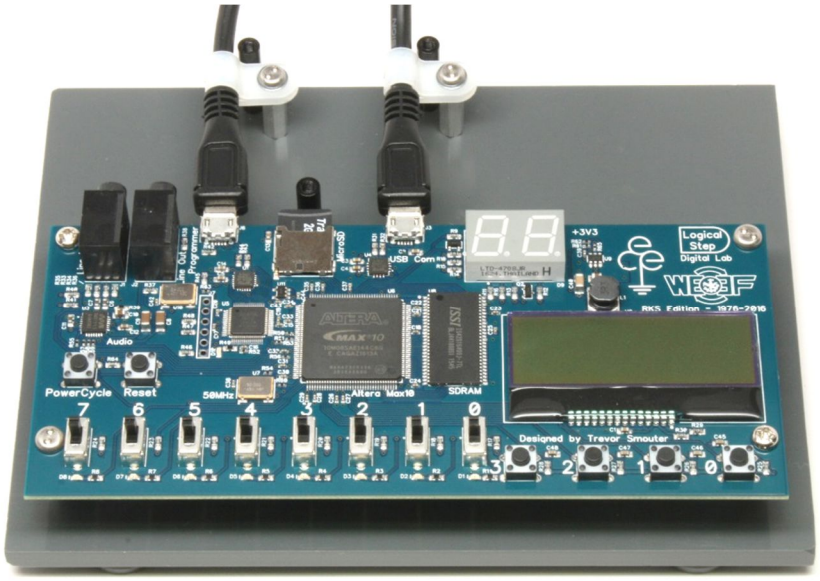
University of Waterloo
Faculty of Engineering
Department of Electrical and Computer Engineering



LogicalStep Lab Manual

for

Embedded Microprocessor Systems and Interfacing

Using this Lab Manual Effectively

Considerable care has gone into developing this lab manual. The philosophy behind its creation was to provide a document that complements in-lab instruction and support. This learning aid is not intended to stand on its own as a complete resource which would allow students to complete the labs successfully on their own – it is designed to complement a lesson plan delivered in the lab. This document also intentionally avoids incomplete coverage of details that are well documented elsewhere. If required, you will be provided with advice on where to get the information you require.

Throughout this document there are textboxes that highlight different types of information to facilitate learning. The purpose of each is described below.

Watch out!

The Watch Out textbox is used to provide cautionary information that can help you avoid common pitfalls that students experience. These pitfalls can result in road blocks that are either time consuming to fix or will possibly require the help of an expert such as the Lab Instructor or TA's to solve.

Deep Dive

The Deep Dive textboxes are used to provide background information about the topic and usually contain advanced level information. If you're interested in getting more familiar with the topic or want to understand at a deeper level, then read these textboxes. If you are a beginner struggling with the concepts introduced, you can safely ignore the information in these textboxes and move on with the core concepts.

Tips and Tricks

The Tips and Tricks textboxes provide suggestions that can help to speed up your work and generally make your life easier.

Take Away

The Take Away textbox provides a summary of important points recently discussed in the manual. It's a good idea to read the Take Away textboxes to ensure you've gotten the main points recently introduced and haven't missed any details.

Let's try out some textboxes to see them in action.

Tips and Tricks

In this document when you see underlined and blue words presented like the following, 'See: Tips on writing lab reports' it is a recommended search engine phrase to find out more details about the topic online.

Watch out!

The software tools introduced in this lab are complex and targeted at an expert level audience (as is the case with all FPGA development tools). Tool problems that impede development are common. Ensure that you use your scheduled lab time wisely - every hour spent in the lab is equivalent to three hours working without the support of the Lab Instructor and/or TAs.

Table of Contents

List of Figures

List of Tables

1 Introduction

The state of the art in FPGAs provides the hardware description language (HDL) designer the ability to define whole processor systems within the logical fabric contained within an FPGA. This powerful capability provides students with the ability to study both processors and the code that runs on them in an environment where the student can actually design and compile a new processor if required.

Here in the embedded microprocessor system lab we will be doing the labs on an Altera (Intel[1]) FPGA called the Max10. This silicon device contains logic elements that can be assembled into complex arrangements such as a microprocessor simply by defining the device in a hardware description language (HDL) and downloading the compiled result into the FPGA. Altera provides a set of tools (a tool chain) to design, program, compile and download both the hardware and software descriptions into the FPGA. Altera also provides intellectual property (IP cores) which defines commonly used hardware that designers can benefit from using. In this lab we will be assembling a bunch of the Altera IP cores and some custom developed cores into a hardware project that will define our processor system on the FPGA. The central IP core we will be using is Altera's NIOS II (pronounced nee-os) processor and we will attach to it the peripherals we require for the lab activities. We'll get into this more a little later but for now let's discuss the Altera tool chain.

1.1 Altera Toolchain

The tool chain provided by Altera can be divided into two main tools described below:

- Quartus Prime[2] - This software tool is used to develop the hardware (the logic) that will be put on the FPGA. In Quartus Prime there are numerous tools which help design, synthesize, assemble, and program the HDL into downloadable logic. An important tool you'll be using in Quartus Prime is Qsys which provides designers the ability to assemble many different IP cores such as the NIOS II processor and selected peripherals into a single instance which can be placed in your design. This means all the details such as busses and bus arbitration are handled behind the scenes and you only need worry

[1] Intel has now purchased Altera. You may find online resources that have been re-branded as Intel. The software in the lab is branded Altera, so we will refer to it as such in this manual.

[2] We will be using Quartus 15.1 which is now known as Quartus Prime. For brevity, we often refer to it simply as Quartus.

about the input and output ports to the instance compiled by Qsys. You will be using Verilog as the HDL language for this course.

- NIOS II Software Build Tools – This software tool is provided by Altera and built upon a popular software development environment called Eclipse. In this tool you will write the code for your custom processor. It also provides the ability to download and debug your code on the NIOS II processor. You will use the C language to write your software in this course.

These are the two main tools you will use in this lab and we will go into detail on how to use them in Section 2 and 3. For now, we will start with a top down view of how the tools fit together as misunderstandings here tend to cause students a lot of trouble. A quick overview should give you a better understanding of what is happening in the tools and why certain steps are required.

Figure 1 below gives a top down view of how Quartus Prime and NIOS II – Eclipse relate. In the design flow, the NIOS II processor and other IP Cores are assembled in Qsys. When the Qsys project is ready, generating the HDL synthesises the high level design and creates numerous files that serve to describe the hardware. The main files we are concerned with are the *.qip, *.sopcinfo and the *_inst.v files.

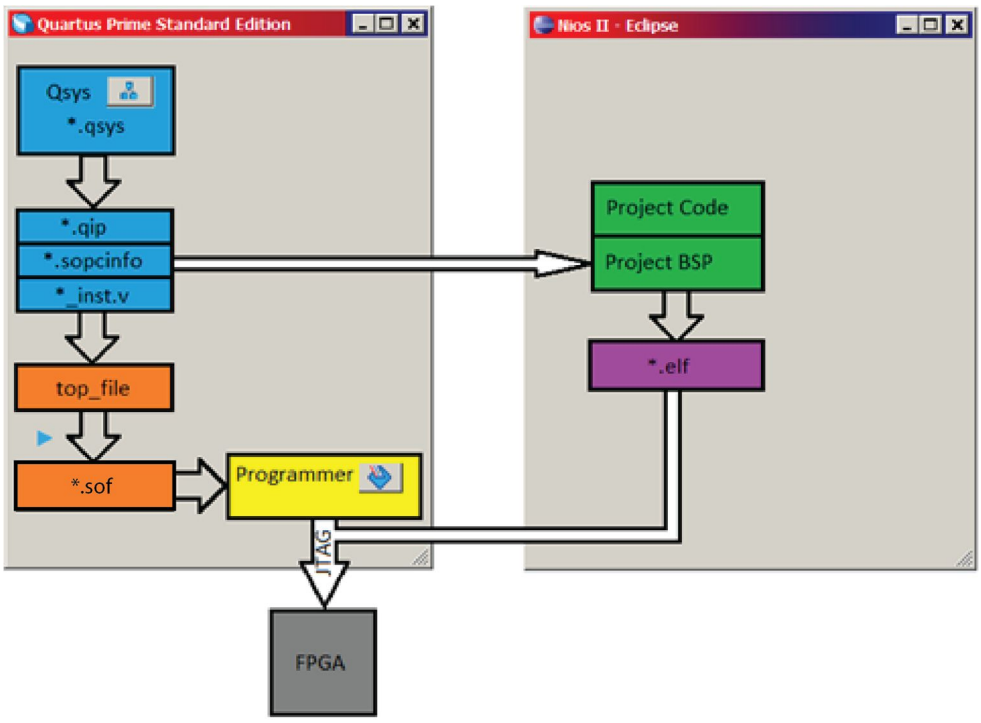


Figure 1. Quartus Prime and the Nios II Eclipse tool relationship

The *.qip file needs to be included into the Quartus project to define the processor system and other hardware. The *.sopcinfo file is generated by Qsys to describe the addresses of

peripherals and other details that the NIOS II software will require to be able to adapt the code to the hardware – since this is a soft core processor (i.e., made from logic elements in an FPGA) the hardware can be customized and the details of the processor need to be available to the software compiler when you compile your code.

The third file we are concerned with is the *_inst.v file. This file contains the template of the Verilog HDL code you will need to include in your Quartus project to instantiate (include) the processor you designed in Qsys.

In Quartus you define the hardware in what is called the top file. The top file can be thought of as similar to the main() function in C programming. The Quartus compiler starts at the top file and any logic that needs to be synthesized will be instantiated in the top file. The top file also defines the ports of the design (signals coming into or out of the project) and since it is the top file these ports are special – they are the signals that are routed to the pins of the FPGA chip. For this lab, you will be given the top file with the ports already declared but you will be required to instantiate the Qsys module in the top file using the output from Qsys (*_inst.v). When your top file is complete, you will have Quartus compile the complete hardware and if successful it will generate a *.sof file. The *.sof is the data file that gets downloaded to the FPGA via the Quartus programmer to configure the FPGA hardware. When you download the *.sof file the programmer uses a JTAG connection to load the code defining your hardware into the FPGA. It is also this connection that the NIOS II – Eclipse tool will use to program the software into the processor after your hardware is loaded, as shown in Figure 1.

When you create the NIOS II project there will be two associated projects, the project itself and the board support package (BSP). Again the BSP is generated using information from the *.sopcinfo file to know all the hardware addresses etc. Upon successful compilation of the project an *.elf file is generated which can be downloaded onto the NIOS II processor that was defined in the FPGA logic – at which point the processor will start to run.

2 Hardware Development

At this point we will start the lab by building the NIOS II processor hardware and related peripherals. There are a lot of small steps in this process, and a checklist has been provided in Appendix E. You should print this as a reference and keep it handy as you proceed.

For this lab you must use Subversion to store your work after every lab. Please refer to Appendix D now for information on how to use TortoiseSVN to setup your project folder. Once your project folder is setup, launch Altera Quartus Prime (Quartus 15.1) from the start menu and then create a new project by launching the new project wizard, 'File>New Project Wizard…' In the New Project Wizard set the working directory to the project as shown in Figure 2, the folder that you setup following the directions in Appendix D.

Watch out!

Using a working directory or project name with spaces (e.g., '\My Documents\') will create serious problems that will require the project being rebuilt from scratch to fix the issues. Be very careful to ensure there are no spaces in either parameter.

Be careful to set the project name ('LogicalStep') and the top file name ('LogicalStep_top') exactly as shown to match with the course supplied top file you will be given.

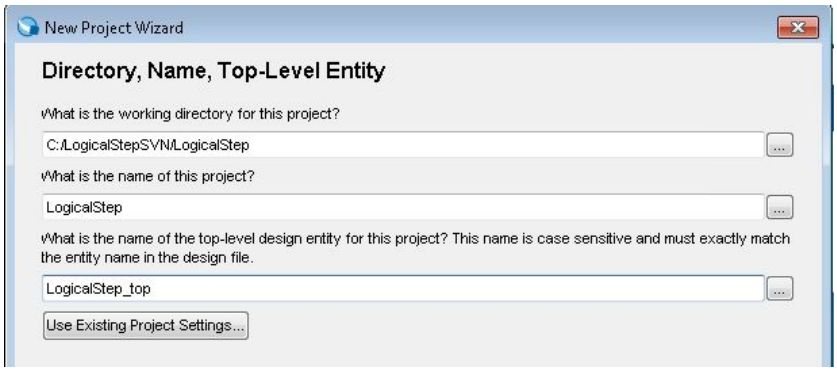


Figure 2: New Project Wizard

With the project names in place click 'Next' until you get to the 'Add Files' dialog box. At this point we need to add the course supplied project files to the project, you would have downloaded them to the project folder while following Appendix D. Use the browse button as shown in Figure 3 to add the files.

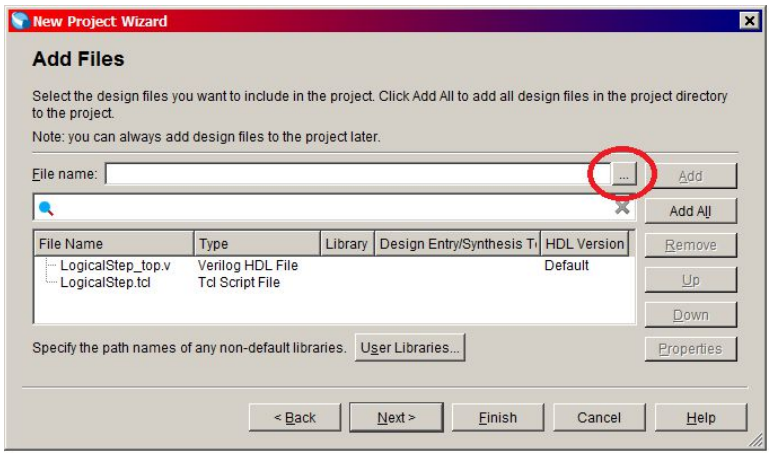


Figure 3: The 'Add Files' dialog box

Be sure to change the file filter drop down box to 'All Files (*.*)' to ensure all files are visible and add "LogicalStep_top.v" and "LogicalStep.tcl" to the project.

Click 'Next' to get to the 'Family and Device Settings' dialog box and choose the Max10 device '10M08SAE144C8G' which is the device on the LogicalStep development board. You can significantly shorten the list of devices by setting the dropdown boxes as shown in Figure 4. With the correct device selected in the 'Available devices' window click the 'Finish' button.

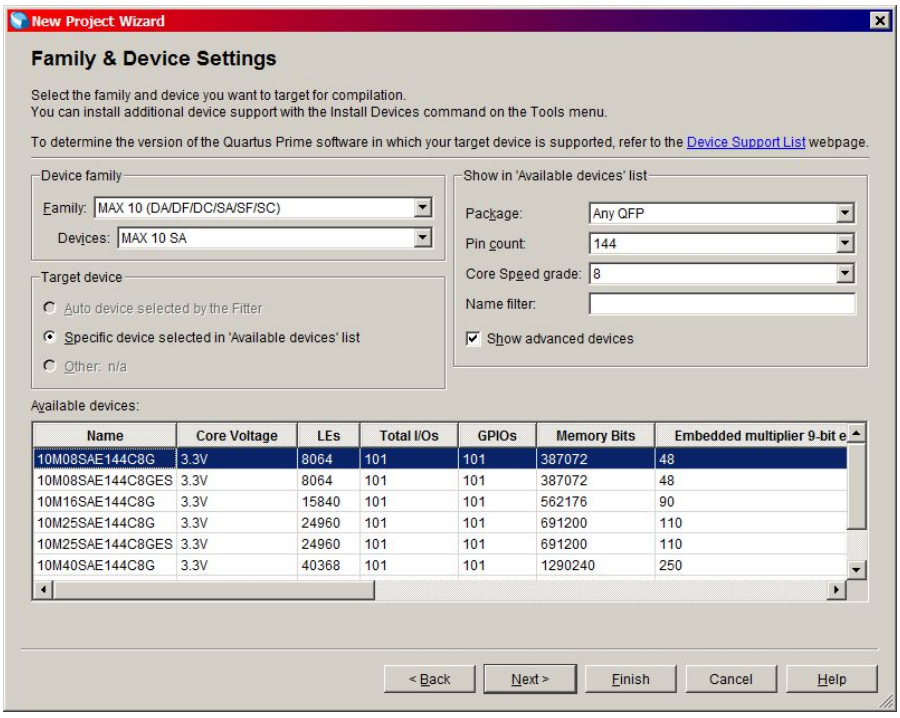


Figure 4: Family and Device Settings dialog box

Your project will now be created by Quartus. Since you provided the new project wizard with a top file name and added a matching top file to the project Quartus will automatically set the top file in the project for you.

2.1 The Top File

At this point let's examine the top file you were provided, double click on 'LogicalStep_top' in the 'Project Navigator' as shown in Figure 5.

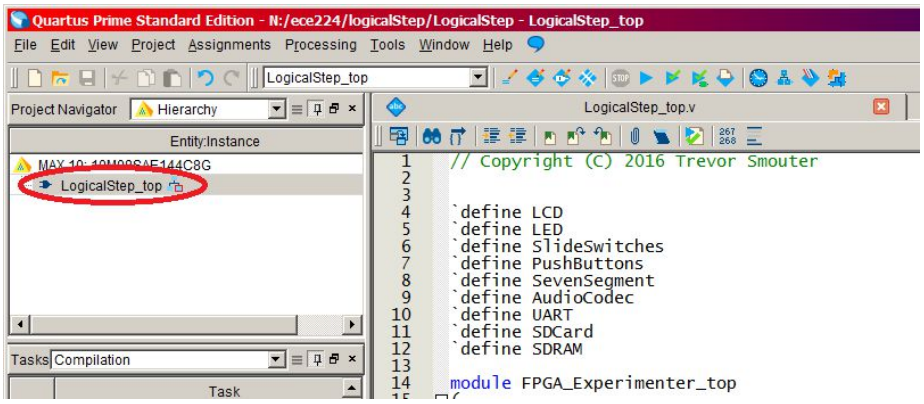


Figure 5: Selecting the top file in the Project Navigator

The top file will now be displayed (if this doesn't work then you may have made an error in naming the top file or including the top file in the project during the creation of the project). As was mentioned previously the top file is called the top file because it is the top of the hierarchical design. It can be thought of as the main() if it was a software language. Any and all logic in your design must be instantiated in the top file to be included in the compile. It is not unusual to only instantiate one piece of logic in the top file but normally the one instantiated logic will in turn instantiate much more logic itself in a hierarchical fashion. In our case the top file will instantiate the generated output from Qsys which in turn instantiates a lot of different IP cores that were included into the Qsys design which we will see later. The hierarchy for our design can be seen in Figure 6.

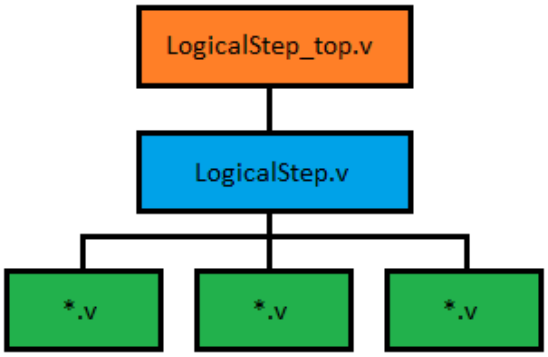


Figure 6: Hierarchy for the LogicalStep design

Upon closer inspection of the top file we see that it starts with a number of define statements. These defines allow you to leave out portions of the design if you are not using the hardware. Note that these defines indicate different hardware sections on the LogicalStep board that connect to the FPGA.

Below the defines is the module declaration of 'LogicalStep_top'. It is important that this name match the top file. Inside here all the top level ports are declared and you will notice that the only ports not within a define statement is the clock and reset at the top, every HDL design needs a clock and a reset. The rest of the ports are grouped together within their related define statement to allow easy removal of a feature should you desire to remove it. It is interesting to note that if you comment out a define such as the SDRAM, all of the pins related to the SDRAM will be removed from the design but it doesn't stop there. During compilation Quartus will notice the missing pins and will subsequently remove everything in the design related to the SDRAM all the way back to the NIOS processor if it is included in the design. This is because Quartus can see that nothing can influence the SDRAM if the pins are missing and it is useless to have any associated hardware in the design therefore it will optimize it away.

At the bottom of the file is a space where you will instantiate your Qsys project once it is complete. We will discuss this in more detail later.

2.2 Assigning Pins

The next step to build the project is to assign all of the IO ports to pins. Normally with a fresh project you would have to grab the board schematic and manually enter the pin numbers for each of the ports in the top file using the pin planner in Quartus. Instead we have provided a .tcl script (pronounced "tickle") to automatically assign the port pins for you. Since you already included the file in the project you can simply go to 'Tools>Tcl Scripts…' select the LogicalStep.tcl

file and click 'Run'. Execution of this tcl script will automatically set the pin assignments in your project.

2.3 Qsys

It's now time to assemble the components of the NIOS system and this is done in Qsys. Before you open Qsys and start building the system you need to copy the IP folder (that was supplied in the lab materials on Desire2Learn) into your LogicalStep project folder – this should have already been completed while following Appendix D.

Watch out!

If you don't have the IP folder in your project folder then the custom IP cores that were developed for this lab will not be available in the Qsys 'IP Catalog' to add into your project. Ask the lab staff for help in resolving this problem.

To open the Qsys tool in Quartus select 'Tools>Qsys'. If you're familiar with the old Quartus SOPC Builder you'll notice that Qsys is similar but has more features and is more concerned with allowing easy plug and play of IP cores other than just processor peripherals etc. The whole FPGA project can be built in Qsys out of IP cores if desired and that is what we will do for the embedded microprocessor system labs.

With Qsys open you'll start from scratch building up the project hardware and Qsys starts with an empty project except for a clock source core. There are a few things you may have to change on every core that you add to the project so we'll go over each step first with the 'Clock Source' core. You may be required to change some of the settings for each IP core from the default settings provided. In this part of the manual there is a section for each core that needs to be added. In each cores section there is description of the internal settings (the 'Parameters') that need to be changed as well as some of the nomenclature changes required for the project to work with the software. Now is a good time to save the Qsys file, use the name 'LogicalStep' for the Qsys project (i.e., C:\LogicalStepSVN\LogicalStep\LogicalStep.qsys).

Tips and Tricks

Get the name right as the name of the core affects how the software accesses the core. If the name isn't right you may run into problems during software development.

We'll go over the settings for the clock source first to get an idea of what is required for the rest of the cores. In the 'System Contents' tab of Qsys, in the 'Description' column, double click

'Clock Source' to open the IP core 'Parameters' as shown in Figure 7. Ensure that the 'Clock Frequency' is 50 MHz and close the tab.

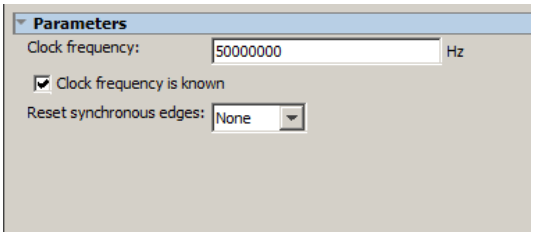


Figure 7: Clock Source Core Parameter Settings

Next you want to ensure the core's name is correct and add any required exports as shown in Table 1. Note the table indicates the type of IP core (here it is 'Clock Source'), the name that should be used to identify the core in Qsys (here it is 'clk_50') and the exports if any that should be added.

Table 1: Clock Source Core Name and Exports

IP Block	Name	Export
Clock Source	clk_50	clk_in = "clk_50" clk_in_reset = "reset"

So, from the table above change the name of the clock source to 'clk_50' as shown in Figure 8 by right clicking it and selecting rename. Then double click the export column for 'clk_in' to add a top-level export for the clock input and name it as shown in the table above.

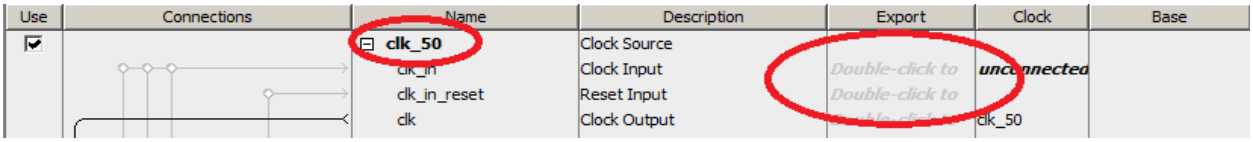


Figure 8: Naming an IP Core in Qsys

Which should look like Figure 9 when it is finished,

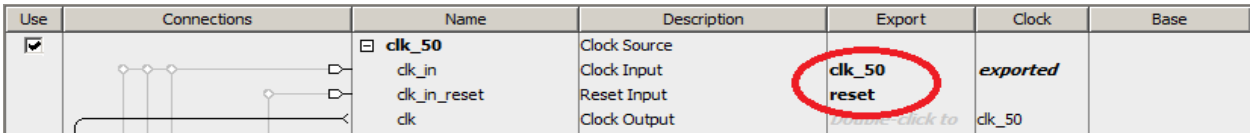


Figure 9: Export Settings of a Core in Qsys

Deep Dive

When you finish assembling your Qsys project and generate it, any exports included will become ports that will connect in the top file. These ports allow you to connect signals into and out of your Qsys hardware in the Top file. In the case of our project the top file is mainly used to connect pins on the FPGA package to signals exported from the Qsys project.

The next section directs you on how to configure each IP block as you add it to the Qsys project. Once you configure each block click 'Finish' and the IP will be added to the project. Ensure the name and the exports are set correctly as provided in the IP Block table for each IP core shown below.

Tips and Tricks

There are two ways to find the cores you need to add in the Qsys 'IP Catalog'. The tree path has been listed under each core title, or you can use the search function with the core name.

2.3.1 Adding the 'Avalon ALTPLL' IP Core

(Library/Basic Functions/Clocks; PLLs and Resets/PLL)

Find the IP core in the IP catalog and double click it to add it to the project.

Set the parameters under 'Parameter Settings'>'General/Modes' and as shown in Figure 10

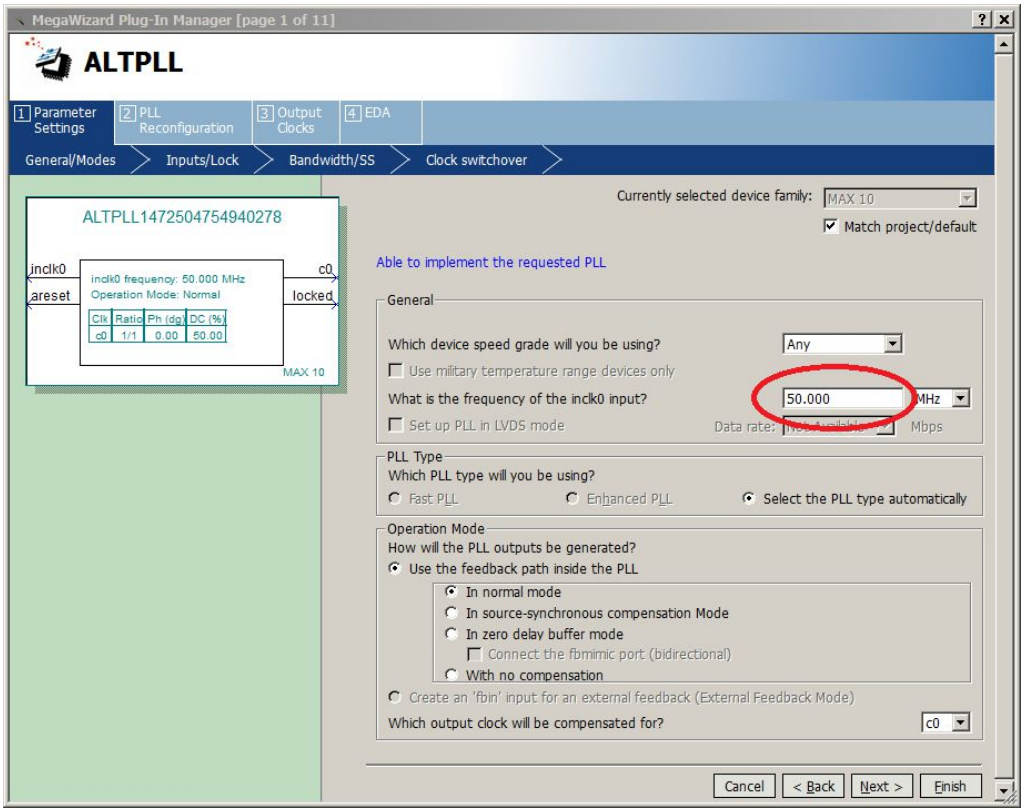


Figure 10: AltPll Core Parameter Settings

Set the parameters under 'Output Clocks'>'clk c0' and as shown in Figure 11. A bug has been observed in Quartus where the -3 ns setting on the clock shift may not immediately appear in the "Actual Settings" column. If this happens, try closing and re-opening the core.

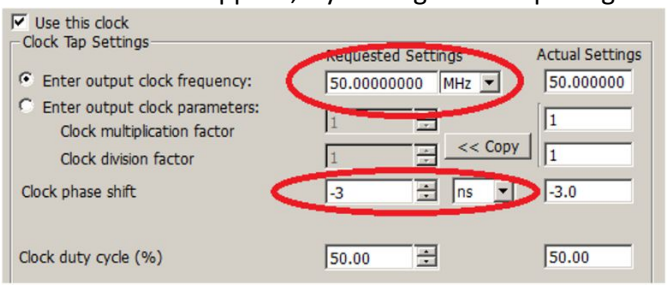


Figure 11: AltPll C0 Parameter Settings

Set the parameters under 'Output Clocks'>'clk c1' and as shown in Figure 12.

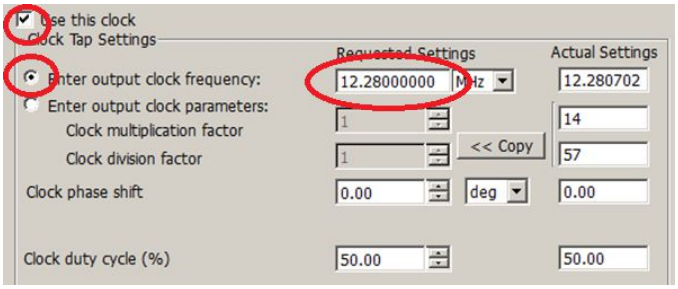


Figure 12: AltPll C1 Parameter Settings

Set the parameters under 'Output Clocks'>'clk c2' and as shown in Figure 13.

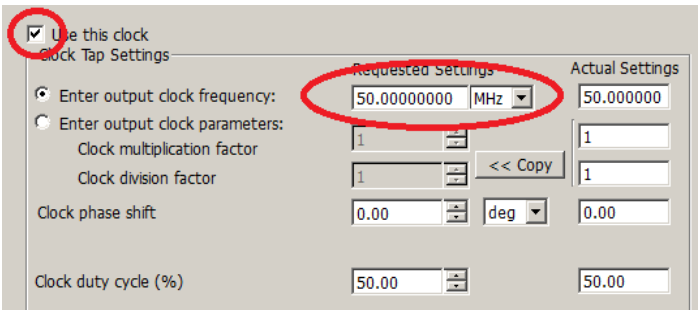


Figure 13: AltPll C2 Parameter Settings

Under the 'Inputs/Lock' tab uncheck the 'Create locked output' as shown in Figure 14.

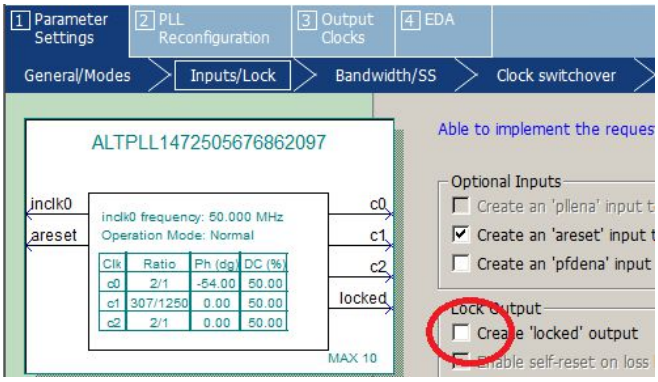


Figure 14: Removing the AltPll locked output

Tips and Tricks

As you add cores you'll noticed that Qsys will start reporting errors about base addresses – it is safe to ignore this as the errors will be corrected at a later step.

Then click 'Finish' and set the name and exports as shown in Table 2. C2 is not exported as we will use it internally. Likewise, the clock input is unconnected as we will connect it to the clock_50 source we added previously within the module.

Table 2: AltPll Core Name and Exports

IP Block	Name	Export
Avalon ALTPLL	*Use default name*	c0 = "sdram_clk" c1 = "audio_mclk"

2.3.2 Adding the 'NIOS II Processor' Module

(Library/Processor and Peripherals/Embedded Processors)

Find the IP core in the IP catalog and double click it to add it to the project.

At this point the only setting you need to change is to select the 'f' variant of the processor.

Table 3: NIOS II Core Name and Exports

IP Block	Name	Export
NIOS II Processor	*Use default name*	*N/A*

2.3.3 Adding the 'SDRAM Controller' IP Core

(Library/Memory Interfaces and Controllers/SDRAM)

Find the IP block in the IP catalog and double click it to add it to the project.

Set the parameters as shown in Figure 15.

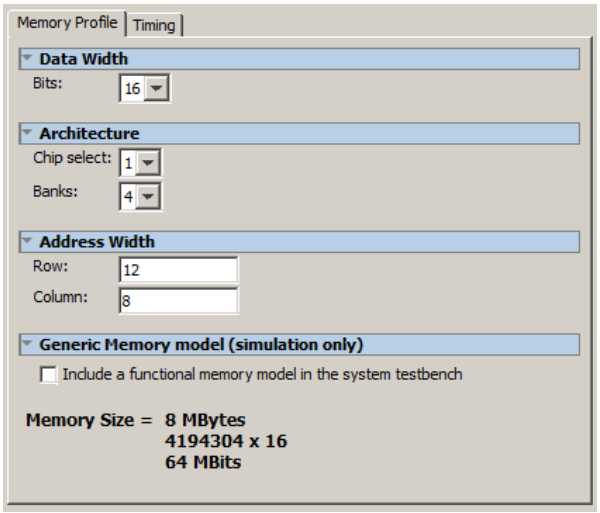


Figure 15: SDRAM Controller Core Parameter Settings

Then click 'Finish' and set the name and exports as shown in Table 4.

Table 4: SDRAM Controller Core Name and Exports

IP Block	Name	Export
SDRAM Controller	sdram_0	wire = "sdram_0"

2.3.4 Adding the 'System ID Peripheral' IP Core

(Library/Basic Functions/Simulation; Debug and Verification/Debug and Performance)

Find the IP core and add it to the project. Just use the default settings for this IP Core.

Table 5: System ID Peripheral Core Name and Exports

IP Block	Name	Export
System ID Peripheral	*Use default name*	*N/A*

2.3.5 Adding the 'JTAG UART' IP Core

(Library/Interface Protocols/Serial)

Find the IP core and add it to the project. Just use the default settings for this IP Core.

Table 6: JTAG UART Core Name and Exports

IP Block	Name	Export
JTAG UART	*Use default name*	*N/A*

2.3.6 Making Qsys Connections

With these first few IP cores now in the project you can wire them up using the 'Connections' column as shown in Figure 17. First take a moment to compare the connections column in your Qsys project to the connections column in the project shown in Figure 17. Note how the different cores have been wired together by **clicking the appropriate connection dots as shown in Figure 16**. You can make the connections according to the three figures in this section that depict the connections. Alternatively you can use the 'Deep Dive' textbox below to understand why certain connections need to be made and then try do it yourself. When done, ensure your connections are correct.

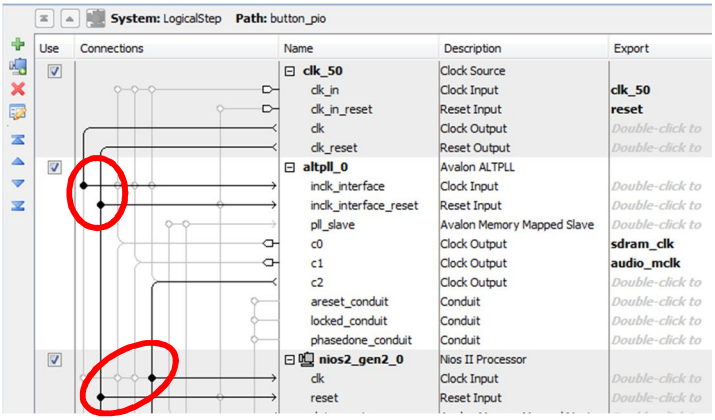


Figure 16: Example of the Connection Dots Used for Wiring

Deep Dive

Clocks – Note how the 50 MHz clock comes from the top file through the 'Clock Source' IP core and passes into the 'altpll_0' which is a PLL core that provides different clocks to the processor design. The c0 clock goes to the 'sdram_clk' export and provides a phase shifted clock output to the top file which will be connected to the SDRAM memory on the board. The 'audio_mclk' export is set to support the bitrate required to provide the 44.1 kHz sampling frequency to the audio codec on the board. Finally, the c2 clock you will notice provides a 50 Mhz clock to every other core in the processor. Note that every cores clock must be connected to this.

Reset – Note how the 'clk_reset' output of the 'clk_50' Clock Source core also connects to every core in the system. This is the reset signal which is exported to the top file and will be connected to the reset button on the LogicalStep board allowing you to do a full reset of each core when you press the reset button.

There are three other signals you will be connecting to complete your system which interface with the NIOS processor.

Data_master – This is the Avalon bus connection that allows the different IP Blocks to communicate with the NIOS processor. Any block where data is sent to or from the NIOS, or is controlled by the NIOS will have a port that can connect to the NIOS' data_master.

Instruction_master – This is the NIOS connection where the NIOS accesses its program memory. Note that this is connected to the SDRAM in our project because when you download code to the board from the NIOS build tools you download it to the SDRAM. Since the NIOS also uses the SDRAM as memory for you project you'll notice that both the instruction_master and data_master are connected to the SDRAM.

IRQ – Any cores that are set to generate interrupts must be connected to the IRQ input of the NIOS processor otherwise interrupts will not work for that core.

Tips and Tricks

Don't worry about the 'Base' and 'End' addresses shown in the Qsys window below. They will automatically be added in a later step.

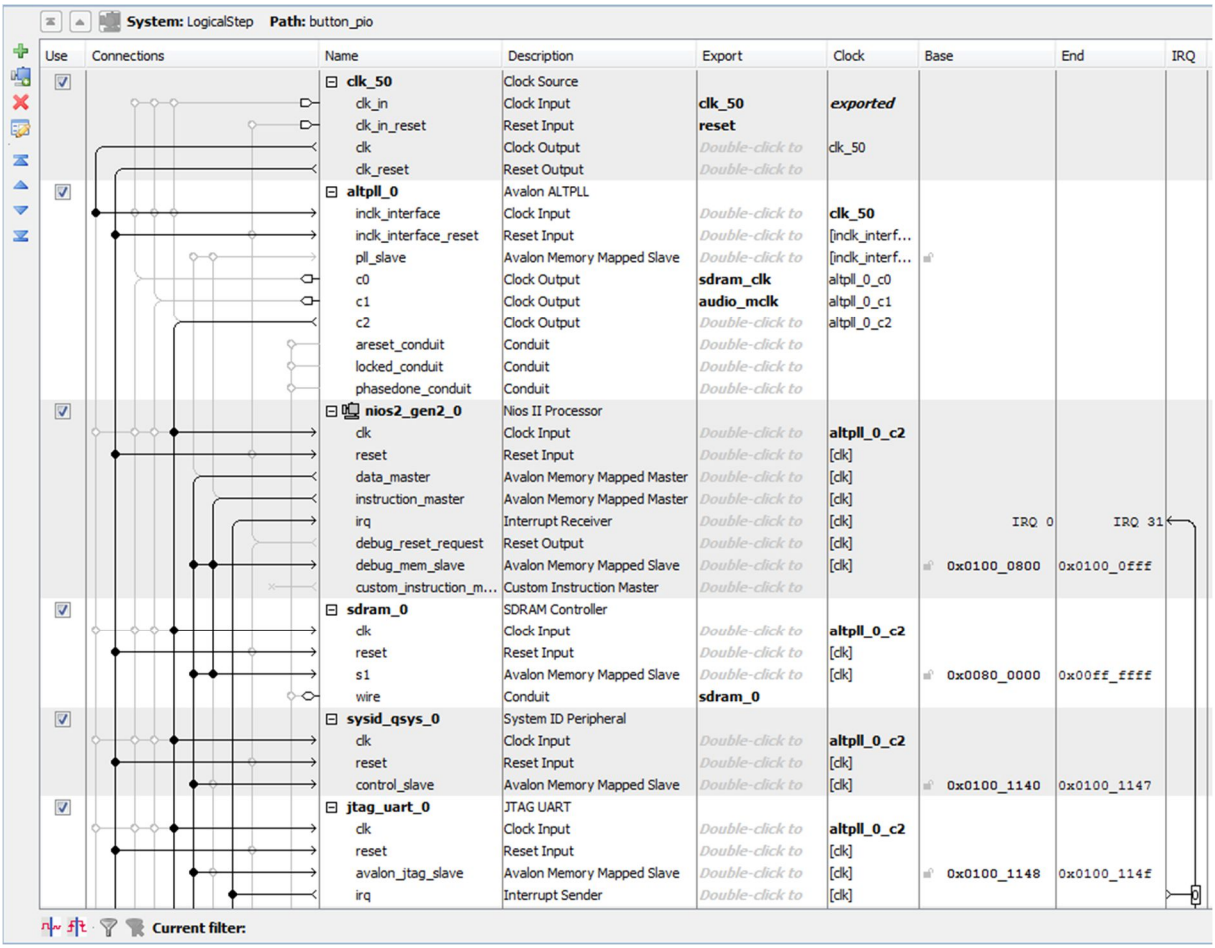


Figure 17: Connecting Cores in Qsys

Now we'll continue to add more cores to the system.

2.3.7 Adding the 'PIO' IP Core for LEDs

(Library/Processors and Peripherals/Peripherals)

Find the IP core and add it to the project. Use the settings as shown in Figure 18.

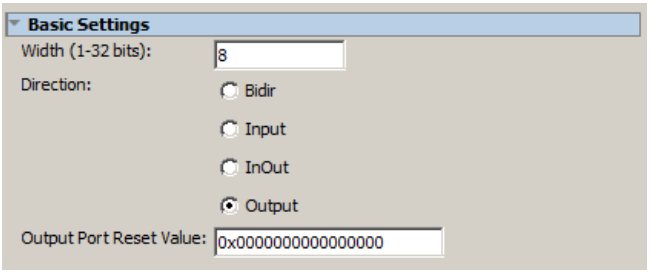


Figure 18: LED PIO Core Parameter Settings

Then click 'Finish' and set the name and exports as shown in Table 7.

Table 7: LED PIO Core Name and Exports

IP Block	Name	Export
PIO (Parallel I/O)	led_pio	external_connection = "led_pio"

2.3.8 Adding the 'PIO' IP Core for push buttons

(Library/Processors and Peripherals/Peripherals)

Find the IP core in the IP catalog and double click it to add it to the project.

Set the parameters as shown in Figure 19.

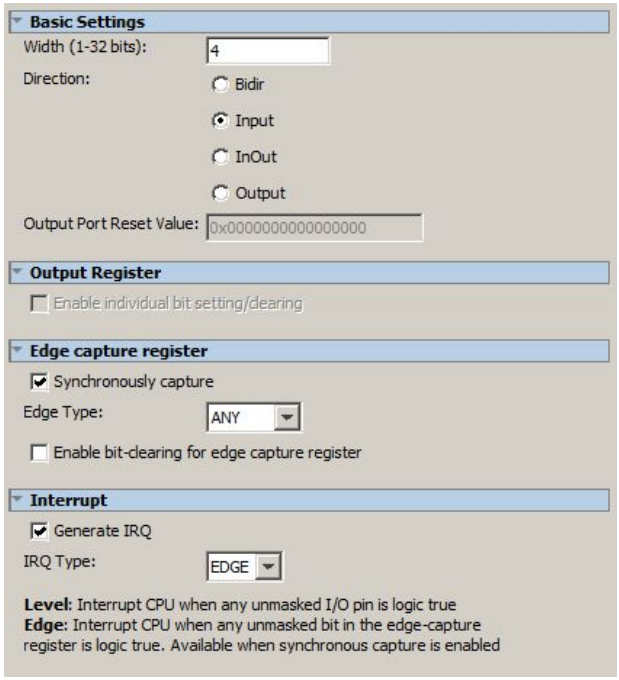


Figure 19: Push Button PIO Core Parameter Settings

Then click 'Finish' and set the name and exports as shown in Table 8.

Table 8: Push Button PIO Core Name and Exports

IP Block	Name	Export
PIO (Parallel I/O)	button_pio	external_connection = "button_pio"

2.3.9 Adding the 'PIO' IP Core for switches

(Library/Processors and Peripherals/Peripherals)

Find the IP core in the IP catalog and double click it to add it to the project.

Set the parameters as shown in Figure 20.

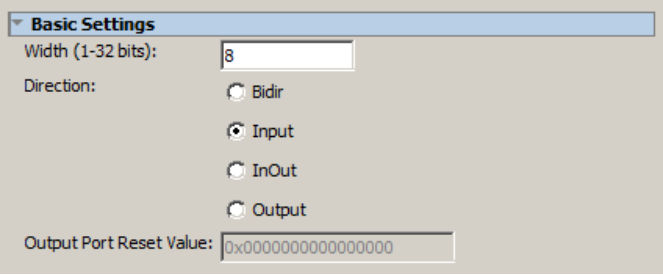


Figure 20: Switch PIO Core Parameter Settings

Then click 'Finish' and set the name and exports as shown in Table 9.

Table 9: Switch PIO Core Name and Exports

IP Block	Name	Export
PIO (Parallel I/O)	switch_pio	external_connection = "switch_pio"

2.3.10 Adding the 'Altera Avalon LCD 16207' IP Core

(Library/Processors and Peripherals/Peripherals)

Find the IP core in the IP catalog and double click it to add it to the project. Just use the default parameter settings for this core but update name and exports as shown in Table 10.

Table 10: Altera Avalon LCD 16207 Core Name and Exports

IP Block	Name	Export
Altera Avalon LCD 16207	lcd_display	external = "lcd_display"

2.3.11 Adding the 'Audio and Video Config' IP Core

(Library/University Program/Audio & Video)

Find the IP core in the IP catalog and double click it to add it to the project.

Set the parameters as shown in Figure 21 and the name and exports as shown in Table 11.

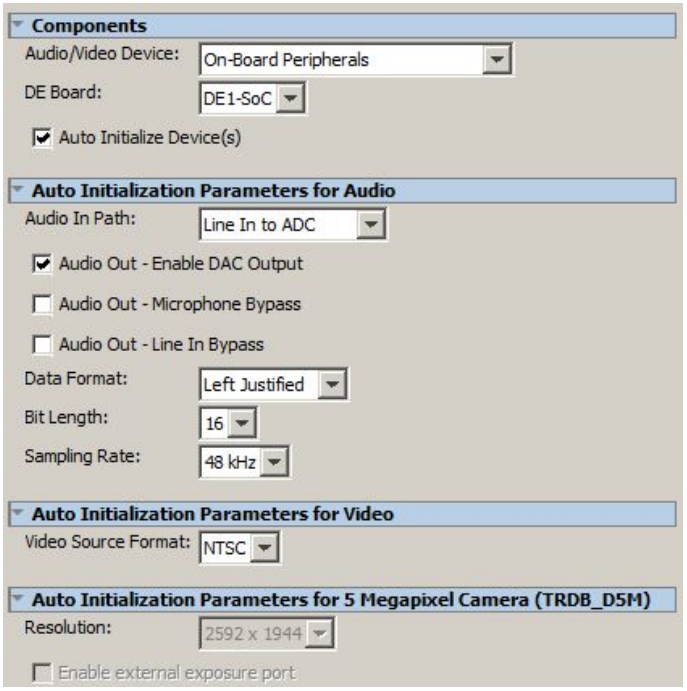


Figure 21: Audio and Video Config Core Parameter Settings

Table 11: Audio and Video Config Core Name and Exports

IP Block	Name	Export
Audio and Video Config	audio_i2c_config	external_interface = "audio_i2c"

2.3.12 Adding the 'Audio' IP Core

(Library/University Program/Audio & Video)

Find the IP core in the IP catalog and double click it to add it to the project. Set the name and exports as shown in Table 12. Use the default parameters for all other settings.

Table 12: Audio Core Name and Exports

IP Block	Name	Export
Audio	Audio (**ensure this name is capitalized to avoid problems later**)	external_interface = "audio_out"

2.3.13 Adding the 'UART (RS-232 Serial Port)' IP Core

(Library/Interface Protocols/Serial)

Find the IP block in the IP catalog and double click it to add it to the project. Set the name and exports as shown in Table 13. Use the default parameters for all other settings.

Table 13: UART Core Name and Exports

IP Block	Name	Export
UART (RS-232 Serial Port)	uart	external_connection = "uart"

At this point you can make the connections for the newly inserted cores as shown in Figure 22.

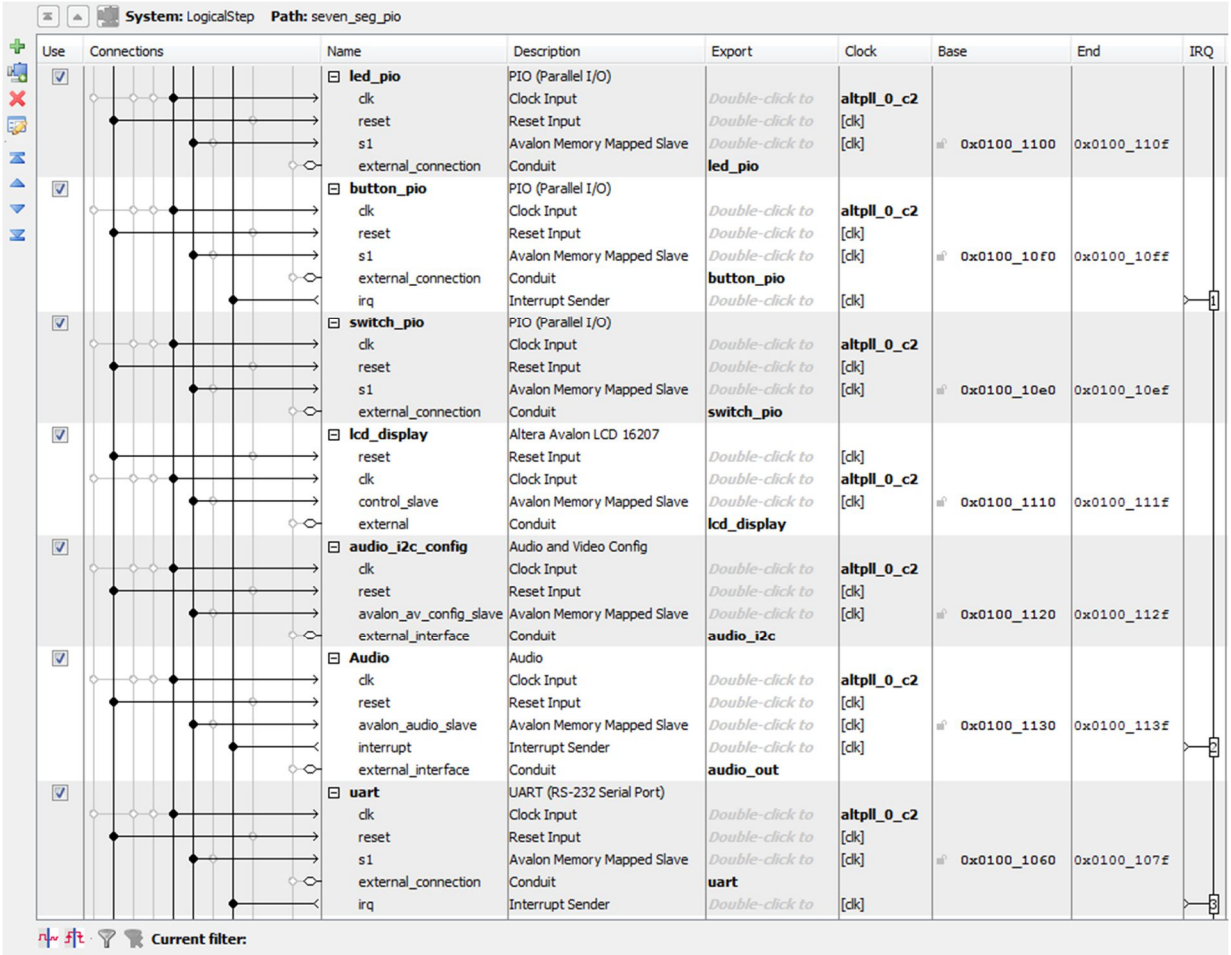


Figure 22: More Qsys Core Connections

2.3.14 Adding the 'Interval Timer' IP Core

(Library/Processors and Peripherals/Peripherals)

Find the IP core in the IP catalog and double click it to add it to the project. Set the name and exports as shown in Table 14.

Table 14: System Timer Core Name and Exports

IP Block	Name	Export
Interval Timer	system_timer	N/A

2.3.15 Adding another 'Interval Timer' IP Core

(Library/Processors and Peripherals/Peripherals)

Find the IP core in the IP catalog and double click it to add it to the project. Just use the default settings for this core.

2.3.16 Adding the 'SPI Master (3 Wire Serial)' IP Core

(Project)

Find the IP core in the IP catalog and double click it to add it to the project. There are no parameter settings required for this core but set the name and exports as shown in Table 15. If you did not place the IP files according to Appendix D, you will not see the Project category. Ask a teaching team member for help in resolving this issue.

Table 15: SPI Master Core Name and Exports

IP Block	Name	Export
SPI Master (3 wire serial)	spi_master	external = "spi_master"

2.3.17 Adding the 'Dual 7 Segment' IP Core

(Project)

Find the IP core in the IP catalog and double click it to add it to the project. There are no parameter settings required for this core but set the name and exports as shown in Table 16.

Table 16: Dual 7 Segment Core Name and Exports

IP Block	Name	Export
Dual 7 Segment	seven_seg_pio	dual_7_segment = "segment_drive"

2.3.18 Adding the 'EGM' IP Core

(Project)

Find the IP core in the IP catalog and double click it to add it to the project. There are no settings required for this module but set the name and exports as shown in Table 17.

Table 17: EGM Core Name and Exports

IP Block	Name	Export
EGM	egm	interface = "egm_interface"

2.3.19 Adding the 'PIO' IP Core for stimulus_in

(Library/Processors and Peripherals/Peripherals)

Find the IP core in the IP catalog and double click it to add it to the project. Set the parameters as shown in

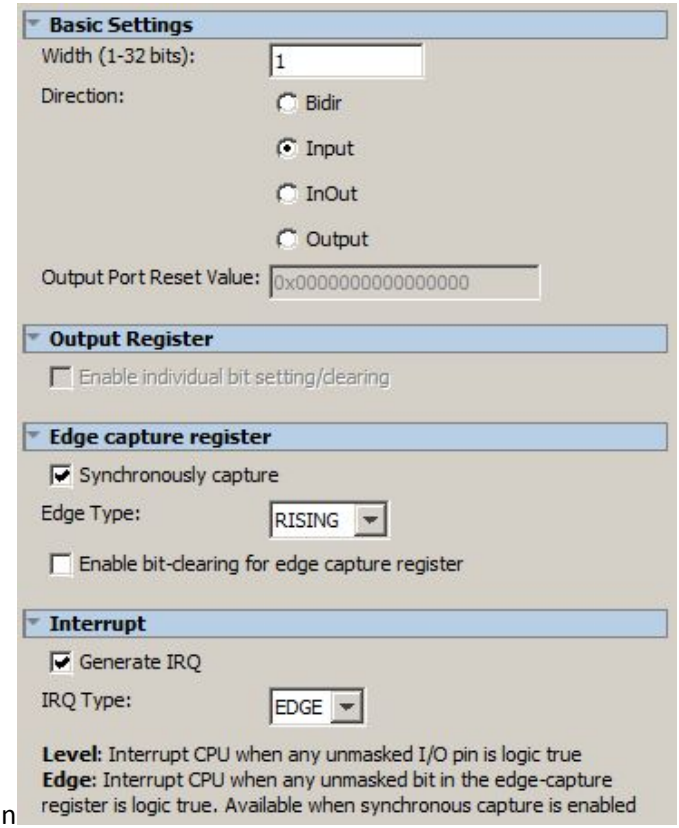


Figure 23 and then name and exports as shown in

Table 18.

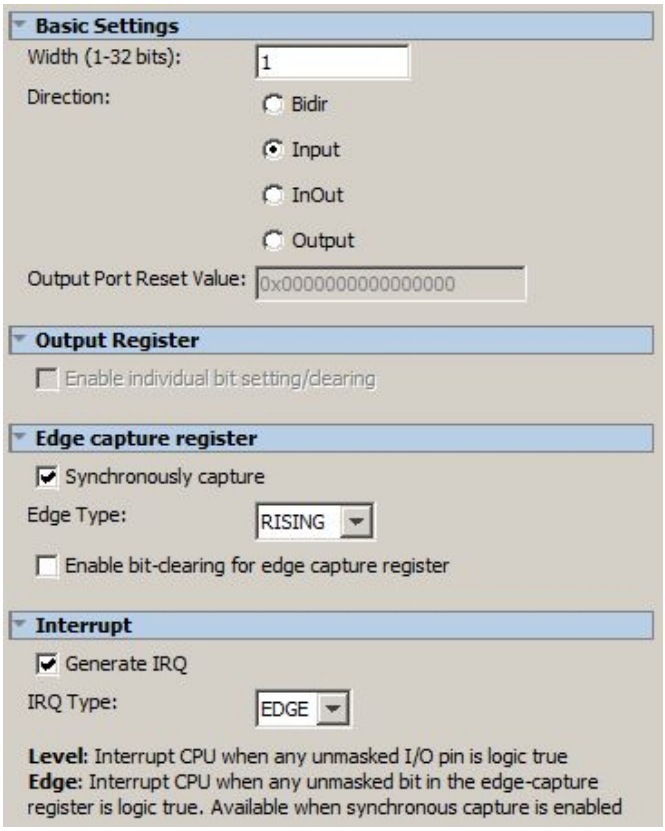


Figure 23: Stimulus In PIO Core Parameter Settings

Table 18: Stimulus In PIO Core Name and Exports

IP Block	Name	Export
PIO (Parallel I/O)	stimulus_in	external_connection = “stimulus_in”

2.3.20 Adding the ‘PIO’ IP Core for response_out

(Library/Processors and Peripherals/Peripherals)

Find the IP core in the IP catalog and double click it to add it to the project. Set the parameters as shown in Figure 24 and the name and exports as shown in Table 19.

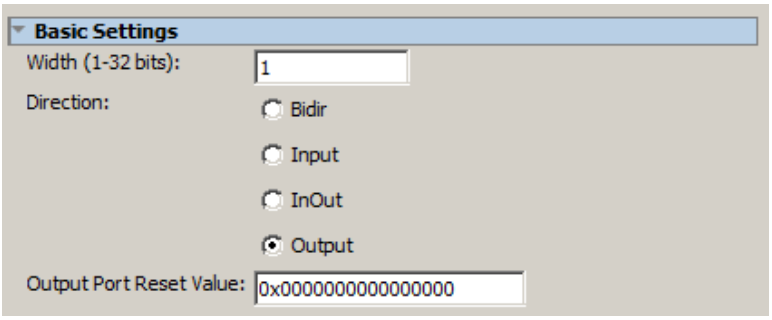


Figure 24: Response Out PIO Core Parameter Settings

Table 19: Response Out Core Name and Exports

IP Block	Name	Export
PIO (Parallel I/O)	response_out	external_connection = “response_out”

At this point you can make the connections for the rest of the core as shown in Figure 25.

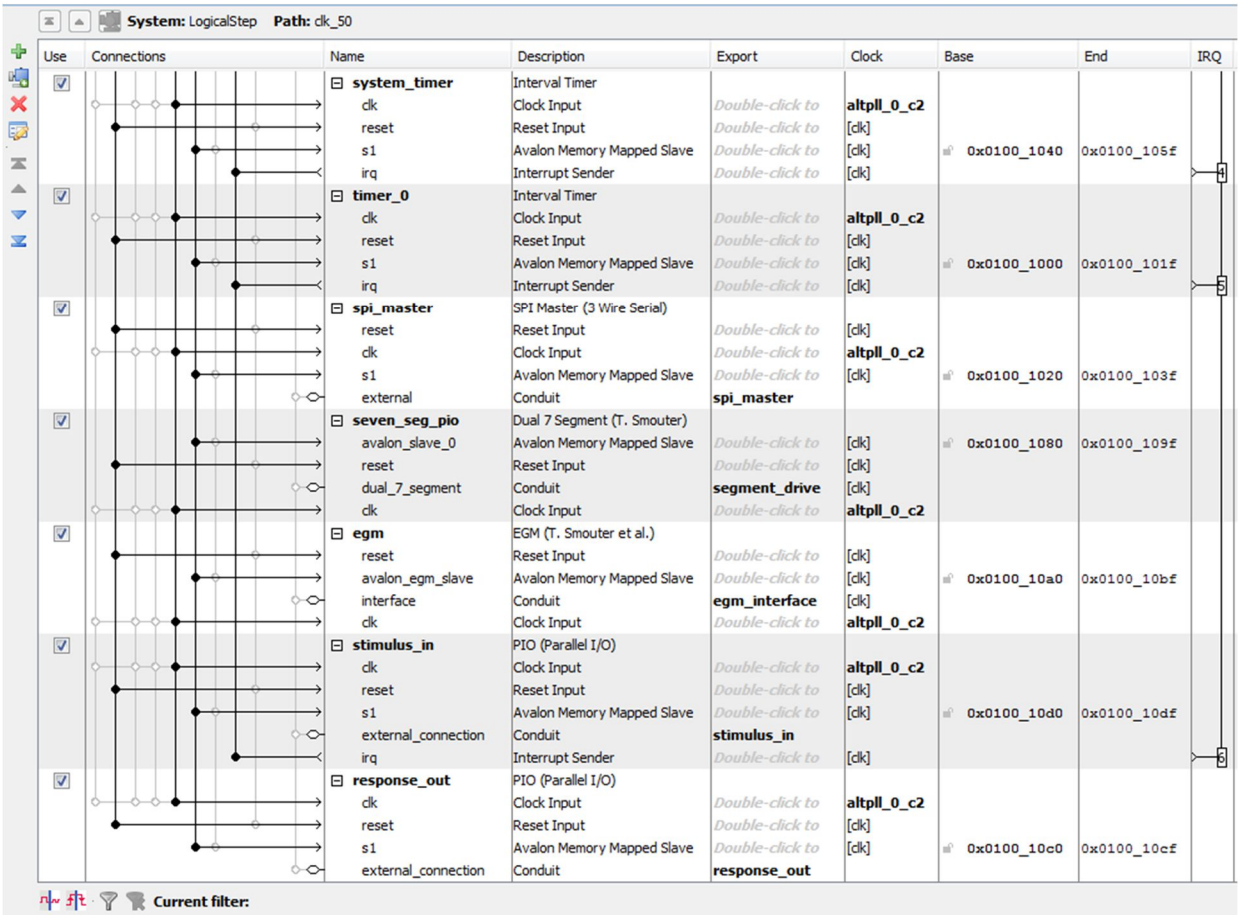


Figure 25: Remaining Qsys Connections

Now you should set the base addresses of each core so that each core has a unique address that doesn’t overlap with other cores (see Tips and Tricks below).

Tips and Tricks

Qsys has the ability to set all the base addresses automatically without errors which saves considerable time for the designer. Just use, ‘System>Assign Base Addresses’ to assign the base addresses so that they are not overlapping.

With all the cores now added to the Qsys project you can set the memory vectors in the NIOS processor by double clicking on the processor core (added in section 0), select the 'Vectors' tab and adjust the settings as shown in Figure 26.

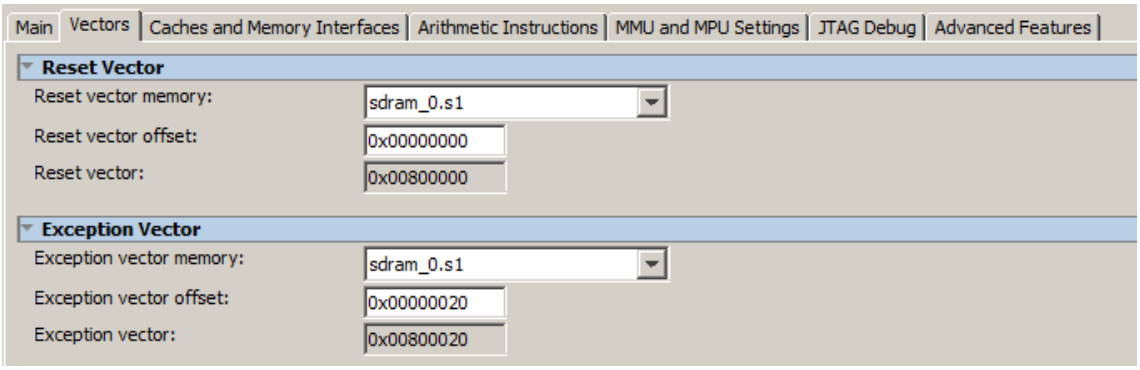


Figure 26: NIOS Memory Vector Parameters

Hopefully everything is now ready to go. Click the 'Generate HDL…' button in the bottom right hand corner of the Qsys window and after the 'Save' dialog is finished and closed the system will show the Generation Window in Figure 27 – then press 'Generate'.

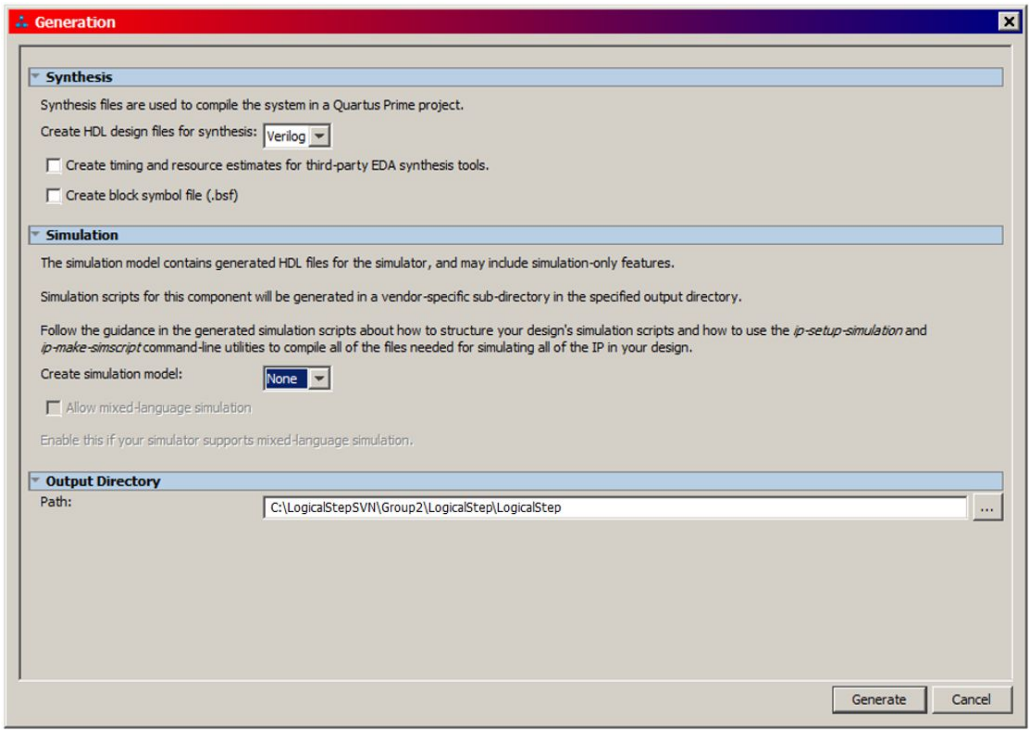


Figure 27: Qsys Project Generation

2.4 Adding your Qsys project to the top file

At this point you want to add the LogicalStep.qip file to your Quartus project. The .qip file was generated during the Qsys compile and it tells Quartus certain information about what Qsys did and where to locate the required files. In the top left of the Quartus project click the drop down that says 'Hierarchy' and change it to 'Files'. In the 'Files' pane right click on 'Files' to add files to the project. Click the '…' browse button beside the 'File name' field and navigate to the .qip file 'LogicalStep>Synthesis>LogicalStep.qip' and click 'Open'. Then click 'Add' and 'OK' to add the file to the project.

2.4.1 Editing the Top File to Instantiate the Qsys Project

In the 'Project Navigator' right click on the LogicalStep_top.v file and select 'Set as Top Level Entity' which sets the provided top file to be the top file according to Quartus. Click the dropdown in the 'Project Navigator' and set it to 'Hierarchy' then double click 'LogicalStep_top' to open the top file.

Scroll down to the bottom of the top file to the place where it says "Place Qsys instance below here". Now you need to get the automatically generated Qsys instance template which can be found under 'File>Open>LogicalStep>LogicalStep_inst.v'. Copy the Verilog instantiation text that is in this file and paste it into the top file between the comments.

Now you need to assign the top level ports (which are in fact the pins on the FPGA that connect to the different interfaces on the LogicStep board) to the Qsys module you created. This is done by replacing the corner bracket text in the instantiation code to the proper input or output ports in the top file. For instance the following line:

```
.clk_50_clk          (<connected-to-clk_50_clk>),
```

Needs to be changed to:

```
.clk_50_clk          (clkin_50),
```

Where clkin_50 is an input signal that can be found close to the top of the top file. Go ahead and assign all the inputs and outputs in this manner trying your best to match up the signals provided in the top file.

The connections for the SPI (serial peripheral interface) are not intuitive, therefore the assignments are provided to you here:

```
.spi_master_cs      (sd_dat3),    //  spi_master.cs
.spi_master_sclk    (sd_clk),     //        .sclk
.spi_master_mosi    (sd_cmd),     //        .mosi
.spi_master_miso    (sd_dat0),    //        .miso
.spi_master_cd      (),           //        .cd
.spi_master_wp      (),           //        .wp
```

There are two 'wires' in the top file called "stimulus" and "response". These wires are for connecting .egm_interface_stimulus to .stimulus_in_export and .egm_interface_response to .response_out_export as these signals come to the top file only to connect these cores together – these signals do not leave the FPGA.

2.5 Compiling the Project

At this point you should be able to compile the project. Select 'Processing>Start Compilation' to compile the project and cross your fingers for a successful compile. If you run into trouble here please ask the lab instructor or TAs for help.

2.6 Hardware Wrap-up

With the hardware design complete the next step is to configure the FPGA on the board with the image that was generated by Quartus during the compile procedure. The Quartus programmer can be accessed from the menu bar at 'Tools>Programmer'. With the programmer window open ensure the LogicalStep board is detected by the programmer as shown in Figure 28.

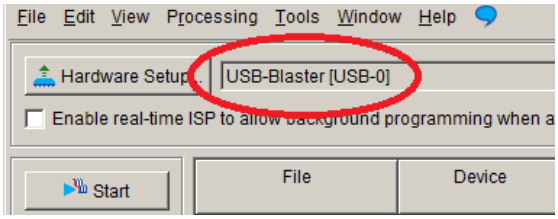


Figure 28: Qartus Programmer Setup

If you do not see a USB-Blaster connected click the 'Hardware Setup…' button and double click the USB-Blaster from the list of 'Available Hardware Items' and close the 'Hardware Setup' window.

Click the ‘Add File…’ button on the left side of the Programmer window, navigate to the ‘output_files’ folder within your project folder and select the .sof file generated by Quartus during the compile. Ensure the ‘Program/Configure’ checkbox is checked and then click the ‘Start’ button to configure the FPGA with the hardware design you generated in Quartus. Once the FPGA is successfully configured, you have actually turned the logic building blocks within the FPGA into an arrangement that encompasses all of the logic required for the FPGA to be a complete microprocessor – known as a soft core processor. The next step is develop the code that will run the processor.

3 Software Development

At this point it is time to start working with the NIOS II Software Build Tools, which are based on the Eclips IDE (integrated development environment), discussed earlier and generate software to exercise the hardware. The Altera Eclipse environment has a built in project called Board Diagnostics that will work with the hardware you built and configured the FPGA with. It is designed to allow users to exercise the processor and some of the peripherals from a console interface within the NIOS II Software Build Tools. Starting with the board diagnostics program is a quick and easy way to learn how to build a software project and get it working on the board – while at the same time ensuring much of the hardware you assembled in Qsys is working.

3.1 Building the Board Diagnostics Project in the NIOS II Software Build Tools

Start by launching the NIOS II Software Build Tools from the Windows start menu – it is best to navigate through 'All Apps' to the Altera folder instead of using the Start menu search function as it is easy to find the wrong program using this approach.

When you open Eclipse for the first time it will ask you for your workspace. Navigate to your project folder and create a folder named 'Software' within your project folder and set this as your workspace.

Watch out!

It is important that you don't change your workspace relative to your project folder or use a copy of your project folder moved to another location or rename your project folder. Eclipse remembers file paths to make things easier but if you move things around it's easy to break the project which can be difficult to fix because all the default paths are wrong.

To create the board diagnostics project in Eclipse select 'File>New>Nios II Application and BSP from Template'. In the dialog window that opens you need to select the projects .sopcinfo file. If you have followed the earlier directions properly then the file you want to select here is the LogicalStep.sopcinfo file. After the Eclipse is done loading the .sopcinfo file which can take a few seconds the CPU name field will also be populated with the name of your NIOS processor as defined in Qsys. Note that the .sopcinfo file contains all the addresses and details about the hardware you defined in Qsys and is required by Eclipse so that your software can interact with the hardware appropriately – more on this will be discussed later.

Give your software project a name in the 'Project name' field such as "board_diagnostics", select the 'Board Diagnostics' template in the 'Project Template' section and click finish to create the project.

Once the project make is complete you'll see two projects in the 'Project Explorer' pane on the left. One is the project itself and the other is the BSP for the project. Anytime you create a new project or if you change the hardware and recompile in Quartus you will need to regenerate the BSP. This is done by right clicking the bsp project in the 'Project Explorer' pane and then selecting 'NIOS II>Generate BSP'.

To compile the project, the BSP and download the program to the NIOS processor that resides on the configured FPGA simply right click the project in the 'Project Explorer' pane and select 'Run As>NIOS II Hardware'. Once the code downloads to the board the 'Console' window at the bottom of Eclipse will display a menu that allows you to test the LEDs, LCD, pushbuttons, and the seven segment display. If you are missing one of these menu items it means there is a problem with the hardware – either the hardware block wasn't added in Qsys, the hardware block was incorrectly configured or the name is incorrect in Qsys. Take a few minutes now to test each of the hardware components and ensure they are working properly.

3.2 Getting to know the NIOS/Eclipse IDE

Before we send you off into the wild world of embedded development for your lab 1 deliverable it is worthwhile to take a short tour of the IDE environment and some of the resources available to you so you are better equipped for success.

We'll start with a description of how the hardware that you built can be controlled from within the code. The process of creating the project and pointing the environment to the .sopcinfo file is the first step and generating the BSP is the second step – as we did above. Let's take a quick look at the result of these actions and how it can help us. In the Eclipse 'Project Explorer' pane click the '+' beside the 'board_diagnostics_bsp' to expand the project you just built. Then expand the file 'system.h' and you will see a long list of define statements in the 'Project Explorer' pane. Scroll down the list to find the define 'BUTTON_PIO_BASE' and double click it. You'll notice that the 'BUTTON_PIO_BASE' represents a long hex number which is actually the hardware address of the button PIO. So, in your code when you want to access/refer to the push buttons you'll use this base address.

Deep Dive

If you go back to Quartus, open Qsys and find the button_pio block you'll note that in the 'Base' column for that IP the address displayed is the same as the one reported in the system.h file. The step of generating the BSP pulls all the Qsys base addresses into Eclipse.

Going forward you'll want to refer to the system.h file anytime you want to access new hardware to find the define for its base address. At the top of your C file, you'll also need to add:

```
#include "system.h"
```

Altera has provided some handy macros to allow easy access to the PIO hardware included in your project. To use the macros you need to add the include statement:

```
#include "altera_avalon_pio_regs.h"
```

With this header file included you'll be able to easily read and write to the PIO registers. For instance, to read the pushbutton switches and save it to a variable called buttons you would write:

```
buttons = IORD(BUTTON_PIO_BASE,0);
```

Which calls the input/output read macro and passes the address and register offset as arguments. For the PIO cores a read or write to the base address (i.e., register offset of 0) will read or write the data from/to the parallel port.

Reading the pushbuttons using the code above stores a number that represents the state of the four push buttons simultaneously according to how their signals appear on the PIO. The push buttons are active low. If no push buttons are being pressed when the port is read, then the port would be b'1111 or 15 in decimal. A read while push button 0 is being pressed results in b'1110 and therefore the 'buttons' variable above would contain the number 14.

As a complement to the read macro Altera provides the IOWR macro to enable writes to a PIO. If you would like to turn on every other LED on the logical step board you could write:

```
IOWR(LED_PIO_BASE,0,0xAA);
```

Where there are now three arguments, the hardware address, the register offset, and data to write to the parallel port. In this case the data is 0xAA or b'10101010 which turns on every other LED.

To understand what the register offset argument is, take a look at the table below that describes the assignment of the PIO that drives the seven segment display and the segment map in Figure 29. The port is 16 bits wide to drive all the LEDS but you'll notice there are two totally different mappings that can be used as shown in Table 20. This was custom designed hardware for the LogicalStep board to support both Altera's board diagnostic software at register offset of 0 and a more user friendly mapping at a register offset of 1. Both registers are part of the seven segment core, so they have the same base address but they have a different register offset.

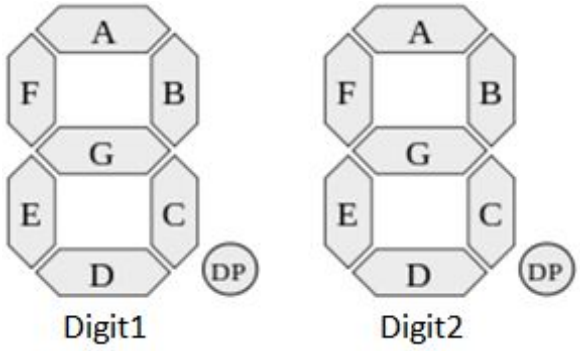


Figure 29: Segment Mapping for Dual 7 Segment Core

Table 20: Register Offsets for Dual 7 Segment Core

Bit Number →		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register Offset	Dir.	Digit 1								Digit 2							
0	W	$\overline{DP}$	$\overline{A}$	$\overline{B}$	$\overline{C}$	$\overline{D}$	$\overline{E}$	$\overline{F}$	$\overline{G}$	$\overline{DP}$	$\overline{A}$	$\overline{B}$	$\overline{C}$	$\overline{D}$	$\overline{E}$	$\overline{F}$	$\overline{G}$
1	W	DP	G	F	E	D	C	B	A	DP	G	F	E	D	C	B	A

Depending on which IP block in Qsys you are trying to access through your code there will be different registers available. For instance, a simple read from or write to a PIO uses the base address with a register offset of 0. However, there are a total of six registers in the PIO core four of which are both read and write. The details of the PIO core can be found in the Altera's PIO Core datasheet which provides a wealth of information about how it works and how to use it.

The majority of the IP cores in your project are provided by Altera and information on how to use them can be easily found online.

Tips and Tricks

It is worth looking up both the PIO Core and the Interval Timer Core datasheets from Altera to assist you while working in the lab. See Altera PIO core or Altera Timer Core.

The EGM and Dual 7 Segment core were made in-house at the University of Waterloo, the details of these cores can be found in Appendix A.

Deep Dive

The NIOS processor communicates to peripheral devices using the Avalon bus which was developed by Altera. The Avalon bus is in fact not a bus in the conventional sense as all bus connections are made in the FPGA fabric like all other hardware configured in the FPGA. The Avalon bus is better understood as a construct that defines the available interface between a master and slave device, which in this case is the NIOS and a PIO in Qsys. The register offset is born out of a 32 bit address that is defined by the Avalon bus, an address often used to access different features within an Avalon slave device.

Take Away

Use the system.h file to determine the defined base address of a hardware core you are trying access. Use Altera's IORD and IOWR macros and the base address to move data to and from the IP cores in your Qsys system. Download IP core datasheets from Altera for more information about the register offsets and how to use their cores. See Altera PIO core or Altera Timer Core. Check Appendix A for details about the EGM and Dual 7 Segment cores.

3.3 Lab 1 Software Project Setup

Now that you have verified the functionality of your hardware using the Board Diagnostics template project, it is time to create a project for your Lab 1 code.

1. From the top menu select File > New > Nios II Application and BSP from Template
2. Select your .sopcinfo file as you did for the Board Diagnostics project using your project path
3. Give your project a name (no spaces)
4. Select the Hello World project from the Templates list as shown in Figure 30
5. Click Finish

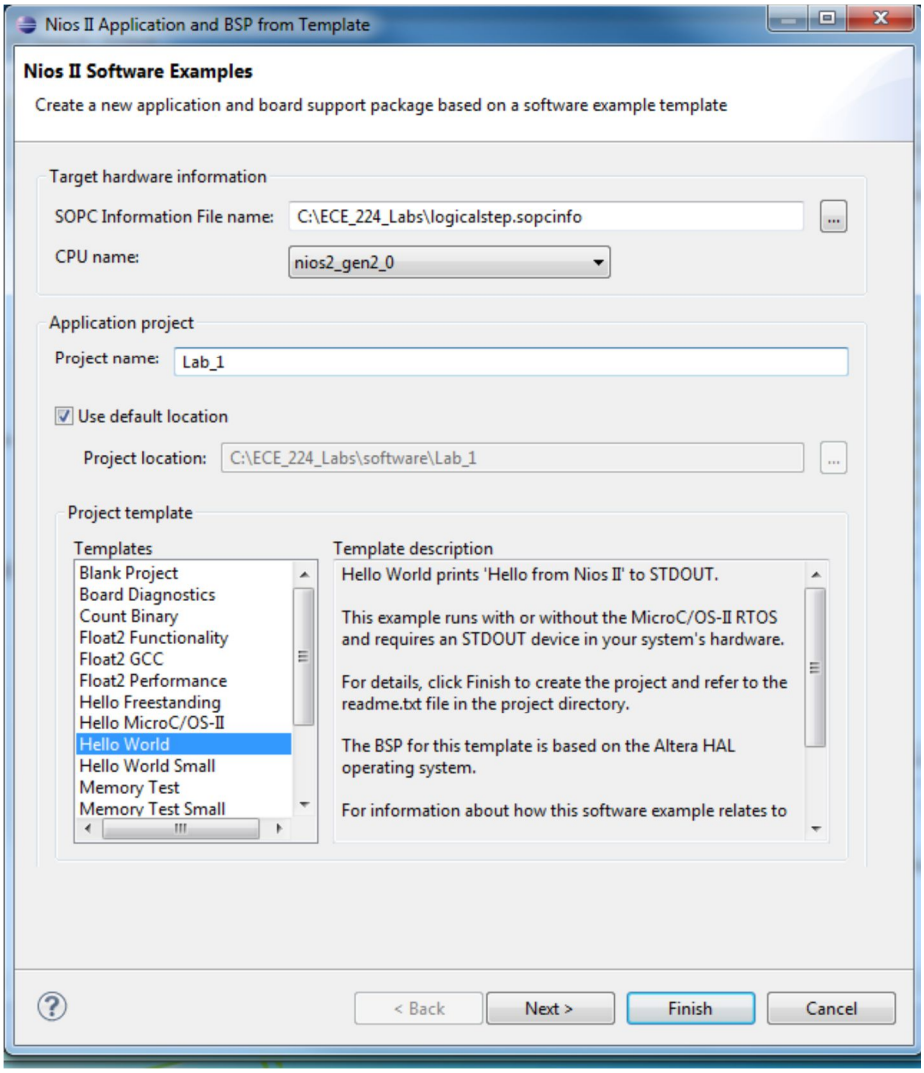


Figure 30: Configuring a New Application from a Template

Now that we have the project setup there are two changes to the BSP settings that need to be made to ensure the data produced for Lab 1 is as clean as possible.

1. Right click on the "*project_name*_bsp [logical step]" and selct Nios II -> BSP Editor…
2. When the BSP Editor opens, it should be on the main tab by default. Change the "sys_clk_timer" to "none" as shown in Figure 31.

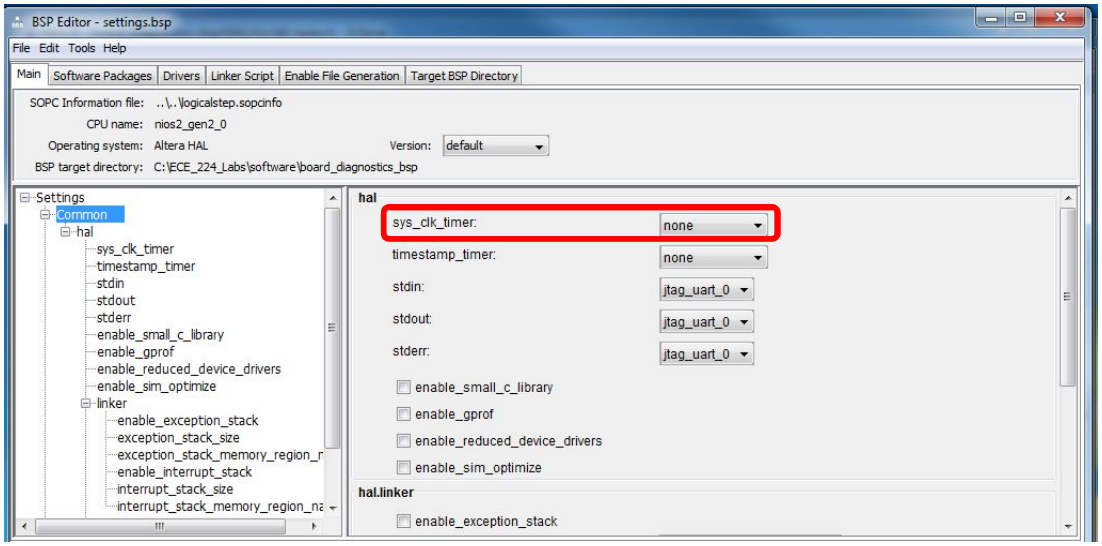


Figure 31: Changing the sys_clk_timer Settings in the BSP

3. Now select the Drivers tab. Check the box to "enable_small_driver" for the "altera_avalon_jtag_uart_driver" as shown in Figure 32.
4. Click Generate

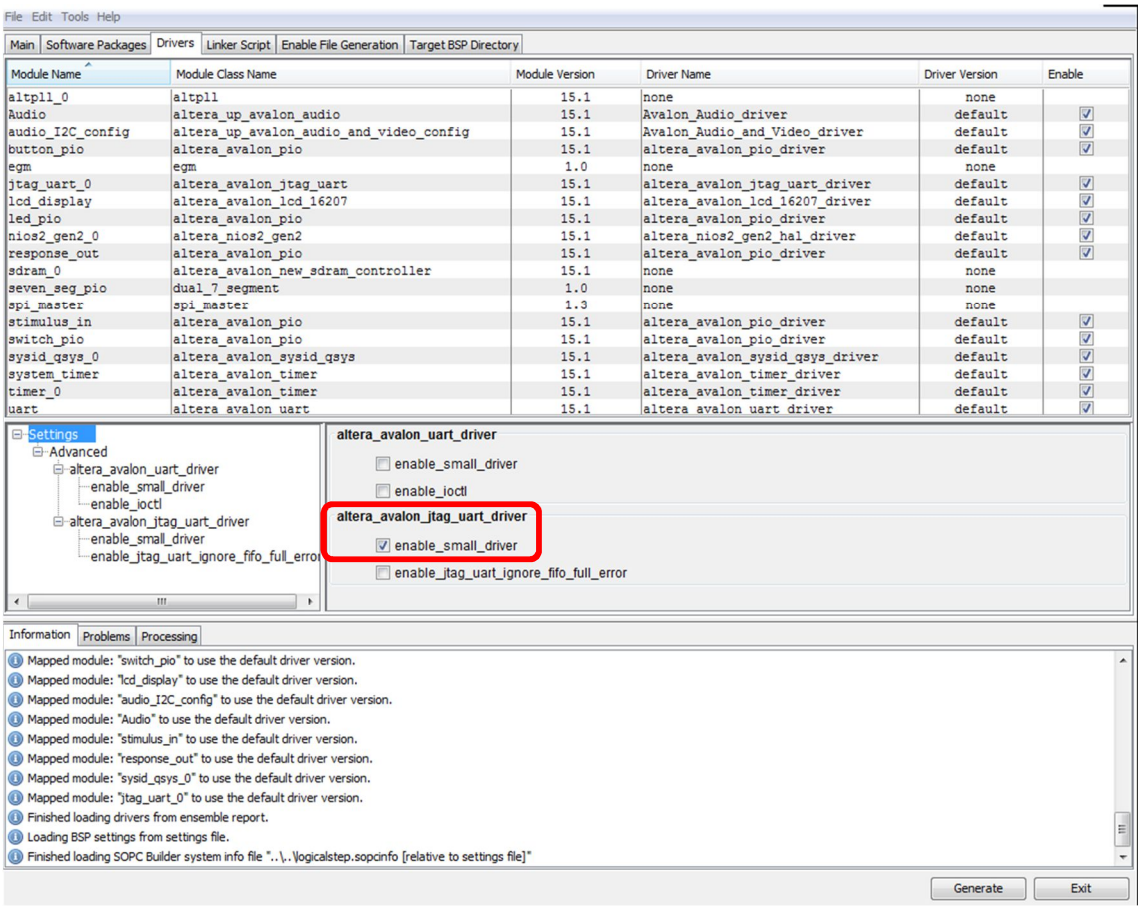


Figure 32: Enabling the Small Driver for the JTAG UART

3.4 Try It

Before you begin writing your code for lab 1, test your understanding of reading and writing to the PIO core by completing the following exercise:

1. In the application you just created, use the IORD and IOWR macros to write code so that when PB0 is pressed, LED 0 turns on, when PB1 is pressed, LED 1 turns on etc. In other words, pushing each push button should turn on the corresponding LED and the LED should turn off when the push button is released.
2. Demonstrate your working code by the start of your second lab session

Take Away

You have now verified that your hardware works using the Board Diagnostics project. You have also practiced how to read and write peripherals connected to your system by the PIO core using your Lab 1 project. You will be asked to demonstrate the control of the LEDs from the pushbuttons to a lab team member at the start of your second lab session.

4 Lab 1: Experimenting with Polling and Interrupts

4.1 Introduction

The objective of lab 1 is to provide students with a thorough understanding of how a processor deals with external events. Often the main approaches used in dealing with external events are polling and interrupts so for this lab you will be required to implement both tight polling and interrupts to respond to external events.

Deep Dive

The approach to tight polling used in lab 1 is a hybrid approach as the processor is still able to get background tasks done during part of the cycle before it starts tight polling for the stimulus pulse. This approach results in latency values identical to normal tight polling but allows background tasks to be completed as well by using a characterization approach to determine if background tasks can be safely completed between stimulus pulses.

With both implementations you will be required to run experiments and collect data that will allow you to evaluate the benefits and drawbacks of each approach to handling external events. To facilitate experimental data collection you will have to write a program that can automate the experiments to test a large number of variables automatically and gather the required data.

4.2 Overview

In lab 1 you will be required to use the NIOS to service external events using tight polling and interrupts. The external events that you need to service are created by a hardware core in Qsys called the Event Generation Module (EGM). The EGM sends a pulse to the NIOS processor (the stimulus) and your code needs to detect the stimulus and send a pulse back to the EGM (the response) as soon as possible. Every time the EGM sends a stimulus pulse the NIOS must send a response pulse back as soon as possible. The EGM internally measures the NIOS's response latency by measuring the time elapsed between the stimulus pulse and the response pulse – this latency can be used to help evaluate the difference between polling and interrupt based approaches to dealing with external events.

In your code you will also be required to call a background task function that simulates other work a processor could be required to do while waiting for an external event to happen. Therefore the difference between polling and interrupts can be evaluated experimentally by

comparing both the response latency and the amount of background work each approach can complete.

4.3 EGM Module

The EGM IP core is a custom core developed at the University of Waterloo for this lab. When you built the hardware for the lab in Qsys you included the EGM core and two PIO cores named stimulus_in and response_out. In the top file you used two 'wires' to connect the EGM stimulus and response ports to the related PIO cores as shown in Figure 33.

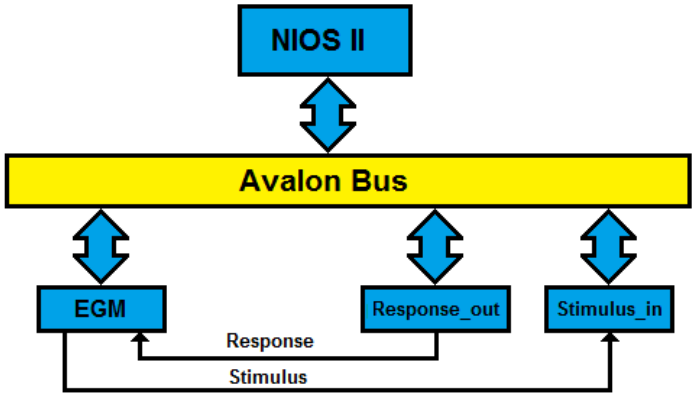


Figure 33: EGM and NIOS Topology

Note that the NIOS processor can communicate with all three cores over the Avalon bus. To gather data for the lab report you will be required to implement and test both approaches (polling and interrupts) on the Stimulus_in PIO core. When your code detects the stimulus you will then simply send a response pulse using the Response_out PIO core.

To start the experiment you will have to configure the EGM to send out the appropriate type of pulses and then enable the EGM. Once the EGM is enabled it will send out a stream of pulses for a specific period of time and then it will stop the pulses automatically. Once the EGM stops, your code will be required to access the EGM and gather the average response latency and total missed pulses for the experiment and then disable the module to reset it.

To configure the EGM the user must set a 'pulse width' and 'period' value in the respective registers of the EGM prior to enabling the EGM to run the test. The EGM has an internal counter that is incremented by the clock. When the EGM is enabled it will set the stimulus pulse high until the counter value matches the user defined 'pulse width' value at which point the EGM will set the pulse low. The stimulus is kept low until the internal EGM counter equals the

user defined 'period' value at which point the counter is reset and the EGM cycle restarts by setting the stimulus pulse high again as shown in Figure 34.

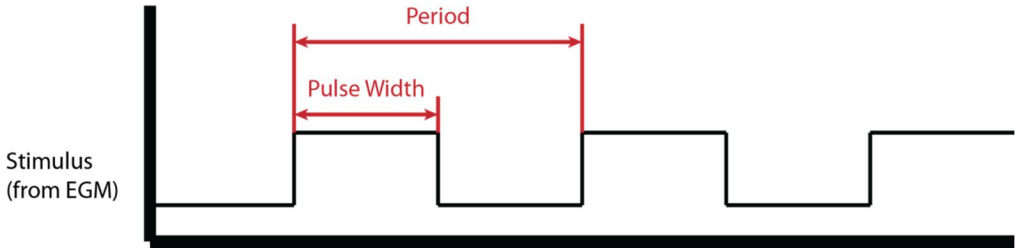


Figure 34: EGM Stimulus Pulse-train Settings

The stimulus cycle continues to repeat until the EGM is finished the test and it stops pulsing. The user must query the EGM's 'busy' register during the test to determine when the EGM is finished. A value of 0 in the EGM's 'busy' register indicates the EGM is finished the test. When the test is complete the user must query the 'missed pulses' and 'average latency' registers to record the results for that test before setting the **EGM enable low**. The register offset values for interfacing with the EGM over the Avalon bus are described in Appendix A.

Every cycle the EGM measures the response latency of your code implementation by counting the elapsed clock cycles between the leading edge of the stimulus pulse (sent by the EGM to the NIOS) to the leading edge of the response pulse (sent by your code in the NIOS back to the EGM) as shown in Figure 35.

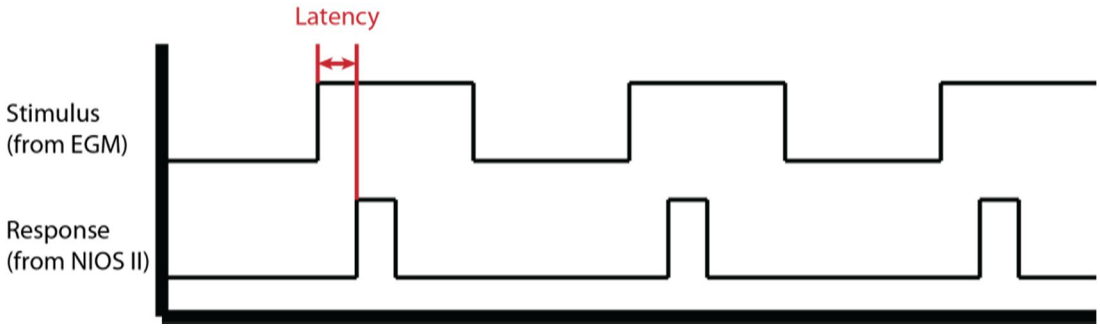


Figure 35: EGM Latency Measurement

Every latency measurement gathered in a test is used by the EGM to generate an exponentially weighted moving average of the response latency that is available to the user at the end of the test.

Watch out!

While the trailing edge of the pulses are ignored by the EGM's latency measurement, the user must ensure the response pulse is set to low before the EGM sends the next stimulus pulse to avoid a missed pulse being counted.

4.4 Code Development for Executing a Test

For lab 1 the first objective is writing code to execute a test using the EGM for both tight polling and interrupt response to external stimulus. Each test must meet several objectives in the following order of importance,

1. No Missed Pulses: tests that end with some pulses being missed are considered failed tests. Under certain circumstances having tests that result in missed pulses will be unavoidable. For those tests record the test data but graph the data in a different color to indicate the test results included missing pulses.
2. Shortest Latency: The code should be written to achieve the lowest average response latency.
3. Highest Background Task Completion: The code should be written to complete the most amount of background tasks during the test.

Tips and Tricks

Since shortest latency is more important than completing the most background tasks the focus must be maintaining the lowest latency first while completing the highest amount of background tasks second. This concept is most critical during tight polling code development.

The following background task function should be copied into your .c file:

```
int background()
{
        int j;
        int x = 0;
        int grainsize = 4;
        int g_taskProcessed = 0;

        for(j = 0; j < grainsize; j++)
        {
                g_taskProcessed++;
        }

        return x;
}
```

4.5 Recommended Approach to Interrupt Test Development

The interrupt based stimulus response is easier to complete then the tight polling approach as the background task call can simply be executed in a while loop that is interrupted each time a stimulus pulse is received. It is recommended to use a simple approach to a response pulse within the 'stimulus_in' ISR (i.e., set 'response_out' high then immediately set the 'response_out' low within the ISR code). Additional information on interrupts in the Nios II system can be found in Appendix B.

Tips and Tricks

It is recommended to implement the interrupt approach to tight polling first. Once this approach is working then start on the tight polling approach.

Watch out!

Regardless of whether you are doing tight polling or interrupts, to record good data you **must** ensure the background tasks are not being run after the EGM has disabled itself. This means that **before every new background task call you must ensure the EGM is still active** and by extension the test is still running. If you don't do this your results will be incorrect.

4.6 Recommended Approach to Tight Polling Test Development

Completing the tight polling test is more challenging than interrupts because of the requirement to do background work while maintaining a low amount of response latency. The way to achieve both objectives is to poll for the edge of the stimulus and once the stimulus is detected, run the background task a given number of times before starting tight polling again to detect the next stimulus pulse as shown in Figure 36.

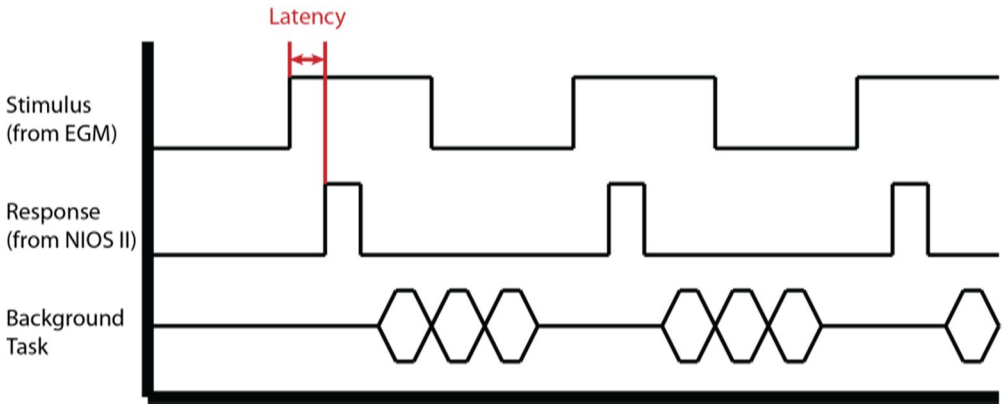


Figure 36: Background Tasks during Tight Polling

Note how in the figure the background task is run four times after the stimulus is responded to. After the four background tasks are complete, the NIOS resumes tight polling to detect the next stimulus pulse with a minimum response latency while at the same time completing the most background tasks. Note that you **must** ensure the EGM is still enabled anytime you intend to call another background task (otherwise your results will be incorrect).

The problem with this scheme is that it is impossible to know *a priori* how many background tasks can be executed and completed before the next stimulus pulse is sent to ensure the lowest latency objective is met.

A characterization approach **must** be used during the first cycle of the test to automatically determine the number of background tasks that can safely fit within each period. With the test characterized, every subsequent cycle can be executed with a safe number of background tasks to ensure minimum cycle latency. An example of how this would work is shown in Figure 37.

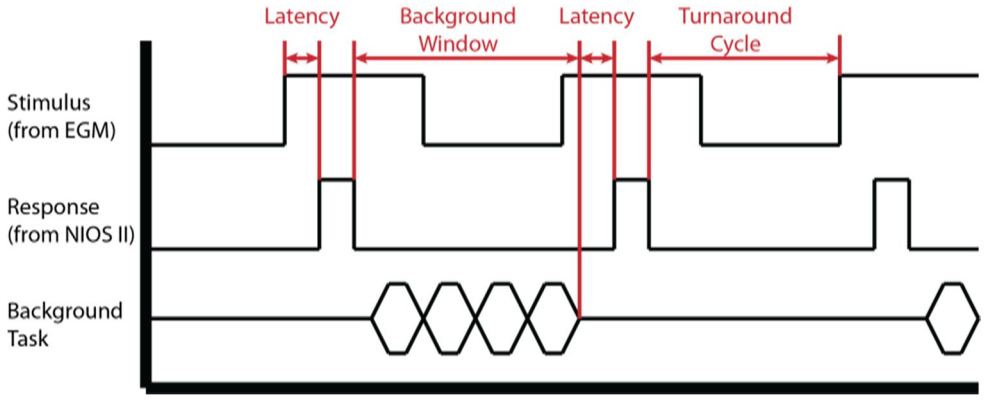


Figure 37: Characterising Background Tasks during Tight Polling

For the first cycle the NIOS would use tight polling to respond to the first stimulus pulse. As soon as the response pulse is sent the NIOS will call a background task. When the background task is complete the NIOS will check to see if a new stimulus pulse is received. As long as no new stimulus pulse is detected the NIOS continues to call the background task. Eventually the NIOS will detect a new stimulus pulse, in the figure above it occurred after four background task calls. The characterization code then determines that four calls to the background task are safe (no latency attributed to background calls) between responding to a stimulus pulse and starting tight polling to detect the next stimulus pulse.

Tips and Tricks

During the characterization cycle above there will be extra latency recorded for the second cycle because of the extra background task. However, because the average latency is calculated using an exponentially weighted moving average it will have little to no effect on the final reported average latency.

You are required to write code that automatically characterizes the number of background tasks that can be called each cycle while maintaining the shortest possible response latency. The code must have a way to detect the new stimulus pulse as distinct from the previous stimulus pulse.

Watch out!

If you use a while loop to do tight polling one of the conditions of while statement must be that the EGM is still running – if not your code will hang at what appears to be random times in an experiment because the tight polling is waiting for the next stimulus pulse after the EGM has been disabled.

Tips and Tricks

During the cycles within the test you will want to be checking the EGM busy register to determine when the EGM is disabled and the test is complete.

4.7 Code Development for Executing an Experiment

An experiment in the context of lab 1 is a series of tests (where a test is one run of the EGM until the EGM stops pulsing) which are run in an automated fashion by code that you will write. The 'experiment code' will run the 'test code' over and over with different values for the period and pulse width until the entire test space has been suitably documented with test results (it is recommended to print the test data to the console using CSV format so you can easily import the data into excel). The automatic characterization for the tight polling test is required because the experiment code will run the test code thousands of times to gather the required data. You can write one experiment code for testing tight polling and a second experiment code for testing interrupt response – however they must both be in the same source file using #ifdef statements.

The latency and background task results should cover all the required period values as specified in the Lab 1 Objectives and Deliverables section.

4.8 Lab 1 Objectives and Deliverables

Using the information presented here in chapter 0 as a background to the lab, you need write code to complete two experiments. One experiment for tight polling and one for interrupts with both implementations in the same source file (use #ifdef statements to select between the different implementations at compile time). Each experiment must test the full range of EGM stimulus pulse period settings from a period of 2 to a period 5000 in steps of 2 clock cycles (i.e., 2, 4, 6, 8…). You should always maintain a duty cycle of 50% (i.e., if you run a test with the EGM period setting at 400 then the pulse width setting should be 200). The experiment code can be

a 'for' loop that increments through different EGM period settings each time the test is run. To gather the data you can use comma delimited printf() statements to report the period, dutycycle, total background tasks run, average latency, missed pulses out to the console. From there you can move it into excel to graph it for analysis.

Table 21: Summary of Project 1 Deliverables

Deliverable	Deadline	Percentage of Lab 1 Grade
Session 2 Demo	Beginning of second lab session	10 %
Full Demo	End of third lab session	40 %
Report	See syllabus for deadline and information on grace days	50 %

4.8.1 Session 2 Demo

At the beginning of the second lab session, you will need to show that your hardware is working and you are able to use the IORD and IOWR macros by demonstrating the functionality described in Section 3.3.

4.8.2 Full Demo

You should be ready to demonstrate your implementation by the start of the third lab session. For the demo, you will be expected to explain how you implemented your experiments. You will also be asked to run your code. Both partners are expected to be able to answer questions regarding the implementation of your project.

4.8.3 Experiment Results

For lab 1 you are required to test tight polling and interrupts as discussed previously in this section by gathering data that relates the EGM period setting to both the background tasks completed and the average latency. Therefore, you should have four datasets gathered to write your report. It is recommended that when you are gathering and analysing data in the lab you

ask the lab instructor or TAs for guidance on whether your graphed data looks correct before starting the report.

4.8.4 Report

Your report must **not** include a full dataset or source code; your report must include the four graphs of the collected data as described below. Your marks for the report will be based on your ability to recognize important details in the presented graphs and your ability to correctly explain the root cause of those details. For each graph presented it is important to call attention to any details in the graphed data (overall trend, slopes, curves, discontinuities etc) and provide a concise description of the root cause for the trend in the data. This will require a thorough understanding of how your code works and how the EGM works within your approach to tight polling or interrupts so that you can accurately explain the trends in your graphed data.

Your report should include four sections that cover the following details:

- Present your background task completion data from the tight polling experiment in a plot ('Background tasks vs EGM period' from a period setting of 2 to 1500 clock cycles) and explain the results presented with the aid of logic waveforms. You must point out sections in the graph and explain the root cause relating to: missing pulses, the overall trend-line, discontinuities and the negative sloped curve between discontinuities. **(20%)**
- Present your background tasks data from the interrupt experiment in a plot ('Background tasks vs EGM period' from a period setting of 2 to 1500 clock cycles) and explain the results presented. You must point out sections in the graph and explain the root cause relating to: missing pulses, the overall trend-line, background task efficiency. **(20%)**
- Present your latency data from both the tight polling and interrupt experiments in a single plot (Latency vs EGM period from a period of 2-1500 clock cycles) and draw comparison conclusions about the different approaches using the data presented. You must point out sections in the graph and explain the differences observed – there are two key differences we expect you to explain. **(20%)**
- Present your background tasks data from both the tight polling and interrupt experiments in a single plot (Background tasks vs EGM period from a period of 2-5000 clock cycles) and draw comparison conclusions about the different approaches using the data presented. You must point out sections in the graph and explain the differences observed – there are two key differences we expect you to explain. **(20%)**
- You will also be graded on the quality of your presentation. Areas of interest in the graphs must be clearly labeled on the graph and the description presented under the relevant label in the text. The report presentation should be consistent with a formal lab report for full marks in presentation. The report must not exceed 5 pages in length, one page per graph and explanation, plus one title page – minimum 11 point font **(20%)**

Tips and Tricks

When discussing data in the report be sure to pay careful attention to discontinuities, slopes, curves etc. in the graphs of recorded data.

Watch out!

A written explanation of what the graph looks like will receive no marks. Marks will only be awarded for specific details in the graph (dominant details) that are called out and the underlying root cause is explained with reference to the code or function of the experiment.

Appendix A: Custom IP Cores

Dual 7 Segment Core Register Description

The Dual 7 Segment core has two register offsets, 0 and 1. The register offset 0 is the default offset and is mainly used by the NIOS II Build Tools Board Diagnostics template to drive the dual 7 segment display.

Users wishing to write their own code to interface with the seven segments on the LogicalStep board are encouraged to use the more straight forward register 2 offset in Table 22. As an example, to display '7.0' on the seven segment display using register offset 1 you would write b'1000011100111111 or 0x873F to the register. However both offsets are detailed in the table below.

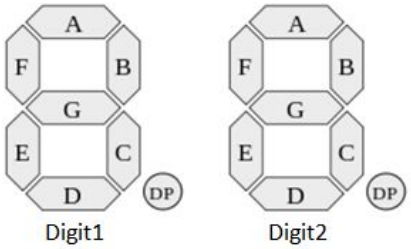


Figure 38: Segment Mapping for Dual 7 Segment Core

Table 22: Register Offsets for Dual 7 Segment Core

Bit Number →		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register Offset	Dir.	Digit 1								Digit 2							
0	W	$\overline{DP}$	$\overline{A}$	$\overline{B}$	$\overline{C}$	$\overline{D}$	$\overline{E}$	$\overline{F}$	$\overline{G}$	$\overline{DP}$	$\overline{A}$	$\overline{B}$	$\overline{C}$	$\overline{D}$	$\overline{E}$	$\overline{F}$	$\overline{G}$
1	W	DP	G	F	E	D	C	B	A	DP	G	F	E	D	C	B	A

EGM Core Register Description

The EGM core has 6 register offsets that can be accessed using Altera's IORD and IOWR macros. Table 23 shows the register map of the 6 register offsets, the direction of the register and which bits are applicable for that register. The typical approach to using the EGM core is to first set the period and pulse width registers to define the type of stimulus pulse you want the EGM to send. Figure 39 shows the effect of setting the period and pulse width settings on the stimulus pulse. Note that the period setting defines the number of clock cycles between rising edges of the stimulus pulse train and the pulse width setting defines the number of clock cycles that the

stimulus pulse stays high. The pulse width setting should always be shorter than the period setting for predictable operation of the EGM.

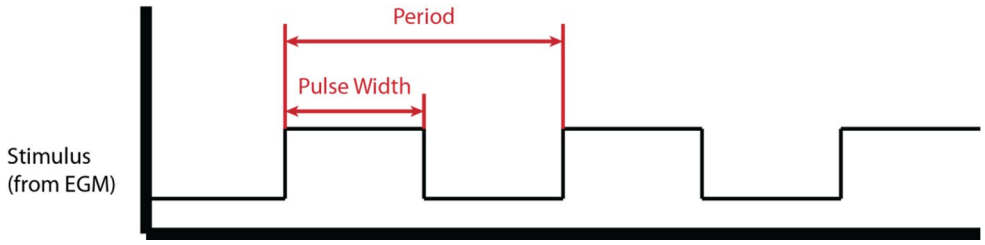


Figure 39: EGM stimulus pulse

Once the period and pulse width has been set the user can start the EGM by setting the enable bit. The EGM will send the defined pulse train for a predefined interval during which the average response pulse latency is measured using an exponentially weighted moving average filter and missed pulses are incremented every time a stimulus pulse does not get a response. There is no latency penalty for missed pulses – missed pulses are completely ignored by the latency tracker.

Since the EGM runs for a predefined interval once enabled the user must continuously sample the busy register to determine when the busy bit is cleared and the EGM stimulus pulse train is complete. Once the busy bit is cleared the user can then read the final latency and missed pulse values. The user **must** also clear the enable bit before being able to restart the EGM for another test.

Watch out!

If you don't disable the EGM after the final run the hardware will not work the next time you try to use it. Enabling the EGM when it is already enabled will not start an EGM cycle.

Table 23: Register Offsets for EGM Core

Register	Dir.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0:Enable (BASE)	W																X
1:Busy	R																X
2:Period	W	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
3:Pulse Width	W	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
4:Latency	R	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
5:Missed	R	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Appendix B: NIOS Interrupts

Interrupts have many useful purposes, in embedded applications the concept of interrupt hardware provides the ability for the processor to continue doing useful work while waiting for an external event to happen. This is in contrast with polling which monopolizes the processor while waiting for an external event.

As is always the case with soft core processors on FPGAs there is a both a hardware component and software component to consider. Since the hardware is flexible and a function of design we must first consider the hardware design to determine the requirements of the software. In the case of handling interrupts, the code you will write that utilizes interrupts needs to consider the hardware you have designed. So we'll discuss the hardware first.

Hardware Considerations

When you define a NIOS system in Qsys there is the ability to add signals specifically designed to interrupt the processor via the NIOS 'Interrupt Receiver' which is standard on the NIOS processor. Should the NIOS be interrupted by a core the NIOS can query the core over the Avalon bus to determine the reason for the interrupt in more complex cores such as the PIO.

Deep Dive

The NIOS interrupt receiver is integrated into the NIOS processor and can accept up to 32 interrupt signals. Normally each IP core that uses interrupts will be assigned a single interrupt line. The NIOS II/f processor also supports an external vectored interrupt controller which is significantly more advanced than the internal controller. See Altera VIC.

Before any interrupt code will work, the core you intend to use interrupts with needs to be connected to the NIOS Interrupt Receiver in Qsys as shown below. The number in the connection point defines the interrupt priority and must be unique as shown in Figure 40.

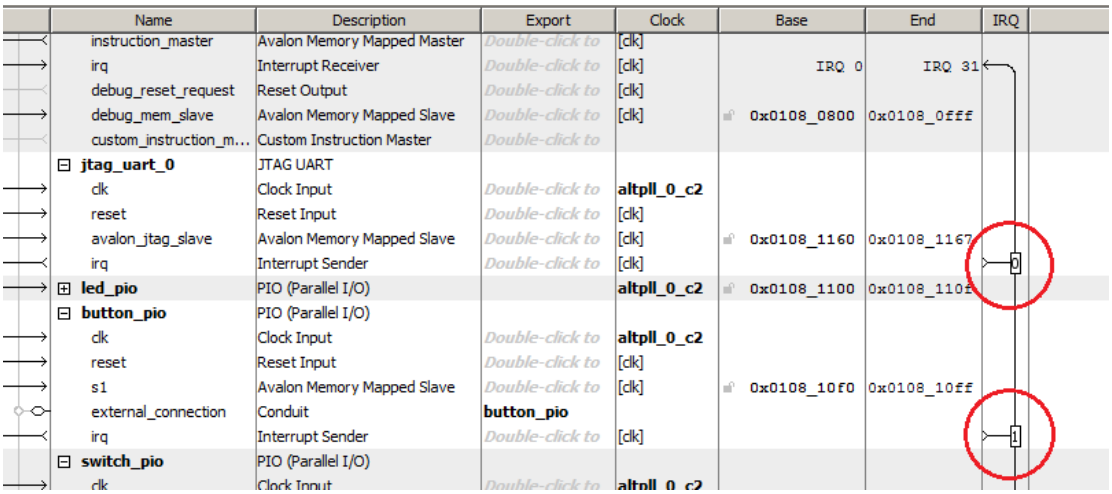


Figure 40: Interrupt Connections in Qsys

The interrupt priority is important because interrupts with higher priority take precedence over lower priority interrupts. Therefore only a higher priority interrupt can interrupt a lower priority interrupt. Note that setting interrupt priority in the NIOS processor can only be accomplished in hardware.

Watch out!

Putting function calls (such as a printf) within an interrupt service routine is considered very bad form. If you did put a printf in the ISR for the button_pio shown above and the interrupt priorities of the jtag_uart and button_pio were reversed you would have problems. The button_pio would have priority and the printf would deadlock the system waiting for an interrupt that is lower priority. Always ensure only appropriate code goes into an ISR – no function calls, spin-waits or while statements. ISR rule of thumb – simply get in and get out.

Some cores have the ability to generate interrupts as you have seen during the building of hardware for the lab. Depending on how the core was designed some of the interrupt settings can only be changed in the hardware which is done by accessing the core in Qsys and then regenerating and recompiling the project. Other interrupt settings can only be accessed in software by adjusting registers in the core over the Avalon bus.

Tips and Tricks

For the Altera PIO core, settings such as interrupting on a level or interrupting on an edge can only be changed in hardware. Settings such as which bits in the port will actually generate an interrupt (interrupt mask) can only be changed in software. See Altera PIO core

Armed with this understanding of hardware side of the interrupts, we'll move on to using interrupts in software.

Take Away

If you want a core to generate interrupts it needs to be connected to the NIOS Interrupt Receiver in Qsys. Interrupt priority is set in Qsys and cannot be changed in software. IP cores can include interrupt settings that can only be configured in hardware or only configured in software or both.

Software Considerations

Addressing interrupts in software is a little more involved than the hardware side. However we can start with the knowledge that our core is connected to the NIOS interrupt receiver and that we have set an appropriate interrupt priority.

There are three main components required in your code to use interrupts:

1. An include for Altera's interrupt controller interface code
2. An interrupt service routine (ISR) which is a special function in your code that is automatically called by the NIOS hardware abstraction layer (HAL) exception handling system when the associated interrupt is triggered and it has the highest interrupt priority.
3. A statement in your main function that registers the ISR with the NIOS HAL exception handling system.

So at the top of your c file you should add the include,

```
#include "sys/alt_irq.h"
```

This include gives you access to the required calls needed to manage your interrupts as described below. Somewhere in your code before your main you should put your ISR, a generic ISR would look like this,

```
static void name_of_your_ISR (void* context, alt_u32 id)
{
    //ISR code goes here

    //Command to clear the interrupt goes at the end of the ISR
}
```

Note that the ISR needs to be declared static to ensure the compiler does not optimize it away. When you write your own ISR give it a more descriptive name then we have given ours (i.e., don't use 'name_of_your_ISR'). The pointer 'context' is provided to pass user specific information to the ISR, it is safe to ignore this as we can get by without it. The 'id' is simply the

hardware interrupt number of the core that is generating the interrupt, this can also be safely ignored as you won't need to use it. Write the code that you want to execute during the interrupt inside the ISR. Generally this code is related to data associated with the device generating the interrupt. In the case of a PIO that has buttons connected to it you would probably want to determine which button was pressed in the ISR and respond to or store the data appropriately.

Tips and Tricks

Usually in an ISR you'll want to signal your main program loop about the data received in the ISR. An often used approach is to declare a global variable in your code to use as a flag that can be evaluated in your main loop. If the interrupt happens the ISR will set the flag and the main program loop will detect it the next time it tests the flag.

Watch out!

If you use a global variable as a flag that is set only in your ISR you should declare the variable as volatile to ensure the compiler does not optimize it away.

The most important part of the code in the ISR is the command that clears the interrupt condition and it should be the last command issued in the ISR.

Watch out!

In the hardware you defined in Qsys the interrupts are being generated by IP cores external to the NIOS. Any core that can generate an interrupt for the NIOS must include a way for the NIOS to access it over the Avalon bus and reset the condition that is causing the interrupt. Because every core is different you will need to check the datasheet of the core to determine which register is used to reset the interrupt and add the required code to do so at the end of your ISR. Failure to clear the interrupt in the ISR will result in deadlock; the processor will never return from the ISR.

Below we have applied the ISR to handle an interrupt from an Altera PIO core named BUTTON_PIO. For the PIO core it turns out writing anything at all to the base register offset 3 causes the interrupt to be cleared. See Altera PIO core

```
static void name_of_your_ISR (void* context, alt_u32 id)
{
    //ISR code goes here

    IOWR(BUTTON_PIO_BASE, 3, 0x0);
}
```

Finally we need to cover registering the ISR with the HAL exception handling system. The point of registering an ISR is to tell the HAL which ISR to execute for a given hardware interrupt id. Here we show how to register the ISR shown just above,

```
alt_irq_register( BUTTON_PIO_IRQ, (void *)0, name_of_your_ISR );
```

This function is called within main and it registers the ISR by passing in the hardware interrupt number, a null pointer and the ISR name. The null pointer we can ignore because it is simply the pointer 'context' passed into the ISR that we are not using. And that's all there is to setting up interrupts in your NIOS code.

Tips and Tricks

You'll want to use the BSP defined hardware interrupt id similar to the registering command above 'BUTTON_PIO_IRQ'. If you forget the BSP define for a core you are registering just search the 'system.h' file in your BSP project in eclipse which includes all the defines for the cores you have added to your hardware in Qsys.

In the case above, if the hardware is properly setup and the interrupt has been registered then if any of the buttons are pressed the BUTTON_PIO core will generate an interrupt and the code will jump into the related ISR. In a random scenario the ISR code could read the BUTTON_PIO register that is associated with the input data and determine which button was pressed. It could also set a global variable with a number that indicates which button was pressed and then clear the interrupt by writing a value to the third register in the PIO core. At that point the ISR would complete and the code would resume where it left off.

Deep Dive

In actual fact the interrupt handling is a little more complex. When an interrupt is received the program is directed to the exception address where code resides that can handle every type of exception that occurs in a NIOS processor. The handler stores register data onto the stack for safe keeping and then the code starts a process to figure out what just happened. After checking several possible causes it will determine that there is a hardware interrupt event and it will determine which interrupt occurred and run the related ISR. When the ISR completes the exception handler code will restore the register data from the stack and reset the program counter to where it was before the interrupt occurred.

Take Away

There are three key elements required in code to use interrupts. An include statement, an ISR and call to register the interrupt in the main. The last thing an ISR needs to do is clear the interrupt that that invoked it. The way an interrupt is cleared is unique to the particular core that generated the interrupt so if in doubt - read the cores datasheet.

Appendix C: Solving NIOS Software Download Issues

The following download process was detailed in step by step fashion in the board diagnostics project so this section will only provide the high level details to aid in troubleshooting download issues. Assuming a stable hardware project, successful compiles start by creating a NIOS project and BSP from template as was described in making the board diagnostics project. When you are ready to compile for the first time three things need to happen:

1. Save the project (this is usually only required for the first build)
2. Generate the BSP
3. Run project as NIOS II Hardware

The third step above actually starts a sequence of events:

1. The BSP is built
2. The project is built
3. A connection to the hardware is made via Quartus
4. The hardware is checked to ensure the hardware on the board exactly matches the hardware described in the BSP
5. The .elf file is downloaded to the soft core processor at the processor is started

All these steps need to happen successfully for your code to run. There are reasons that any of these steps could fail and sometimes it is difficult to figure out which step failed or why.

Step 1 Failure: This is an early failure in the build process and Eclipse mentions an error such as,

make[1]: *** [public.mk] Error 1

But earlier in the console printout you'll notice it mentions that the BSP needs to be generated. It is often related to recompiling the hardware and forgetting the Generate BSP step before compiling the project.

Step 2 Failure: This failure is usually related to syntax errors in the code but Eclipse usually continues through the remaining steps regardless of a failed compile until step 5 when it fails because no .elf file was created in step 2. Use the 'Problems' tab in Eclipse to determine the related problem(s). You can double click syntax related problems and it will show the issue in the code. Syntax errors will be underlined in red in the code and changes to syntax will not be evaluated until the next compile (i.e., red underline will not be removed until after the next compile). If it seems nothing will eliminate a red underline, even deleting the text completely then be sure to save before recompiling.

Watch out!

Error messages that indicate the .elf file is missing are almost always related to syntax errors in the code and indicate a step 2 failure. If you have not already verified your hardware is working by running the Board Diagnostic sample project, then it is also possible there is an error in your hardware. For example, an improperly named or configured SDRAM interface will cause the download to fail.

Step 3 Failure: This failure often occurs during the first time downloading software to the board. If the 'Run Configurations' window sudden pops up during the compile indicating Eclipse can't connect to the hardware follow these steps:

- Click on the 'Target Connection' tab.
- Scroll the window to the right if required and click the 'Refresh Connections' button.
- When done click the 'Refresh Connections' button again if required.
- When the USB Blaster shows up in the lists click the 'Run' button in the bottom of the window

If you get a message that Eclipse can't connect but the 'Run Confirurations' window doesn't open then open the window manually from the menu bar, 'Run>Run Configurations' and follow the procedure above.

Step 4 Failure: If you get an error message that indicates a timestamp mismatch the issue is that the software you are trying to download was not compiled for the hardware on the board and there can be many reasons for this.

- You didn't download the hardware to the board before trying to download software
- You downloaded hardware from a different project folder then the software you are trying to download (verify the file path in the Quartus programmer is correct)
- You changed your workspace relative to your project folder
- You copied your project folder moved it to another location
- You renamed you project folder
- You didn't select the correct .sopcinfo file when you created you software project

If you followed the advice to only ever have one project folder and never move it, copy it or rename it then the main cause would be you forgot to download the hardware to the board. If you can't figure out what's wrong and you didn't make the above mentioned mistakes it is recommend that you follow these steps in order,

- Regenerate in Qsys

- Recompile in Quartus
- Generate BSP in Eclipse
- Run as NIOS II Hardware in Eclipse

If this doesn't work you should try and create a new project and BSP from template in Eclipse and be sure that the .sopcinfo file you are using is actually in your project folder – then copy your source files from the old project to the new project.

Step 5 Failure: This step never fails, if you think this step is failing reread the 'Step 2 Failure'.

Appendix D: Using SVN for Quartus and Nios II Projects

The lab has been setup to use subversion to store your lab projects. Using subversion provides several advantages in the lab,

- Both group members will have access to the lab materials – they will not be tied to one of the group members Nexus account
- In the event of a corruption of your project data you be able to simply restore the latest checked-in version of your project
- If you break previously working functionality in your code while adding new features you have the ability to go back and see the differences
- Using subversion as directed will allow much faster compiles which will save you time in the lab.

Follow these steps to setup and use subversion in the lab:

1. Before you start lab 1 you need to create your project folder. These instructions are written as if you are Group002 – please ensure you use your proper assigned group number when following these steps. Navigate to "C:\LogicalStepSVN" and add a folder inside called **exactly** "LogicalStep". This will be your project folder, anywhere in the manual where it says 'project folder' this folder is what it is referring to.
2. Now right click on the 'LogicalStep' project folder you just created and select 'SVN Checkout…'.
3. The URL is https://ecesvn.uwaterloo.ca/courses/ece224/2017S/labs/GroupXXX/LogicalStep where XXX is your three-digit group number found on Learn. An example for Group002 is shown in Figure 41. Note that the "ece224" part of the path does not change, even if you are an MTE student. Leave the other options as defaults.
4. Click 'Okay' to check-out the repository.

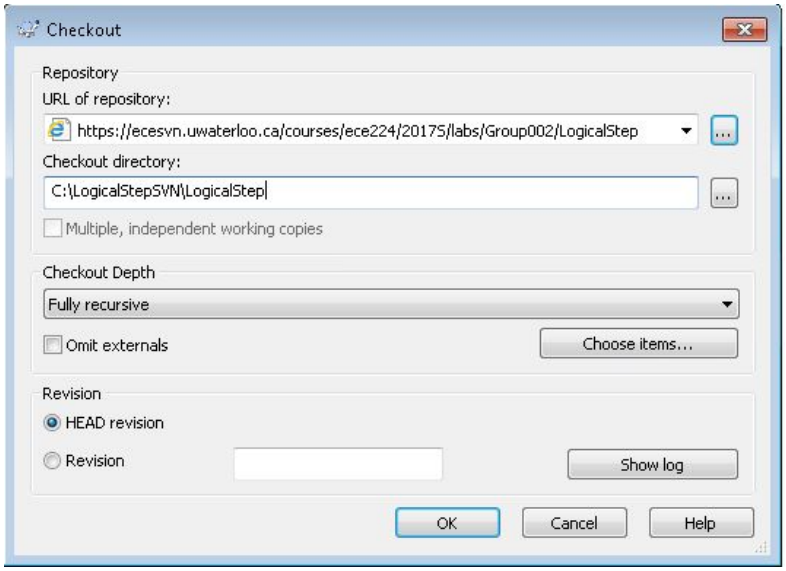


Figure 41: SVN repository checkout

5. You will notice we have prefilled the repository with the file structure and empty files with the names you will be required to check-in at the end of the lab. Please ensure your directory tree now looks exactly like Figure 42. **Important** - if your tree is different than in the figure ask the lab staff for help.

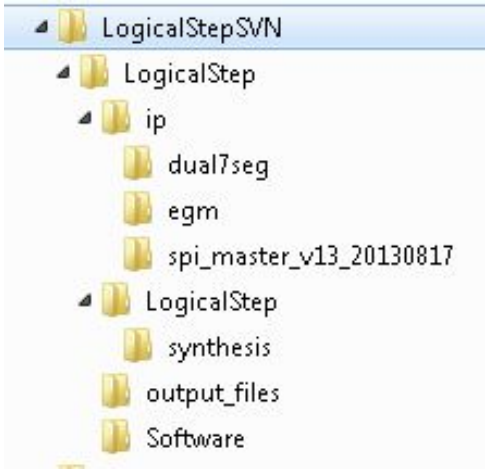


Figure 42: Directory tree of complete project folder

6. Now delete except the 'Software' folder from the LogicalStep project folder (files shown in Figure 43) but do not delete the LogicalStep project folder itself. Go to learn and download the 'LogicalStep.zip' file from the 'Lab 1 Support Files' folder, unzip the folder and copy the contents of LogicalStep folder into your project folder – do not move the whole folder.

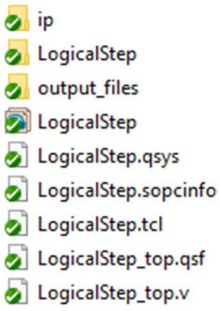


Figure 43: Files to delete

At this point you are ready to start on working on your project (Section 2 of this manual) using this 'LogicalStep' project folder for your project files to go in.

Before the end of the lab session return to this appendix and follow steps 7 and onwards.

7. Once your project is built the files are added, commit them to the SVN by right-clicking on the 'LogicalStep' folder you created and selecting 'SVN Commit…'. This will push a copy of all your files to the ece svn server.
8. To be safe, create a folder to hold your project on your N: drive and follow the steps above to perform the SVN checkout again. A copy of all your files that were checked in should appear in your folder on the N: drive, specifically these files:

LogicalStep.qpf
LogicalStep.qsys
LogicalStep.sopcinfo
LogicalStep.tcl
LogicalStep_top.qsf
LogicalStep_top.v
LogicalStep/synthesis/LogicalStep.qip
LogicalStep/synthesis/LogicalStep.regmap
LogicalStep/synthesis/LogicalStep.v
ip/
output_files/LogicalStep_top.sof

9. Once you are convinced you have a copy of all the important lab files on the N: drive you need to delete your 'LogicalStep' project folder and all its contents from the C: drive to prevent possible plagiarism.

When you get father into lab 1 (i.e., session 2 of lab 1) you will also want to add your software workspace (the 'Software' folder in your LogicalStep project folder) to the repository. When adding the software folder contents, do not add 'RemoteSystemsTempFiles' or the 'obj' to the repository to save space and increase commit speed.

Appendix E: Lab 1 Activity Checklist

☐ Launch Quartus Prime
☐ Setup SVN repository (Appendix D)
☐ Create a new project
☐ Make sure project name is “LogicalStep”
☐ Make sure top file is “LogicalStep_top”
☐ Click next till “Add Files”
☐ Downloads files from LEARN
☐ Add files that you downloaded (change filter to *.*) add LogicalStep_top.v and LogicalStep.tcl
☐ Click next till “Family and Device Settings”
☐ Choose Max10 “10M08SAE144C8G”
☐ Click “Finish”
☐ Select “Tools>Tcl Scripts..” select LogicalStep.tcl, click run (assigns pins)
☐ Select “Tools>Qsys”
☐ Save Qsys file, call it LogicalStep (make sure name is correct)
☐ In “System Contents”, “Description”, double click “Clock Source”
☐ Make sure Clock frequency is 50000000 (50Mhz)
The following shows the IP Block, Name and Exports for each IP block:
☐ Clock Source, clk_50, clk_in=”clk_50” clk_in_reset=”reset”
☐ Avalon ALTPLL, use default name, c0=”sdram_clk” c1=”audio_mclk”
ALTPLL parameters: inclk: 50Mhz, c0 phase shift: -1.5 ns, c1 freq out:12.28 Mhz
☐ Uncheck “Create ‘locked’ output
☐ NIOS II Processor, use default name, no exports
☐ SDRAM Controller, sdram_0, wire=”sdram_0” SDRAM Controller parameters: bits 16, Chip select 1, Banks 4, Row 12, Column 8
☐ System ID Peripheral, use default name, no exports
☐ JTAG UART, use default name, no exports
☐ Make connections in QSYS as shown in Figure 16
Add more cores:
☐ PIO (Parallel I/O),, led_pio,external_connection=”led_pio”
PIO parameters: 8 bits, output
☐ PIO (Parallel I/O),, button_pio, external connection=”button_pio”
PIO parameters: 4 bits, input, Synchronously capture: ANY, Generate IRQ: EDGE
☐ PIO (Parallel I/O),, switch_pio, external connection=”switch_pio”
PIO parameters: 8 bits, input
☐ Altera Avalon LCD 16207, lcd_display, external=”lcd_display”
☐ Audio and Video Config, audio_I2C_config, external_interface=”audio_i2c”
☐ Audio, Audio *(caps are important),* external_interface=”audio_out
☐ UART (RS-232 Serial Port), uart, external_connection=”uart”

☐ Make the connections in QSY as shown in Figure 21
Add more cores:
☐ Interval Timer, system_timer, no exports
☐ Interval Timer, use default name, no exports
☐ SPI Master (3 wire serial), spi_master, external=”spi_master”
☐ Dual 7 Segment, seven_seg_pio, dual_7_segment=”segment_drive”
☐ EGM, Egm, interface=”egm_interface”
☐ PIO (Parallel I/O), stimulus_in, external_connection=”stimulus_in”
PIO parameters: 1 bit, Synchronously capture: ANY, Generate IRQ: EDGE
☐ PIO (Parallel I/O), response_out, external_connection=”response_out”
☐ Make the final connections in QSYS as shown in Figure 24
☐ Auto assign base addresses
☐ Set Reset vector in NIOS processor as shown in figure 25
☐ Set Exception vector in NIOS processor as shown in figure 25
Finish rest of hardware project:
☐ Click “Generate HDL…” button
☐ Save
☐ Click “Generate” (if you get errors go back and correct them.)
☐ Click “Hierarchy” and change to “Files”
☐ Add “LogicalStep>Synthesis>LogicalStep.qip” to project
☐ Edit the “LogicalStep_top.v” file
☐ Copy text from “LogicalStep_inst.v” to the section in the top file below “Place Qsys instance below here”
☐ Edit the lines in the top file: IE .clk_50_clk (<connected-to-clk_50_clk>) to .clk_50_clk (clkin_50)
☐ Compile the project: “Processing>Start Compilation” (Ask the TAs or Lab Instructor if you cannot fix any errors generated by this step)
☐ Program the FPGA: “Tools>Programmer” (if you do not have a USB Blaster, consult the Lab Manual