

* Instruction Set Reference
* Embedded Peripheral IP User Guide
* File example\_main.s
* File example\_interrupt.c

# Lab-Procedure

## Part One – VHDL implementation

Figure 1 on the next page shows the system we will implement using VHDL.

From MyCourses download the following files into your cpet561/lab2/src directory.

* accumulator.vhd
* debounce.vhd
* synchronizer.vhd

In order to complete the system, you will need to understand the requirements, and write the accum\_mod.vhd and hexDisplayDriver.vhd files based on the desired system function (see Techinical Objective section of this document).

You will follow basically the same procedure for creating your Quartus project as was outlined in the first lab. There is one difference however; you will use the cpet561/lab2/src directory to hold your VHDL source code. Therefore when you add the VHDL files to your project, you will need to navigate to this directory. If we were using revision control, this source directory would be part of your repository.

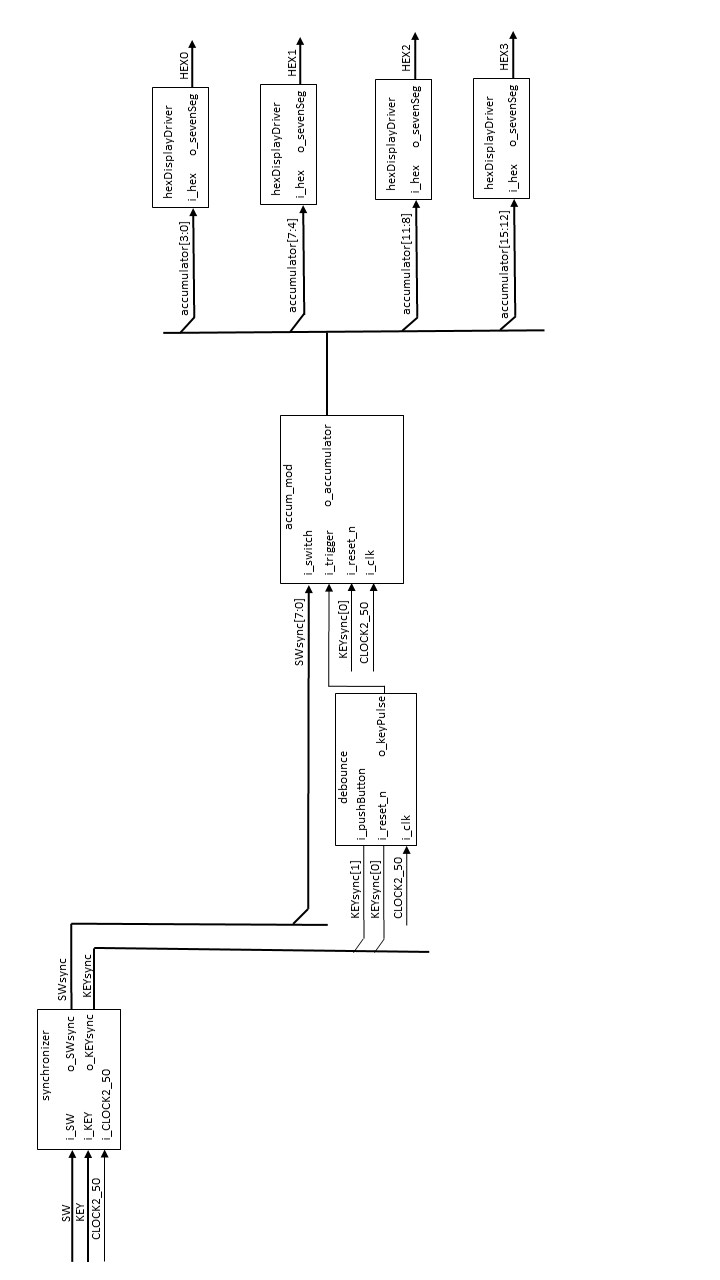


Figure 1: Block diagram of the system. Each block represents a VHDL entity, which is instantiated in the top level VHDL.

## Part Two – Nios Implementation Using C code

For this section of the lab, we will replace the accum\_mod.vhd module with a Nios processor. The Nios will run a **main()** function written in C. The code will consist of an infinite loop, that is constantly reading the **i\_trigger** input, and adding the binary input from **i\_switch** to the accumulator every time **i\_trigger** is asserted high. Additionally, the Nios processor will output the binary value of the accumulator to output port **o\_accumulator**.

For more information on how to use the GPIO, see the document ug\_embedded\_ip (i.e. the Embedded Peripheral Users Guide) and look in the PIO Core section.

There are a couple of additional things we need to deal with, before we can get this to work as intended…

1. As currently implemented, the debounce.vhd enitity’s output, **o\_keypulse**, is only one 50MHz clock cycle wide. This is a problem, because the polling of the Nios will not be sufficiently fast, and it will most likely miss this short input pulse. One way to fix this, is to modify the debounce.vhd entity to output a longer pulse. Modify the VHDL code to output a pulse approximately 75ms long every time the key is pressed.
2. The second problem, now that the pulse will be 75ms wide, is that the Nios is likely to poll it many times for each keypress. The result will be that when we press the key, we will add the switch input many times, instead of the once that we intended. You will need to write your C code to add the accumulator each time the **i\_trigger** input transitions from zero to one. This is to say that you are implementing an edge triggered functionality in C.

The nios processor will contain GPIO ports, added basically the same way as in the first lab. The other peripherals will remain exactly as in the first lab. The GPIO ports we need to add are as follows…

* A one bit input port for the **i\_trigger** input.
* An 8 bit input port for the switch inputs, **i\_switch**.
* A 16 bit output port for the accumulator output **o\_accumulator**.

## Part Three – Nios Implementation Using Assembly Code

In this section, you will write code to implement the same functionality as the C code you wrote in section two, but this time using assembly language. An example of how to implement a main function using assembly is supplied on MyCourses, it is called example\_main.s. The code in example\_main.s implements the same functionality as lab 1, using assembly language. The code you will need to write in this section will need to utilize some of the same instructions as example\_main.s, but you also need conditional branch, arithmetic, and bit-wise logic instructions.

## Part Four – Using Interrupts

For more information on how to use interrupts with the GPIO, see the document ug\_embedded\_ip.pdf (i.e. the Embedded Peripheral Users Guide) and look in the PIO Core section. Also see the reference code in example\_interrupt.vhd.

For this part, the **i\_trigger** input will cause an interrupt routine to run. Since the interrupt will be configured to happen on the rising edge of the **i\_trigger** input, you no longer need to worry about detecting the edge of the **i\_trigger** input in your code.

There are two basic ways you can use your interrupt routing…

1. Do all the work in the interrupt routine, in which case your main function could be an infinite loop that does nothing.
2. Have the interrupt set a flag in a global variable, the main function when it sees this flag set will update the accumulator, and then clear the flag.

You will use the second option for this exercise.

Follow the following procedure to add interrupt functionality to your **i\_trigger** GPIO input.

1. The PIO core provides the capability to generate an interrupt as soon as the button is pushed. In QSYS, edit the configuration of your 1-bit push button PIO core. Under the interrupt section, select Generate IRQ and choose Edge for the IRQ Type. Click Finish when done.
2. Notice the push button will generate an interrupt, but we need to connect that interrupt to an IRQ signal of the Nios II processor. Move your mouse to IRQ column within QSYS and assign IRQ0 to the interrupt going to the Nios processor as shown in Figure 2.

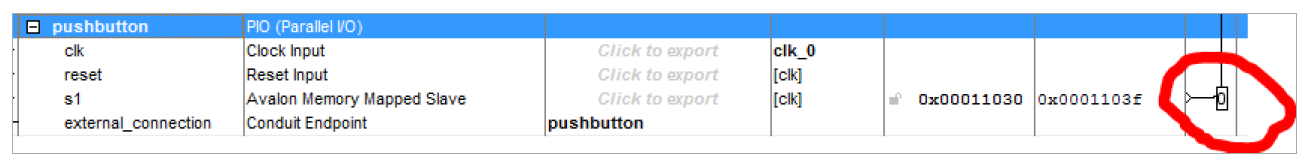


Figure 2

1. Modify your Nios II application code to register your interrupt handler. This can be done using the alt\_ic\_isr\_register() function call (see Nios II Software Developer’s Handbook). After you register the interrupt handler, enable the interrupts from the PIO core. (Why would you want to register the interrupt handler before you enable the interrupt?)