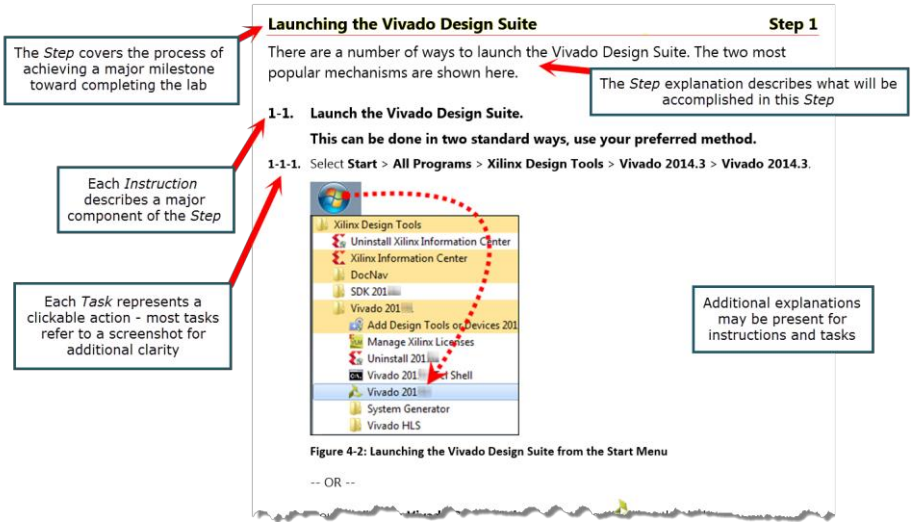
C-based Design: High-Level Synthesis with the Vivado HLS Tool Lab Workbook

dsp-hls-2016.1-wkb-lab-rev1

C-based Design: High-Level Synthesis with the Vivado HLS Tool Lab Workbook 2016.1

Lab FAQ

- Where can I get the files for the labs?
  - www.xilinx.com/training/downloads.htm
  - These are original files and do not contain any work that you may have performed.
  - Labs were developed using version 2016.1 of the tools. Later versions may work and will likely require you to update various pieces of IP. See the "Updating IP" topic in the *Lab Reference Guide* for instructions on how to update IP (Vivado Design Suite Operations > Vivado IP Integrator Operations > Updating IP).
- What if I cannot answer a question in the lab?
  - Do your best! The questions are meant to stimulate thought, not to test your knowledge. After you have pondered a question you can find the answer at the end of each lab.
- Where can I get more detailed information on a topic?
  - The *Lab Reference Guide* is a collection of "how to" topics for commonly performed tasks categorized by tool (Vivado Design Suite, Vivado Analyzer, SDK, etc.) and subdivided into major areas within the tool.
  - The *Lab Reference Guide* is available from the lab files download as well as from www.xilinx.com/training/downloads.htm.
- How do the instructions work?
  - The instructions are provided in layers.
    - Steps – these are the major/broadest aspects of solving the problem that the lab poses (X).
    - Instructions – these represent the significant instructions towards solving the issue outlined by the step (X-X).
    - Tasks – these are the finest granularity items that (when combined with the other tasks) solve the instruction they are subordinate to. Every task is a "clickable" event (X-X-X).



- What do KCU105, KC705, and KC7xx mean in the lab instructions and lab files?
  - There are two boards, the Kintex® UltraScale™ FPGA KCU105 evaluation board and the Kintex-7 FPGA KC705 evaluation board—each representing a different type of architecture.
    - The Kintex UltraScale FPGA KCU105 evaluation board represents the UltraScale architecture with the specific part number of **xcku040-ffva1156-2-e**.
    - The Kintex-7 FPGA KC705 evaluation board represents the 7 series architecture with the specific part number of **xc7k325tffg900-2**.
  - In addition, there is also a *fictitious* board KC7xx representing the 7 series architecture with a part number of **xc7k70tfbg676-2**.

  - The lab files are provided for both the UltraScale and 7 series architectures.
  - The lab instructions have been written such that both UltraScale and 7 series architecture users can work through the lab.
  - Follow the lab instructions based on the board that you choose to work on. Since it can be quite verbose to repeat board terminology throughout a lab, the following is used:
    - "KCU105" represents the Kintex UltraScale FPGA KCU105 evaluation board with the xcku040-ffva1156-2-e part.
    - "KC705" represents the Kintex-7 FPGA KC705 evaluation board with the xc7k325tffg900-2 part.
    - "KC7xx" represents the fictitious KC7xx board with the xc7k70tfbg676-2 part.

Table of Contents

Lab 1: Introduction to the Vivado HLS Tool Flow

2016.1

Abstract

This lab introduces how to perform basic actions using the Vivado® High-Level Synthesis (HLS) design flow.

This lab should take approximately 45 minutes.

Objectives

After completing this lab, you will be able to:

- Create a new project in the Vivado HLS tool GUI
- Simulate a C design by using a self-checking testbench
- Synthesize the design
- Perform design analysis using the Analysis Perspective view
- Perform co-simulation on a generated RTL design by using a provided C testbench
- Implement the design

Introduction

This lab provides an introduction to the major features of the Vivado® High-Level Synthesis (HLS) tool GUI flow. You will use the Vivado HLS tool in GUI mode to create a project. You will also simulate, synthesize, and implement the design provided.

In this lab, you will be using a C design to implement a discrete cosine transformation (DCT). The function implements a 2D DCT algorithm by first processing each row of the input array via a 1D DCT, then processing the columns of the resulting array through the same 1D DCT. It calls the *read_data*, *dct_2d*, and *write_data* functions.

The *read_data* function is defined at line 54 and consists of two loops: *RD_Loop_Row* and *RD_Loop_Col*. The *write_data* function is defined at line 66 and consists of two loops to perform writing the result. The *dct_2d* function, defined at line 23, calls the *dct_1d* function and performs transpose.

Finally, the *dct_1d* function, defined at line 4, uses *dct_coeff_table* and performs the required function by implementing a basic iterative form of the 1D Type-II DCT algorithm.

The following figure shows the function hierarchy on the left-hand side, the loops in the order they are executed, and the flow of data on the right-hand side.

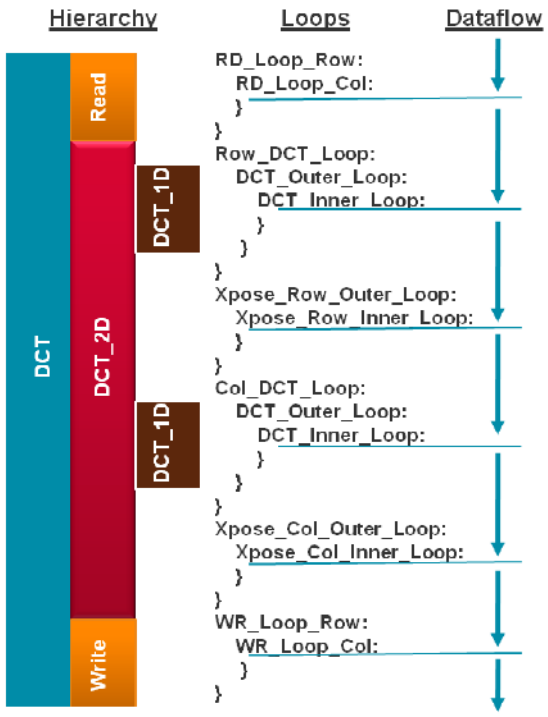
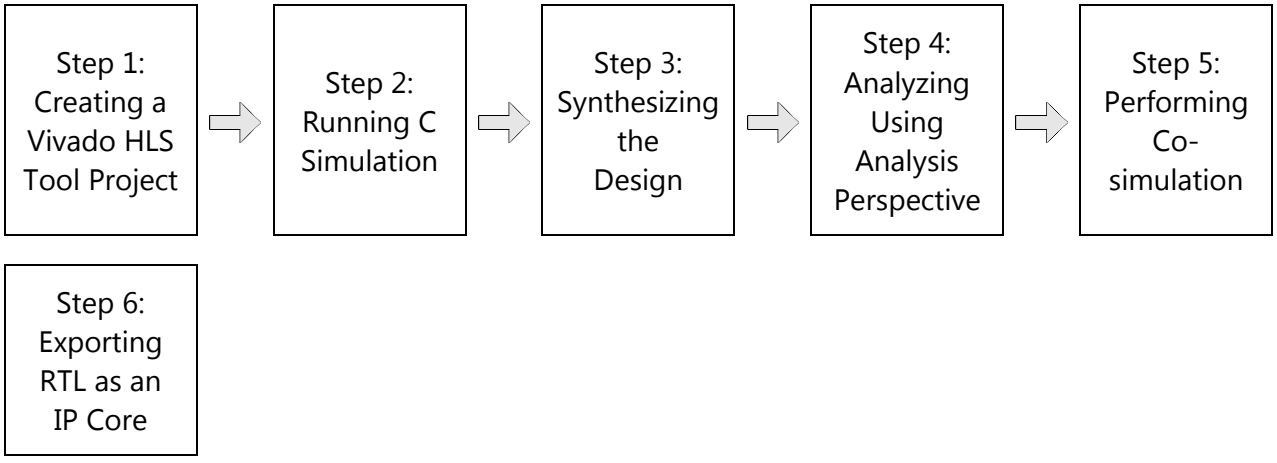


Figure 1-1: Design Hierarchy and Dataflow

General Flow



Creating a Vivado HLS Tool Project Step 1

In this step, you will launch the Vivado HLS tool GUI and create a new project for the provided C-based discrete cosine transformation (DCT) design.

There are a number of ways to launch the Vivado HLS tool. The two most popular mechanisms are shown here.

1-1. Launch the Vivado HLS tool.

1-1-1. Select **Start** > **All Programs** > **Xilinx Design Tools** > **Vivado 2016.1** > **Vivado HLS** > **Vivado HLS 2016.1**.

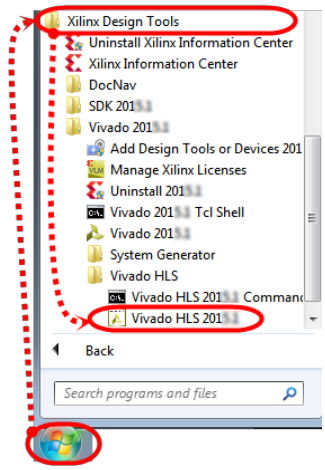


Figure 1-2: Launching the Vivado HLS Tool

-- OR --

Double-click the **Vivado HLS** shortcut icon ( ) on the desktop.

The Vivado HLS tool opens to the Welcome window. From the Welcome window you can create a new project, open examples, and access documentation and examples.

Figure 1-3: Vivado HLS Welcome Screen

Here you will learn to create a new Vivado HLS project from scratch.

1-2. Create a Vivado HLS project named *dct_prj*.

1-2-1. From the Welcome Page, click **Create New Project**.

Figure 1-4: Creating a New Vivado HLS Project

1-3. The Project Configuration dialog box asks for a project name and location.

1-3-1. Enter **dct_prj** in the Project name field (1).

1-3-2. Enter **C:\training\hls\labs\hls_tool_flow\zc702\dct** in the Location field (2).

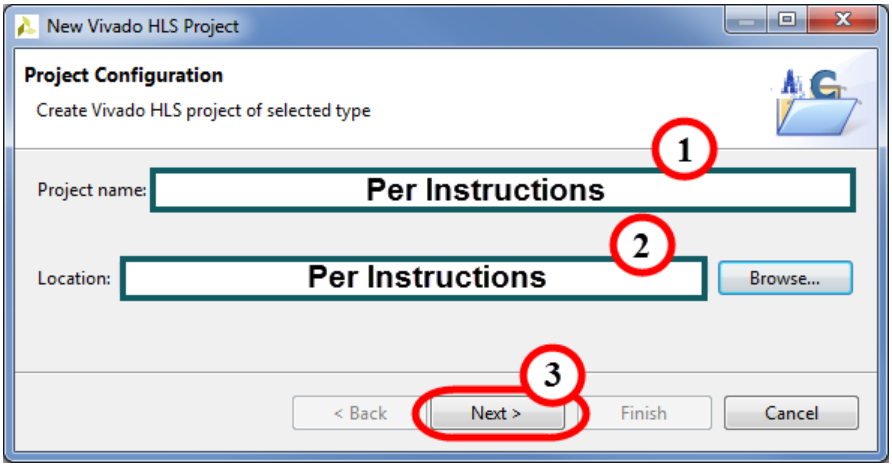


Figure 1-5: Configuring a New HLS Project

1-3-3. Click **Next** (3).

1-4. The Add/Remove Files dialog box opens. Here you will be invited to add existing files or create new sources.

1-4-1. Click **Add Files**.

The Open File dialog box opens.

Note: If do not have existing files at this moment and you want to create new ones, click **New File**.

1-4-2. Browse to *C:\training\hls\labs\hls_tool_flow\zc702\dct*.

1-4-3. Select **dct.c**.

The Vivado HLS tool automatically adds the working directory (project directory) and any directory that contains C files added to the project to the search path. Hence, header files that reside in these directories are automatically included in the project (no need to explicitly specify them). You must specify the path to all other header files (if any) by clicking the Edit CFLAGS button.

1-4-4. Click **Open** to add these files.

Note that you can add compiler directives specific to each entry at this point.

1-4-5. Click **Browse** next to the Top Function field.

The Select Top function dialog box opens, which lists all the functions available from the specified source files.

1-4-6. Select **dct (dct.c)** from the list.

Note: You can also manually enter the name of the top function in the Top Function field.

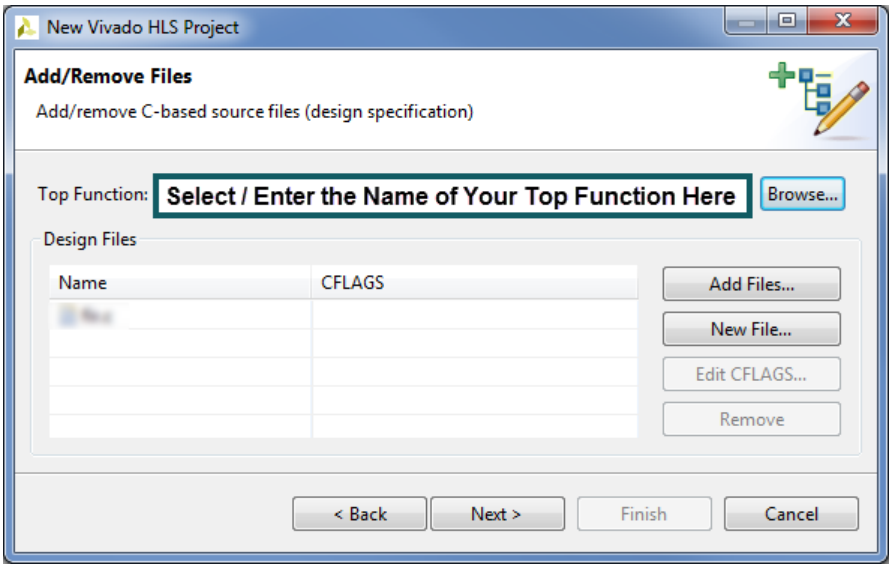


Figure 1-6: Adding Files to a New Vivado HLS Project

1-4-7. Click **Next**.

1-5. Add any existing testbench files.

If you have (or want) any testbench files they can be entered here. Sometimes the testbench is built into the synthesizable file.

1-5-1. Click **Add Files**.

1-5-2. Navigate to *C:\training\hls\labs\hls_tool_flow\zc702\dct*.

1-5-3. Select **dct_test.c, in.dat and out.golden.dat**.

1-5-4. Click **Open** to add these files.

1-5-5. Click **Next**.

1-6. Finally, it is time to specify some of the physical parameters of the design.

1-6-1. By default, **solution1** is populated in the Solution Name field.

No changes are required.

1-6-2. Set the clock period to **10**.

You can leave the Uncertainty field blank.

1-6-3. Click the **Browse** button to select a part or board.

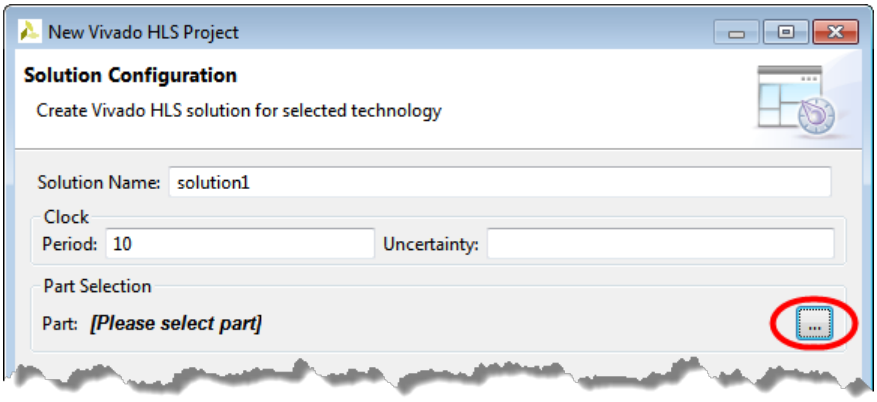


Figure 1-7: Locating the Board Browse Button

1-6-4. Click **Boards** as shown below.

1-6-5. Enter **zynq** in the Search field.

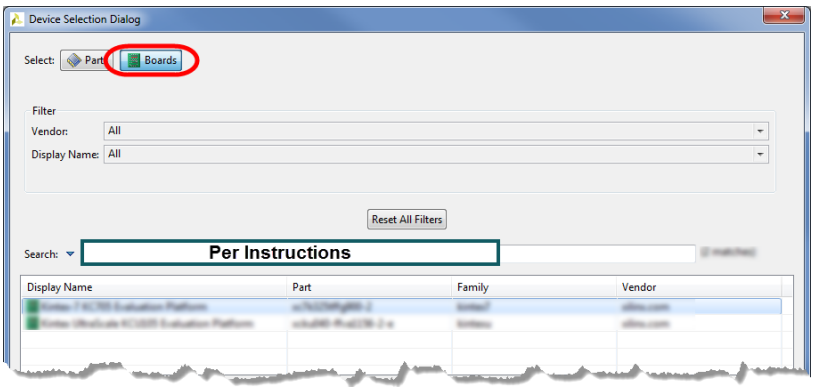


Figure 1-8: Filtering to Quickly Locate Target Platforms

1-6-6. Select **Zynq ZC702 Evaluation Board** from the search list.

1-6-7. Click **OK** to select the board.

1-6-8. Click **Finish**.

You will see the created project in the Explorer tab.

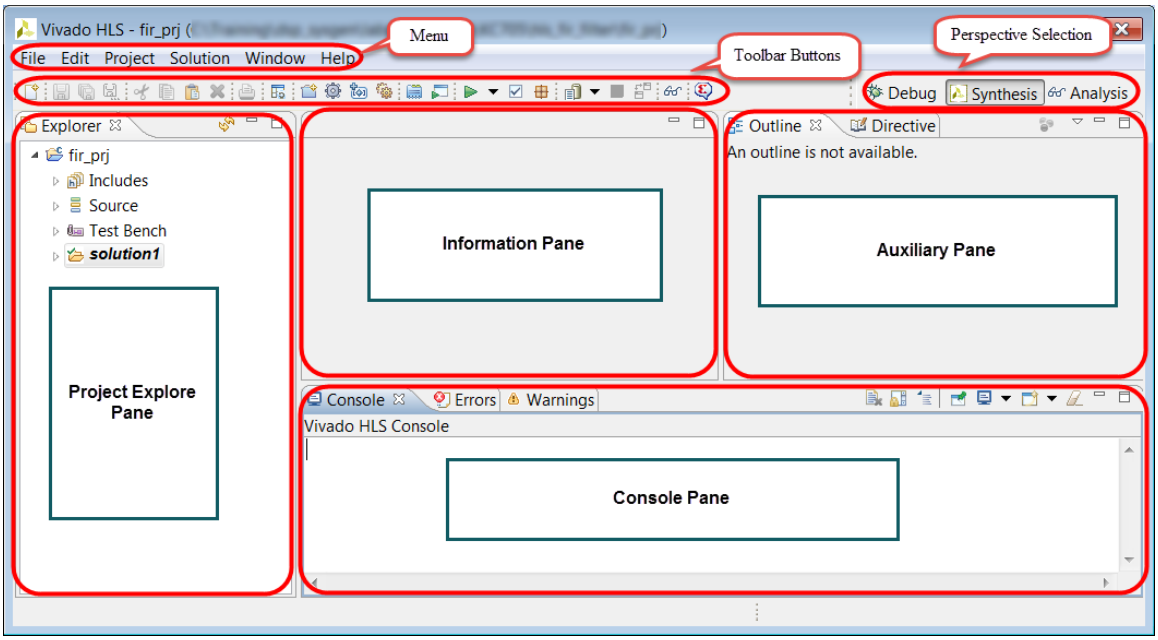


Figure 1-9: Vivado HLS with Newly Created Project

The Vivado HLS tool GUI consists of various panes to proceed with the development work. You will see the created project in the Explorer view. Expand various sub-folders to see the entries under each sub-folder. The Source folder consists of source files associated with the project and the Test Bench folder consists of testbench files associated with the project.

Running C Simulation Step 2

After creating the project, the next step is to validate the design before proceeding with synthesizing the design. In this step, you will validate the design by using the provided self-checking C testbench.

2-1. Become familiar with the provided source and testbench files.

2-1-1. In the Project Explorer pane, expand the **Source** folder.

2-1-2. Double-click **dct.c** to open the file.

This will open the source file in the Information pane.

2-1-3. Become familiar with the code and data structures.

2-1-4. In the Project Explorer pane, expand the **Includes** > **C:\training\hls\labs\hls_tool_flow\zc702\dct** folder.

2-1-5. Double-click **dct.h** to open the header file.

2-1-6. Review the contents of the header file.

2-1-7. In the Project Explorer pane, expand the **Test Bench** folder.

2-1-8. Double-click **dct_test.c** to open it in the Information pane.

This testbench is a self-checking testbench; i.e., the computed output is compared against a reference golden output and returns either pass or fail.

2-2. Simulate the Vivado HLS tool design.

2-2-1. Select **Project** > **Run C Simulation** or click the **Run C Simulation** icon ( ).

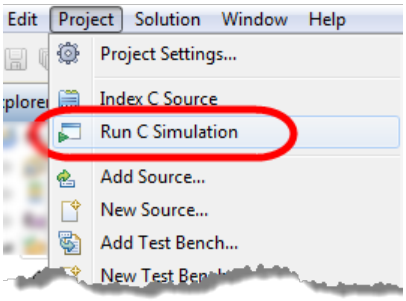


Figure 1-10: Launching the C Simulation

The Run C Simulation dialog box opens.

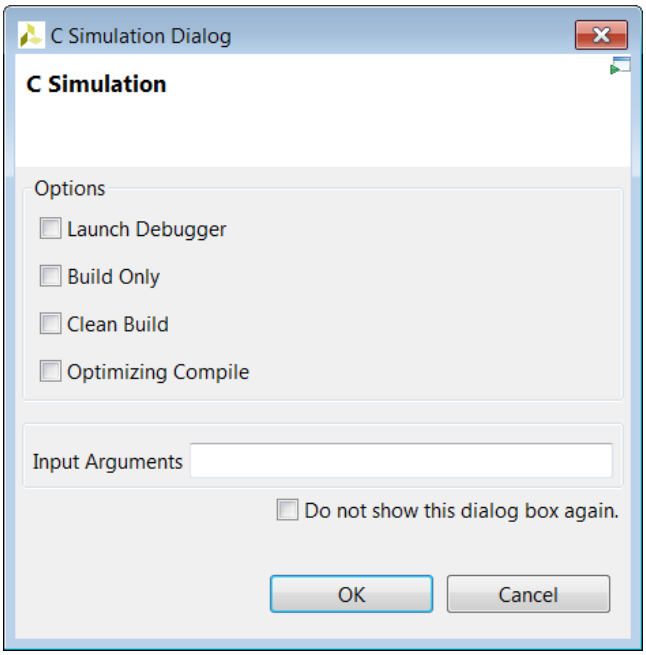


Figure 1-11: C Simulation Dialog Box

Each of the options controls how simulation is run:

- Launch Debugger: After compilation the debug perspective automatically opens for you to step through the code.
- Build Only: Compile the object/netlist files but do not execute.
- Clean Build: Remove previously compiled files before compiling.
- Optimizing Compile: Use the gcc/g++ -O option (no debug info and this is mutually exclusive with debug options; this may run faster, but the difference is not substantial).

2-2-2. Select **Default Options (i.e. select nothing)**.

2-2-3. Click **OK**.

The simulation log will be displayed in the editor pane.

2-3. View the simulation report.

The information generated by the Vivado HLS tool can be found in two places, both described here.

The first is the Console window, which reports not only the output produced by the code being simulated, but all of the simulation engine messages as well. The simulation log provides only a few simulation engine messages and the simulated code output.

2-3-1. Select the **Console** tab in the lower portion of the tool's GUI.

You may need to scroll to view all the output produced by the simulation.

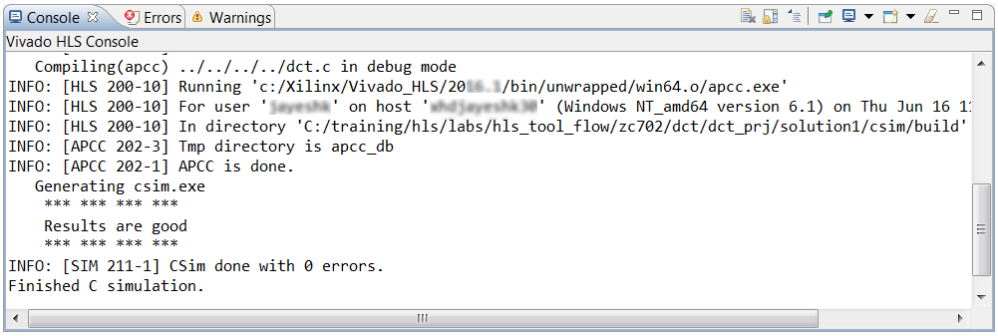


Figure 1-12: Example Output After a Simulation

The other location, described below, provides only a few simulation engine messages and the simulated code output. Typically this is opened after the simulation completes; however, if you need to access it after closing the log pane, here's how to access the simulation report.

2-3-2. Expand **dct_prj** > **solution1** > **csim** > **report** in the Explorer pane.

2-3-3. Double-click the log file name to open it in the editor pane.

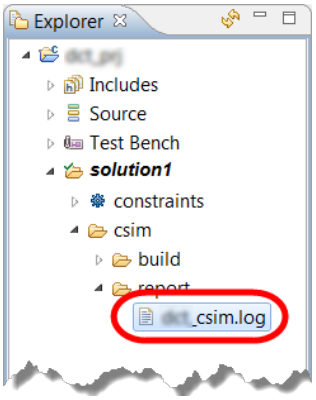


Figure 1-13: Locating the Simulation Log File

You should see the *Results are good* message in the simulation log file and in the console area. If you do not see this message, ask for help from your instructor.

Synthesizing the Design Step 3

In this step, you will synthesize the design by using Vivado HLS tool defaults and analyze how many resources are utilized to implement the C design.

3-1. Synthesize the design.

3-1-1. Select **Solution** > **Run C Synthesis** > **Active Solution** or click the **Run Synthesis** icon in the menu bar.

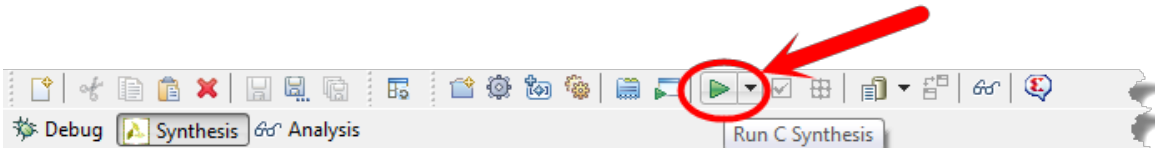


Figure 1-14: Launching Synthesis

This option synthesizes the currently selected solution.

All solutions (or selected solutions) can be synthesized by using the drop-down menu next to the synthesis icon. You can synthesize all solutions or synthesize selected solutions in addition to the default.

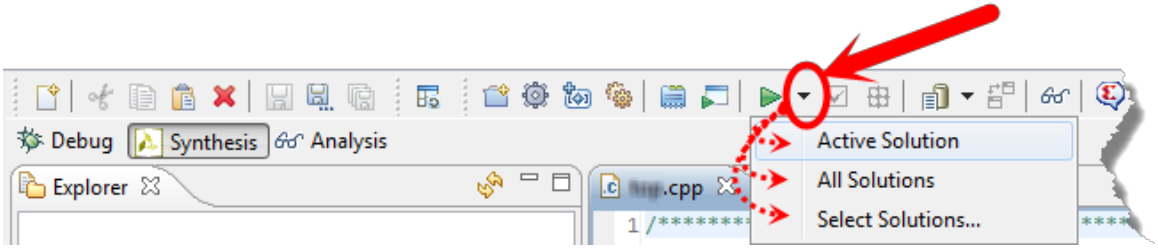


Figure 1-15: Options for What to Synthesize

When the synthesis completes, the Synthesis report will be displayed in the Information pane.

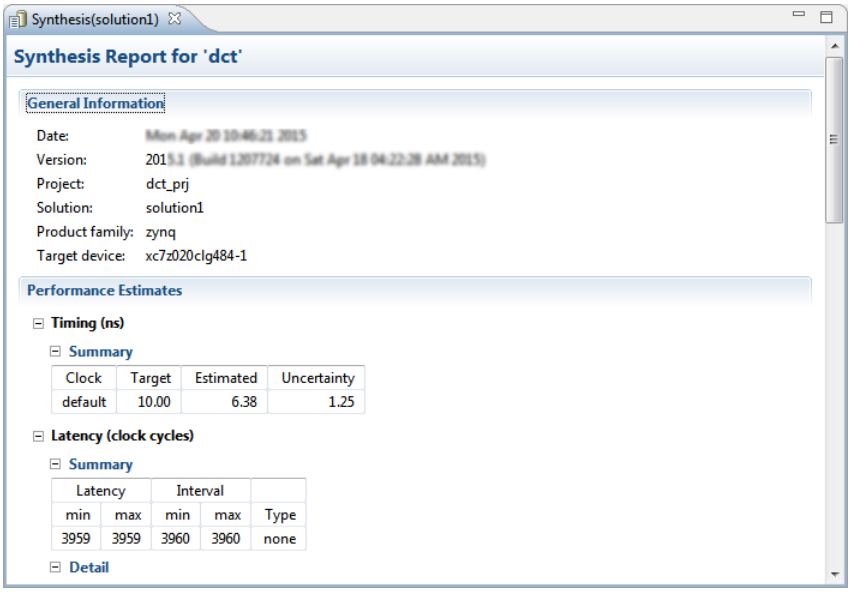


Figure 1-16: Synthesis Report

The Synthesis report shows the performance and area estimates as well as estimated latency in the design.

3-1-2. In the Outline pane, click **Performance Estimates** and **Utilization Estimates** to answer the following question.

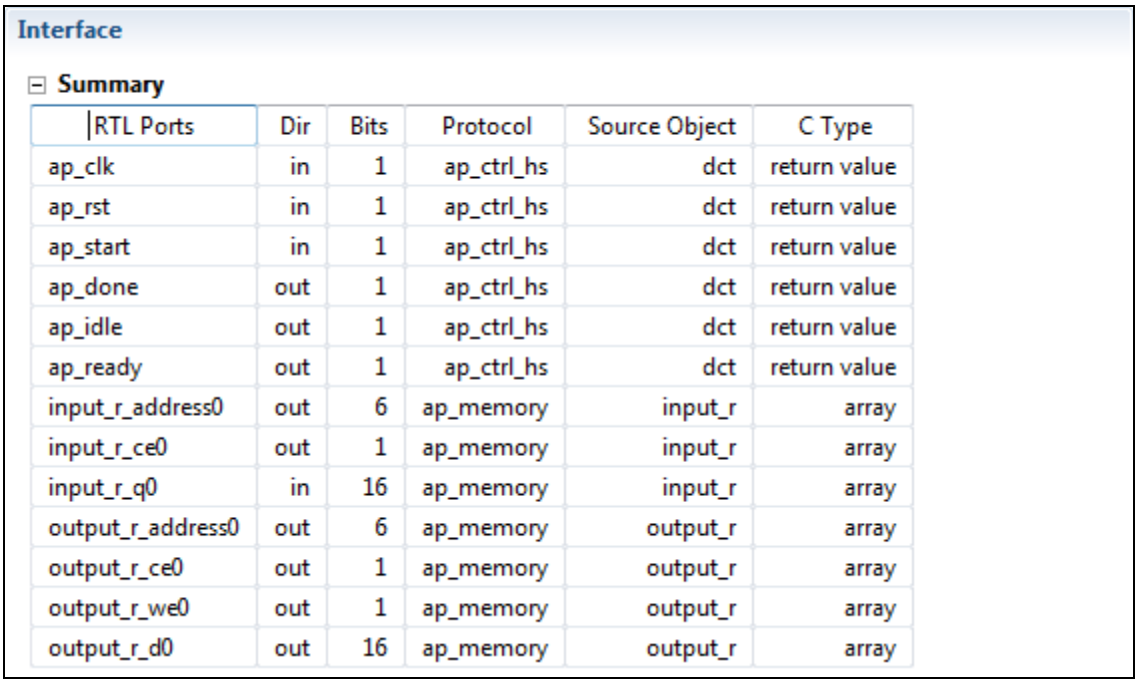
Question 1

Write down the following details from the Synthesis report:

- Estimated clock frequency:
- Worst case latency:
- Number of DSP48E used:
- Number of FFs used:
- Number of LUTs used:

3-1-3. In the Outline pane, select **Interface** > **Summary**.

The report also shows the top-level interface signals generated by the tools.



Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	dct	return value
ap_rst	in	1	ap_ctrl_hs	dct	return value
ap_start	in	1	ap_ctrl_hs	dct	return value
ap_done	out	1	ap_ctrl_hs	dct	return value
ap_idle	out	1	ap_ctrl_hs	dct	return value
ap_ready	out	1	ap_ctrl_hs	dct	return value
input_r_address0	out	6	ap_memory	input_r	array
input_r_ce0	out	1	ap_memory	input_r	array
input_r_q0	in	16	ap_memory	input_r	array
output_r_address0	out	6	ap_memory	output_r	array
output_r_ce0	out	1	ap_memory	output_r	array
output_r_we0	out	1	ap_memory	output_r	array
output_r_d0	out	16	ap_memory	output_r	array

Figure 1-17: Generated Interface Signals

You can see that *ap_clk* and *ap_rst* are automatically added. *ap_start*, *ap_done*, and *ap_idle* are top-level signals used as handshaking signals to indicate when the design is able to accept the next computation command (*ap_idle*), when the next computation is started (*ap_start*), and when the computation is completed (*ap_done*). Other signals are generated based on the design itself.

3-1-4. Select the **Console** tab.

The Synthesis log is available in the Vivado HLS Console.

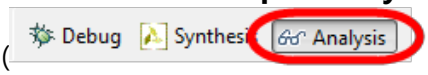
Note that when the *Solution1* > *Syn* folder is expanded in the Explorer view, it will show the *report*, *systemc*, *verilog*, and *vhdl* sub-folders under which the report files and generated source files (VHDL, Verilog, header, and cpp) are available. Double-clicking any of these entries will open the corresponding file in the Information pane.

Also note that the target design has hierarchical functions, and reports corresponding to lower-level functions are also created (in this example *dct_dct_1d2_csynth.rpt* and *dct_dct2d_csynth.rpt* in addition to **dct_csynth.rpt**). By default, the report for the top-level function is displayed in the Information pane once synthesis is completed.

Analyzing the Design Using the Analysis Perspective Step 4

The Analysis perspective is used after synthesis completes for analyzing the design in detail. This perspective provides considerable more details than the synthesis report.

4-1. Switch to the Analysis perspective and understand the design behavior.

4-1-1. Select **Solution** > **Open Analysis Perspective** or click the **Analysis** perspective icon (

) to open the analysis viewer.

The Analysis perspective consists of five panes as shown below. Note that the module and loops hierarchies are displayed unexpanded by default.

The Module Hierarchy pane provides an overview of the entire RTL design:

- This view can navigate throughout the design hierarchy.
- The Module Hierarchy pane shows the resources and latency contribution for each block in the RTL hierarchy.

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 synthesis report.

The Performance view [Performance(solution1)] is also shown in the right-hand side pane. This view shows how the operations in this particular block are scheduled into clock cycles.

- The left-hand column lists the resources.
- The top row lists the control states (C0 to C5) in the design. Control states are the internal states used by the Vivado HLS tool to schedule operations into clock cycles. There is a close correlation between the control states in the RTL FSM, but there is no one-to-one mapping.

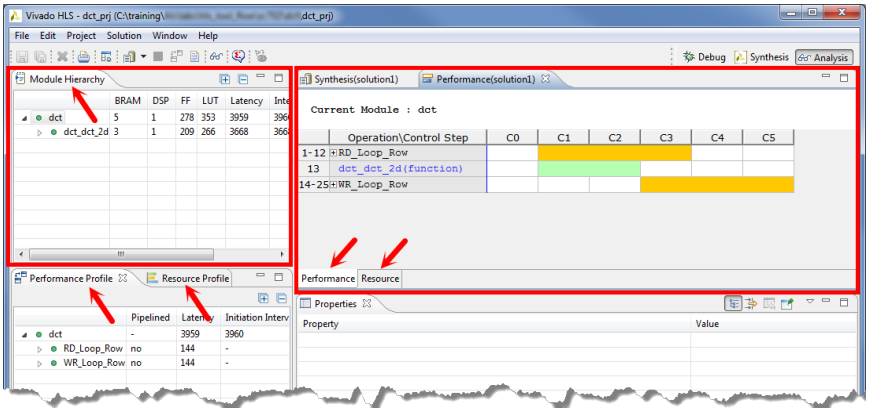


Figure 1-18: Analysis Perspective Panes

4-2. Analyze the performance of the dct module.

4-2-1. Click the loop **RD_Loop_Row** and click the sub-loop **RD_Loop_Col**.

4-2-2. Similarly, click **WR_Loop_Row** and click the sub-loop **WR_Loop_Col**.

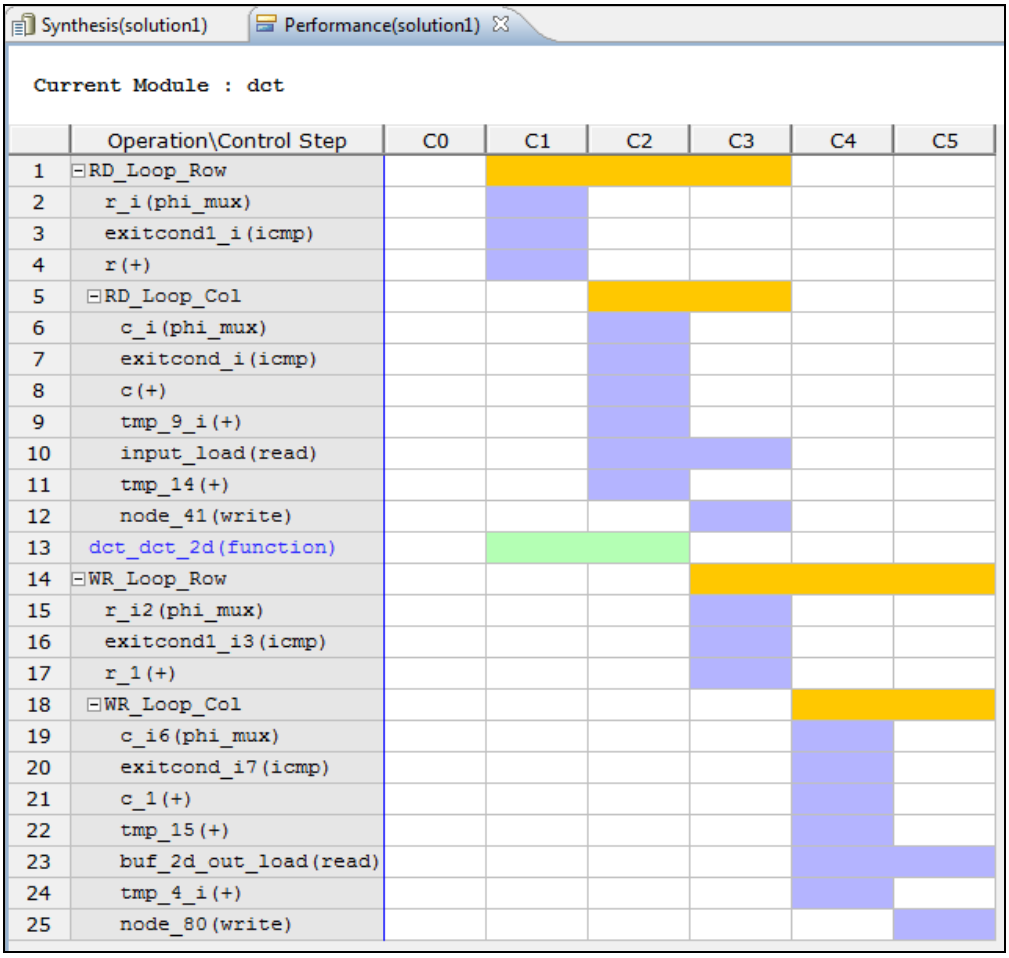


Figure 1-19: Performance of the dct Loop Operation

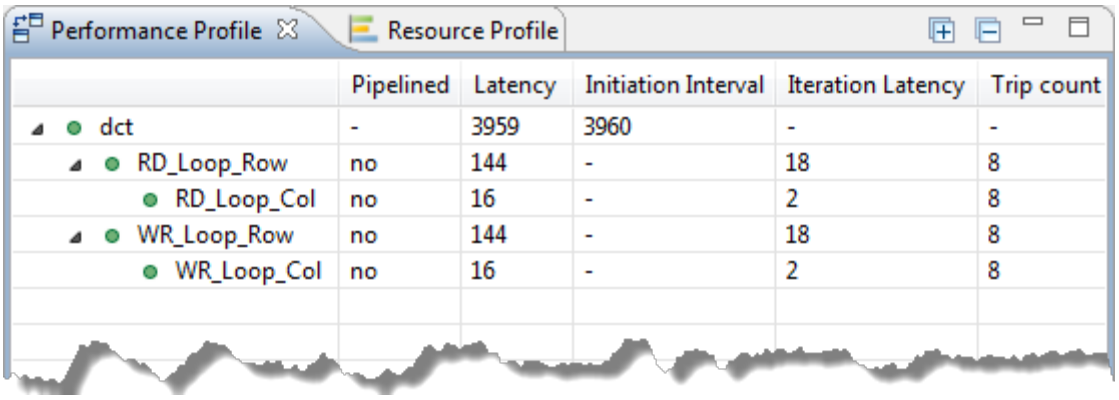
The dct module has three main resources:

- A loop called **RD_Loop_Row**.
- A sub-block called **dct_2d**.
- A loop called **WR_Loop_Row**.

The information presented in the Schedule view is explained below by reviewing the first set of resources to be execute: the **RD_Loop_Row** loop:

- The design starts in the C0 state.
- It then starts to execute the logic in the loop **RD_Loop_Row**.
  - **Note:** In the first state of the loop, the exit condition is checked and there is an add operation.
- The loop executes over three states: C1, C2 and C3.

4-2-3. Review the Performance Profile pane information.



	Pipelined	Latency	Initiation Interval	Iteration Latency	Trip count
dct	-	3959	3960	-	-
RD_Loop_Row	no	144	-	18	8
RD_Loop_Col	no	16	-	2	8
WR_Loop_Row	no	144	-	18	8
WR_Loop_Col	no	16	-	2	8

Figure 1-20: Performance Profile Pane

The Performance Profile pane shows this loop has a tripcount of eight. It therefore iterates around these three states eight times.

The Performance Profile pane shows that the loop **RD_Loop_Row** takes 144 clock cycles to execute:

- One cycle at the start of the loop *RD_Loop_Row*.
- It takes **16** clock cycles to execute all operations of the loop **RD_Loop_Col**.
- Plus a clock cycle to return to the start of the loop **RD_Loop_Row** for a total of **18** cycles per loop iteration.
- Eight iterations of 18 cycles is why it takes 144 clock cycles to complete.

Within the loop **RD_Loop_Col** you can see there are some adders, a two-cycle read operation and a write operation.

4-2-4. Select the purple color block for the tmp_14(+) operation (line no. 11) as shown below.

4-2-5. Right-click and select **Goto Source**.

Alternatively, you can double-click the purple to open the source code.

4-2-6. Select **read_data** at line no. 62 and click **OK**.

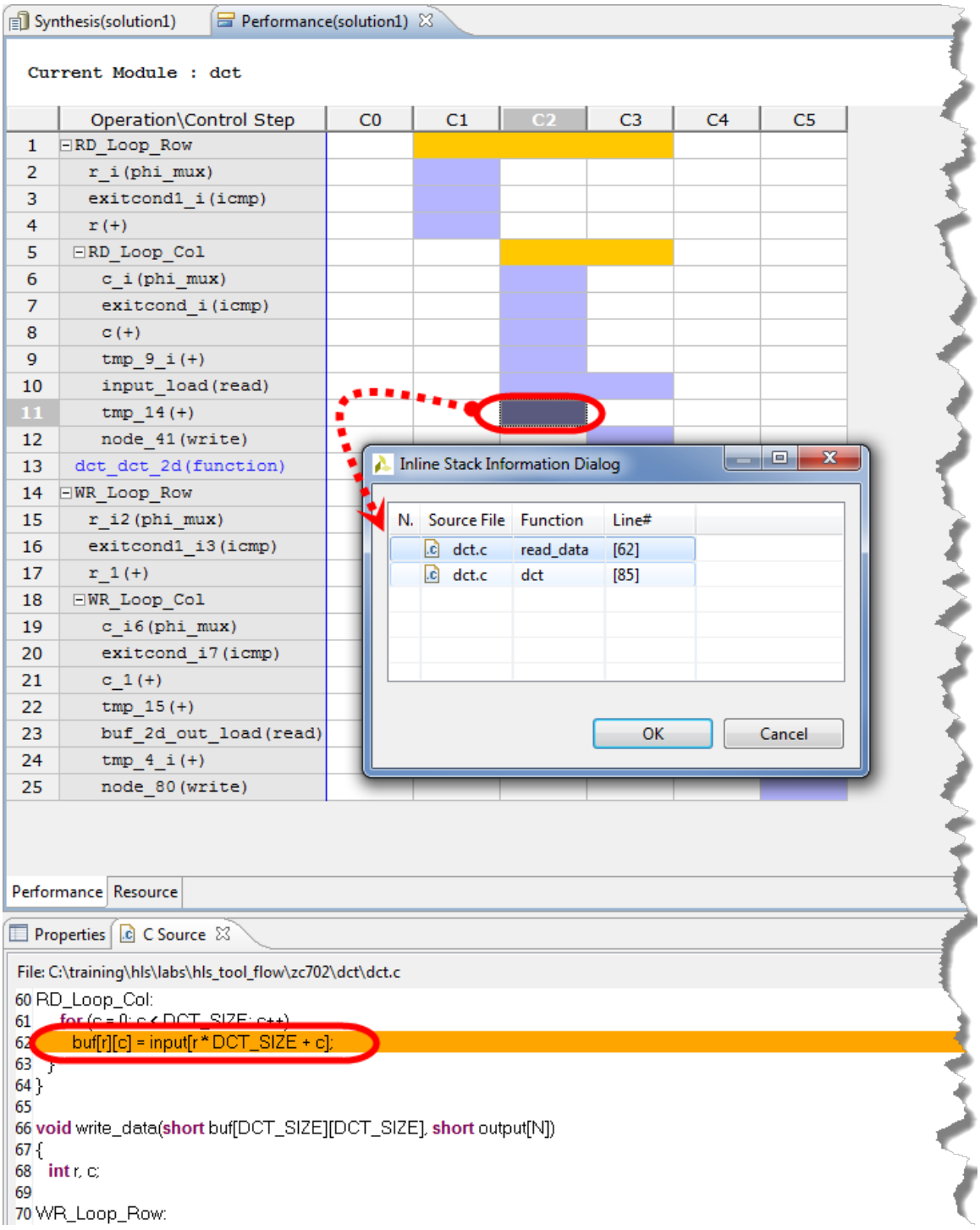


Figure 1-21: C Source Code Correlation

You can see that the write operation is implementing the writing of data into the buf array from the input array variable.

4-3. Analyze the resource usage for the design.

4-3-1. Select the **Resource Profile** pane and the **Resource** tab as shown below.

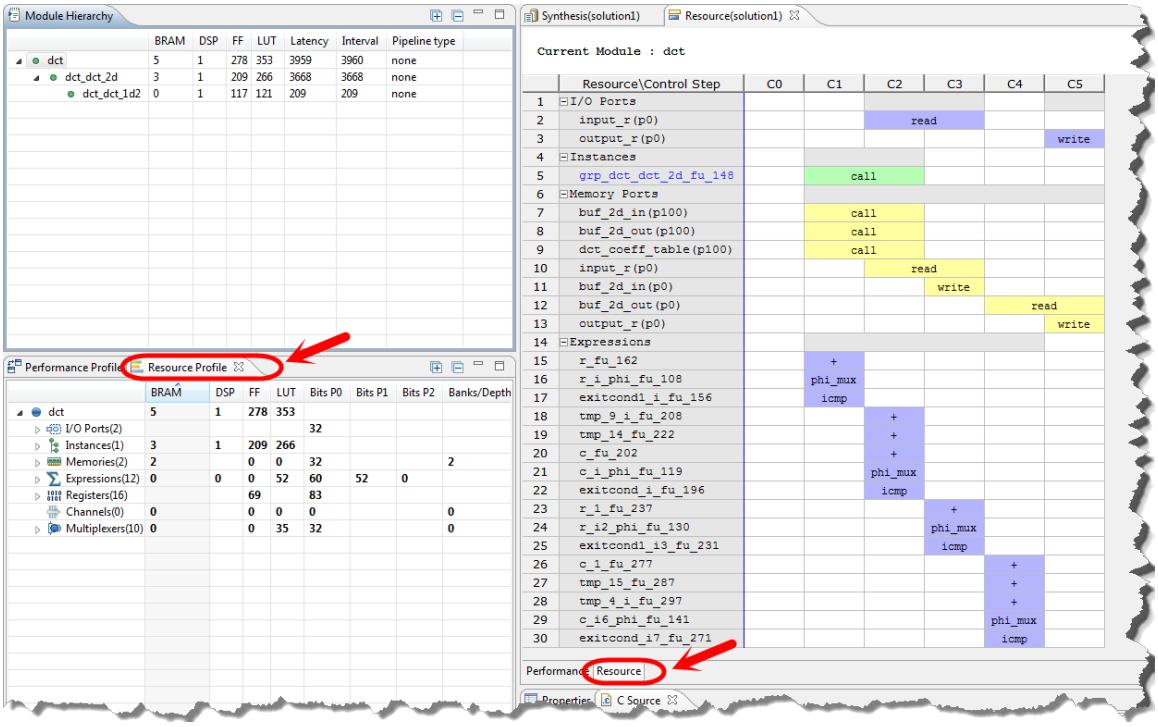


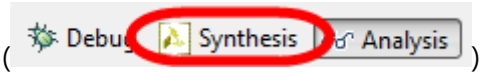
Figure 1-22: Analysis Perspective – Resource Usage

The Resource Profile pane shows the resources used at this level of hierarchy. In this design, you can see that the most of the resources are due to the instances—blocks that are instantiated inside this block.

By expanding the **Expressions** category, you can see that most of the resources at this level of hierarchy are used to implement adders.

The Resource pane shows the control state of the operations used. In this design, all the adder operations are associated with a different adder resource. There is no sharing of the adders. More than one add operation on each horizontal line indicates that the same resource is used multiple times in different states or clock cycles.

The adders are used in the same cycles that are memory accessed and are dedicated to each memory. Cross correlation with the C code can be used to confirm.

4-3-2. Click the **Synthesis** perspective icon (

) to return to the synthesis view.

Performing C/RTL Co-simulation Step 5

Now that you have performed high-level synthesis on the C design, you will perform RTL co-simulation on the generated RTL using the C testbench.

Run C/RTL co-simulation, selecting Verilog and skipping VHDL. Verify that the simulation passes.

5-1. Cosimulate the Vivado HLS tool design.

5-1-1. Select **Solution** > **Run C/RTL Cosimulation** or click the **Run C/RTL Cosimulation** icon ( ).

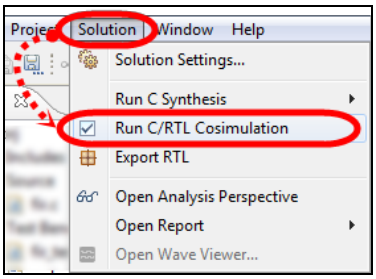


Figure 1-23: Launching from the Menu

The Run C/RTL Co-simulation dialog box opens.

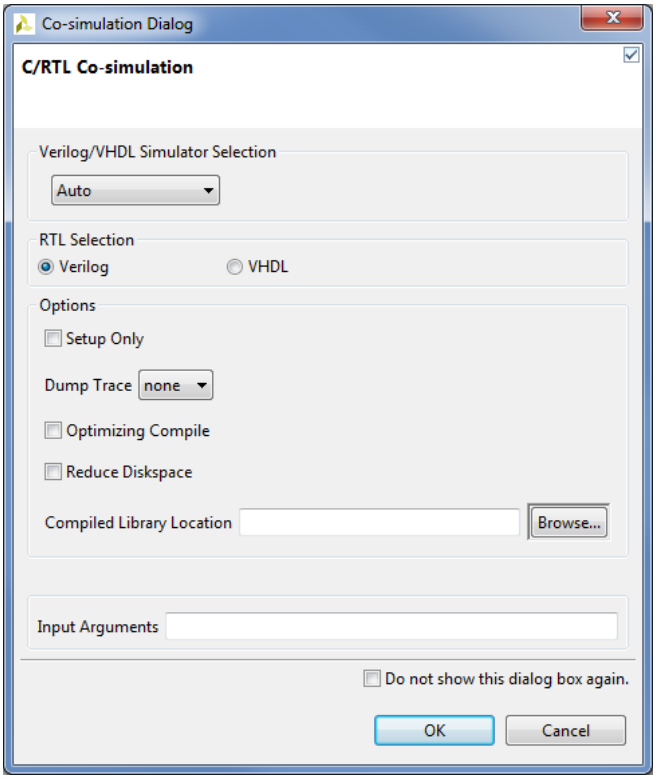


Figure 1-24: Co-simulation Dialog Box

Each of the options controls how C/RTL Cosimulation is run:

- RTL Selection: Select the RTL that is simulated (Verilog/VHDL).
- Setup Only: This creates all the files (wrappers, adapters and scripts) required to run the simulation but does not execute the simulator.
- Dump Trace: During RTL verification, the trace files can be saved and viewed using an appropriate viewer. By selecting this option, the trace file will be saved to the *<solution>/sim/<RTL>* folder.
- Optimizing Compile: This ensures a high level of optimization is used to compile the C testbench. Using this option increases the compile time but the simulation executes faster.

5-1-2. Select **Default Options (i.e. select nothing)**.

5-1-3. Click **OK**.

The simulation log will be displayed in the editor pane.

5-2. View the Cosimulation report.

The information generated by the Vivado HLS tool can be found in two places, both described here.

The first is the Console window, which reports not only the output produced by the code being simulated, but all of the simulation engine messages as well. The simulation log provides only a few simulation engine messages and the simulated code output.

5-2-1. Select the **Console** tab in the lower portion of the tool's GUI.

You may need to scroll to view all the output produced by the cosimulation.

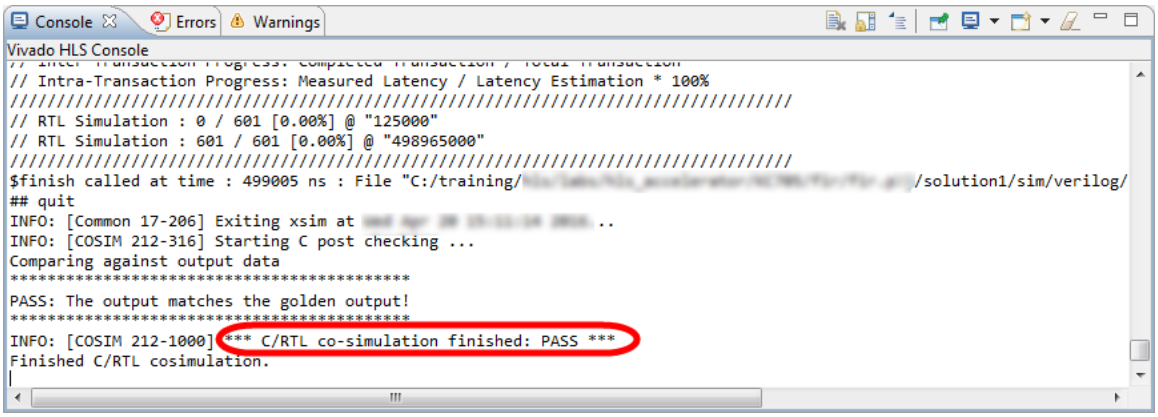


Figure 1-25: Example Output After a C/RTL Co-simulation

The other location, described below, provides only a few simulation engine messages and the simulated code output. Typically this is opened after the simulation completes; however, if you need to access it after closing the log pane, here's how to access the simulation report.

5-2-2. Expand **dct_prj** > **solution1** > **sim** > **report** in the Explorer pane.

5-2-3. Double-click the log file name to open it in the editor pane.

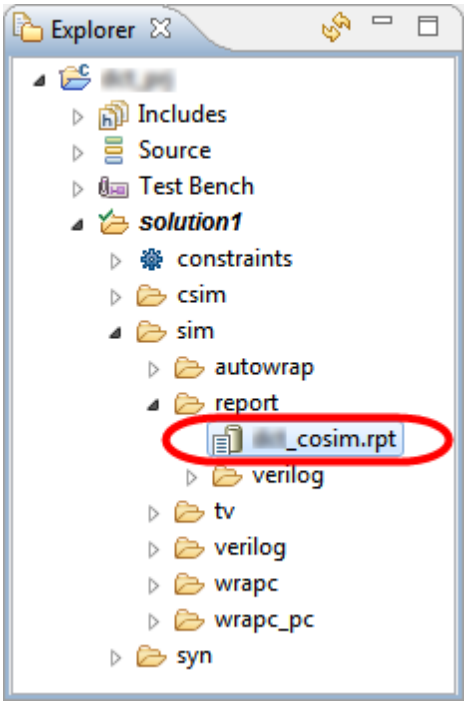


Figure 1-26: Locating the Co-Simulation Log File

The Cosimulation Report in HTML format will be displayed in the main viewing area.

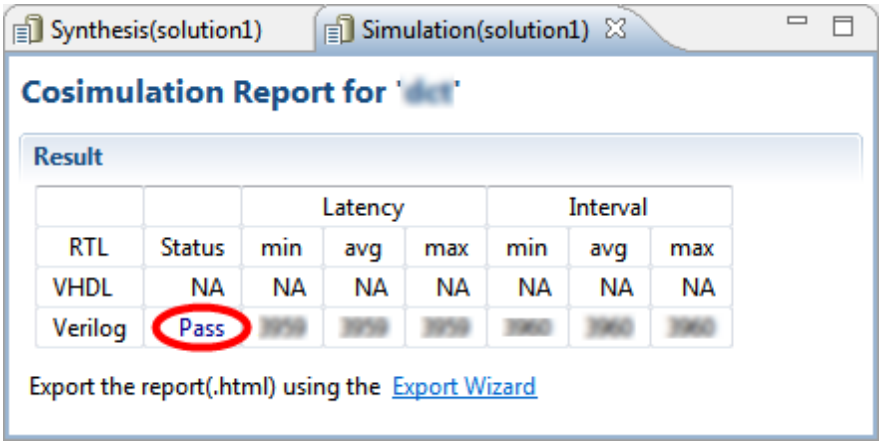


Figure 1-27: Cosimulation Report – HTML

5-2-4. You can quickly verify the cosimulation status here.

This process will take a few minutes to complete. After C/RTL cosimulation has been completed, the Cosimulation report will be accessible in the Information pane, including the latency information.

Also, in the Console tab, notice the *Results are Good* message that is displayed.

Exporting the RTL as an IP Core Step 6

In this step you will export the RTL as an IP core to be used with the top-level design.

6-1. Export the RTL, selecting Verilog as the language.

6-1-1. Select **Solution** > **Export RTL**.

Alternatively, you can click the  toolbar button.

6-1-2. Ensure that **IP Catalog** is selected from the Format Selection drop-down list.

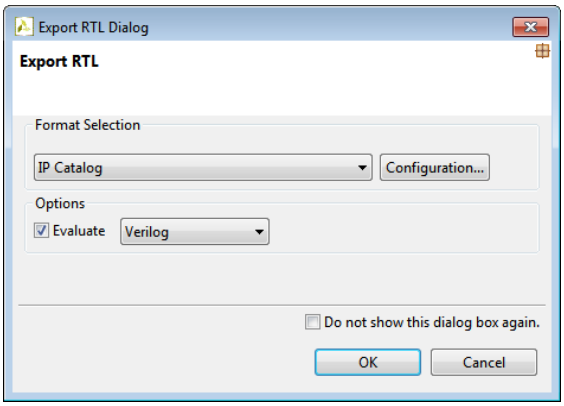


Figure 1-28: Export RTL Dialog Box

6-1-3. Click **Configuration** next to the Format Selection drop-down list.

Notice that you can provide information about the IP, such as the vendor, library, version, and description in the IP Identification dialog box.

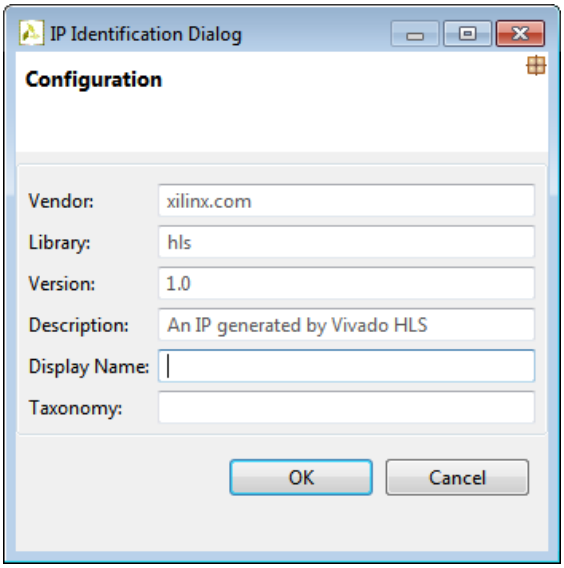


Figure 1-29: IP Identification Dialog Box

6-1-4. Click **Cancel** in the IP Identification dialog box.

6-1-5. Make certain that the **Evaluate** option is selected and that **Verilog** is selected as the RTL to be evaluated.

This will perform Vivado RTL synthesis and implementation on the generated IP. Implementation is run to evaluate and provide confidence that the RTL will meet its estimated timing and area goals and that these results are not included as part of the exported package.

6-1-6. Click **OK** in the Export RTL dialog box.

You can observe the progress in the Console tab. When the run is complete, the Implementation Report is displayed in the Information pane.

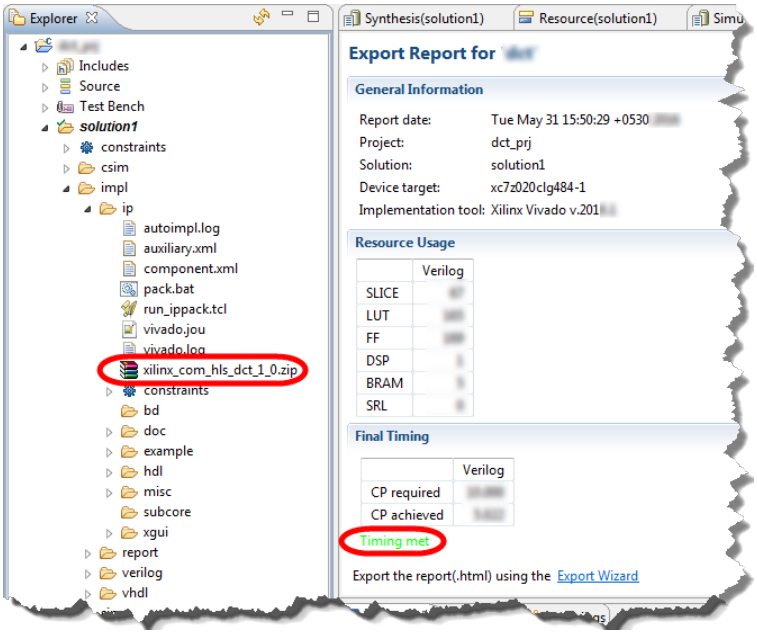


Figure 1-30: Implementation Results in the Vivado HLS Tool

Once implementation completes, the Implementation report will open in the Information pane. The final timing of the implemented design has been achieved.

The exported IP is available in the *<solution_directory>\impl\ip* folder.

6-1-7. Select **File** > **Exit** to close the Vivado HLS tool.

Summary

In this lab, you learned how to create a new Vivado HLS tool project in the GUI and execute major steps (simulate, synthesize, co-simulate and export) in the HLS Vivado Design Suite flow.

In the following labs, you will examine some of the software reports, determine how the design was implemented, and determine whether or not design goals for area and performance were met.

Answers

1. Write down the following details from the Synthesis report:

Estimated clock frequency:	6.38
Worst case latency:	3959
Number of DSP48E used:	1
Number of FFs used:	278
Number of LUTs used:	353

Lab 2: Introduction to the Vivado HLS Tool CLI Flow

2016.1

Abstract

This lab introduces how to perform basic actions in the Vivado® HLS command prompt.

This lab should take approximately 30 minutes.

Objectives

After completing this lab, you will be able to:

- Create a new project in the Vivado HLS tool CLI
- Simulate a C design by using a self-checking testbench
- Synthesize the design
- Simulate an RTL design by using a C testbench
- Implement the design

Introduction

This lab provides an introduction to the major features of the Vivado® High-Level Synthesis (HLS) tool Command Line Interface (CLI) flow. You will use the Vivado HLS tool in CLI mode to create a project. You will also simulate, synthesize, and implement the design provided.

In this lab, you will be using a C design to implement a DCT. The function implements a 2D DCT algorithm by first processing each row of the input array via a 1D DCT, then processing the columns of the resulting array through the same 1D DCT. It calls the read_data, dct_2d, and write_data functions.

The read_data function is defined at line 54 and consists of two loops: RD_Loop_Row and RD_Loop_Col. The write_data function is defined at line 66 and consists of two loops to perform writing the result. The dct_2d function, defined at line 23, calls the dct_1d function and performs transpose.

Finally, the dct_1d function, defined at line 4, uses the dct_coeff_table and performs the required function by implementing a basic iterative form of the 1D Type-II DCT algorithm.

The following figure shows the function hierarchy on the left-hand side, the loops in the order they are executed, and the flow of data on the right-hand side.

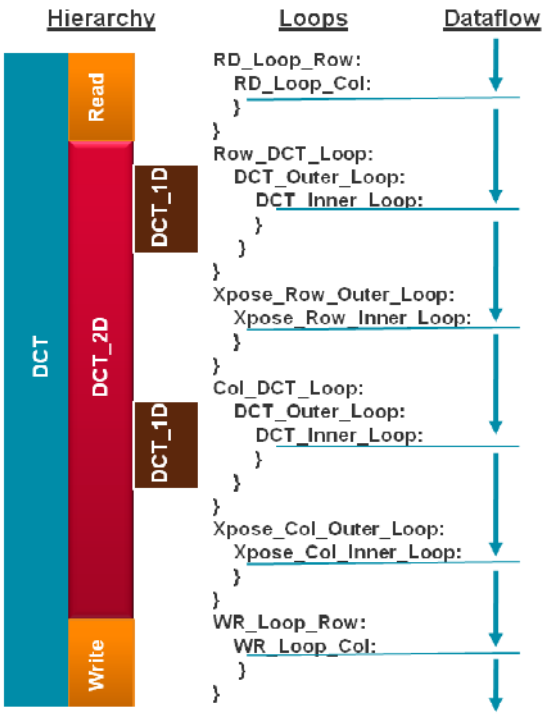
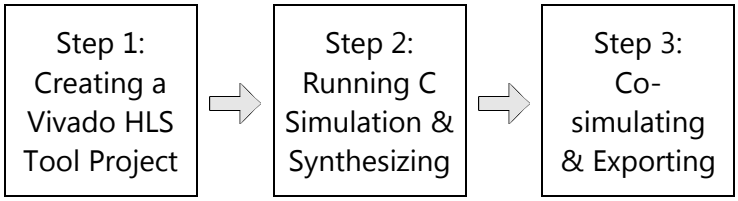


Figure 2-1: Design Hierarchy and Dataflow

General Flow



Creating a Vivado HLS Tool Project Step 1

In this step, you will create a new Vivado HLS tool project, add source files, and provide solution settings for the default solution using the Vivado HLS tool command prompt. Later, you will open the created project in the Vivado HLS tool GUI to have a quick review of the settings were applied.

1-1. Launch the Vivado HLS tool command prompt and change the directory to the *hls_cli_flow* working directory.

1-1-1. Select **Start** > **All Programs** > **Xilinx Design Tools** > **Vivado 2016.1** > **Vivado HLS** > **Vivado HLS 2016.1 Command Prompt** to launch the Vivado HLS tool command prompt.

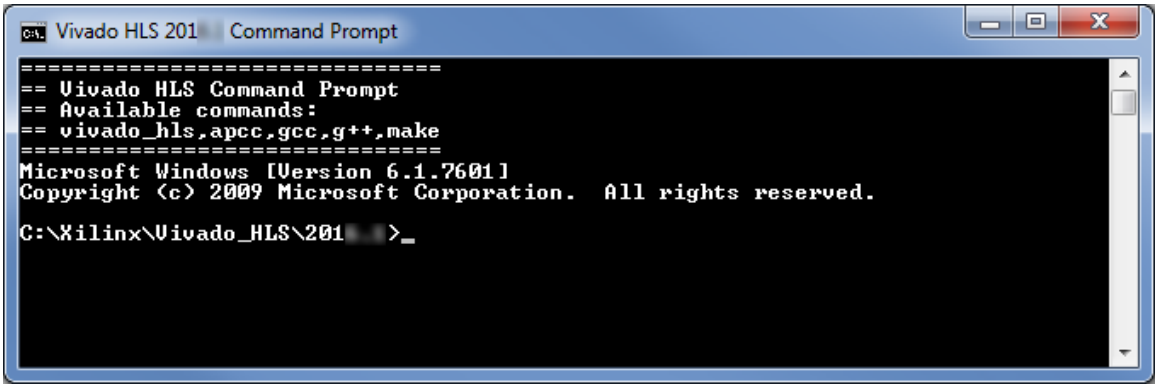


Figure 2-2: Vivado HLS Command Prompt

1-1-2. Enter the following command to change the directory to the current working directory:

```
cd C:\training\hls\labs\hls_cli_flow\dct\kintex7
```

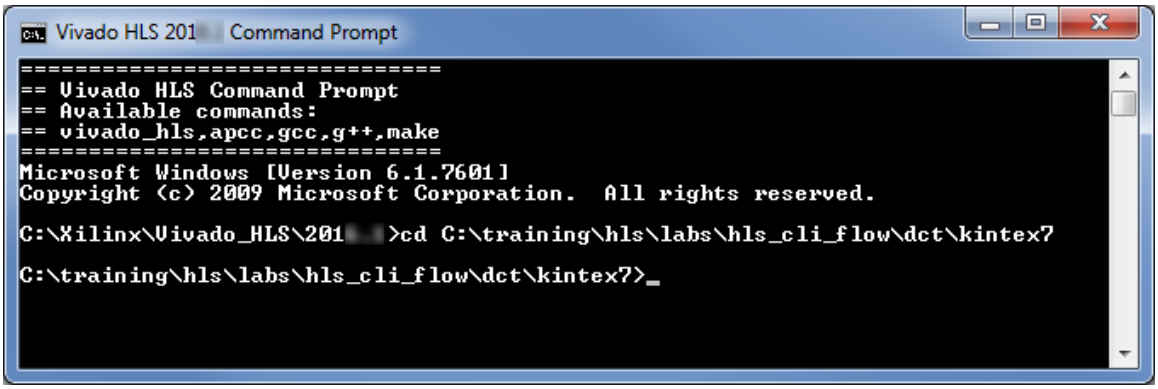


Figure 2-3: Changing the Directory

1-2. Write the commands to create a new project, associate source files, and configure solution settings.

1-2-1. Using Windows Explorer, browse to the *C:\training\hls\labs\hls_cli_flow\dct\kintex7* directory and open **dct.tcl** with your preferred text editor.

1-2-2. Insert the following command at line no. 2 of *dct.tcl* to create a new project named *dct_prj*:

```
open_project –reset dct_prj
```

Question 1

What does the –reset switch do in the previous command?

1-2-3. Enter the following command at line no. 4 to specify *dct* as a top-level function to be synthesized in the design:

```
set_top dct
```

1-2-4. Enter the following command to add *dct.c* as a design source file to the project:

```
add_files dct.c
```

Question 2

How does the above command differ from adding testbench files to the project?

1-2-5. Enter the following commands to associate *dct_test.c*, *in.dat*, and *out.golden.dat* as testbench source files to the project:

```
add_files –tb dct_test.c
add_files –tb in.dat
add_files –tb out.golden.dat
```

1-2-6. Enter the following command at line no.12 to create the solution to the project named *solution1*:

```
open_solution solution1 -reset
```

1-2-7. Insert the command at line no.14 to associate the xc7k70tfbg484-2 device to the solution:

```
set_part {xc7k70tfbg484-2}
```

1-2-8. Insert the command at line no.16 to associate the clock with a 10ns period to solution1:

```
create_clock -period 10ns
```

After writing the above commands, the *dct.tcl* file should look like the figure below.

dct.tcl

```
1  # Insert the command to create new project
2  open_project -reset dct_prj
3  # Insert the command to specify the top-level function
4  set_top dct
5  # Insert the command to add design file named dct.c
6  add_files dct.c
7  # Insert the command to add testbench files named dct_test.c, in.dat and out.golden.dat
8  add_files -tb dct_test.c
9  add_files -tb in.dat
10 add_files -tb out.golden.dat
11 # Insert the command to create the solution named solution1
12 open_solution solution1 -reset
13 # Insert the command to associate xc7k70tfbg484-2 device to the solution1
14 set_part {xc7k70tfbg484-2}
15 # Insert the command to associate clock with 10ns period to the solution1
16 create_clock -period 10ns
17 # Insert the comamnd to run C simulaiton
18
19 # Insert the comamnd to Synthesize the design
20
21 # Insert the comamnd to perform C/RTL Cosimmulation
22
23 # Insert the comamnd to Export RTL as IP
```

Figure 2-4: dct.tcl with Project Information

1-2-9. Save the **dct.tcl** file.

1-2-10. Exit the text editor to close the Tcl file.

1-3. Run the *dct.tcl* file to create the project and open the created project in the Vivado HLS tool GUI.

1-3-1. In the Vivado HLS tool command prompt, enter the following command to run the *dct.tcl* file:

```
vivado_hls -f dct.tcl
```

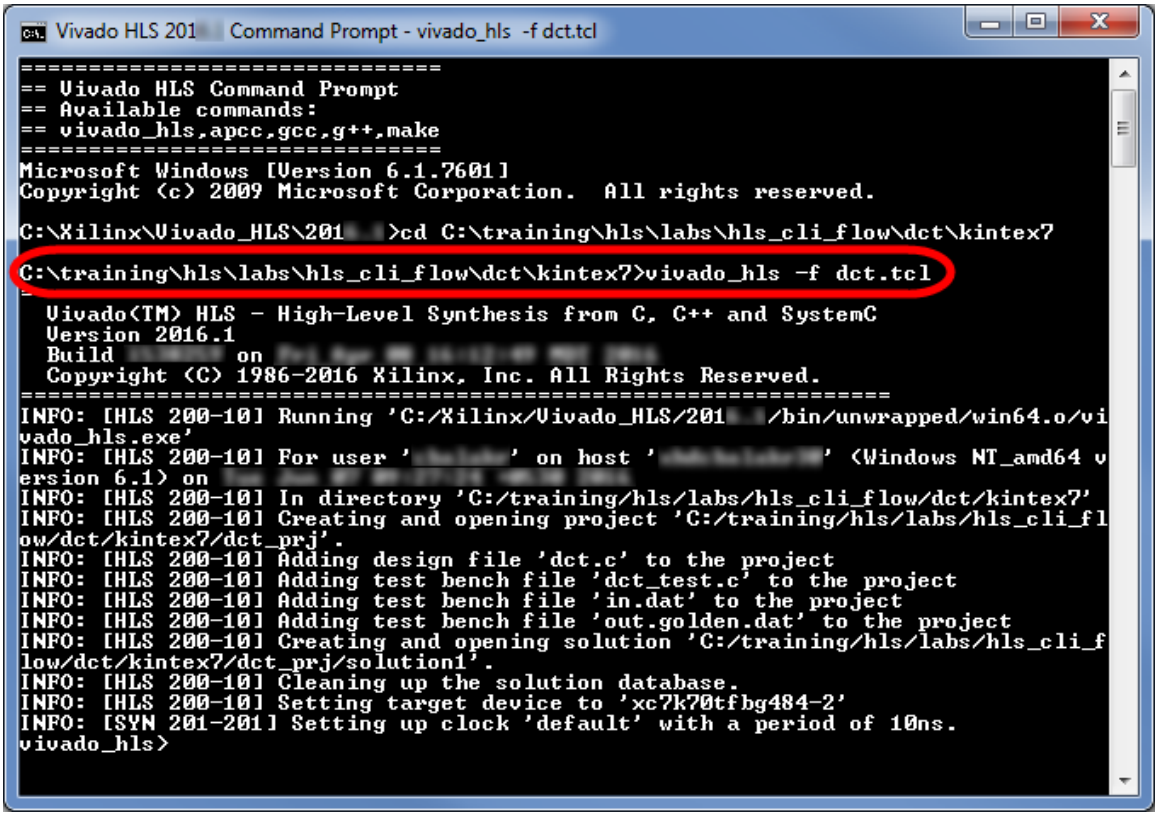


Figure 2-5: Creating the Project

1-3-2. Enter the following command to launch the GUI with the recently created project:

```
vivado_hls -p dct_prj
```

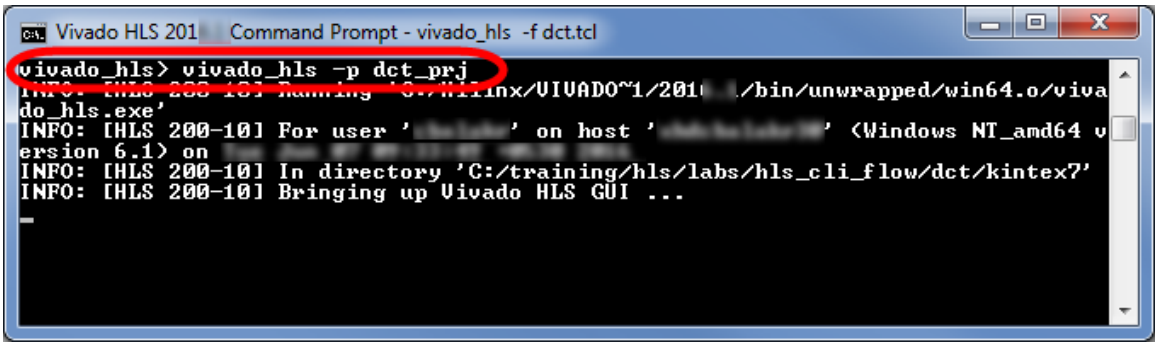


Figure 2-6: Launching the GUI from the CLI

You should see the created project in the Explorer view.

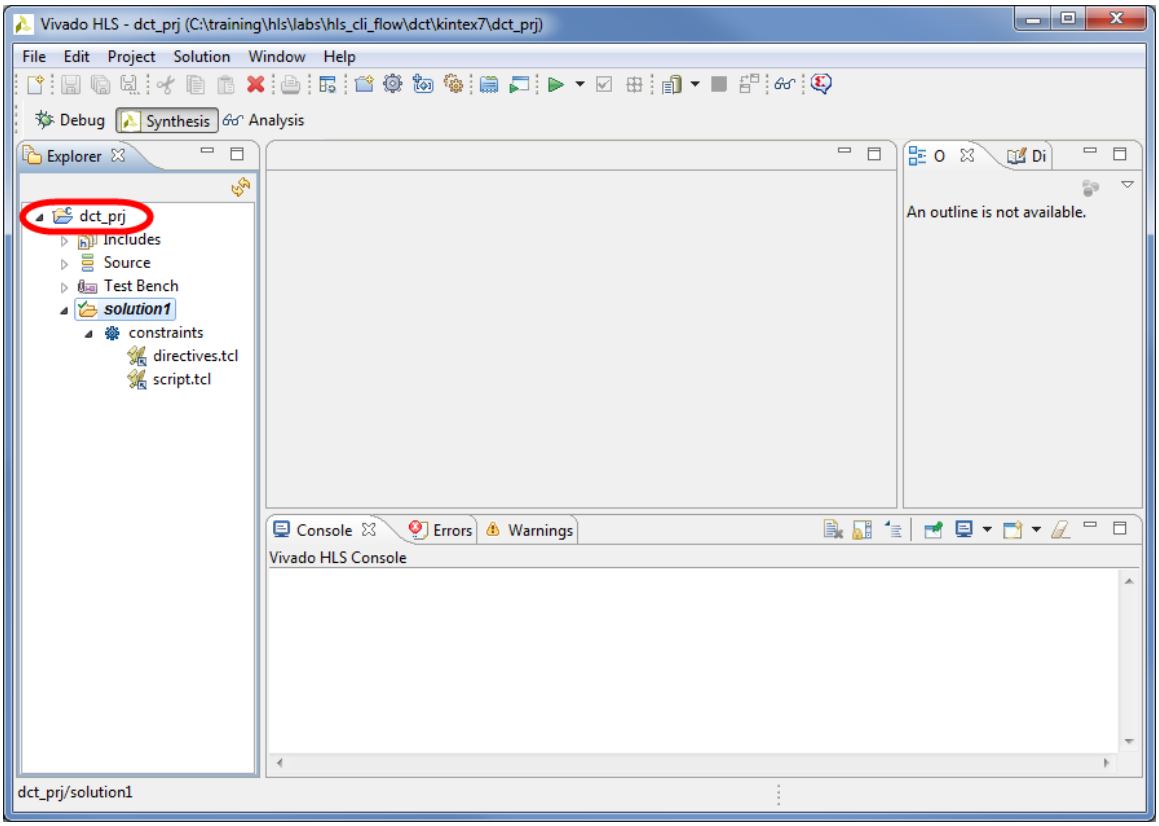


Figure 2-7: Vivado HLS GUI with Recently Created Project

1-3-3. In the Vivado HLS tool GUI, select **Project** > **Project Settings**.

The Project Settings dialog box opens.

1-3-4. Select **Simulation** and **Synthesis** to ensure that the design and testbench sources were added to the project as entered in the Tcl file.

1-3-5. Close the Project Settings dialog box once the source files are verified.

1-3-6. In the Vivado HLS tool GUI, select **Solution** > **Solution Settings**.

The Solution Settings dialog box for the active solution, i.e. *Solution1*, the only solution that exists in this current project, opens.

1-3-7. Select **Synthesis** to ensure that the clock frequency and devices settings were the ones described in the Tcl file.

1-3-8. Close the Solution Settings dialog box once the clock frequency and device settings are verified.

1-3-9. In the Vivado HLS tool GUI, select **File** > **Exit** to exit the GUI.

Running C Simulation and Synthesizing the Design Step 2

In this step, you will run C simulation and synthesize the design from the Vivado HLS tool command prompt and then open the Vivado HLS tool GUI to review the project status and Synthesis report.

2-1. Simulate the design.

2-1-1. In the Vivado HLS tool command prompt, enter the following command to run C simulation:

```
csim_design
```

This command compiles and runs pre-synthesis C simulation of the design using the provided C testbench.

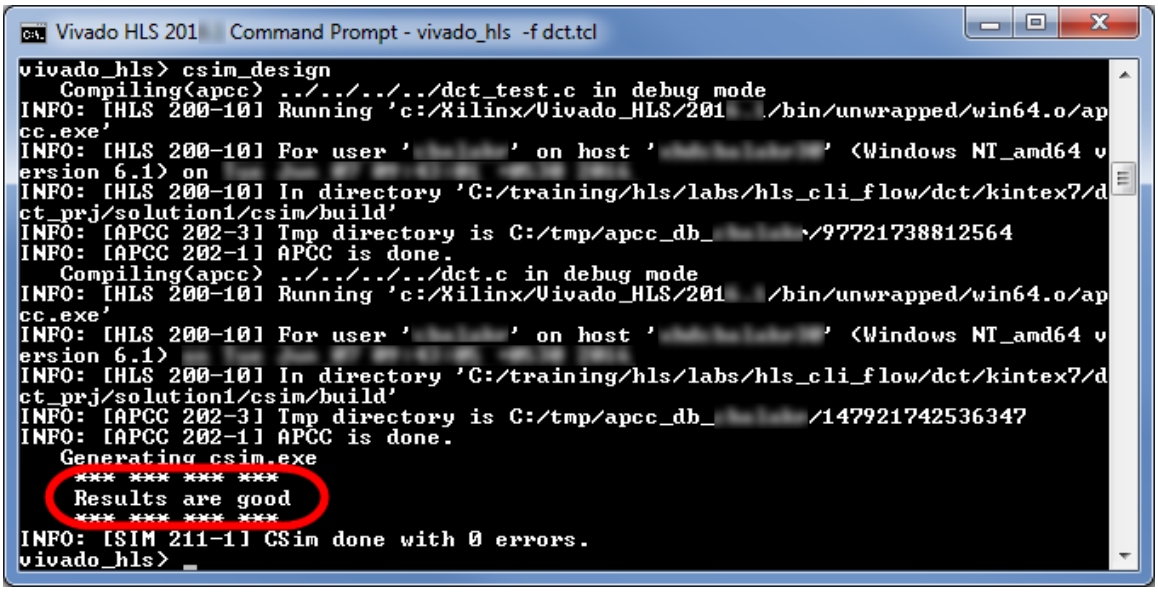


Figure 2-8: Running C Simulation from the Command Prompt

2-2. Synthesize the design.

2-2-1. In the Vivado HLS tool command prompt, enter following command to synthesize the design:

```
csynth_design
```

This will elaborate and perform high-level synthesis on the source files added to the project. This also analyzes the sources files, validates the directives if they are present, and performs some initial code transformations.

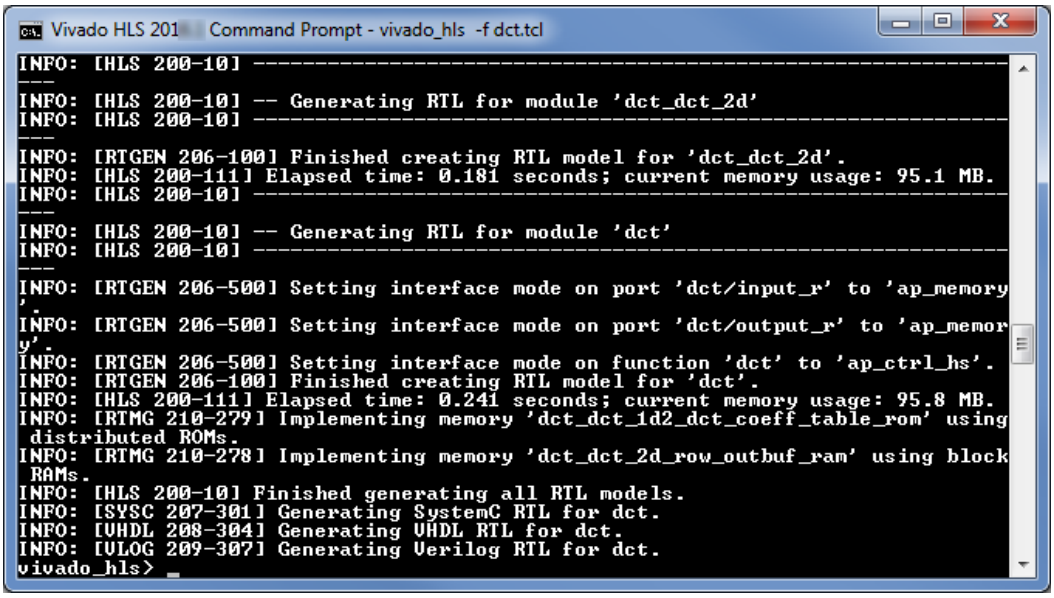


Figure 2-9: Performing Synthesis from the Command Prompt

2-3. Verify the Synthesis report in the Vivado HLS tool GUI.

2-3-1. Enter the following command to launch the GUI:

```
vivado_hls -p dct_prj
```

2-3-2. If the Synthesis report does not open automatically in the Vivado HLS tool GUI, select **Solution** > **Open Report** > **Synthesis**.

2-3-3. Analyze the report file.

Question 3

Write down the following details from the Synthesis report:

- Estimated clock frequency:
- Worst case latency:
- Number of DSP48E used:
- Number of FFs used:
- Number of LUTs used:

2-3-4. In the Outline pane, select **Interface** > **Summary**.

The report also shows the top-level interface signals generated by the tools.

2-3-5. In the Vivado HLS tool GUI, select **File** > **Exit** to exit the GUI.

Co-simulating and Exporting the RTL Step 3

In this step, you will perform C/RTL co-simulation on the generated RTL files by using the C testbench from the Vivado HLS tool command prompt. Also, you will generate the IP from the Vivado HLS tool command prompt.

3-1. Perform C/RTL co-simulation.

3-1-1. In the Vivado HLS tool command prompt, enter the following command to execute post-synthesis co-simulation:

```
cosim_design
```

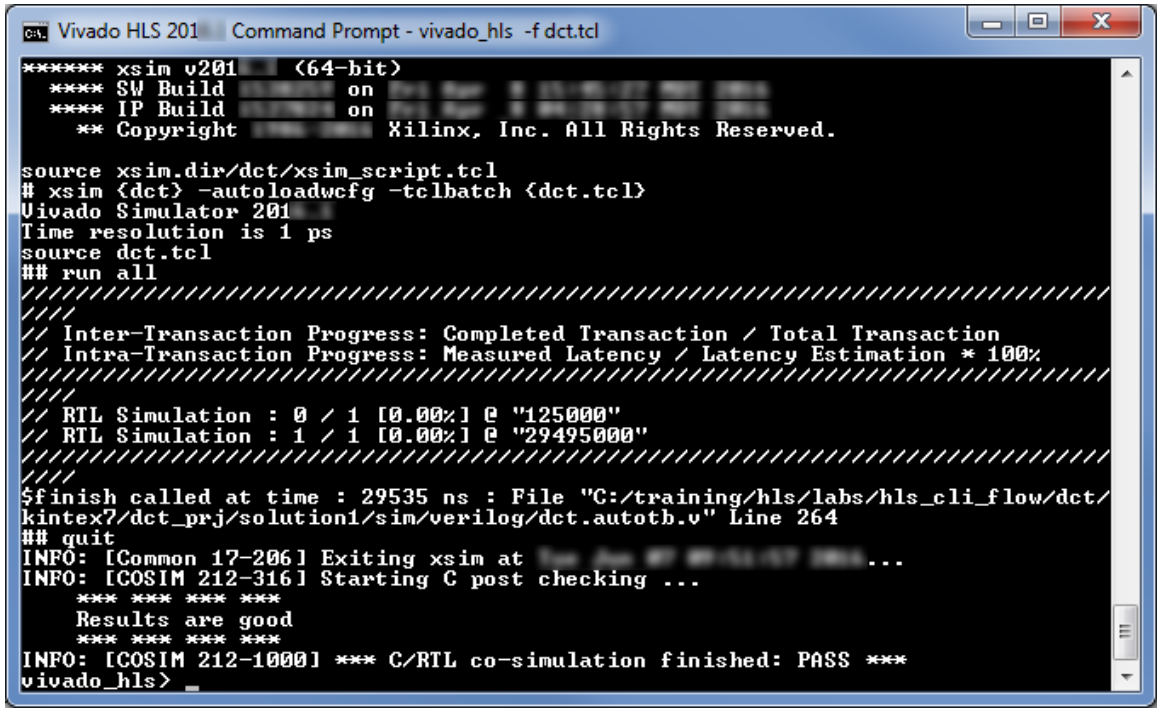


Figure 2-10: C/RTL Co-simulation

Notice that the message ***** C/RTL co-simulation finished: PASS ***** is displayed.

3-2. Export the design to IP.

3-2-1. Enter the following command to perform an RTL implementation for Verilog HDL:

```
export_design -evaluate verilog -format ipxact
```

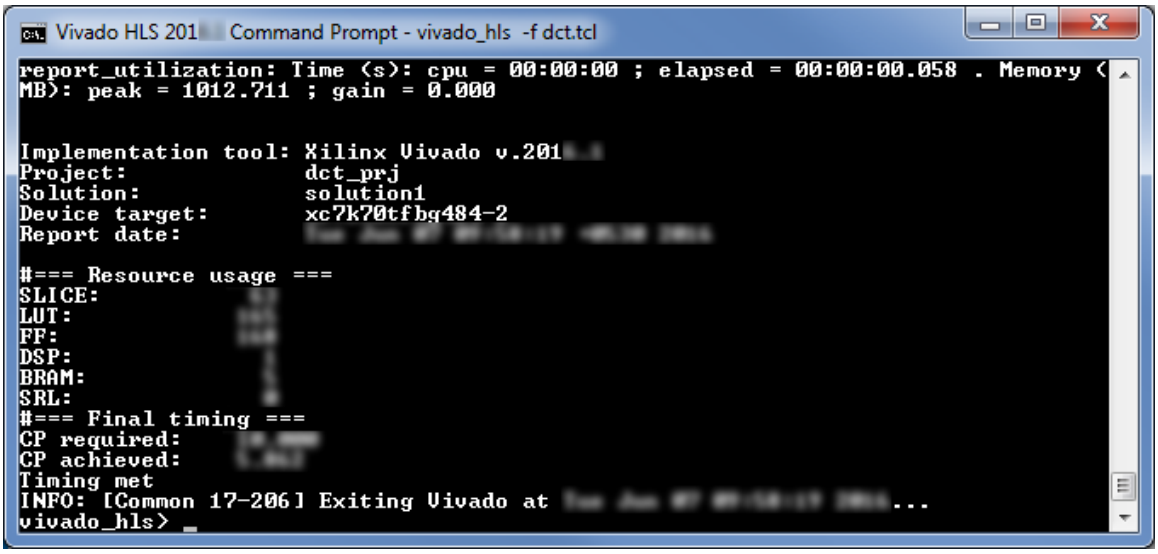


Figure 2-11: Exporting Synthesized RTL as an IP

Note: The `-evaluate` option will perform Vivado RTL synthesis and implementation on the generated IP. Implementation is run to evaluate and provide confidence that the RTL will meet its estimated timing and area goals and that these results are not included as part of the exported package.

3-3. Verify the results in the Vivado HLS tool GUI.

3-3-1. Enter the following command to launch the GUI:

```
vivado_hls -p dct_prj
```

3-3-2. In the Vivado HLS tool GUI, select **Solution** > **Open Report** > **Co-simulation** to open the RTL Simulation report.

3-3-3. Select **Solution** > **Open Report** > **Export RTL** to open the Export RTL report.

3-3-4. Analyze the reports and close the Vivado HLS tool GUI.

3-3-5. Enter the following command to close the Vivado HLS tool command prompt:

```
exit
```

Summary

In this lab, you learned how to use the Vivado HLS tool command prompt to:

- Create the Vivado HLS tool project
- Create a solution with the desired settings
- Execute major actions in the Vivado HLS (simulate, synthesize, cosimulate and export the RTL of the design) design flow

Answers

1. What does the –reset switch do in the previous command?

   This option resets the project by removing any project data that already exists. Later if you wish to run the project again, this option helps reset the project data and creates a fresh one.

2. How does the above command differ from adding testbench files to the project?

   To add the testbench source, the "`-tb`" switch should be added to the `add_files` command.

3. Write down the following details from the Synthesis report:

Estimated clock frequency:	7.48
Worst case latency:	2935
Number of DSP48E used:	1
Number of FFs used:	246
Number of LUTs used:	353

Lab 3: Interface Synthesis

2016.1

Abstract

This lab demonstrates how to implement I/O ports and protocols using high-level synthesis.

This lab should take approximately 45 minutes.

Objectives

After completing this lab, you will be able to:

- Identify the default I/O protocol for ports
- Select the appropriate I/O protocol
- Add directives to select the interface type

Introduction

This lab illustrates how RTL ports are added to a C design. Interface synthesis assigns a protocol to each of these ports. This synchronizes data transfer with the synthesized logic.

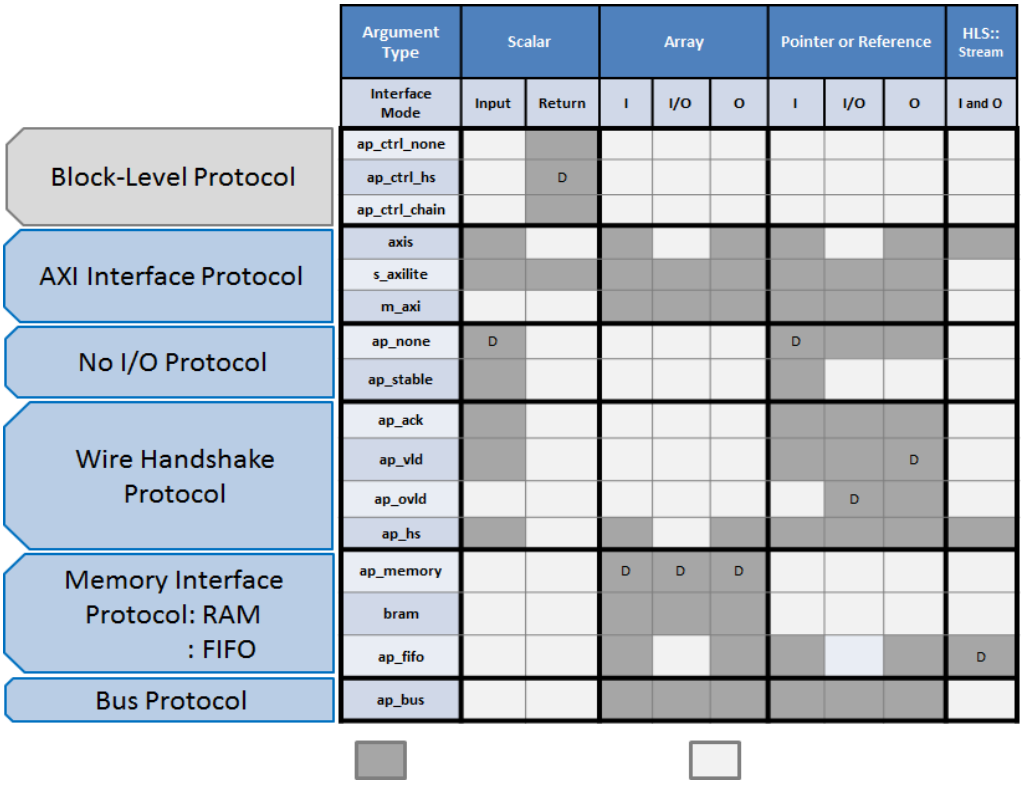


Figure 3-1: I/O Protocol Types

Block-Level Interface Protocol

By default, a block-level interface protocol is added to the design. The Vivado® HLS tool uses the interface types *ap_ctrl_none*, *ap_ctrl_hs*, and *ap_ctrl_chain* to specify whether the RTL is implemented with block-level handshake signals. Block-level handshake signals specify the following:

- When the design can start to perform the operation
- When the operation ends
- When the design is idle and ready for new inputs

These signals control the block independently of any port-level I/O protocols. These ports control when the block can start processing data (*ap_start*), indicate when it is ready to accept new inputs (*ap_ready*), and indicate if the design is idle (*ap_idle*) or has completed operation (*ap_done*).

You can specify these block-level I/O protocols on the function or the function return. If the C code does not return a value, you can still specify the block-level I/O protocol on the function return. The Vivado HLS tool creates an output port *ap_return* for the return value.

Port-Level Interface Protocol

The final group of signals are the data ports. The I/O protocol that is created depends on the type of C argument and on the default. A complete list of all possible I/O protocols is shown in the table.

After the block-level protocol has been used to start the operation of the block, the port-level I/O protocols are used to sequence data into and out of the block.

General Flow

Creating a Vivado HLS Tool Project Step 1

In this step, you will create a new Vivado® HLS tool project, add source files, and provide solution settings for the default solution using the Vivado HLS tool command prompt.

1-1. Launch the Vivado HLS tool command prompt and change the directory to the *interface_synthesis* working directory.

1-1-1. Select **Start** > **All Programs** > **Xilinx Design Tools** > **Vivado 2016.1** > **Vivado HLS** > **Vivado HLS 2016.1 Command Prompt** to launch the Vivado HLS tool command prompt.

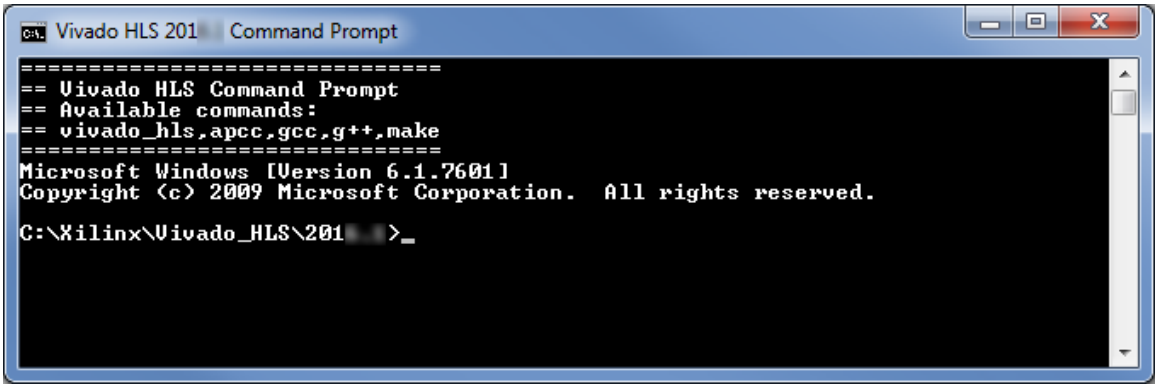


Figure 3-2: Vivado HLS Command Prompt

1-1-2. Enter the following command to change the directory to the current working directory:

```
cd C:\training\hls\labs\interface_synthesis
```

1-2. Review and run the *run_interface_syn_hls.tcl* file to create the project and open the created project in the Vivado HLS tool GUI.

1-2-1. Using any of the text editors available under the `C:\training\hls\labs\interface_synthesis` directory, open the **`run_interface_syn_hls.tcl`** file.

1-2-2. Review the following sections:

- Project name *(adders_io_prj)*
- Source files *(ardders_io.c)* and testbench *(adders_io_test.c)* added to the project
- Creating a solution *(solution1)*
- Selecting the Xilinx device *(xc7k160tfbg484-1)* and clock period *(2)*
- Running the C simulation

1-2-3. Close the **`interface_syn_hls.tcl`** file.

1-2-4. In the Vivado HLS tool command prompt, enter the following command to run the `run_interface_syn_hls.tcl` file:

```
vivado_hls -f run_interface_syn_hls.tcl
```

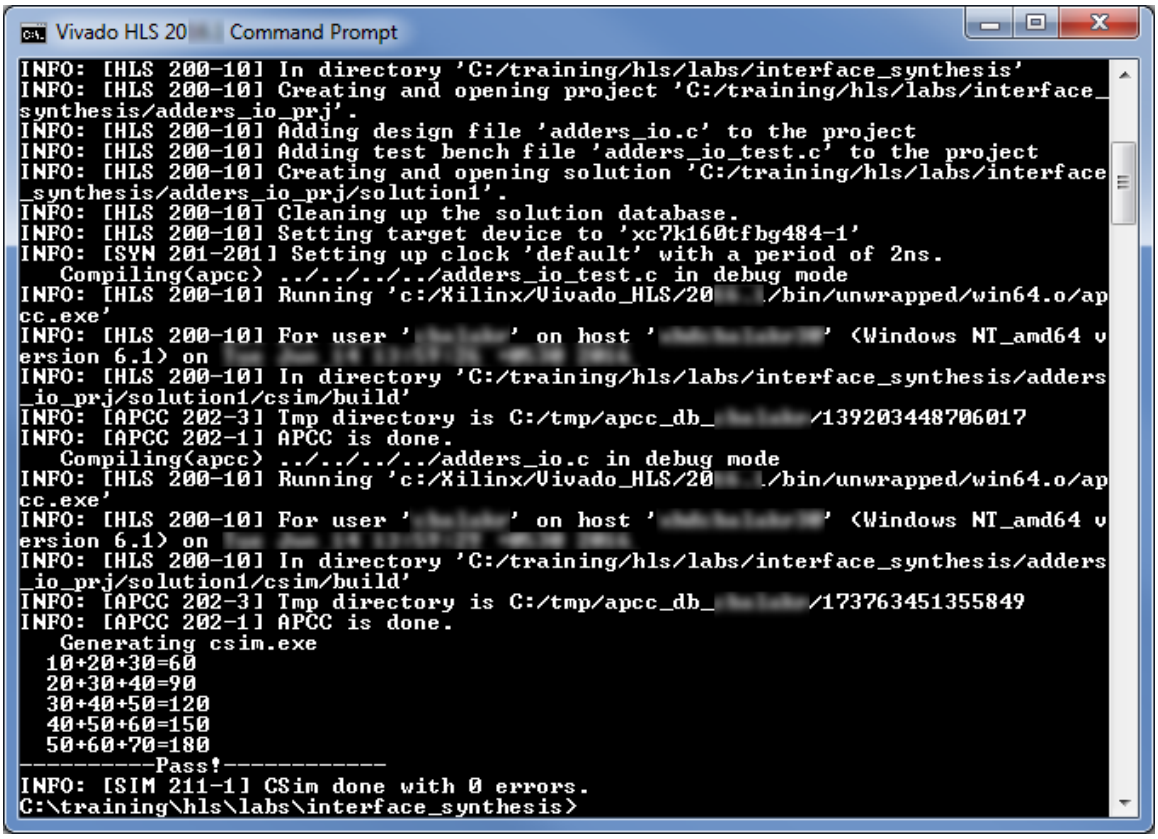


Figure 3-3: Creating the Project and Running the C Simulation

Note: Observe that the C simulation passed.

1-2-5. Enter the following command to open the Vivado HLS tool project:

```
vivado_hls -p adders_io_prj
```

The Vivado HLS tool GUI opens.

1-2-6. Expand **adders_io_prj** > **Source** in the Explorer pane.

1-2-7. Double-click the **adders_io.c** file to open it.

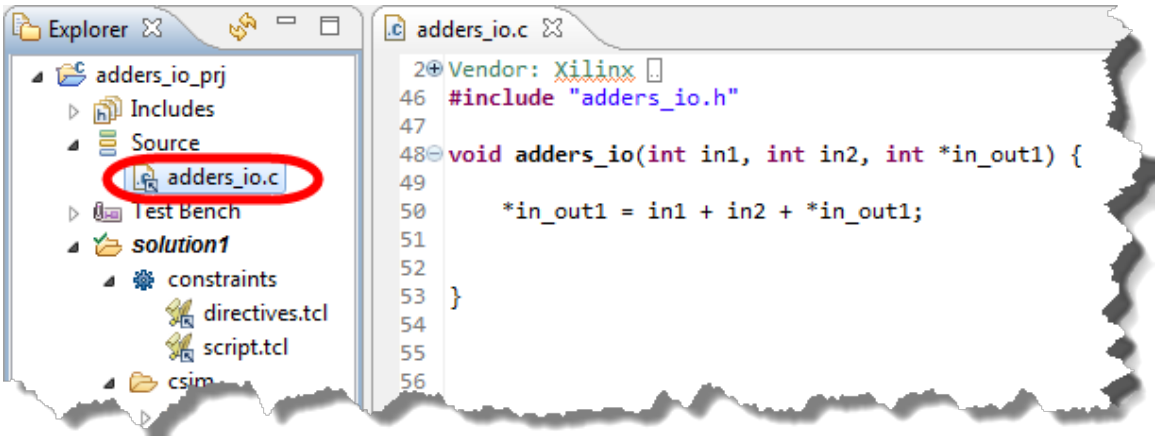


Figure 3-4: Opening the Source Code

1-2-8. Review the source file.

The source code for this lab is relatively simple to help you focus on the interface behavior and not the core logic.

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

Verifying the Default I/O Protocols Step 2

In this step, you will verify the default I/O protocols generated by the tool.

The types of I/O protocol that you can add to C function arguments by interface synthesis depends on the argument type.

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

The possible options for each function argument are described below.

Function Argument	I/O Protocol Options
in1 and *in2*	These are pass-by-value arguments that can be implemented with the following I/O protocols: • *ap_none:* No I/O protocol. This is the default for inputs. • *ap_stable:* No I/O protocol. • *ap_ack:* Implemented with an associated output acknowledge port. • *ap_vld:* Implemented with an associated input valid port. • *ap_hs:* Implemented with both input valid and output acknowledge ports.
in_out1	This is a pass-by-reference output that can be implemented with the following I/O protocols: • *ap_none:* No I/O protocol. This is the default for inputs. • *ap_stable:* No I/O protocol. • *ap_ack:* Implemented with an associated input acknowledge port. • *ap_vld:* Implemented with an associated output valid port. This is the default for outputs. • *ap_ovld:* Implemented with an associated output valid port (no valid port for the input part of any inout ports). • *ap_hs:* Implemented with both input valid port and output acknowledge ports • *ap_fifo:* A FIFO interface with associated output write and input FIFO full ports. • *ap_bus:* A Vivado HLS tool bus interface protocol.

2-1. Synthesize the design.

2-1-1. Select **Solution** > **Run C Synthesis** > **Active Solution** or click the **Run Synthesis** icon in the menu bar.

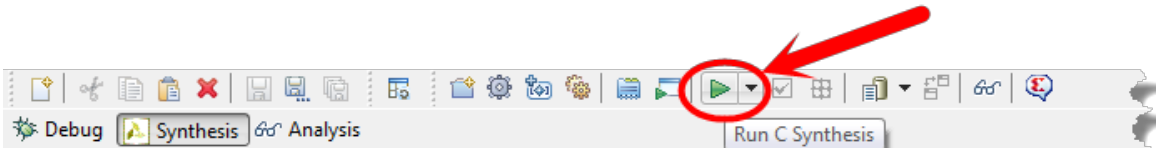


Figure 3-5: Launching Synthesis

This option synthesizes the currently selected solution.

All solutions (or selected solutions) can be synthesized by using the drop-down menu next to the synthesis icon. You can synthesize all solutions or synthesize selected solutions in addition to the default.

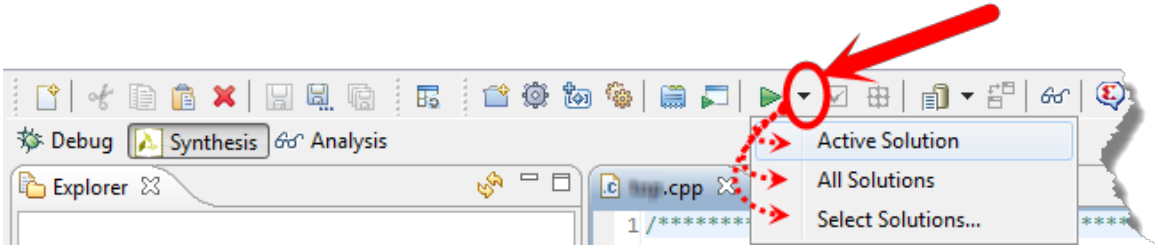


Figure 3-6: Options for What to Synthesize

2-2. View the Synthesis report.

You should see that the Synthesis report in the Information window opens automatically when synthesis is completed.

2-2-1. In the Outline pane, select **Interface** to review the Interface Summary.

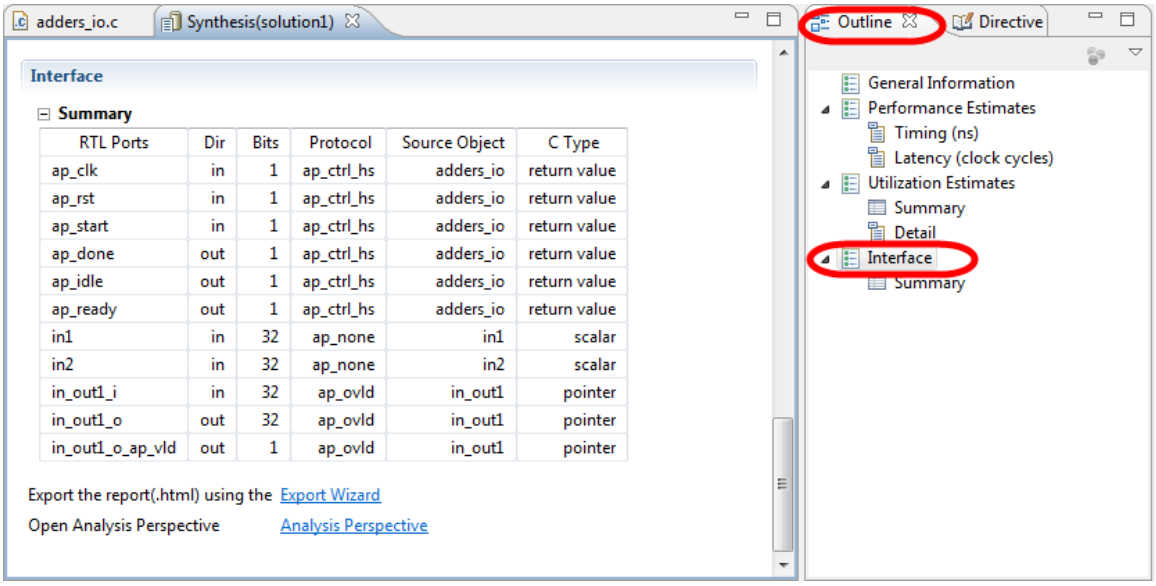


Figure 3-7: Interface Summary Report

The Interface Summary shows how the arguments in the C source are by default synthesized into RTL RAM ports:

- The design has a *clock*, *reset*, and the default block-level I/O protocol *ap_ctrl_hs*.
- Input data (*in1* and *in2*) with the port-level I/O protocol *ap_none*
- Input and output ports (*in_out1_i* and *in_out1_o*) with the port-Level I/O protocol *ap_ovld*
  - *ap_ovld* is a subset of the *ap_hs* interface type. The *ap_ovld* port-level I/O protocol provides the following signals:
    - Data port
    - Valid signal to indicate when data is read
    - For input arguments and the input half of inout arguments, the design defaults to type *ap_none*.
    - For output arguments and the output half of inout arguments, the design implements type *ap_vld*.

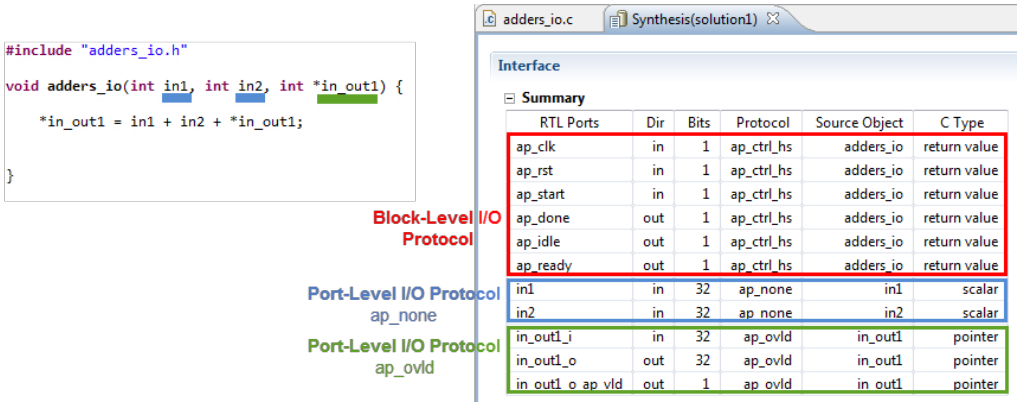


Figure 3-8: Interface Summary Report - Block-Level and Port-Level I/O Protocols

Specifying the Port-Level I/O Protocols — Step 3

3-1. Specify the I/O protocols for the ports *in1*, *in2*, and *in_out1*.

3-1-1. Ensure that the **adders_io.c** file is opened.

3-1-2. In the Directive tab, right-click **in1** and select **Insert Directive**.

3-1-3. Select **INTERFACE** from the Directive drop-down list.

3-1-4. Select **ap_vld** from the mode (optional) drop-down list.

3-1-5. Click **OK**.

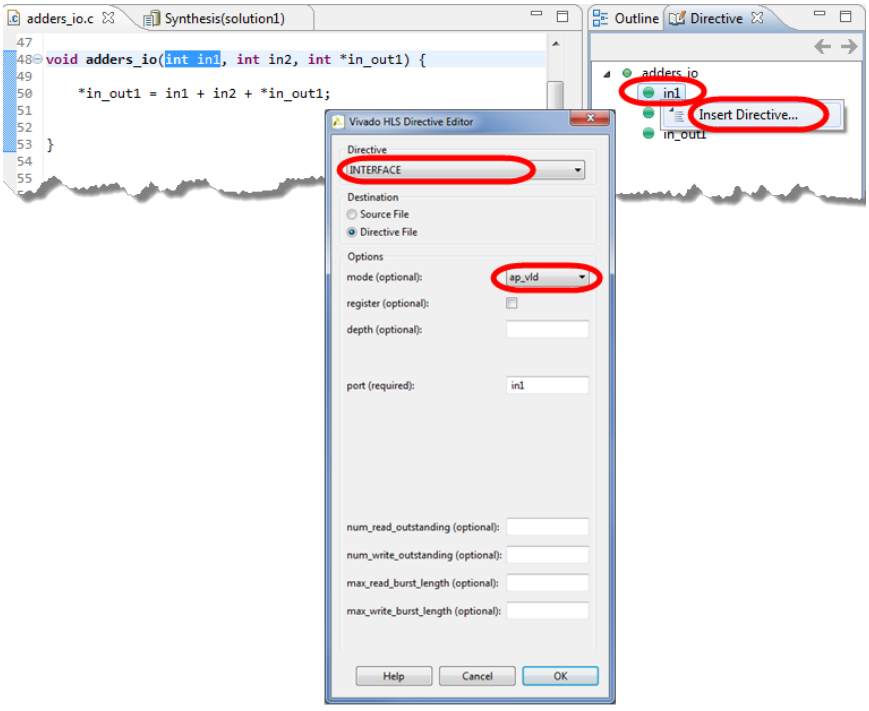


Figure 3-9: Selecting the Interface Type as ap_vld for the in1 Port

3-1-6. Similarly, select **in2** and select **INTERFACE** from the Directive list and **ap_ack** from the mode (optional) drop-down list.

3-1-7. Similarly, select **in_out1** and select **INTERFACE** from the Directive list and **ap_hs** from the mode (optional) drop-down list.

The final view of the INTERFACE directives of the ports in the Directive tab should look similar to the figure below.

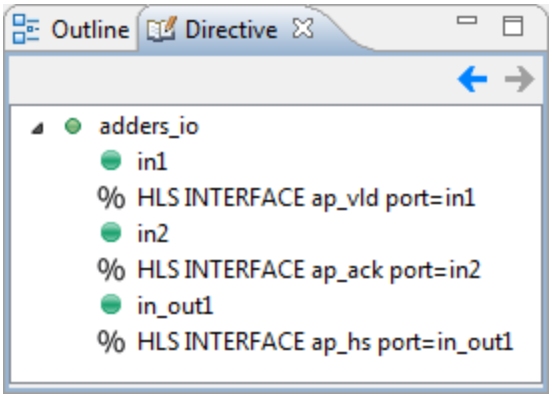


Figure 3-10: FInal View of the Directive Tab

3-1-8. In the Explorer pane, expand the **adders_io_prj** > **solution1** > **Constraints** folder.

3-1-9. Double-click the **directives.tcl** file to open it.

You should see that the directives added through the GUI are now in the *directives.tcl* file.

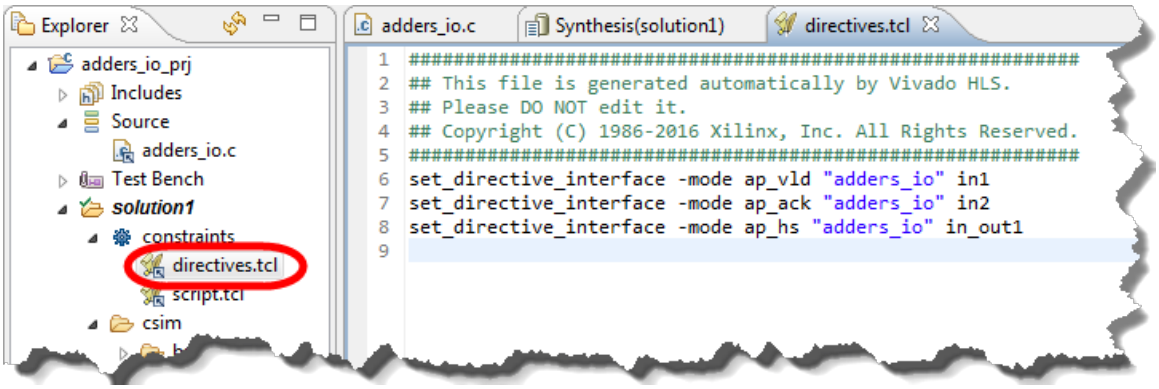


Figure 3-11: Opening the directives.tcl File

3-2. Synthesize the Vivado HLS design.

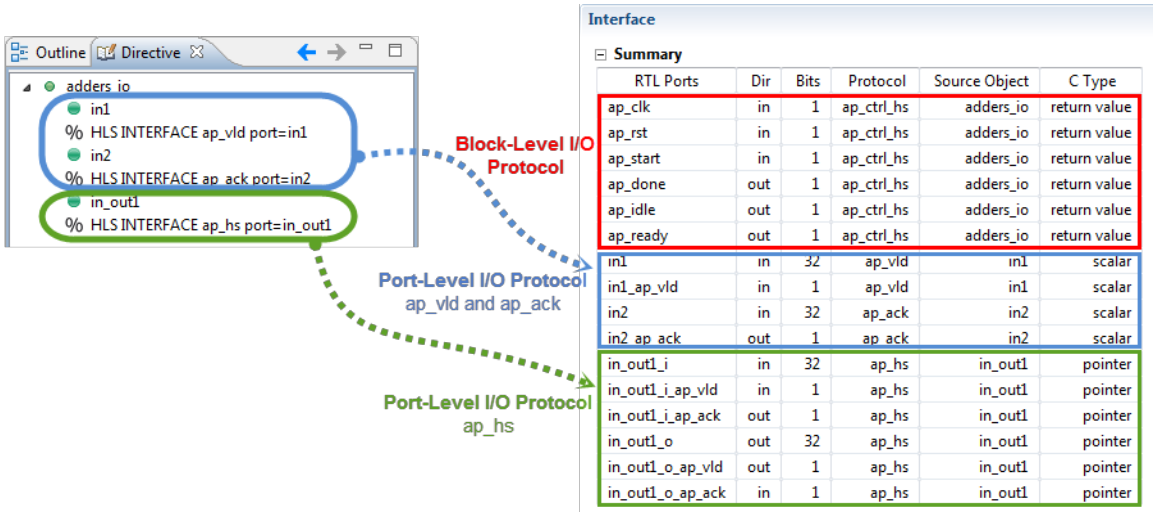
If you do not recall how to perform this task, refer to the "Synthesizing the Vivado HLS Design" section under Vivado HLS Operations in the *Lab Reference Guide*.

3-3. View the Synthesis report.

You should see that the Synthesis report in the Information window opens automatically when synthesis is completed.

3-3-1. In the Outline pane, select **Interface** to review the Interface Summary:

- The design has a clock (*ap_clk*) and reset (*ap_rst*).
- The default block-level I/O protocol signals (*ap_start*, *ap_done*, *ap_idle* and *ap_ready*) are present.
- The port *in1* is implemented with a data port and an associated input valid signal.
- The data on port *in1* is only read when the port *in1_ap_vld* is active High.
- The port *in2* is implemented with a data port and an associated output acknowledge signal.
- The port *in2_ap_ack* will be active High when the data port *in2* is read.
- *inout_i* identifies the input part of the argument *inout1*. This has the associated input valid port *inout1_i_ap_vld* and the output acknowledge port *inout1_i_ap_ack*.
- The output part of the argument *inout1* is identified as *inout_o*. This has the associated output valid port *inout1_o_ap_vld* and the input acknowledge port *inout1_o_ap_ack*.



RTL Ports	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 3-12: Interface Summary Report After Applying INTERFACE Directives

3-3-2. Select **File** > **Exit** to close the Vivado HLS tool.

3-3-3. Close the Vivado HLS Command Prompt.

Summary

In this lab, you learned how to apply directives to implement different types of port-Level I/O protocols. You also reviewed the Interface Summary to see the types of port-level I/O protocols that were implemented.

Answers

Since there were no questions in this lab, this section is intentionally left blank.

Lab 4: Improving Performance

2016.1

Abstract

This lab describes how to improve design performance.

This lab should take approximately 45 minutes.

Objectives

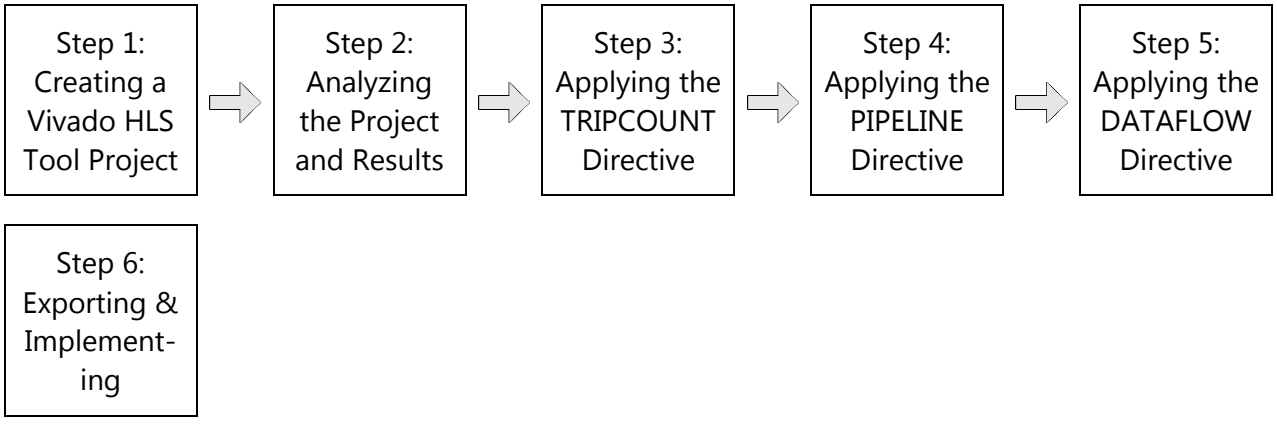
After completing this lab, you will be able to:

- Identify the effect of applying the INLINE directive
- Improve performance by using the PIPELINE directive
- Distinguish between the DATAFLOW directive and the Configuration Command feature

Introduction

This lab introduces various techniques and directives that can be used in the Vivado® HLS tool to improve design performance. The design under consideration in the lab accepts an image in a custom RGB format, converts it to the YUV color space, applies a filter to the YUV image, and converts it back to RGB.

General Flow



Creating a Vivado HLS Tool Project Step 1

In this step, you will create a new Vivado HLS tool project, add source files, and provide solution settings for the default solution using the Vivado HLS tool command prompt.

1-1. Launch the Vivado HLS tool command prompt and change the directory to the *improve_performance* working directory.

1-1-1. Select **Start** > **All Programs** > **Xilinx Design Tools** > **Vivado 2016.1** > **Vivado HLS** > **Vivado HLS 2016.1 Command Prompt** to launch the Vivado HLS tool command prompt.

1-1-2. Enter the following command to change the directory to the current working directory:

```
cd C:\training\hls\labs\improve_performance
```

1-2. Review and run the *run_yuv_filter.tcl* file to create the project and open the created project in the Vivado HLS tool GUI.

1-2-1. Using any of the text editors available under the `C:\training\hls\labs\improve_performance` directory, open the **`run_yuv_filter.tcl`** file.

1-2-2. Review the following sections:

- Project name *(yuv_filter_prj)*
- Source files *(yuv_filterc)* and testbench *(yuv_filter_test.c, image_aux.c*, and *output.golden.dat)* added to the project
- Creating a solution *(solution1)*
- Selecting the Xilinx device *(xc7z020clg484-1)* and clock period *(10)*
- Running the C simulation
- Synthesizing the design

1-2-3. Close the **`run_yuv_filter.tcl`** file.

1-2-4. In the Vivado HLS tool command prompt, enter the following command to run the `run_yuv_filter.tcl` file:

```
vivado_hls -f run_yuv_filter.tcl
```

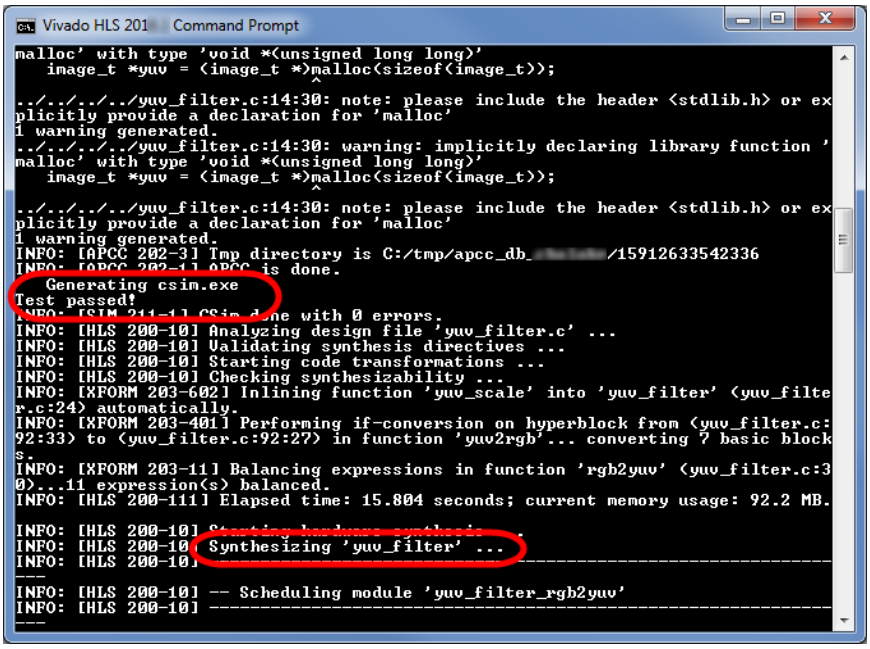


Figure 4-1: Viewing the C Simulation Results

Note: Observe that the C simulation passed and that the design starts synthesizing.

1-2-5. Enter the following command to open the Vivado HLS tool project:

```
vivado_hls -p yuv_filter_prj
```

The Vivado HLS tool GUI opens.

Analyzing the Project and Results Step 2

2-1. Open the source file and review the three functions. Look at the results and observe that the latencies are undefined (represented by "?").

2-1-1. Expand **yuv_filter_prj** > **Source** in the Project Explorer pane.

2-1-2. Double-click the **yuv_filter.c** file to open it and review the design.

The design has three functions: `rgb2yuv`, `yuv_scale`, and `yuv2rgb`.

Each of these filter functions iterates over the entire source image (which has maximum dimensions specified in `image_aux.h`.), requiring a single-source pixel to produce a pixel in the resulting image. The `image_aux.h` file can be accessed by expanding the **yuv_filter_prj** > **Includes** > **C:/training/hls/labs/improve_performance** folder in the Project Explorer pane.

The scale function simply applies individual scale factors, supplied as top-level arguments to the YUV components.

Notice that most of the variables are of user-defined (typedef) and aggregate (e.g. structure, array) types.

Also notice that the original source used *malloc()* to dynamically allocate storage for the internal image buffers. While appropriate for such large data structures in software, *malloc()* is not synthesizable and is not supported by the Vivado HLS tool.

A viable workaround is conditionally compiled into the code, leveraging the __SYNTHESIS__ macro. The Vivado HLS tool automatically defines the __SYNTHESIS__ macro when reading any code. This ensure that the original *malloc()* code is used outside of synthesis but the Vivado HLS tool will use the workaround when synthesizing.

2-1-3. Expand the **yuv_filter_prj** > **solution1** > **syn** > **report** folder in the Project Explorer pane.

2-1-4. Double-click the **yuv_filter_csynth.rpt** file to view the Synthesis report.

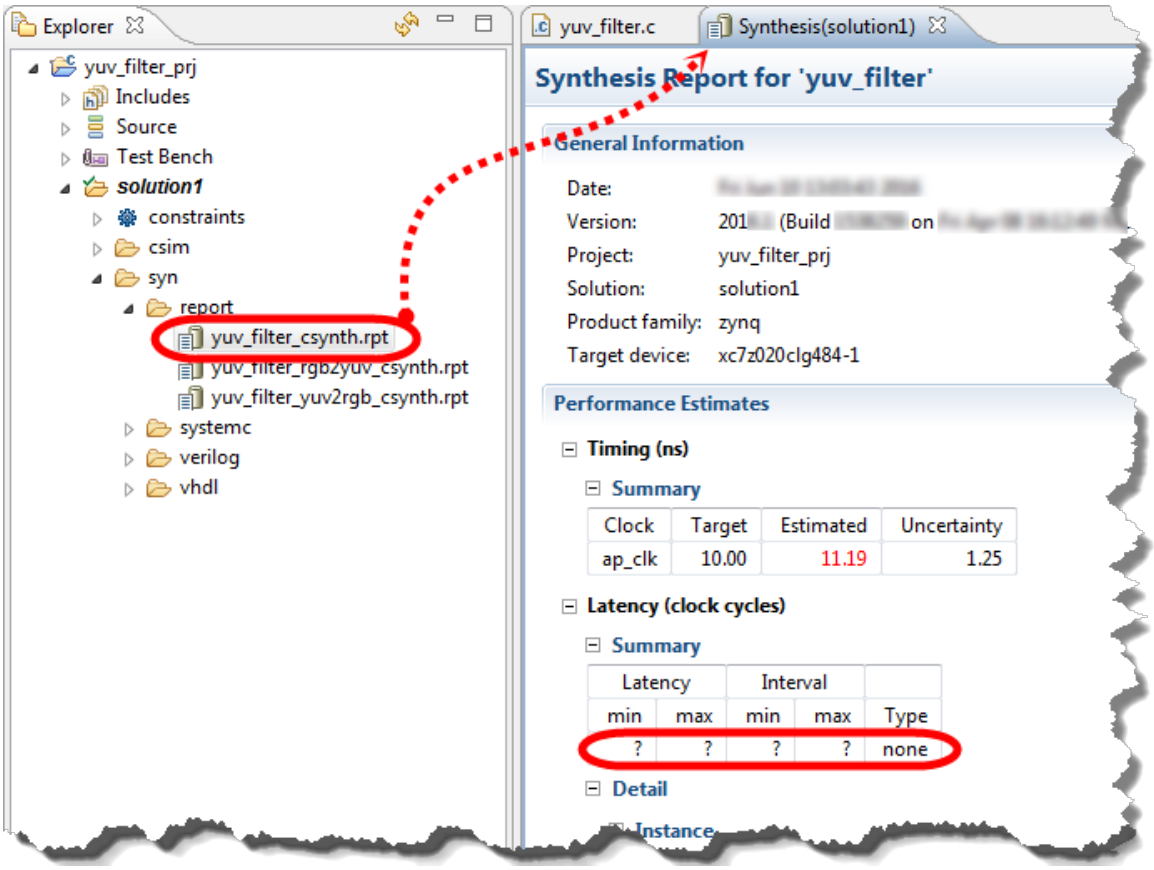


Figure 4-2: Viewing the Synthesis Report – Latency Computation

Each of the loops in this design has variable bounds—the width and height are defined by members of input type `image_t`. When variables bounds are present on loops, the total latency of the loops cannot be determined. This impacts the ability to perform analysis using reports. Hence, **"?"** is reported for various latencies.

Applying the TRIPCOUNT Directive Step 3

To assist in providing loop-latency estimates, the Vivado HLS tool provides a TRIPCOUNT directive that allows limits on the variable bounds to be specified by the user. In this design, such directives have been embedded in the source code in the form of #pragma statements.

In this step, you will open the source file and uncomment the pragma lines, resynthesize, and observe the resources used as well as the estimated latencies.

3-1. Uncomment the pragma lines to add the TRIPCOUNT directive.

3-1-1. Expand **yuv_filter_prj** > **Source** in the Project Explorer pane.

3-1-2. Double-click the **yuv_filter.c** file to open it.

3-1-3. Uncomment the #pragma lines (50, 53, 90, 93, 130, 133) to define the loop bounds.

3-1-4. Press <**Ctrl** + **S**> to save the file.

3-2. Synthesize the Vivado HLS design.

If you do not recall how to perform this task, refer to the "Synthesizing the Vivado HLS Design" section under Vivado HLS Operations in the *Lab Reference Guide*.

3-3. View the Synthesis report.

3-3-1. View the Synthesis report in the Information window.

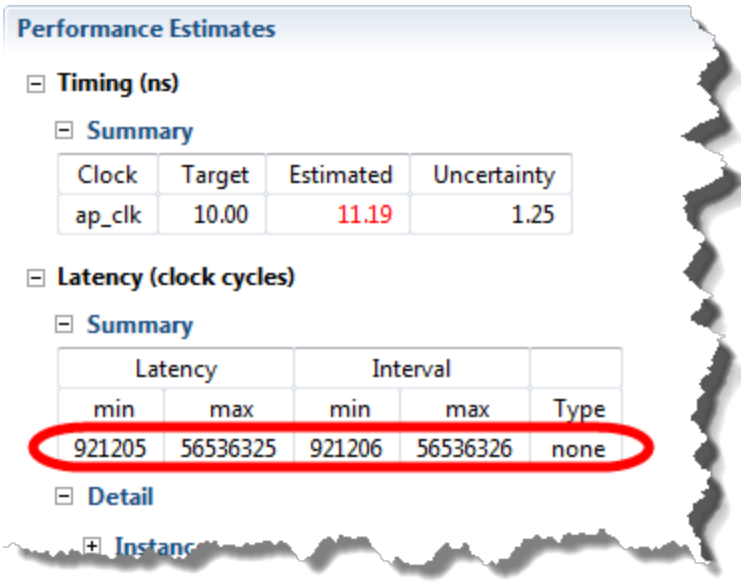


Figure 4-3: Latency Computation After Applying the TRIPCOUNT Pragma

Question 1

Looking at the report, fill in the table below.

Estimated clock period	
Worst case latency	
Number of DSP48E used	
Number of BRAMs used	
Number of FFs used	
Number of LUTs used	

Performance and Utilization Estimates After the TRIPCOUNT Directive

3-3-2. In the Console tab, observe that the `yuv_scale` function is automatically inlined into the `yuv_filter` function.

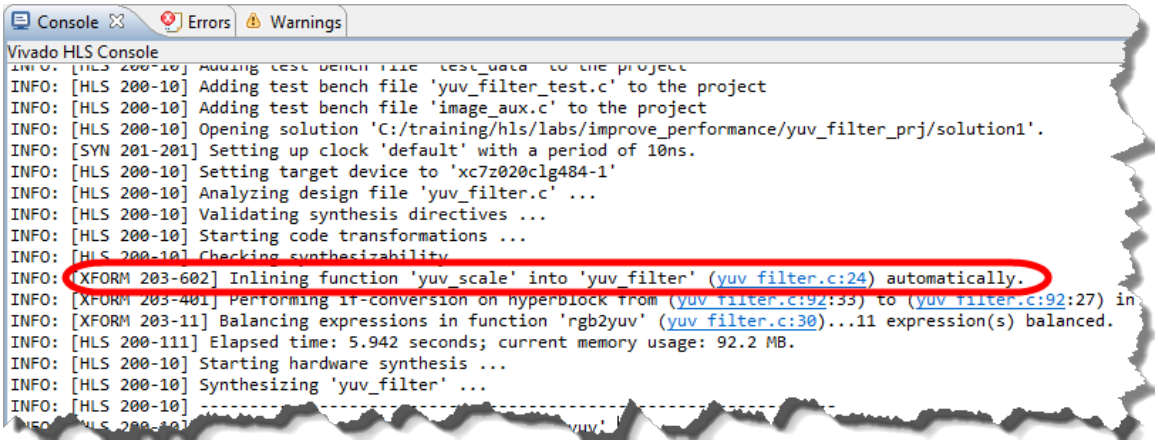


Figure 4-4: Observing the Vivado HLS Tool Automatically Inlining the Function yuv_scale

Observe that there are three synthesis reports under the *syn* > *report* folder in the Explorer view:

- `yuv_filter` (*yuv_filter_csynth.rpt*)
- `rgb2yuv` (*yuv_filter_rgb2yuv_csynth.rpt*)
- `yuv2rgb` (*yuv_filter_yuv2rgb_csynth.rpt*)

There is no entry for `yuv_scale` since this function was inlined into the `yuv_filter` function.

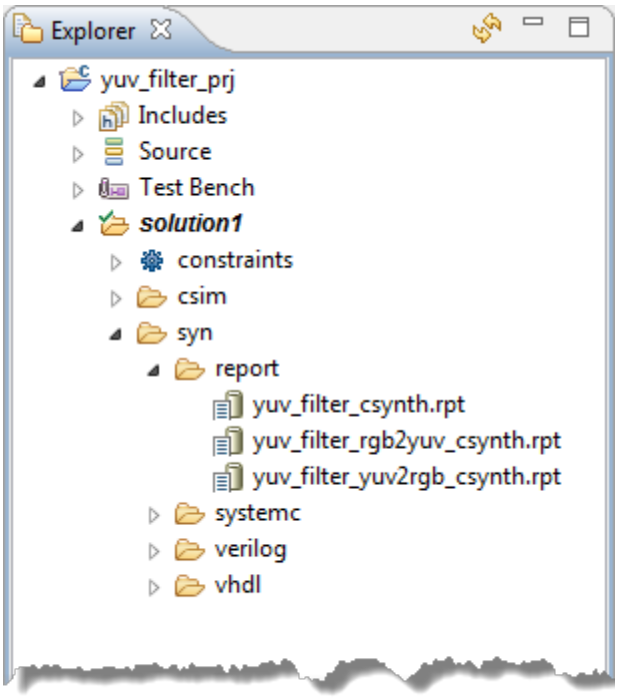
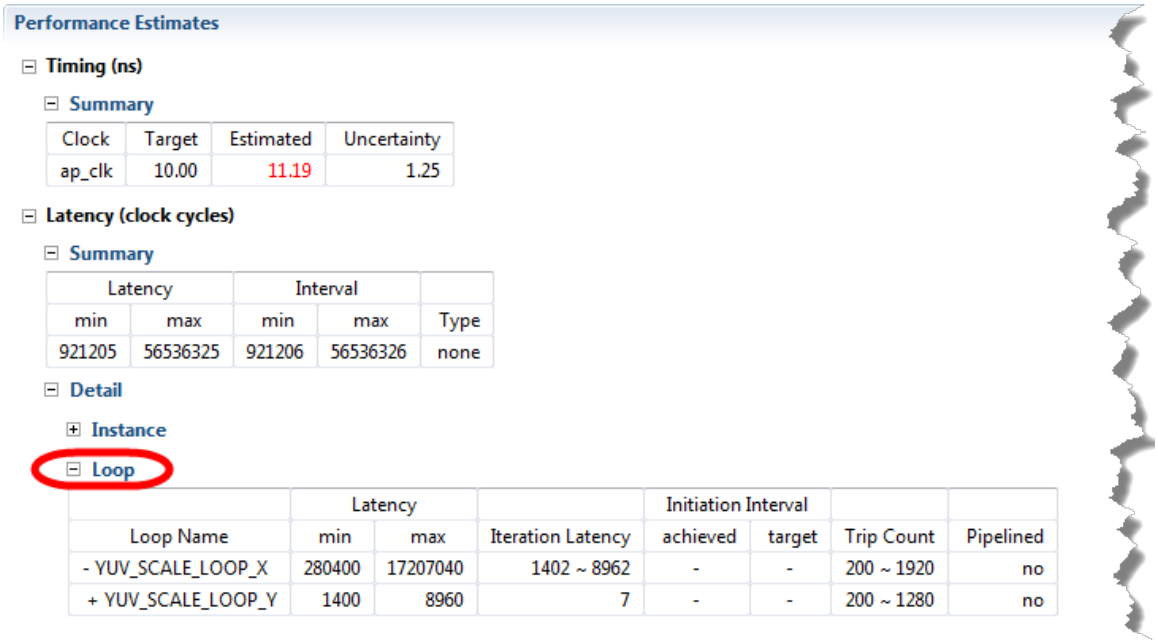


Figure 4-5: Viewing the Synthesis Report (After Applying the TRIPCOUNT Pragma)

3-3-3. Expand the **Summary** of the Loop latency.



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	11.19	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
921205	56536325	921206	56536326	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- YUV_SCALE_LOOP_X	280400	17207040	1402 ~ 8962	-	-	200 ~ 1920	no
+ YUV_SCALE_LOOP_Y	1400	8960	7	-	-	200 ~ 1280	no

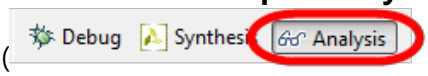
Figure 4-6: Viewing the Details of the Loop Latency

Note the latency and trip count numbers for the `yuv_scale` function.

Also note that the YUV_SCALE_LOOP_Y loop latency is 7X the specified TRIPCOUNT, implying that seven cycles are used for each of the iterations of the loop.

You can verify this by opening an analysis perspective view, expanding the YUV_SCALE_LOOP_X entry, and then expanding the YUV_SCALE_Y entry.

3-4. Switch to the Analysis perspective and understand the design behavior.

3-4-1. Select **Solution** > **Open Analysis Perspective** or click the **Analysis** perspective icon (

) to open the analysis viewer.

3-4-2. Expand **YUV_SCALE_LOOP_X** and **YUV_SCALE_LOOP_Y**.

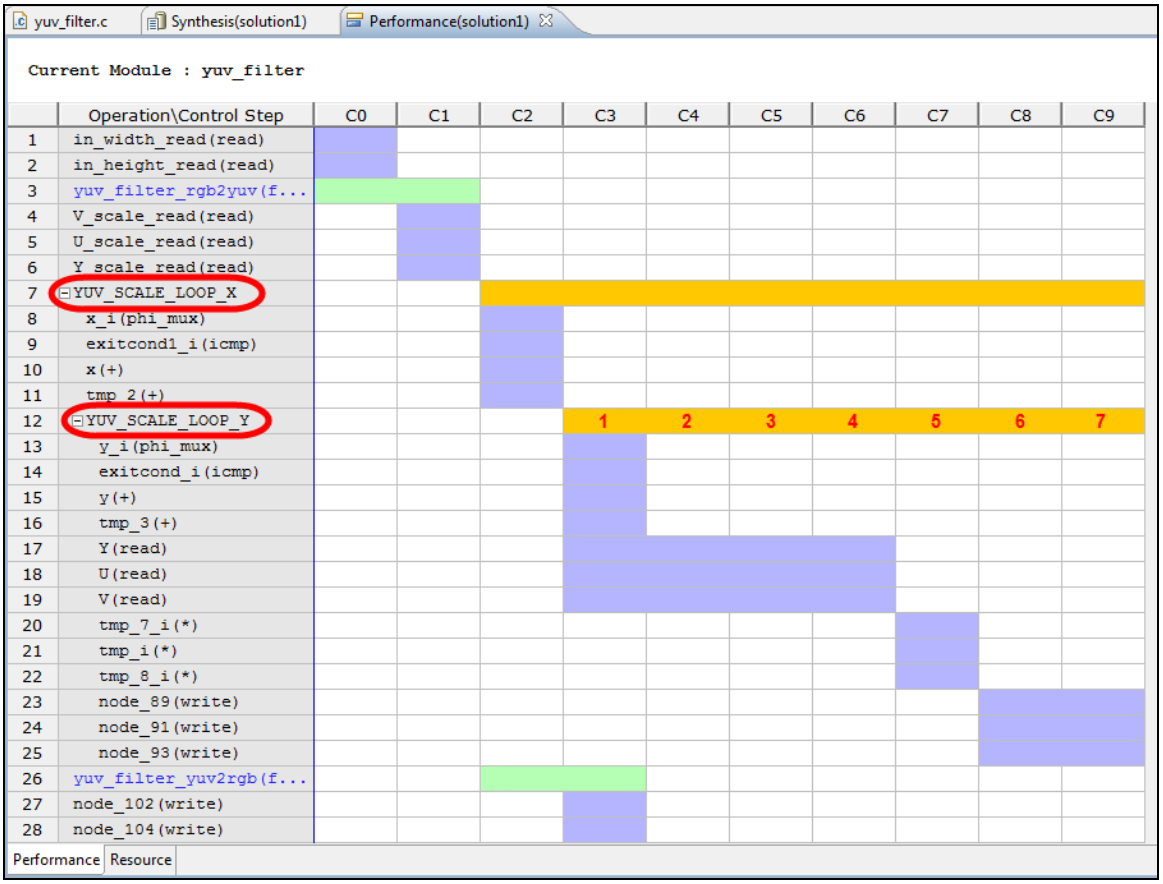


Figure 4-7: Design Analysis View of the YUV_SCALE_LOOP_Y Loop

3-4-3. Double-click **yuv_filter_rgb2yuv_csynth.rpt** under the yuv_filter_prj > solution1 > syn > report folder.

3-4-4. In the Synthesis report, under the Utilization Estimates section, note the resources used for the `rgb2yuv` function.

Question 2

Looking at the report, fill in the table below.

Estimated clock period	
Worst case latency	
Number of DSP48E used	
Number of BRAMs used	
Number of FFs used	
Number of LUTs used	

Performance and Utilization Estimates for the *rgb2yuv* Function

3-4-5. Similarly, open **yuv_filter_yuv2rgb_csynth.rpt** under the yuv_filter_prj > solution1 > syn > report folder.

3-4-6. In the Synthesis report, under the Utilization Estimates section, note the resources used for the `yuv2rgb` function.

Question 3

Looking at the report, fill in the table below

Estimated clock period	
Worst case latency	
Number of DSP48E used	
Number of BRAMs used	
Number of FFs used	
Number of LUTs used	

Performance and Utilization Estimates for the *yuv2rgb* Function

For the `rgb2yuv` function, the worst-case latency is reported as 19664641 clock cycles.

The reported latency can be estimated as follows:

- RGB2YUV_LOOP_Y total loop latency = 8 x 1280 = 10240 cycles
- One entry and one exit clock for loop RGB2YUV_LOOP_Y = 10242 cycles

- RGB2YUV_LOOP_X loop body latency = 10242 cycles
- RGB2YUV_LOOP_X total loop latency = 10242 x 1920 = 19664640 cycles

Turning Off INLINE and Applying the PIPELINE Directive Step 4

In this step, you will create a new solution by copying the previous solution settings. You will then configure the new solution by tuning off the automatic INLINE directive and applying PIPELINE directive.

4-1. Create a new solution by copying the previous solution (solution1) settings.

4-1-1. Select **Project** > **New Solution** or click the **New Solution** icon ( ) from the toolbar.

4-1-2. Leave the options at their default settings.

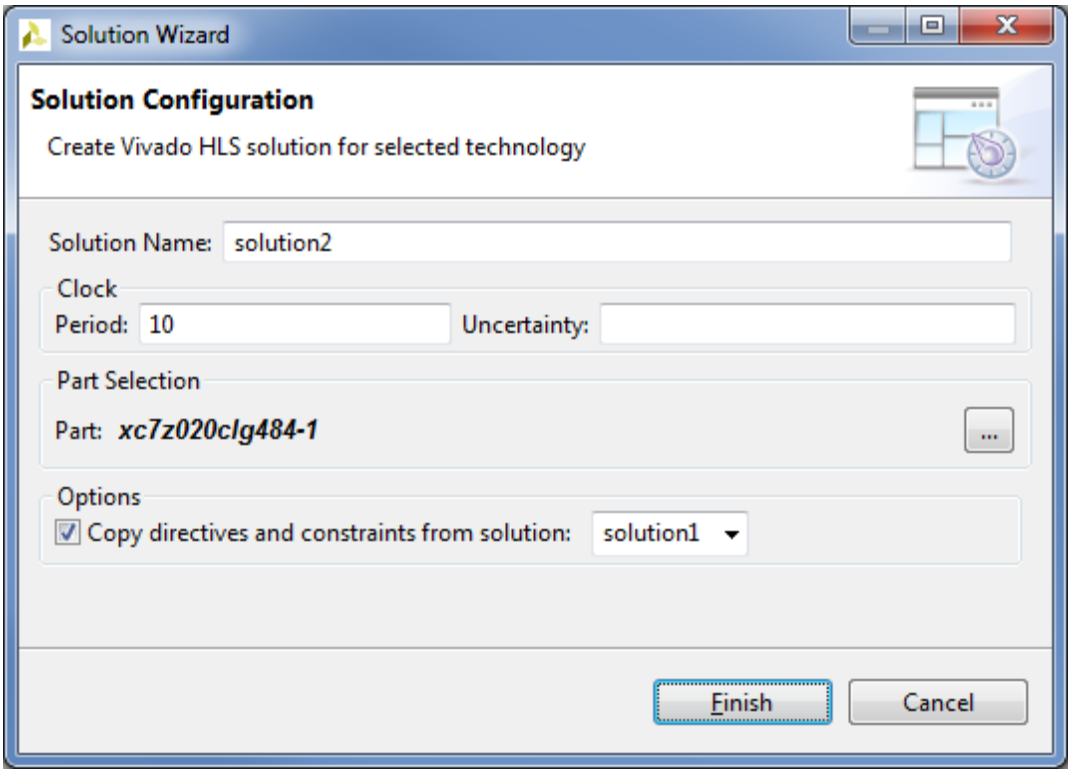


Figure 4-8: Creating a New Solution (solution2)

4-1-3. Click **Finish**.

4-2. Turn off the INLINE directive to the function *yuv_scale*.

4-2-1. Select **Project** > **Close Inactive Solution Tabs** to close all inactive solution windows.

4-2-2. Ensure that the **yuv_filter.c** file is open and active in the Information pane.

4-2-3. Select **yuv_filter** in the Directive tab.

4-2-4. In the Directive tab, right-click the **yuv_filter** function and select **Insert Directive**.

4-2-5. Select **INLINE** from the Directive drop-down list.

4-2-6. Select the **off (optional)** option.

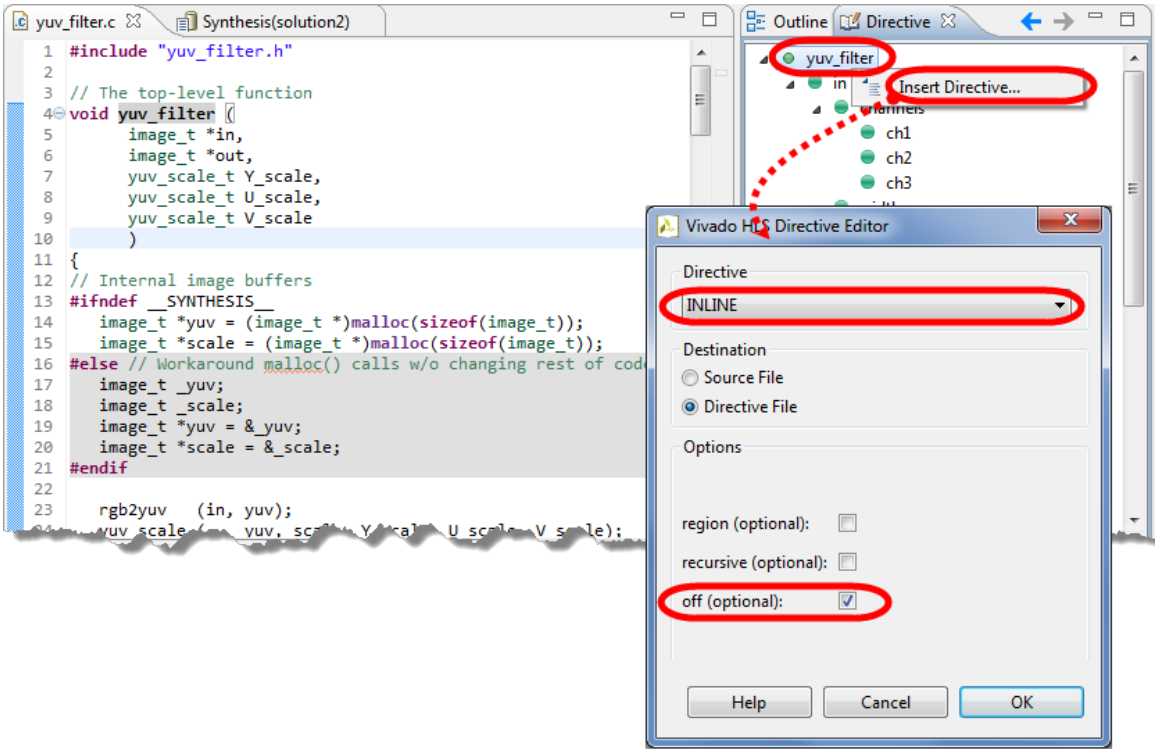


Figure 4-9: Turning Off the INLINE Function

4-2-7. Click **OK**.

When an object (function or loop) is pipelined, all the loops below it, down through the hierarchy, will be automatically unrolled.

In order for a loop to be unrolled, it must have fixed bounds: All the loops in this design have variable bounds, defined by an input argument variable to the top-level function.

Note that the **TRIPCOUNT** directive on the loops only influences reporting, it does not set bounds for synthesis.

Neither the top-level function nor any of the sub-functions are pipelined in this example.

The **PIPELINE** directive must be applied to the inner-most loop in each function. The innermost loops have no variable-bounded loops inside of them which are required to be unrolled and the outer loop will simply keep the inner loop fed with data.

4-3. Apply the PIPELINE directive to the *YUV_SCALE_LOOP_Y* loop.

4-3-1. Expand **yuv_scale** in the Directive tab.

4-3-2. In the Directive tab, right-click the **YUV_SCALE_LOOP_Y** function and select **Insert Directive**.

4-3-3. Select **PIPELINE** from the Directive drop-down list.

4-3-4. Leave the **II (optional)** field blank as the Vivado HLS tool will try for an II=1, one new input every clock cycle.

4-3-5. Click **OK**.

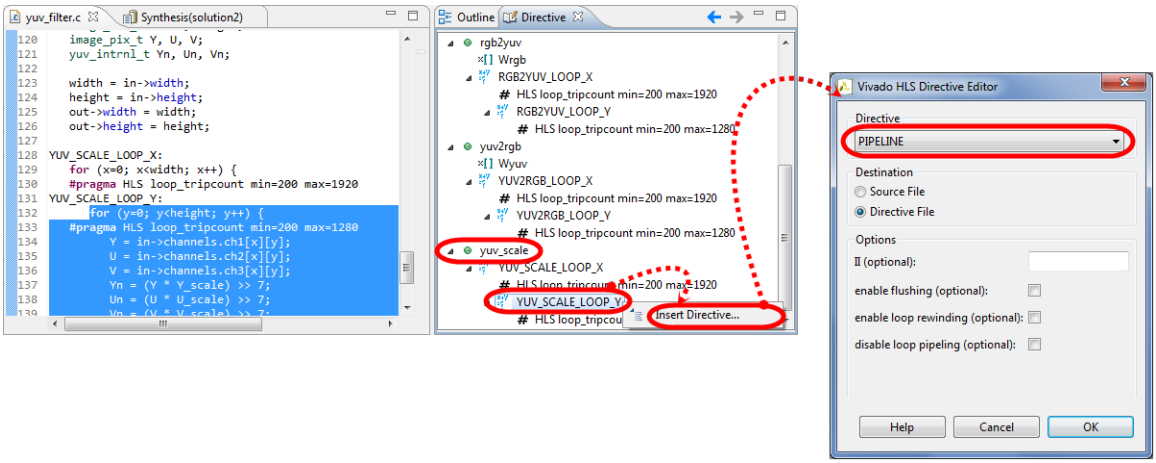


Figure 4-10: Applying the PIPELINE Directive for the YUV_SCALE_LOOP_Y Loop

4-3-6. Similarly, apply the PIPELINE directive to the **YUV2RGB_LOOP_Y** and **RGB2YUV_LOOP_Y** loops.

The final view of PIPELINE directives in the Directive tab should look similar to the figure below.

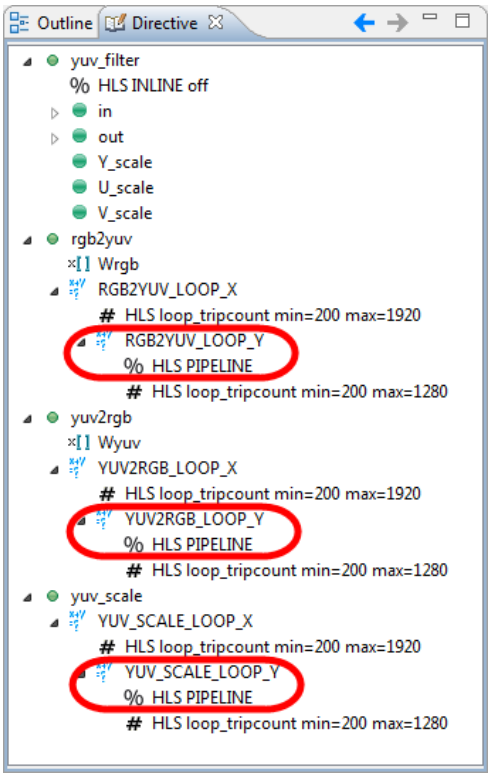


Figure 4-11: Final View of PIPELINE Directives

4-4. Synthesize the Vivado HLS design.

If you do not recall how to perform this task, refer to the "Synthesizing the Vivado HLS Design" section under Vivado HLS Operations in the *Lab Reference Guide*.

4-5. View the Synthesis report.

You should see that the Synthesis report in the Information window opens automatically when synthesis is completed.

4-5-1. Select **Project** > **Compare Reports** or click the **Compare Reports** icon ( ) to compare the results of the two solutions (*solution1* and *solution2*).

4-5-2. Select **solution1** and **solution2** from the Available solutions section.

4-5-3. Click **Add**.

4-5-4. Click **OK**.

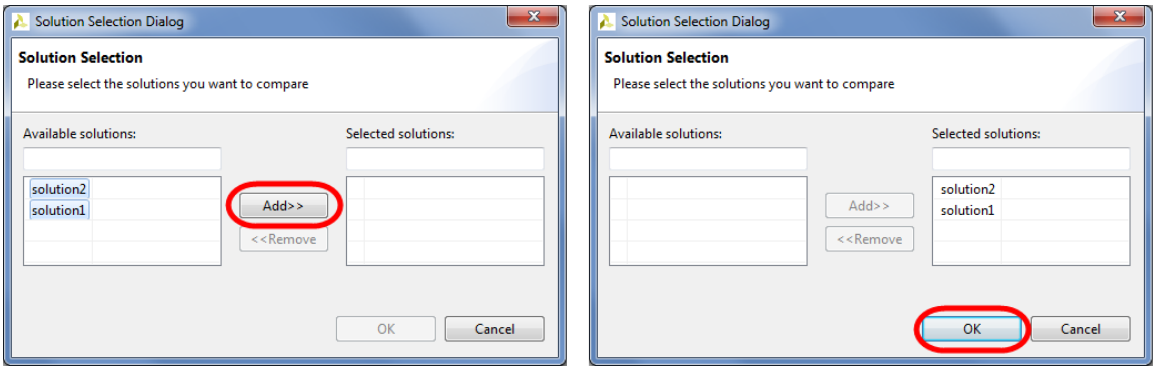
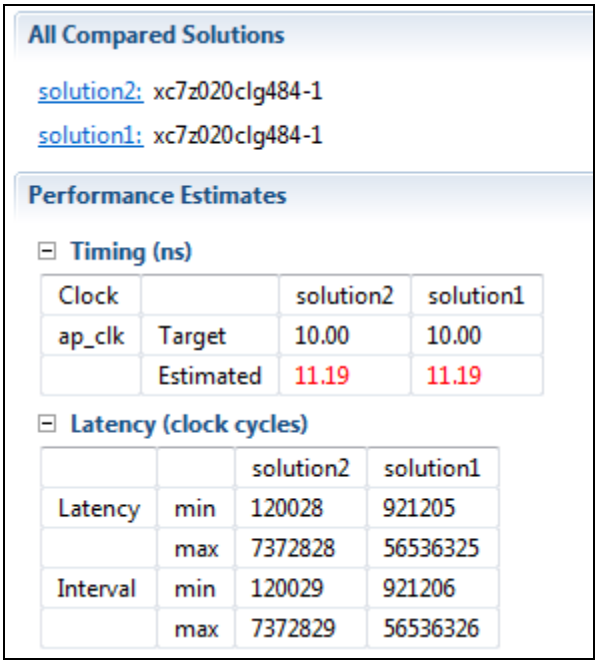


Figure 4-12: Selecting solution1 and solution2

You should see the compared results of *solution1* and *solution2* as shown in the figure below.



All Compared Solutions

solution2: xc7z020clg484-1

solution1: xc7z020clg484-1

Performance Estimates

Timing (ns)

Clock		solution2	solution1
ap_clk	Target	10.00	10.00
	Estimated	11.19	11.19

Latency (clock cycles)

		solution2	solution1
Latency	min	120028	921205
	max	7372828	56536325
Interval	min	120029	921206
	max	7372829	56536326

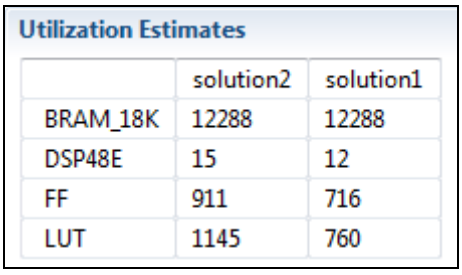
Figure 4-13: Performance Comparison After Pipelining

Observe that the latency reduced from 56536325 to 7372828 clock cycles.

In *solution1*, the total loop latency of the innermost loop was **loop_body_latency** x **loop iteration count**, whereas in *solution2* the new total loop latency of the innermost loop was **loop_body_latency** + **loop iteration count**.

4-5-5. Scroll down in the comparison report to view the resource utilization.

Observe that the flip-flops, LUTs, and DSP48E utilization increased, whereas block RAM remained the same.



Utilization Estimates

	solution2	solution1
BRAM_18K	12288	12288
DSP48E	15	12
FF	911	716
LUT	1145	760

Figure 4-14: Comparison of Resource Estimates

Applying the DATAFLOW Directive and Configuration Command Step 5

In this step, you will create a new solution by copying the previous solution settings. You will then configure the new solution by applying the DATAFLOW directive.

5-1. Create a new solution by copying the previous solution (solution2) settings.

5-1-1. Select **Project** > **New Solution** or click **New Solution** icon ( ) from the toolbar.

5-1-2. Leave the options at their default settings.

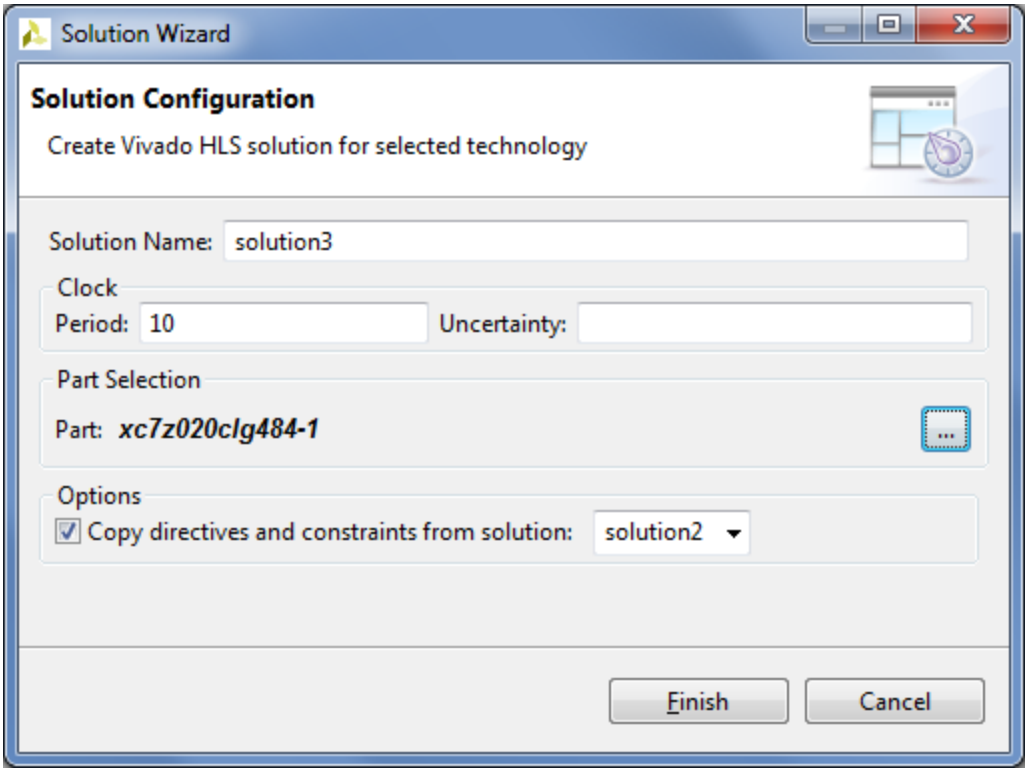


Figure 4-15: Creating a New Solution (solution3)

5-1-3. Click **Finish**.

5-2. Apply the DATAFLOW directive to the *YUV_filter* function.

5-2-1. Select **Project** > **Close Inactive Solution Tabs** to close all inactive solution windows.

5-2-2. Ensure that the **yuv_filter.c** file is open and active in the Information pane.

5-2-3. Select **yuv_filter** in the Directive tab.

5-2-4. In the Directive tab, right-click the **yuv_filter** function and select **Insert Directive**.

5-2-5. Select **DATAFLOW** from the Directive drop-down list.

5-2-6. Click **OK**.

5-3. Synthesize the Vivado HLS design.

If you do not recall how to perform this task, refer to the "Synthesizing the Vivado HLS Design" section under Vivado HLS Operations in the *Lab Reference Guide*.

5-4. View the Synthesis report.

You should see that the Synthesis report in the Information window opens automatically when synthesis is completed.

5-4-1. Observe that additional information, "Type dataflow", in the Performance Estimates section is now indicated.

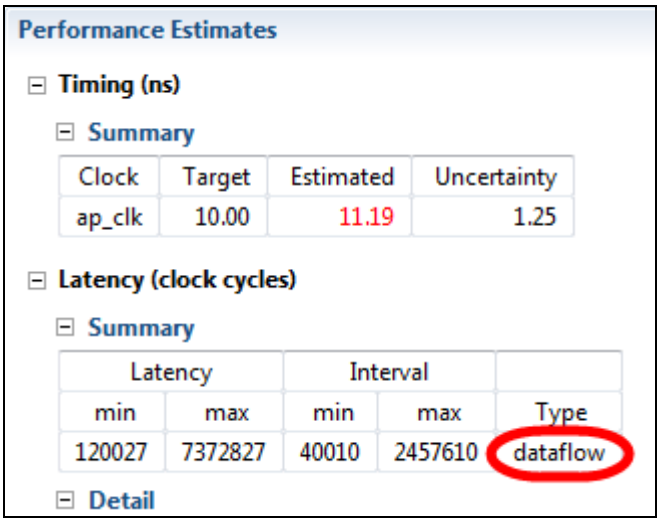


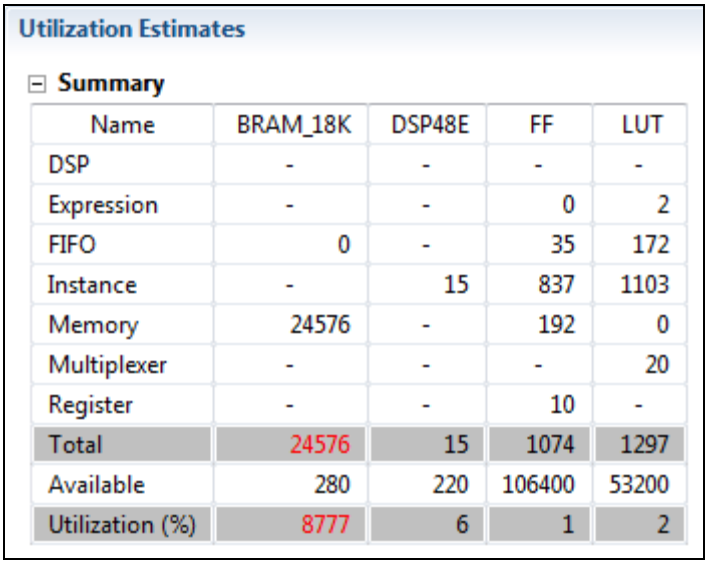
Figure 4-16: Performance Estimate After Applying the DATAFLOW Directive

The Dataflow pipeline throughput indicates the number of clocks cycles between each set of input reads. If this throughput value is less than the design latency, it indicates that the design can start processing new inputs before the current input data is output.

While the overall latencies have not changed significantly, the dataflow throughput is showing that the design can achieve close to the theoretical limit (1920x1280 = 2457600) of processing one pixel every clock cycle.

5-4-2. Scroll down in the report to view the resource utilization.

Observe that the number of block RAM required has been doubled. This is due to the default dataflow **ping-pong** buffering.



Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	2
FIFO	0	-	35	172
Instance	-	15	837	1103
Memory	24576	-	192	0
Multiplexer	-	-	-	20
Register	-	-	10	-
Total	24576	15	1074	1297
Available	280	220	106400	53200
Utilization (%)	8777	6	1	2

Figure 4-17: Resource Estimate with DATAFLOW Directive

When DATAFLOW optimization is performed, memory buffers are automatically inserted between the functions to ensure that the next function can begin operation before the previous function has finished. The default memory buffers are ping-pong buffers sized to fully accommodate the largest producer or consumer array.

The Vivado HLS tool allows the memory buffers to be the default ping-pong buffers or FIFOs. Since this design has data accesses that are fully sequential, FIFOs can be used. Another advantage to using FIFOs is that the size of the FIFOs can be directly controlled (not possible in ping-pong buffers where random accesses are allowed).

The memory buffers type can be selected by using the Vivado HLS tool Configuration command.

5-5. Apply the DATAFLOW configuration command and select the FIFO for the memory buffers.

5-5-1. Select **Solution** > **Solution Settings** or click the **Solution Settings** icon ( ) to access the configuration command settings.

5-5-2. In the Configuration Settings dialog box, select General and click **Add**.

5-5-3. Select **config_dataflow** from the Command drop-down list.

5-5-4. Select the **fifo** from the default_channel drop-down list.

5-5-5. Enter **2** in the fifo_depth field and click **OK**.

5-5-6. Click **OK**.

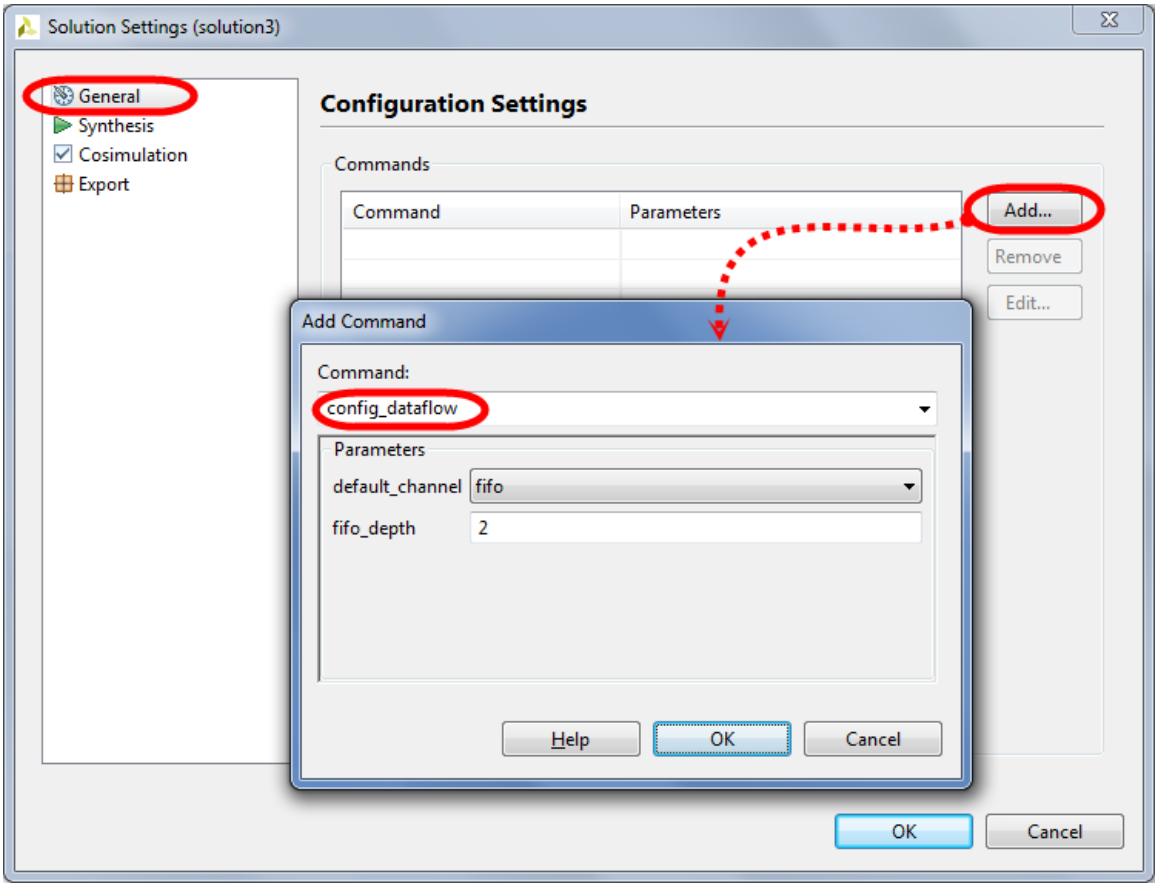


Figure 4-18: Selecting the Dataflow Configuration Command and FIFO as the Buffer

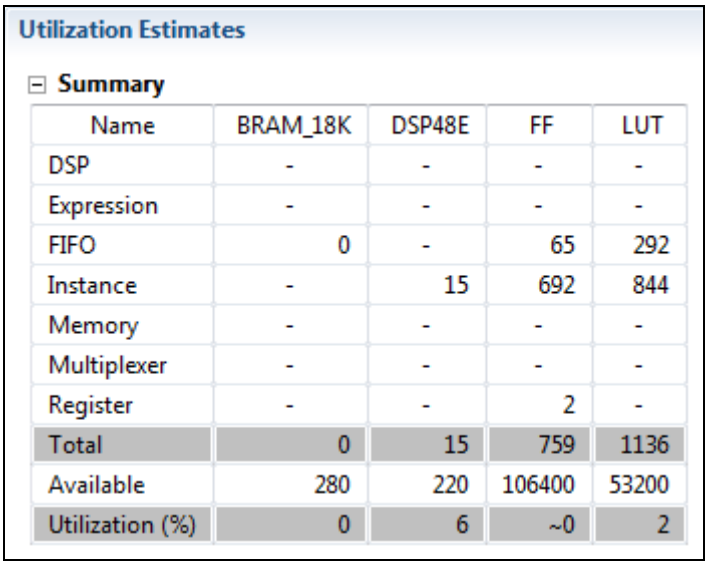
5-6. Synthesize the Vivado HLS design.

If you do not recall how to perform this task, refer to the "Synthesizing the Vivado HLS Design" section under Vivado HLS Operations in the *Lab Reference Guide*.

5-7. View the Synthesis report.

You should see that the Synthesis report in the Information window opens automatically when synthesis is completed.

Note that the performance parameter has not changed; however, resource estimates show that the design is not using any block RAM, and other resource usage (flip-flops, LUTs) has also reduced.



Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	0	-	65	292
Instance	-	15	692	844
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	2	-
Total	0	15	759	1136
Available	280	220	106400	53200
Utilization (%)	0	6	~0	2

Figure 4-19: Resource Estimation After Dataflow Configuration Command

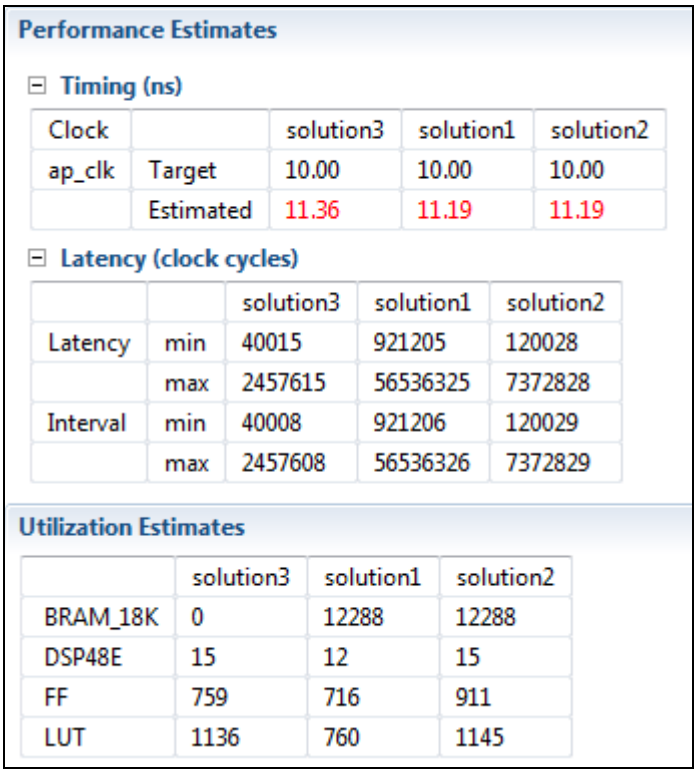
5-8. Compare the performance and resource estimates of all the solutions.

5-8-1. Select **Project** > **Compare Reports** or click the **Compare Reports** icon ( ) to compare the results of all three solutions.

5-8-2. Select **solution1**, **solution2**, and **solution3** from the Available solutions section.

5-8-3. Click **Add**.

5-8-4. Click **OK**.



Performance Estimates

Timing (ns)

Clock		solution3	solution1	solution2
ap_clk	Target	10.00	10.00	10.00
	Estimated	11.36	11.19	11.19

Latency (clock cycles)

		solution3	solution1	solution2
Latency	min	40015	921205	120028
	max	2457615	56536325	7372828
Interval	min	40008	921206	120029
	max	2457608	56536326	7372829

Utilization Estimates

	solution3	solution1	solution2
BRAM_18K	0	12288	12288
DSP48E	15	12	15
FF	759	716	911
LUT	1136	760	1145

Figure 4-20: Comparison of Performance and Resource Estimates for All Solutions

Exporting and Implementing the Design — Step 6

In this step, you will export the RTL as an IP core to be used with the top-level design.

6-1. Export the RTL, selecting Verilog as the language.

6-1-1. Select **Solution** > **Export RTL**.

Alternatively, you can click the  toolbar button.

6-1-2. Ensure that **IP Catalog** is selected from the Format Selection drop-down list.

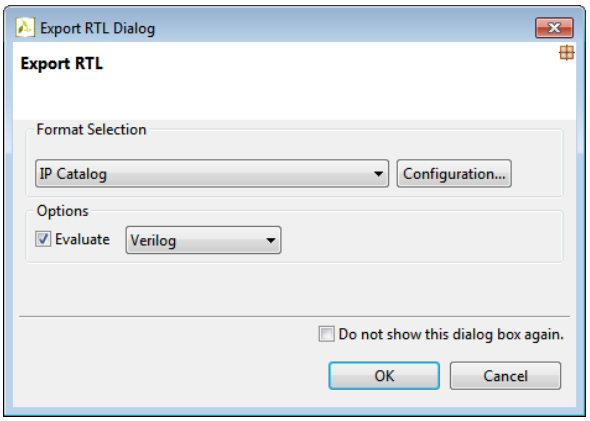


Figure 4-21: Export RTL Dialog Box

6-1-3. Make certain that the **Evaluate** option is selected and that **Verilog** is selected as the RTL to be evaluated.

This will perform Vivado RTL synthesis and implementation on the generated IP. Implementation is run to evaluate and provide confidence that the RTL will meet its estimated timing and area goals and that these results are not included as part of the exported package.

6-1-4. Click **OK** in the Export RTL dialog box.

You can observe the progress in the Console tab. When the run is complete, the Implementation report is displayed in the Information pane.

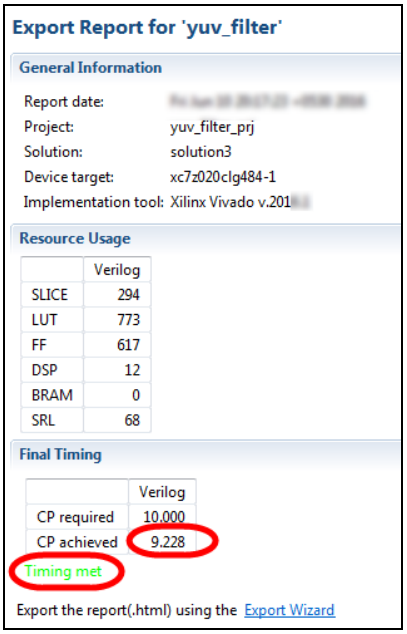


Figure 4-22: Export Report

6-1-5. Select **File** > **Exit** to close the Vivado HLS tool.

Summary

In this lab, you learned that even though a design could not be pipelined at the top level, a strategy of pipelining the individual loops and then using dataflow optimization to make the functions operate in parallel was able to achieve the same high throughput, processing one pixel per clock.

When the DATAFLOW directive is applied, the default memory buffers (of the ping-pong type) are automatically inserted between the functions. Using the fact that the design used only sequential (streaming) data accesses allowed the costly memory buffers associated with dataflow optimization to be replaced with simple two element FIFOs via the Dataflow command configuration.

Answers

1. Looking at the report, fill in the table below.

Estimated clock period	11.19
Worst case latency	56536325
Number of DSP48E used	12
Number of BRAMs used	12288
Number of FFs used	716
Number of LUTs used	760

Performance and Utilization Estimates After the TRIPCOUNT Directive

2. Looking at the report, fill in the table below.

Estimated clock period	8.34
Worst case latency	19664641
Number of DSP48E used	5
Number of FFs used	198
Number of LUTs used	253

Performance and Utilization Estimates for the *rgb2yuv* Function

3. Looking at the report, fill in the table below.

Estimated clock period	11.19
Worst case latency	19664641
Number of DSP48E used	4
Number of FFs used	203
Number of LUTs used	238

Performance and Utilization Estimates for the *yuv2rgb* Function

Lab 5: Implementing Arrays as RTL Interfaces

2016.1

Abstract

This lab demonstrates how to implement arrays as RTL interfaces.

This lab should take approximately 45 minutes.

Objectives

After completing this lab, you will be able to:

- Identify the default I/O protocol for ports
- Select the dual-port RAM and FIFO interfaces for input and output arrays
- Partition an array interface into an arbitrary number of ports
- Partition an array interface into individual ports

Introduction

This lab demonstrates how array arguments on the top-level function interface can be implemented as a number of different types of RTL ports.

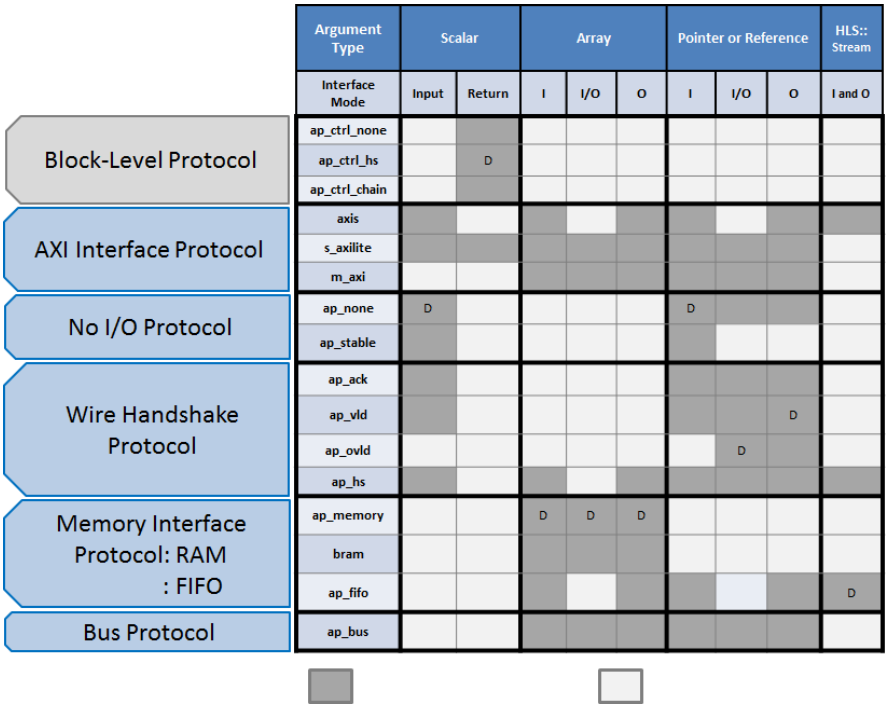


Figure 5-1: I/O Protocol Types

Array arguments are implemented by default as an *ap_memory* interface. This is a standard block RAM interface with data, address, chip-enable, and write-enable ports. An *ap_memory* interface can be implemented as a single-port of dual-port interface.

If the Vivado® HLS tool can determine that a using a dual-port interface will reduce the initial interval, it will automatically implement a dual-port interface. The RESOURCE directive is used to specify the memory resource and if this directive is specified on the array with a single-port block RAM, a single-port interface will be implemented.

Conversely, if a dual-port interface is specified using the RESOURCE directive and the Vivado HLS tool determines that this interface provides no benefit, it will automatically implement a single-port interface. The *bram* interface mode is functionally identical to the *ap_memory* interface.

The only difference is how the ports are implemented when the design is used in the Vivado IP integrator:

- An *ap_memory* interface is displayed as multiple and separate ports.
- A *bram* interface is displayed as a single grouped port that can be connected to a Xilinx block RAM using a single point-to-point connection.

If the array is accessed in a sequential manner an *ap_fifo* interface can be used. As with the *ap_hs* interface, the Vivado HLS tool will halt if it determines that the data access is not sequential, report a warning if it cannot determine if the access is sequential, or issue no message if it determines the access is sequential. The *ap_fifo* interface can only be used for reading or writing, not both.

General Flow

Creating a Vivado HLS Tool Project Step 1

In this step, you will create a new Vivado® HLS tool project, add source files, and provide solution settings for the default solution using the Vivado HLS tool command prompt.

1-1. Launch the Vivado HLS tool command prompt and change the directory to the *array_interfaces* working directory.

1-1-1. Select **Start** > **All Programs** > **Xilinx Design Tools** > **Vivado 2016.1** > **Vivado HLS** > **Vivado HLS 2016.1 Command Prompt** to launch the Vivado HLS tool command prompt.

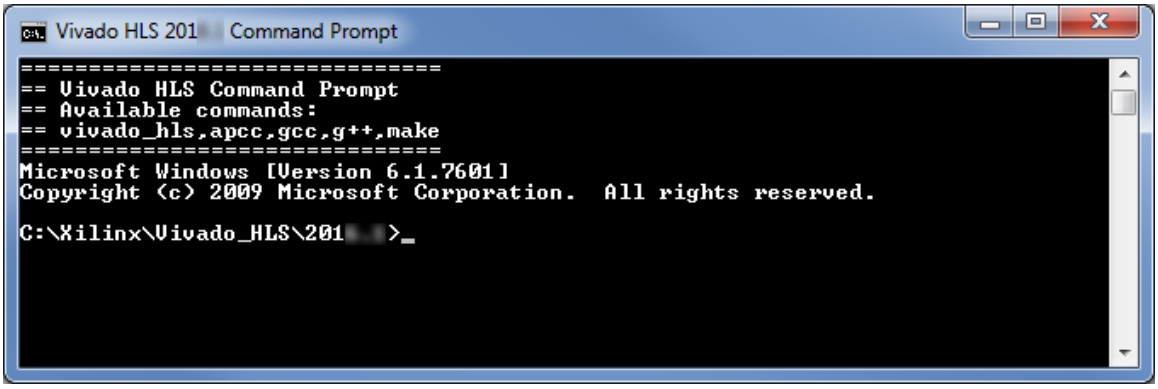


Figure 5-2: Vivado HLS Command Prompt

1-1-2. Enter the following command to change the directory to the current working directory:

```
cd C:\training\hls\labs\array_interfaces
```

1-2. Review and run the *run_hls.tcl* file to create the project and open the created project in the Vivado HLS tool GUI.

1-2-1. Using any of the text editors available under the `C:\training\hls\labs\array_interfaces` directory, open the **`run_hls.tcl`** file.

1-2-2. Review the following sections:

- Project name *(array_io_prj)*
- Source files *(array_io.c)* and testbench *(array_io_test.c* and *result.golden.dat)* added to the project
- Creating a solution *(solution1)*
- Selecting the Xilinx device *(xc7k160tfbg484-1)* and clock period *(4)*
- Running the C simulation

1-2-3. Close the **`run_hls.tcl`** file.

1-2-4. In the Vivado HLS tool command prompt, enter the following command to run the `run_hls.tcl` file:

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

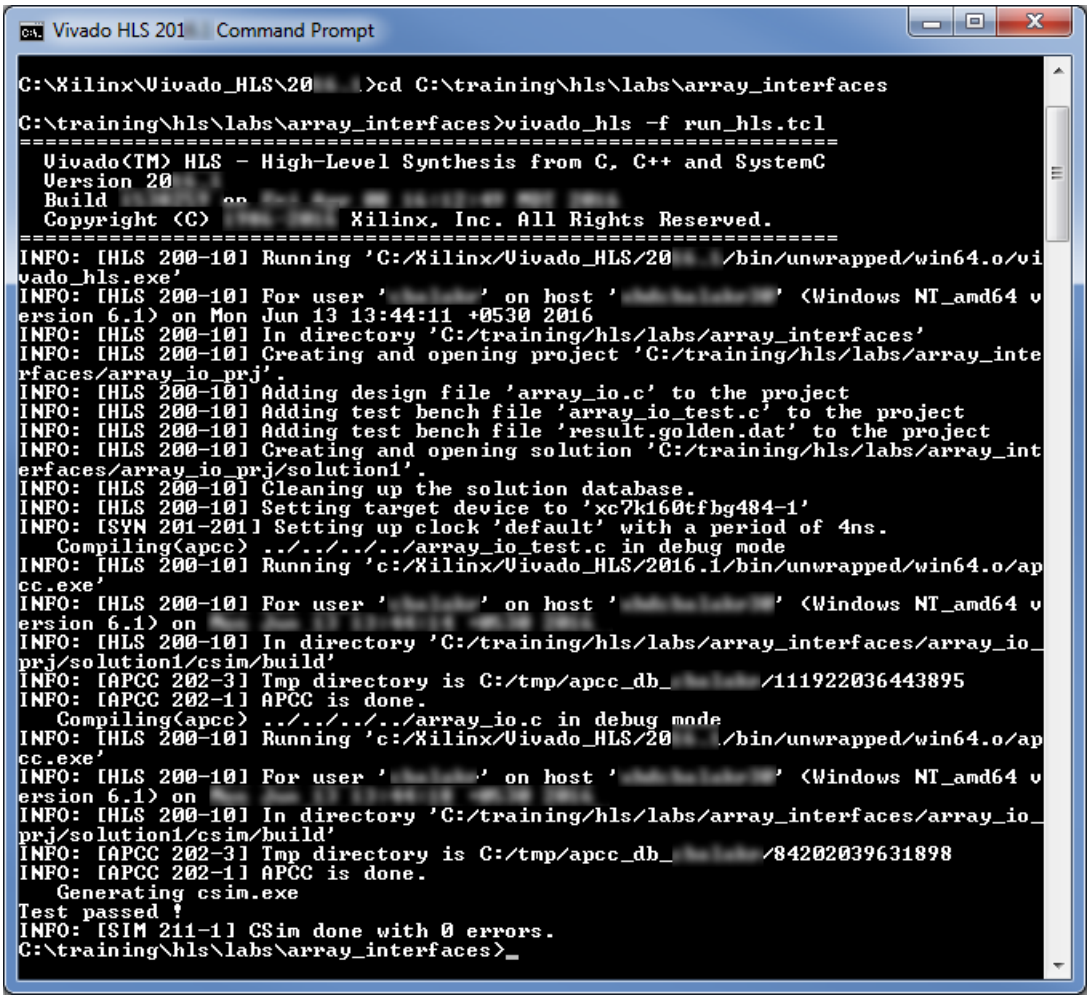


Figure 5-3: Creating the Project and Running the C Simulation

1-2-5. Enter the following command to open the Vivado HLS tool project:

```
vivado_hls -p array_io_prj
```

The Vivado HLS tool GUI opens.

1-2-6. Expand **array_io_prj** > **Source** in the Project Explorer pane.

1-2-7. Double-click the **array_io.c** file to open it.

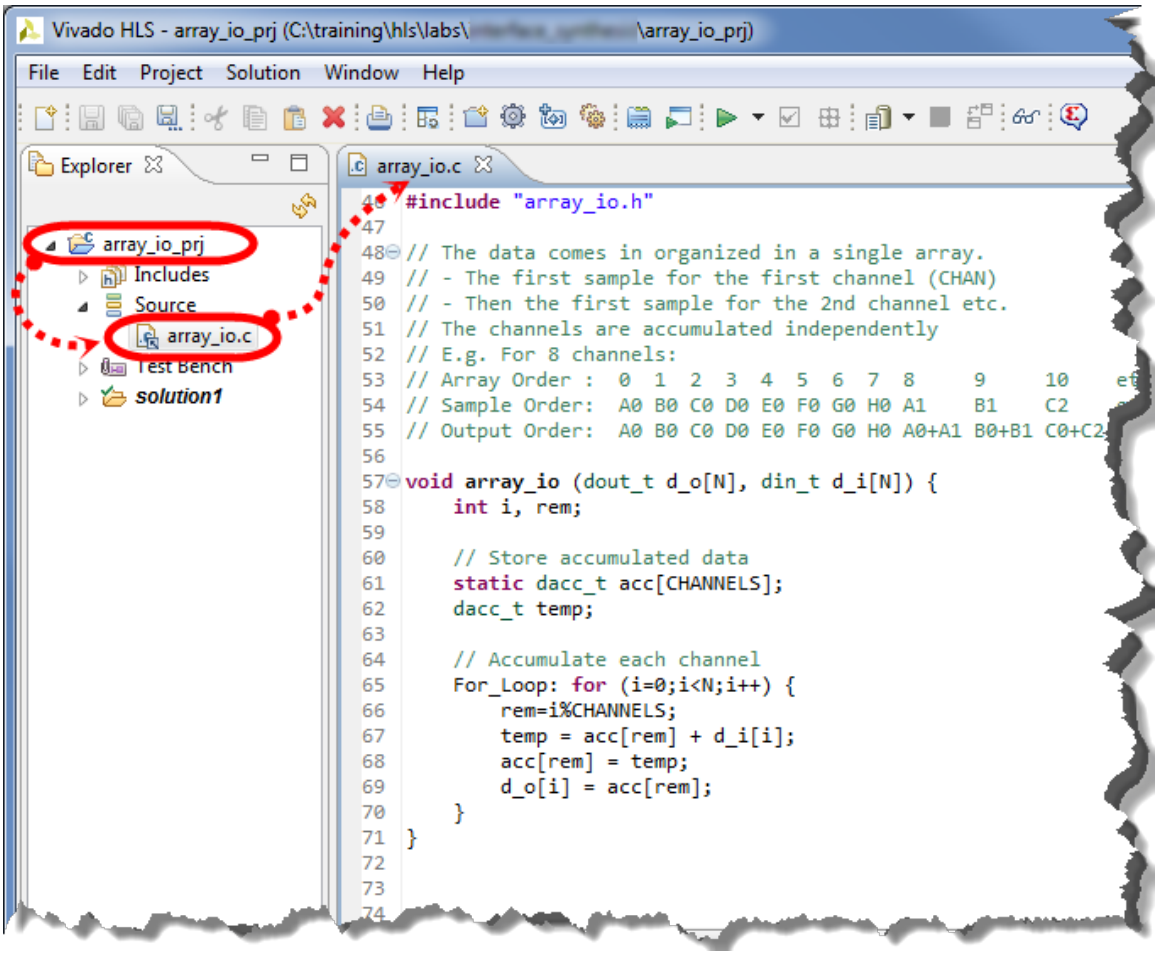


Figure 5-4: Opening the Source File

1-2-8. Review the source file.

This design has an input array and an output array. The data in the input array is ordered as channels and on how the channels are accumulated.

You can review the test bench and the input and output data in the `result.golden.dat` file.

Synthesizing the Design Step 2

In this step, you will synthesize the design by using Vivado HLS tool defaults and review how the array ports are synthesized to RAM ports.

2-1. Synthesize the design.

2-1-1. Select **Solution** > **Run C Synthesis** > **Active Solution** or click the **Run Synthesis** icon in the menu bar.

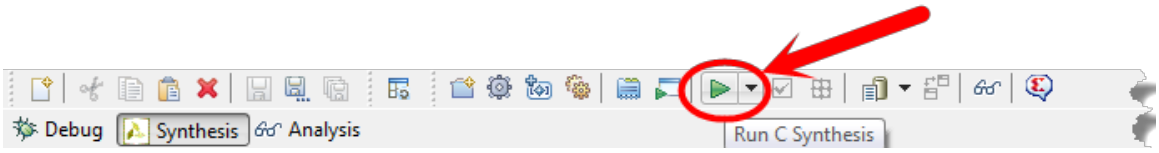


Figure 5-5: Launching Synthesis

This option synthesizes the currently selected solution.

All solutions (or selected solutions) can be synthesized by using the drop-down menu next to the synthesis icon. You can synthesize all solutions or synthesize selected solutions in addition to the default.

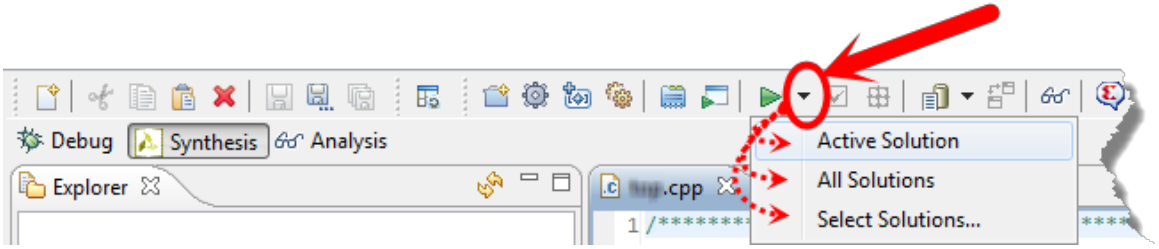


Figure 5-6: Options for What to Synthesize

When the synthesis completes, the Synthesis report will be displayed in the Information pane.

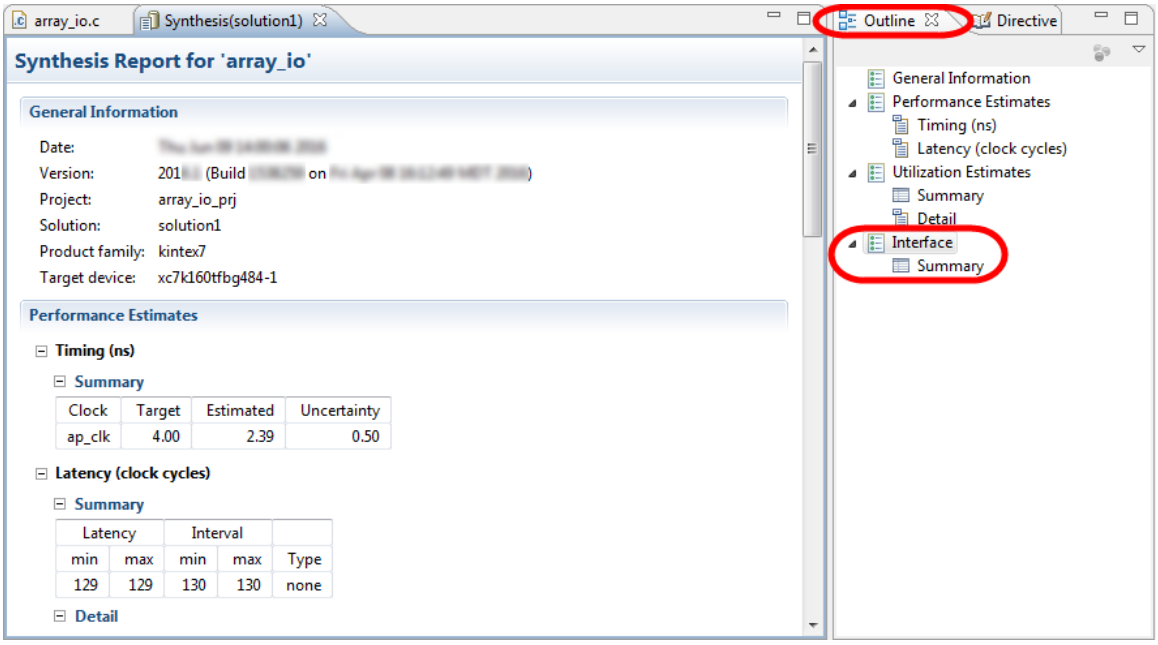


Figure 5-7: Synthesis Report

The Synthesis report shows the performance, area estimates, and interface summary in the design.

2-1-2. In the Outline pane, select **Interface** to review the Interface Summary (shown in the above figure).

Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	array_io	return value
ap_rst	in	1	ap_ctrl_hs	array_io	return value
ap_start	in	1	ap_ctrl_hs	array_io	return value
ap_done	out	1	ap_ctrl_hs	array_io	return value
ap_idle	out	1	ap_ctrl_hs	array_io	return value
ap_ready	out	1	ap_ctrl_hs	array_io	return value
d_o_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 5-8: Interface Summary Report

The Interface Summary shows how array arguments in the C source are by default synthesized into RTL RAM ports:

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

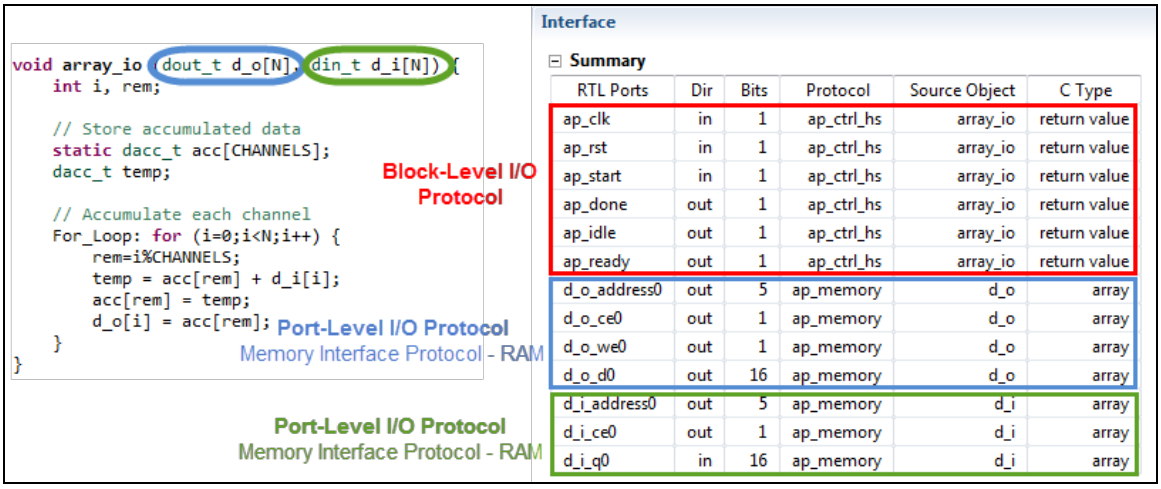


Figure 5-9: Memory Interface Protocol – RAM

Question 1

What is the width of the data ports and address ports?

Using Dual-Port RAM and FIFO Interfaces Step 3

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

When loops are *rolled*, synthesis creates the logic for one iteration of the loop, and the RTL design executes this logic for each iteration of the loop in sequence. Using optimization directives, you can *unroll* loops, which allows all iterations to occur in parallel.

By default (as shown in the previous step), the tool created a single-port RAM interface because the *for* loop in the source code is by default left **rolled**.

Each iteration of the loop is executed in turn:

- Read the input port (*d_i*).
- Read the accumulated result (*acc[rem]*) from the internal RAM.
- Sum the accumulated and new data (*acc[rem] + d_i[i]*) and write into the internal RAM (*acc[rem]*).
- Write the result to the output port (*do[i]*).
- Repeat for the next iteration of the loop.

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

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

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

3-1. Create a new solution and unroll the UNROLL.

3-1-1. Select **Project** > **New Solution** or click **New Solution** icon ( ) from the toolbar.

3-1-2. Leave the options at their default settings.

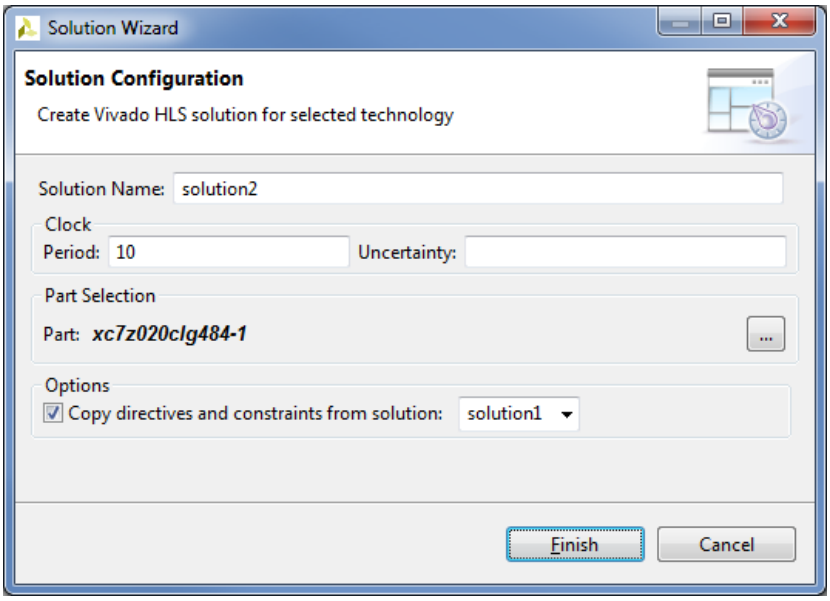


Figure 5-10: Creating a New Solution (solution2)

3-1-3. Click **Finish**.

3-2. Add the directive UNROLL to unroll the *for* loop.

3-2-1. Ensure that the **array_io.c** file is opened and active in the Information pane.

3-2-2. In the Directive tab, right-click **For_Loop** and select **Insert Directive**.

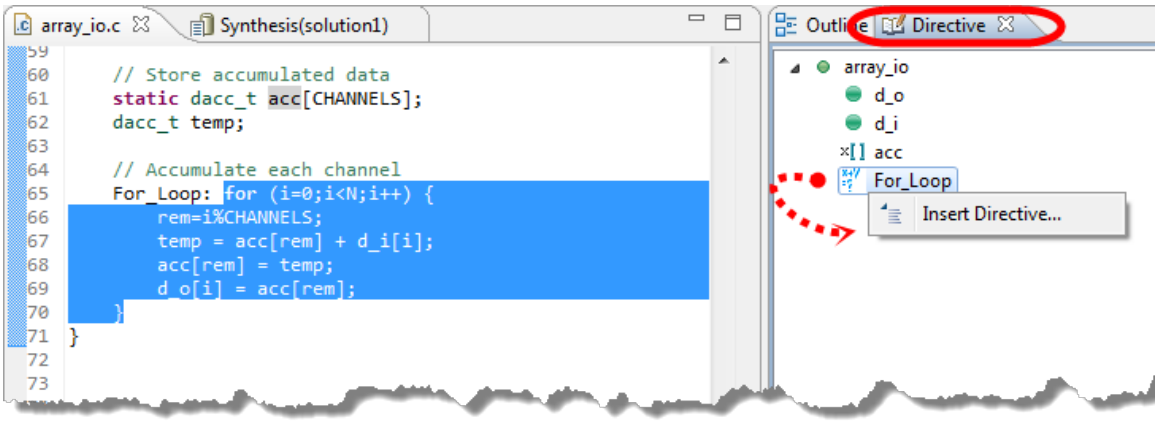


Figure 5-11: Selecting the Directive for For_Loop

3-2-3. Select **UNROLL** from the Directive drop-down list.

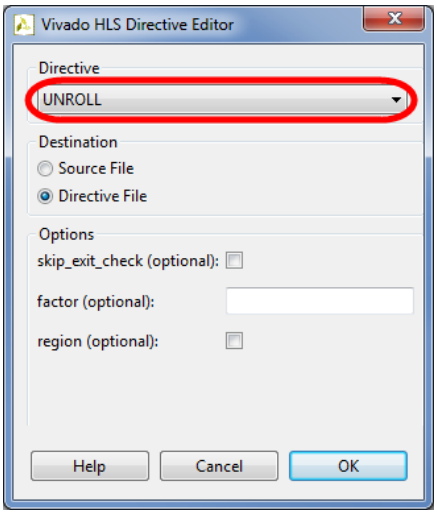


Figure 5-12: Selecting the Directive UNROLL

3-2-4. Click **OK**.

3-3. Specify a dual-port RAM for input reads.

The Resource directive indicates the type of RAM connected to an interface.

3-3-1. Ensure that the **array_io.c** file is opened.

3-3-2. In the Directive tab, right-click **d_i** and select **Insert Directive**.

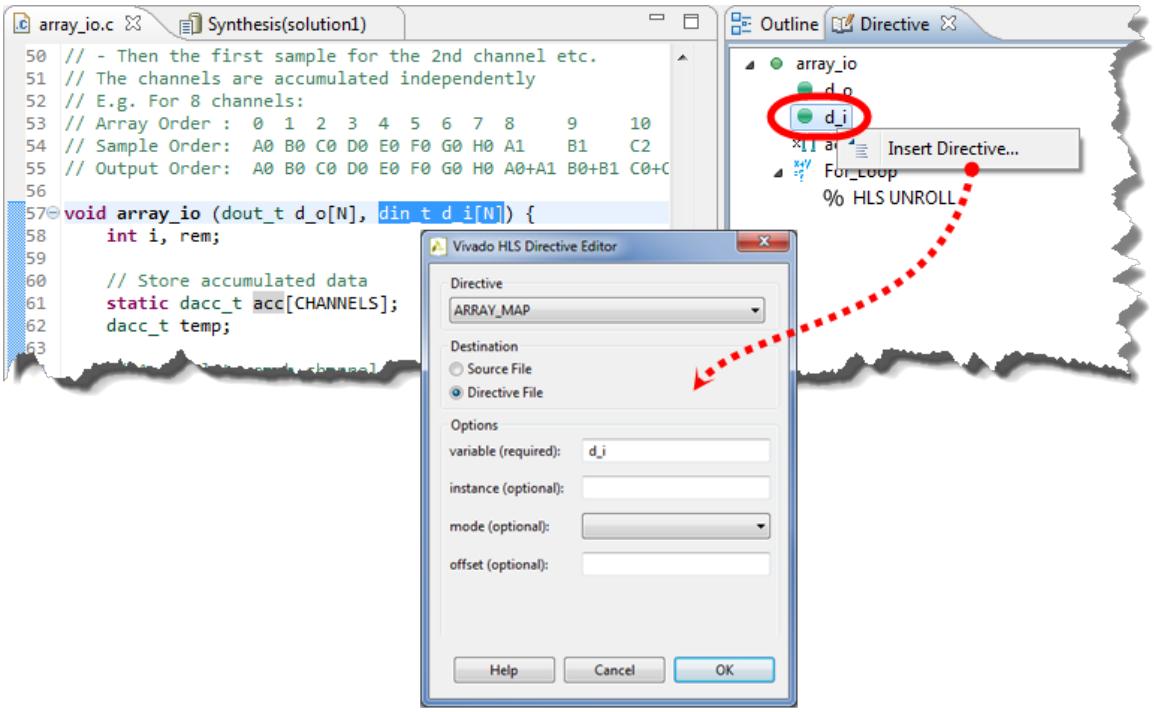


Figure 5-13: Selecting the Directive for the Input Array d_i

3-3-3. Select **RESOURCE** from the Directive drop-down list.

3-3-4. Click the **core (optional)** box.

3-3-5. Select **RAM_2P_BRAM [storage]** from the Vivado HLS core selection dialog box and click **OK**.

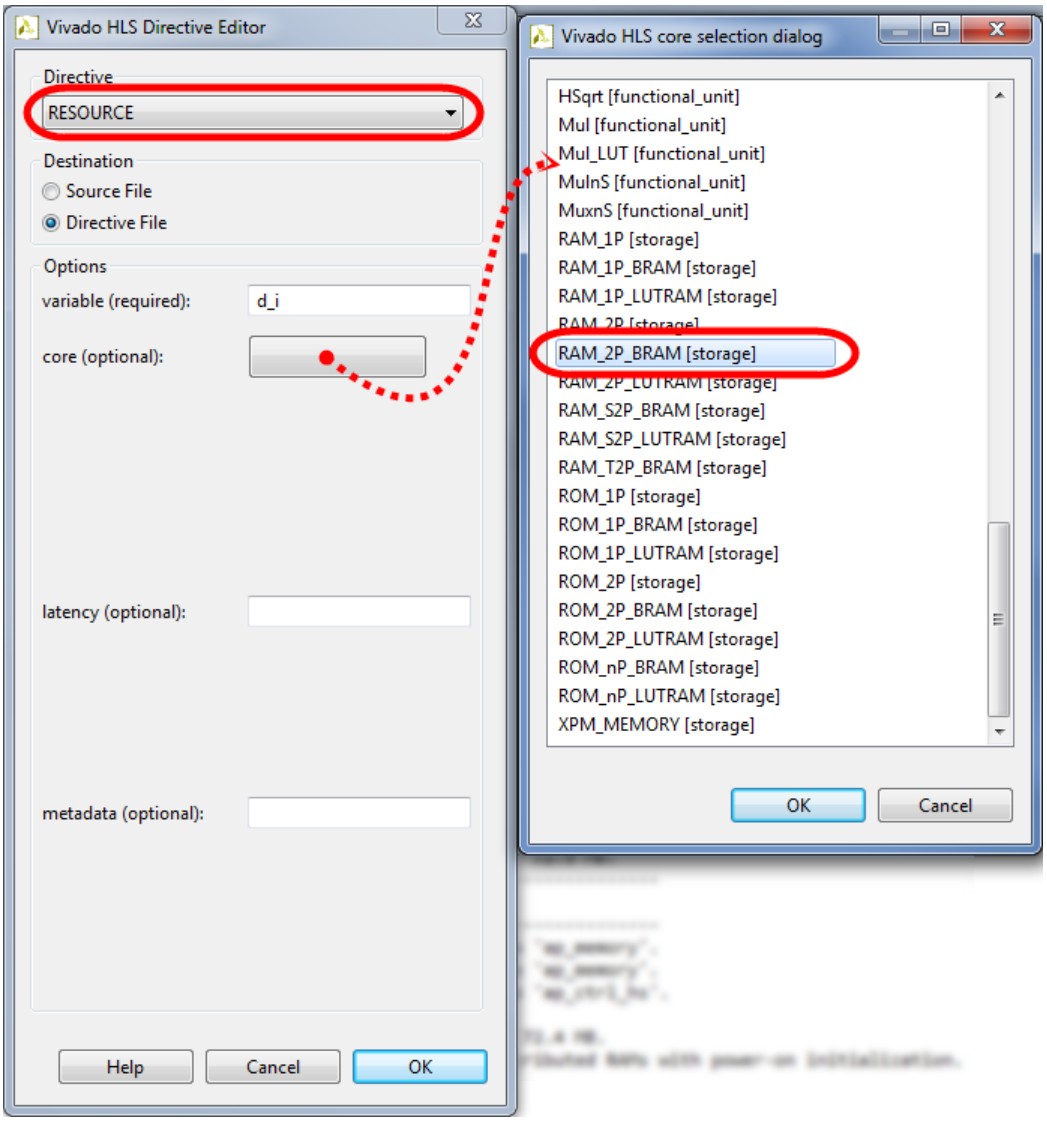


Figure 5-14: Selecting the Dual-Port Block RAM as the Resource for the Input Array d_i

Note that the Resource directive allows you to choose various types of RAM.

3-3-6. Click **OK** in the Vivado HLS Directive Editor dialog box.

3-4. Implement the output port using a FIFO interface.

3-4-1. In the Directive tab, right-click **d_o** and select **Insert Directive**.

3-4-2. Select **INTERFACE** from the Directive drop-down list.

3-4-3. Select **ap_fifo** from the Mode drop-down list.

3-4-4. Click **OK**.

The Directive tab should now look similar to the figure below.

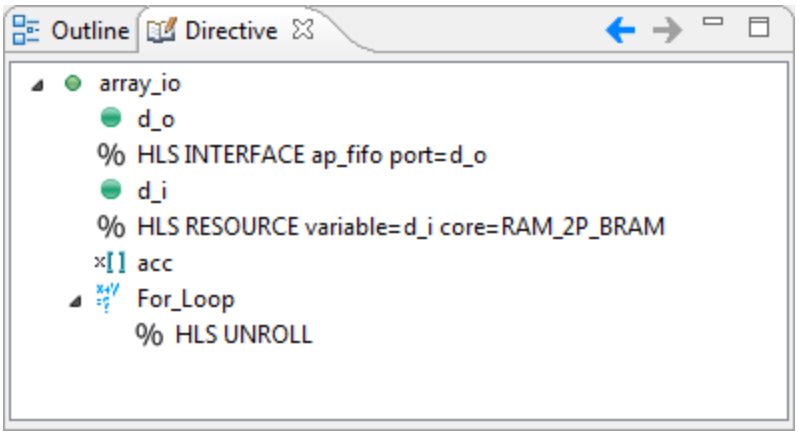


Figure 5-15: Directive Tab with Directives for UNROLL, Dual-Port BRAM, and FIFO Interface

3-5. Synthesize the design.

3-5-1. Select **Solution** > **Run C Synthesis** > **Active Solution** or click the **Run Synthesis** icon in the menu bar.

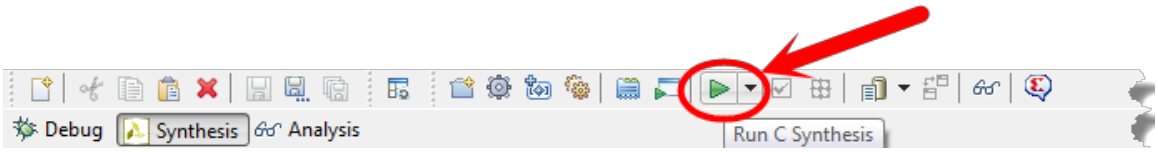


Figure 5-16: Launching Synthesis

This option synthesizes the currently selected solution.

All solutions (or selected solutions) can be synthesized by using the drop-down menu next to the synthesis icon. You can synthesize all solutions or synthesize selected solutions in addition to the default.

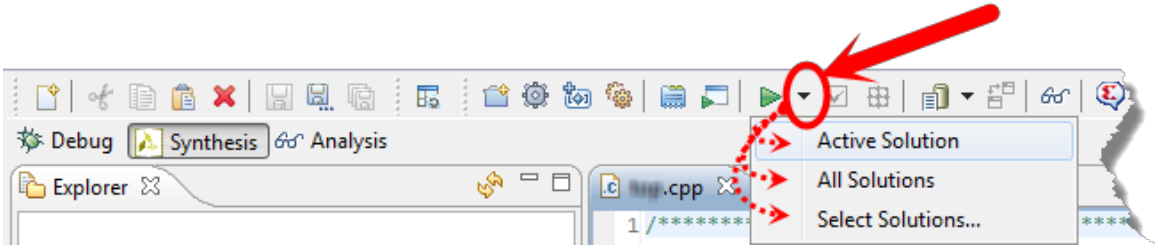


Figure 5-17: Options for What to Synthesize

You can view the synthesis information in the Console tab.

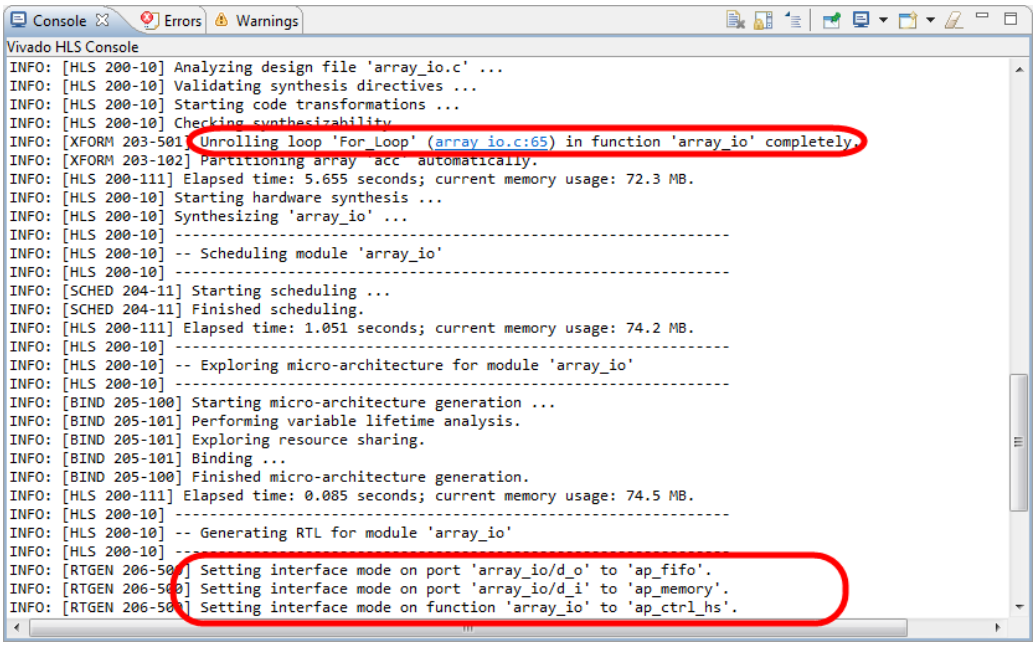
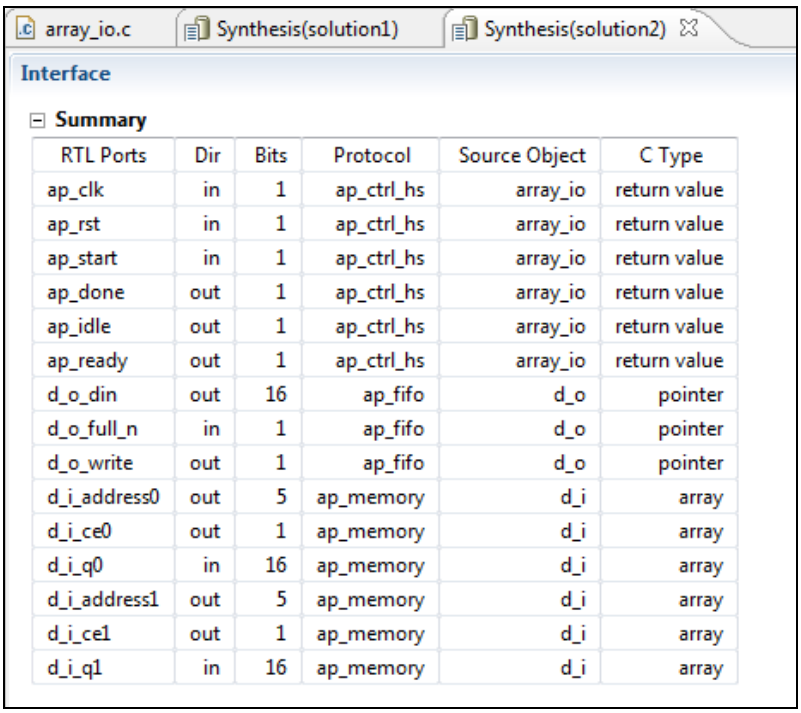


Figure 5-18: Synthesis Information in the Console Tab

When synthesis completes, the Synthesis report will be displayed in the Information pane.

3-5-2. In the Outline pane, select **Interface** to review the Interface Summary.



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

Figure 5-19: Selecting the Interface Summary (Solution2)

The Interface Summary shows how the directives are synthesized into the RTL dual-port block RAM ports and FIFO interface:

- The design has the standard *clock*, *reset*, and block-level I/O ports (*ap_ctrl_hs*).
- The 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.
- The argument *d_i* has been implemented as a dual-port RAM interface.

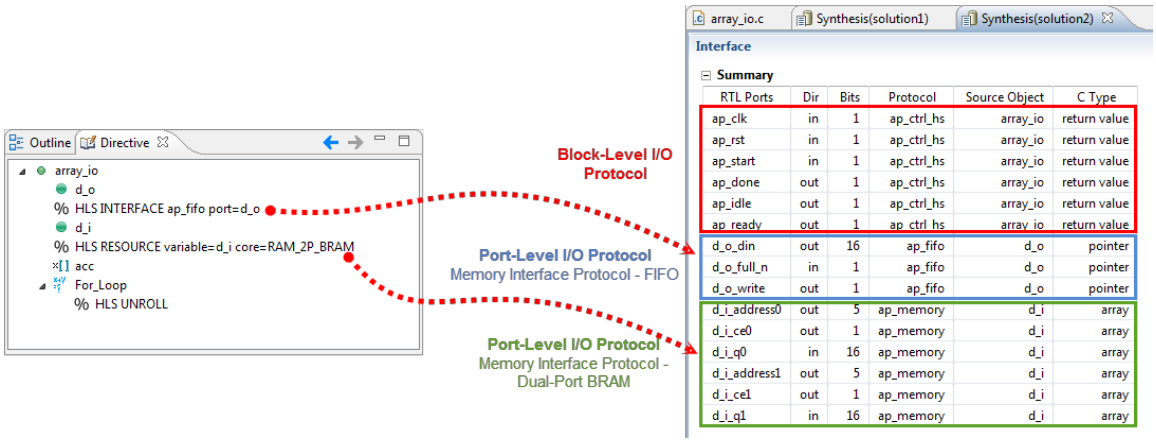


Figure 5-20: Memory Interface Protocol – Dual Port Block RAM and FIFO

By using a dual-port RAM interface, this design can accept input data at twice the rate of the previous design. Because the *for* loop was unrolled, the logic in the loop is able to consume data at this rate.

Unrolling the loops allows more reads to be performed (but creates N copies of the logic). However, with the use of a single-port FIFO interface on the output, the output data rate is the same as before.

Partitioning RAM and FIFO Array Interfaces Step 4

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

4-1. Create a new solution by copying the previous solution (solution2) settings.

4-1-1. Select **Project** > **New Solution** or click the **New Solution** icon ( ) from the toolbar.

4-1-2. Leave the options at their default settings (with *solution2* selected).

4-1-3. Click **Finish**.

4-2. Apply the ARRAY_PARTITION directive to the d_o array.

4-2-1. Select **Project** > **Close Inactive Solution Tabs** to close all inactive solution windows.

4-2-2. Ensure that the **array_io.c** file is open and active in the Information pane.

4-2-3. Expand the **array_io** function in the Directive tab.

4-2-4. In the Directive tab, right-click the **d_o** array of the *array_io* function and select **Insert Directive**.

4-2-5. Select **ARRAY_PARTITION** from the Directive drop-down list.

4-2-6. Select the *type (optional)* as **block**.

4-2-7. Enter **4** in the *factor (optional)* field.

4-2-8. Click **OK**.

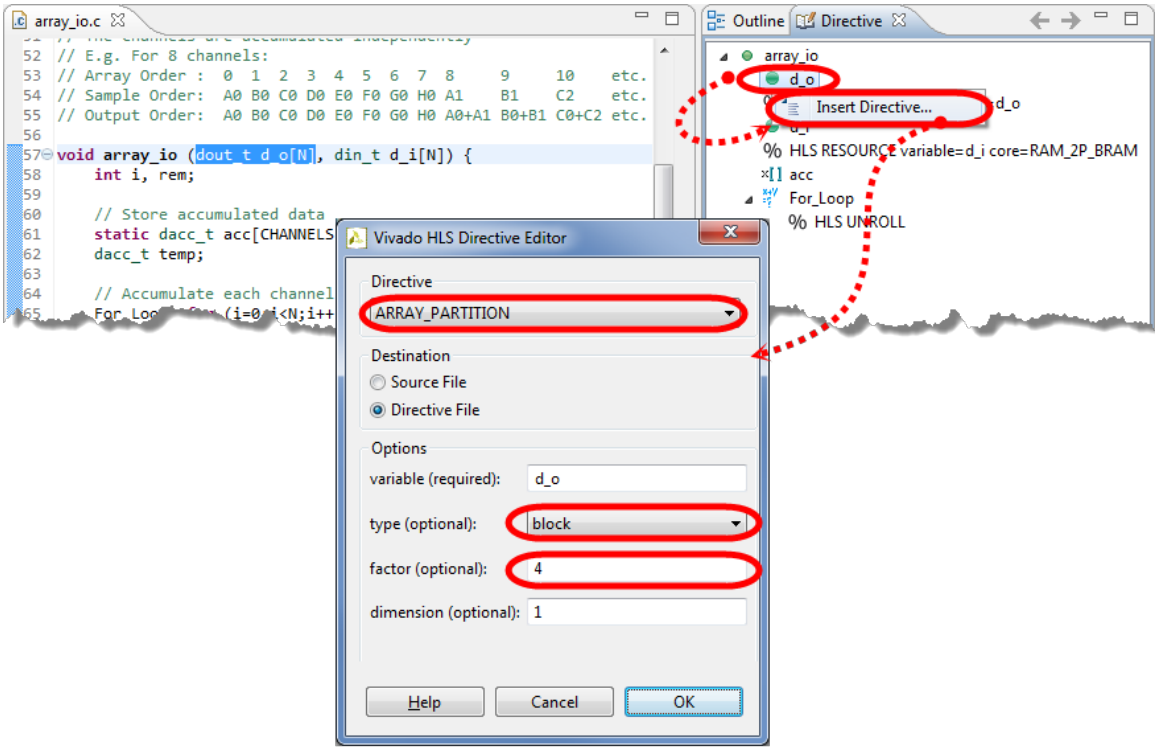


Figure 5-21: Applying the ARRAY_PARTITION Directive for the d_o Array

4-3. Apply the ARRAY_PARTITION directive to the d_i array.

4-3-1. In the Directive tab, right-click the **d_i** array of the *array_io* function and select **Insert Directive**.

4-3-2. Select **ARRAY_PARTITION** from the Directive drop-down list.

4-3-3. Select the *type (optional)* as **block**.

4-3-4. Enter **2** in the *factor (optional)* field.

4-3-5. Click **OK**.

The final view of the ARRAY_PARTITION directives in the Directive tab should look similar to the figure below.

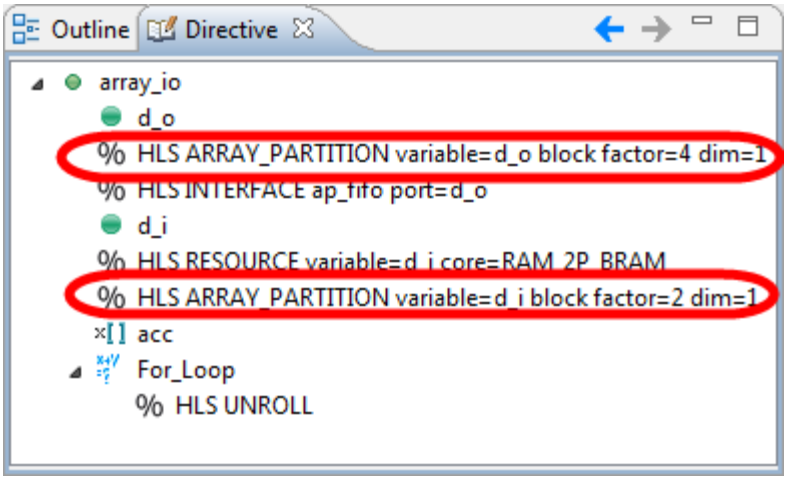


Figure 5-22: Final View of ARRAY_PARTITION Directives

4-4. Synthesize the Vivado HLS design.

If you do not recall how to perform this task, refer to the "Synthesizing the Vivado HLS Design" section under Vivado HLS Operations in the *Lab Reference Guide*.

4-5. View the Synthesis report.

You should see that the Synthesis report in the Information window opens automatically when synthesis is completed.

4-5-1. In the Outline pane, select **Interface** to review the Interface Summary.

The design has the standard clock, reset, and block-level I/O ports.

The array argument **d_o** has been implemented as four separate FIFO interfaces.

The argument **d_i** has been implemented as two separate RAM interfaces, each of which uses a dual-port interface. (If you see four separate RAM interfaces, confirm that the partition factor for d_i is two and not four).

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

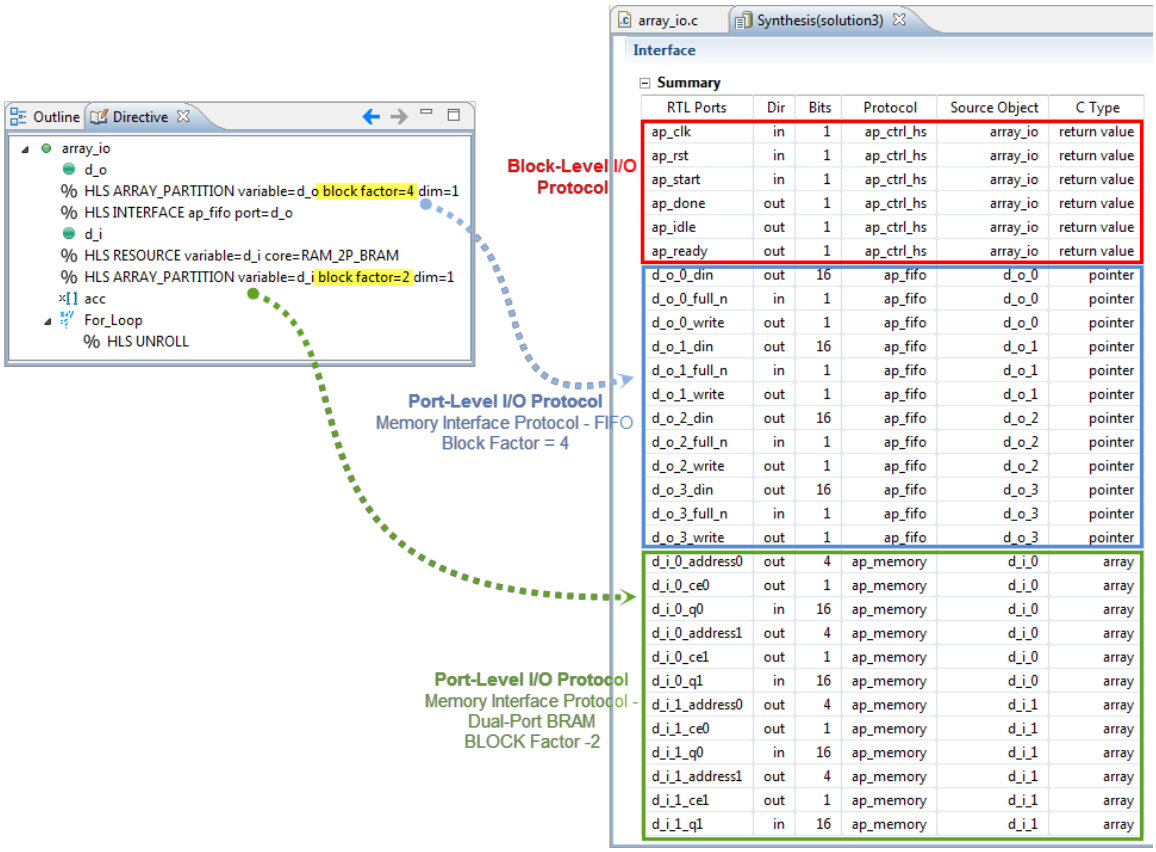


Figure 5-23: Interface Report for Partitioned Interfaces

Fully Partitioning the Array Interfaces Step 5

In this step, you will partition an array interface into individual ports.

5-1. Create a new solution by copying the previous solution (solution3) settings.

5-1-1. Select **Project** > **New Solution** or click the **New Solution** icon ( ) from the toolbar.

5-1-2. Leave the options at their default settings (with *solution3* selected).

5-1-3. Click **Finish**.

5-2. Modify the ARRAY_PARTITION directive of the d_o array to partition fully.

5-2-1. Select **Project** > **Close Inactive Solution Tabs** to close all inactive solution windows.

5-2-2. Ensure that the **array_io.c** file is open and active in the Information pane.

5-2-3. Expand the **array_io** function in the Directive tab.

5-2-4. In the Directive tab, right-click the **ARRAY_PARTITION** of the *d_o* array of the *array_io* function and select **Modify Directive**.

5-2-5. Select **ARRAY_PARTITION** from the Directive drop-down list.

5-2-6. Delete **4** from the *factor (optional)* field.

5-2-7. Select the *type (optional)* as **complete**.

5-2-8. Click **OK**.

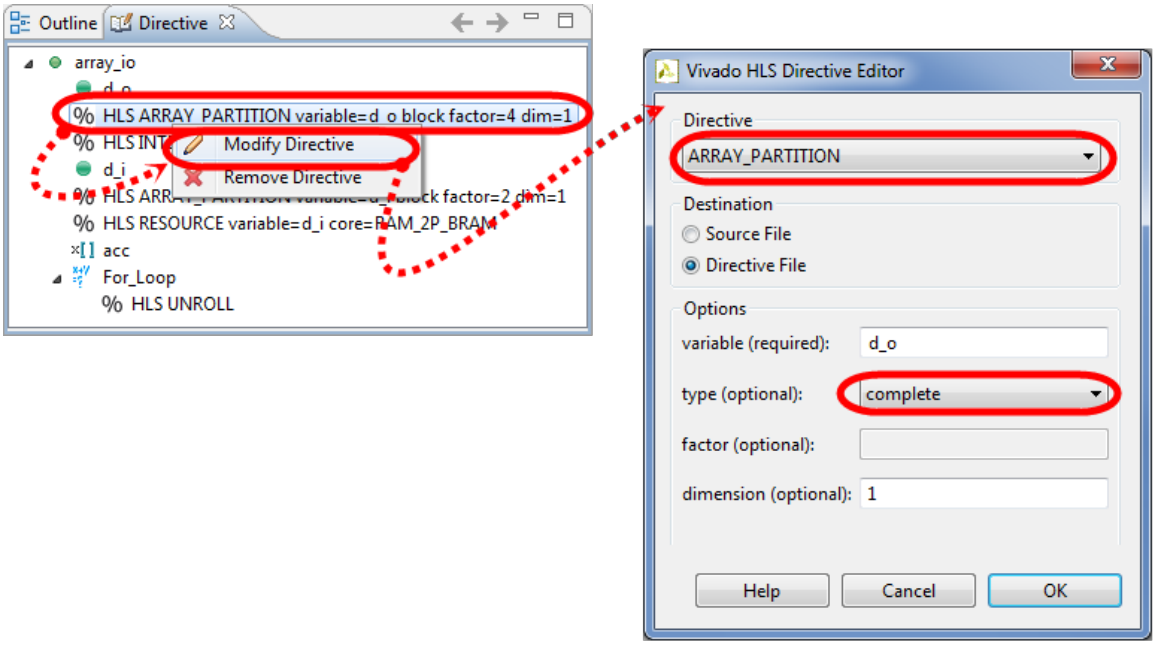


Figure 5-24: Applying Full Partition to the d_o Array

5-2-9. Similarly, repeat the previous tasks to completely partition the *d_i* array.

5-3. Remove the RESOURCE directive of the d_i array.

5-3-1. Right-click the **RESOURCE** directive of *d_i* and select **Remove Directive**.

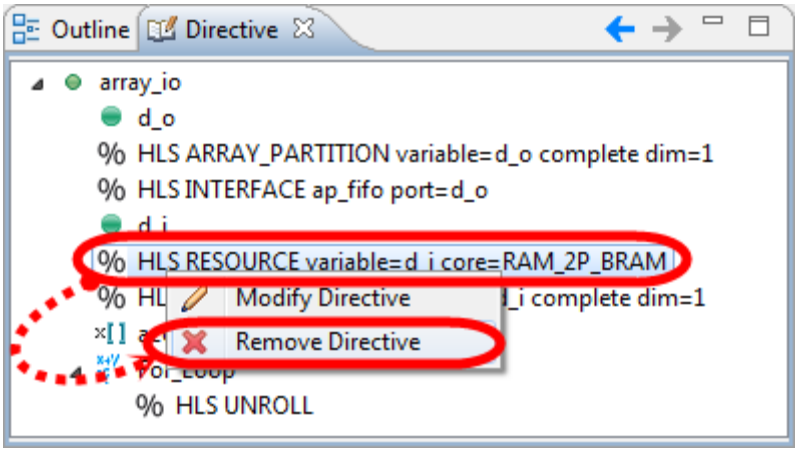


Figure 5-25: Removing the RESOURCE Directive

The final view of the fully partitioned array directives in the Directive tab should look similar to the figure below.

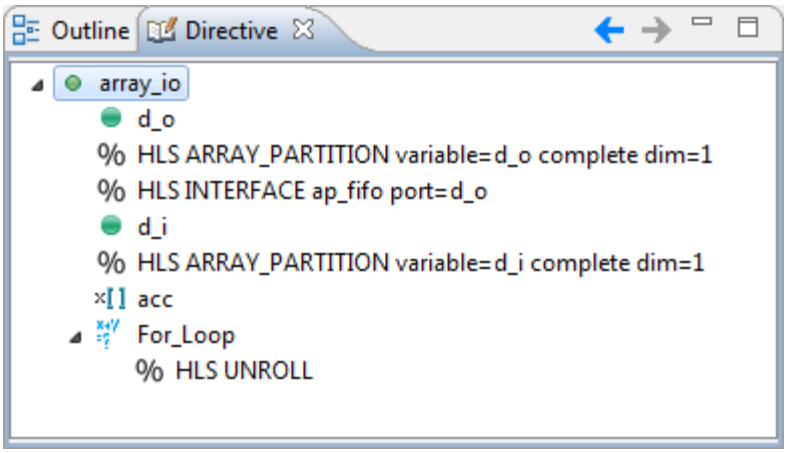


Figure 5-26: Final View of Fully Partitioned Directives

5-4. Synthesize the Vivado HLS design.

If you do not recall how to perform this task, refer to the "Synthesizing the Vivado HLS Design" section under Vivado HLS Operations in the *Lab Reference Guide*.

5-5. View the Synthesis report.

You should see that the Synthesis report in the Information window opens automatically when synthesis is completed.

5-5-1. In the Outline pane, select **Interface** to review the Interface Summary.

The design has the standard clock, reset, and block-level I/O ports.

The array argument **d_o** has been implemented as 32 separate FIFO interfaces.

The argument **d_i** has been implemented as 32 separate scalar ports. Because the default interface for input scalars is not in the I/O protocol, they have the I/O protocol **ap_none**.

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

5-6. Compare all four solutions.

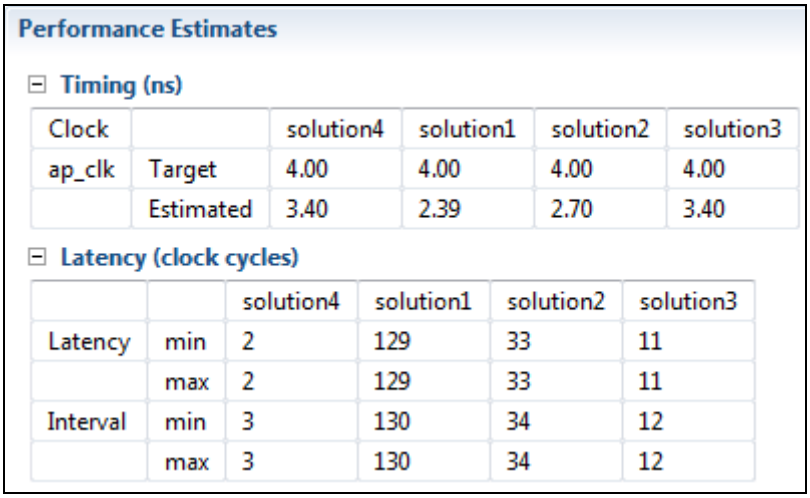
5-6-1. Select **Project** > **Compare Reports** or click the **Compare Reports** icon ( ) to compare the results of all the solutions (*solution1*, *solution2*, *solution3*, and *solution4*).

5-6-2. Select **solution1**, **solution2**, **solution3**, and **solution4** from the Available solutions section.

5-6-3. Click **Add**.

5-6-4. Click **OK**.

You should see the compared results of *solution1*, *solution2*, *solution3*, and *solution4* as shown in the figure below.



Performance Estimates

Timing (ns)

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

Latency (clock cycles)

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

Figure 5-27: Comparison of Performance Estimates of All Solutions

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

5-6-5. Scroll down in the comparison report to view the resource utilization.

Observe that the solutions with more I/O ports (*solution 2*, *solution3*, and *solution4*), allow more parallel processing, but also use more considerably more resources.

Utilization Estimates

	solution4	solution1	solution2	solution3
BRAM_18K	0	0	0	0
DSP48E	0	0	0	0
FF	1155	186	1266	1228
LUT	1058	55	1322	1195

Figure 5-28: Comparison of Utilization Estimates of All Solutions

Summary

In this lab, you have learned how to implement a memory interface protocol such as *ap_memory* and *ap_fifo*. You also used directives to implement an input array as dual-port block RAM. In addition, you used directives to fully partition the array and analyzed the synthesized report.

Answers

1. What is the width of the data ports and address ports?

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

Lab 6: Improving Area and Resource Utilization

2016.1

Abstract

This lab introduces various techniques and directives that can be used in the Vivado® HLS tool to improve design performance as well as area and resource utilization.

This lab should take approximately 60 minutes.

Objectives

After completing this lab, you will be able to:

- Describe the effect of the PIPELINE directive on resource utilization and performance
- Identify the effect of memory partition techniques on resource utilization and performance
- Describe the impact of various directives on resource utilization and performance

Introduction

This lab introduces various techniques and directives which can be used in Vivado HLS to improve design performance as well as area and resource utilization. The design under consideration performs discrete cosine transformation (DCT) on an 8x8 block of data.

In this lab, you will be using a C design to implement a DCT. The function implements a 2D DCT algorithm by first processing each row of the input array via a 1D DCT, then processing the columns of the resulting array through the same 1D DCT. It calls the read_data, dct_2d, and write_data functions.

The read_data function is defined at line 54 and consists of two loops: RD_Loop_Row and RD_Loop_Col. The write_data function is defined at line 66 and consists of two loops to perform writing the result. The dct_2d function, defined at line 23, calls the dct_1d function and performs transpose.

Finally, the dct_1d function, defined at line 4, uses the dct_coeff_table and performs the required function by implementing a basic iterative form of the 1D Type-II DCT algorithm.

The following figure shows the function hierarchy on the left-hand side, the loops in the order they are executed, and the flow of data on the right-hand side.

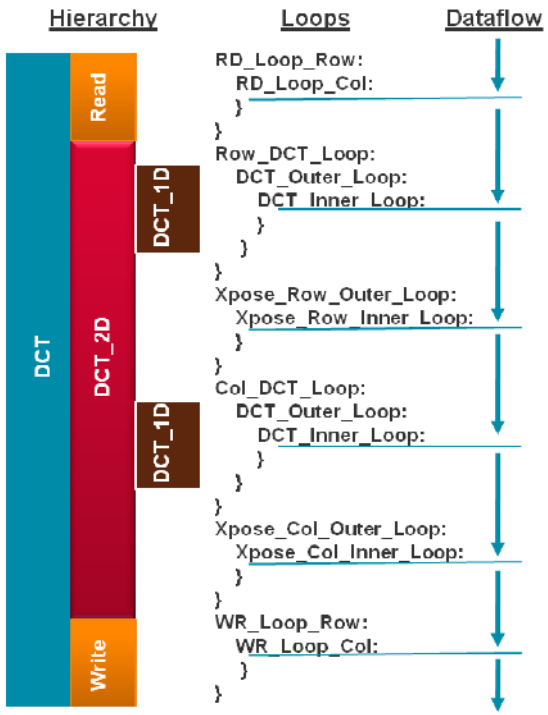
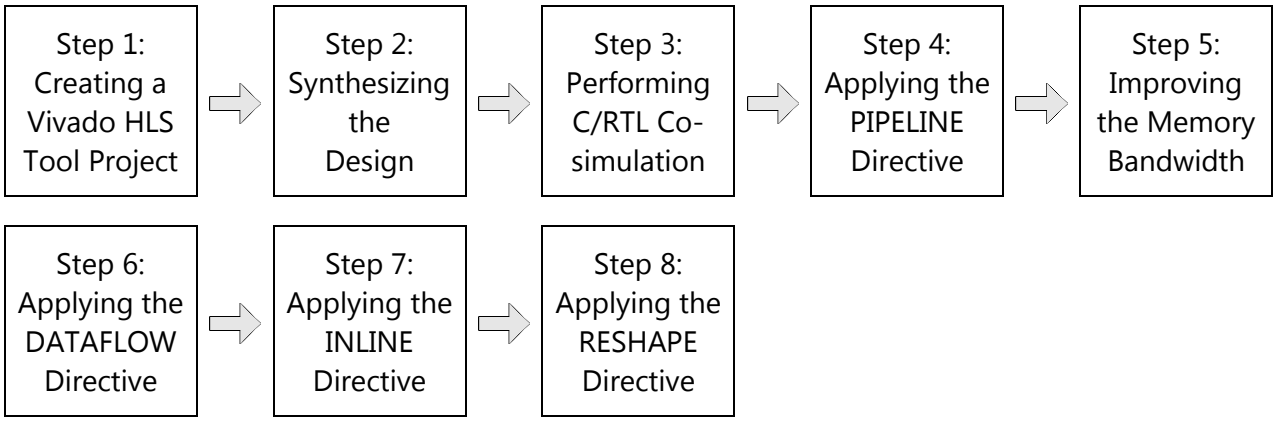


Figure 6-1: Design Hierarchy and Dataflow

General Flow



Creating a Vivado HLS Tool Project Step 1

There are a number of ways to launch the Vivado HLS tool. The two most popular mechanisms are shown here.

1-1. Launch the Vivado HLS tool.

1-1-1. Select **Start** > **All Programs** > **Xilinx Design Tools** > **Vivado 2016.1** > **Vivado HLS** > **Vivado HLS 2016.1**.

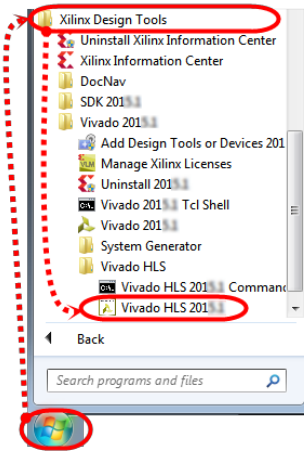


Figure 6-2: Launching the Vivado HLS Tool

-- OR --

Double-click the **Vivado HLS** shortcut icon () on the desktop.

The Vivado HLS tool opens to the Welcome window. From the Welcome window you can create a new project, open examples, and access documentation and examples.

Figure 6-3: Vivado HLS Welcome Screen

Here you will learn to create a new Vivado HLS project from scratch.

1-2. Create a Vivado HLS project named *dct_prj*.

1-2-1. From the Welcome Page, click **Create New Project**.

Figure 6-4: Creating a New Vivado HLS Project

1-3. The Project Configuration dialog box asks for a project name and location.

1-3-1. Enter **dct_prj** in the Project name field (1).

1-3-2. Enter **C:\training\hls\labs\improve_area** in the Location field (2).

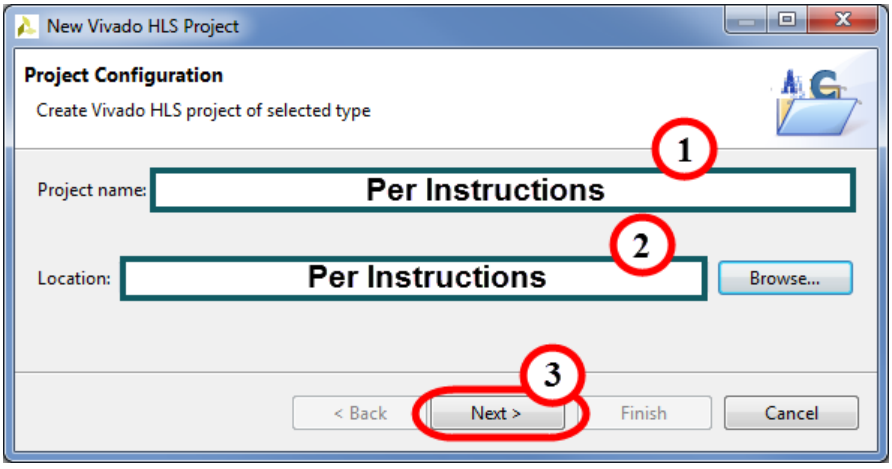


Figure 6-5: Configuring a New HLS Project

1-3-3. Click **Next** (3).

1-4. The Add/Remove Files dialog box opens. Here you will be invited to add existing files or create new sources.

1-4-1. Click **Add Files**.

The Open File dialog box opens.

Note: If do not have existing files at this moment and you want to create new ones, click **New File**.

1-4-2. Browse to *C:\training\hls\labs\improve_area*.

1-4-3. Select **dct.c**.

The Vivado HLS tool automatically adds the working directory (project directory) and any directory that contains C files added to the project to the search path. Hence, header files that reside in these directories are automatically included in the project (no need to explicitly specify them). You must specify the path to all other header files (if any) by clicking the Edit CFLAGS button.

1-4-4. Click **Open** to add these files.

Note that you can add compiler directives specific to each entry at this point.

1-4-5. Click **Browse** next to the Top Function field.

The Select Top function dialog box opens, which lists all the functions available from the specified source files.

1-4-6. Select **dct (dct.c)** from the list.

Note: You can also manually enter the name of the top function in the Top Function field.

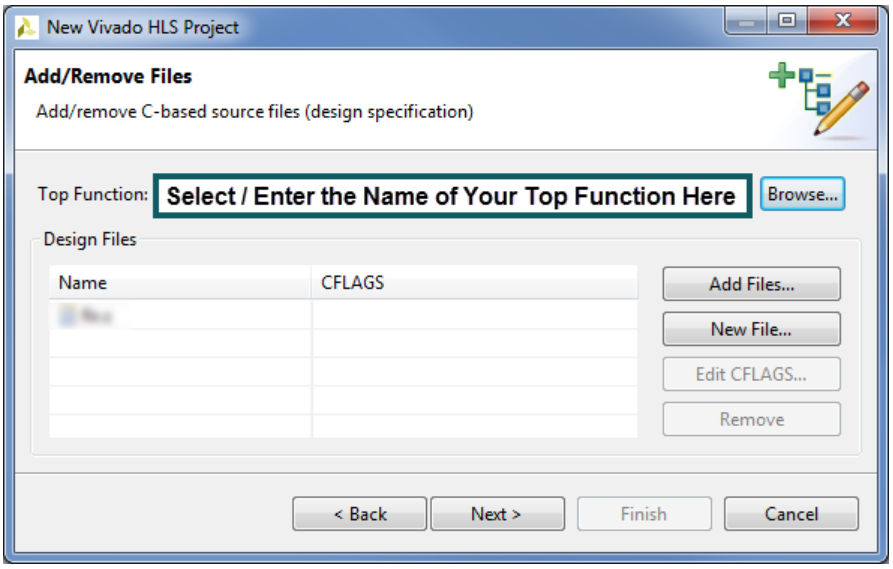


Figure 6-6: Adding Files to a New Vivado HLS Project

1-4-7. Click **Next**.

1-5. Add any existing testbench files.

If you have (or want) any testbench files they can be entered here. Sometimes the testbench is built into the synthesizable file.

1-5-1. Click **Add Files**.

1-5-2. Navigate to *C:\training\hls\labs\improve_area*.

1-5-3. Select **dct_test.c, in.dat and out.golden.dat**.

1-5-4. Click **Open** to add these files.

1-5-5. Click **Next**.

1-6. Finally, it is time to specify some of the physical parameters of the design.

1-6-1. By default, **Solution1** is populated in the Solution Name field.

No changes are required.

1-6-2. Set the clock period to **10**.

You can leave the Uncertainty field blank.

1-6-3. Click the **Browse** button to select a part or board.

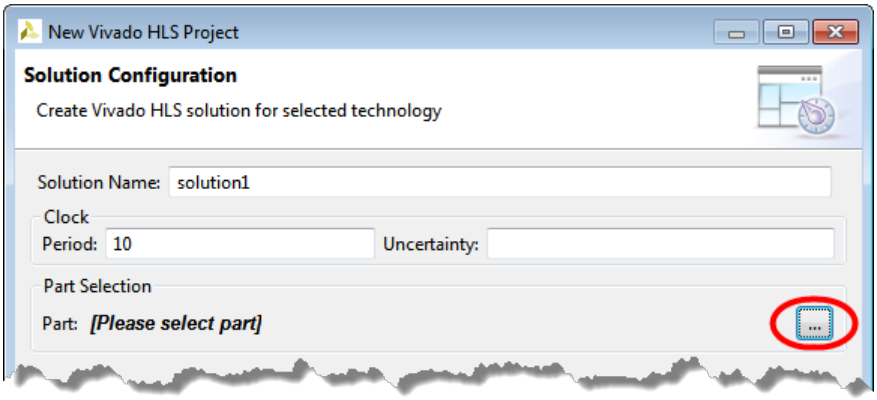


Figure 6-7: Locating the Board Browse Button

1-6-4. Click **Boards** as shown below.

1-6-5. Enter **zynq** in the Search field.

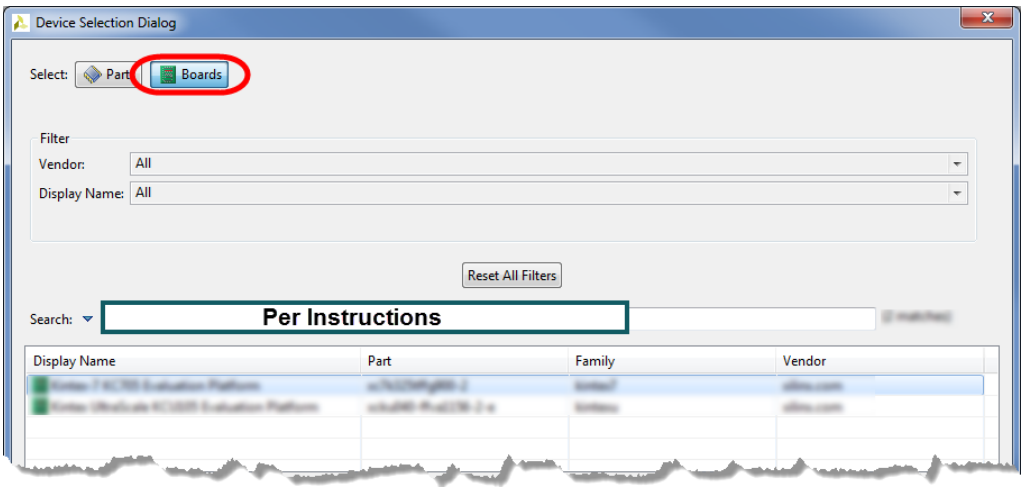


Figure 6-8: Filtering to Quickly Locate Target Platforms

1-6-6. Select **Zynq ZC702 Evaluation Board** from the search list.

1-6-7. Click **OK** to select the board.

1-6-8. Click **Finish**.

You will see the created project in the Explorer tab.

Synthesizing the Design — Step 2

In this step, you will synthesize the design by using Vivado HLS tool defaults and analyze how many resources are utilized to implement the C design.

2-1. Synthesize the design.

2-1-1. Select **Solution** > **Run C Synthesis** > **Active Solution** or click the **Run Synthesis** icon in the menu bar.

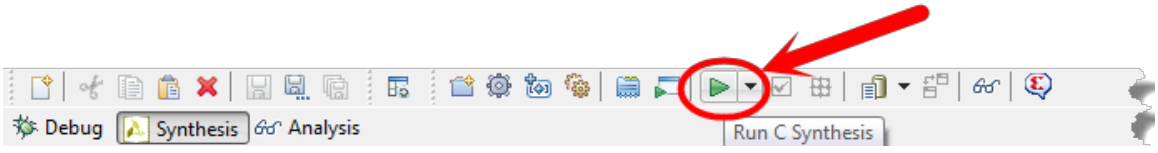


Figure 6-9: Launching Synthesis

This option synthesizes the currently selected solution.

All solutions (or selected solutions) can be synthesized by using the drop-down menu next to the synthesis icon. You can synthesize all solutions or synthesize selected solutions in addition to the default.

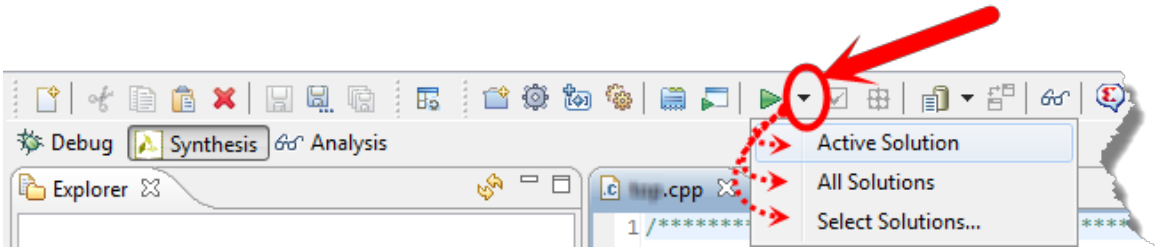


Figure 6-10: Options for What to Synthesize

2-2. View the Synthesis report.

You should see that the Synthesis report in the Information window opens automatically when synthesis is completed.

2-2-1. View the synthesis results in the Console tab.

Note that the Synthesis Report section in the Explorer view only shows the *dct_dtc_1d2_csynth.rpt*, *dct_dct_2d_csynth.rpt*, and *dct.rpt* entries. The *read_data* and *write_data* functions reports are not listed. This is because these two functions are inlined.

2-2-2. Verify this by scrolling up in the Console tab.

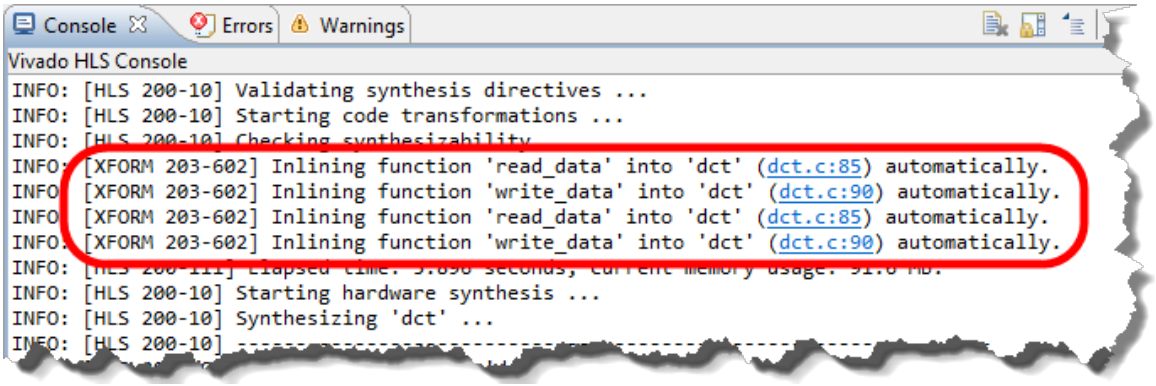
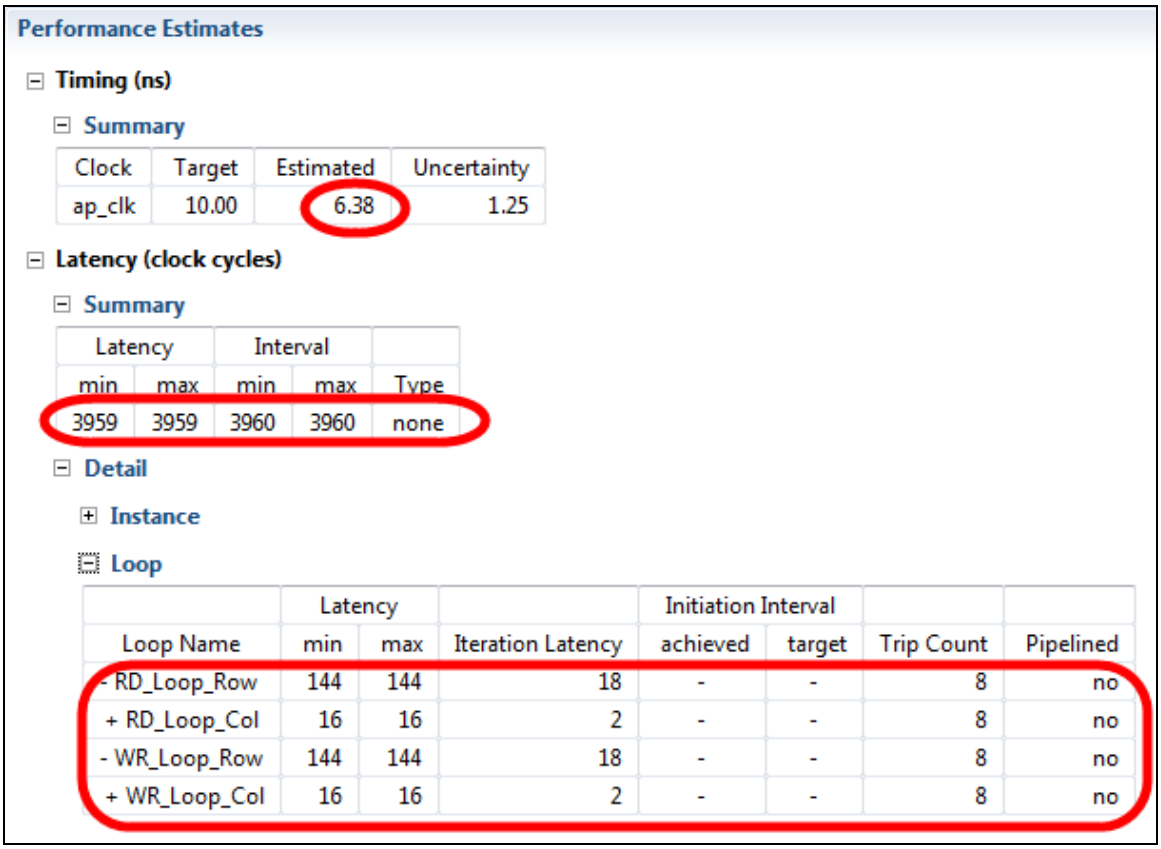


Figure 6-11: Inlining of read_data and write_data Functions

2-2-3. Review the performance and resource estimates as well as the estimated latency.



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	6.38	1.25

Latency (clock cycles)

Summary

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

Detail

Instance

Loop

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

Figure 6-12: Synthesis Report

Question 1

Looking at the report, fill in the table below.

Estimated clock period	
Worst case latency	
Number of DSP48E used	
Number of BRAMs used	
Number of FFs used	
Number of LUTs used	

Performance and Utilization Estimates

2-2-4. View the top-level signals generated by the tools.

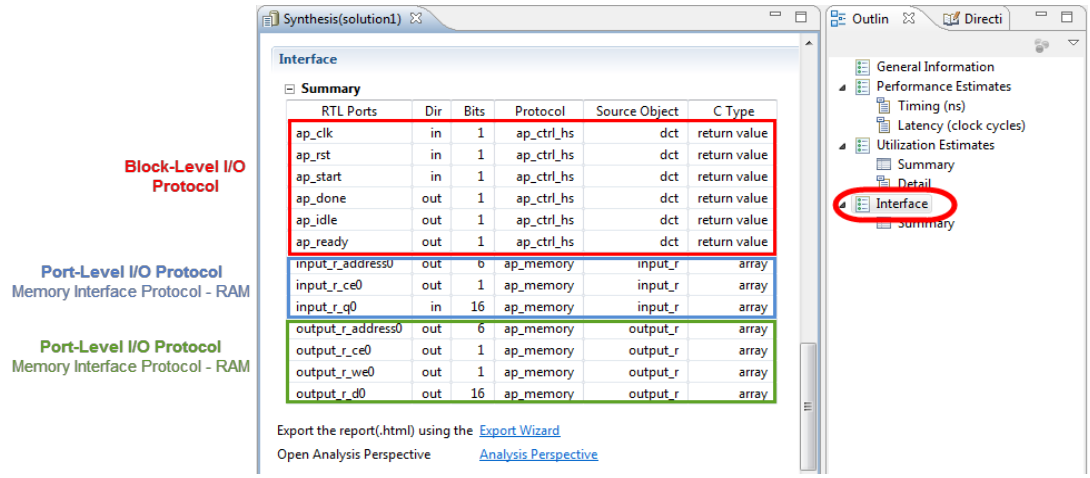


Figure 6-13: Generated Interface Signals

You can see that *ap_clk*, *ap_rst* are automatically added. The *ap_start*, *ap_done*, *ap_idle*, and *ap_ready* signals are top-level signals used as handshaking signals to indicate when the design is able to accept the next computation command (*ap_idle*), when the next computation is started (*ap_start*), and when the computation is completed (*ap_done*).

The top-level function has input and output arrays, hence an *ap_memory* interface is generated for each of them.

2-2-5. Open the **dct_dct_1d2_csynth.rpt** and **dct_dct_2d_csynth.rpt** synthesis reports under syn > report in the Explorer view.

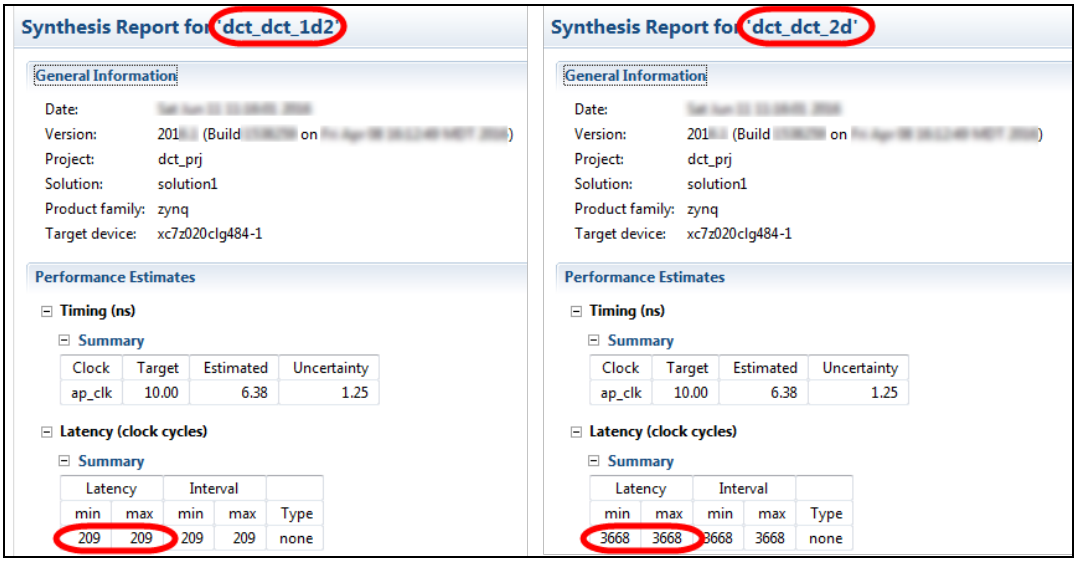


Figure 6-14: Synthesis Reports of dct1d2 and dct2d

The report for *dct_2d* clearly indicates that most of the design cycles (3668) are spent doing the row and column DCTs. Also the *dct_1d* report indicates that the latency is 209 clock cycles ((24+2)*8+1).

Performing C/RTL Co-simulation Step 3

Now that you have performed high-level synthesis on the C design, you will perform RTL co-simulation on the generated RTL using the C testbench.

Run C/RTL co-simulation, selecting Verilog and skipping VHDL. Verify that the simulation passes.

3-1. Cosimulate the Vivado HLS tool design.

3-1-1. Select **Solution** > **Run C/RTL Cosimulation** or click the **Run C/RTL Cosimulation** icon ( ).

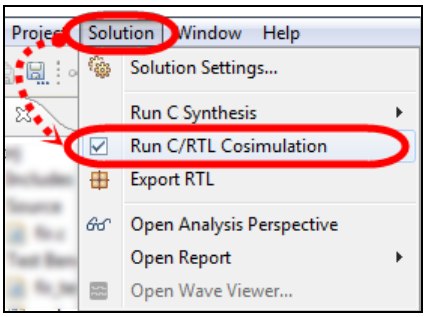


Figure 6-15: Launching from the Menu

The Run C/RTL Co-simulation dialog box opens.

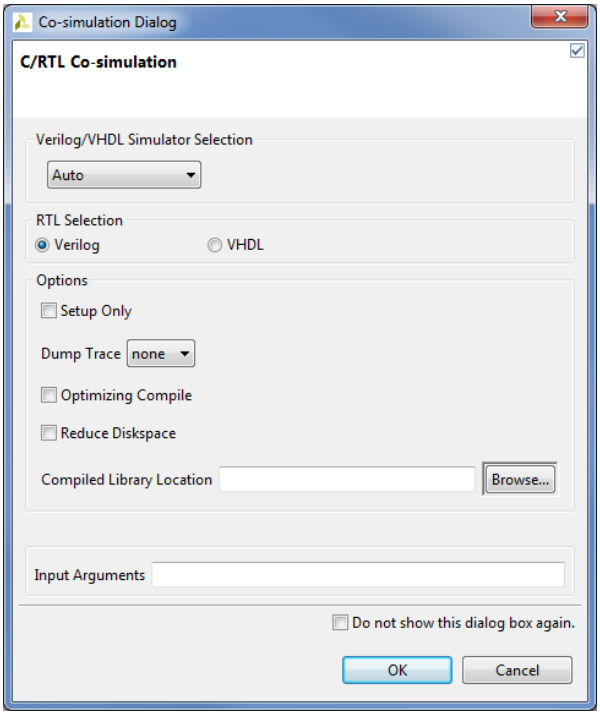


Figure 6-16: Co-simulation Dialog Box

Each of the options controls how C/RTL Cosimulation is run:

- RTL Selection: Select the RTL that is simulated (Verilog/VHDL).
- Setup Only: This creates all the files (wrappers, adapters and scripts) required to run the simulation but does not execute the simulator.
- Dump Trace: During RTL verification, the trace files can be saved and viewed using an appropriate viewer. By selecting this option, the trace file will be saved to the *<solution>/sim/<RTL>* folder.
- Optimizing Compile: This ensures a high level of optimization is used to compile the C testbench. Using this option increases the compile time but the simulation executes faster.

3-1-2. Select **Default Options (i.e. select nothing)**.

3-1-3. Click **OK**.

The simulation log will be displayed in the editor pane.

3-2. View the Cosimulation report.

The information generated by the Vivado HLS tool can be found in two places, both described here.

The first is the Console window, which reports not only the output produced by the code being simulated, but all of the simulation engine messages as well. The simulation log provides only a few simulation engine messages and the simulated code output.

3-2-1. Select the **Console** tab in the lower portion of the tool's GUI.

You may need to scroll to view all the output produced by the cosimulation.

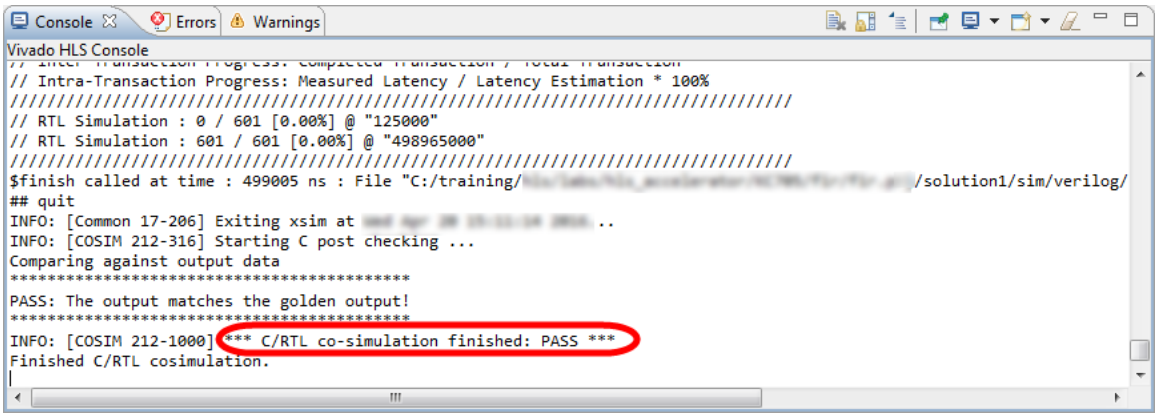


Figure 6-17: Example Output After a C/RTL Co-simulation

The other location, described below, provides only a few simulation engine messages and the simulated code output. Typically this is opened after the simulation completes; however, if you need to access it after closing the log pane, here's how to access the simulation report.

3-2-2. Expand **dct_prj** > **solution1** > **sim** > **report** in the Explorer pane.

3-2-3. Double-click the log file name to open it in the editor pane.

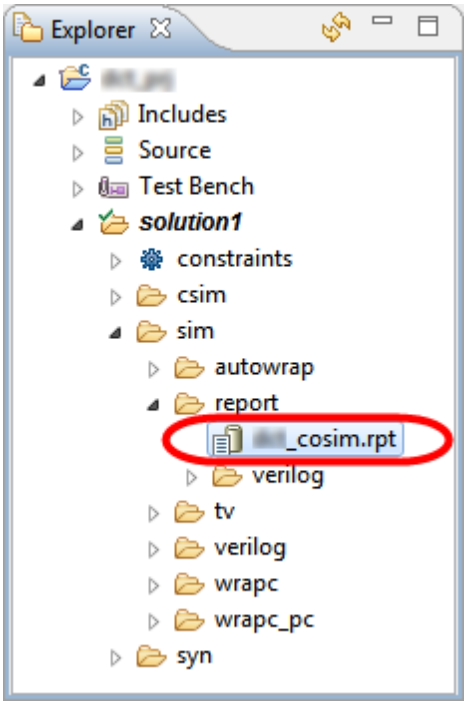


Figure 6-18: Locating the Co-Simulation Log File

The Cosimulation Report in HTML format will be displayed in the main viewing area.

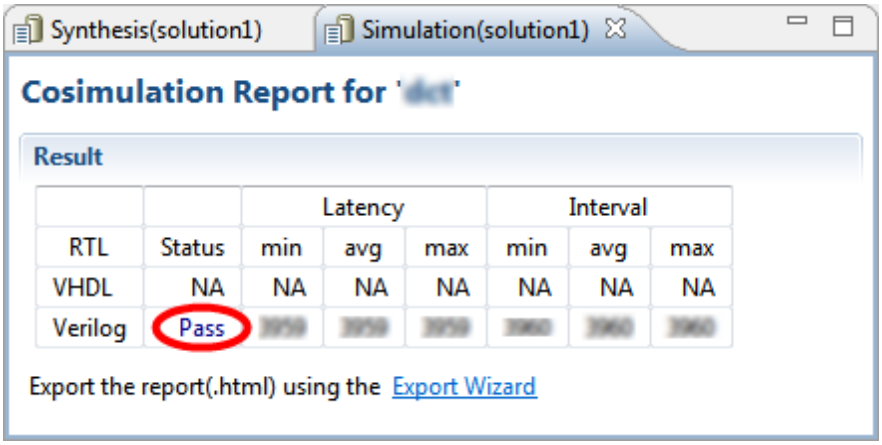


Figure 6-19: Cosimulation Report – HTML

You can quickly verify the cosimulation status here.

Applying the PIPELINE Directive Step 4

In this step, you will create a new solution by copying the previous solution settings.

You will then apply the PIPELINE directive to the following:

- *DCT_Inner_Loop*
- *Xpose_Row_Inner_Loop*
- *Xpose_Col_Inner_Loop*
- *RD_Loop_Col*
- *WR_Loop_Col*

4-1. Create a new solution by copying the previous solution (*solution1*) settings.

4-1-1. Select **Project** > **New Solution** or click the **New Solution** icon ( ) from the toolbar.

4-1-2. Leave the options at their default settings.

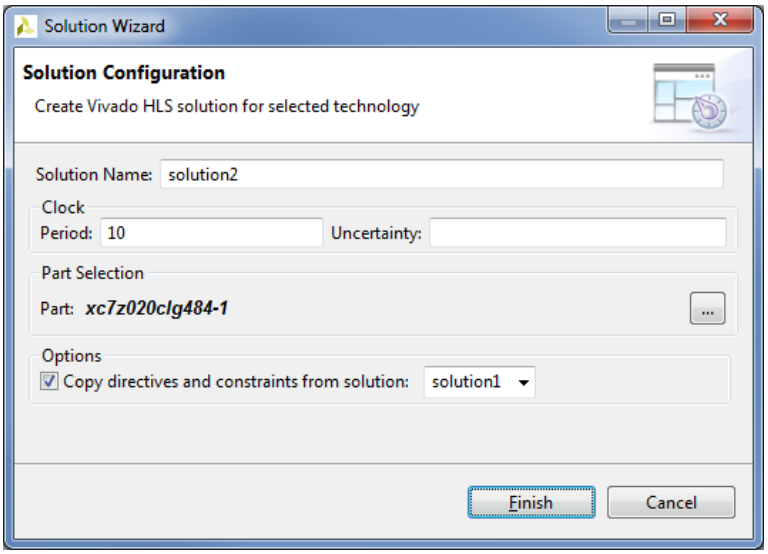


Figure 6-20: Creating a New Solution (solution2)

4-1-3. Click **Finish**.

4-2. Apply the PIPELINE directive to the *DCT_Inner_Loop* loop.

4-2-1. Select **Project** > **Close Inactive Solution Tabs** to close all inactive solution windows.

4-2-2. Ensure that the **dct.c** file is open and active in the Information pane.

4-2-3. Expand **dct_1d** in the Directive tab.

4-2-4. In the Directive tab, right-click the **DCT_Inner_Loop** loop and select **Insert Directive**.

4-2-5. Select **PIPELINE** from the Directive drop-down list.

4-2-6. Leave the **II (optional)** field blank as the Vivado HLS tool will try for an II=1, one new input every clock cycle.

4-2-7. Click **OK**.

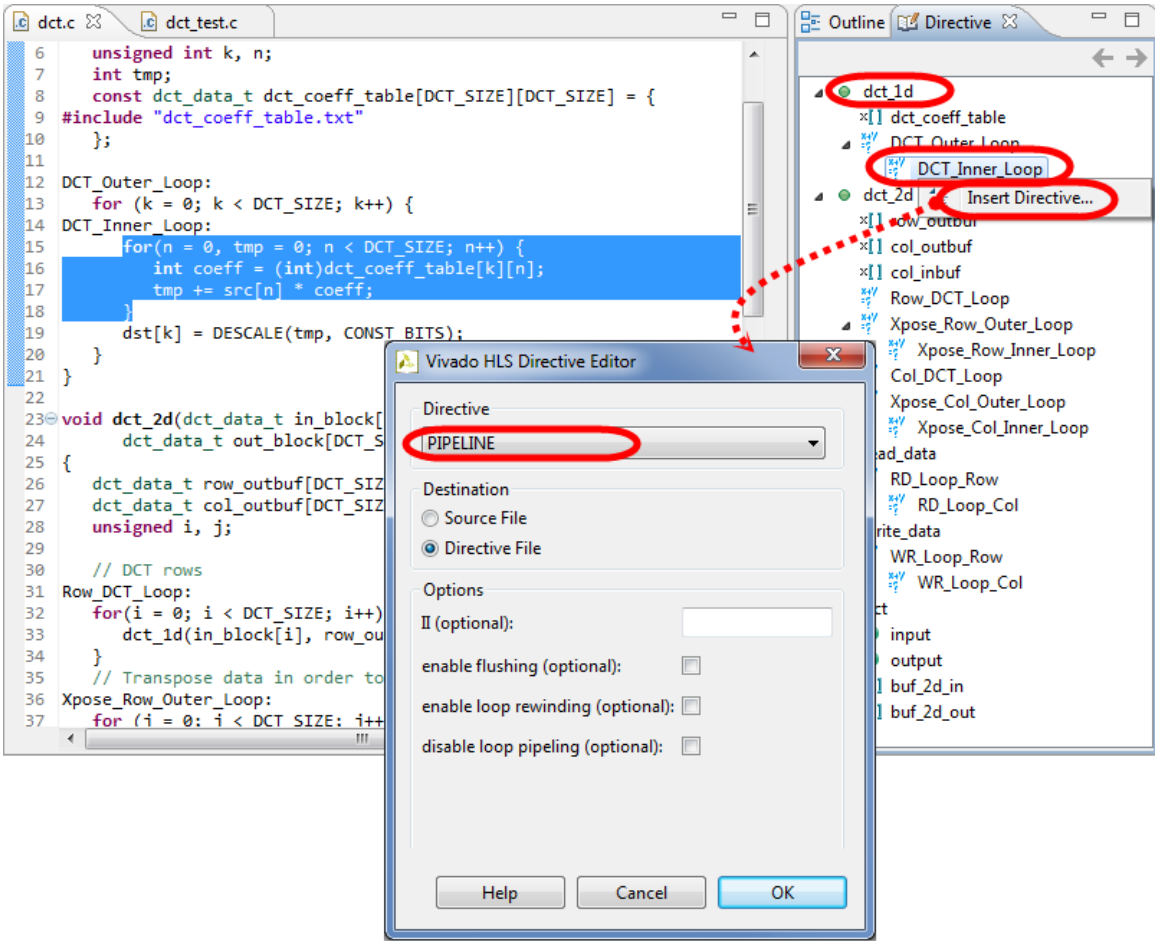


Figure 6-21: Selecting the Directive PIPELINE for DCT_Inner_Loop

4-2-8. Similarly, apply the PIPELINE directive to the **Xpose_Row_Inner_Loop** and **Xpose_Col_Inner_Loop** loops of the *dct_2d* function, **RD_Loop_Col** of the *read_data* function, and **WR_Loop_Col** of the *write_data* function.

The final view of the PIPELINE directives in the Directive tab should look similar to the figure below.

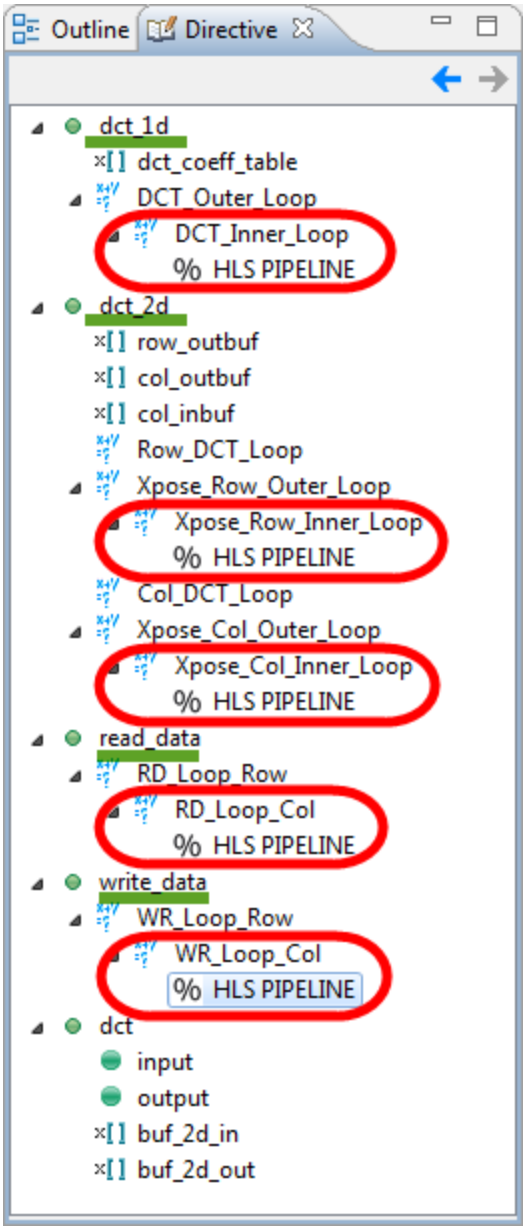


Figure 6-22: Final View of PIPELINE Directives

4-3. Synthesize the Vivado HLS design.

If you do not recall how to perform this task, refer to the "Synthesizing the Vivado HLS Design" section under Vivado HLS Operations in the *Lab Reference Guide*.

4-4. View the Synthesis report.

You should see that the Synthesis report in the Information window opens automatically when synthesis is completed.

4-4-1. Select **Project** > **Compare Reports** or click the **Compare Reports** icon ( ) to compare the results of the two solutions (*solution1* and *solution2*).

4-4-2. Select **solution1** and **solution2** from the Available solutions section.

4-4-3. Click **Add**.

4-4-4. Click **OK**.

You should see the compared results of *solution1* and *solution2* as shown in the figure below.

Performance Estimates

Timing (ns)

Clock		solution2	solution1
ap_clk	Target	10.00	10.00
	Estimated	7.68	6.38

Latency (clock cycles)

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

Figure 6-23: Performance Comparison After Pipelining

Observe that the latency reduced from 3959 to 1850 clock cycles.

4-4-5. Scroll down in the comparison report to view the resource utilization.

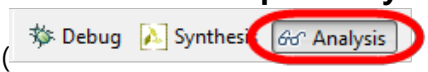
Observe that the LUT utilization increased and flip-flop decreased, whereas block RAM and DSP48E remained the same.

Utilization Estimates

	solution2	solution1
BRAM_18K	5	5
DSP48E	1	1
FF	255	278
LUT	457	353

Figure 6-24: Comparison of Resource Estimates After Pipelining

4-5. Switch to the Analysis perspective and determine where most of the clock cycles are spent, i.e. where the large latencies are.

4-5-1. Select **Solution** > **Open Analysis Perspective** or click the **Analysis** perspective icon (

) to open the analysis viewer.

4-5-2. In the Module Hierarchy tab, select **dct**.

4-5-3. View the entries under the Performance Profile tab.

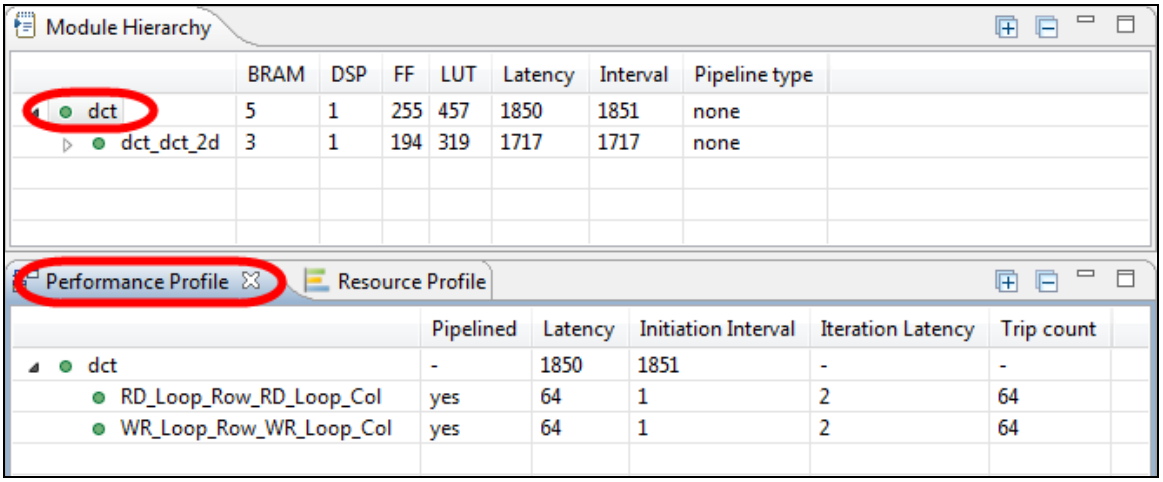


Figure 6-25: Performance Profile at the dct Function Level

Observe the **RD_Loop_Row_RD_Loop_Col** and **WR_Loop_Row_WR_Loop_Col** entries.

These are two nested loops that have been flattened and given new names formed by appending the inner loop name to the out loop name. You can also verify this via the messages in the Console view.

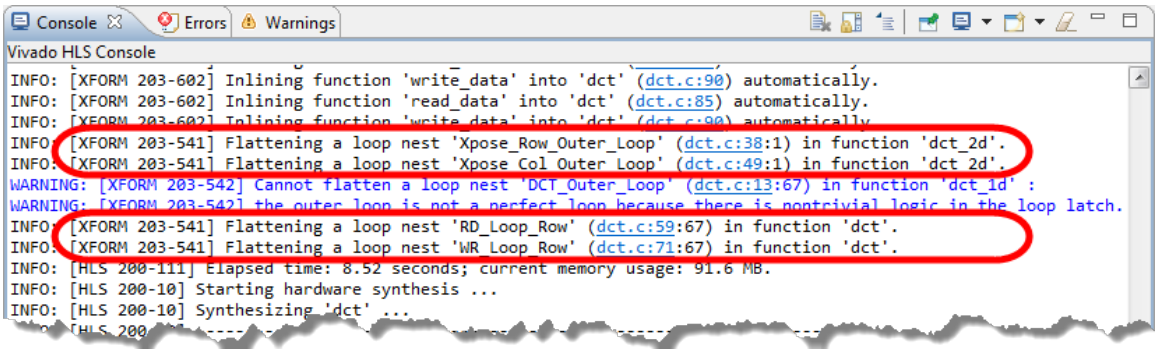


Figure 6-26: Viewing the Console Tab Indicating Loops Flattening

4-5-4. In the Module Hierarchy tab, expand **dct** > **dct_dct_2d** > **dct_dct_1d2**.

Notice that the most of the latency occurs in the *dct_2d* function.

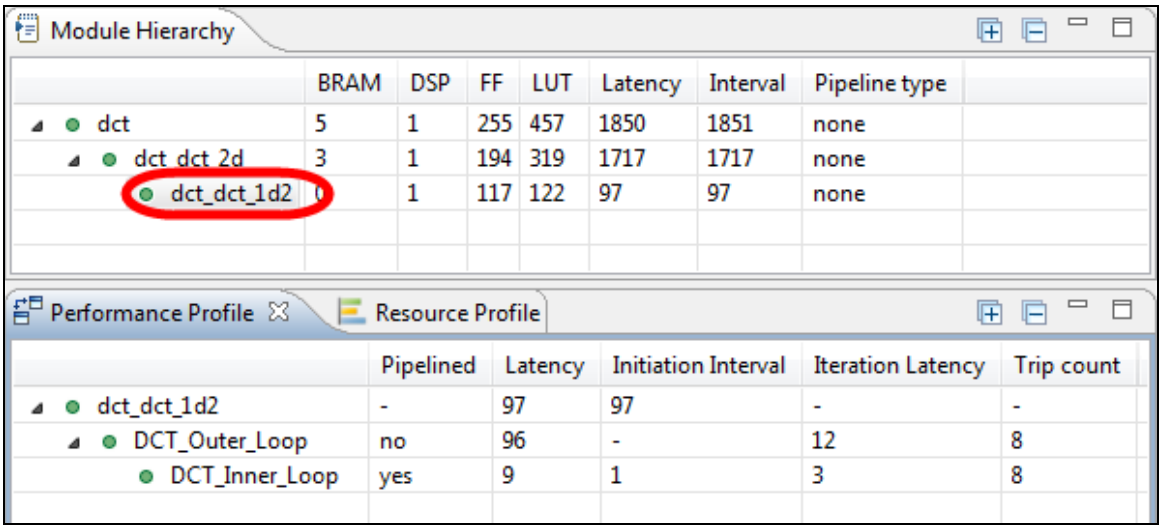


Figure 6-27: dct_1d Function Performance Profile

4-5-5. In the Performance Profile tab, select **DCT_Inner_Loop** (1).

4-5-6. Right-click the **node_60 (write)** block in the C3 state in the Performance view (2).

4-5-7. Select **Goto Source** (3).

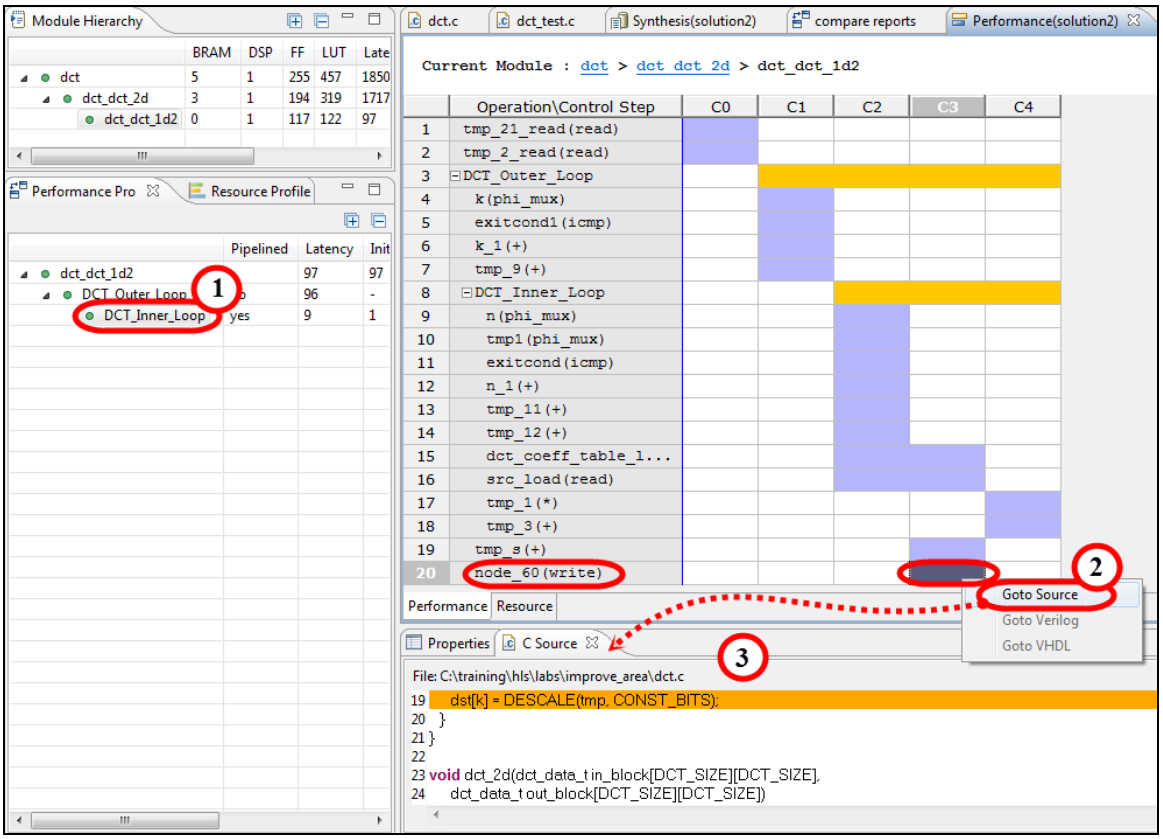


Figure 6-28: Understanding What is Preventing DCT_Outer_Loop Flattening

Notice that line no. 19 is highlighted, which is preventing the flattening of **DCT_Outer_Loop**.

4-5-8. Switch to the **Synthesis** perspective.

Moving the PIPELINE directive from the inner loop to the outer loop of *dct_1d* will lead to more parallelism of the multiply and add operations. That is, eight (8) multiply-and-add operations are done concurrently, thus minimizing the number of cycles required to compute each value in the output array.

4-6. Create a new solution by copying the previous solution (*solution2*) settings.

4-6-1. Select **Project** > **New Solution** or click the **New Solution** icon ( ) from the toolbar.

4-6-2. Leave the options at their default settings (with *solution2* selected).

4-6-3. Click **Finish**.

4-7. Apply the PIPELINE directive to the *DCT_Outer_Loop* loop of *dct_1d* and remove the PIPELINE directive of the *DCT_Inner_Loop* loop of the *dct_1d* function.

4-7-1. Select **Project** > **Close Inactive Solution Tabs** to close all inactive solution windows.

4-7-2. Ensure that the **dct.c** file is open and active in the Information pane.

4-7-3. Expand **dct_1d** in the Directive tab.

4-7-4. In the Directive tab, select the **PIPELINE** directive of the *DCT_Inner_Loop* loop of the *dct_1d* function.

4-7-5. Right-click and select **Remove Directive**.

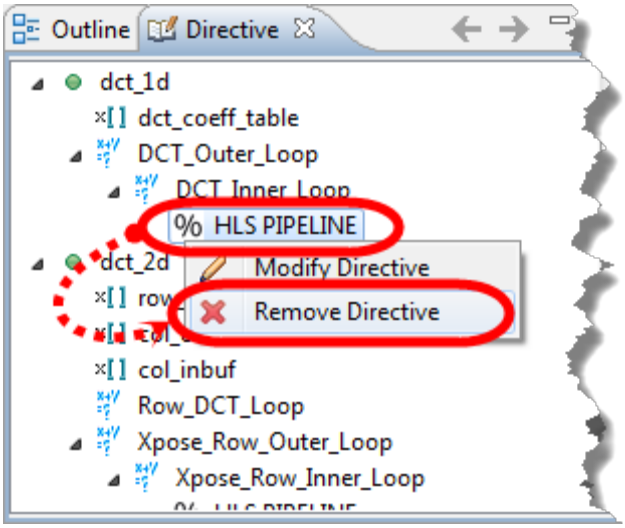


Figure 6-29: Removing the PIPELINE Directive of DCT_Inner_Loop

4-7-6. In the Directive tab, right-click the **DCT_Outer_Loop** loop and select **Insert Directive**.

4-7-7. Select **PIPELINE** from the Directive drop-down list.

4-7-8. Click **OK**.

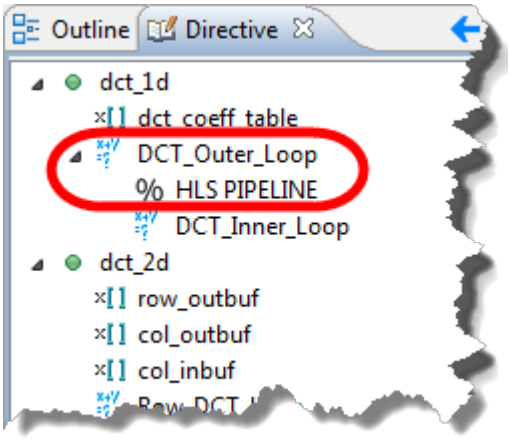


Figure 6-30: PIPELINE Directive for the DCT_Outer_Loop of the dct_1d Function

By pipelining an outer loop, all inner loops will be unrolled automatically (if legal), so there is no need to explicitly apply an UNROLL directive to **DCT_Inner_Loop**. Simply move the pipeline to the outer loop: the nested loop will still be pipelined but the operations in the inner-loop body will operate concurrently.

4-8. Synthesize the Vivado HLS design.

If you do not recall how to perform this task, refer to the "Synthesizing the Vivado HLS Design" section under Vivado HLS Operations in the *Lab Reference Guide*.

4-9. View the Synthesis report.

You should see that the Synthesis report in the Information window opens automatically when synthesis is completed.

4-9-1. Select **Project** > **Compare Reports** or click the **Compare Reports** icon ( ) to compare the results of the two solutions.

4-9-2. Select **solution2** and **solution3** from the Available solutions section.

4-9-3. Click **Add**.

4-9-4. Click **OK**.

You should see the compared results of *solution2* and *solution3* as shown in the figure below.

Performance Estimates

Timing (ns)

Clock		solution3	solution2
ap_clk	Target	10.00	10.00
	Estimated	9.40	7.68

Latency (clock cycles)

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

Figure 6-31: Performance Comparison After Pipelining DCT_Outer_Loop of dct_1d

Observe that the latency reduced from 1850 to 874 clock cycles.

4-9-5. Scroll down in the comparison report to view the resource utilization.

Observe that the utilization of all resources (except block RAM) increased.

Since *DCT_Inner_Loop* was unrolled, the parallel computation requires eight DSP48E resources.

Utilization Estimates

	solution3	solution2
BRAM_18K	5	5
DSP48E	8	1
FF	677	255
LUT	518	457

Figure 6-32: Comparison of Resource Estimates After Pipelining DCT_Outer_Loop of dct_1d

4-9-6. Double-click **dct_dct_1d2_csynth.rpt** under dct_prj > solution3 > syn > report in the Explorer pane.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	9.40	1.25

Latency (clock cycles)

Summary

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

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- DCT_Outer_Loop	34	34	7	4	1	8	yes

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	8	-	-
Expression	-	-	0	128
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	0	-	119	16
Multiplexer	-	-	-	21
Register	-	-	420	-
Total	0	8	539	165
Available	280	220	106400	53200
Utilization (%)	0	3	~0	~0

Figure 6-33: Synthesis Report of 1d2 Function - Increased Resource Utilization

Observe that the pipeline **Initiation Interval (II)** is four cycles, not one as might be hoped and there are now eight block RAMs being used for the coefficient table.

Looking closely at the synthesis log, notice that the coefficient table was automatically partitioned, resulting in eight separate ROMs. This helped reduce the latency by keeping the unrolled computation loop fed. However the input arrays to the *dct_1d* function were not automatically partitioned.

The reason that the II is four rather than the eight one might expect is because the Vivado HLS tool automatically uses dual-port RAMs when beneficial to scheduling operations.

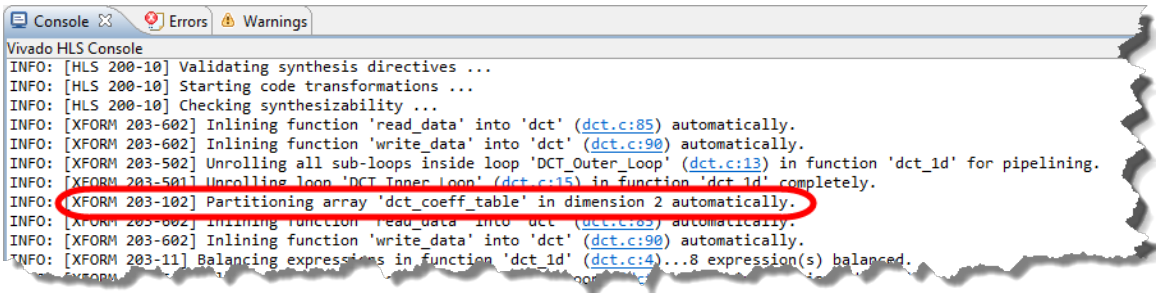


Figure 6-34: Automatic Partitioning of dct_coeff_table

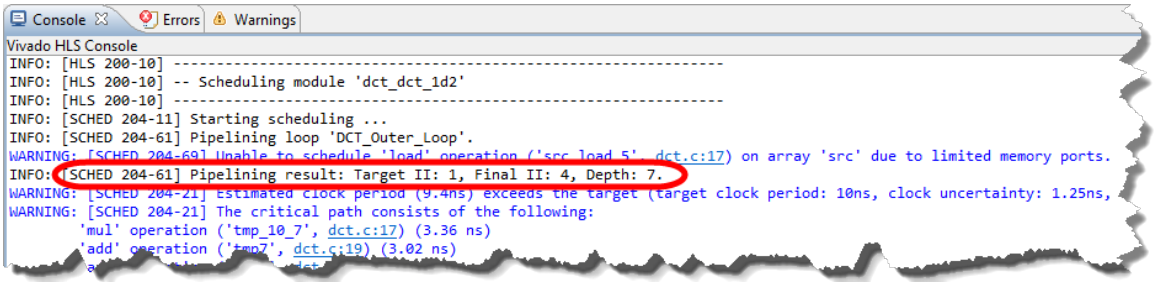
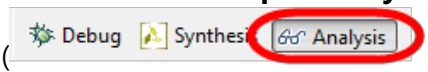


Figure 6-35: Initiation Interval of 4

4-10. Switch to the Analysis perspective and look at the *dct_1d* Performance view.

4-10-1. Select **Solution** > **Open Analysis Perspective** or click the **Analysis** perspective icon (

) to open the analysis viewer.

4-10-2. In the Module Hierarchy tab, expand **dct** > **dct_dct_2d** > **dct_dct_1d2**.

4-10-3. Select **dct_dct_1d2**.

Observe that the **DCT_Outer_Loop** is pipelined.

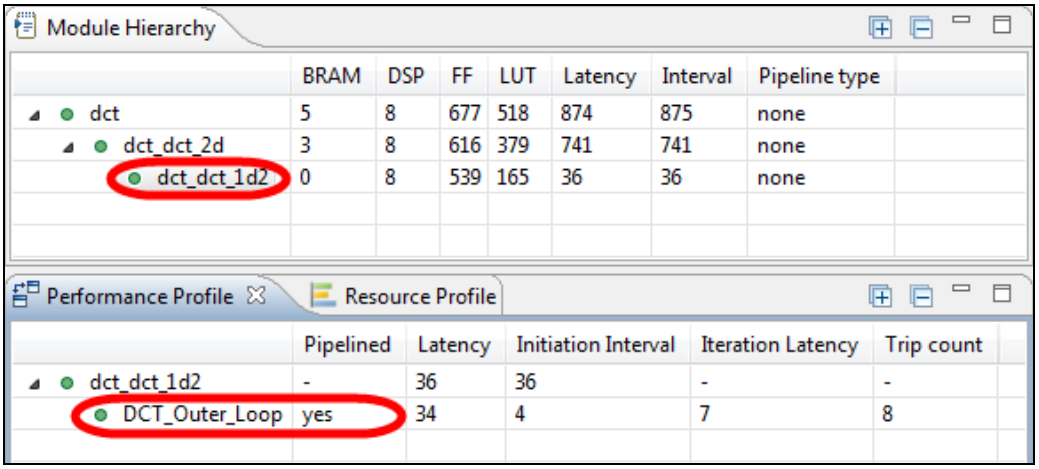


Figure 6-36: DCT_Outer_Loop Pipelined

Also observe that **DCT_Outer_Loop** spans over eight states in the Performance View.

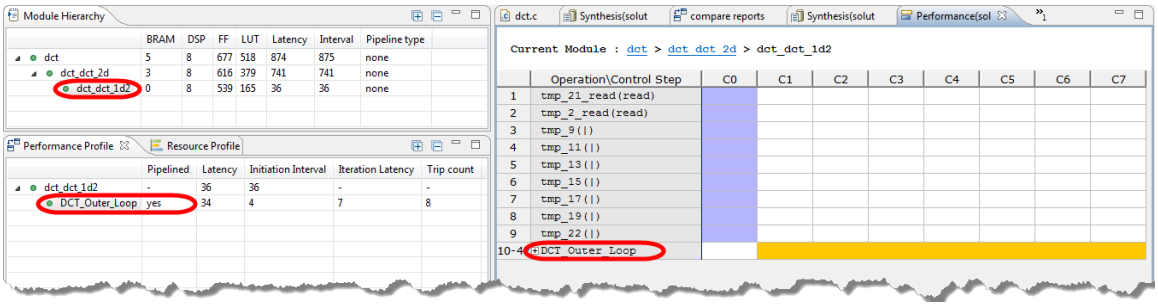


Figure 6-37: DCT_Outer_Loop in Performance View

4-10-4. Select the **Resource** tab and expand **Memory Ports**.

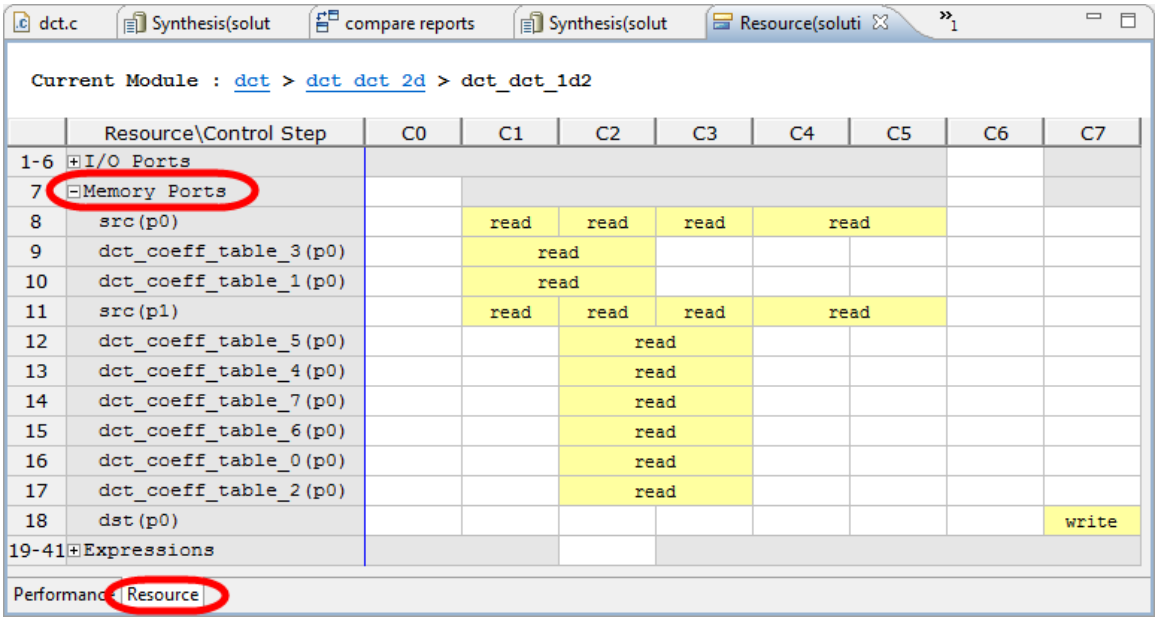


Figure 6-38: Resource Tab – Memory Ports View

Observe that the memory accesses on the block RAM src are being used to the maximum in every clock cycle.

A block RAM at most can be dual port and both ports are being used. This is a good indication that the design may be bandwidth limited by the memory resources.

Improving the Memory Bandwidth Step 5

In this step, you will apply the ARRAY_PARTITION directive to *buf_2d_in* of *dct* (since the bottleneck was on the *src* port of the *dct_1d* function, which was passed via *in_block* of the *dct_2d* function, which in turn was passed via *buf_2d_in* of the *dct* function) and *col_inbuf* of *dct_2d*.

5-1. Create a new solution by copying the previous solution (*solution3*) settings.

5-1-1. Select **Project** > **New Solution** or click the **New Solution** icon ( ) from the toolbar.

5-1-2. Leave the options at their default settings (with *solution3* selected).

5-1-3. Click **Finish**.

5-2. Apply the ARRAY_PARTITION directive to *buf_2d_in* of the *dct* function and *col_inbuf* of the *dct_2d* function.

5-2-1. Select **Project** > **Close Inactive Solution Tabs** to close all inactive solution windows.

5-2-2. Ensure that the **dct.c** file is open and active in the Information pane.

5-2-3. Expand **dct** in the Directive tab.

5-2-4. In the Directive tab, right-click the **buf_2d_in** array of the *dct* function and select **Insert Directive**.

The buf_2d_in array is selected since the bottleneck was on the *src* port of the *dct_1d* function, which was passed via *in_block* of the *dct_2d* function, which in turn was passed via *buf_2d_in* of the *dct* function).

5-2-5. Select **ARRAY_PARTITION** from the Directive drop-down list.

5-2-6. Select the *type (optional)* as **complete**.

5-2-7. Enter **2** in the *dimension (optional)* field.

5-2-8. Click **OK**.

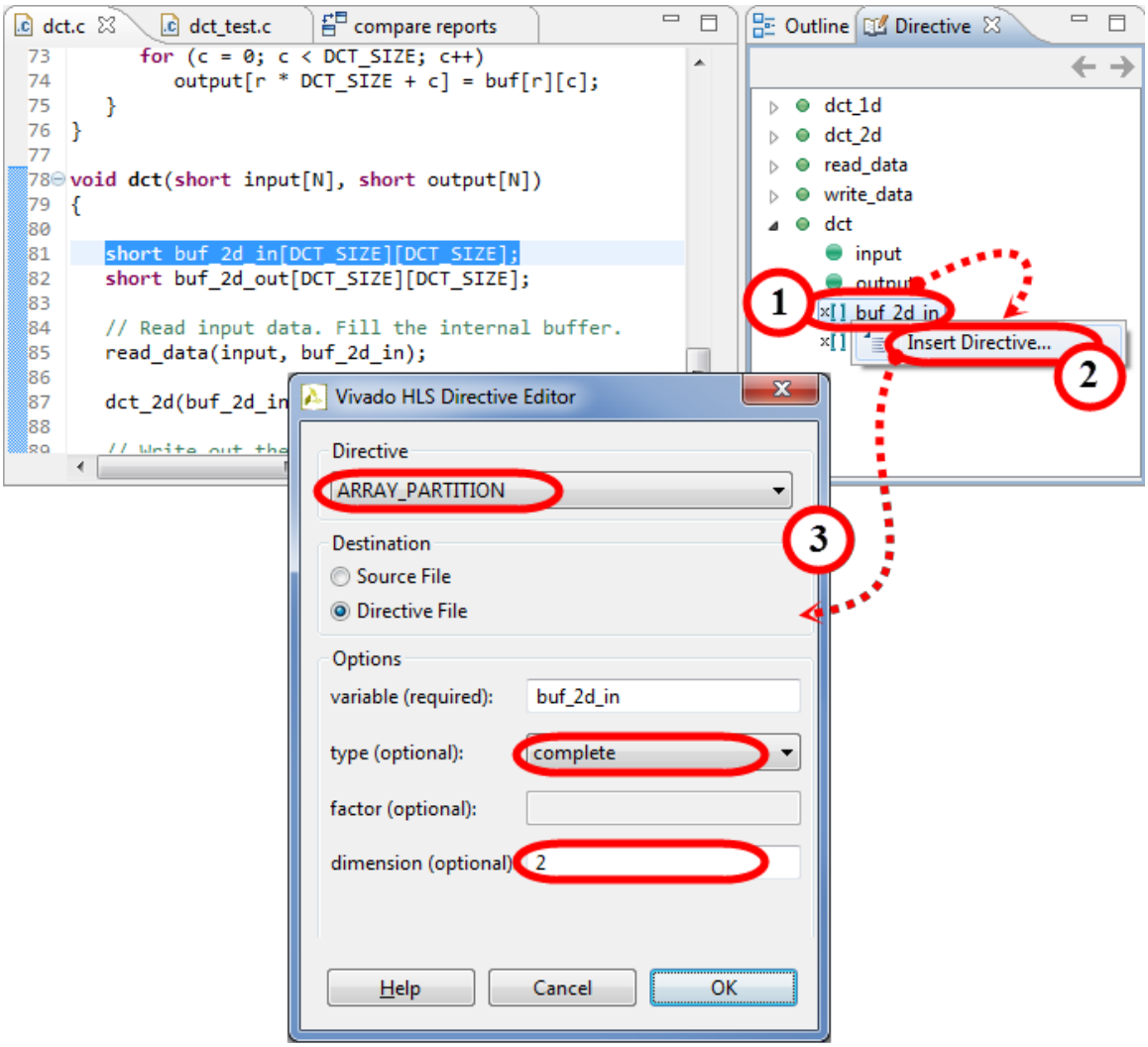


Figure 6-39: Applying the ARRAY_PARTITION Directive to the Memory Buffer

5-2-9. Similarly, apply the **ARRAY_PARTITION** directive with *dimension* of **2** and *type* as **complete** to the *col_inbuf* array of the *dct2d* function.

The final view of the ARRAY_PARTITION directives in the Directive tab should look similar to the figure below.

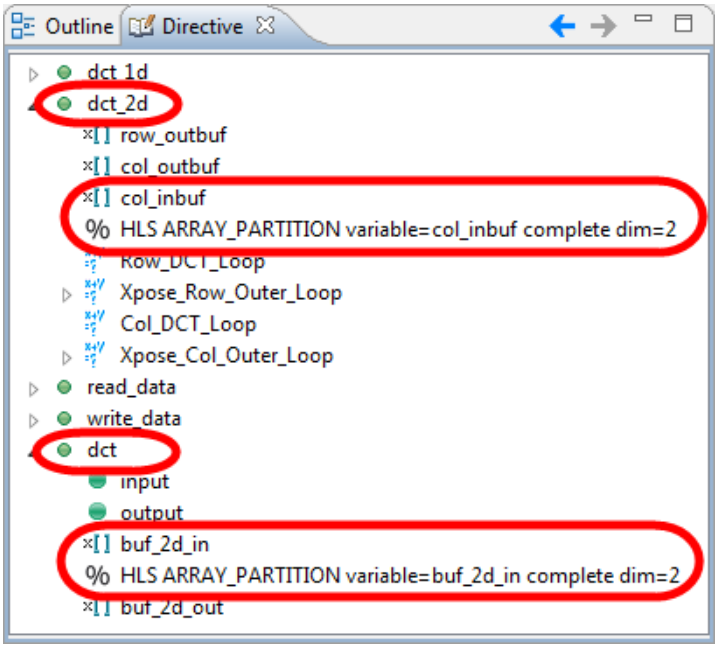


Figure 6-40: Final View of ARRAY_PARTITION Directives

5-3. Synthesize the Vivado HLS design.

If you do not recall how to perform this task, refer to the "Synthesizing the Vivado HLS Design" section under Vivado HLS Operations in the *Lab Reference Guide*.

5-4. View the Synthesis report.

You should see that the Synthesis report in the Information window opens automatically when synthesis is completed.

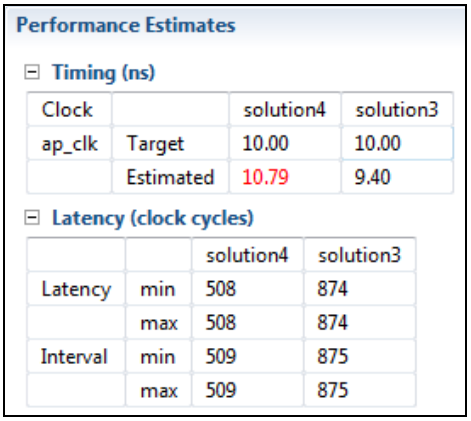
5-4-1. Select **Project** > **Compare Reports** or click the **Compare Reports** icon ( ) to compare the results of the two solutions (*solution3* and *solution4*).

5-4-2. Select **solution3** and **solution4** from the Available solutions section.

5-4-3. Click **Add**.

5-4-4. Click **OK**.

You should see the compared results of *solution3* and *solution4* as shown in the figure below.



Performance Estimates

Timing (ns)

Clock		solution4	solution3
ap_clk	Target	10.00	10.00
	Estimated	10.79	9.40

Latency (clock cycles)

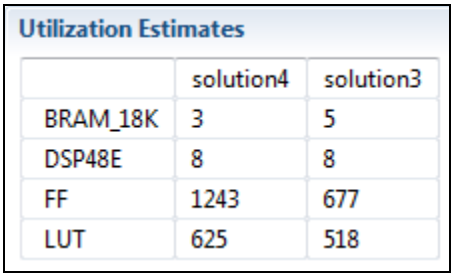
		solution4	solution3
Latency	min	508	874
	max	508	874
Interval	min	509	875
	max	509	875

Figure 6-41: Performance Comparison After Array Partitioning

Observe that the latency reduced from 874 to 508 clock cycles.

5-4-5. Scroll down in the comparison report to view the resource utilization.

Observe that flip-flop usage has increased by almost double.



Utilization Estimates

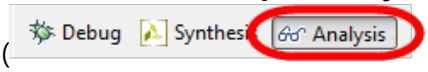
	solution4	solution3
BRAM_18K	3	5
DSP48E	8	8
FF	1243	677
LUT	625	518

Figure 6-42: Comparison of Resource Estimates After Array Partitioning

5-4-6. Double-click **dct_csynth.rpt** under dct_prj > solution4 > syn > report in the Explorer pane.

5-4-7. Expand the **Loop** entry in the *dct* report and observe that the **Pipeline II** is now 1.

5-5. Switch to the Analysis perspective and look at the *dct* resources profile view.

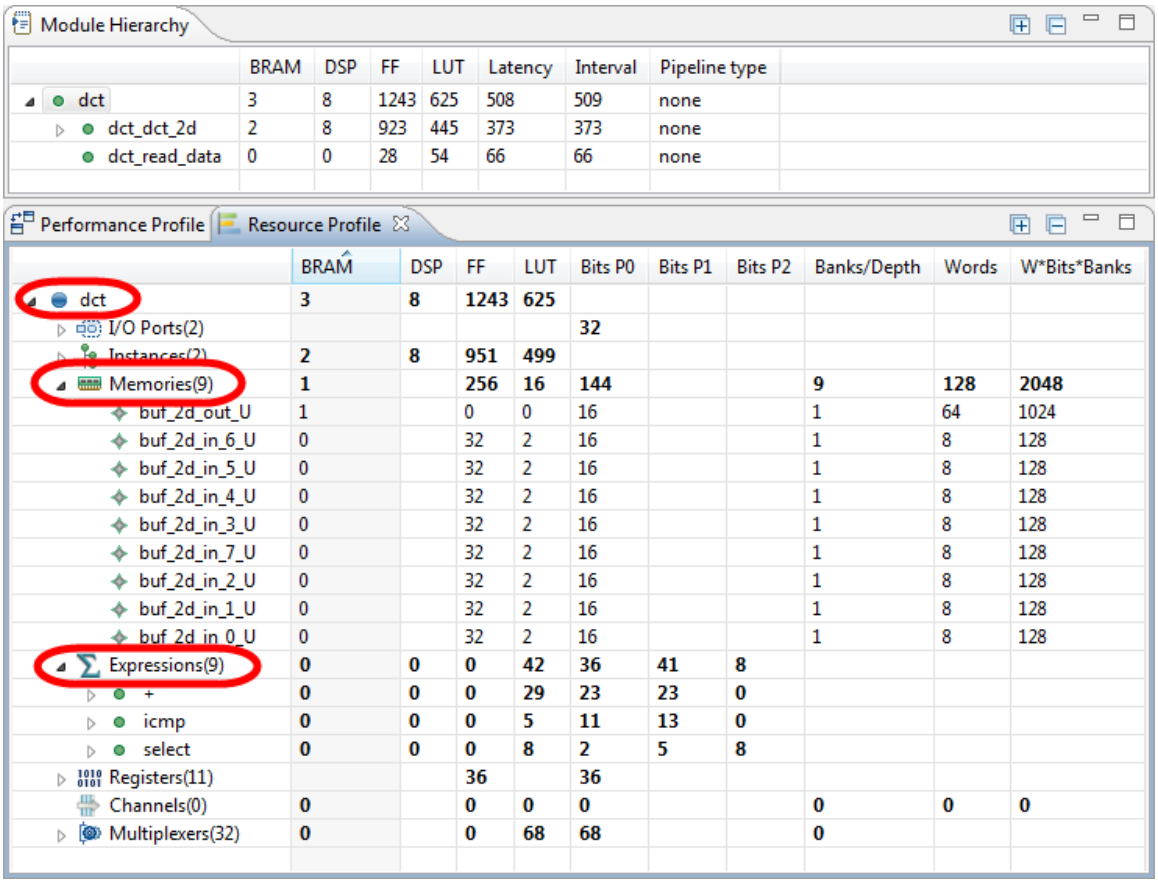
5-5-1. Select **Solution** > **Open Analysis Perspective** or click the **Analysis** perspective icon (

) to open the analysis viewer.

5-5-2. In the Module Hierarchy tab, select **dct**.

5-5-3. Select the **Resource Profile** tab.

5-5-4. Expand the **Memories** and **Expressions** entries.



	BRAM	DSP	FF	LUT	Latency	Interval	Pipeline type
dct	3	8	1243	625	508	509	none
dct_dct_2d	2	8	923	445	373	373	none
dct_read_data	0	0	28	54	66	66	none

	BRAM	DSP	FF	LUT	Bits P0	Bits P1	Bits P2	Banks/Depth	Words	W*Bits*Banks
dct	**3**	**8**	**1243**	**625**						
I/O Ports(2)					**32**					
Instances(2)	**2**	**8**	**951**	**499**						
Memories(9)	**1**		**256**	**16**	**144**			**9**	**128**	**2048**
buf_2d_out_U	1		0	0	16			1	64	1024
buf_2d_in_6_U	0		32	2	16			1	8	128
buf_2d_in_5_U	0		32	2	16			1	8	128
buf_2d_in_4_U	0		32	2	16			1	8	128
buf_2d_in_3_U	0		32	2	16			1	8	128
buf_2d_in_7_U	0		32	2	16			1	8	128
buf_2d_in_2_U	0		32	2	16			1	8	128
buf_2d_in_1_U	0		32	2	16			1	8	128
buf_2d_in_0_U	0		32	2	16			1	8	128
Expressions(9)	**0**	**0**	**0**	**42**	**36**	**41**	**8**			
+	**0**	**0**	**0**	**29**	**23**	**23**	**0**			
icmp	**0**	**0**	**0**	**5**	**11**	**13**	**0**			
select	**0**	**0**	**0**	**8**	**2**	**5**	**8**			
Registers(11)			**36**		**36**					
Channels(0)	**0**		**0**	**0**	**0**			**0**	**0**	**0**
Multiplexers(32)	**0**		**0**	**68**	**68**			**0**		

Figure 6-43: Resource Profile After Partitioning Buffers

Observe that the most of the resources are consumed by instances. The *buf_2d_in* array is partitioned into multiple memories and most of the operations are done in addition and comparison.

5-5-5. Switch to Synthesis perspective.

Applying the DATAFLOW Directive Step 6

In this step, you will apply the DATAFLOW directive to improve the throughput.

6-1. Create a new solution by copying the previous solution (*solution4*) settings.

6-1-1. Select **Project** > **New Solution** or click the **New Solution** icon ( ) from the toolbar.

6-1-2. Leave the options at their default settings (with *solution4* selected).

6-1-3. Click **Finish**.

6-2. Apply the DATAFLOW directive to the *dct* function.

6-2-1. Select **Project** > **Close Inactive Solution Tabs** to close all inactive solution windows.

6-2-2. Ensure that the **dct.c** file is open and active in the Information pane.

6-2-3. In the Directive tab, right-click the **dct** function and select **Insert Directive**.

6-2-4. Select **DATAFLOW** from the Directive drop-down list to improve the throughput.

6-2-5. Click **OK**.

6-3. Synthesize the Vivado HLS design.

If you do not recall how to perform this task, refer to the "Synthesizing the Vivado HLS Design" section under Vivado HLS Operations in the *Lab Reference Guide*.

6-4. View the Synthesis report.

You should see that the Synthesis report in the Information window opens automatically when synthesis is completed.

6-4-1. Observe that the **dataflow** type pipeline throughput is listed in the Performance Estimates.

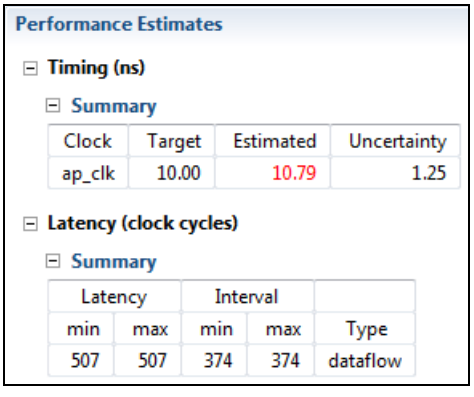


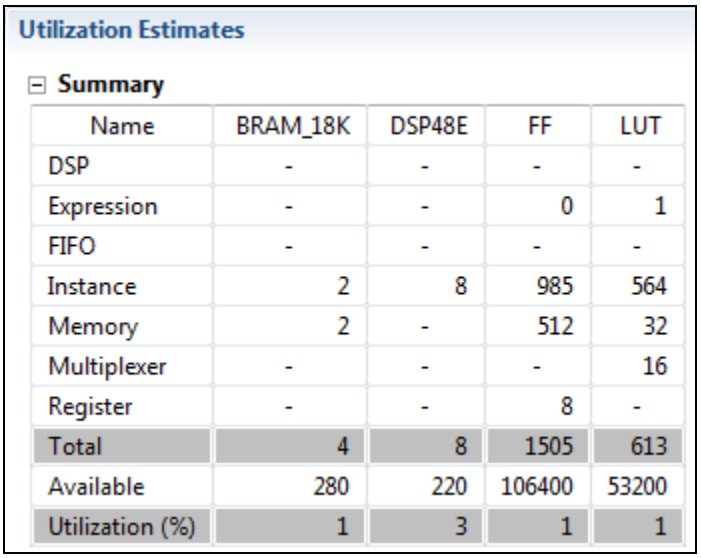
Figure 6-44: Performance Estimates After Applying the DATAFLOW Directive

The Dataflow pipeline throughput indicates the number of clock cycles between each set of inputs reads (interval parameter). If this value is less than the design latency, it indicates that the design can start processing new inputs before the currents input data are output.

Note that the dataflow is only supported for the functions and loops at the top level, not those that are down through the design hierarchy. Only loops and functions exposed at the top level of the design will get benefit from dataflow optimization.

6-4-2. Scroll down in the comparison report to view the resource utilization.

Observe that the number of BRAM_18K resources required at the top level has increased from three to four.



Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	1
FIFO	-	-	-	-
Instance	2	8	985	564
Memory	2	-	512	32
Multiplexer	-	-	-	16
Register	-	-	8	-
Total	4	8	1505	613
Available	280	220	106400	53200
Utilization (%)	1	3	1	1

Figure 6-45: Resource Utilization After DATAFLOW Directive

6-4-3. In the Console view, notice that **dct_coeff_table** is automatically partitioned in dimension 2.

The *buf_2d_in* and *col_inbuf* arrays are partitioned as you had applied the directive in the previous run. The dataflow is applied at the top level, which created channels between the top-level functions *read_data*, *dct_2d*, and *write_data*.

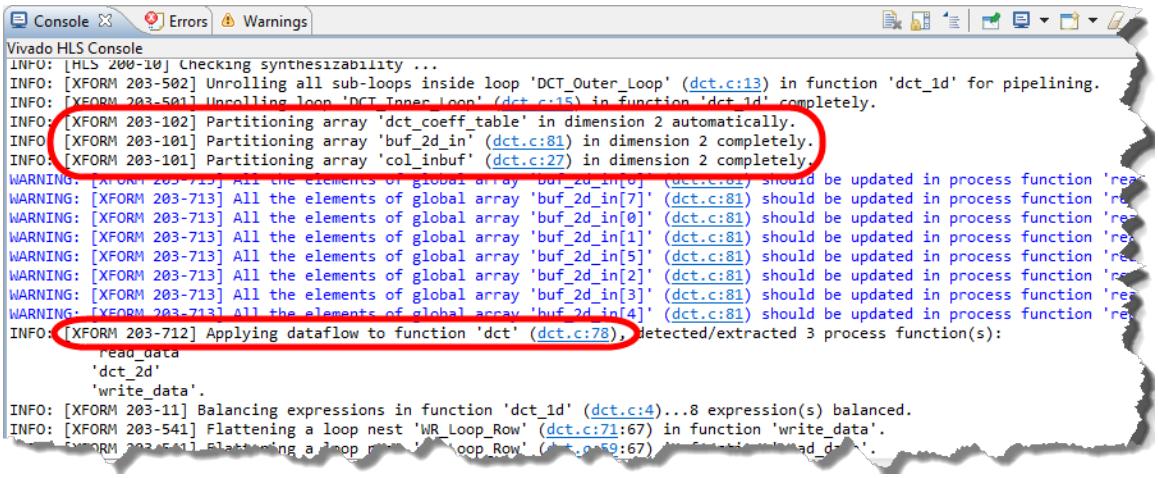
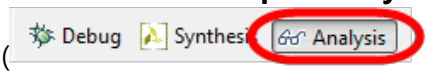


Figure 6-46: Synthesis Process After the DATAFLOW Directive

6-5. Switch to the Analysis perspective and look at the *dct* performance profile view.

6-5-1. Select **Solution** > **Open Analysis Perspective** or click the **Analysis** perspective icon (

) to open the analysis viewer.

6-5-2. In the Module Hierarchy tab, select **dct_2d**.

6-5-3. Select the **Performance Profile** tab.

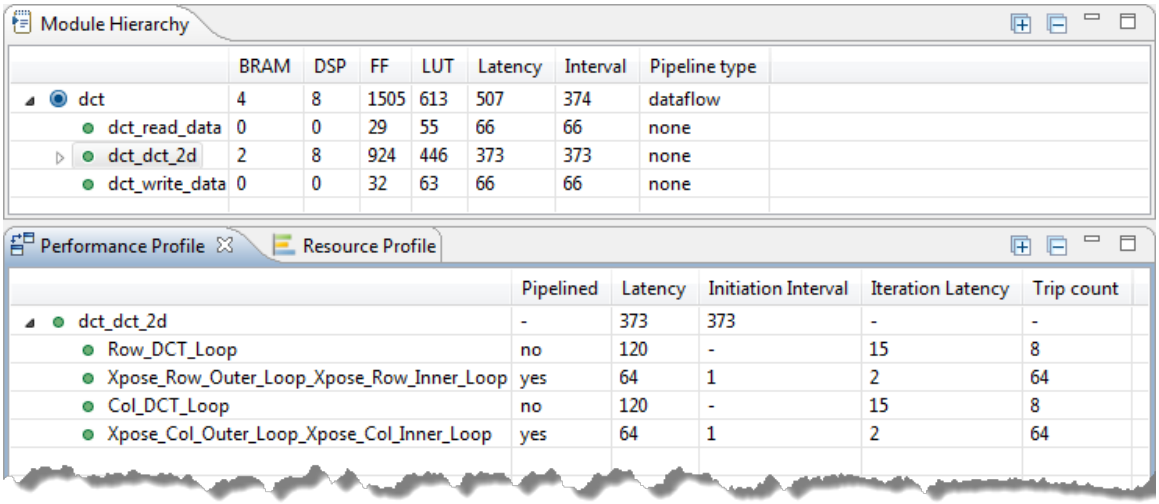


Figure 6-47: Performance Analysis After the DATAFLOW Directive

One of the limitations of the dataflow optimization is that it only works on top-level loops and functions.

One way to have the blocks in *dct_2d* operate in parallel would be to pipeline the entire function. This however would unroll all the loops 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, i.e. inline the *dct_2d* function.

6-5-4. Switch to the Synthesize perspective.

Applying the INLINE Directive Step 7

7-1. Create a new solution by copying the previous solution (*solution5*) settings.

7-1-1. Select **Project** > **New Solution** or click the **New Solution** icon ( ) from the toolbar.

7-1-2. Leave the options at their default settings (with *solution5* selected).

7-1-3. Click **Finish**.

7-2. Apply the INLINE directive to the *dct_2d* function.

7-2-1. Select **Project** > **Close Inactive Solution Tabs** to close all inactive solution windows.

7-2-2. Ensure that the **dct.c** file is open and active in the Information pane.

7-2-3. In the Directive tab, right-click the **dct_2d** function and select **Insert Directive**.

7-2-4. Select **INLINE** from the Directive drop-down list.

The INLINE directive causes the function to which it is applied to be inlined—its hierarchy is dissolved.

7-2-5. Click **OK**.

7-3. Synthesize the Vivado HLS design.

If you do not recall how to perform this task, refer to the "Synthesizing the Vivado HLS Design" section under Vivado HLS Operations in the *Lab Reference Guide*.

7-4. View the Synthesis report.

You should see that the Synthesis report in the Information window opens automatically when synthesis is completed.

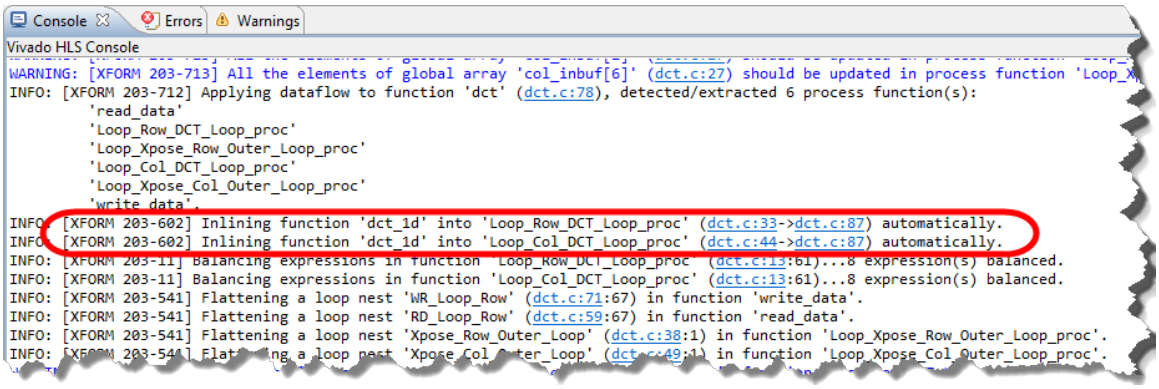
7-4-1. Observe that the latency reduced from 507 to 479 clock cycles and the DATAFLOW pipeline throughput drastically reduced from 374 to 106 clock cycles in the Performance Estimates.

7-4-2. Examine the synthesis log to see what transformations were applied automatically.

The *dct_1d* function calls are now automatically inlined into the loops from which they are called, which allows the loop nesting to be flattened automatically.

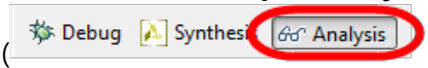
Note also that the DSP48E usage has doubled (from 8 to 16). This is because previously a single instance of *dct_1d* was used to do both row and column processing. Now that the row and column loops are executing concurrently, this can no longer be the case and two copies of *dct_1d* are required. The Vivado HLS tool will seek to minimize the number of clocks, even if it means increasing the area.

Block RAM usage has increased once again (from 4 to 6), due to ping-pong buffering between more dataflow processes.



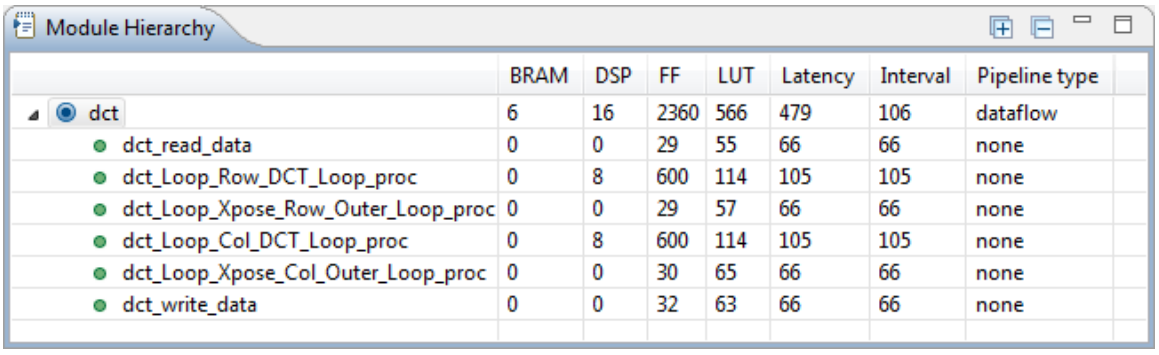
Figureb 6-48: Syntheis Process in the Console Tab After INLINE Directive Applied to the dct_2d Function

7-5. Switch to the Analysis perspective and review the module hierarchy.

7-5-1. Select **Solution** > **Open Analysis Perspective** or click the **Analysis** perspective icon (

) to open the analysis viewer.

7-5-2. In the Module Hierarchy tab, select **dct**.



	BRAM	DSP	FF	LUT	Latency	Interval	Pipeline type
dct	6	16	2360	566	479	106	dataflow
dct_read_data	0	0	29	55	66	66	none
dct_Loop_Row_DCT_Loop_proc	0	8	600	114	105	105	none
dct_Loop_Xpose_Row_Outer_Loop_proc	0	0	29	57	66	66	none
dct_Loop_Col_DCT_Loop_proc	0	8	600	114	105	105	none
dct_Loop_Xpose_Col_Outer_Loop_proc	0	0	30	65	66	66	none
dct_write_data	0	0	32	63	66	66	none

Figure 6-49: Performance Analysis After the INLINE Directive

Now that the *dct_2d* function is inlined, observe that the *dct_2d* entry is replaced with the following:

- *dct_Loop_Row_DCT_Loop_proc*
- *dct_Loop_Xpose_Row_Outer_Loop_proc*
- *dct_Loop_Col_DCT_Loop_proc*
- *dct_Loop_Xpose_Col_Outer_Loop_proc*

Also observe that all the functions are operating in parallel, yielding the top-level function interval (throughput) of 106 clock cycles.

7-5-3. Switch to the Synthesize perspective.

Applying the RESHAPE Directive Step 8

The ARRAY_RESHAPE directive combines ARRAY_PARTITIONING with the vertical mode of ARRAY_MAP and is used to reduce the number of block RAM while still allowing the beneficial attributes of partitioning: parallel access to the data.

The ARRAY_RESHAPE directive allows more data to be accessed in a single clock cycle. In cases where more data can be accessed in a single clock cycle, the Vivado HLS tool may automatically unroll any loops consuming this data, if doing so will improve the throughput. The loop can be fully or partially unrolled to create enough hardware to consume the additional data in a single clock cycle.

8-1. Create a new solution by copying the previous solution (*solution6*) settings.

8-1-1. Select **Project** > **New Solution** or click the **New Solution** icon ( ) from the toolbar.

8-1-2. Leave the options at their default settings (with *solution6* selected).

8-1-3. Click **Finish**.

8-2. Apply the ARRAY_SHAPE directive to the *buf_2d_in* array of the dct function, the *col_inbuf* array of the *dct_2d* function, and the *dct_coeff_table* array of the *dct_1d* function.

8-2-1. Select **Project** > **Close Inactive Solution Tabs** to close all inactive solution windows.

8-2-2. Ensure that the **dct.c** file is open and active in the Information pane.

8-2-3. In the Directive tab, right-click the **ARRAY_PARTITION** directive of the *buf_2d_in* array of the *dct* function and select **Modify Directive**.

8-2-4. Select **ARRAY_SHAPE** from the Directive drop-down list.

8-2-5. Select the *type (optional)* as **complete**.

8-2-6. Enter **2** in the *dimension (optional)* field.

8-2-7. Click **OK**.

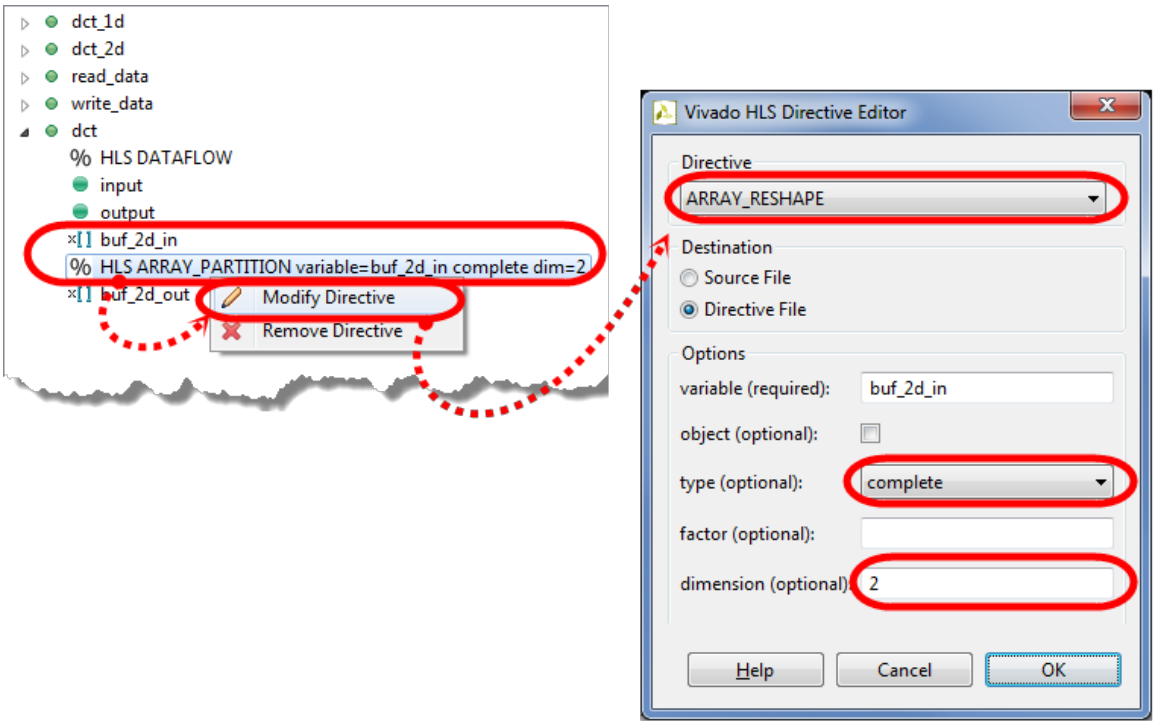


Figure 6-50: Applying the ARRAY_RESHAPE Directive

8-2-8. Similarly, change the ARRAY_PARTITION directive applied to the *col_inbuf* array of the *dct_2d* function in the Directive pane to **ARRAY_RESHAPE** with a dimension of **2** and and type as **complete**.

8-2-9. Assign the **ARRAY_RESHAPE** directive with dimension of **2** and type as **complete** to the *dct_2coeff_table* array.

The final view of the ARRAY_RESHAPE directives in the Directive tab should look similar to the figure below.

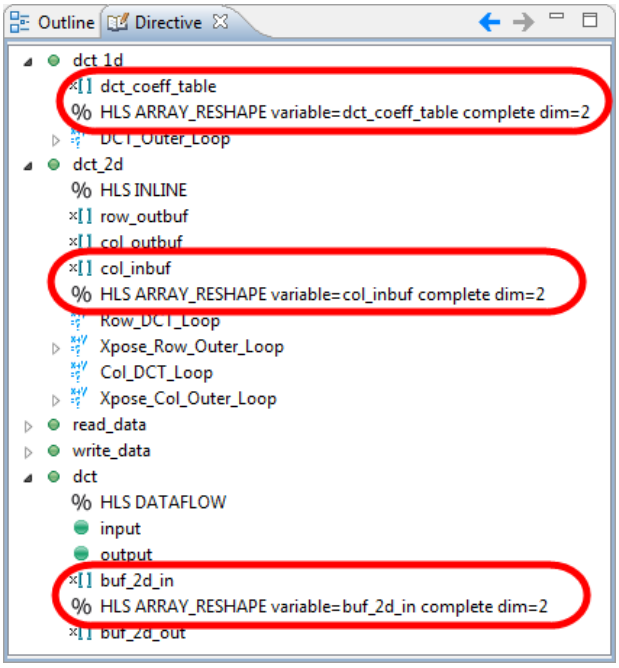


Figure 6-51: Final View of ARRAY_RESHAPE Directives

8-3. Synthesize the Vivado HLS design.

If you do not recall how to perform this task, refer to the "Synthesizing the Vivado HLS Design" section under Vivado HLS Operations in the *Lab Reference Guide*.

8-4. View the Synthesis report.

You should see that the Synthesis report in the Information window opens automatically when synthesis is completed.

8-4-1. Observe that the latency increased from 479 to 607 clock cycles and the DATAFLOW pipeline throughput increased from 106 to 131 clock cycles in the Performance Estimates.

The block RAM resource utilization increased from 6 to 22.

8-4-2. Examine the synthesis log to see what transformations were applied automatically.

Reviewing the synthesis log will provide some clues. There are warnings in the scheduling phase for *read_data* stating that II=1 could not be achieved. In fact, *read_data* complains about conflicting read and write operations.

The problem here is due to the fact that an update to a single element in a reshaped array requires that the entire word be read, the single element updated, and the entire

word written back. An array that has been reshaped requires a read-modify-write cycle (the Vivado HLS tool does not implement byte-masking on writes).

This operation negatively impacts the maximum write bandwidth for such an array.

Thus it can be seen the directives have to be applied carefully.

8-4-3. Select **File** > **Exit** to close the Vivado HLS tool.

Summary

In this lab, you learned various techniques to improve performance and balance resource utilization.

The PIPELINE directive, when applied to an outer loop, will automatically cause the inner loop to unroll. When a loop is unrolled, resource utilization increases as operations are done concurrently. Partitioning memory may improve performance but will increase block RAM utilization.

When the DATAFLOW directive is applied, the default memory buffers (of ping-pong type) are automatically inserted between the top-level functions and loops. When the INLINE directive is applied to a function, the lower level hierarchy is automatically dissolved.

The RESHAPE directive will allow multiple accesses to block RAM. However, care should be taken if a single element requires modification as it will result in a read-modify-write operation for the entire word.

The Analysis perspective and console logs can provide insight on what is going on.

Answers

1. Looking at the report, fill in the table below.

Estimated clock period	6.38
Worst case latency	3959
Number of DSP48E used	1
Number of BRAMs used	5
Number of FFs used	278
Number of LUTs used	353

Performance and Utilization Estimates

Lab 7: HLx Flow – System Integration

2016.1

Abstract

This lab introduces the HLx design flow for generating IP from the Vivado® HLS tool and using the generated IP to create a subsystem with the MicroBlaze™ processor using the Vivado IP integrator.

This lab should take approximately 75 minutes.

Objectives

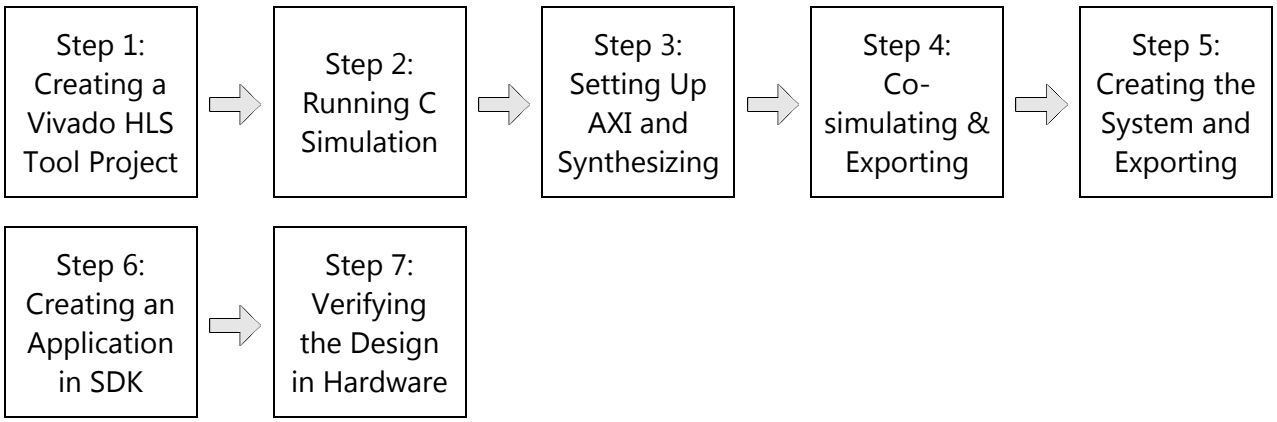
After completing this lab, you will be able to:

- Describe the HLx design flow
- Identify the steps and directives involved in creating an IP within the Vivado HLS tool
- Create a system with the MicroBlaze processor and IP (fir filter) created with the Vivado HLS tool using the IP integrator flow
- Use software and run it on the created subsystem

Introduction

In this lab, you will use a C design for an **11 tap FIR (Finite Impulse Response) filter**. This lab requires you to develop an IP of the designed filter using the Vivado HLS tool. Using the IP integrator flow, you will create a system with the MicroBlaze processor and the IP created in the HLS tool. In the Software Development Kit (SDK), you will use the test application provided to write into and read from the IP interface registers.

General Flow



Creating a Vivado HLS Tool Project Step 1

In this step, you will create a new Vivado HLS project to implement a FIR filter.

There are a number of ways to launch the Vivado HLS tool. The two most popular mechanisms are shown here.

1-1. Launch the Vivado HLS tool.

1-1-1. Select **Start** > **All Programs** > **Xilinx Design Tools** > **Vivado 2016.1** > **Vivado HLS** > **Vivado HLS 2016.1**.

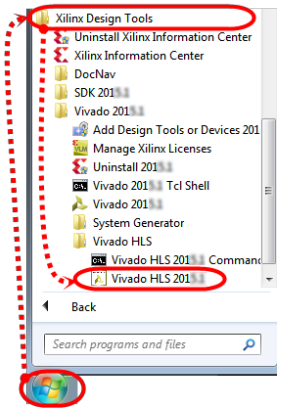


Figure 7-1: Launching the Vivado HLS Tool

-- OR --

Double-click the **Vivado HLS** shortcut icon () on the desktop.

The Vivado HLS tool opens to the Welcome window. From the Welcome window you can create a new project, open examples, and access documentation and examples.

Figure 7-2: Vivado HLS Welcome Screen

Here you will learn to create a new Vivado HLS project from scratch.

1-2. Create a Vivado HLS project named *fir_prj*.

1-2-1. From the Welcome Page, click **Create New Project**.

Figure 7-3: Creating a New Vivado HLS Project

1-3. The Project Configuration dialog box asks for a project name and location.

1-3-1. Enter **fir_prj** in the Project name field (1).

1-3-2. Enter **C:\training\hls\labs\hlx_system_integration\KC705\fir** in the Location field (2).

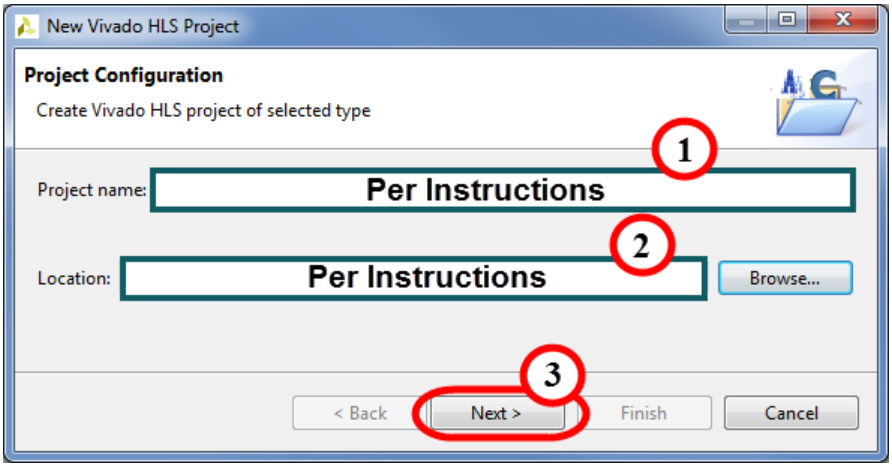


Figure 7-4: Configuring a New HLS Project

1-3-3. Click **Next** (3).

1-4. The Add/Remove Files dialog box opens. Here you will be invited to add existing files or create new sources.

1-4-1. Click **Add Files**.

The Open File dialog box opens.

Note: If do not have existing files at this moment and you want to create new ones, click **New File**.

1-4-2. Browse to *C:\training\hls\labs\hlx_system_integration\KC705\fir*.

1-4-3. Select **fir.c**.

The Vivado HLS tool automatically adds the working directory (project directory) and any directory that contains C files added to the project to the search path. Hence, header files that reside in these directories are automatically included in the project (no need to explicitly specify them). You must specify the path to all other header files (if any) by clicking the Edit CFLAGS button.

1-4-4. Click **Open** to add these files.

Note that you can add compiler directives specific to each entry at this point.

1-4-5. Click **Browse** next to the Top Function field.

The Select Top function dialog box opens, which lists all the functions available from the specified source files.

1-4-6. Select **fir (fir.c)** from the list.

Note: You can also manually enter the name of the top function in the Top Function field.

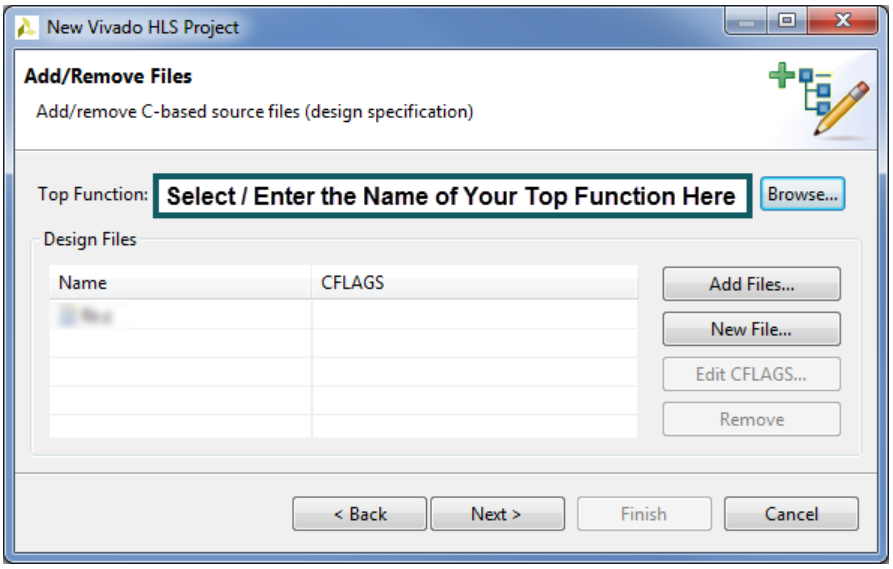


Figure 7-5: Adding Files to a New Vivado HLS Project

1-4-7. Click **Next**.

1-5. Add any existing testbench files.

If you have (or want) any testbench files they can be entered here. Sometimes the testbench is built into the synthesizable file.

1-5-1. Click **Add Files**.

1-5-2. Navigate to *C:\training\hls\labs\hlx_system_integration\KC705\fir*.

1-5-3. Select **fir_test.c, out.gold.dat, and out.gold.8.dat**.

1-5-4. Click **Open** to add these files.

1-5-5. Click **Next**.

1-6. Finally, it is time to specify some of the physical parameters of the design.

1-6-1. By default, **solution1** is populated in the Solution Name field.

No changes are required.

1-6-2. Set the clock period to **10**.

You can leave the Uncertainty field blank.

1-6-3. Click the **Browse** button to select a part or board.

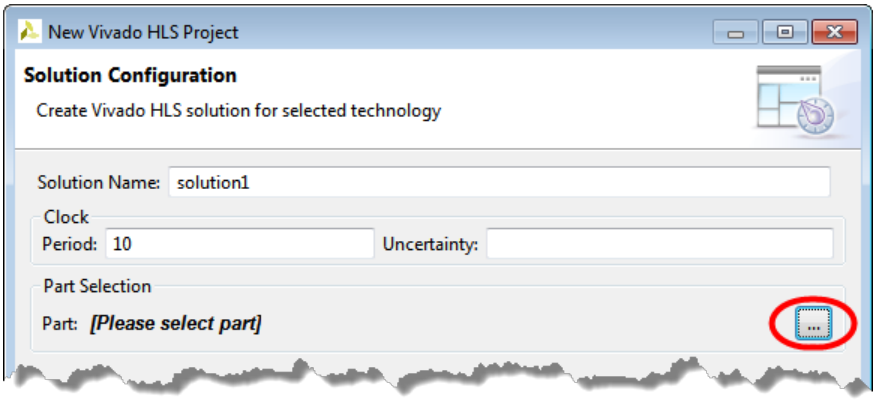


Figure 7-6: Locating the Board Browse Button

1-6-4. Click **Boards** as shown below.

1-6-5. Enter **kintex** in the Search field.

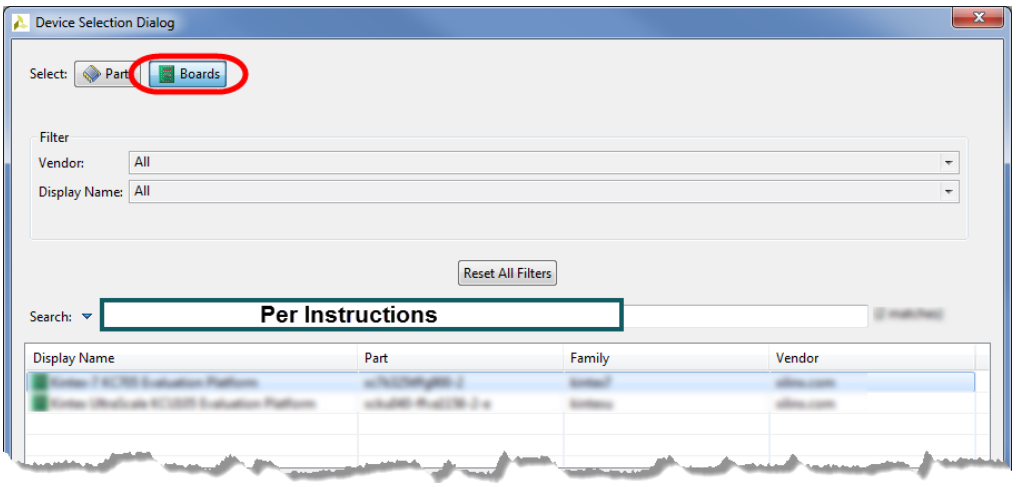


Figure 7-7: Filtering to Quickly Locate Target Platforms

1-6-6. Select **Kintex-7 KC705 Evaluation Platform** from the search list.

1-6-7. Click **OK** to select the board.

1-6-8. Click **Finish**.

1-6-9. You will see the created project in the Explorer tab.

1-7. Explore the design.

1-7-1. Expand the various sub-folders to see the entries under each sub-folder.

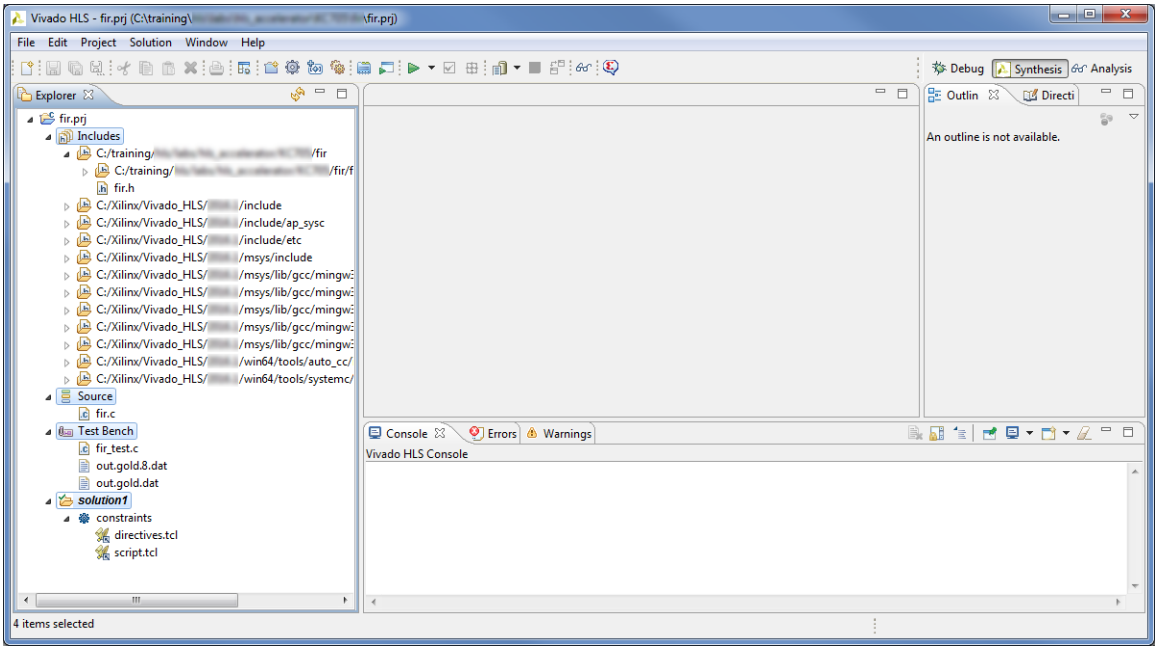


Figure 7-8: Newly Created Vivado HLS Tool Project

1-7-2. In the Explorer tab, expand the **Source** folder.

1-7-3. Double-click **fir.c**.

This will open the *fir.c* file in the Information pane.

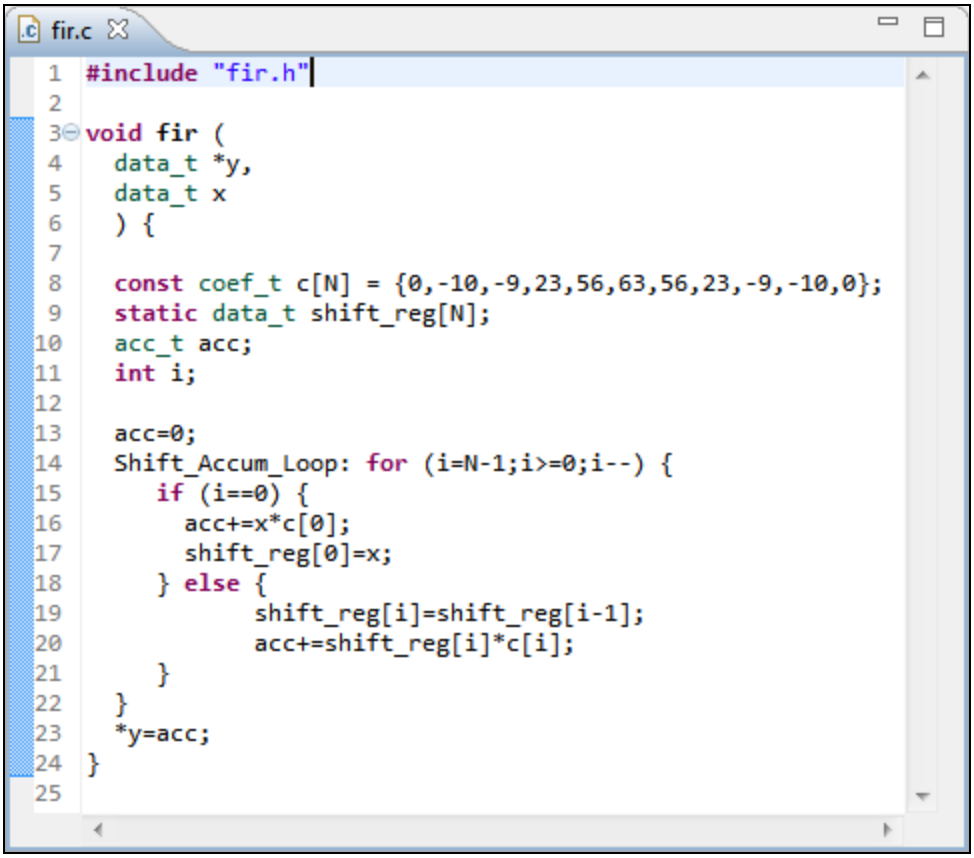


Figure 7-9: Design Under Consideration

The FIR filter expects x as a sample input and a pointer to the computed sample out. Both of these are defined as having a data type, data_t. The coefficients are loaded in array c of type coef_t through initialization. The sequential algorithm is applied and the accumulated value is computed out.

1-7-4. In the Explorer tab, expand the **Includes** > **C:/training/hls/labs/hlx_system_integration/KC705/fir** folder.

1-7-5. Double-click **fir.h** to open it.

This header file has the constant and type declarations associated with the *fir.c* design.

1-7-6. Similarly, open the **fir_test.c** file located in the *fir.prj* > *Test Bench* folder to explore its contents.

Running C Simulation Step 2

In this step, you will validate the FIR filter design with the provided C testbench.

2-1. Simulate the Vivado HLS tool design.

2-1-1. Select **Project** > **Run C Simulation** or click the **Run C Simulation** icon ( ).

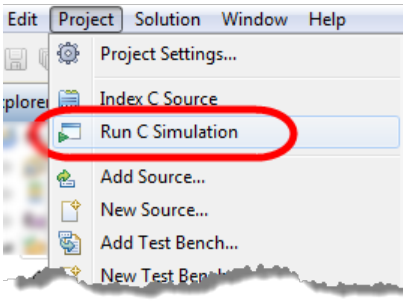


Figure 7-10: Launching the C Simulation

The Run C Simulation dialog box opens.

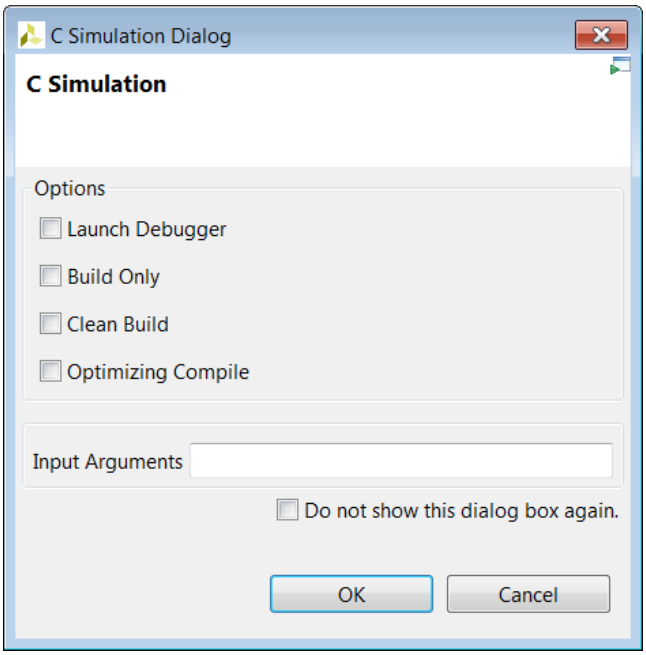


Figure 7-11: C Simulation Dialog Box

Each of the options controls how simulation is run:

- Launch Debugger: After compilation the debug perspective automatically opens for you to step through the code.
- Build Only: Compile the object/netlist files but do not execute.
- Clean Build: Remove previously compiled files before compiling.

- Optimizing Compile: Use the gcc/g++ -O option (no debug info and this is mutually exclusive with debug options; this may run faster, but the difference is not substantial).

2-1-2. Select **the default options (i.e. select nothing)**.

2-1-3. Click **OK**.

The simulation log will be displayed in the editor pane.

2-2. View the simulation report.

The information generated by the Vivado HLS tool can be found in two places, both described here.

The first is the Console window, which reports not only the output produced by the code being simulated, but all of the simulation engine messages as well. The simulation log provides only a few simulation engine messages and the simulated code output.

2-2-1. Select the **Console** tab in the lower portion of the tool's GUI.

You may need to scroll to view all the output produced by the simulation.

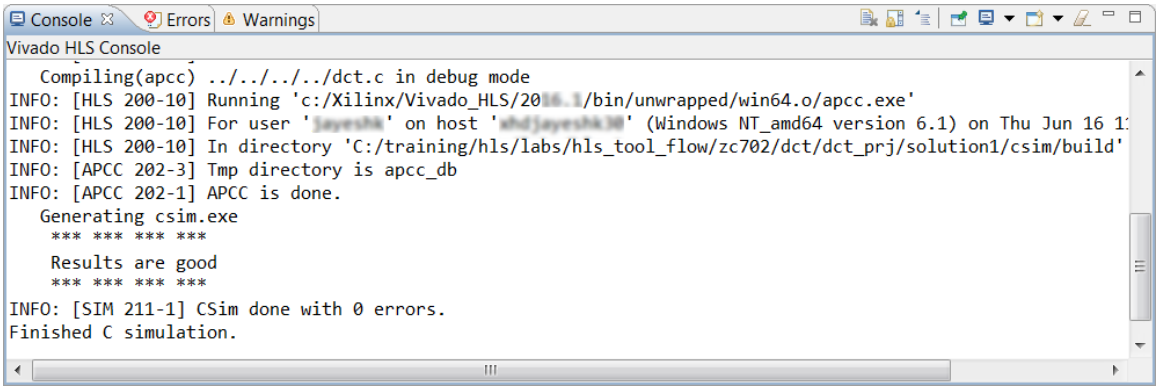


Figure 7-12: Example Output After a Simulation

The other location, described below, provides only a few simulation engine messages and the simulated code output. Typically this is opened after the simulation completes; however, if you need to access it after closing the log pane, here's how to access the simulation report.

2-2-2. Expand **fir_prj** > **solution1** > **csim** > **report** in the Explorer pane.

2-2-3. Double-click the log file name to open it in the editor pane.

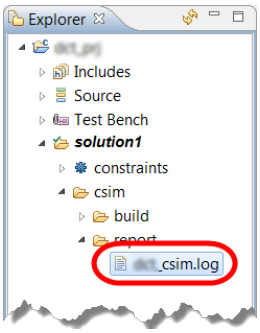


Figure 7-13: Locating the Simulation Log File

If you do not see the above message, ask for help from your instructor.

This testbench is a self-checking testbench; that is, the computed output is compared against a reference golden output and returns whether the test passed or failed.

Setting Up the AXI Lite Adapters and Synthesizing the Design Step 3

In this step, you will apply the Resource directive to the FIR interface signals to generate an AXI Lite interface for the filter. You will also perform synthesis on the design.

3-1. Add the Resource directive to create the AXILiteS adapters.

3-1-1. Select the **fir.c** tab in the Editor window (1).

3-1-2. In the Auxiliary pane, select the **Directive** tab (2).

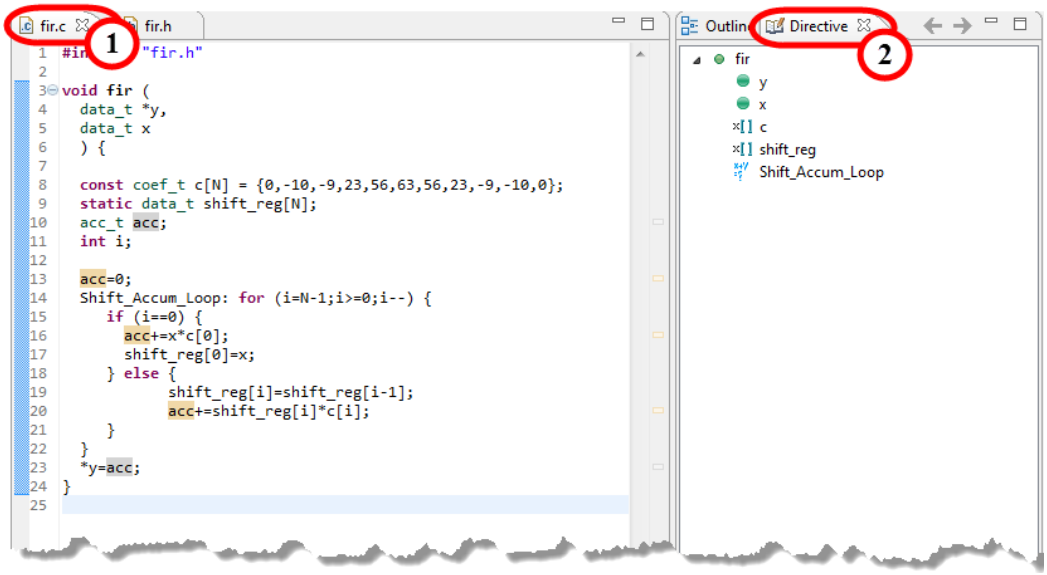


Figure 7-14: Selecting the Directive Tab

3-1-3. In the Directive tab, right-click the variable **x** and select **Insert Directive**.

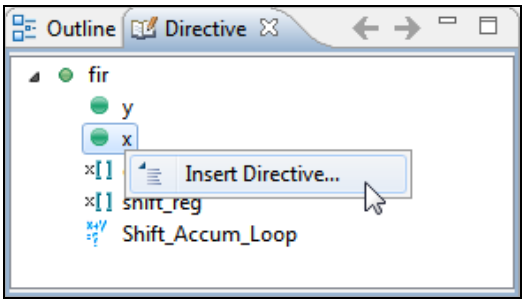


Figure 7-15: Inserting a Directive for the Variable x

The Vivado HLS Directive Editor dialog box opens.

3-1-4. Select **INTERFACE** from the Directive drop-down list (1).

3-1-5. Select **s_axilite** from the mode (optional) drop-down list (2).

3-1-6. Enter **fir_io** in the bundle (optional) field to associate the input and output into one s_axilite interface called *fir_io* (3).

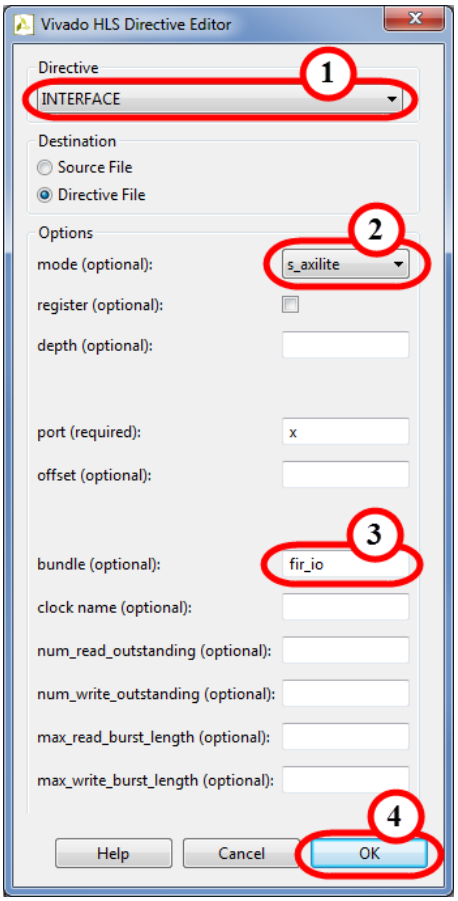


Figure 7-16: Selecting the s_axilite Interface Directive

3-1-7. Click **OK** (4).

3-1-8. Repeat the previous steps to apply the INTERFACE directive to the **y** output.

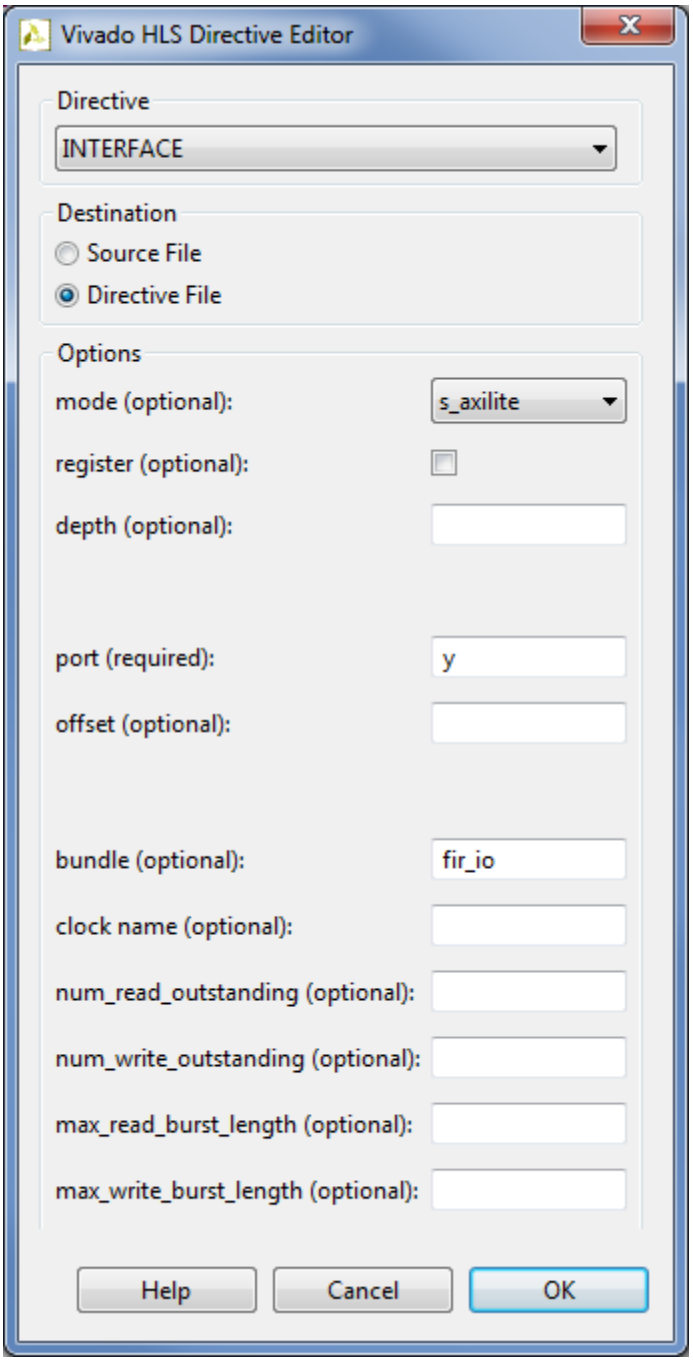


Figure 7-17: Inserting the Directive for the Variable y

When the design is synthesized, this will generate the ap_start signal.

3-1-9. Repeat the previous steps to apply the INTERFACE directive to the top-level module **fir** in the Directive tab.

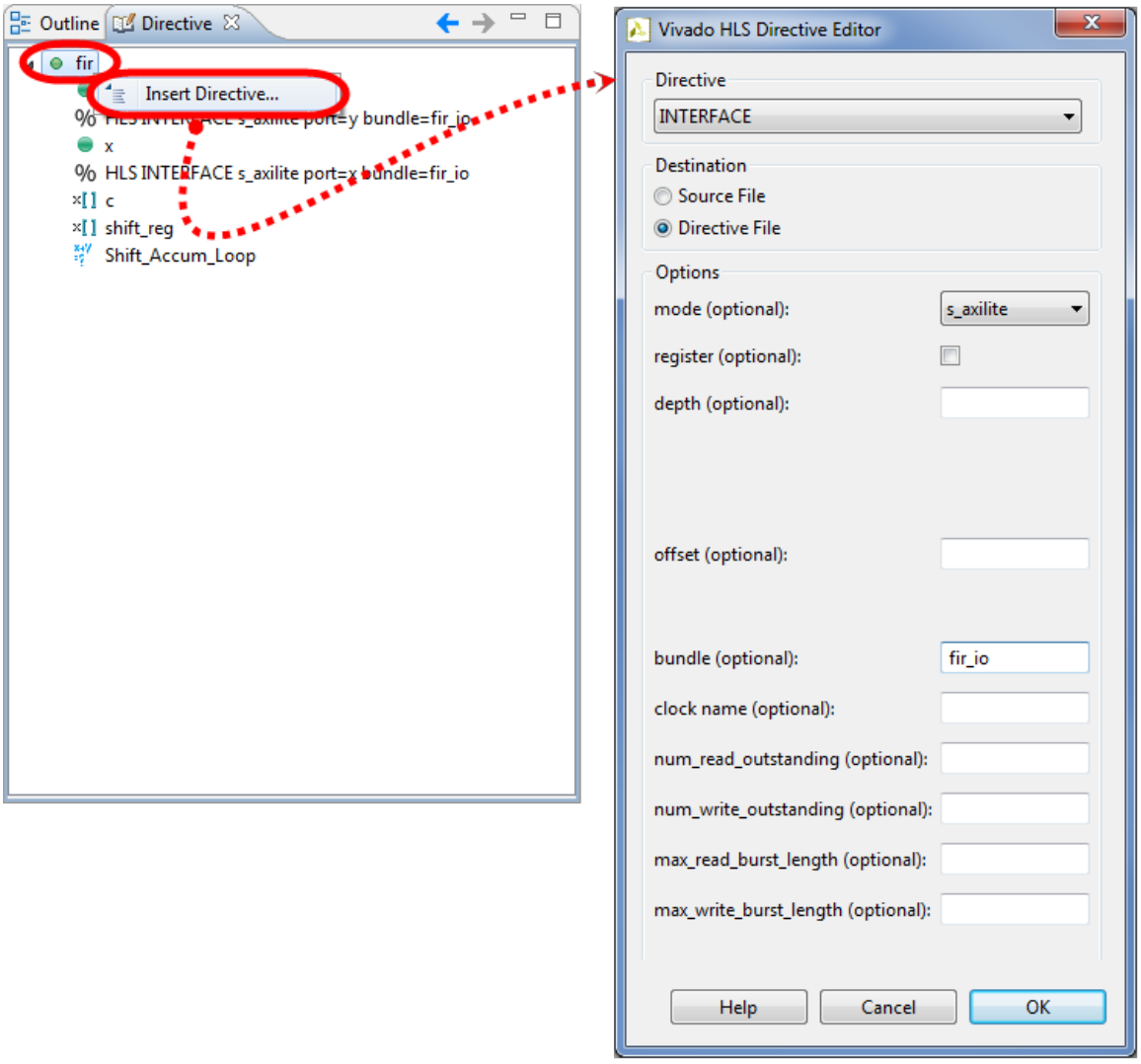


Figure 7-18: Inserting the Directive for the Top Module FIR

This will create address maps for the x, y, ap_start, ap_valid, ap_done, and ap_idle signals, which can be accessed via software.

Alternatively, the ap_start, ap_valid, ap_done, and ap_idle signals can be generated as separate ports on the core if you do not apply this directive. These ports will then have to be connected in a processing system using the GPIO IP core.

3-2. Synthesize the design.

3-2-1. Select **Solution** > **Run C Synthesis** > **Active Solution** or click the **Run Synthesis** icon in the menu bar.

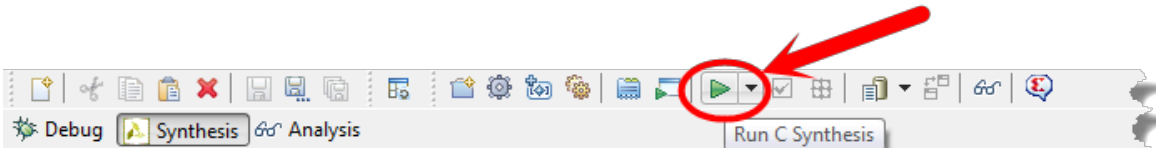


Figure 7-19: Launching Synthesis

This option synthesizes the currently selected solution.

All solutions (or selected solutions) can be synthesized by using the drop-down menu next to the synthesis icon. You can synthesize all solutions or synthesize selected solutions in addition to the default.

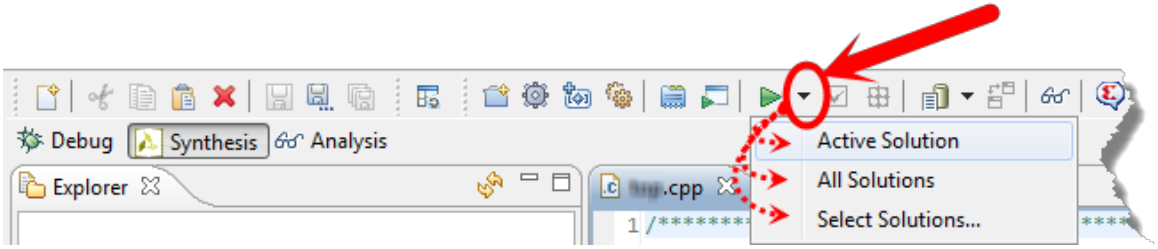


Figure 7-20: Options for What to Synthesize

When synthesis completes, the Synthesis report will be displayed in the Information pane.

3-2-2. Review the Synthesis report.

3-2-3. Go to the Interface summary and view the I/O ports generated by the tool.

Performing Co-simulation and Exporting the Design as an IP Step 4

Now that you have completed high-level synthesis on the C design, you will perform RTL co-simulation on the generated RTL using the C testbench and then export the design as an IP.

4-1. Cosimulate the Vivado HLS tool design.

4-1-1. Select **Solution** > **Run C/RTL Cosimulation** or click the **Run C/RTL Cosimulation** icon ( ).

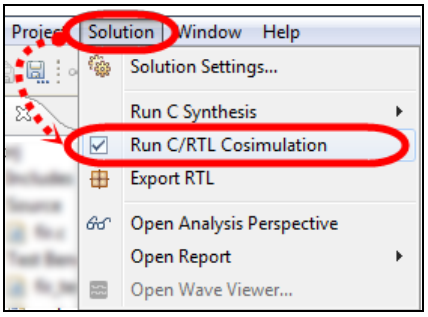


Figure 7-21: Launching from the Menu

The Run C/RTL Co-simulation dialog box opens.

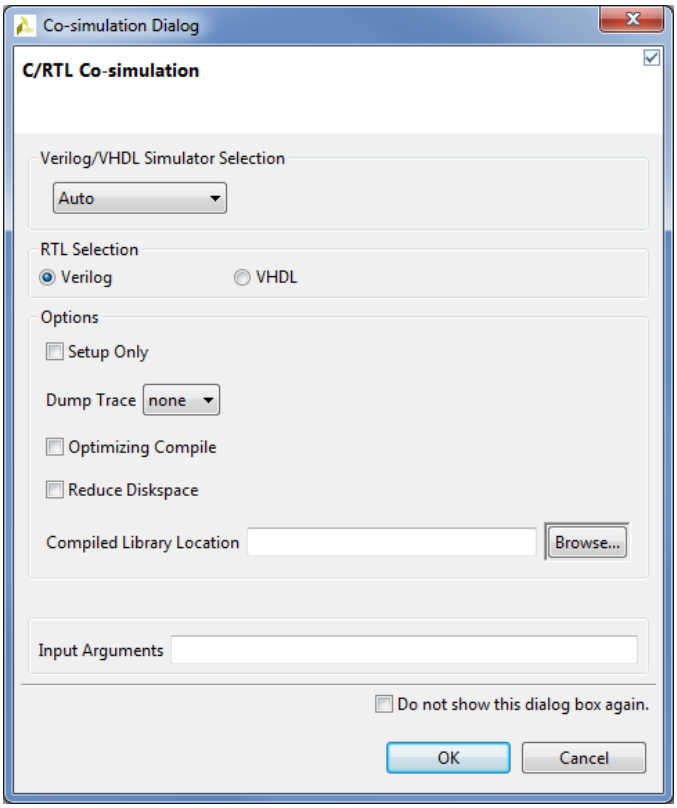


Figure 7-22: Co-simulation Dialog Box

Each of the options controls how C/RTL Cosimulation is run:

- RTL Selection: Select the RTL that is simulated (Verilog/VHDL).
- Setup Only: This creates all the files (wrappers, adapters and scripts) required to run the simulation but does not execute the simulator.
- Dump Trace: During RTL verification, the trace files can be saved and viewed using an appropriate viewer. By selecting this option, the trace file will be saved to the *<solution>/sim/<RTL>* folder.
- Optimizing Compile: This ensures a high level of optimization is used to compile the C testbench. Using this option increases the compile time but the simulation executes faster.

4-1-2. Select **the default options (i.e. select nothing)**.

4-1-3. Click **OK**.

The simulation log will be displayed in the editor pane.

4-2. View the Cosimulation report.

The information generated by the Vivado HLS tool can be found in two places, both described here.

The first is the Console window, which reports not only the output produced by the code being simulated, but all of the simulation engine messages as well. The simulation log provides only a few simulation engine messages and the simulated code output.

4-2-1. Select the **Console** tab in the lower portion of the tool's GUI.

You may need to scroll to view all the output produced by the cosimulation.

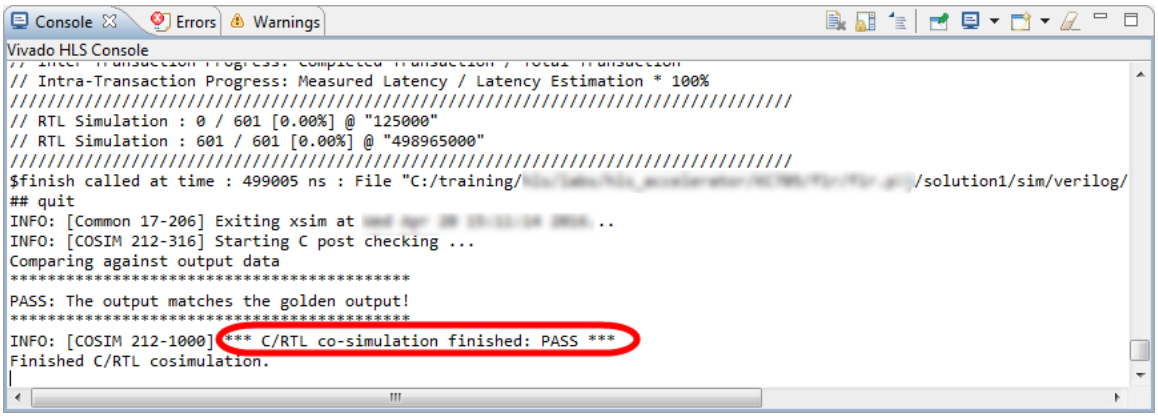


Figure 7-23: Example Output After a C/RTL Co-simulation

The other location, described below, provides only a few simulation engine messages and the simulated code output. Typically this is opened after the simulation completes; however, if you need to access it after closing the log pane, here's how to access the simulation report.

4-2-2. Expand **fir_prj** > **solution1** > **sim** > **report** in the Explorer pane.

4-2-3. Double-click the log file name to open it in the editor pane.

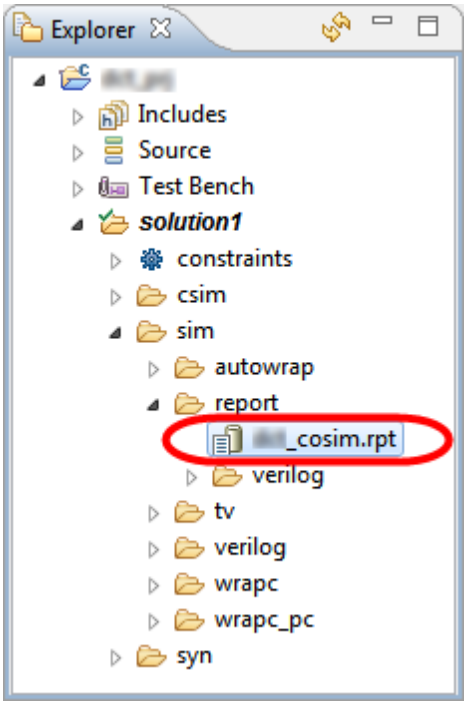


Figure 7-24: Locating the Co-Simulation Log File

The Cosimulation Report in HTML format will be displayed in the main viewing area.

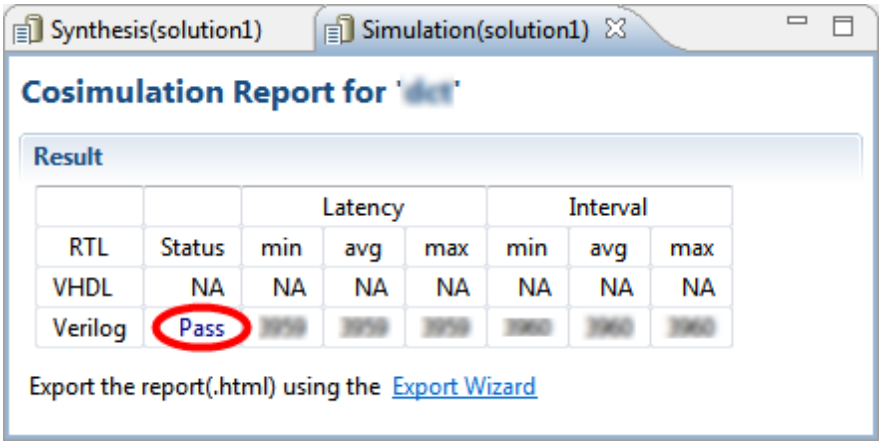


Figure 7-25: Cosimulation Report – HTML

You can quickly verify the cosimulation status here.

You will now export the RTL as an IP core to be used with the top-level design.

4-3. Export the RTL, selecting Verilog as the language.

4-3-1. Select **Solution** > **Export RTL**.

Alternatively, you can click the  toolbar button.

4-3-2. Ensure that **IP Catalog** is selected from the Format Selection drop-down list.

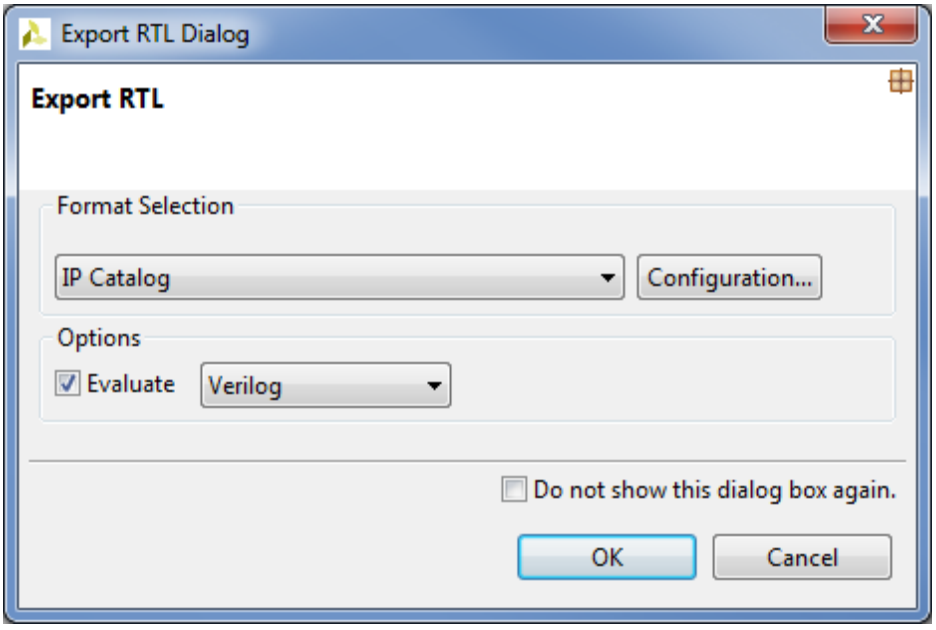


Figure 7-26: Export RTL Dialog Box

4-3-3. Make certain that the **Evaluate** option is selected and that **Verilog** is selected as the RTL to be evaluated.

This will perform Vivado RTL synthesis and implementation on the generated IP. Implementation is run to evaluate and provide confidence that the RTL will meet its estimated timing and area goals and that these results are not included as part of the exported package.

4-3-4. Click **OK** in the Export RTL dialog box.

You can observe the progress in the Console tab. When the run is complete, the Implementation Report is displayed in the Information pane. The final timing of the implemented design has been achieved.

The exported IP is available in the *<solution_directory>\impl\ip* folder.

4-3-5. When implementation completes, expand the **impl\ip** folder under *solution1* in the Explorer tab view and observe the various generated folders and files (IP, hardware files, software drivers, and IP documentation).

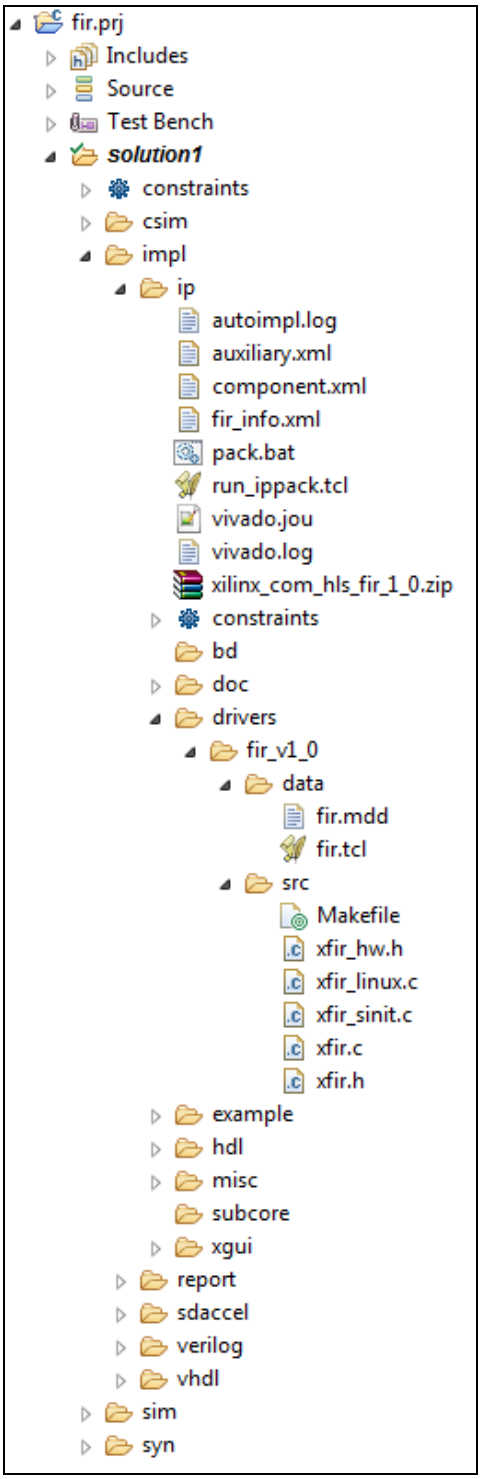


Figure 7-27: Exploring the Exported IP under the impl Directory

4-3-6. Select **File** > **Exit** to close the Vivado HLS tool.

Creating a MicroBlaze Processor System Using the Vivado IPI and Exporting Hardware to SDK Step 5

In this step you will create a MicroBlaze processor system using the Vivado IP integrator (IPI), add the IP generated in the Vivado HLS tool, generate the bitstream for the system, and export the design to the SDK tool.

There are a number of ways to launch the Vivado Design Suite. The two most popular mechanisms are shown here.

5-1. Launch the Vivado Design Suite.

This can be done in two standard ways, use your preferred method.

5-1-1. Select **Start** > **All Programs** > **Xilinx Design Tools** > **Vivado 2016.1** > **Vivado 2016.1**.

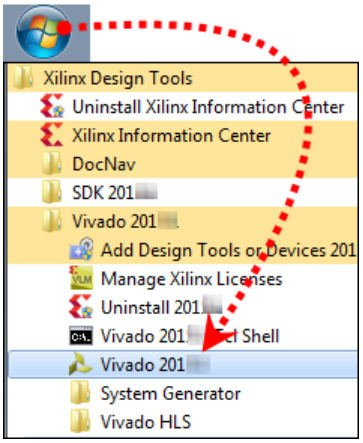


Figure 7-28: Launching the Vivado Design Suite from the Start Menu

-- OR --

Double-click the **Vivado Design Suite** shortcut icon () on the desktop.

The Vivado Design Suite opens to the Welcome window. From the Welcome window you can create a new project, open an existing project, or enter Tcl commands directly into the Vivado Design Suite as well as access documentation and examples.

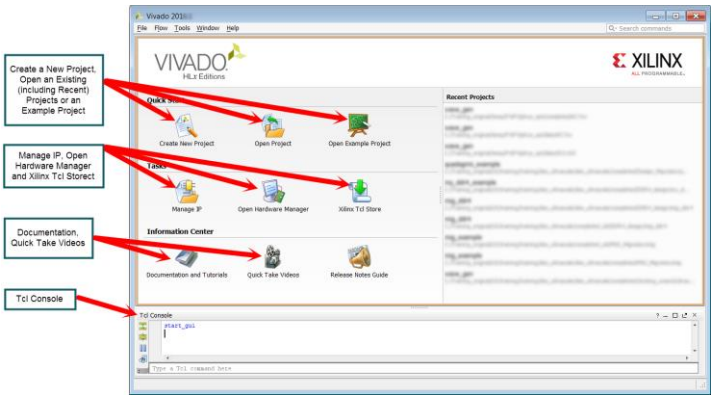


Figure 7-29: Vivado Design Suite Welcome Screen

This launches the New Project wizard.

"Create New Project" is the starting point for all designs. Projects contains sources, settings, graphics, IP, and other elements that are used to build a final bitstream and analyze a design. The Create New Project Wizard in the Vivado Design Suite allows you to specify HDL and other project resource files that will be included in the project.

5-2. Create a new blank Vivado Design Suite project.

5-2-1. Click **Create New Project** to begin the process (1).

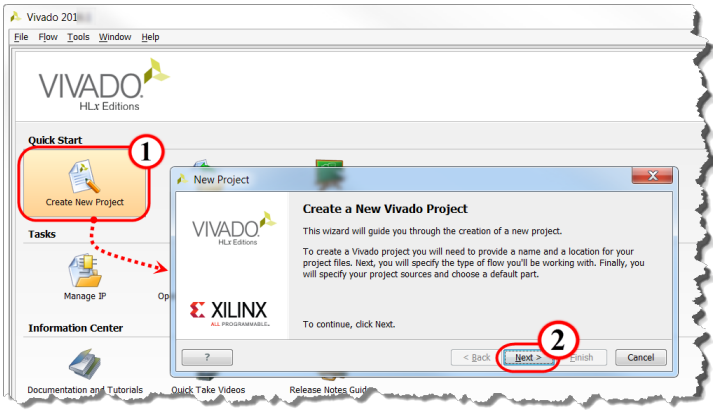


Figure 7-30: Creating a New Vivado Design Suite Project

This will launch the New Project Wizard.

5-2-2. Click **Next** to exit the introductory dialog box and begin entering in project-specific information (2).

5-3. You will now encounter a series of dialog boxes asking you to enter different pieces of information describing the project.

5-3-1. Enter **system** in the Project name field.

5-3-2. Enter **C:\training\HLx_Flow\labs\hlx_system_integration\KC705\fir** in the Project location field.

Alternatively, you can use the browse feature to navigate to where you want the project to reside.

5-3-3. Deselect the **Create Project Subdirectory** option as leaving this checked will create an unnecessary level of hierarchy for this lab.

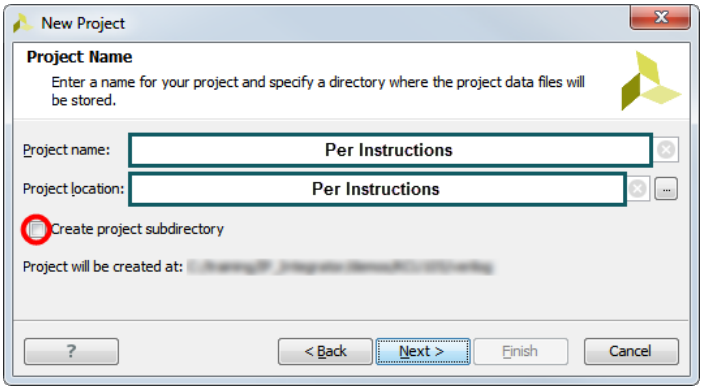


Figure 7-31: Entering the Project Name and Location

5-3-4. Click **Next** to advance to the next dialog box.

Here you will choose between an RTL project or a post-synthesis project. Simply put, an RTL project enables you to add or create new HDL files and synthesize them, whereas the post-synthesis project requires pre-synthesized files. When an empty design is created, an RTL project is used.

5-3-5. Select **RTL Project** (1).

5-3-6. Select **Do not specify sources at this time** (2), which creates a blank project.

While existing sources could be entered at this time, you will enter them later so that you can to move through this portion of the project creation process more quickly.

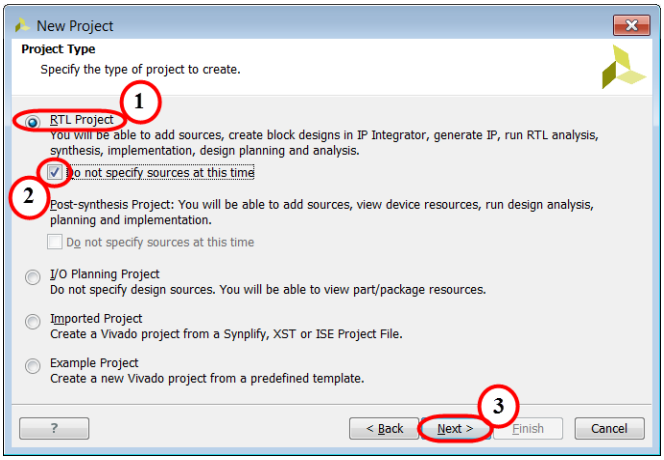


Figure 7-32: Selecting Project Type

5-3-7. Click **Next** to advance to the target device/platform selection (3).

5-4. Select the target part by first filtering by board and then by family. If you are not using a supported board, you will need to filter by part.

5-4-1. Select **Boards** from the Select area to filter by board rather than by the specific part (1).

5-4-2. Select **All** from the Vendor drop-down list in the Filter area (2).

This limits the number of boards seen to those manufactured by the specified vendor.

5-4-3. Select **Kintex-7 KC705 Evaluation Platform** from the board list.

Alternatively you can select the board directly from the list at any time while in this dialog box.

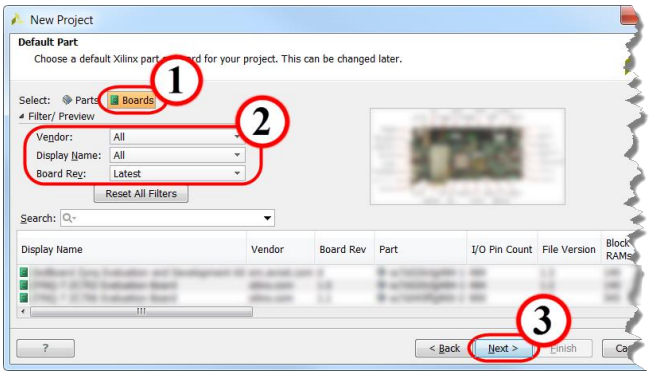


Figure 7-33: Selecting the Board for the Project

5-4-4. Click **Next** to advance to the summary (3).

A summary of your project is displayed. If you want to change any of the information that you entered, you can do so now by clicking **Back** until you reach the correct dialog

box and making the correction, or you can create the project now and edit the project properties, add or remove files, etc. later.

5-4-5. Click **Finish** to accept these settings and build the project.

Your project is constructed and leaves you in the operational portion of the Vivado Design Suite GUI.

5-5. Add the IP generated by the Vivado HLS to tool to the IP catalog.

5-5-1. Select **Tools** > **Project Settings**.

5-5-2. Click **IP** in the Project Settings dialog box.

5-5-3. Select the **Repository Manager** tab in the dialog box.

5-5-4. Click **+** to add a repository.

5-5-5. Select the **C:\training\HLx_Flow\labs\hlx_system_integration\KC705\fir\fir_prj\solution1\impl\ip** directory.

5-5-6. Click **OK** in the Add Repository dialog box.

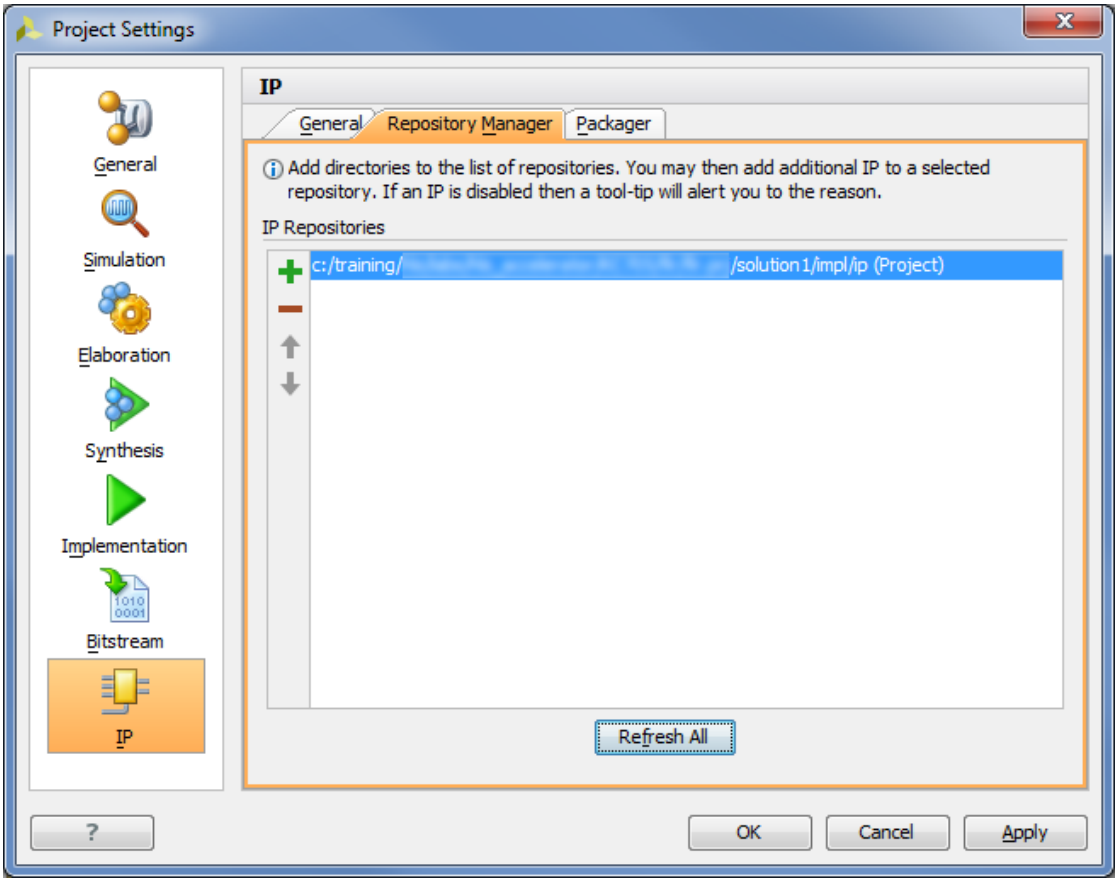


Figure 7-34: Adding Vivado HLS Tool IP to the IP Repository

5-5-7. Click **OK** to add the Fir IP generated in the Vivado HLS tool to the IP catalog.

5-5-8. Select **IP Catalog** from the Flow Navigator under *Project Manager*.

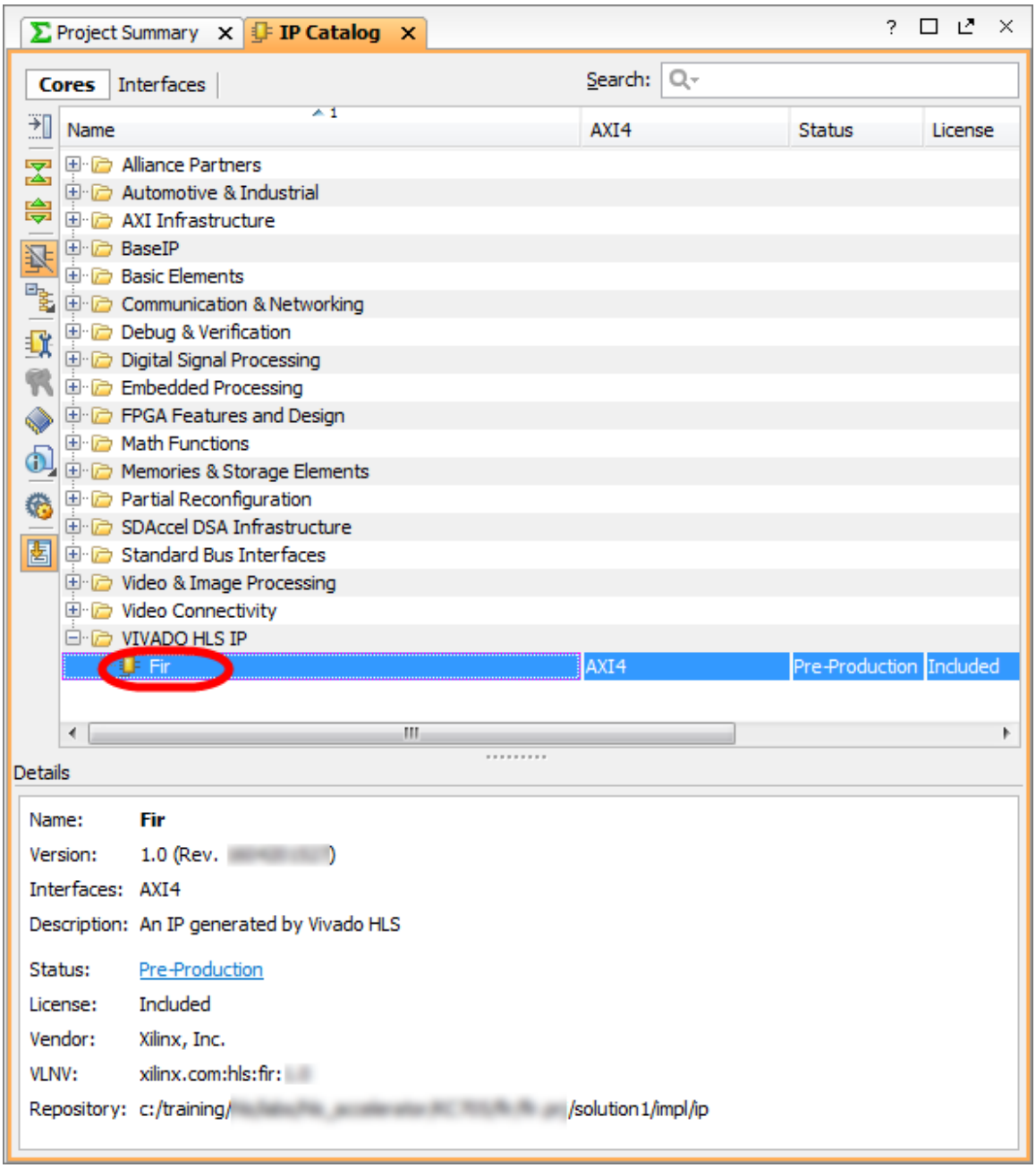


Figure 7-35: IP Catalog with Vivado HLS Tool IP

Note that the Vivado HLS tool IP is added to the IP catalog.

5-5-9. Click **X** in the IP Catalog tab to close the IP catalog.

The Vivado IP integrator is a graphical tool that assists you in "stitching" together various pieces of IP. This tool can be used for both embedded and non-embedded designs.

5-6. Create an IP integrator block diagram.

5-6-1. If necessary, expand the **IP Integrator** field under the Flow Navigator (1).

5-6-2. Click **Create Block Design** to start creating a new IP subsystem (2).

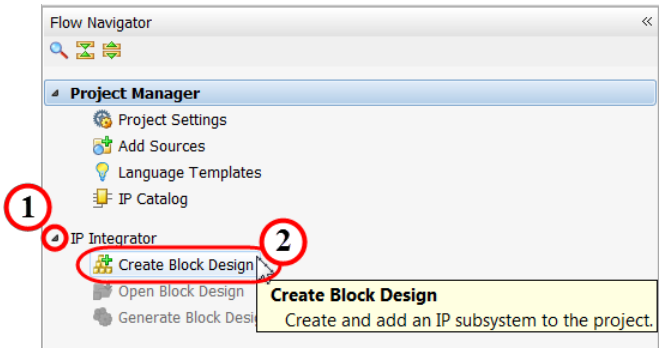


Figure 7-36: Launching the IP Integrator

5-6-3. When the Create Block Design dialog box opens, name the design **microblaze_design** (1).

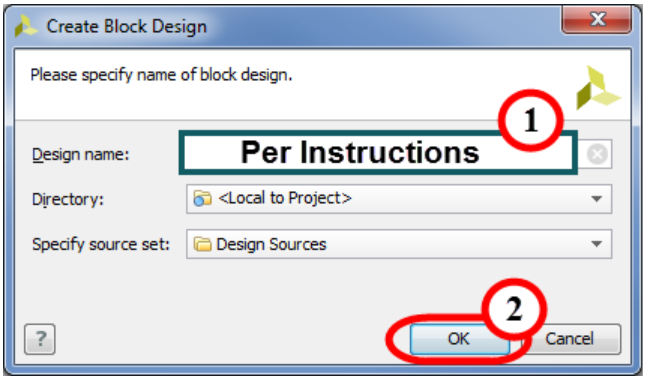


Figure 7-37: Creating an IP Integrator Block Design

5-6-4. Click **OK** to open a new, blank IP integrator canvas (2).

The IP integrator workspace opens with a note in the canvas area inviting you to begin adding IP.

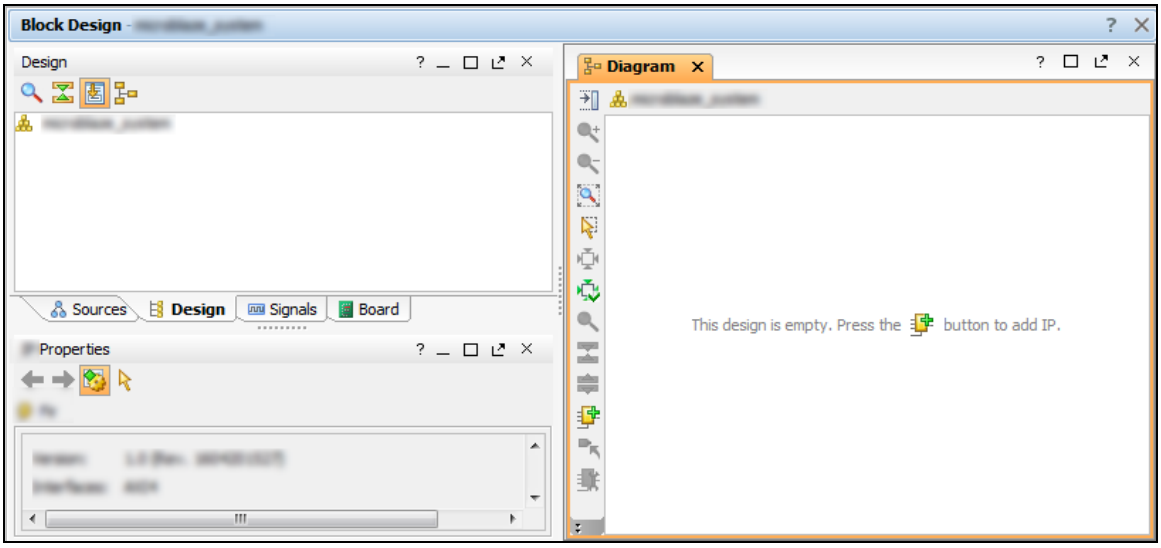


Figure 7-38: Initial View of the IP Integrator Tool Showing an Informational Message

5-7. Add a MicroBlaze processor to the IP integrator canvas.

5-7-1. Open the IP catalog in one of two ways:

- Click the **Add IP** icon on the left border of the workspace.

Figure 7-39: Opening the IP Catalog from the Add IP Icon

-- OR --

- Right-click any background space in the workspace and select **Add IP**.

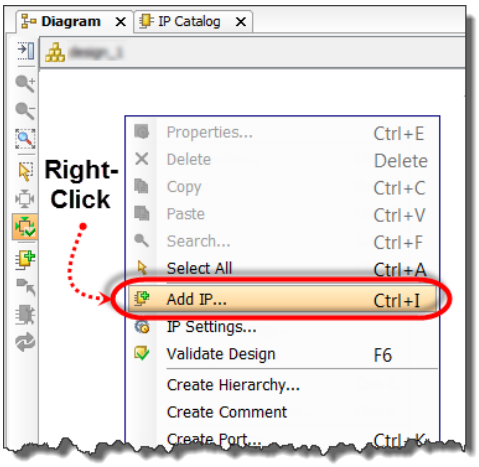


Figure 7-40: Opening the IP Catalog from Right-Clicking the Workspace

Important Note: Using Windows > IP Catalog will add the new IP to the top level of hierarchy of the design—it will NOT add the IP to the diagram! You can, however, float the Windows > IP Catalog and drag-and-drop from the catalog onto the canvas of the block design

Once the IP catalog opens, you can search for the processor block.

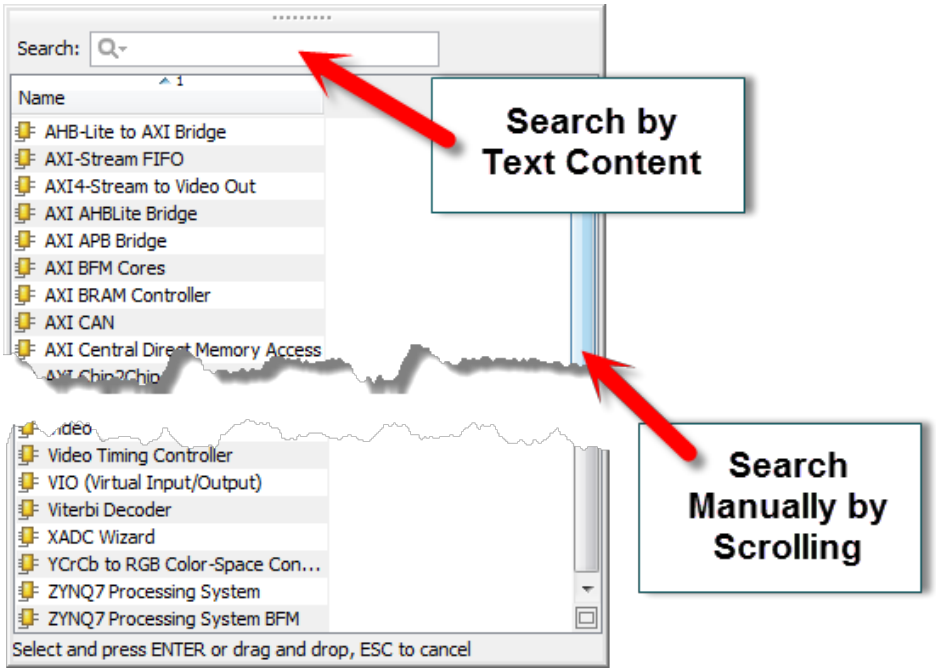


Figure 7-41: Two Mechanisms to Search for IP

5-7-2. Enter part of the processor name into the Search field to narrow the search parameters (1).

For example, **zynq** for the Zynq All Programmable SoC or **microblaze** for the MicroBlaze processor. Note that the Zynq AP SoC PS IP will only appear when Zynq-7000 AP SoCs or boards with these parts have been selected for the design.

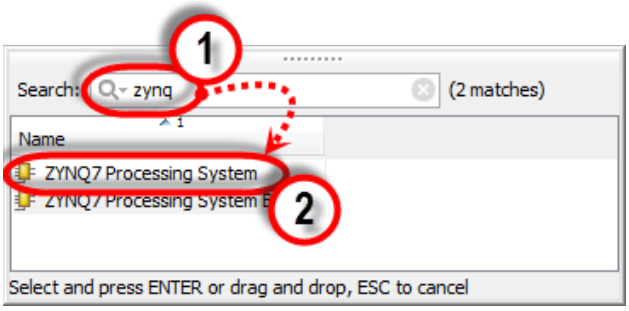


Figure 7-42: Narrowing Search Parameters (Zynq AP SoC)

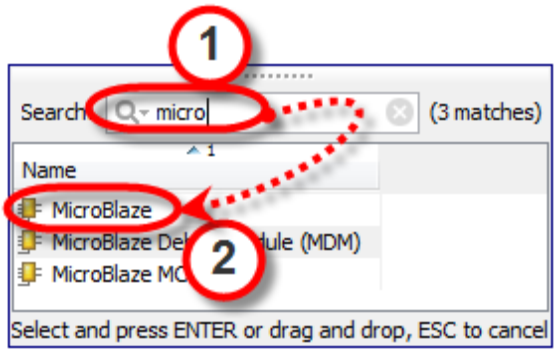


Figure 7-43: Narrowing Search Parameters (MicroBlaze Processor)

5-7-3. Double-click the **MicroBlaze** processor name entry to add its IP block to the design (2).

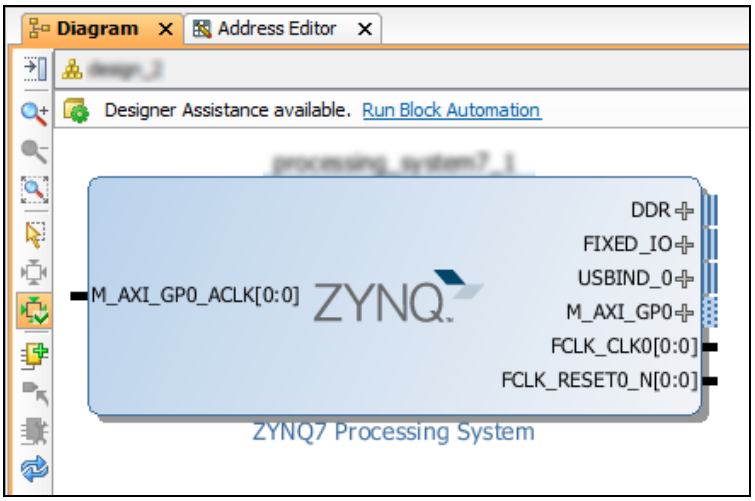


Figure 7-44: Zynq Processing System IP

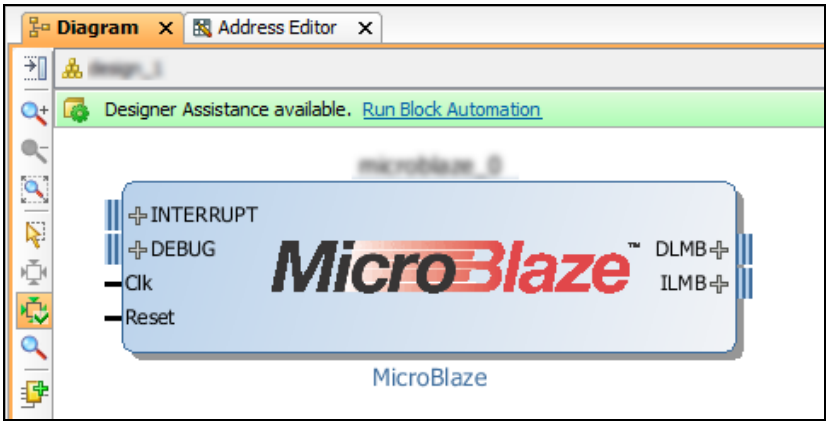


Figure 7-45: MicroBlaze Processor IP

5-8. Configure the MicroBlaze processor to include the following options:

- **Local Memory: 16KB**
- **Local Memory ECC: None**
- **Cache Configuration: 16KB**
- **Debug Module: Debug Only**
- **Peripheral AXI port: Enabled**
- **Interrupt controller: Enabled**
- **Clock Connection: New Clocking Wizard (100 MHz)**

5-8-1. Click **Run Block Automation**.

5-8-2. Configure the options according to the figure below.

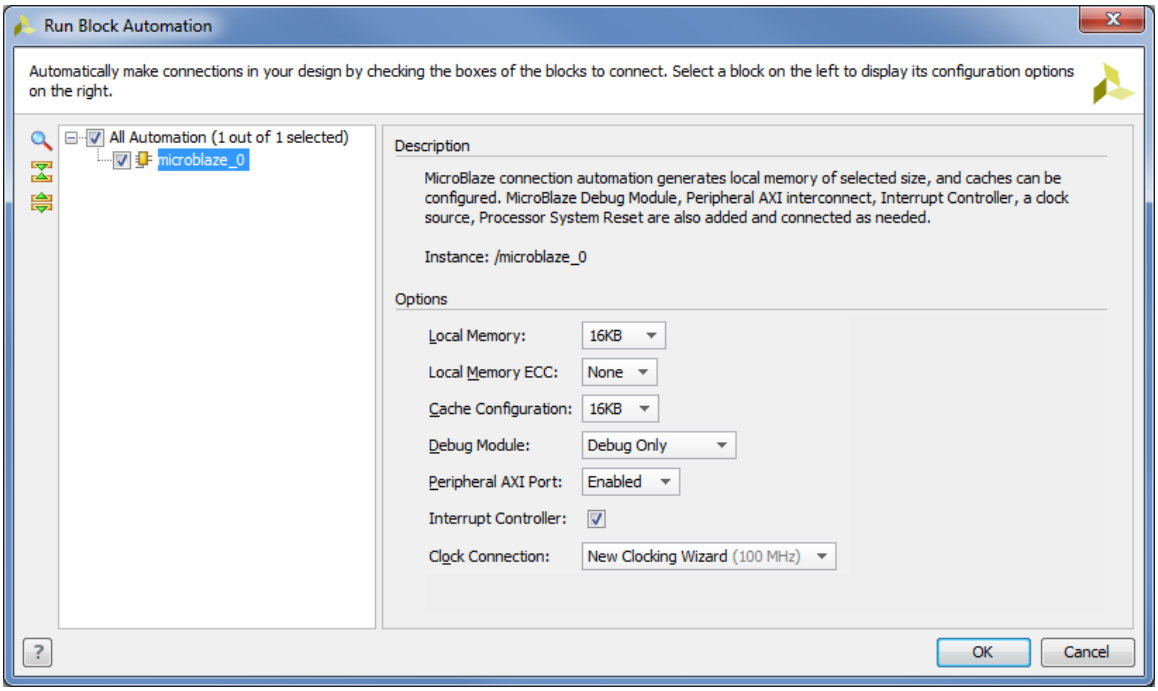


Figure 7-46: MicroBlaze Processor Options

5-8-3. Click **OK**.

5-8-4. Click the **Regenerate Layout** icon ( ) in the vertical toolbar on the left side of the canvas.

You should see the following block diagram.

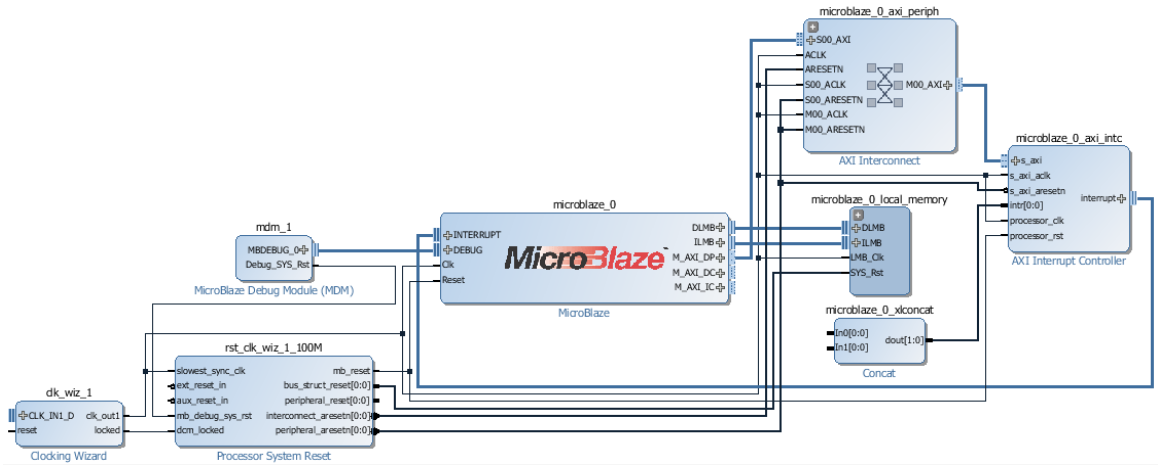


Figure 7-47: MicroBlaze Processor Configuration

5-9. Make the connections to the external clock and reset.

5-9-1. Click **Run Connection Automation**.

5-9-2. Select the interfaces to connect to the external clock and reset as shown in the figure below.

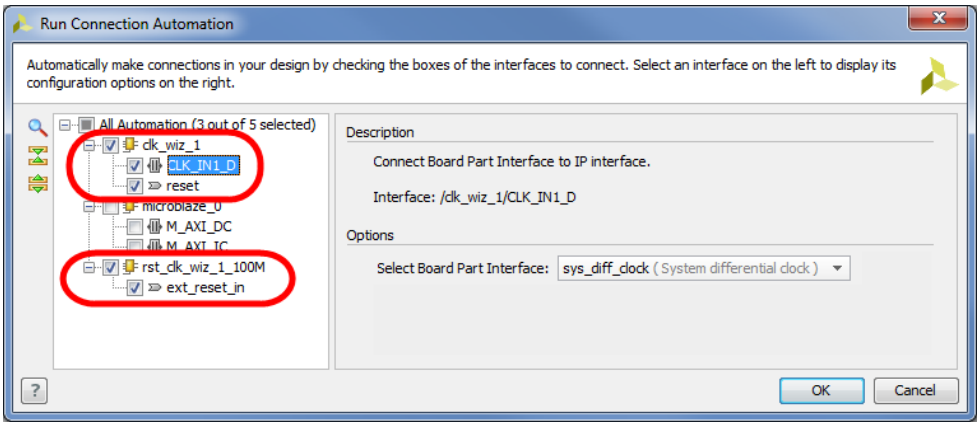


Figure 7-48: Connections to the External Clock and Reset

5-9-3. Click **OK**.

5-10. Add and connect the axi_uartlite to the design.

5-10-1. Right-click in the Diagram tab window and select **Add IP**.

5-10-2. In the Search field, enter **uart**.

5-10-3. Double-click **AXI Uartlite**.

The AXI Uartlite will appear in the Diagram window.

5-10-4. Click **Run Connection Automation**.

5-10-5. Select the interfaces to connect as shown in the figure below.

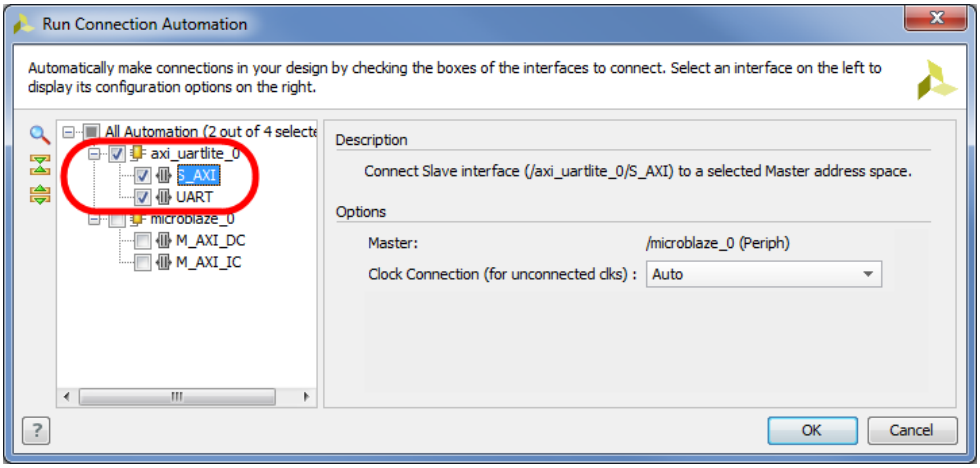


Figure 7-49: Connections to the Slave Interface and UART Ports

5-10-6. Click **OK**.

5-11. Configure the AXI Uartlite.

5-11-1. Double-click the **AXI Uartlite** block.

The Re-customize IP window opens.

5-11-2. In the IP Configuration tab, set the baud rate as **115200**.

5-11-3. Click **OK**.

5-12. Add the Fir IP (generated from the Vivado HLS tool) to the design.

5-12-1. Right-click the block diagram and select **Add IP**.

5-12-2. In the Search field, enter **Fir**.

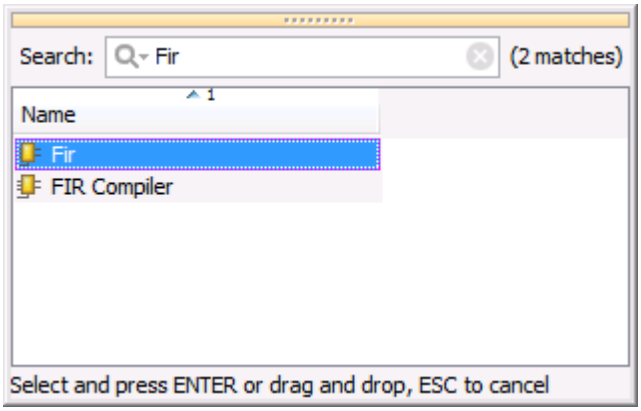


Figure 7-50: Searching for the Fir IP in the IP Catalog

5-12-3. Double-click **Fir** to add the Fir IP to the block diagram.

5-13. Run Connection Automation.

5-13-1. Click **Run Connection Automation** from the Information bar.

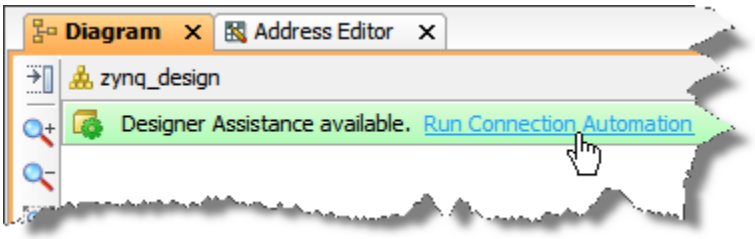


Figure 7-51: Running Connection Automation

5-13-2. Select the slave interface to connect as shown below.

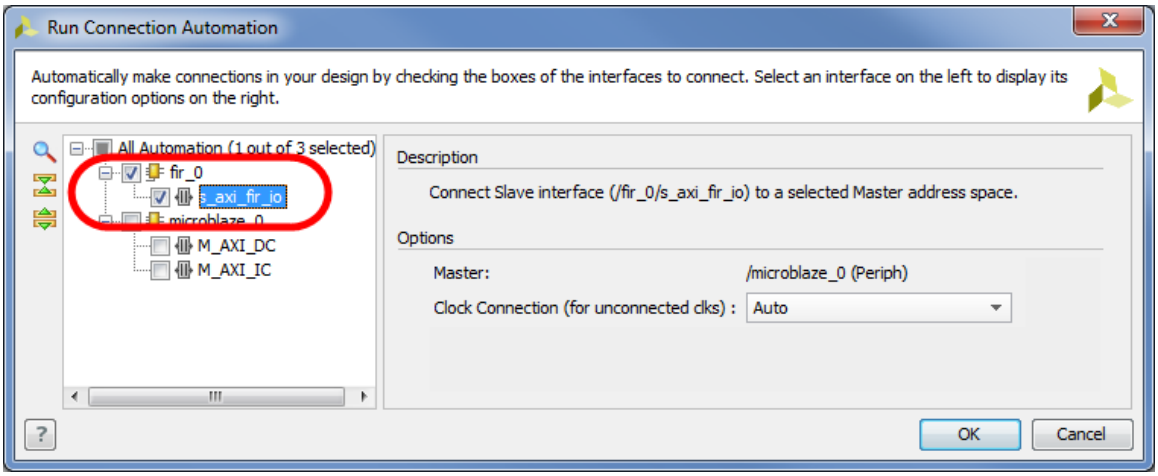


Figure 7-52: Connections to the FIR Slave Interface

5-13-3. Click **OK** in the Run Connection Automation dialog box.

This will automatically connects the clock, reset and slave interface.

5-14. Connect the FIR interrupt to the interrupt controller to complete the design.

5-14-1. Connect the **interrupt** port of the fir_0 instance to the **In0** port of the *Concat* instance.

5-14-2. Double-click the **Concat** instance.

5-14-3. In the Re-customize IP window, change the Number of Ports to **1**.

5-15. Regenerate the layout.

5-15-1. Click the **Regenerate Layout** icon ( ) in the vertical toolbar on the left side of the canvas.

The block diagram should resemble the following figure.

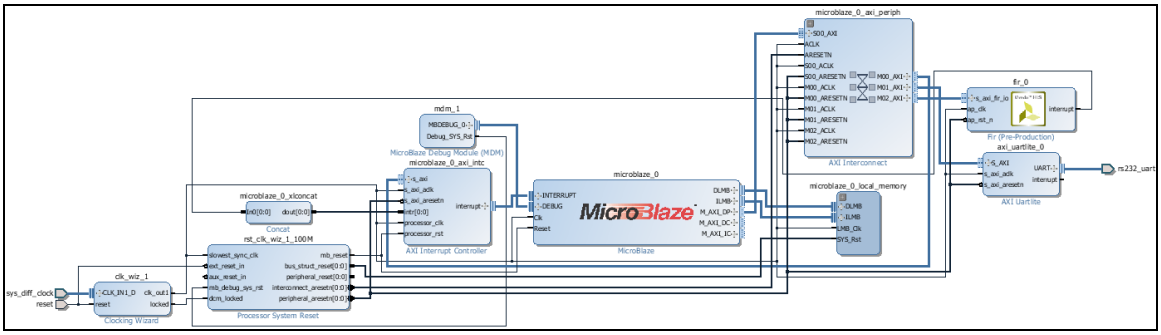


Figure 7-53: Completed Design

5-16. Verify that addresses are assigned to the peripherals.

5-16-1. Select the **Address Editor** tab.

5-16-2. From the vertical toolbar, click **Expand All** ( ).

The addresses are automatically assigned to the peripherals when connection automation was run.

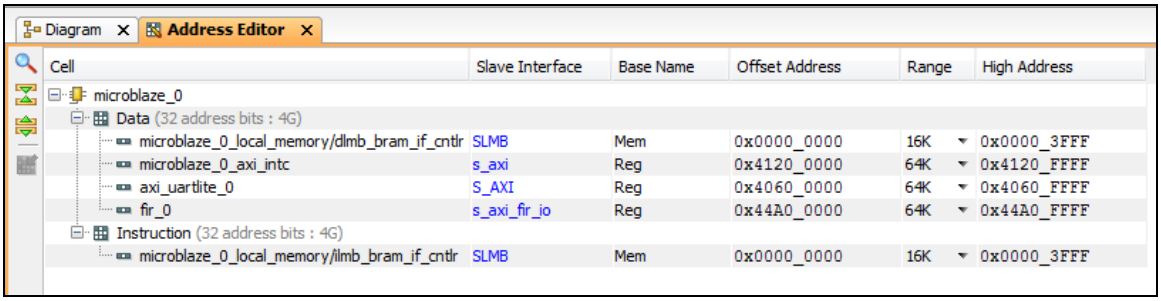


Figure 7-54: Performance estimate after applying DATAFLOW directive

5-16-3. Click the **Validate Design** ( ) vertical toolbar button in the Diagram tab.

You should see a validation successful dialog box. If you see any errors in the design, ask for help from your instructor.

5-16-4. Click **OK** in the Validate Design dialog box.

5-16-5. Select **File** > **Save Block Design** to save the design.

5-17. Generate the bitstream for the design.

5-17-1. In the Sources window, right-click **microblaze_system (microblaze_system.bd)** and select **Create HDL Wrapper**.

5-17-2. Accept the default settings and click **OK** in the Create HDL Wrapper dialog box.

5-17-3. In the Flow Navigator, click **Generate Bitstream** under Program and Debug.

5-17-4. Click **Yes** in the No Implementation Results Available dialog box if it appears.

Generating the bitstream will take approximately five minutes.

5-17-5. Once the bitstream generation completed dialog box appears, select **Open Implemented Design** and click **OK**.

5-17-6. Select **File** > **Export** > **Export Hardware**.

5-17-7. In the Export Hardware for SDK dialog box, select the **Include bitstream** option.

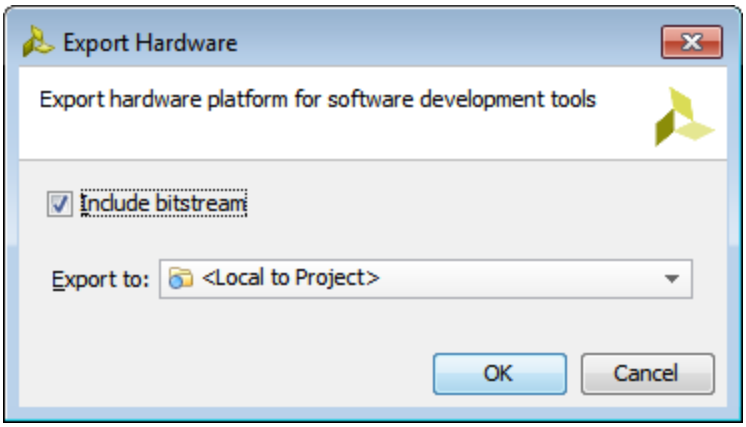


Figure 7-55: Exporting Hardware to SDK

5-17-8. Click **OK**.

This exports the hardware to the path that is local to the project.

5-17-9. Select **File** > **Launch SDK**.

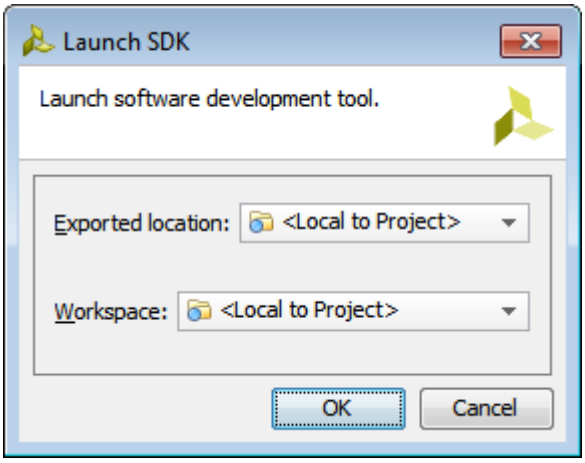


Figure 7-56: Launch SDK Dialog Box

5-17-10. Accept the default settings and click **OK**.

This will launch the SDK tool.

Creating an Application in SDK Step 6

In this step you will create an application in the SDK environment to test the FIR functionality.

6-1. Add the Fir IP drivers to the SDK repository.

6-1-1. Select **Xilinx Tools** > **Repositories**.

6-1-2. Click **New** in the Local Repositories section.

6-1-3. Browse to the *C:\training\hls\labs\hlx_system_integration\KC705\fir\fir_prj\solution1\impl\ip* directory.

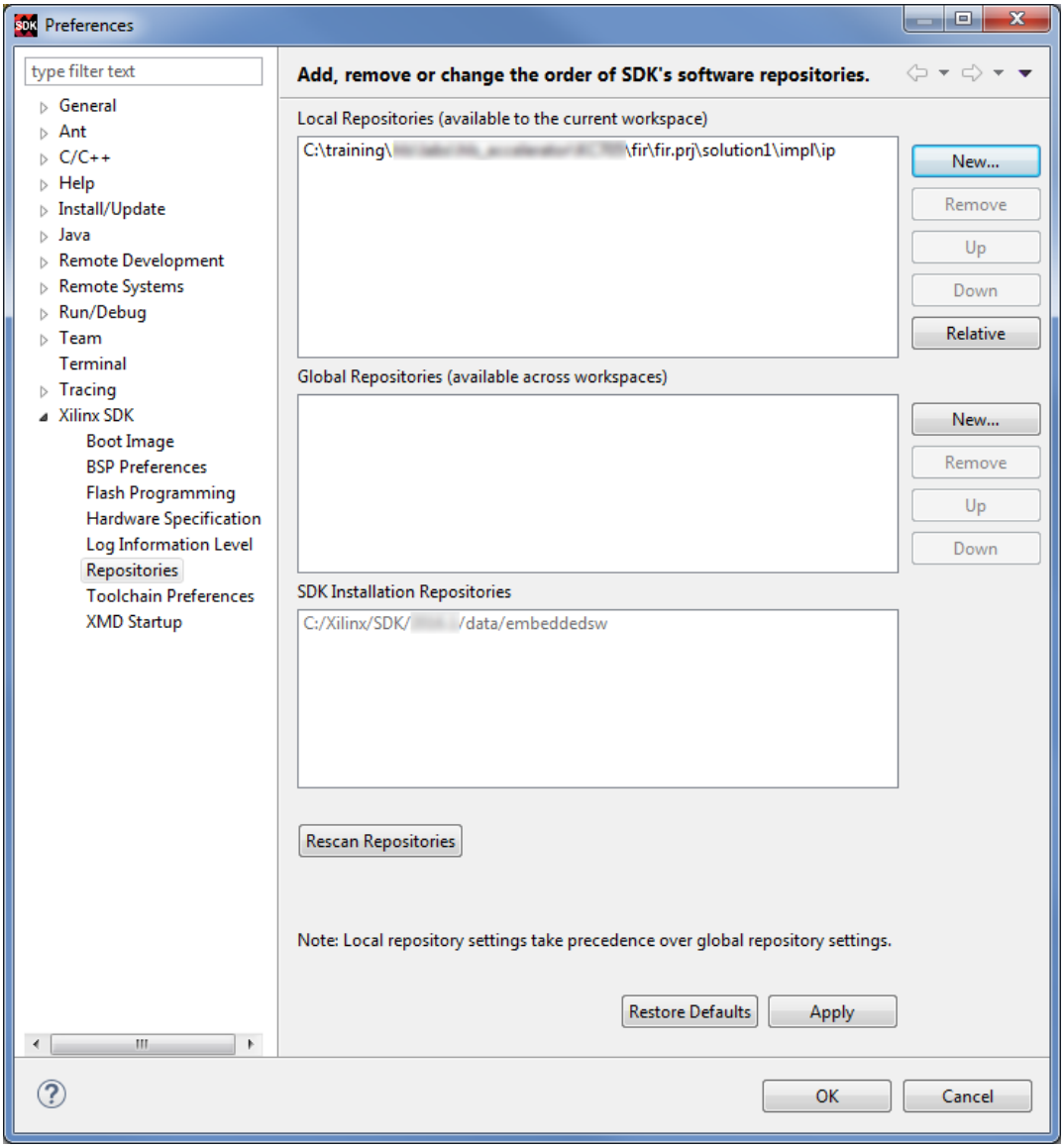


Figure 7-57: Adding HLS IP Drivers to the SDK Repository

6-1-4. Click **OK**.

6-2. Create an application in SDK.

6-2-1. Select **File** > **New** > **Application Project**.

6-2-2. Enter **fir_test_application** in the Project name field.

6-2-3. Use the default name for the Board Support Package name, **fir_test_application_bsp**.

6-2-4. Click **Next**.

6-2-5. Select **Empty Application** from the Available Templates section.

6-2-6. Click **Finish**.

Wait until the building workspace process completes.

6-3. Import the provided test application source into the SDK application project.

6-3-1. In the Project Explorer tab, expand **fir_test_application** (1).

6-3-2. Right-click the **src** folder and select **Import** (2).

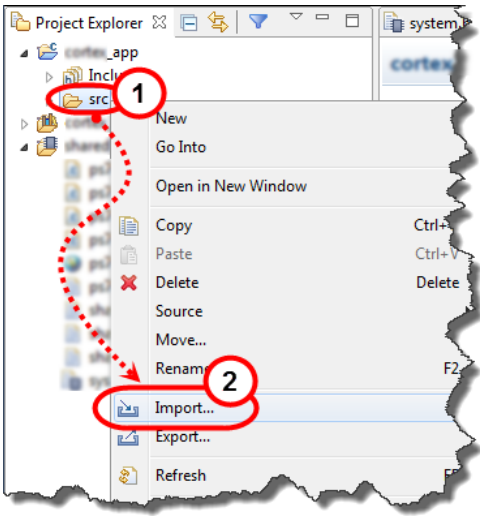


Figure 7-58: Importing a Resource File

6-3-3. Expand **General** and select **File System**.

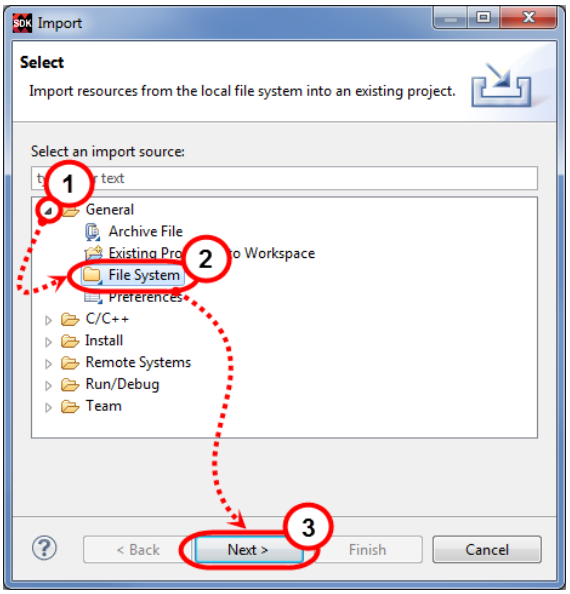


Figure 7-59: Selecting File System

6-3-4. Click **Next**.

6-3-5. In the From directory field, click **Browse**.

6-3-6. Browse to the *C:\training\hls\support* directory.

6-3-7. Click **OK**.

6-3-8. Select the **fir_test_application.c** file from the support folder.

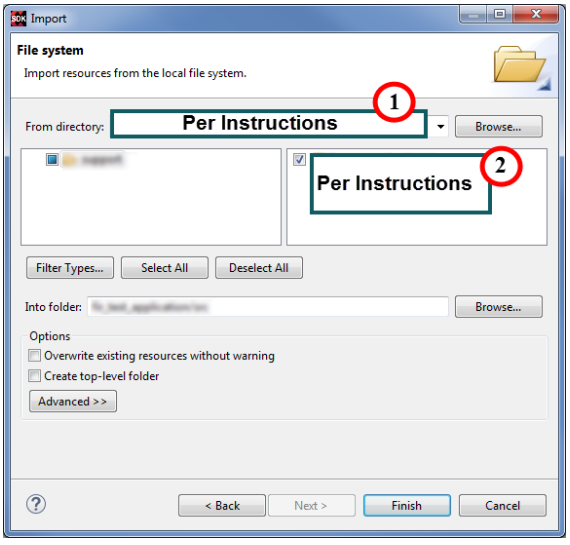


Figure 7-60: Selecting the Application Source File(s)

6-3-9. Click **Finish**.

After importing *fir_test_application.c*, SDK will automatically build the project and generate the corresponding ELF (Executable and Load Format) file.

6-3-10. In the Project Explorer tab, expand **fir_test_application** > **src**.

6-3-11. Double-click **fir_test_application.c** to review the provided test program.

The test application will apply an impulse sequence, read the transfer function of the filter from the output, and compare against the reference and display whether the test has passed or failed.

Verifying the Design in Hardware Step 7

You will now configure the FPGA, download the application, and test the program for proper operation.

Set up and connect the hardware evaluation board, or verify that this has properly been done before turning on the power.

7-1. Connect the board to your machine as shown below.

7-1-1. Connect a USB cable from a USB port on your computer to the USB UART connector on the evaluation board.

7-1-2. Similarly, connect the USB cable to the Digilent USB JTAG interface.

7-1-3. Ensure that the power cord is plugged in and turn on the evaluation board.

7-1-4. Make sure that the board settings are proper.

7-1-5. Ensure that all DIP switches (SW11) are in the off position.

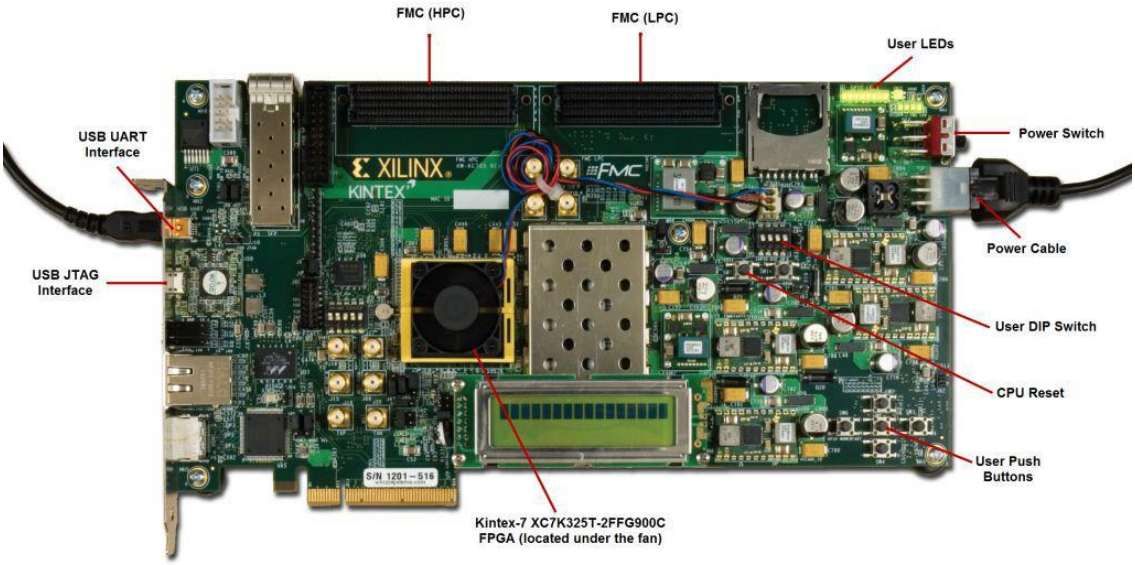


Figure 7-61: KC705 Evaluation Board

You may be prompted to install drivers when the board is first connected. Do not allow the driver installation to search the Web, but allow it to search for the drivers on your computer.

7-2. Determine which COM to use to access the USB serial port on the KC705.

7-2-1. Make sure that the development board is powered on and the serial UART device USB cable is in place.

This ensures that the USB-to-serial bridge will be enumerated by the PC host.

7-2-2. Select **Start** > **Devices and Printers**.

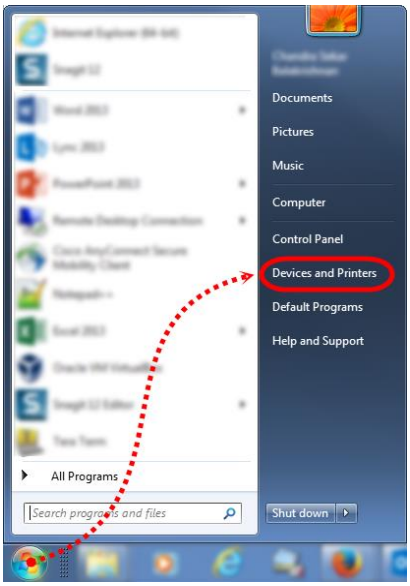


Figure 7-62: Opening the Device and Printers Panel

7-2-3. Locate **Silicon Labs CP210x USB to UART Bridge (COM#)**.

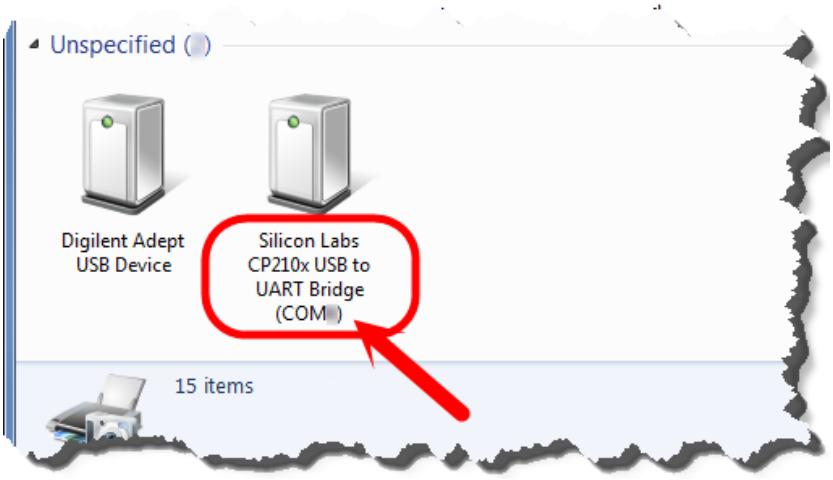


Figure 7-63: Locating the Silicon Labs Com Port Driver

The **#** indicates the port number for this serial connection.

7-2-4. Close the **Devices and Printers** window by clicking the red 'X' in the upper right of the window.

7-3. Set up the COM port in the SDK terminal settings.

7-3-1. In the SDK Terminal tab, click the **Connect to Serial Port** icon to configure the serial port settings as shown in the following figure.

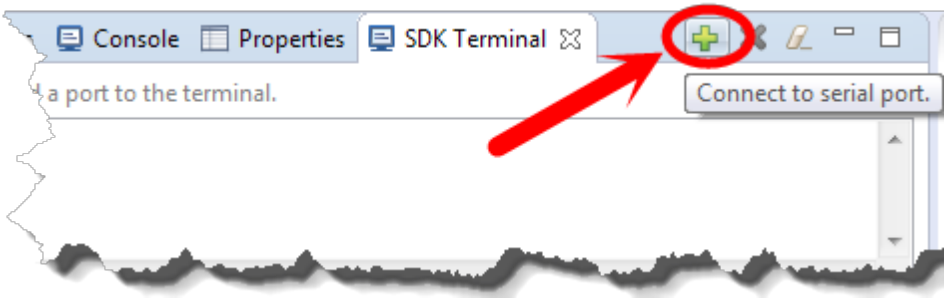


Figure 7-64: Connecting to the Serial Port

7-3-2. In the **Connect to serial port** window, configure the settings as shown in the following figure.

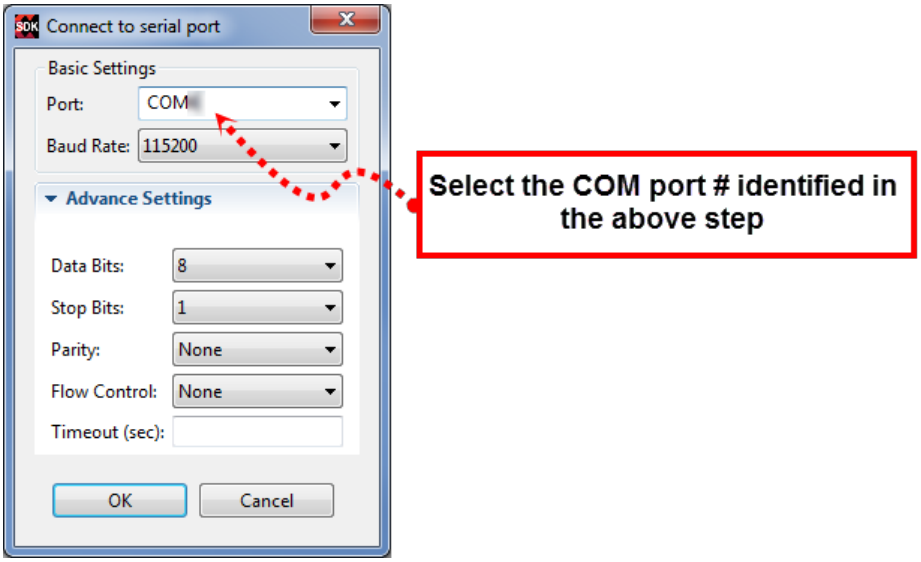


Figure 7-65: Serial Port Settings

7-3-3. Click **OK**.

7-4. Program the hardware and run the software application.

7-4-1. Select **Xilinx Tools** > **Program FPGA**.

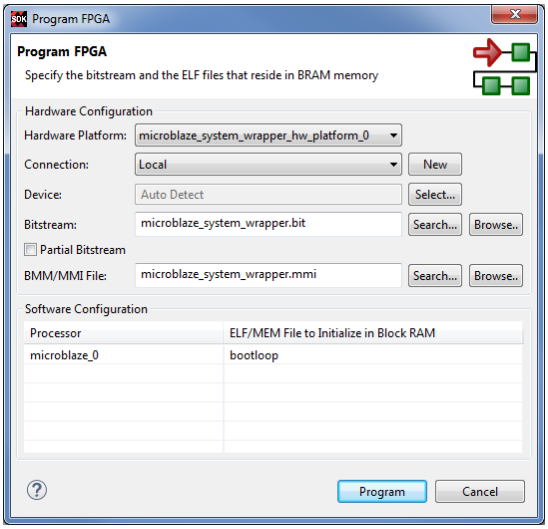


Figure 7-66: Programming the Device

7-4-2. Click **Program**.

7-4-3. In the Project Explorer tab, right-click **fir_test_application** and select **Run as** > **Run Configurations**.

7-4-4. Double-click **Xilinx C/C++ application (System Debugger)**.

7-4-5. Ensure that **fir_test_application Debug** is selected as shown below (1).

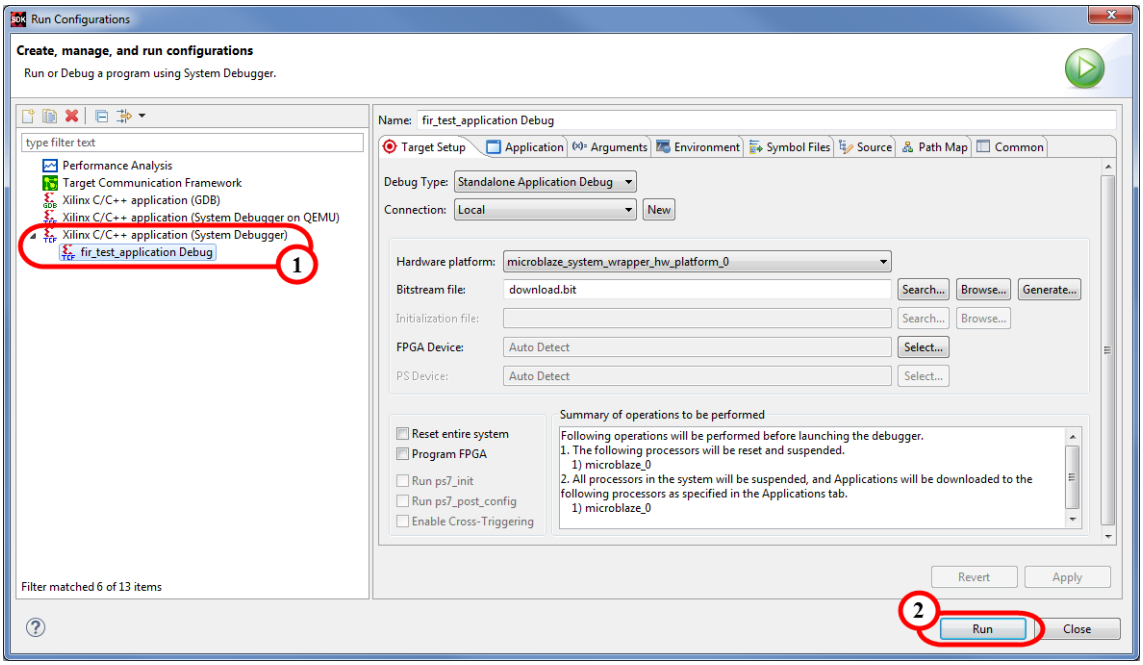


Figure 7-67: Verifying the Run Configuration

7-4-6. Click **Run** to run the SDK application on the hardware (2).

7-4-7. Select the **SDK Terminal** tab to view the result.

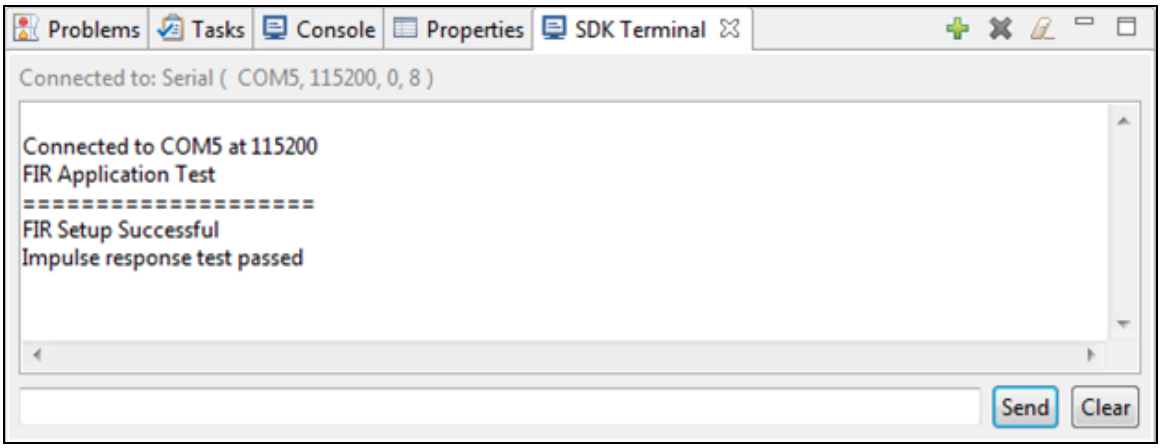


Figure 7-68: Viewing in Terminal after Downloading the Program

7-4-8. Switch off the board.

7-4-9. Close the SDK and Vivado tools.

Summary

In this lab, you developed an IP from the C design of a FIR filter using the Vivado HLS tool. Using the IP integrator, you created a system with the MicroBlaze processor and the IP. You then used a test application to write into and read from the IP interface registers.

Answers

Since there were no questions in this lab, this section is intentionally left blank.