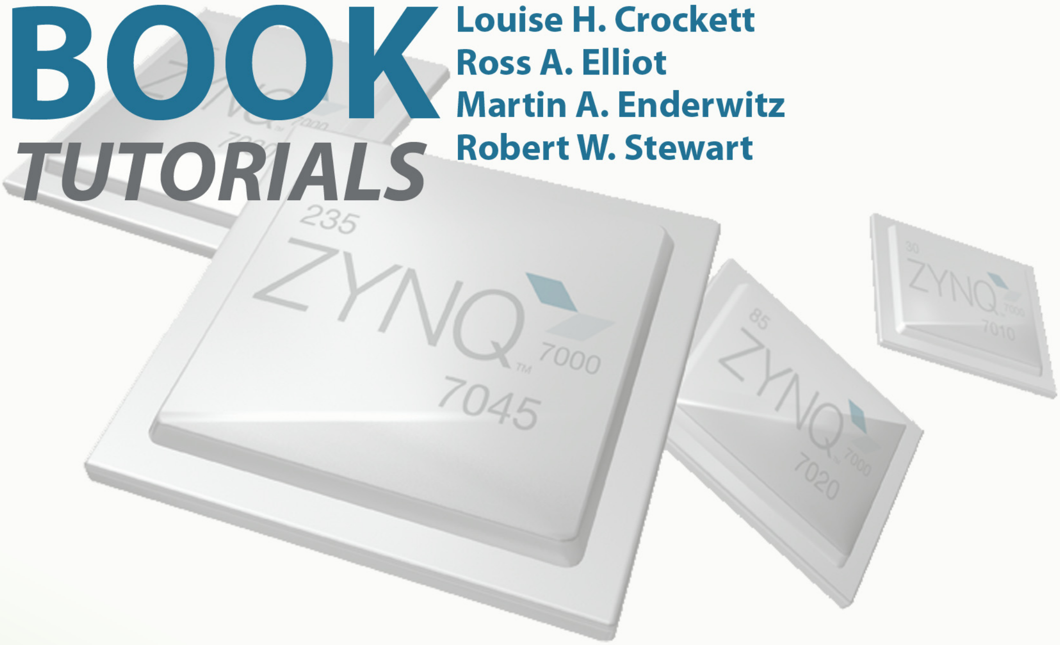
THE ZYNQ® BOOK TUTORIALS

Louise H. Crockett
Ross A. Elliot
Martin A. Enderwitz
Robert W. Stewart



In association with
XILINX®
ALL PROGRAMMABLE™

University of Strathclyde Glasgow

The Zynq Book Tutorials

Louise H. Crockett

Ross A. Elliot

Martin A. Enderwitz

Robert W. Stewart

Department of Electronic and Electrical Engineering

University of Strathclyde

Glasgow, Scotland, UK

v1.2 - September 2014

Acknowledgements

There are a number of people whom we would like to thank specifically for their help and contribution of feedback with regards to the practical tutorials.

Once again, our sincerest thanks must go to Cathal McCabe of Xilinx University Program, who has not only provided vital feedback and support in the creation of the tutorial material, but has also coordinated the distribution of those materials to others. Our thanks must also go to Austin Lesea and Y.C. Wang at Xilinx for taking the time to attempt the tutorial exercises at an early stage, and for providing us with valuable feedback and suggestions. We also greatly appreciate the support of our colleagues from University of Strathclyde, Iain Chalmers, Sarunas Kalade, and David Northcote, who have likewise been a great help in working through the tutorials and feeding back their experiences.

Louise Crockett, Ross Elliot, Martin Enderwitz, and Bob Stewart.

July 2014.

Contents

1. First Designs on Zynq 1

Creating a First IP Integrator Design 4
Creating a Zynq System in Vivado 10
Creating a Software Application in the SDK 20

2. Next Steps in Zynq SoC Design 29

Expanding the Basic IP Integrator Design 32
Creating a Zynq System with Interrupts in Vivado 35
Creating a Software Application in the SDK 45
Adding a Further Interrupt Source 52

3. Designing With Vivado High Level Synthesis 57

Creating Projects in Vivado HLS 60
Design Optimisation in Vivado HLS 66
Interface Synthesis 78

4. IP Creation 81

Creating IP in HDL 84
Creating IP in MathWorks HDL Coder 103
Creating IP in Vivado HLS 111

5. Adventures with IP Integrator 119

Importing IP to the Vivado IP Catalog 122
ZedBoard Audio in Vivado IP Integrator 129
Creating an Audio Software Application in SDK 139

The Zynq Book Tutorial 1

First Designs on Zynq

v1.3, April 2014

Revision History

Date	Version	Changes
14/06/2013	1.0	First release for Vivado Design Suite version 2013.1
19/06/2013	1.1	Updated for changes in Vivado Design Suite version 2013.2
27/01/2014	1.2	Updated for changes in Vivado Design Suite version 2013.4
30/04/2014	1.3	Updated for changes in Vivado Design Suite version 2014.1

Introduction

This tutorial will guide you through the process of creating a first Zynq design using the Vivado™ Integrated Development Environment (IDE), and introduce the IP Integrator environment for the generation of a simple Zynq processor design to be implemented on the ZedBoard. The Software Development Kit (SDK) will then be used to create a simple software application which will run on the Zynq's ARM Processing System (PS) to control the hardware that is implemented in the Programmable Logic (PL).

The tutorial is split into three exercises, and is organised as follows:

Exercise 1A - This exercise will guide you through the process of launching Vivado IDE and creating a project for the first time. The various stages of the *New Project Wizard* will be introduced.

Exercise 1B - In this exercise, we will use the project that was created in Exercise 1A to build a simple Zynq embedded system with the graphical tool, IP Integrator, and incorporating existing IP from the Vivado IP Catalog. A number of design aids will be used throughout this exercise, such as the Board Automation feature which automates the customisation of IP modules for a specified device or board; in this case we will be using the ZedBoard Zynq Evaluation and Development Kit. The Designer Assistance feature, which assists with the connections between the Zynq PS and the IP modules in the PL will also be demonstrated.

Once the design is finished, a number of stages will be undertaken to complete the hardware system and generate a bitstream for implementation in the PL. The completed hardware design will then be exported to the Software Development Kit (SDK) for the development of a simple software application in **Exercise 1C.**

Exercise 1C - In this short third exercise, the SDK will be introduced, and a short software application will be created to allow the Zynq processor to interact with the IP implemented in the PL. A connection to the hardware server that allows the SDK to communicate with the Zynq processors will be established. The software drivers that are automatically created by the Vivado IDE for IP modules will be explored and integrated into the software application, before finally building and executing the software application on the ZedBoard.

NOTE: Throughout all of the practical tutorial exercise we will be using C:\Zynq_Book as the working directory. If this is not suitable, you can substitute it for a directory of your choice, but you should be aware that you will be required to make alterations to some source files in order for exercises to complete successfully.

Exercise 1A Creating a First IP Integrator Design

In this exercise we will create a new project in Vivado IDE by moving through the stages of the Vivado IDE *New Project Wizard*.

We will start by launching the Vivado IDE.

(a) Launch Vivado by double-clicking on the Vivado desktop icon: , or by navigating to **Start > All Programs > Xilinx Design Tools > Vivado 2014.1> Vivado 2014.1**

(b) When Vivado loads, you will be presented with the *Getting Started* screen as in Figure 1.1.

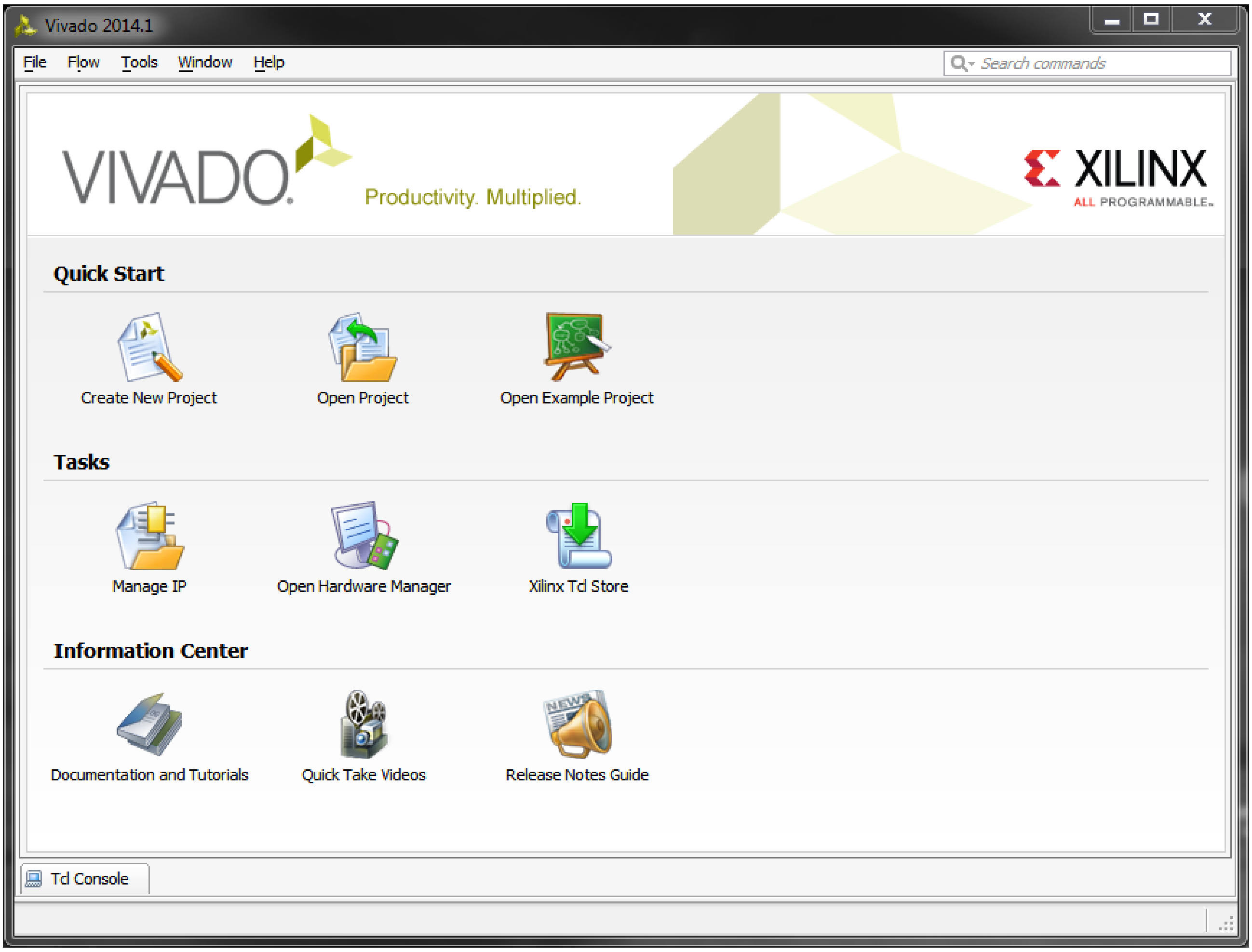
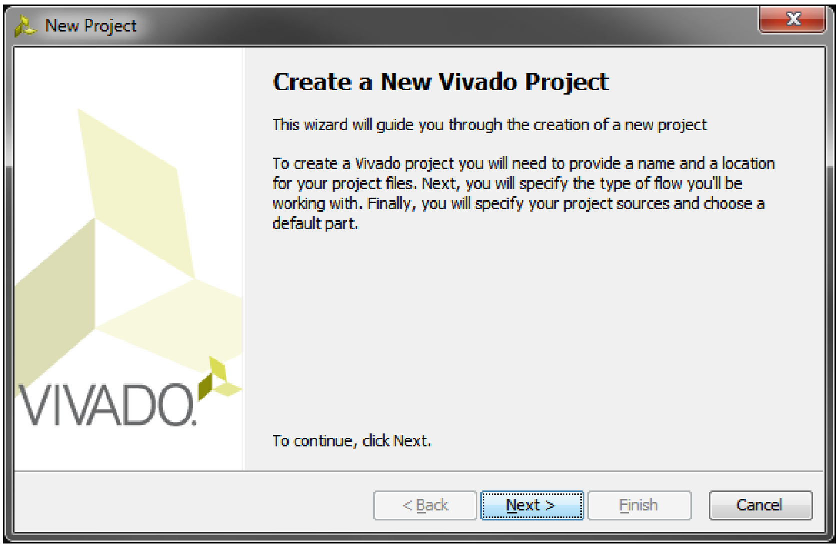


Figure 1.1: Vivado IDE Getting Started Screen

(c) Select the option to ***Create New Project*** and the *New Project Wizard* will open, as in Figure 1.2.



Create a New Vivado Project

This wizard will guide you through the creation of a new project

To create a Vivado project you will need to provide a name and a location for your project files. Next, you will specify the type of flow you'll be working with. Finally, you will specify your project sources and choose a default part.

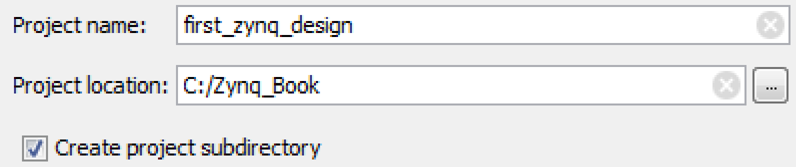
To continue, click Next.

Figure 1.2: New Project Dialogue

Click ***Next***.

(d) At the Project Name dialogue, enter ***first_zynq_design*** as the ***Project name*** and ***C:/Zynq_Book*** as ***Project location***.

Make sure that you select the option to ***Create project subdirectory***. All options should be the same as shown below:



Project name: first_zynq_design

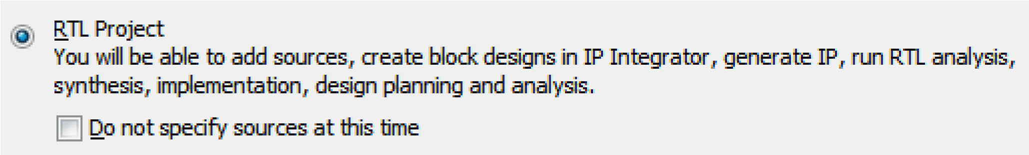
Project location: C:/Zynq_Book

☑ Create project subdirectory

Click ***Next***.

A directory named ***Zynq_Book*** will be created on your ***C drive*** if it did not already exist.

(e) At the *Project Type* dialogue, select ***RTL Project*** and ensure that the option ***Do not specify sources at this time*** is not selected:



● RTL Project
You will be able to add sources, create block designs in IP Integrator, generate IP, run RTL analysis, synthesis, implementation, design planning and analysis.

□ Do not specify sources at this time

Click ***Next***.

(f) Select ***VHDL*** as the ***Target language*** in the *Add Sources* dialogue.

If existing sources, in the form of HDL or netlist files, were to be added to the project they could be imported at this stage.

As we do not have any sources to add to the project, click ***Next***.

(g) The *Add Existing IP (optional)* dialogue will open.

If existing IP sources were to be included in the project, they could be added here.

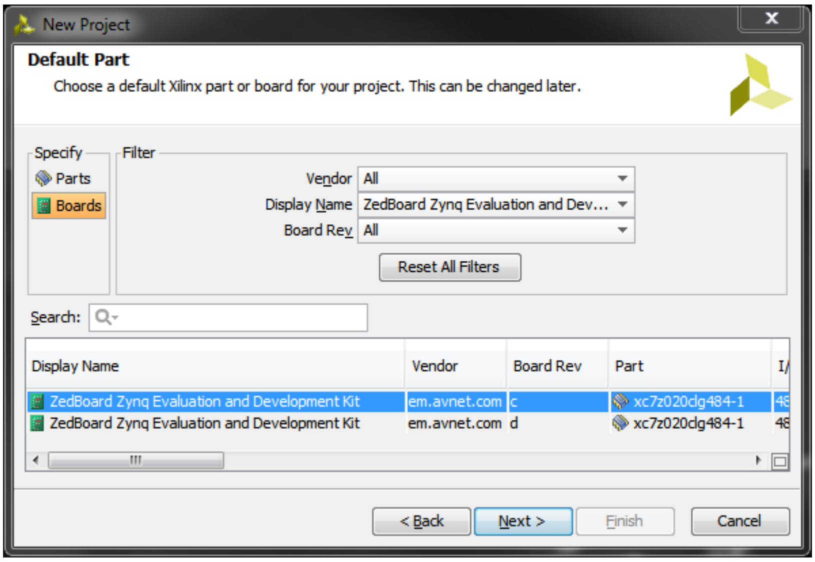
As we do not have any existing IP to add, click ***Next***.

(h) The *Add Constraints (optional)* dialogue will open.

This is the stage where any physical or timing constraints files could be added to the project.

As we do not have any constraints files to add, click ***Next***.

(i) From the *Default Part* dialogue, select ***Boards*** from the *Specify* box and select ***ZedBoard Zynq Evaluation and Development Kit*** from the *Display Name* list and **All** from the *Board Rev* list, as shown in Figure 1.3. Select the appropriate revision for your board (in this case **Rev. C** has been selected).



New Project

Default Part

Choose a default Xilinx part or board for your project. This can be changed later.

Specify: Parts, Boards (selected)

Filter

Vendor: All

Display Name: ZedBoard Zynq Evaluation and Dev...

Board Rev: All

Search:

Display Name	Vendor	Board Rev	Part	I/
ZedBoard Zynq Evaluation and Development Kit	em.avnet.com	c	xc7z020clg484-1	48
ZedBoard Zynq Evaluation and Development Kit	em.avnet.com	d	xc7z020clg484-1	48

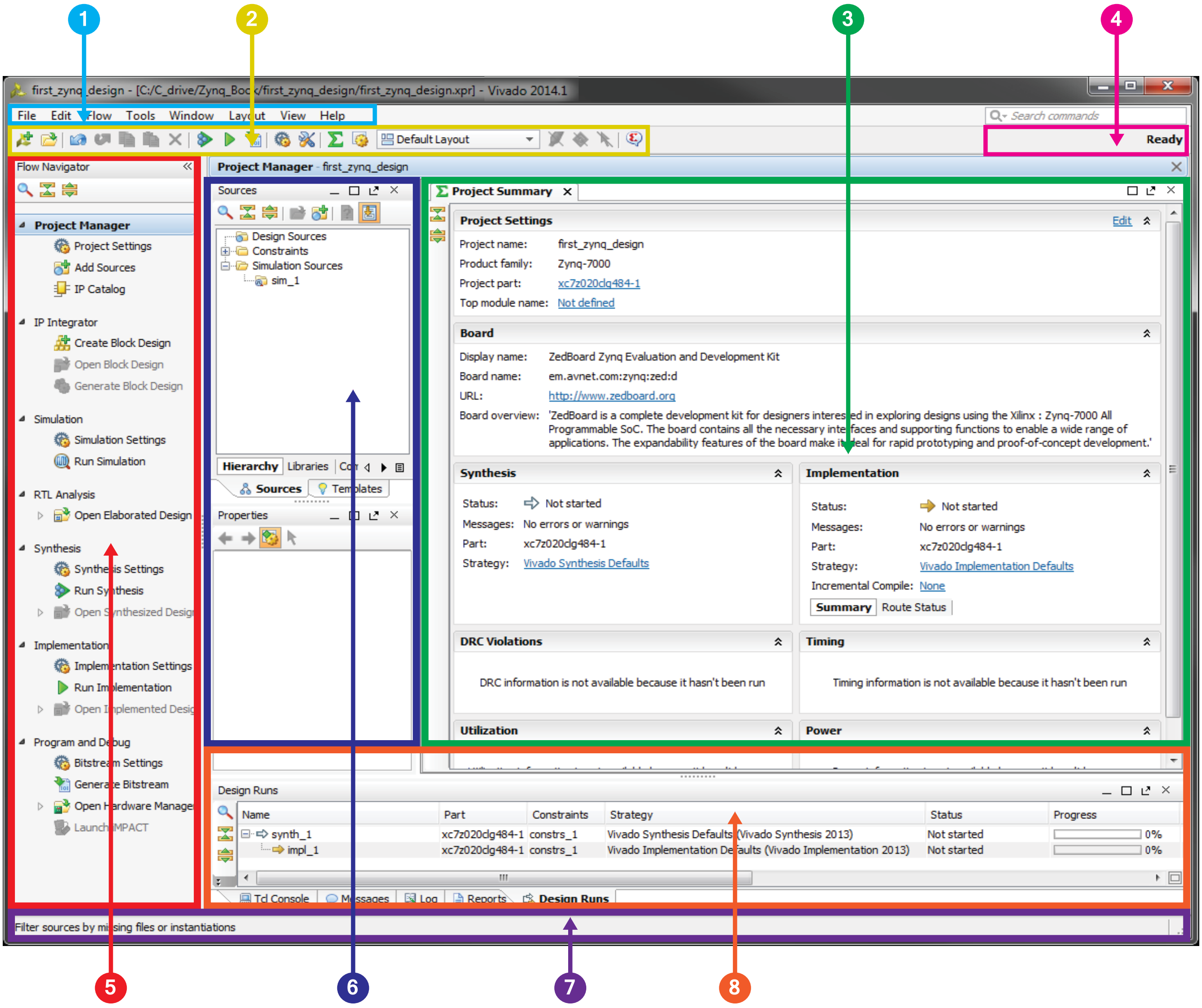
Figure 1.3: Default Part Dialogue Options

Click ***Next***.

(j) In the *New Project Summary* dialogue, review the specified options, and click ***Finish*** to create the project.

Now that we have created our first project in Vivado IDE, we can now move on to creating our first Zynq embedded system design.

Before doing that, the Vivado IDE tool layout should be introduced. The default Vivado IDE environment layout is shown in Figure 1.4 (other layouts can be chosen by selecting different perspectives).



Project Settings

Project name: first_zynq_design
Product family: Zynq-7000
Project part: xc7z020clg484-1
Top module name: Not defined

Board

Display name: ZedBoard Zynq Evaluation and Development Kit
Board name: em.avnet.com:zynq:zed:d
URL: http://www.zedboard.org
Board overview: 'ZedBoard is a complete development kit for designers interested in exploring designs using the Xilinx : Zynq-7000 All Programmable SoC. The board contains all the necessary interfaces and supporting functions to enable a wide range of applications. The expandability features of the board make it ideal for rapid prototyping and proof-of-concept development.'

Synthesis

Status: Not started
Messages: No errors or warnings
Part: xc7z020clg484-1
Strategy: Vivado Synthesis Defaults

Implementation

Status: Not started
Messages: No errors or warnings
Part: xc7z020clg484-1
Strategy: Vivado Implementation Defaults
Incremental Compile: None

DRC Violations

DRC information is not available because it hasn't been run

Timing

Timing information is not available because it hasn't been run

Utilization

Power

Design Runs

Name	Part	Constraints	Strategy	Status	Progress
synth_1	xc7z020clg484-1	constrs_1	Vivado Synthesis Defaults (Vivado Synthesis 2013)	Not started	0%
impl_1	xc7z020clg484-1	constrs_1	Vivado Implementation Defaults (Vivado Implementation 2013)	Not started	0%

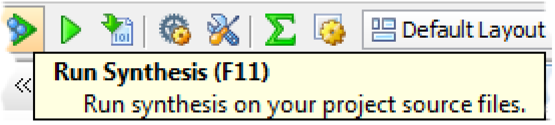
Filter sources by missing files or instantiations

Figure 1.4: Vivado IDE Environment Layout

With reference to the numbered labels in Figure 1.4, the main components of the Vivado IDE environment are:

1. **Menu Bar** - The main access bar gives access to the Vivado IDE commands.
2. **Main Toolbar** - The main toolbar provides easy access to the most commonly used Vivado IDE commands. Tooltips that provide information for each command on the toolbar can be accessed

by hovering the mouse pointer over the corresponding button, as shown in Figure 1.5.



Run Synthesis (F11)
Run synthesis on your project source files.

Figure 1.5: Toolbar tooltips

3. **Workspace** - The workspace provides a larger area for panels which require a greater screen space and those with a graphical interface, such as:
   - Schematic panel
   - Device panel
   - Package panel
   - Text editor panel
4. **Project Status Bar** - The project status bar displays the status of the currently active design.
5. **Flow Navigator** - The Flow Navigator provides easy access to the tools and commands that are necessary to guide your design from start to finish, starting in the *Project Manager* section with design entry and ending with bitstream generation in the *Program and Debug* section. Run commands are available in the *Simulation*, *Synthesis* and *Implementation* sections to simulate, synthesise and implement the active design.
6. **Data Windows Pane** -The Data Windows pane, by default, displays information that relates to design data and sources, including:
   - **Properties window** - Shows information about selected logic objects or device resources.
   - **Netlist window** - Provides a hierarchical view of the synthesised or elaborated logic design.
   - **Sources window** - Shows IP Sources, Hierarchy, Libraries and Compile Order views.
7. **Status Bar** - The status bar displays a variety of information, including:
   - Detailed information regarding menu bar and toolbar commands will be shown in the lower left side of the status bar when the command is accessed.
   - When hovering over an object in the Schematic window with the mouse pointer, the object details appear in the status bar.
   - During constraint and placement creation in the Device and Package windows, validity and constraint type will be shown on the left side of the status bar. Site coordinates and type will

be shown in the right side.

- The task progress of a running task will be relocated to the right side of the status bar when the **Background** button is selected.

8. **Results Window Area** -The Results Window displays the status and results of commands in a set of windows grouped in the bottom of the Vivado IDE environment. As commands progress, messages are generated and log files and reports are created. The related information is shown here. The default windows are:

- **Messages** - Displays all messages for the active design.
- **Tcl Console** - Tcl commands can be entered here an a history of previous commands and outputs are also available.
- **Reports** - Quick access is provided to the reports generated throughout the design flow.
- **Log** -Displays the log files generated by the simulation, synthesis and implementation processes.
- **Design Runs** -Manages runs for the current project.

Additional windows that can appear in this area as required are: Find Results window, Timing Results window and Package Pins window.

With the layout of the Vivado IDE environment introduced, we can now move on to creating the Zynq system.

Exercise 1B Creating a Zynq System in Vivado

In this exercise we will be create a simple Zynq embedded system which implements a General Purpose Input/Output (GPIO) controller in the PL of the Zynq device on the ZedBoard. The GPIO controller will connect to the LEDs. It will also be connected to the Zynq processor via an AXI bus connection, allowing the LEDs to be controlled by a software application which we will create in Exercise 1C.

A graphical representation of the Zynq embedded design is provided in Figure 1.6.

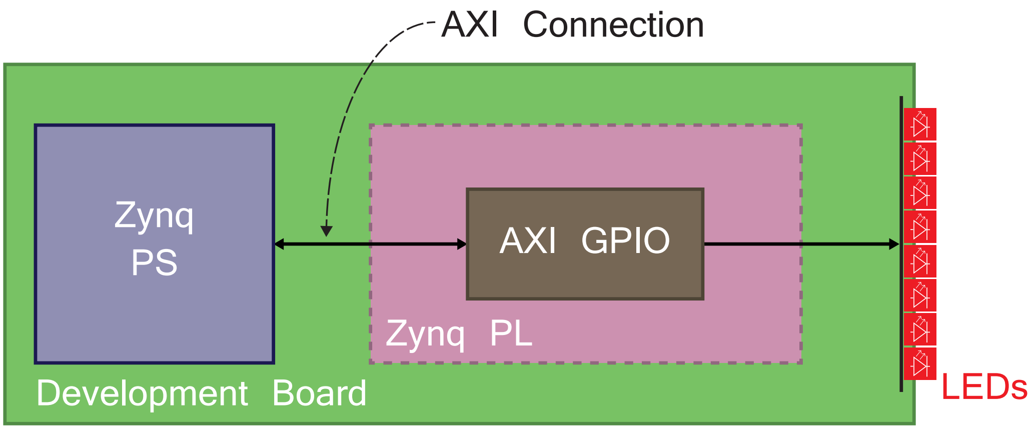


Figure 1.6: Zynq Embedded Design for Exercise 1B

We will begin by creating a new Block Design in Vivado IDE.

(a) In the *Flow Navigator* window, select ***Create Block Design*** from the *IP Integrator* section, as in Figure 1.7:

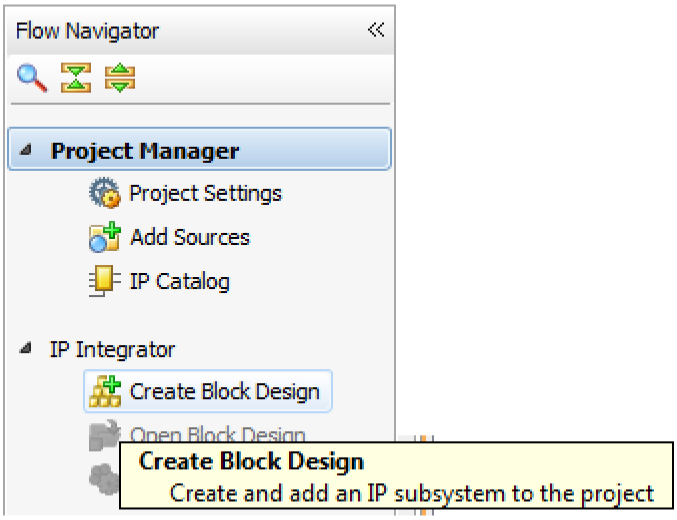
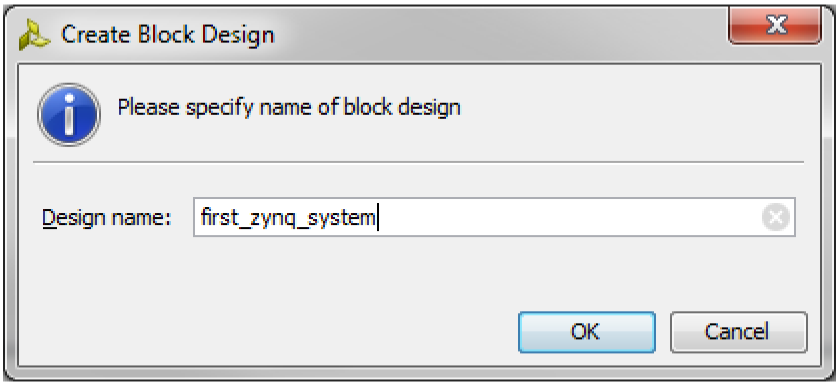


Figure 1.7: Creating a new Block Design in Flow Navigator

The *Create Block Design* dialogue will open.

(b) Enter ***first_zynq_system*** in the Design name box, as in Figure 1.8:



Create Block Design

Please specify name of block design

Design name: first_zynq_system

Figure 1.8: Create Block Design dialogue

Click ***OK***. The *Vivado IP Integrator Diagram* canvas will open in the *Workspace*.

The first block that we will add to our design will be a Zynq Processing System.

(c) In the *Vivado IP Integrator Diagram* canvas, right-click anywhere and select ***Add IP***, as in Figure 1.9.

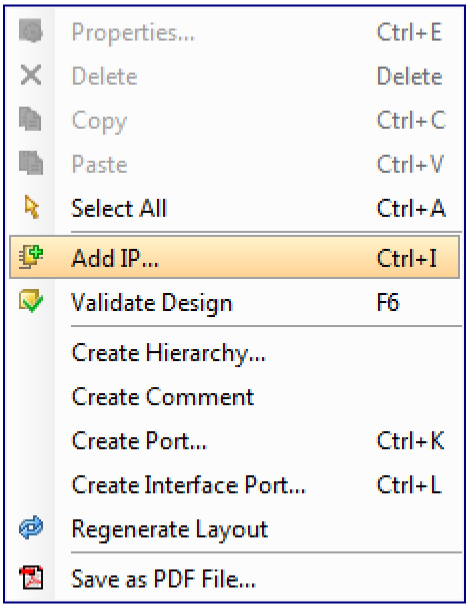
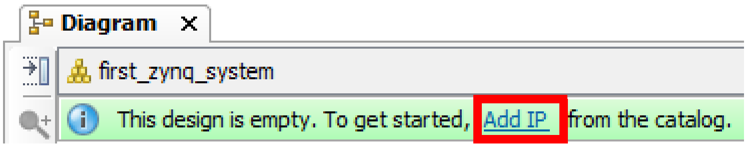


Figure 1.9: Add IP Option

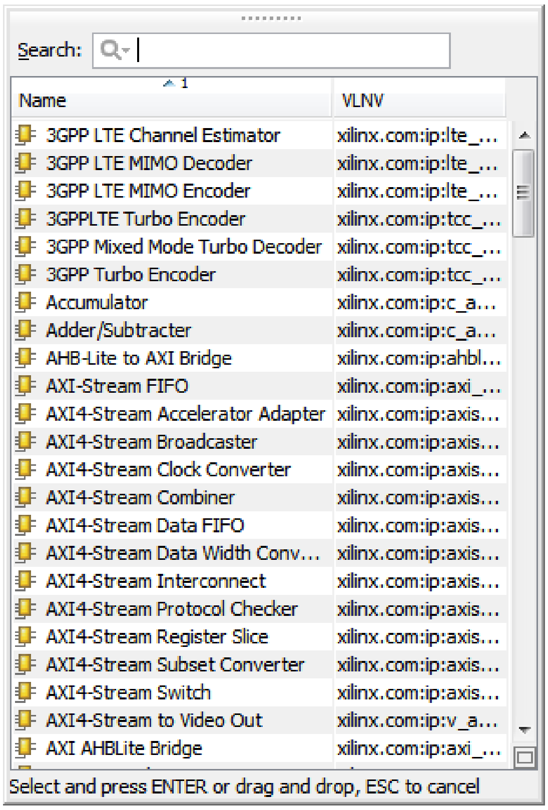
Alternatively, select the ***Add IP*** option from the information message at the top of the canvas, shown in Figure 1.10.



This design is empty. To get started, Add IP from the catalog.

Figure 1.10: Add IP option in IP Integrator canvas information message

The pop-up IP Catalog window will open, as in Figure 1.11.



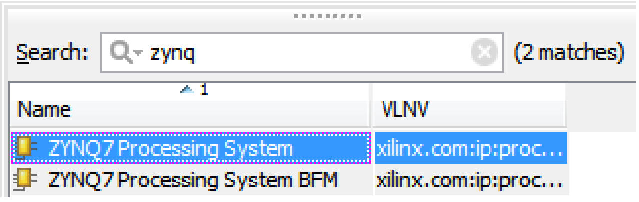
Search:

Name	VLNV
3GPP LTE Channel Estimator	xilinx.com:ip:lte_...
3GPP LTE MIMO Decoder	xilinx.com:ip:lte_...
3GPP LTE MIMO Encoder	xilinx.com:ip:lte_...
3GPPLTE Turbo Encoder	xilinx.com:ip:tcc_...
3GPP Mixed Mode Turbo Decoder	xilinx.com:ip:tcc_...
3GPP Turbo Encoder	xilinx.com:ip:tcc_...
Accumulator	xilinx.com:ip:c_a...
Adder/Subtracter	xilinx.com:ip:c_a...
AHB-Lite to AXI Bridge	xilinx.com:ip:ahbl...
AXI-Stream FIFO	xilinx.com:ip:axi_...
AXI4-Stream Accelerator Adapter	xilinx.com:ip:axis...
AXI4-Stream Broadcaster	xilinx.com:ip:axis...
AXI4-Stream Clock Converter	xilinx.com:ip:axis...
AXI4-Stream Combiner	xilinx.com:ip:axis...
AXI4-Stream Data FIFO	xilinx.com:ip:axis...
AXI4-Stream Data Width Conv...	xilinx.com:ip:axis...
AXI4-Stream Interconnect	xilinx.com:ip:axis...
AXI4-Stream Protocol Checker	xilinx.com:ip:axis...
AXI4-Stream Register Slice	xilinx.com:ip:axis...
AXI4-Stream Subset Converter	xilinx.com:ip:axis...
AXI4-Stream Switch	xilinx.com:ip:axis...
AXI4-Stream to Video Out	xilinx.com:ip:v_a...
AXI AHBLite Bridge	xilinx.com:ip:axi_...

Select and press ENTER or drag and drop, ESC to cancel

Figure 1.11: Pop-up IP Catalog Window

(d) Enter ***zynq*** in the search field and select the ***ZYNQ7 Processing System***, as shown in Figure 1.12, and press the ***Enter*** key on your keyboard.



Search: zynq (2 matches)

Name	VLNV
ZYNQ7 Processing System	xilinx.com:ip:proc...
ZYNQ7 Processing System BFM	xilinx.com:ip:proc...

Figure 1.12: Adding ZYNQ7 Processing System from IP

You should see a similar message to the following in the *Tcl Console* window to confirm that the processing system has indeed been configured correctly:

```
create_bd_cell -type ip -vlnv xilinx.com:ip:processing_system7:5.4
processing_system7_0
```

Messages like this will be displayed in the *Tcl Console* window for all actions carried out on IP Integrator blocks.

The next step is to connect the **DDR** and **FIXED_IO** interface ports on the Zynq PS to the top-level interface ports on the design.

(e) Click the ***Run Block Automation*** option from the *Designer Assistance* message at the top of the Diagram window and select ***/processing_system7_0***, as shown in Figure 1.13.

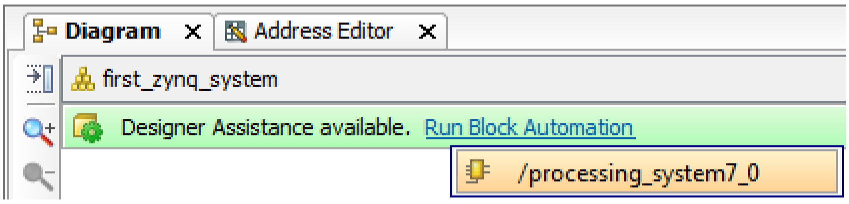


Figure 1.13: Run Block Automation - Processing System

You should notice that the selected item, in this case the ZYNQ7 Processing System, is highlighted in green.

Select ***OK***, to generate the external connections for both the **DDR** and **FIXED_IO** interfaces, ensuring that the option to ***Apply Board Preset*** is selected.

Your block diagram should now resemble Figure 1.14.

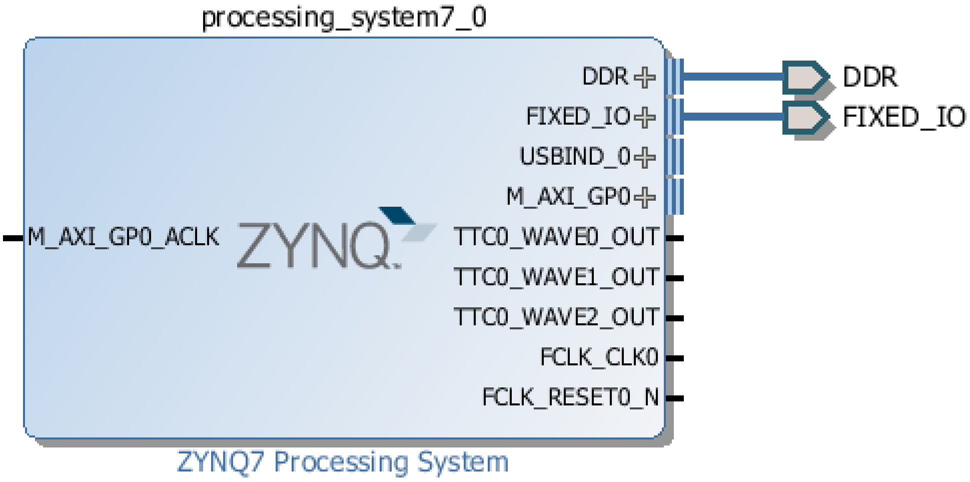


Figure 1.14: ZYNQ7 Processing System External Connections

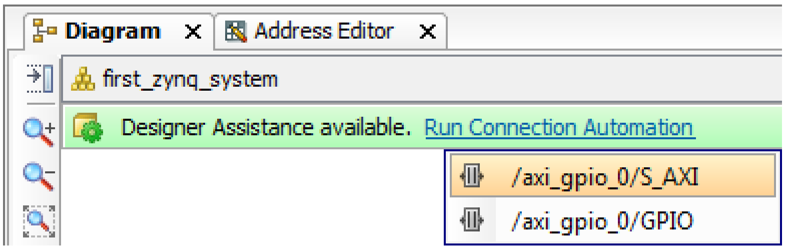
As we are using the ZedBoard platform, and we specified this when creating the project, Vivado will configure the Zynq processor block accordingly.

Now that the main Zynq PS has been added to our design and configured, we can now add further blocks which will be placed in the PL to add functionality to the system. In this case we will only be adding a single block, **AXI GPIO**, to allow us to access the **LEDs** on the **ZedBoard**.

(f) Right-click in an empty area of the *Diagram* window and select ***Add IP***. Enter ***GPIO*** in the search field and add an instance of the ***AXI GPIO*** IP.

We will now use the *IP Integrator Designer Assistance* tool to automate the connection of the **AXI GPIO** block to the **ZYNQ7 Processing System**.

(g) Click ***Run Connection Automation*** from the *Designer Assistance* message at the top of the *Diagram* window and select ***/axi_gpio_0/S_AXI***, as shown Figure 1.15.



Designer Assistance available. Run Connection Automation

Figure 1.15: Run Block Automation - GPIO

This will automate the process of connecting the GPIO to an AXI port, and will automatically instantiate two further IP blocks:

- **Processor System Reset Module** - This provides customised resets for an entire processing system, including the peripherals, interconnect and the processor itself.
- **AXI Interconnect** - Provides an AXI interconnect for the system, allowing further IP and peripherals in the PL to communicate with the main processing system.

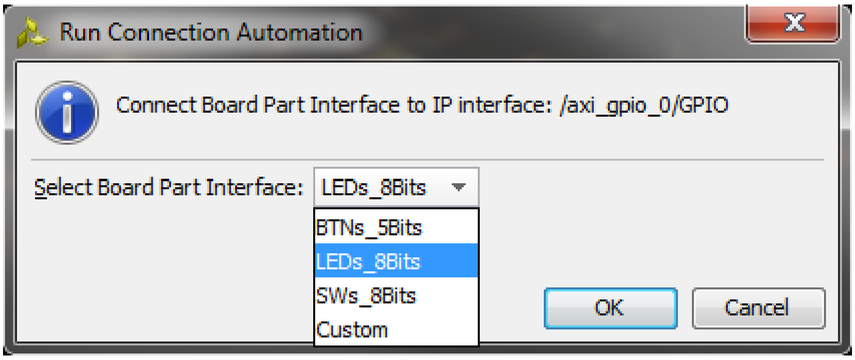
Leave the option for *Clock Connection (for unconnected clks)* to **Auto**, and Click ***OK***.

All connections between the blocks should be made automatically.

One final connection is required to connect the **AXI GPIO** block to the **LEDs** on the ZedBoard. This can also be completed using *Designer Assistance*.

(h) Click ***Run Connection Automation*** from the *Designer Automation* message at the top of the *Diagram* window and select ***/axi_gpio_0/GPIO***.

The Run Connection Automation dialogue will open, as in Figure 1.16.



Run Connection Automation

Connect Board Part Interface to IP interface: /axi_gpio_0/GPIO

Select Board Part Interface: LEDs_8Bits

Figure 1.16: Run Connection Automation Dialogue - GPIO

Select ***LEDs_8Bits*** from the drop-down menu, and click ***OK***.

The gpio interface of the AXI GPIO block will automatically be connected to the LEDs on the ZedBoard.

(i) Click the *Regenerate Layout* button to tidy up the design schematic. Your complete design should resemble Figure 1.17.

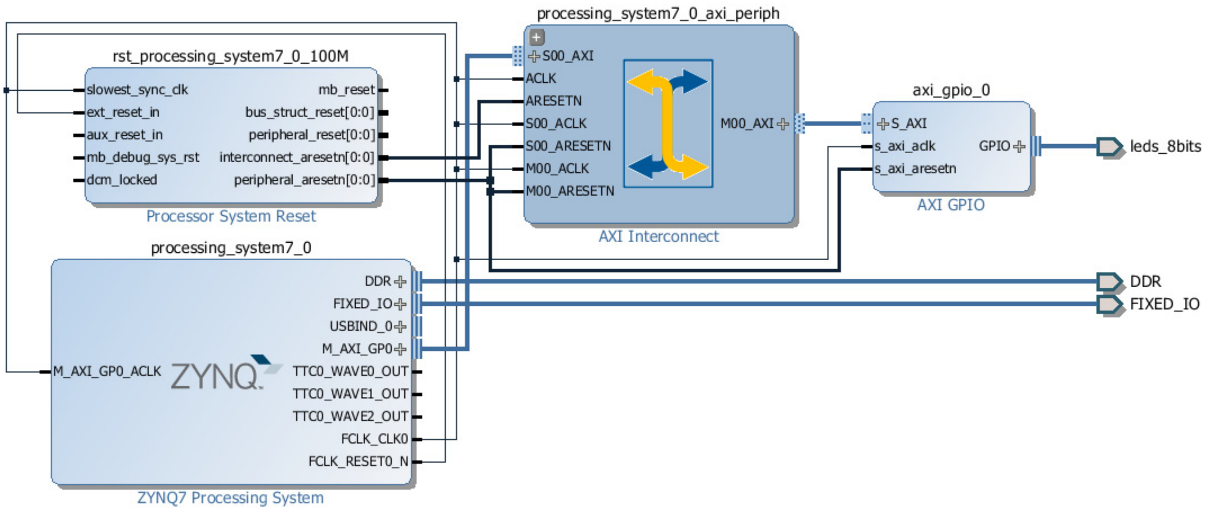
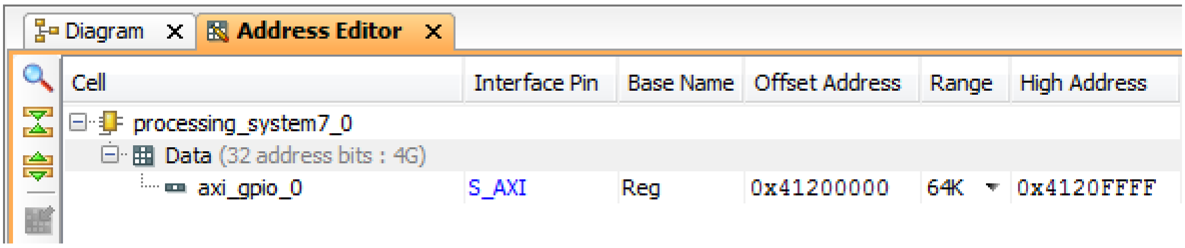


Figure 1.17: Zynq Processor System

The positions of the individual IP blocks in your design may vary slightly from Figure 1.17, but the blocks and their connections should be the same.

IP Integrator will automatically assign a memory map for all IP that is present in the design. We will not be changing the memory map in this tutorial, but for future reference we will take a look at the Address Editor.

(j) Select the Address Editor tab from the top of the Workspace window, as shown in Figure 1.18, and expand the Data group.



Cell	Interface Pin	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 4G)					
axi_gpio_0	S_AXI	Reg	0x41200000	64K	0x4120FFFF

Figure 1.18: Address Editor Tab

You can see that IP Integrator has already assigned a memory map (the mapping of specific sections of memory to the memory-mapped registers of the IP blocks in the PL) to the to the AXI GPIO interface, and that it has a range of **64K**.

Now that our system is complete, we must first validate the design before generating the HDL design files.

(k) Save your design by selecting ***File*** > ***Save Block Design*** from the *Menu Bar*.

(l) Validate the design by selecting ***Tools*** > ***Validate Design*** from the *Menu Bar*. This will run a Design-Rule-Check (DRC).

Alternatively, select the Validate Design button, ✓, from the Main Toolbar, or right-click anywhere in the *Diagram* canvas and select ***Validate Design***.

(m) A *Validate Design* dialogue should appear to confirm that validation of the design was successful. Click ***OK,*** to dismiss the message.

With the design successfully validated, we can now move on to generating the HDL design files for the system.

(n) Switch to the **Sources Tab** by selecting ***Window*** > ***Sources*** from the *Menu Bar*.

(o) Still in the **Sources** window, right-click on the top-level system design, which in this case is **first_zynq_system**, and select ***Create HDL Wrapper***, as shown in Figure 1.19.

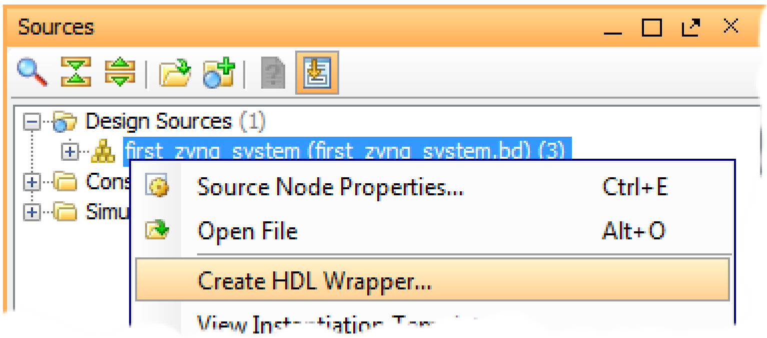


Figure 1.19: Create HDL Wrapper

The *Create HDL Wrapper* dialogue window will open. Select *Let Vivado manage wrapper and auto-update*, and click ***OK***.

This will generate the top level HDL wrapper for our system.

All of the source files for the IP blocks that were used in the IP Integrator block diagram, as well as any relevant constraints files, will be generated during the synthesis process. As we specified VHDL as the target language when creating the project in Exercise 1A, all generated source files will be VHDL.

With all HDL design files generated, the next step in Vivado is to implement our design and generate a bitstream file.

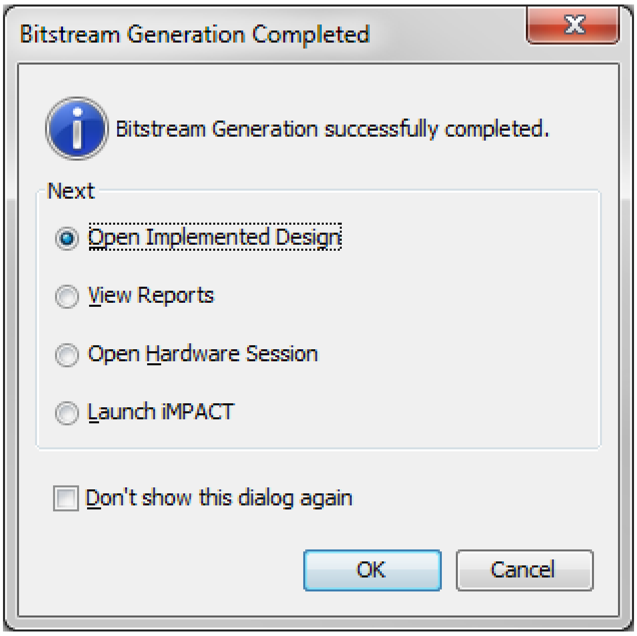
(p) In *Flow Navigator*, click ***Generate Bitstream*** from the *Program and Debug* section.

If a dialogue window appears prompting you to save your design, click ***Save***.

(q) A dialogue window will open requesting that you launch synthesis and implementation before starting the *Generate Bitstream* process. Click ***Yes*** to accept.

The combination of running the synthesis, implementation and bitstream generation processes back-to-back may take a few minutes, depending on the power of your computer system.

(r) Once the bitstream generation is complete a dialogue window will open to inform you that the process has been completed successfully, as in Figure 1.20.



Bitstream Generation Completed
Bitstream Generation successfully completed.
Next
● Open Implemented Design
○ View Reports
○ Open Hardware Session
○ Launch iMPACT
□ Don't show this dialog again
OK
Cancel

Figure 1.20: Bitstream Generation Completion Dialogue Window

Select ***Open Implemented Design***, and click ***OK***.

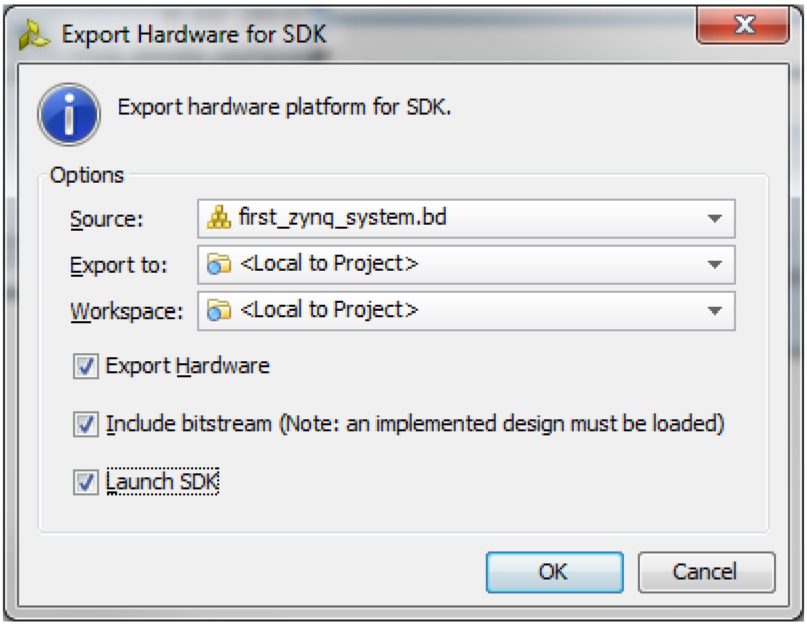
At this point you will be presented with the *Device* view, where you can see the PL resources which are utilised by the design.

With the bitstream generation complete, the building of the hardware image is complete. It must now be exported to a software environment where we will build a software application to control and interact with the custom hardware.

The final step in Vivado is to export the design to the SDK, where we will create the software application that will allow the Zynq PS to control the LEDs on the ZedBoard.

(s) Select ***File > Export > Export Hardware for SDK...*** from the *Menu Bar*.

(t) The *Export Hardware for SDK* dialogue window will open. Ensure that the options to ***Include bitstream*** and ***Launch SDK*** are selected, as in Figure 1.21, and click ***OK***.



Export Hardware for SDK

Export hardware platform for SDK.

Options

Source: first_zynq_system.bd

Export to: <Local to Project>

Workspace: <Local to Project>

☑ Export Hardware

☑ Include bitstream (Note: an implemented design must be loaded)

☑ Launch SDK

Figure 1.21: Export Hardware for SDK

NOTE: For the option to *Include bitstream* to be enabled, an implemented design must be active. This is the reason that we opened the implemented design in Step (r).

This concludes the steps that are required in Vivado IDE. All hardware components of the system have been configured and generated. In the next exercise we will move on to creating a simple software component which will control the system.

Exercise 1C Creating a Software Application in the SDK

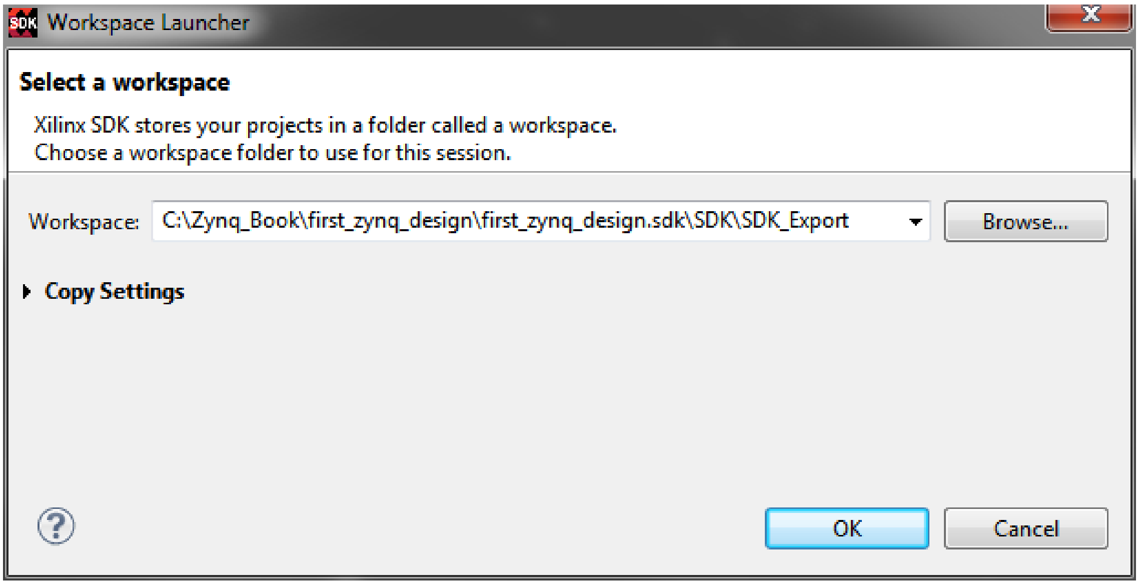
In this exercise we will create a simple software application which will control the LEDs on the ZedBoard. The software application will run on the Zynq processing system and communicate with the AXI GPIO block which is implemented in the PL. We will take a look at the software drivers that are created by IP Integrator, for each of the IP modules, before building and executing the software on the ZedBoard.

The SDK should have opened after the conclusion of Exercise 1B. If it did not open, you can open the SDK by navigating to **Start > All Programs > Xilinx Design Tools > Vivado 2014.1 > Xilinx SDK 2014.1**

When launching the SDK from the start menu, you will need to specify the workspace that was created when the Vivado IP Integrator design was exported in Exercise 1B. It should be:

C:\Zynq_Book\first_zynq_design\first_zynq_design.sdk\SDK\SDK_Export

Enter this in the *Workspace* field of the *Workspace Launcher* dialogue window, as shown in Figure 1.22.



Workspace Launcher

Select a workspace

Xilinx SDK stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

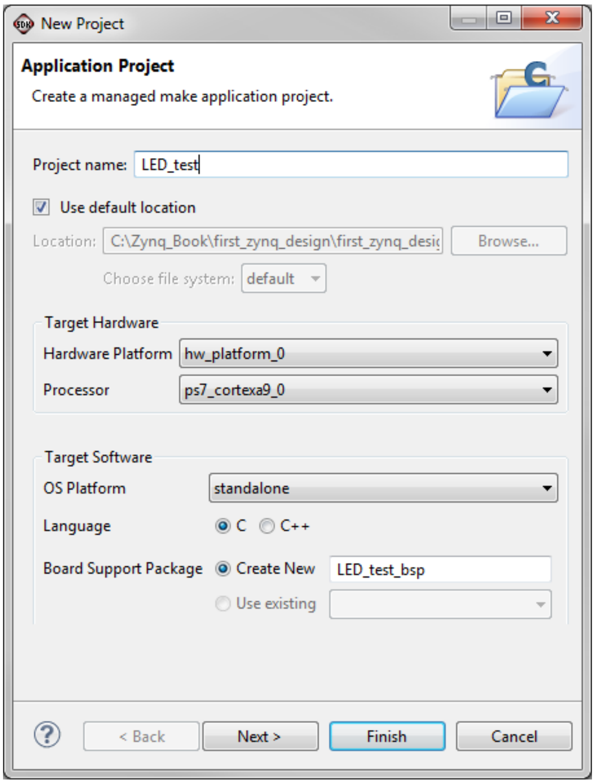
Workspace: C:\Zynq_Book\first_zynq_design\first_zynq_design.sdk\SDK\SDK_Export

Figure 1.22: SDK Workspace Launcher Dialogue Window

With the SDK open, we can begin the creation of our software application.

(a) Select **File > New > Application Project** from the *Menu bar*.

(b) The *New Project* dialogue window will open. Enter ***LED_test*** in the *Project name* field, as shown in Figure 1.23, keeping all other options with the default settings. Click ***Next***.



New Project

Application Project

Create a managed make application project.

Project name: LED_test

☑ Use default location

Location: C:\Zynq_Book\first_zynq_design\first_zynq_desi(

Choose file system: default

Target Hardware

Hardware Platform: hw_platform_0

Processor: ps7_cortexa9_0

Target Software

OS Platform: standalone

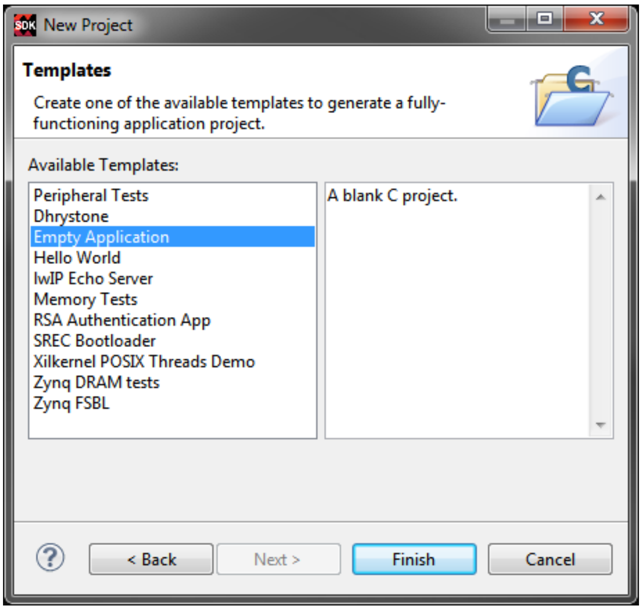
Language: ● C ○ C++

Board Support Package: ● Create New LED_test_bsp

○ Use existing

Figure 1.23: New Application Project Dialogue

(c) At the *New Project Templates* screen, select ***Empty Application***, as in Figure 1.24, and click ***Finish*** to create the project.



New Project

Templates

Create one of the available templates to generate a fully-functioning application project.

Available Templates:

Peripheral Tests
Dhrystone
Empty Application
Hello World
lwIP Echo Server
Memory Tests
RSA Authentication App
SREC Bootloader
Xilkernel POSIX Threads Demo
Zynq DRAM tests
Zynq FSBL

A blank C project.

Figure 1.24: New Project Template Dialogue

NOTE: the new project should open automatically. If it doesn't, you may need to close the Welcome tab in order to view the project.

With the new Application Project created, we can now import some pre-prepared source code for the application.

(d) In the *Project Explorer* panel, expand **LED_test** and highlight the *src* directory. Right-click and select ***Import...***, as shown in Figure 1.25.

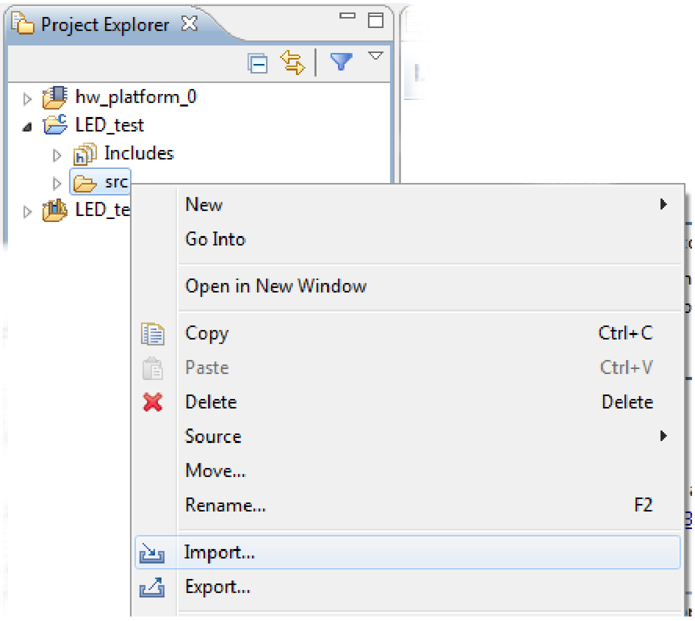
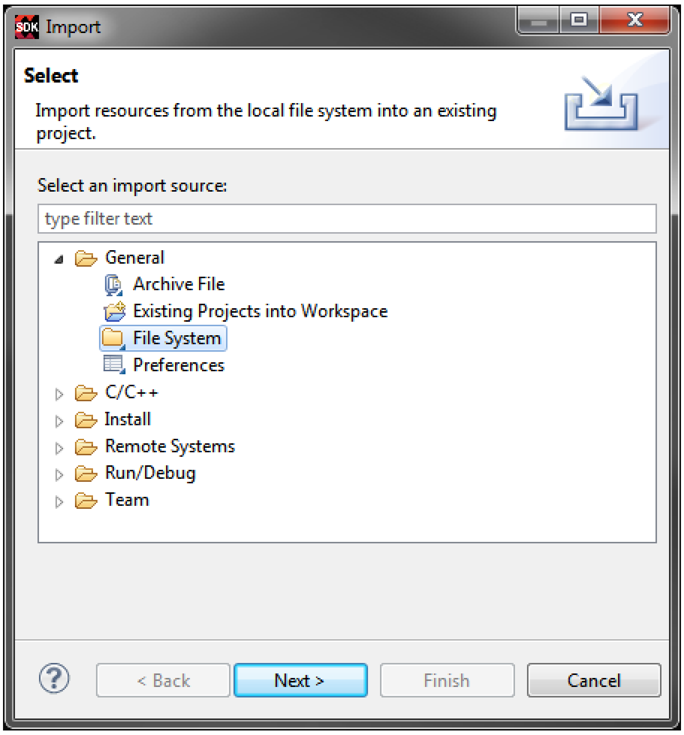


Figure 1.25: Import Source Files to Project

(e) The *Import* window will open. Expand the **General** option and highlight ***File System***, as in Figure 1.26, and click ***Next***.


Import

Select

Import resources from the local file system into an existing project.

Select an import source:

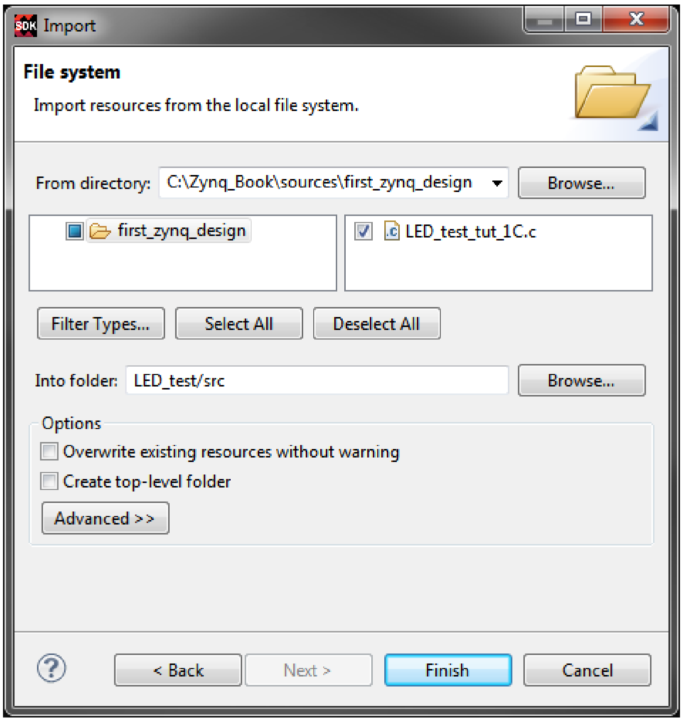
type filter text

Figure 1.26: Import File System

(f) In the *Import File System* window, click the ***Browse...*** button.

(g) Navigate to the directory: **C:\Zynq_Book\sources\first_zynq_design** and click ***OK.***

(h) Select the file **LED_test_tut_1C.c**, as shown in Figure 1.27, and click ***Finish***.



From directory: C:\Zynq_Book\sources\first_zynq_design

■ first_zynq_design

☑ LED_test_tut_1C.c

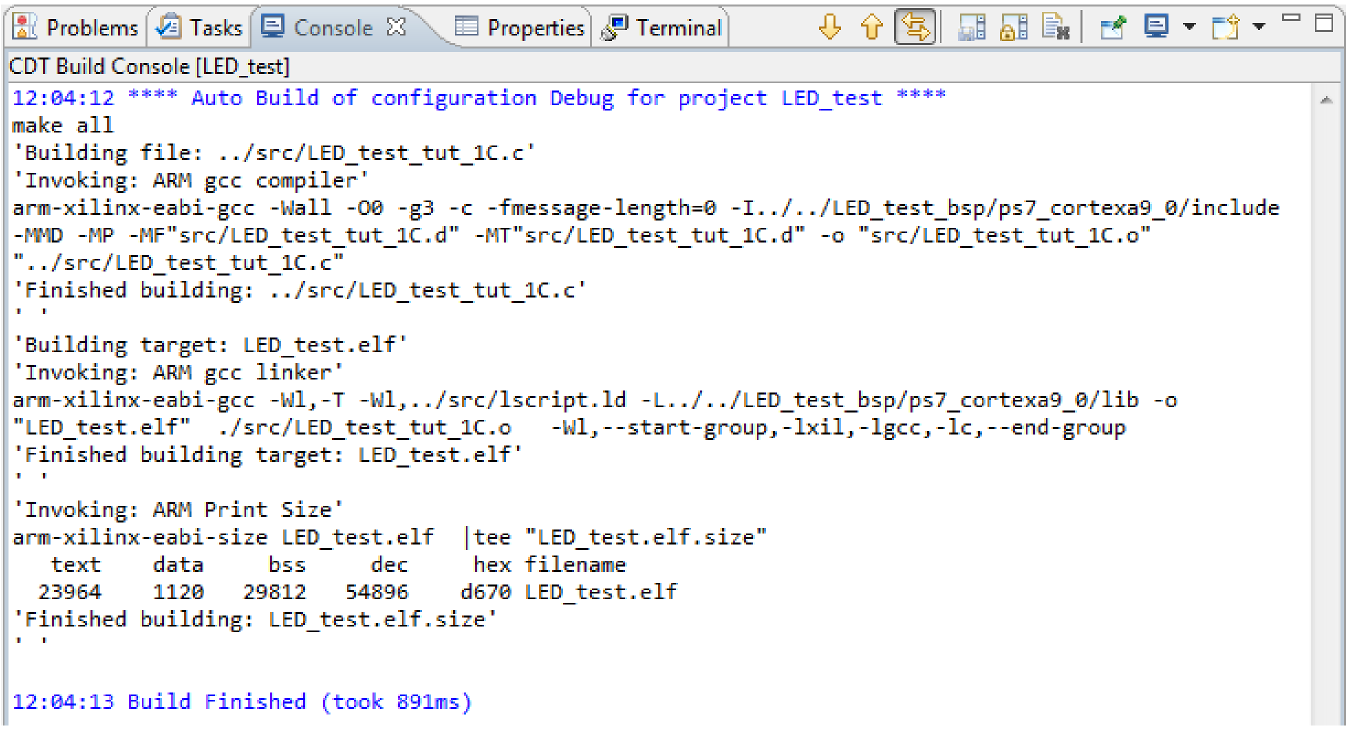
Into folder: LED_test/src

□ Overwrite existing resources without warning

□ Create top-level folder

Figure 1.27: Import C Source File

The C source file will be imported and the project should automatically build. You should see a similar message to Figure 1.28 in the *Console* window.



```
CDT Build Console [LED_test]
12:04:12 **** Auto Build of configuration Debug for project LED_test ****
make all
'Building file: ../src/LED_test_tut_1C.c'
'Invoking: ARM gcc compiler'
arm-xilinx-eabi-gcc -Wall -O0 -g3 -c -fmessage-length=0 -I../../LED_test_bsp/ps7_cortexa9_0/include
-MMD -MP -MF"src/LED_test_tut_1C.d" -MT"src/LED_test_tut_1C.d" -o "src/LED_test_tut_1C.o"
"../src/LED_test_tut_1C.c"
'Finished building: ../src/LED_test_tut_1C.c'
' '
'Building target: LED_test.elf'
'Invoking: ARM gcc linker'
arm-xilinx-eabi-gcc -Wl,-T -Wl,../src/lscript.ld -L../../LED_test_bsp/ps7_cortexa9_0/lib -o
"LED_test.elf"  ./src/LED_test_tut_1C.o   -Wl,--start-group,-lxil,-lgcc,-lc,--end-group
'Finished building target: LED_test.elf'
' '
'Invoking: ARM Print Size'
arm-xilinx-eabi-size LED_test.elf  |tee "LED_test.elf.size"
   text    data     bss     dec     hex filename
  23964    1120   29812   54896    d670 LED_test.elf
'Finished building: LED_test.elf.size'
' '

12:04:13 Build Finished (took 891ms)
```

Figure 1.28: Build Finished Console Message

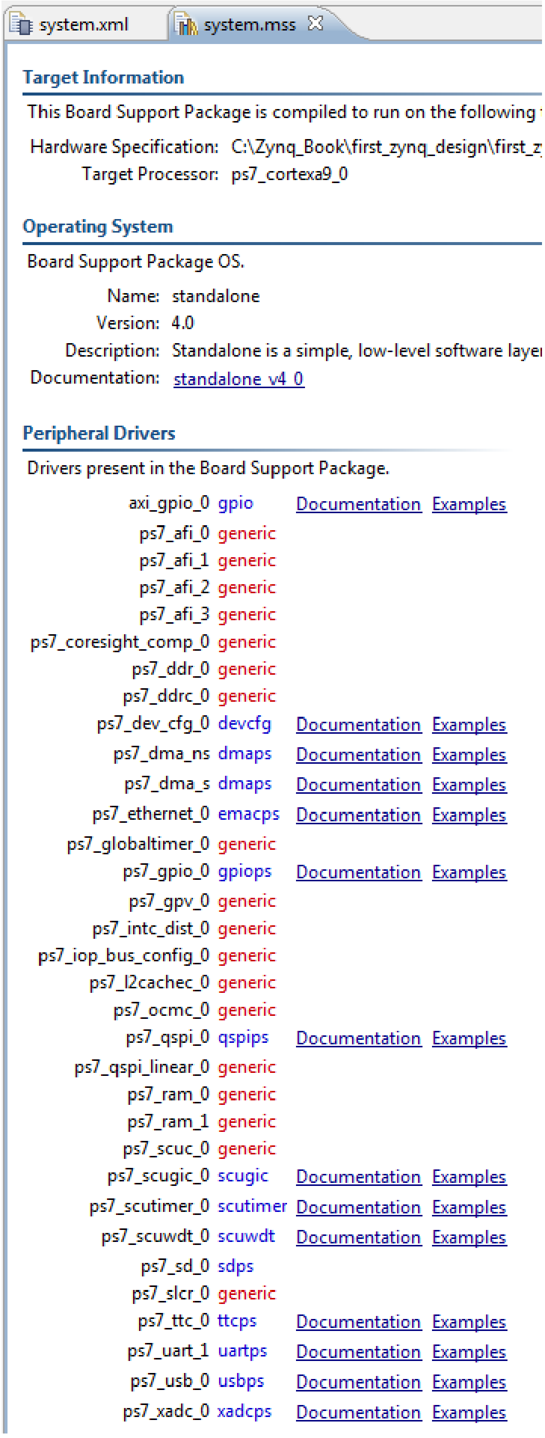
(i) Open the imported source file by expanding the *src* folder and double-clicking on **LED_test_tut_1C.c**, and explore the code.

Note the command `XGpio_Initialize(&Gpio, GPIO_DEVICE_ID);` This is a function provided by the GPIO device driver in the file `xgpio.h`. It initialises the XGpio instance, *Gpio*, with the unique ID of the device specified by `GPIO_DEVICE_ID`.

If you look toward the top of the source file you will see that `GPIO_DEVICE_ID` is defined as `XPAR_AXI_GPIO_0_DEVICE_ID`. The value of `XPAR_AXI_GPIO_0_DEVICE_ID` can be found by opening the file, `xparameters.h`, which is automatically generated by Vivado IDE when exporting a hardware design to the SDK. It contains definitions of all the hardware parameters of the system.

The function, `XGpio_SetDataDirection(&Gpio, LED_CHANNEL, 0xFF);` is also provided by the GPIO device driver, and sets the direction of the specified GPIO port. As we are specifying the LEDs in this case, it is specifying an output. Bits set to '0' are output, and bits set to '1' are input. As there are 8 LEDs, by setting the LED channel direction to a value of `0x00`, or `00000000` in binary, we are setting all 8 LEDs as outputs.

Further information on the peripheral drivers can be found by selecting the **system.mss** tab. A list of all the peripherals in the system is provided, along with links to available documentation and examples, as shown in Figure 1.29.



Target Information

This Board Support Package is compiled to run on the following

Hardware Specification: C:\Zynq_Book\first_zynq_design\first_z
Target Processor: ps7_cortexa9_0

Operating System

Board Support Package OS.

Name: standalone
Version: 4.0
Description: Standalone is a simple, low-level software layeı
Documentation: standalone_v4_0

Peripheral Drivers

Drivers present in the Board Support Package.

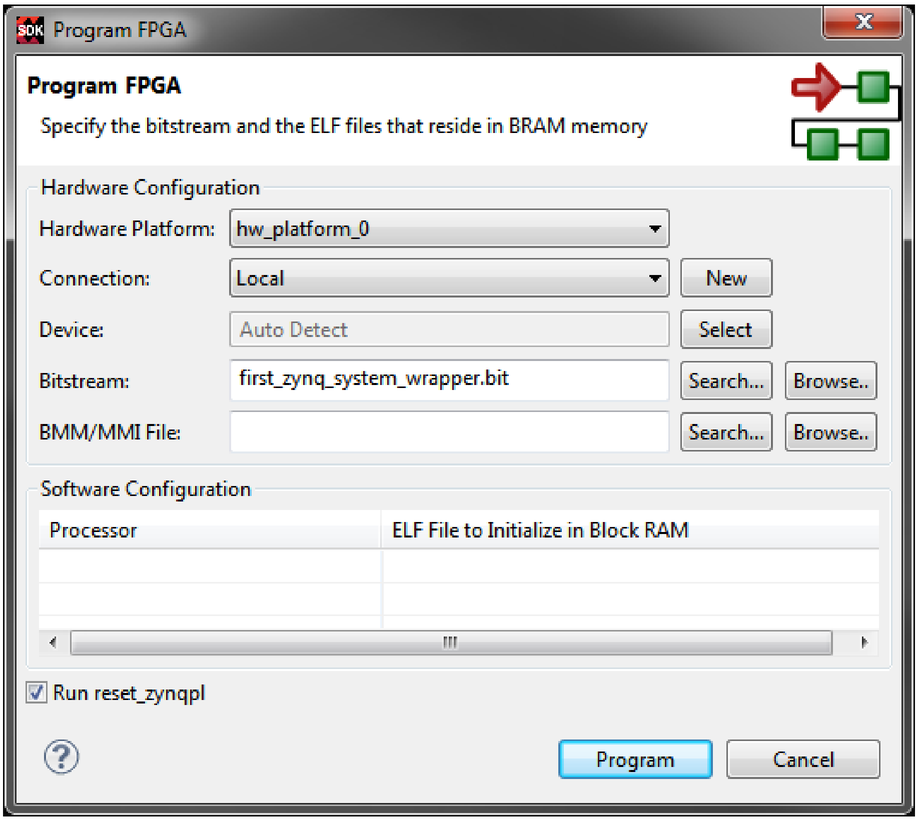
axi_gpio_0	gpio	Documentation	Examples
ps7_afi_0	generic		
ps7_afi_1	generic		
ps7_afi_2	generic		
ps7_afi_3	generic		
ps7_coresight_comp_0	generic		
ps7_ddr_0	generic		
ps7_ddrc_0	generic		
ps7_dev_cfg_0	devcfg	Documentation	Examples
ps7_dma_ns	dmaps	Documentation	Examples
ps7_dma_s	dmaps	Documentation	Examples
ps7_ethernet_0	emacps	Documentation	Examples
ps7_globaltimer_0	generic		
ps7_gpio_0	gpiops	Documentation	Examples
ps7_gpv_0	generic		
ps7_intc_dist_0	generic		
ps7_iop_bus_config_0	generic		
ps7_l2cachec_0	generic		
ps7_ocmc_0	generic		
ps7_qspi_0	qspips	Documentation	Examples
ps7_qspi_linear_0	generic		
ps7_ram_0	generic		
ps7_ram_1	generic		
ps7_scuc_0	generic		
ps7_scugic_0	scugic	Documentation	Examples
ps7_scutimer_0	scutimer	Documentation	Examples
ps7_scuwdt_0	scuwdt	Documentation	Examples
ps7_sd_0	sdps		
ps7_slcr_0	generic		
ps7_ttc_0	ttcps	Documentation	Examples
ps7_uart_1	uartps	Documentation	Examples
ps7_usb_0	usbps	Documentation	Examples
ps7_xadc_0	xadcps	Documentation	Examples

Figure 1.29: Peripheral Documentation and Drivers in system.mss tab

The next step is to program the Zynq PL with the bitstream file that we generated in Exercise 1B.

Ensure that the ZedBoard is powered on and that the JTAG port is connected to the PC via the provided USB-A to USB-B cable.

(j) Download the bitstream to the Zynq PL by selecting **Xilinx Tools > Program FPGA** from the *Menu bar*. The *Program FPGA* window will appear. The Bitstream field should already be populated with the correct bitstream file, as in Figure 1.30.



Program FPGA

Specify the bitstream and the ELF files that reside in BRAM memory

Hardware Configuration

Hardware Platform: hw_platform_0

Connection: Local

Device: Auto Detect

Bitstream: first_zynq_system_wrapper.bit

BMM/MMI File:

Software Configuration

Processor	ELF File to Initialize in Block RAM

☑ Run reset_zynqpl

Figure 1.30: Program FPGA dialogue Window

NOTE: Once the device has successfully been programmed, the *DONE LED* on the ZedBoard will turn blue.

With the Zynq PL successfully configured with the bitstream file, we can now launch our software application on the Zynq PS.

(k) Select the project ***LED_test*** in *Project Explorer*. Right-click and select **Run As > Launch on Hardware (GDB)**.

After a few seconds the LEDs on the ZedBoard should begin to flash between the states highlighted in Figure 1.31.

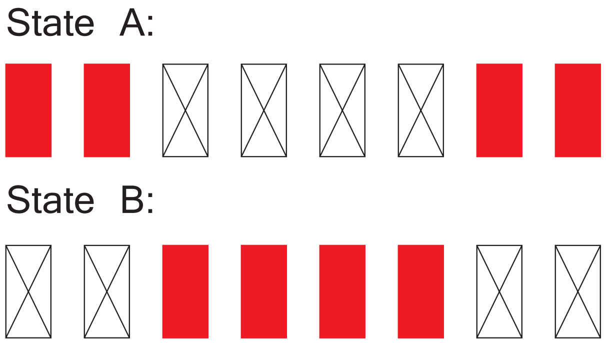


Figure 1.31: LED Flashing States

You have successfully created and executed your first software application on the Zynq processing system.

The Zynq Book Tutorial 2

Next Steps in Zynq SoC Design

v1.2, April 2014

Revision History

Date	Version	Changes
13/09/2013	1.0	First release for Vivado Design Suite version 2013.2
27/01/2014	1.1	Updated for changes in Vivado Design Suite version 2013.4
30/04/2014	1.2	Updates for changes in Vivado Design Suite version 2014.1

Introduction

This tutorial will guide you through the process of creating a Zynq design utilising interrupts. Using the Vivado™ Integraded Development Environment (IDE) and the IP Integrator environment, a simple Zynq™ processor design, to be implemented on the ZedBoard, will be generated. The Software Development Kit (SDK) will then be used to create a simple software application which will run on the Zynq's ARM Processing System (PS) to control the hardware that is implemented in the Programmable Logic (PL). This tutorial leads on from the previous one, expanding on the skills acquired in it.

The tutorial is split into four exercises, and is organised as follows:

Exercise 2A — This exercise provides a further guide to the process of launching Vivado IDE and creating a project using *New Project Wizard*

Exercise 2B — In this exercise, we will use the project that was created in Exercise 2A to build a Zynq embedded system utilising interrupts with IP Integrator and incorporating existing IP from the Vivado IP Catalog. This will expand on previous knowledge gained in creating and connecting a block based system in IP Integrator. The completed design will have an associated bitstream generated and will be exported to the Xilinx SDK for creating of a test application.

Exercise 2C — In the Xilinx SDK, a test software application for the generated hardware system will be created and explained. Running this application on the ZedBoard will demonstrate the function of interrupts and how the application is coded to utilise them.

Exercise 2D — Finally, we will return to the system from Exercise 2B and include an additional source of interrupt, making the necessary connections, and generating a bitstream and exporting to the Xilinx SDK. We will then modify our previous software application to inspect the operation of the altered system.

Exercise 2A Expanding the Basic IP Integrator Design

In this exercise we will expand upon the previous project in Vivado IDE by adding additional GPIO and configuring the system to utilise interrupts. For the sake of clarity and understanding, we will run through the building of a basic system once more. Start by launching the Vivado IDE.

(a) Launch Vivado by double-clicking on the Vivado desktop icon: , or by navigating to **Start > All Programs > Xilinx Design Tools > Vivado 2014.1 > Vivado 2014.1**

(b) When Vivado loads, you will be presented with the *Getting Started* screen as in Figure 2.1.

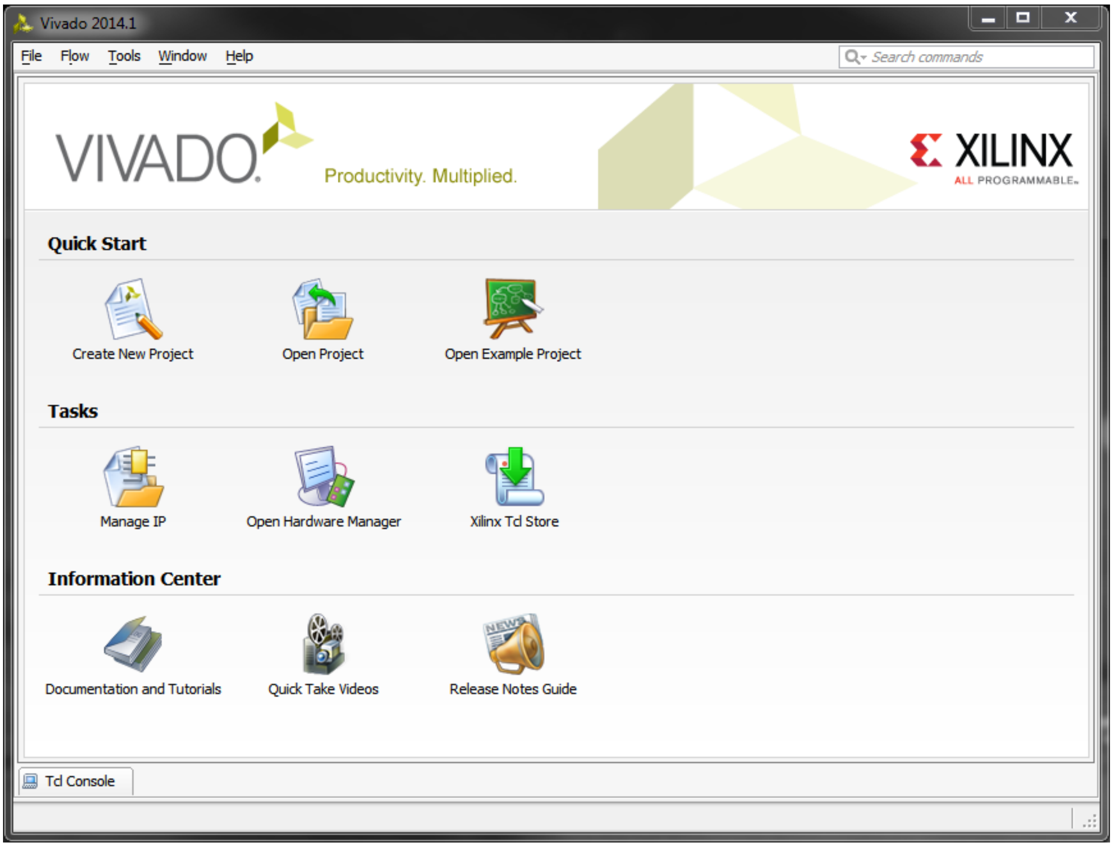
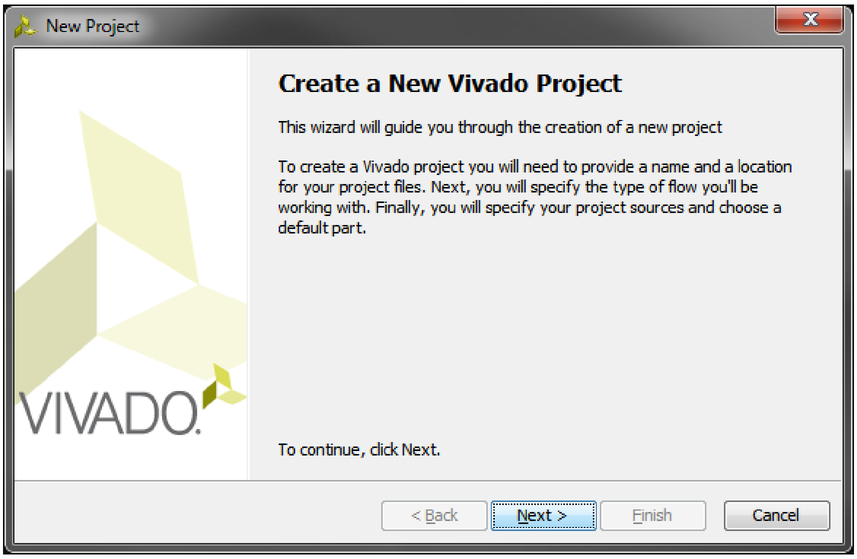


Figure 2.1: Vivado IDE Getting Started screen

(c) Select the option to ***Create New Project*** as in Figure 2.2



Create a New Vivado Project

This wizard will guide you through the creation of a new project

To create a Vivado project you will need to provide a name and a location for your project files. Next, you will specify the type of flow you'll be working with. Finally, you will specify your project sources and choose a default part.

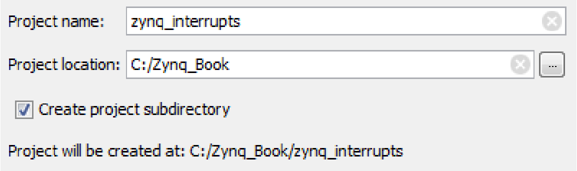
To continue, click Next.

Figure 2.2: New Project dialogue

Click ***Next***.

(d) At the Project Name dialogue, enter ***zynq_interrupts*** as the ***Project name*** and ***C:/Zynq_Book*** as ***Project location***.

Make sure that you select the option to ***Create project subdirectory***. All options should be the same as shown below:



Project name: zynq_interrupts

Project location: C:/Zynq_Book

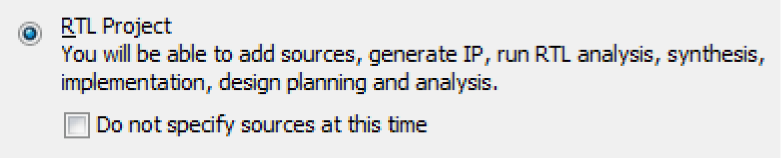
☑ Create project subdirectory

Project will be created at: C:/Zynq_Book/zynq_interrupts

Click ***Next***.

A directory named ***Zynq_Book*** will be created on your ***C drive*** if it did not already exist.

(e) At the *Project Type* dialogue, select ***RTL Project*** and ensure that the option ***Do not specify sources at this time*** is not selected:



● RTL Project
You will be able to add sources, generate IP, run RTL analysis, synthesis, implementation, design planning and analysis.

□ Do not specify sources at this time

Click ***Next***.

(f) Select ***VHDL*** as the ***Target language*** in the *Add Sources* dialogue.

If existing sources, in the form of HDL or netlist files, were to be added to the project they could be imported at this stage.

As we do not have any sources to add to the project, click ***Next***.

(g) The *Add Existing IP (optional)* dialogue will open.

If existing IP sources were to be included in the project, they could be added here.

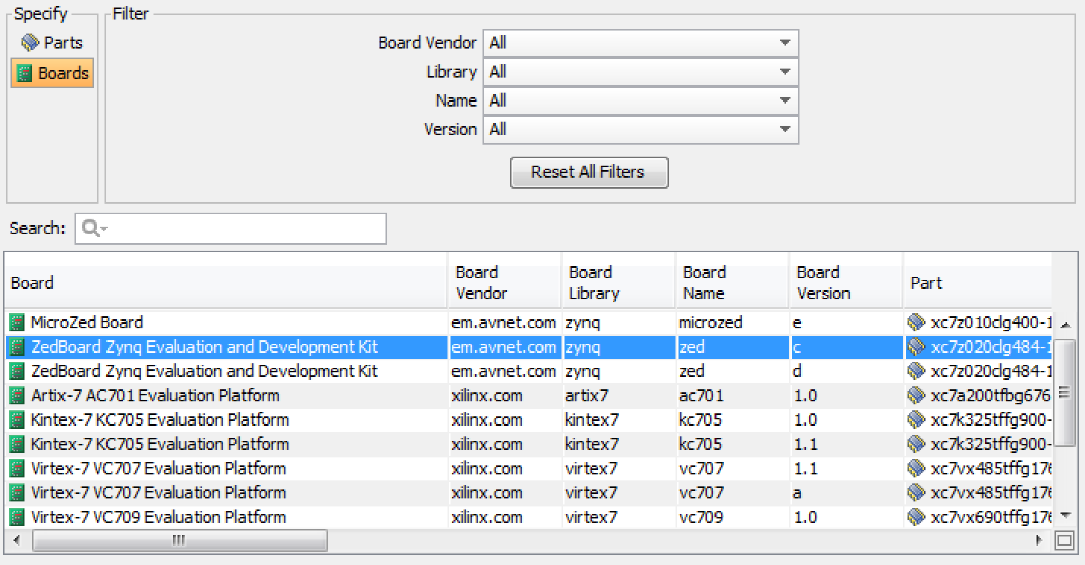
As we do not have any existing IP to add, click ***Next***.

(h) The *Add Constraints (optional)* dialogue will open.

This is the stage where any physical or timing constraints files could be added to the project.

As we do not have any constraints files to add, click ***Next***.

(i) From the *Default Part* dialogue, select ***Boards*** from the *Specify* box and choose ***ZedBoard Zynq Evaluation and Development Kit***, Board Version ***c*** from the list of boards, as shown in Figure 2.3.



Board Vendor: All

Library: All

Name: All

Version: All

Search:

Board	Board Vendor	Board Library	Board Name	Board Version	Part
MicroZed Board	em.avnet.com	zynq	microzed	e	xc7z010clg400-1
ZedBoard Zynq Evaluation and Development Kit	em.avnet.com	zynq	zed	c	xc7z020clg484-1
ZedBoard Zynq Evaluation and Development Kit	em.avnet.com	zynq	zed	d	xc7z020clg484-1
Artix-7 AC701 Evaluation Platform	xilinx.com	artix7	ac701	1.0	xc7a200tfbg676
Kintex-7 KC705 Evaluation Platform	xilinx.com	kintex7	kc705	1.0	xc7k325tffg900-
Kintex-7 KC705 Evaluation Platform	xilinx.com	kintex7	kc705	1.1	xc7k325tffg900-
Virtex-7 VC707 Evaluation Platform	xilinx.com	virtex7	vc707	1.1	xc7vx485tffg17
Virtex-7 VC707 Evaluation Platform	xilinx.com	virtex7	vc707	a	xc7vx485tffg17
Virtex-7 VC709 Evaluation Platform	xilinx.com	virtex7	vc709	1.0	xc7vx690tffg17

Figure 2.3: Default part dialogue options

Click ***Next***.

(j) In the *New Project Summary* dialogue, review the specified options, and click ***Finish*** to create the project.

As in the previous tutorial we will now create the basic Zynq embedded system design before adding and configuring additional IP to utilise hardware interrupts.

Exercise 2B Creating a Zynq System with Interrupts in Vivado

In this exercise we will create a simple Zynq embedded system which implements two General Purpose Input/Output (GPIO) controllers in the PL of the Zynq device on the ZedBoard, one of which uses the push buttons to generate interrupts. The other GPIO controller will connect to the LEDs. Both will also be connected to the Zynq processor via an AXI bus connection, allowing the LEDs to be controlled by a software application which we will create in Exercise 2C.

(a) In the *Flow Navigator* window, select ***Create Block Design*** from the *IP Integrator* section, as in Figure 2.4:

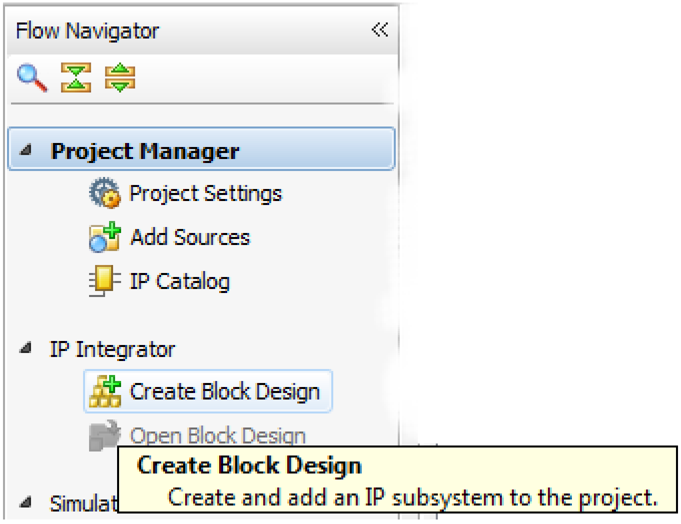
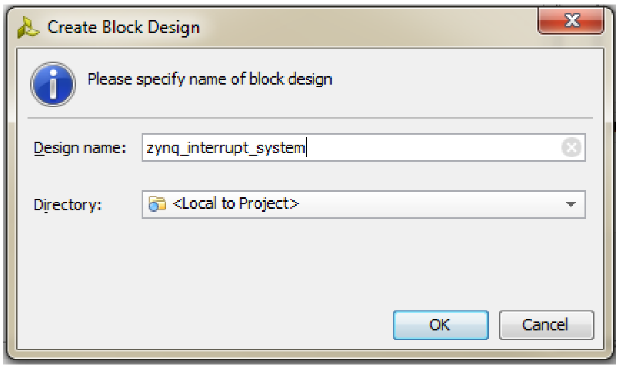


Figure 2.4: Creating a new Block Design in Flow Navigator

The *Create Block Design* dialogue will open.

(b) Enter ***zynq_interrupt_system*** in the Design name box, as in Figure 2.5:



Please specify name of block design

Design name: zynq_interrupt_system

Directory: <Local to Project>

Figure 2.5: Create Block Design dialogue

Click ***OK***. The *Vivado IP Integrator Diagram* canvas will open in the *Workspace*.

The first block that we will add to our design will be a Zynq Processing System.

(c) In the *Vivado IP Integrator Diagram* canvas, right-click anywhere and select ***Add IP***, as in Figure 2.6.

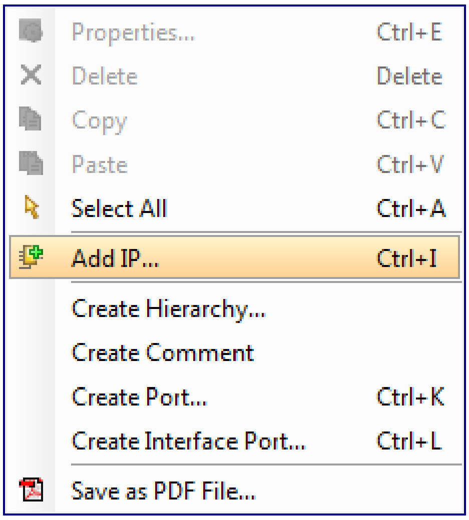
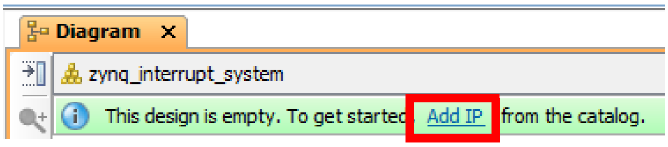


Figure 2.6: Add IP option

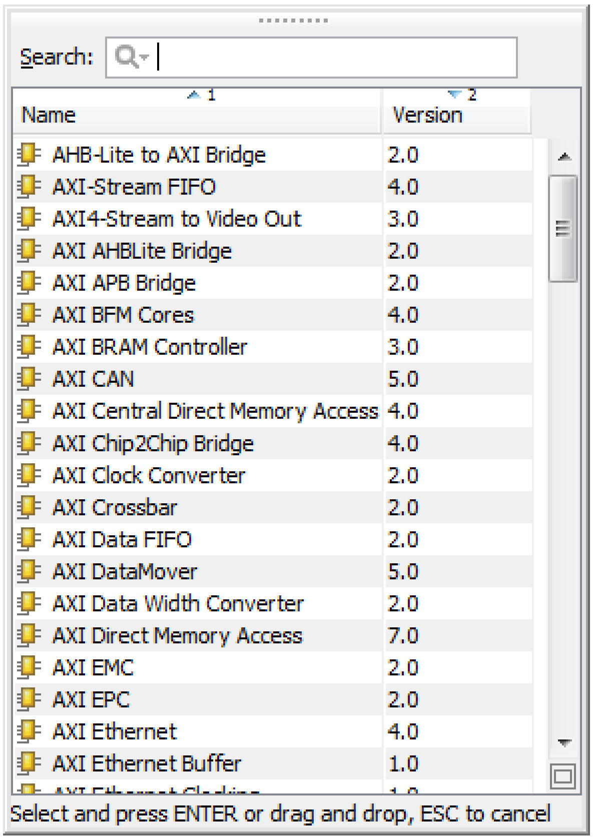
Alternatively, select the ***Add IP*** option from the information message at the top of the canvas, shown in Figure 2.7.



This design is empty. To get started, Add IP from the catalog.

Figure 2.7: Add IP option in IP Integrator canvas information message

The pop-up IP Catalog window will open, as in Figure 2.8.



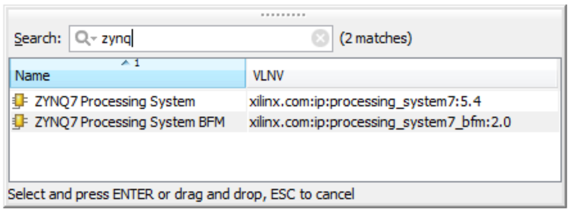
Search:

Name	Version
AHB-Lite to AXI Bridge	2.0
AXI-Stream FIFO	4.0
AXI4-Stream to Video Out	3.0
AXI AHBLite Bridge	2.0
AXI APB Bridge	2.0
AXI BFM Cores	4.0
AXI BRAM Controller	3.0
AXI CAN	5.0
AXI Central Direct Memory Access	4.0
AXI Chip2Chip Bridge	4.0
AXI Clock Converter	2.0
AXI Crossbar	2.0
AXI Data FIFO	2.0
AXI DataMover	5.0
AXI Data Width Converter	2.0
AXI Direct Memory Access	7.0
AXI EMC	2.0
AXI EPC	2.0
AXI Ethernet	4.0
AXI Ethernet Buffer	1.0

Select and press ENTER or drag and drop, ESC to cancel

Figure 2.8: Pop-up IP Catalog window

(d) Enter ***zynq*** in the search field and select the ***ZYNQ7 Processing System***, ensuring that you select the option for *Version 5.4*, as shown in Figure 2.9, and press the ***Enter*** key on your keyboard.



Search: zynq (2 matches)

Name	VLNV
ZYNQ7 Processing System	xilinx.com:ip:processing_system7:5.4
ZYNQ7 Processing System BFM	xilinx.com:ip:processing_system7_bfm:2.0

Select and press ENTER or drag and drop, ESC to cancel

Figure 2.9: Adding ZYNQ7 Processing System from IP Catalog

As in the previous tutorial, the next step is to connect the **DDR** and **FIXED_IO** interface ports on the Zynq PS to the top-level interface ports on the design.

(e) Select the ***Run Block Automation*** option from the *Designer Assistance* message at the top of the Diagram window. Select ***OK***, ensuring that the option to ***Apply Board Preset*** is selected, to generate the external connections for both the **DDR** and **FIXED_IO** interfaces, and apply the relevant board presets.

Your block diagram should now resemble Figure 2.10.

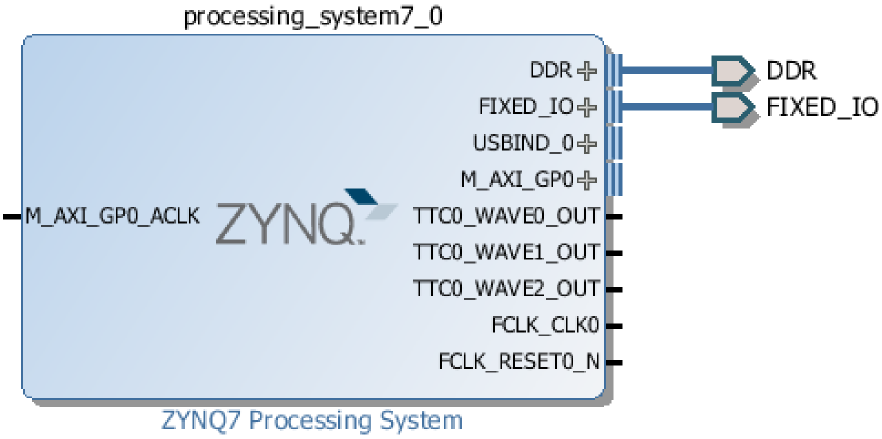


Figure 2.10: ZYNQ7 Processing System external connections

Now that the main Zynq PS has been added to our design and configured, we can now add further blocks which will be placed in the PL to add functionality to the system. In this case we require an **AXI GPIO** block for the **LEDs** and another for the **push buttons**.

(f) Right-click in an empty area of the *Diagram* window and select ***Add IP***. Enter ***GPIO*** in the search field and add an instance of the ***AXI GPIO*** IP. Repeat this procedure to add a second **AXI GPIO** block to the design.

We will now use the *IP Integrator Designer Assistance* tool to automate the connection of the **AXI GPIO** blocks to the **ZYNQ7 Processing System**.

(g) Click ***Run Connection Automation*** from the *Designer Assistance* message at the top of the *Diagram* window and select ***/axi_gpio_0/S_AXI***, as shown Figure 2.11.

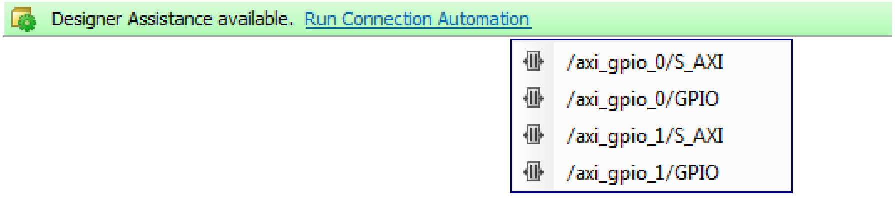
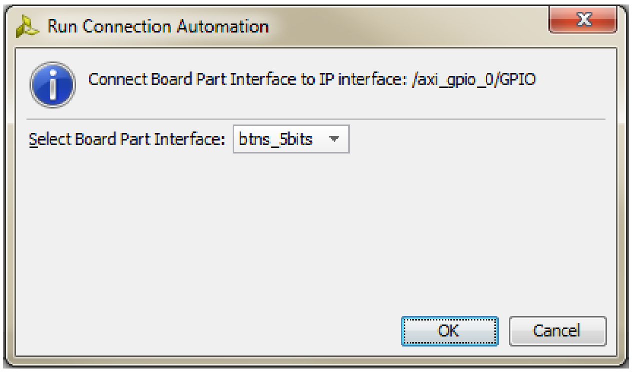


Figure 2.11: Run Block Automation - GPIOinstance 1

Click ***OK*** to ensure automatic clock connection, which adds the **Processor System Reset Module** and the **AXI Interconnect** blocks.

(h) Click ***Run Connection Automation*** from the *Designer Automation* message at the top of the *Diagram* window and select ***/axi_gpio_0/GPIO***.

The Run Connection Automation dialogue will open, as in Figure 2.12. Select ***btns_5bits*** from the drop-down menu, and click ***OK***.


Run Connection Automation

Connect Board Part Interface to IP interface: /axi_gpio_0/GPIO

Select Board Part Interface: btns_5bits

Figure 2.12: Run Connection Automation dialogue — GPIO

(i) Repeat steps **(g)** and **(h)** for the second **GPIO** block, this time selecting ***leds_8bits*** for ***/axi_gpio_1/GPIO***.

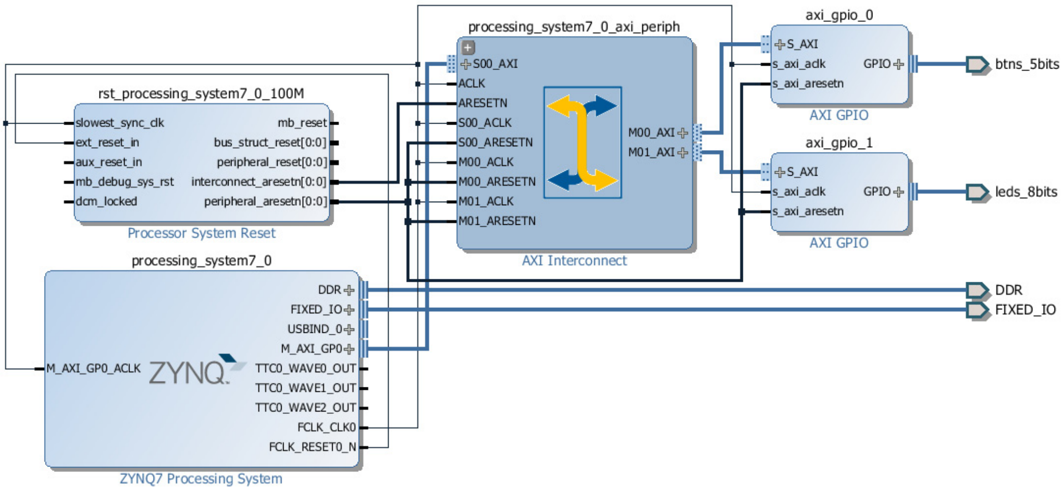
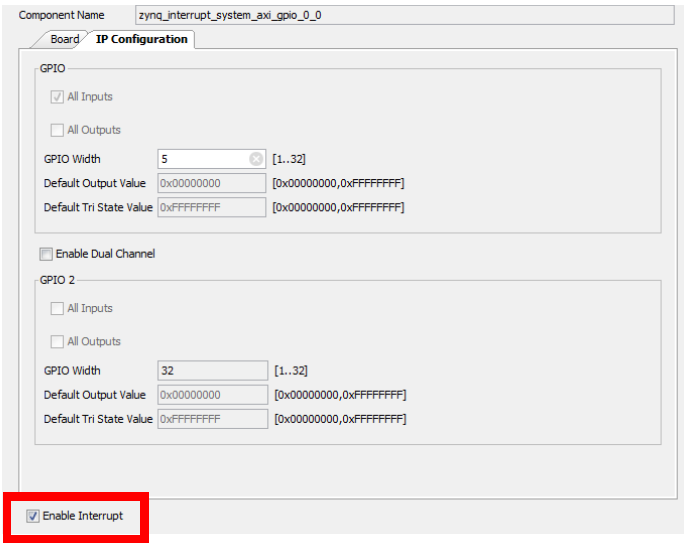


Figure 2.13: Zynq processor system

You will now have a system that is similar to Figure 2.13. We now need to configure the system to utilise hardware interrupts from the **push buttons** to trigger functions in the **Zynq PS**.

(j) Double--click on the **GPIO** block connected to the **push buttons**, ***axi_gpio_0***, to open the *Re-customize IP* window,.



Component Name: zynq_interrupt_system_axi_gpio_0_0

Board | **IP Configuration**

GPIO

☑ All Inputs

□ All Outputs

GPIO Width: 5 [1..32]

Default Output Value: 0x00000000 [0x00000000,0xFFFFFFFF]

Default Tri State Value: 0xFFFFFFFF [0x00000000,0xFFFFFFFF]

□ Enable Dual Channel

GPIO 2

□ All Inputs

□ All Outputs

GPIO Width: 32 [1..32]

Default Output Value: 0x00000000 [0x00000000,0xFFFFFFFF]

Default Tri State Value: 0xFFFFFFFF [0x00000000,0xFFFFFFFF]

☑ Enable Interrupt

Figure 2.14: Enabling GPIO interrupts

Click the **IP Configuration** tab and enable interrupts from the **push buttons** by clicking in the box highlighted in Figure 2.14 and click ***OK.*** This will add an additional output port for the interrupt request to the **GPIO** block as in Figure 2.15.

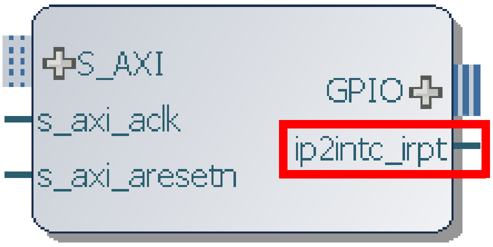
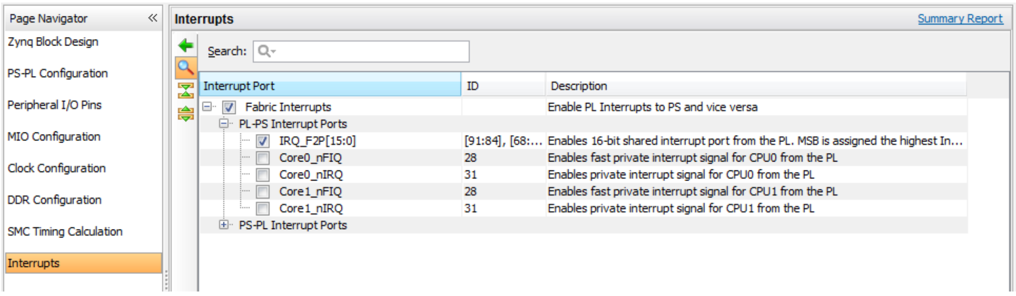


Figure 2.15: GPIO block with interrupt port

Now we must configure the **Zynq PS** to accept interrupt requests.

(k) Double-click on the **Zynq PS** block, ***processing_system7_0***, to open the *Re-Customize IP* window.

(l) Select *Interrupts* from the *Page Navigator* on the left-hand side and expand the menu on the right as in Figure 2.16. Since we want to allow interrupts from the **programmable logic** to the **processing system,** tick the box to enable **Fabric Interrupts**, then click to enable the shared interrupt port as in Figure 2.16. This means interrupts from the **PL** can be connected to the interrupt controller within the **Zynq PS**. Click ***OK.***



Interrupt Port	ID	Description
☑ Fabric Interrupts		Enable PL Interrupts to PS and vice versa
PL-PS Interrupt Ports		
☑ IRQ_F2P[15:0]	[91:84], [68:...	Enables 16-bit shared interrupt port from the PL. MSB is assigned the highest In...
□ Core0_nFIQ	28	Enables fast private interrupt signal for CPU0 from the PL
□ Core0_nIRQ	31	Enables private interrupt signal for CPU0 from the PL
□ Core1_nFIQ	28	Enables fast private interrupt signal for CPU1 from the PL
□ Core1_nIRQ	31	Enables private interrupt signal for CPU1 from the PL
PS-PL Interrupt Ports		

Figure 2.16: Configuring Zynq PS to utilise interrupts

(m) Make a connection between the interrupt request of the **GPIO** block and the newly created interrupt port of the **Zynq PS**, highlighted in Figure 2.17.

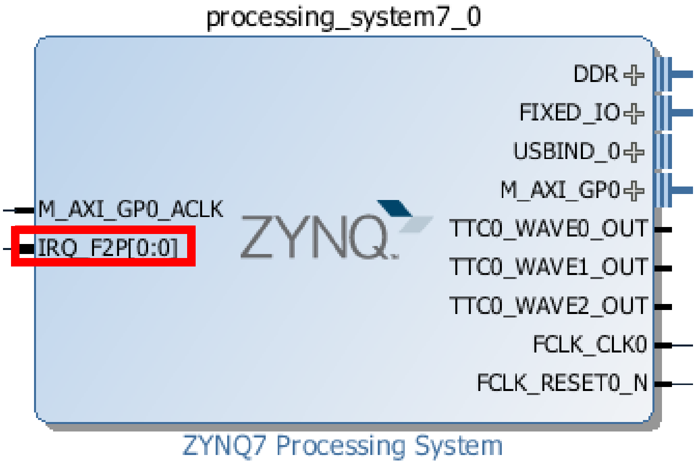


Figure 2.17: Zynq PS with interrupt port

Your final design should resemble Figure 2.18, although the positioning of your blocks may be different.

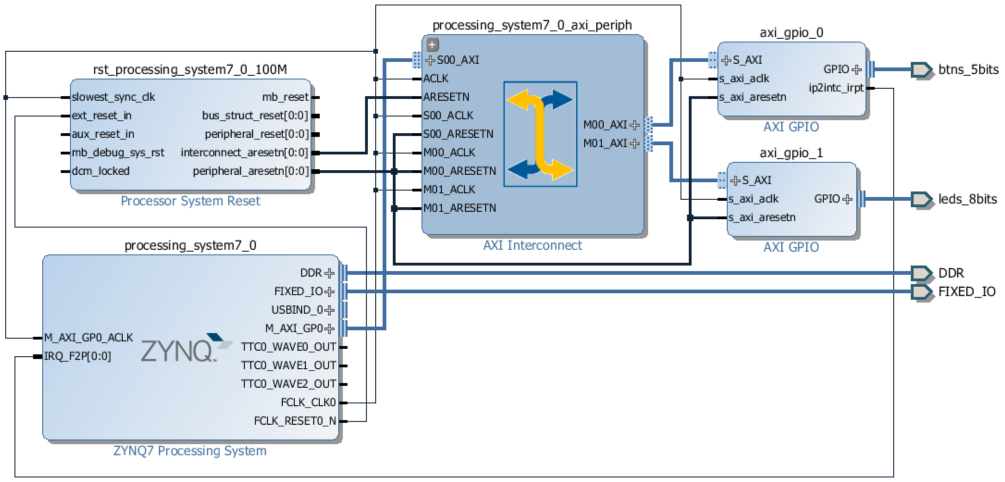


Figure 2.18: Zynq processor system with interrupts

(n) Save your design by selecting ***File > Save Block Design*** from the *Menu Bar.*

(o) Validate the design by selecting ***Tools > Validate Design*** from the *Menu Bar*. This will run a Design-Rule-Check (DRC).

Alternatively, select the Validate Design button, ✓, from the Main Toolbar.

(p) A *Validate Design* dialogue should appear to confirm that validation of the design was successful. Click ***OK,*** to dismiss the message.

With the design successfully validated, we can now move on to generating the HDL design files for the system. The procedure here is identical to the previous tutorial, ***First Designs on Zynq***.

(q) In the **Sources** window of the *Data Windows* pane, select the ***Sources*** tab.

(r) Right-click on the top-level system design, which in this case is **zynq_interrupt_system**, and select ***Create HDL Wrapper***.

The *Create HDL Wrapper* dialogue window will open. Accept the default option specifying that VIvado should manage the wrapper and click ***OK***.

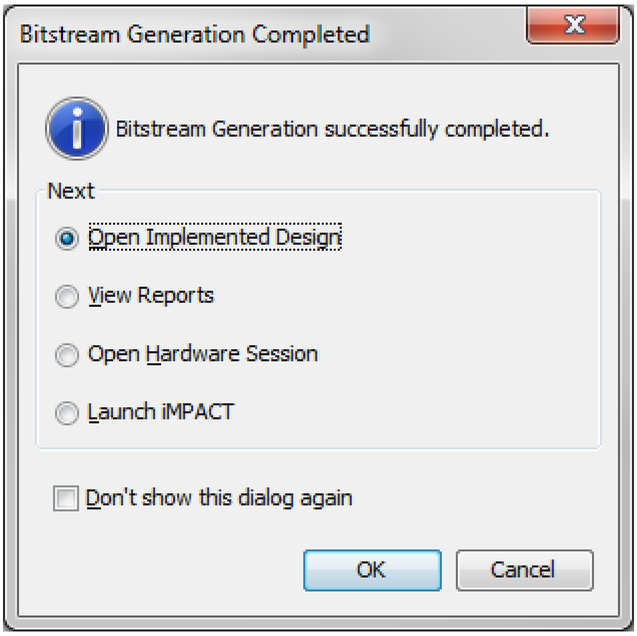
With all HDL design files generated, the next step in Vivado is to implement our design and generate a bitstream file.

(s) In *Flow Navigator*, click ***Generate Bitstream*** from the *Program and Debug* section.

If a dialogue window appears prompting you to save your design, click ***Save***.

The combination of running the synthesis, implementation and bitstream generation processes back-to-back may take a few minutes, depending on the power of your computer system.

(t) Once the bitstream generation is complete a dialogue window will open to inform you that the process has been completed successfully, as in Figure 2.19.



Bitstream Generation Completed

Bitstream Generation successfully completed.

Next

● Open Implemented Design

○ View Reports

○ Open Hardware Session

○ Launch iMPACT

□ Don't show this dialog again

Figure 2.19: Bitstream Generation completion dialogue window

Select ***Open Implemented Design***, and click ***OK***.

At this point you will be presented with the *Device* view, where you can see the PL resources which are utilised by the design.

With the bitstream generation complete, the final step in Vivado is to export the design to the SDK, where we will create the software application that will allow the Zynq PS to control the LEDs on the ZedBoard.

(u) Select ***File* > *Export* > *Export Hardware for SDK...*** from the *Menu Bar*.

(v) The *Export Hardware for SDK* dialogue window will open. Ensure that the options to ***Include bitstream*** and ***Launch SDK*** are selected, and Click ***OK***.

This concludes the steps that are required in Vivado IDE. All hardware components of the system have been configured and generated. In the next exercise we will create the software application that utilises this hardware system.

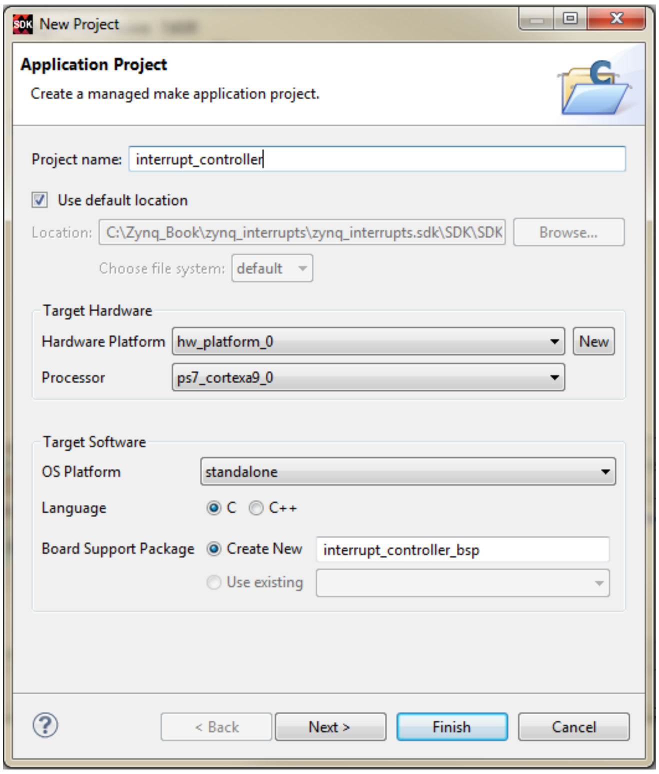
Exercise 2C Creating a Software Application in the SDK

In this exercise a software application will be created that utilises hardware interrupts on the Zedboard. The push buttons will be used to increment a counter by different values, and the count will be continuously displayed on the LEDs in binary form, where LED0 corresponds to the least significant bit (LSB) and LED7 the most significant bit (MSB). This application will run on the Zynq processing systems, communicating with the AXI GPIO blocks implemented in the PL.

The SDK should have opened after the conclusion of Exercise 2B. If it did not open, you can open the SDK by navigating to **Start > All Programs > Xilinx Design Tools >Vivado 2013.4>SDK> Xilinx SDK 2013.4** and specifying the workspace as in Exercise 2A.

(a) Select **File > New > Application Project** from the *Menu bar*.

(b) The *New Project* dialogue window will open. Enter ***interrupt_counter*** in the *Project name* field, as shown in Figure 2.20, keeping all other options with the default settings. Click ***Next***.



New Project

Application Project

Create a managed make application project.

Project name: interrupt_controller

☑ Use default location

Location: C:\Zynq_Book\zynq_interrupts\zynq_interrupts.sdk\SDK\SDK Browse...

Choose file system: default

Target Hardware

Hardware Platform: hw_platform_0 New

Processor: ps7_cortexa9_0

Target Software

OS Platform: standalone

Language: ● C ○ C++

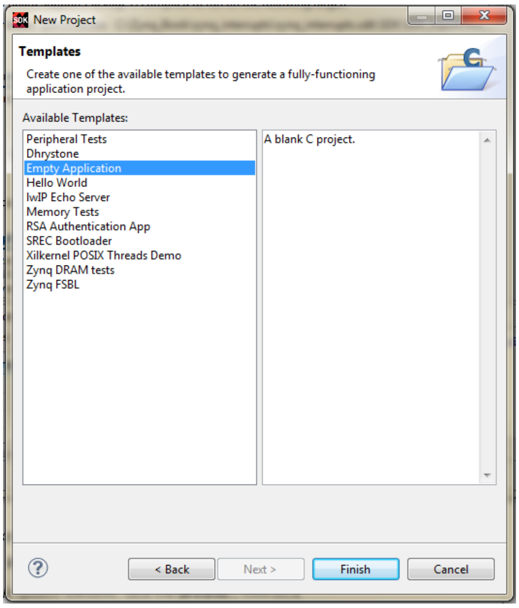
Board Support Package: ● Create New interrupt_controller_bsp

○ Use existing

< Back | Next > | Finish | Cancel

Figure 2.20: New Application Project dialogue

(c) At the *New Project Templates* screen, select ***Empty Application***, as in Figure 2.21, and click ***Finish*** to create the project.



New Project

Templates

Create one of the available templates to generate a fully-functioning application project.

Available Templates:

- Peripheral Tests
- Dhrystone
- Empty Application
- Hello World
- lwIP Echo Server
- Memory Tests
- RSA Authentication App
- SREC Bootloader
- Xilkernel POSIX Threads Demo
- Zynq DRAM tests
- Zynq FSBL

A blank C project.

Figure 2.21: New Project Template dialogue

NOTE: the new project should open automatically. If it doesn't, you may need to close the Welcome tab in order to view the project.

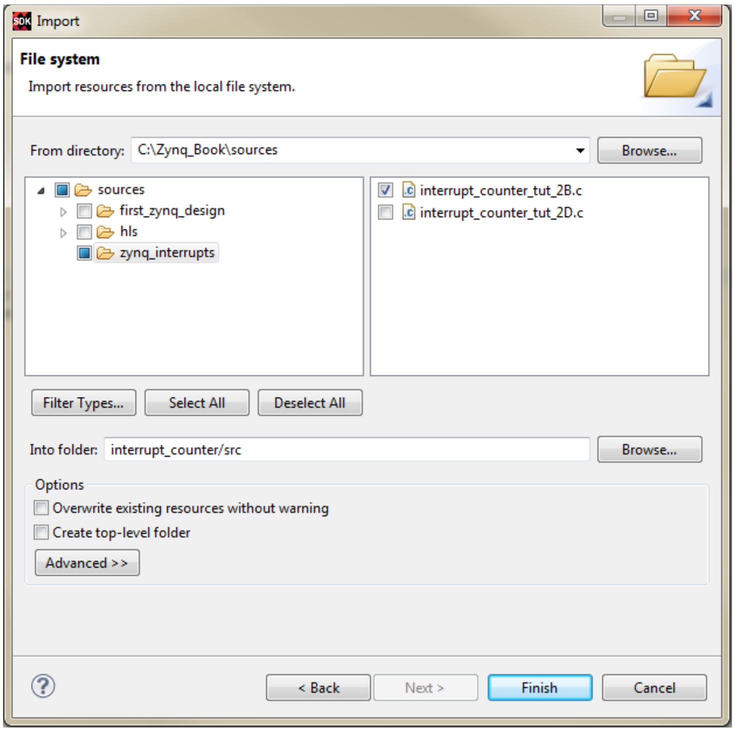
With the new Application Project created, we can now import some pre-prepared source code for the application.

(d) In the *Project Explorer* panel, expand ***interrupt_counter*** and highlight the *src* directory. Right-click and select ***Import...***, choosing ***General*** > ***File System*** as an import source.

(e) In the *Import File System* window, click the ***Browse...*** button.

(f) Navigate to the directory: **C:\Zynq_Book\sources\zynq_interrupts** and click ***OK.***

(g) Select the file **interrupt_counter_tut_2B.c**, as shown in Figure 2.22, and click ***Finish***.



From directory: C:\Zynq_Book\sources

- ▪ sources
  - □ first_zynq_design
  - □ hls
  - ▪ zynq_interrupts

☑ interrupt_counter_tut_2B.c
□ interrupt_counter_tut_2D.c

Into folder: interrupt_counter/src

Options
□ Overwrite existing resources without warning
□ Create top-level folder

Figure 2.22: Import C source file

This file contains C Code that has been written to perform the interrupt triggered counter operation on the ZedBoard.

(h) Open the imported source file by expanding the *src* folder and double-clicking on **interrupt_counter_tut_2B.c**, and explore the code.

The code has been fully commented, but will be briefly discussed here for clarity. Note that this file contains several portions of code which have been commented out; these will be utilised and discussed further in the next exercise and can be ignored for now.

By now, you should be familiar with the use of drivers and parameters in configuring and operating the GPIO. Remember, detailed information of these drivers can be found in the **system.mss** file, explaining the purpose of each function and the parameters passed to it. Predesignated parameters can also be found in `xparameters.h`.

The Zynq PS features a built in interrupt controller, initialised here as `XScuGic INTCInst`. This handles all incoming interrupt requests passed to the PS and performs the function associated with each interrupt source. It is also capable of prioritising multiple interrupt sources to the requirements of the application.

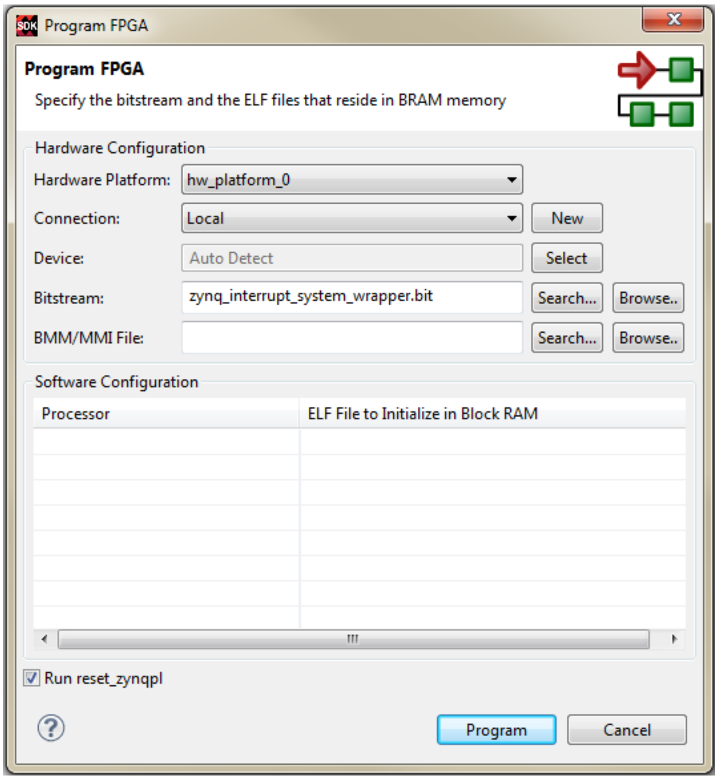
Of particular note is the inclusion of the function `BTN_Intr_Handler(void *InstancePtr);`. This is the interrupt handler function for the push buttons and is called every time an interrupt request from the push buttons in the PL is received in the PS. This performs a counter increment on each call and displays the value of the counter on the LEDs in binary.

An initial setup function can be found below the main function. This is `InterruptSystemSetup(XScuGic *XScuGicInstancePtr);`. The function initialises and configures the interrupt controller in the Zynq PS, connecting the interrupt handler to the interrupt source. It also makes a call to the latter function which enables the interrupt sources and registers exceptions.

The next step is to program the Zynq PL with the bitstream file that we generated in Exercise 2B.

Ensure that the ZedBoard is powered on and that the JTAG port is connected to the PC via the provided USB-A to USB-B cable.

(i) Download the bitstream to the Zynq PL by selecting **Xilinx Tools > Program FPGA** from the *Menu bar*. The *Program FPGA* window will appear. The Bitstream field should already be populated with the correct bitstream file, as in Figure 2.23.



Program FPGA

Program FPGA

Specify the bitstream and the ELF files that reside in BRAM memory

Hardware Configuration

Hardware Platform: hw_platform_0

Connection: Local

Device: Auto Detect

Bitstream: zynq_interrupt_system_wrapper.bit

BMM/MMI File:

Software Configuration

Processor	ELF File to Initialize in Block RAM

☑ Run reset_zynqpl

Figure 2.23: Program FPGA dialogue window

If it is not, enter:

zynq_interrupt_system_wrapper.bit

and click ***Program***.

As in the previous tutorial, once the device has successfully been programmed, the *DONE LED* on the ZedBoard will turn blue.

With the Zynq PL successfully configured with the bitstream file, we can now launch our software application on the Zynq PS.

(j) Select ***interrupt_counter*** in *Project Explorer*. Right-click and select **Run As > Launch on Hardware**.

The counter increments by different values based on the push button which is pressed. The counter operates as demonstrated in Figure 2.24.

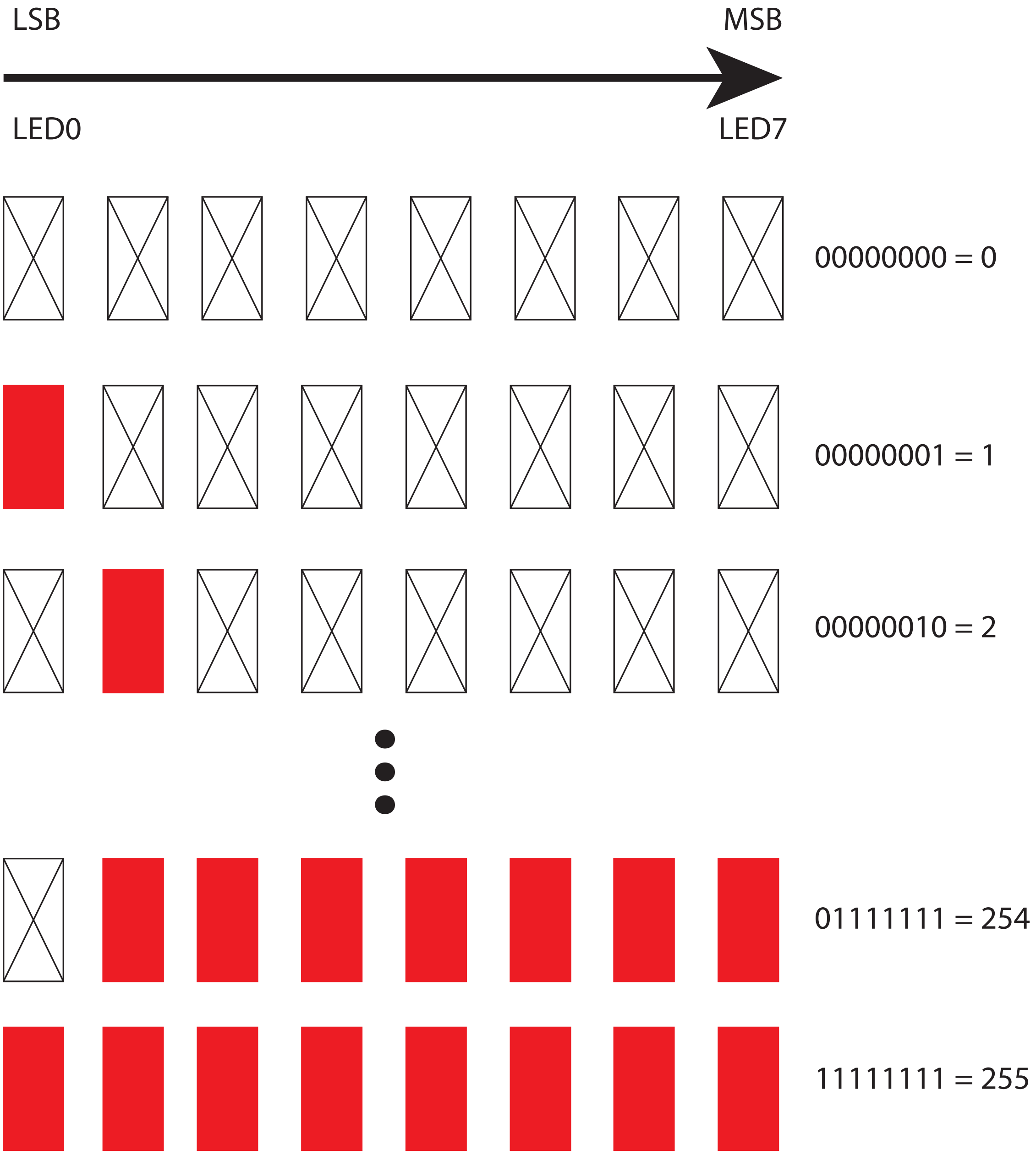


Figure 2.24: LED flashing states

(k) Try pressing different push buttons and observing how the counter increments (or does it increment at all?) Based on your findings, can you determine the value assigned to each of the push buttons (BTNU, BTND, BTNL, BTNR and BTNC as noted on the ZedBoard)?

You have successfully created and executed a software application utilising interrupts on the Zynq PS. The next step is to go back and add an additional interrupt source with higher priority to alter the functionality of the system.

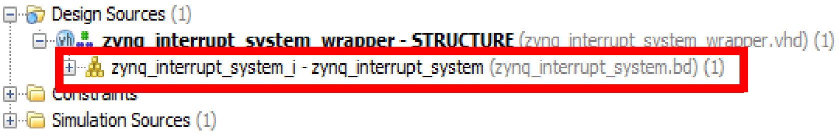
Exercise 2D Adding a Further Interrupt Source

In this exercise we will add an additional source of interrupt to the project created in Exercise 2B in the form of an **AXI Timer**.

(a) Launch Vivado by double-clicking on the Vivado desktop icon: , or by navigating to **Start > All Programs > Xilinx Design Tools > Vivado 2014.1 > Vivado 2014.1**

(b) When the program launches, open the previously created project by selecting ***Open Project***. The previously created project should appear in the list of recent projects as ***C:/Zynq_Book/zynq_interrupts/zynq_interrupts.xpr*** so click on it. If it doesn't, click ***Browse Projects...*** and navigate to that directory, selecting ***zynq_interrupts.xpr*** and clicking open.

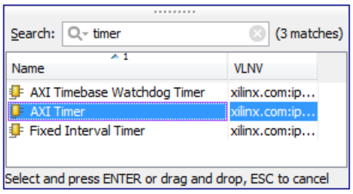
(c) Open the block design from the sources panel by expanding the sources and double clicking on the block design as highlighted in Figure 2.25.



```
Design Sources (1)
  zynq_interrupt_system_wrapper - STRUCTURE (zynq_interrupt_system_wrapper.vhd) (1)
    zynq_interrupt_system_i - zynq_interrupt_system (zynq_interrupt_system.bd) (1)
Constraints
Simulation Sources (1)
```

Figure 2.25: Opening an existing block diagram

(d) With the block diagram now open we will add an **AXI Timer** to the design. In the *Vivado IP Integrator Diagram* canvas, right-click anywhere and select ***Add IP***. Enter ***timer*** in the search field and add the IP ***AXI TIMER*** to the design by either dragging it onto the canvas or selecting it and pressing ENTER.



Search: timer (3 matches)

Name	VLNV
AXI Timebase Watchdog Timer	xilinx.com:ip...
AXI Timer	xilinx.com:ip...
Fixed Interval Timer	xilinx.com:ip...

Select and press ENTER or drag and drop, ESC to cancel

Figure 2.26: AXI Timer in the IP Catalog

(e) Select ***Run Connection Automation*** option from the *Designer Assistance* message at the top of the Diagram window and select**/axi_timer_0/S_AXI** and click ***OK*** to connect the timer to the AXI Interconnect.

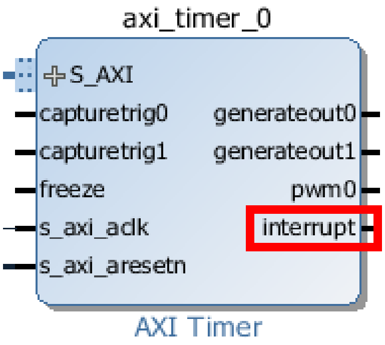


Figure 2.27: AXI Timer in the block design

(f) Note that in Figure 2.27 the AXI Timer features an interrupt request, which requires connection to the Zynq PS. However, we already have an interrupt connected to the input of the PS. This input is a shared interrupt port, and so accepts multiple interrupts via one signal. We therefore require an additional IP block to concatenate these two interrupt requests into one signal. In the canvas, right-click anywhere and select ***Add IP***. Enter ***concat*** in the search field and add the IP ***Concat*** to the design.

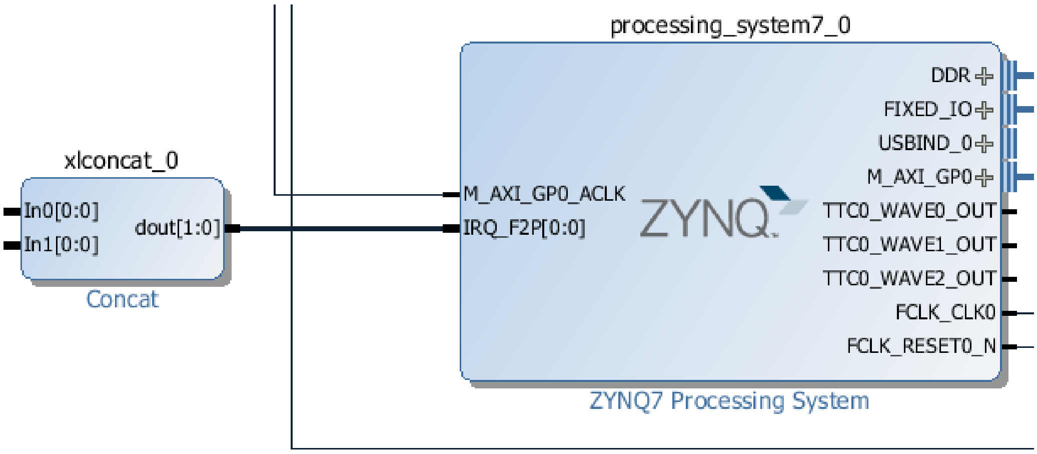


Figure 2.28: Concat in the block design

(g) Remove the connection to **IRQ_F2P[0:0]** on the Zynq PS by clicking it and pressing DELETE. Connect the output from the ***Concat*** block, ***xlconcat_0*** to this instead. Then, connect the interrupt request from the GPIO to **In0[0:0]** and the interrupt from the timer to **In1[0:0]**, creating a shared interrupt signal that is passed to the PS. Your block diagram should be similar to Figure 2.29.

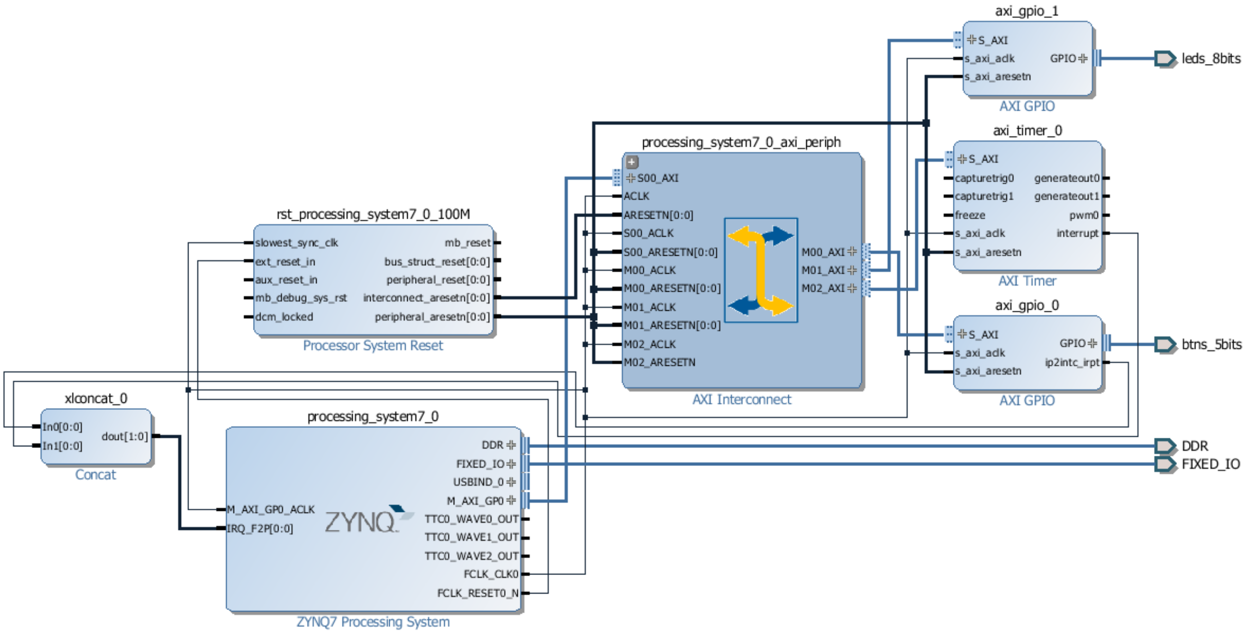


Figure 2.29: Complete system with multiple interrupt sources

We now need to regenerate the output products, update the HDL wrapped and generate a new bitstream for our altered design.

(h) Right-click on the top-level system design and select ***Create HDL Wrapper...*** selecting the default option as previous. Click ***OK***.

(i) In *Flow Navigator*, click ***Generate Bitstream*** from the *Program and Debug* section.

If a dialogue window appears prompting you to save your design, click ***Save***.

(j) A dialogue window will open requesting that you launch synthesis and implementation before starting the *Generate Bitstream* process. Click ***Yes*** to accept.

Again these back-to-back processes may take a few minutes, depending on the power of your computer system.

(k) When this process is completed click ***OK***.

(l) Select ***File*** > ***Export*** > ***Export Hardware for SDK...*** from the *Menu Bar*.

(m) The *Export Hardware for SDK* dialogue window will open. Ensure that the options to ***Include bitstream*** and ***Launch SDK*** are selected, and Click ***OK***. A dialog will be presented asking if you wish to overwrite an exported file, which is the initial system featuring a single interrupt. Select ***Yes*** for this and any further prompts.

(n) Once the SDK opens and builds the project, we will alter our application to make use of the new interrupt source. Right-click on the project **interrupt_counter** in the ***Project Explorer*** and select ***Delete***.

Repeat for the BSP, **interrupt_controller_bsp**.

Repeat the steps outlined in Exercise 2B (a) to (h) for creating a new application project, BSP and importing a source file, this time selecting **interrupt_counter_tut_2D.c**.

Notice the inclusion of a second interrupt handler, `TMR_Intr_Handler(void *data);` which will increment the value of the counter after the timer has expired three times, writing the new value to the LEDs.

Additional code has been included in the main to configure and start the timer, and full details of these functions can be found in the **system.mms**. The function `IntcInitFunction(u16 DeviceId, XTmrCtr *TmrInstancePtr,XGpio *GpioInstancePtr);` also contains additional code to connect the timer interrupt to the handler and enable it.

In brief, the timer is loaded with a value `TMR_LOAD` and configured to automatically reload on each expiration. The interrupt handler keeps track of the number of expirations and after three expirations performs the required steps, otherwise it simply increments the variable storing the number of expirations.

Save the file.

(o) Download the bitstream to the Zynq PL by selecting **Xilinx Tools > Program FPGA** from the *Menu bar*.

(p) Once the blue LED signalling successful programming lights, select ***zynq_interrupts*** in *Project Explorer*. Right-click and select **Run As > Launch on Hardware**.

Note that the counter will increment by 1 every time the timer expires three times. The buttons still operate as in the previous exercise.

This completes this tutorial and systems utilising both a single and multiple interrupt sources have been created and tested.

The Zynq Book Tutorial 3

Designing With Vivado High Level Synthesis

v1.2, May 2014

Revision History

Date	Version	Changes
30/10/2013	1.0	First release for Vivado Design Suite version 2013.2
28/01/2014	1.1	Updated for changes in Vivado Design Suite version 2013.4
06/5/2014	1.2	Updated for changes in Vivado Design Suite version 2014.1

Introduction

This tutorial presents an introduction to High Level Synthesis using the Vivado™ HLS environment. The creation of projects manually through the GUI, and automatically through scripting will be covered. The process of simulating, synthesising and analysing a Vivado HLS design will then be explored, with sufficient design optimisation and solution comparison along the way.

The tutorial is split into three exercises, and is organised as follows:

Exercise 3A — This exercise concerns the creation of projects using both the Vivado HLS GUI and use of Tcl scripting. It details the inclusion of relevant source and test files and generation of a project for use in the proceeding exercise.

Exercise 3B — This exercise involves design optimization of a matrix multiplication function through use of various directives. It presents the Vivado HLS design environment and method of synthesis and analysis of project solutions.

Exercise 3C — Finally, a more detailed look at how Vivado HLS synthesises interfaces is investigated.

Exercise 3A Creating Projects in Vivado HLS

In this exercise we will present the creation of Vivado HLS projects using both the Vivado HLS GUI and the use of Tcl scripting to expedite the process.

(a) Before we begin it is necessary to copy the files from **C:\Zynq_Book\sources\hls** to a new directory, **C:\Zynq_Book\hls.**

(b) Launch the Vivado HLS GUI by navigating to **Start** > **All Programs** > **Xilinx Design Tools** > **Vivado 2014.1**> **Vivado HLS** > **Vivado HLS 2014.1**

(c) When the Vivado HLS GUI loads, you will be presented with the *Welcome* screen as in Figure 3.1.

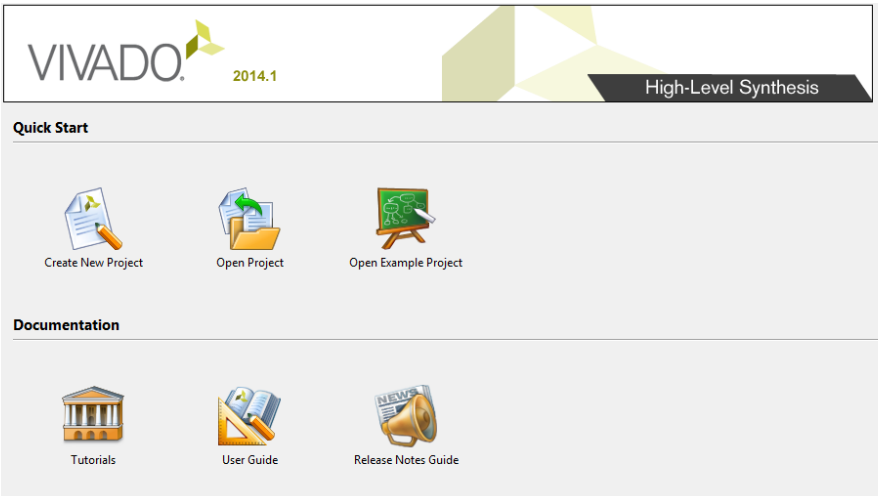


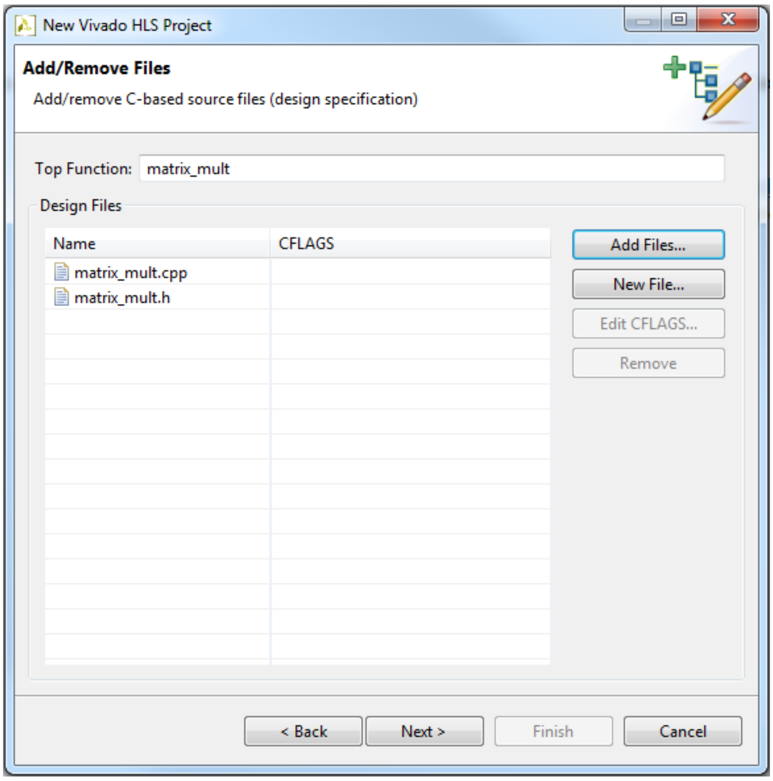
Figure 3.1: Vivado HLS welcome screen

(d) Select the option to ***Create New Project*** in Figure 3.1

(e) At the Project Name dialogue, enter ***matrix_mult_prj*** as the ***Project name*** and ***C:/ Zynq_Book/hls/tut3A*** as ***Project location***.

Click ***Next***.

(f) You will now be prompted to add or remove source files for the project. All C-based source files for this tutorial have been created in advance, as we seek to guide the design flow rather than the programming itself. Click ***Add Files...*** and navigate to **C:\Zynq_Book\hls\tut3A**



New Vivado HLS Project

Add/Remove Files

Add/remove C-based source files (design specification)

Top Function: matrix_mult

Design Files

Name	CFLAGS
matrix_mult.cpp	
matrix_mult.h	

Figure 3.2: Adding files to a Vivado HLS project

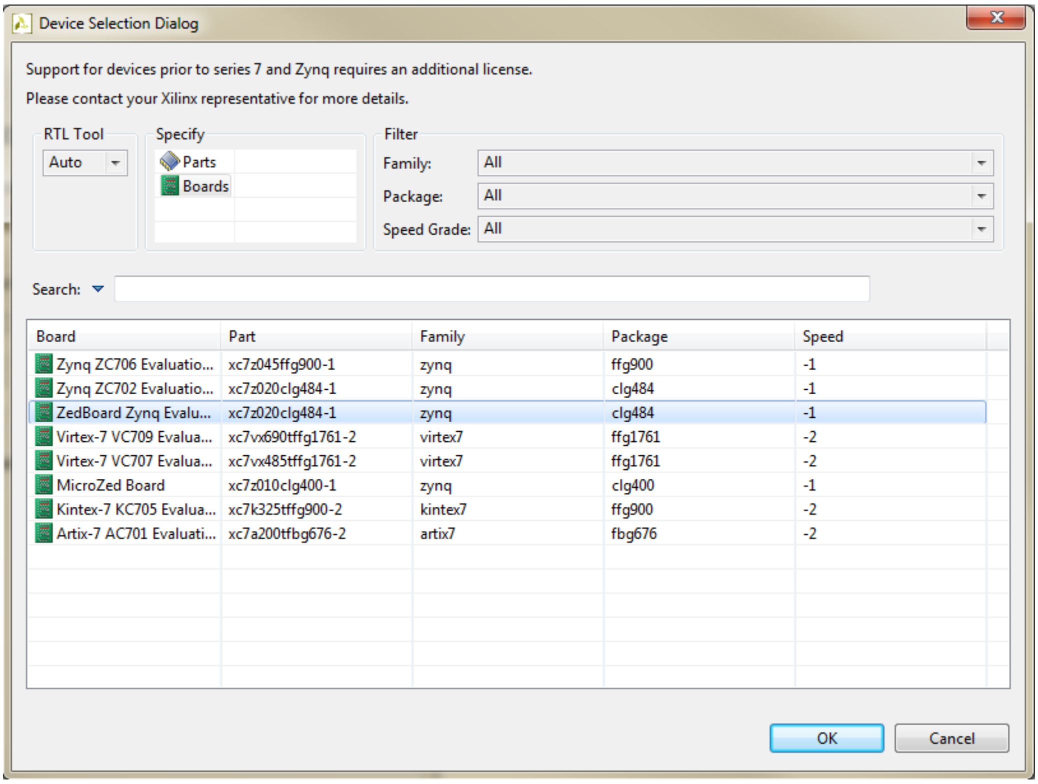
Select the files ***matrix_mult.cpp*** and ***matrix_mult.h*** and click ***Open.*** Set the top function to ***matrix_mult*** as in Figure 3.2.

Click ***Next***.

(g) You will now be prompted to add a testbench file for design testing. Once more, click ***Add Files...*** and navigate to the previous directory this time adding the file ***matrix_mult_test.cpp*** and clicking ***Next***.

(h) The next step is configuring a solution for a specific FPGA technology. In this case, leave the solution name and clock settings as the default options.

Since we are using the ZedBoard with the Zynq-7020 FPGA click **...** in the part selection panel.



Support for devices prior to series 7 and Zynq requires an additional license.
Please contact your Xilinx representative for more details.

RTL Tool: Auto

Specify: Parts, Boards

Filter
Family: All
Package: All
Speed Grade: All

Search:

Board	Part	Family	Package	Speed
Zynq ZC706 Evaluatio...	xc7z045ffg900-1	zynq	ffg900	-1
Zynq ZC702 Evaluatio...	xc7z020clg484-1	zynq	clg484	-1
ZedBoard Zynq Evalu...	xc7z020clg484-1	zynq	clg484	-1
Virtex-7 VC709 Evalua...	xc7vx690tffg1761-2	virtex7	ffg1761	-2
Virtex-7 VC707 Evalua...	xc7vx485tffg1761-2	virtex7	ffg1761	-2
MicroZed Board	xc7z010clg400-1	zynq	clg400	-1
Kintex-7 KC705 Evalua...	xc7k325tffg900-2	kintex7	ffg900	-2
Artix-7 AC701 Evaluati...	xc7a200tfbg676-2	artix7	fbg676	-2

Figure 3.3: The device selection dialogue

Under the ***Specify*** section select *Boards* and then select the *ZedBoard Zynq Evaluation and Development Kit* before clicking ***OK*** as in Figure 3.3.

Click ***Finish***.

(i) The project will be generated and the workspace will open in ***Synthesis*** mode for the generated project and solution as in Figure 3.4.

Expanding the ***Source*** and ***Test Bench*** sections in the ***Explorer*** tab on the left side shows the inclusion of the source and test files from the previous steps. Double clicking on these files opens them in the editor view for examination and editing.

The project consists of a matrix multiplier, which multiplies two matrices *inA* and *inB* to produce the output *prod*. The testbench performs the multiplication of two known matrices and checks the value of *prod* against expected values.

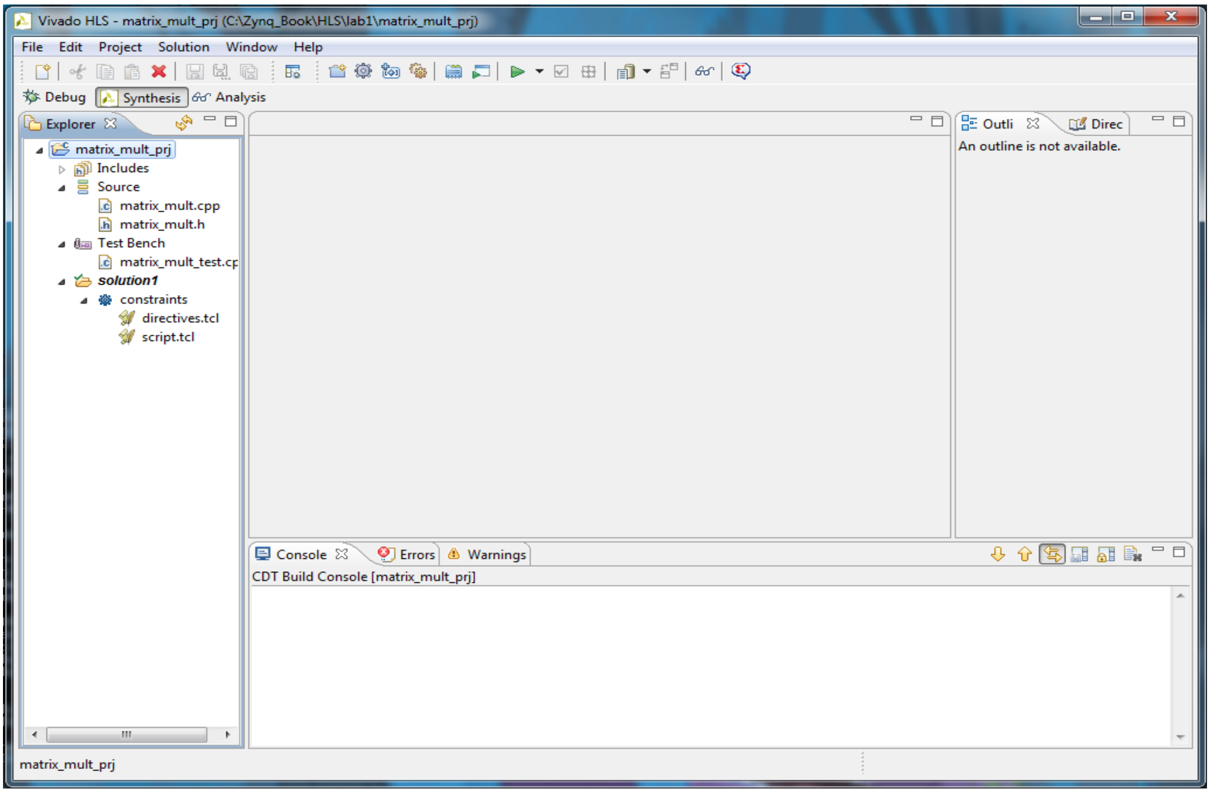
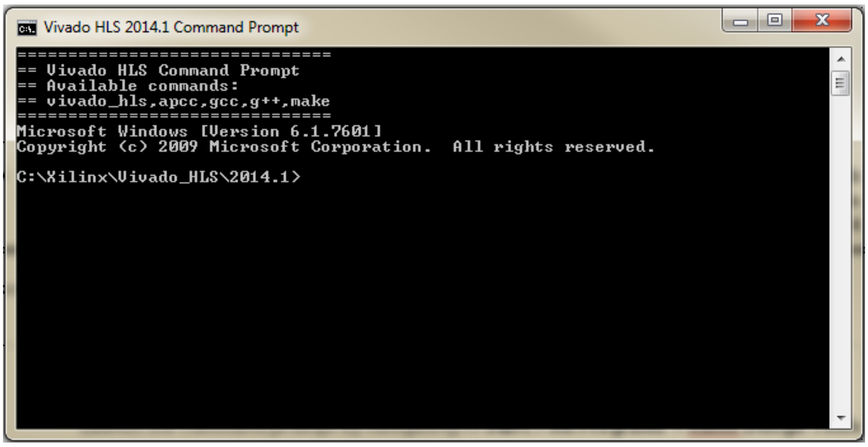


Figure 3.4: Synthesis view in the workspace

While the process of getting to this stage of HLS development is relatively straightforward, it can be quite repetitive and so can be facilitated by use of Tcl scripting. This automates the process of project naming and adding files. As such, we will now demonstrate the creation of the same project using the aforementioned scripting approach.

(j) First, close the Vivado HLS GUI. We will now open the Vivado HLS Command Prompt.

Launch the command prompt by navigating to **Start > All Programs > Xilinx Design Tools > Vivado 2014.1 > Vivado HLS > Vivado HLS 2041.1 Command Prompt**.



```
================================
== Vivado HLS Command Prompt
== Available commands:
== vivado_hls,apcc,gcc,g++,make
================================
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Xilinx\Vivado_HLS\2014.1>
```

Figure 3.5: Vivado HLS command prompt

(k) It is observed that the default directory for commands is the install directory of Vivado HLS, as in Figure 3.5. To change this to the working directory for this tutorial, use the following commands, followed by pressing the ***Enter*** key.

- **cd..** — This is a change directory command which moves up a level in the directory. Repeat this until you have reached the level of the C: drive.
- **cd Zynq_Book** — This changes directory to the Zynq_Book folder.
- **cd HLS** — This changes directory to Zynq_Book/HLS.
- **cd tut3A** — This changes directory to Zynq_Book/HLS/tut3A.

The command prompt should now be in the working directory **C:\Zynq_Book\HLS\tut3A**. This folder contains the source and test files for a project, and also the Tcl script required to build the project, ***run_hls.tcl***.

(l) With the correct working directory and the required files present in that directory, we can now build the project. This is achieved through simply running the Tcl script using the command:

```
vivado_hls -f run_hls.tcl
```

This will begin the process of creating the project and adding source and test bench files. A HLS solution is then created before configuring the project for the target device. Finally a C simulation is run which utilises the test bench to ensure the project operates correctly.

The testbench performs identical multiplications using the HLS hardware solution and software, and compares the results. If these results are identical, a *"Test passed!"* message is displayed.:

(m) To open the project in the VIvado HLS GUI enter the following command:

```
vivado_hls -p matrix_mult_prj
```

And press ***Enter***. This will open the Vivado HLS GUI for the project, which we will utilise in the next exercise.

Using the project generated in the previous exercise, we will now investigate the process of design optimisation in Vivado HLS. This will also provide an insight into the flow from project creation to C synthesis and C/RTL cosimulation. We will also discuss the use of the *Analysis* perspective in analysing a HLS solution.

Exercise 3B Design Optimisation in Vivado HLS

(a) You should already have the GUI open from the previous exercise, but if you don't open the project ***matrix_mult_prj*** in the directory **C:\Zynq_Book\HLS\tut3A** and save in to the **\tut3B** directory using **File > Save As** and selecting the **\tut3B** directory as the location.

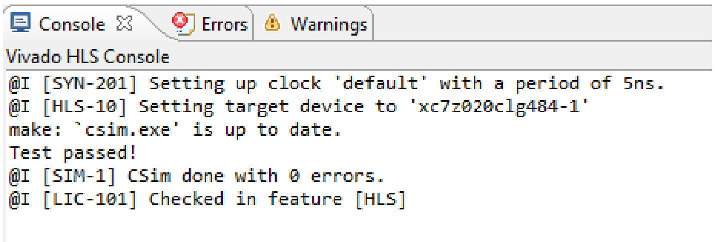
(b) Expand the tabs for ***Source*** and ***Test Bench*** in the ***Explorer*** tab of the *Synthesis* view. As before, this shows that the source and test files have been successfully added to the project. Double clicking on each of these will open them in the editor allowing the code to be inspected and altered as required.

matrix_mult.cpp contains code that performs the multiplication of two matrices through use of iterative loops that run through the rows and columns of the matrices to calculate the product.

matrix_mult.h contains definitions and the prototype function for the matrix multiplication.

matrix_mult_test.cpp is the test bench file which calculates the product of two given matrixes using both the HLS hardware solution and software, comparing to two to ensure successful operation.

(c) Click the ***Run C Simulation*** button in the toolbar to run a C simulation of the solution. Leave the options as default (no boxes checked, no input arguments) and click ***OK***. Upon completion of the simulation, the *"Test passed!"* message will be displayed in the console in the bottom of the screen as in Figure 3.6.

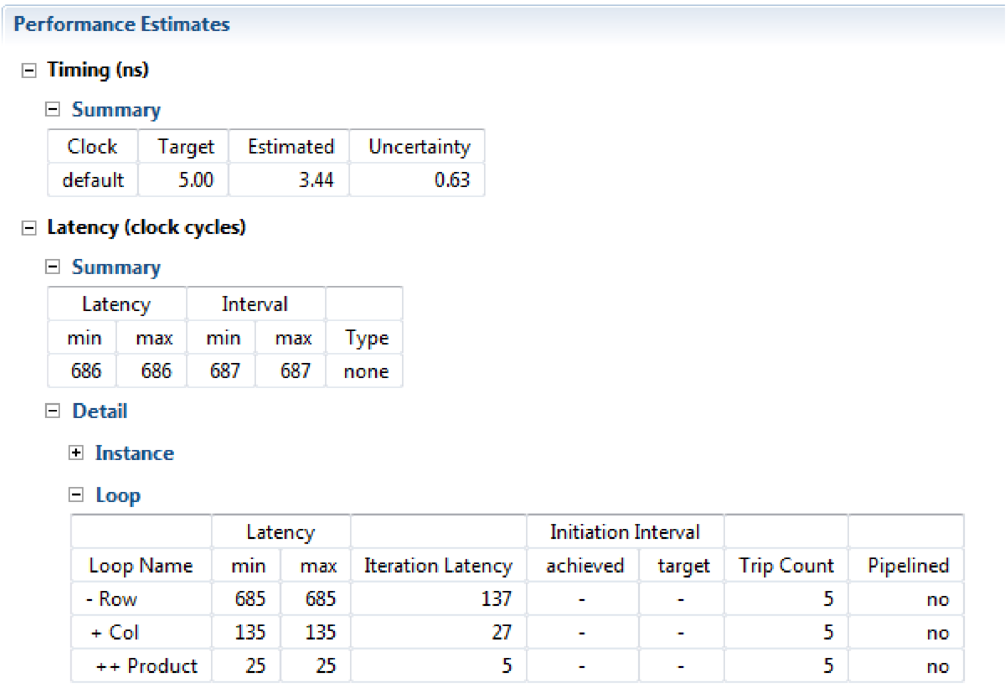


```
@I [SYN-201] Setting up clock 'default' with a period of 5ns.
@I [HLS-10] Setting target device to 'xc7z020clg484-1'
make: `csim.exe' is up to date.
Test passed!
@I [SIM-1] CSim done with 0 errors.
@I [LIC-101] Checked in feature [HLS]
```

Figure 3.6: Vivado HLS console detailing successful testing

(d) The next step is to synthesise the C++ code using HLS. Click the ***C Synthesis*** button in the toolbar. Vivado HLS will begin the process of converting the C++ code into an RTL model with associated VHDL/Verilog/SystemC code. The console details the steps performed in achieving this.

Upon completion, a *Synthesis Report* will open automatically. This details various aspects of the synthesised design, such as information concerning timing and latency and FPGA resource utilisation estimates.



Performance Estimates

⊟ Timing (ns)

⊟ Summary

Clock	Target	Estimated	Uncertainty
default	5.00	3.44	0.63

⊟ Latency (clock cycles)

⊟ Summary

Latency		Interval		
min	max	min	max	Type
686	686	687	687	none

⊟ Detail

⊞ Instance

⊟ Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Row	685	685	137	-	-	5	no
+ Col	135	135	27	-	-	5	no
++ Product	25	25	5	-	-	5	no

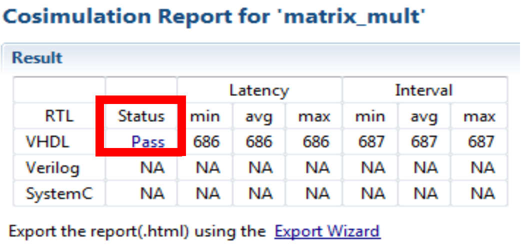
Figure 3.7: Synthesis report for the matrix multiplier, solution1

The synthesised design has an interval of 687clock cycles. Each input array contains 25 elements (as it used 5x5 matrices) and so this suggests roughly 27 clock cycles per input read.

(e) We can now run a C/RTL cosimulation to ensure that the synthesised RTL behaves exactly the same as the C++ code under test.

Click the ***Run C/RTL Cosimulation*** button ☑ . For the RTL selection, ensure VHDL is selected and click ***OK***. Cosimulation will now begin, with the RTL system being generated using VHDL. This process make take a short while to complete but progress can be viewed in the console.

Upon completion, the *Cosimulation Report* will be opened as in Figure 3.8



Cosimulation Report for 'matrix_mult'

Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	Pass	686	686	686	687	687	687
Verilog	NA	NA	NA	NA	NA	NA	NA
SystemC	NA	NA	NA	NA	NA	NA	NA

Export the report(.html) using the Export Wizard

Figure 3.8: Cosimulation report for the matrix multiplier, solution1

Note the *"Pass"* message of Figure 3.8 indicating that the RTL behaves the same as the C++ source code.

(f) Create a new solution for the design by either clicking the ***New Solution*** button in the toolbar or the menu option ***Project*** > ***New Solution***. Click ***Finish*** to accept the defaults for *solution2*.

(g) Double click on *matrix_mult.cpp* in the ***Source*** section of the ***Explorer*** tab to ensure the code is visible in the workspace. We will now insert a directive which will pipeline the nested loops of the matrix multiplication code. This will perform loop flattening, removing the need for loop transitions.

Open the ***Directive*** tab to the right of the workspace. Click on ***Product*** and you will observe the associated portion of code highlighted in the editor, in this instance the multiplication of array elements to produce the product elements of the resulting matrix. Right click on ***Product*** and select *Insert Directive*. This will open the Directives Editor. Use the type drop-down menu to select the option *PIPELINE*. Click ***OK*** to accept the default options. The directives tab should now resemble Figure 3.9.

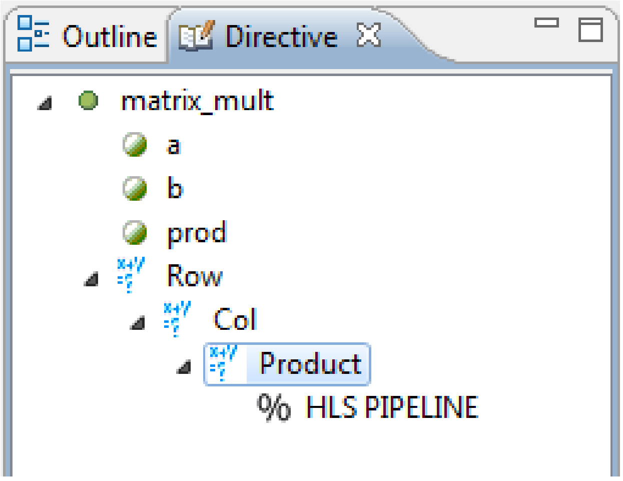
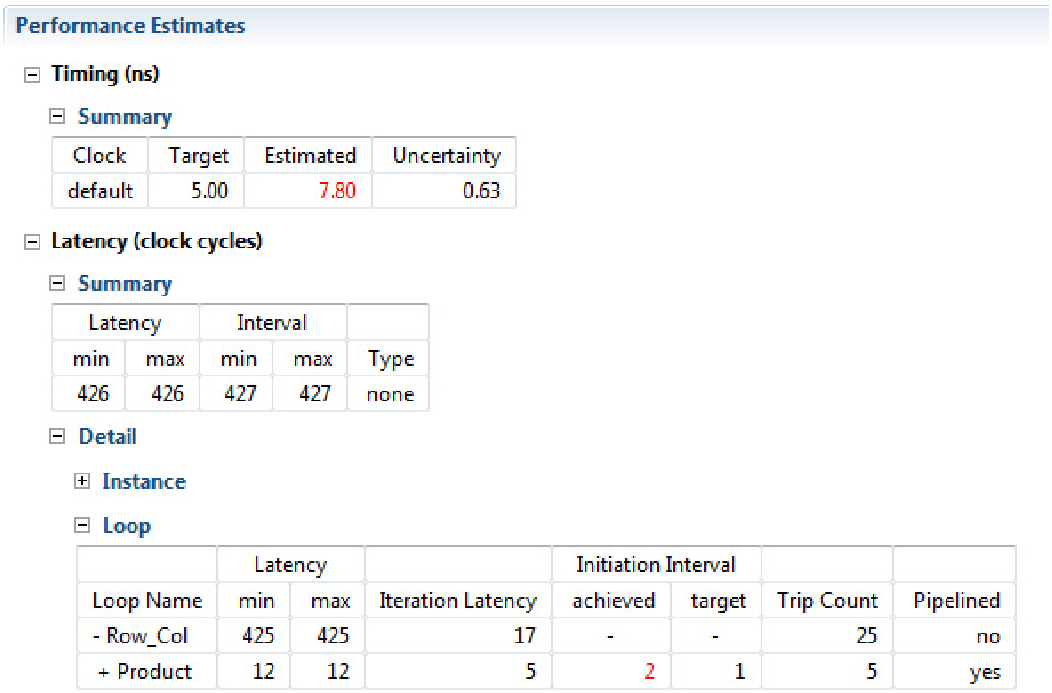


Figure 3.9: Pipelining nested loops in HLS

(h) Click the ***C Synthesis*** button to synthesise the RTL design. The console yields some information about the process of flattening the *Row* loop. It also explains that the default initiation internal (II) target of 1 could not be met for the *Product* loop. This is due to loop dependency.



Performance Estimates

⊟ Timing (ns)

⊟ Summary

Clock	Target	Estimated	Uncertainty
default	5.00	7.80	0.63

⊟ Latency (clock cycles)

⊟ Summary

Latency		Interval		
min	max	min	max	Type
426	426	427	427	none

⊟ Detail

⊞ Instance

⊟ Loop

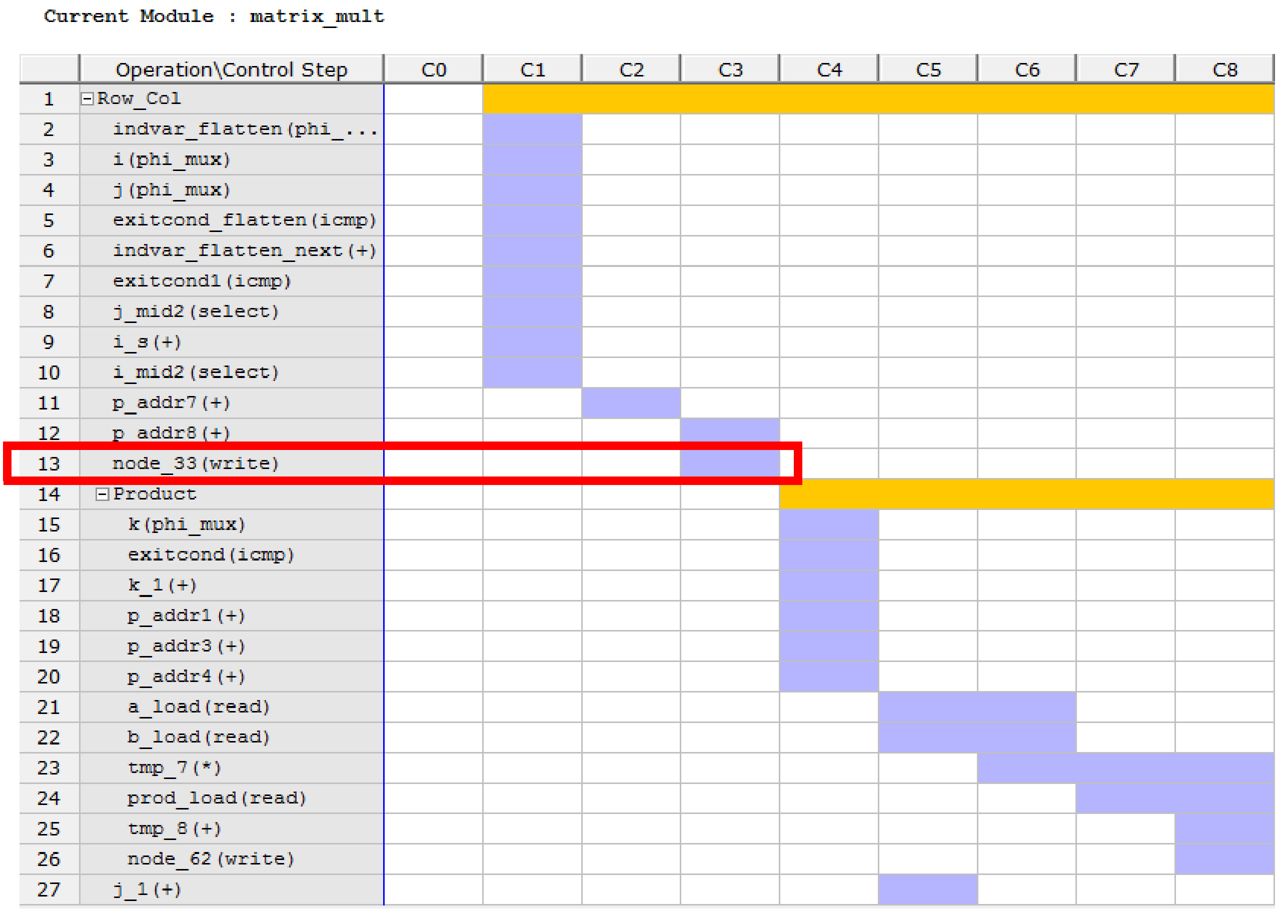
	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Row_Col	425	425	17	-	-	25	no
+ Product	12	12	5	2	1	5	yes

Figure 3.10: Synthesis report for the matrix multiplier, solution2

From the synthesis report shown in Figure 3.10 it is observed that the top level loop, *Row_Col* has not been pipelined as loop *Col* was not flattened. It is also observed an II of 2 was achieved despite the target of 1.

(i) Open the *Analysis* perspective by clicking on Analysis . This will also open the *Performance* view showing how the various operations within the code are scheduled as clock cycles.

(j) Expand the loops *Row_Col* and *Product* by clicking on them to obtain the view shown in Figure 3.11.



Current Module : matrix_mult

	Operation\Control Step	C0	C1	C2	C3	C4	C5	C6	C7	C8
1	⊟Row_Col		🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
2	indvar_flatten(phi_...		🟦							
3	i(phi_mux)		🟦							
4	j(phi_mux)		🟦							
5	exitcond_flatten(icmp)		🟦							
6	indvar_flatten_next(+)		🟦							
7	exitcond1(icmp)		🟦							
8	j_mid2(select)		🟦							
9	i_s(+)		🟦							
10	i_mid2(select)		🟦							
11	p_addr7(+)			🟦						
12	p_addr8(+)				🟦					
13	node_33(write)				🟦					
14	⊟Product					🟧	🟧	🟧	🟧	🟧
15	k(phi_mux)					🟦				
16	exitcond(icmp)					🟦				
17	k_1(+)					🟦				
18	p_addr1(+)					🟦				
19	p_addr3(+)					🟦				
20	p_addr4(+)					🟦				
21	a_load(read)						🟦	🟦		
22	b_load(read)						🟦	🟦		
23	tmp_7(*)							🟦	🟦	🟦
24	prod_load(read)								🟦	🟦
25	tmp_8(+)									🟦
26	node_62(write)									🟦
27	j_1(+)						🟦			

Figure 3.11: Performance view for solution2

Note that the highlighted write operation occurs in state C3, *node_33(write)*. Right clicking on this cell and selecting *Goto Source* will highlight the associated line of code in the source file. This is a write operation initialised as a write to a port in the RTL which occurs before any operations in the loop, *Product*, can be executed. This prevents the flattening of loop *Product* in to *Row_Col*.

Furthermore, the inability to meet the target of II = 1 can be explained by considering consecutive iterations of the loop. Consulting the console reveals the following message:

```
@W [SCHED-68] Unable to enforce a carried dependency constraint (II =
1, distance = 1) between ‘store’ operation (matrix_mult.cpp:16) of
variable ‘tmp_8’ on array ‘prod’ and ‘load’ operation (‘prod_load’,
matrix_mult.cpp16) on array ‘prod’.
```

There exists a dependency between iterations of the operation at line 18 of the source code, which is the operation within the *Product* loop.

```
prod[i][j] += a[i][k] * b[k][j];
```

Due to the presence of the += operator, this line of code contains a read from array *prod* (the aforementioned load operation) and a write to array *prod* (a store operation). With an II of 1, a succeeding *Product* loop iteration would occur one clock cycle after the initiation of the first iteration. This is visualised in Figure 3.12. With II set to 1, the highlighted overlap is observed. Arrays are mapped to BRAM by default, and since this overlap requires a *read* and a *write* operation to be performed on the same clock cycle, this is simply not possible as both operations cannot occur on the BRAM at the same time. Therefore, setting the II to 2 allows the *write* operation to be completed before the *read* operation of the next loop iteration begins.

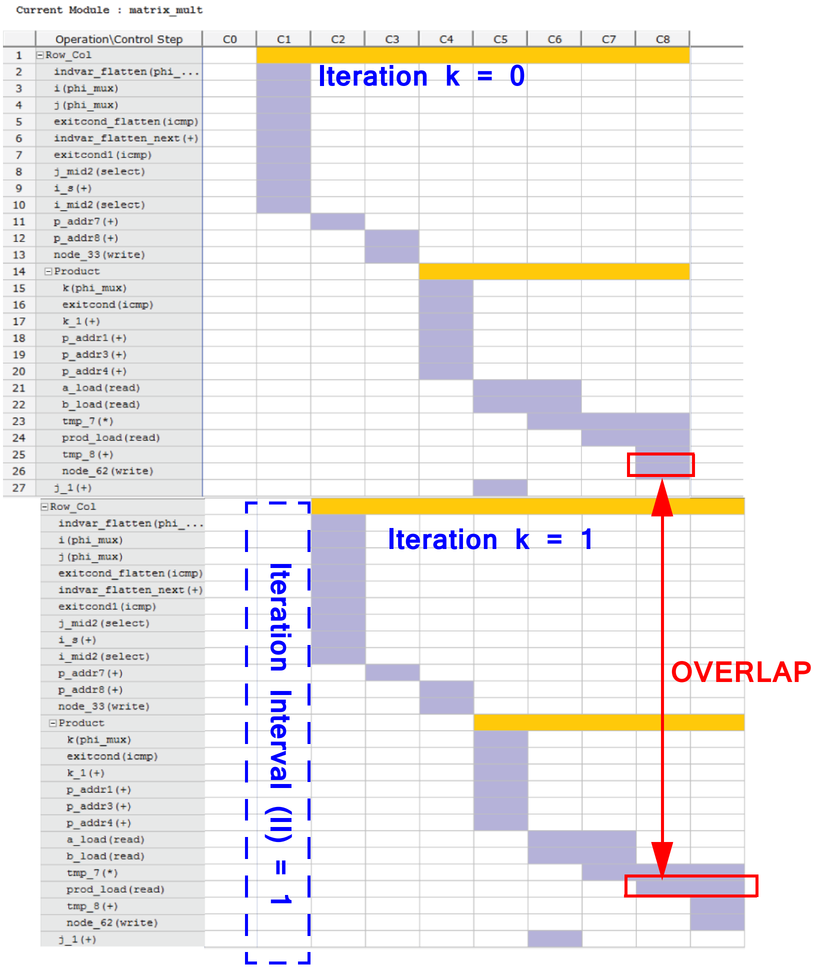
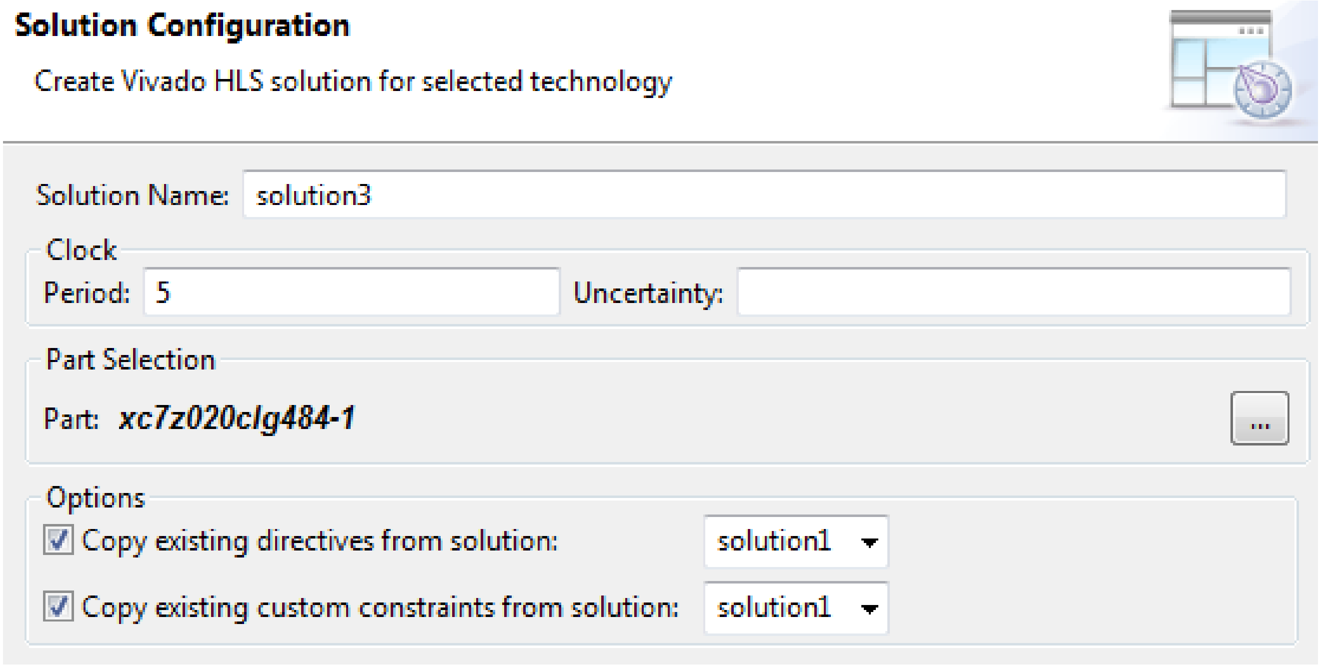


Figure 3.12: Consecutive iterations of Product loop with II = 1

(k) Return to the *Synthesis* perspective by clicking on Synthesis .

We will now create a new solution which pipelines the *Col* loop, unrolling the *Product* loop at to eliminate inter-iteration dependency but at the cost of increased operators and hence hardware cost.

(l) Create a new solution for the design by either clicking the ***New Solution*** button in the toolbar or the menu option ***Project*** > ***New Solution***. From the drop-down menus, ensure solution1 is selected, as in Figure 3.13, as this contains no existing directives or constraints.



Solution Configuration

Create Vivado HLS solution for selected technology

Solution Name: solution3

Clock

Period: 5 Uncertainty:

Part Selection

Part: ***xc7z020clg484-1***

Options

☑ Copy existing directives from solution: solution1

☑ Copy existing custom constraints from solution: solution1

Figure 3.13: Configuring solution3

Click ***Finish*** to create the solution.

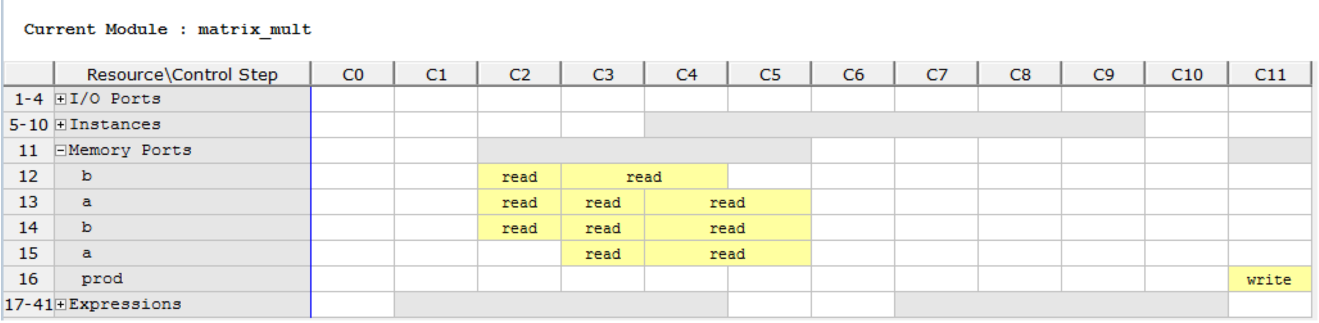
(m) Ensure the source code *matrix_mult.cpp* is visible in the editor. In the ***Directives*** tab, right-click on loop ***Col*** and select *Insert Directive*. From the drop-down menu, select directive type *PIPELINE* and click ***OK*** to select the directive with the defaults (II = 1).

(n) Click the ***C Synthesis*** button to synthesise the RTL design. Observing the *Console* will show that while *Product* was unrolled and loop *Row* was flattened the II target of 1 could not be met for loop *Row_Col*, this time due to limitations in the resources.

```
@W [SCHED-69] Unable to schedule ‘load’ operation (‘b_load_4’,
matrix_mult.cpp:16) on array ‘b’ due to limited memory ports.
```

(o) Open the *Analysis* perspective by clicking on Analysis . This will open the *Performance* view. Switch to the *Resource* view by clicking the tab at the bottom of the screen.

(p) Expand the *Memory Ports* to view resource sharing on the memory within the system.



Current Module : matrix_mult

	Resource\Control Step	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11
1-4	⊞I/O Ports												
5-10	⊞Instances												
11	⊟Memory Ports												
12	b			read	read								
13	a			read	read	read							
14	b			read	read	read							
15	a				read	read							
16	prod												write
17-41	⊞Expressions												

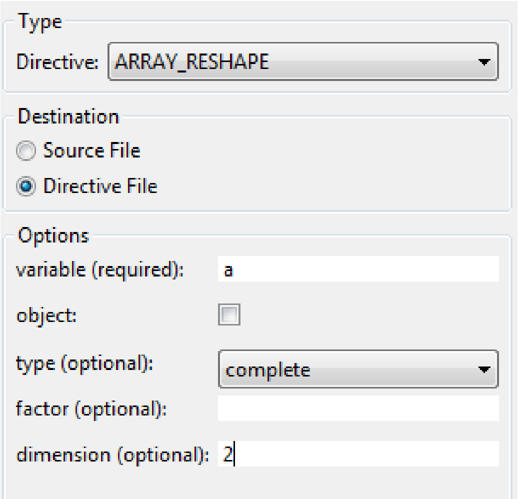
Figure 3.14: Resource sharing on memory ports of solution3

Figure 3.14 shows the operations per resource on each clock cycle. In actual fact, the 2 cycle read operation on *b* beginning in C3 overlaps with those in C4 so only a single cycle is visible. There are instances of both *a* and *b* being subjected to 3 read operations at once, which you will remember is not possible for dual-port BRAM. It is therefore necessary to partition these arrays into smaller sections, allowing modification of the array without altering the source code.

(q) Return to the *Synthesis* perspective by clicking on Synthesis .
Create a new solution for the design by either clicking the ***New Solution*** button in the toolbar or the menu option ***Project*** > ***New Solution***. Click ***Finish*** to accept the defaults for solution4.

For this solution, we will reshape the input arrays using directives. The ***Product*** loop is accessed via loop index *k,* therefore arrays ***a*** and ***b*** should be partitioned along their *k* dimension. Inspecting line 16 of *matrix_mult.cpp* it is observed that for *a[i][k]* this is dimension 2 and for *b[k][j]* dimension 1.

(r) Ensure the source code *matrix_mult.cpp* is visible in the editor, and open the ***Directives*** tab. Right-click on variable ***a*** and select *Insert Directive*. Ensure the directive is configured as in Figure 3.15, with *ARRAY_RESHAPE* selected as directive type and dimension specified as 2.



Type

Directive: ARRAY_RESHAPE

Destination

○ Source File

● Directive File

Options

variable (required): a

object: □

type (optional): complete

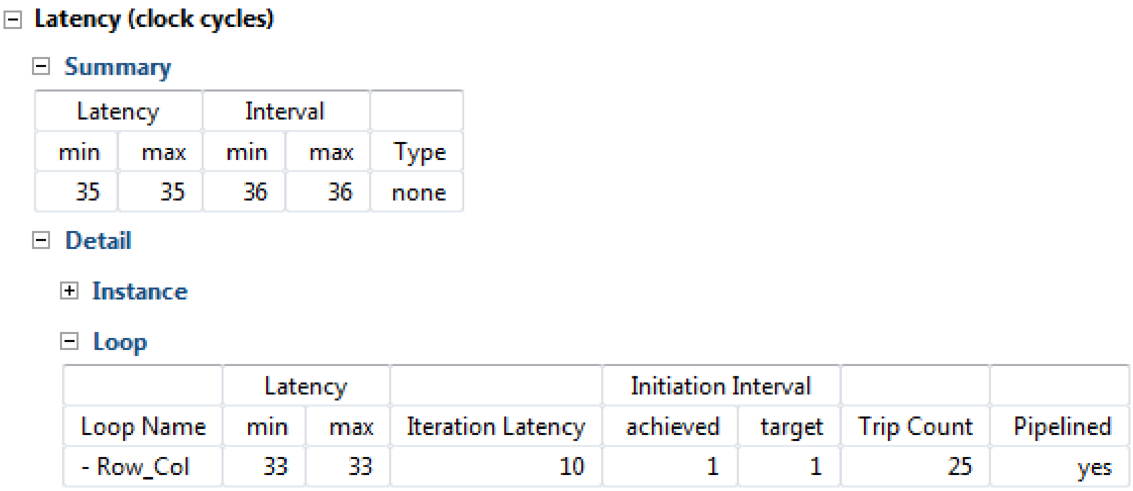
factor (optional):

dimension (optional): 2

Figure 3.15: Directive configurations for reshaping array a

(s) Repeat for array ***b***, this time ensuring dimension is set to 1.

(t) Click the ***C Synthesis*** button to synthesise the RTL design. The synthesis report will open, showing that the target II of 1 has now been met.



⊟ **Latency (clock cycles)**

⊟ **Summary**

Latency		Interval		
min	max	min	max	Type
35	35	36	36	none

⊟ **Detail**

⊞ **Instance**

⊟ **Loop**

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Row_Col	33	33	10	1	1	25	yes

Figure 3.16: Synthesis report for solution4

The top-level of the design takes 35 clock cycles for completion, with the *Row_Col* loop outputting a sample after an iteration latency of 10. A sample is then read in every cycle (due to an II of 1), and after 25 counts all samples have been read in. The 35 clock cycle life of this design is therefore justified by the 25 counts plus the latency of 10, as 25 + 10 = 35.

The function then proceeds to calculate the next set of data.

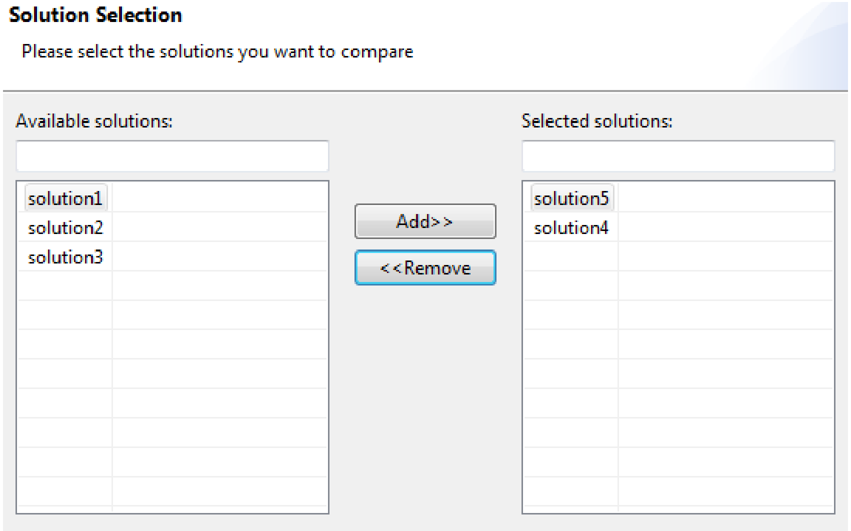
(u) The final optimisation in this exercise is to pipeline the function, rather than the loops within that function for comparison. Create a new solution for the design by either clicking the ***New Solution*** button in the toolbar or the menu option ***Project > New Solution***. Click ***Finish*** to accept the defaults for solution5.

(v) Ensure the source code *matrix_mult.cpp* is visible in the editor, and open the ***Directives*** tab. First, remove the previously inserted pipeline directive on loop *Col*. Right-click on the directive and select *Remove Directive*.

(w) Right-click on the top level function *matrix_mult* and select *Insert Directive*. Select *PIPELINE* as the directive type and click ***OK***.

(x) Click the ***C Synthesis*** button to synthesise the RTL design.

(y) Vivado HLS provides a tool for comparing synthesis reports. Click the button or the menu option ***Project > Compare Reports***.



Solution Selection

Please select the solutions you want to compare

Available solutions:

solution1
solution2
solution3

Selected solutions:

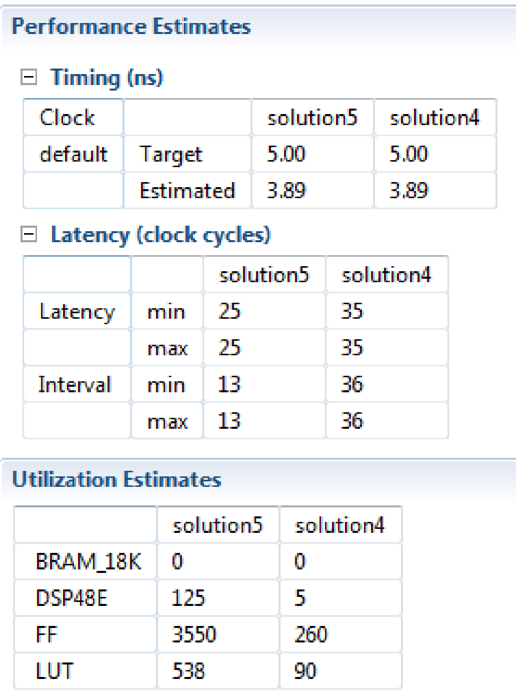
solution5
solution4

Figure 3.17: Solution selection for comparison

Ensure solution4 and solution5 are added as in Figure 3.17. Click ***OK***.

(z) Figure 3.18 shows the comparison of synthesis report for solution4 (with loop pipelining) and solution5 (with top level function pipelining). It is observed that pipelining the top level function results in a design which reaches completion in fewer clocks, requiring only 13 clock cycles to begin a new transaction, rather than 36 for pipelining the loop.

However, this comes at the cost of increased hardware utilisation due to unrolling of all loops within the design. A tradeoff is therefore necessary between system performance and the hardware utilisation of the design, and it is possible that a partially unrolled design may meet the performance requirements at a reduced hardware cost.



Performance Estimates

⊟ Timing (ns)

Clock		solution5	solution4
default	Target	5.00	5.00
	Estimated	3.89	3.89

⊟ Latency (clock cycles)

		solution5	solution4
Latency	min	25	35
	max	25	35
Interval	min	13	36
	max	13	36

Utilization Estimates

	solution5	solution4
BRAM_18K	0	0
DSP48E	125	5
FF	3550	260
LUT	538	90

Figure 3.18: Comparison of solution4 and solution5

(aa) This completes the exercise. Close the Vivado HLS GUI.

We will now briefly explore the concept of interface synthesis in Vivado HLS, using the matrix multiplier function of the previous two exercises.

Exercise 3C Interface Synthesis

(a) Launch the command prompt by navigating to **Start** > **All Programs** > **Xilinx Design Tools** > **Vivado 2014.1** > **Vivado HLS** > **Vivado HLS 2014.1 Command Prompt**.

(b) Change the working directory to **C:\Zynq_Book\HLS\tut3C**. This folder contains the source and test files for a project, and also the Tcl script required to build the project, ***run_hls.tcl***.

(c) Run the Tcl script using the command:

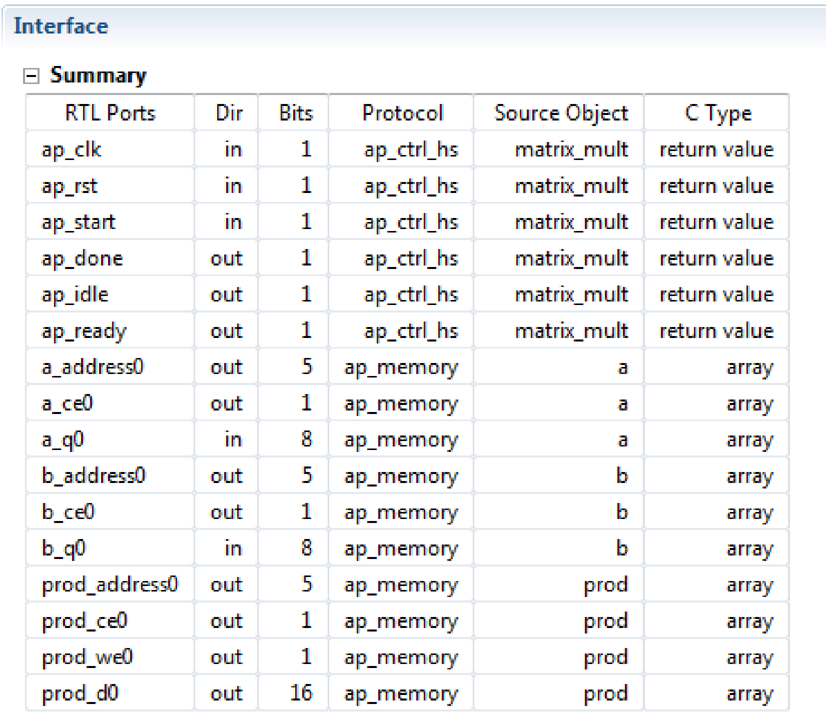
```
vivado_hls -f run_hls.tcl
```

(d) To open the project in the Vivado HLS GUI enter the following command:

```
vivado_hls -p matrix_mult_prj
```

And press ***Enter***. This will open the Vivado HLS GUI for the project, which we will utilise in the next exercise.

(e) Open the source file *matrix_mult.cpp* from the ***Source*** section of the ***Explorer*** tab and click the ***C Synthesis*** button to synthesise the RTL design. When the synthesis report opens, scroll to the *Interface* section.



Interface

⊟ Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	matrix_mult	return value
ap_rst	in	1	ap_ctrl_hs	matrix_mult	return value
ap_start	in	1	ap_ctrl_hs	matrix_mult	return value
ap_done	out	1	ap_ctrl_hs	matrix_mult	return value
ap_idle	out	1	ap_ctrl_hs	matrix_mult	return value
ap_ready	out	1	ap_ctrl_hs	matrix_mult	return value
a_address0	out	5	ap_memory	a	array
a_ce0	out	1	ap_memory	a	array
a_q0	in	8	ap_memory	a	array
b_address0	out	5	ap_memory	b	array
b_ce0	out	1	ap_memory	b	array
b_q0	in	8	ap_memory	b	array
prod_address0	out	5	ap_memory	prod	array
prod_ce0	out	1	ap_memory	prod	array
prod_we0	out	1	ap_memory	prod	array
prod_d0	out	16	ap_memory	prod	array

Figure 3.19: Interface summary for solution1

Note that the input arrays ***a*** and ***b***, and the resultant product array ***prod*** have been implemented using the *ap_memory* protocol. This is inferred from the C++ source code, as the array type corresponds with the structure of memory.

Input arrays ***a*** and ***b*** are both 8 -bit signals on ports *a_q0* and *b_q0*. The output array, ***prod*** is a 16-bit signal on port *prod_d0*. Each signal has a corresponding 5-bit address port, designated as *a_address0, b_address0* and *prod_address0.*

The protocol also requires clock enable signals (*a_ce0* and *b_ce0*), and a write enable (*prod_we0).*

Since the design requires more than one clock cycle to complete and is therefore synchronous, a clock and reset port have been synthesised as *ap_clk* and *ap_rst,* and both are 1-bit signals.

A block level control protocol with handshaking, *ap_ctrl_hs*, has also been implemented (*ap_start, ap_done, ap_idle* and *ap_ready).*

- The *ap_start* input is asserted, prompting block operation. This produces three output control signals indicating the stage of operation.
- *ap_ready* indicates that the block is ready for new inputs.
- *ap_idle* is an indication that data is currently processing data.
- *ap_done* indicates that output data has been processed and is available.

Recalling Exercise 3B, the arrays were partitioned to reduce each into several smaller sections with expanded ports, control signals and implementation resources. This increased the bandwidth. This directly influenced the interface synthesis through use of directives.

This concludes this introduction to the design flow of Vivado HLS. This tool will be used further in future exercises, and synthesised RTL will be implemented as part of a larger functional model.

The Zynq Book Tutorial 4

IP Creation

v1.2, May 2014

Revision History

Date	Version	Changes
22/10/2013	1.0	First release for Vivado Design Suite version 2013.3
28/01/2014	1.1	Updated for changes in Vivado Design Suite version 2013.4
06/05/2014	1.2	Updated for changes in Vivado Design Suite version 2014.1

Introduction

The exercises in this tutorial will guide you through the process of creating custom IP modules, that are compatible with Vivado IP Integrator, from a variety of different sources. All created IP will be compatible with the Xilinx supported AXI-Lite interface, and will be connected as slave devices when implemented in Vivado IP Integrator.

All IP creation methods that are covered here coincide with those covered in the book:

- HDL
- MathWorks HDL Coder
- Xilinx Vivado HLS

The tutorial is split into three exercises, and is organised as follows:

Exercise 4A - In this exercise, HDL will be used to create a controller which will allow the LEDs on the ZedBoard to be controlled by software running on the PS. The Create and Package IP Wizard will be used to create an AXI-Lite interface wrapper which the LED control process and interface will be added to. The IP packaging process will then be used to create an IP block which is compatible with IP Integrator.

Exercise 4B - HDL Coder, the MathWorks HDL generation tool, will be explored in this exercise. A Least Mean Squares (LMS) adaptive filter will be created and tested in the Simulink workspace. The LMS design will then be used to generate HDL code by invoking the HDL Coder Workflow Advisor, where the option to generate a Xilinx IP Core will be selected. The various stages of the workflow will verify the design to ensure that it is HDL Coder compliant and produce the HDL code in a format that is compatible with IP Integrator.

Exercise 4C - In this final exercise, Vivado HLS will be used to create an IP core for a Numerically Controlled Oscillator (NCO). An existing C-code algorithm will be simulated for testing, and ran through the various stages of synthesis in order to create an IP Integrator compatible IP core.

Exercise 4A Creating IP in HDL

With Zynq devices comprising of both PS and PL parts, most IP that is created to run in PL should be able to communicate with software running on the PS. This requires that IP should be packaged with an interface that is compatible with the PS (in this case the AXI interface).

When creating IP in HDL, Vivado provides a set of AXI interface templates which can be created and customised via the *Create and Package IP Wizard*. The wizard, as the name suggests, facilitates two major functions: the creation of AXI4 IP peripherals; and the packaging of existing source files into an IP package which is compatible with the IP Integrator tool.

In this exercise we will actually be making use of both of these features to firstly create an AXI4-Lite IP template to which we will add functionality to allow the LEDs on the ZedBoard to be controlled via a software application running on the Zynq PS. Once the functionality has been added to the template, the source files will be packaged into an IP Integrator compatible IP block which will be included in a simple Zynq processor system.

We will start by creating a new Vivado project.

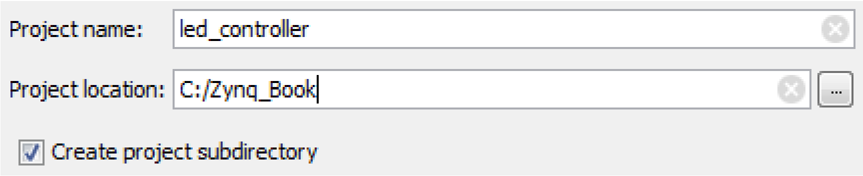
(a) Launch Vivado by double-clicking on the Vivado desktop icon: , or by navigating to **Start > All Programs > Xilinx Design Tools > Vivado 2014.1 > Vivado 2014.1**

(b) Select ***Create New Project*** from the *Getting Started* screen.

(c) The *New Project* dialogue will open. Click ***Next***.

(d) At the Project Name dialogue, enter ***led_controller*** as the ***Project name*** and ***C:/Zynq_Book*** as ***Project location***.

Make sure that you select the option to ***Create project subdirectory***. Ensure that all options match Figure 4.1.



Project name: led_controller

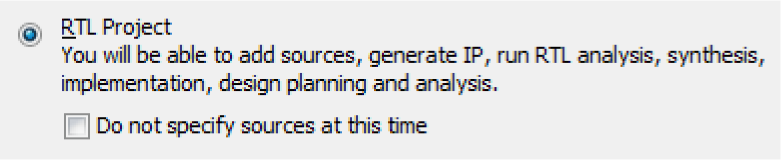
Project location: C:/Zynq_Book

☑ Create project subdirectory

Figure 4.1: Vivado Project Name specification - led_controller

Click ***Next***.

(e) Select ***RTL Project*** at the *Project Type* dialogue, and ensure that the option ***Do not specify sources at this time*** is not selected:



● RTL Project
You will be able to add sources, generate IP, run RTL analysis, synthesis, implementation, design planning and analysis.
□ Do not specify sources at this time

Click ***Next***.

(f) Select ***VHDL*** as the ***Target language*** in the *Add Sources* dialogue.

If existing sources, in the form of HDL or netlist files, were to be added to the project they could be imported at this stage.

As we do not have any sources to add to the project, click ***Next***.

(g) The *Add Existing IP (optional)* dialogue will open.

If existing IP sources were to be included in the project, they could be added here.

As we do not have any existing IP to add, click ***Next***.

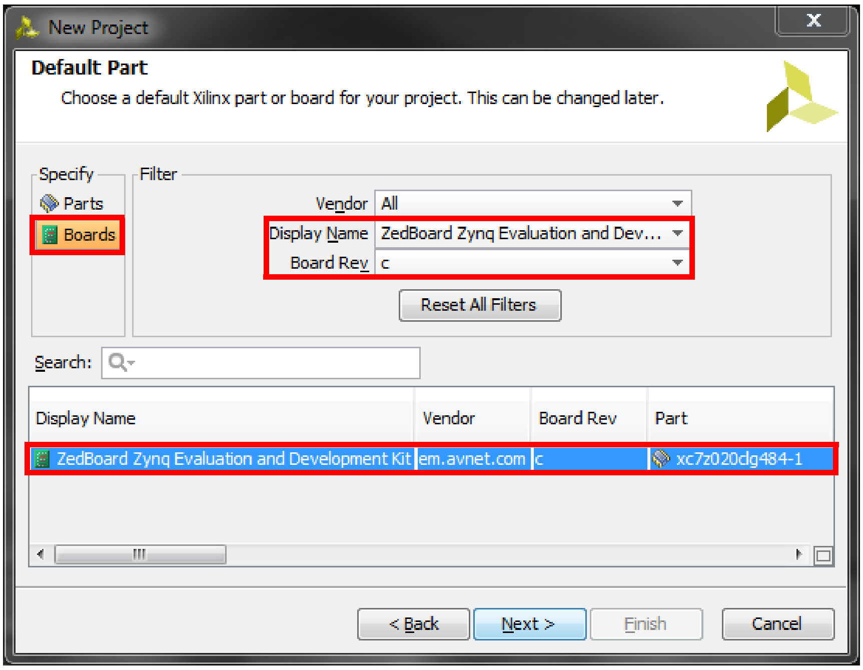
(h) The *Add Constraints (optional)* dialogue will open.

This is the stage were any physical or timing constraints files could be added to the project.

As we do not have any constraints files to add, click ***Next***.

(i) The *Default Part* dialog will open. Here we will be selecting the Zynq part which we are targeting. In this particular case we will be targeting the Zynq-7020 on the ZedBoard, but if you have a different development board, it is easy to choose your particular board instead.

Select ***Boards*** from the *Specify* pane, ***ZedBoard Zynq Evaluation and Development Kit*** as the *Display Name*, and finally select the *Board Rev* which you have. In Figure 4.2 **version C** of the **ZedBoard** has been selected.



New Project

Default Part

Choose a default Xilinx part or board for your project. This can be changed later.

Specify: Parts, Boards

Filter

Vendor: All

Display Name: ZedBoard Zynq Evaluation and Dev...

Board Rev: c

Search:

Display Name	Vendor	Board Rev	Part
ZedBoard Zynq Evaluation and Development Kit	em.avnet.com	c	xc7z020clg484-1

Figure 4.2: Vivado Default Part dialogue

Click ***Next***.

(j) Review the *New Project Summary* dialogue, and click ***Finish*** to create the project.

With the new project created, we can begin the process of creating our HDL-based IP.

(k) From the menu bar, select **Tools > Create and Package IP ...**, as in Figure 4.3, to launch the ***Create and Package IP Wizard***.

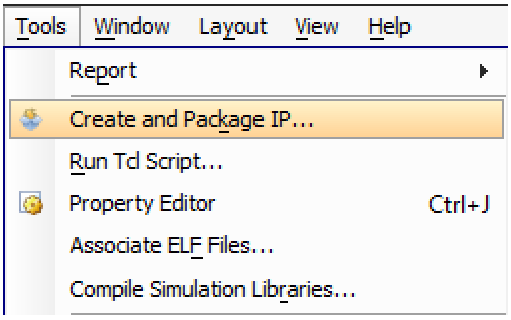
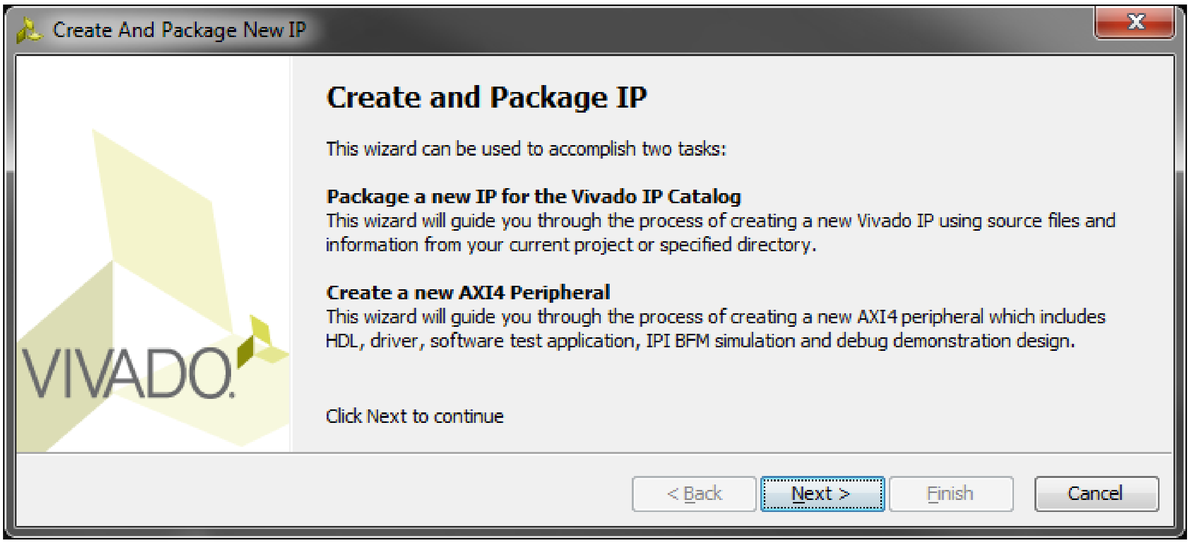


Figure 4.3: Create and Package IP menu bar selection

(l) The *Create and Package IP Wizard* dialogue will launch, as shown in Figure 4.4.



Create and Package IP

This wizard can be used to accomplish two tasks:

Package a new IP for the Vivado IP Catalog
This wizard will guide you through the process of creating a new Vivado IP using source files and information from your current project or specified directory.

Create a new AXI4 Peripheral
This wizard will guide you through the process of creating a new AXI4 peripheral which includes HDL, driver, software test application, IPI BFM simulation and debug demonstration design.

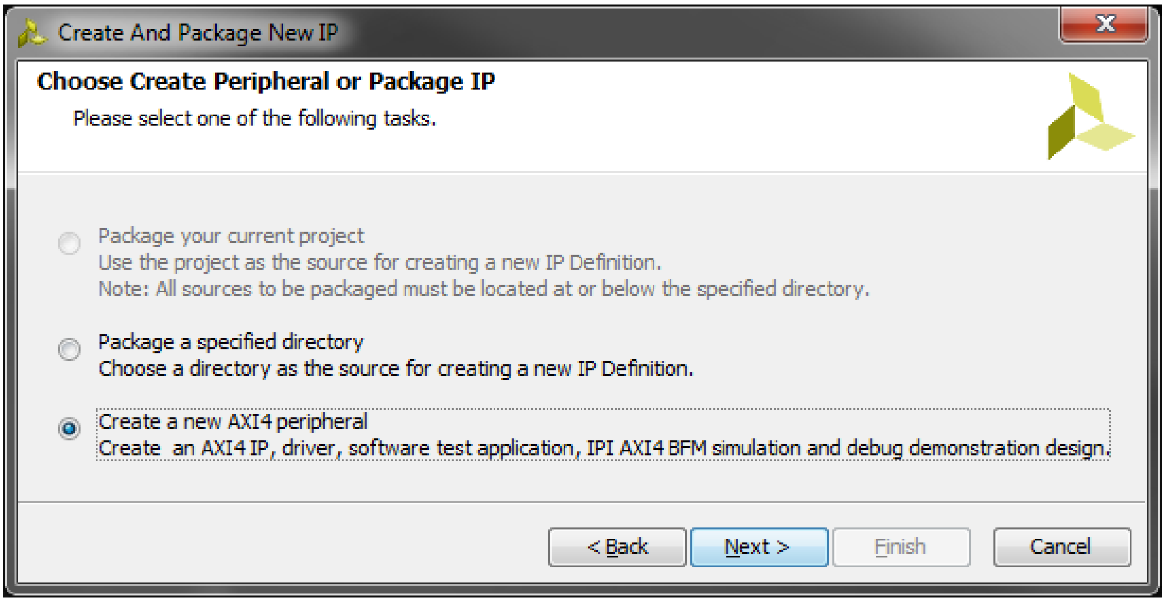
Click Next to continue

Figure 4.4: Create and Package IP Wizard dialogue

Click ***Next***.

The *Choose Create Peripheral or Package IP* dialogue (Figure 4.5) is where we specify whether to create a new peripheral template file or to package existing source files into an IP core. In our case we want to create a new IP template.

(m) Select ***Create new AXI peripheral***, as shown in Figure 4.5.



Choose Create Peripheral or Package IP

Please select one of the following tasks.

○ Package your current project
Use the project as the source for creating a new IP Definition.
Note: All sources to be packaged must be located at or below the specified directory.

○ Package a specified directory
Choose a directory as the source for creating a new IP Definition.

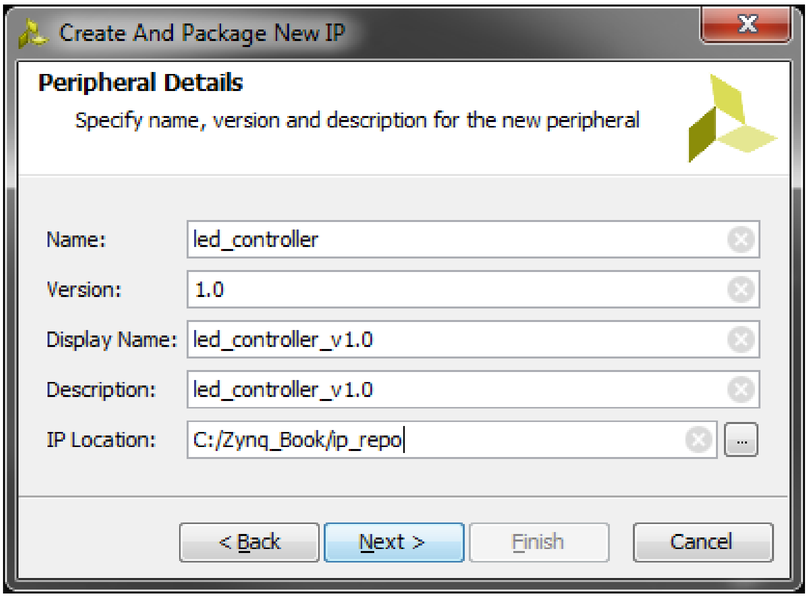
● Create a new AXI4 peripheral
Create an AXI4 IP, driver, software test application, IPI AXI4 BFM simulation and debug demonstration design.

Figure 4.5: Choose Create or Package IP dialogue

Click ***Next***.

The *Peripheral Details* dialogue allows you to specify the **Vendor, Library, Name and Version (VLNV)** information, as well as other details, for the new peripheral, leaving the *IP Location* as the default.

(n) Fill in the details as shown in Figure 4.6.



Peripheral Details

Specify name, version and description for the new peripheral

Name: led_controller

Version: 1.0

Display Name: led_controller_v1.0

Description: led_controller_v1.0

IP Location: C:/Zynq_Book/ip_repo

Figure 4.6: Peripheral Details dialogue

Click ***Next***.

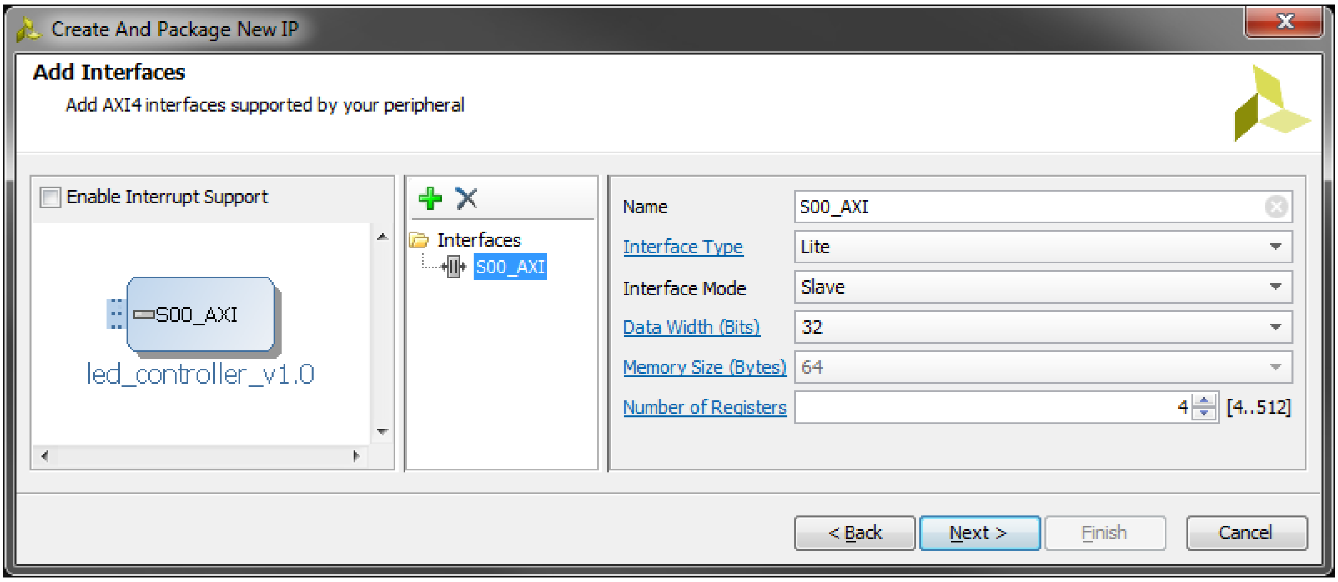
The *Add Interface* dialogue allows you to specify the AXI4 interface(s) that will be present in your custom peripheral. Here you can specify:

- Number of interfaces
- Interface type (AXI-Lite, AXI-Stream or AXI-Full)
- Interface mode (slave or master)
- Interface data width

Features specific to individual interface types will also be available when the corresponding type is selected.

As our peripheral is a simple controller for the LEDs which only requires single values to be transferred to it, an AXI-Lite slave interface is sufficient. Only one memory mapped register is required for our simple controller, but as the minimum number that can be specified in the dialogue is 4, we will choose that.

(o) Specify the *Add Interface* dialogue as shown in Figure 4.7.



□ Enable Interrupt Support

Name: S00_AXI
Interface Type: Lite
Interface Mode: Slave
Data Width (Bits): 32
Memory Size (Bytes): 64
Number of Registers: 4 [4..512]

Figure 4.7: Add Interface dialogue

Click ***Next***.

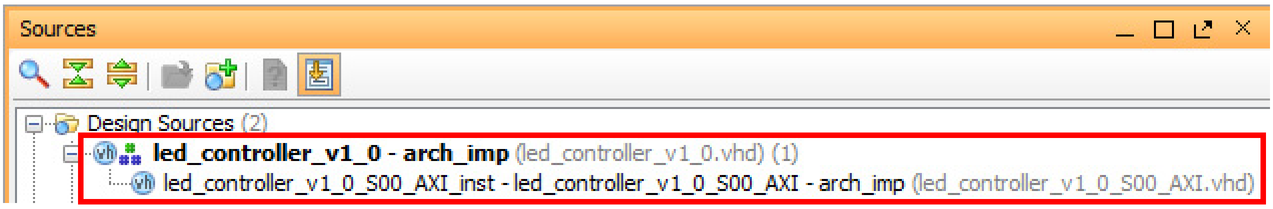
(p) Review the information in the *Create Peripheral* dialogue, which details the output files which will be created.

Select the option to ***Edit IP***. This will create the IP peripheral files and create a new Vivado project where the functionality of the peripheral can be modified in the source HDL code, and then packaged.

Click ***Finish*** to close the *Wizard* and create the peripheral template.

A new Vivado project, named **edit_led_controller_v1_0**, will open.

In the *Sources* pane, you should see two HDL source files:



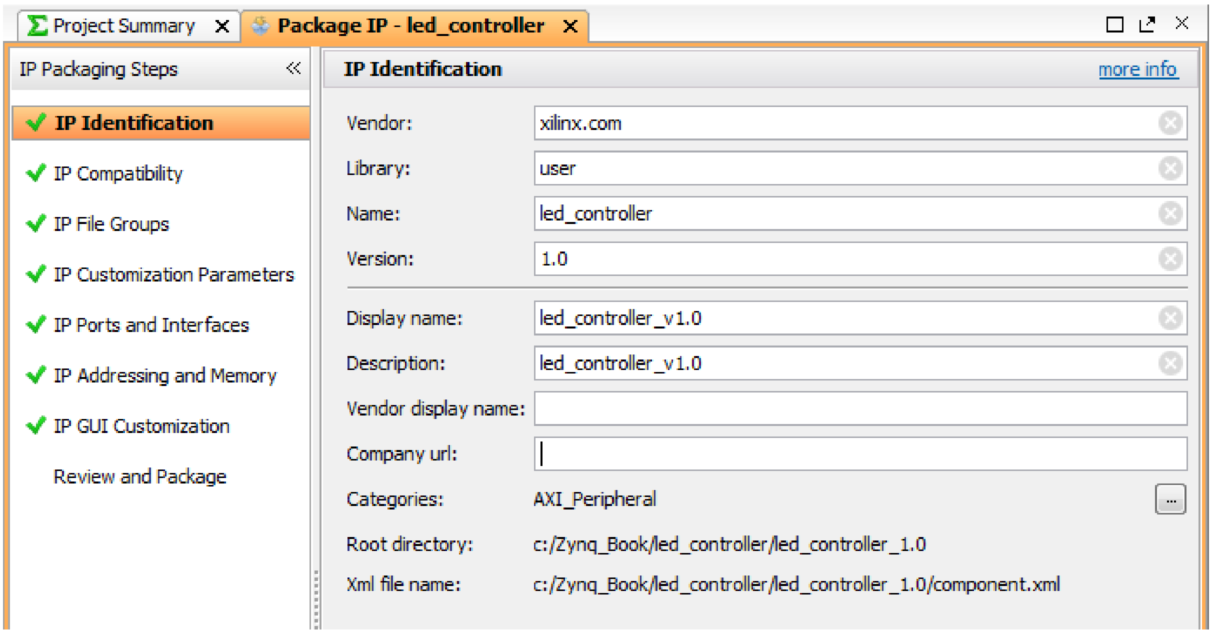
Design Sources (2)
led_controller_v1_0 - arch_imp (led_controller_v1_0.vhd) (1)
led_controller_v1_0_S00_AXI_inst - led_controller_v1_0_S00_AXI - arch_imp (led_controller_v1_0_S00_AXI.vhd)

As we specified our target language as **VHDL** in Step (f) earlier, the template files have been generated in VHDL. Had we specified Verilog as the target language, Verilog source files would have been created.

The two source files are:

- **led_controller_v1_0.vhd** — This file instantiates all AXI-Lite interfaces. In this case, only one interface is present.
- **led_controller_v1_0_S00_AXI.vhd** — This file contains the AXI4-Lite interface functionality which handles the interactions between the peripheral in the PL and the software running on the PS.

The *IP Packager* pane will also be open in the *Workspace*:



IP Identification

Vendor: xilinx.com
Library: user
Name: led_controller
Version: 1.0
Display name: led_controller_v1.0
Description: led_controller_v1.0
Vendor display name:
Company url:
Categories: AXI_Peripheral
Root directory: c:/Zynq_Book/led_controller/led_controller_1.0
Xml file name: c:/Zynq_Book/led_controller/led_controller_1.0/component.xml

The information that we specified about our peripheral in Step (n) will be visible.

We can now add the functionality to our **led_controller** peripheral. We will be adding a new output port to the peripheral template to allow it to connect to the LED pins on the Zynq device, as well as assigning the value received from the Zynq PS to the new output port.

(q) Open ***led_controller_v1_0_S00_AXI.vhd*** by double-clicking on it in the *Sources* pane. The file will open in the *Workspace*.

(r) Scroll down until you see the following comment in the entity port declaration:

```
-- Users to add ports here
```

and add the following port definition directly below the comment:

```
LEDs_out : out std_logic_vector(7 downto 0);
```

This creates a new output port with a width of 8-bits (a single bit to represent each of the LEDs on the ZedBoard).

(s) Scroll to the bottom of the file. You should see the following comment:

```
-- Add user logic here
```

and add the following port/signal assignment:

```
LEDs_out <= slv_reg0(7 downto 0);
```

This assigns the value that is received from the Zynq PS (stored in the signal slv_reg0) to the output port that we created in the previous step.

(t) Save the file by selecting **File > Save File** from the Menu Bar, or using the keyboard shortcut **Ctrl+S**.

(u) Open ***led_controller_v1_0.vhd*** by double-clicking on it in the *Sources* pane. The file will open in the *Workspace*.

We must once again create a new output port to the top-level source file, and map it to the equivalent port that we created in the AXI4-Lite interface file in the previous steps.

(v) Scroll down until you see the following comment in the entity port declaration:

```
-- Users to add ports here
```

and add the following port definition directly below the comment:

```
LEDs_out : out std_logic_vector(7 downto 0);
```

As we added a new port to the AXI4-Lite interface file, we must also add it to the component declaration in the top-level file.

(w) Scroll down until you see the comment:

```
-- component declaration
```

A few lines further down you will see the component port declaration:

```
port (
```

Inside the port declaration (below the “`port (`“line), add the following output port definition:

```
LEDs_out : out std_logic_vector(7 downto 0);
```

Finally, we must add a port mapping between the LED output ports of the top-level file and the AXI4-Lite interface file.

(x) Scroll down until you see the comment:

```
-- Instantiation of Axi Bus Interface S00_AXI
```

A few lines further down you will see the component port map:

```
port map (
```

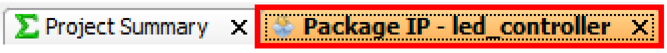
Inside the component port map (below "`port map (`" line), add the following port map:

```
LEDs_out => LEDs_out,
```

(y) Save the file.

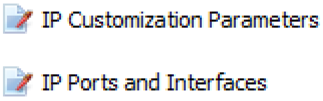
Now that we have made the necessary modifications to the peripheral source files, we must repackage the IP to merge the changes.

(z) Return to **IP Packager** by selecting the ***Package IP - led_controller*** tab in the *Workspace*:



Project Summary ✕ Package IP - led_controller ✕

IP Packager will detect the changes to the source files, and the areas which need refreshed will be highlighted with the following icon: ✎. You should see that the following two areas need refreshed:

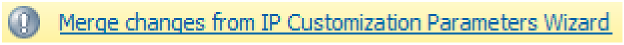


IP Customization Parameters

IP Ports and Interfaces

(aa) Select ***IP Customization Parameters*** in the *IP Packager* pane.

You should see the following information message at the top of the pane:



Merge changes from IP Customization Parameters Wizard

Click ***Merge changes from IP Customization Parameters Wizard***

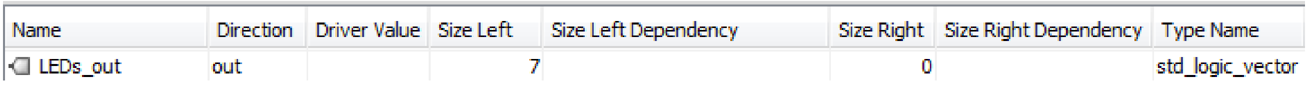
This will update the IP Packager information to reflect the changes made in the HDL source files.

NOTE: This process updates IP Packager information for all areas. You should see that the area of IP Ports no longer needs updated, and the ✎ icon has now been removed.

To verify that IP Packager has updated the IP Ports area, we will open it and check.

(ab) Select ***IP Ports and Interfaces*** from the *IP Packager* pane.

You should notice that the **LEDs_out** port that we added to the source files has been added to the *IP Ports and Interfaces* pane:

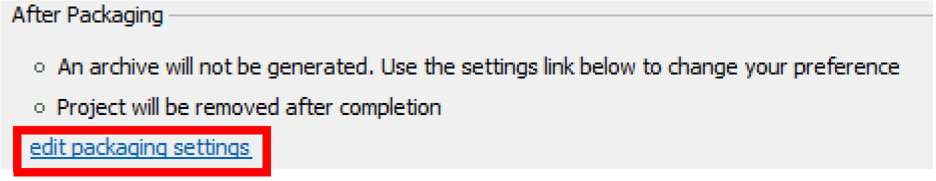


Name	Direction	Driver Value	Size Left	Size Left Dependency	Size Right	Size Right Dependency	Type Name
LEDs_out	out		7		0		std_logic_vector

The final step in creating our new IP peripheral, is to package the IP.

(ac) Select ***Review and Package*** from the *IP Packager* pane.

(ad) In the *After Packaging* panel, click ***edit packaging settings*** at the bottom:

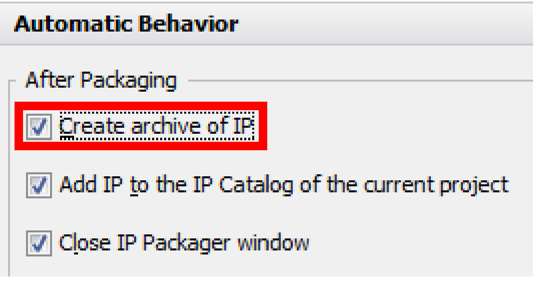


After Packaging

- An archive will not be generated. Use the settings link below to change your preference
- Project will be removed after completion

edit packaging settings

(ae) In the *Automatic Behaviour* panel, enable the option to ***Create archive of IP***:



Automatic Behavior

After Packaging

☑ Create archive of IP

☑ Add IP to the IP Catalog of the current project

☑ Close IP Packager window

This makes a ZIP file archive of the packaged IP.

(af) Click ***OK*** to apply the setting.

(ag) Review the information provided in the *Review and Package* window, and click ***Re-Package IP***.

(ah) The changes made to the IP peripheral will be included in the repackaged IP, and the Vivado project will close.

We will now return to our original Vivado project, and create a simple Zynq processor block design to check that the functionality of our LED controller peripheral.

To start, we will create a new Block Design and add the IP peripheral which we just created to the design.

(ai) In the *Flow Navigator* window, select ***Create Block Design*** from the *IP Integrator* section.

Enter ***led_test_system*** in the *Design name* box, and click ***OK*** to create the blank design.

(aj) Right-click anywhere in the blank canvas, and select ***Add IP***. Alternatively, use the keyboard shortcut ***Ctrl+I***. This will bring up to pop-up IP Catalog window.

Enter ***led*** in the *Search* box, and double-click ***led_controller_v1_0*** to add an instance of the LED controller IP to the design.

An led_controller_v1_0 block will now be present in the block design, as shown in Figure 4.8.

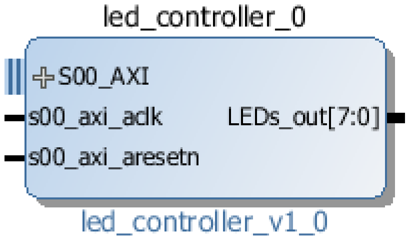


Figure 4.8: led_controller block

The 8-bit **LEDs_out** port that we added to the peripheral is present on the right side of the block.

To enable the peripheral to connect to the LEDs on the ZedBoard, we must make the **LEDs_out** port external. This allows the output port to be connected to specific physical pins on the Zynq device, which are connected to the LEDs.

(ak) Hover the mouse pointer over the **LEDs_out** interface (the little black stub next to the interface name) on the **led_controller** block until the cursor changes to a pencil. Right-click and select ***Make External***. Alternatively, select the interface and use the keyboard shortcut ***Ctrl+T***.

The block design should now resemble Figure 4.9.

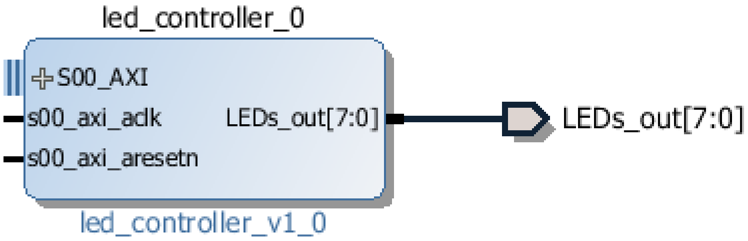
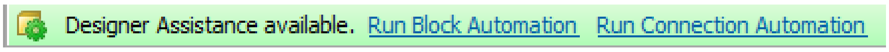


Figure 4.9: led_controller block with external port

The next step is to add a Zynq Processing System block to the design and connect the LED Controller to it.

(al) Add an instance of the **Zynq7 Processing System**, using the same procedure as in Step (aj).

(am)The *Designer Assistance* message at the top of the canvas will appear:



Designer Assistance available. Run Block Automation Run Connection Automation

Click ***Run Block Automation*** and select ***processing_system7_0***.

An information message will appear. Ensure that ***Apply Board Preset*** is selected, and click ***OK***.

This will make all necessary modifications to the Zynq processing system that relate to the board preset (in this case the ZedBoard) and make required external connections.

The next step that has to be carried out to the block design, is to connect the LED Controller to the Zynq Processing System. This step can also be carried out using Designer Assistance.

(an) In the *Designer Assistance* message, click ***Run Connection Automation*** and select ***led_controller_0/S00_AXI***.

An information message will appear. Click ***OK***.

This will add some additional blocks to the design which are required to connect the LED Controller to the Zynq Processing System.

Our block design is now complete.

(ao) Validate the design by selecting **Tools > Validate Design** from the *Menu Bar*. Alternatively, select the ***Validate Design*** button, ✓, from the *Main Toolbar*, or use the keyboard shortcut ***F6***.

Dismiss the *Validate Design* message by clicking ***OK***.

We can now generate the HDL files for the design.

(ap) In the *Sources* pane, right-click on the **led_test_system** block design and select ***Create HDL Wrapper***.

Select ***Let Vivado manage wrapper and auto-update*** and click ***OK***.

This will create the top-level HDL file for the design.

We must now connect the LEDs_out port of the design to the correct pins on the Zynq device. This is done through the specification of constraints in an XDC file.

(aq) In the *Flow Navigator* window, select ***Add Sources*** from the *Project Manager* section.

The *Add Sources* dialogue will open.

Select ***Add or Create Constraints***, and click ***Next***.

(ar) Click ***Create File...***

The *Create Constraints File* dialogue will open.

Select ***XDC*** as the *File type* and enter ***led_constraints*** as the *File name*.

Click ***OK***.

(as) Click ***Finish*** to create the file and close the dialogue.

(at) In the ***Sources*** tab, expand the **Constraints** entry and open the newly created XDC file by double-clicking on ***led_constraints.xdc***.

The file will open in the *Workspace*.

(au) Add the following lines to the constraints file. Alternatively, they can be copied from the source file available at **C:\Zynq_Book\sources\led_controller**:

```
set_property PACKAGE_PIN T22          [get_ports {LEDs_out[0]}]
set_property IOSTANDARD LVCMOS33      [get_ports {LEDs_out[0]}]
set_property PACKAGE_PIN T21          [get_ports {LEDs_out[1]}]
set_property IOSTANDARD LVCMOS33      [get_ports {LEDs_out[1]}]
set_property PACKAGE_PIN U22          [get_ports {LEDs_out[2]}]
set_property IOSTANDARD LVCMOS33      [get_ports {LEDs_out[2]}]
set_property PACKAGE_PIN U21          [get_ports {LEDs_out[3]}]
set_property IOSTANDARD LVCMOS33      [get_ports {LEDs_out[3]}]
set_property PACKAGE_PIN V22          [get_ports {LEDs_out[4]}]
set_property IOSTANDARD LVCMOS33      [get_ports {LEDs_out[4]}]
set_property PACKAGE_PIN W22          [get_ports {LEDs_out[5]}]
set_property IOSTANDARD LVCMOS33      [get_ports {LEDs_out[5]}]
set_property PACKAGE_PIN U19          [get_ports {LEDs_out[6]}]
set_property IOSTANDARD LVCMOS33      [get_ports {LEDs_out[6]}]
set_property PACKAGE_PIN U14          [get_ports {LEDs_out[7]}]
set_property IOSTANDARD LVCMOS33      [get_ports {LEDs_out[7]}]
```

This connects each individual bit of the LEDs_out port to a specific pin on the Zynq device. The specific pins are connected to the LEDs on the board.

(av) Save the constraints file.

Our simple design is now complete. We can now generate a bitstream.

(aw) In *Flow Navigator*, select ***Generate Bitstream*** from the *Program and Debug* section.

If a dialogue window appears prompting you to save your design, click ***Save***.

A dialogue window may open requesting that you launch synthesis and implementation before starting the *Generate Bitstream* process. If it does, click ***Yes*** to accept.

The combination of running the synthesis, implementation and bitstream generation processes back-to-back may take a few minutes, depending on the power of your computer system.

(ax) When bitstream generation is complete a dialogue window will open to inform you that the process as been completed.

Select ***Open Implemented Design***, and click ***OK***.

With the bitstream generation complete, the final step in Vivado is to export the design to the SDK, where we will create the software application that will allow the Zynq PS to control the LEDs on the ZedBoard.

(ay) Select ***File > Export > Export Hardware for SDK...*** from the *Menu Bar*.

The *Export Hardware for SDK* dialogue window will open. Ensure that the options to ***Include bitstream*** and ***Launch SDK*** are selected, and Click ***OK***.

The SDK will launch.

(az) Once the SDK has launched, create a new **Application Project** by selecting **File > New > Application Project** from the *Menu Bar*.

In the *New Project* dialogue, enter ***LED_Controller_test*** as the *Project name*.

By default the option to create a new board support package will be selected.

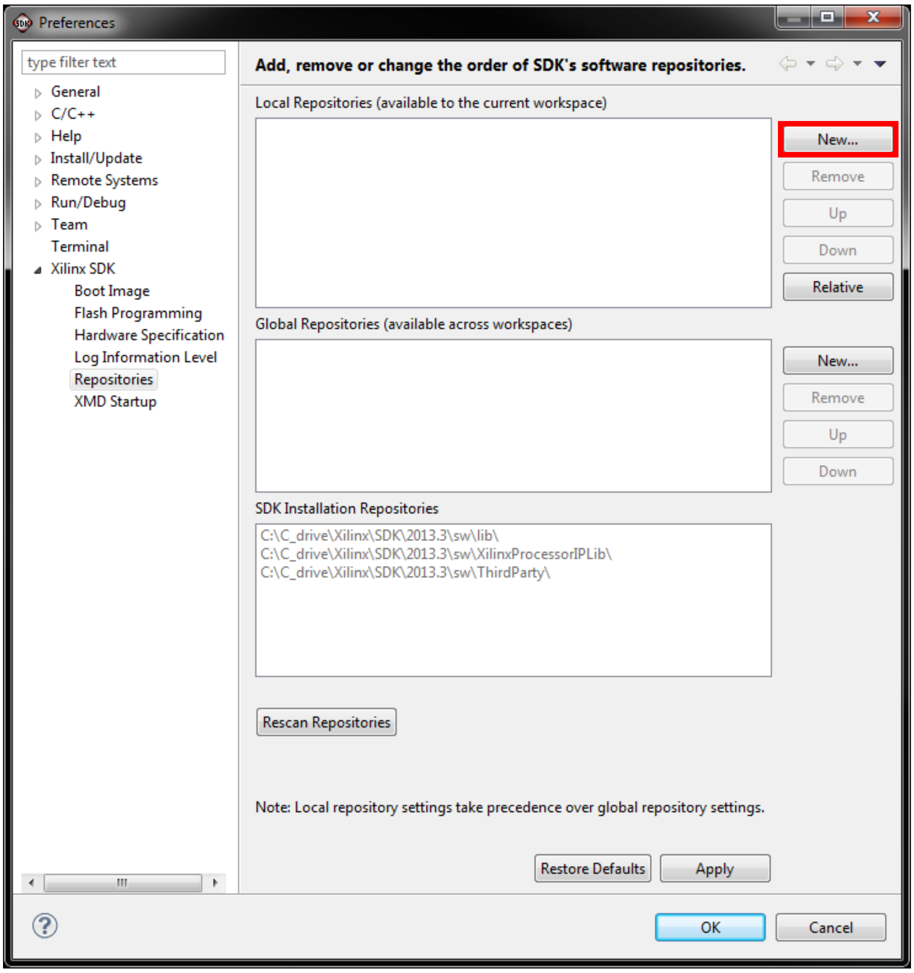
Click ***Next***.

(ba) In the *Templates* dialogue, select ***Empty Application***, and click ***Finish***.

You should recall that when we created the peripheral in the previous stages of this exercise that a set of software driver files were generated. We must now point SDK to those driver files. This is done by adding a new repository to the SDK project.

(bb)Navigate to **Xilinx Tools > Repositories** in the *Menu Bar*.

In the *Repositories Preferences* window, click on ***New***, as shown in Figure 4.10.



Add, remove or change the order of SDK's software repositories.

```
C:\C_drive\Xilinx\SDK\2013.3\sw\lib\
C:\C_drive\Xilinx\SDK\2013.3\sw\XilinxProcessorIPLib\
C:\C_drive\Xilinx\SDK\2013.3\sw\ThirdParty\
```

Note: Local repository settings take precedence over global repository settings.

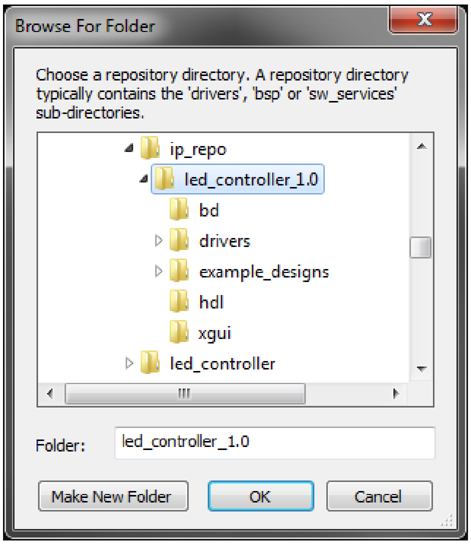
Figure 4.10: SDK Repository Peripherals window

(bc) Browse to the directory **C:\Zynq_Book\ip_repo\led_controller_1.0**, as in Figure 4.11, and click ***OK***.

(bd)Close the Repository Preferences window by clicking ***OK***.

Upon closing the preferences window, SDK will automatically scan the repository and rebuild the project to include the driver files.

We must now assign the newly imported drivers to the LED Controller peripheral.



Choose a repository directory. A repository directory typically contains the 'drivers', 'bsp' or 'sw_services' sub-directories.

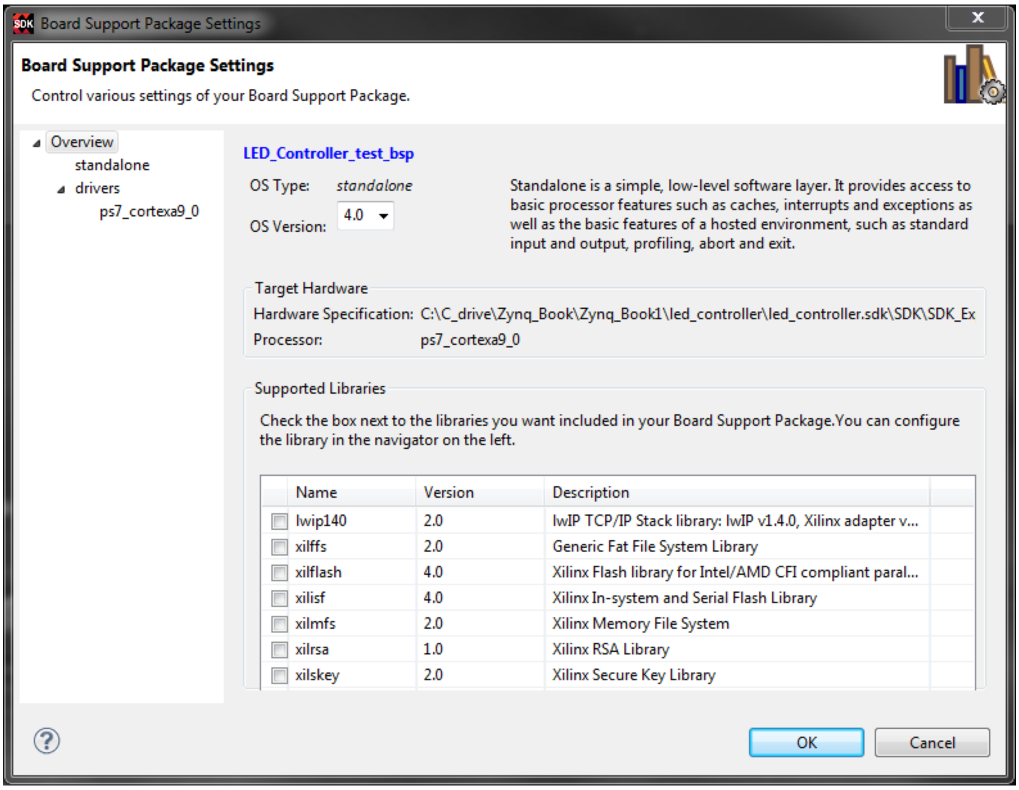
Folder: led_controller_1.0

Figure 4.11: led_controller repository selection

(be)The **system.mss** tab should be open in the Workspace. If it is not, open it by expanding **LED_Controller_test_bsp** in *Project Explorer* and double-clicking on ***system.mss.***

(bf) At the top left of the **system.mss** tab, click ***Modify this BSP's Settings***.

The Board Support Package Settings window will open, as in Figure 4.12.



Board Support Package Settings

Control various settings of your Board Support Package.

LED_Controller_test_bsp

OS Type: *standalone*

OS Version: 4.0

Standalone is a simple, low-level software layer. It provides access to basic processor features such as caches, interrupts and exceptions as well as the basic features of a hosted environment, such as standard input and output, profiling, abort and exit.

Target Hardware

Hardware Specification: C:\C_drive\Zynq_Book\Zynq_Book1\led_controller\led_controller.sdk\SDK\SDK_Ex

Processor: ps7_cortexa9_0

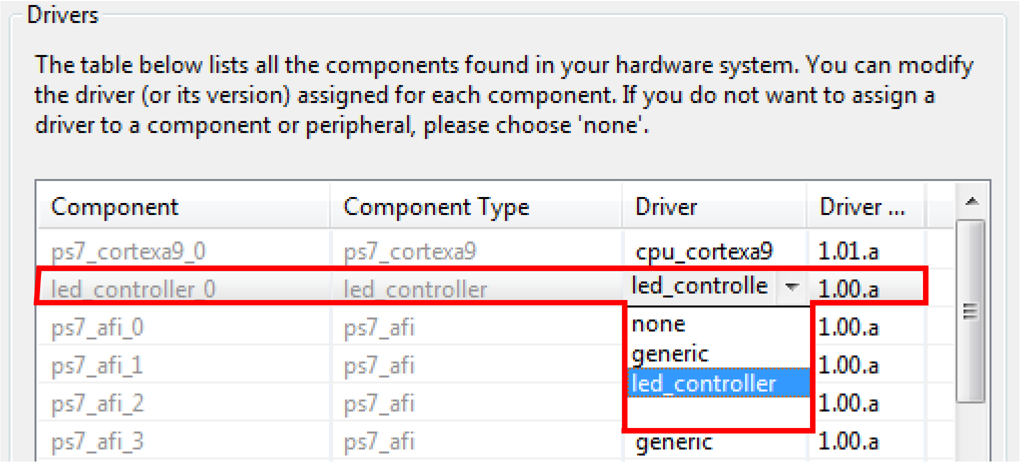
Supported Libraries

Check the box next to the libraries you want included in your Board Support Package.You can configure the library in the navigator on the left.

Name	Version	Description
□ lwip140	2.0	lwIP TCP/IP Stack library: lwIP v1.4.0, Xilinx adapter v...
□ xilffs	2.0	Generic Fat File System Library
□ xilflash	4.0	Xilinx Flash library for Intel/AMD CFI compliant paral...
□ xilisf	4.0	Xilinx In-system and Serial Flash Library
□ xilmfs	2.0	Xilinx Memory File System
□ xilrsa	1.0	Xilinx RSA Library
□ xilskey	2.0	Xilinx Secure Key Library

Figure 4.12: Board Support Package Settings window

(bg)Select ***drivers*** from the left-hand menu. From the list of components in the *Drivers* pane, identify **led_controller_0** and select ***led_controller*** from the drop-down menu in the **Driver** column, as shown in Figure 4.13.



Drivers

The table below lists all the components found in your hardware system. You can modify the driver (or its version) assigned for each component. If you do not want to assign a driver to a component or peripheral, please choose 'none'.

Component	Component Type	Driver	Driver ...
ps7_cortexa9_0	ps7_cortexa9	cpu_cortexa9	1.01.a
led_controller_0	led_controller	led_controlle	1.00.a
ps7_afi_0	ps7_afi		1.00.a
ps7_afi_1	ps7_afi		1.00.a
ps7_afi_2	ps7_afi		1.00.a
ps7_afi_3	ps7_afi	generic	1.00.a

Figure 4.13: LED Controller driver selection

Click ***OK***.

The project will now rebuild.

We can now create a simple C application to control the LEDs. In this instance we will be importing a pre-written source file.

(bh)In *Project Explorer*, right-click on **LED_Controller_Test** and select ***Import***.

In the *Import* window, expand **General** and double-click on ***File System***.

Click ***Browse*** in the top right corner, and navigate to **C:\Zynq_Book\sources\led_controller**.

Click ***OK***.

In the right-hand panel, select ***led_controller_test_tut_4A.c*** and click ***Finish.***

The project will once again rebuild to include the new source file.

Open ***led_controller_test_tut_4A.c*** and examine the functionality.

Before launching the application on the ZedBoard, we must program the Zynq PL and create a new terminal connection.

(bi) From the *Menu Bar*, select **Xilinx Tools > Program FPGA**.

The Bitstream entry should already be populated with the corresponding bitstream that we exported from Vivado earlier.

Click ***Program***, to program the Zynq PL.

NOTE: Once the device has successfully been programmed, the *DONE LED* on the ZedBoard will turn blue.

(bj) Select the ***Terminal*** tab from the *Console* window at the bottom of the workspace, as in Figure 4.14.

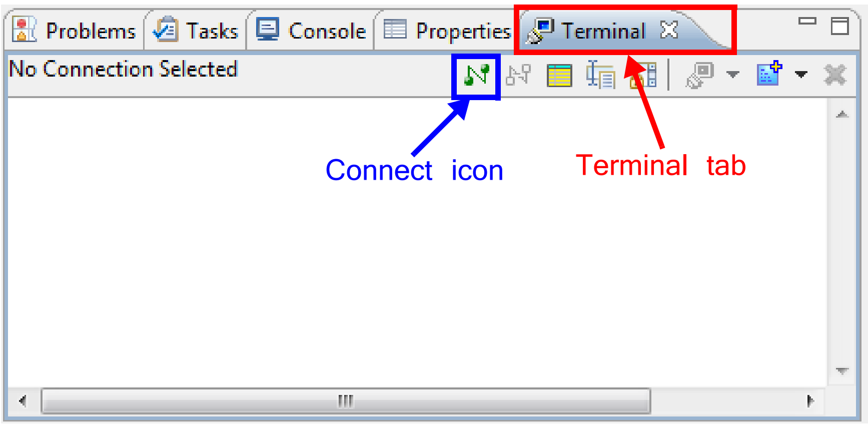


Figure 4.14: SDK Terminal tab

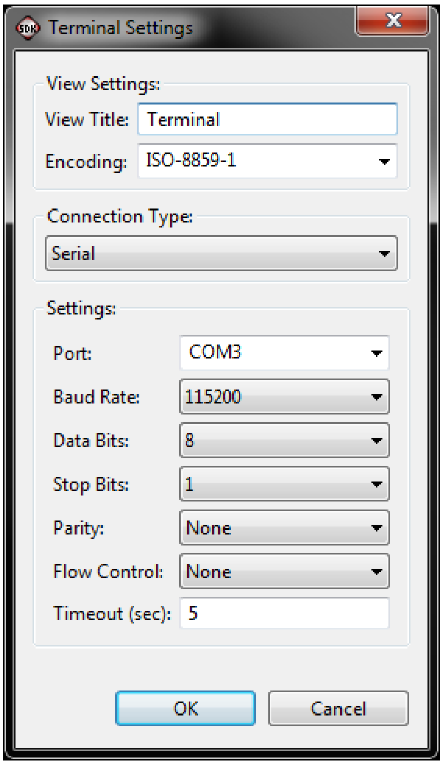
(bk) Click the Connect icon (as highlighted in Figure 4.14).

(bl) The Terminal Settings window will open. Configure the settings as specified in Figure 4.15.

NOTE: The value of the *Port* entry will vary depending on which the USB UART cable is connected to.

In order to determine this value on a Windows system, open the Device Manager and identify the COM port.

(bm) Click OK to initiate the new Terminal connection.



View Settings:
View Title: Terminal
Encoding: ISO-8859-1

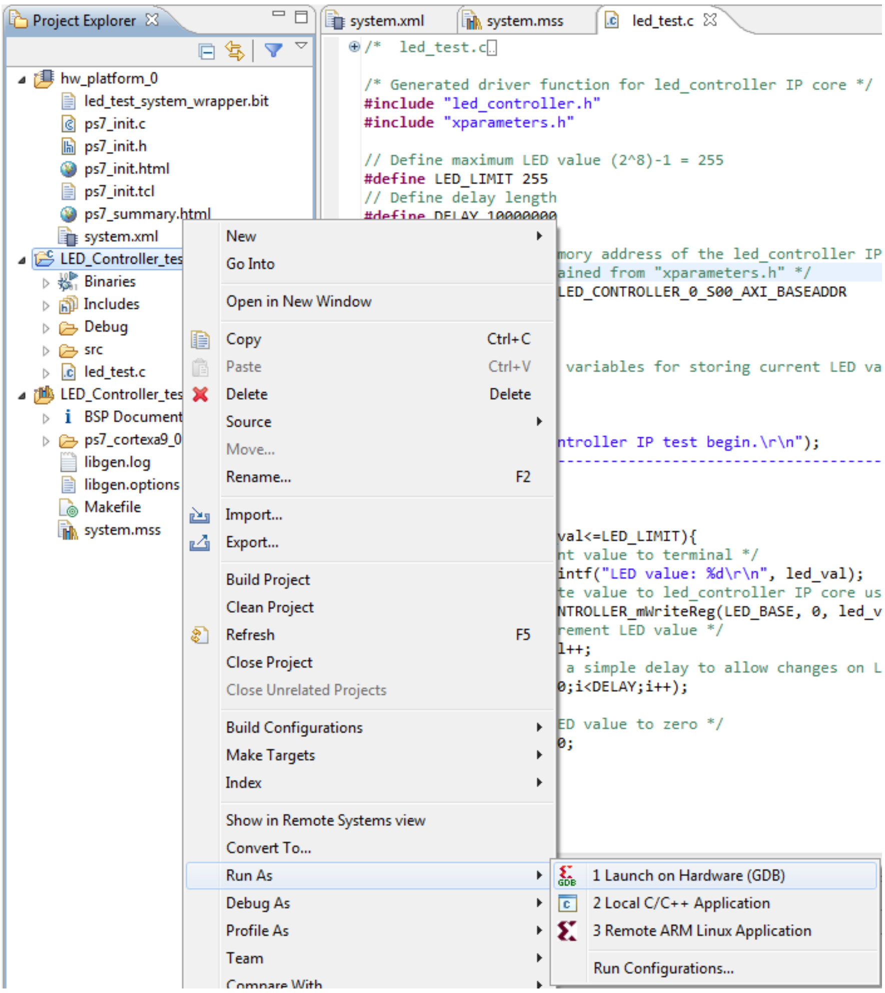
Connection Type:
Serial

Settings:
Port: COM3
Baud Rate: 115200
Data Bits: 8
Stop Bits: 1
Parity: None
Flow Control: None
Timeout (sec): 5

Figure 4.15: Terminal Settings

Now that the Zynq PL is programmed, and the Terminal connection has been created, we can program the Zynq PS with our software application.

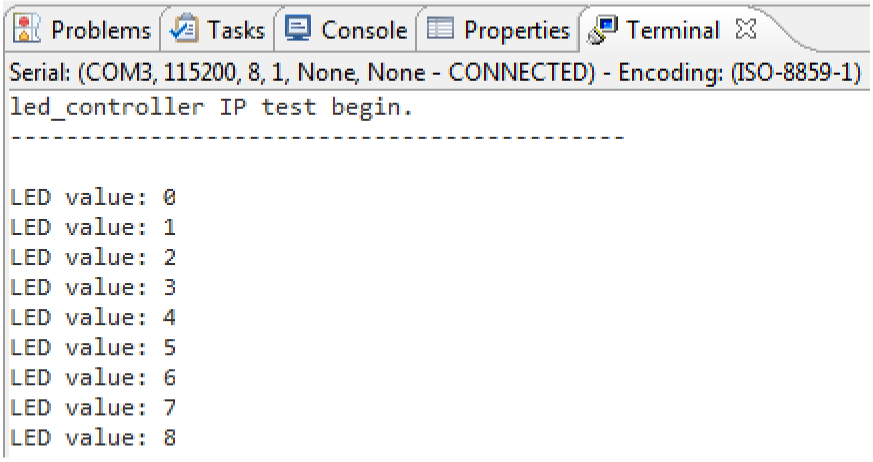
(bn) In Project Explorer, right-click on **LED_Controller_test** and select ***Run As > Launch on Hardware (GDB)***, as shown in Figure 4.16.



```
/*  led_test.c

/* Generated driver function for led_controller IP core */
#include "led_controller.h"
#include "xparameters.h"

// Define maximum LED value (2^8)-1 = 255
#define LED_LIMIT 255
// Define delay length
#define DELAY 10000000

mory address of the led_controller IP
ained from "xparameters.h" */
LED_CONTROLLER_0_S00_AXI_BASEADDR

 variables for storing current LED va

ntroller IP test begin.\r\n");
----------------------------------------

val<=LED_LIMIT){
nt value to terminal */
intf("LED value: %d\r\n", led_val);
te value to led_controller IP core us
NTROLLER_mWriteReg(LED_BASE, 0, led_v
rement LED value */
l++;
 a simple delay to allow changes on L
0;i<DELAY;i++);

ED value to zero */
0;
```

Figure 4.16: Run Application on hardware

(bo)Switch to the Terminal tab of the Console window, and confirm that the LED value is being output, as in Figure 4.17.



```
Serial: (COM3, 115200, 8, 1, None, None - CONNECTED) - Encoding: (ISO-8859-1)
led_controller IP test begin.
--------------------------------------------

LED value: 0
LED value: 1
LED value: 2
LED value: 3
LED value: 4
LED value: 5
LED value: 6
LED value: 7
LED value: 8
```

Figure 4.17: Terminal tab displaying LED values

You should also see the LEDs on the ZedBoard displaying the corresponding LED values.

This concludes this exercise on designing Zynq IP in HDL. You should now be familiar with:

- Creating AXI interface templates with the Create and Package IP Wizard.
- Adding functionality to HDL IP peripherals in Vivado and IP Packager.
- How to connect packaged IP to a Zynq Processing System in IP Integrator.
- Creating software applications to control the HDL IP using the generated C software drivers, and executing them on the ZedBoard.

Exercise 4B Creating IP in MathWorks HDL Coder

In this exercise, we will be creating an IP core which will perform the function of an LMS noise cancellation filter. Mathworks HDL Coder will be used to transform an existing Simulink block-based model into an RTL description which will be packaged for use in the Vivado IP Catalog. The...

We will start by opening the Simulink model in MatLab.

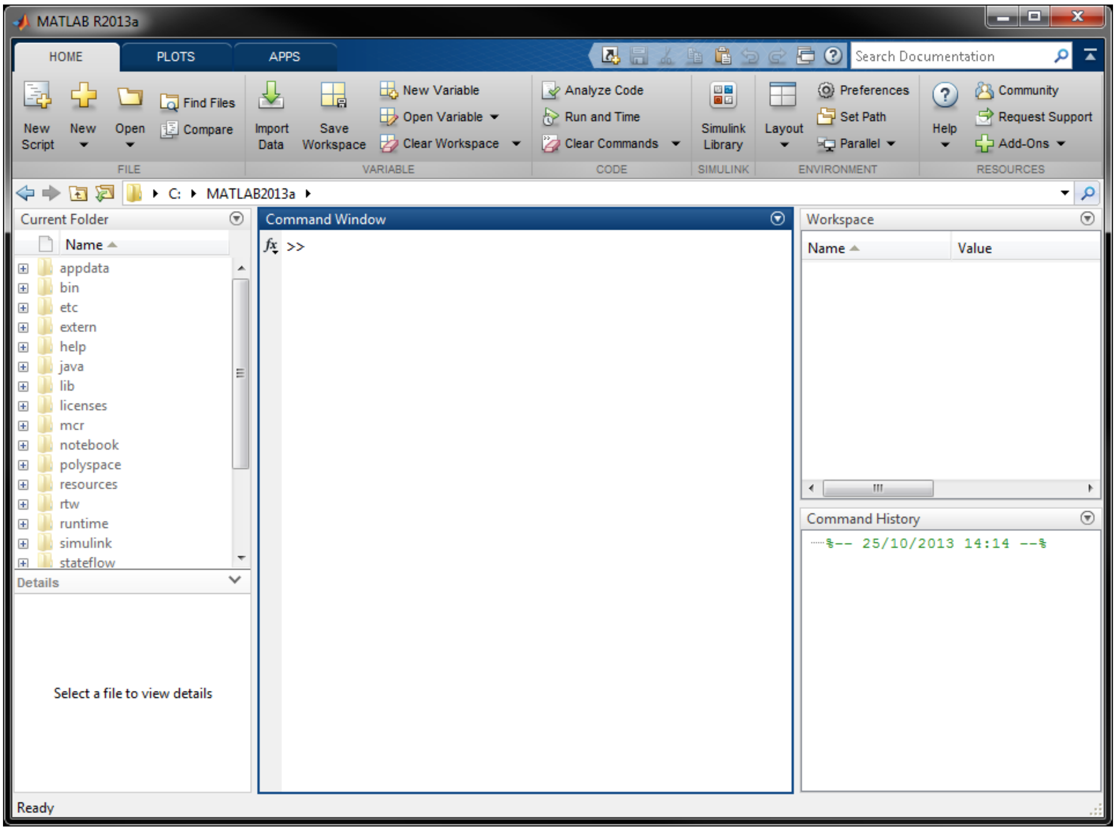
Before starting this exercise, you are required to copy some source files into a new working directory.

(a) In Windows Explorer, navigate to **C:\Zynq_Book\sources\hdl_coder_lms** and copy the contents of the directory to a new directory called **C:\Zynq_Book\hdl_coder_lms**.

(b) Launch MatLab by navigating to **Start > All Programs > MATLAB > R2013a > MATLAB R2013a**

Note: This workbook uses version R2013a of MatLab. If you have a different MatLab version you will need to replace R2013a with your own version (i.e. R2012a/R2012b).

MatLab will open and you will see the main workspace, as shown in Figure 4.18(or a variation thereof).



>>

%-- 25/10/2013 14:14 --%

Select a file to view details

Figure 4.18: MatLab workspace environment

(c) Enter **C:\Zynq_Book\hdl_coder_lms** as the working directory, as highlighted in Figure 4.19.

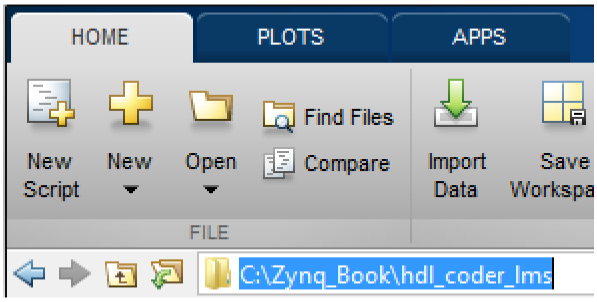


Figure 4.19: Setting the MatLab working directory

In the *Current Folder* pane, you should also see four files:

- **original_speech.wav** — A short audio clip of speech.
- **setup.m** — Performs setup commands to import the audio samples into the MatLab workspace and set the system sample rate accordingly.
- **lms.slx** — A simulink model which implements and LMS noise cancellation process.
- **playback.m** —Can be used to verify the LMS filtering process via audio playback of the various stages.

The setup commands in setup.m are automatically called when the Simulink simulation is initialised.

(d) Open the LMS Simulink model by double-clicking on ***lms.slx*** in *Current Folder* pane. The model should open and you should see the LMS system, as shown in Figure 4.20.

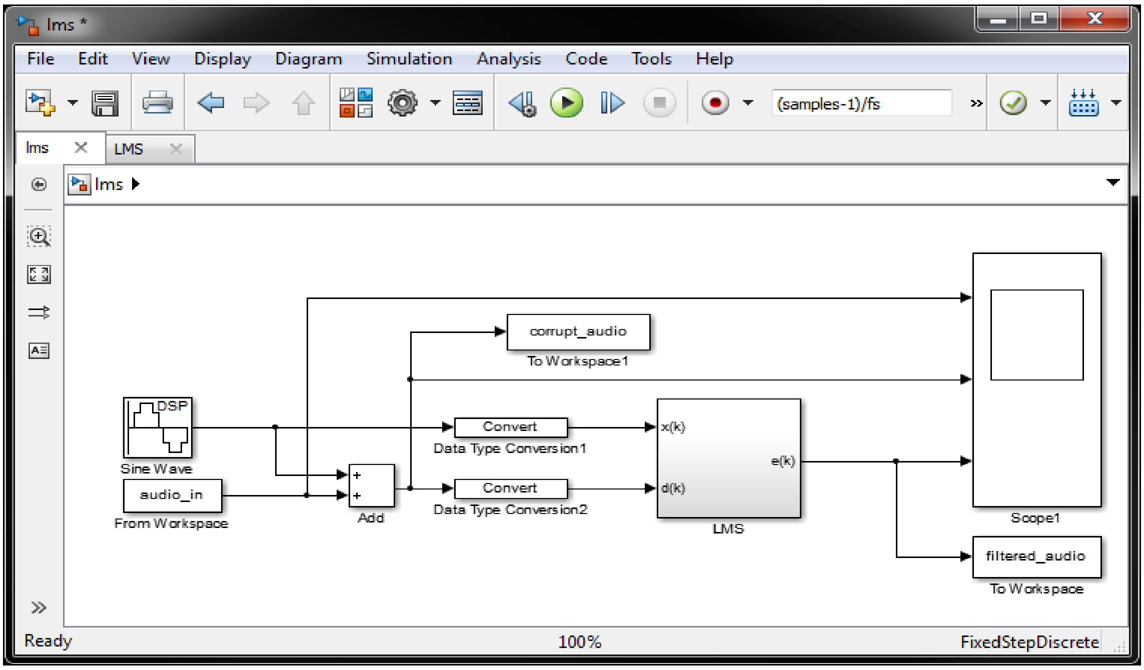


Figure 4.20: LMS model in Simulink

The model features two sources:

- a ***Sine Wave*** block which generates tonal noise.
- A ***From Workspace*** block which imports the audio samples from the MatLabe Workspace.

The tonal noise is then added to the audio samples to create a corrupted audio signal.

In order to generate HDL code for the Simulink LMS model using HDL Coder, the inputs to the system must be in fixed-point numerical format. Two ***Data Type Conversion*** blocks are used to convert the corrupt audio signal and the tonal noise signal to fixed-point format. The fixed-point signals are then input to an LMS subsystem, which we will explore in the next step.
At the output of the LMS subsystem, the error signal, **e(k)**, is input to a scope along with the corrupt audio and tonal noise inputs, for visual inspection of the signals. Two ***To Workspace*** blocks are also present to allow the LMS output and the corrupt audio signals to be output to the MatLab workspace for audio playback.

(e) Drill down into the LMS subsystem block by double-clicking on it. You will see the system in Figure 4.21.

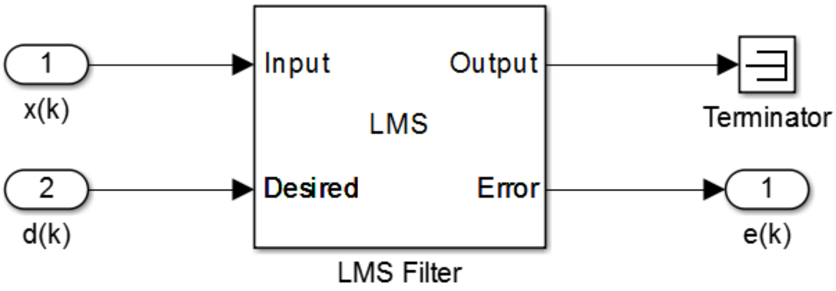


Figure 4.21: LMS subsystem

It features a single ***LMS Filter*** block. As we are not interested in the ***Output*** signal, it is unconnected.

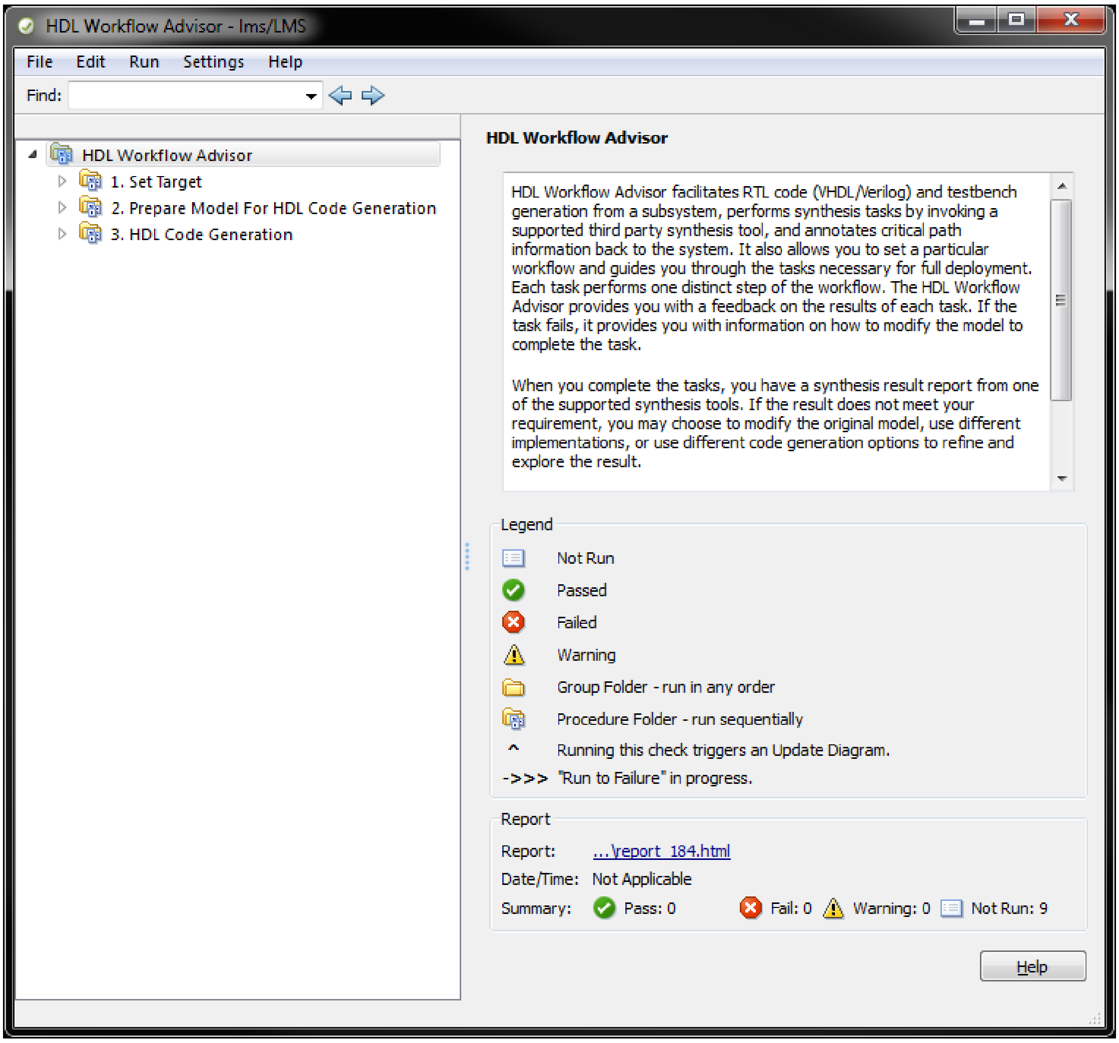
(f) Open the *LMS Filter Block Parameters* by double-clicking on the ***LMS Filter*** block.
Take a moment to explore the parameters. You should be able to determine that there are 16 ***adaptive filter coefficients*** and a ***step size*** of 0.1.

(g) Close the Parameters window, and return to the main Simulink model by clicking the ***Up To Parent*** button, ⇧ .

We will be generating HDL code for the LMS subsystem only.

Right-click on the LMS subsystem and select **HDL Code > HDL Workflow Advisor**.

The HDL Workflow Advisor window will open, as in Figure 4.22.


HDL Workflow Advisor - lms/LMS
File Edit Run Settings Help
Find:
HDL Workflow Advisor
1. Set Target
2. Prepare Model For HDL Code Generation
3. HDL Code Generation
Help

HDL Workflow Advisor

HDL Workflow Advisor facilitates RTL code (VHDL/Verilog) and testbench generation from a subsystem, performs synthesis tasks by invoking a supported third party synthesis tool, and annotates critical path information back to the system. It also allows you to set a particular workflow and guides you through the tasks necessary for full deployment. Each task performs one distinct step of the workflow. The HDL Workflow Advisor provides you with a feedback on the results of each task. If the task fails, it provides you with information on how to modify the model to complete the task.

When you complete the tasks, you have a synthesis result report from one of the supported synthesis tools. If the result does not meet your requirement, you may choose to modify the original model, use different implementations, or use different code generation options to refine and explore the result.

Legend

☰	Not Run
✓	Passed
✗	Failed
⚠	Warning
📁	Group Folder - run in any order
📁	Procedure Folder - run sequentially
^	Running this check triggers an Update Diagram.
->>>	"Run to Failure" in progress.

Report

Report: ...\report_184.html

Date/Time: Not Applicable

Summary: ✓ Pass: 0 ✗ Fail: 0 ⚠ Warning: 0 ☰ Not Run: 9

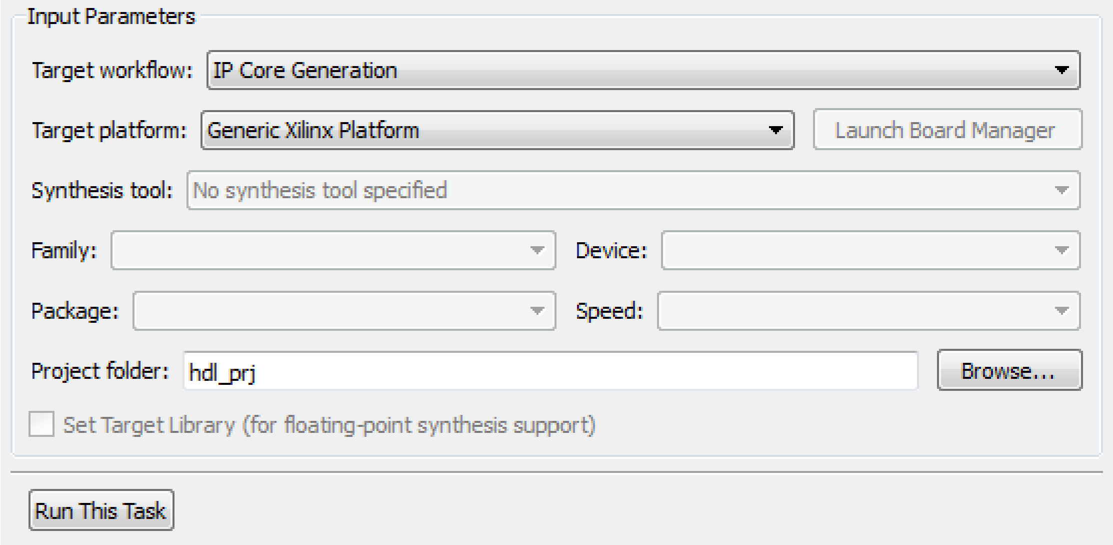
Figure 4.22: HDL Workflow Advisor window

The HDL Workflow Advisor guides you through the steps required to generate RTL code for your design.

(h) In the left-hand panel, expand ***Set Target*** and select ***1.1. Set Target Device and Synthesis Tool***.

Here we specify the output format of the RTL and the target platform.

(i) In the *Input Parameters* pane, select ***IP Core Generation*** as the *Target workflow*, and ***Generic Xilinx Platform*** as the *Target platform*, as shown in Figure 4.23.



Input Parameters

Target workflow: IP Core Generation

Target platform: Generic Xilinx Platform

Synthesis tool: No synthesis tool specified

Family:

Device:

Package:

Speed:

Project folder: hdl_prj

□ Set Target Library (for floating-point synthesis support)

Figure 4.23: HDL Workflow Advisor Input Parameters

(j) Click ***Run This Task*** to apply the settings.

(k) Select ***Set Target Interface*** from the left hand panel.

Here we specify the target interface for the HDL code generation.

In the Input Parameters pane, select ***Coprocessing - blocking*** as the *Processor/FPGA synchronization*. This will automatically infer an **AXI4-Lite** interface for all ports in the design, and specify a memory address for each.

(l) Click ***Run This Task*** to apply the settings.

(m) Expand ***Prepare Model for HDL Code Generation*** in the left hand panel, and select ***Check Global Settings***.

Here, model-level settings will be checked to verify if the model is ready for HDL code generation.

(n) Click ***Run This Task*** to check the model-level settings.

If this step fails, click ***Modify All*** to allow *HDL Workflow Advisor* to modify the settings.

This step should now pass, and you will be presented with a table of the results.

The next few steps are all checks, and can be performed in batch.

(o) Right-click on *Check Sample Times* in the left hand pane, and select ***Run to Selected Task***.

This will perform the checks one after another to prevent you from running each individually.

All check should pass.

The final steps involve specifying basic settings about the RTL code, such as what language to use (VHDL/Verilog), and what code generation reports to generate. Finally the HDL code will be generated.

(p) Expand *HDL Code Generation* in the left hand pane, and further expand ***Set Code Generation Options***.
Click on ***Set Basic Options***.

(q) Select ***VHDL*** as the **Language** in the *Target* pane.
You can also select any of the *Code generation reports* that you would like.

(r) Select ***Set Advanced Options*** in the left hand panel.
Here you can specify more advanced options for the HDL code.
We will be leaving the values as default, but you may wish to explore the settings for future use.

(s) Right-click on *Set Advanced Options*, and select ***Run to Selected Task*** to apply the settings.

(t) Finally, select *Generate RTL Code and IP Core* from the left hand panel.
This is the step which will finally generate the HDL code for out LMS IP Core.
Set the *IP core name* as ***lms_pcore*** and click ***Run This Task***.

Once HDL Coder has finished generating the HDL code, the Code Generation Report window will open. This provides a summary of the HDL Coder results and provides further information on the target interface and clocking.

The final stage of creating our LMS IP core is to package it with IP Packager so that we can use it in IP Integrator designs. To do this we will need to create a new Vivado project.

(u) Launch Vivado and create a new project called **lms_packaging** at the following location: **C:\Zynq_Book\hdl_coder_lms**, ensuring that the option to ***create a project subdirectory*** is selected. Also select the ***VHDL*** as the *target language*, and the ***ZedBoard*** as the *default part*.

For more detail on the process of creating a new Vivado project, refer to **Step (a)** of **Exercise 4A**.

(v) When the project has been created and opened, select ***Tools*** > ***Create and Package IP*** from the *menu bar*, and Click ***Next***.

(w) Select the option to ***Package a specified directory***, and click ***Next***.

(x) Enter **C:/Zynq_Book/hdl_coder_lms/hdl_prj/ipcore/lms_pcore_v1_00_a** as the *IP Location*.

(y) Click ***Next*** to move to the *Edit in IP Packager Project Name* dialogue, and click Next to accept the default *Project Name* and *Project Location*.

(z) At the *Summary* window, and click ***Finish*** to launch **IP Packager**.

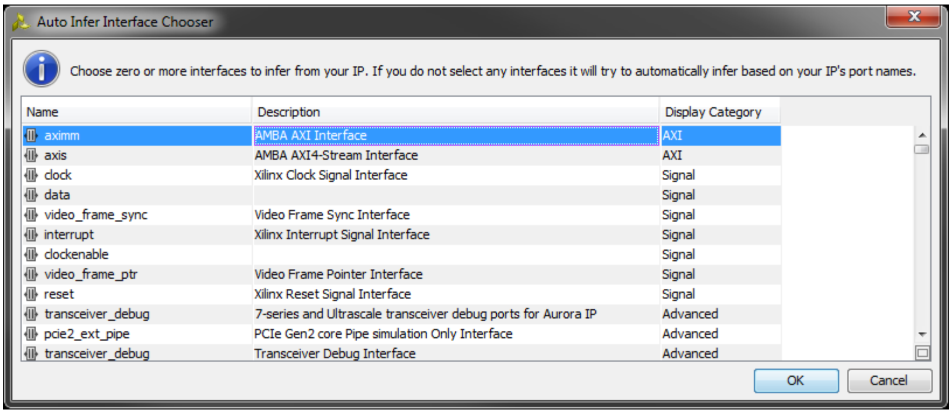
(aa) In the left hand panel of the *IP Packager* window, select ***IP Ports and Interfaces***.

The *IP Interfaces* panel will open, and you should see that **IP Packager** has identified the individual AXI ports, but has not inferred an AXI interface.

To infer an AXI interface:

(ab) Right-click on a blank section of the *IP Ports and Interfaces* pane, and select ***Auto Infer Interface ...***

(ac) The *Auto Infer Interface Chooser* window will open:



Choose zero or more interfaces to infer from your IP. If you do not select any interfaces it will try to automatically infer based on your IP's port names.

Name	Description	Display Category
aximm	AMBA AXI Interface	AXI
axis	AMBA AXI4-Stream Interface	AXI
clock	Xilinx Clock Signal Interface	Signal
data		Signal
video_frame_sync	Video Frame Sync Interface	Signal
interrupt	Xilinx Interrupt Signal Interface	Signal
clockenable		Signal
video_frame_ptr	Video Frame Pointer Interface	Signal
reset	Xilinx Reset Signal Interface	Signal
transceiver_debug	7-series and Ultrascale transceiver debug ports for Aurora IP	Advanced
pcie2_ext_pipe	PCIe Gen2 core Pipe simulation Only Interface	Advanced
transceiver_debug	Transceiver Debug Interface	Advanced

Select ***aximm*** from the list, as shown, and click ***OK***.

The individual AXI ports in our design will be mapped to an **AXILite** interface.

(ad) Select *IP **Addressing and Memory*** from the left hand panel. Here, IP Packager has incorrectly specified an address **Range** of **4294967296**. Click on the **Range**, and change the value to ***32***.

(ae) Finally, select ***Review and Package*** from the left hand menu.

Review the information provided, and click ***Package IP***.

This completes the generation of an LMS component from Mathworks HDL Coder. You should now be familiar with:

- Using the Simulink block-based design environment for the design and simulation of IP.
- Using the HDL Workflow Advisor to guide you through the steps of generating RTL code

and IP cores for existing Simulink designs.

- Packaging HDL Coder generated IP blocks in IP Packager for use in Vivado IP Integrator designs.

Exercise 4C Creating IP in Vivado HLS

In this final exercise, we will creating an IP core that will implement the functionality of an NCO. The tool that we will be using is Vivado HLS, and we shall explore some of the features which allow us to specify arbitrary precision fixed-point data types, as well as the directives required to export IP with an AXI-Lite slave interface, to allow the IP core to interface with the Zynq processor.

We will start by creating a new project in Vivado HLS.

(a) Launch Vivado HLS by double-clicking on the Vivado HLS desktop icon: Vivado HLS 2014.1, or by navigating to **Start > All Programs > Xilinx Design Tools > Vivado 2014.1 > Vivado HLS > Vivado HLS 2014.1**

(b) When Vivado HLS loads, you will be presented with the *Getting Started* screen, as in Figure 4.24.

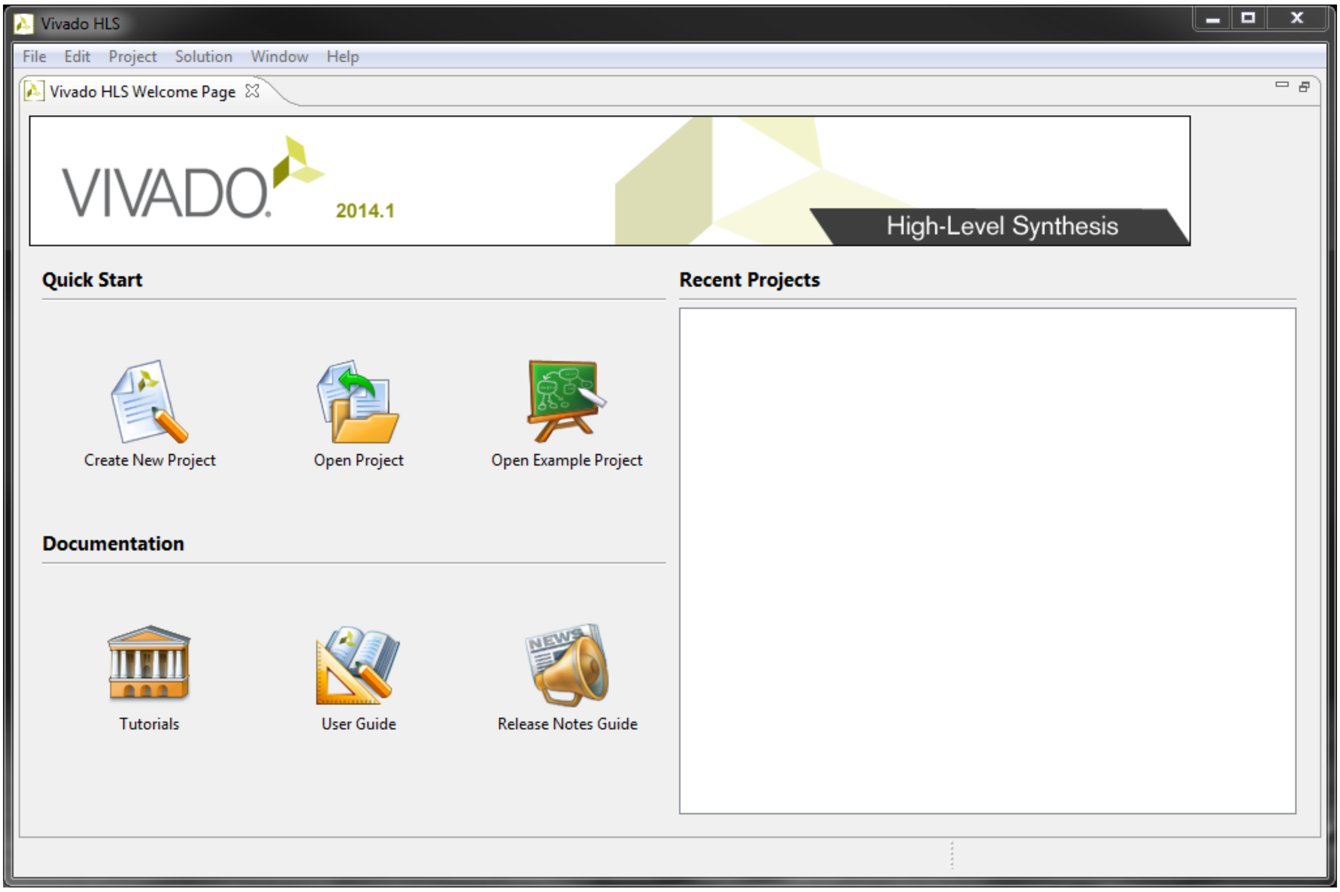
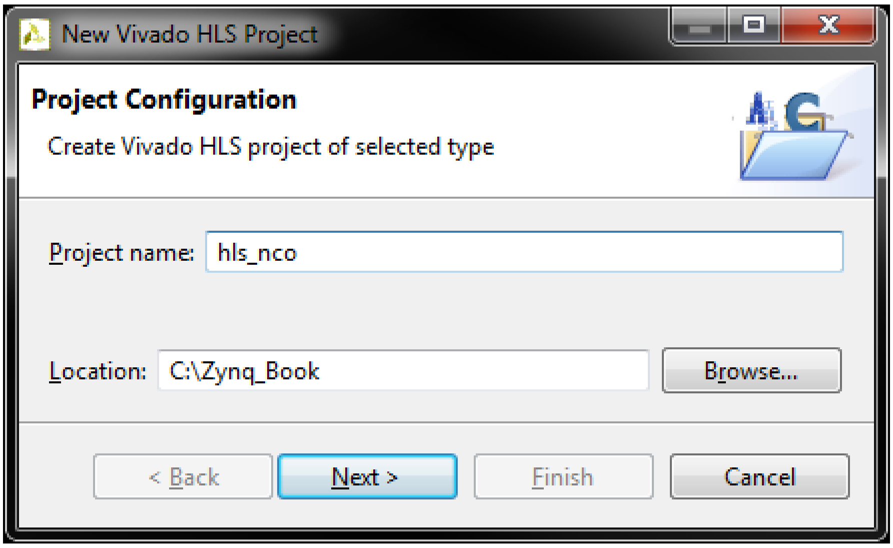


Figure 4.24: Vivado HLS Getting Started screen

(c) Select the option to ***Create New Project*** and the *New Vivado HLS Project Wizard* will open, as in Figure 4.25.



Project name: hls_nco

Location: C:\Zynq_Book

Figure 4.25: Vivado HLS New Project Wizard

Enter ***hls_nco*** as the *Project name*, and ***C:\Zynq_Book*** as *Location*.

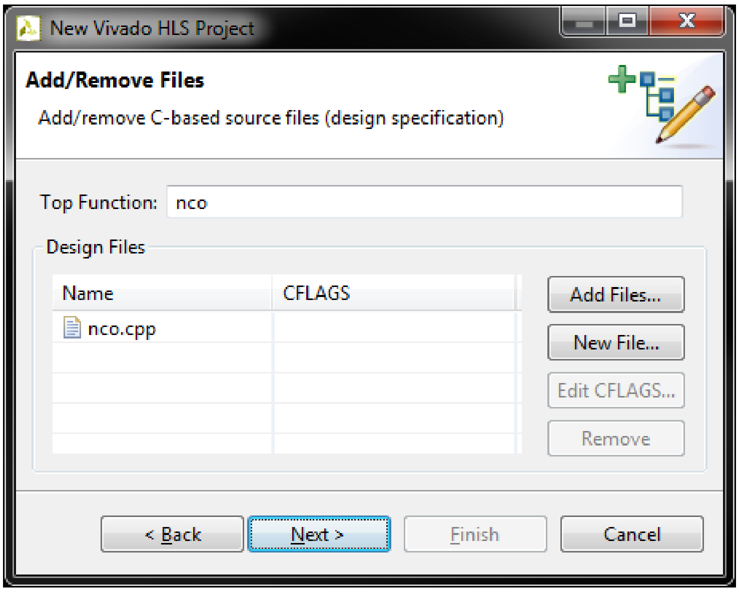
Ensure that the options match those in Figure 4.25, and click ***Next***.

(d) The *Add/Remove Files* dialogue will appear. This is where existing C-based source files can be added to the project, or new files created.

Enter ***nco*** as the *Top Function* and click ***Add Files...***

Navigate to ***C:\Zynq_Book\sources\hls_nco*** and select ***nco.cpp***. Click ***Open***.

The dialogue should now resemble Figure 4.26.



Top Function: nco

Name	CFLAGS
nco.cpp	

Figure 4.26: Vivado HLS New Project Wizard (Add/Remove Files)

Click ***Next***.

(e) A second *Add/Remove Files* dialogue will appear. This is where C-based testbench files can be added to the project, or new files created.

Click ***Add Files...*** and navigate to ***C:\Zynq_Book\sources\hls_nco***. Select ***nco_tb.cpp*** and click ***Open*** to add the testbench file to the project.

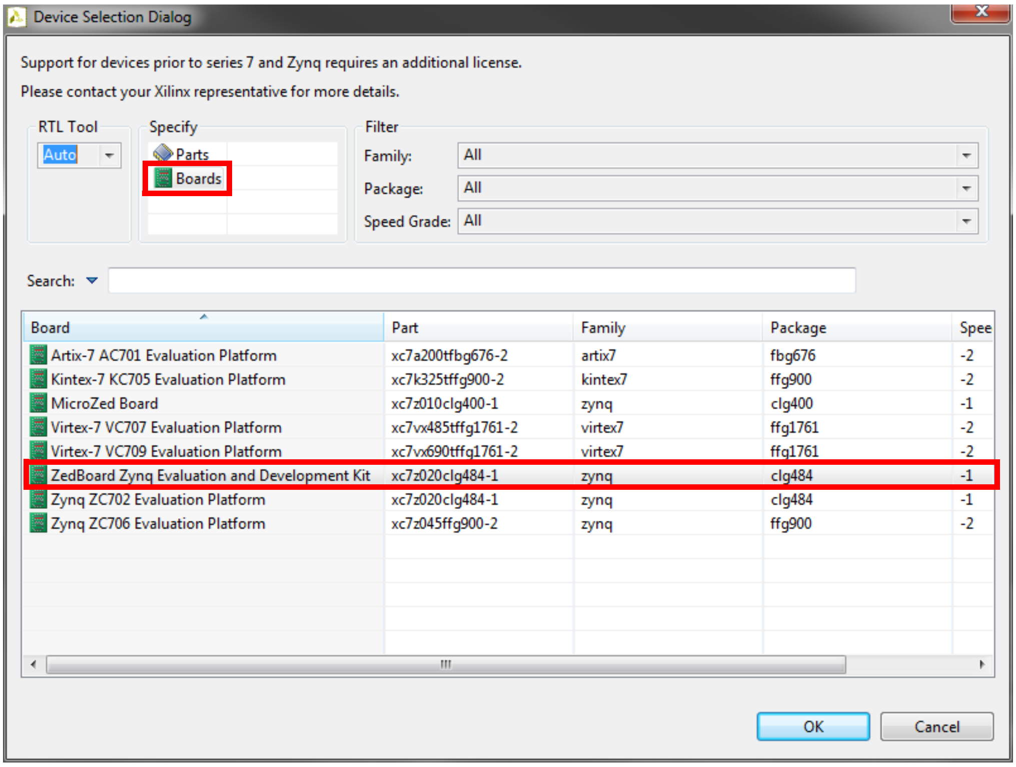
Click ***Next***.

(f) The *Solution Configuration* dialogue will open. Here we will be selecting the part which we will be targeting. In this particular case we will be targeting the Zynq-7020 on the ZedBoard, but if you have a different development board, it is easy to choose your particular board instead.

Click the selection button, ..., in the *Part Selection* pane.

The *Device Selection Dialog* will open.

As we are targeting the ZedBoard, select ***Boards*** in the *Specify* pane and choose ***ZedBoard Zynq Evaluation and Development Kit***, as in Figure 4.27.



Device Selection Dialog

Support for devices prior to series 7 and Zynq requires an additional license.
Please contact your Xilinx representative for more details.

RTL Tool: Auto

Specify: Parts, Boards

Filter
Family: All
Package: All
Speed Grade: All

Search:

Board	Part	Family	Package	Spee
Artix-7 AC701 Evaluation Platform	xc7a200tfbg676-2	artix7	fbg676	-2
Kintex-7 KC705 Evaluation Platform	xc7k325tffg900-2	kintex7	ffg900	-2
MicroZed Board	xc7z010clg400-1	zynq	clg400	-1
Virtex-7 VC707 Evaluation Platform	xc7vx485tffg1761-2	virtex7	ffg1761	-2
Virtex-7 VC709 Evaluation Platform	xc7vx690tffg1761-2	virtex7	ffg1761	-2
ZedBoard Zynq Evaluation and Development Kit	xc7z020clg484-1	zynq	clg484	-1
Zynq ZC702 Evaluation Platform	xc7z020clg484-1	zynq	clg484	-1
Zynq ZC706 Evaluation Platform	xc7z045ffg900-2	zynq	ffg900	-2

Figure 4.27: Vivado HLS Device Selection Dialog

Click ***OK*** to close the dialogue and return to the *New Project Wizard*.

(g) Click ***Finish*** to close the *New Project Wizard* and to create the project.

The Vivado HLS workspace will open.

(h) In the *Explorer* panel, expand the ***Source*** and ***Test Bench*** headings. You should see the source files that we specified in the *New Project Wizard*, as in Figure 4.28.

(i) Open ***nco.cpp*** and examine the contents of the file.

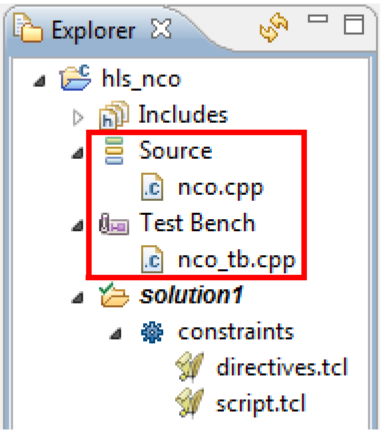


Figure 4.28: Vivado HLS Explorer panel

You should notice the inclusion of the header file ***ap_fixed.h*** on the first line. This is the arbitrary precision fixed-point library which adds support for the use of fixed-point data types in C++.

The next thing that you should see is the global declaration of a 2^{12} = 4096 value array:

```
const ap_fixed<16,2> sine_lut[4096] ...
```

This forms the sinewave lookup table. It is defined as an array of type `ap_fixed<16,2>`, which means that all values are16-bit, signed fixed-point (2 integer bits and 14 fractions bits). Further information on fixed-point data types in Vivado HLS can be found in ***Chapter15 - Vivado HLS: A Closer Look*** of the **Zynq Book.**

The functionality of the NCO is contained in the function:

```
void nco (ap_fixed<16,2> *sine_sample, ap_ufixed<16,12> step_size)
```

It takes two arguments:

- **`*sine_sample`** — A pointer to a 16-bit, signed fixed-point variable which forms the output sample of the NCO.
- **`step_size`** — 16-bit, unsigned fixed-point value which provides the step size input for the NCO.

(j) Explore the **nco** function, ensuring that you understand it all.

Open ***nco_tb.cpp***. This is the testbench file which is used to ensure that the functionality of the C-based source file is correct.

Explore the code in the file, ensuring that you understand the functionality.

This is a simple file which opens a text file in write-mode, to allow you to output the sinusoidal samples. It then calls the **nco** function from within a *for-loop* in order to generate a finite number of sinusoidal samples, which are then output to the text file.

The text file is formatted in a way which easily allows you to import the samples into MatLab for analysis.

Note: The location of the output file is determined by the following line in the testbench file:

```
char *outfile = "E:\\nco_sine.m";
```

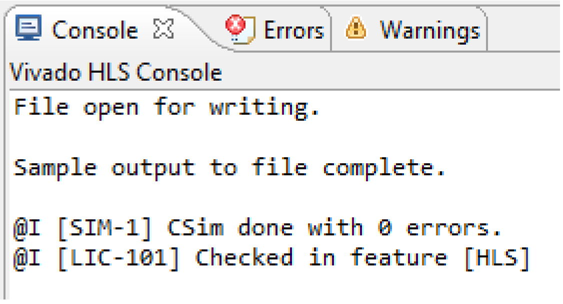
You should change the output file path accordingly to a location on your local machine.

We will now run a C simulation.

(k) Click the ***Run C Simulation*** button, ▶, from the *Main Toolbar*.

The *C Simulation Dialog* window will open. Click ***OK*** to run the simulation with the default settings.

The C simulation will run, and you should see the following output in the Console window:



```
File open for writing.

Sample output to file complete.

@I [SIM-1] CSim done with 0 errors.
@I [LIC-101] Checked in feature [HLS]
```

The sine wave samples that were generated by the NCO will have been output to the location which you specified in the previous step.

If you wish, you can import the sine wave samples into MATLAB using the output file to verify that the NCO has correctly generated a sine wave. This should be done at your own discretion, and will not be covered in this exercise.

The process of HLS has been covered previously in **The Zynq Book Tutorial: Designing With Vivado High Level Synthesis**, and you should refer to it for more detailed information on the various steps involved. For the purposes of this exercise, it is presumed that you have a reasonable knowledge of the Vivado HLS tool.

As we want to allow our NCO peripheral to be controlled by a Zynq PS, it is necessary to give it an AXI interface. This is done in Vivado HLS through the use of directives.

(l) Ensure that **nco.cpp** is the active source file, and select the ***Directive*** tab in the right-hand side of the Vivado HLS workspace, as shown in Figure 4.29.

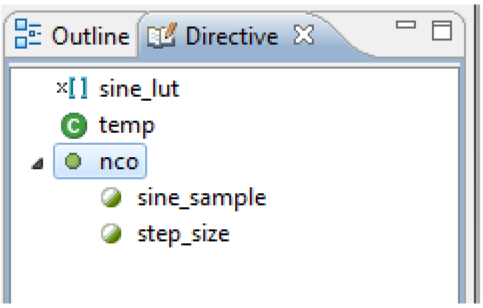


Figure 4.29: Vivado HLS Directive tab

First, we will define the interface of the NCO as an **AXI-Lite slave**.

(m) Right-click on ***nco*** in the *Directive* tab, and select ***Insert Directive***.

As the *Directive Type*, select ***RESOURCE***.

and select ***AXI4LiteS [adapter]*** as the *core*, from the pop-up list.

Leave *Destination* as ***Directive File***, and click ***OK***.

We will now define the NCO as having a **ap_ctrl_none** interface, to remove unneeded control signals.

(n) Right-click on ***nco*** in the *Directive* tab, and select ***Insert Directive***.

As the *Directive Type*, select ***INTERFACE***.

and select ***ap_ctrl_none*** as the *mode*, from the drop-down list.

Leave *Destination* as ***Directive File***, and click ***OK***.

Finally, we will be defining the two variables, **sine_sample** and **step_size**, as ports on the **AXI-LIte slave** interface.

(o) Right-click on ***sine_sample*** in the *Directive* tab, and select ***Insert Directive***.

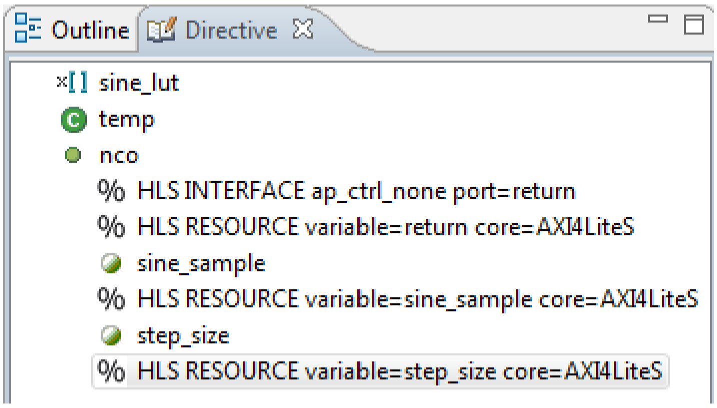
As the *Directive Type*, select ***RESOURCE***.

and select ***AXI4LiteS [adapter]*** as the *core*, from the pop-up list.

Leave *Destination* as ***Directive File***, and click ***OK***.

(p) Repeat the previous step for the **step_size** variable in the *Directive* tab.

On completion, the Directive tab should look like Figure 4.30.



sine_lut
temp
nco
% HLS INTERFACE ap_ctrl_none port=return
% HLS RESOURCE variable=return core=AXI4LiteS
sine_sample
% HLS RESOURCE variable=sine_sample core=AXI4LiteS
step_size
% HLS RESOURCE variable=step_size core=AXI4LiteS

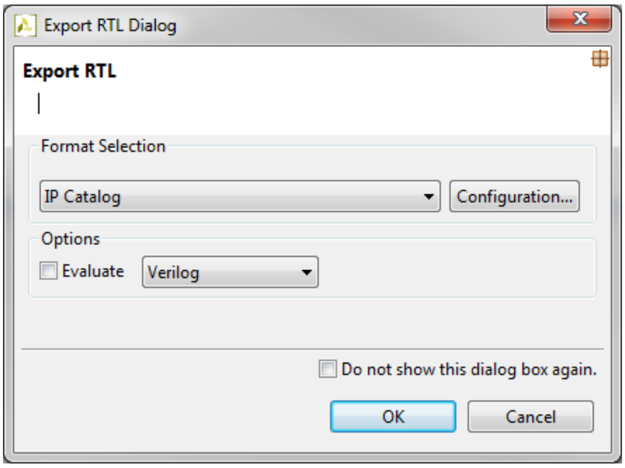
Figure 4.30: Complete Directive tab

We can now run HLS.

(q) Run C Synthesis by clicking the ***Run C Synthesis*** button, ▶, from the *Main Toolbar*.

(r) Click the Export RTL button, ⊞, from the Main Toolbar.

The Export RTL Dialog window will open, as shown in Figure 4.31.



IP Catalog

□ Evaluate Verilog

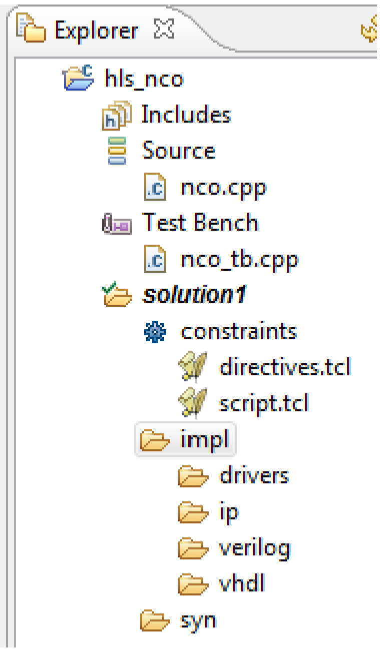
□ Do not show this dialog box again.

Figure 4.31: Vivado HLS Export RTL Dialog Window

(s) Select IP Catalog as the Format Selection.

If you choose, you can edit the *IP Identification* data by clicking the ***Configuration*** button.

(t) Click ***OK*** to generate the IP core.



When RTL Generation has completed, a directory named ***impl*** will be visible in the *Explorer* panel

This directory contains the ***ip*** subdirectory which contains the generated IP package.

Take a moment to explore the contents of the ***ip*** directory.

With the IP generated, the next step would be to include it in an IP Integrator design (which will be covered in the next tutorial). For future reference, however, it is worth briefly describing how this would be done.

In order to include HLS generated IP in IP Integrator, it must first be added to the Vivado IP Catalog. To do this you must add the output from HLS to an IP repository. This can be achieved by either adding the HLS generated output directory to an existing IUP repository directory, or by creating a new repository. In either case, the directory is the same. In this case:

C:\Zynq_Book\hls_nco\solution1\impl\ip

We have now completed the generation of the NCO component as an IP Integrator compatible AXI-Lite block. You should now be familiar with:

- Specifying directives in Vivado HLS designs which define the control interface of the exported RTL.
- The process of specifying and AXI4 interface for a design, to enable a Vivado HLS system to be easily connected to the Zynq PS.
- Exporting a Vivado HLS design as an IP core that is compatible with the Vivado IP Catalog and IP Integrator.

The Zynq Book Tutorial 5

Adventures with IP Integrator

v1.2.1, September 2014

Revision History

Date	Version	Changes
22/10/2013	1.0	First release for Vivado Design Suite version 2013.3
28/01/2014	1.1	Updated for changes in Vivado Design Suite version 2013.4
06/05/2014	1.2	Updated for changes in Vivado Design Suite version 2014.1
10/09/2014	1.2.1	Minor corrections.

Introduction

In this tutorial you will bring together all of the custom IP modules that you created in the previous set of practical exercises, along with other IP from the Vivado IP Catalog, to create a DSP system for implementation on the ZedBoard. IP for the control of the control of the audio codec on the ZedBoard will be introduced and all modifications to the IP Integrator design will be carried out. A software application will be developed in the SDK which will configure all of the IP modules and control the interactions between them and the PS.

The tutorial is split into three exercises as follows:

Exercise 5A - This exercise focuses on importing all of the custom IP modules into the Vivado IP Catalog for inclusion in an IP Integrator DSP design. The individual IP blocks will be explored, along with their customisable parameters.

Exercise 5B - The Analog Devices ADAU1761 audio codec on the ZedBoard will be introduced in this exercise, with the inclusion of some prepackaged IP. This IP implements the I^{2}S serial communication for sending and receiving audio samples to/from the audio codec. The audio samples are transfered between the PL and the PS via a standard AXI-Lite connection. In order to use the audio codec, a variety of modifications must be made to the Zynq PS, such as the inclusion of second fabric clock to drive the codec, and the enabling of a I^{2}C interface for the communication of control signals between the PS and the codec.

In order to map the external interfaces in the design to physical pins on the Zynq device, a Xilinx Design Constraints (XDC) file must be created and included in the design. This informs the synthesis and implementation processes in Vivado where to route the external interface signals. The format of the XDC file will be explored before generating the hardware for the finalised design.

Exercise 5C - In this final exercise, the finalised design from Exercise 5B will be exported to the SDK for software development. Here, the application which will control the interactions between the various custom IP modules, the PS and the audio codec will be created. The various software driver files will also be explored before building and running the application on the ZedBoard for testing.

NOTE: Exercise 5C requires you to be able to send keyboard commands to the Zynq PS via the UART terminal. To do this, it is necessary to use third-party terminal program. In this tutorial, we shall be using PuTTY which can be downloaded for free from the following link:
http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html
You can download PuTTY as a standalone executable file so that no installation is required. To download the standalone executable, select the **putty.exe** download from the **Binaries** section.

Exercise 5A Importing IP to the Vivado IP Catalog

In this exercise we will be concentrating on importing existing custom IP into the Vivado IP Catalog. We will be importing the various IP blocks which we created in **The Zynq Book Tutorial IP Creation**.

We will start by creating a new Vivado Project.

(a) Launch Vivado 2014.1 and create a new project called ***adventures_with_ip*** in the **C:\Zynq_Book** directory, ensuring that the option to ***Create project subdirectory*** is selected. Select ***VHDL*** as the *Target language* and the ***ZedBoard*** as the *Default Part*.

(b) From *Flow Navigator*, select ***IP Catalog*** from the **Project Manager** section.

The **IP Catalog** will open in the *Workspace*, as seen in Figure 5.1.

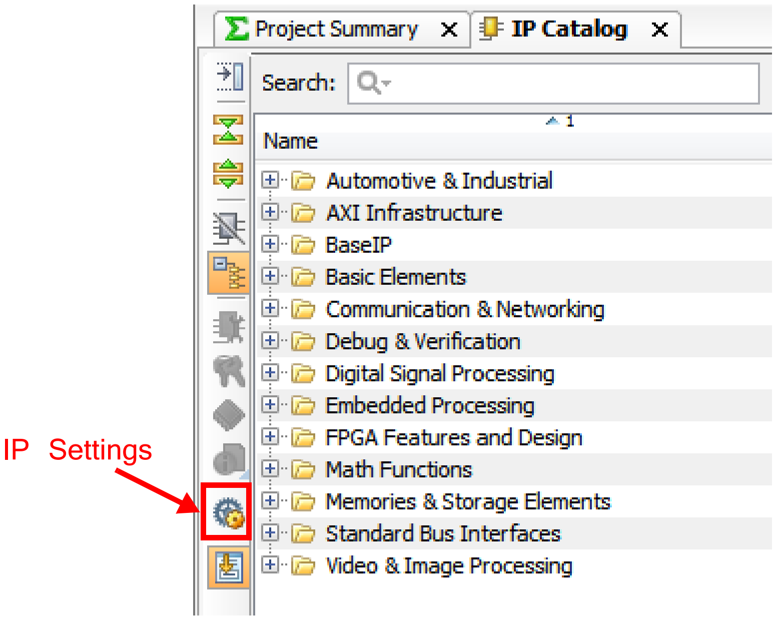


Figure 5.1: Vivado IP Catalog

In order to import our custom IP into the IP Catalog, we must add a new software repository to the IP Catalog. We will create a new directory to act as our IP repository and all of our IP sources to it.

(c) In Windows Explorer, navigate to the location: **C:\Zynq_Book\ip_repo**. This is the IP repository that we created in **Tutorial 4**.

We must now add each of the IP sources which we created in **The Zynq Book Tutorial IP Creation** to our repository.

As the LED controller IP is already present in the IP repository, we do not need to import it.

(d) In Windows Explorer, navigate to **C:\Zynq_Book\hdl_coder_lms\hdl_prj\ipcore\lms_pcore_v1_00_a** and copy the archived IP ZIP file, ***xilinx.com_user_lms_pcore_1.0.zip*** to the **ip_repo** directory.

(e) In Windows Explorer, navigate to **C:\Zynq_Book\hls_nco\solution1\impl\ip** and copy the archived IP ZIP file, ***xilinx_com_hls_nco_1_0.zip*** to the **ip_repo** directory.

That completes the copying of our custom made IP sources to our newly created IP repository. We will now add one more IP source to our repository — an existing IP block which controls the audio codec on the ZedBoard.

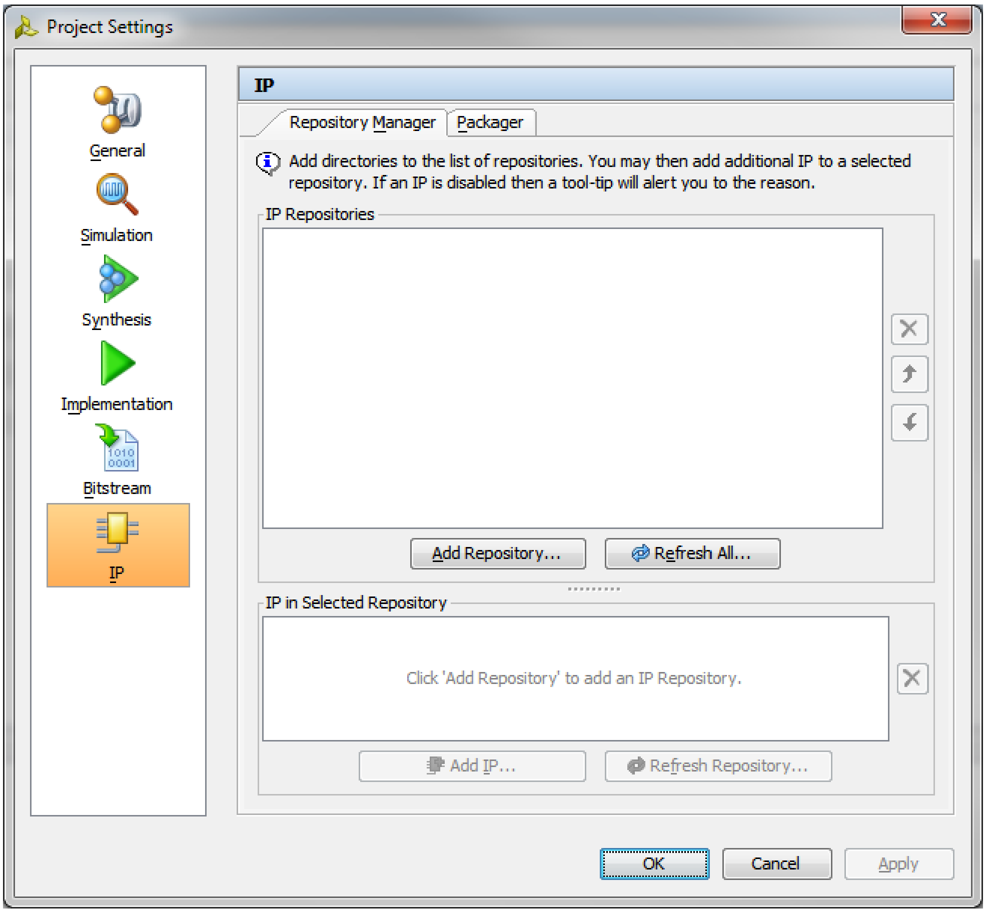
(f) In Windows Explorer, navigate to **C:\Zynq_Book\sources\adventures_with_ip_integrator\ip** and copy the archived IP ZIP file, ***zed_audio_ctrl.zip*** to the **ip_repo** directory that we located in Step (c).

If you have not completed the previous tutorial, a master set of the IP sources is contained in **C:\Zynq_Book\sources\adventures_with_ip_integrator\ip** which you can copy into the repository for use in this tutorial.

Now that we have created the IP repository and added all of our existing IP sources, we can now add the repository to the IP Catalog.

(g) In the Vivado IP Catalog tab, click the ***IP Settings*** button, ⚙, as highlighted in Figure 5.1.

The IP Settings window will open, as shown in Figure 5.2.



IP

Add directories to the list of repositories. You may then add additional IP to a selected repository. If an IP is disabled then a tool-tip will alert you to the reason.

Click 'Add Repository' to add an IP Repository.

Figure 5.2: IP Settings Window

(h) Click ***Add Repository*** in the *IP Repositories* panel, and browse to **C:\Zynq_Book\ip_repo**.

Click ***Select*** to add the repository to the IP Catalog.

You should see that the LED Controller IP is already present in the *IP in Selected Repository* pane as it is in un-archived format.

We must now add the other IP sources to the repository by un-archiving them.

(i) In the IP in Selected Repository panel, shown in Figure 5.2, click ***Add IP***.

The Select IP TO Add To Repository window will open:

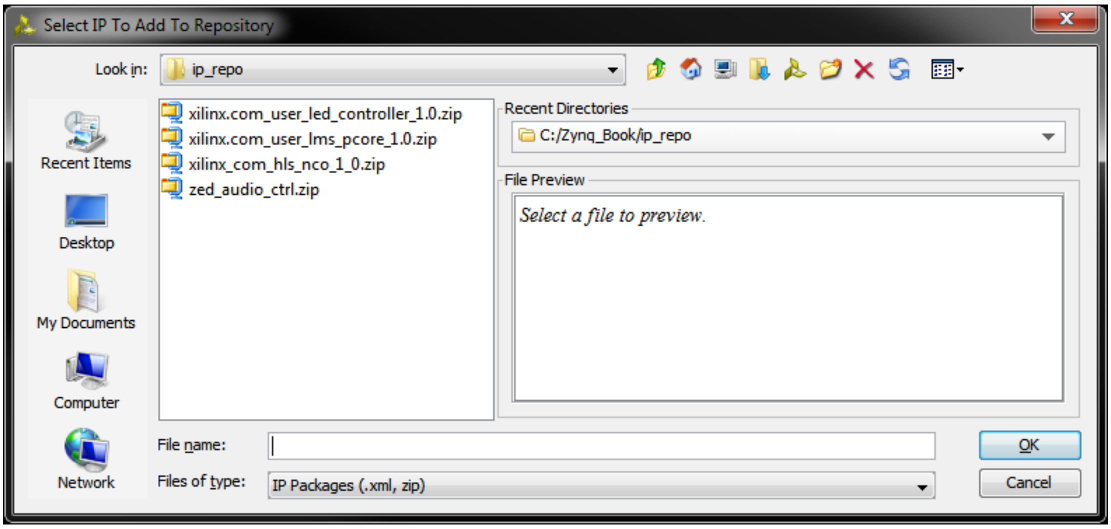

Select IP To Add To Repository
Look in: ip_repo
Recent Items
Desktop
My Documents
Computer
Network
xilinx.com_user_led_controller_1.0.zip
xilinx.com_user_lms_pcore_1.0.zip
xilinx_com_hls_nco_1_0.zip
zed_audio_ctrl.zip
Recent Directories
C:/Zynq_Book/ip_repo
File Preview
Select a file to preview.
File name:
Files of type: IP Packages (.xml, zip)
OK
Cancel

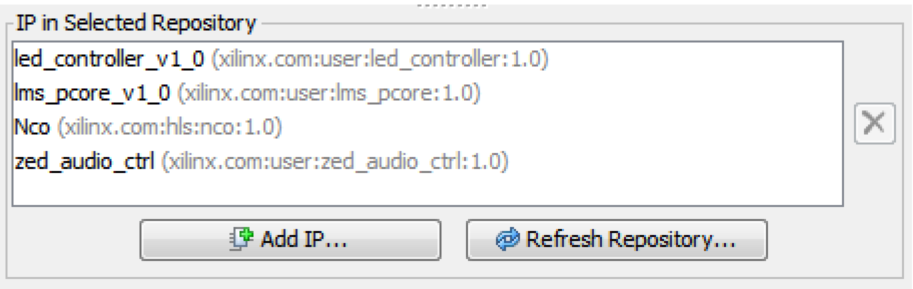
Figure 5.3: Select IP to Add to Repository

Select **xilinx.com_user_led_controller_1.0.zip** and click **OK**. This will extract the archived IP sources into a usable format in the repository.

(j) Repeat this procedure for the remaining IP sources:

- **xilinx.com_user_lms_pcore_1.0.zip**
- **xilinx_com_hls_nco_1_0.zip**
- **zed_audio_ctrl.zip**

The resulting IP in Selected Repository panel is shown in Figure 5.4.



led_controller_v1_0 (xilinx.com:user:led_controller:1.0)
lms_pcore_v1_0 (xilinx.com:user:lms_pcore:1.0)
Nco (xilinx.com:hls:nco:1.0)
zed_audio_ctrl (xilinx.com:user:zed_audio_ctrl:1.0)

Figure 5.4: All IP sources added to IP Catalog

Click **OK**.

With all of our IP now imported into the IP Catalog, we can now create an IP Integrator block design which incorporates all of the IP blocks.

(k) In *Flow Navigator*, select ***Create Block Design***.

(l) In the *Create Block Design* window, set the **Design name** as ***ip_design***, and click ***OK***.

(m) In the block design canvas, right-click and select ***Add IP***.

In the Search box, enter ***led_controller*** and double-click ***led_controller_v1_0*** to add an instance of the LED controller IP to the design.

(n) Repeat Step (m) searching for:

- ***nco*** and double-clicking ***Nco***
- ***lms*** and double-clicking ***lms_pcore_v1_0***

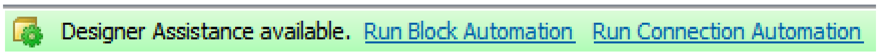
We have now added all of the custom IP that we created in the previous tutorial. At this point we will avoid adding the audio controller IP, as it is the focus of the next exercise.

In order to connect and control all of the IP, we must now add an instance of a Zynq Processor.

(o) In the block design canvas, right-click and select ***Add IP***.

In the Search box, enter ***zynq*** and double-click ***ZYNQ7 Processing System***.

At this stage, Designer Assistance should be available:



Designer Assistance available. Run Block Automation Run Connection Automation

(p) Click ***Run Block Automation*** for **processing_system7_0** and click ***OK*** to complete configuration.

(q) ***Run Connection Automation*** for each of the three IP blocks, to connect them to the Zynq7 Processing System block, via and AXI Interconnect block.

You may recall that to allow the LED Controller block to control the LEDs on the board, the LEDs_out port must be made external.

(r) Hover the mouse pointer over the **LEDs_out** interface on the **led_controller** block until the cursor changes to a pencil. Right-click and select ***Make External***. Alternatively, select the interface and use the keyboard shortcut ***Ctrl+T***.

Notice that the **lms_pcore_0** block has two unconnected input ports, as highlighted in Figure 5.5.

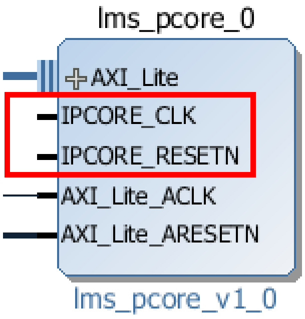
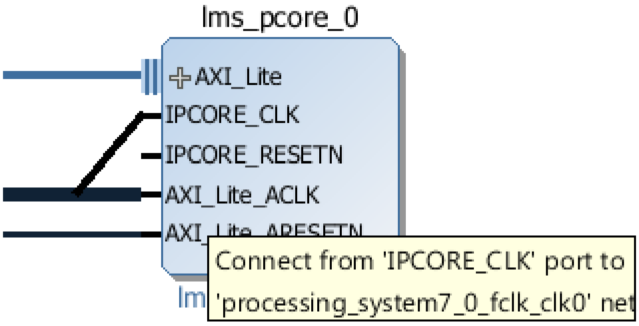


Figure 5.5: LMS IP block

These are the CLK and reset ports of the IP, and must be connected in order for the IP to be functional.

(s) Hover the mouse pointer over the **IPCORE_CLK** interface on the **lms_pcore_0** block until the cursor changes to a pencil. Click and drag the mouse pointer until it is hovering over the wire that connects to the **AXI_Lite_ACLK** interface and the wire is highlighted, as shown in Figure 5.6, and release the mouse button to create the connection.



Connect from 'IPCORE_CLK' port to 'processing_system7_0_fclk_clk0' net

Figure 5.6: Manually connecting the LMS IP CLK

You should also see a pop-up message notifying you of the net which you are connecting to.

(t) Repeat the procedure of the previous step to, this time, connect the **IPCORE_RESETN** interface to the wire which connects to the **AXI_Lite_ARESETN** interface.

At this stage we must now add and configure the audio controller IP, and so we will conclude this first exercise on importing custom IP to the Vivado IP Catalog. You should now be familiar with:

- Adding an IP repository to the Vivado IP Catalog.
- Importing and adding archived IP files to a custom IP repository.
- Adding custom IP to a Vivado IP Integrator block design.

Note: Do not close the current Vivado project as we will be using it again in the next exercise.

Exercise 5B ZedBoard Audio in Vivado IP Integrator

In this exercise we will be focusing on adding an audio controller IP instance to an existing Vivado IP Integrator design, and the modifications which must be made to the Zynq Processor block in order to use the ADAU1761 audio codec on the ZedBoard. Such modifications include the addition of a second PL fabric clock and the enabling of the I^{2}C interface for the communication of control signals between the Zynq PS and the codec.

We will begin by adding an instance of the audio controller IP to the block design.

(a) In the Vivado IP Integrator block design canvas, right-click and select ***Add IP***.
Search for **audio** and double-click on ***zed_audio_ctrl***, to add an instance to the block design.

The zed_audio_ctrl block should now be visible on the canvas, as shown in Figure 5.7.

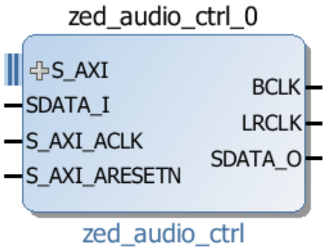


Figure 5.7: ZedBoard Audio Controller block

(b) Make the initial connection between the Zynq PS and the zed_audio_ctrl block by clicking ***Run Connection Automation***.

You should notice that there are still four unconnected ports. These are required to be made external to connect to the physical pins of the ZedBoard's audio codec.

(c) Hover the mouse pointer over each of the unconnected interfaces on the **zed_audio_ctrl** block until the cursor changes to a pencil. Right-click and select ***Make External***. Alternatively, select the interface and use the keyboard shortcut ***Ctrl+T***.

The next step is to make the necessary modifications to the Zynq7 PS block.

(d) Double-click on the ***Zynq7 Processing System*** block to open the Re-customize IP window, as shown in Figure 5.8.

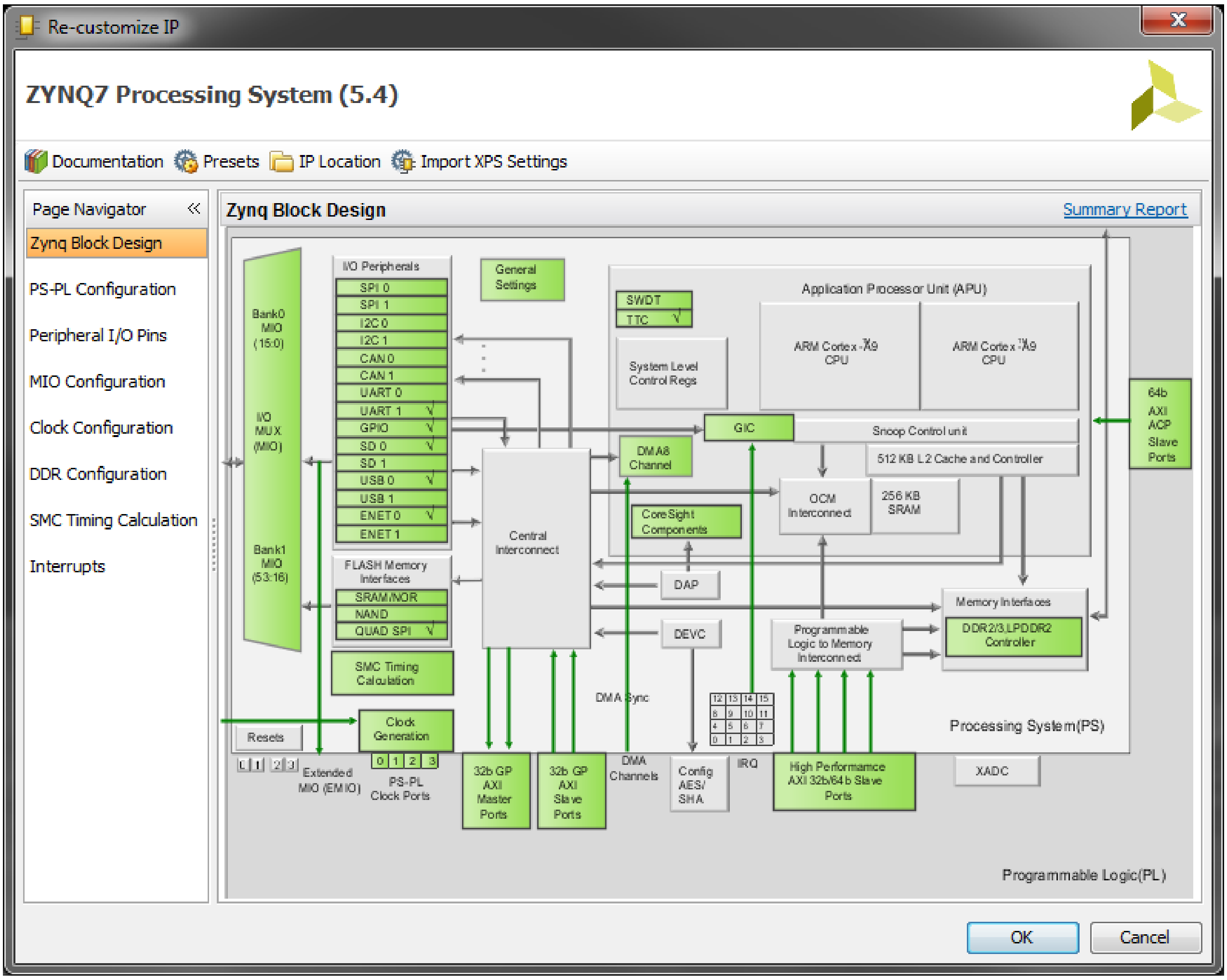
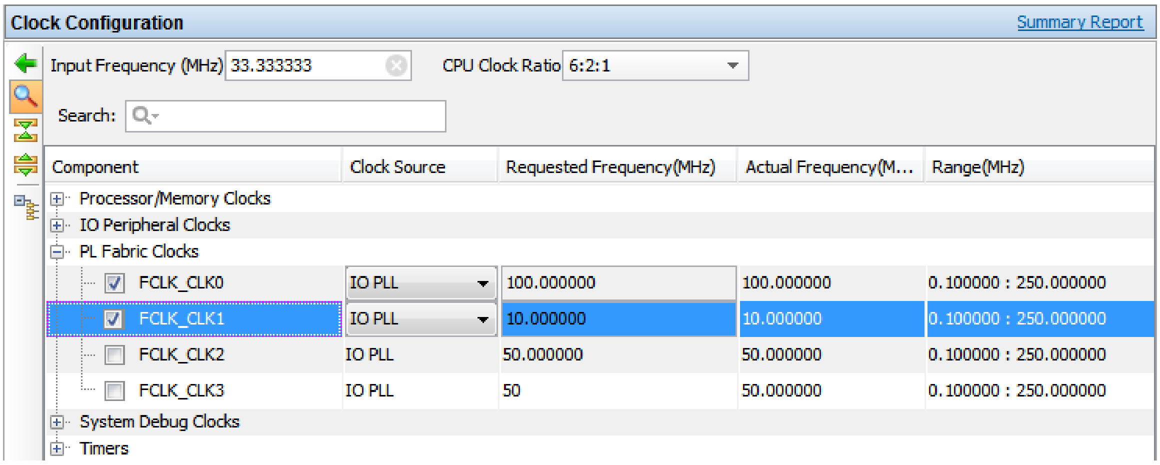


Figure 5.8: Re-customize IP window for Zynq PS

This view allows you to make changes to the configuration of the Zynq PS. As IP Integrator is *board aware*, all of the basic settings that apply to the ZedBoard have been made for us. There are a few changes, however, that must be made when using the audio codec.

First we will add a second PL fabric clock as a separate **10 MHz** clock is required for the **MCLK** pin on the audio codec.

(e) Click on ***Clock Configuration*** in the *Page Navigator* panel on the left hand side of the window. Expand ***PL Fabric*** clocks in the *Clock Configuration* panel, and enable ***FCLK_CLK1***. Change the ***Requested Frequency*** of **FCLK_CLK1** to ***10 MHz***, as shown in Figure 5.9.



Clock Configuration

Input Frequency (MHz) 33.333333   CPU Clock Ratio 6:2:1

Search:

Component	Clock Source	Requested Frequency(MHz)	Actual Frequency(M...	Range(MHz)
⊞ Processor/Memory Clocks				
⊞ IO Peripheral Clocks				
⊟ PL Fabric Clocks				
☑ FCLK_CLK0	IO PLL	100.000000	100.000000	0.100000 : 250.000000
☑ FCLK_CLK1	IO PLL	10.000000	10.000000	0.100000 : 250.000000
□ FCLK_CLK2	IO PLL	50.000000	50.000000	0.100000 : 250.000000
□ FCLK_CLK3	IO PLL	50	50.000000	0.100000 : 250.000000
⊞ System Debug Clocks				
⊞ Timers				

Figure 5.9: Adding a 10 MHz fabric clock

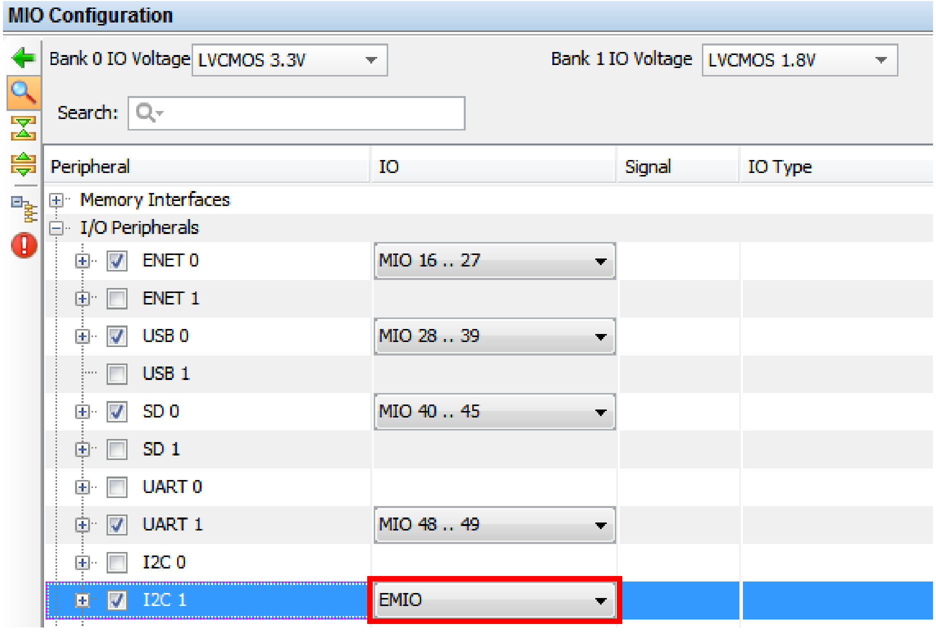
Next, we must enable one of the Zynq PS's I^{2}C communication interfaces to allow the PS to communicate with the audio codec.

(f) Select ***MIO Configuration*** from the *Page Navigator* panel.

This configuration view allows us to enable/disable the PS peripherals. These peripherals can be routed through the dedicated **Multiplexed I/Os (MIO)** on the device, or through the **Extended Multiplexed I/Os (EMIOs)** which route to the PL fabric.

As we want to communicate with the audio codec (which is connected to fabric pins of the Zynq device) we will be routing the I^{2}C signals through the EMIOs.

(g) Enable the ***I2C 1*** peripheral in the *MIO Configuration* panel. ***EMIO*** should automatically be selected for **IO**, as shown in Figure 5.10.



MIO Configuration

Bank 0 IO Voltage LVCMOS 3.3V   Bank 1 IO Voltage LVCMOS 1.8V

Search:

Peripheral	IO	Signal	IO Type
Memory Interfaces			
I/O Peripherals			
☑ ENET 0	MIO 16 .. 27		
□ ENET 1			
☑ USB 0	MIO 28 .. 39		
□ USB 1			
☑ SD 0	MIO 40 .. 45		
□ SD 1			
□ UART 0			
☑ UART 1	MIO 48 .. 49		
□ I2C 0			
☑ I2C 1	EMIO		

Figure 5.10: Configuring the I2C interface

No more changes to the Zynq PS are required.

(h) Close the *Re-customize IP* window and apply the changes to the PS by clicking ***OK***.

The IP Integrator canvas should update, and the Zynq7 Processing System block should now look like Figure 5.10.

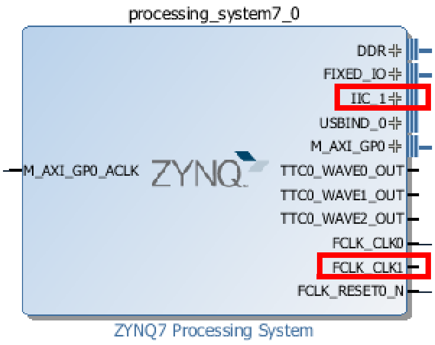


Figure 5.11: Zynq7 Processing System block

You should note the addition of the two new interfaces, **IIC_1** and **FCLK_CLK1**. As these will be driving signals on the audio codec, which is situated on the board (external to the Zynq device), we must make these external.

(i) Hover the mouse pointer over each of the **IIC_1** and **FCLK_CLK1** interfaces on the processing_system1_0 block until the cursor changes to a pencil. Right-click and select ***Make External***. Alternatively, select the interface and use the keyboard shortcut ***Ctrl+T***.

The final addition to the Block design that we need to make, is to add two GPIO instances:

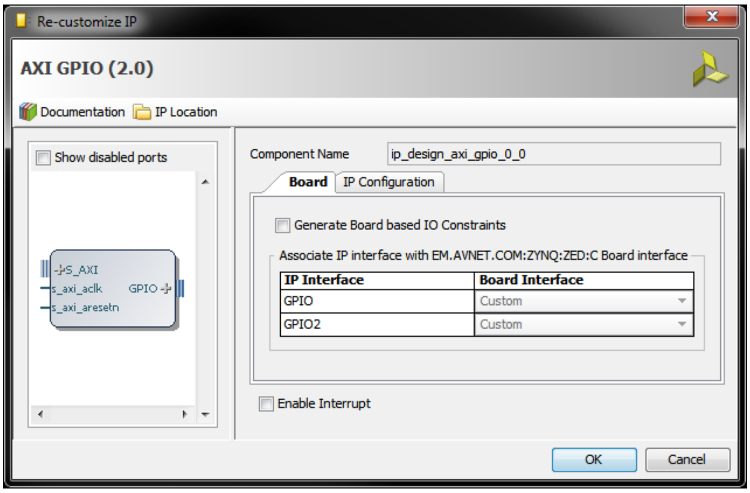
- Single-channel GPIO with a width of 2-bits to connect to the audio codec's ADDR pins.
- Dual-channel GPIO with a width of 32-bits to connect to the push buttons and slide switches on the ZedBoard, for user input.

First we will add the GPIO to connect to the codec's ADDR pins.

(j) In the Vivado IP Integrator block design canvas, right-click and select ***Add IP***. Search for **gpio** and double-click on ***AXI_GPIO***, to add an instance to the block design.

(k) ***Run Connection Automation*** for the **axi_gpio_0/S_AXI interface**, to connect the GPIO controller to the Zynq PS via the AXI Interconnect.

(l) Open the *Re-customize IP* window by double-clicking on the **axi_gpio_0** block. The window, as shown in Figure 5.12, will open.



AXI GPIO (2.0)

□ Show disabled ports

Component Name: ip_design_axi_gpio_0_0

Board | IP Configuration

□ Generate Board based IO Constraints

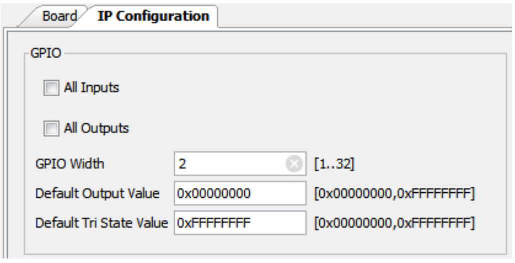
Associate IP interface with EM.AVNET.COM:ZYNQ:ZED:C Board interface

IP Interface	Board Interface
GPIO	Custom
GPIO2	Custom

□ Enable Interrupt

Figure 5.12: Re-customize IP window (GPIO)

(m) Select the ***IP Configuration*** tab. Enter ***2*** as the **GPIO Width**, as shown in Figure 5.13, and close the window by clicking ***OK***.



GPIO

□ All Inputs

□ All Outputs

GPIO Width	2	[1..32]
Default Output Value	0x00000000	[0x00000000,0xFFFFFFFF]
Default Tri State Value	0xFFFFFFFF	[0x00000000,0xFFFFFFFF]

Figure 5.13: GPIO width setting

(n) Make the ***GPIO*** interface of the **axi_gpio_0** block external.

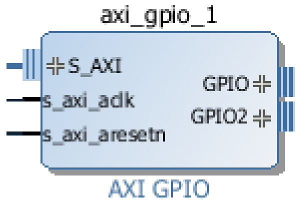
Next we will add a second instance of the AXI GPIO Controller.

(o) Add an instance of the ***AXI_GPIO*** IP to the block design and ***Run Connection Automation*** for the **axi_gpio_1/S_AXI interface**, to connect the GPIO controller to the Zynq PS via the AXI Interconnect.

(p) Double-click on the **axi_gpio_1** block to open the *Re-customize IP* window.

In the *IP Configuration* tab, select the option to ***Enable Dual Channel***, and click ***OK***.

You should see that the **axi_gpio_1** block now has two output ports, 1 each to connect to the **push buttons** and the **slide switches** on the ZedBoard:



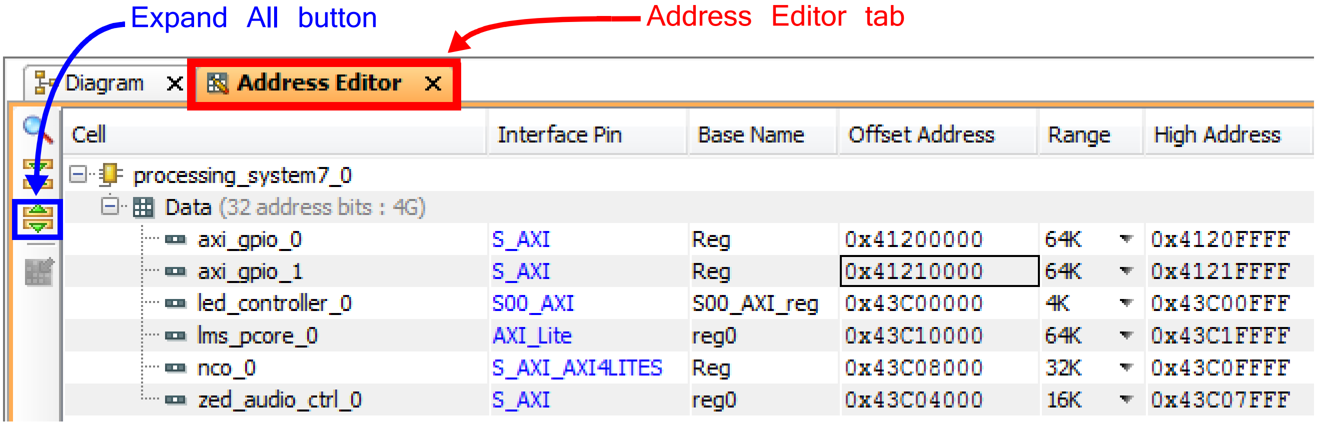
(q) ***Run Connection Automation*** for **/axi_gpio_1/GPIO** and select ***BTNs_5Bits*** as the option for **Select Board Interface**.

Click ***OK***.

(r) ***Run Connection Automation*** for **/axi_gpio_2/GPIO1** and select ***SWs_8Bits*** as the option for **Select Board Interface**.

Click ***OK***.

(s) Select the ***Address Editor*** tab from the *Block Design* window, as highlighted in Figure 5.14.



Expand All button

Address Editor tab

Cell	Interface Pin	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 4G)					
axi_gpio_0	S_AXI	Reg	0x41200000	64K	0x4120FFFF
axi_gpio_1	S_AXI	Reg	0x41210000	64K	0x4121FFFF
led_controller_0	S00_AXI	S00_AXI_reg	0x43C00000	4K	0x43C00FFF
lms_pcore_0	AXI_Lite	reg0	0x43C10000	64K	0x43C1FFFF
nco_0	S_AXI_AXI4LITES	Reg	0x43C08000	32K	0x43C0FFFF
zed_audio_ctrl_0	S_AXI	reg0	0x43C04000	16K	0x43C07FFF

Figure 5.14: Address Editor tab

(t) Click the ***Expand All*** button, as highlighted in Figure 5.14.

Check the assigned *Offset Address* and *Range* for each of the peripheral Cells.

If they do not match those in Figure 5.14, you must reassign the addresses by following the procedure in this step. If they match those in Figure 5.14, you can skip this step and move on to **Step (u)**.

- Highlight all of the peripheral *Cells* by holding the **Ctrl** key on the keyboard while clicking on each cell in turn.
- Right-click on any of the selected *Cells* and select ***Unmap Segment***. This will unmap the addresses for all of the peripherals
- Expand the **Unmapped Slaves** section and highlight all of the *Cells*.
- Right-click on any of the *Cells* and select ***Assign Address***.
- The *Offset Address* and *Range* for each peripheral *Cell* should now match those in Figure 5.14. If they don't, you can edit the *Offset Address* and *Range* values manually.

(u) Return to the block design by selecting the ***Diagram*** tab in the *IP Integrator* window.

(v) Click the Regenerate Layout button, , to regenerate the layout of the various IP blocks and make the block design easier to follow. Your complete block design should be similar to Figure 5.15.

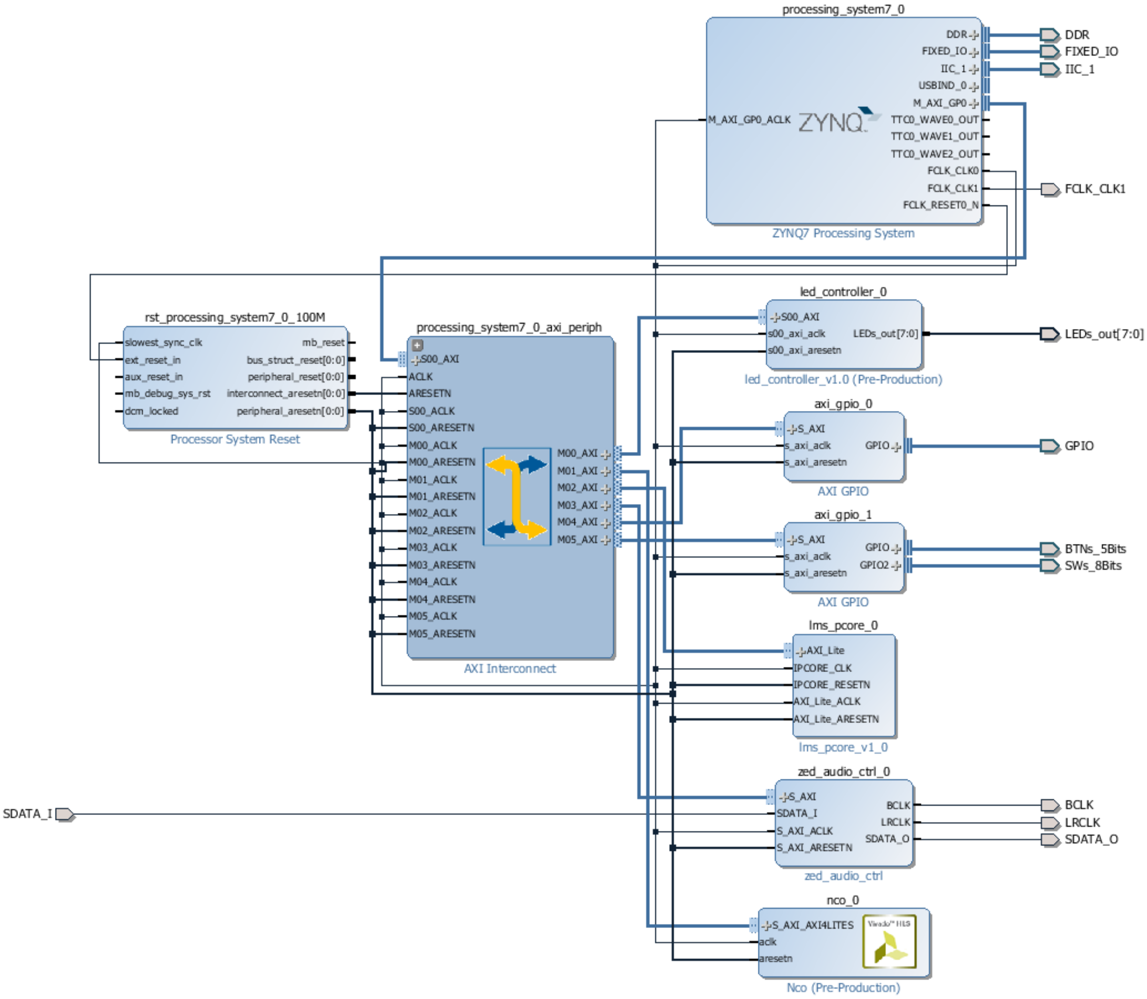


Figure 5.15: Completed block design

(w) Save the block design.

Before we can run synthesis and implementation for our design, we must generate the RTL files for our block design.

(x) Generate a top-level HDL wrapper file, by right-clicking on **ip_design** in the **Sources** tab and selecting ***Create HDL Wrapper***.
In the *Create HDL Wrapper* window, select **Let Vivado manage wrapper and auto-update**, and click ***OK***.

The next task that we have to do in Vivado before we can synthesis and implementation of the design, is to add a constraints file which will map the external interfaces of our design to specific pins on the Zynq device.

(y) Select ***Add Sources*** form the **Project Manager** section of *Flow Navigator*.
In the *Add Sources* window, select ***Add or Create Constraints***, and click ***Next***.
In the *Add or Create Constraints* window, select ***Add Files.***
Navigate to **C:/Zynq_Book/sources/adventures_with_ip_integrator/constraints**, select **adventures_with_ip.xdc**, and click ***OK***.
Click ***Finish*** to close the *Add Sources* window, and import the constraints file.

(z) Open the constraints file by expanding the *Constraints* section of **Sources** tab, and double-clicking on ***adventures_with_ip.xdc***.

The top section of the file contains the constraints which map the individual bits of the LEDs_out interface to the corresponding pins on the Zynq device, and you will have seen these before in the first exercise of the previous tutorial.

The bottom section of the file, as shown in Figure 5.16, contains the constraints which map the various external ports of the design which relate to the audio codec, to their corresponding pins on the Zynq device.

```
# ZedBoard Audio Codec Constraints
set_property PACKAGE_PIN AA6 [get_ports BCLK]
set_property IOSTANDARD LVCMOS33 [get_ports BCLK]

set_property PACKAGE_PIN Y6 [get_ports LRCLK]
set_property IOSTANDARD LVCMOS33 [get_ports LRCLK]

set_property PACKAGE_PIN AA7 [get_ports SDATA_I]
set_property IOSTANDARD LVCMOS33 [get_ports SDATA_I]

set_property PACKAGE_PIN Y8 [get_ports SDATA_O]
set_property IOSTANDARD LVCMOS33 [get_ports SDATA_O]

#MCLK
set_property PACKAGE_PIN AB2 [get_ports FCLK_CLK1]
set_property IOSTANDARD LVCMOS33 [get_ports FCLK_CLK1]

set_property PACKAGE_PIN AB4 [get_ports IIC_1_scl_io]
set_property IOSTANDARD LVCMOS33 [get_ports IIC_1_scl_io]

set_property PACKAGE_PIN AB5 [get_ports IIC_1_sda_io]
set_property IOSTANDARD LVCMOS33 [get_ports IIC_1_sda_io]

set_property PACKAGE_PIN AB1 [get_ports {GPIO_tri_io[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {GPIO_tri_io[0]}]

set_property PACKAGE_PIN Y5 [get_ports {GPIO_tri_io[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {GPIO_tri_io[1]}]
```

Figure 5.16: ZedBoard audio codec constraints

Next, we will create a bitstream so that we can program the PL of the Zynq device with our design.

(aa) In *Flow Navigator*, select ***Generate Bitstream*** from the **Program and Debug** section.

At the *No Implementation Results Available* window, click ***Yes*** to launch synthesis and implementation.

When bitstream generation is complete, select ***Open Implemented Design*** in the *Bitstream Generation Completed* window, and click ***OK***.

Finally, we can export the hardware to the SDK, where we will create a software application to control the system in the next exercise.

(ab) Select **File > Export > Export Hardware for SDK** from the *Menu Bar*.

Ensure that the options to ***Include Bitstream*** and ***Launch SDK*** are selected, and click ***OK***.

This concludes this exercise on audio of the ZedBoard. You should now be familiar with:

- Making the required changes to the Zynq PS in order to use the audio codec on the ZedBoard.
- Making the required external connections to allow the Zynq PL to be connected to the audio codec via the external Zynq device pins.
- Using a constraints file to map the external interfaces of the design which relate to the audio codec, to the corresponding pins on the Zynq device.

Exercise 5C Creating an Audio Software Application in SDK

In this final exercise we will be creating a software application which ties together all of the IP modules which we have created, to create a DSP-oriented system. The procedure of setting up the ZedBoard audio codec via the hardware registers will also be introduced.

Once the SDK has launched from the previous exercise, we can start by creating a new application.

(a) Select **File > New > Application Project** from the *Menu Bar*.

In the *New Project* dialogue, enter ***adventures_with_ip*** as the *Project name*.

By default the option to create a new **Board Support Package** will be selected.

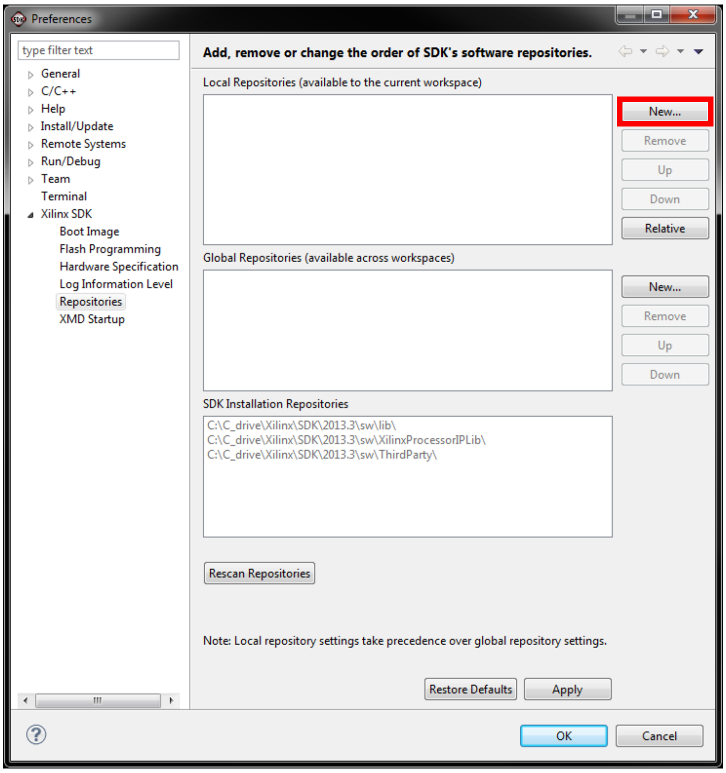
Click ***Next***.

(b) In the *Templates* dialogue, select ***Empty Application***, and click ***Finish***.

You should recall that when we created the custom IP peripherals in the previous tutorial that a set of software driver files were generated for each. We must now point SDK to those driver files. This is done by adding a new repositories to the SDK project.

(c) Navigate to **Xilinx Tools > Repositories** in the *Menu Bar*.

In the *Repositories Preferences* window, click on ***New***, as shown in Figure 5.17.



Add, remove or change the order of SDK's software repositories.

Local Repositories (available to the current workspace)

Global Repositories (available across workspaces)

SDK Installation Repositories

C:\C_drive\Xilinx\SDK\2013.3\sw\lib\
C:\C_drive\Xilinx\SDK\2013.3\sw\XilinxProcessorIPLib\
C:\C_drive\Xilinx\SDK\2013.3\sw\ThirdParty\

Note: Local repository settings take precedence over global repository settings.

Figure 5.17: SDK Repository Peripherals window

(d) Add the LED Controller drivers by browsing to the directory:

C:\Zynq_Book\ip_repo\led_controller_1.0 and clicking ***OK***.

(e) Click ***New***.

Add the NCO drivers by browsing to the directory:

C:\Zynq_Book\ip_repo\xilinx_com_hls_nco_1_0

and clicking ***OK***.

Upon closing the preferences window, SDK will automatically scan the repository and rebuild the project to include the driver files.

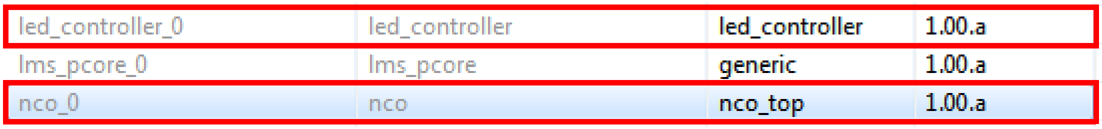
We must now assign the newly imported drivers to their corresponding peripherals.

(f) The **system.mss** tab should be open in the Workspace. If it is not, open it by expanding **adventures_with_ip_bsp** in *Project Explorer* and double-clicking on ***system.mss.***

(g) At the top left of the **system.mss** tab, click ***Modify this BSP's Settings***.

The *Board Support Package Settings* window will open.

(h) Select drivers from the left-hand menu and assign the **led_controller** driver to the **led_controller_0** component and the **nco_top** driver to the **nco_0** component, as highlighted in Figure 5.18.



led_controller_0	led_controller	led_controller	1.00.a
lms_pcore_0	lms_pcore	generic	1.00.a
nco_0	nco	nco_top	1.00.a

Figure 5.18: Driver assignment

Click ***OK***.

The project will now rebuild.

The LMS IP core that we created with Mathworks HDL Coder and the audio codec IP also have software drivers, but due to their directory structure, we must import their drivers to the workspace rather than use a repository.

(i) In the *Project Explorer* panel, expand **adventures_with_ip**, right-click on **src** and select ***Import***.

In the *Import* window, expand **General** and double-click on ***File System***.

Click ***Browse*** in the top right corner, and navigate to

C:\Zynq_Book\hdl_coder_lms\hdl_prj\ipcore\lms_pcore_v1_00_a\include.

Click ***OK***, to import the LMS IP driver.

In the right-hand panel, select ***lms_pcore_addr.h*** and click ***Finish.***

Note: This directory will only be available if you have completed **Exercise 4B** of **Tutorial 4**.

If you have not completed this exercise, you can obtain ***lms_pcore_addr.h*** from the source directory **C:\Zynq_Book\sources\adventures_with_ip_integrator\drivers**.

(j) Similarly, import the audio controller IP driver, **audio.h**, from the directory **C:\Zynq_Book\sources\adventures_with_ip_integrator\drivers**.

With all the driver files for the IP imported, we can import the source files for our application.

(k) Follow the same procedure as in **Step (i)** to import the following files from the **C:\Zynq_Book\sources\adventures_with_ip_integrator\software** directory:

- **adventures_with_ip.h**
- **adventures_with_ip.c**
- **audio.c**
- **ip_functions.c**

The source files will be imported and the application should build.

(l) Open the header file **adventures_with_ip.h** by double-clicking on it in *Project Explorer*.

This is the main header file for the software application. At the top of the file you should see a list of included header files, which define a variety of functions which are used in the software application.

Further down the file you should see the inclusion of the custom IP header files which we imported earlier:

```
/* --------------------------------------------------------------------------- *
 *                          Custom IP Header Files                             *
 * --------------------------------------------------------------------------- */
#include "audio.h"
#include "lms_pcore_addr.h"
#include "xnco.h"
```

As an example of one of the header files that was created during the IP creation process, we will open the header for the LMS IP core.

(m) In the *Outline* tab on the right hand side of the SDK window, double click on **lms_pcore_addr.h**.

In the LMS header file, you should see the following definitions:

```
#define  IPCore_Reset_lms_pcore    0x0  //write 0x1 to bit 0 to reset IP core
#define  IPCore_Enable_lms_pcore   0x4  //enabled (by default) when bit 0 is 0x1
#define  IPCore_Strobe_lms_pcore   0x8  //write 1 to bit 0 after write all input data
#define  IPCore_Ready_lms_pcore 0xC  //wait until bit 0 is 1 before read output data
#define  x_k__Data_lms_pcore       0x100  //data register for port x(k)
#define  d_k__Data_lms_pcore       0x104  //data register for port d(k)
#define  e_k__Data_lms_pcore       0x108  //data register for port e(k)
```

These define the memory-mapped address offsets of the various signals of the LMS peripheral. Data can be transferred between the peripheral in the PL and the software in the PS by writing to, or reading from the these offset addresses. The actual address that would be used to access these signals would be **BASE ADDRESS + OFFSET**.

Each IP peripheral that we added to our block design in IP Integrator is automatically assigned a base address in memory. These addresses can be determined from a *Xilinx parameters* C header file which is automatically created when exporting an IP Integrator design which contains a Zynq Processing System. The header file is called **xparameters.h**.

We shall now explore the *Xilinx parameters* header file.

(n) Switch back to the **adventures_with_ip.h** tab in the *Editor* window.

xparameters.h is included in this main header file, and is therefore accessible from the *Outline* tab.

(o) Open **xparameters.h** by double-clicking on it in the *Outline* tab.

Here you should see a list of memory *base address* definitions, along with a number of other parameters.

As we were previously looking at the LMS header file, we will look at the definition of the base address for the LMS peripheral.

(p) Scroll down the file until you see the following lines:

```
/* Definitions for peripheral LMS_PCORE_0 */
#define XPAR_LMS_PCORE_0_BASEADDR 0x43C10000
#define XPAR_LMS_PCORE_0_HIGHADDR 0x43C1FFFF
```

Here we see the definitions of both the base and high addresses in memory for the LMS peripheral. As the difference between the high address and the base address is `0xFFFF`, the LMS peripheral has an addressable range of **65536 bits**, or **64 Bytes**.

Referring back to the memory address offsets for the LMS block in Step (m), if we, for example, wanted to write data to the input port **`x(k)`**, we would do this by writing the desired value to the BASE ADDRESS + OFFSET, which in this case would be:

`XPAR_LMS_PCORE_0_BASEADDR + x_k__Data_lms_pcore = 0x43C10000 + 0x100`

Giving a unique address of **`0x43C10100`**.

We will now take a look at the main software application file.

(q) Open the source file **adventures_with_ip.c** by double-clicking on it in *Project Explorer*.

This file contains the main function, and another function which implements an interactive menu that allows the user to control the system using keyboard commands via the terminal.

Take a moment to look over the file and note the function calls which are made.

In the **`main()`** function, the first set of functions are called to setup and configure the audio coded. These functions are defined in **audio.c**, which we will look at next.

(r) Open **audio.c**.

Here we have the functions which are called to initialise the audio codec and the required I^{2}C interface in the Zynq PS.

We don’t want to go into great detail about the functionality contained here, but in basic terms the purpose of these functions is to configure the audio codec by writing to the codec’s control registers.

Each control register has a unique address which can be accessed via the I^2C serial interface. The control register addresses are defined in the **audio.h** header file.

(s) Open **audio.h**.

This file contains a number of definitions relating to the audio codec and the I^2C interface, as well as some prototype function definitions.

You should see an enumerated type which lists all of the audio codec's control register addresses, which were mentioned in the previous step.

More information on the audio codec can be found in the data sheet:
http://www.analog.com/static/imported-files/data_sheets/ADAU1761.pdf

Next we will have a look at the functions which control the custom IP peripherals in the PL.

(t) Open **ip_functions.c**.

This file contains the functions which control the IP peripherals, as well as some functions to initialise drivers for the GPIO and NCO.

The three functions of interest are:

- **`audio_stream()`** — Implements stereo audio loopback between the input and output ports of the audio codec. Left and right audio samples are read in from the audio controller peripheral's I^2S receive register and are then written back out to the controller's I^2S transmit register.
- **`tonal_noise()`** — This function builds upon the audio loopback in **audio_stream()**. A step size value is input via the slide switches on the board. The corresponding value is then output to the LEDs on the board by writing to the memory-mapped register of the LED controller peripheral. The step size value is also output to the NCO peripheral using the **`XNco_SetStep_size_v()`** function defined by the NCO driver file. A sinusoidal sample created by the NCO peripheral is the read in by the **`XNco_GetSine_sample_v()`** NCO driver function and, as in the previous audio streaming function, left and right audio samples are received from the audio codec. The sinusoidal noise component is then added to the left and right audio samples before being written to the audio controller for output to the codec.

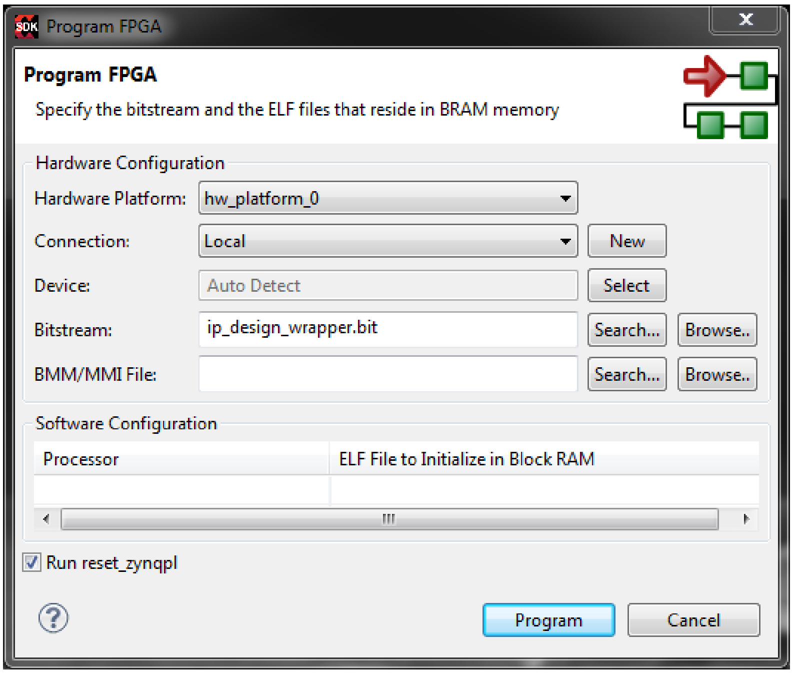
- **`lms_filter()`** — This function combines the functionality of the NCO and the LMS peripherals to create system which add tonal noise to an audio signal, before using an LMS adaptive filter for noise cancellation to remove the added noise. As in the **`tonal_noise()`** function, sinusoidal samples are generated from the NCO peripheral and added to the left and right audio samples from the audio controller. The sinusoidal sample is then input to the LMS as the input sample **`x(k)`** and the sample with added tonal noise is input as the desired signal **`d(k)`**. The resulting output of the LMS peripheral is only read if the user presses any of the push buttons on the board, otherwise the corrupted audio sample is retained. This allows the user to verify that the LMS filter peripheral is removing the noise.

Now that we have had a look at the functions and definitions contained in the various source and header files, we can move on to actually implementing the system on the ZedBoard.

To begin, we will program the Zynq PL with the bitstream that we generated in the previous exercise.

Note: At this stage ensure that the ZedBoard is powered on and both the **PROG** and **UART** USB ports are connected to your host computer.
You should also ensure that the board is configured to boot from **JTAG**.

(u) Select ***Xilinx Tools > Program FPGA*** from the *Menu Bar*. The Program FPGA window should be configured as in Figure 5.19.



Program FPGA

Specify the bitstream and the ELF files that reside in BRAM memory

Hardware Configuration

Hardware Platform: hw_platform_0

Connection: Local

Device: Auto Detect

Bitstream: ip_design_wrapper.bit

BMM/MMI File:

Software Configuration

Processor	ELF File to Initialize in Block RAM

☑ Run reset_zynqpl

Figure 5.19: Program FPGA window

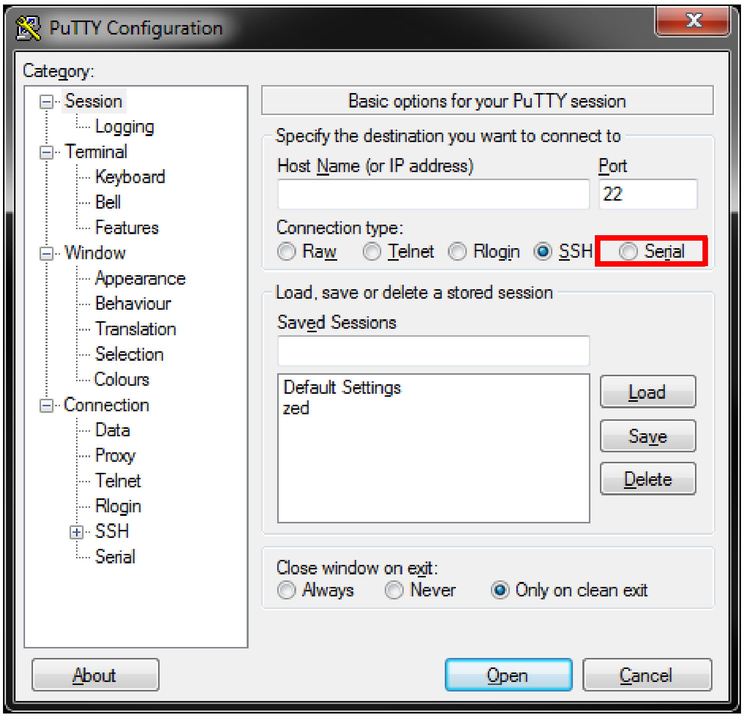
Click ***Program***.

The Zynq PL on the board will be configured with the bitstream and the **DONE** LED should turn blue.

At this stage we must invoke PuTTY — the terminal program which you should have downloaded at the beginning of this tutorial.

(v) At the location which you downloaded PuTTY, double-click ***PuTTY.exe***. As you downloaded the executable file, Windows may present a security warning. Accept the warning by clicking ***Run***.

(w) *PuTTY Configuration* should open, as shown in Figure 5.20.



Specify the destination you want to connect to

Host Name (or IP address):

Port: 22

Connection type: ○ Raw ○ Telnet ○ Rlogin ● SSH ○ Serial

Load, save or delete a stored session

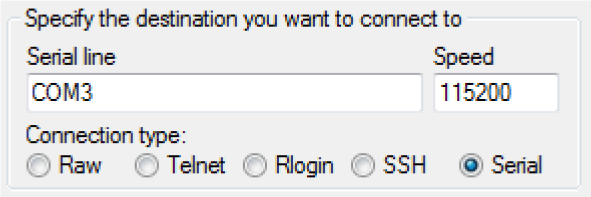
Saved Sessions

Default Settings
zed

Close window on exit: ○ Always ○ Never ● Only on clean exit

Figure 5.20: PuTTY

(x) Select ***Serial*** as *Connection type* (highlighted in Figure 5.20) and configure the settings as specified in Figure 5.21.



Specify the destination you want to connect to

Serial line: COM3

Speed: 115200

Connection type: ○ Raw ○ Telnet ○ Rlogin ○ SSH ● Serial

Figure 5.21: PuTTY configuration

NOTE: The value of the *Serial line* entry will vary depending on which the USB UART cable is connected to.

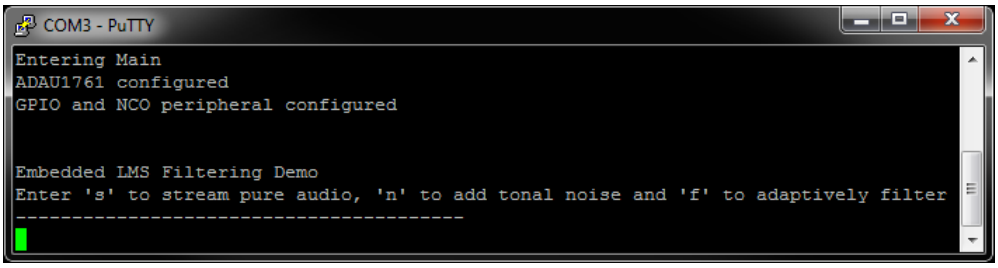
In order to determine this value on a Windows system, open the Device Manager and identify the COM port.

(y) Click ***Open***, to open the terminal connection. The PuTTY terminal window will open.

With the terminal connection open, the final step is the run the software on the Zynq PS.

(z) Right-click on ***adventures_with_ip*** in Project explorer and select ***Run As > Launch on Hardware (GDB)***.

In the PuTTY terminal you should see the following output:



```
Entering Main
ADAU1761 configured
GPIO and NCO peripheral configured


Embedded LMS Filtering Demo
Enter 's' to stream pure audio, 'n' to add tonal noise and 'f' to adaptively filter
----------------------------------------
```

Note: At this point you should attach an audio patch cable between the PC speaker output and the board's Line IN input. Also, connect headphones to the board's Line OUT input. These connections are highlighted in Figure 5.22.

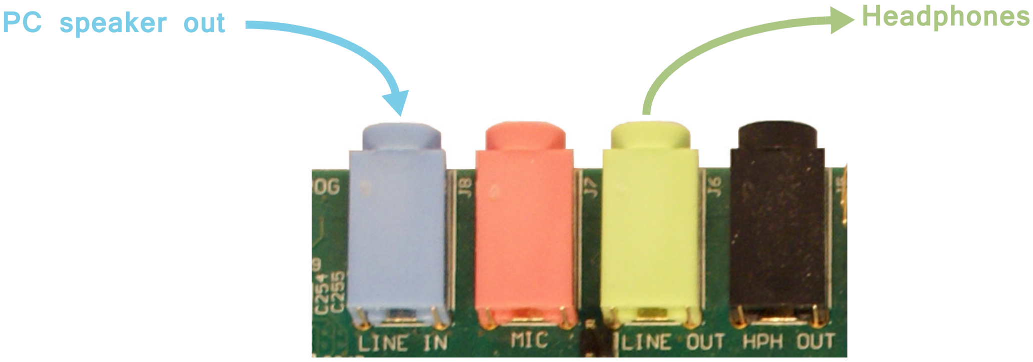


Figure 5.22: ZedBoard audio jacks

(aa) Open the audio file

C:\Zynq_Book\sources\adventures_with_ip_integrator\original_speech.wav

in an audio player, and begin playback.

Note: It may be useful to turn on the repeat setting in the audio player for continuous playback.

(ab) In the **PuTTY** terminal window, press the ***'s'*** key on your keyboard.

This will prompt the software application to enter the **`audio_stream()`** function which we looked at earlier.

You should be able to hear audio of speech via the headphone connection.

(ac) Press the ***'q'*** key on the keyboard to return to the **menu**.

(ad) Press the ***'n'*** key on the keyboard. This will prompt the application to enter the **`tonal_noise()`** function.

Initially you should hear the same audio signal.

You should note that currently there is no step size being input to the NCO.

Push slide switch **SW0** into the on position. You should now be able to hear a sinusoidal tone which has been added to the audio signal. **LED 0** should also be lit.

Experiment with different step size values by varying the on/off values of slide switches **SW1** and **SW2**. This will vary the frequency of the tonal noise.

(ae) Press the ***'q'*** key on the keyboard to return to the **menu**.

(af) Press the ***'f'*** key on the keyboard. This will prompt the application to enter the **`lms_filter()`** function. The basic functionality here is the same as in the previous NCO function, and you can add tonal noise to the audio signal using the slide switches.

With tonal noise being added to the audio signal, press any of the push buttons on the board. The sinusoidal tone will be adaptively filtered by the LMS, and the tonal noise removed.

This concludes this exercise on the creation of an audio application in the SDK. You should now be familiar with:

- The automatically generated **xparameters.h** header file, and its contents.
- Identifying memory-mapped base addresses and offsets for communication between software running on the Zynq PS and peripherals in the PL.
- The procedure of configuring the ZedBoard's ADAU1761 audio codec via the control register addresses.
- Receiving and sending audio samples to/from the audio codec via an audio controller block in the PL.
- The process of communicating with custom peripherals in the PL via generated software drivers.