University of Waterloo

Department of Electrical and Computer Engineering

WATERLOO
ENGINEERING

ECE 224 / MTE 325
Embedded Microprocessor Systems

Laboratory Manual

William Bishop, Carol Hulls, Wayne Loucks, Roger Sanderson

September 10, 2014

Over the past 15 years, numerous people have been involved in the development of this lab manual. The authors wish to acknowledge the significant contributions of Jeff Dungen, John Freeman, Rob Gorbet, Adam Neale, Amanda Pileggi, and Jason Shirtliff. Also, it should be noted the Microsoft Canada helped to fund updates to the laboratory studies in this course as part of the Microsoft Online Learning Initiatives in Electrical and Computer Engineering at the University of Waterloo.

Contents

List of Figures

List of Tables

Chapter 1

General Course Information

This section of the lab manual contains information pertaining to the course. While not specifically related to the lab, this is a convenient location to collect this material.

1.1 Marking Scheme, Course Schedule Information and Due Dates

The grading scheme, course schedule, and list of due dates can be found in the course syllabus. A copy of the course syllabus can be found on the University of Waterloo learning management system (LEARN).

1.2 Course Resources & Reference Material

Course Text

ECE 224 and MTE 325 differ in terms of the expected background of the students and in terms of the depth and breadth to which certain topics are covered. This fact combined with the highly specialized nature of the topics in this course make finding a suitable course textbook challenging.

There is no required course textbook for either of the courses. In the past, students have been referred to the out-of-print text *Microcomputer Structures*:

> Z. Vranesic & S. Zaky, *Microcomputer Structures*, Saunders College Publishing, QA76.5.V73 1989.

A copy of this text has been put on reserve in the Davis Centre Library.

Accessing Print Reserves

All print reserve material is listed on TRELLIS, the uWaterloo library catalogue, under the "Course Reserve" tab. Refer to the course notes for a brief description of the reserve material.

Online Resources

The following web sites and other online resources may be of interest to students in the course.

uWaterloo learning management system The course will be using the uWaterloo Desire2Learn course tools. Your feedback to improve the usage of these tools in this course would be appreciated.

`http://www.altera.com`
The labs use Altera's Nios II Embedded Processor Development Kit. The Altera website has a wealth of information on this kit and the components which make up the hardware and software students will use in the lab. Topics of interest on the Altera web site include (URLs begin with `http://www.altera.com/`):

- Cyclone II (`products/devices/cyclone2/cy2-index.jsp`)
- Nios II Processor (`products/ip/processors/nios2/ni2-index.html`)
- DE2 Development & Education Board (`literature/nv/06_q1/articles/technology_de2_dev_board.html`)
- Quartus II (`products/software/pld/products/q2/qts-index.html`)
- Literature (`literature/lit-index.html`)

Chapter 2

Laboratory Guide

2.1 Safety First

Safety should always be the first thing on your mind, hence this section comes first **"Safety First"**. The ECE 224 / MTE 325 lab rooms are relatively safe. We have no chemicals, high voltages, or other dangerous lab equipment. However, this does not mean that safety should not be on your mind. When you first enter the lab, make note of where the nearest Exits are, where the phone is, and where the safety information is posted. Keep your work area tidy, and make sure backpacks etc. are not in the aisles so as to prevent a tripping hazard. If you notice hazards like frayed power cords on the equipment, notify the lab staff. Safety is everyone's responsibility, please work safe.

2.2 Introduction

The laboratory component of ECE 224 / MTE 325 consists of a Lab Tools Tutorial and 2 labs. The goal of the tutorial and the labs is to provide students with hands-on experience with interfacing devices to a microprocessor. The labs make use of Altera's Development and Education (DE2) Board and the Nios II Embedded Processor Development System. Students implement embedded systems using an Altera Cyclone II programmable logic device mounted on a DE2 development board. Software tools allow the Cyclone II device to be configured with an assortment of hardware devices, including Altera's Nios II processor. The development board integrates several peripherals of different types (e.g., push buttons, LEDs, switches) for interfacing to the processor. The individual labs are described briefly below, and in more detail in the corresponding chapters found later in this manual.

Lab Tools Tutorial (LTT) The Lab Tools Tutorial provides an introduction to the Altera Nios II Embedded Processor Development System used in the course. It consists of two sessions: a "software tools" session and a "hardware" session. During the first session, students learn how to use the Nios II Embedded Processor Development Tools to design, compile, and simulate hardware for the Cyclone II device. **The processor built during this session is used as the basis for the computer used in the Lab Studies.** The second session, which is completed during the time scheduled for Lab 1, students learn how to configure the Cyclone II device to implement a custom hardware system and learn how to compile and execute software on the system.

Lab 1 Lab 1 introduces students to the design of software for the Nios II processor. You are provided with a hardware configuration for the Cyclone II device. This configuration will require you to augment the processor developed in the LTT. You write software to implement functionality on the development board.

The software uses several I/O synchronization techniques to respond to events generated by a hardware device while performing a background task on the processor.

Lab 2 Lab 2 allows you to apply the knowledge gained in Lab 1 to augment the LTT processor to produce a more involved System-On-a-Programmable-Chip (SOPC). In this lab, you interface an SD Card and an audio CODEC to the Nios II processor using the SD bus for the SD card and I^2C bus for the audio CODEC. You design and implement both the hardware and software required to implement a simple WAV file player.

2.3 Operational Specifics

2.3.1 Lab Hours

The lab (E2 2364) will be open for use most weekdays from 8:30am to 5:30pm. However, supervision will only be provided during scheduled lab periods. During a scheduled lab period, students who have not signed up for the scheduled lab period will be asked to leave the lab room.

2.3.2 Lab Groups

The first session of the Lab Tools Tutorial is done individually. The second session of the Lab Tools Tutorial is done in groups of two. All labs are to be done in groups of two. Groups of three or more students are strictly prohibited.

You will be assigned to a group based on your scheduled lab time as described in the course syllabus.

2.3.3 Use of Scheduled Lab Time

Most of the work you will be doing in the labs involves paper design and the use of software tools for hardware and software design. Hardware can be designed, compiled, and simulated[1] without access to the development board. Similarly, software can be designed, written, and compiled without the board. You are encouraged to do as much preparatory work as you can outside the lab so that you may make efficient use of your scheduled lab period.

Having said that, the full suite of software tools for the Altera Development and Education (DE2) Board and the Nios II Embedded Processor System are not available on the entire NEXUS network. They are installed only on the Windows 7/Nexus machines in E2-2356, E2-2363 and E2-2364. Students are strongly encouraged to use these machines outside official lab periods to prepare their designs and software ahead of their scheduled lab. This will allow you to optimize your use of the supervised time available in the lab. Students with computers at home may wish to download the free Quartus II Web Edition. It is available from
http://www.altera.com/products/software/download/altera_design/Quartus_we/dnl-quartus_we.jsp.
Make sure you get version 10.1, which is want we are currently using.

2.3.4 Collaboration

It is to be expected that members of the class will discuss various aspects of the labs, and we encourage you to collaborate within acceptable limits. Collaboration can be an excellent learning tool, but also leads some students

[1] In theory, you could simulate a complete system, including the Nios II processor. In practice, this is not likely to be feasible due to the limited capabilities of the computers in the labs. You are encouraged to simulate the individual hardware components of your designs independently.

to fall on the wrong side of uWaterloo Policy 71[2]. To avoid the pitfalls associated with collaboration, and help you to learn the material individually, we have the following two requirements:

1. Don't exchange files. This simple advice means that anything you use in the labs is your own creation.
2. If you discuss writing (e.g., software, lab reports) with other groups, do an unrelated task or activity prior to doing your own writing. If you still remember the ideas, you will have learned the material and the writing can be justified as being your own work.

You will be expected to include a signed declaration of authorship form for every submission of work. In the event that you have shared intellectual property with another group, your declaration of authorship must clearly state this fact including the details of the shared intellectual property and the extent of the sharing (including a list of all students involved). If the sharing is clearly documented, it is not subject to disciplinary action. Of course, you will not receive credit for the work of other students and if the work has been created collaboratively, individual marks may be reduced for everyone involved.

2.3.5 Deliverables

The Lab Tools Tutorial has no deliverables.

For each of Lab 1 and Lab 2 the deliverables consist of a demonstration, a report, authorship verification and submission of your code. Each of these are described below.

Demonstration

A demonstration of your working system (or as much of your submission as is completed) by your scheduled demonstration time. Demonstrations will be scheduled during your last scheduled lab session for each lab. Sign up for a demonstration slot early. If possible, reserve an early session so that groups requiring more time can use the later sessions. The details for signing up for a demonstration slot will be announced on the lab forum on uWaterloo Desire2Learn .

As you enter the lab room pick up a lab demonstration checklist sheet and complete your information on the sheet and hand to the the TA as the demo starts. The lab demonstration checklist sheet is also available in the appropriate lab folder in uWaterloo Desire2Learn should you wish to review it prior to coming to the demonstration.

At your demonstration, you will be asked by the TA to briefly outline your C code, (re)compile your code, and download the program to the board. The TA will then verify the functionality of your system. The TA will determine the operation of your embedded system and ask you questions about your design. You will also be required to show (and submit) your C/C++ code.

Once your demo is complete the TA will keep the lab demonstration checklist sheet. The C/C++ code will be submitted electronically at this time on uWaterloo Desire2Learn .

Report

A report, no more than 9 pages[3] in length on aspects of the lab. The details regarding the report are provided in the Lab 1 (Section 5.7) and Lab 2 (Section 6.9) sections of this manual. The submission will also include the following items.

[2] uWaterloo Policy 71 is the Student Academic Discipline Policy. The full text of this policy is available on-line at http://www.adm.uwaterloo.ca/infosec/Policies/policy71.htm.

[3] Page counts refer to the main body, and assume a reasonable font size (10 pt to 12 pt) and line spacing (1.5 to 2).

- The lab report, as described in each lab study (Chapters 5 and 6). The lab report is to be submitted to a drop box in uWaterloo Desire2Learn .
- Contribution and Reflection. Each of the reports submitted is to contain a paragraph (one-half page in length) from each member of the group describing their individual contribution to the work of the lab study as well as their personal reflection on the most important learning outcome from your contribution to the lab study.

Other Deliverables

The lab demonstration checklist sheet to be left with the TA when the lab demonstration is complete. The lab sign-off sheet is provided uWaterloo Desire2Learn in the section associated with each lab study. This sheet includes a declaration of authorship.

You may also be asked to submit VHDL code and C code to other drop boxes at the same time.

Chapter 3

The ALTERA Nios II Embedded Processor Development System

The Nios II Embedded Processor Development System consists of an Altera Development and Education (DE2) Board, a PC connected to the board via a JTAG[1] interface using a USB cable, and a suite of software tools running on the PC.

3.1 Altera Development and Education (DE2) Board

The heart of the Altera DE2 Board is the Cyclone II programmable logic device. The board is built around this chip, specifically to provide a flexible environment for development and testing of embedded system applications for the Cyclone II device. Typically, the development board is used in industry to prototype application systems. The prototype design can then be replicated by configuring thousands (perhaps millions) of Cyclone II devices and these devices are embedded into a final product. These products may include things such as automobiles, robots, toys, or the infamous "microprocessor-controlled toaster."

The Cyclone II device can be configured to implement various hardware devices including a processor, timers, parallel and serial ports, memory interfaces, and custom-designed hardware components. Custom-designed components can be specified using a hardware description language such as VHDL or Verilog.

In addition to the Cyclone II device, the board includes a fixed set of hardware peripherals: RAM, ROM, VGA interface, USB ports, push buttons, LEDs, switches, and a LCD. The board also supports a fixed set of peripheral interfaces: VGA interface, USB ports, serial port, SD card slot, and several other interfaces. The peripherals and peripheral interfaces are all hardwired to the Cyclone II device, and can therefore be used in your designs.

To communicate with the development board, the *JTAG interface* is used. The JTAG interface connects to the USB port on the development PC. The JTAG interface may be used to transfer hardware designs directly into the Cyclone II programmable device. This process is referred to as "hardware configuring," or simply "configuring" the device. The interface is also used for communication between a Nios II processor configured into the Cyclone II, and the development PC. It is used to download software to run on the Nios II processor. This process is referred to as "software programming," or simply "programming" the device. Finally, the interface is also used at program run-time to communicate text data between the PC and the Nios II processor.

This is only an overview of the development board. For more details, you are encouraged to read the documentation available on uWaterloo Desire2Learn .

[1] Joint Test Action Group

3.2 Development Software

The software tools which come with the Nios II Embedded Processor Development System, and which are installed on Nexus, are Quartus II, and the Nios II IDE. The role of each of these is briefly described below.

Quartus II Quartus II is the "top-level" tool for design for all families of Altera programmable logic devices. You will use Quartus II to build a Nios II embedded processor and interfaces to different devices. The Quartus II tool provides an environment for you to create, compile, and simulate your hardware system, and to program it into the Cyclone II device on the development board.

Nios II IDE This package provides a development environment in which you will run the tools used to compile and download code for the Nios II processor, and to communicate directly with the processor via the console. In this environment, the user has the option of testing directly on the board, or in a simulated environment using just the console window.

3.3 Designing for the Nios II Embedded Processor Development System

A system "design" created by the Nios II Embedded Processor Development System includes both hardware and software. Rather than writing software for an off-the-shelf processor, or running pre-compiled applications on a suitable custom processor, the Nios II Embedded Processor Development System allows you to design both hardware *and* software. The simultaneous design of both of these components for a single target and application is referred to as *co-design*.

The development PC is the primary design platform. It is used for three distinct tasks.

- Task 1: Hardware design, simulation, compilation, and programming.
- Task 2: Software coding, compilation, and downloading.
- Task 3: Interfacing with a running Nios II processor via a USB connection.

This list of tasks assumes you have taken an overall specification for the system to be built, and you have decided how to split the required functionality between hardware and software. Of course, there are many ways to do this for a given system.

Task 1: Hardware

The steps involved in designing a hardware system for implementation in the Cyclone II device for a particular application are listed below. This list illustrates the steps typically used in the labs in this course and does not attempt to illustrate all possible steps. Of course, engineering design is an iterative process so it may be necessary to repeat several steps prior to producing a suitable system design.

Step 1. Functional Specification: Given the overall functionality of the device being created, determine the function(s) that must be provided by the hardware.

Step 2. Design Specification: Given the required hardware functionality, determine the component(s) required by the system.

Step 3. Component Selection: Determine if any components in the available the library of hardware components would be suitable for use in the system. If suitable components are available, select and use them in your system design.

Step 4. Component Creation: Design any custom components required by the system. You will need to go through an entire design cycle for each of the custom hardware components in your system.

Using Quartus II, you can include custom hardware components at various levels and in various ways. You can wire gate-level components together and group them as a functional block. You can translate a functional specification to VHDL and include the resulting description in your design. You can also download third-party intellectual property cores (which Altera refers to as *MegaFunctions*[2]) and include them in your design[3]. For the purpose of the labs in this course, you will build custom components in VHDL that may make use of gate-level components.

This step generally will include the simulation and re-design of the custom hardware components, as necessary.

Step 5. Design Implementation: Using Quartus II, instantiate and wire the components of your hardware system. Specify the system settings, the target device for the system, and the pinouts of the system design.

Step 6. Compilation: Convert the specified system into a file suitable for configuration of the target device.

Step 7. Configuration: Configure the Cyclone II device with the custom hardware system via the JTAG interface.

Step 8. Test & Debug: Test the hardware, debug, and redesign as necessary. Testing will often require the development of custom software.

Task 2: Software

Writing software to run on a Nios II processor system is similar to cross-platform software development for other computing platforms.

Step 1. Functional Specification: Given the overall functionality of the software being created, determine the function(s) that must be provided by the software.

Step 2. Design Specification: Given the required software functionality, determine the design of the required software.

Step 3. Implementation: Construct software for the system using the tools provided for the Nios II processor. The tools support software development in assembler, C, or C++. In this course, you will not be required to write assembly code or C++ code. You may write your C code in any text editor, although the editor in Quartus II provides context-sensitive colour highlighting. **NOTE: The C++ language is not fully supported so it is not used in this course.**

Step 4. Compilation: Compile your code using the `Build` command in the Nios II IDE.

Step 5. Download & Execute: Execute the software on the Nios II processor using the `Run As Nios II Hardware` command in the Nios II IDE. Programs can also be simulated using the instruction set simulator (ISS) from the Nios II IDE.

Step 6. Test & Debug: Test the software, debug, and re-design as necessary.

[2] http://www.altera.com/products/ip/ipm-index.html

[3] Of course, any third-party components that you include in your lab solutions must be properly referenced.

Task 3: Communications

In addition to configuring the hardware and programming software, the development PC can act as a dumb terminal for communication with the Nios II processor. To enable this functionality, you must include a UART in your system. This is very useful for displaying program output as well as testing and debugging software.

3.4 Other Resources

Further information on the Nios II Embedded Processor Development System is available at the Altera web site (http://www.altera.com) and in the documentation provided on uWaterloo Desire2Learn .

Chapter 4

Lab Tools Tutorial

4.1 Purpose

The purpose of Lab Tools Tutorial is to provide you with an opportunity to familiarize yourself with the tools used when designing systems for the Altera Development and Education (DE2) Board.

4.2 Tutorial Outline

The Lab Tools Tutorial is split into two sessions. The first session is held in the multimedia lab (CPH 1346) outside of regularly scheduled lab and tutorial hours. The second session is held in the course lab room (E2 2364) in one of the regularly scheduled lab time slots. The sessions are described briefly below, and in more detail in the following sections.

Session 1: This session introduces you, interactively, to the use of the Quartus II development environment. The session consists of two parts. In the first part (Section 4.4) of this session, you work individually to compile a VHDL design for a full adder, using Quartus II. This gives you some exposure to how Quartus II can be used to build a simple hardware design. In the second part (Section 4.5) of this session, you work individually to complete a slightly modified version of Altera's Nios II Tutorial (refer to Appendix A) that defines a Nios II processor system and compiles the design. The result of the compilation is saved for use in the 2nd tutorial session.

This session is scheduled very early in the term. Refer to the course syllabus on the uWaterloo Desire2Learn site to determine the date and time of this session of the Lab Tools Tutorial. Students are encourage to attend this session to obtain the background knowledge necessary to complete the labs in this course.

Session 2: In this session (completed during the first part of the first scheduled lab period), you work in your group to learn to configure the hardware on the Altera Development and Education (DE2) Board, compile software for a Nios II processor system, and execute software on the DE2 Board. This session completes the modified version of the Altera Nios II Tutorial described in Appendix A. You begin by configuring the Cyclone II device with one of your saved hardware designs from the 1st tutorial session, and use the features of the design to verify that the download has been successful. You then compile a simple board diagnostics application and run it on the Nios II processor.

4.3 Preparation

In preparation for this lab, read through Chapters 3 and 4 of this manual, as well as the Nios II Tutorial in Appendix A.

4.4 Part I: Development of a Full Adder

In Part I of this lab, you use Altera Quartus II to build a full adder design. The full adder is a 1-bit addition circuit that is a common building block for the construction of larger, more complex adders. The full adder takes 3 input signals (*a*, *b*, and *cin*) and produces 2 output signals (*sum* and *cout*) as shown in Figure 4.1. The signals *a* and *b* represent the data input bits to the adder. Signal *cin* represents the carry input bit. The output signal *sum* is asserted high if either 1 or 3 of the input bits are asserted high. The *sum* is the data output bit of the adder. The output signal *cout* is asserted high if two or more input bits are asserted high. The *cout* signal is the carry output bit. The *cout* signal can be used to implement a ripple carry chain when connecting several full adders together.

Full Adder Truth Table

Inputs			Outputs	
a	b	cin	sum	cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Full Adder Symbol

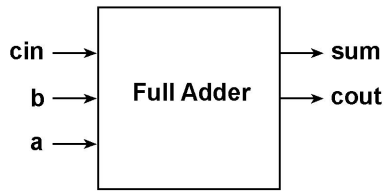

Figure 4.1: Full Adder Functionality

Figure 4.2 demonstrates how four full adders could be used to construct a 4-bit ripple-carry adder. Similarly, you could connect four 4-bit ripple-carry adders to form a 16-bit ripple-carry adder. In this lab, you will not be required to develop a ripple-carry adder due to time constraints.

The hardware development process is a complex, iterative process. The development of a hardware design for a Programmable Logic Device (PLD) can be broken down into the following stages of development:

1. **Design**
   The design of hardware components can be complicated. When designing a new hardware component, you should ask yourself the following questions (and possibly many more...):
   - What signals will the hardware component require?
   - Should the hardware design use sequential logic, combinational logic, or a combination of both?
   - How can I best describe the operation of the hardware component?
   - Is it easier to describe the hardware component using one design file or a hierarchy of design files?

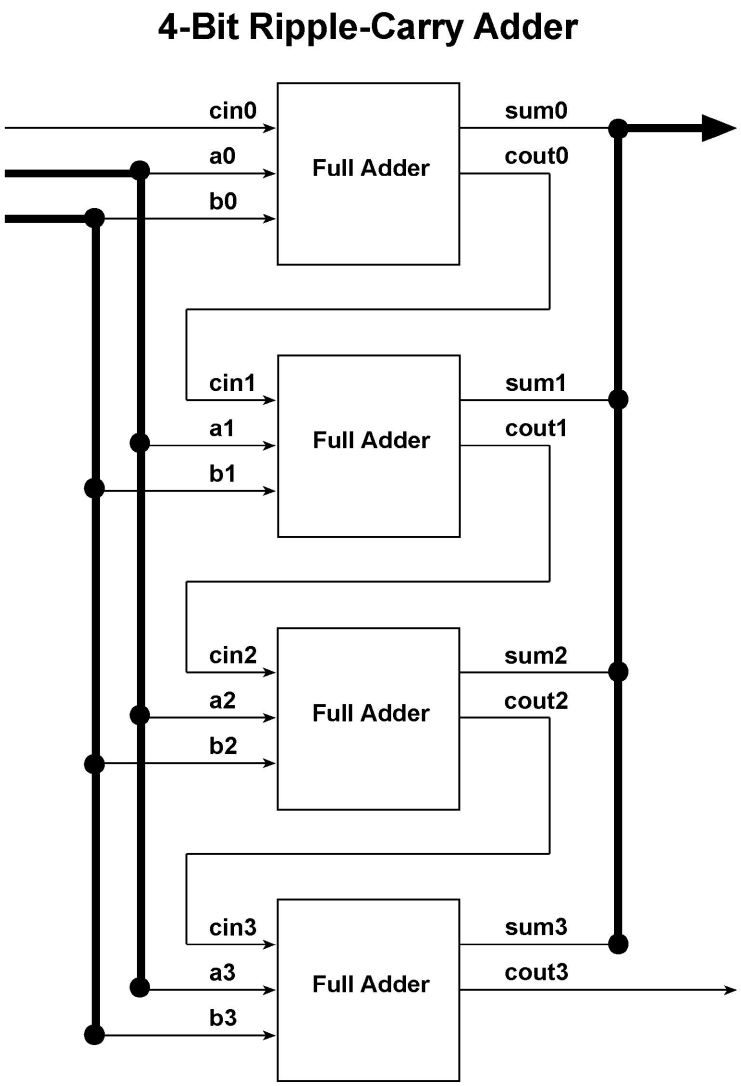

Figure 4.2: 4-Bit Ripple-Carry Adder

- Can I reuse hardware components from a library or a previous design?
- What design entry method is best suited to the task of describing the hardware design?

2. **Compilation**
   (a) Design Entry: This involves the creation of a suitable description of the hardware design using a combination of textual and graphical techniques.
   (b) Elaboration: This involves the syntax checking of a design and all of its hardware components. This stage also involves checking that all hardware components used by the design have been connected properly.
   (c) Logic Synthesis: This involves the conversion of a hardware design into an intermediate netlist format suitable for processing by subsequent tools. An important part of this stage is the mapping of hardware design elements to PLD technologies.
   (d) Placement and Routing: This involves the physical assignment of the hardware design to specific resources within a PLD. At the end of this stage of development, the tool creates a description of the hardware design suitable for PLD configuration.
   (e) Timing Analysis: This involves an analysis of the performance of a hardware design. This is a critical step when working with modern PLDs and high-performance circuits. A design that fails to achieve specified performance constraints will not function correctly.
3. **Testing**
   (a) Configuration: This involves the configuration of a PLD with the programming files produced by the tool used for placement and routing.
   (b) Verification: This involves the rigorous testing of the configured PLD.

Figure 4.3 illustrates the stages of hardware development for a PLD. Although iteration is not shown in the figure, the process relies heavily on iteration. At any stage in the development of a hardware design, it may be necessary to return to a previous stage to correct a design flow.

Step I: Project Creation

The first step in using Quartus II for hardware development involves the creation of a project directory. To create a new project, perform the following steps:

1. Close any unnecessary programs. The Quartus II software uses memory resources extensively. It will run faster if you have fewer programs active.
2. Start Quartus II. The software should not take more than a minute to load. If you are asked whether you would like to create a new project, select "No".
3. Select "File / New Project Wizard..."
4. The window [Page 1 of 5] should now allow you to specify the project directory, the name of the project, and the name of the top-level design entity.
5. In the first box, enter `N:\ece224\tutorial\full_adder` as the directory to be used by Quartus II for your project. If the directory does not exist, it will be created.
   NOTE: The tools do not fully support spaces in directories or filenames so you must not place your designs in `N:\My Documents`*!*

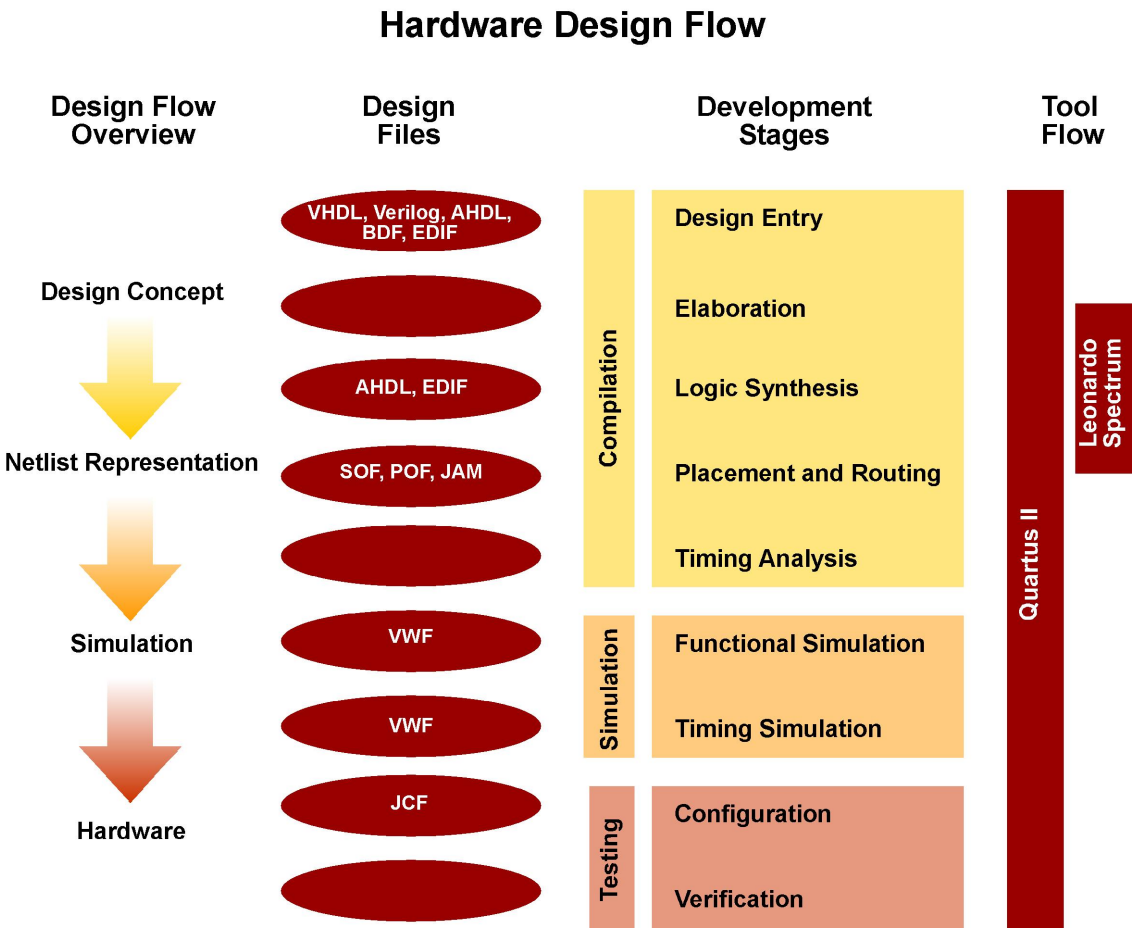

Figure 4.3: Hardware Design Flow for Programmable Logic Devices

6. Enter the name of the project as "full_adder". The name of the top-level design entity may automatically change to "full_adder". If it does not, modify it manually so that it reads "full_adder".
7. Verify that there are no spaces in either the project directory or the project name. Including spaces in the path name will lead to problems with the SOPC Builder later on. Click "Next".
8. There are a number of other pages to the New Project Wizard, which allow an advanced designer to specify project settings from the outset:

   [Page 2 of 5] Files can be added to the project. For this project, nothing needs to be added so click "Next".

   [Page 3 of 5] Select: Cyclone II, FBGA, 672 pins, speed grade 6 to filter the list of available devices. Specify EP2C35F672C6 as the target device, and click "Next".

   [Page 4 of 5] All boxes should be unchecked, then click "Next".

   [Page 5 of 5] Verify that the summary reflects the correct settings.
9. Click "Finish". Congratulations, you have now created your first Quartus II Project.

Step II: Design Entry

Now, we will depart from using Quartus II momentarily to copy the design file from uWaterloo Desire2Learn to your project directory. If you have not already done so, then without closing Quartus II, copy the file named `full_adder.vhd` to your new project directory, `N:\ece324\tutorial\full_adder`. The source design file is located on the uWaterloo Desire2Learn website for the course. This design file is shown in Figure 4.4.

VHDL is a language for the description of hardware designs. The acronym stands for VHSIC (Very High-Speed Integrated Circuit) Hardware Description Language. The VHDL design for the full adder includes several comments that help to document the design. The full adder is built using two Boolean equations. The first Boolean equation describes the *sum* output and the second Boolean equation describes the *cout* output. These equations are shown below:

$$sum = a \text{ XOR } b \text{ XOR } cin \tag{4.1}$$

$$cout = (a \text{ AND } b) \text{ OR } (a \text{ AND } cin) \text{ OR } (b \text{ AND } cin) \tag{4.2}$$

If you do not understand the VHDL code for the full adder right now, don't worry. A full understanding of the VHDL code is not necessary to complete this tutorial. However, it is highly recommended that you review the on-line VHDL Tutorial early this term. A link to an online VHDL Tutorial is provided on the uWaterloo Desire2Learn website for the course.

Now, it is time to add the design file to the project. Simply copying the file to the working directory does not add the file to the project.

Perform the following steps to add the design file to the project:

1. Ensure that the Quartus II window is maximized.
2. Select "File / Open..." A window should appear that shows design files in your new project directory. If you have successfully copied the design file to your project directory, you should see a file named "full_adder.vhd"
3. Select "full_adder.vhd" and select the check box for "Add file to current project".

```
1  -- ECE 324 - Lab 0
2  -- VHDL Design of a Full Adder
3  --
4  -- William D. Bishop
5  -- Department of Electrical and Computer Engineering
6  -- University of Waterloo, Waterloo, Ontario, CANADA
7  -- N2L 3G1
8
9  -- The following lines include the definitions necessary to use
10 -- the STD_LOGIC and STD_LOGIC_VECTOR types that are commonly used
11 -- in VHDL.  The IEEE library also defines the operation of
12 -- the XOR, OR, and AND functions used in this design.
13 LIBRARY ieee;
14 USE ieee.std_logic_1164.all;
15
16 -- The entity declaration defines the INTERFACE to the design.  It
17 -- specifies the PORTS (pins) and the GENERICS (parameters) to
18 -- the design.  A full adder design has five pins as shown below.
19 -- This design does not require any GENERICS.
20 ENTITY full_adder IS
21
22     PORT
23     (
24         a       : IN    STD_LOGIC;
25         b       : IN    STD_LOGIC;
26         cin     : IN    STD_LOGIC;
27         sum     : OUT   STD_LOGIC;
28         cout    : OUT   STD_LOGIC
29     );
30
31 END full_adder;
32
33 -- The architecture declaration defines the IMPLEMENTATION of the
34 -- design.  You can have multiple architecture declarations for
35 -- complex VHDL designs.  For example, you might have an
36 -- architecture optimized for performance and another optimized
37 -- for efficient device utilization.
38 ARCHITECTURE REV001 OF full_adder IS
39
40 BEGIN
41
42     -- These two equations specify the functionality of a full adder
43     -- using combinational logic.  These statements are concurrent
44     -- so the order in which they appear is irrelevant.
45     sum <= a XOR b XOR cin;
46     cout <= (a AND b) OR (a AND cin) OR (b AND cin);
47
48 END REV001;
```

Figure 4.4: VHDL Design of a Full Adder

4. Click "Open" This causes Quartus II to open the design file and adds the file to the current project. You can verify that the file has been added to the project by selecting the "Files" tab at the bottom of the Project Navigator window in Quartus II. The default location for this window is on the left side of the screen. The file should be listed in the "Device Design Files" folder.

Compilation

Now, you are ready to perform elaboration, logic synthesis, placement and routing. These stages are performed by Quartus II using these steps:

1. Select "Processing / Start Compilation" Quartus II performs a full compilation of your design. Quartus II automatically performs elaboration and may perform a timing analysis, depending upon your compiler settings. For the full adder design, this step should not take more than one minute. For the larger designs in this course, compilation may take up to 10 minutes to complete. A pop-up window appears to indicate that "Full compilation was successful". A warning message is displayed that indicates there are output pins without an output pin load capacitance assignment. This is normal and can be ignored because physical output pins have not yet been assigned.
2. Click "OK". This closes the pop-up window and allows you to view the compilation report.
3. Open the "Analysis & Synthesis" folder in the compilation report and select "Messages". If any warnings or errors occur, they are listed. It is very important that you pay close attention to any warnings issued. Warnings produced during hardware compilation cannot be ignored. They often indicate a serious problem within your design.
4. Explore the rest of the compilation report and become familiar with the types of information presented. Since you will not configure a DE2 Board to implement the full adder design, the compilation target is immaterial. However, for all designs that will be used to configure a DE2 Board in the lab, the target device *must* be set to EP2C35F672C6 prior to compilation. You should always verify that the compilation report indicates the correct target device prior to proceeding further.
5. Examine the **Timing Analyzer** folder of the compilation report. You should notice several sections, including "Settings", "Summary" and "tpd". The "Settings" section lists any timing settings that you made prior to the compilation of the design. The "tpd" section lists some of the delay paths in your hardware design (t_{pd} represents the "propagation delay time").
6. Close the compiler report before continuing with the Simulation section.

Simulation and Verification

At this point, you would normally use Quartus II to simulate your hardware design, make corrections (if necessary), configure a device, and finally verify the correctness of your design. Since you do not have a simulation tool or one of the Nios II Embedded Processor Development Boards at your workstation, you will not be able to perform these stages of hardware development. You will have an opportunity to configure a Cyclone II device with a design and then run software to test the design in a future lab session.

Archiving the Project

Quartus II provides the ability to archive a project in order to save disk space or transfer an entire project (more) easily from one computer to another. This will be important in the networked environment, with limited disk space, in which the course is run. This step will be critical for larger projects when they are compiled in the local

C:\Temp directory for speed purposes and need to be saved on your own N:\ directory when you are finished. To archive a project, follow these steps:

1. Choose "Project / Archive Project"
2. Click "OK"
3. Quartus II will inform you of the successful creation of the archive file. Click "OK".
4. Close Quartus II
5. The files which constitute the archive are "full_adder.qar" and "full_adder.qarlog" Move these files to your `N:\ece224\tutorial` directory.
6. You may now safely delete the entire "full_adder" directory.

The archive may be restored by starting Quartus II and choosing "Project / Restore Archived Project".

4.5 Part II: Building a Nios II System Module

In the second part of the first session of the Lab Tools Tutorial, students walk through the Altera Nios II Tutorial. An updated version of the Altera Nios II Tutorial is used. This version is designed for the versions of Quartus II and Nios II currently in use at uWaterloo and it has been adapted to the uWaterloo computing environment.

The tutorial (located in Appendix A of this manual) leads you through the creation of a custom Nios II system module with a Nios II processor and various internal and external peripherals (RAM, hardware timer) and interfaces (parallel and serial ports). A block diagram of the Nios II system module in the Tutorial is shown in Figure 4.5

Synthesizable hardware description files of each of the blocks in Figure 4.5 are provided in the Nios II HDK and are accessible to Quartus II. You will use the MegaFunction Wizard and SOPC Builder to configure each of these blocks and include them in your design. The last step in the SOPC Builder will be to Generate the Nios II system module. In addition, Quartus II can create a symbol for your custom Nios II system module which you can use to represent it in your design (this will look something like Figure 4.6).

Use Quartus II to wire the inputs and outputs of this Nios II system module to physical pins on the Cyclone II device, and hence to physical devices on the Altera Development and Education (DE2) Board (see Figure 4.7). Compile the design into a hardware configuration file for the Cyclone II device to conclude the first session of the Lab Tools Tutorial.

4.6 Part III: Configuring and Testing the Altera DE2 Board

In the 2nd session of the Lab Tools Tutorial, you complete the Nios II Tutorial by configuring a Cyclone II device on a DE2 development board with your hardware design, compile a simple application for board diagnostics, and execute the program on the system.

If you complete the Tutorial before the end of the scheduled time slot, you should attempt one or more of the following:

- Modify the `board_diag.c` program (you may use your imagination, but if you don't have any, try to make it count in the opposite direction),
- Explore the Quartus II Tutorial (in Quartus II, choose Help/Tutorial),
- Begin building the hardware for Lab 1 (see the section on Lab 1 for more details).

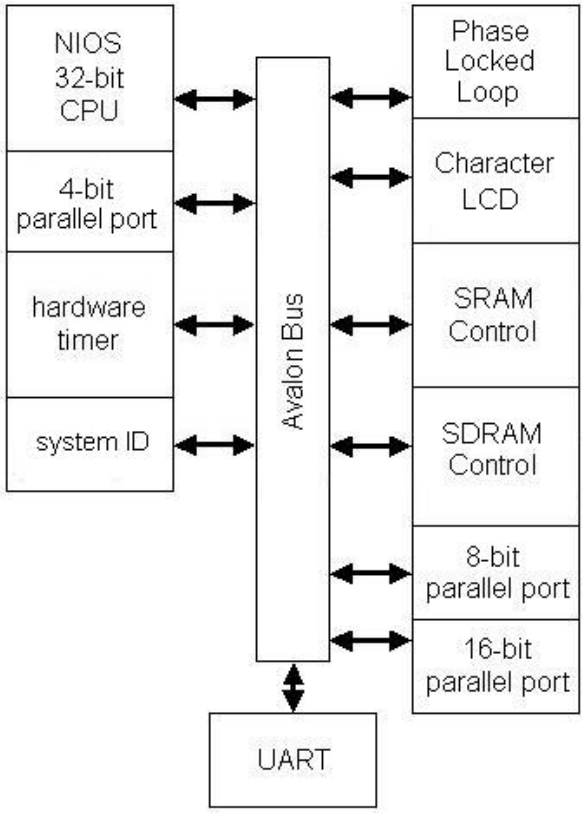

Figure 4.5: The Nios II system module block diagram. Each block represents a component of the complete system module. External connections have been left off. Blocks are arranged so that their locations correspond for the most part with the locations of the corresponding pins on the Quartus II-generated symbol (see Figure 4.6).

Figure 4.6: The Nios II System Module Symbol

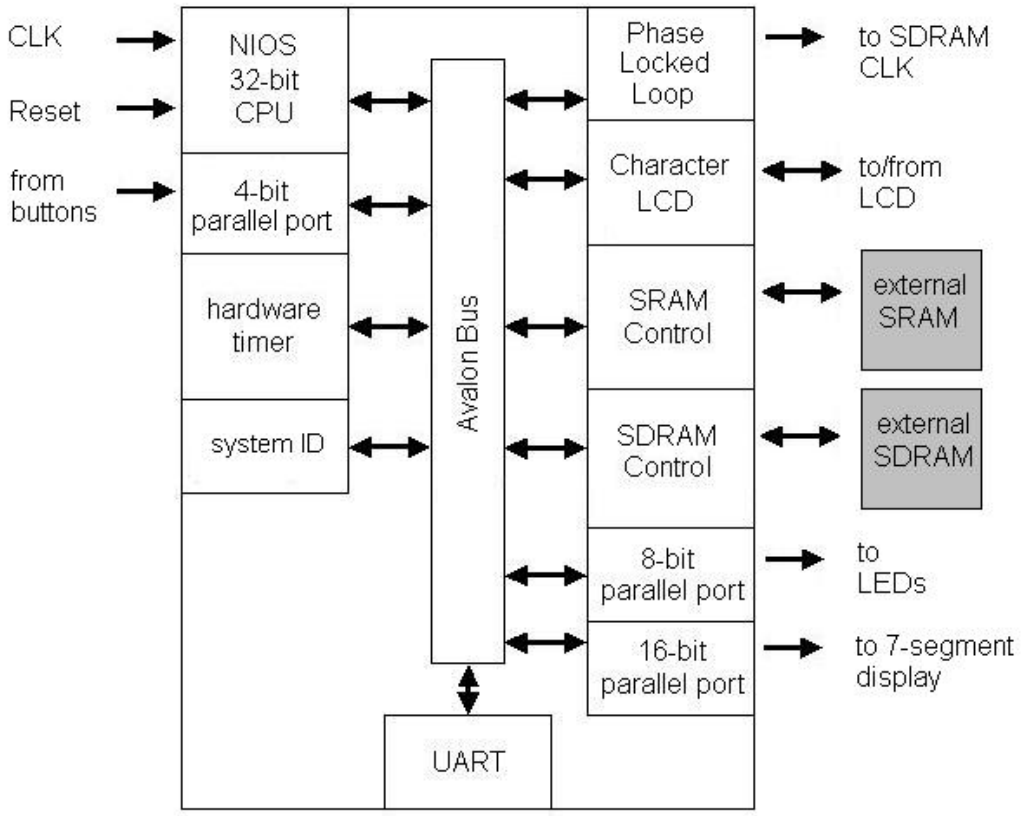

Figure 4.7: The complete Nios II Tutorial system. The portion in the box represents the Nios II system module; the memories are external to the system module.

Chapter 5

Lab 1

In this lab study you are to start from your lab tools tutorial (LTT) design and modify it to for this lab study. To help with this the following steps are recommended to capture your LTT design and to start from a known point with this lab study.

- Copy your project from lab tools tutorial using the command from the menu "Project / Copy Project". Select the proper destination where you will copy the new project and name it "lab_1". Verify the open new project box is checked and accept the creation from the new directory file.
- Open "myEsystem.bdf" and double click on your microcontroller symbol to open the SOPC builder and edit the components in the project.

5.1 Purpose

The purpose of Lab 1 is twofold:

1. To further familiarize students with Quartus II and the other software tools used with the Altera DE2 Board, and
2. To have students write software using polling loops and interrupt handlers for the purpose of interfacing with devices.

In the first phase of this lab, you construct C application software to interact directly with low level devices using interrupt synchronization techniques and polling synchronization techniques. Your software will display a sequence of bits on the board, one bit at a time.

In the second phase of this lab, you construct C application software to examine the impact of various synchronization techniques on the operation of the system as a whole. The system attempts to make progress on a background (software) task while monitoring and responding to external stimulus. The impact of your design decisions on these two goals is measured in terms of the speed of response (latency) to an external stimulus, task completion, reliability and other performance indicators.

The goal of the second phase is to get you to think about how to design an experiment to characterize the performance of a system with foreground and background tasks. This does not mean that you need to have the fastest possible code, only that you need to demonstrate that you know how to compare the performance of several solutions. In the past, some students have been tempted to "optimize" their design in ways that would not

normally be possible. You should start the problem on the assumption that you do not know how long execution times are in the system.

Upon completion of this lab, students will have had an opportunity to do the following:

1. Use the computer designed in the Lab Tools Tutorial (**myESystem**) to complete Phase 1.
2. Augment (modify) that computer (**myESystem**) to complete Phase 2 by adding a pre-specified Event Generation Module (egm) using parallel I/O ports as specified later in this lab specification.
3. The modified design is to be compiled and the Cyclone II device configured with that design (as described in the Lab Tools tutorial Appendix A, and
4. Write, compile, download, test, and debug C application software to accomplish the following tasks:
   - Read and write to a parallel port,
   - Use polling loops to monitor the status of a device,
   - Handle multiple interrupts from different sources, and
   - Use hardware timers.

5.2 Lab Outline

In this lab you will use an embedded system that provides a platform for you to develop your skills with the Nios II system as well as to examine the issues associated with some of the synchronization techniques discussed in class. The system to be implemented provides access to push buttons, DIP switches and timers for basic interfacing as well as access to an instrumented background task to be performed while your control program is monitoring a parallel port for an event. In this case, the events are generated by the EGM. You will use the various techniques available (tight polling, interrupts, and others) to monitor and respond to these events. The goal of this lab is to examine the advantages and disadvantages of various synchronization techniques and the trade-off between performance and response time.

Figure 5.1 shows a block diagram of the Nios II system module and the EGM, and the connections from the embedded system on the Cyclone II device to the LEDs, push buttons, and other external devices. You are provided with library components for the EGM. This component is also available as VHDL (VHSIC Hardware Description Language) designs on uWaterloo Desire2Learn for your examination and use. Figure 5.2 illustrates the output ports that control the EGM.

In Phase 1 of the lab, you reuse the hardware system from the Lab Tools Tutorial (**myESystem**).

In Phase 2 of the lab, you modify the hardware system as follows: This includes the following steps:

1. Use the System-On-a-Programmable-Chip (SOPC) Builder in Quartus II to enhance the Nios II system module (**myESystem**) built in the Lab Tools Tutorial according to the specification provided in Section 5.4.2 and insert it into a schematic design of the system,
2. Insert an EGM component into the schematic design and connect it to the Nios II system module,
3. Complete the schematic design in Quartus II, according to the specifications provided in Section 5.4.2, and
4. Compile the schematic design and configure the Cyclone II device on the Altera DE2 Board.

The software is designed in two distinct phases:

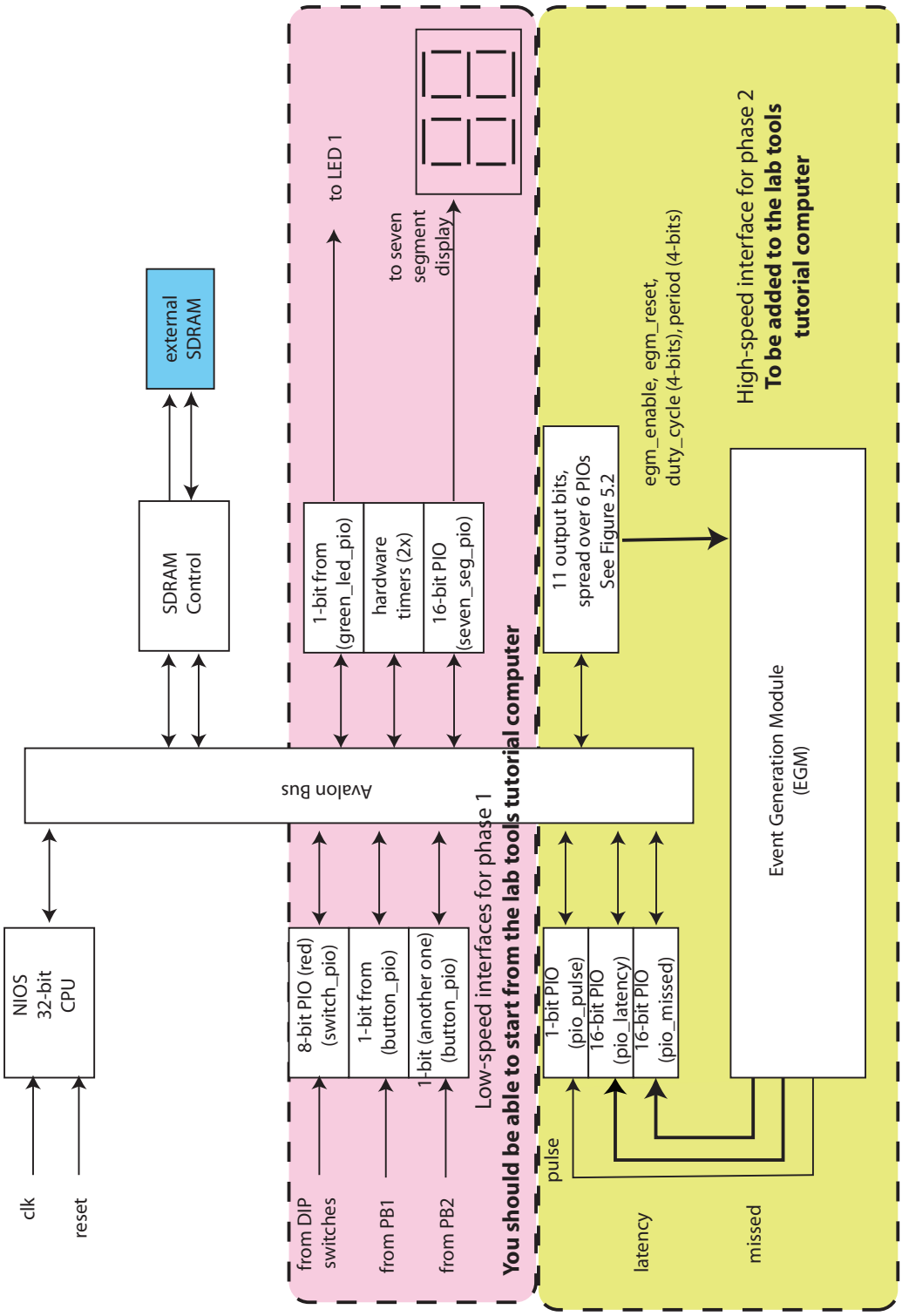

Figure 5.1: Block Diagram of Lab 1 Hardware System

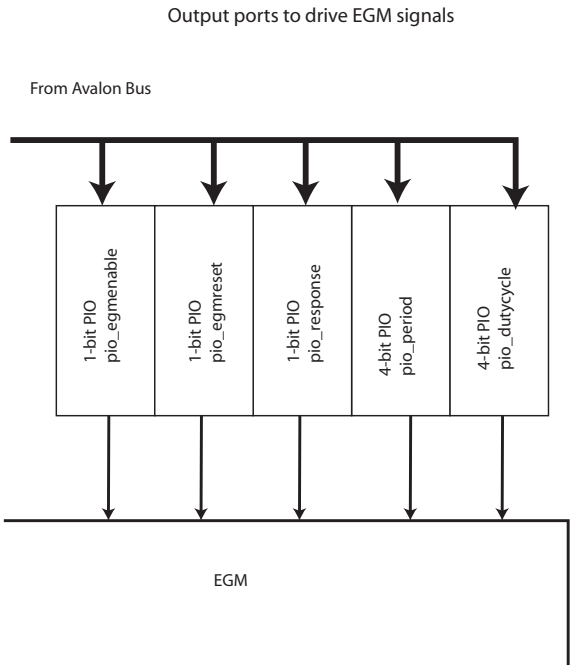

Figure 5.2: Block Diagram of output PIO ports to control EGM

Phase 1: In this phase your application software interacts with push buttons, DIP switches, LEDs and timers at a very low-level. The interfacing should be based on human input times and constraints for pushing buttons and reading displays. Some of the issues that you may have to consider in your design are: the issues associated with debouncing, the details of the specifications provided by the manufacturer, and the issues associated with interrupts from multiple sources.

Phase 2: In this phase you are provided with source code to implement a background task and you are asked to build application software to examine the response to a device (the EGM built into the design in Part 1) while performing the background task. Your application software should be capable of using two different approaches for synchronization with the device. Using your software, you design an experiment to examine the synchronization latency and the progress of the background task given a set of timing parameters. This particular phase requires the following steps:

1. Decide, given the components in the Nios II system, how to implement the required software functionality as specified in Section 5.5.2,
2. Write C application software to perform the background task and service the EGM using the functions provided, and
3. Compile, download, test, and debug the software on the Nios II system.

5.3 Preparation

Prior to starting this lab, you should be familiar with the Quartus II software. If you have not done so already, complete Lab Tools Tutorial prior to attempting Lab 1. You may also wish to explore the Quartus II Tutorial, which can be accessed using the *Help/Tutorial* menu in Quartus II. *NOTE: The Quartus II Tutorial explores the use of Quartus II and provides tips on how to best use the tool. This should not be confused with the Nios II Tutorial in Appendix A which describes how to build a Nios II Embedded Processor System.*

It is also highly recommended that you review the documentation on the course website. Pay particular attention to the following topics:

- Timer usage in the Nios II Timer Manual,
- Parallel I/O interfacing in the Nios II PIO Manual,
- Interrupt handling in the Nios II Software Developer's Handbook, and
- LEDs and push button usage in the Nios II Processor Reference Handbook.

Download (from the uWaterloo Desire2Learn course site) the files listed below into the directory used to create the **myESystem** project during the Lab Tool Tutorial. **(Before doing this you would be wise to make a backup of this directory and its sub-directories.)**

- the files include both: hardware and software files and all start with `ece324_egm.`
- `ece324_egm.c`. the source code file for the background task required for Phase 2 of the lab and
- `ece324_egm.` with various extensions for the hardware support. Note: there is a `.txt` extension included to make the download work, you will need to remove that extension once the file has been downloaded.

5.4 Part I: Hardware

This section of the lab describes the hardware used in this lab. The hardware for Phase 1 can be exactly the **myESystem** prepared during the Lab Tools Tutorial. You require augmented hardware for Phase 2.

The material in this Section focuses on the Phase 2 hardware and provides some reminders for the hardware creation process. however the steps to add hardware, configure and compile your design is the same as that described in Appendix A starting in Section A.2.2 except that this time when you start Quartus you will be opening an existing project (**myESystem**).

5.4.1 Phase 1 Hardware

As indicated above you can start immediately programming your solution to the Phase 1 problem described in Section 5.5.2.

5.4.2 Phase 2 Hardware

The Phase 2 hardware requires you to re-open your **myESystem** hardware design and to add several PIOs, then you need to connect (and reconnect) the pins, compile the result before you can use the hardware. This process is the same as the one described in Appendix A starting in Section A.2.2 except that this time when you start Quartus you will be opening an existing project (**myESystem**).

Generating the Nios II System Module

Use the SOPC Builder to augment the **my_controller** system module (in **myESystem**) with the specifications and peripherals shown in Table 5.1.

Table 5.1: Properties for Lab 1 Additional Nios II Peripherals. For the software provided in ece324_egm.c to work, the EGM PIO names shown here must be used for the EGM interfaces.

Peripheral	Name	Properties
Interfaces to EGM		
PIO	pio_pulse	1 bit, input only, synchronous, any-edge, edge-triggered IRQ
PIO	pio_latency	16 bits, input only, asynchronous, no IRQ
PIO	pio_missed	16 bits, input only, asynchronous, no IRQ
PIO	pio_dutycycle	4 bits, output only
PIO	pio_period	4 bits, output only
PIO	pio_response	1 bit, output only
PIO	pio_egmenable	1 bit, output only
PIO	pio_egmreset	1 bit, output only

It is recommended that you configure your peripherals with the settings shown in Table 5.1. However, you may modify your design as you see fit. Not all of the peripherals shown in Table 5.1 are required to successfully complete the lab. Some of the peripherals are provided simply to aid you with debugging your design.

Once you have added the peripherals you will need to take all the steps to generate the resulting system illustrated in the lab tools tutorial.

Completing the Hardware Design

Now, you can build your lab 1 design file. to do this open your previous hardware design and insert the symbols found on uWaterloo Desire2Learn for the EGM to your schematic. It is recommended that you place the EGM symbol to the left of the **my_controller** system module as shown in Figure 5.3.

Finally, add the signals, buses, and logical pins as illustrated in Figure 5.3. You should only have to add the signals for the new peripherals added, the other names should be correct from the Lab Tools Tutorial. Refer to Appendix A if you are not sure how to perform this task. *Be sure to label your logical pins EXACTLY as indicated in Figure 5.3. These names will be assumed by the* `.csv` *file that is used to map logical pin names to physical pin locations.*

5.4.3 Compiling the Design

Now that you have a complete design at the logical level, you must translate it to the physical level, compile it, and configure the Cyclone II device on the Altera DE2 Board.

Be sure to reload the pin names from the `.csv` file as described in Appendix A.

When all of the previous steps have been performed, set the compilation focus to `myESystem.bdf` and compile the design.

5.4.4 Configuring the Cyclone II Device

Once the compilation is complete, you may configure the Cyclone II device. Open the programmer (*Tools/Programmer*) and proceed as described in Appendix A.

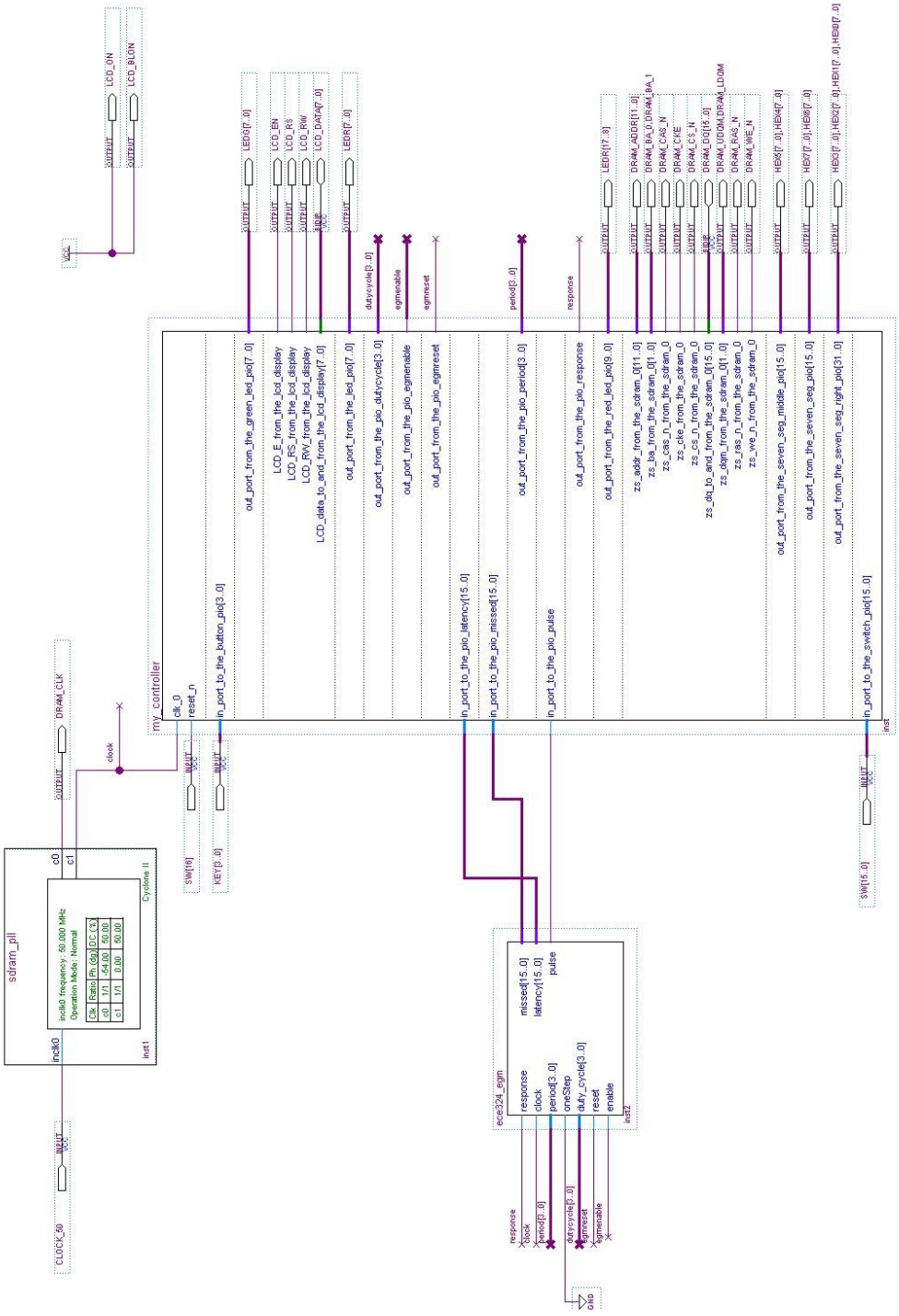

Figure 5.3: Schematic design file for Lab 1 not really

Once the Cyclone II device has been successfully configured with your hardware system, you may proceed to Section 5.5.

5.5 Part II: Software

In Part II, you will write software for your custom System-On-a-Programmable-Chip (SOPC) to implement the functionality described in Section 5.5.2. The Nios II embedded system you have implemented on the Cyclone II device has the hardware necessary for the lab, but there may be more than one way to use the hardware to achieve a successful design. For example, in this lab you will implement different synchronization methods to compare different port-monitoring techniques. There is no single "correct" solution. Part of the experience of this lab (and indeed, the entire course) is to investigate other ways of doing things, and to rise to the challenge of making a better design.

Once the software is written and successfully compiled, you will download it to the RAM on the Altera DE2 Board and run it on the Nios II processor. You should have the initial version of your software designed and coded before you come to your first scheduled lab period.

5.5.1 Getting Ready

The Nios II IDE will generate a custom board support package (bsp) for your Nios II system module and place it in the project subdirectory `software\project_name_bsp` when the project is built. To begin, open the Nios II IDE and set the workspace to `\software` directory for **myESystem**. Create a new C Application Project by selecting *File/New/Project*, then selecting `Nios II Application and BSP from Template` and clicking Next. On the following screen, select `Blank Project` (or `Board Diagnostics`) in the `Select Project Template` box, name the project, and browse to the `.sopcinfo` file that corresponds to your processor in the `SOPC information filename` box. This file will be named the same as the component name you entered when you first started the SOPC Builder, with the `.sopcinfo` extension. Clicking `Finish` will create the project.

If you have not already done so, you should read the documentation on timers and parallel ports, to familiarize yourself with the function of these devices and their registers. This documentation is provided in the data sheets on the Altera website: PIO Manual (`http://www.altera.com/literature/hb/nios2/n2cpu_nii51007.pdf`), Timer Manual (`http://www.altera.com/literature/hb/nios2/n2cpu_nii51008.pdf`), and JTAG UART Manual (`http://www.altera.com/literature/hb/nios2/n2cpu_nii51009.pdf`). You should also review the C tutorials provided on uWaterloo Desire2Learn . Examples of C applications that interface with timers and parallel ports are provided in these tutorials.

To interact with the devices in your embedded system, you will need to read from, and write to, the device registers. The header files for your Nios II system will be located in the `project_name_bsp` directory of your bsp. You should familiarize yourself with the contents of the `project_name.bsp` directory. The most important header file is `system.h`. This file is created when the bsp is built for the first time. To do so, right click on the `project_name.bsp` project in the Eclipse Project Navigator on the left and select `NIOS II/Generate BSP`.

`system.h` This header file contains definitions for configuration information pertaining to the peripherals included in your Nios II system module. It defines the device register names and offsets, as well as the bit masks you may use to control the operation of the devices.

`drivers/inc/altera_avalon_timer_regs.h` This header contains definitions of all the registers included in the timer and their offsets, as well as masks for setting and testing conditions. These include the status, control, and period registers.

`drivers/inc/altera_avalon_pio_regs.h` This header contains definitions of all the registers included in a PIO peripheral and their offsets, as well as masks for setting and testing conditions. These include the data, direction, interrupt and edge capture registers.

For example, if you have a PIO device named `led_pio`, `system.h` defines a label called `LED_PIO_BASE` which is the address corresponding to the `led_pio` on the Avalon Bus of the Nios II CPU. You may then write to the data register in your PIO using the following syntax:

```
IOWR(LED_PIO_BASE, 0, 0x3); \\ write 0x3 to the register at the LED_PIO_BASE address plus an offset of 0
```

Similarly, reading from a data register is accomplished using the following syntax[1]:

```
IORD(DIP_SWITCHES_PIO_BASE, 0); \\ read the value stored in the register at the DIP_SWITCHES_PIO_BASE address plus an offset of 0
```

You will need to set up interrupt handlers for your interrupting devices. Documentation on how the Nios II processor handles exceptions can be found in the Nios II Processor Reference Handbook. The Nios II Software Developer's Handbook discusses the `alt_irq_register()` routine, which can be used to setup an interrupt handler. Both of these documents are available on the course website, or in Start/Programs/Altera/Nios.../Altera Nios Documentation index.

To begin coding, select `File/New/File`, make sure your project folder (not the system library) is selected in the parent folder, and name the file (make sure to add the `.c` or `.h` extension). You may then begin writing code.

5.5.2 Desired Functionality

Phase 1: Interfacing push buttons, DIP switches, LEDs, timers and the seven-segment display - using both polling and interrupt synchronization techniques

Although the specification given below is fairly detailed, the actual goal of this part of the lab is to facilitate learning how the various interface alternatives function on this platform.

Your software (`lab1_phase1.c`) is to use push button and timer interrupts to implement the following functionality on the Altera DE2 Board:

- Pressing push button 1 (PB1) on the board[2] will cause the 8 bits set on the switches labelled SW0 through SW7 to be flashed to LED1, one bit at a time. If a bit is `0`, the LED will remain off for one second, if it is `1`, the LED will light for 1 second. The entire process will take 8 seconds, at the end of which the LED will turn off.
- Pressing push button 2 (PB2) on the board[3] will cause the 8 bits set on the switches labelled SW0 through SW7 to be displayed on the seven-segment display, one bit at a time. Each bit is to remain displayed for 1 second. The entire process will take 8 seconds, at the end of which the display will turn off.
- The device should respond to either push button at any time, rather than waiting until one or the other display is finished flashing.
- If the seven-segment display is flashing, it should complete its 8 second cycle unless PB2 (the seven-segment push button) is pressed before it is finished. If that happens, it will start flashing the current DIP switch byte again from the beginning.

[1] Note that reading from a write-only register, such as the data register in an output-only PIO, produces an undefined result.
[2] Labelled `KEY0` on the development board.
[3] Labelled `KEY1` on the development board.

- If the LED is flashing, it should complete its 8 second cycle unless PB1 (the LED push button) is pressed before it is finished. If that happens, it will start flashing the current DIP switch byte again from the beginning.
- Changing the DIP switches once a push button has been pressed should have no effect until the next push button is pressed.

Suggested approach to solving this phase of the lab study

You may wish to consider approaching the software design in stages. The following intermediate design stages may be useful:

1. Write a program to turn a LED on, pause for a fixed period of time, using an interrupt handler to keep track of time, and turn the LED off. Expand this program to control the seven-segment display by writing a "1" and then a "0".
2. Write a program which uses polling to detect a push button press. The program should print a character to the PC (use `printf`) when one of the buttons is pressed. Expand this program to work for two push buttons and use two different characters. You may wish to use the current value of the DIP switch byte as one of the "characters".
3. Write a program which sets up a 1 second timer interrupt, and use that interrupt to flash the LED every second.
4. Modify your previous code for the push buttons to use interrupt synchronization, rather than polling.
5. Combine all of the above functionality to achieve the desired specification.

Phase 2: Dealing with repeated input requests in the presence of a background task

In this phase of the lab, you are to examine the impact of various synchronization alternatives on the device driver's latency and your system's ability to complete a background task. In this lab, we provide two tools to facilitate these measurements.

1. A (simple) background task that performs a simple arithmetic operation. The granularity of the background task, specifying the number of calculations performed for each activation, is passed as a parameter.
2. The operation of an external device that issues pulses requiring a response is simulated by the EGM. The EGM also measures the latency of the response by the device driver. These pulses are issued with a specified period. It is possible to specify a granularity of background task in a polling environment that results in an entire pulse being missed. The data stored in the EGM is the maximum latency experienced (since initialization) as well as the number of pulses that were missed completely (that is pulses for which there was no response prior to the next pulse). The stored information is printed on the screen at the end of the experiment.

Figure 5.4 illustrates the control flow for an experiment. The function `init` (described below) initializes both the background task's variables and the EGM. The function `finalize` prints the results (latency as well as background task's progress) on the screen. In between these function calls is the experiment. In the experiment you must call `background` with an appropriate parameter to make progress on the background task. In addition, you must respond to pulse signals received from the EGM.

The functions (`init`, `background` and `finalize`) are provided in the source code file `ece324_egm.c` available on uWaterloo Desire2Learn .

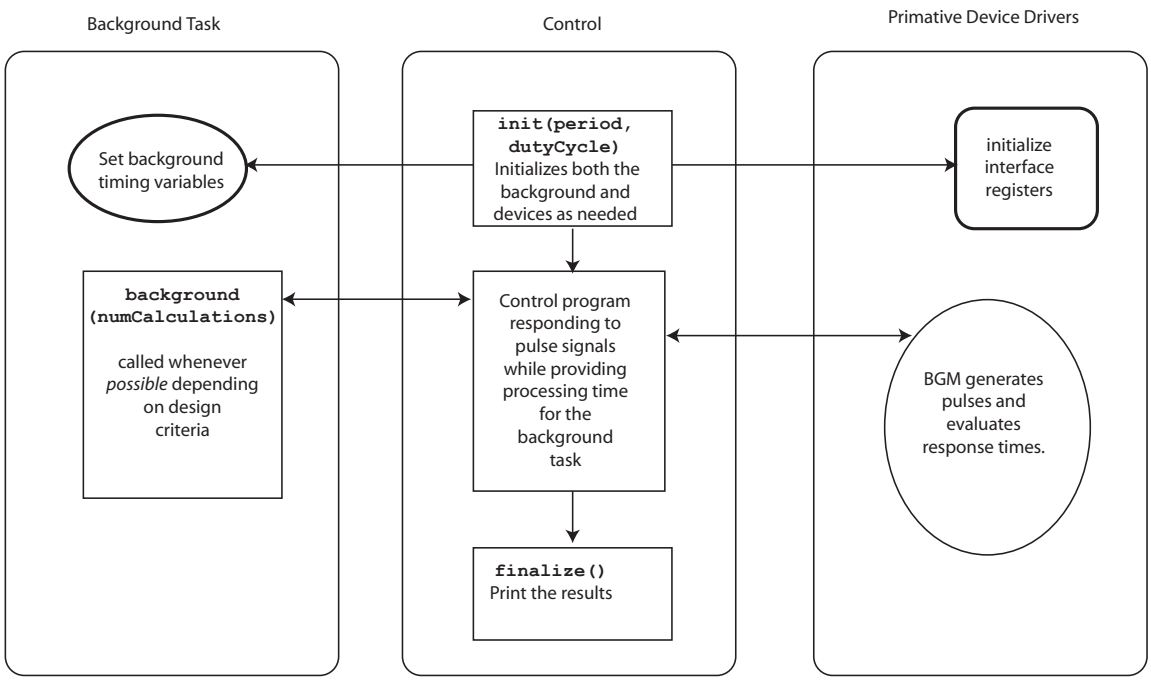

Figure 5.4: Software structure for Phase 2

The code below shows a basic implementation of a test. Note that it is necessary to include the file `io.h` to read and write to registers on the processor for input and output, as this is where the routines IOWR() and IORD() are defined.

```
// File: lab1.c
//
// Contents: Helper functions to complete lab 1

void init(int, int);
void background(int);
void finalize(void);

#include "system.h"
#include <io.h>

// our test
int main(void)
{
  int i;
  init(6, 8);
  for(i=0; i<100; i++)
  {
    while(IORD(PIO_PULSE_BASE, 0) == 0) {}
    IOWR(PIO_RESPONSE_BASE, 0, 1);
    background(20);
    while(IORD(PIO_PULSE_BASE, 0) == 1) {}
    IOWR(PIO_RESPONSE_BASE, 0, 0);
    background(20);
  }
  finalize();
}
```

It is of note in this code that both the rising and falling edge of the pulse required synchronization. It is also of note that there is no attempt to enable the background task while waiting for either the rising or falling edge of the pulse.

This points out one of the very important aspects of this lab. You must select design criteria, and then use that set of criteria for your designs. For example, you might decide that you can tolerate long latencies as long as you do not miss any of the pulses. Alternatively you could set out to minimize latency (or even to limit latency to some value). It would also be possible to set a goal of working with a given minimum period. In the report you will need to explain your decisions on this point and then explain how your experiments verify this aspect of your design.

Figure 5.5 illustrates the timing relationship for the parameters provided to `init(m,n)`. Figure 5.5 also illustrates two responses (produced by your program). In the case of response 1, the request isZZ serviced prior to the arrival of the next pulse. The latency, that is recorded in the EGM is also shown. In the case of the second response, the system did not respond in time and is considered a miss. A miss may result from either slow processing of the request, or from the control program missing the pulse altogether. Although these are different reasons, we will treat them as misses in this study.

Your software (`lab1_phase2.c`) should incorporate the following functionality:

- Access to the three functions described above and detailed below must be provided. The access involves placing prototypes for the functions in your .c file and then during compilation (the nr command) including the `ece324_egm.c` file in the list of files to compile.
  - During the control program's idle time, the software must perform iterations of the background task by calling the `background()` function.

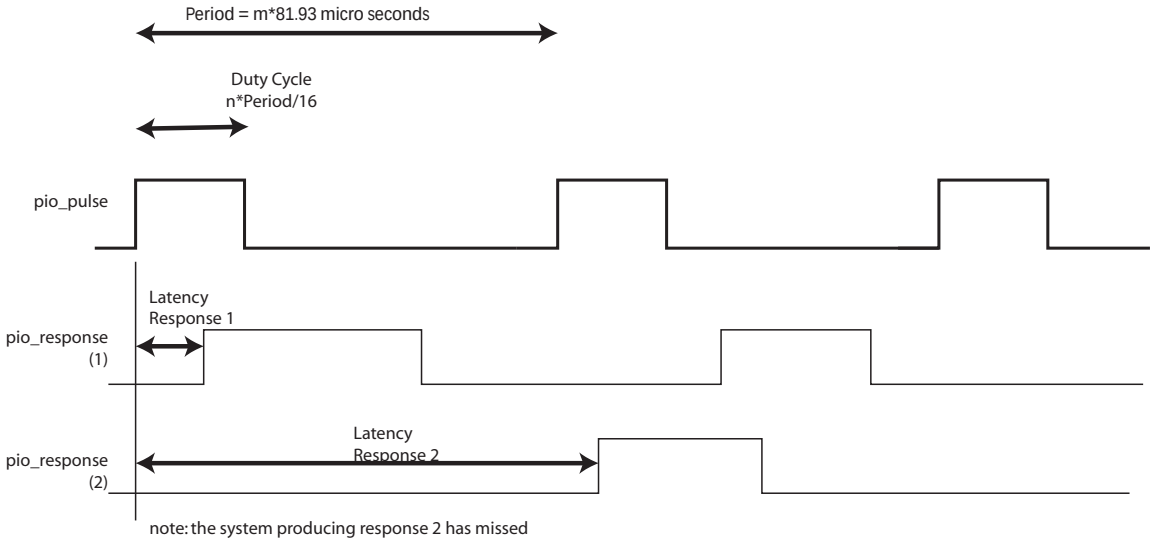

Figure 5.5: Period and Duty Cycle for Phase 2

```
int background( int numCalculations )
{
  // This function performs a specified number of calculations
  // and returns the result of the calculation.  For the purpose
  // of this lab, the return value can be safely ignored.
}
```

– Your program must call the provided `init()` function prior to the main loop and the `finalize()` function after the main loop to initialize the EGM and report statistics respectively. These functions have the following prototypes:

```
void init( int period, int dutyCycle )
{
  // This function initializes the EGM to generate events with
  // a specified event period and duty cycle.  The period and
  // duty cycle must have values between 1 and 14 (inclusive).
  // The resulting event period is calculated by:
  //   event period = [period + 1] * 81.93 uS
  //
// a period specification of 0, results in fixed period of 1 Hz.
}

void finalize()
{
  // This function outputs event statistics.
}
```

- When an event (i.e., a rising edge on the "pulse" pio) is encountered, your software must respond to the event by outputting a high signal on the "response" pio. This signal must be reset to low before the next event. The response signal must be set to a high signal for a minimum duration of 1/1024th of a period to ensure that the signal is observed by the EGM.

- In addition to performing these two operations, you are to design an experiment that will permit you to

compare the operation of 2 different synchronization techniques (different from the one described in the sample code above). The following information is provided for your experiment.

- You must compare two substantially different synchronization methods to respond to an event generated by the EGM. Although the synchronization techniques must be different, you must set out to use the same design objectives. For example, if the goal were to be to minimize latency, this would be the same objective for both designs.
- Since each experiment will require many tests (for different granularity, period and duty cycle values), you may wish to increase the complexity of your control program to permit several different values to be tested on each run of the control program.
- Your goal is to analyze the performance of the synchronization methods with respect to latency, missed events, and progress of the background task.
- The program should continue to operate until 100 events have been observed. The data that results from this number of pulses will be reasonable for analysis. However, your analysis will have to take into consideration that each experiment will run for a different (wall-clock) time. For example: when you evaluate your system with a period of 4 against the performance with a period of 9, 100 events will take more than twice as much time in the second case, and as a result you would expect significantly more background progress in the second case.

Suggested approach to solving this phase of the lab study

You may wish to consider approaching the software design in several stages.

Stage 1: Write a small program that uses the `init` and `finalize`, however set the period to 1 Hz, as described above, (you specify a period of 0 and the actual period is 1 Hz) and a 50% duty cycle pulse. Your software can then look for this pulse on the pulse PIO and indicate its presence by lighting the LED (from Phase 1). You should end up with the LED flashing at a frequency of 1 Hz with a 50% duty cycle. Once you are sure that you are properly reading the pulses generated by the EGM, you may proceed to write the required software.

Stage 2: Use the sample program, shown above to confirm that tight polling produces results.

Stage 3: Using your design objectives and one synchronization technique, develop your first experiment. The experimental results should show the performance for a range of values for period, dutycycle, and background granularity.

Stage 4: Modify your implementation to a quite different synchronization technique (e.g., interrupts, periodic polling, or occasional polling) and repeat.

5.5.3 Compiling & Downloading to the Nios II Embedded Processor Development Board

To build your project, open the Nios II IDE and set your workspace to be the directory where you placed the project (usually *Quartus_project_folder*\`software`). The project should appear on the left of your screen in the Project Navigator column. Right click on the project in the Project Navigator box on the left side of the screen and select *Build* from the context menu. You can also select *Build Project* from the *Project* menu or press *Ctrl+Shirt+F3* on the keyboard. This will build the project and compile both the system library and the code you have written. Note that to view the system library (and `system.h`) for the first time, it is necessary to build the `board support package` project so that the Nios II IDE can create all the necessary system description components and the `system.h` library. If you do not wish to look at this file for reference, it is not necessary to perform this task prior to compiling your code.

While compiling, the *Console* tab should appear or come into focus on the bottom part of the Nios II IDE Window. This console will display any messages that the compiler generates, including errors and warnings. These errors and warnings will also appear in the *Problems* tab that is in the same section of the IDE as the *Console* tab, where they can be double-clicked to locate them in the code.

Once your code has compiled correctly, you may download it to the Nios II processor on your Altera DE2 Board. To do this, ensure that you have configured your project hardware using Quartus II. Once this is complete, from within the Nios II IDE, right click the project and select *Run As / Nios II Hardware* from the context menu. This can also be done by selecting *Run As/Nios II Hardware* from the *Run* menu, or by clicking the down arrow to the right of the green Run button on the Nios II IDE toolbar and selecting *Run As / Nios II Hardware*. This begins the process of downloading your code to the processor's memory. Once this process is complete, the processor will be reset and will then execute the code.

5.6 Testing, Debugging, and Going Further

Should you run into a situation where it appears that both hardware and software have compiled and downloaded correctly, but the system doesn't function as expected, you will need to debug the situation. Debugging embedded systems can be difficult unless the system has been designed with debugging in mind. You may wish to consider this, and add some software functionality which will allow you to verify aspects of your design.

Some of the debugging strategies you attempt may involve hardware redesign. For example, you may wish to have LED2 respond to the pulse signal. This may require regenerating the Nios II system module with different peripherals and peripheral settings.

Even if your design is successful, you may wish to modify or enhance it in some way. You are encouraged to think up modifications to the design and explore what would be required to implement them. We welcome interesting suggestions for future labs, as well.

In doing any of these things, you will want to make extensive use of the Nios II documentation, which is available in full on uWaterloo Desire2Learn . In addition, feel free to consult with any of the teaching staff, who will help you to the best of their abilities.

5.7 Deliverables

The emphasis of Lab 1 is learning the system, developing your skills at investigating performance and developing your skills at managing conflicting requirements.

The deliverables described in Section 2.3.5 provides the basic requirements for the deliverables. The remainder of this section provides a description of the content for this lab.

5.7.1 Demonstration

The demonstration required for this lab study is as described in Section 2.3.5

5.7.2 Report

A report describing certain aspects of your design (described in detail below) is to be submitted to uWaterloo Desire2Learn .

1. The general deliverables for all reports as described in Section 2.3.5.

2. For phase 1 the report is to include the following (in no more than three pages):

   (a) A short overview of your software design, using block diagrams, pseudocode, prose, or a mixture of some or all of these.

   (b) A discussion of which synchronization method seems to perform the best. (What is important in this view of best?)

   (c) A description of the pros and cons of your synchronization choice, with respect to efficiency, ease of implementation, and ease of testing/debugging.

   (d) A description of your testing strategy, one problem you ran into, how your testing strategy exposed the problem and how you went about debugging it. If you encountered *no* problems that required debugging, explain how your testing strategy verified the complete (correct) operation of your system. You should keep a lab diary so that you can easily include this information in your report. The report should contain a summary of your debugging approach, rather than 10 pages rewritten or photocopied from the diary.

3. The report for phase 2 is to include the following items (in four pages):

   (a) A clear specification of how you implemented the **two** tested synchronization techniques. (It may be possible to specify this with pseudocode and an explanation.)

   (b) A clear presentation of your experimental results and an analysis of the results. Your experiments should include a variety of periods and duty cycles.

   (c) The presentation should include a description of the criteria used for comparison with reference to latency and background task progress.

5.7.3 Code Submission

An electronic copy of the C code that you have used during the demonstration. This code is to be submitted through the uWaterloo Desire2Learn system. (We reserve the right to use an automated procedure to compare your code to other code submitted for this lab study.)

Chapter 6

Lab 2

6.1 Moving your design from Lab 1 to Lab 2

As in lab study 1, you are to start from your last lab design and modify it to for this lab study. To help with this the following steps are recommended to capture your lab 1 design and to start from a known point with this lab study.

- Copy your project from lab tools tutorial or lab 1 using the command from the menu Project Copy Project. Select the proper destination where you will copy the new project and name it "lab_2". Verify the open new project box is checked and accept the creation from the new directory file. Also download the lab 2 support files from the uWaterloo Desire2Learn to this directory.
- Open myEsystem.bdf and double click on your microcontroller symbol to open the SOPC builder and edit the components in the project.
- If you are starting from the Lab 1 project then you should delete the following components:
  - PIO_pulse
  - PIO_latency
  - PIO_missed
  - PIO_dutycycle
  - PIO_period
  - PIO_response
  - PIO_egmenable and
  - PIO_egmreset
- It will be necessary to add the following (as described in the lab description that follows.)
  - aud_full
  - sd_dat
  - sd_cmd
  - sd_clk
  - UW ECE/audio
  - UW ECE/open_i2c_o

  The final system block diagram will be similar to the one shown in Figure 6.1.

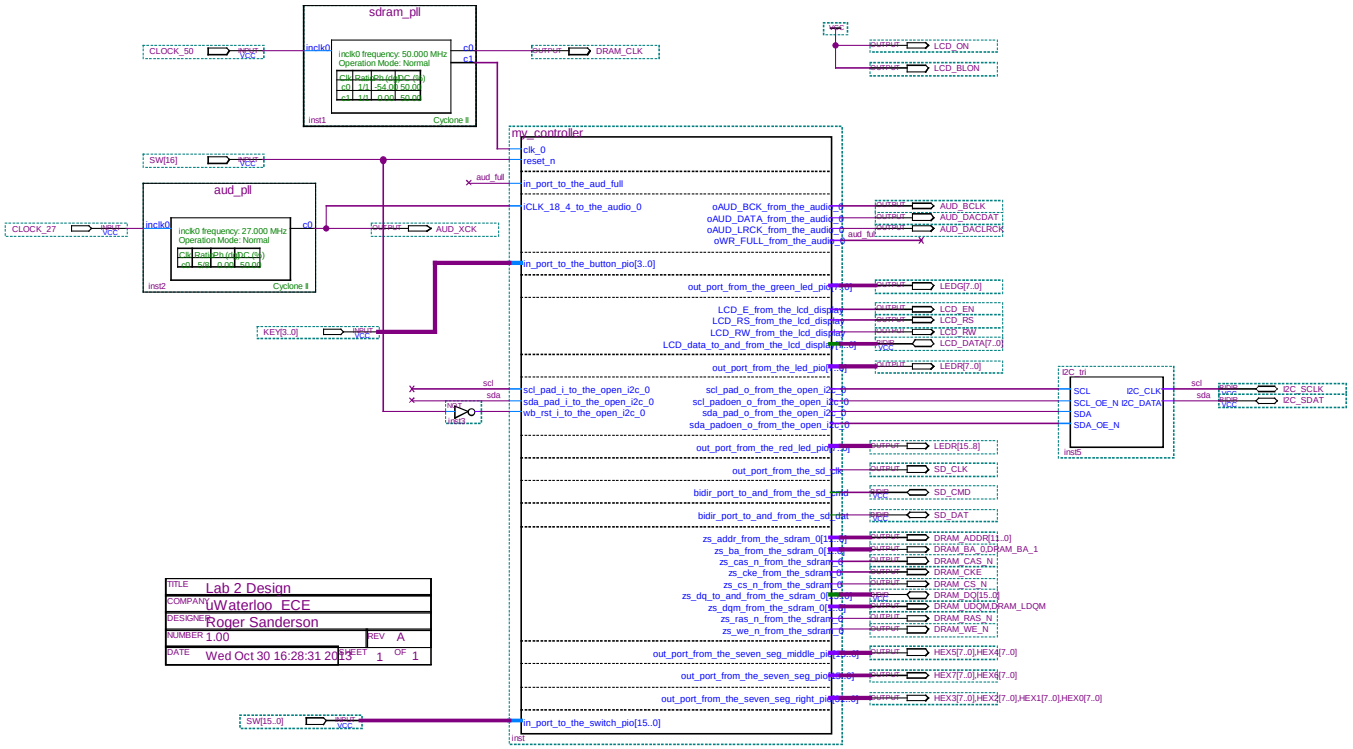

Figure 6.1: Lab 2 Nios Block Diagram (a bigger version is on uWaterloo Desire2Learn)

6.2 Purpose

These labs use relatively new tools. This brings the exciting opportunity to work with a current leading edge system and set of tools. The tools provide a great deal of flexibility in terms of how your system may be implemented. Also, Lab 2 involves a significantly more challenging problem to be solved in comparison to previous lab. It is important to remember as you work on the lab that the goal of the lab is not necessarily to have a perfectly working system, but to learn about the process of designing, implementing, and testing an embedded system that reads data from external memory and uses an analog device (speaker/headphones).

The purpose of Lab 2 is to familiarize students with embedded system design, implementation, and testing. Upon completion of this lab, you will have had an opportunity to do the following:

1. design and implement hardware interfaces within a Nios II system;
2. interface a microcontroller with a SD memory card and an analog speaker or headphones;
3. design, simulate and verify the operation of a VHDL core;
4. read data from a FAT file system;
5. apply simple audio processing techniques;
6. design and implement C/C++ device drivers; and
7. design and program C/C++ software to operate an embedded system.

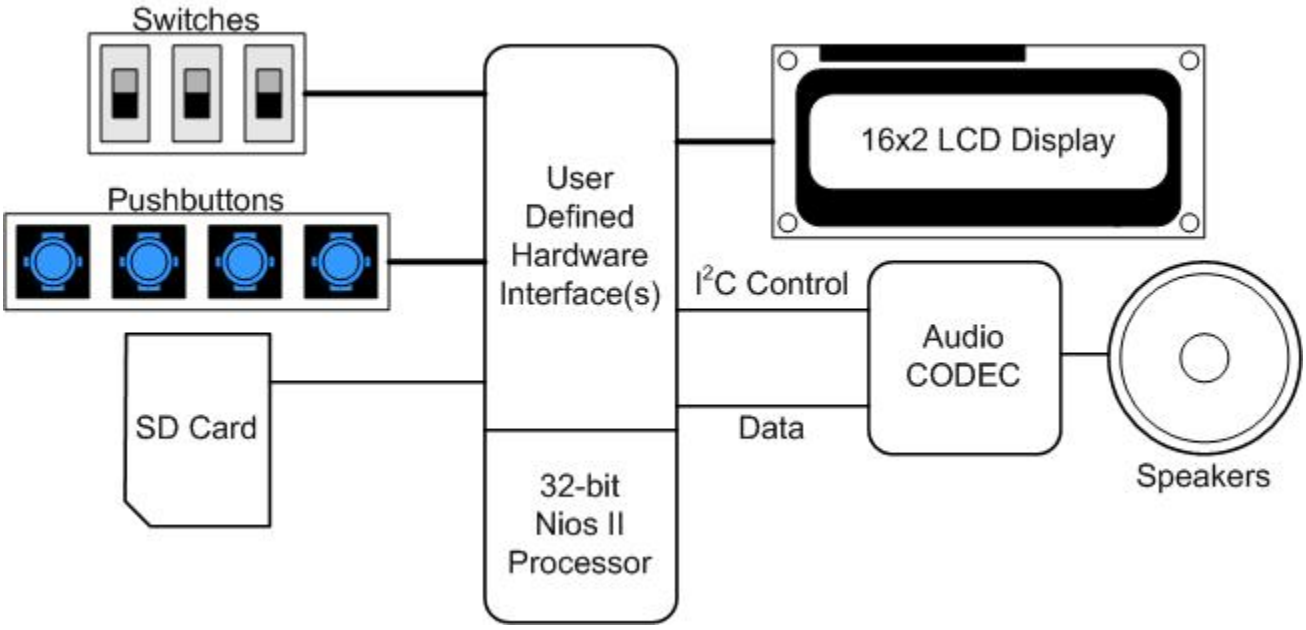

Figure 6.2: Lab 2 Complete System Diagram

6.3 Lab Outline

In this lab you will build a complete embedded system. A block diagram of the system is shown in Figure 6.2. The system reads .wav files from a SD card organized by a file allocation table (FAT) file system, and then sends the audio data through an audio CODEC to a speaker/headphones for playback.

The system can operate in five different modes depending on the position of the switches. The implementation of the mode selection is up to you, the designer, but the system should be capable of playing songs at normal, double, and half speeds, delaying one channel a full second behind the other channel, and playing songs in reverse.

When push button two is pressed and released, the system should browse the file system on the SD card for the next song to be played. The song title and song mode (as set by the switches) should be displayed in the LCD display. When the last file on the SD card is being displayed, pushing push button two should skip back to the first file on the SD card.

When push button three is pressed and released, the system should browse the file system on the SD card for the previous song. The song title and song mode should be displayed in the LCD display. **Note**, that when the first file on the SD card is being displayed, the player does not have to skip to the last song on the SD card when push button three is pressed.

When push button one is pressed and released, the system should start playing the audio file currently displayed by the LCD display in the mode selected by the switches. Users should not be able to browse the file system while a song is playing.

When push button zero is pressed and released, the system should stop playing the audio file currently playing. If no file is playing, pushing button zero has no effect.

The system uses the following components. It is a good idea to read the datasheets associated with each component, however it is not necessary to read a components entire datasheet to understand its use in the lab.

- Secure Digital Card

(Located on LEARN under Content, Labs, Lab Datasheets),

- FAT File System
(Located on LEARN under Content, Labs, Lab Datasheets),
- Wolfson Microelectronics WM8731 audio CODEC
(Located on LEARN under Content, Labs, Lab Datasheets);
- an audio speaker;
- a microcontroller system that includes a Nios II processor, timer(s), a phase locked loop (PLL), parallel I/O port(s), LCD display controller, I^2C controller, JTAG UART to communicate with the PC, and custom interfaces implemented using VHDL;
- Nios II Development and Education (DE2) board components including the switches, LCD display, and push buttons.

The lab will be divided into 3 phases of development as follows:

Part I
- implementation of a Nios II soft core processor
- implementation of a PLL to produce the proper clock rate for the audio CODEC
- implementation of a VHDL tristate interface for the I^2C controller
- the complete embedded hardware system for audio wave file player

Part II
- Implementation of a device driver to read data from the SD card
- implementation of a device driver to read information from the FAT file system
- implementation of a device driver to initialize the audio CODEC on the I^2C bus
- implementation of a system to play the first .wav file from the SD card

Part III
- implementation of the five different playback modes as described earlier
- implementation of the dip switches to control the playback mode
- implementation of a system to accept interrupts from the DE2's push buttons with functionality as described earlier
- Use the LCD display to show the song file name and playback mode

6.4 Preparation

Prior to attempting this lab, you should be familiar with the Quartus II software and VHDL. It is assumed that you have successfully completed Labs 0 and 1 before attempting this lab.

It is highly recommended that you review the Altera documentation on the course website, particularly on the following topics (although only the first two should be new to you at this point):

- the Nios II Mega Wizard Plug-in Manager for PLL's,
- the Nios II LCD Peripheral,
- the Nios II Timer Peripheral,
- the Nios II Parallel I/O Port (PIO), and
- interrupts in the Nios II processor.

Table 6.1: Properties for Lab 2 Nios II Peripherals

peripheral	properties
audio_0 (in UW ECE folder)	*default properties*
aud_full	1 bit PIO, input, no edge capture, no interrupts
open_i2c_0 (in UW ECE Folder)	*default properties*
sd_dat	1 bit PIO, bidirectional, no edge capture, no interrupts
sd_cmd	1 bit PIO, bidirectional, no edge capture, no interrupts
sd_clk	1 bit PIO, output

Note: You should use the same names for the peripherals as listed above to make the peripherals accessible to the software provided.

It is also highly recommended that you review the following specification sheets included on the course website. Of particular importance are the command codes for communicating with the audio CODEC.

- Audio CODEC WM8731, (Located on uWaterloo Desire2Learn under Lessons, documentation)
- SD Specification, (Located on uWaterloo Desire2Learn under Lessons, documentation)
- FAT Specification (although the specification includes FAT32 in its title it has information on FAT16 and FAT12 as well), (Located on uWaterloo Desire2Learn under Lessons, documentation)
- I^2C Controller Datasheet, (Located on uWaterloo Desire2Learn under Lessons, documentation)

Students with weak programming skills may benefit from reading Chapter 2 of Valvano.

Finally, it is important that you review the course notes on the following topics:

- Serial Communication.

Prior to doing any development, create a project directory for this lab on your user account on Nexus. The pin assignments for this lab, ECE_DE2_pins_2011.csv, are the same generic pin assignments for the DE2 board that you've used for the two previous labs.

It is very important for this lab to have your hardware and software designs completed, and your software coded, BEFORE you come to your scheduled lab period.

6.5 Part I: Initial Microcontroller System

In Part I, you will build a Nios II system module using the Quartus II MegaFunction Wizard, according to specification. You will also use the MegaFunction Wizard to create a phase locked loop to be used by the audio CODEC to generate the necessary sample rates. You will then instantiate the Nios II system module, the PLL, and a tri-state component for the Inter-Integrated Circuit communication bus, and wire the design. Finally, you will compile the design and program it into the Cyclone II device on the DE2 Development and Education Board.

6.5.1 Nios System Module

Within your lab2 project, from the Tools menu, select MegaWizard Plug-In Manager. Follow the steps to configure your Nios II 32-bit processor. Your Nios II 32-bit processor will contain the basic components shown in

Table 6.2: Properties for Lab 2 Audio PLL

parameter	**setting**
Device Family	Cyclone II
Input Clock Frequency	27.0 MHz
Operation Mode	Normal Mode
page 4 of 10	uncheck all boxes
page 5 of 10	all defaults
page 6 of 10	output clock parameters, multiplier 5, divider 8, phase shift 0, duty cycle 50%
page 7 of 10	all defaults
page 8 of 10	all defaults
page 9 of 10	all defaults
page 10 of 10	all defaults

Table 6.1, plus any other components such as hardware timers, parallel I/O ports, etc. that you feel are needed to implement your design for Part I. Components not listed in this table should be configured as required by your overall system design. **Note:** Choose your interrupt priorities wisely for your Nios design. Remember that the lower the IRQ, the higher the priority.

Once you have added the peripherals you will need to take all the steps to generate the resulting system illustrated in the lab tools tutorial

6.5.2 Audio Phase Locked Loop

The Audio CODEC on the DE2 Development and Education Board is made by Wolfson Microelectronics and is capable of encoding and decoding audio at sample rates of up to 96kHz. In order to be able to sample at that rate, it must be provided with data at that rate, so a data clock must be provided to achieve that. However, a much higher speed clock is needed to do the actual analog-to-digital or digital-to-analog conversion. For the mode that will be used in this lab, a clock frequency of 16.9344 MHz is needed. Since none of the clocks on the DE2 board operate at this frequency, it is necessary to use a phase locked loop to achieve this clock rate.

The Audio interface component in the SOPC Builder contains several clock dividers that are used to divide this clock into the necessary data rate clocks as required by the audio CODEC. Therefore, it is necessary to connect this clock to the interface component in the Nios II block via the `iCLK_18_4` pin as well as to the audio CODEC master clock (`AUD_XCK`).

The PLL requires an input clock to synchronize with, but only one PLL may be run off of any clock signal for stability reasons. Since we have already used one PLL on the 50 MHz clock for the SDRAM, we will need to use the second clock available on the DE2, the 27 MHz clock.

To build the PLL, open the MegaFunction Wizard and select *Create a new custom megafunction variation*. On the following screen, in the megafunction list on the left, expand the *I/O* folder and select *ALTPLL*. Ensure that the device family is set to Cyclone II, VHDL is selected as the output file type, and the folder in the name box is the working directory of the project you have created. At the end of this path, type a name appropriate to the Audio PLL. Choose next and the ALTPLL MegaWizard Plug-In Manager [page 3 of 10] appears.

Table 6.2 shows the properties used to configure the PLL to produce the necessary output frequency to run the audio CODEC clocks. The wizard is the same as the one used to configure the SDRAM PLL; only the settings are different. All settings should be left as defaults unless explicitly mentioned. When you have entered all the settings, press finish to complete the part. It will now appear in the *Insert Symbol* interface in the *Libraries* box under the *Project* folder with the name you gave it at the beginning of the process.

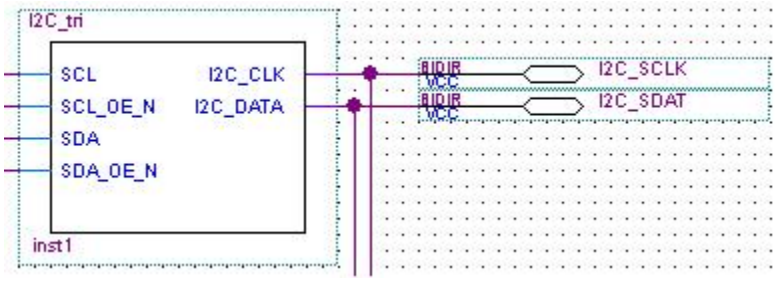

Figure 6.3: I^{2}C Tristate Buffer Block

6.5.3 Inter-Integrated Circuit Bus

The I^{2}C bus is a serial bus developed by Philips for low-speed communication between integrated circuits in a system. It uses a single data line and a clock line to transmit and receive data. Because it can both send and receive data on the two lines, it is necessary to use bidirectional pins in your top-level diagram.

The I^{2}C component provided for the course is an I^{2}C bus master. It was designed with separate inputs and outputs for each of the data and clock lines as external connections to the component. Because these four lines must connect to the same two pins on the FPGA to communicate with the audio CODEC, it is necessary to tristate the outputs so that the master is not driving the lines when a slave is trying to respond. For this purpose, the I^{2}C master is equipped with output enable pins for each of the clock and data lines. You have been provided with a tristate buffer component, `I2C_tri.vhd & I2C_tri.bsf`, on the course website that will permit you to buffer the output when the enable is low and set the output to high impedance when the enable is high so that a slave may drive the signal lines. Figure 6.3 shows the tristate buffer component and its connections.

6.5.4 Completing the Hardware Design

Note: we have provided an example schematic of the top level design, you can download it from uWaterloo Desire2Learn (in the lab 2 area). Your schematic may differ if you chose different PIOs etc., but the basic layout should be similar.

Combine your Nios II system module, your PLL block, the I^{2}C Tristate Buffer component, and any other necessary symbols. Wire and name your pins. Make sure you use Bidirectional pins for the bi-directional signals. Note: I2C_SCLK and I2C_SDAT are bi-directional.

In order match the csv script, the following names must be used.

- For Data bus, Address bus, and related control signals use the following signal names: DRAM_CLK DRAM_ADDR DRAM_BA_0,DRAM_BA_1 DRAM_CAS_N DRAM_CKE DRAM_CS_N DRAM_DQ DRAM_UDQM,DRAM_LDQM DRAM_RAS_N DRAM_WE_N
- For the clocks use the following names CLOCK_50 CLOCK_27
- For the Audio Interface use the following signal names: AUD_XCK AUD_BCLK AUD_DACDAT AUD_DACLRCK
- For the I2C bus us the following signal names: I2C_SCLK I2C_SDAT
- For the LCD display use the following signal names: LCD_EN LCD_RS LCD_RW LCD_DATA LCD_BLON LCD_ON
- For PIOs use the following signal names: SW KEY[3..0] sd_clk sd_cmd sd_dat

Note that there are six pins dedicated to the LCD, four of which appear on the controller. The other two are control lines for turning on the power to the panel and the power to the backlight on the panel. Both are necessary to use the LCD display.

You are provided with a csv file, `ECE_DE2_pins_2011.csv` on the course website, which will do the pin assignments for you. When you are confident that your pins are labelled as listed above, import this file from the *Assignments**Import Assignments* menu. Verify that the pin assignment has been made correctly by examining the Assignment Editor (found in the Assignment pull-down menu). Also ensure that the target device is set to EP2C35F672C6 and explicitly **set the unused pins to tri-stated inputs.** This is done through the Device section of the Settings dialog (found in the Assignment pull-down menu).

When this is done, set the compilation focus to `meEsystem.bdf` and compile the design. Remember to investigate all compiler warnings as they may indicate problems with your wiring, pin names and pin assignments. Once the compilation is complete, you may program the hardware into the Cyclone II device.

6.6 Part II - Reading a Single Wav File

The second part of the lab focuses on playing a single .wav file from the SD card and hearing its output on the speaker/headphones. In Part III of the lab, the functionality of the player is increased by including the push buttons, dip switches, and LCD display.

Since reading a file from the file system of an external memory card and interfacing with an audio CODEC is a non-trivial task, significant portions of the code for this lab have been provided.

The following files have been included on the course website for use with this lab, and can be imported into your Nios II project as per the method outlined in lab 1:

- SD_Card.h - provides a means for reading data from the SD card
- fat.h - provides functionality for interfacing with a FAT file system
- LCD.c, LCD.h - provides a simplified interface for the 16x2 LCD display (used in Part III)
- basic_io.h - Used by Nios II private functions

Copy these files into the software project working directory. In addition to the files listed above, you also require: `Open_I2C.h, Open_I2C.c, wm8731.h`, and `wm8731.c` that are in the `software_V10.zip` file on uWaterloo Desire2Learn .

The functions included in these files are shown in Figure 6.4. More detail on these functions is included in the subsequent sections below.

6.6.1 SD Card

The Secure Digital Card (SD Card) is a flash-based memory card. It is based on a 9-pin interface consisting of a clock, command, 4 data, and 3 power lines. To communicate with the device, a command code is sent from the host (processor) to the device followed by any necessary additional data. The device will then respond accordingly. For instance, to read a single logical block of data from the SD card, the user would issue the `single block read command (CMD17)` followed by the logical block address of the block to be read. **All of the low level SD card commands are encapsulated in the functions provided in SD_Card.h**. Of the functions provided, the only **2** that you as the developer may need to use are:

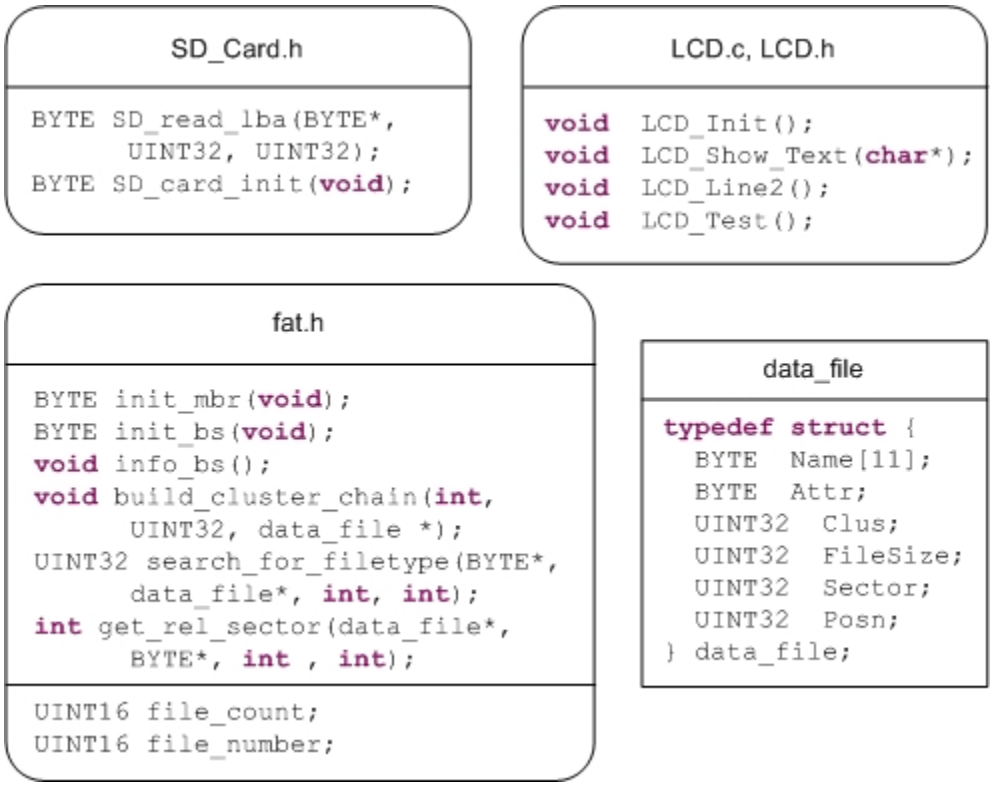

Figure 6.4: Provided Code for Lab 2

BYTE SD_card_init(**void**)
This function is used to initialize the SD card. It must be called before any communication with the SD card can be performed.

Parameters:

- None

Returns:

- 1 - if initialization encounters an error
- 0 - if initialization is successful

BYTE SD_read_lba(BYTE ***buff**, UINT32 **lba**, UINT32 **seccnt**)
Reads a logical block address(LBA) from the SD card. Each logical block consists of 512 bytes of data. The addressing starts from LBA 0 and continues sequentially throughout the size of the card.

Parameters:

- BYTE ***buff** - this is a buffer that stores the block of data read from the SD card. This buffer must be created prior to calling the function, and its size must be at least the size of one logical block (i.e.

  ```
  BYTE buffer[512] = {0};
  ```

- UINT32 **lba** - this is the logical block address to be read and stored in the buffer

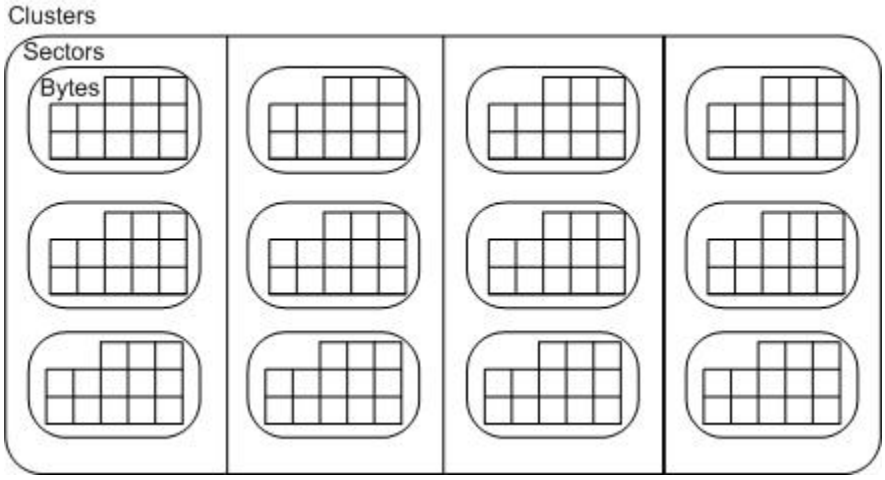

Figure 6.5: FAT Data Organization

- `UINT32` **seccnt** - this is the count of how many sectors are to be read into the buffer. For the purpose of this design, this should always have a value of 1.

`Returns`:

- `0` - once the data has been transferred into the buffer

More information can be found on the SD card by referring to the SD card specification on the course website. Of particular interest is the timing diagram for the single block read command under section 4.11.2.

6.6.2 FAT File System

FAT stands File Allocation Table, but it is also used as the family name for FAT file systems using the File Allocation Table. The FAT file system is divided into 4 major areas:

- Boot Sector - This area defines the volume and contains the location for the other 3 areas. Two important constants defined in the boot sector are: BPB_BytsPerSec and BPB_SecPerClus. These constants will be discussed later and must be used for various calculations throughout the project.
- File Allocation Table - This area is a series of addresses that is accessed as a lookup table to see which cluster comes next when loading a file or traversing a directory. Clusters are discussed below.
- Root Directory - Contains informations such as name, starting location, and size for all of the files and directories on the volume.
- Data Area - Fills the rest of the volume; it is here that the file data is stored.

The data area is divided into partitions known as clusters as shown in Figure 6.5. Each cluster is a single unit of data storage at the FAT file system logical level. Each cluster contains a fixed amount of sectors. A sector is a single unit of storage at the physical disk level. Within each sector there are a certain amount of bytes of information. The amount of bytes per sector and sectors per cluster are stored as constants in the boot sector labelled BPB_BytsPerSec and BPB_SecPerClus respectively. If these two constants are multiplied together, the result would be the amount of bytes per cluster on the FAT volume.

Currently there are three FAT file systems types: FAT12, FAT16, and FAT32. The basic difference in these FAT sub types, and the reason for the names, is the size, in bits, of the entries in the actual FAT structure on the disk. There are 12 bits in a FAT12 FAT entry, 16 bits in a FAT16 FAT entry and 32 bits in a FAT32 FAT entry. The functions in fat.h have been developed to support each of the three file system types.

Defined within the file `fat.h` are 6 functions that may be used for this system. All other functions are called by these 6 functions, and you should not need to use them when developing your system. If you wish the learn more about these supporting functions, you can read the FAT specification on the course website. A brief description of the 6 functions is provided below:

`BYTE` init_mbr(**void**)
This function is used to initialize the master boot record (MBR). It contains the location of the Boot Sector for the file system. It must be called before any communication with the file system can be performed.

`Parameters`:

- None

`Returns`:

- `1` - if initialization encounters an error
- `0` - if initialization is successful

`BYTE` init_bs(**void**)
This function is used to initialize the boot sector for accessing the FAT file system. It contains information regarding the file system, including: the amount of bytes per sector, sectors per cluster, location of the file allocation table, root director, and data sector. It must be called before any communication with the file system can be performed.

`Parameters`:

- None

`Returns`:

- `1` - if initialization encounters an error
- `0` - if initialization is successful

`void` info_bs(**void**)
This function is used to print information regarding the boot sector, as well as the results of calculations based off of values in the boot sector, such as the total number of sectors and clusters, to the console. This information is provided for debugging purposes only.

`UINT32` search_for_filetype(`BYTE` ***extension**, `data_file` ***df**, `int` **sub_directory**, `int` **search_root**)
This function is used to search the file system for a given file type as designated by **extension**. This function works in conjunction with two global variables, `UINT16` **file_number**, and `UINT16` **file_count**. The function works by starting at the beginning of the volume, and searching for the n^{th} file with the extension of interest, where n is designated by **file_number**. The function keeps a count of how many files of the given extension it has found with the variable **file_count**, when **file_count** is equal to **file_number**, the file information for that particular file is packaged (as described below) and the **file_number** variable is incremented so as to search for the next file on the volume the next time the function is called.

`Parameters`:

- BYTE ***extension** - This is the extension type that the function is searching for. To search for a particular extension, the extension must be enclosed in double quotes, " ", and all characters must be capitalized. (i.e. to search for a wave file, the parameter "WAV" must be used)
- data_file ***df** - This is a structure used to store the information returned from the function. The structure data_file is described below
- int **sub_directory** - This parameter is used to begin the search from a particular sub directory. This parameter is used recursively within the search_for_filetype function, and should be **set to 0** when being called outside of the function to start the search from the root directory.
- int **search_root** - This parameter indicates if the directory being searched in the root directory. This parameter, like **sub_directory**, is used recursively within the function, and should be **set to 1** when being called outside of the function to indicate that the search is starting from the root directory.

Returns:

- 1 - if a file with the given file extension has **not** been found
- 0 - if a file with the given file extension has been found

void build_cluster_chain(int **cc[ ]**, UINT32 **length**, data_file ***df**)
This function builds an array of clusters in the order that they appear in a given file. It is necessary to buffer this cluster information before playing a .wav file to ensure that data can be read from the SD card and delivered to the audio CODEC at the expected rate. **Failing to buffer the cluster information before playing a file will be very noticeable during audio playback**.

Parameters:

- int **cc[ ]** - This is the array that the cluster chain will be returned in
- UINT32 **length** - This is the length of the cc[ ] array, it is equal to the amount of clusters used by the audio file. This value must be calculated before instantiating the array. This value is calculated by the given equation: 1 + ceil(file size in bytes / bytes per cluster)
- data_file ***df** - This is the audio file that the cluster chain is being built for

int get_rel_sector(data_file ***df**, BYTE ***buffer**, int **cc[ ]**, int **sector**)
This function is used to read a sector from a file. It functions similar to SD_read_lba, however the sector number that is read is relative to the file in the parameter **df**. For this function to perform correctly, the user must pass it a file with a constructed cluster chain.

Parameters:

- data_file ***df** - This is the data file from which the sector is to be read
- BYTE ***buffer** - This is the buffer that the sector data will be returned in
- int **cc[ ]** - This is the cluster chain for the file
- int **sector** - This is the sector number relative to the file being read. **Note**: The first relative sector of the file is considered to be sector zero (0)

Returns:

- -1 - if the sector number is less than 0 or greater than the last sector

- 0 - if the sector number is a valid sector
- <bytes in last sector> - if the sector number is the last sector in the file

Data Structure `data_file`
The `data_file` data structure used to store file information returned from the `search_for_filetype` function is outlined below.

```
typedef struct {
  BYTE    Name[11];   //FileName
  BYTE    Attr;       //File attribute tag
  UINT32  Clus;       //First cluster of file data
  UINT32  FileSize;   //Size of the file in bytes
  UINT32  Sector;     //First sector of file data
  UINT32  Posn;       //FilePtr byte absolute to the file
} data_file;
```

- `BYTE` Name[11] - This is the file's 11 character short file name as stored by the file system
- `BYTE` Attr - The attribute stores information regarding whether the FAT entry is a file or directory. A description of the attribute values is provided in the FAT specification on the course website
- `UINT32` Clus - This is initialized to the first cluster that the file resides in
- `UINT32` FileSize - **This is perhaps the most important piece of data_file information when building your system** it contains the total length of the file in bytes
- `UINT32` Sector - This is initialized to the first sector in the first cluster of the file
- `UINT32` Posn - This is initialized to zero, and can store a file pointer's relative position within a file

If you choose, these fields can be updated as a file is read to keep track of the current cluster, sector, and byte that the program is reading.

6.6.3 Configuring the Audio CODEC via the I^{2}C bus

Note: This section goes into some detail about programming the CODEC. This term we have given you most of the routines that you need to use this device. They are in the `WM8731.c` file. The CODEC initialization code has been set to reflect a speed of 44.1 kHz.

The audio CODEC must be configured before it can be used. The DE2 connects the audio CODEC to the FPGA using a serial bus called the I^{2}C bus, which is controlled by the I^{2}C controller you included in your SOPC Builder system. The I^{2}C controller must also be configured before it can be used. There are three registers that determine how the controller behaves. These are the PeriodL, PeriodH, and Control registers. The register map of the I^{2}C controller is found in Table 6.3.

The PeriodL and PeriodH registers control the prescale division of the clock from the system clock to the bus clock. The internal clock for the controller operates at $1/5$ of the system clock, and this has to be divided down to 100 kHz for the I^{2}C bus clock. An example calculation is done in the I^{2}C controller datasheet; the Nios II uses an internal clock of 50 MHz. Each of the registers is only one byte wide, so the upper byte must be written to the PeriodH register (offset 0x01 from the I^{2}C base address) and the lower byte to the PeriodL register (offset 0x00 from the I^{2}C base address).

The Control register has two important bits; these are the Interrupt Enable bit and the Core Enable bit. The Interrupt Enable bit (bit 6 of the register) is used to enable the core to interrupt the CPU when it needs servicing.

Table 6.3: Register Map of the Open I^2C controller

offset from base	read/write	register
0x00	R/W	PeriodL
0x01	R/W	PeriodH
0x02	R/W	Control
0x03	W	Transmit
0x03	R	Receive
0x04	W	Command
0x04	R	Status

The Core Enable bit (bit 7 of the register) is used to enable the core. The other bits in this register are reserved. The Control register is located at offset 0x02 from the I^2C base address.

Once the I^2C core has been configured, it should be possible to communicate with the audio CODEC. More information can be found on the I^2C controller datasheet on the course website.

The I^2C protocol specifies that after every transfer the slave must acknowledge the receipt of the data by pulling the data line low. I^2C transfers data serially, one bit at a time, one byte per transfer. To send data, you write the byte into the Transmit register in the I^2C controller (offset 0x03), and then write a 1 to the Write bit of the Command register (offset 0x04 of the I^2C controller). It is good practice to wait at this point until the transfer is compete to ensure that the intended recipient of the data acknowledges by waiting until the Transfer In Progress bit of the Status Register (offset 0x04 of the controller) goes low and then checking the Received Acknowledge bit in the Status Register.

I^2C uses the following procedure to send data to a slave. First, the slave's address on the I^2C bus is sent to ensure the chip is on the bus and responding to communication with an ack. The audio CODEC is located at address 0x34 on the I^2C bus (this address includes the R/W bit as its bit 0 as shown in Figure 6.6). As this is the first of a series of byte transfers, it is necessary to generate a start condition. To do this, you would also write a 1 to the Start bit in the Command register when you are writing the Write bit. Then a series of bytes of data are sent, each time with the slave acknowledging. The last transfer in the chain must be accompanied by a stop condition, which is generated by the controller when you write a 1 to the Stop bit in the Command register when you are writing the Write bit. This signifies the end of the transfer. See Figure 6.6 to see a visual explanation of the start and stop conditions, and the audio CODEC datasheet for more information.

The WM8731 uses a series of two byte transfers, each time sending the chip's address with a start condition and the last bit with a stop condition. The two bytes of data that are sent to the CODEC to write one of its registers consist of an address and the register data. The first 7 bits of the 16 bits of data are the address of the register on the CODEC. The following 9 bits are the data to be written to that register. This means that the first byte consists of the address shifted left one bit, with the most significant bit of the data. Refer to Figure 6.6 for a diagram of this communication protocol.

The audio CODEC has many more features than will be used in this lab. It has line and microphone inputs, an analog to digital converter (ADC) used to sample these signals and output digital audio in PCM format, a digital to analog converter (DAC) to convert digital PCM audio streams to analog sound signals for speaker output, as well as several analog and digital filters on both input and output for sound quality improvement. To control these features and configure the chip to function as needed for this lab, the CODEC has several internal registers. A discussion of the pertinent registers for this lab follows. Complete information on all the registers and all the functionality provided by the audio CODEC is available in its datasheet: on uWaterloo Desire2Learn , under Lessons, documentation. The settings needed to be applied to all registers except those left to you to design can be found below in Table 6.4.

- The Analog Audio Path Control register controls the routing of the signal from the inputs to the ADC and the outputs. The CODEC is capable of passing audio signals directly through without any signal processing.

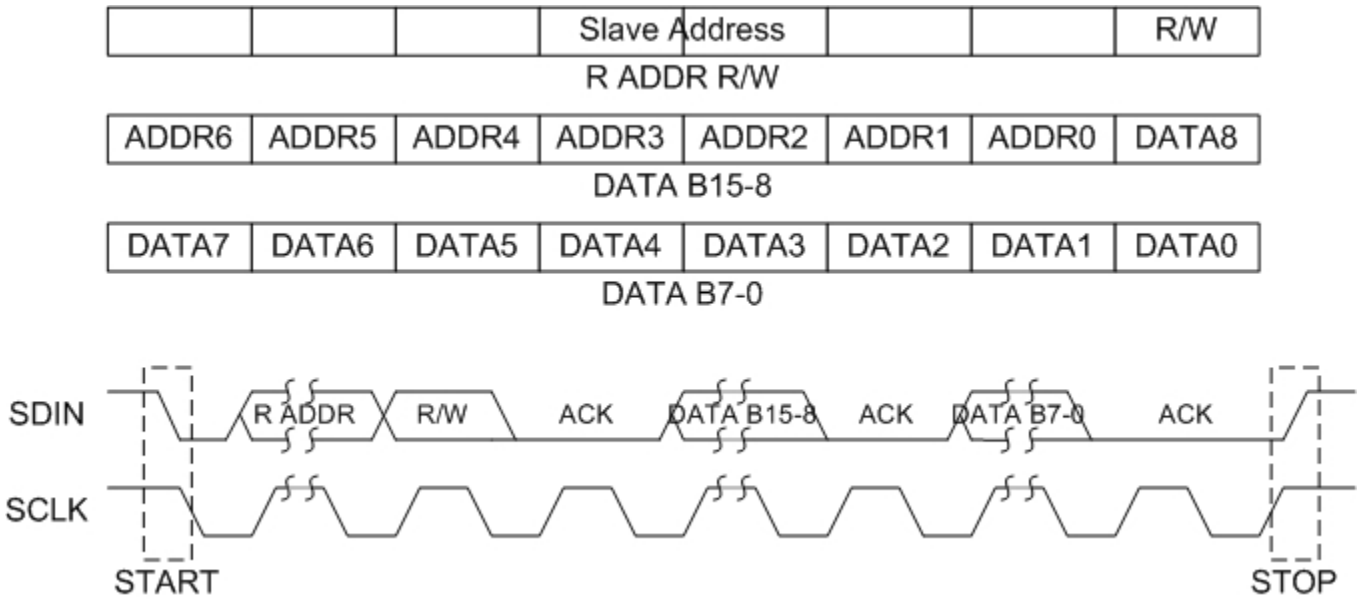

Figure 6.6: I^2C Communication with WM8731

This register controls the bypass of the line input and the microphone, as well as selecting which one of these is connected to the input of the ADC. Most importantly for our purposes, this register also controls whether the DAC output is connected to the line out.

- The Digital Audio Path Control register controls the application of digital filters to the DAC and ADC. The ADC is equipped with a high pass filter to remove any noise in the low frequency range that is below the hearing threshold. The DAC has de-emphasis control for removing traces of the sampling frequency from the signal for improved signal-to-noise ratio in the output. The DAC is also equipped with a software mute that must be turned off in order for the output to pass to the line out.

- The Power Down Control register turns off and on different parts of the CODEC. The ADC and the DAC can be independently shut off, as can the inputs and the outputs and the whole chip. This would be useful in a portable audio player, where the unused parts of the chip could be turned off to conserve power. In this application, it is acceptable to leave everything turned on.

- The Digital Audio Interface Format register controls how the CODEC will interpret data sent to it. Settings include data format, data length, control clock phase, channel swap, and master/slave mode. You must determine what data to write to this register. To do this, you need to know that the data is being sent in left-justified format, with 16 bit samples, the right channel data when the left-right clock is high, no channel swapping, and the chip is operating in slave mode with a non-inverted bit clock.

- The Sampling Control register controls the mode of operation of the chip, which determines what clock rate is expected and how the clock is interpreted to derive internal sample clocks. It also controls the sample rate. You must determine what data to write to this register to properly configure the CODEC. To do this, you need to know that the CODEC will be operating in Normal mode, at the frequency you set in the Audio PLL, with a sampling rate of 44.1 kHz, and the clocks are not divided.

- The Active Control register controls when the audio data interface is actively accepting and processing data.

Table 6.4: Settings for Wolfson Microelectronics WM8731LS Audio CODEC for lab 2 purposes

register	address	data
Analog Audio Path Control	0x04	0x12
Digital Audio Path Control	0x05	0x00
Power Down Control	0x06	0x00
Digital Audio Interface Format	0x07	*You must determine this. Consult the datasheet.*
Sampling Control	0x08	*You must determine this. Consult the datasheet.*
Active Control	0x09	0x01

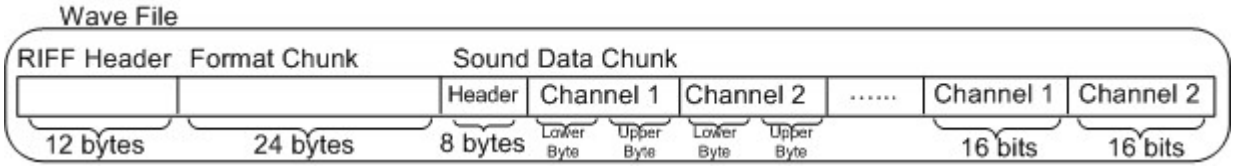

Figure 6.7: Wave File Format

6.6.4 Wave File Format

The wave file format is Windows' native file format for storing audio data. Wave files use the standard RIFF file structure which groups the files contents (sample format, digital audio samples, etc.) into separate block, or chunks of data. The simplest form of wave file contains 2 chunks and is shown in Figure 6.7. The first being a **format chunk**, and the second the **sound data chunk**. The format chunk contains header information for the file such as the sample rate, number of channels, bits per sample etc. For most CD quality audio, and for the purpose of our lab, it will be assumed that all wave files have an audio sample rate of 44.1kHz, audio sample size of 16-bit, and have 2 channels (stereo) of audio. The second chunk contains the raw pulse code modulation (PCM) data that is processed by the audio CODEC. The audio data within stereo wave files is interleaved between the left and right channel. This means that the data within the sound data chunk alternates between data for the first channel and data for the second channel as shown in Figure 6.7.

6.6.5 Using the audio CODEC

Once the audio CODEC has been initialized, audio data can be sent to the CODEC where it is processed and output to the speakers/headphones.

Within the audio component instantiated in the SOPC builder in Part I is a FIFO queue that is used for buffering the audio data that must be sent to the CODEC. A few things to keep in mind when sending data to be processed by the CODEC is that:

- the CODEC is expecting 16-bit audio data
- information in the .wav files is interleaved between the left and right channels, and the FIFO is expecting data to be sent to it in this manner.

To ensure that there is space for the audio data in the FIFO, the FIFO full PIO must return false. Assuming that there is space in the FIFO to write audio data, the 16-bit audio data must be read from the 8-bit data_file sector buffer array containing the audio data. Once this data has been read, it can be written into the FIFO for playback. This is illustrated in the code segment below:

```
UINT16 tmp;  //Create a 16-bit variable to pass to the FIFO

while(IORD( AUD_FULL_BASE, 0 ) ) {}  //wait until the FIFO is not full

tmp = ( buffer[ i + 1 ] << 8 ) | ( buffer[ i ] );  //Package 2 8-bit bytes from the
                                                    //sector buffer array into the
                                                    //single 16-bit variable tmp

IOWR( AUDIO_0_BASE, 0, tmp );  //Write the 16-bit variable tmp to the FIFO where it
                                //will be processed by the audio CODEC
```

6.6.6 Before Compilation

It has been found that the levels of compiler optimization and debugging that are applied to this project prior to compilation have a significant impact on the audio quality when a wave file is played. To ensure the highest audio quality, the level of optimization must be increased and debugging level decreased. To do so, perform the following:

- In the Nios II IDE goto **Project** -> **Properties** and click on **Nios II application Properties** on the left hand side
- On the right hand side select **Optimization Levels** dropdown box and select `Level 3` (Note: one of the TAs in a previous term found an optional setting of 2 was better.)
- Now, on the right hand side select the **Debug Level** dropdown box and select `Off`
- Click **Apply** followed by **OK**

The necessity to increase the optimization and decrease the debugging done by the compiler suggests that there are very few processor cycles available when the audio CODEC is receiving audio data. This should be kept in mind when writing the code.

NOTE: Prior to executing your code, be sure that there is an SD card properly inserted into the SD slot, and that the speakers or headphones are connected to the **green** lineout jack on the DE2. **DO NOT REMOVE THE SD CARD FROM THE SD SLOT WHILE RUNNING YOUR PROGRAM!**

6.6.7 A Few Words About Software Design...

Good software engineering will save you time and effort in this lab. For example, use constant declarations for all constants - in C with a `#define` statement, in C++ with a `type const NAME = value;` statement. Use a header file (.h) for your software. In it, include your constant declarations, enumerated type definitions, and `struct` or `class` definitions. This will greatly simplify debugging your design. Design, implement, and test your software as separate modules. Write functions that can be used for all phases of the lab. As you design your software, think about how you are going to test it. You may find functions in the C/C++ math library (`<math.h>`) useful.

6.6.8 Desired Functionality

Upon completion of Part II, your system should have the following functionality:

- Read a single logical block using the `SD_read_lba` function

- Search the SD card for a file extension defined by the `search_for_filetype` function
- Build a cluster chain for a .wav file using the `build_cluster_chain` function
- Read a sector of data from a file using the `get_rel_sector` function
- Send data to the CODEC to produce output on the speakers
- **When the system is run, it should be able to locate the first .wav file on a SD card and play it through the audio CODEC to the speakers/headphones**

6.7 Part III - The Complete System

In Part III, the system is to be provided with the user interface. The user should be able to browse to different wave files on the file system, and also start and stop playing the files using the push buttons. The user should be able to select the playback mode for the file based on the position of the switches. The playback modes for this lab include: normal speed, double speed, half speed, delay channel, and reverse play. A brief description of each playback mode is provided. In addition, information about the file currently playing or to be played should be displayed in the LCD display.

6.7.1 Playback Modes

For this lab there are five different playback modes to be implemented. These are named playback modes rather than filters because they should only change the order or rate of how the audio data is fed into the audio CODEC rather than altering or manipulating the data that is fed into the CODEC. The playback mode that a file will be played with should be determined by the position of the dip switches. Each modes is explained in more detail below:

Normal Speed This is the default playback mode for the audio player. It is the mode that was implemented in Part II.

Double Speed The file should be played in half the time that it takes for the file to play at normal speed. To implement this, only every other sample should be sent to the CODEC. Be sure to keep in mind that the data within the wave file is interleaved between the left and right channels. The stereo sound should be maintained, while increasing the play speed. **NOTE**: This mode may sound slightly distorted on playback because not all of the audio data is being played.

Half Speed The file should be played in twice the time that it takes for the file to play at normal speed. To implement this, every packet of audio data should be sent to the CODEC twice.

Delay Channel This playback mode should delay one channel a full second behind the other channel. To do so, one channel can be played at normal speed while the other channel is played from a buffer that stores a few seconds worth of audio data for the lagging channel.

Reverse Play This mode should play the entire file in reverse starting from the last audio sample to the first.

6.7.2 LCD Display

The LCD display should be used to provide relevant information to the user. While the user is browsing the file system for a file to play, the short 11 character file name of the file currently being browsed should be displayed on the top line of the LCD display, and the playback mode according to the current position of the dip switches should be displayed on the second line.

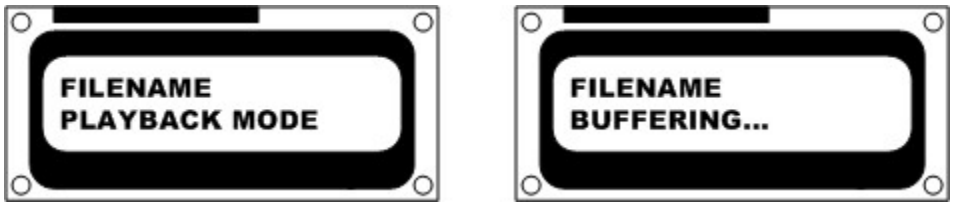

Figure 6.8: LCD Display

An important note to make, is that while the system is building a cluster chain, a significant amount of SD card reads must be performed to buffer the data. During this time it may appear that the system has crashed, when it is in fact operating properly taking time to buffer the file. This fact should be displayed on the LCD display when the system is using the `build_cluster_chain` function. Examples of the LCD display are provided in Figure 6.8.

6.7.3 Desired Functionality

The finished system should included the following functionality:

- Push button 2 should allow the user to cycle forward through files with the .wav file extension on the SD card. If the button is pressed when the user is on the last .wav file of the SD card, the system should loop back to the first .wav file on the SD card
- Push button 3 should allow the user to cycle backward through files with the .wav file extension on the SD card. If the button is pressed when the user is on the first .wav file on the SD card, the system can either remain on the first .wav file of the SD card, or loop around to the last .wav file on the SD card
- Push button 1 should allow the user to play the file that is currently being browsed to within the file system with the currently selected playback mode
- Push button 0 should allow the user to stop playing a file that is currently playing. Pressing this button while no file is playing should have no effect
- The position of the dip switches when the play button (push button 1) is pressed determines the playback mode that will be applied to the file as it is played
- The LCD display should display the short 11 character file name of the file currently being browsed to within the file system on the first line, and the playback mode according to the current position of the dip switches on the second line. When the system is building a cluster chain there should be a indication of this on the LCD display to indicate that the system is processing
- When a file is playing, only the stop button (push button 0) should be enabled, the other buttons should be disabled.

6.7.4 Intermediate Design Stages

You may wish to consider approaching the software design in stages. The following intermediate design stages may be useful.

1. Write a program that accepts interrupts from the push buttons and dip switches, and display an output on the LCD display.
2. Write a program that browses through the files on the SD card only applying the normal speed filter.
3. Then modify your code to fully implement the desired functionality.

6.8 Testing, Debugging, and Going Further

Test your design thoroughly to ensure that it meets the required functionality as described herein. This lab is more complex than those previous, so if you have questions regarding the lab requirements and the expected functionality of your system, do not hesitate to ask a TA.

You will doubtlessly encounter some bugs in your software and hardware. Choose a reliable and effective debugging strategy and record in your lab diary all bugs and corrective measures. Check the lab FAQ on the course webpage and the course newsgroup for solutions to typical problems. You may wish to put `printf` statements in your code to verify program flow and execution. But remember to remove all debug output (especially `printfs`) before your demo. Review the debugging suggestions given in the course notes.

6.9 Deliverables

The deliverables described in Section 2.3.5 provides the basic requirements for the deliverables. The remainder of this section provides a description of the content for this lab.

6.9.1 Demonstration

The demonstration required for this lab study is as described in Section 2.3.5

6.9.2 Report for Lab 2

In your report, you should have the following sections.

- The general deliverables for all reports as described in Section 2.3.5.
- An overview of your complete system design, using block diagrams, pseudocode, prose, or a mixture of some or all of these. (This section should focus on the parts of the design that were completed by you, and should not be a rewording of what is given in the lab manual.)
- A discussion of the issues associated with one aspect of your hardware design or the software interfacing part of your design, for example, the choice of interrupt priorities or issues with respect to interfacing to the buttons, switches or LCD display.
- An indication of one hardware or software interfacing design decision you made given the above issues, and at least one alternative which you discarded.
- A description of the testing and debugging strategy you used for one part of your hardware or software interfacing design and at least one alternative strategy that you discarded. Indicate any intermediate stages used in your design.
- A discussion of the issues that impacted on the audio playback performance for each of the playback modes.
- A discussion of the issues associated with writing efficient software so as to meet the performance requirements associated with the system.
- An indication of one software design decision you made given the above two performance issues, and at least one alternative which you discarded.

- A description of the testing and debugging strategy you used for one part of your software with regards to the audio playback, and at least one alternative strategy that you discarded. Indicate any intermediate stages used in your software design.
- A brief discussion of a possible future extension to the lab. Given more choice for a final project (and possibly 4-5 weeks in which to do it) and the same equipment, what would you have rather done?

The "design decision" sections should include a description of the pros and cons of your choice, with respect to efficiency, ease of implementation, and ease of testing/debugging.

Note that the "testing" and "debugging" sections ask for your strategy. This may be a story about how you debugged a specific problem that you ran into. It may also be a decision you made about how you were going to test a specific portion of your functionality. You should keep a lab diary so that you can easily include this in your report. The report should contain a summary of your test and debug approach, rather than 10 pages rewritten or photocopied from the diary. The chances are incredibly small that everything compiled, simulated, downloaded, and ran perfectly the first time, and that you never ran into a single bug, problem, or difficulty. However, even in this case, you must still show that you had in place a logical, systematic strategy for testing your hardware, software, and integration.

6.9.3 Code Submission

An electronic copy of the C code that you have used during the demonstration. This code is to be submitted through the uWaterloo Desire2Learn system. (We reserve the right to use an automated procedure to compare your code to other code submitted for this lab study.)

Appendix A

Nios II Tutorial

A.1 About this Tutorial

A.1.1 Introduction

This tutorial[1] introduces you to the Nios II® embedded processor. It shows you how to use the SOPC Builder and the Quartus® II software to create and process your own Nios II system module design that interfaces with components provided on the Nios II development board. Associated with this tutorial is Appendix B that provides all of the screen shots referred to in this tutorial.

This tutorial is for Nios II novices or users who are new to using embedded systems in PLDs. The tutorial guides you through the steps necessary to create and compile a 32-bit Nios II system module design, called **myESystem**, and download it into the Nios II development board. This simple, single-master Nios II system module has a Nios II embedded processor and associated system peripherals and interconnections.

One way to approach this system, is to consider the structure illustrated in Figure A.1. In this figure the board is illustrated with serial drivers, a display driver a SDRAM component and the large (blue) rectangle representing the FPGA that is on the DE2 board[2].

We could use various tools to compile and instantiate various elements within the FPGA. However, this term we will be specifying those elements in VHDL or will be using a MegaWizard to create the VHDL for us. There are 3 components illustrated in the FPGA (in Figure A.1): the phase-locked loop (PLL), a custom VHDL component the EGM (used in lab study 1) and the large yellow component labelled as "specified in the SOPC builder".

This tutorial includes the following sections:

- Use of the SOPC builder to specify a controller (processor and I/O system) that provides access to most of the DE2 features that are used this term in the lab studies. You should be able to use the resulting system as a starting point for the lab studies this term.
- Use of the PLL MegaWizard to specify a phase-locked-loop so that we can use the SDRAM reliably as our system memory.
- Use of Quartus II to compile and download the resulting design into a DE2 board.

Figure A.2 illustrates the system to created in this tutorial. This figure illustrates the elements of the board in an abstract manner and does not attempt to replicate the actual board layout.

[1] draft 10.1.sp1.30
[2] Of course the DE2 board has many more components

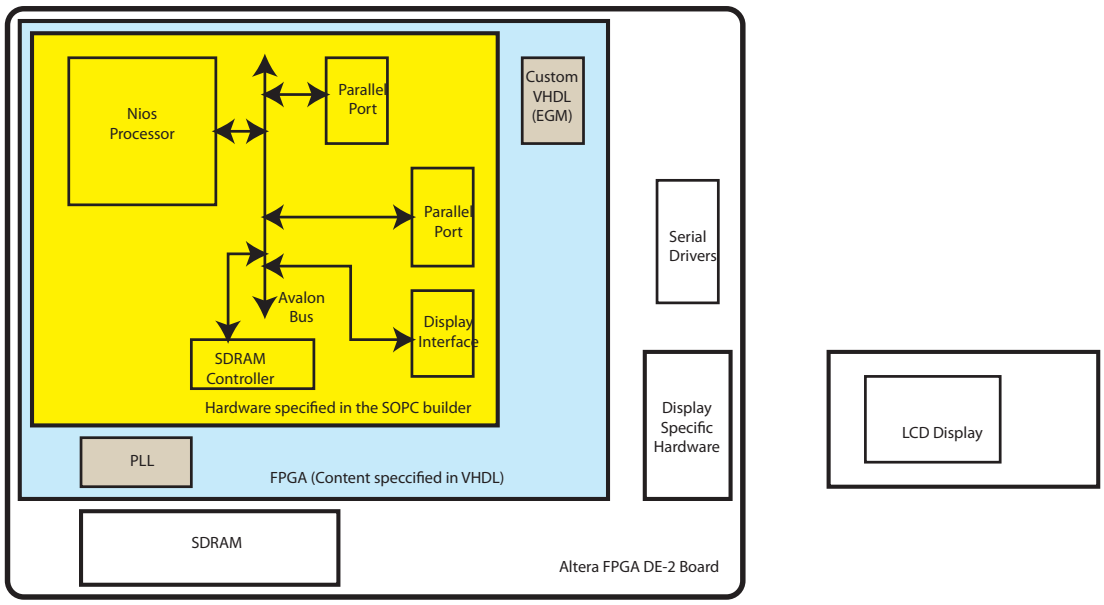

Figure A.1: Structure for systems developed using the SOPC builder

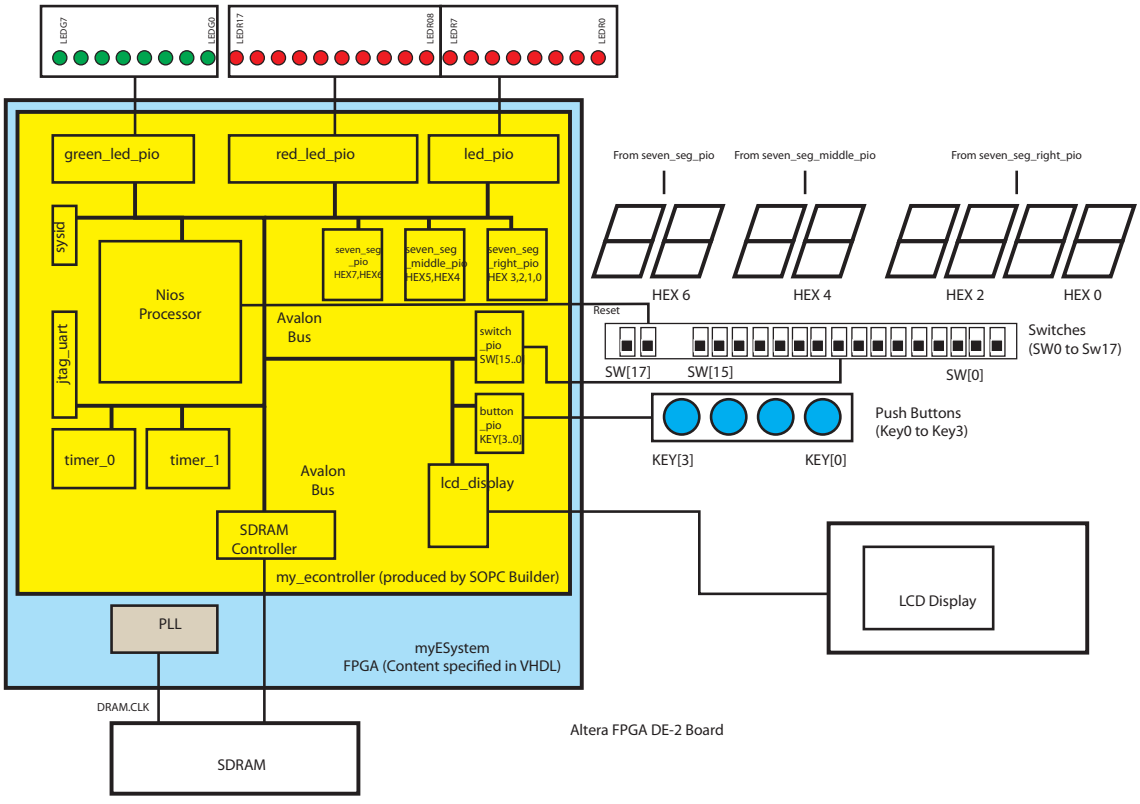

Figure A.2: Structure for systems developed in this Tutorial using the SOPC builder

After you create the **myESystem** design and connect to external pins, you can download it to the Altera® Cyclone II™ device on the Nios II development board. The external physical pins on the Cyclone II device are in turn connected to other hardware components on the Nios II development board, allowing the Nios II embedded processor to interface with RAM, flash memory, LEDs, LCDs, switches, buttons, and other peripherals.

This tutorial is divided into the following three sections:

- **"Design Entry" on page 66** illustrates how to create the Nios II system module in a Block Design File (.bdf) using the Megawizard® Plug-In Manager and the SOPC Builder. This section also illustrates how to connect the system module ports to pins in the Cyclone II device.
- **"Compilation" on page 77** illustrates how to compile the Nios II design using Compiler settings, pin assignments, and EDA tool settings to control compilation processing.
- **"Programming" on page 83** illustrates how to use the Quartus II Programmer and the USBBlaster™ cable to download the design to a Cyclone II device on the Nios II development board.

The Nios II embedded processor version 10.1 includes the SOPC Builder, which supports features such as profiling Nios II systems and interfacing to external processors. This tutorial does not cover these features. For more information on these topics, refer to:

- *AN 391: Profiling Nios II Systems*
- *AN 397: Interfacing to External Processors*

A.1.2 Hardware & Software Requirements

This tutorial requires the following hardware and software:

- A PC running the Windows XP, Windows Vista or Windows 7 operating system
- Nios II software development tools version 10.1, service pack 1 or higher
- Quartus II software version 10.1, service pack 1 or higher
- A Nios II development board, set up as described in the *Nios II Embedded Processor Getting Started User Guide*; The USBBlaster driver, installed as described in the *Quartus II Installation & Licensing for PCs* manual

A.1.3 Tutorial Files

This tutorial assumes that you create and save your files in a working directory on the N: drive on your computer. If your working directory is on another drive, substitute the appropriate drive name. Before you start Quartus, download the files as explained on ACE under Lessons/Labs/Lab Tools Tutorial.

The Nios II embedded processor software installation creates the directories shown in Table A.1 in the **/altera/.../quartus/sopc_builder** directory by default:

A.1.4 More Information

Refer to "Nios II Documentation" in the *Nios II Embedded Processor Getting Started User Guide* for a listing of the documentation provided with the DE2 Development Kit, featuring the Nios II embedded processor or view the on-line literature for any of the Altera components at altera.com/literature/lit-index.html.

Table A.1: Directory Structure

Directory Name	Description
bin	Contains the SOPC Builder components used to create a system module.
components	Contains all of the SOPC Builder peripheral components. Each peripheral has its own subdirectory with a class.ptf file that describes the component.

A.2 Design Entry

The following tutorial sections guide you through the steps required to create the **myESystem** project, and then explain how to create a top-level BDF that contains the Nios II system module. You create and instantiate the Nios II system module using the SOPC Builder.

The instructions in this section assume that you are familiar with the Quartus II software interface, specifically the toolbars. Refer to Quartus II Help for information that is not included in the tutorial.

A.2.1 Create a Quartus II Project

Start the Quartus II Software

In this section, you start the Quartus II software and begin creating your project.

To start the Quartus II software, use one of the following methods:

Choose **All Programs** → **Altera** → **Quartus II** (Windows Start menu).

or

Type `Quartus` and press the **Enter** key at the command prompt.

Create a Project

Before you begin. you must create a new Quartus II project. With the **New Project** wizard, you specify the working directory for the project, assign the project name, and designate the name of the top-level design entity. To create a new project, perform the following steps:

1. Choose **New Project Wizard** (File Menu).
2. Click **Next** in the introduction but please do not turn off the introduction page as this is set for the computer, and the intro page may be necessary for others who follow.
3. Specify the working directory for your project. **Do not have any spaces (blank characters) in the directory path. This also means do not use directories such as "My Documents".**
4. Specify the name of the project and the top-level design entity. This tutorial uses **myESystem**. See FIGURE B.2[3].
5. Click **Next**, Click **Yes** if you are asked to create the working directory.
6. Click **Next**

[3]You will note that some figures in Appendix B are not referred to in the text. The unreferenced screen shots are provided to prove added steps between the labeled figures.

7. Select **Cyclone II** in the Family drop down menu
8. In the Filters box, select FBGA, 672 pins, and speed 6. Choose EP2C35F672C6 (the one with 33216 LEs) under Available Devices. (see FIGURE B.3.)
9. Click **Next**
10. Leave all the EDA tools unchecked, click **Next**
11. Note the summary and click **Finish**.

You have finished creating your new Quartus II project. The top-level design entity name appears in the Hierarchies tab of the Project Navigator window. See FIGURE B.6.

A.2.2 Create a Nios II System Module

This section describes how to create the top-level Block Design File (**.bdf**) that contains a Nios II system module. After creating a design file, you use the SOPC builder to create the Nios II embedded processor, configure system peripherals, and connect these elements to make a Nios II system module. Next, you connect the Nios II module ports to the Cyclone II device pins that are connected to hardware components on the Nios II development board.

Section Overview:

Create a New .bdf

In this step you create a new BDF called **myESystem.bdf**. This file is the top-level design entity of the **myESystem** project.

To create a new BDF, follow these steps:

1. Choose **New** (File menu). The **Device Design Files** tab of the **New** dialog box appears automatically.
2. In the **Device Design Files** tab, select **Block Diagram/Schematic File.**
3. Click **OK.** A new **Block Editor** window appears.
4. Choose **Save As** (File menu).
5. The **Save As** dialog box automatically displays the project directory that you entered. You will save the file into this directory.
6. In the **File name** box, type `myESystem` as the name of the BDF, if necessary.
7. Make sure **Add file to current project** is turned on. See FIGURE B.7.

8. Click **Save**. The file is saved and added to the project. Leave the **.bdf** open for the remainder of the Design Entry section.

You are now ready to create your Nios II design. In the following sections, you will use various tools in the Block Editor toolbar. FIGURE B.8 describes the tools that are available in this toolbar.

Start the SOPC Builder

The MegaWizard Plug-In Manager (Tools menu) allows you to create (or modify) design files that contain custom variations of MegaFunctions. The SOPC Builder is a wizard that guides you through the process of creating a Nios II system module or a more general multi-master SOPC module. A complete Nios II system module contains a Nios II embedded processor and its associated system peripherals.

The SOPC Builder helps you easily specify options for the Nios II system module. The wizard prompts you for the values you want to set for parameters and which optional ports and peripherals you want to use. Once the wizard generates the Nios II system module, you can instantiate it in the design file.

Follow these steps to start the SOPC Builder:

1. Double-click an empty space in the Block Editor window. The **Symbol** dialog box appears. See FIGURE B.9.
2. Click MegaWizard Plug-In Manager. The first page of the MegaWizard Plug-In Manager is displayed. (Select create.)
3. Under **Which action do you want to perform?**, select **Create a new custom megafunction variation** and click **Next**. MegaWizard Plug-In Manager appears.
4. Under **Installed Plug-Ins**, select the **Altera** SOPC Builder. (FIGURE B.10.)
5. Specify the following responses to the remaining wizard options:
   - Ensure device family box indicates **Cyclone II**
   - **Which type of output file do you want to create?**
     VHDL
   - **What name do you want for the output file?**
     `C(orN:):yourPath\ANT\my_controller` [4]
6. Click **Next**. The **Altera SOPC Builder - my_controller** wizard begins loading (this may take a while), and when it appears the **System Contents** tab is displayed.

You are now ready to add the Nios II CPU and peripherals to your system.

Add CPU & Peripherals

The Nios II system peripherals allow the Nios II embedded processor to connect and communicate with internal logic in the Cyclone II device, or external hardware on the Nios II development board. Use the SOPC Builder to specify the name, type, memory map addresses, and interrupts of the system peripherals for your Nios II system module.

[4] You could use either C: or N:. If C: is used, then it may be faster, however you will need to copy the resultant directory to your N: drive before leaving the lab. Alternatively, if you use the N: drive the result may be a slower compile.

The following specifications ensure that the **myESystem** design functions correctly on the Nios II development board, and allow you to run the software examples provided in your project's software development kit (SDK) directory.

You will add the following modules to the SOPC Builder:

- Nios 32-Bit CPU (cpu_0)
- System ID (sysid)
- JTAG UART (jtag_uart_0)
- External SDRAM Interface (sdram_0)
- LCD PIO (lcd_display)
- LED PIO (led_pio)
- Seven Segment PIO (seven_seg_pio)
- Button PIO (button_pio)
- Timers (timer_0 and timer_1)
  - System timer, 1 ms (timer_0)
  - High resolution timer, 10 μs (timer_1)
- Parallel Ports (PIOs) for other devices
  - (output) middle digits seven segment displays (seven_seg_middle_pio 16-bits)
  - (output) right (4) digits seven segment displays (seven_seg_right_pio 32-bits)
  - (output) port for remaining red LEDs (red_led_pio 8-bits)
  - (output) port for green LEDs (green_led_pio 8-bits)
  - (input) port for the slide switches (switch_pio 16-bits)
- Phase Locked Loop

Some of the peripherals drive components on the Nios II development board. Peripheral names are shown in parentheses.

After starting the COPS builder in the previous step, the opening screen should be similar to the one shown in FIGURE B.11. On this screen you will note a list of components (or component types) listed in the Component Library (Such are Processors or Interface Protocols. In this tutorial the location of the various components is listed as: CL: Group : Item. One example would be the Nios Processor located at: CL:Processors:NIOS II Processor.

Nios 32-Bit CPU (cpu_0) CL: Processors : NIOS II Processor

To add the Nios 32-Bit CPU, **cpu**, perform the following steps:

1. Ensure the **target device family** is Cyclone II, the clock frequency is 50 MHz, the source is external, pipeline is unchecked, and leave the board field as **Unspecified Board**.
2. Select **Nios II Processor** under **Processors**.(FIGURE B.12.)
3. Click **Add**. The **Altera Nios II - cpu_0** wizard displays.

4. Specify the following options in the **Nios II Core** tab (see FIGURE B.13):
   - **Nios II Selector Guide:** Nios II/s
   - **Hardware Multiply:** None
   - **Hardware Divide:** Unchecked
   - **Documentation:** This button is common to many sections of the SOPC builder and provides a straightforward way to access the Alter documentation.
5. Click **Next**. The Caches and Tightly Coupled Memories tab appears.FIGURE B.14
6. Specify the following settings:
   - **Instruction Cache:** 4 Kbytes
   - **Enable Bursts (Burst Size: 32 Bytes):** Unchecked
   - **Include Tightly Coupled Instruction Master Port(s):** Unchecked
     This preset configuration automatically sets the options in the remaining Nios II wizard tabs. If you want to view these options or to customize the settings, click next or the desired tab.
   - Click **Finish**. You are returned to the **Altera SOPC Builder - my_controller** window. See FIGURE B.15. As a result of adding the Nios II core, SOPC Builder presents an additional tab titled **Nios II More "cpu" Settings**, which allows you to further configure the Nios II core. Error messages appear in the SOPC Builder Messages window. These messages are normal; you will fix them in later steps.

System ID (sysid) CL: Peripherals : Debug and Performance : System ID Peripheral

The system ID peripheral safeguards against accidentally downloading software compiled for a different Nios II system. For example, webserver software cannot run on a Nios II system lacking an Ethernet MAC. If the system includes the system ID peripheral, the Nios II IDE prevents you from downloading programs compiled for a different system. Perform the following steps to add the system ID peripheral:

1. In the list of available components, select **Peripherals** and then **Debug and Performance** and then **System ID Peripheral**.
2. Click **Add** You return to the SOPC Builder System Contents tab, and an instance of the system ID peripheral named **sysid** now appears in the table of available components. The system ID peripheral has no user-configurable options, and therefore it doesn't have a configuration wizard.
3. Once you return to the SOPC Builder main page, you will need to rename the sysid_0 to sysid. This can be done by right-clicking on the name sysid_0 and selecting rename.

The resulting SOPC Builder main page is illustrated in FIGURE B.18.

JTAG UART (jtag_uart_0) CL: Interface Protocols : Serial : JTAG UART

The JTAG UART provides a convenient way to communicate character data with the Nios II processor through the USBBlaster download cable. To add the JTAG UART peripheral, **uart_0**, perform the following steps:

1. Under **Communication** select **JTAG UART** and click **Add**. The **Avalon UART - uart_0** wizard displays.
2. Do not change the default settings (see FIGURE B.19).
3. Click **Finish**. You are returned to the **Altera SOPC Builder - my_controller** window. (FIGURE B.20.)

External SDRAM Interface CL: Memories and Memory Controller : SDRAM : SDRAM Controller

To add the external SDRAM peripheral, **sdram_0**, perform the following steps:

1. Select the SDRAM controller from the component library as listed above and then click **Add** and the SDRAM controller wizard starts (FIGURE B.22)
2. In the **Presets** drop down box select **(Custom)**. Under the **Memory Profile** tab set the **Data Width** to 16 bits See FIGURE B.23. All the other settings can be left the default values.
3. Click **Finish**. You are returned to the **Altera SOPC Builder - my_controller** window.

Set the Reset Vector and the Exception Vector At this point you have added the SDRAM controller, later you will attach the controller to the memory (FIGURE B.24).

Now that we have added memory, we can return to the Nios Processor wizard and eliminate 2 of the errors currently showing on the SOPC Builder front page.

You can re-enter one of the wizards by double clicking on the name of the component. In this case we must double click on the cpu_0 name. This will re-open the Nios II Processor Wizard. (FIGURE B.27)

Once in the wizard, select sdram_0 for both of the vectors (Reset and Exception). leave the offset values as their default values.

Once you return to the SOPC Builder front page you will note that two of the errors have now been resolved.

LCD DISPLAY CL: Peripherals : Display : Character LCD CharacterDisplay

To add the LCD display peripheral, **lcd_display**, perform the following steps:

1. Once selected at the location above, click **Add**. There are no options for the LCD display so it is added directly to the list of components. (FIGURE B.28 AND FIGURE B.29)
2. Right-click **lcd_0** under **Module Name**.
3. Choose **Rename** from the pop-up menu.
4. Rename **lcd_0** as **lcd_display**. Press enter when you are finished typing the new name to save your setting.

LED PIO (led_pio) CL: Peripherals: Microcontroller Peripherals: PIO (Parallel I/O) To add the LED PIO peripheral, **led_pio**, perform the following steps:

1. Select the pio from the Component Library and click **Add** FIGURE B.31. The **Avalon PIO - pio_0 wizard displays**.
2. Specify the following options:
   - **Width:** 8 bits
   - **Direction:** Output FIGURE B.32
3. Click **Finish**. You are returned to the **Altera SOPC Builder - my_controller** window. FIGURE B.33
4. Right-click **pio_0** under **Module Name**.
5. Choose **Rename** from the pop-up menu.
6. Rename **pio_0** as **led_pio**. Press enter when you are finished typing the new name to save your setting.

Seven Segment PIO (seven_seg_pio) To add the seven segment PIO peripheral, **seven_seg_pio**, perform the following steps:

1. Select as in previous step.
2. Specify the following options:
   - **Width:** 16 bits
   - **Direction:** Output ports only
3. Click **Finish**. You are returned to the **Altera SOPC Builder - my_controller** window.
4. Right-click **pio_0** under **Module Name**.
5. Choose **Rename** from the pop-up menu.
6. Rename **pio_0** as **seven_seg_pio**. Press enter when you are finished typing the new name to save your setting.

Button PIO button_pio Location: Peripherals : Peripherals: Microcontroller Peripherals : PIO (Parallel IO)

To add the button PIO peripheral, **button_pio**, perform the following steps:

1. Select as in previous step.
2. Specify the following options (see FIGURE B.35):
   - **Width:** 4 bits
   - **Direction:** Input

   When you select the **Input** option, the screen displays input options.
3. Turn on **Synchronously capture** under **Edge Capture Register**.
4. Select **Any** edge from the pull-down menu.
5. Turn on **Generate IRQ** under **Interrupt**.
6. Select **Edge**. See FIGURE B.35.
7. Click **Finish**. You are returned to the **Altera SOPC Builder - my_controller** window.
8. Right-click **pio_0** under **Module Name**.
9. Choose **Rename** from the pop-up menu.
10. Rename **pio_0** as **button_pio**. Press enter when you are finished typing the new name to save your setting.

After adding these PIOs the SOPC Builder front page should resemble FIGURE B.36

Table A.2: Additional PIOs to Connect with the Other LEDs and Switches

PIO Name	Direction	Width (bits)	Features
green_led_pio	output	8-bits	default values
red_led_pio	output	8-bits	default values
seven_seg_middle_pio	output	16-bits	default values
seven_seg_right_pio	output	32-bits	default values
switch_pio	input	16-bits	no interrupt or edge capture at this time

Timers (timer_0 and timer_1) CL: Peripherals : Microcontroller Peripherals : Interval Timer

Most control systems use a timer peripheral to enable precise calculation of time. The Nios II hardware abstraction layer requires a timer to provide a periodic system clock tick. To add the timer peripheral, **timer_0**, perform the following steps:

1. Select the timer as listed above and click **Add**. The **Avalon Timer - timer_0** wizard displays.
2. Do not change the default settings (see FIGURE B.37):
3. Click **Finish**. You are returned to the **Altera SOPC Builder-my_controller** window.

Repeat the previous steps to add a timer (timer_1) with a 10 μs period. (See FIGURE B.38).

Other PIOS (red_led_pio, green_led_pio, seven_seg_middle_pio, seven_seg_right_pio and switch_pio) Now add the other pios with the characteristics as listed in Table A.2 FIGURES B.39 TO FIGURE B.45.

Now we need to go back and add a bit more to the NIOS. Right click on the NIOS II in the list of items and select edit. The screen shows two fields: reset vector and Exception vector. In these fields select sdram_0.

You are finished adding peripherals. In the remaining Design Entry sections, you will set options in the SOPC Builder.

Specify Base Addresses & IRQs

The SOPC Builder provides the **Auto-Assign Base Addresses** command which makes assigning peripheral base addresses easy. For many systems, including this example, **Auto-Assign Base Addresses** is adequate. However, the address can also be adjusted manually if necessary.

The Nios II hardware abstraction layer interprets low IRQ values as higher priority. The timer peripheral must have the highest IRQ priority to keep the system clock tick as accurate as possible.

To assign the base addresses and IRQs, perform the following steps:

1. Choose **Auto-Assign Base Addresses** (System Menu) to assign functional base addresses to each component in the system.
2. **Choose auto assign IRQ** to specify the IRQ values which is sufficient for this computer. If you wanted to alter the IRQ value, you can click on it and enter a value.

FIGURE B.46 shows the state of the SOPC Builder **System Contents** tab with the complete system.

Generate my_controller and Add It to the Design

Before you compile the Nios II design with the Quartus II software, you must synthesize the design logic (i.e., generate the design).

Before generating the design, perform the following steps:

1. In the **System Generation** tab set the following, (see FIGURE B.47):
   - **Simulation. Create simulator project files:** Unchecked.

   For more information on the simulation files that are generated, refer to AN 351:*Simulating Nios II Embedded Processor Designs*.

To synthesize the design, perform the following steps.

1. Click **Generate** in the SOPC Builder. The SOPC Builder performs a variety of actions during design generation, depending on which options you have specified. For a design which has all available SOPC Builder options turned on, the SOPC Builder performs the following actions:
   - Saves your design when asked (**select yes**)
   - Compiles the library for your system
   - Generates the HDL source files
   - Creates the simulation project and source files
   - Synthesizes the HDL files and produces an EDIF netlist file

During generation, information and messages appear in the message box in the **System Generation** tab. Be patient as this step takes a while to complete.

2. When generation is complete (see FIGURE B.49), the SYSTEM GENERATION WAS SUCCESSFUL message displays. There will be a 2 warnings and a message that appear in the message box window of the SOPC builder. These warnings can be ignored. Click **Exit** to exit the SOPC Builder, which returns you to the **Symbol** dialog box. FIGURES B.48, B.49 AND B.50
3. Click OK. **Exit ...**

For more information on the SOPC Builder, refer to the *SOPC Builder Data Sheet*.

Add the Symbol to the BDF

During generation, the SOPC Builder created a symbol for your Nios II system module. You can add the my_controller symbol to your BDF. After exiting the previous step, you should have the symbol for my_controller attached to your cursor.[5] To add the symbol, perform the following steps:

1. To place the symbol, click an empty space in the **Block Editor** window. The `my_controller` symbol is instantiated in the BDF as shown in FIGURE B.51.
2. Choose **Save** (File menu).

Adding a PLL for the memory clock

The combination of the elements that we have used and the operation of the DE2 board results in a need for a special clock signal for the SDRAM. In this case we need to clock the SDRAM *before* the clock to the Nios Processor. Although this seems to be a significant challenge, we can use a Phase-locked loop to prepare the two clock signals.

We construct the phase-locked loop using another MegaWizard, the ALTPLL wizard.

The wizard is started the same way as we started the SOPC builder, double-clicking in an open space in the .bdf file and then selecting MegaWizard Plug-in Manager. We select create (FIGURE B.52). (Press **Next**.)

Then the following steps are taken:

1. Under the I/O folder select ALTPLL, in VHDL, for the Cyclone II with a file name of **sdram_pll.vhd**. As shown in FIGURE B.53.
2. on page 3, select 50 MHz. FIGURE B.54
3. on page 4, Deselect all boxes (note now one input and one output). FIGURE B.55
4. on page 5, Leave with default values. FIGURE B.56
5. on page 6, Specify Clock c0: 50 MHz, -3 ns delay. FIGURE B.57

[5] If the symbol is not attached to the cursor complete the following steps.

1. Double Click on the empty space in the **Block Editor** window. The **Symbol** dialog box will appear.
2. Under Libraries expand the Project folder and select the `my_controller` symbol.
3. Click **OK** to instantiate the `my_controller` symbol in the BDF. An outline of the `my_controller` symbol is attached to the pointer.

6. on page 7, Add and specify clock c1 (50 MHz, 0 ns delay). FIGURE B.58
7. on page 8, We do not need c2 so confirm it is deselected.. FIGURE B.59
8. on page 9, Leave as default. FIGURE B.60
9. on page 10, Summary information only. FIGURE B.61

When **Finish** is pressed, you are returned to the symbol insert FIGURE B.62 and you can add the PLL to the .bdf as illustrated in FIGURE B.63.

Add Pins & Primitives

To enter input, output, and bidirectional pins and primitives, perform the following steps:

1. Turn on rubberbanding if it is not already turned on by clicking the Use rubberbanding icon on the Block Editor toolbar (refer to Figure B.8 on page 92 for a description of the tools in the Block Editor toolbar).
2. The following steps are easier to perform with the diagram editor in full screen mode (click the full screen button in the toolbar).
3. Click the **Symbol Tool** button on the Block Editor toolbar. The **Symbol** dialog box appears. Opening the **Symbol** dialog box using the toolbar button enables the **Repeat-insert mode** option. This option makes it easy for you to add multiple instances of a symbol.
4. In the **Libraries** list of the **Symbol** dialog box, click the + icon to expand the **libraries** folder (called **c:/.../quartus/libraries**) in the default installation but is actually located at C:\software\eng\ece\altera\...\quartus\libraries on Nexus installations). Expand the **primitives** folder and then expand the **pin** folder.
5. In the **pin** folder, select the **input** primitive. (FIGURE B.65)
6. Click **OK**.
7. Click an empty space four times to insert a total of 4 `INPUT` pin symbols on the left-hand side of the `my_controller` symbol. Position each symbol so that the right side of it touches the `my_controller` (or the `sdram_pll` symbol and the horizontal line in each `INPUT` pin symbol meets a horizontal line on the `my_controller` symbol. Connect these input pins to see Figure B.66:
   - What will be `CLOCK_50` to `inclk0` (on the PLL),
   - What will be `KEY[3..0]` to `in_port_to_the_button_pio[3..0]`
   - What will be `SW[16]` to `reset_n` and
   - What will be `SW[15..0]` to `in_port_to_the_switch_pio`.

   Note, when you attach the pins to my_controller, there will not be names associated with the pin. That is completed in a later step. The names (such as SW[16]) are used to indicate the future connection.

   By positioning the pins using this method (with rubberbanding turned on), you are connecting them to the `my_controller` symbol. If you select one of the pin symbols and move it away from the `my_controller` symbol, a connection line appears between them.

   ⋆ Symbols are (initially) automatically named as `pin_name<number>` in sequence. Press the Esc key when you are done adding the symbols.
8. Repeat steps 1 through 6 to insert and position a total of 21 `OUTPUT` pins and 2 `BIDIR` pins in the file in the locations shown in FIGURES B.67 TO FIGURE B.69. You may wish to take a peek at FIGURE B.71 to see what the final layout will be. Refer to TABLE A.3 to determine where to connect the pins.

9. Double-click a blank space in the BDF. The **Symbol** dialog box appears.
10. In the **Libraries** list of the **Symbol** dialog box, click the + icon to expand the **C:\software\eng\ece\altera\.../quartus\libraries** folder. Expand the **primitives** folder and then expand the **other** folder.
11. In the **other** folder, select the **vcc** primitive and select **Repeat-insert mode**.
12. Click **OK**
13. Click two empty spaces in the BDF next to the OUTPUT pin at the top of the BDF to insert two VCC symbols.
14. Connect the Vcc symbols the two dangling outputs as shown in FIGURE B.71. Connections can be made by clicking on the end of a wire and dragging it to the wire to which it needs to be connected.
NOTE: A small box will appear over the wire when a valid connection can be made.
15. Choose **Save** (File menu).

Name the Pins

You can now name the pins. To do so, perform the following steps:

1. Double click the pin *name*.
2. Enter the new name, as given in TABLE A.3.
3. Press enter to move vertically down to the next pin.
4. Repeat for all vertically aligned pins.

You may also change individual pin names with the following method:

1. With the Selection Tool, double-click the first input pin symbol that you entered. The **General** tab of the **Pin Properties** dialog box appears.
2. In the **Pin name(s)** box, type `clk` to replace the default name of the pin, i.e., to replace `pin_name`.
3. Click **OK**.

For the design to work properly, you must name the pins as shown in Table A.3. Later in this tutorial you will use a **.csv file** to make pin assignments that reflect these names.

NOTE: Vcc is not the name of a pin, but the connection terminal of the Vcc primitive created earlier.

Choose **Save** (File menu), The BDF is complete. See FIGURE B.75.

A.3 Compilation

A.3.1 Overview

The Quartus II Compiler consists of a series of modules that check a design for errors, synthesize the logic, fit the design into an Altera device, and generate output files for simulation, timing analysis, and device programming.

The following tutorial sections guide you through the steps necessary to create Compiler settings, assign signals to device pins, specify EDA tool settings, and compile the design. The compilation tutorial sections include:

Table A.3: Input, Output, & Bidirectional Pin Names

Pin Name	Type	Connected to my_controller Signal
Input Pins		
CLOCK_50	Input	inclk0
KEY[3..0]	Input	in_port_to_the_button_pio[3..0]
SW[15..0]	Input	in_port_to_the_switch_pio[15..0]
SW[16]	Input	reset_n
Internal Wires		
DRAM_CLK	Internal Wire	PLL c0
LCD_ON	Internal Wire	Vcc
LCD_BLON	Internal Wire	Vcc
Output and Bidirectional Pins		
LCD_EN	Output	LCD_E_from_the_lcd_display
LCD_RS	Output	LCD_RS_from_the_lcd_display
LCD_RW	Output	LCD_RW_from_the_lcd_display
LCD_DATA[7..0]	Bidirectional	LCD_data_to_and_from_the_lcd_display[7..0]
LEDR[7..0]	Output	out_port_from_the_led_pio[7..0]
LEDR[15..8]	Output	out_port_from_the_red_led_pio[7..0]
LEDG[7..0]	Output	out_port_from_the_green_led_pio[7..0]
DRAM_ADDR[11..0]	Output	zs_addr_from_the_sdram_0[11..0]
DRAM_BA_0,DRAM_BA_1	Output	zs_ba_from_the_sdram_0[1..0]
DRAM_CAS_N	Output	zs_cas_n_from_the_sdram_0
DRAM_CKE	Output	zs_cke_from_the_sdram_0
DRAM_CS_N	Output	zs_cs_n_from_the_sdram_0
DRAM_DQ[15..0]	Bidirectional	zs_dq_to_and_from_the_sdram_0[15..0]
DRAM_UDQM,DRAM_LDQM	Output	zs_dqm_from_the_sdram_0[1..0]
DRAM_RAS_N	Output	zs_ras_n_from_the_sdram_0
DRAM_WE_N	Output	zs_we_n_from_the_sdram_0
HEX7[7..0],HEX6[7..0]	Output	out_port_from_the_seven_seg_pio[15..0]
HEX5[7..0],HEX4[7..0]	Output	out_port_from_the_seven_seg_middle_pio[15..0]
HEX3[7..0],HEX2[7..0],HEX1[7..0],HEX0[7..0]	Output	out_port_from_the_seven_seg_right_pio[31..0]

NOTE: Vcc is not the name of a pin, but the connection terminal of the Vcc primitive created earlier.

A.3.2 Create Compiler Settings

You can create Compiler settings to control the compilation process. The Compiler settings specify the compilation focus, the type of compilation to perform, the device to target, and other options. This section includes the following steps:

View the Compiler Settings

The **Compiler Settings** menu allows you to select an existing group of Compiler settings for use during compilation, define and save a new group of Compiler settings, specify the compilation focus, or delete existing settings.

To view the default Compiler settings created for the current project, follow these steps:

1. Open the settings dialog box, select **Assignments** → **Settings**.

At this point in the tutorial, the **General** view displays only the default Compiler general settings created by the Quartus II software when the project was initially created. These default settings are given the name of the top-level design entity in the project (**myESystem** See Figure A.3).

Specify the Device Family & Device

The Nios II development board includes a Cyclone II EP2C35F672C6 device. In this section, you will target this device in the Compiler settings.

The **Device** tab of the **Settings** (Assignment) dialog box allows you to select the family and device you want to target for compilation.

To select the device family and device, follow these steps:

1. In the **Settings** (Assignment) category select **Device.**
2. In the **Family** list, select Cyclone II.
3. If you receive a message that asks if you want the Quartus II software to choose a device automatically, click **No**.
4. Under **Target device**, select **Specific device selected in 'Available devices' list**.
5. Under **Show in 'Available devices' list**, select the following options (note that as you change these options from **Any** the number of available devices listed decreases):

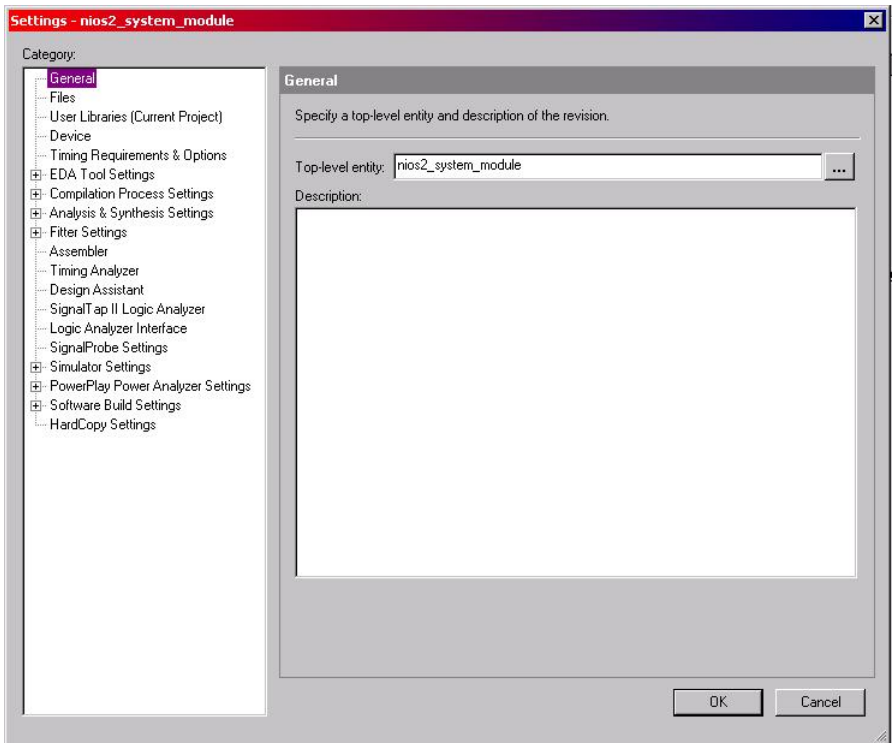

Figure A.3: Default Compiler Settings

(a) In the **Package** list, select **FBGA**.

(b) In the **Pin count** list, select **672**.

(c) In the **Speed grade** list, choose **6**.

6. In the **Available devices** list, select **EP2C35F672C6**. See Figure A.4.

7. To accept the defaults for the remaining Compiler settings, click **OK**.

A.3.3 Assign Signals to Device Pins

During compilation, the Compiler assigns the logic of your design to physical device resources. You can also make pin assignments to direct the Compiler to assign signals in your design to specific pins in the target device. Because the targeted Cyclone II device is already mounted on the Nios II development board, you must assign the signals of the design to specific device pins.

The Quartus II software provides several methods for making pin assignments. You can assign pins individually with the **Assignment Editor** (Assignments menu), or you can assign all necessary pins at once with a .csv file. Because of the number of specific pin assignments to be made for this tutorial, you should use the provided .csv file (**ECE_DE2_Pins_2011.csv**) to make the appropriate pin assignments easily.

This section includes the following steps:

1. "Assign Pins with a .csv file" on page 81

2. "Verify the Pin Assignments" on page 81

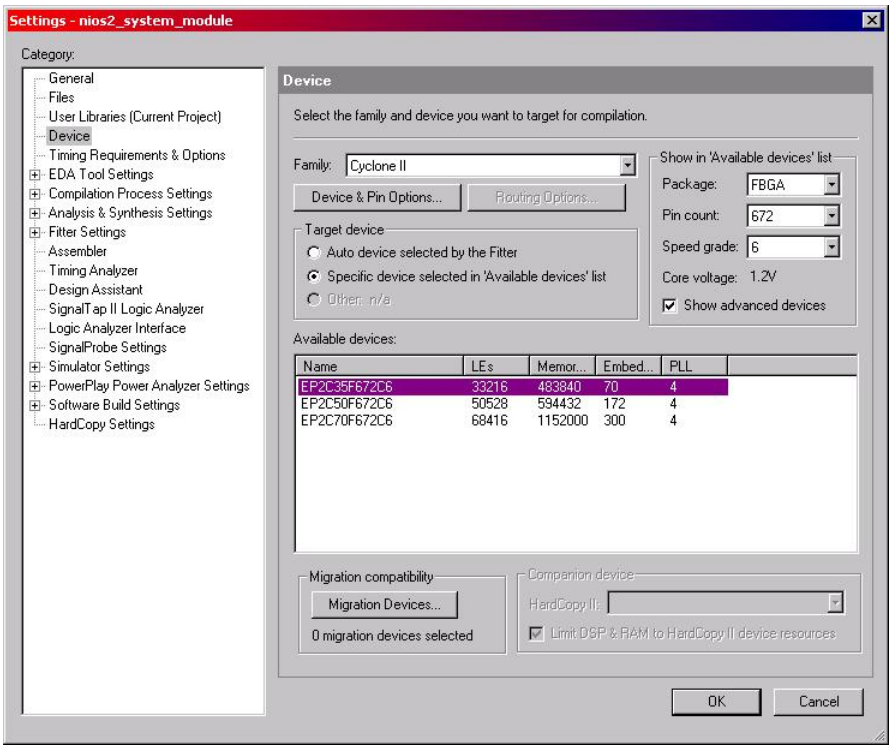

Figure A.4: Device Settings

Assign Pins with a .csv file

To make pin assignments with the provided **ECE_DE2_Pins_2011.csv** file, follow these steps:

1. Open the **Import Assignments** dialog box under the Assignments Menu
2. Type the file location **file location** in the File Name box or browse to the location the **ECE_DE2_Pins_2011.csv** file
3. Click **OK**
4. You should see a message stating that the import was completed

This file contains all of the pin assignments for the DE2 board, including those not used in this project. Adding this additional information will not affect the project in anyway, but the information is already there if you wish to interface with additional I/O's on the board.

Verify the Pin Assignments

To verify the pin assignments, follow these steps.

1. Enter the **Assignment Editor** dialog box under the Assignments menu. All of the pin assignments for the device will be labelled there. See Figure A.5.
2. Close the Assignment Editor window without making any changes.

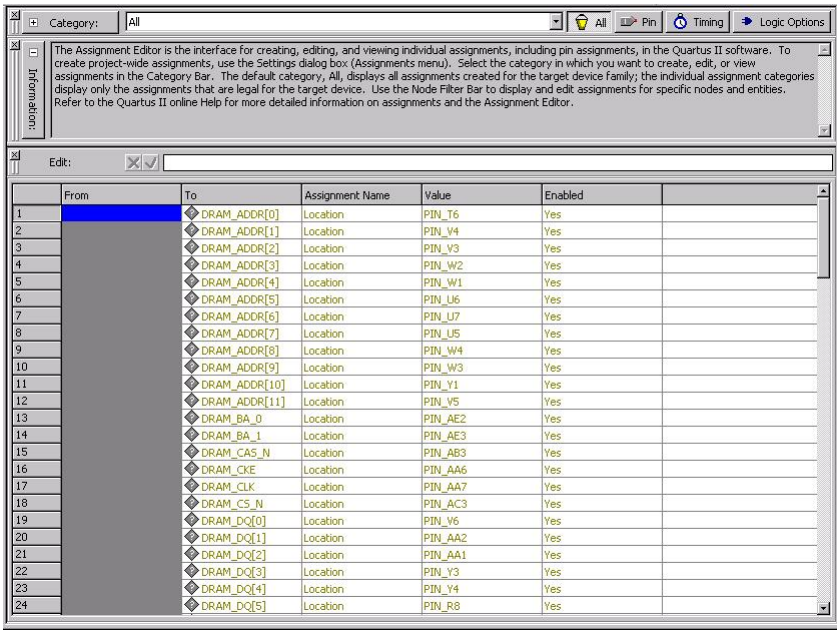

Figure A.5: Verify Pin Assignments

A.3.4 Specify Device, Programming & EDA Tool Settings

Before compiling the design, you must specify options that control the use of unused pins, and EDA tool settings. This section includes the following steps:

1. "Reserve Unused Pins" on page 82
2. "Specify EDA Tool Settings on page 82

Reserve Unused Pins

To specify options for reserving unused pins, follow these steps:

1. Open the **Device** dialog box under the Assignments Menu
2. Press the **Device & Pin Options** button
3. Click the **Unused Pins** tab
4. Under **Reserve all unused pins**, select **As input tri-stated**. See Figure A.6.
5. **Click OK**

Specify EDA Tool Settings

To specify the appropriate EDA tool settings for use when compiling a synthesized design, follow these steps:

1. Choose the **EDA Tool Settings** dialog under the Assignments $->$ Settings Menu.
2. Ensure that all of the tool settings are set to <**None**>. See Figure A.7.

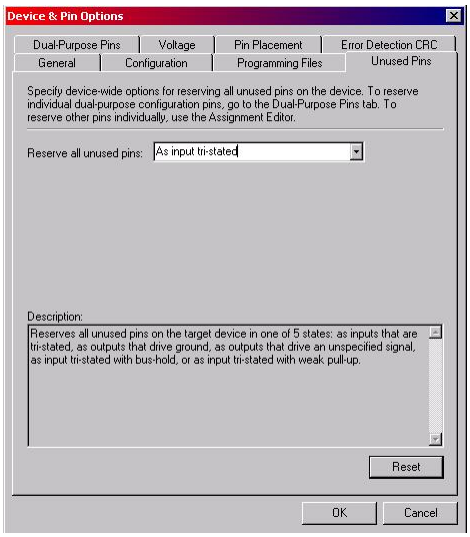

Figure A.6: Unused Pins

3. Click **OK**.

A.3.5 Compile the Design

During compilation the Compiler locates and processes all design and project files, generates messages and reports related to the current compilation, and creates the SOF and any optional programming files.

To compile the **myESystem** design, follow these steps:

1. Choose **Start Compilation** from the Processing menu.

   The Compiler immediately begins to compile the **myESystem** design entity, and any subordinate design entities, using the **myESystem** Compiler settings. As the design compiles, the Status window automatically displays, as a percentage, the total compilation progress and the time spent in each stage of the compilation. The results of the compilation are updated in the Compilation Report window. The total compilation time may be 10 minutes or more, depending on your system.

   The Compiler may generate one or more warning messages that do not affect the outcome of your design. **You can safely ignore most of the warning messages it produces unless you encounter problems**.

2. If the Compiler displays any error messages, you should correct them in your design and recompile it until it is error free before proceeding with the tutorial. You can select the message and choose **Locate** (right button pop-up menu) to find its source(s), and/or choose **Help** to display help on the message.

3. When compilation completes, you can view the results in the myESystem Report window.

A.4 Programming

A.4.1 Overview

In Session 1, the programming step will be demonstrated, however you will perform the programming step in Session 2.

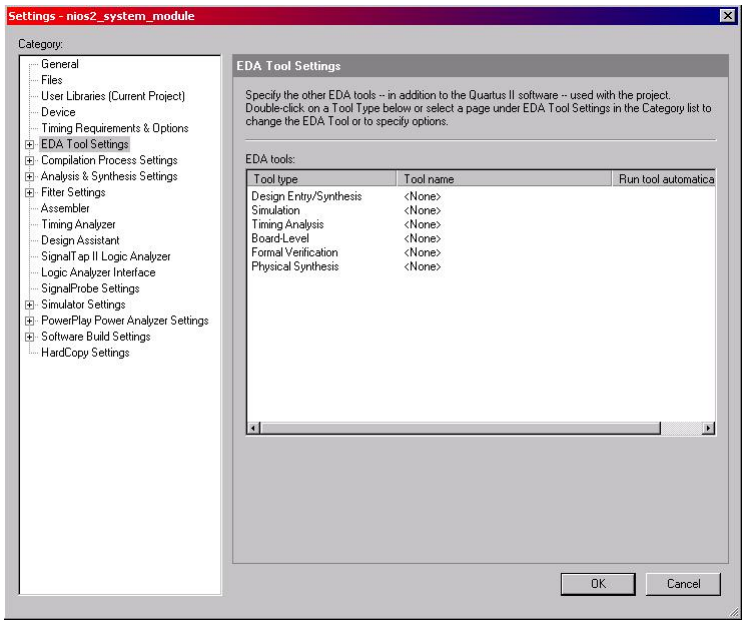

Figure A.7: EDA Tool Settings

After a successful compilation, the Quartus II Compiler generates one or more programming files that the Programmer can use to program or configure a device. You can download configuration data directly into the Cyclone II device with the USBBlaster communications cable connected to the JTAG port (J9), located next to the power supply connector (J8) on the Nios II development board.

A.4.2 Configure a Cyclone II Device

Once you have properly connected and set up the USBBlaster cable to transmit configuration data over the JTAG port, you can configure the Cyclone II device on the Nios II Development board with your design.

To configure the Cyclone II device on the Nios II development board with the **myESystem** design, follow these steps:

1. Choose **Programmer** (Tools Menu).
2. Normally the Programmer window opens the Chain Description File (.cdf), as shown in Figure A.8 associated with your design[6].
3. If it is not already on, press the red power button (SW18) to turn the board on.
4. In the **Mode** list of the Programmer Window, make sure **JTAG** is selected.
5. Click the button labelled **Hardware Setup**.

6

(a) Choose **Save As** (File menu).

(b) In the **Save As** dialog box, type **myESystem** in the **File Name** box.

(c) In the **Save as type** list, make sure **Chain Description File** is selected.

(d) Click **Save**.

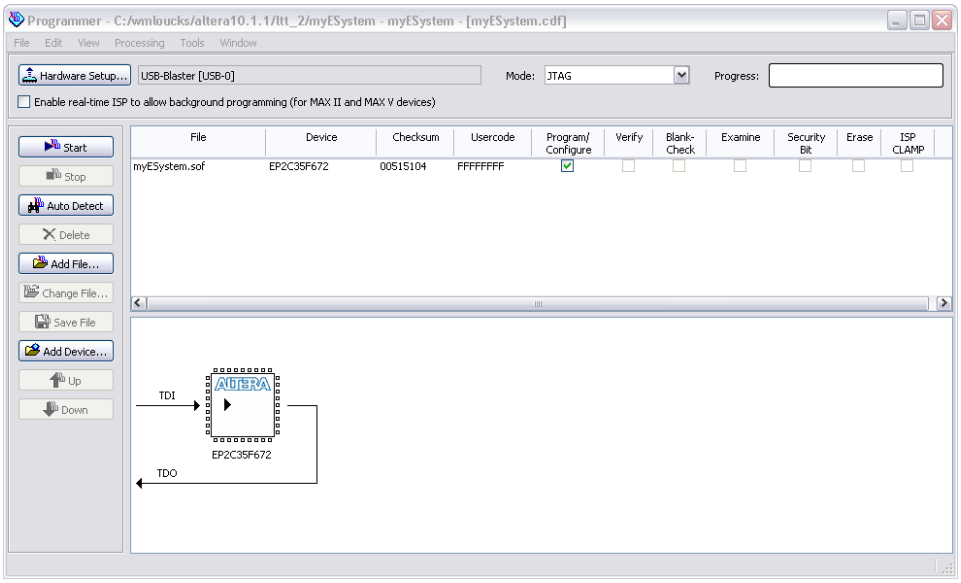

Figure A.8: JTAG Chain

6. If the USBBlaster hardware is not already listed – click **Add Hardware**, select **USBBlaster** and, if necessary, select **USB-0** as the port.

7. Click **OK**.

8. From the **Currently selected hardware:** drop down list, select **USBBlaster [USB-0]**.

9. Click **Close** to exit the **Hardware Setup** window.

10. If the .sof file is not already listed – click the **Add File** icon on the toolbar (or in the right-click menu). The **Select Programming File** dialog box appears.

11. Specify the **myESystem.sof** file in the **File name** box.

12. Click **Open**. The SOF is listed in the Programmer window.

13. In the Programmer window, turn on **Program/Configure**. See Figure A.8.

14. On the Nios II development board, make sure the **RUN/PROG switch** (SW19), beside the LCD panel, is in the **RUN** position. Figure A.9 illustrates the correct configuration of the JTAG switches. Also make sure that SW[16] is "up" (the myESystem reset switch).

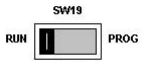

Figure A.9: Nios II Development Board JTAG Switch Configuration

15. Click **Start** (Processing menu, or icon on Programmer toolbar). The Programmer begins to download the configuration data to the Cyclone II device. The **Progress** field displays the percentage of data that is downloaded.

A.4.3 Software Development and Testing for your Nios II System

In this section you will start the Nios II software build tools and compile a simple C language program for testing your system. This section presents only the most basic software development steps to demonstrate software running on the hardware system you created in previous sections.

In this section you will perform the following actions:

1. Create a New C/C++ Application Project on page 86
2. Compile the Project on page 87
3. Run the Application on page 87

To perform this section you must have successfully generated a SOPC Builder information file (**.sopcinfo**) previously.

Create a New C/C++ Application Project

In this section you will create a new Nios II C/C++ Application Project. Perform the following steps:

1. Start the Nios II Software Build Tools. Choose **Start/All Programs/Altera/Nios II EDS 10.1sp1/Nios II 10.1 Software Build Tools**.
2. When the **Workspace Launcher** dialog box appears, use the Browse button to move to your project folder on your N: drive and create a new (software) folder using the Make New Folder button to store your application software within your project folder. Click **OK** to accept this folder as your workspace location.
3. Upon opening the Nios II Software Build Tools, a welcome window may appear, this window can be closed/ignored.
4. Choose **File/New/Nios II Application and BSP from Template** to open the new project wizard.
5. Click the **...** button beside the **SOPC Information File name:** text box. The Open window should appear.
6. Browse to the location of your project folder and select **MyController.sopcinfo**.
7. Click **Open**. Your computer may take a minute to return to the previous window and process the file. Once the file has been processed, your CPU name should appear as **cpu**.
8. Name your project **BoardTest** by entering the name in the **Project name:** text box.
9. Select **Board Diagnostics** in the **Project template** list.

10. Click **Finish**. The tool will create the files necessary for the application program (**BoardTest**) and the board support package (**BoardTest_bsp**).

The Project Explorer will list two new projects on the left-hand side of the workbench: **BoardTest** and **BoardTest_bsp**. **BoardTest** is your C/C++ application project, and **BoardTest_bsp** is a board support package that encapsulates the details of the Nios II system hardware. If you open the **board_diag.c** source file within the **BoardTest** project, you will find a relatively large but well commented C program that allows you to test two seven segment displays, 8 LEDs, a UART, a LCD display, and 4 switches. Although this application has not been designed specifically for the DE2 Board, it is compatible with it (provided that you have named your hardware components of your design appropriately).

The **board_diag.c** file illustrates the use of several peripherals used in this course. The program is reasonably well commented. It is highly recommended that you spend some time reviewing this code an analyzing its behavior. In particular, note the use of **IOWR_ALTERA_AVALON_PIO_DATA()** and **IORD_ALTERA_AVALON_PIO_DATA()**. These macros will be used extensively to communicate with devices connected to parallel I/O ports on your system. The addresses of all important data and control registers are conveniently defined for you in **system.h** in the **BoardTest_bsp** project.

Compile the BSP and the Application

To compile the project, perform the following steps:

1. Right click the **BoardTest_bsp** project in the Project Explorer and select **Nios II/Generate BSP**.
2. Click **Project/Build All** in the main menu system. This step should take a significant amount of time to complete.
3. When the compilation completes, the message "BSP build complete" and "BoardTest build complete" appears in the Console view.

Running the Program

In this section you will download the program to the DE2 Board and execute it. Prior to attempting to run your application, make sure you have configured your Cyclone II FPGA on the DE2 Board with your system. If you need to configure your FPGA, you can do so by selecting **Nios II/Quartus II Programmer** from the main menu system. You will need to select **Add File** and then select the appropriate .sof file for your system. Once you are ready to configure your FPGA, press **Start**.

To download software applications to the target board, perform the following steps:

1. Right click the **BoardTest** project and choose **Run As/Nios II Hardware**. The tool will download the application program to the system on the DE2 board and then automatically start executing it.

   When the hardware starts executing the program, you will see a menu system in the Console view. You may interact with the menu system using the keyboard on your computer. Test out the functionality of your DE2 Board using the menu options provided.
2. Click **Terminate** (the red square) on the toolbar at the upper-right hand corner of the Console view to terminate the run session. When you click **Terminate**, the tool disconnects from the target hardware and leaves the Nios II processor running.
3. To shutdown your hardware, if applicable, go back into Quartus II and click the **Cancel** button in the open dialog box and press the power button on the DE2 board.

For information on running and debugging programs on target hardware, refer to the *Nios II Software Build Tools for Eclipse* manual online.

Appendix B

Nios II Figures

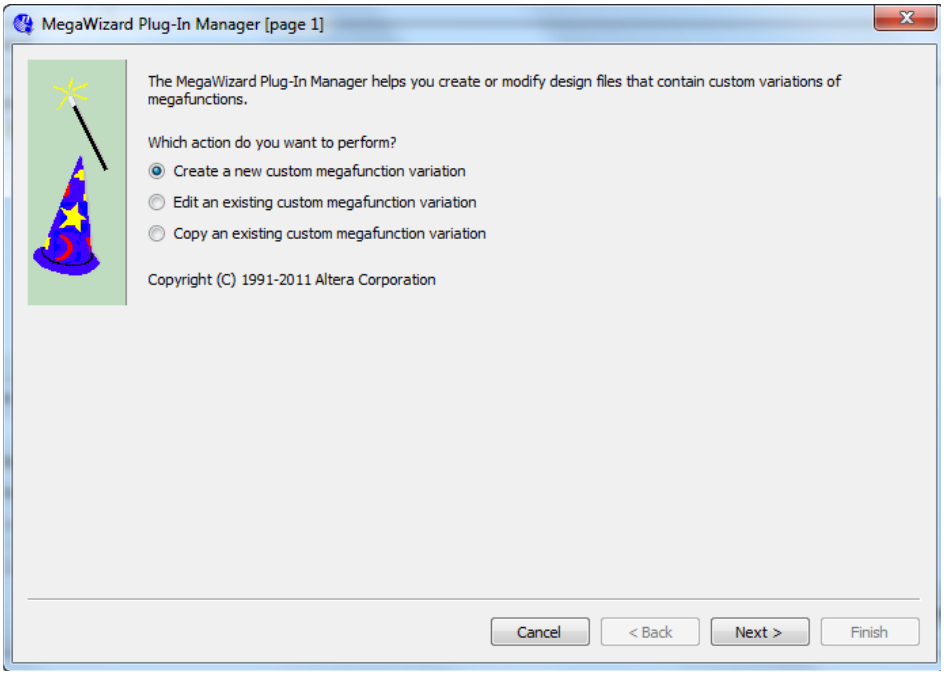

Figure B.1: MegaWizard entry page, select create

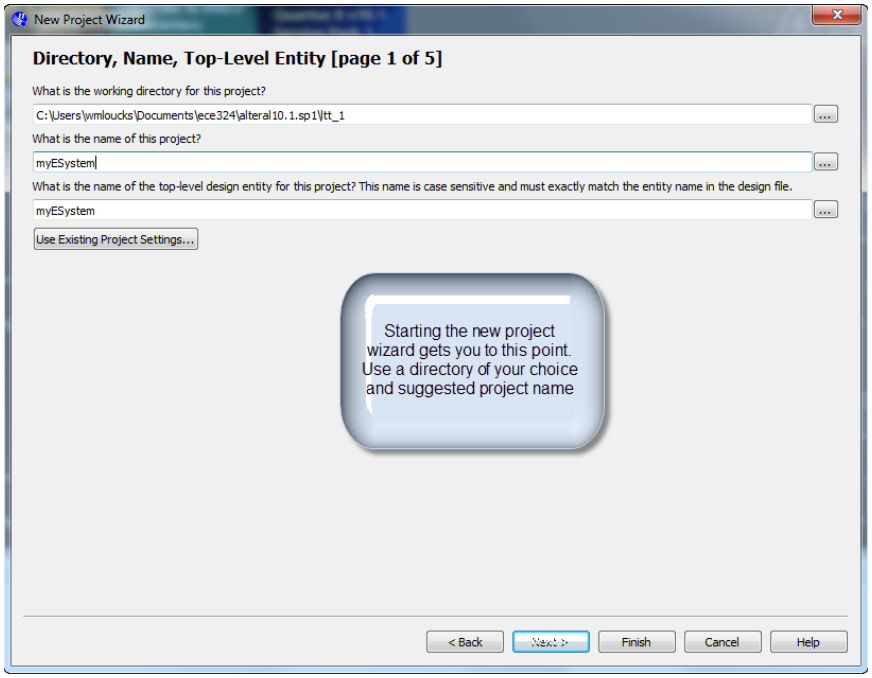

Figure B.2: Specifying the Project Name and Directory

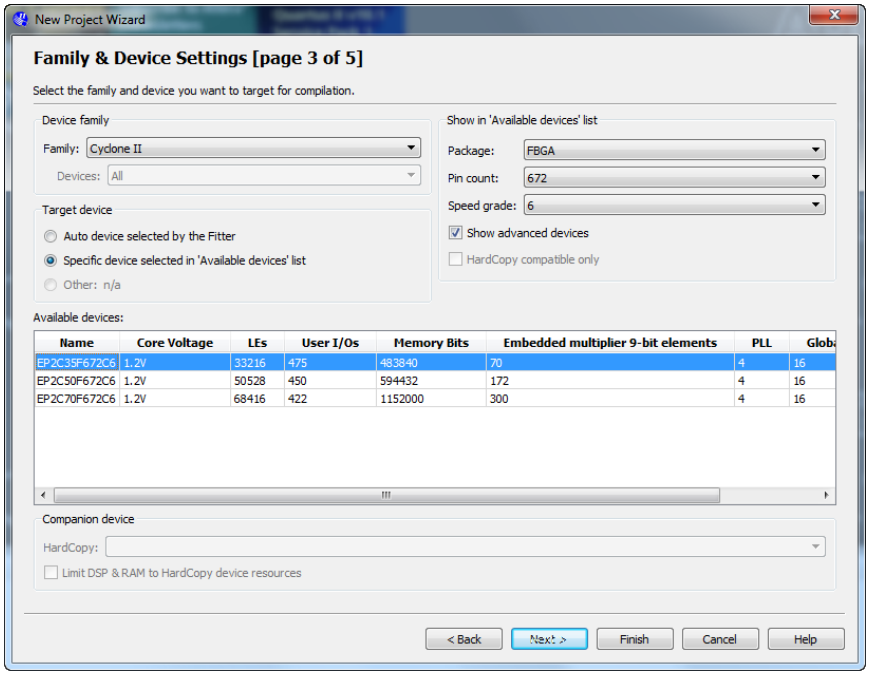

Figure B.3: Specifying the device

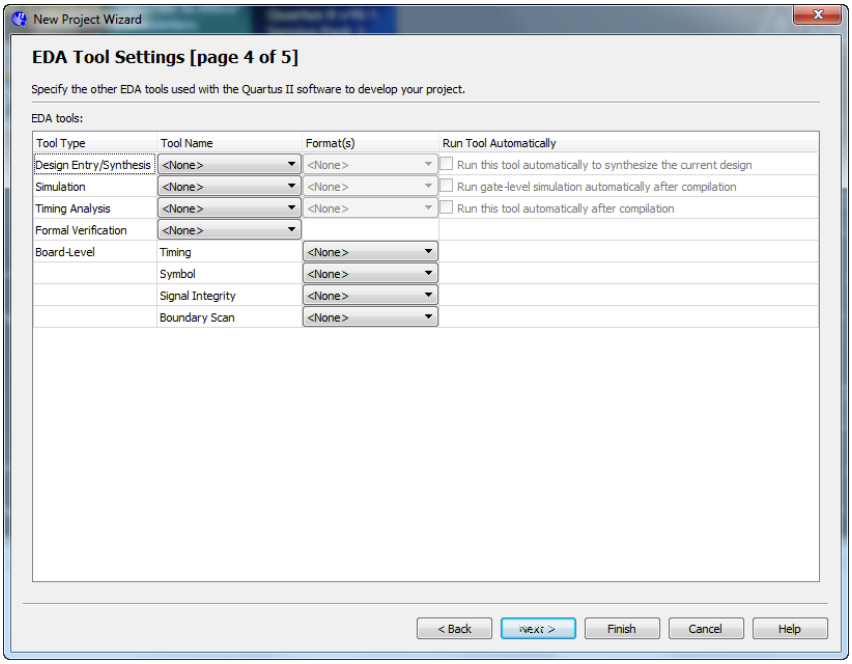

Figure B.4: EDA Tool specification

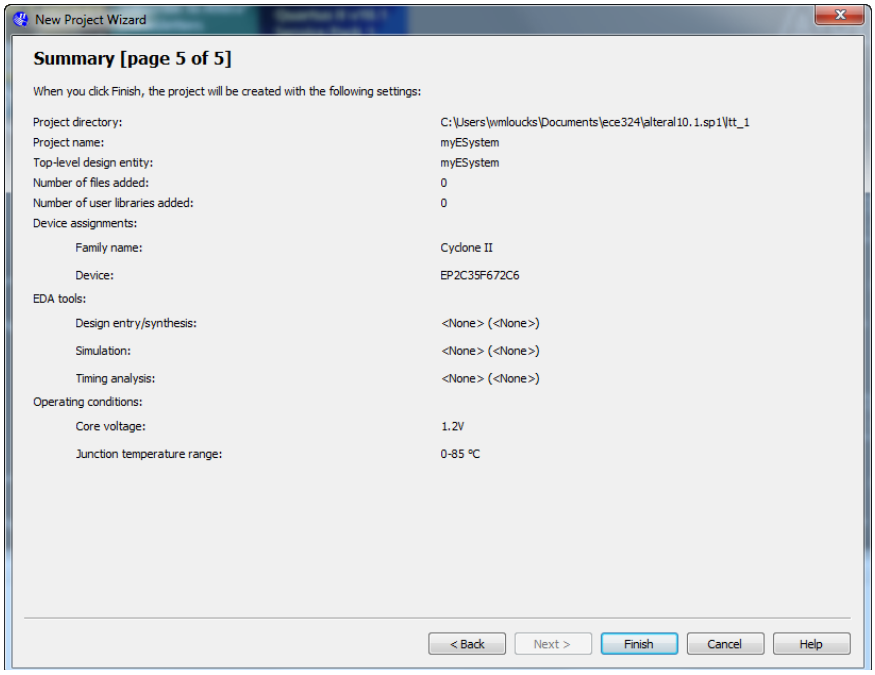

Figure B.5: New Project Wizard Summary

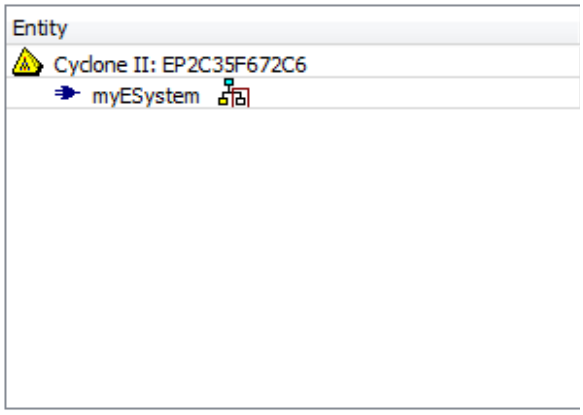

Figure B.6: Project Navigator Window

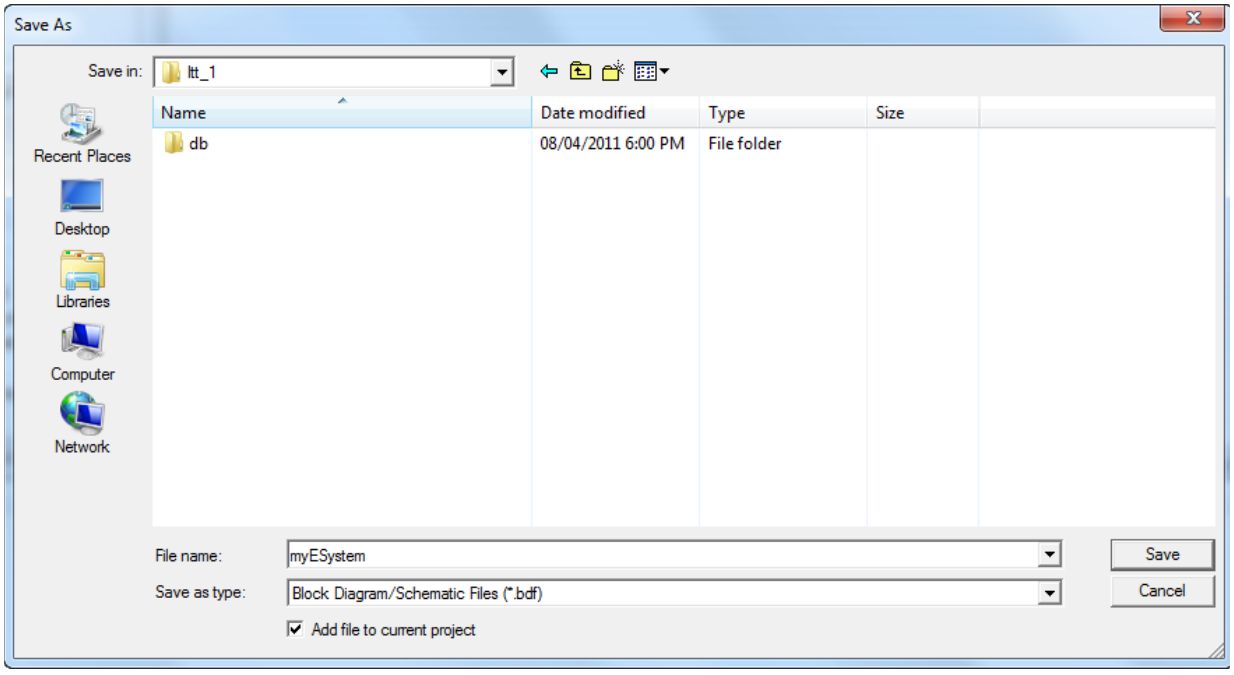

Figure B.7: Saving the File

Figure B.8: Block Editor Toolbar

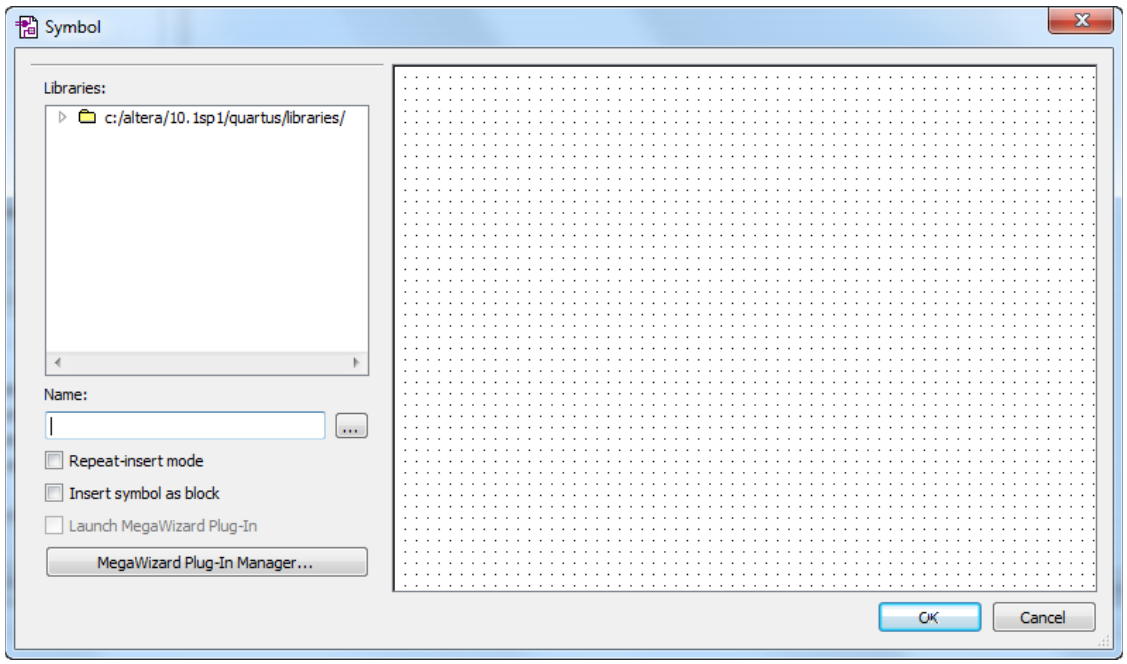

Figure B.9: Symbol Dialog Box

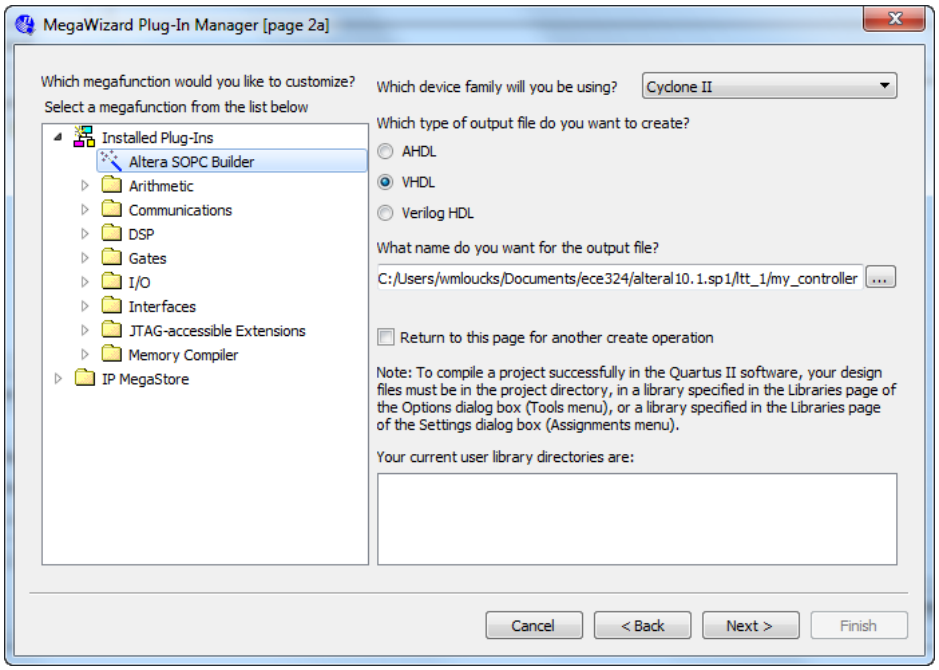

Figure B.10: Megawizard Opening Page to Start the SOPC Builder

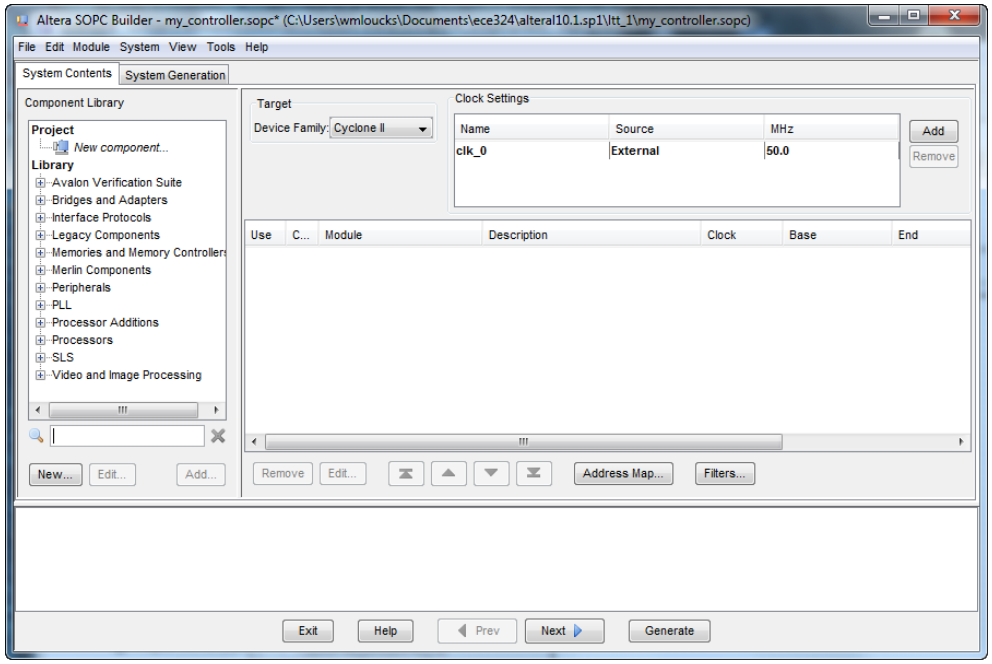

Figure B.11: SOPC Builder Blank Page

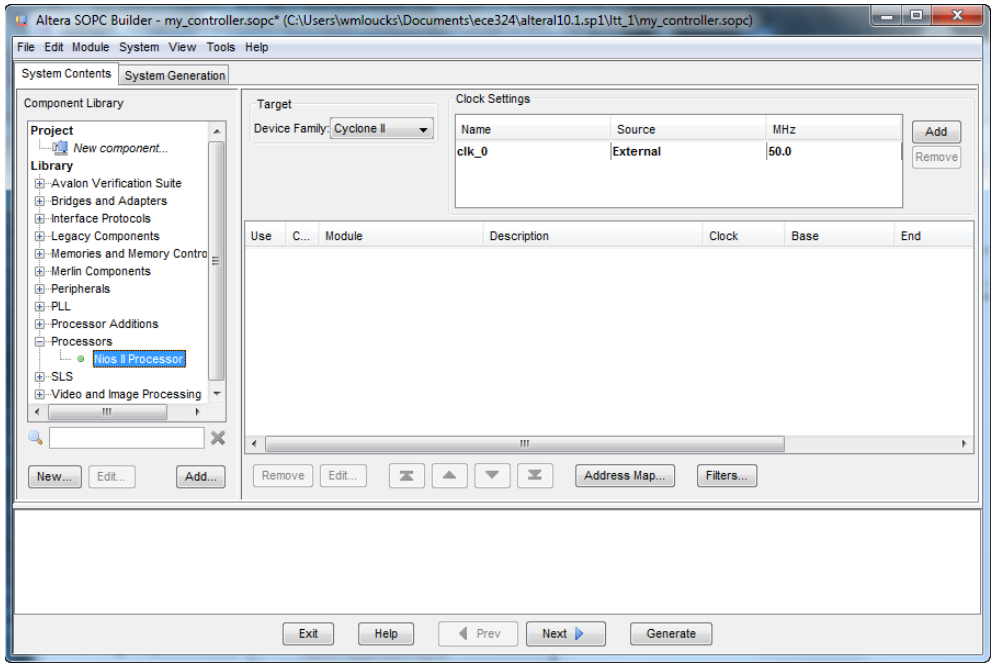

Figure B.12: Nios II CPU Core Tab

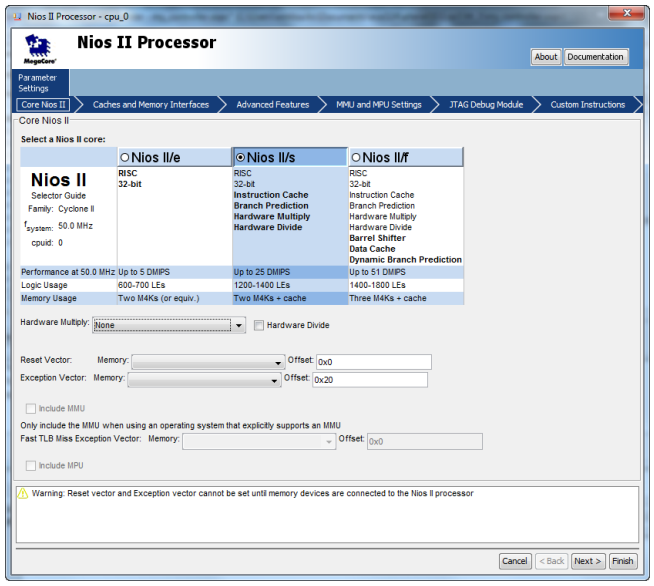

Figure B.13: Nios II Front Page

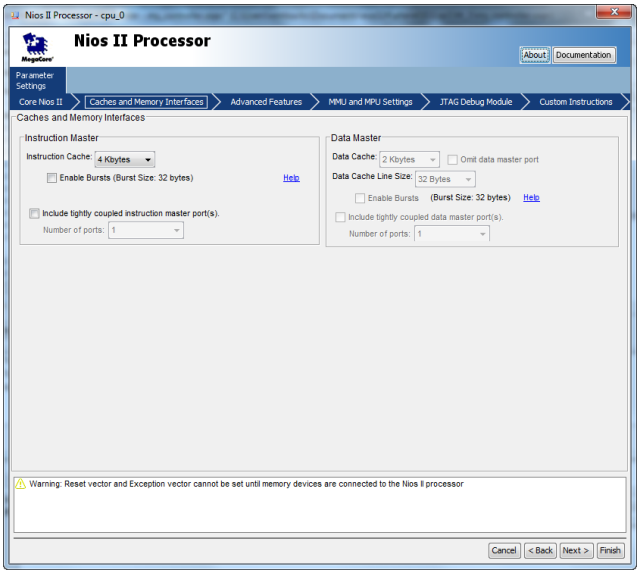

Figure B.14: Nios II CPU Core Tab

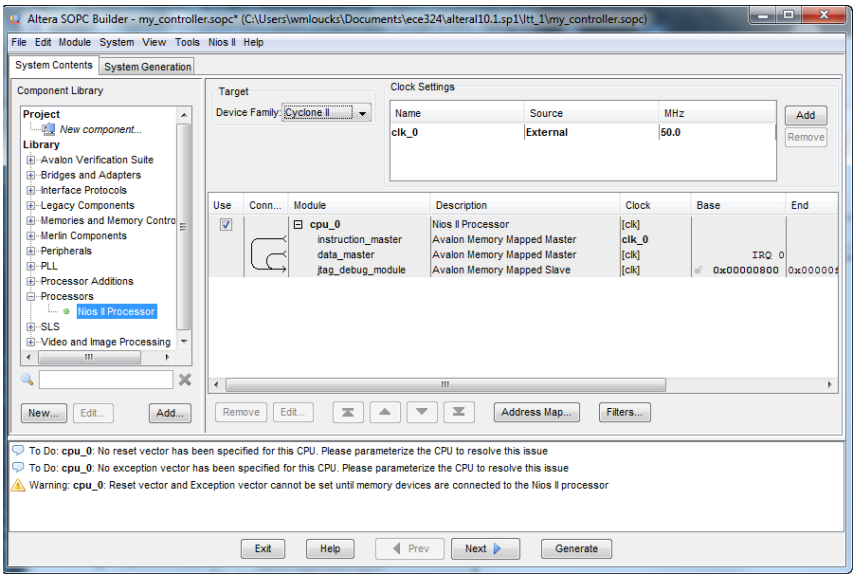

Figure B.15: SOPC Builder with CPU

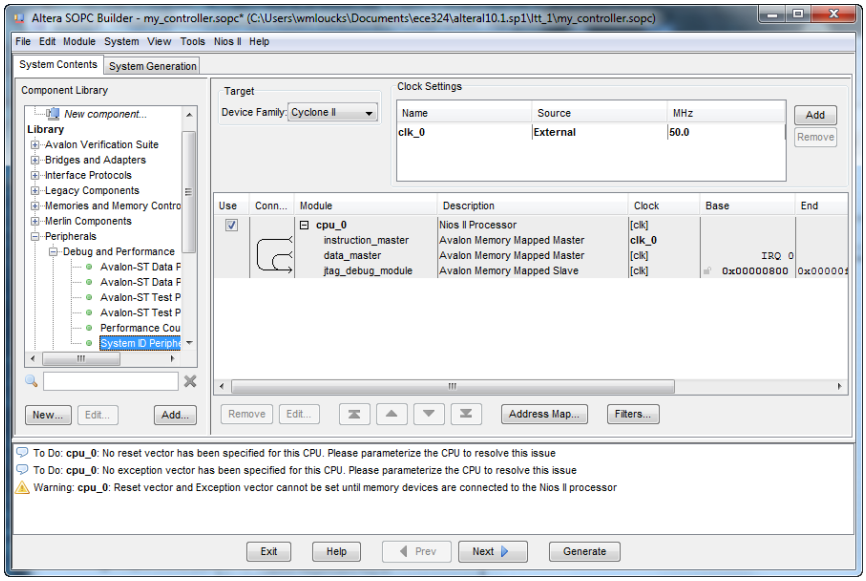

Figure B.16: SOPC Builder with CPU and select sysid

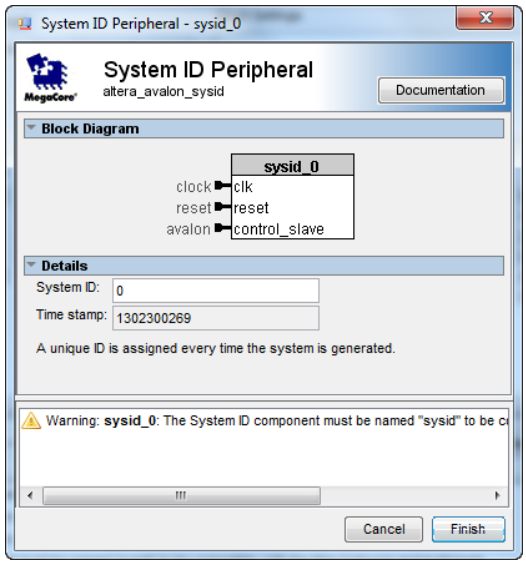

Figure B.17: SOPC Builder with CPU and select sysid wizard

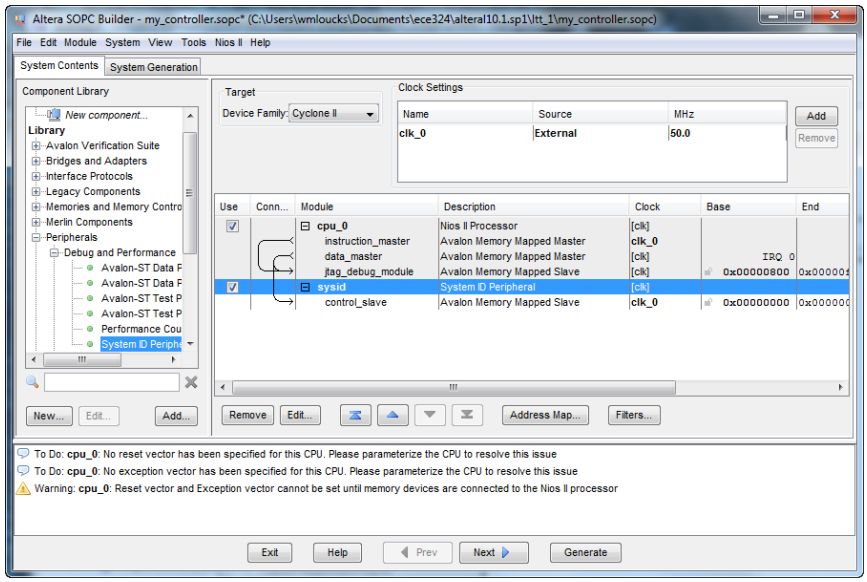

Figure B.18: SOPC Builder with CPU and select sysid after wizard and rename

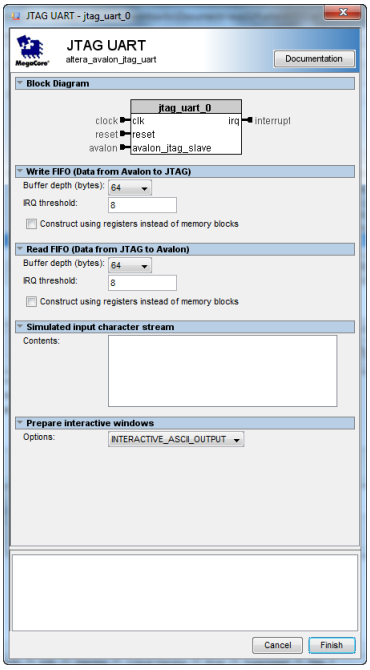

Figure B.19: Communications UART Wizard Settings

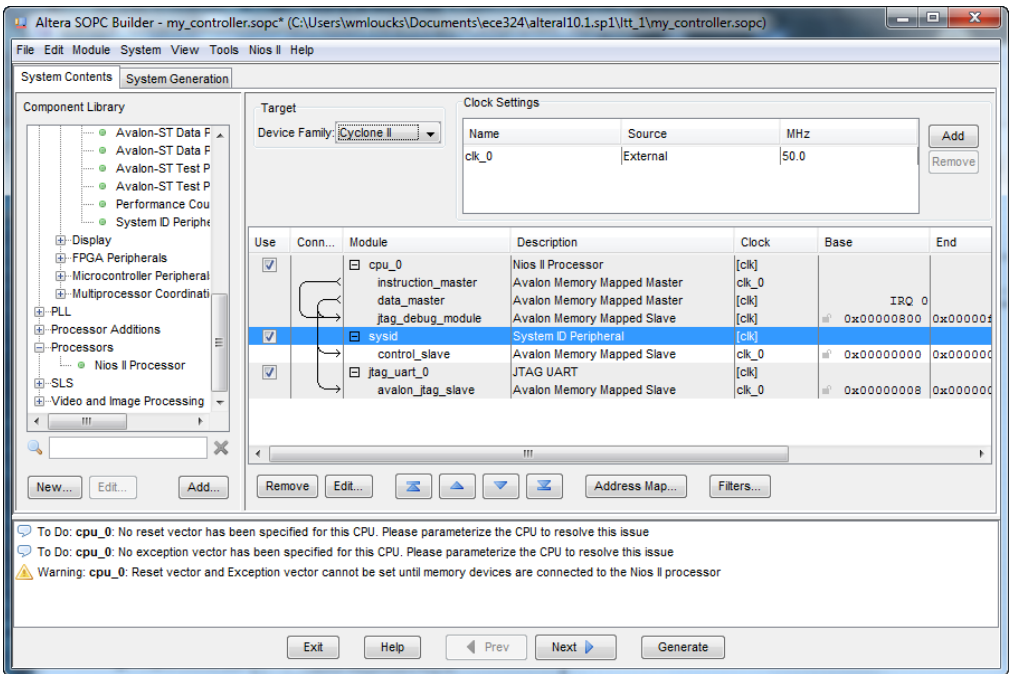

Figure B.20: SOPC After Jtag Uart Added

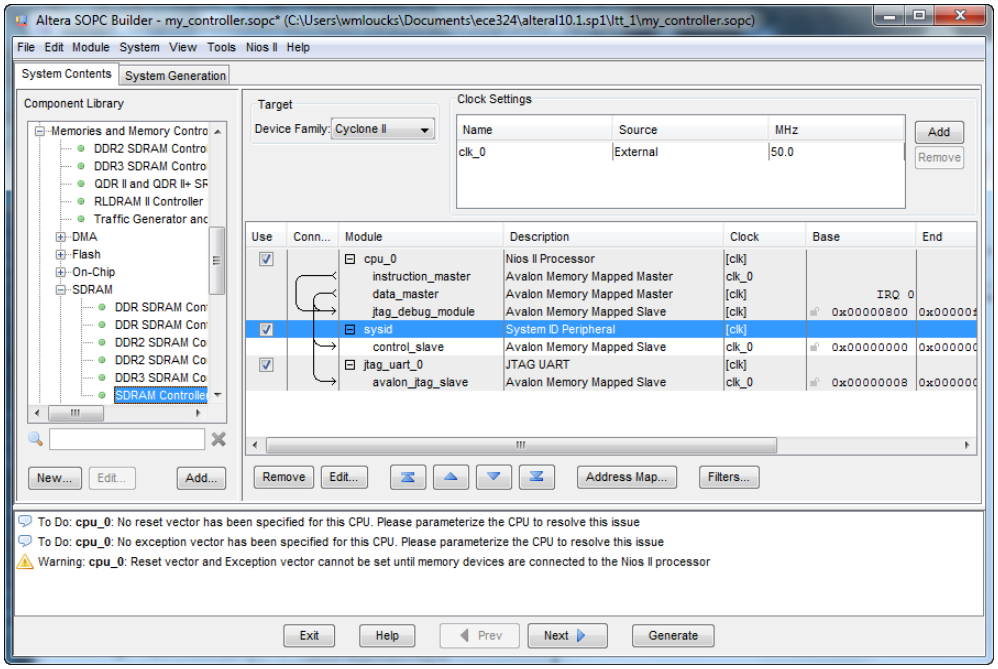

Figure B.21: SOPC Builder Before SDRAM

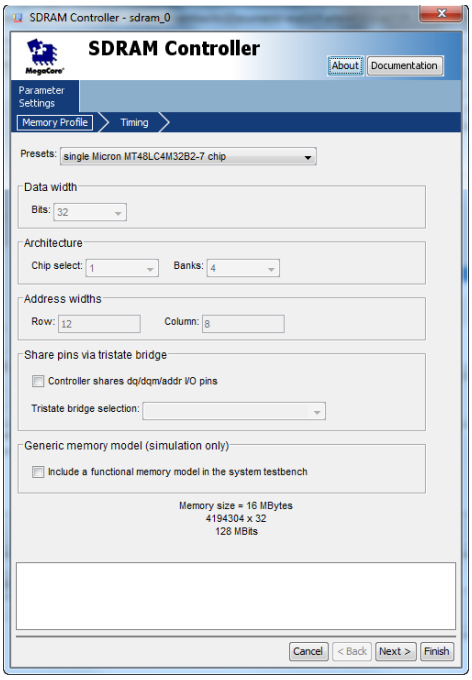

Figure B.22: External SDRAM Wizard - Before

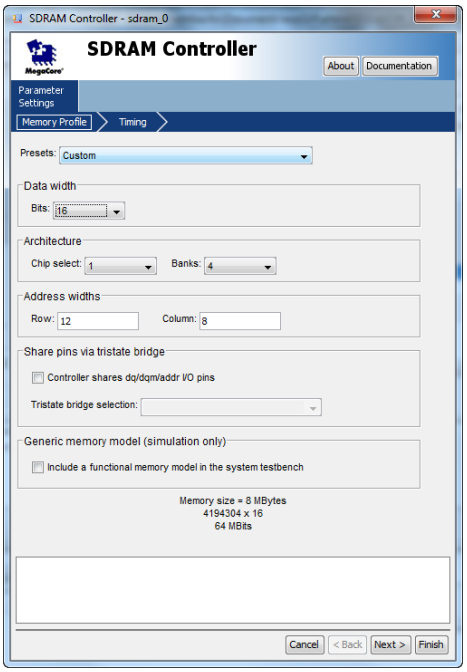

Figure B.23: External SDRAM Wizard - As Modified

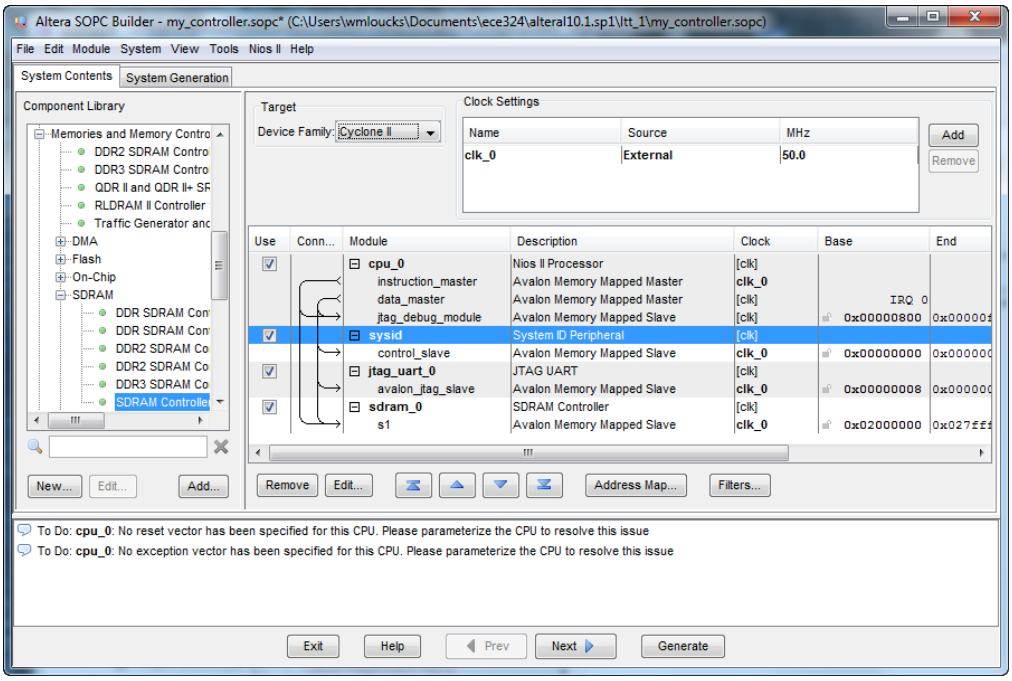

Figure B.24: External SDRAM Wizard after also before re-entry to NIOS

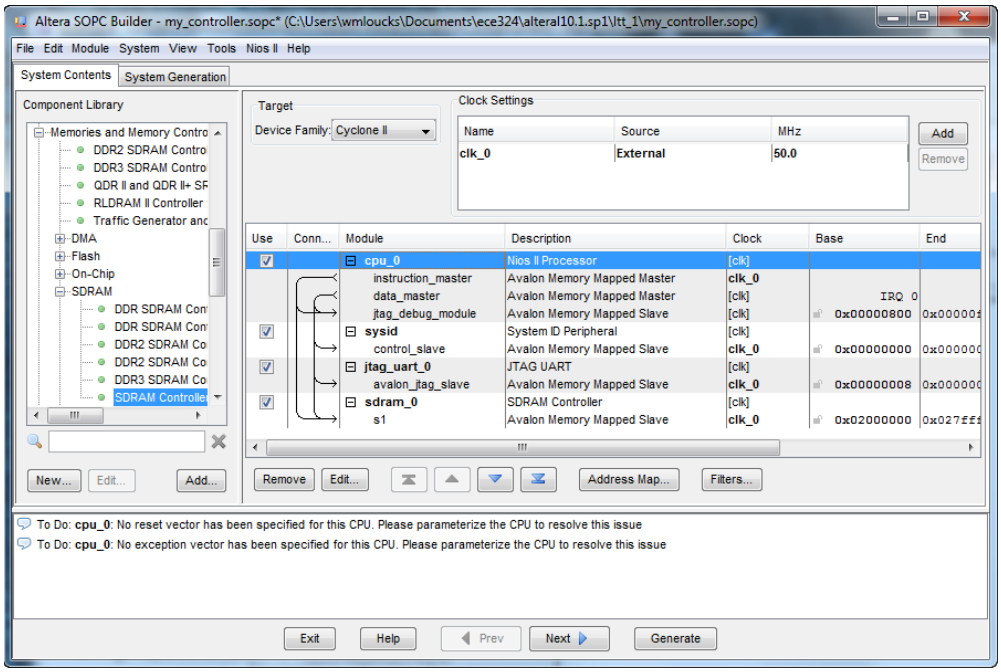

Figure B.25: External SDRAM Wizard after also before re-entry to NIOS

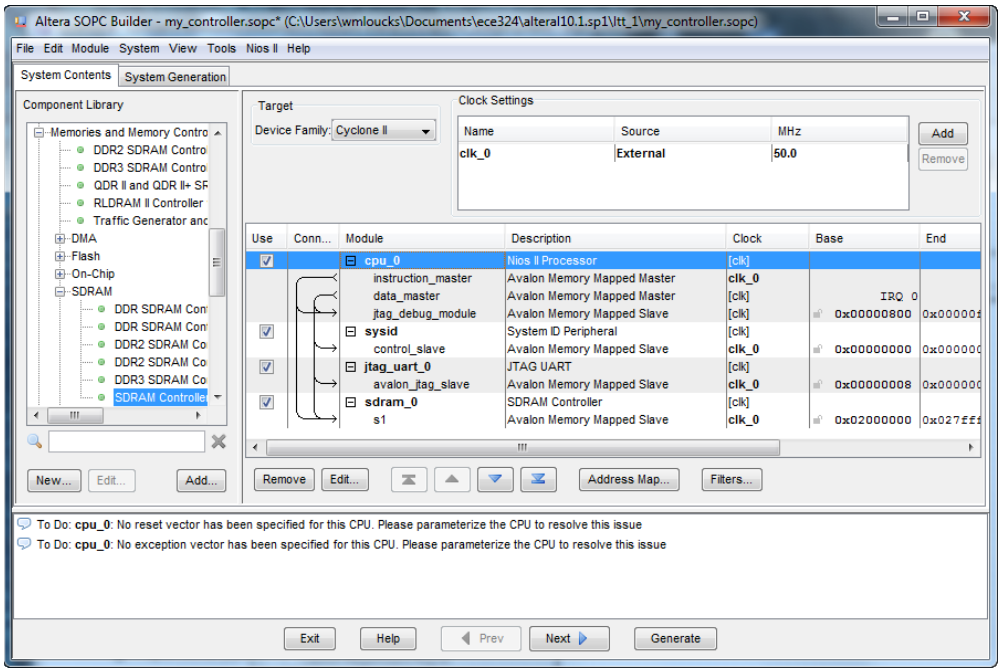

Figure B.26: Double Click on cpu_0 to Re-enter Nios II Wizard

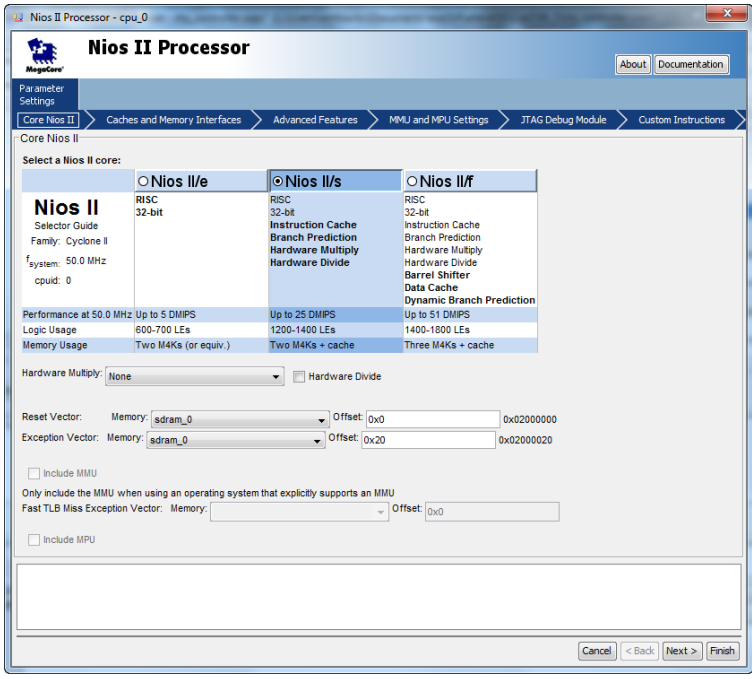

Figure B.27: Modify the Reset and Exception Vectors

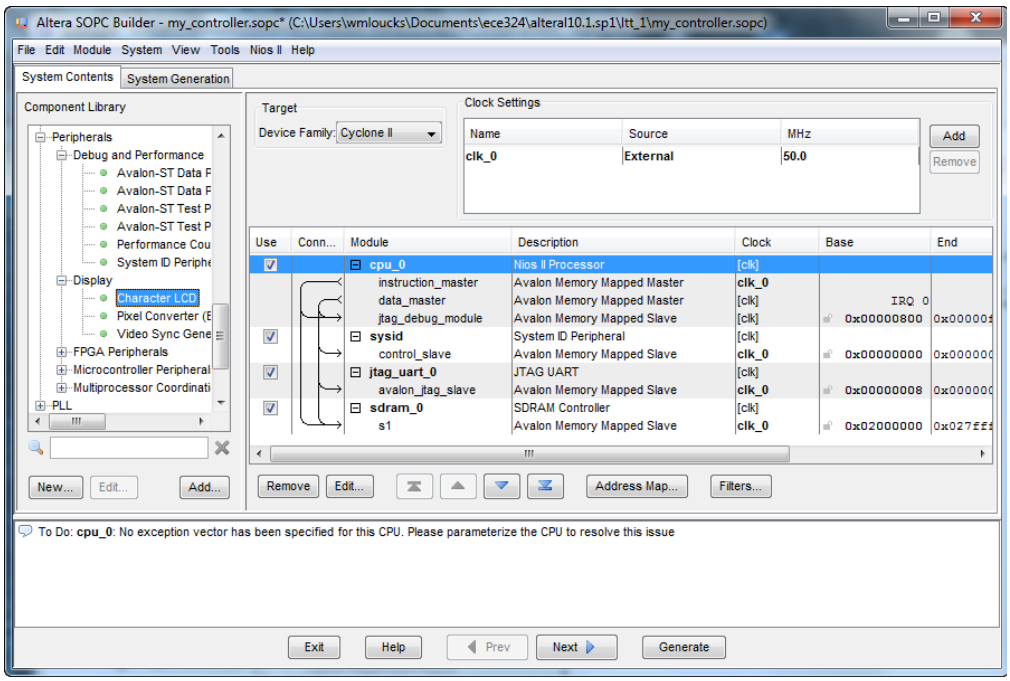

Figure B.28: Select LCD Component

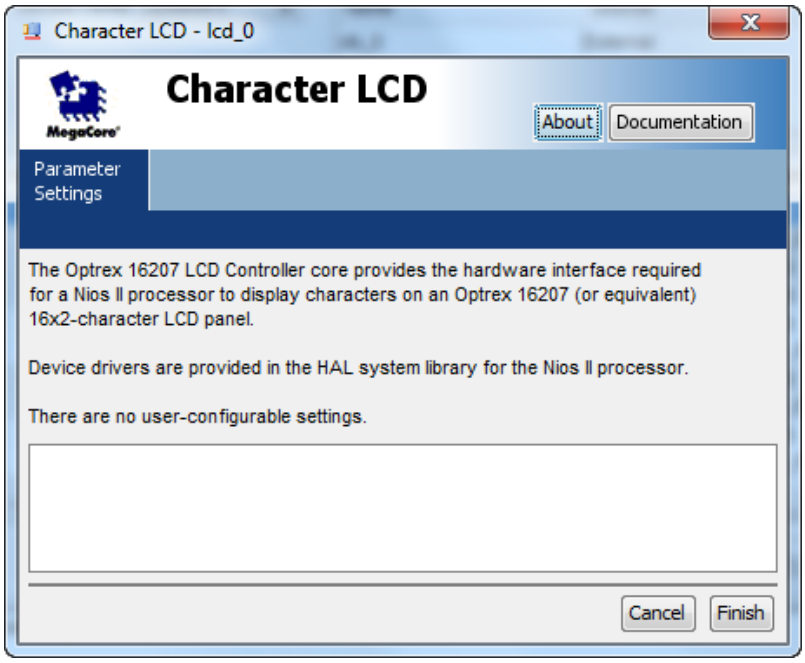

Figure B.29: LCD Wizard Page

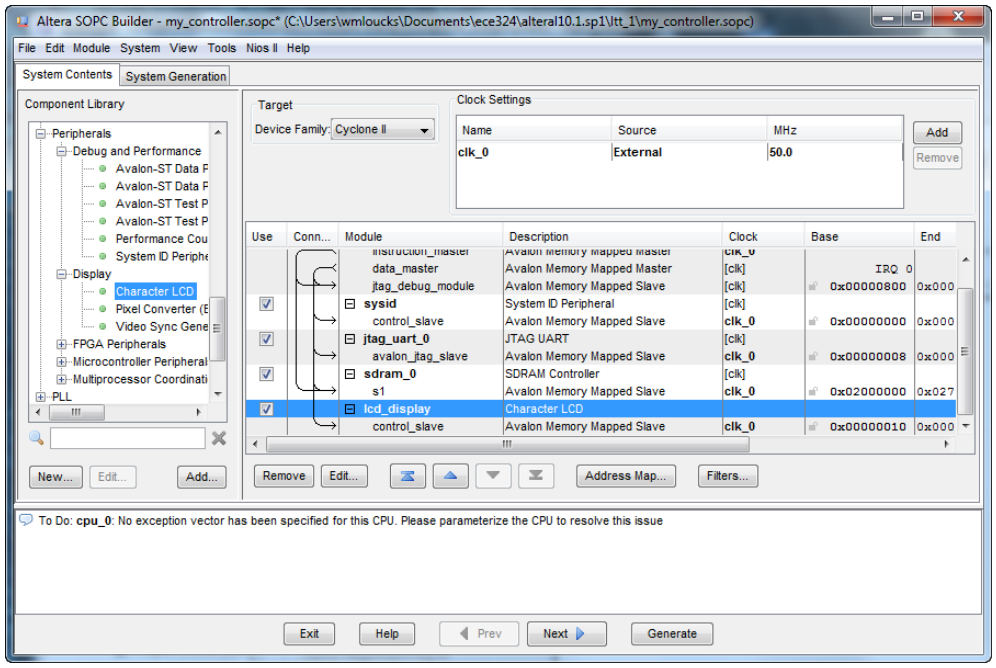

Figure B.30: SOPC After LCD Added

PIOS

First the output PIOS

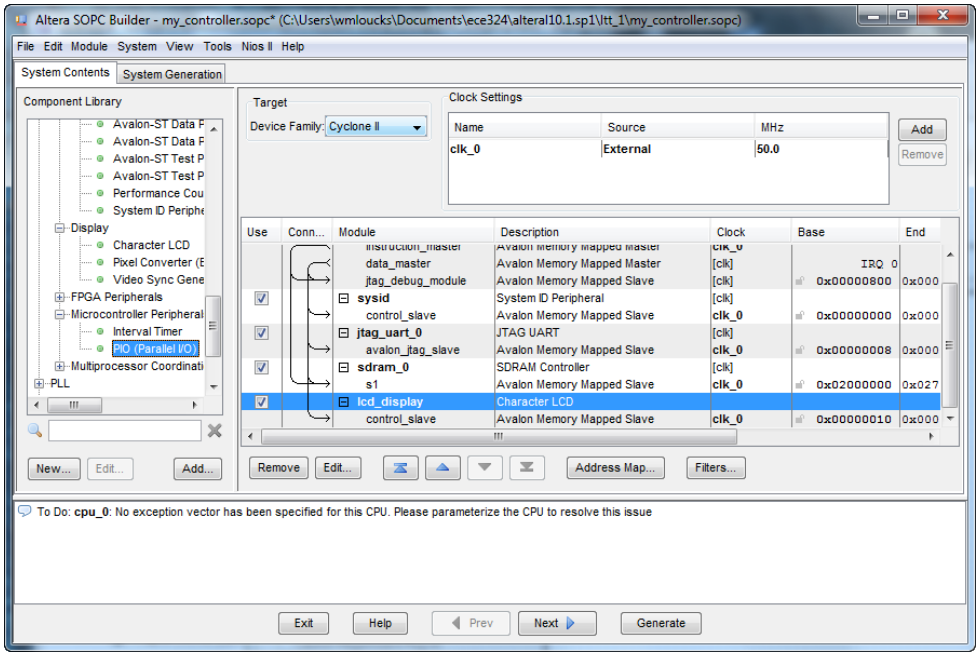

Figure B.31: Select PIO

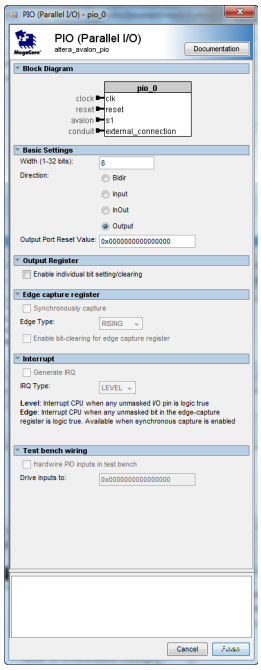

Figure B.32: PIO Wizard Showing 8 bits out

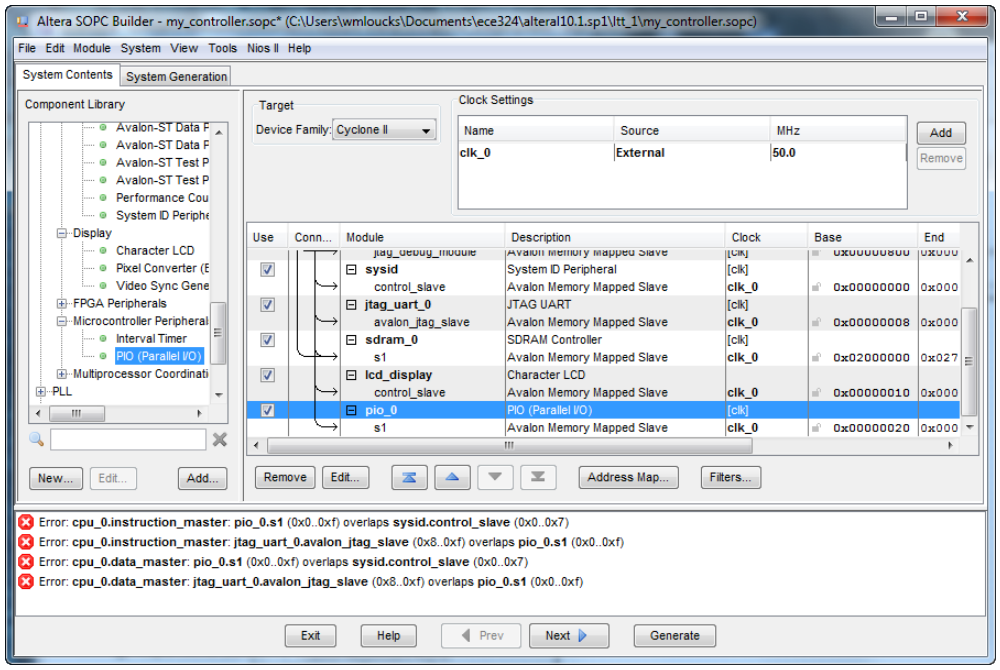

Figure B.33: pio Information after return from wizard before name change

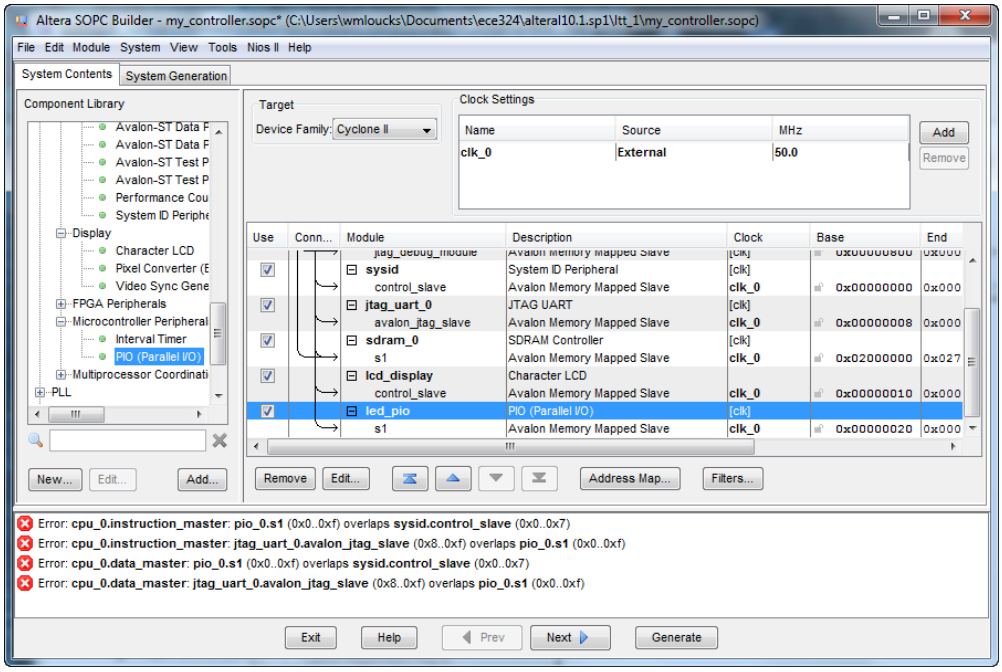

Figure B.34: SOPC led_pio After Name Change

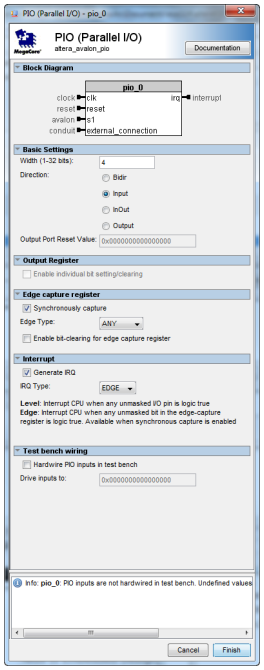

Figure B.35: SOPC Builder Select Button Pio Input Controls button_pio Wizard

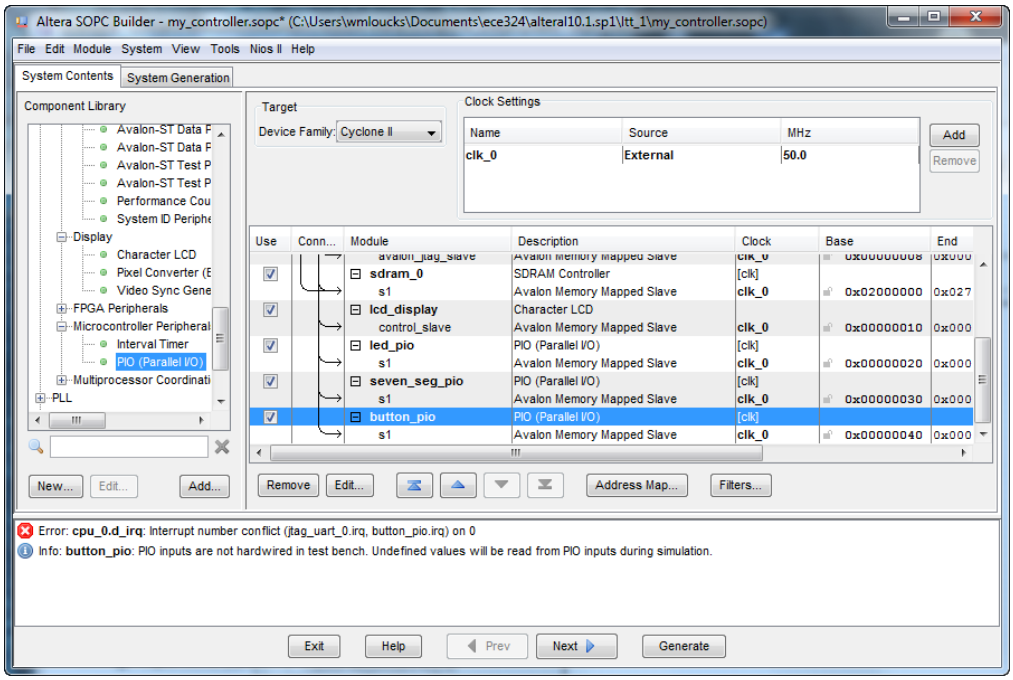

Figure B.36: SOPC Builder After adding seven_seg_pio and button_pio

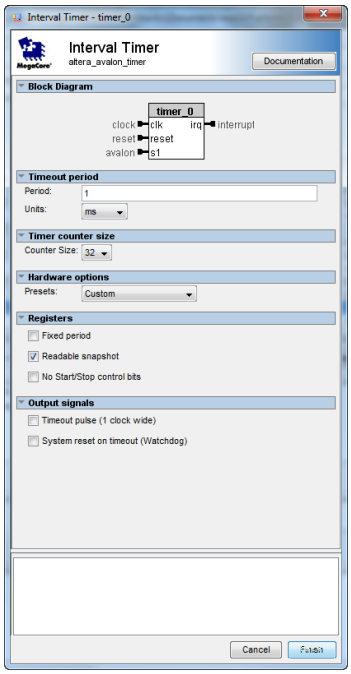

Figure B.37: Timer Wizard Settings timer_0 (default)

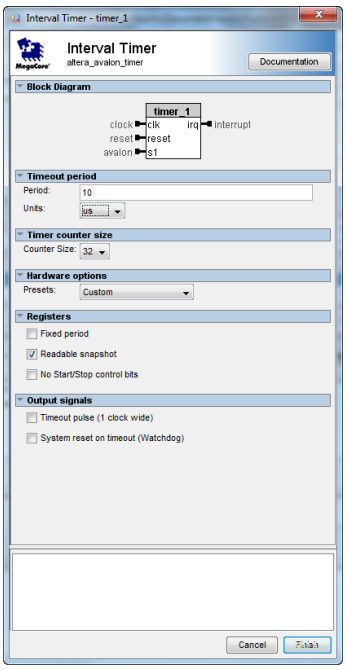

Figure B.38: Timer Wizard Settings timer_1

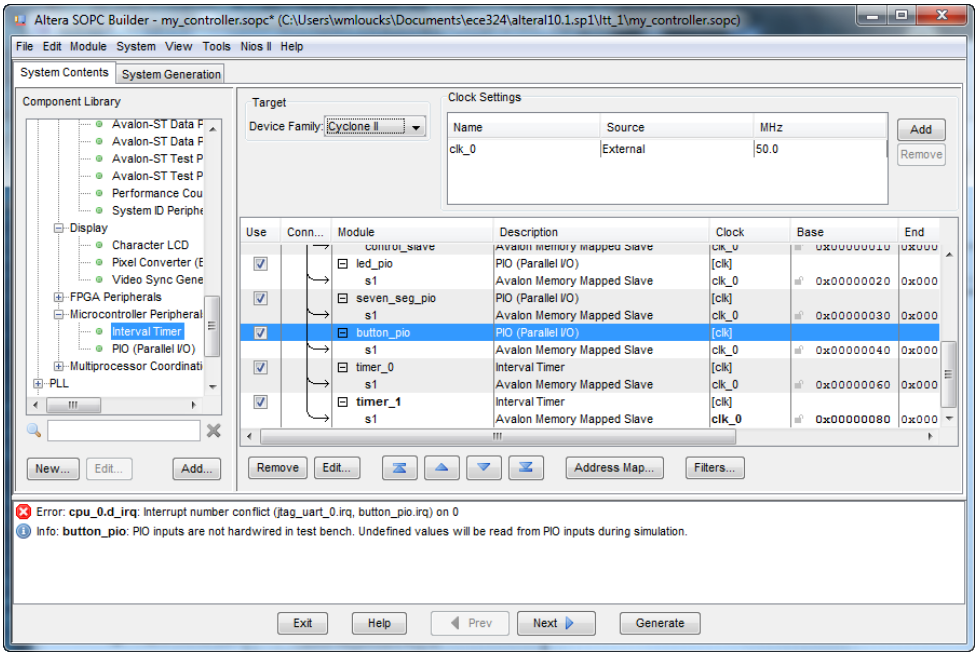

Figure B.39: SOPC Builder After Adding timer_1

Other PIOs

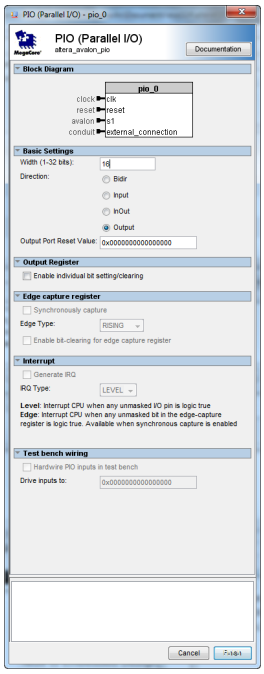

Figure B.40: PIO Wizard - 16 bit output

next one needs to be replaced with a 16 bit version!

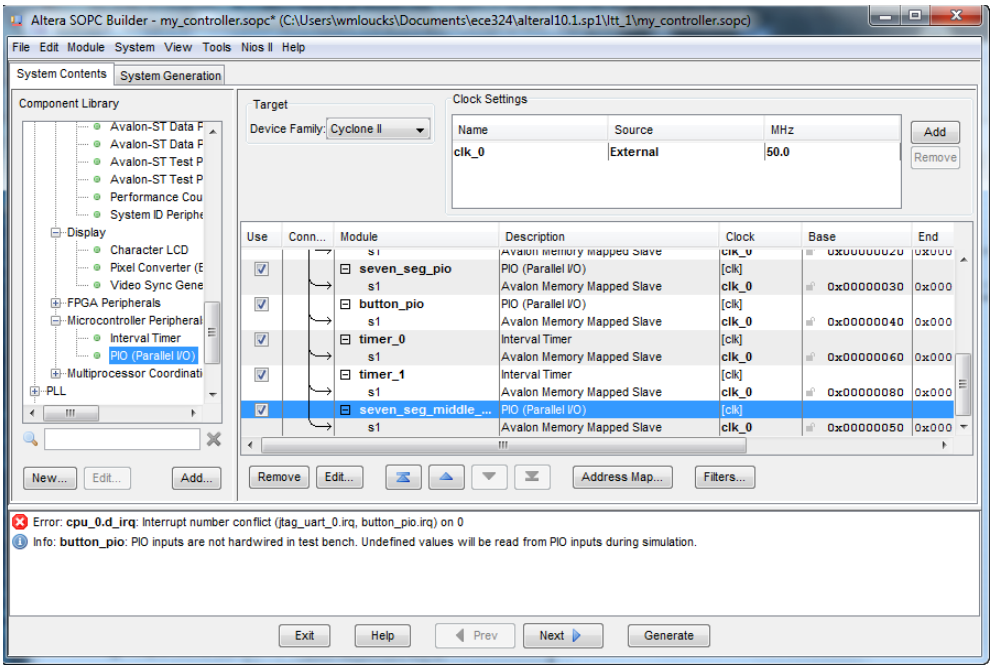

Figure B.41: SOPC Builder after adding middle seven segment pio

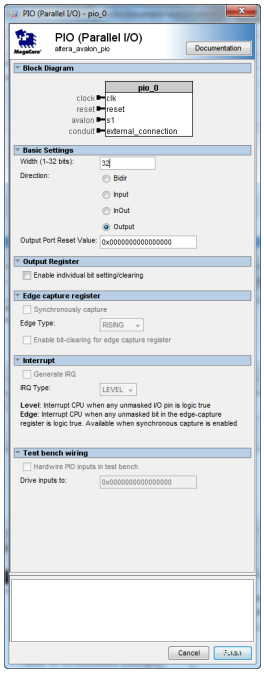

Figure B.42: PIO Wizard - 32 bit output

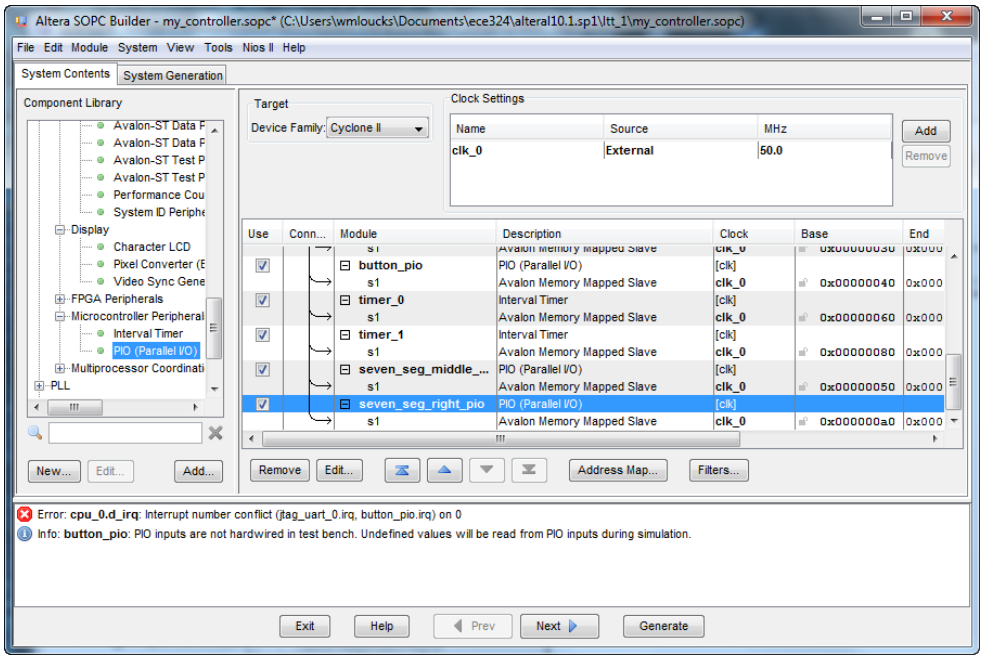

Figure B.43: SOPC Builder after adding seven Segment PIOs

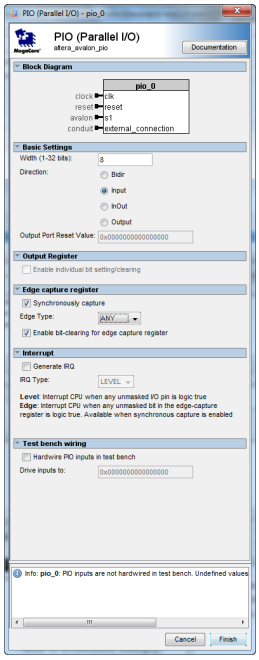

Figure B.44: PIO Wizard - 16 bit input

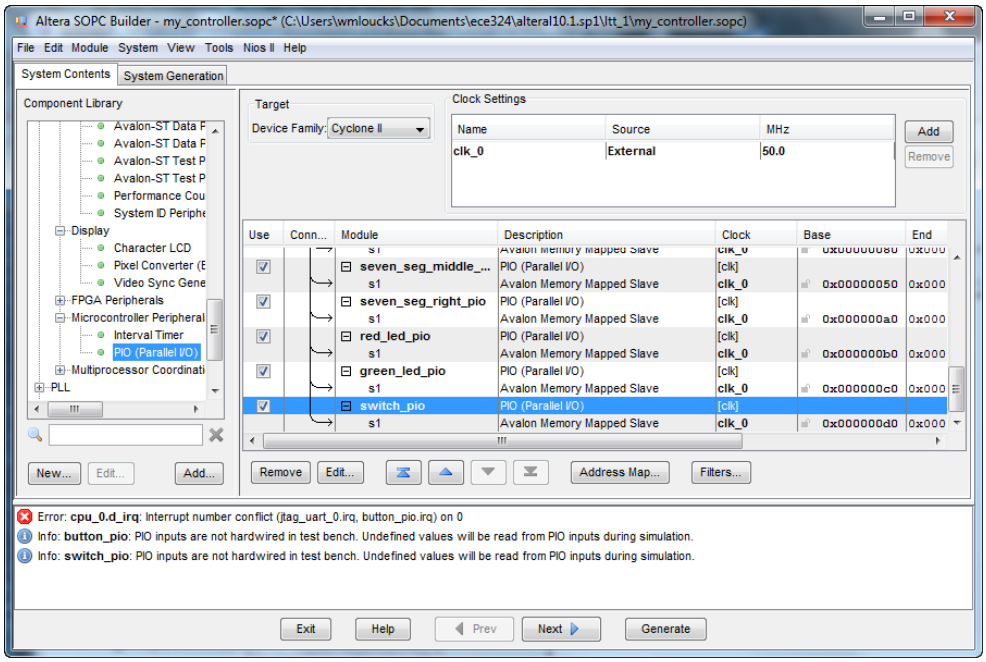

Figure B.45: SOPC Builder after adding all other PIOs

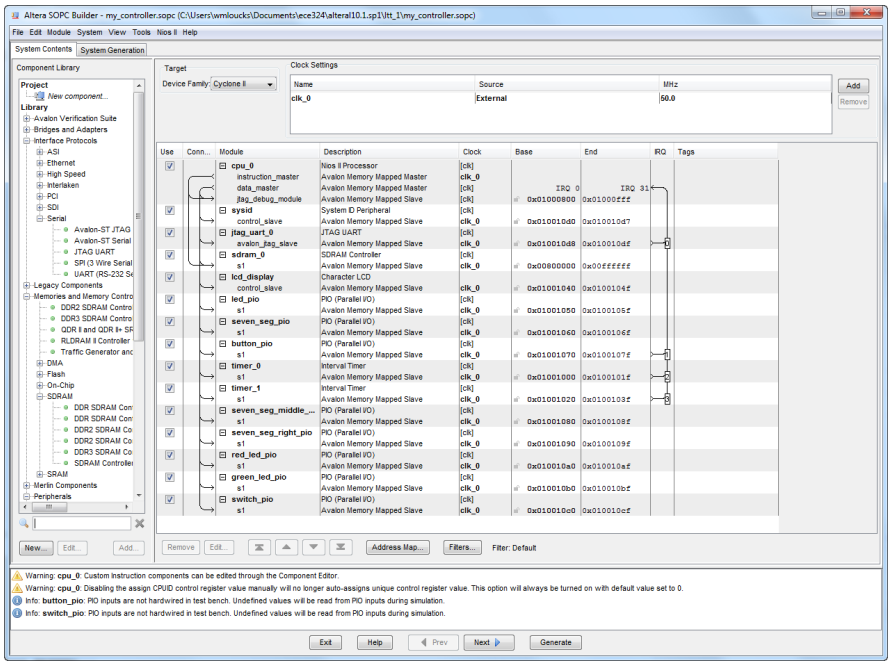

Figure B.46: Base Address & IRQ Values Added

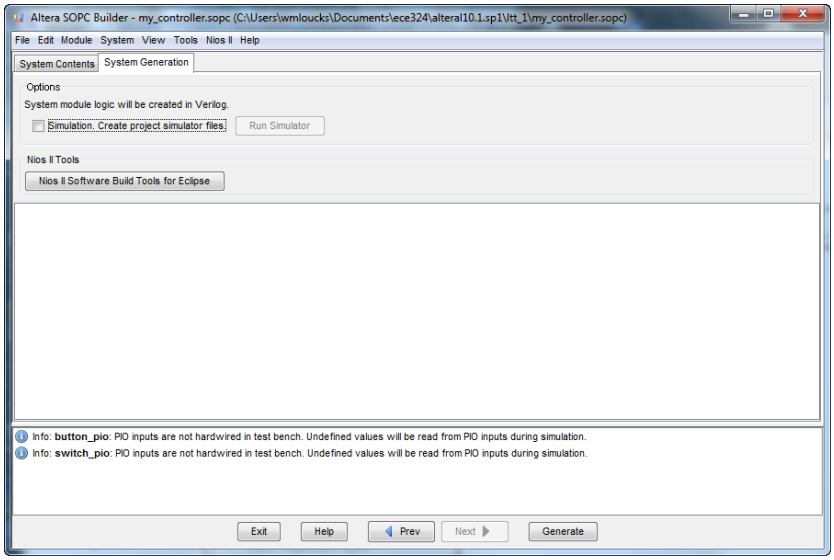

Figure B.47: SOPC Builder System Generation Tab - Before

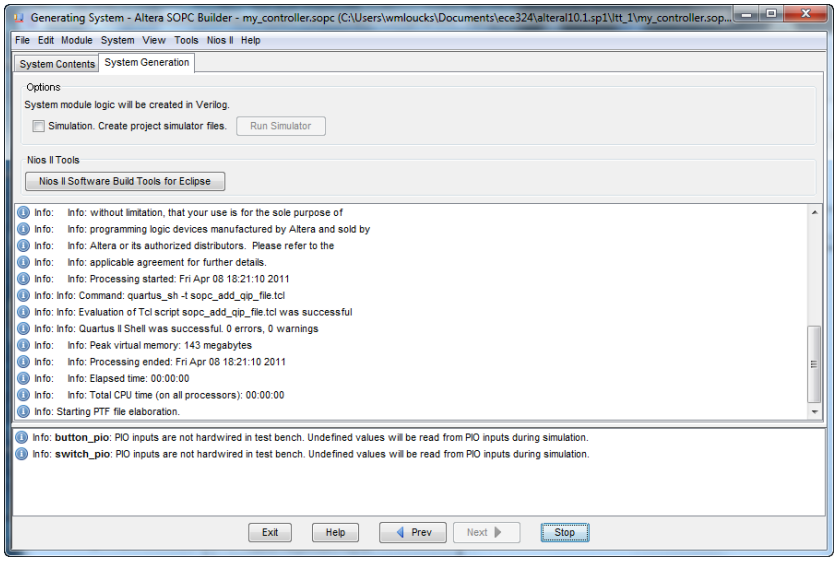

Figure B.48: SOPC Builder System Generation Tab while running

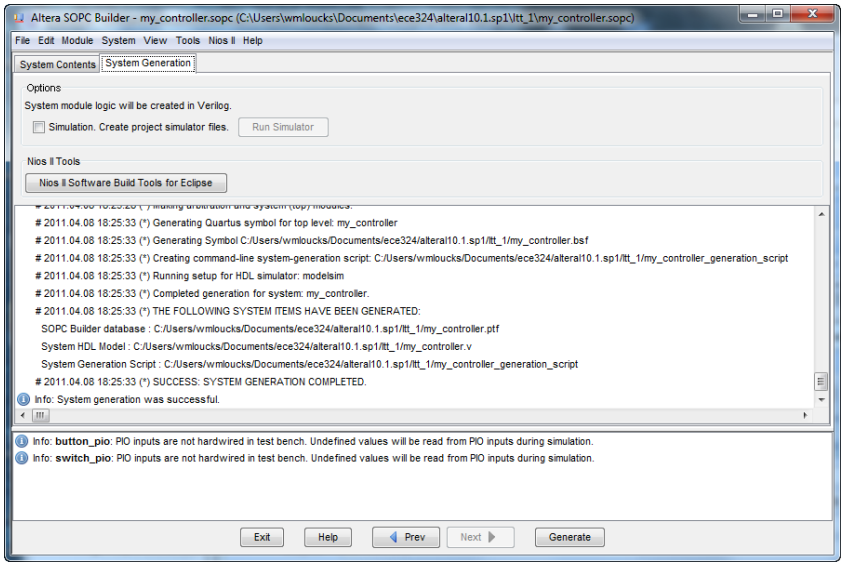

Figure B.49: SOPC Builder System Generation Tab success

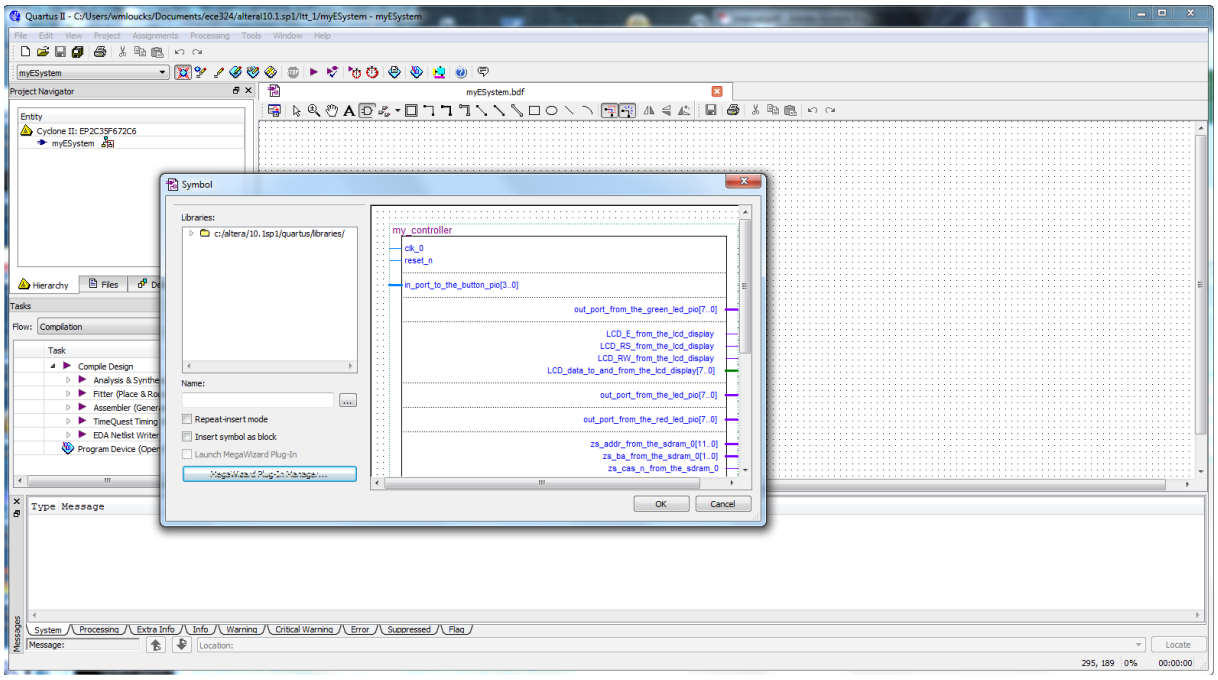

Figure B.50: Adding the my_controller Symbol in Symbol Tab

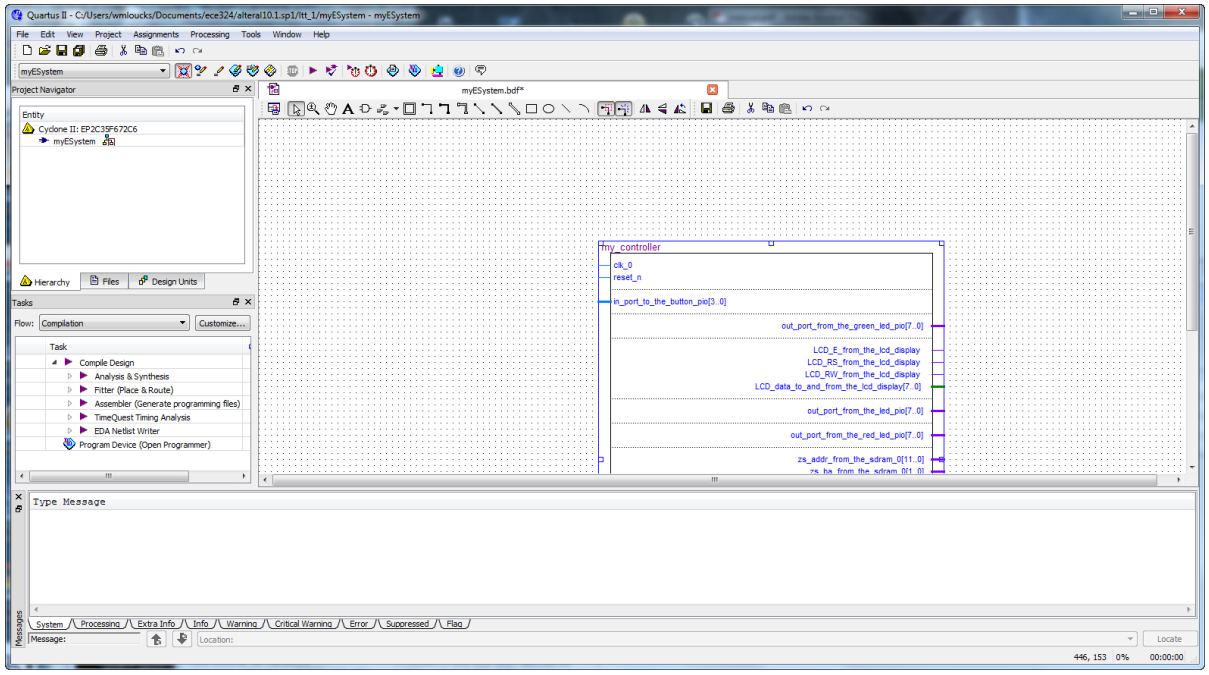

Figure B.51: Adding the my_controller Symbol to the BDF

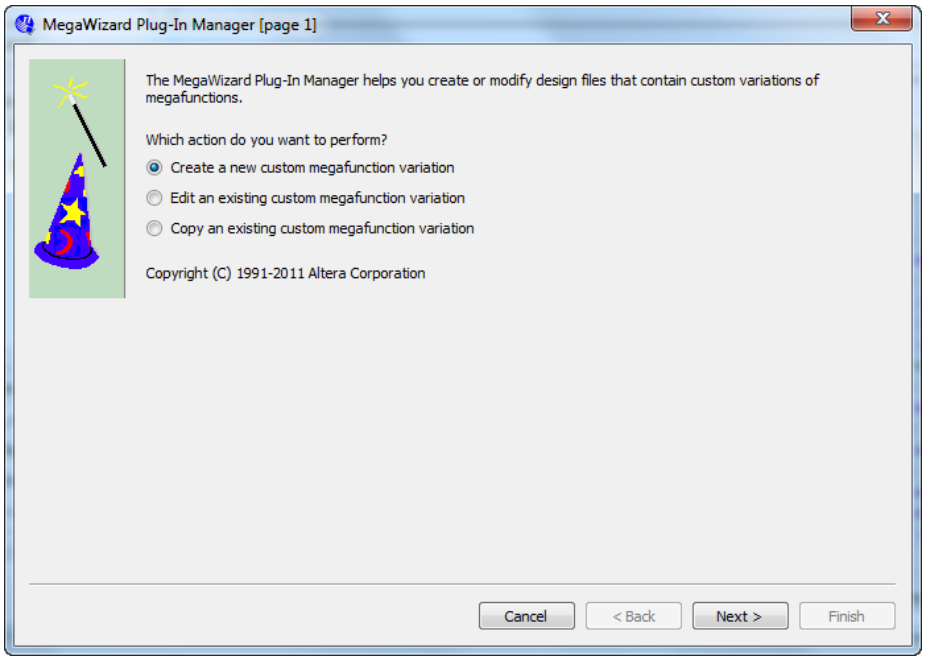

Figure B.52: PLL Wizard - Start

Adding Input Pin figures

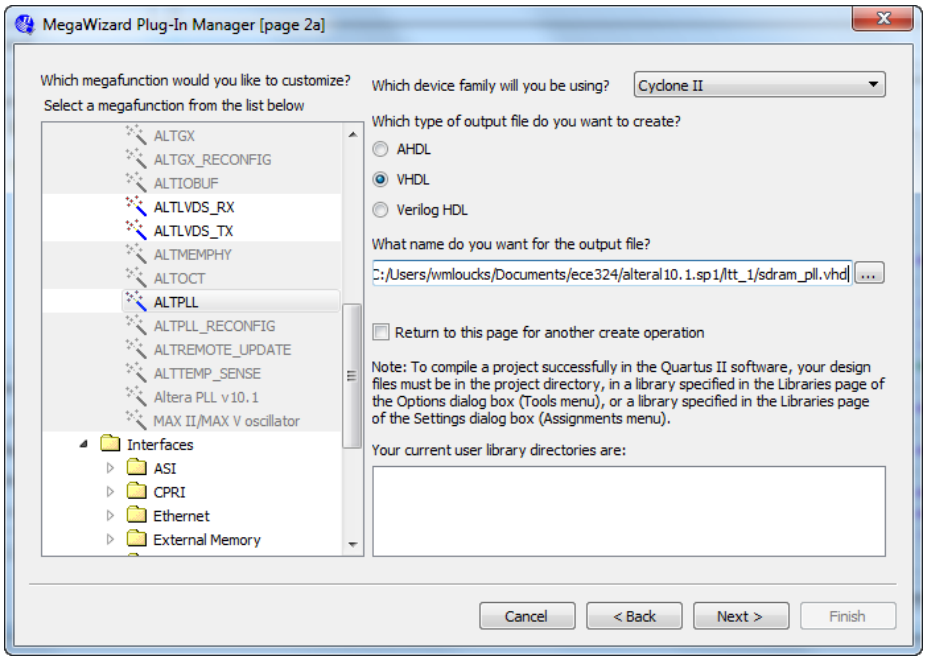

Figure B.53: PLL Wizard - select ALTPLL and filename page 2a

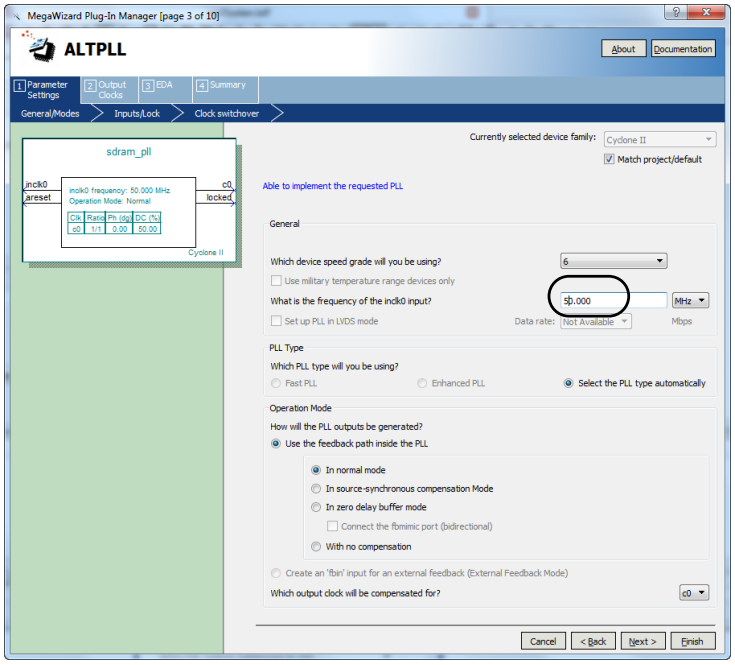

Figure B.54: PLL Wizard - final page 3

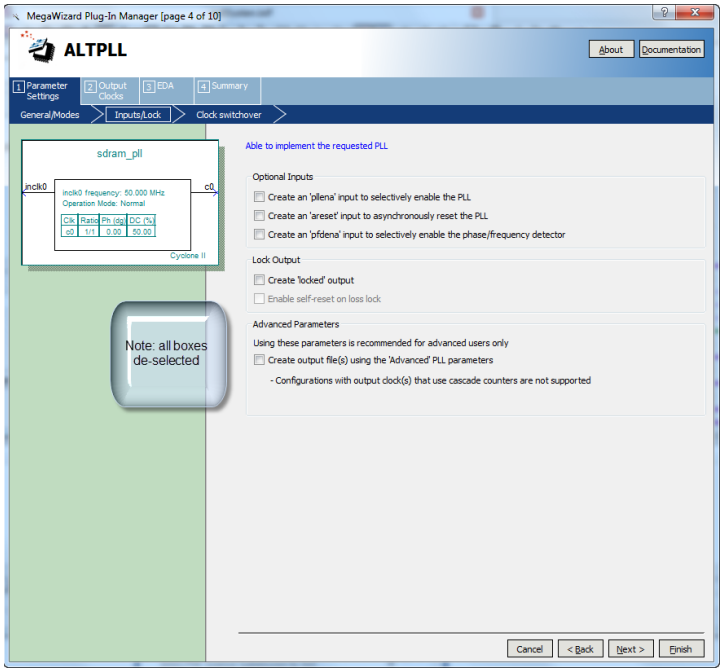

Figure B.55: PLL Wizard - final page 4

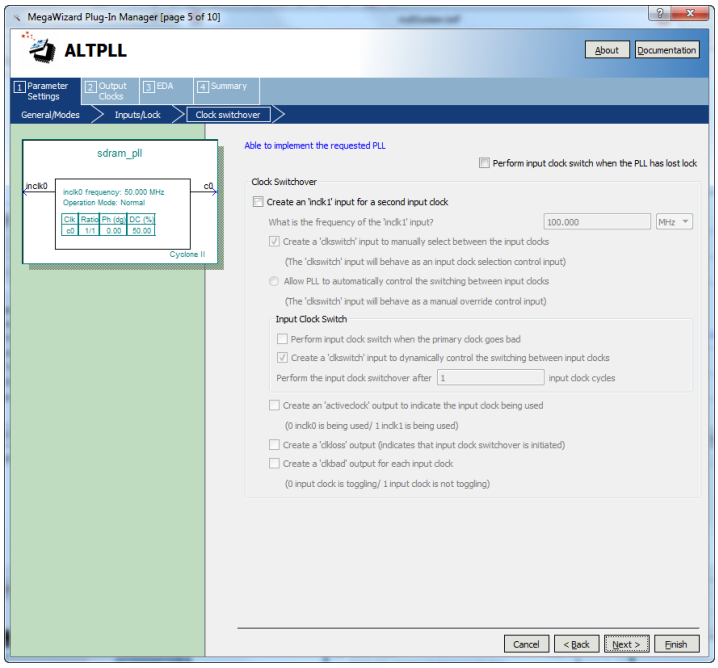

Figure B.56: PLL Wizard - final page 5 (no change)

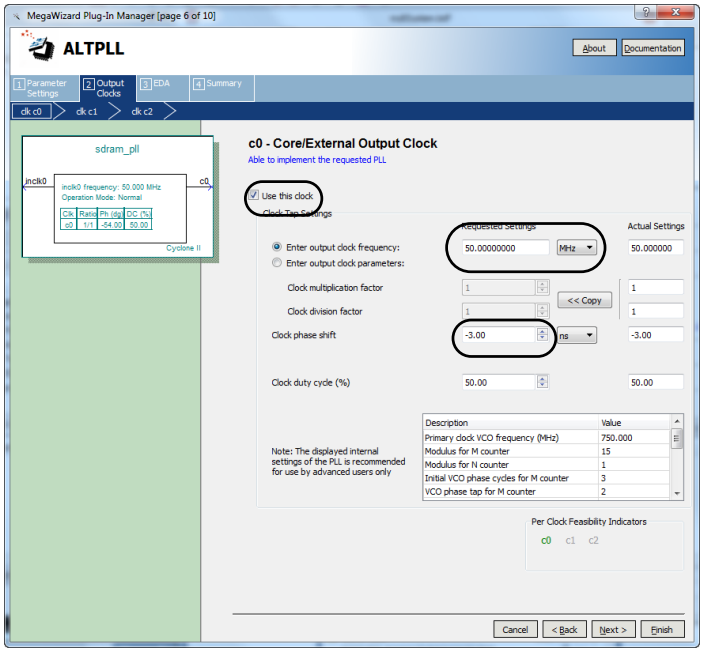

Figure B.57: PLL Wizard - final page 6

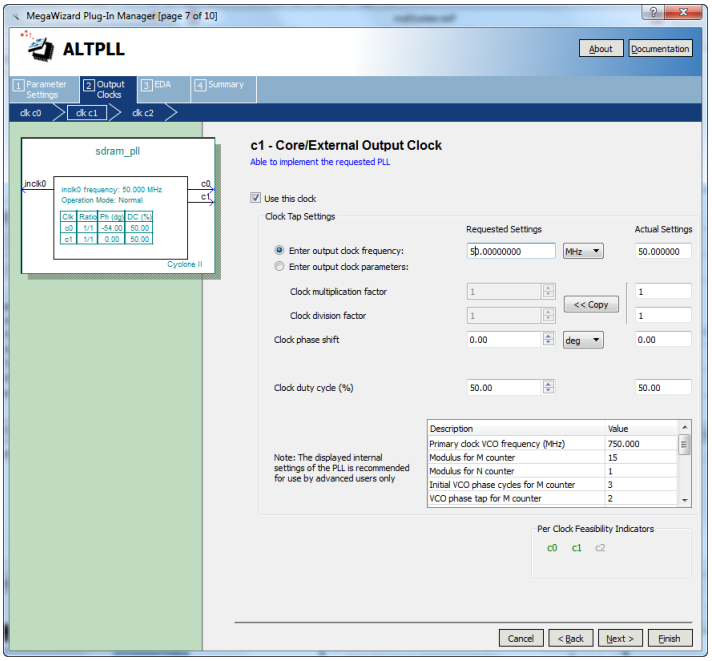

Figure B.58: PLL Wizard - final page 7 - clock 2 no delay

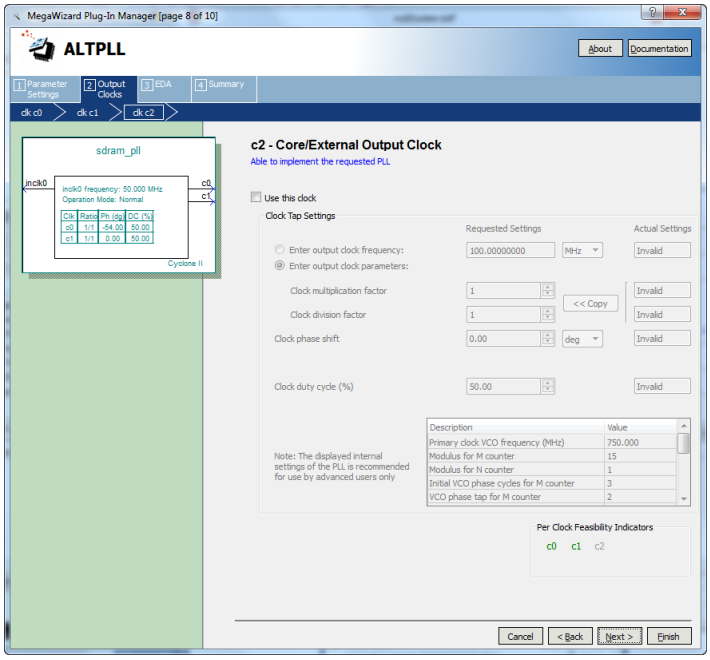

Figure B.59: PLL Wizard - final page 8 - no need for c2

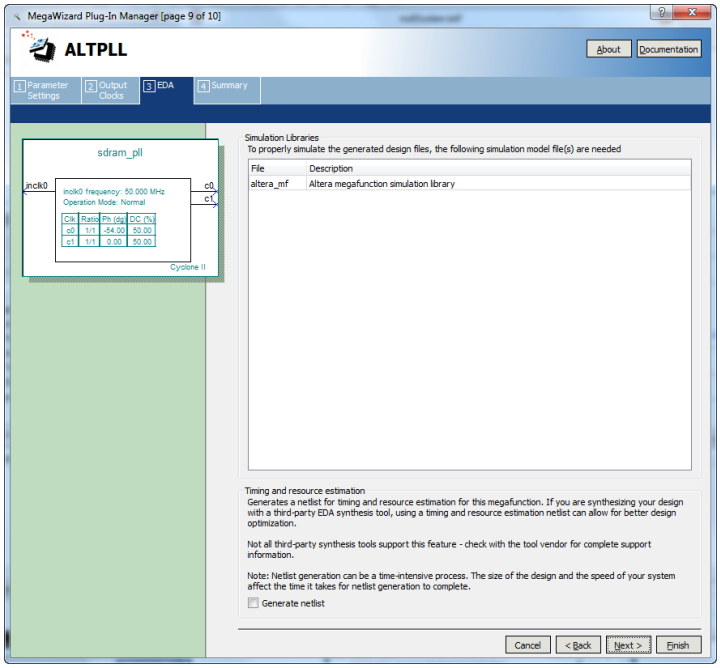

Figure B.60: PLL Wizard - final page 9

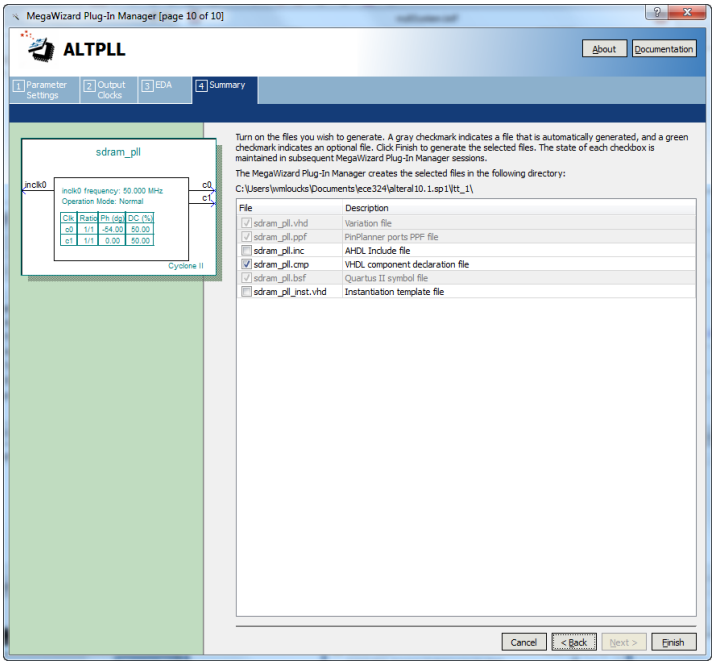

Figure B.61: PLL Wizard - final page 10

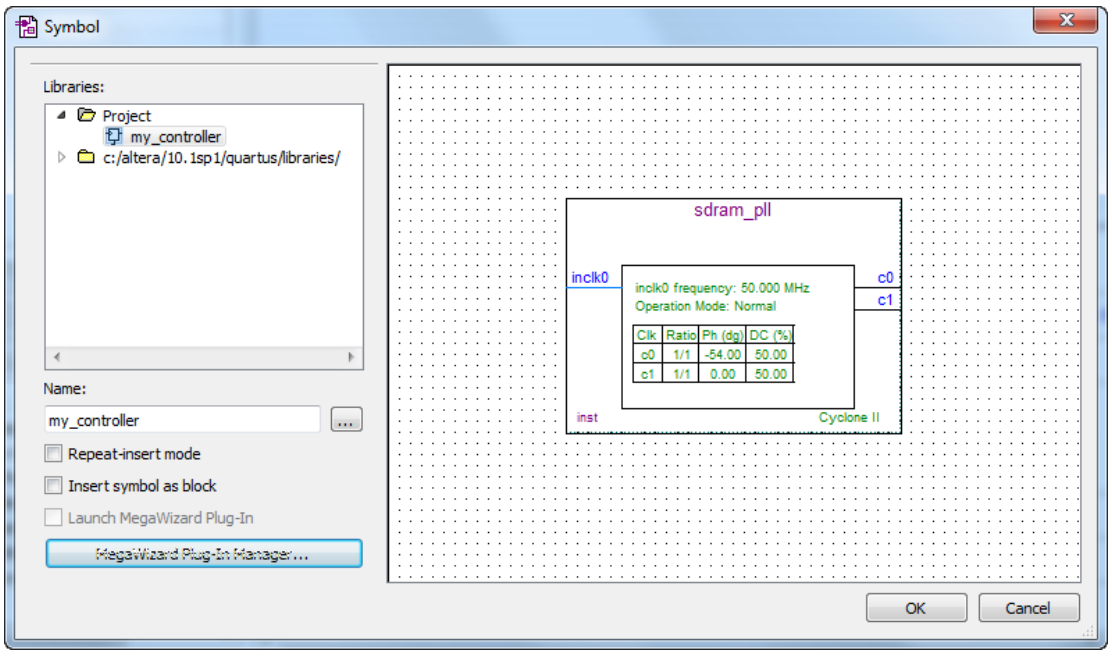

Figure B.62: PLL Wizard - symbol insert

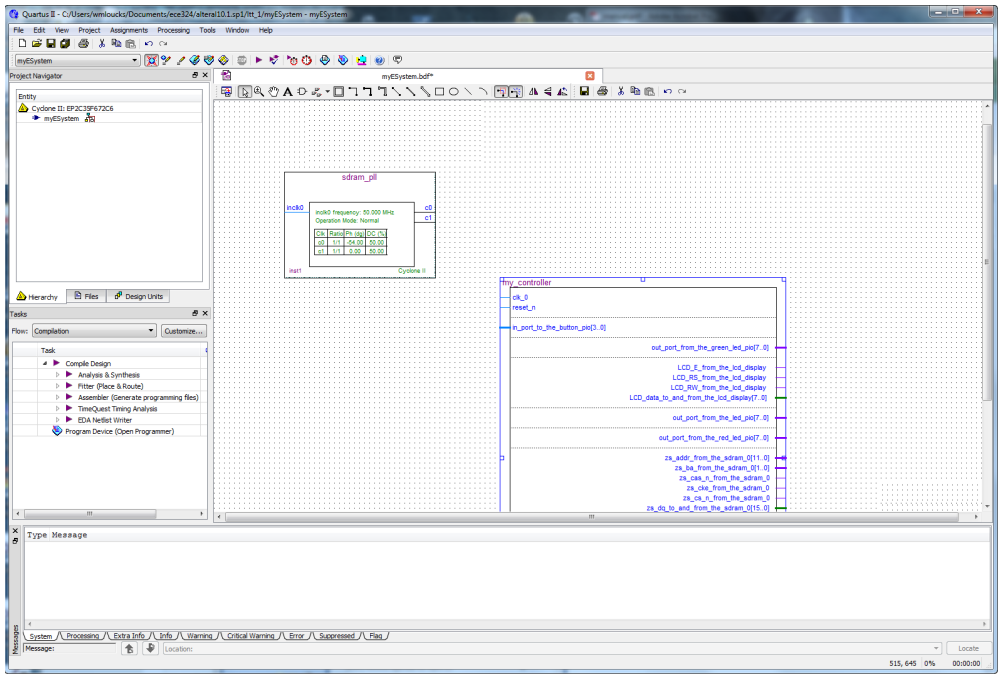

Figure B.63: BDF after PLL added

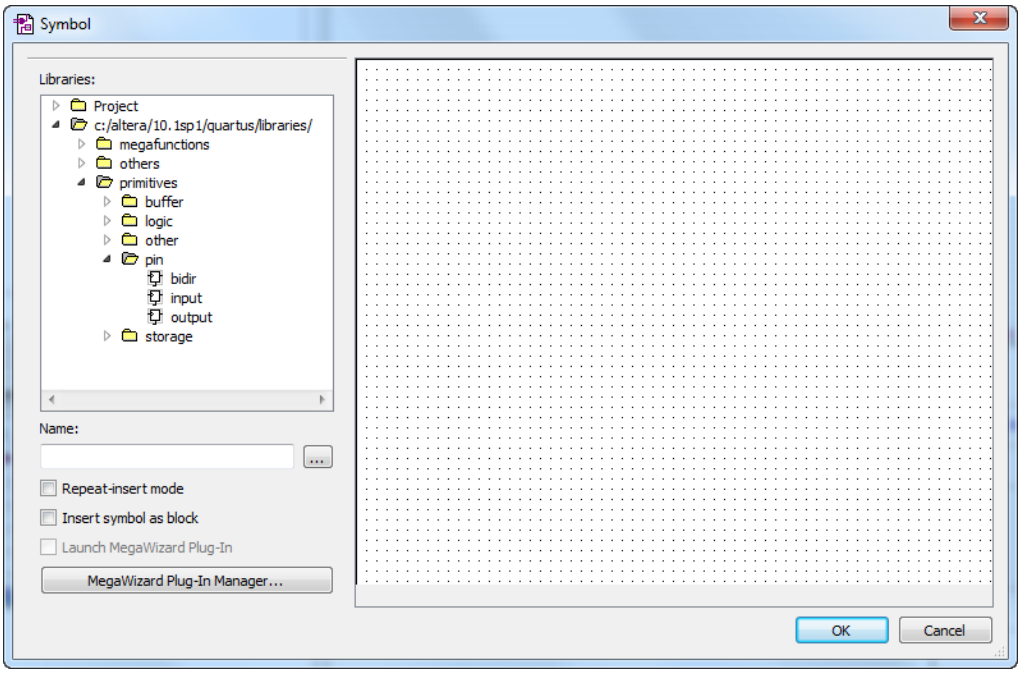

Figure B.64: Symbol Insert, Select Input Pin

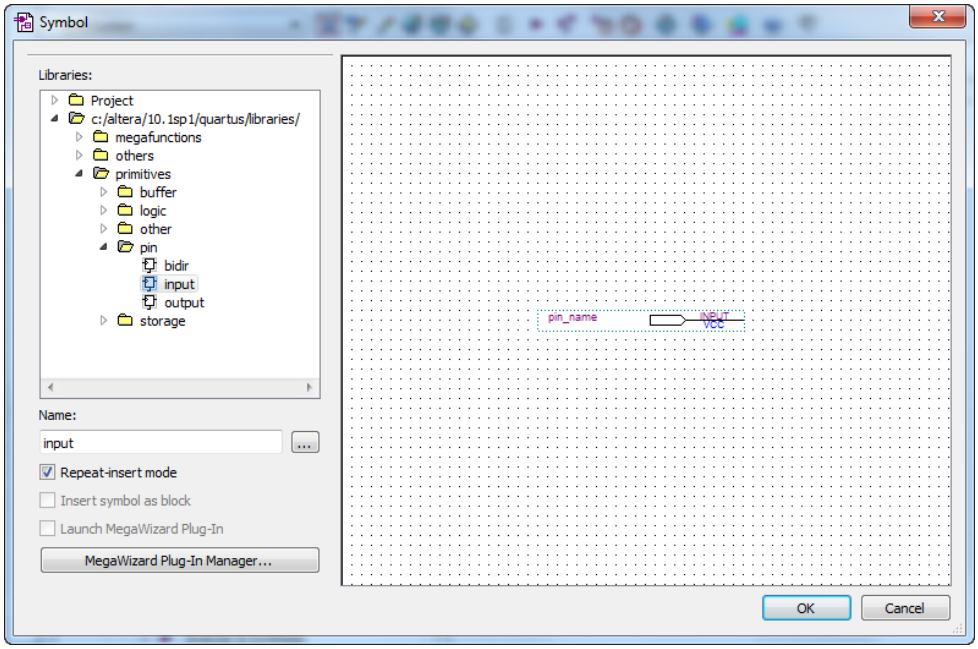

Figure B.65: Symbol Insert, Input Pin, repeat-insert mode

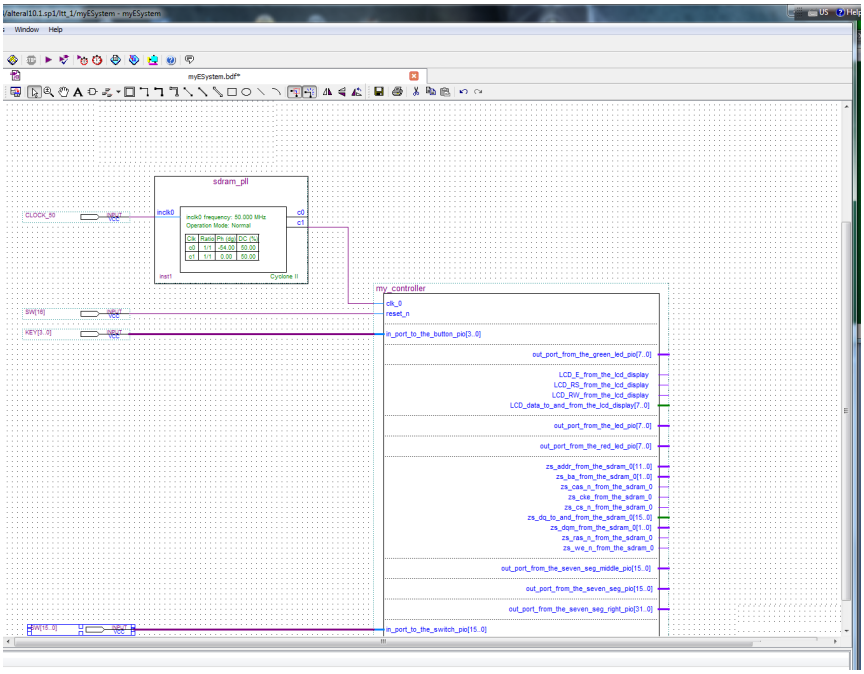

Figure B.66: Insert input pins and Rename

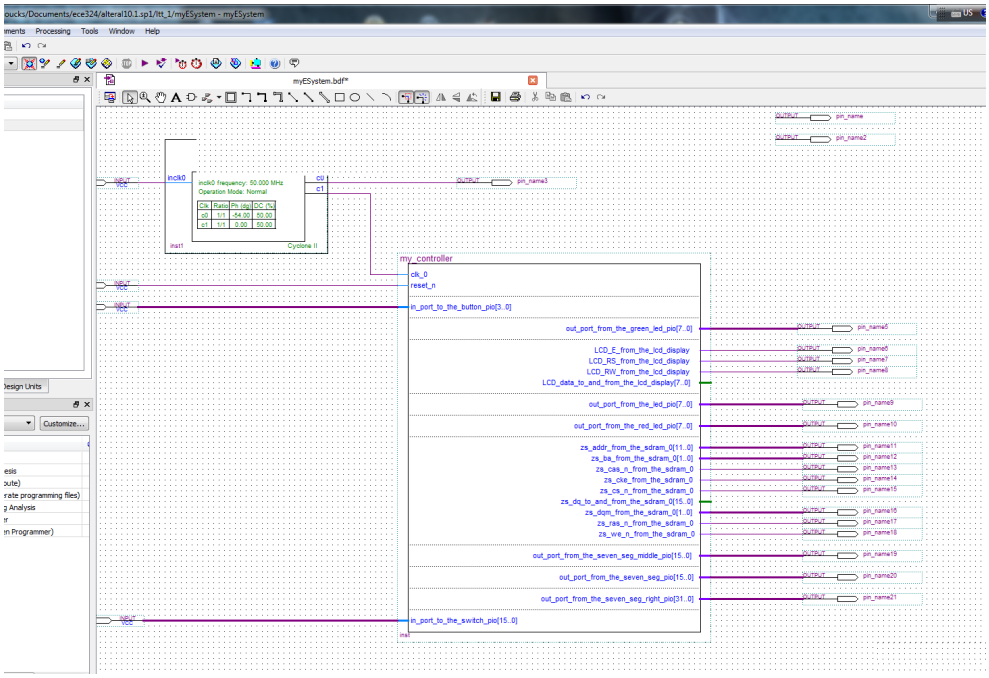

Figure B.67: Insert Output Pins and Rename

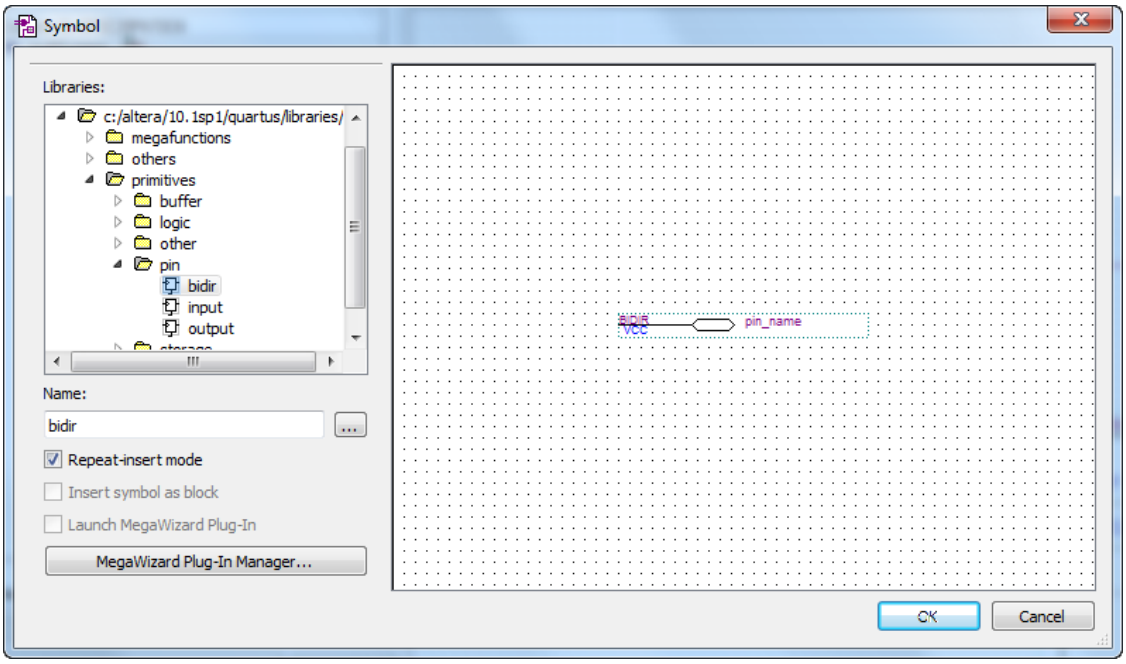

Figure B.68: Select BIDIR primitive

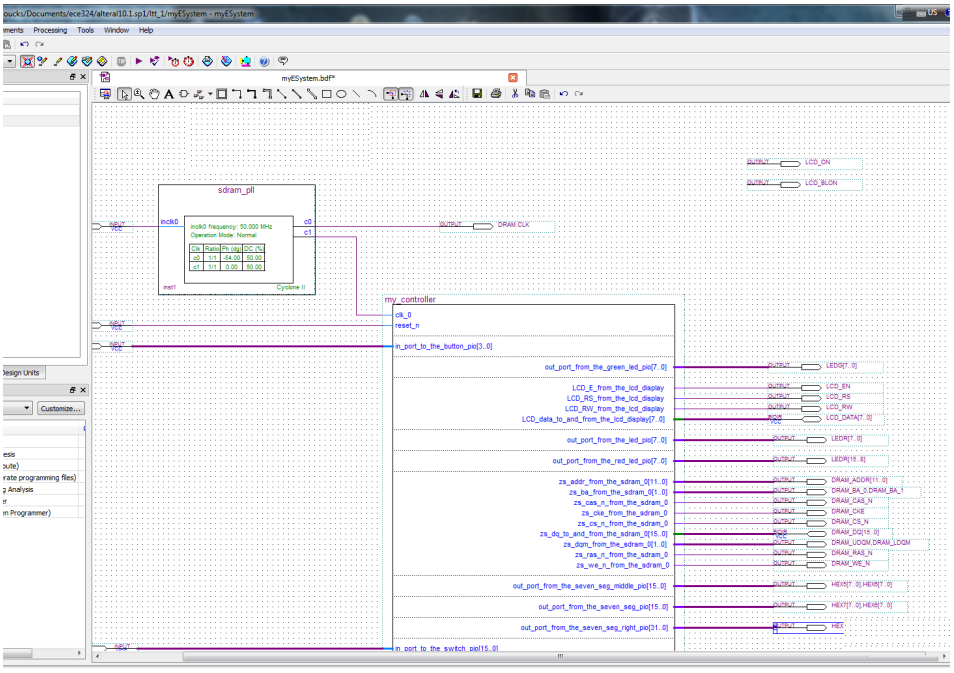

Figure B.69: After Insert BIDIR primitive

VCC

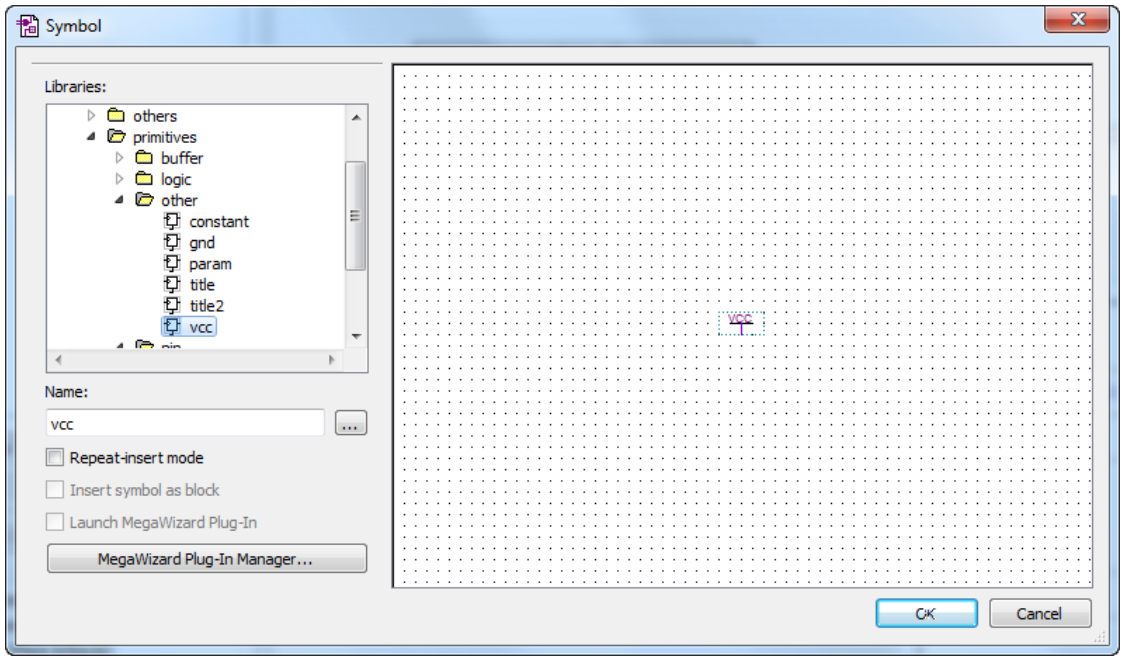

Figure B.70: Select VCC primitive

After VCC, note LCD pins

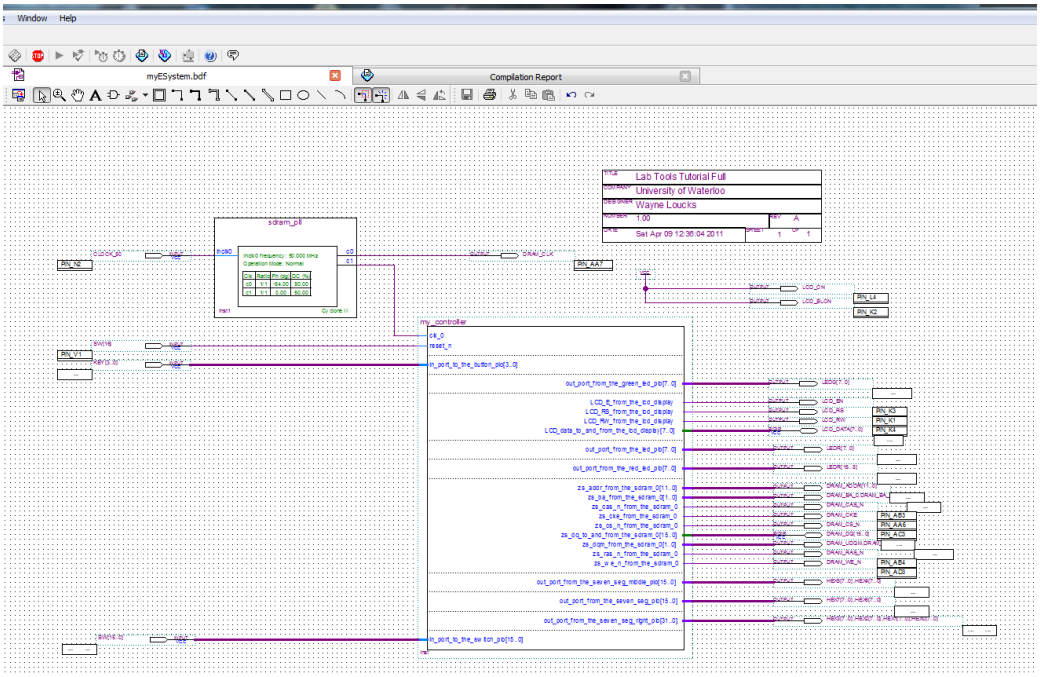

Figure B.71: Completed BDF

Set unused pins

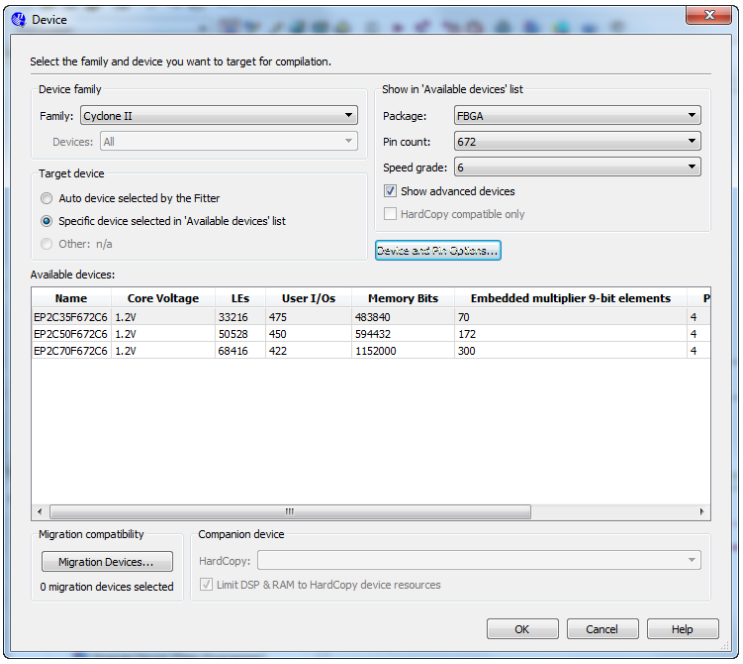

Figure B.72: Setting unused pin - 1

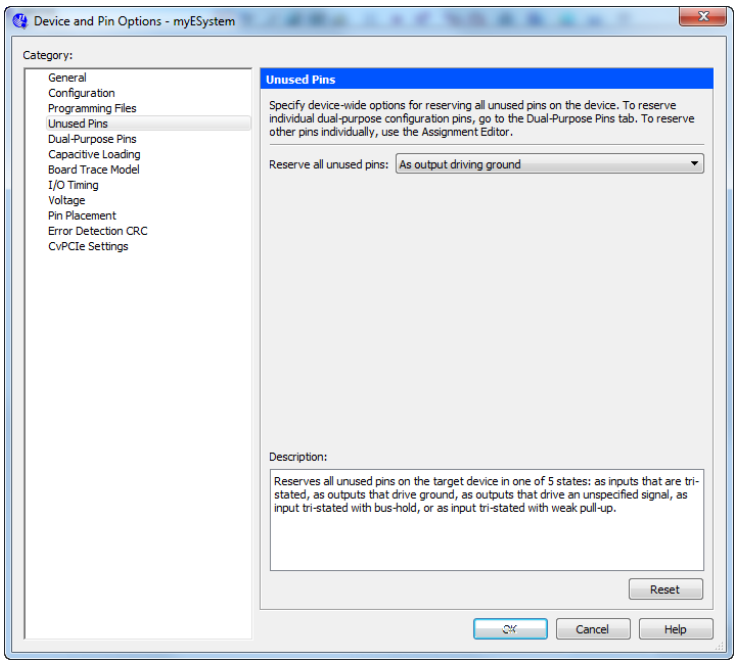

Figure B.73: Setting unused pin - 2

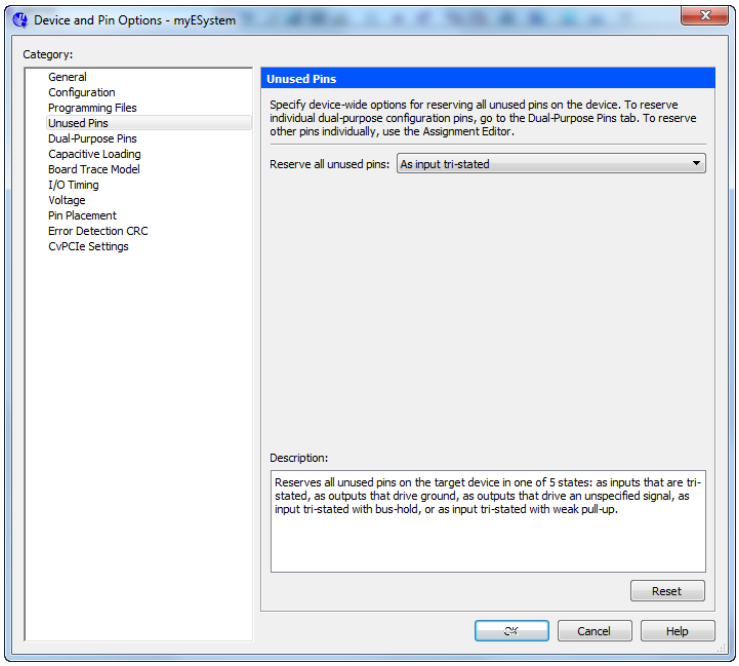

Figure B.74: Setting unused pin - 3

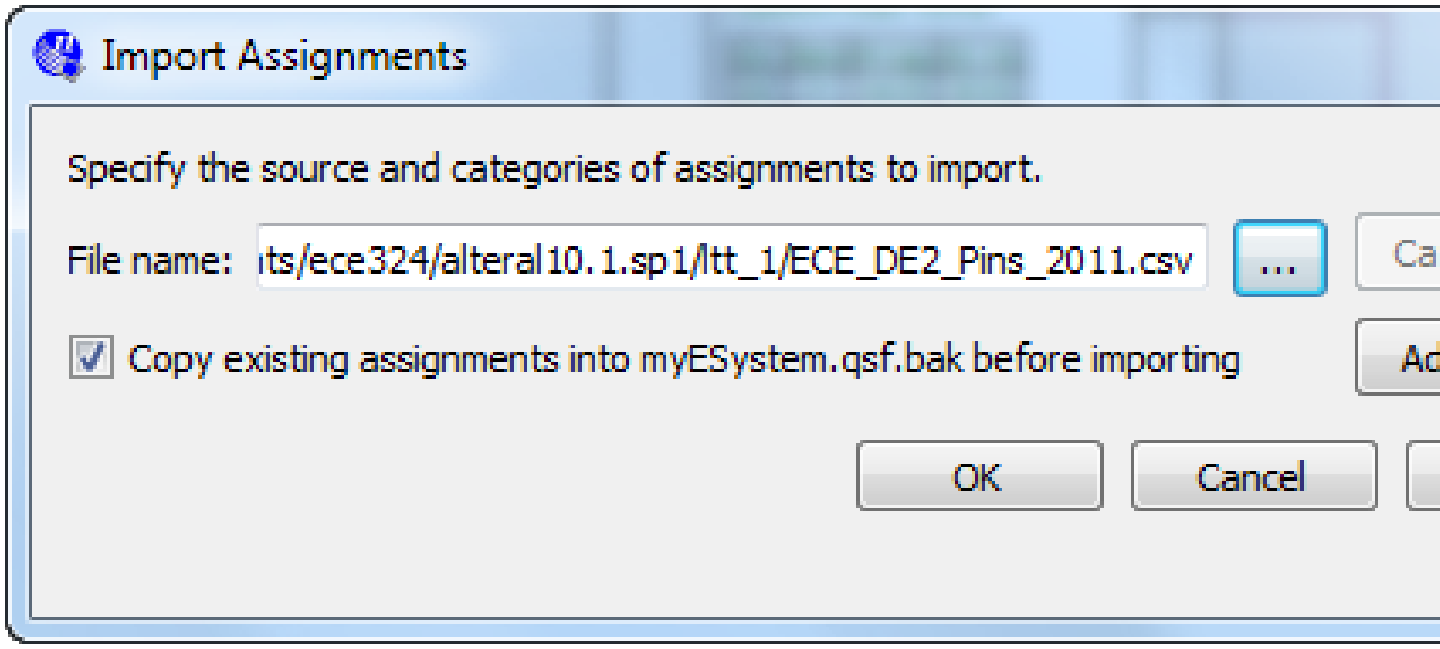

Figure B.75: Input Pin Assignments

Appendix C

Nios Peripheral Quickstart Descriptions

C.1 PIO

PIO is an acronym for parallel input output. A PIO is a port the allows parallel communication with an internal or external device.

C.1.1 Description of Use

PIO allows parallel communications between devices. Communication could be synchronous or asynchronous, and can generate interrupts at edges or levels. Using parallel communications means handshaking signals are not required.

C.1.2 PIO Configuration Settings

The possible configuration settings are shown in Table C.1. These settings will be different depending on the device.

Configuration Option	Setting
Width	1-32 bits
Direction	input, output, both
Edge Capture	synchronous, asynchronous rising, falling or either edge
IRQ	level, edge, none

Table C.1: PIO Configuration Settings

C.1.3 Common PIO Problems

Before generating your Nios II core, make sure that you have included all the required PIO's so that you do not have to re-generate. It is a good idea to provide a PIO for LED's and pushbuttons for debugging purposes. Also avoid level triggered interrupts where possible since it is difficult to avoid multiple interrupt requests. Keep in mind that each bit in the data register can only be read or written depending on whether it is set as an input or an

output. If you try to read from an output only PIO you will not get an error, but the value will be meaningless. If you are having problems with the PIO or the PIO interrupt, perform the following debugging steps before asking for help:

1. Verify that you have enabled the interrupt for *each bit* of the PIO (as required) by changing the interrupt mask
2. Verify that you have installed the interrupt service routine (ISR)
3. Verify that the ISR is being called by setting a flag in the ISR
4. If you cannot exit the ISR, verify that the edge capture register has been cleared by performing a write to the edge capture register