Vivado Design Suite Tutorial

High-Level Synthesis

UG871 (v2015.3) September 30, 2015

XILINX
ALL PROGRAMMABLE™

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
9/30/2015	2015.3	Updated the steps in the Lab 2: Viewing Trace Files in Vivado section. Updated the figures in the following chapters: • Chapter 2, High-Level Synthesis Introductory Tutorial • Chapter 3, C Validation • Chapter 4, Interface Synthesis • Chapter 5, Arbitrary Precision Types • Chapter 6, Design Analysis • Chapter 7, Design Optimization • Chapter 8, RTL Verification • Chapter 9, Using HLS IP in IP Integrator • Chapter 11, Using HLS IP in System Generator for DSP
7/21/2015	2015.2	Updated Chapter 1, Tutorial Description, Chapter 2, High-Level Synthesis Introductory Tutorial, Chapter 3, C Validation, Chapter 4, Interface Synthesis, Chapter 5, Arbitrary Precision Types, Chapter 6, Design Analysis, Chapter 7, Design Optimization, Chapter 8, RTL Verification, Chapter 9, Using HLS IP in IP Integrator, and Chapter 11, Using HLS IP in System Generator for DSP
4/01/2015	2015.1	Updated Chapter 1, Tutorial Description, Chapter 2, High-Level Synthesis Introductory Tutorial, Chapter 3, C Validation, Chapter 4, Interface Synthesis, Chapter 5, Arbitrary Precision Types, Chapter 6, Design Analysis, Chapter 7, Design Optimization, Chapter 8, RTL Verification, Chapter 9, Using HLS IP in IP Integrator, and Chapter 11, Using HLS IP in System Generator for DSP.

Table of Contents

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. 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.

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.

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, AXI4-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 Design Optimization 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 automatically verify 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 AP SoC Design

In addition to using an HLS IP block in a Zynq®-7000 APSoC 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 2015.3 release or later is installed.

Hardware Requirements

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

Locating the Tutorial Design Files

As shown in Figure 1-1, 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 at:* Reference Design Files

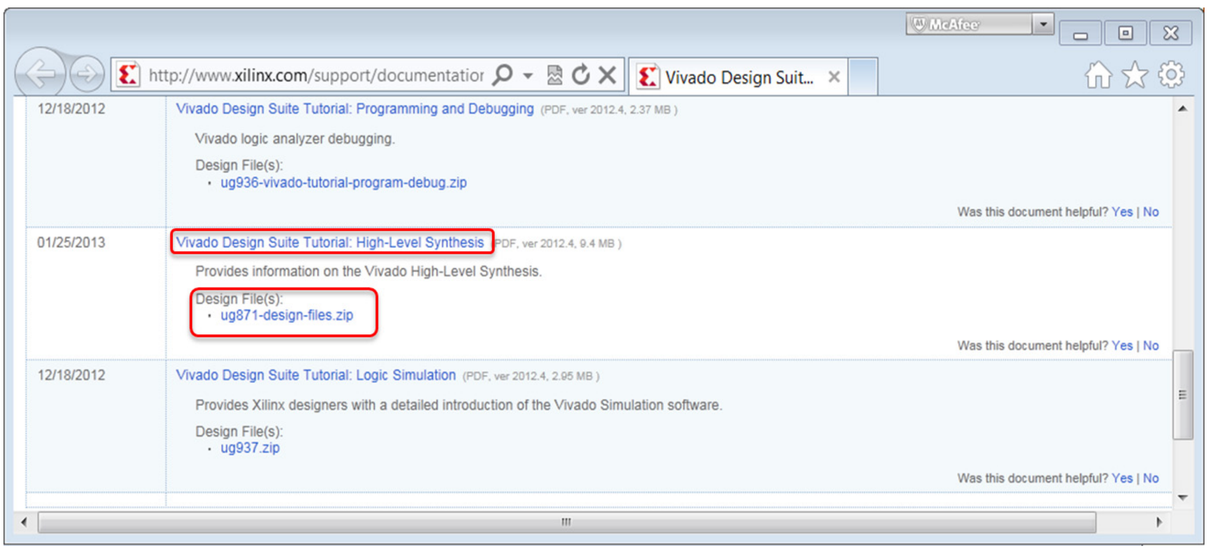


Figure 1-1: **High-Level Synthesis Tutorial Design Files**

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.*

High-Level Synthesis Introduction

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 Description

Explains how to set up a High-Level Synthesis (HLS) project and perform all the major steps in the HLS design flow:

- Validate the C code.
- Create and synthesize a solution.
- Verify the RTL and package the IP.

Lab 2 Description

Demonstrates how to use the Tcl interface.

Lab 3 Description

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, see Locating the Tutorial Design Files.

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 goal for this FIR design project is:

- 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.

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 2015.3** desktop icon.
   - On Linux systems, type `vivado_hls` at the command prompt.

Figure 2-1: **The Vivado HLS Desktop Icon**

TIP: *You can also open Vivado HLS using the Windows menu* ***Start > All Programs > Xilinx Design Tools > Vivado 2015.3 > Vivado HLS > Vivado HLS 2015.3****.*

Vivado HLS opens with the Welcome Screen as shown below. If any projects were previously opened, they are shown in the Recent Project pane, otherwise this window is not shown in the Welcome screen.

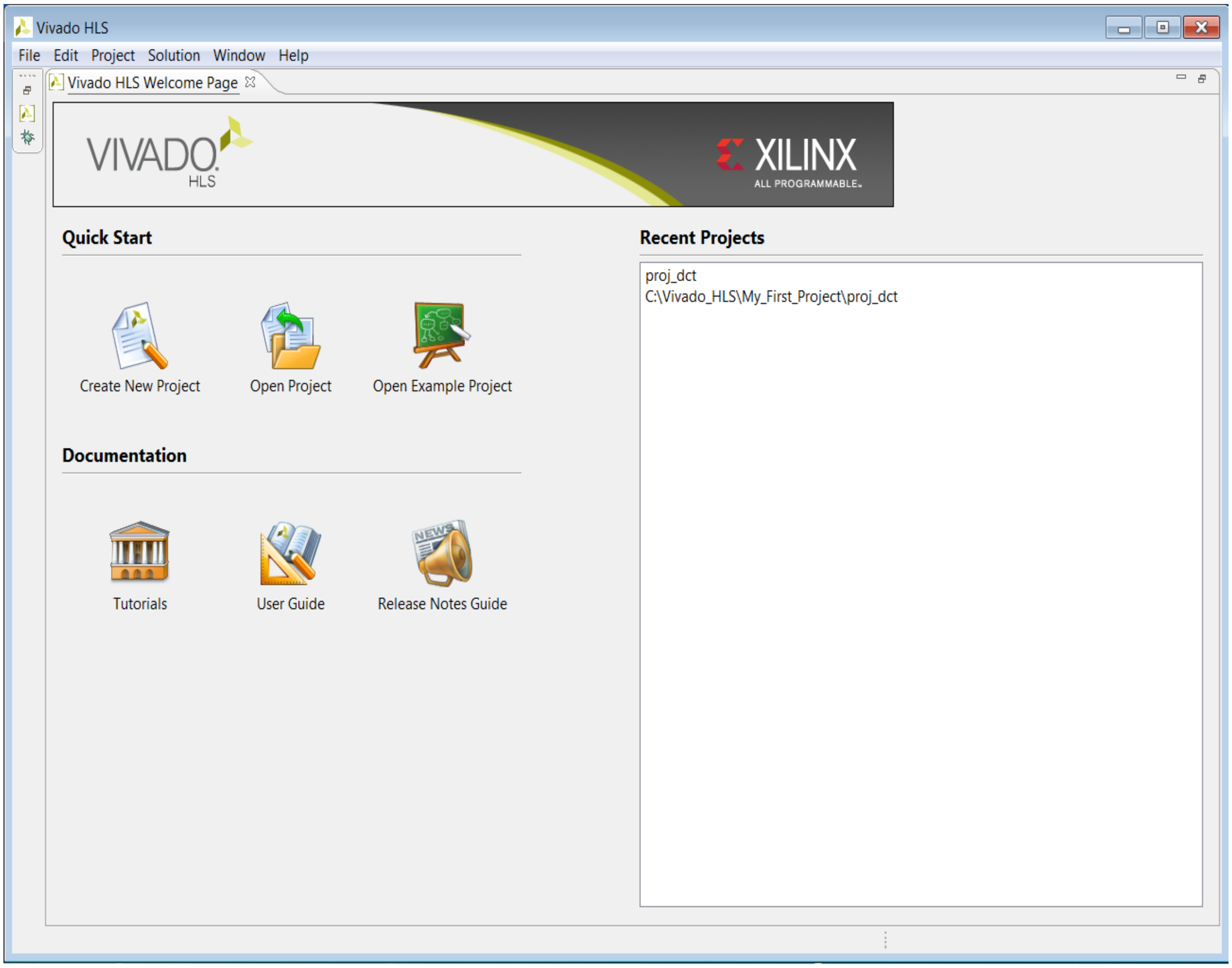


Figure 2-2: **The Vivado Welcome Page**

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

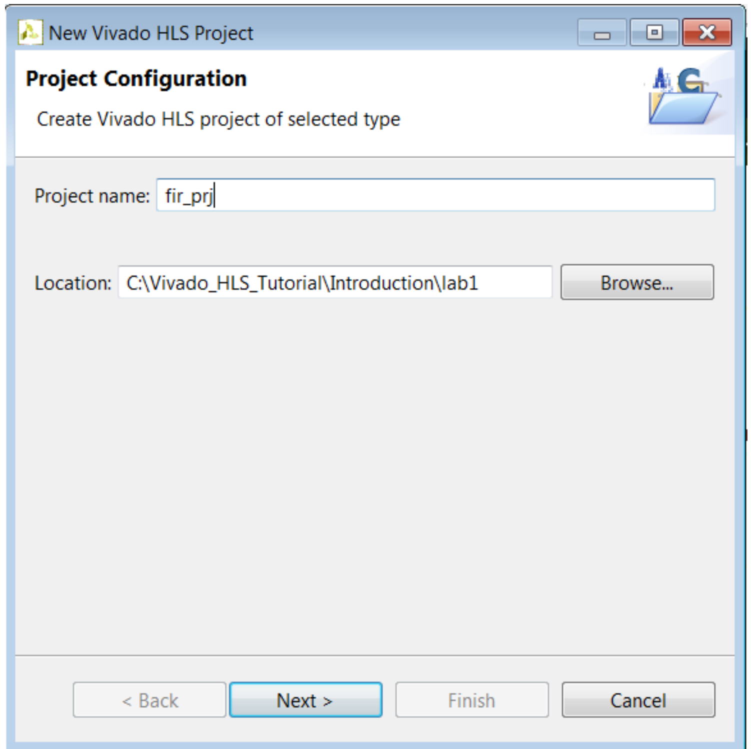


Figure 2-3: **Project Configuration**

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**.

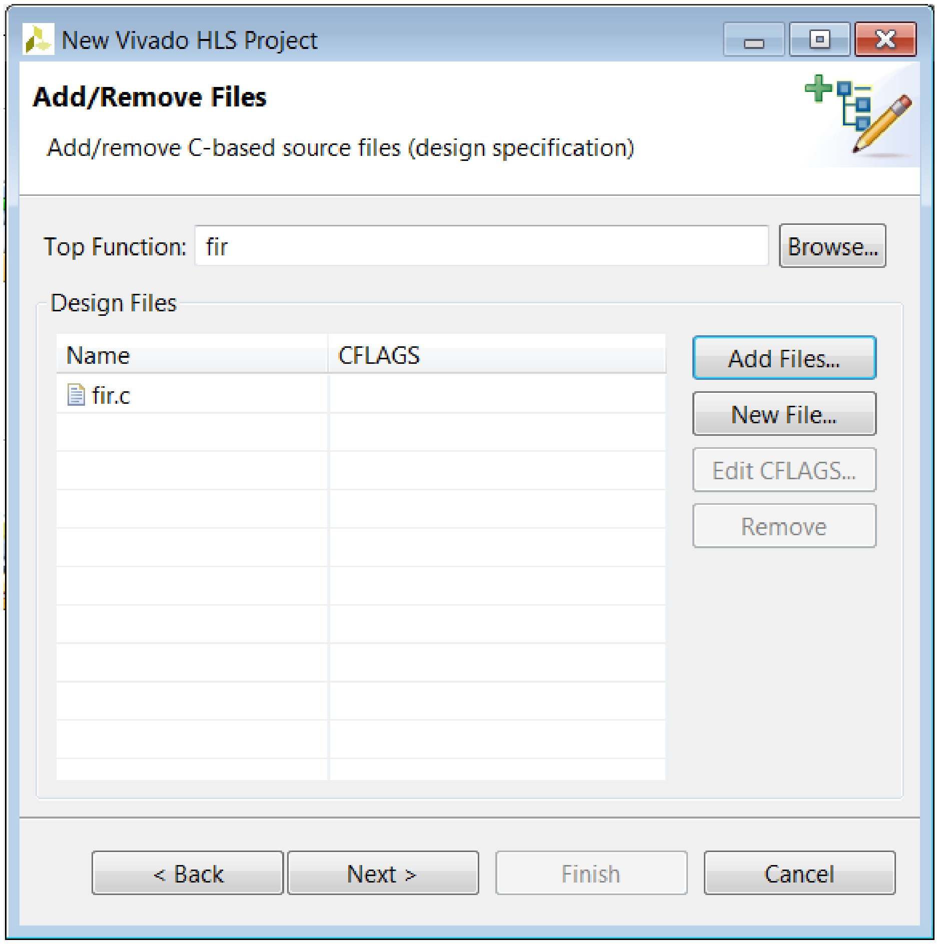


Figure 2-4: **Project Design Files**

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 2-5 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.

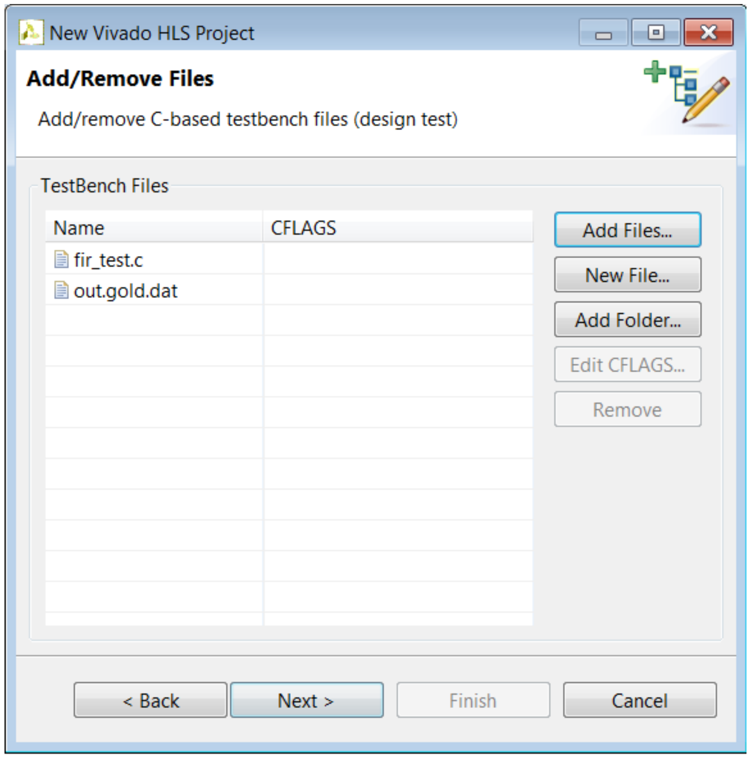


Figure 2-5: **Test Bench Files**

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 subdirectories 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 due to an inability to find the data files.

The Solution Configuration window (shown in Figure 2-6) 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.

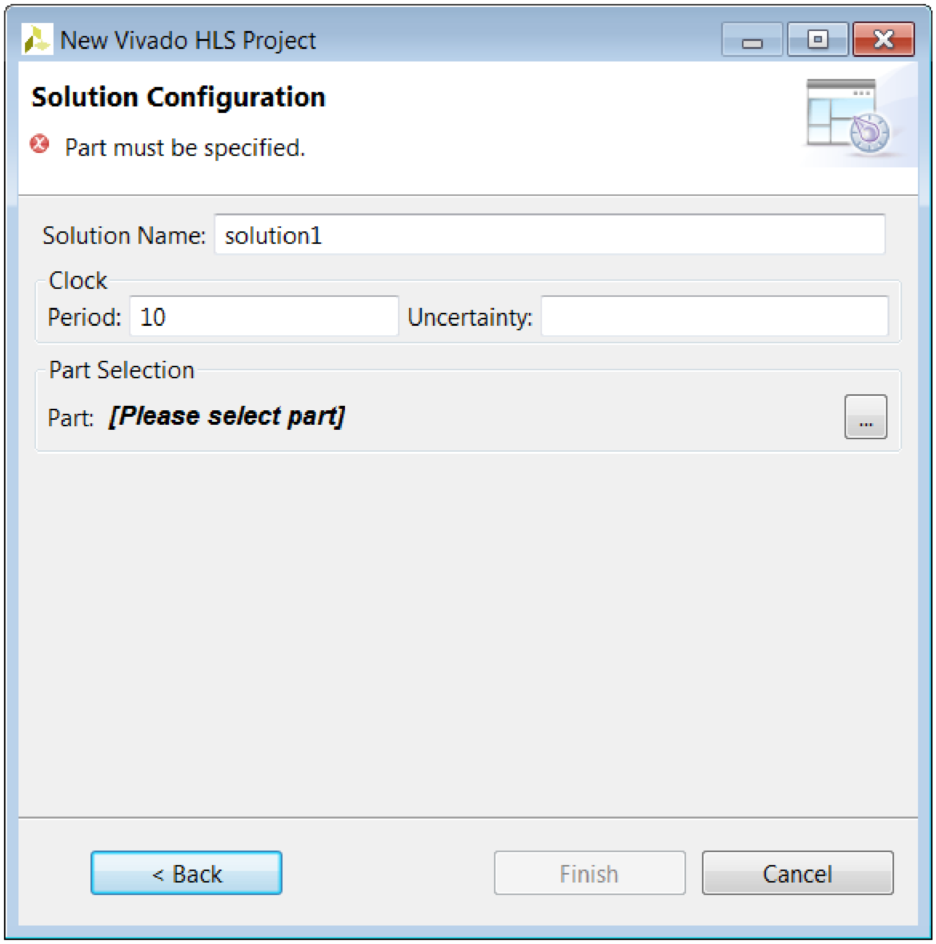


Figure 2-6: **Solution Configuration**

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 drop-down 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: **All**
10. Select **xc7k160tfbg484-2**.
11. Click **OK**.

In the Solution Configuration dialog box (shown in Figure 2-6, above), the selected part name now appears under the Part Selection heading.

12. Click **Finish** to open the Vivado HLS project, as shown in Figure 2-7.

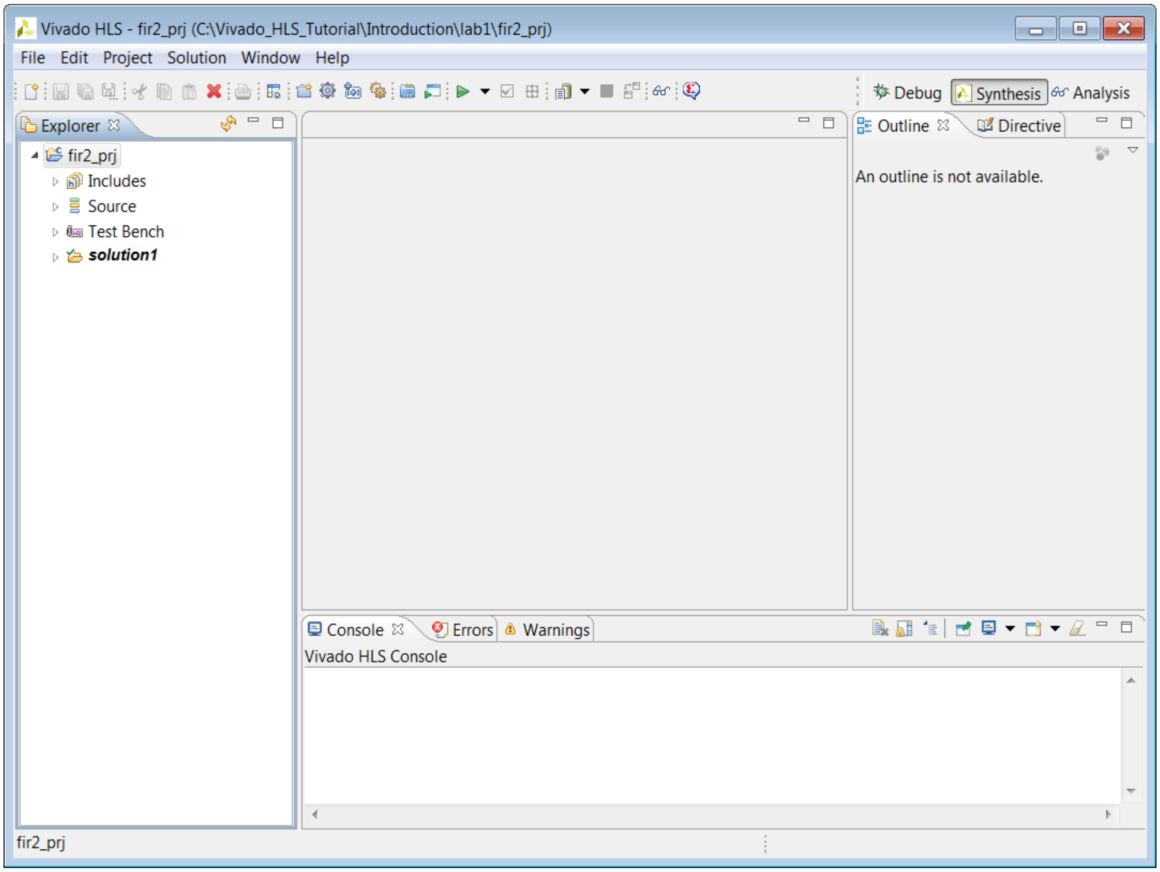


Figure 2-7: **Vivado HLS Project**

- The project name appears on the top line of the Explorer window.
- A Vivado HLS project arranges information 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.
- 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 2-8 shows an overview of the regions, and describes each below.

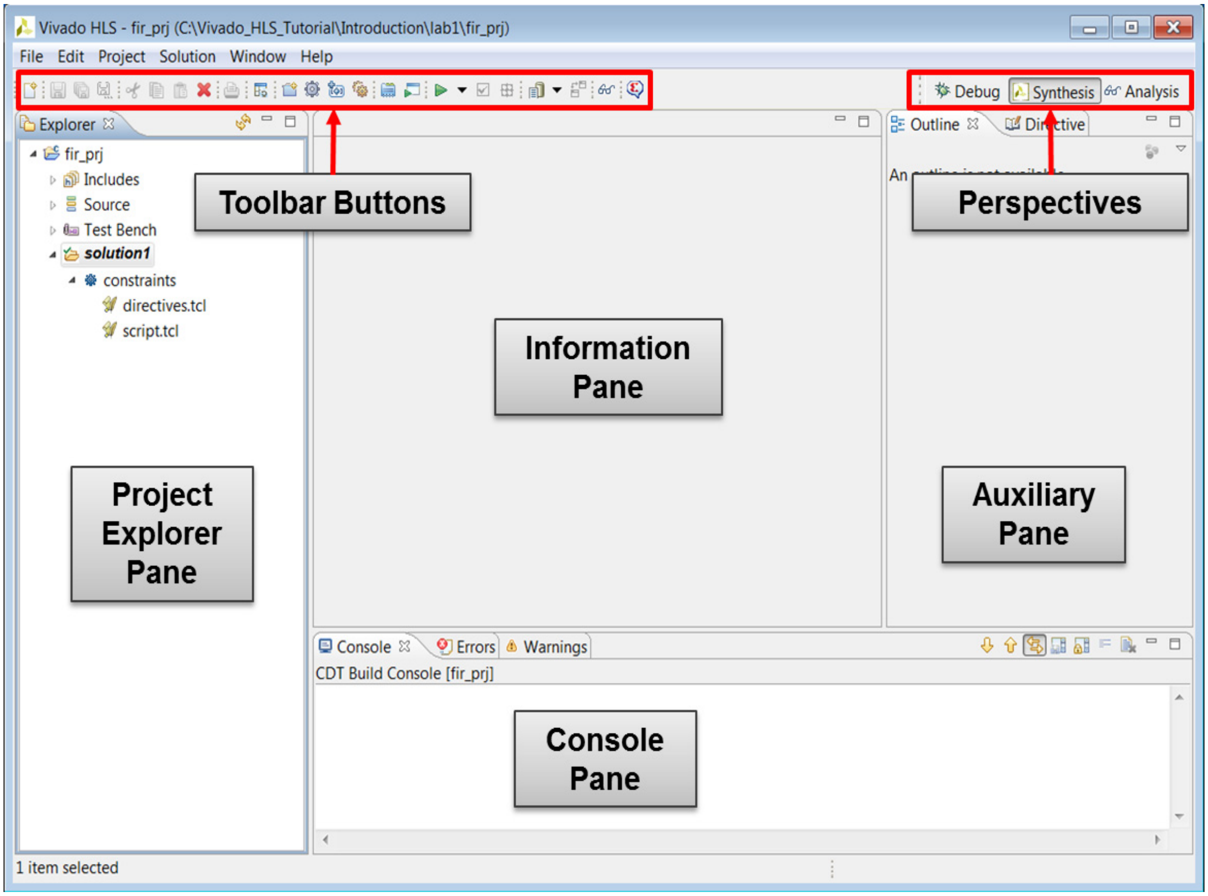


Figure 2-8: **Vivado HLS Graphical User Interface**

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 tool tip opens, explaining the function. Each button also has an associated menu item available from the pull-down 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 disables 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 2-9 shows the result of these actions

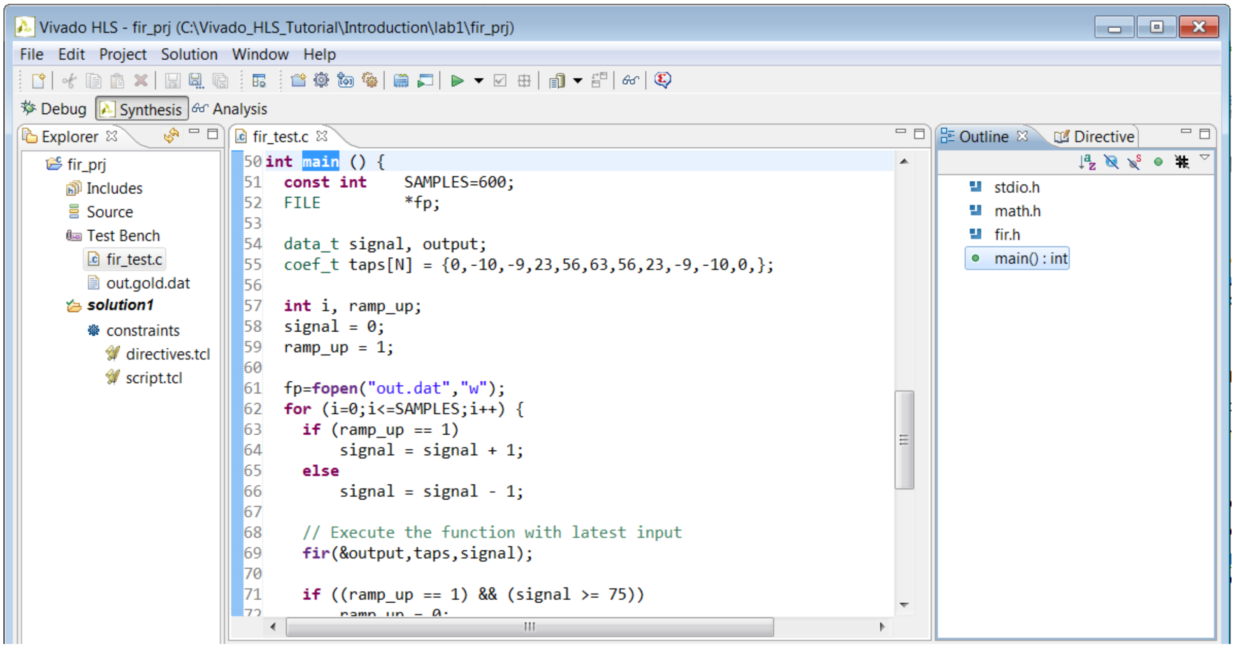


Figure 2-9: **Reviewing the Test Bench Code**

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.

If the test bench has the previously described self-checking characteristics, the RTL results are automatically checked during RTL verification. Vivado HLS re-uses the test bench during RTL verification and 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. 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 2-10) confirms the simulation executed successfully.

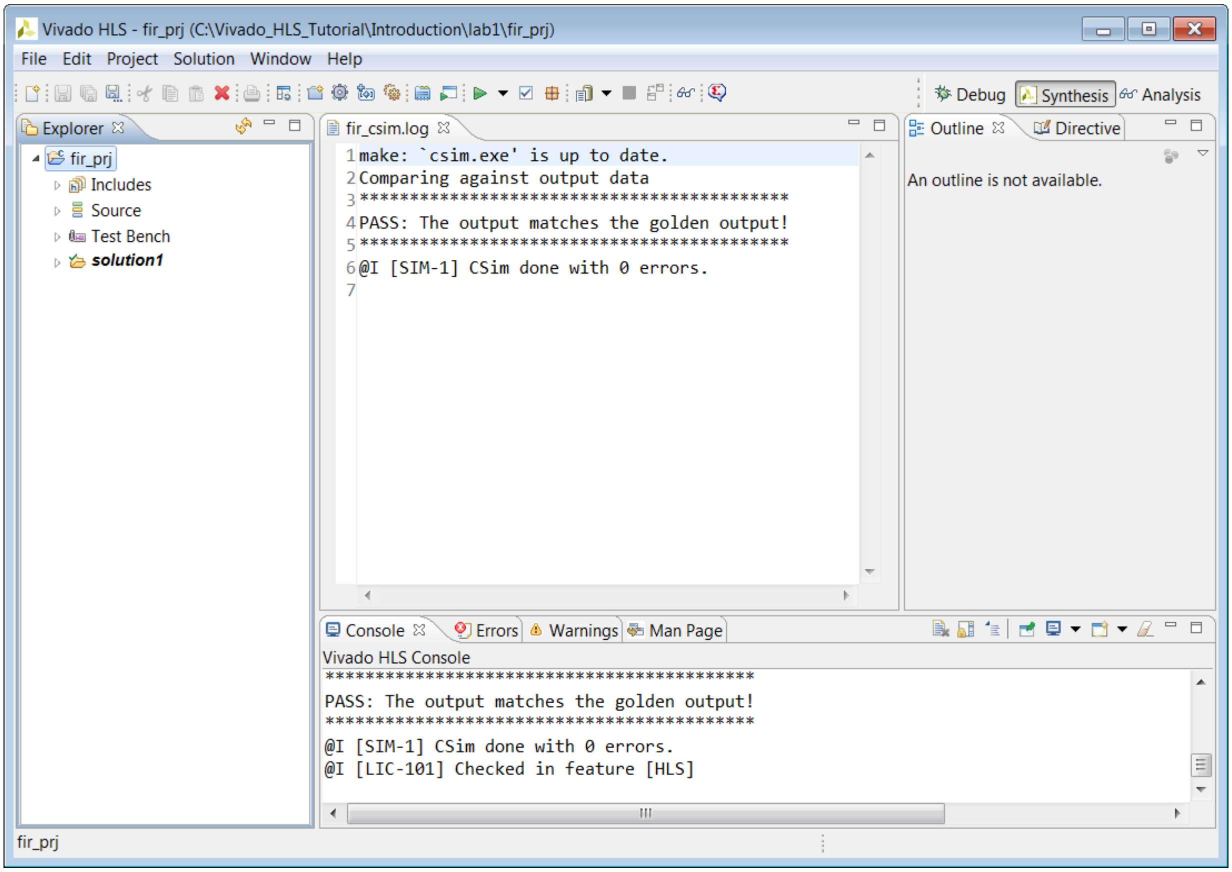


Figure 2-10: **Results of C Simulation**

TIP: *If the C simulation ever fails, 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 Estimates** in the Outline tab (Figure 2-11).
3. In the Detail section of the Performance Estimates, expand the **Loop** view.

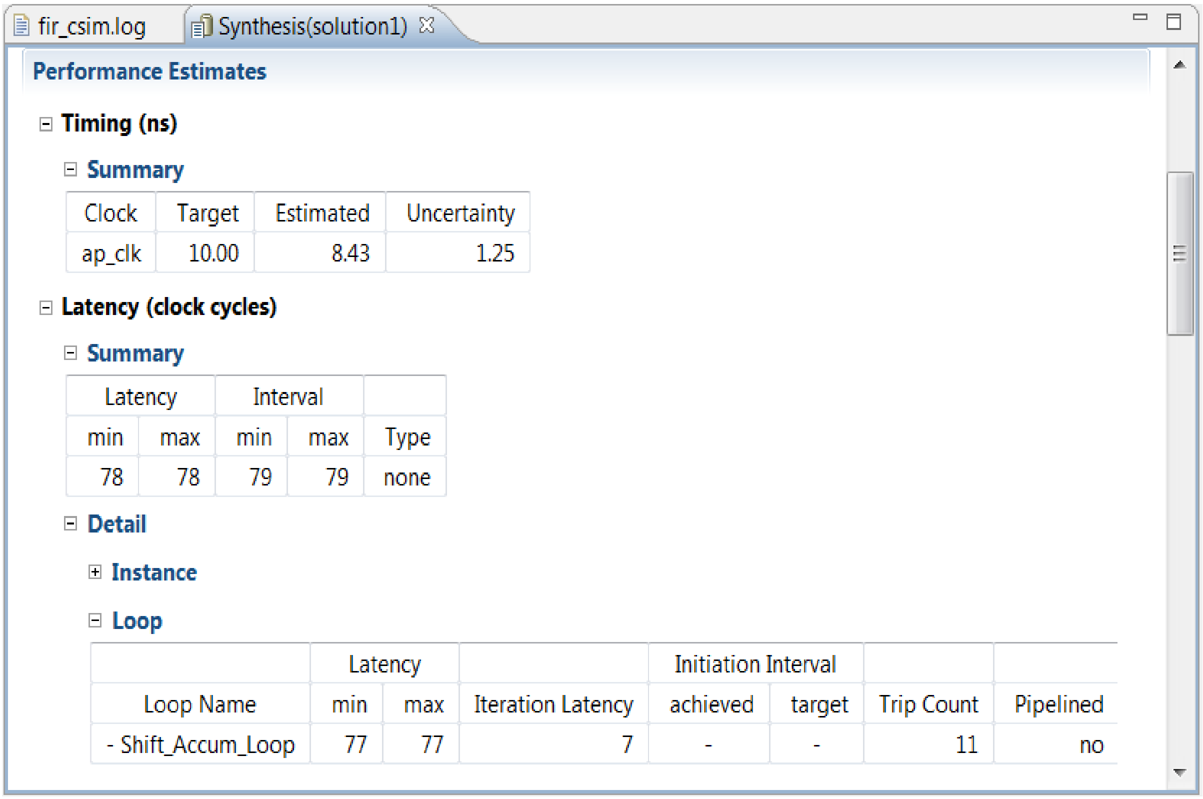


Figure 2-11: **Performance Estimates**

In the Performance Estimates pane, shown in Figure 2-12, you can see that the clock period is set to 10 ns. Vivado HLS targets a clock period of Clock Target minus Clock Uncertainty (10.00-1.25 = 8.75ns in this example).

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

The estimated clock period (worst-case delay) is 8.43 ns, which meets the 8.75 ns timing requirement.

In the Summary section, you can see:

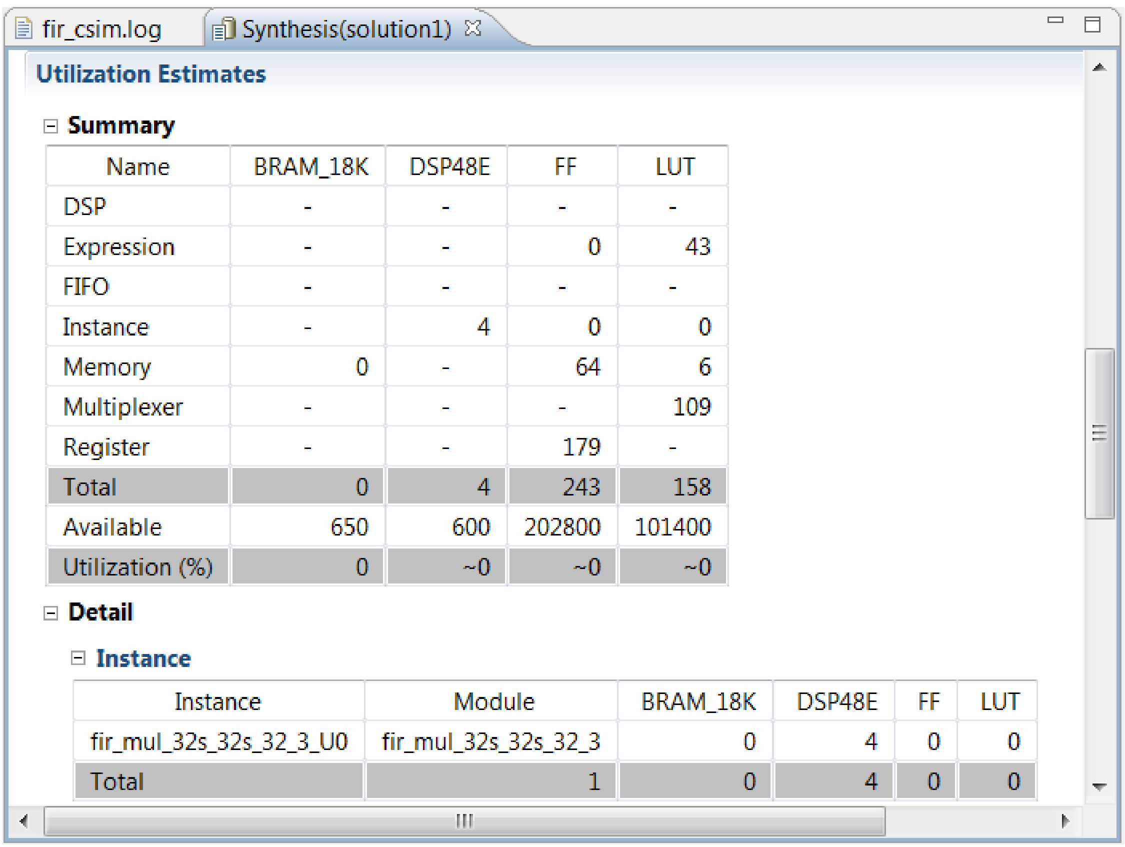
- The design has a latency of 78-clock cycles: it takes 78 clocks to output the results.
- The interval is 79 clock cycles: the next set of inputs is read after 79 clocks. This is one cycle after the final output is written. This indicates the design is not pipelined. The

next execution of this function (or next transaction) can only start when the current transaction completes.

The Detail section shows:

- There are no sub-blocks in this design. Expanding the Instance section shows no submodules in the hierarchy.
- All the latency 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 7 clock cycles (Iteration Latency), for a total of 77 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).

4. In the Outline tab, click **Utilization Estimates** (Figure 2-12).



Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	43
FIFO	-	-	-	-
Instance	-	4	0	0
Memory	0	-	64	6
Multiplexer	-	-	-	109
Register	-	-	179	-
Total	0	4	243	158
Available	650	600	202800	101400
Utilization (%)	0	~0	~0	~0

Instance	Module	BRAM_18K	DSP48E	FF	LUT
fir_mul_32s_32s_32_3_U0	fir_mul_32s_32s_32_3	0	4	0	0
Total	1	0	4	0	0

Figure 2-12: **Utilization Estimates**

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

The design uses a single memory implemented as LUTRAM (since it contains less than 1024 elements), 4 DSP48s, and approximately 200 flip-flops and LUTs. At this stage, the device resource numbers are estimates.

- The resource utilization numbers are estimates because 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. This is because the data is a C integer type, which is 32-bit. It requires more than one DSP48 to multiply 32-bit data values.
- The multiplier instance shown in the Instance view accounts for all the DSP48s.
- The multiplier is a pipelined multiplier. It appears in the Instance section indicating it is a sub-block. Standard combinational multipliers have no hierarchy and are listed in the Expressions section (indicating a component at this level of hierarchy).

In **HLS**: Lab 3: Using Solutions for Design Optimization, you optimize this design.

6. In the Outline tab, click **Interface** (Figure 2-13).

fir_csim.log | Synthesis(solution1)

Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	fir	return value
ap_rst	in	1	ap_ctrl_hs	fir	return value
ap_start	in	1	ap_ctrl_hs	fir	return value
ap_done	out	1	ap_ctrl_hs	fir	return value
ap_idle	out	1	ap_ctrl_hs	fir	return value
ap_ready	out	1	ap_ctrl_hs	fir	return value
y	out	32	ap_vld	y	pointer
y_ap_vld	out	1	ap_vld	y	pointer
c_address0	out	4	ap_memory	c	array
c_ce0	out	1	ap_memory	c	array
c_q0	in	32	ap_memory	c	array
x	in	32	ap_none	x	scalar

Figure 2-13: **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 indicated by Source Object fir. 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 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 C/RTL Co-simulation dialog box to execute the RTL simulation.

The default option for RTL co-simulation is to perform the simulation using the Vivado simulator and Verilog RTL. To perform the verification using a different simulator, VHDL, or SystemC RTL use the options in the C/RTL Co-simulation dialog box.

When RTL co-simulation completes, the report opens automatically in the Information pane, and the Console displays the message shown in Figure 2-14. This is the same message produced at the end of C simulation.

- The C test bench generates input vectors for the RTL design.
- 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 Vivado HLS indicates that simulation passes if the test bench returns a value of 0. It is the value of the return variable in the test bench, and this alone, which indicates if the simulation was successful. It is important that the test bench returns a value of 0 only if the results are correct.

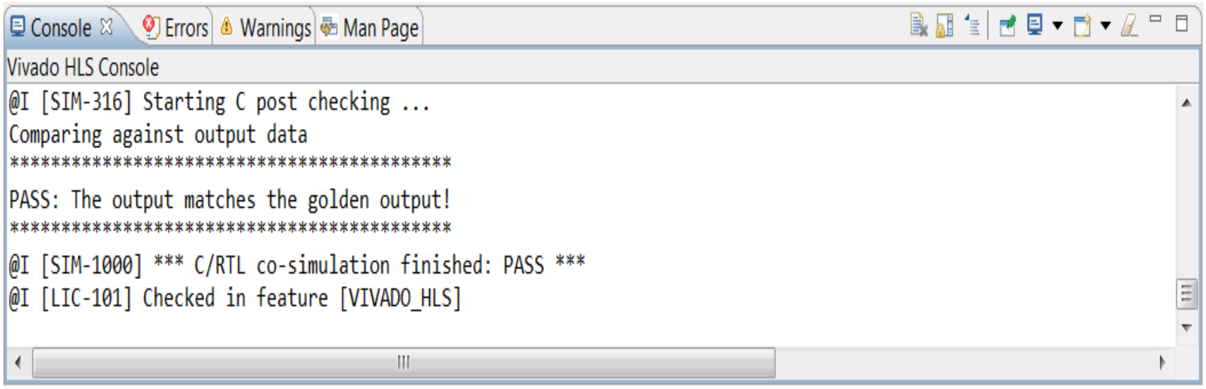


Figure 2-14: **RTL Verification Results**

The Chapter 7, RTL Verification tutorial 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 Vivado Design Suite.

1. Click the **Export RTL** toolbar button or use the menu **Solution > Export RTL**.
2. Ensure the Format Selection drop-down 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, a Synthesized Checkpoint format for Vivado, 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 2-15).

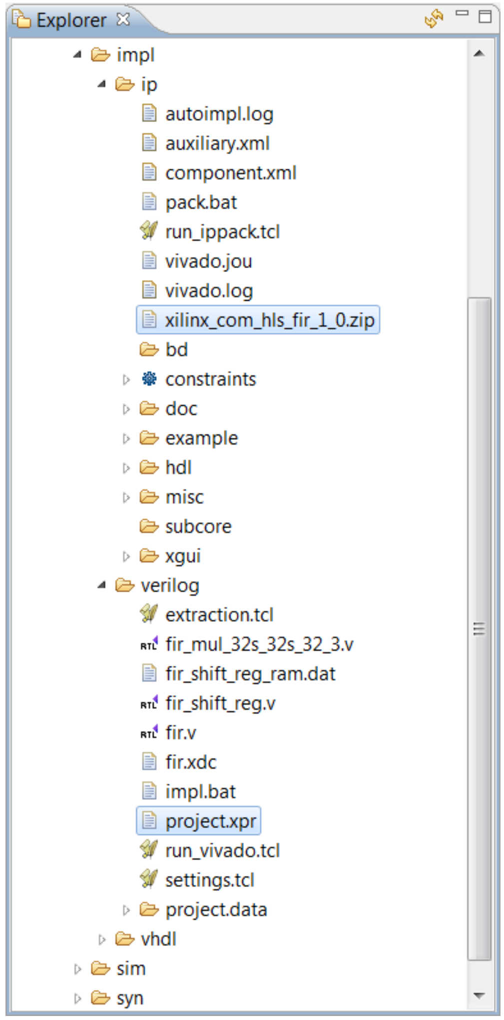


Figure 2-15: **RTL Verification Results**

Note: In Figure 2-15, 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: *This Vivado project is provided only as a convenient way to analyze the design inside the Vivado IDE. This project should not be used to implement your design: there are no top-level IO buffers in this project. The recommended methodology for using the output of Vivado HLS in your own design is to incorporate the IP package, or one of the other output formats, into your own Vivado project. Additional tutorials in this guide demonstrate how to use the Vivado HLS output as IP in your project.*

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.

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.

Step 1: Create a Tcl file

1. Open the Vivado HLS Command Prompt.
   - On Windows, use **Start > All Programs > Xilinx Design Tools > Vivado 2015.3 > Vivado HLS > Vivado HLS 2015.3** Command Prompt (Figure 2-16).
   - On Linux, open a new shell.

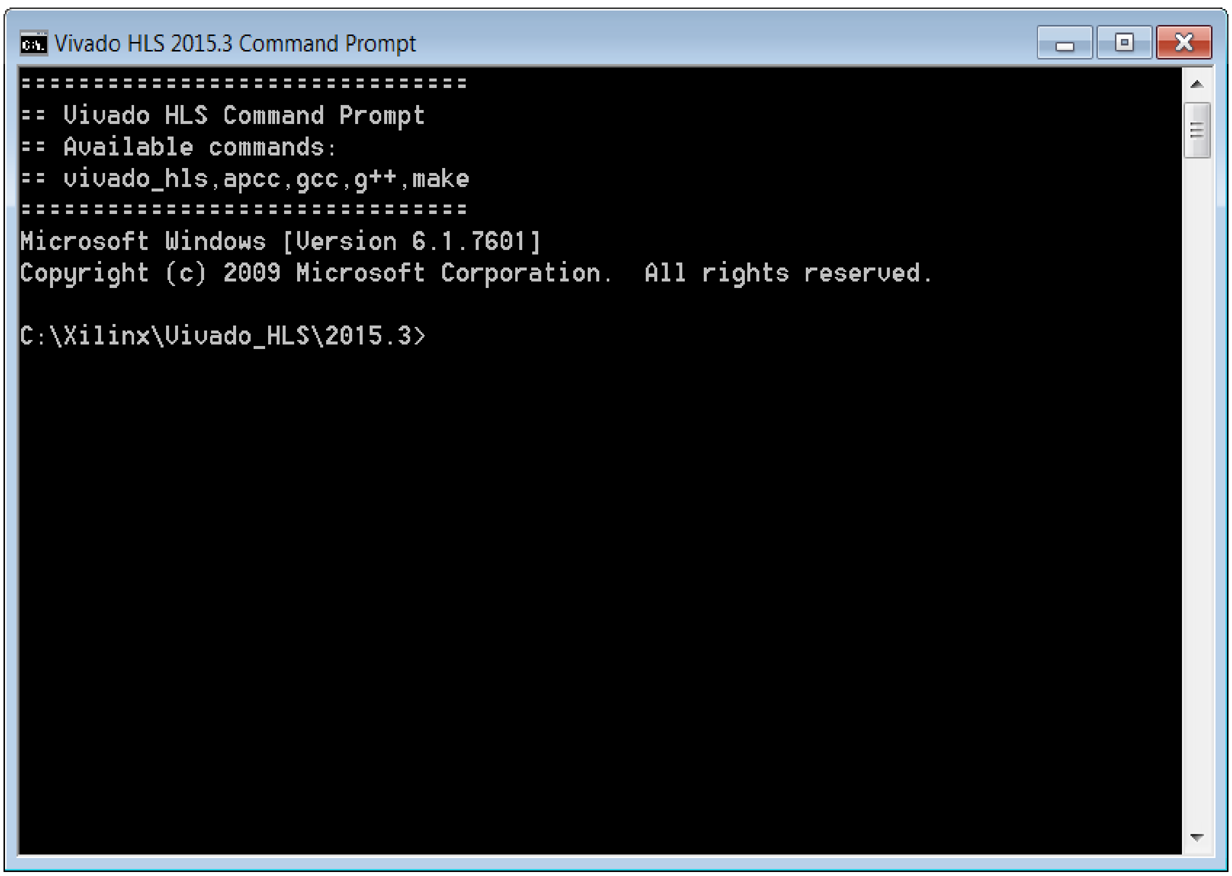


Figure 2-16: **The Vivado HLS Command Prompt**

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 2-17).

2. 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.

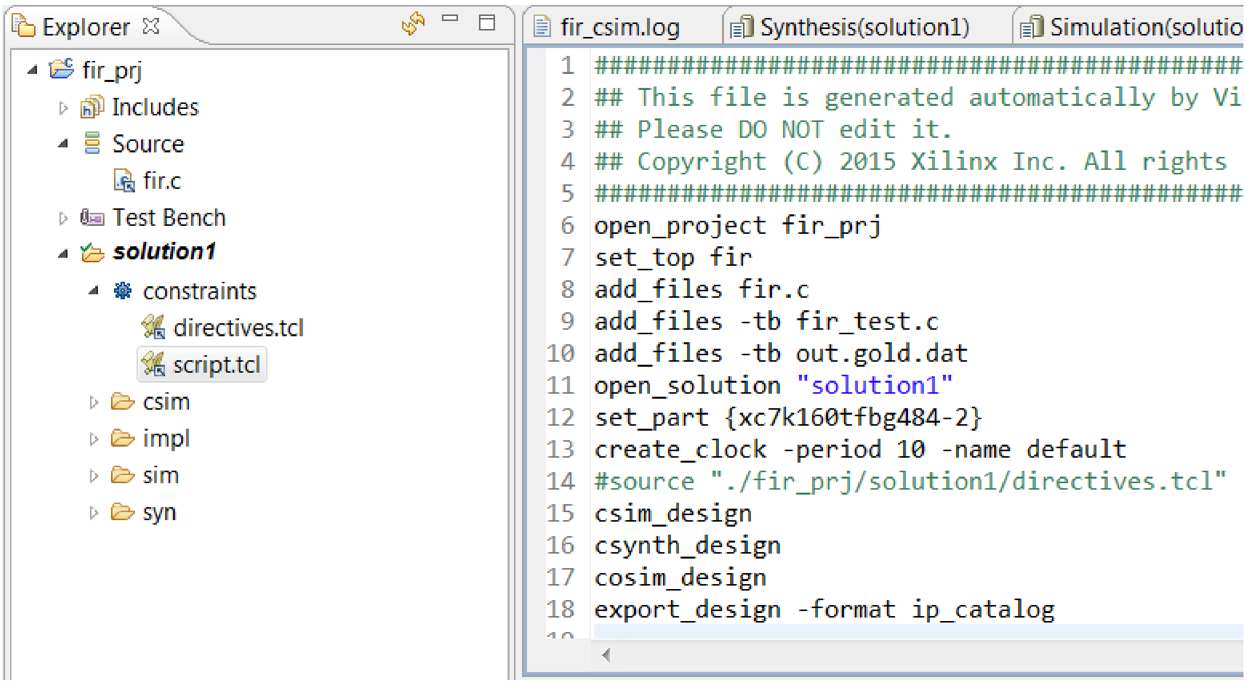


Figure 2-17: **The Vivado HLS Project Tcl Files**

- 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.

3. Close the Vivado HLS GUI from Lab 1. This is project no longer needed.
4. In the Vivado HLS Command Prompt, use the following commands (also shown in Figure 2-18) 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.

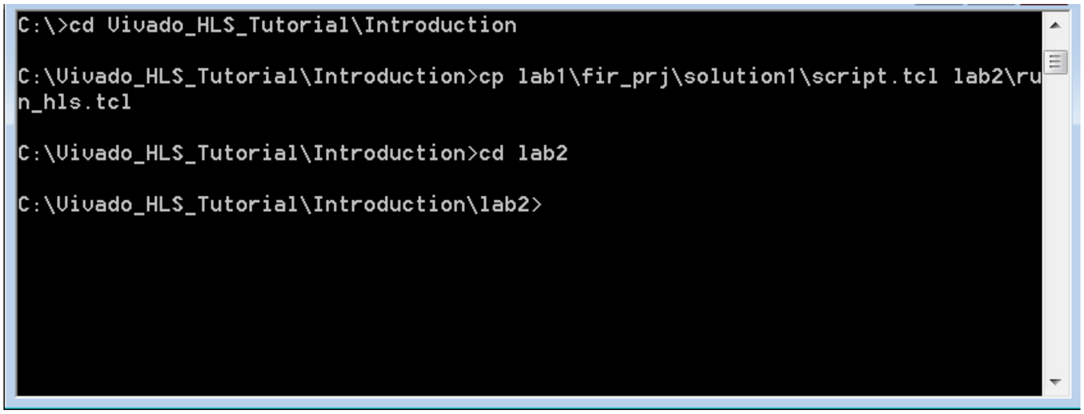


Figure 2-18: **Copying the Lab 1 Tcl file to Lab 2**

5. 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 2-19.

   a. 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.

   b. 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.

   c. Leave the source command commented. If the previous project contains any directives you wish to re-use, you can copy the directives directly into this file.

   d. Add the **`exit`** command.

   e. Save the file.

run_hls.tcl

```
############################################################
## This file is generated automatically by Vivado HLS.
## Please DO NOT edit it.
## Copyright (C) 2015 Xilinx Inc. All rights reserved.
############################################################

# Reset the project with the -rset option
open_project -reset fir_prj
set_top fir
add_files fir.c
add_files -tb fir_test.c
add_files -tb out.gold.dat

# Reset the project with the -rset option
open_solution -reset "solution1"
set_part {xc7k160tfbg484-2}
create_clock -period 10 -name default

# Leave the previous directives commented out
#source "./fir_prj/solution1/directives.tcl"

csim_design
csynth_design
cosim_design
export_design -format ip_catalog

# Exit vivado HLS
exit
```

Figure 2-19: **Updated run_hls.tcl file for Lab 2**

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

6. 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 only output from synthesis. Additional processing is performed by Vivado HLS during* `export_design` *before you can use this RTL in other design tools.*

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.
   - On Windows, use **Start > All Programs > Xilinx Design Tools > Vivado 2015.3 > Vivado HLS > Vivado HLS 2015.3** Command Prompt
   - 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 2-20, with the synthesis for solution1 already complete.

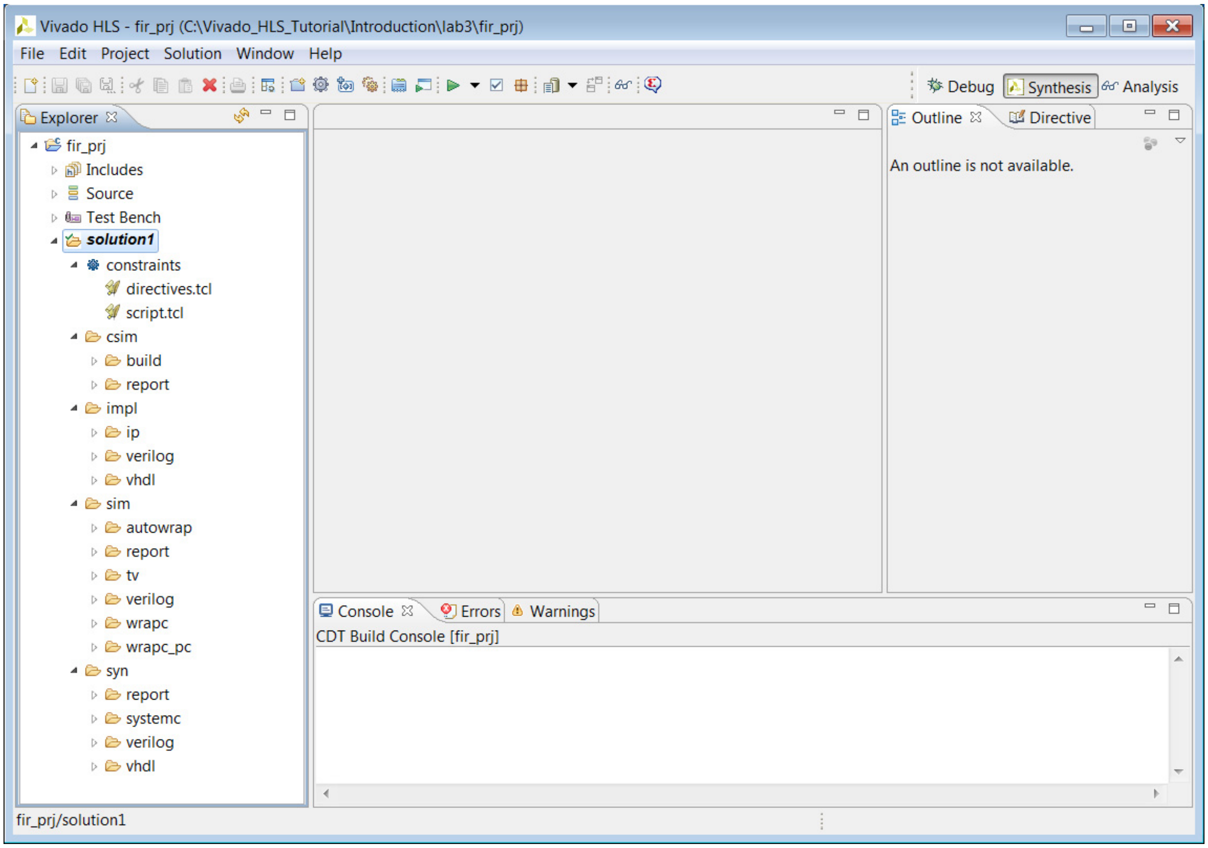


Figure 2-20: **Introduction Lab 3 Initial Solution**

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 2-13), 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 doing so creates a design with a higher throughput. If a single-port is required, 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, because the default implementation is what is required, but if it is a requirement, it is a good practice to specify it.

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 sets it as the default solution. To confirm you can verify that the current active solution2 is highlighted in bold in the Explorer pane.

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 (as shown in Figure 2-21).
5. Double-click `fir.c` to open the file in the Information pane.
6. Activate the **Directive** 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.

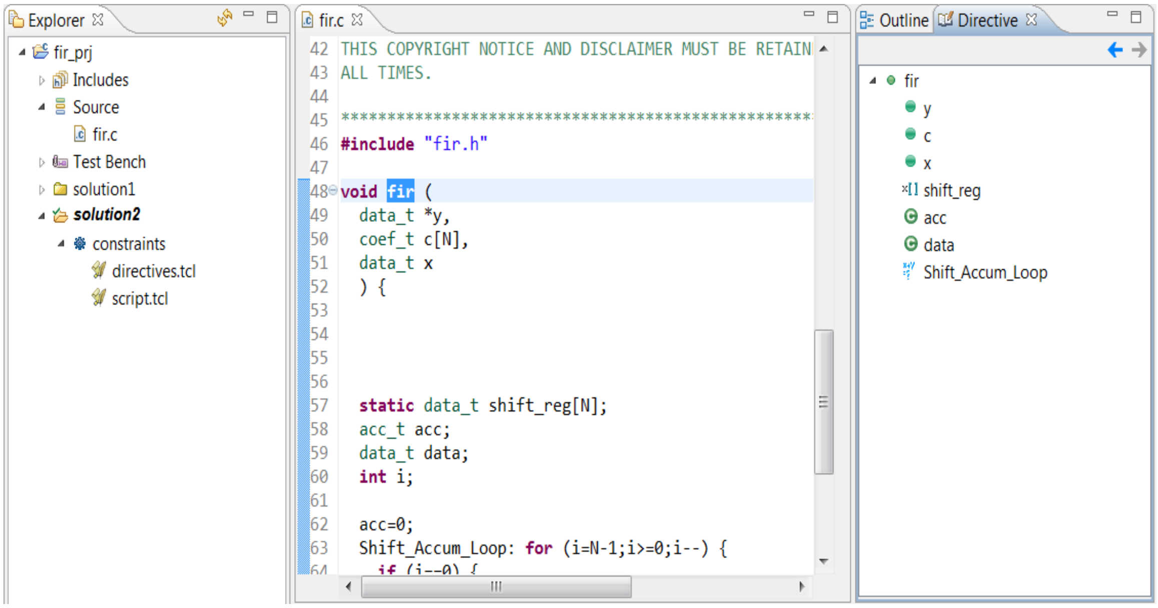


Figure 2-21: **Opening the Directives Tab**

The Directives tab, shown on the right side of Figure 2-21, lists all of the objects in the design that can be optimized. In the Directive 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 Directive**.
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 2-22.

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**.

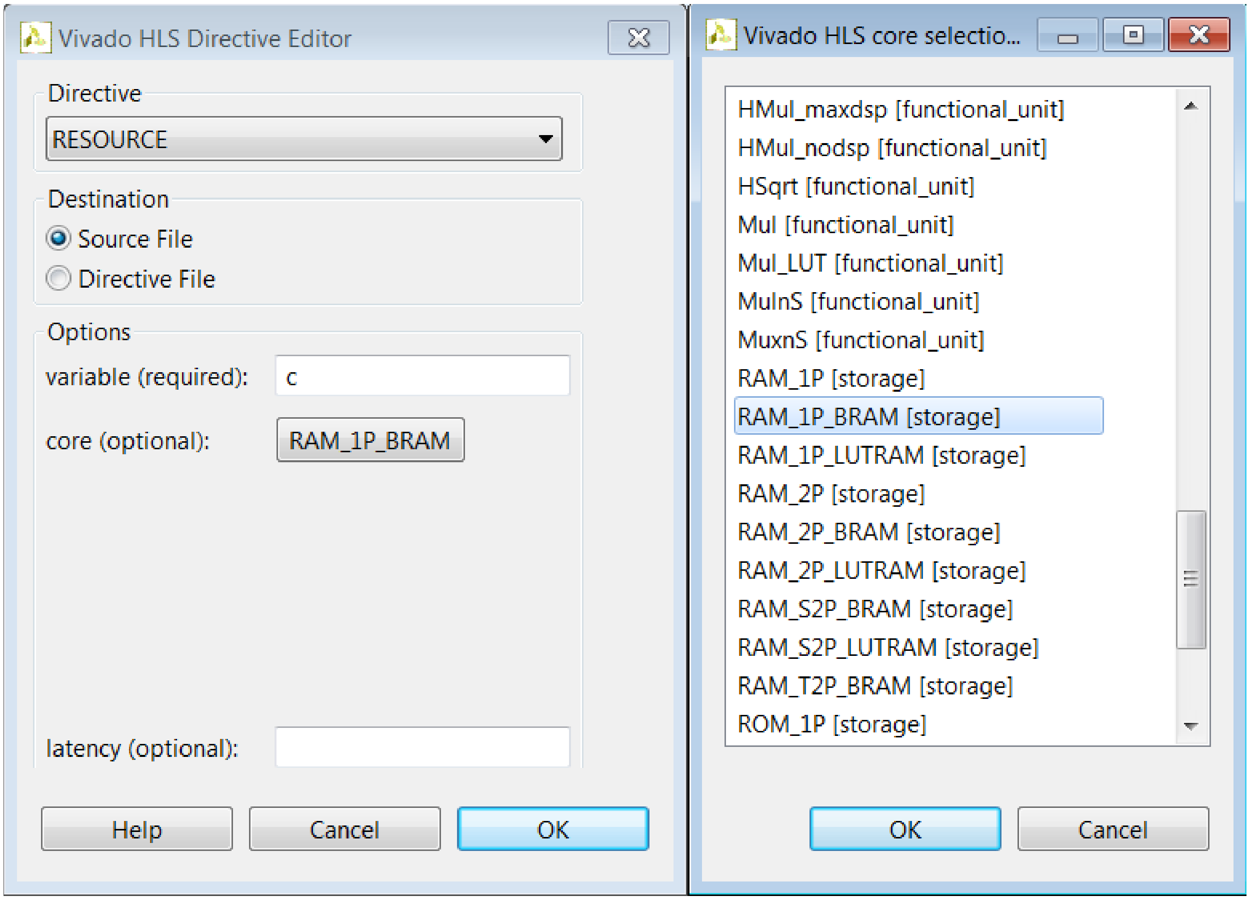


Figure 2-22: **Adding a Resource Directive**

TIP: *If you wish to change the destination of any directive, double-click on the directive In the Directive tab and modify the destination.*

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 Directive**.
    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 Directive**.

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 tab are correct as shown in Figure 2-23. Right-click on any incorrect directive to modify it.

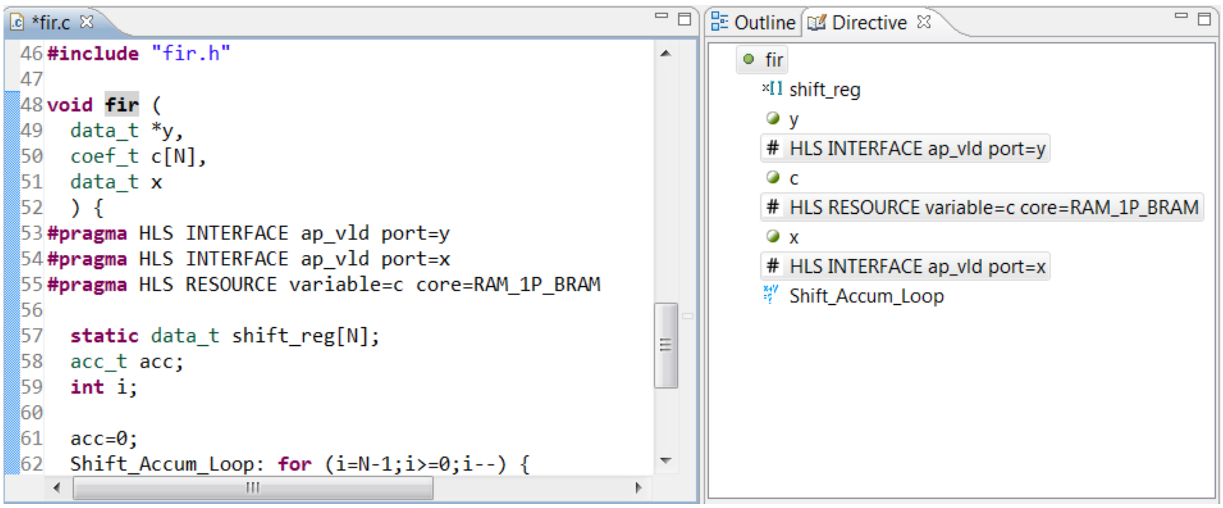


Figure 2-23: **I/O Directives for solution2**

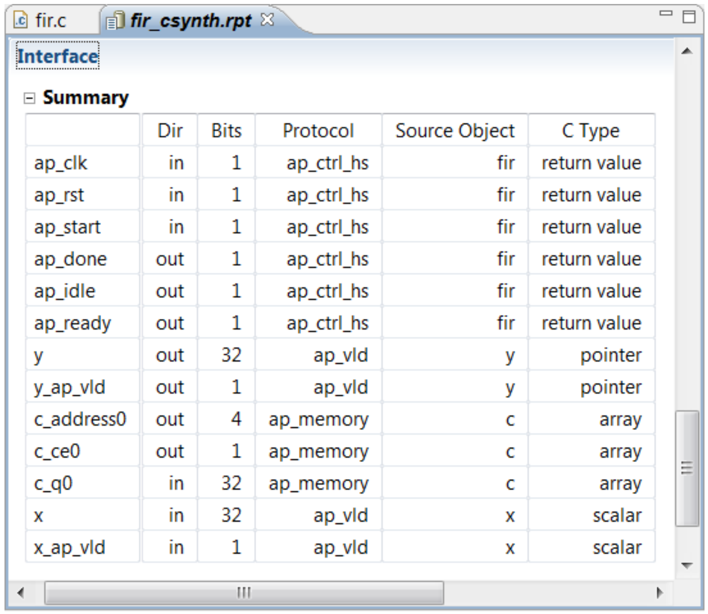
14. Click the **Run C Synthesis** toolbar button to synthesize the design.

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 2-24 shows that the ports now have the correct I/O protocols.



fir.c | fir_csynth.rpt

Interface

Summary

	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	fir	return value
ap_rst	in	1	ap_ctrl_hs	fir	return value
ap_start	in	1	ap_ctrl_hs	fir	return value
ap_done	out	1	ap_ctrl_hs	fir	return value
ap_idle	out	1	ap_ctrl_hs	fir	return value
ap_ready	out	1	ap_ctrl_hs	fir	return value
y	out	32	ap_vld	y	pointer
y_ap_vld	out	1	ap_vld	y	pointer
c_address0	out	4	ap_memory	c	array
c_ce0	out	1	ap_memory	c	array
c_q0	in	32	ap_memory	c	array
x	in	32	ap_vld	x	scalar
x_ap_vld	in	1	ap_vld	x	scalar

Figure 2-24: **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 2-25:

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 2-25 is used shortly in an explanation; it is not part of the view.
- The Chapter 6, Design Analysis tutorial 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.

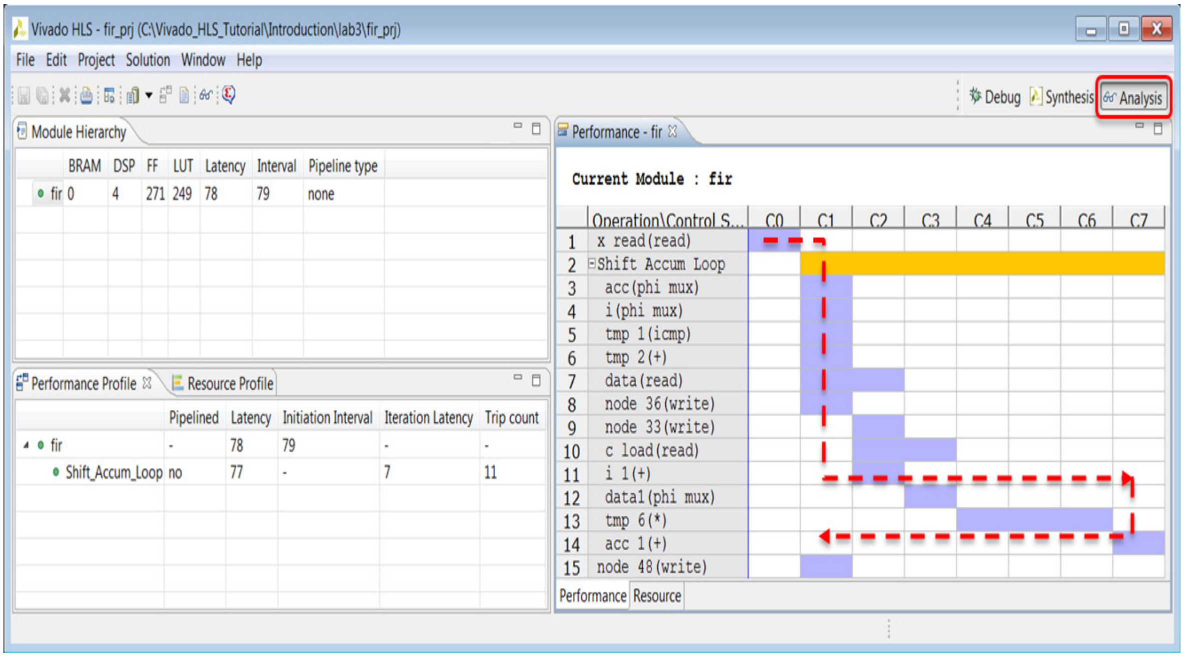


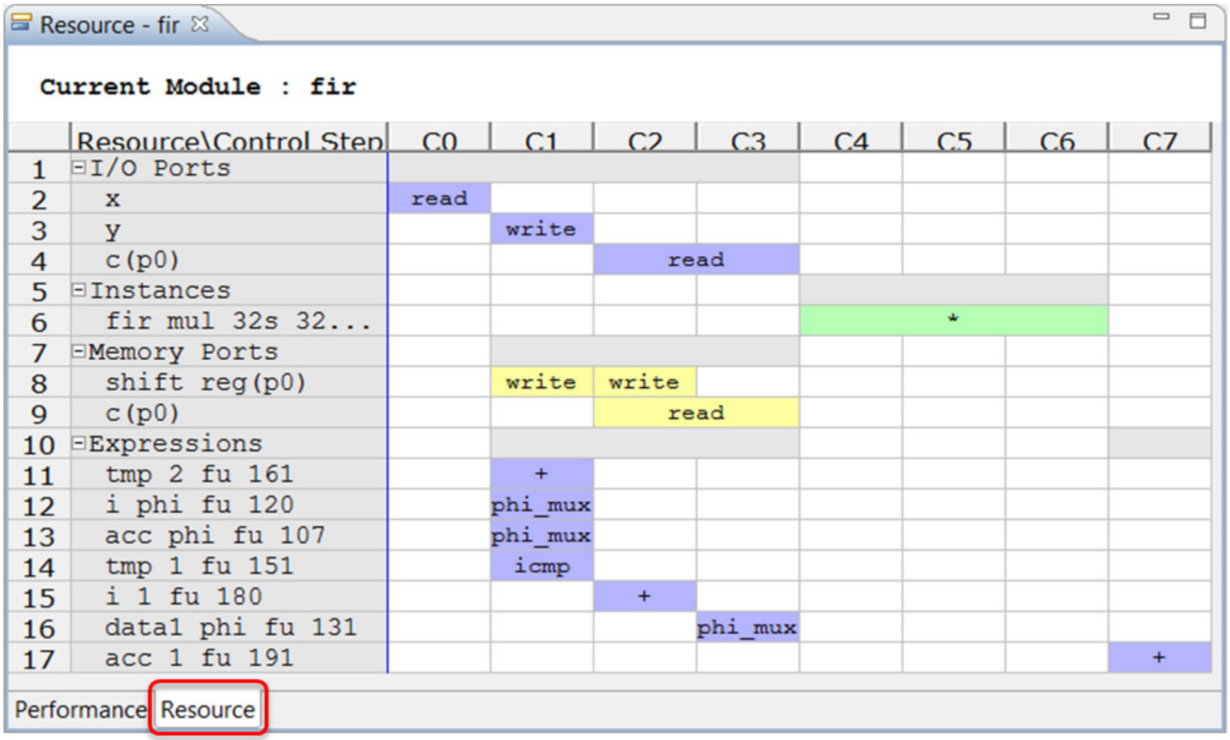
Figure 2-25: **Solution2 Analysis Perspective: Performance**

The explanation presented here follows the path of the dotted red line in Figure 2-25. 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 the yellow loop body in this view shows the loop details: 7 cycles, 11 iterations for a total latency of 77.
- 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 is a memory read on the c port.
- The multiplication operation takes 3 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 2-26.
4. Expand all the resource groups (also shown in Figure 2-26).



	Resource\Control Step	C0	C1	C2	C3	C4	C5	C6	C7
1	I/O Ports								
2	x	read							
3	y		write						
4	c(p0)			read					
5	Instances								
6	fir mul 32s 32...						*		
7	Memory Ports								
8	shift reg(p0)		write	write					
9	c(p0)			read					
10	Expressions								
11	tmp 2 fu 161		+						
12	i phi fu 120		phi_mux						
13	acc phi fu 107		phi_mux						
14	tmp 1 fu 151		icmp						
15	i 1 fu 180			+					
16	data1 phi fu 131				phi_mux				
17	acc 1 fu 191								+

Figure 2-26: **Solution2 Analysis Perspective: Resource**

Figure 2-27 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.

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

Before concluding the analysis, it is worth commenting on the multicycle 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 Arbitrary Precision Types tutorial 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 2-27 explain how to unroll the loop.

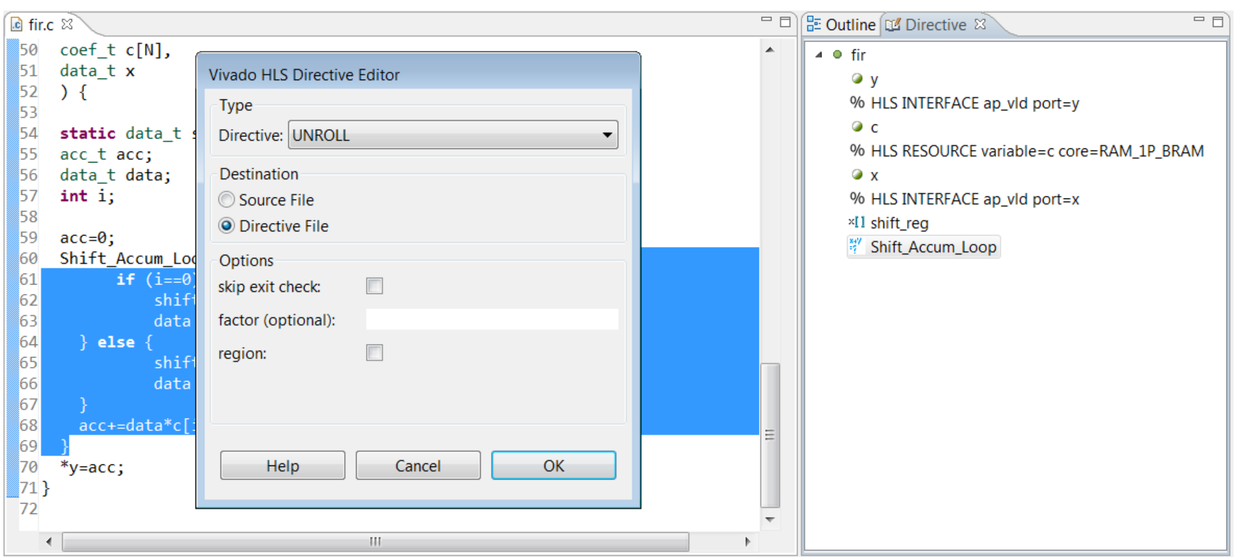


Figure 2-27: **Unrolling FOR Loop**

5. In the Directive tab, select loop **Shift_Accum_Loop**.

IMPORTANT: *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 Directive**.
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 Directive**.
   c. Select **Array_Partition** from the Directive drop-down menu.
   d. Specify the type as **complete**.
   e. Select **OK** to apply the directive.

With the directives embedded in the code from `solution2` and the two new directives just added, the directive pane for `solution3` appears as shown in Figure 2-28.

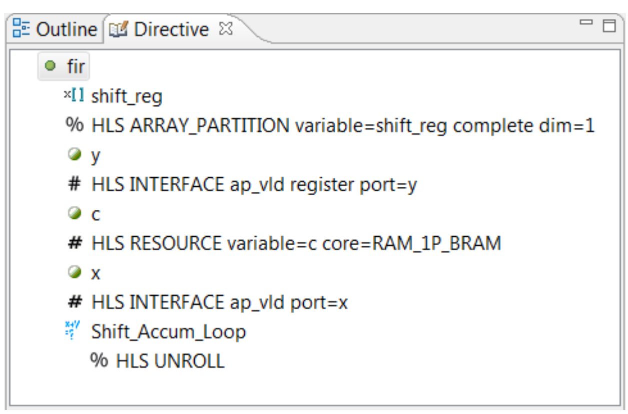


Figure 2-28: **Solution3 Directives**

In Figure 2-28, 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, as shown next.

10. In the Explorer pane, expand the **Constraint** folder in `Solution3` as shown in Figure 2-29.
11. Double-click the `solution3 directives.tcl` file to open it in the Information pane.

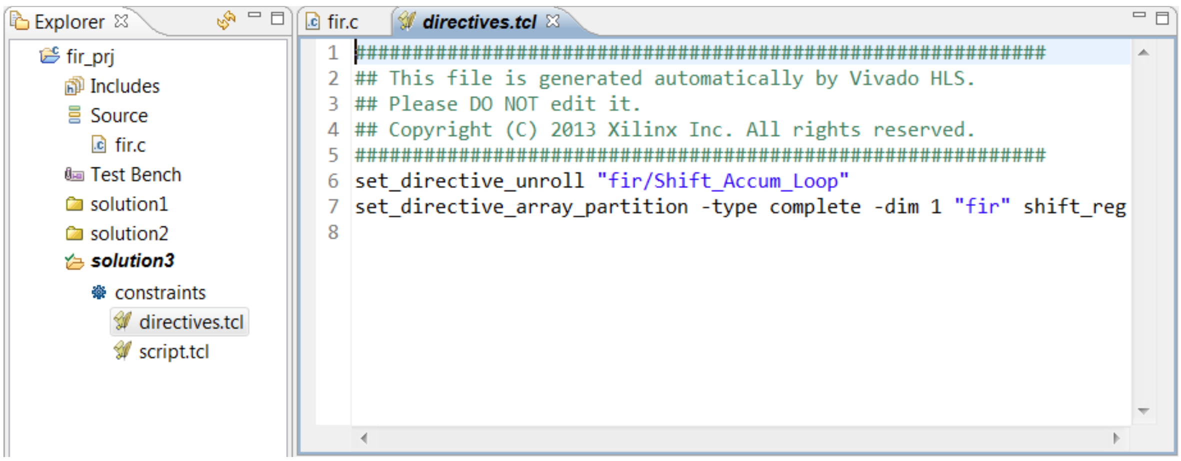


Figure 2-29: **Solution4 Directives.tcl File**

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 2-30 shows the comparison of the reports. `solution3` has the smallest initiation interval and can process data much faster. As the interval is only 16, it starts to process a new set of inputs every 16 clock cycles.

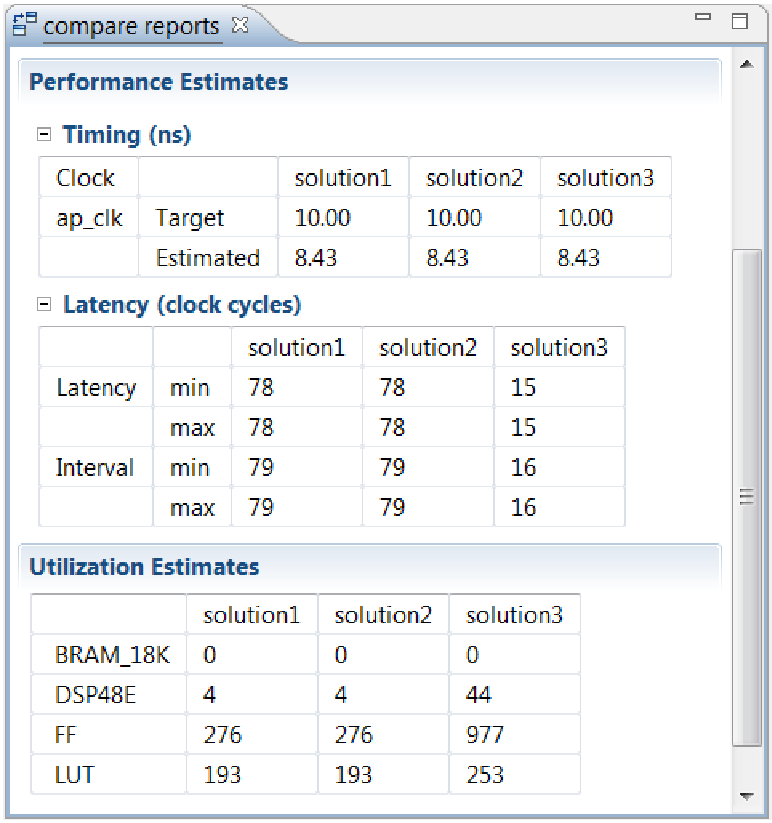


Figure 2-30: **Solution**

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 Chapter 7, Design Optimization tutorial 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 Chapter 5, Arbitrary Precision Types tutorial.

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.

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 that are incorrect "by design" and ensures the RTL verification can be performed automatically.

This tutorial consists of three lab exercises.

Lab 1 Description

Reviews the aspects of a good C test bench, the basic operations for C validation and the C debugger.

Lab 2 Description

Validates and debugs a C design using arbitrary precision C types.

Lab 3 Description

Validates and debugs 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 Locating the Tutorial Design Files.

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.

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.
   - On Windows use **Start > All Programs > Xilinx Design Tools > Vivado 2015.3 > Vivado HLS > Vivado HLS 2015.3 Command Prompt** (Figure 3-1).
   - On Linux, open a new shell.

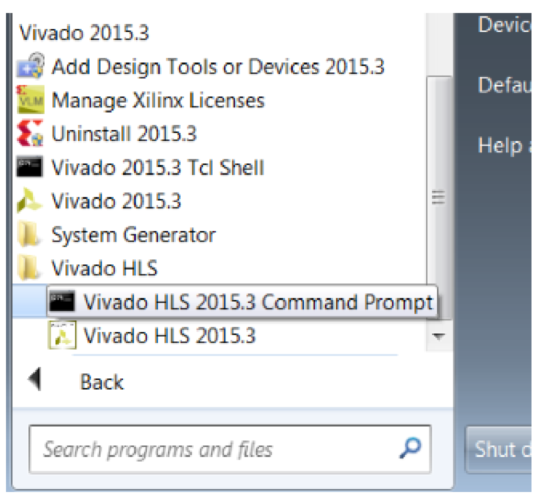


Figure 3-1: **Vivado HLS Command Prompt**

2. Using the command prompt window (Figure 3-2), 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 -f run_hls.tcl` as shown in Figure 3-2.

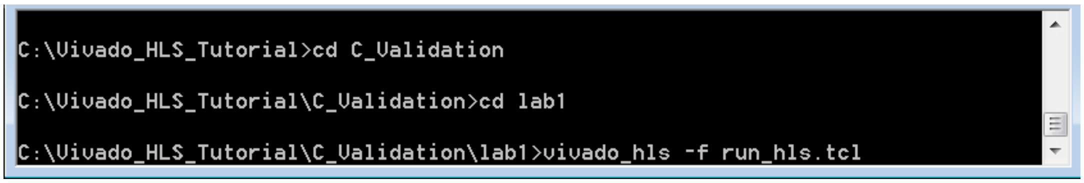


Figure 3-2: **Setup the Tutorial Project**

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 3-3.

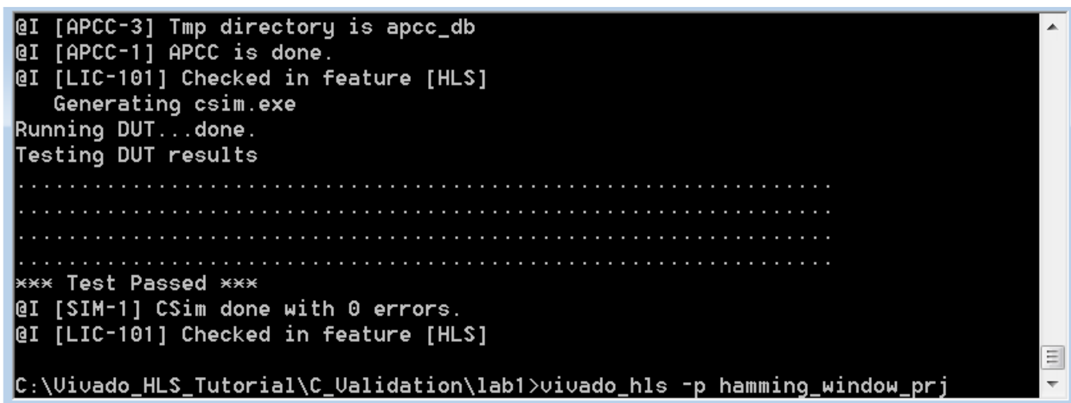


Figure 3-3: **Initial Project for C Validation Lab 1**

Step 2: Review Test Bench and Run C Simulation

1. Open the C test bench for review by double-clicking `hamming_window_test.c` in the Test Bench folder (Figure 3-4).

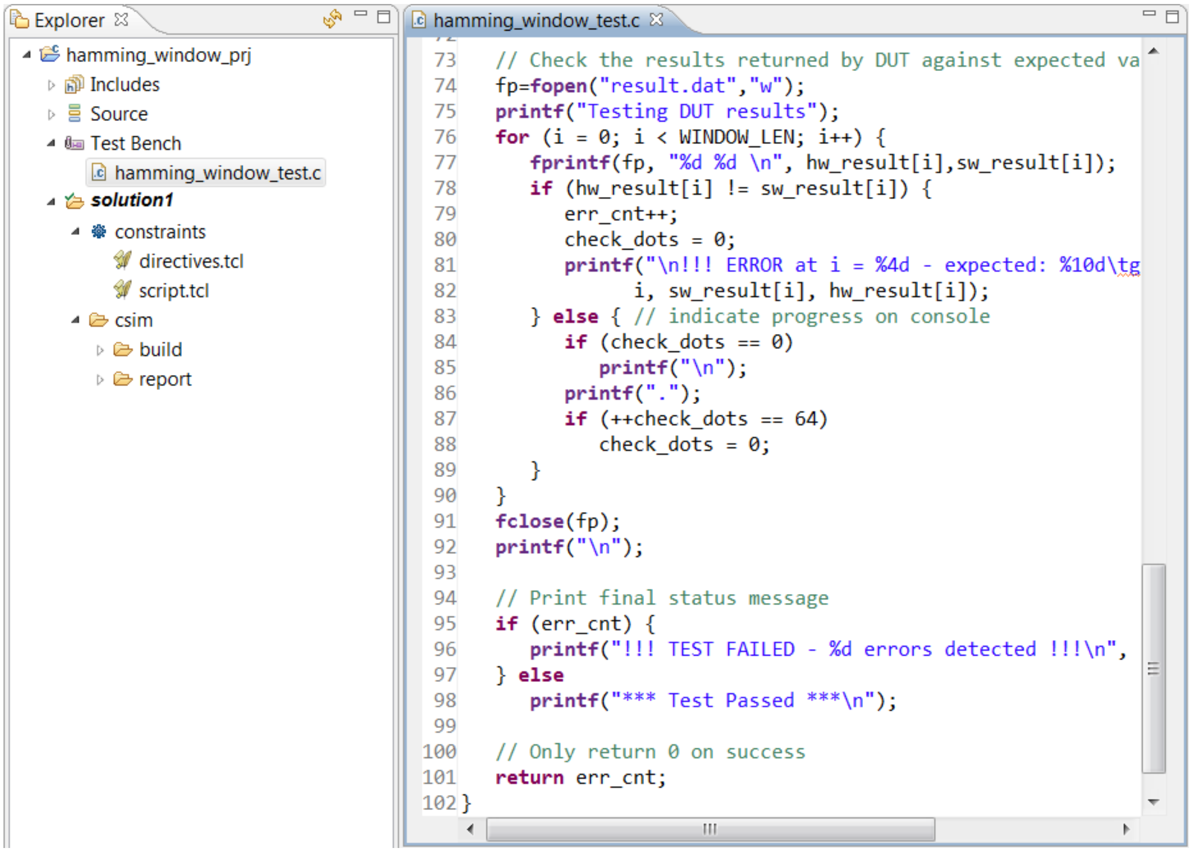


Figure 3-4: **C Test Bench for C Validation Lab 1**

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

- The test bench:
  - 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 3-5.

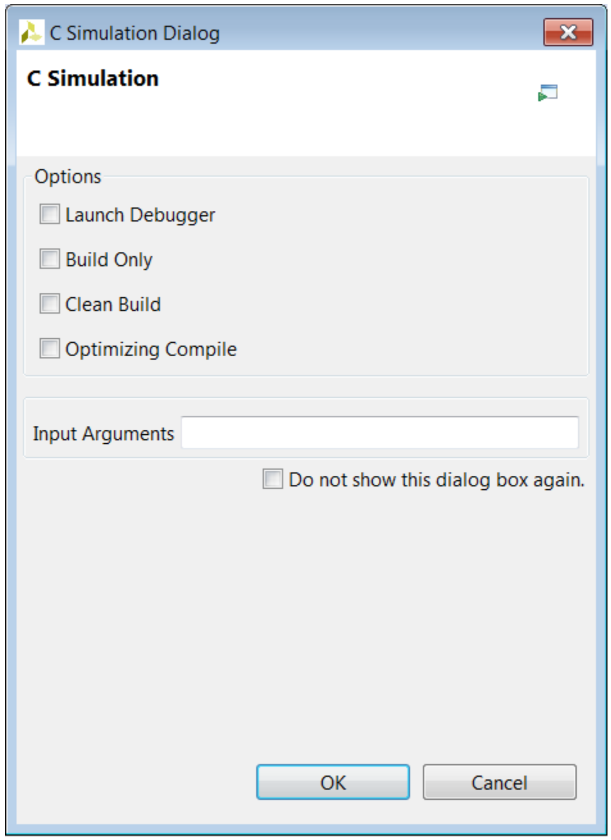


Figure 3-5: **Run C Simulation Dialog Box**

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

As shown in Figure 3-6, 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 subdirectory `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 highlighted in Figure 3-6.

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.

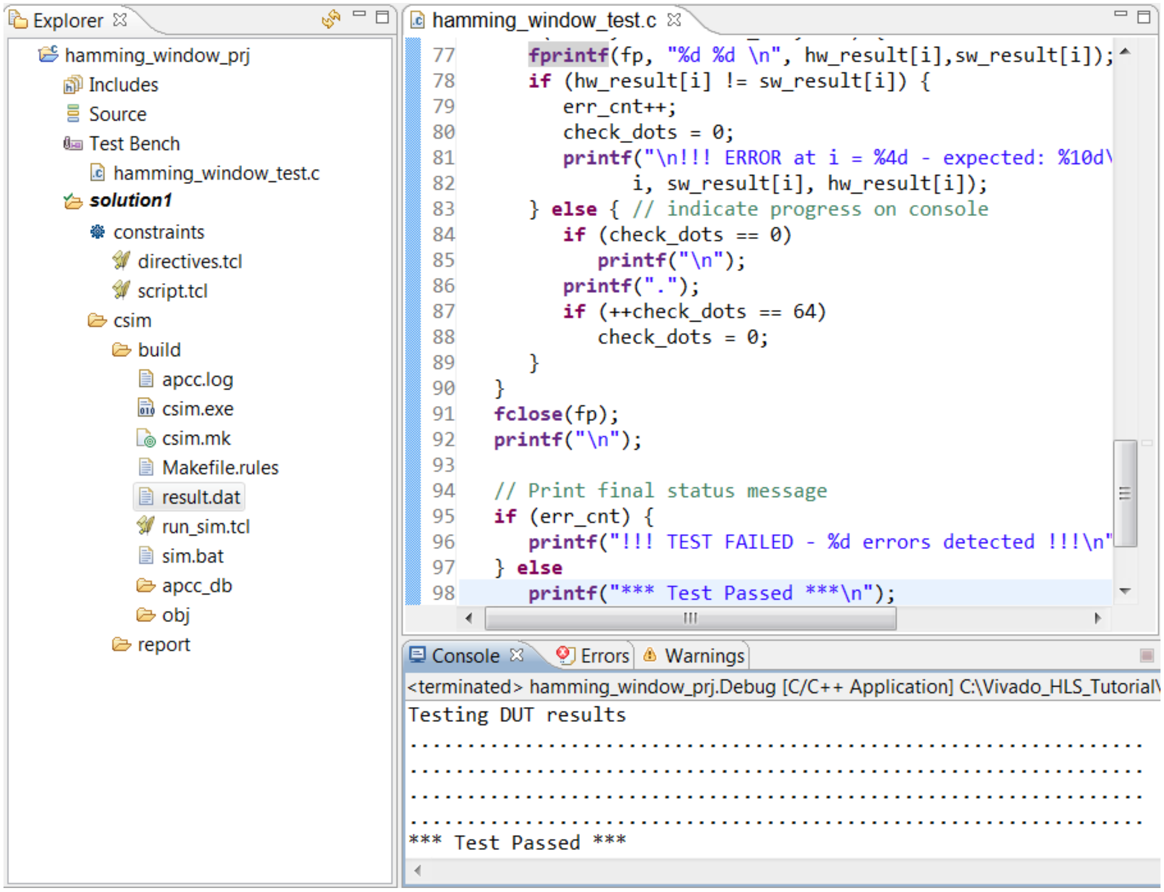


Figure 3-6: **C Simulation Results**

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 **Launch Debugger** option as shown in Figure 3-7.
3. Click **OK** to run the simulation.

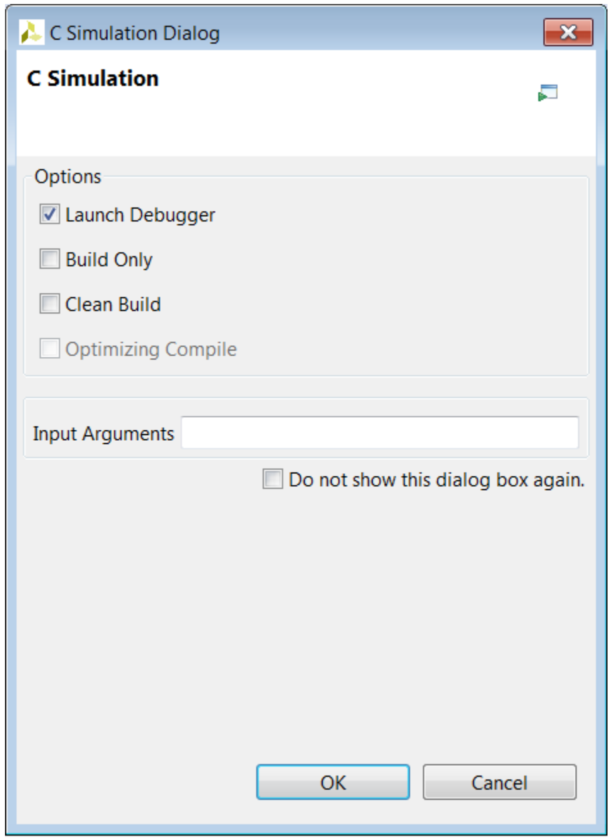


Figure 3-7: **C Simulation Dialog Box**

The Debug option compiles the C code and then opens the Debug environment, as shown in Figure 3-8. Before proceeding, note the following:

- Highlighted at the top-left in Figure 3-8, 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.

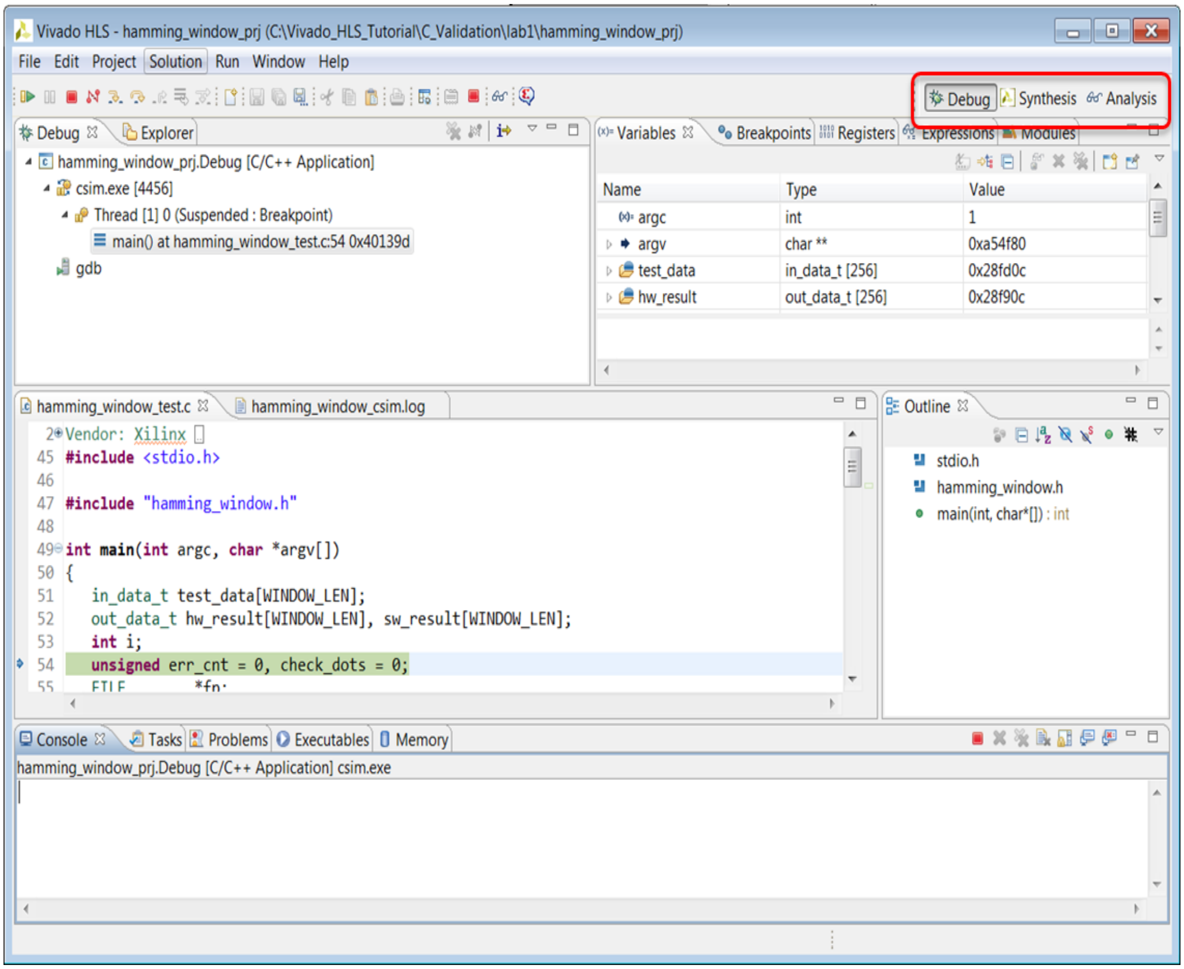


Figure 3-8: **The HLS Debug Perspective**

You can use the **Step Into** button (Figure 3-9) to step through the code line-by-line.

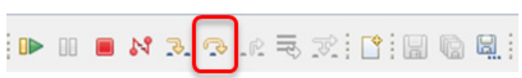


Figure 3-9: **The Debug Step Into Button**

4. Expand the Variables window to see the `sw_results` array.
5. Expand the `sw_results` array to the view shown in Figure 3-10.
6. Click the **Step Into** button (or key F5) repeatedly until you see the values being updated in the Variables window.

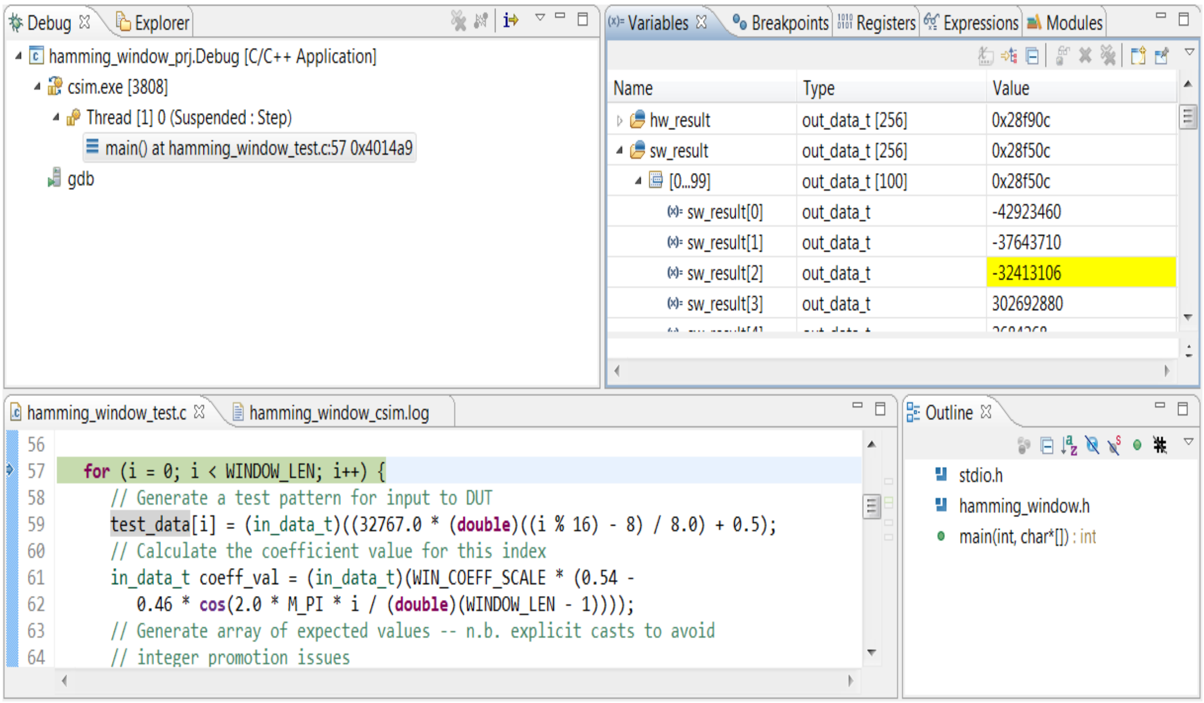


Figure 3-10: **Analysis of C Variables**

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. Place the cursor in the left-hand margin on line 69, right-click with the mouse button and select Toggle Breakpoint. A breakpoint (blue dot) is indicated in the margin, as shown in Figure 3-11.
9. Activate the Breakpoints tab, also shown in Figure 3-11, to confirm there is a breakpoint set at line 69.
10. Click the **Resume** button (highlighted in Figure 3-11) or the F8 key to execute up to the breakpoint.

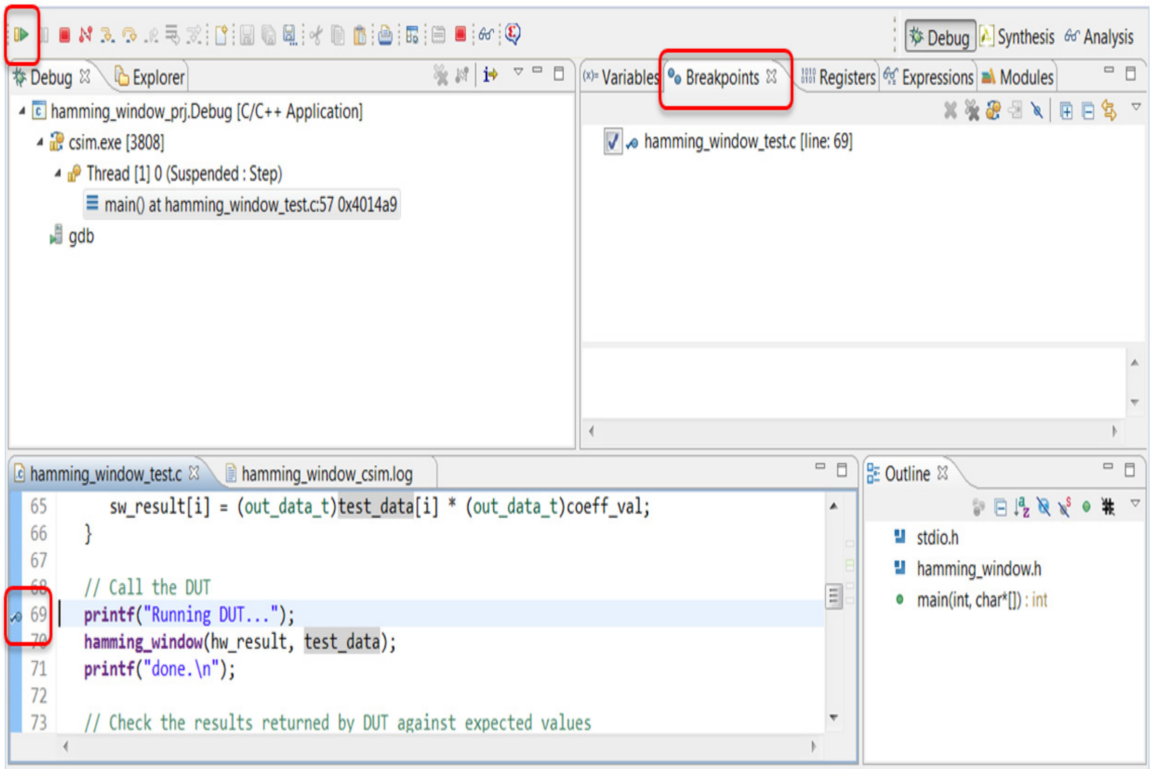


Figure 3-11: **Using Breakpoints**

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.

You can use the Run C simulation button to 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 3-12.
2. To create a new **Vivado HLS** project, type `vivado_hls -f run_hls.tcl`.

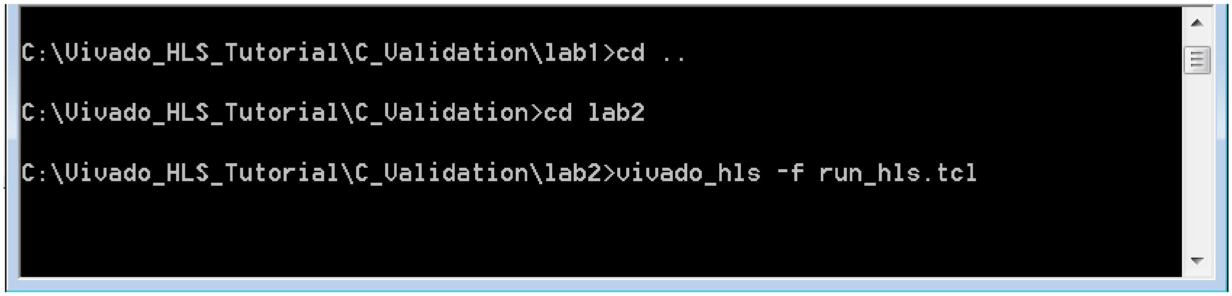


Figure 3-12: **Setup for Interface Synthesis Lab 2**

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 3-13.

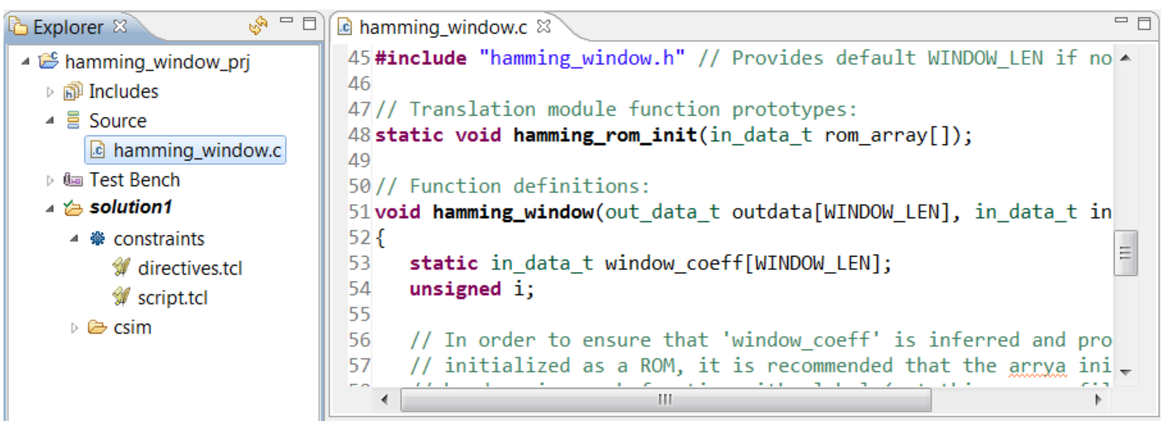


Figure 3-13: **C Code for C Validation Lab 2**

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 3-14).

```
// scaled integer, which may be interpreted as a signed fixed po
// with WIN_COEFF_FRACBITS bits after the binary point.

//typedef int16_t       in_data_t;
//typedef int32_t       out_data_t;
#include "ap_cint.h"
typedef int16       in_data_t;
typedef int32       out_data_t;

void hamming_window(out_data_t outdata[], in_data_t indata[]);

#endif // HAMMING_WINDOW_H_ not defined
```

Figure 3-14: **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 Chapter 5, Arbitrary Precision Types tutorial. 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 **Launch Debugger** option.
3. Click **OK** to run the simulation.

The warning and error message shown in Figure 3-15 appears.

The message in the console pane and log file indicate 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.*

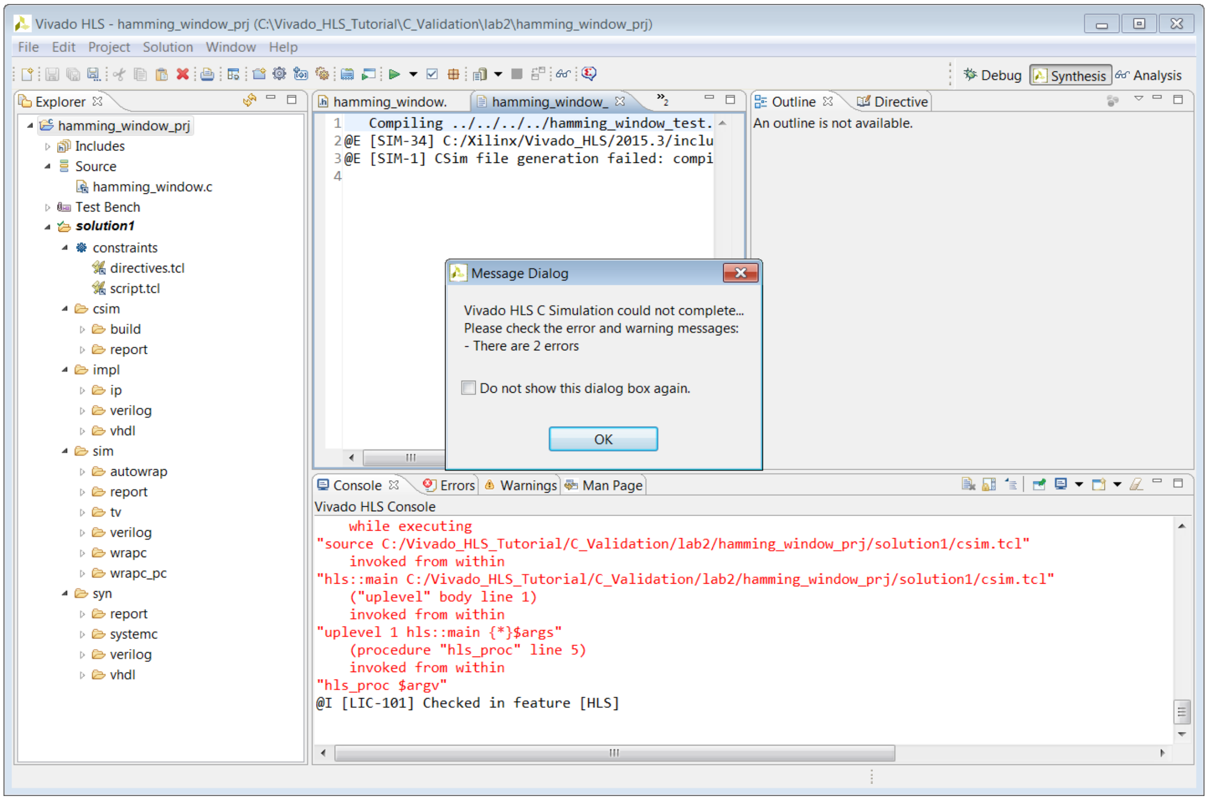


Figure 3-15: **C Simulation Dialog Box**

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 3-16).

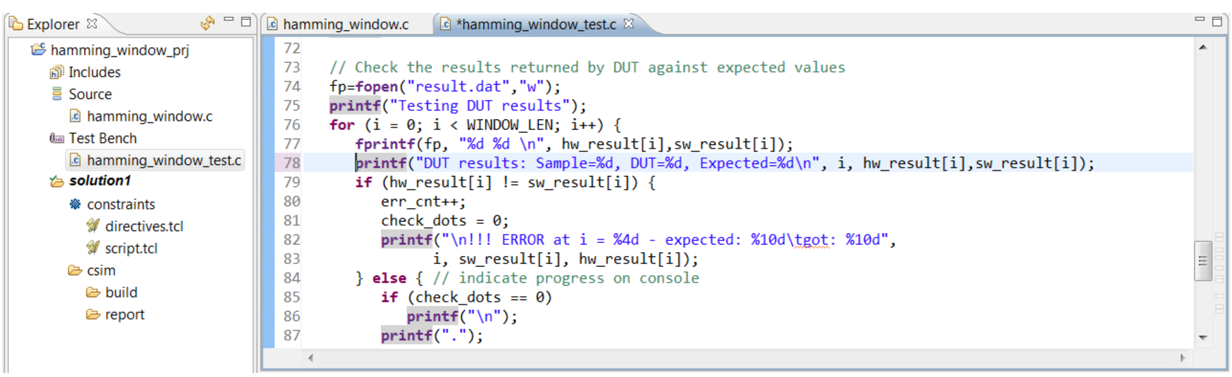


Figure 3-16: **Enable Printing of the Results**

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. Ensure the **Launch Debugger** option is **not selected**.
10. Click **OK** to run the simulation.

The results appear in the console window (Figure 3-17).

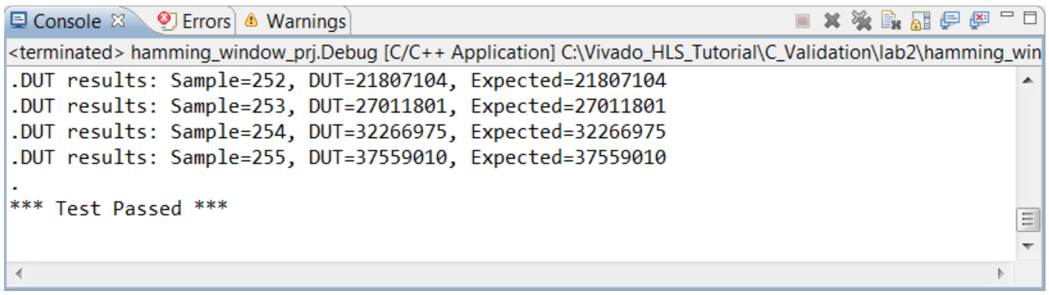


Figure 3-17: **C Validation Lab 2 Results**

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

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 3-18.

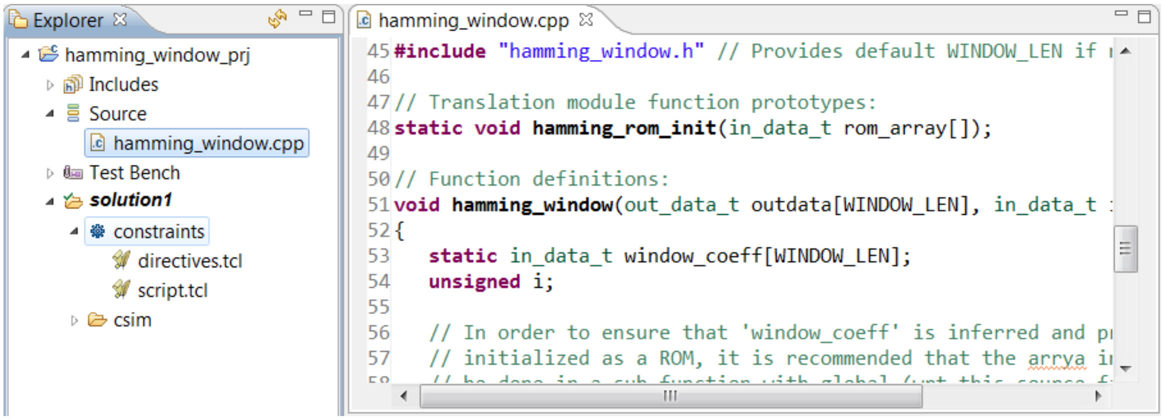


Figure 3-18: **C++ Code for C Validation Lab 3**

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 3-19).

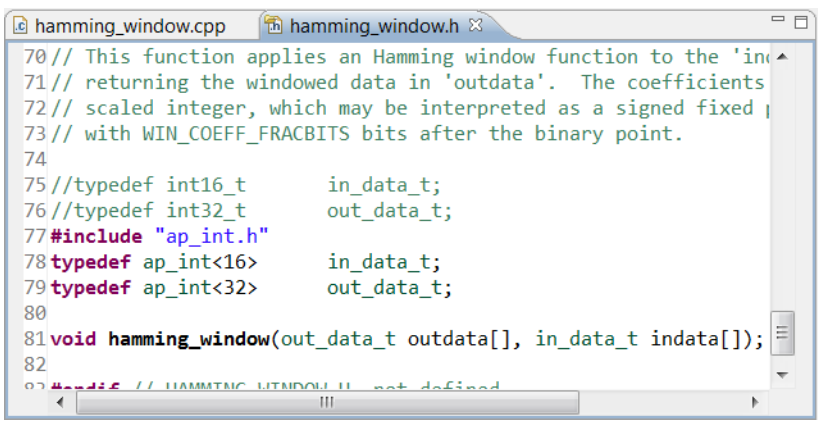


Figure 3-19: **Type Definitions for C Validation Lab 3**

Note: In this lab, the design is the same as in Lab 1 and Lab 2, with one exception. The design is now C++ and the types have been updated to use the C++ arbitrary precision types, ap_int<#N>, provided by Vivado HLS 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 **Launch Debugger** 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 3-20.
6. Click the **Resume** button (or key F8) to execute the code up to the breakpoint.

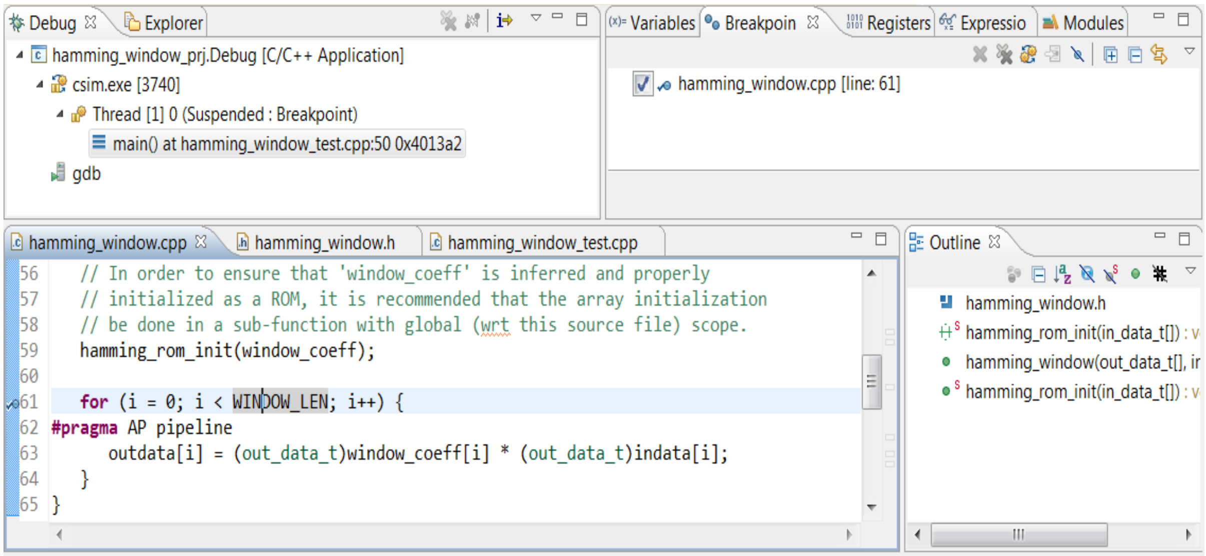


Figure 3-20: **Debug Environment for C Validation Lab 3**

7. Click the **Step Into** button (or press the **F5** key) twice to see the view in Figure 3-21.

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.

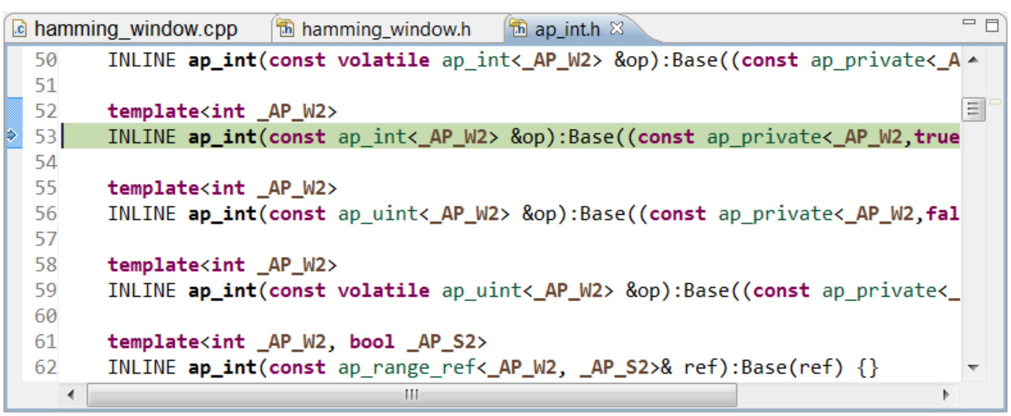


Figure 3-21: **Arbitrary Precision Header File**

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 3-22 to see the value of the variable shown in the VAL parameter.

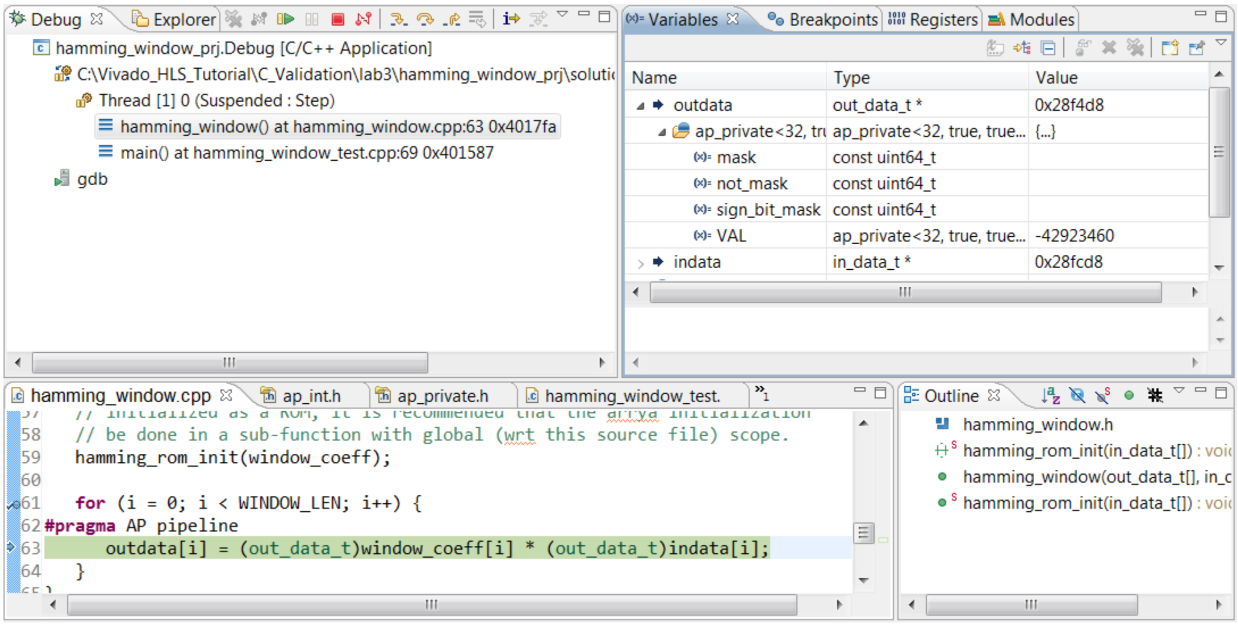


Figure 3-22: **Arbitrary Precision Variables**

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

Overview

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 synchronized automatically and optimally with the internal logic.

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

Lab 1 Description

Review the function return and block-level protocols.

Lab 2 Description

Understand the default I/O protocol for ports and learn how to select an I/O protocol.

Lab 3 Description

Review how array ports are implemented and can be partitioned.

Lab 4 Description

Create an optimized implementation of the design and add AXI4 interfaces.

Tutorial Design Description

Download tutorial design file from the Xilinx website. See Locating the Tutorial Design Files.

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 multichannel 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.

Lab 1: Block-Level I/O Protocols

Overview

This lab explains what block-level I/O protocols are and how 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.
   - On Windows use **Start > All Programs > Xilinx Design Tools > Vivado 2015.3 > Vivado HLS > Vivado HLS 2015.3 Command Prompt** (Figure 4-1).
   - In Linux, open a new shell.

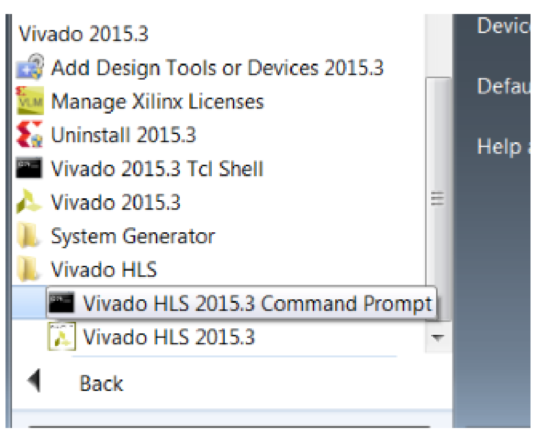


Figure 4-1: **Vivado HLS Command Prompt**

2. Using the command prompt window (Figure 4-2), 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 4-2.

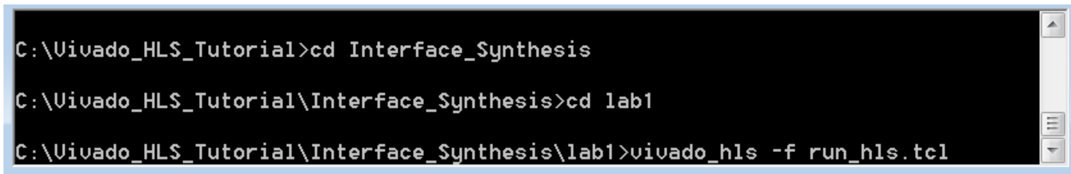


Figure 4-2: **Setup the Tutorial Project**

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 4-3.

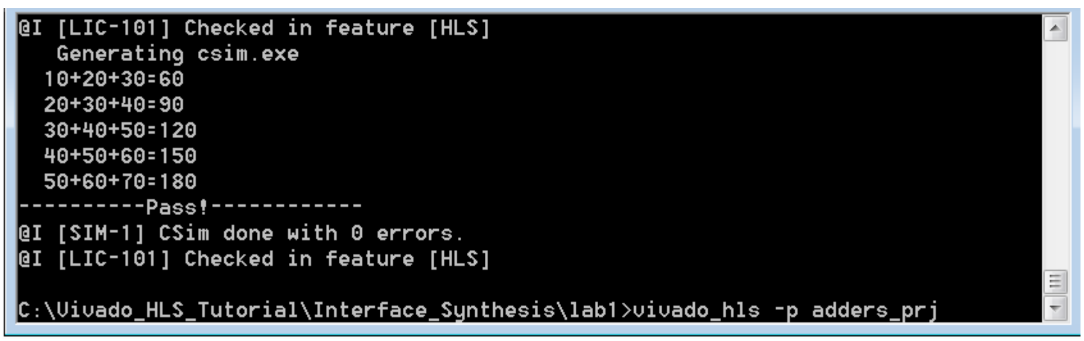


Figure 4-3: **Initial Project for Interface Synthesis Lab 1**

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

1. Double-click `adders.c` in the **Source** folder to pen the source code for review (Figure 4-4).

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.

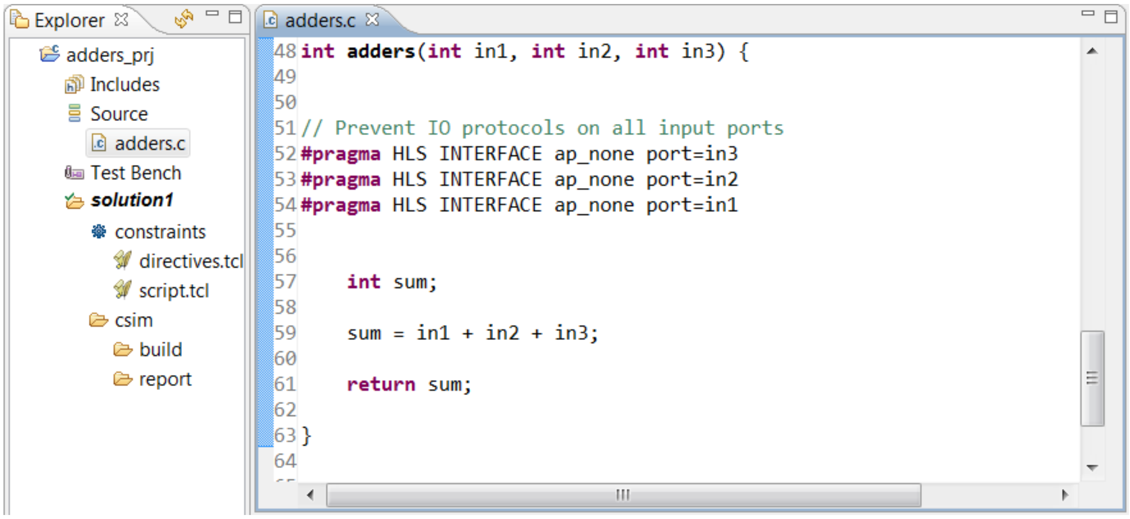


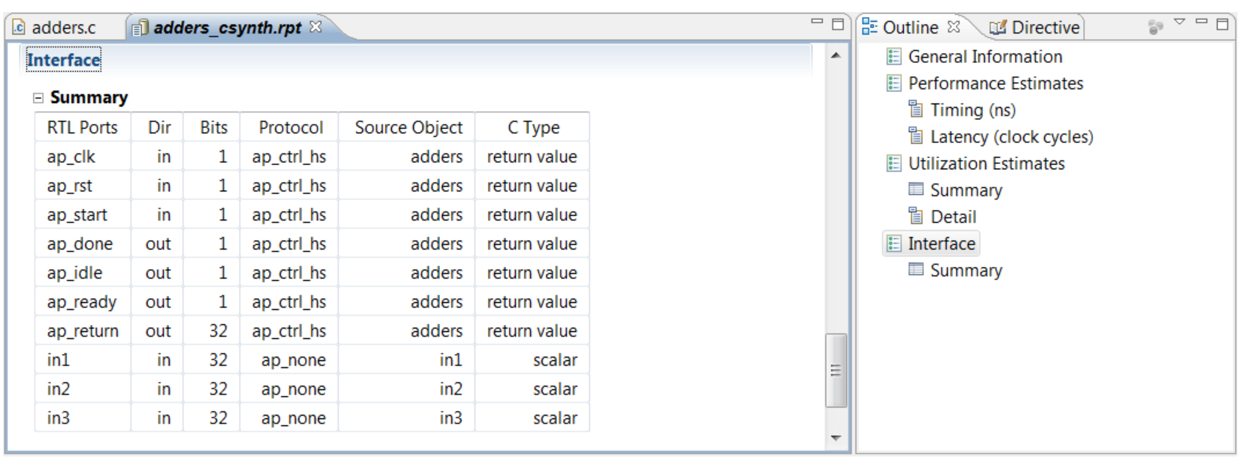
Figure 4-4: **C Code for Interface Synthesis Lab 1**

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.

The Interface summary and Outline tab are shown in Figure 4-5.



RTL Ports	Dir	Bits	Protocol	Source Object	C Type
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

Figure 4-5: **Interface Summary**

There are four types of ports to review:

- 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 4-5).
  - 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 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 4-6 shows this protocol is associated with the function return value (this is true even if the function has no return value specified in the code).

Table 4-1 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).

Table 4-1: **Block Level I/O protocol ap_ctrl_hs**

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.

Table 4-1: **Block Level I/O protocol ap_ctrl_hs**

Exercise	Description
`ap_done`	This signal indicates when the design has completed all operations in the current transaction. 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.

You can observe the behavior of these signals by viewing the trace file produced by RTL cosimulation. This is discussed in Chapter 7, RTL Verification tutorial, but Figure 4-6 shows the waveforms for the current synthesis results.

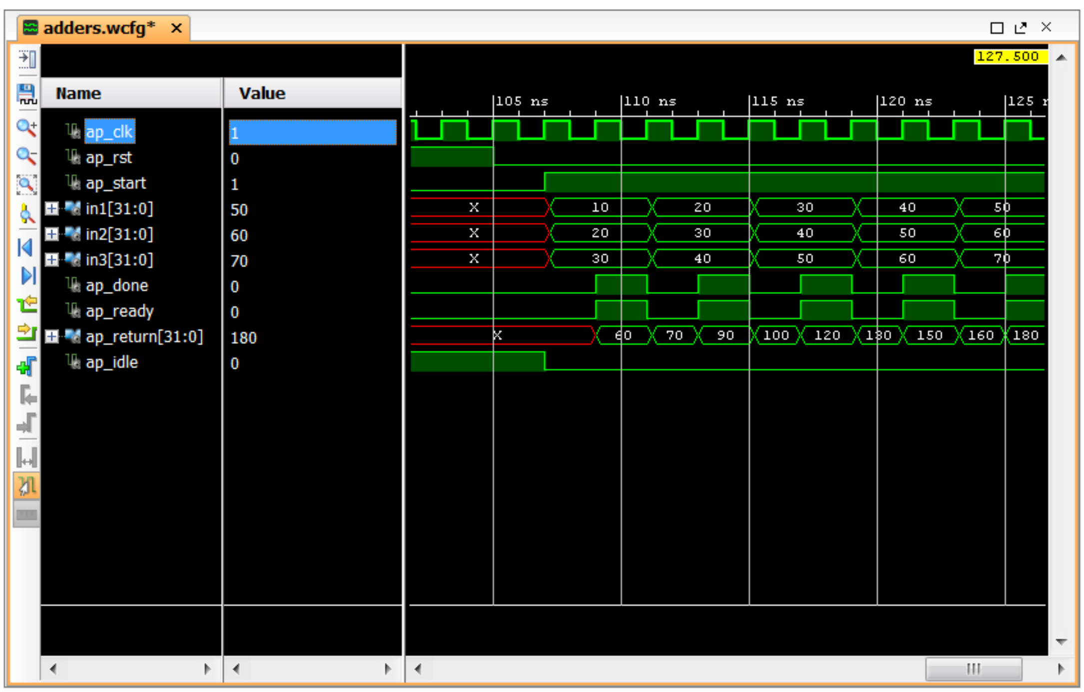


Figure 4-6: **RTL Waveforms for Block Protocol Signals**

The waveforms in Figure 4-6 show the behavior of the block-level I/O signals.

- 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 4-6 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 **Finish**.
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 4-7.

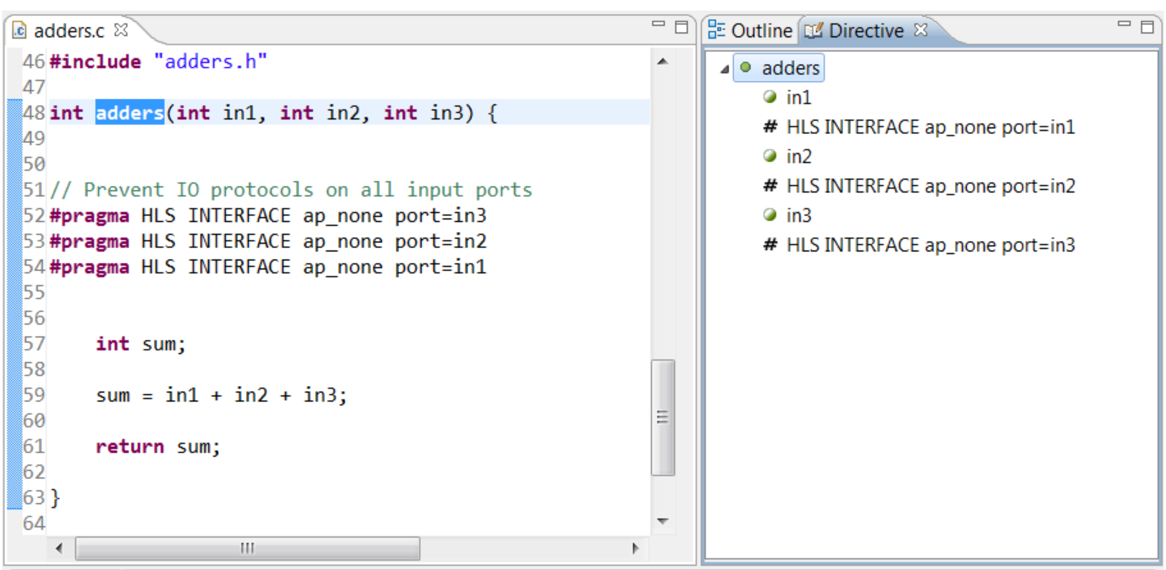


Figure 4-7: **Top-Level Function Selected**

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

The Directives Editor dialog box opens.

Figure 4-8 shows this dialog box with the drop-down menu for the interface mode activated.

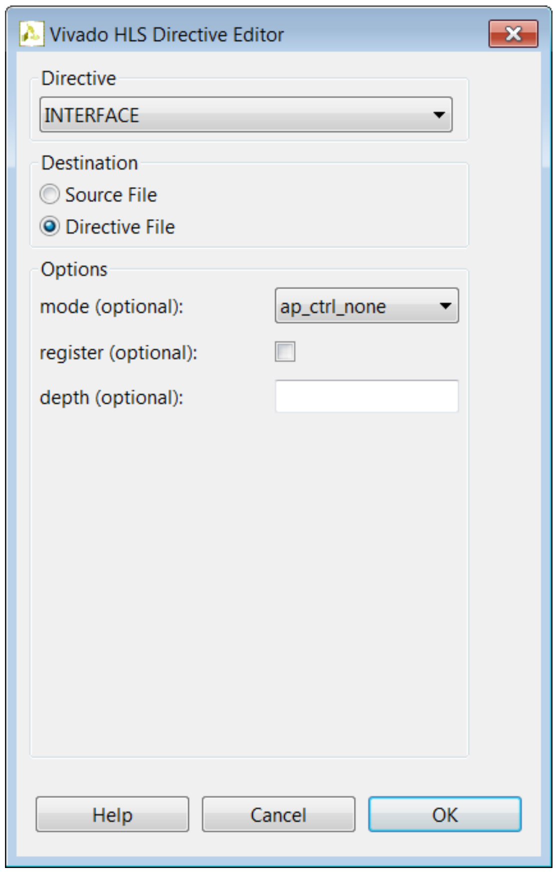


Figure 4-8: **Directive Dialog Box for ap_ctrl_none**

The drop-down menu shows there are four 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.

- `s_axilite`: May be applied in addition to `ap_ctrl_hs` or `ap_ctrl_chain` to implement the block-level I/O protocol as an AXI Slave Lite interface in place of separate discrete I/O ports.

The block-level I/O 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 proceed. This allows downstream blocks to apply back-pressure on the system and halt further processing when they are unable to continue accepting 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 existing I/O directives.

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

The source file now has a new directive, highlighted in both the source code and directives tab in Figure 4-9.

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.

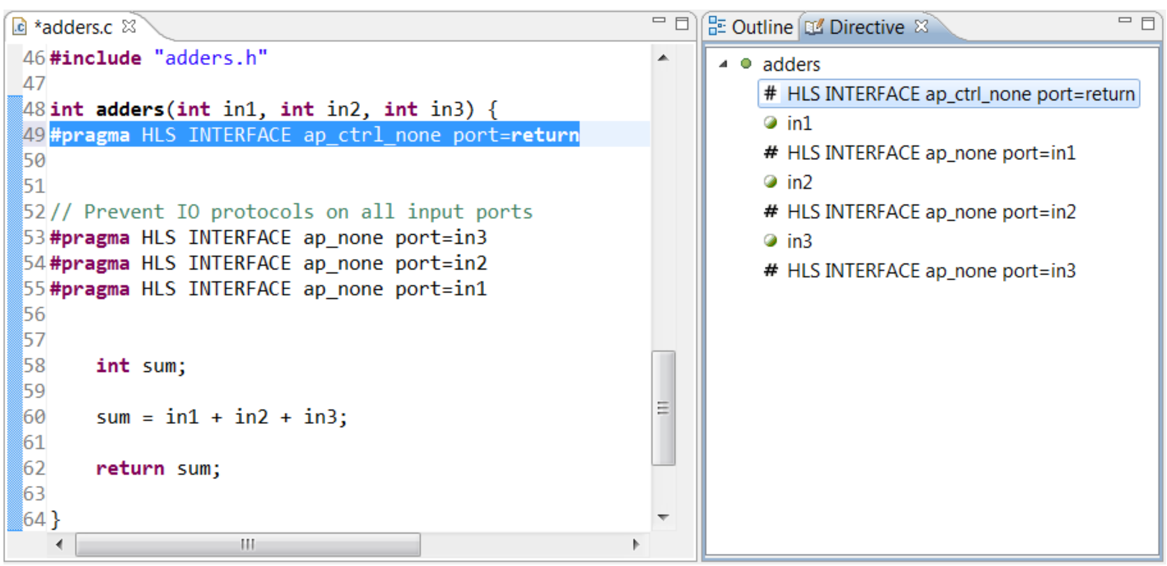


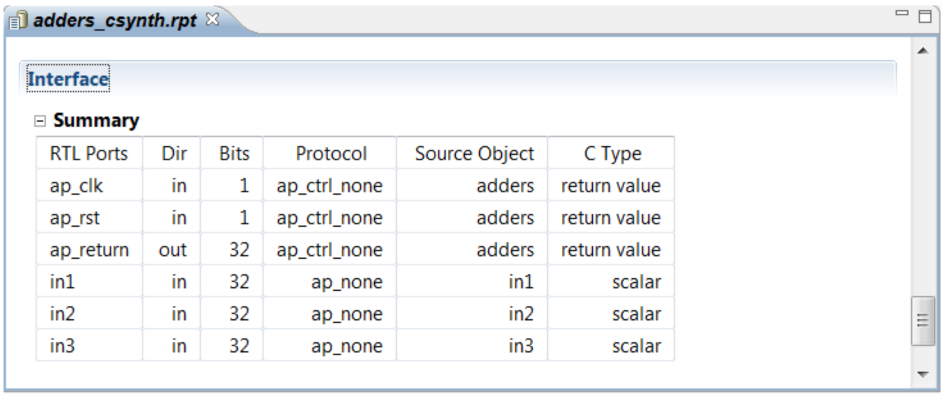
Figure 4-9: **Block-Level Interface Directive ap_ctrl_none**

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 4-9 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 4-10.



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 4-10: **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.

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 4-11.
2. Type `vivado_hls -f run_hls.tcl` to create a new Vivado HLS project.

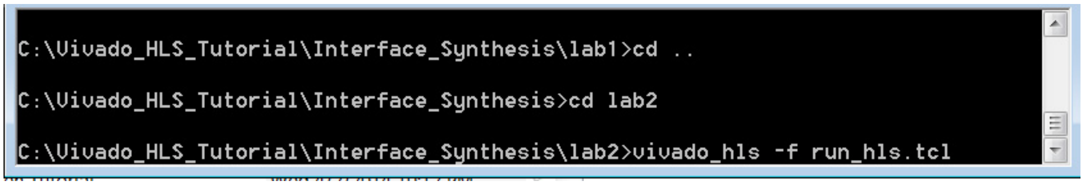


Figure 4-11: **Setup for Interface Synthesis Lab 2**

3. Type `vivado_hls -p adders_io_prj` to open the Vivado HLS GUI project.
4. Open the source code as shown in Figure 4-12.

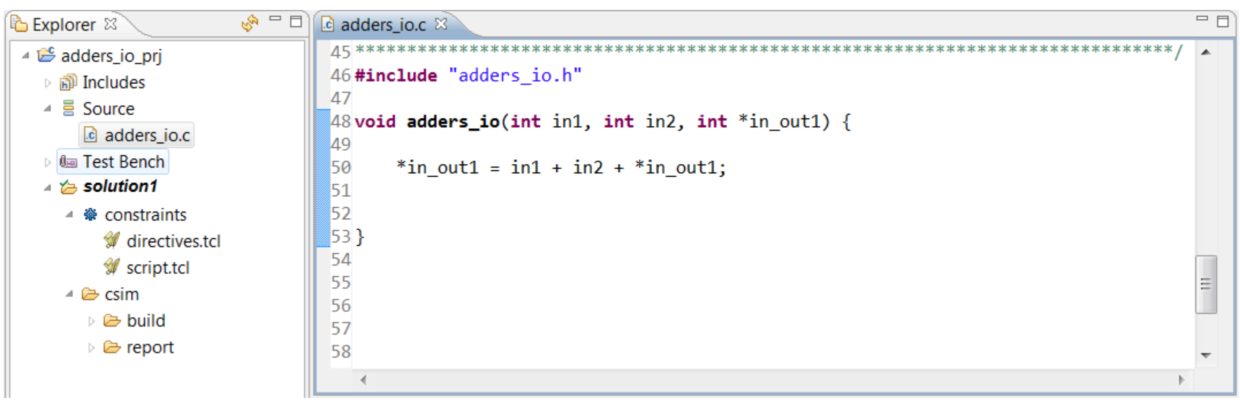


Figure 4-12: **C Code for Interface Sythesis Lab 2**

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 bidirectional (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® Design Suite User Guide: High-Level Synthesis* (UG902) [Ref 2].

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.

For the code shown in Figure 4-12, the possible options for each function argument are described in Table 4-2.

Table 4-2: **Port Level I/O Protocol Options for Lab 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: • ap_none: No I/O protocol. This is the default for inputs. • ap_stable: No I/O protocol. • ap_ack: Implemented with an associated output acknowledge port. • ap_vld: Implemented with an associated input valid port. • ap_hs: Implemented with both input valid and output acknowledge ports.
in_out1	This is a pass-by-reference output that can be implemented with the following I/O protocols: • ap_none: No I/O protocol. This is the default for inputs. • ap_stable: No I/O protocol. • ap_ack: Implemented with an associated input acknowledge port. • 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. • ap_bus: A Vivado HLS bus interface protocol.

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.

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 4-13.

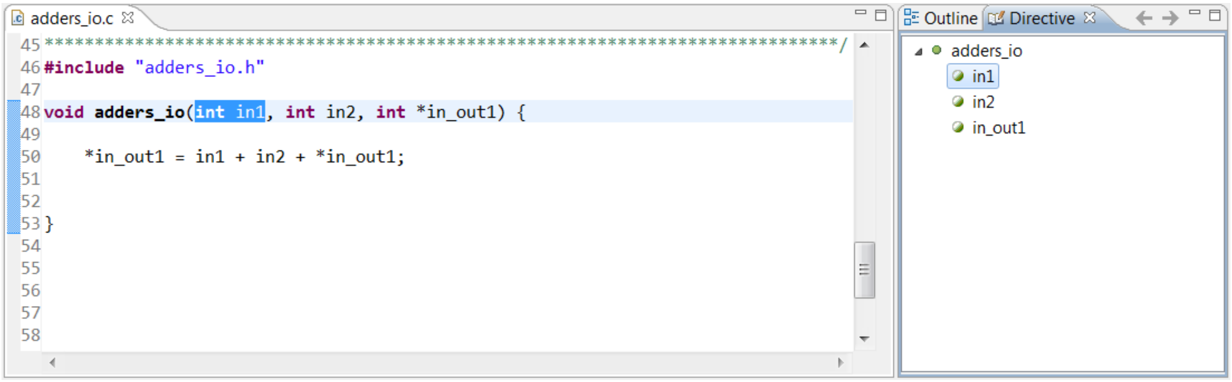


Figure 4-13: **Adding Port I/O Protocols**

3. Right-click and select **Insert Directives**.
4. When the Directives Editor opens leave the Directive 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 4-14.

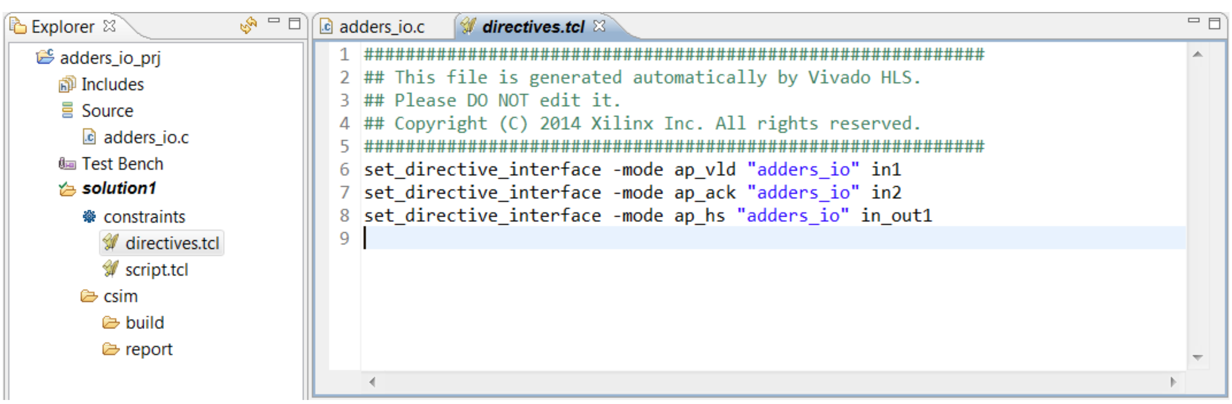


Figure 4-14: **Directives for Lab 2**

8. Synthesize the design.
9. Review the Interface summary when the report file opens (Figure 4-15).

adders_io.c | directives.tcl | adders_io_csynth.rpt

Interface

Summary

	Dir	Bits	Protocol	Source Object	C Type
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
in1	in	32	ap_vld	in1	scalar
in1_ap_vld	in	1	ap_vld	in1	scalar
in2	in	32	ap_ack	in2	scalar
in2_ap_ack	out	1	ap_ack	in2	scalar
in_out1_i	in	32	ap_hs	in_out1	pointer
in_out1_i_ap_vld	in	1	ap_hs	in_out1	pointer
in_out1_i_ap_ack	out	1	ap_hs	in_out1	pointer
in_out1_o	out	32	ap_hs	in_out1	pointer
in_out1_o_ap_vld	out	1	ap_hs	in_out1	pointer
in_out1_o_ap_ack	in	1	ap_hs	in_out1	pointer

Figure 4-15: **Interface Summary for Lab 2**

- 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.

Lab 3: Implementing Arrays as RTL Interfaces

Introduction

This exercise shows how array arguments on the top-level function interface can be implemented 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 array_io_prj`.
4. Open the source code as shown in Figure 4-16.

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`.

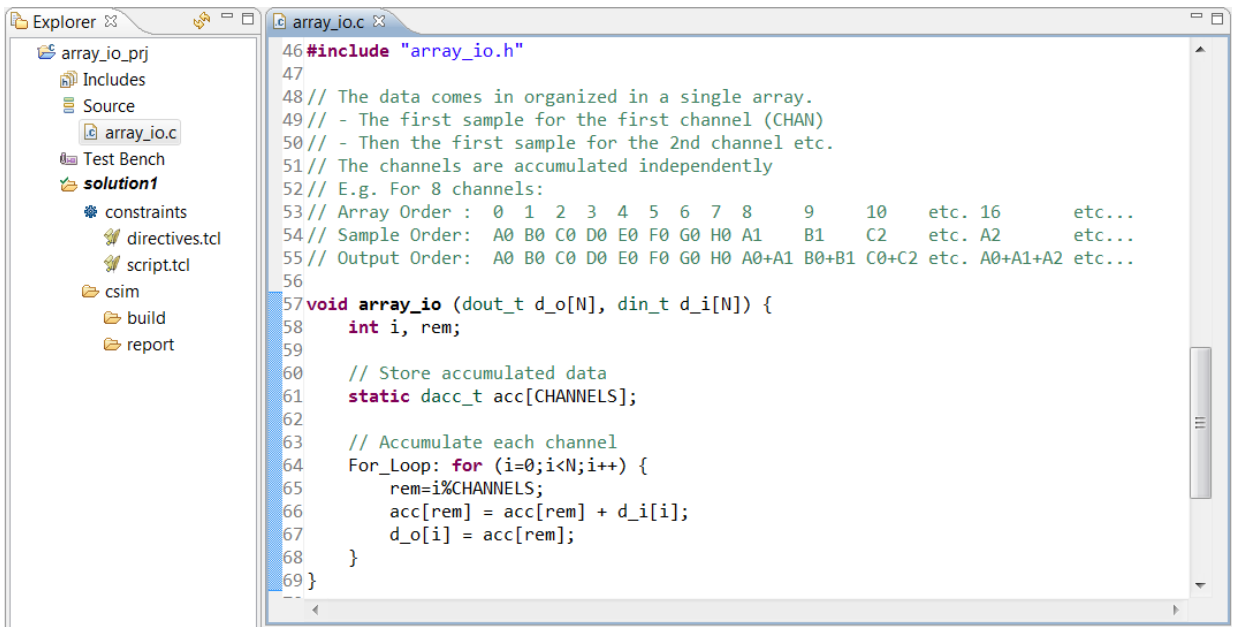


Figure 4-16: **C Code for Interface Synthesis Lab 3**

Step 2: Synthesize Array Function Arguments to RAM Ports

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

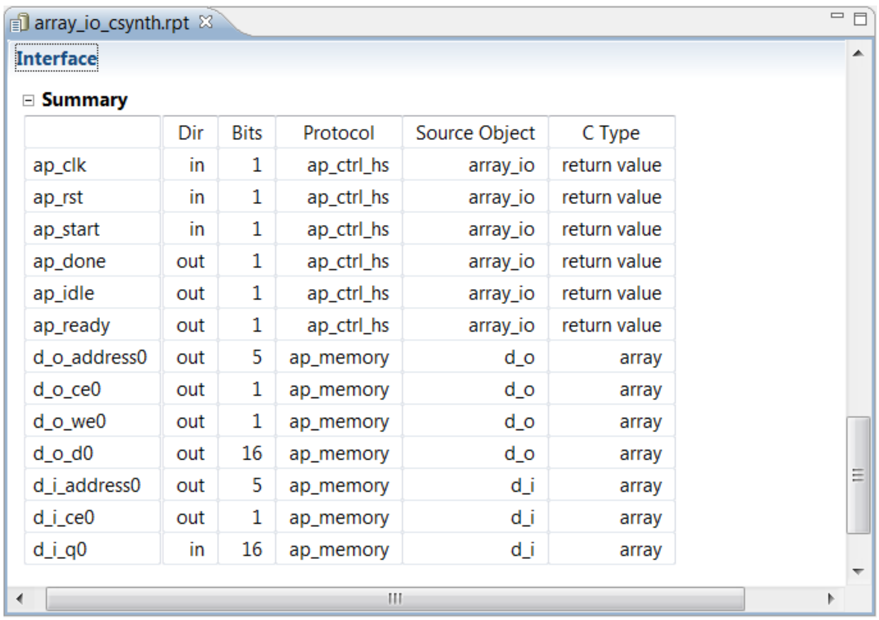
1. Synthesize the design and review the Interface summary when the report opens (Figure 4-17).

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 a 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.

In both cases, the data port is the width of the data values in the C source (16-bit integers in this case) and the width of the address port has been automatically sized to match the number of addresses that must be accessed (5-bit for 32 addresses).



	Dir	Bits	Protocol	Source Object	C Type
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_address0	out	5	ap_memory	d_o	array
d_o_ce0	out	1	ap_memory	d_o	array
d_o_we0	out	1	ap_memory	d_o	array
d_o_d0	out	16	ap_memory	d_o	array
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

Figure 4-17: **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 ports.

Step 3: Using Dual-Port RAM and FIFO Interfaces

High-Level Synthesis lets you specify a RAM interface as 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 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 **Finish**.
3. Ensure the C source code is visible in the Information pane.
4. In the Directive tab select **For_Loop**, and right-click to open the **Directives Editor** dialog box.
    a. In the Directives Editor dialog box activate the Directive drop-down menu at the top and select **UNROLL**.
    b. With the Directives Editor as shown in Figure 4-18, click **OK**.

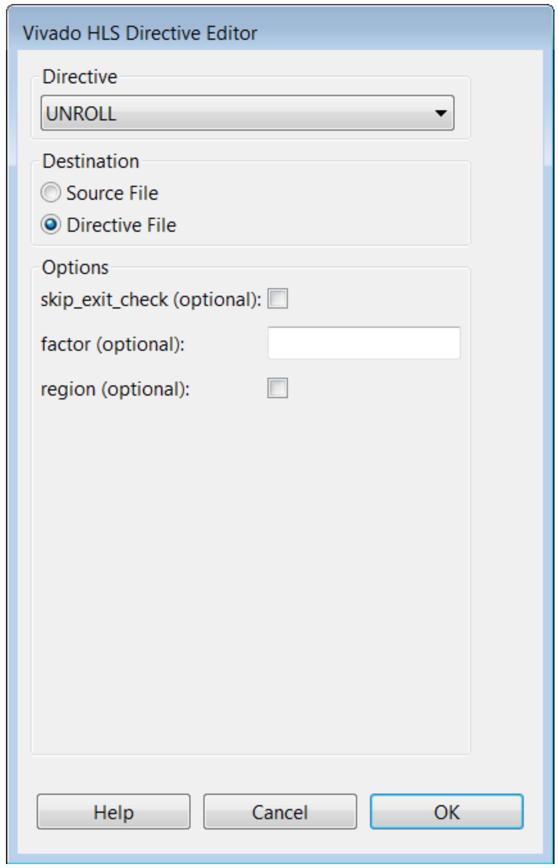


Figure 4-18: **Directives Editor to Unroll For_Loop**

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

   a. In the Directive tab, select **port d_i** and right-click to open the **Directives Editor** dialog box.

   b. In the Directives Editor activate the Directive drop-down menu at the top and select **RESOURCE**.

   c. Click the **core options** box and select **RAM_2P_BRAM**.

   d. Verify that the settings in the Directives Editor dialog box are as shown in Figure 4-19 and click **OK**.

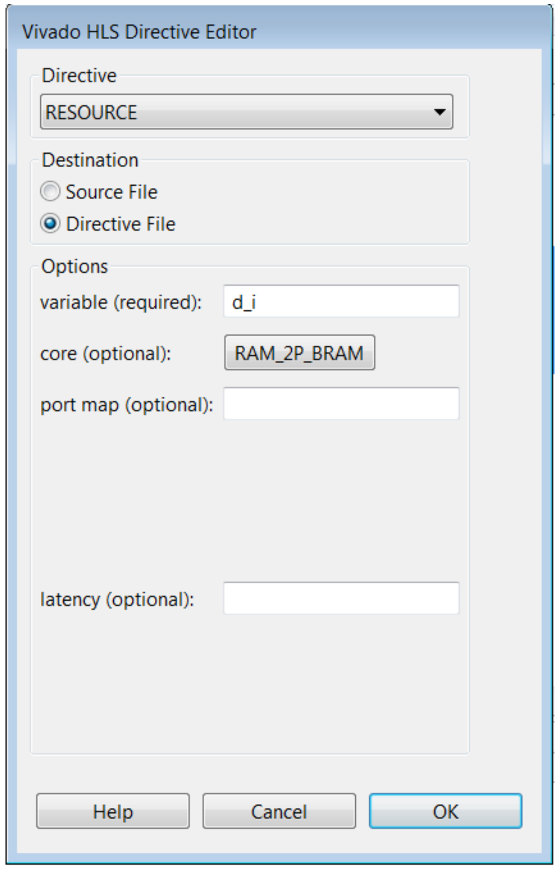


Figure 4-19: **Directives Editor for Specifying a Dual-port RAM**

6. Implement the output port using a FIFO interface.
   a. In the Directive tab, select port **d_o** and right-click to open the **Directives Editor** dialog box.
   b. In the Directives Editor, ensure the directive is **Interface**.
   c. From the Mode drop-down menu, select **ap_fifo**.
   d. Click **OK**.

The **Directive** tab shows the directives now applied to the design (Figure 4-20).

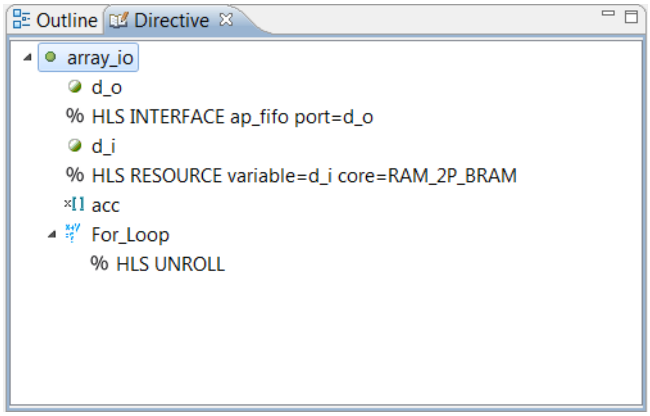
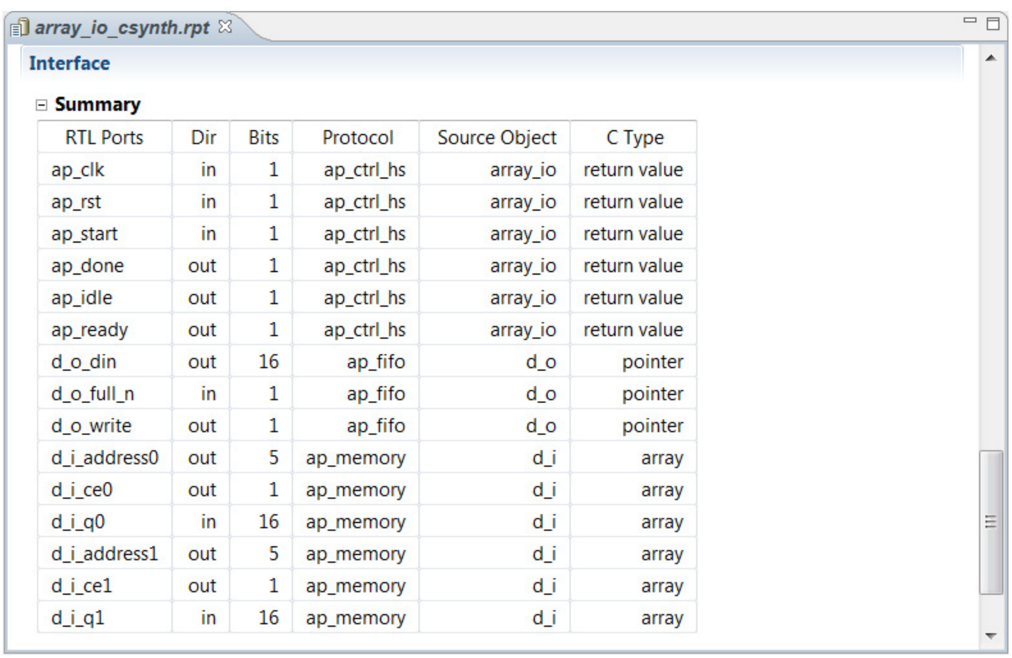


Figure 4-20: **Directives Summary for Lab 2 Solution**

7. Synthesize the design.

When the report opens in the Information pane, the Interface summary is as shown in Figure 4-21.

- 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.


array_io_csynth.rpt

Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
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 4-21: **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. Because the for-loop was unrolled, the logic in the loop is able to consume data at this rate. By default, each loop iteration is executed in turn. This implementation code limits the logic to one read on `d_i` in each iteration. Unrolling the loops allows more reads to be performed (but creates N copies of the logic). 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 **Finish**. This includes copying existing directives from solution2.
3. Ensure the C source code is visible in the Information pane.
4. In the Directive tab, select d_o and right-click to open the **Directives Editor** dialog box.
   a. In the Directives Editor dialog box activate the Directive drop-down menu at the top and select **ARRAY_PARTITION**.
   b. Activate the options type drop down 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 4-22, click **OK**.

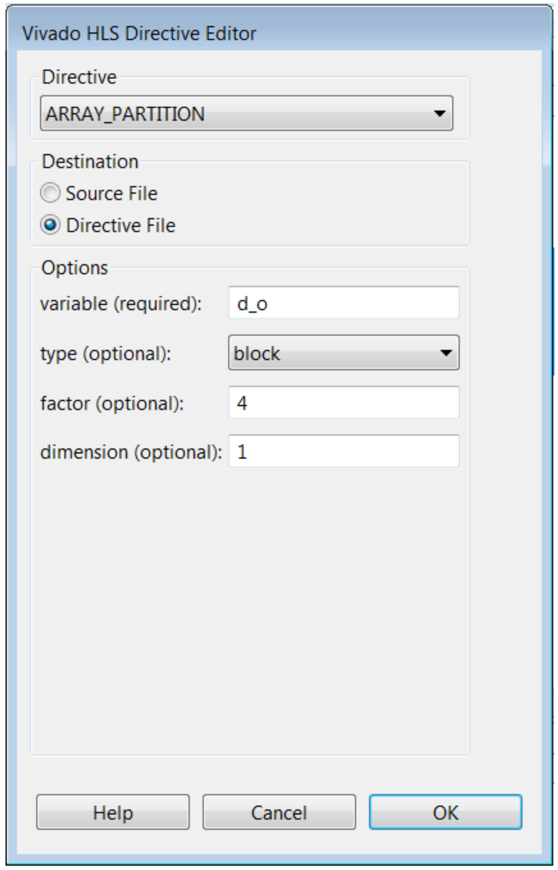


Figure 4-22: **Directives Editor for Partitioning Array d_o**

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

5. In the Directive 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 4-23).

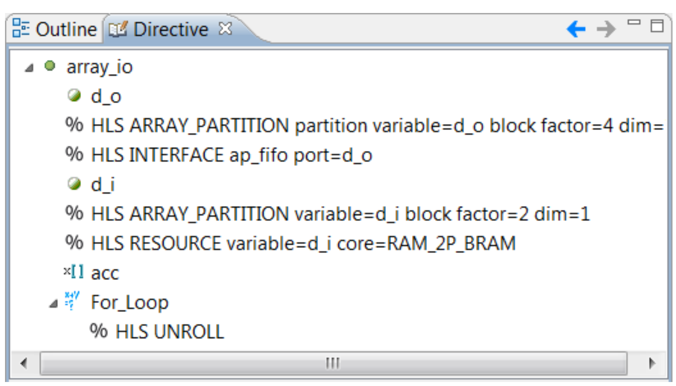


Figure 4-23: **Directives Summary for Lab 2 Solution3**

6. Synthesize the design.

When the report opens in the Information pane, the Interface summary is as shown in Figure 4-24.

- 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 four separate RAM interfaces, confirm a partition factor for `d_i` is two and not four).

array_io_csynth.rpt

Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
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 4-24: **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 eight inputs at once.

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

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 **Finish** 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 4-25.
5. Right-click and select **Modify Directive**.

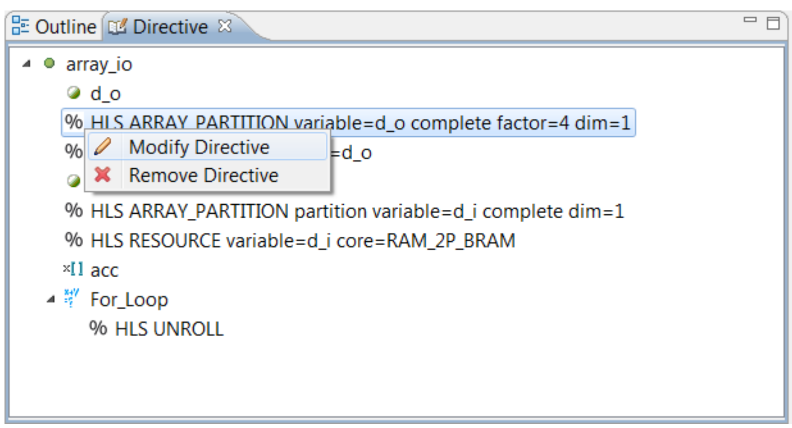


Figure 4-25: **Modifying the Directive for d_o**

6. In the Directives Editor dialog box:
   a. In the Factor dialog box, delete the value 4. Since this array will be completely partitioned into registers, the partitioning factor is no longer relevant. (If you leave it there, it will be ignored).
   b. Activate the type (optional) drop down and modify the partitioning style to **Complete**.
   c. With the Directives Editor as shown in Figure 4-26, click **OK**.

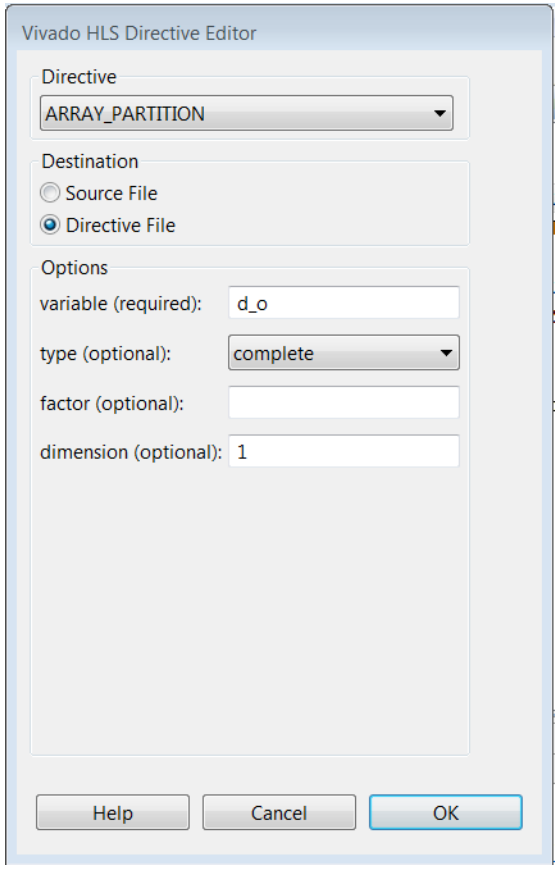


Figure 4-26: **Directives Editor for Partitioning Array d_o**

7. In the Directive tab, select `d_i` and repeat the previous step to completely partition the `d_i` array.
8. In the Directive tab, select the **RESOURCE** directive on `d_i`, right-click with the mouse and select **Remove Directive**. If the array is partitioned into individual elements, it cannot be assigned to a block RAM.

The Directives tab shows the directives now applied to the design (Figure 4-27).

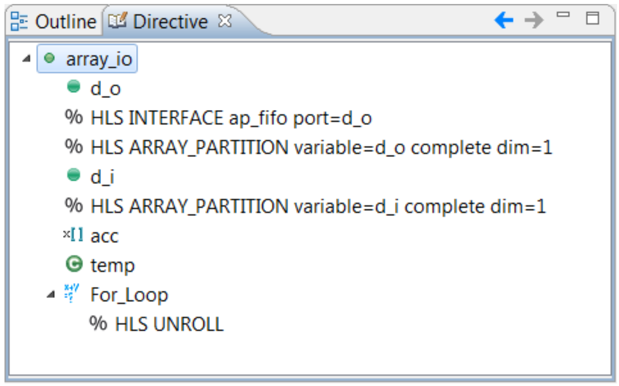


Figure 4-27: **Directives Summary for Lab 2 Solution3**

9. Synthesize the design.
10. 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 32 separate FIFO interfaces.
- Argument `d_i` has been implemented as 32 separate scalar ports. Because the default interface for input scalars is not in the 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 is performance across all four solutions.

11. Select **Compare Reports** from the toolbar or the Project menu to open a comparison of the solutions.
12. In the Solution Selection dialog box, add each of the four solutions to the Selected Solutions pane (Figure 4-28).
13. Click **OK**.

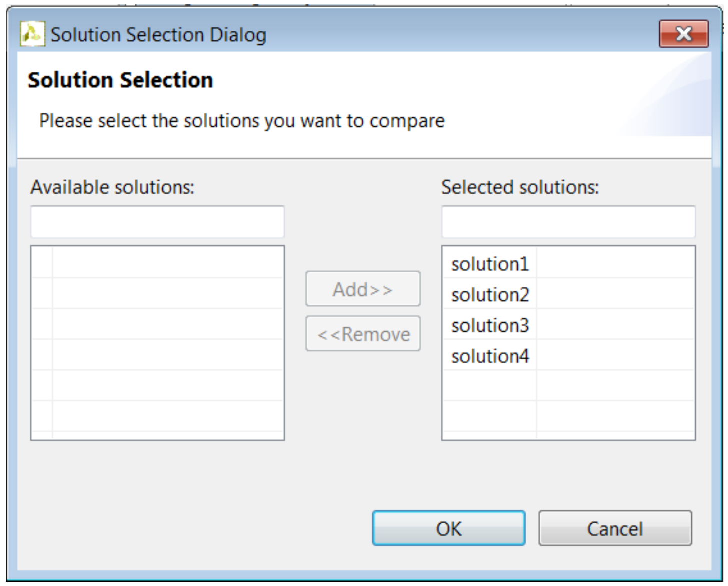
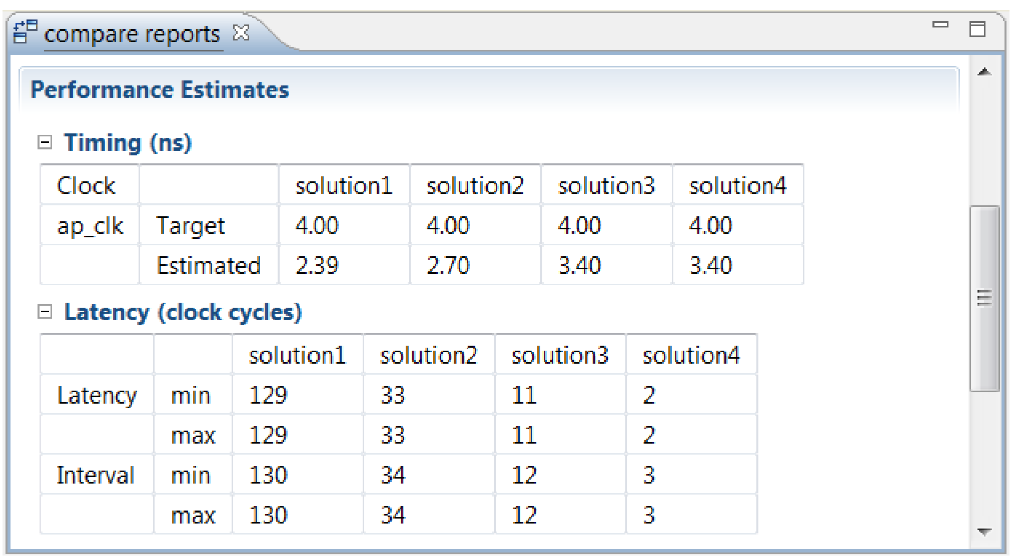


Figure 4-28: **Compare All Solutions for Lab 3**

When the solutions comparison report opens (Figure 4-29), 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.)


compare reports

Performance Estimates

Timing (ns)

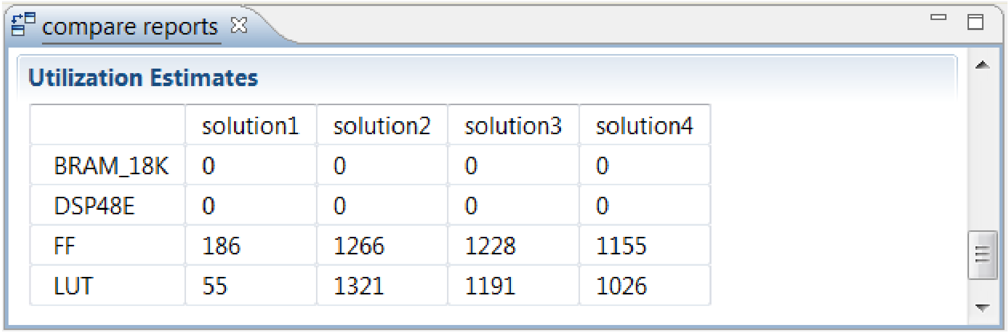
Clock		solution1	solution2	solution3	solution4
ap_clk	Target	4.00	4.00	4.00	4.00
	Estimated	2.39	2.70	3.40	3.40

Latency (clock cycles)

		solution1	solution2	solution3	solution4
Latency	min	129	33	11	2
	max	129	33	11	2
Interval	min	130	34	12	3
	max	130	34	12	3

Figure 4-29: **Performance Comparisons for All Lab 3 Solutions**

Scroll further down the comparison report (Figure 4-30) and note that solutions with more I/O ports (solutions 2, 3, and 4), allows more parallel processing, but also use considerably more resources.



Utilization Estimates

	solution1	solution2	solution3	solution4
BRAM_18K	0	0	0	0
DSP48E	0	0	0	0
FF	186	1266	1228	1155
LUT	55	1321	1191	1026

Figure 4-30: **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 AXI4 interfaces to the design.

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

Lab 4: Implementing AXI4 Interfaces

Introduction

This exercise explains how to specify AXI4 bus interfaces for the I/O ports. In addition to adding AXI4 interfaces this exercise also shows how to create an optimal design by using interface and logic directives together.

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_interfaces_prj`.
4. Open the source code as shown in Figure 4-31.

axi_interfaces.c

```c
#include "axi_interfaces.h"

// The data comes in organized in a single array.
// - The first sample for the first channel (CHAN)
// - Then the first sample for the 2nd channel etc.
// The channels are accumulated independently
// E.g. For 8 channels:
// Array Order :  0  1  2  3  4  5  6  7  8     9     10    etc. 16       etc...
// Sample Order:  A0 B0 C0 D0 E0 F0 G0 H0 A1    B1    C2    etc. A2       etc...
// Output Order:  A0 B0 C0 D0 E0 F0 G0 H0 A0+A1 B0+B1 C0+C2 etc. A0+A1+A2 etc...

void axi_interfaces (dout_t d_o[N], din_t d_i[N]) {
    int i, rem;

    // Store accumulated data
    static dacc_t acc[CHANNELS];

    // Accumulate each channel
    For_Loop: for (i=0;i<N;i++) {
        rem=i%CHANNELS;
        acc[rem] = acc[rem] + d_i[i];
        d_o[i] = acc[rem];
    }
}
```

Figure 4-31: **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 with AXI4-Stream Interfaces

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® Design Suite User Guide: High-Level Synthesis* (UG902) [Ref 2], but basically means each array element is, in turn, sorted into a different partition.

In this exercise, you specify the array arguments to be implemented as AXI4-Stream interfaces. If the arrays are partitioned into channels, you can 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 Directive tab, select **d_o** and right-click to open the Directives Editor dialog box.
   a. Select the **Directive** drop-down menu at the top and select **ARRAY_PARTITION**.
   b. Click the **Directive** 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.)
   d. With the Directives Editor dialog box filled in as shown in Figure 4-32, click **OK**.

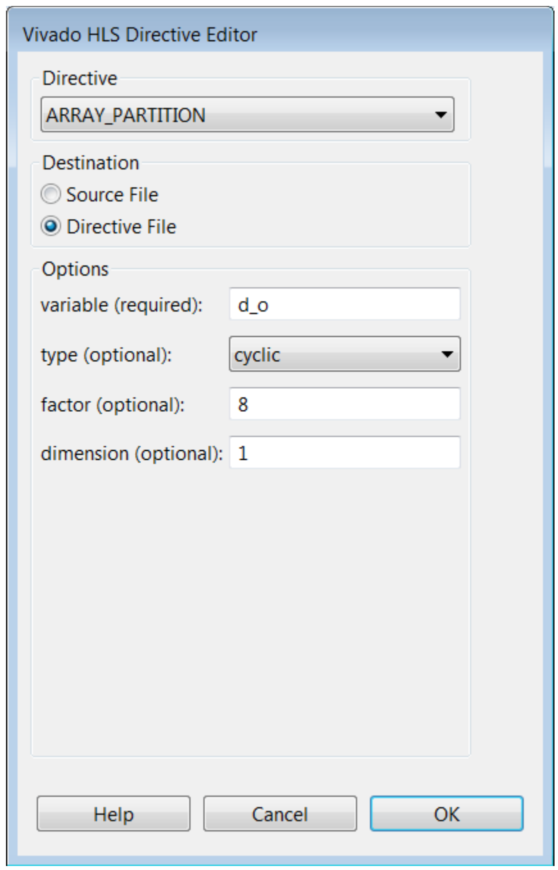


Figure 4-32: **Directives Editor for Cyclic Partitioning**

3. In the Directive tab, select **d_o** again and right-click to open the Directives Editor dialog box.
   a. Activate the **Directive** drop-down menu at the top and select **INTERFACE**.
   b. Click the **Mode** drop-down menu to specify an **axis** interface.
   c. Click **OK**.
4. In the Directive tab, select **d_i** and repeat steps 2 and 3 above.
   a. Apply **cyclic** partitioning with a factor of **8**.
   b. Apply an **axis** interface.

5. Next, partially unroll and pipeline the for-loop:

   a. In the Directive tab, select **For_Loop** and right-click to open the **Directives Editor** dialog box.

   b. Activate the **Directive** drop-down menu at the top and select **UNROLL**.

   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).

   Click **OK**.

   a. In the Directive tab, select **For_Loop** again and right-click to open the **Directives Editor** dialog box.

   Activate the **Directive** drop-down menu at the top and select **PIPELINE**.

   Leave the Interval blank and let it default to 1.

   Select **enable loop rewinding**.

   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 4-33. Be sure to check all options are correctly applied. If not, double-click the directive to re-open the **Directives Editor**.

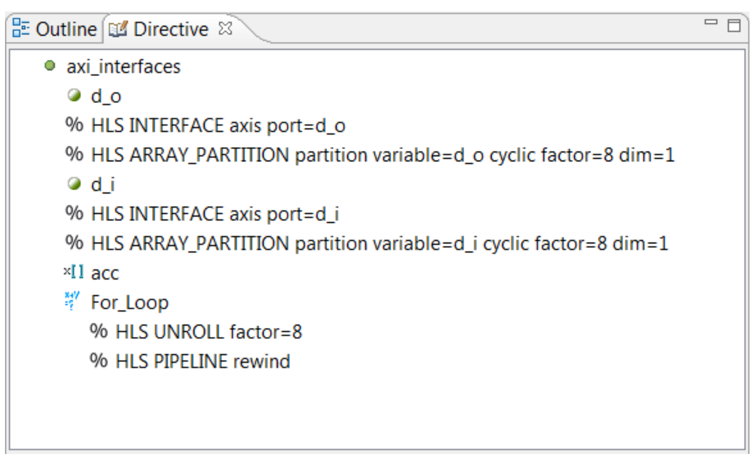


Figure 4-33: **Directives Tab for Lab 4 Solution1**

6. Synthesize the design.

When the report opens in the information pane, confirm both d_i and d_o are implemented as eight separate AXI4-Stream ports.

7. In the performance section of the report, confirm that the for-loop processes one sample every clock cycle (Interval 1) with a latency of 3 (and max 4), and that the design has less area than solutions 2, 3, or 4 in Lab 3 (Figure 4-33).

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

In this exercise, 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 a new solution.
2. Accept the defaults and click **Finish**. This includes copying existing directives from solution1.
3. Ensure the C source code is visible in the Information pane.
4. In the Directive tab, select the top-level function **axi_interfaces** and right-click to open the **Directives Editor** dialog box.
    a. Activate the **Directive** drop-down menu at the top and select **INTERFACE**.
    b. Activate the **mode** drop-down menu and select **s_axilite**. This specifies that the ports associated with the function return (the block-level I/O ports) are implemented as an AXI4Lite interface. Since the default mode for the function return is ap_ctrl_hs, there is no requirement to specify this I/O protocol.
    c. Click **OK**.

The **Directives** tab appears, as shown in Figure 4-34.

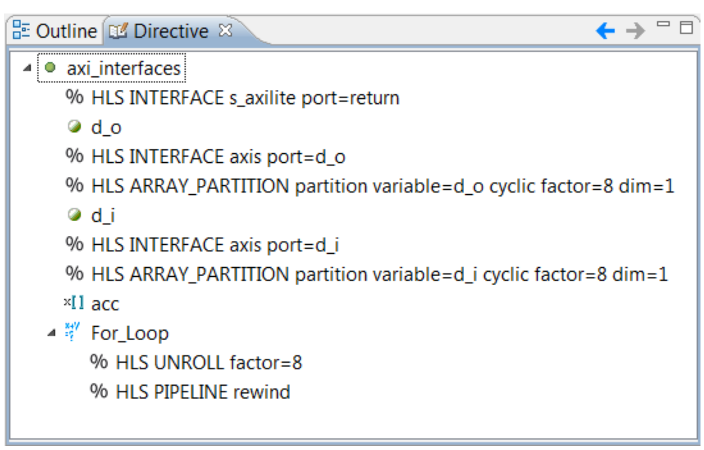


Figure 4-34: **Directives for Specifying AXI4 Interfaces**

5. Synthesize the design.

When the report opens, review the interface summary to confirm the block-level I/O protocol ports (`ap_start`, `ap_done`, etc.) have been replaced with an AXI4Lite interface.

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. Because you used the Vivado IP Catalog format, the package is in the ip folder.

The `ip` folder includes the `drivers` subfolder, as shown in Figure 4-35.

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).

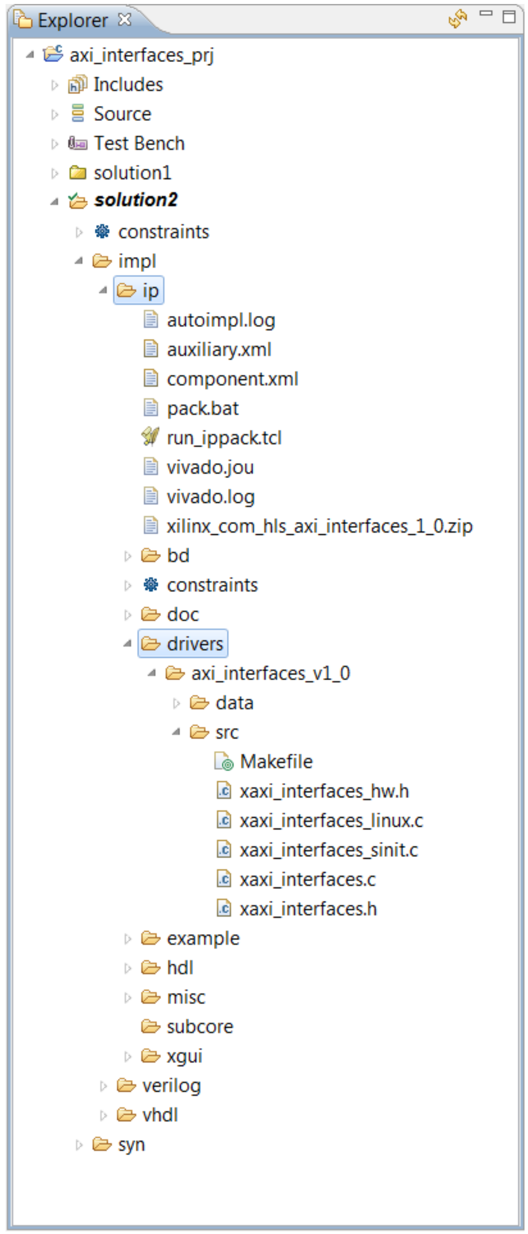


Figure 4-35: **IP Package with AXI4 Interfaces**

8. Double-click the `xaxi_interfaces_hw.h` file to open it in the Information pane.

This shows the addresses to access and control the block-level interface signals For example, setting control register 0x0 bit 0 to the value 1 will enable the ap_start port, or alternatively, setting bit 7 will enable the auto-restart and the design will re-start automatically at the end of each transaction.

The remaining C driver files are used to integrate control of the AXI4 Slave Lite interface into the code running on a CPU or microcontroller and are included in the packaged IP.

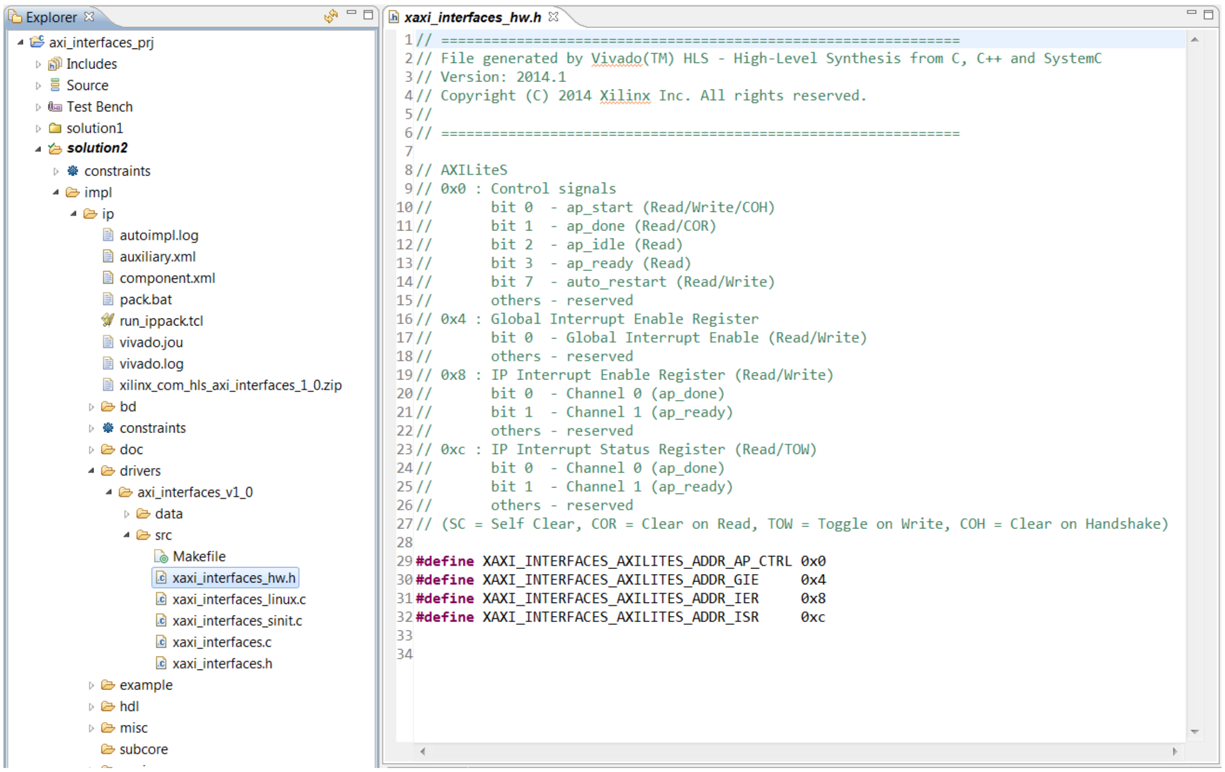


Figure 4-36: **IP Software Driver Files**

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.

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 data-types, allowing you to model variables using any (arbitrary) width.

This tutorial consists of a two lab exercises:

Lab 1 Description

Synthesize a design using floating-point types and review the results. The design uses standard C++ floating-point types.

Lab 2 Description

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 Locating the Tutorial Design Files.

This tutorial uses the design files in the tutorial directory `Vivado_HLS_Tutorial\Arbitary_Precision`.

Lab 1: Arbitrary Precision

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 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 2015.3 > Vivado HLS > Vivado HLS 2015.3 Command Prompt** (Figure 5-1).
    b. On Linux, open a new shell.

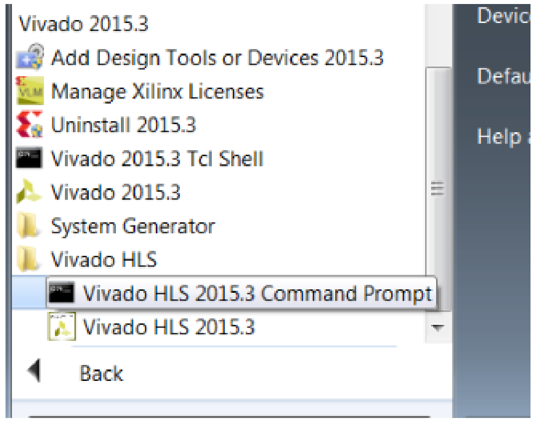


Figure 5-1: **Vivado HLS Command Prompt**

2. In the command prompt window (Figure 5-2), 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 5-2:

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

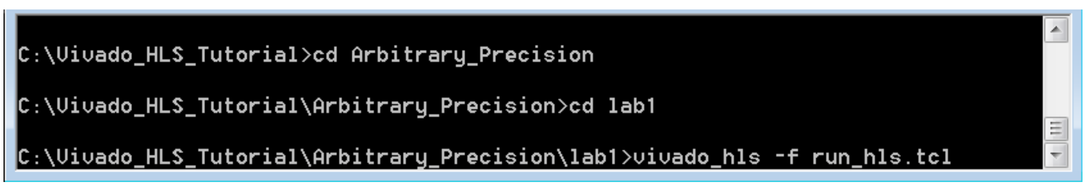


Figure 5-2: **Setup the Tutorial Project**

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 5-3.

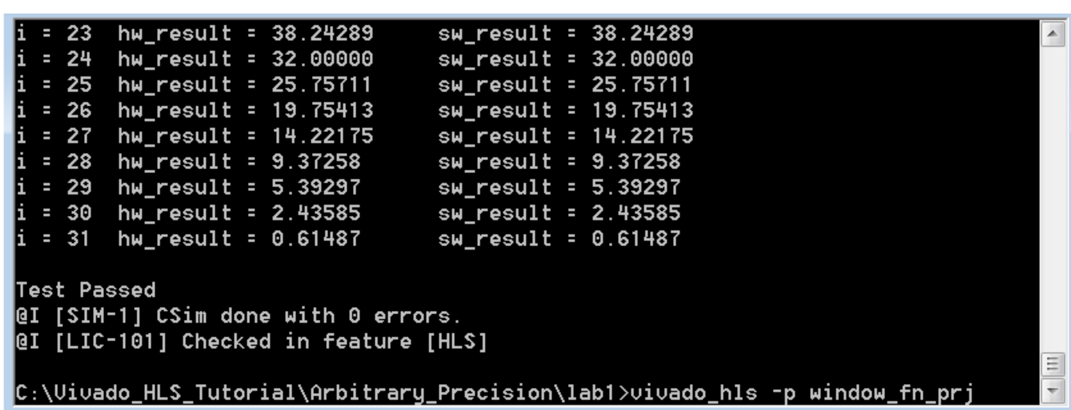


Figure 5-3: **Initial Project for Arbitrary Precision Lab1**

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 5-4.

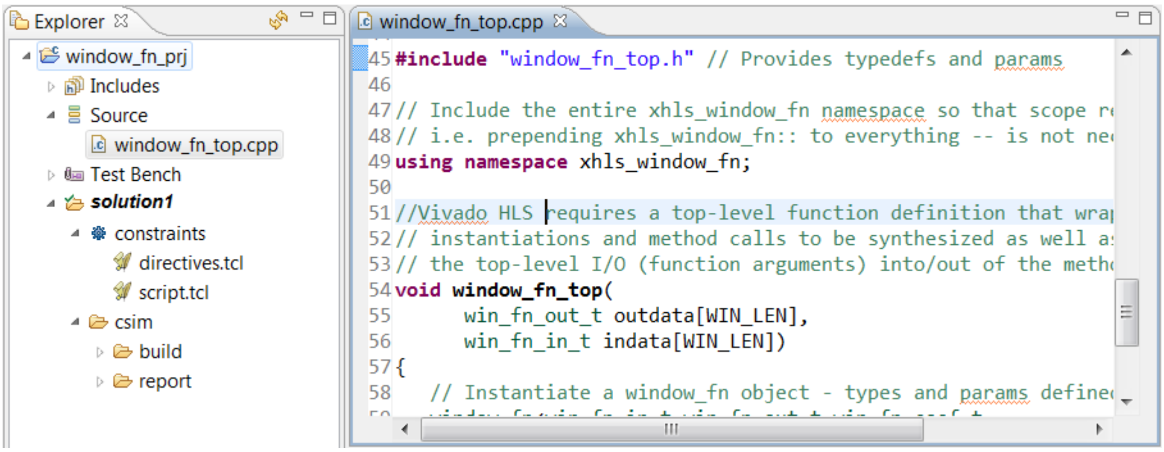


Figure 5-4: **C Code for C Validation Lab 3**

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 5-5).

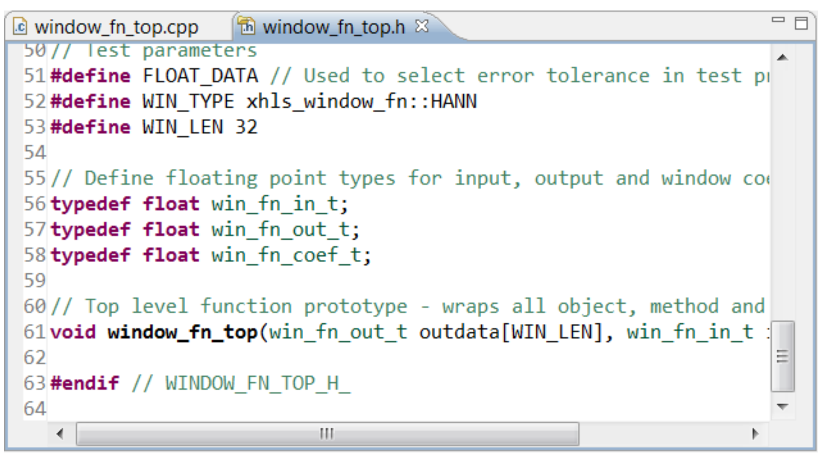


Figure 5-5: **Type Definitions for C Validation Lab 3**

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 (+, -, *, %).

When using math functions from math.h or cmath.h, see the *Vivado® Design Suite User Guide: High-Level Synthesis* (UG902) [Ref 2] for details on which math functions are supported for synthesis.

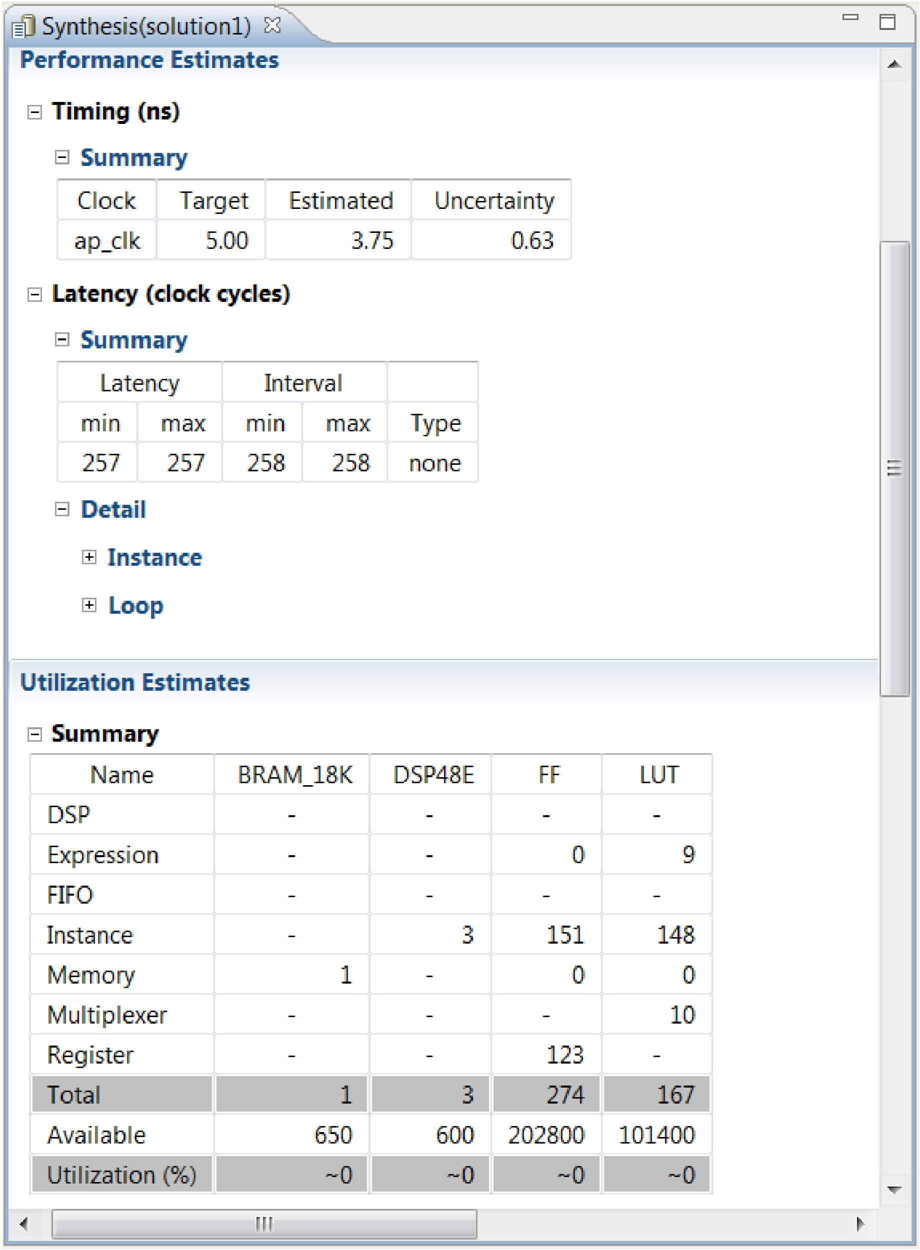
4. Click the **Run C Simulation** toolbar button to open the C Simulation Dialog box.
5. Accept the default setting (no options selected) and click **OK**.

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 Synthesis** toolbar button to synthesize the design to RTL.

When synthesis completes, the synthesis report opens automatically. Figure 5-6 shows the synthesis report.



Synthesis(solution1)

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	5.00	3.75	0.63

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
257	257	258	258	none

Detail

Instance

Loop

Utilization Estimates

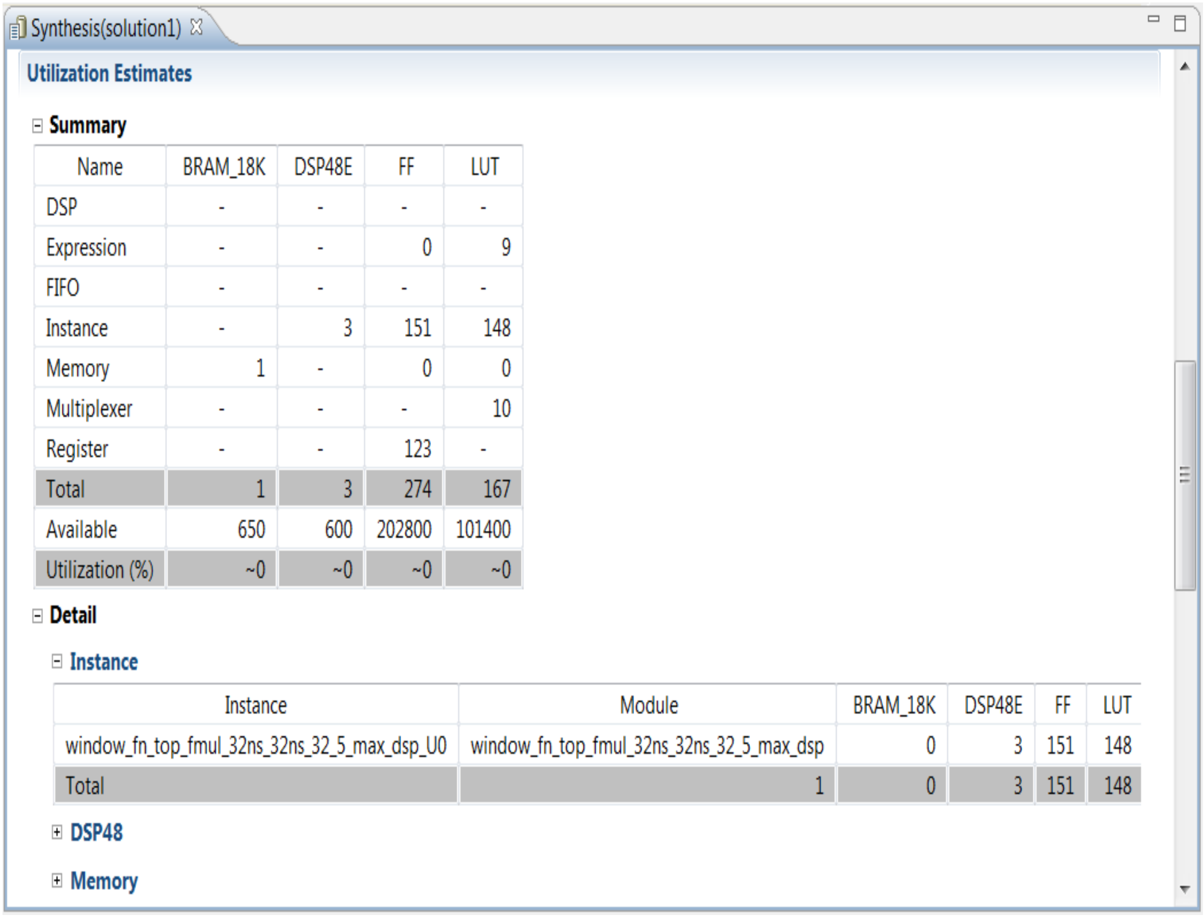
Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	9
FIFO	-	-	-	-
Instance	-	3	151	148
Memory	1	-	0	0
Multiplexer	-	-	-	10
Register	-	-	123	-
Total	1	3	274	167
Available	650	600	202800	101400
Utilization (%)	~0	~0	~0	~0

Figure 5-6: **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 5-7).


Synthesis(solution1)

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	9
FIFO	-	-	-	-
Instance	-	3	151	148
Memory	1	-	0	0
Multiplexer	-	-	-	10
Register	-	-	123	-
Total	1	3	274	167
Available	650	600	202800	101400
Utilization (%)	~0	~0	~0	~0

Detail

Instance

Instance	Module	BRAM_18K	DSP48E	FF	LUT
window_fn_top_fmul_32ns_32ns_32_5_max_dsp_U0	window_fn_top_fmul_32ns_32ns_32_5_max_dsp	0	3	151	148
Total	1	0	3	151	148

DSP48

Memory

Figure 5-7: **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 5-8) 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 Chapter 6, Design Analysis tutorial. 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.

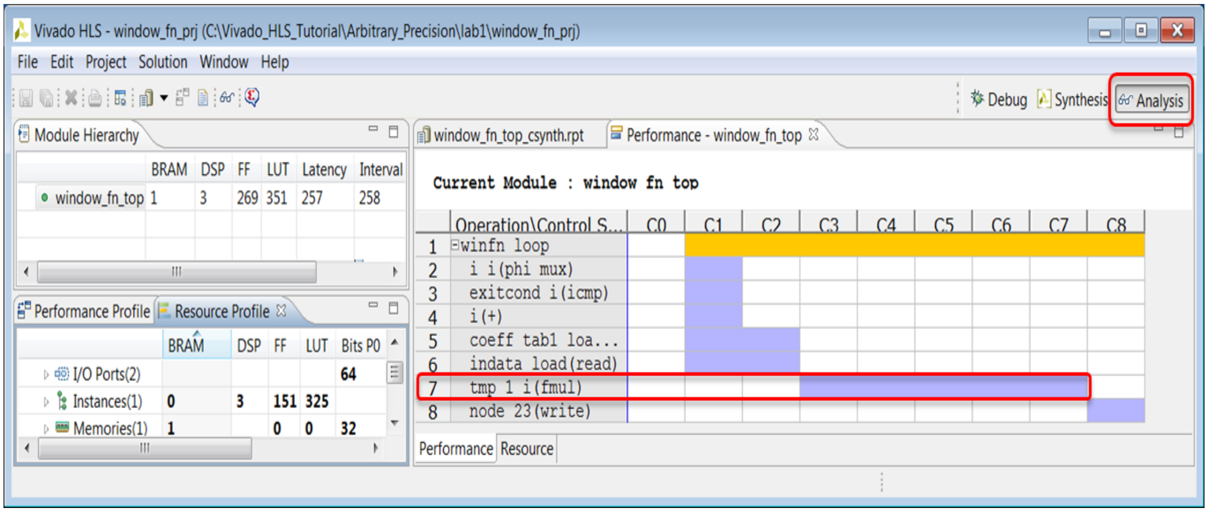


Figure 5-8: **Performance Details for Floating Point Design**

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

Lab 2: Arbitrary Precision

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 5-9.
2. Create a new Vivado HLS project by typing `vivado_hls -f run_hls.tcl`.

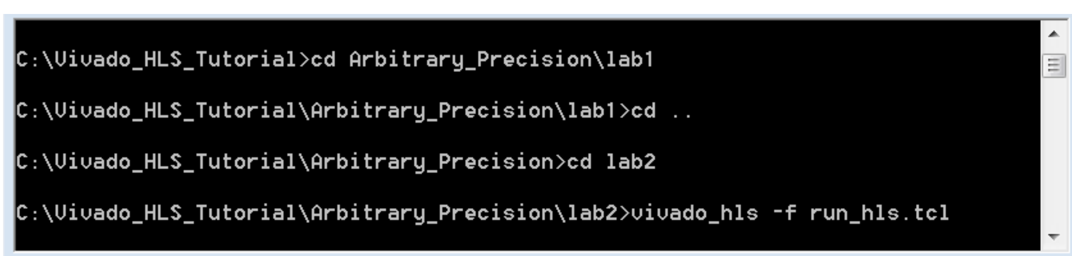


Figure 5-9: **Setup for Interface Synthesis Lab 2**

3. Open the Vivado HLS GUI project by typing `vivado_hls -p window_fn_prj`.
4. Open the **Source** folder in the Explorer pane and double-click **window_fn_top.cpp** to open the code as shown in Figure 5-10.

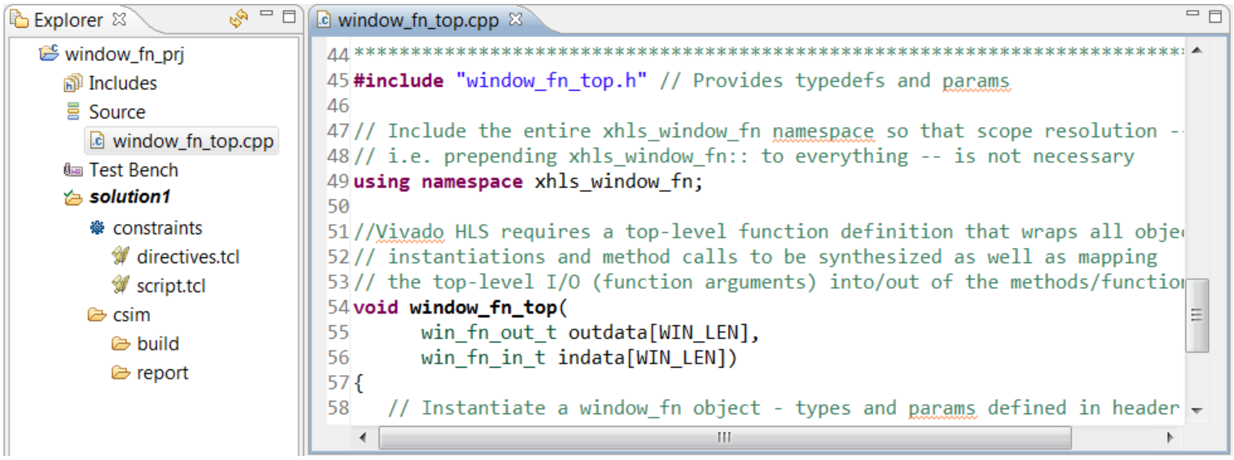


Figure 5-10: **C Code for Arbitrary Precision Lab 2**

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 5-11).

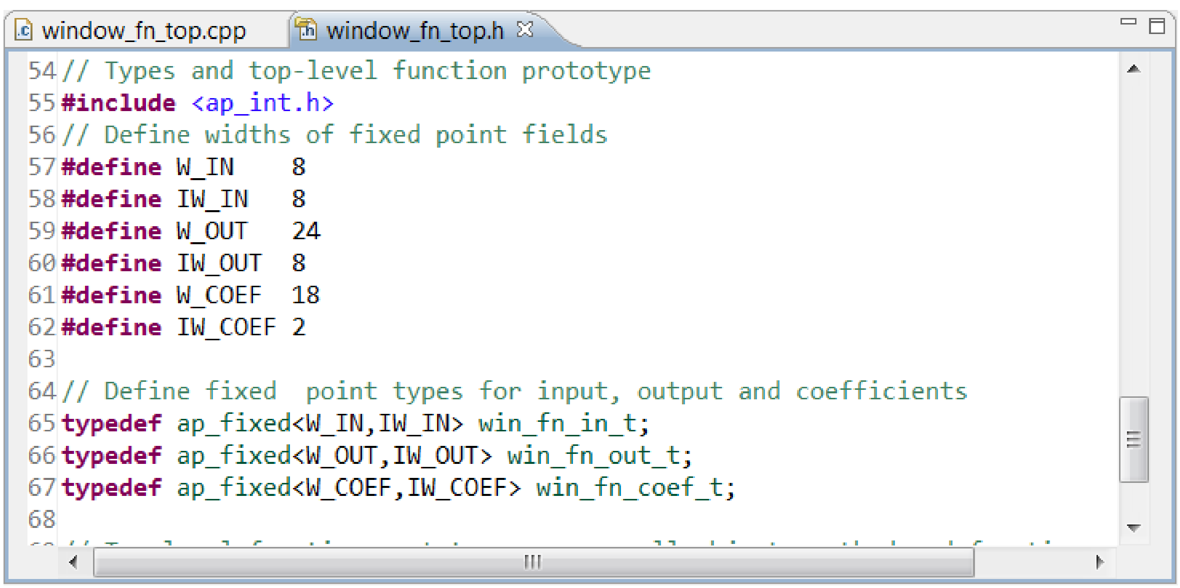


Figure 5-11: **Type Definitions for Arbitrary Precision Lab 2**

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 fewer clock cycles to complete.

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.

7. 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 5-12.

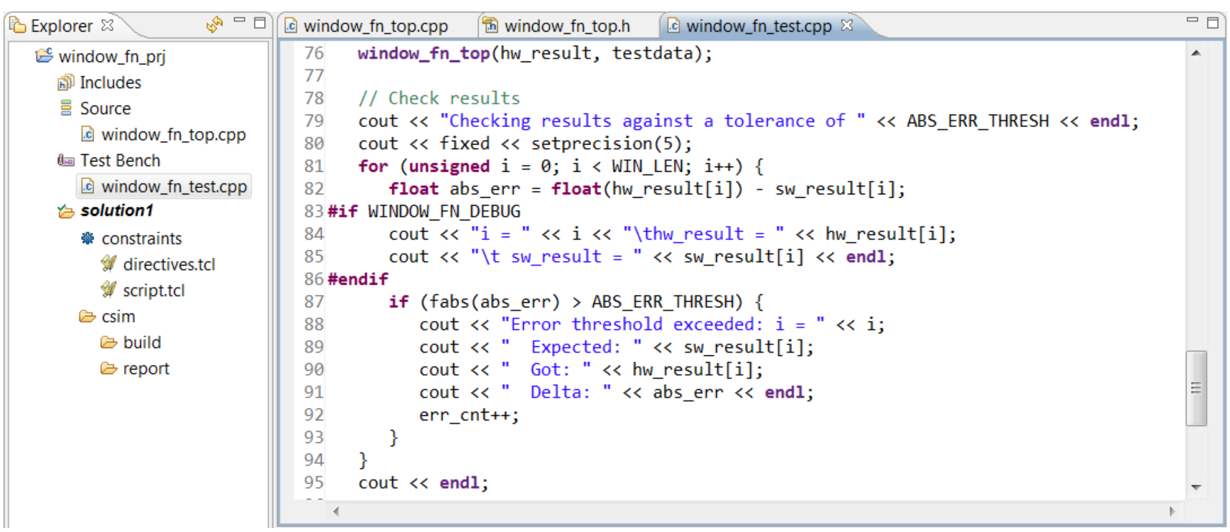


Figure 5-12: **Test Bench for Arbitrary Precision Lab 2**

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.

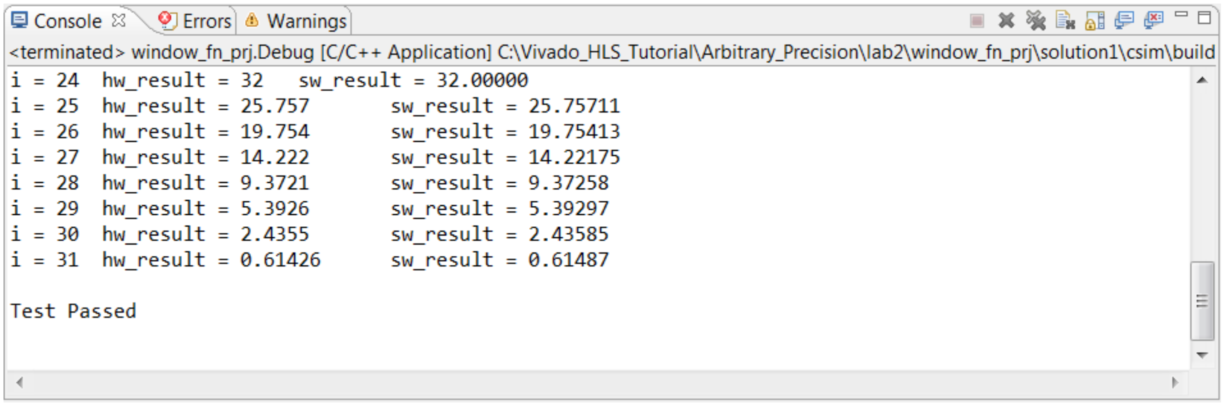


Figure 5-13: **C Simulation Results for Fixed Point Types**

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 5-14 shows the synthesis report.

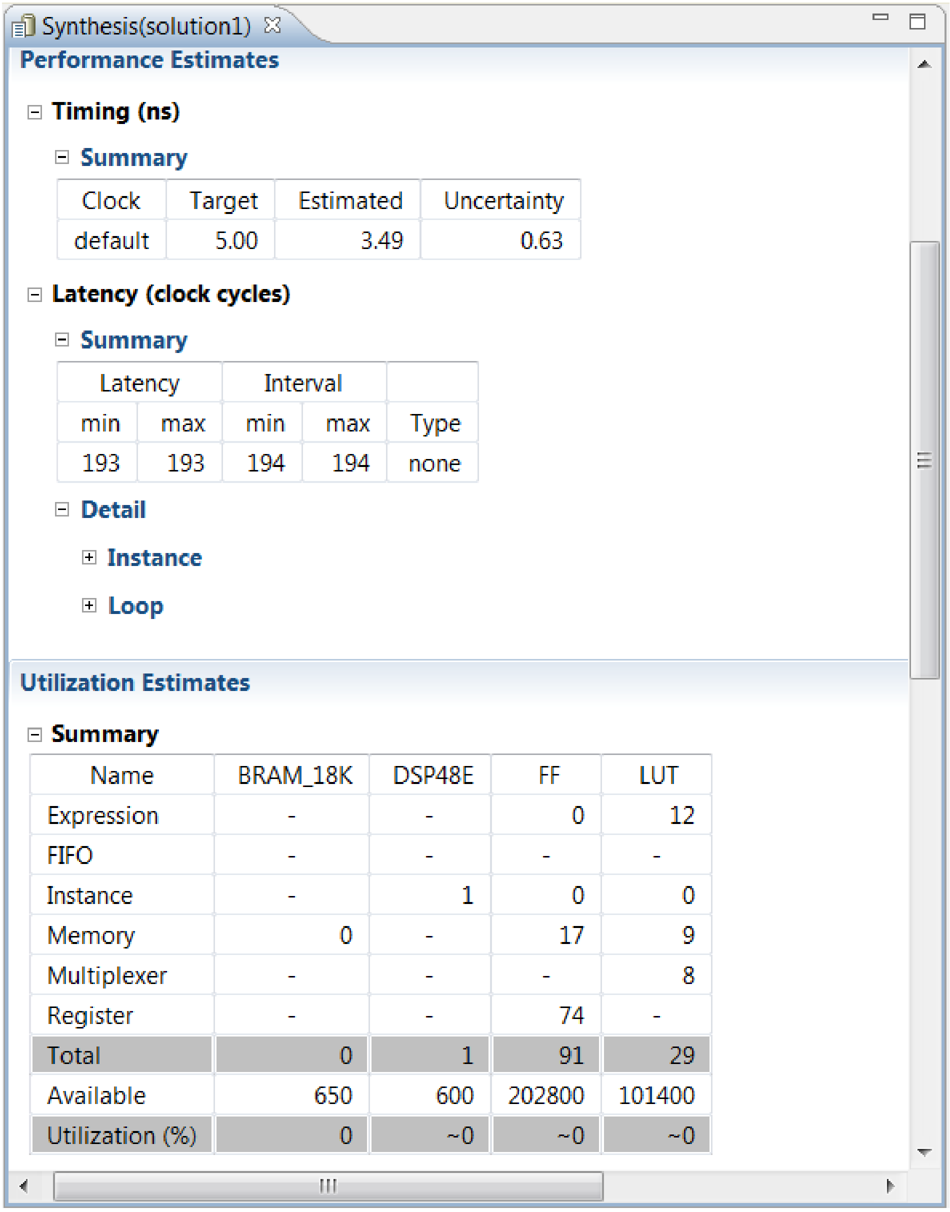


Figure 5-14: **Synthesis Report for Fixed Point Design**

Note that through use of arbitrary precision types, you have reduced both the latency and the area (by 50% and 80% respectively), and the operations in the RTL hardware are no larger than necessary. Since the total number of bits in the memory is now less than 1024-bit, it is now automatically implemented with LUTs and FFs rather than with a block RAM.

2. Scroll down the report to the Interface summary (Figure 5-15).

Figure 5-15 shows the data ports are now 8-bit and 24-bit.

window_fn_top_csynth.rpt

Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
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 5-15: **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 data-types.

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.

Lab 1 Description

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. See the information in Locating the Tutorial Design Files.

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.
   - On Windows click **Start** > **All Programs** > **Xilinx Design Tools** > **Vivado 2015.3** > **Vivado HLS** > **Vivado HLS 2015.3 Command Prompt** (Figure 6-1).
   - On Linux, open a new shell.

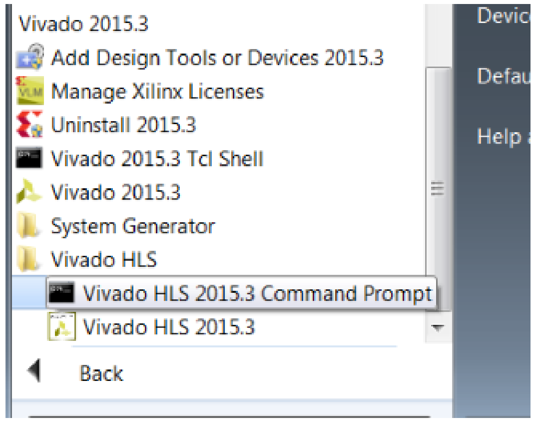


Figure 6-1: **Vivado HLS Command Prompt**

2. Using the command prompt window (Figure 6-2), 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 6-2.

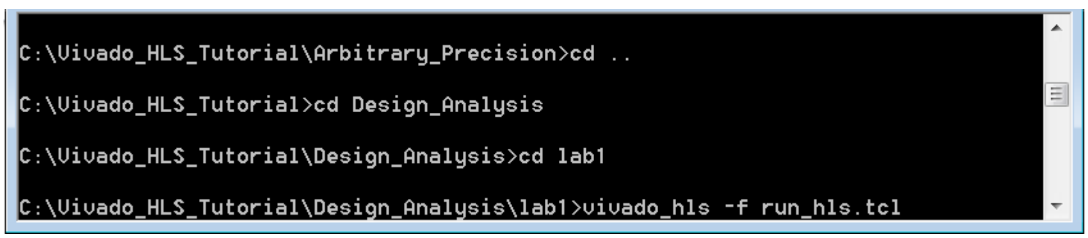


Figure 6-2: **Setup the Design Analysis Tutorial Project**

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 6-3.

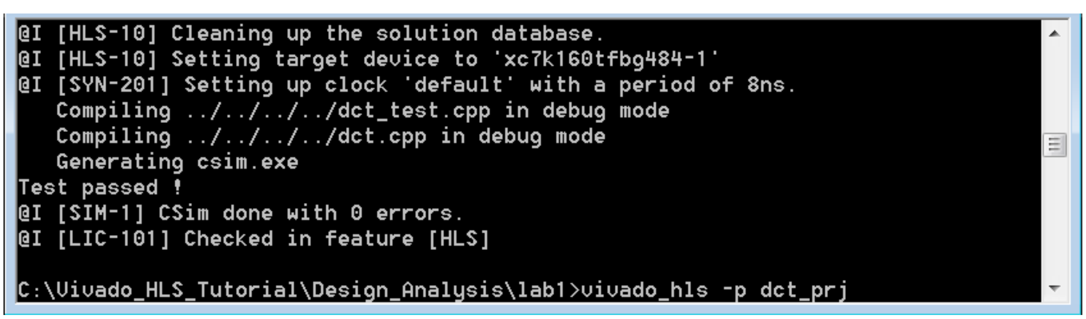


Figure 6-3: **Open Design Analysis Project for Lab 1**

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 6-4 shows an overview of this code.

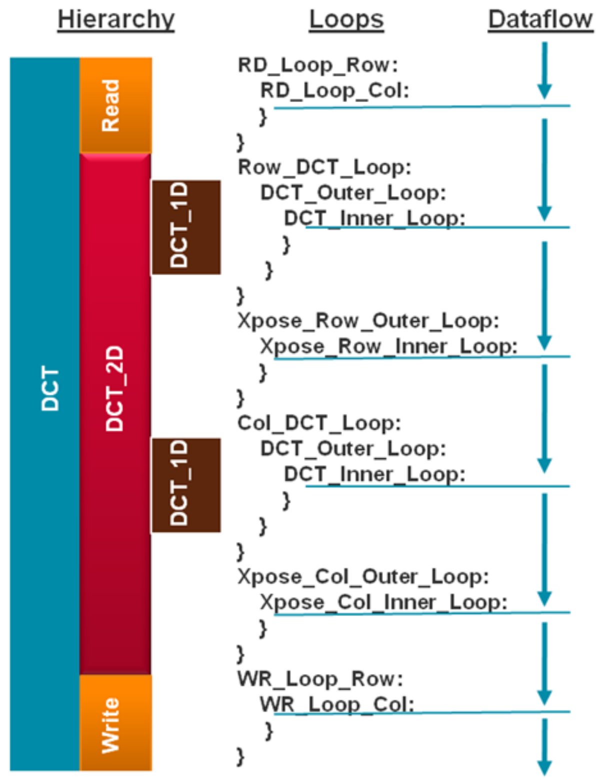


Figure 6-4: **Overview of the DCT Design**

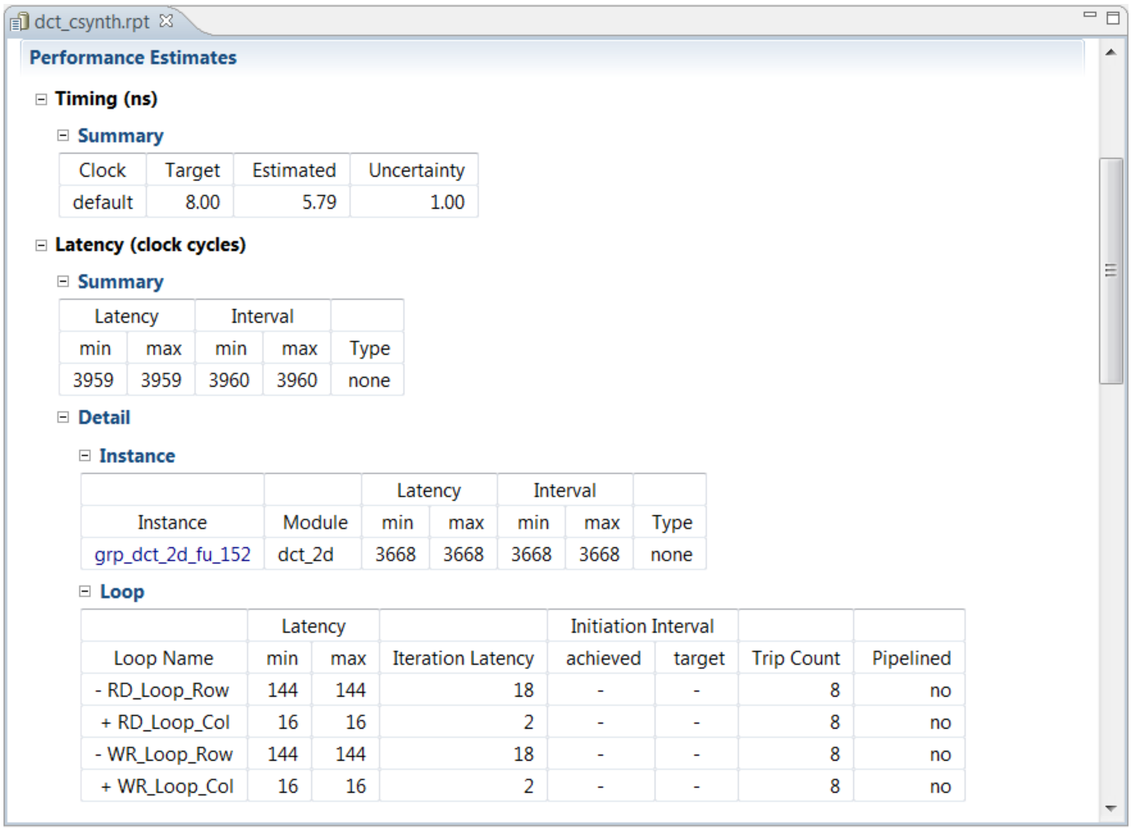
- The left side of Figure 6-5 shows the code hierarchy.
  - Top-level function dct has three sub-functions: read_data, dct_2d and write_data.
  - Function dct_2d has a single sub-function dct_1d.
- The center of Figure 6-5 shows loops inside each of the functions.
- The right side of Figure 6-5 shows 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 completes, 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.

- Click the **Run C Synthesis** 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 6-5 shows the performance section of the report.


dct_csynth.rpt

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
default	8.00	5.79	1.00

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
3959	3959	3960	3960	none

Detail

Instance

		Latency		Interval		
Instance	Module	min	max	min	max	Type
grp_dct_2d_fu_152	dct_2d	3668	3668	3668	3668	none

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- RD_Loop_Row	144	144	18	-	-	8	no
+ RD_Loop_Col	16	16	2	-	-	8	no
- WR_Loop_Row	144	144	18	-	-	8	no
+ WR_Loop_Col	16	16	2	-	-	8	no

Figure 6-5: **Report for Initial DCT Design**

Figure 6-6 highlights the following information.

- The clock frequency of 8 ns has been met.
- The top-level design takes 3959 clock cycles to write all the outputs.
- You can apply new inputs after 3960 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: `type` is set to `none` and not pipelined.
- The top level has a single instance, which has a latency and initiation interval of 3668.
    - This block also has no pipelining and accounts for most of the clock cycles.
- Notice that the functions `read_data` and `write_data` are not noted here as instances of the top level.
    - Figure 6-6 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 to any function being automatically inlined.

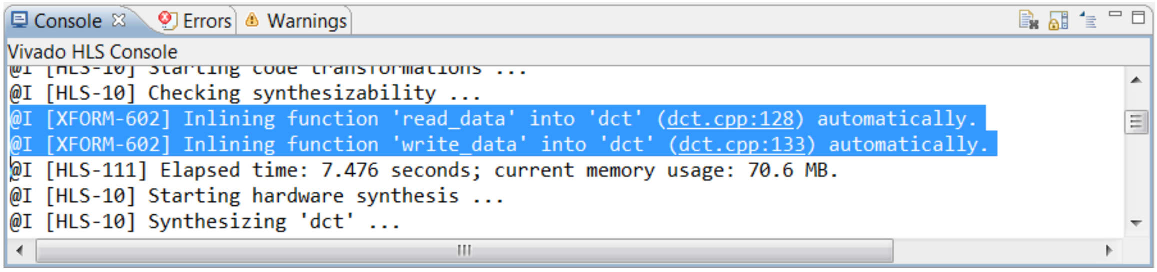


Figure 6-6: **Directives Editor Specifying Block RAM Interface**

- 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 6-7).
- 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:
  - 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/report folder under solution1 in the Explorer pane.

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

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 6-7) to begin interactive design analysis.

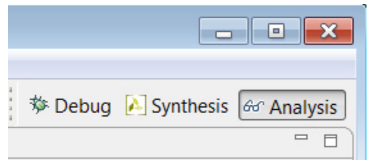


Figure 6-7: **Opening the Analysis Perspective**

The Analysis perspective consists of five panes, each of which is highlighted in Figure 6-8. You use all of these in the tutorial. The module and loops hierarchies are shown expanded (by default, they are shown collapsed).

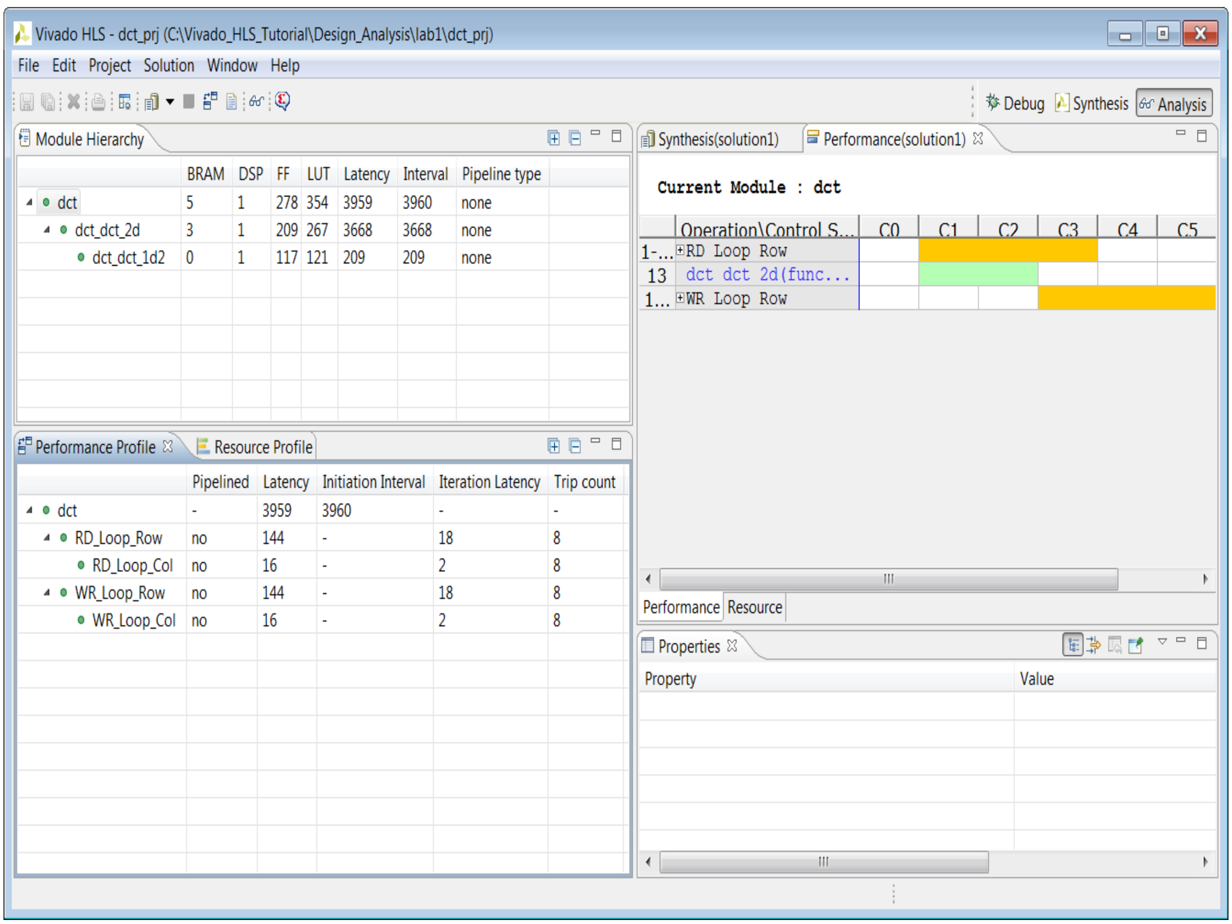


Figure 6-8: **Overview of the Analysis Perspective**

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 6-9). 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.

2. Click loop **RD_Loop_Row** and sub-loop **RD_Loop_Col** to fully expand the loop hierarchy (Figure 6-9).

Figure 6-9: **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 **Go to Source** (Figure 6-10).

a. When the dialog box opens, press OK to select item 0.

This opens the C source code to highlight the operation in the C source that created this adder. From the details on screen (also shown in Figure 6-10), you can determine it is indeed the loop counter. It is the only addition on this line, and the variable is named “r”.

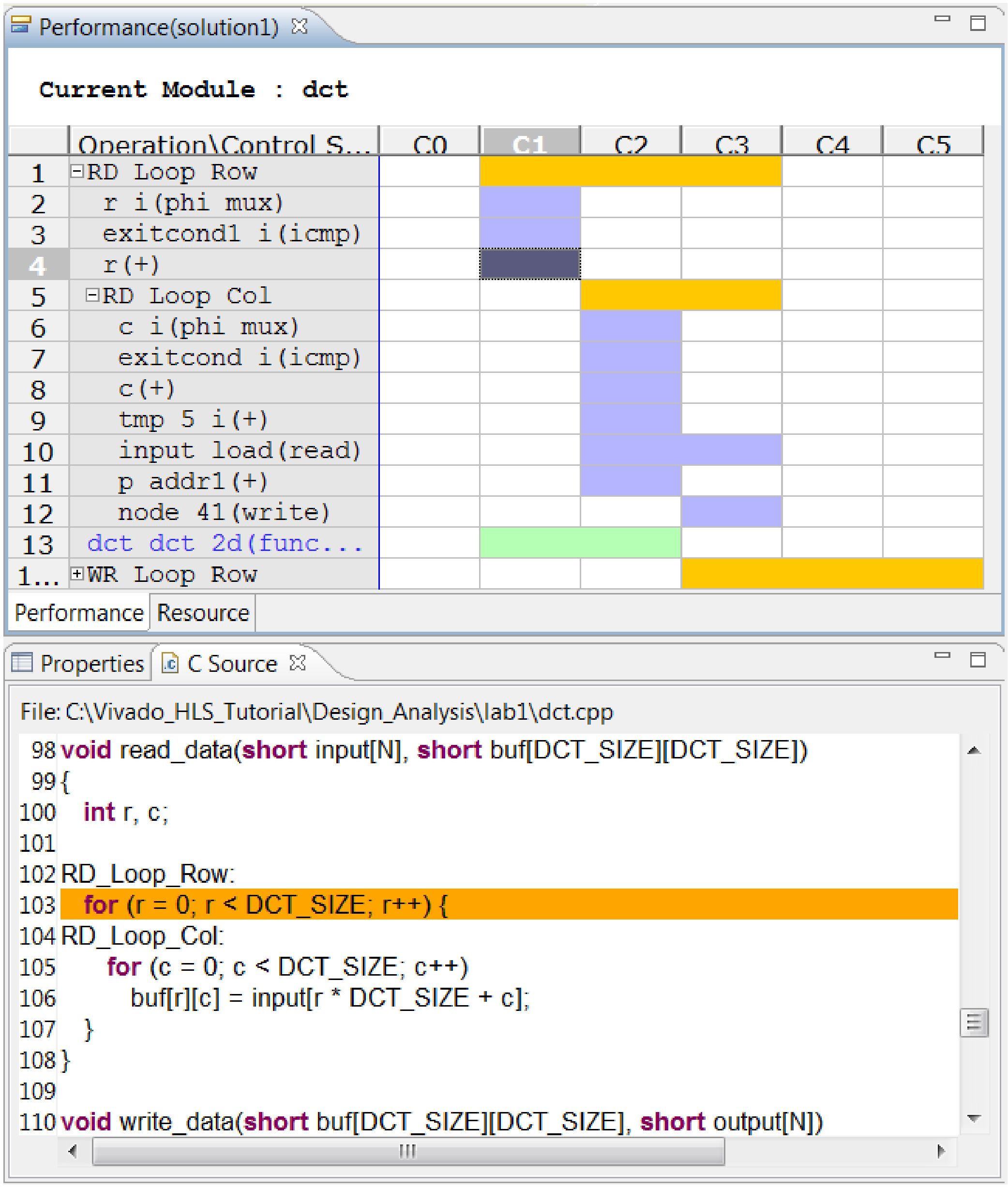


Figure 6-10: **C Source Code View**

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

4. Click any of the 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).
- 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 **X** in the C Source pane tab to close this window.
6. In the Module Hierarchy pane, click the function dct_2d to navigate into the view for this function (Figure 6-11).

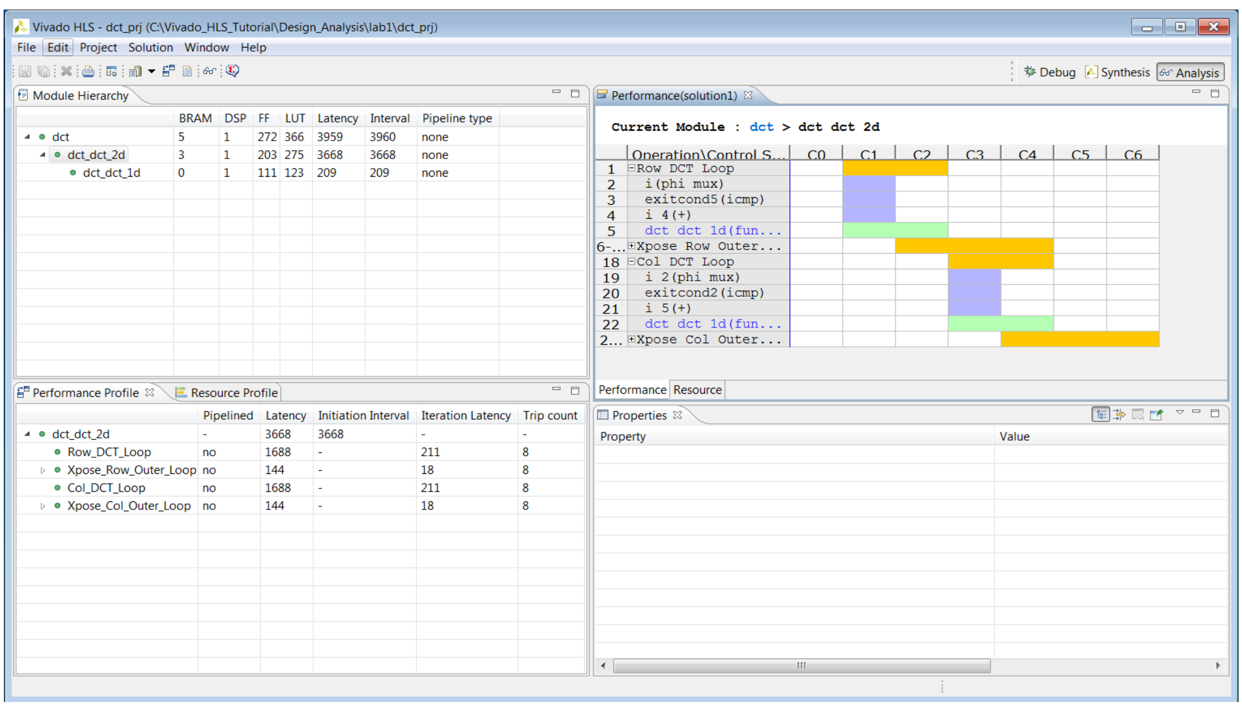


Figure 6-11: **DCT_2D Performance View**

Again, you can see a number of loops (shown in yellow in Figure 6-12). 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 6-12.

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 6-12.

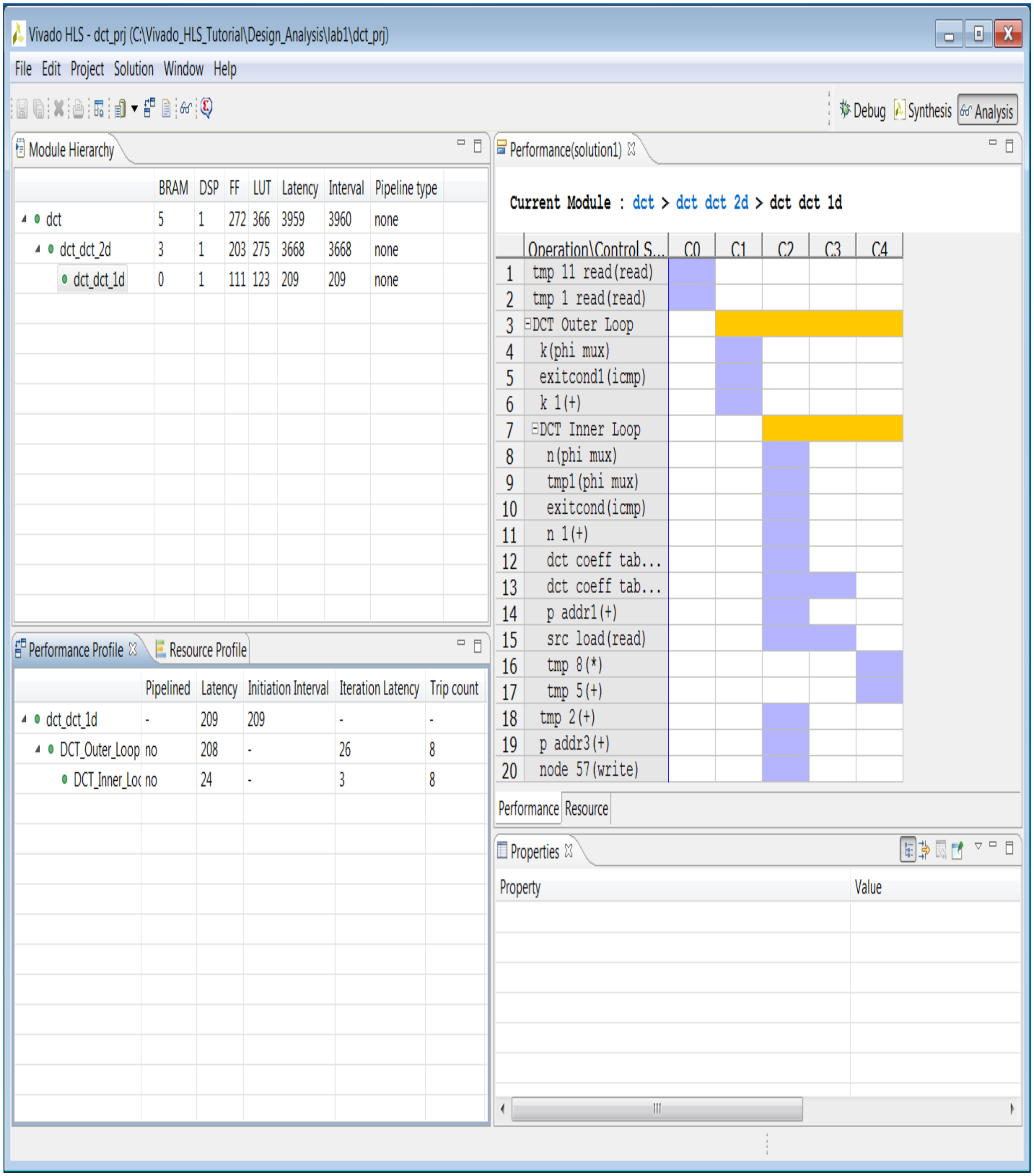


Figure 6-12: **DCT_1D Performance View**

In Figure 6-13 you can see a series of nested loops that 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 Chapter 7, Design Optimization tutorial. 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.

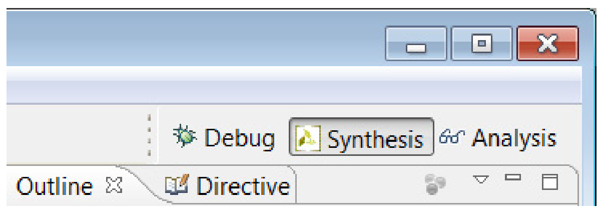


Figure 6-13: **Re-Opening the Synthesis Perspective**

Step 5: Apply Loop Pipelining and 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, see the Chapter 7, Design Optimization tutorial.

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 defaults.
3. Ensure that you can see the C source code in the Information pane.
4. In the Directive tab, add a pipeline directive to loop DCT_Inner_Loop in function dct_1d.
   a. Right-click DCT_Inner_Loop in the Directive pane and select **Insert Directive**.
   b. In the Directives Editor dialog box activate the Directive drop-down menu at the top and select **PIPELINE**.
   c. Click **OK** to 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.

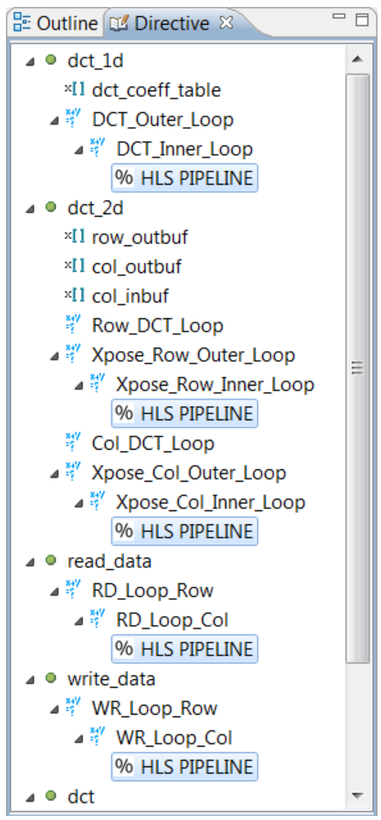
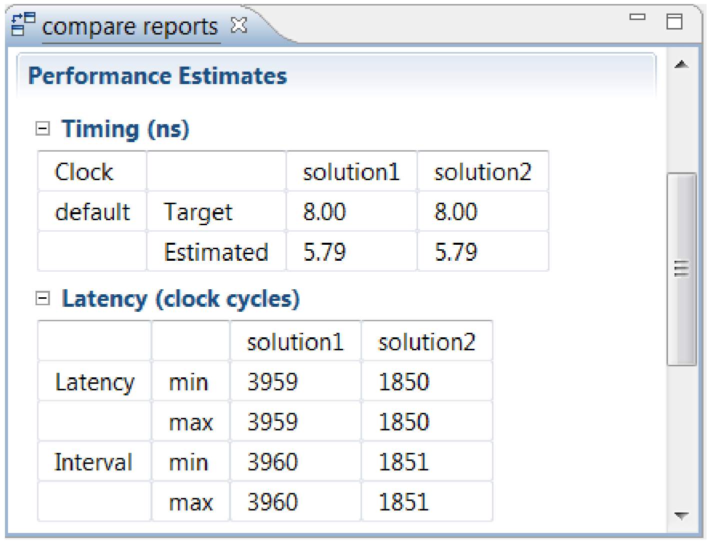


Figure 6-14: **Optimization Directive for DCT Loop Pipelines**

6. Click the **Run C Synthesis** 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 1 and 2.

Figure 6-15 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.



Clock		solution1	solution2
default	Target	8.00	8.00
	Estimated	5.79	5.79

		solution1	solution2
Latency	min	3959	1850
	max	3959	1850
Interval	min	3960	1851
	max	3960	1851

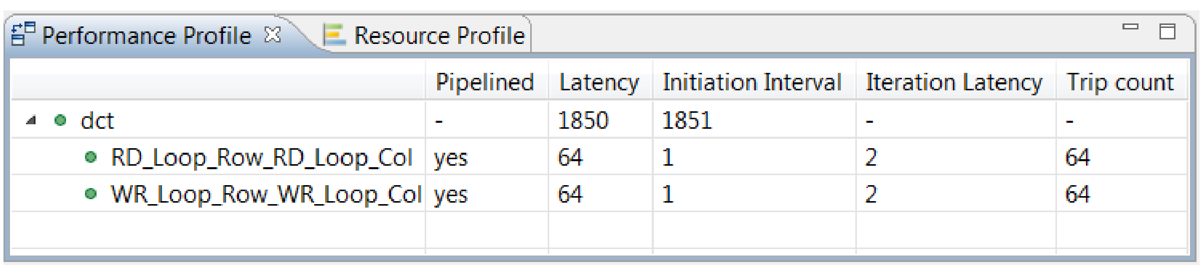
Figure 6-15: **DCT Solution1 and Solution2 Comparison**

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 6-16) shows that the latency of both loops has been reduced from 144 clock cycles in solution1 to only 64 clock cycles.



	Pipelined	Latency	Initiation Interval	Iteration Latency	Trip count
dct	-	1850	1851	-	-
RD_Loop_Row_RD_Loop_Col	yes	64	1	2	64
WR_Loop_Row_WR_Loop_Col	yes	64	1	2	64

Figure 6-16: **DCT Solution2 Performance of Top-Level Loops**

Pipelining loops transforms the latency from

```
Latency = iteration latency * (tripcount * interval)
```

to

```
Latency = iteration latency + (tripcount * interval)
```

Vivado 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 6-17 shows the loops that have been automatically optimized.

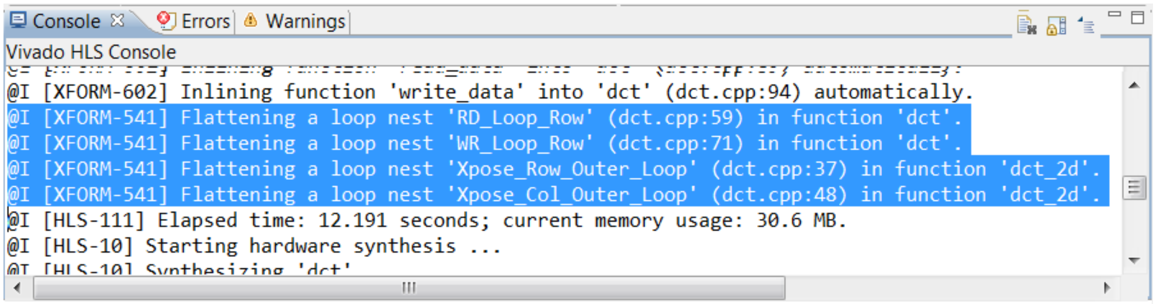


Figure 6-17: **DCT Solution2 Loop Flattening**

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 6-11). 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 6-18) 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 6-18, so no loop flattening occurred).

Performance Profile | Resource Profile

	Pipelined	Latency	Initiation Interval	Iteration Latency	Trip count
dct_dct_1d	-	97	97	-	-
DCT_Outer_Loop	no	96	-	12	8
DCT_Inner_Loop	yes	9	1	3	8

Figure 6-18: **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 6-19).
12. Select the **write operation in state C3**.
13. Right-click and select **Go to Source**.

Figure 6-19 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 6-19, below.

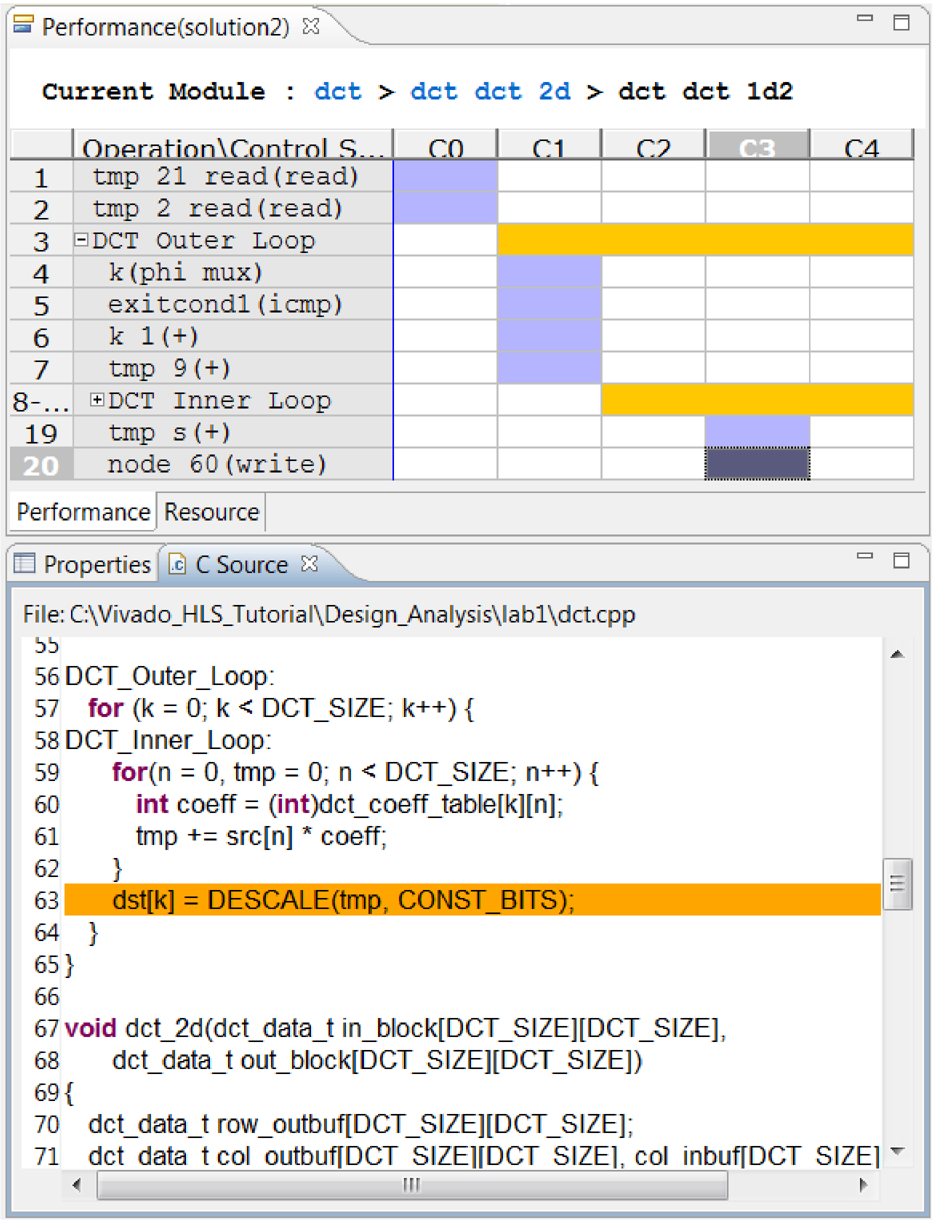


Figure 6-19: **DCT Solution2 dct_1d Performance View**

The write to the array cannot be flattened into the inner loop. To achieve an interval of 1 on `DCT Outer Loop` you will need to pipeline the output loop - there is no benefit in simply pipelining the inner loop itself.

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 **Finish** and accept the defaults to create solution3.
3. Ensure the C source code is visible in the Information pane.
4. In the **Directive** 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 **Directive** drop-down menu at the top and select **PIPELINE**.
    f. Click **OK** to select the default maximum pipeline rate (II=1).

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

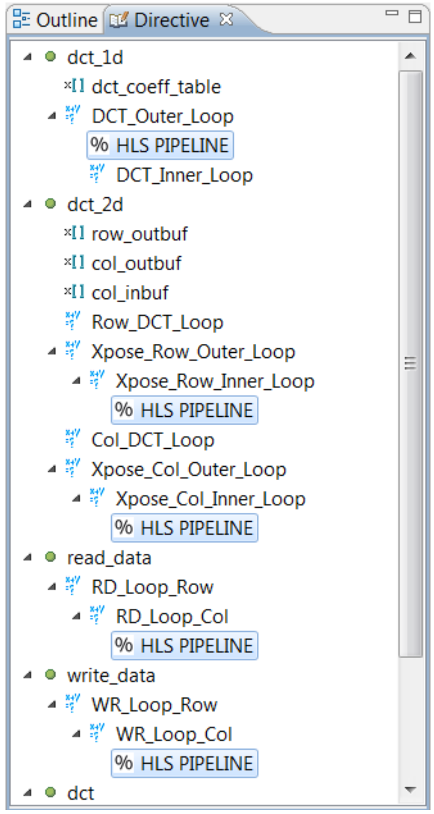
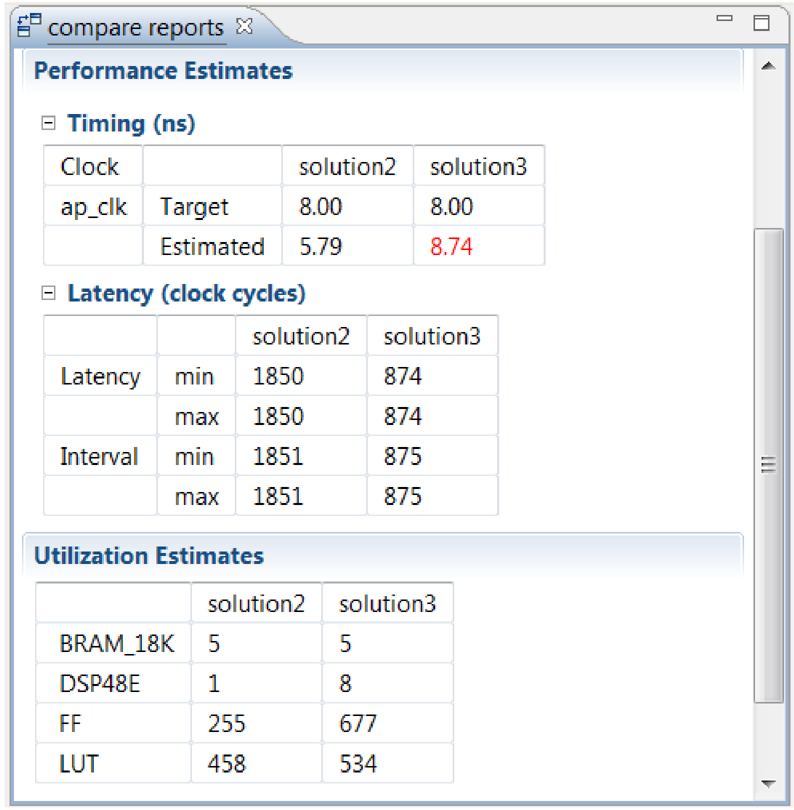


Figure 6-20: **Updated Optimization Directives for DCT Loop Pipelines**

5. Click the **Run C Synthesis** 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 6-21 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 reports

Performance Estimates

Timing (ns)

Clock		solution2	solution3
ap_clk	Target	8.00	8.00
	Estimated	5.79	8.74

Latency (clock cycles)

		solution2	solution3
Latency	min	1850	874
	max	1850	874
Interval	min	1851	875
	max	1851	875

Utilization Estimates

	solution2	solution3
BRAM_18K	5	5
DSP48E	1	8
FF	255	677
LUT	458	534

Figure 6-21: **DCT Solution2 and Solution3 Comparison**

In this case, the report indicates the clock period for solution3 cannot be achieved. Vivado HLS will sometimes create a design in which the estimated clock period fails to meet the required clock period. Typically, the design will meet timing after RTL synthesis - in this case, you can confirm this by using the Export RTL feature and selecting evaluate. In the event you encounter a case where the design fails to meet timing after RTL synthesis, use LATENCY directive in conjunction with regions in the C code to force Vivado HLS to register intermediate points on the failing RTL path.

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.
- Use 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 6-22, `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 6-22 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.

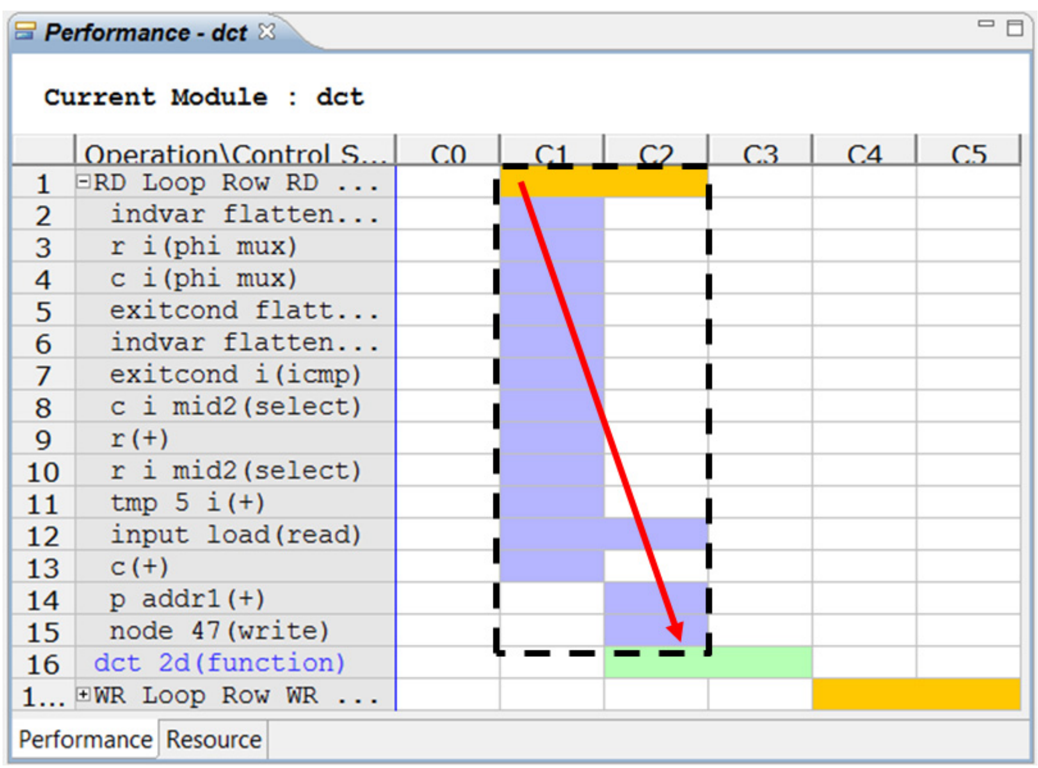


Figure 6-22: **Analysis of DCT RD_Loop_Row**

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 6-23.

Figure 6-23 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).

Figure 6-23: Analysis of `dct_1d RD_Loop_Row`.

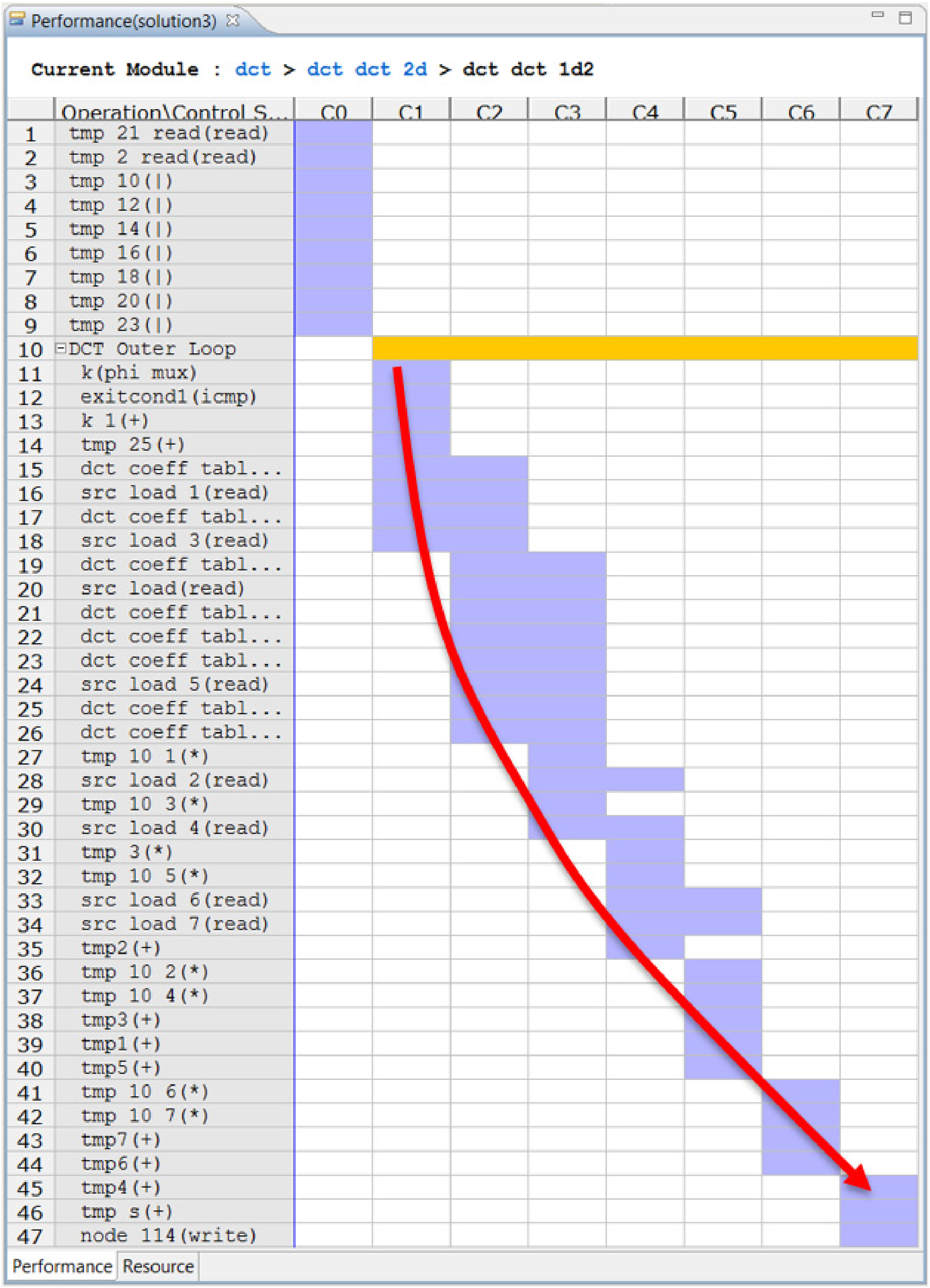
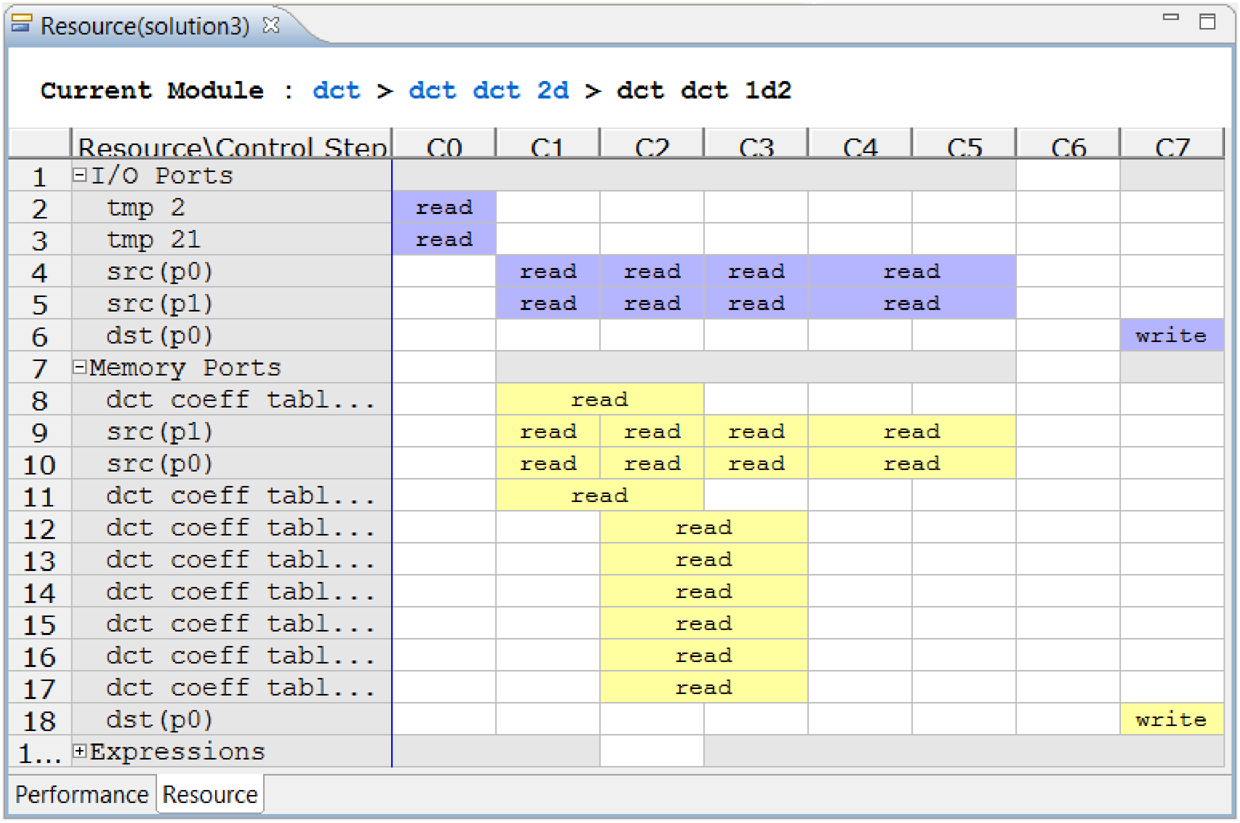


Figure 6-23: **Analysis of dct_1d RD_Loop_Row**

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** tab at the bottom of the window.

15. Expand the Memory Ports, as shown in Figure 6-24.


Resource(solution3)

Current Module : dct > dct dct 2d > dct dct 1d2

	Resource\Control Step	C0	C1	C2	C3	C4	C5	C6	C7
1	⊟I/O Ports								
2	tmp 2	read							
3	tmp 21	read							
4	src(p0)		read	read	read	read			
5	src(p1)		read	read	read	read			
6	dst(p0)								write
7	⊟Memory Ports								
8	dct coeff tabl...		read						
9	src(p1)		read	read	read	read			
10	src(p0)		read	read	read	read			
11	dct coeff tabl...		read						
12	dct coeff tabl...			read					
13	dct coeff tabl...			read					
14	dct coeff tabl...			read					
15	dct coeff tabl...			read					
16	dct coeff tabl...			read					
17	dct coeff tabl...			read					
18	dst(p0)								write
1...	⊞Expressions								

Performance Resource

Figure 6-24: **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 6-25, 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 6-25 shows the memory accesses on block RAM `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 6-25.

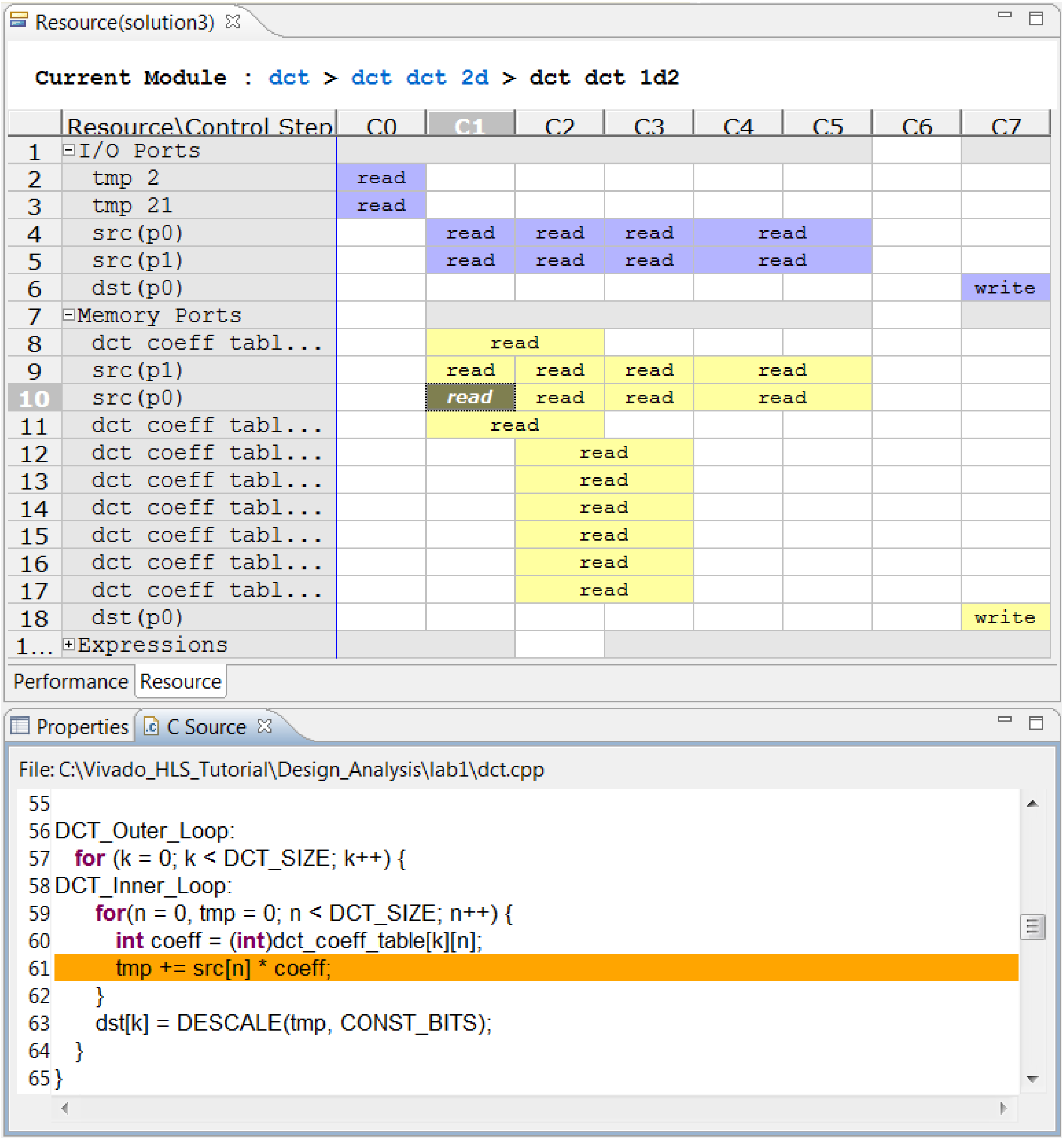


Figure 6-25: **Memory Resource SRC and Source Code**

Figure 6-26 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 6-26, 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 array 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 second 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, solution4.
2. Click **Finish** and accept the defaults to create solution4.
3. Ensure the C source code is visible in the Information pane.
4. In the Directive 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 **Directive** 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 second 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 6-26 (the two new directives are highlighted).

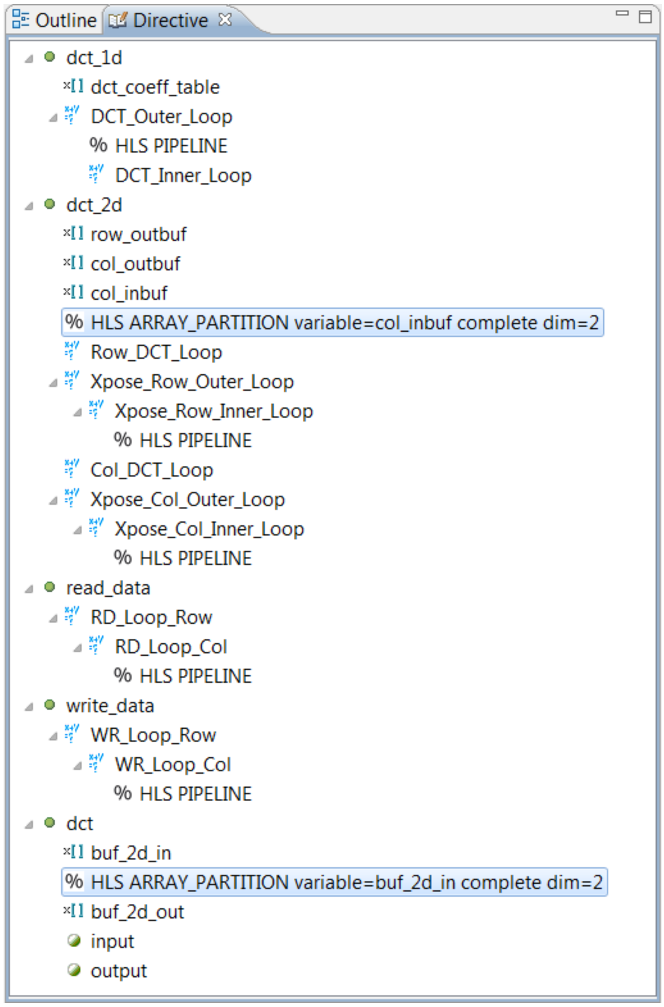
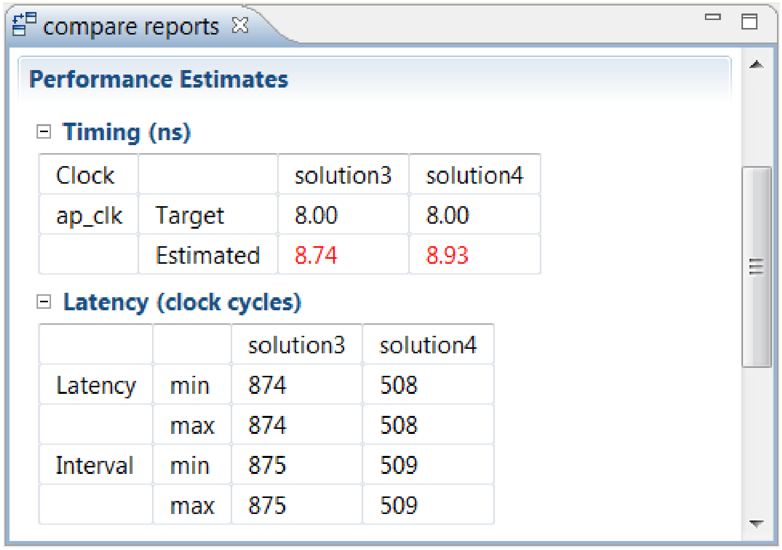


Figure 6-26: **Optimization Directives for Array Partitioning**

6. Click the Click the **Run C Synthesis** 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 6-27 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.



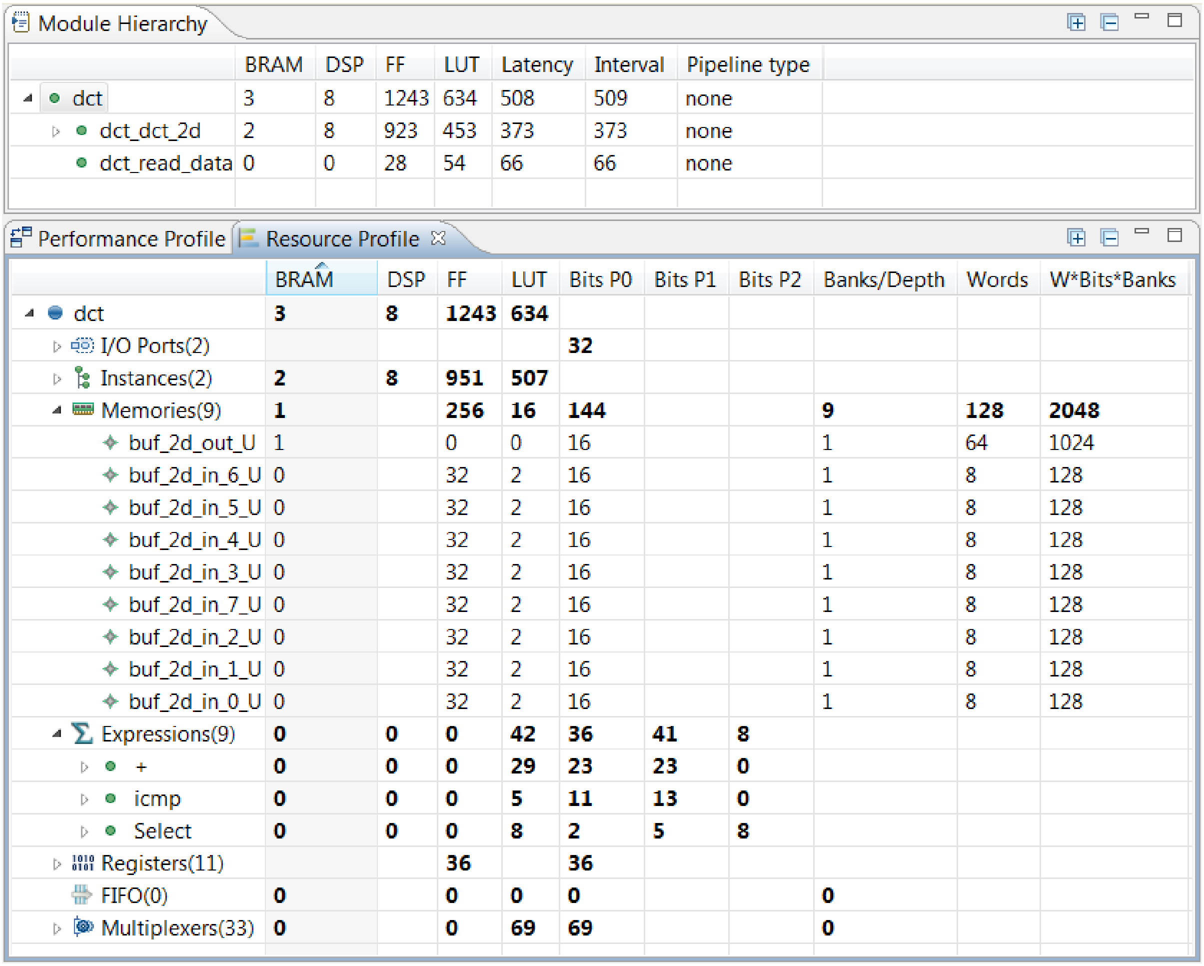
Clock		solution3	solution4
ap_clk	Target	8.00	8.00
	Estimated	8.74	8.93

		solution3	solution4
Latency	min	874	508
	max	874	508
Interval	min	875	509
	max	875	509

Figure 6-27: **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 6-28.



Module Hierarchy

	BRAM	DSP	FF	LUT	Latency	Interval	Pipeline type
dct	3	8	1243	634	508	509	none
dct_dct_2d	2	8	923	453	373	373	none
dct_read_data	0	0	28	54	66	66	none

Performance Profile / Resource Profile

	BRAM	DSP	FF	LUT	Bits P0	Bits P1	Bits P2	Banks/Depth	Words	W*Bits*Banks
dct	**3**	**8**	**1243**	**634**						
I/O Ports(2)					**32**					
Instances(2)	**2**	**8**	**951**	**507**						
Memories(9)	**1**		**256**	**16**	**144**			**9**	**128**	**2048**
buf_2d_out_U	1		0	0	16			1	64	1024
buf_2d_in_6_U	0		32	2	16			1	8	128
buf_2d_in_5_U	0		32	2	16			1	8	128
buf_2d_in_4_U	0		32	2	16			1	8	128
buf_2d_in_3_U	0		32	2	16			1	8	128
buf_2d_in_7_U	0		32	2	16			1	8	128
buf_2d_in_2_U	0		32	2	16			1	8	128
buf_2d_in_1_U	0		32	2	16			1	8	128
buf_2d_in_0_U	0		32	2	16			1	8	128
Expressions(9)	**0**	**0**	**0**	**42**	**36**	**41**	**8**			
+	**0**	**0**	**0**	**29**	**23**	**23**	**0**			
icmp	**0**	**0**	**0**	**5**	**11**	**13**	**0**			
Select	**0**	**0**	**0**	**8**	**2**	**5**	**8**			
Registers(11)			**36**		**36**					
FIFO(0)	**0**		**0**	**0**	**0**			**0**		
Multiplexers(33)	**0**		**0**	**69**	**69**			**0**		

Figure 6-28: **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 6-29 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. Since they are less than 1024 bits they are automatically implemented as LUTRAM.
- 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 6-29, however, shows that you can only accept new data every 525 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, solution5.
2. Click **Finish** and accept the defaults to create solution5.
3. Ensure the C source code is visible in the Information pane.
4. In the **Directive** tab
   a. Select the top-level function `dct`.
   b. Right-click and select **Insert Directive**.
   c. In the **Directives Editor** dialog box activate the **Directive** drop-down menu and select **DATAFLOW**.
   d. Click **OK**.

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

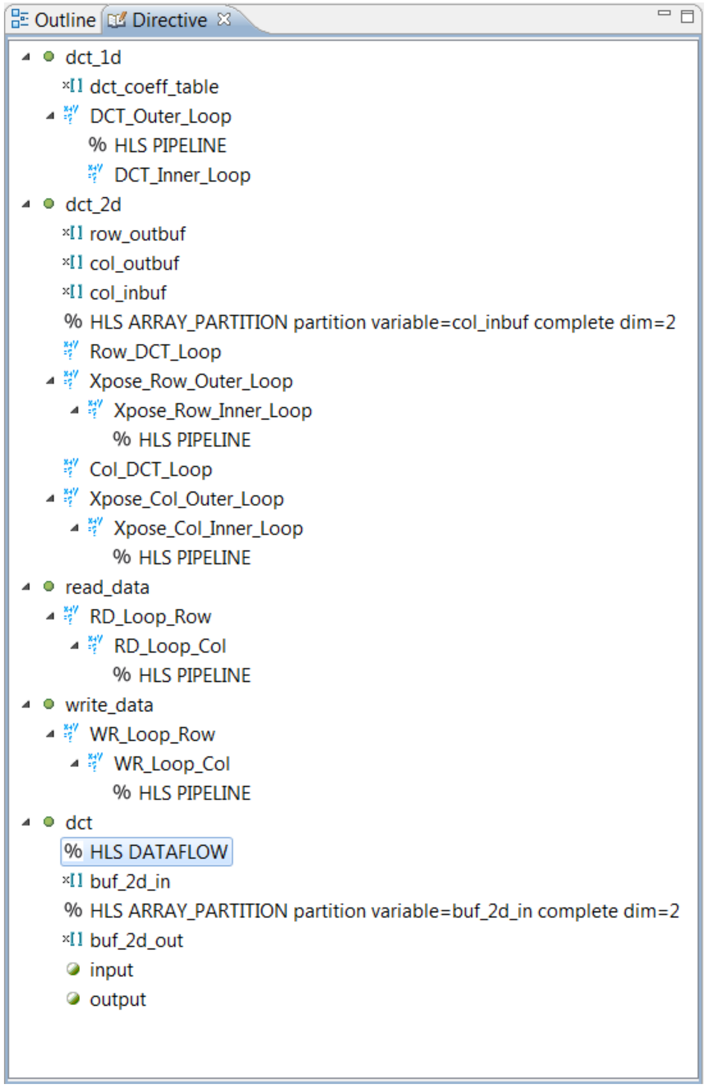
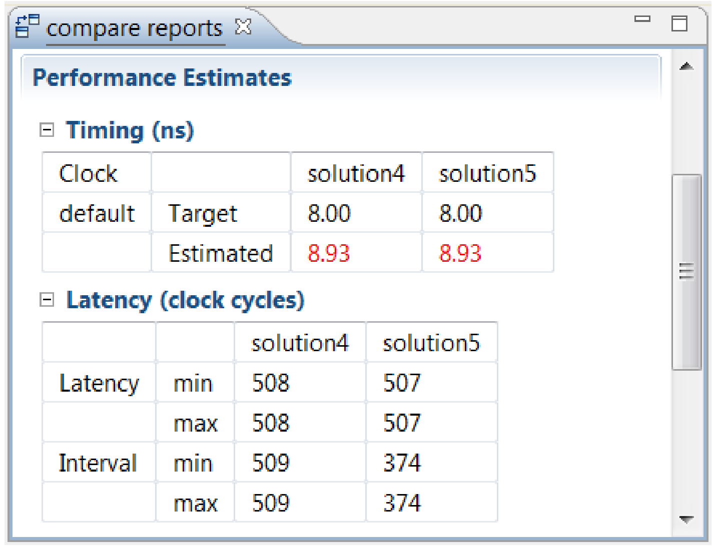


Figure 6-29: **Dataflow Optimization for the DCT Design**

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

Figure 6-30 shows the results of comparing solution4 and solution5, and you can see the interval has improved. The design takes 525 clocks cycles to produce the outputs but can now accept new inputs every 390 clocks.



compare reports

Performance Estimates

Timing (ns)

Clock		solution4	solution5
default	Target	8.00	8.00
	Estimated	8.93	8.93

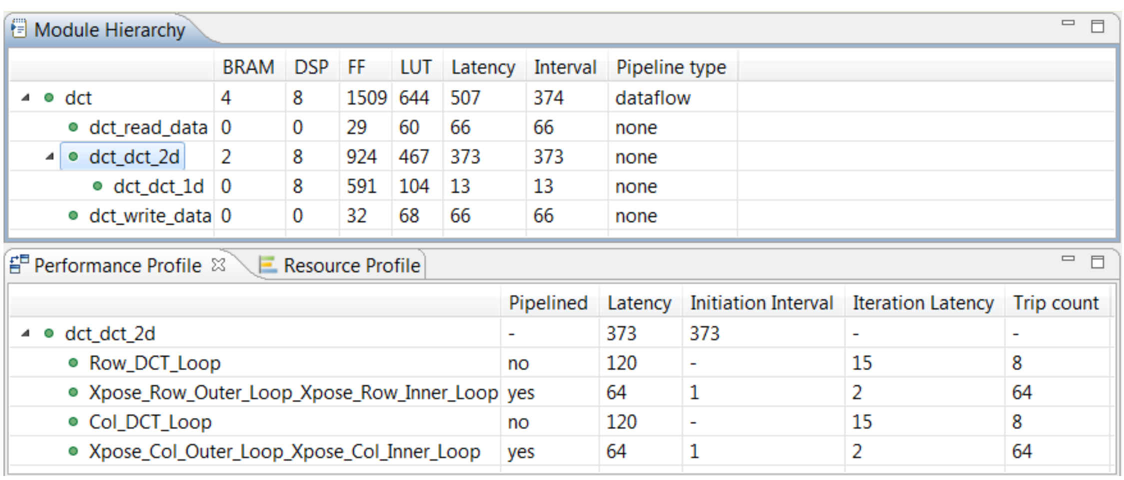
Latency (clock cycles)

		solution4	solution5
Latency	min	508	507
	max	508	507
Interval	min	509	374
	max	509	374

Figure 6-30: **DCT Solution4 and Solution5 Comparison**

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

7. Click the **Analysis** perspective button to begin interactive design analysis.
8. 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 6-31.



Module Hierarchy

	BRAM	DSP	FF	LUT	Latency	Interval	Pipeline type
dct	4	8	1509	644	507	374	dataflow
dct_read_data	0	0	29	60	66	66	none
dct_dct_2d	2	8	924	467	373	373	none
dct_dct_1d	0	8	591	104	13	13	none
dct_write_data	0	0	32	68	66	66	none

Performance Profile | Resource Profile

	Pipelined	Latency	Initiation Interval	Iteration Latency	Trip count
dct_dct_2d	-	373	373	-	-
Row_DCT_Loop	no	120	-	15	8
Xpose_Row_Outer_Loop_Xpose_Row_Inner_Loop	yes	64	1	2	64
Col_DCT_Loop	no	120	-	15	8
Xpose_Col_Outer_Loop_Xpose_Col_Inner_Loop	yes	64	1	2	64

Figure 6-31: **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.

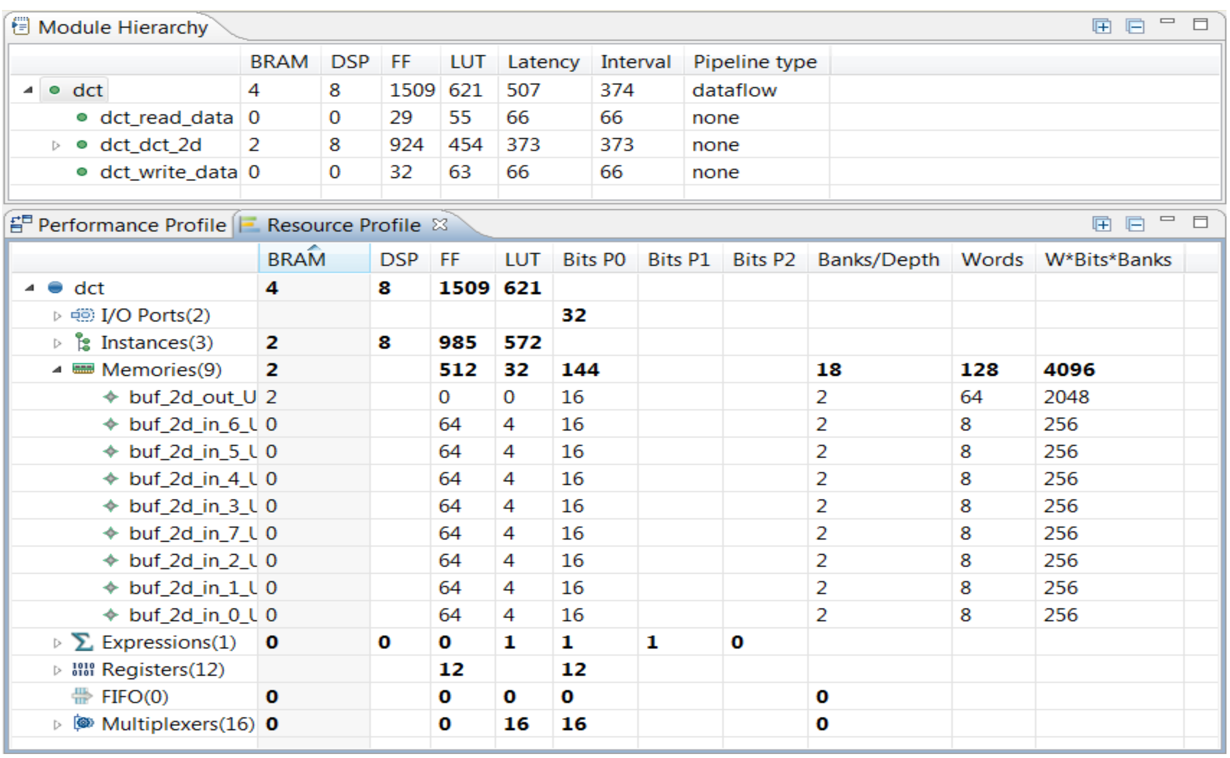
Because the `dct_2d` block is selected in the Module Hierarchy and the Performance Profile shows the details for this block. Figure 6-32 shows the interval is the same as the latency, so none of these blocks operate in parallel.

One way to have the blocks in `dct_2d` 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 `dct_2d`.

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.

9. In the **Module Hierarchy**, ensure module `dct` is selected.
10. Activate the **Resource Profile** view.
11. Expand the memories to see the view in Figure 6-32.



Module Hierarchy

	BRAM	DSP	FF	LUT	Latency	Interval	Pipeline type
dct	4	8	1509	621	507	374	dataflow
dct_read_data	0	0	29	55	66	66	none
dct_dct_2d	2	8	924	454	373	373	none
dct_write_data	0	0	32	63	66	66	none

Performance Profile | Resource Profile

	BRAM	DSP	FF	LUT	Bits P0	Bits P1	Bits P2	Banks/Depth	Words	W*Bits*Banks
dct	4	8	1509	621						
I/O Ports(2)					32					
Instances(3)	2	8	985	572						
Memories(9)	2		512	32	144			18	128	4096
buf_2d_out_U	2		0	0	16			2	64	2048
buf_2d_in_6_U	0		64	4	16			2	8	256
buf_2d_in_5_U	0		64	4	16			2	8	256
buf_2d_in_4_U	0		64	4	16			2	8	256
buf_2d_in_3_U	0		64	4	16			2	8	256
buf_2d_in_7_U	0		64	4	16			2	8	256
buf_2d_in_2_U	0		64	4	16			2	8	256
buf_2d_in_1_U	0		64	4	16			2	8	256
buf_2d_in_0_U	0		64	4	16			2	8	256
Expressions(1)	0	0	0	1	1	1	0			
Registers(12)			12		12					
FIFO(0)	0		0	0	0			0		
Multiplexers(16)	0		0	16	16			0		

Figure 6-32: **DCT Resource Profile**

As compared with Figure 6-28, the amount of logic to implement the memories at this level of hierarchy has doubled (the number flip-flops and LUTs has doubled). 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.

12. 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, solution6.
2. Click **Finish** and accept the defaults to create solution6.
3. Ensure the C source code is visible in the Information pane.
4. In the **Directive** tab:
   a. Select function `dct_2d`.
   b. Right-click and select **Insert Directive**.
   c. In the **Directives Editor** dialog box activate the **Directive** 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).

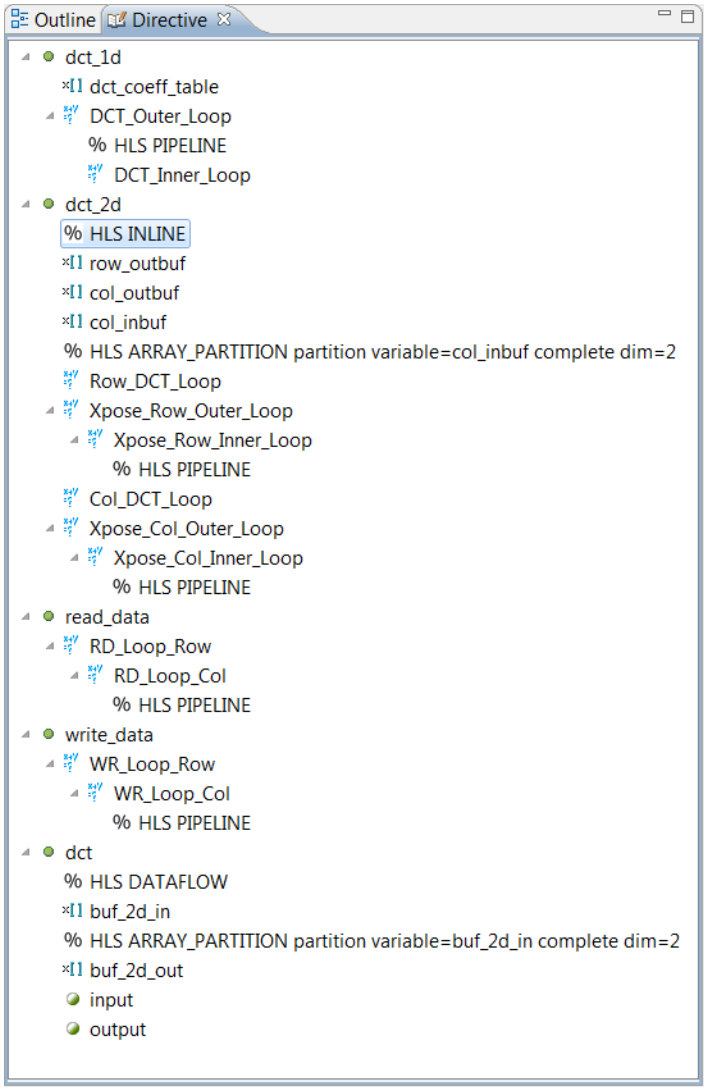
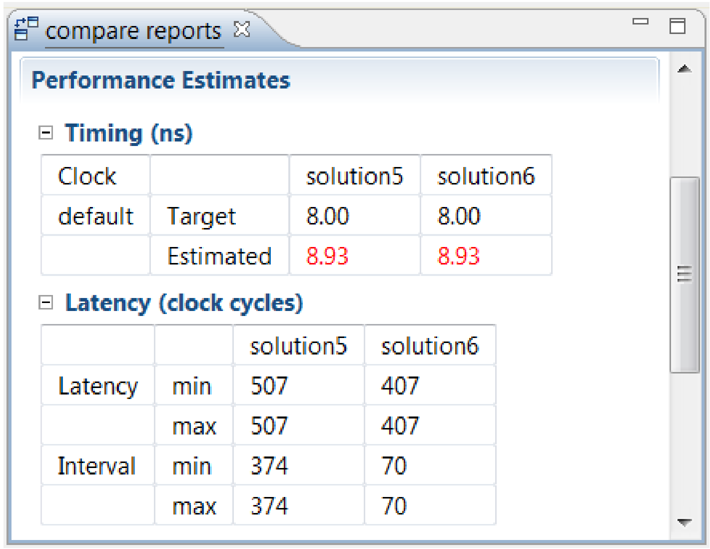


Figure 6-33: **Dataflow Optimization for the DCT Design**

5. Click the **Run C Synthesis** 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 6-34 shows the results of comparing solution5 and solution6. You can see the interval has improved substantially.



compare reports

Performance Estimates

Timing (ns)

Clock		solution5	solution6
default	Target	8.00	8.00
	Estimated	8.93	8.93

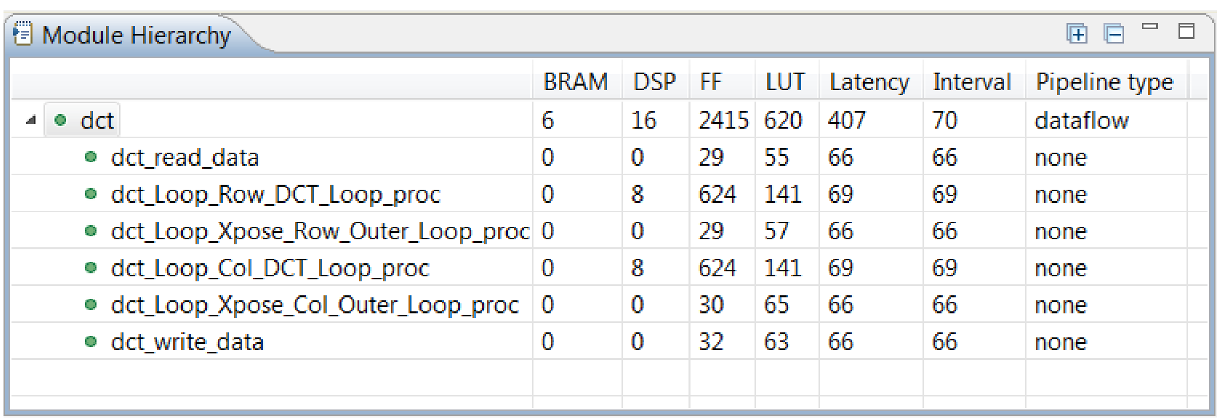
Latency (clock cycles)

		solution5	solution6
Latency	min	507	407
	max	507	407
Interval	min	374	70
	max	374	70

Figure 6-34: **DCT Solution5 and Solution6 Comparison**

The interval is now below the 100 clock target. This design can accept a new set of inputs 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 6-35.



Module Hierarchy

	BRAM	DSP	FF	LUT	Latency	Interval	Pipeline type
dct	6	16	2415	620	407	70	dataflow
dct_read_data	0	0	29	55	66	66	none
dct_Loop_Row_DCT_Loop_proc	0	8	624	141	69	69	none
dct_Loop_Xpose_Row_Outer_Loop_proc	0	0	29	57	66	66	none
dct_Loop_Col_DCT_Loop_proc	0	8	624	141	69	69	none
dct_Loop_Xpose_Col_Outer_Loop_proc	0	0	30	65	66	66	none
dct_write_data	0	0	32	63	66	66	none

Figure 6-35: **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.

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 may perform the analysis in these lab exercises using the Analysis perspective. A prerequisite for this tutorial is completion of the Chapter 6, Design Analysis tutorial.

Lab 1 Description

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.

Lab 2 Description

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. See the information in Locating the Tutorial Design Files.

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.
   - On Windows use **Start > All Programs > Xilinx Design Tools > Vivado 2015.3 > Vivado HLS > Vivado HLS 2015.3 Command Prompt** (Figure 7-1).
   - On Linux, open a new shell.

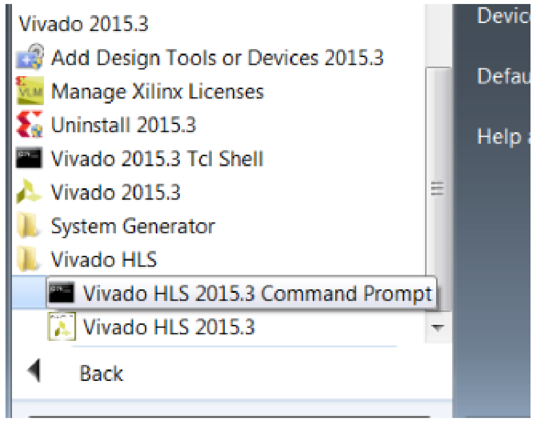
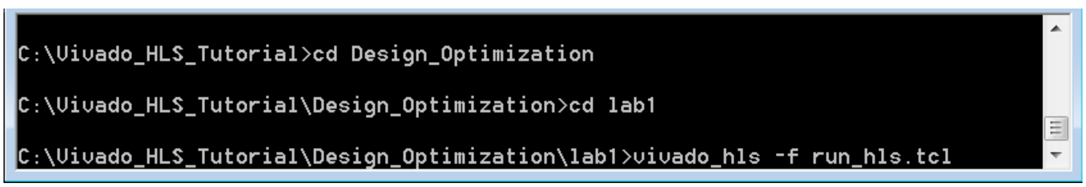


Figure 7-1: **Vivado HLS Command Prompt**

2. Using the command prompt window (Figure 7-2), change directory to the **Design Optimization** 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 7-2.



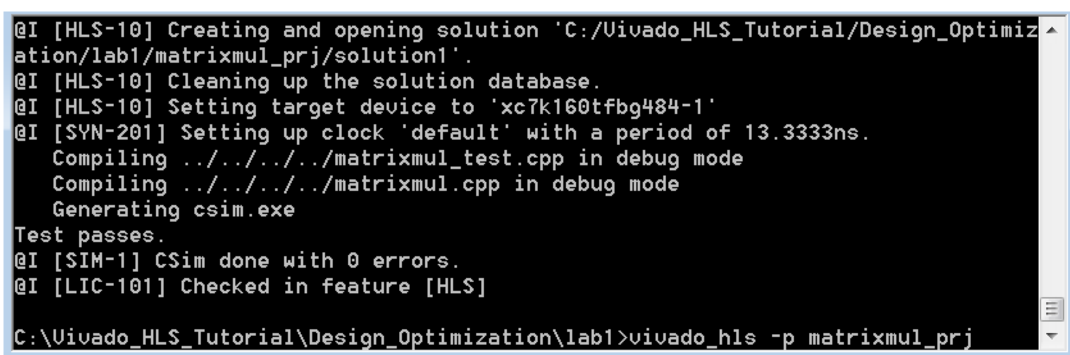
```
C:\Vivado_HLS_Tutorial>cd Design_Optimization

C:\Vivado_HLS_Tutorial\Design_Optimization>cd lab1

C:\Vivado_HLS_Tutorial\Design_Optimization\lab1>vivado_hls -f run_hls.tcl
```

Figure 7-2: **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 7-3.



```
@I [HLS-10] Creating and opening solution 'C:/Vivado_HLS_Tutorial/Design_Optimiz
ation/lab1/matrixmul_prj/solution1'.
@I [HLS-10] Cleaning up the solution database.
@I [HLS-10] Setting target device to 'xc7k160tfbg484-1'
@I [SYN-201] Setting up clock 'default' with a period of 13.3333ns.
   Compiling ../../../../matrixmul_test.cpp in debug mode
   Compiling ../../../../matrixmul.cpp in debug mode
   Generating csim.exe
Test passes.
@I [SIM-1] CSim done with 0 errors.
@I [LIC-101] Checked in feature [HLS]

C:\Vivado_HLS_Tutorial\Design_Optimization\lab1>vivado_hls -p matrixmul_prj
```

Figure 7-3: **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 7-4).

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 7-4) to see that each is three-by-three for a total of nine elements.

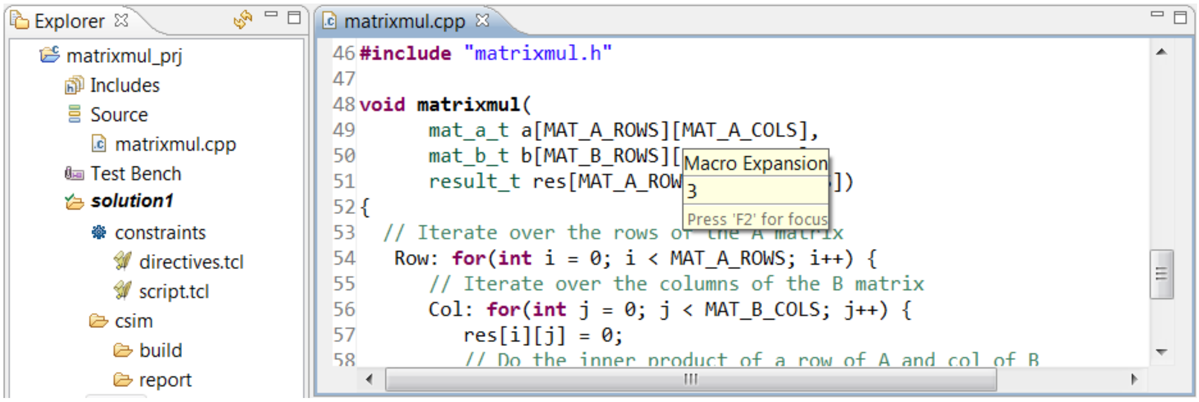


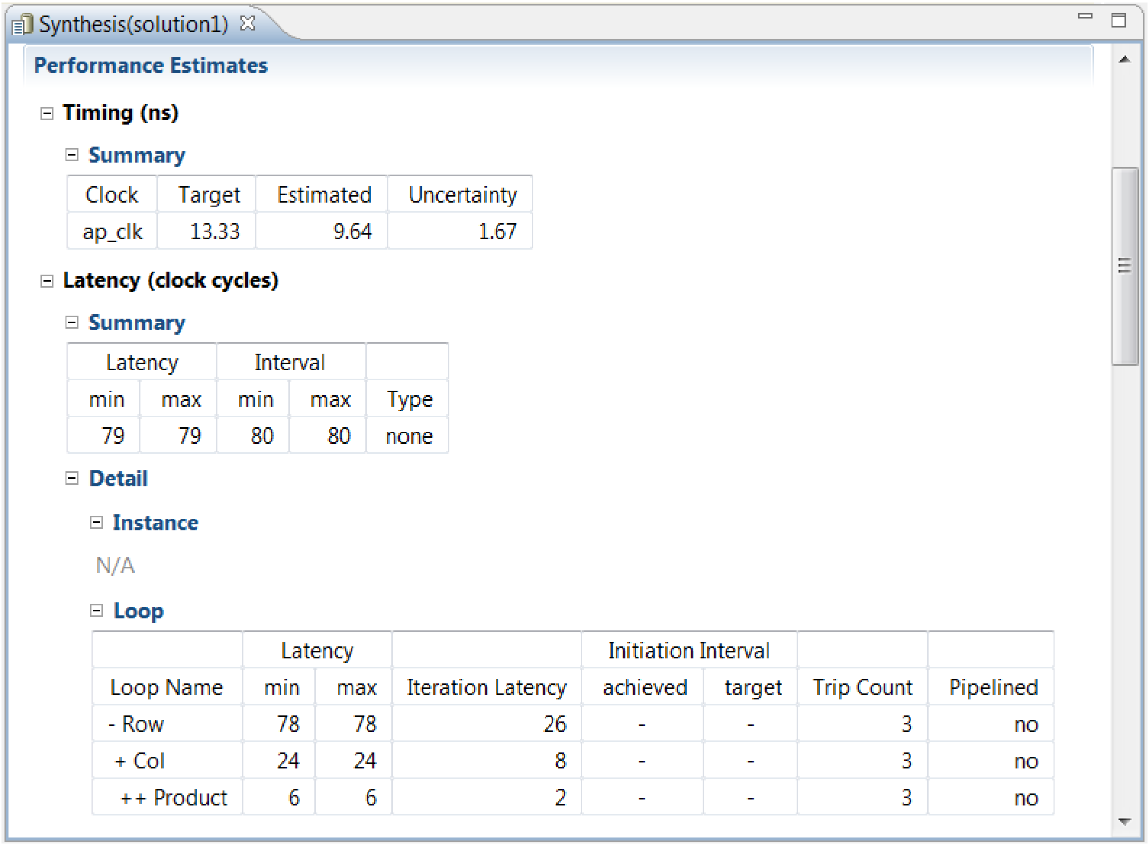
Figure 7-4: **Source Code for the Matrix Multiplier**

Step 2: Synthesize and Analyze the Design

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

When synthesis completes, the synthesis report opens (Figure 7-5), 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.
  - 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.



Synthesis(solution1)

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	13.33	9.64	1.67

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
79	79	80	80	none

Detail

Instance

N/A

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Row	78	78	26	-	-	3	no
+ Col	24	24	8	-	-	3	no
++ Product	6	6	2	-	-	3	no

Figure 7-5: **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, `solution2`.
2. Click **Finish** 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 (essentially creating a single loop with more iterations but overall fewer clock cycles).

4. In the **Directive** tab:
   a. Select loop **Product**.
   b. Right-click and select **Insert Directive**.
   c. In the **Directives Editor** dialog box, activate the **Directive** 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.)

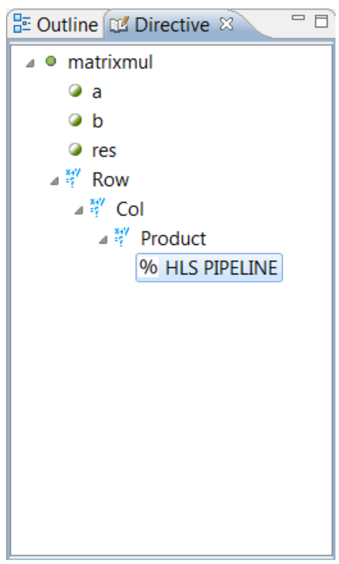


Figure 7-6: **Initial Pipeline Directive**

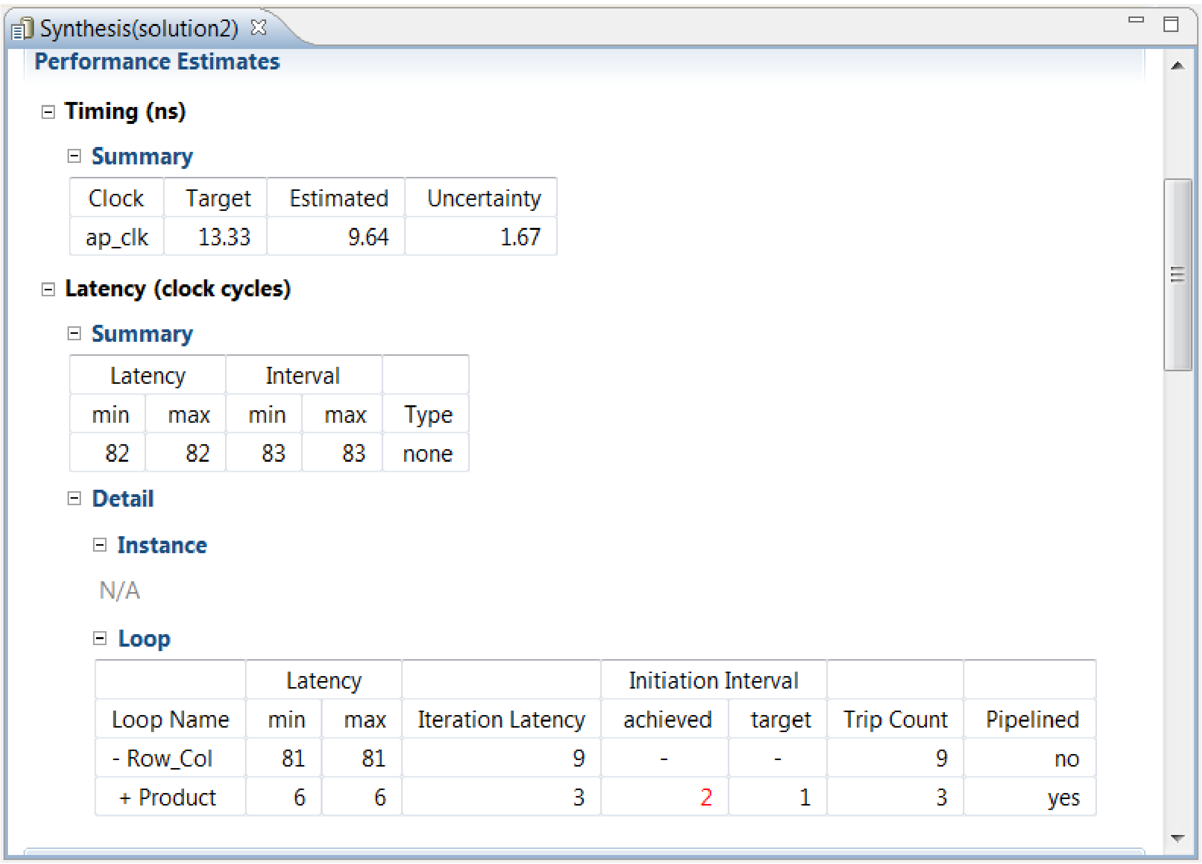
5. Click the **Run C Synthesis** toolbar button to synthesize the design to **RTL**.

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_8' 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: 2.
```

The synthesis report (Figure 7-7) shows that although the Product loop is pipelined with an interval of 2, the interval of top-level loop is not pipelined.


Synthesis(solution2)

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	13.33	9.64	1.67

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
82	82	83	83	none

Detail

Instance

N/A

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Row_Col	81	81	9	-	-	9	no
+ Product	6	6	3	2	1	3	yes

Figure 7-7: **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 7-8.

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.

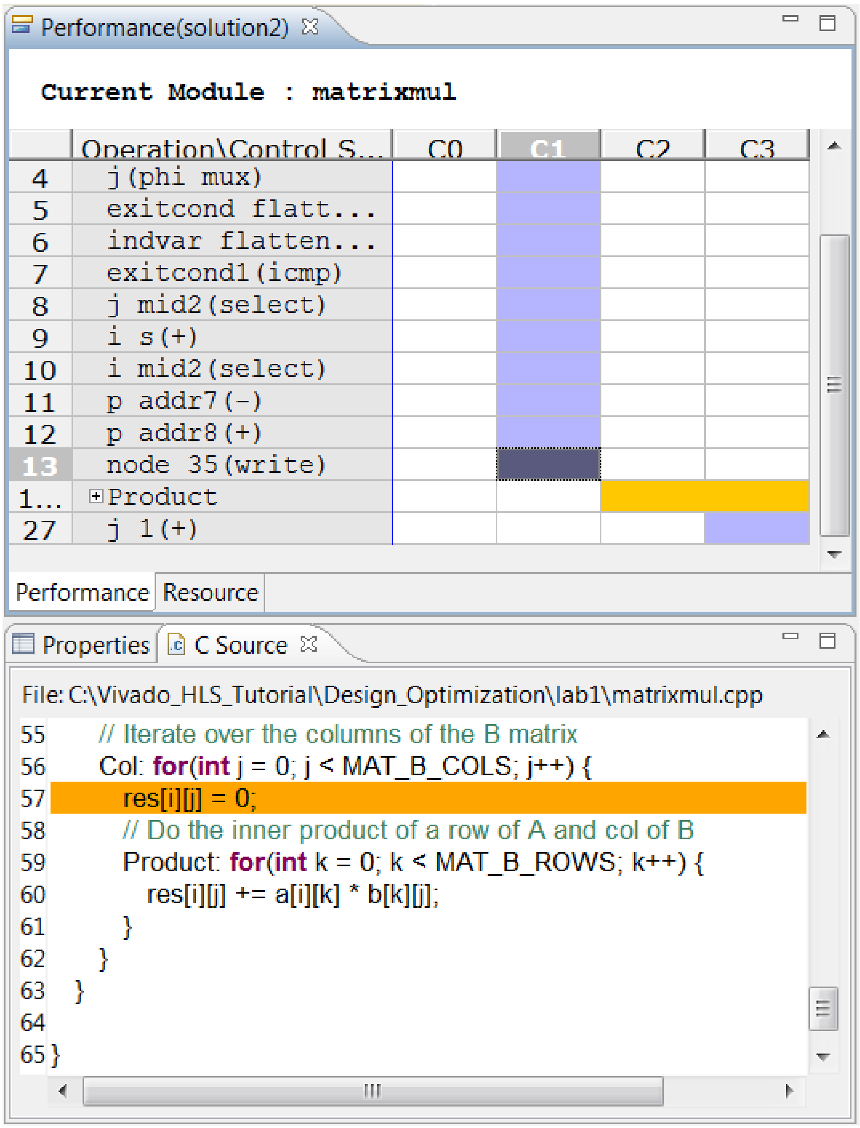


Figure 7-8: **Matrixmul Initial Performance View**

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_8' 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 7-9 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 7-9 shows a number of copies of the schedule for the Product loop to highlight how this issue can be understood. The operations on the res array, a two-cycle read and write, are highlighted.

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 highlighted cells 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.

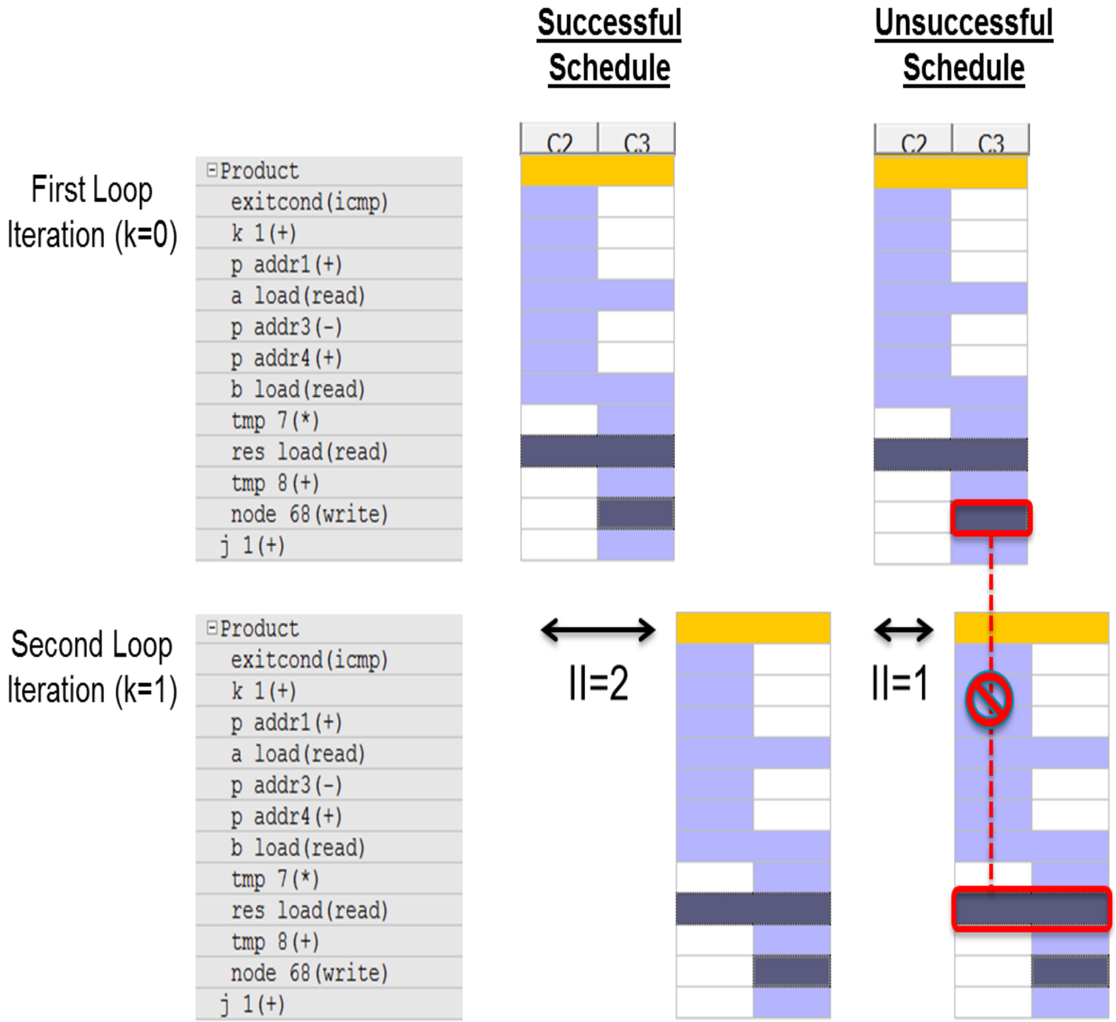


Figure 7-9: **Carried Dependency Analysis**

You cannot pipeline the Product loop with an initiation interval of 1. The next lab exercise shows how re-writing the code can remove this limitation (any technique that does not write back to the same array/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, **solution3**.
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 **Directive** tab:
    a. Select loop **Col**.
    b. Right-click and select **Insert Directive**.
    c. In the **Directives Editor** dialog box activate the **Directive** 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).

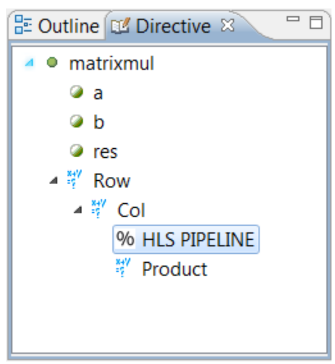


Figure 7-10: **Col Pipeline Directive**

6. Click the **Run C Synthesis** 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', 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 7-2. 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.

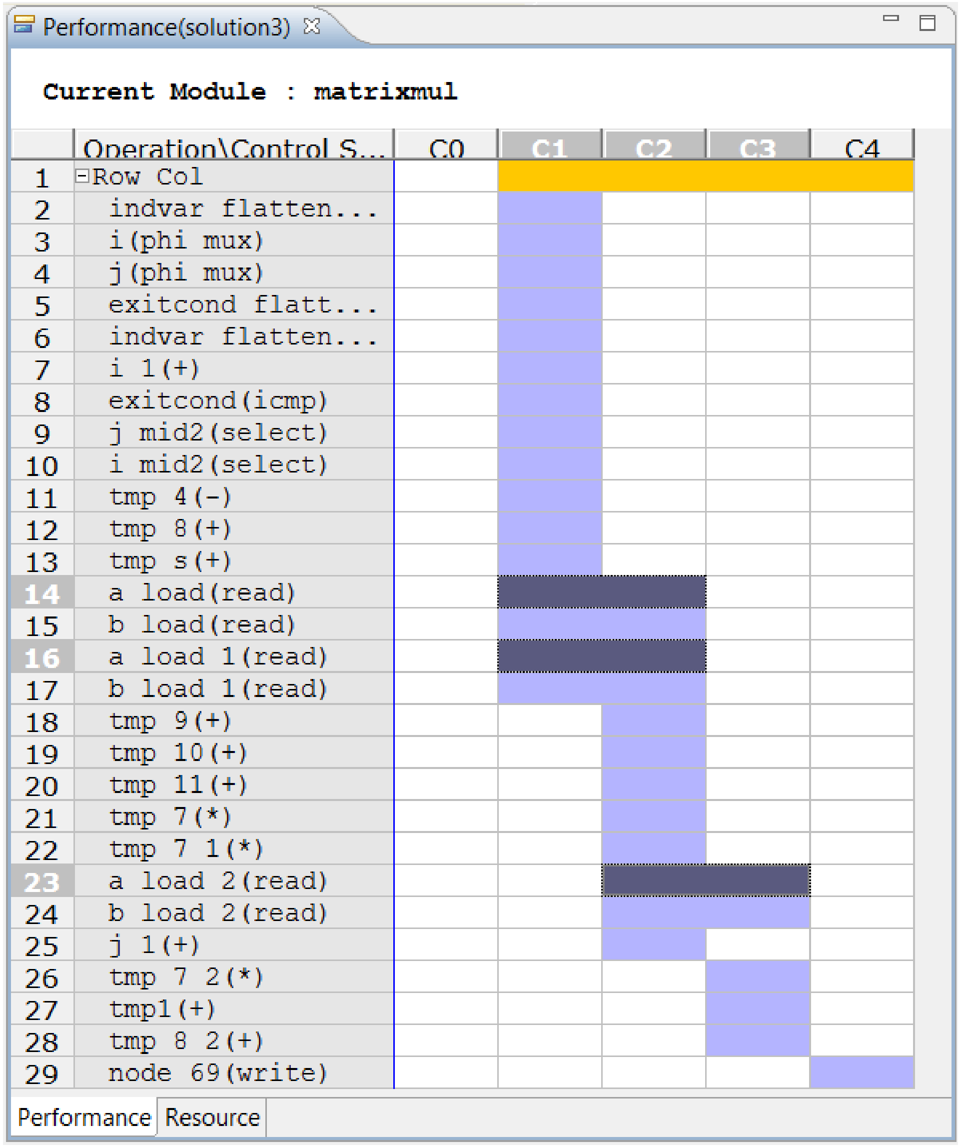


Figure 7-11: **Matrixmul Pipeline Col Performance View**

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

9. Click the **Resource** tab.
10. Expand the memories to see the view shown in Figure 7-12.

In Figure 7-12 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.

Resource(solution3)

Current Module : matrixmul

	Resource\Control Step	C0	C1	C2	C3	C4
1-6	⊞I/O Ports					
7	⊟Memory Ports					
8	b(p1)		read			
9	b(p0)		read	read		
10	a(p0)		read	read		
11	a(p1)		read			
12	res(p0)					write
1...	⊞Expressions					

Performance | Resource

Figure 7-12: **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, `solution4`.

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.

2. Open the C source code `matrixmul.cpp` to make it visible in the Information pane.
3. In the **Directive** tab
   a. Select **variable a**.
   b. Right-click and select **Insert Directive**.
   c. In the **Directives Editor** dialog box activate the **Directive** drop-down menu at the top and select **ARRAY_RESHAPE**.
   d. Set the dimension to **2**.
   e. Click **OK**.
4. Repeat this process for variable **b**, but set the **dimension** to **1**.

The Directive pane should show the following optimization directives.

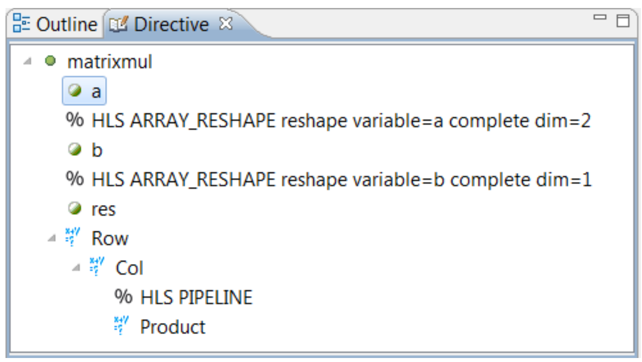
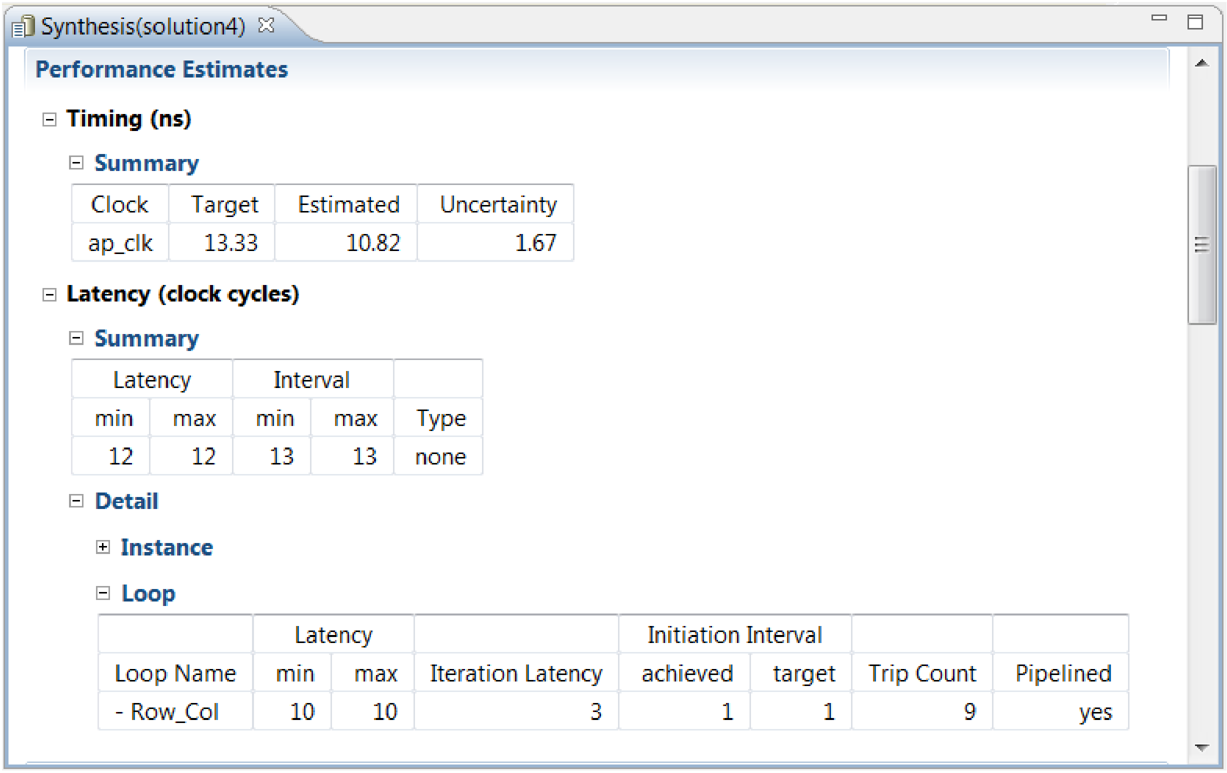


Figure 7-13: **Array Reshape Directive**

5. Click the **Run C Synthesis** 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 7-14).



Synthesis(solution4)

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	13.33	10.82	1.67

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
12	12	13	13	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Row_Col	10	10	3	1	1	9	yes

Figure 7-14: **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 **Directive** tab
    a. Select **variable a**.

   b. Right-click and select **Insert Directive**.

   c. In the **Directives Editor** dialog box activate the **Directive** 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 variables `b` and variable `res`.

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

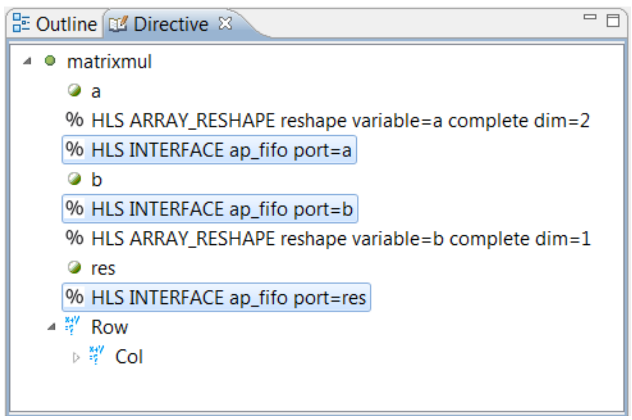


Figure 7-15: **Matrixmul FIFO Directives**

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

Figure 7-16 shows the **Console** display after synthesis runs.

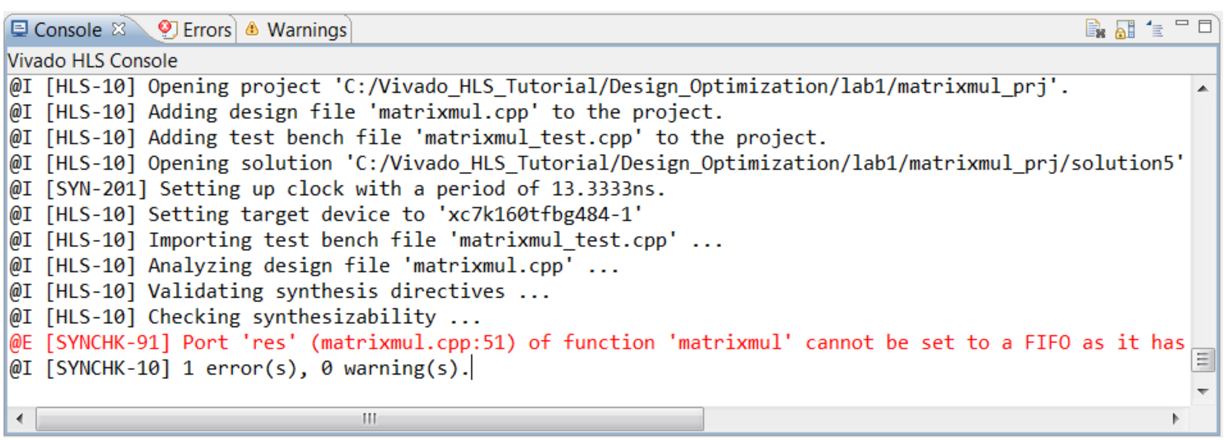


Figure 7-16: **FIFO Synthesis Warning**

From the code shown in Figure 7-17, 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.

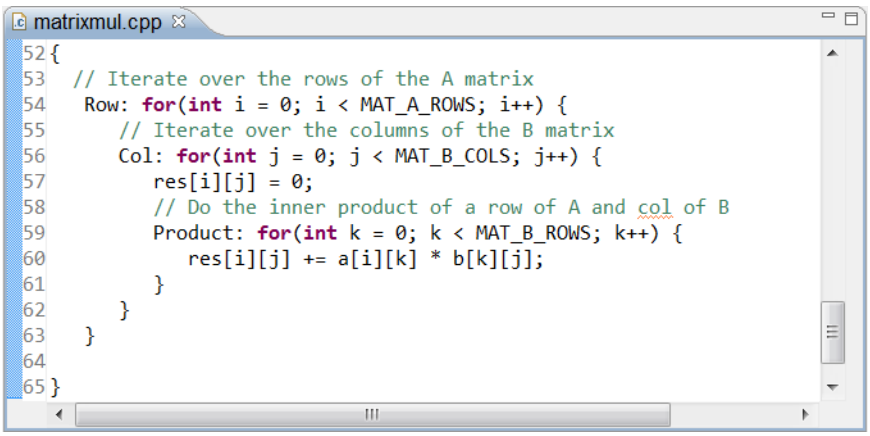


Figure 7-17: **Matrixmul Code**

Examining the code in Figure 7-17 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, solution6.

IMPORTANT: *In this step, copy the directives from solution4 as this solution does not 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 7-18.

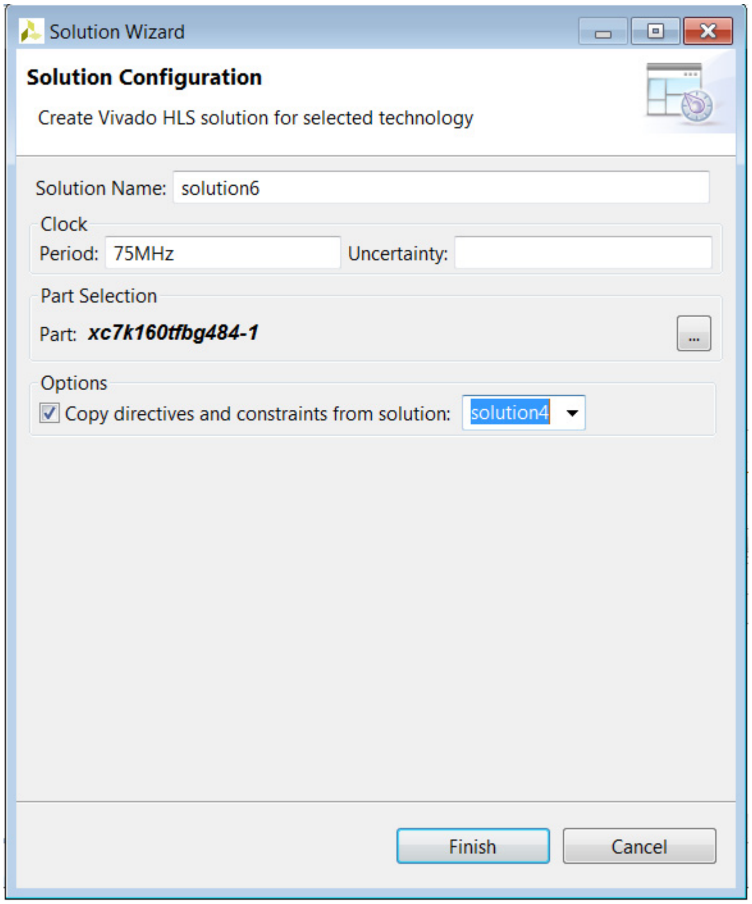


Figure 7-18: **New Solution Based on Solution4 Directives**

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 **Directive** 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 **Directive** drop-down menu at the top and select **PIPELINE**.
    f. Click **OK**.

The **Directives** tab should appear as Figure 7-19.

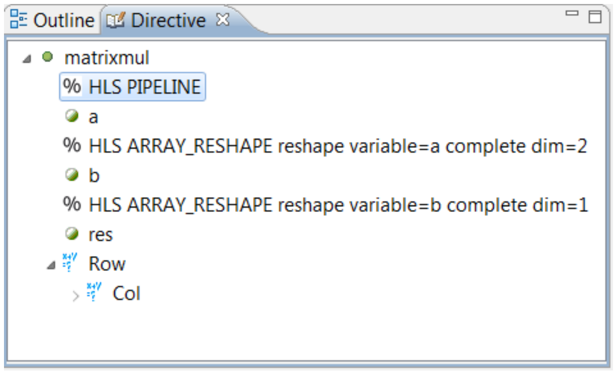


Figure 7-19: **Directives for Solution6**

6. Click the **Run C Synthesis** 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 7-20.

compare reports

Performance Estimates

Timing (ns)

Clock		solution4	solution6
ap_clk	Target	13.33	13.33
	Estimated	10.82	10.82

Latency (clock cycles)

		solution4	solution6
Latency	min	12	6
	max	12	6
Interval	min	13	5
	max	13	5

Utilization Estimates

	solution4	solution6
BRAM_18K	0	0
DSP48E	3	27
FF	40	487
LUT	59	191

Figure 7-20: **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 clock 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 7-21 shows the I/O access pattern for the code in Lab 1. Out of necessity the address values are shown in a small font.

As variables `i`, `j` and `k` iterate from 0 to 3, the lower part of Figure 7-21 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.

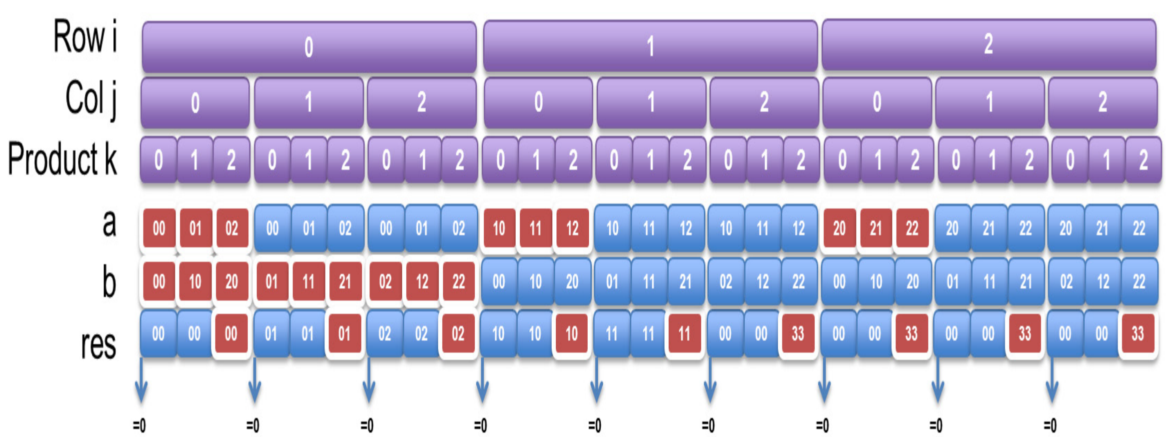


Figure 7-21: **Matrix Multiplier Address Accesses**

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 7-22.
2. Create a new **Vivado HLS** project by typing `vivado_hls -f run_hls.tcl`.

```
C:\Vivado_HLS_Tutorial\Design_Optimization\lab1>cd ..

C:\Vivado_HLS_Tutorial\Design_Optimization>cd lab2

C:\Vivado_HLS_Tutorial\Design_Optimization\lab2>vivado_hls -f run_hls.tcl
```

Figure 7-22: **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 7-23.

matrixmul.cpp

```
{
#pragma HLS ARRAY_RESHAPE variable=b complete dim=1
#pragma HLS ARRAY_RESHAPE variable=a complete dim=2
#pragma HLS INTERFACE ap_fifo port=a
#pragma HLS INTERFACE ap_fifo port=b
#pragma HLS INTERFACE ap_fifo port=res
  mat_a_t a_row[MAT_A_ROWS];
  mat_b_t b_copy[MAT_B_ROWS][MAT_B_COLS];
  int tmp = 0;

  // Iterate over the rowa of the A matrix
  Row: for(int i = 0; i < MAT_A_ROWS; i++) {
    // Iterate over the columns of the B matrix
    Col: for(int j = 0; j < MAT_B_COLS; j++) {
#pragma HLS PIPELINE
      // Do the inner product of a row of A and col of B
      tmp=0;
      // Cache each row (so it's only read once per function)
      if (j == 0)
        Cache_Row: for(int k = 0; k < MAT_A_ROWS; k++)
          a_row[k] = a[i][k];

       // Cache all cols (so they are only read once per function)
     if (i == 0)
            Cache_Col: for(int k = 0; k < MAT_B_ROWS; k++)
               b_copy[k][j] = b[k][j];

      Product: for(int k = 0; k < MAT_B_ROWS; k++) {
        tmp += a_row[k] * b_copy[k][j];
```

Figure 7-23: **C Code with Updated I/O 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 row 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 Synthesis** 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.

Lab 1 Description

Perform RTL verification steps and understand the importance of the C test bench in verifying the RTL.

Lab 2 Description

Create RTL trace files and analyze them using the Vivado Design Suite.

Lab 3 Description

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. See the information in Locating the Tutorial Design Files.

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.
   - On Windows use **Start > All Programs > Xilinx Design Tools > Vivado 2015.3 > Vivado HLS > Vivado HLS 2015.3 Command Prompt** (Figure 8-1).
   - On Linux, open a new shell.

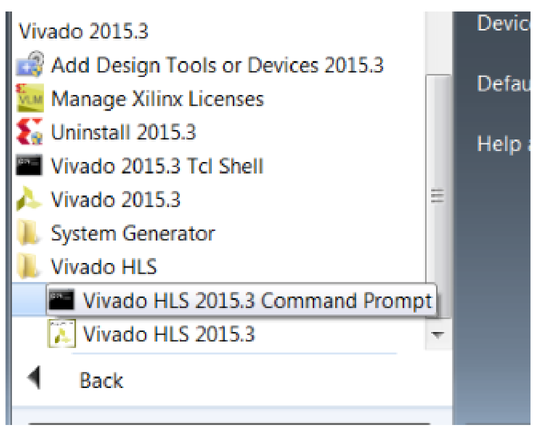


Figure 8-1: **Vivado HLS Command Prompt**

2. Using the command prompt window (Figure 8-2), 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 8-2.

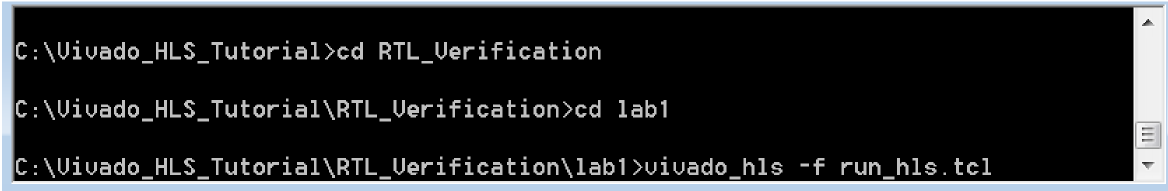


Figure 8-2: **Setup the RLTL Verification Tutorial Project**

4. When Vivado HLS completes, open the project in the Vivado HLS GUI using the command `vivado_hls -p duc_prj`, as shown in Figure 8-3.

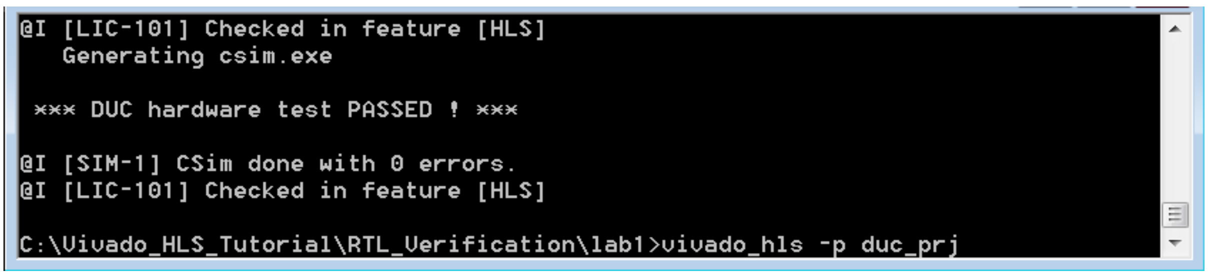


Figure 8-3: **Open RTL Verification Project for Lab 1**

Step 2: Perform RTL Verification

1. Click the **Run C Synthesis** toolbar button to synthesize the design to RTL.
2. When synthesis completes, use the Run **C/RTL Cosimulation** toolbar button (Figure 8-4) to launch the Cosimulation dialog box.

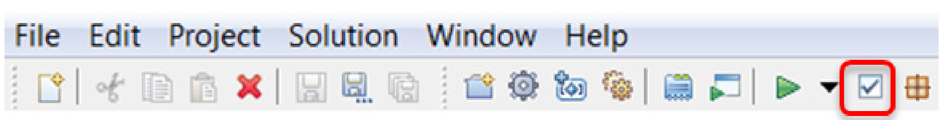


Figure 8-4: **Run C/RTL Cosimulation Toolbar Button**

The Cosimulation Dialog box opens, as shown in Figure 8-5.

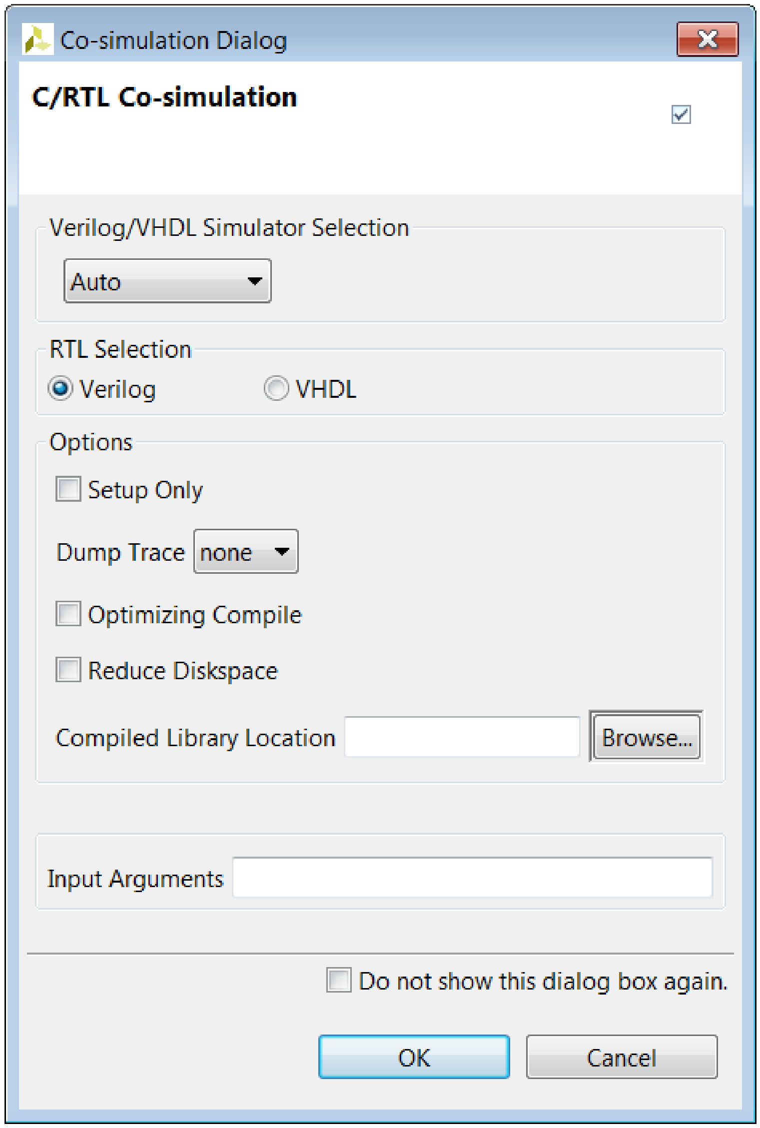
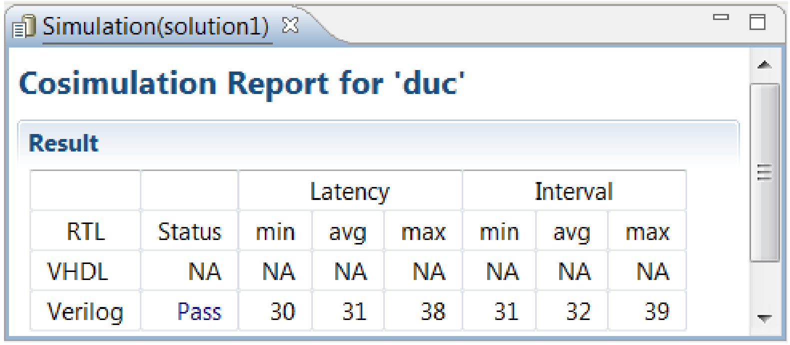


Figure 8-5: **Cosimulation Dialog Box**

The drop-down menu allows you to select the RTL simulator for HDL simulation. For this exercise, you use the default Vivado Simulator with Verilog RTL for cosimulation.

3. Click **OK** to start RTL verification.

When RTL Verification completes, the simulation report opens automatically (Figure 8-6). The report indicates if the simulation passed or failed. In addition, the report indicates the measured latency and interval.



Cosimulation Report for 'duc'

Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	30	31	38	31	32	39

Figure 8-6: **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 8-7).
2. Double-click `duc_test.c` to open the C test bench in the Information pane.

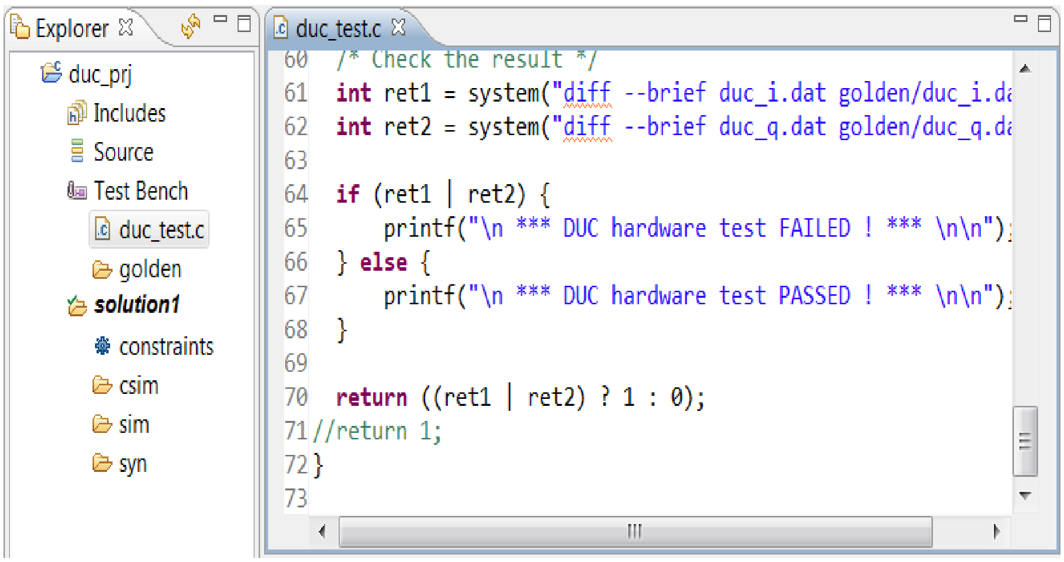


Figure 8-7: **RTL Test Bench**

3. Scroll to the end of the file to see the code shown in Figure 8-8.
4. Edit the return statement to match Figure 8-8 and ensure the test bench always returns the value 1.

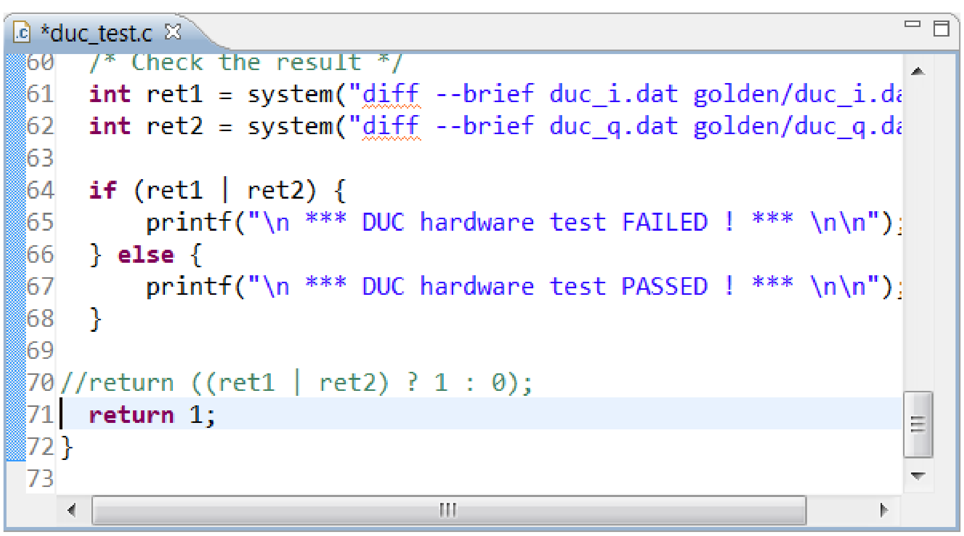


Figure 8-8: **Modified RTL Test Bench**

5. Save the file.
6. Click the **Run C Synthesis** 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 8-9).

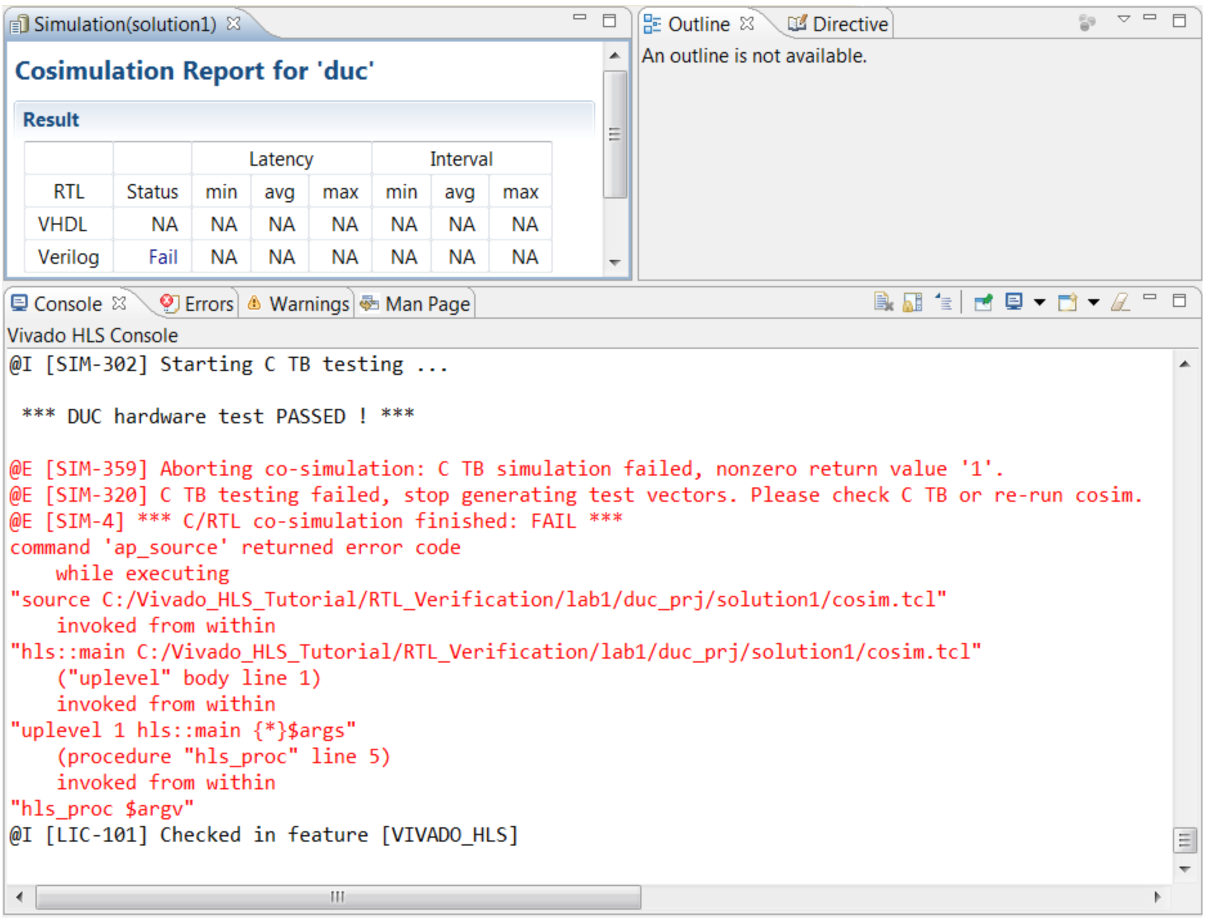


Figure 8-9: **Cosimulation Report Failure**

In Figure 8-9, 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 8-10 shows the solution directory, with the output files highlighted.

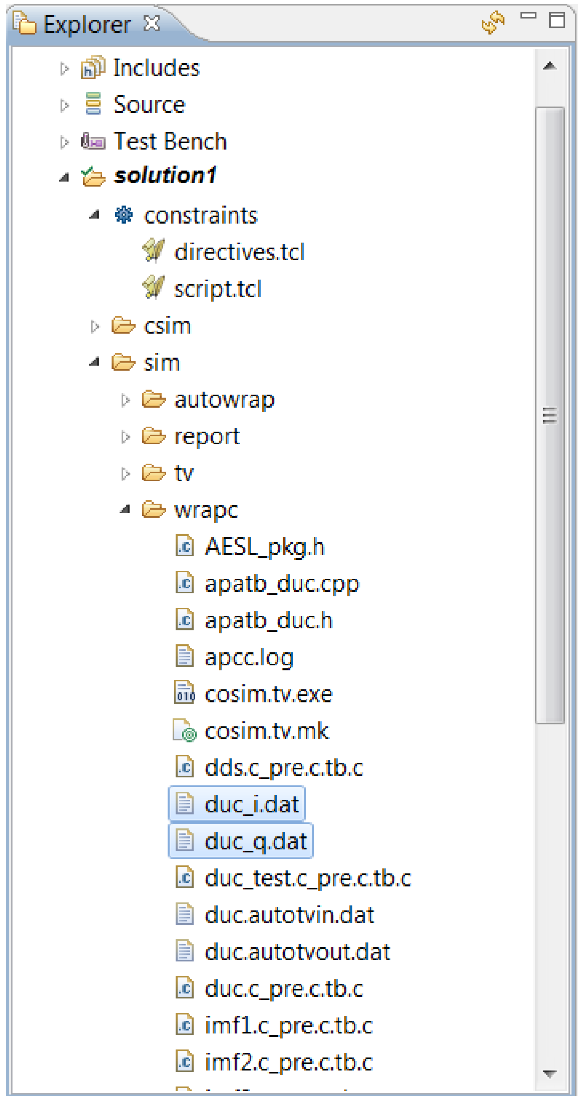


Figure 8-10: **Cosimulation Output Files**

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 Vivado Simulator

1. From the Vivado HLS command prompt you used in Lab 1, change to the lab2 directory as shown in Figure 8-11.
2. Create a new Vivado HLS project by typing `vivado_hls -f run_hls.tcl`.

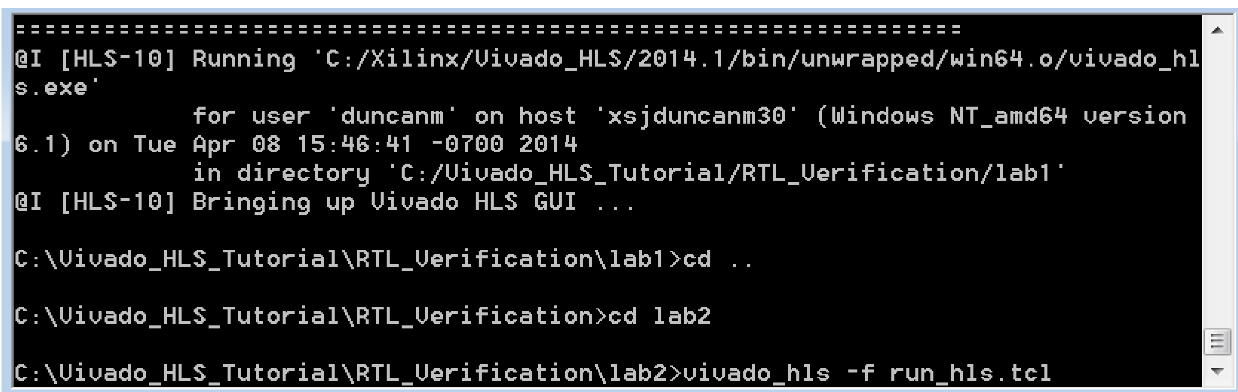


Figure 8-11: **Setup for RTL Verification Lab 2**

3. Open the Vivado HLS GUI project by typing `vivado_hls -p duc_prj`.
4. Click the **Run C Synthesis** toolbar button to synthesize the design to RTL.
5. Click the **Run C/RTL Cosimulation** toolbar button to launch the Cosimulation Dialog box.

In this case, you will produce a trace file you can open using the Vivado Simulator.

6. In the Co-simulation Dialog box:
   a. Leave the default auto selection (using Vivado Simulator and Verilog).
   b. Activate the **Dump Trace** drop-down menu and select the **all** option, to have the options shown in Figure 8-12.

c. Click **OK** to execute RTL cosimulation.

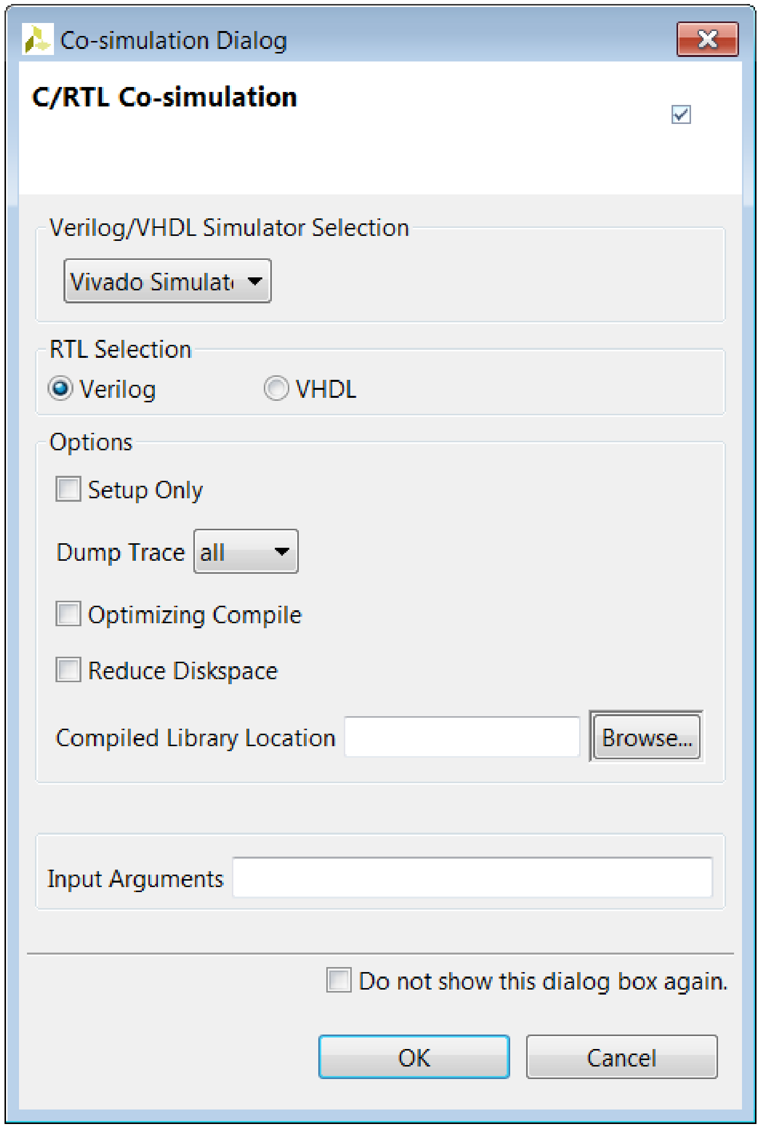


Figure 8-12: **Cosimulation Dialog Box for Lab 2**

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 Vivado Simulator simulator option and because Verilog was selected, two trace files are now present in the Verilog simulation directory. These are shown highlighted in Figure 8-13.

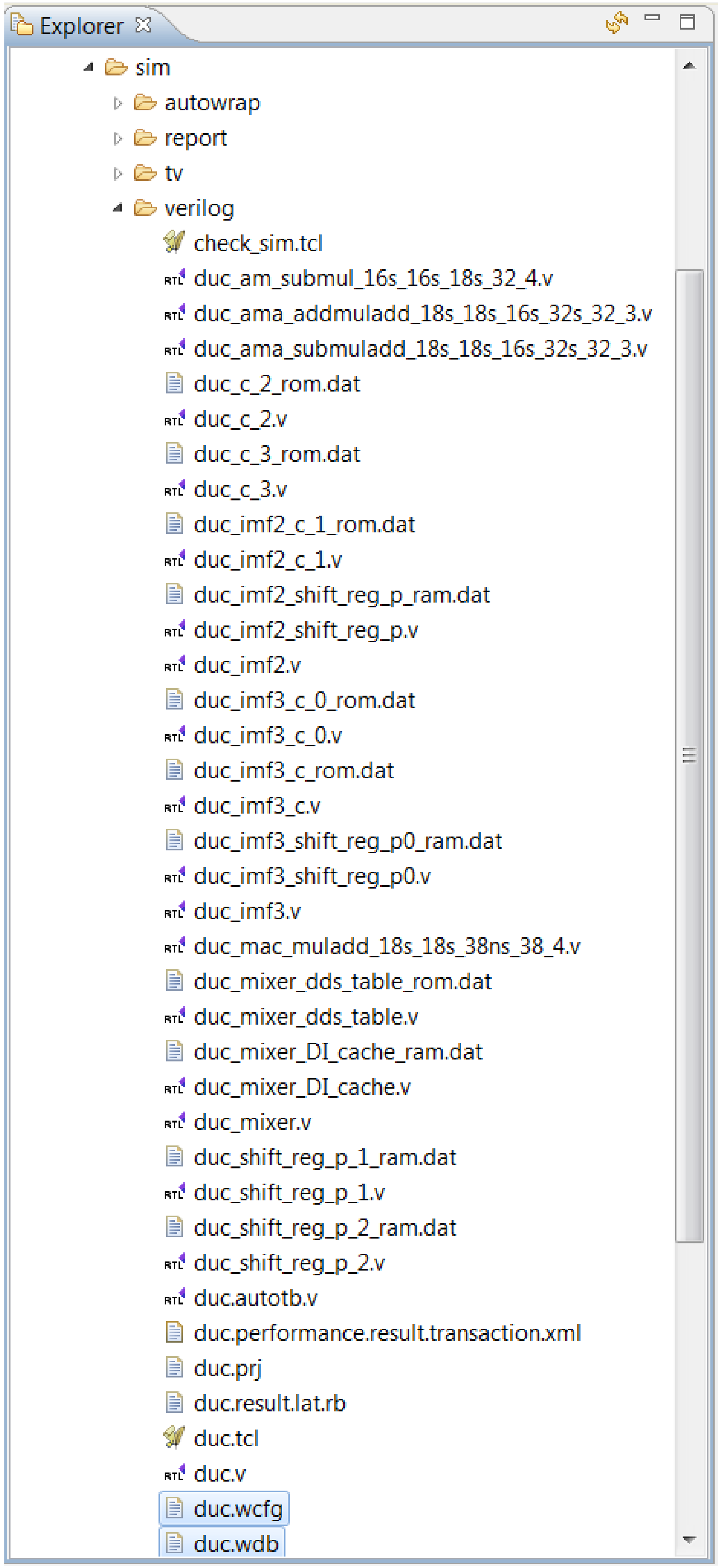


Figure 8-13: **Verilog Vivado Simulator Cosimulation Results**

The next step is to view the trace files inside the Vivado Design Suite.

Since waveform trace data has been generated for the Vivado Simulator, the **Open Wave Viewer** toolbar button is now highlighted, as shown in Figure 8-14.

Note: The **Open Wave Viewer** toolbar button cab only be used when Vivado Simulator is selected as the Verilog/VHDL Simulator.

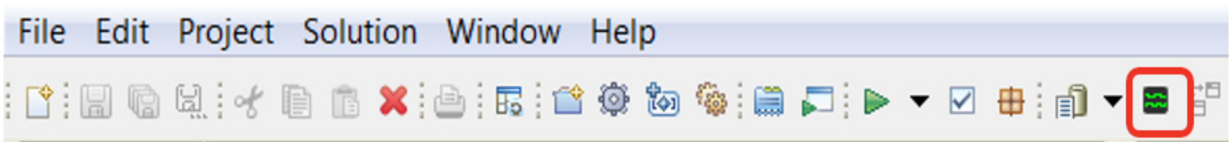


Figure 8-14: **Opening the Trace File in Vivado**

7. Click on the **Open Wave Viewer** toolbar button to open the Vivado IDE with the RTL waveforms traces.

Note: The only functionality provided by the Vivado IDE by this action is the viewing and analysis of RTL waveforms.

You can then view the waveforms in the waveform viewer. Figure 8-15 shows the zoomed waveforms where the output data ports and their associated I/O protocol signals (output valid signals) are shown highlighted.

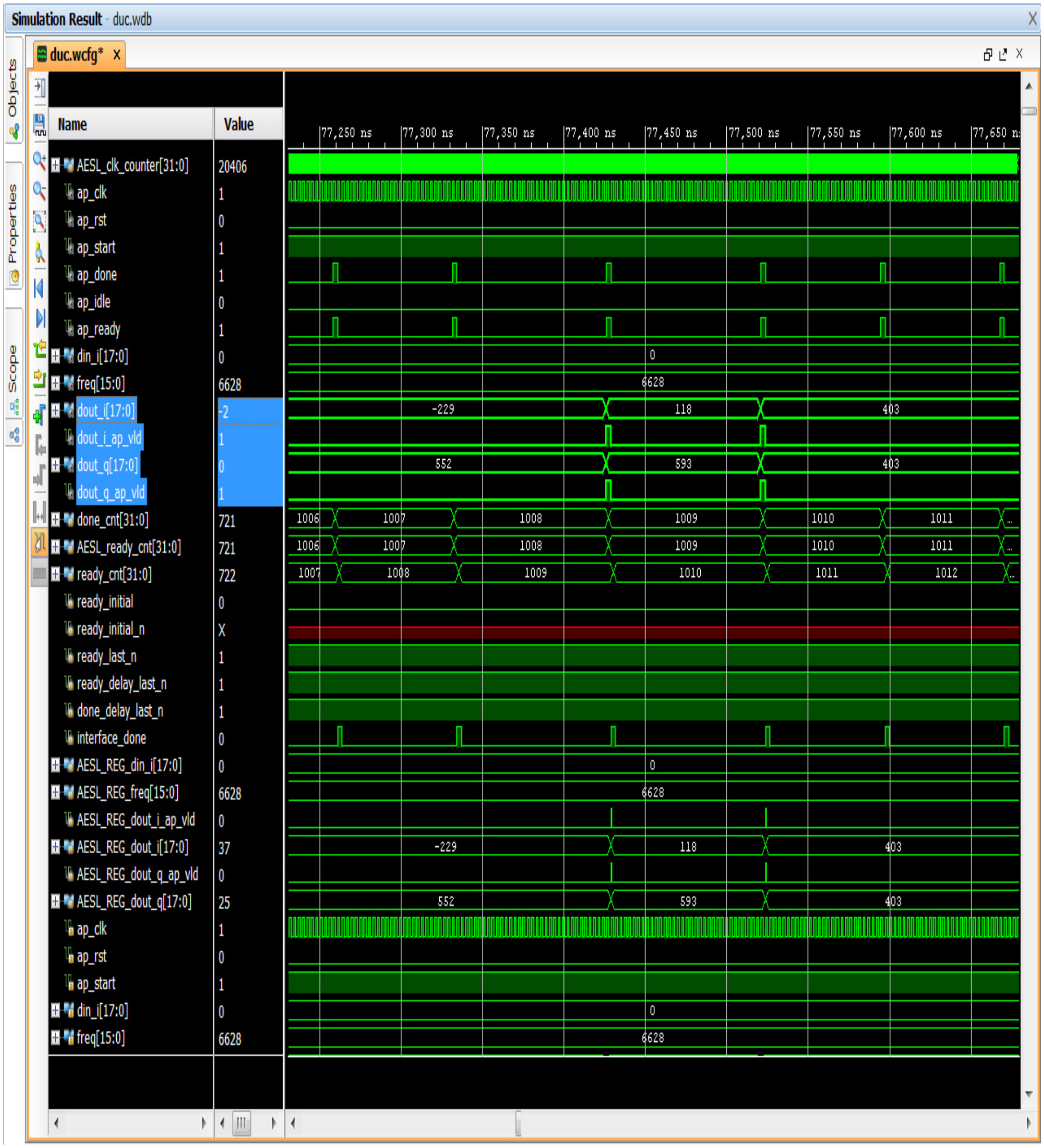


Figure 8-15: **Analyzing the RTL Trace File**

8. Exit the Vivado IDE.
9. Exit and close the Vivado GUI.
10. 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 Synthesis** 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 box:
   a. Select **ModelSim** from the Verilog/VHDL Simulator Selector.
   b. Select **VHDL**.
   c. Activate the **Dump Trace** drop-down menu and select the **all** option, to have the options shown in Figure 8-16.
   d. Click **OK** to execute RTL cosimulation.

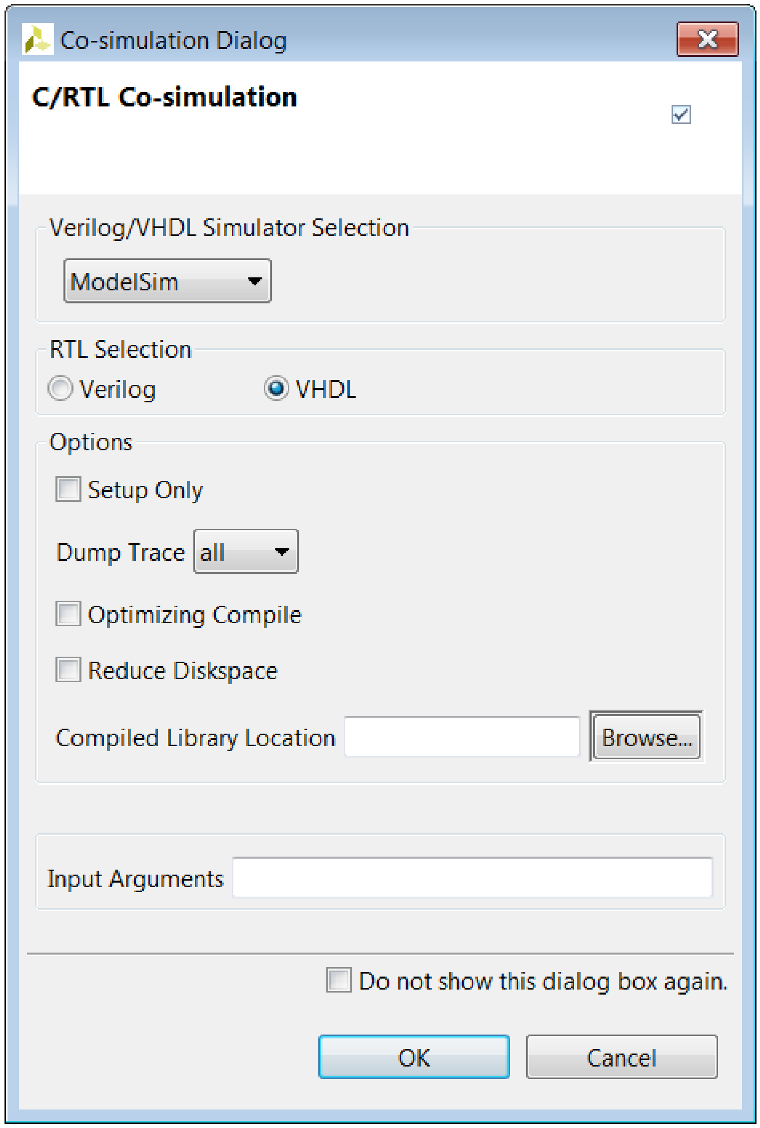


Figure 8-16: **Cosimulation Dialog Box for Lab 3**

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 8-17.

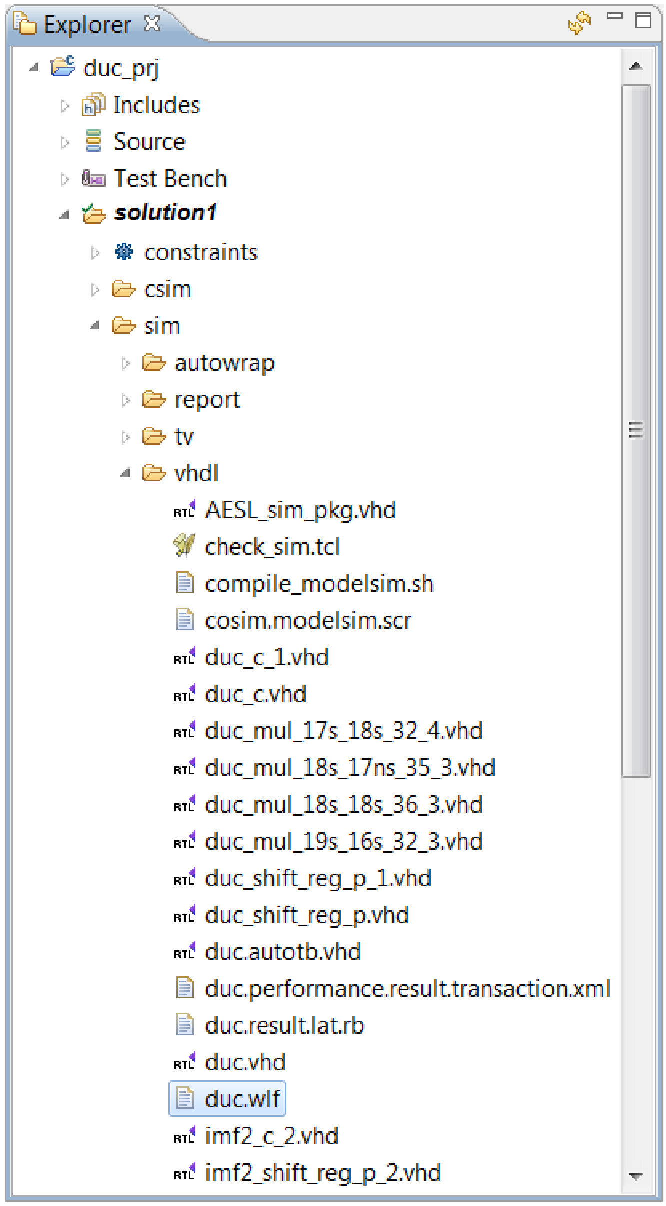


Figure 8-17: **VHDL ModelSim Trace File**

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 8-18).

4. Navigate to the VHDL simulation directory and select `duc.wlf`.
5. Click **Open**.

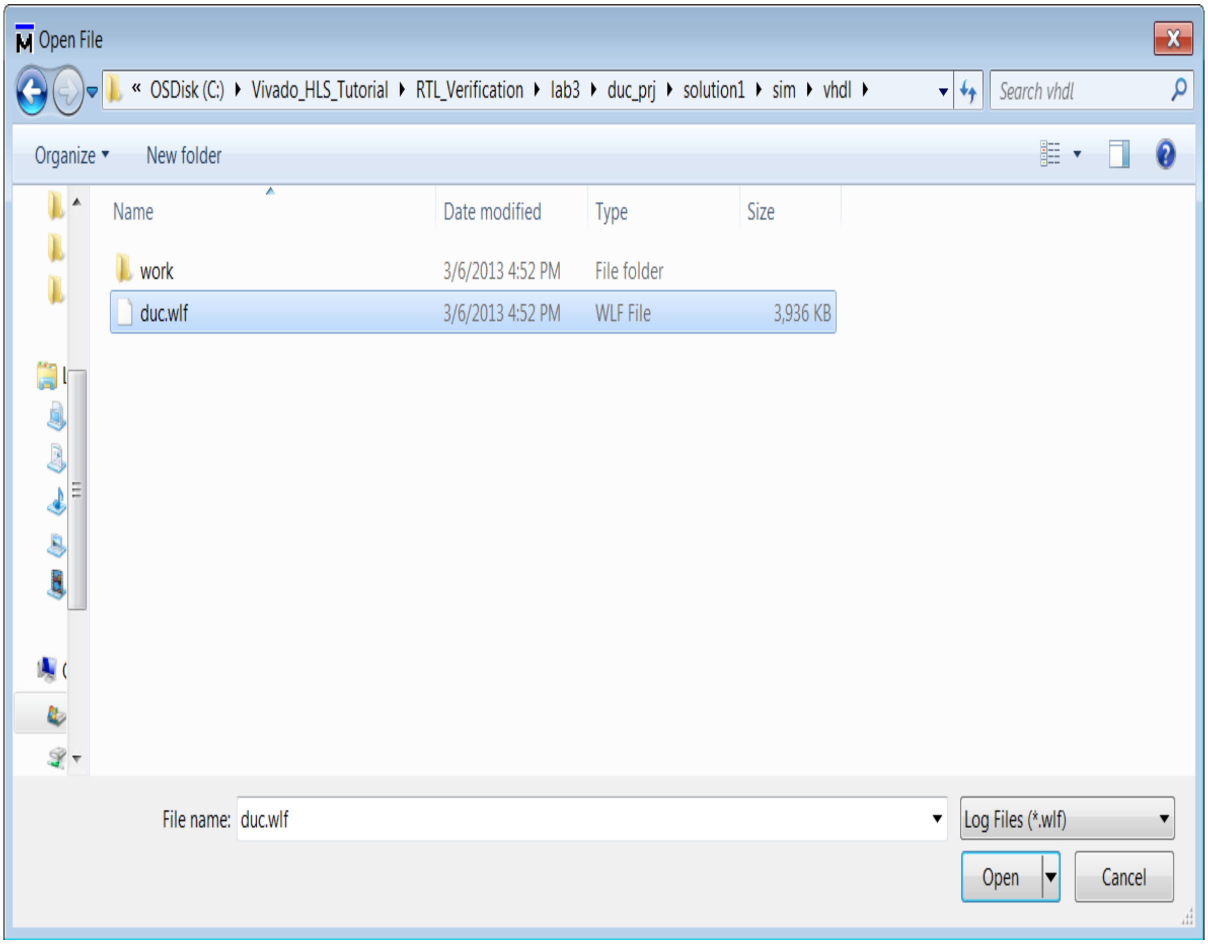


Figure 8-18: **ModelSim Open File WLF**

6. Add the signals to the trace window and adjust to see a view similar to Figure 8-19.

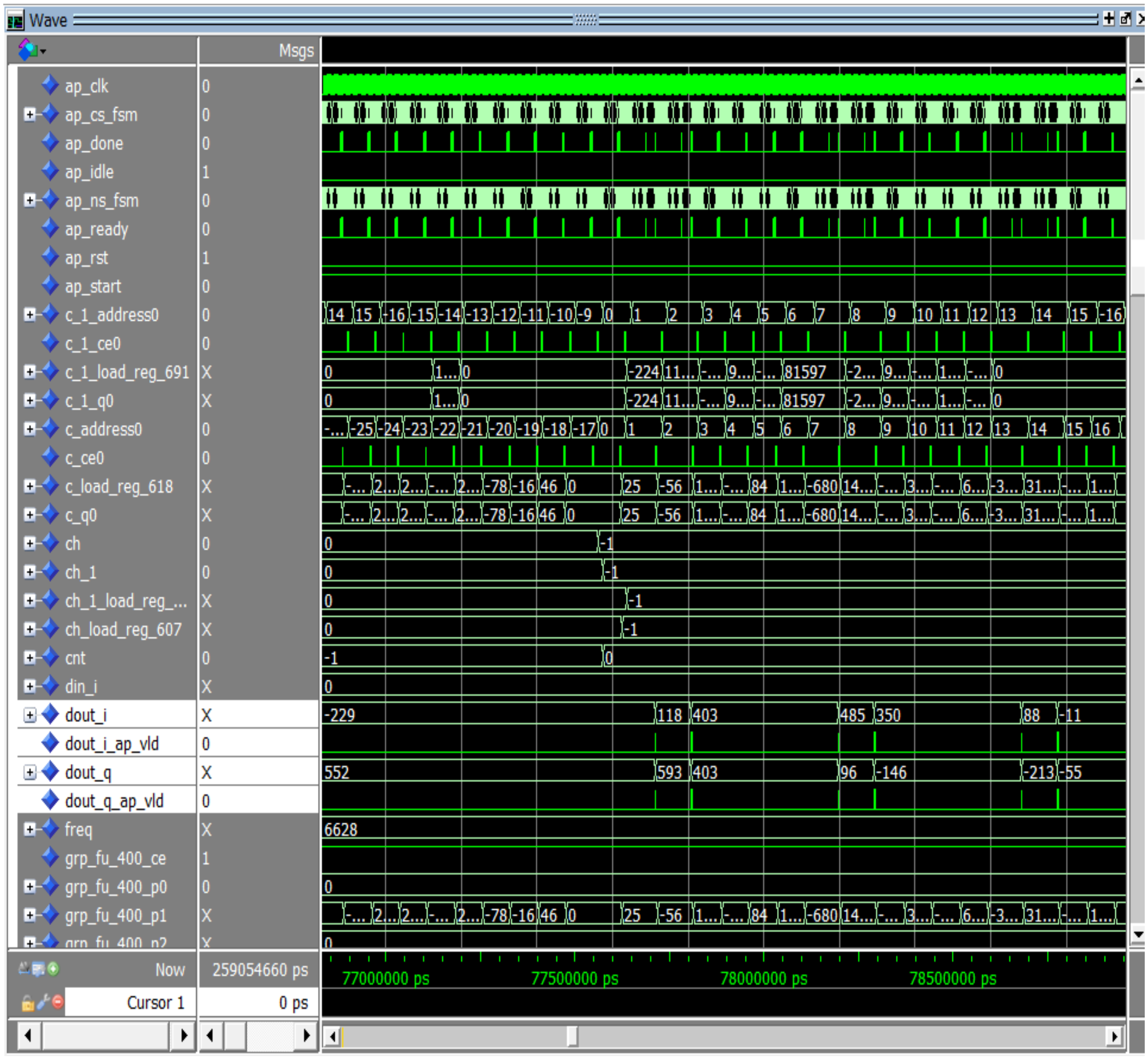


Figure 8-19: **Viewing the Trace File in ModelSim**

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.

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 to take HLS IP and use it in IP Integrator as part of a larger design.

This tutorial consists of a single lab exercise.

Lab 1 Description

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. See the information in Locating the Tutorial Design Files.

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 two HLS IP blocks are combined with a Xilinx IP FFT in IP Integrator and the design verified 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.
   - On Windows use **Start > All Programs > Xilinx Design Tools > Vivado 2015.3 > Vivado HLS > Vivado HLS 2015.3 Command Prompt**.
   - On Linux, open a new shell.

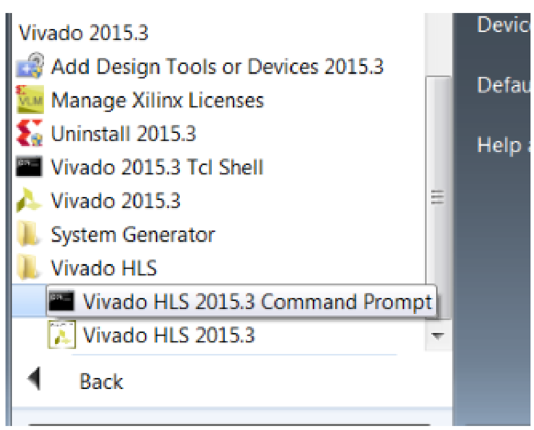


Figure 9-1: **Vivado HLS Command Prompt**

2. Using the command prompt window, change the directory to `Vivado_HLS_Tutorial\Using_IP_with_IPI\lab1\hls_designs` (Figure 9-2).
3. Type `vivado_hls -f run_hls.tcl` to create the HLS IP (Figure 9-2).

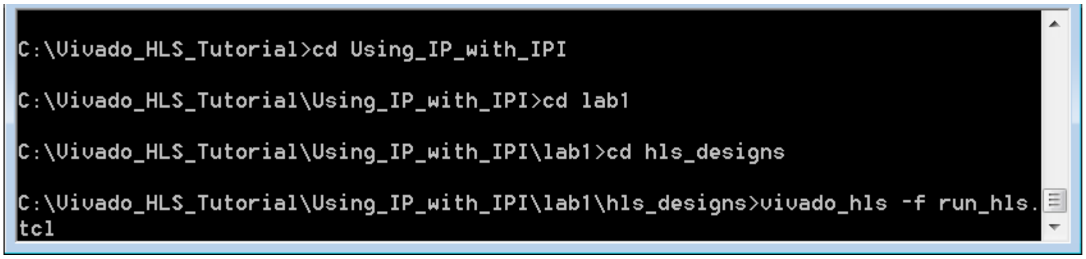


Figure 9-2: **Create the HLS Design for IPI**

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 "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/`

The remainder of this tutorial 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):
    - On Windows use **Start > All Programs > Xilinx Design Tools > Vivado 2015.3 > Vivado 2015.3**.
    - On Linux, type vivado in the shell.
2. From the Welcome screen, click **Create New Project** (Figure 9-3).

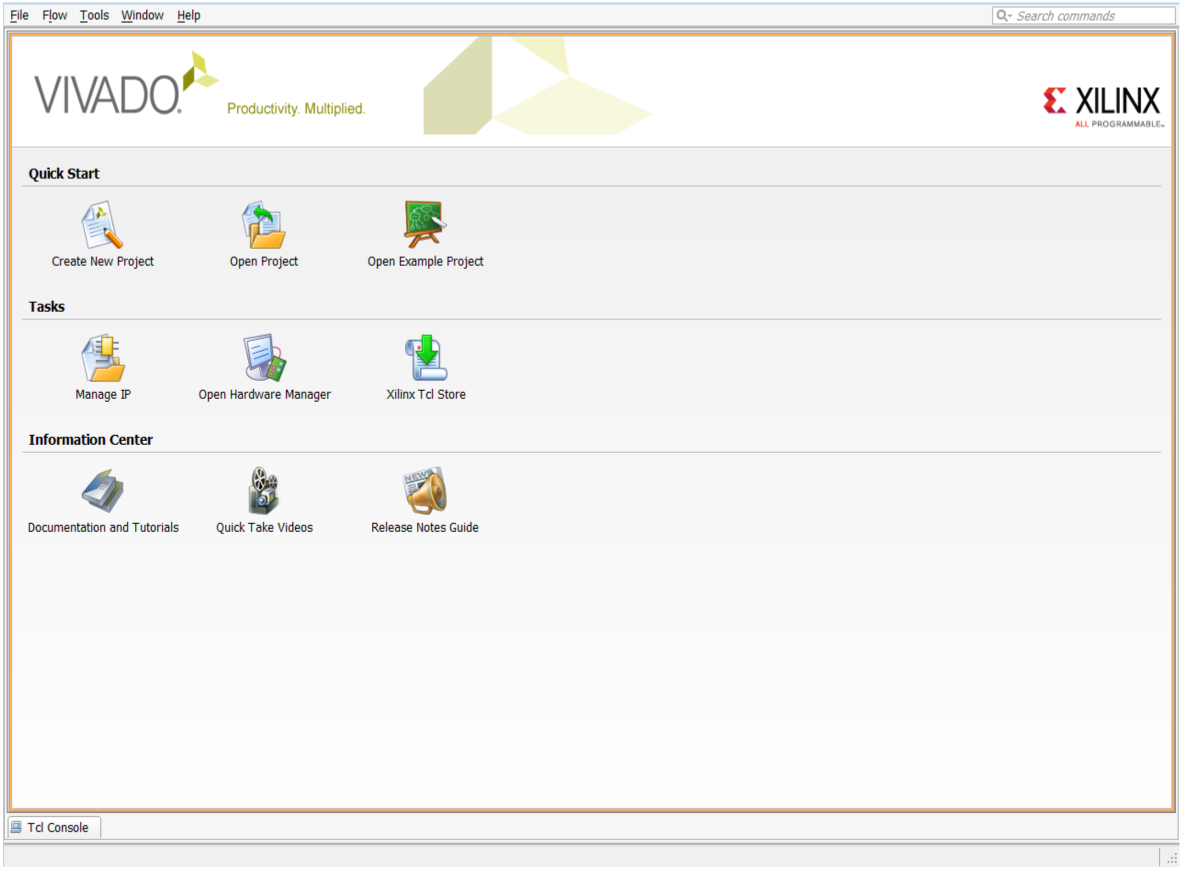


Figure 9-3: **Create a Vivado Project**

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 and select the tutorial directory (Figure 9-4).

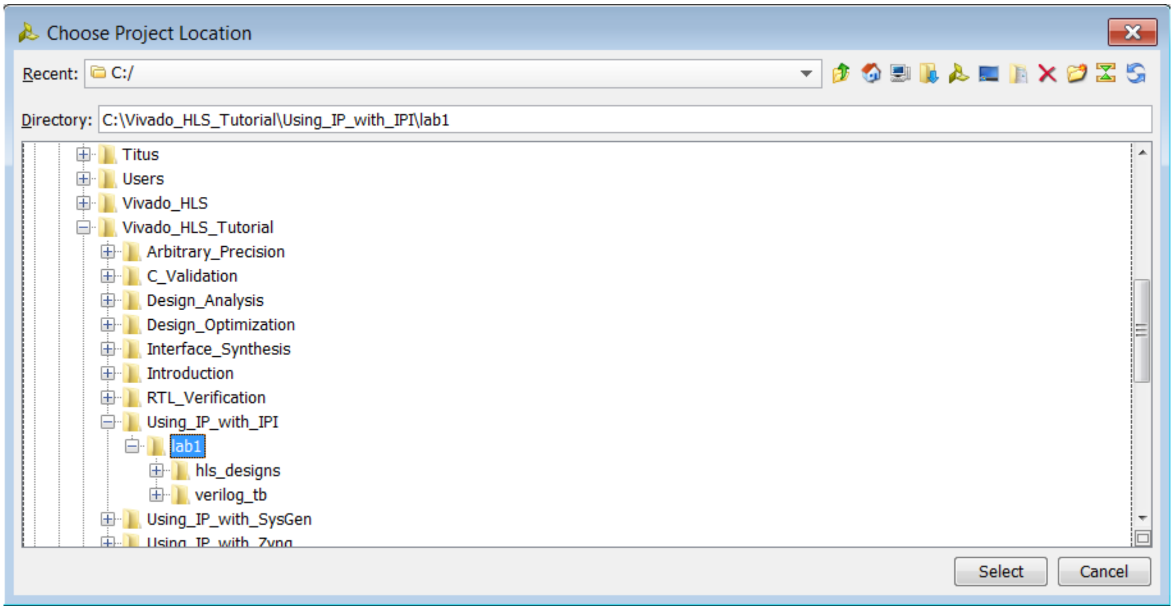


Figure 9-4: **Path to the Vivado Design Suite Project**

5. Click **Next** to move to the Project Type page of the wizard.
   a. Select **RTL Project**.
   b. Select **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 9-5 and press Next.

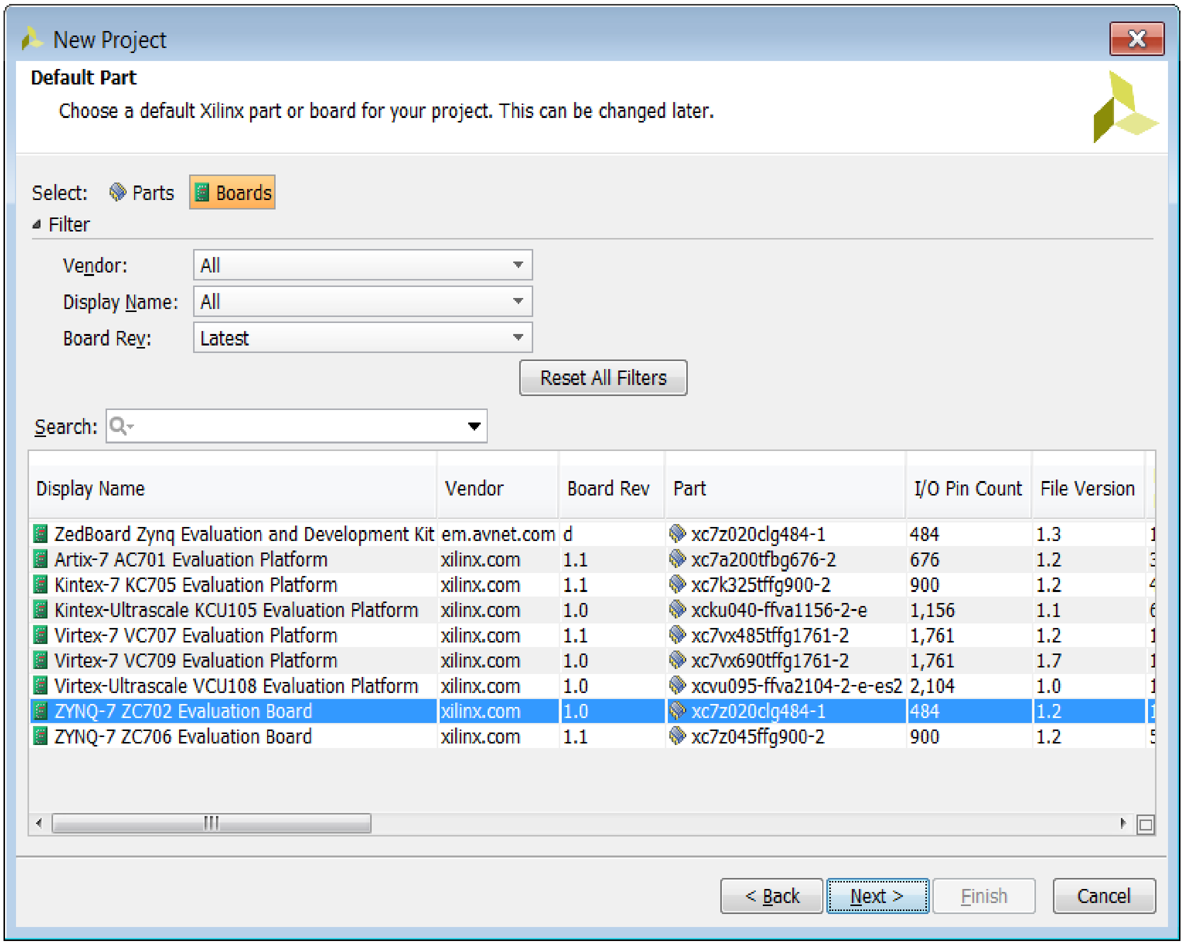


Figure 9-5: **Vivado Project Specification**

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 9-6.

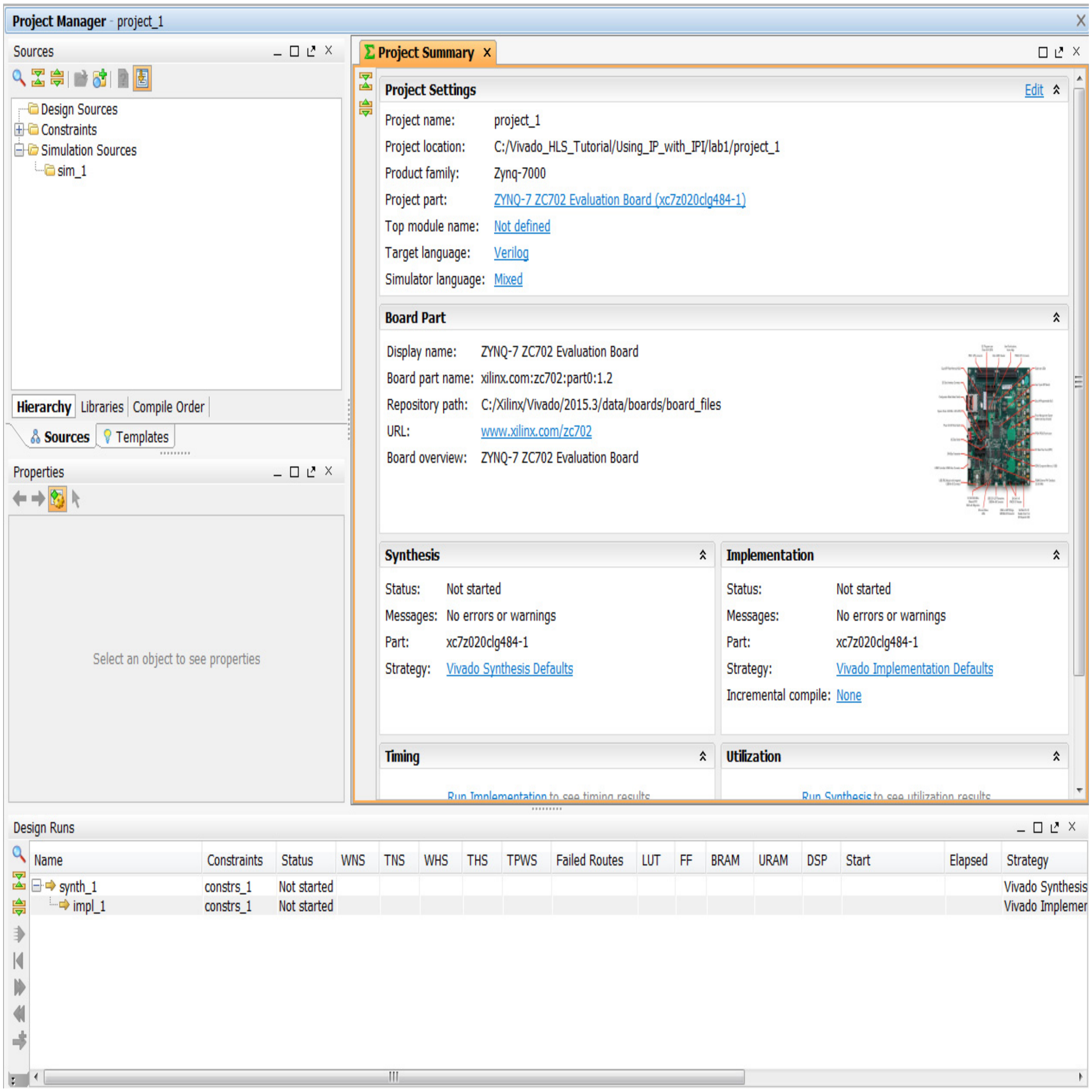


Figure 9-6: **Vivado Project**

Step 3: Add HLS IP to an IP Repository

1. In the Project Manager area of the Flow Navigator pane, click **IP Catalog**.

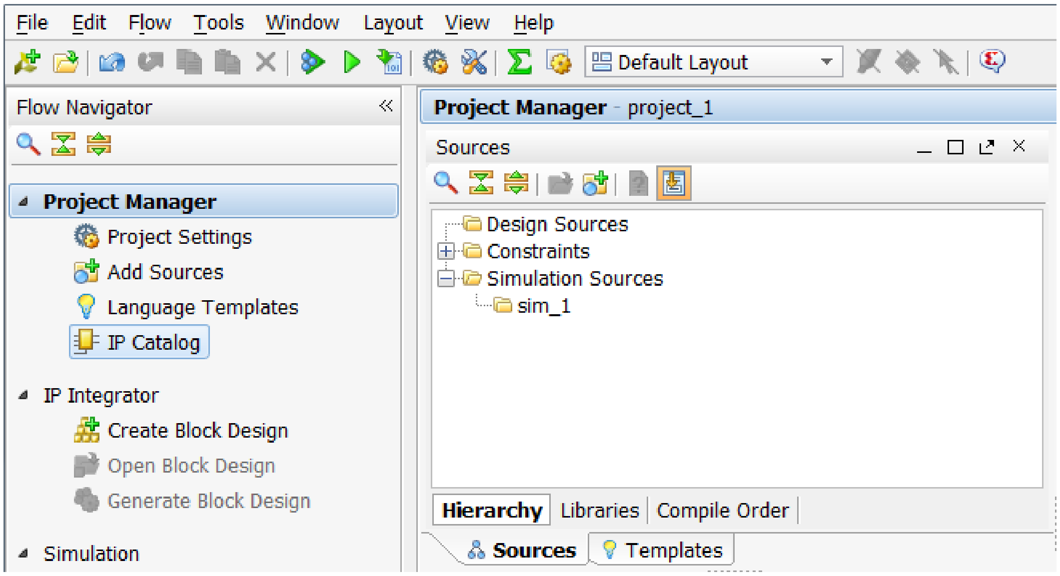


Figure 9-7: **Open the IP Catalog**

2. The IP Catalog appears in the main pane of the workspace. Click the **IP Settings** icon.

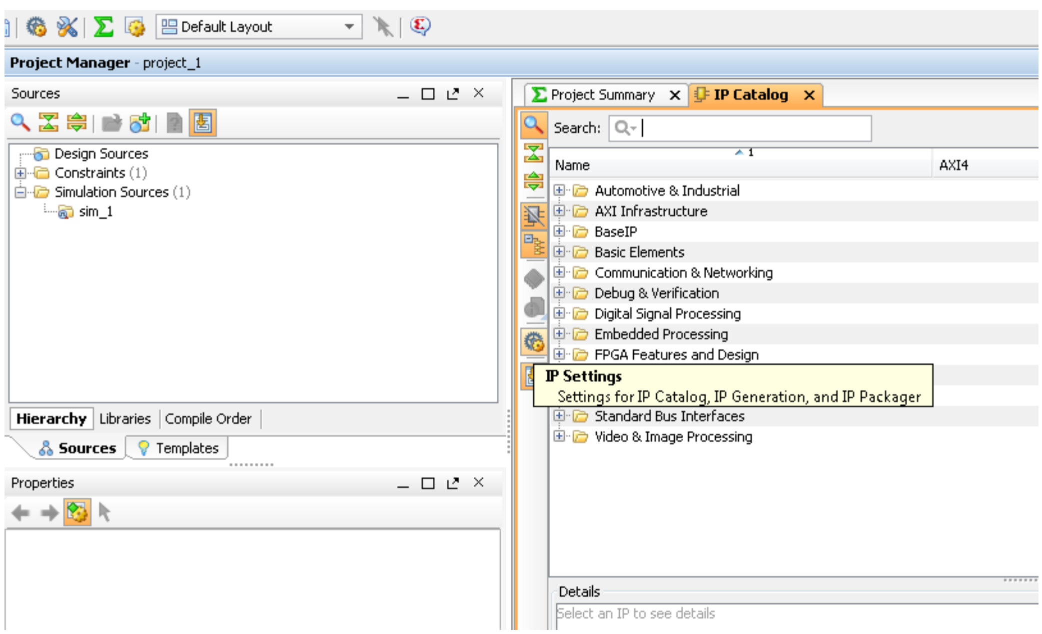


Figure 9-8: **Open the IP Catalog Settings**

3. In the IP section of the Project Settings dialog, select the **Repository Manager** tab and click on the "+" symbol to **Add Repository**.

4. In the IP Repositories dialog:

   a. Browse to the tutorial directory, `Using_IP_with_IPI\lab1\hls_designs\fe_vhls_prj\IPXACTExport\impl\ip` as shown in Figure 9-9.

b. Click **Select** to close the IP Repositories window.

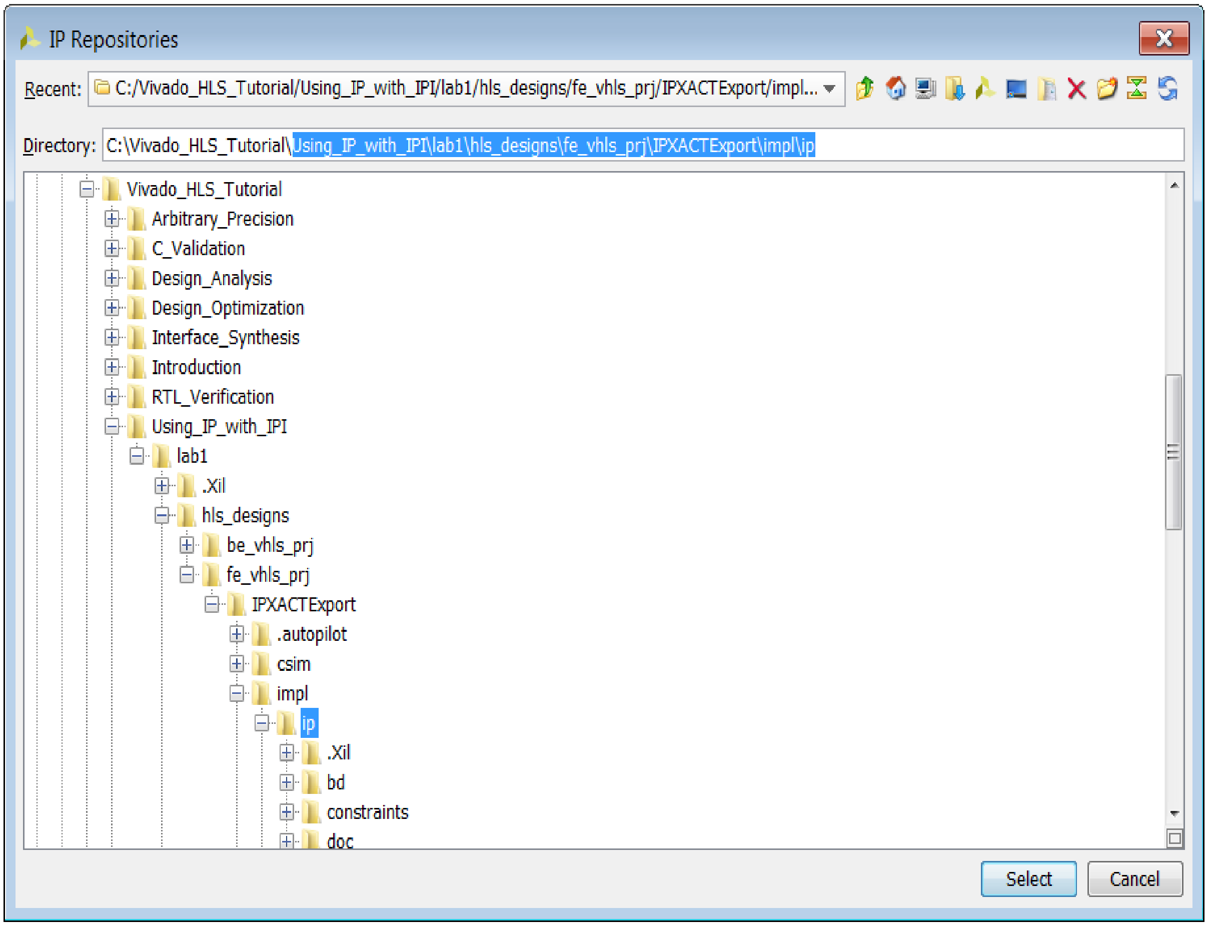


Figure 9-9: **Create a New IP Repository**

5. Select **OK** to accept the new repository.
6. Follow the same procedure to add the second HLS IP package: `lab1/hls_designs/be_vhls_prj/IPXACTExport/impl/ip/`.
7. Click **OK** to exit the dialog box.

A Vivado HLS IP category now appears in the IP Catalog as HLS IP (Figure 9-10).

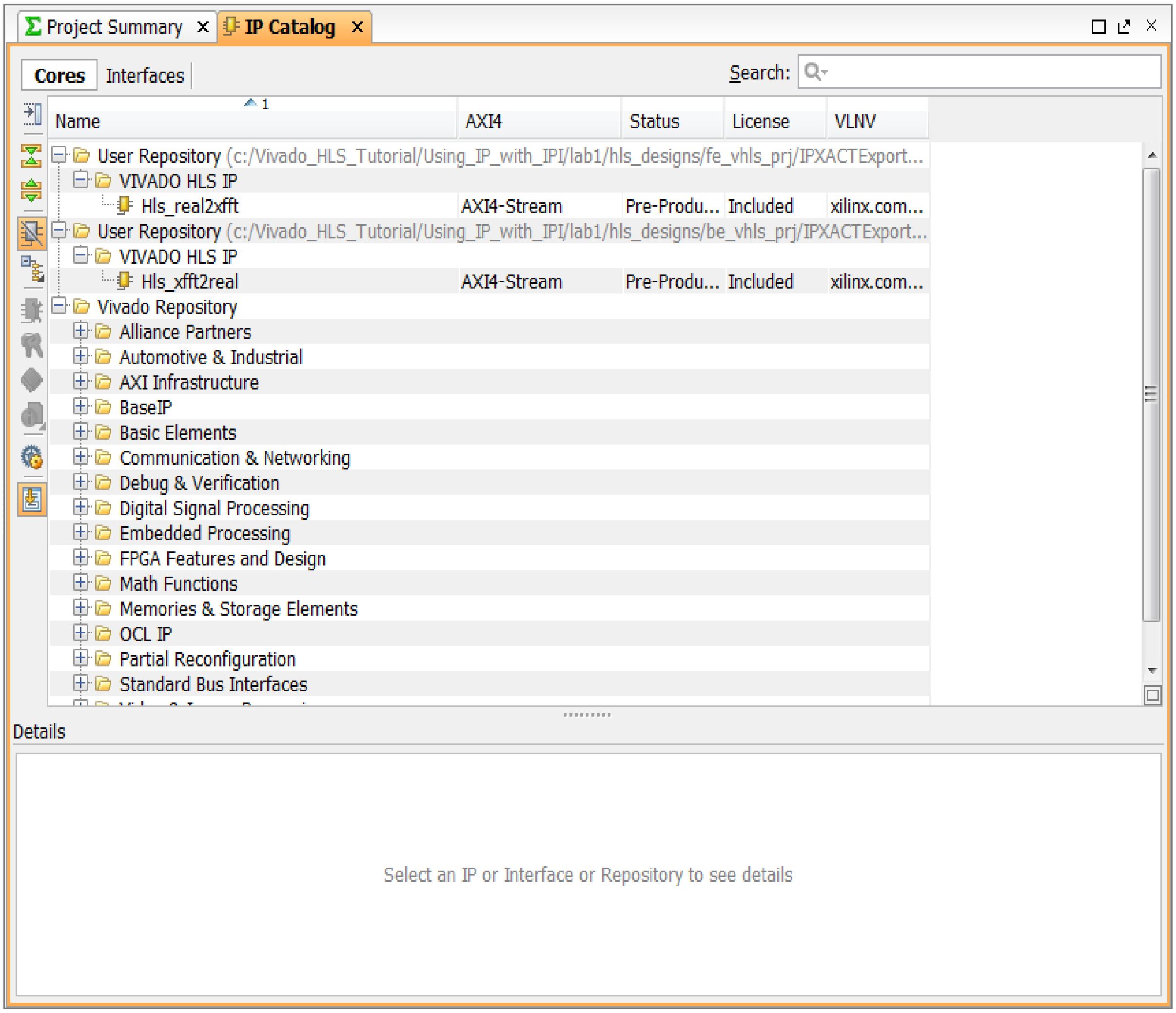


Figure 9-10: **IP Catalog with HLS IP**

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**.

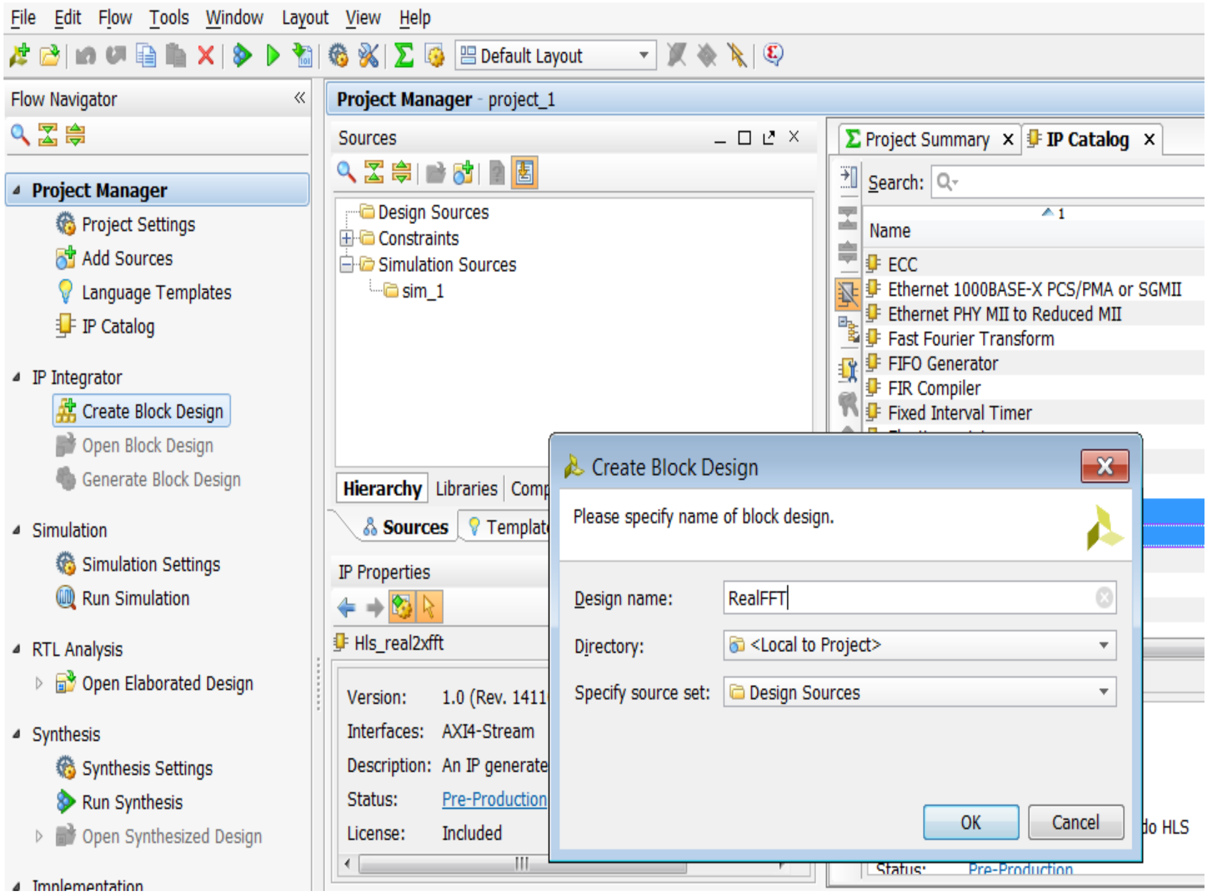


Figure 9-11: **Create Block Diagram**

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 (Figure 9-12).
   a. In the Search box type `fourier`.
   b. **Select** Fast Fourier Transform.
   c. Press **Enter**.

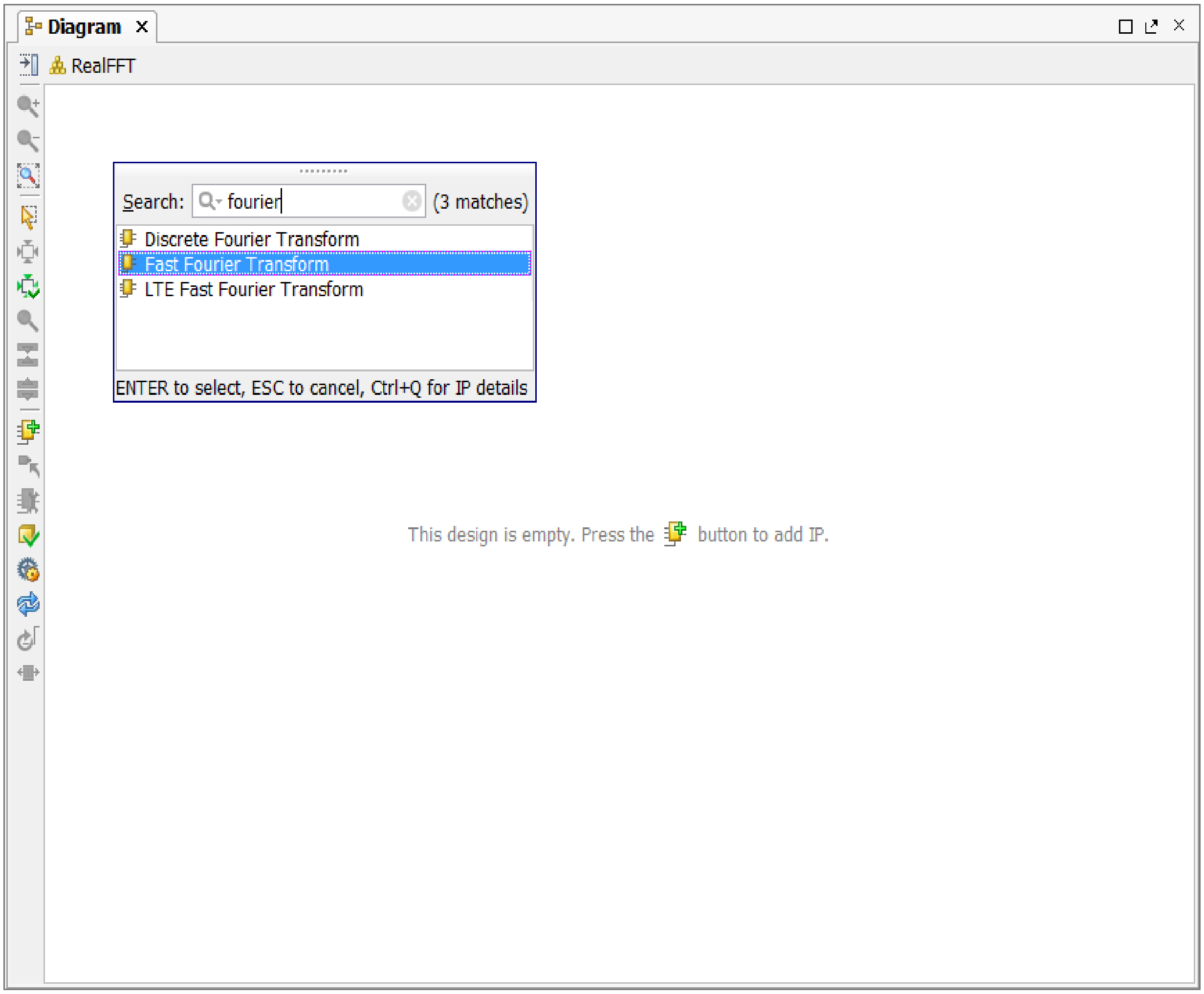


Figure 9-12: **Add the Xilinx FFT IP**

The Xilinx IP block FFT is now instantiated in the design, as shown in Figure 9-13.

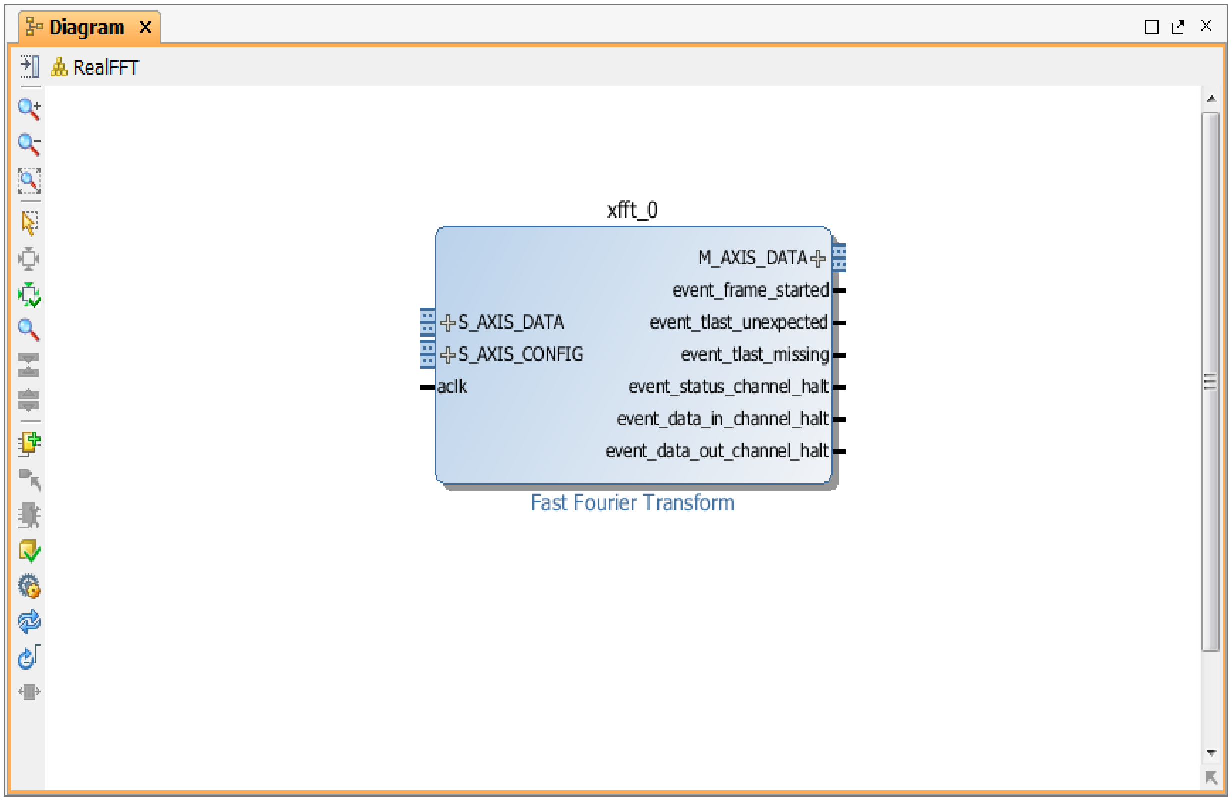


Figure 9-13: **Xilinx FFT IP**

3. Double-click the new Fast Fourier Transform IP symbol to open the Re-customize IP dialog box.
4. On the **Configuration** tab (Figure 9-14):
   a. Change the **Transform Length** to 512.
   b. Select **Pipelined, Streaming I/O** in the Architecture Choice section.

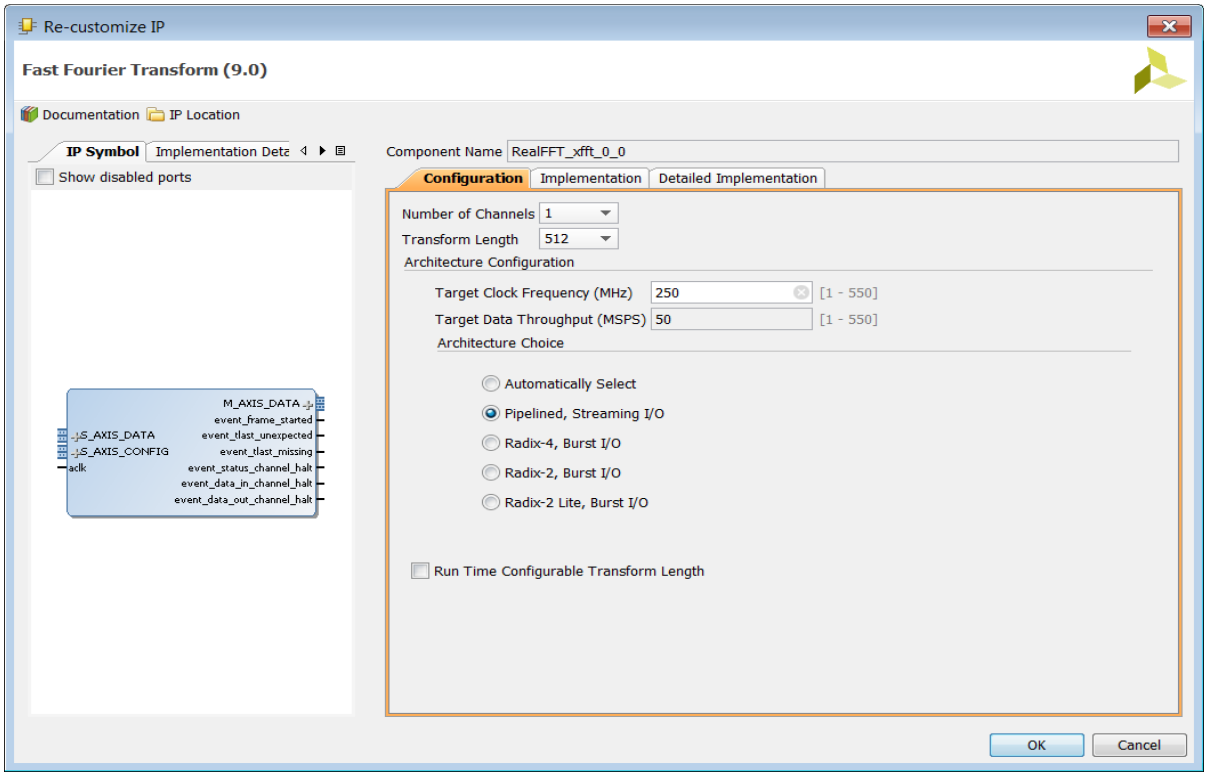


Figure 9-14: **Xilinx FFT Configuration**

5. Select the **Implementation** tab (Figure 9-15):
    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.

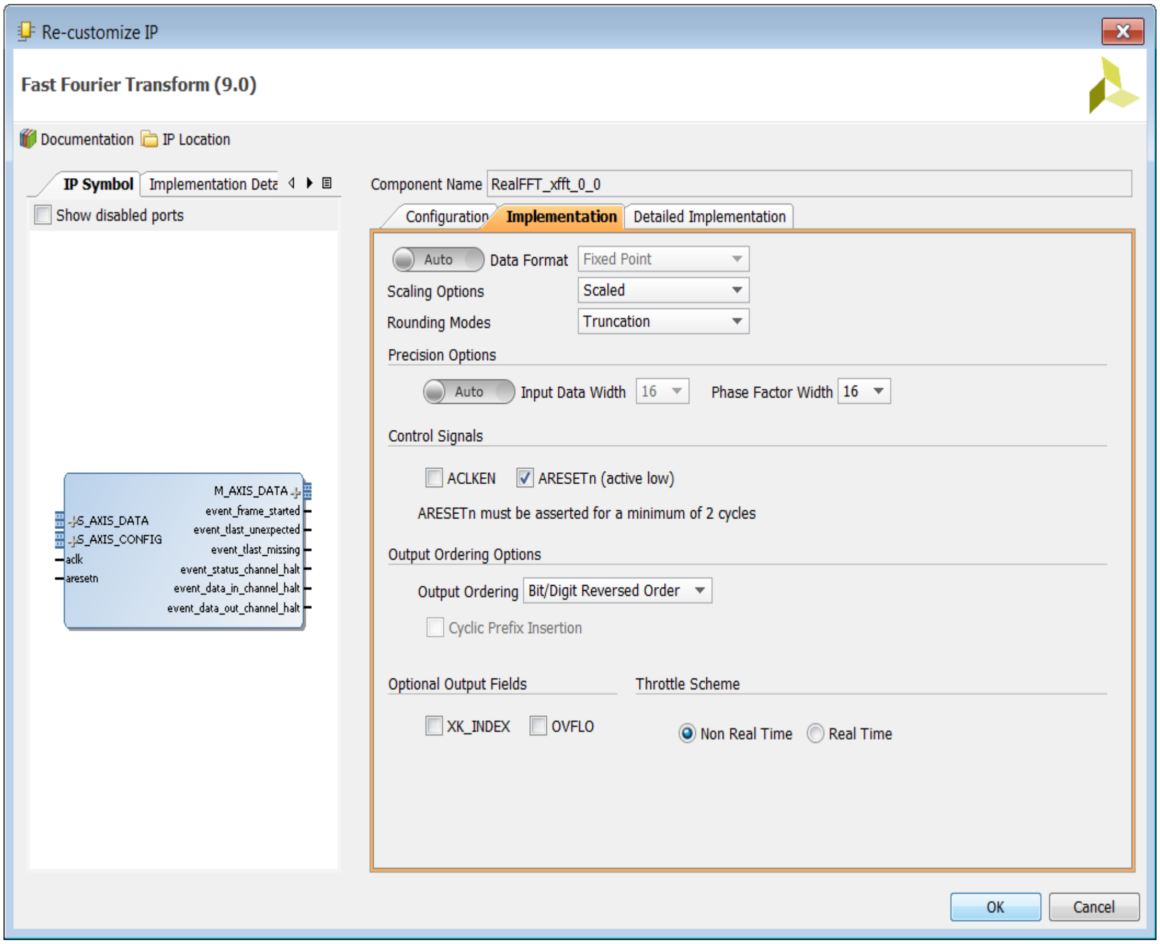


Figure 9-15: **Xilinx FFT Implementation**

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 9-16).

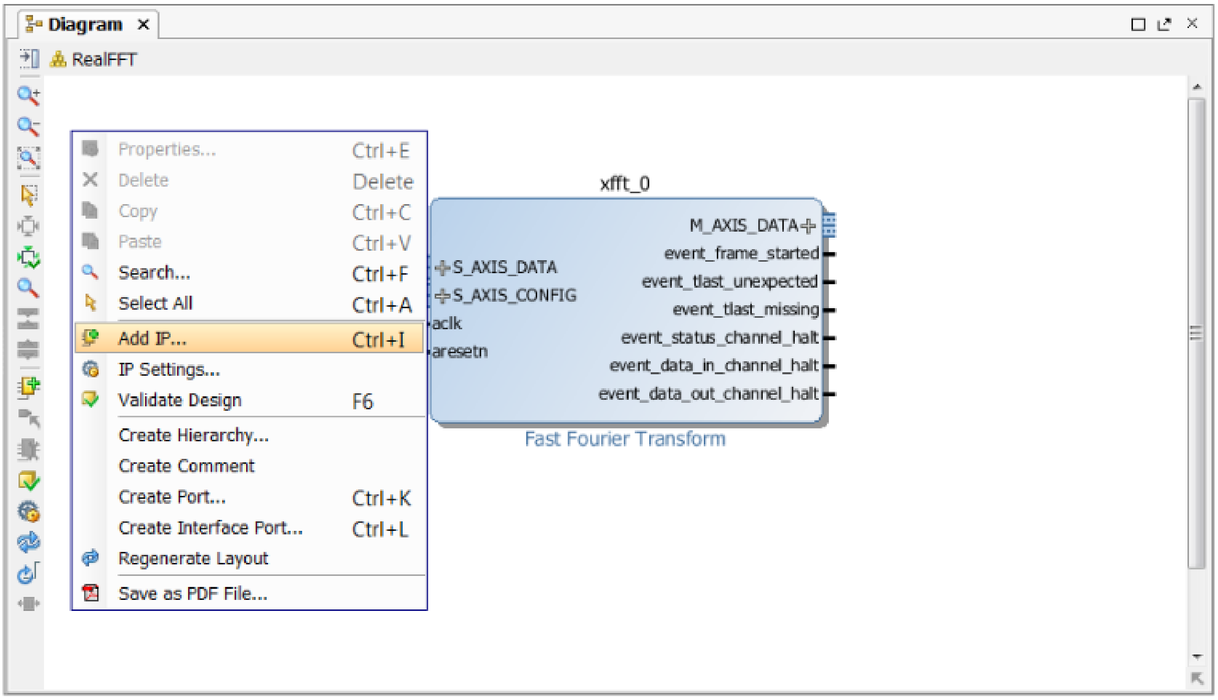


Figure 9-16: **Add IP Blocks**

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 has three IP blocks, as shown in Figure 9-17.

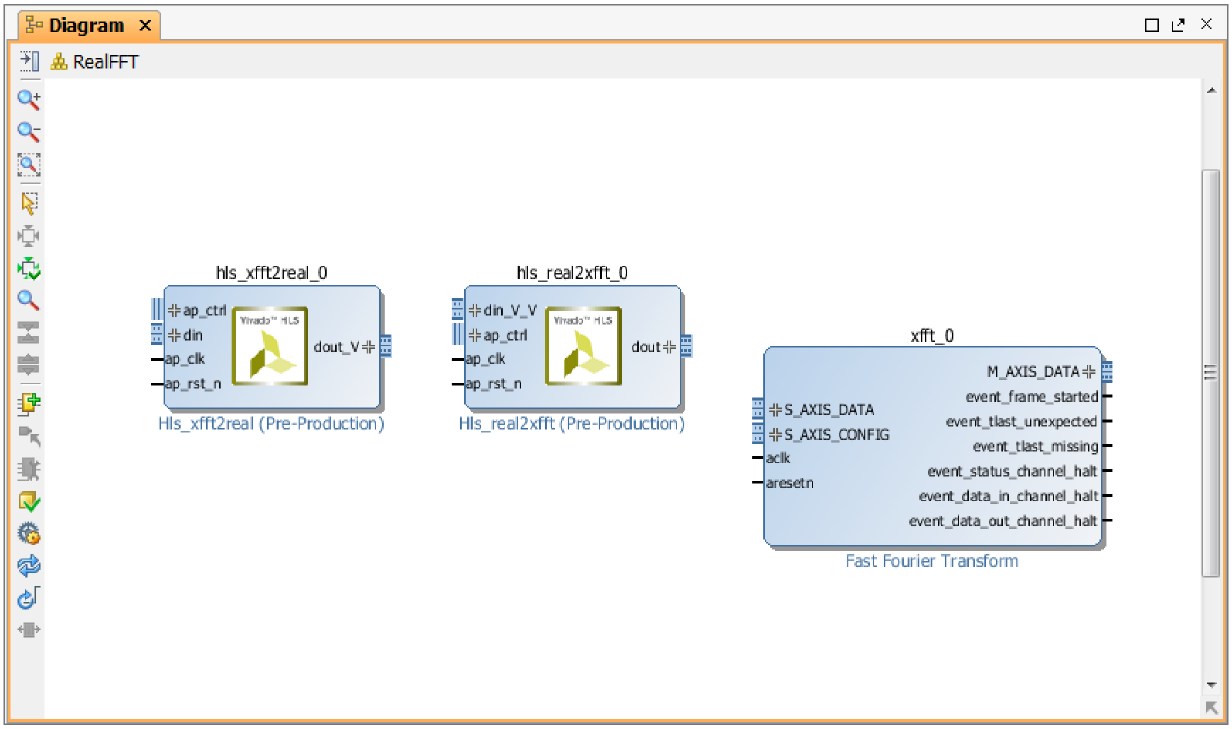


Figure 9-17: **RealFFT IP Blocks**

The next step is to connect HLS blocks to the FFT block and ports.

8. Hover the cursor over the **dout** interface connector of the `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 the **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 **din** interface of the `Hls_xfft2real` block.

The two connections are shown in Figure 9-18.

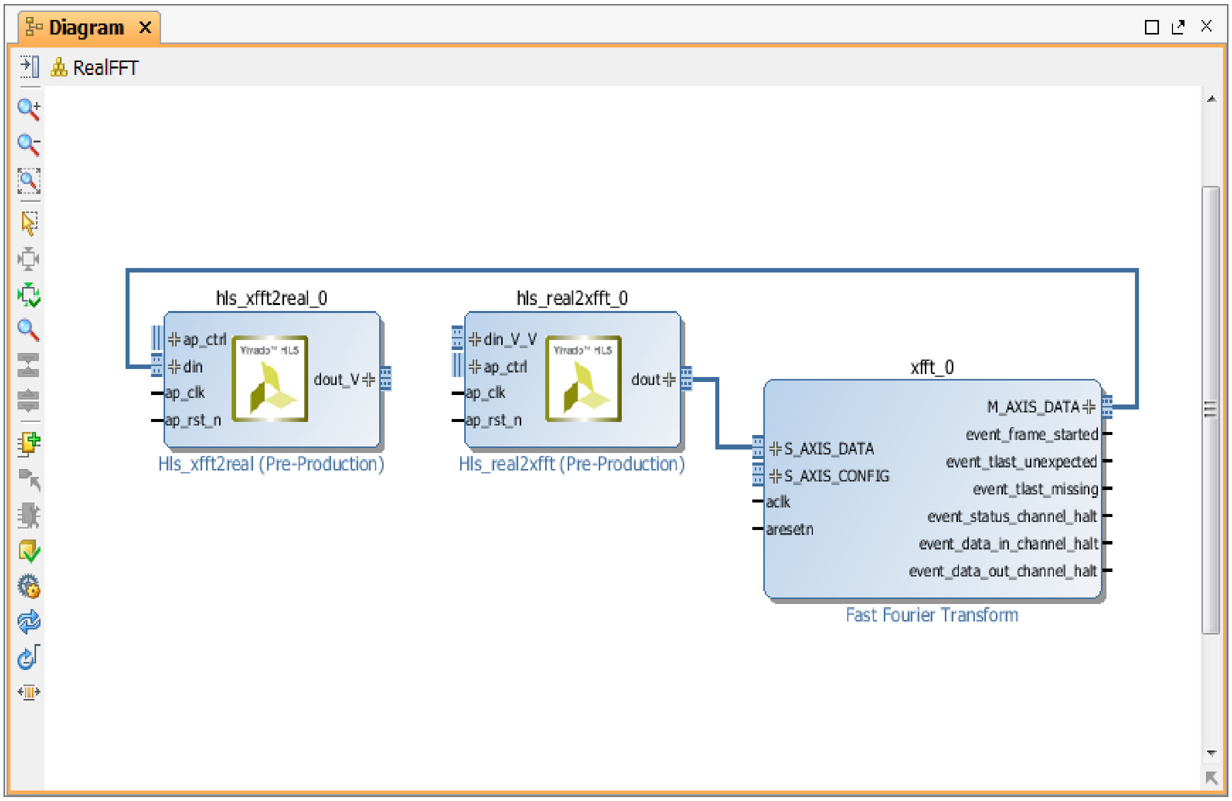


Figure 9-18: **Connecting Ports on the IP Blocks**

To create I/O ports for the design, make some external connections.

10. Right-click the **din_V_V** interface connector on the `hls_real2xfft` block and select **Make External** (Figure 9-19).

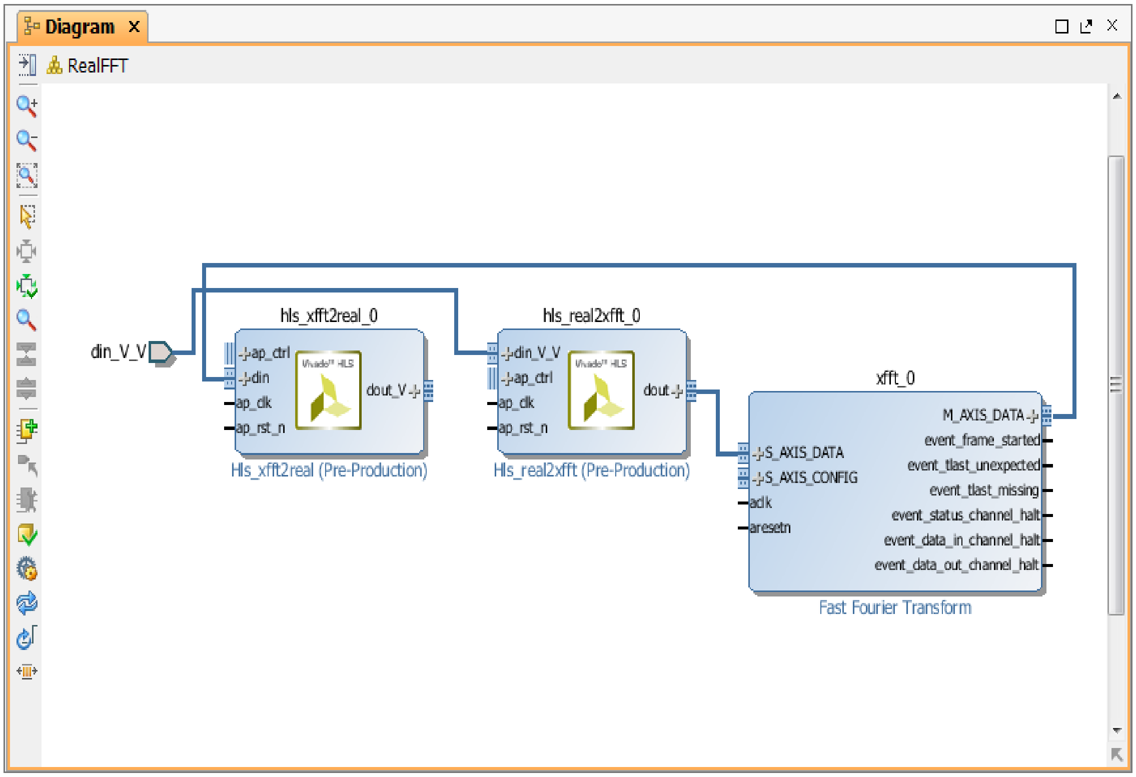


Figure 9-19: **Make External Connections**

11. Give the new interface port a clearly unique name.

    a. Click the port symbol to highlight it.

    b. In the External Interface Properties pane (Figure 9-20), double-click in the Name text entry box to highlight **din_V**.

    c. Type `real2xfft_din` and press **Enter**.

IMPORTANT: *Property changes might not take effect if this re-naming step is not done.*

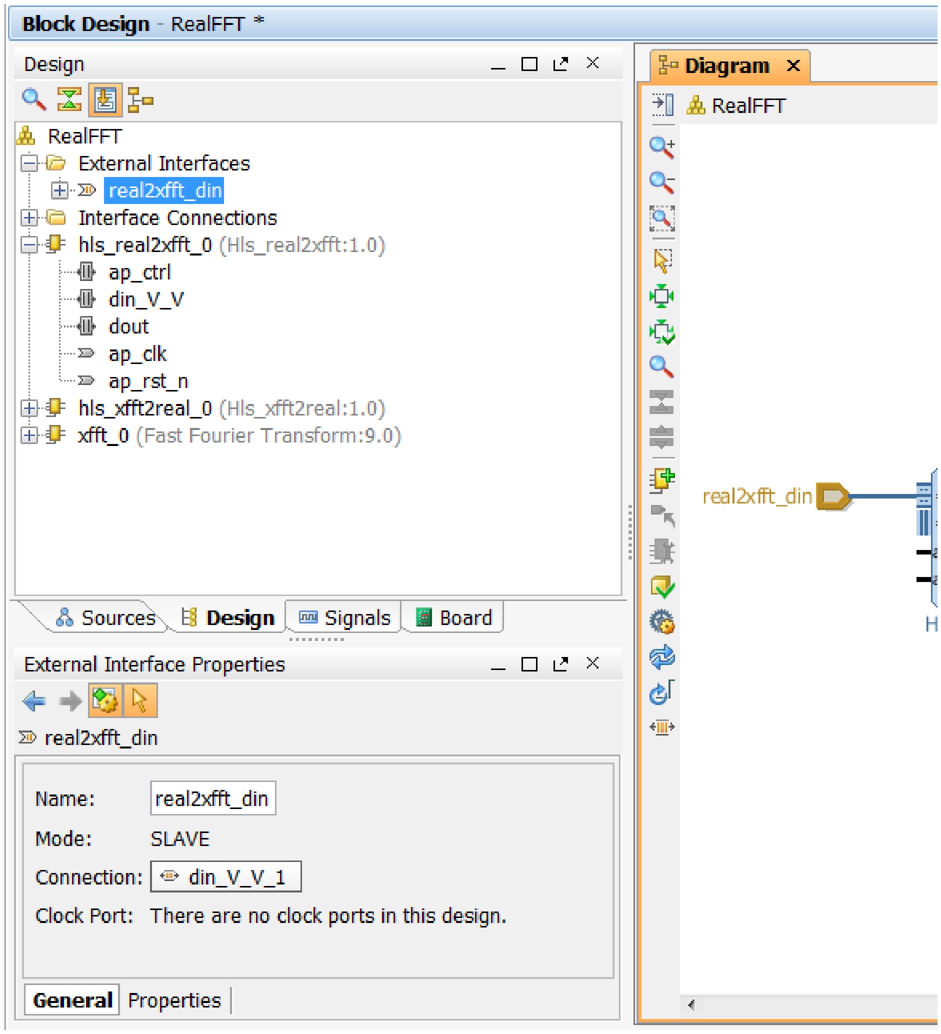


Figure 9-20: **Port Naming**

12. In a similar manner to the previous step:

    a. Make the `dout_V` interface of the `Hls_xfft2real` block external and rename it **xfft2real_dout**.

13. Right-click the `aclk` connector of FFT block and select **Make External**.

14. Right-click the `aresetn` connector of the FFT block and select **Make External**.

15. Tie the `ap_start` ports of both HLS blocks High.

    a. Right-click the canvas and select **Add IP**.

    b. Type `const` into the **Search** text entry box.

    c. Select **Constant IP**.

    d. Double-click the Constant IP symbol (Figure 9-21) and verify that **Const Width** and **Const Val** are set to 1.

e. Click **OK** to close Re-customize IP dialog box.

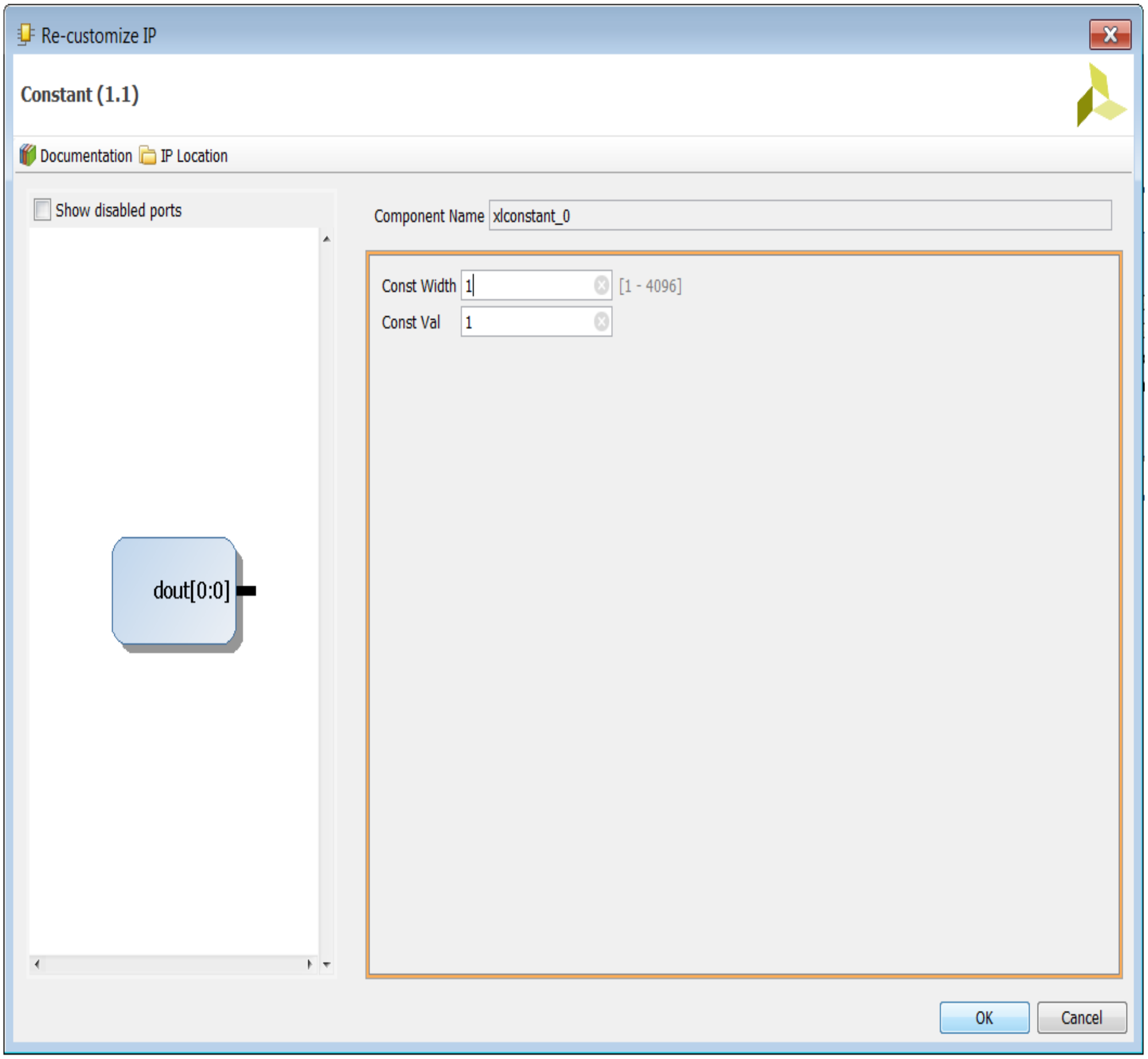


Figure 9-21: **Constant IP Properties**

f. Expand the `ap_ctrl` bus port on both `hls_xfft2real` and `hls_real2xfft` (click the plus symbol associated with each port).

g. Connect `ap_start` in both HLS blocks to the Constant block (Figure 9-22).

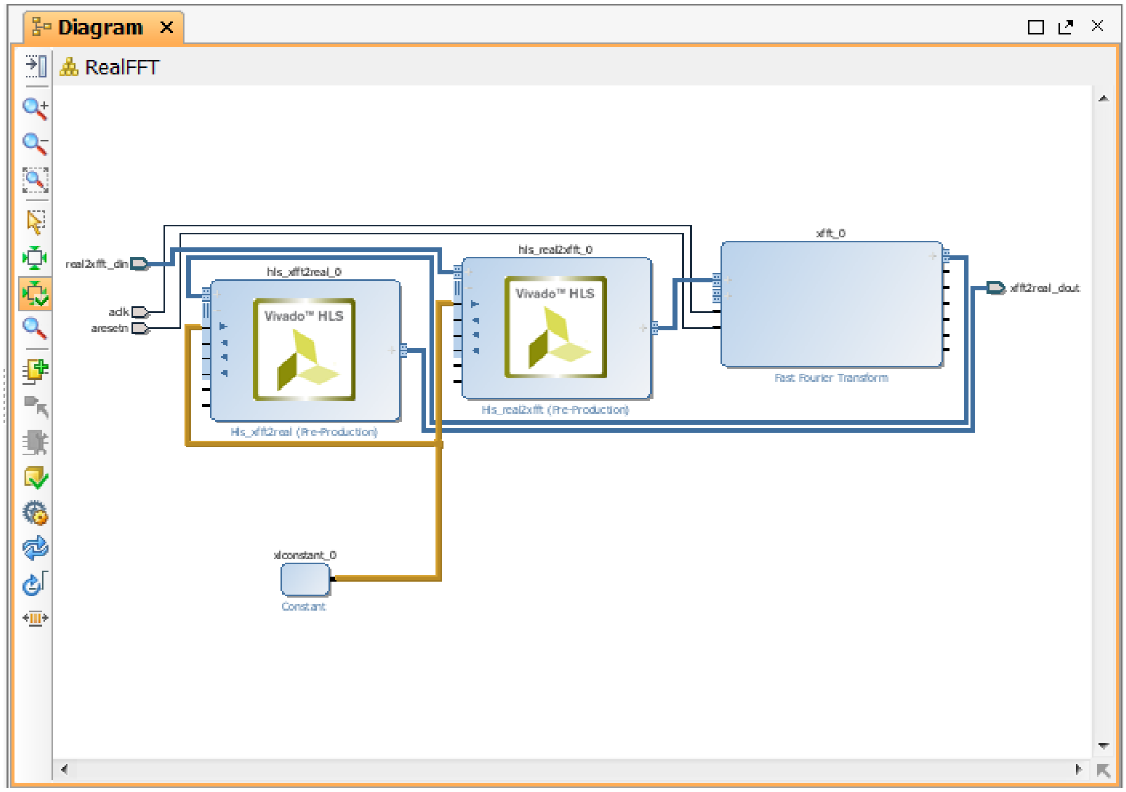


Figure 9-22: **Connect AP_START to Constant**

16. Make the remaining connections.

    a. Click and drag from the `aclk` connector of `hls_real2xfft` and `hls_xfft2real` blocks to the `aclk` external port (or `aclk` connector on `FFT` block or anywhere on "wire" connecting them).

    b. Connect `ap_rst_n` of the `hls_real2xfft` and `hls_xfft2real` blocks to the `aresetn` network.

17. Click the **Regenerate Layout** icon to clean up and reorganize the Block Design.

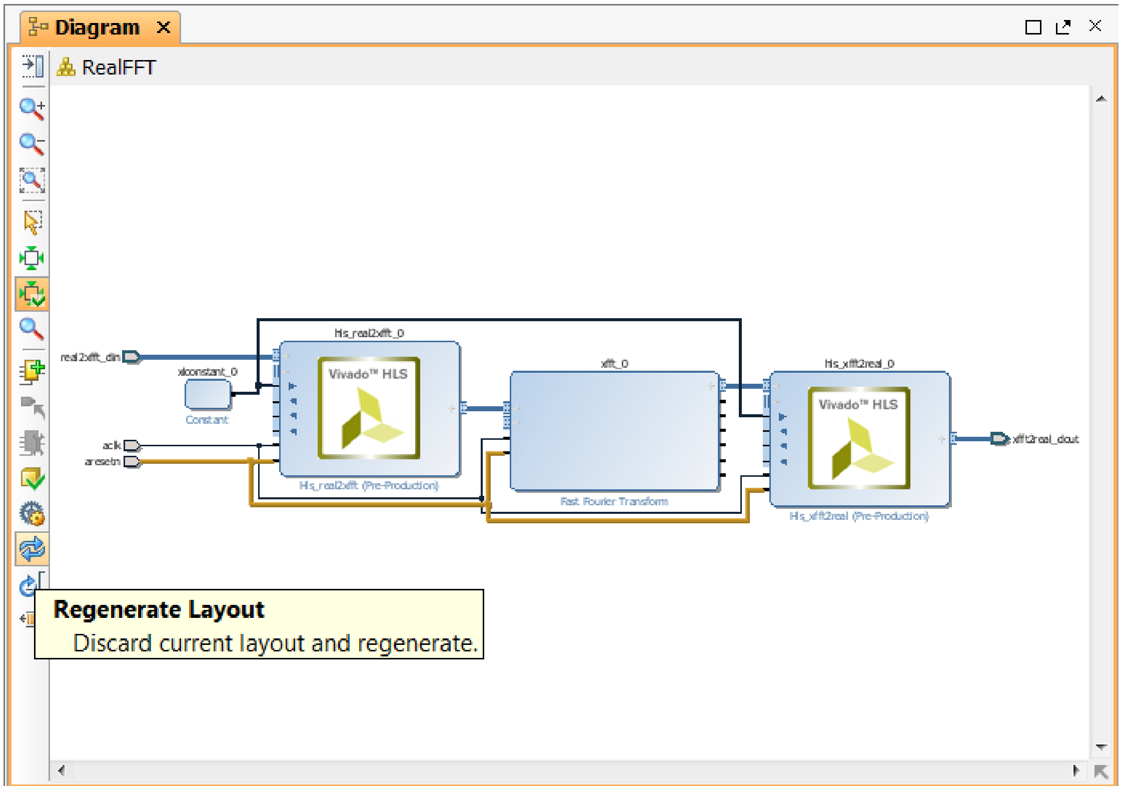


Figure 9-23: **Re-generated Design Diagram**

18. Click the Validate Design button to validate the design is correct.

The vlaidate design will show some warnings. These are related to the s_axis_config pin of the FFT.

   a. The `XFFT` configuration interface is left unconnected because this design always operates in the default mode of the core.

   b. Click **OK** to close the messages..

19. Click **File > Save Block Design**.

20. Close the Block Design.

21. The next step is to generate output products.

   a. In the Sources tab of Project Manager pane (Figure 9-24), right-click **RealFFT.bd** and select Generate Output Products.

   b. Click **Generate** in the resulting dialog to initiate the generation of all output products.

   c. Select **OK** to ignore the warnings discussed above.

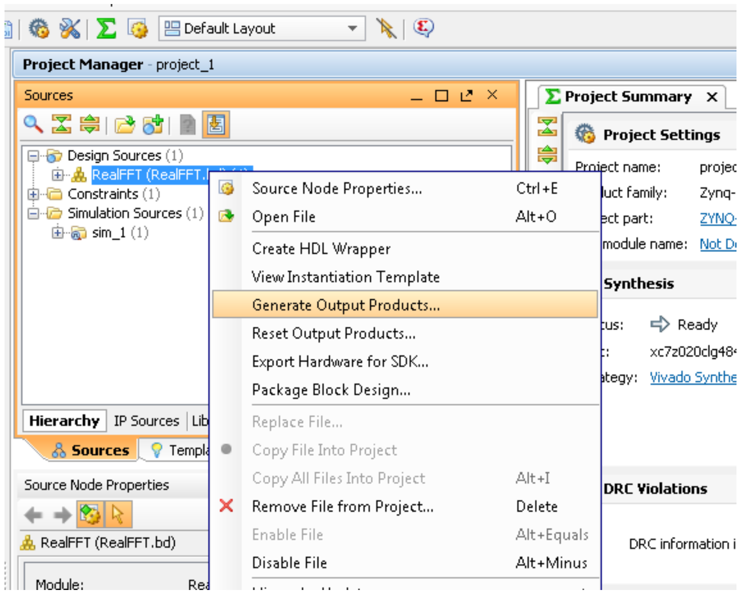


Figure 9-24: **Generating Output Products**

22. 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** and let Vivado manage the wrapper.

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 the Sources tab of the Project Manager pane (Figure 9-25).
2. Select **Add Sources**.

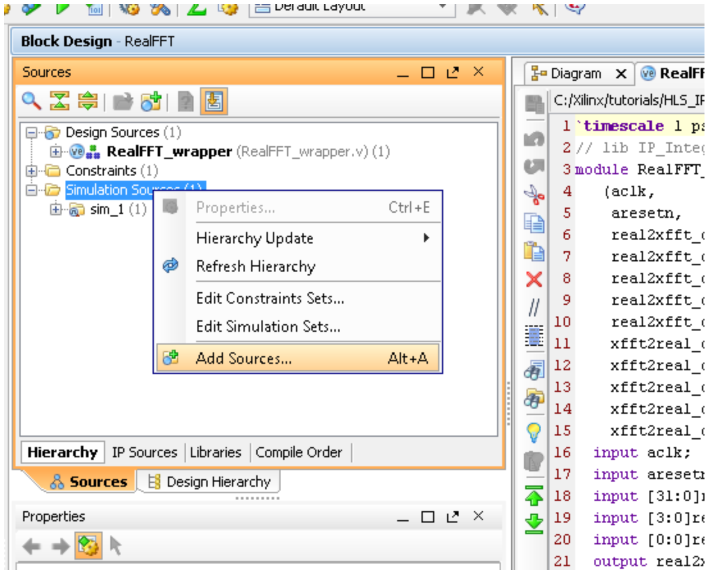


Figure 9-25: **Adding Simulation Sources**

3. Select **Add or Create Simulation Sources** in the Add Sources dialog box.
4. Click **Next**.
5. In the Add Sources dialog box, click the "+" symbol Figure 9-26 and select **Add Files**.

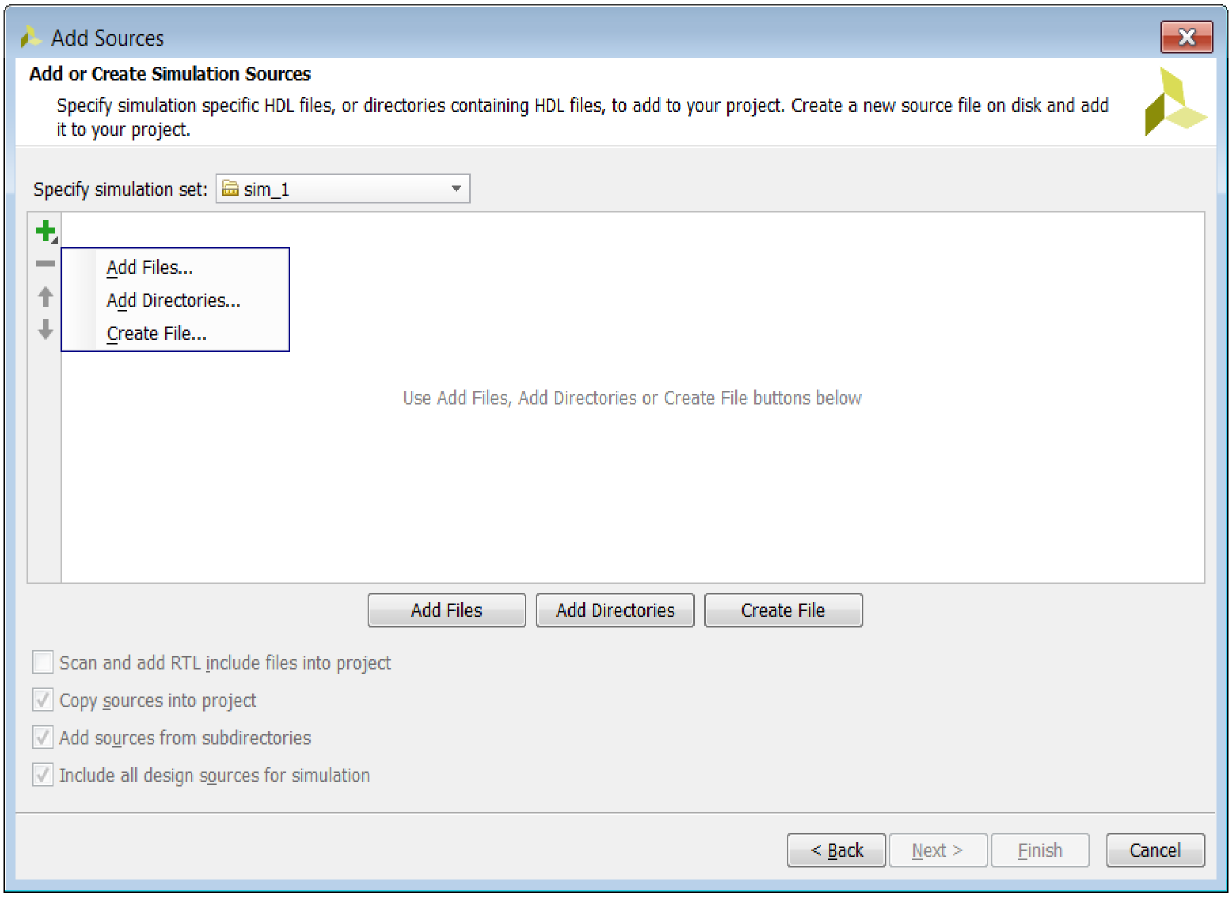


Figure 9-26: **Add Source Dialog Window**

6. Browse to the file `realfft_rtl_tb.v` in the tutorial directory `Using_IP_with_IPI\lab1\verilog_tb`.
7. Select it and click **OK**.
8. Select the checkbox **Copy sources into the project** (Figure 9-27).

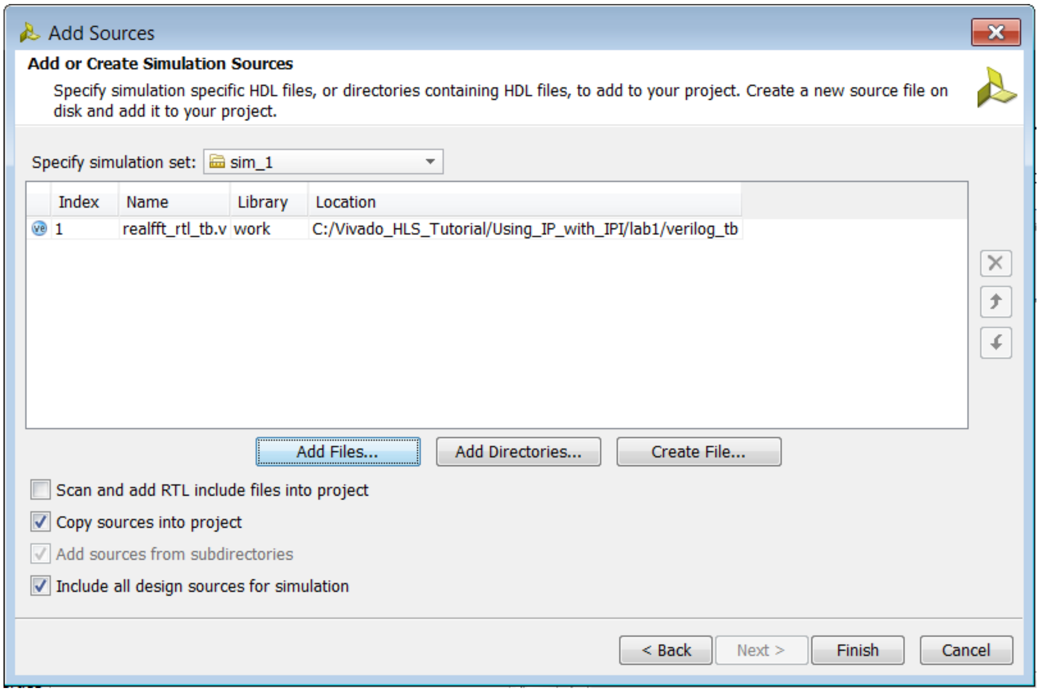


Figure 9-27: **Copy Design Sources**

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 9-28) and select **Run Behavioral Simulation**.

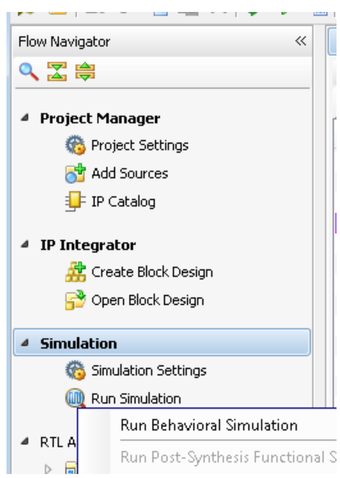


Figure 9-28: **Execute Simulation**

11. Once the simulation has started, click the **Run All** icon to complete simulation.

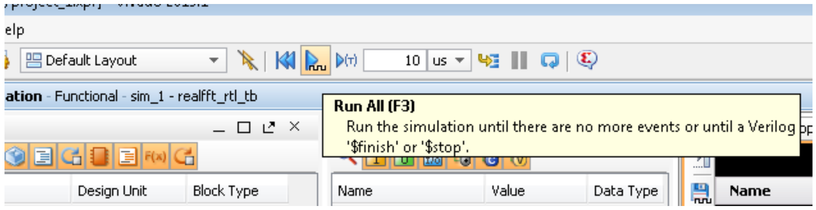


Figure 9-29: **Run the Simulation to Conclusion**

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.

Using HLS IP in a Zynq AP SoC 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:

Lab 1 Description

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.

Lab 2 Description

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. See the information in Locating the Tutorial Design Files.

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

1. Open the Vivado HLS Command Prompt.
   - On Windows use **Start > All Programs > Xilinx Design Tools > Vivado 2015.3 > Vivado HLS > Vivado HLS 2015.3 Command Prompt** (Figure 10-1).
   - On Linux, open a new shell.

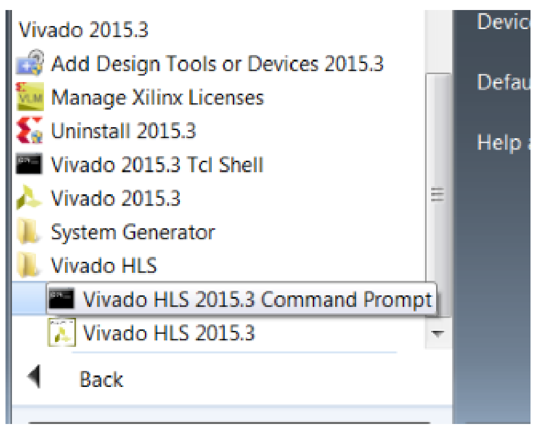


Figure 10-1: **Vivado HLS Command Prompt**

2. Using the command prompt window, change the directory to `Vivado_HLS_Tutorial\Using_IP_with_Zynq\lab1\hls_macc` (Figure 10-2).
3. Type `vivado_hls -f run_hls.tcl` to create the HLS IP (Figure 10-2).

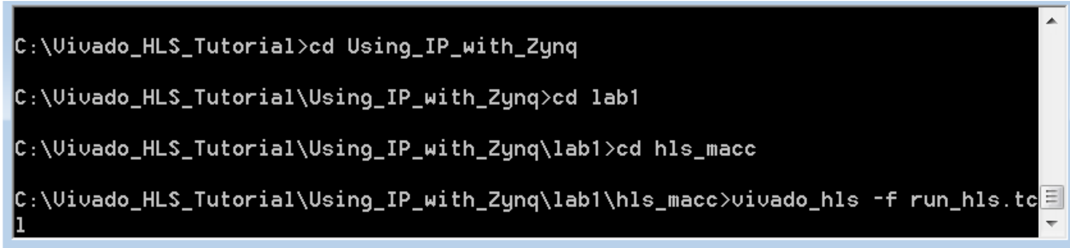


Figure 10-2: **Create the HLS Design**

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):
   - On Windows use **Start > All Programs > Xilinx Design Tools > Vivado 2015.3 > Vivado 2015.3**.
   - On Linux, type `vivado` in the shell.
2. From the Welcome screen, click **Create New Project** (Figure 10-3).

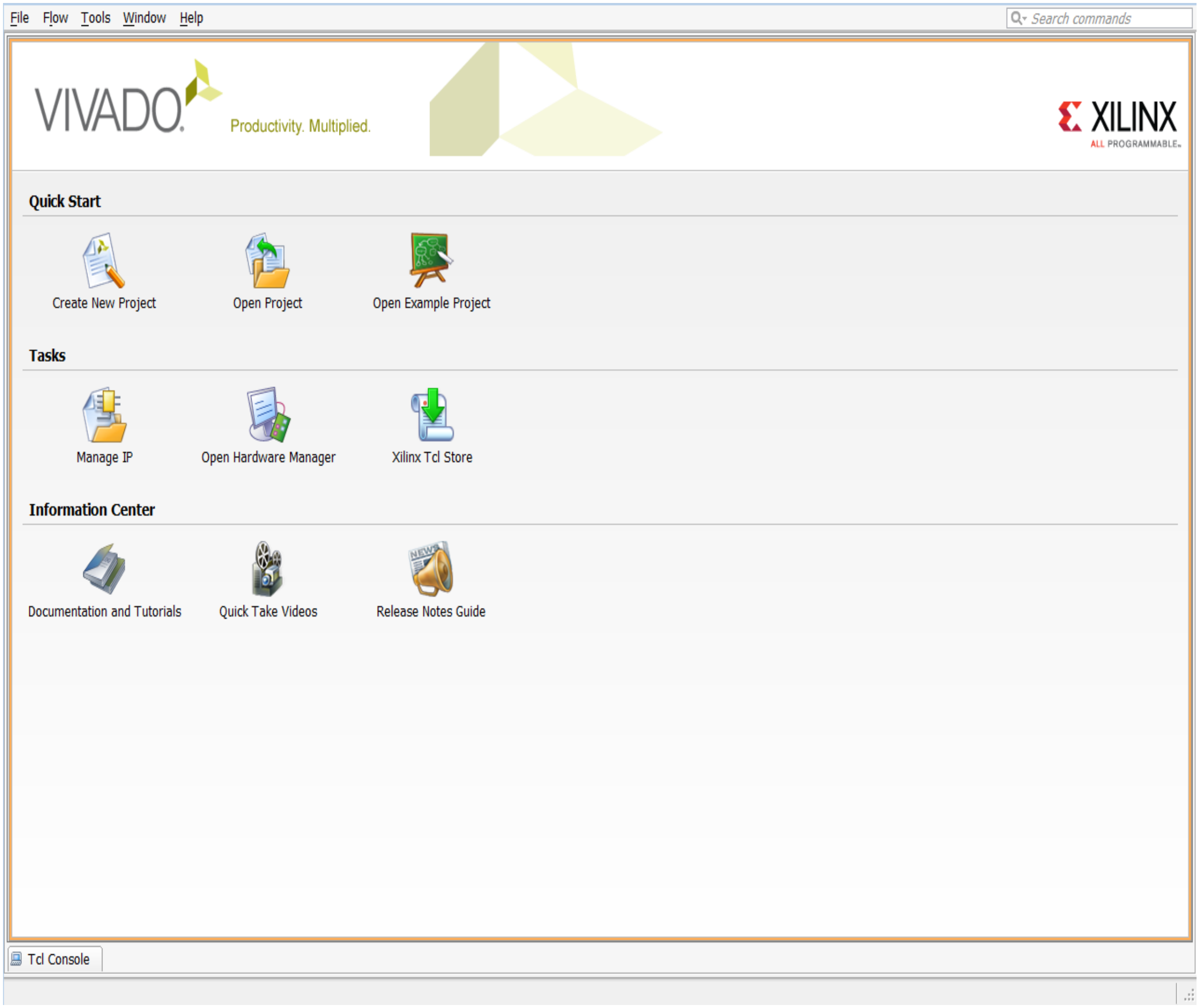


Figure 10-3: **Vivado Welcome Screen**

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 `Using_IP_with_Zynq\lab1` and click **Next** (Figure 10-4).
    c. On the Project Type page, select RTL Project and **Do not specify sources at this time** (if it is not the default).
    d. Click **Next**.

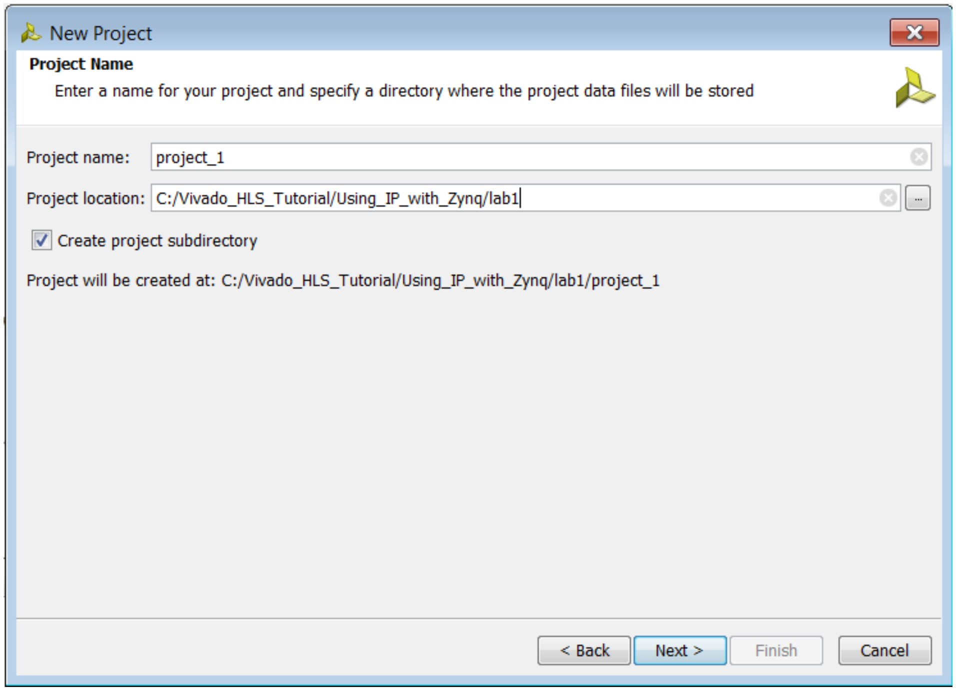


Figure 10-4: **Specify the Vivado Project Directory**

4. On the Default Part page:
   a. Click **Boards**.
   b. Select the **ZYNQ-7 ZC702 Evaluation Board** (Figure 10-5).

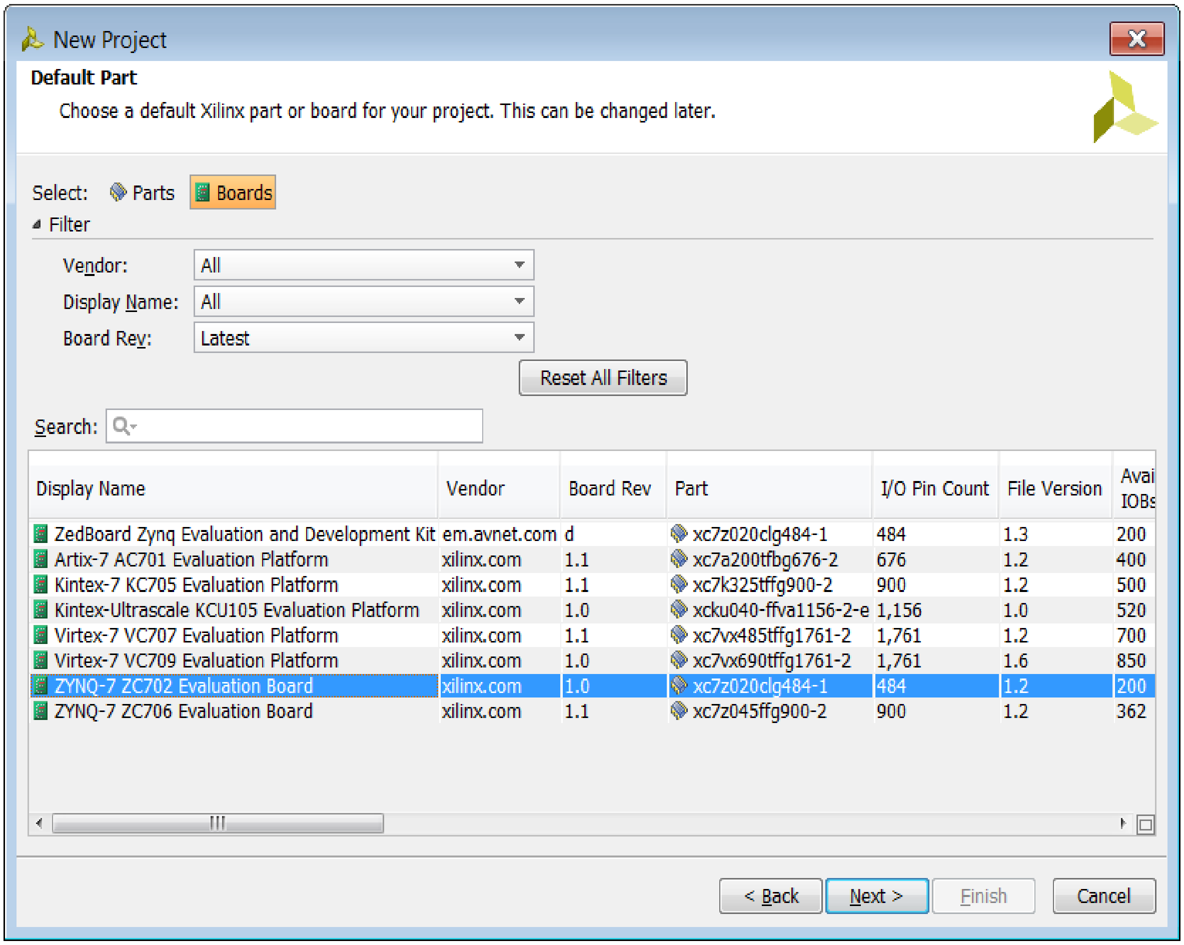


Figure 10-5: **Specify the Vivado Project Details**

a. Click **Next**.

b. Click **Finish** on the New Project Summary Page.

The project workspace opens as shown in Figure 10-6.

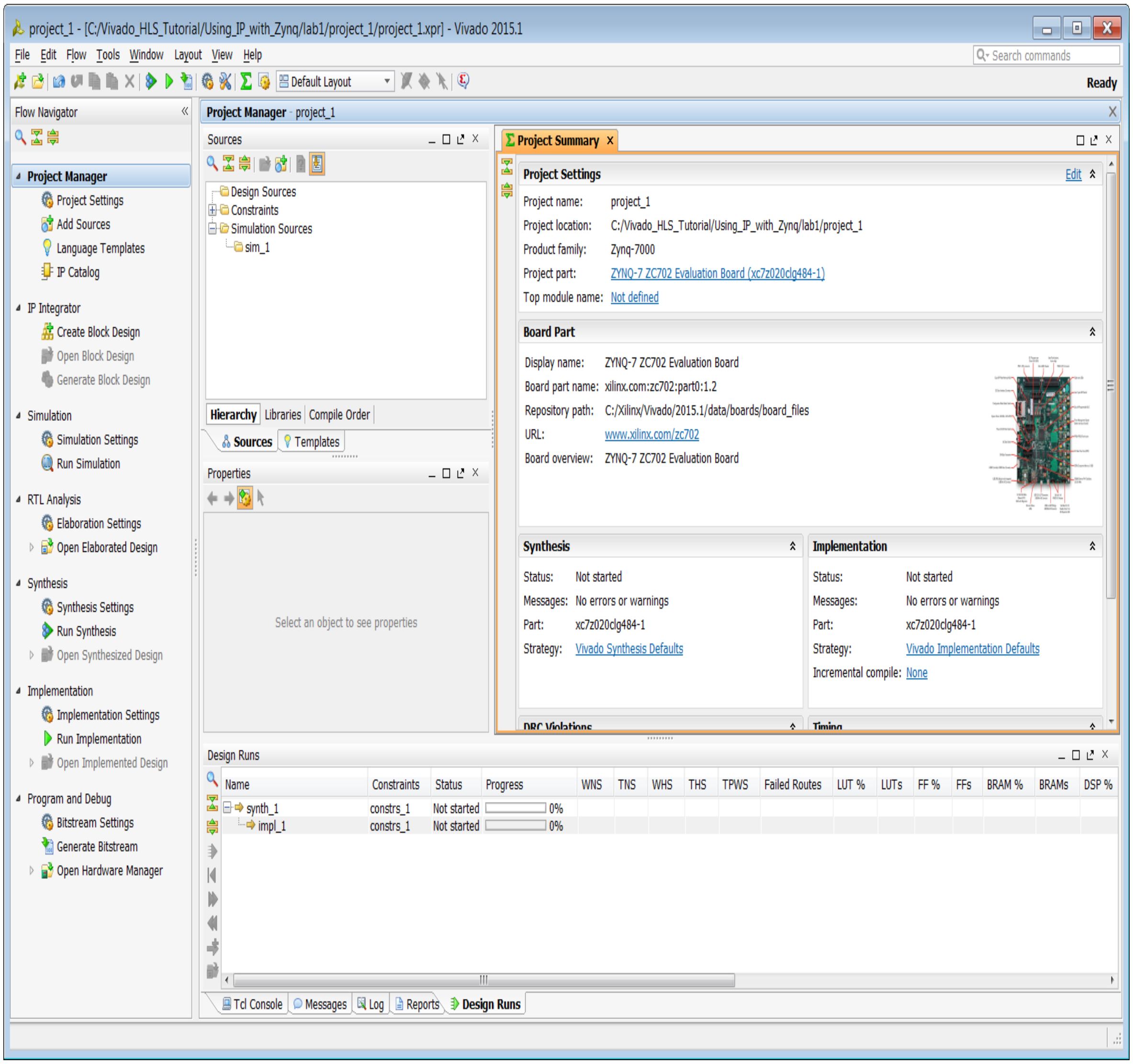


Figure 10-6: **Initial Vivado Zynq Project**

Step 3: Add HLS IP to the IP Catalog

1. In the Project Manager area of the Flow Navigator pane, click **IP Catalog**.

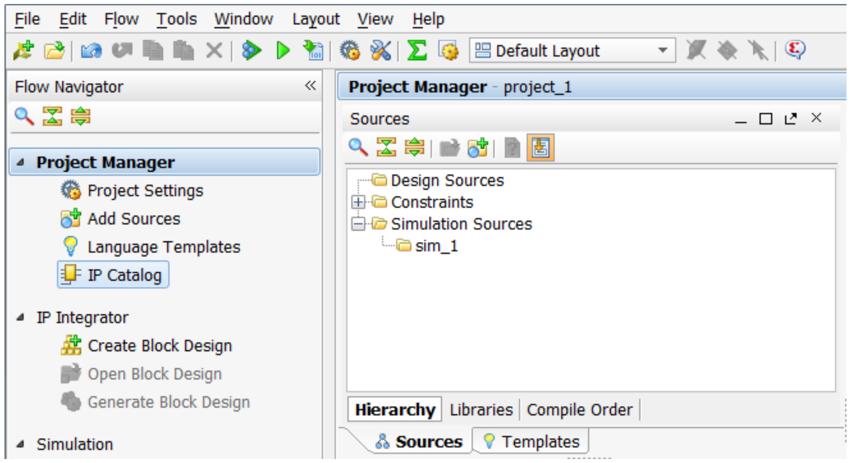


Figure 10-7: **Open the IP Catalog**

The IP Catalog appears in the main pane of the workspace.

2. Click the **IP Settings** icon (Figure 10-8).

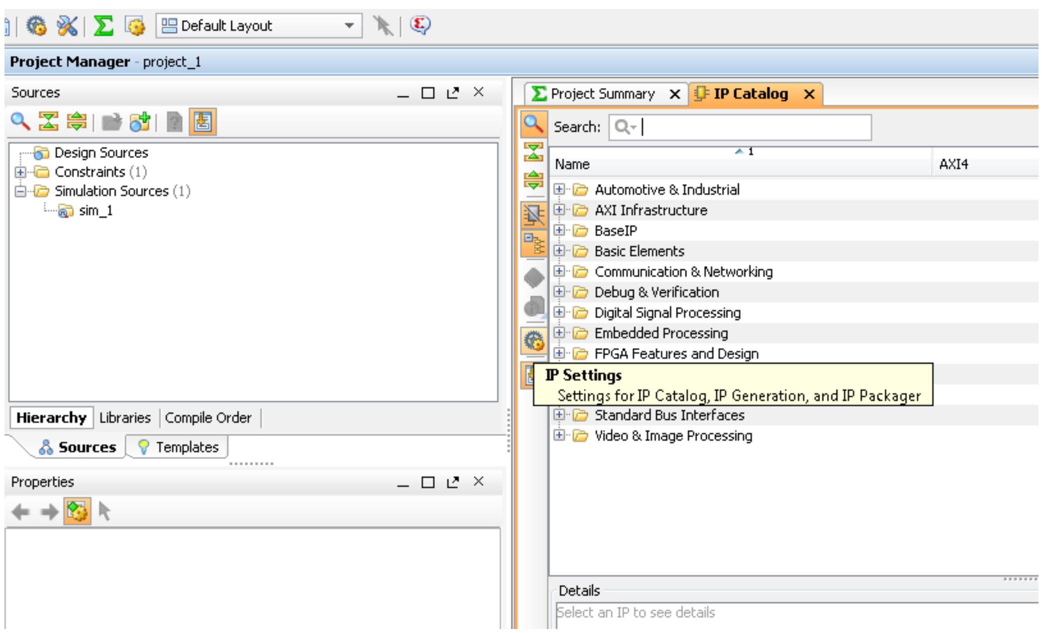


Figure 10-8: **Open the IP Catalog Settings**

3. In the IP section of the Project Settings dialog box, click the "+" symbol to **Add Repository**.
4. In the IP Repositories dialog box:
    a. Browse to the location of the IP created by Vivado HLS, Using_IP_with_Zynq\lab1\hls_macc\vhls_prj\solution1\impl\ip and click **Select**.

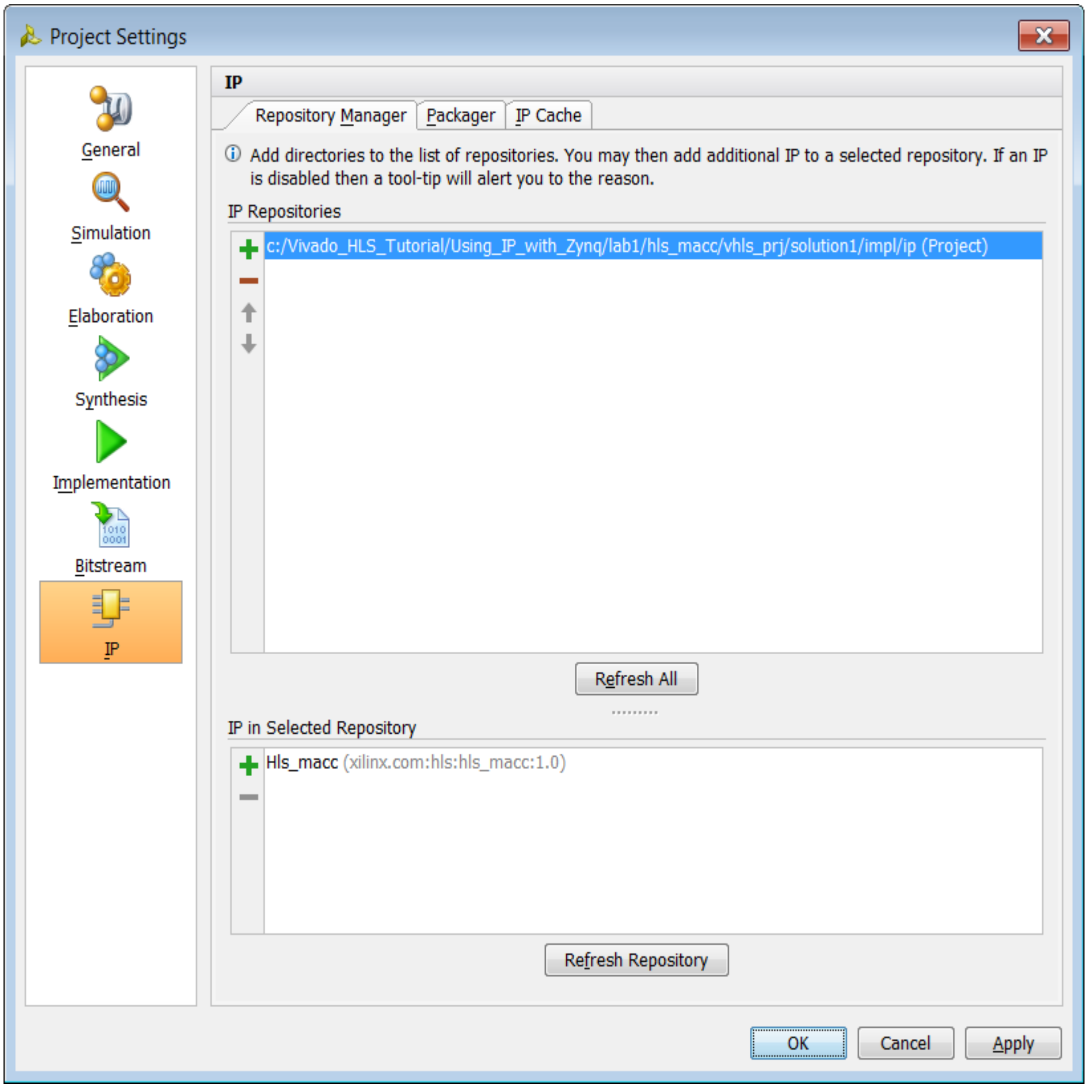


Figure 10-9: **IP Repository**

5. Click **OK** to close the IP repository manager.

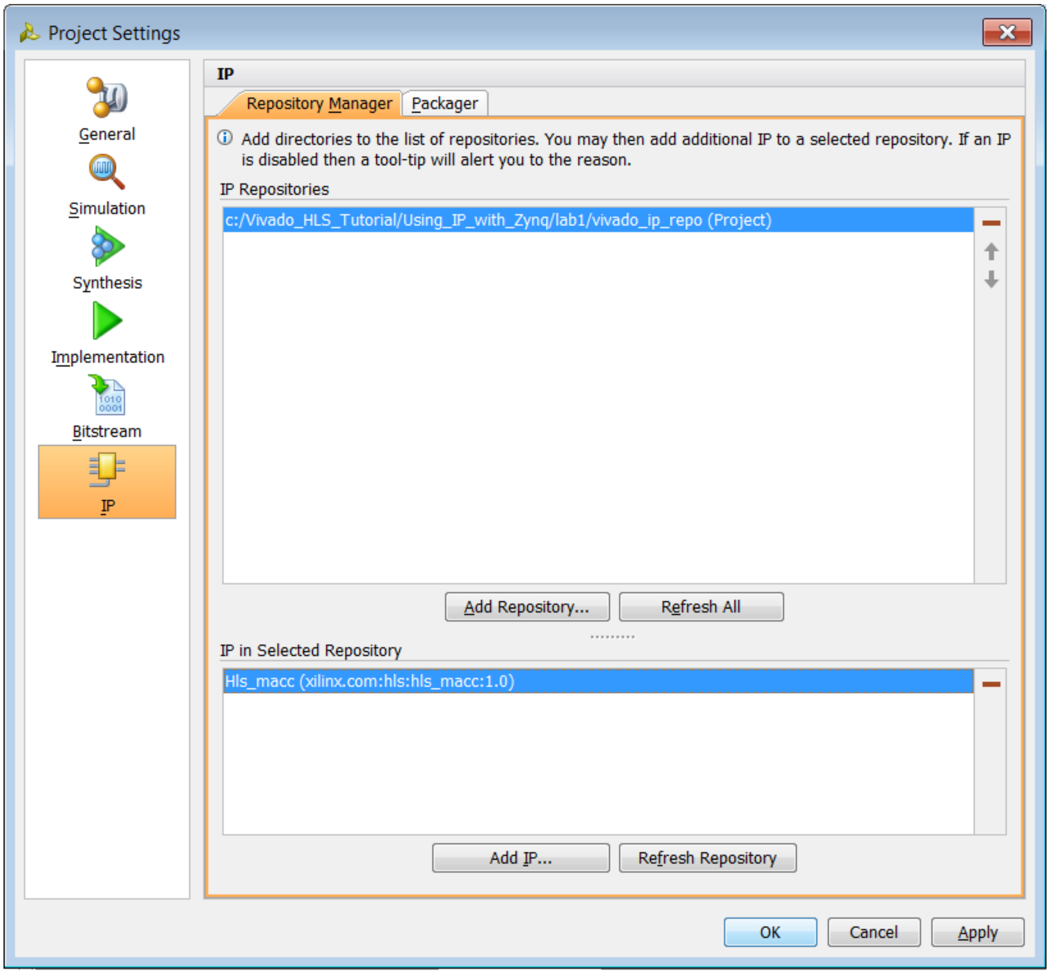


Figure 10-10: **HLS IP in the Repository**

6. There is now an HLS IP in the IP Catalog, `HLS_macc`.

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 type `Zynq_Design` in the dialog box.

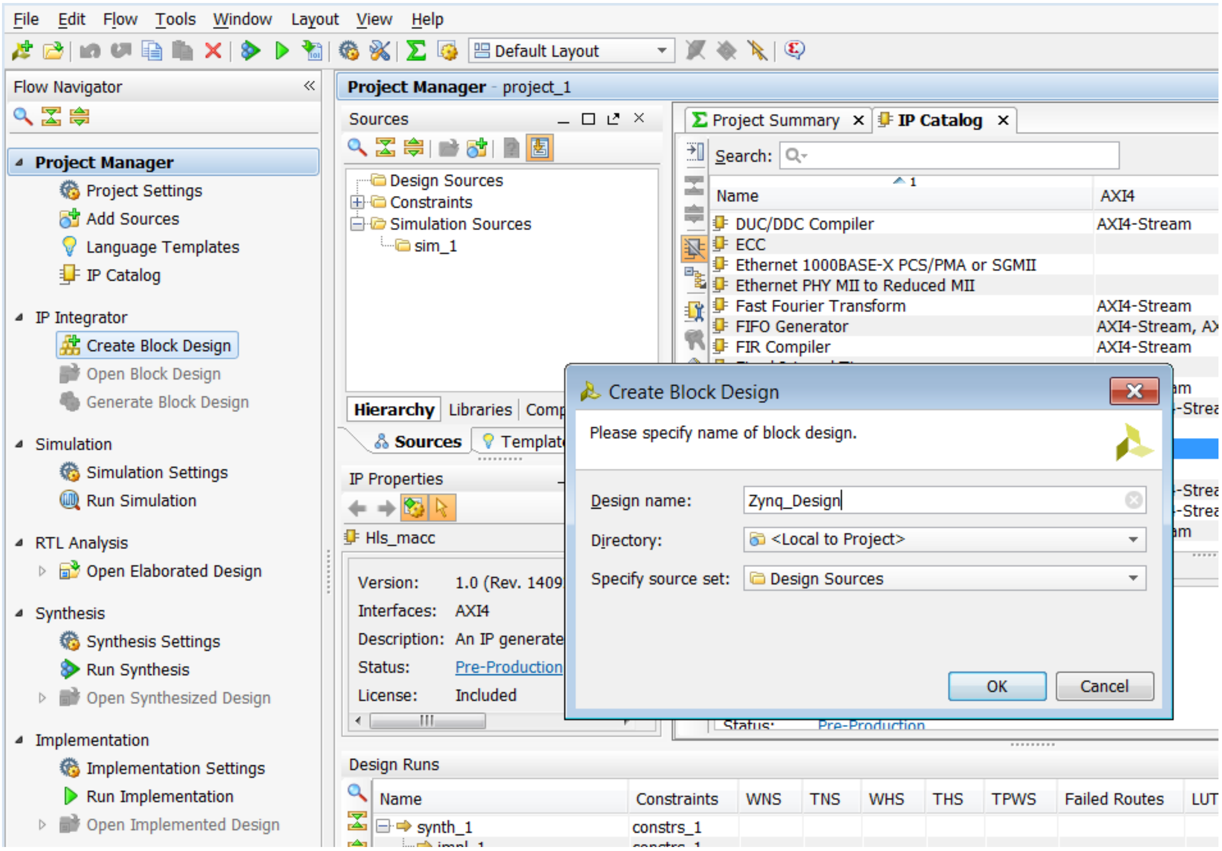


Figure 10-11: **Create the Zynq Design**

The Block Design view opens in the main pane, with a new Diagram tab, containing a blank Block Design canvas.

2. Press the **Add IP** button on the main screen open the IP search dialog.
   a. Type `zynq` into the Search text entry box.
   b. Select **ZYNQ7 Processing System** and press **Enter**.

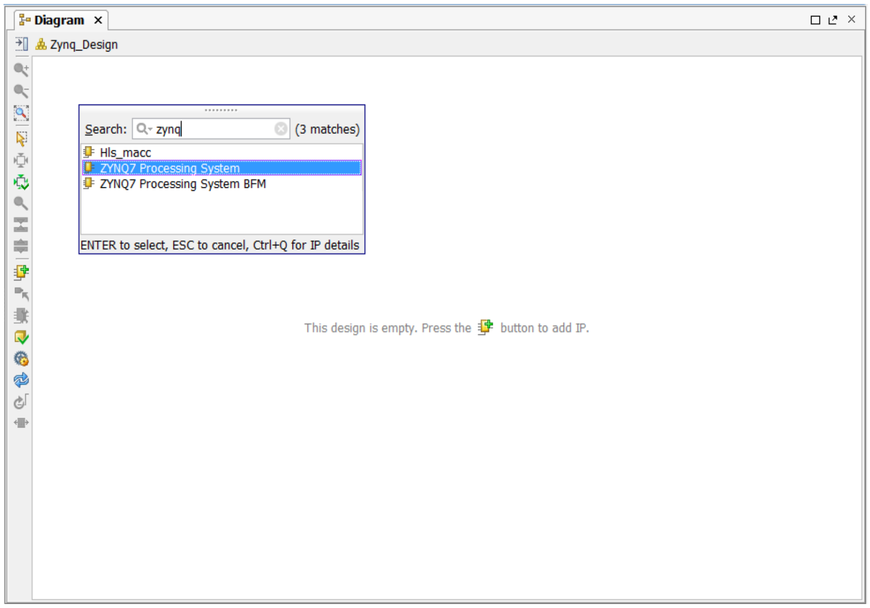


Figure 10-12: **Add a CPU Processor to the Design**

An IP symbol for the ZYNQ7 Processing System appears on the canvas.

3. Double-click the **ZYNQ IP** symbol to open the associated Re-customize IP dialog box.
   a. Click the **Presets** icon and select **ZC702** (Figure 10-13).

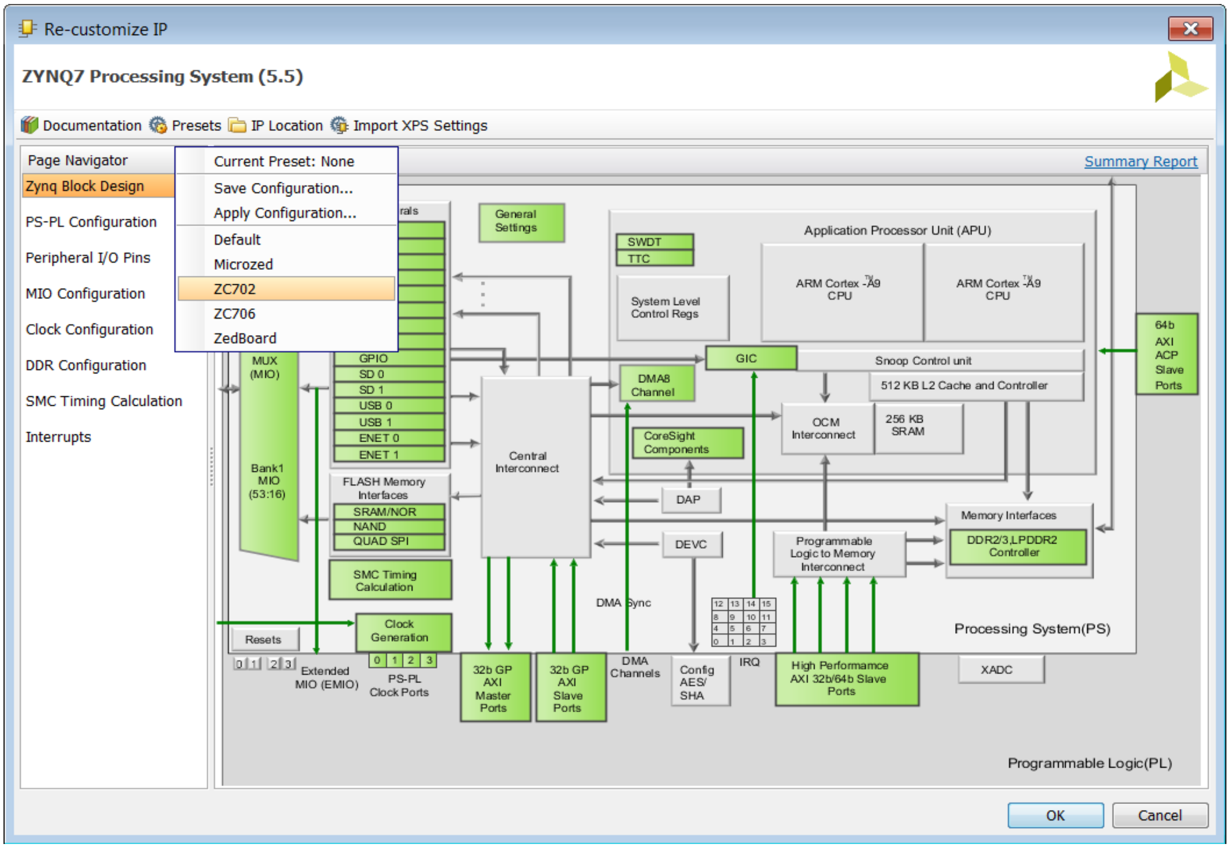


Figure 10-13: **Configure the Zynq AP SoC**

4. Click **MIO Configuration** in the Page Navigator pane.
   a. Expand the **Application Processor Unit** tree view.
   b. Deselect **Timer 0** (or any other timers if they are selected).

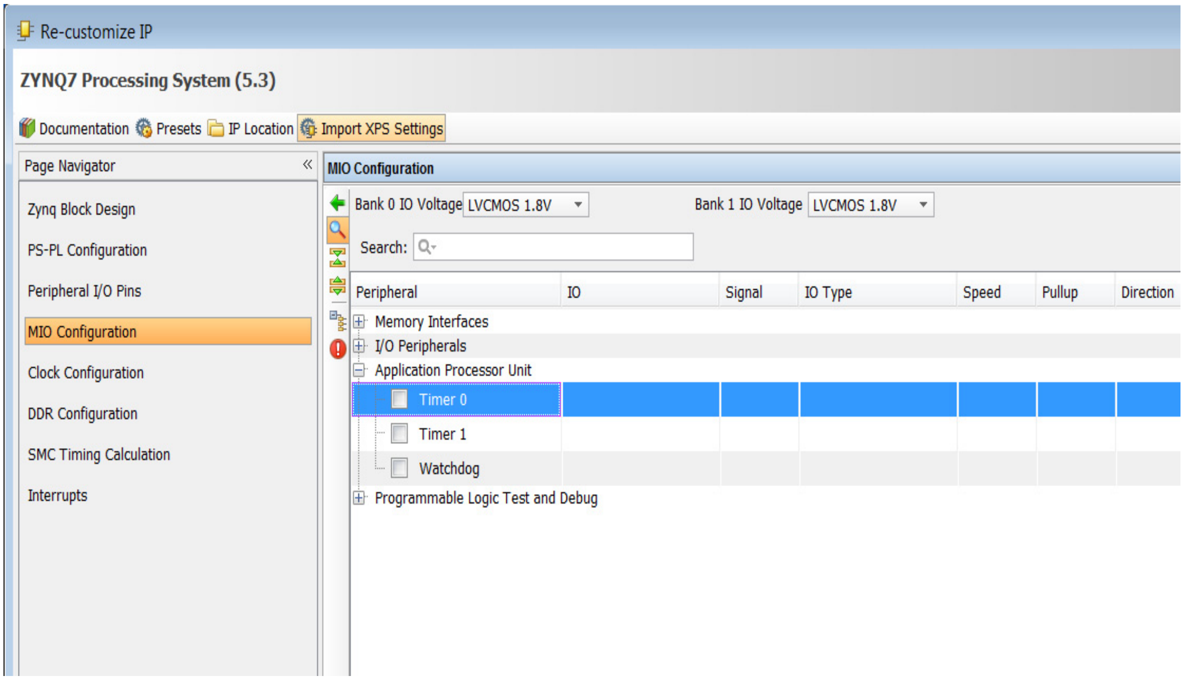


Figure 10-14: **Zynq AP SoC Interrupt Configuration**

5. 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.

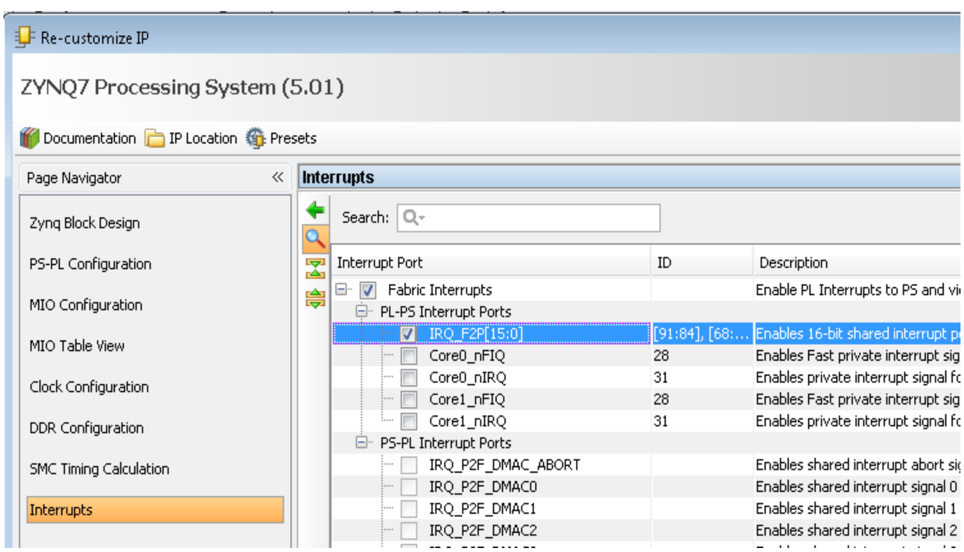


Figure 10-15: **Zynq AP SoC Interrupt Configuration**

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.

6. Click the **Run Block Automation** link under the title bar (Figure 10-16).
   a. Ensure **processing_system7_0** is selected.
   b. Ensure **Apply Board Presets** is deselected. If this remains selected it re-applies the timers that were disabled in step 4 and results in additional ports on the Zynq block in Figure 10-16.
   c. Click **OK** to complete in the resulting dialog box.

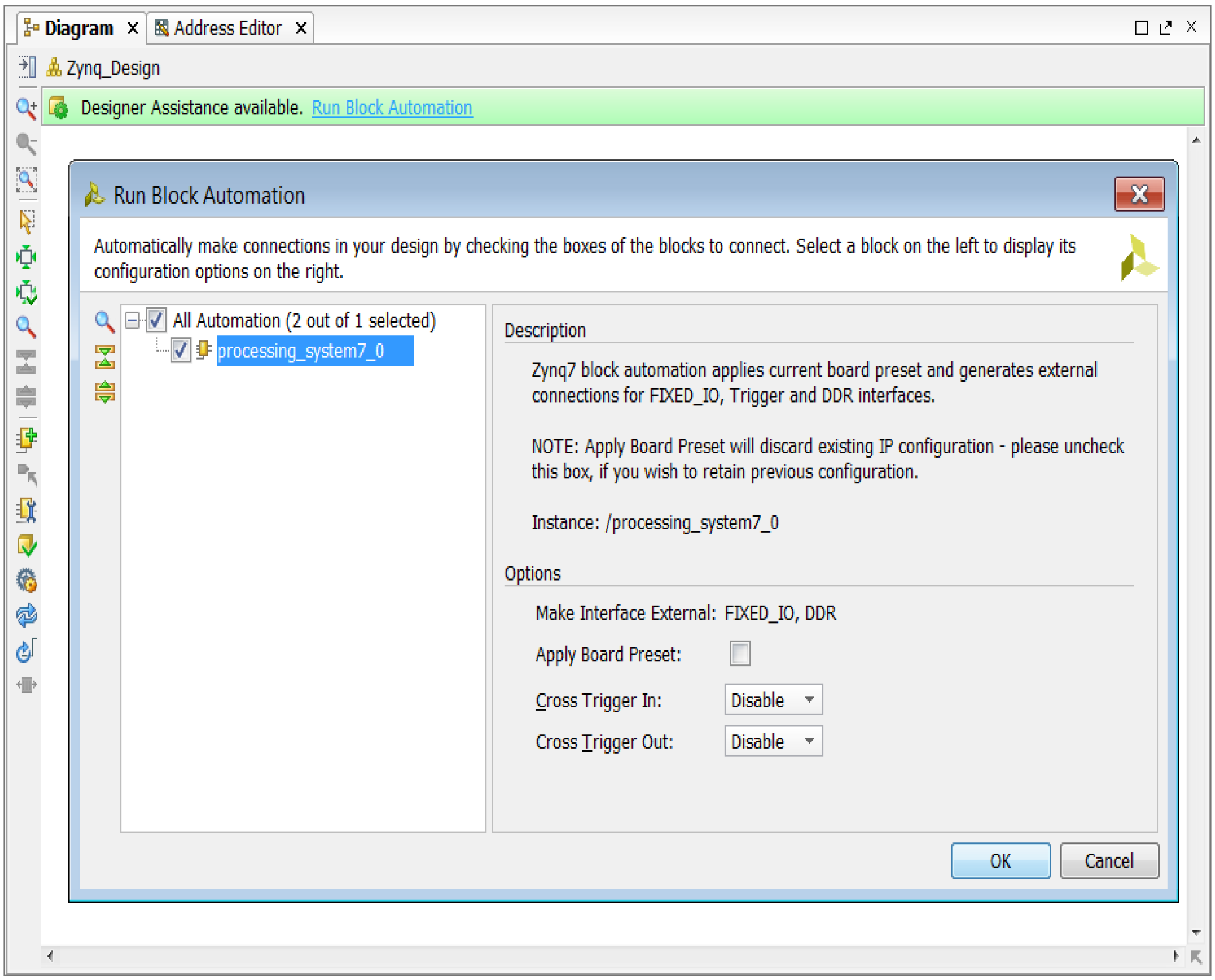


Figure 10-16: **Run Automation**

7. To add HLS IP to the design:
   a. right-click in an open space of canvas and select **Add IP** from the context menu.
   b. Type `hls` in the Search text entry box and press **Enter** to add it to design (Figure 10-17).

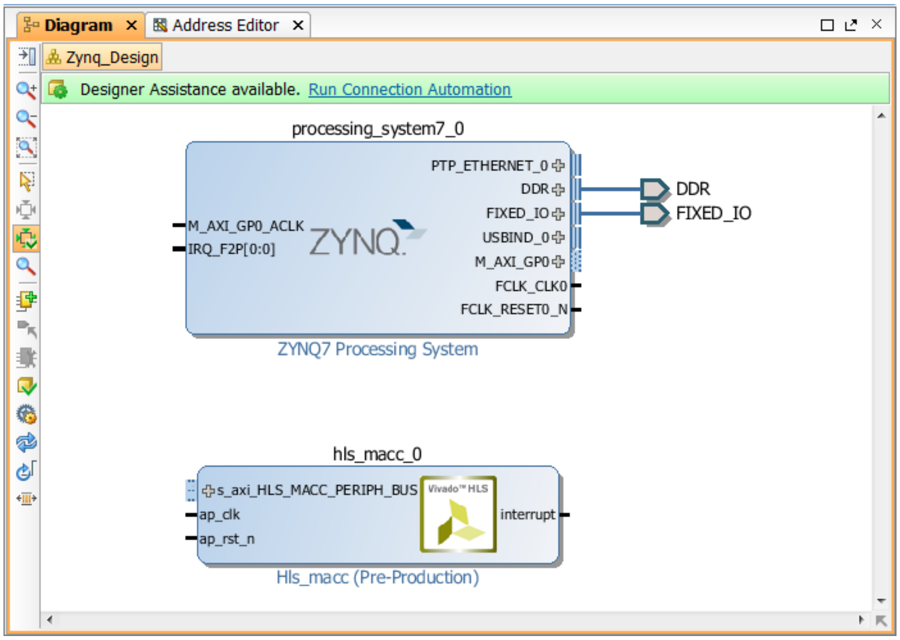


Figure 10-17: **Processor and HLS IP**

Designer assistance is also available to automate the interconnection of IP blocks.

8. Click the **Run Connection Automation** link at the top of the canvas.
9. Select `/hls_macc_0/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 (block instance: `processing_system7_0_axi_periph`), a Proc Sys Reset block and makes all necessary AXI related connections to create the design shown in Figure 10-18.

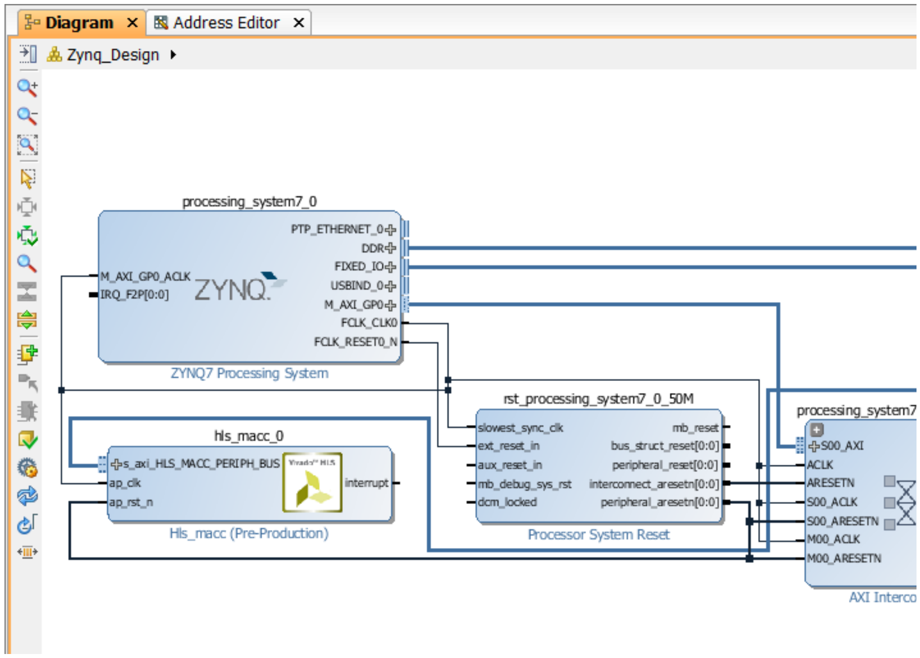


Figure 10-18: **AXI4 Interconnect**

The only remaining connection necessary is from the HLS interrupt port to the `PS7 IRQ_F2P` port.

10. Mouse over the interrupt pin on the `hls_macc_0` IP symbol. 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.
11. Select the **Address Editor** tab and confirm that the `hls_macc_0` peripheral has been assigned a master address range. If it has not, click the **Auto Assign Address** icon.

Figure 10-19: **Address Editor**

The final step in the Block Diagram design entry process is to validate the design.

12. Click the **Validate Design** icon in the toolbar.
13. Upon successful validation, save 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 called `Zynq_Design` is at the top of the Design Sources tree view (Figure 10-20). Right-click this object and select **Generate Output Products**.
3. In the resulting dialog box, click **Generate** to start the process of generating the necessary source files.

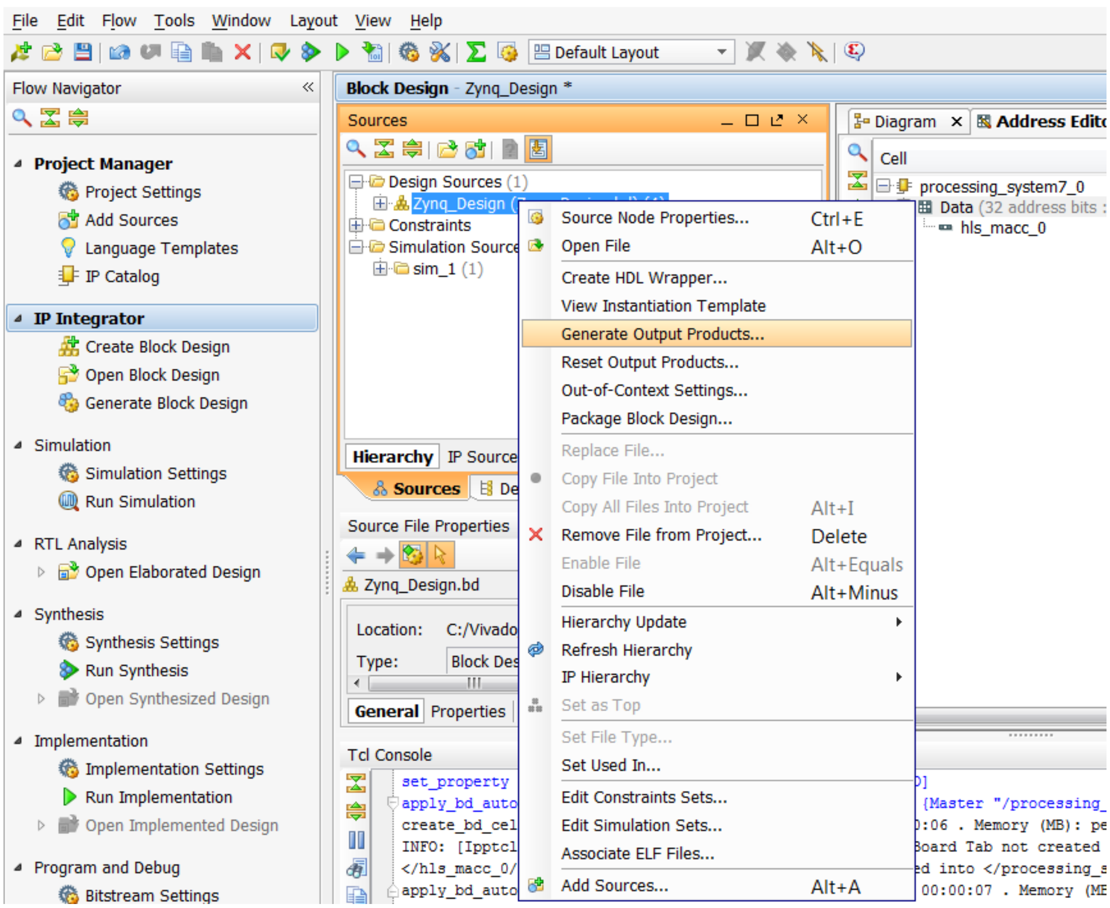


Figure 10-20: **Wrapper Generation**

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.
   a. Click **Yes** to implement the design.
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, you create 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**.

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 10-21), ensure that the **Include Bitstream** is enabled and click **OK**.

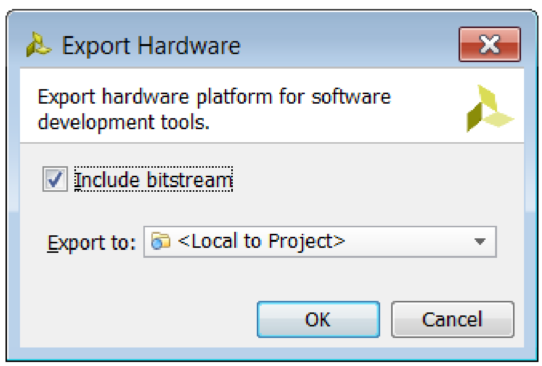


Figure 10-21: **Export Hardware Dialog Box**

3. From the Vivado **File** menu, select **Launch SDK**.
4. Click **OK** to open SDK.
5. From the SDK File menu, select **New > Application Project**.
   a. In the New Project dialog enter the project name `Zynq_Design_Test`.
   b. Click **Next**.
   c. Select the **Hello World** template.
   d. Click **Finish**.

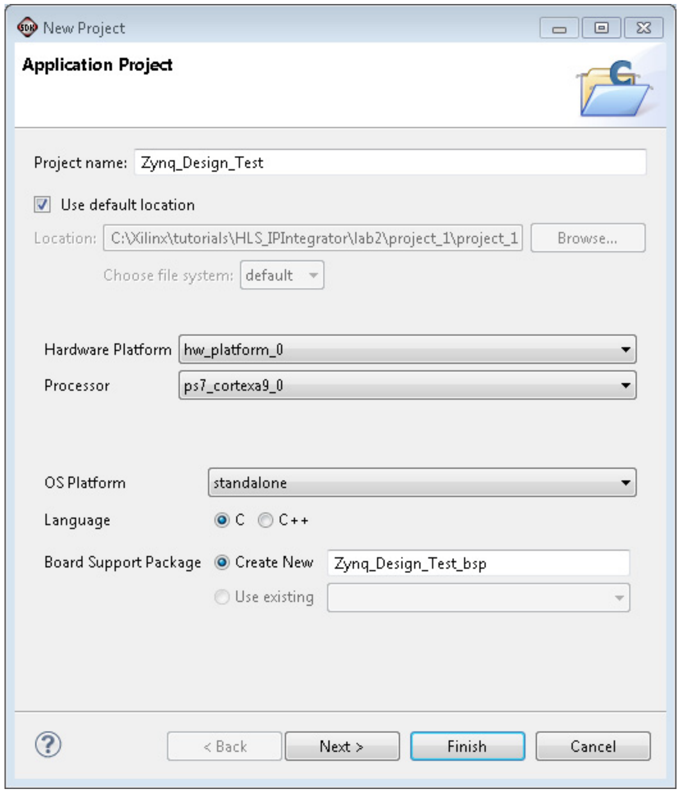


Figure 10-22: **Application Project**

6. Power up the ZC702 board and test the Hello World application. Ensure the board has all the connections to allow you to download the bitstream on the FPGA device. See the documentation that accompanies the ZC702 development board.
7. Click **Xilinx Tools > Program FPGA** (or toolbar icon).

Notice that the Done LED (DS3) is now on.

8. Setup a Terminal in the tab at bottom of workspace:
   a. Click the **Connect** icon (Figure 10-23).

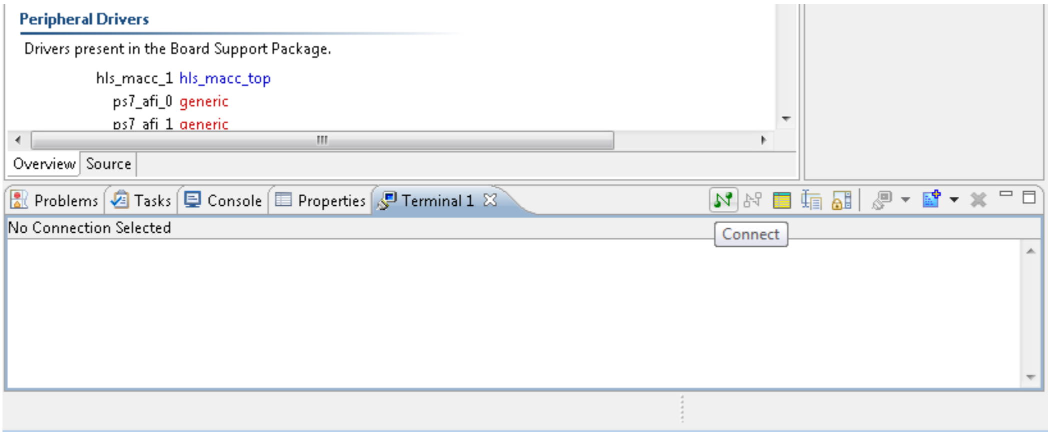


Figure 10-23: **The Connect Icon**

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 10-24).

e. Click **OK** to exit the Terminal Settings dialog box.

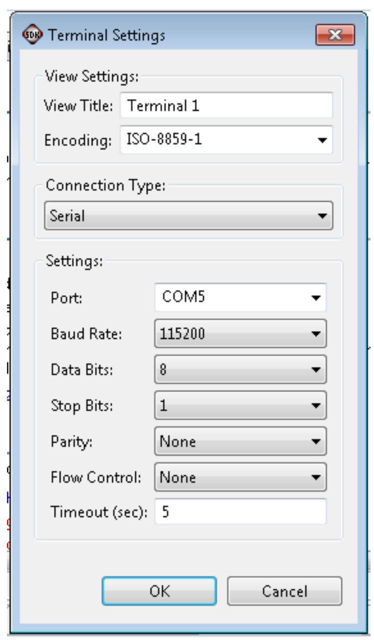


Figure 10-24: **Terminal Settings**

9. Right-click the application project **Zynq_Design_Test** in the Explorer pane (Figure 10-25).

   a. Click **Run As > Launch on Hardware**.

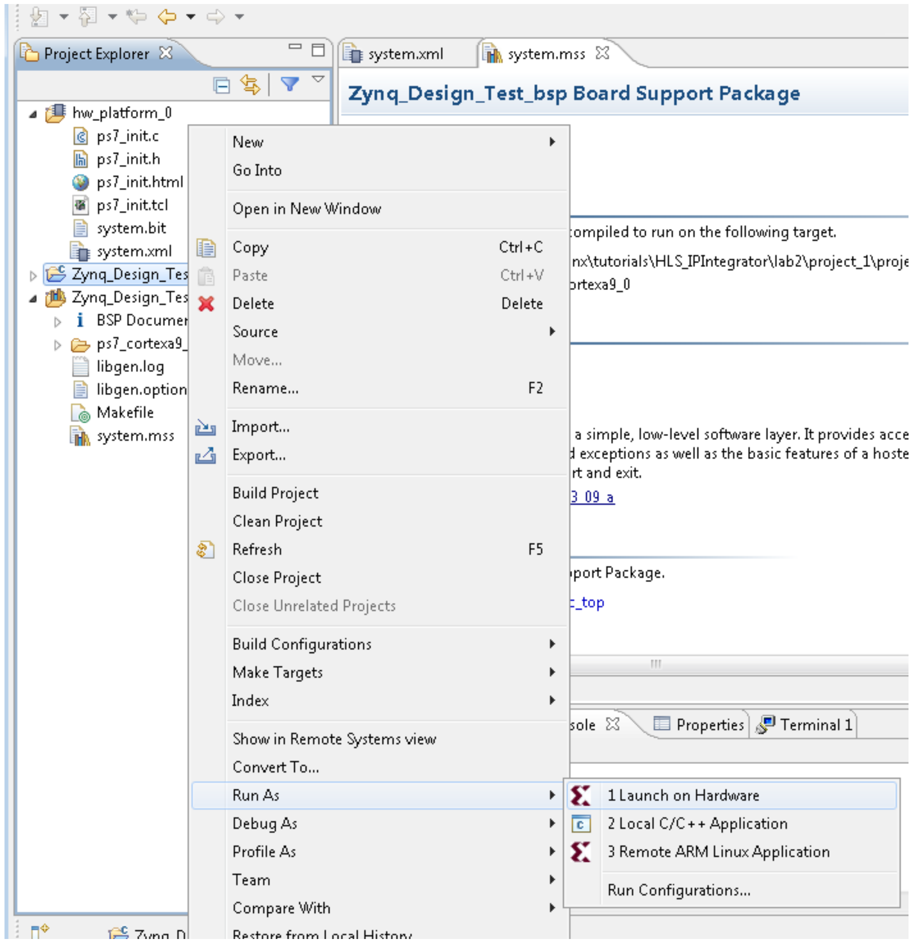


Figure 10-25: **Run the Application Project**

10. Switch to the Terminal tab and confirm that `Hello World` was received.

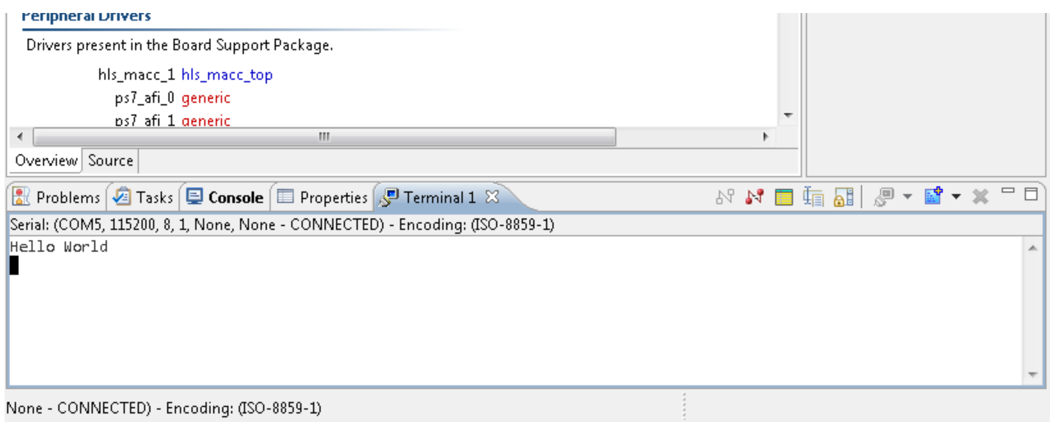


Figure 10-26: **Console Output**

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){
  XHls_macc *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
  //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 the modified source file. When you save the file, 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 10-27.

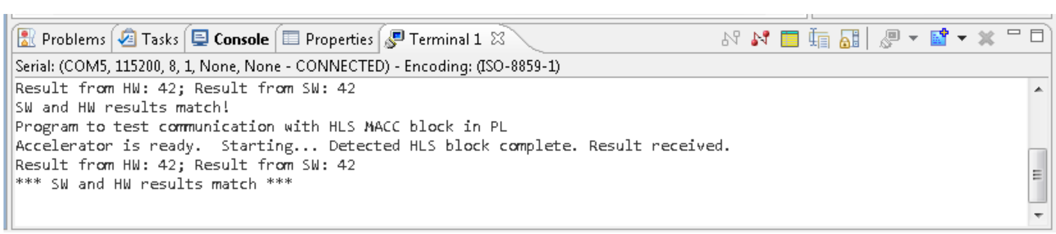


Figure 10-27: **Console Output with Updated C Program**

Lab 2: Streaming Data Between the Zynq 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 HP0 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.

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 10-27.
2. Run Vivado HLS to create two HLS IP blocks by typing `vivado_hls -f run_hls.tcl`.

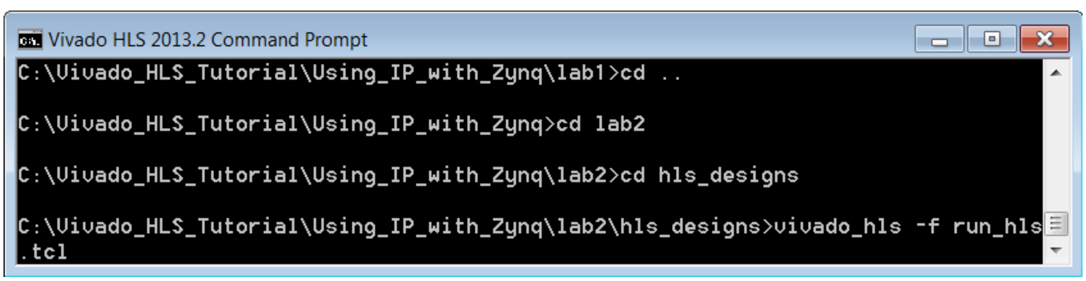


Figure 10-28: **Setup for Zynq Lab 2**

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):

- On Windows use Start > All Programs > Xilinx Design Tools > Vivado 2015.3 > Vivado 2015.3.
- 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. Set the project name to **project_1**, if it is not already specified.
6. 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); just click **Next**.
7. On the Default Part page, under Specify, click **Boards** and select the **ZYNQ-7 ZC702 Evaluation Board**. Click **Next**.
8. 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. Click the **IP Settings** icon.
3. In the IP section of the Project Settings dialog box, click the "+" symbol to **Add Repository**.
4. In the IP Repositories dialog box:
   a. Browse to the `lab2` tutorial directory .
   b. Click the **Create New Folder** icon.
   c. Enter `vivado_ip_repo` in the resulting dialog box.
   d. Click **OK**.
   e. Click **Select** to close the IP Repository window.
5. On returning to the IP Setting dialog box:
   a. Click the "+" symbol to **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.
   d. Click **OK**.

6. Follow the same procedure to add the second 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.
8. Click **OK** to exit the dialog box.
9. There is now HLS IP in the IP Catalog (Hls_real2xfft and Hls_xfft2real).

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** button to add IP
   a. In the Search box, type `fourier`.
   b. Select the Fast Fourier Transform and double-click with the mouse.
3. Double-click the new **Fast Fourier Transform IP** symbol to open the Re-customize IP dialog box. On the Configuration tab:
   a. Change the **Transform Length** to 512.
   b. Change the **Target Clock Frequency** to 100 MHz.
   c. 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 Orde**r 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**.
6. Connect the HLS blocks to the FFT block.
   a. Mouse over the `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.

7. In a similar fashion:

   a. Connect the FFT's `M_AXIS_DATA` interface to the `din` input interface of the `hls_xfft2real` block.

8. Put the data processing blocks into their own level of hierarchy.

   a. Select everything in the current digram by pressing **Ctrl+A**.

   b. Right-click the canvas and select **Create Hierarchy** from the context menu.RealFFT

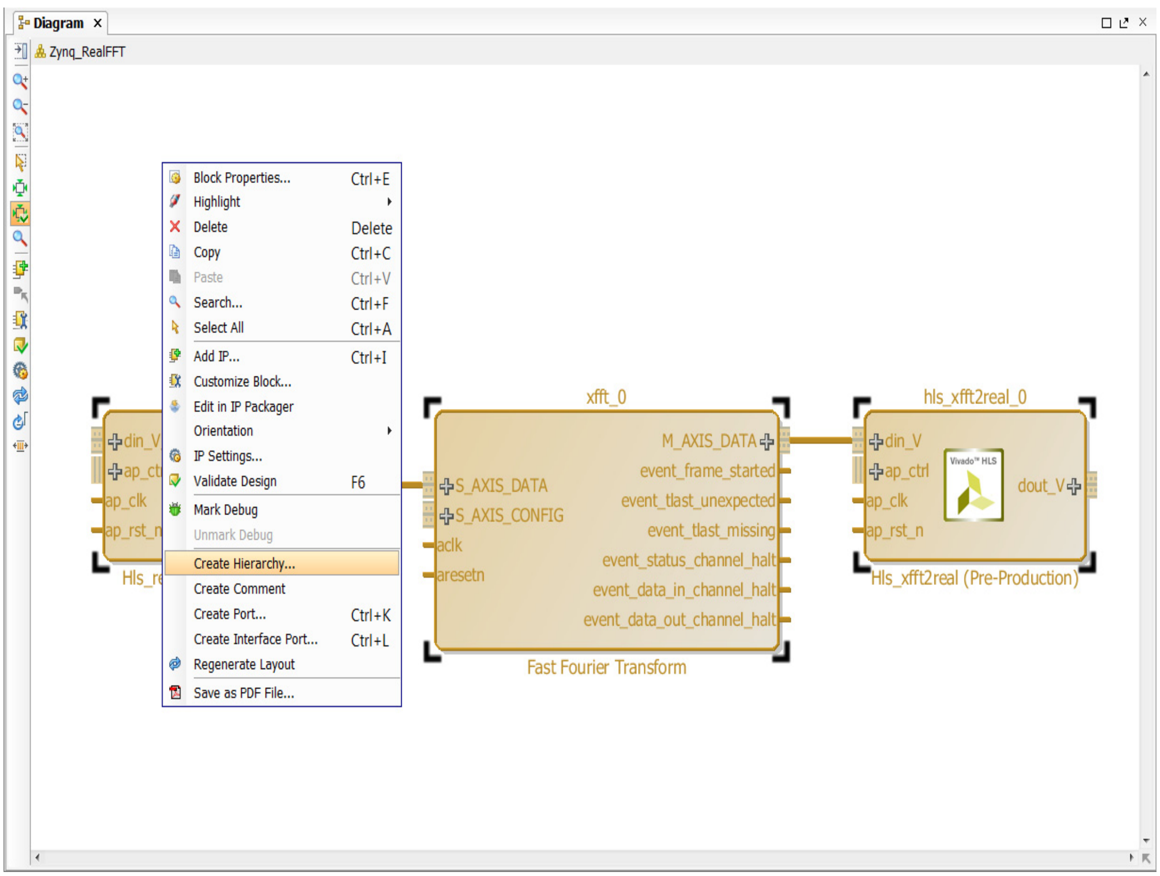


Figure 10-29: **Create a Hierarchy Block**

c. In the Create Hierarchy dialog box, enter `RealFFT` as the Cell name.

d. Ensure that the **Move '3' selected blocks to new hierarchy** option is checked, as shown in Figure 10-30.

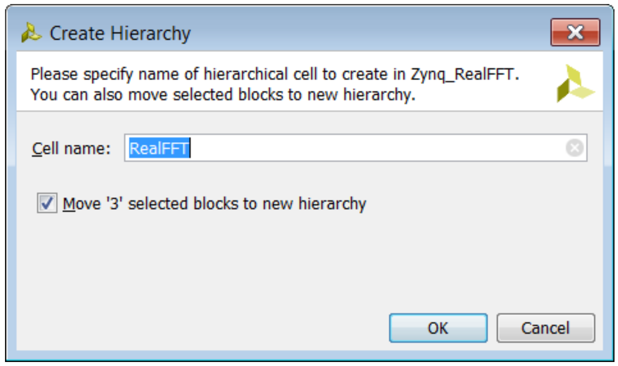


Figure 10-30: **Name Hierarchy Block**

e. Click **OK**.

The diagram will appear as shown in Figure 10-31.

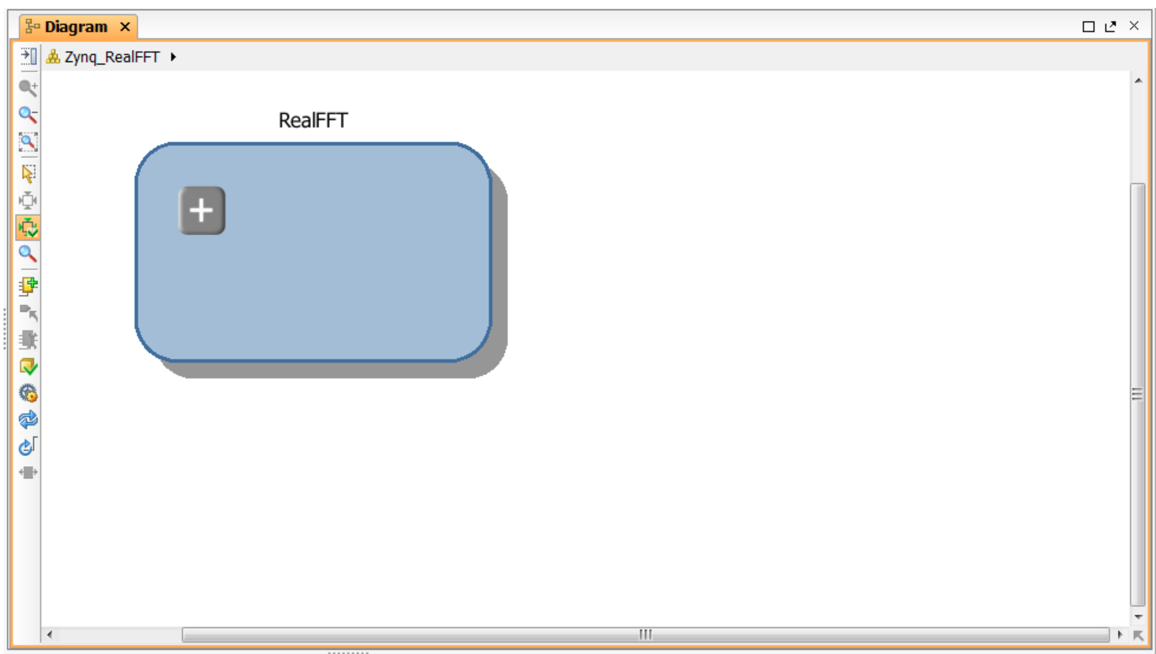


Figure 10-31: **New Hierarchy Block**

Add pins to the `RealFFT` hierarchical block so that you can connect it at the top-level.

9. Double-click the `RealFFT` block to open its diagram.

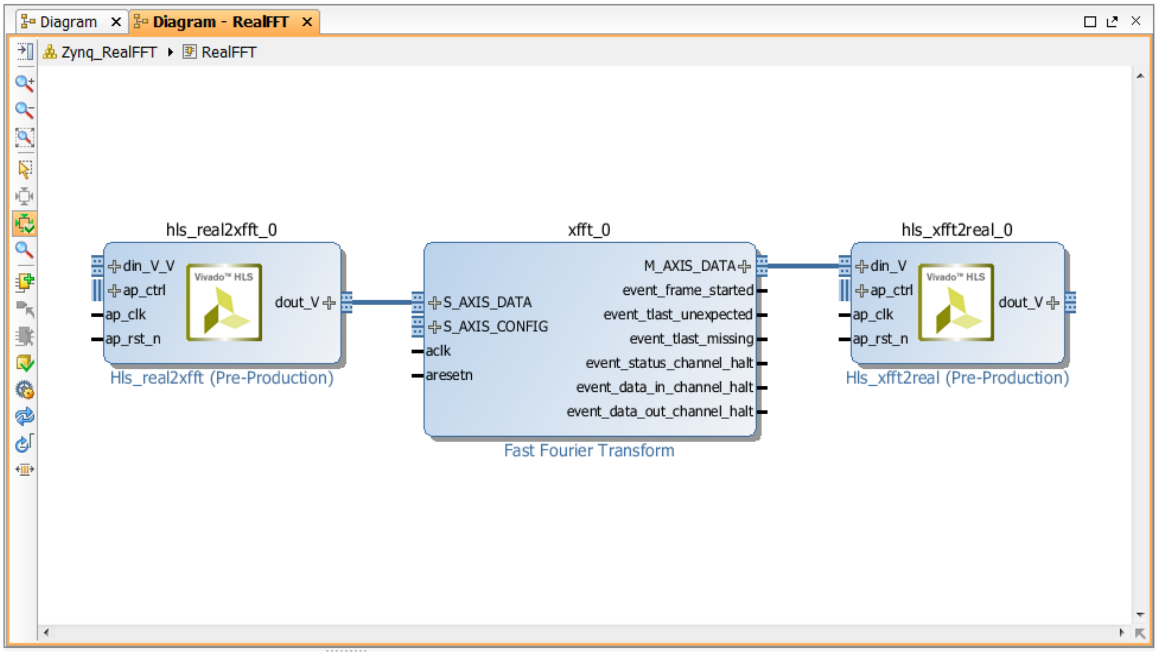


Figure 10-32: **RealFFT Diagram**

10. Right-click the `din_V_V` pin of the `hls_real2xfft_0` block and select **Create Interface Pin** from the context menu.

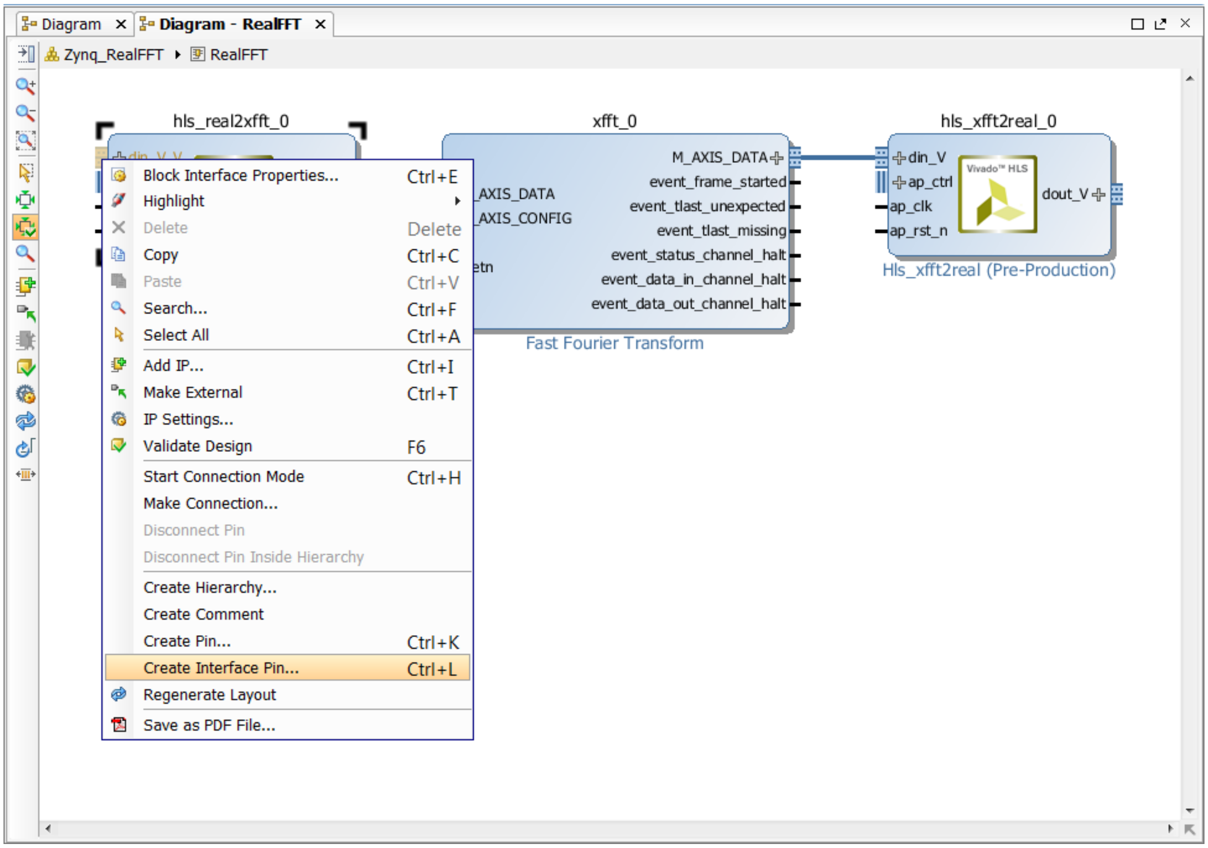


Figure 10-33: **Creating an Interface Pin**

11. In the Create Interface Pin dialog box, change the Interface name to `realfft_s_axis_din`.
    a. Accept all other defaults and click **OK**.

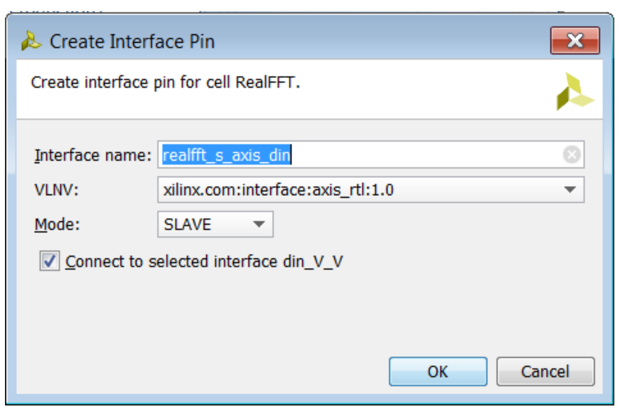


Figure 10-34: **Naming an Interface Pin**

12. Right-click the `ap_clk` pin of the `hls_real2xfft_1` block and select **Create Pin** from the context menu.

a. Change the name to `aclk` and click **OK**.

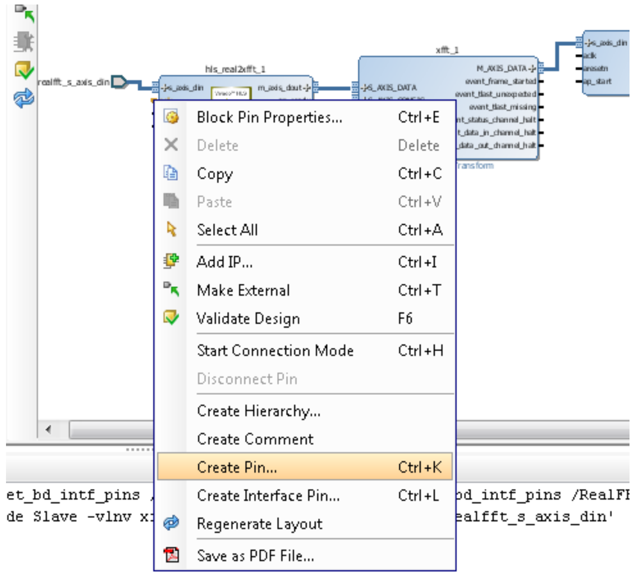


Figure 10-35: **Create a Clock Pin**

After you create this clock pin, the `RealFFT` diagram appears as shown in Figure 10-36.

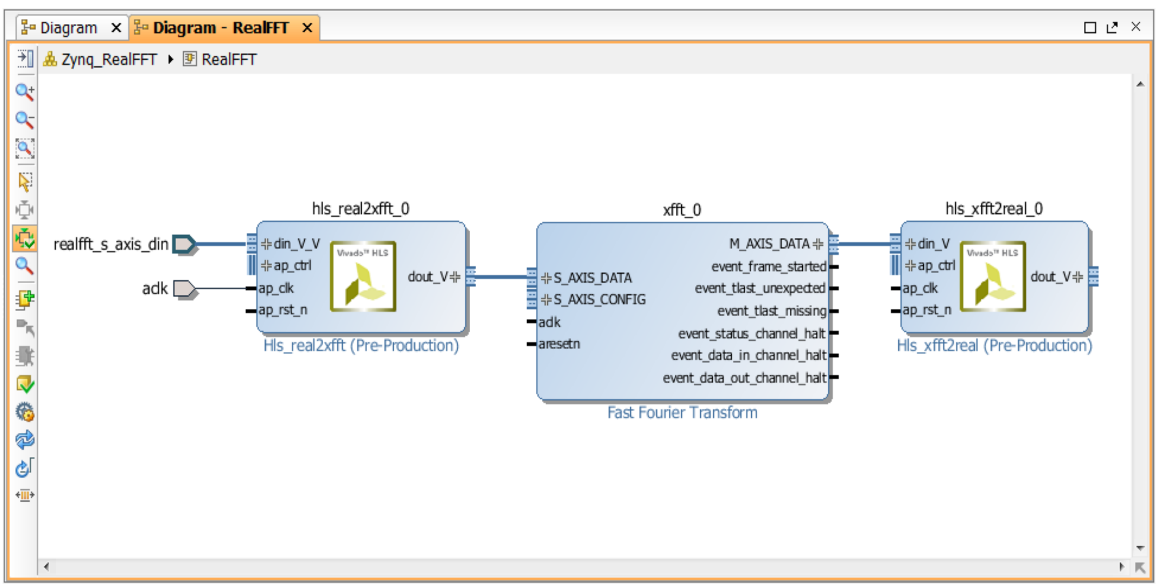


Figure 10-36: **RealFFT Diagram with Interface Pin and Clock Pin**

13. Following the procedures in steps 10 to 12:

    a. Create an interface pin called `realfft_m_axis_dout` connected to the `dout_V` pin of the `hls_xfft2real` component.

    b. Create a pin for `aresetn` (from any one of the blocks).

After this step, the `RealFFT` diagram appears as shown in Figure 10-37.

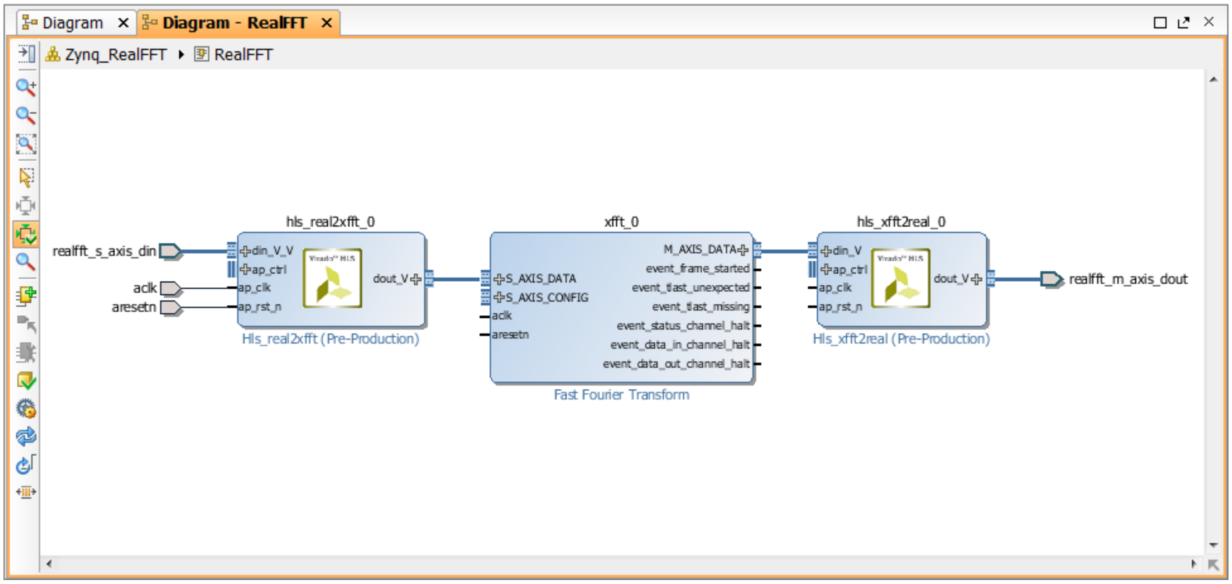


Figure 10-37: **RealFFT Diagram with All Pins**

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.

14. 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_0` component and verify that the Const Val field in the Customize IP dialog is set to 1.

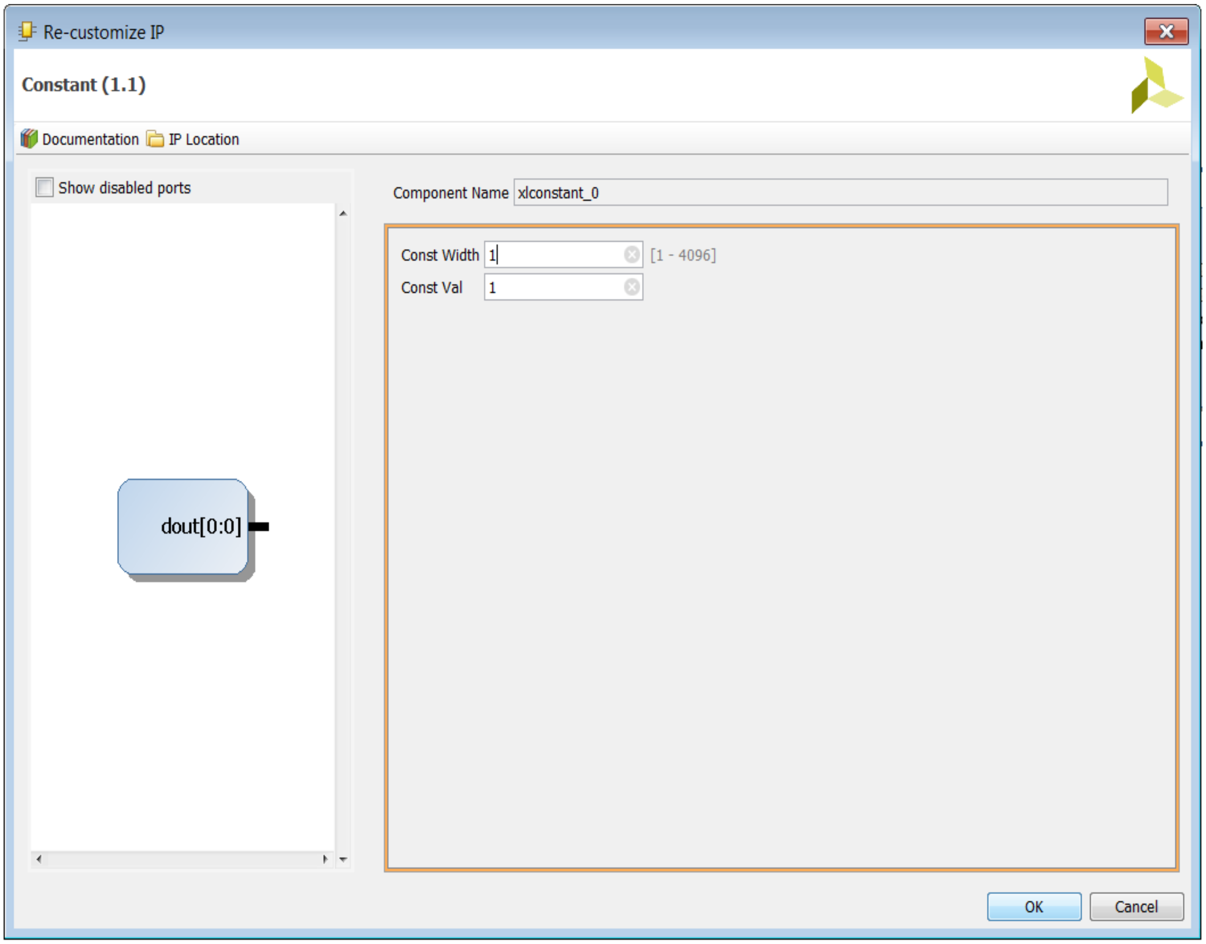


Figure 10-38: **Create a Constant 1 Tie-Off**

15. Expand the `ap_ctrl` interface by clicking the **+** sign next to it on the `hls_real2xfft` and `hls_xfft2real` block symbols and:
    a. Connect the output pin of `xlconstant_0` to the `ap_start` pin of `hls_real2xfft_0`.
    b. Connect the output pin of `xlconstant_0` to the `ap_start` pin of `hls_xfft2real_0`.
16. Similarly, connect all remaining component `dout_V` and `reset` pins to the `RealFFT` block diagram `aclk` and `aresetn` pins respectively.
17. Add another `xlconstant` block and configure it with a Const Width of 16 and Const Val of 0.
18. Expand the `S_AXIS_CONFIG` interface of the FFT block and connect `s_axis_config_tdata` and `s_axis_config_tvalid` to the new constant block.

Leave all other output pins of the components disconnected. The final `RealFFT` diagram appears with the connections shown in Figure 10-39.

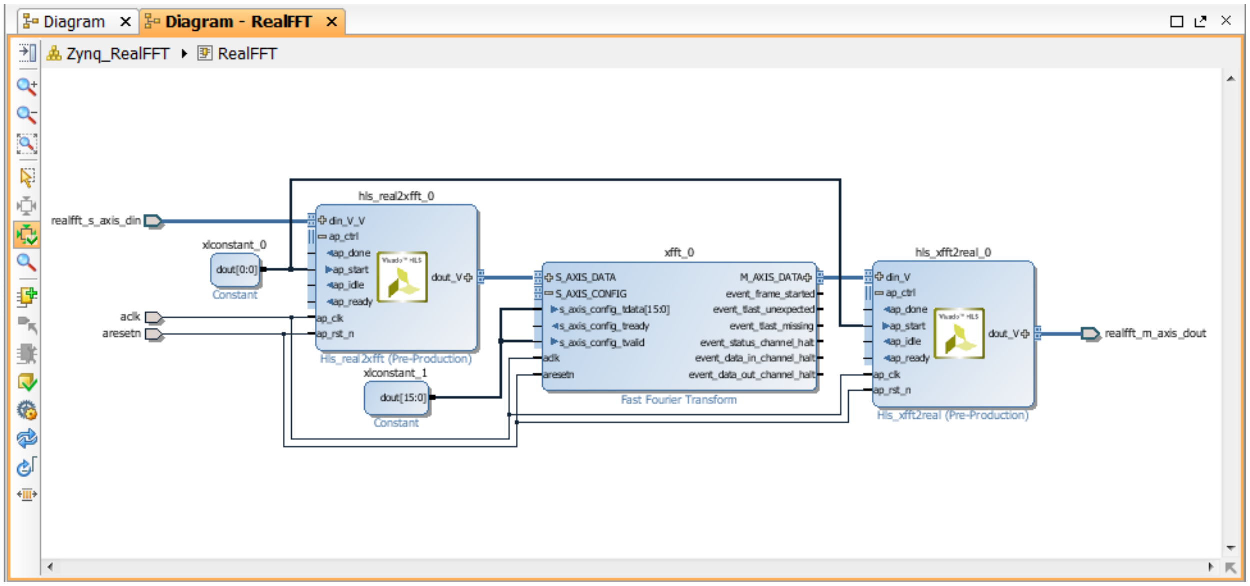


Figure 10-39: **Final RealFFT Diagram**

19. Close the `RealFFT` diagram tab and return to the top-level `Zynq_RealFFT` diagram.

20. Create the Zynq system.

    a. Right-click the canvas of the top-level diagram and select **Add IP** from the context menu.

    b. Type `zynq` in the search box, select **ZYNQ7 Processing System** and press **Enter**.

    c. Notice that designer assistance is available and click the **Run Block Automation** link. Accept the defaults in the dialog by clicking **OK**.

    d. Double-click the **processing_system7_0** component to enter the Re-customize IP wizard for the ZYNQ7.

    e. Click the **Presets** button near the top of the wizard screen, select the **ZC702 Development Board Template**, and click **OK**.

    f. Click **PS-PL Configuration** in the Page Navigator pane on the left of the wizard.

    g. 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.

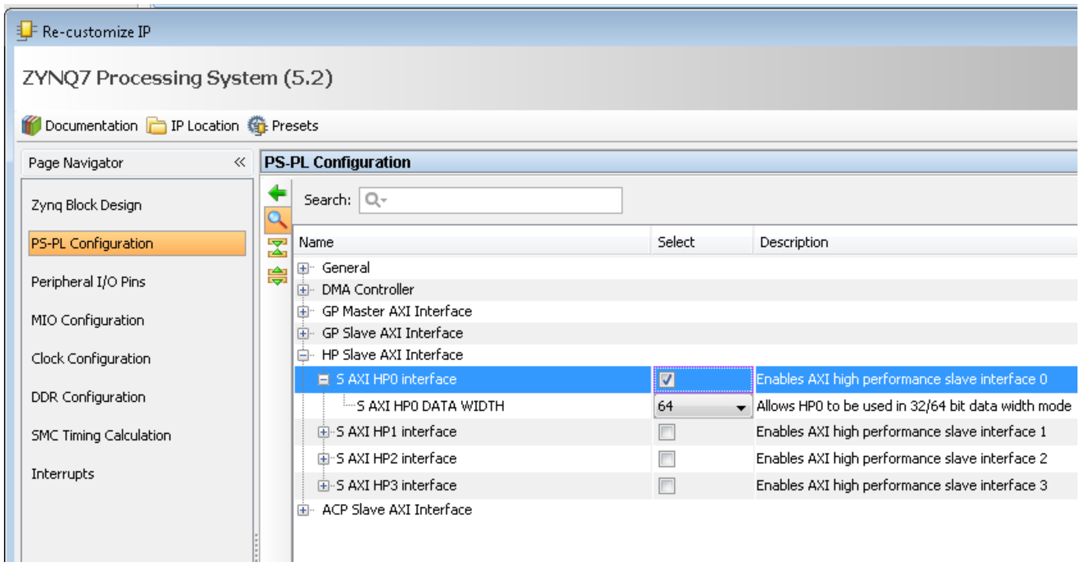


Figure 10-40: **Configuring Port HP0**

h. Select **Clock Configuration** in the Page Navigator, expand PL Fabric Clocks, and change the requested frequency to 100 (MHz).

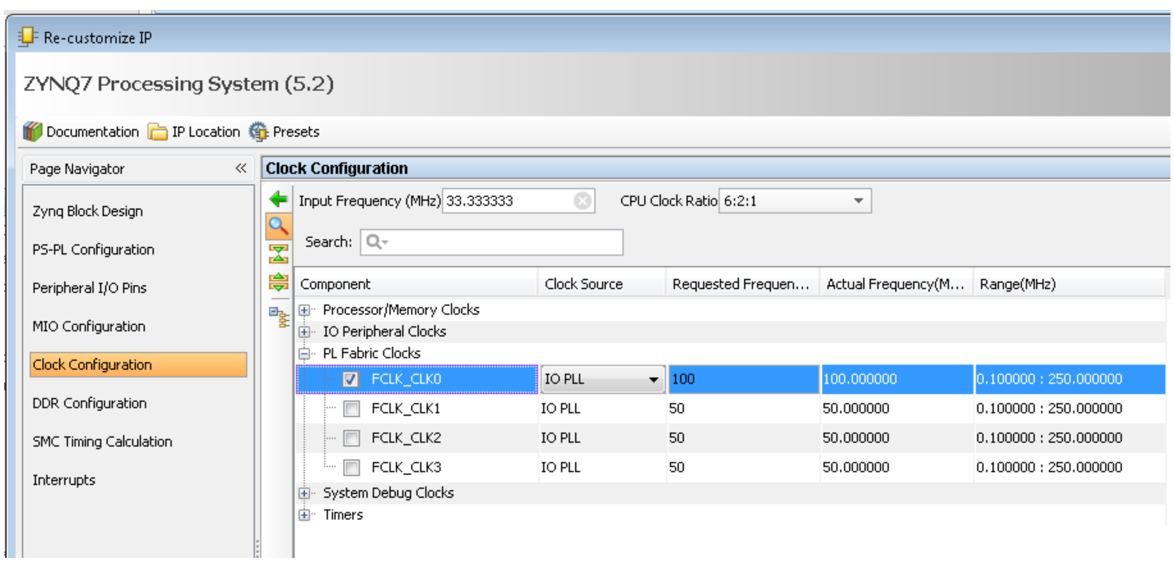


Figure 10-41: **Configuring the Clock**

i. Leave all other settings at their defaults; click **OK** to apply customizations.

21. Make a connection from RealFFT block's realffft_s_axis_din to Zynq AP SoC's S_AXI_HP0, accept the defaults in the Make Connection dialog and click OK.

IPI will place several new blocks require to complete the connection automatically, including an AXI DMA core, an AXI Interconnect and a Processor System Reset block.

22. Make a connection from the RealFFT block's realfft_m_axis_dout to the Zynq's S_AXI_HP0 interface. Accepting the defaults in the Make Connection dialog will cause IPI to use the existing AXI DMA (which has an unused write channel) and AXI Interconnect to make the 'S2MM' connection.
23. Note that Designer Assistance is again available. Run Connection Automation on /axi_dma/S_AXI_LITE and click **OK** in the resulting dialog box.
24. Connect the `aclk` and `aresetn` ports of the RealFFT hierarchical block to nets processing_system7_0 pin FCLK_CLK0 and rst_processing_system7_0_100M pin peripheral_aresetn respectively.
25. To complete the design, run Validate Design. When validation completes successfully, the block diagram should look like Figure 10-42.

Step 5: Implementing the System

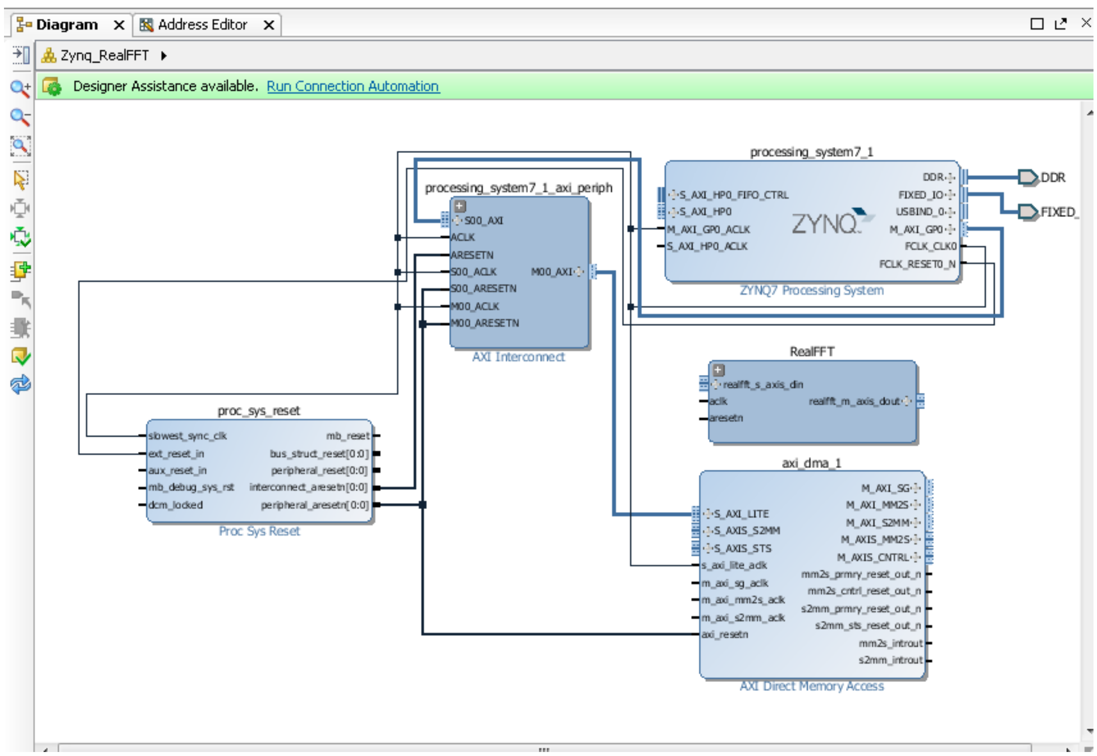


Figure 10-42: **Zynq Diagram with Internal Connections**

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 called `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 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** option is checked, and click **OK**.
3. From the Vivado **File** menu, select **Launch SDK**.
4. Click **OK** to launch SDK.
5. 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**.
6. Power up the ZC702 board and program the FPGA.

   Ensure the board has all the connections to allow you to download the bitstream on the FPGA device. Refer to the documentation that accompanies the ZC702 development board.
7. Click **XilinxTools > Program FPGA**. The Done LED (DS3) goes on.

8. 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.
9. Right-click application project `Zynq_Design_Test` in the Explorer pane.
   a. Select **Run As > Launch on Hardware**.
10. Switch to the **Terminal** tab and confirm that `Hello World` was received.
11. 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 10-43).

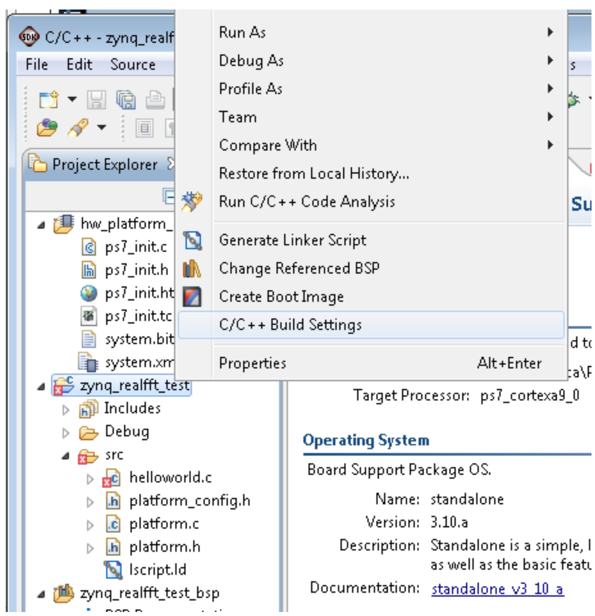


Figure 10-43: **Specify C/C++ Build Settings**

   b. Add the ARM gcc linker libraries.
      i. In the Tool Settings tab, select **'ARM gcc linker' > Libraries**.
      ii. Click the **Add** icon.

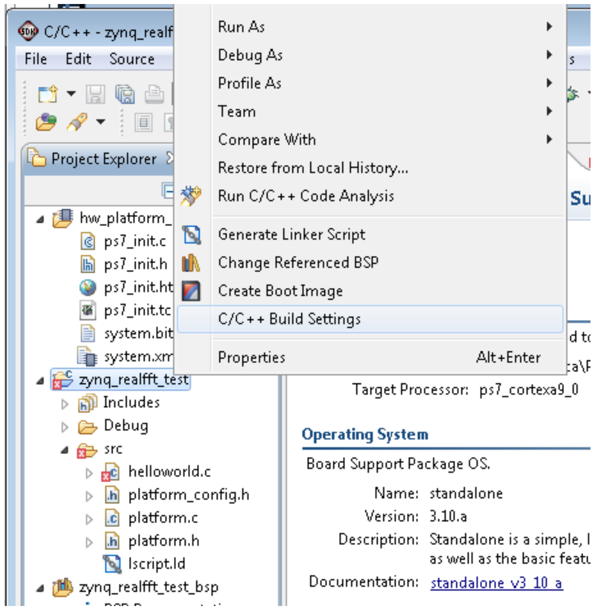


Figure 10-44: **C/C++ Build Settings**

c. Enter `m` in the text field in the Enter Value dialog box and click **OK**.

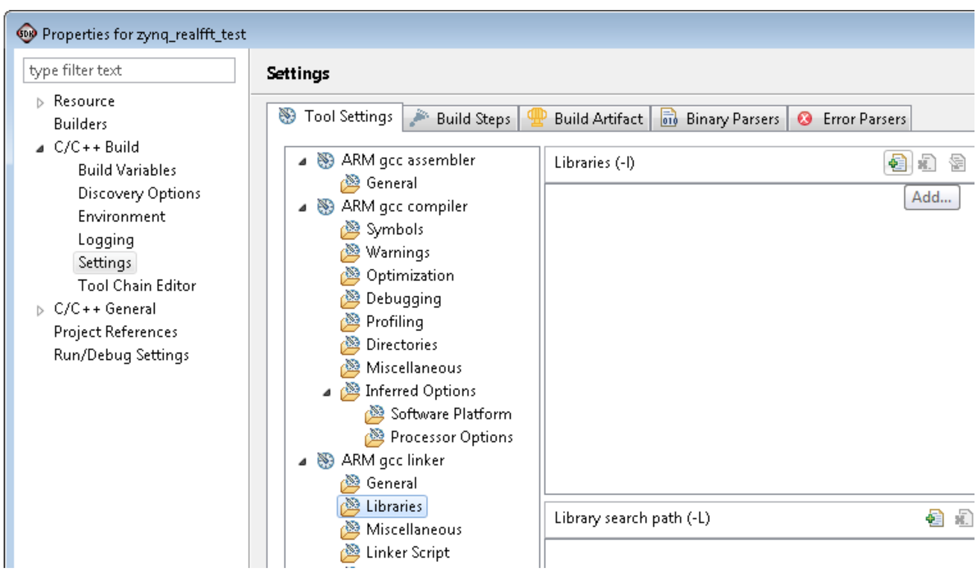


Figure 10-45: **Library Setting**

d. Click **OK** to exit the Properties for the `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
#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 to use 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
init_platform();
print("---------------------------------------\n\r");
print("- RealFFT PL accelerator test program -\n\r");
```

```
print("-------------------------------------\n\r");

// Initialize the (simple) DMA engine
status = init_dma(&axiDma);
if (status != XST_SUCCESS) {
   exit(-1);
}
```

d. Generate a stimulus waveform:

```
// Generate a waveform to be input to FFT
for (i = 0; i < 4; i++)
  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:

```
// DMA enough data to push out first result data set completely
status = XAxiDma_SimpleTransfer(&axiDma, (u32)realdata,
   4 * REAL_FFT_LEN * sizeof(short), XAXIDMA_DMA_TO_DEVICE);

// Do multiple DMA xfers 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++) {
```

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);
} while(status);
```

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.

```
// 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++) {
           // 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");
   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);
   }
}
```

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");
        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 re-build 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 an HLS design as IP into IP Integrator.
- How to connect HLS IP to a Zynq AP SoC using AXI4-Lite interfaces and AXI4-Stream interfaces.
- How to configure HLS IP with AXI4-Lite in software.
- How to control DMAs using AXI4 Streams in software.

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.

Lab 1 Description

Generates 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. See the information in Locating the Tutorial Design Files.

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.
   - On Windows, go to **Start > All Programs > Xilinx Design Tools > Vivado 2015.3 > Vivado HLS > Vivado HLS 2015.3 Command Prompt**.
   - On Linux, open a new shell.

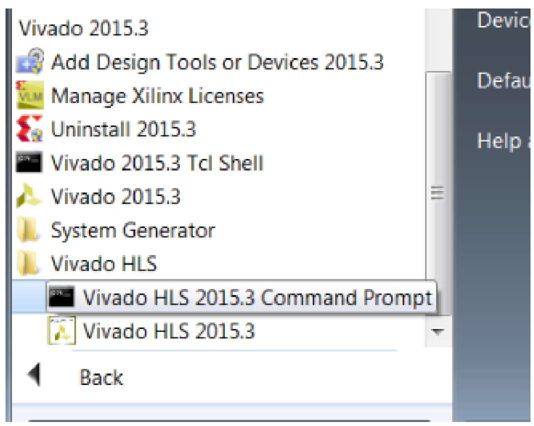


Figure 11-1: **Vivado HLS Command Prompt**

2. Using the command prompt window, change the directory to `Vivado_HLS_Tutorial\Using_IP_with_SysGen\lab1`.
3. Type `vivado_hls -f run_hls.tcl` to create the HLS IP.

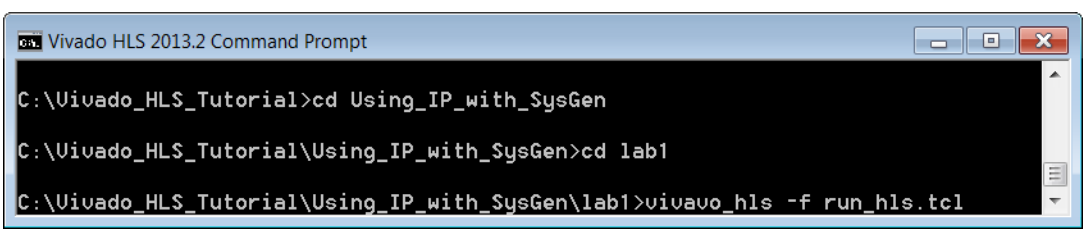


Figure 11-2: **Create the HLS Design**

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.
    - On Windows use the desktop icon.
    - On Linux, open a new shell and type **sysgen**.

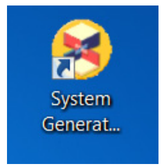


Figure 11-3: **System Generator Icon**

2. When Matlab invokes, click the **Open** toolbar button. As shown in Figure 11-4.

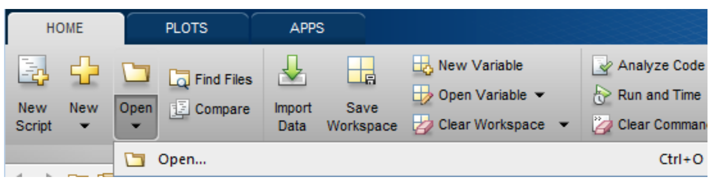


Figure 11-4: **Open the System Generator Design**

3. Navigate to the tutorial directory `Vivado_HLS_Tutorial\Using_IP_with_SysGen\lab1` and select the file `fir_sysgen.slx`, as shown in Figure 11-5.

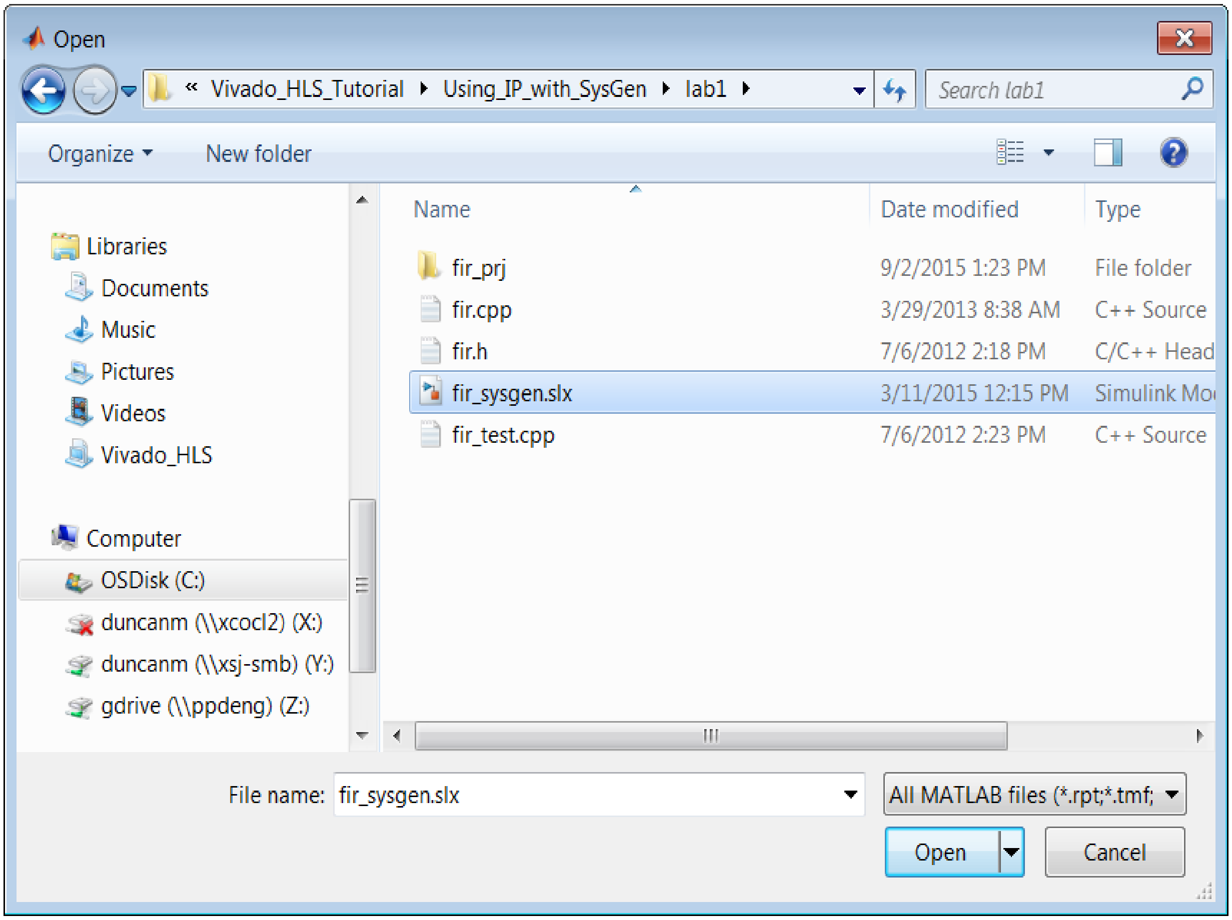


Figure 11-5: **Select File fir_sysgen.slx**

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 11-6.

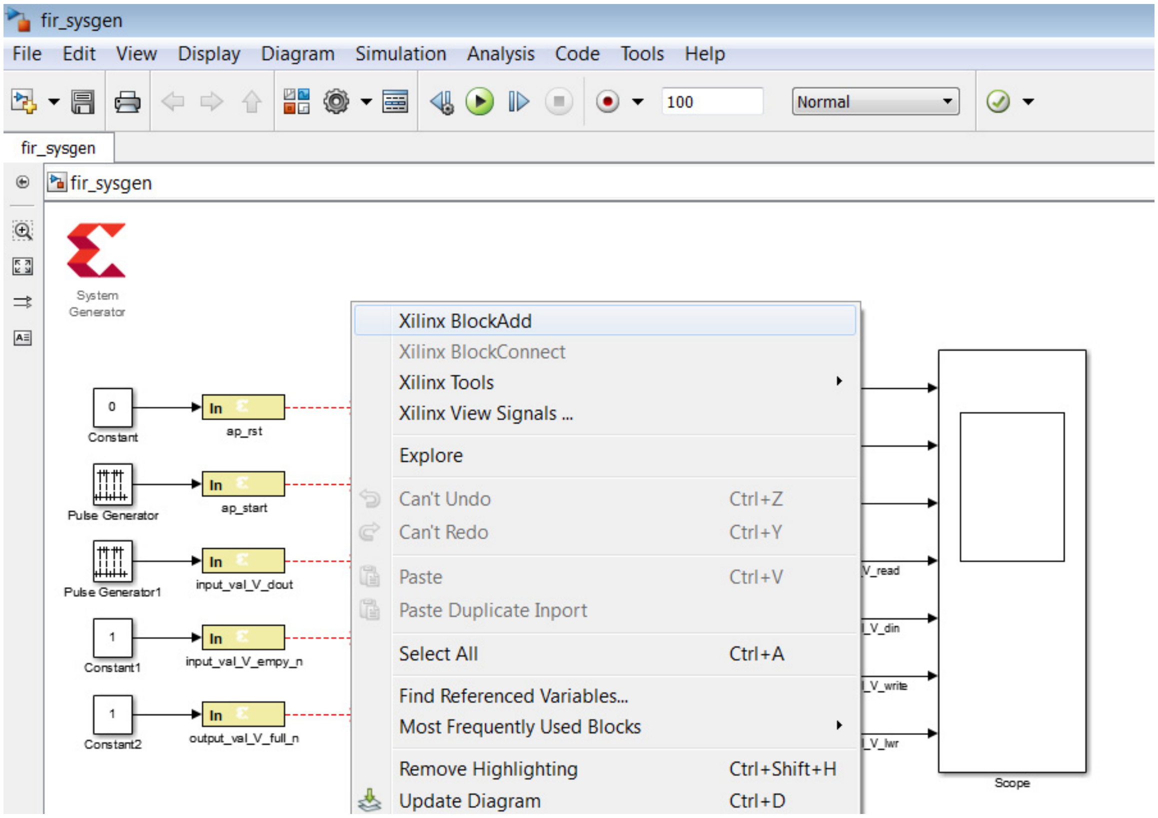


Figure 11-6: **Adding a New Block**

5. Type `hls` in the Add Block field.
6. Select **Vivado HLS**.

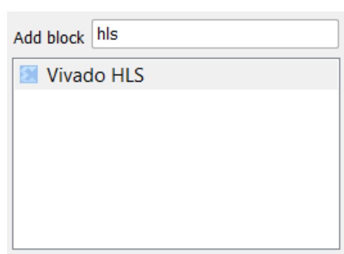


Figure 11-7: **Selecting a Vivado HLS IP Block**

7. Double-click the **Vivado HLS** block to open the Vivado HLS dialog box.
8. Navigate to the `fir_prj` project and click **Choose** to select the `solution1` folder.

IMPORTANT: *System Generator for DSP uses the location of the solution folder to identify the IP.*

9. Click **OK** to load the IP block, as shown in Figure 11-8.

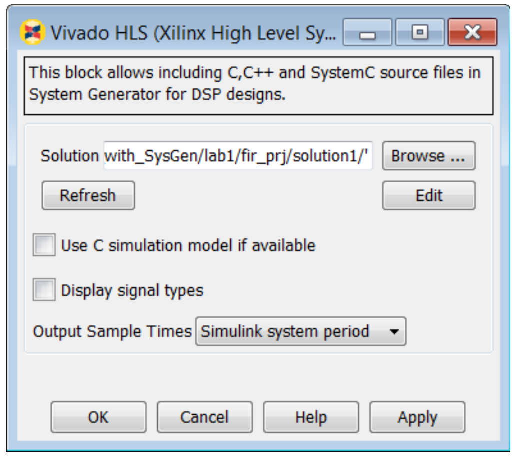


Figure 11-8: **Selecting the FIR IP Block**

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 Figure 11-9.

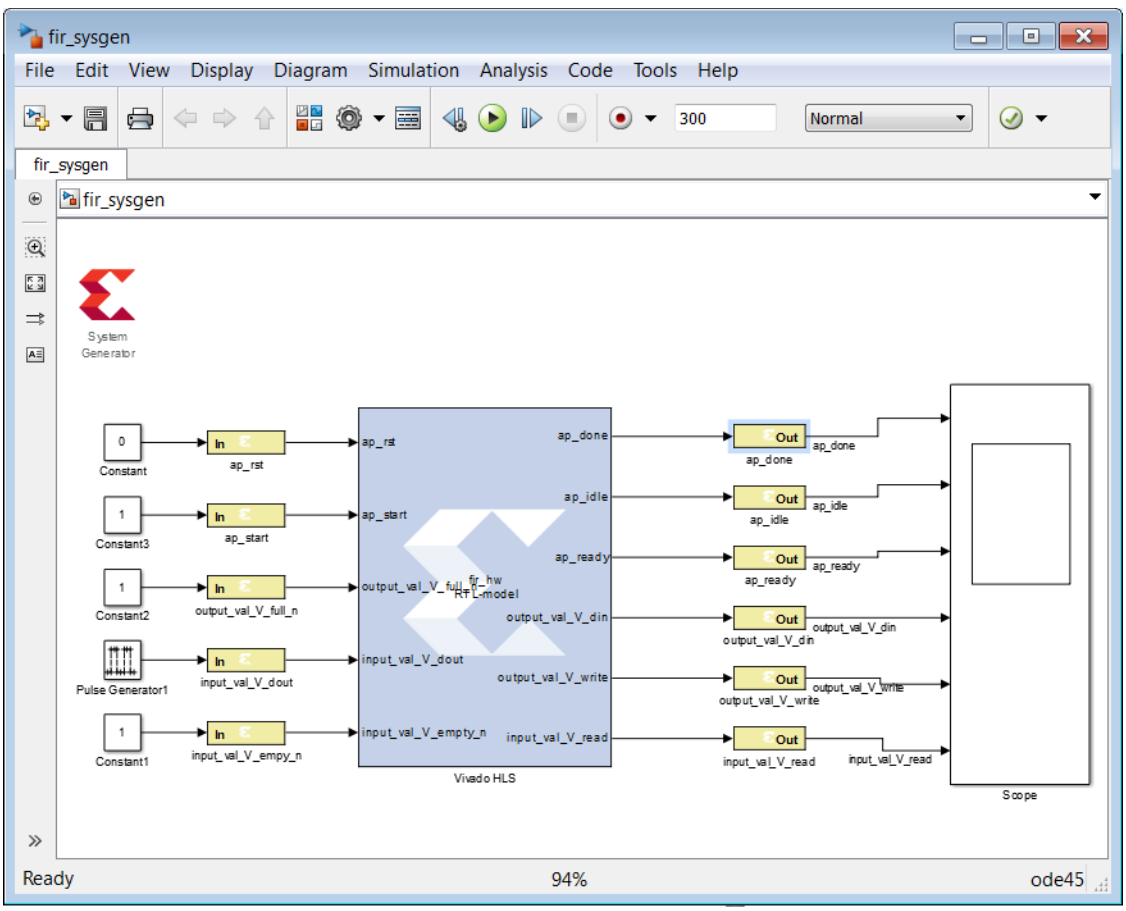


Figure 11-9: **Design with All Connections**

11. Ensure the simulation stop time says 300.

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.

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

Solution Centers

See the Xilinx Solution Centers for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips.

References

1. *Introduction to FPGA Design with Vivado High-Level Synthesis* (UG998)
2. *Vivado® Design Suite User Guide: High-Level Synthesis* (UG902)
3. *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing* (UG973)
4. Vivado Design Suite Documentation

Training Resources

Xilinx provides a variety of training courses and QuickTake videos to help you learn more about the concepts presented in this document. Use these links to explore related training resources:

1. C-based Design: High-Level Synthesis with the Vivado HLS Tool Training Course
2. C-based HLS Coding for Hardware Designers Training Course
3. C-based HLS Coding for Software Designers Training Course

4. Vivado Design Suite QuickTake Video Tutorials
5. Vivado Design Suite QuickTake Video Tutorials: Vivado High-Level Synthesis
6. Vivado Design Suite QuickTake Video: Getting Started with High-Level Synthesis
7. Vivado Design Suite QuickTake Video: Verifying your Vivado HLS Design
8. Vivado Design Suite QuickTake Video: Analyzing your Vivado HLS Design

Please Read: Important Legal Notices