Vivado Design Suite Tutorial

High-Level Synthesis

UG871 (v 2013.3) November 8, 2013





Notice of Disclaimer

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of the Limited Warranties which can be viewed at http://www.xilinx.com/warranty.htm; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in Critical Applications http://www.xilinx.com/warranty.htm#critapps.

©Copyright 2013 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
11/08/2013	2013.3	New Lab content details and editorial updates.
06/20/2013	2013.2	New Lab 2 added to Using HLS IP in a Zynq Processor Design.
04/03/2013	2013.2	New Release for Vivado Design Suite 2013.2.



Table of Contents

Revision History	2
Chapter 1 Tutorial Description	6
Overview	6
Software Requirements	7
Hardware Requirements	7
Obtaining the Tutorial Designs	
Preparing the Tutorial Design Files	8
Chapter 2 High-Level Synthesis Introductory Tutorial	9
Overview	9
Tutorial Design Description	9
HLS Lab 1: Creating a High-Level Synthesis Project	
HLS: Lab 2 Using the Tcl Command Interface	24
HLS: Lab 3: Using Solutions for Design Optimization	29
Chapter 3 C Validation	41
Overview	41
Tutorial Design Description	41
Lab 1: C Validation and Debug	42
Lab 2: C Validation with ANSI C Arbitrary Precision Types	50
Lab 3: C Validation with C++ Arbitrary Precision Types	55
Chapter 4 Interface Synthesis	60
Tutorial Design Description	60
Interface Synthesis Lab 1: Block-Level I/O protocols	62
Interface Synthesis Lab 2: Port I/O protocols	71
Interface Synthesis Lab 3: Implementing Arrays as RTL Interfaces	77
Interface Synthesis Lab 4: Implementing AXI Interfaces	92
Chapter 5 Arbitrary Precision Types	102
Overview	

Arbitrary Precision: Lab 1	
Arbitray Precision: Lab 2	
Chapter 6 Design Analysis	
Overview	
Tutorial Design Description	
Lab 1: Design Optimization	
Chapter 7 Design Optimization	
Overview	
Tutorial Design Description	
Lab 1: Optimizing a Matrix Multiplier	
Lab 2: C Code Optimized for I/O Accesses	
Conclusion	
Chapter 8 RTL Verification	
Overview	
Tutorial Design Description	
Lab 1: RTL Verification and the C test bench	
Lab 2: Viewing Trace Files in Vivado	
Lab 3: Viewing Trace Files in ModelSim	
Conclusion	
Chapter 9 Using HLS IP in IP Integrator	
Overview	
Tutorial Design Description	
Lab 1: Integrate HLS IP with a Xilinx IP Block	
Conclusion	
Chapter 10 Using HLS IP in a Zynq Processor Design	
Overview	
Tutorial Design Description	
Lab 1: Implement Vivado HLS IP on a Zynq Device	
Lab 2: Streaming data between the Zyng CPU and HLS Accelerator I	
Conclusion	

Chapter 11 Using HLS IP in System Generatorfor DSP......263

www.xilinx.com

4

Overview	263
Tutorial Design Description	263
Lab 1: Package HLS IP for System Generator	264
Conclusion	268



Chapter 1 Tutorial Description

Overview

This Vivado[®] tutorial is a collection of smaller tutorials that explain and demonstrate all steps in the process of transforming C, C++ and SystemC code to an RTL implementation using High-Level Synthesis.

High-Level Synthesis Introduction

This tutorial introduces Vivado High-Level Synthesis (HLS). You can learn the primary tasks for performing High-Level Synthesis using both the Graphical User Interface (GUI) and Tcl environments.

The tutorial shows how you create an initial RTL implementation and then you transform it into both a low-area and high-throughput implementation by using optimization directives without changing the C code.

C Validation

This tutorial reviews the aspects of a good C test bench and demonstrates the basic operations of the Vivado High-Level Synthesis C debug environment. The tutorial also shows how to debug arbitrary precision data types.

Interface Synthesis

The interface synthesis tutorial reviews all aspect of creating ports for the RTL design. You can learn how to control block-level I/O port protocols and port I/O protocols, how arrays in the C function can be implemented as multiple ports and types of interface protocol (RAM, FIFO, AXIA Stream), and how AXI4 bus interfaces are implemented.

The tutorial completes with a design example in which the I/O accesses and the logic are optimized together to create an optimal implementation of the design.

Arbitrary Precision Types

The lab exercises in this tutorial contrast a C design written in native C types with the same design written with Vivado High-Level Synthesis arbitrary precision types, showing how the latter improves the quality of the hardware results without sacrificing accuracy.

Design Analysis

This tutorial uses a DCT function to explain the features of the interactive design analysis features in Vivado High-Level Synthesis. The initial design takes you through a number of analysis and optimization stages that highlight all the features of the analysis perspective and provide the basis for a design optimization methodology.





Design Optimization

Using a matrix multiplier example, this tutorial reviews two-design optimization techniques. The first lab explains how a design can be pipelined, contrasting the approach of pipelining the loops versus pipelining the functions.

The tutorial shows you how to use the insights learned from analyzing to update the initial C code and create a more optimal implementation of the design.

RTL Verification

This tutorial shows how you can use the RTL cosimulation feature to verify automatically the RTL created by synthesis. The tutorial demonstrates the importance of the C test bench and shows you how to use the output from RTL verification to view the waveform diagrams in the Vivado and Mentor Graphics ModelSim simulators.

Using HLS IP in IP Integrator

This tutorial shows how RTL designs created by High-Level Synthesis are packaged as IP, added to the Vivado IP Catalog, and used inside the Vivado Design Suite.

Using HLS IP in a Zynq Processor Design

In addition to using an HLS IP block in a Zynq[®]-7000 SoC design, this tutorial shows how the C driver files created by High-Level Synthesis are incorporated into the software on the Zynq Processing System (PS).

Using HLS IP in System Generator for DSP

This tutorial shows how RTL designs created by High-Level Synthesis can be packaged as IP and used inside System Generator for DSP.

Software Requirements

This tutorial requires that the Vivado Design Suite 2013.3 release or later is installed.

Hardware Requirements

Xilinx recommends a minimum of 2 GB of RAM when using the Vivado tools.

Obtaining the Tutorial Designs

As shown in **Figure 1**, designs for the tutorial exercises are available as a zipped archive on the Xilinx Website, tutorial documentation page.

IMPORTANT: All the tutorial examples for Vivado High-Level Synthesis are available for download at http://www.xilinx.com/cgibin/docs/rdoc?v=20133;t=vivado+tutorials.

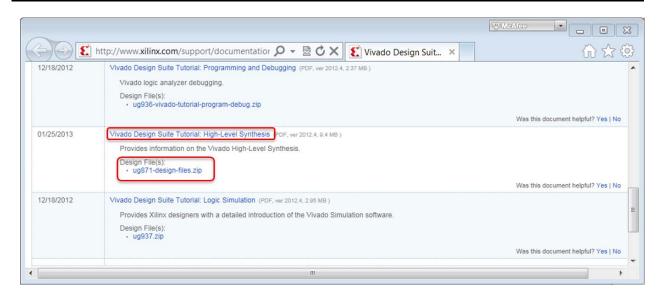


Figure 1: High-Level Synthesis Tutorial Design Files

Preparing the Tutorial Design Files

Extract the zip file contents into any write-accessible location.

This tutorial assumes that you have placed the unzipped design files in the location C:\Vivado_HLS_Tutorial.



IMPORTANT: If the Vivado_HLS_Tutorial directory is unzipped to a different location, or if it resides on Linux, adjust the pathnames to the location at which you have placed the Vivado_HLS_Tutorial directory.





Chapter 2 High-Level Synthesis Introductory Tutorial

Overview

This tutorial introduces Vivado[®] High-Level Synthesis (HLS). You can learn the primary tasks for performing High-Level Synthesis using both the Graphical User Interface (GUI) and Tcl environments.

The tutorial shows how use of optimization directives transforms an initial RTL implementation into both a low-area and high-throughput implementation.

Lab 1

Explains how to:

- Set up a High-Level Synthesis (HLS) project
- Perform all major steps in the HLS design flow:
 - o Validate the C code
 - o Create and synthesize a solution
 - Verify the RTL and package the IP.

Lab 2

Demonstrates how to use the Tcl interface.

Lab 3

Shows you how to optimize the design using optimization directives. This lab creates multiple versions of the RTL implementation and compares the different solutions.

Tutorial Design Description

To obtain the tutorial design file, refer to the section





Obtaining the Tutorial Designs.

This tutorial uses the design files in the tutorial directory Vivado_HLS_Tutorial\Introduction.

The sample design used in this tutorial is a FIR filter. The hardware goals for this FIR design project are:

• Create a version of this design with the highest throughput

The final design must process data supplied with an input valid signal and produce output data accompanied by an output valid signal. The filter coefficients are to be stored externally to the FIR design, in a single port RAM.

HLS Lab 1: Creating a High-Level Synthesis Project

Introduction

This lab shows how to create a High-Level Synthesis project, validate the C code, synthesize the design to RTL, and verify the RTL.



IMPORTANT: The figures and commands in this tutorial assume the tutorial data directory Vivado_HLS_Tutorial files are unzipped and placed in the location C: \Vivado_HLS_Tutorial.

Step 1: Creating a New Project

- 1. Open the Vivado[®] HLS Graphical User Interface (GUI):
 - On Windows systems, open Vivado HLS by double-clicking the Vivado HLS 2013.3 desktop icon.
 - On Linux systems, type vivado_hls at the command prompt.



Figure 2: The Vivado HLS Desktop Icon



TIP: You can also open Vivado HLS using the Windows menu **Start > All Programs > Xilinx Design Tools > Vivado 2013.3 > Vivado HLS > Vivado HLS 2013.3**.





Vivado HLS opens with the Welcome Screen as shown in Figure 3.



Figure 3: The Vivado Welcome Page

- 2. In the Welcome Page, select Create New Project to open the Project wizard.
- 3. As shown in Figure 4:
 - a. Enter the project name fir_prj.
 - b. Click **Browse** to navigate to the location of the lab1 directory.
 - c. Select the lab1 directory and click **OK**.
 - d. Click Next.

A New Vivado HLS Project	
Project Configuration	AG
Create Vivado HLS project of selected type	
Project name: fir_prj	
Location: C:\Vivado_HLS_Tutorial\Introduction\lab1	Browse
< Back Next > Finish	Cancel
< Back Next > Finish	Cancel

Figure 4: Project Configuration





This information defines the name and location of the Vivado HLS project directory. In this case, the project directory is fir_prj and it resides in the lab1 folder.

- 4. Enter the following information to specify the C design files:
 - a. Specify fir as the top-level function.
 - b. Click Add Files.
 - c. Select fir.c and click **Open**.
 - d. Click Next.

💫 New Vivado HLS I	Project	
Add/Remove File Add/remove C-bas	ed source files (design specification)	+
Top Function: fir Design Files		
Name	CFLAGS	Add Files
fir.c		New File
		Edit CFLAGS
		Remove
< Back	Next > Finish	Cancel

Figure 5: Project Design Files





IMPORTANT: In this lab there is only one C design file. When there are multiple C files to be synthesized, you must add all of them to the project at this stage. Any header files that exist in the local directory lab1 are automatically included in the project. If the header resides in a different location, use the **Edit CFLAGS** button to add the standard gcc/g++ search path information (for example, – I<path_to_header_file_dir>).

Figure 6 shows the input window for specifying the test bench files. The test bench and all files used by the test bench (except header files) must be included. You can add files one at a time, or select multiple files to add using the **Ctrl** and **Shift** keys.

\lambda New Vivado HLS	Project	- • •
Add/Remove File Add/remove C-bas	es sed testbench files (design test)	+
TestBench Files		
Name	CFLAGS	Add Files
fir_test.c		New File
📄 out.gold.dat		Add Folder
		Edit CFLAGS
		Remove
< Back	Next > Finish	Cancel

Figure 6: Test Bench Files

- 5. Click the **Add Files** button to include both test bench files: fir_test.c and out.gold.dat.
- 6. Click Next.





Both C simulation (and RTL cosimulation) execute in sub-directories of the solution.

If you do not include all the files used by the test bench (for example, data files read by the test bench, such as out.gold.dat), C and RTL simulation might fail after synthesis due to an inability to find the data files.

The Solution Configuration window (shown in **Figure 7**) specifies the technical specifications of the first solution.

A project can have multiple solutions, each using a different target technology, package, constraints, and/or synthesis directives.

💫 New Vivado HLS Project	- • ×
Solution Configuration Create Vivado HLS solution for selected technology	E
Solution Name: solution1 Clock Period: 10 Uncertainty: Part Selection Part: [Please select part]	
< Back Finish	Cancel

Figure 7: Solution Configuration

- 7. Accept the default solution name (**solution1**), clock period (**10 ns**) and clock uncertainty (defaults to 12.5% of the clock period, when left blank/undefined).
- 8. Click the part selection button [...] to open the part selection window.
- 9. Select **Device xc7k160tfbg484-2** from the list of available devices. Select the following from the dropdown filters to help refine the parts list:





- a. Product Category: General Purpose
- b. Family: Kintex®-7
- c. Sub-Family: Kintex-7
- d. Package: **fbg484**
- e. Speed Grade: -2
- f. Temp Grade: Any

10. Click **OK.**

In the Solution Configuration dialog box (shown in **Figure 7**, above), the selected part name now appears under the Part Selection heading.

11. Click Finish to open the Vivado HLS project, as shown in Figure 8.

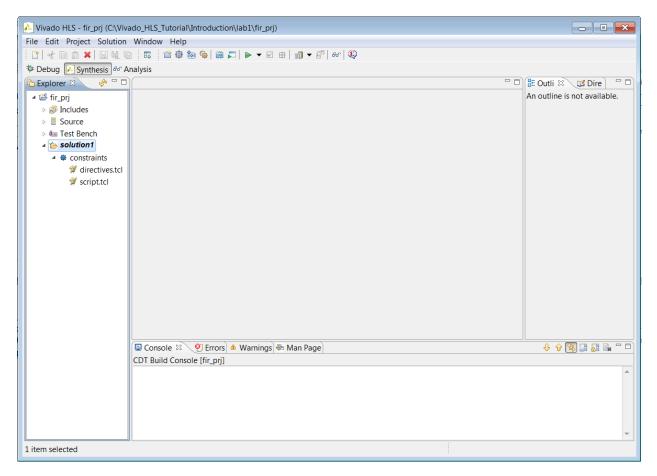


Figure 8: Vivado HLS Project

- The project name appears on the top line of the Explorer window.
- A Vivado HLS project arranges data in a hierarchical form.
- The project holds information on the design source, test bench, and solutions.
- The solution holds information on the target technology, design directives, and results.



15



• There can be multiple solutions within a project, and each solution is an implementation of the same source code.

TIP: At any time, you can change project or solution settings using the corresponding Project Settings and/or Solution Settings buttons in the toolbar.

Understanding the Graphical User Interface (GUI)

Before proceeding, review the regions in the Graphical User Interface (GUI) and their functions. **Figure 9** shows an overview of the regions, and each is described below.

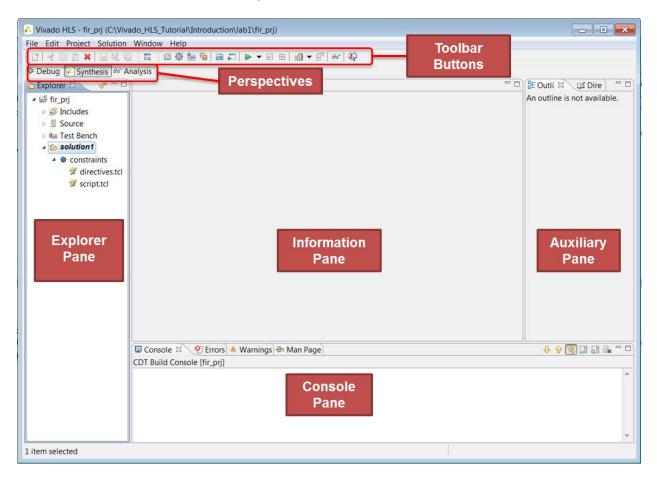


Figure 9: Vivado HLS Graphical User Interface

Explorer Pane

Shows the project hierarchy. As you proceed through the validation, synthesis, verification, and IP packaging steps, sub-folders with the results of each step are created automatically inside the solution directory (named csim, syn, sim, and impl respectively).

When you create new solutions, they appear inside the project hierarchy alongside solution1.





Information Pane

Shows the contents of any files opened from the Explorer pane. When operations complete, the report file opens automatically in this pane.

Auxiliary Pane

Cross-links with the Information pane. The information shown in this pane dynamically adjusts, depending on the file open in the Information pane.

Console Pane

Shows the messages produced when Vivado HLS runs. Errors and warnings appear in Console pane tabs.

Toolbar Buttons

You can perform the most common operations using the Toolbar buttons.

When you hold the cursor over the button, a popup dialog box opens, explaining the function. Each button also has an associated menu item available from the pulldown menus.

Perspectives

The perspectives provide convenient ways to adjust the windows within the Vivado HLS GUI.

Synthesis Perspective

The default perspective allows you to synthesize designs, run simulations, and package the IP.

Debug Perspective

Includes panes associated with debugging the C code. You can open the Debug Perspective after the C code compiles (unless you use the Optimizing Compile mode as this disable debug information).

Analysis Perspective

Windows in this perspective are configured to support analysis of synthesis results. You can use the Analysis Perspective only after synthesis completes.

Step 2: Validate the C Source Code

The first step in an HLS project is to confirm that the C code is correct. This process is called *C Validation* or *C Simulation*.

In this project, the test bench compares the output data from the fir function with known good values.

- 1. Expand the Test Bench folder in the Explorer pane.
- 2. Double-click the file fir_test.c to view it in the Information pane.
- 3. In the Auxiliary pane, select main() in the Outline tab to jump directly to the main() function.



Figure 10 shows the result of these actions

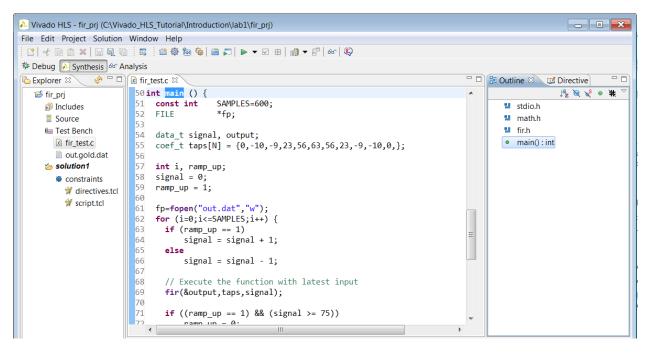


Figure 10: Reviewing the Test Bench Code

The test bench file, fir_test.c, contains the top-level C function main(), which in turn calls the function to be synthesized (fir). A useful characteristic of this test bench is that it is self-checking:

- The test bench saves the output from the fir function into the output file, out.dat.
- The output file is compared with the golden results, stored in file out.gold.dat.
- If the output matches the golden data, a message confirms that the results are correct, and the return value of the test bench main() function is set to 0.
- If the output is different from the golden results, a message indicates this, and the return value of main() is set to 1.

The Vivado HLS tool can reuse the C test bench to perform verification of the RTL.

HLS confirms the successful verification of the RTL if the test bench returns a value of 0. If any other value is returned by main(), including no return value, it indicates that the RTL verification failed.

If the test bench has the previously described self-checking characteristics, the RTL results are automatically checked during RTL verification. There is no requirement to create an RTL test bench. This provides a robust and productive verification methodology.

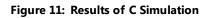
- 4. Click the **Run C Simulation** button, or use menu **Project > Run C Simulation**, to compile and execute the C design.
- 5. In the C Simulation dialog box, click **OK.**





The Console pane (Figure 11) confirms the simulation executed successfully.

😑 Console 🕺 🔮 Errors 💩 Warnings 🖶 Man Page	
Vivado HLS Console	
Generating csim.exe Comparing against output data ***********************************	•
PASS: The output matches the golden output!	
@I [SIM-1] CSim done with 0 errors. @I [LIC-101] Checked in feature [VIVADO_HLS]	E
4	•



TIP: If the C simulation failed, select the **Debug** option in the C Simulation dialog box, compile the design, and automatically switch to the Debug perspective. There you can use a C debugger to fix any problems

The C Validation tutorial module provides more details on using the Debug environment.

The design is now ready for synthesis.

Step 3: High-Level Synthesis

In this step, you synthesize the C design into an RTL design and review the synthesis report

1. Click the **Run C Synthesis** toolbar button or use the menu **Solution > Run C Synthesis**.

When synthesis completes, the report file opens automatically. Because the synthesis report is open in the Information pane, the Outline tab in the Auxiliary pane automatically updates to reflect the report information.

- 2. Click Performance Estimate in the Outline tab (Figure 12).
- 3. In the Details section of the Performance Estimates, expand the **Loop** view.





r_csynth.r	pt 🛛								
rformanc	e Estin	nates							
Timing (I	ıs)								
🗉 Summ	ary								
Clock	Tar	get Es	timate	d Unce	ertainty				
default	10	.00	7.1	3	1.25				
Later	-	Interv	al						
Summ	-	-							
min	max		max	Туре					
89	89	90	90	none					
Detail									
⊡ Inst ⊡ Loo									
_			1.2	tency		Initiation	Interval		
			La	tericy					
	oop N	ame	min	1	Iteration Latence	achieved	target	Trip Count	Pipelined

Figure 12: Performance Estimates

In the Performance Estimates pane, shown in **Figure 12**, you can see that the clock period is set to 10 ns. The estimated clock period (worst-case delay) is 8.43 ns. This includes an uncertainty of 1.25 ns. Therefore, the worst-case delay is roughly 8.43 - 1.25 = 7.18 ns.

The clock uncertainty ensures there is some timing margin available for the (at this stage) unknown net delays due to place and routing.

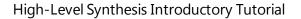
In the Summary section, you can see:

- The design has a latency of 89-clock cycles: it takes 89 clocks to output the results.
- The interval is 90 clock cycles: the next set of inputs is read after 89 clocks. This is one cycle after the final output is written. This indicates the design is not pipelined. The next execution of this function (or next transaction) can only start when the current transaction completes.
- The message "design is not pipelined" is also included under the pipelined type: no pipelining is performed.

The Details section shows:

- There are no sub-blocks in this design. Expanding the Instance section shows no submodules in the hierarchy.
- All the delay is due to the RTL logic synthesized from the loop named Shift_Accum_Loop. This logic executes 11 times (Trip Count). Each execution requires 8 clock cycles (Iteration Latency), for a total of 88 clock cycles, to execute all iterations of the logic synthesized from this loop (Latency).







• The total latency is one clock cycle greater than the loop latency. It requires one clock cycle to enter and exit the loop (in this case, the design finishes when the loop finishes, so there is no exit cycle).

ir_csynth.rpt 🛛									
ilization Estima	ites								
Summary									
Name	BRAM_18K	DSP48E	FF	LUT					
Expression	-	-	0	44					
FIFO	-	-	-	-					
Instance	-	4	45	21					
Memory	1	-	0	0					
Multiplexer	-	-	-	105					
Register	-	-	111	-					
ShiftMemory	-	-	-	-					
Total	1	4	156	170					
Available	650	600	202800	101400					
Utilization (%)	~0	~0	~0	~0					
Detail									
Instance									
Instar	nce	Mod	ule	BRAM_18K	DSP48E	FF	LUT		
fir_mul_32s_3	2s_32_6_U0	fir_mul_32s_	32s_32_6	0	4	45	21		
Total	Î		1	0	4	45	21		

4. In the Outline tab, click **Utilization Estimate** (Figure 13).

Figure 13: Utilization Estimates

5. In the **Details** section of the Utilization Estimates, expand the Instance view.

The design uses a single block RAM, 4 DSP48s, and approximately 150 flip-flops and LUTs. At this stage, the area numbers are estimates.

- RTL synthesis might be able to perform additional optimizations, and these figures might change after RTL synthesis.
- The number of DSP48s seems larger than expected for a FIR filter.
- The multiplier instance shown in the Instance view accounts for all the DSP48s.
- Because this multiplier appear in the Instance section and not the Expressions section, it must have been implemented as a pipelined multiplier. Standard combinational multipliers appear in the expressions section.

In **HLS:** Lab 3: Using Solutions for Design Optimization, you analyze these results further.

6. In the Outline tab, click **Interface** (Figure 14).





fir_test.c	🛾 🗊 fil	r_csyn	th.rpt 🛿			- 0)	🗄 Outl 🛛 🚺 Dire
						•	🗄 General Informat
iterface							Performance Est
Summary							🖺 Timing (ns)
	Dir	Bits	Protocol	Source Object	C Type		Latency (clock
ap_clk	in	1	ap_ctrl_hs	fir	return value		E Utilization Estima
ap_rst	in	1	ap_ctrl_hs	fir	return value		Summary
ap_start	in	1	ap_ctrl_hs	fir	return value		Detail
ap_done	out	1	ap_ctrl_hs	fir	return value		Summary
ap_idle	out	1	ap_ctrl_hs	fir	return value		Summary
ap_ready	out	1	ap_ctrl_hs	fir	return value		
у	out	32	ap_vld	у	pointer		
y_ap_vld	out	1	ap_vld	У	pointer		
c_address0	out	4	ap_memory	с	array		
c_ce0	out	1	ap_memory	с	array	≡	
c_q0	in	32	ap_memory	с	array		
x	in	32	ap_none	х	scalar		

Figure 14: Interface Report

The Interface section shows the ports and I/O protocols created by interface synthesis:

- The design has a clock and reset port (ap_clk and ap_reset). These are associated with the Source Object fir: the design itself.
- There are additional ports associated with the design as Source Object. Synthesis has automatically added some block level control ports: ap_start, ap_done, ap_idle and ap_ready.
- The Interface Synthesis tutorial provides more information about these ports.
- The function output y is now a 32-bit data port with an associated output valid signal indicator y_ap_vld.
- Function input argument c (an array) has been implemented as a block RAM interface with a 4-bit output address port, an output CE port and a 32-bit input data port.
- Finally, input argument x is simply implemented as a data port with no I/O protocol (ap_none).

Later in this tutorial, **HLS:** Lab 3: Using Solutions for Design Optimization explains how to optimize the I/O protocol for port x.

Step 4: RTL Verification

High-Level Synthesis can re-use the C test bench to verify the RTL using simulation.

- 1. Click the **Run C/RTL Cosimulation** toolbar button or use the menu **Solution > Run C/RTL Cosimulation**.
- 2. Click **OK** in the Co-simulation dialog box to execute the RTL simulation.

The default option for RTL Co-simulation is to perform the simulation using the SystemC RTL. This allows the simulation to run using the built-in C compiler. To perform the verification using Verilog and/or VHDL, select the HDL and choose the simulator from the drop-down menu. When RTL co-simulation completes, the report opens automatically in the





Information pane, and the Console displays the message shown in **Figure 15**. This is the same message produced at the end of C simulation.

- o The C test bench generates input vectors for the RTL design.
- o The RTL design is simulated.
- The output vectors from the RTL are applied back into the C test bench and the results-checking in the test bench verify whether or not the results are correct.





The RTL Verification tutorial (page 171) provides additional information.

Step 5: IP Creation

The final step in the High-Level Synthesis flow is to package the design as an IP block for use with other tools in the Xilinx Design Suite.

- 1. Click the **Export RTL** toolbar button or use the menu **Solution** > **Export RTL**.
- 2. Ensure the Format Selection dropdown menu shows IP Catalog.
- 3. Click **OK**.

The IP packager creates a package for the Vivado IP Catalog. (Other options available from the drop-down menu allow you to create IP packages for System Generator for DSP or a pcore for Xilinx Platform Studio.)

- 4. Expand **Solution1** in the Explorer.
- 5. Expand the **impl** folder created by the Export RTL command.
- 6. Expand the *ip* folder and find the IP packaged as a zip file, ready for adding to the Vivado IP Catalog (Figure 16).





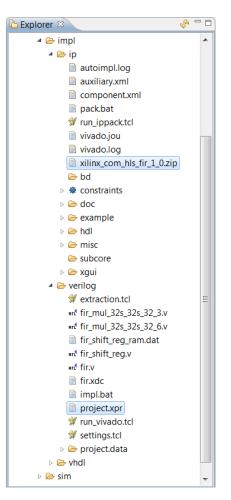


Figure 16: RTL Verification Results

Also note, in **Figure 16**, that if you expand the Verilog or VHDL folders inside the impl folder, there is a Vivado project ready for opening in the Vivado Design Suite.

RECOMMENDED: In this Vivado project, the HLS design is the top-level. This project provides an additional means of analyzing the design. The recommended approach is to add the IP package to the Vivado IP catalog, and add it as IP to the design that uses the HLS design.

Note: There is no project file created for devices synthesized by ISE (6 series or earlier devices).

At this stage, leave the Vivado HLS GUI open. You will return to this in the next lab exercise.

HLS: Lab 2: Using the Tcl Command Interface

Introduction

This lab exercise shows how to create a Tcl command file based on an existing Vivado HLS project and use the Tcl interface.

High-Level Synthesis UG871 (v 2013.3) November 8, 2013





Step 1: Create a Tcl file

- 1. Open the Vivado HLS Command Prompt.
- 2. On Windows, use Start > All Programs > Xilinx Design Tools > Vivado 2013.3 > Vivado HLS > Vivado HLS 2013.3 Command Prompt (Figure 17).
- 3. On Linux, open a new shell.

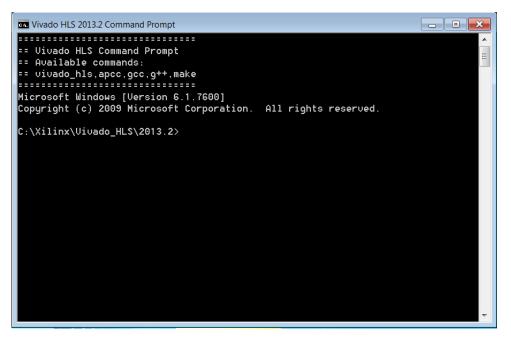


Figure 17: The Vivado HLS Command Prompt

When you create a Vivado HLS project, Tcl files are automatically saved in the project hierarchy. In the GUI still open from Lab 1, a review of the project shows two Tcl files in the project hierarchy (Figure 18).

4. In the GUI, still open from Lab 1, expand the Constraints folder in solution1 and double-click the file script.tcl to view it in the Information pane.





ि Explorer 🛛 🔗 🗖 🗖	🖋 script.tcl 🛛	
Explorer ☆ □ □ Fir_prj Includes Source Test Bench Solution1 Constraints directives.tcl Script.tcl csim sim syn	<pre>1 ####################################</pre>	
	19	* •

Figure 18: The Vivado HLS Project Tcl Files

- The file script.tcl contains the Tcl commands to create a project with the files specified during the project setup and run synthesis.
- The file directives.tcl contains any optimizations applied to the design. No optimization directives were used in Lab 1 so this file is empty.

In this lab exercise, you use the script.tcl from Lab 1 to create a Tcl file for the Lab 2 project.

- 5. Close the Vivado HLS GUI from Lab 1. This is project no longer needed.
- 6. In the Vivado HLS Command Prompt, use the following commands (also shown in **Figure 19**) to create a new Tcl file for Lab 2.
 - a. Change directory to the Introduction tutorial directory C:\Vivado_HLS_Tutorial\Introduction.
 - b. Use the command cp lab1\fir_prj\solution1\script.tcl lab2\run_hls.tcl to copy the existing Tcl file to Lab 2. (The Windows command prompt supports auto-completion using the Tab key: press the tab key repeatedly to see new selections).
 - c. Use the command cd lab2 to change into the lab2 directory.





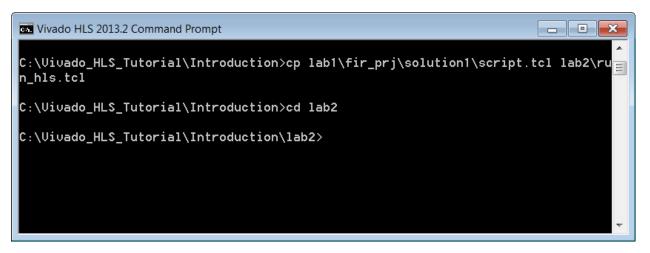
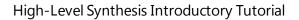


Figure 19: Copying the Lab 1 Tcl file to Lab 2

- d. Using any text editor, perform the following edits to the file run_hls.tcl in the lab2 directory. The final edits are shown in Figure 20.
 - i. Add a **-reset** option to the open_project command. Because you typically run Tcl files repeatedly on the same project, it is best to overwrite any existing project information.
 - ii. Add a **-reset** option to the open_solution command. This removes any existing solution information when the Tcl file is re-run on the same solution.
 - iii. Delete the source command. If a previous project contains any directives you wish to re-use, you can copy the directives.tcl file from that project to a local path, or you can copy the directives directly into this file.
 - iv. Add the exit command.
 - v. Save the file.





run hls.tcl 🗙

```
2 ## This file is generated automatically by Vivado HLS.
3 ## Please DO NOT edit it.
4 ## Copyright (C) 2013 Xilinx Inc. All rights reserved.
7 # Reset the project with the -reset option
8 open project -reset fir_prj
9 set top fir
10 add_files fir.c
11 add files -tb fir test.c
12 add files -tb out.gold.dat
13
14 # Reset the solution with the -reset option
15 open_solution -reset "solution1"
16 set_part {xc7k160tfbg484-2}
17 create clock -period 10
18
19 # Remove the link to any existing directives
20 #source "./fir prj/solution1/directives.tcl"
21
22 # If directives exist or are required, copy them into this file
23
24 csim design
25 csynth design
26 cosim_design -trace_level none
27 export_design -format ip_catalog -description "An IP generated by Vivado HLS" -ven
28
29 # Exit Vivado HLS
30 exit
31
```

Figure 20: Updated run_hls.tcl file for Lab 2

You can run the Vivado HLS in batch mode using this Tcl file.

e. In the Vivado HLS Command Prompt window, type vivado_hls _f run_hls.tcl.

Vivado HLS executes all the steps covered in lab1. When finished, the results are available inside the project directory fir_prj.

- The synthesis report is available in fir_prj\solution1\syn\report.
- The simulation results are available in fir_prj\solution\sim\report.
- The output package is available in fir_prj\solution1\impl\ip.
- The *final output RTL* is available in fir_prj\solution1\impl and then Verilog or VHDL.

CAUTION! When copying the RTL results from a Vivado HLS project, you must use the RTL from the *impl* directory.

For designs using floating-point operators or AXI4 interfaces, the RTL files in the syn directory are only the output from synthesis. Additional processing is performed by Vivado HLS during export_design before you can use this RTL in other design tools.



HLS: Lab 3: Using Solutions for Design Optimization

Introduction

This lab exercise uses the design from Lab 1 and optimizes it.

Step 1: Creating a New Project

- 1. Open the Vivado HLS Command Prompt.
 - a. On Windows, use Start > All Programs > Xilinx Design Tools > Vivado 2013.3 > Vivado HLS > Vivado HLS 2013.3 Command Prompt
 - b. On Linux, open a new shell.
- 2. Change to the Lab 3 directory: cd C:\Vivado_HLS_Tutorial\Introduction\lab3.
- 3. In the command prompt window, type: vivado_hls -f run_hls.tcl

This sets up the project.

4. In the command prompt window, type **vivado_hls -p fir_prj** to open the project in the Vivado HLS GUI.

Vivado HLS opens, as shown in Figure 21, with the synthesis for solution1 already complete.

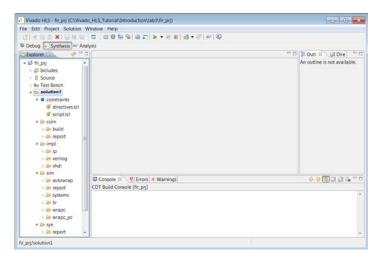


Figure 21: Introduction Lab 3 Initial Solution

As stated earlier, the design goals for this design are:

- Create a version of this design with the highest throughput
- The final design should be able to process data supplied with an input valid signal.
- Produce output data accompanied by an output valid signal.





• The filter coefficients are to be stored externally to the FIR design, in a single port RAM.

Step 2: Optimize the I/O Interfaces

Because the design specification includes I/O protocols, the first optimization you perform creates the correct I/O protocol and ports. The type of I/O protocol you select might affect what design optimizations are possible. If there is an I/O protocol requirement, you should set the I/O protocol as early as possible in the design cycle.

You reviewed the I/O protocol for this design in Lab 1 (Figure 14), and you can review the synthesis report again by navigating to the report folder inside the solution1\syn folder. The I/O requirements are:

- Port C must have a single port RAM access.
- Port X must have an input data valid signal.
- Port Y must have an output data valid signal.

Port C already is a single-port RAM access. However, if you do not explicitly specify the RAM access type, High-Level Synthesis might use a dual-port interface. HLS takes this action if the resulting design has higher throughput. Therefore, you should explicitly add to the design the I/O protocol requirement to use a single-port RAM.

Input port X is by default a simple 32-bit data port. You can implement it as an input data port with an associated data valid signal by specifying the I/O protocol ap_vld.

Output port Y already has an associated output valid signal. This is the default for pointer arguments. You do not have to specify an explicit port protocol for this port, since the default implementation is what is required

To preserve the existing results, create a new solution, solution2.

- 1. Click the **New Solution** toolbar button to create a new solution.
- 2. Leave the default solution name as solution2. Do not change any of the technology or clock settings.
- 3. Click Finish.

This creates solution2 and set it as the default solution - confirm that solution2 is highlighted in bold in the Explorer pane, indicating that it is the current active solution.

To add optimization directives to define the desired I/O interfaces to the solution, perform the following steps.

- 4. In the Explorer pane, expand the Source container in solution2 (as shown in Figure 22).
- 5. Double-click fir.c to open the file in the Information pane.
- 6. Activate the **Directives** tab in the Auxiliary pane and select the top-level function fir to jump to the top of the fir function in the source code view (Figure 22).





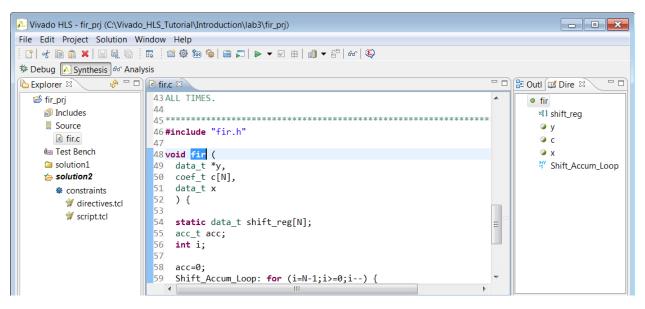


Figure 22: Opening the Directives Tab

The Directives tab, shown on the right side of **Figure 22**, lists all of the objects in the design that can be optimized. In the Directives tab, you can add optimization directives to the design. You can view the Directives tab only when the source code is open in the Information pane.

Apply the optimization directives to the design.

- 7. In the Directive tab, select the **c** argument/port (green dot).
- 8. Right-click and select **Insert Directives**.
- 9. Implement the single-port RAM interface by performing the following:
 - a. Select **RESOURCE** from the Directive drop-down menu.
 - b. Click the **core** box.
 - c. Select **RAM_1P_BRAM**, as shown in Figure 23.
 - d. Click **OK**.

The steps above specify that array c be implemented using a single-port block RAM resource. Because array c is in the function argument list, and hence is outside the function., a set of data ports are automatically created to access a single-port block RAM outside the RTL implementation.

Because I/O protocols are unlikely to change, you can add these optimization directives to the source code as pragmas to ensure that the correct I/O protocols are embedded in the design.

10. In the **Destination** section of the Directives Editor, select **Source File**.

11. To apply the directive, click **OK**.



Explorer 22 0 °	D fir_esynth.pt B fire 2 Shole risk and liability of any use of XI 39 subject only to applicable laws and regulatelity.		• 0	Outline III Directive II fir y
 Source If fact Test Bench Solution1 Solution2 e constraints f directives.tcl f' script.tcl 	Vivado HLS Directive Editor Type Directive: RESOURCE Destination Source File Options variable (required): c core (required): c core (required): RAM_IP_BRAM port map (optional): metadata (optional):	Wivado HLS core selectio. Image: Core selectio. Muli6S [functional_unit] MulinS [functional_unit] RAM_1P_BRAM [storage] RAM_2P_1UTRAM [storage] RAM_2P_1Storage] RAM_2P_1Storage] RAM_2P_1Storage] RAM_2P_1Storage] RAM_2P_1Storage] RAM_2P_1Storage] RAM_2P_1Storage] RAM_1P_BRAM [storage] RAM_1P_BRAM [storage] ROM_1P_1S[storage] ROM_2P_Storage] ROM_2P_Storage	Ħ	♀ c ● x *1 shift_reg ∜ Shift_Accum_Loop
	Help Cancel OK	OK Cancel		e

Figure 23: Adding a Resource Directive

- 12. Next, specify port **x** to have an associated valid signal/port.
 - a. In the **Directive** tab, select input port **x** (green dot).
 - b. Right-click and select Insert Directives.
 - c. Select **Interface** from the Directive Editor drop-down menu.
 - d. Select **Source File** from the **Destination** section of the dialog box
 - e. Select **ap_vld** as the mode.
 - f. Click **OK** to apply the directive.
- 13. Finally, explicitly specify port **y** to have an associated valid signal/port.
 - a. In the **Directive** tab, select input port **y** (green dot).
 - b. Right-click and select Insert Directives.
 - c. Select Source File from the Destination section of the dialog box
 - d. Select **Interface** from the Directive drop-down menu.
 - e. Select **ap_vld** for the mode.
 - f. Click **OK** to apply the directive

When complete, verify that the source code and the Directive are as shown in Figure 24. Rightclick any incorrect directive to modify it.





€ *fir.c ⊠	- 0)	🗄 Outline 🖾 Directive 🛛 👘 🗖
<pre>4 + include "fir.h" 47 48 void fir (49 data_t *y, 50 coef_t c[N], 51 data_t x 52) { 53 #pragma HLS INTERFACE ap_vld port=y 54 #pragma HLS INTERFACE ap_vld port=x 55 #pragma HLS RESOURCE variable=c core=RAM_1P_BRAM 56 57 static data_t shift_reg[N]; 58 acc_t acc; 59 int i; 60 61 acc=0; 62 Shift_Accum_Loop: for (i=N-1;i>=0;i) { </pre>		 E Outline us Directive & L fir *Il shift_reg y # HLS INTERFACE ap_vid port=y c # HLS RESOURCE variable=c core=RAM_1P_BRAM x # HLS INTERFACE ap_vid port=x * Shift_Accum_Loop

Figure 24: I/O Directives for solution2

- 14. Click the Run C Synthesis toolbar button to synthesize the design.
- 15. When prompted, click **Yes** to save the contents of the C source file. Adding the directives as pragmas modified the source code.

When synthesis completes, the report file opens automatically.

16. Click the **Outline** tab to view the Interface results, or simply scroll down to the bottom of the report file.

Figure 25 shows the ports now have the correct I/O protocols.

Summary					
	Dir	Bits	Protocol	Source Object	С Туре
ap_clk	in	1	ap_ctrl_hs	fir	return value
ap_rst	in	1	ap_ctrl_hs	fir	return value
p_start	in	1	ap_ctrl_hs	fir	return value
p_done	out	1	ap_ctrl_hs	fir	return value
p_idle	out	1	ap_ctrl_hs	fir	return value
p_ready	out	1	ap_ctrl_hs	fir	return value
r	out	32	ap_vld	у	pointer
_ap_vld	out	1	ap_vld	у	pointer
_address0	out	4	ap_memory	с	array
_ce0	out	1	ap_memory	с	array
_q0	in	32	ap_memory	с	array
c .	in	32	ap_vld	x	scalar
_ap_vld	in	1	ap_vld	x	scalar

Figure 25: I/O Protocols for solution2



Step 3: Analyze the Results

Before optimizing the design, it is important to understand the current design. It was shown in Lab 1 how the synthesis report can be used to understand the implementation, however, the Analysis perspective provides greater detail in an inter-active manner.

While still in solution2, and as shown in Figure 26:

- 1. Click the Analysis perspective button.
- 2. Click the Shift_Accum_Loop in the Performance window to expand it.
- The red-dotted line in Figure 26 is used shortly in an explanation; it is not part of the view.
- The tutorial **Design Analysis** provides a more complete understanding of the Analysis perspective, but the following explains what is required to create the smallest and fastest RTL design from this source code.
- The left column of the Performance pane view shows the operations in this module of the RTL hierarchy.
- The top row lists the control states in the design. Control states are the internal states High-Level Synthesis uses to schedule operations into clock cycles. There is a close correlation between the control states and the final states in the RTL Finite State Machine (FSM), but there is no one-to-one mapping.

×		t Soluti																	
Debug	Syr	thesis	Analys	is															
Modu	le Hierar	chy						- 0	Pe Pe	erformance - fir 🛛									6
	BRAM	DSP FI	LUT	Laten	cy Inte	rval Pipeline type			0	urrent Module : fir									
• fir	1	4 189 204 89 90 none					Cuttent Moute . 111												
										Operation\Control S	CO	C1	C.2	C.3	C4	C.5	C.6	C.7	C8
									1	x read(read)									
										Shift Accum Loop		- î-			_	_			
Berformance Profile 12 E Perource Profile								3	tmp 1(icmp)	-		-							
Performance Profile 🛛 📜 Resource Profile 👘 🗖									4	tmp 2(+)		1	-						
			Pipeli	ned L	atency	Initiation Interval	Iteration Latency	Trip count	5										
• fir				8	9	90	2	14	6										
	Shift Ac	cum_Loo	n no	8	8	-	8	11	7	node 36(write)									
							2	200	8	c load(read)	-	1							
									9	tmp 6(*)								_	-
									10			-		_	-		-		1
									11			-	-	_					_
									12	node 48 (write)			<-	_					-
									-	· · · · · · · · · · · · · · · · · · ·									
										ormance Resource									

Figure 26: Solution2 Analysis Perspective: Performance

The explanation presented here follows the path of the dotted red line in **Figure 26**. Some of the objects here correlate directly with the C source code. Right-click the object to cross-reference with the C code.

- The design starts in the first state with a read operation on port x.
- In the next state, it starts to execute the logic created by the for-loop Shift_Accum_Loop. Loops are shown in yellow, and you can expand or collapse them. Holding the cursor over



34



the yellow loop body in this view shows the loop details: 8 cycles, 11 iterations for a total latency of 88.

- In the first state, the loop iteration counter is checked: addition, comparison, and a potential loop exit.
- There is a two-cycle memory read operation on the block RAM synthesized from array data (one cycle to generate the address, one cycle to read the data).
- There are memory reads on the c port.
- A multiplication operations each takes 6 cycles to complete.
- The for-loop is executed 11 times.
- At the end of the final iteration, the loop exits in state c1 and the write to port y occurs.

You can also use the Analysis perspective to analyze the resources used in the design.

- 3. Click the **Resource** view, as shown in Figure 27.
- 4. Expand all the resource groups (also shown in Figure 27).

			1							1.000
	Resource\Control Step	<u>C.0</u>	<u>C1</u>	C.2	C.3	C.4	C.5	<u>C6</u>	C.7	<u>C8</u>
1	BI/O Ports		1							
2	С									
3	х	read								
4	У		write							
5	Instances									
6	grp fu 184							•		
7	BMemory Ports									
8	shift reg		write	write						
9	С			re	ad					
10	Expressions									
11	tmp 1 fu 149		icmp							
12	tmp 2 fu 159		+				Ì			
13	i 1 fu 178			+						
14	acc 1 fu 190									+

Figure 27: Solution2 Analysis Perspective: Resource

Figure 27 shows:

- The reads on the ports x and y. Port c is reported in the memory section because this is also a memory port.
- There are two multipliers being used in this design.
- There is a read and write operation on the memory shift_reg.
- None of the other resources are being shared because there is only one instance of each operation on each row or clock cycle.





With the insight gained through analysis, you can proceed to optimize the design.

Before concluding the analysis, it is worth commenting on the multi-cycle multiplication operations, which require multiple DSP48s to implement. The source code uses an int data-type. This is a 32-bit data-type that results in large multipliers. A DSP48 multiplier is 18-bit and it requires multiple DSP48s to implement a multiplication for data widths greater than 18-bit.

The tutorial **Arbitrary Precision Types** shows how you can create designs with more suitable data types for hardware. Use of arbitrary precision types allows you to define data types of any arbitrary bit size.(more than the standard C/C + + 8-, 16-, 32- or 64-bit types).

Step 4: Optimize for the Highest Throughput (lowest interval)

The two issues that limit the throughput in this design are:

- The for loop. By default loops are kept rolled: one copy of the loop body is synthesized and re-used for each iteration. This ensures each iteration of the loop is executed sequentially. You can unroll the for loop to allow all operations to occur in parallel.
- The block RAM used for shift_reg. Because the variable shift_reg is an array in the C source code, it is implemented as a block RAM by default. However, this prevents its implementation as a shift-register. You should therefore partition this block RAM into individual registers.

Begin by creating a new solution.

- 1. Click the **New Solution** button.
- 2. Leave the solution name as solution3.
- 3. Click **Finish** to create the new solution.
- 4. In the Project menu, select **Close Inactive Solution Tabs** to close any existing tabs from previous solutions.

The following steps, summarized in Figure 28 explain how to unroll the loop.



36



ld fir.c ⊠		- 0	🗄 Outline 🗹 Directive 🛛 🖓 🗖
<pre> firc ☆ firc ☆ forc ☆ coef_t c[N], t data_t x coef_t ata_t x coef_t c[N], fit coef_t ata_t coef_t coef_t coef_t coef_t coef_t coef_t coef_t coef_t coef_t coef_t coef_t coef_</pre>	Destination Source File Directive File Options skip exit check:		E Outline C Directive C fir y HLS INTERFACE ap_vld port=y c HLS RESOURCE variable=c core=RAM_1P_BRAM x HLS INTERFACE ap_vld port=x II shift_reg % Shift_Accum_Loop
63 data 64 } else { 65 shir 66 data 67 } 68 acc+=data*c 69 } 70 *y=acc; 71 } 72	region:		

Figure 28: Unrolling FOR Loop

- 5. In the Directive tab, select loop **Shift_Accum_Loop**. (Reminder: the source code must be open in the Information pane to see any code objects in the Directive tab).
- 6. Right-click and select Insert Directives.
- 7. From the Directive drop-down menu, select Unroll.

Leave the Destination as the Directive File.

When optimizing a design, you must often perform multiple iterations of optimizations to determine what the final optimization should be. By adding the optimizations to the directive file, you can ensure they *are not* automatically carried forward to the next solution. Storing the optimizations in the solution directive file allows different solutions to have different optimizations. Had you added the optimizations as pragmas in the code, they would be automatically carried forward to new solutions, and you would have to modify the code to go back and re-run a previous solution.

Leave the other options in the Directives window unchecked and blank to ensure that the loop is fully unrolled.

- 8. Click **OK** to apply the directive.
- 9. Apply the directive to partition the array into individual elements.
- a) In the Directive tab, select array shift_reg.
- b) Right-click and select Insert Directives.
- c) Select **Array_Partition** from the Directive drop-down menu.
- d) Specify the type as **complete**.
- e) Select **OK** to apply the directive.

www.xilinx.com





With the directives embedded in the code from solution2 and the two new directives just added, the directive pane for solution4 appears as shown in Figure 29.

E Outline 🕼 Directive 🛛 📃	
• fir	
×I1 shift_reg	
% HLS ARRAY_PARTITION variable=shift_reg complete dim=1	
У	
# HLS INTERFACE ap_vld register port=y	
# HLS RESOURCE variable=c core=RAM_1P_BRAM	
# HLS INTERFACE ap_vld port=x	
Shift_Accum_Loop	
% HLS UNROLL	

Figure 29: Solution4 Directives

In Figure 29, notice the directives applied in solution2 as pragmas have a different annotation (#HLS) than those just applied and saved to the directive file (%HLS). You can view the newly added directives in the Tcl file.

10. In the Explorer pane, expand the **Constraint** folder in Solution4 as shown in Figure 30.

11. Double-click the solution4 directives.tcl file to open it in the Information pane.

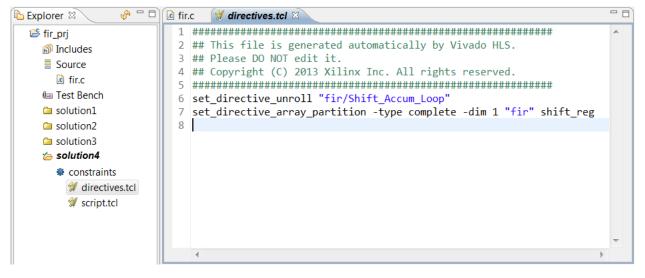


Figure 30: Solution4 Directives.tcl File

12. Click the **Synthesis** toolbar button to synthesize the design.

When synthesis completes, the synthesis report automatically opens.



- 13. Compare the results of the different solutions.
- 14. Click the **Compare Reports** toolbar button.

Alternatively, use **Project > Compare Reports**.

- 15. Add solution1, solution2, and solution3 to the comparison.
- 16. Click **OK**.

Figure 31 shows the comparison of the reports. solution3 has the smallest initiation interval and can process data much faster. As the interval is only 18, it starts to process a new set of inputs every 18 clock cycles.

🖥 compare r	reports	x								
Performan	ce Esti	mates								*
Timing	(ns)									
Clock			solution1 so		soluti	solution2 sol		olution3		
default	Target		10.00		10.00		10.00			
	Estimated		7.13		7.13		8.43			
Latency	(clock	cycles)							
		solution1 solution2 solution3								
Latency	min	min 89		89		15				
	max 89			89		15				
Interval	min	90		90		16				Ξ
	max			90		16				_
Utilization	Estima	tes								
	solution1 solution2 solution3									
BRAM_18	3К 1		1		0					
DSP48E			4		44					
FF	15	56	18	9	97	8				
LUT	17	70	204	4	38	5				-

Figure 31: Solution Comparisons

It is possible to perform additional optimizations on this design. For example, you could use Pipelining to further improve the throughput and lower the interval. The tutorial **Design Optimization** provides details on using pipelining to improve the interval.

As mentioned earlier, you could modify the code itself to use arbitrary precision types. For example, if the data types are not required to be 32-bit int types, you could use bit-accurate types (for example, 6-bit, 14-bit or 22-bit types), provided that they satisfy the required accuracy. For more details on using arbitrary precision type see the tutorial **Arbitrary Precision Types**.





Conclusion

In this tutorial, you learned how to:

- Create a Vivado High-Level Synthesis project in the GUI and Tcl environments.
- Execute the major steps in the HLS design flow.
- Create and use a Tcl file to run Vivado HLS.
- Create new solutions, add optimization directives, and compare the results of different solutions.





Chapter 3 C Validation

Overview

Validation of the C algorithm is an important part of the High-Level Synthesis (HLS) process. The time spent ensuring the C algorithm is performing the correct operation and creating a C test bench, which confirms the results are correct, reduces the time spent analyzing designs which are incorrect "by design" and ensures the RTL verification can be performed automatically.

This tutorial consists of three lab exercises.

- Lab1: Review the aspects of a good C test bench, the basic operations for C validation and the C debugger.
- Lab2: Validate and debug a C design using arbitrary precision C types.
- Lab3: Validate and debug a design using arbitrary precision C++ types.

Tutorial Design Description

You can download the tutorial design file from the Xilinx website. See the information in **Obtaining the Tutorial Designs**.

This tutorial uses the design files in the tutorial directory Vivado_HLS_Tutorial\C_Validation.

The sample design used in this tutorial is a Hamming Window FIR. There are three versions of this design:

- Using native C data types.
- Using ANSI C arbitrary precision data types.
- Using C++ arbitrary precision data types.

This tutorial explains the operation and methodology for C validation using High-Level Synthesis. There are no design goals for this tutorial.

www.xilinx.com



Lab 1: C Validation and Debug

Overview

This exercise reviews the aspects of a good C test bench and explains the basic operations of the High-Level Synthesis C debug environment.



IMPORTANT: The figures and commands in this tutorial assume the tutorial data directory Vivado_HLS_Tutorial is unzipped and placed in the location C:\Vivado_HLS_Tutorial. If the tutorial data directory is unzipped to a different location, or on Linux systems, adjust the few pathnames referenced, to the location you have chosen to place the Vivado HLS_Tutorial directory.

Step 1: Create and Open the Project

- 1. Open the Vivado HLS Command Prompt.
 - a. On Windows use Start > All Programs > Xilinx Design Tools > Vivado 2013.3 > Vivado HLS > Vivado HLS 2013.3 Command Prompt (Figure 32).
 - b. On Linux, open a new shell.



Figure 32: Vivado HLS Command Prompt

- 2. Using the command prompt window (Figure 33), change the directory to the C Validation tutorial, lab1.
- 3. Execute the Tcl script to setup the Vivado HLS project, using the command **vivado_hls_frun_hls.tcl** as shown in 33**Figure 33**.



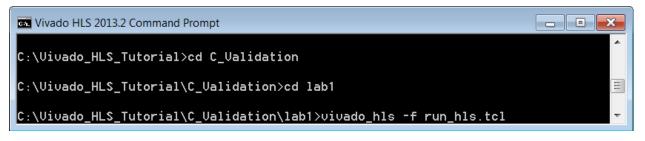


Figure 33: Setup the Tutorial Project

4. When Vivado HLS completes, open the project in the Vivado HLS GUI using the command **vivado_hls – p hamming_window_prj** as shown in **Figure 34**.

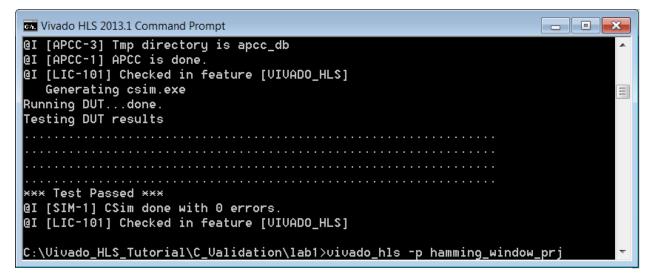


Figure 34: Initial Project for C Validation Lab 1





Step 2: Review Test Bench and Run C Simulation

1. Open the C test bench for review by double-clicking **hamming_window.c** in the Test Bench folder (Figure 35).

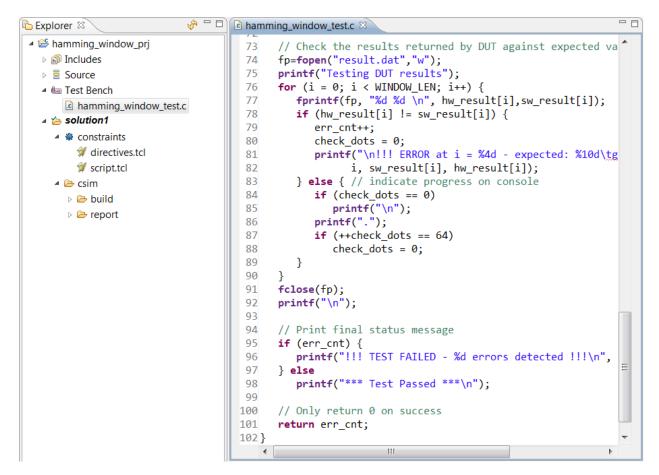


Figure 35: C Test Bench for C Validation Lab 1

A review of the test bench source code shows the following good practices:

- The test bench:
 - o Creates a set of expected results that confirm the function is correct.
 - Stores the results in array sw_result.
- The Design Under Test (DUT) is called to generate results, which are stored in array hw_result.. Because the synthesized functions use the hw_result array, it is this array that holds the RTL-generated results later in the design flow.
- The actual and expected results are compared. If the comparison fails, the value of variable err_cnt is set to a non-zero value.
- The test bench issues a message to the console if the comparison failed, but more importantly returns the results of the comparison. If the return value is zero the test bench validates the results are good.





This process of checking the results and returning a value of zero if they are correct automates RTL verification.

You can execute the C code and test bench to confirm that the code is working as expected.

2. Click the **Run C Simulation** toolbar button to open the C Simulation Dialog box, shown in **Figure 36**.

💫 C Simulation Dialog		×
C Simulation		P
Options Debug Build Only Clean Build Optimizing Compile		
Input Arguments		
	ОК	Cancel

Figure 36: Run C Simulation Dialog box

3. Select **OK** to run the C simulation.

As shown in Figure 37, the following actions occur when C simulation executes:

- The simulation output is shown in the Console window.
- Any print statements in the C code are echoed in the Console window. This example shows the simulation passed correctly.





• The C simulation executes in the solution sub-directory csim. You can find any output from the C simulation in the build folder, which is the location at which you can see the output file result.dat written by the fprintf command.

Because the C simulation is not executed in the project directory, you must add any data files to the project as C test bench files (so they can be copied to the csim/build directory when the simulation runs). Such files would include, for example, input data read by the test bench.

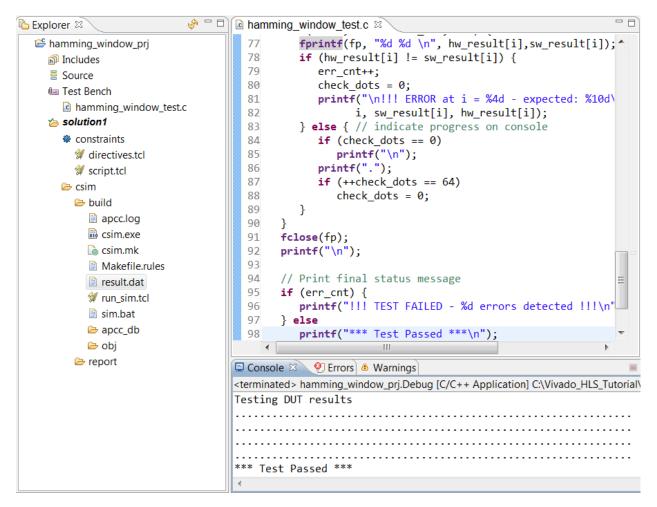


Figure 37: C Simulation Results

Step 3: Run the C Debugger

A C debugger is included as part of High-Level Synthesis.

- 1. Click the Run C Simulation toolbar button to open the C Simulation Dialog box.
- 2. Select the **Debug** option as shown in Figure 38.
- 3. Click **OK** to run the simulation.



E XILINX.

🔁 C Simulation Dialog	×
C Simulation	
Options Debug Build Only Clean Build Optimizing Compile 	
Input Arguments	
OK Cancel	

Figure 38: C Simulation Dialog Box

www.xilinx.com

The Debug option compiles the C code and then opens the Debug environment, as shown in **Figure 39**. Before proceeding, note the following:

- Highlighted at the top-left in Figure 39, you can see that the perspective has changed from Synthesis to Debug. Click the perspective buttons to return to the synthesis environment at any time.
- By default, the code compiles in debug mode. The Debug option automatically opens the debug perspective at time 0, ready for debug to begin. To compile the code without debug information, select the Optimizing Compile option in the **C Simulation** dialog box.

Debug 😰 🔥 Explorer 💥 ピ 📭 💷 🕷 🧟 😤 🧟 🗟 😾 🛒 🏱 🗖 🖸 hamming_window_prj.Debug [C/C++ Application]	⋈= Variables 🛛 🍳	Breakpoints III Registers	Modules 5 ≪ ⊟ # × %	-
C:\Vivado_HLS_Tutorial\C_Validation\lab1\hamming_window_prj\solution	Name	Туре	Value	
If Thread [1] 0 (Suspended : Breakpoint)	(x)= argc	int	1	
≡ main() at hamming_window_test.c:54 0x40139d	argv	char **	0x5619a0	
📓 gdb	🥭 test_data	in_data_t [256]	0x28fd0c	
	•	III		P.
Ш				F.
hamming window test.c		🗝 🗖 🔚 Outline	x 1ª x x° • ¥	~ =
54 unsigned err_cnt = 0, check_dots = 0; 55 FILE *fp; 56 *for (i = 0; i < WINDOW_LEN; i++) { 57 57 for (i = 0; i < WINDOW_LEN; i++) { 58 58 // Generate a test pattern for input to DUT	₹ 16) ₋ 9) / 9 6	e () + 0.		
<pre>59 test_data[i] = (in_data_t)((32767.0 * (double)((i 60 // Calculate the coefficient value for this index</pre>	in the second second	-		
59 test_data[i] = (in_data_t)((32767.0 * (double)((i	in the second second			
<pre>59 test_data[i] = (in_data_t)((32767.0 * (double)((i 50 // Calculate the coefficient value for this index 51 in_data_t coeff_val = (in_data_t)(WIN_COEFF_SCALE</pre>	in the second second	•	• × % • • • •	<u></u>

Figure 39: The HLS Debug Perspective

You can use the **Step Into** button (Figure 40) to step through the code line-by-line.



Figure 40: The Debug Step Into Button

4. Expand the Variables window to see the sw_results array.





- 5. Expand the sw_results array to the view shown in Figure 41.
- 6. Click the **Step Into** button (or key F5) repeatedly until you see the values being updated in the Variables window.

Image: Comparison of the second se			5 📲 🖻 🖉 🗙 🔆 I	🖞 🛃 🗋
	tic Name	Туре	Value	-
Thread [1] 0 (Suspended : Step)	🥭 test_data	in_data_t [256]	0x28fd0c	8
main() at hamming_window_test.c:57 0x4014a9	/ hw_result	out_data_t [256]	0x28f90c	
📕 gdb	🥭 sw_result	out_data_t [256]	0x28f50c	
	⇔ sw_result[0]	out_data_t	-42923460	
	(x): sw_result[1]	out_data_t	-37643710	
	(×)= sw_result[2]	out_data_t	-32413106	
	(x)= sw_result[3]	out_data_t	-27256218	
	(x)= sw_result[4]	out_data_t	2684268	
	ME ON FACULTS1	t etch tuo	2002627527	-
		III		•
				-
(•			•
		- D E Outline	x 1 1 × 1 × • #	
hamming_window_test.c 🛙	4	▲ stdi	io.h	
a hamming_window_test.c ।≋	4	stdi han	io.h nming_window.h	
6 hamming_window_test.c ☆ 56 57 for (i = 0; i < WINDOW_LEN; i++) {		stdi han mai	io.h	
a hamming_window_test.c ☆ 56 57 for (i = 0; i < WINDOW_LEN; i++) { 58 // Generate a test pattern for input to DUT	i % 16) - 8) / 8.0)	stdi han mai	io.h nming_window.h	
<pre>Ammming_window_test.c ☆ for (i = 0; i < WINDOW_LEN; i++) { for (i = 0; i < WINDOW_LEN; i++) { for (input to DUT for input to DUT for input to DUT for inst_data[i] = (in_data_t)((32767.0 * (double))(for input to DUT) </pre>	i % 16) - 8) / 8.0) × E * (0.54 -	stdi han mai	io.h nming_window.h	

Figure 41: Analysis of C Variables

In this manner, you can analyze the C code and debug it if the behavior is incorrect.

For more detailed analysis, to the right of the Step Into button are the Step Over (F6), Step Return (F7) and the Resume (F8) buttons.

- 7. Scroll to line 69 in the source code window.
- 8. Double-click in the left margin to create a breakpoint (blue dot), as shown in Figure 42.
- 9. Activate the Breakpoints tab, also shown in **Figure 42**, to confirm there is a breakpoint set at line 69.
- 10. Click the **Resume** button (highlighted in **Figure 42**) or the F8 key to execute up to the breakpoint.

www.xilinx.com





🌣 Debug 🖄 🔥 Explorer 🕷 ₩ 🕪 🔳 🕷 🖄 🐟 🦟 🧮 i 🏕 🕱 ▽ 🖓 Debug 🔯 Variables ▲ 🖸 hamming_window_prj.Debug [C/C++ Application]	ତ Breakpoints 🖄 🔠 Registers 🛋 Modules 🛛 🖓 🗖 🗮 🖻 😓
 ⁽⁴⁾ C:\Vivado_HLS_Tutorial\C_Validation\lab1\hamming_window_prj\soluti ⁽⁴⁾ M Thread [1] 0 (Suspended : Breakpoint) ⁽⁴⁾ main() at hamming_window_test.c:69 0x4014c1 ⁽⁵⁾ gdb 	nming_window_test.c [line: 69]
No details to	o display for the current selection.
<pre>hamming_window_test.c ©mingw_CRTStartup() at ./mingw/crt1.c:250 0x4010bb</pre>	_val;
<pre>>> dp printf("Running DUT"); hamming_window(hw_result, test_data); printf("done.\n"); 72 73 // Check the results returned by DUT against expected values 74 fp=fopen("result.dat","w");</pre>	
<pre>74 ip-iopen(resultdur, w); 75 printf("Testing DUT results"); 76 (</pre>	

Figure 42: Using Breakpoints

- 11. Click the **Step Into** button (or key F5) multiple times to step into the hamming_window function.
- 12. Click the **Step Return** button (or key F7) to return to the main function.
- 13. Click the red **Terminate** button to end the debug session.

The Terminate button becomes the Run C Simulation button. You can restart the debug session from within the Debug perspective.

14. Exit the Vivado HLS GUI and return to the command prompt.

Lab 2: C Validation with ANSI C Arbitrary Precision Types

Introduction

This exercise uses a design with arbitrary precision C types. You will review and debug the design in the GUI.

Step 1: Create and Open the Project

- 1. From the Vivado HLS command prompt used in Lab 1, change to the **lab2** directory, as shown in **Figure 43**.
- 2. To create a new Vivado HLS project, type **vivado_hls _f run_hls.tcl**.

High-Level Synthesis UG871 (v 2013.3) November 8, 2013 www.xilinx.com

Send Feedback



Tivado HLS 2013.1 Command Prompt
for user 'duncanm' on host 'xsjduncanm-w7' (Windows NT_intel version 6.1) on Thu Mar 07 14:02:06 -0800 2013 in directory 'C:/Vivado_HLS_Tutorial/C_Validation/lab1' @I [HLS-10] Bringing up Vivado HLS GUI
C:\Vivado_HLS_Tutorial\C_Validation\lab1>cd
C:\Vivado_HLS_Tutorial\C_Validation>cd lab2
C:\Vivado_HLS_Tutorial\C_Validation\lab2>vivado_hls -f run_hls.tcl

Figure 43: Setup for Interface Synthesis Lab 2

- 3. To open the Vivado HLS GUI project, type vivado_hls -p hamming_window_prj.
- 4. Open the Source folder in the explorer pane and double-click **hamming_window.c** to open the code, as shown in **Figure 44**.

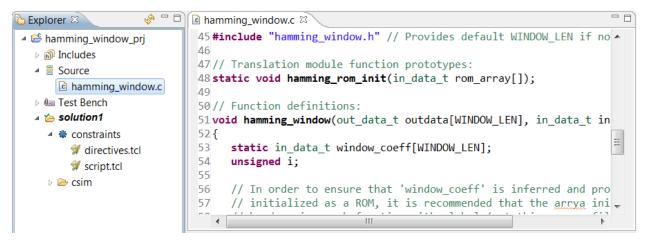


Figure 44: C Code for C Validation Lab 2

- 5. Hold down the **Ctrl** key and click **hamming_window.h** on line 45 to open this header file.
- 6. Scroll down to view the type definitions (Figure 45).





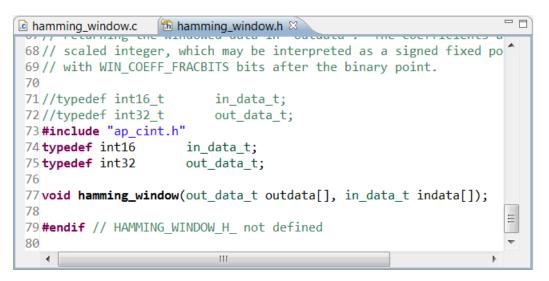


Figure 45: Type Definitions for C Validation Lab 2

In this lab, the design is the same as Lab 1, however, the types have been updated from the standard C data types (int16_t and int32_t) to the arbitrary precision types provided by Vivado High-Level Synthesis and defined in header file ap_cint.h.

More details for using arbitrary precision types are discussed in the tutorial **Arbitrary Precision Types**. An example of using arbitrary precision types would be to change this file to use 12-bit input data types: standard C types only support data widths on 8-bit boundaries.

This exercise demonstrates how such types can be debugged.

Step 2: Run the C Debugger

- 1. Click the Run C Simulation toolbar button to open the C Simulation Dialog box.
- 2. Select the **Debug** option.
- 3. Click **OK** to run the simulation.

The warning and error message shown in Figure 46 appears.

You cannot debug the arbitrary precision types used for ANSI C designs in the debug environment.



IMPORTANT! When working with arbitrary precision types you can use the Vivado HLS debug environment only with C++ or SystemC. When using arbitrary precision types with ANSI C, the debug environment cannot be used. With ANSI C, you must instead use printf or fprintf statements for debugging.





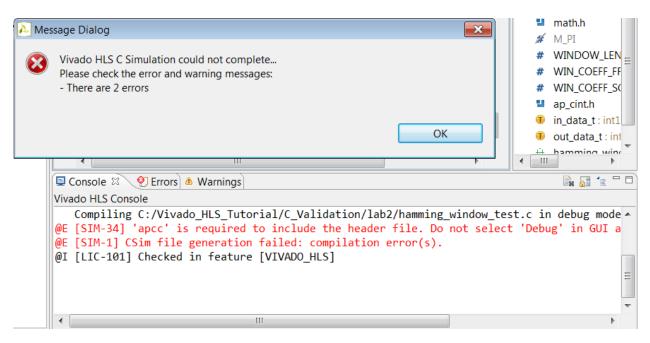


Figure 46: C Simulation Dialog Box

- 4. Expand the Test Bench folder in the Explorer pane.
- 5. Double-click the file hamming_window_test.c.
- 6. Scroll to line 78 and remove the comments in front of the printf statement (as shown in Figure 47).

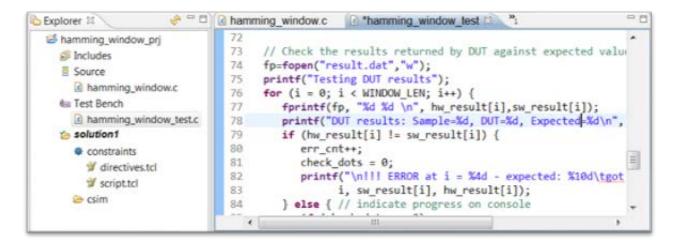


Figure 47: Enable Printing of the Results

- 7. Save the file.
- 8. Click the **Run C Simulation** toolbar button or the menu **Project > Run C simulation** to open the C Simulation Dialog box.
- 9. Click **OK** to run the simulation.





The results appear in the console window (Figure 48).

💷 Console 🗵 🔮 Errors] 💩 Warnings] 💿 🗰 🙀 🐺] - 8)
<terminated> hamming_window_prj.Debug [C/C++ Application] C:\Vivado_HLS_Tutorial\C_Validation\lab2\hammin</terminated>	ng_win
.DUT results: Sample=252, DUT=21807104, Expected=21807104	
.DUT results: Sample=253, DUT=27011801, Expected=27011801	
.DUT results: Sample=254, DUT=32266975, Expected=32266975	
.DUT results: Sample=255, DUT=37559010, Expected=37559010	
*** Test Passed ***	
	-
	•

Figure 48: C Validation Lab 2 Results

10. Exit the Vivado HLS GUI and return to the command prompt.

www.xilinx.com



Lab 3: C Validation with C++ Arbitrary Precision Types

Overview

This exercise uses a design with arbitrary precision C++ types. You will review and debug the design in the GUI.

Step 1: Create and Open the Project

- 1. From the Vivado HLS command prompt used in Lab 2, change to the lab3 directory.
- 2. Create a new Vivado HLS project by typing vivado_hls _f run_hls.tcl.
- 3. Open the Vivado HLS GUI project by typing vivado_hls -p hamming_window_prj.
- 4. Open the Source folder in the explorer pane and double-click hamming_window.cpp to open the code, as shown in Figure 49.

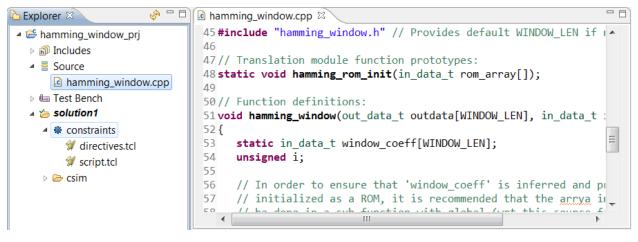


Figure 49: C++ Code for C Validation Lab 3

5. Hold down the **Ctrl** key down and click hamming_window.h on line 45 to open this header file.



6. Scroll down to view the type definitions (Figure 50).

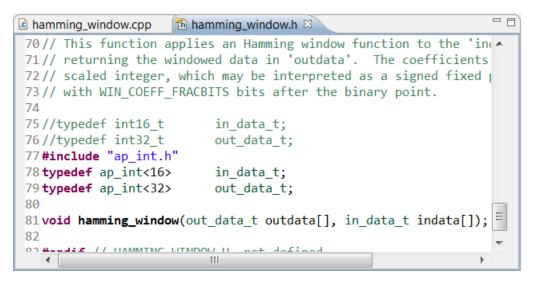


Figure 50: Type Definitions for C Validation Lab 3

Note: In this lab, the design is the same as in Lab 1 and Lab 2, with one exception. The types have been updated to use the C++ arbitrary precision types, ap_int<#N>, provided by Vivado High-Level Synthesis and defined in header file ap_int.h.





Step 2: Run the C Debugger

- 1. Click the Run C Simulation toolbar button to open the C Simulation Dialog box.
- 2. Select the **Debug** option.
- 3. Click **OK**.

The debug environment opens.

- 4. Select the hamming_window.cpp code tab.
- 5. Set a breakpoint at line 61 as shown in Figure 51.
- 6. Click the **Resume** button (or key F8) to execute the code up to the breakpoint.

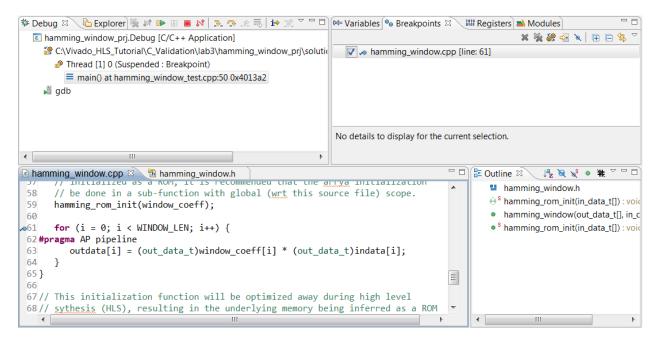


Figure 51: Debug Environment for C Validation Lab 3

7. Click the Step Into button (or the F5 key) twice to see the view in Figure 52.

The variables in the design are now C++ arbitrary precision types. These types are defined in header file ap_int.h. When the debugger encounters these types, it follows the definition into the header file.

As you continue stepping through the code, you have the opportunity to observe in greater detail how the results for arbitrary precision types are calculated.

www.xilinx.com



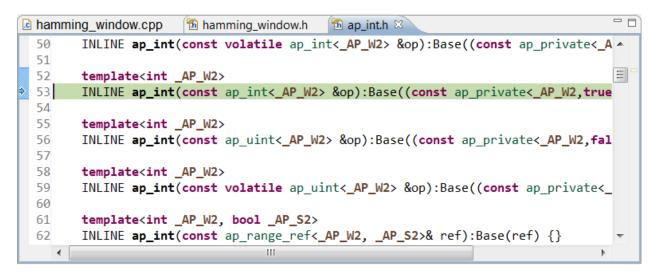


Figure 52: Arbitrary Precision Header File

A more productive methodology is to exit the ap_int.h header file and return to view the results.

- 8. Click the **Step Return** button (or the **F7** key) to return to the calling function.
- 9. Select the Variables tab.
- 10. Expand the outdata variable, as shown in Figure 53 to see the value of the variable shown in the VAL parameter.

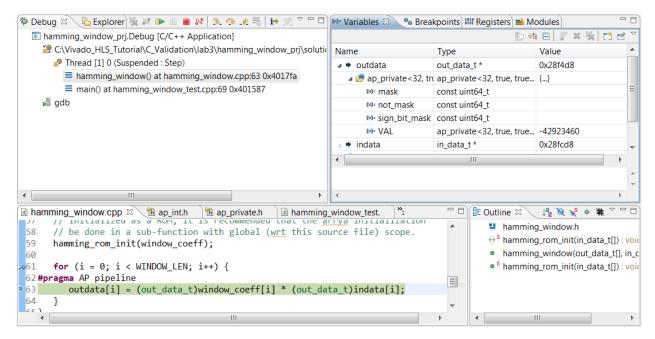


Figure 53: Arbitrary Precision Variables

Arbitrary precision types are a powerful means to create high-performance, bit-accurate hardware designs. However, in a debug environment, your productivity can be reduced by





stepping through the header file definitions. Use breakpoints and the step return feature to skip over the low-level calculations and view the value of variables in the Variables tab.

Conclusion

In this tutorial, you learned:

- The importance of the C test bench in the simulation process.
- How to use the C debug environment, set breakpoints and step through the code.
- How to debug C and C++ arbitrary precision types.





Chapter 4 Interface Synthesis

Interface synthesis is the process of adding RTL ports to the C design. In addition to adding the physical ports to the RTL design, interface synthesis includes an associated I/O protocol, allowing the data transfer through the port to be automatically and optimally synchronized with the internal logic.

This tutorial consists of four lab exercises that cover the primary features and capabilities of interface synthesis.

- Lab 1: Review the function return and block-level protocols.
- Lab 2: Understand the default I/O protocol for ports and learn how to select an I/O protocol.
- Lab 3: Review how array ports are implemented and can be partitioned.
- Lab 4 : Create an optimized implementation of the design and add AXI4 interfaces.

Tutorial Design Description

Download tutorial design file from the Xilinx website. Refer to the information in





Obtaining the Tutorial Designs.

This tutorial uses the design files in the tutorial directory Vivado_HLS_Tutorial\Interface_Synthesis.

About the Labs

- The sample design used in the first two labs in this tutorial is a simple one, which helps the focus to remain on the interfaces.
- The final two lab exercises use a multi-channel accumulator.
- This tutorial explains how to implement I/O ports and protocols using High-Level Synthesis.
- In Lab 4, you create an optimal implementation of the design used in Lab3.



Interface Synthesis Lab 1: Block-Level I/O protocols

Overview

This lab explains what block-level I/O protocols are and to control them.



IMPORTANT: The figures and commands in this tutorial assume the tutorial data directory Vivado_HLS_Tutorial is unzipped and placed in the location C:\Vivado_HLS_Tutorial. If the tutorial data directory is unzipped to a different location, or on Linux systems, adjust the few pathnames referenced, to the location you have chosen to place the Vivado_HLS_Tutorial directory.

Step 1: Create and Open the Project

- 1. Open the Vivado HLS Command Prompt.
 - a. On Windows use Start > All Programs > Xilinx Design Tools > Vivado 2013.3 > Vivado HLS > Vivado HLS 2013.3 Command Prompt (Figure 54).
 - b. In Linux, open a new shell.



Figure 54: Vivado HLS Command Prompt





- 2. Using the command prompt window (Figure 55), change directory to the Interface Synthesis tutorial, lab1.
- 3. Execute the Tcl script to setup the Vivado HLS project, using the command **vivado_hls –f run_hls.tcl**, as shown in **Figure 55**.

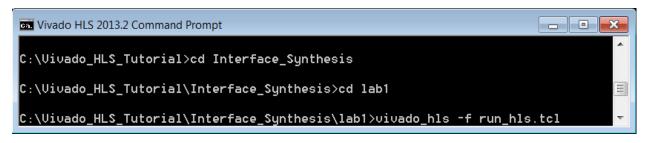


Figure 55: Setup the Tutorial Project

4. When Vivado HLS completes, open the project in the Vivado HLS GUI using the command **vivado_hls – p adders_prj**, as shown in **Figure 56**.

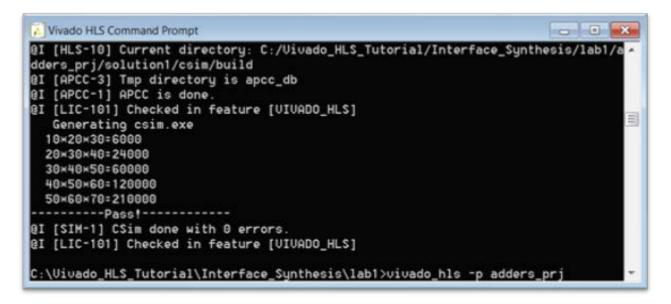


Figure 56: Initial Project for Interface Synthesis Lab 1

Step 2: Create and Review the Default Block-Level I/O Protocol

1. Double-click **adder.c** in the Source folder to pen the source code for review (Figure 57).

This example uses a simple design to focus on the I/O implementation (and not the logic in the design). The important points to take from this code are:

• Directives in the form of pragmas have been added to the source code to prevent any I/O protocol being synthesized for any of the data ports (inA, inB and inC). I/O port protocols are reviewed in the next lab exercise.





• This function returns a value and this is the only output from the function. As seen in later exercises, not all functions return a value. The port created for the function return is discussed in this lab exercise.

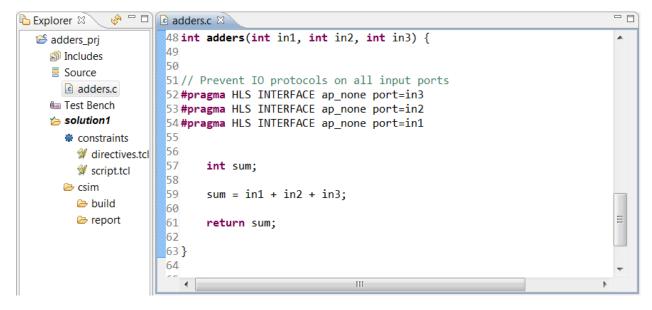


Figure 57: C Code for Interface Synthesis Lab 1

2. Execute the **Run C Synthesis** command using the dedicated toolbar button or the **Solution** menu.

When synthesis completes, the synthesis report opens automatically.

3. To review the RTL interfaces scroll to the Interface summary at the end of the synthesis report.

dders.c	🗊 add	ers_cs	ynth.rpt 🖾		
terface					
Summary					
RTL Ports	Dir	Dite	Drotocol	Source Object	C Turne
		Bits	Protocol	Source Object	С Туре
ap_clk	in	1	ap_ctrl_hs	adders	return value
ap_rst	in	1	ap_ctrl_hs	adders	return value
ap_start	in	1	ap_ctrl_hs	adders	return value
ap_done	out	1	ap_ctrl_hs	adders	return value
ap_idle	out	1	ap_ctrl_hs	adders	return value
ap_ready	out	1	ap_ctrl_hs	adders	return value
ap_return	out	32	ap_ctrl_hs	adders	return value
in1	in	32	ap_none	in1	scalar
in2	in	32	ap_none	in2	scalar
in3	in	32	ap_none	in3	scalar

The Interface summary and Outline tab are shown in Figure 58.

Figure 58: Interface Summary

There are three types of ports to review:

www.xilinx.com





- The design takes more than one clock cycle to complete, so a clock and reset have been added to the design: ap_clk and ap_rst. Both are single-bit inputs.
- A block-level I/O protocol has been added to control the RTL design: ports ap_start, ap_done, ap_idle and ap_ready. These ports will be discussed shortly.
- The design has four data ports.
 - Input ports In1, In2, and In3 are 32-bit inputs and have the I/O protocol ap_none (as specified by the directives in Figure 58).
 - The design also has a 32-bit output port for the function return, ap_return.

The block-level I/O protocol allows the RTL design to be controlled by via additional ports independently of the data I/O ports. This I/O protocol is associated with the function itself, not with any of the data ports. The default block-level I/O protocol is called ap_ctrl_hs. Figure 58 shows this protocol is associated with the design.

Table 1 summarizes the behavior of the signals for block-level I/O protocol ap_ctrl_hs.

Note: The explanation here uses the term "transaction". In the context of high-level synthesis, a transaction is equivalent to one execution of the C function (or the equivalent operation in the synthesized RTL design).

Exercise	Description
ap_start	This signal controls the block execution and must be asserted to logic 1 for the design to begin operation.
	It should be held at logic 1 until the associated output handshake ap_ready is asserted. When ap_ready goes high, the decision can be made on whether to keep ap_start asserted and perform another transaction or set ap_start to logic 0 and allow the design to halt at the end of the current transaction.
	If ap_start is asserted low before ap_ready is high, the design might not have read all input ports and might stall operation on the next input read.
ap_ready	This output signal indicates when the design is ready for new inputs.
	The ap_ready signal is set to logic 1 when the design is ready to accept new inputs, indicating that all input reads for this transaction have been completed.
	If the design has no pipelined operations, new reads are not performed until the next transaction starts.
	This signal is used to make a decision on when to apply new values to the inputs ports and whether to start a new transaction should using the ap_start input signal.
	If the ap_start signal is not asserted high, this signal goes low when the design completes all operations in the current transaction.
ap_done	This signal indicates when the design has completed all operations in the current transaction.

www.xilinx.com



Exercise	Description
	A logic 1 on this output indicates the design has completed all operations in this transaction. Because this is the end of the transaction, a logic 1 on this signal also indicates the data on the ap_return port is valid.
	Not all functions have a function return argument and hence not all RTL designs have an ap_return port.
ap_idle	This signal indicates if the design is operating or idle (no operation).
	The idle state is indicated by logic 1 on this output port. This signal is asserted low once the design starts operating.
	This signal is asserted high when the design completes operation and no further operations are performed.

Table 1: Block Level I/O protocol ap_ctrl_hs

You can observe the behavior of these signals by viewing the trace file produced by RTL cosimulation. This is discussed in the tutorial **RTL Verification**, but **Figure 59** shows the waveforms for the current synthesis results.

adders.wcfg* ×		다 관 × 127.500
name Name	Value	105 ns 110 ns 115 ns 120 ns 125 r
💐 🛛 🍇 ap_clk	1	
🔍 🕼 ap_rst	0	
🔍 🛝 ap_start	1	
🗼 🎞 📲 in1[31:0]	50	X 10 20 30 40 50
Image: Image	60	X 20 X 30 X 40 X 50 X 60
Here in3[31:0]	70	X 30 40 50 60 70
ap_uone	0	
	0	
² ⊞ [™] ap_return[31:0]		<u> </u>
मु ि 🕼 ap_idle	0	
E .		
<u>П</u>		
↔		
<u>H1</u>		
٠	• • •	•

Figure 59: RTL Waveforms for Block Protocol Signals

The waveforms in **Figure 56** show the behavior of the block-level I/O signals.

www.xilinx.com





- The design does not start operation until ap_start is set to logic 1.
- The design indicates it is no longer idle by setting port ap_idle low.
- Five transactions are shown. The first three input values (10, 20 and 30) are applied to input ports In1, In2 and In3 respectively.
- Output signal ap_ready goes high to indicate the design is ready for new inputs on the next clock.
- Output signal ap_done indicates when the design is finished and that the value on output port ap_return is valid (the first output value, 60, is the sum of all three inputs).
- Because ap_start is held high, the next transaction starts on the next clock cycle.

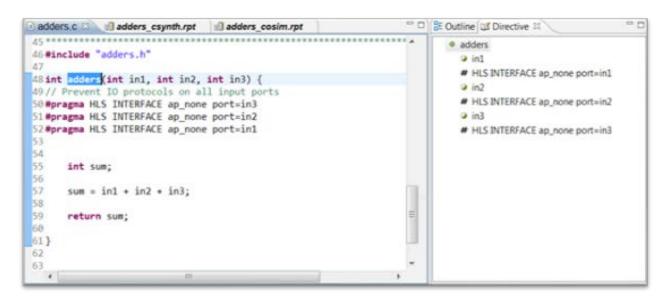
Note: In RTL Cosimulation, all design and port input control signals are always enabled. For example, in **Figure 59** signal ap_start is always high.

In the 2nd transaction, notice on port ap_return, the first output has the value 70. The result on this port is not valid until the ap_done signal is asserted high.

Step 3: Modify the Block-Level I/O protocol

The default block-level I/O protocol is the ap_ctrl_hs protocol (the Control Handshake protocol). In this step, you create a new solution and modify this protocol.

- 1. Select **New Solution** from the toolbar or Project menu to create a new solution.
- 2. Leave all settings in the new solution dialog box at their default setting and click **OK**.
- 3. Select the C source code tab in the Information pane (or re-open the C source code if it was closed).
- 4. Activate the Directives tab and select the top-level function, as shown in Figure 60.









Because the block-level I/O protocols are associated with the function, you must specify them by selecting the top-level function.

5. In the Directives tab, mouse over the top-level function adders, right-click, and select **Insert Directives**.

The Directives Editor dialog box opens.





Figure 61 shows this dialog box with the drop-down menu for the interface mode activated.

Туре	
Directive: INTERFA	CE .
Destination	
Source File	
Directive File	
Options	
mode (optional):	ap_ctrl_none •
register:	ap_ctrl_none
depth (optional):	ap_ctrl_hs ap_ctrl_chain
	Cancel

Figure 61: Directive Dialog box for ap_ctrl_none

The drop-down menu shows there are three options for the block-level interface protocol:

- ap_ctrl_none: No block-level I/O control protocol.
- ap_ctrl_hs: The block-level I/O control handshake protocol we have reviewed.
- ap_ctrl_chain: The block-level I/O protocol for control chaining. This I/O protocol is primarily used for chaining pipelined blocks together.

The block-level IO protocol ap_ctrl_chain is not covered in this tutorial. This protocol is similar to ap_ctrl_hs protocol but with an additional input signal, ap_continue, which must be high when ap_done is asserted for the next transaction to continue. This allows downstream blocks to apply back-pressure on the system and halt further processing when they are unable to accept new data.

6. In the Destination section of the Directives Editor dialog box, select Source File.

By default, directives are placed in the directives.tcl file. In this example, the directive is placed in the source file with the the existing I/O directives.

- 7. From the drop-down menu, select **ap_none**.
- 8. Click **OK**.





The source file now has a new directive, highlighted in both the source code and directives tab in **Figure 62**.

The new directive shows the associated function argument/port called return. All interface directives are attached to a function argument. For block-level I/O protocols, the return argument is used to specify the block-level interface. This is true even if the function has no return argument in the source code.

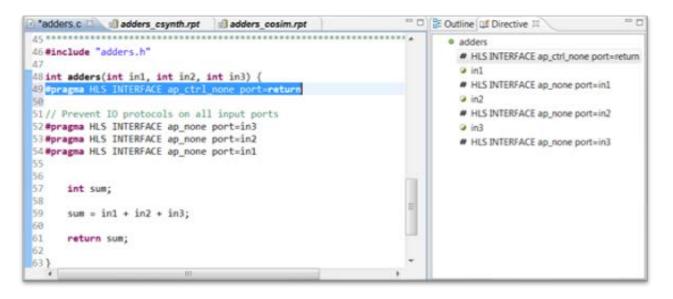


Figure 62: Block-Level Interface Directive ap_ctrl_none

9. Click the **Run C Synthesis** toolbar button or use the menu **Solution > Run C Synthesis** to synthesize the design.

Adding the directive to the source file modified the source file. **Figure 62** shows the source file name as *adders.c. The asterisk indicates that the file is modified but not saved.

10. Click **Yes** to accept the changes to the source file.

When the report opens, the Interface summary appears, as shown in Figure 63.





erface						
Summary						
RTL Ports	Dir	Bits	Protocol	Source Object	C Type	
ap_clk	in	1	ap_ctrl_none	adders	return value	
ap_rst	in	1	ap_ctrl_none	adders	return value	
ap_return	out	32	ap_ctrl_none	adders	return value	
in1	in	32	ap_none	in1	scalar	
in2	in	32	ap_none	in2	scalar	
in3	in	32	ap_none	in3	scalar	

Figure 63: Interface summary for ap_ctrl_none

When the interface protocol ap_ctrl_none is used, no block-level I/O protocols are added to the design. The only ports are those for the clock, reset and the data ports.

Note that without the ap_done signal, the consumer block that accepts data from the ap_return port now has no indication when the data is valid.

In addition, the RTL cosimulation feature requires a block-level I/O protocol to sequence the test bench and RTL design for cosimulation automatically. Any attempt to use RTL cosimulation results in the following error message and RTL cosimulation with halt:

```
@E [SIM-345] Cosim only supports the following 'ap_ctrl_none' designs: (1)
combinational designs; (2) pipelined design with task interval of 1; (3)
designs with array streaming or hls_stream ports.
@E [SIM-4] *** C/RTL co-simulation finished: FAIL ***
```

Exit the Vivado HLS GUI and return to the command prompt.

Interface Synthesis Lab 2: Port I/O protocols

Overview

This exercise explains how to specify port I/O protocols..

Step 1: Create and Open the Project

- 1. From the Vivado HLS command prompt used in Lab 1, change to the \lab2 directory as shown in Figure 64.
- 2. Type vivado_hls _f run_hls.tcl to create a new Vivado HLS project.





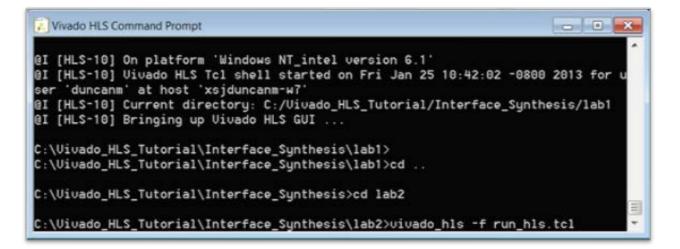


Figure 64: Setup for Interface Synthesis Lab 2

- 3. Type vivado_hls -p adders_io_prj to open the Vivado HLS GUI project.
- 4. Open the source code as shown in Figure 65.

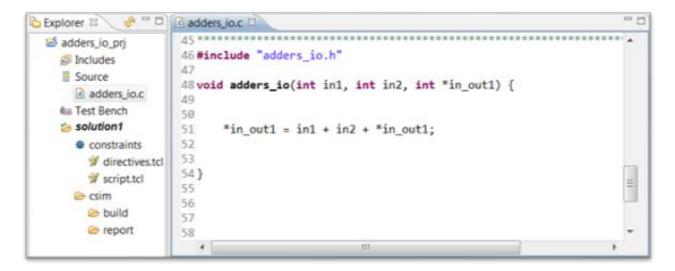


Figure 65: C Code for Interface Synthesis Lab 2

The source code for this exercise is similar to the simple code used in Lab 1. For similar reasons, it helps focus on the interface behavior and not the core logic.

This time, the code does not have a function return, but instead passes the output of the function through the pointer argument *in_out1. This also provides the opportunity to explore the interface options for bi-directional (input and output) ports.

The types of I/O protocol that you can add to C function arguments by interface synthesis depends on the argument type. These options are fully described in the Vivado High-Level Synthesis User Guide (UG902).





The pointer argument in this example is both an input and output to the function. In the RTL design, this argument is implemented as separate input and output ports.

www.xilinx.com





For the code shown in **Figure 65**, the possible options for each function argument are described in Table 2.

Function Argument	I/O protocol Options
In1 and In2	These are pass-by-value arguments that can be implemented with the following I/O Protocols:
	• <i>ap_none: No I/O protocol. This is the default for inputs.</i>
	• ap_stable: No I/O protocol.
	• <i>ap_ack: Implemented with an associated output acknowledge port.</i>
	• <i>ap_vld: Implemented with an associated input valid port.</i>
	• <i>ap_hs: Implemented with both input valid and output acknowledge</i> ports.
in_out1	This is a pass-by-reference output that can be implemented with the following I/O protocols:
	• <i>ap_none: No I/O protocol. This is the default for inputs.</i>
	• ap_stable: No I/O protocol.
	• <i>ap_ack: Implemented with an associated input acknowledge port.</i>
	 ap_vld: Implemented with an associated output valid port. This is the default for outputs.
	 ap_ovld: Implemented with an associated output valid port (no valid port for the input part of any inout ports).
	 ap_hs: Implemented with both input valid port and output acknowledge ports.
	 ap_fifo: A FIFO interface with associated output write and input FIFO full ports.
	• <i>ap_bus: A Vivado HLS bus interface protocol.</i>

Table 2: Port Level I/O Protocol Options for Lab 2

Note: The port directives applied in Lab 1 were not actually necessary because ap_none is the default I/O protocol for these C arguments. The directives were provided to avoid addressing any I/O port protocol behavior in that exercise, default behavior or not.

In this exercise, you implement a selection of I/O protocols.



E XILINX.

Step 2: Specify the I/O Protocol for Ports

- 1. Ensure that you can see the C source code in the Information pane.
- 2. Activate the Directives tab and select input argument in1, as shown in Figure 66.

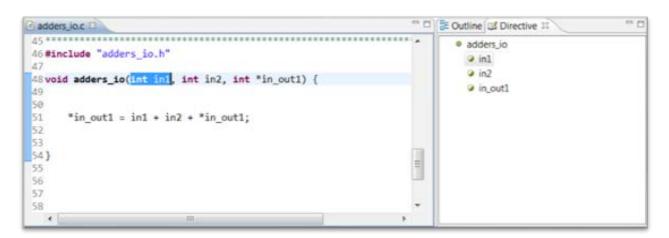


Figure 66: Adding Port I/O Protocols

- 3. Right-click and select Insert Directives.
- 4. When the Directives Editor opens leave the directives drop-down menu as INTERFACE.
 - a. Leave the destination at the default value. This time, the directives are stored in the directives.tcl file.
 - b. Select ap_vld from the mode drop-down menu
 - c. Click OK.
- 5. Select argument in2 and add an interface directive to specify the I/O protocol ap_ack.
- 6. Select argument in_out1 and add an interface directive to specify the I/O protocol ap_hs.



7. In the Explorer pane, expand the Constraints folder and double-click the directives.tcl file to open it, as shown in Figure 67.

💫 Explorer 🕴 🛛 🤣 😁 🗖	🖪 adders_io.c 🔰 directives.tcl 🕴	- 0
<pre>adders_io_prj Includes Source Test Bench of solution1 constraints firectives.tcl fi</pre>	<pre>1 ####################################</pre>	*

Figure 67: Directives for Lab 2

- 8. Synthesize the design.
- 9. Review the Interface summary when the report file opens (Figure 68).

terface						
Summary						
	Dir	Bits	Protocol	Source Object	С Туре	
ap_clk	in	1	ap_ctrl_hs	adders_io	return value	
ap_rst	in	1	ap_ctrl_hs	adders_io	return value	
ap_start	in	1	ap_ctrl_hs	adders_io	return value	
ap_done	out	1	ap_ctrl_hs	adders_io	return value	
ap_idle	out	1	ap_ctrl_hs	adders_io	return value	
ap_ready	out	1	ap_ctrl_hs	adders_io	return value	
n1	in	32	ap_vld	in1	scalar	
n1_ap_vld	in	1	ap_vld	in1	scalar	
n2	in	32	ap_ack	in2	scalar	
n2_ap_ack	out	1	ap_ack	in2	scalar	
n_out1_i	in	32	ap_hs	in_out1	pointer	
n_out1_i_ap_vld	in	1	ap_hs	in_out1	pointer	
in_out1_i_ap_ack	out	1	ap_hs	in_out1	pointer	
n_out1_o	out	32	ap_hs	in_out1	pointer	
n_out1_o_ap_vld	out	1	ap_hs	in_out1	pointer	
in_out1_o_ap_ack	in	1	ap_hs	in out1	pointer	

Figure 68: Interface summary for Lab 2

www.xilinx.com



- The design has a clock and reset.
- The default block-level I/O protocol signals are present.
- Port in1 is implemented with a data port and an associated input valid signal.
- The data on port in1 is only read when port in1_ap_vld is active high.
- Port in2 is implemented with a data port and an associated output acknowledge signal.
- Port in2_ap_ack will be active high when data port in2 is read.
- The inout_i identifies the input part of argument inout1. This has associated input valid port inout1_i_ap_vld and output acknowledge port inout1_i_ap_ack.
- The output part of argument inout1 is identified as inout_o. This has associated output valid port inout1_o_ap_vld and input acknowledge port inout1_o_ap_ack.
- 10. Exit the Vivado HLS GUI and return to the command prompt.

Interface Synthesis Lab 3: Implementing Arrays as RTL Interfaces

Introduction

This exercise shows how array arguments on functions you can implement as a number of different types of RTL port.

Step 1: Create and Open the Project

- 1. From the Vivado HLS command prompt window used in the previous lab, change to the lab3 directory.
- 2. Create a new Vivado HLS project by typing vivado_hls -f run_hls.tcl
- 3. Open the Vivado HLS GUI project by typing vivado_hls -p arrays_io_prj





4. Open the source code as shown in Figure 69.

This design has an input array and an output array. The comments in the C source explain how the data in the input array is ordered as channels and how the channels are accumulated. To understand the design, you can also review the test bench and the input and output data in file result.golden.dat.

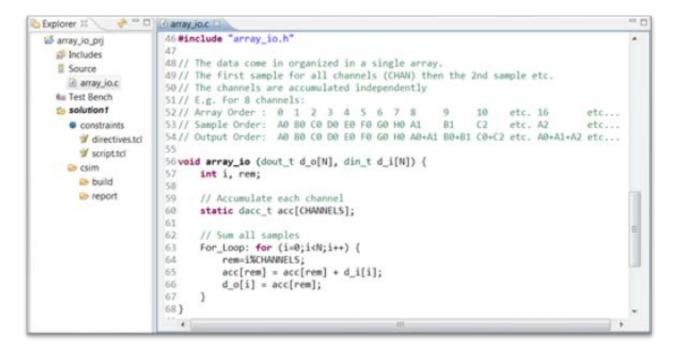


Figure 69: C Code for Interface Synthesis Lab 3

Step 2: Synthesize Array Function Arguments to RAM ports

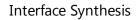
In this step, you review how array ports are synthesized to RAM ports.

1. Synthesize the design and review the Interface summary when the report opens (Figure 70).

The interface summary shows how array arguments in the C source are by default synthesized into RTL RAM ports.

- The design has a clock, reset and the default block-level I/O protocol ap_ctrl_hs (noted on the clock in the report).
- The d_o argument has been synthesized to a RAM port (I/O protocol ap_memory).
- A data port (d_o_d0).
- An address port (d_o_address0).
- Control ports for chip-enable (d_o_ce0) and a write-enable port (do_we0).
- The d_i argument has been synthesized to a similar RAM interface, but has an input data port (d_i_q0) and no write-enable port because this interface only reads data.

www.xilinx.com





In both cases, the data port is the width of the data values in the C source (32-bit integers in this case) and the width of the address port has been automatically sized match to the number of addresses that must be accessed (5-bit for 32 addresses).

Bits 1 1 1 1 1	Protocol ap_ctrl_hs ap_ctrl_hs ap_ctrl_hs ap_ctrl_hs	Source Object array_io array_io array_io array_io	C Type return value return value return value	
1 1 1 1	ap_ctrl_hs ap_ctrl_hs ap_ctrl_hs	array_io array_io array_io	return value return value return value	
1 1 1	ap_ctrl_hs ap_ctrl_hs	array_io array_io	return value return value	
1	ap_ctrl_hs	array_io	return value	
1	-			
	ap_ctrl_hs	array io	and the second sec	
1		anay_io	return value	
1 1	ap_ctrl_hs	array_io	return value	
1	ap_ctrl_hs	array_io	return value	
5	ap_memory	d_o	array	
1	ap_memory	d_o	array	
1	ap_memory	d_o	array	
16	ap_memory	d_o	array	
5	ap_memory	d_i	array	
1	ap_memory	d_i	array	
16	ap_memory	d_i	array	
	5 1 1 16 5 5 1	 5 ap_memory 1 ap_memory 1 ap_memory 16 ap_memory 5 ap_memory 1 ap_memory 16 ap_memory 16 ap_memory 	: 5 ap_memory d_o : 1 ap_memory d_o : 1 ap_memory d_o : 16 ap_memory d_o : 5 ap_memory d_o : 16 ap_memory d_o : 5 ap_memory d_i : 1 ap_memory d_i	5ap_memoryd_oarray1ap_memoryd_oarray1ap_memoryd_oarray16ap_memoryd_oarray5ap_memoryd_iarray1ap_memoryd_iarray1ap_memoryd_iarray1ap_memoryd_iarray16ap_memoryd_iarray16ap_memoryd_iarray

Figure 70: Interface Summary for Initial Lab 3 design

Synthesizing array arguments to RAM ports is the default. You can control how these ports are implemented using a number of other options. The remaining steps in Lab 3 demonstrate these options:

- Using a single-port or dual-port RAM interface.
- Using FIFO interfaces.
- Partitioning into discrete port.

Step 3: Using Dual-port RAM and FIFO interfaces

High-Level Synthesis lets you specify a RAM interface a single-port or dual-port. If you do not make such a selection, Vivado HLS automatically analyzes the design and selects the number of ports to maximize the data rate.

Step 2 used a single-port RAM interface because the for-loop in the source code (Figure 69) is by default left rolled: each iteration of the loop is executed in turn:

- Read the input port.
- Read the accumulated result from the internal RAM.
- Sum the accumulated and new data and write into the internal RAM.
- Write the result to the output port.
- Repeat for the next iteration of the loop.

This ensures only a single input read and output write is ever required. Even if multiple input and outputs are made available, the internal logic cannot take advantage of any additional ports.

Note: If you specify a dual-port RAM and Vivado HLS can determine only a single port is required, it uses a single-port and over-ride the dual-port specification.

In this design, if you want to implement an array argument using multiple RTL ports, the first thing you must do is unroll the for-loop and allow all internal operations to happen in parallel, otherwise there is no benefit in multiple ports: the rolled for-loop ensure only one data sample can be read (or written) at a time.

- 1. Select **New Solution** from the toolbar or Project menu to create a new solution.
- 2. Accept the defaults, and click **OK**.
- 3. Ensure the C source code is visible in the Information pane.
- 4. In the Directives tab, select the for-loop, For_Loop, and right-click to open the **Directives Editor** dialog box.
 - a. In the Directives Editor dialog box activate the Directives drop-down menu at the top and select **UNROLL**.





b. With the Directives Editor as shown in Figure 71, click **OK**.

Туре		
Directive: UNROLL		•
Destination O Source File		
Options	-	
skip exit check:		
factor (optional):		
region:		
Help	Cancel	OK

Figure 71: Directives Editor to Unroll For_Loop





Next, specify a dual-port RAM for input reads. The Resource directive indicates the type of RAM connected to an interface.

- 5. In the Directives tab, select **port d_i** and right-click to open the Directives Editor dialog box.
 - a. In the Directives Editor activate the Directives drop-down menu at the top and select **RESOURCE**.
 - b. Click the core options box and select RAM_2P_RAM.
 - c. Verify that the settings in the Directives Editor dialog box are as shown in Figure 72 and click **OK**.

Гуре	
Directive: RESOUR	CE 🔹
Destination	
Source File	
Directive File	
Options	
ariable (required):	d_i
core (optional):	RAM_2P_BRAM
oort map (optional)	c
netadata (optional):

Figure 72: Directives Editor for Specifying a Dual-port RAM





Implement the output port using a FIFO interface.

- 6. In the Directives tab, select port **d_o** and right-click to open the **Directives Editor** dialog box.
 - a. In the Directives Editor, leave the directive as **Interface**.
 - b. From the Mode drop-down menu, select **ap_fifo**.
 - c. Click **OK**.

The Directive tab shows the directives now applied to the design (Figure 73).

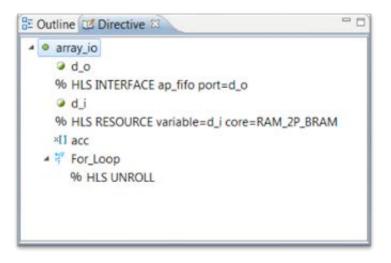
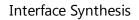


Figure 73: Directives Summary for Lab 2 Solution2

7. Synthesize the design.







When the report opens in the Information pane, the Interface summary is as shown in Figure 74.

- The design has the standard clock, reset and block-level I/O ports.
- Array argument d_o has been implemented as a FIFO interface with a 16-bit data port (d_o_din) and associated output write (d_o_write) and input FIFO full (d_o_full_n) ports.
- Argument d_i has been implemented as a dual-port RAM interface.

terface						
Summary						
RTL Ports	Dir	Bits	Protocol	Source Object	С Туре	
ap_clk	in	1	ap_ctrl_hs	array_io	return value	
ap_rst	in	1	ap_ctrl_hs	array_io	return value	
ap_start	in	1	ap_ctrl_hs	array_io	return value	
ap_done	out	1	ap_ctrl_hs	array_io	return value	
ap_idle	out	1	ap_ctrl_hs	array_io	return value	
ap_ready	out	1	ap_ctrl_hs	array_io	return value	
d_o_din	out	16	ap_fifo	d_o	pointer	
d_o_full_n	in	1	ap_fifo	d_o	pointer	
d_o_write	out	1	ap_fifo	d_o	pointer	
d_i_address0	out	5	ap_memory	d_i	array	
d_i_ce0	out	1	ap_memory	d_i	array	
d_i_q0	in	16	ap_memory	d_i	array	
d_i_address1	out	5	ap_memory	d_i	array	
d_i_ce1	out	1	ap_memory	d_i	array	
d_i_q1	in	16	ap_memory	d_i	array	

Figure 74: Directives Editor Specifying Block RAM Interface

By using a dual-port RAM interface, this design can accept input data at twice the rate of the previous design. However, by using a single-port FIFO interface on the output the output data rate is the same as before.





Step 4: Partitioned RAM and FIFO Array interfaces

In this step, you learn how to partition an array interface into any arbitrary number of ports.

- 1. Select **New Solution** from the toolbar or the Project menu and create a new solution.
- 2. Accept the defaults, and click **OK**. This includes copying existing directives from solution2.
- 3. Ensure the C source code is visible in the Information pane.
- 4. In the directives tab, select d_o and right-click to open the **Directives Editor** dialog box.
 - a. In the Directives Editor dialog box activate the Directives drop-down menu at the top and select ARRAY_PARTITION.
 - b. Activate the type drop-down menu and select block to partition the array into blocks.
 - c. In the Factor dialog box, enter the value 4.
 - d. With the Directives Editor as shown in Figure 75, click OK.

Туре		_
Directive: ARRAY_PAR	RTITION	•
Destination		
Source File		
Directive File		
Options		
variable (required):	d_o	
type (optional):	block	•
factor (optional):	4	_
dimension (optional):	1	
Help	Cancel OK	

Figure 75: Directives Editor for Partitioning Array d_o

Now, partition the input array into two blocks (not four).

5. In the Directives tab, select d_i and repeat the previous step, but this time partition the port with a factor of 2.





The directives tab shows the directives now applied to the design (Figure 76 76).

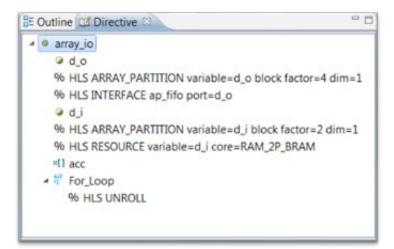


Figure 76: Directives Summary for Lab 2 Solution3

6. Synthesize the design.

When the report opens in the Information pane, the Interface summary is as shown in Figure 77.

- The design has the standard clock, reset and block-level I/O ports.
- Array argument d_o has been implemented as a four separate FIFO interfaces.
- Argument d_i has been implemented as a two separate RAM interfaces, each of which uses a dual-port interface. (If you see 4 separate RAM interfaces, confirm a partition factor for d_i is 2 and not 4).





rray_io_csynth.r terface						
Summary						
RTL Ports	Dir	Bits	Protocol	Source Object	С Туре	
ap_clk	in	1	ap_ctrl_hs	array_io	return value	
ap_rst	in	1	ap_ctrl_hs	array_io	return value	
ap_start	in	1	ap_ctrl_hs	array_io	return value	
ap_done	out	1	ap_ctrl_hs	array_io	return value	
ap_idle	out	1	ap_ctrl_hs	array_io	return value	
ap_ready	out	1	ap_ctrl_hs	array_io	return value	
d_o_0_din	out	16	ap_fifo	d_o_0	pointer	
d_o_0_full_n	in	1	ap_fifo	d_o_0	pointer	
d_o_0_write	out	1	ap_fifo	d_o_0	pointer	
d_o_1_din	out	16	ap_fifo	d_o_1	pointer	
d_o_1_full_n	in	1	ap_fifo	d_o_1	pointer	
d_o_1_write	out	1	ap_fifo	d_o_1	pointer	
d_o_2_din	out	16	ap_fifo	d_o_2	pointer	
d_o_2_full_n	in	1	ap_fifo	d_o_2	pointer	
d_o_2_write	out	1	ap_fifo	d_o_2	pointer	
d_o_3_din	out	16	ap_fifo	d_o_3	pointer	-
d_o_3_full_n	in	1	ap_fifo	d_o_3	pointer	
d_o_3_write	out	1	ap_fifo	d_o_3	pointer	
d_i_0_address0	out	4	ap_memory	d_i_0	array	
d_i_0_ce0	out	1	ap_memory	d_i_0	array	
d_i_0_q0	in	16	ap_memory	d_i_0	array	
d_i_0_address1	out	4	ap_memory	d_i_0	array	
d_i_0_ce1	out	1	ap_memory	d_i_0	array	
d_i_0_q1	in	16	ap_memory	d_i_0	array	
d_i_1_address0	out	4	ap_memory	d_i_1	array	
d_i_1_ce0	out	1	ap_memory	d_i_1	array	
d_i_1_q0	in	16	ap_memory	d_i_1	array	
d_i_1_address1	out	4	ap_memory	d_i_1	array	
d_i_1_ce1	out	1	ap_memory	d_i_1	array	
d_i_1_q1	in	16	ap_memory	d_i_1	array	

Figure 77: Interface Report for Partitioned Interfaces

If input port d_i was partitioned into four, only a single-port RAM interface would be required for each port. Because the output port can only output four values at once, there would be no benefit in reading 8 inputs at once.

The final step in this tutorial on arrays is to partition the arrays completely.

www.xilinx.com

E XILINX.

Step 5: Fully Partitioned Array interfaces

This step shows you how to partition an array interface into individual ports.

- 1. Select **New Solution** from the toolbar and create a new solution.
- 2. Click **OK** and accept the defaults. This includes copying existing directives from solution3.
- 3. Ensure the C source code is visible in the Information pane.
- 4. In the Directive tab, select the existing partition directive for d_o as shown in Figure 78.
- 5. Right-click and select **Modify Directive**.

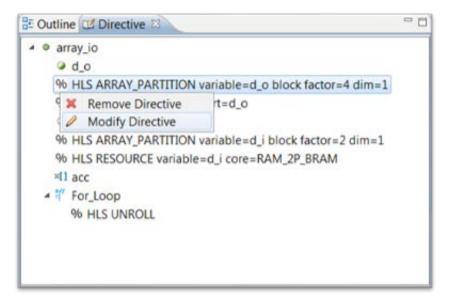


Figure 78: Modifying the Directive for d_o

- 6. In the Directives Editor dialog box:
 - a. Activate the **Type** drop-down menu and modify the partitioning style to **Complete**.
 - b. In the Factor dialog box, the you can remove the value 4 or leave it as-is. The factor is ignored for this type of partitioning.





c. With the Directives Editor as shown in Figure 79, click **OK**.

Туре	
Directive: ARRAY_PA	ARTITION
Destination	
Source File	
Directive File	
Options	
variable (required):	d_o
type (optional):	complete 👻
factor (optional):	
dimension (optional)	: 1
r	

Figure 79: Directives Editor for Partitioning Array d_o

7. In the Directives tab, select d_i and repeat the previous step to completely partition the d_i array.

Optionally, you can delete the directive on d_i specifying the resource. If the array is partitioned into individual elements, the Resource directive, which specifies a RAM resource, is ignored.



The Directives tab shows the directives now applied to the design (Figure 80).

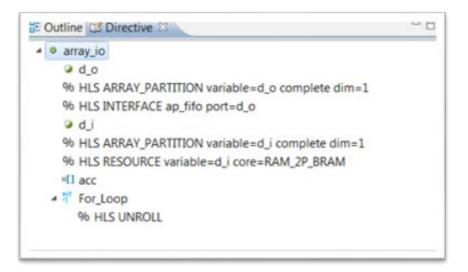


Figure 80: Directives Summary for Lab 2 Solution3

- 8. Synthesize the design.
- 9. When the report opens in the Information pane, review the interface summary. Note the following:
- The design has the standard clock, reset and block-level I/O ports.
- Array argument d_o has been implemented as a 32 separate FIFO interfaces.
- Argument d_i has been implemented as a 32 separate scalar port. Because the default interface for input scalars in no I/O protocol, they have the I/O protocol ap_none.

Although this tutorial has focused exclusively on the I/O interfaces, at this point it is worth examining the differences in performance across all four solutions.

10. Select Compare Reports from the toolbar or the Project menu to open a comparison of the solutions.

www.xilinx.com





- 11. In the Solution Selection dialog box, add each of the four solutions to the Selected Solutions pane (Figure 81 81).
- 12. Click **OK**.

Available solutions:		Selected solutions:
	Add>> < <remove< th=""><th>solution1 solution2 solution3 solution4</th></remove<>	solution1 solution2 solution3 solution4

Figure 81: Compare All Solutions for Lab 3

When the solutions comparison report opens (**Figure 82**), it shows that solution4, using a unique port for each array element, is much faster than the previous solutions. The internal logic can access the data as soon as it is required. (There is no performance bottleneck due to port accesses.)

erforman	ce Estim	ates								
Timing	(ns)									
Clock			soluti	on1	soluti	on2	soluti	on3	solutio	on4
default	Target		4.00		4.00		4.00		4.00	
	Estimat	ted	2.39		3.45		3.45		3.40	
Latency	(clock c	ycles	5)							
		solu	ition1	solu	ution2	solu	ition3	solu	ition4	
Latency	min	129		33		11		2		
	max	129		33		11		2		
Interval	min	130		34		12		3		

Figure 82: Performance Comparisons for All Lab 3 Solutions

www.xilinx.com



Scroll further down the comparison report (Figure 83) and note that solutions with more I/O ports (solutions 2, 3 and 4), allowing more parallel processing, also use considerably more resources.

🗄 compare repo	orts 🛛				-	
Utilization Est	imates					*
	solution1	solution2	solution3	solution4		
BRAM_18K	1	0	0	0		
DSP48E	0	0	0	0		
FF	120	1238	1220	1026		Ξ
LUT	53	1261	1186	1026		
						Ŧ

Figure 83: Resource Comparisons for All Lab 3 Solutions

In the next exercise, you implement this same design with an optimum balance between the ports and resources. In addition to this more optimal implementation, the next exercise shows how to add AXI interfaces to the design.

13. Exit the Vivado HLS GUI and return to the command prompt.

Interface Synthesis Lab 4: Implementing AXI Interfaces

Introduction

This exercise explains how to specify AXI bus interfaces for the I/O ports.

Step 1: Create and Open the Project

- 1. From the Vivado HLS command prompt window used in the previous lab, change to the lab4 directory.
- 2. Create a new Vivado HLS project by typing vivado_hls -f run_hls.tcl
- 3. Open the Vivado HLS GUI project by typing vivado_hls -p axi_interface_prj
- 4. Open the source code as shown in Figure 84.





```
and interfaces.c D
                                                                                                                                        .
46#include "axi_interfaces.h"
47
48// The data comes in organized in a single array.
49// - The first sample for the first channel (CHAN)
50// - Then the first sample for the 2nd channel etc.
 51// The channels are accumulated independently
52// E.g. For 8 channels:

      53// Array Order:
      0
      1
      2
      3
      4
      5
      6
      7
      8
      9
      10
      etc.
      16
      etc...

      54// Sample Order:
      A0
      B0
      C0
      D0
      E0
      F0
      60
      H0
      A1
      B1
      C2
      etc.
      A2
      etc...

      55// Output Order:
      A0
      B0
      C0
      D0
      E0
      F0
      60
      H0
      A0+A1
      B0+B1
      C0+C2
      etc...

57 void axi_interfaces (dout_t d_o[N], din_t d_i[N]) {
58
        int i, rem;
59
       // Store accumulated data
60
61
       static dacc_t acc[CHANNELS];
62
63
       // Accumulate each channel
        For_Loop: for (i=0;i<N;i++) {
64
65
               rem=i%CHANNELS;
               acc[rem] = acc[rem] + d_i[i];
66
67
               d_o[i] = acc[rem];
68
         }
69 }
70
```

Figure 84: Source code for Lab 4

This design uses similar source C code as Lab 3: with the design renamed to axi_interfaces.

Step 2: Create an Optimized Design

In the optimal performance implementation of this design, the data for each channel would be processed in parallel, with dedicated hardware for each channel.

The key to understanding how best to perform this optimization is to recognize that the channels in the input and output arrays lend themselves to cyclic partitioning. Cyclic partitioning is fully explained in the *Vivado HLS User Guide (UG902*, but basically means each array element is, in turn, sorted into a different partition.

If the I/O arrays are partitioned into channels, you can use FIFO interfaces to stream the samples for each channel through the design in parallel.

Finally, if the I/O ports are configured to supply and consume individual streams of channel data, partial unrolling of the for-loop can ensure dedicated hardware processes each channel.

First, partition the arrays:

- 1. Ensure the C source code is visible in the Information pane.
- 2. In the Directives tab, select d_o and right-click to open the Directives Editor dialog box.
 - a. Select the **Directives** drop-down menu at the top and select **ARRAY_PARTITION**.
 - b. Click the Type drop-down menu to specify cyclic partitioning.
 - c. In the **Factor** dialog box, enter the value **8**, to create eight separate partitions. (This results in eight ports.)



E XILINX.

www.xilinx.com





d. With the Directives Editor dialog box filled in as shown in Figure 85, click OK.

Туре	
Directive: ARRAY_PA	RTITION
Destination	
O Source File	
Directive File	
Options	
variable (required):	d_o
type (optional):	cyclic 👻
factor (optional):	8
dimension (optional):	1
Help	Cancel OK

Figure 85: Directives Editor for Cyclic Partitioning

- 3. In the Directives tab, select **d_o** again and right-click to open the **Directives Editor** dialog box.
 - a. Activate the **Directives** drop-down menu at the top and select **INTERFACE**.
 - b. Click the Mode drop-down menu to specify an ap_fifo interface.
 - c. Click **OK**.
- 4. In the Directives tab, select **d**_**i** and repeat steps 2 and 3 above.
 - a. Apply cyclic partitioning with a factor of 8.
 - b. Apply an ap_fifo interface.
- 5. Next, partially unroll and pipeline the for-loop:
 - a. In the Directives tab, select **For_Loop** and right-click to open the **Directives Editor** dialog box.





- b. Activate the **Directives** drop-down menu at the top and select **UNROLL**.
 - i. Select a factor of **8** to partially unroll the for-loop. This is equivalent to re-writing the C code to execute eight copies of the loop-body in each iteration of the loop (where the new loop only executes for four iterations in total, not 32).
 - ii. Click **OK**.
- c. In the Directives tab, select **For_Loop** again and right-click to open the **Directives Editor** dialog box.
 - i. Activate the **Directives** drop-down menu at the top and select **PIPELINE**.
 - ii. Leave the Interval blank and let it default to 1.
 - iii. Select enable loop rewinding.
 - iv. Click **OK**.

When the top-level of the design is a loop, you can use the pipeline rewind option. This informs Vivado HLS that when implemented in RTL, this loop runs continuously (with no end of function and function re-start cycles).

After performing the above steps, the Directives tab should be as shown in **Figure 86**. Be sure to check all options are correctly applied. If not, double-click the directive to re-open the **Directives Editor**.

(王)	Outlin	ne 😅 Directive 🖾	
	• ax	ci_interfaces	
	9	d_o	
	96	HLS ARRAY_PARTITION variable=d_o cyclic f	factor=8 dim=1
	96	HLS INTERFACE ap_fifo port=d_o	
		d_i	
	96	HLS INTERFACE ap_fifo port=d_i	
	96	HLS ARRAY_PARTITION variable=d_i cyclic fa	actor=8 dim=1
	×[]	acc	
	a 11	For_Loop	
		% HLS UNROLL factor=8	
		% HLS PIPELINE rewind	

Figure 86: Directives tab for Lab 4 Solution1

6. Synthesize the design.

When the report opens in the information pane, confirm both d_i and d_o are implemented as eight separate FIFO ports.

www.xilinx.com





7. In the performance section of the design, confirm that the for-loop processes one sample every clock cycle (Interval 1) with a latency of 2, and that the design has less area than solutions 2, 3, or 4 in Lab 3 (Figure 83).

Cyclic partitioning of the array interfaces and partial for-loop unrolling has allowed implementation of this C code as eight separate channels in the hardware.

Step 3: Implementing an AXI4-Lite Interfaces

Adding an AXI4-Lite interface is a two-step process:

- First, you specify the interface to have an I/O protocol, using the Interface directive.
- Second, you add a Resource directive to the RTL port so that an AXI4 interface connects to the port. (This is similar to the method for specifying RAM interfaces.) The AXI4 interfaces are added to the design during the IP creation stage.

In this exercise, you specify FIFO interfaces as AXI4 Stream interfaces. You group block-level I/O protocol ports into a single AXI4 Lite interface, which allows these block-level control signals to be controlled and accessed from a CPU.

- 1. Select **New Solution** from the toolbar or the **Project** menu to create and new solution.
- 2. Accept the defaults and click **OK.** This includes copying existing directives from solution3.
- 3. Ensure the C source code is visible in the Information pane.
- 4. In the Directives tab, select the top-level function **axi_interfaces** and right-click to open the **Directives Editor** dialog box.
 - a. Activate the **Directives** drop-down menu at the top and select **RESOURCE**.
 - b. Because you selected the axi_interfaces function, the variable field completes automatically with the function return.
 - c. Click the **Core options** box and select **AXI4LiteS**. This specifies the ports associated with the function return (the block-level I/O ports) are connected to an AXI4Lite interface.
 - d. Click **OK**.

The Directives tab appears, as shown in Figure 87.

www.xilinx.com

E XILINX.

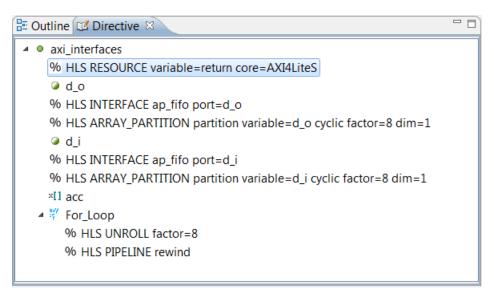


Figure 87: Directives for Specifying AXI4 Interfaces

5. Synthesize the design.

When the report opens, only the RTL ports appear in the Interface summary. AXI4 interfaces are not added to the design until it is packaged as IP.

- 6. Select **Export RTL** from the toolbar or the Solution menu, to create an IP package.
- 7. Leave the Format Selection as IP Catalog and click **OK**.



You can see the IP package in the solution2/impl folder (Figure 88). Because you used the Vivado IP Catalog format, the package is in the ip folder.

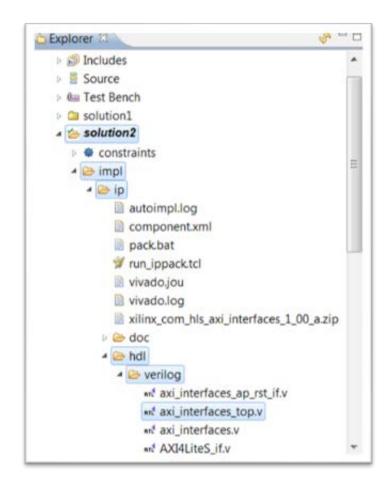


Figure 88: IP Package with AXI4 Interfaces

The top-level HDL is in the verilog subfolder, as shown in Figure 88. The extension _top identifies the top-level file. In this case, the top-level file is axi_interfaces_top.v .

When AXI4 interfaces are added, only Verilog HDL is currently created.

8. Double-click the axi_interfaces_top.v file to open it in the Information pane.





- 8

Review the top-level HDL file to view the added AXI4 Slave Lite interface (Figure 89)

RTL axi_interfaces_top.v 🛛

```
1// ------
                                                                             .
 2// File generated by Vivado(TM) HLS - High-Level Synthesis from C, C++ and SystemC
3// Version: 2013.3
 4// Copyright (C) 2013 Xilinx Inc. All rights reserved.
 5//
 6// ------
 7
8 timescale 1 ns / 1 ps
9 module axi_interfaces_top (
10 s_axi_AXI4LiteS_AWADDR,
11 s axi AXI4LiteS AWVALID,
12 s axi AXI4LiteS AWREADY,
13 s axi AXI4LiteS WDATA,
14 s axi AXI4LiteS WSTRB,
15 s axi AXI4LiteS WVALID,
16 s_axi_AXI4LiteS_WREADY,
17 s_axi_AXI4LiteS_BRESP,
18 s_axi_AXI4LiteS_BVALID,
19 s_axi_AXI4LiteS_BREADY,
20 s axi AXI4LiteS ARADDR,
21 s axi AXI4LiteS ARVALID,
22 s axi AXI4LiteS ARREADY,
23 s_axi_AXI4LiteS_RDATA,
24 s_axi_AXI4LiteS_RRESP,
25 s_axi_AXI4LiteS_RVALID,
26 s_axi_AXI4LiteS_RREADY,
27 interrupt,
28 aresetn,
29 aclk,
  €.
```

Figure 89: IP HDL with AXI4 Interfaces

This design was synthesized with an AXI4-Lite interface (for the block-level protocol ports). When you add an AXI4-Lite interface to the design, the IP packaging process also creates software driver files to enable an external block, typically a CPU, to control this block (start it, stop it, set port values, review the interrupt status).





Figure 90 shows the software drivers created in the impl directory with one of the files open in the Information pane.

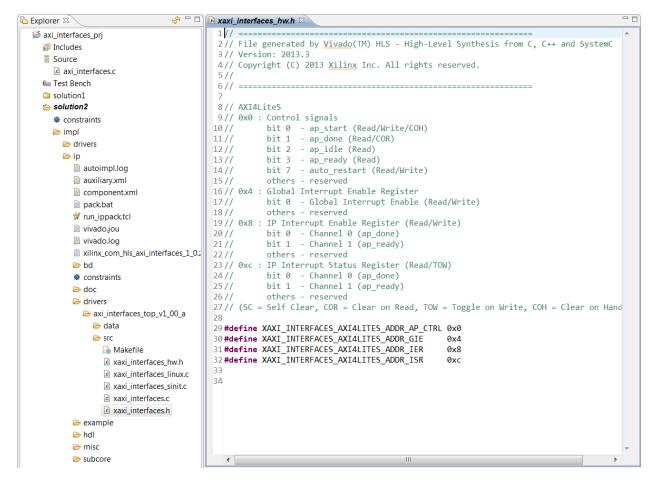


Figure 90: IP Software Driver Files

Conclusion

In this tutorial, you learned:

- What block-level I/O protocols are and how to control them.
- How to specify and apply port-level I/O protocols.
- How to specify array ports as RAM and FIFO interfaces.
- How to partition RAM and FIFO interfaces into sub-ports.
- How to use both I/O directives and optimization directives to create an optimal design with AXI4 interfaces.





Chapter 5 Arbitrary Precision Types

Overview

C/C++ provided data types are fixed to 8-bit boundaries:

- char (8-bit)
- short (16-bit)
- int (32-bit)
- long long (64-bit)
- float (32-bit)
- double (64-bit)
- Exact width integer types such as int16_t (16-bit) and int32_t (32-bit)

When creating hardware, it is often the case that more accurate bit-widths are required. Consider, for example, a case in which the input to a filter is 12-bit and the accumulation of the results only requires a maximum range of 27 bits. Using standard C data types for hardware design results in unnecessary hardware costs. Operations can use more LUTs and registers than needed for the required accuracy, and delays might even exceed the clock cycle, requiring more cycles to compute the result.

Vivado High-Level Synthesis (HLS) provides a number of bit-accurate or arbitrary precision datatypes, allowing you to model variables using any (arbitrary) width.

This tutorial consists of a two lab exercises:

- Lab1 Synthesize a design using floating-point types and review the results. The design uses standard C++ floating-point types.
- Lab2 -Synthesize the same function used in Lab 1 using arbitrary precision fixed-types highlighting the benefits in accuracy and results. This exercise shows how this same design can be converted to the Vivado HLS ap_fixed types, retaining the required accuracy but creating a more optimal hardware implementation

Tutorial Design Description

Download the tutorial design file from the Xilinx website. See the information in





Obtaining the Tutorial Designs. This tutorial uses the design files in the tutorial directory **Vivado_HLS_Tutorial\Arbitary_Precision**.

Arbitrary Precision: Lab 1

Arbitrary Precision Lab 1: Review a Design using Standard C/C++ types

In this lab, you synthesize a design using standard C types. You use this design as a reference for the design using arbitrary precision types, which is the basis for Lab 2.

IMPORTANT: The figures and commands in this tutorial assume the tutorial data directory *Vivado_HLS_Tutorial* is unzipped and placed in the location



If the tutorial data directory is unzipped to a different location, or on Linux systems, adjust the few pathnames referenced, to the location you have chosen to place the **Vivado_HLS_Tutorial** directory.

Step 1: Create and Open the Project

- 1. Open the Vivado HLS Command Prompt.
 - a. On Windows use Start > All Programs > Xilinx Design Tools > Vivado 2013.3 > Vivado HLS > Vivado HLS 2013.3 Command Prompt (Figure 91).
 - b. On Linux, open a new shell.

C:\Vivado HLS Tutorial.



Figure 91: Vivado HLS Command Prompt





- 2. In the command prompt window (Figure 92), change the directory to the Arbitrary Precision tutorial, lab1.
- 3. Execute the Tcl script to setup the Vivado HLS project, using the command as shown in **Figure 92**:

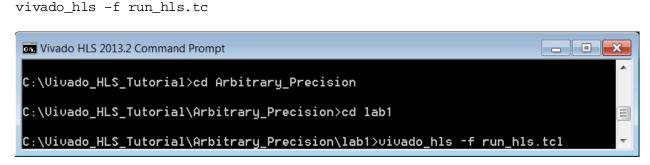


Figure 92: Setup the Tutorial Project

4. When Vivado HLS completes, open the project in the Vivado HLS GUI using the command vivado_hls –p window_fn_prj as shown in **Figure 93**.

<pre>i = 22 hw_result = 44.24587</pre>
i = 24 hw_result = 32.00000
i = 25 hw_result = 25.75711
i = 26 hw_result = 19.75413
$i = 27$ hw_result = 14.22175 sw_result = 14.22175
i = 28 hw_result = 9.37258
i = 29 hw_result = 5.39297
i = 30 hw_result = 2.43585
i = 31 hw_result = 0.61487
Test Passed
@I [SIM-1] CSim done with 0 errors.
@I [LIC-101] Checked in feature [VIVADO_HLS]
C:\Vivado_HLS_Tutorial\Arbitrary_Precision\lab1>vivado_hls -p window_fn_prj 🗾 🔻

Figure 93: Initial Project for Arbitrary Precision Lab 1

Step 2: Review Test Bench and Run C Simulation

1. Open the Source folder in the explorer pane and double-click window_fn_top.cpp to open the code as shown in Figure 94.





🔁 Explorer 🛛 🛛 🤣 🧧 🗖	i window_fn_top.cpp ⊠	- 0
 ✓ window_fn_prj ▷ ĵĵi Includes ▲ Ξ Source ☑ window_fn_top.cpp 	45 #include "window_fn_top.h" // Provides typedefs and params 46 47// Include the entire xhls_window_fn namespace so that scope r 48// i.e. prepending xhls window fn:: to everything is not ne	
 Image: Barrier Bench Image: Amage: Amage: Bench Image: Amage: Amage: Bench Image: Amage: Bench Image: Benc Image: Bench<!--</th--><td>49 using namespace xhls_window_fn; 50 51//Vivado HLS requires a top-level function definition that wra</td><td>31</td>	49 using namespace xhls_window_fn; 50 51//Vivado HLS requires a top-level function definition that wra	31
 Constraints directives.tcl script.tcl 	52// instantiations and method calls to be synthesized as well a 53// the top-level I/O (function arguments) into/out of the meth 54 void window fn top(a!
 ✓ Scripter ✓ Csim ✓ ▷ build ✓ ▷ ▷ report 	<pre>55 win_fn_out_t outdata[WIN_LEN], 56 win_fn_in_t indata[WIN_LEN]) 57 { 58 // Instantiate a window fn object - types and params define</pre>	
		Ŧ

Figure 94: C Code for C Validation Lab 3

- 2. Hold down the Control key and click the window_fn_top.h on line 45 to open this header file.
- 3. Scroll down to view the type definitions (Figure 95).

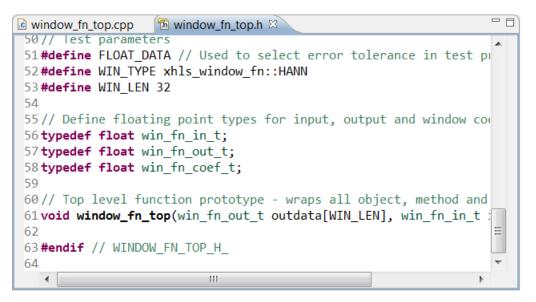


Figure 95: Type Definitions for C Validation Lab 3

This design uses standard C/C++ floating-point types for all data operations. Vivado High-Level Synthesis can synthesize floating-point types directly into hardware, provided the operations are standard arithmetic operations (+, -, *, % etc.).

When using math functions from math.h or cmath.h, refer to the *Vivado HLS User Guide (ug902)* for details on which math functions are supported for synthesis.

- 4. Click the Run C Simulation toolbar button to open the C Simulation Dialog box
- 5. Accept the default setting (no options selected) and click **OK**.





The Console pane shows that the design simulates with the expected results.

Step 3: Synthesize the Design and Review Results

1. Click the **Run C Synthesize** toolbar button to synthesize the design to RTL.

When synthesis completes, the synthesis report opens automatically. **Figure 96** shows the synthesis report.

			h.rpt 🖾				
erformar	nce Estir	nates					
Timing	(ns)						
🗉 Sumi							
Clock		rget	Estimat	ed l	Uncertainty		
defau		5.00		75	0.63		
Latency	(clock	cycles)					
		cycles					
	ency	Int	erval				
min	max	min	max	Тур	e		
257	257	258	258	non			
□ Deta			1				
⊞ Ins	" stance						
⊞ In: ⊡ Lo	stance						
	stance						
	stance op	tes					
	stance op Estima	tes					
. Lo tilization	stance op Estima ry		И_18К	DSP4	8E FF	LUT	
E Lo tilization Summa	estance op Estima iry		И_18К	DSP4	8E FF C		
Lo tilization Summa Nan	estance op Estima iry		Л_18К				
Lo tilization Summa Nam Expression	stance op Estima ry ne on			-	C) 12	
E Lo tilization Summa Nam Expression FIFO Instance Memory	stance op Estima ry ne on	BRAN -		-	C -) 12	
E Lo tilization Summa Nam Expression FIFO Instance Memory Multiple	stance op Estima ry ne on on	BRAN -	1	-	- 3 151) 12	
E Lo tilization Summa Nam Expression FIFO Instance Memory Multiple Register	stance op Estima ry ne on	BRAN - -	1	-	- 3 151 -) 12 - 1 325 - 6	
E Lo tilization Summa Nam Expression FIFO Instance Memory Multiple	stance op Estima ry ne on	BRAN - -	1	-	- 3 151 - -) 12 - 1 325 - 6	

Figure 96: Synthesis Report for Floating Point Design

Instances in the top-level design account for most of the area used.

2. Scroll down the report and expand the Instances in the Details section of the Area Estimates (Figure 97).



	Detail						Ξ
Ξ	E Instance						
	Instance	Module	BRAM_18K	DSP48E	FF	LUT	
	window_fn_top_fmul_32ns_32ns_32_5_max_dsp_U1	window_fn_top_fmul_32ns_32ns_32_5_max_dsp	0	3	151	325	
	Total	1	0	3	151	325	
	Memory						
	Shift register						
	Expression						
	Multiplexer						
	Register						
							T

Figure 97: Area Details for Floating Point Design

The details show this is a floating-point multiplier (fmul). Floating-point operations are costly in terms of area and clock cycles. The Analysis perspective (Figure 98) shows this operator is also responsible for most of the clock cycles (five of the eight states it takes to execute the logic created by loop winfn).

More details on using the Analysis perspective are available in the tutorial **Design Analysis**. For the purposes of understanding this design, two of the operations in the first state are two-cycle read-from-memory operations, and the operation in the final state is a write-to-memory operation.

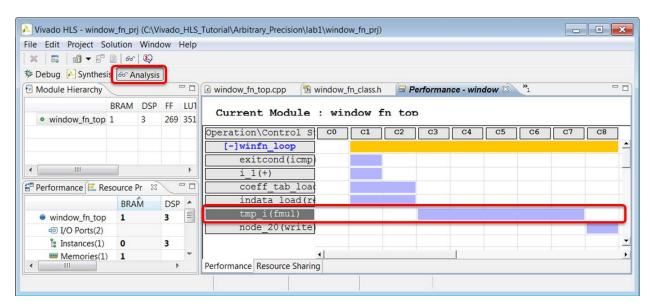


Figure 98: Performance Details for Floating Point Design

3. Exit the Vivado HLS GUI and return to the command prompt

www.xilinx.com



Arbitray Precision: Lab 2

Review a Design using Arbitrary Precision types

Introduction

This lab exercise uses the same design as Lab 1, however, the data types are now arbitrary precision types. You first review the design and then examine the synthesis results.

Step 1: Create and Simulate the Project

- 1. From the Vivado HLS command prompt used in Lab 1, change to the lab2 directory as shown in **Figure 99**.
- 2. Create a new Vivado HLS project by typing:

vivado_hls -f run_hls.tcl

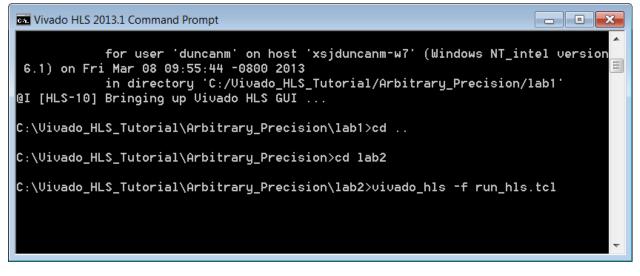


Figure 99: Setup for Interface Synthesis Lab 2

3. Open the Vivado HLS GUI project by typing vivado_hls -p window_fn_prj.

www.xilinx.com





4. Open the Source folder in the explorer pane and double-click **window_fn_top.cpp** to open the code as shown in **Figure 100**.

င်္ခ Explorer 🛛 🛛 🤣 🗖 🗖	window_fn_top.cpp X	
😂 window_fn_prj	44 ************************************	ko 🔺
🗊 Includes	45 #include "window_fn_top.h" // Provides typedefs and params	
Source	46	
window_fn_top.cpp	47// Include the entire xhls_window_fn namespace so that scope resolution -	
🕮 Test Bench	<pre>48// i.e. prepending xhls_window_fn:: to everything is not necessary 49 using namespace xhls window fn;</pre>	
╆ solution1	50	
constraints	51//Vivado HLS requires a top-level function definition that wraps all obje	20
🚿 directives.tcl	52// instantiations and method calls to be synthesized as well as mapping	
缓 script.tcl	53// the top-level I/O (function arguments) into/out of the methods/function	
🗁 csim	54 void window_fn_top(Ξ
🗁 build	55 win_fn_out_t outdata[WIN_LEN],	
🗁 report	56 win_fn_in_t indata[WIN_LEN]) 57{	
	58 // Instantiate a window_fn object - types and params defined in header	• •
	•	

Figure 100: C Code for Arbitrary Precision Lab 2

- 5. Hold the Control key down and click **window_fn_top.h** on line 45 to open this header file.
- 6. Scroll down to view the type definitions (Figure 101).

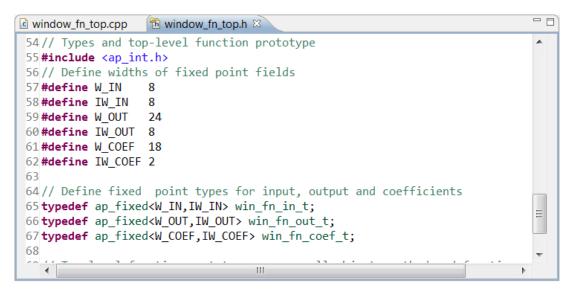


Figure 101: Type Definitions for Arbitrary Precision Lab 2

This header file, window_fn_top.h, is the only file that is different from Lab 1. The data types have been changed to ap_fixed point types, which are similar to float and double types in that they support integer and fractional bit representations. These data types are defined in the header file ap_fixed.h. The definitions in the header file define sizes of the data types:

- The first term defines the total word length.
- The Second term defines the number of integer bits.
- The number of fractional bits is therefore the first term minus the second.





When you revise C code to use arbitrary precision types instead of standard C types, one of the most common changes you must make is to reduce the size of the data types. In this case, you change the design to use 8-bit, 24-bit, and 18-bit words instead of 32-bit float types. This results in smaller operators, reduced area, and faster timing.

Similar optimizations help when you change more common C types such as int, short, and char. For example, changing a data type that only needs to be 18-bit from int (32-bit) ensures that only a single DSP48 is required to perform any multiplications.

In both cases, you must confirm that the design still performs the correct operation and that it does so with the required accuracy. The benefit of the arbitrary precision types provided with Vivado High-Level Synthesis is that *you can simulate* the updated C code to confirm its function and accuracy.

- Open the Test Bench folder in the Explorer pane and double-click window_fn_top_test.cpp to open the code.
- 8. Scroll down to see the view shown in Figure 102.

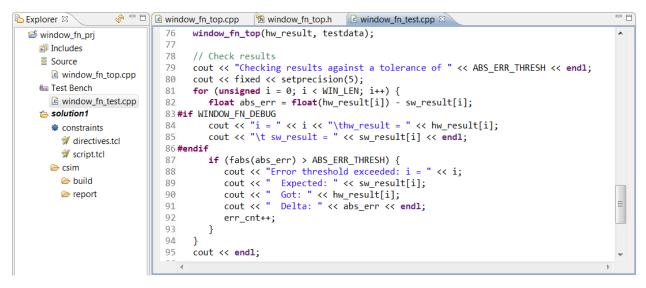


Figure 102: Test Bench for Arbitrary Precision Lab 2

The test bench for this design contains code to check the accuracy of the results. The expected results are still generated using float types. The result checking verifies that the results are within a specified range of accuracy (in this case, within 0.001 of the expected result).

This allows the updated design to be validated quickly and efficiently in C, with fast compile and run times.

9. Click the Run C Simulation toolbar button to open the C Simulation Dialog box

10. Accept the default setting (no options selected) and click OK.

The Console pane shows the results of the C simulation. With the updated data types, the results are no longer identical to the expected results. However, they are within tolerance.





🗐 Console 🖾 🥺 Errors 💩 Warnings		🔲 🗶 🎉 🕞 🚮 💭 💯 🗖 🗋
<terminated> window_fn_prj.Debug [C/C++</terminated>	Application] C:\Vivado_HLS_Tutorial\Arbitrary_Precision\lab2\windo	ow_fn_prj\solution1\csim\build
i = 24 hw_result = 32 sw_resu i = 25 hw_result = 25.757 i = 26 hw_result = 19.754 i = 27 hw result = 14.222	lt = 32.00000 sw_result = 25.75711 sw_result = 19.75413 sw result = 14.22175	*
i = 28 hw_result = 9.3721 i = 29 hw_result = 5.3926 i = 30 hw_result = 2.4355 i = 31 hw result = 0.61426	<pre>sw_result = 9.37258 sw_result = 5.39297 sw_result = 2.43585 sw result = 0.61487</pre>	
Test Passed		=
٩		Þ

Figure 103: C Simulation Results for Fixed Point Types

Step 2: Synthesize the Design and Review Results

1. Click the **Run C Synthesis** toolbar button to synthesize the design to RTL.

When synthesis completes, the synthesis report opens automatically. **Figure 104** shows the synthesis report.





forman	ce Estin	nates	h.rpt 🛛			
iming		inates				
∃ Sumr						
Clock		get	Estimat	ed l	Incertainty	
defau		5.00		.49	0.63	
Latency	(clock	(vcles)				
□ Sumr		-,,				
	ency	Int	terval			
min	max	min		Тур	e	
193	193	194		non		
Detai	1					
Detai						
• Ins	tance					
	tance					
• Ins	tance op	tes				
E Ins E Lo	tance op Estima	tes				
	tance op Estima		M 18K	DSP4	3F FF	IUT
 Ins E Lo lization Summa Nam 	tance op Estimat ry e	BRAN	M_18K	DSP4	BE FF 0	LUT 12
	tance op Estimat ry e	BRAN				
 Ins Isation Loc Summa Nam Expression 	tance op Estimat ry e	BRAN		-	C	12
Insert I	tance op Estimat ry e	BRAN	-	-	0 -	12
Ilization Summa Nam Expression IFIFO	tance op Estimat ry e on	BRAN	-	-	0 - 1 18	12 - 5
Ilization Summa Nam Expressio IFFO Instance Memory	tance op Estimat ry e on	BRAN	- - - 1	-	0 - 1 18 -	12 - 5 - 6
Ilization Summa Nam Expressio FIFO Nemory Multiple	tance op Estimat ry e on ker	BRAN	- - - 1		0 - 1 18 - -	12 - 5 - 6

Figure 104: Synthesis Report for Fixed Point Design

Note that through use of arbitrary precision types, you have reduced both the latency and the area (by 25% and 60% respectively), and the operations in the RTL hardware are no larger than necessary.

2. Scroll down the report to the Interface summary (Figure 105).

Figure 105 shows the data ports are now 8-bit and 24-bit.





erface						
Summary						
RTL Ports	Dir	Bits	Protocol	Source Object	С Туре	
ap_clk	in	1	ap_ctrl_hs	window_fn_top	return value	
ap_rst	in	1	ap_ctrl_hs	window_fn_top	return value	
ap_start	in	1	ap_ctrl_hs	window_fn_top	return value	
ap_done	out	1	ap_ctrl_hs	window_fn_top	return value	
ap_idle	out	1	ap_ctrl_hs	window_fn_top	return value	
ap_ready	out	1	ap_ctrl_hs	window_fn_top	return value	
outdata_V_address0	out	5	ap_memory	outdata_V	array	
outdata_V_ce0	out	1	ap_memory	outdata_V	array	
outdata_V_we0	out	1	ap_memory	outdata_V	array	
outdata_V_d0	out	24	ap_memory	outdata_V	array	
indata_V_address0	out	5	ap_memory	indata_V	array	
indata_V_ce0	out	1	ap_memory	indata_V	array	
indata_V_q0	in	8	ap_memory	indata_V	array	

Figure 105: Fixed Point Interface Summary

3. Exit the Vivado HLS GUI and return to the command prompt.

Conclusion

In this tutorial, you learned:

- How to update the existing standard C types to Vivado High-Level Synthesis arbitrary precision types.
- The advantages in terms of hardware performance and area of using bit-accurate datatypes.





Chapter 6 Design Analysis

Overview

The general design methodology for creating an RTL implementation from C, C++ or SystemC includes the following tasks:

- Synthesizing the design.
- Reviewing the results of the initial implementation.
- Applying optimization directives to improve performance.

You can repeat the steps above until the required performance is achieved. Subsequently, you can revisit the design to improve area.

A key part of this process is the analysis of the results. This tutorial explains how to use the reports and the GUI Analysis perspective to analyze the design and determine which optimizations to apply.

This tutorial consists of a single lab exercise that:

- Demonstrates the HLS interactive analysis feature
- Takes you through one design from the initial implementation through six steps and multiple optimizations to produce the final optimized design

As demonstrated throughout the tutorial, performing these steps in a single project gives you the ability to compare the different solutions easily.

Lab1

Synthesize and analyze a DCT design. Use the insights from the design analysis to apply optimizations and judge the effectiveness of the optimization.

Tutorial Design Description

You can download the tutorial design file from the Xilinx Website. Refer to the information in **Obtaining the Tutorial Designs**.

This tutorial uses the design files in the tutorial directory **Vivado_HLS_Tutorial\Design_Analysis.**



The sample designs used in the lab exercise is a 2-D DCT function. To highlight the design analysis feature, your goal is to have this design operate with an interval of 100 or less. The design should be able to process a new set of input data at least every 100 clock cycles.

Lab 1: Design Optimization

This exercise explains the basic operations of the GUI Analysis perspective and how you can use it to drive design optimization.



IMPORTANT: The figures and commands in this tutorial assume the tutorial data directory **Vivado_HLS_Tutorial** is unzipped and placed in the location **C:\Vivado_HLS_Tutorial**.

If the tutorial data directory is unzipped to a different location, or if it is on a Linux system, adjust the few pathnames referenced to the location at which you placed the **Vivado_HLS_Tutorial** directory.

Step 1: Create and Open the Project

- 1. Open the Vivado HLS Command Prompt.
 - a. On Windows click Start > All Programs > Xilinx Design Tools > Vivado 2013.3 > Vivado HLS > Vivado HLS 2013.3 Command Prompt (Figure 106).
 - b. On Linux, open a new shell.



Figure 106: Vivado HLS Command Prompt

- 2. Using the command prompt window (**Figure 107**), change the directory to the Design Analysis tutorial, lab1.
- 3. Execute the Tcl script to setup the Vivado HLS project, using the command vivado_hls –f run_hls.tcl, as shown in Figure 107.



115

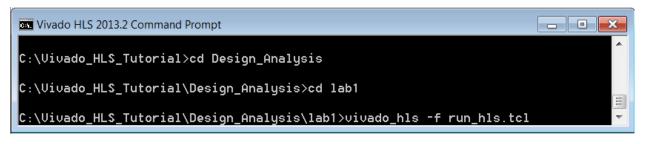


Figure 107: Setup the Design Analysis Tutorial Project

4. When Vivado HLS completes, open the project in the Vivado HLS GUI using the command vivado_hls –p dct_prj as shown in **Figure 108**.

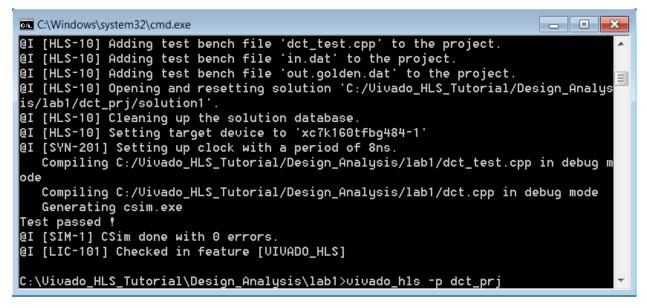
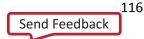


Figure 108: Open Design Analysis Project for Lab 1

Step 2: Review the source Code and Create the Initial Design

1. Double-click the file dct.cpp in the Source folder to open the source code for review. .

This example uses a DCT function. Figure 109 shows an overview of this code.



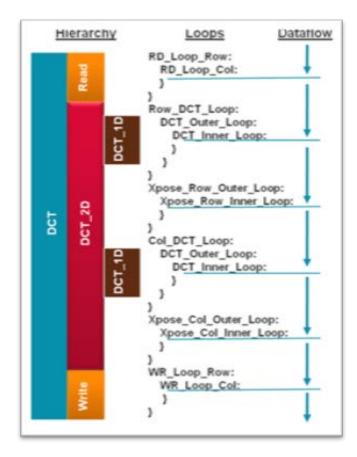


Figure 109: Overview of the DCT design

- The right side of Figure 109 shows the code hierarchy.
 - Top-level function dct has three sub-functions: read_data, dct_2d and write_data.
 - Function dct_2d has a single sub-function dct_1d.
- The center of Figure 109 shows loops inside each of the functions.
- The right side of Figure 109 shows the how the data is processed through the functions and loops.
 - The read_data function executes, and the data is processed through loop RD_Loop_Row, which has a sub-loop RD_Loop_Col.
 - After the read_data function complete, function dct_2d executes.
 - In function dct_2d, Row_DCT_Loop processes the data. Row_DCT_Loop has two nested loops inside it: DCT_output_loop and DCT_inner_loop.
 - DCT_inner_loop calls function dct_1d.

And so on, until the function write_data processes the data.

2. Click the **Run C Synthesize** toolbar button to synthesize the design to RTL.





Step 3: Review the performance using the Synthesis Report

When synthesis completes, the synthesis report opens automatically. **Figure 110** shows the performance section of the report.

ormance	e Estim	nates										
ming (r	ıs)											
Summa	ary											
Clock	Targ	get l	Estimat	ed	Uncert	ainty						
default	8	.00	5.	79		1.00						
tency (clock c	ycles)										
Summa	ary	-										
Later	ncy	Int	erval									
min	max	min	max	Ту	/pe							
959	3959	3960	3960	nc	one							
Detail												
Detail												
	ance											
	ance				Late	ency	Int	erval				
∃ Insta	ance Instanc	e	Mod	dule	Late	ency max	Int min	erval max	Туре			
∃ Insta	Instanc	:e _fu_152				1			Type none			
∃ Inst a	Instanc dct_2d_	-			min	max	min	max				
□ Insta	Instanc dct_2d_	-		2d	min	max	min	max 3668				
□ Insta grp_c □ Loop	Instanc dct_2d_	fu_152	dct_2	2d	min 3668	max	min 3668	max 3668	none n Interval	Trip Count	Pipelined	
□ Insta grp_c □ Loop	Instanc dct_2d_ p	fu_152	dct_2	2d ncy	min 3668	max 3668	min 3668	max 3668 Initiation	none n Interval	Trip Count 8	Pipelined	
□ Insta grp_c □ Loop Loo	Instanc dct_2d_ p op Nan	fu_152 ne Row	dct_2 Late min	2d ncy max	min 3668	max 3668	min 3668 ency	max 3668 Initiation achieved	none n Interval target			
□ Insta grp_c □ Loop Loop + RD - WR	Instanc dct_2d_ op Nan Loop_I	fu_152 ne Row _Col Row	dct_2 Late min 144	2d ncy max 144	min 3668	max 3668	min 3668 ency 18	max 3668 Initiation achieved	none n Interval target	8	no	

Figure 110: Report for initial DCT Design

Figure 110 highlights the following information.

- The clock frequency of 8 ns has been met.
- The top-level design takes 3539 clock cycles to write all the outputs.
- You can apply new inputs after 3560 clock cycles. This is one clock cycle after the output data has been written. This immediately reveals that the design is not pipelined, but this fact is also noted in the report.
- The top level has a single instance, which has a latency and initiation interval of 3668 and 3669 respectively.
 - This block also has no pipelining and accounts for most of the clock cycles.
- Noticethat the functions read_data and write_data are not noted here as instances of the top level.





- **Figure 111** shows that, during synthesis, these blocks were automatically inlined (the hierarchy was removed).
- High-level synthesis might automatically inline small functions to improve the quality of results (QoR). You can prevent this by adding the Inline directive with the -off option.

	Console 💷 🕐 Errors 🔺 Warnings 🛛 🕞 🕯	1 - 0
Viva	ado HLS Console	
ei ei ei ei	<pre>[HLS-10] Checking synthesizability [HLS-10] Starting code transformations [XFORM-602] Inlining function 'read_data' into 'dct' (dct.cpp:89) automatically. [XFORM-602] Inlining function 'write_data' into 'dct' (dct.cpp:94) automatically. [HLS-111] Elapsed time: 16.722 seconds; current memory usage: 30.5 MB. [HLS-10] Starting hardware synthesis</pre>	
eI	[HLS-10] Synthesizing 'dct'	*
4	III	

Figure 111: Automatic Inlining for Functions

- The loops in the read_data and write_data functions are therefore implemented at the top level and are reported as loops in the top-level function (Figure 110).
- Each loop has a latency of 144 clock cycles. (Because the loops are not pipelined, there is no initiation interval.)
- Using RD_Loop_Row as an example, you can see why the loop latency is 144.
 - Sub-loop RD_Loop_Col has a latency of 2 cycles for each iteration of the loop (iteration latency) and a tripcount of 8: 2 x 8 = 16 clock cycles total latency for the loop.
 - From RD_Loop_Row, it takes 1 clock to enter loop RD_Loop_Col and 1 clock cycle to return to RD_Loop_Row. The iteration latency for RD_Loop_Row is therefore (1 + 16 +1) 18 clock cycles.
 - RD_Loop_Row has a tripcount of 8 so the total loop latency is 8 x 18 = 144 clock cycles.
- The total latency for the dct block is therefore:
 - o 144 clocks for RD_Loop_Row.
 - Plus 3668 clock cycles for dct_2d.
 - Plus 144 clock cycles for WR_Loop_Row.
 - Plus a clock cycle to enter each block.

To review the details of the instantiated sub-blocks dct_2d and dct_1d, open their respective reports from the syn/reports folder under solution1 in the Explorer pane.

You can also use the design analysis perspective to review these details in a more interactive manner.



🐮 XILINX:

Step 4: Review the Performance using the Analysis Perspective

Invoke the Analysis perspective any time after synthesis completes.

1. Click the **Analysis** perspective button (Figure 112 112) to begin interactive design analysis.

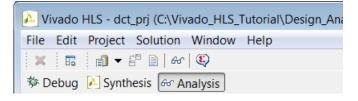


Figure 112: Opening the Analysis perspective

The Analysis perspective consists of five panes, each of which is highlighted in **Figure 113**. You use all of these in the tutorial. The module and loops hierarchies are shown expanded (by default, they are shown collapsed).

	Analysi:	S							
Adule Hierarchy							- 0	Schedule Viewer - dct	
BR	AM DSP	FF	LUT	Latency	Interval	Pipeline type			
• dct 6	1		334	3959	3960	none		Current Module : dct	
• dct_2d 4	1	128	248	3668	3669	none		Operation\Control St C0 C1 C2 C3 C4	C
• dct_1d 1	1	55	104	209	210	none		[+]RD Loop Row	
								dct 2d(function)	
								[+]WR Loop Row	
	-								
erformance Profile	E Res	ource	Profi	e			- 0		
erformance Profile	_				n Interval	Iteration Late			
	_	d La			n Interval	Iteration Late			
	Pipeline -	d La	itency 159	Initiatio	n Interval	Iteration Late - 18			
• dct	Pipeline - no	d La 39 14	itency 159	Initiatio 3960	n Interval	2 .	Trip count		
 dct RD_Loop_Row 	Pipeline - no no	d La 39 14 16	itency 159 14	Initiatio 3960 -	n Interval	- 18	Trip count - 8		
RD_Loop_Col	Pipeline - no no no	d La 39 14 16	itency 159 14 5	Initiatio 3960 - -	n Interval	- 18 2	Trip count - 8 8		
 dct RD_Loop_Row RD_Loop_Col WR_Loop_Row 	Pipeline - no no no	d La 39 14 16 14	itency 159 14 5	Initiatio 3960 - -	n Interval	- 18 2 18	Trip count - 8 8 8 8		
 dct RD_Loop_Row RD_Loop_Col WR_Loop_Row 	Pipeline - no no no	d La 39 14 16 14	itency 159 14 5	Initiatio 3960 - -	n Interval	- 18 2 18	Trip count - 8 8 8 8		
 dct RD_Loop_Row RD_Loop_Col WR_Loop_Row 	Pipeline - no no no	d La 39 14 16 14	itency 159 14 5	Initiatio 3960 - -	n Interval	- 18 2 18	Trip count - 8 8 8 8		
 dct RD_Loop_Row RD_Loop_Col WR_Loop_Row 	Pipeline - no no no	d La 39 14 16 14	itency 159 14 5	Initiatio 3960 - -	n Interval	- 18 2 18	Trip count - 8 8 8 8		

Figure 113: Overview of the Analysis perspective

Use the Module Hierarchy pane to navigate through the hierarchy. The Module Hierarchy pane shows both the performance and area information for the entire design. The Performance Profile pane shows the performance details for this level of hierarchy. The information in these two panes is similar to the information you reviewed earlier in the report (for the top-level dct block).

The Performance view is also shown (on the right side of **Figure 113**). This view shows how the operations in this particular block are scheduled into clock cycles.

• The left column lists the resources.





- Sub-blocks are green.
- Operations resulting from loops in the source code are yellow.
- Standard operations are purple.
- Notice that the dct has three main resources:
 - A loop called RD_Loop_Row. The plus symbol (+) indicates that the loop has hierarchy and that you can expand the loop to view it.
 - A sub-block called dct_2d.
 - A loop called WR_Loop_Row.

The top row lists the control states in the design. Control states are the internal states High-Level Synthesis uses to schedule operations into clock cycles. There is a close correlation between the control states and the final states in the RTL Finite State Machine (FSM), but there is no one-to-one mapping.

 Click loop RD_Loop_Row and sub-loop RD_Loop_Col to fully expand the loop hierarchy (Figure 114).

	Operation\Control S	C0	C1	C2	C3	C4	C5
1	∃RD Loop Row						
2	exitcond1 i(icmp)						
3	r(+)						
4	⊡RD Loop Col						
5	exitcond i(icmp)						
6	c (+)						
7	tmp 5 i(+)						
8	input load(read)						
9	p addr1(+)						
10	node 41(write)						
11	dct 2d(function)						
12-21	∃WR Loop Row						

Figure 114: Expanded View of RD_Loop_Row

From this, you can see that in the first state (C1) of the RD_Loop_Row, the loop exit condition is checked and an add operation performed. This addition is likely the counter for the loop iterations, and we can confirm this.

3. Select the adder in state C1, right-click and select **C source** code (Figure 115).





This opens the C source code to highlight which operation in the C source created this adder. From the details on screen (also shown in **Figure 115**), you can determine it is indeed the loop counter. It is the only addition on this line, and the variable is named "r".

E Performance	- dct 🛛						- 8	C Source 🛛 🗖 🗖
Current Mo	odule : dct							File: C:\Vivado_HLS_Tutorial\Design_Analysis\lab1\dct
	Operation\Control S	C0	C1	C2	C3	C4	C5	101
1	∃RD Loop Row							102 RD_Loop_Row:
2	exitcond1 i(icmp)							103 for (r = 0; r < DCT_SIZE; r++) {
3	r(+)							104 RD Loop Col:
4	∃RD Loop Col			Goto So	ource			105 for (c = 0; c < DCT_SIZE; c++)
5	exitcond i(icmp)			Goto Ve	erilog			106 $buf[r][c] = input[r * DCT SIZE + c];$
6	c (+)			Goto VI	HDL			107 }
7	tmp 5 i(+)		_					108}
8	input load(read)							109
9	p addr1(+)							110 void write data(short buf[DCT SIZE][DC]
10	node 41(write)							111{
11	dct 2d(function)							112 int r, c;
12-21	∃WR Loop Row							113
								114 WR_Loop_Row:
								115 for (r = 0; r < DCT_SIZE; r++) {
Performance Res	source							• III •

Figure 115: C Source Code View

In the next state of loop RD_Loop_Row (C2), loop RD_Loop_Col starts to execute..

4. Click operations in the RD_Loop_Col to see the source code highlighting update.

This should help confirm your understanding of how the operations in the C source code are implemented in the RTL.

- The loop exit condition is checked.
- This is an adder for loop count variable "c".
- A read from a RAM performed (one cycle to generate the address, one cycle to read the data).
- o A write operation is performed to a RAM.

Loops in the Performance view mean that the design iterates around these states multiple times. The number of iterations is noted as the loop tripcount and shown in the Performance Profile.

To improve performance, these loops should be pipelined. You can review the rest of the design for other performance optimization opportunities.

- 5. Click the C Source Code pane to close this window.
- 6. In the Module Hierarchy, click the function dct_2d to navigate into the view for this function (Figure 116).



Design Analysis



	ct Solu			JW H	elp													
× 6 10.				1														
Debug 🛃 Syr	10000000000	ee' An	Hysis	5						-	rmance - dct_2d							
										- Репо	rmance - dct_2d &							
							Pipeline typ	e		Curr	ent Module : dct > dct 2d							
dct dct_2d				331 247		3960 3668	none				Operation\Control Step	0	61	C2	0	64	CE	06
• dct_2d				107		209	none			1		1.0	1 61	1.62	Lat	1.4	1.5	1.0
• act_	1	1	82	107	209	209	none				exitcond7(icmp)							
											i 4(+)							
										4	dct 1d(function)							
											*Xpose Row Outer Loop =Col DCT Loop							
										15	exitcond4(icmp)	-			-	-	-	-
											i S(+)							
										18						-		
											EXpose Col Outer Loop							-
Performance P	rofile 2		Resc	ource P	rofile				- 0									
			Pi	peline	d Laten	cy Initia	tion Interval	Iteration Latency	Trip count									
dct_2d					3668				-									
· Row_D	T_Loop		nc	2	1688			211	8									
 Xpose_I 	Row_Ou	ter_Lo	op no		144			18	8									
· Col_DC	T_Loop		no	5	1688			211	8									
 Xpose_0 	Col_Oute	er_Loo	p no	5	144			18	8									
										Death	ance Resource							

Figure 116: DCT_2D Performance View

Again, you can see a number of loops (shown in yellow in **Figure 116**). Loops ensure the design will have small area but the design will take multiple iterative states to complete: each iteration of the loop will complete before the next iteration starts.

You can pipeline the loops to improve the performance. The details in the Performance Profile show that most of the latency is caused by loops Row_DCT_Loop and Col_DCT_Loop.

7. Click loops Row_DCT_LOOP and Col_DCT_LOOP in the performance viewer to fully expand them, as shown in Figure 117.

Expanding these loops in Performance view shows both loops call function dct_1d. Unless this function itself is pipelined, there is no benefit in pipelining the loop. TheModule Hierarchy shows the interval for dct_1d is 210 clock cycles, which means it can only accept a new input every 210 clock cycles.

- 8. In the Module Hierarchy, click function dct_1d to navigate into the view for this function.
- 9. Expand the loops in the Performance Profile and Performance view to see the view shown in Figure 117.



123



			-													
EDebug 🛛 Synthe		nalysis	;													
Module Hierarchy									- 0	Perform	nance - dct_1d 🛿					
	BRAM	DSP	FF	LUT	Latency	Interval	Pipeline type			_						
dct	6	1	183	331	3959	3960	none			Currer	nt Module : dct > dct	2a > a	ςτ τα			
ø dct 2d	4	1	135	247	3668	3668	none				Operation\Control S	CO	C1	C2	C3	C4
• dct 1d	1	1		107		209	none			1	tmp 11 read(read)					
- uci_iu	-	-	02	107	205	205	none			2	tmp 1 read(read)					
										3	BDCT Outer Loop					
										4	exitcond1(icmp)					
										5	k 1(+)					
										6	BDCT Inner Loop					
										7	exitcond(icmp)					
										8	n 1(+)					
										9	dct coeff tab					
										10	dct coeff tab					
										11	p addr1(+)					
										12	<pre>src load(read)</pre>					
										13	tmp 8(*)					
										14	tmp 5(+)					
Performance Prof	ile 🛛	E Res	ource	Profil	e					14						
Performance Prof	ile 🛛									15	tmp 2(+)					
	ile 🛛	Pipe		Late	ency Ini		erval Iteration	Latency			tmp 2(+) p addr3(+)					
▲ ● dct_1d					ency Ini		rval Iteration	Latency	Trip count	15	tmp 2(+)					
		Pipe		Late	ency Init			Latency		15 16	tmp 2(+) p addr3(+)					
▲ ● dct_1d	_Loop	Pipe - no		Late 209	ency Init		-	Latency	-	15 16	tmp 2(+) p addr3(+)					
-	_Loop	Pipe - no		Late 209 208	ency Ini 209 -		- 26	Latency	- 8	15 16	tmp 2(+) p addr3(+)					
 dct_1d DCT_Outer 	_Loop	Pipe - no		Late 209 208	ency Ini 209 -		- 26	Latency	- 8	15 16	tmp 2(+) p addr3(+)					
 dct_1d DCT_Outer 	_Loop	Pipe - no		Late 209 208	ency Ini 209 -		- 26	Latency	- 8	15 16	tmp 2(+) p addr3(+)					
 dct_1d DCT_Outer 	_Loop	Pipe - no		Late 209 208	ency Ini 209 -		- 26	Latency	- 8	15 16	tmp 2(+) p addr3(+)					
 dct_1d DCT_Outer 	_Loop	Pipe - no		Late 209 208	ency Ini 209 -		- 26	Latency	- 8	15 16	tmp 2(+) p addr3(+)					
 dct_1d DCT_Outer 	_Loop	Pipe - no		Late 209 208	ency Ini 209 -		- 26	Latency	- 8	15 16	tmp 2(+) p addr3(+)					
 dct_1d DCT_Outer 	_Loop	Pipe - no		Late 209 208	ency Ini 209 -		- 26	Latency	- 8	15 16	tmp 2(+) p addr3(+)					
 dct_1d DCT_Outer 	_Loop	Pipe - no		Late 209 208	ency Ini 209 -		- 26	Latency	- 8	15 16	tmp 2(+) p addr3(+)					
 dct_1d DCT_Outer 	_Loop	Pipe - no		Late 209 208	ency Ini 209 -		- 26	Latency	- 8	15 16	tmp 2(+) p addr3(+)					
 dct_1d DCT_Outer 	_Loop	Pipe - no		Late 209 208	ency Ini 209 -		- 26	Latency	- 8	15 16	tmp 2(+) p addr3(+)					

Figure 117: DCT_1D Performance View

In Figure 117 you can see a series of nested loops which can be pipelined.

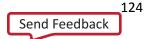
You can choose to do one of the following:

- You can pipeline the function and then pipeline the loop that calls it. (Because the function is pipelined, the loop can take advantage of using a pipelined part.)
- You can pipeline the loops within this function and simply make this function execute faster.

Pipelining the function unrolls all the loops within it, and thus greatly increases the area. If the objective is to get the highest possible performance with no regard for area, this may be the best optimization to perform.

You can find more details on pipelining loops and functions in the tutorial **Design Optimization**. For this case, the approach is to optimize the loops and keep the area at a minimum.

10. Click the **Synthesis** perspective button to return to the main synthesis view.



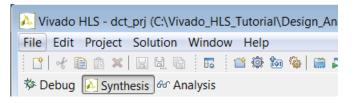


Figure 118: Re-Opening the Synthesis Perspective

Step 5: Apply Loop Pipelining & Review for Loop Optimization

In this step, you create a new solution and add pipelining directives to the loops.

When pipelining nested loops, it is generally best to pipeline the inner-most loop. Typically, High-Level Synthesis can generally flatten the loop nest automatically (allowing the outer loop to simply feed the inner loop). For more information on why it is better to perform certain loop optimizations rather than others, refer to the tutorial "Design Optimization".

- 1. Select the **New Solution** toolbar button or use the menu **Project > New Solution** to create a new solution.
- 2. Click **OK** and accept the defaults.
- 3. Ensure that you can see the C source code in the Information pane.
- 4. In the **Directives** tab, add a pipeline directive to loop DCT_Inner_Loop in function dct_1d.
 - a. Right-click DCT_Inner_Loop in the Directives pane and select Insert Directive
 - b. In the **Directives Editor** dialog box activate the **Directives** drop-down menu at the top and select **PIPELINE**.
 - c. Click **OK** and select the default maximum pipeline rate (II=1)
- 5. Repeat step 4 for the following loops:
 - a. In function dct_2d loop Xpose_Row_Inner_Loop
 - b. In function dct_2d loop Xpose_Col_Inner_Loop
 - c. In function read_data loop RD_Loop_Col
 - d. In function write_data loop WR_Loop_Col

The **Directive** pane shows the following (highlighted) optimization directives applied.





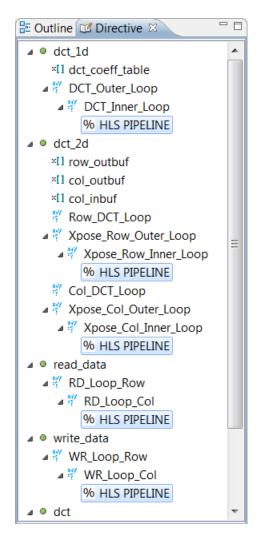


Figure 119: Optimization Directives for DCT Loop Pipelines

- 6. Click the Run C Synthesize toolbar button to synthesize the design to RTL.
- When synthesis completes, use the Compare Reports toolbar button or the menu Project > Compare Reports to compare solutions 1 and 2.

Figure 120 shows the results of comparing solution1 and solution2. Pipelining the loops has improved the latency of the design with an almost 50% reduction in solution2.



🖥 compare r	eports	×						
Performan	ce Estin	nates						*
🗆 Timing ((ns)							
Clock			soluti	on1	solutio	on2		_
default	Target		8.00		8.00			
	Estima	ted	5.79		5.80			Ξ
Latency	(clock o	cycles	5)					
		solu	ition1	solu	ition2			
Latency	min	395	9	197	8			
	max	395	9	197	8			
Interval	min	396	0	197	9			
	max	396	0	197	9			÷

Figure 120: DCT Solution1 and Solution2 Comparison

Next, you once again open the Analysis perspective, analyze the results, and determine whether or not there are more opportunities to for optimization.

8. Click the **Analysis** perspective button to begin interactive design analysis.

When the Analysis perspective opens, you can see that the majority of the latency is still due to block dct_2d. Before proceeding to analyze further, you can review how the loops at this level have been optimized.

The Performance Profile (Figure 121) shows that the latency of both loops has been reduced from 144 clock cycles in solution1 to only 65 clock cycles.

🖥 Performance Profile 🛛 📃 Resour	ce Profile				
	Pipelined	Latency	Initiation Interval	Iteration Latency	Trip count
₄ ● dct	-	1978	1979	-	-
RD_Loop_Row_RD_Loop_Col	yes	64	1	2	64
WR_Loop_Row_WR_Loop_Col	yes	64	1	2	64

Figure 121: DCT Solution2 Performance of top-level Loops

Pipelining loops transforms the latency from

Latency = iteration latency * tripcount

to

Latency = iteration latency + tripcount

HLS also made this possible by automatically performing loop flattening (there is no longer any loop hierarchy). You can see this by reviewing the Console pane, or log file, for solution2. Figure 122 shows the loops that have been automatically optimized.





🖃 Console 🛛 🔮 Errors 💩 Warnings	
Vivado HLS Console	
<u>e= p::::::::::::::::::::::::::::::::::::</u>	
<pre>@I [XFORM-602] Inlining function 'write_data' into 'dct' (dct.cpp:94) automatically.</pre>	<u> </u>
@I [XFORM-541] Flattening a loop nest 'RD_Loop_Row' (dct.cpp:59) in function 'dct'.	
<pre>@I [XFORM-541] Flattening a loop nest 'WR_Loop_Row' (dct.cpp:71) in function 'dct'.</pre>	
<pre>@I [XFORM-541] Flattening a loop nest 'Xpose_Row_Outer_Loop' (dct.cpp:37) in function</pre>	'dct_2d'.
<pre>@I [XFORM-541] Flattening a loop nest 'Xpose_Col_Outer_Loop' (dct.cpp:48) in function</pre>	'dct_2d'. 亘
@I [HLS-111] Elapsed time: 12.191 seconds; current memory usage: 30.6 MB.	
<pre>@I [HLS-10] Starting hardware synthesis</pre>	
MT [HIS-10] Synthesizing 'det'	*

Figure 122: DCT Solution2 Loop Flattening

9. In the Module Hierarchy, click function dct_2d to navigate into the view for this function.

In the Performance Profile you can see that the latency of all the loops has been substantially reduced (Row_DCT_Loop and Col_DCT_loop have been approximately halved from the earlier report in **Figure 116**). However, the majority of the latency is still due to these two loops, each of which calls the dct_1b block.

10. In the Module Hierarchy, click function dct_1d to navigate into the view for this function.

The Performance Profile (Figure 123) shows the loop latencies have been reduced, but there is still a loop hierarchy here. (There is still loop DCT_Outer_Loop, shown in Figure 123, so no loop flattening occured).

🖆 Performance Profile 🛛 🗌	E Resource	Profile			
	Pipelined	Latency	Initiation Interval	Iteration Latency	Trip count
▲ ● dct_1d	-	105	106	-	-
IDCT_Outer_Loop	no	104	-	13	8
DCT_Inner_Loop	yes	10	1	3	8

Figure 123: DCT Solution2 Performance of dct_1d Loops

Viewing these loops in Performance view shows why this loop was not optimized further.

- 11. In the **Performance** view, click loops DCT_Outer_Loop and DCT_Inner_Loop to view the loop hierarchy (Figure 124).
- 12. Select the write operation in state C5.
- 13. Right-click and select Goto Source.





Figure 124 shows that this loop was not flattened because additional operations outside of DCT_Inner_Loop, at the level of DCT_Outer_Loop, prevented loop flattening. One of the operations that prevented loop flattening is highlighted in **Figure 124**, below.

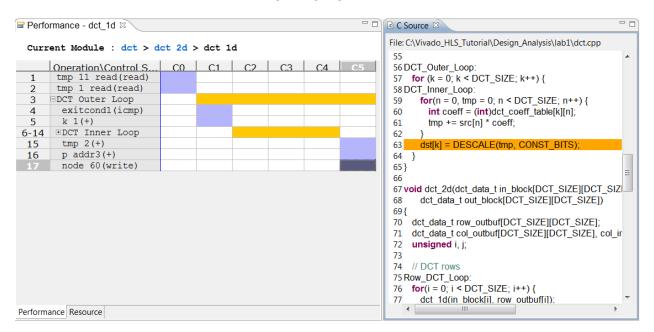


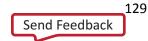
Figure 124: DCT Solution2 dct_1d Performance View

In this case, pipelining the inner-most loop does not provide the biggest benefit. You should pipeline the outer loop instead. This causes the inner loop to be completely unrolled. An increase in area results, but you are still far from the throughput goal of 100 and not yet ready to pipeline the entire function (and see an even greater area increase, as the outer loop is also completely unrolled).

14. Click the **Synthesis** perspective button to return to the main synthesis view.

Step 6: Apply Loop Optimization and Review for Bottlenecks

- 1. Select the **New Solution** toolbar button or use the menu **Project > New Solution** to create a new solution.
- 2. Click **OK** and accept the defaults to create solution3.
- 3. Ensure the C source code is visible in the Information pane.
- 4. In the **Directives** tab
 - a. In function dct_1d, select the pipeline directive on loop DCT_Inner_Loop.
 - b. Right-Click and select Remove Directive.
 - c. Still in function dct_1d, select **loop DCT_Outer_Loop**.
 - d. Right-click and select Insert Directive.





- e. In the **Directives Editor** dialog box activate the **Directives** drop-down menu at the top and select **PIPELINE**.
- f. Click **OK** and select the default maximum pipeline rate (II=1).

The Directive pane should show the following (highlighted) optimization directives applied.

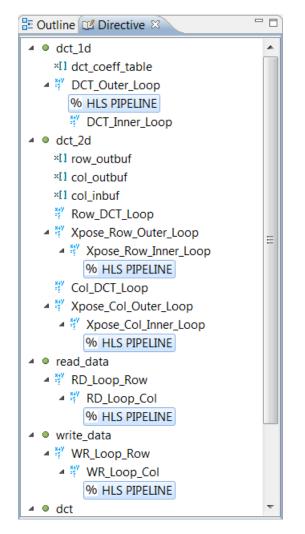
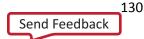


Figure 125: Updated Optimization Directives for DCT Loop Pipelines

- 5. Click the **Run C Synthesize** toolbar button to synthesize the design to RTL.
- 6. When synthesis completes, click the **Compare Reports** toolbar button to compare solutions 2 and 3.

Figure 126 shows the results of comparing solution2 and solution3. Pipelining the outer-loop has in fact resulted in an increase to the performance and the area.

The significant latency benefit is achieved because multiple loops in the design call the dct_1d function multiple times. Saving latency in this block is multiplied because this function is used inside many loops.



compare I	epo	orts	23						
erforman	ce E	istin	nates						
Timing	(ns))							
Clock				so	oluti	on2	soluti	on3	
default	Та	rget	:	8.	00		8.00		
	Es	stimated			80		5.90		
Latency	(clo	ock	cycles	;)					
			solu	ition2		solut	tion3		
Latency	m	nin 1978		8		890			
	m	ax	197	8		890			
Interval	m	in	197	1979 1979		891			:
	m	ax	197			891			
Itilization	Esti	imat	tes						
		so	ution	2	sol	ution	3		
BRAM_18	3K	۲ (13				
DSP48E		1			8				
FF		24	1		55	6			
LUT		45	1		49	8			

Figure 126: DCT Solution2 and Solution3 Comparison

Now that all the loops are pipelined, it is worthwhile to review the design to see if there are performance-limiting "bottlenecks." Bottlenecks are limitations in the flow of data that can prevent the logic blocks from working at their maximum data rate.

Such limitations in the data flow can come from a number of sources, for example, I/O ports and arrays implemented as block RAM. In both cases, the finite number of ports (on the I/O or block RAM) limits the rate at which data can be read or written.

Another source of bottlenecks is data dependencies in the original source code. In some cases, these data dependencies are inherent in how the algorithm operates, as when a calculation cannot be performed until an earlier calculation has completed. Sometimes, however, the use of an optimization directive or a minor change to the C code can remove them.

The first task is to identify such issues in the RTL design. There are a number of approaches you can take:

- Start with the largest latency of interval in the Module Hierarchy report and navigate down the hierarchy to find the source of any large latency or interval.
- Click the Resource Profile to examine I/O and memory usage.
- Click the power of the graphical viewer and look for patterns in the Performance view which indicate a limitation in data flow.





In this case, you will use the latter approach. You can use the Analysis perspective to identify such places in the design quickly.

- 7. Click the **Analysis** perspective button to begin interactive design analysis.
- 8. In the **Module Hierarchy**, ensure module dct is selected.
- 9. In the **Performance** view, expand the first loop in the design as shown in **Figure 127**, RD_Loop_Row_RD_Loop_Col (these loops were flattened and the name is now a concatenation of both loops).

This loop is implemented in two states. The red arrow in **Figure 127** shows the path from the start of the loop to the end of the loop: the arrow is almost vertical (everything happens in two clock cycles) and this loop is well implemented in terms of latency.

	Contraction of the second second second						
	Operation\Control S	C.0	_C1_	_C2_	C.3	C.4	C.5
1	BRD Loop Row RD		1				
2	exitcond flatt						
3	indvar flatten						
4	exitcond i(icmp)						
5	c i mid2(select)						
6	r(+)		· \				
7	r i mid2(select)						
8	tmp 5 i(+)						
9	input load (read)						
10	p addr1(+)						
11	node 47 (write)						
12							
	dct 2d(function)						
	WR Loop Row WR						

Figure 127: Analysis of DCT RD_Loop_Row

- 10. In the Performance view, expand the WR_Loop_Row and perform similar analysis. It is similarly well optimized for latency.
- 11. Double-click function dct_2d and navigate into the dct_2d function.

You can use same analysis process down through the hierarchy. If you perform this analysis you will discover that all the function blocks and loops have a similar optimal (few cycles) implementation, until the dct_1d block is examined.

- 12. In the **Performance** view, double-click function dct_1d and navigate into the dct_1d function.
- 13. Expand the DCT_Outer_Loop to see the view shown in Figure 128.





Figure 128 shows a very different view from the earlier loop schedules (which had only a few cycles of latency). The schedule shows a long drift from input to output (as shown by the red arrow).

	erformance-dct_1d ⊠ urrent Module : dct >	dct 2	d > dct	t 1d						
	Operation\Control S	C0	C1	C2	C.3	C4	C.5	C.6	C.7	C.8
1	tmp 11 read(read)								10.000	
2	tmp 1 read(read)									
3	DCT Outer Loop									
4	exitcond1(icmp)			1						
5	k 1(+)			1						
6	dct coeff tabl									
7	<pre>src load(read)</pre>									
8	tmp 8(*)									
9	dct coeff tabl									
10	<pre>src load 1(read)</pre>									
11	tmp 8 1(*)									
12	dct coeff tabl			1						
13	src load 2(read)									
14	tmp 8 2(*)									
15	dct coeff tabl									
16	<pre>src load 3(read)</pre>									
17	tmp 8 3(*)									
18	dct coeff tabl									
19	src load 4(read)									
20	tmp 8 4(*)									
21	dct coeff tabl									
22	<pre>src load 5(read)</pre>									
23	tmp 8 5(*)									
24	dct coeff tabl			C						
25	<pre>src load 6(read)</pre>			1						
26	tmp 8 6(*)									
27	dct coeff tabl			6						
28	src load 7(read)			1						
29	tmp 8 7(*)									
30	tmp2(+)									
31	tmp3(+)					1		Sec.		
32	tmp1(+)									
33	tmp5(+)									
34	tmp7(+)									
35	tmp6(+)									
36	tmp4(+)									
37	tmp 2(+)									-
38	p addr(+)									-
39	node 114 (write)									

Figure 128: Analysis of dct_1d RD_Loop_Row

There are typically two things that cause this type of schedule: data dependencies in the source code and limitations due to I/O or block RAM. You will now examine the resources sharing in this block.

14. In the **Performance** view, click the **Resource Sharing** tab at the bottom of the window.

15. Expand the Memory Ports, as shown in Figure 129.





Curre	nt Module : dct > dct	2d > d	lct 1d							
	Resource\Control Step	C0	C1	C2	C3	C4	C5	C6	C7	C8
1-5	±I/O Ports									
6-9	Instances									
10	⊡Memory Ports									
11	src		read	read	read	re	ad			
12	dct coeff table 1		re	ad						
13	dct coeff table 0		re	ad						
14	src		read	read	read	re	ad			
15	dct coeff table 5			re	ad					
16	dct coeff table 4			re	ad					
17	dct coeff table 7			re	ad					
18	dct coeff table 6			re	ad					
19	dct coeff table 2			re	ad					
20	dct coeff table 3			re	ad					
21	dst									write
22-38	*Expressions									

Figure 129: Resource Sharing of Memory Ports in dct_1d

The Resource Sharing view shows how the resources in the design are used in different control states.

The rows list the resources in the design. In Figure 129, the memory resources are expanded.

The columns show the control states in which the resource is used. If a resource is active in multiple states, the resource is being re-used in different clock cycles.

Figure 129 shows the memory accesses on BRAM src are being used to the maximum in every clock cycle. (At most, a block RAM can be dual-port and both ports are being used). This is a good indication the design may be bandwidth-limited by the memory resource. To determine if this really is the case, you can examine further.

16. Select one of the read operations for the src block RAM.

17. Right-click and select **Goto Source** to see the view shown in Figure 130.





🖶 Resourd	ce - dct_1d 🛛									- 0	C Source 🛛 🦳 🗖
Currer	nt Module : dct > dct	2d > 0	dct 1d								File: C:\Vivado_HLS_Tutorial\Design_Analysis\lab1\dct.cpp 57 for (k = 0; k < DCT SIZE; k++) {
	Resource\Control Step	C0	C1	C2	C3	C4	C5	C6	C7	C8	58 DCT_Inner_Loop:
1-5	±I/O Ports										59 for(n = 0, tmp = 0; n < DCT_SIZE; n++) {
6-9	Instances										60 int coeff = (int)dct_coeff_table[k][n];
10	∃Memory Ports										61 tmp += src[n] * coeff;
11	src		read	read	read	re	ad				62 }
12	dct coeff table 1		re	ad							63 dst[k] = DESCALE(tmp, CONST_BITS);
13	dct coeff table 0		re	ad							64 }
14	src		read	read	read	re	ad				65 }
15	dct coeff table 5			re	ad						66
16	dct coeff table 4			re	ad						67 void dct_2d(dct_data_t in_block[DCT_SIZE][DCT_SIZ
17	dct coeff table 7			re	ad						68 dct_data_t out_block[DCT_SIZE][DCT_SIZE])
18	dct coeff table 6			re	ad						69 {
19	dct coeff table 2			re	ad						70 dct_data_t row_outbuf[DCT_SIZE][DCT_SIZE];
20	dct coeff table 3			re	ad						71 dct_data_t col_outbuf[DCT_SIZE][DCT_SIZE], col_ir
21	dst									write	72 unsigned i, j;
22-38	Expressions										73
Performan	ce Resource										• III •

Figure 130: Memory resource src and Source Code

Figure 130 shows this read on the src variable is from the read operation inside loop DCT_Inner_Loop. This loop was automatically unrolled when DCT_Outer_Loop was pipelined and all operations in this loop can occur in parallel (if data dependencies allow).

The eight reads are being forced to occur over multiple cycles because the array src is implemented as a block RAM in the RTL and a block RAM can only allow two reads (maximum) in any one clock cycle. In **Figure 130**, the read operations take 2 clocks cycles: a cycle to generate the address for the block RAM and a cycle to read the data. Only the launch (address generation cycle) is shown because it overlaps with the operation in the next clock cycle.

You can optimize the block RAM accesses using optimization directives to partition the block RAM. The block RAM that function dct_1d accesses is defined as an input argument to the function and therefore resides outside this block.

- The input array to the first instance of dct_1d is buf_2d_in in function dct.
- The input array to the second instance of dct_1d is col_inbuf in function dct_2d.

In both cases, the arrays are 2-dimensional of size DCT_SIZE by DCT_SIZE (8x8). By default, this results in a single block RAM with 64 elements. Because the arrays are configured in the code in the form of Row by Column, we can partition the 2nd dimension and create eight separate Block RAMs: one for each row, allowing the row data to be accessed in parallel.

18. Click the **Synthesis** perspective button to return to the main synthesis view.

Step 7: Partition Block RAMs and Analyze Concurrency

- 1. Select the **New Solution** toolbar button or use the menu **Project > New Solution** to create a new solution.
- 2. Click **OK** and accept the defaults to create solution4.
- 3. Ensure the C source code is visible in the Information pane.
- 4. In the Directives tab:
 - a. In function dct, select array buf_2d_in.





- b. Right-click and select Insert Directive.
- c. In the **Directives Editor** dialog box, activate the **Directives** drop-down menu at the top and select **ARRAY_PARTITION**.
- d. Leave the type as **Complete**.
- e. Change the dimension setting to 2 to partition the array along the 2nd dimension.
- f. Click **OK**.
- 5. Repeat this process for array col_inbuf in function dct_2d.

The **Directive** pane displays optimization directives, as shown in **Figure 131** (the two new directives are highlighted).

E Outline 🖬 Directive 🛛	- 8
⊿ ● dct_1d	
×I1 dct_coeff_table	
⊿ ∰ DCT_Outer_Loop	
% HLS PIPELINE	
W DCT_Inner_Loop	
⊿ ● dct_2d	
×I1 row_outbuf	
×I1 col_outbuf	
×II col_inbuf	
% HLS ARRAY_PARTITION variable=col_inbuf complete dim=2	
Row_DCT_Loop	
✓ [#] Xpose_Row_Outer_Loop	
⊿ [#] ″ Xpose_Row_Inner_Loop	
% HLS PIPELINE	
Col_DCT_Loop	
⊿ [₩] Xpose_Col_Outer_Loop	
A 👯 Xpose_Col_Inner_Loop	
% HLS PIPELINE	
⊿ ● read_data	
⊿ [#] ″ RD_Loop_Row	
⊿ 🥙 RD_Loop_Col	
% HLS PIPELINE	
⊿ ● write_data	
⊿ [#] ″ WR_Loop_Row	
⊿ [∰] WR_Loop_Col	
% HLS PIPELINE	
⊿ ● dct	
×I1 buf_2d_in	,
% HLS ARRAY_PARTITION variable=buf_2d_in complete dim=2	J
×II buf_2d_out	
input	
output	

Figure 131: Optimization Directives for Array Partitioning



- 6. Click the Click the **Run C Synthesize** toolbar button to synthesize the design to RTL.
- 7. When synthesis completes, use the **Compare Reports** toolbar button to compare solutions 3 and 4.

Figure 132 shows the results of comparing solution3 and solution4. Improving access to the data in the src block RAM in the dct_1d block has improved the overall performance because the dct_1d block executes frequently.

eports	X 🗸							
e Estin	nates							*
ns)								
		soluti	on3	soluti	ion4			-
Target		8.00		8.00				
Estima	ted	5.90		5.90				Ξ
(clock d	ycles	5)						
	solu	ition3	solu	ition4]			
min	890		508					
max	890		508					
min	891		509					
max	891		509					Ļ
	e Estin ns) Target Estima (clock of min max min	Target Estimated (clock cycles (clock solu min 890 max 890 min 891	ee Estimates ns) Target 8.00 Estimated 5.90 (clock cycles) (clock solution3 min 890 max 890 min 891	se Estimates ns) Target 8.00 Estimated 5.90 (clock cycles) solution3 solution3 solution3 min 890 508 min 891	solution3 solution3 Target 8.00 8.00 Estimated 5.90 5.90 (clock cycles) solution3 solution4 min 890 508 max 891 509	se Estimatesns)solution3solution3Target8.00Estimated5.90(clock cycles)solution3solution4min890508min891509	se Estimatesns)solution3solution4Target 8.00 8.00 Estimated 5.90 5.90 (clock cycles)solution3solution4min 890 508 max 891 509	se Estimatesns)solution3solution4Target 8.00 8.00 Estimated 5.90 5.90 (clock cycles)solution3solution4min 890 508 max 890 508 min 891 509

Figure 132: DCT Solution3 and Solution4 Comparison

You can review the impact of the partitioning directive on the device resource.

- 8. Click the **Analysis** perspective button to begin interactive design analysis.
- 9. In the **Module Hierarchy**, ensure module dct is selected.
- 10. Select the **Resource Profile** in the lower-left by selecting the **Resource Profile** tab.
- 11. Expand the **Memories and Expressions** see the view in Figure 133.

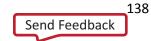


	BRAM	DSP	FF	LUT	Late	ency	Inte	erval	Pip	eline typ	e	
• dct	27	8	517	591	508		509	509		none		
dct_2d	18	8	457	437	373		373	373 none				
read_data	0	0	27	58	66		66		noi	ne		
Performance Pro	file 匡 R	lesour	e Pro	ofile 8	3							
		BR	AÂ	D	SP F	F	LUT	Bits	PO	Bits P1	Bits P2	Banks/Depth
dct		27		8	5	17	591					
I/O Ports(2	2)							32				
Instances		18		8	4	84	495					
🔺 🎟 Memories	(9)	9			0)	0	144				9
buf_2d		1			0		0	16				1
♦ buf_2d_		1			0	0		16				1
buf_2d_		1			0)	0	16				1
♦ buf_2d_		1			0		0	16				1
buf_2d_		1			0		0	16				1
buf_2d_		1			0		0	16				1
buf_2d_		1			0		0	16				1
buf_2d_		1			0)	0	16				1
♦ buf_2d_		1			0		0	16				1
Expression Expression	is(9)	0		0	0		47	42		35	8	
Þ ● +		0		0	0		29	29		17	0	
icmp		0		0	0		10	11		13	0	
Select		0		0	0		8	2		5	8	
IIII Registers()	11)					3	_	33				-
# FIFO(0)		0			0		0 0					0
Multiplexe	ers(13)	0			0)	49	49				0

Figure 133 DCT Resource Profile

The Resource Profile shows the resources being using at the current level of hierarchy (the block selected in the Module Hierarchy pane). **Figure 133** shows:

- This block has two I/O ports.
- Most of the area is due to instances (sub-blocks) within this block.
- There are nine memories, eight of which are the partitioned buf_2d_in block RAM.
- Most of the logic (expressions) at this level of hierarchy is due to adders, with some due to comparators and selectors.





The important point from the previous optimization is that you can see there are now additional memories due to the array partitioning optimization.

You still have a goal to ensure that the design can accept a new set of samples every 100 clock cycles. **Figure 132**, however, shows that you can only accept new data every 509 clocks. This is much better than the original, pre-optimized design (approx. 3700 clock cycles), but further optimization is required.

Up to this point, you have focused on improving the latency and interval of each of the individual loops and functions in the design. You must now apply the *dataflow optimization*, which enables the individual loops and functions to execute in parallel, thus improving the overall design interval.

12. Click the **Synthesis** perspective button to return to the main synthesis view.

Step 8: Partition Block RAMs and Apply Dataflow optimization

- 1. Select the **New Solution** toolbar button or use the menu **Project > New Solution** to create a new solution.
- 2. Click **OK** and accept the defaults to create solution5.
- 3. Ensure the C source code is visible in the Information pane.
- 4. In the **Directives** tab
 - a. Select the top-level function dct.
 - b. Right-click and select Insert Directive.
 - c. In the **Directives Editor** dialog box activate the **Directives** drop-down menu and select **DATAFLOW**.
 - d. Click OK.
- 5. Repeat this process for array col_inbuf in function dct_2d.

The Directive pane now displays the following optimization directives (the new directive is highlighted).



139

E XILINX.

🗄 Outline 🔟 Directive 😣 👘 🗖
▲ ● dct_1d
×II dct_coeff_table
✓ [#] DCT_Outer_Loop
% HLS PIPELINE
DCT_Inner_Loop
▲ ● dct_2d
×II row_outbuf
×II col_outbuf
×II col_inbuf
% HLS ARRAY_PARTITION partition variable=col_inbuf complete dim=2
Row_DCT_Loop
Xpose_Row_Outer_Loop
Xpose_Row_Inner_Loop
% HLS PIPELINE
Col_DCT_Loop
Xpose_Col_Outer_Loop
Xpose_Col_Inner_Loop
% HLS PIPELINE
read_data
RD_Loop_Row
A 👯 RD_Loop_Col
% HLS PIPELINE
write_data
✓ ₩ WR_Loop_Row
⊿ 🖞 WR_Loop_Col
% HLS PIPELINE
▲ ● dct
% HLS DATAFLOW
×I1 buf_2d_in
% HLS ARRAY_PARTITION partition variable=buf_2d_in complete dim=2
×I1 buf_2d_out
input
output

Figure 134: Dataflow Optimization for the DCT design

- 6. Click the Click the **Run C Synthesize** toolbar button to synthesize the design to RTL.
- 7. When synthesis completes, use the **Compare Reports** toolbar button or the menu **Project** > **Compare Reports** to compare solutions 4 and 5.

Figure 135 shows the results of comparing solution4 and solution5, and you can see the interval has improved. The design takes 539 clocks cycles to produce the outputs but can now accept new inputs every 405 clocks.



compare r	eports	x	_				
erforman	ce Estin	nates					
Timing	(ns)						
Clock			solution5		soluti	on4	ſ
default	Target		8.00		8.00		
	Estimated		5.90		5.90		
Latency	(clock	cycles	5)				
		solu	ition5	solu	ition4		l
Latency	min	507		508			
	max	507		508			
Interval	min	374		509			
	max	374		509			

Figure 135: DCT Solution4 and Solution5 Comparison

This is still greater than the 100 cycles required, so you must analyze the current performance.

- 8. Click the Analysis perspective button to begin interactive design analysis.
- 9. In the **Module Hierarchy**, you can see dct_2d accounts for most of the interval. Ensure module dct_2d is selected to see the view in Figure 136.

	BRAM	DSP	FF	LUT	Latency	Interval	Pipeline type	2		
dct	36	8	528	569	507	374	dataflow			
read_data	0	0	28	60	66	66	none			
• dct_2d	18	8	458	439	373	373	none			
dct_1d	8	8	386	115	13	13	none			
			2.1	60	66	66	2020			
write_data	NC	0	31	68	66 file	00	none			
 write_data erformance Prov 	NC							Initiation Interval	Itoration Latoncy	
erformance Pro	NC					Pipelin	ned Latency		Iteration Latency	Trip cou
	file 🛛							Initiation Interval 373		
erformance Pro	file 🛛	E Re	sourc	e Prot	file	Pipelin - no	ned Latency 373	373	-	Trip cou -
erformance Pro dct_2d Row_DCT_	file ¤ Loop v_Outer_	E Re	sourc	e Prot	file	Pipelin - no	ned Latency 373 120	373 -	- 15	Trip cou - 8

Figure 136: DCT Analysis View after Dataflow Optimization

Here, you can see two things:





- The interval of the dct block is less than the sum of the individual latencies (for read_data, dct_2d and write_data). This means the blocks are operating in parallel.
- The interval of dct is the same as the interval for sub-block dct_2d. The dct_2d block is therefore the limiting factor.

Because the dct_2d block is selected in the Module Hierarchy, the Performance Profile shows the details for this block. **Figure 136** shows the interval is the same as the latency, so none of these blocks operate in parallel.

One way to have the blocks in dct_2b operate in parallel would be to pipeline the entire function. This, however, would unroll all the loops, which can sometimes lead to a large area increase. An alternative is use dataflow optimization on function dcs_2b.

Another alternative is to use a less obvious technique: raise these loops up to the top-level of hierarchy, where they will be included in the dataflow optimization already applied to the top-level. This can be achieved by using an optimization directive to remove the dct_2d hierarchy: inline the dct_2d function.

Before performing this optimization, review the area increase caused by using dataflow optimization.

- 10. In the Module Hierarchy, ensure module dct is selected.
- 11. Activate the **Resource Profile** view.
- 12. Expand the memories to see the view in Figure 137.

M DS 8 8 8	528	LUT 569 567 0	Bits P0 32 144	Bits P1	Bits P2	
	517 0 0	567 0	144			
8	0	0	144			
8	0	0				
	0	-				
	-	0				18
	0	-	16			2
	0	0	16			2
	0	0	16			2
	0	0	16			2
	0	0	16			2
	0	0	16			2
	0	0	16			2
	0	0	16			2
	0	0	16			2
0	0	2	1	1	0	
	11		11			
	0	0	0			0
	0	0	0			0
	0	0 0 0 0 0 0 0 0 0 0 0 0 0 11	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 11 11 0 0 0 0	0 0 16 0 0 16 0 0 16 0 0 16 0 0 16 0 0 16 0 0 16 0 0 16 0 0 16 0 0 16 0 0 16 0 0 16 0 0 16 0 0 16 11 11 11	0 0 16 0 0 16 0 0 16 0 0 16 0 0 16 0 0 16 0 0 16 0 0 16 0 0 16 0 0 16 0 0 16 0 0 16 0 0 16 11 11 11 0 0 0	0 0 16

Figure 137: DCT Resource Profile



As compared with **Figure 133**, you can see there are now twice as many memories at this level of hierarchy (18 vs. 9). Each memory has been transformed into a Ping-Pong buffer to support dataflow. In this case, no "new" memories were added; the existing memories were converted into dataflow Ping-Pong memory channels. This doubled the number of block RAMs.

13. Click the **Synthesis** perspective button to return to the main synthesis view.

Step 9: Optimize the Hierarchy for Dataflow

- 1. Select the **New Solution** toolbar button to create a new solution.
- 2. Click **OK** and accept the defaults to create solution6.
- 3. Ensure the C source code is visible in the Information pane.
- 4. In the **Directives** tab:
 - a. Select function dct_2d.
 - b. Right-click and select Insert Directive .
 - c. In the **Directives Editor** dialog box activate the **Directives** drop-down menu at the top and select **INLINE**.
 - d. Click OK.

The Directive pane now shows the following optimization directives (the new directive is highlighted).



E XILINX.

🗄 Outline 🔯 Directive 🛛 🖓 🗖
▲ ● dct_1d
×II dct_coeff_table
⊿ ₩ DCT_Outer_Loop
% HLS PIPELINE
DCT_Inner_Loop
▲ ● dct_2d
% HLS INLINE
×II row_outbuf
×II col_outbuf
×II col_inbuf
% HLS ARRAY_PARTITION partition variable=col_inbuf complete dim=2
W Row_DCT_Loop
⊿ [₩] Xpose_Row_Outer_Loop
Xpose_Row_Inner_Loop
% HLS PIPELINE
Col_DCT_Loop
⊿ [₩] Xpose_Col_Outer_Loop
Xpose_Col_Inner_Loop
% HLS PIPELINE
✓ ● read_data
⊿ ₩ RD_Loop_Row
W RD_Loop_Col
% HLS PIPELINE
🔺 🔍 write_data
⊿ [₩] WR_Loop_Row
⊿ ¥″ WR_Loop_Col
% HLS PIPELINE
⊿ ● dct
% HLS DATAFLOW
×II buf_2d_in
% HLS ARRAY_PARTITION partition variable=buf_2d_in complete dim=2
×[] buf_2d_out
input
output
1

Figure 138: Dataflow Optimization for the DCT design

- 5. Click the **Run C Synthesize** toolbar button to synthesizes the design to RTL.
- 6. When synthesis completes, use the **Compare Reports** toolbar button or the menu **Project** > **Compare Reports** to compare solutions 5 and 6.

Figure 139 shows the results of comparing solution5 and solution6. You can see the interval has improved substantially.



compare r	eports	x)	-					° [
Performan	ce Estin	nates						1
🗆 Timing ((ns)							
Clock			soluti	on5	soluti	on6		ſ
default	Target	t	8.00		8.00			
	Estima	ated	5.90		5.90			=
Latency	(clock	cycles	5)					
		solu	ition5	solu	ition6			
Latency	min	507		407				
	max	507	507					
Interval	min	374	374					
	max	374		70				

Figure 139: DCT Solution5 and Solution6 Comparison

The interval is now below the 100 clock target. This design can accept a new set of input data every 70 clock cycles.

You can also see the details (1) in the synthesis report, which opens automatically after synthesis completes and (2) in the Analysis perspective, as shown in **Figure 140**.

	BRAM	DSP	FF	LUT	Latency	Interval	Pipeline type
• dct	54	16	914	580	407	70	dataflow
read_data	0	0	28	60	66	66	none
dct_Loop_Row_DCT_Loop_proc	8	8	388	158	69	69	none
dct_Loop_Xpose_Row_Outer_Loop_proc	0	0	28	62	66	66	none
dct_Loop_Col_DCT_Loop_proc	8	8	388	158	69	69	none
dct_Loop_Xpose_Col_Outer_Loop_proc	0	0	29	70	66	66	none
write_data	0	0	31	68	66	66	none

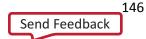
Figure 140: DCT Solution6 Module Hierarchy



Conclusion

In this tutorial, you learned:

- How to analyze a design using the analysis perspective.
- How to cross-link operations in the views with the C code.
- How to apply and judge optimizations.
- A methodology for taking the initial design results and creating an implementation which satisfies the design goals.





Chapter 7 Design Optimization

Overview

A crucial part of creating high quality RTL designs using High-Level Synthesis is having the ability to apply optimizations to the C code. High-Level Synthesis always tries to minimize the latency of loops and functions. To achieve this, within the loops and functions, it tries to execute as many operations as possible in parallel. At the level of functions, High-Level Synthesis always tries to execute functions in parallel.

In addition to these automatic optimizations, directives are used to:

- Execute multiple tasks in parallel, for example, multiple executions of the same function or multiple iterations of the same loop. This is pipelining.
- Restructure the physical implementation of arrays (block RAMs), functions, loops and ports to improve the availability of data and help data flow through the design faster.
- Provide information on data dependencies, or lack of them, allowing more optimizations to be performed.

The final optimization technique is to modify the C source code to remove unintended dependencies in the code that may limit the performance of the hardware.

This tutorial consists of two lab exercises.. You perform the analysis in these lab exercises using the Analysis perspective. A prerequisite for this tutorial is completion of the **Design Analysis** tutorial.

Lab1

Contrast the uses of loop and function pipelining to create a design that can process one sample per clock. This lab includes examples that give you the opportunity to analyze the two most common causes for designs failing to meet performance requirements: loop dependencies and data flow limitations or bottlenecks.

Lab2

This lab shows how modifications to the code from Lab 1 can help overcome some performance limitations inherent, but unintended, in the code.



Tutorial Design Description

You can download the tutorial design file from the Xilinx Website. Refer to the information in **Obtaining the Tutorial Designs**.

For this tutorial you use the design files in the tutorial directory Vivado_HLS_Tutorial\Design_Optimization.

The sample design you use in the lab exercise is a matrix multiplier function. The design goal is to process a new sample every clock period and implement the interfaces as streaming data interfaces.

Lab 1: Optimizing a Matrix Multiplier

This exercise uses a matrix multiplier design to show how you can fully optimize a design heavily based on loops. The design goal is to read one sample per clock cycle using a FIFO interface, while minimizing the area.

The analysis includes a comparison of a methodology that optimizes at the loop level with one that optimizes at the function level.

IMPORTANT: The figures and commands in this tutorial assume the tutorial data directory *Vivado_HLS_Tutorial* is unzipped and placed in the location *C:\Vivado_HLS_Tutorial*.



If the tutorial data directory is unzipped to a different location, or on Linux systems, adjust the few pathnames referenced, to the location you have chosen to place the **Vivado_HLS_Tutorial** directory.

Step 1: Create and Open the Project

- 1. Open the Vivado HLS Command Prompt.
 - a. On Windows use Start > All Programs > Xilinx Design Tools > Vivado 2013.3 > Vivado HLS > Vivado HLS 2013.3 Command Prompt (Figure 141).
 - b. On Linux, open a new shell.



Figure 141: Vivado HLS Command Prompt



- 2. Using the command prompt window (Figure 142), change directory to the RTL Verification tutorial, lab1.
- 3. Execute the Tcl script to set up the Vivado HLS project, using the command vivado_hls –f run_hls.tcl, as shown in **Figure 142**.

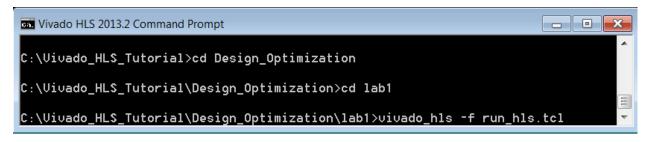


Figure 142: Setup the Design Optimization Tutorial Project

4. When Vivado HLS completes, open the project in the Vivado HLS GUI using the command vivado_hls –p matrixmul_prj, as shown in **Figure 143**.

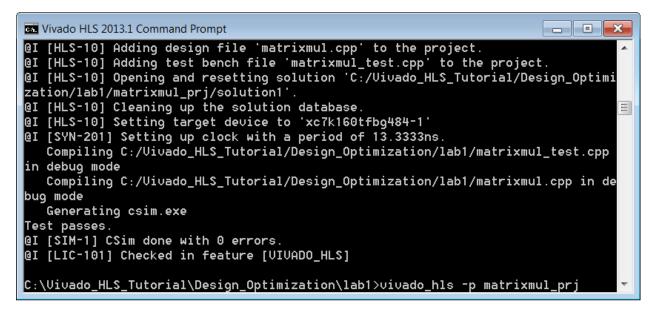
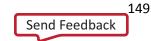


Figure 143: Open Design Optimization Project for Lab 1

5. Expand the Sources folder in the Explorer pane and double-click matrixmul.cpp to view the source code (Figure 144).

Scroll down the file to see that the source code has two input arrays, a and b, and output array res. Hold the mouse over the macros (as shown in **Figure 144**) to see that each is three-by-three for a total of nine elements.





🔁 Explorer 🛛 🛛 🤣 🤷 🗖 🗖	i matrixmul.cpp ⊠	
😂 matrixmul_prj	46 #include "matrixmul.h"	
🗊 Includes	47	
Source	48 void matrixmul(
matrixmul.cpp	49 mat_a_t a[MAT_A_ROWS][MAT_A_COLS], 50 mat b t b[MAT B ROWS][Macro Expansion	
🜆 Test Bench	50 mat_b_t b[MAT_B_ROWS][Macro Expansion 51 result_t res[MAT_A_ROW])	
🖕 solution1	521	
constraints	53 // Iterate over the rows or the A matrix	
🞾 directives.tcl	54 Row: for(int i = 0; i < MAT_A_ROWS; i++) {	=
🖋 script.tcl	55 // Iterate over the columns of the B matrix	=
🗁 csim	56 Col: for(int j = 0; j < MAT_B_COLS; j++) {	
i build	57 res[i][j] = 0;	-
i pana i	58 // Do the inner product of a row of A and col of B	•

Figure 144: Source Code for the Matrix Multiplier

Step 2: Synthesize and Analyze the Design

1. Click the Click the Run C Synthesize toolbar button to synthesize the design to RTL.

When synthesis completes, the synthesis report opens (Figure 145), and the Performance estimates appears:

- The interval is 80 clock cycles. Because there are nine elements in each input array, the design takes approximately nine cycles per input read.
- The interval is one cycle longer than the latency, so there is no parallelism in the hardware at this point.
- The latency/interval is due to nested loops.
 - The inner loop called Product:
 - Has a latency of 2 clock cycles
 - Has 6 clock cycles total for all iterations.
 - o The Col loop:
 - It requires 1 clock to enter loop Product and 1 clock to exit
 - It takes 8 clock cycles for each iteration (1+6+1)
 - Has 24 cycles for all iterations to complete.
 - The top-level loop has a latency of 26 clock cycles per iteration, for a total of 78 clock cycles for all iterations of the loop.





formand	e Estim	ates							
'iming (ns)								
Summ	ary								
Clock	Targ	et	Estimated	Und	ertainty				
default	13.	33	8.18		1.67				
Late		Inte	erval						
Summ	(clock cy ary								
min	max	min	max	Туре					
79	79	80	80	none					
Detail Inst	ance								
		l	atency			Initiation I	Interval		
Loo	p Name	mi	in max	Itera	tion Latency	achieved	target	Trip Count	Pipelined
- Ro	w	7	8 78		26	-	-	3	no
		2	4 24		8	-	-	3	no
+ C	01								

Figure 145: Synthesis Report for the Matrix Multiplier

You can do one of two things to improve the initiation interval: Pipeline the loops or pipeline the entire function. You begin by pipelining the loops and then compare those results to pipelining the entire function.

When pipelining loops, the initiation interval of the loops is the important metric to monitor. As seen in this exercise, even when the design reaches the stage at which the loop can process a sample every clock cycle, the initiation interval of the function is still reported as the time it takes for the loops contained within the function to finish processing all data for the function,

Step 3: Pipeline the Product Loop

- 1. Select the **New Solution** toolbar button or use the menu **Project > New Solution** to create a new solution.
- 2. Click **OK** and accept the defaults to create solution2.
- 3. Ensure the C source code is visible in the Information pane.

When pipelining nested loops, you realize the greatest benefit by pipelining the inner-most loop, which processes a sample of data. High-Level Synthesis automatically applies loop flattening, collapsing the nested loops, removing the loop transitions, allowing the outer loops simply to feed the inner loop with data.





- 4. In the **Directives** tab:
 - a. Select loop Product.
 - b. Right-click and select Insert Directive.
 - c. In the **Directives Editor** dialog box, activate the **Directives** drop-down menu at the top and select **PIPELINE**.
 - d. Click **OK**. With the default options, an initiation interval (II) of 1 (one new loop iteration per clock) will be the default.

The Directive pane should show the following optimization directives. (The new directive is highlighted.)

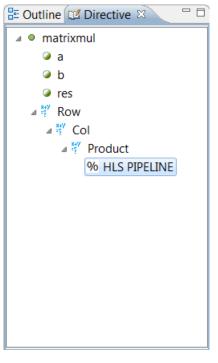


Figure 146: Initial Pipeline Directive

5. Click the Click the Run C Synthesize toolbar button to synthesize the design to RTL.

During synthesis, the information reported in the Console pane shows loop flattening was performed on loop Row and that the default initiation internal target of 1 could not be achieved on loop Product due to a dependency.

```
@I [XFORM-541] Flattening a loop nest 'Row' (matrixmul.cpp:54) in function
'matrixmul'.
...
@I [SCHED-61] Pipelining loop 'Product'.
@W [SCHED-68] Unable to enforce a carried dependency constraint (II = 1,
distance = 1) between 'store' operation (matrixmul.cpp:60) of variable
'tmp_3' on array 'res' and 'load' operation ('res_load', matrixmul.cpp:60)
on array 'res'.
@I [SCHED-61] Pipelining result: Target II: 1, Final II: 2, Depth: 3.
```

www.xilinx.com

152 Send Feedback



The synthesis report (Figure 147) shows that although the Product loop is pipelined with an interval of 2, the interval of top-level loop is not pipelined.

forman	ce Estin	nates							
Timing	(ns)								
🗆 Sumr	nary								
Clock	Tar	get	Estimated	Un	certainty				
defaul	t 13	.33	10.57	'	1.67				
atency	(clock c	ycles)							
Sumr									
	ency	Int	erval						
min	max	min	max	Туре					
82	82	83	83	none					
	tance								
E Lo	op		atency			Initiation	Interval		
Loo	p Name			Itera	tion Latency	achieved	target	Trip Count	Pipelined
De	w_Col	8	1 81		9	-	-	9	no
- KC									

Figure 147: Matrixmul Initial Pipeline Report

The reason the top-level loop is not pipelined is that loop flattening only occurred on loop Row. There was no loop flattening of loop Col into the Product loop. To understand why loop flattening was unable to flatten all nested loops, use the Analysis perspective.

- 6. Open the Analysis perspective.
- 7. In the Performance View, expand loops Row_Col and Product.
- 8. Select the write operation in state C1.
- 9. Right-click and select **Goto Source** to see the view in Figure 148.

The write operation in state C1 is due to the code that sets res to zero before the Product loop. Because res is a top-level function argument, it is a write to a port in the RTL: This operation must happen before the operations in loop Product are executed. Because it is not an internal operation but has an impact on the I/O behavior, this operation cannot be moved or optimized. This prevents the Product loop from being flattened into the Row_Col loop.





😑 Pe	erformance - matrixmul 🛛	~			- 0	C Source 🛛 🖓 🗖
Cı	urrent Module : matri	xmul				File: C:\Vivado_HLS_Tutorial\Design_Optimization\lab1\matrixmul.c
	Operation\Control S	C0	C1	C2	C3	41
1	∃Row Col					42 THIS COPYRIGHT NOTICE AND DISCLAIMER MUST E
2	exitcond flatt					43 ALL TIMES.
3	indvar flatten					44
4	exitcond1(icmp)					45 ************************************
5	j mid2(select)					46 #include "matrixmul.h"
6	i s(+)					47
7	i mid2(select)					48 void matrixmul(
8	p addr7(-)					49 mat a ta[MAT A ROWS][MAT A COLS],
9	p addr8(+)					50 mat b t b[MAT B ROWS][MAT B COLS],
10	node 35(write)					51 result t res[MAT A ROWS][MAT B COLS])
11	□Product					52{
12	exitcond(icmp)					53 // Iterate over the rows of the A matrix
13	k 1(+)					54 Row: for(int i = 0; i < MAT A ROWS; i++) {
14	p addr1(+)					55 // Iterate over the columns of the B matrix
15	a load(read)					56 Col: for(int j = 0; j < MAT B COLS; j++) {
16	p addr3(-)					57 res[i][i] = 0;
17	p addr4(+)					58 // Do the inner product of a row of A and col of B
18	b load(read)					59 Product: for(int k = 0; k < MAT B ROWS; k++) {
19	tmp 7(*)					60 res[i][i] += a[i][k] * b[k][i];
20	res load(read)					61 }
21	tmp 8(+)					62 }
22	node 68(write)					63 }
23	j 1(+)					64
						65 }
						66
						67 👻
Perf	ormance Resource					۰ III ا
- Citt	Annunce Resource					

Figure 148: Matrixmul Initial Performance View

More importantly, it is worth addressing why only an II of 2 was possible for the Product loop.

The message SCHED-68 tells you:

```
@W [SCHED-68] Unable to enforce a carried dependency constraint (II = 1,
distance = 1) between 'store' operation (matrixmul.cpp:60) of variable
'tmp_3' on array 'res' and 'load' operation ('res_load', matrixmul.cpp:60)
on array 'res'.
```

- The issue is a carried dependency. This is a dependency between an operation in one iteration of a loop and an operation in a different iteration of the same loop. For example, an operation when k=1 and when k=2 (where k is the loop index).
- The first operation is a store (memory read operation) on array res on line 60.
- The second operation is a load (memory write operation) on array res on line 60.

From **Figure 148** you can see line 60 is a read from array res (due to the += operator) and a write to array res. An array is mapped into a block RAM by default and the details in the Performance View can show why this conflict occurred.

The Performance view shows in which states the operations are scheduled. Figure 149 shows a number of copies of the schedule for the Product loop to highlight how this issue can be understood. Start with the basic view shown in the top-right. The operations on the res array, a two-cycle read and write, are highlighted in red.





In the successful schedule, the next iteration of the Product loop appears as shown below. In this schedule, the initiation interval (II)=2 and the loop operations re-start every two cycles. There is no conflict between any block RAM accesses. (None of the red highlights overlap across iterations.)

The unsuccessful schedule shows why the loop cannot be pipelined with an II=1. In this case, the next iteration would need to start after 1 clock cycle. The write to the block RAM in the first iteration is still occurring when the second iteration tries to apply an address for a read operation. These addresses are different. Both cannot be applied to the block RAM at the same time.

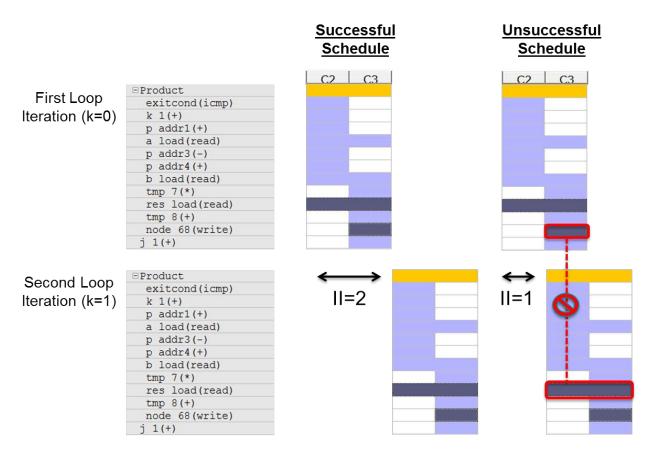
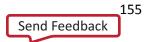


Figure 149: Carried Dependency Analysis

You cannot pipeline the Product loop with an initiation interval of 1. The next lab exercise shows how writing the code can remove this limitation (any technique that does not write back to the same block RAM). In this lab exercise you optimize the code as it is.

The next step is to pipeline the loop above, the Col loop. This automatically unrolls the Product loop and creates more operators and hence more hardware resources, but it ensures there is no dependency between different iterations of the Product loop.

10. Return to the Synthesis perspective.





Step 4: Pipeline the Col Loop

- 1. Select the **New Solution** toolbar button to create a new solution.
- 2. Because solution2 already has a directive added, use the drop-down menu to **select solution1** as the source for existing directives and constraints (solution1 has none).
- 3. Click **Finish** and accept the default solution name, solution3.
- 4. Open the C source code matrixmul.cpp to make it visible in the Information pane.
- 5. In the **Directives** tab:
 - a. Select loop Col.
 - b. Right-click and select Insert Directive
 - c. In the **Directives Editor** dialog box activate the **Directives** drop-down menu at the top and select **PIPELINE**.
 - d. Click **OK**. With the default options, an initiation interval (II) of 1 (one new loop iteration per clock) becomes the default.

The Directive pane, shown below, displays the following optimization directives (the new directive is highlighted).

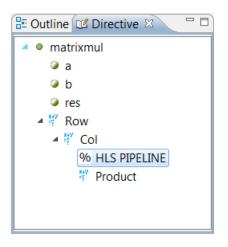


Figure 150: Col Pipeline Directive

6. Click the Click the Run C Synthesize toolbar button to synthesize the design to RTL.

During synthesis, the information reported in the Console pane shows that loop Product was unrolled, loop flattening was performed on loop Row, and the default initiation internal target of 1 could not be achieved on loop Row_Col due resource limitations on the memory for array a.

```
@I [XFORM-502] Unrolling all sub-loops inside loop 'Col'
(matrixmul.cpp:56) in function 'matrixmul' for pipelining.
@I [XFORM-501] Unrolling loop 'Product' (matrixmul.cpp:59) in function
'matrixmul' completely.
@I [XFORM-541] Flattening a loop nest 'Row' (matrixmul.cpp:54) in function
'matrixmul'.
```





. . .

```
...
@I [SCHED-61] Pipelining loop 'Row_Col'.
@W [SCHED-69] Unable to schedule 'load' operation ('a_load_1',
matrixmul.cpp:60) on array 'a' due to limited memory ports.
@I [SCHED-61] Pipelining result: Target II: 1, Final II: 2, Depth: 4.
```

Reviewing the synthesis report shows, as noted above, that the interval for loop Row_Col is only two: the target is to process one sample every cycle. Once again, you can use the Analysis perspective to highlight why the initiation target was not achieved.

- 7. Open the Analysis perspective.
- 8. In the Performance View, expand the Row_Col loop

The operations on array a (mentioned in the SCHED-69 message above) are highlighted in **Figure 151**. There are three read operations on array a. Two operations start in state C1 and a third read operation starts in state C2.

Arrays are implemented as block RAMs and arrays which are arguments to the function are implemented as block RAM ports. In both cases a block RAM can only have a maximum of two ports (for dual-port block RAM). By accessing array a through a single block RAM interface, there are not enough ports to be able to read all three values in one clock cycle.





冒 Pe	rformance - matrixmul 🛛	_			- 8
Cu	rrent Module : matri	xmul			
	Operation\Control S	C0	C1	C2	C3
-	BRow Col				_
2	exitcond flatt				
3	indvar flatten				
4	exitcond(icmp)				
5	j mid2(select)				
6	i s(+)				
7	i mid2(select)				
8	p addr(-)				
9	p addr2(+)			-	
10	a load(read)				
11	b load(read)				
12	tmp 7(*)				
13	p addr4(+)				
14	a load 1(read)				
15	p addr3(+)				
16	b load 1(read)			1	
17	tmp 7 1(*)				
18	p addr1(+)				
19	a load 2(read)				
20	p addr9(+)				
21	b load 2(read)				
22	tmp 7 2(*)				
23	tmp1(+)				
24	tmp 8 2(+)				
25	node 74(write)				
26	j 1(+)				
Perfo	rmance Resource				

Figure 151: Matrixmul Pipeline Col Performance View

Another way to view this resource limitation is to use to the Resource pane.

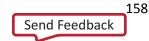
9. Click the **Resource Sharing** tab.

10. Expand the memories to see the view shown in Figure 152.

In **Figure 152** the 2-cycle read operations in state C1 overlap with those starting in state C2 and so only a single cycle is visible: however, it is clear that this resource is used in multiple states.

In looking at this view, it is clear that even when the issue with port a is resolved, the same issue occurs with port b: it also has to perform 3 reads.

High-Level Synthesis can only report one schedule error or warning at a time, because, as soon as the first issue occurs, the actions to create an achievable schedule invalidates any other infeasible schedules.





😑 Resoul	Resource - matrixmul 🖾 🛛 🖓 🖓											
Curre	Current Module : matrixmul											
	Resource\Control Step	C0	C1	C2	C3							
1-4	∃I/O Ports											
5	⊡Memory Ports											
6	b		read	read read								
7	a		read	re	ad							
8	a		re	ad								
9	b		re	ad								
10	res				write							
11-29 ■Expressions												
Performa	Performance Resource											

Figure 152: Matrixmul Pipeline Col Resource Sharing View

High-Level Synthesis allows arrays to be partitioned, mapped together and re-shaped. These techniques allow the access to array to be modified without changing the source code.

11. Return to the Synthesis perspective.

Step 5: Reshape the Arrays

- 1. Select the **New Solution** toolbar button or use the menu **Project > New Solution** to create a new solution.
- 2. Click **Finish** and accept the default solution name solution4.

Because the loop index for the Product loop is k, both arrays should be partitioned along their respective k dimension: the design needs to access more than two values of k in each clock cycle.

For array a, this is dimension 2 because its access patterns is a[i][k]; for array b, this is dimension 1 because its access pattern is b[k][j].

Partitioning these arrays creates k arrays - in this case, k number ports. Alternatively, we can use re-shape instead of partition allowing one wide array (port) to be created instead of k ports.

After this transformation, the data in the block RAM outside this block must be reshaped in an identical manner: if this process is not done by HLS, the data must be arranged as:

- For array a: i elements, each of width data_word_size times k.
- For array b: j elements, each of width data_word_size times k.
- 3. Open the C source code matrixmul.cpp to make it visible in the Information pane.
- 4. In the **Directives** tab
 - a. Select variable a.
 - b. Right-click and select Insert Directive.





- c. In the **Directives Editor** dialog box activate the **Directives** drop-down menu at the top and select **ARRAY_RESHAPE**.
- d. Set the dimension to **2**.
- e. Click **OK**.
- 5. Repeat this process for variable **b**.

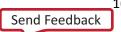
The Directive pane should show the following optimization directives (the new directive is highlighted).

E Outline 📴 Directive 🛛	- 8
▲ ● matrixmul	
a	
% HLS ARRAY_RESHAPE reshape variable=a complete dim=2	2
b	
% HLS ARRAY_RESHAPE reshape variable=b complete dim=1	
res	
✓ [#] [*] Row	
Col	
% HLS PIPELINE	
Product	

Figure 153: Array Reshape Directive

6. Click the **Run C Synthesize** toolbar button to synthesize the design to RTL.

The synthesis report shows the top-level loop Row_Col is now processing data at 1 sample per clock period (Figure 154).



160



	l_csynth								
ormar	ice Estim	nates							
ming	(ns)								
Sum	mary								
Clock	c Tar	get	Estimated	l Unc	ertainty				
defau	lt 13	.33	11.13	3	1.67				
	ency		erval						
Lat	encv	Inte	erval						
min	max	min	max	Туре					
12	12	13	13	none					
Deta	il								
🗉 In:	stance								
🗆 Lo	00								
	44		atency			Initiation	Interval		
	n Nama			Itorati	ion Latonau			Trip Count	Dipolipod
LOC	op Name ow_Col	10		Iterati	ion Latency 3	achieved 1	target 1	Trip Count 9	Pipelined
_									yes

Figure 154: Optimized Loop Processing report

- The top-level module takes 12 clock cycles to complete.
- The Row_Col loop outputs a sample after 3 cycles (iteration latency).
- It then reads 1 sample every cycle (Initiation Interval).
- After 9 iterations/samples (Trip count) it completes all samples.
- 3 + 9 = 12 clock cycles

The function can then complete and return to start to process the next set of data.

Now, change the block RAM interfaces to FIFO interfaces to allow for streaming data.

Step 6: Apply FIFO Interfaces

- 1. Select the **New Solution** toolbar button to create a new solution.
- 2. Click Finish and accept the default solution name, solution5.
- 3. Open the C source code matrixmul.cpp to make it visible in the Information pane.
- 4. In the **Directives** tab
 - a. Select variable a.
 - b. Right-click and select Insert Directive.





- c. In the **Directives Editor** dialog box activate the **Directives** drop-down menu at the top and select **INTERFACE**.
- d. Click the **mode** drop-down menu to select ap_fifo.
- e. Click **OK**.
- 5. Repeat this process for variable b and variable res..

The Directive pane displays the following optimization directives. (The new directives are highlighted).

=2
=1

Figure 155: Matrixmul FIFO Directives

6. Click the Run C Synthesize toolbar button to synthesizes the design to RTL.

Figure 156 shows the Console display after synthesis runs.

💷 Console 🕴 🧕 Errors 💩 Warnings 🛛 👫 🐻 🖆	
Vivado HLS Console	
@I [HLS-10] Opening project 'C:/Vivado_HLS_Tutorial/Design_Optimization/lab1/matrixmul_prj'.	
@I [HLS-10] Adding design file 'matrixmul.cpp' to the project.	
@I [HLS-10] Adding test bench file 'matrixmul_test.cpp' to the project.	
@I [HLS-10] Opening solution 'C:/Vivado_HLS_Tutorial/Design_Optimization/lab1/matrixmul_prj/solution	15'
@I [SYN-201] Setting up clock with a period of 13.3333ns.	
<pre>@I [HLS-10] Setting target device to 'xc7k160tfbg484-1'</pre>	
<pre>@I [HLS-10] Importing test bench file 'matrixmul_test.cpp'</pre>	
@I [HLS-10] Analyzing design file 'matrixmul.cpp'	
@I [HLS-10] Validating synthesis directives	
@I [HLS-10] Checking synthesizability	
@E [SYNCHK-91] Port 'res' (matrixmul.cpp:51) of function 'matrixmul' cannot be set to a FIFO as it h	nas 🔄
@I [SYNCHK-10] 1 error(s), 0 warning(s).	=
	-
✓	•

Figure 156: FIFO Synthesis Warning

From the code shown in **Figure 157**, array res performs writes in the following sequence (MAT_B_COLS = MAT_B_ROWS = 3):





- Write to [0][0] on line57.
- Then a write to [0][0] on line 60.
- Then a write to [0][0] on line 60.
- Then a write to [0][0] on line 60.
- Write to [0][1] on line 57 (after index J increments).
- Then a write to [0][1] on line 60.
- Etc.

Four consecutive writes to address [0][0] does not constitute a streaming access pattern; this is random access.

) m	atrix	mul.c	pp 🛛	- 8
52	{			
53	1	/ Ite	rate over the rows of the A matrix	
54	1	Row:	<pre>for(int i = 0; i < MAT_A_ROWS; i++) {</pre>	
55		- 11	'Iterate over the columns of the B matrix	
56		Co	ol: for(int j = 0; j < MAT_B_COLS; j++) {	
57			res[i][j] = 0;	
58			<pre>// Do the inner product of a row of A and col of B</pre>	
59			Product: for(int k = 0; k < MAT_B_ROWS; k++) {	
60			res[i][j] += a[i][k] * b[k][j];	
61			}	
62		}		
63		}		=
64				
65	}			-
	•		III	P

Figure 157: Matrixmul Code

Examining the code in **Figure 157** reveals that there are similar issues reading arrays a and b. It is impossible to use a FIFO interface for data access with the code as written. To use a FIFO interface, the optimization directives available in Vivado High-Level Synthesis are inadequate because the code currently enforces a certain order of reads and writes. Further optimization requires a re-write of the code, which you accomplish in Lab 2.

Before modifying the code, however, it is worth pipelining the function instead of the loops to contrast the difference in the two approaches.

Step 7: Pipeline the Function

1. Select the **New Solution** toolbar button to create a new solution.



IMPORTANT: In this step, copy the directives from solution4 as this solution <u>does not</u> have FIFO interfaces specified.





2. Select **solution4** from both the drop down menus in the **Options** section. The Solution Wizard appears as shown in **Figure 158**.

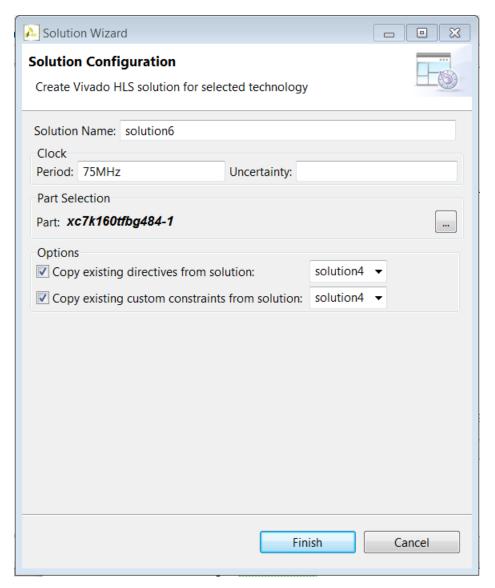
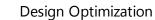


Figure 158: New Solution Based on Solution4 Directives

- 3. Click Finish and accept the default solution name, solution6.
- 4. Open the C source code matrixmul.cpp to make it visible in the Information pane.
- 5. In the **Directives** tab:
 - a. Select the pipeline directive on loop Col.
 - b. Right-click and select Remove Directive.
 - c. Select the top-level function matrixmul.
 - d. Right-click and select Insert Directive.







- e. In the **Directives Editor** dialog box activate the **Directives** drop-down menu at the top and select **PIPELINE**.
- f. Click **OK**.

The Directives tab should appear as Figure 159.

🗄 Outline 🖾 Directive 🖾 👘 🖓
 matrixmul HLS PIPELINE a HLS ARRAY_RESHAPE reshape variable=a complete dim=2 b HLS ARRAY_RESHAPE reshape variable=b complete dim=1 res #" Row #" Col

Figure 159: Directives for Solution6

- 6. Click the **Run C Synthesize** toolbar button to synthesize the design to RTL.
- 7. Click the **Compare Reports** toolbar button.
 - a. Add solution4.
 - b. Add solution6.
 - c. Click **OK**.

The comparison of solutions 4 and 6 is shown in Figure 160.



compare r	eport	5 🖾						-	E
Performan	ce Est	imates							1
Timing	(ns)								
Clock			solut	tion6	soluti	ion4			
default	Targ	et	13.3	3	13.33				
	Estin	nated	11.13	3	11.13				
Latency	(cloc	c cycle	5)						
		solu	ition6	solu	solution4				
Latency	min	6		12					
	max	6		12	12				
Interval	min	5		13					-
	max	5		13	13				
Utilization	Estim	ates							
	s	olution	6 so	olution	4				
BRAM_18	BRAM_18K 0		0						
DSP48E	2	7	3						
FF 51		17	5	5					
LUT	4	4	3	7					

Figure 160: Loop versus Function Pipelining

The design now completes in fewer clocks and can start a new transaction every 5 clock cycles. However, the area and resources have increased substantially because all the loops in the design were unrolled.

@I [XFORM-502] Unrolling all loops for pipelining in function 'matrixmul'
(matrixmul.cpp:51).
@I [XFORM-501] Unrolling loop 'Row' (matrixmul.cpp:54) in function 'matrixmul'
completely.
@I [XFORM-501] Unrolling loop 'Col' (matrixmul.cpp:56) in function 'matrixmul'
completely.
@I [XFORM-501] Unrolling loop 'Product' (matrixmul.cpp:59) in function
'matrixmul' completely.

Pipelining loops allows the loops to remain rolled, thus providing a good means of controlling the area. When pipelining a function, all loops contained in the function are unrolled, which is a requirement for pipelining. The pipelined function design can process a new set of 9 samples every 5 clock cycles. This exceeds the requirement of 1 sample per second because the default behavior of High-Level Synthesis is to produce a design with the highest performance.

The pipelined function results in the best performance. However, if it exceeds the required performance, it might take multiple additional directives to slow the design down. Pipelining loops gives you an easy way to control resources, with the option of partially unrolling the design to meet performance.



Lab 2: C Code Optimized for I/O Accesses

In Lab 1, you were unable to use streaming interfaces. The nature of the C code, which specified multiple accesses to the same addresses, prevented streaming interfaces being applied.

- In a streaming interface, the values must be accessed in sequential order.
- In the code, the accesses were also port accesses, which High-Level Synthesis is unable to move around and optimize. The C code specified writing the value zero to port res at the start of every product loop. This may be part of the intended behavior. HLS cannot simply decide to change the specification of the algorithm.

The code intuitively captured the behavior of a matrix multiplication, but it prevented a required behavior in the hardware: streaming accesses.

This lab exercise uses an updated version of the C code you worked with in Lab 1. The following explains how the C code was updated.

Figure 161 shows the I/O access pattern for the code in Lab 1. Out of necessity the address values are shown is a small font.

As variables i, j and k iterate from 0 to 3, the lower part of Figure 161 shows the addresses generated to read a, b and write to res. In addition, at the start of each Product loop, res is set to the value zero.

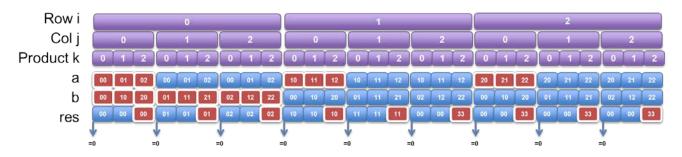


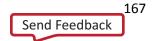
Figure 161: Lab 1 Matrix Multiplier Address Accesses

To have a hardware design with sequential streaming accesses, the ports accesses can only be those shown highlighted in red. For the read ports, the data must be cached internally to ensure the design does not have to re-read the port. For the write port res, the data must be saved into a temporary variable and only written to the port in the cycles shown in red.

The C code in this lab reflects this behavior.

Step 1: Create and Open the Project

- 1. From the Vivado HLS command prompt used in Lab 1, change to the lab2 directory as shown in **Figure 162**.
- 2. Create a new Vivado HLS project by typing vivado_hls -f run_hls.tcl.





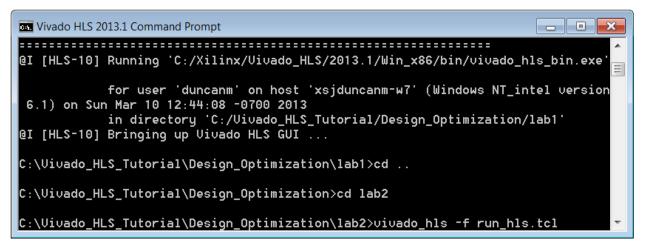


Figure 162: Setup for Interface Synthesis Lab 2

- 3. Open the Vivado HLS GUI project by typing vivado_hls -p matrixmul_prj.
- 4. Open the Source folder in the explorer pane and double-click matrixmul.cpp to open the code as shown in **Figure 163**.



```
🖻 matrixmul.cpp 🖾
52 {
                                                                          .
53 #pragma HLS ARRAY RESHAPE variable=b complete dim=1
54 #pragma HLS ARRAY RESHAPE variable=a complete dim=2
55 #pragma HLS INTERFACE ap_fifo port=a
56 #pragma HLS INTERFACE ap_fifo port=b
57 #pragma HLS INTERFACE ap fifo port=res
58 mat_a_t a_row[MAT_A_ROWS];
    mat b t b copy[MAT B ROWS][MAT B COLS];
59
60
    int tmp = 0;
61
62
    // Iterate over the rowa of the A matrix
     Row: for(int i = 0; i < MAT A ROWS; i++) {
63
64
      // Iterate over the columns of the B matrix
65
       Col: for(int j = 0; j < MAT_B_COLS; j++) {
66 #pragma HLS PIPELINE
         // Do the inner product of a row of A and col of B
67
68
         tmp=0;
         // Cache each row (so it's only read once per function)
69
70
         if (j == 0)
71
           Cache_Row: for(int k = 0; k < MAT_A_ROWS; k++)</pre>
72
             a_row[k] = a[i][k];
73
74
          // Cache all cols (so they are only read once per function)
75
        if (i == 0)
76
               Cache_Col: for(int k = 0; k < MAT_B_ROWS; k++)</pre>
77
                  b_copy[k][j] = b[k][j];
78
79
         Product: for(int k = 0; k < MAT B ROWS; k++) {</pre>
80
           tmp += a_row[k] * b_copy[k][j];
   .€
                                111
```

Figure 163: C Code with updated IO accesses

Review the code and confirm the following:

- The directives from Lab 1, including the FIFO interfaces, are specified in the code as pragmas.
- For-loops have been added to cache the rol and column reads.
- A temporary variable is used for the accumulation and port res is only written to when the final result is computed for each value.
- Because the for-loops to cache the row and column would require multiple cycles to perform the reads, the pipeline directive has been applied to the Col for-loop, ensuring these cache for-loops are automatically unrolled.

Synthesize the design and verify the RTL using co-simulation.

- 5. Click the Run C Synthesize toolbar button to synthesize the design to RTL.
- 6. When synthesis completes, use the **Run C/RTL Cosimulation** toolbar button to launch the **Cosimulation Dialog** box.





7. Click **OK** to start RTL verification.

The design has been now been fully synthesized to read one sample every clock cycle using streaming FIFO interfaces.

Conclusion

In this tutorial, you:

- Learned how to analyze pipelined loops and understand exactly which limitations prevent optimizations targets from being achieved.
- The advantages and disadvantages of function versus loop pipelining.
- How unintended dependencies in the code can prevent hardware design goals from being realized and how they can be overcome by modifications to the source code.





Chapter 8 RTL Verification

Overview

The High Level Synthesis tool automates the process of RTL verification and allows you to use RTL verification to generate trace files that show the activity of the waveforms in the RTL design. You can use these waveforms to analyze and understand the RTL output. This tutorial covers all aspects of the RTL verification process.

To perform RTL verification, you use both the RTL output from High-Level Synthesis (Verilog, VHDL or SystemC) and the C test bench. RTL verification is often called "cosimulation" or "C/RTL cosimulation"; because both C and RTL are used in the verification.

This tutorial consists of three lab exercises.

Lab1

Perform RTL verification steps and understand the importance of the C test bench in verifying the RTL.

Lab2

Create RTL trace files and analyze them using the Vivado Design Suite.

Lab3

Create RTL trace files and analyze them using a third-party RTL simulator. This lab requires a license for Mentor Graphics ModelSim simulator. (You can use an alternative, third-party simulator with minor modifications to the steps).

Tutorial Design Description

You can download the tutorial design file from the Xilinx website. Refer to the information in





Obtaining the Tutorial Designs.

This tutorial uses the design files in the tutorial directory **Vivado_HLS_Tutorial\RTL_Verification.**

The sample design used in the lab exercise is a DUC (digital up converter) function. The purpose of this lab is to demonstrate and explain the features of RTL verification. There are no design goals for these lab exercises.

Lab 1: RTL Verification and the C test bench

This exercise explains the basic operations for RTL verification and highlights the importance of the C test bench.

IMPORTANT: The figures and commands in this tutorial assume the tutorial data directory *Vivado_HLS_Tutorial* is unzipped and placed in the location *C:\Vivado_HLS_Tutorial*.



If the tutorial data directory is unzipped to a different location, or on Linux systems, adjust the few pathnames referenced, to the location you have chosen to place the **Vivado_HLS_Tutorial** directory.

Step 1: Create and Open the Project

- 1. Open the Vivado HLS Command Prompt.
 - a. On Windows use Start > All Programs > Xilinx Design Tools > Vivado 2013.3 > Vivado HLS > Vivado HLS 2013.3 Command Prompt (Figure 164).
 - b. On Linux, open a new shell.



Figure 164: Vivado HLS Command Prompt

- 2. Using the command prompt window (Figure 165), change directory to the RTL Verification tutorial, lab1.
- 3. Execute the Tcl script to setup the Vivado HLS project, using the command vivado_hls –f run_hls.tcl, as shown in **Figure 165**.



wivado HLS 2013.2 Command Prompt	
C:\Vivado_HLS_Tutorial>cd RTL_Verification	^
C:\Vivado_HL\$_Tutorial\RTL_Verification>cd lab1	
C:\Vivado_HLS_Tutorial\RTL_Verification\lab1>vivado_hls -f run_hls.tcl	*

Figure 165: Setup the RTL Verification Tutorial Project

4. When Vivado HLS completes, open the project in the Vivado HLS GUI using the command vivado_hls –p dct_prj, as shown in **Figure 166**.

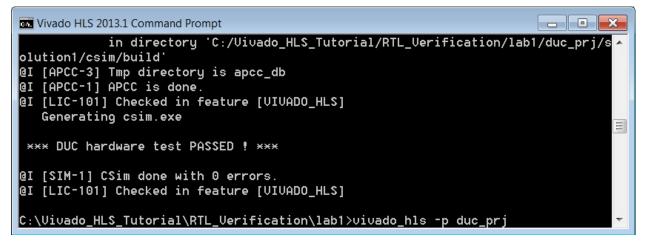


Figure 166: Open RTL Verification Project for Lab 1

Step 2: Perform RTL Verification

- 1. Click the Run C Synthesize toolbar button to synthesize the design to RTL.
- 2. When synthesis completes, use the **Run C/RTL Cosimulation** toolbar button (Figure 167) to launch the **Cosimulation Dialog** box.

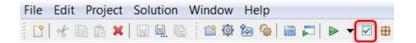


Figure 167: Run C/RTL Cosimulation Toolbar button

The Cosimulation Dialog box shown in Figure 168 opens.



E XILINX.

🔁 Co-simulation Dialog									
C/RTL Co-simulation									
Verilog/VHDL Simulator Selection									
RTL Selection Image: System C Image: Verilog Image: VHDL									
Options Options Dump Trace Optimizing Compile									
Input Arguments									
Do not show this dialog box OK Cance									

Figure 168: Cosimulation Dialog Box

The drop-down menu allows you to select the RTL simulator for HDL simulation. For this exercise, you use the SystemC RTL for cosimulation. No HDL simulator is required; it can be left in the default state or changed. It makes no difference in this first lab.

The default RTL Selection is SystemC, and, in this exercise, you use the SystemC RTL for simulation. Because this can be compiled with the built-in C compiler, you do not need an HDL simulator.

3. Click **OK** to start RTL verification.

When RTL Verification completes, the simulation report opens automatically (Figure 169). The report indicates if the simulation passed or failed. In addition, the report indicates the measured latency and interval.



duc_cosir	n.rpt 🖾							
osimula	tion R	eport	t for	'duc'				
esult								
			Latency	/		Ξ		
RTL	Status	min	avg	max	min	avg	max	
VHDL	NA	NA	NA	NA	NA	NA	NA	
Verilog	NA	NA	NA	NA	NA	NA	NA	
SystemC	Pass	30	31	38	31	32	39	

Figure 169: Cosimulation Report

RTL simulation completes in three steps. To better understand how the RTL verification process is performed, scroll up in the console window to confirm that the messages described below were issued.

First, the C test bench is executed to generate input stimuli for the RTL design.

```
@I [SIM-14] Instrumenting C test bench ...
< C simulation executes to generate input stimuli >
```

At the end of this phase, the simulation shows any messages generated by the C test bench. The output from the C function is not used in the C test bench at this stage, but any messages output by the test bench can be seen in the console.

```
@I [SIM-302] Generating test vectors ...
*** DUC hardware test PASSED ! ***
```

An RTL test bench with newly generated input stimuli is created and the RTL simulation is then performed.

```
@I [SIM-333] Generating C post check test bench ...
@I [SIM-12] Generating RTL test bench ...
...
@I [SIM-11] Starting SystemC simulation ...
```

Finally, the output from the RTL simulation is re-applied to the C test bench to check the results. Once again, you can see any message output by the C test bench in the console. Finally, RTL verification issues message SIM-1000 if the RTL verification passed.

```
SystemC: simulation stopped by user.
@I [SIM-316] Starting C post checking ...
*** DUC hardware test PASSED ! ***
@I [SIM-1000] *** C/RTL co-simulation finished: PASS ***
```





To fully understand why the C test bench should check the results and how message SIM-1000 is generated, you will modify the C test bench.

Step 3: Modify the C test bench

- 1. Expand the **Test Bench** folder in the **Explorer** pane (Figure 170).
- 2. Double-click dut_test.c to open the C test bench in the Information pane.

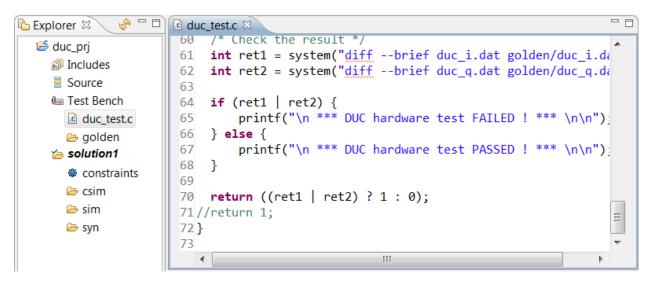


Figure 170: RTL Test bench

- 3. Scroll to the end of the file to see the code shown in Figure 171.
- 4. Edit the return statement to match **Figure 171** and ensure the test bench always returns the value 1.

🗈 *duc_test.c 🛛 🗖 🗖
60 /* Check the result */
61 int ret1 = system("diffbrief duc_i.dat golden/duc_i.da
62 int ret2 = system("diffbrief duc_q.dat golden/duc_q.da
63
64 if (ret1 ret2) {
65 printf("\n *** DUC hardware test FAILED ! *** \n\n")
66 } else {
67 printf("\n *** DUC hardware test PASSED ! *** \n\n");
68 }
69
70//return ((ret1 ret2) ? 1 : 0);
71 return 1; =
72 }
73 🔻
۰ III ا

Figure 171: Modified RTL Test bench

5. Save the file.







- 6. Click the **Run C Synthesize** toolbar button to synthesize the design to RTL.
- 7. Click the **Run C/RTL Cosimulation** toolbar button to launch the **Cosimulation Dialog** box.
- 8. Leave the Cosimulation options at their default value and click **OK** to execute the RTL cosimulation.

When RTL cosimulation completes, the cosimulation report opens and says the verification has failed (Figure 172).

ſ	duc_cosi	im.rpt 🛛									3 » 1	
C	osimul	ation	Repo	rt							An outlir not avail	
C	Result										not avair	able
	RTL Status Latency Interval											
			min	avg	max	min	avg	max				
	VHDL	NA	NA	NA	NA	NA	NA	NA				
	Verilog	NA	NA	NA	NA	NA	NA	NA				
	SystemC	Fail	NA	NA	NA	NA	NA	NA				
	Console		Errors	🛎 Wa	rnings							
VI C	/ado HLS (Console										
	Genera	ting co	osim.	tv.e>	æ							
@	[SIM-3	02] Gei	nerat	ing t	test \	ecto	rs					
;	*** DUC	hardwai	re te	st PA	SSED	! **:	*					
	200											
-	-	-							ro returr	I Vá	alue '1	's -
	E [SIM-3									**		
	<pre>@E [SIM-4] *** C/RTL co-simulation finished: FAIL *** @I [SIM-335] Co-sim total time used: 34 seconds.</pre>											
	[LIC-1											
												-
•			111									•

Figure 172: Cosimulation Report Failure

In **Figure 172**, you can see from the message printed to the console (DUC hardware test PASSED) that the results are correct, however, the verification report says the RTL verification failed.

If required, you can confirm the results are correct. To do this, compare the output files created by the RTL simulation with the golden results. The RTL simulation is executed in the simulation





directory wrapc, which is inside the solution directory. **Figure 173** shows the solution directory, with the output files highlighted.

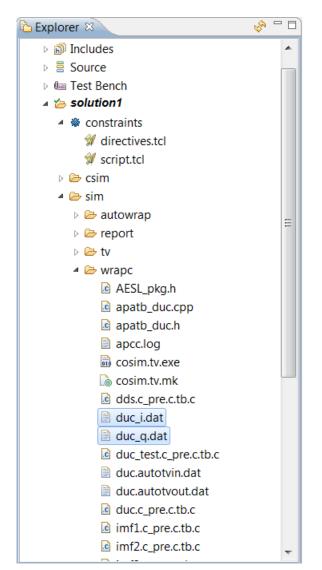


Figure 173: Cosimulation Output Files

RTL Cosimulation only reports a successful verification when the test bench returns a value of 0 (zero). Modifying the test bench to return a non-zero value ensures RTL verification (and C simulation if it was performed) would always report a failure.

To ensure that the RTL results are automatically verified: the C test bench must always check the output from the C function to be synthesized and return a 0 (zero) if the results are correct OR return any other value if they are not correct.

When RTL Verification is performed, the same testing occurs in the test bench, and the output from the RTL block is automatically checked. This is why it is important for the C test bench to check the results and return a zero value only if they are correct (or return a non-zero value if they are incorrect).





9. Exit the Vivado HLS GUI and return to the command prompt.

Lab 2: Viewing Trace Files in Vivado

This exercise explains how to generate RTL trace files and how to view them using the Vivado Design Suite tools.

Step 1: Create an RTL Trace File using Xsim

- 1. From the Vivado HLS command prompt you used in Lab 1, change to the lab2 directory as shown in **Figure 174**.
- 2. Create a new Vivado HLS project by typing vivado_hls -f run_hls.tcl

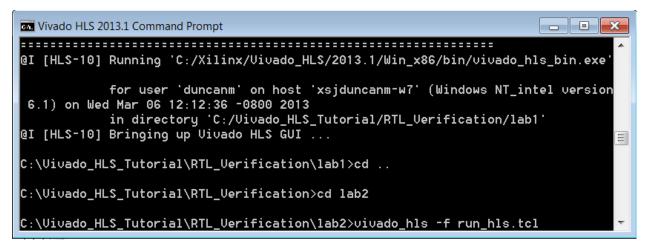


Figure 174: Setup for RTL Verification Lab 2

- 3. Open the Vivado HLS GUI project by typing vivado_hls -p duc_prj.
- 4. Click the **Run C Synthesize** toolbar button to synthesize the design to RTL.
- 5. Click the Run C/RTL Cosimulation toolbar button to launch the Cosimulation Dialog box.

This exercise could use SystemC as in Lab 1, however, the trace file produced by a SystemC simulation is a VCD file. In this case, you produce a trace file you can open using the Vivado Simulator (Xsim).

- 6. In the **Co-simulation Dialog** window:
 - a. Select Xsim from the Verilog/VHDL Simulator Selector (Figure 175).
 - b. De-select SystemC.
 - c. Select Verilog.
 - d. Select the **Dump Trace** option, to have the options shown in Figure 175.
 - e. Click **OK** to execute RTL cosimulation.



E XILINX.

🔁 Co-simulation Dialog									
C/RTL Co-simulation									
Verilog/VHDL Simulator Selection									
RTL Selection									
SystemC Verilog VHDL									
Options Setup Only Dump Trace Optimizing Compile									
Input Arguments									
Do not show this dialog bo OK Cance									

Figure 175: Cosimulation Dialog Box For Lab 2

When RTL verification completes, the cosimulation report automatically opens. The report shows that the Verilog simulation has passed (and the measured latency and interval). In addition, because the Dump Trace option was used with the Xsim simulator option and because Verilog was selected, two traces files are now present in the Verilog simulation directory. These are shown highlighted in **Figure 176**.



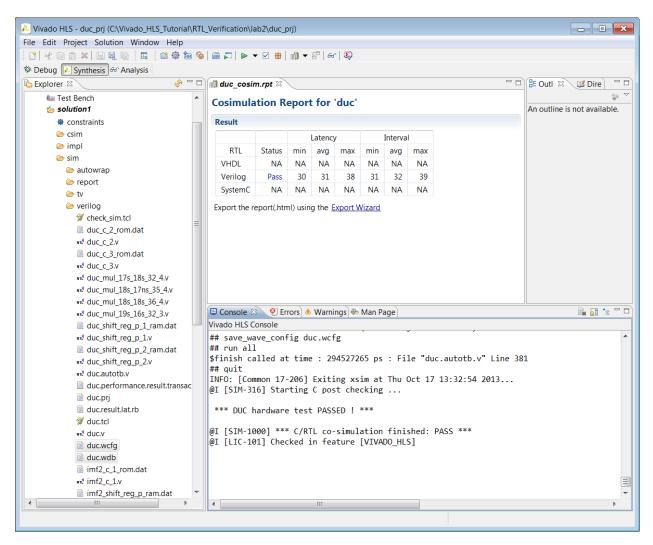


Figure 176: Verilog Xsim Cosimulation Results

The next step is to view the trace files inside the Vivado Design Suite.

7. Exit the Vivado HLS GUI and return to the command prompt.

Step 2: View the RTL Trace File in Vivado

- 1. Launch the Vivado Design Suite (not Vivado HLS):
 - a. On Windows use Start > All Programs > Xilinx Design Tools > Vivado 2013.3 > Vivado 2013.3
 - b. On Linux, type vivado in the shell.
- 2. In the Vivado Tcl Console, enter the following commands, as shown in **Figure 177**. This example assumes the top-level tutorial directory is C:\Vivado_HLS_Tutorial :
 - a. cd /Vivado_HLS_Tutorial/RTL_Verification/lab2/duc_prj/solution1/sim/verilog
 - b. current_fileset





- c. open_wave_database duc.wdb
- d. open_wave_config duc.wcfg

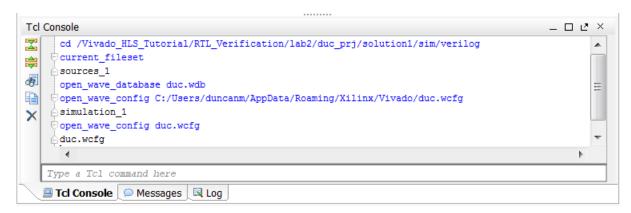


Figure 177: Opening the Trace File in Vivado

You can then view the waveforms in the waveform viewer. **Figure 178** shows the zoomed waveforms where the output data ports and their associated I/O protocol signals (output valid signals) are shown highlighted.

duc.wcfg* ×		_											රි එ
*0													
Rame	Value	. I	77,250 ns	77,300 ns	77,350 ns	77,400 n	s	77,450 ns	77,500	ns	77,550 ns	77,600 ns	77,650
🂐 ⊞ 📲 AESL_clk_counter[31:0]	20406												
💐 👍 ap_clk	1												
🔍 🕼 ap_rst	0												
🐛 🕼 ap_start	1												
ap_done	1										N		
ue an Idle	0												
▶ Wap_ready	1									1			
🖆 🖬 📲 din_i[17:0]	0							0					
🏜 🖬 🖬 freq[15:0]	6628						é	628					
🖵 🎞 dout_i[17:0]	-2			-229		X		118			4	03	
- Un dout i ap vld	1												
t+ dout_q[17:0]	0			552		X		593			4	03	
biv de b juob e	1												
🕂 🖬 📢 done_cnt[31:0]	721	1006	X 100	7 X	1008	X		1009	\square		1010 X	101	1 X
📶 🖽 🐋 AESL_ready_cnt[31:0]	721	1006	X 100	<u>ν χ</u>	1008	X		1009	$ \rightarrow $		1010 X	101	1χ
🔤 🖽 🔜 ready_cnt[31:0]	722	1007	X 100	98 X	1009		X	1010		Х	1011	10	12 X.
😼 ready_initial	0												
🐌 ready_initial_n	x												
🐌 ready_last_n	1												
🐌 ready_delay_last_n	1												
😼 done_delay_last_n	1												
1 interface_done	0		_1										
H M AESL_REG_din_i[17:0]	0							0					
H M AESL_REG_freq[15:0]	6628						6	628					
AESL_REG_dout_i_ap_vld	0												
H M AESL_REG_dout_i[17:0]	37			-229			< <u> </u>	118		×	4	03	
AESL_REG_dout_q_ap_vld	0												
H M AESL_REG_dout_q[17:0]	25			552			<u> </u>	593		×		03	
ၢ <mark>ធ</mark> ap_clk	1												
1∎ ap_rst	0												
ap_start	1												
🖽 📲 din_i[17:0]	0							0					
표 📲 freq[15:0]	6628						é	628					

Figure 178: Analyzing the RTL Trace File

3. Exit and close the Vivado GUI.



4. Type exit to close the Vivado Tcl command prompt.

Lab 3: Viewing Trace Files in ModelSim

This exercise explains how you can generate and view RTL trace files and using the Mentor Graphics ModelSim RTL simulator. Other third-party simulators are supported, and similar process can be used if another RTL simulator is selected.



CAUTION! This lab exercise requires that the executable for ModelSim is defined in the system search path and that the required license to perform HDL simulation is available on the system.

Step 1: Create an RTL Trace File using ModelSim

- 1. From the Vivado HLS command prompt you used in Lab 2, change to the lab3 directory.
- 2. Create a new Vivado HLS project by typing vivado_hls -f run_hls.tcl.
- 3. Open the Vivado HLS GUI project by typing vivado_hls -p duc_prj.
- 4. Click the **Run C Synthesize** toolbar button to synthesize the design to RTL.
- 5. Click the **Run C/RTL Cosimulation** toolbar button to launch the **Cosimulation Dialog** box.

This exercise uses the Mentor Graphics ModelSim RTL simulator. The path to the simulator executable must be set in your system search path.

- 6. In the **Co-simulation Dialog** window:
 - a. Select ModelSim from the Verilog/VHDL Simulator Selector.
 - b. Unselect SystemC.
 - c. Select VHDL.
 - d. Select the **Dump Trace** option, to have the options shown in Figure 179.
 - e. Click OK to execute RTL cosimulation.



A Co-simulation Dialog	×
C/RTL Co-simulation	
Verilog/VHDL Simulator Selection ModelSim	
RTL Selection	
SystemC Verilog VHDL	
Options Setup Only Dump Trace Optimizing Compile	
Input Arguments	
Do not show this dialog bo	

Figure 179: Cosimulation Dialog Box For Lab 3

When RTL verification completes, the cosimulation report automatically opens, showing the VHDL simulation has passed (and the measured latency and interval). In addition, because the Dump Trace option was used with the ModelSim simulator option and because VHDL was selected, a trace file is now present in the VHDL simulation directory. The trace file is shown highlighted in **Figure 180**.



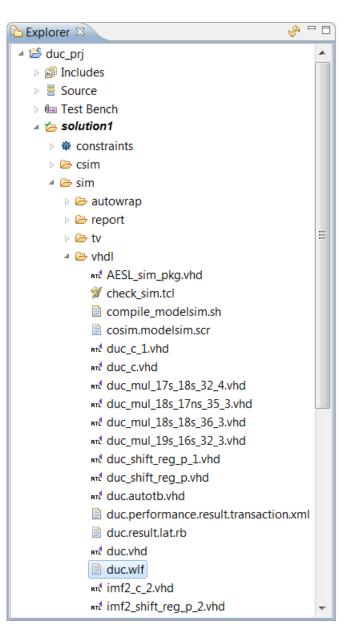


Figure 180: VHDL ModelSim Trace File

The next step is to view the trace files inside ModelSim.

7. Exit the Vivado HLS GUI and return to the command prompt.

Step 2: View the RTL Trace File in ModelSim

- 1. Launch the Mentor Graphics ModelSim RTL Simulator.
- 2. Click the menu File > Open.
- 3. Select Log Files as the file type (Figure 181).
- 4. Navigate to the VHDL simulation directory and select duc.wlf.





5. Click **OK**.

OSDisk (C:) Vivado_HLS_Tutorial New folder ork uc.wlf	 RTL_Verification > lab3 Date modified 3/6/2013 4:52 PM 3/6/2013 4:52 PM 	 duc_prj i solut Type File folder WLF File 	Size	 ✓ 4y Search vhdl ﷺ ✓ 	ر ۲
e fork	3/6/2013 4:52 PM	File folder			. 0
rork	3/6/2013 4:52 PM	File folder			
			3.936 KB		
uc.wlf	3/6/2013 4:52 PM	WLF File	3.936 KB		
File name: duc.wlf				✓ Log Files (*.wlf) ✓ Open	▼ Cancel
	File name: duc.wlf	File name: duc.wlf	File name: duc.wlf	File name: duc.wlf	

Figure 181: ModelSim Open File WLF

6. Add the signals to the trace window and adjust to see a view similar to Figure 182.



Wave		
<u></u>	Msgs	
🔷 ap_clk	0	
	0	
ap_done	0	جنوبة الشريق المرتبة وتوريف المرتبة الوتوجية المرتبة الوتوجية توريقا فراق تقريف
🔷 ap_idle	1	
	0	<u>10 10 10 10 10 10 10 10 10 10 10 10 10 1</u>
🔷 ap_ready	0	
🔷 ap_rst	1	
🔷 ap_start	0	
	0	<u>14 15 16 15 14 13 12 11 12 13 14 15 16 7 18 9 10 11 12 13 14 15 16 17 18 19 10 11 12 13 14 15 16 17 18 19 10 11 12 13 14 15 16 16 16 16 16 16 16 16 16 16 16 16 16 </u>
🔶 c_1_ce0	0	
	X	0 <u>[1]0]-224[11]⁻]9]⁻]81597]⁻2<u>9]⁺]1]⁻[0</u></u>
	X	0 <u>110 [-224]117[97-1]124]117[97]81597 [-2]97]17[0</u> 7-25]-24]-23]-22]-21]-20]-19]-18]-17/0 11 22 13 14 15 16]
E-	0	<u>}-25}-24}-23}-22}-21}-20}-19}-18}-17}0 1 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 1</u>
✓ c_ceu III- ↓ c_load_reg_618	u V	<u>]]2]2]]2]-78]-16]46]0]25]-56]1]]84]1]-680[14]]3]]6]-3]31]]1]</u>
	^ ¥	<u></u>
	^ 0	
	0	
	x	0 γ-1
	x	0 -1
	0	-1)0
	x	0
	x	-229
dout_i_ap_vld	0	
	x	552 593 403 96 -146 -213 -55
dout_q_ap_vld	0	
■-> freq	x	6628
grp_fu_400_ce	1	
	0 ×	0 2 2 2 -78 -16 46 0 25 -56 1 84 1 -680 14 3 6 -3 31 1
	Ŷ	<u>,, /2,/2,/,/2,/ 70/210/40 /0 /2 /23 //30 /11/"/04 /1/"060/14/"/5/5/5/5/1/1</u>
ART Now	259054660 ps	
Gel Cursor 1	0 ps	77000000 ps 77500000 ps 78000000 ps 78500000 ps

Figure 182: Viewing the Trace File in ModelSim

7. Exit and close the ModelSim RTL simulator.

Conclusion

In this tutorial, you learned how to:

- Perform RTL verification on a design synthesized from C and the importance of the test bench in this process.
- Create and open waveform trace files using the Vivado Design Suite.
- Create and open waveform trace files using a third-party HDL simulator (ModelSim) and view the trace file created by RTL verification.





Chapter 9 Using HLS IP in IP Integrator

Overview

You can package the RTL from High-Level Synthesis and use it inside IP Integrator. This tutorial demonstrates how take HLS IP and use it in IP Integrator as part of a larger design.

This tutorial consists of a single lab exercise.

Lab1

Complete the steps to generate two HLS blocks for the IP catalog and use them in a design with Xilinx IP, an FFT. You validate and verify the final design using an RTL test bench.

Tutorial Design Description

You can download the tutorial design file from the Xilinx Website. Refer to the information in





Obtaining the Tutorial Designs.

This tutorial uses the design files in the tutorial directory **Vivado_HLS_Tutorial**\ **Using_IP_with_IPI**.

The design blocks in this tutorial process the data for a complex FFT.

- The Xilinx FFT IP block only operates on complex data. Although you can perform an FFT of real data on a complex data set with all imaginary components set to zero, it can be done more efficiently by pre-processing the data.
- The front-end HLS block in this lab applies a Hamming windowing function to the 1024 (N) real data samples and sends even/odd pairs to an N/2-point XFFT as though they are complex data.
- The back-end HLS block takes bit-reverse ordered data, puts it in natural order and applies an O(N) transformation to FFT output to extract the spectral data for the N-point real data set. Note, the first output pair packs the 0th and 512th (purely real) spectral data point into the real and imaginary parts, respectively.
- The designs are fully-pipelined, streaming designs for high throughput; intended for continuous processing of data, but with throttling capability (stalls if input stalls).
- AXI4 Streaming interfaces are used to connect all blocks in IP Integrator (IPI).

Lab 1: Integrate HLS IP with a Xilinx IP Block

This lab exercise shows how generated two HLS IP blocks, combined them with a Xilinx IP FFT in IP Integrator and verify the design in the Vivado Design Suite.

IMPORTANT: The figures and commands in this tutorial assume the tutorial data directory *Vivado_HLS_Tutorial* is unzipped and placed in the location *C:\Vivado_HLS_Tutorial*.



If the tutorial data directory is unzipped to a different location, or on Linux systems, adjust the few pathnames referenced, to the location you have chosen to place the **Vivado_HLS_Tutorial** directory.

Step 1: Create Vivado HLS IP Blocks

Create two HLS blocks for the Vivado IP Catalog using the provide Tcl script. The script runs HLS C-synthesis, RTL co-simulation and package the IP for the two HLS designs (hls_real2xfft and hls_xfft2real).

- 1. Open the Vivado HLS Command Prompt.
 - a. On Windows use Start > All Programs > Xilinx Design Tools > Vivado 2013.3 > Vivado HLS > Vivado HLS 2013.3 Command Prompt (Figure 183).
 - b. On Linux, open a new shell.





Figure 183: Vivado HLS Command Prompt

- 2. Using the command prompt window, change the directory to Vivado_HLS_Tutorial\
- 3. Using_IP_with_IPI\lab1\hls_designs (Figure 184).
- 4. Type vivado_hls -f run_hls.tcl to create the HLS IP (Figure 184).

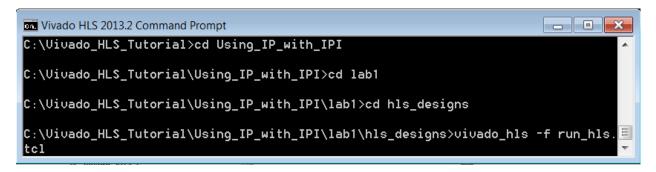


Figure 184: Create the HLS Design for IPI

When the script completes, there are two Vivado HLS project directories, fe_vhls_prj and be_vhls_prj, which contain the HLS IP, including the Vivado IP Catalog archives for use in Vivado designs.

- The "frontend" IP archive is located at fe_vhls_prj/IPXACTExport/impl/ip/
- The "frontend" IP archive is located at be_vhls_prj/IPXACTExport/impl/ip/

The remainder of this tutorial exercise shows how the Vivado HLS IP blocks can be integrated into a design (in IP Integrator) and verified.

Step 2: Create a Vivado Design Suite Project

- 1. Launch the Vivado Design Suite (not Vivado HLS):
 - a. On Windows use Start > All Programs > Xilinx Design Tools > Vivado 2013.3 > Vivado 2013.3
 - b. On Linux, type vivado in the shell.
- 2. From the Welcome screen, click **Create New Project** (Figure 185 185).





Using HLS IP in IP Integrator

👃 Vivado 2013.3	
File Flow Tools Window Help	Q- Search commands
VIVADO 2013.3	E XILINX ALL PROGRAMMABLE.
Getting Started	Documentation
Create New Project New Project Wizard will guide you through the process of selecting design sources and a target device for a new project.	Documentation and Tutorials Invaluable for first time users or to try new features.
Open Project Open one of the most recently used projects or any previously created project.	More detailed info on Vivado commands, dialogs, and buttons.
Open Example Project Open one of the tutorial projects.	Quick Take Videos View a series of short videos on various topics from design flows overview to recommended methodology.
Tcl Console	

Figure 185: Create a Vivado Project

- 3. Click Next on the first page of the Create a New Vivado Project wizard.
- 4. Click the **ellipsis** button to the right of the **Project location text entry box** and browse to the tutorial directory (**Figure 186**).

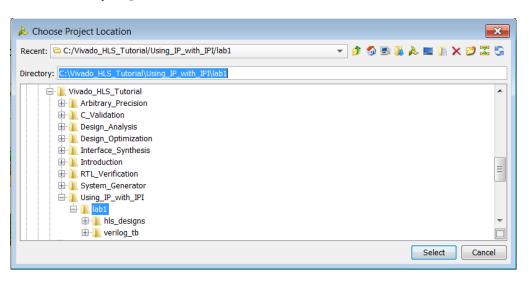


Figure 186: Path to the Vivado Design Suite Project





- 5. Click **Next** to move to the **Project Type** page of the wizard.
 - a. Select RTL Project.
 - b. Do not specify sources at this time (if not the default).
 - c. Click Next.
- 6. On the Default Part page, under Specify, click **Boards** and select the **ZYNQ-7 ZC702 Evaluation Board**, as shown in **Figure 187**.

🕹 New Project					×
Default Part					
Choose a default Xilinx part or board for your	project. This ca	n be changed	l later.		
Specify Filter					
Parts Board Vendor	All			•	
Boards Library	All				
Boards Name					
Version					
version	Latest				
	Reset All	Filters			
Search: Q-					
Board	Board	Board	Board	Board	Part
Board	Vendor	Library	Name	Version	Part
MicroZed Board	em.avnet.com	zynq	microzed	e	🔷 xc7z 🔺
ZedBoard Zynq Evaluation and Development Kit	em.avnet.com	zynq	zed	d	🔷 xc7z
Artix-7 AC701 Evaluation Platform	xilinx.com	artix7	ac701	1.0	🔷 xc7a
Kintex-7 KC705 Evaluation Platform		kintex7	kc705	1.1	
Virtex-7 VC707 Evaluation Platform		virtex7	vc707	1.1	♦ xc7\
Virtex-7 VC709 Evaluation Platform		virtex7	vc709	1.0	
ZYNQ-7 ZC702 Evaluation Board		zynq	zc702	1.0	🔷 xc7z 🚽
ZYNO-7 7C706 Evaluation Board	xilinx.com	zvna	zc706	1.1	🔕 xc77
III					
·					

Figure 187: Vivado Project Specification

7. On the **New Project Summary Page**, click **Finish** to complete the new project setup.

The Vivado workspace populates and appears as shown in Figure 188.



Using HLS IP in IP Integrator

E XILINX.

	grator/lab1/project_1/project_1.xpr] - Vivado 2013.1					_	
File Edit Flow Tools Window Layo						Q - Search commands	
🗱 📴 in 🕫 🏢 👘 🗙 🖻 🛓							Ready
Flow Navigator	Project Manager - project_1						×
🔍 🛣 🖨	Sources _ D 2 ×	Σ	Project Summary ×				ΠĿΧ
Project Manager Roject Settings Add Sources Protect add g	Q Image: I		Project Settings Project name: project_1 Product family: zynq-7000 Project part: 2/ND-7 ZC702 Evaluation Boan Top module name: Not Defined	Edit 🔕	Messages Summary: 0 errors 1 critical warning 0 warnings		۲
4 IP Integrator			> Synthesis	(*)	Implementation		۲
	Hierarchy Libroies Compile Order	Statu Part:	Status: 🖒 Ready	×	Status: Ready Part: xc7z020dg48	34-1 mentation Defaults	~
A RTL Analysis	& Sources 💡 Templates				Summary more states		
Den Elaborated Design	Properties _ L ² ×		ORC Violations	۲	🍏 Timing		۲
 Synthesis Synthesis Settings Run Synthesis 	← → ⁵ / ₂	1	DRC information is not available because it h		Timing information is not availa	able because it hasn't b	
Open Synthesized Design			Utilization	۲	lower 🗐		۲
 Implementation Gi Implementation Settings ▶ Run Implementation ▶ mill Open Implemented Design 			Utilization information is not available because i	t hasn't been run	Power information is not availa	ible because it hasn't b	een run
Program and Debug Bitstream Settings	Design Runs	_					_ 0 & ×
Generate Bitstream	Name Part Constraints	Strate	nv	Status Progres	is Start	Elapsed WNS	
Centrate Elistream	Quarter Part Curror anics Curror anics D=r0=synth_1 xx720201dg48+1 constre_1 D=r0=synth_1 xx720201dg48+1 constre_1	Vivado	Synthesis Defaults (Vivado Synthesis 2013) Implementation Defaults (Vivado Implementation 2013	Not started	s 2007	ciapseu WW2	
							•
	🔚 Td Console 🗋 🗩 Messages 🗌 🗔 Log 🗋 Reports 🖉 Design Run	5					

Figure 188: Vivado Project

Step 3: Add HLS IP to an IP Repository

1. In the Project Manager area of the Flow Navigator pane, click **IP Catalog**.

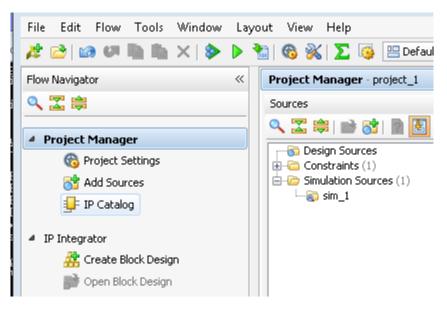


Figure 189: Open the IP Catalog

2. The IP Catalog appears in the main pane of the workspace. Click the **IP Settings** icon.

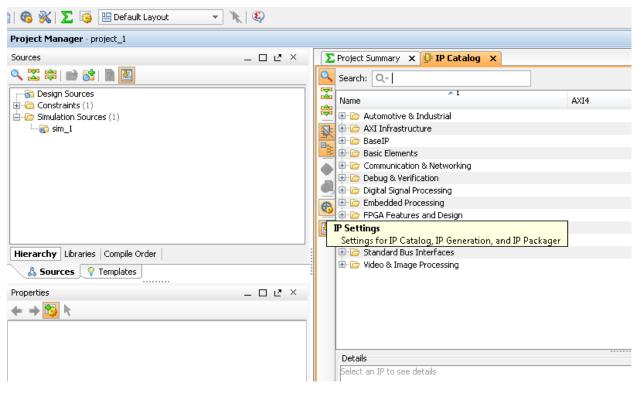


Figure 190: Open the IP Catalog Settings

- 3. In the IP Settings dialog, click Add Repository.
- 4. In the IP Repositories dialog:





- a. Browse to the tutorial files set location.
- b. Click the **Create New Folder** icon.
- c. Enter "vivado_ip_repo" in the resulting dialog (Figure 191).
- d. Click OK.
- e. Click **Select** to close the IP Repository window.

👃 IP Repositories	×
Recent: 🗀 C:/Vivado_HLS_Tutorial/Using	_IP_with_IPI/lab1/viivado_ip_repo 🛛 🚽 🎓 🎲 🗐 🖳 🚴 📰 📗 🗙 🧭 🛣 😘
Directory: C:\Vivado_HLS_Tutorial\Using_` rrogram rises (xoo) ProgramData SymCache Titus Vivado_HLS Vivado_HLS_Tutorial Vivado_HLS_Tutorial C_Validation C_Validation Design_Analysis Design_Optimization Design_Optimization RTL_Verification C_Nalidation Director	
	Select Cancel

Figure 191: Create a New IP Repository

- 5. Back in the IP Setting dialog:
 - a. Click Add IP.
 - b. In the Select IP to Add to Repository dialog box, browse to the location of the HLS IP lab1/hls_designs/fe_vhls_prj/IPXACTExport/impl/ip/.
 - c. Select the xilinx_com_hls_hls_real2xfft_1_0.zip file (Figure 192).
 - d. Click **OK**.





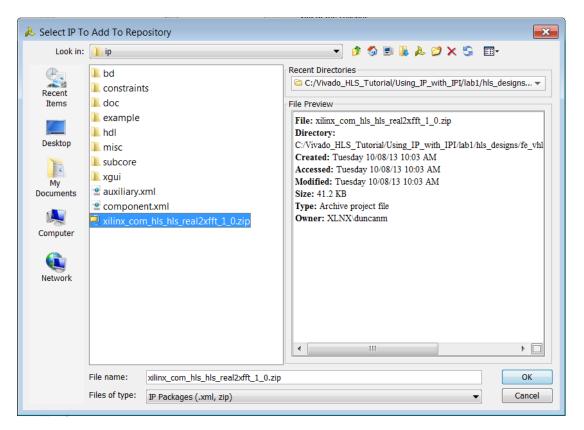


Figure 192: Add the HLS IP to the Repository

- 6. Follow the same procedure to add the 2nd HLS IP package to the repository: xilinx_com_hls_hls_xfft2real_1_0.zip.
- 7. The new HLS IP should now show up in the IP Setting dialog (Figure 193).
- 8. Click **OK** to exit the dialog box.



🚴 Project Settings	×
9.0	IP
<u></u>	Repository Manager Generation Packager
<u>G</u> eneral	Add directories to the list of repositories. After hitting Apply you will be able to see the IP within each repository. You may then add additional IP. If an IP is disabled then a tool-tip
	will alert you to the reason.
Simulation	IP Repositories
	C:/Xilinx/tutorials/HLS_IPIntegrator/lab1/vivado_jp_repo (Project)
Synthesis	×
	*
Implementation	
1010	-
Bitstream	
∎ _ =	Add Repository Ø Refresh All
Īb	IP in Selected Repository
	Hs_real2xfft (xilinx.com:hls:hls_real2xfft:1.00.a)
	Hls_xfft2real (xilinx.com:hls:hls_xfft2real:1.00.a)
	P Add IP
	OK Cancel Apply

Figure 193: IP Repository with HLS IP

A Vivado HLS IP category now appears in the IP Catalog and, if expanded, the HLS IP displays (**Figure 194**).

Σ	Project Summary 🗙 🗜 IP Catalog 🗙					
٩	Search: Q,-					
	Name ^ 1	AXI4	Status License			
a	🗈 🗁 Automotive & Industrial					
1	🕀 🗁 AXI Infrastructure					
	🔁 🖶 🗁 BaseIP					
1 8	🕀 🗁 Basic Elements					
	🖻 🗁 Communication & Networking					
-	🚡 🕀 🗁 Debug & Verification					
	🖳 🖶 🗁 Digital Signal Processing					
6	🐅 🕀 🗁 Embedded Processing					
_	🔤 🖽 🖅 🗁 FPGA Features and Design					
	🗄 🗁 Math Functions					
	🖭 🗁 Memories & Storage Elements					
	🗄 🗁 Standard Bus Interfaces					
	🖶 🗁 Video & Image Processing					
	🖻 🗁 VIVADO HLS IP					
		AXI4-Stream	Pre-production Included			
	Ils_xfft2real	AXI4-Stream	Pre-production Included			

Figure 194: IP Catalog with HLS IP

Step 4: Create a Block Design for RealFFT

- 1. Click **Create Block Diagram** under IP Integrator in the Flow Navigator.
 - a. In the resulting dialog box, name the design RealFFT.
 - b. Click **OK**.

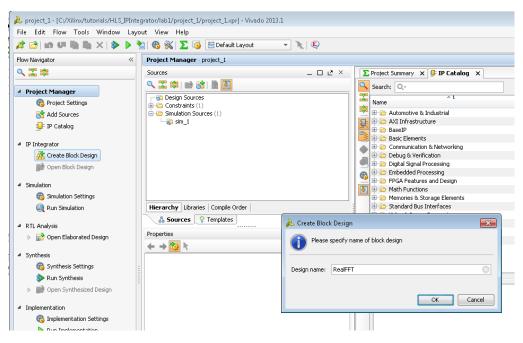


Figure 195: Create Block Diagram

The upper-right pane now has a Diagram tab. Add a Xilinx FFT IP block to the design and customize it.

- 2. In the Diagram tab click the Add IP link in the "get started" message (Figure 196).
 - a. In the Search box type "fourier".
 - b. Press Enter.



199



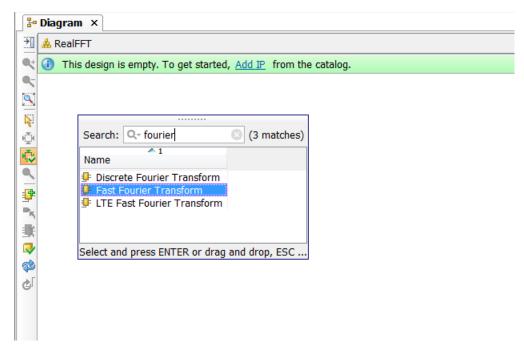


Figure 196: Add the Xilinx FFT IP

The Xilinx IP block FFT is now instantiated in the design, as shown in 197Figure 197.

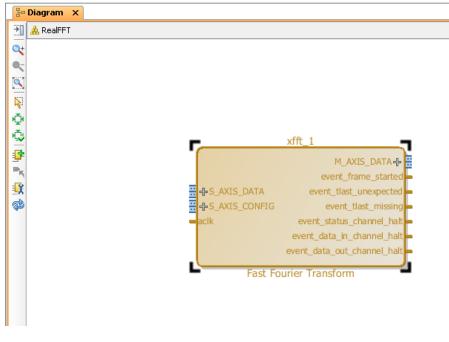


Figure 197: Xilinx FFT IP

3. Double-click the new **Fast Fourier Transform IP Symbol** to open the Re-customize IP dialog box.





4. On the **Configuration** tab (Figure 198):

a. Change the Transform Length to 512.

b. Select Pipelined, Streaming I/O Architecture Choice.

🖵 Re-customize IP	
Fast Fourier Transform (9.0)	🚴
🍘 Documentation 📄 IP Location	
	Component Name RealFFT_xfft_1_0 Configuration Implementation Number of Channels I Transform Length 512 Architecture Configuration Target Clock Frequency (MHz) Target Clock Frequency (MHz) 250 Radix-ture Choice Automatically Select Pipelined, Streaming I/O Radix-4, Burst I/O Radix-2, Burst I/O Radix-2 Lite, Burst I/O Target Data Throughput (MSPS) 50 Range: 1550
<u>«</u>	OK Cancel

Figure 198: Xilinx FFT Configuration

- 5. Select the Implementation tab (Figure 199):
 - a. Select **ARESETN** (active low) in the Control Signals group.
 - b. Verify that Non Real Time is selected as Throttle Scheme.
 - c. Click OK to exit the Re-customize IP dialog box.





🗜 Re-customize IP	
Fast Fourier Transform (9.0)	è.
🍘 Documentation 📄 IP Location	
IP Symbol Implementation Deta 4 I I	Component Name RealFFT_xfft_1_0 Configuration Implementation Detailed Implementation Data Format Fixed Point Scaling Options Scaled Rounding Modes Truncation Precision Options Input Data Width 16 Phase Factor Width 16
M_AXIS_DATA - P - HS_AXIS_CONFIG event_data_in_channel_halt - HS_AXIS_DATA event_data_out_channel_halt - HS_AXIS_DATA event_trame_started - aclk event_status_channel_halt - aresetn event_status_channel_halt - event_tlast_missing - event_tlast_unexpected -	Control Signals ACLKEN ARESETN (active low) ARESETN must be asserted for a minimum of 2 cycles Output Ordering Options Output Ordering Bit/Digit Reversed Order Cyclic Prefix Insertion
<	Optional Output Fields XK_INDEX OVFLO Image: Non Real Time Real Time
	OK Cancel

Figure 199: Xilinx FFT Implementation

Add one instance of each of the HLS generated blocks to the design.

6. Right-click in any space in the canvas and select Add IP (Figure 200).

1-	Diagram ×				
¥.	🙏 RealFFT				
0+					
0,					
0					
S					
Ū.					
ŵ		15	Properties	Ctrl+E	
		\times	Delete	Delete	xfft_1
9		Rh.	Сору	Ctrl+C	
		- Th	Paste	Ctrl+V	event_frame_started
		Pa .	Select All	Ctrl +A	event_tlast_unexpected
4		9	Add IP	Ctrl +I	event_tlast_missing
			Create Hierarchy		event_status_channel_halt
			Create Comment		event_data_in_channel_halt -
			Create Port	Ctrl+K	event_data_out_channel_halt
			Create Interface Port	Ctrl +L	irier Transform
		1	Save as PDF File		
					1

Figure 200: Add IP blocks





- 7. Type "hls" into the Search text entry box.
 - a. Highlight both IPs (Click the control key and select both)
 - b. Press Enter.

The design block now as three IP blocks are shown in Figure 201.

-		
_ *	^a Diagram X	□ Ľ ×
¥.	🛙 📥 RealFFT	
	hls_real2xfft_1 hls_real2xfft_1 ack ap_ready arcsetn ap_done ack ap_ready arcsetn ap_idle His_real2xfft His_real2xfft His_real2xfft	^
	편 thS_AXI 편 thS_AXI ack aresetn	S_CONFIG event_tast_unexpected event_tast_missing event_status_channel_hat event_data_out_channel_hat Fast_Fourier_Transform
	∢	- 一下

Figure 201: RealFFT IP Blocks

The next step is to connect HLS blocks to the FFT block and ports.

- 8. Hover the cursor over the "m_axis_dout" interface connector of Hls_real2xftt block until pencil cursor appears.
 - a. Left-click and hold down the mouse button to start a connection.
 - b. Drag the connection line to "S_AXIS_DATA" port connector of FFT block and release (when green check mark appears next to it).
- 9. In a similar fashion, connect the FFT's "M_AXIS_DATA" interface to the "s_axis_din" interface of the Hls_xfft2real" block.

The two connections are shown in Figure 202.



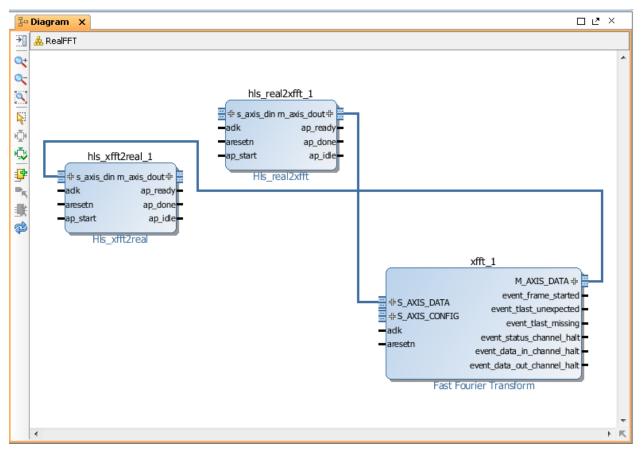


Figure 202: Connecting Ports on the IP Blocks

To create I/O ports for the design, make some external connections.

10. Right-click the "**s_axis_din**" interface connector on Hls_real2xfft block and select **Make External (Figure 203)**.

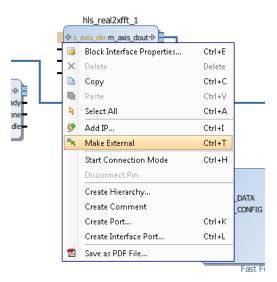


Figure 203: Make External Connections





Give the new interface port a clearly unique name.

- a. Click **port symbol** to highlight it.
- b. In the External Interface Properties pane (Figure 204).
- c. Double-click in the **Name** text entry box to highlight "s_axis_din".
- d. Type in "real2xfft_din" and press Enter.

IMPORTANT: Property changes might not take effect if this re-naming step is not done.

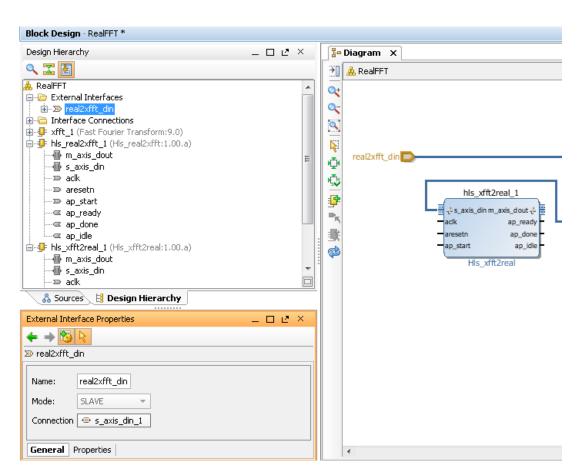


Figure 204: Port Naming

11. In a similar manner to the previous step:

- a. Make the "m_axis_dout" interface of Hls_xfft2real block external and rename it "xfft2real_dout"
- b. Right-click aclk connector of Hls_real2xfft block and select Make External.
- c. Right-click aresetn connector of HIs_real2xfft block and select Make External.
- 12. Tie ap_start ports of HLS blocks high





- a. Right-click canvas, select Add IP.
- b. Type "const" into **Search text** entry box.
- c. Select Constant IP.
- d. Press Enter.
- e. Double-click **Constant IP Symbol** (Figure 205) and verify that the settings for Const Width and Const Val are both '1' and click **OK** to close Re-customize IP dialog box.

IP Re-customize IP					
Constant (1.0)					
W Documentation 📄 IP Location					
Show disabled ports	Component Name RealFFT_xlconstant_1_0				
•	Const Width 1 Range: 14096				
	Const Val 1				
const[0:0]					
< +					
	ок Са	ancel			

Figure 205: Constant IP Properties

f. Connect ap_start of both HLS blocks to the Constant block (Figure 206).



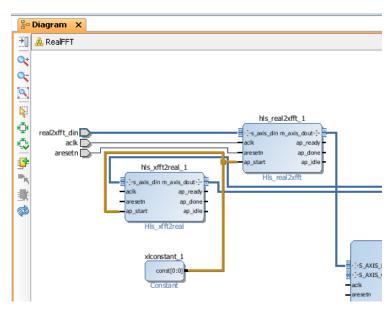


Figure 206: Connect AP_START to Constant 1

- 13. Make the remaining connections.
 - a. Click and drag from the aclk connector of FFT and Hls_xfft2real blocks to the aclk external port (or aclk connector on Hls_real2xfft block or anywhere on "wire" connecting them).
 - b. Connect aresetn of FFT and Hls_xfft2real blocks to aresetn network.
 - c. The XFFT configuration interface is left unconnected, as this design always operates in the default mode of the core.
- 14. Click the **Regenerate** icon to clean up and reorganize the Block Design.



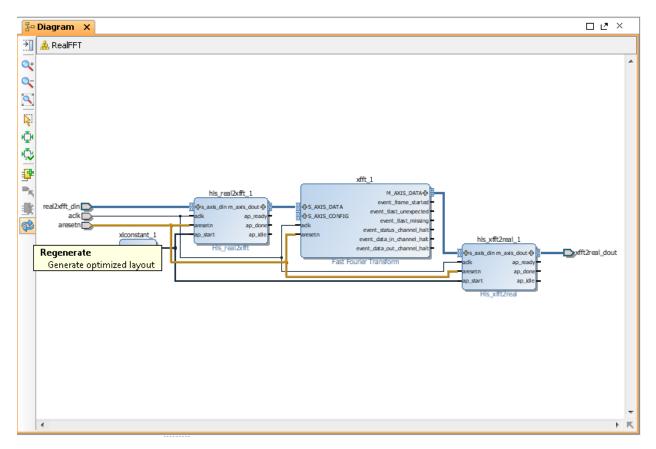


Figure 207: Re-generated Design Diagram

15. Validate the Block Design by clicking the **Validate Design** icon on the toolbar.

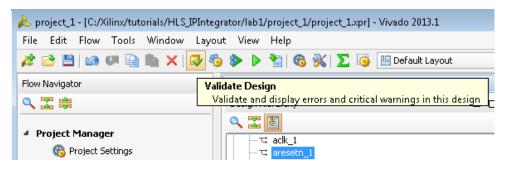


Figure 208: Design Validation

16. Click File > Save Block Design.

- 17. Close the Block Design.
- 18. The next step is to generate output products.
 - a. In the **Sources** tab of Project Manager pane (**Figure 209**), right-click **RealFFT.bd** and select **Generate Output Products**.
 - b. Click **OK** in the resulting dialog to initiate the generation of all output products.



👔 🍪 💥 ∑ 闷 🔚 Default Layout 🔹 🥆 🍳					
Project Manager - project_1					
Sources	Sources _ 🗆 🗠 🗡 🔽 Proje				
🔍 🛣 😂 📄 🛃			Project Settings		
Design Sources (1)		e Pro	iect name: projec		
⊕-	Source Node Properties	Ctrl+E	luct family: Zynq-		
📄 🗁 Simulation Sources (1)	Open File	Alt+O	ect part: ZYNQ		
⊞ ‱ sim_1 (1)	Create HDL Wrapper		module name: <u>Not D</u>		
	View Instantiation Template		Synthesis		
	Generate Output Products				
	Reset Output Products		tus: 📫 Ready		
	Export Hardware for SDK		:: xc7z020clg484		
	Package Block Design		itegy: <u>Vivado Synthe</u>		
Hierarchy IP Sources Lib	Replace File				
🔒 Sources 🛛 🖓 Templa 🖣	Copy File Into Project				
Source Node Properties	Copy All Files Into Project	Alt+I	DRC Violations		
	Remove File from Project	Delete			
🙏 RealFFT (RealFFT.bd)	Enable File	Alt+Equals	DRC information i		
	Disable File	Alt+Minus	_		
Module: Rea	Module: Rei				

Figure 209: Generating Output Products

- 19. Create an HDL Wrapper.
 - a. In the **Sources** tab of the Project Manager pane, right-click **RealFFT.bd** and select **Create HDL Wrapper**. (This is the same procedure and menu as described in the previous step.)
 - b. Click **OK** to clear the resulting notification window.

Step 5: Verify the Design

The next step in creating the final design is to verify design with the HDL test bench provided in the lab exercise: realfft_rtl_tb.v.

- 1. Right-click Simulation Sources in Sources tab of Project Manager pane (Figure 210).
- 2. Select Add Sources.





Block Design - RealFFT						
Sources		_ 🗆 🖻 ×		🚰 Diagr	ram 🗙 🔞 RealFl	
🔍 🔀 ⊜ 📑 🚼				C:/Xilinx/tutorials/HLS_IF		
🔁 🛅 Constraints (1)	rapper (RealFFT_wrappe	r.v) (1)	6	2 /	timescale l p // lib IP_Inte module RealFFT	
È Comparison Sources È Comparison Sources È Comparison Sources	Properties	Ctrl+E	9	4 5	(aclk,	
					aresetn, real2xfft	
	Hierarchy Update	•		7	real2xfft	
Ø	Refresh Hierarchy			8	real2xfft	
	Edit Constraints Sets				real2xfft	
	Edit Simulation Sets			/ 9 10		
C.A.		A11 A		11	xfft2real_	
8	Add Sources	Alt+A	d	12	xfft2real_	
				13	xfft2real_	
				_ 14	xfft2real_	
				15	xfft2real_	
Hierarchy IP Sources	Libraries Compile Order			16	input aclk;	
👌 Sources 🛛 😫 De	sign Hierarchy			17	input areset	
		- D & X		18	input [31:0]	
Properties	1	19	input [3:0]r			
← → <mark>🏷</mark> k				20	input [0:0]r	
				21	output real2	

Figure 210: Adding Simulation Sources

- 3. Select Add or Create Simulation Sources in the Add Sources dialog.
- 4. Click next.
- 5. In the Add Sources dialog box, click the **Add Files** button highlighted in Figure 212.



Add or Create Simulation Sources Specify simulation specific HDL files, or directories containing HDL files, to add to your project. Create a new source file on disk and add it to your project. Specify simulation set: in sim_1 Id Neme Library Location Add Files Add Directories Create File Add Files Add Directories Create File Add Sources into project Add sources from subdirectories Include all design sources for simulation Caped Next > Firish Cancel	🚴 Add Sources	8
specify simulation set: Specify simulation set: Id Specify simulation set: Id Specify simulation set: Id Specify simulation set: Id Specify simulation set: Id Specify simulation set: Id Specify simulation set: Id Specify simulation set: Id Specify simulation set: Add Files Add Directories Create File Create File Add Surces from subdirectories Add sources from subdirectories Include all design sources for simulation	Add or Create Simulation Sources	
Id Name Library Location Id Name Library Location Image: State of the state		
Add Files Add Directories Create File Add Files Add Directories Create File Add Surces from subdirectories I Include all design sources for simulation	Specify simulation set: 🔚 sim_1 💌	
Add Files Add Directories Create File Scan and add RTL include files into project Copy sources into project Add sources from subdirectories Include all design sources for simulation	Id Name Library Location	
Add Files Add Directories Create File Scan and add RTL include files into project Copy sources into project Add sources from subdirectories Include all design sources for simulation		
Add Files Add Directories Create File Scan and add RTL include files into project Copy sources into project Add sources from subdirectories Include all design sources for simulation		
Add Files Add Directories Create File Scan and add RTL include files into project Copy sources into project Add sources from subdirectories Include all design sources for simulation		
Add Files Add Directories Create File Scan and add RTL include files into project Copy sources into project Add sources from subdirectories Include all design sources for simulation		
Scan and add RTL include files into project Copy sources into project Add sources from subdirectories Include all design sources for simulation		F
Scan and add RTL include files into project Copy sources into project Add sources from subdirectories Include all design sources for simulation		
Scan and add RTL include files into project Copy sources into project Add sources from subdirectories Include all design sources for simulation		
Scan and add RTL include files into project Copy sources into project Add sources from subdirectories Include all design sources for simulation		
Copy sources into project Add sources from subdirectories Include all design sources for simulation	Add Files Add Directories Create File	
Add sources from subdirectories Include all design sources for simulation	Scan and add RTL include files into project	
☑ Include all design sources for simulation	Copy sources into project	
	✓ Add sources from subdirectories	
< Back Next > Finish Cancel	☑ Include all design sources for simulation	
< Back Next > Finish Cancel		
	<pre>Gack Next > Finish Ca</pre>	ancel

Figure 211: Add Source Dialog Window

- 6. Browse to the realfft_rtl_tb.v file in the Using_IP_with_IPI\lab1\verilog_tb tutorial directory.
- 7. Select it and click **OK**.
- 8. Select the checkbox Copy sources into the project (Figure 212).



Add Sources						
	Add or Create Simulation Sources Specify simulation specific HDL files, or directories containing HDL files, to add to your project. Create a new source file on disk and add it to your project.					
Specify simulation set: 📾 sim_1 🔹						
	Index	Name	Library	Location		
	1	realfft_rtl_tb.	V WOIK	C:/Vivado_HLS_Tutorial/Using_IP_with_IPI/lab1/verilog_tb	×	
Add Files Add Directories Create File						
Scan and add RTL include files into project						
Copy sources into project						
Add sources from subdirectories						
✓ Include all design sources for simulation						
< Back Next > Finish Cance					ncel	

Figure 212: Copy Design Sources

Note: When you copy the design source files into the project, edits to the file(s) are not automatically propagated to the original source file.

- 9. Click Finish.
- 10. Click Run Simulation in the Flow Navigator (Figure 213).

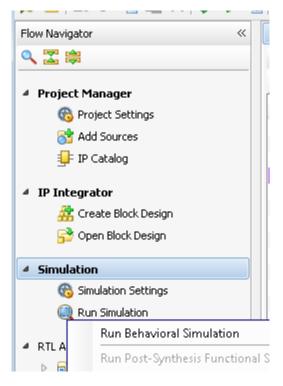


Figure 213: Execute Simulation

11. Once the simulation has started, click the **Run All** icon to complete simulation.

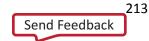
historian i augus corre						
elp						
🖡 😬 Default Layout 💿 🔭 🔣 🔛	n 🕅 🖬 🕶 🧏 📗 🗔 🔇					
ation - Functional - sim_1 - realfft_rtl_tb	Run All (F3)					
	Run the simulation until there are no more events or until a Verilog or '\$finish' or '\$stop'.					
Design Unit Block Type	Name Value Data Type 🎇 Name					

Figure 214: Run The Simulation to Conclusion

Conclusion

In this tutorial, you learned:

- How to create Vivado HLS IP using a Tcl script.
- How to import create a design using IP integrator (IPI) and include both Xilinx IP and the Vivado IP blocks.
- How to verify the design in IPI.





Chapter 10 Using HLS IP in a Zynq Processor Design

Overview

A common use of High-Level Synthesis design is to create an accelerator for a CPU – to move code that executes on the CPU into the FPGA programmable logic to improve performance. This tutorial shows how you can incorporate a design created with High-Level Synthesis into a Zynq device.

This tutorial consists of two lab exercises.

Lab1

You create and configure a simple HLS design to work with the CPU on a Zynq device. The HLS design used in this lab is simple to allow the focus of the tutorial to be on explaining the connections to the CPU and how to configure the software drivers created by High-Level Synthesis to control the device and manage interrupts.

Lab2

This lab illustrates a common high performance connection scheme for connecting hardware accelerator blocks that consume data originating in the CPU memory and/or producing data destined for it in a streaming manner. The lab highlights the software requirements to avoid cache coherency issues.

Tutorial Design Description

You can download the tutorial design file can be downloaded from the Xilinx Website. Refer to the information in



Obtaining the Tutorial Designs.

This tutorial uses the design files in the tutorial directory **Vivado_HLS_Tutorial**\ **Using_IP_with_Zynq**.

The sample design is a simple multiple accumulate block. The focus of this tutorial exercise is the methodology, connections and integration of the software drivers. (The tutorial does not focus on the logic in the design itself.)

Lab 1: Implement Vivado HLS IP on a Zynq Device

This lab exercise integrates both the High-Level Synthesis IP and the software drivers created by HLS to control the IP in a design implemented on a Zynq device.

IMPORTANT: The figures and commands in this tutorial assume the tutorial data directory *Vivado_HLS_Tutorial* is unzipped and placed in the location *C:\Vivado_HLS_Tutorial*.



If the tutorial data directory is unzipped to a different location, or on Linux systems, adjust the few pathnames referenced, to the location you have chosen to place the **Vivado_HLS_Tutorial** directory.

Step 1: Create a Vivado HLS IP Block

Create two HLS blocks for the Vivado IP Catalog using the Tcl script provided. The script runs HLS C-synthesis, runs RTL co-simulation, and packages the IP for the two HLS designs (hls_real2xfft and hls_xfft2real).

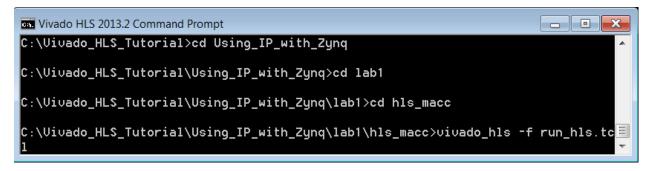
- 1. Open the Vivado HLS Command Prompt.
 - a. On Windows use Start > All Programs > Xilinx Design Tools > Vivado 2013.3 > Vivado HLS > Vivado HLS 2013.3 Command Prompt (Figure 215).
 - b. On Linux, open a new shell.

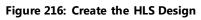


Figure 215: Vivado HLS Command Prompt

- 2. Using the command prompt window, change the directory to Vivado_HLS_Tutorial\Using_IP_with_Zynq\lab1\hls_macc (Figure 216).
- 3. Type vivado_hls -f run_hls.tcl to create the HLS IP (Figure 216).







When the script completes, there is a Vivado HLS project directory vhls_prj, which contains the HLS IP, including the Vivado IP Catalog archive for use in Vivado designs.

The remainder of this tutorial exercise shows how the Vivado HLS IP blocks can be integrated into a Zynq design using IP Integrator.

Step 2: Create a Vivado Zynq Project

- 1. Launch the Vivado Design Suite (not Vivado HLS):
 - a. On Windows use Start > All Programs > Xilinx Design Tools > Vivado 2013.3 > Vivado 2013.3.
 - b. On Linux, type vivado in the shell.
- 2. From the Welcome screen, click Create New Project (Figure 217).





	3 ols Window Help		Q+ Search commands
Getting	Started	Docume	ntation
	<u>Create New Project</u> New Project Wizard will guide you through the process of selecting design sources and a target device for a new project.		Documentation and Tutorials
P	Open Project Open one of the most recently used projects or any previously created project.		User Guide More detailed info on Vivado commands, dialogs, and buttons.
	Open Example Project Open one of the tutorial projects.	8	Quick Take Videos View a series of short videos on various topics from design flows overview to recommended methodology.
Tcl Console			

Figure 217: Vivado Welcome Screen

- 3. In the New Project wizard:
 - a. Click Next.
 - b. In the Project Location text entry box, browse to the location of the tutorial file directory and click **Next (Figure 218)**.
 - c. On the Project Type page, select **"Do not specify sources at this time"** (if it is not the default).
 - d. Click Next.





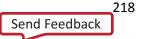
🚴 New Project 🧱
Project Name
Enter a name for your project and specify a directory where the project data files will be stored
Device and the second sec
Project name: project_1
Project location: C:/Vivado_HLS_Tutorial/Using_IP_with_Zynq/lab1
Create project subdirectory
Project will be created at: C:/Vivado_HLS_Tutorial/Using_IP_with_Zynq/lab1/project_1
< Back Next > Finish Cancel

Figure 218: Specify the Vivado Project Directory

- 4. On the Default Part page:
 - a. Click Boards.
 - b. Select the ZYNQ-7 ZC702 Evaluation Board (Figure 219).

🚴 New Project					×
Default Part					
Choose a default Xilinx part or board for your p	project. This ca	in be change	ed later.		
Specify Filter					
Parts Board Vend	for All			*	
() Tarto					
Boards	ary All			*	
	me All			T	
Versi	on Latest			*	
	Rese	t All Filters			
Search: Q-					
Board	Board Vendor	Board Library	Board Name	Board Version	Part
MicroZed Board	em.avnet.com	zyng	microzed	e	xc7z010clg4 _
ZedBoard Zynq Evaluation and Development Kit	em.avnet.com	zyng	zed	d	xc7z020clg4
Artix-7 AC701 Evaluation Platform	xilinx.com	artix7	ac701	1.0	🔷 xc7a200tfbc
Kintex-7 KC705 Evaluation Platform	xilinx.com	kintex7	kc705	1.1	🔷 xc7k325tffg 😑
	xilinx.com	virtex7	vc707	1.1	xc7vx485tff
Virtex-7 VC709 Evaluation Platform	xilinx.com	virtex7	vc709	1.0	xc7vx690tff
ZYNQ-7 ZC702 Evaluation Board	xilinx.com	zynq	zc702	1.0	🔷 xc7z020clg4
CYNO-7 ZC706 Evaluation Board	xilinx.com	7010	70706	1.1	⊗ xc7z045ffa9
		< Ba	ock Next	> Fini	ish Cancel

Figure 219: Specify the Vivado Project Details





- c. Click Next.
- d. Click Finish on the New Project Summary Page.

The project workspace opens as shown in Figure 220.

File Edit Flow Tools Window La	yout View Help					Q - search converse	da .	
2 (2) (2) (2) (3) (3) (3) (3) (3) (3) (3) (3) (3) (3	🐮 🚳 💥 ∑ 🧑 🖾 Defait Layout	· * *					R	rad
Plow Navigator <	Project Manager - project 1							
Q.I.B	Sources	- 0 C ×	E Project Summary X				0.0	x
	へ工中 # 21 目 回		I Partert Settings				and the second	100
# Freject Hanager	- B) Design Sources		and respect seconds.	Edt. (8)	Messages			۲
Project Settings Add Sources P Catalog P Catalog	Constraints (1) Sinulation Sources (1) Light stm_1		Project name: project_3 Project name: 2/nc-7000 Project part: c2/nc/2/de/84-1 Top module name: Not Defined		Sunmary: 0 errors 0 citocal warnings 0 warnings			
Create Block Design			> Synthesis	۲	Implementation			۲
Simulation Simulation Settings	-		Status: ≕\$ Ready Part: xc1s020dq404-1 Strategy: Wiedo Switheres Defaults		Status: Part: xc?a020dg Strategy: Str			
Run Simulation	Hierarchy Libraries Comple Order							
# FTL Analysis	A Sources 🖓 Templatei		1.00		Summery Route Status			
> 📑 Open Elaborated Design	Properties	- 0 ¢ ×	ORC Violations	8	Timing			۲
 Synthesis Synthesis Settings 	* + <mark>2</mark> k		DRC information is not available because it hasn't been run Traing information is not available because it hasn't					
 Run Synthesis III Open Synthesized Design 			📕 Utilization	۲	👹 Power			۲
Inplementation G Implementation Settings P Run Implementation I III Open Implementation			Utilization information is not available becau	se it hasn't been run	Power information is not an	valable because it hasn't l	been run	
Frogram and Debug Bistream Settings	Design Runs		L	_			- 0 0	×
Gerierate Bitstream	Name Part	Constrants 5	trategr	Status Prog	ress Start	Elapsed wh	NS 1115	w
Geon Handware Seasion			vado Synchraes Della de (Novado Synchraes (2011) vado Inglementation Della dis (Novado Inglementation)	Not started [2013) Not started [0%			
	÷) E
	Td Cansole Messages GLog 2	and the second second second						1.5

Figure 220: Initial Vivado Zynq Project

Step 3: Add HLS IP to the IP Catalog

1. In the Project Manager area of the Flow Navigator pane, click IP Catalog.

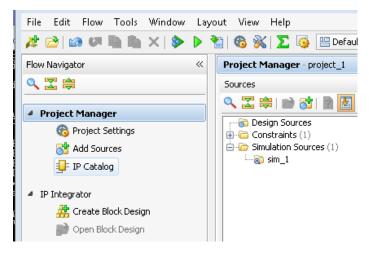


Figure 221: Open the IP Catalog

The IP Catalog appears in the main pane of the workspace.

2. Click the IP Settings icon (Figure 222).



E XILINX.

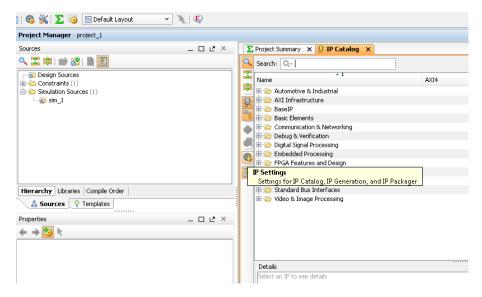


Figure 222: Open the IP Catalog Settings

- 3. In the IP Settings dialog, click Add Repository.
- 4. In the IP Repositories dialog box:
 - a. Browse to the tutorial directory location and click the Create New Folder icon.
 - b. Enter "vivado_ip_repo" in the resulting dialog (Figure 223).
 - c. Click OK.
 - d. Click Select to close the IP Repository.

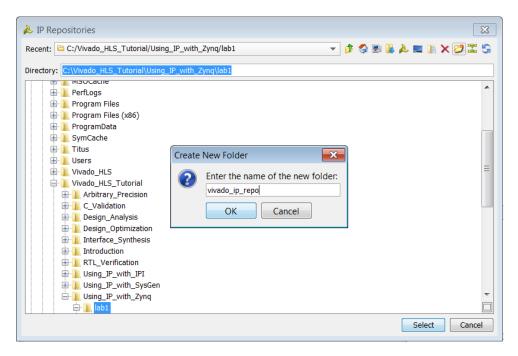


Figure 223: IP Repository





- 5. Returning to the IP Setting dialog box:
 - a. Click Add IP.
 - b. In the Select IP to Add to Repository dialog, browse to the location of the HLS IP: Using_IP_with_Zynq/lab1/hls_macc/vhls_prj/solution1/impl/ip/.
 - c. Select the IP Catalog package Xilinx_com_hls_hls_macc_1_00)a.zip file (Figure 224).
 - d. Click OK.

🚴 Select IP T	o Add To Rep	ository	
Look in:	🚺 ip		🗾 🔮 🗙 🛃 📕 🧟 🏂 📰
Recent Items Desktop Wy Documents Computer Ketwork	 bd constrain doc drivers example hdl misc subcore xgui auxiliary. compone xilinx_core 	m	Recent Directories © C:/Vivado_HLS_Tutorial/Using_IP_with_Zynq/lab1/vivado_ip * File Preview File: xilinx_com_hls_hls_macc_1_0.zip Directory: C/Vivado_HLS_Tutorial/Using_IP_with_Zynq/lab1/hls_macc/vhls_p Created: Tuesday 10/08/13 10:05 AM Modified: Tuesday 10:08/13 10:05 AM Size: 21.1 KB Type: Archive project file Owner: XLNXI duncanm
	File name: Files of type:	xilinx_com_hls_hls_macc_1_0.zip	OK Cancel
	rice of type.	IP Packages (.xml, zip)	▼ Cancei

Figure 224: Add IP to the Repository

6. The new HLS IP should now appear in the IP Settings dialog box.



Project Settings	
	IP
30	Repository Manager Generation Packager
<u>G</u> eneral	Add directories to the list of repositories. After hitting Apply you will be able to see the IP within each repository. You may then add additional IP. If an IP is disabled then a tool-tip will alert you to the reason.
Simulation	IP Repositories
	C;/Xilinx/tutorials/HLS_IPIntegrator/lab2/vivado_jp_repo (Project)
Synthesis	×
Implementation	
1010	¥
Bitstream	
<u>•</u>	Add Repository Ø Refresh All
ĪÞ	IP in Selected Repository
	Hls_macc (xilinx.com:hls:hls_macc:1.00.a)
	\mathbf{X}
	I Add IP Ø Refresh Repository
	OK Cancel Apply

Figure 225: HLS IP in the Repository

- 7. Click **OK** to exit the dialog box.
- 8. There is now a Vivado HLS IP category in the IP Catalog and, if expanded, the Hls_macc IP diplays (Figure 226).

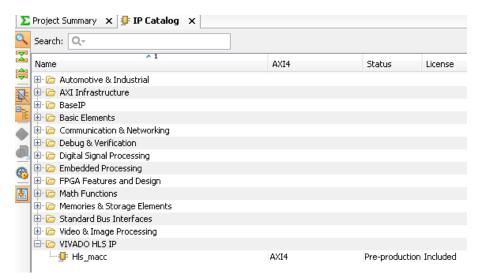


Figure 226: HLS IP in the IP Catalog

Step 4: Creating an IP Integrator Block Design of the System

1. In the IP Integrator area of the Flow Navigator, click **Create Block Design** and enter "Zynq_Design" in the dialog box.





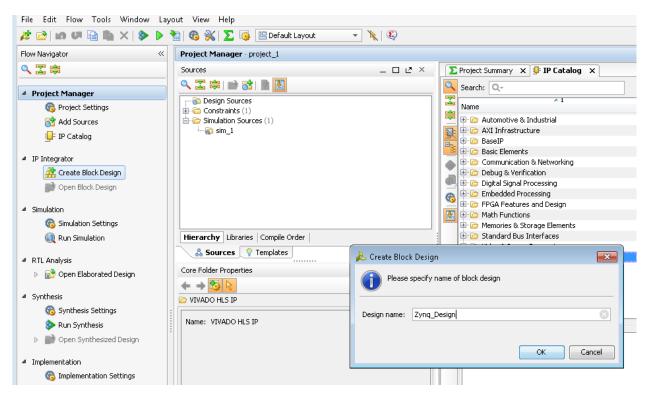


Figure 227: Create the Zynq Design

The Block Design view opens in the main pane, with a new Diagram tab, containing a blank Block Design canvas.

- 2. Click the Add IP link under the title bar, which pops up an IP search dialog.
 - a. Type in "proce" into the Search text entry box.
 - b. Select the **ZYNQ7 Processing System** item and press **Enter**.



X 3	🌾	¢,	
×	🔓 Dia	igram X	
	21 🙏	Zynq_Design	
	•	This design is empty. To get started, Add IP from the catalog.	
	•		
	I		
	Щ.	Search: Q- proce (2 matches)	
	\$	Name	
		ZYNQ7 Processing System ZYNQ7 Processing System BFM	
:	.		
	~	Select and press ENTER or drag and drop, ESC to cancel	

Figure 228: Add a CPU Processor to the Design

An IP symbol for the ZYNQ7 Processing System appears on the canvas.

- 3. Double-click the **ZYNQ IP** symbol to open its **Re-customize IP** dialog.
 - a. Click the **Presets** icon and select **ZC702** (Figure 229).

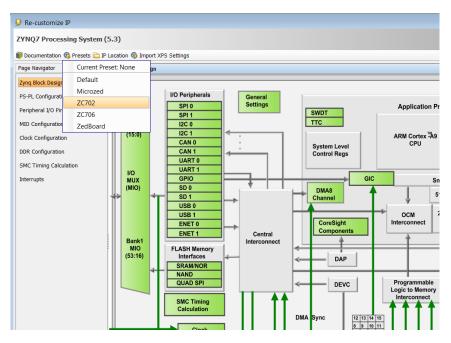


Figure 229: Configure the Zynq Processor

4. Click Interrupts in the Page Navigator pane.





- a. Select Fabric Interrupts and expand its tree view.
- b. Select **IRQ_F2P[15:0]** and click **OK** to close the Re-customize IP dialog box.

🖵 Re-customize IP			
ZYNQ7 Processing System (
Page Navigator 🛛 🐇	Interrupts		
Zyng Block Design	Search: Q-		
PS-PL Configuration	Interrupt Port	ID	Description
MIO Configuration	Fabric Interrupts PL-PS Interrupt Ports		Enable PL Interrupts to PS and vi
MIO Table View		[91:84], [68: 28	Enables 16-bit shared interrupt po Enables Fast private interrupt sig
Clock Configuration	Core0_nIRQ	31 28	Enables private interrupt signal fo Enables Fast private interrupt sig
DDR Configuration	Core1_nIRQ PS-PL Interrupt Ports	31	Enables private interrupt signal fo
SMC Timing Calculation	IRQ_P2F_DMAC_ABORT		Enables shared interrupt abort sig Enables shared interrupt signal 0
Interrupts	IRQ_P2F_DMAC1		Enables shared interrupt signal 1
	···· 🔲 IRQ_P2F_DMAC2		Enables shared interrupt signal 2

Figure 230: Zynq Processor Interrupt Configuration

IPI provides Designer Assistance to automate certain tasks, such as making the correct external connections to DDR memory and Fixed I/O for the ZYNQ PS7.

- 5. Click the Run Block Automation link under the title bar (Figure 231).
 - a. Select /processing_system7_1.
 - b. Click **OK** to complete in the resulting dialog box.



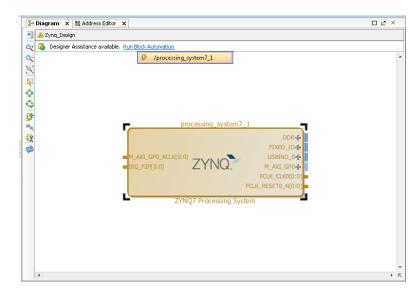


Figure 231: Run Automation

- 6. Add HLS IP to the design by right-clicking in an open space of canvas and by selecting **Add IP** from the context menu.
 - a. Type "hls" in the Search text entry box and press Enter to add it to design (Figure 232).

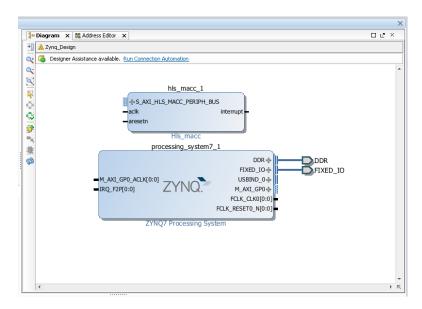


Figure 232: processor and HLS IP

Designer assistance is also available to automate the interconnection of IP blocks.

- 7. Click the **Run Connection Automation** link at the top of the canvas.
- 8. Select /hls_macc_1/S_AXI_HLS_MACC_PERIPH_BUS and click **OK** in the resulting dialog box to automatically connect the HLS IP to the M_AXI_GP0 interface of the PS7.





This adds an AXI Interconnect (instance: processing_system7_1_axi_periph), a Proc Sys Reset block (instance: proc_sys_reset) and makes all necessary AXI related connections to create the design shown in **Figure 233**.

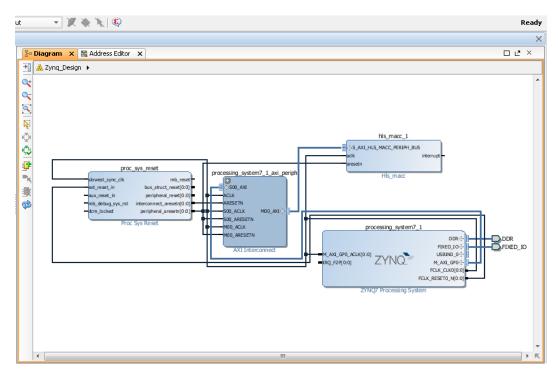


Figure 233: AXI4 Interconnect

The only remaining connection necessary is from the HLS interrupt port to the PS7 IRQ_F2P port.

- 9. Bring the cursor over the interrupt pin on the hls_macc_1 IP symbol.
 - a. When the cursor changes to pencil shape, click and drag to the IRQ_F2P[0:0] port of the PS7 and release, completing the connection
- 10. Bring the **Address Editor** tab forward and confirm that the hls_macc_1 peripheral has been assigned a master address range. If it has not, click the **Auto Assign Address** icon.





🗄 Diagram 🗙 🔣 Address Editor 🗙				
Cell	Base Name	Offset Address	Range	High Address
🔀 🖃 🕩 /processing_system7_1				
😝 🖻 🖽 Data				
/hls_macc_1	Reg	0x43C00000	64K	0x43C0FFFF

Figure 234: Address Editor

The final step in the Block Diagram design entry process is to validate the design.

- 11. Click the **Validate Design** icon in the toolbar.
- 12. Upon successful validation, save (control-s) the Block Design.

Step 5: Implementing the System

Before proceeding with the system design, you must generate implementation sources and create an HDL wrapper as the top-level module for synthesis and implementation.

- 1. Return to the Project Manager view by clicking on **Project Manager** in the Flow Navigator.
- 2. In the Sources browser in the main workspace pane, a Block Diagram object named Zynq_Design is at the top of the Design Sources tree view (Figure 235). Right-click this object and select **Generate Output Products**.
- 3. In the resulting dialog box, click **OK** to start the process of generating the necessary source files.





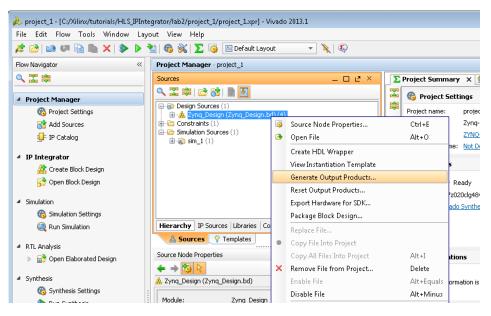


Figure 235: Wrapper Generation

4. Right-click the **Zynq_Design** object again, select **Create HDL Wrapper**, and click **OK** to exit the resulting dialog box.

The top-level of the Design Sources tree becomes the Zynq_Design_wrapper.v file. The design is now ready to be synthesized, implemented, and to have an FPGA programming bitstream generated.

- 5. Click Generate Bitstream to initiate the remainder of the flow.
- 6. In the dialog that appears after bitstream generation has completed, select **Open Implemented Design** and click **OK**.

Step 6: Developing Software and Running it on the ZYNQ System

You are now ready to export the design to Xilinx SDK. In SDK, youcreate software that runs on a ZC702 board (if available). A driver for the HLS block was generated during HLS export of the Vivado IP Catalog package. This driver must be made available in SDK so that the PS7 software can communicate with the block.

1. From the Vivado File menu select Export > Export Hardware for SDK.

Note: Both the IPI Block Design and the Implemented Design must be open in the Vivado workspace for this step to complete successfully.

2. In the Export Hardware for SDK dialog box (Figure 236), ensure that the Include Bitstream and Launch SDK options are enabled and click OK.





🚴 Export Hardv	ware for SDK	×
Export h	nardware platform for SDK.	
Options		
Source:	🙏 Zynq_Design.bd	-
Export to:	🛜 <local project="" to=""></local>	-
Workspace:	🛜 <local project="" to=""></local>	-
🔽 Export Ha	ardware	
🔽 Include b	itstream (Note: an implemented design must be load	ed)
🔽 Launch S	DK	
	ОК	incel

Figure 236: Export to SDK Dialog Window

- 3. SDK opens. If the Welcome page is open, close it.
- 4. Create a new SDK software repository and add the HLS block drivers to it.
 - a) From the XilinxTools menu, select **Repositories.**
 - b) In the Repositories Preferences page click **New** (upper right).
 - c) In the Browse For Folder dialog, navigate to the IP repository directotry vivado_ip_repo directory and select the IP pacjkage xilinx_com_hls_hls_macc_1_0 as shown in Figure 237.
 - d) Select OK to close the specify the repository.
 - e) Select OK to close the SDK project Preferences dialog window.





Preferences		
	Add remove or change the order of SDK's software repositories.	
type filter text General C/C++ Help Install/Update Remote Systems Run/Debug Team Terminal Xilinx SDK Boot Image Flash Programming Hardware Specifici Log Information Le	Add, remove or change the order of SDK's software repositories.	New New New New
Repositories XMD Startup	> > > > <td>Remove Up Down</td>	Remove Up Down
< <u> </u>	Folder: XIIINX_com_nis_nis_macc_1_0 Resc Make New Folder OK Cancel Note: Local repository settings take precedence over global repository settings. Restore Defaults Apply OK	Cancel

Figure 237: SDK Project Properties

- 5. From the SDK File menu, select **New > Application Project**.
 - a) In the New Project dialog enter a project name: Zynq_Design_Test
 - b) Click Next.
 - c) Select the **Hello World** template.
 - d) Click Finish.
- 6. Create a Hello World application (also creates BSP):
 - a) Click File > New > Application Project.

🐮 XILINX.

- b) Enter the project name Zynq_Design_Test
- c) Click Next.
- d) Select Hello World (if not default).
- e) Click Finish.

😡 New Project			
Application Project			6
Project name: Zynq_De:	sign_Test		
Use default location			
Location: C:\Xilinx\tuto	rials\HLS_IPIntegra	tor\lab2\project_1\project_1	Browse
Choose file sys	tem: default 💌		
Hardware Platform hw	_platform_0		•
Processor psi	_cortexa9_0		•
OS Platform	standalone		•
Language	◉C ◯ C++		
Board Support Package	Oreate New	Zynq_Design_Test_bsp	
	🔘 Use existing		
	2		
_			
	< Back	Jext > Finish	Cancel

Figure 238: Application Project

- 7. Power up the ZC702 board and test the Hello World application:
 - b. Ensure the board has all the connections to allow you to download the bit stream on the FPGA device. Refer to the documentation that accompanies the ZC702 development board.
- 9. Click XilinxTools > Launch Hardware Server (Figure 239).
 - C.





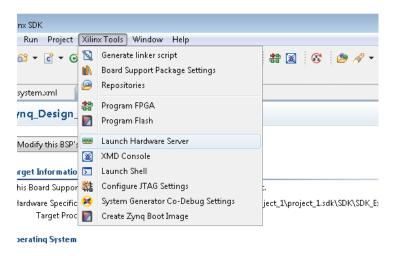


Figure 239: Launch Hardware

10. Click XilinxTools > Program FPGA (or toolbar icon).

Notice that theDone LED (DS3) is now on.

11. Setup a Terminal in the tab at bottom of of workspace:

a) Click the **Connect** icon (Figure 240).

Peripheral Drivers			
Drivers present in the Board Support Package.			
hls_macc_1 hls_macc_top			
ps7_afi_0 generic			
ps7 afi 1 generic		*	
•	*		
Overview Source			
🖹 Problems 🕢 Tasks 🖳 Console 🔲 Properties 🔊 Terminal 1 🛛	14 14	- 🗊 🚮 🖉 🔻	🖬 🕶 🗶 🗖 🗖
No Connection Selected	Connect	7	
			*
			-
	1		

Figure 240: The Connect Icon

- b) Select Connection Type > Serial.
- c) Select the COM port to which the USB UART cable is connected (generally *not* COM1 or COM3)On Windows, if you are not sure, open the Device Manager and identify the port with the Silicon Labs driver under Ports (COM & LPT).
- d) Change the Baud Rate to 115200 (Figure 241).
- e) Click **OK** to exit the **Terminal Settings** dialog box.



E XILINX_{*}

🐵 Terminal Settin	gs 💌						
View Settings: View Title: Ter Encoding: ISO	minal 1 -8859-1 👻						
Connection Typ Serial	Connection Type:						
Settings:							
Port: Baud Rate:	COM5 • 115200 •						
Data Bits: Stop Bits:	8 • 1 •						
Parity:	None •						
Flow Control: Timeout (sec):	None						
OK Cancel							

Figure 241: Terminal Settings

- 12. Right-click the application project **Zynq_Design_Test** in the Explorer pane (Figure 242).
 - a. Click **Run As > Launch on Hardware**.

🗅 Project Explorer 🛛 🕄		👔 system.xml 🙀 system.mss 🛛
a 📁 hw_platform_0	🖻 🔄 🔻 🎽	Zynq_Design_Test_bsp Board Support Package
ps7_init.c ps7_init.h ps7_init.h ps7_init.html ps7_init.tcl system.bit system.xml <u>system.xml Zynq_Design_Tes</u>		Vindow Ctrl + C Ctrl + C Ctrl + V Delete
 i BSP Documer ps7_cortexa9_ libgen.log libgen.option Makefile system.mss 	Source Move	F2 a simple, low-level software layer. It provides acc d exceptions as well as the basic features of a host
	Build Project Clean Project Refresh Close Project Close Unrelated	F5 port Package.
	Build Configura Make Targets Index Show in Remot	ations
	Convert To Run As Debug As	Launch on Hardware Z Local C/C++ Application
	Profile As Team Compare With	3 Remote ARM Linux Application Run Configurations

Figure 242: Run the Application Project

13. Switch to the **Terminal** tab and confirm that "Hello World" was received (Figure 243).





renpneral Drivers		
Drivers present in the Board Support Package.		
hls_macc_1 hls_macc_top		
ps7_afi_0_generic		
ps7 afi 1 generic	_	
•	4	
Overview Source		
🔀 Problems 🖉 Tasks 📮 Console 🔲 Properties 🔎 Terminal 1 🛛	0 " 🗶 🗕 🖬 🖓 🔛 🖬 🖬 🖓 14	
Serial: (COM5, 115200, 8, 1, None, None - CONNECTED) - Encoding: (ISO-8859-1)		
Hello World	A.	I
		I
		I
		Ш
		Ш
		Ш
	-	
None - CONNECTED) - Encoding: (ISO-8859-1)		-



Step 7: Modify software to communicate with HLS block

The completely modified source file is available in the arm_code directory of the tutorial file set. The modifications are discussed in detail below.

- 1. Open the helloworld.c source file.
- 2. Several BSP (and standard C) header files need to be included:

```
#include <stdlib.h> // Standard C functions, e.g. exit()
#include <stdbool.h> // Provides a Boolean data type for ANSI/ISO-C
#include "xparameters.h" // Parameter definitions for processor
peripherals
#include "xscugic.h" // Processor interrupt controller device driver
#include "XHls_macc.h" // Device driver for HLS HW block
```

3. Define variables for the HLS block and interrupt controller instance data. The variables will be passed to driver API calls as handles in the respective hardware.

```
// HLS macc HW instance
XHls_macc HlsMacc;
//Interrupt Controller Instance
XScuGic ScuGic;
```

4. Define global variables to interface with the interrupt service routine (ISR).

```
volatile static int RunHlsMacc = 0;
volatile static int ResultAvailHlsMacc = 0;
```

5. Define a function to wrap all run-once API initialization function calls for the HLS block.

```
int hls_macc_init(XHls_macc *hls_maccPtr)
{
    XHls_macc_Config *cfgPtr;
    int status;
```





}

```
cfgPtr = XHls_macc_LookupConfig(XPAR_XHLS_MACC_0_DEVICE_ID);
if (!cfgPtr) {
    print("ERROR: Lookup of accelerator configuration failed.\n\r");
    return XST_FAILURE;
}
status = XHls_macc_CfgInitialize(hls_maccPtr, cfgPtr);
if (status != XST_SUCCESS) {
    print("ERROR: Could not initialize accelerator.\n\r");
    return XST_FAILURE;
}
return status;
```

6. Define a helper function to wrap the HLS block API calls required to enable its interrupt and start the block.

```
void hls_macc_start(void *InstancePtr){
    XHls_macc *pAccelerator = (XHls_macc *)InstancePtr;
    XHls_macc_InterruptEnable(pAccelerator,1);
    XHls_macc_InterruptGlobalEnable(pAccelerator);
    XHls_macc_Start(pAccelerator);
}
```

An interrupt service routine is required in order for the processor to respond to an interrupt generated by a peripheral.

Each peripheral with an interrupt attached to the PS must have an ISR defined and registered with the PS's interrupt handler.

The ISR is responsible for clearing the peripheral's interrupt and, in this example, setting a flag that indicates that a result is available for retrieval from the peripheral. In general, ISRs should be designed to be lightweight and as fast as possible, essentially doing the minimum necessary to service the interrupt. Tasks such as retrieving the data should be left to the main application code.

```
void hls_macc_isr(void *InstancePtr){
    <u>XHls_macc</u> *pAccelerator = (XHls_macc *)InstancePtr;
    //Disable the global interrupt
    XHls_macc_InterruptGlobalDisable(pAccelerator);
    //Disable the local interrupt
    XHls_macc_InterruptDisable(pAccelerator, 0xffffffff);
    // clear the local interrupt
    XHls_macc_InterruptClear(pAccelerator,1);
    ResultAvailHlsMacc = 1;
    // restart the core if it should run again
    if(RunHlsMacc){
        hls_macc_start(pAccelerator);
    }
}
```

7. Define a routine to setup the PS interrupt handler and register the HLS peripheral's ISR.





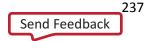
```
int setup_interrupt()
{
   //This functions sets up the interrupt on the ARM
   int result;
  XScuGic_Config *pCfg =
XScuGic_LookupConfig(XPAR_SCUGIC_SINGLE_DEVICE_ID);
   if (pCfg == NULL){
         print("Interrupt Configuration Lookup Failed\n\r");
         return XST FAILURE;
   }
   result = XScuGic_CfgInitialize(&ScuGic,pCfg,pCfg->CpuBaseAddress);
   if(result != XST SUCCESS){
         return result;
   }
   // self-test
  result = XScuGic_SelfTest(&ScuGic);
   if(result != XST_SUCCESS){
        return result;
   }
   // Initialize the exception handler
  Xil_ExceptionInit();
   // Register the exception handler
   //print("Register the exception handler\n\r");
  Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
         (Xil_ExceptionHandler)XScuGic_InterruptHandler, &ScuGic);
   //Enable the exception handler
  Xil_ExceptionEnable();
   // Connect the Adder ISR to the exception table
   //\text{print}("Connect the Adder ISR to the Exception handler table\n\r");
  result = XScuGic_Connect(&ScuGic,
XPAR_FABRIC_HLS_MACC_0_INTERRUPT_INTR,
         (Xil_InterruptHandler)hls_macc_isr,&HlsMacc);
   if(result != XST_SUCCESS){
         return result;
   }
   //print("Enable the Adder ISR\n\r");
  XScuGic_Enable(&ScuGic,XPAR_FABRIC_HLS_MACC_0_INTERRUPT_INTR);
  return XST_SUCCESS;
}
```

8. Define a software model of the HLS hardware functionality with which you can compare reference results.

```
void sw_macc(int a, int b, int *accum, bool accum_clr)
{
    static int accum_reg = 0;
    if (accum_clr)
        accum_reg = 0;
    accum_reg += a * b;
    *accum = accum_reg;
}
```

9. Modify main() to use the HLS device driver API and the functions defined above to test the HLS peripheral hardware.

int main()





```
{
    print("Program to test communication with HLS MACC peripheral in
PL(n(r');
    int a = 2, b = 21;
    int res hw;
    int res_sw;
    int i;
    int status;
    //Setup the matrix mult
    status = hls_macc_init(&HlsMacc);
    if (status != XST_SUCCESS) {
         print("HLS peripheral setup failed\n\r");
         exit(-1);
    }
    //Setup the interrupt
    status = setup_interrupt();
    if(status != XST_SUCCESS){
        print("Interrupt setup failed\n\r");
        exit(-1);
    }
    //set the input parameters of the HLS block
    XHls_macc_SetA(&HlsMacc, a);
    XHls_macc_SetB(&HlsMacc, b);
    XHls_macc_SetAccum_clr(&HlsMacc, 1);
    if (XHls_macc_IsReady(&HlsMacc))
        print("HLS peripheral is ready. Starting... ");
    else {
        print("!!! HLS peripheral is not ready! Exiting...\n\r");
        exit(-1);
    }
    if (0) { // use interrupt
        hls macc start(&HlsMacc);
        while(!ResultAvailHlsMacc)
               ; // spin
        res_hw = XHls_macc_GetAccum(&HlsMacc);
        print("Interrupt received from HLS HW.\n\r");
    } else { // Simple non-interrupt driven test
XHls macc Start(&HlsMacc);
        do {
               res_hw = XHls_macc_GetAccum(&HlsMacc);
         while (!XHls_macc_IsReady(&HlsMacc));
        print("Detected HLS peripheral complete. Result received.\n\r");
    }
    //call the software version of the function
    sw_macc(a, b, &res_sw, false);
    printf("Result from HW: %d; Result from SW: %d\n\r", res_hw, res_sw);
    if (res_hw == res_sw) {
        print("*** Results match ***\n\r");
        status = 0;
    }
    else {
        print("!!! MISMATCH !!!\n\r");
```



```
status = -1;
    }
    cleanup_platform();
    return status;
}
```

10. Save (control-s) the modified source file, and SDK automatically attempts to re-build the application executable. If the build fails, fix any outstanding issues.

Run the new application on the hardware and verify that it works as expected. Ensure that a TCF hardware server is running, that the FPGA is programmed and a terminal session is connected to the UART. Then Launch on Hardware, as you did for the previous Hello World application code.

Upon success, the Terminal session looks similar to Figure 244.

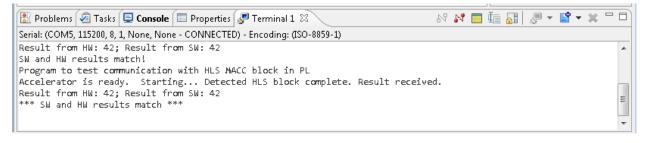


Figure 244: Console Output with Updated C Program

Lab 2: Streaming data between the Zyng CPU and HLS **Accelerator Blocks**

This lab illustrates a common high-performance connection scheme for connecting hardware accelerator blocks that consume data originating in the CPU memory and/or producing data destined for it, in a streaming manner.

- This tutorial uses the same Vivado HLS and XFFT IP blocks created in Lab 1 of the tutorial "Using HLS IP in IP Integrator". In this lab exercise these blocks are connected to the HPO Slave AXI4 port on a Zynq7 processing system via an AXI DMA IP core.
- The hardware accelerator blocks are free-running and do not require drivers; as long as data is pushed in and pulled out by the CPU (often simply referred to as the Processing System or PS).
- The lab highlights the software requirements to avoid cache coherency issues.



239



Step 1: Generate the HLS IP

- 1. From the Vivado HLS command prompt used in Lab 1, change to the lab2 directory as shown in Figure 245.
- 2. Run Vivado HLS to create two HLS IP blocks by typing vivado_hls -f run_hls.tcl.

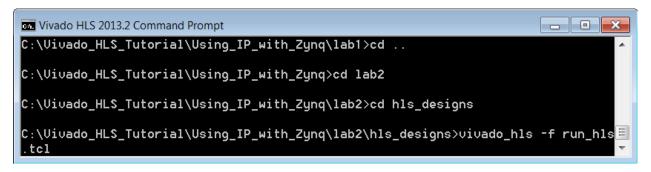


Figure 245: Setup for Zynq Lab 2

When the script completes, there are two Vivado HLS project directories, fe_vhls_prj and be_vhls_prj, which contain the HLS IP, including the Vivado IP Catalog archives for use in Vivado designs.

- The "front-end" IP archive is located at fe_vhls_prj/IPXACTExport/impl/ip/
- The "back-end" IP archive is located at be_vhls_prj/IPXACTExport/impl/ip/

Step 2: Create a Vivado Design Suite Project

- 1. Launch the Vivado Design Suite (not Vivado HLS):
 - a. On Windows use Start > All Programs > Xilinx Design Tools > Vivado 2013.1 > Vivado 2013.1
 - b. On Linux, type vivado in the shell.
- 2. From the Welcome screen, select Create New Project.
- 3. Click **Next** on the first page of the Create a New Vivado Project wizard.
- 4. Click the ellipsis button to the right of the Project location text entry box and browse to the lab2 tutorial directory.
- 5. Click **Next** to move to the Project Type page of the wizard.
 - a. Select **RTL Project** and click **Next**.
 - b. Do not specify sources at this time (if not the default); just click Next.
 - c. Do not add any Existing IP; just click Next.
 - d. Do not add any constraints; just click Next.
- 6. On the Default Part page click **Boards** under Specify and select the **ZYNQ-7 ZC702 Evaluation Board**. Click **Next**.





7. On the New Project Summary Page, click **Finish** to complete the new project setup.

Step 3: Add HLS IP to an IP Repository

- 1. In the Project Manager area of the Flow Navigator pane, click IP Catalog.
- 2. The IP Catalog appears in the main pane of the workspace.
 - a. Click the **IP Settings** icon.
- 3. In the IP Settings dialog box, click **Add Repository**.
- 4. In the IP Repositories dialog box:
 - a. Browse to the Lab 2 tutorial directory lab2.
 - b. Click the **Create New Folder** icon.
 - c. Enter "vivado_ip_repo" in the resulting dialog.
 - d. Click OK.
 - e. Click **Select** to close the IP Repository window.
- 5. On returning to the IP Setting dialog box:
 - a. Click Add IP.
 - b. In the Select IP to Add to Repository dialog box, browse to the location of the HLS IP lab2/hls_designs/fe_vhls_prj/IPXACTExport/impl/ip/ or, if using IP created in previous tutorial, browse to the corresponding path.
 - c. Select the xilinx_com_hls_hls_real2xfft_1_00_a.zip file (Figure 192).
 - d. Click OK.
- 6. Follow the same procedure to add the2nd HLS IP package, in directory lab2/hls_designs/be_vhls_prj/IPXACTExport/impl/ip/ , to the repository: xilinx_com_hls_hls_xfft2real_1_00_a.zip.
- 7. The new HLS IP now appears in the IP Setting dialog box (Figure 193).
- 8. Click **OK** to exit the dialog box.
- 9. There is now a Vivado HLS IP category in the IP Catalog and, if expanded, the HLS IP displays.

Step 4: Create a Top-level Block Design

- 1. Click Create Block Diagram under IP Integrator in the Flow Navigator.
 - a. In the resulting dialog box, name the design Zynq_RealFFT.
 - b. Click **OK**.
- 2. In the Diagram tab, click the **Add IP** link in the "get started" message.
 - a. In the Search box, type "fourier".



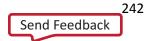


b. Press Enter.

- 3. Double-click the new **Fast Fourier Transform IP** symbol to open the Re-customize IP dialog box. On the Configuration tab:
 - c. Change the Transform Length to 512.
 - d. Change the Target Clock Frequency to 100 MHz.
 - e. In the Architecture Choice section, select Pipelined, Streaming I/O
- 4. Select the **Implementation** tab:
 - a. Select **ARESETN** (active low) in the Control Signals group
 - b. Verify that Bit/Digit Reversed Order is selected under Output Ordering Options
 - c. Verify that Non Real Time is selected as Throttle Scheme.
 - d. Click **OK** to exit Re-customize IP dialog
- 5. Add one instance of each of the HLS generated blocks to the design
 - a. Right-click in any space in the canvas and select Add IP.
 - b. Type "hls" into the Search text entry box.
 - c. Highlight both IPs (Click the control key and select both)
 - d. Press Enter.

Because the output AXI4-Stream interface of the hls_xfft2real block does not include a TKEEP signal, it cannot be directly connected to the AXI DMA (which will be added later). For that reason, you add a Xilinx AXI4-Stream Subset converter: this block configures automatically.

- 6. Right-click in any space in the canvas and select Add IP.
 - a. Type "subset" into the Search text entry box.
 - b. Press Enter..
- 7. Connect the HLS blocks to the FFT block.
 - a. Hover the cursor over the "m_axis_dout" interface connector of the Hls_real2xftt block until a pencil cursor appears.
 - b. Left-click and hold down the mouse button to start a connection.
 - c. Drag the connection line to the "S_AXIS_DATA" input port connector of the FFT block and release when a green check mark appears next to it.
- 8. In a similar fashion:
 - a. Connect the FFT's "M_AXIS_DATA" interface to the "s_axis_din" input interface of the Hls_xfft2real" block.
 - b. Connect the m_axis_dout pin of the hls_xfft2real_1 component to the S_AXIS pint of the axis_subset_converter_1 component





- 9. Now put the data processing blocks into their own level of hierarchy.
 - a. Select everything in the current digram by entering Ctrl+A.
 - b. Right-click the canvas and select Create Hierarchy from the context menu (Figure 246).



Figure 246: Create a Hierarchy Block

- c. In the Create Hierarchy dialog box, enter RealFFT as the Cell name.
- d. Ensure that the **Move '4' selected blocks to new hierarchy** option is checked, as shown in **Figure 247**.





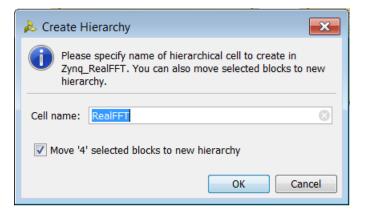


Figure 247: Name Hierarchy Block

e. Click **OK**.

The diagram will look as Figure 248.

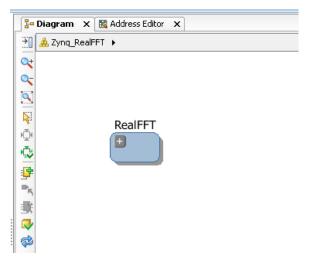
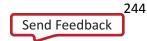


Figure 248: New Hierarchy Block

Add pins to the RealFFT hierarchical block so that you can connect it at the top-level 10. Double-click the **RealFFT** block to open its diagram (**Figure 249**).





1	Diagram	🗙 🔣 Add	ress Editor	🗙 🖁 🏪 Diag	jram - RealFF	тх			00	×
	🙏 Zyng	_RealFFT 🕨	🖲 RealFFT							
Q+										^
Q-										
٩										
2 2 2										
ŵ										
\$										
© ₽										
100										
.										
-			his_real2xfft_1			xft_1 M_AXIS_DATA-P	≓ds_axis_din	ft2real_1	axis_subset_converter_1	
ø		÷jes_ax is_	din North M.	axis_dout-}	-FS_AXIS_DATA	overt_frame_started overt_tlast_uncoped.ed	acik ar esetn	ap_ready ap_done	acik M_AXIS-2	
		ar esetn ap_start		ap_done ap_ide	++s_axis_conFig adk	event_tlast_missing event_status_channel_halt event_status_channel_halt	ap_start His_	apjde dft2real	AX14-Stream Subset Converte	¢
		_	His_real2xfft		Fact Free	event_data_out_channel_halt				
					100.100					

Figure 249: RealFFT Diagram

11. Right-click the s_axis_din pin of the hls_real2xfft_1 block and select **Create Interface Pin** from the context menu (**Figure 250**).

	P								
	-							his_xfft2real_1	
	.					xft_1	⊕s_axis_dn	Water its m_a	kls_dout 🕀 🗄
-	-	his_real2xfft_1				M_AXIS_DA TA 🕀 🗄	adk aresetn ap_start	\mathbf{A}	ap_ready ap_done
-	\$	-	9	Block Interface Properties	Ctrl+E	event_frame_started event_tlast_unexpected event_tlast_missing	арзкат	His_xfft2real	ap_jdle
		1	×	Delete	Delete	event_status_channd_halt event_status_channel_halt			
			Ð	Сору	Ctrl+C	event_data_out_channel_hait			
			th,	Paste	Ctrl+V	rier Transform			
			R.	Select All	Ctrl+A				
			9	Add IP	Ctrl +I				
			₽5	Make External	Ctrl+T				
				Validate Design	F6				
				Start Connection Mode	Ctrl+H				
				Disconnect Pin					
				Create Hierarchy					
		•		Create Comment		III			•
í				Create Pin	Ctrl+K				- 0 2 3
3	is_mm	231		Create Interface Pin	Ctrl+L				
			Ø	Regenerate Layout					
1	/Real: /Rea	I	1	Save as PDF File					

Figure 250: Creating an Interface Pin

- 12. In the Create Interface Pin dialog box, change the Interface name to realfft_s_axis_din as shown in **Figure 251**.
 - a. Accept all other defaults and click **OK**.





🚴 Create Interface Pin 📃 🔀					
Create in	terface pin for cell RealFFT.				
Interface name:	realfft_s_axis_din				
VLNV:	xilinx.com:interface:axis_rtl:1.0				
Mode:	SLAVE 👻				
🔽 Connect to s	elected interface s_axis_din				
	OK Cancel				

Figure 251: Naming an Interface Pin

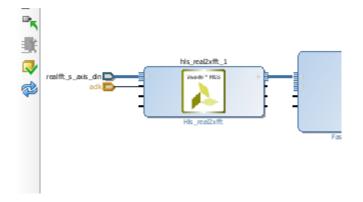
- 13. Right-click the **aclk pin** of the hls_real2xfft_1 block and select **Create Pin** from the context menu.
 - a. Click **OK** to accept all defaults in the Create Pin dialog.

realft. s. axis., din D-	-j-s_iods	hls_real2cfft_1	xfft_	1 M_MS_DATA-J wont_frame_startad wont_flate_unepetid
~~~	<u>6</u>	Block Pin Properties	Ctrl +E	event_tlast_missing nt_status_chennel_heit t_data_in_chennel_heit
	×	Delete	Delete	data_uut_dhannel_halt
	Ð.	Сору	Ctrl+C	ir ans form
	nh.	Paste	Ctrl+V	
	R	Select All	Ctrl+A	
	9	Add IP	Ctrl +I	
	°, ,		Ctrl+T	
			F6	
		Start Connection Mode	Ctrl+H	
		Disconnect Pin		
		Create Hierarchy		
•		Create Comment		
		Create Pin	Ctrl+K	
et_bd_intf_pins ,		Create Interface Pin		od_intf_pins /RealFH
de Slave -vlnv x:	ø	Regenerate Layout		ealfft_s_axis_din'
	1	Save as PDF File		

Figure 252: Create a Clock Pin







Once you create this clock pin, the RealFFT diagram appears as shown in Figure 253.

Figure 253: RealFFT Diagram with Interface Pin and clock pin

- 14. Following the procedures in steps 11 to 13:
  - a. Create an interface pin called 'realfft_m_axis_dout' connected to the M_AXIS pin of the axis_subset_converter_1 component.
  - b. Create a pin for aresetn (from any one of the blocks).

After this step, the RealFFT diagram appears as shown in Figure 254.

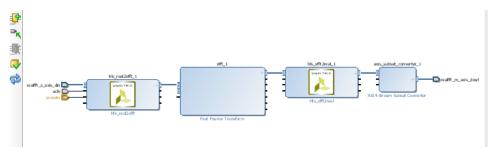


Figure 254: RealFFT Diagram with all pins

Finalize RealFFT block internal connections. The ap_start pins for the HLS blocks are tied HIGH, and the aclk and aresetn pins on all blocks are tied together.

- 15. Right-click the canvas and select **Add IP** from the context menu.
  - a. Type 'const' into the search box and press **Enter**.
  - b. Double-click the **xlconstant_1** component and verify that the Const Val field in the Customize IP dialog is set to '1' (Figure 255).





🗜 Re-customize IP	×
Constant (1.0)	A
🖗 Documentation 늡 IP Location	
Component Name Zynq_RealFFT_xtconstant_1_0 Const Width 1 Const Val	
ОК	Cancel

Figure 255: Create A Constant 1 Tie-Off

- 16. Following techniques covered in Lab1 of this tutorial:
  - a. Connect the output pin of xlconstant_1 to the ap_start pin of hls_real2xfft_1.
  - b. Connect the output pin of xlconstant_1 to the ap_start pin of hls_xfft2real_1.
- 17. Similarly, connect all remaining component aclk and aresetn pins to the RealFFT block diagram aclk and aresetn pins respectively.

Leave the S_AXIS_CONFIG input interface of xfft_1 unconnected. For this tutorial, the default operating modes suffice. Also, leave all other output pins of the components unconnected. The final RealFFT diagram appears with the connections shown in Figure 256.



248



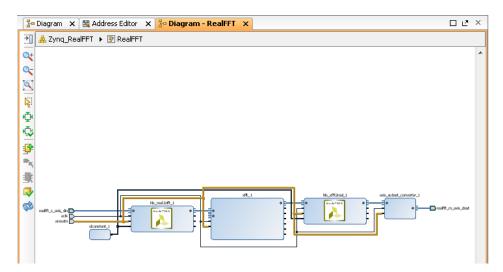


Figure 256: Final RealFFT Diagram

- 18. Close the RealFFT diagram tab and return to the top-level Zynq_RealFFT diagram.
- 19. Create the Zynq system.
  - a. Right-click the canvas of the top-level diagram and select **Add IP** from the context menu.
  - b. Type 'proce' in the search box, select **ZYNQ7 Processing System** and press **Enter**.
  - c. Double-click the **processing_system7_1** component to enter the Re-customize IP wizard for the ZYNQ7.
  - d. Click the **Presets** button near the top of the wizard screen, select the **ZC702 Development Board Template**, and click **OK**.
  - e. Click **PS-PL Configuration** in the Page Navigator pane on the left of the wizard.
  - f. Expand the HP Slave AXI Interface category and check the box for the S AXI HP0 interface, leaving the S AXI HP0 DATA WIDTH at 64.



249



	System	(5.2)		
Documentation 🛅 IP Loc	ation 🍈 P	resets		
age Navigator	« P5	S-PL Configuration		
Zynq Block Design	4	Search: Q-		
PS-PL Configuration		Name	Select	Description
Peripheral I/O Pins		🕀 General		
MIO Configuration		GP Master AXI Interface     GP Slave AXI Interface		
Clock Configuration		HP Slave AXI Interface		
ODR Configuration		S AXI HPO Interface	64	Enables AXI high performance slave interface 0 Allows HPO to be used in 32/64 bit data width mo
5MC Timing Calculation		s AXI HP1 interface		Enables AXI high performance slave interface 1
				Enables AXI high performance slave interface 2
interrupts			(TT)	Enables AXI high performance slave interface 3

Figure 257: Configuring Port HP0

g. Select **Clock Configuration** in the Page Navigator, expand PL Fabric Clocks, and change the requested frequency to 100 (MHz).

YNQ7 Processing	System	(5.2)					
in the interesting i	oystem	(012)					
Documentation 🛅 IP Loc	ation 🍈 Pr	resets					
Page Navigator	« Cla	ock Configuration					
Zynq Block Design	4	Input Frequency (MHz) 33.33333	з 📀 СРИ	Clock Ratio 6:2:1	-		
PS-PL Configuration		Search: Q-					
Peripheral I/O Pins	e 🛛 😂	Component	Clock Source	Requested Frequen	Actual Frequency(M	Range(MHz)	
MIO Configuration	8	Processor/Memory Clocks     IO Peripheral Clocks					
Clock Configuration		🖨 PL Fabric Clocks	r				
DDR Configuration		- FCLK_CLK0	IO PLL IO PLL	▼ 100 50	100.000000 50.000000	0.100000 : 250.000000 0.100000 : 250.000000	
5MC Timing Calculation		- FCLK_CLK2	IO PLL	50	50.000000	0.100000 : 250.000000	
- h		FCLK_CLK3	IO PLL	50	50.000000	0.100000 : 250.000000	
Interrupts							

Figure 258: Configuring the Clock

- h. Leave all other settings at their defaults; click **OK** to apply customizations.
- 20. Note the Designer Assitance Available notification at the top of the screen.
  - a. Run Block Automation on /processing_system7_1.
  - b. Click **OK** in the resulting dialog box.
- 21. Add AXI DMA IP to allow the PS to stream data to/from the RealFFT block via its HP0 Slave AXI interface
  - a. Right-click the canvas and select **Add IP** from the context menu.





- b. Type 'direct' into the search box and select AXI Direct Memory Access from the menu and press **Enter**.
- 22. Double-click the **axi_dma_1** component to open its Re-customize IP dialog and make the following changes (Figure 259):
  - a. Disable the Scatter Gather Engine (deselect the option).
  - b. Set the Memory Map Data Width to 64 for both Read and Write channels.
  - c. Set the Stream Data Width to **16** for the Read channel (MM2S).
  - d. Leave the Stream Data Width at 32 for the Write channel (S2MM).
  - e. Set the Max Burst Size to 128 for both channels.
  - f. Enable Allow Unaligned Transfers for both channels.

🗜 Re-customize IP						
AXI Direct Memory Access (7.0)	è.					
🖉 Documentation 📄 IP Location						
Show disabled ports	Component Name Zynq_RealFFT_axi_dma_1_0					
·	Enable Asynchronous Clocks (Auto)					
	Enable Scatter Gather Engine					
	Enable Multi Channel Support					
₩ S_AXIS_S2MM mm2s_prmry_reset_out_n	Enable Control / Status Stream					
	Width of Buffer Length Register (8-23) 14 💿 bits					
-axi_resetn M_AXI_MM2S	Enable Read Channel      Enable Write Channel					
m_axi_mm2s_ack M_AXI_S2MM+ m_axi_s2mm_ack	Number of Channels 1  Number of Channels 1					
s_axi_lite_aclk s2mm_introut	Memory Map Data Width 64   Stream Data Width 16   Stream Data Width 32					
	Max Burst Size 128  Max Burst Size 128					
	Allow Unaligned Transfers					
	Use Rxlength In Status Stream					
	OK Cancel					

Figure 259: Configuring the AXI Direct Memory Access

23. Note that Designer Assistance is again available. Run **Connection Automation** on /axi_dma_1/S_AXI_LITE and click **OK** in the resulting dialog box.

After running Design Assistance, the diagram appears similar to the one shown in Figure 260.





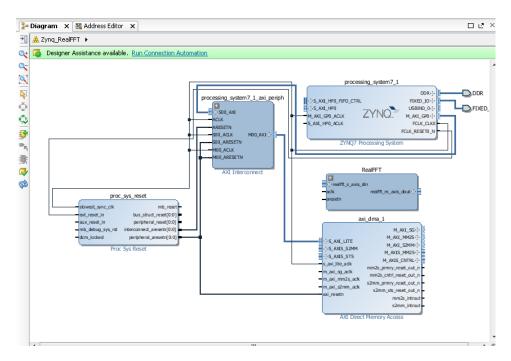


Figure 260: Zynq Diagram with Internal Connections

24. Run Connection Automation on /processing_system7_1/S_AXI_HP0 and click **OK** to accept the default connection in the dialog box.

**Note**: the Connection Automation only connects one of the AXI DMA components M_AXI_* ports through the axi_mem_intercon component.

25. Double-click the **axi_mem_intercon** component to re-customize it.

- a. Change the Number of Slave Interfaces from 1 to 2. (Figure 261).
- b. Click OK.





🗜 Re-customize IP			X						
AXI Interconnect (2.0)									
🍘 Documentation 💼 IP Location									
Component Name Zyng_RealFFT_axi_mem_inter	con_3								
Top Level Settings Slave Interfaces	Master Interfaces								
Number of Slave Interfaces	2								
Number of Master Interfaces	1 ~	•							
Interconnect Optimization Strategy	Custom	•							
AXI Interconnect v2.0 includes IP Integrator a When the endpoint IPs attached to the in width, dock or protocol, a converter IF a converter IP is inserted, IP integrat configures the converter to match the v To see which conversion IPs have been 'expand hierarchy' buttons to explore in NOTE:Addressing information for AXI Intercom Enable Advanced Configuration Options	interfaces of the AXI Interc IP will automatically be adde or's parameter propagation design. inserted, use the IP integra iside the AXI Interconnect h	connect differ Id inside the interconnect. automatically stor ierarhcy.	OK Cancel						

Figure 261: Customizing the AXI Interconnect

- 26. Make a connection between the M_AXI_S2MM port on axi_dma_1 component and S01_AXI port on the axi_mem_intercon component.
- 27. Connect the clocks and reset ports.
  - a. Connect the axi_mem_intercon S01_ACLK and S01_ARESETN ports to the appropriate nets already present in the diagram (processing_system7_1_fclk_clk0 and proc_sys_reset_peripheral_aresetn, respectively).
  - b. Connect the m_axi_s2mm_aclk port of the axi_dma_1 component to the clock network.
- 28. Connect the RealFFT block to rest of the sytem.
  - a. Make a connection between the realfft_s_axis_din input of the RealFFT block and the M_AXIS_MM2S output of the axi_dma_1 component.
  - b. Make a connection between the realfft_m_axis_dout output of the RealFFT block and the S_AXIS_S2MM input of the axi_dma_1 component.
  - c. Connect the aclk and aresetn pin of the RealFFT block to the existing networks.
- 29. Finalize the IPI block diagram design.
  - a. Select the Address Editor tab and click the Auto Assign Address icon (Figure 262).





	Base Name	Offset Address	Range	Hig
S_AXI_LITE	Reg	0x40400000	64K 🔹	n Ox4
o all unmapped slaves				
•	S_AXI_LITE			

Figure 262: Auto Assign System Addresses

30. To view the completed design, run Validate Design by clicking the icon in the toolbar (Figure 263).

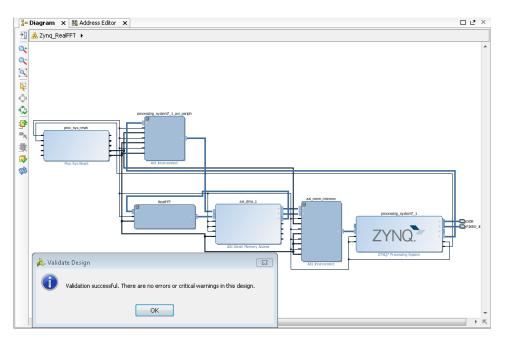


Figure 263: Final Validated Design

## **Step 5: Implementing the System**

Before proceeding with the system design, you must generate implementation sources and create an HDL wrapper as the top-level module for synthesis and implementation.

- 1. Return to the Project Manager view by clicking **Project Manager** in the Flow Navigator.
- 2. In the Sources browser in the main workspace pane, a Block Diagram object named Zynq_ RealFFT appears at the top of the Design Sources tree view. Right-click this object and select **Generate Output Products**.
- 3. In the resulting dialog box, click **OK** to start the process of generating the necessary source files.





4. Right-click the **Zynq_RealFFT** object again, select **Create HDL Wrapper**, and click **OK** to exit the resulting dialog box.

The top-level of the Design Sources tree becomes the Zynq_ RealFFT _wrapper.v file. You are now ready to synthesize, implement, and generate an FPGA programming bitstream for the design.

- 5. Click **Generate Bitstream** to initiate the remainder of the flow.
- 6. In the dialog that appears after bitstream generation has completed, select **Open Implemented Design** and click **OK**.

## Step 6: Setup SDK and test the ZYNQ System

You are now ready to export the design to Xilinx SDK. In SDK, you create software to be run on a ZC702 board (if available). A driver for the HLS block was generated during HLS export of the Vivado IP Catalog package and must be made available in SDK for the PS7 software to communicate with the block.

1. From the Vivado File menu select **Export > Export Hardware for SDK.** 

*Note*: Both the IPI Block Design and the Implemented Design must be open in the Vivado workspace for this step to complete successfully.

- 2. In the Export Hardware for SDK dialog box, ensure that the **Include Bitstream** and **Launch SDK** options are checked, and click **OK**.
- 3. SDK opens. If the Welcome page is open, close it.
- 4. Create a Hello World application (also creates BSP).
  - a. Select File > New > Application Project.
  - b. Enter the project name Zynq_RealFFT_Test.
  - c. Click Next.
  - d. Select Hello World (if it is not the default).
  - e. Click Finish.
- 5. Power up the ZC702 board and program the FPGA.

Ensure the board has all the connections to allow you to download the bit stream on the FPGA device. Refer to the documentation that accompanies the ZC702 development board.

- a. Select **XilinxTools > Program FPGA**. The Done LED (DS3) goes on.
- 6. Set up a Terminal in the tab at bottom of workspace:
  - a. Click the **Connect** icon.
  - b. Select **Connection Type > Serial.**
  - c. Select the COM port to which the USB UART cable is connected (generally *not* COM1 or COM3)On Windows, if you are not sure, open the Device Manager and identify the port with the Silicon Labs driver under Ports (COM & LPT).





- d. Change the Baud Rate to 115200.
- e. Click **OK** to exit Terminal Settings dialog box.
- f. Check that terminal is connected by message in tab title bar.
- 7. Right-click application project Zynq_Design_Test in the Explorer pane
  - a. Select Run As > Launch on Hardware.
- 8. Switch to the Terminal tab and confirm that "Hello World" was received.
- 9. This project uses the C math library (libm), so you must adjust the build settings to link to it.
  - a. Right-click the **zynq_realfft_test project** in the Project Explorer pane and select **C/C+ Build Settings** (Figure 264).

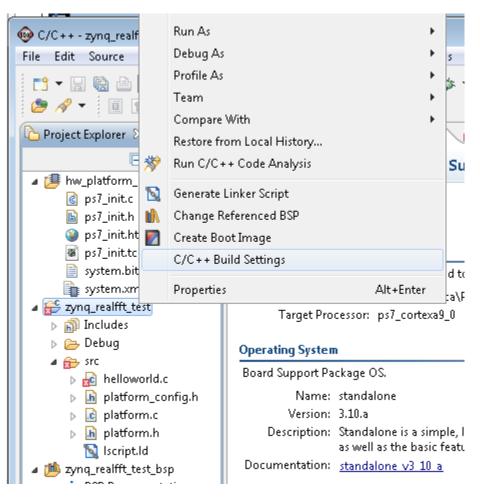


Figure 264: Specify C/C++ Build Settings

- b. Add the ARM gcc linker libraries.
  - i. In the Tool Settings tab, select 'ARM gcc linker' > Libraries.
  - ii. Click the **Add** icon (see Figure 265).





Properties for zynq_realfft_test		
type filter text	Settings	
<ul> <li>Resource Builders</li> <li>C/C++ Build Build Variables Discovery Options Environment Logging Settings</li> <li>Tool Chain Editor</li> <li>C/C++ General Project References Run/Debug Settings</li> </ul>	<ul> <li>Tool Settings Build Steps</li> <li>ARM gcc assembler</li> <li>General</li> <li>ARM gcc compiler</li> <li>Symbols</li> <li>Warnings</li> <li>Optimization</li> <li>Debugging</li> <li>Profiling</li> <li>Directories</li> <li>Miscellaneous</li> <li>Inferred Options</li> <li>Software Platform</li> <li>Processor Options</li> <li>ARM gcc linker</li> <li>General</li> </ul>	Build Artifact 💼 Binary Parsers 📀 Error Parsers
	🖄 Libraries 2 Miscellaneous 2 Linker Script	Library search path (-L) 🕢

Figure 265: C/C++ Build Settings

c. Enter 'm' in the text box in the Enter Value dialog box and click **OK**.

🚳 Enter Value	
Libraries (-I)	
m	
	OK Cancel

#### Figure 266: Library Setting

d. Click **OK** to exit the Properties for zynq_realfft_test dialog box.

### Step 7: Modify software to communicate with HLS block

The completely modified source file is available in the arm_code directory of the tutorial file set. The modifications are discussed in detail below.

- 1. Open the helloworld.c source file.
- 2. Several BSP (and standard C) header files must be included:

```
#include <stdlib.h> // Std C functions, e.g. exit()
#include <math.h> // libm header: sqrt(), cos(), etc
```

High-Level Synthesis UG871 (v 2013.3) November 8, 2013





#include "xparameters.h" // System parameter definitions
#include "xaxidma.h" // Device driver API for AXI DMA

3. Define the (real data) transform length of the FFT:

#define REAL_FFT_LEN 1024

4. Define a custom complex data type with 16-bit real and imaginary members:

```
typedef struct {
    short re;
    short im;
} complex16;
```

5. Declare helper functions before the definition of main(); they will be defined later.

**Note:** The init_dma() function wraps up all run-once, initialization AXI DMA driver API calls and checks that hardware initialization is successful before returning or exiting on an error condition. The generate_waveform() function is fills an array with a simple, periodic waveform to be used as input stimulus for the RealFFT accelerator.

```
int init_dma(XAxiDma *axiDma);
void generate_waveform(short *signal_buf, int num_samples);
```

6. Modify main() to generate and send input data to the RealFFT accelerator and receive the spectral data from it via the AXI DMA engine. Sections of particular importance will be discussed in detail.

```
// Program entry point
int main()
{
```

a. Declare an XAxiDma instance that will be used as a handle to the AXI DMA hardware:

```
// Declare a XAxiDma object instance
XAxiDma axiDma;
```

b. Declare variable for local data storage:

```
// Local variables
int i, j;
int status;
static short realdata[4*REAL_FFT_LEN];
volatile static complex16 realspectrum[REAL_FFT_LEN/2];
```

c. Run platform and DMA initialization functions:

```
// Initialize the platform
```



d. Generate a stimulus waveform:

```
// Generate a waveform to be input to FFT
for (i = 0; i < 4; i++)</pre>
```

```
generate_waveform(realdata + i * REAL_FFT_LEN, REAL_FFT_LEN);
```

e. Before making the DMA transfer request, the buffer containing the data must be flushed from the processor's data cache. Without this step, the DMA might pull stale data from the DRAM.

```
// *IMPORTANT* - flush contents of 'realdata' from data cache to memory
// before DMA. Otherwise DMA is likely to get stale or uninitialized data
Xil_DCacheFlushRange((unsigned)realdata, 4 * REAL_FFT_LEN * sizeof(short));
```

f. Request DMA transfer from PS to PL. Enough data to fill the front-end block and the FFT processing pipelines must be sent in order for spectral data to be ready when the PL to PS transfer is requested. Therefore, four data sets are sent before the first output set is requested:

// Do multiple DMA <u>xfers</u> from the RealFFT core's output stream and // display data for bins with significant energy. After the first frame, // there should only be energy in bins around the frequencies specified // in the generate_waveform() function - currently bins 191~193 only for (i = 0; i < 8; i++) {</pre>

g. Request DMA transfer of a frame of FFT spectral data from PL to PS then poll for completion of the transfer before proceeding.

// Setup DMA from PL to PS memory using

- // AXI DMA's 'simple' transfer mode
- status = XAxiDma_SimpleTransfer(&axiDma, (u32)realspectrum,



```
REAL_FFT_LEN / 2 * sizeof(complex16), XAXIDMA_DEVICE_TO_DMA);
// Poll the AXI DMA core
do {
    status = XAxiDma_Busy(&axiDma, XAXIDMA_DEVICE_TO_DMA);
```

h. Before attempting to use the spectral data, the processor's data cache copy of the buffer must be invalidated to avoid use of stale data.

} while(status);

```
// Data cache must be invalidated for 'realspectrum' buffer after DMA
Xil_DCacheInvalidateRange((unsigned)realspectrum,
```

```
REAL_FFT_LEN / 2 * sizeof(complex16));
```

i. Push another set of stimulus data to the PL in order to start the accelerator processing the next frame:

```
// DMA another frame of data to PL
```

if (!XAxiDma_Busy(&axiDma, XAXIDMA_DMA_TO_DEVICE))

```
status = XAxiDma_SimpleTransfer(&axiDma, (u32)realdata,
```

```
REAL_FFT_LEN * sizeof(short), XAXIDMA_DMA_TO_DEVICE);
```

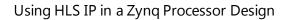
```
printf("\n\rFrame #%d received:\n\r");
```

j. Do something to verify that the accelerator is functioning. In this case, the spectral data is scanned for bins that contain significant energy. The expectation is to detect only energy in bins around the single tone (192) generated by the generate_waveform() function.

```
// Detect energy in spectral data above a set threshold
         for (j = 0; j < REAL_FFT_LEN / 2; j++) {</pre>
             // Convert the fixed point (s.15) values into floating point
values
                float real = (float)realspectrum[j].re / 32767.0f;
                float imag = (float)realspectrum[j].im / 32767.0f;
                float mag = sqrtf(real * real + imag * imag);
                if (mag > 0.00390625f) {
                 printf("Energy detected in bin %3d - ",j);
                  printf("{%8.5f, %8.5f}; mag = %8.5f\n\r", real, imag, mag);
                }
         }
         printf("End of frame.\n\r");
   }
   printf("***********\n\r");
   printf("* End of test *\n\r");
   printf("********\n\r\n\r");
```

www.xilinx.com

260 Send Feedback





#### return 0;

}

7. Define the helper function that generates the waveform data sets. This version simply fills a buffer with a single tone with 192 cycles per num_samples data window with values in a S.15 fixed point format.

```
void generate_waveform(short *signal_buf, int num_samples)
{
    const float cycles_per_win = 192.0f;
    const float phase = 0.0f;
    const float ampl = 0.9f;
    int i;
    for (i = 0; i < num_samples; i++) {
        float sample = ampl *
            cosf((i * 2 * M_PI * cycles_per_win / (float)num_samples) + phase);
        signal_buf[i] = (short)(32767.0f * sample);
    }
}</pre>
```

8. Define a routine to set up the and initialize the AXI DMA engine, wrapping all driver API calls that only need to be run once at startup.

```
int init_dma(XAxiDma *axiDmaPtr){
         XAxiDma Config *CfgPtr;
         int status;
         // Get pointer to DMA configuration
         CfgPtr = XAxiDma_LookupConfig(XPAR_AXIDMA_0_DEVICE_ID);
         if(!CfgPtr){
             print("Error looking for AXI DMA config\n\r");
             return XST_FAILURE;
         }
         // Initialize the DMA handle
         status = XAxiDma_CfgInitialize(axiDmaPtr,CfgPtr);
         if(status != XST_SUCCESS){
             print("Error initializing DMA\n\r");
             return XST FAILURE;
         }
         //check for scatter gather mode - this example must have simple mode only
         if(XAxiDma_HasSg(axiDmaPtr)){
             print("Error DMA configured in SG mode\n\r");
High-Level Synthesis
                                       www.xilinx.com
                                                                                    261
```



```
return XST_FAILURE;
}
//disable the interrupts
XAxiDma_IntrDisable(axiDmaPtr, XAXIDMA_IRQ_ALL_MASK,XAXIDMA_DEVICE_TO_DMA);
XAxiDma_IntrDisable(axiDmaPtr, XAXIDMA_IRQ_ALL_MASK,XAXIDMA_DMA_TO_DEVICE);
return XST_SUCCESS;
}
```

- 9. Save the modified source file. As soon as you save the file, SDK automatically attempts to rebuild the application executable. If the build fails, fix any outstanding issues.
- 10. Run the new application on the hardware and verify that it works as expected. Ensure that the FPGA is programmed and a terminal session is connected to the UART. Then **Launch on Hardware**, as done for the previous Hello World application code.

# Conclusion

In this tutorial, you learned:

- How to create Vivado HLS IP using a Tcl script.
- How to import a design into IP integrator (IPI) and connect it to a Zynq PS.
- How to create a software program to control the HLS IP and run this on a board.
- How to create a streaming system with HLS IP.





# Chapter 11 Using HLS IP in System Generator for DSP

## Overview

The RTL created by High-Level Synthesis can be packaged as IP and used inside System Generator for DSP (Vivado). This tutorial shows how this process is performed and demonstrates how the design can be used inside System Generator for DSP.

This tutorial consists of a single lab exercise.

#### Lab1 Description

Generate a design using Vivado HLS and package the design for use with System Generator for DSP. Then include the HLS IP into a System Generator for DSP design and execute an RTL simulation.

## **Tutorial Design Description**

You can download the tutorial design file from the Xilinx Website. Refer to the information in





#### **Obtaining the Tutorial** Designs.

This tutorial uses the design files in the tutorial directory Vivado_HLS_Tutorial\Using_IP_with_SysGen.

The sample design is a FIR filter that uses streaming interfaces modeled with the High-Level Synthesis hls::stream class. The design is fully pipelined at the function level. The optimization directives are embedded into the C code as pragmas.

# Lab 1: Package HLS IP for System Generator

This lab exercise integrates the High-Level Synthesis IP into System Generator for DSP.

*IMPORTANT:* The figures and commands in this tutorial assume the tutorial data directory *Vivado_HLS_Tutorial* is unzipped and placed in the location *C:\Vivado_HLS_Tutorial*.



If the tutorial data directory is unzipped to a different location, or on Linux systems, adjust the few pathnames referenced, to the location you have chosen to place the **Vivado_HLS_Tutorial** directory.

### Step 1: Create a Vivado HLS IP Block

Create two HLS blocks for the Vivado IP Catalog using the provided Tcl script. The script runs HLS C-synthesis, runs RTL co-simulation, and package the IP for the two HLS designs (hls_real2xfft and hls_xfft2real).

- 1. Open the Vivado HLS Command Prompt.
  - a. On Windows, go to Start > All Programs > Xilinx Design Tools > Vivado 2013.3 > Vivado HLS > Vivado HLS 2013.3 Command Prompt (Figure 267).
  - b. On Linux, open a new shell.



Figure 267: Vivado HLS Command Prompt

- 2. Using the command prompt window, change the directory to Vivado_HLS_Tutorial\Using_IP_with_SysGen\lab1 (Figure 268).
- 3. Type vivado_hls –f run_hls.tcl to create the HLS IP (Figure 268).



Vivado HLS 2013.2 Command Prompt	×
C:\Vivado_HLS_Tutorial>cd Using_IP_with_SysGen	^
C:\Vivado_HLS_Tutorial\Using_IP_with_SysGen>cd lab1	III
C:\Vivado_HLS_Tutorial\Using_IP_with_SysGen\lab1>vivavo_hls -f run_hls.tcl	*

Figure 268: Create the HLS Design

A key aspect of the Tcl script used to create this IP is the command **export_design –format sysgen**. This command creates an IP package for System Generator. When the script completes there is a Vivado HLS project directories fir_prj, which contains the HLS IP, including the IP package for use in a System Generator for DSP design.

The remainder of this tutorial exercise shows how to integrate the Vivado HLS IP block into a System Generator design.

## Step 2: Open the System Generator Project

- 1. Open System Generator for DSP.
  - a. On Windows use the desktop icon (Figure 269).
  - b. On Linux, open a new shell and type **sysgen**.



Figure 269: System Generator 2013.3 Icon

2. When Matlab invokes, click the **Open** toolbar button (Figure 270).

HOME	PLOTS	APPS		
New New Script 👻	Open	Import Save	<ul> <li>New Variable</li> <li>Open Variable ▼</li> <li>Clear Workspace ▼</li> </ul>	Analyze Code
	🛅 Open			Ctrl+O

Figure 270: Open the System Generator Design

3. Navigate to the tutorial directory Vivado_HLS_Tutorial\Using_IP_with_SysGen\lab1 and select the file fir_sysgen.mdl (Figure 271).





OSDisk (C:) ► Vivac	lo_HLS_Tutorial	۲	Using_IP_wit
New folder			
Name	A		
) fir_prj fir.cpp fir.h			
🔰 fir_sysgen.md	I		
ir_test.cpp			

Figure 271: Select File fir_sysgen.mdl

When System Generator invokes, all blocks and ports *except the HLS IP* are already instantiated in the design.

4. Right-click in the canvas and select Xilinx BlockAdd, as shown in Figure 272.

🍋 1	fir_sysgen															
File	Edit View	Display	Diagram	Sim	ulation	Analysis	Cod	e Tools	s Help							
▶.	- 8 8 <	≎ ⇒ {		) - (	•			• •	100		Normal		·	]	•	
fir_	sysgen															
۲	hir_sysgen 🔁															
Đ,	_															
	5															
K N K	System															
⇒	Generator				Xilinx B	lockAdd										
AE					Xilinx B	lockCon	nect					1				
					Xilinx T	ools					+	H				
		→ In ap_n			Xilinx V	iew Sign	als					L				
	Constant	- <u>-</u>			Explore	e						F	-			
		→ In 🐔		5	Can't U	ndo			(	Ctrl+Z		L.,				
	Pulse Generator	ap_st	art	¢	Can't Re					Ctrl+Y						
	<u> </u>	→ In 🐔										V_read	<b></b>			
	Pulse Generator1	input_val_	V_dout		Paste		<b>.</b> .		C	Ctrl+V		1				
		→ In &		G	Paste D	uplicate	Inport					_V_dir	, <b>→</b>			
	Constant1	input_val_V	_empy_n		Select /	All			C	Ctrl+A		L				
					Find Re	ferenced	l Variab	les				_V_wr	ite			
		→ In [®] output_val	V full n		Most Fr	requently	Used E	Blocks			+					
	Constant2	oocput_var_	<u>, , , , , , , , , , , , , , , , , , , </u>		Remov	e Highlig	ihtina		(	Ctrl+Sh	ift+H	LV_hv				
				*		e Diagran	-			Ctrl+D		L		Scop	e	
				3												

Figure 272: Adding an new Block

- 5. Type "hls" in the Add Block field (Figure 273).
- 6. Select Vivado HLS.







Figure 273: Selecting a Vivado HLS IP Block

- 7. Double-click the Vivado HLS block to open the Vivado HLS dialog box.
- 8. Navigate to the fir_prj project and select the solution1 folder (Figure 274).



**IMPORTANT:** System Generator for DSP uses the location of the solution folder to identify the IP.

9. Click **OK** to load the IP block.

😸 Vivado HLS (Xilinx High Level Sy 👝 🔳 💌						
This block allows including C,C++ and SystemC source files in System Generator for DSP designs.						
Solution with_SysGen/lab1/fir_prj/solution1/' Browse Refresh Edit						
Use C simulation model if available						
Display signal types						
Output Sample Times Simulink system period 🔹						
OK Cancel Help Apply						

Figure 274: Selecting the FIR IP Block

The FIR IP block is instantiated into the design.

10. Connect the design I/O ports to the ports on the FIR IP block, as shown in 275Figure 275.





✤ fir_sysgen File Edit View Display D	iagram Simulation Analysis Code Tools Help	
		• Ø •
fir_sysgen		
©		
0       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I       I	tt tinput_val_V_dout tt output_val_V_write output_val_V_din output_val_V_din output_val_V_din output_val_V_din output_val_V_din output_val_V_din output_val_V_din output_val_V_din output_val_V_din output_val_V_din output_val_V_din output_val_V_din output_val_V_din output_val_V_din output_val_V_din output_val_V_din output_val_V_din output_val_V_din output_val_V_din output_val_V_din output_val_V_din output_val_V_din output_val_V_din output_val_V_din output_val_V_din output_val_V_din output_val_V_din output_val_V_din output_val_V_din output_val_V_din output_val_V_din output_val_V_din output_val_V_din output_val_V_din output_val_V_din output_val_V_din output_val_V_din output_val_V_write Out Out	→ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
»		
Ready	94%	ode45 🦽

Figure 275: Design with All Connections

- 11. Ensure the simulation stop time says 300 (Figure 275).
- 12. Click the **Run** button on the toolbar to execute simulation.
- 13. Double-click the **Scope** block to view the simulation waveforms.

# Conclusion

In this tutorial, you learned:

- How to create Vivado HLS IP using a Tcl script.
- How to import an HLS design as IP into System Generator for DSP.

