Vivado Design Suite Tutorial:

High-Level Synthesis

UG871 (v 2013.1) April 3, 2013

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
04/03/2013	2013.1	Revised and updated the Tutorial with 10 new chapters.

Table of Contents

High-Level Synthesis Tutorials

Overview

This Vivado High-Level Synthesis tutorial contains a number tutorial exercises to help explain and demonstrate all steps in the process of transforming C, C++ and SystemC code to an RTL implementation using High-Level Synthesis.

The following summarizes each tutorial exercise.

High-Level Synthesis Introduction

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

The tutorial shows how an initial RTL implementation is transformed into both a low area and high throughput implementation by using optimization directives.

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 concludes by showing how arbitrary precision types can be debugged.

Interface Synthesis

The interface synthesis tutorial reviews all aspect of creating ports for the RTL design. You will learn how to control block-level IO port protocols, port IO protocols, how arrays in the C function can be implemented as multiple implementations and how AXI4 bus interfaces are implemented.

The tutorial completes with a design example where the IO 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 the accuracy of the results.

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 is taken through a number of analysis and optimization stages that highlight all the features of the analysis perspective and provide the basis for a design optimization methodology.

Design Optimization

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

The tutorial concludes by showing how the insights learned from analyzing the design can be used to update the initial C code and create a more optimal implementation of the design.

RTL Verification

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

Using HLS IP in IP Integrator

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

Using HLS IP in a Zynq Processor Design

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

Using HLS IP in System Generator for DSP

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

Software Requirements

This tutorial requires that the Vivado Design Suite 2013.1 release or later is installed.

Hardware Requirements

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

Obtaining the Tutorial Designs

The designs used in the tutorial exercises are available from the Xilinx website. As shown in Figure 1, they are available as a zipped archive on the Tutorial Documentation page.

IMPORTANT: *All the tutorial examples for Vivado High-Level Synthesis are available for download from www.xilinx.com. Please refer to http://www.xilinx.com/cgi-bin/docs/rdoc?v=2013.1;t=vivado+tutorials*

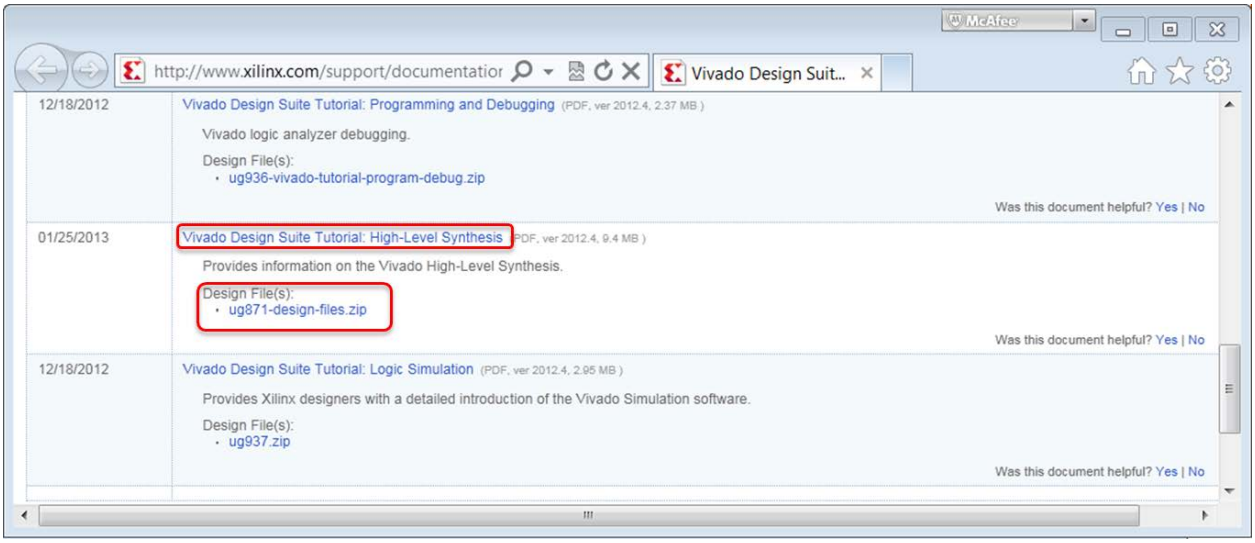


Figure 1: High-Level Synthesis Tutorial Design Files

Preparing the Tutorial Design Files

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

This tutorial assumes the tutorial data directory **Vivado_HLS_Tutorial** is unzipped and placed in the location **C:\Vivado_HLS_Tutorial**.

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

High-Level Synthesis Introduction

Overview

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

The tutorial shows how an initial RTL implementation is transformed into both a low area and high throughput implementation by using optimization directives.

Lab1

Explains how to setup a High-Level Synthesis (HLS) project and perform all major steps in the HLS design flow: validate the C code, create and synthesize a solution, verify the RTL and package the IP.

Lab2

Demonstrates how to us the Tcl interface.

Lab3

Optimize the design using optimization directives. This lab creates multiple versions of the RTL implementation and compares the different solutions.

Tutorial Design Description

The tutorial design file can be download from the Xilinx website. Refer to the information in Obtaining the Tutorial Designs.

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

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

- Create a version of the design with the smallest area
- 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 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

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

IMPORTANT: *This 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: Creating a New Project

1. Open the Vivado HLS Graphical User Interface (GUI):
    a. On windows systems, open Vivado HLS by double-clicking on the **Vivado HLS 2013.1** desktop icon (Figure 2)
    b. On Linux systems, type **vivado_hls** at the command prompt.

Figure 2: The Vivado HLS Desktop Icon

TIP: *You can also open Vivado HLS using the Windows menu* ***Start*** *>* ***All Programs*** *>* ***Xilinx Design Tools*** *>* ***Vivado 2013.1*** *>* ***Vivado HLS*** *>* ***Vivado HLS 2013.1****.*

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

Figure 3: The Vivado Welcome Page

2. In the Welcome Page, select **Create New Project** to open the Project Wizard.

3. As shown in Figure 4,
    a. Enter the project name as **fir_prj**.
    b. Click **Browse** to navigate to the location of the **lab1** directory.
    c. Select the **lab1** directory and click **OK**.
    d. Click **Next**.

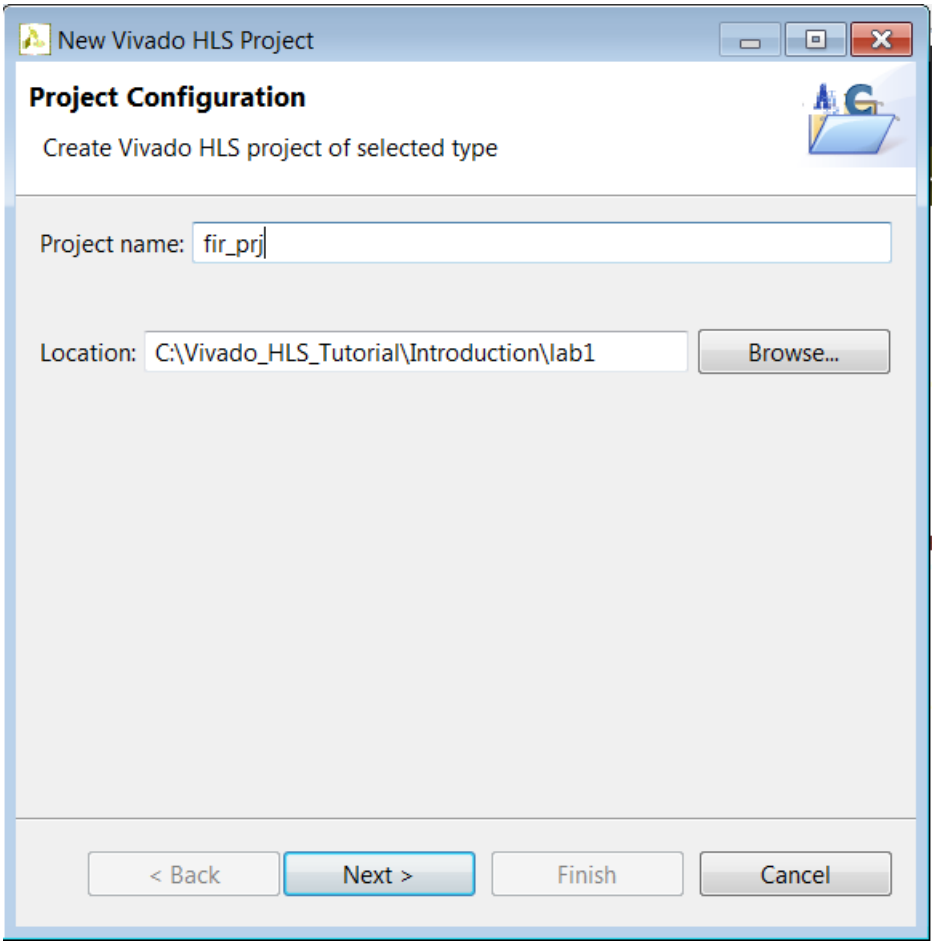


Figure 4: Project Configuration

This information defines the name and location of the Vivado HLS project directory. In this case, the project directory is **fir_prj** and it will reside 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 press **Open**.
    d. Click **Next**.

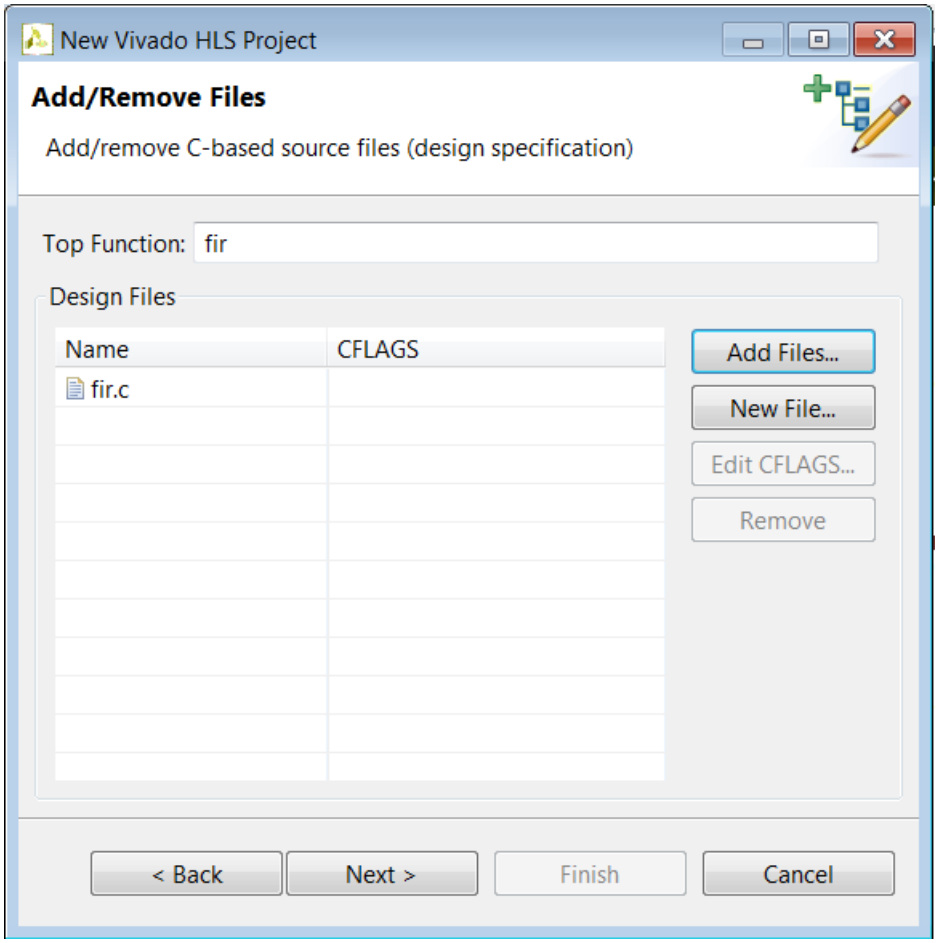


Figure 5: Project Design Files

In this lab exercise there is only a single C design file. In cases where there are multiple C files to be synthesized, they should all be added to the project at this stage.

Any header files, which 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 (e.g. -I<path_to_header_file_dir).

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

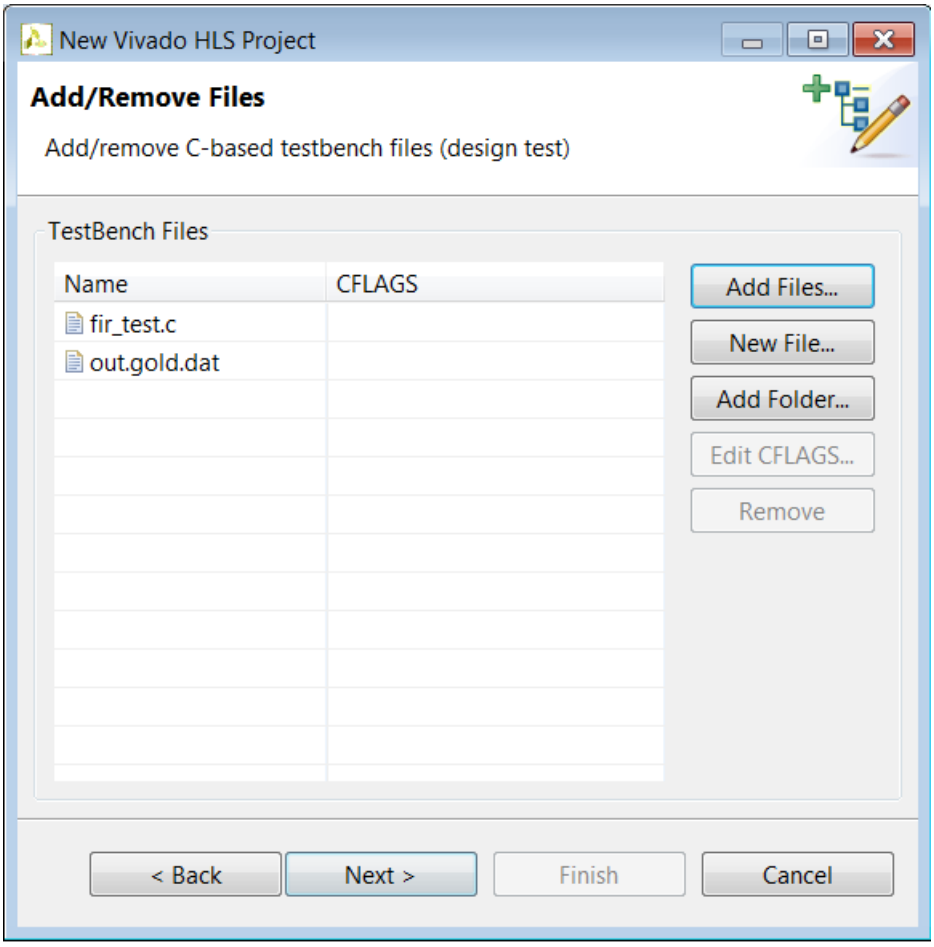


Figure 6: Test Bench Files

5. Use 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) are executed in sub-directories of the solution. If you do not include all the files used by the test bench (for example, data files which are read by the test bench, such as out.gold.dat), C and RTL simulation might fail after synthesis due to an inability to find the data files.

The Solution Configuration window (shown in Figure 7) specifies the technical specifications of the first solution. A project can have multiple solutions, each using a different target technology, package, constraints, and/or synthesis directives.

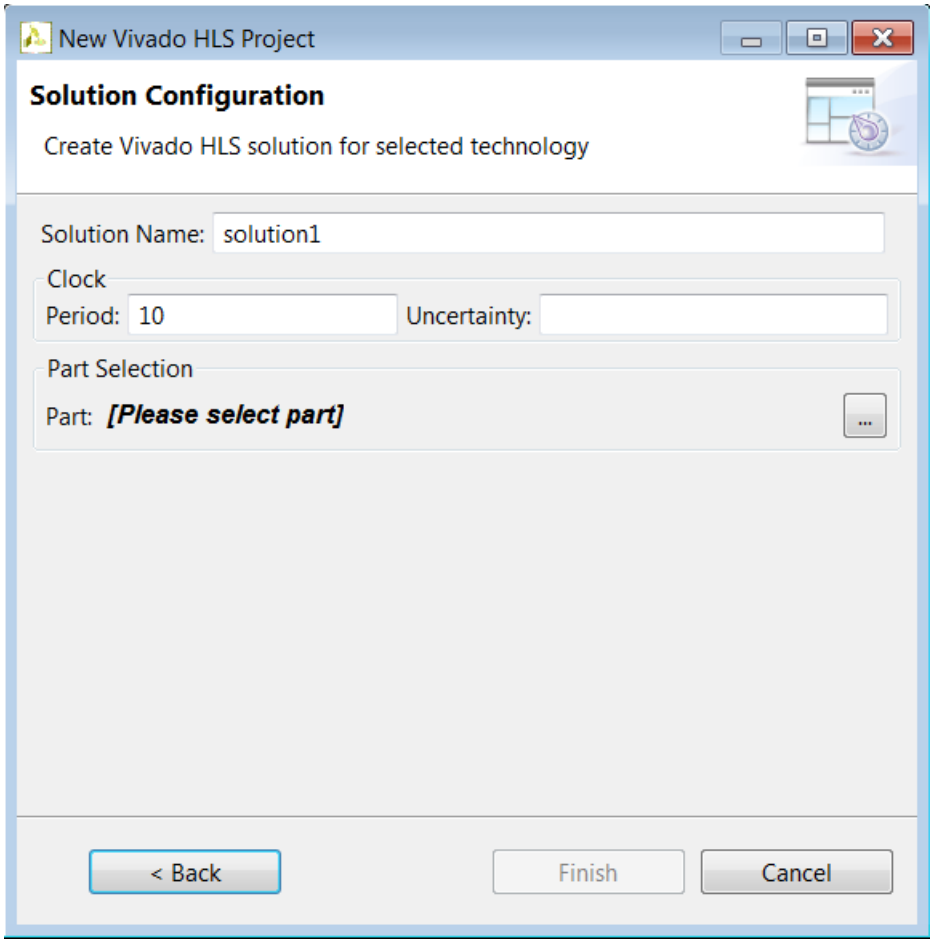


Figure 7: Solution Configuration

7. Accept the default solution name (**solution1**), clock period (**10ns**) 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. The following selections in the drop-down filters can be used to help refine the parts list.
   - Product Category: General Purpose
   - Family: Kintex®-7
   - Sub-Family: Kintex-7
   - Package: fbg484
   - Speed Grade: -2
   - Temp Grade: Any
10. Click **OK**

The statement "Please Select Part" statement shown in Figure 7 will now show the selected part.

11. Press **Finish** to open the Vivado HLS project as shown in Figure 8.

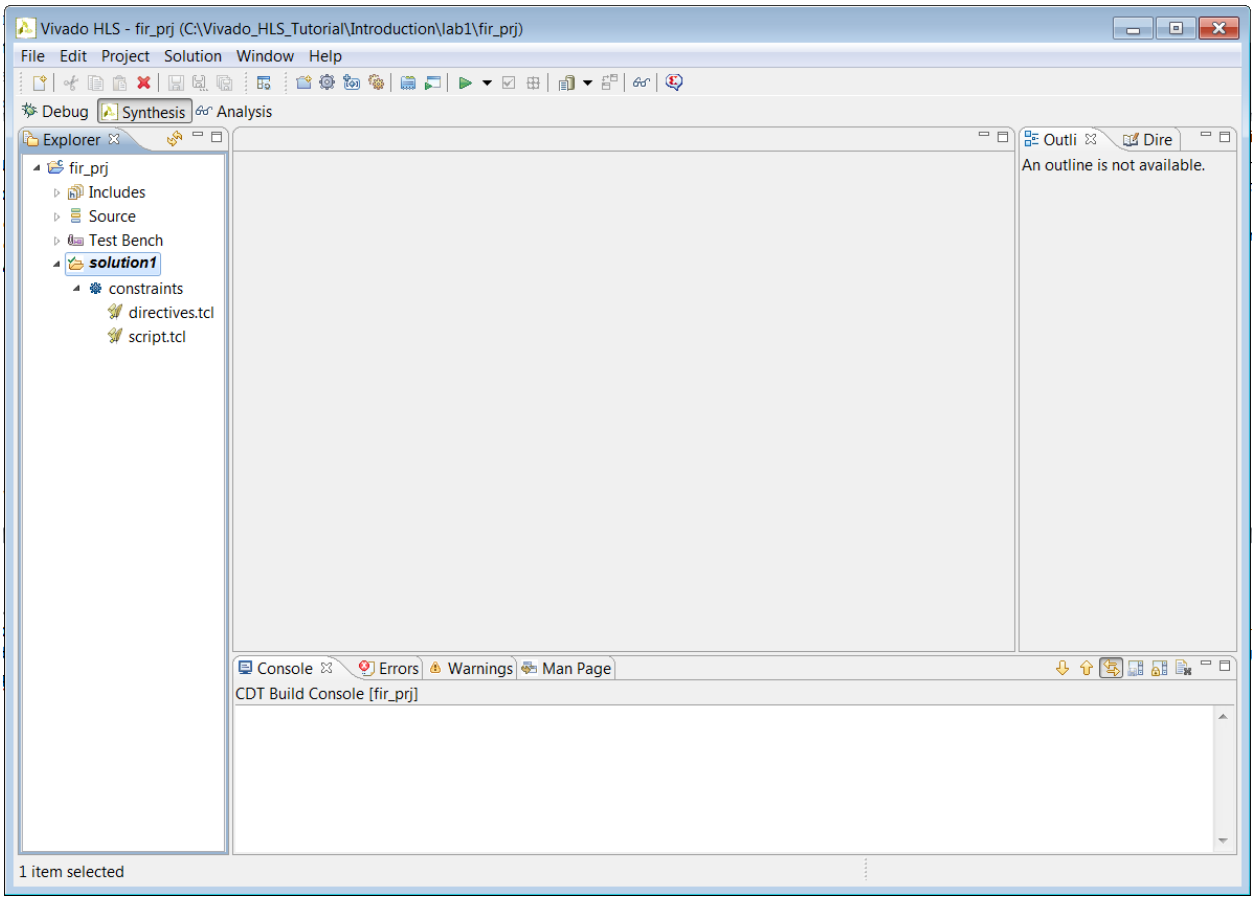


Figure 8: Vivado HLS Project

You can see the project name on the top line of the Explorer pane. A Vivado HLS project arranges data in a hierarchical form.

- The project holds information on the design source, test bench, and solutions.
- The solution holds information on the target technology, design directives, and results.
- There can be multiple solutions within a project and each solution is an implementation of the same source code.

TIP: *It is always possible to access and change project or solution settings by using the corresponding Project Settings and/or Solution Settings buttons in the toolbar.*

Before proceeding further, it is worth reviewing the regions in the Graphical User Interface (GUI) and their functions. Figure 9 shows an overview of the regions.

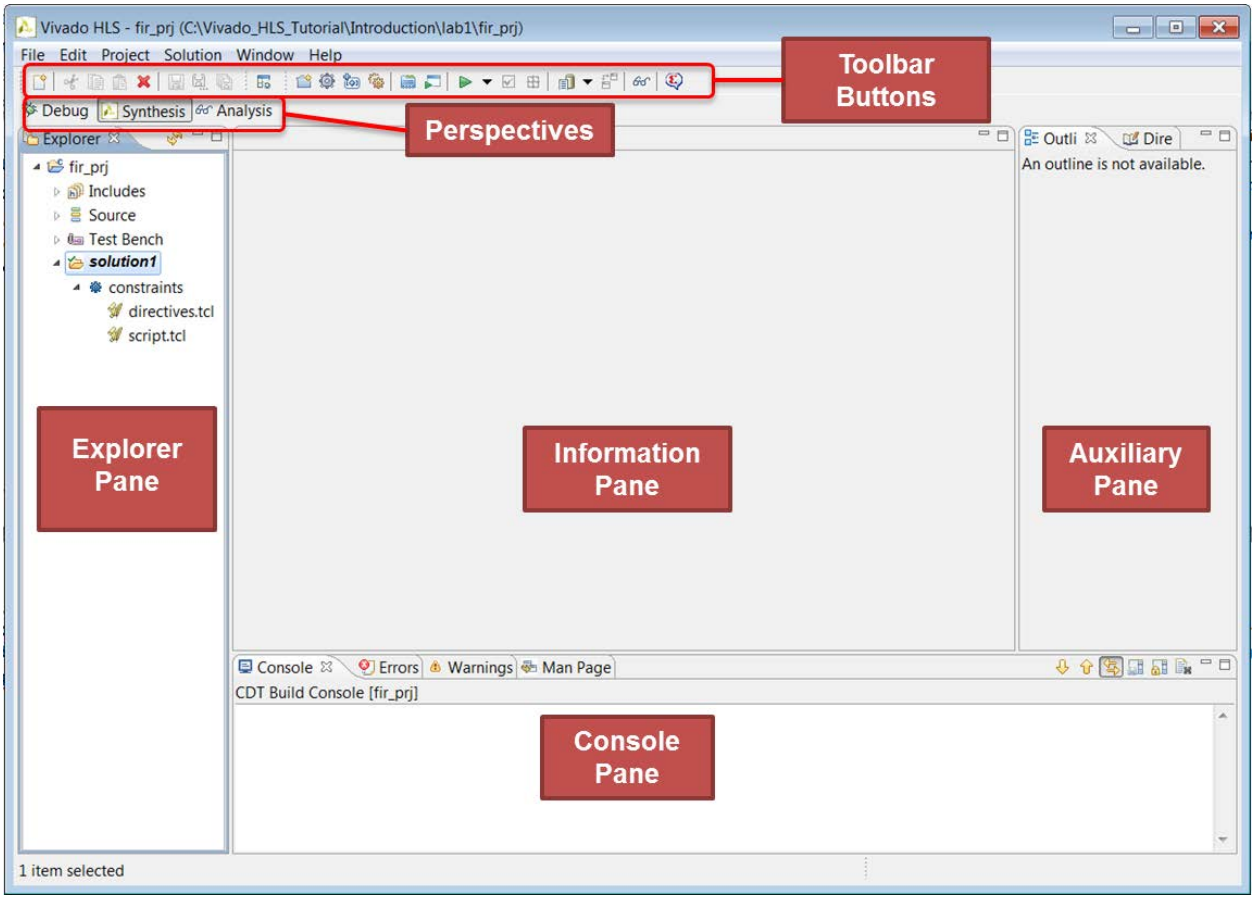


Figure 9: Vivado HLS Graphical User Interface

Explorer Pane

This shows the project hierarchy. As we proceed through the design steps of validation, synthesis, verification and IP packaging, sub-folders with the results of each step will be created inside the solution directory (named csim, syn, sim and impl respectively).

When new solutions are created, they will appear inside the project hierarchy alongside solution1.

Information Pane

This 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

This pane is cross-linked with the **Information** pane. The information shown in this pane will dynamically adjust depending on the file open in the **Information** pane.

Console Pane

This pane shows the messages produced when Vivado HLS runs. Errors and warnings are shown in tabs in the Console pane.

Toolbar Buttons

The most common operations can be performed using the Toolbar buttons. Holding the cursor over the button will cause a pop-up dialog box to appear, explaining the function. Each button also has an associated menu item available from the pull-down menus.

Perspectives

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

The default perspective is the **Synthesis** perspective, which is used for synthesizing designs., running simulations and packaging the IP.

The **Debug** perspective has panes associated with debugging the C code and can be opened after the C code has been compiled (unless Optimizing Compile mode is used).

In the **Analysis** perspective the windows are configured for analyzing the results of synthesis – this can be used only after the design has been synthesized.

Step 2: Validate the C Source Code

The first step in an HLS project is to confirm the C code is correct. This process if referred to as 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 on 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.

The result of these actions is shown in Figure 10.

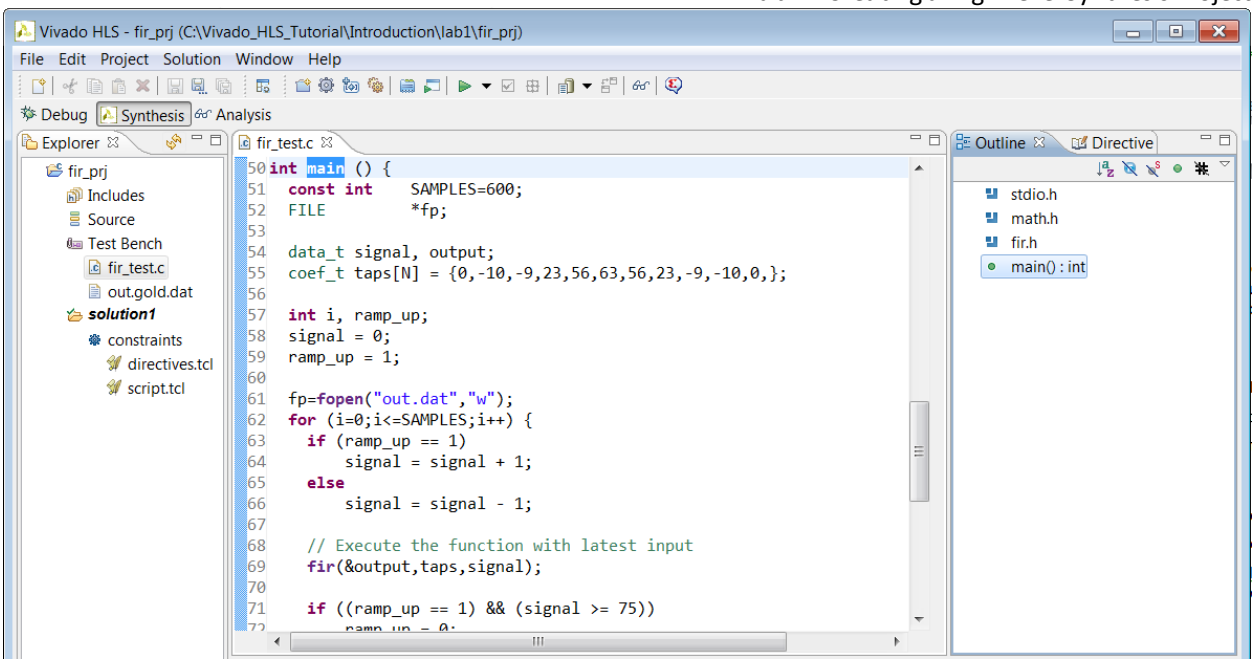


Figure 10: Reviewing the Test Bench Code

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

- The test bench saves the output from function fir into 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 (one).

The Vivado HLS tool can reuse the C test bench to perform verification of the RTL. It confirms the successful verification of the RTL if the test bench returns a value of 0. If any other value is returned by main(), including no return value, it indicates that the RTL verification failed.

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

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

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

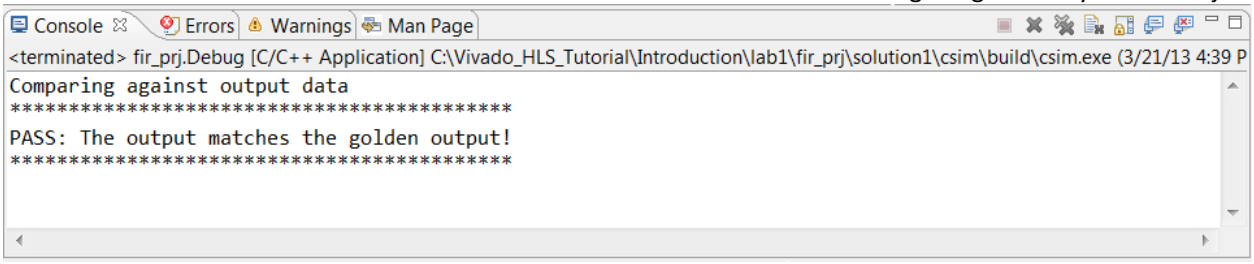


Figure 11: Results of C simulation

If the C simulation had failed, the Debug option in C Simulation dialog box can be selected to compile the design and automatically switch to the Debug perspective where a C debugger can be used to analyze and fix any problems. More details on using the Debug environment are provided in the tutorial C Validation (page 47).

The design is now ready for synthesis.

Step 3: High-Level Synthesis

In this step, the C design is synthesized into an RTL design and the synthesis report is reviewed.

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

When synthesis completes the report file opens automatically. Since 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 on **Performance Estimate** in the **Outline** tab (Figure 12).
3. In the **Details** section of the Performance Estimates, expand the **Loop** view.

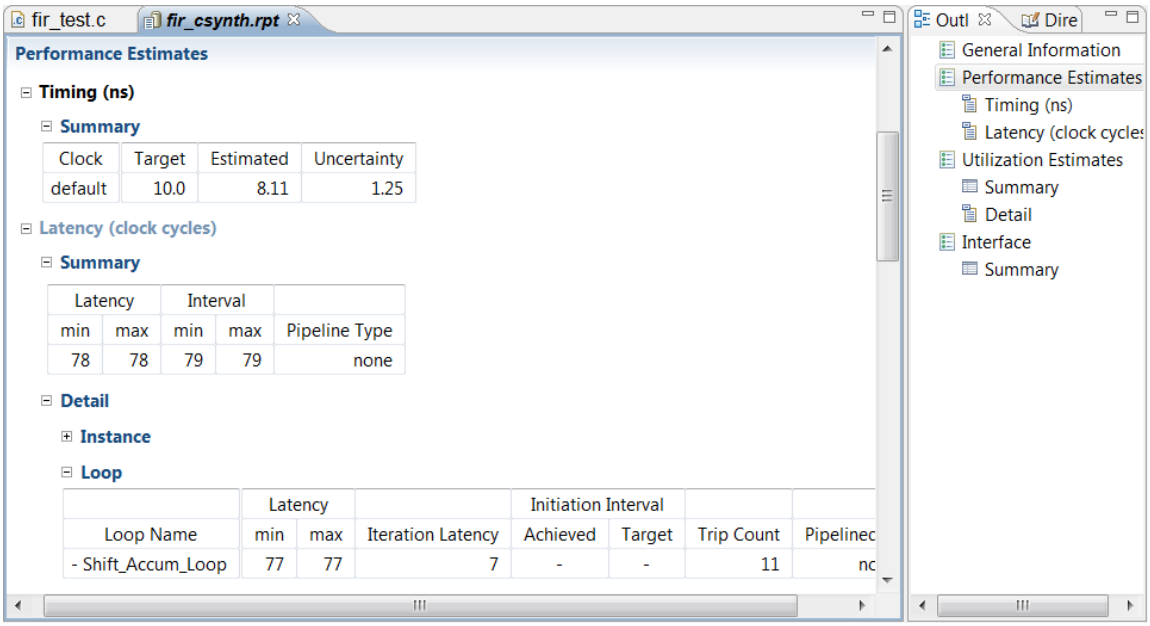


Figure 12: Performance Estimates

The clock period was set to 10ns. The estimated clock period (worst-case delay) is 8.11ns. This includes an uncertainty of 1.25ns so the worst-case delay is actually estimated at 8.11 – 1.25 = 6.86ns. The clock uncertainty ensures there is some timing margin available for the (at this stage) unknown net delays due to place and routing.

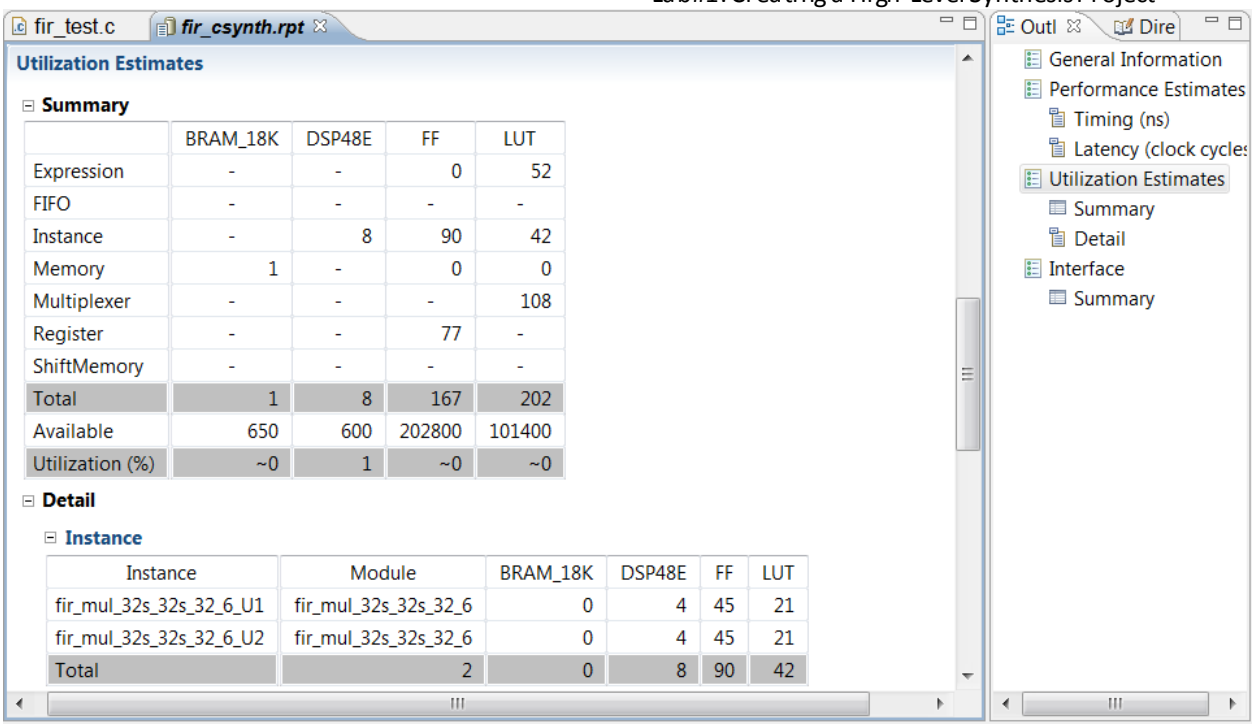
In the **Summary** section, you can see:

- The design has a latency of 78-clock cycle: it will take 78 clocks to output the results.
- The interval is 79 clock cycles: the next set of inputs will be 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) will only start when the current transaction completes.
- The fact that the design is not pipelined is also noted under the pipelined type: no pipelining is performed.

The **Details** section shows:

- There are no sub-blocks in this design. Expanding the Instance section will show no sub-modules in the hierarchy.
- All of the delay is due to the RTL logic synthesized from the loop named **Shift_Accum_Loop**. This logic executes 11 times (Trip Count). Each execution requires 7 clock cycles (Integration Latency) for a total of 77 clock cycles to execute all iterations of the logic synthesized from this loop (Latency).
- The total latency is 1 clock cycle greater than the loop latency. It requires 1 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. Click on Utilization Estimate in the Outline tab (Figure 13).
5. In the **Details** section of the Utilization Estimates, expand the Instance view.



	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	52
FIFO	-	-	-	-
Instance	-	8	90	42
Memory	1	-	0	0
Multiplexer	-	-	-	108
Register	-	-	77	-
ShiftMemory	-	-	-	-
Total	1	8	167	202
Available	650	600	202800	101400
Utilization (%)	~0	1	~0	~0

Instance	Module	BRAM_18K	DSP48E	FF	LUT
fir_mul_32s_32s_32_6_U1	fir_mul_32s_32s_32_6	0	4	45	21
fir_mul_32s_32s_32_6_U2	fir_mul_32s_32s_32_6	0	4	45	21
Total		2	0	8	90	42

Figure 13: Utilization Estimates

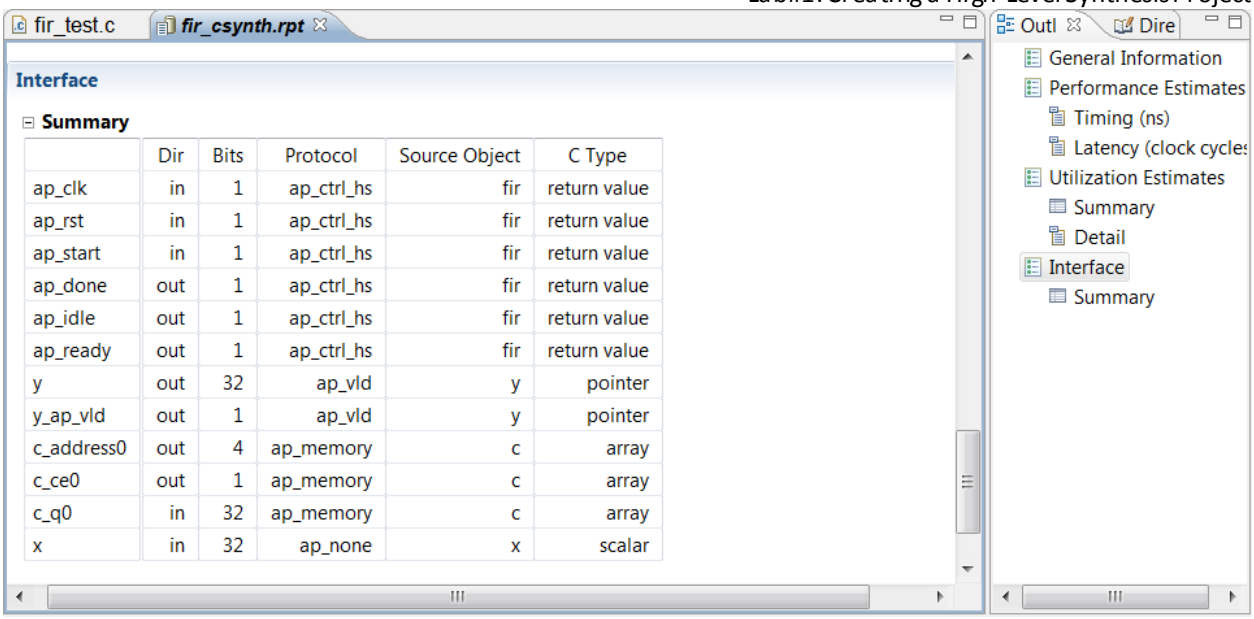
The design is using a single BRAM, 8 DSP48s and approximately 150 flip-flops and 200 LUTs.

At this stage the area numbers are estimates. RTL synthesis may be able to perform additional optimizations and these figures may change after RTL synthesis.

The number of DSP48s seems larger than expected for a FIR filter. All the DSP48s are accounted for by the two instances of multipliers shown in the instance view. Since these multipliers are noted under the **Instance** section and not the **Expressions**, these must have been implemented as pipelined multipliers: standard combinational multipliers are reported in the expressions section.

Further analysis of these results is performed in Lab#3.

6. Click on Interface in the Outline tab (Figure 14).



	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 14: Interface Report

The **Interface** section shows the ports and IO protocols created by interface synthesis:

- The design has a clock and reset port (ap_clk and ap_reset). These are associated with the Source Object fir: the design itself.
- There are additional ports associated with the design as Source Object. Synthesis has automatically added some block level control ports : ap_start, ap_done, ap_idle and ap_ready. These ports are discussed in detail in the tutorial Interface Synthesis (page 65).
- The function output **y** is now a 32-bit data port with an associated output valid signal to indicate when the output data is valid.
- Function input argument **c** (an array) has been implemented as a BRAM interface with a 4-bit output address port, an output CE port and a 32-bit input data port.
- Finally, input argument **x** is simply implemented as a data port with no IO protocol (ap_none).

Later in this tutorial, Lab#3 explains how to optimize the IO 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 on the **Run C/RTL Cosimulation** toolbar button or use the menu **Solution** > **Run C/RTL Cosimulation**.
2. Press OK in the Co-simulation dialog box to execute the RTL simulation.

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

When RTL co-simulation completes the report opens automatically in the Information pane and the Console displays the message shown in Figure 15. This is the exact same message produced at the end of C simulation.

- The C test bench is used to generate 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 is used to verify the results are correct.

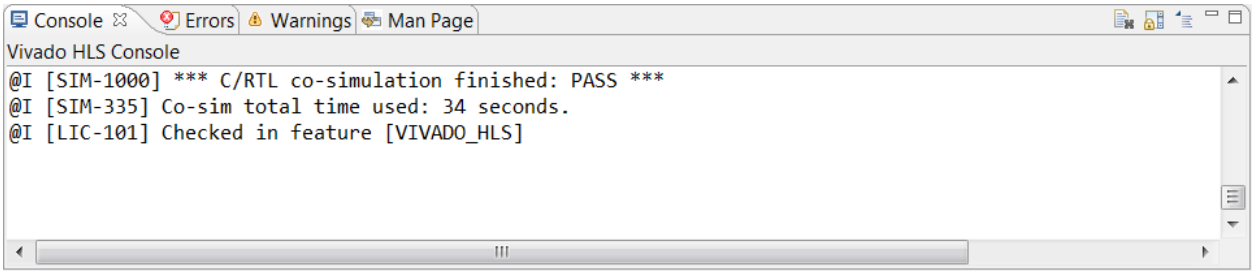


Figure 15: RTL Verification Results

More details on RTL Verification is available in the tutorial RTL Verification (page 167).

Step 5: IP creation

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

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

The IP packaging process with create an package for the Vivado IP Catalog. Other options, available from the drop-down menu are to create IP packages for System Generator for DSP or a pcore for Xilinx Platform Studio.

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

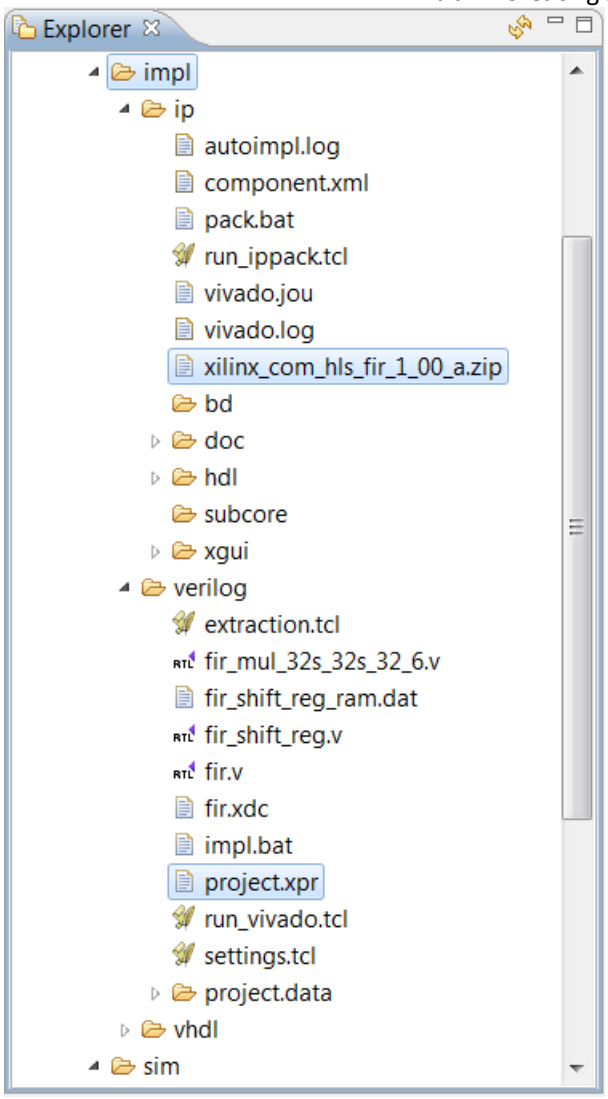


Figure 16: RTL Verification Results

Also note in Figure 16, 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.

IMPORTANT: *In this Vivado project, the HLS design is the top-level. This project is provided as an additional means of analyzing the design. The recommended approach is to add the IP package to the Vivado IP catalog and add it as IP to the design which will use the HLS design.*

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

At this stage, leave the Vivado HLS GUI open. We will return to this in the next lab exercise. This completes this lab exercise on using the Vivado High-Level Synthesis GUI.

Lab#2: Using the Tcl Command Interface

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.
    a. On Windows use **Start** > **All Programs** > **Xilinx Design Tools** > **Vivado 2013.1** > **Vivado HLS** > **Vivado HLS 2013.1 Command Prompt (**Figure 17).
    b. On Linux, open a new shell.

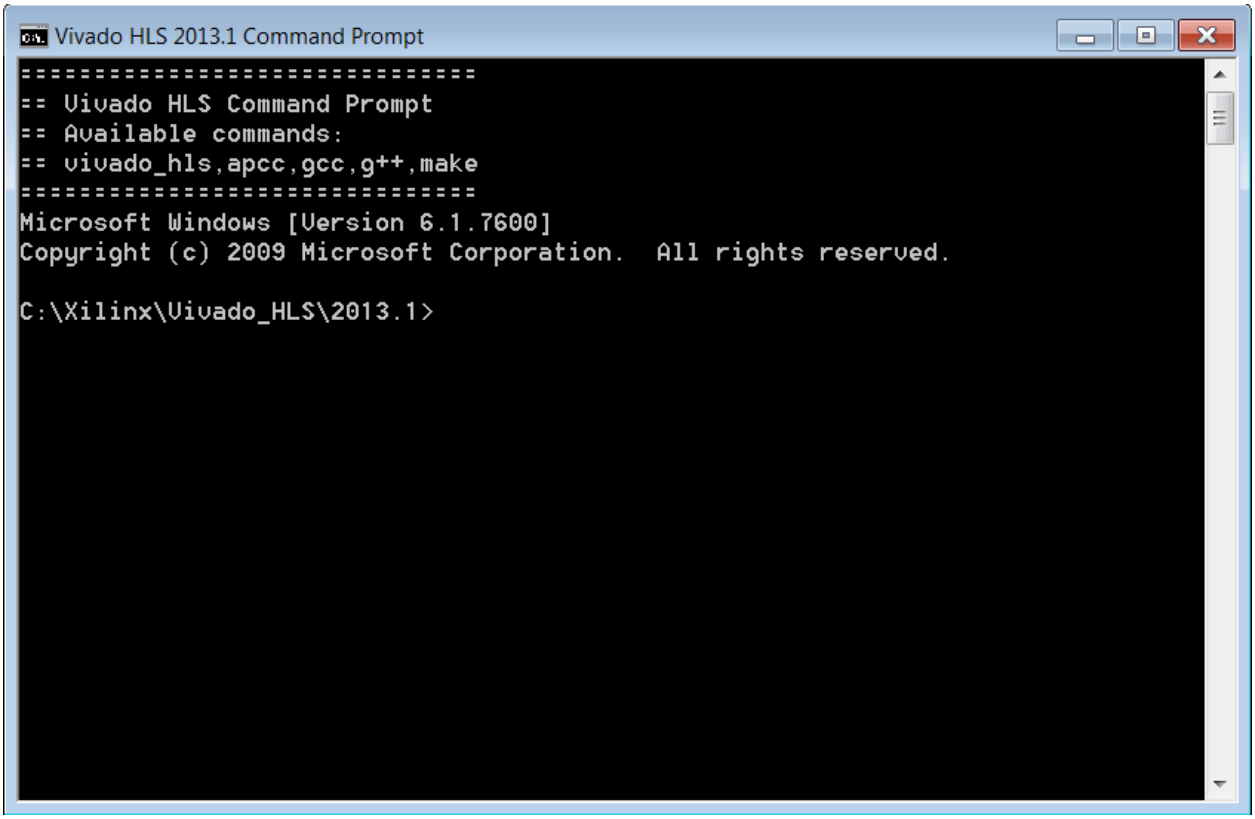


Figure 17: The Vivado HLS Command Prompt

When a Vivado HLS project is created, Tcl files are automatically save in the project hierarchy. In the GUI still open from Lab#1, a review of the project in the GUI shows two Tcl files in the project hierarchy (18).

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

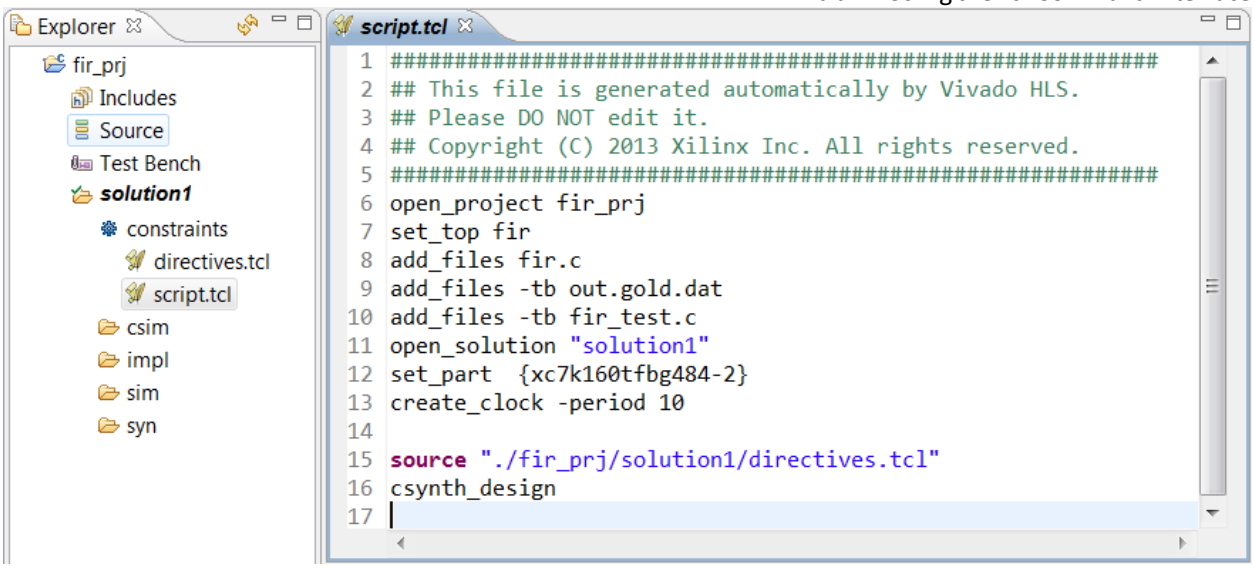


Figure 18: The Vivado HLS Project Tcl Files

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

In this lab exercise, the **script.tcl** from Lab#1 will be used 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 19) to create a new Tcl file for Lab#2.
    a. Change directory to the Introduction tutorial directory C:\Vivado_HLS_Tutorial\Introduction.
    b. Use the command **cp lab1\fir_prj\solution1\script.tcl lab2\run_hls.tcl** to copy the existing Tcl file to Lab#2. (The Windows command prompt supports auto-completion using the tab-key: press the tab key multiple time to see new selections).
    c. Use **cd lab2** to change into the lab2 directory.

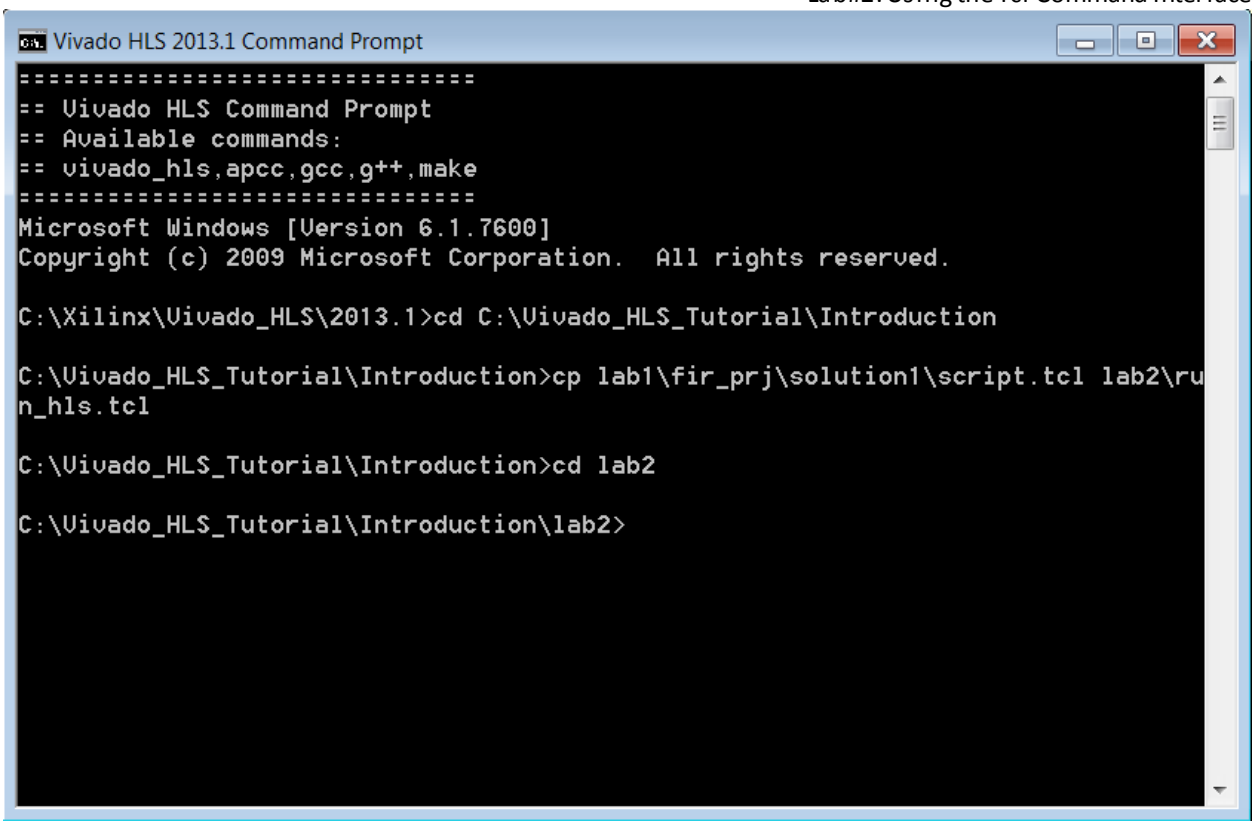


Figure 19: Copying the Lab#1 Tcl file to Lab#2

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 20.

    d. Add a –reset option to the open_project command. Since Tcl files are typically run repeatedly on the same project, it is desirable to over-write any existing project information.

    e. Add a –reset option to the open_solution command – to remove any existing solution information when the Tcl file is re-run on the same solution.

    f. Add the command csim_design to run C simulation.

    g. Delete the source command. If a previous project had any directives you wish to re-use, the directives.tcl file from that project can be copied to a local path or the directives can be copied directly into this file.

    h. Add the command cosim_design to execute RTL verification.

    i. Add the command export_design to create the IP package.

    j. Add the exit command.

    k. Save the file

Open Files

run_hls.tcl

```
1  ############################################################
2  ## This file is generated automatically by Vivado HLS.
3  ## Please DO NOT edit it.
4  ## Copyright (C) 2013 Xilinx Inc. All rights reserved.
5  ############################################################
6
7  # Step 3a: add -reset to the open_project command
8  open_project -reset fir_prj
9  set_top fir
10 add_files fir.c
11 add_files -tb out.gold.dat
12 add_files -tb fir_test.c
13
14 #Step 3b: add -reset to the open_solution command
15 open_solution -reset "solution1"
16 set_part  {xc7k160tfbg484-2}
17 create_clock -period 10
18
19 # step 3c: run C simulation
20 csim_design
21 # step 3d: remove this line.
22 #source "./fir_prj/solution1/directives.tcl"
23 csynth_design
24 # step 3e: run RTL verification
25 cosim_design
26 # step 3f: create the IP package
27 export_design
28 # step 3g: exit Vivado HLS
29 exit
30
```

Figure 20: Updated run_hls.tcl file for Lab#2

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

6. In the **Vivado HLS Command Prompt** window, type **vivado_hls –f run_hls.tcl.**

Vivado HLS will execute all the steps covered in lab1. When finished, the results will be 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, the RTL from the impl directory must be used.*

For designs using floating-point operators or AXI4 interfaces, the RTL files in the syn directory are only the output from synthesis: additional processing is performed during export_design before this RTL can be used in other design tools.

Lab#3: Using Solutions for Design Optimization

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

Step 1: Creating a New Project

1. Open the Vivado HLS Command Prompt.
    a. On Windows use **Start** > **All Programs** > **Xilinx Design Tools** > **Vivado 2013.1** > **Vivado HLS** > **Vivado HLS 2013.1 Command Prompt**
    b. On Linux, open a new shell.
2. Change to the Lab#3 directory: **cd C:\Vivado_HLS_Tutorial\Introduction\lab3**
3. In the command prompt window, type **vivado_hls –f run_hls.tcl** to setup the project.
4. In the command prompt window, type **vivado_hls –p fir_prj** to open the project in the Vivado HLS GUI.

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

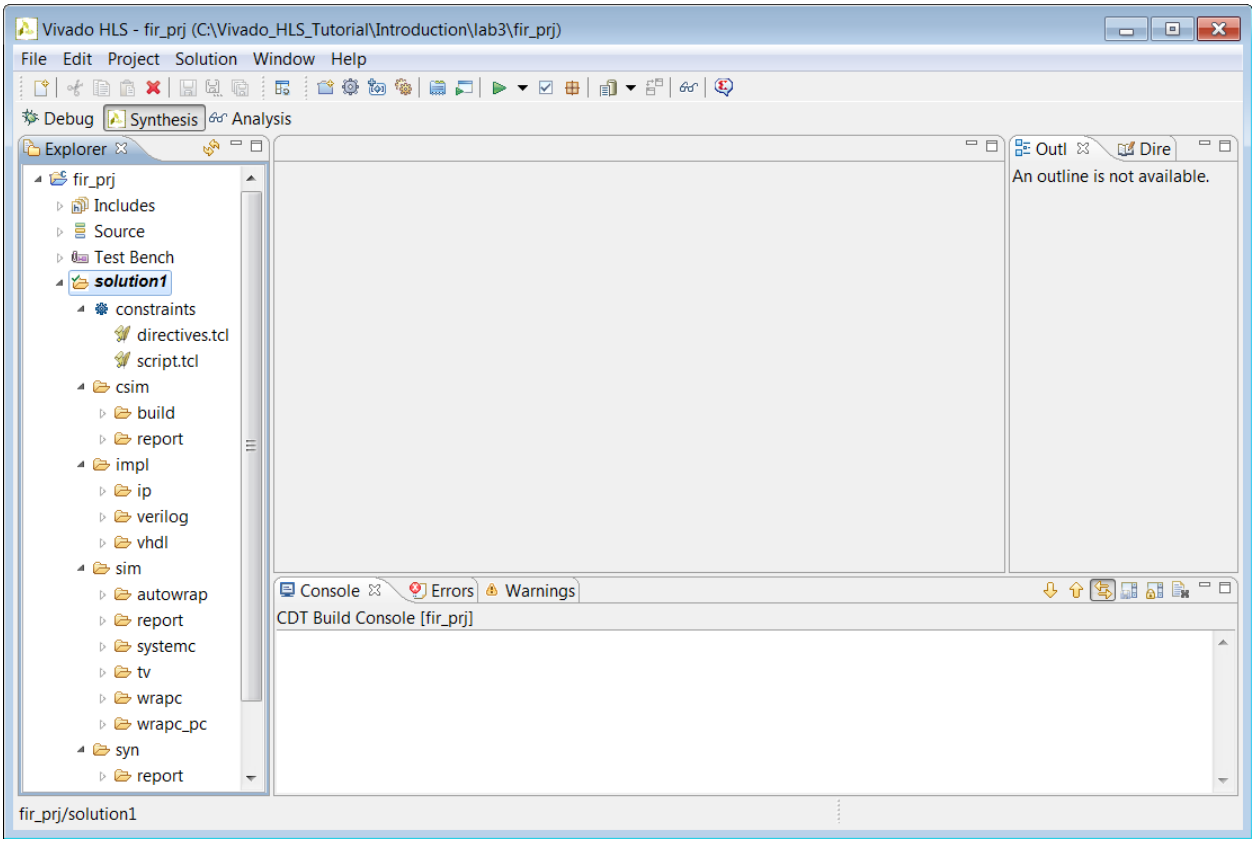


Figure 21: Introduction Lab#3 Initial Solution

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

- Create a version of the design with the smallest area
- 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 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.

Step 2: Optimize the IO Interfaces

Since the design specification includes IO protocols, the first optimization to perform is to create the correct IO protocol and ports. The type of IO protocol selected may affect what design optimizations are possible so if there is an IO protocol requirement, the IO protocol should be fixed as early as possible in the design cycle.

The IO protocol for this design was reviewed in Lab#1 (Figure 14) and the synthesis report can be reviewed again by navigating to the report folder inside the solution1\syn folder. The IO requirements are:

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

Port C already is a single port RAM access. However, if this is not explicitly specified High-Level Synthesis may use a dual-port interface if it results in a design with higher throughput. The IO protocol requirement to use a single-port RAM should be explicitly added to the design.

Input port X is by default simply a 32-bit data port. This can be implemented as an input data port with an associated data valid signal by specifying the IO protocol ap_vld.

Output port Y already has an associated output valid signal. This is the default for pointer arguments. There is no need to specify an explicit port protocol for this port but it will be added, just to be explicit.

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

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

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

To add optimization directives to define the desired IO interfaces to the solution, perform the following steps.

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

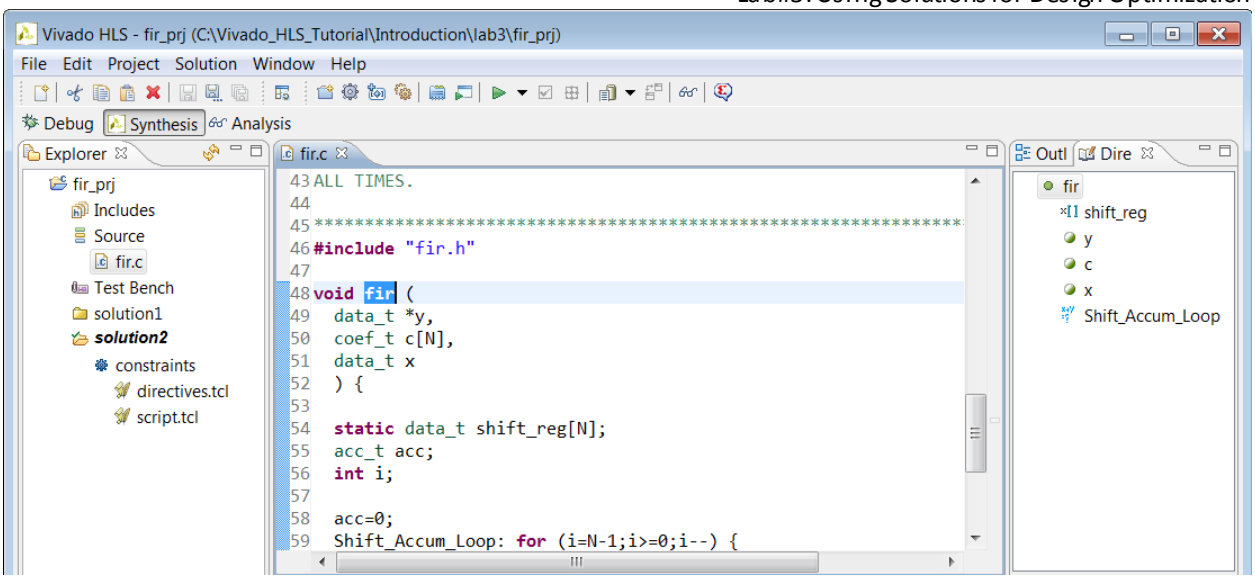


Figure 22: Opening the Directives Tab

The Directives tab, shown on the right-hand side of Figure 22, lists all of the objects in the design that can be optimized. The Directives tab is used to add optimization directives to the design. The Directives tab can only when viewed when the source code is open in the Information pane.

Apply the optimization directives to the design.

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

This specifies that array c is to be implemented using a single-port BRAM resource. Since array c is in the function argument list, and hence is outside the function, this causes a set of data ports to be created to access a single-port BRAM outside the RTL implementation.

Since IO protocols are unlikely to change, it is useful to add these optimization directives to the source code as pragmas: this ensures the correct IO 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**.

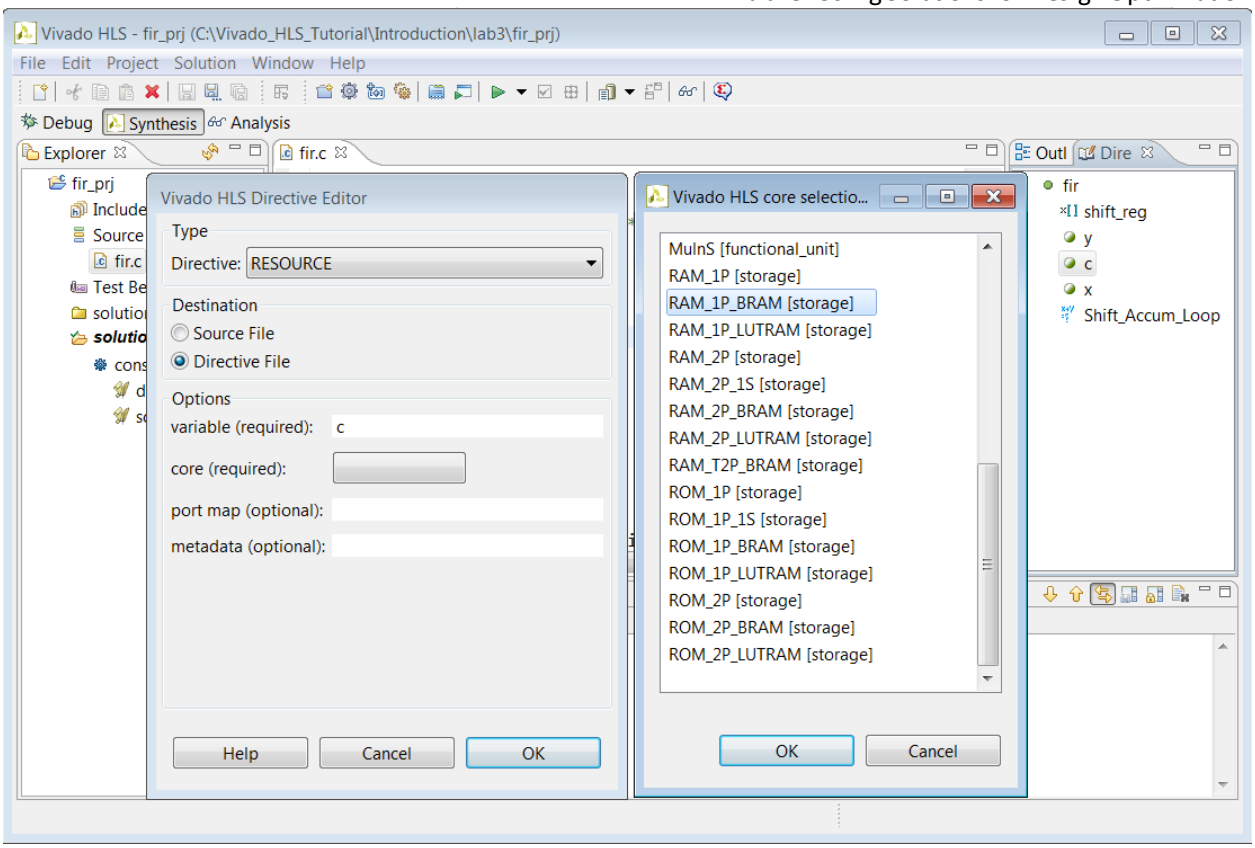


Figure 23: Adding a Resource Directive

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

When complete, the source code and the Directive should be as shown in Figure 24. Select any incorrect directive, use the mouse to right-click, and modify it.

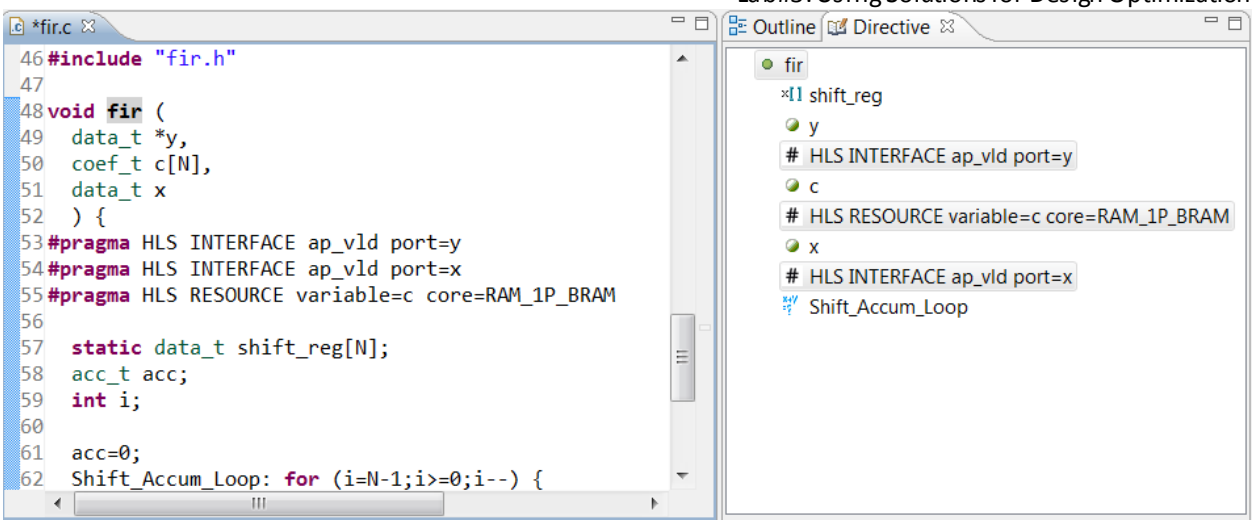
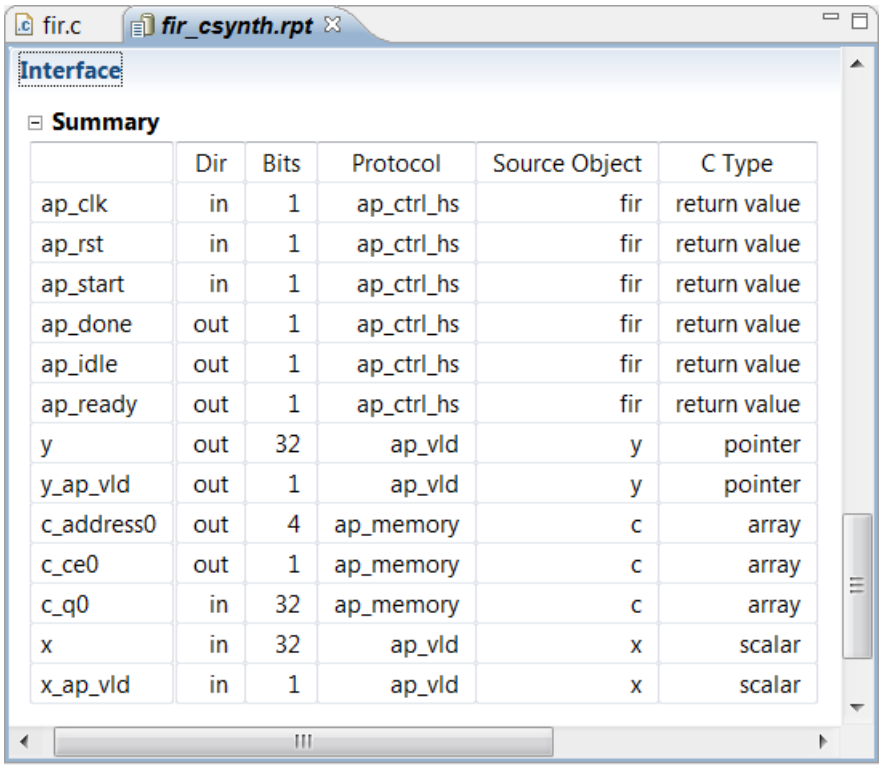


Figure 24: IO Directives for solution2

14. Press the Run C Synthesis toolbar button or menu Solution > Run C Synthesis to synthesize the design
15. When prompted, press 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.

Use the Outline tab to view the Interface results or simply scroll down to the bottom of the report file. Figure 25 shows the ports now have the correct IO protocols.



	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 25: IO Protocols for solution2

Step 3: Analyze the Results

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

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

1. Press the **Analysis** perspective button.
2. Click on the **Shift_Accum_Loop** in the **Performance** window to expand it.

The red-dotted line in Figure 26 will be used shortly in an explanation: it is not part of the view.

The tutorial Design Analysis (page 111) provides a more complete understanding of the Analysis perspective but the following explains how we can understand what is required to create the smallest and fastest RTL design from this source code.

The Performance pane view shows the operations in this module of the RTL hierarchy down the left-hand column.

The top row lists the control states in the design. Control states are the internal states used by High-Level Synthesis 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.

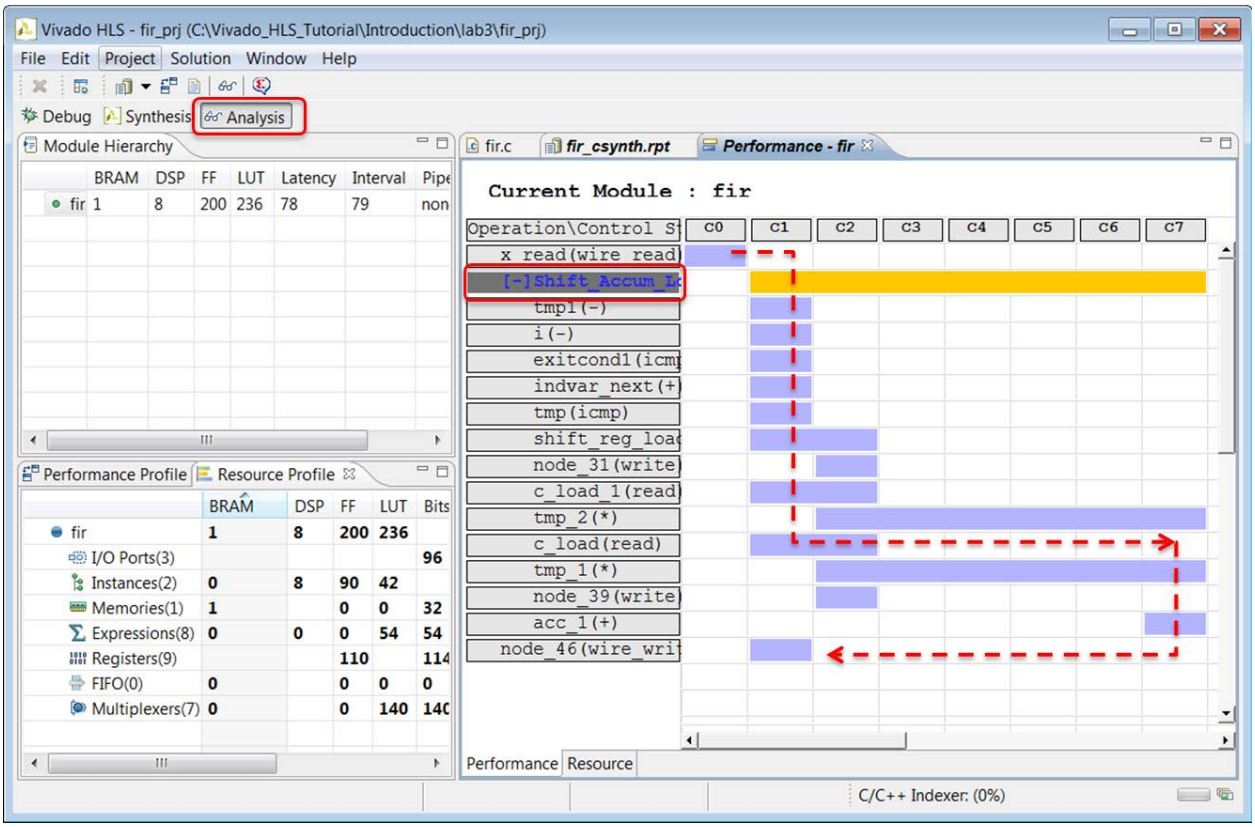


Figure 26: Solution2 Analysis Perspective: Performance

The explanation presented here will follow the path of the dotted red in Figure 26. Some of the objects here can be directly correlated with the C source code: select the object and right-click with the mouse 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 can be expanded or collapsed. Holding the cursor over the yellow loop body in this view will show 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 BRAM synthesized from array shift_reg (one cycle to generate the address, one cycle to read the data).
- There are memory reads on the c port.
- Two multiplication operations each take 6 cycles to complete.
- The for-loop is executed 7 times.
- At the end of the final iteration, the loop exits in state c1 and the write to port y occurs.

The analysis perspective can also be used to analyze the resourced used in the design.

3. Click on the Resource view as shown in Figure 27.
4. Expand all of the resource groups (also shown in Figure 27).

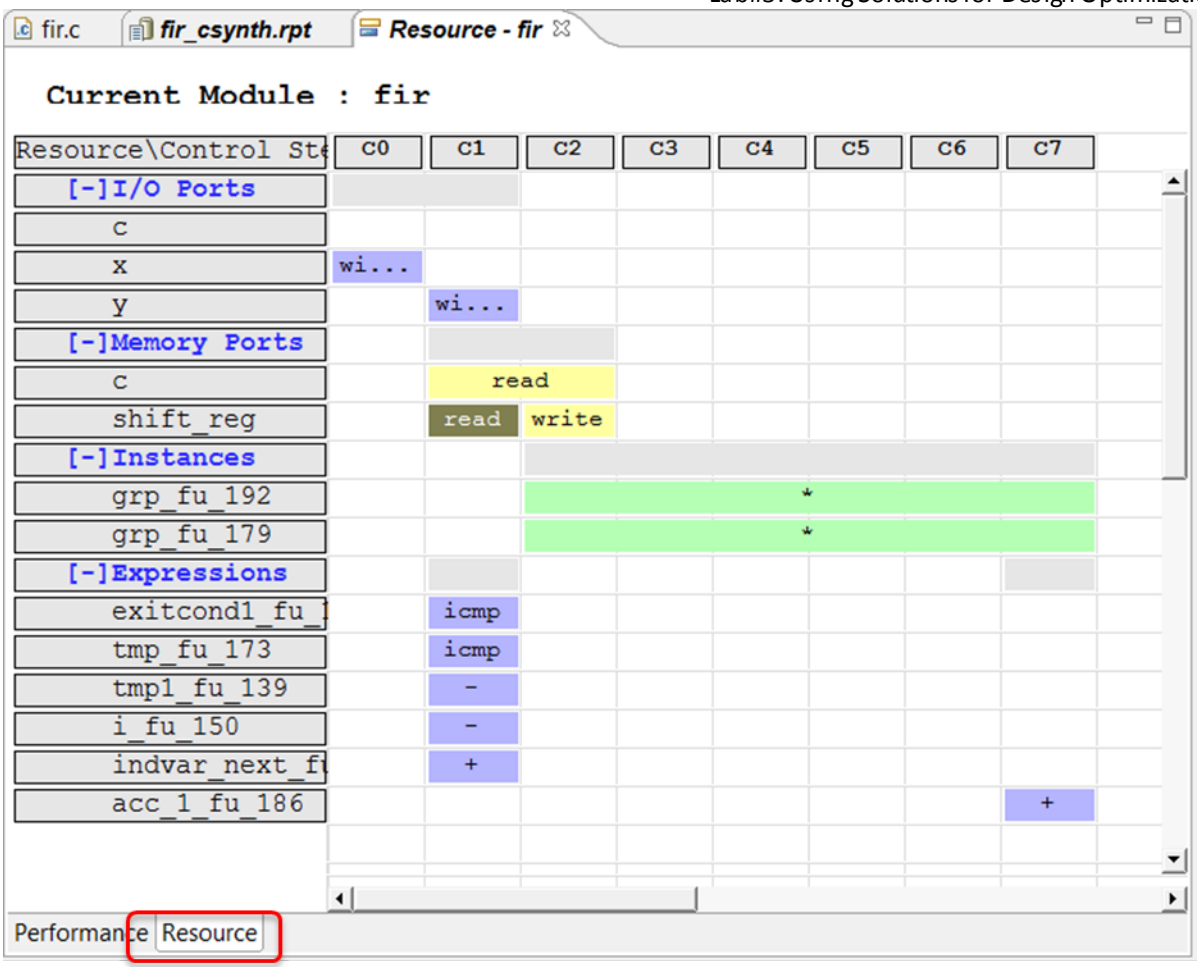


Figure 27: Solution2 Analysis Perspective: Resource

Figure 27 shows:

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

The insight gained by analyzing the design can be used to optimize the designs.

Before concluding the analysis, it is worth commenting on the multi-cycle multiplication operations, which require multiple DSP48s to implement. The source code uses an int data-type. This is a 32-bit data-type. This is the cause of the large multipliers: a DSP48 multiplier is 18-bit.

The tutorial Arbitrary Precision Types (page 99)shows how arbitrary precision types can be used to create designs with more suitable data-types for hardware, by allowing data-types of any arbitrary bit-size to be defined: more than the standard C/C++ 8, 16, 32 or 64-bit types.

Step 4: Optimize for the smallest Area.

1. In solution2, **press** the **Synthesis Perspective** button to return to the synthesis perspective.
2. Click the **New Solution** button to create a new solution.
3. Leave the solution name as **solution3**. The solution names default to solution1, 2, 3, etc. but can be named anything.
4. Press Finish to create **solution3**.
5. In the Project menu, select **Close Inactive Solution Tabs** to close any existing tabs from previous solutions.

This design captures all the logic in a for-loop. Using a loop is one of the best techniques to minimize the area: the logic, which represents the loop body, is synthesized once then executed multiple times.

The design is using only a single BRAM, however it is using 2 multipliers. The design is using a separate multiplier for the if-branch and the else-branch. (Selecting the operations in the Performance pane and cross-referencing to the C code highlights this).

In some cases, if Vivado HLS determines an additional multiplexor in front of an operation may introduce timing issues, it will be conservative and not share the operators. Since this multiplier is an expensive resource in terms of area, and the timing report looks acceptable, we can force sharing to occur.

To force sharing of the multipliers, use a configuration setting as follows.

6. Use the Solution Settings toolbar button or the menu **Solution > Solution Settings** to open the solution settings menu.
7. Select **General** on the left-hand side menu.
8. Click **Add** to open the list of configuration settings.
9. Select **config_bind** from the drop-down menu.
10. Specify **mul** in the **min_op** (minimize operator) field, as shown in Figure 28.
11. Click **OK** to set the configuration.
12. Click **OK** again to close the Solution Settings window.

The config_bind command controls the binding phase, where operators inferred from the code are bound to cores from the library. The min_op option tells Vivado HLS to minimize the number of the specified operators (mul operations, in this case) and overrides any mux delay estimation.

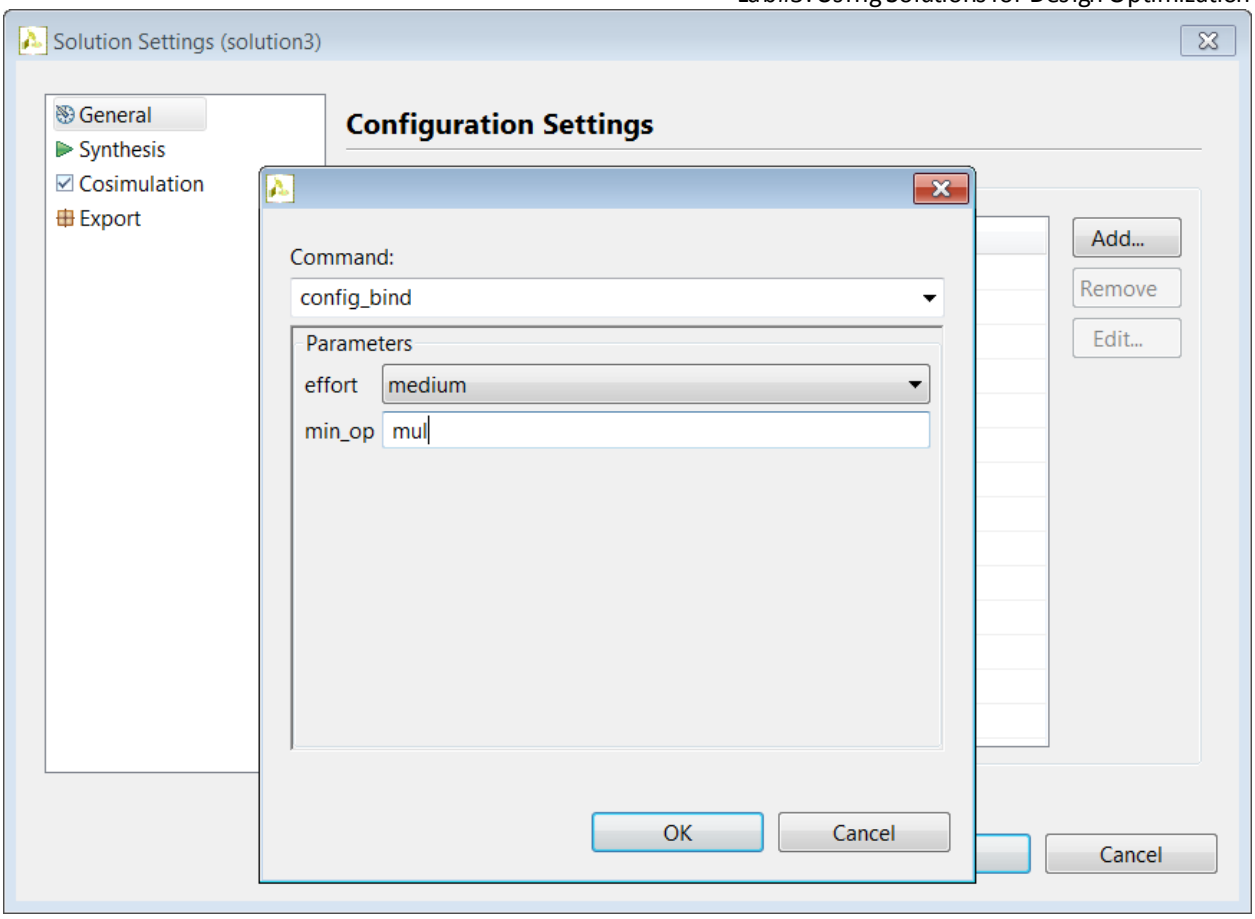


Figure 28: Solution3 Configuration Settings

13. Click the **Synthesis** button to synthesize the design.

When synthesis completes, the synthesis report opens showing that the configuration command was successful and only a single multiplier is now used in the design.

This design uses the same hardware resources to implement every iteration of the loop. This is the smallest number of resources that this FIR filter can be implemented with: a single multiplier, a single BRAM, some flip-flops and LUTs.

Step 5: Optimize for the Highest Throughput (lowest interval)

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

- The for-loop. This ensures each iteration of the loop is executed sequentially. The for-loop can be unrolled to allow all operations to occur in parallel.
- The BRAM used for shift_reg. Since variable shift_reg is an array in the C source code, it is implemented as a BRAM by default. However, this prevents it being implemented as a shift-register. This BRAM should be partitioned into individual registers.

Begin by creating a new solution.

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

2. Leave the solution name as solution4.
3. **Unselect** both the **Copy existing directives from solution** check boxes. This time, the directives added to solution3 are not required: new directives will be added to this solution. (Remember, the IO directives are embedded in the source code).
4. Press **Finish** to create the new solution.
5. In the Project menu, select **Close Inactive Solution Tabs** to close any existing tabs from previous solutions.

The following steps, summarized in Figure 29, explain how to unroll the loop.

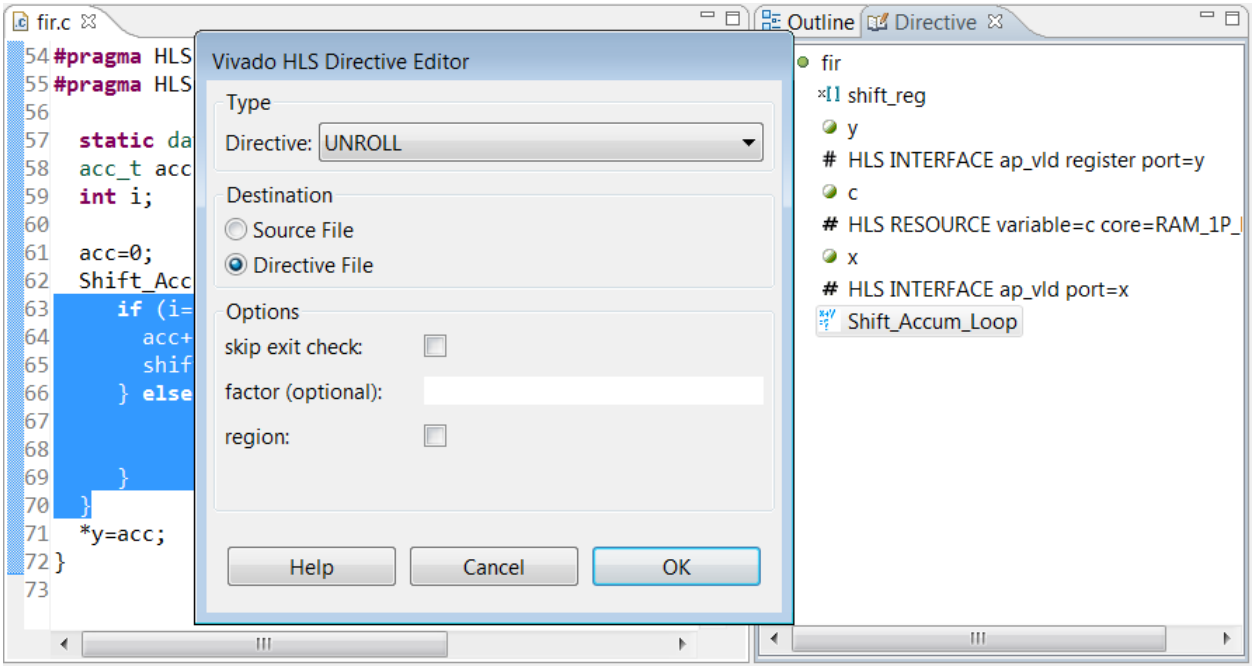


Figure 29: Unrolling FOR Loop

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

Leave the Destination as the Directive File. When optimizing a design, it is often the case that multiple iterations of optimizations will be performed until the final design optimizations are determined.

By adding the optimizations to the directives file, we 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. If the optimizations were added as pragmas in the code, they would be automatically carried forward to new solutions and they code would need to be changed if you were 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.

9. Select **OK** to apply the directive.

Apply the directive to partition the array into individual elements.

10. In the Directive tab, select array **shift_reg**.
11. Right-click and select **Insert Directives**.
12. Select **Partition** from the Directive drop-down menu.
13. Specify the type as **complete**.
14. 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 solution4 is now as shown in Figure 30.

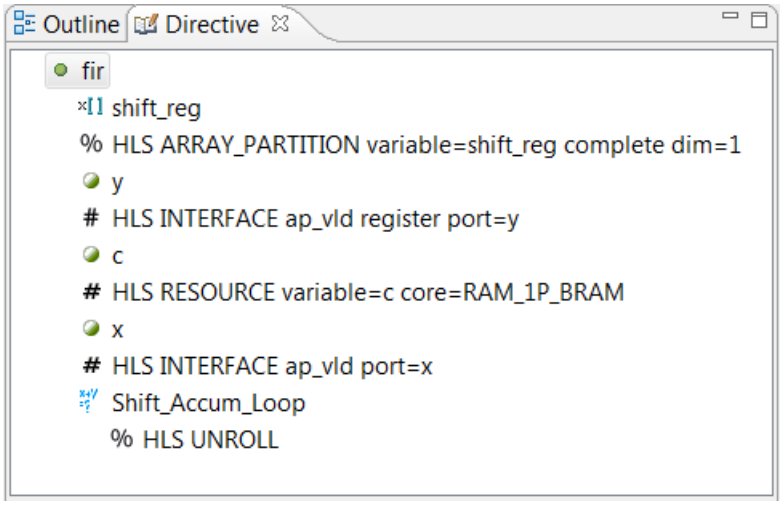


Figure 30: Solution4 Directives

In Figure 30, notice the directives applied in solution2 as pragmas have a different annotation (#HLS) than those just applied and saved to the directives file (%HLS). The newly added directives can be viewed in the Tcl file.

15. In the Explorer pane, expand the Constraint folder in Solution4 as shown in Figure 31.
16. Double-click on the solution4 directives.tcl file to open it in the Information pane.

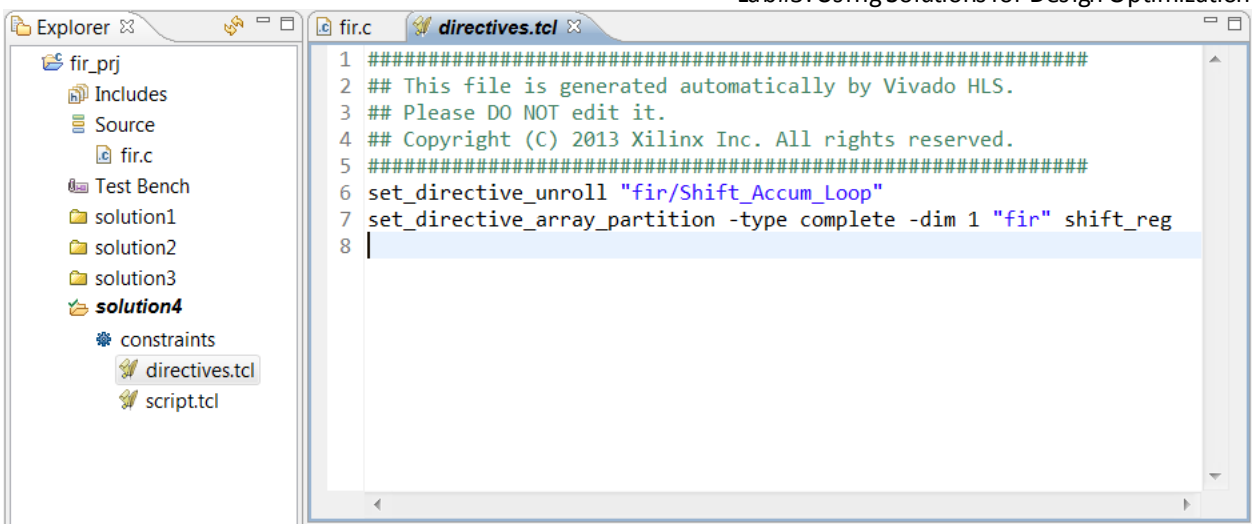


Figure 31: Solution4 directives.tcl File

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

When synthesis completes, the synthesis report automatically opens.

Now, compare the results of the different solutions.

18. Use the **Compare Reports** toolbar button or menu Project > Compare Reports.
19. Add **solution2, solution3** and **solution4** to the comparison.
20. Click **OK**.

Figure 32 shows the comparison of the reports. Solution3 is the smallest of the implementations and solution4 is the highest throughout as the interval is only 19 – it starts to process a new set of inputs every 19-clock cycles.

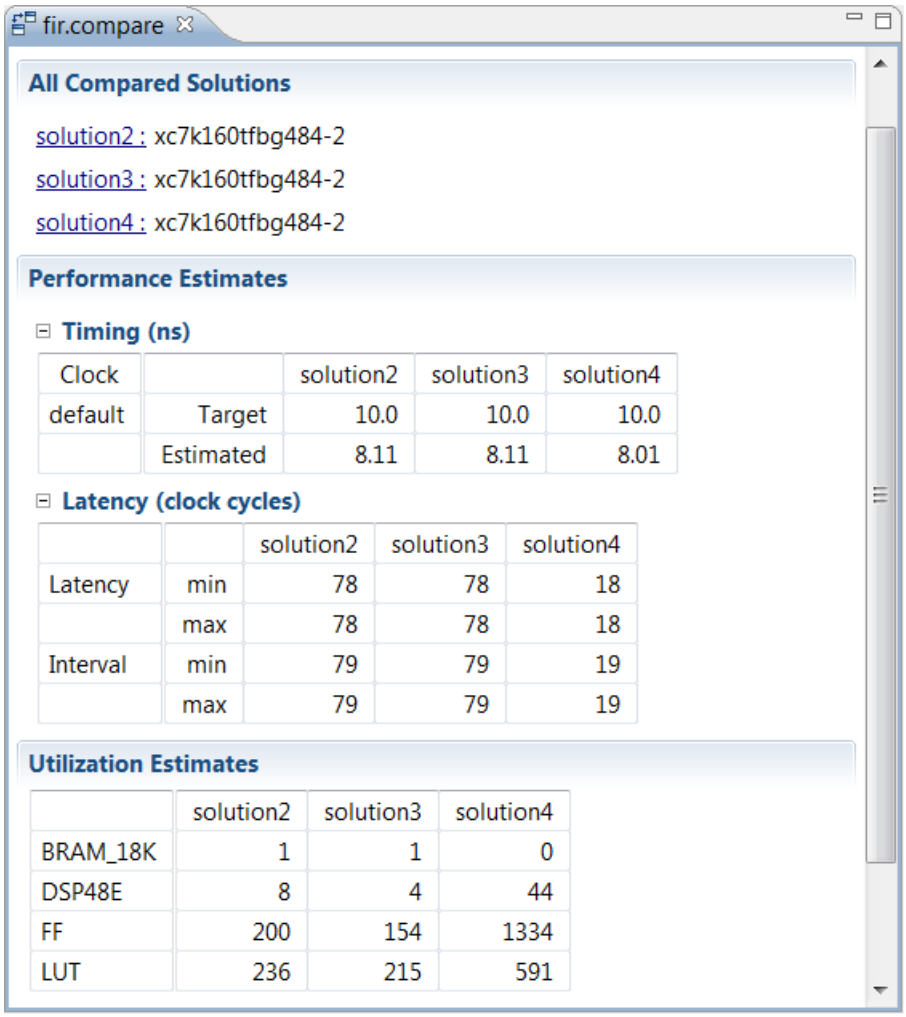


Figure 32: Solution Comparisons

Additional optimizations can be performed on this design. Pipelining could be used to further improve the throughput and lower the interval. The tutorial Design Optimization (page 143) provides details on using pipelining to improve the interval.

As mentioned earlier in this tutorial, the code itself could be modified to use arbitrary precision types, for example if the data types are not required to be 32-bit int types, bit-accurate types (for example, 6-bit, 14-bit or 22-bit types) could be used provided they satisfy the required accuracy. More details on using arbitrary precision type can be found in the tutorial Arbitrary Precision Types (page 65).

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.

- Learned how to create and use a Tcl file to run Vivado HLS.
- Create new solutions, add optimization directives and compare the result of different solutions.

C Validation

Overview

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

This tutorial consists of three lab exercises.

Lab1

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

Lab2

Validate and debug a C design using arbitrary precision C types.

Lab3

Validate and debug a design using arbitrary precision C++ types.

Tutorial Design Description

The tutorial design file can be download from the Xilinx website. Refer to the information in Obtaining the Tutorial Designs.

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

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

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

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

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

Step 1: Create and Open the Project

1. Open the Vivado HLS Command Prompt.
    a. On Windows use **Start** > **All Programs** > **Xilinx Design Tools** > **Vivado 2013.1** > **Vivado HLS** > **Vivado HLS 2013.1 Command Prompt (**Figure 33).
    b. On Linux, open a new shell.

Figure 33 Vivado HLS Command Prompt

2. Using the command prompt window (34), change 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 34.

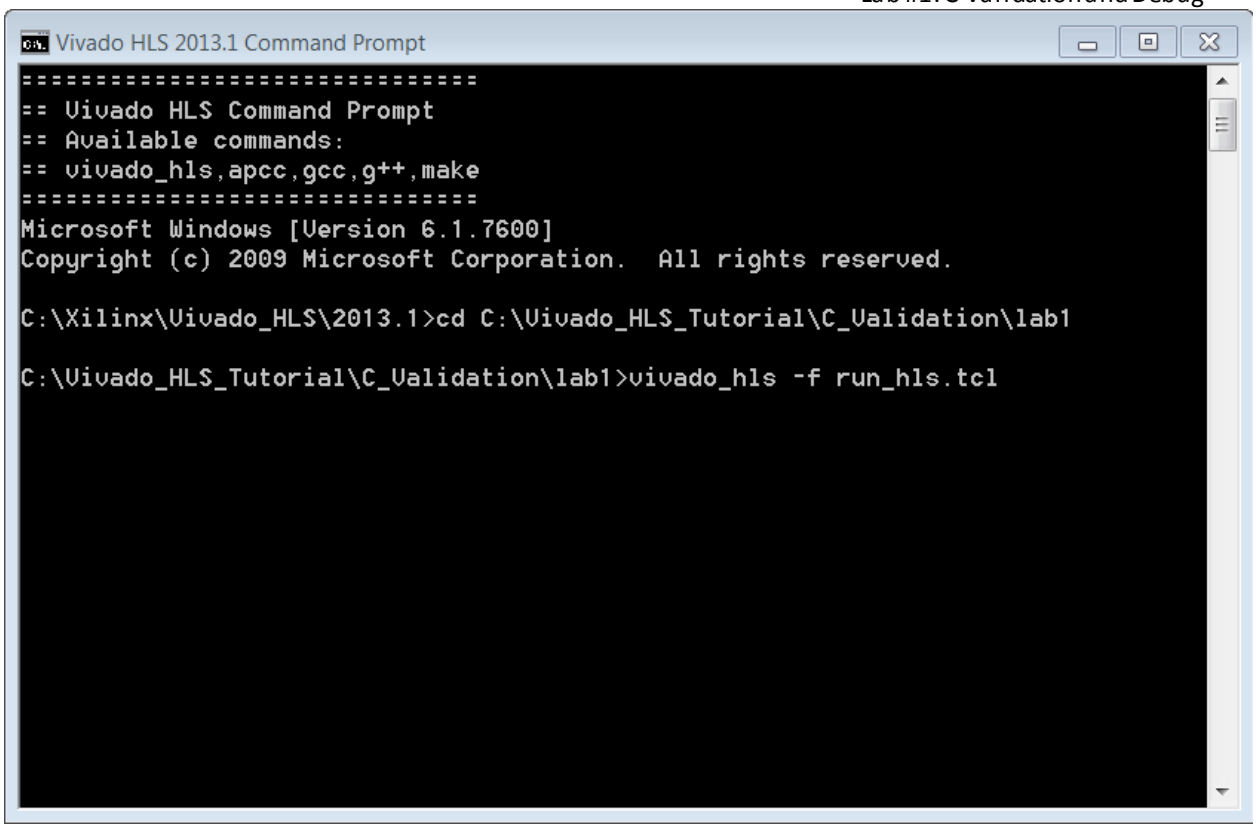


Figure 34 Setup the Tutorial Project

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 35.

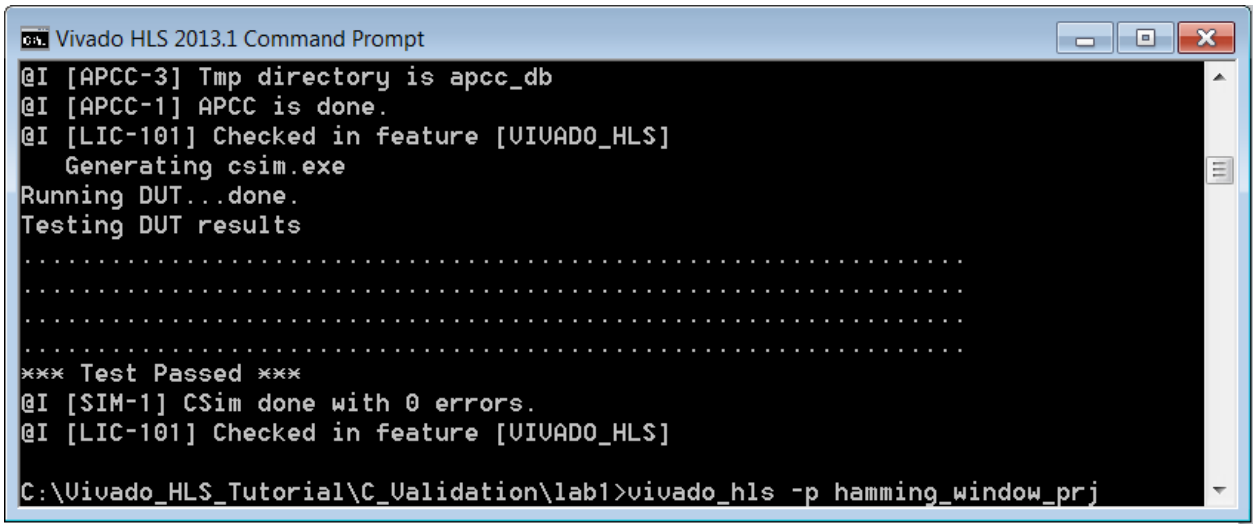


Figure 35 Initial Project for C Validation Lab#1

Step 2: Review Test Bench and Run C Simulation

1. Open the C test bench for review by double-clicking on hamming_window.c in the Test Bench folder Figure 36.

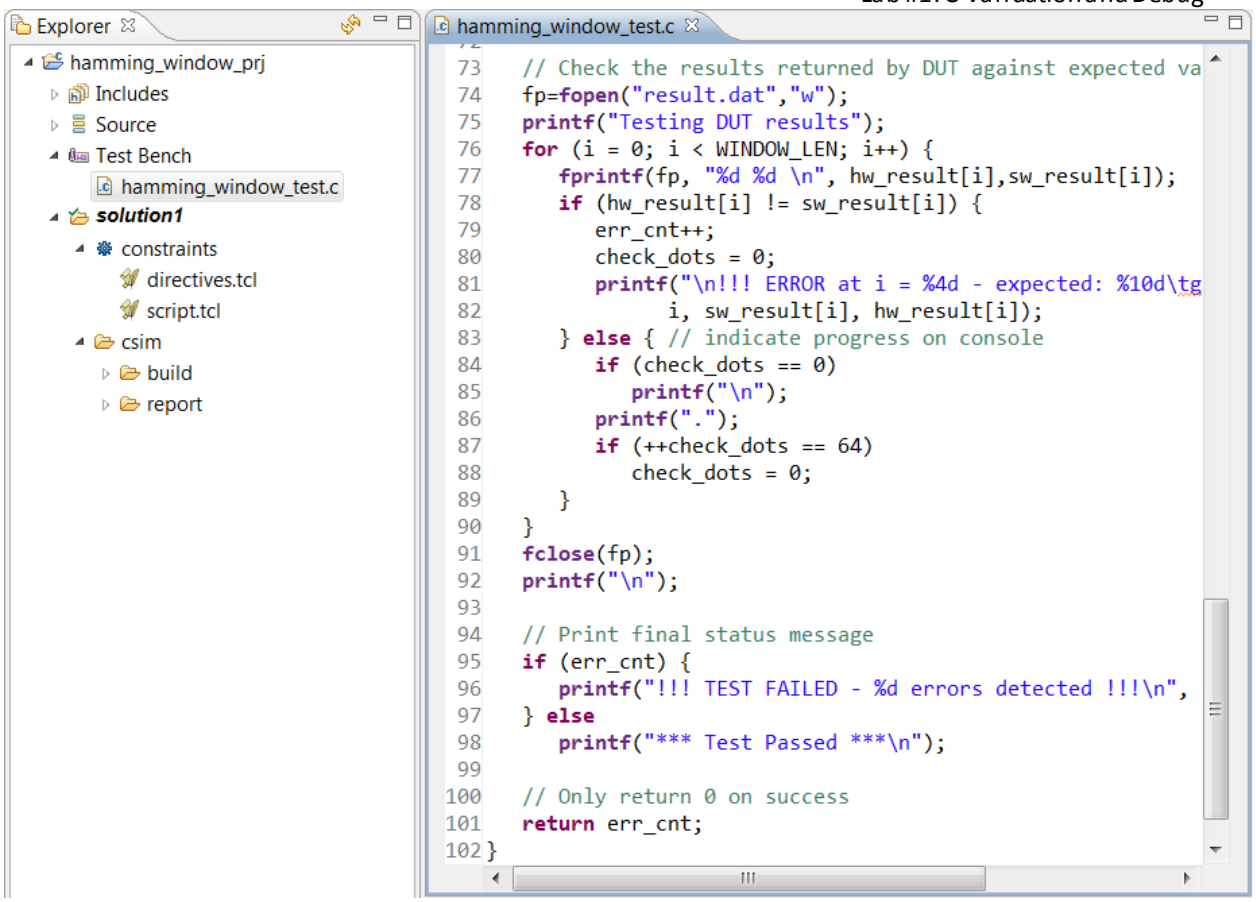


Figure 36 C Test Bench for C Validation Lab#1

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

- The test bench creates a set of expected results and stores them in array sw_result. These will be used to confirm the function is correct. These could have been read from an input file, in which case the input file must be added to the project as a test bench file.
- The DUT is called to generate results which are stored in array hw_result. Since this array is used by the function which will be synthesized, later in the design flow this array will hold the results generated by the RTL.
- The actual and expected results are compared. If the comparison fails, the value of value 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 is used the RTL verification to automatically check and verify the RTL results.

The C code and test bench can be executed to confirm it's working as expected.

2. Use the Run C Simulation toolbar button or the menu Project > Run C simulation to open the C Simulation Dialog box in Figure 37.

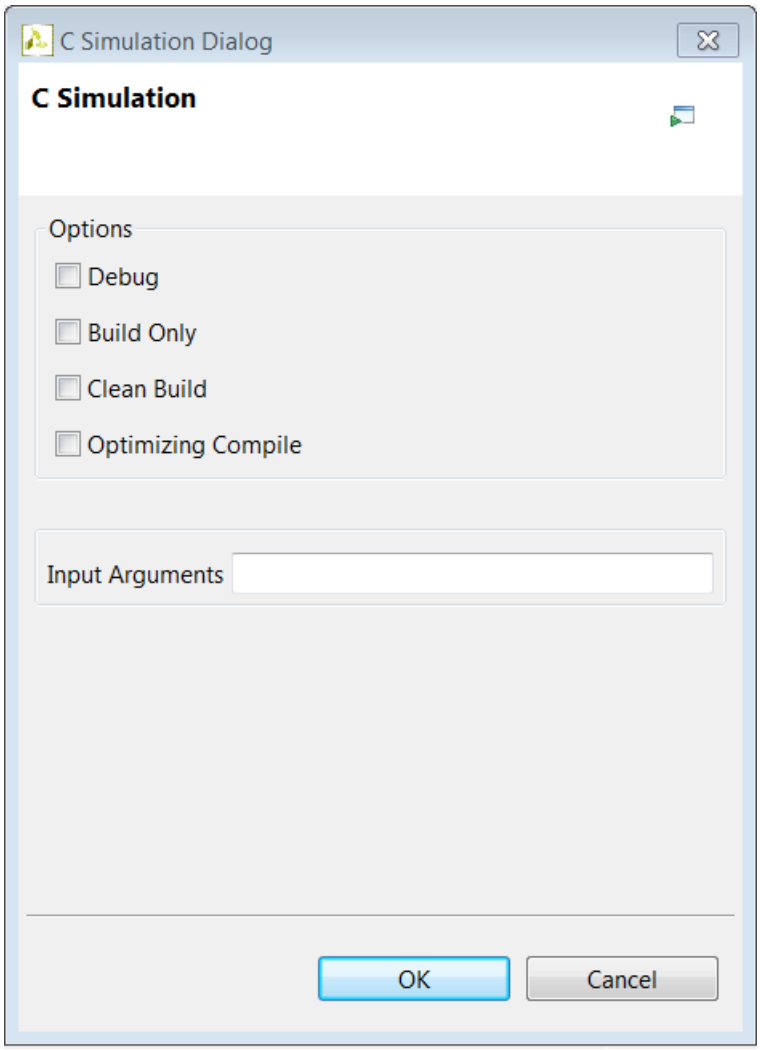


Figure 37 Run C Simulation Dialog box

3. Select OK to run the C simulation.

As shown in Figure 38, the following actions are performed when C simulation executes.

- The simulation output is shown in the Console window.
- Any print statements in the C code are echoed to the Console window. This example shows the simulation passed correctly.
- The C simulation is executed in the solution directory csim and any output from the C simulation can be found in folder build: here we can see the output file result.dat written by the fprintf command.

The C simulation is not executed in the project directory. For this reason, any data files used, such as input data read by the test bench, must be added to the project as a C test bench file (so it can be copied to csim/build directory when the simulation runs).

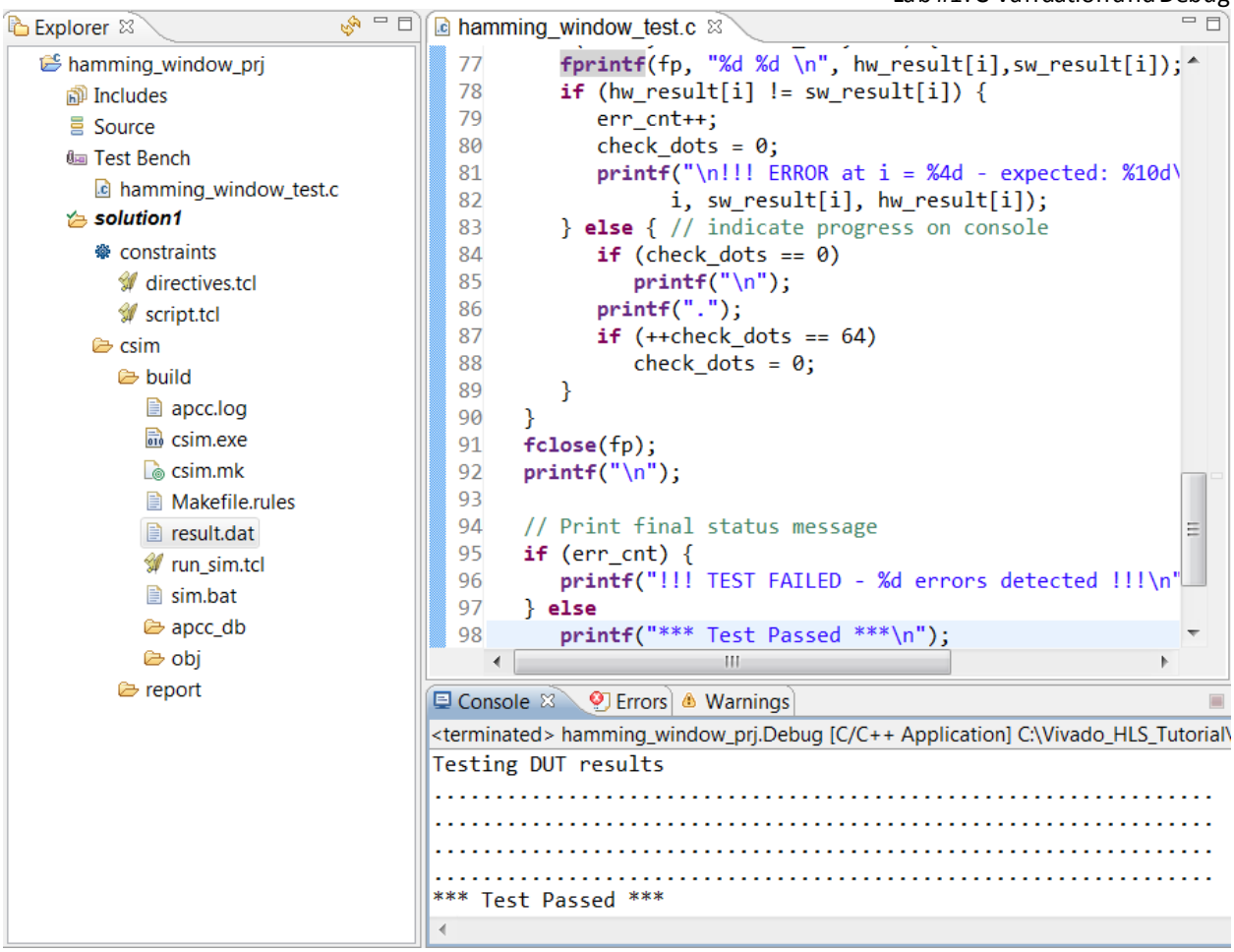


Figure 38 C Simulation Results

Step 3: Run the C Debugger

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

1. Use the **Run C Simulation** toolbar button or the menu **Project** > **Run C simulation** to open the **C Simulation Dialog** box.
2. Select the **Debug** option as shown in Figure 39.
3. Press **OK** to run the simulation.

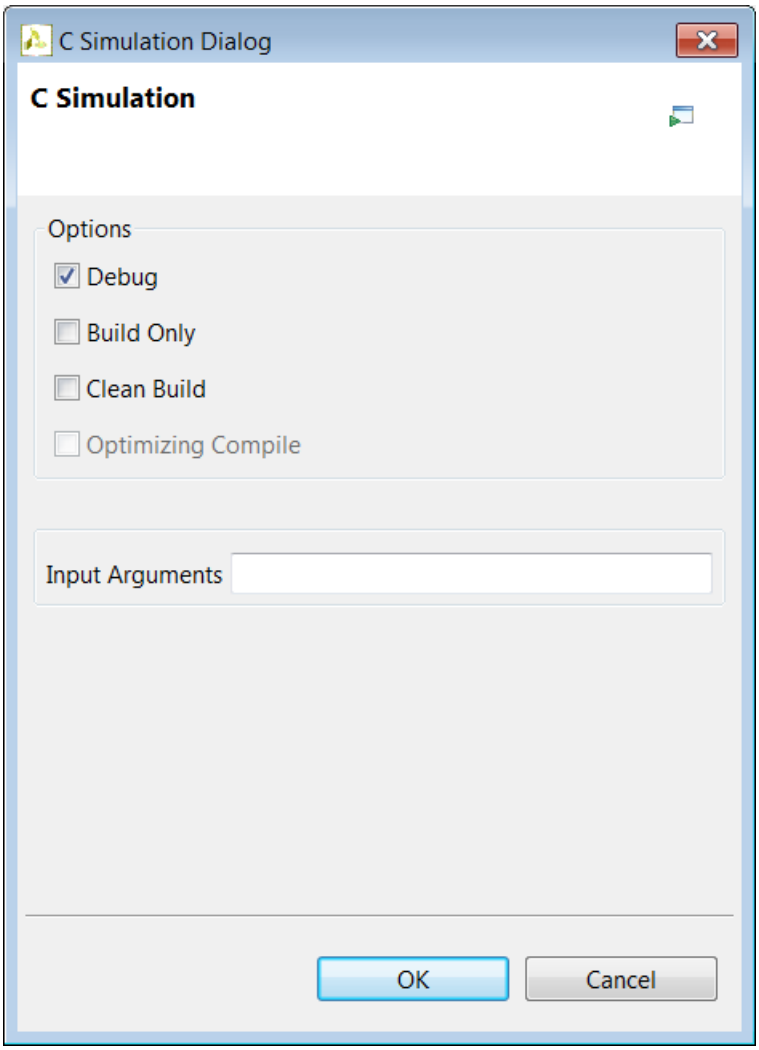


Figure 39 C Simulation Dialog Box

The Debug option compiles the C code and then opens the Debug environment as shown in Figure 40. Some things worth noting before proceeding:

- Highlighted at the top-left in Figure 40 the perspective has changed from Synthesis to Debug. The perspective buttons can be used to return to the synthesis environment at any time.
- The code is compiled in debug mode by default. The Debug option simply opens the debug perspective at time 0, ready for debug to begin. To compile the code without debug information, the Optimizing Compile option in the C simulation Dialog box should be selected.

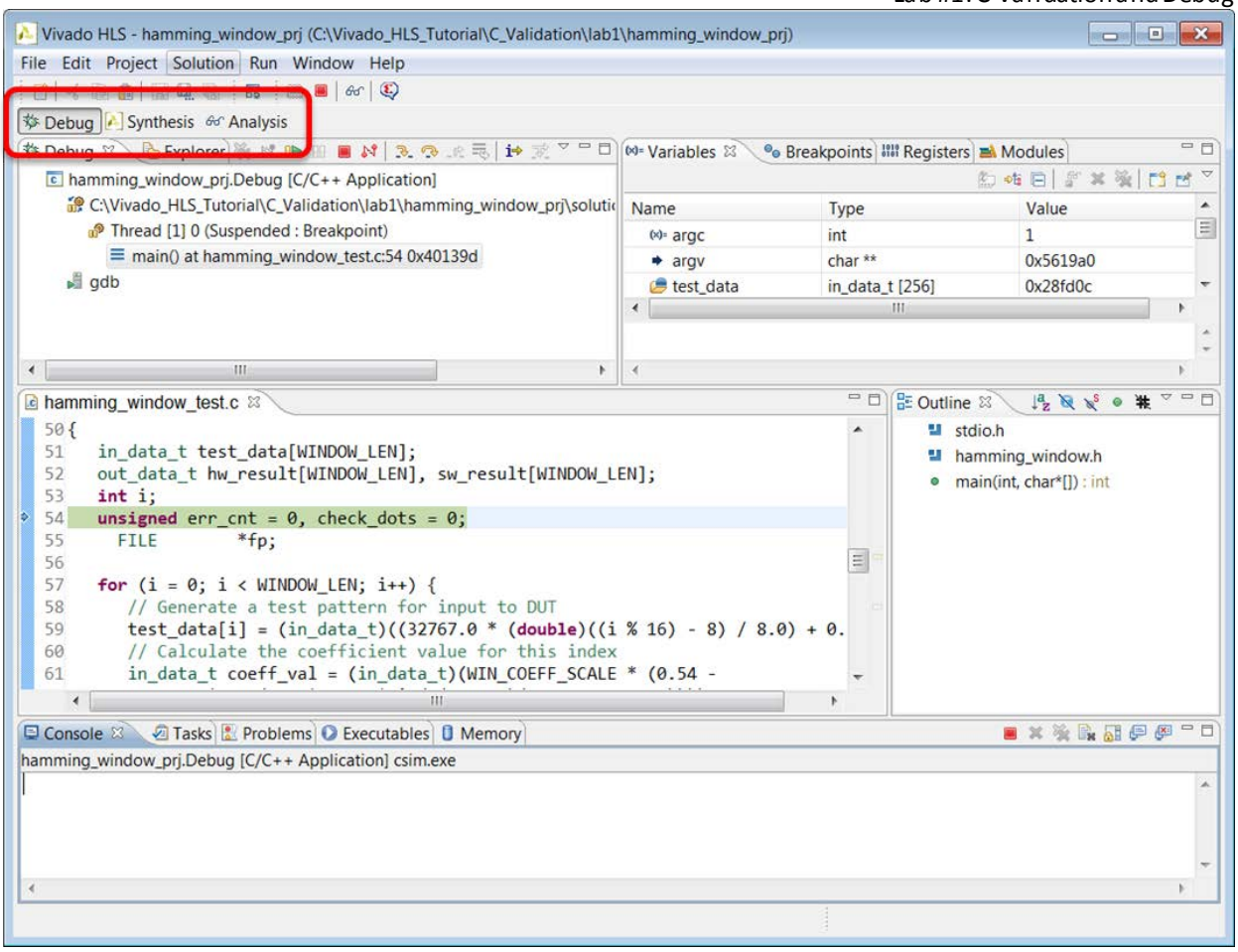


Figure 40 The HLS Debug Perspective

The Step Into button (Figure 41) can be used to step through the code line-by-line.

Figure 41 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 42.
6. Press the Step Into button (or key F5) multiple times until you can see the values being updated in the Variables window.

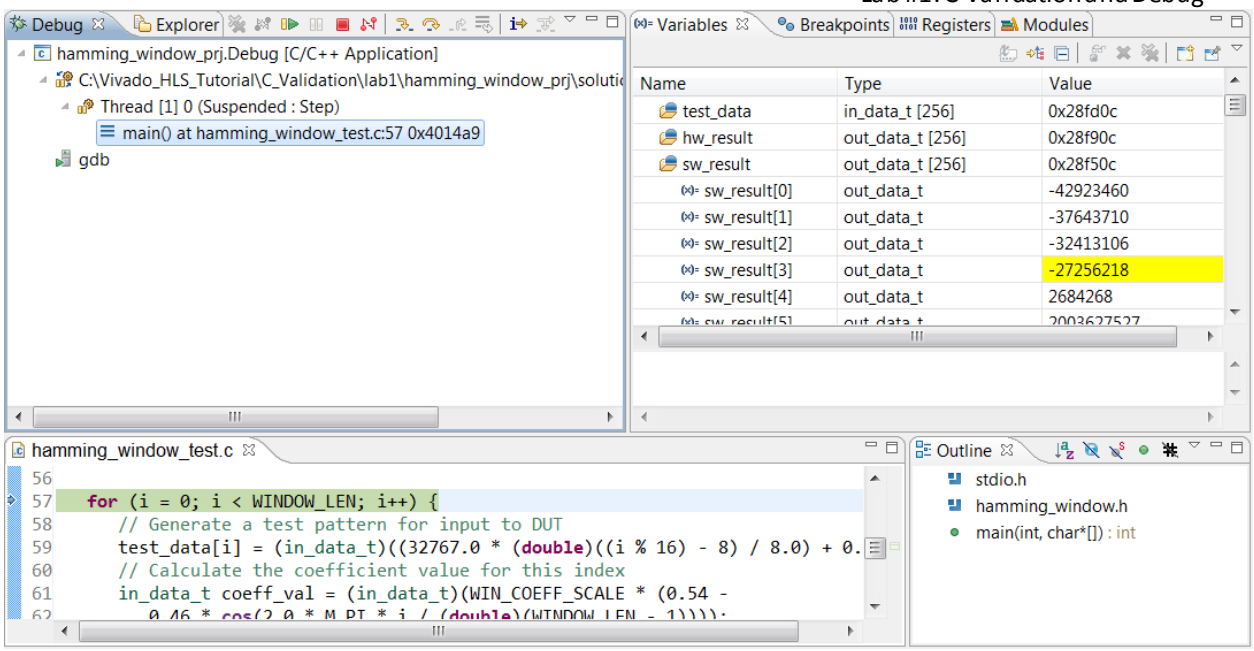


Figure 42 Analysis of C Variables

In this manner the C code can be analyzed and debugged if the behavior is incorrect. For more detailed analysis, to the right of the Step Into button are the Step Over (F6) and Step Return (F7) and to the left is the Resume (F8).

7. Scroll to line 69 in the source code window.
8. Double-Click in the left-hand margin to create a breakpoint (blue dot) as shown in Figure 43.
9. Activate the Breakpoints tab, also shown in Figure 43, to confirm there is a breakpoint set at line 69.
10. Press the Resume button (highlighted in Figure 43) or the F8 key to execute up to the breakpoint.

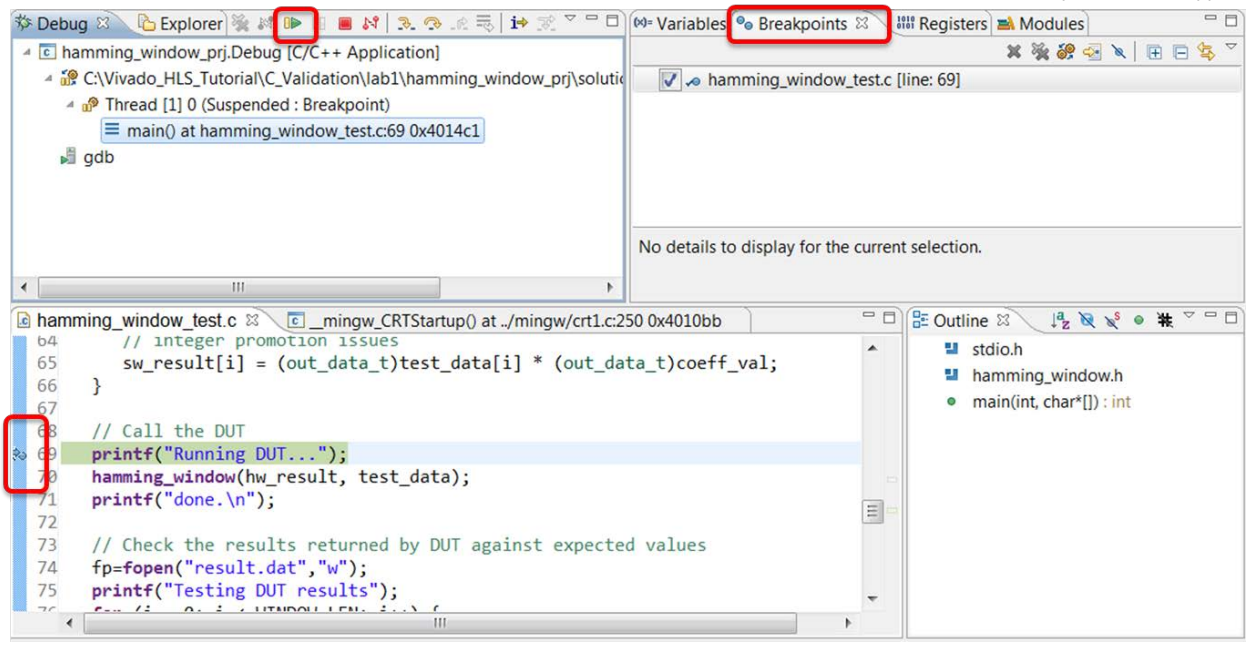


Figure 43 Using Breakpoints

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

The Terminate button transforms into the Run C Simulation button. The debug session can be re-started 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

This exercise uses a design with arbitrary precision C types. The design will be reviewed and the debugged 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 44.
2. Create a new Vivado HLS project by typing **vivado_hls –f run_hls.tcl.**

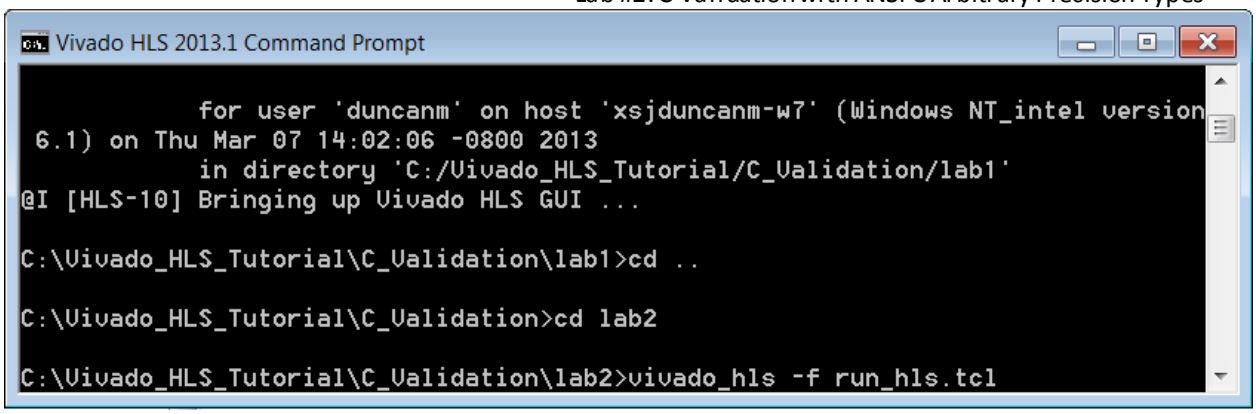


Figure 44 Setup for Interface Synthesis Lab#2

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 on hamming_window.c to open the code as shown in 45.

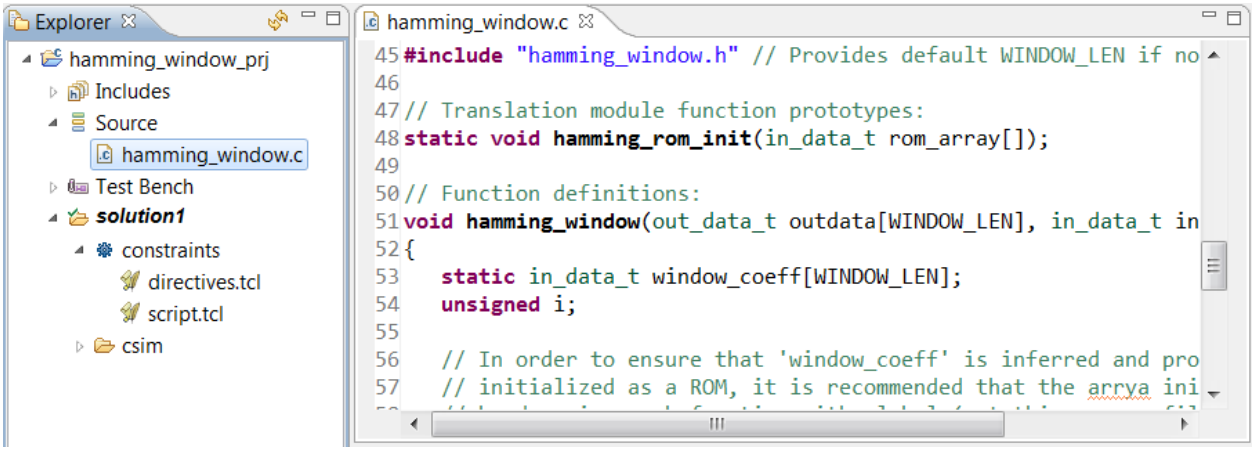


Figure 45 C Code for C Validation Lab#2

5. Hold the Control key down and click on the hamming_window.h on line 45 to open this header file.
6. Scroll down to view the type definitions (Figure 46).

hamming_window.c | hamming_window.h

```
68 // scaled integer, which may be interpreted as a signed fixed po
69 // with WIN_COEFF_FRACBITS bits after the binary point.
70
71 //typedef int16_t       in_data_t;
72 //typedef int32_t       out_data_t;
73 #include "ap_cint.h"
74 typedef int16       in_data_t;
75 typedef int32       out_data_t;
76
77 void hamming_window(out_data_t outdata[], in_data_t indata[]);
78
79 #endif // HAMMING_WINDOW_H_ not defined
80
```

Figure 46 Type Definitions for C Validation Lab#2

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

More details for using arbitrary precision types are discussed in the tutorial Arbitrary Precision Types (page 99). 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 will demonstrate how such types can be debugged.

Step 2: Run the C Debugger

1. Use the Run C Simulation toolbar button or the menu Project > Run C simulation to open the C Simulation Dialog box.
2. Select the Debug option.
3. Press OK to run the simulation.

The warning and error message shown in Figure 47 is presented.

The arbitrary precision types used for ANSI C design cannot be debugged in the debug environment. There are no issues with the C++ or SystemC arbitrary precision types, only those used with ANSI C designs.

CAUTION! *When using arbitrary precision types with C, the debug environment cannot be used.*

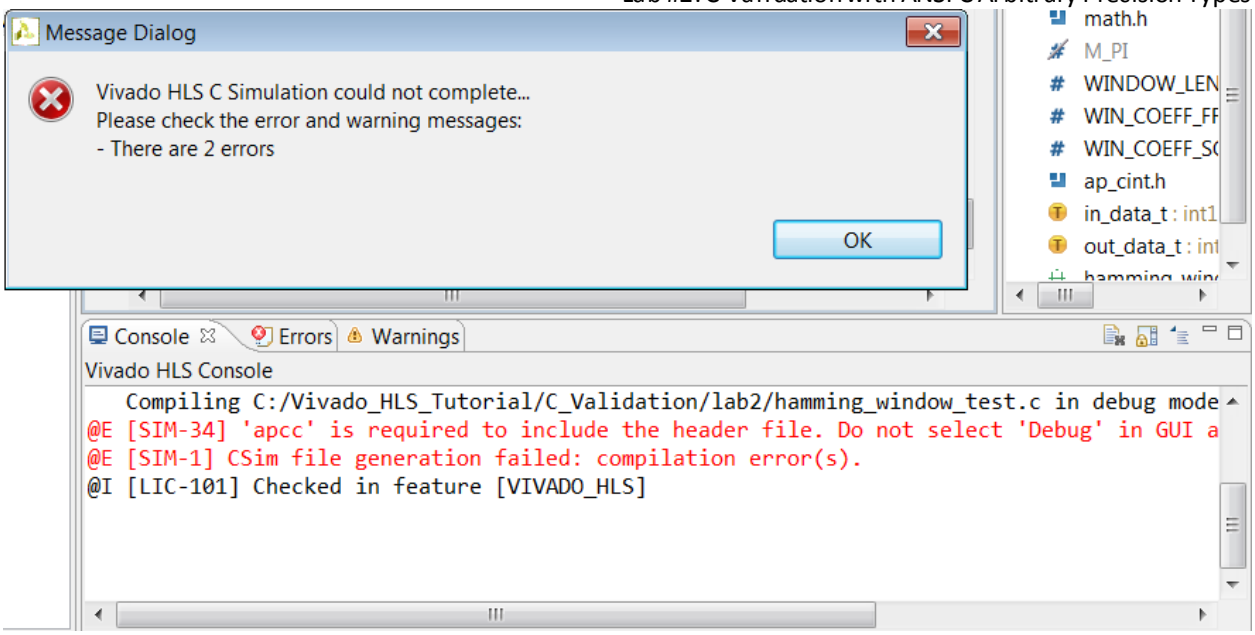


Figure 47 C Simulation Dialog Box

4. Expand the Test Bench folder in the Explorer pane.
5. Double-click on 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 48).

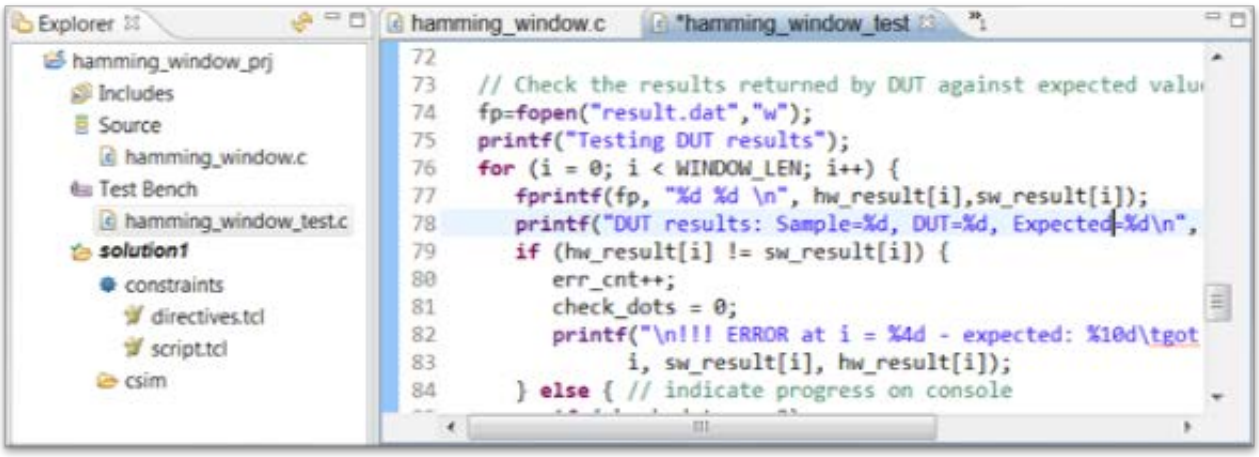


Figure 48 Enable Printing of the Results

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

The results are shown in the console window (Figure 49).

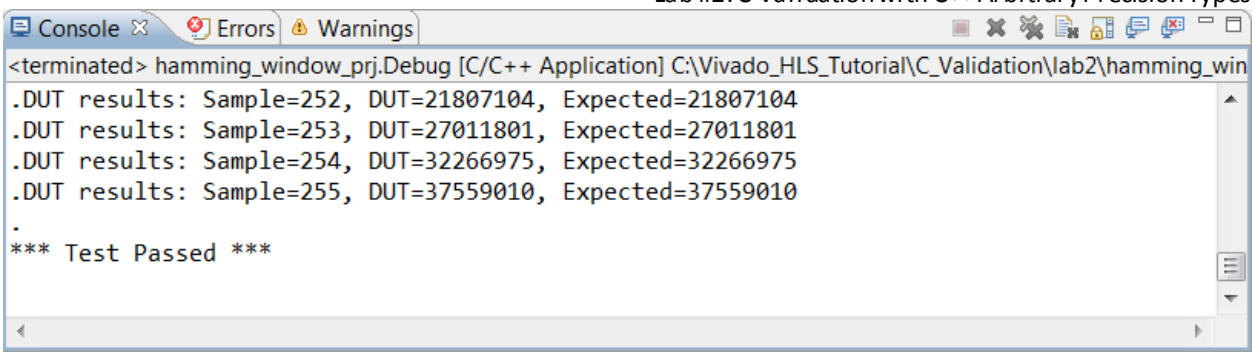


Figure 49 C Validation Lab#2 Results

When using arbitrary precision types in an ANSI C design, the debug environment cannot be used and printf or fprintf statements must be used to debug the design.

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

Lab #2: C Validation with C++ Arbitrary Precision Types

This exercise uses a design with arbitrary precision C++ types. The design will be reviewed and the debugged 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 on hamming_window.cpp to open the code as shown in Figure 50.

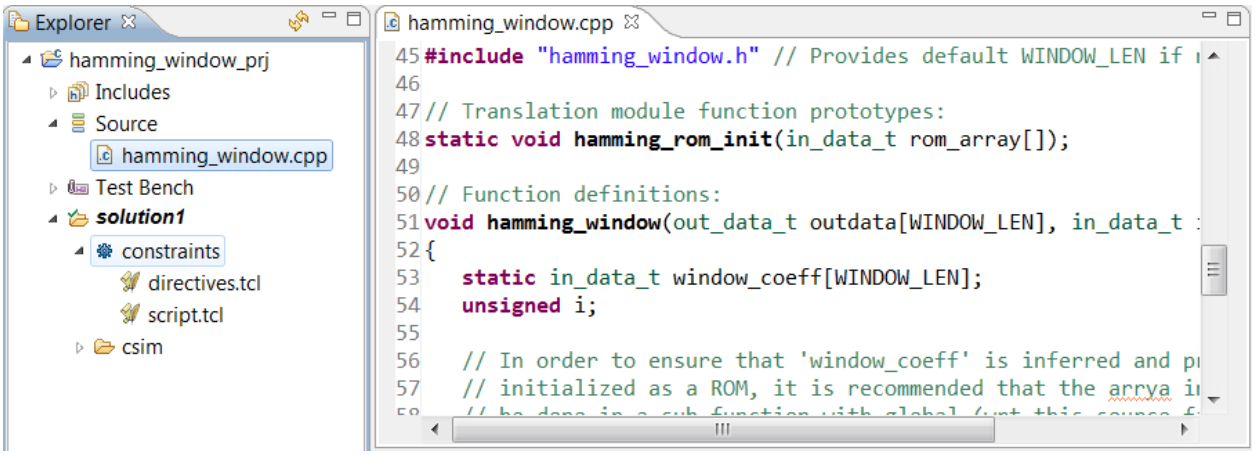


Figure 50 C++ Code for C Validation Lab#3

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

6. Scroll down to view the type definitions (Figure 51).

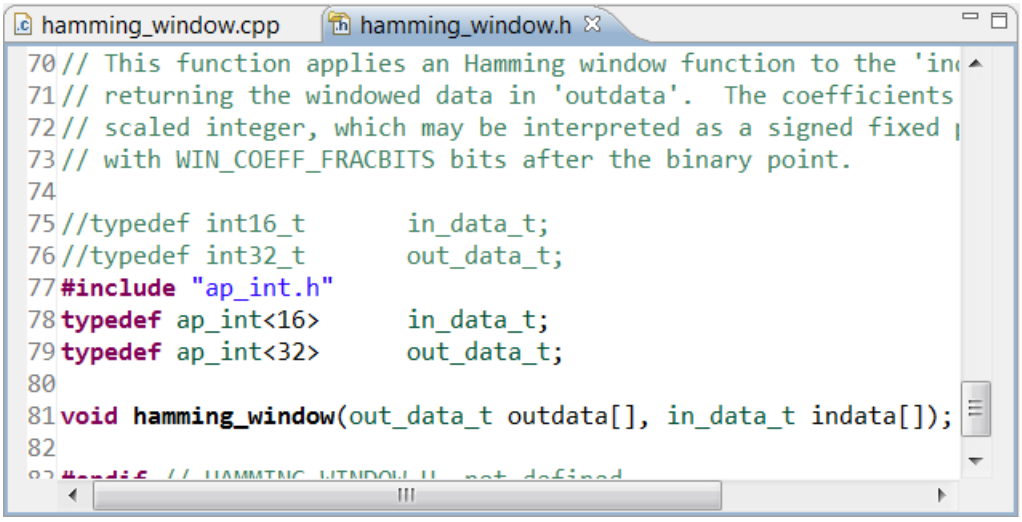


Figure 51 Type Definitions for C Validation Lab#3

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

Step 2: Run the C Debugger

1. Use the Run C Simulation toolbar button or the menu Project > Run C simulation to open the C Simulation Dialog box.
2. Select the Debug option.
3. Press OK.

The debug environment opens.

4. Switch to the hamming_window.cpp code tab.
5. Set a breakpoint at line 61 as shown in Figure 52.
6. Press the Resume button (or key F8) to execute the code up to the breakpoint.

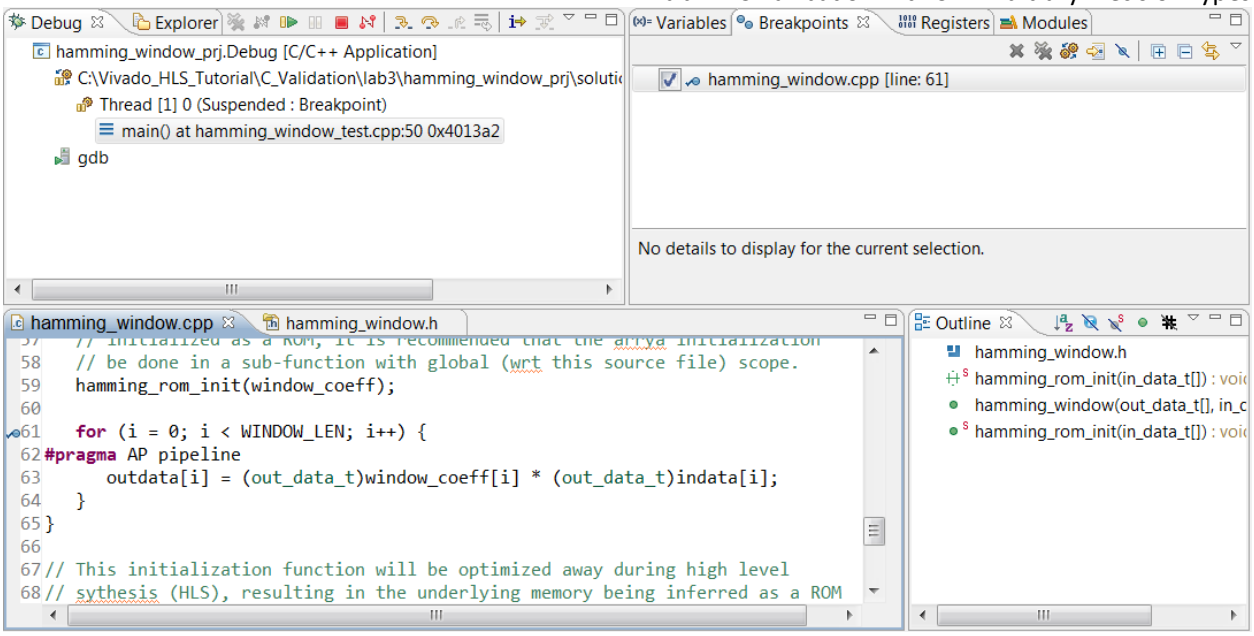


Figure 52 Debug Environment for C Validation Lab#3

7. Press the Step Into button (or key F5) twice to see the view in Figure 53.

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

Proceeding to step through the code will entail viewing the low-level details of how the results for arbitrary precision types are calculated.

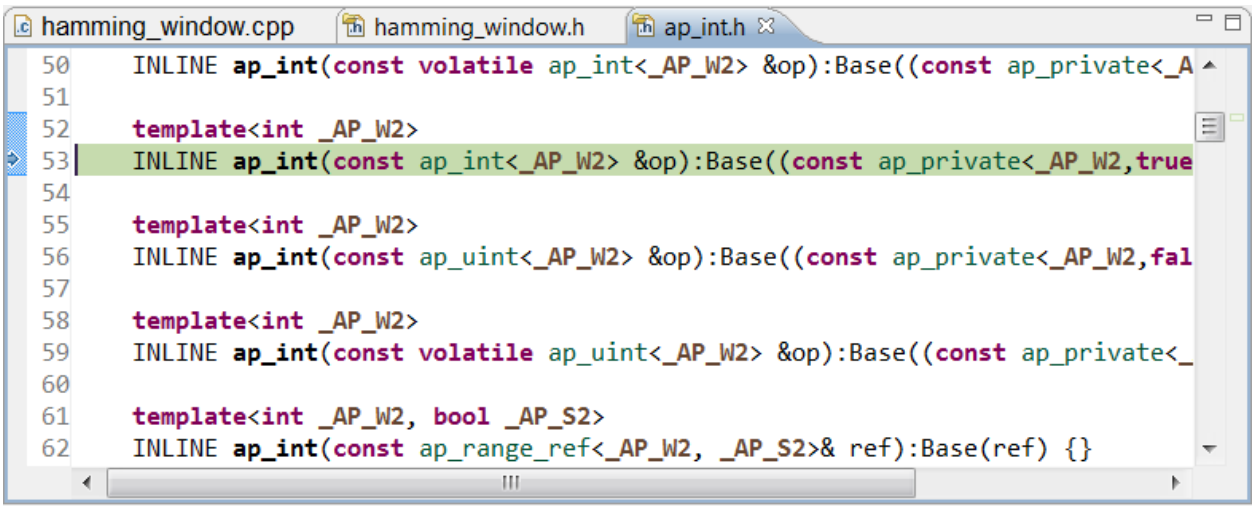


Figure 53 Arbitrary Precision Header File

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

8. Press the Step Return button (or key F7) to return to the calling function.
9. Activate the Variables tab.

10. Expand the outdata variable as shown in Figure 54 to see the value of the variable shown in the VAL parameter.

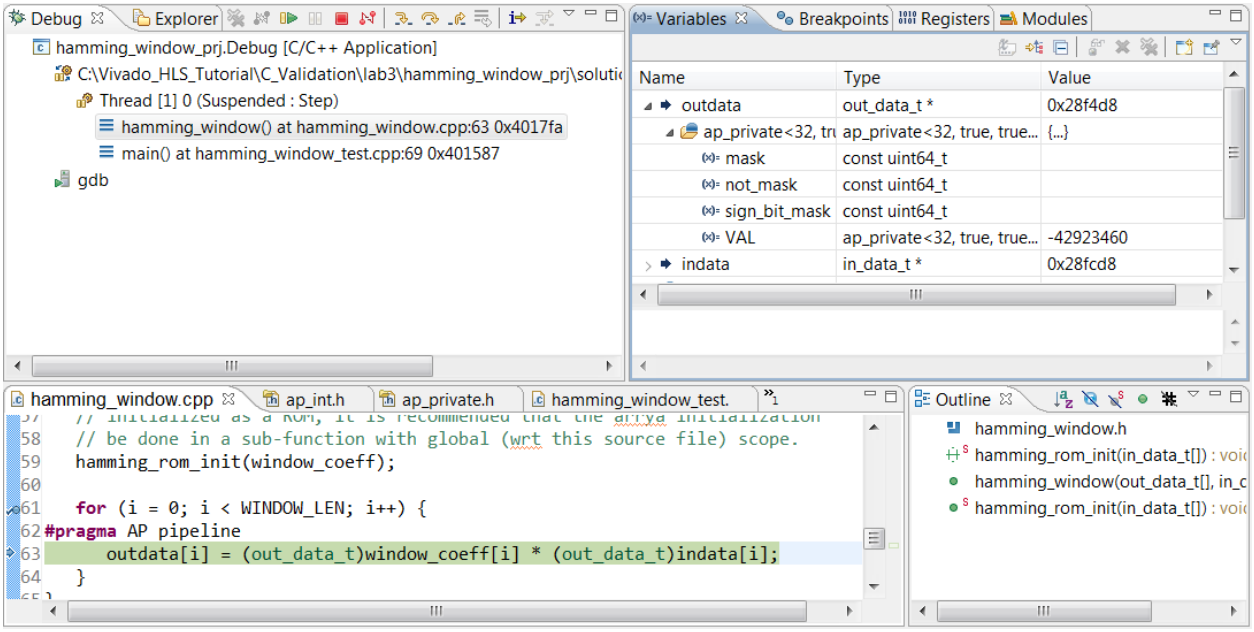


Figure 54 Arbitrary Precision Variables

Arbitrary precision types are a powerful means to create high-performance bit-accurate hardware designs. However, in a debug environment, 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.

This completes the tutorial on C validation.

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.

Interface Synthesis

Overview

Interface synthesis is the process of adding RTL ports to the C design. The term interface synthesis is used because this process is not simply about adding the physical port to the RTL design but includes adding an associated IO protocol, allowing the data transfer through the port to be automatically and optimally synchronized with the internal logic.

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

Lab1

Review the function return and block-level protocols.

Lab2

Understand the default IO protocol for ports and learn how to select an IO protocol.

Lab3

Review how array ports are implemented and can be partitioned.

Lab4

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

Tutorial Design Description

The tutorial design file can be download from the Xilinx website. Refer to the information in Obtaining the Tutorial Designs.

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

The sample designs used in the first two exercises in this tutorial is a simple design to allow the focus to remain on the interfaces. The final two lab exercises use a multi-channel accumulator.

This tutorial explains how IO ports and protocols are implemented using High-Level Synthesis. In Lab#4 the design goals are to create an optimal implementation of the design.

Lab #1: Block-Level IO protocols

This exercise will explain what block-level IO protocols are and how they can be controlled.

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

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

Step 1: Create and Open the Project

1. Open the Vivado HLS Command Prompt.
    a. On Windows use **Start** > **All Programs** > **Xilinx Design Tools** > **Vivado 2013.1** > **Vivado HLS** > **Vivado HLS 2013.1 Command Prompt (**Figure 55).
    b. On Linux, open a new shell.

Figure 55 Vivado HLS Command Prompt

2. Using the command prompt window (Figure 56), 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 56.

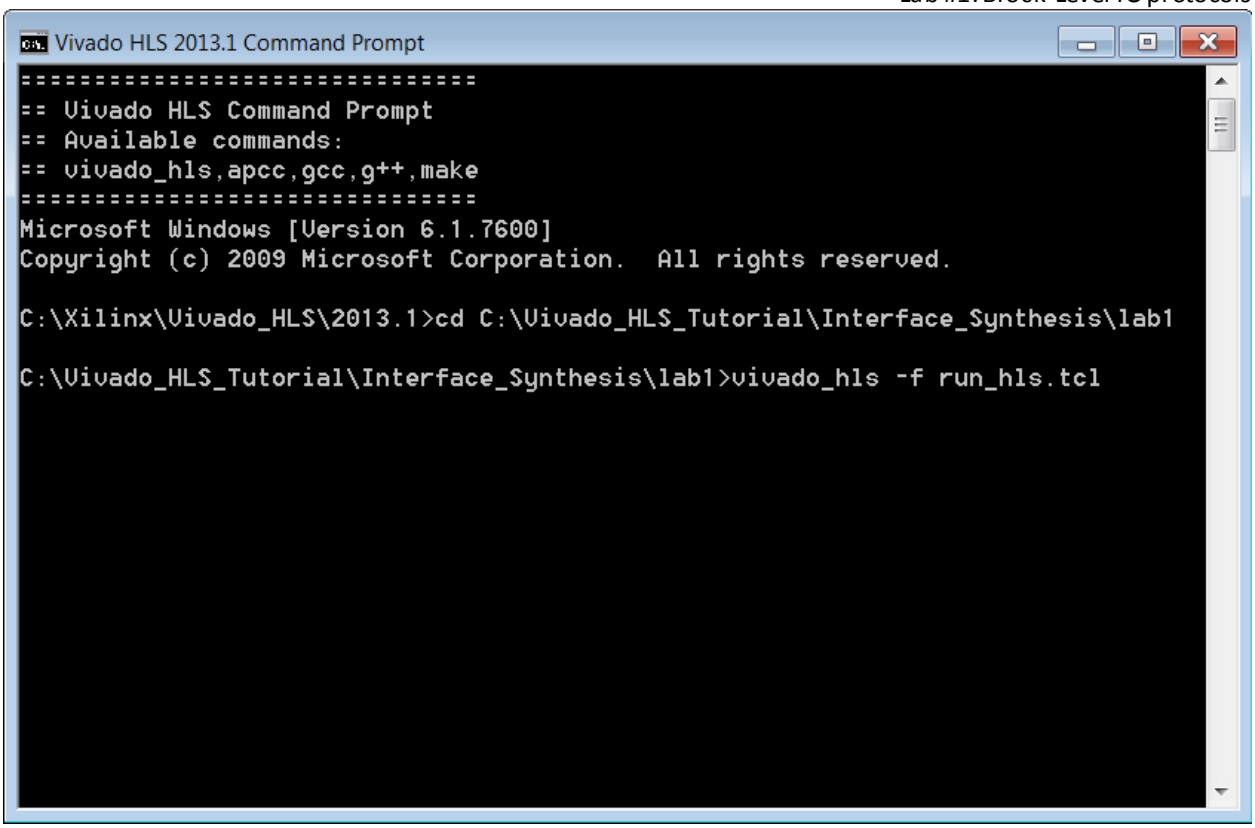


Figure 56 Setup the Tutorial Project

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 57.

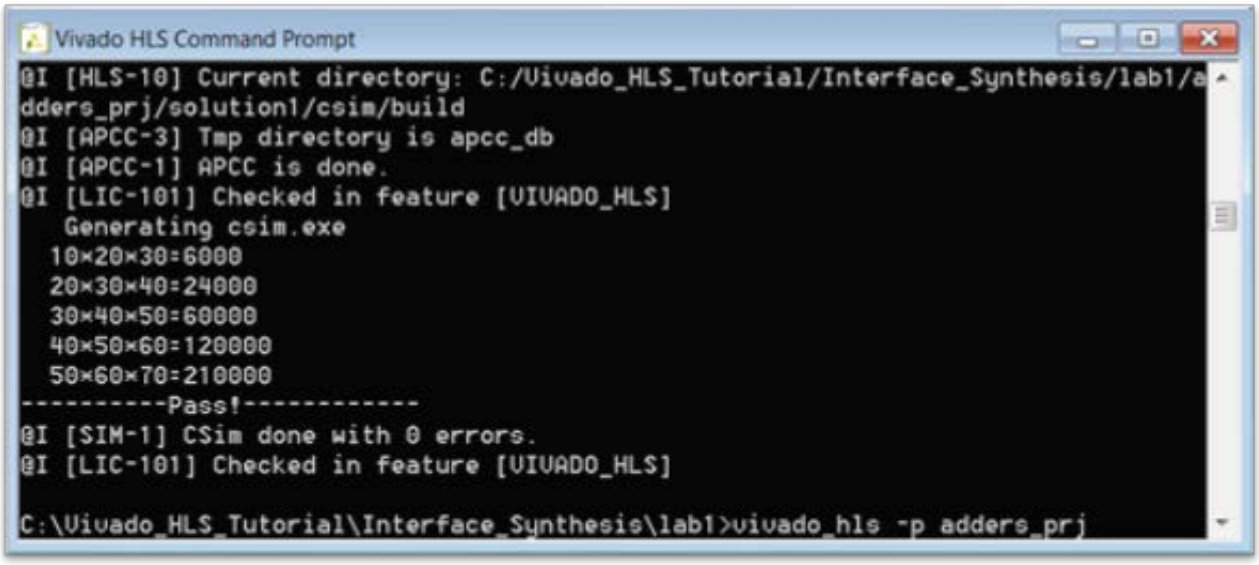


Figure 57 Initial Project for Interface Synthesis Lab#1

Step 2: Create and Review the Default Block-Level IO Protocol

1. Open the source code for review by double-clicking on adder.c in the Source folder (Figure 58).

This example uses a simple design to focus on the IO 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 IO protocol being synthesized for any of the data ports (inA, inB and inC). IO port protocols will be reviewed in the next lab exercise.
- This function returns a value and this is the only output from the function. As will be seen in later exercises, not all functions return a value. The port created for the function return will be discussed in this lab exercise.

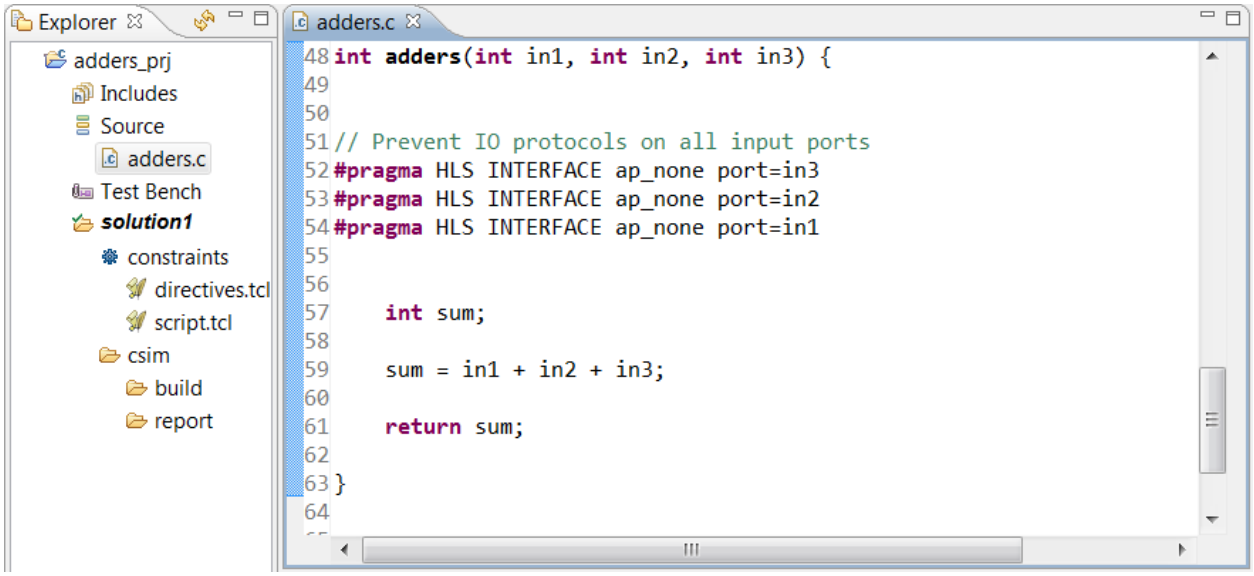


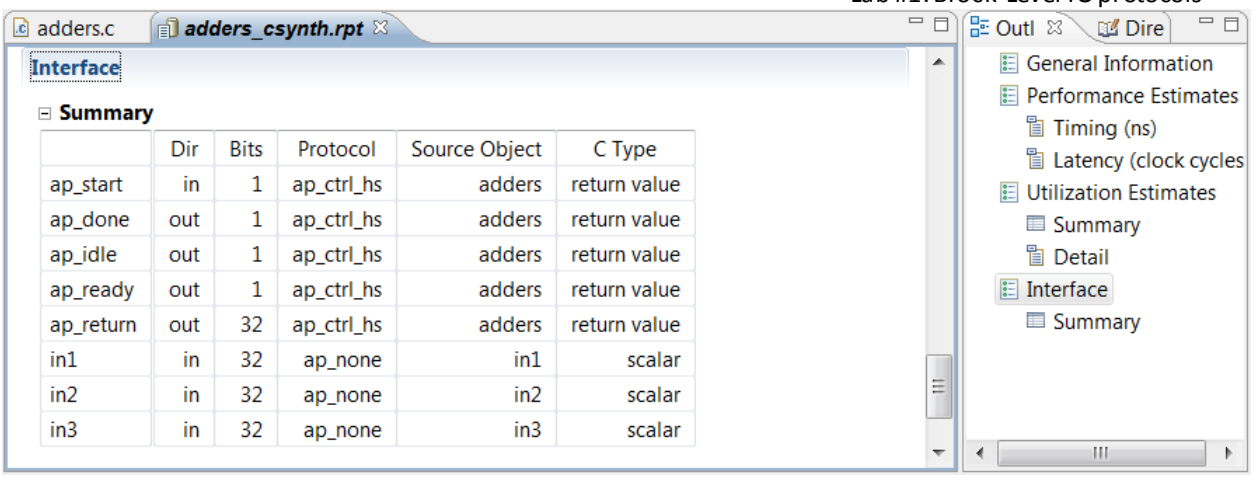
Figure 58 C Code for Interface Synthesis Lab#1

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

When synthesis completes, the synthesis report will open automatically.

3. To review the RTL interfaces scroll to the Interface summary at the end of the synthesis report **or** use the Outline tab and select Interface summary.

The Interface summary and Outline tab are shown in Figure 59.



	Dir	Bits	Protocol	Source Object	C Type
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 59 Interface Summary

There are three types of ports to review here:

- 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 IO 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 IO protocol ap_none (as specified by the directives in Figure 59).
    - The design also has a 32-bit output port for the function return, ap_return.

The block-level IO protocol allows the RTL design to be controlled independently of the IO ports. This IO protocol is associated with the function itself, not any of the data ports. The default block-level IO protocol is named ap_ctrl_hs and Figure 59 shows this protocol as being associated with the clock.

Table 1 summarizes the behavior of the signals for block-level IO protocol ap_ctrl_hs.

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

Exercise	Description
ap_start	This signal controls the block execution and must be asserted to logic 1 for the design to begin operation. It should be held at logic 1 until the associated output handshake ap_ready is asserted. When ap_ready goes high, the decision can be made on whether to keep ap_start asserted and perform another transaction or set ap_start to logic 0 and allow the design to halt at the end of the current transaction. If ap_start is asserted low before ap_ready is high, the design may not have read all input ports and may stall operation on the next input read.

Exercise	Description
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 will not be performed until the next transaction starts. This signal should be used to make a decision on when to apply new values to the inputs ports and whether a new transaction should be started using the ap_start input signal. If the ap_start signal is not asserted high, this signal will go low when the design completes all operations in the current transaction.
ap_done	This signal indicates when the design has completed all operations in the current transaction. A logic 1 on this output indicates the design has completed all operations in this transaction. Since 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 will be performed.

Table 1 Block Level IO protocol ap_ctrl_hs

The behavior of these signals can be seen by viewing the trace file produced by RTL cosimulation. This is discussed in the tutorial RTL Verification (page 167), but Figure 60 shows the waveforms for the current synthesis results.

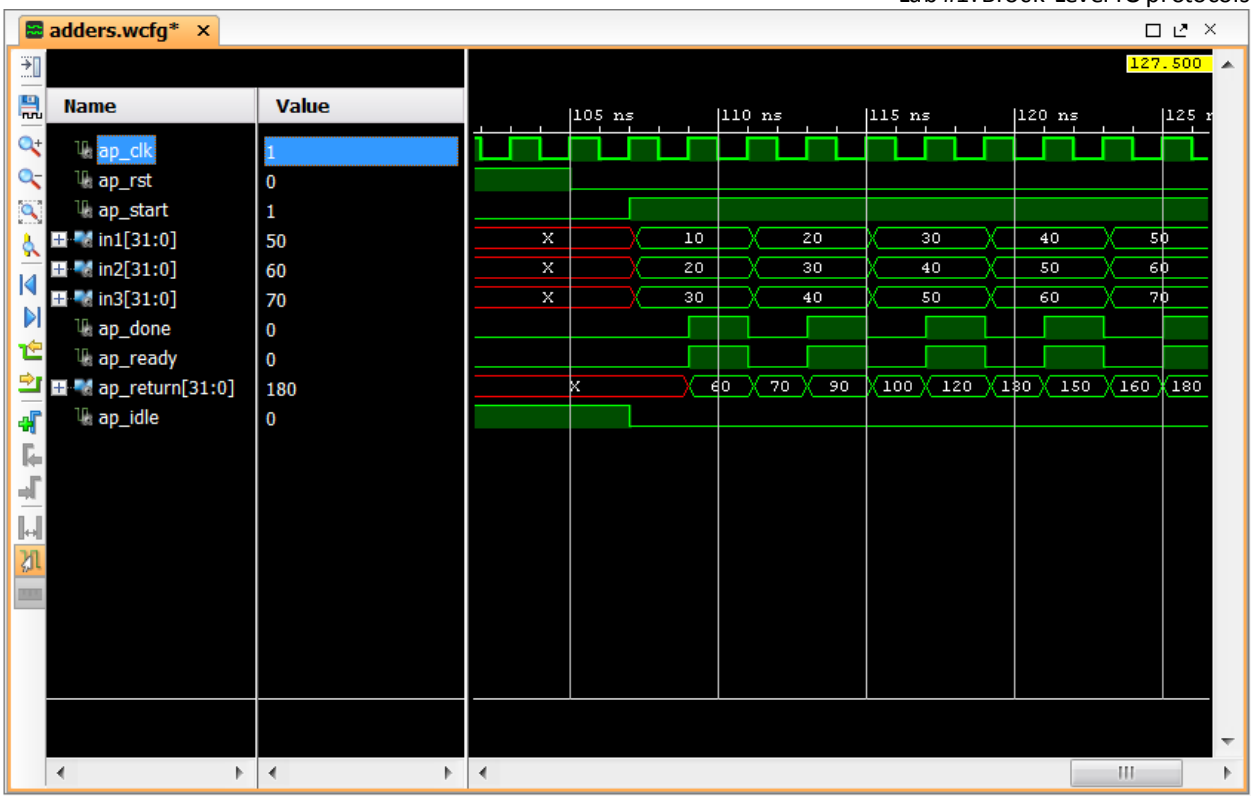


Figure 60 RTL Waveforms for Block Protocol Signals

The waveforms in Figure 60 show the behavior of the block-level IO signals.

- The design does not start operation until ap_start is set to logic 1.
- The design indicates it is no longer idle.
- Five transactions are shown. The first 3 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_start is valid (the first output value 60, is the sum of all three inputs).
- Since 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 60 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 cannot be considered valid until the ap_done signal is asserted high.

Step 3: Modify the Block-Level IO protocol

The default block-level IO protocol is the ap_ctrl_hs protocol (the Control Handshake protocol). In this step we will 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 press OK

3. Select the C source code tab in the Information pane (or re-open the C source code if it was closed).
4. Activate the Directives tab and select the top-level function, as shown in Figure 61.

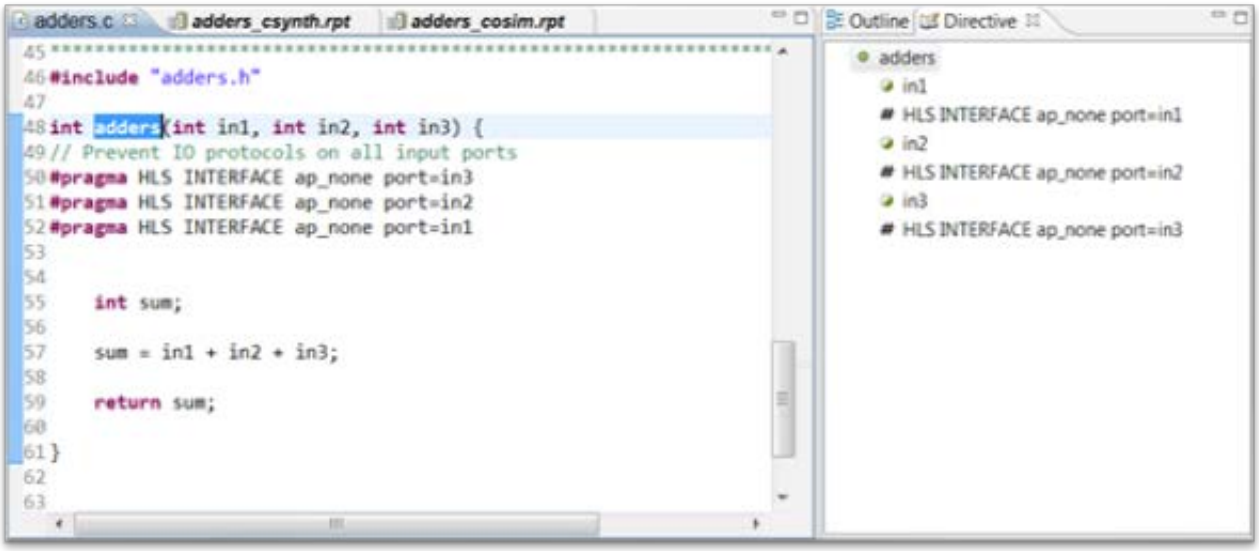


Figure 61 Top-level Function Selected

Since the block-level IO protocols are associated with the function, they must be accessed by selecting the top-level function.

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

The Directives Editor dialog box will open. Figure 62 shows this dialog box with the drop-down menu for the interface mode activated.

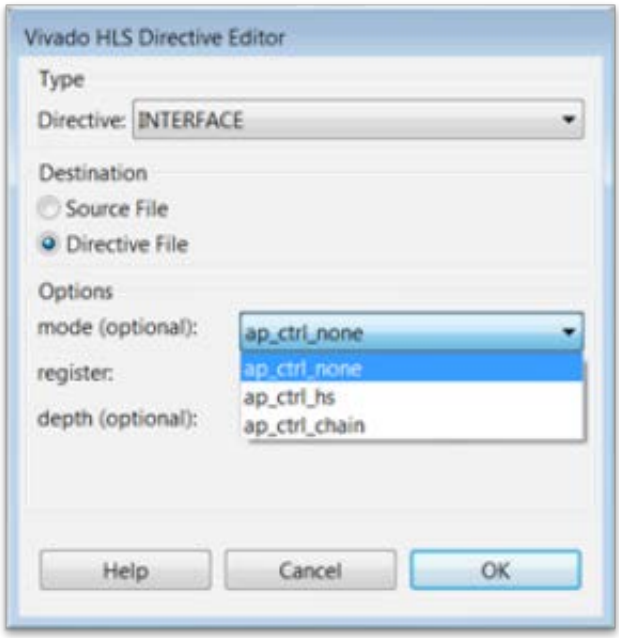


Figure 62 Directive Dialog box for ap_ctrl_none

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

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

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

7. 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 will be placed in the source file where the existing IO directives are located.

8. Select ap_none from the drop-down mode menu.
9. Press OK.

The source file now has a new directive, highlighted in both the source code and directives tab in Figure 63.

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

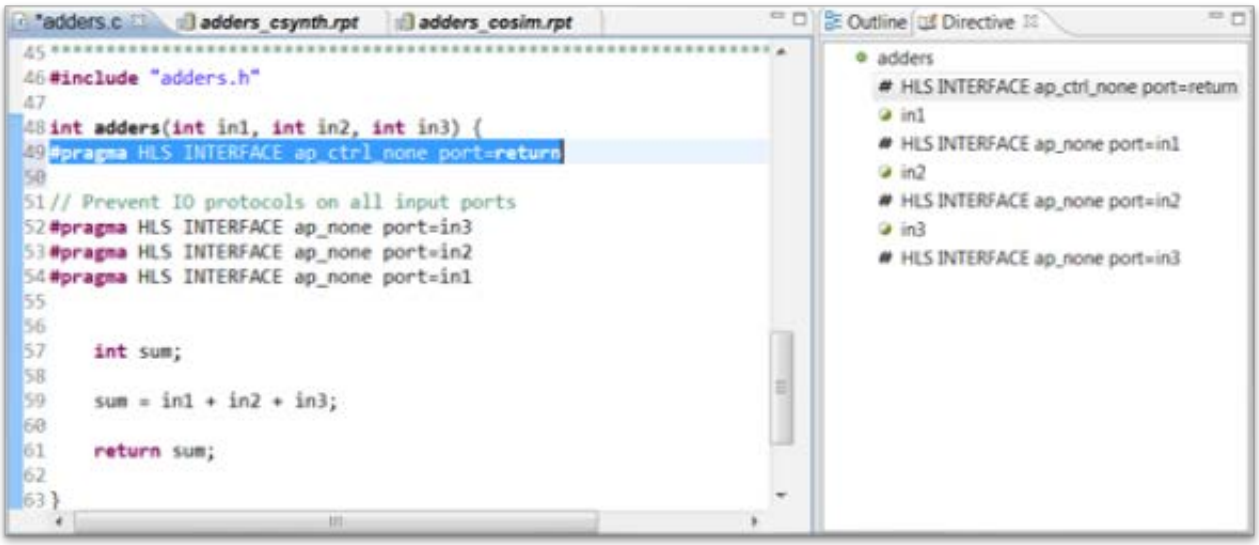


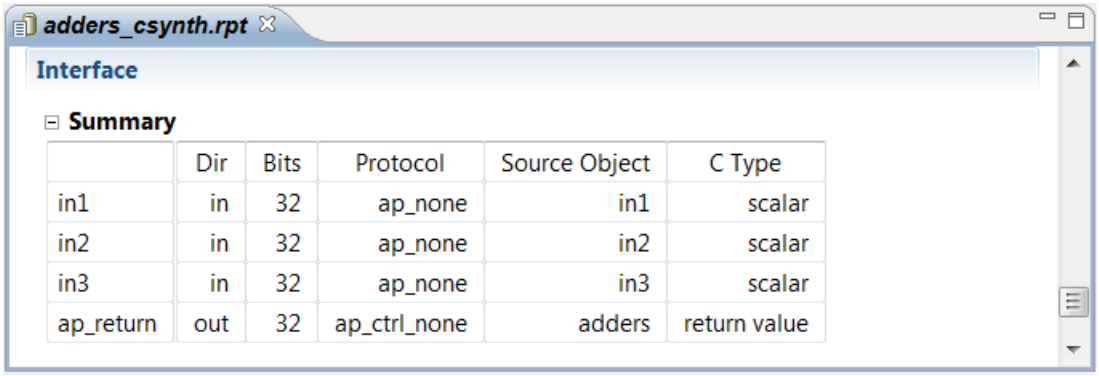
Figure 63 Block-Level Interface Directive ap_ctrl_none

10. Press 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 63 shows the source file name as *adders.c indicating the file has been modified but not saved.

11. Press Yes to accept the changes to the source file.

When the report opens, the Interface summary will be as shown in Figure 64.



	Dir	Bits	Protocol	Source Object	C Type
in1	in	32	ap_none	in1	scalar
in2	in	32	ap_none	in2	scalar
in3	in	32	ap_none	in3	scalar
ap_return	out	32	ap_ctrl_none	adders	return value

Figure 64 Interface summary for ap_ctrl_none

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

It should be noted that without the ap_done signal, the consumer block which accepts this data from the ap_return port is now responsible for knowing when the data on the ap_return port is valid: there is now no indication from the design as to when this data is valid.

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

```
@E [SIM-44] The design was synthesized without function protocol ('ap_ctrl_none').
Test bench cannot be automatically generated. Please either re-synthesize the design
by adding back the function protocol or use a manual test bench to verify the RTL.

@E [SIM-4] *** C/RTL co-simulation finished: FAIL ***
```

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

Lab #2: Port IO protocols

This exercise will explain how port IO protocols can be specified.

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 65.
2. Create a new Vivado HLS project by typing vivado_hls –f run_hls.tcl

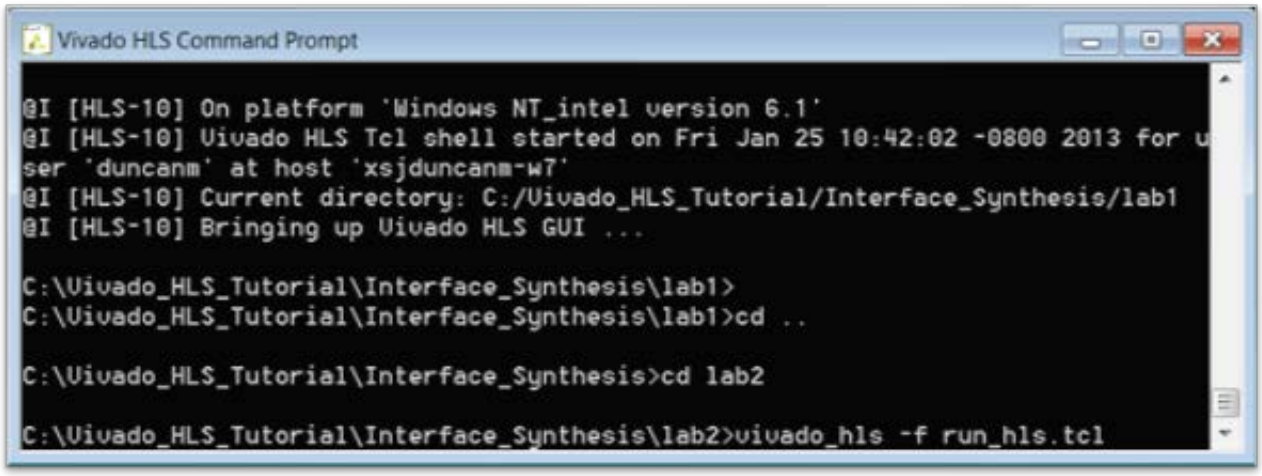


Figure 65 Setup for Interface Synthesis Lab#2

3. Open the Vivado HLS GUI project by typing vivado_hls –p adders_io_prj
4. Open the source code as shown in Figure 66.

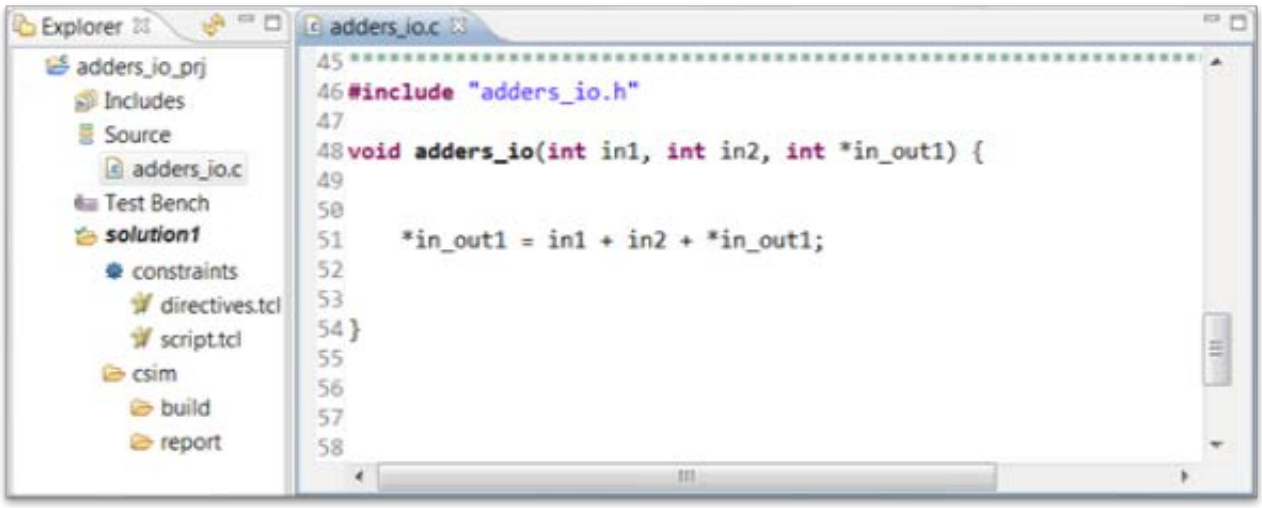


Figure 66 C Code for Interface Synthesis Lab#2

The source code for this exercise is similar to the simple code used in Lab#1 – again, for similar reasons, allowing us to 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 pointer is also used to accept input data (in place of in3 in the previous example). This will provide the opportunity to explore the interface options for inout and output ports.

The types of IO protocol which can be added to C function arguments by interface synthesis, depends on the argument type. These options are fully explained in the Vivado High-Level Synthesis User Guide (UG902).

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

For the code shown in Figure 66, the possible options for each function argument are described in Table 2.

Function Argument	IO protocol Options
In1 and In2	These are pass-by-value arguments and may be implemented with the following IO Protocols: • ap_none: No IO protocol. This is the default for inputs. • ap_stable: No IO 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 and may be implemented with the following IO protocols: • ap_none: No IO protocol. This is the default for inputs. • ap_stable: No IO 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.

Table 2 Port Level IO Protocol Options for Lab#2

Note: The port directives applied in Lab#1 where not in fact necessary since ap_none is the default IO protocol for these C arguments. They were provided to ensure we did not address any IO port protocol behavior in that exercise, default behavior or not.

We will implement a selection of IO protocols for this exercise.

Step 2: Specify the IO Protocol for Ports

1. Ensure the C source code can be viewed in the Information pane.
2. Activate the Directives tab and select input argument in1 as shown in Figure 67.

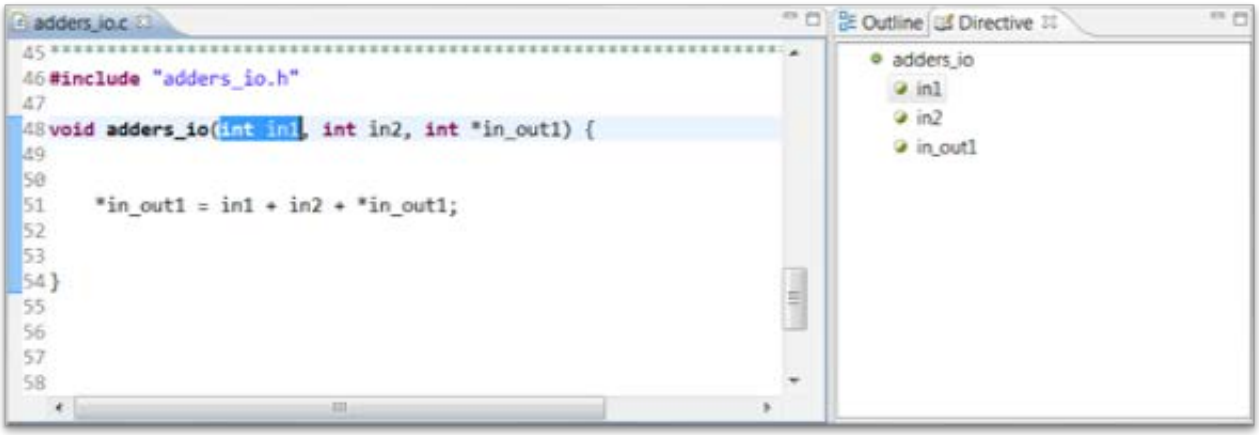


Figure 67 Adding Port IO Protocols

3. Right-click with the mouse and select Insert Directives.
4. When the Directives Editor opens leave the directives drop-down menu as INTERFACE.
    a. Leave the destination at the default value. This time, the directives will be stored in the directives.tcl file.
    b. Select ap_vld from the mode drop-down menu
    c. Press OK.
5. Select argument in2 and add an interface directive to specify the IO protocol ap_ack.
6. Select argument in_out1 and add an interface directive to specify the IO protocol ap_hs.
7. In the Explorer pane, expand the Constraints folder and Double-click on the directives.tcl file to open it, as shown in Figure 68.

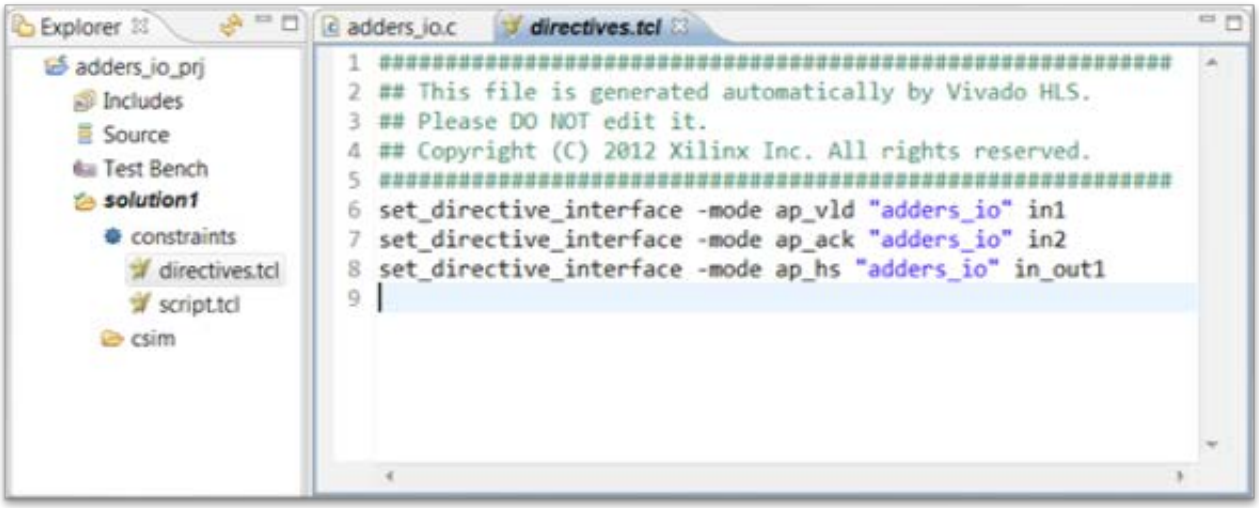
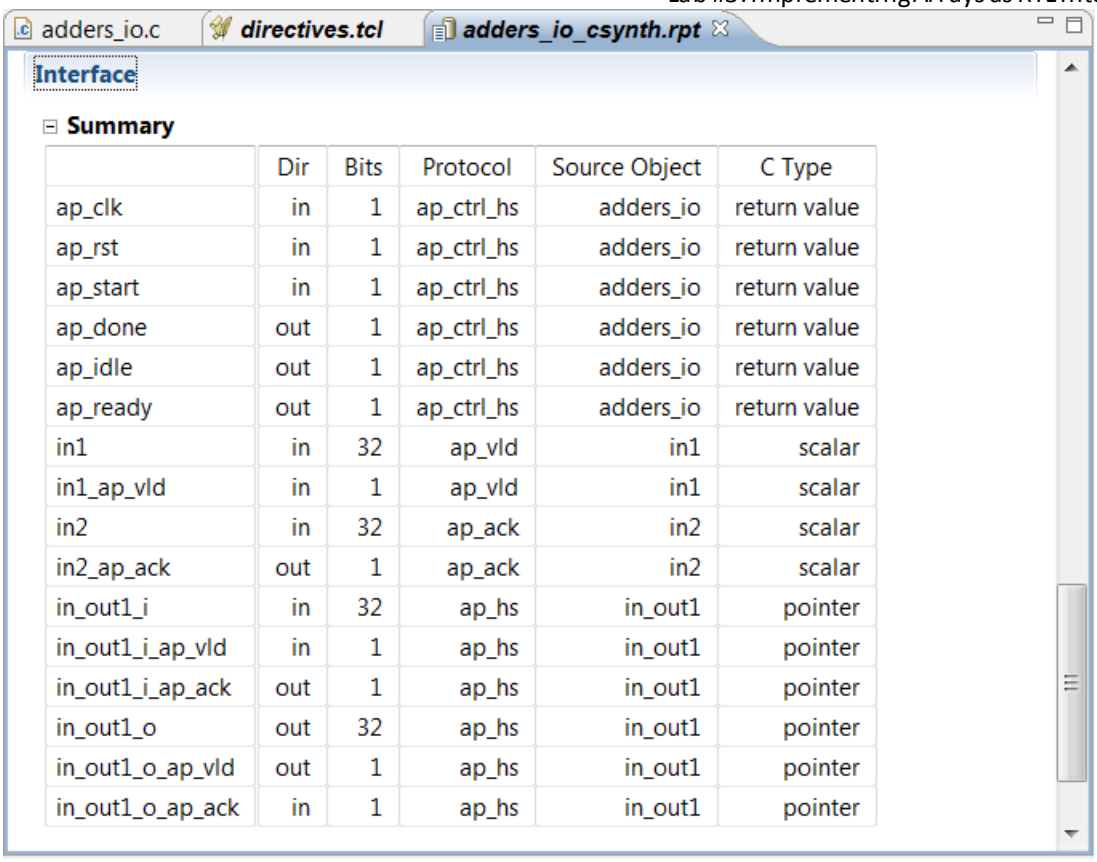


Figure 68 Directives for Lab#2

8. Synthesize the design.
9. Review the Interface summary when the report file opens (Figure 69).


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 69 Interface summary for Lab#2

- The design has a clock and reset.
- The default block-level IO protocol signals are present.
- Port in1 is implemented with a data port and an associated input valid signal.
    - The data on port in1 will only be 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 input part of argument inout1 is identified as inout_i. 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.

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

Lab #3: Implementing Arrays as RTL Interfaces

This exercise will show how array arguments on functions 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 arrays_io_prj
4. Open the source code as shown in Figure 70.

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.

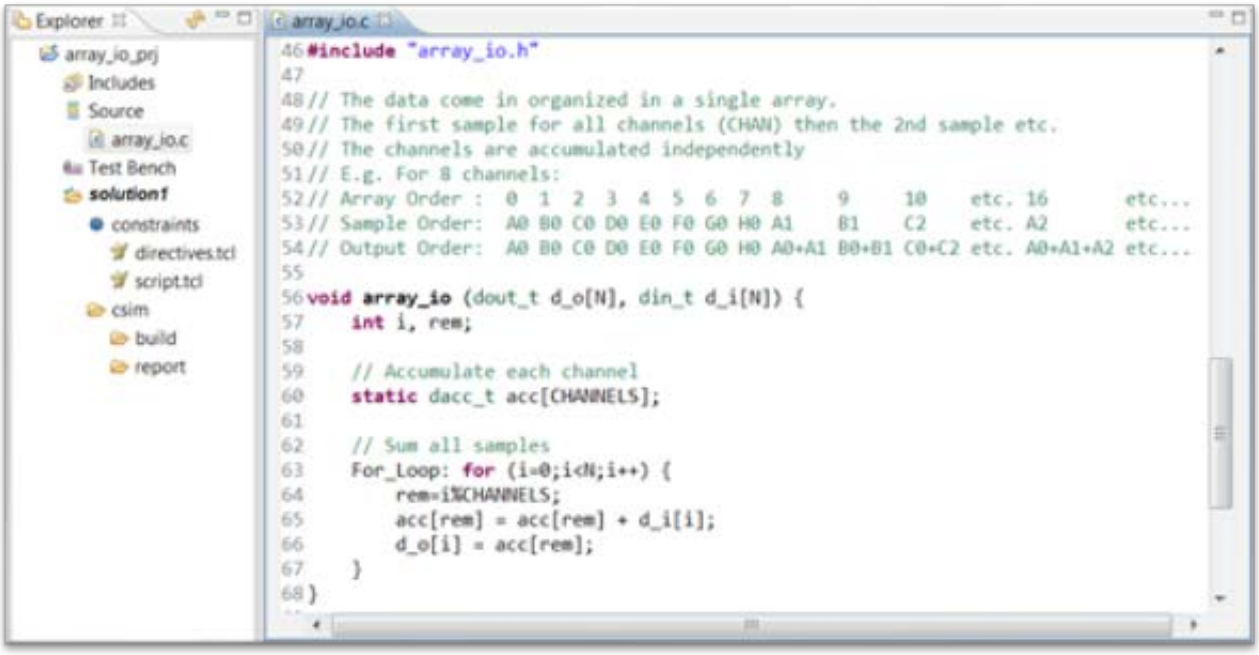


Figure 70 C Code for Interface Synthesis Lab#3

Step 2: Synthesize Array Function Arguments to RAM ports

In this step we will review how array ports are synthesized to RAM ports.

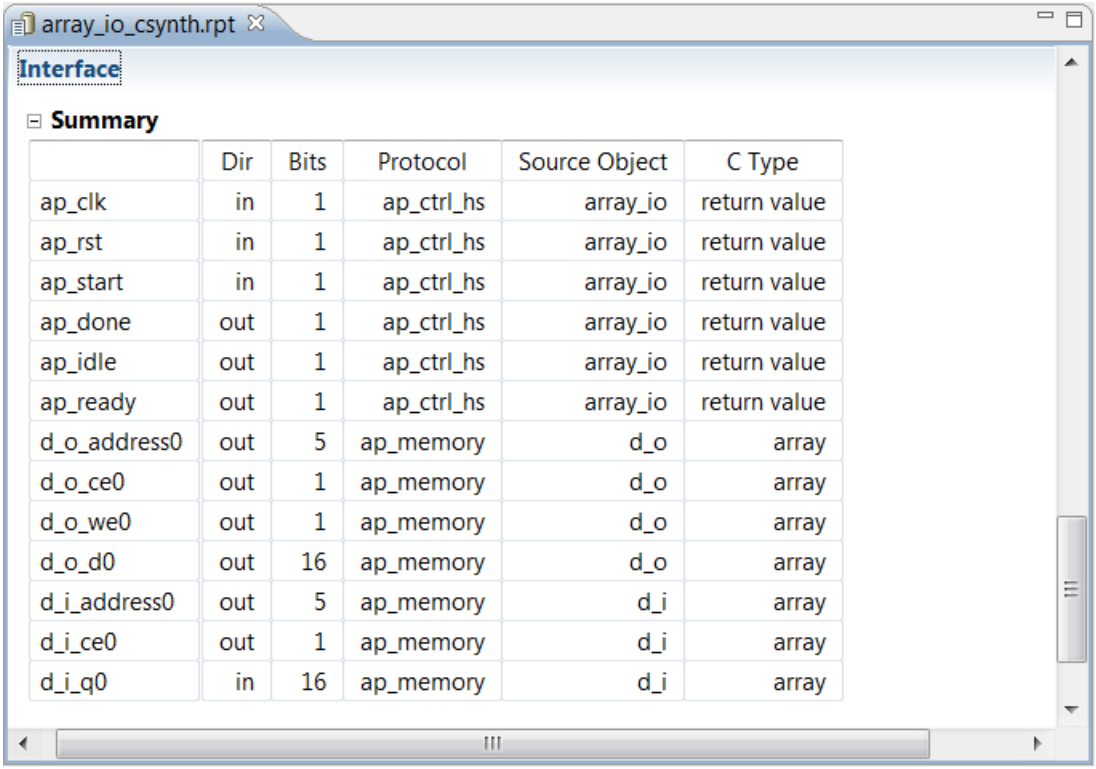
1. Synthesize the design and review the Interface summary when the report opens (Figure 71).

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 IO protocol ap_ctrl_hs (noted on the clock in the report).
- The d_o argument has been synthesized to a RAM port (IO protocol ap_memory).
  - A data port (d_o_d0).
  - An address port (d_o_address0).
  - Control ports for chip-enable (d_o_ce0) and a write-enable port (do_we0).

- The d_i argument has been synthesized to a similar RAM interface, but has an input data port (d_i_q0) and no write-enable port since this interface only reads data.

In both cases, the data port is the width of the data values in the C source (32-bit integers in this case) and the width of the address port has been automatically sized match to the number of addresses which 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 71 Interface Summary for Initial Lab#3 design

Synthesizing array arguments to RAM ports is the default. There are a number of other options to control how these ports are implemented and the remaining steps in Lab#3 will demonstrate these options:

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

Step 3: Using Dual-port RAM and FIFO interfaces

High-Level Synthesis allows a RAM interface to be specified as a single-port or dual-port. If no such selection is made, Vivado HLS will automatically analyze the design and select the number of ports to maximize the data rate.

In step 2, a single-port RAM interface is used because the for-loop in the source code (Figure 70) 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 a dual-port RAM is specified and Vivado HLS can determine only a single port is required, it will use a single-port and over-ride the dual-port specification.*

With this design, the first requirement in demonstrating how an array argument can be implemented using multiple RTL ports is to unroll the for-loop and allow all internal operations to happen in parallel, as much as data dependencies allow.

1. Select New Solution from the toolbar or Project menu to create a new solution.
2. Press OK and accept the defaults.
3. Ensure the C source code is visible in the Information pane.
4. In the Directives tab, select the for-loop For_Loop and right-click with the mouse to open the Directives Editor dialog box.
    a. In the Directives Editor dialog box activate the Directives drop-down menu at the top and select UNROLL.
    b. With the Directives Editor as shown in Figure 72, click OK.

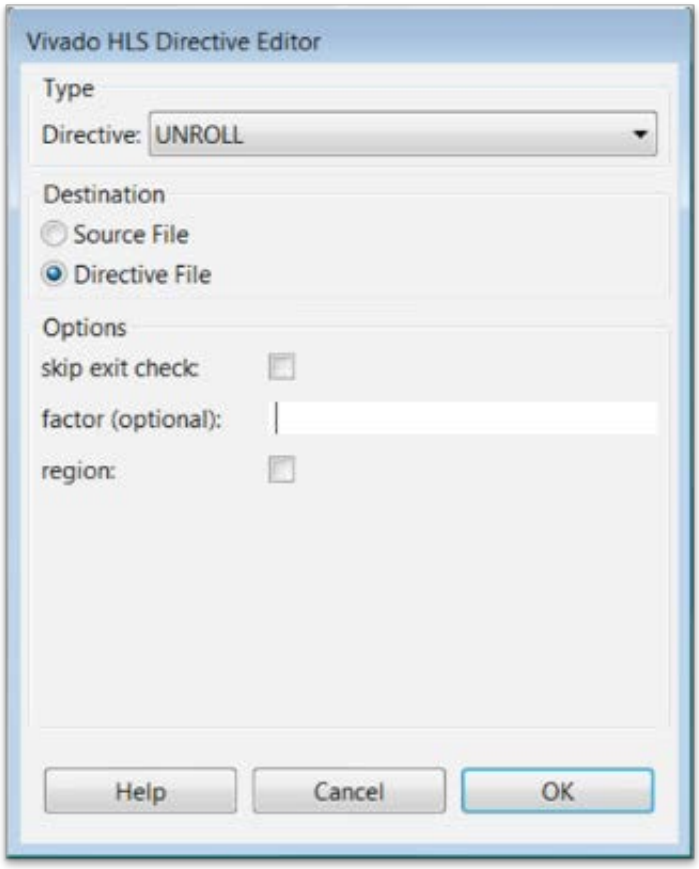


Figure 72 Directives Editor to Unroll For_Loop

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

5. In the Directives tab, select port d_i and right-click with the mouse to open the Directives Editor dialog box.
    a. In the Directives Editor activate the Directives drop-down menu at the top and select RESOURCE.
    b. Click on the core options box and select RAM_2P_RAM.
    c. With the Directives Editor as shown in Figure 73, click OK.

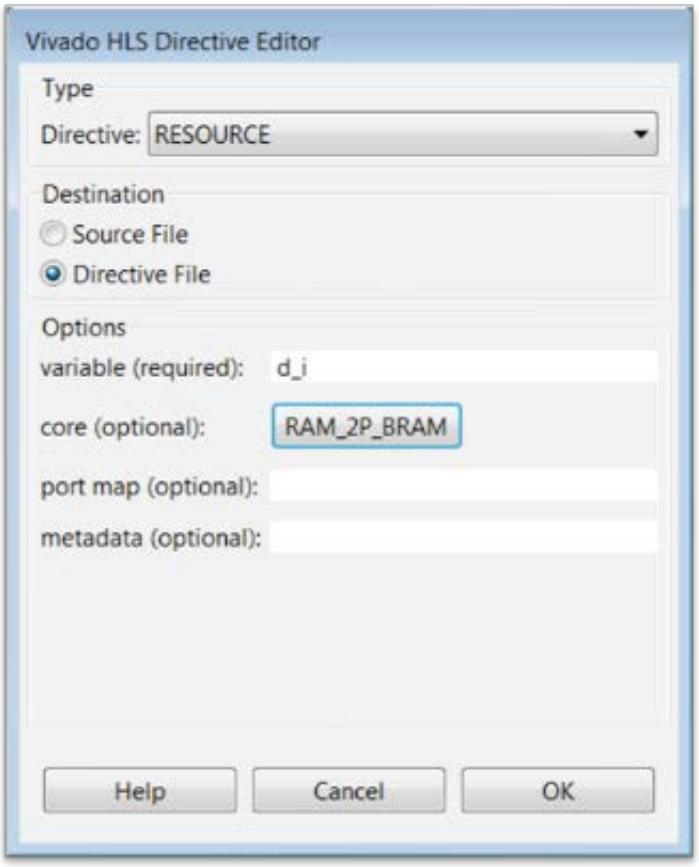


Figure 73 Directives Editor for Specifying a Dual-port RAM

For the output port, we will implement it using a FIFO interface.

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

The directives tab should show the following directives have been applied to the design (Figure 74).

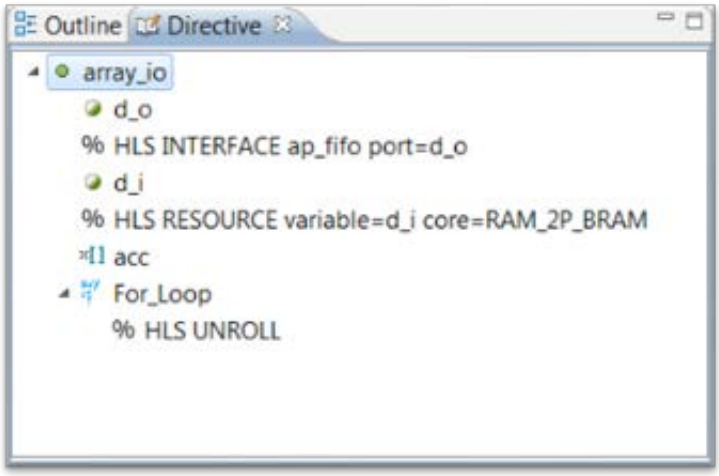
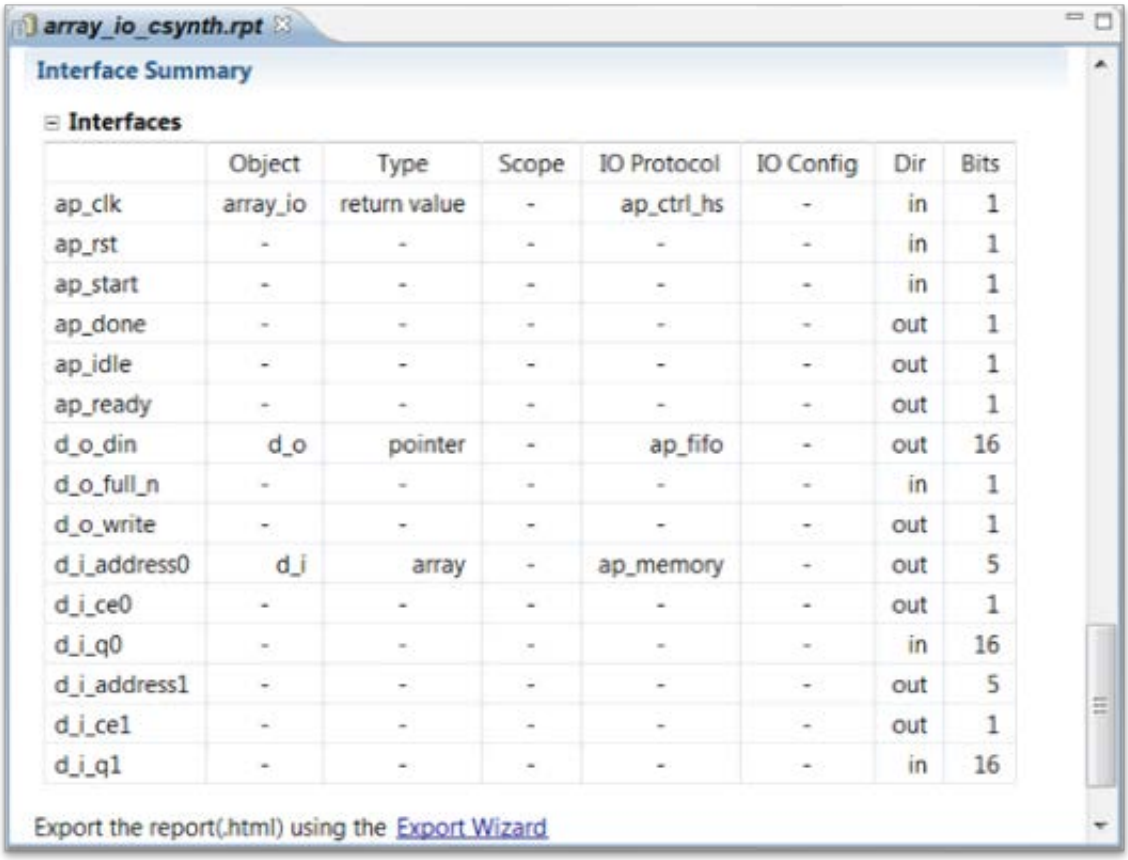


Figure 74 Directives Summary for Lab#2 Solution2

7. Synthesize the design.

When the report opens in the Information pane, the Interface summary will be as shown in Figure 75.

- The design has the standard clock, reset and block-level IO 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

Interfaces

	Object	Type	Scope	IO Protocol	IO Config	Dir	Bits
ap_clk	array_io	return value	-	ap_ctrl_hs	-	in	1
ap_rst	-	-	-	-	-	in	1
ap_start	-	-	-	-	-	in	1
ap_done	-	-	-	-	-	out	1
ap_idle	-	-	-	-	-	out	1
ap_ready	-	-	-	-	-	out	1
d_o_din	d_o	pointer	-	ap_fifo	-	out	16
d_o_full_n	-	-	-	-	-	in	1
d_o_write	-	-	-	-	-	out	1
d_i_address0	d_i	array	-	ap_memory	-	out	5
d_i_ce0	-	-	-	-	-	out	1
d_i_q0	-	-	-	-	-	in	16
d_i_address1	-	-	-	-	-	out	5
d_i_ce1	-	-	-	-	-	out	1
d_i_q1	-	-	-	-	-	in	16

Export the report(.html) using the Export Wizard

Figure 75 Directives Editor Specifying BRAM Interface

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

The next step is to show how array arguments can be partitioned into multiple ports.

Step 4: Partitioned RAM and FIFO Array interfaces

In this step, we will show how array interface can be partition into any arbitrary number of ports.

1. Select New Solution from the toolbar or the Project menu and create a new solution.
2. Press OK and accept the defaults.
    a. This includes copying existing directives from solution2.
3. Ensure the C source code is visible in the Information pane.
4. In the directives tab, select d_o and right-click with the mouse to open the Directives Editor dialog box.
    a. In the Directives Editor dialog box activate the Directives drop-down menu at the top and select ARRAY_PARTITION.
    b. Activate the type drop-down menu and select block to partition the array into blocks.
    c. In the Factor dialog box, enter the value 4.
    d. With the Directives Editor as shown in Figure 76, click OK.

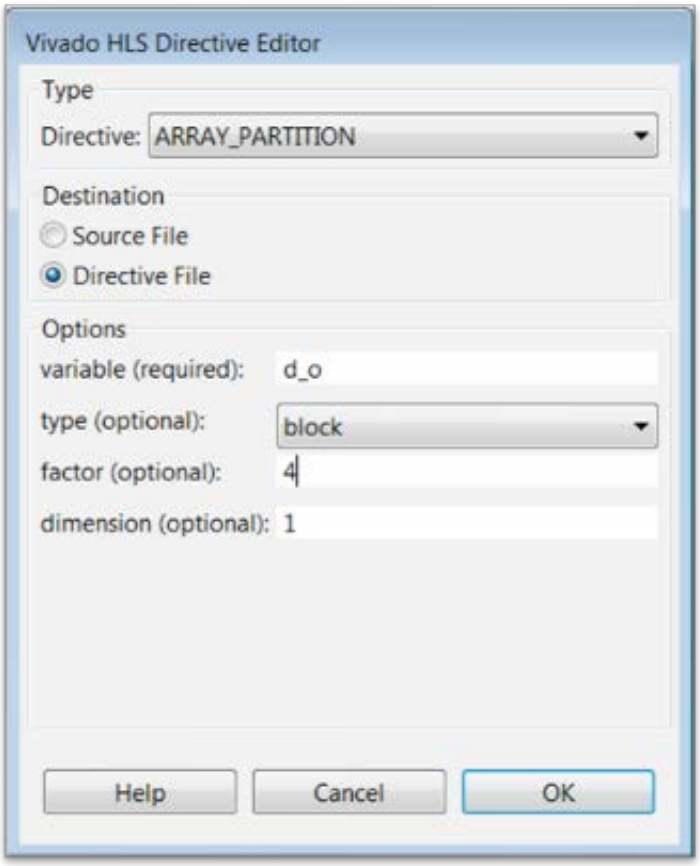


Figure 76 Directives Editor for Partitioning Array d_o

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

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

The directives tab should show the following directives have been applied to the design (77).

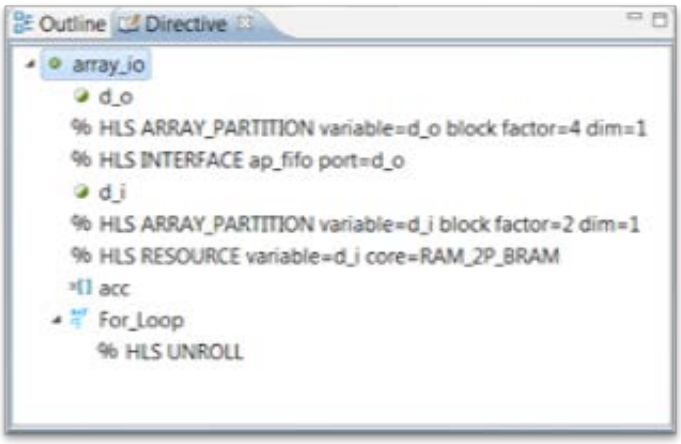
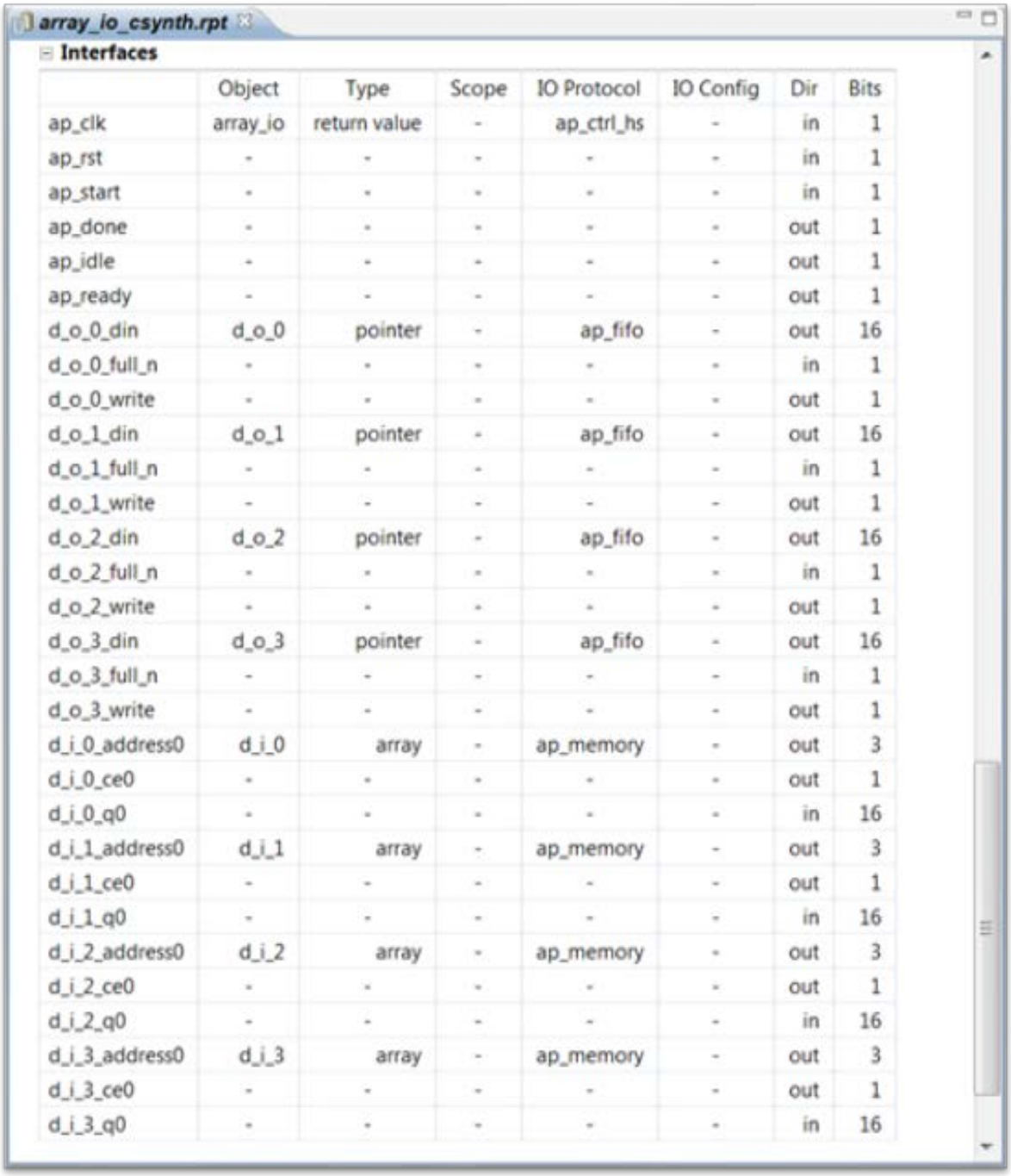


Figure 77 Directives Summary for Lab#2 Solution3

6. Synthesize the design.

When the report opens in the Information pane, the Interface summary will be as shown in Figure 78.

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



array_io_csynth.rpt

Interfaces

	Object	Type	Scope	IO Protocol	IO Config	Dir	Bits
ap_clk	array_io	return value	-	ap_ctrl_hs	-	in	1
ap_rst	-	-	-	-	-	in	1
ap_start	-	-	-	-	-	in	1
ap_done	-	-	-	-	-	out	1
ap_idle	-	-	-	-	-	out	1
ap_ready	-	-	-	-	-	out	1
d_o_0_din	d_o_0	pointer	-	ap_fifo	-	out	16
d_o_0_full_n	-	-	-	-	-	in	1
d_o_0_write	-	-	-	-	-	out	1
d_o_1_din	d_o_1	pointer	-	ap_fifo	-	out	16
d_o_1_full_n	-	-	-	-	-	in	1
d_o_1_write	-	-	-	-	-	out	1
d_o_2_din	d_o_2	pointer	-	ap_fifo	-	out	16
d_o_2_full_n	-	-	-	-	-	in	1
d_o_2_write	-	-	-	-	-	out	1
d_o_3_din	d_o_3	pointer	-	ap_fifo	-	out	16
d_o_3_full_n	-	-	-	-	-	in	1
d_o_3_write	-	-	-	-	-	out	1
d_i_0_address0	d_i_0	array	-	ap_memory	-	out	3
d_i_0_ce0	-	-	-	-	-	out	1
d_i_0_q0	-	-	-	-	-	in	16
d_i_1_address0	d_i_1	array	-	ap_memory	-	out	3
d_i_1_ce0	-	-	-	-	-	out	1
d_i_1_q0	-	-	-	-	-	in	16
d_i_2_address0	d_i_2	array	-	ap_memory	-	out	3
d_i_2_ce0	-	-	-	-	-	out	1
d_i_2_q0	-	-	-	-	-	in	16
d_i_3_address0	d_i_3	array	-	ap_memory	-	out	3
d_i_3_ce0	-	-	-	-	-	out	1
d_i_3_q0	-	-	-	-	-	in	16

Figure 78 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. Since the output port can only output four values at once, there would be no benefit in reading 8 inputs at once.

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

Step 5: Fully Partitioned Array interfaces

In this step, we will show how array interface can be partition into individual ports.

1. Select New Solution from the toolbar or the Project menu and create a new solution.
2. Press OK and accept the defaults.
    a. This includes copying existing directives from solution3.
3. Ensure the C source code is visible in the Information pane.
4. In the Directives tab, select the existing partition directive for d_o as shown in Figure 79
    a. Right-click with the mouse and select Modify Directive

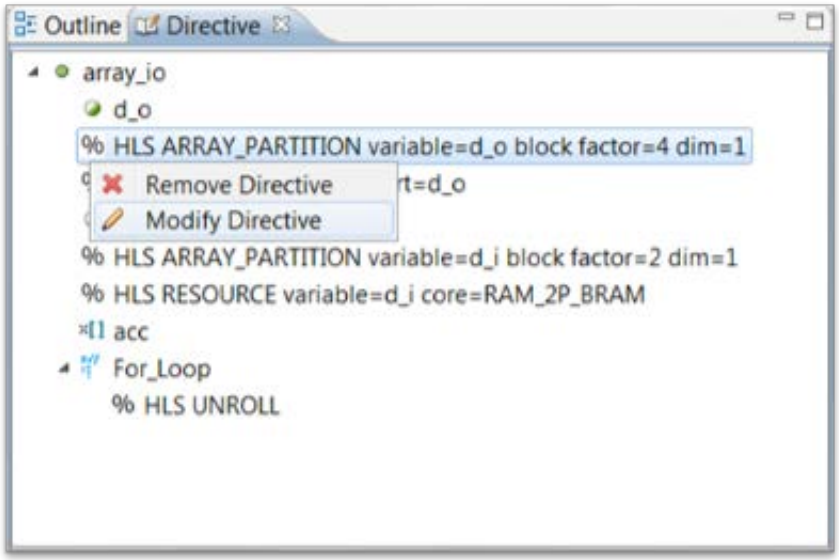


Figure 79 Modifying the Directive for d_o

5. In the Directives Editor dialog box.
    a. Activate the Type drop-down menu and modify the partitioning style to Complete.
    b. In the Factor dialog box, the value 4 can be removed or not: the factor will be ignored for this type of partitioning.
    c. With the Directives Editor as shown in Figure 80, click OK.

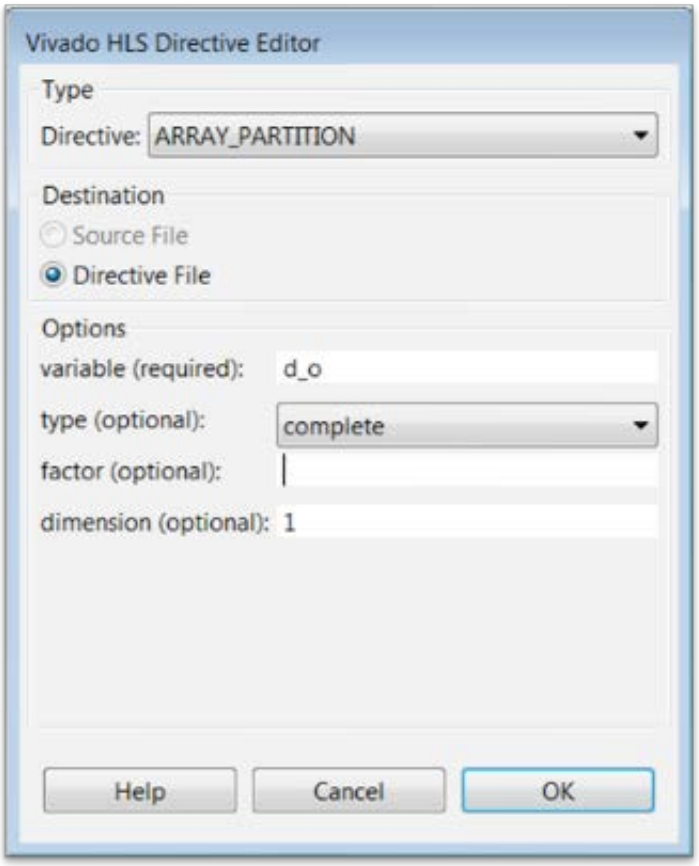


Figure 80 Directives Editor for Partitioning Array d_o

6. In the Directives tab, select d_i and repeat the previous step to completely partition the d_i array.
7. Optionally, the directive on d_i specifying the resource can be deleted: if the array is partitioned into individual elements the Resource directive, which specifies a RAM resource, will be ignored.

The Directives tab should show the following directives have been applied to the design (Figure 81).

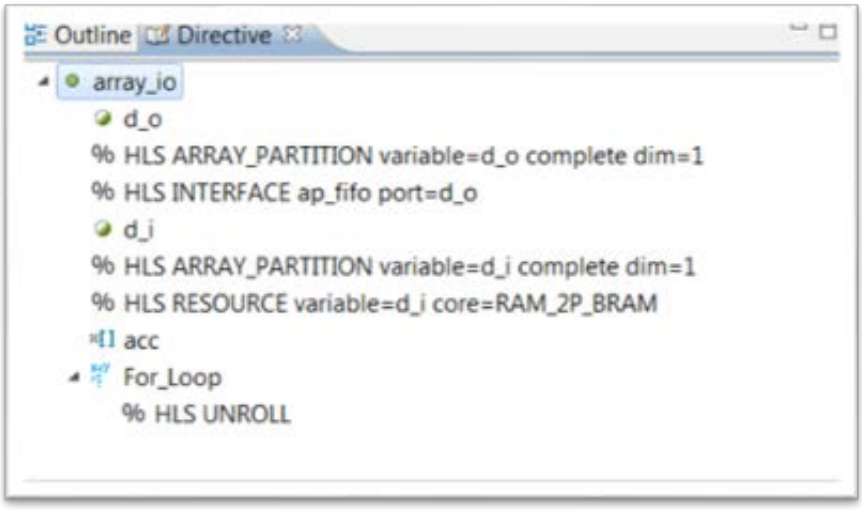


Figure 81 Directives Summary for Lab#2 Solution3

8. Synthesize the design.

When the report opens in the Information pane, review the interface summary.

- The design has the standard clock, reset and block-level IO ports.
- Array argument d_o has been implemented as a 32 separate FIFO interfaces.
- Argument d_i has been implemented as a 32 separate scalar port. Since the default interface for input scalars in no IO protocol, they have the IO protocol ap_none.

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

9. Select Compare Reports from the toolbar or the Project menu to open a comparison of the solutions.
10. In the Solution Selection dialog box, add each of the four solutions to the Selected Solutions pane (82).
11. Press OK.

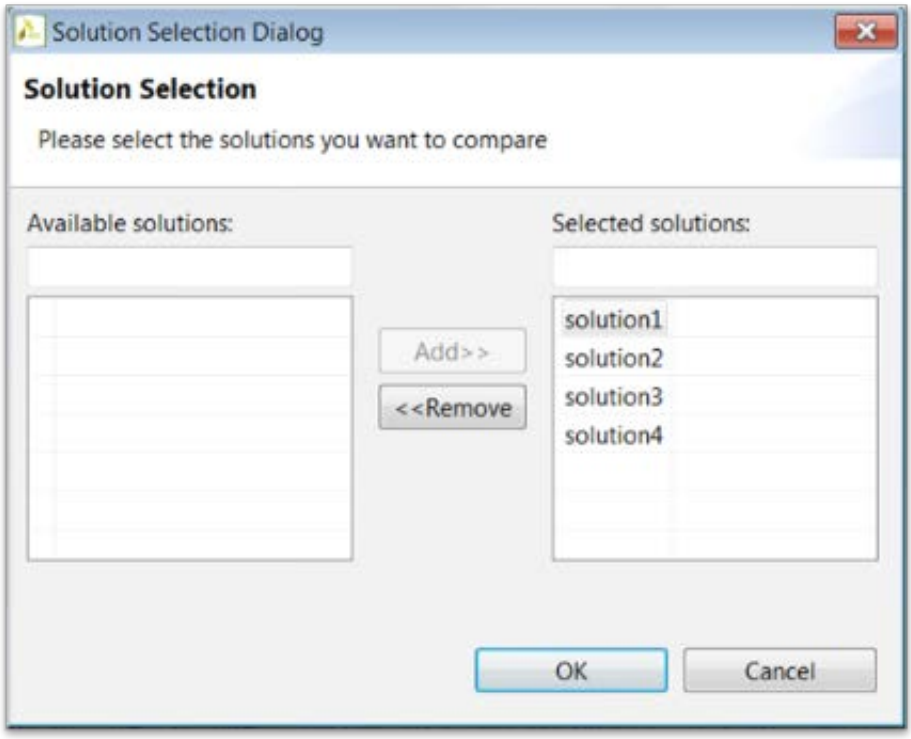
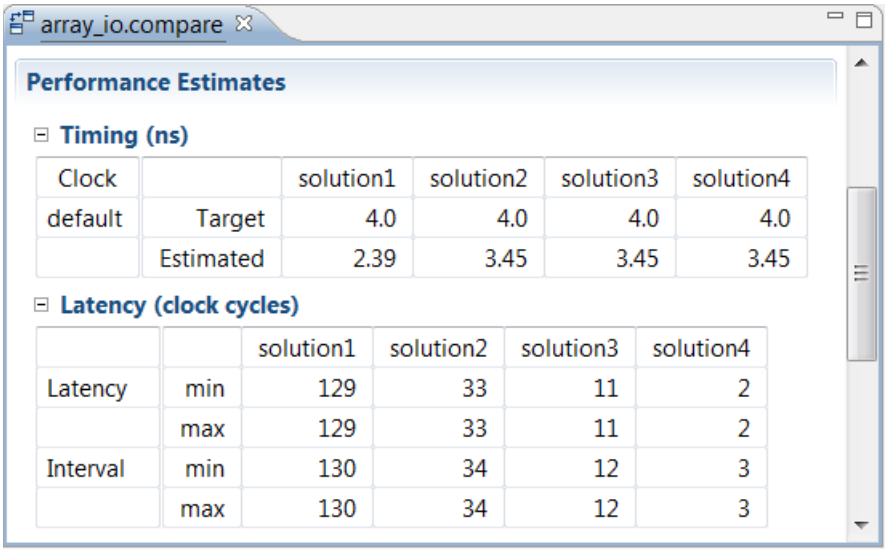


Figure 82 Compare All Solutions for Lab#3

When the solutions comparison report opens (Figure 83) it show that solution4, using a unique port for each array element, is much faster than any of the earlier solutions: the data can be accessed as soon as it is required by the internal logic (there is no performance bottleneck due to port accesses).

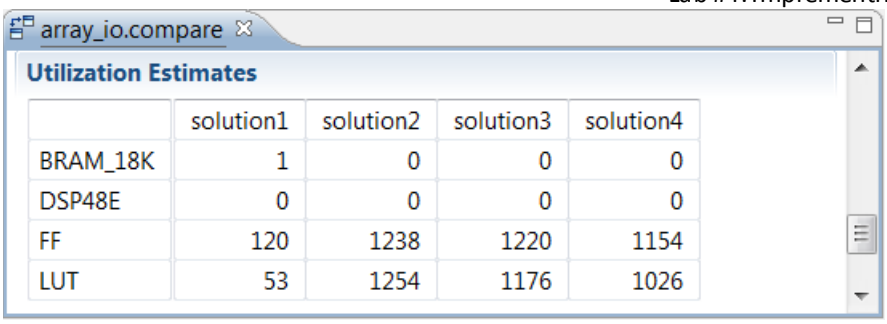


Clock		solution1	solution2	solution3	solution4
default	Target	4.0	4.0	4.0	4.0
	Estimated	2.39	3.45	3.45	3.45

		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 83 Performance Comparisons for All Lab#3 Solutions

Scrolling down further comparison report will show (Figure 84) that solutions with more IO ports (solutions 2, 3 and 4) allowing more parallel processing also use considerable more resources.



	solution1	solution2	solution3	solution4
BRAM_18K	1	0	0	0
DSP48E	0	0	0	0
FF	120	1238	1220	1154
LUT	53	1254	1176	1026

Figure 84 Resource Comparisons for All Lab#3 Solutions

In the next exercise, this same design will be implemented with an optimum balance of between the ports and resources. In addition to this more optimum implementation, the next exercise will show how AXI interfaces can be added to the design.

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

Lab #4: Implementing AXI Interfaces

This exercise will explain how AXI bus interfaces can be specified for the IO ports.

Step 1: Create and Open the Project

1. From the Vivado HLS command prompt window used in the previous lab, change to the lab4 directory.
2. Create a new Vivado HLS project by typing vivado_hls –f run_hls.tcl
3. Open the Vivado HLS GUI project by typing vivado_hls –p axi_interface_prj
4. Open the source code as shown in Figure 85.

This design uses similar source C code as Lab#3: the design has been simply been renamed to axi_interfaces. The explanation of this source code was already provided in Lab#3 but is provided again here:

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.

axi_interfaces.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 85 Source code for Lab#4

Step 2: Create an Optimized Design

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

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

Once the data has been partitioning into channels, FIFO interfaces can then be used to stream the samples for each channel through the design in parallel.

Finally, if the IO ports have been configured to supply and consume individual streams of channel data, partial unrolling of the for-loop can ensure dedicated hardware is used to process each channel.

First, partition the arrays.

1. Ensure the C source code is visible in the Information pane.
2. In the Directives tab, select d_o and right-click with the mouse to open the Directives Editor dialog box.
    a. Activate the Directives drop-down menu at the top and select ARRAY_PARTITION.
    b. Use the Type drop-down menu to specify cyclic partitioning.

c. In the Factor dialog box, enter the value 8, to create 8 separate partitions (this will result in 8 ports).
d. With the Directives Editor as shown in Figure 86, press OK.

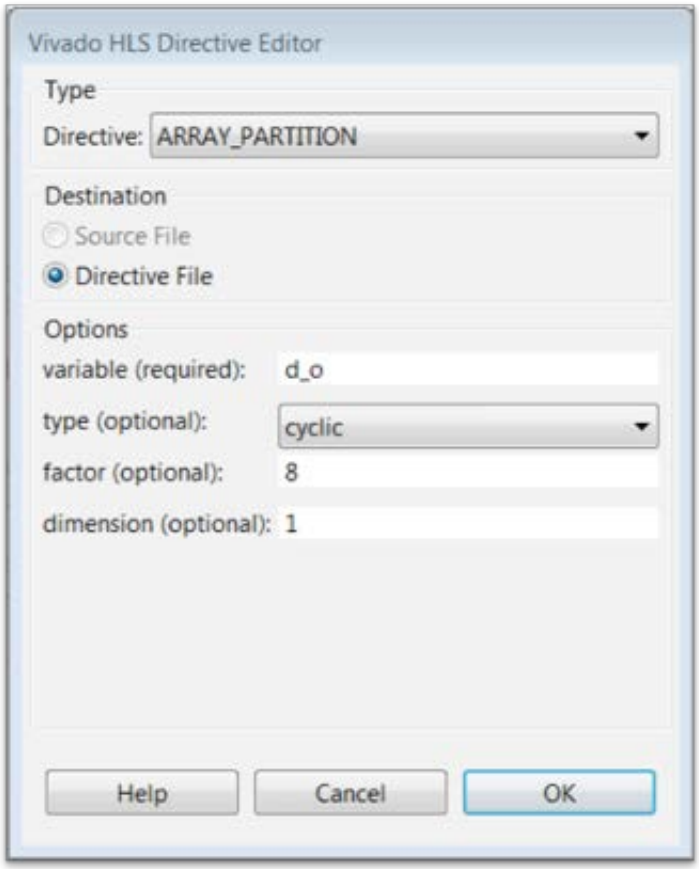


Figure 86 Directives Editor for Cyclic Partitioning

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

Next, partially unroll and pipeline the for-loop.

5. In the Directives tab, select for-loop For_Loop and right-click with the mouse to open the Directives Editor dialog box.
   f. Activate the Directives drop-down menu at the top and select UNROLL.
   g. Select a factor of 8 to partially unroll the for-loop. This is equivalent to re-writing the C code to execute 8 copies of the loop-body in each iteration of the loop (where the new loop will only execute for 4 iterations in total, not 32).
   h. Click OK.

6. In the Directives tab, select for-loop For_Loop again and right-click with the mouse to open the Directives Editor dialog box.
    i. Activate the Directives drop-down menu at the top and select PIPELINE.
    j. Leave the Interval blank and let it default to 1.
    k. Select enable loop rewinding.
    l. Click OK.

The pipeline rewind option can be used when the top-level of the design is a loop. This informs Vivado HLS that when implemented in RTL, this loop is expected to run forever (with no end of function and function re-start cycles).

After performing the above steps, the Directives tab should be as shown in Figure 87. Be sure to check all options are correctly applied. If not, double-click on the directive to re-open the Directives Editor.

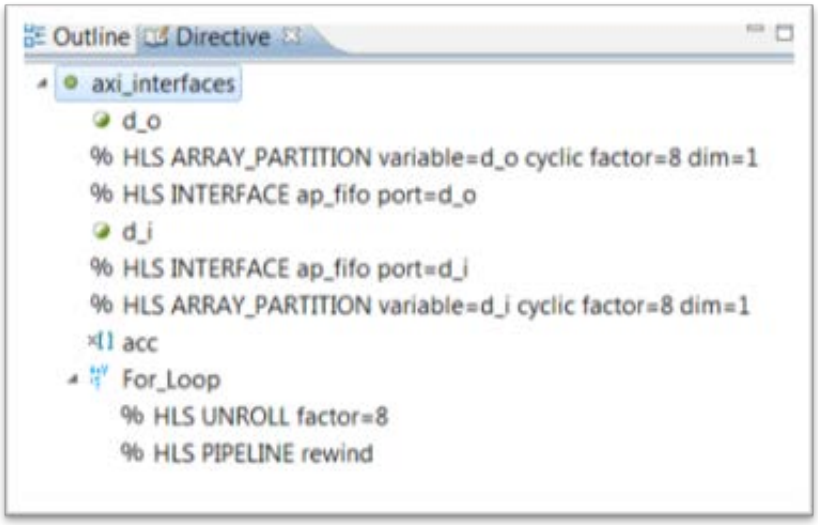


Figure 87 Directives tab for Lab#4 Solution1

7. Synthesize the design.

When the report opens in the information pane, confirm both d_i and d_o are implemented as 8 separate FIFO ports.

In the performance section of the design, confirm for-loop For_Loop processes 1 sample every clock cycle (Interval 1) with a latency of 2 and that the design has less area than solutions 2, 3 or 4 in Lab#3 (Figure 84).

Cyclic partitioning of the array interfaces and partial unrolling has indeed allowed us to implement this C code as 8 separate channels in the hardware.

Step 3: Implementing AXI4 Interfaces

AXI4 interfaces are added as a two-step process:

- The interface is specified to have an IO protocol using the Interface directive.

- A Resource directive is then added to the RTL port to specify that an AXI4 interface will connect to the port. (Similar to how RAM interfaces are specified). The AXI4 interfaces are added to the design during the IP creation stage.

In this exercise, the FIFO interfaces will be specified to be AXI4 Stream interfaces and the block-level IO protocol ports will be grouped into a single AXI4 Lite interface allowing these block-level control signals be to controlled and accessed from a CPU.

1. Select New Solution from the toolbar or the Project menu to create and new solution.
2. Press OK and accept the defaults.
    a. This includes copying existing directives from solution3.
3. Ensure the C source code is visible in the Information pane.
4. In the Directives tab, select d_o and right-click with the mouse to open the Directives Editor dialog box.
    a. Activate the Directives drop-down menu at the top and select RESOUCE.
    b. Click on the Core options box and select AXI4Stream. This indicates an AXI4Stream interface is connected to this interface.
    c. Click OK.
5. Repeat step4 with d_i.
6. In the Directives tab, select the top-level function axi_interfaces and right-click with the mouse to open the Directives Editor dialog box.
    a. Activate the Directives drop-down menu at the top and select RESOUCE.
    b. Since the axi_interfaces function was selected, the variable field is automatically completed with the function return.
    c. Click on the Core options box and select AXI4LiteS. This specifies the ports associated with the function return (the block-level IO ports) are connected to an AXI4Lite interface.
    d. Click OK.

The Directives tab should be as shown in Figure 88.

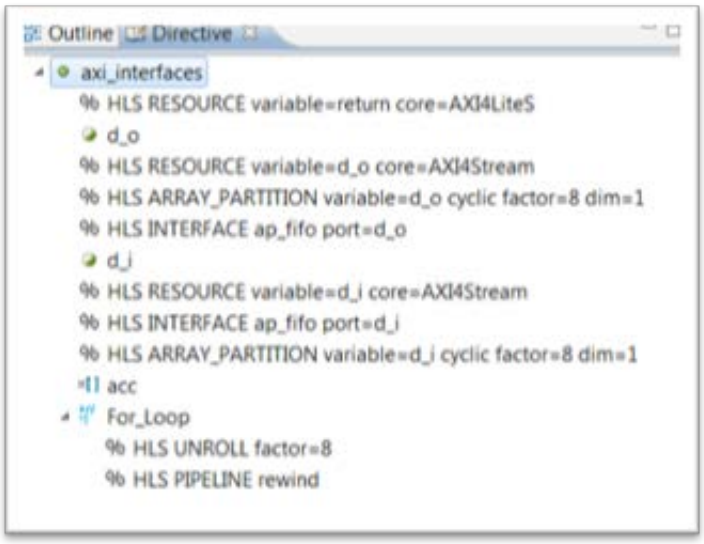


Figure 88 Directives for Specifying AXI4 Interfaces

7. Synthesize the design.

When the report opens, the Interface summary will not show any AXI4 interfaces. AXI4 interfaces are added to the design when it is packaged as IP. Only the RTL ports are shown in the Interface summary.

8. Select Export RTL from the toolbar or the Solution menu, to create an IP package.
9. Leave the Format Selection as IP Catalog and press OK.

The IP package can be seen in the solution impl folder (Figure 89): since the Vivado IP Catalog format was used, the package is located in the ip folder.

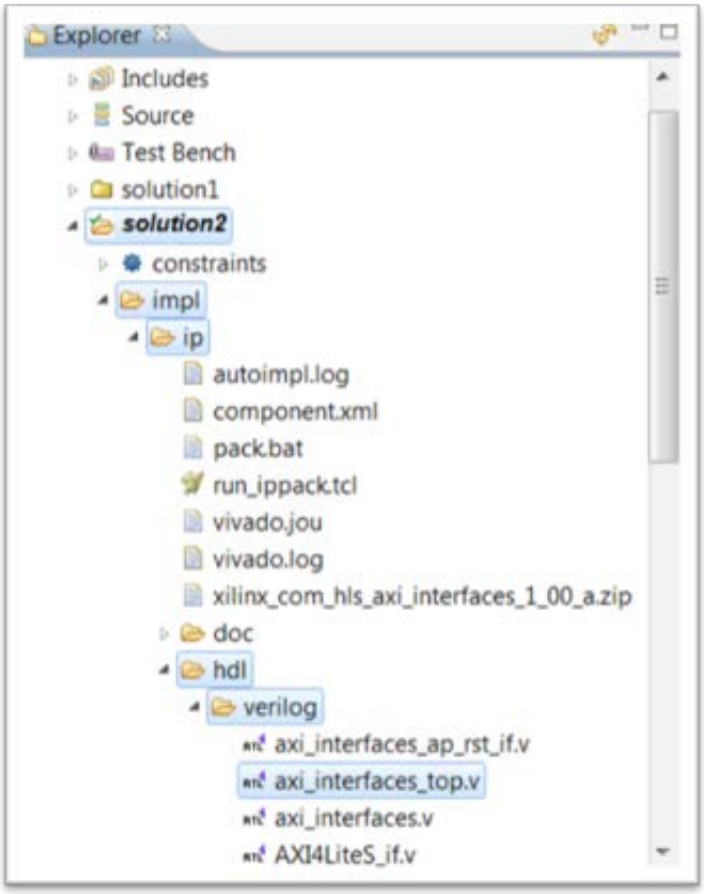


Figure 89 IP Package with AXI4 Interfaces

The top-level HDL can be located in the verilog sub-folder, also shown in the Figure 89 and is identified by the function name appended with _top: in this case, the top-level file is axi_interfaces_top.v

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

10. Double-click on file axi_interfaces_top.v to open the file in the information pane.

Reviewing the top-level HDL file shows that AXI4 interfaces have been added. 90 shows an AXI4 Lite interface and the first of 16 AXI4 Stream interfaces.

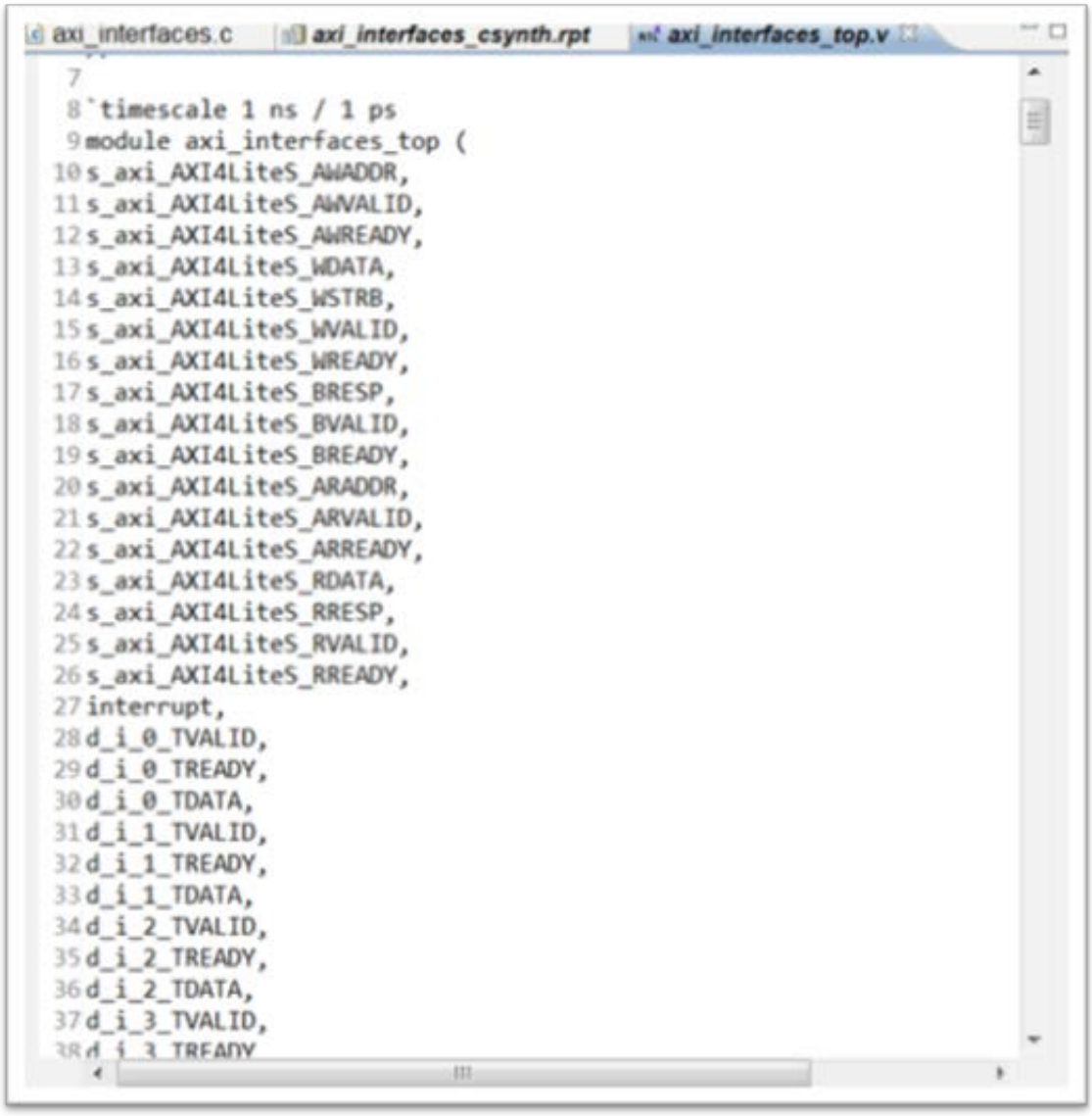


Figure 90 IP HDL with AXI4 Interfaces

This design was synthesized with an AXI4 Lite interface (for the block-level protocol ports). When an AXI4 Lite interface is added to the design, software driver files are created during IP packaging to access this interface from a CPU.

Figure 91 shows the software drivers created in the impl directory. One of the files is shown open in the Information pane.

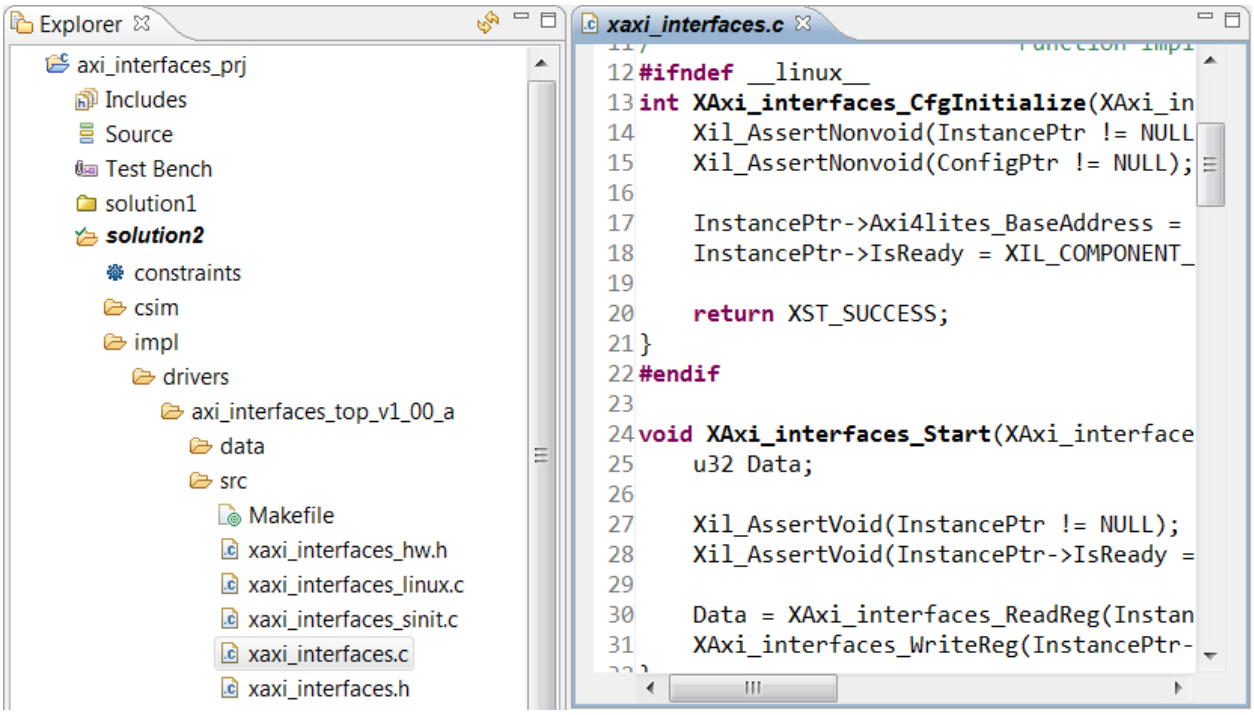


Figure 91 IP Software Driver Files

This completes this tutorial on interface synthesis.

Conclusion

In this tutorial, you learned:

- What block-level IO protocols are and how to control them.
- How to specify and apply port-level IO protocols.
- How array ports can be specified as RAM interfaces, FIFO interfaces and partitioned into sub-ports.
- Finally, how use both IO directives and synthesis directives to create an optimal design with AXI4 interfaces.

Arbitrary Precision Types

Overview

The data types provided by C/C++ 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. An example is a case where 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 result in unnecessary hardware costs: operations can use more LUTs and registers than needed for the required accuracy and delays may even exceed the clock cycle, requiring more cycles to compute the result.

Vivado High-Level Synthesis provides a number of bit-accurate or arbitrary precision data-types allowing variables to be modeled using any (arbitrary) width.

This tutorial consists of a two lab exercises.

Lab1

Synthesize a design using floating-point types and review the results.

Lab2

Synthesize the same function used in Lab#1 using arbitrary precision fixed-types highlighting the benefits in accuracy and results.

Tutorial Design Description

The tutorial design file can be downloaded from the Xilinx website. Refer to the information in Obtaining the Tutorial Designs.

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

The sample designs used in the two exercises in this tutorial windowing function. In the first lab exercise, the design uses standard C++ floating-point types. The second lab 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.

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

This exercise will synthesize a design using standard C types. This will be used as a reference for the design using arbitrary precision types which is used in 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 2013.1** > **Vivado HLS** > **Vivado HLS 2013.1 Command Prompt (**Figure 92).
   b. On Linux, open a new shell.

Figure 92 Vivado HLS Command Prompt

2. Using the command prompt window (Figure 93), change directory to the Arbitrary Precision 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 93.

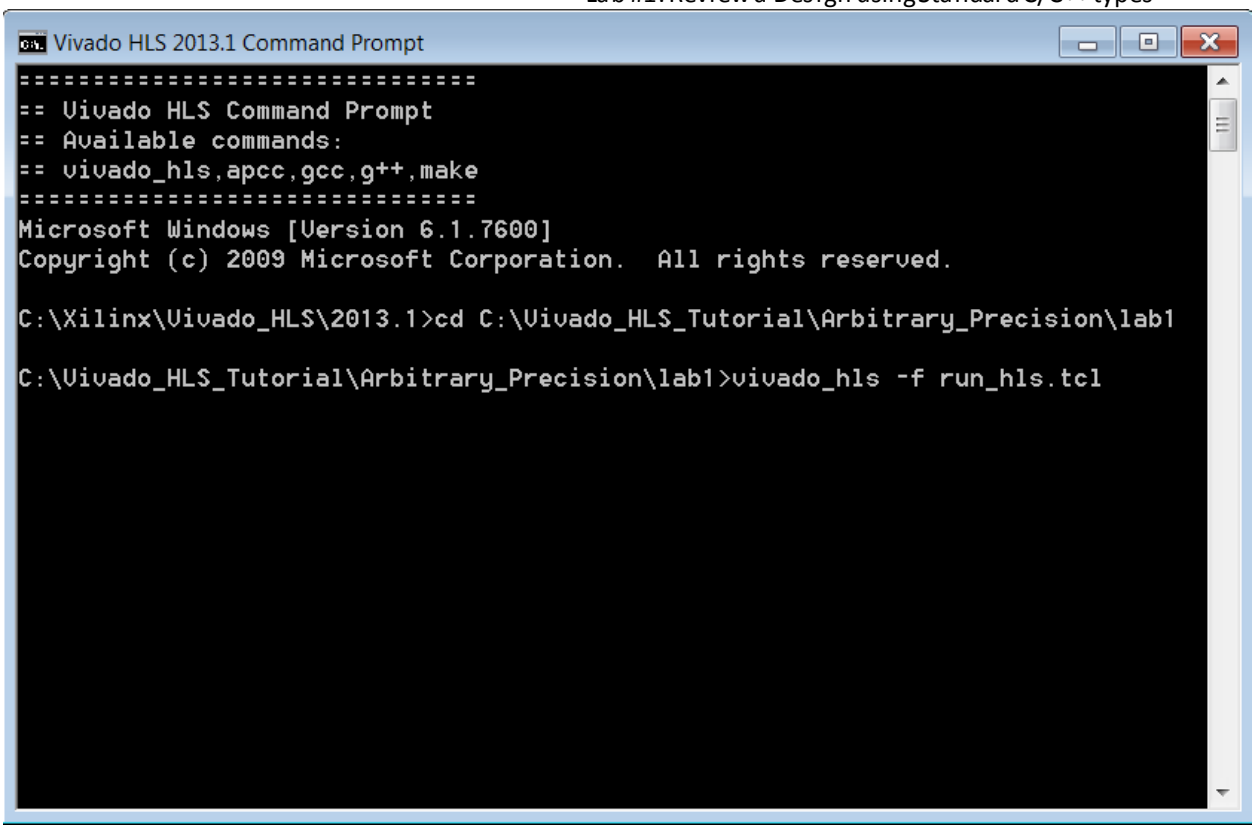


Figure 93 Setup the Tutorial Project

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 94.

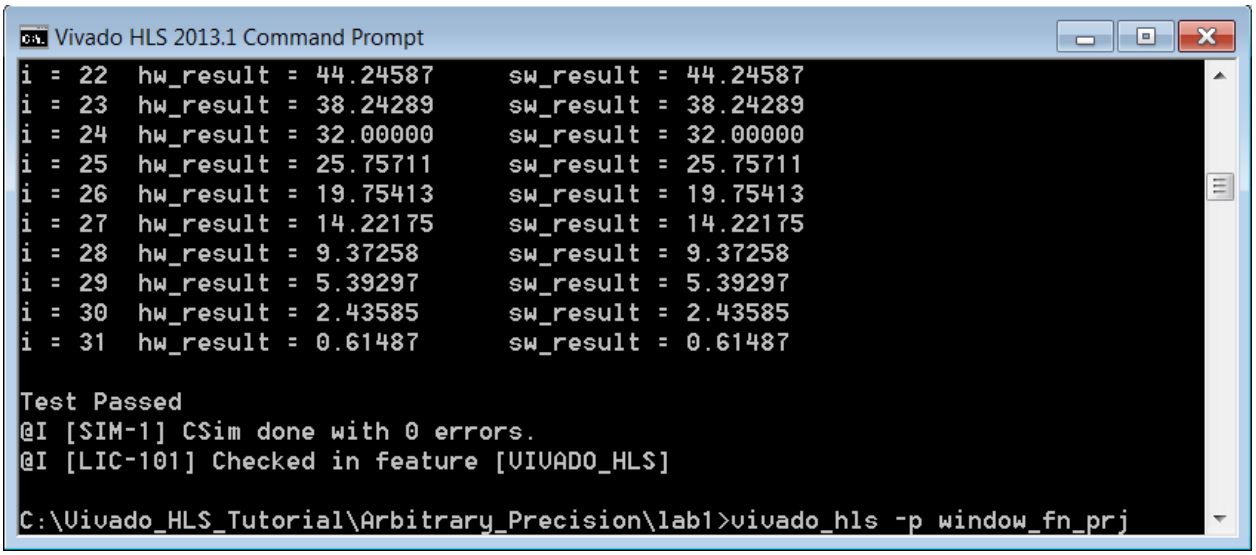


Figure 94 Initial Project for Arbitrary Precision Lab#1

Step 2: Review Test Bench and Run C Simulation

1. Open the Source folder in the explorer pane and double-click on window_fn_top.cpp to open the code as shown in Figure 95.

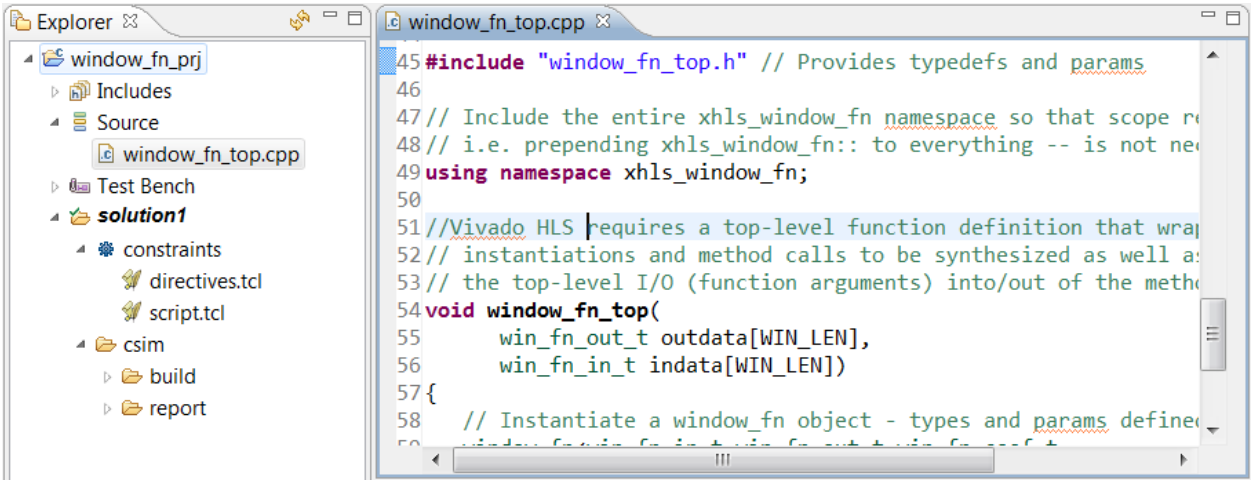


Figure 95 C Code for C Validation Lab#3

2. Hold the Control key down and click on the window_fn_top.h on line 45 to open this header file.
3. Scroll down to view the type definitions (Figure 96).

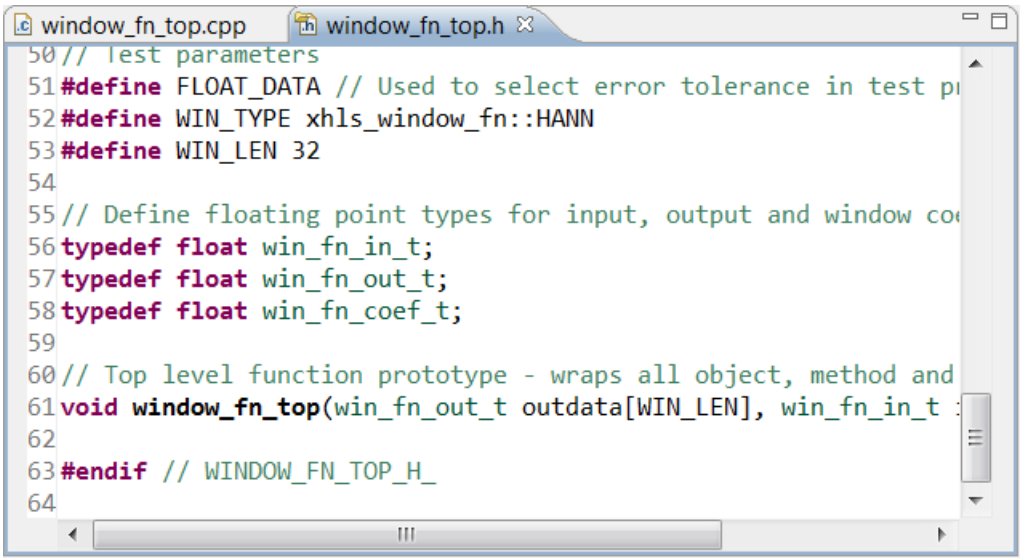


Figure 96 Type Definitions for C Validation Lab#3

This design uses standard C/C++ floating-point types for all data operations. Vivado High-Level Synthesis can directly synthesize floating-point types into hardware provided the operations are standard arithmetic operations (+, -, *, % etc.) When using math functions from math.h or cmath.h, refer to the Vivado HLS User Guide (ug902) for details on which math functions are supported for synthesis.

4. Use the Run C Simulation toolbar button or menu Project > Run C Simulation to open the C Simulation Dialog box
5. Accept the default setting (no options selected) and press OK.

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

Step 3: Synthesize the Design and Review Results

1. Use the Run C Synthesize toolbar button or menu Solution > Run C Synthesis to synthesizes the design to RTL.

When synthesis completes, the synthesis report opens automatically. Figure 97 shows the synthesis report.

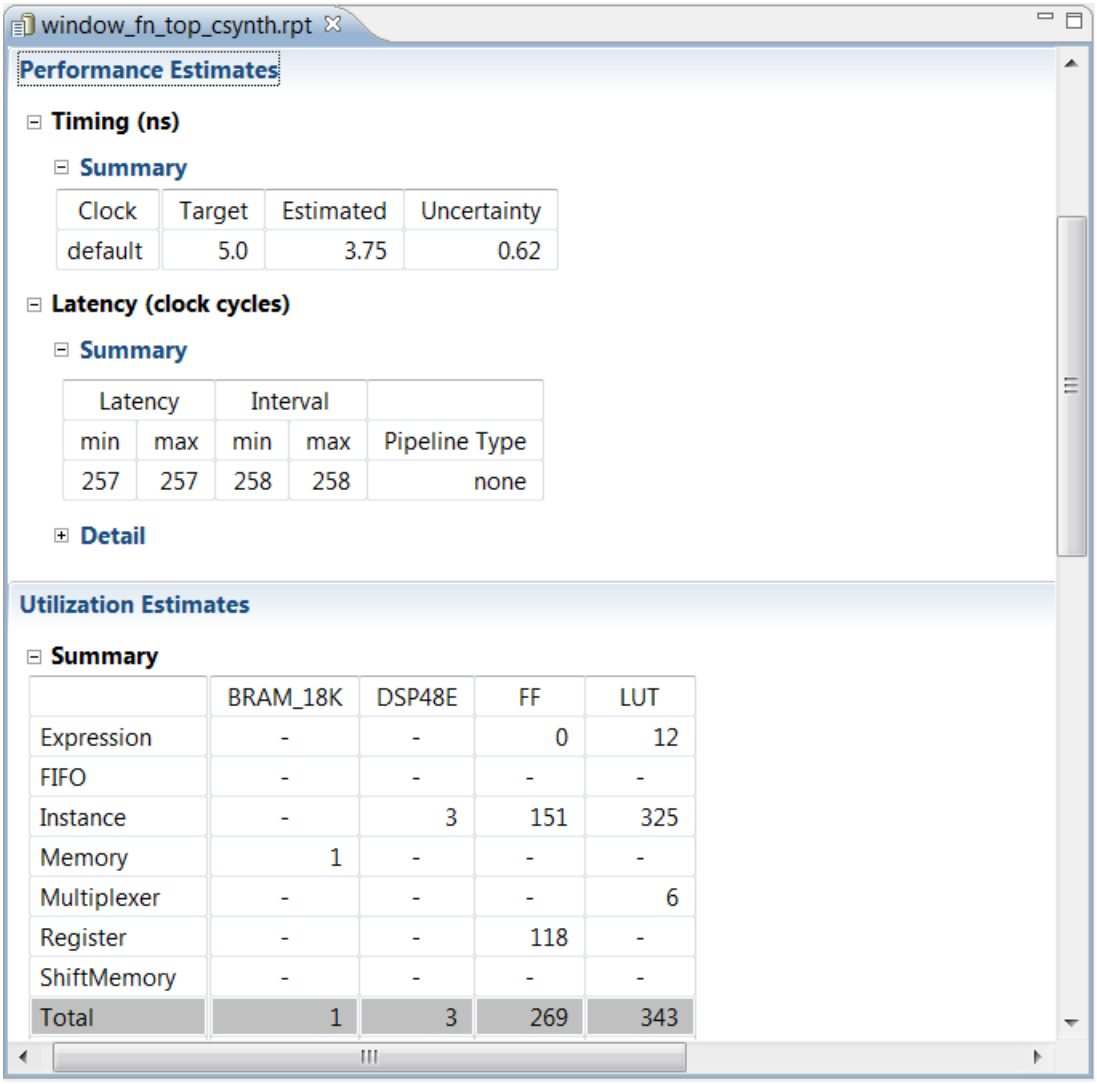


Figure 97 Synthesis Report for Floating Point Design

Most of the area is due to Instances in the top-level design.

2. Scroll down the report and expand the Instances in the Details section of the Area Estimates (Figure 98).

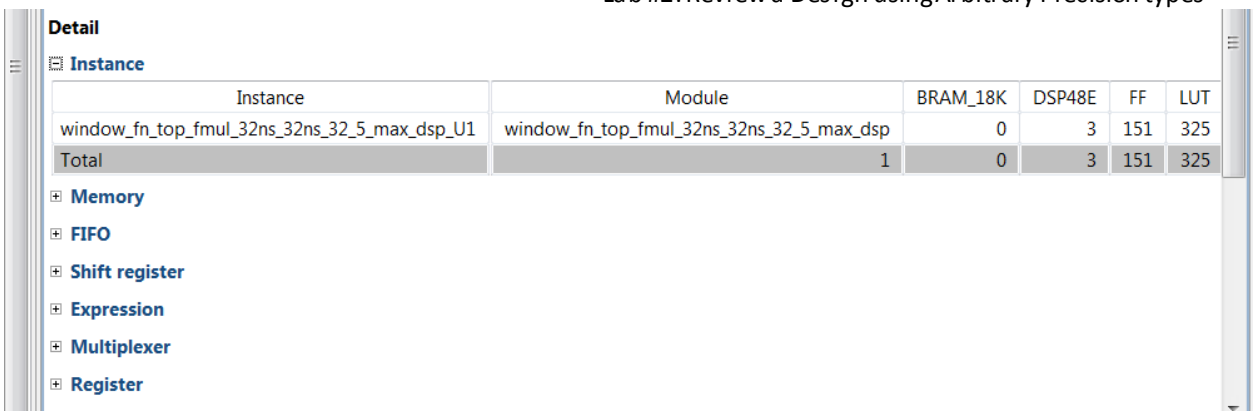


Figure 98 Area Details for Floating Point Design

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 99) shows this operator is also responsible for most of the clock cycles (5 out of the 8 states it takes to execute the logic created by loop winfn).

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

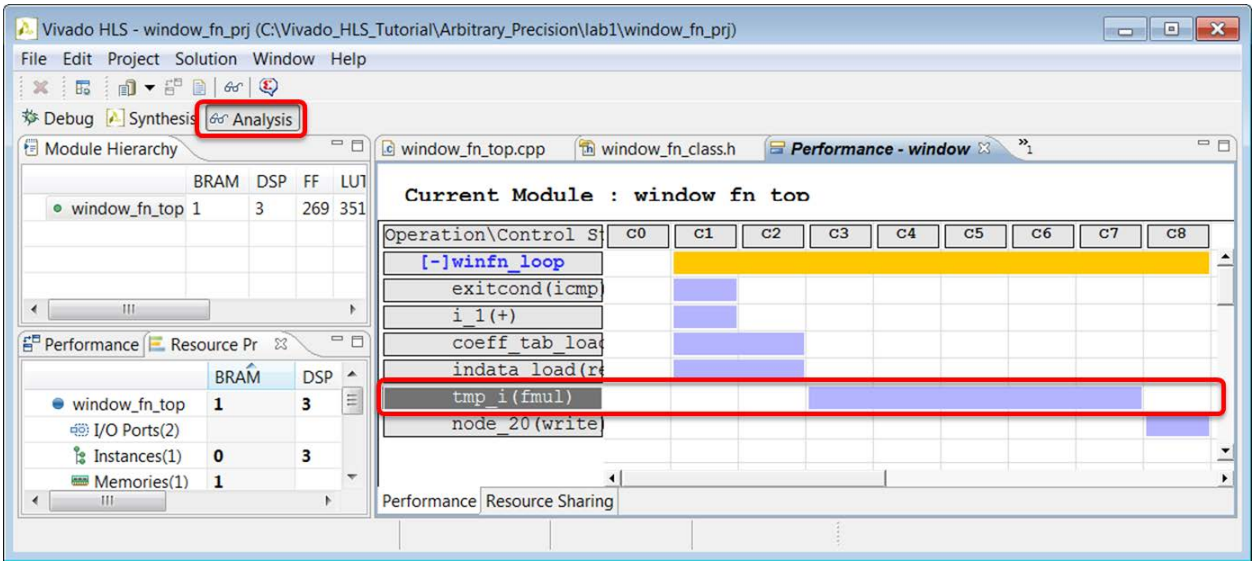


Figure 99 Performance Details for Floating Point Design

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

Lab #2: Review a Design using Arbitrary Precision types

This lab exercise uses the same design as Lab#1 however the data types are now arbitrary precision types. The design will first be reviewed and then the results of synthesis examined.

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 100.
2. Create a new Vivado HLS project by typing vivado_hls –f run_hls.tcl

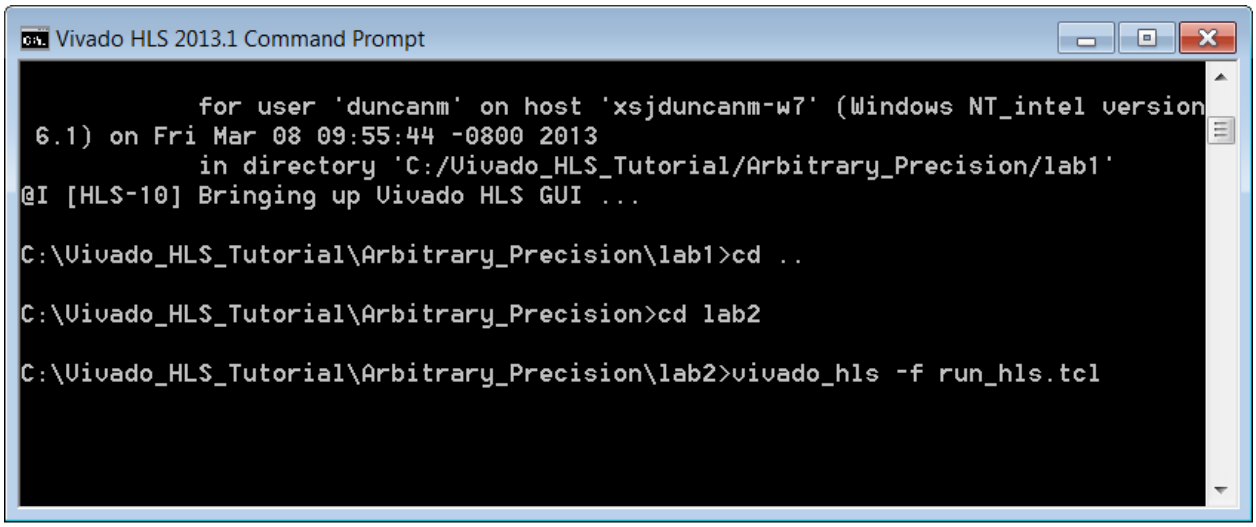


Figure 100 Setup for Interface Synthesis Lab#2

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 on window_fn_top.cpp to open the code as shown in Figure 101.

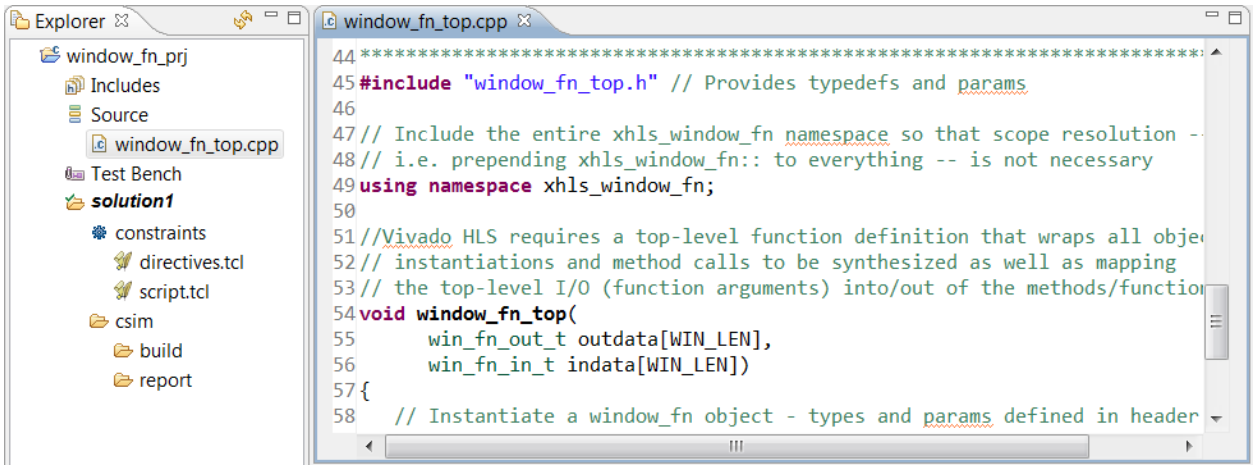


Figure 101 C Code for Arbitrary Precision Lab#2

5. Hold the Control key down and click on the window_fn_top.h on line 45 to open this header file.
6. Scroll down to view the type definitions (Figure 102).

window_fn_top.cpp | window_fn_top.h

```
54 // Types and top-level function prototype
55 #include <ap_int.h>
56 // Define widths of fixed point fields
57 #define W_IN    8
58 #define IW_IN   8
59 #define W_OUT   24
60 #define IW_OUT  8
61 #define W_COEF  18
62 #define IW_COEF 2
63
64 // Define fixed  point types for input, output and coefficients
65 typedef ap_fixed<W_IN,IW_IN> win_fn_in_t;
66 typedef ap_fixed<W_OUT,IW_OUT> win_fn_out_t;
67 typedef ap_fixed<W_COEF,IW_COEF> win_fn_coef_t;
68
```

Figure 102 Type Definitions for Arbitrary Precision Lab#2

This header file, window_fn_top.h, is the only file which is different from Lab#1. The data types have been changed to ap_fixed point types which like float and double types 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 changing C code from using standard C types to using arbitrary precision types one of the most common changes is to reduce the size of the data types. In this case the design is changed from 32-bit float types to use 8-bit, 24-bit and 18-bit words. This results in smaller operators, reduced area and faster timing.

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

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

7. Open the Test Bench folder in the Explorer pane and double-click on window_fn_top_test.cpp to open the code.
8. Scroll down to see the view shown in Figure 103.

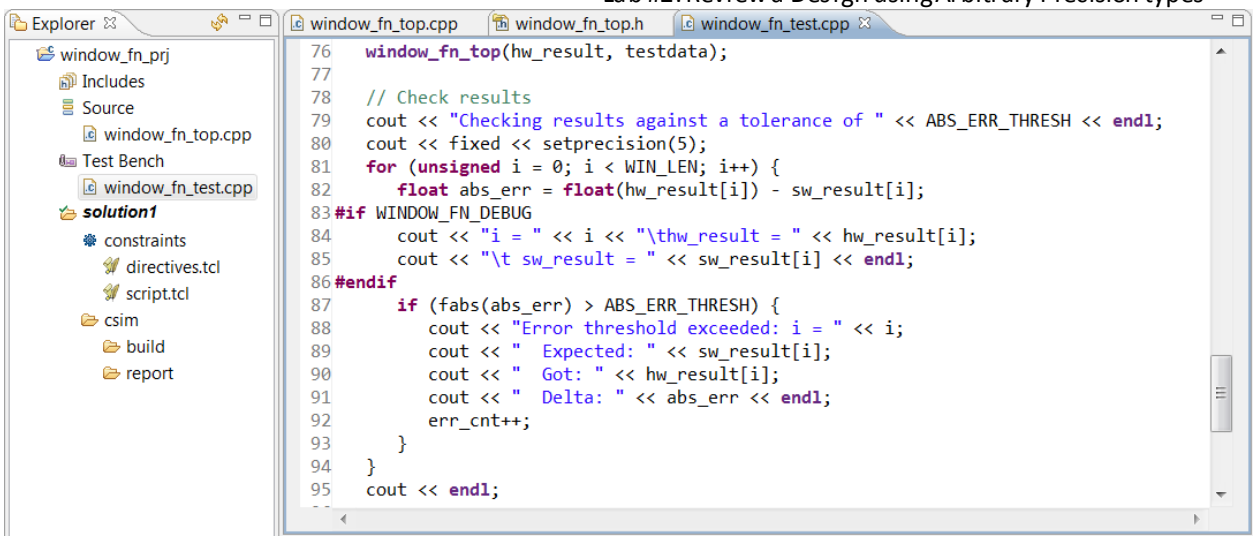


Figure 103 Test Bench for Arbitrary Precision Lab#2

The test bench used in this design contains code to check the accuracy of the results. The expected results are still generated using float types. The result checking checks 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 quickly and efficiently validated in C, with fast compile and run times.

9. Use the Run C Simulation toolbar button or menu Project > Run C Simulation to open the C Simulation Dialog box
10. Accept the default setting (no options selected) and press 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.

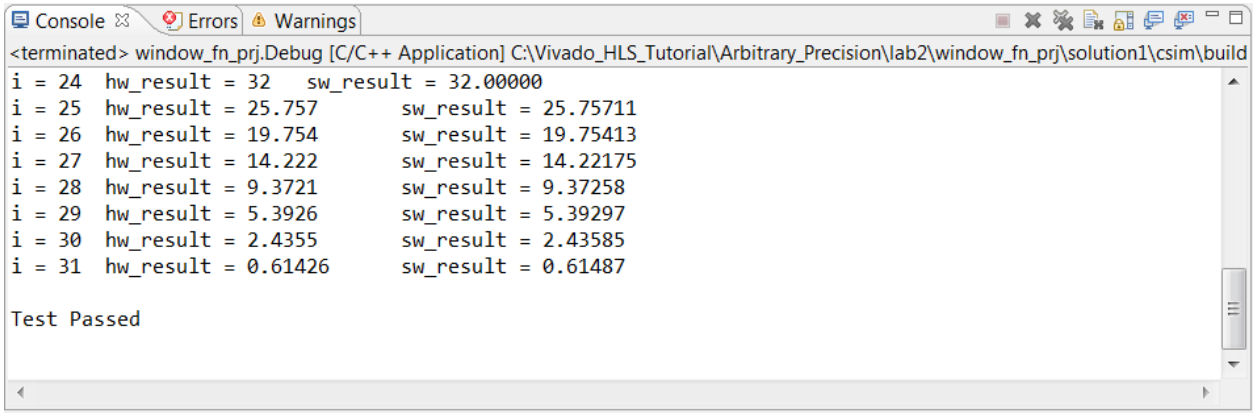


Figure 104 C Simulation Results for Fixed Point Types

Step 2: Synthesize the Design and Review Results

1. Use the Run C Synthesize toolbar button or menu Solution > Run C Synthesis to synthesizes the design to RTL.

When synthesis completes, the synthesis report opens automatically. Figure 105 shows the synthesis report.

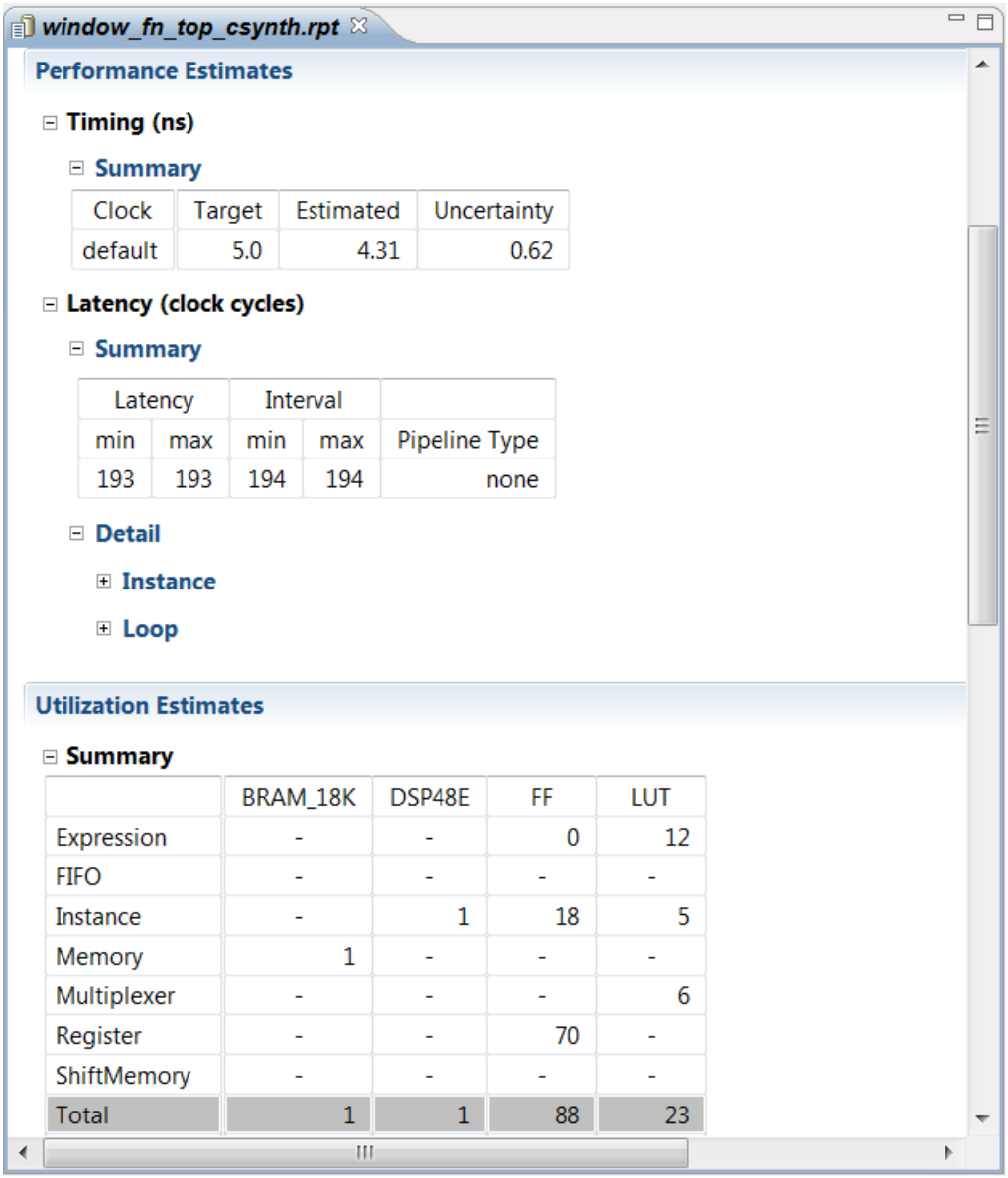
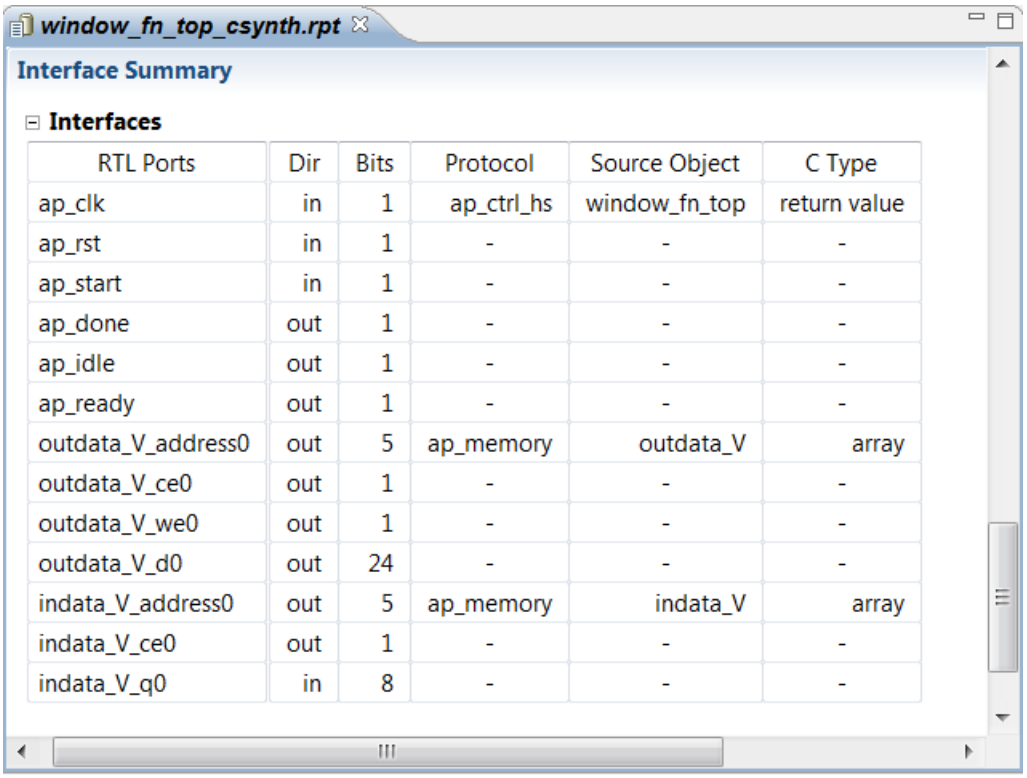


Figure 105 Synthesis Report for Fixed Point Design

Both the latency and the area have been reduced (by 25% and 60% respectively) by using arbitrary precision types: the operations in the RTL hardware are only as large as they need to be.

2. Scroll down the report to the Interface summary (Figure 106).

Figure 106 shows the data ports are now 8-bit and 24-bit.


window_fn_top_csynth.rpt

Interface Summary

Interfaces

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_start	in	1	-	-	-
ap_done	out	1	-	-	-
ap_idle	out	1	-	-	-
ap_ready	out	1	-	-	-
outdata_V_address0	out	5	ap_memory	outdata_V	array
outdata_V_ce0	out	1	-	-	-
outdata_V_we0	out	1	-	-	-
outdata_V_d0	out	24	-	-	-
indata_V_address0	out	5	ap_memory	indata_V	array
indata_V_ce0	out	1	-	-	-
indata_V_q0	in	8	-	-	-

Figure 106 Fixed Point Interface Summary

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

This completes this tutorial on using arbitrary precision types.

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.

Design Analysis

Overview

The general design methodology for creating an RTL implementation from C, C++ or SystemC is to

- Synthesize the design.
- Review the results of the initial implementation.
- Then apply optimization directives to improve performance.
- This process can be repeated until the required performance is achieved.
- Once performance has been achieved, revisit the design to improve area.

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

This tutorial consists of a single lab exercise in which one design is taken from the initial implementation through six steps to produce the final optimized design. As will be demonstrated throughout the tutorial, an advantage of performing this in a single project is the ability to easily compare the different solutions.

This tutorial consists of a single lab exercises that steps a design through multiple optimizations to highlight operation of the Vivado HLS interactive analysis feature.

Lab1

Synthesis 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

The tutorial design file can be downloaded from the Xilinx website. Refer to the information in Obtaining the Tutorial Designs.

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

The sample designs used in the lab exercise is a 2-D DCT function. To highlight the design analysis feature, the 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 will explain the basic operations of the GUI Analysis perspective and how it can be used 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 on Linux systems, adjust the few pathnames referenced, to the location you have chosen to place the ***Vivado_HLS_Tutorial*** *directory.*

Step 1: Create and Open the Project

1. Open the Vivado HLS Command Prompt.
    a. On Windows use **Start** > **All Programs** > **Xilinx Design Tools** > **Vivado 2013.1** > **Vivado HLS** > **Vivado HLS 2013.1 Command Prompt (**Figure 107).
    b. On Linux, open a new shell.

Figure 107 Vivado HLS Command Prompt

2. Using the command prompt window (Figure 108), change 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 108.

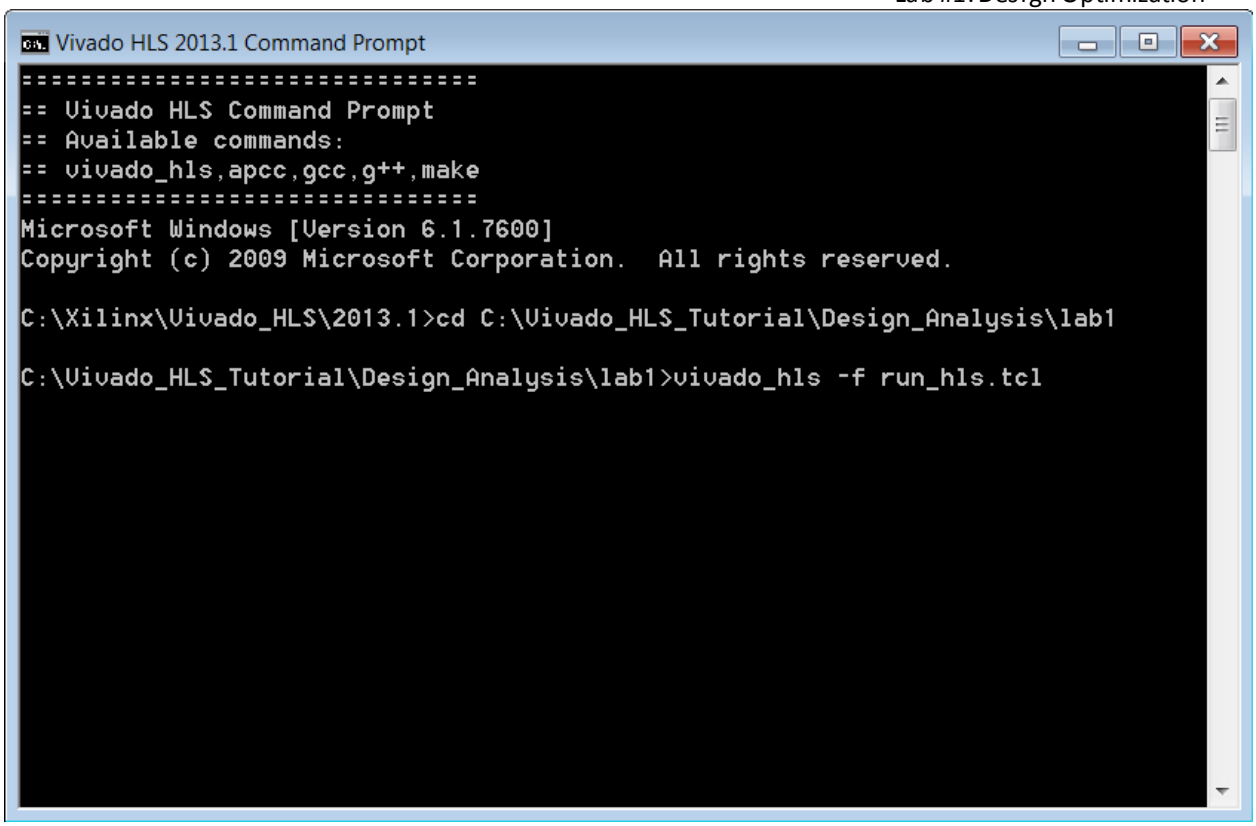


Figure 108 Setup the Design Analysis Tutorial Project

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 109.

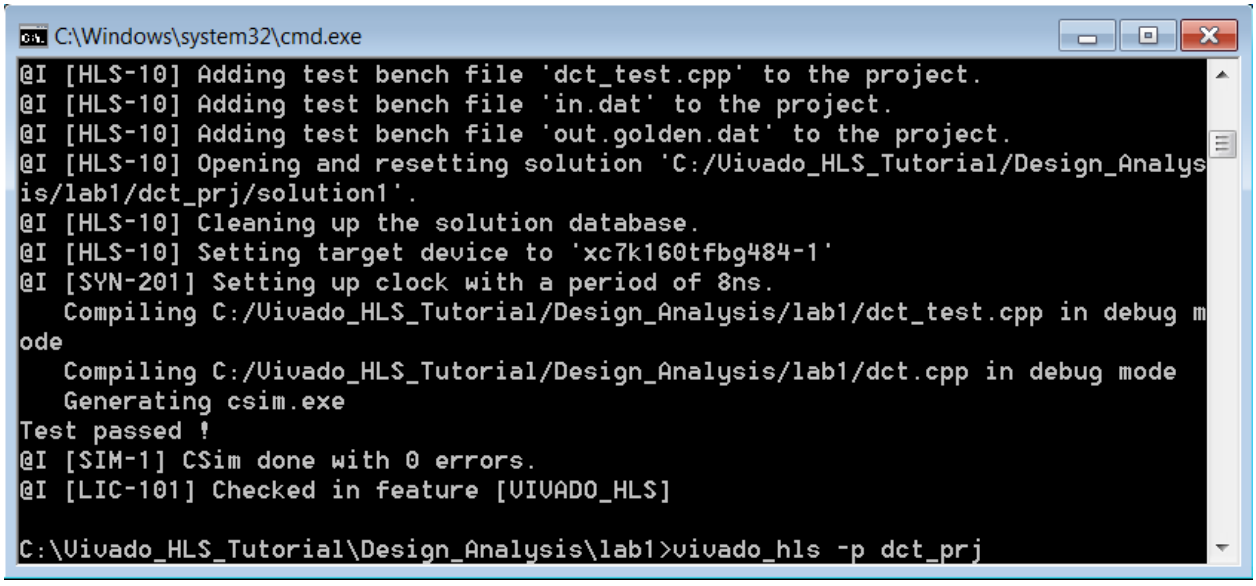


Figure 109 Open Design Analysis Project for Lab#1

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

1. Open the source code for review by double-clicking on file dct.cpp in the Source folder.

This example uses a DCT function. Figure 110 shows an overview of this code.

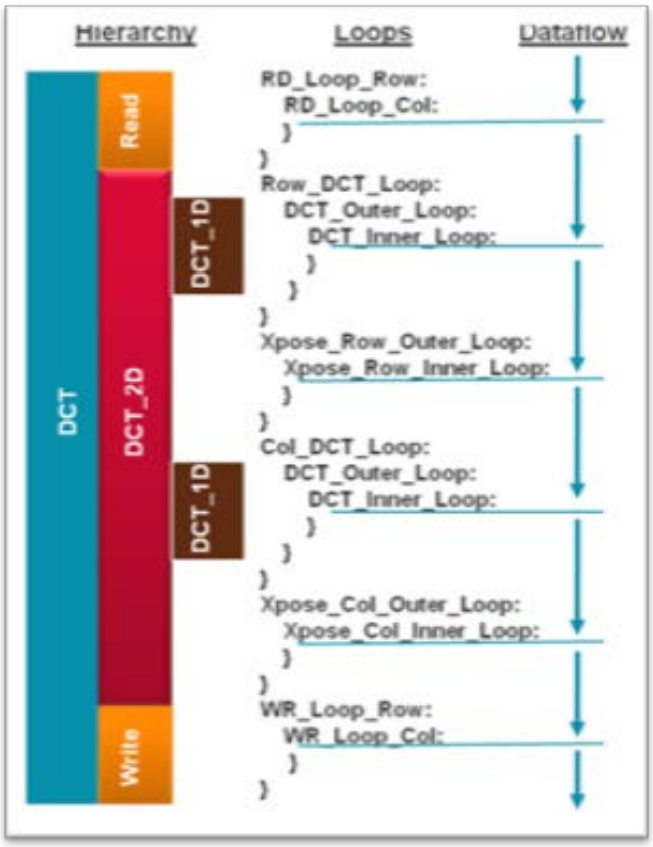


Figure 110 Overview of the DCT design

The right-hand side of Figure 110 shows the code hierarchy.

- Top-level function dct has 3 sub-functions: read_data, dct_2d and write_data.
- Function dct_2d has a single sub-function dct_1d.

The center of Figure 110 shows loops inside each of the functions.

The right-hand side of Figure 110 shows the how the data is processed through the functions and loops.

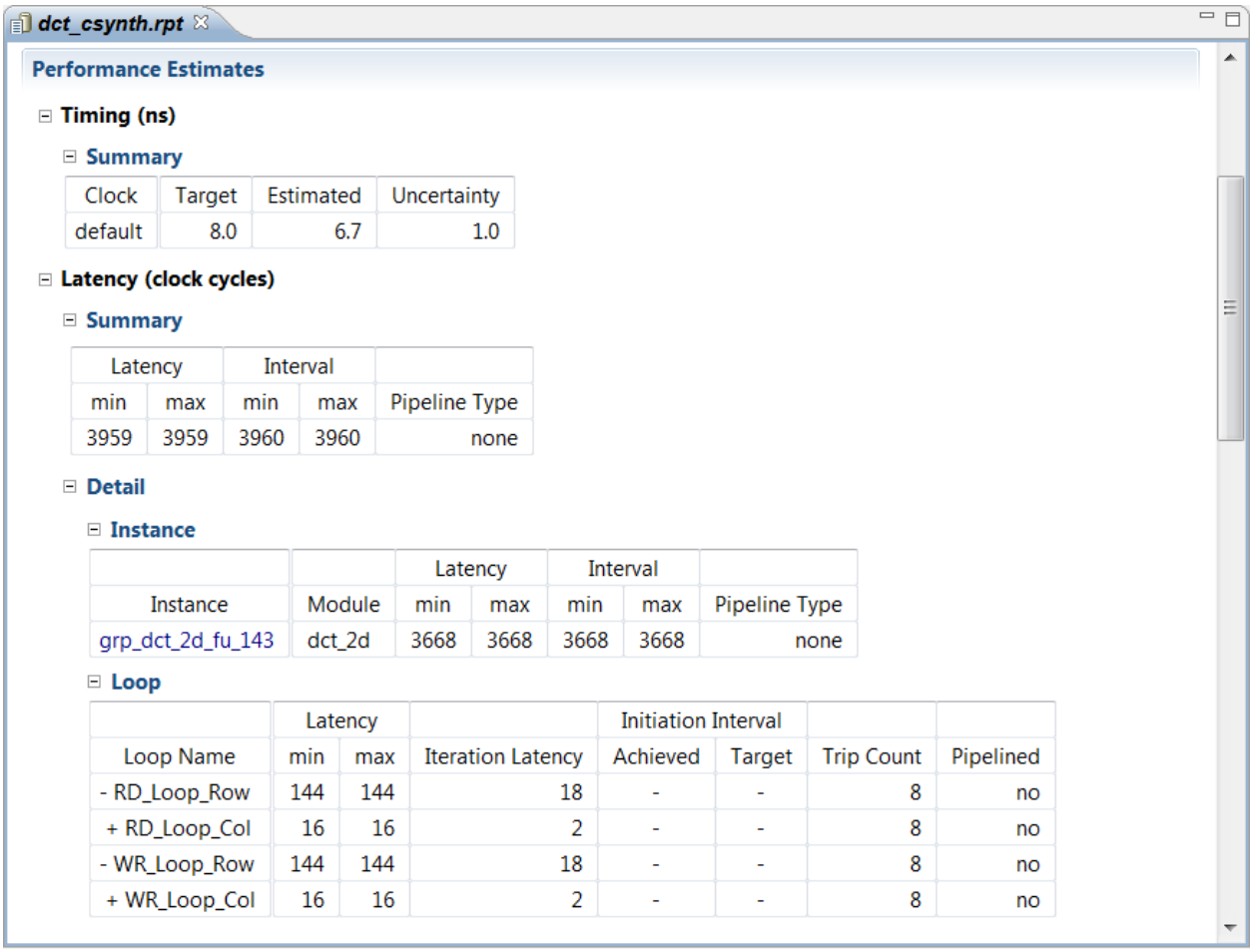
- The `read_data` function is executed and the data processed through loop RD_Loop_Row which has a sub-loop RD_Loop_Col.
- After the read_data function complete, function dct_2d executes.
- In function dct_2d, the data is processed by loop Row_DCT_Loop which has two nested loops inside it: DCT_output_loop and DCT_inner_loop.
- DCT_inner_loop calls function dct_1d.

And so on, until the data is processed by the function write_data.

2. Use the Run C Synthesize toolbar button or menu Solution > Run C Synthesis to synthesizes the design to RTL.

Step 3: Review the performance using the Synthesis Report

When synthesis completes, the synthesis report opens automatically. Figure 111 shows the performance section of the report.



dct_csynth.rpt

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
default	8.0	6.7	1.0

Latency (clock cycles)

Summary

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

Detail

Instance

		Latency		Interval		
Instance	Module	min	max	min	max	Pipeline Type
grp_dct_2d_fu_143	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 111 Report for initial DCT Design

Figure 111 highlights the following information.

- The clock frequency of 8ns has been met.
- The top-level design takes 3539 clock cycles to write all the outputs.
- New inputs can be applied after 3560 clock cycles.
    - This is one clock cycle after the output data has been written and immediately tells us the design is not pipelined but this fact is also noted in the report.
- The top-level has a single instance which has a latency and initiation interval of 3668 and 3669 respectively.
    - This block also has no pipelining and accounts for most of the clock cycles.

- Notice, the functions read_data and write_data are not noted here as instances of the top-level.
    - Figure 112 shows that during synthesis, these blocks where automatically inlined (the hierarchy was removed).
    - High-level synthesis may automatically inline small functions to improve the quality of results (QoR): this can be prevented by using adding the Inline directive with the -off option.

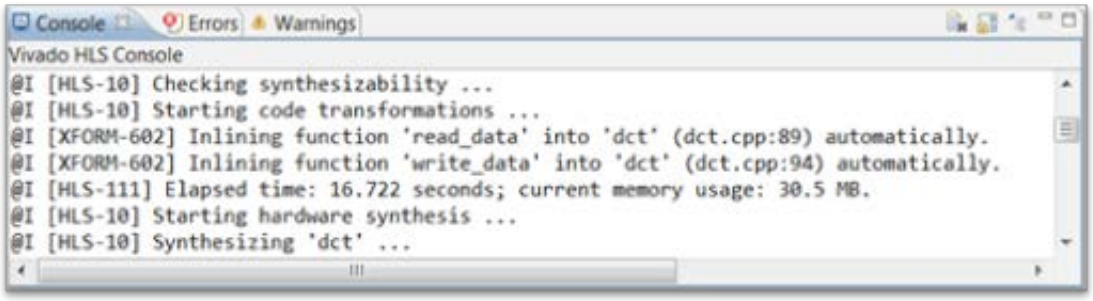


Figure 112 Automatic Inlining for Functions

- The loops in the read_data and write_data functions are therefore implemented at the top-level and are reported as loops in the top-level function (Figure 111).
- Each loop has a latency of 144 clock cycles (since the loops are not pipelined, there is no initiation interval).
- Using RD_Loop_Row as an example, we 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, their respective reports can be opened from the syn/reports folder under solution1 in the Explorer pane.

The design analysis perspective can also be used to review these details in a more interactive manner.

Step 4: Review the Performance using the Analysis Perspective

The Analysis perspective can be invoked any time after synthesis completes.

1. Press the Analysis perspective button (113) to begin interactive design analysis.

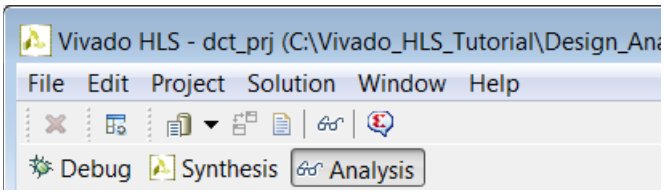


Figure 113 Opening the Analysis perspective

The Analysis perspective consists of 5 panes, each of which is shown highlighted in Figure 114. Each of these will be used in the tutorial. The module and loops hierarchies are shown expanded (by default, they will be shown collapsed).

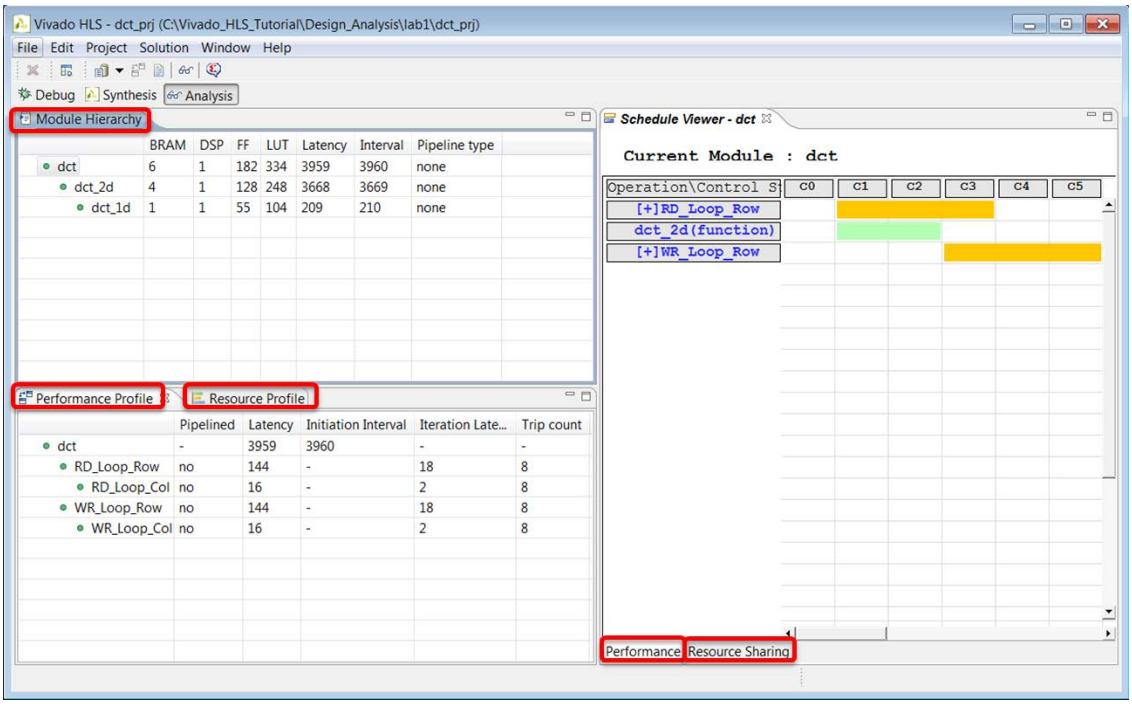


Figure 114 Overview of the Analysis perspective

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

The Performance view is also shown (on the right-hand side of Figure 114). This view shows how the operations in this particular block are scheduled into clock cycles.

- The left-hand column lists the resources.
  - Sub-blocks are green.
  - Operations resulting from loops in the source code are colored yellow.
  - Standard operations are purple.
- We can see the dct has three main resources:

- A loop called RD_Loop_Row. The plus symbol "+" indicates the loop has hierarchy and the loop can be expanded 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 used by High-Level Synthesis 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 on loop RD_Loop_Row and sub-loop RD_Loop_Col to fully expand the loop hierarchy (Figure 115).

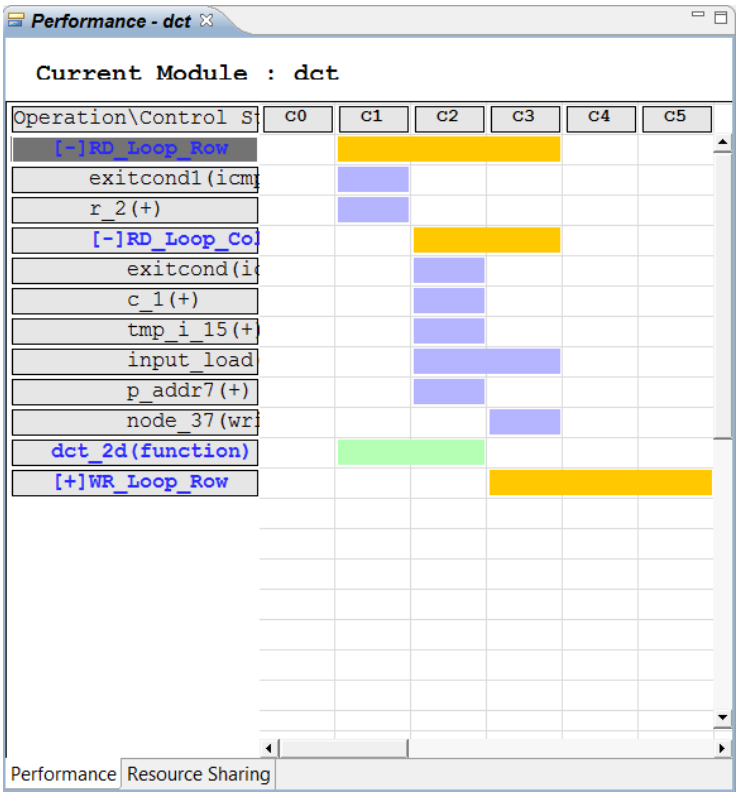


Figure 115 Expanded View of RD_Loop_Row

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

3. Select the adder in state C1, right-click with the mouse and select C source code (Figure 116).

This opens the C source code to highlight which operation in the C source caused this adder to be created. From Figure 116 we can determine it is indeed the loop counter: it is the only addition on this line and the variable is named "r".

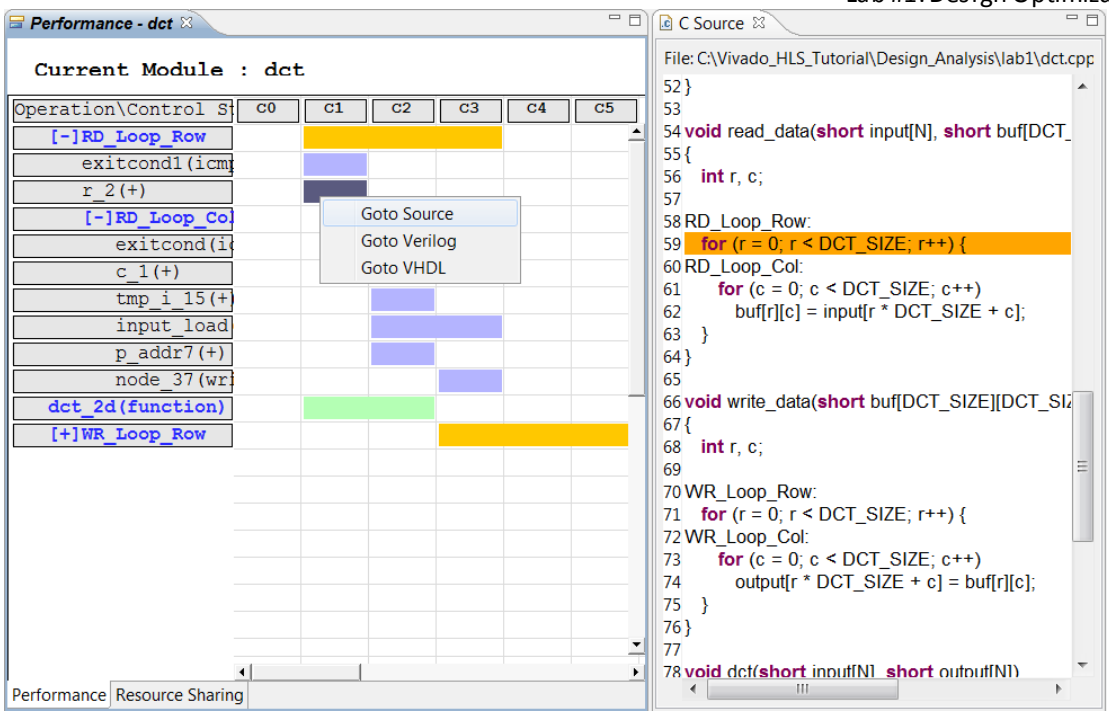


Figure 116 C Source Code View

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

4. Click on 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 or by holding the mouse over the loop in the Performance view (a dialog box will show the loop statistics).

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

5. Click on the C Source Code pane to close this window.
6. In the Module Hierarchy, click on function **dct_2d** to navigate into the view for this function (Figure 117).

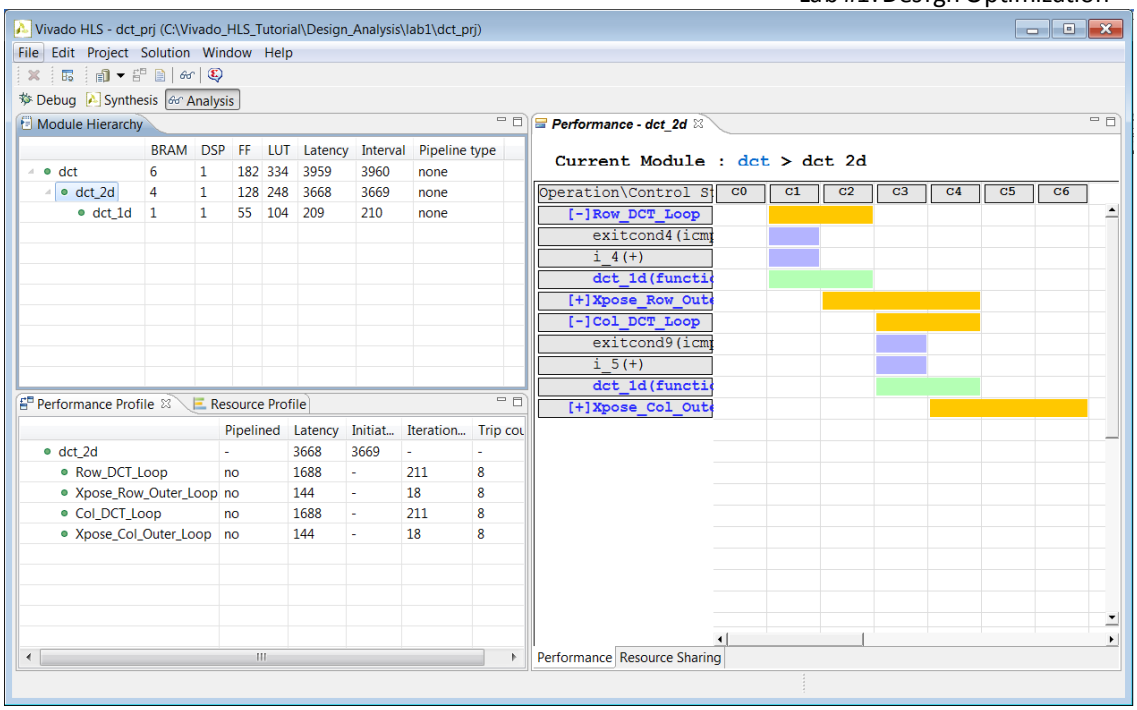


Figure 117 DCT_2D Performance View

Again, you can see a number of loops (shown in yellow in Figure 117): multiple iterative states. These can be pipelined to improve the performance. The details in the Performance Profile shows most of the latency is caused by loops Row_DCT_Loop and Col_DCT_Loop.

7. Click on loops Row_DCT_Loop and Col_DCT_Loop in the performance viewer to fully expand them and see the view in Figure 118.

Expanding these loops in Performance view shows both of these loops call function dct_1d. Unless this function itself is pipelined, there will be no benefit in pipelining the loop - the Module Hierarchy shows the interval for dct_1d is 210 clock cycles: meaning it can only accept a new input every 210 clock cycles.

8. In the Module Hierarchy, click on 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 118.

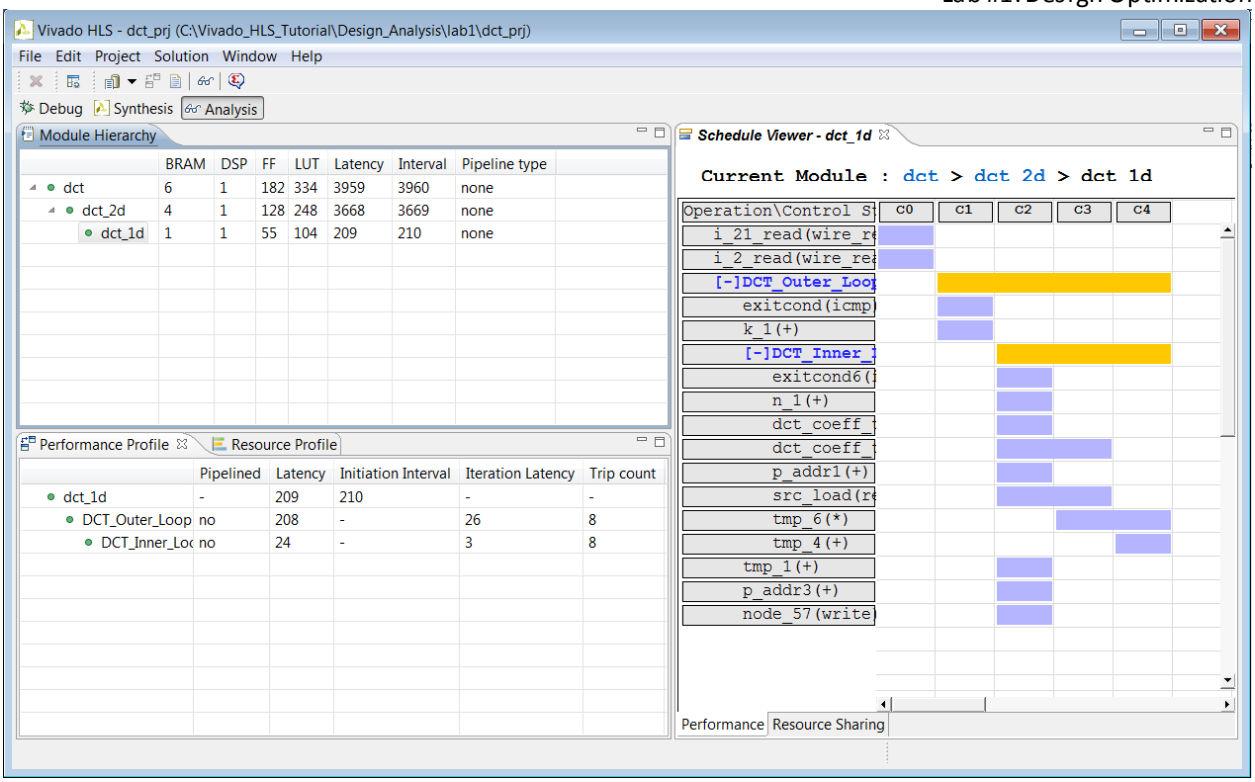


Figure 118 DCT_1D Performance View

In Figure 118 you can see a series of nested loops which can be pipelined.

We have a choice here:

- We can pipeline the function and then pipeline the loop which calls it (since the function is pipelined, the loop can take advantage of using a pipelined part).
- Or we can pipeline the loops within this function and simply make this function execute faster.

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

More details on pipelining loops and functions can be found in the tutorial Design Optimization (page 143). For this case, the approach will be to optimize the loops and keep the area at a minimum.

10. Press the Synthesis perspective button to return to the main synthesis view.

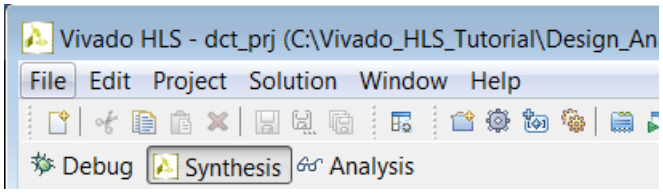


Figure 119 Re-Opening the Synthesis perspective

Step 5: Apply Loop Pipelining & Review for Loop Optimization

Create a new solution and add pipelining directives to the loops.

When pipelining nested loops, it is generally recommended to pipeline the inner-most loop. High-Level Synthesis will generally be able to automatically flatten the loop nest (allowing the outer loop to simply feed the inner loop). For more details on why certain loop optimizations are performed rather than others, please refer to the tutorial "Design Optimization with Pipelining".

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

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

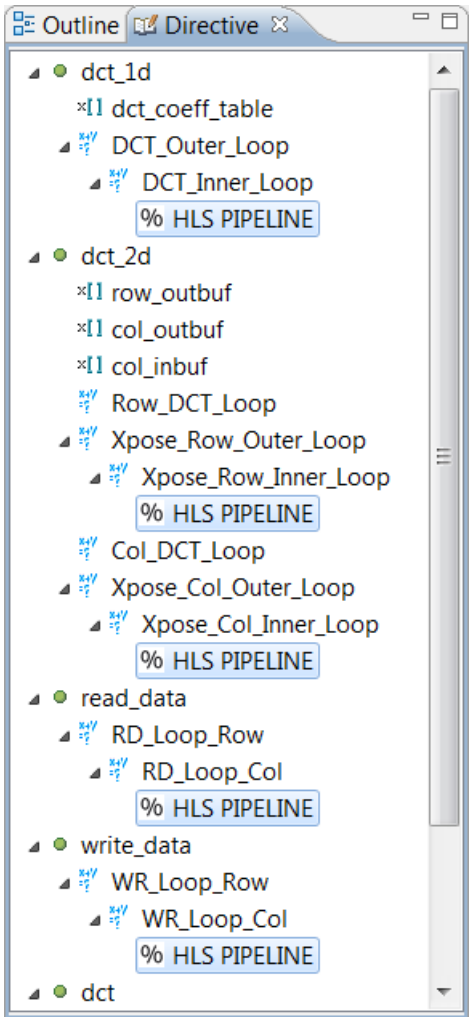


Figure 120 Optimization Directives for DCT Loop Pipelines

6. Use the Run C Synthesize toolbar button or menu Solution > Run C Synthesis to synthesizes 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 121 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.

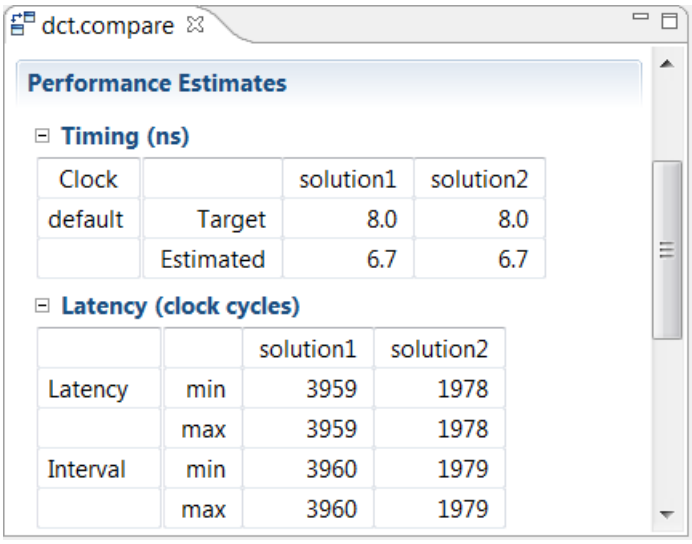


Figure 121 DCT Solution1 and Solution2 Comparison

The next step is to once again open the Analysis perspective, analyze the results and determine if there are more opportunities to for optimization.

8. Press the Analysis perspective button to begin interactive design analysis.

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

The Performance Profile (Figure 122) shows the latency of both loops has been reduced from 144 clock cycles in solution1 to only 65 clock cycles.

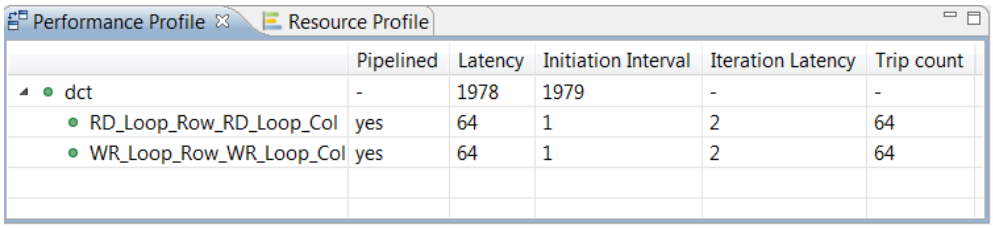


Figure 122 DCT Solution2 Performance of top-level Loops

Pipelining loops transforms the latency from

- Latency = iteration latency * tripcount

to

- Latency = iteration latency + tripcount

This was also made possible by HLS automatically performing loop flattening (there is no longer any loop hierarchy). This can be seen by reviewing the Console pane, or log file, for solution2. Figure 123 shows the loops which have been automatically optimized.

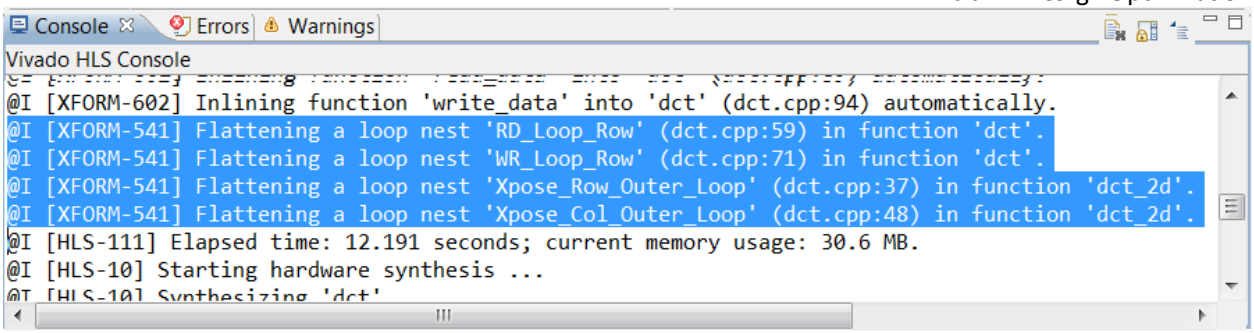


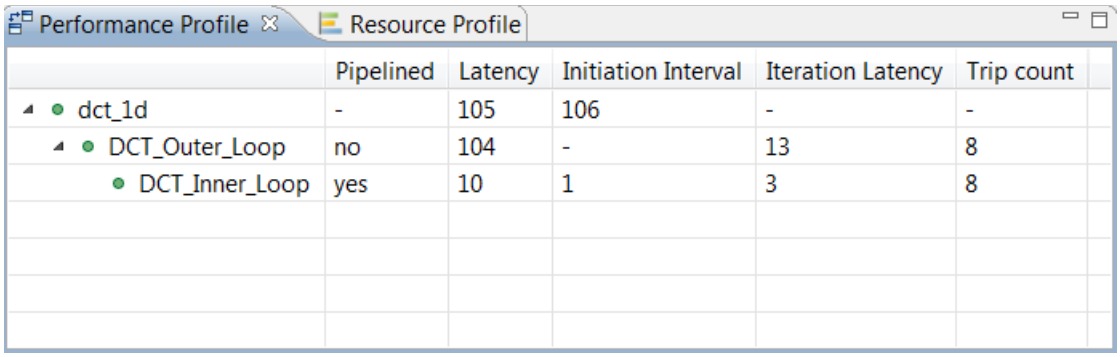
Figure 123 DCT Solution2 Loop Flattening

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

In the Performance Profile we can see 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 117). 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 on function dct_1d to navigate into the view for this function.

The Performance Profile (Figure 124) shows the loop latencies have been reduced, but there is still a loop hierarchy here (there is still loop DCT_Outer_Loop in Figure 124, so no loop flattening was able to occur).



	Pipelined	Latency	Initiation Interval	Iteration Latency	Trip count
dct_1d	-	105	106	-	-
DCT_Outer_Loop	no	104	-	13	8
DCT_Inner_Loop	yes	10	1	3	8

Figure 124 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 on loops DCT_Outer_Loop and DCT_Inner_Loop to view the loop hierarchy (Figure 125).
12. Select the write operation in state C5.
13. Right-click with the mouse and select Goto Source.

Figure 125 shows that this loop was not flattened because there are additional operations (one of the operations is highlighted in the figure) outside of DCT_Inner_Loop, at the level of DCT_Outer_Loop, which prevented loop flattening.

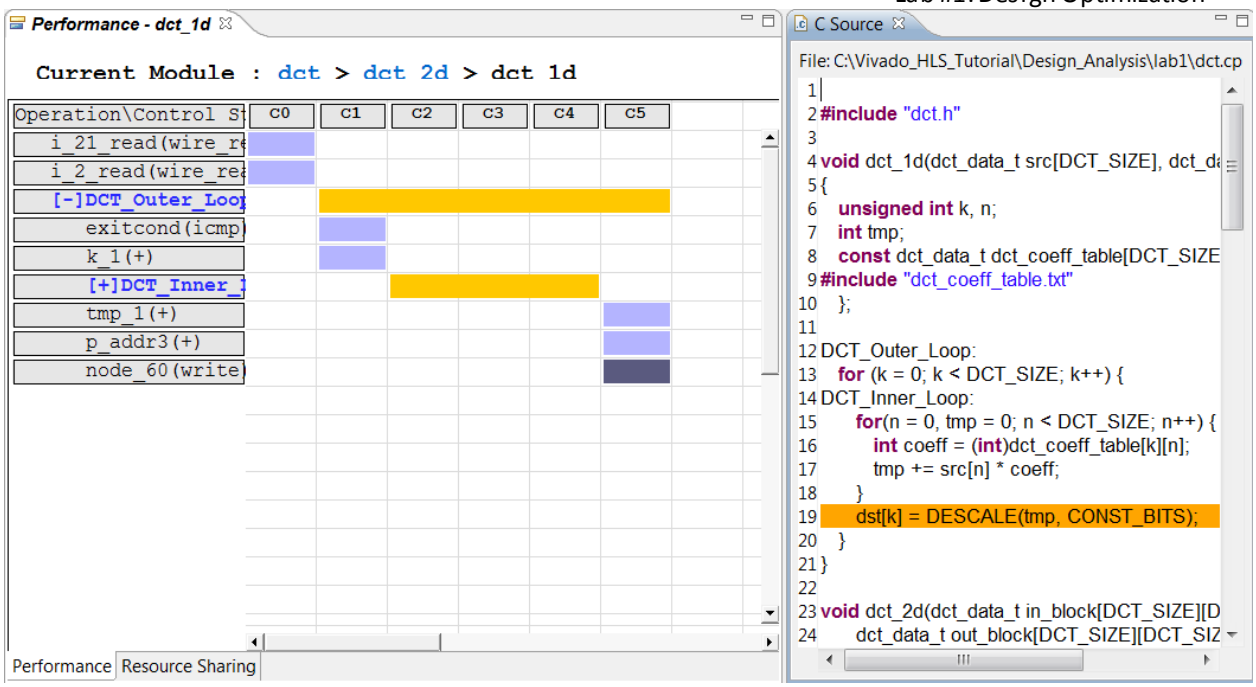


Figure 125 DCT Solution2 dct_1d Performance View

This is a case where pipelining the inner-most loop does not give us the biggest benefit. In this case, the outer loop should be pipelined. This will cause the inner loop to be completely unrolled and an increase in area, but we are still far from our 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. Press 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. Press OK and accept the defaults to create solution3.
3. Ensure the C source code is visible in the Information pane.
4. In the Directives tab
    a. In function dct_1d, select the pipeline directive on loop DCT_Inner_Loop
    b. Right-Click with the mouse and select Remove Directive.
    c. Still in function dct_1d, select loop DCT_Outer_Loop.
    d. Right-click with the mouse and select Insert Directive
    e. In the Directives Editor dialog box activate the Directives drop-down menu at the top and select PIPELINE.
    f. Press OK and select the default maximum pipeline rate (II=1)

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

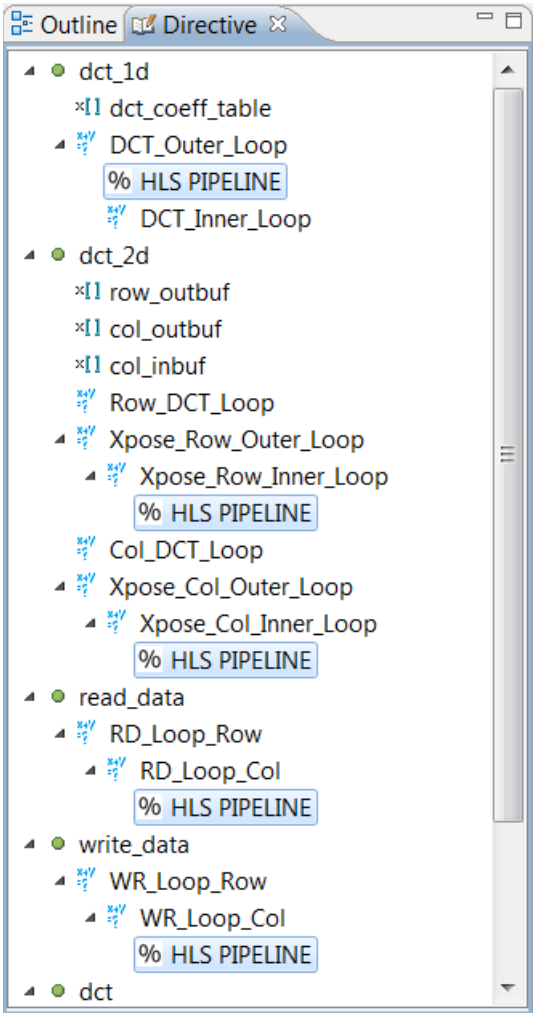


Figure 126 Updated Optimization Directives for DCT Loop Pipelines

5. Use the Run C Synthesize toolbar button or menu Solution > Run C Synthesis 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 2 and 3.

Figure 127 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 great latency benefit was achieved because the dct_1d function is called multiple times, by multiple loops in the design: saving latency in this block is multiplied because the design is used inside many loops.

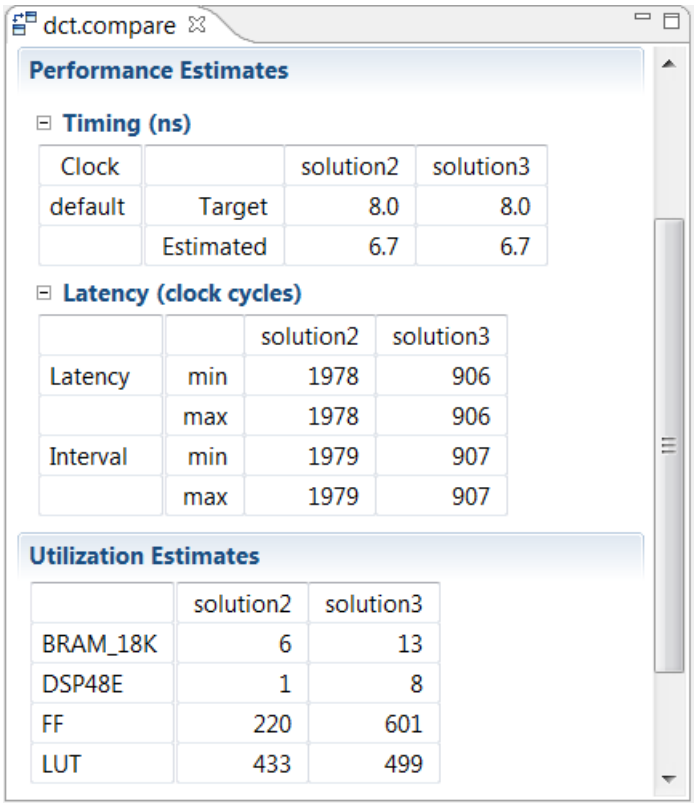


Figure 127 DCT Solution2 and Solution3 Comparison

All the loops are now pipelined. It is now worthwhile reviewing the design to see if there are bottlenecks which are limiting performance. Bottlenecks are limitations in the flow of data which 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. IO ports and arrays implemented as BRAM are one source: in both cases the rate at which data can be read or written is limited by a finite number of ports (on the IO or BRAM).

Another source 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. In some cases 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 issue in the RTL design. There are a number of approaches which can be used.

- 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.
- Use the Resource Profile to examine IO 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, we will use the latter approach. The Analysis perspective can be used to quickly identify such places in the design.

7. Press 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 128, 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 2 states. The red arrow in Figure 128 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.

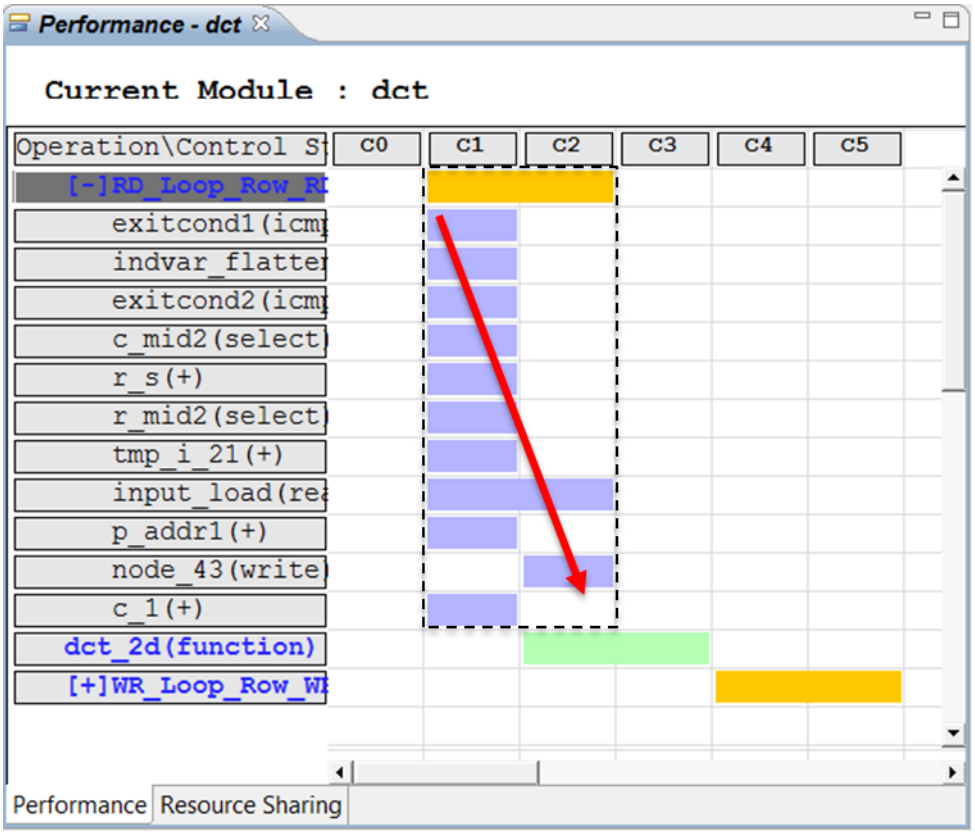


Figure 128 Analysis of DCT RD_Loop_Row

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

The same analysis process can be used down through the hierarchy. All of the function blocks and loops will have a similar optimal (few cycle) implementation, until the dct_1d block is examined.

12. In the Performance view, double-click on function dct_1d and navigate into the dct_1d function.
13. Expand the DCT_Outer_Loop to see the view shown in Figure 129.

Figure 129 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).

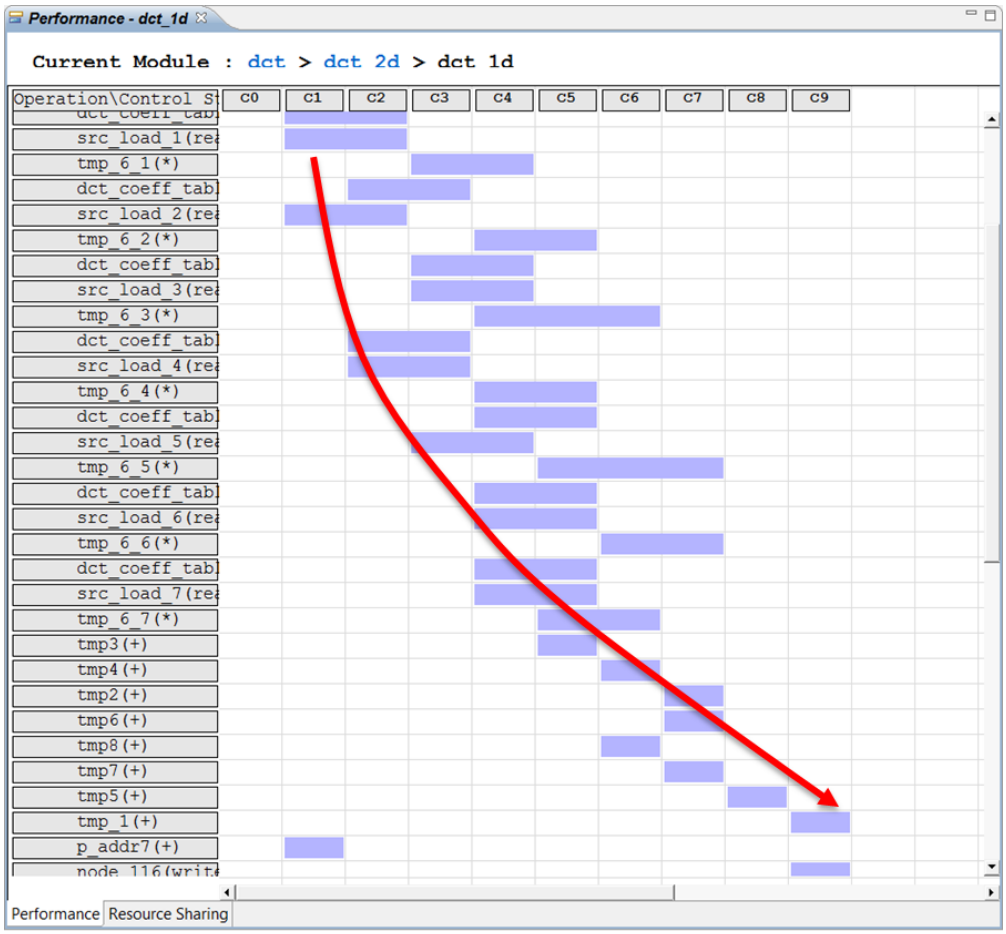


Figure 129 Analysis of dct_1d RD_Loop_Row

There are typically two things which cause this type of schedule, data dependencies in the source code and limitations due to IO or BRAM. Let's examine the resources sharing in this block.

14. In the Performance view, click on the Resource Sharing tab at the bottom of the window.
15. Expand the Memory Ports to see the view in Figure 130.

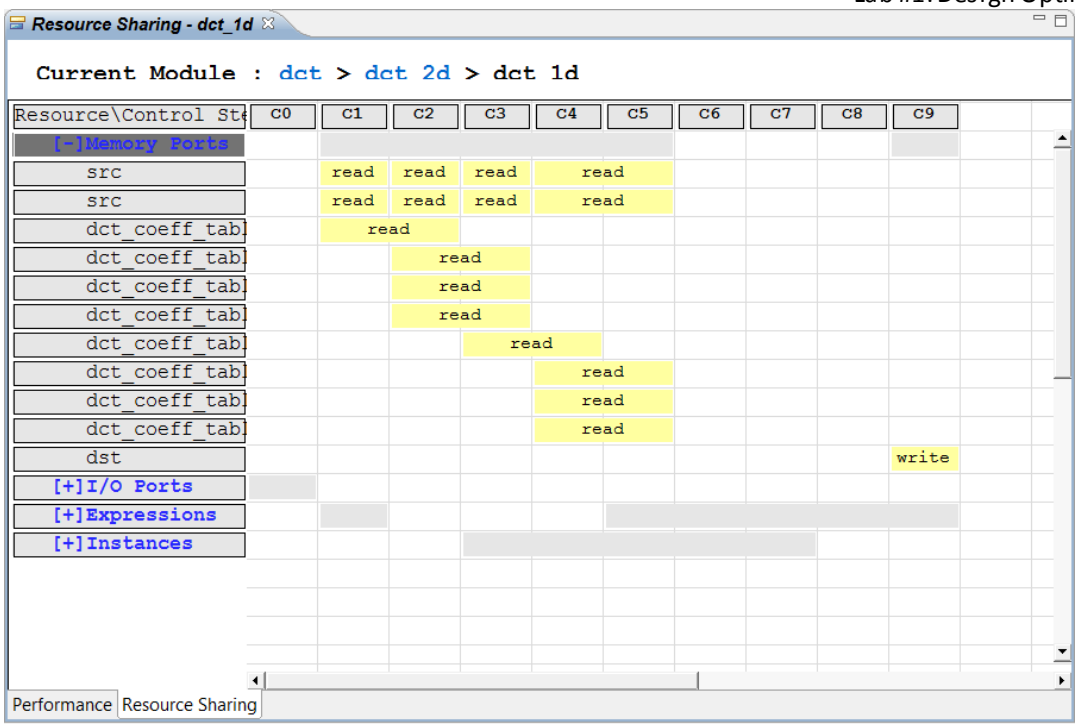


Figure 130 Resource Sharing of Memory Ports in dct_1d

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 130, 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 130 shows the memory accesses on BRAM src are being used to the maximum in every clock cycle. (At most a BRAM 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, we can examine further.

16. Select one of the read operations for the src BRAM.
17. Right-click with the mouse and select Goto Source to see the view shown in Figure 131.

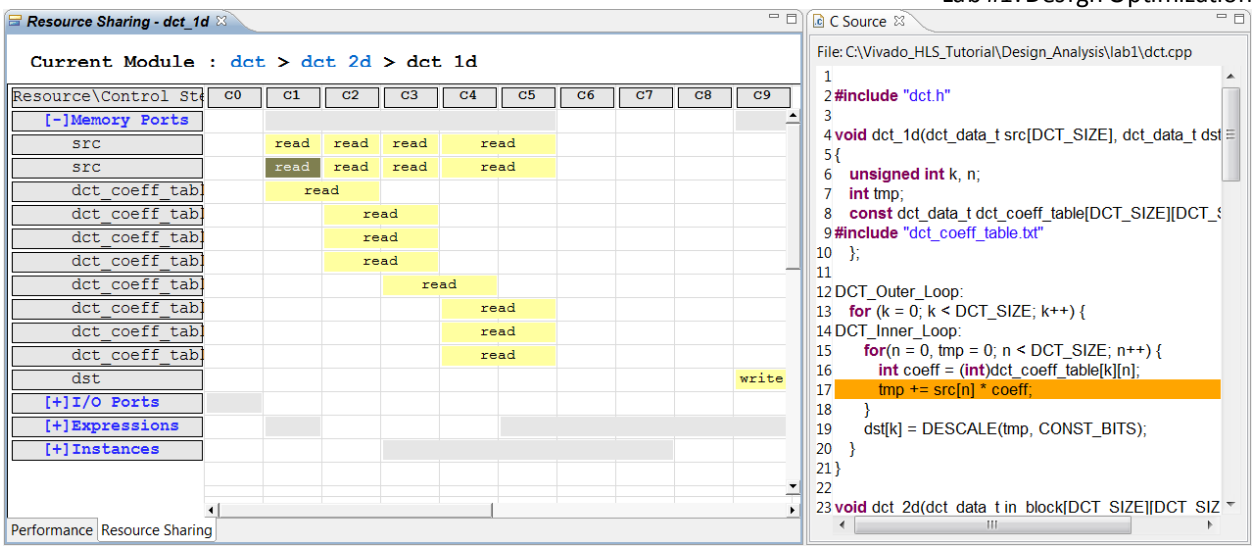


Figure 131 Memory resource src and Source Code

Figure 131 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 8 reads are being forced to occur over multiple cycles because the array src is implemented as a BRAM in the RTL and a BRAM can only allow 2 reads (maximum) in any one clock cycle. In Figure 131, the read operations take 2 clocks cycle: a cycle to generate the address for the BRAM 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.

The BRAM accesses can be optimized by partitioning the BRAM using optimization directives. The BRAM which 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-dimentional of size DCT_SIZE by DCT_SIZE (8x8). By default this results in a single BRAM with 64 elements. Since the arrays are configured in the code in the form of Row by Column, we can partition the 2nd dimension and create 8 separate BRAMS: one for each row, allowing the row data to be accessed in parallel.

18. Press the Synthesis perspective button to return to the main synthesis view.

Step 7: Partition BRAMs and Analyze Concurrency

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

c. In the Directives Editor dialog box activate the Directives drop-down menu at the top and select ARRAY_PARTITION.
d. Leave the type as Complete.
e. Change the dimension setting to 2: this will partition the array along the 2nd dimension.
f. Press OK

5. Repeat this process for array col_inbuf in function dct_2d.

The Directive pane should show the following optimization directives (the two new directives are highlighted).

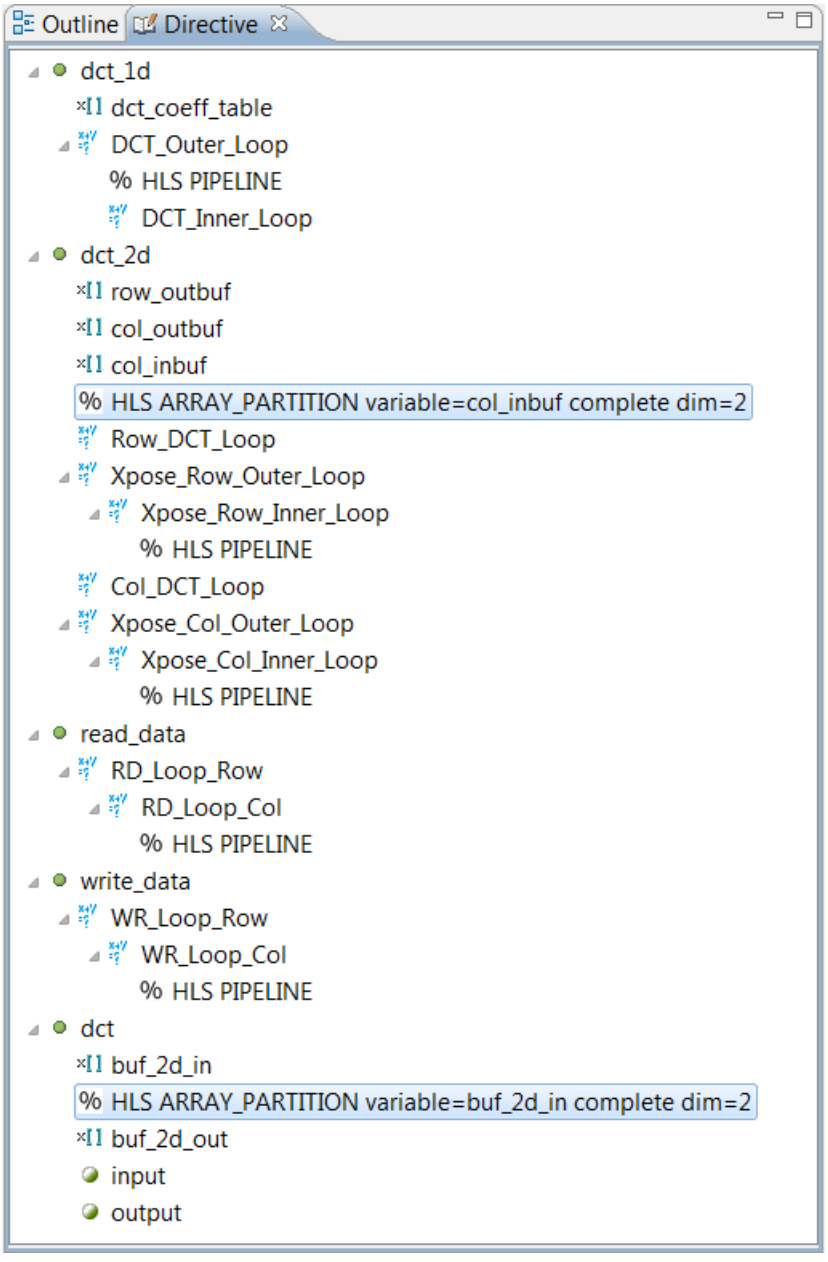


Figure 132 Optimization Directives for Array Partitioning

6. Use the Run C Synthesize toolbar button or menu Solution > Run C Synthesis to synthesizes the design to RTL.
7. When synthesis completes, use the Compare Reports toolbar button or the menu Project > Compare Reports to compare solutions 3 and 4.

Figure 133 shows the results of comparing solution3 and solution4. Improving access to the data in the src BRAM in the dct_1d block has improved the overall performance because the dct_1d block is executed often.

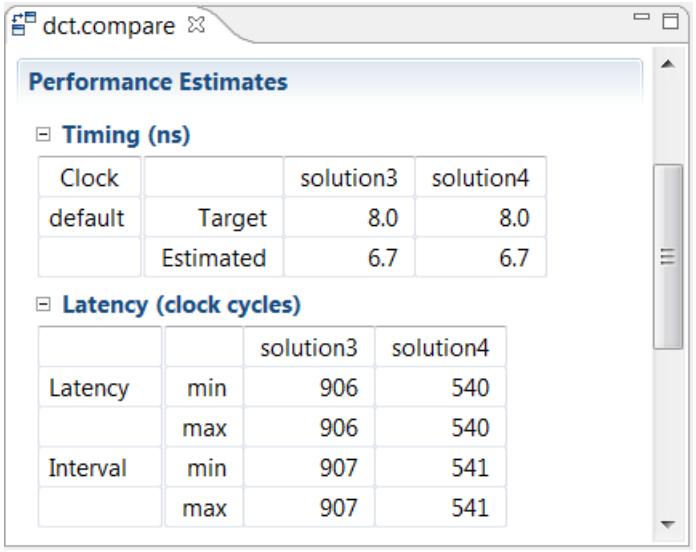


Figure 133 DCT Solution3 and Solution4 Comparison

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

8. Press 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 134.

Module Hierarchy

	BRAM	DSP	FF	LUT	Latency	Interval	Pipeline type
dct	27	8	606	607	540	541	none
dct_2d	18	8	546	450	405	406	none
read_data	0	0	27	60	66	67	none

Performance Profile | Resource Profile

	BRAM	DSP	FF	LUT	Bits P0	Bits P1	Bits P2	Banks/Depth
dct	**27**	**8**	**606**	**607**				
I/O Ports(2)					**32**			
Instances(2)	**18**	**8**	**573**	**510**				
Memories(9)	**9**		**0**	**0**	**144**			**9**
buf_2d_in_0_U	1		0	0	16			1
buf_2d_in_1_U	1		0	0	16			1
buf_2d_in_2_U	1		0	0	16			1
buf_2d_in_3_U	1		0	0	16			1
buf_2d_in_4_U	1		0	0	16			1
buf_2d_in_5_U	1		0	0	16			1
buf_2d_in_6_U	1		0	0	16			1
buf_2d_in_7_U	1		0	0	16			1
buf_2d_out_U	1		0	0	16			1
Expressions(9)	**0**	**0**	**0**	**45**	**40**	**33**	**8**	
+	**0**	**0**	**0**	**27**	**27**	**15**	**0**	
icmp	**0**	**0**	**0**	**10**	**11**	**13**	**0**	
Select	**0**	**0**	**0**	**8**	**2**	**5**	**8**	
Registers(11)			**33**		**33**			
FIFO(0)	**0**		**0**	**0**	**0**			**0**
Multiplexers(14)	**0**		**0**	**52**	**52**			**0**

Figure 134 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 134 shows:

- This block has 2 IO ports.
- Most of the area is due to instances (sub-blocks) within this block.
- There are 9 memories of which 8 are the partitioned buf_2d_in BRAM.
- 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 we can see we now have addition memories due to the array partitioning optimization.

We still have a target of being able to accept a new set of samples every 100 clock cycles. Figure 133 however shows we can only accept new data every 541 clocks. This is much better than the original un-optimized design (approx. 3700 clock cycles) but still not enough.

Up to this point we have focused on improving the latency and interval of each of the individual loops and functions in the design. It is now time to improve the overall design interval by enabling the individual loops and functions to execute in parallel.

The optimization for performing this is the dataflow optimization.

12. Press the Synthesis perspective button to return to the main synthesis view.

Step 8: Partition BRAMs and Apply Dataflow optimization

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

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

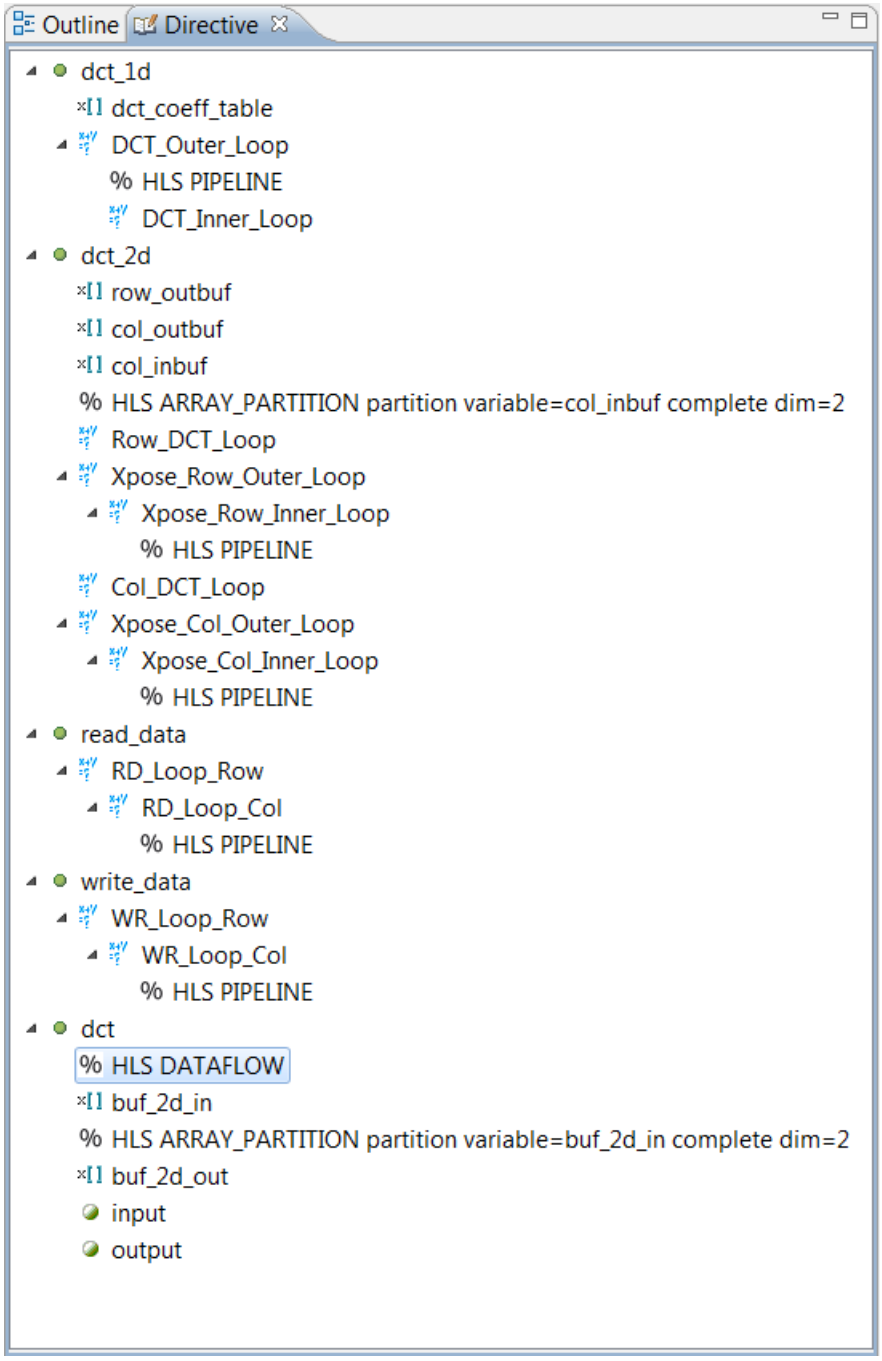


Figure 135 Dataflow Optimization for the DCT design

6. Use the Run C Synthesize toolbar button or menu Solution > Run C Synthesis to synthesizes the design to RTL.
7. When synthesis completes, use the Compare Reports toolbar button or the menu Project > Compare Reports to compare solutions 4 and 5.

Figure 136 shows the results of comparing solution4 and solution5 and we can see the interval has improve: the design takes 539 clocks cycles to produce the outputs but can now accept new inputs every 405 clocks.

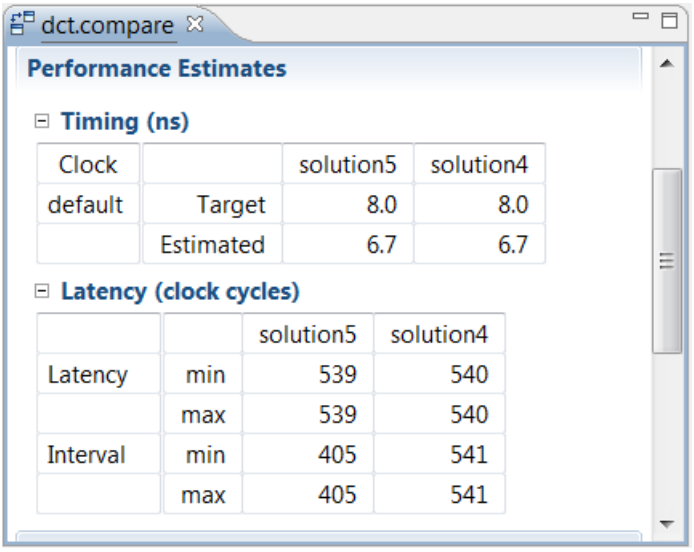


Figure 136 DCT Solution4 and Solution5 Comparison

This is still greater than the 100 cycles we require so let's analyze the current performance.

8. Press the Analysis perspective button to begin interactive design analysis.
9. In the Module Hierarchy, you can see dct_2d accounts for most of the interval. Ensure module dct_2d is selected to see the view in Figure 137.

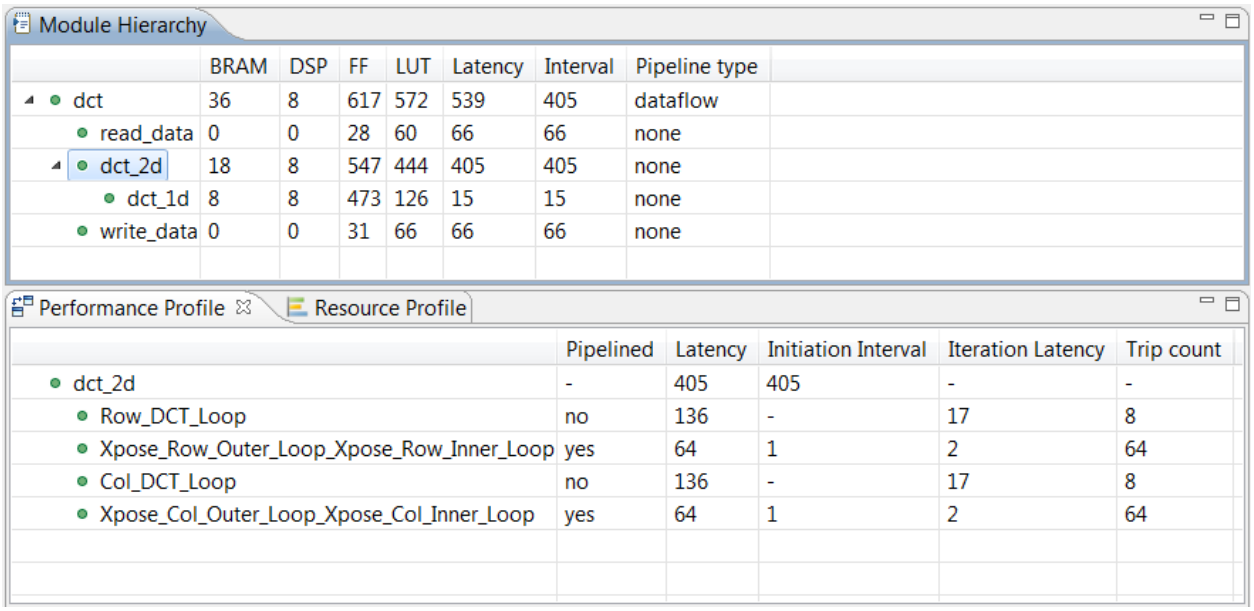


Figure 137 DCT Analysis View after Dataflow Optimization

Here, we can see two things:

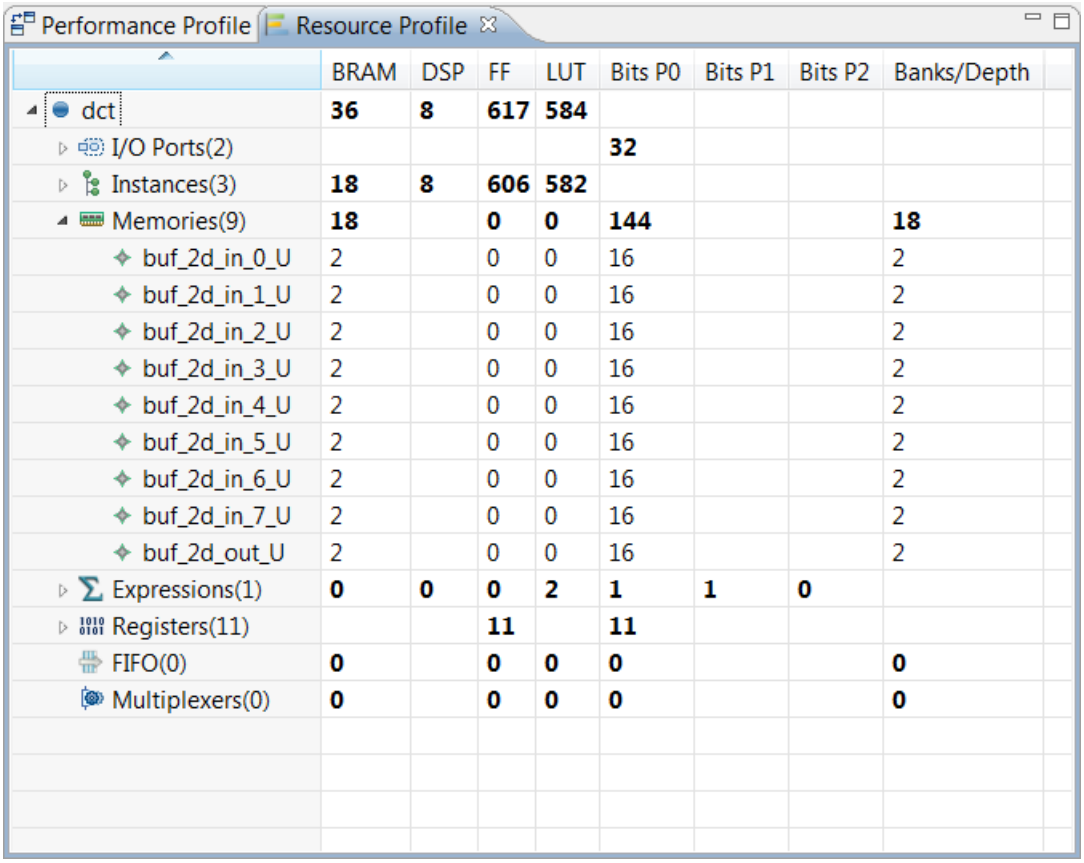
- First, the interval of the dct block is less than the sum of the individual latencies (for read_data, dct_2d and write_data). This means the blocks are operating in parallel.
- Secondly, the interval of dct is the same as the interval for sub-block dct_2d. The dct_2d block is therefore the limiting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

Since the dct_2d block is selected in the Module Hierarchy, the Performance Profile shows the details for this block. Figure 137shows the interval is the same as the latency, so none of these blocks are operating in parallel.

One of the limitations of dataflow optimization is that it only works on top-level loops and functions. One way to have the blocks in dct_2b operate in parallel would be to pipeline the entire function. This however would unroll all the loops and can sometimes lead to a large area increase. An alternative is to raise these loops up to the top-level of hierarchy, where dataflow optimization can be applied, by removing the dct_2d hierarchy: inline the dct_2d function.

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

10. In the Module Hierarchy, ensure module dct is selected.
11. Activate the Resource Profile view.
12. Expand the memories to see the view in Figure 138.



	BRAM	DSP	FF	LUT	Bits P0	Bits P1	Bits P2	Banks/Depth
dct	**36**	**8**	**617**	**584**				
I/O Ports(2)					**32**			
Instances(3)	**18**	**8**	**606**	**582**				
Memories(9)	**18**		**0**	**0**	**144**			**18**
buf_2d_in_0_U	2		0	0	16			2
buf_2d_in_1_U	2		0	0	16			2
buf_2d_in_2_U	2		0	0	16			2
buf_2d_in_3_U	2		0	0	16			2
buf_2d_in_4_U	2		0	0	16			2
buf_2d_in_5_U	2		0	0	16			2
buf_2d_in_6_U	2		0	0	16			2
buf_2d_in_7_U	2		0	0	16			2
buf_2d_out_U	2		0	0	16			2
Expressions(1)	**0**	**0**	**0**	**2**	**1**	**1**	**0**	
Registers(11)			**11**		**11**			
FIFO(0)	**0**		**0**	**0**	**0**			**0**
Multiplexers(0)	**0**		**0**	**0**	**0**			**0**

Figure 138 DCT Resource Profile

As compared with Figure 134, we can see there are now twice as many memories at this level of hierarchy (18 vs. 9). Each of the memories has been transformed into a Ping-Pong buffer to support dataflow. In this case, no "new" memories were added, the existing memories were converted into dataflow Ping-Pong memory channels: this doubled the number of BRAM.

13. Press 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 or use the menu Project > New Solution to create a new solution.
2. Press OK and accept the defaults to create solution6.
3. Ensure the C source code is visible in the Information pane.
4. In the Directives tab
    a. Select function dct_2d
    b. Right-click with the mouse and select Insert Directive
    c. In the Directives Editor dialog box activate the Directives drop-down menu at the top and select INLINE.
    d. Press OK

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

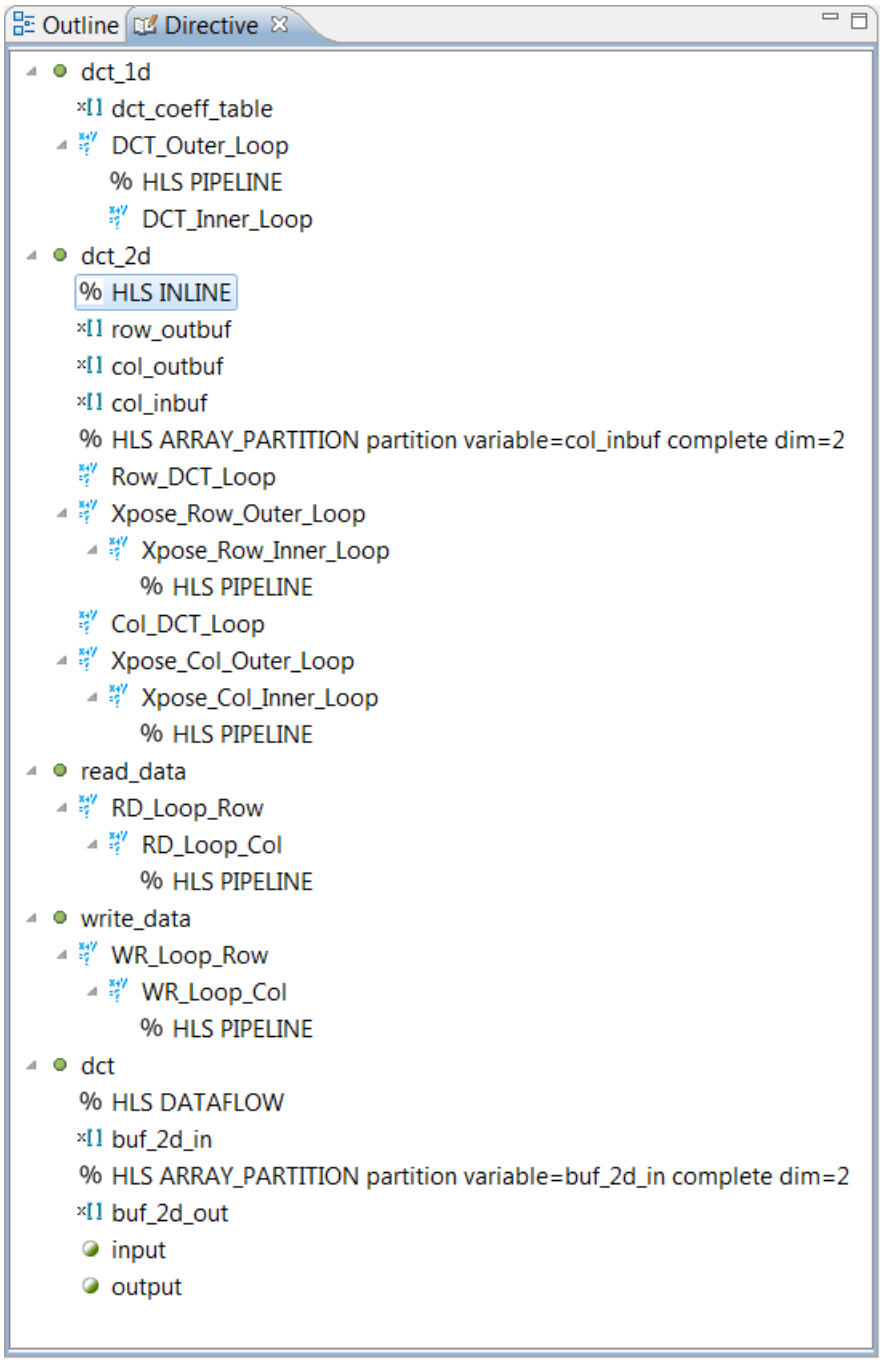


Figure 139 Dataflow Optimization for the DCT design

5. Use the Run C Synthesize toolbar button or menu Solution > Run C Synthesis 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 140 shows the results of comparing solution5 and solution6 and we can see the interval has improve substantially.

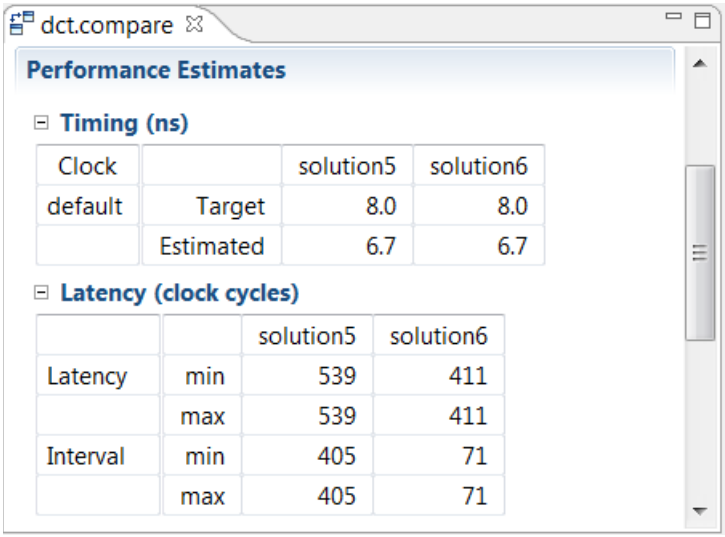


Figure 140 DCT Solution5 and Solution6 Comparison

The interval is now below the 100 clock target: this design can accept a new set of input data every 71 clock cycles.

The details also be seen in the synthesis report which opens automatically after synthesis completes and in the Analysis perspective as shown in Figure 141

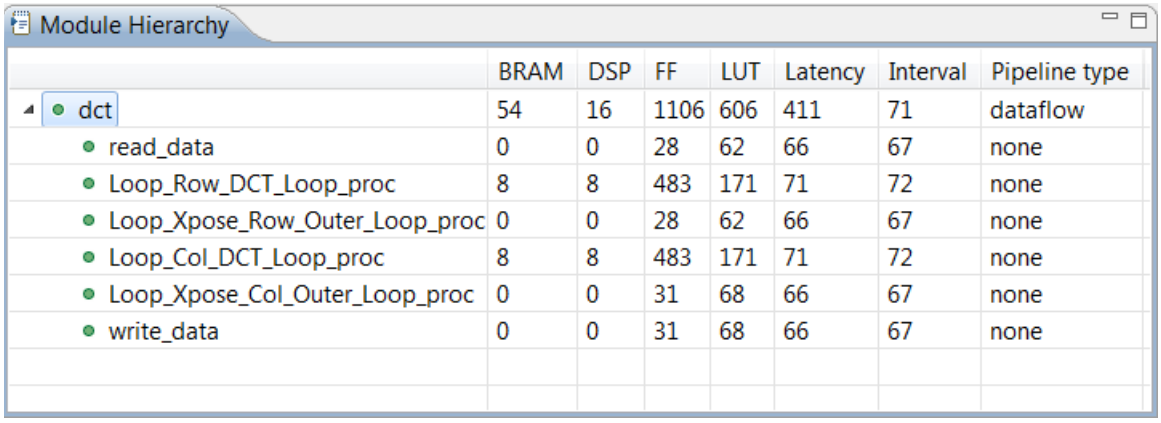


Figure 141 DCT Solution6 Module Hierarchy

This completes this tutorial on using arbitrary precision types.

Conclusion

In this tutorial, you learned:

- How to analyze a design using the analysis perspective.
- To cross-link operations in the views with the C code.
- Apply and judge optimizations.
- And 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 being able to apply optimizations to the C code. High-Level Synthesis will always try to minimize the latency of loops and functions – within the loops and functions it will try to execute as many operations as possible in parallel to achieve this. At the level of functions, High-Level Synthesis will always try to execute functions in parallel.

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

- To execute multiple tasks in parallel, for example, multiple executions of the same function or multiple iterations of the same loop. This is pipelining.
- To restructure the physical implementation of arrays (BRAMs), functions, loops and ports to improve the availability of data and help data flow through the design faster.
- To 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 exercise. . The analysis in these lab exercises is performed using the Analysis perspective and it is recommended to have completed the Design Analysis tutorial (page 111) before starting this tutorial.

Lab1

Contrast the uses of loop and function pipelining to create a design that can process one sample per clock. This lab includes examples analyzing the two most common reasons designs fail to meet performance: loop dependencies and data flow limitations or bottlenecks

Lab2

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

Tutorial Design Description

The tutorial design file can be downloaded from the Xilinx website. Refer to the information in Obtaining the Tutorial Designs.

This tutorial uses the design files in the tutorial directory
Vivado_HLS_Tutorial\Design_Optimization

The sample design used 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 will use a matric multiplier design to show how a design heavily based on loops can be fully optimized. The target is to be able to read one sample per clock cycle using a FIFO interface while minimizing the area.

The analysis will include a comparison of a methodology which optimizes at the loop level and one which optimizes at the function level.

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

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

Step 1: Create and Open the Project

1. Open the Vivado HLS Command Prompt.
    a. On Windows use **Start** > **All Programs** > **Xilinx Design Tools** > **Vivado 2013.1** > **Vivado HLS** > **Vivado HLS 2013.1 Command Prompt (**Figure 142).
    b. On Linux, open a new shell.

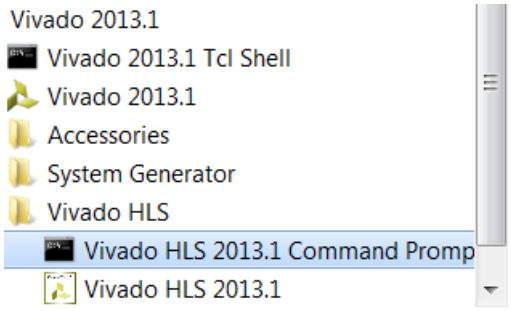


Figure 142 Vivado HLS Command Prompt

2. Using the command prompt window (Figure 143), 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 143.

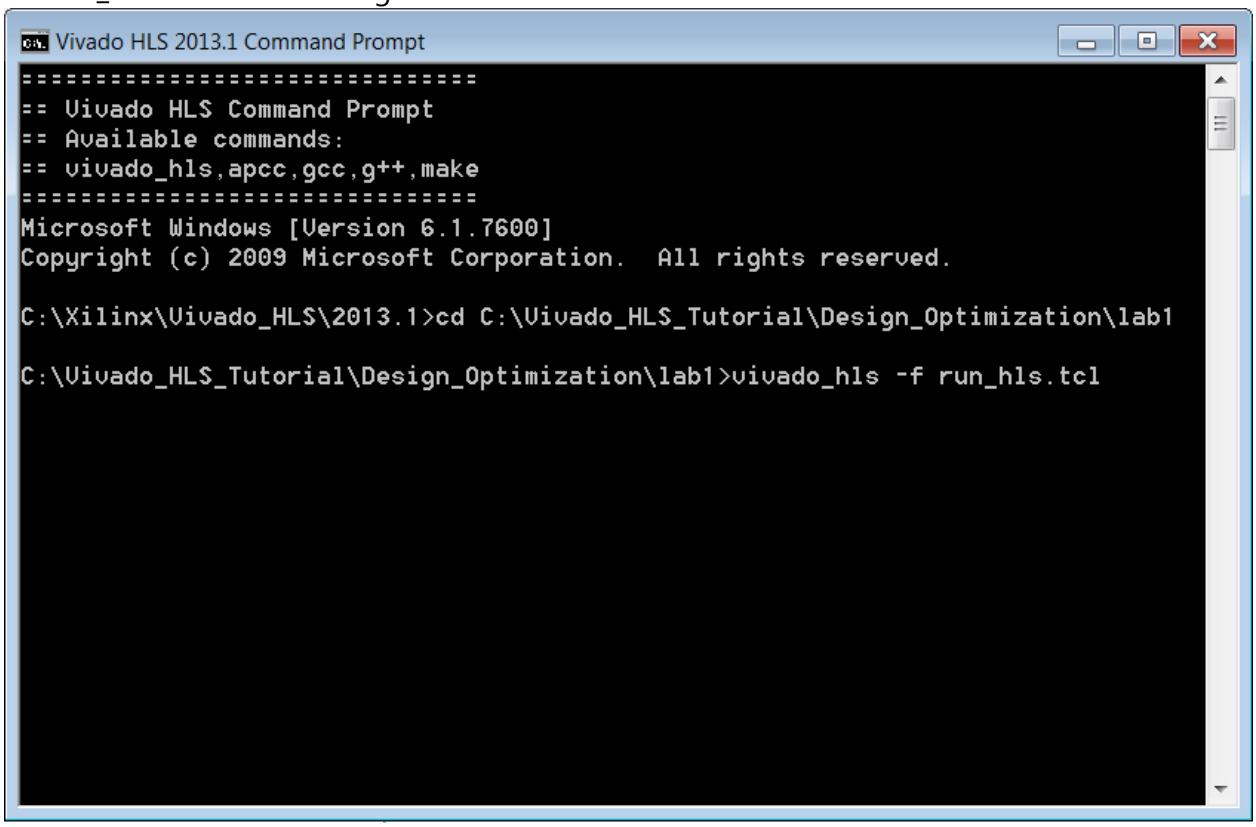


Figure 143 Setup the Design Optimization Tutorial Project

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 144.

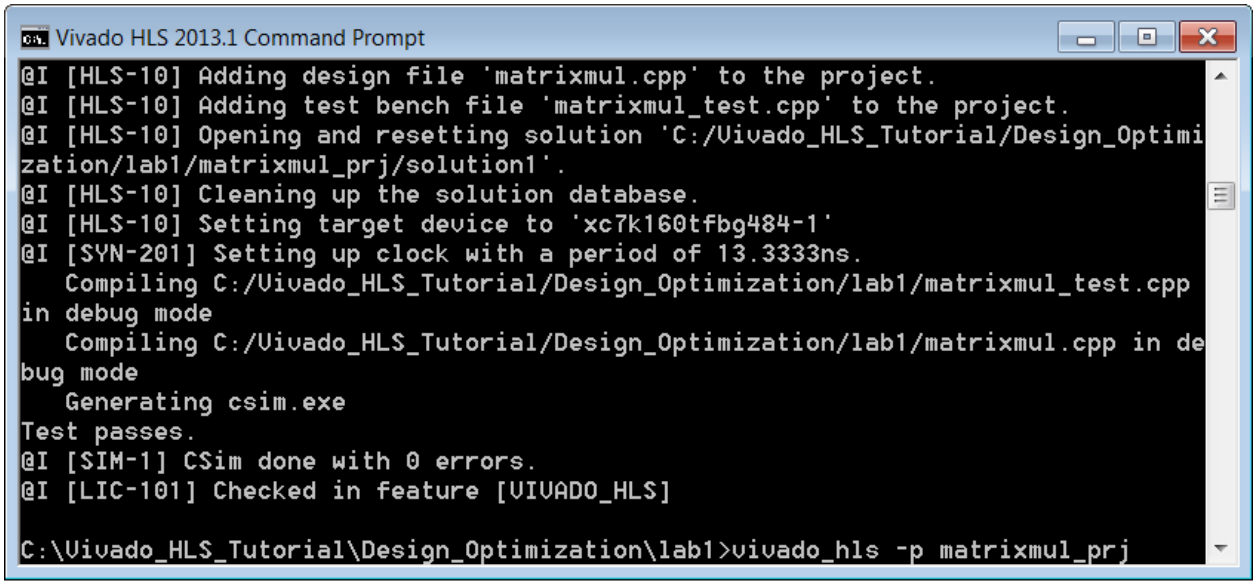


Figure 144 Open Design Optimization Project for Lab#1

5. Expand the Sources folder in the Explorer pane and double-click on matrixmul.cpp to view the source code (Figure 145).

Scrolling down the file shows the source code has two input arrays, a and b and output array res. Holding the mouse over the macros (as shown in Figure 145) shows each is 3 by 3 for a total of 9 elements.

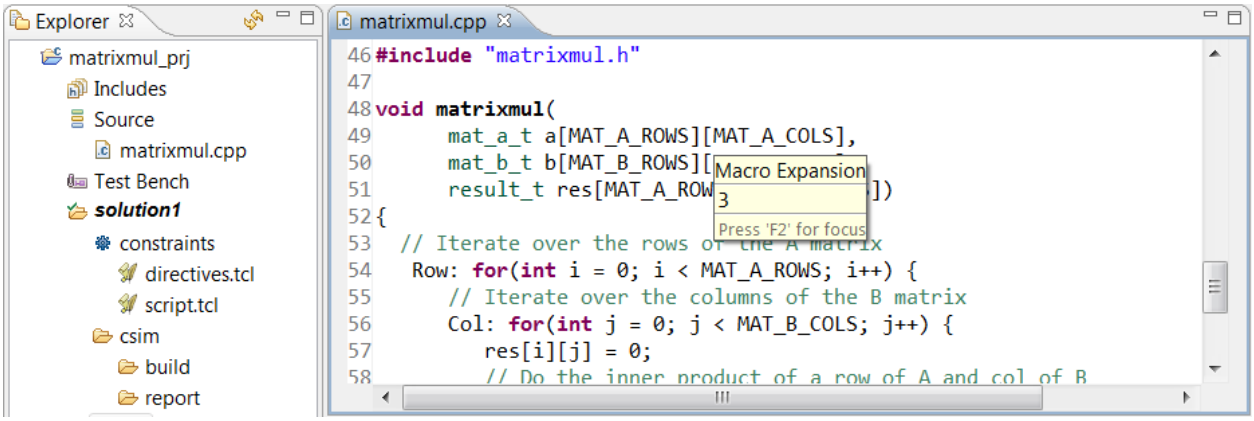


Figure 145 Source Code for the Matrix Multiplier

Step 2: Synthesize and Analyze the Design

1. Use the Run C Synthesize toolbar button or menu Solution > Run C Synthesis to synthesize the design to RTL.

When synthesis completes, the synthesis report opens (Figure 146) and the Performance estimates show:

- The interval is 80 clock cycles. Since there are 9 elements in each input array, the design it taking approximately 9 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 has a latency of and total 6 clock cycles for all iterations.
    - The loop above that takes 8 for each iteration and 24 cycles overall.
    - And the top-level loop has a latency 26 clock cycle per iteration for a total of 78 clock cycles for all iterations of the loop.

matrixmul_csynth.rpt

Performance Estimates

Summary of timing analysis

Target clock period (ns): 13.33
Clock uncertainty (ns): 1.67
Estimated clock period (ns): 9.57

Estimated latency (clock cycles)

Instance Latency

Instance	Module	Latency		Interval		Pipeline Type
		Min	Max	Min	Max	
-	matrixmul	79	79	80	80	none

Loop Latency

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	Min	Max		Achieved	Target		
- Row	78	78	26	-	-	3	no
+ Col	24	24	8	-	-	3	no
++ Product	6	6	2	-	-	3	no

Figure 146 Synthesis Report for the Matrix Multiplier

There are two possibilities to improve the initiation interval. Pipeline the loops or pipeline the entire function. We'll first start with pipelining the loops and then compare those results to pipelining the entire function.

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

Step 3: Pipeline the Product Loop

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

When pipelining nested loops, the greatest benefit is achieved by pipelining the inner most loop which processes a sample of data. High-Level Synthesis will automatically apply loop flattening: collapsing the nested loops, removing the loop transitions, allowing the outer loops to simple feed the inner loop with data.

4. In the Directives tab
    a. Select loop Product.
    b. Right-click with the mouse and select Insert Directive

c. In the Directives Editor dialog box activate the Directives drop-down menu at the top and select PIPELINE.
d. Press 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).

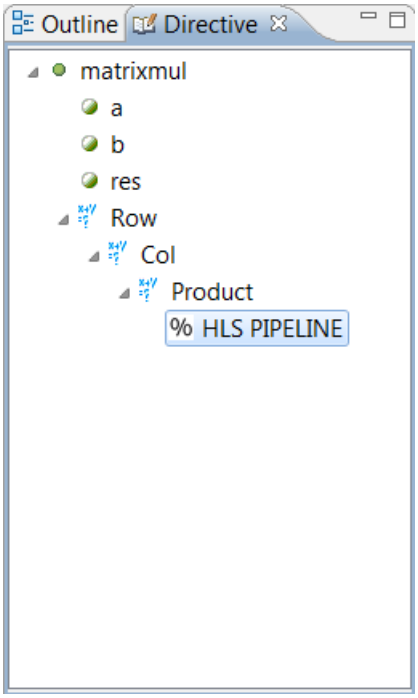


Figure 147 Initial Pipeline Directive

5. Use the Run C Synthesize toolbar button or menu Solution > Run C Synthesis to synthesizes the design to RTL.

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

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

The synthesis report (Figure 148) shows that although the Product loop is pipelined with an interval of 2, the interval of top-level loop is not pipelined.

matrixmul_csynth.rpt

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
default	13.33	9.57	1.67

Latency (clock cycles)

Summary

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

Detail

Instance

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 148 Matrixmul Initial Pipeline Report

The reason the top-level loop is not pipelined is because 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, the Analysis perspective can be used.

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 with the mouse and select Goto Source to see the view in Figure 149.

The write operation in state C1 is due to the code which sets res to zero before the Product loop. Since res is a function argument, this means it is a write to a port in the RTL: this operation must happen before the operations in loop Product are executed. Since it is not an internal operation, but has an impact on the IO behavior this operation cannot be moved or optimized. This prevents the Product loop being flattened in to the Row_Col.

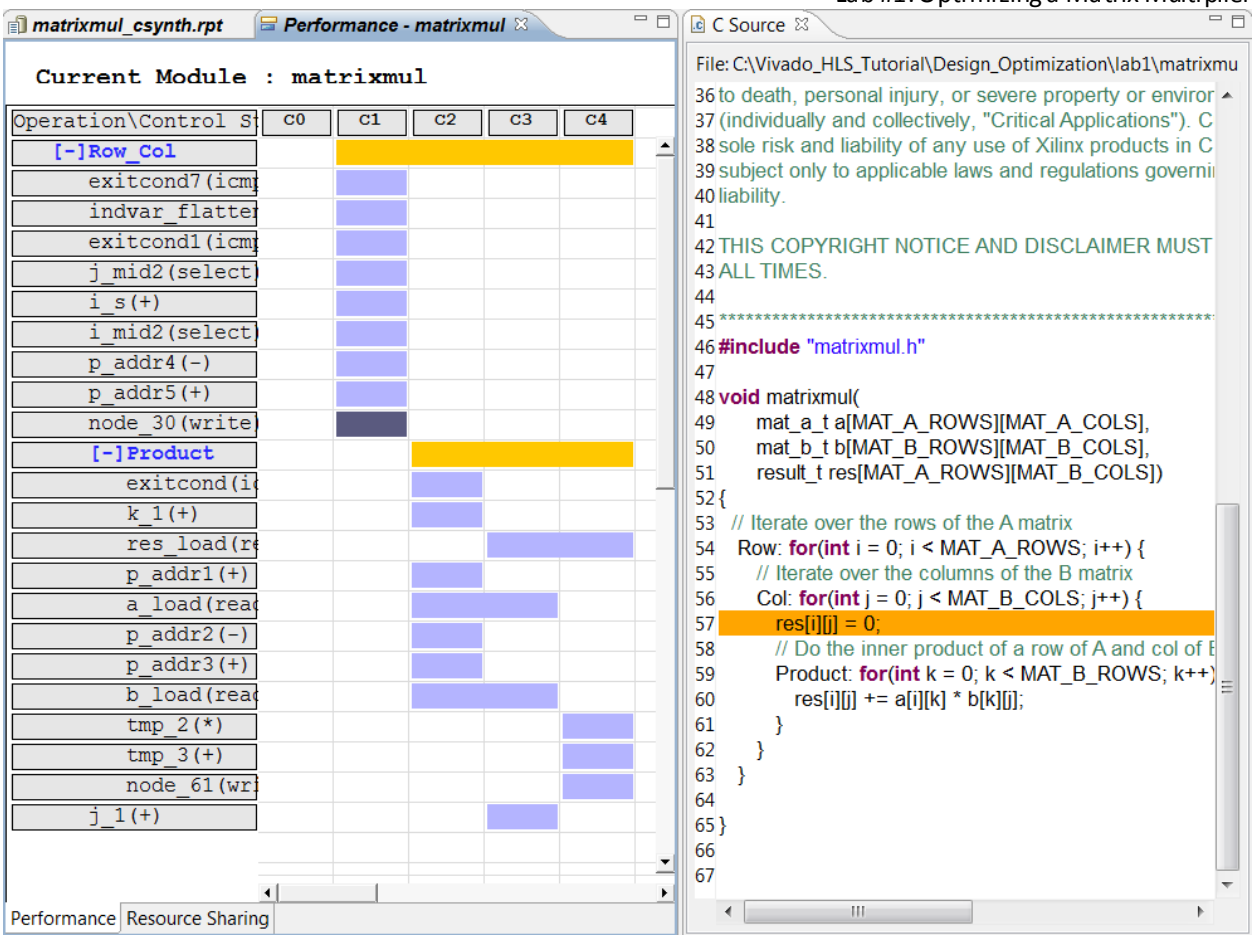


Figure 149 Matrixmul Initial Performance View

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

The message SCHED-68 tells us:

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

- The issue is a carried dependency. This is 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 149 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 BRAM 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 schedule. Figure 150 shows a number of copies of the schedule for the Product loop to highlight how this issue can be understood. Start with the basic view shown in the top-right. The operations on the res array, a two-cycle read and write, are highlighted in red.

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

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

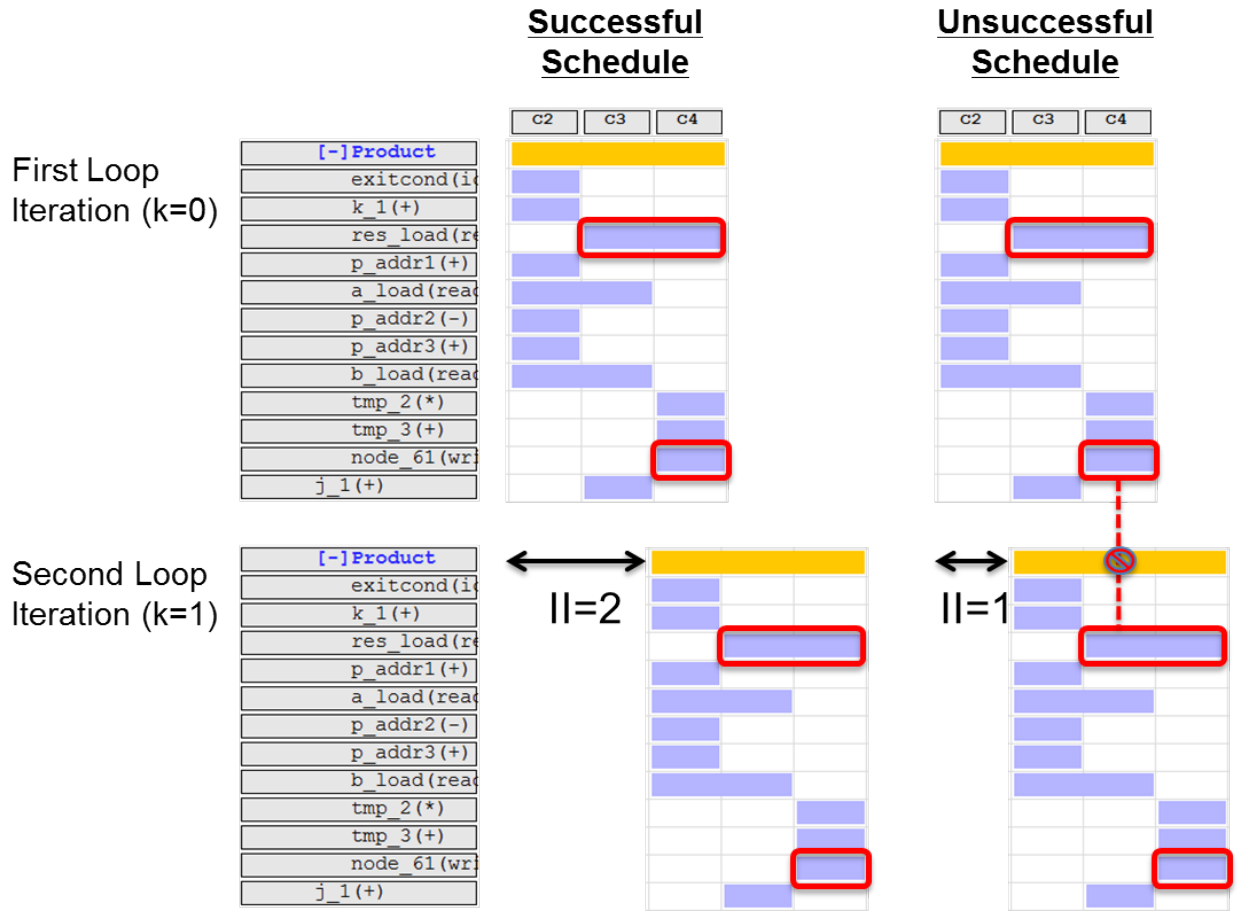


Figure 150 Carried Dependency Analysis

The Product loop cannot be pipelined with an initiation interval of 1. The next lab exercise shows how writing the code can remove this limitation (any technique which does not write back to the same BRAM). This lab exercise will seek to optimize the code as it is.

The next step in trying to pipeline the design is to pipeline the loop above, the Col loop. This will automatically unroll the Product loop creating more operators and hence more hardware resources but it will ensure 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 or use the menu Project > New Solution to create a new solution.
2. Since 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. Press Finish and accept the default solution name solution3.
4. Open the C source code matrixmul.cpp to make it visible in the Information pane.
5. In the Directives tab
    a. Select loop Col.
    b. Right-click with the mouse and select Insert Directive
    c. In the Directives Editor dialog box activate the Directives drop-down menu at the top and select PIPELINE.
    d. Press 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).

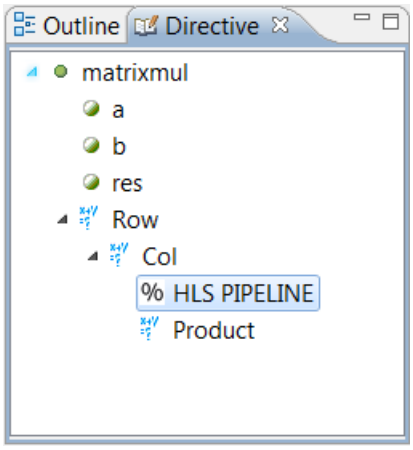


Figure 151 Col Pipeline Directive

6. Use the Run C Synthesize toolbar button or menu Solution > Run C Synthesis to synthesizes the design to RTL.

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

```
@I [XFORM-502] Unrolling all sub-loops inside loop 'Col'
(matrixmul.cpp:56) in function 'matrixmul' for pipelining.
@I [XFORM-501] Unrolling loop 'Product' (matrixmul.cpp:59) in function
'matrixmul' completely.
@I [XFORM-541] Flattening a loop nest 'Row' (matrixmul.cpp:54) in function
'matrixmul'.
...
...
@I [SCHED-61] Pipelining loop 'Row_Col'.
```

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

Reviewing the synthesis report shows, as noted above, that the interval for loop Row_Col is only two: the target is to process one sample every cycle. Once again, the Analysis perspective can be used 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 152. There are 3 read operations on array a. Two operations start in state C1 and a third read operation starts in state C2.

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

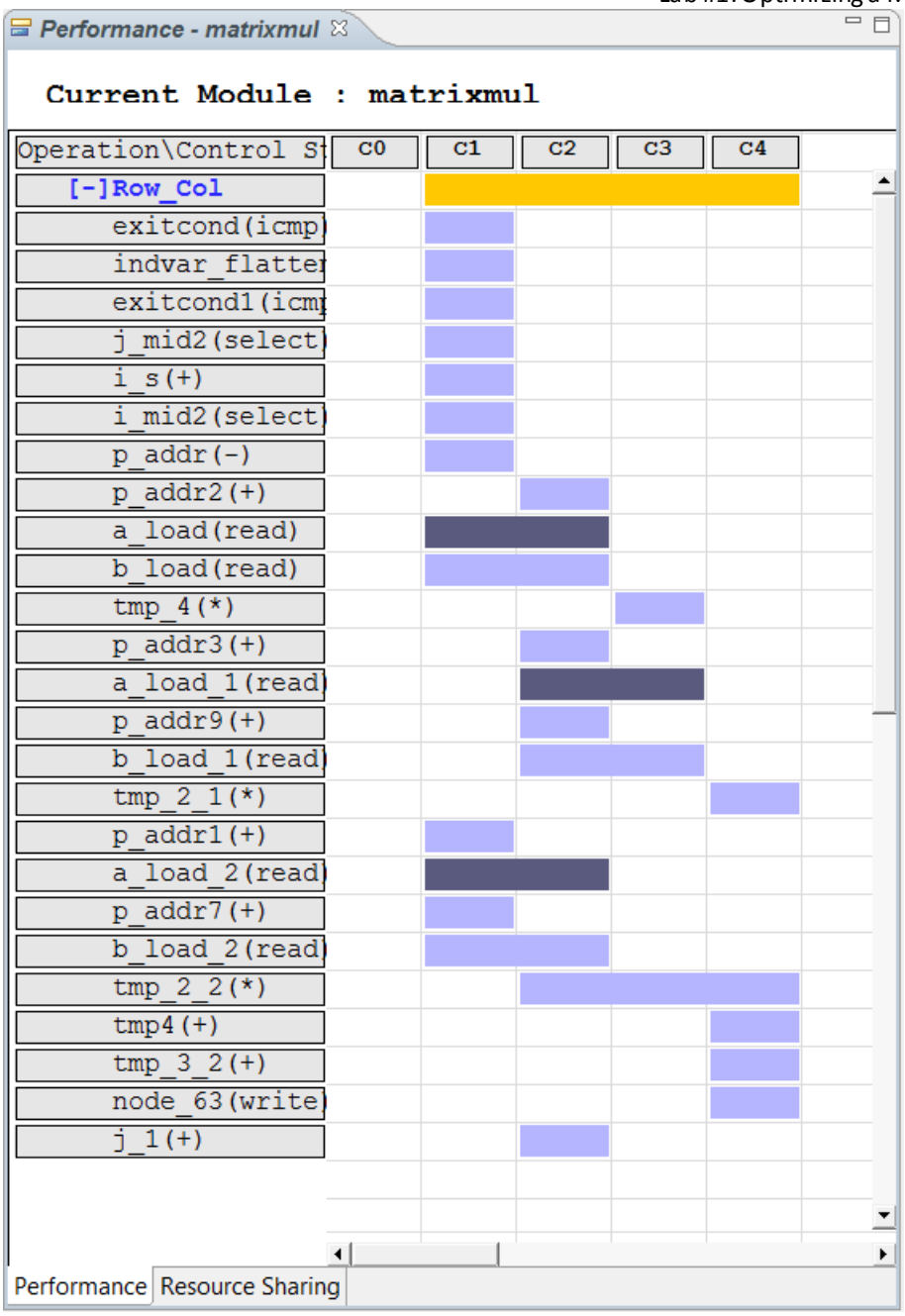


Figure 152 Matrixmul Pipeline Col Performance View

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

9. Click on the Resource Sharing tab.
10. Expand the memories to see the view shown in Figure 153.

In Figure 153 the 2 cycles 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 will occur with port b: it also has to perform 3 reads.

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

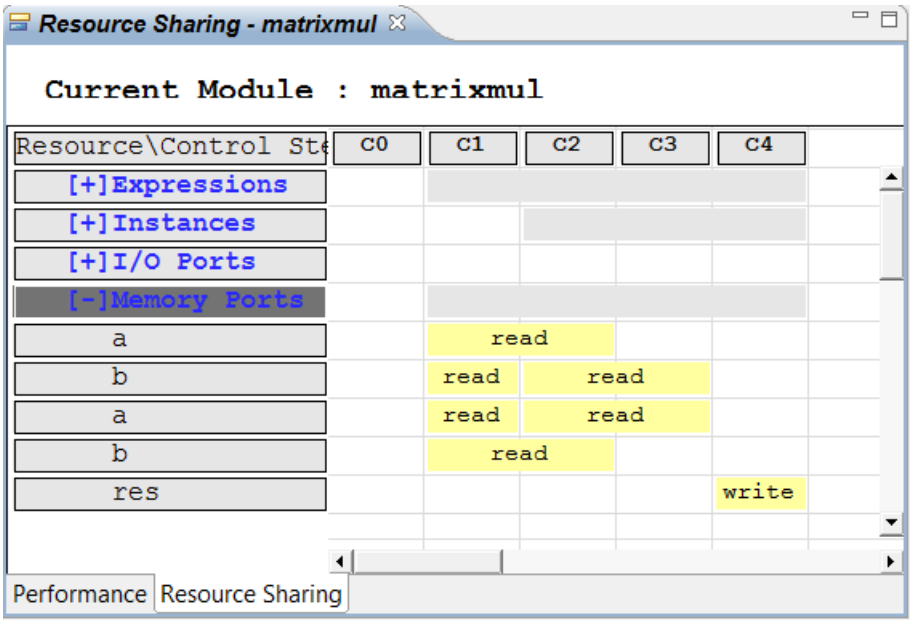


Figure 153 Matrixmul Pipeline Col Resource Sharing View

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

11. Return to the Synthesis perspective.

Step 5: Reshape the Arrays

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

Since the loop index for the Product loop is k, both arrays a 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 since it's access patterns is a[i][k] and for array b dimension 1 since it's access pattern is b[k][j].

Partitioning these arrays will create 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 BRAM outside this block must be reshaped in an identical manner: if this process is not done by HLS, the data must be arranged as:

- For array a: i elements, each of width data_word_size times k.
- For array b: j elements, each of width data_word_size times k.

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

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

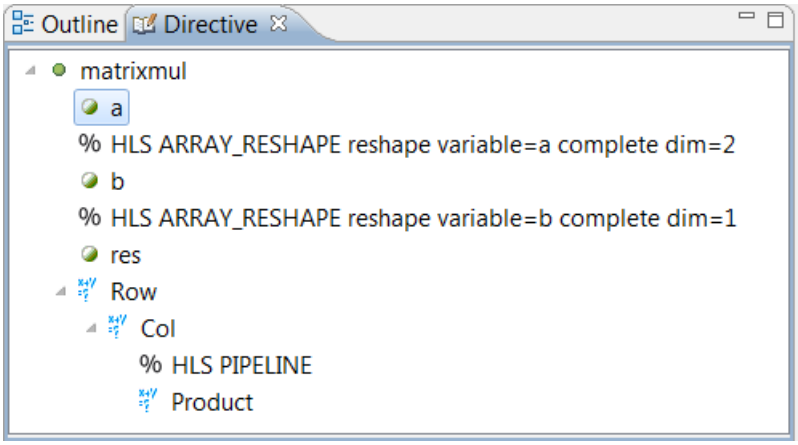


Figure 154 Array Reshape Directive

6. Use the Run C Synthesize toolbar button or menu Solution > Run C Synthesis to synthesizes 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 155).

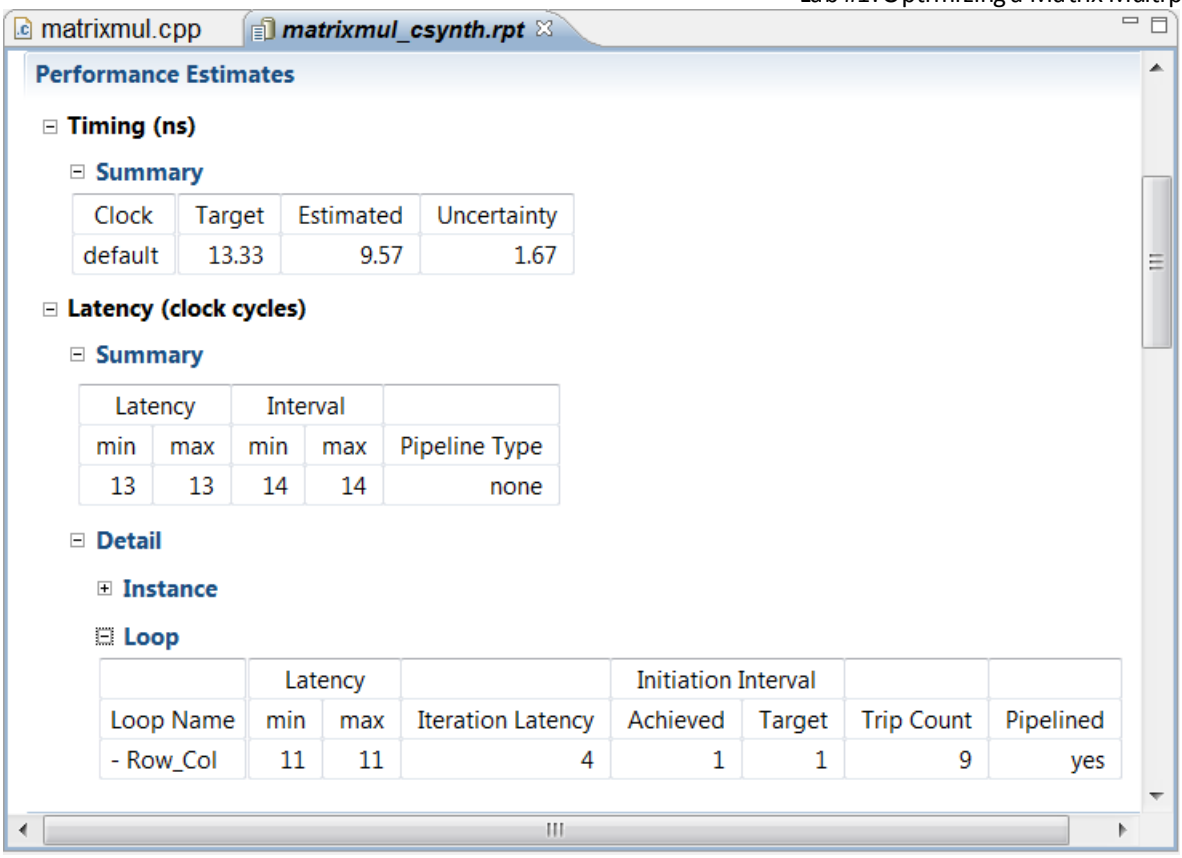


Figure 155 Optimized Loop Processing report

- The top-level module takes 13 clock cycles to complete.
- The Row_Col loop outputs a sample after 4 cycles (iteration latency).
- It then reads 1 sample every cycle (Initiation Interval).
- After 9 iterations/samples (Trip count) it completes all samples.
- 4 + 9 = 13 clock cycles

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

Now, let us change the interfaces from BRAM interfaces to FIFO interfaces to allow for streaming data.

Step 6: Apply FIFO Interfaces

1. Select the New Solution toolbar button or use the menu Project > New Solution to create a new solution.
2. Press Finish and accept the default solution name solution5.
3. Open the C source code matrixmul.cpp to make it visible in the Information pane.
4. In the Directives tab
    a. Select variable a.
    b. Right-click with the mouse and select Insert Directive
    c. In the Directives Editor dialog box activate the Directives drop-down menu at the top and select INTERFACE.

d. Use the mode drop-down menu to select ap_fifo
e. Press OK.

5. Repeat this process for variable b.
   a. Repeat this process for variable res.

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

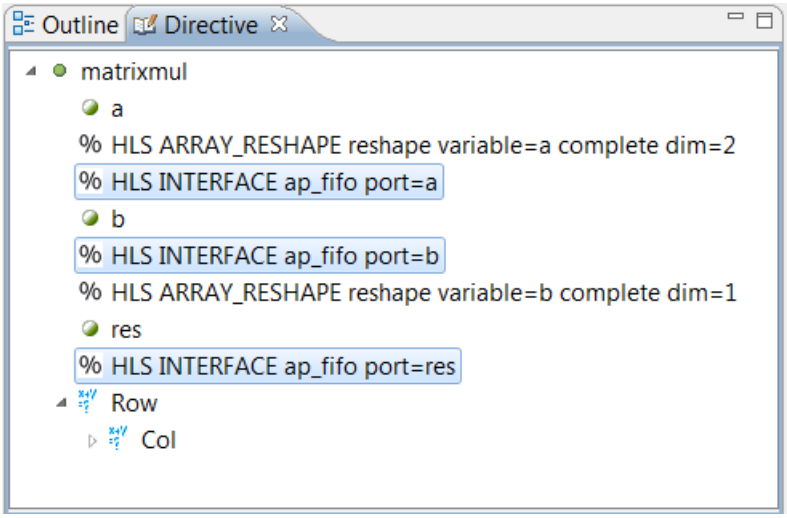


Figure 156 Matrixmul FIFO Directives

6. Use the Run C Synthesize toolbar button or menu Solution > Run C Synthesis to synthesizes the design to RTL.

Figure 157 shows the Console after trying to synthesize the design.

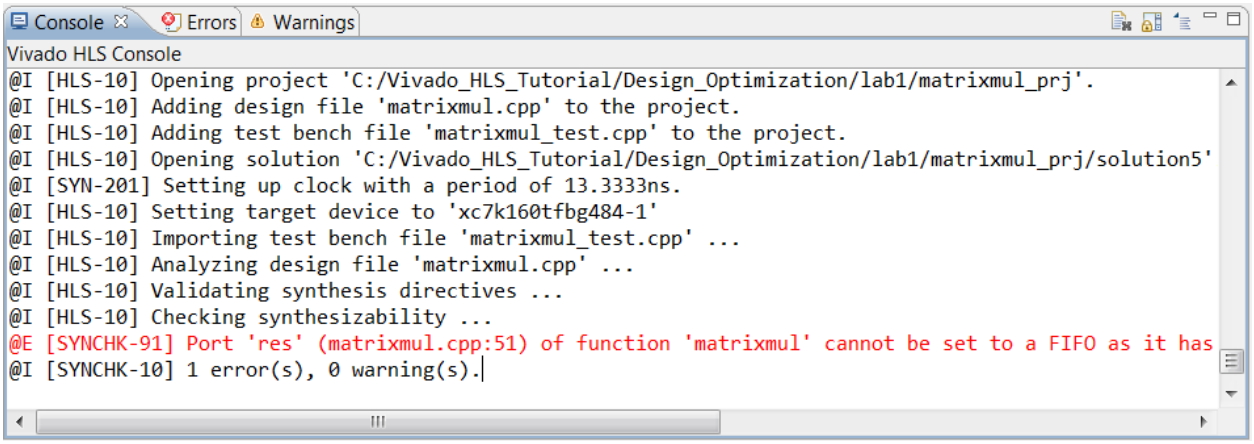


Figure 157 FIFO Synthesis Warning

From the code shown in Figure 158, 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] is not a streaming access pattern: this is random access.

matrixmul.cpp

```
52 {
53   // Iterate over the rows of the A matrix
54    Row: for(int i = 0; i < MAT_A_ROWS; i++) {
55       // Iterate over the columns of the B matrix
56       Col: for(int j = 0; j < MAT_B_COLS; j++) {
57          res[i][j] = 0;
58          // Do the inner product of a row of A and col of B
59          Product: for(int k = 0; k < MAT_B_ROWS; k++) {
60             res[i][j] += a[i][k] * b[k][j];
61          }
62       }
63    }
64
65 }
```

Figure 158 Matrixmul Code

Examining the code in Figure 158 shows 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 as the code currently enforces a certain order of reads and writes. Further optimization requires a re-write of the code and this is performed in Lab #2.

Before modifying the code, 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 or use the menu Project > New Solution to create a new solution.
2.

IMPORTANT: *In this step, copy the directives from solution4 as this does not have FIFO interfaces specified.*

3. Select solution4 from both the drop down menus in the Options section. The Solution Wizard should look like Figure 159.

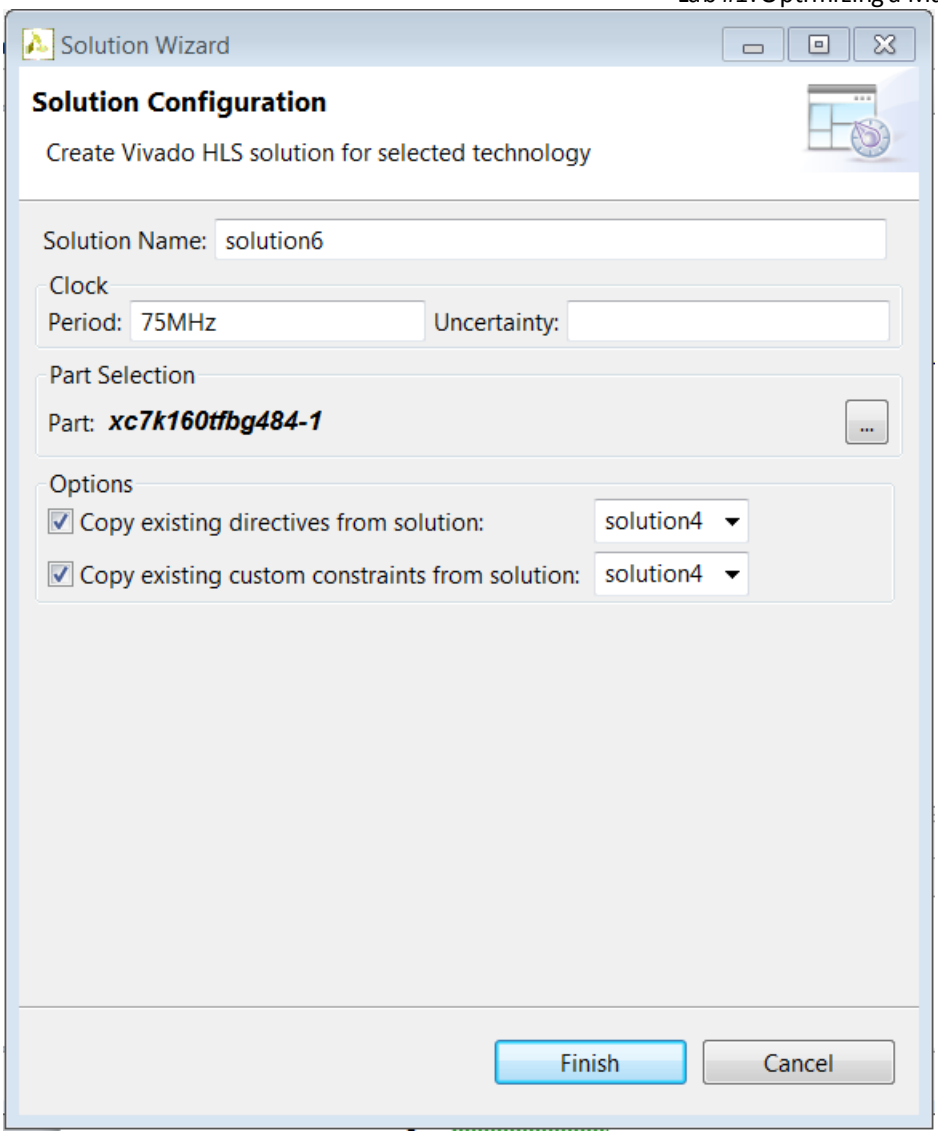


Figure 159 New Solution Based on Solution4 Directives

4. Press Finish and accept the default solution name solution6.
5. Open the C source code matrixmul.cpp to make it visible in the Information pane.
6. In the Directives tab
    a. Select the pipeline directive on loop Col
    b. Right-click with the mouse and select Remove Directive
    c. Select the top-level function matrixmul
    d. Right-click with the mouse and select Insert Directive
    e. In the Directives Editor dialog box activate the Directives drop-down menu at the top and select PIPELINE.
    f. Press OK.

The Directives tab should appear as Figure 160.

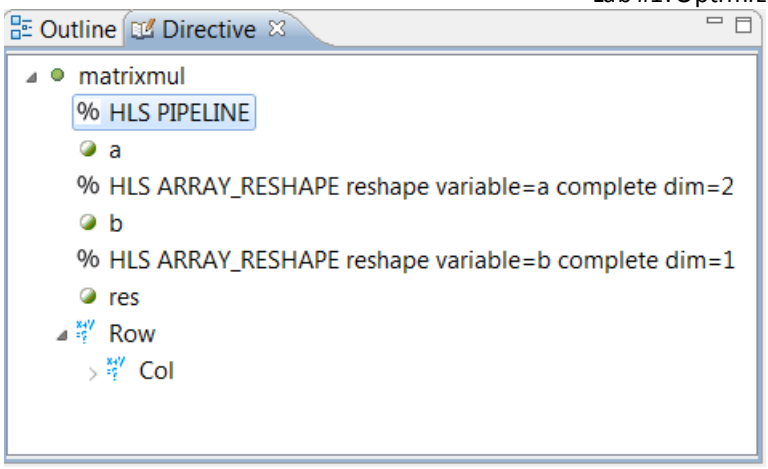


Figure 160 Directives for Solution6

6. Use the Run C Synthesize toolbar button or menu Solution > Run C Synthesis to synthesizes the design to RTL.
7. Use the Compare Reports toolbar button or menu Project > Compare Reports
   a. Add solution4
   b. Add solution6
   c. Press OK

The comparison of solutions 4 and 6 is shown in Figure 161.

matrixmul.compare

Performance Estimates

Timing (ns)

Clock		solution6	solution4
default	Target	13.33	13.33
	Estimated	9.57	9.57

Latency (clock cycles)

		solution6	solution4
Latency	min	7	13
	max	7	13
Interval	min	5	14
	max	5	14

Utilization Estimates

	solution6	solution4
BRAM_18K	0	0
DSP48E	27	3
FF	517	83
LUT	37	40

Figure 161 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, due to the fact that 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 will be unrolled: it's a requirement for pipelining. The pipelined function design can process a new set of 9 samples every 5 clock cycles: this exceeds the requirements of 1 sample per second because the default behavior of High-Level Synthesis is to produce a design with the highest performance.

The pipelined function will be the best performance, however if it exceeds the required performance, it may then take multiple addition directives to slow the design down. Pipelining loops give you an easy way to control resources, with the option of partially unrolling the design to meet performance.

Lab #2: C Code Optimized for IO 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 to write 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 used in Lab#1. The following explains how the C code was update.

Figure 162 shows the IO access pattern for the code in Lab#1. Out of necessity the address values are shown is a small font.

As variables i, j and k iterate from 0 to 3, the lower part of Figure 162 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.

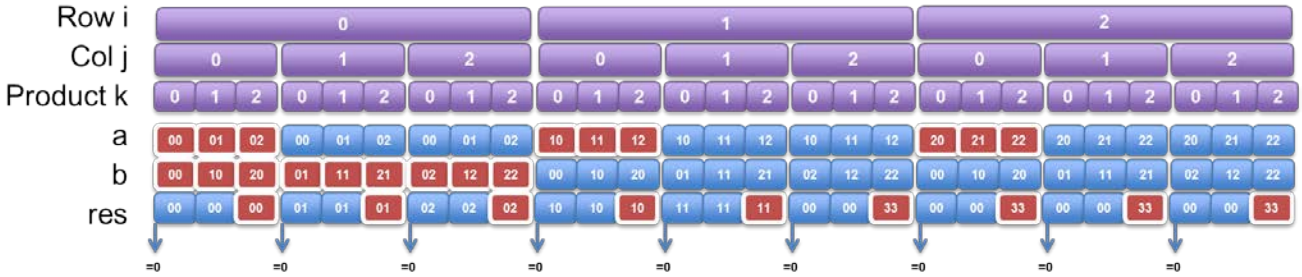


Figure 162 Lab#1 Matrix Multiplier Address Accesses

To have a hardware design with sequential streaming accesses, the ports accesses can only be those shown highlighted in red. For the read ports, the data must be cached internally after these reads to ensure the design does not have to re-read the port. For the write port res, the data must be save in a temporary variable and only written at the locations 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 163.
2. Create a new Vivado HLS project by typing vivado_hls –f run_hls.tcl

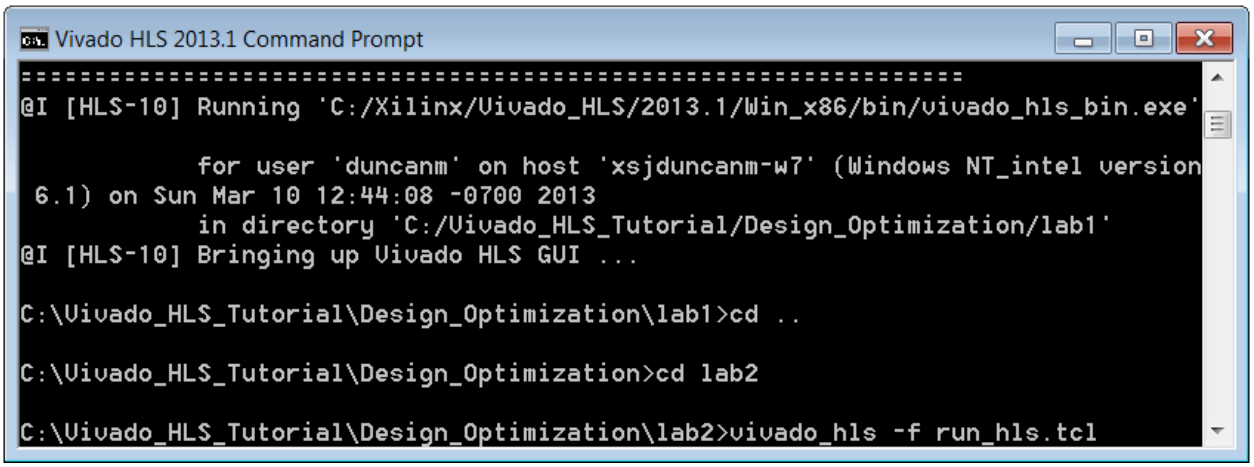


Figure 163 Setup for Interface Synthesis Lab#2

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 on matrixmul.cpp to open the code as shown in Figure 164.

matrixmul.cpp

```
52 {
53 #pragma HLS ARRAY_RESHAPE variable=b complete dim=1
54 #pragma HLS ARRAY_RESHAPE variable=a complete dim=2
55 #pragma HLS INTERFACE ap_fifo port=a
56 #pragma HLS INTERFACE ap_fifo port=b
57 #pragma HLS INTERFACE ap_fifo port=res
58   mat_a_t a_row[MAT_A_ROWS];
59   mat_b_t b_copy[MAT_B_ROWS][MAT_B_COLS];
60   int tmp = 0;
61
62   // Iterate over the rowa of the A matrix
63   Row: for(int i = 0; i < MAT_A_ROWS; i++) {
64     // Iterate over the columns of the B matrix
65     Col: for(int j = 0; j < MAT_B_COLS; j++) {
66 #pragma HLS PIPELINE
67       // Do the inner product of a row of A and col of B
68       tmp=0;
69       // Cache each row (so it's only read once per function)
70       if (j == 0)
71         Cache_Row: for(int k = 0; k < MAT_A_ROWS; k++)
72           a_row[k] = a[i][k];
73
74        // Cache all cols (so they are only read once per function)
75      if (i == 0)
76             Cache_Col: for(int k = 0; k < MAT_B_ROWS; k++)
77                b_copy[k][j] = b[k][j];
78
79       Product: for(int k = 0; k < MAT_B_ROWS; k++) {
80         tmp += a_row[k] * b_copy[k][j];
```

Figure 164 C Code with updated IO accesses

Review the code and confirm the following aspects:

- The directives from Lab#1, including the FIFO interfaces, are specified in the code as pragmas.
- For-loops have been added to cache the rol and column reads.
- A temporary variable is used for the accumulation and port res is only written to when the final results is computed for each value.
- Since 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 will be automatically unrolled.

Synthesize the design and verify the RTL using co-simulation.

5. Use the Run C Synthesize toolbar button or menu Solution > Run C Synthesis to synthesize the design to RTL.
6. When synthesis completes, use the Run C/RTL Cosimulation toolbar button or menu Solution > Run C/RTL Cosimulation to launch the Cosimulation Dialog box.
7. Press OK to start RTL verification.

The design has been now been fully synthesized to read one sample every clock cycle using streaming FIFO interfaces.

This completes this tutorial on using arbitrary precision types.

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.

RTL Verification

Overview

Verification of the RTL created by High-Level Synthesis can be automatically performed. In addition, RTL verification can be used to generate trace files, showing the activity of the waveforms in the RTL design. These waveforms can be used for analyzing and understanding the RTL output. This tutorial covers all aspects of the RTL verification process.

RTL verification is performed using both the RTL output by High-Level Synthesis (Verilog, VHDL or SystemC) and the C test bench. For this reason, RTL verification is often referred to as "cosimulation" or "C/RTL cosimulation" since both C and RTL are used in the verification.

This tutorial consists of three lab exercises.

Lab1

Learn the steps for performing RTL verification and understand the importance of the C test bench in verifying the RTL.

Lab2

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

Lab3

Create RTL trace files and analyze them using a third-party RTL simulator. This lab requires a license for Mentor Graphics ModelSim simulator. (An alternative third-party simulator can be used with minor modifications to the steps).

Tutorial Design Description

The tutorial design file can be downloaded from the Xilinx website. Refer to the information in Obtaining the Tutorial Designs.

This tutorial uses the design files in the tutorial directory **Vivado_HLS_Tutorial\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 will explain the basic operations for RTL verification and highlight the importance of the C test bench.

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

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

Step 1: Create and Open the Project

1. Open the Vivado HLS Command Prompt.
    a. On Windows use **Start** > **All Programs** > **Xilinx Design Tools** > **Vivado 2013.1** > **Vivado HLS** > **Vivado HLS 2013.1 Command Prompt (**Figure 165).
    b. On Linux, open a new shell.

Figure 165 Vivado HLS Command Prompt

2. Using the command prompt window (Figure 166), 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 166.

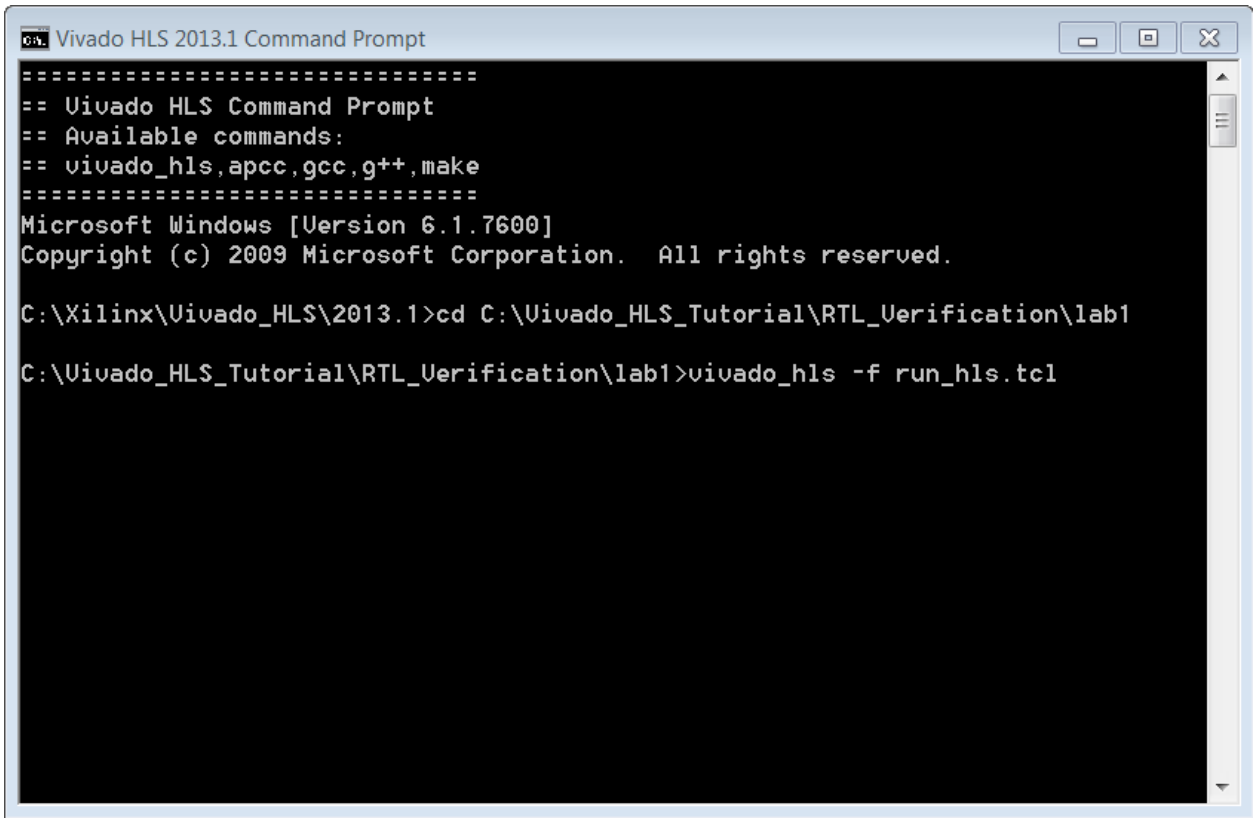


Figure 166 Setup the RTL Verification Tutorial Project

4. When Vivado HLS completes, open the project in the Vivado HLS GUI using the command vivado_hls –p dct_prj as shown in Figure 167.

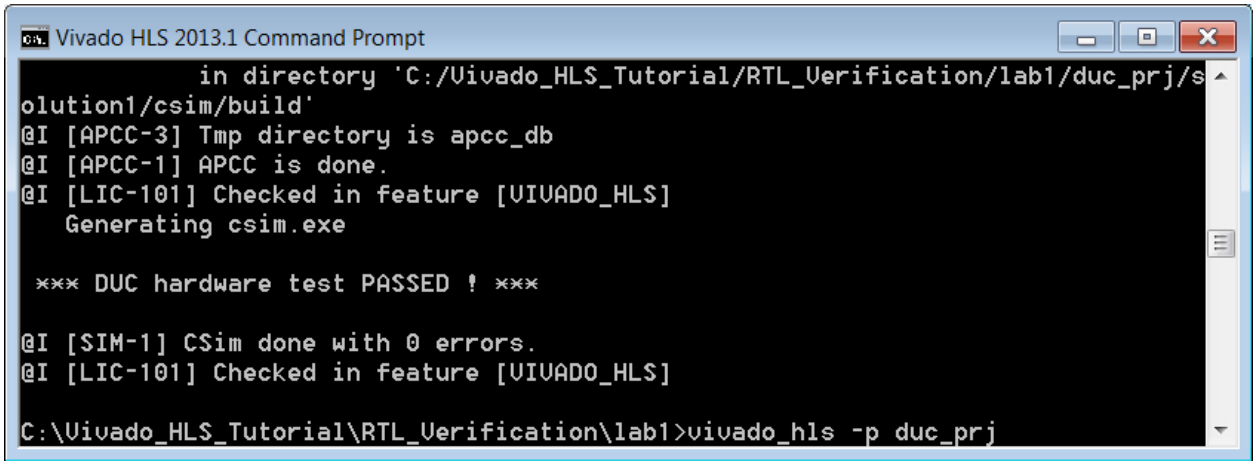


Figure 167 Open RTL Verification Project for Lab#1

Step 2: Perform RTL Verification

8. Use the Run C Synthesize toolbar button or menu Solution > Run C Synthesis to synthesize the design to RTL.
9. When synthesis completes, use the Run C/RTL Cosimulation toolbar button (Figure 168) or menu Solution > Run C/RTL Cosimulation to launch the Cosimulation Dialog box.

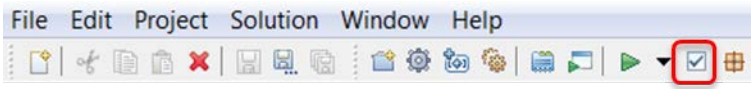


Figure 168 Run C/RTL Cosimulation Toolbar button

The Cosimulation Dialog box is shown in Figure 169.

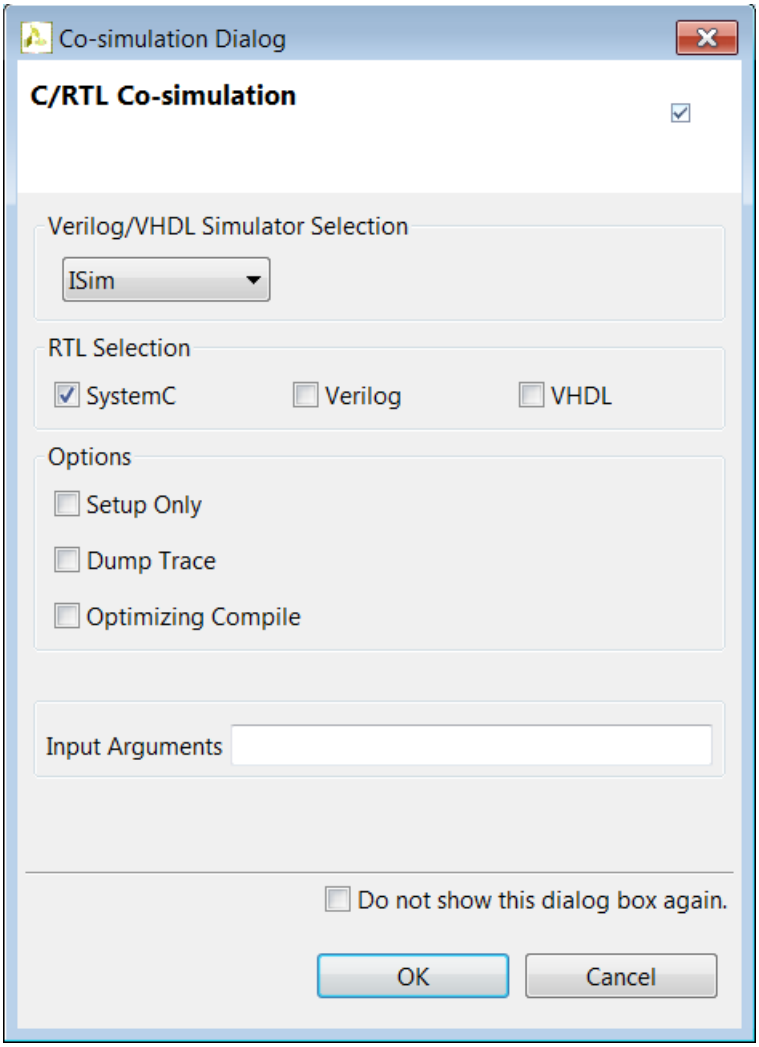


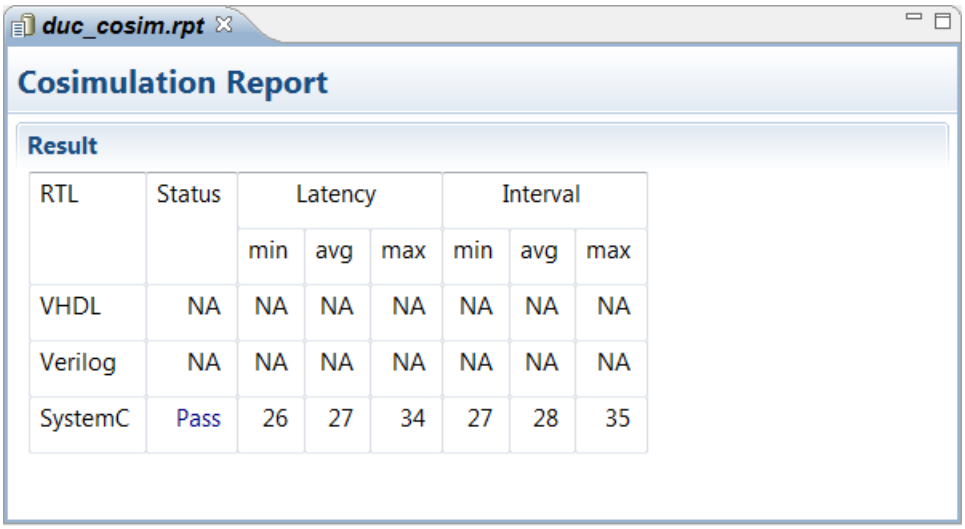
Figure 169 Cosimulation Dialog Box

The drop-down menu allows the RTL simulator to be selected for HDL simulation. For this exercise the SystemC RTL will be used for cosimulation and no HDL simulator is required: it can be left in the default state or changed, it will make no difference in this first lab.

The RTL Selection has SystemC selected by default. In this exercise the SystemC RTL will be used for simulation. Since this can be compiled with the built-in C compiler, no HDL simulator will be used for the simulation.

10. Press OK to start RTL verification.

When RTL Verification completes, the simulation report opens automatically (Figure 170). The report indicates if the simulation passed or failed. In addition, the report indicated the measured latency and interval.



RTL	Status	Latency			Interval		
		min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	NA	NA	NA	NA	NA	NA	NA
SystemC	Pass	26	27	34	27	28	35

Figure 170 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 the following messages 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 will show 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 may 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 is re-applied to the C test bench to check the results. Once again, any message output by the C test bench can be seen in the console. In this second execution of the C test bench, the RTL results are being used by the C test bench and ideally checked. 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 ideally check the results and how message SIM-1000, let's modify the C test bench.

Step 3: Modify the C test bench

1. Expand the Test Bench folder in the Explorer pane (171).
2. Double-click on dut_test.c to open the C test bench in the Information pane.
3. Scroll to the end of the file to see the view shown in 171.

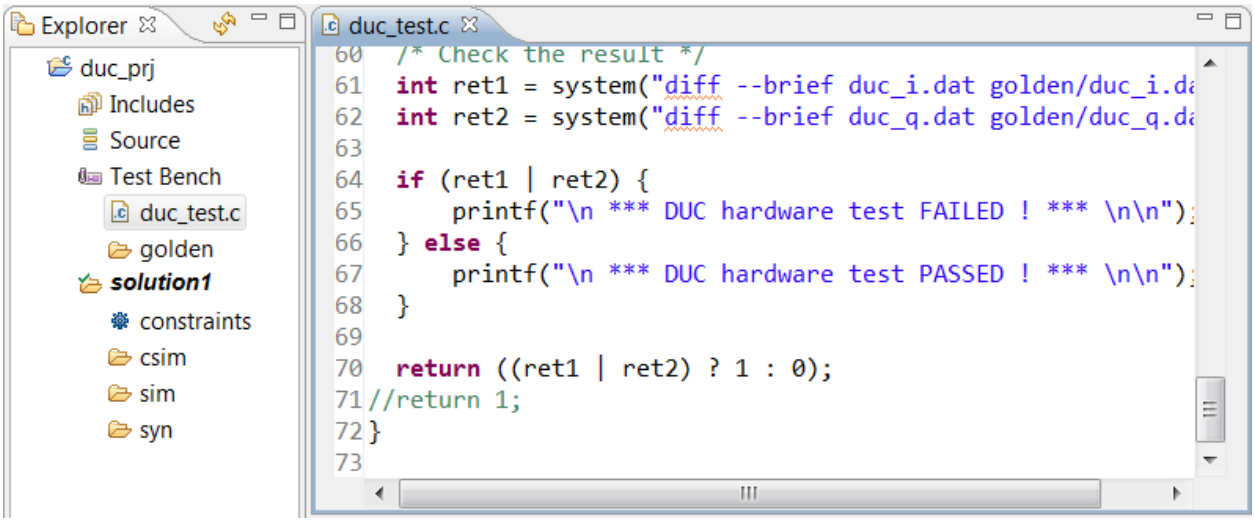


Figure 171 RTL Test bench

4. Edit the return statement to match Figure 172 and ensure the test bench always returns the value 1.

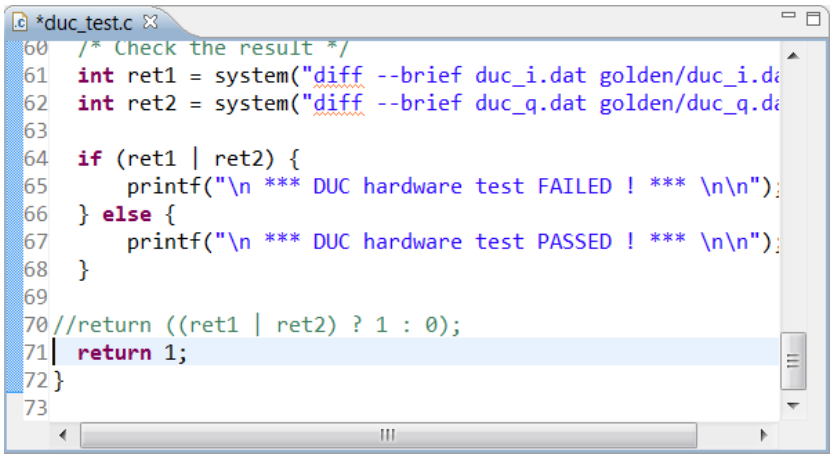


Figure 172 Modified RTL Test bench

5. Save the file.
6. Use the Run C Synthesize toolbar button or menu Solution > Run C Synthesis to synthesize the design to RTL.
7. Use the Run C/RTL Cosimulation toolbar button or menu Solution > Run C/RTL Cosimulation to launch the Cosimulation Dialog box.
8. Leave the Cosimulation options at their default value and press OK to execute the RTL cosimulation.

When RTL cosimulation completes, the cosimulation report opens and says the verification has failed (Figure 173).

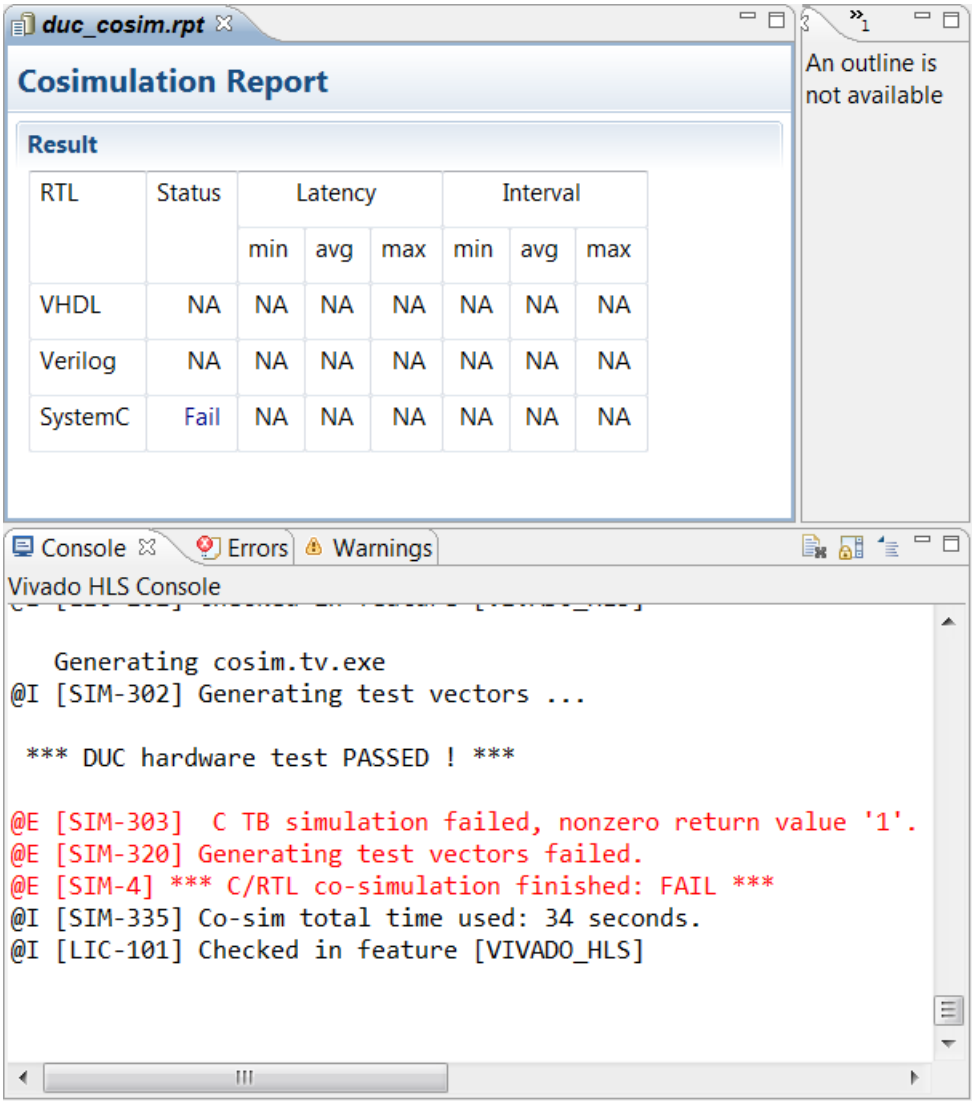


Figure 173 Cosimulation Report Failure

In Figure 173 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 by comparing the output files created by the RTL simulation with the golden results. The RTL simulation is executed in the simulation

directory wrapc directory, shown inside the solution directory in Figure 174, where the output files are shown highlighted.

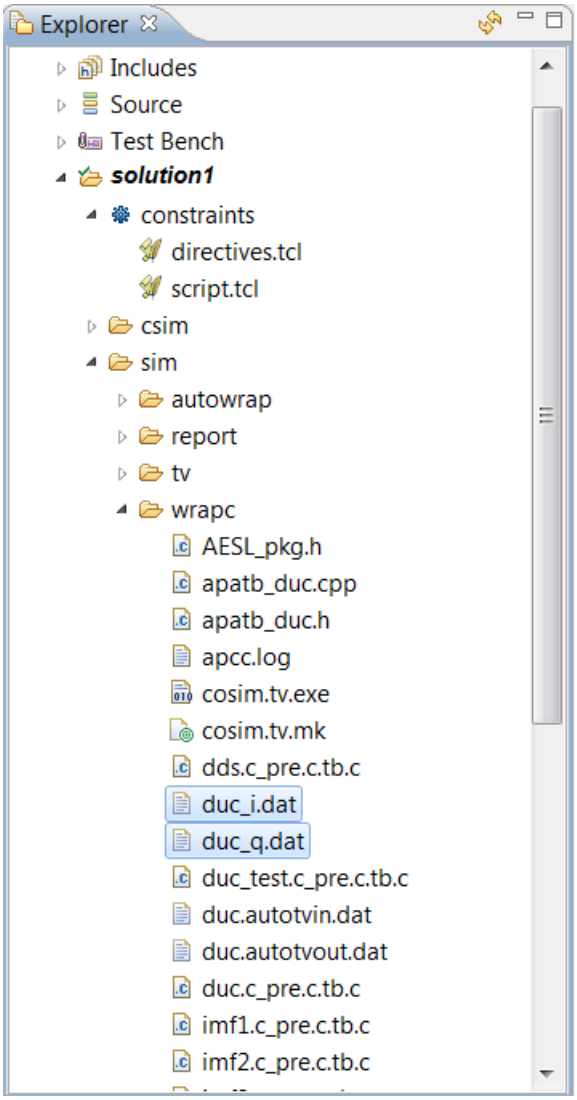


Figure 174 Cosimulation Output Files

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 ensured RTL verification (and C simulation if it was performed) would always report a failure.

To have the RTL results automatically verified, the C test bench should always check the output from the C function to be synthesized and return a 0 (zero) if they are correct OR return any other value if they are not correct.

When RTL Verification is performed, the same testing will be performed in the test bench and the output from the RTL block will be 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 will explain how RTL trace files can be generated and viewed using the Vivado Design Suite.

Step 1: Create an RTL Trace File using Xsim

1. From the Vivado HLS command prompt used in Lab#1, change to the lab2 directory as shown in Figure 175.
2. Create a new Vivado HLS project by typing vivado_hls –f run_hls.tcl

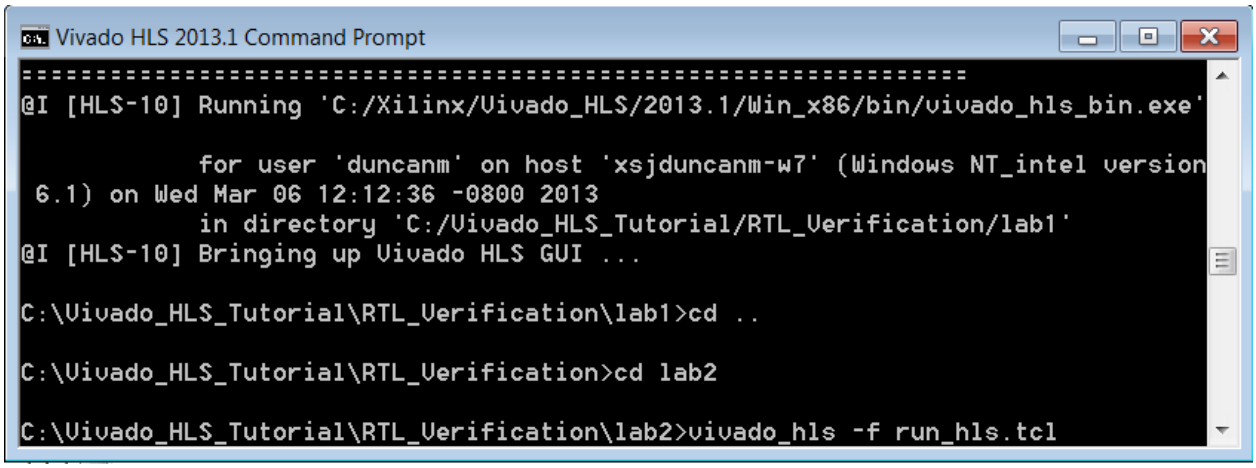


Figure 175 Setup for RTL Verification Lab#2

3. Open the Vivado HLS GUI project by typing vivado_hls –p duc_prj
4. Use the Run C Synthesize toolbar button or menu Solution > Run C Synthesis to synthesize the design to RTL.
5. Use the Run C/RTL Cosimulation toolbar button or menu Solution > Run C/RTL Cosimulation to launch the Cosimulation Dialog box.

This exercise could use SystemC as in Lab#1, however the trace file produced by a SystemC simulation is a VCD file. In this case, we wish to produce a trace file we can open using the Vivado Simulator (Xsim).

6. In the Co-simulation Dialog window:
   a. Select Xsim from the Verilog/VHDL Simulator Selector (Figure 176).
   b. Unselect SystemC.
   c. Select Verilog.
   d. Select the Dump Trace option, to have the options shown in Figure 176.
   e. Press OK to execute RTL cosimulation.

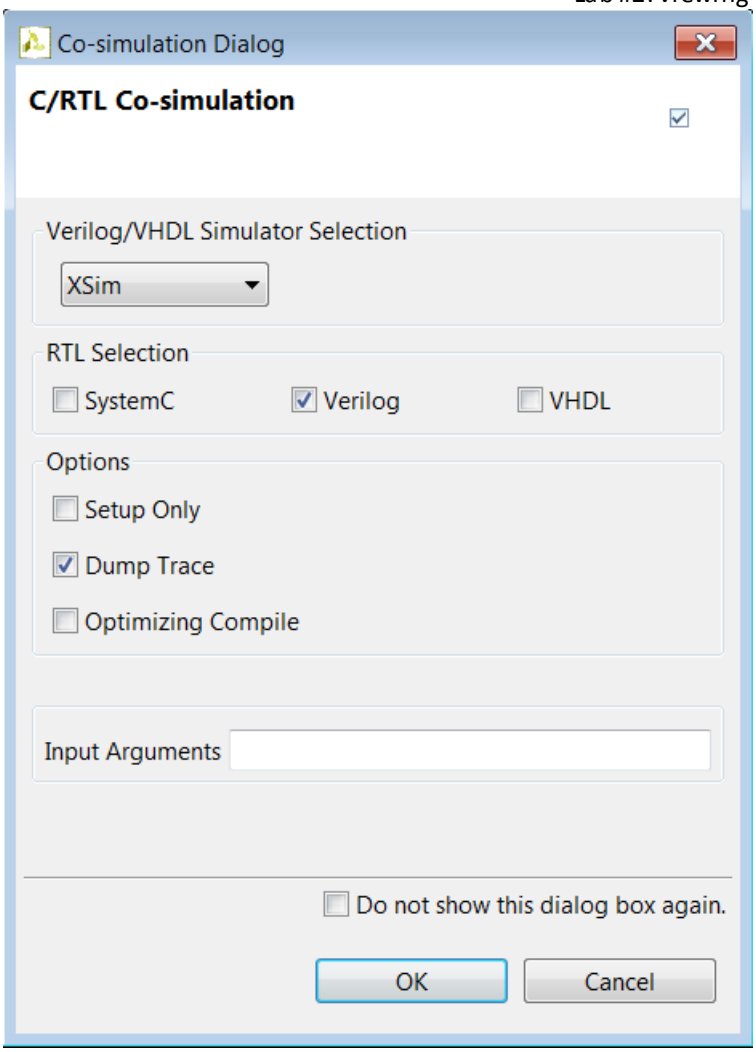


Figure 176 Cosimulation Dialog Box For Lab #2

When RTL verification completes the cosimulation report automatically opens showing the Verilog simulation has passed (and the measured latency and interval). In addition, because the Dump Trace option was used with the Xsim simulator option and Verilog was selected, two traces file are now present in the Verilog simulation directory and shown highlighted in Figure 177.

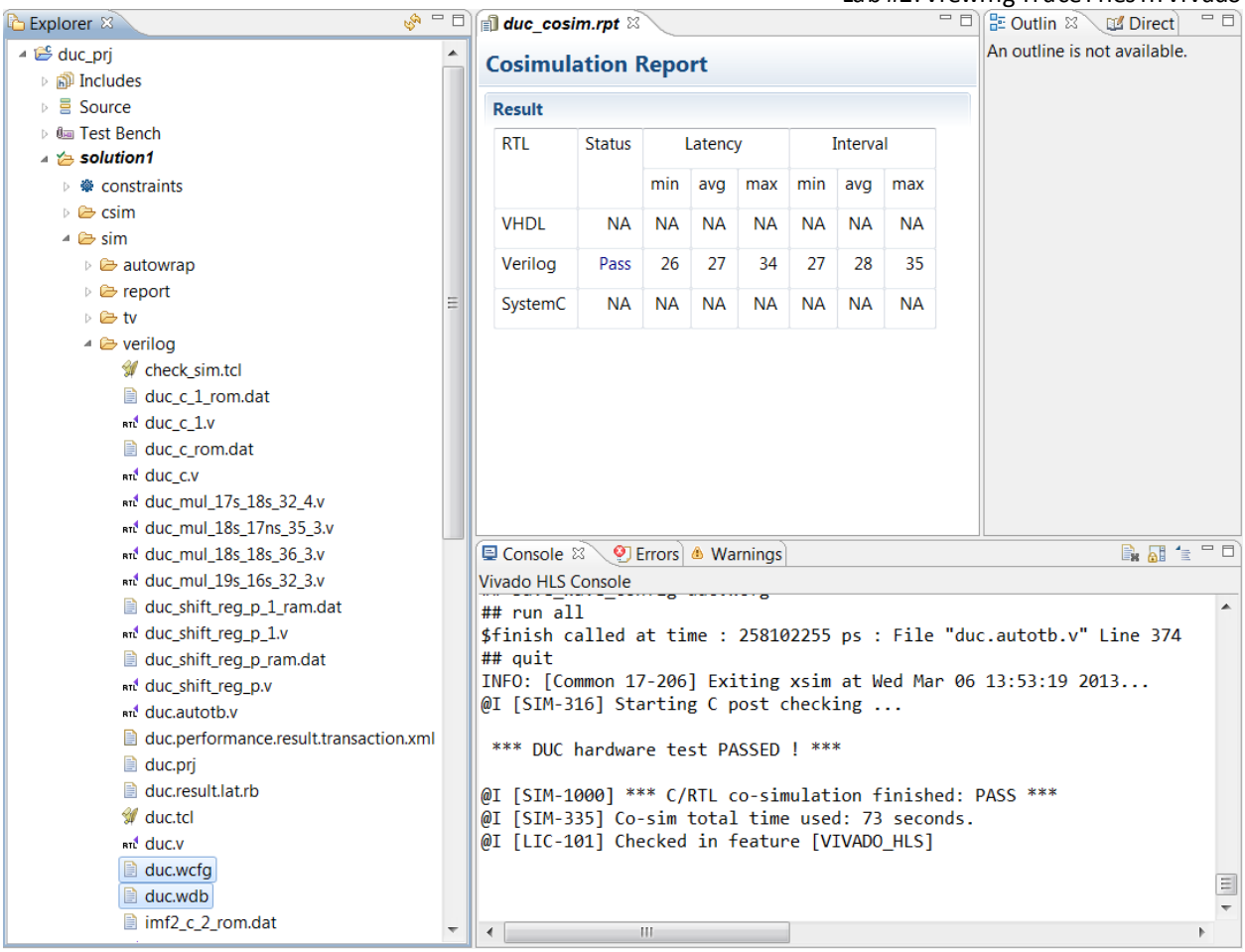


Figure 177 Verilog Xsim Cosimulation Results

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

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

Step 2: View the RTL Trace File in Vivado

1. Launch the Vivado Design Suite (not Vivado HLS):
   a. On Windows use **Start** > **All Programs** > **Xilinx Design Tools** > **Vivado 2013.1** > **Vivado 2013.1**
   b. On Linux, type vivado in the shell.
2. In the Vivado Tcl Console enter the following commands, as shown in Figure 178. This example assumes the top-level tutorial directory is C:\Vivado_HLS_Tutorial :
   a. cd /Vivado_HLS_Tutorial/RTL_Verification/lab2/duc_prj/solution1/sim/verilog
   b. current_fileset
   c. open_wave_database duc.wdb
   d. open_wave_config duc.wcfg

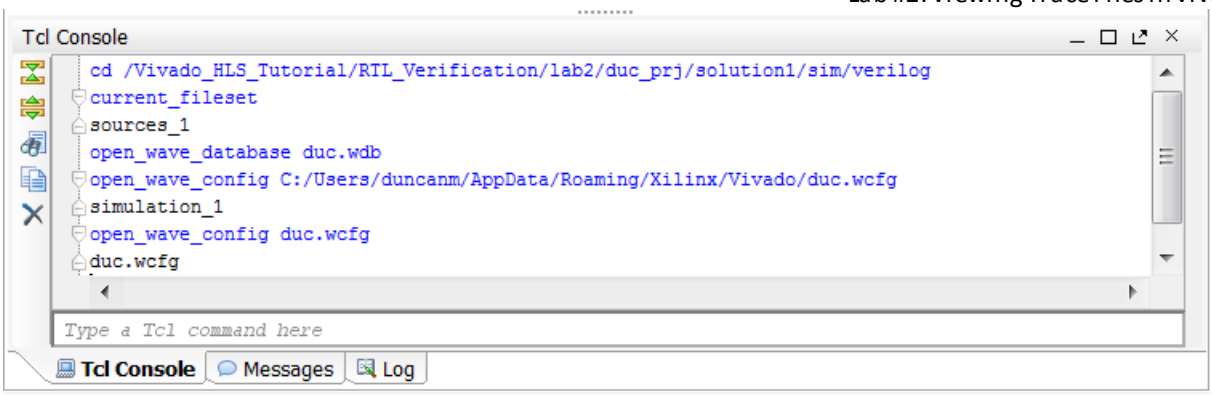


Figure 178 Opening the Trace File in Vivado

The waveforms can then be viewed in the waveform viewer. Figure 179 shows the zoomed waveforms where the output data ports and their associated IO protocol signals (output valid signals) are shown highlighted.

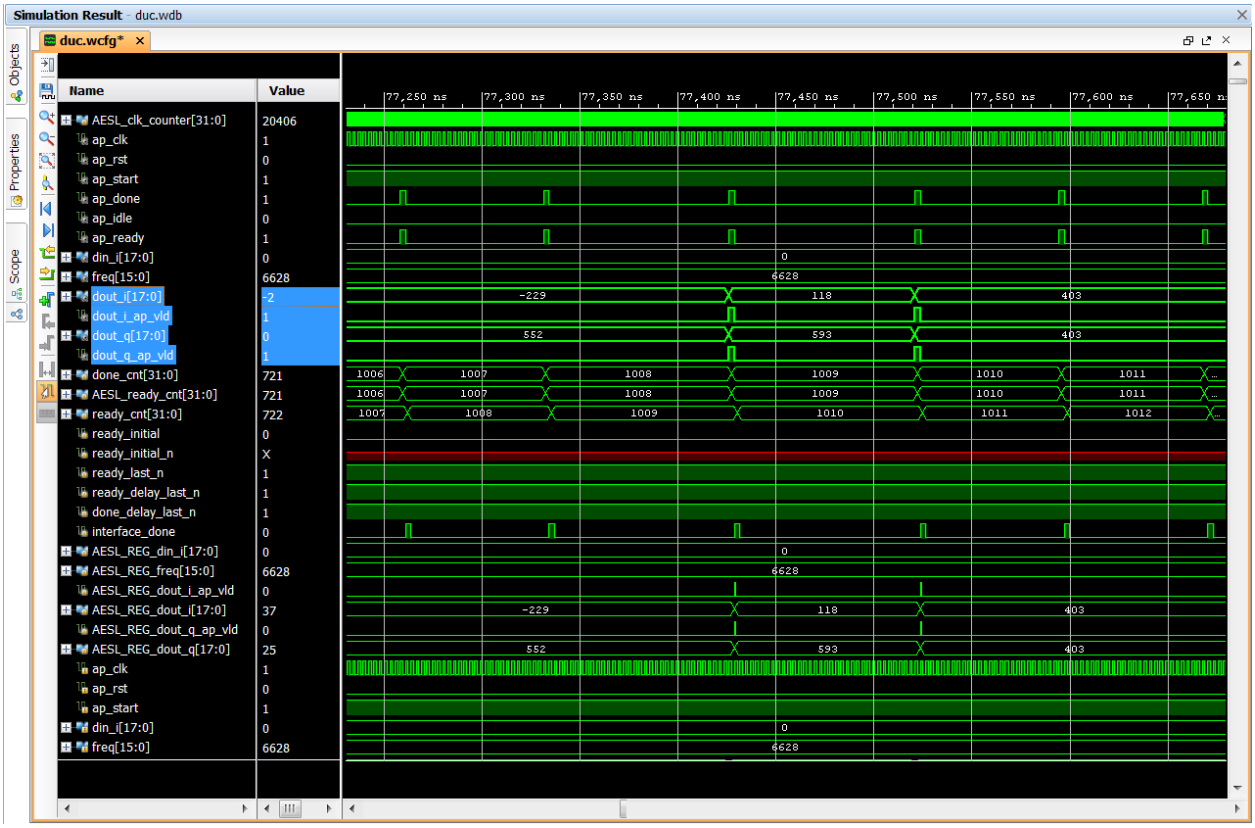


Figure 179 Analyzing the RTL Trace File

3. Exit and close the Vivado GUI.
4. Type exit to close the Vivado Tcl command prompt.

Lab #3: Viewing Trace Files in ModelSim

This exercise will explain how RTL trace files can be generated and viewed 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 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 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. Use the Run C Synthesize toolbar button or menu Solution > Run C Synthesis to synthesize the design to RTL.
5. Use the Run C/RTL Cosimulation toolbar button or menu Solution > Run C/RTL Cosimulation to launch the Cosimulation Dialog box.

This exercise uses the Mentor Graphics ModelSim RTL simulator. The path to the simulator executable must be set in your system search path.

6. In the Co-simulation Dialog window:
    a. Select ModelSim from the Verilog/VHDL Simulator Selector.
    b. Unselect SystemC.
    c. Select VHDL.
    d. Select the Dump Trace option, to have the options shown in Figure 180.
    e. Press OK to execute RTL cosimulation.

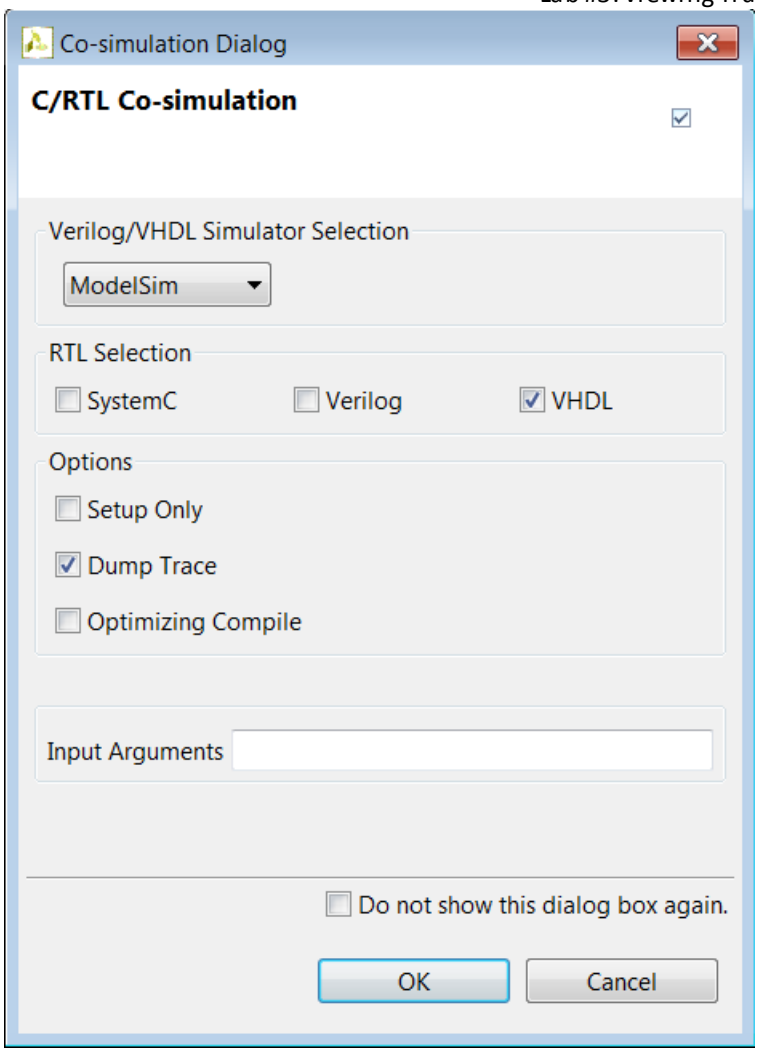


Figure 180 Cosimulation Dialog Box For Lab #3

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 VHDL was selected, a trace file is now present in the VHDL simulation directory and shown highlighted in Figure 181.

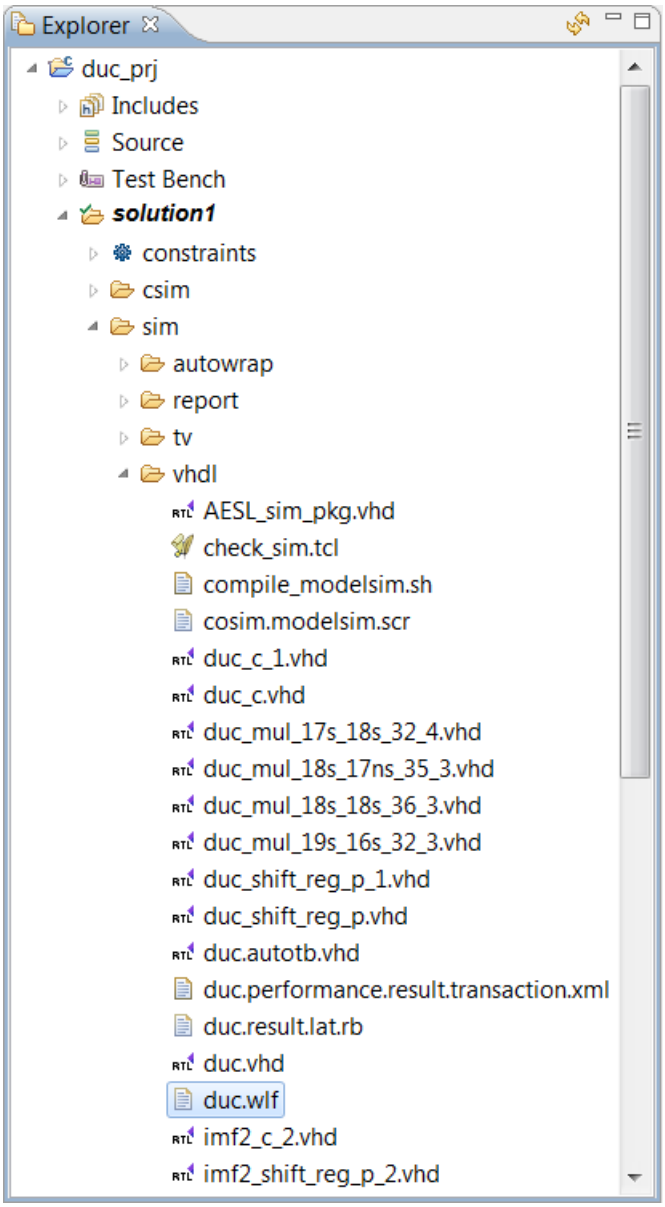


Figure 181 VHDL ModelSim Trace File

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. Use the menu File > Open.
3. Select Log Files as the file type (Figure 182).
4. Navigate to the VHDL simulation directory and select duc.wlf.
5. Press OK.

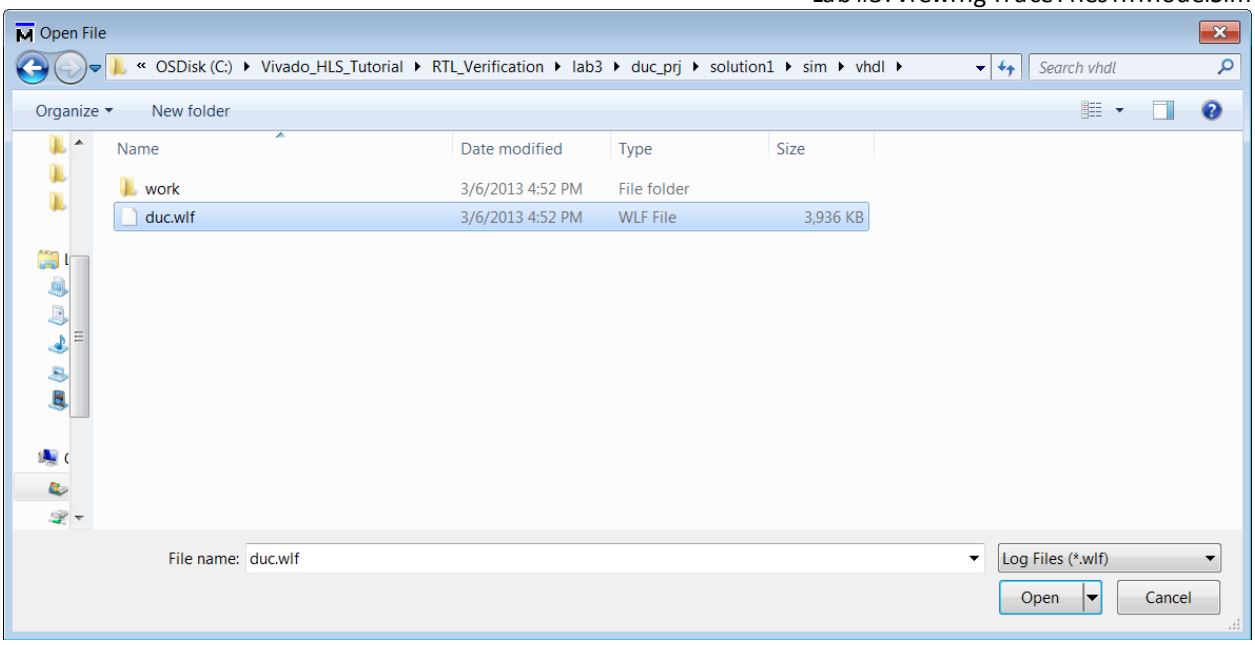


Figure 182 ModelSim Open File WLF

6. Add the signals to the trace window and adjust to see a view similar to Figure 183.

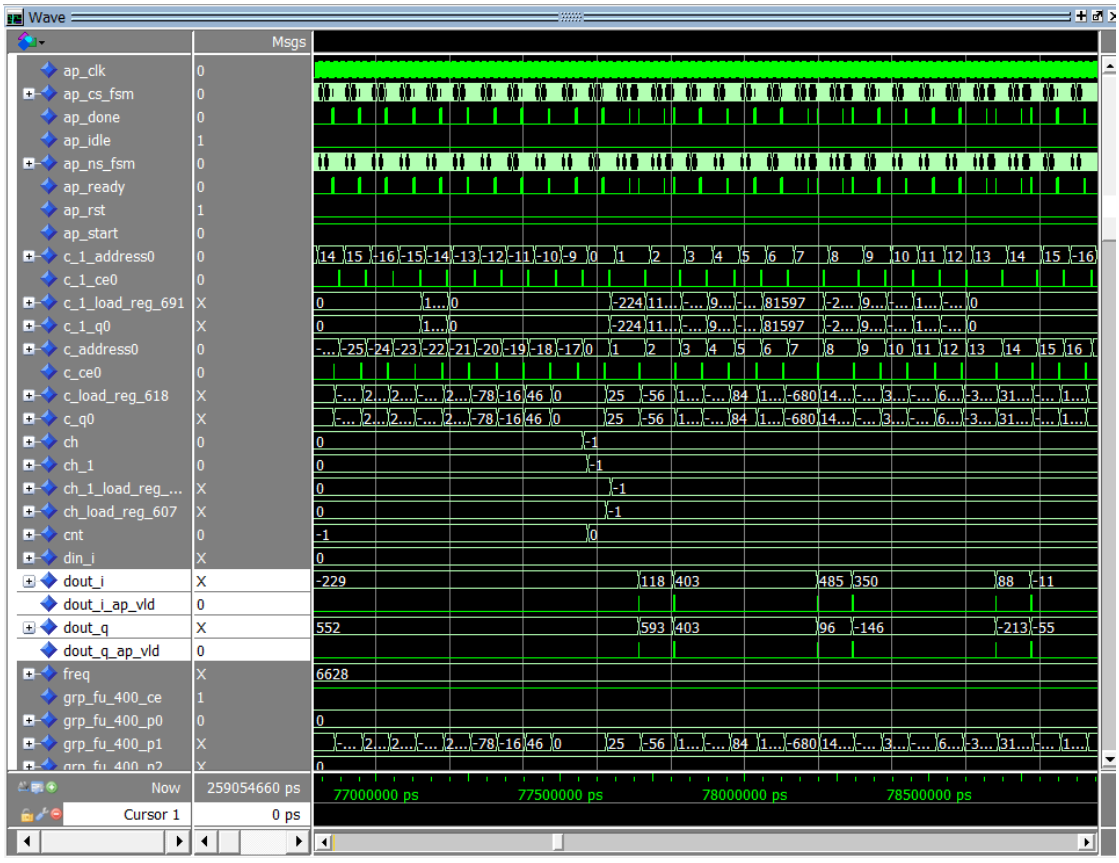


Figure 183 Viewing the Trace File in ModelSim

7. Exit and close the ModelSim RTL simulator.

This completes this tutorial on using RTL Verification.

Conclusion

In this tutorial, you learned:

- To perform RTL verification on a design synthesized from C and the importance of the test bench in this process.
- To create and open waveform trace files using the Vivado Design Suite.
- To 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

The RTL from High-Level Synthesis can be packaged and used inside IP Integrator. This tutorial demonstrates how take HLS IP and use it in IP Integrator as part of a larger design.

This tutorial consists of a single lab exercises.

Lab1

Complete the steps to generate two HLS blocks for the IP catalog and use them in a design with Xilinx IP, an FFT. The final design is validated and verified using an RTL test bench.

Tutorial Design Description

The tutorial design file can be downloaded from the Xilinx website. Refer to the information in Obtaining the Tutorial Designs.

This tutorial uses the design files in the tutorial directory **Vivado_HLS_Tutorial\Using_IP_with_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 an FFT of real data can be performed on a complex data set with all imaginary components set to zero, it may 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 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 0^{th} and 512^{th} (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 (will stall if input stalls)
- AXI4 Streaming interfaces are used to connect all blocks in IP Integrator (IPI).

Lab #1: Integrate HLS IP with a Xilinx IP Block

This lab exercise shows how generated two HLS IP blocks, combined them with a Xilinx IP FFT in IP Integrator and verify the design in the Vivado Design Suite.

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

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

Step 1: Create Vivado HLS IP Blocks

Create two HLS blocks for the Vivado IP Catalog using the provide Tcl script. The script will run HLS C-synthesis, RTL co-simulation and package the IP for the two HLS designs (hls_real2xfft and hls_xfft2real).

1. Open the Vivado HLS Command Prompt.
    a. On Windows use **Start** > **All Programs** > **Xilinx Design Tools** > **Vivado 2013.1** > **Vivado HLS** > **Vivado HLS 2013.1 Command Prompt (**Figure 184).
    b. On Linux, open a new shell.

Figure 184 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 185).
3. Type vivado_hls –f run_hls.tcl to create the HLS IP (Figure 185).

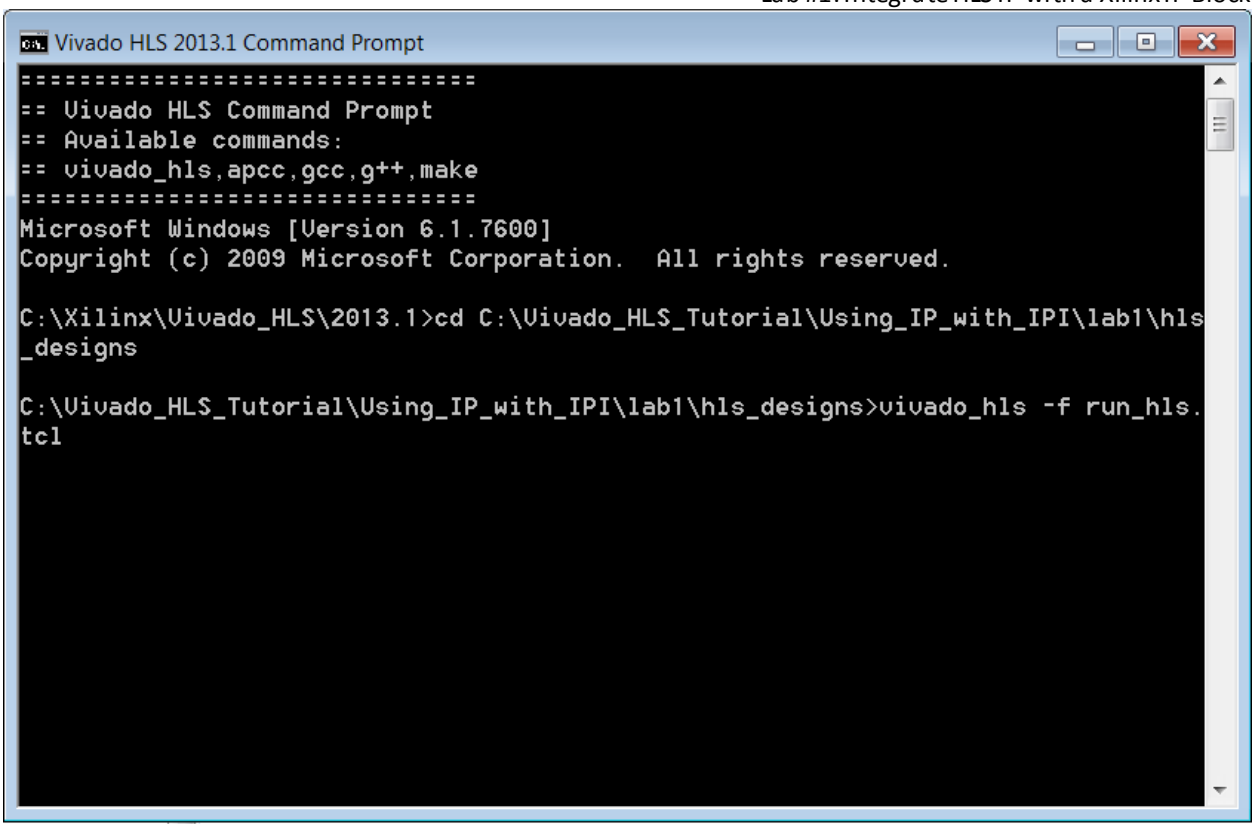


Figure 185 Create the HLS Design for IPI

When the script completes there will be two Vivado HLS project directories, fe_vhls_prj & be_vhls_prj, which contain the HLS IP, including the Vivado IP Catalog archives for use in Vivado designs.

- The "frontend" IP archive is located at fe_vhls_prj/IPXACTExport/impl/ip/
- The "frontend" IP archive is located at be_vhls_prj/IPXACTExport/impl/ip/

The remainder of this tutorial exercise shows how the Vivado HLS IP blocks can be integrated into a design in IP Integrator and verified.

Step 2: Create a Vivado Design Suite Project

1. Launch the Vivado Design Suite (not Vivado HLS):
   a. On Windows use **Start** > **All Programs** > **Xilinx Design Tools** > **Vivado 2013.1** > **Vivado 2013.1**
   b. On Linux, type vivado in the shell.
2. From the Welcome screen, click on Create New Project (186)

Figure 186 Create a Vivado Project

3. Click Next on the first page of the Create a New Vivado Project wizard.
4. Click on the ellipsis button to the right of the Project location text entry box and browse to the tutorial directory (Figure 187).

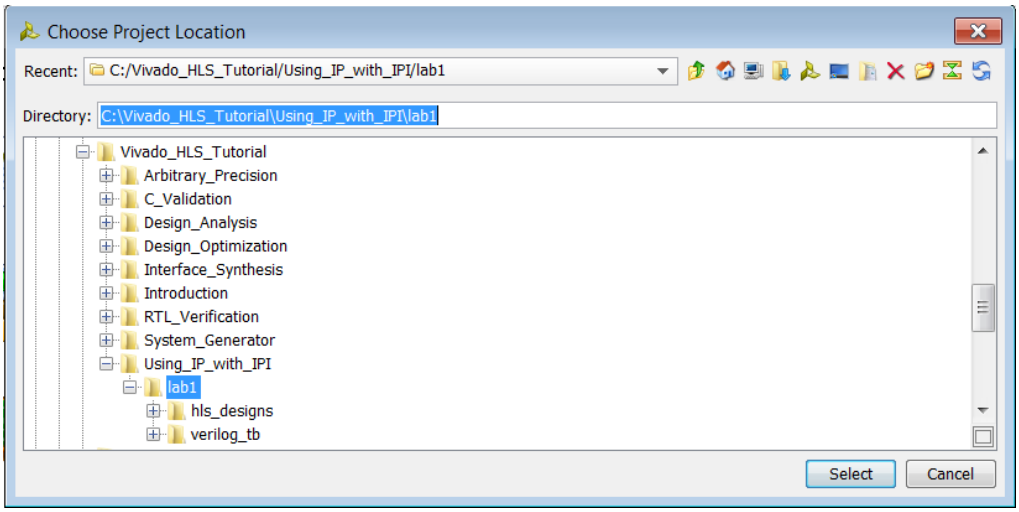


Figure 187 Path to the Vivado Design Suite Project

5. Click Next to move to the Project Type page of the wizard.
   a. Select RTL Project
   b. Do not specify sources at this time (if not the default)
   c. Click Next.
6. On the Default Part page click on Boards under Specify and select the ZYNQ-7 ZC702 Evaluation Board as shown in Figure 188.

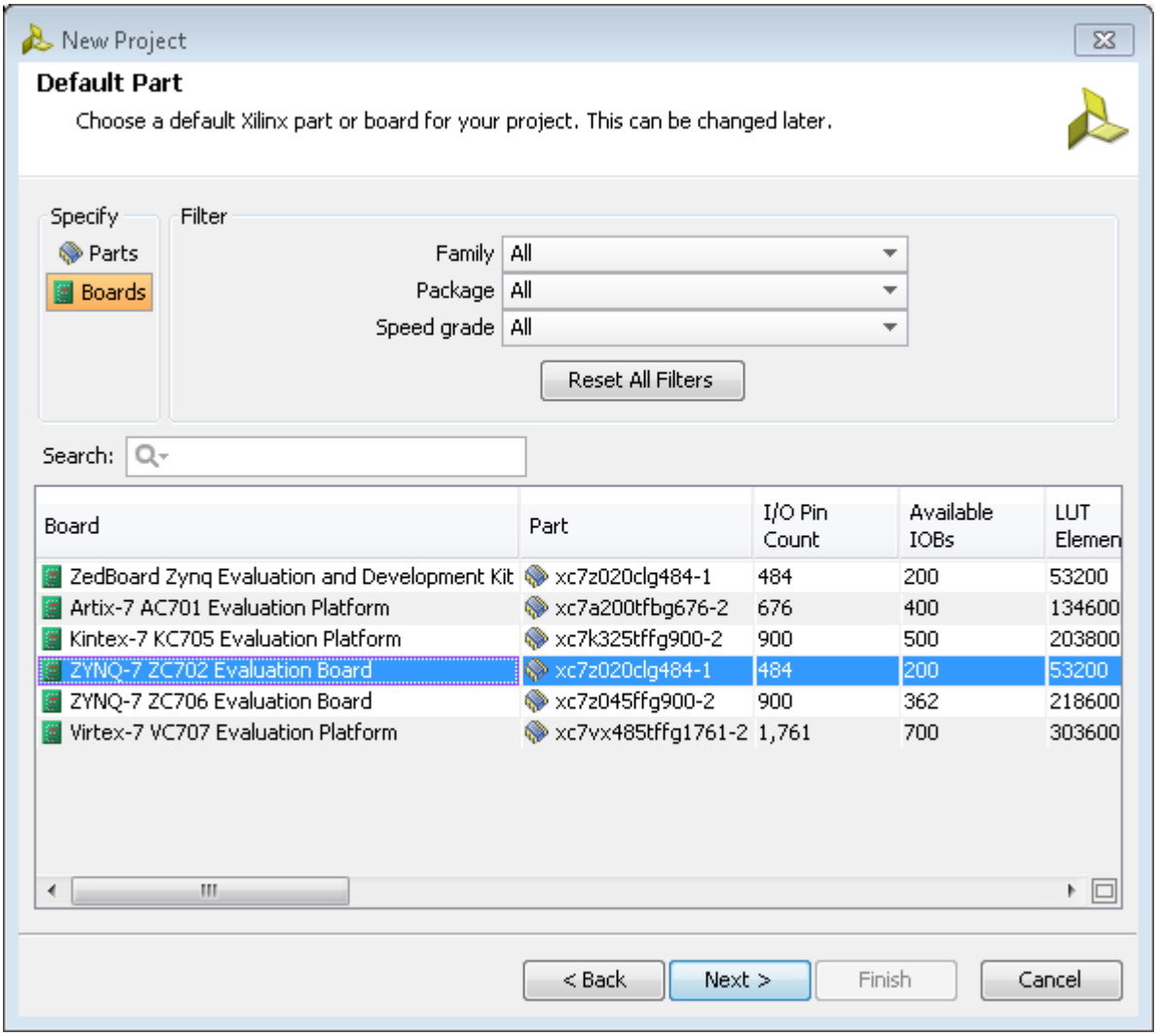


Figure 188 Vivado Project Specification

7. On the New Project Summary Page, click Finish to complete the new project setup.

The Vivado workspace will populate and look like Figure 189.

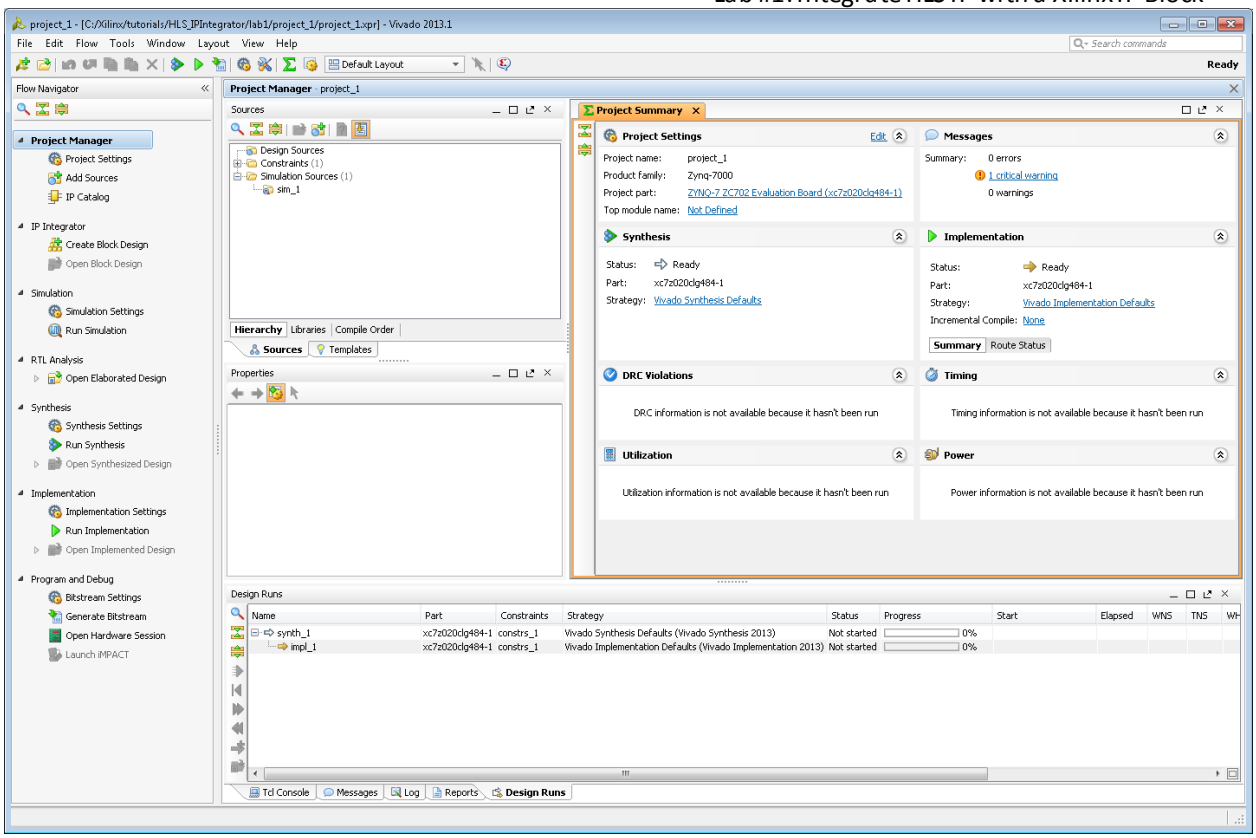


Figure 189 Vivado Project

Step 3: Add HLS IP to an IP Repository

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

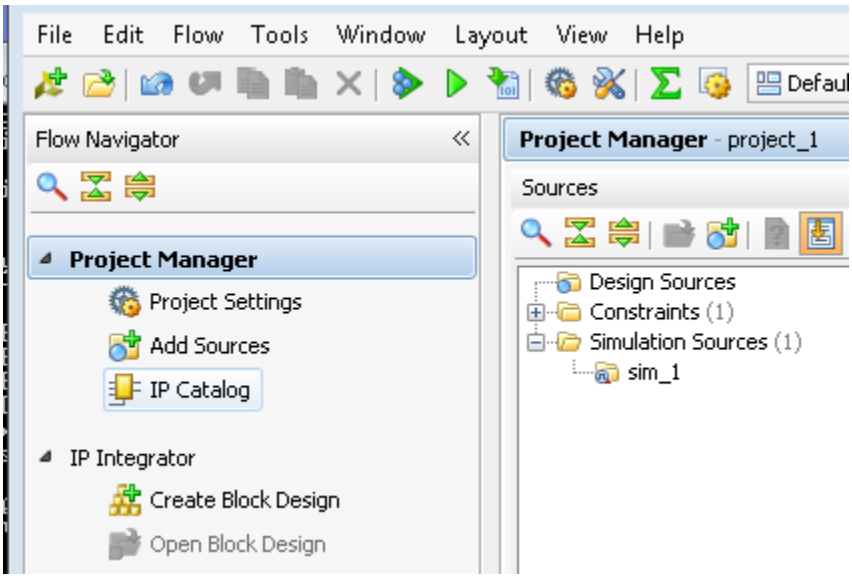


Figure 190 Open the IP Catalog

2. The IP Catalog will appear in the main pane of the workspace. Click on the IP Settings icon.

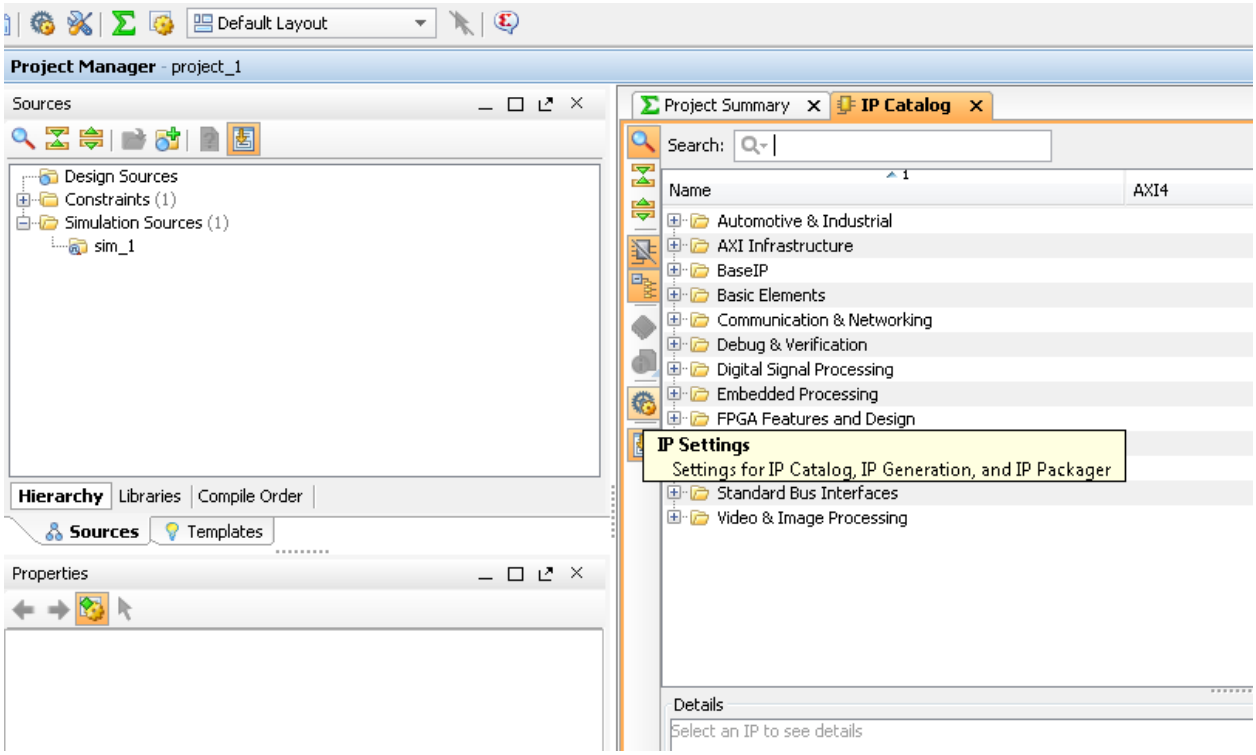


Figure 191 Open the IP Catalog Settings

3. In the IP Settings dialog, click on Add Repository…
4. In the IP Repositories dialog,
    a. Browse to the tutorial files set location
    b. click on the Create New Folder icon
    c. Enter "vivado_ip_repo" in the resulting dialog (Figure 192).
    d. Click OK
    e. Then press Select to close the IP Repository window.

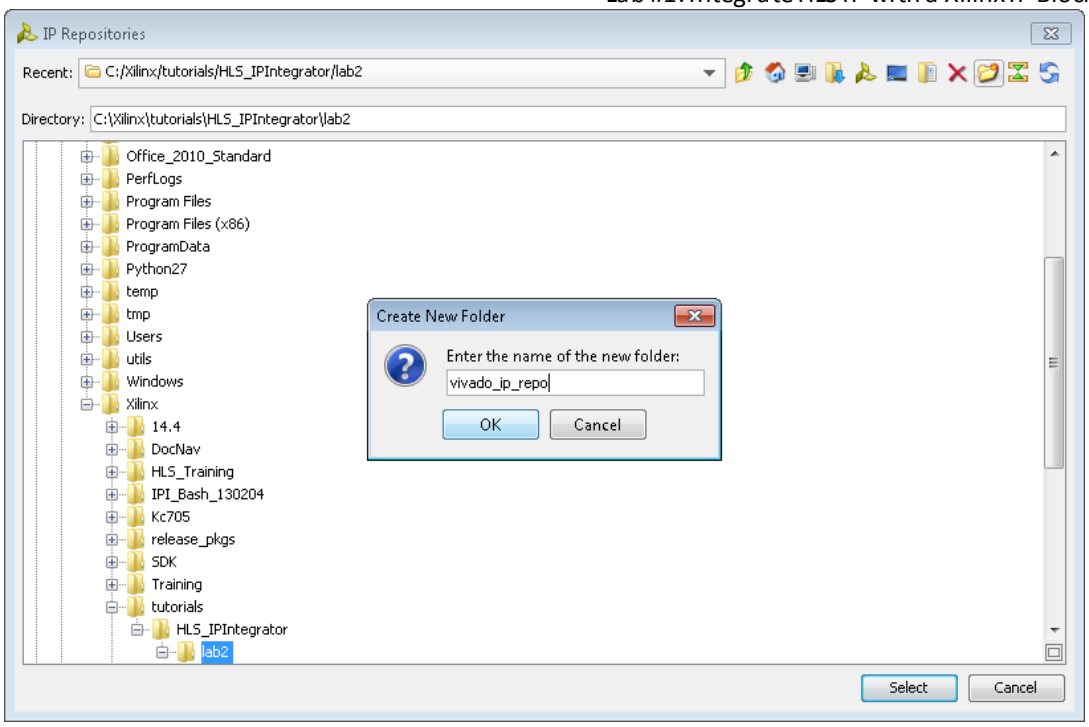


Figure 192 Create a New IP Repository

5. Back in the IP Setting dialog:
    a. Click Add IP.
    b. In the Select IP to Add to Repository dialog, browse to the location of the HLS IP lab1/hls_designs/fe_vhls_prj/IPXACTExport/impl/ip/
    c. Select the xilinx_com_hls_hls_real2xfft_1_00_a.zip file (Figure 193)
    d. Click OK

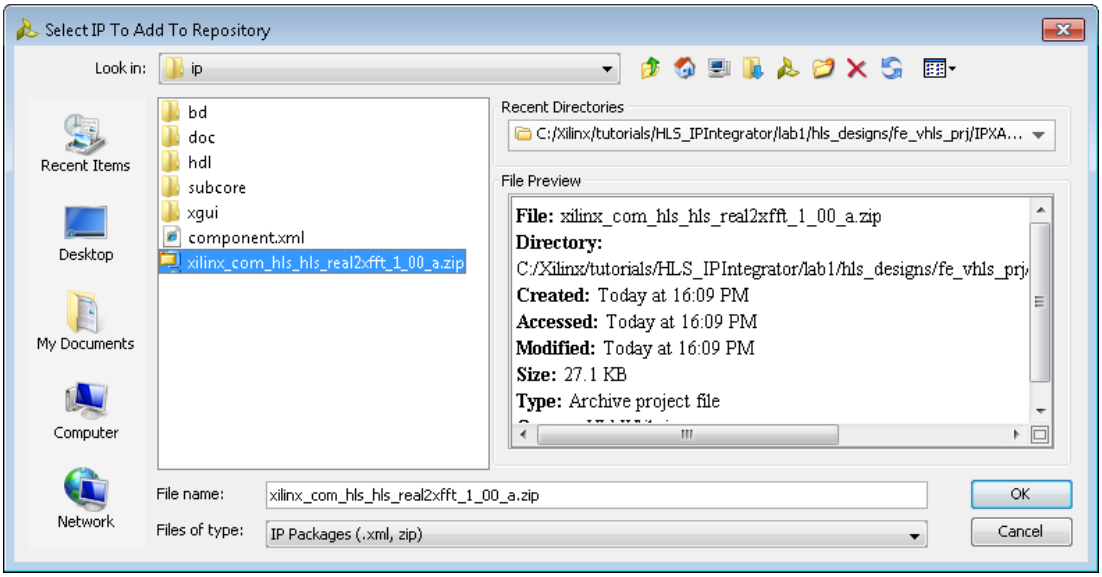


Figure 193 Add the HLS IP to the Repository

6. Follow the same procedure to add the2nd HLS IP package to the repository: xilinx_com_hls_hls_xfft2real_1_00_a.zip.
7. The new HLS IP should now show up in the IP Setting dialog (Figure 194).
8. Click OK to exit dialog.

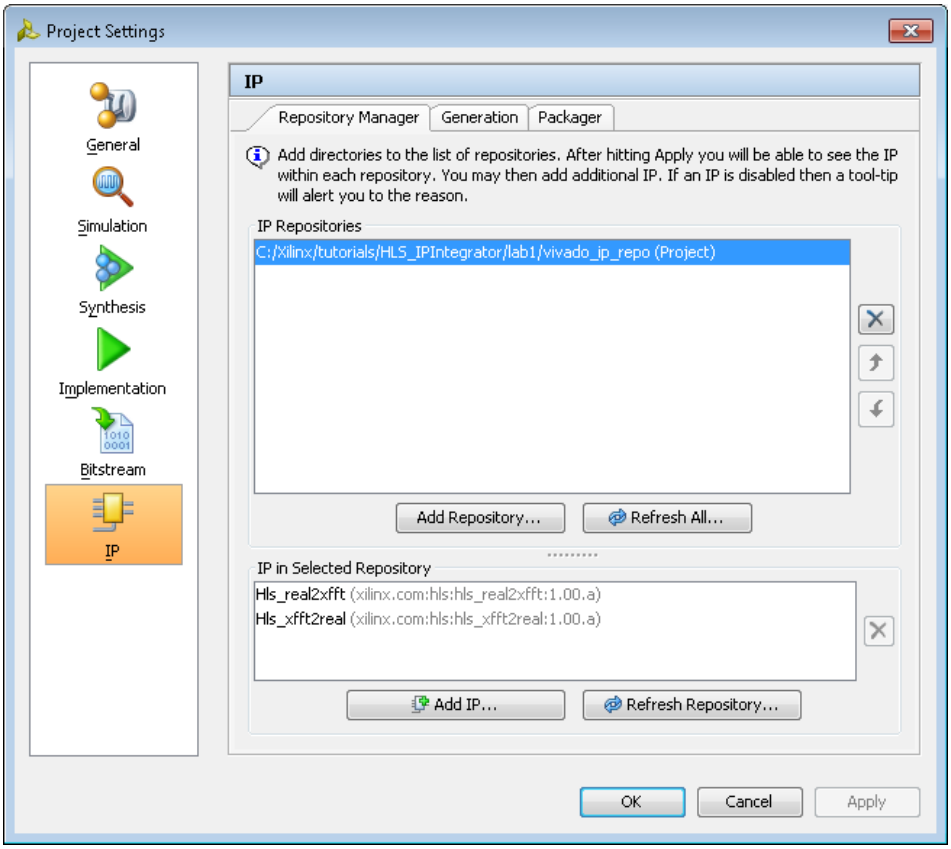


Figure 194 IP Repository with HLS IP

There should now be a Vivado HLS IP category in the IP Catalog and if expanded the HLS IP should be displayed. (Figure 195).

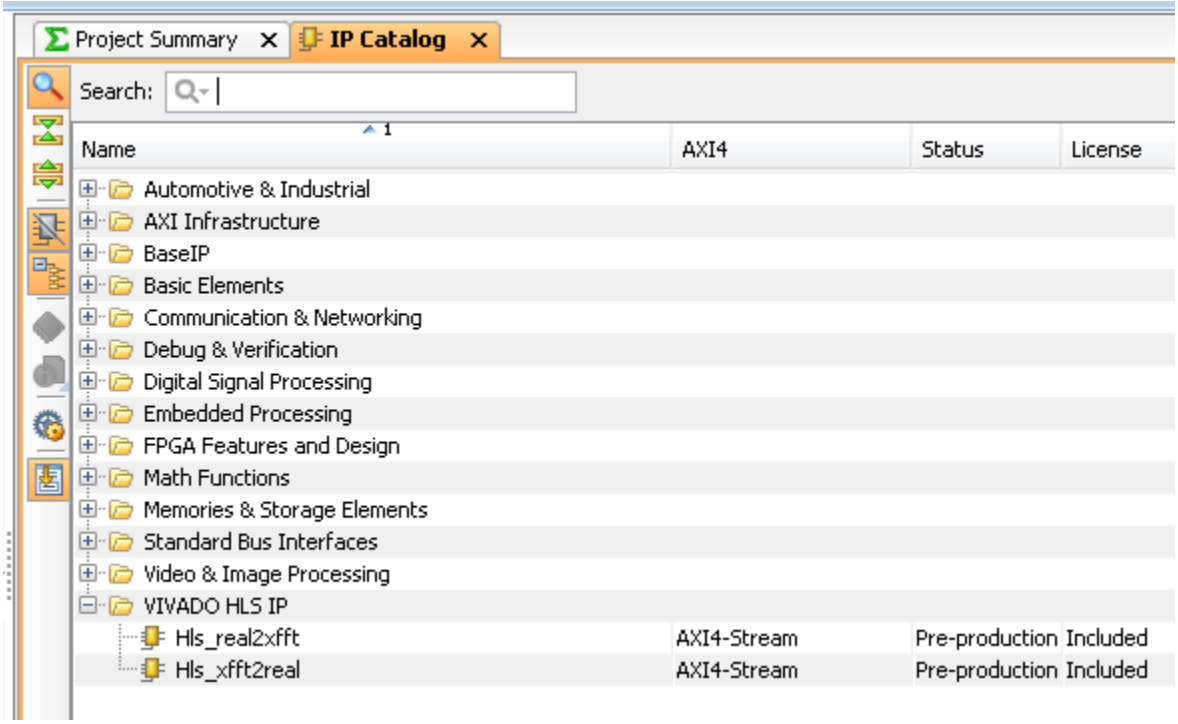


Figure 195 IP Catalog with HLS IP

Step 4: Create a Block Design for RealFFT

1. Click on Create Block Diagram under IP Integrator in the Flow Navigator.
    a. In the resulting dialog, name the design RealFFT
    b. Click OK.

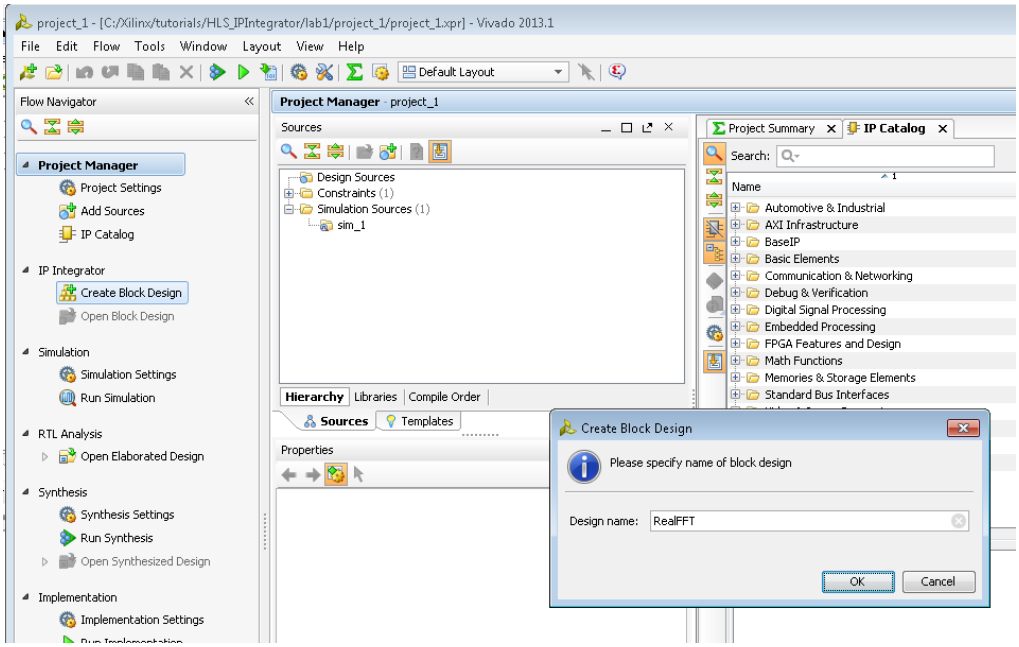


Figure 196 Create Block Diagram

The upper-right pane now has a Diagram tab. Add and customize a Xilinx FFT IP block to the design.

2. In the Diagram tab click on the Add IP link in the "get started" message (Figure 197)
   a. In the Search box type "fourier".
   b. Press the enter key.

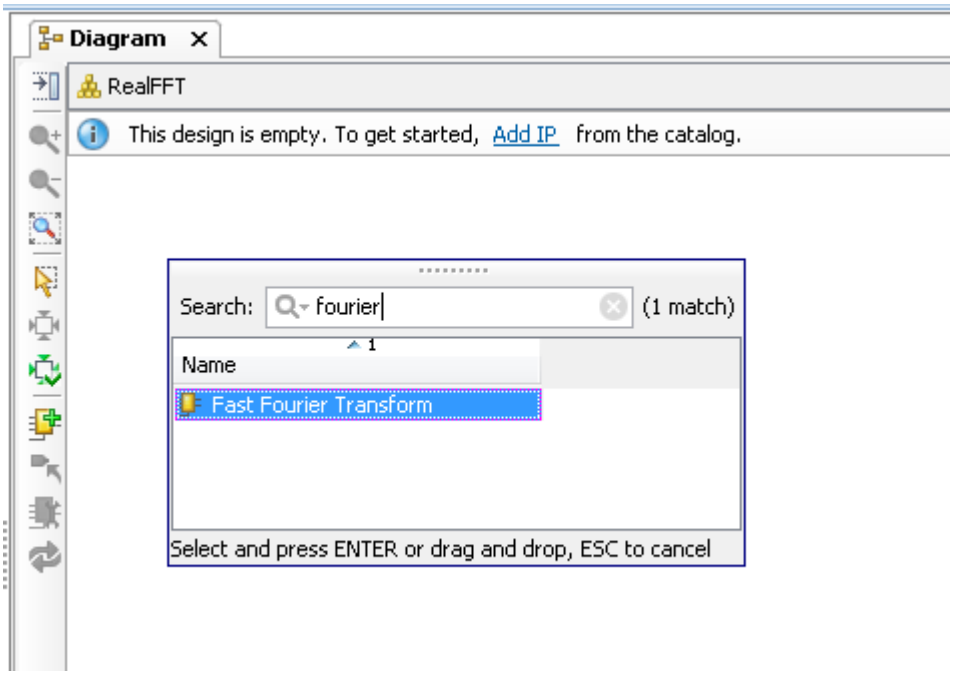


Figure 197 Add the Xilinx FFT IP

The Xilinx IP block FFT is now instantiated in the design as shown in 198.

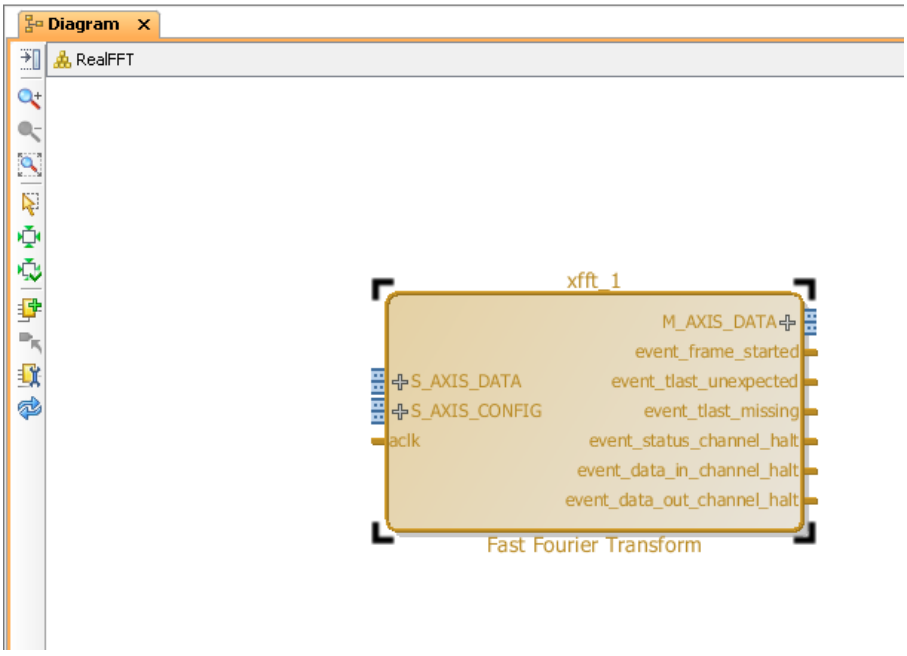


Figure 198 Xilinx FFT IP

3. Double-click on the new Fast Fourier Transform IP Symbol to open the Re-customize IP dialog
4. On the Configuration tab (Figure 199):
    a. Change the Transform Length to 512
    b. Select Pipelined, Streaming I/O Architecture Choice

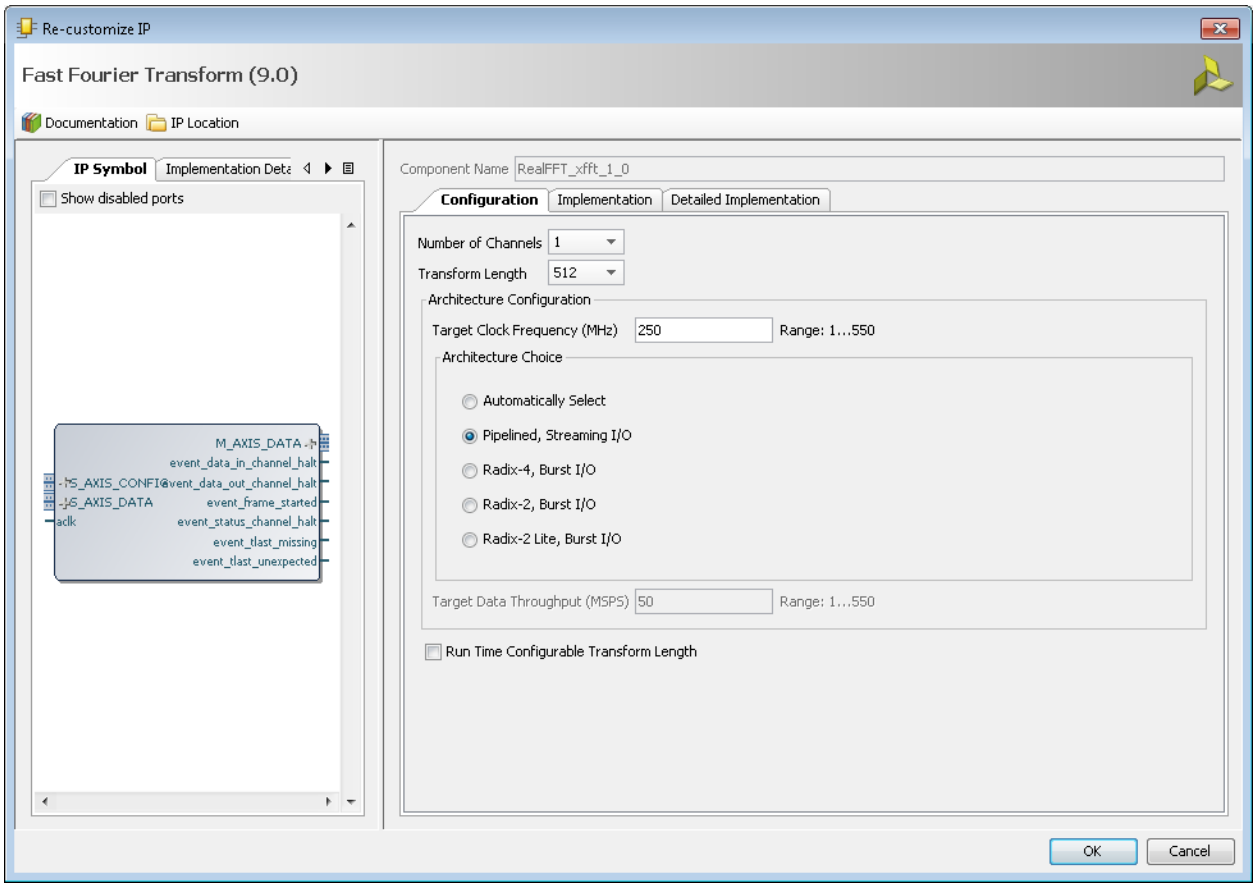


Figure 199 Xilinx FFT Configuration

5. Select the Implementation tab (Figure 200):
    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 Re-customize IP dialog

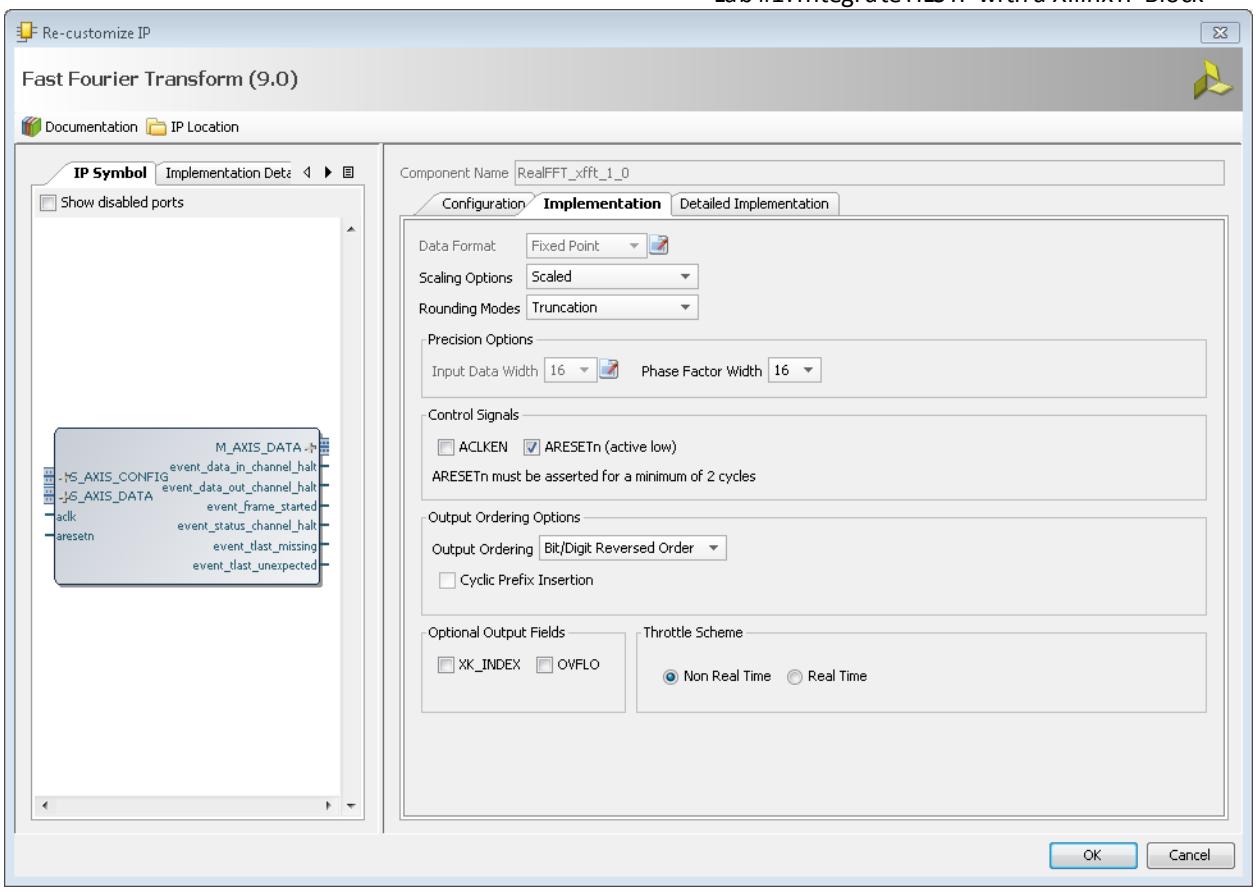


Figure 200 Xilinx FFT Implementation

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 201).

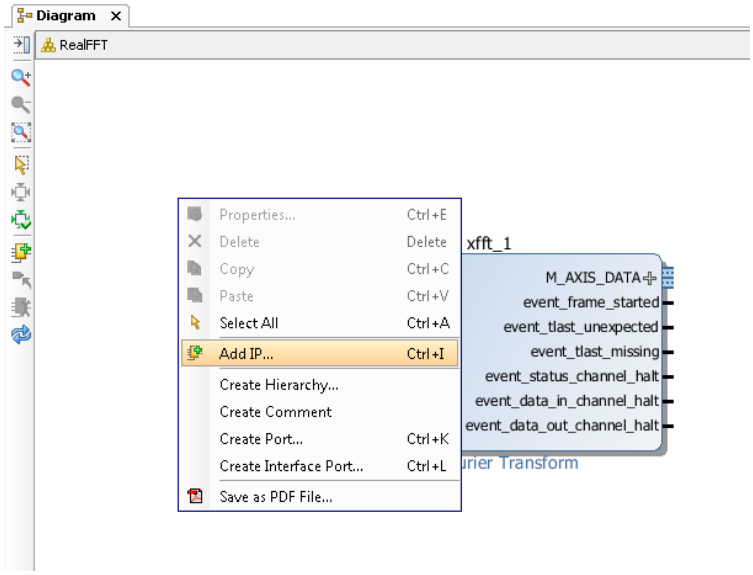


Figure 201 Add IP blocks

7. Type "hls" into the Search text entry box.
    a. Highlight both IPs (Use the control key and select both);
    b. Press the enter key.

The design block now as three IP blocks are shown in Figure 202.

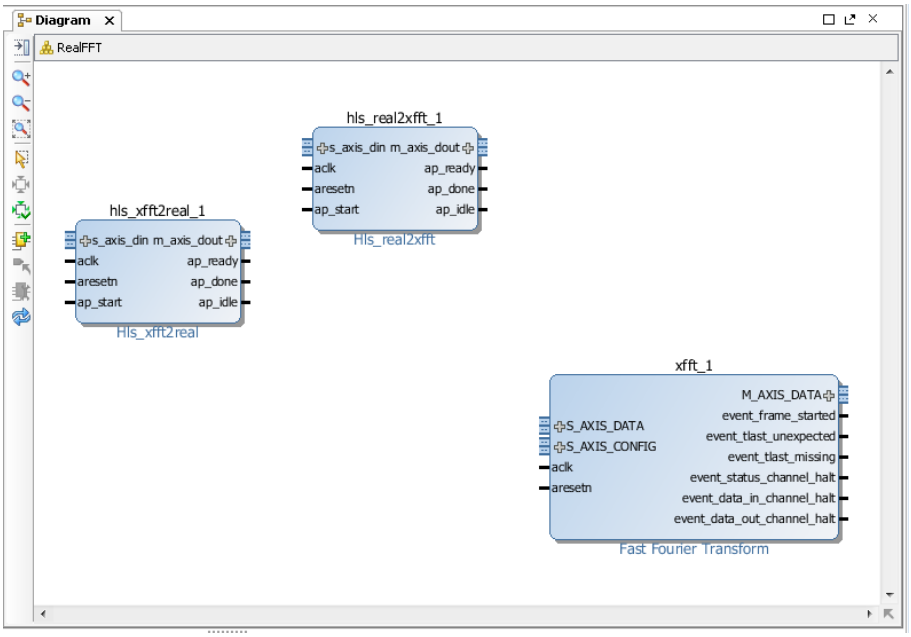


Figure 202 RealFFT IP Blocks

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

8. Hover cursor over the "m_axis_dout" interface connector of Hls_real2xftt block until pencil cursor appears.
    a. Left-click with the mouse and hold down the mouse button to start a connection.
    b. Drag the connection line to "S_AXIS_DATA" port connector of FFT block and release (when green check mark appears next to it).
9. In a similar fashion, connect the FFT's "M_AXIS_DATA" interface to the "s_axis_din" interface of the Hls_xfft2real" block.

The two connections are shown in Figure 203.

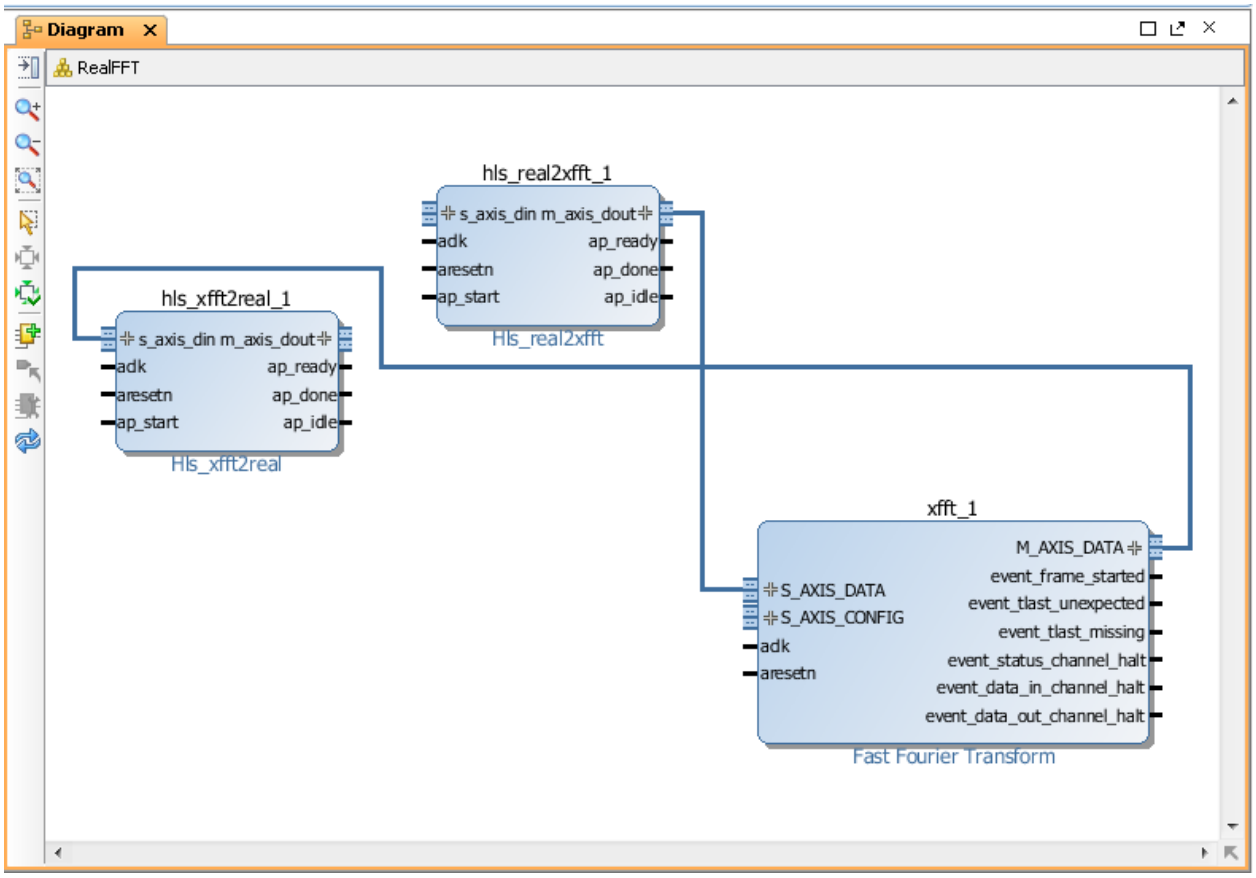


Figure 203 Connecting Ports on the IP Blocks

To create IO ports for the design, make some external connections.

10. Right-click on "s_axis_din" interface connector on Hls_real2xfft block and select Make External (Figure 204)

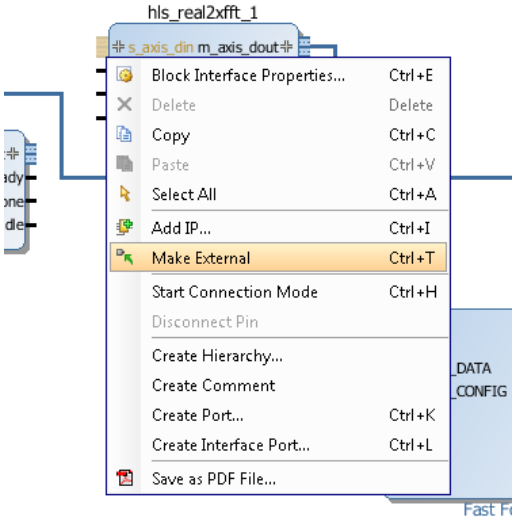


Figure 204 Make External Connections

Give the new interface port a more unique name.

a. Click on port symbol to highlight it.
b. In the External Interface Properties pane (Figure 205).
c. Double-click in the Name text entry box to highlight "s_axis_din"
d. Type in "real2xfft_din" and hit enter

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

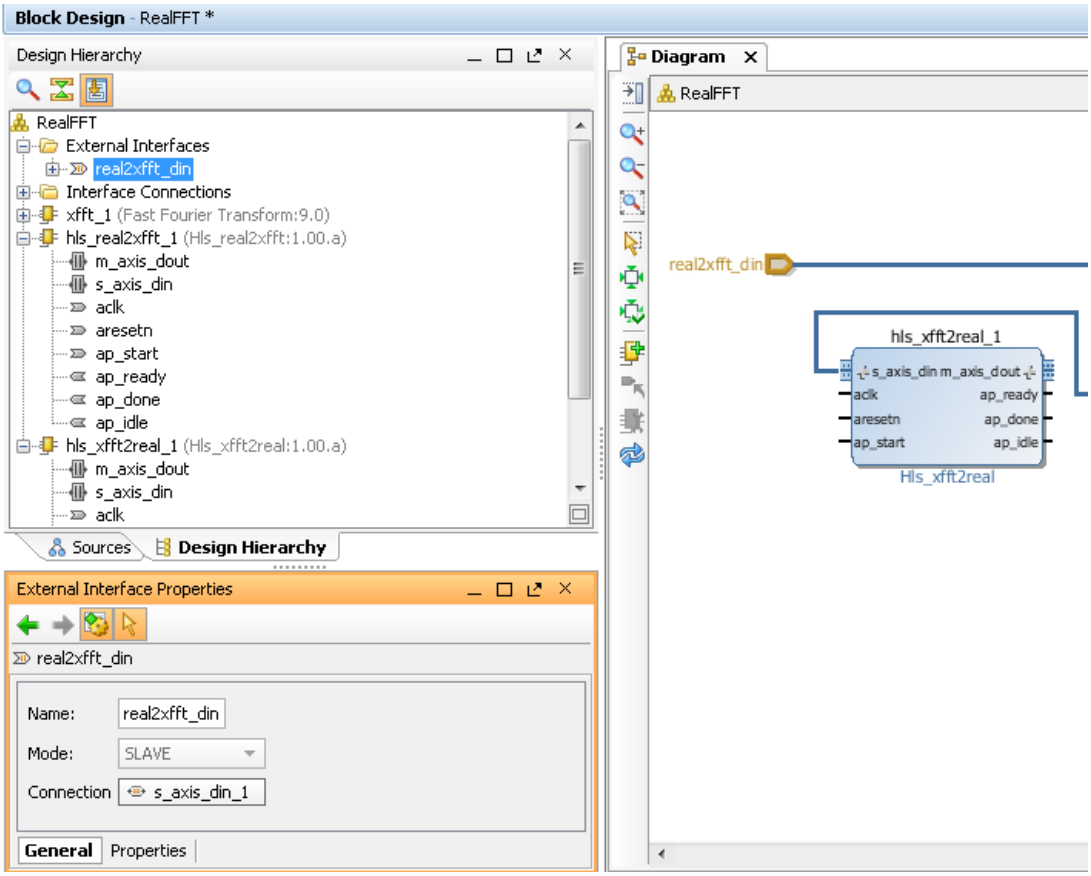


Figure 205 Port Naming

11. In a similar manner to the previous step:
    a. Make the "m_axis_dout" interface of Hls_xfft2real block external and rename it to "xfft2real_dout"
    b. Right-click on aclk connector of Hls_real2xfft block and select Make External.
    c. Right-click on aresetn connector of Hls_real2xfft block and select Make External.
12. Tie ap_start ports of HLS blocks high
    a. Right-click on canvas, select Add IP.
    b. Type "const" into Search text entry box.
    c. Select Constant IP.

d. Press the enter key.

e. Double-click on Constant IP Symbol (Figure 206) and verify that the settings for Const Width and Const Val are both '1' then click OK to close Re-customize IP dialog.

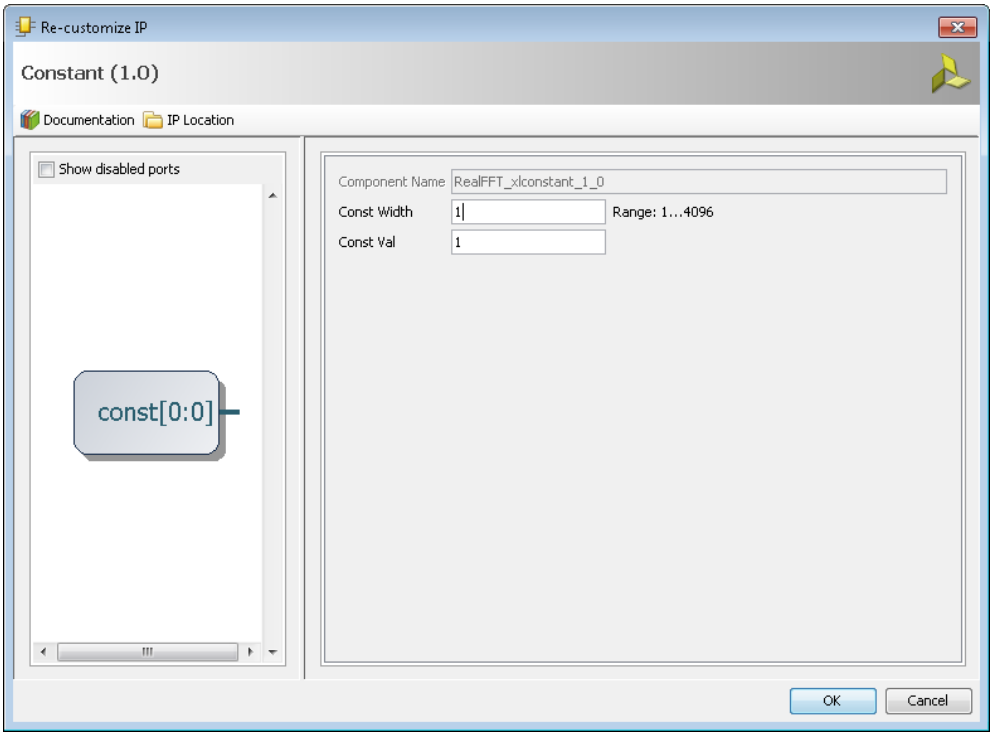


Figure 206 Constant IP Properties

f. Connect ap_start of both HLS blocks to the Constant block (Figure 207)

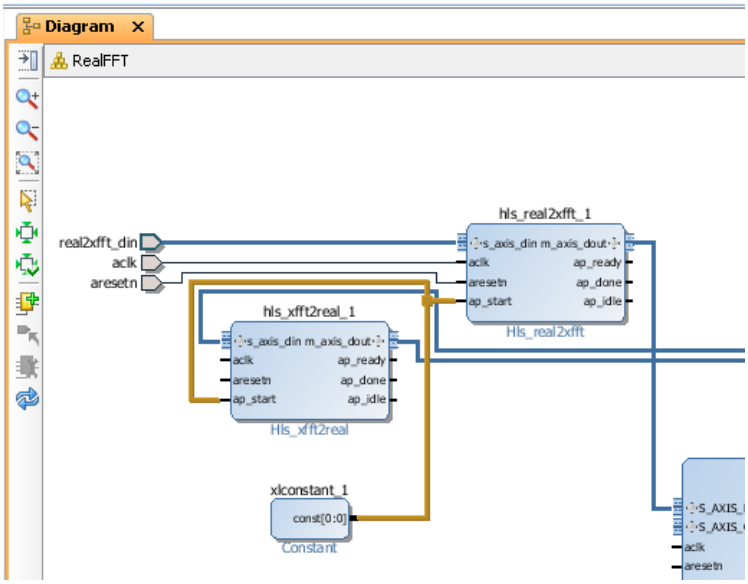


Figure 207 Connect AP_START to Constant 1

13. Make remaining connections
    a. Click and drag from the aclk connector of FFT and Hls_xfft2real blocks to the aclk external port (or aclk connector on Hls_real2xfft block or anywhere on "wire" connecting them)
    b. Connect aresetn of FFT and Hls_xfft2real blocks to aresetn network
    c. The XFFT configuration interface will be left unconnected, as this design always operates in the default mode of the core.
14. Click on the Regenerate icon to clean up and reorganize the Block Design

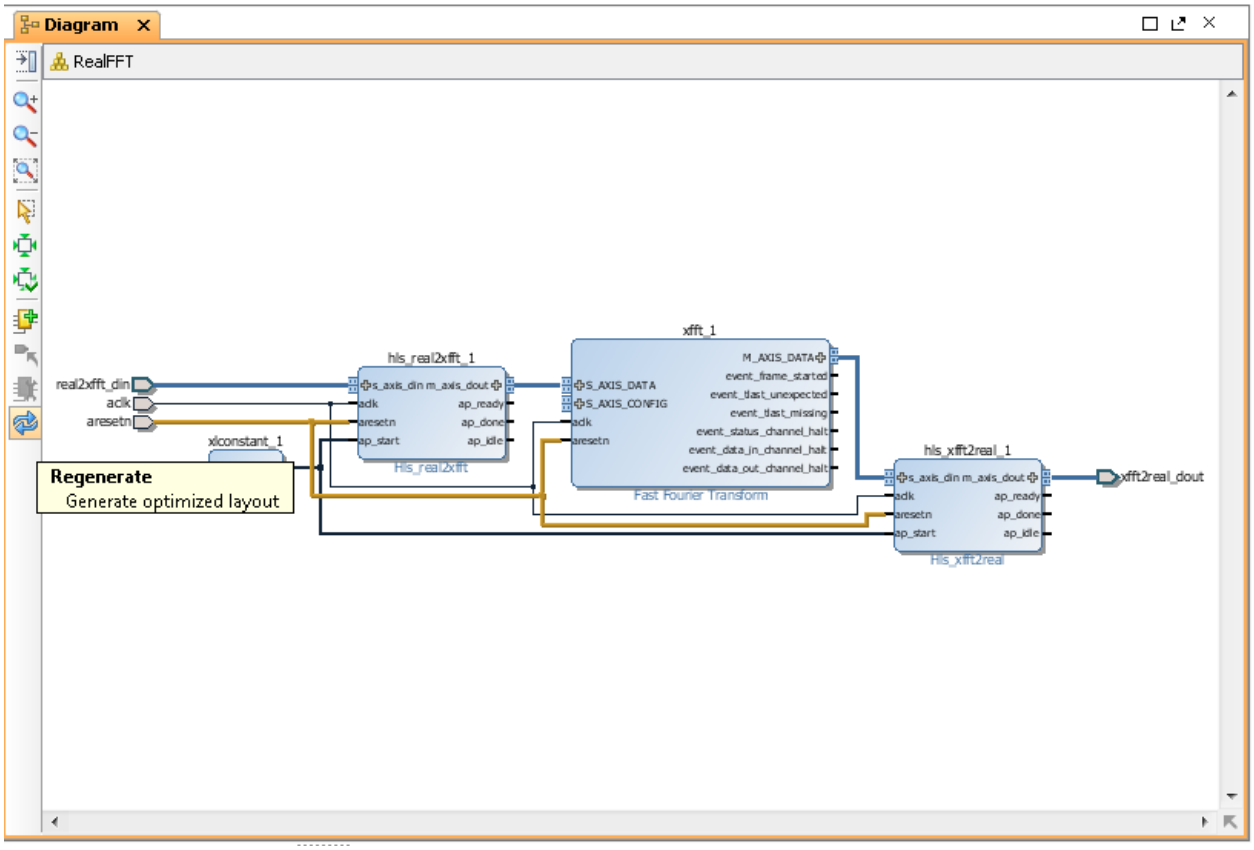


Figure 208 Re-generated Design Diagram

15. Validate the Block Design by clicking on the Validate Design icon on the toolbar.

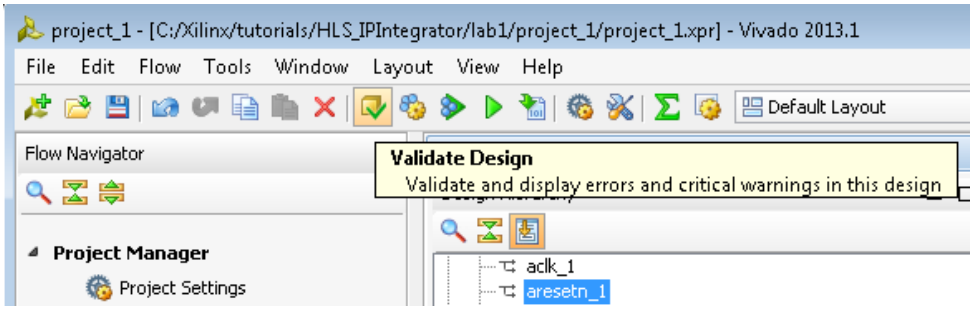


Figure 209 Design Validation

This may generate some Critical Messages regarding interface ports not being associated with a clock.

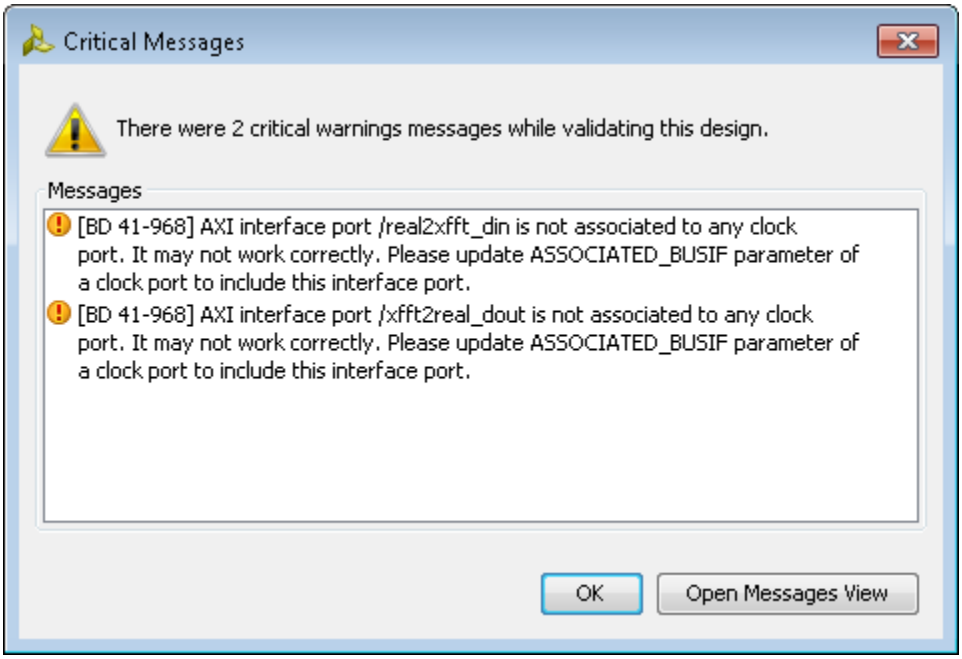


Figure 210 Clock Warnings

To address such warnings before proceeding return to the design diagram.

16. In the design Diagram tab,
    a. Select the aclk external interface port (by clicking on it).
    b. Select the Properties tab in the External Port Properties pane (Figure 211).
    c. Expand the CONFIG property
    d. In the ASSOCIATED_BUSIF text entry box add real2xfft_din:xfft2real_dout (***Note***: a colon separates each bus interface).

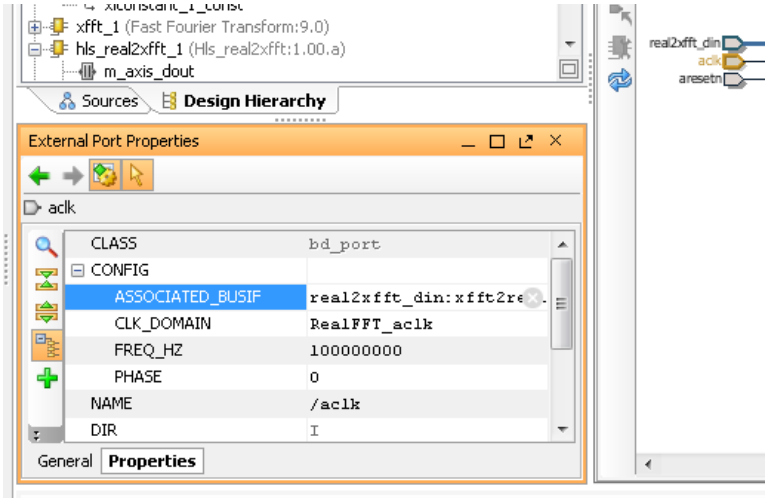


Figure 211 Clock Warnings

a. Press the enter key.
b. Re-run validation to see it complete.

17. Save the Block Design by using the menu File > Save Block Design (or Ctrl-S).
18. Close the Block Design.

The next step is to Generate Output Products.

a. In the Sources tab of Project Manager pane (Figure 212), right-click on RealFFT.bd and select Generate Output Products.
b. Click OK in the resulting dialog to initiate the generation of all output products.

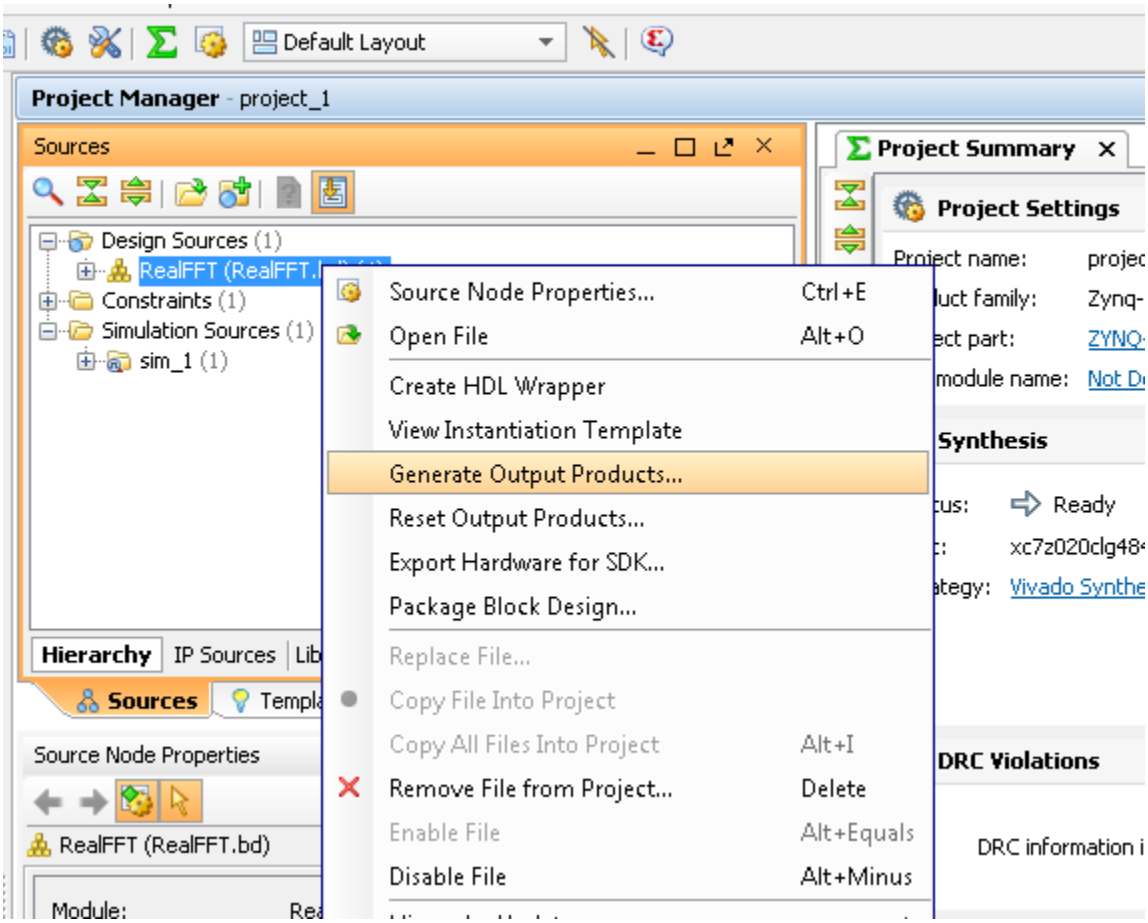


Figure 212 Generating Output Products

19. Create an HDL Wrapper
    a. In Sources tab of Project Manager pane, this is the same procedure and menu as the previous step, right-click on RealFFT.bd and select Create HDL Wrapper.
    b. Click OK to clear the resulting notification window

Step 5: Verify the Design

The next step in creating the final design is to verify design with HDL test bench provided in the lab exercise: realfft_rtl_tb.v.

1. Right-click on Simulation Sources in Sources tab of Project Manager pane (Figure 213).
2. Select Add Sources.

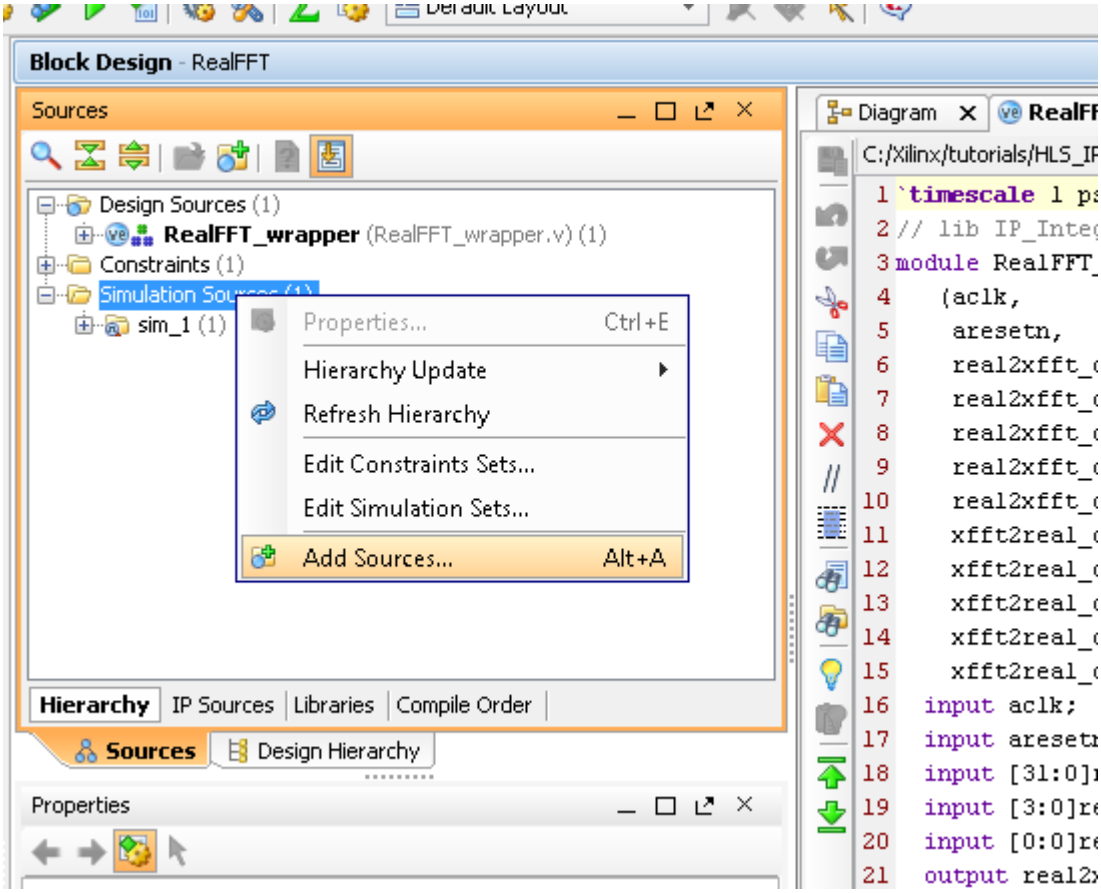


Figure 213 Adding Simulation Sources

3. Select Add or Create Simulation Sources in the Add Sources dialog.
4. Click next.
5. In the Add Sources dialog box, click on the Add Files button highlighted in Figure 215.

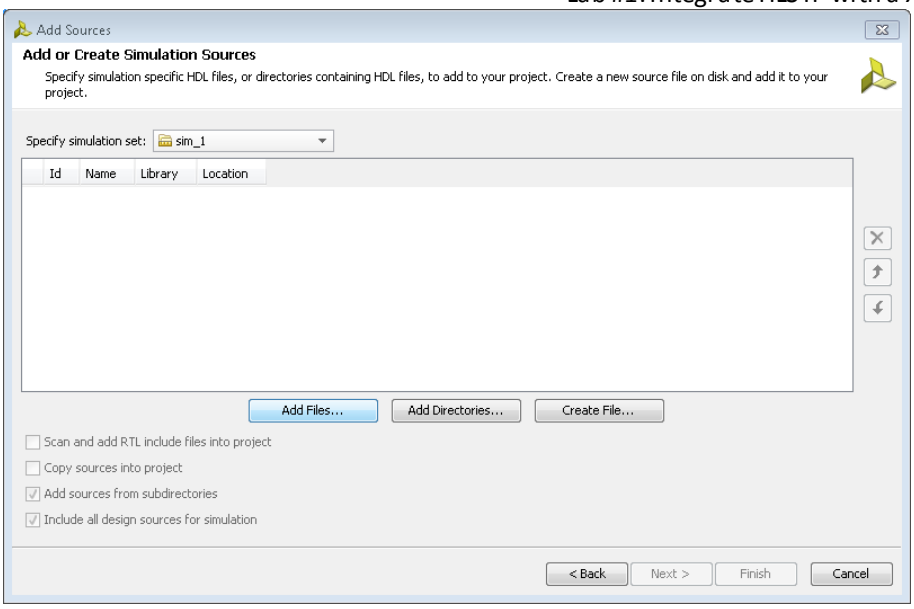


Figure 214 Add Source Dialog Window

6. Browse to the realfft_rtl_tb.v file in the Using_IP_with_IPI\lab1\verilog_tb tutorial directory.
7. Select it and click OK.
8. Select to copy sources into the project (Figure 215).

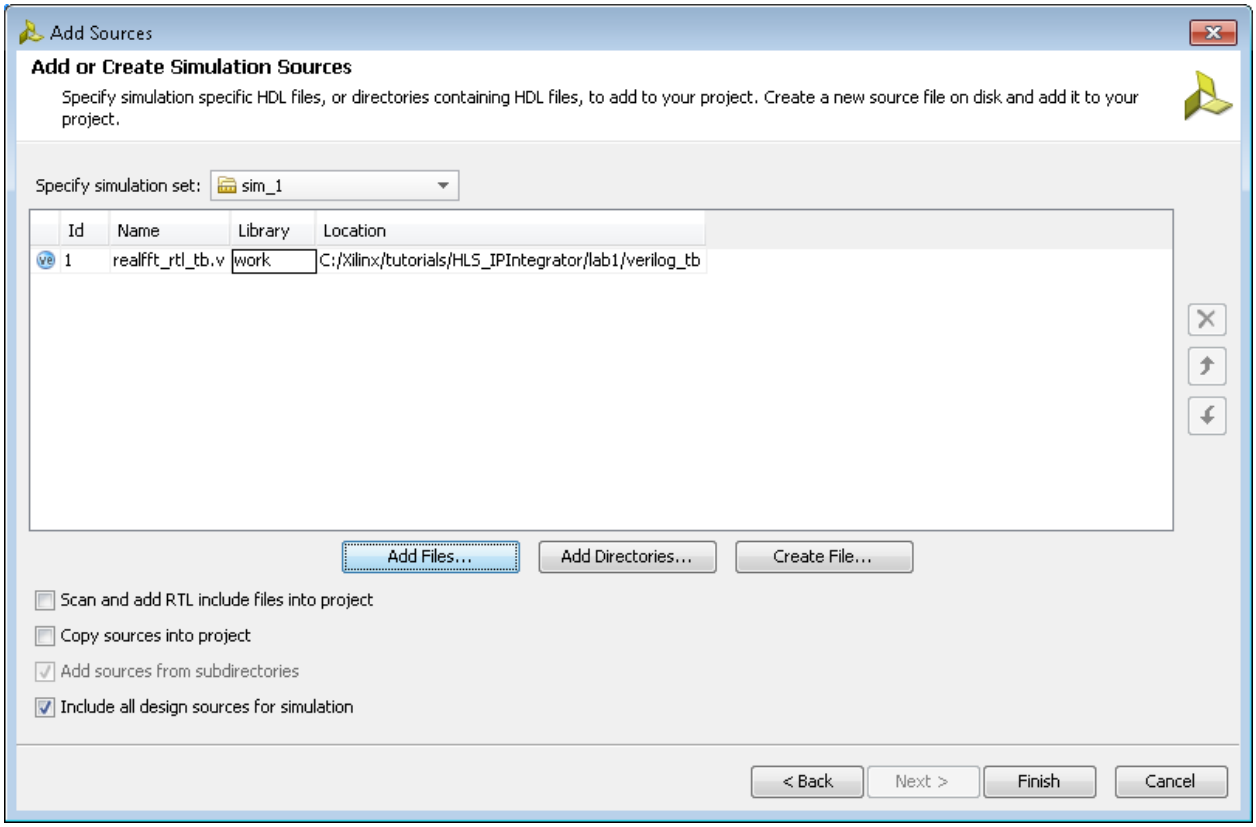


Figure 215 Copy Design Sources

Note: When the design sources are copied into the project, edits to the file(s) will not automatically be propagated back to the original source file.

9. Click Finish.
10. Click on Run Simulation in the Flow Navigator (Figure 216).

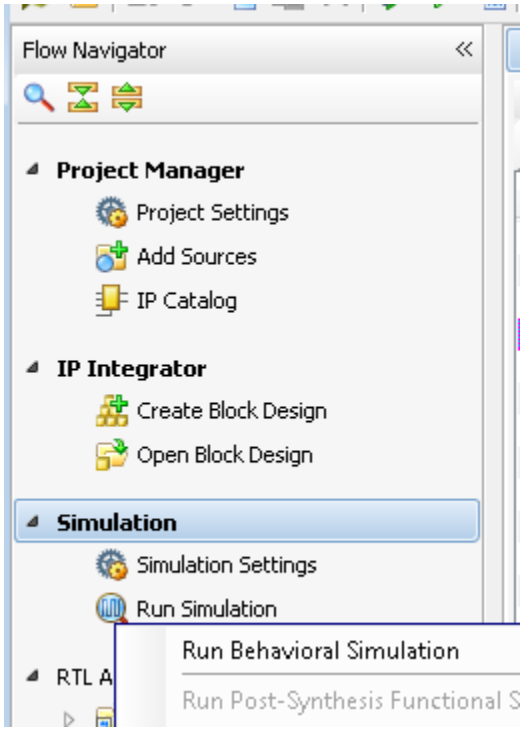


Figure 216 Execute Simulation

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

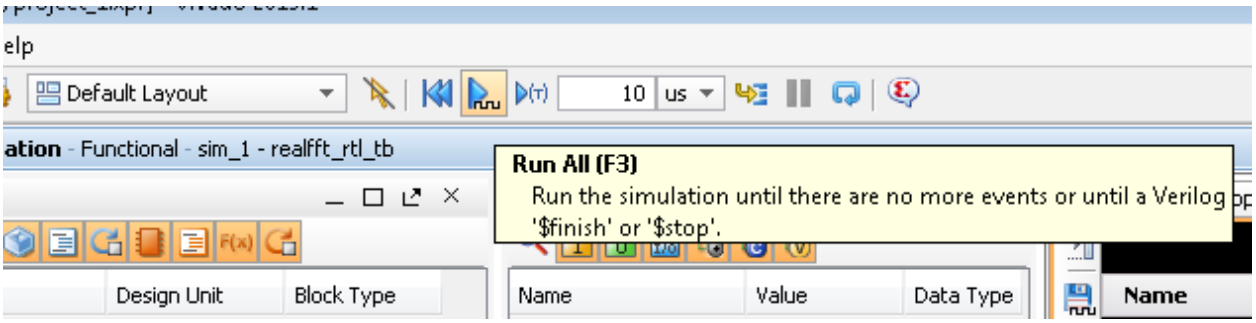


Figure 217 Run The Simulation to Conclusion

This completes this tutorial on using Vivado High-Level Synthesis blocks as IP inside IP Integrator.

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.
- Finally, how to verify the design in IPI.

Using HLS IP in a Zynq Processor Design

Overview

A common use of High-Level Synthesis design is to create an accelerator for a CPU – to move code which executes on the CPU into the FPGA programmable logic to improve performance. This tutorial shows how design created with High-Level Synthesis can be incorporated into a Zynq device

This tutorial consists of a single lab exercise.

Lab1

A simple HLS design is created and configured to work with the CPU on a Zynq device. The HSL 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.

Tutorial Design Description

The tutorial design file can be downloaded from the Xilinx website. Refer to the information in Obtaining the Tutorial Designs.

This tutorial uses the design files in the tutorial directory **Vivado_HLS_Tutorial\Using_IP_with_Zynq**

The sample design is a simple multiple accumulate block. The focus of this tutorial exercise is the methodology, connections and integration of the software drivers: not the logic in the design itself.

Lab #1: Implement Vivado HLS IP on a Zynq Device

This lab exercise will integrate both the High-Level Synthesis IP and the software drivers created by HLS to control the IP in a design implemented on a Zynq device.

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

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

directory.

Step 1: Create a Vivado HLS IP Block

Create two HLS blocks for the Vivado IP Catalog using the provide Tcl script. The script will run HLS C-synthesis, RTL co-simulation and package the IP for the two HLS designs (hls_real2xfft and hls_xfft2real).

1. Open the Vivado HLS Command Prompt.
    a. On Windows use **Start** > **All Programs** > **Xilinx Design Tools** > **Vivado 2013.1** > **Vivado HLS** > **Vivado HLS 2013.1 Command Prompt (**Figure 218).
    b. On Linux, open a new shell.

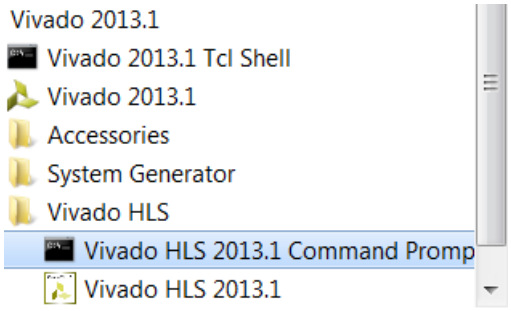


Figure 218 Vivado HLS Command Prompt

2. Using the command prompt window, change the directory to Vivado_HLS_Tutorial\Using_IP_with_Zynq\lab1\hls_macc (Figure 219).
3. Type vivado_hls –f run_hls.tcl to create the HLS IP (Figure 219).

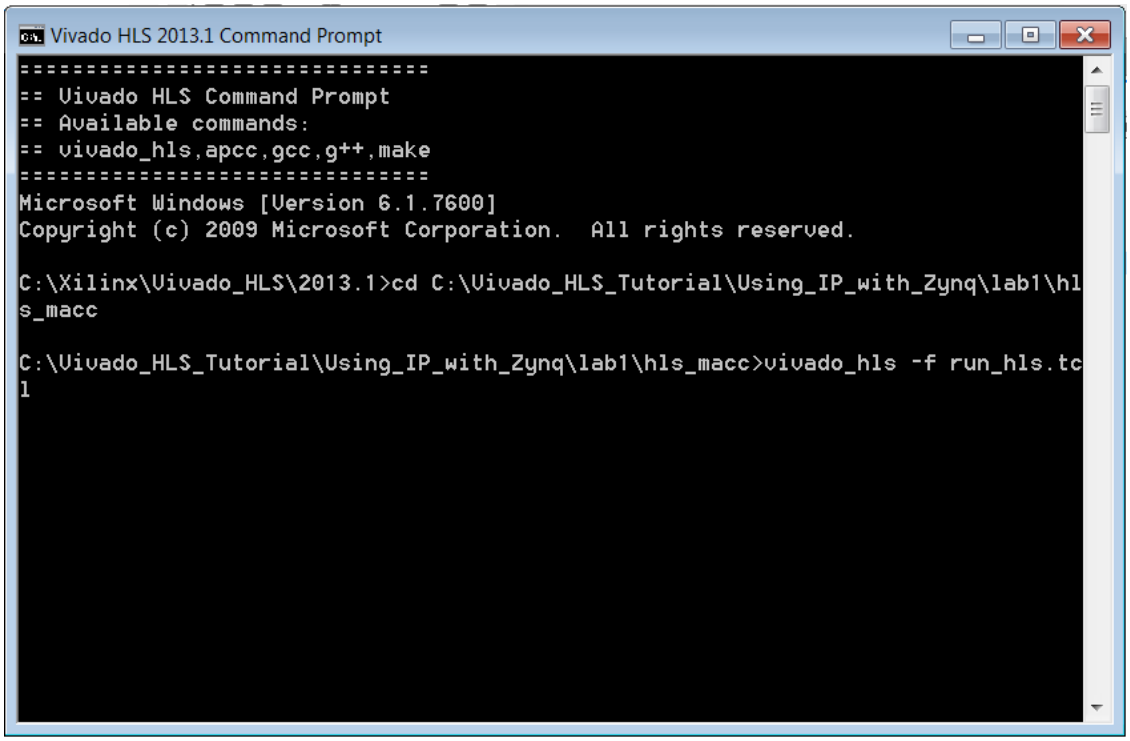


Figure 219 Create the HLS Design

When the script completes there will be a Vivado HLS project directories vhls_prj, which contains the HLS IP, including the Vivado IP Catalog archive for use in Vivado designs.

The remainder of this tutorial exercise shows how the Vivado HLS IP blocks can be integrated into a Zynq design using IP Integrator.

Step 2: Create a Vivado Zynq Project

1. Launch the Vivado Design Suite (not Vivado HLS):
    a. On Windows use **Start** > **All Programs** > **Xilinx Design Tools** > **Vivado 2013.1** > **Vivado 2013.1**
    b. On Linux, type vivado in the shell.
2. From the Welcome screen, click on Create New Project (Figure 220)

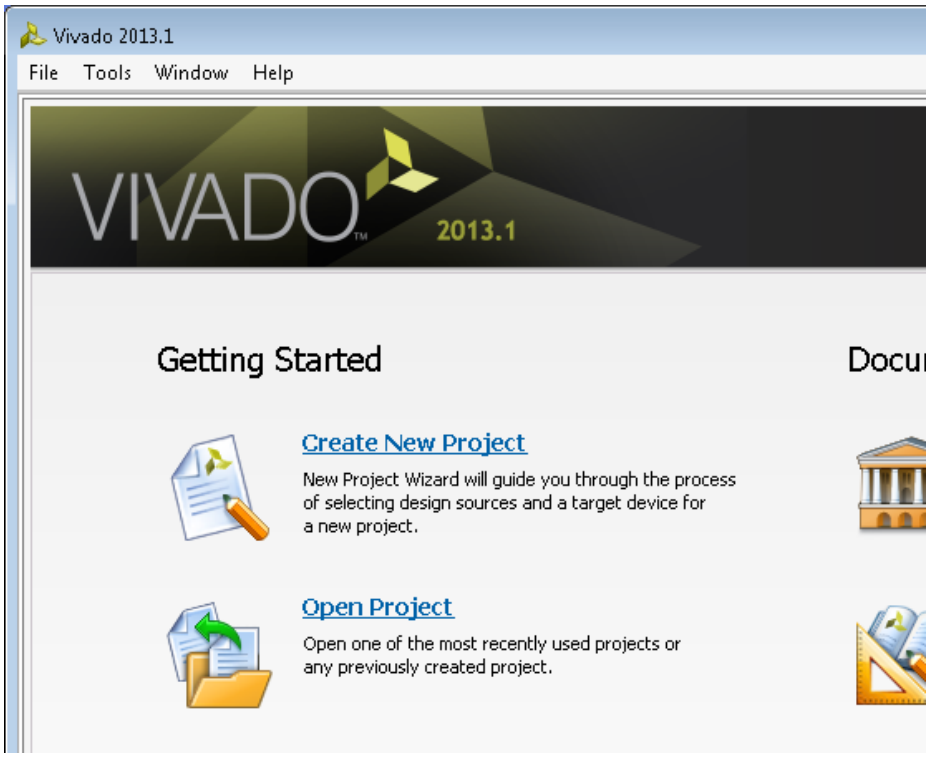


Figure 220 Vivado Welcome Screen

3. In the New Project wizard,
    a. Click Next;
    b. In Project Location text entry box, browse to location of tutorial file directory and click Next (Figure 221).
    c. On the Project Type page, select "Do not specify sources at this time" (if not default)

d. Click Next

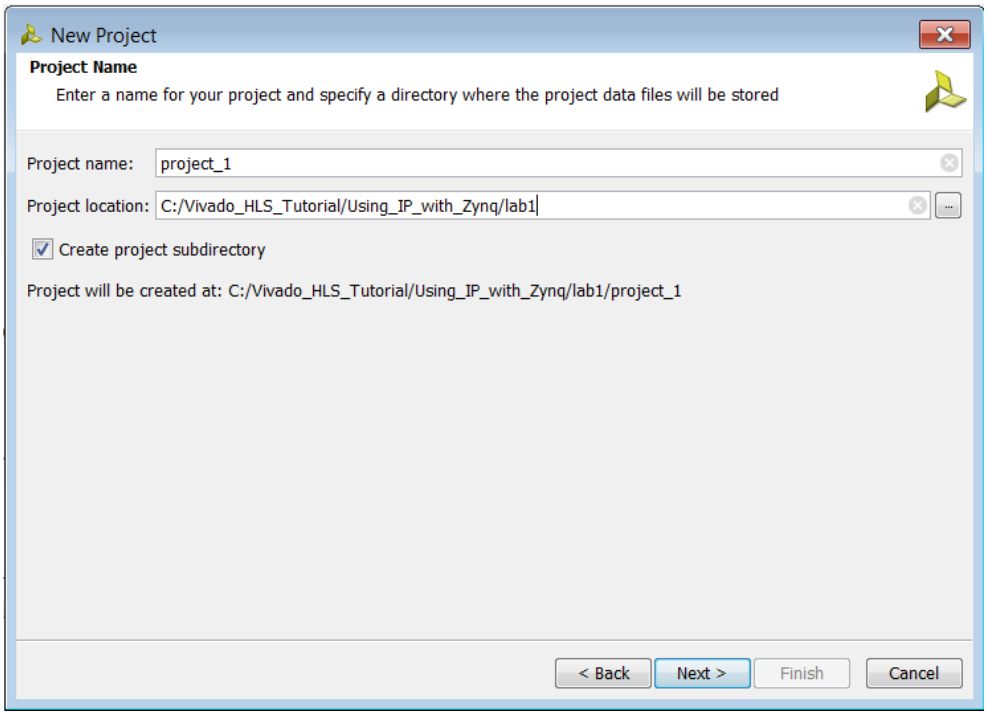


Figure 221 Specify the Vivado Project Directory

4. On the Default Part page,
   a. Click on Boards
   b. Select the ZYNQ-7 ZC702 Evaluation Board (Figure 222).

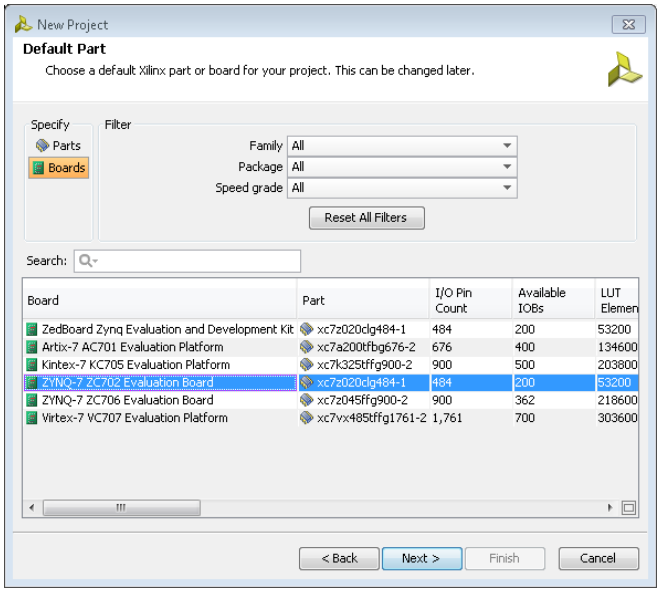


Figure 222 Specify the Vivado Project Details

c. Click next

d. Click Finish on the New Project Summary Page

The project workspace will open as shown in Figure 223.

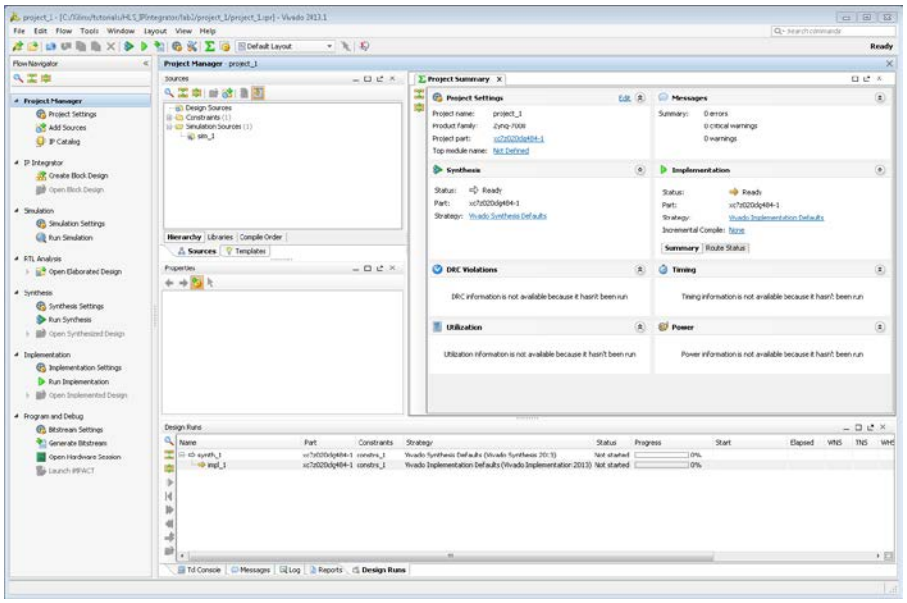


Figure 223 Initial Vivado Zynq Project

Step 3: Add HLS IP to the IP Catalog

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

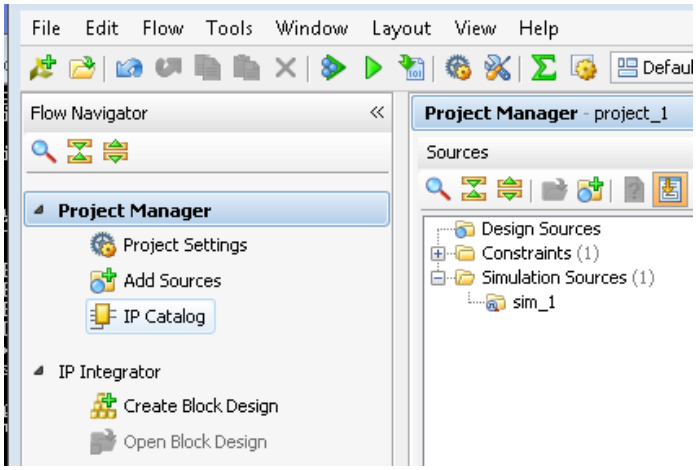


Figure 224 Open the IP Catalog

The IP Catalog will appear in the main pane of the workspace.

2. Click on the IP Settings icon (Figure 225)

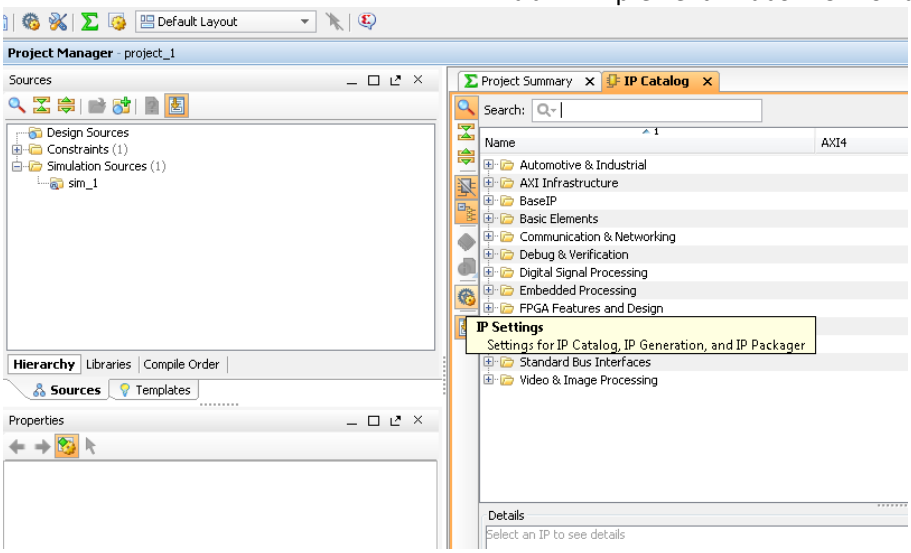


Figure 225 Open the IP Catalog Settings

3. In the IP Settings dialog, click on Add Repository…
4. In the IP Repositories dialog,
    a. Browse to the tutorial directory location and click on the Create New Folder icon.
    b. Enter "vivado_ip_repo" in the resulting dialog (Figure 226).
    c. Click OK.
    d. Press Select to close the IP Repository.

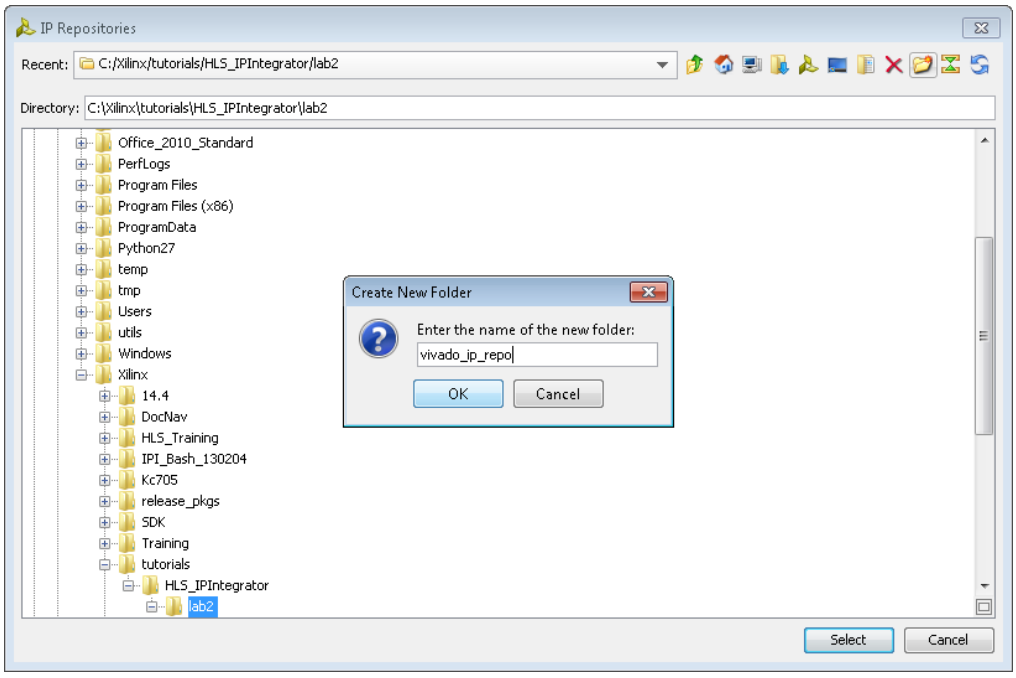


Figure 226 IP Repository

5. Back in the IP Setting dialog,
    a. Click Add IP.
    b. In the Select IP to Add to Repository dialog, browse to the location of the HLS IP: Using_IP_with_Zynq/lab1/hls_macc/vhls_prj/solution1/impl/ip/
    c. Select the IP Catalog package Xilinx_com_hls_hls_macc_1_00)a.zip file (Figure 227)
    d. Click OK

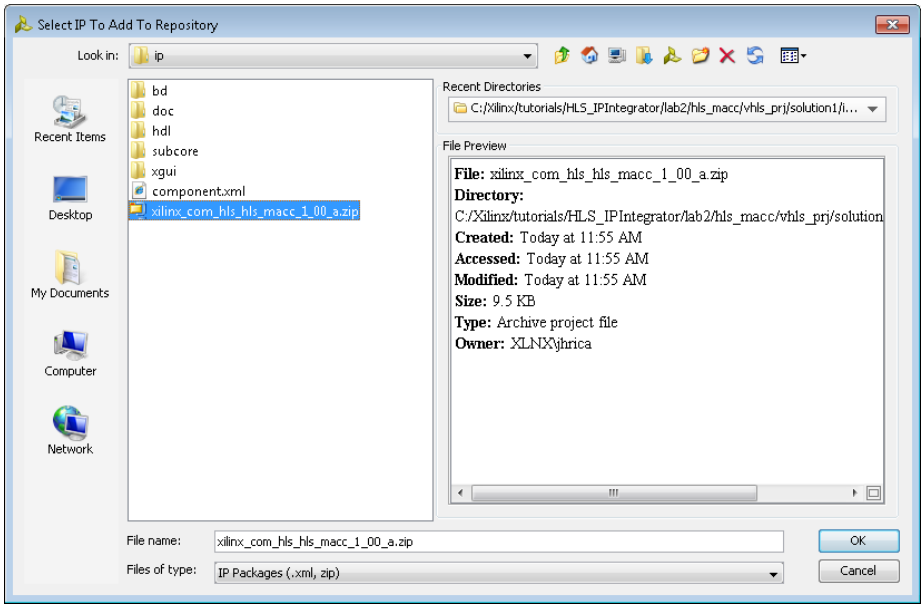


Figure 227 Add IP to the Repository

6. The new HLS IP should now show up in the IP Settings dialog

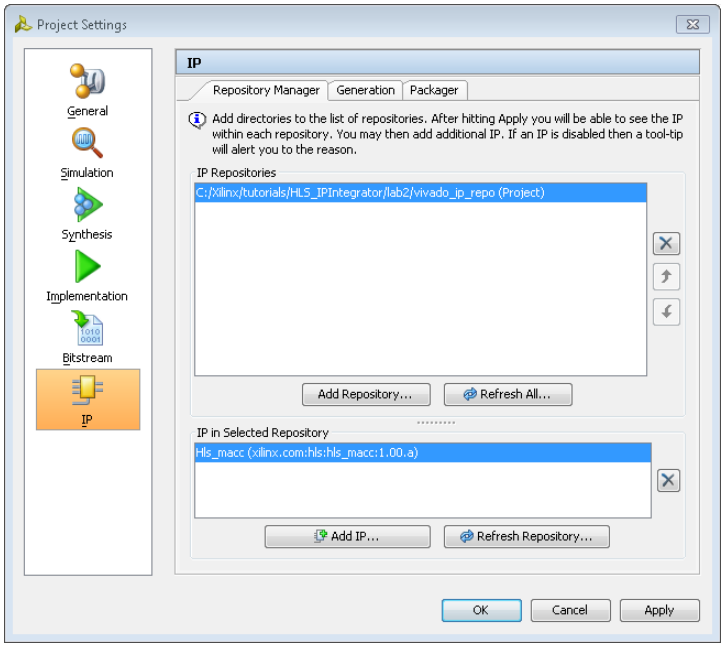


Figure 228 HLS IP in the Repository

7. Click OK to exit dialog
8. There should now be a Vivado HLS IP category in the IP Catalog and if expanded the Hls_macc IP should be displayed (Figure 229).

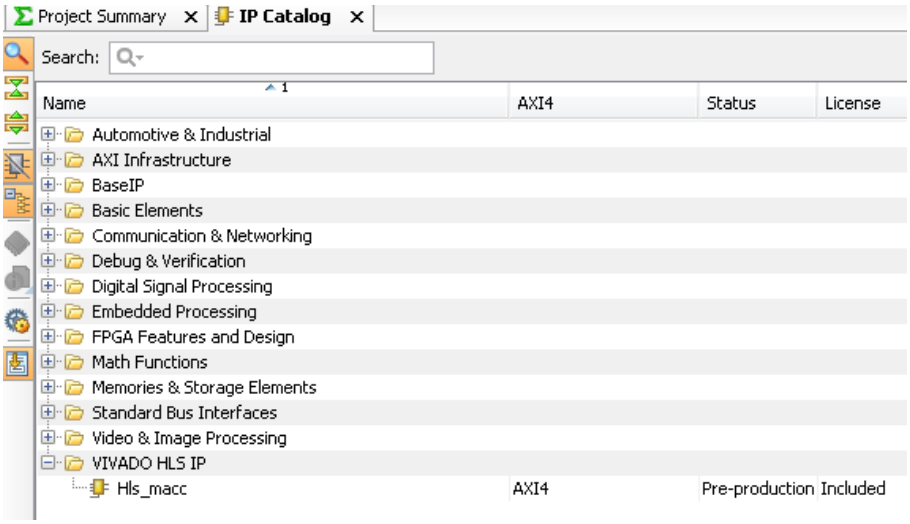


Figure 229 HLS IP in the IP Catalog

Step 4: Creating an IP Integrator Block Design of the System

1. In the IP Integrator area of the Flow Navigator, click on Create Block Design and enter "Zynq_Design" in the dialog

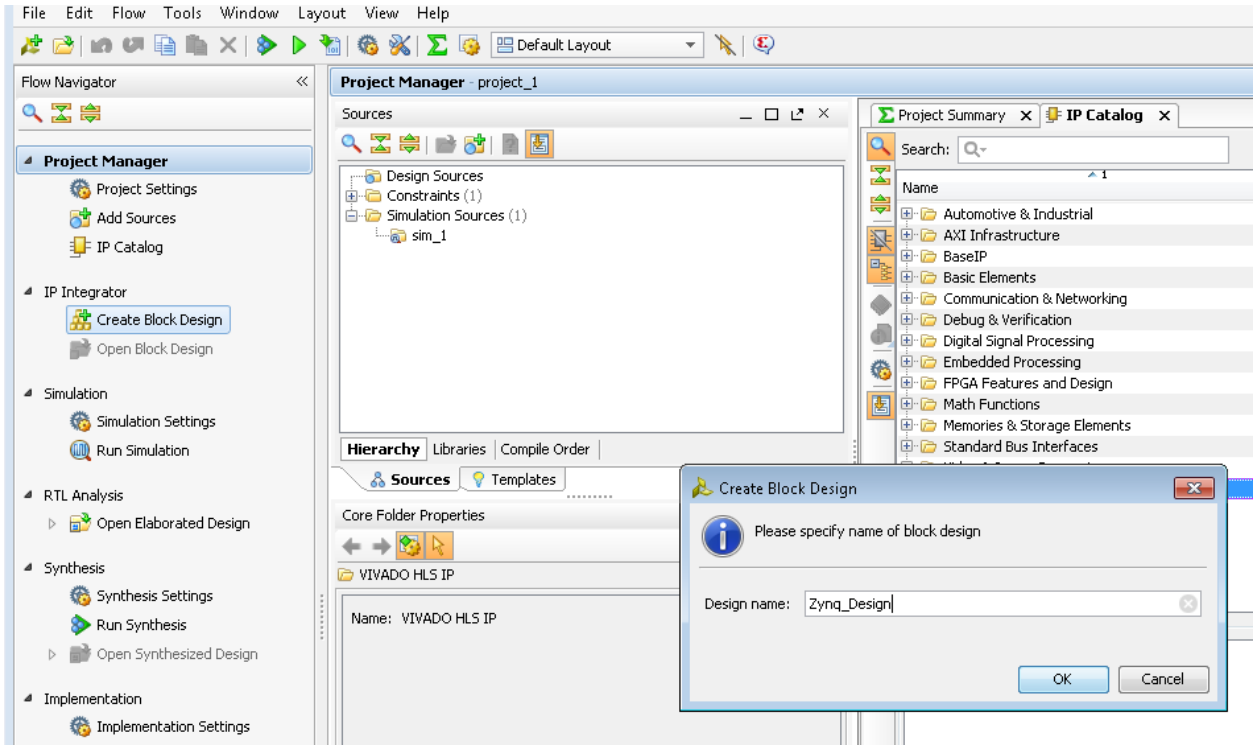


Figure 230 Create the Zynq Design

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

2. Click on the Add IP link under the title bar, which will pop up an IP search dialog.
    a. Type in "proce" into the Search text entry box.
    b. Select the ZYNQ7 Processing System item and hit enter

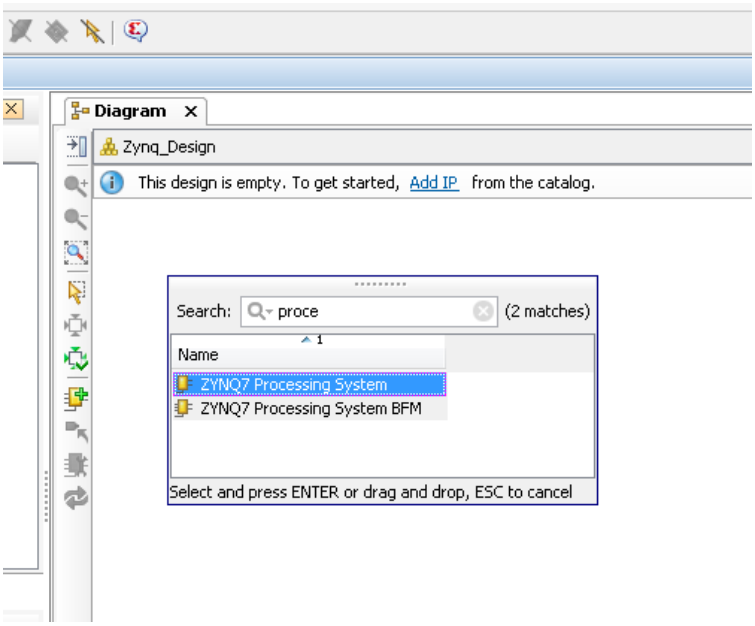


Figure 231 Add a CPU Processor to the Design

An IP symbol for the ZYNQ7 Processing System will appear on the canvas

3. Double-click on the ZYNQ IP symbol to open its Re-customize IP dialog.
    a. Click on Presets icon, select ZC702 Development Board Template (Figure 232).
    b. Click OK to exit the Configurations Presets dialog

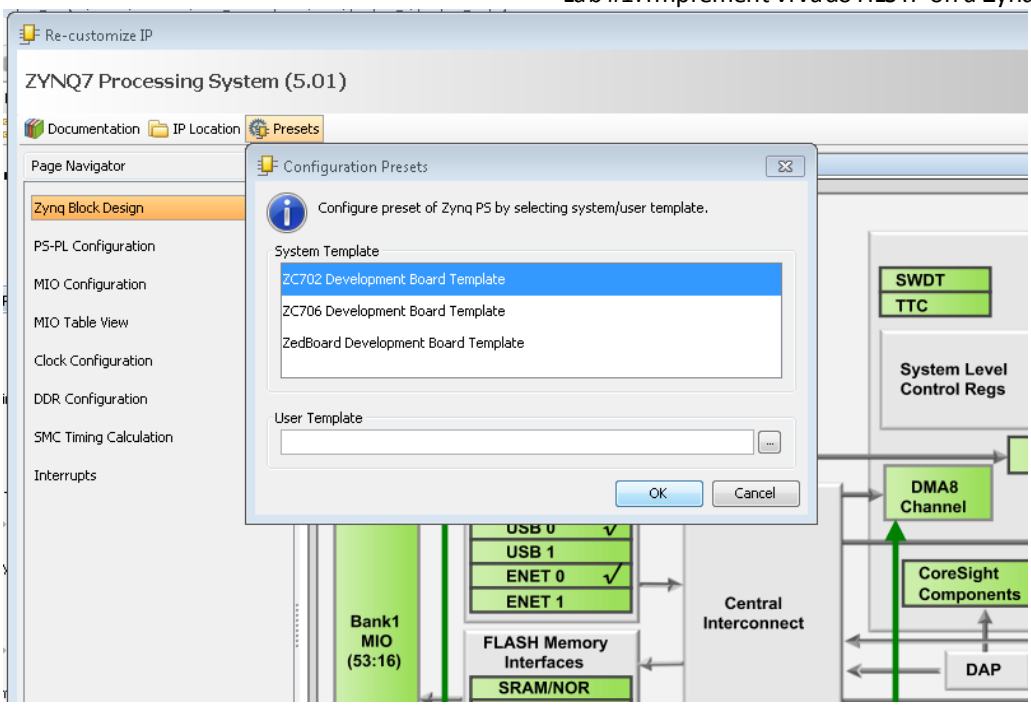


Figure 232 Configure the Zynq Processor

4. Click on 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

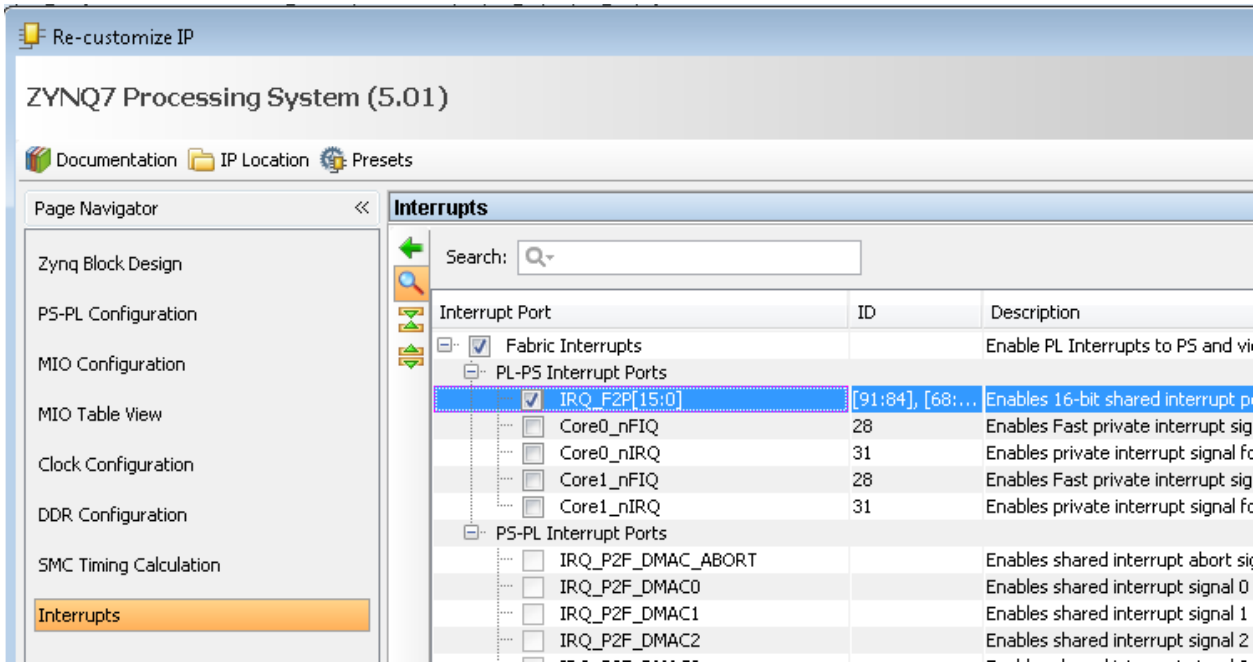


Figure 233 Zynq Processor Interrupt Configuration

IPI provides Designer Assistance to automate certain tasks, such as making the correct external connections to DDR memory and Fixed IO for the ZYNQ PS7.

5. Click on the Run Block Automation link under the title bar (Figure 234).
    a. Select /processing_system7_1.
    b. Click OK to complete in the resulting dialog.

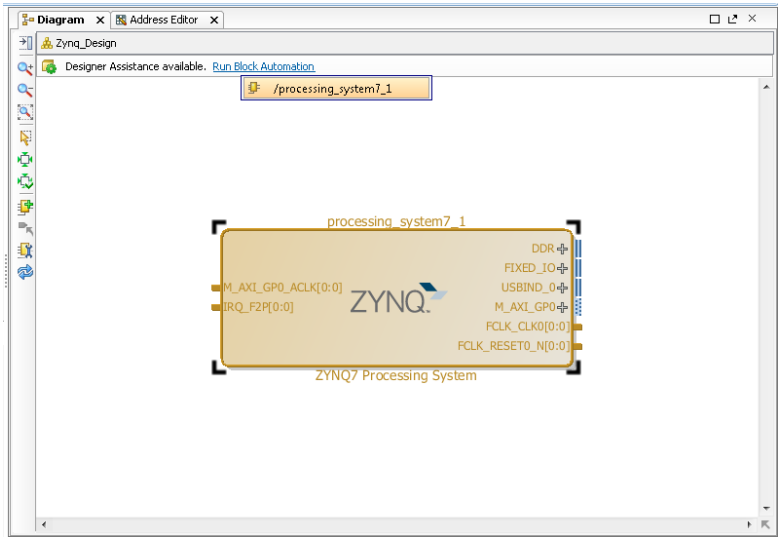


Figure 234 Run Automation

6. Add HLS IP to the design by right-clicking in open space of canvas and selecting Add IP from the context menu.
    a. Type "hls" in the Search text entry box and hit Enter to add it to design (Figure 235)

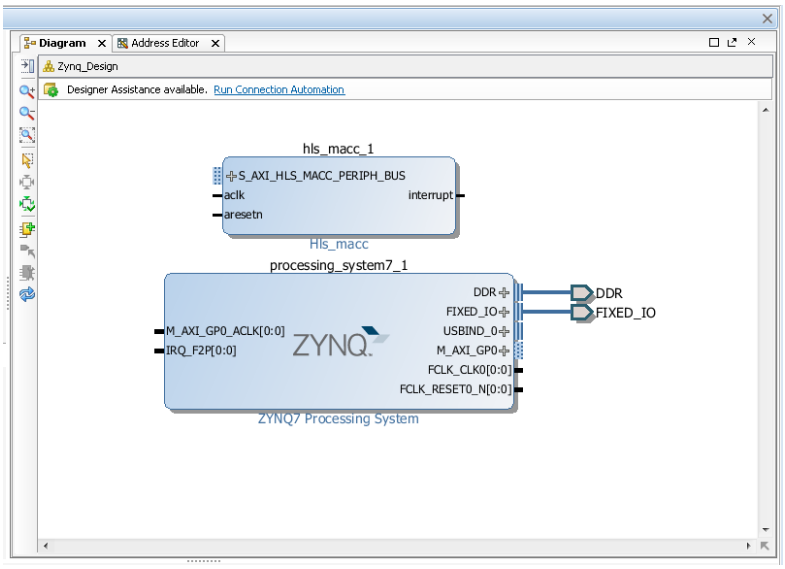


Figure 235 processor and HLS IP

Designer assistance is also available to automate the connection of IP block to each other.

7. Click on the Run Connection Automation link at the top of the canvas.
8. Select /hls_macc_1/S_AXI_HLS_MACC_PERIPH_BUS and click OK in the resulting dialog to automatically connect the HLS IP to the M_AXI_GP0 interface of the PS7.

This will add an AXI Interconnect (instance: processing_system7_1_axi_periph), a Proc Sys Reset block (instance: proc_sys_reset) and make all necessary AXI related connections to create the design shown in Figure 236.

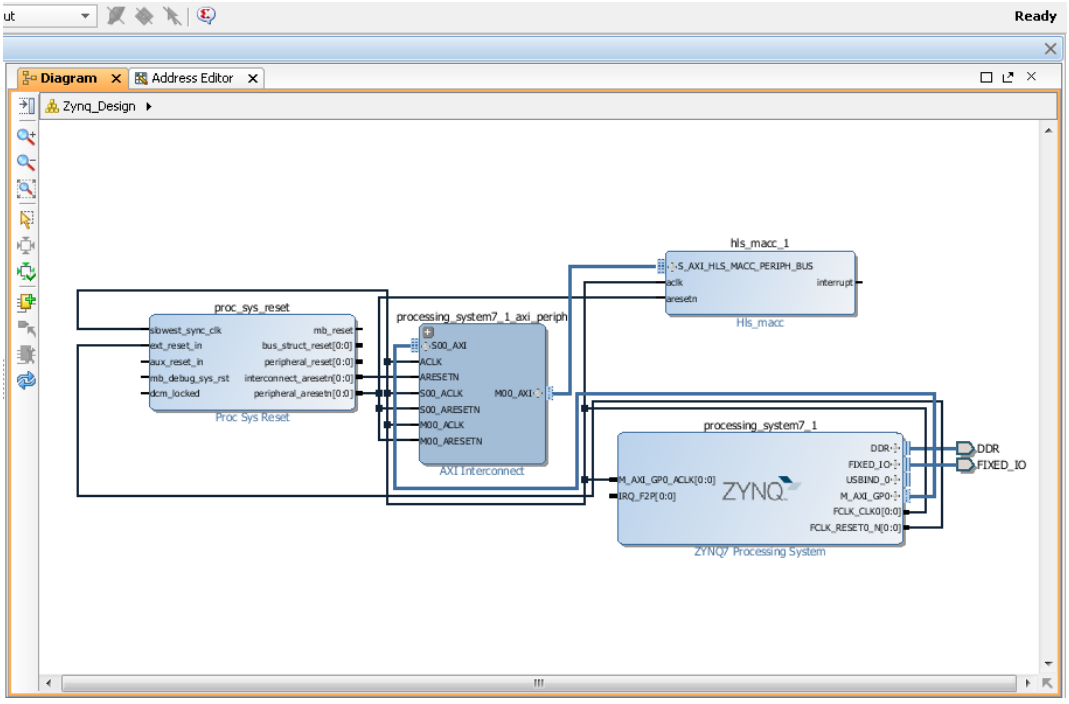


Figure 236 AXI4 Interconnect

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

9. Bring the cursor over the interrupt pin on the hls_macc_1 IP symbol.
    a. When the cursor changes to pencil shape left-click and drag to the IRQ_F2P[0:0] port of the PS7 and release, completing the connection
10. Bring the Address Editor tab forward and confirm that the hls_macc_1 peripheral has been assigned a master address range; if not click on the Auto Assign Address icon

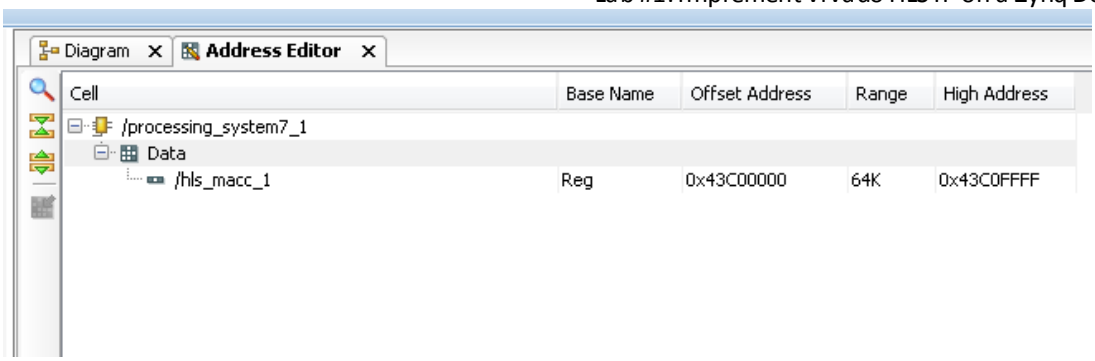


Figure 237 Address Editor

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

11. Click on the Validate Design icon in the toolbar.
12. Upon successful validation, save (control-s) the Block Design.

Step 5: Implementing the System

Before proceeding with the system design, implementation sources must be generated and an HDL wrapper created as the top-level module for synthesis and implementation.

1. Return to the Project Manager view by clicking on Project Manager in the Flow Navigator.
2. In the Sources browser in the main workspace pane a Block Diagram object named Zynq_Design should be at the top of the Design Sources tree view. (Figure 238)
3. Right-click on this object and select Generate Output Products.
4. In the resulting dialog, click OK to start the process of generating the necessary source files

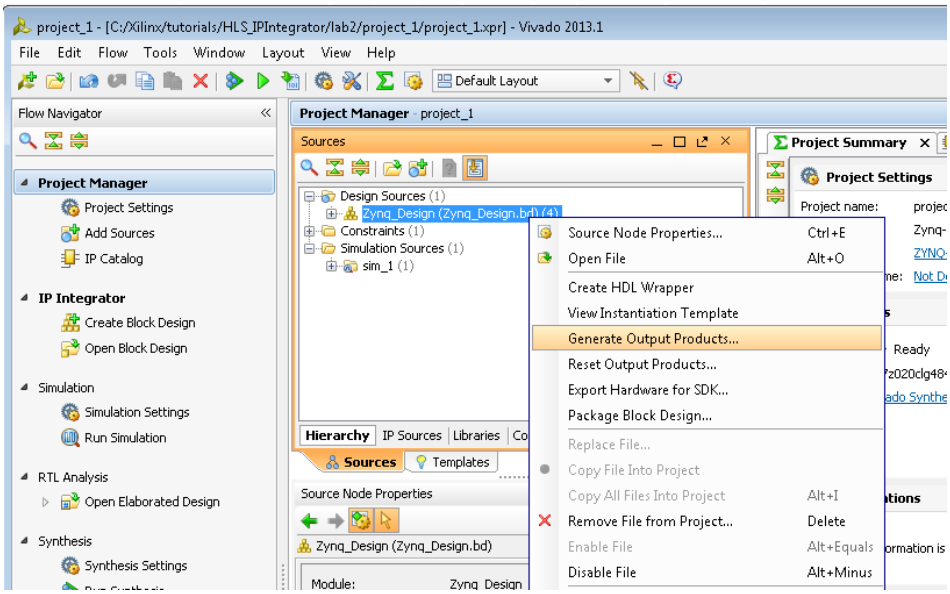


Figure 238 Wrapper Generation

5. Again, right-click on the Zynq_Design object, select Create HDL Wrapper and click OK to exit the resulting dialog.

The top-level of the Design Sources tree will now be Zynq_Design_wrapper.v file. The design is now ready to be synthesized, implemented and have an FPGA programming bitstream generated.

6. The remainder of the flow can be initiated by clicking on Generate Bitstream now.
7. 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

The design is now ready to be exported to Xilinx SDK in order to create software to be run on a ZC702 board (if available). A driver for the HLS block was generated during HLS export of the Vivado IP Catalog package and must be made available in SDK in order 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 (Figure 239) ensure that Include Bitstream and Launch SDK options are ticked then click OK to complete

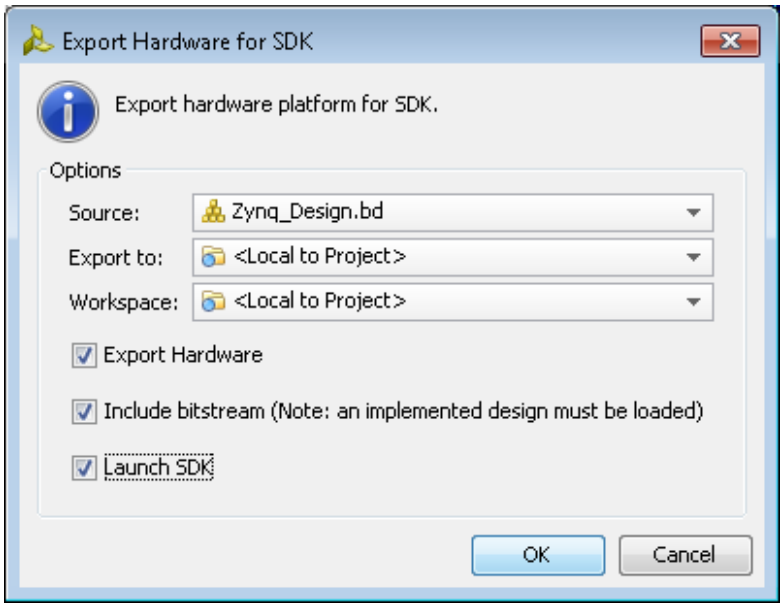


Figure 239 Export to SDK Dialog Window

2. SDK will open. If the Welcome page is open, close it.
3. Create a new SDK software repository and add the HLS block drivers to it

a. From the XilinxTools menu, select Repositories
b. In the Repositories Preferences page click on New (upper right)
c. In the Browse For Folder dialog, navigate to the tutorial file set directory (/lab1)
d. Click on Make New Folder
e. Enter sdk_sw_repo as the new name (Figure 240).

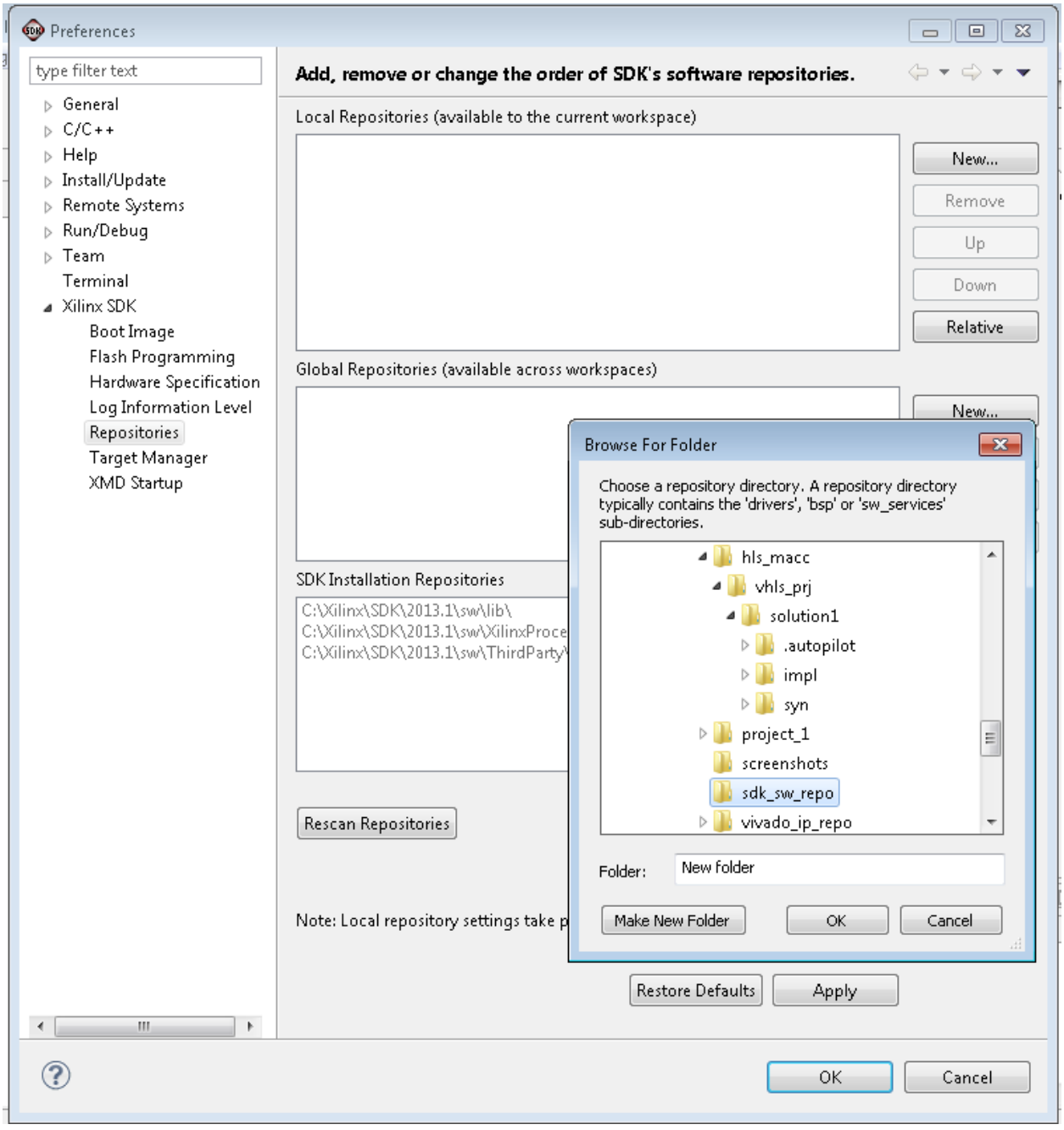


Figure 240 SDK Project Properties

4. From OS (command prompt or GUI file browser
    a. Copy the directory …/hls_macc/vhls_prj/impl/drivers (including all contained directories and files) to the new "sdk_sw_repo" directory. (Figure 241)

IMPORTANT: *The software repository must have a hierarchy of the form …/sdk_sw_repo/drivers/ with any number of sub-directories, such as hls_macc_top_v2_00_a/*

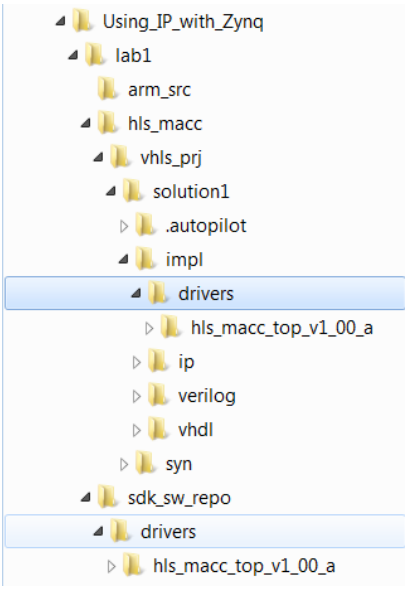


Figure 241 SDK Drivers Repository

    b. Click OK to exit the Preferences dialog
5. From the SDK File menu, select New > Application Project.
    a. In the New Project dialog enter a project name: Zynq_Design_Test,
    b. Click next, select Hello World template
    c. Click Finish
6. Create a Hello World application (also creates BSP)
    a. File > New > Application Project
    b. Enter the project name Zynq_Design_Test
    c. Click next
    d. Select Hello World (if not default).
    e. Click Finish.

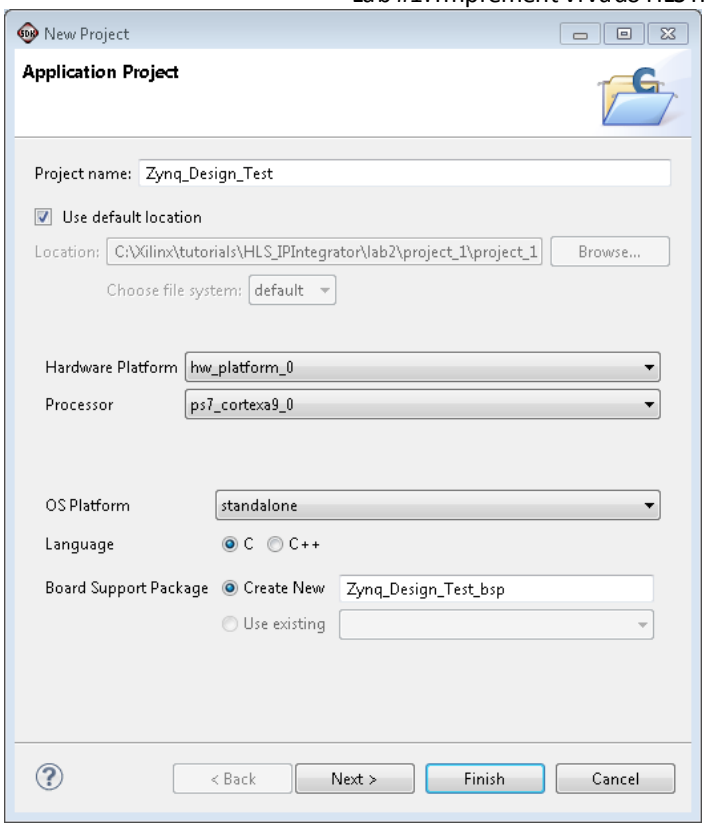


Figure 242 Application Project

7. Power up ZC702 board and test Hello World application
    a. Attach power cable, micro-USB from PC to Digilent JTAG-USB connector and mini-USB to UART (J17) connector. Ensure Silicon Labs USB to UART and Digilent JTAG drivers are setup on system. Switch the power on.
    b. XilinxTools > Launch Hardware Server (or toolbar icon) (Figure 243)

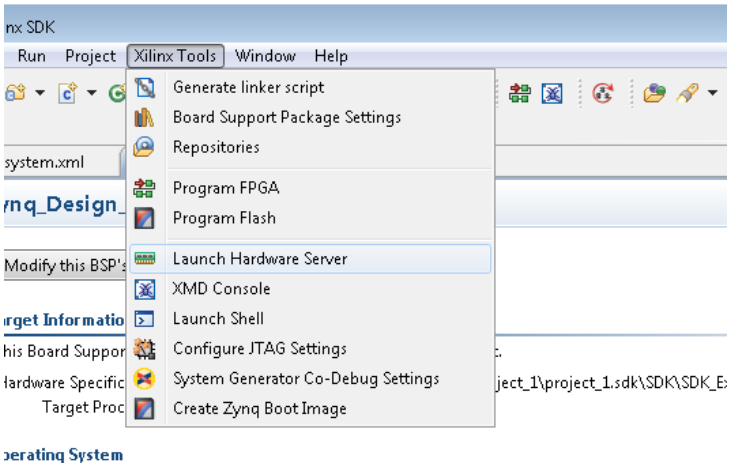


Figure 243 Launch Hardware

c. XilinxTools > Program FPGA (or toolbar icon). The Done LED (DS3) should be lit afterwards.

8. Setup a Terminal in the tab at bottom of workspace:
   a. Click on Connect icon (Figure 244)

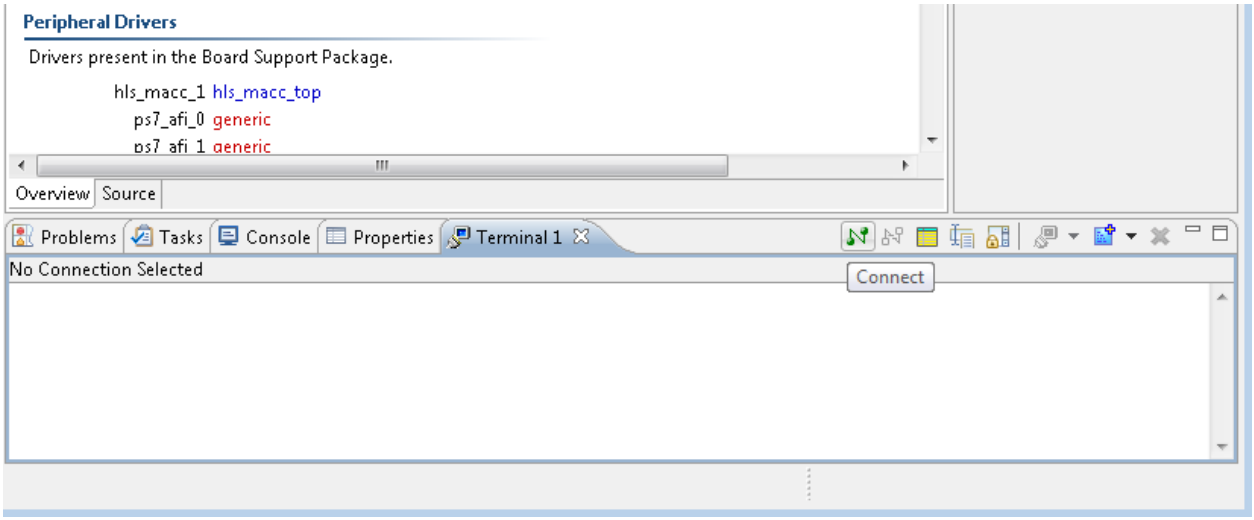


Figure 244 The Connect Icon

   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 not sure, open the Device Manager and identify the port with the Silicon Labs driver under Ports (COM & LPT)
   d. Change Baud Rate to 115200 (Figure 245)
   e. Click OK to exit Terminal Settings dialog

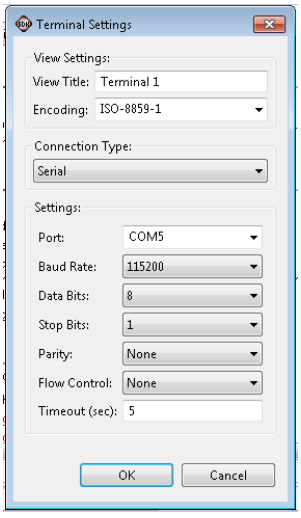


Figure 245 Terminal Settings

   f. Check that terminal is connected by message in tab title bar

9. Right-click on application project (Zynq_Design_Test) in Explorer pane (Figure 246)
    a. Run As > Launch on Hardware

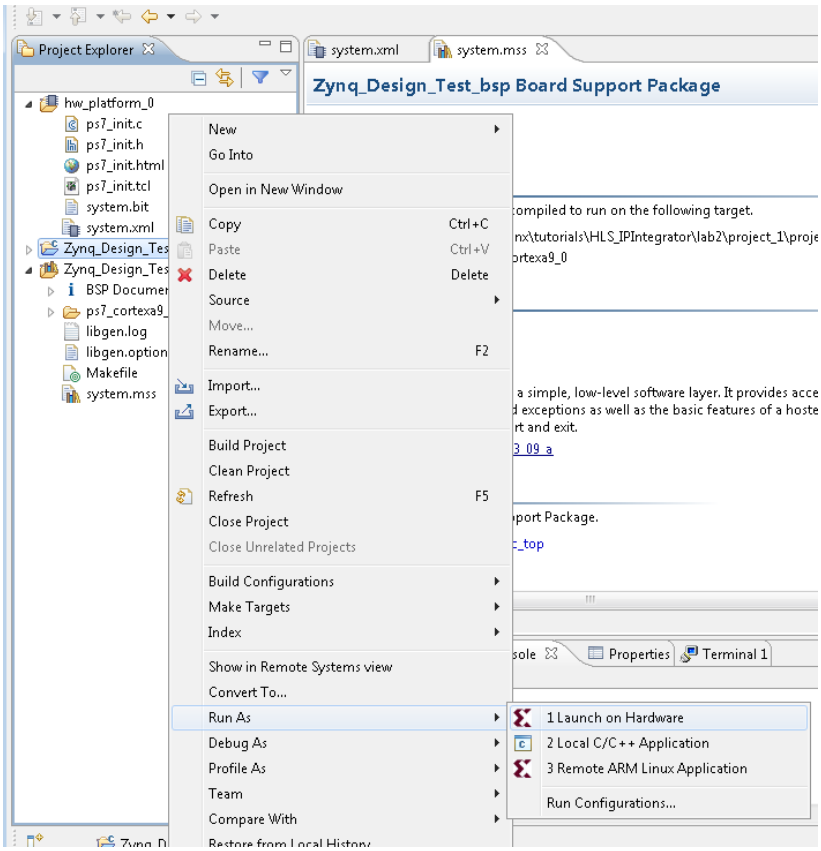


Figure 246 Run the Application Project

10. Switch to Terminal tab and confirm that "Hello World" was received (Figure 247)

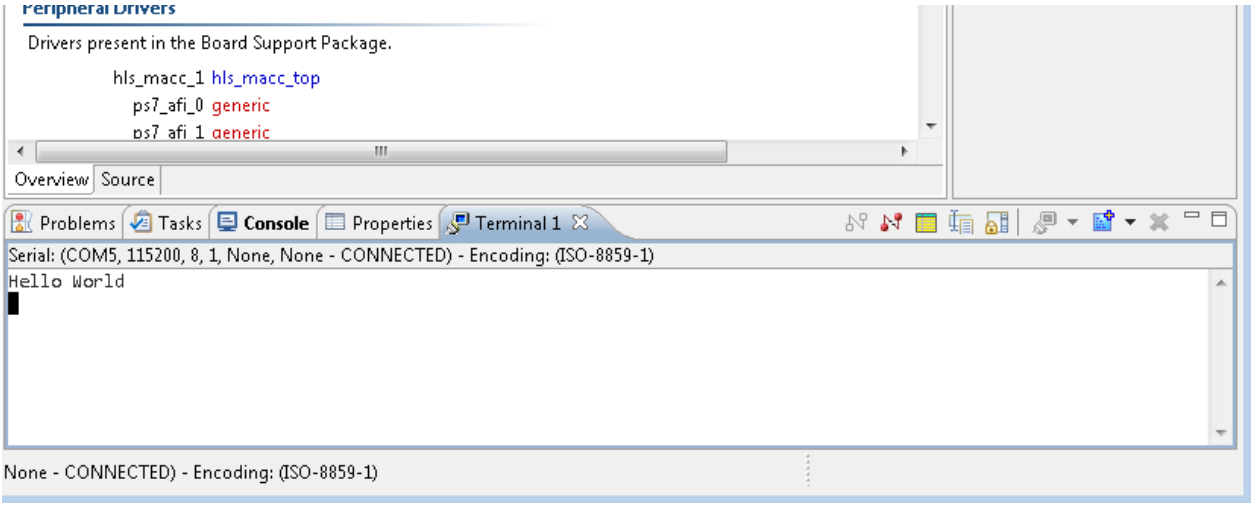


Figure 247 Console Output

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 will be 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 HLS block and interrupt controller instance data, which will be used passed to driver API calls as handles 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);
    }
```

7. 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);
        }
    }
```

8. 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_1_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_1_INTERRUPT_INTR);
    return XST_SUCCESS;
}
```

9. Define a software model of the HLS hardware functionality with which reference results can be compared.

```
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;
}
```

10. 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;
}
```

11. Save (control-s) the modified source file and SDK will automatically attempt 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 done for the previous Hello World application code.

Upon success, the Terminal session should look similar to Figure 248.

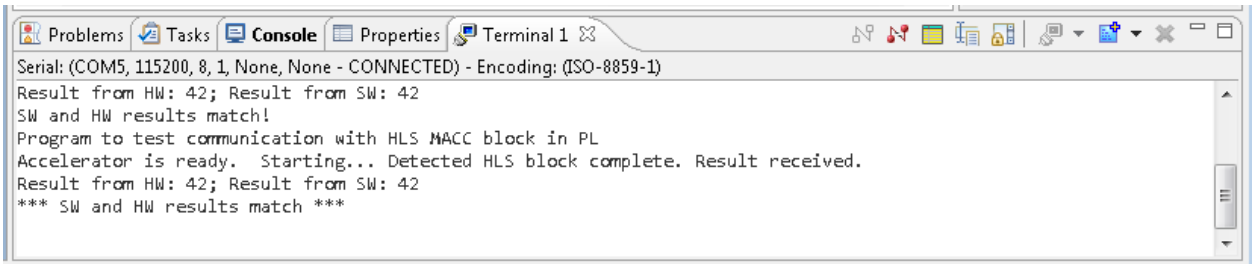


Figure 248 Console Output with Updated C Program

This concludes this tutorial.

Conclusion

In this tutorial, you learned:

- How to create Vivado HLS IP using a Tcl script.
- How to import a design into IP integrator (IPI) and connect it to a Zynq PS.
- Finally, how to create a software program to control the HLS IP and run this on a board.

Using HLS IP in System Generator for DSP

Overview

The RTL created by High-Level Synthesis can be packaged as IP and used inside System Generator for DSP (Vivado). This tutorial shows how this process is performed and demonstrates how the design can be used inside System Generator for DSP.

This tutorial consists of a single lab exercise.

Lab1

Generate a design using Vivado HLS and package the design for use with System Generator for DSP. Then include the HLS IP into a System Generator for DSP design and execute an RTL simulation.

Tutorial Design Description

The tutorial design file can be downloaded from the Xilinx website. Refer to the information in Obtaining the Tutorial Designs.

This tutorial uses the design files in the tutorial directory **Vivado_HLS_Tutorial\Using_IP_with_SysGen**

The sample design is a FIR filter using streaming interfaces modeled using 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 will integrate 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 provide Tcl script. The script will run HLS C-synthesis, RTL co-simulation and package the IP for the two HLS designs (hls_real2xfft and hls_xfft2real).

1. Open the Vivado HLS Command Prompt.
    a. On Windows use **Start** > **All Programs** > **Xilinx Design Tools** > **Vivado 2013.1** > **Vivado HLS** > **Vivado HLS 2013.1 Command Prompt (**Figure 249).
    b. On Linux, open a new shell.

Figure 249 Vivado HLS Command Prompt

2. Using the command prompt window, change the directory to Vivado_HLS_Tutorial\Using_IP_with_SysGen\lab1 (Figure 250).
3. Type vivado_hls –f run_hls.tcl to create the HLS IP (Figure 250).

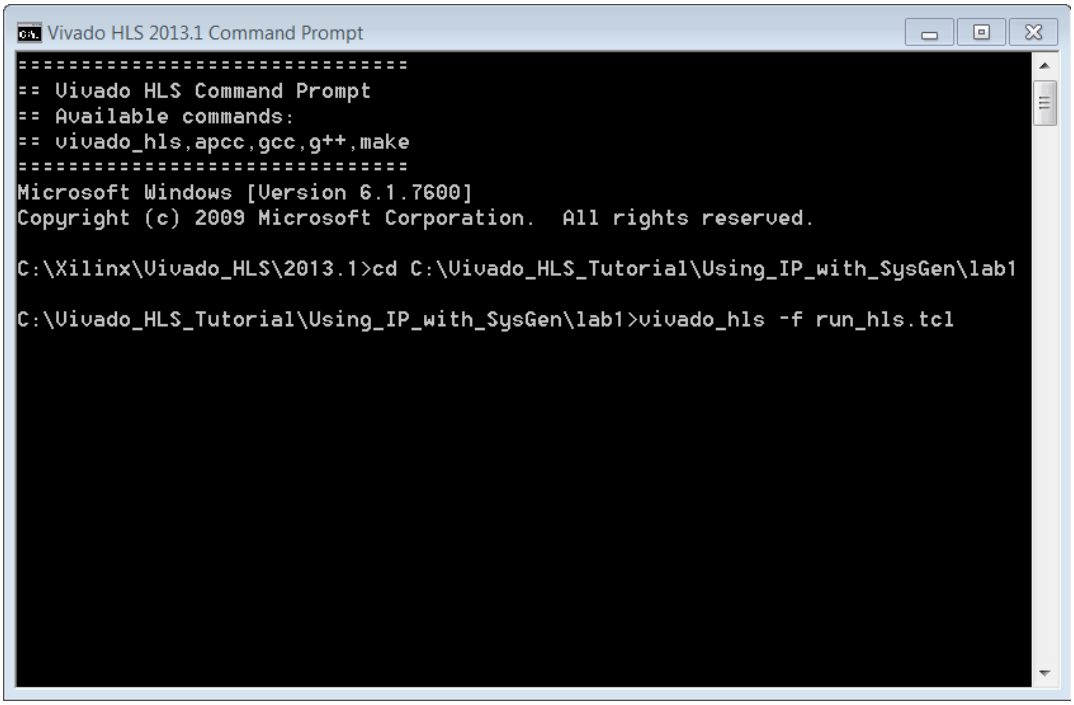


Figure 250 Create the HLS Design

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 will be 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 the Vivado HLS IP block is integrated into a System Generator design.

Step 2: Open the System Generator Project

1. Open System Generator for DSP.
    a. On Windows use the desktop icon **(**Figure 251).
    b. On Linux, open a new shell and type.

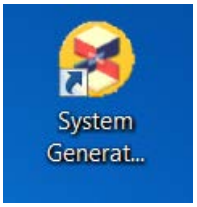


Figure 251 System Generator 2013.1 Icon

2. When Matlab invokes, use the Open toolbar button to select open (Figure 252).

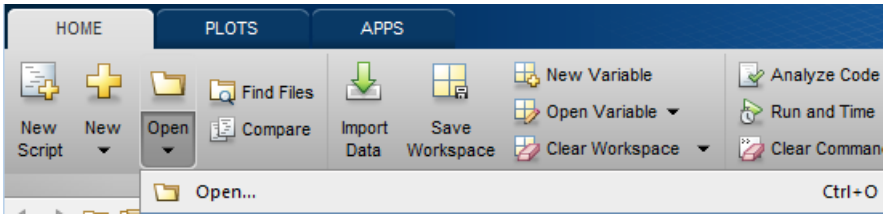


Figure 252 Open the System Generator Design

3. Navigate to the tutorial directory Vivado_HLS_Tutorial\Using_IP_with_SysGen\lab1 and select he file fir_sysgen.mdl (Figure 253).

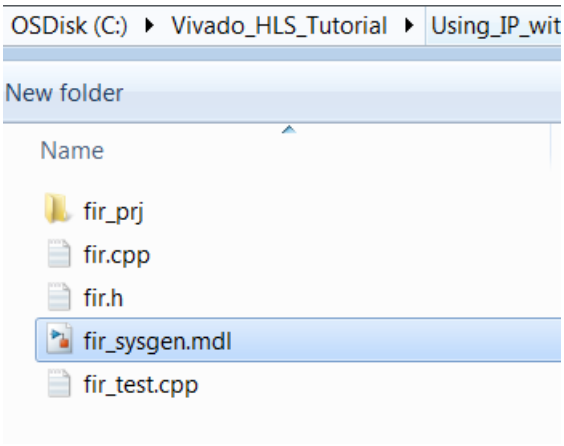


Figure 253 Select File fir_sysgen.mdl

When System Generator invokes, the design has all of the blocks and ports already instantiated: except the HLS IP.

4. Right-click with the mouse in the canvas and select Xilinx BlockAdd as shown in Figure 254.

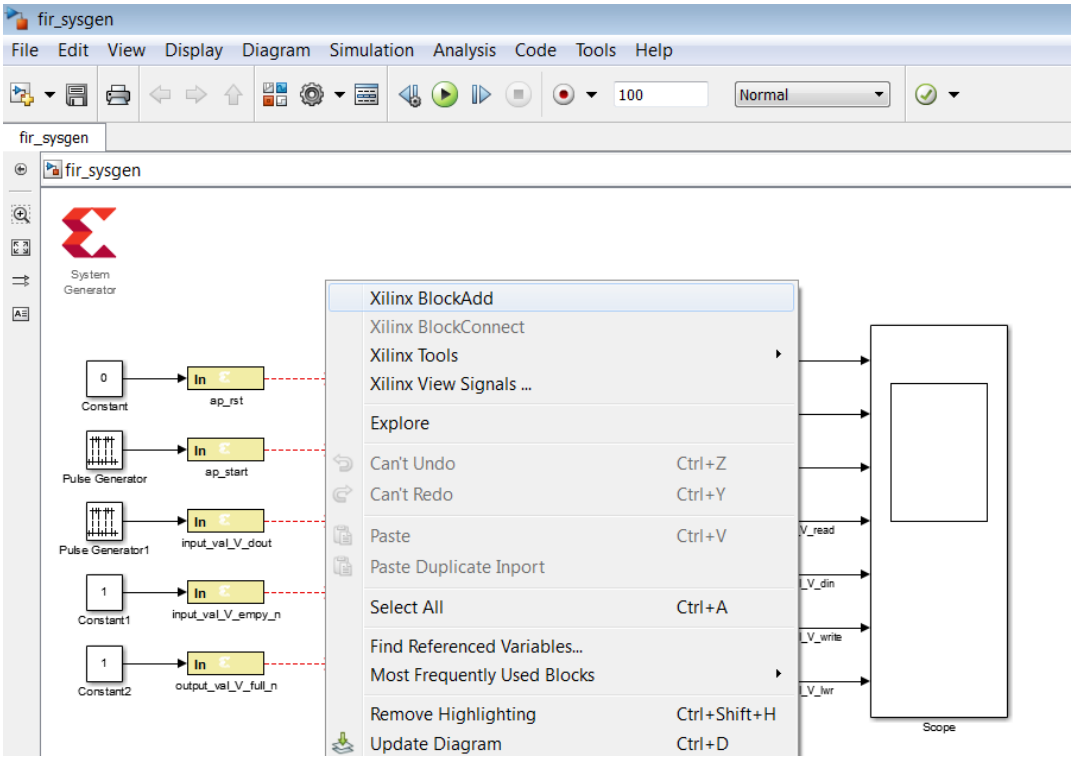


Figure 254 Adding an new Block

5. Type "hls" in the Add Block field (Figure 255).
6. Select Vivado HLS.

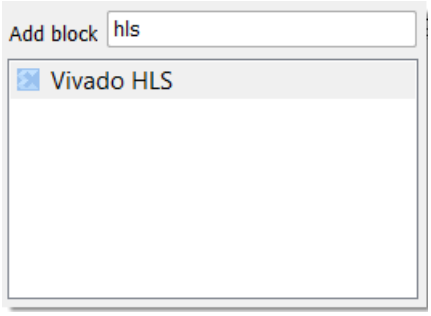


Figure 255 Selecting a Vivado HLS IP Block

7. Double-click on the Vivado HLS block to open the Vivado HLS dialog box.
8. Navigate into the fir_prj project and select the solution1 folder. (Figure 256)

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

9. Click OK to load the IP block.

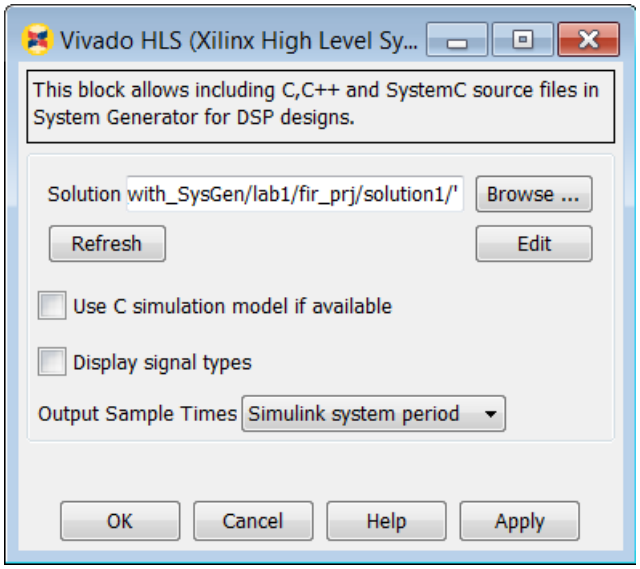


Figure 256 Selecting the FIR IP Block

The FIR IP block in instantiated into the design.

10. Connect up the design IO ports to the ports on the FIR IP block as shown in 257.

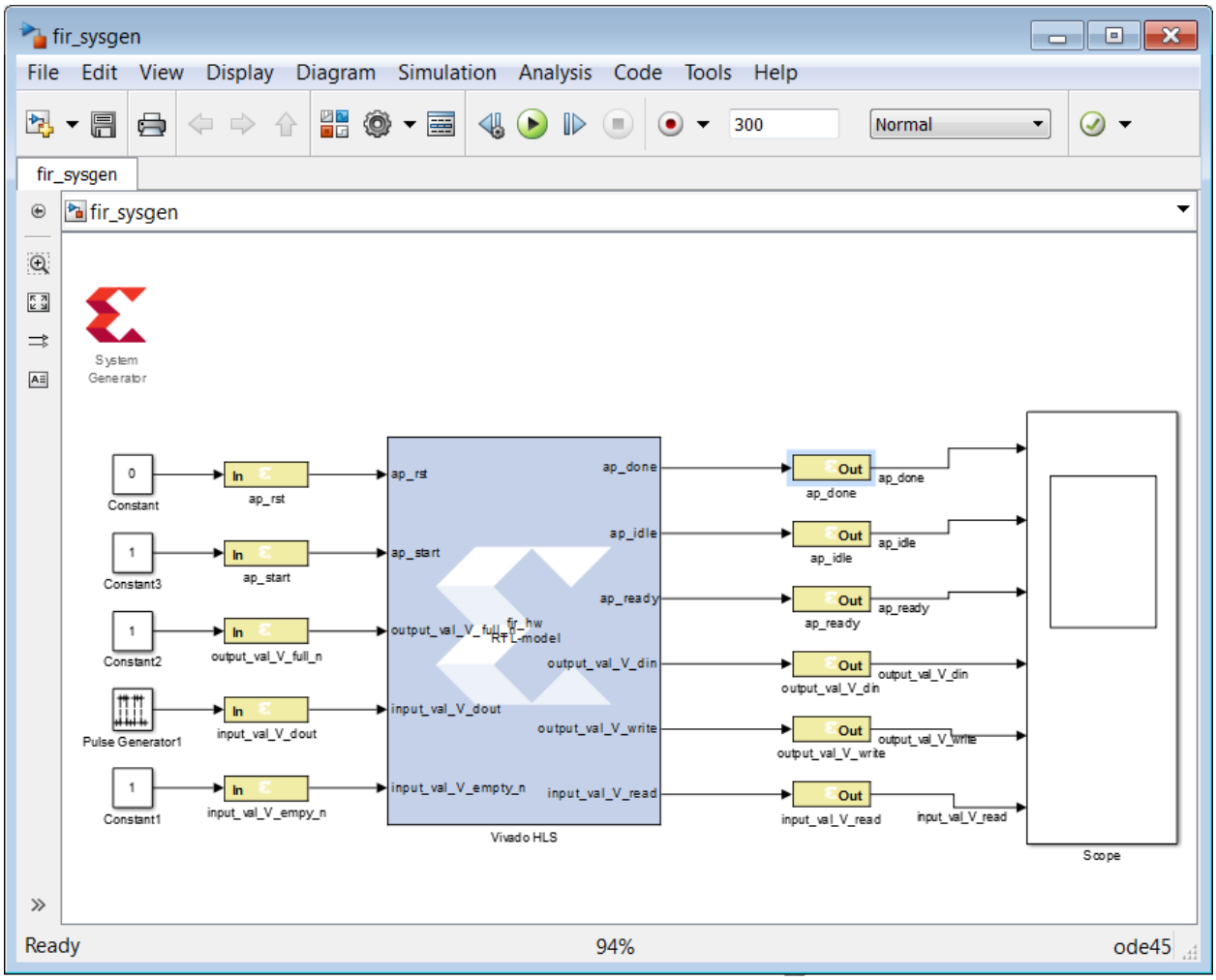


Figure 257 Design with All Connections

11. Ensure the simulation stop time says 300 (Figure 257).
12. Press the Run button on the toolbar to execute simulation.
13. Double-click on the Scope block to view the simulation waveforms.

This concludes the lab on using HLS IP in System Generator for DSP.

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.