Vivado Design Suite Tutorial

High-Level Synthesis

UG871 (v2012.4) January 25, 2013

XILINX®

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
01/25/13	3.0	Updated Xilinx release of the Vivado Design Suite Tutorial: High-Level Synthesis. Includes 2012.4 figure and minor general content updates.
11/16/12	2.0	Updated Xilinx release of the Vivado Design Suite Tutorial: High-Level Synthesis. Includes new Chapter 4, Using AXI4 Interfaces.
8/20/12	1.0	Initial Xilinx release of the Vivado Design Suite Tutorial: High-Level Synthesis.

Table of Contents

Chapter 1: Vivado HLS: Introduction Tutorial

Introduction 4
Licensing and Installation 4
Overview. 5
Starting Your Project 6
C Validation 13
Synthesizing and Analyzing the Design 20
Bit-Accurate Design 32
Design Optimization 40
RTL Verification and Export. 52
The Shell and Scripts 57

Chapter 2: Vivado HLS: Integrating EDK

Introduction 60
Reference Design 61

Chapter 3: Vivado HLS: Integrating System Generator

Introduction 93
Software Application for Vivado HLS 93
Create a Project in Vivado HLS for the FIR Application 96
Create an RTL design 103
Import the Design into System Generator 108

Appendix A: Additional Resources

Xilinx Resources 113
Solution Centers. 113
References 113

XILINX®

Chapter 1

Vivado HLS: Introduction Tutorial

Introduction

This guide provides an introduction to the Xilinx® Vivado High-Level Synthesis (HLS) tool for transforming a C, C++, or SystemC design specification into a Register Transfer Level (RTL) implementation, which can be synthesized into a Xilinx FPGA.

IMPORTANT: *All the tutorial examples for Vivado HLS can be downloaded from the same location on www.xilinx.com as this tutorial guide. Please refer to* http://www.xilinx.com/cgi-bin/docs/rdoc?v=2012.4;t=vivado+tutorials

This document is designed to be used with the FIR design example included with this tutorial.

This tutorial explains how to perform the following tasks using the Vivado HLS tool:

- Create an Vivado HLS project
- Validate the C design
- Perform synthesis and design analysis
- Create and synthesize a bit-accurate design
- Perform design optimization
- Understand how to perform RTL verification and export
- Review using the Vivado HLS tool with Tcl scripts

Licensing and Installation

The first steps in using the Vivado HLS tool are to install the software, obtain a license and configure it. See the *Xilinx Design Tools: Installation and Licensing Guide (UG978)*.

Contact your local Xilinx representative to obtain a license for the Vivado HLS tool.

Overview

This document uses a FIR design example to explain how the Vivado HLS tool is used to synthesize a C design to RTL that meets specific hardware design goals.

Design Goals

The hardware design goals for this FIR design project are to:

- Create a version of the design with the smallest area
- Create a version of this design with the highest throughput

The final design should be able to process 8-bit data supplied with an input valid signal and produce 8-bit output data accompanied by an output valid signal. The filter coefficients are to be stored externally to the FIR design, in a single port RAM.

Tutorial Setup

Begin by copying the `fir` directory to a local work area.

Note: PC users: The path name to the local work area should not contain any spaces. For example, `C:\Documents and Settings\My Name\Examples\fir` is not a valid work area because of the spaces in the path name.

Table 1-1: **Lab 1 File Summary**

Filename	Description
`fir.c`	C code to be synthesized into RTL.
`fir_test.c`	C test bench for the FIR design. It is used to validate that the C algorithm is functioning correctly and is reused by the Vivado HLS tool to verify the RTL.
`fir.h`	Header file for the filter and test bench.
`in.dat`	Input data file used by the test bench.
`out.gold.dat` `out.gold.8.dat`	Data that is expected from the FIR function after normal operation.

Learning Goals

This design example describes how to:

- Use the Vivado HLS Graphical User Interface (GUI) to create an Vivado HLS design project.
- Validate the C code within the Vivado HLS tool.
- Analyze the results of synthesis, understand the Vivado HLS reports, and be able to use the Design Viewer analysis capability.
- Apply optimizations to improve the design.
- Verify that the functionality of the RTL implementation matches that of the original C design.
- Export the design as an IP block to other Xilinx tools.

Optionally run logic synthesis during the RTL Export process to evaluate the timing and area results after logic synthesis.

Starting Your Project

The Vivado HLS Graphical User Interface (GUI) is used to perform all operations in this design tutorial. The Tcl based interactive and batch modes are discussed at the end of the tutorial.

Opening the Vivado HLS GUI

To open the GUI, double-click on the **Vivado HLS GUI** desktop icon.

Note: You can also open the GUI using the Windows menu by selecting **Start** > **All Programs** > **Xilinx Design Tools** > **Vivado 2012.4** > **Vivado HLS GUI**. The Vivado HLS group is shown in Figure 1-1.

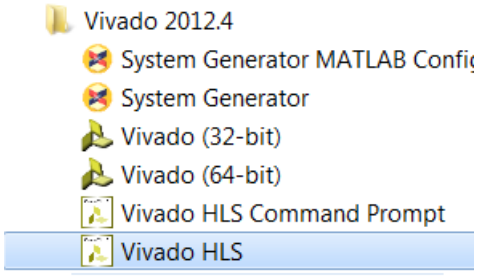


Figure 1-1: **Launching the Vivado HLS GUI**

Vivado HLS opens. The Welcome Page shows the primary starting points for Vivado HLS.

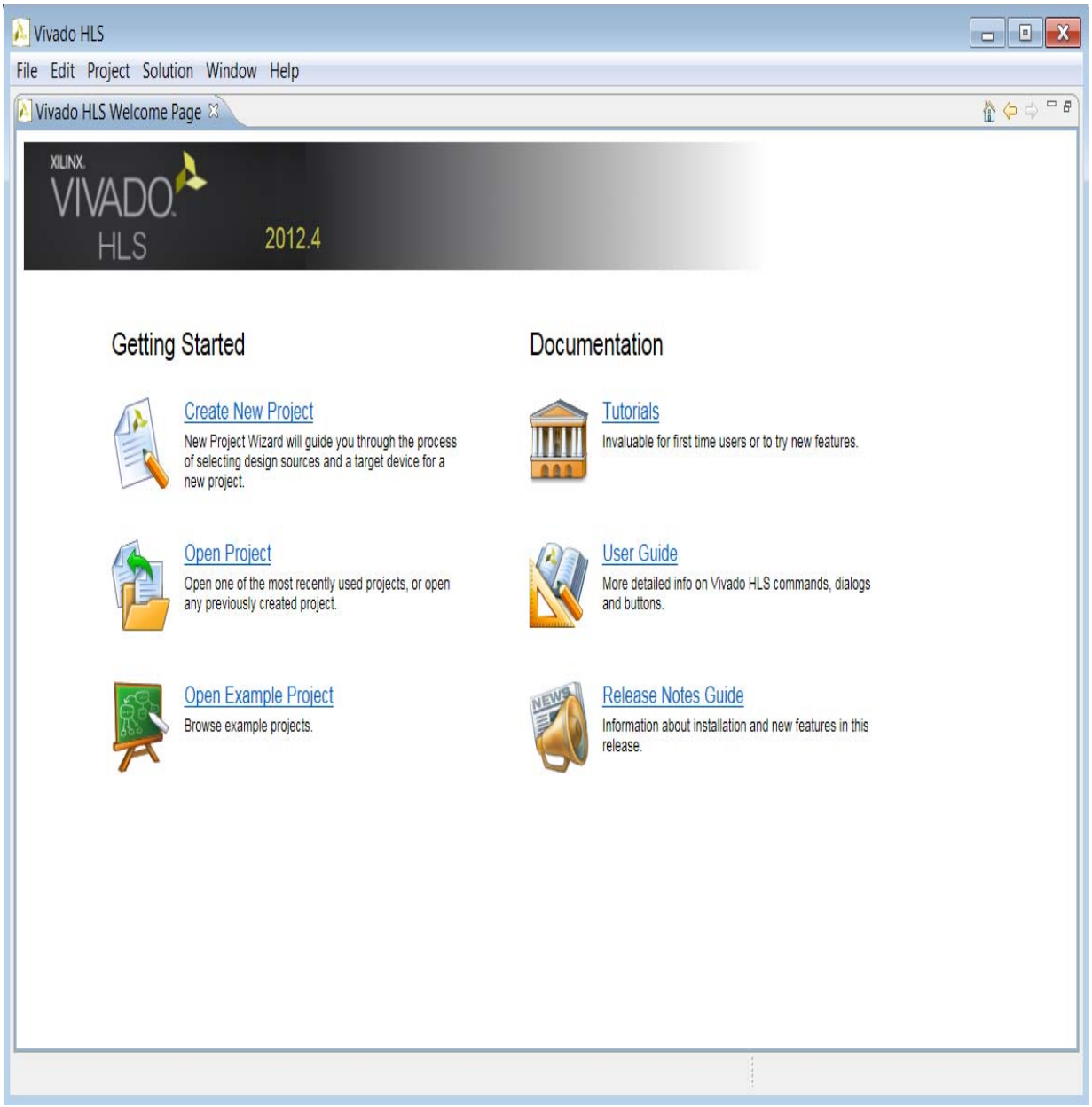


Figure 1-2: **Vivado HLS Welcome Page**

The Getting Started options are:

- **Create New Project**: Launches the project setup wizard.
- **Open Project**: Displays a list of recent projects and provides a link to navigate to an existing project.
- **Open Example Project**: Open Vivado HLS examples. These can also be found in the examples directory in the Vivado HLS installation area.

The Documentation options are:

- **Tutorials**: Opens the Vivado HLS Tutorials.
- **User Guide**: Opens the Vivado HLS User Guide.
- **Release Notes Guide**: Opens the Release Notes for this version of software.

Creating a New Project

1. In the Welcome Page, select **Create New Project** to open the Project Wizard, shown in Figure 1-3.

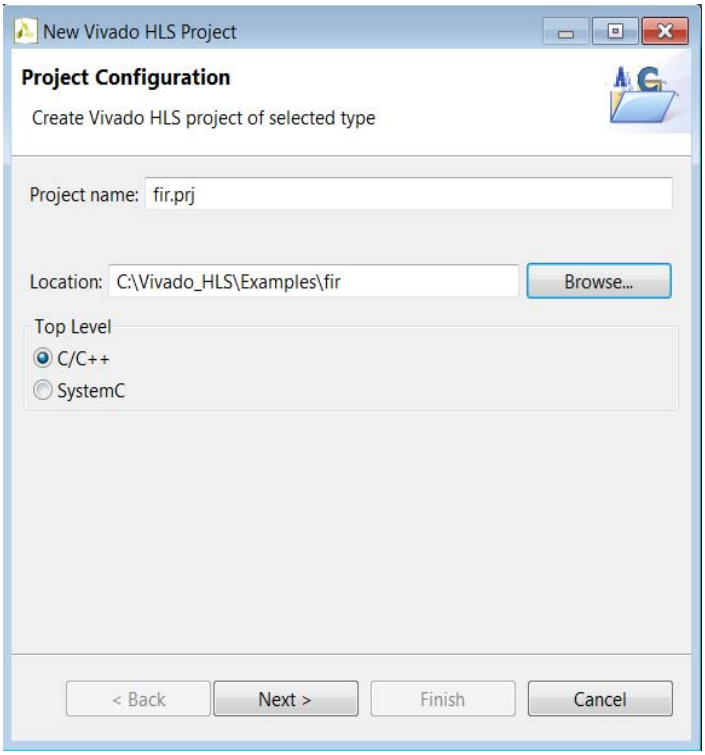


Figure 1-3: **Project Specification**

2. Type the project name, **`fir.prj`**.
3. Click **Browse** to navigate to the location of the **`fir`** directory.
4. Select the `fir` directory and click **OK**.
5. Specify the top-level as **C/C++**.

   Note: A SystemC project is only required when the top-level is a SystemC SC_MODULE.
6. Click **Next**.

   The next window prompts you for information on the design files (see Figure 1-4).

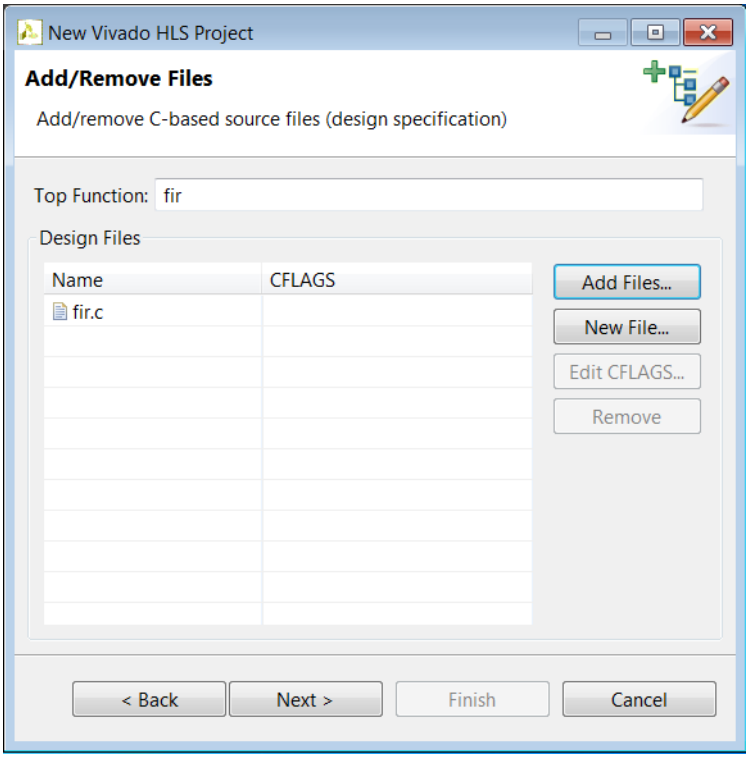


Figure 1-4: **Project Design Files**

7. Specify the top-level function (`fir`) to be synthesized.
8. Click **Add Files**.
9. Specify the C design files. In this case there is only one file, `fir.c`.
10. Click **Next**.

Figure 1-5 shows the window for specifying the test bench files. The test bench and all files used by the test bench, except header files, must be included. You can add files one at a time, or select multiple files to add using the **ctrl** and **shift** keys.

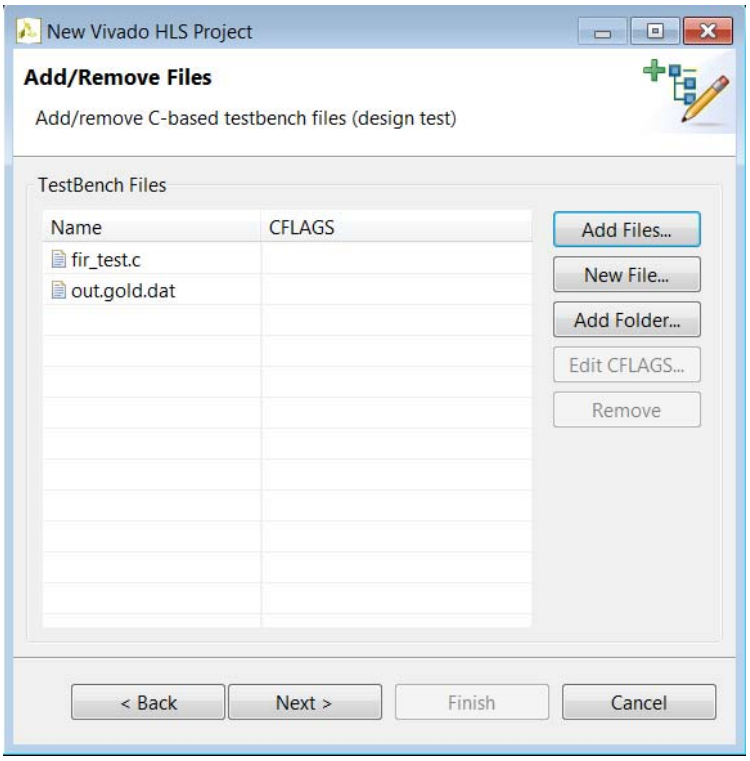


Figure 1-5: **Test Bench Files**

11. Use the **Add Files** button to include both test bench files: `fir_test.c` and `out.gold.dat`.
12. Click **Next**.

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

The Solution Configuration window (shown in Figure 1-6) allows the technical specifications of the solution to be defined. A project can have multiple solutions, each using a different target technology, package, constraints, and/or synthesis directives.

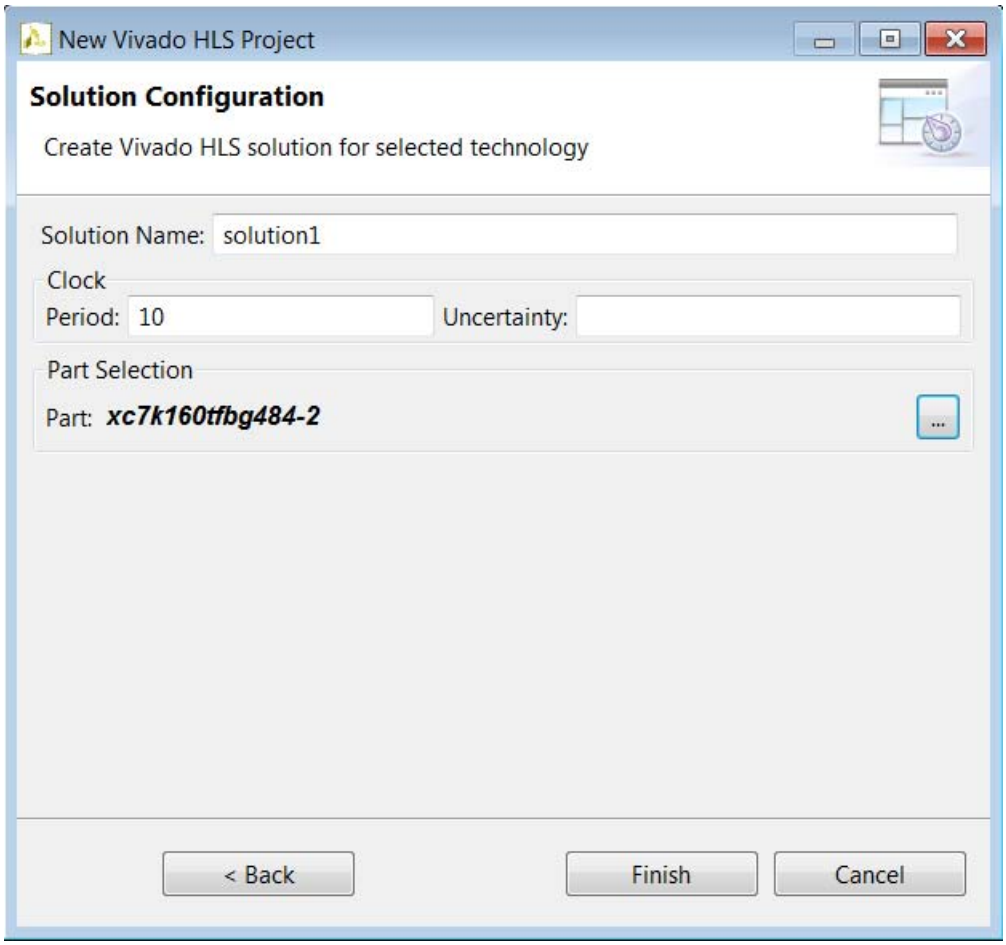


Figure 1-6: **FIR Solution**

13. Accept the default solution name (`solution1`), clock period (`10ns`) and clock uncertainty (defaults to 12.5% of the clock period, when left blank/undefined).
14. Click the part selection button [...] to open the part selection window and make the following selections in the drop-down filters:
    - Product Category: General Purpose
    - Family: Kintex®-7
    - Sub-Family: Kintex-7
    - Package: fbg484
    - Speed Grade: -2
    - Temp Grade: Any
15. Select **Device xc7k160tfbg484-2** from the list of available devices.
16. Click **OK** to see the selection made, as shown in Figure 1-6.

The Vivado HLS GUI opens with the project information included, as shown in Figure 1-7.

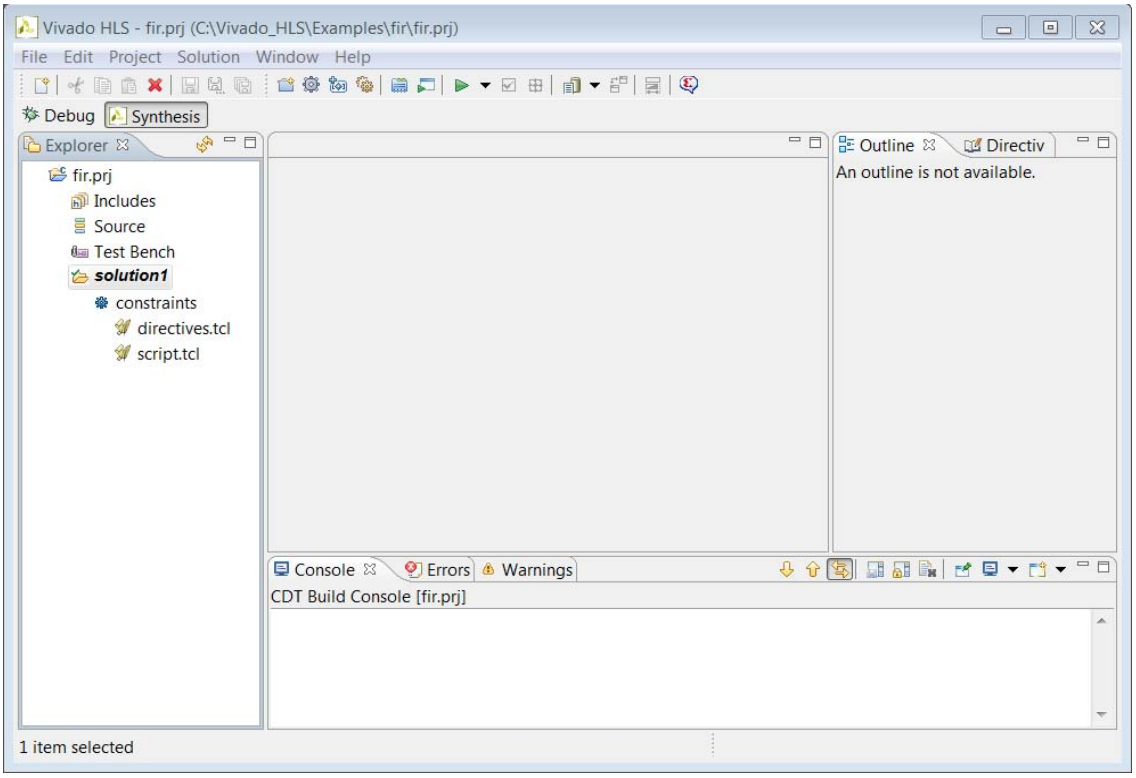


Figure 1-7: **Project GUI**

Note: You can see the project name on the top line of the Project Explorer pane.

An Vivado HLS project arranges data in a hierarchical form.

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

Note: It is always possible to access and change project or solution settings by clicking on the corresponding button in the toolbar, as shown Figure 1-8 and Figure 1-9.

Figure 1-8: **Project Settings**

Figure 1-9: **Solution Settings**

Summary

- You can use the Project wizard to set up an Vivado HLS project.
- Each project is based on the same source code and test bench.
- A project can contain multiple solutions and each solution can use a different clock rate, target technology, package, speed grade, and more typically, different optimization directives.

C Validation

You must validate the C design prior to synthesis to ensure that it is performing correctly. You can perform this validation using the Vivado HLS tool.

Test Bench

The test bench file, `fir_test.c`, contains the top-level C function `main()`, which in turn calls the function to be synthesized (`fir`). A useful characteristic of this test bench is that it is self-checking and returns a value of `0` (zero) to confirm that the results are correct. Some other characteristics of this test bench are:

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

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

If the test bench has the self-checking characteristics mentioned above, the RTL results are automatically checked against the golden data. There is no requirement to create RTL in a test bench. This provides a robust and productive verification methodology.

Types of C Compilation

The Vivado HLS tool provides a number of options for C compilation.

- By default, if no option is selected, the code is compiled in debug mode and executed.
- If the Debug option is selected, the code is compiled and the Vivado HLS debug environment is automatically invoked (this is shown later in this tutorial). This option is option cannot be used with the Optimizing Compile option.
- If option Build Only is selected, the code is compiled but the code is not executed.
- The Clean Build option removes any existing pre-compiled code before starting.
- If the Optimizing Compile option is used the code is compiled without any debug information. This option cannot be used with the Debug option. If this option is used, the Debug environment cannot be invoked.

This tutorial demonstrates the default and debug options.

C Validation

You can perform C simulation to validate the C algorithm by compiling the C function/design and executing it. This first example also opens the compiled C code in the Vivado HLS debug environment.

Figure 1-10 shows the `Run C Simulation` button on the toolbar.

Figure 1-10: **Run C Simulation Toolbar Button**

1. Click the **Run C Simulation** button, shown in Figure 1-10, to compile and run the design.

   This opens the C simulation dialog box, as shown in Figure 1-11.

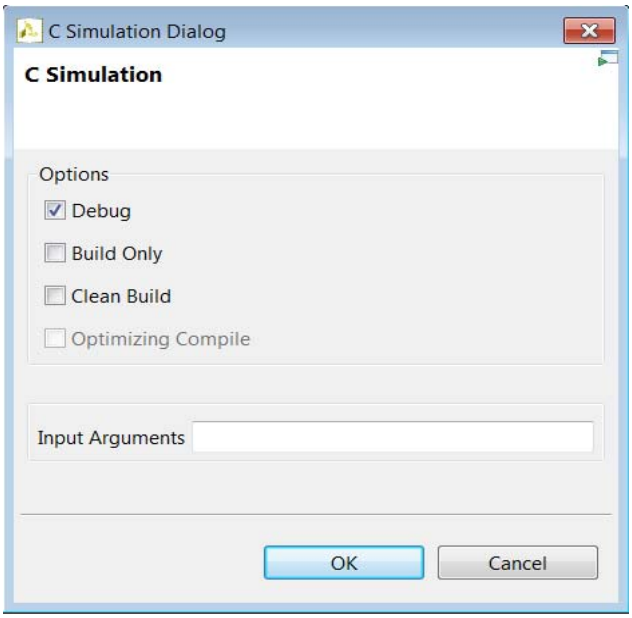


Figure 1-11: **C Simulation Dialog Box**

2. Select the **Debug** option (shown in Figure 1-10) and click **OK** to compile the code and open the debug environment.

   The debugger opens (see Figure 1-12).

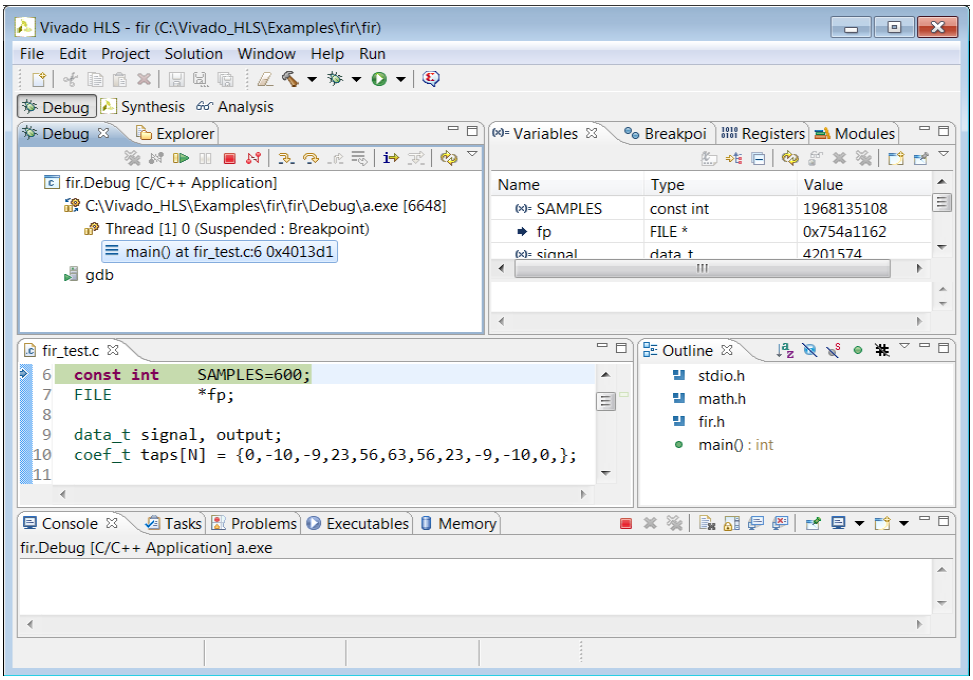


Figure 1-12: **Debugger Window**

The following steps describe using the debugger.

3. Step through the code by clicking the **Step Into** toolbar button, as shown in Figure 1-13.

Figure 1-13: **Step Into Button**

4. Continue stepping through the code until the debugger moves into the FIR code by clicking **Step Into** approximately nine times.

   The code window looks like Figure 1-14.

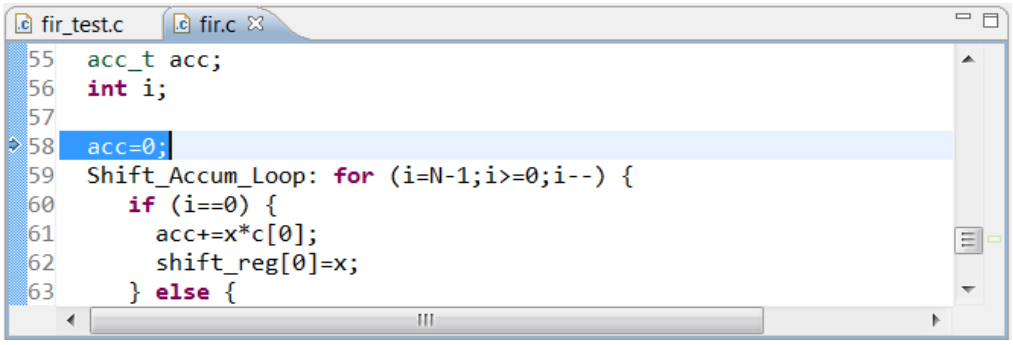


Figure 1-14: **Debug in the FIR Design**

5. To add a breakpoint, in the left-hand margin of the fir.c tab, double-click on line **58**. A breakpoint indication mark appears to the left of the line number, as shown in Figure 1-15.

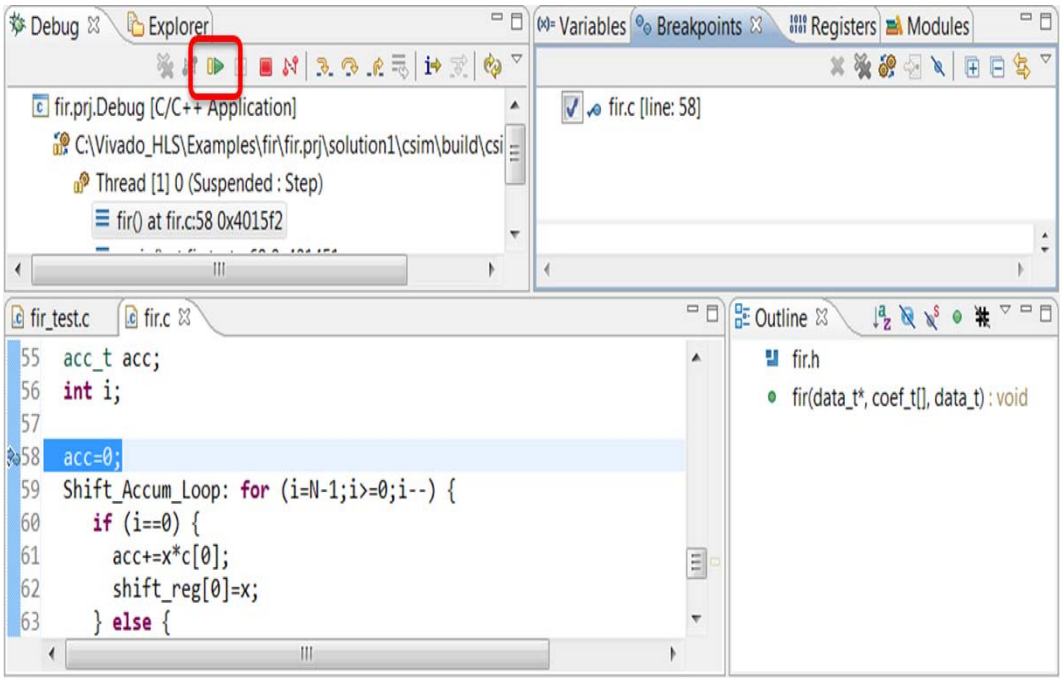


Figure 1-15: **Adding a Breakpoint**

6. To confirm the breakpoint has been added, open the Breakpoints tab, shown in Figure 1-15.
7. Open the Variables tab, shown in Figure 1-16.

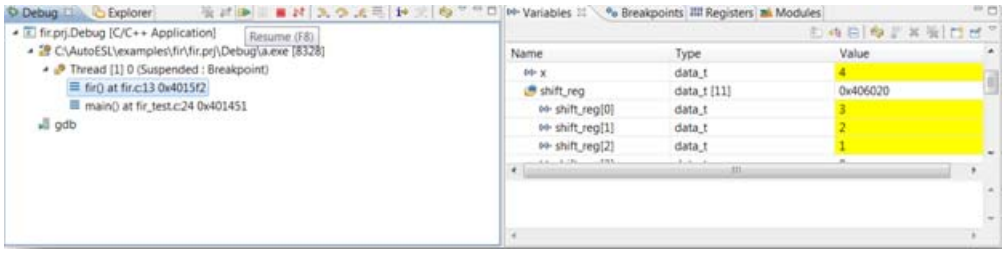


Figure 1-16: **Review the Operation of the C Code**

8. Click the **Resume** button, highlighted in Figure 1-17, to run the code until the next breakpoint.

   The debugger stops each time it reaches line 58.

9. Adjust the Variables window to view the `shift_reg` variable. This updates the shift register.
10. Click the **Resume** button multiple times.
11. Click the **Terminate** button, shown in Figure 1-17, to end the debug session.

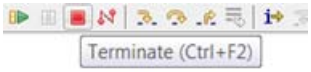


Figure 1-17: **Terminate Button**

12. Click **Synthesis** to return to the Synthesis perspective as shown in Figure 1-18.

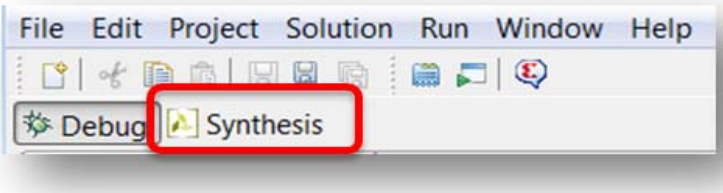


Figure 1-18: **Synthesis Perspective**

13. Click the **Run C Simulation** button and un-select the debug option, shown in Figure 1-19, to compile and run the C design.

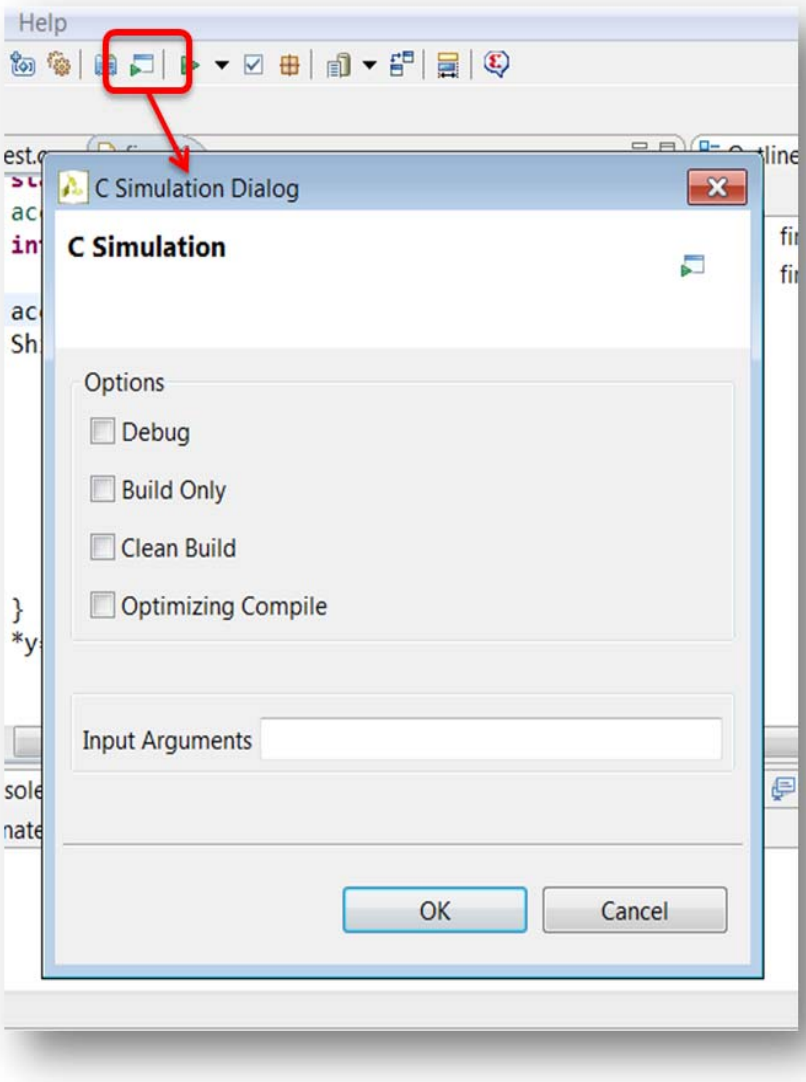


Figure 1-19: **Run C/C++ Project Button**

The results are shown in the console window (see Figure 1-20), indicating that the `fir` function is producing good data and operating correctly. This example assumes that the golden data in the `out.gold.dat` file has already been verified as correct.

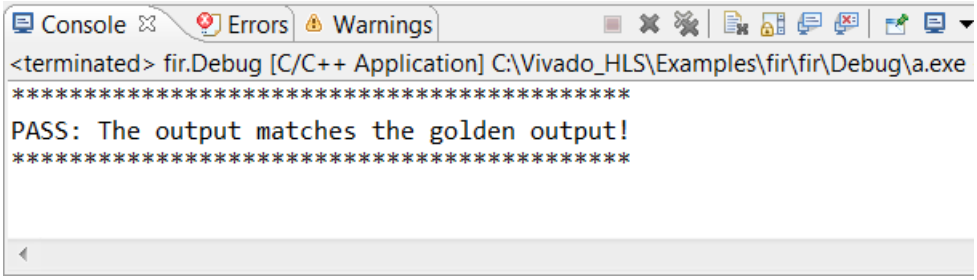


Figure 1-20: **C Validation Results**

Summary

- Validate the C code before high-level synthesis to ensure that it has the correct operation.
- You can enhance overall productivity using a test bench, which can self-check the results.
- You can use the C development environment in the Vivado HLS tool to validate and debug the C design prior to synthesis.

Synthesizing and Analyzing the Design

After C validation, there are three major steps in the Vivado HLS design flow:

- Synthesis: Create an RTL implementation from the C source code.
- Co-simulation: Verify the RTL through co-simulation with the C test bench.
- Export RTL: Export the RTL as an IP block for use with other Xilinx tools.

You can run each of these steps from the toolbar as shown in Figure 1-21. Because Synthesis is the first step in this process, the **Synthesis** button is located on the left side.

Figure 1-21: **Design Steps**

The **Simulation** and **Implementation** buttons are located to the right of the **Synthesis** button. Both simulation and implementation require that synthesis completes before they can be performed and so are currently grayed out in Figure 1-21.

Synthesis

Your design is now ready for synthesis. Click the **Synthesis** button, as shown in Figure 1-21.

When synthesis completes, the GUI updates with the results, as shown in Figure 1-22.

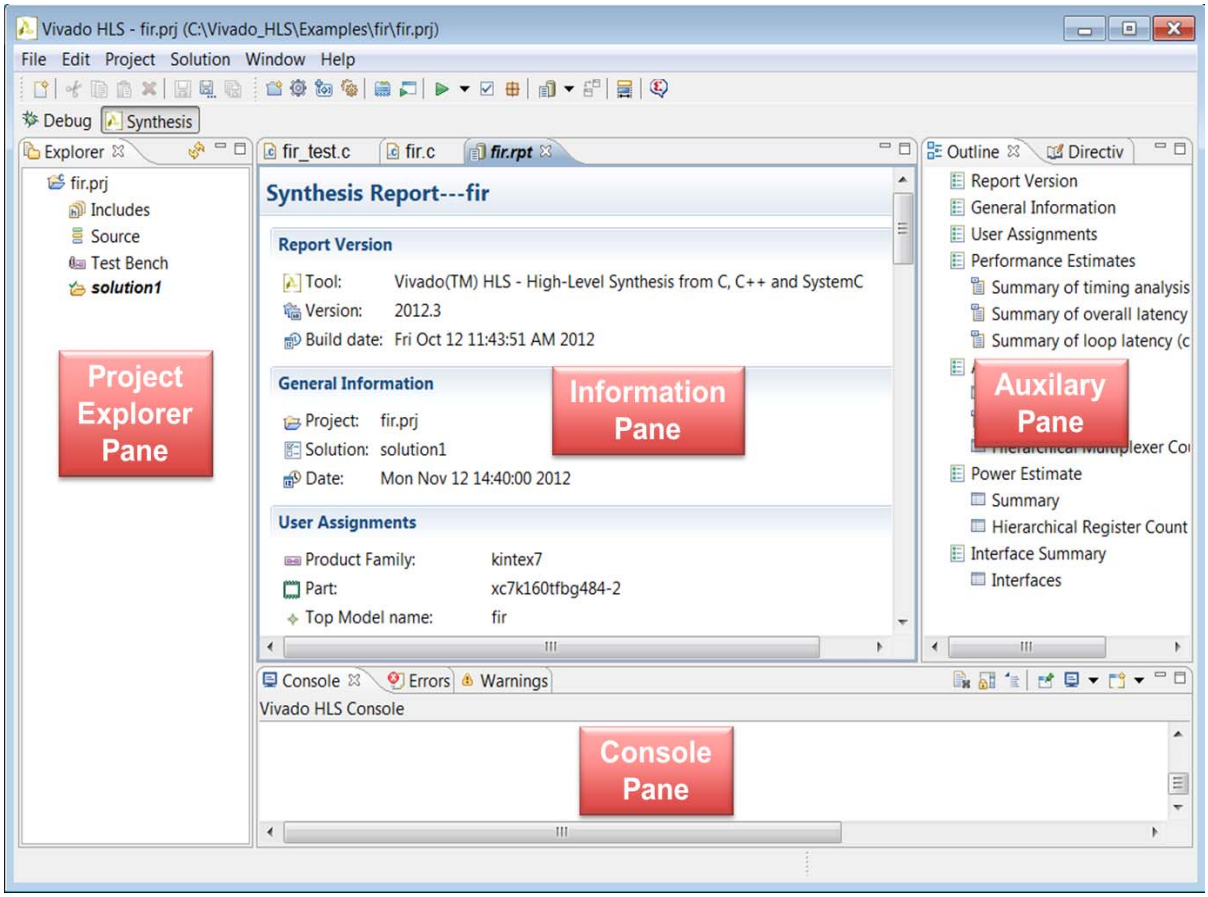


Figure 1-22: **GUI Overview**

Now all the window panes in the GUI are populated with data. The panes are:

- **Project Explorer**: This pane now shows a `syn` container inside `solution1`, indicating that the project has synthesis results. Expand the `syn` container to view containers `report`, `systemc`, `verilog` and `vhdl`.

  The structure in the `solution1` container is reflected in the directory structure inside the project directory. Directory `fir.prj` now contains directory `syn`, which in turn contains directories `report`, `systemc`, `verilog` and `vhdl`.

- **Console**: This pane shows the messages produced during synthesis. Errors and warnings are shown in tabs in the Console pane.

- **Information**: A report on the results automatically opens in the Information pane when synthesis completes. The Information pane also shows the contents of any files opened from the Project Explorer pane.
- **Auxiliary**: This pane is cross-linked with the Information pane. Because the information pane currently shows the synthesis report, the Auxiliary pane shows an outline of this report.

TIP: Click on the items in the Report Outline in the Auxiliary pane to automatically scroll the Information pane to that point of the report.

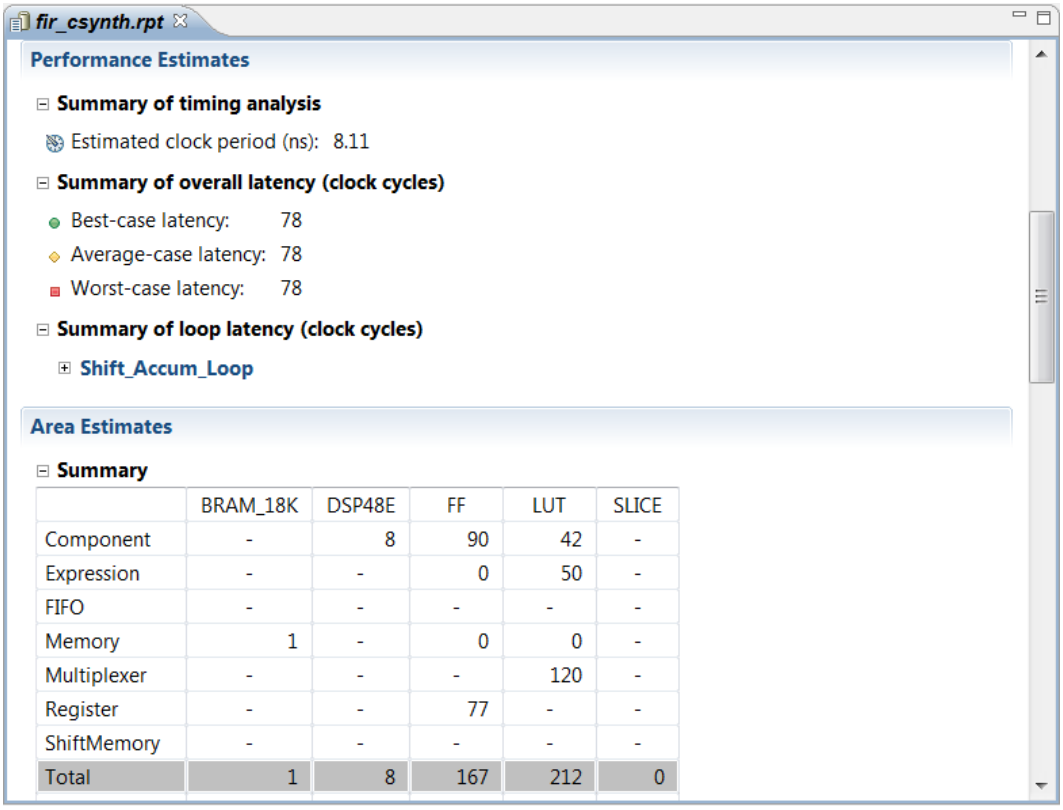
Table 1-2: **Synthesis Report Categories**

Category	Sub-Category	Description
Report Version	---	Details on the version of the Vivado HLS tool used to create the results.
General Information	---	Project name, solution name, and when the solution was ran.
User Assignments	---	Details on the technology, target device attributes, and the target clock period.
Performance Estimates	Summary of timing analysis	The estimate of the fastest achievable clock frequency. This is an estimate because logic synthesis and place and route are still to be performed.
	Summary of overall latency	The latency of the design is the number of clock cycles from the start of execution until the final output is written. If the latency of loops can vary, the best, average, and worse case latencies is different. If the design is pipelined, this section shows the throughput. Without pipelining the throughput is the same as the latency; the next input is read when the final output is written.
	Summary of loop latency	This shows the latency of individual loops in the design. The trip count is the number of iterations of the loop. The latency in this "loop latency" section is the latency to complete all iterations of the loop.

Table 1-2: **Synthesis Report Categories *(Cont'd)***

Category	Sub-Category	Description
Area Estimates	Summary	This shows the resources (such as LUTS, Flip-Flops, and DSP48s) used to implement the design. The sub-categories are explained in the Details section of this table.
	Details: Component	The resources specified here are used by the components (sub-blocks) within the top-level design. Components are created by sub-functions in the design. Unless inclined, each function becomes it's own level of hierarchy. In this example there are no sub-blocks, the design has one level of hierarchy.
	Details: Expression	This category shows the area used by any expressions such as multipliers, adders, and comparators at the current level of hierarchy.
	Details: FIFO	The resources listed here are those used in the implementation of FIFOs at this level of the hierarchy.
	Details: Memory	The resources listed here are those used in the implementation of memories at this level of the hierarchy.
	Details: Multiplexors	All the resources used to implement multiplexors at this level of hierarchy are shown here.
	Details: Registers	This category shows the register resources used at this level of hierarchy.
	ShiftMemory	A summary of all shift registers which were mapped into Xilinx SRL components.
	Hierarchical Multiplexor Count	A summary of the multiplexors throughput the hierarchy.
Power Estimate	Summary	The expected power used by the device. At this level of abstraction the power is an estimate and should be used for comparing the efficiently of different solutions.
	Hierarchical Register Count	The estimated power used by resisters throughput the design hierarchy.
Interface Summary	Interface	This section shows the details on type of interfaces used for the function and the ports, such as port names, directions, and bit-widths.

A section of the report is shown in Figure 1-23.



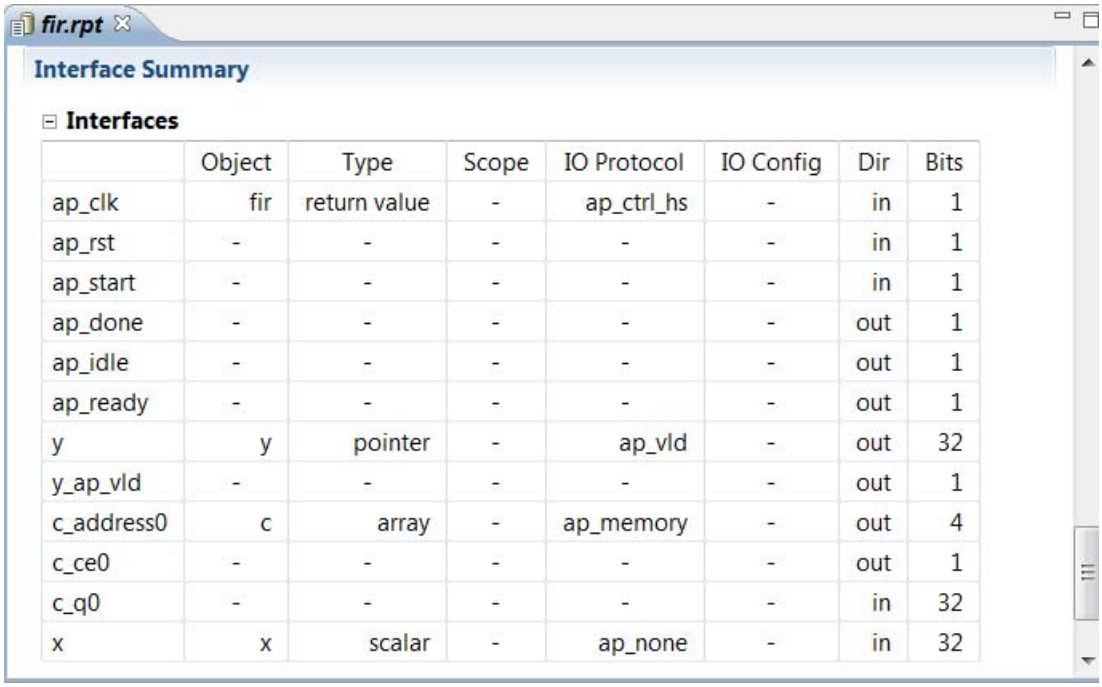
	BRAM_18K	DSP48E	FF	LUT	SLICE
Component	-	8	90	42	-
Expression	-	-	0	50	-
FIFO	-	-	-	-	-
Memory	1	-	0	0	-
Multiplexer	-	-	-	120	-
Register	-	-	77	-	-
ShiftMemory	-	-	-	-	-
Total	1	8	167	212	0

Figure 1-23: **solution1 Performance and Area Summary**

This report shows the initial solution to be:

- Meeting the clock frequency of 10ns
- Taking 78 clock cycles to output data
- Using eight DSP48 blocks
- Using one BRAM memory block.

To view details of the interface, select Interface Summary from the Report Outline in the Auxiliary pane, or scroll down the report in the Information pane. See Figure 1-24.


fir.rpt

Interface Summary

Interfaces

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

Figure 1-24: **solution1 IO Summary**

Note the following:

- A clock and reset port were added to the design.
- Block-level handshake ports were added.
  - By default, block-level handshakes are enabled. These are specified by IO mode `ap_ctrl_hs` and ensure that the RTL design can be automatically verified by the `autosim` feature.
  - This IO protocol ensures that the design does not start operation until input port `ap_start` is asserted (high) `ap_ready` goes high to indicate when new inputs can be applied, it indicates completion and its idle state by asserting `ap_done` and `ap_idle`, respectively.
- The A single-port RAM interface is used for coefficient port, `c`.
  - If no RAM resource is specified for arrays, Vivado HLS determines the most appropriate RAM interface (if a dual-port improves performance, it is used).
  - In this example, a single port interface is required; therefore, it should be explicitly specified.
- The data output port, `y`, is by default using an output valid signal (`y_ap_vld`). This satisfies the requirements on the output port.

- Data input port, x, has no associated handshake signal and requires a valid input.

Design Analysis: The Design Viewer

When synthesis has completed, you can use the Design Viewer to examine the design implementation in detail. You can invoke the Design Viewer can be invoked from the Vivado HLS toolbar (or from the Solutions menu).

To open the Design Viewer, click on the **Design Viewer** button, shown in Figure 1-25.

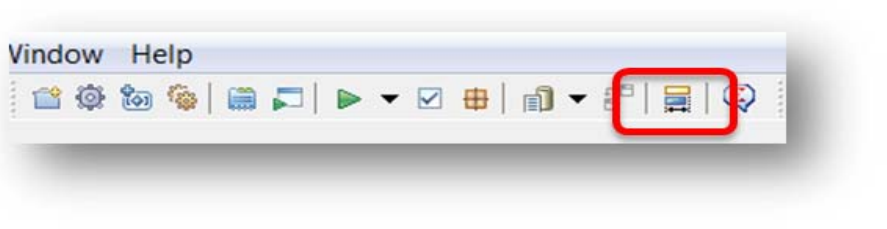


Figure 1-25: **Design Viewer Button**

The Design Viewer opens, as shown in Figure 1-26.

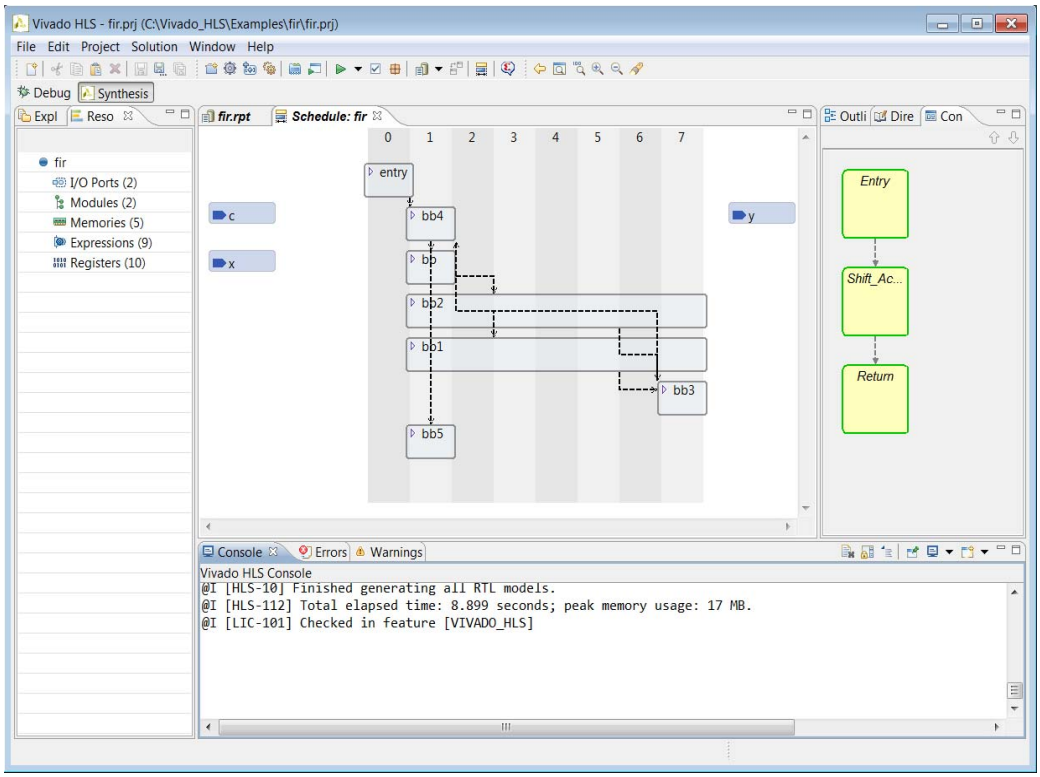


Figure 1-26: **Design Viewer**

The Design Viewer comprises three panes:

- **Control Flow Graph**: This pane is the closest to the software view and is the best place to begin analysis.
- **Schedule Viewer**: This pane shows how the operations are scheduled in internal control steps. These are mapped to clock cycles, and this view might not correlate exactly to clock cycles in all cases.
- **Resource Viewer**: This view shows how the operations in the Schedule Viewer are mapped to specific hardware resources.

In the Control Flow Graph pane, double-click the **Shift_Accum_Loop** block and navigate down into the details of the loop, as shown in Figure 1-27.

When the `Shift_Accum_Loop` is selected, the corresponding items in the Schedule Viewer are also selected.

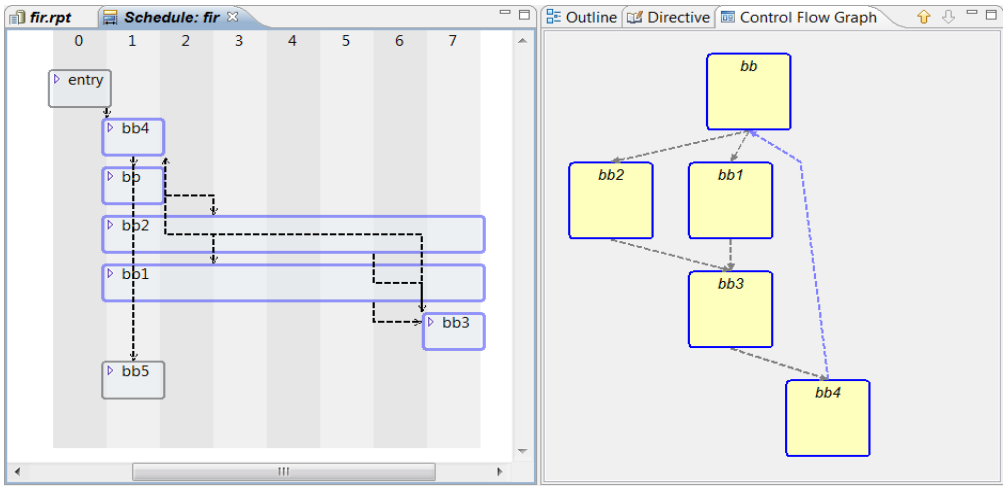


Figure 1-27: **Cross-Probing the CFG and Schedule Viewer**

Because the Control Flow Graph is closest to the software, this view shows how the design is operating. The flow is as follows:

a. The `Shift_Accum_Loop` loop starts in basic block `bb`.

b. The loop proceeds to either block `bb2` or block `bb1`.

c. Both blocks (`bb2` and `bb1`) return control to block `bb3`.

d. The loop ends in block `bb4`, which returns to block `bb`.

The following shows how you can use the Design Viewer to analyze the design:

1. In the Schedule Viewer, expand the first block, **`bb`**, by clicking on the **arrow** in the top-left corner (beside the name `bb`); see Figure 1-27.

2. Select the **icmp** operator and right-click to see the pop-up menu (see Figure 1-28).

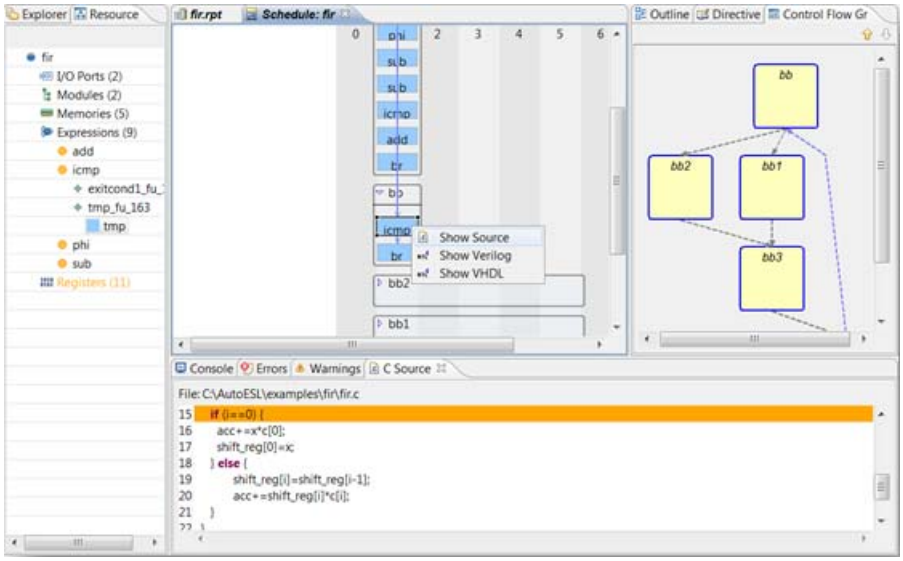


Figure 1-28: **Cross-Probing to the Source Code**

3. Select **Show Source** from the menu to view the source of this comparison operation in the C source code.

   The source code opens, highlighting the comparison operation implemented by this comparator. This indicates that the highlighted comparator (and block `bb`) implements the if-condition at the start of the loop.

   The flow is explained here in more detail, using the other blocks in the design (`bb1-4`) to give a more detailed understanding of how the code in the design is implemented, allowing the initial understanding of the code to be further developed:

   a. The loop starts in block `bb`.

      This is the if-condition at the start of the loop. Because it is a non-conditional for-loop, the loop must be started. The exit condition is checked at the end of the loop.

   b. The loop proceeds to either block `bb2` or block `bb1`.

      Block `bb2` is the else-branch inside the for-loop and performs two memory read (`load`) operations, a memory write (`store`) operation, and a multiplication (`mul`).

      - A load/read operation takes two cycles: one to generate the address and the other to read the data.
      - A complete list of operators is available in the *Vivado Design Suite User Guide: High-Level Synthesis (UG902) > High-Level Synthesis Operator and Core Guide chapter.*

Block `bb1` is the if-branch inside the for-loop and performs a single memory read, write and multiplication. Both blocks (`bb2` and `bb1`) return control to block `bb3`.

This block performs the accumulation common to both branches of the if-else statement.

c. The loop ends in block `bb4`, which returns to block `bb`.

Block `bb4` is the loop-header, which checks the exit condition and increases the loop iteration count.

The Resource Viewer shows more details on the implementation and lists the hardware resources in the design using the following top-level categories:

- Ports
- Modules
- Memories
- Expressions
- Registers

The items under each category represent specific instances of this resource type in the design; for example, a RAM, multiplier, or adder.

The items under each resource instance show the number of unique operations in the C code implemented using this hardware resource. If multiple operations are shown on the same resource, the resource is being shared for multiple operations.

Figure 1-29 shows a more detailed view of how the Resource Viewer shows sharing (or lack of it, in this case).

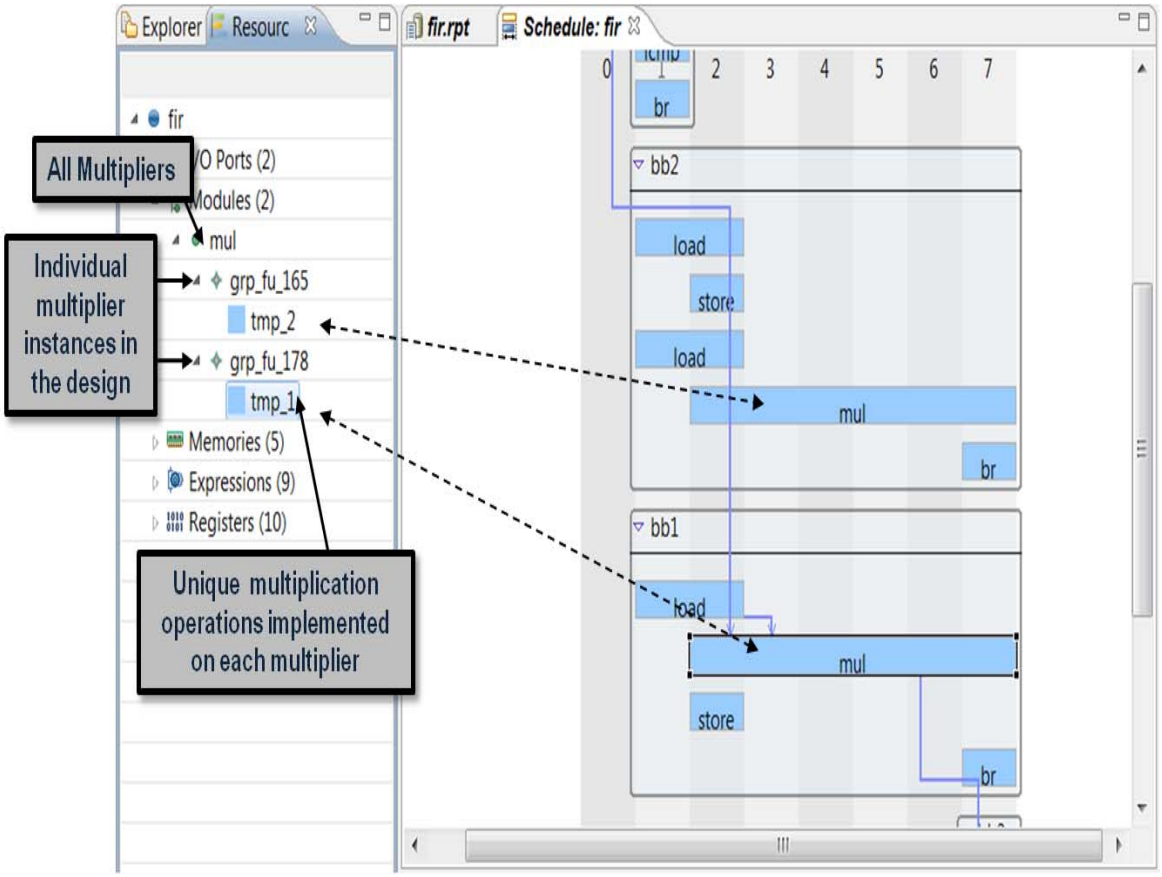


Figure 1-29: **View Sharing in the Design Viewer**

All the multipliers in this design are listed under `mul` in the Modules category. Each item in the `mul` category represents a physical multiplier in the design (the name given is the instance name of the multiplier in the RTL design).

In this example, there are two multipliers (`grp_fu_*`) in the design. You can do the following actions:

- Select a multiplier in the Schedule Viewer to highlight which multiplier instance is used to implement it. If a register is also highlighted, it indicates the output is registered.
- Expand each multiplier in the Resource Viewer to show how many unique multiplication operations in the code (shown as blue squares) are mapped onto each hardware resource.
- Click on the operations (blue squares) to show that the `mul` operation in block `bb1` is implemented on one multiplier and the `mul` operation in block `bb2` is implemented on a different multiplier.

In this example, both multiplier resources are being used to implement a single multiplication (`mul`) operation and there is no sharing of the multipliers.

By contrast, examining the memory operations (`load` and `store`) in the Schedule Viewer shows that multiple read (`load`) and write (`store`) operations are implemented on the same memory resource. This also shows that array `shift_reg` has been implemented as a memory.

Design Analysis Summary

Selecting the operations in the Schedule Viewer and correlating them with the associated elements in the Resource Viewer to show this design and the required optimizations/ changes can be summarized as follows:

- The implementation, like the C code, is iterating around loop `Shift_Accum_Loop` and using the same hardware resources for each iteration.
  - The main operation is six clock cycles through blocks `bb`, `bb1/b2`, `bb3`, etc. repeated 11 times.
  - This keeps the resource count low, because the same resources are used in every iteration of the loop, but it costs cycles because the iterations are ran one after the other.
  - To produce a design with less latency, this loop should be unrolled. Unrolling a loop allows the operations in the loop to occur in parallel, if timing and sequential dependencies (read and writes to registers and memories) allow.
- In this design, the `shift_reg` array is being implemented in an internal RAM.
  - Even if the loop is unrolled, each iteration of the loop requires a read and write operation to this RAM.
  - By default, arrays are implemented as RAMs. The `shift_reg` array can, however, be partitioned into individual elements. Each element is implemented by a register, allowing a shift register to be used for the implementation.
  - Once the loop is unrolled, the Vivado HLS tool can perform this step automatically because it is a small RAM. All optimizations performed on the design are reported in the Console. However, because this is required, it is always better to explicitly specify it.
- The coefficient port `c` is using a single-port RAM interface.
  - This is correct; however, because this is required, it is always better to explicitly specify it.
- Input port `x` is required to have an input valid signal associated with it.
  - This port requires an IO protocol, which uses an input valid signal.
- There are two multipliers being used, but in the C code they are both in mutually exclusive branches.
  - The Vivado HLS tool might not share components if the cost of the multiplexors could mean violating timing.

    - The timing is close in this example: 10ns minus 1.25ns, the default clock uncertainty. However, the only real way to be sure if they could be shared is to view the results after place and route.
    - For this example, sharing is forced. This demonstrates a useful technique for minimizing area.

- Most importantly, The multipliers are taking four cycles each to complete! Additionally, only two multipliers are shown in the Resource Viewer, but the earlier report (Figure 1-23) gave an estimate that six DSP48s are required.

  The multiplication operations are using standard C integer types (32-bit) and it requires three DSP48s to implement a 32-bit multiplication. However, this design is only required to accept 8-bit input data.

IMPORTANT: Ensure that the C code is using the correct bit-accurate types before proceeding to synthesis or it can result in larger and slower hardware.

Before performing any optimizations on this design, you must modify the source code to the required 8-bit data types.

Summary

- When synthesis completes a report on the design, it automatically opens.
- More detailed and in-depth analysis of the implementation can be performed using the Design Viewer.
- In the Design Viewer, start with the Control Flow Graph and work towards the Resource Viewer for a complete understanding of how the C was implemented. The Schedule Viewer allows operations to be correlated with the C source and output HDL code.

Bit-Accurate Design

The first step in bit-accurate design is to introduce the bit-accurate types (also called arbitrary precision types), into the source code.

When arbitrary precision types are added to a C function, it is important to validate the design and ensure that it does what it is supposed to do (rounding and truncation are of critical importance) and validates the results at the C level.

The information to make the source code bit-accurate is already included in the example files.

Update the C Code

Creating a New Solution

To preserve the existing results so they can be compared against the new results, create a new solution.

1. In the Vivado HLS GUI, select the **New Solution** button, shown in Figure 1-30.

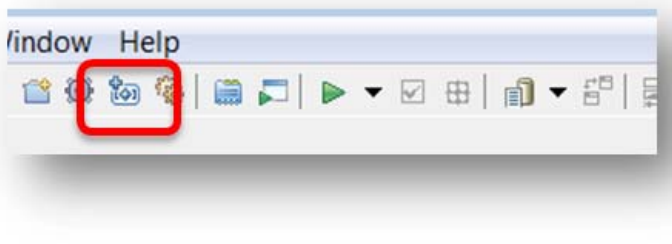


Figure 1-30: **New Solution Toolbar Button**

   The New Solution dialog box opens.

2. Leave the default solution name as `solution2`. Do not change any of the technology or clock settings.
3. Click **Finish**.

   The new solution, `solution2`, is created and opened.

4. Confirm that `solution2` is highlighted in bold in the Project Explorer, indicating that it is the current active solution.

   Note: Open files use up memory. If they are required, keep them open; otherwise it is good practice to close them.

5. Close any existing tabs from previous solutions. In the Project menu, select **Close Inactive Solution Tabs**.

Bit-Accurate Types, Simulation, and Validation

The source already contains the code to use bit-accurate types. The header file `fir.h` contains the following:

```
#ifdef BIT_ACCURATE
#include "ap_cint.h"
typedef int8coef_t;
typedef int8data_t;
typedef int8acc_t;
#else
typedef intcoef_t;
```

```
typedef intdata_t;
typedef intacc_t;
#endif
```

This code ensures that if the macro BIT_ACCURATE is defined during compile or synthesis, the Vivado HLS header file (`ap_cint.h`), which defines bit-accurate C types, is included and 8-bit integer types (`int8`) are used instead of the standard 32-bit integer types.

In addition, new 8-bit data types result in different output data from the `fir` function. The test bench (`fir_test.c`) is also written to ensure that the output data can be easily compared with a different set of golden results, which is done if the macro BIT_ACCURATE is defined.

```
#ifdef BIT_ACCURATE
  printf ("Comparing against bit-accurate data \n");
  if (system("diff -w out.dat out.gold.8.dat")) {
#else
  printf ("Comparing against output data \n");
  if (system("diff -w out.dat out.gold.dat")) {
#endif
```

IMPORTANT: In general, changing the project setting is not a good idea as the project settings affect every solution in the design. If `solution1` is re-executed, it uses these new project settings and gives different results. This technique is shown here for two reasons: to show it is a possible way to compare solutions, and to highlight that the results for `solution1` changes if it is re-executed and the project settings have been changed.

To ensure that the macro BIT_ACCURATE is defined for the C simulation and synthesis, the project setting must be updated.

1. Select the **Project Settings** toolbar button, shown in Figure 1-31.

Figure 1-31: **Project Settings Button**

The next few steps describe updating the settings for the C simulation.

2. Define the macro BIT_ACCURATE by doing the following:
   a. In the Simulation section of the Project Settings, select **`fir_test.c`**.
   b. Click the **Edit CFLAGS** button.

c. Add **-DBIT_ACCURATE** to define the macro.

d. Click **OK**.

The CFLAGS section is used to define any options required to compile the C program. This example uses the compiler option **-D**; however, all `gcc` options are supported in the CFLAGS section (**-I**<*include path*> etc.).

Note: There is no need to include any Vivado HLS header files, such as `ap_cint.h`, using the include flag. The Vivado HLS include directory is automatically searched.

3. Update the data file used by the test bench by doing the following:

a. In the Simulation section, select the **out.gold.dat** file.

b. Click the **Remove** button to remove the file from the project.

If macro BIT_ACCURATE is defined, this file is no longer used by the test bench and is not required in the project.

c. Click the **Add Files** button.

d. Add the **out.gold.8.dat** file to the project.

The updated Simulation section is shown in Figure 1-32.

e. Click **OK**.

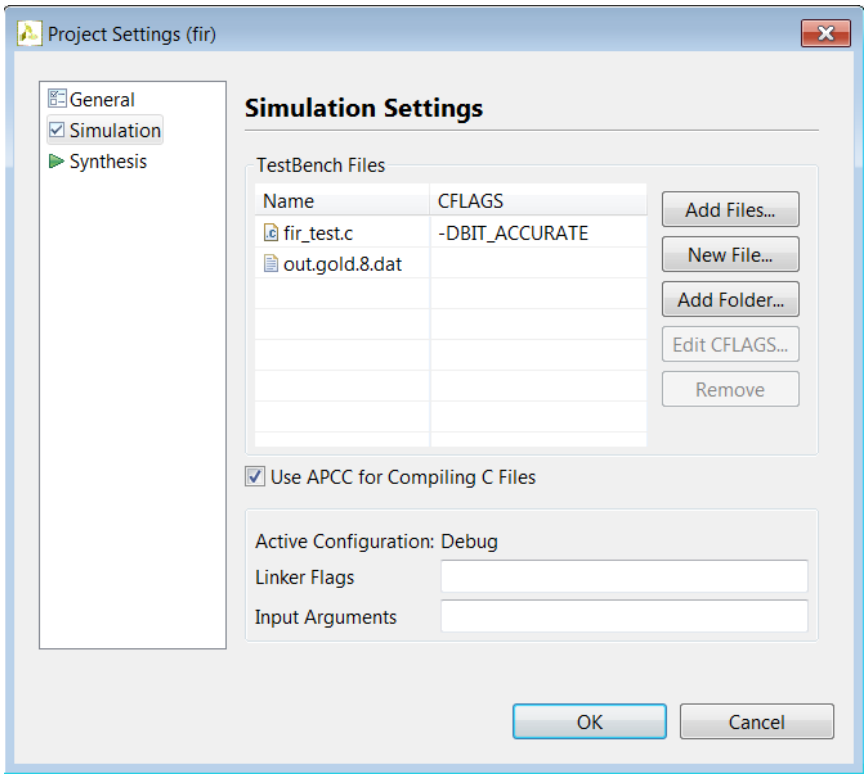


Figure 1-32: **Project Simulation Settings**

4. Ensure that the new C data types are correctly compiled by doing the following:
   a. In the Synthesis section of the Project Simulation Settings dialog box, select source file **`fir.c`**.
   b. Click the **EDIT CFLAGS** button.
   c. Add **`-DBIT_ACCURATE`** into the CFLAGS dialog box.
   d. Click **OK**.

   Figure 1-33 shows the settings for the CFLAGS to synthesize the design using bit-accurate types.

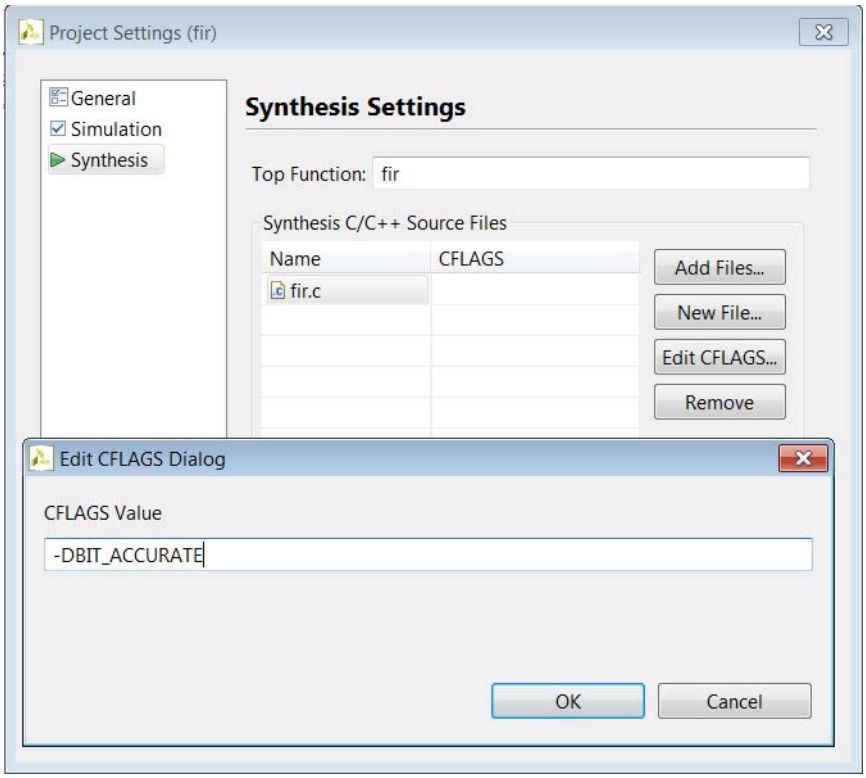


Figure 1-33: **CFLAGS Settings Dialog Box**

   The next step is to confirm that C function is validated with the new project settings.

5. Click the **Run C Simulation** toolbar button and select **OK** with all options un-selected, to recompile and run the C simulation.

   The output displays in the console window. See Figure 1-34.

   The console now shows the message "Comparing against bit-accurate data" as specified in the test bench when the BIT_ACCURATE macro is defined.

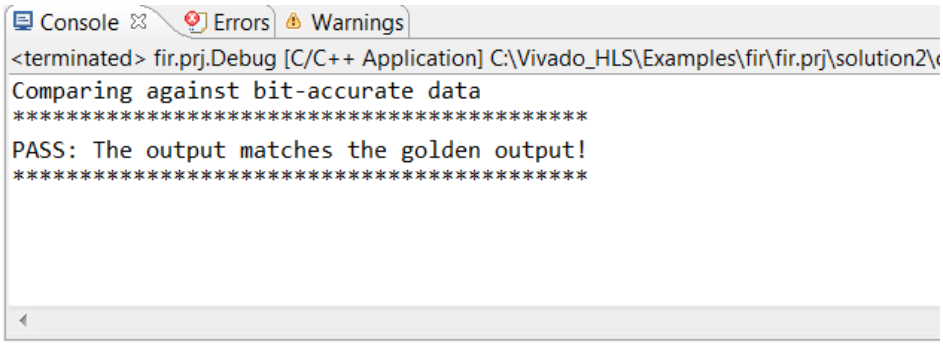
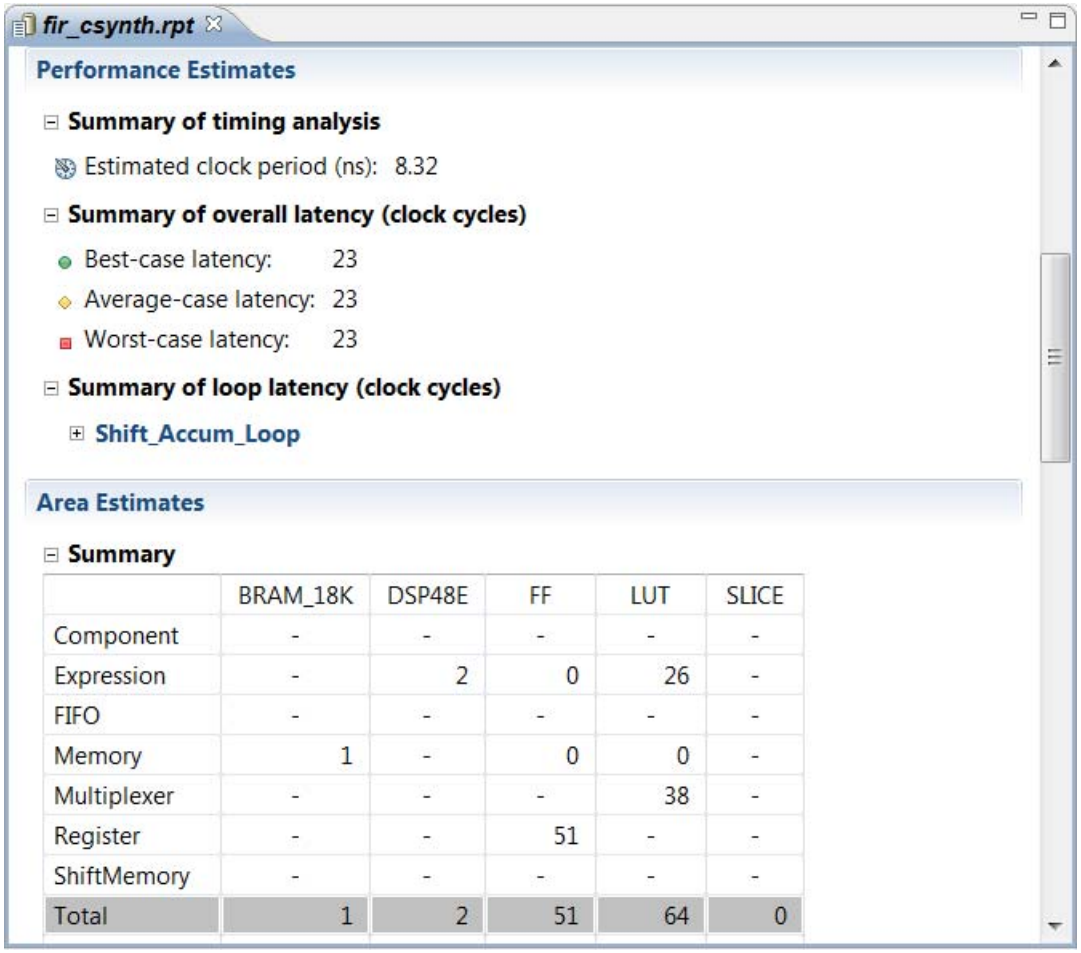


Figure 1-34: **C Simulation Output**

Synthesis and Comparison

Click the **Synthesis** toolbar button to re-synthesize the design.

When synthesis is re-executed for `solution2`, the results are as shown in Figure 1-35, where only two DSP48s are used but the estimated clock frequency is now slower, however still within the specified 10ns target.



	BRAM_18K	DSP48E	FF	LUT	SLICE
Component	-	-	-	-	-
Expression	-	2	0	26	-
FIFO	-	-	-	-	-
Memory	1	-	0	0	-
Multiplexer	-	-	-	38	-
Register	-	-	51	-	-
ShiftMemory	-	-	-	-	-
Total	1	2	51	64	0

Figure 1-35: **Synthesis Results Re-done**

The effect of changing to bit-accurate types can be seen by comparing `solution1` and `solution2`. To easily compare the two solutions, use the **Compare Reports** toolbar button (see Figure 1-36).

Figure 1-36: **Compare Reports Button**

1. Add **`solution1`** and THEN **`solution2`** to the comparison.
2. Click **OK**.

Figure 1-37 shows the comparison of the reports for `solution1` and `solution2`.

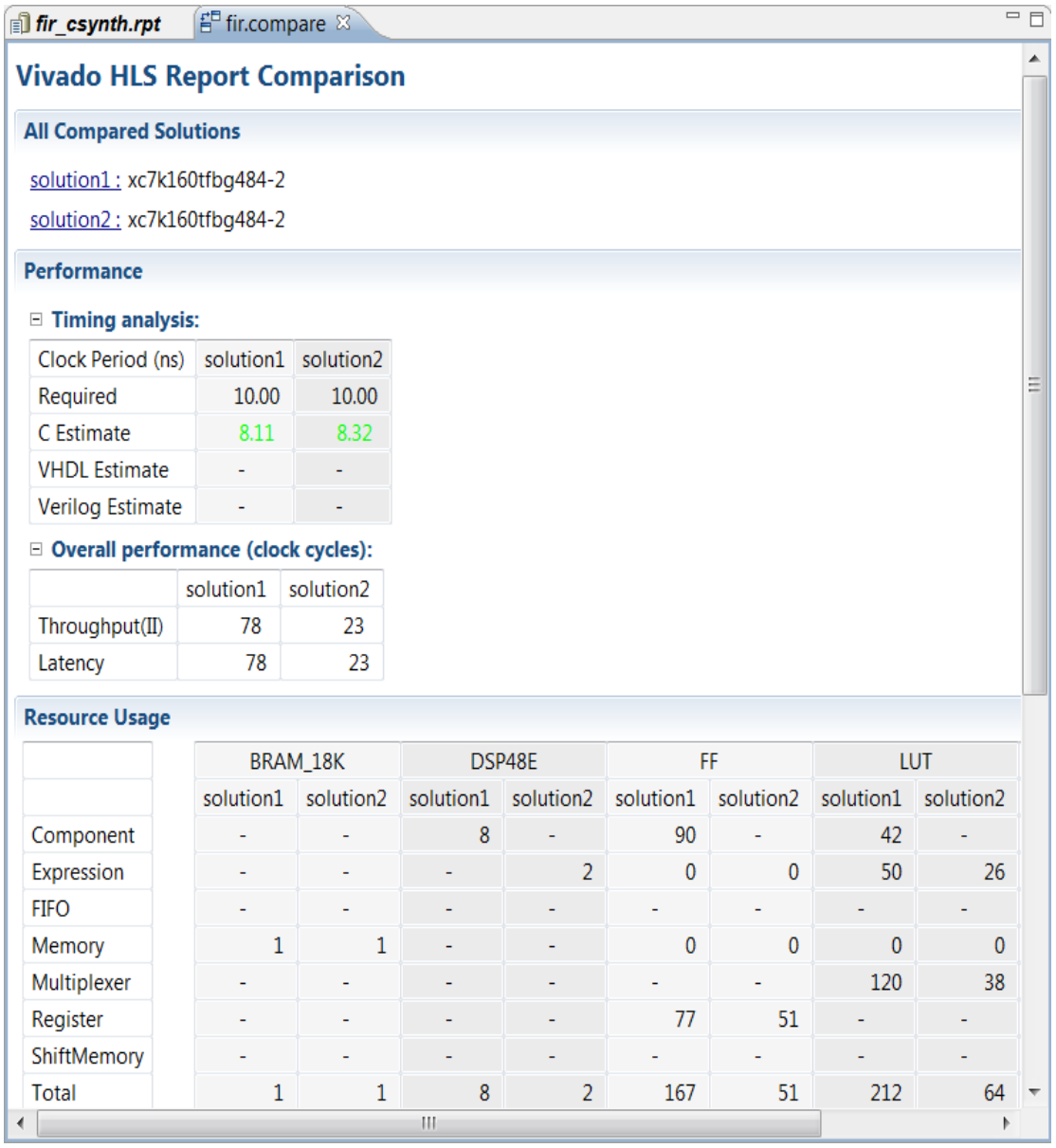


Figure 1-37: **solution1 vs. solution2**

Using bit-accurate data types has resulted in a faster and smaller design. Specifically:

- The number of DSP48s has been reduced to only two.
- Because a single DSP48 is being used for each multiplication instead of four, each multiplication can be performed in one clock cycle and the latency of the design has been reduced.

- There has also been a reduction in the number of registers and LUTs, which is to be expected with a smaller data type.

It is worth noting the following subtlety in the reporting:

- In `solution1`, the multiplications were implemented as pipelined multipliers. These are implemented as sub-blocks (or components) in the RTL and so the DSP were all reported in the components section of the report.
- In `solution2`, the multiplications are single cycle and implemented in the RTL with a multiplication operator ("*") and are therefore listed as expressions; operations at this level of the hierarchy.

Summary

The act of rewriting the design to be bit-accurate was deliberately introduced into this tutorial to show the steps for performing it. They are:

1. Update the code to use bit-accurate types.
2. Include the appropriate header file to define the types.
   - For C designs, `ap_cint.h`

     Be aware bit-accurate types in C must have the `AutoCC` option enabled and cannot be analyzed in the debug environment (C++ and SystemC types can).
   - For C++ design, `ap_int.h`
   - For SystemC designs, `systemc.h`
3. Simulate the design and validate the results before synthesis.

Design Optimization

The following optimizations, discussed earlier, can now be implemented:

- Unroll the `Shift_Accum_Loop` loop to reduce latency.
- Partition the array `shift_reg` to prevent a BRAM being used, and allow a shift register to be used.
- Specify the input array `c` as a single-port RAM in order to guarantee a single-port RAM interface.
- Ensure that the input port `x` uses a valid handshake.
- Force sharing of the multipliers.

The first sets of optimizations to perform are those which must be performed: those associated with the interface. No matter what other optimizations are performed, the RTL interface must match the requirements.

Optimization: IO Interface

The following optimizations must be performed in `solution3`:

- Specify the input array `c` as a single-port RAM in order to create a single-port RAM interface.
- Ensure that the input port `x` uses a valid handshake.

Step 1: Creating a New Solution

To preserve the existing results, create a new solution, `solution3`, by doing the following.

1. Click the **New Solution** button to create a new solution.
2. Leave the default solution name as `solution3`. Do not change any of the technology or clock settings.
3. Click **Finish**.

   `solution3` is created and automatically opens.

   When `solution3` opens, confirm that `solution3` is highlighted in bold in the Project Explorer pane, indicating that it is the current active solution.

   Note: Open files use up memory. If they are required, keep them open; otherwise it is good practice to close them.
4. Close any existing tabs from previous solutions. In the **Project** menu, select **Close Inactive Solution Tabs**.

Step 2: Adding Optimization Directives

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

1. In the Project Explorer, expand the source container in `solution3` (see Figure 1-38).

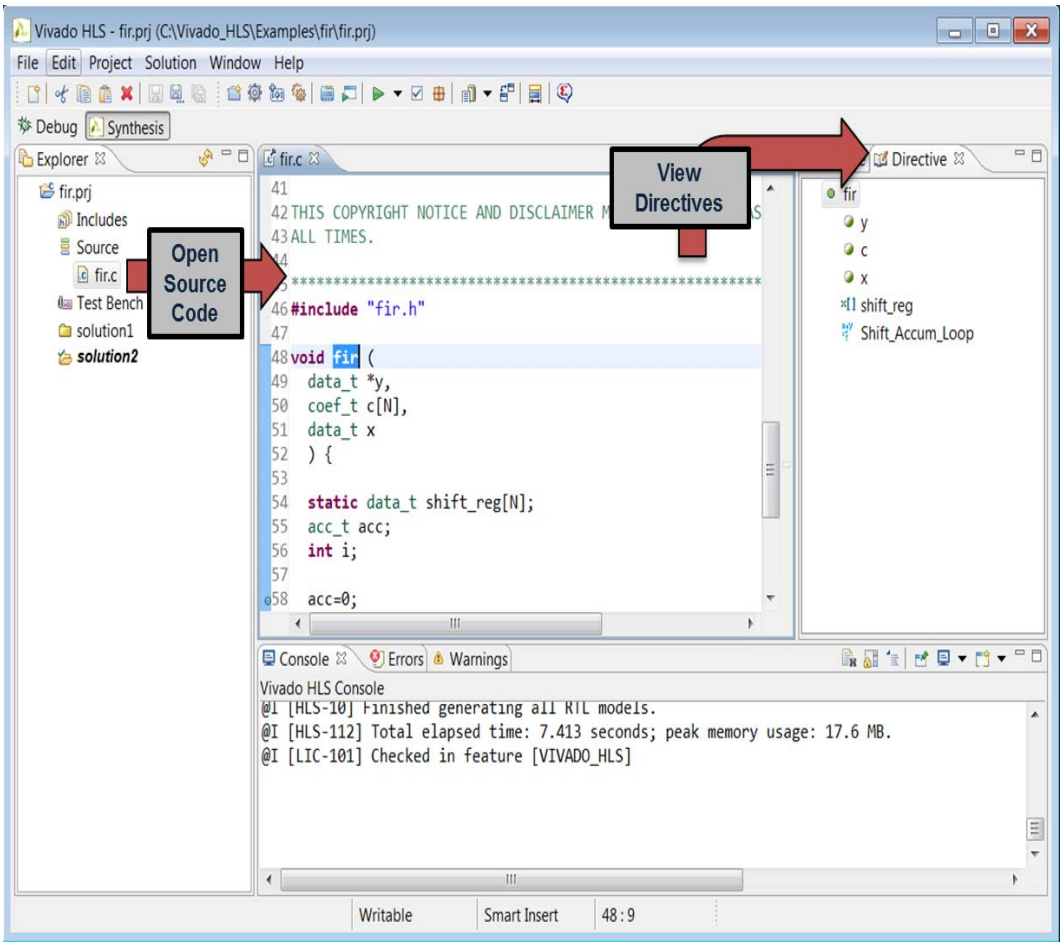


Figure 1-38: **Adding Optimization Directives**

2. Double-click **`fir.c`** to open the file in the Information pane.
3. Click the **Directive Tab** (see Figure 1-38).

   You can now apply the optimization directives to the design.

4. In the Directive tab, select the **`c`** argument/port (green dot).
5. Right-click and select **Insert Directives**.
6. Implement the array by doing the following:
   a. Select **RESOURCE** from the Directive drop-down menu.
   b. Click the **core** box.
   c. Select **RAM_1P_BRAM**, as shown in Figure 1-39.

      This ensures that the array is implemented using a single port BRAM.

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

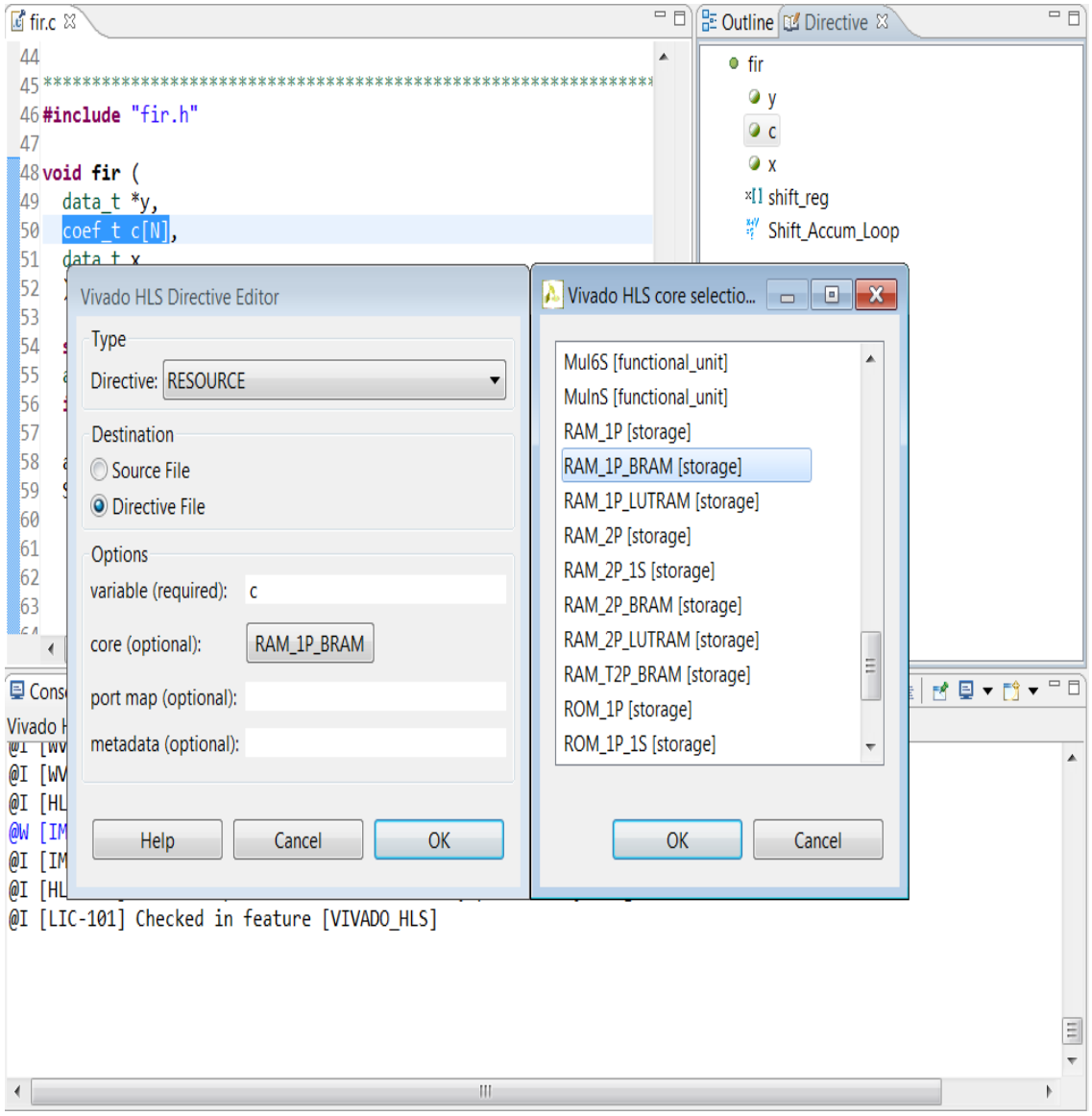


Figure 1-39: **Adding a Resource Directive**

This directive informs the Vivado HLS tool that array `c` is implemented as a single-port RAM. Because the array is on the function interface, this is equivalent to the RAM being "off-chip." In this case, the Vivado HLS tool creates the appropriate interface ports to access it.

The interface ports created (the number of address ports) are determined by pins on the RAM_1P_BRAM core. A complete description of the cores in the Vivado HLS library is provided in the *Vivado Design Suite User Guide: High-Level Synthesis (UG902) > High-Level Synthesis Operator and Core Guide chapter.*

Next, specify port `x` to have an associated valid signal/port.

1. In the Directive tab, select input port `x` (green dot).
2. Right-click and select **Insert Directives**.
3. Select **Interface** from the Directive drop-down menu.
4. Select **`ap_vld`** for the mode.
5. Click **OK** to apply the directive.

   When complete, the Directive pane looks like Figure 1-40. Select any incorrect directive and use the mouse right-click to modify it.

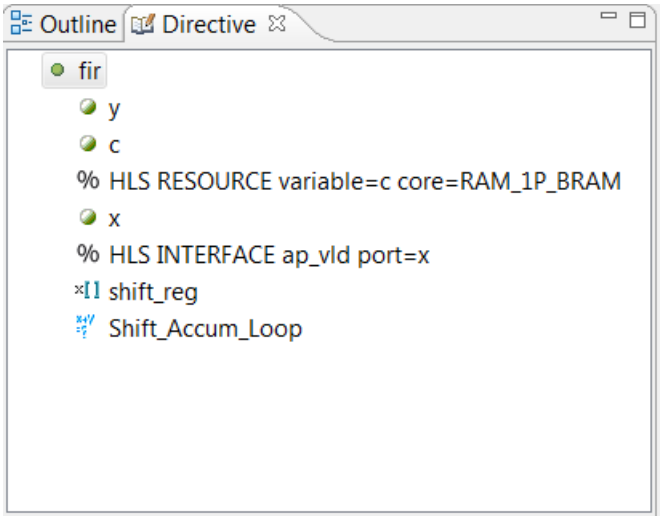
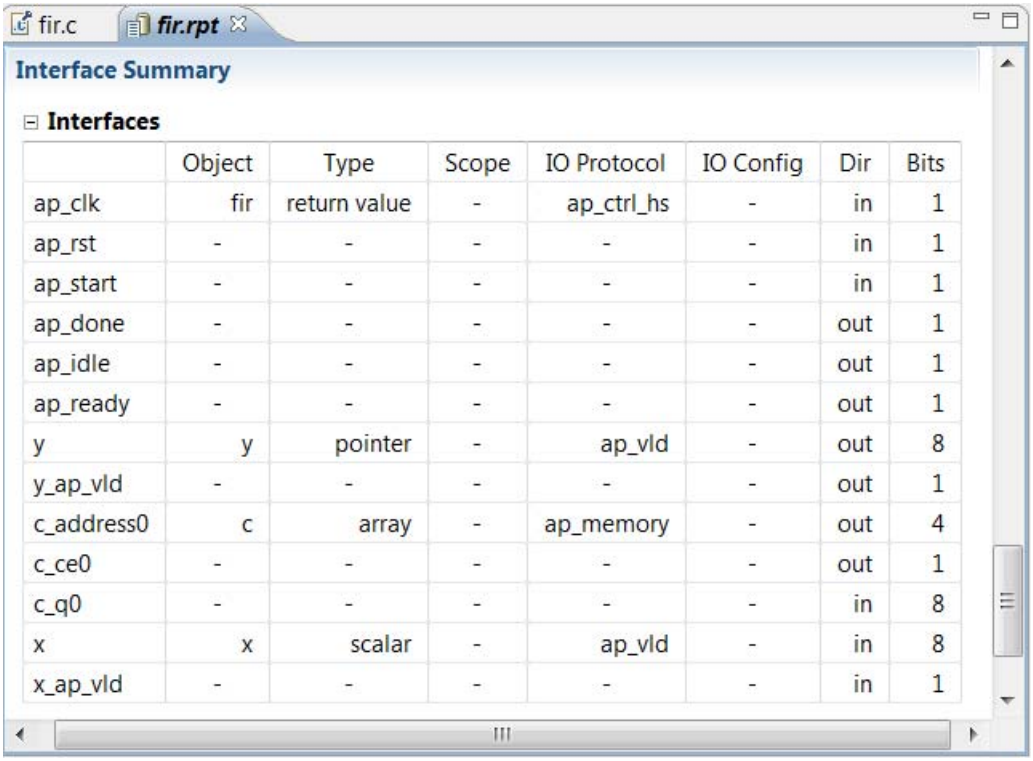


Figure 1-40: **Directive Tab solution3**

Step 3: Synthesis

Now that the optimization directives have been applied, run synthesis on `solution3`. Click the **Synthesis** toolbar button to synthesize the design.

When synthesis completes, the synthesis report automatically opens. Scroll down, or use the outline pane to jump to the interface section. Figure 1-41 shows the interfaces are now correctly defined.



	Object	Type	Scope	IO Protocol	IO Config	Dir	Bits
ap_clk	fir	return value	-	ap_ctrl_hs	-	in	1
ap_rst	-	-	-	-	-	in	1
ap_start	-	-	-	-	-	in	1
ap_done	-	-	-	-	-	out	1
ap_idle	-	-	-	-	-	out	1
ap_ready	-	-	-	-	-	out	1
y	y	pointer	-	ap_vld	-	out	8
y_ap_vld	-	-	-	-	-	out	1
c_address0	c	array	-	ap_memory	-	out	4
c_ce0	-	-	-	-	-	out	1
c_q0	-	-	-	-	-	in	8
x	x	scalar	-	ap_vld	-	in	8
x_ap_vld	-	-	-	-	-	in	1

Figure 1-41: **solution3 Results: Correct IO Interface**

Port `x` is now an 8-bit data port with an associated input valid. The coefficient port `c` is configured to access a single port RAM and output `y` has an associated output valid.

Optimization: Small Area

The design in `solution3` represents the starting point for further optimizations. Begin by creating a new solution, as shown in Figure 1-42.

Step 1: Creating a New Solution

1. Click the **New Solution** button to create a new solution.
2. Name the solution **`solution4_area`**. The solution names default to solution1, 2, 3, and so on, but can be named anything.
3. Now that we have existing directives to copy over, it's important to ensure the **Copy Existing Directives from Solution** boxes are checked (default setting).
4. In the Project menu, select **Close Inactive Solution Tabs** to close any existing tabs from previous solutions.

   When `solution4_area` opens, confirm that it is highlighted in bold in the Project Explorer pane, indicating that it is the current active solution.

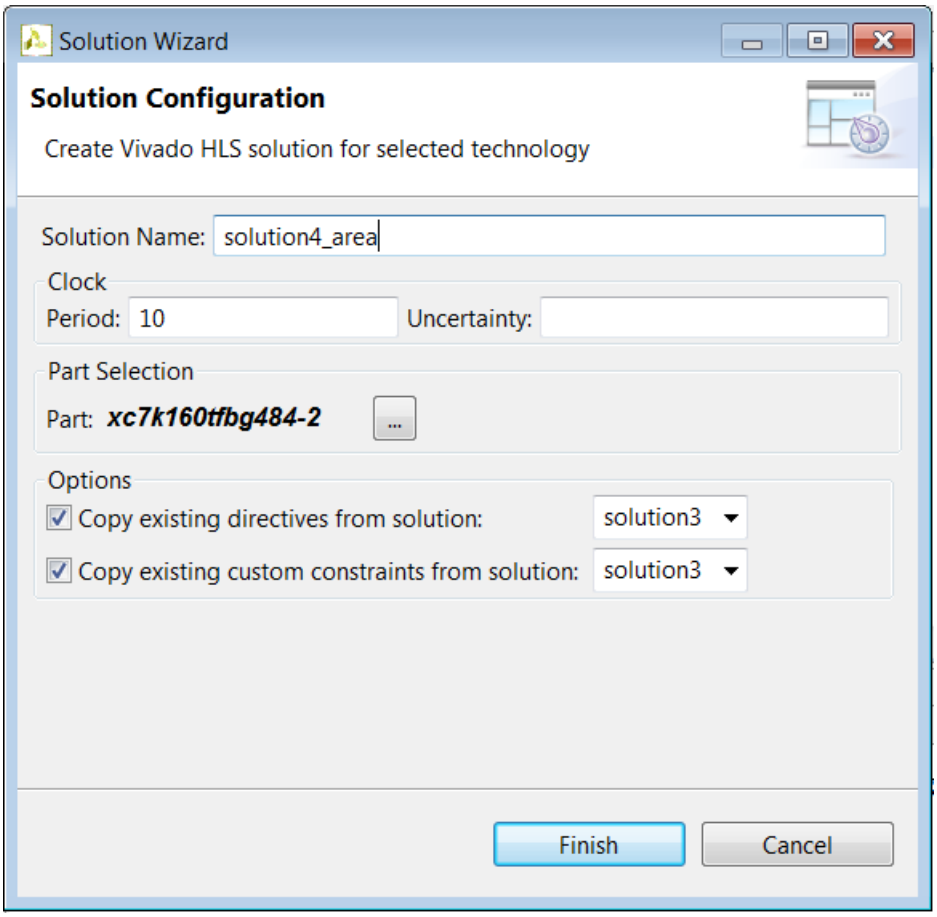


Figure 1-42: **Create solution4_area**

Step 2: Sharing of Multipliers

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

1. Open the solution settings by selecting **Solution > Solution Settings**.
2. Select **General** on the left-hand side menu.
3. Click **Add** to open the list of configurations.
4. Select **`config_bind`** from the drop-down menu.
5. Specify **`mul`** in the **min_op** (minimize operator) field, as shown in Figure 1-43.
6. Click **OK** to set the configuration.
7. Click **OK** again to close the Solution Settings window.

   The `config_bind` command controls the binding phase, where operators inferred from the code are bound to cores from the library. The `min_op` option tells Vivado HLS

to minimize the number of the specified operators (`mul` operations, in this case) and overrides any `mux` delay estimation.

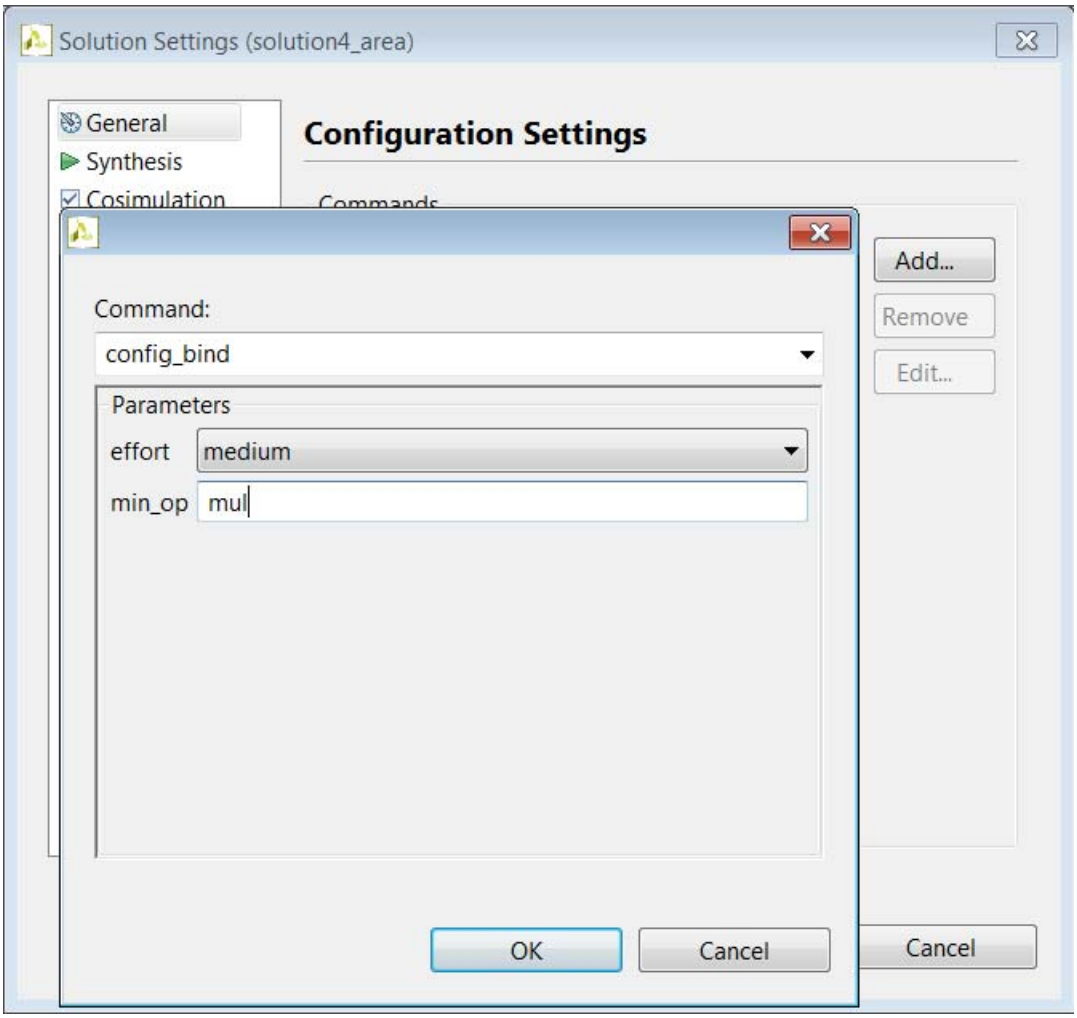
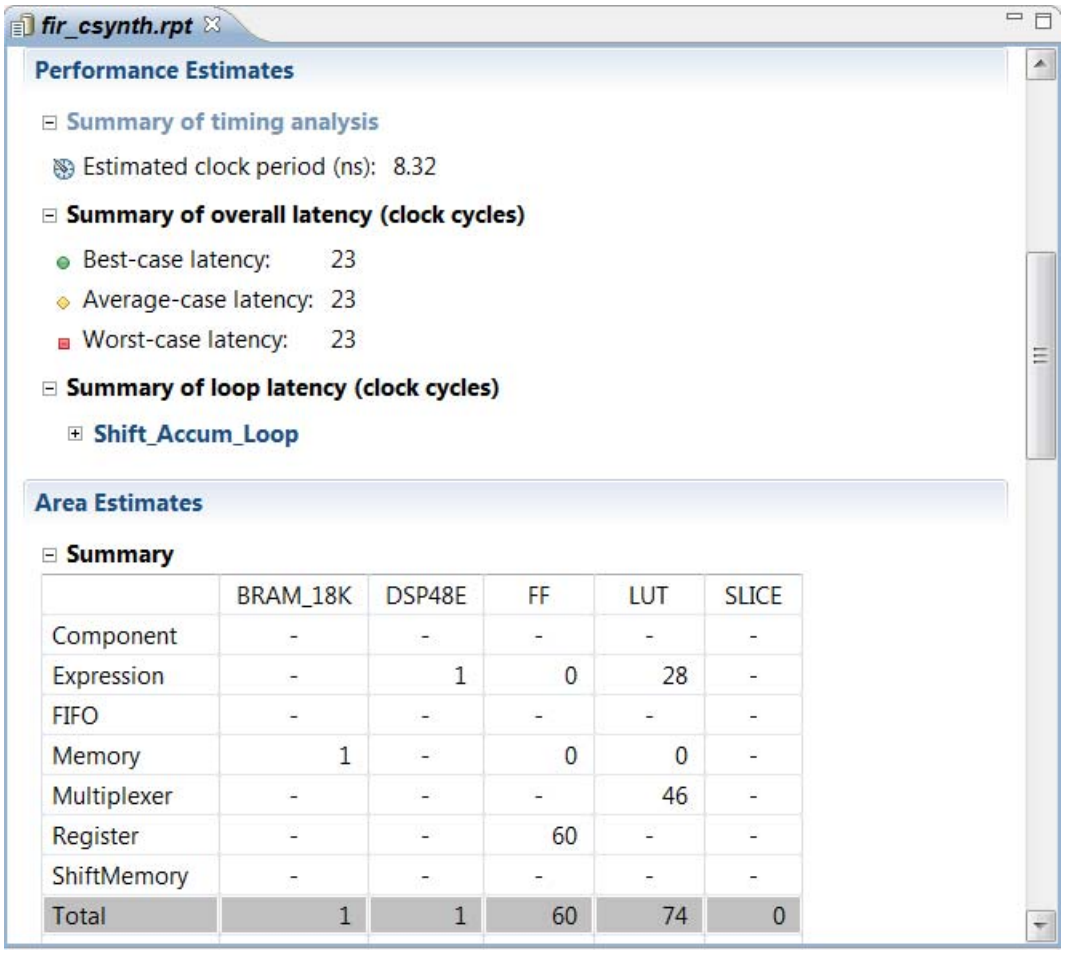


Figure 1-43: **Adding Custom Constraints**

Step 3: Synthesis

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

When synthesis completes, the synthesis report opens showing that the configuration command was successful and only a single multiplier is now used in the design. See Figure 1-44.



	BRAM_18K	DSP48E	FF	LUT	SLICE
Component	-	-	-	-	-
Expression	-	1	0	28	-
FIFO	-	-	-	-	-
Memory	1	-	0	0	-
Multiplexer	-	-	-	46	-
Register	-	-	60	-	-
ShiftMemory	-	-	-	-	-
Total	1	1	60	74	0

Figure 1-44: **Solution4 Results**

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

Optimization: Highest Throughput

To add the optimizations to create a design with the highest throughput, unroll the loop and partition the memory. The solution `solution3`, with the correct IO interface, is used as the starting point.

Step 1: Creating a New Solution

Begin by creating a new solution.

1. Click the **New Solution** button to create a new solution.
2. Name the solution `solution5_throughput`.

3. Select the **Copy existing directives from solution** check box.
4. Select **`solution3`** from the drop-down menu.

   The IO directives specified in `solution3` copy into `solution5_throughput`.

5. In the Project menu, select **Close Inactive Solution Tabs** to close any existing tabs from previous solutions.

Step 2: Unrolling the Loop

The following steps, summarized in Figure 1-45, explain how to unroll the loop.

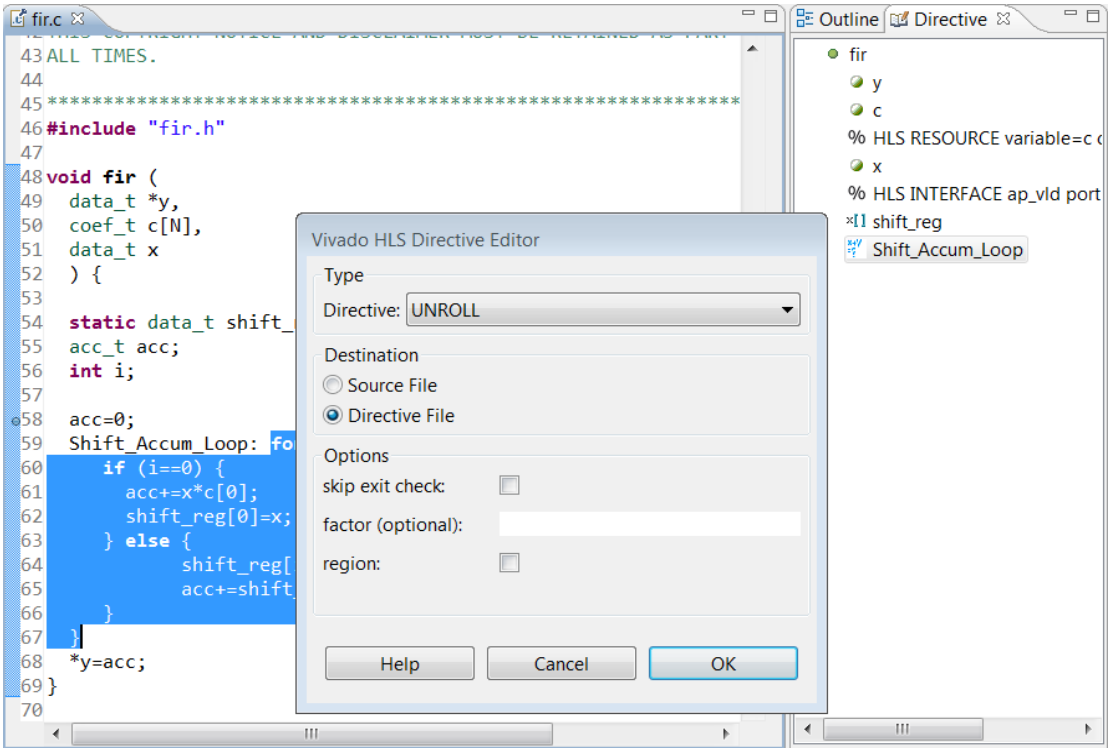


Figure 1-45: **Unrolling FOR Loop**

1. In the Directive tab, select loop **`Shift_Accum_Loop`**.

   Note: Open the source code to see the Directive tab.

2. Right-click and select **Insert Directives**.
3. From the Directive drop-down menu, select **Unroll**.
4. Select **OK** to apply the directive.

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

Apply the directive to partition the array into individual elements, which are then arranged as a shift-register.

5. In the Directive tab, select array **`shift_reg`**.
6. Right-click and select **Insert Directives**.
7. Select **partition** from the Directive drop-down menu.
8. Specify the type as **complete**.
9. Select **OK** to apply the directive.

With the two directives imported from `solution3` and the two new directives just added, the directive pane for `solution5_throughput` is now as shown in Figure 1-46.

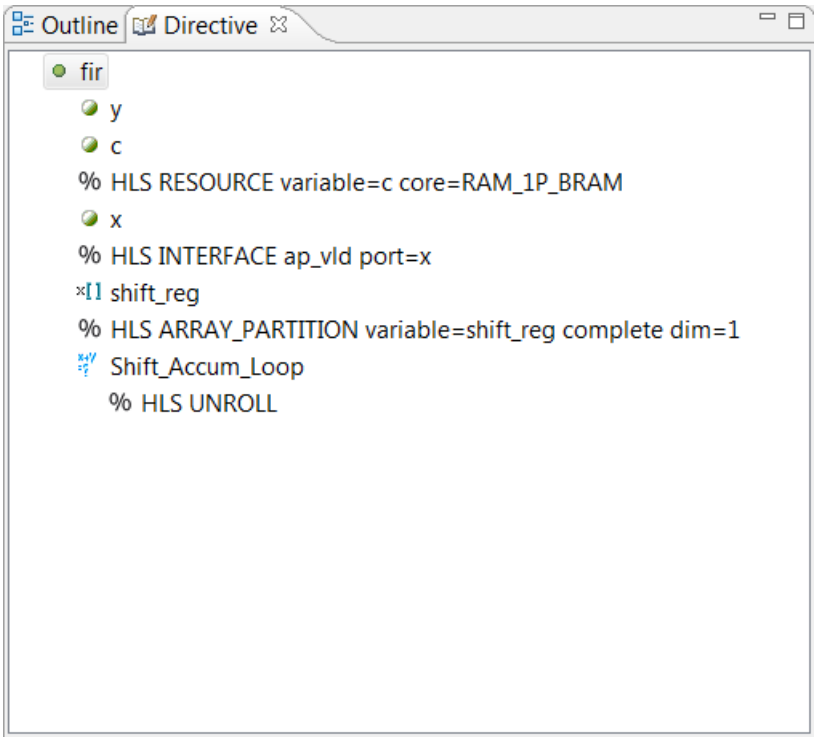


Figure 1-46: **solution5_throughout Directives**

Step 3: Synthesis

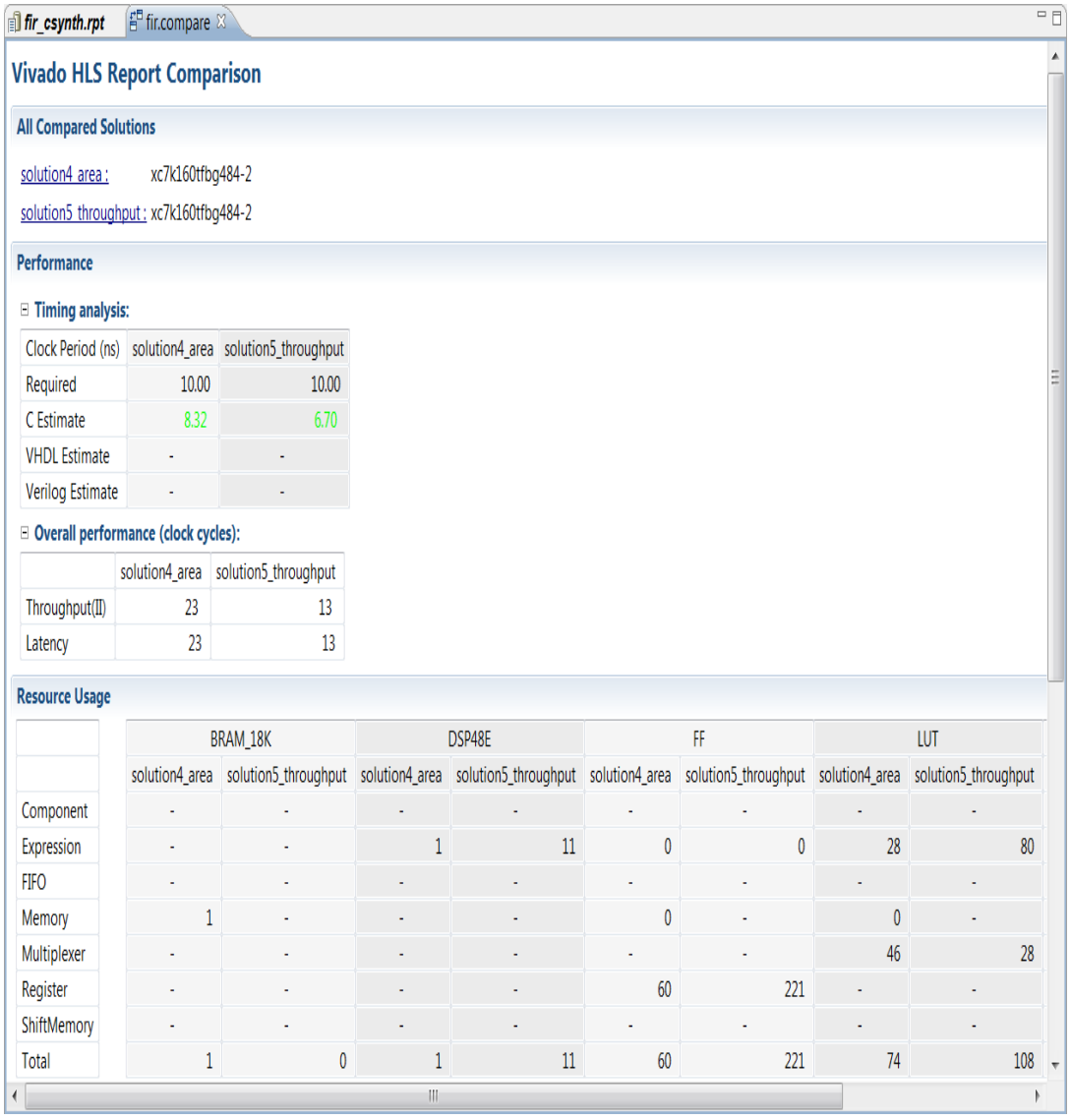
1. Click the **Synthesis** button to synthesize the design.

   When synthesis completes, the synthesis report automatically opens.

2. To compare `solution4_area` with `solution5_throughput`, click the **Compare Reports** button.

3. Add **`solution4_area`** and **`solution5_throughput`** to the comparison.
4. Click **OK**.

   Figure 1-47 shows the comparison of the reports from `solution4_area` and `solution5` (the LUTS are not shown in Figure 1-47 due to the wide nature of the report).


fir_csynth.rpt | fir.compare

Vivado HLS Report Comparison

All Compared Solutions

solution4_area : xc7k160tfbg484-2

solution5_throughput : xc7k160tfbg484-2

Performance

Timing analysis:

Clock Period (ns)	solution4_area	solution5_throughput
Required	10.00	10.00
C Estimate	8.32	6.70
VHDL Estimate	-	-
Verilog Estimate	-	-

Overall performance (clock cycles):

	solution4_area	solution5_throughput
Throughput(II)	23	13
Latency	23	13

Resource Usage

	BRAM_18K		DSP48E		FF		LUT	
	solution4_area	solution5_throughput	solution4_area	solution5_throughput	solution4_area	solution5_throughput	solution4_area	solution5_throughput
Component	-	-	-	-	-	-	-	-
Expression	-	-	1	11	0	0	28	80
FIFO	-	-	-	-	-	-	-	-
Memory	1	-	-	-	0	-	0	-
Multiplexer	-	-	-	-	-	-	46	28
Register	-	-	-	-	60	221	-	-
ShiftMemory	-	-	-	-	-	-	-	-
Total	1	0	1	11	60	221	74	108

Figure 1-47: **solution4_area vs. solution5_throughput**

Both designs operate within the 10ns clock period. The small design is using a BRAM but only one DSP48 and about 60 registers. The small design takes 24 clock cycles to complete.

The high throughput design processes the samples at the highest possible rate. It requires one clock cycle to read each of the 11 coefficients from the RAM plus one cycle overhead to generate the first address. However, it is using 11 DSP48s and more than twice the number of flip-flops as the small design.

Scroll down the report window to view the estimates for power consumption. At this level of abstraction, the power consumption data should only be used to compare different solutions. In this case, it is clear that `solution4_area` uses much less power than `solution5_throughput` and that the increase is caused by both additional registers and expressions (logic).

Summary

- You can add optimization directives to the design using the Directive tab. The source code must be open in the Information Pane in order to view the Directive tab.
- Creating different solutions for each new set of directives allows for the solutions to be easily compared inside the GUI.

RTL Verification and Export

The Vivado HLS tool allows both RTL verification and RTL export to be performed from the GUI. The RTL verification and RTL export menus in the GUI are also supported at the Tcl command level (discussed later).

Details on the various options are not discussed in this tutorial but can be found by reviewing the associated Tcl command, available from the GUI help menu. The Tcl commands for RTL verification and RTL export are `cosim_design` and `export_design`, respectively.

RTL Verification

The generated RTL can now be verified with the original C test bench. A new RTL test bench is NOT required with the Vivado HLS tool.

For RTL simulation, the Vivado HLS tool supports industry standard VHDL and Verilog RTL simulators and includes a SystemC simulation kernel allowing the SystemC RTL output to be verified.

The RTL can always be verified using the SystemC kernel and no 3rd party RTL simulator license is required for this.

To use the other supported simulators, a license for the simulator is required, and the simulator executable should be available in the search path.

In this example, the SystemC RTL will be verified. Start with the `solution5_throughput` solution. Make sure `solution5_throughput` is highlighted in bold in the Project Explorer, indicating it is the currently active solution.

1. Click the **Simulation** button in the toolbar, as shown in Figure 1-48.

Figure 1-48: **Simulation Toolbar Button**

The co-simulation dialog opens, as shown in Figure 1-49.

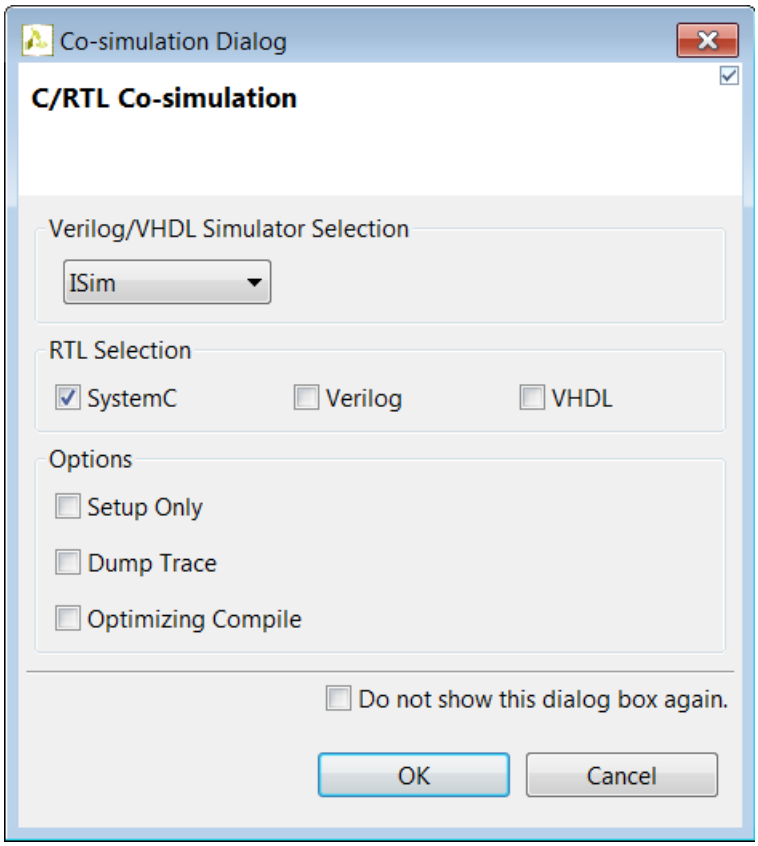


Figure 1-49: **RTL Verification Menu**

2. For VHDL and Verilog, leave the drop-down menus set to Skip, and select **SystemC** from the corresponding SystemC drop-down menu.

3. Click **OK**.

   Simulation starts.

When the simulation ends it automatically opens the simulation report in the Information pane (see Figure 1-50). For every simulation ran, there is an indication of "pass/fail" and the measured minimum/maximum latency.

The results of the simulation can be seen in the Console pane. The simulation ends with the same confirmation message as the original C simulation (since it's the same test bench), confirming the RTL results. The message confirms the bit-accurate behavior of the test bench.

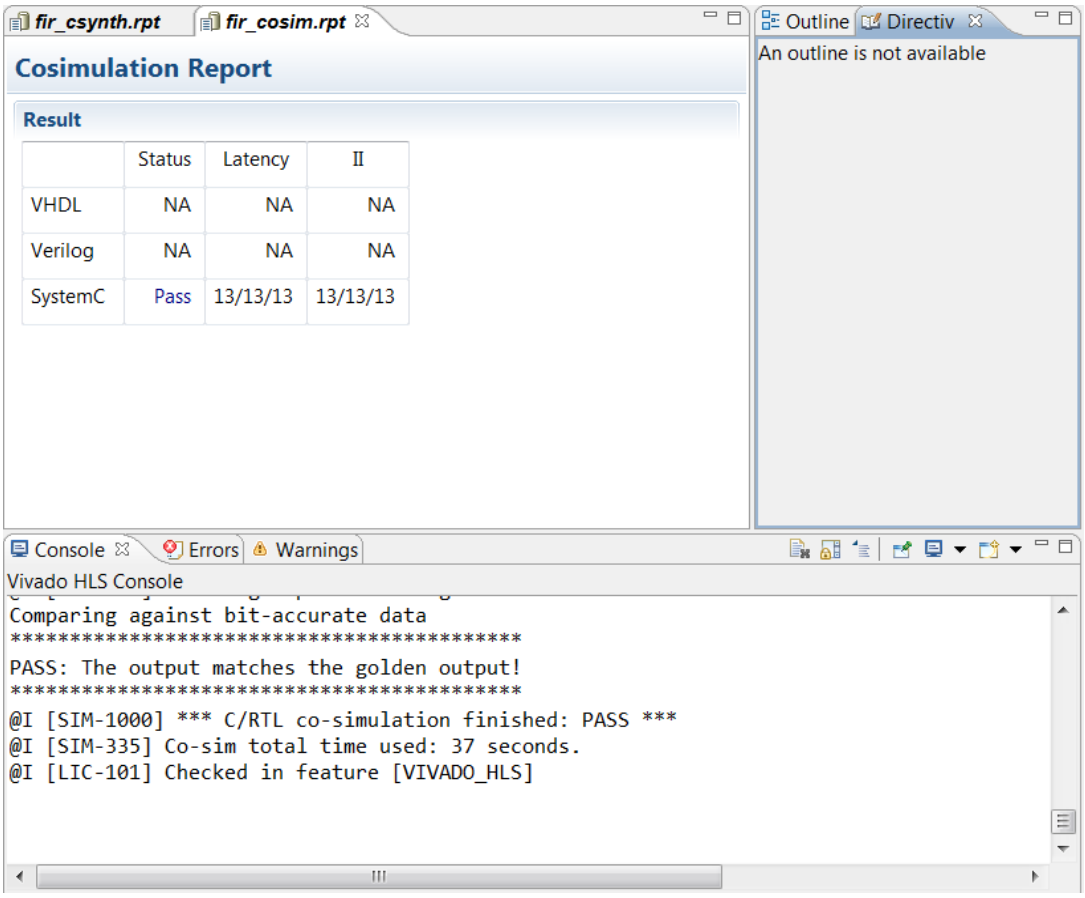


Figure 1-50: **Simulation Report**

RTL Export

The final step in the Vivado HLS flow is to export the RTL design as an IP block for use with other Xilinx tools.

Optionally, RTL logic synthesis can be performed: these logic synthesis results are only to evaluate the RTL and confirm the actual timing and area after logic synthesis is similar to the estimated timing and area predicted by Vivado HLS. These RTL results are not part of the exported IP: the IP includes only the RTL which will be synthesized with the remainder of the design.

To use RTL logic synthesis tools, the executable should be available in the search path. For 7-Series devices the path to executable vivado must be in the search path. For other devices, the ISE executable xtclsh must be in the search path.

1. Click the **Export RTL** button in the toolbar, as shown in Figure 1-51.

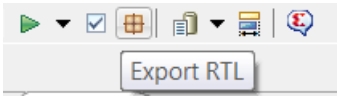


Figure 1-51: **Export RTL Toolbar Button**

The Export RTL dialog opens, as shown in Figure 1-52.

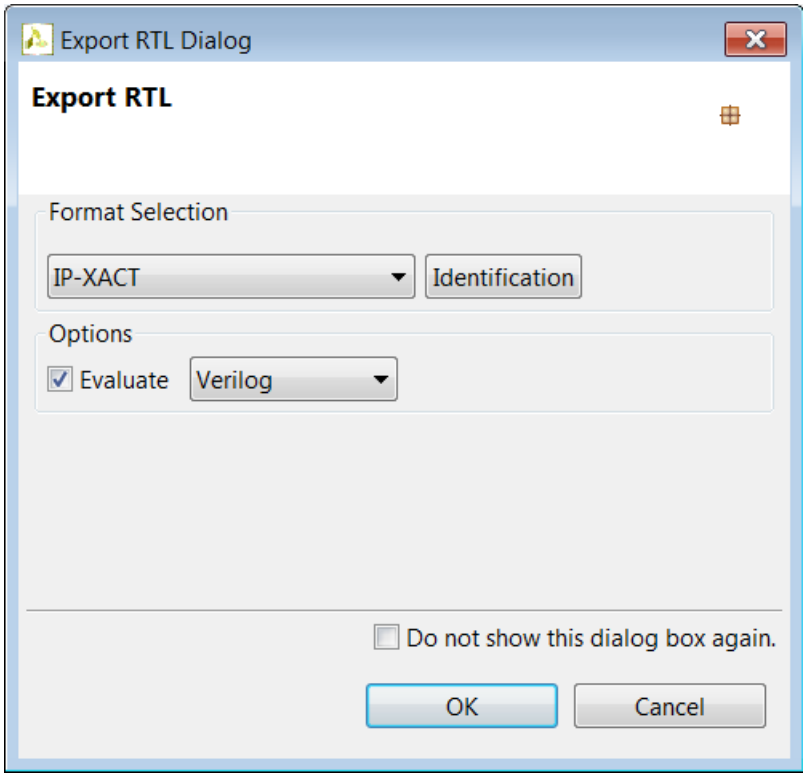


Figure 1-52: **Export RTL Menu**

2. In this example, the design will be exported to IP-XACT format. Refer to the *Vivado Design Suite User Guide: High-Level Synthesis (UG902)* for an explanation of all export formats and how to import them into the appropriate Xilinx design tool.

3. In this example RTL synthesis will be performed: select the evaluate option. For VHDL or Verilog, select from the drop-down menu. In this example, Verilog is used, as shown in Figure 1-52.
4. Click **OK**.

   Implementation starts.

The output files are written to fir.prj/solution5_throughput/impl.

- The IP-XACT IP is available in directory ip.
- The result of Verilog synthesis are in directory verilog.

When RTL synthesis completes, the RTL synthesis report automatically opens (see Figure 1-53).

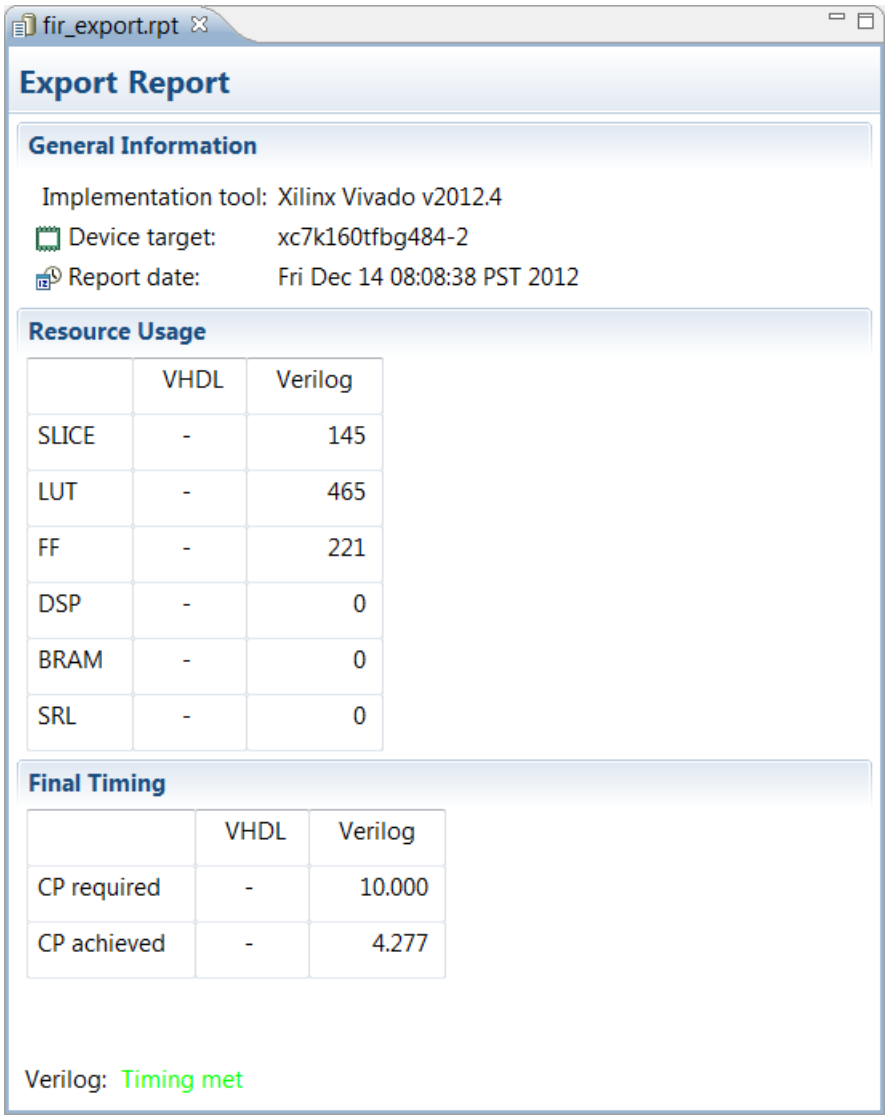


Figure 1-53: **solution5_throughput Report**

The report shows that the design is meeting timing. In some cases, logic synthesis might implement some logic operations, increasing the number of DSP48s and reducing the number of LUTs. Logic synthesis can also be able to decompose and reduce the number of multiplications, thereby reducing the number of DSP48s.

The Vivado HLS tool produces an RTL estimate of the resource. This evaluation step ensures the effects of logic synthesis can be checked while still inside the Vivado HLS tool.

Additionally, the results can be seen in the Console, as shown in Figure 1-54.

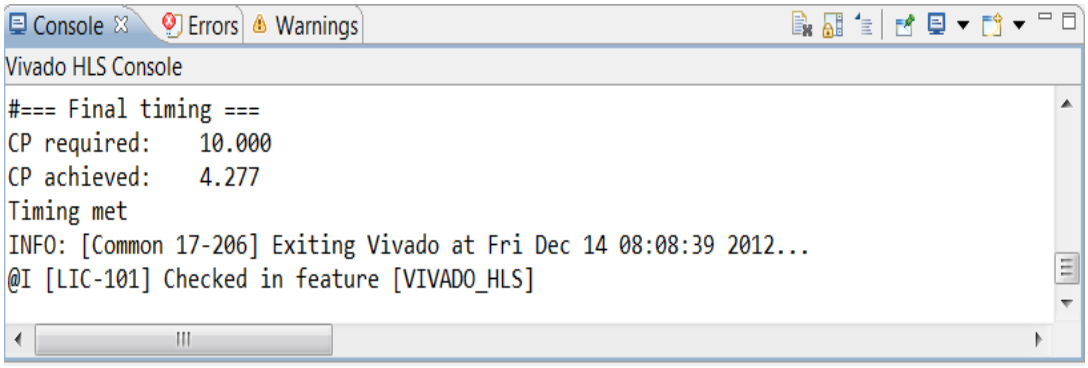


Figure 1-54: **Implementation Summary**

5. Exit the Vivado HLS tool using the menu. Select **File > Exit**.

When the project is reopened, all the results will still be present.

The other solutions can be verified and implemented in an identical manner. First select the solution in the Project Explorer and make it the active solution.

Summary

- The path to verification and implementation tool executables must be in the search path prior to execution from within the Vivado HLS tool. See the *Xilinx Design Tools: Installation and Licensing Guide (UG978)* for details.
  - This is not required for RTL SystemC verification.
- RTL verification does not require an RTL test bench be created.
- The RTL can be verified from within the Vivado HLS tool using the existing C test bench.
- The design can be can be exported as IP and the implementation evaluated using logic synthesis tools from within the Vivado HLS tool.

The Shell and Scripts

Everything which can be performed using the Vivado HLS GUI can also be implemented using Tcl scripts at the command prompt. This section gives an overview of using the Vivado HLS tool at the command prompt and how the GUI generated scripts can be copied and used.

Vivado HLS at the Shell

You can be invoked at the Linux or DOS shell prompt.

1. Invoke a DOS shell from the menu by selecting **Start > All Programs > Xilinx Design Tool > Vivado 2012.4 > Vivado HLS Command Prompt**.

   This ensures that the search paths for the Vivado HLS tool are already defined in the shell.

2. Type `$ vivado_hls` to invoke the GUI.

   It can also be invoked in interactive mode, and the `exit` command can be used to return to the shell.

```
$ vivado_hls -i
Vivado Hls> exit
$
```

The Vivado HLS tool can be run in batch mode using a Tcl script. When the script completes the Vivado HLS tool will remain in interactive mode and if the script has an `exit` command, it will exit and return to the shell.

```
$ vivado_hls -f fir.tcl
```

Additionally, once a project has been created it can be opened directly from the command line. In this example, `project fir.prj` is opened in the GUI:

```
$ vivado_hls -p fir.prj
```

This final option allows scripts to be run in batch mode and then the analysis to be performed using the GUI.

Creating a Script

When a project is created in the GUI, all the commands to re-create the project are provided in the `scripts.tcl` file in the solution directory.

To use the `script.tcl` file, copy it to a new location outside the project directory.

Example `script.tcl` file:

```
############################################################
## This file is generated automatically by Vivado HLS.
## Please DO NOT edit it.
## Copyright (C) 2012 Xilinx Inc. All rights reserved.
############################################################
open_project fir.prj
set_top fir
add_files fir.c -cflags "-DBIT_ACCURATE"
add_files -tb out.gold.8.dat
add_files -tb fir_test.c -cflags "   -DBIT_ACCURATE"
open_solution "solution5_throughput"
set_part  {xc7k160tfbg484-2}
create_clock -period 10
```

```
source "./fir.prj/solution5_throughput/directives.tcl"
csynth_design
```

If any directives where used in the solution, copy the `directives.tcl` file to a location outside the project directory and update the `script.tcl` file as shown, to use the local copy of directives.tcl.

```
############################################################
## This file is generated automatically by Vivado HLS.
## Please DO NOT edit it.
## Copyright (C) 2012 Xilinx Inc. All rights reserved.
############################################################
open_project fir.prj
set_top fir
add_files fir.c -cflags "-DBIT_ACCURATE"
add_files -tb out.gold.8.dat
add_files -tb fir_test.c -cflags "   -DBIT_ACCURATE"
open_solution "solution5_throughput"
set_part  {xc7k160tfbg484-2}
create_clock -period 10

source "./directives.tcl"
```

Example Scripts Directory

The FIR directory contains a scripts directory that has five scripts, used to create each of the five solutions in this tutorial.

Table 1-3: **Summary of Scripts**

Filename	Solution	Description
`run1_hls.tcl`	`solution1`	Creates the first solution, using standard implementation types.
`run2_hls.tcl`	`solution2`	Sets the macro to use Vivado HLS bit-accurate types.
`run3_hls.tcl`	`solution3`	The IO interfaces are defined.
`run4_hls.tcl`	`solution4_area`	Uses the directives from solution3 plus the `config_bind` command to force sharing of the multipliers.
`run5_hls.tcl`	`solution5_throughput`	Optimizations are applied to create a high-throughput version.

You can run these scripts to reproduce all the solutions in this tutorial. You can then open and analyze the project and solutions in the GUI.

Vivado HLS: Integrating EDK

Introduction

This document describes how to create an Embedded Developer Kit (EDK) Pcore with an AXI-LITE interface from the Vivado HLS high-level synthesis tool. It describes the necessary steps for integrating the generated Pcore with the MicroBlaze™ processor using the Xilinx® Platform Studio (XPS) Tool Suite.

The reference design has been verified on the Avnet Spartan®-6 LX9 MicroBoard, shown in Figure 2-1.

Figure 2-1: **Avnet Spartan-6 LX9 MicroBoard**

Software Requirements

The following software is required to test this reference design:

- Xilinx ISE® WebPACK with the EDK add-on, or ISE version 14.1 Embedded or System Edition
- Installed Silicon Labs CP210x USB-to-UART Bridge Driver (see *Silicon Labs CP210x USB-to-UART Setup Guide*, listed at http://em.avnet.com/s6microboard)
- Vivado™ Design Suite High-Level Synthesis (HLS) version 2011.4.2

Reference Design

The reference design consists of an EDK MicroBlaze processor with a custom Pcore generated from the Vivado HLS tool.

You can copy the reference design, AXI_Lite_Interface, from the `examples/tutorial` directory in the Vivado HLS installation area.

The MicroBlaze processor based design was created using the XPS Base System Builder (BSB). Figure 2-2, page 66 shows the final design created and provided with this document.

For information about using the XPS Base System Builder, refer to http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_2/platform_studio/ps_c_bsb_using_bsb.htm.

The reference design with the MicroBlaze processor runs the standalone board support package software with a simple C application that prompts you to enter values for each input variable and outputs the result.

Vivado HLS Pcore Functionality

The Vivado HLS Pcore functionality is an 8 bit adder. The focus of this document is the interface of the pcore to the MicroBlaze processor through the AXI-Lite interface, not the functionality of the pcore.

The Vivado HLS module has three variables: `A`, `B` and `C`. Of these, `A` and `B` are input variables, and `C` is an output variable. These three variables are mapped to three registers in the generated Pcore.

A Vivado HLS module has at least three control signals: `AP_START`, `AP_IDLE`, and `AP_DONE`. These signals are mapped to register in the generated Pcore.

The `AP_START` register is used to control the start of the Pcore and `AP_DONE` indicates when the module operation is done. A signal diagram (waveform of all three involved control signals) should be used to explain the handshaking mechanism.

Additional registers are present in the Pcore to support interrupts.

Block Diagram

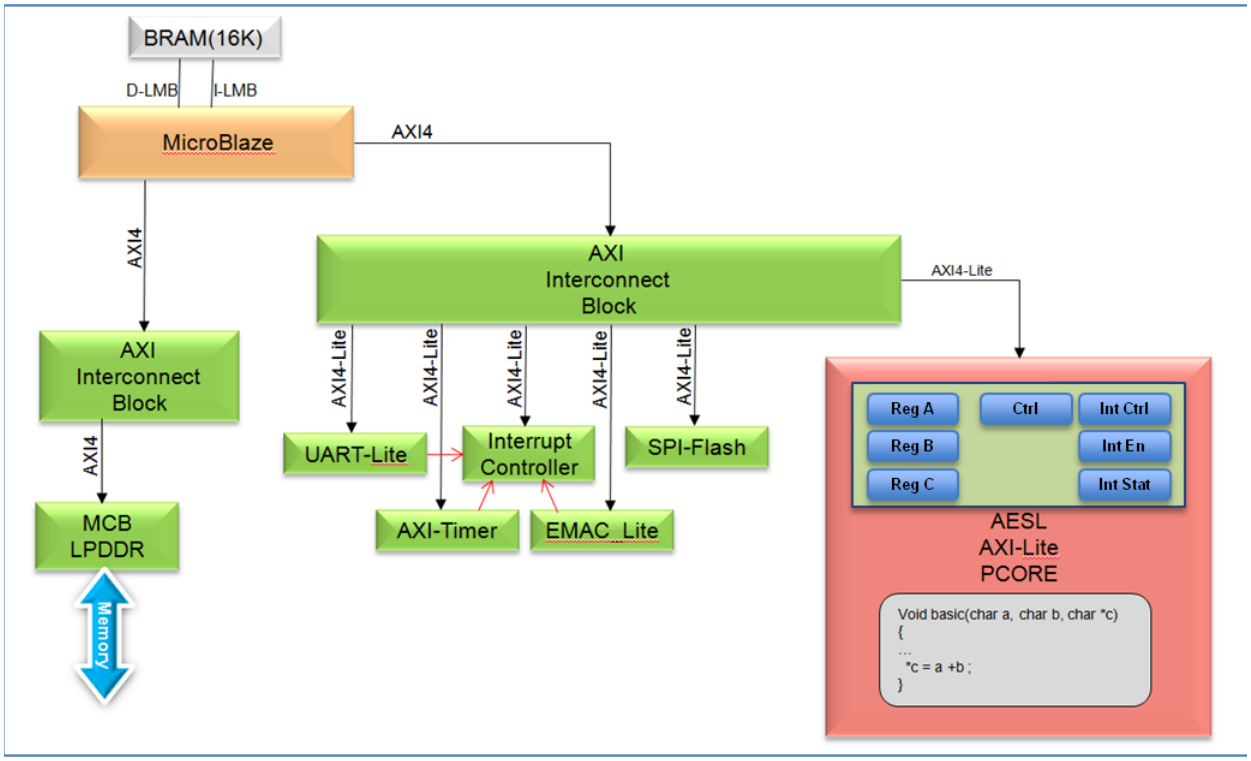


Figure 2-2: **Block Diagram**

The complete architecture consists of:

- MicroBlaze processor
- 16K of Block RAM to run code for the MicroBlaze processor
- Custom Pcore created using the Vivado HLS tool
- UART used for communication with the MicroBlaze processor
- Two AXI-Interconnects

In addition, it includes the following, which are part of the design but are not used in this demo:

- Interrupt controller
- LPDDR
- AXI-Timer
- SPI-Flash
- Ethernet-Lite

Creating EDK Pcore with AXI-LITE

Opening the Vivado HLS Project File

To create the AXI-Lite interface Pcore, the first step is to open the Vivado HLS Project `basic.prj`.

1. Start Vivado HLS.
2. Select **Open Project**.
3. Select `basic.prj`.

Refer to Chapter 1, Vivado HLS: Introduction Tutorial for details about how to create an Vivado HLS project.

Note: Figure 2-3, page 68 shows the C code with explanation.

Generating Pcores Using Vivado HLS

To generate Pcores using Vivado HLS, the header file `ap_interfaces.h` must be included. This header file is a convenient way to define macros that apply standard Vivado HLS directives as pragmas.

The example makes use of the `AP_INTERFACE_REG_AXI4_LITE` and the `AP_CONTROL_BUS_AXI` macros.

The `AP_INTERFACE_REG_AXI4_LITE` macro defines that the three function arguments (`a`, `b`, and `c`) be implemented as registers that are accessed through an AXI4-Lite interface.

- Each port is specified as being in group `BUS_A`. This means they are all grouped into the same AXI4 Lite interface called `BUS_A`.
- The RTL interface is set to type `ap_none`. This means that the RTL implementation only has data ports; there are no associated acknowledge or valid signals with each data port and therefore no associated register in the interface.

The `AP_CONTROL_BUS_AXI` macro adds the block level IO protocol signals to an AXI4-Lite interface.

- The control signals `AP_START`, `AP_DONE`, and `AP_IDLE` are created by default when Vivado HLS synthesizes the top-level function. The default function interface is `ap_ctrl_hs`.
- Specifying the name `BUS_A` ensures that these signals are grouped into the same AXI4 Lite interface as the other ports.

Table 2-1, page 72 describes all the registers created by Vivado HLS for the generated Pcore.

Creating EDK Pcores

The steps to create the EDK Pcore with the AXI-Lite interface are:

1. Open the Vivado HLS project `basic.prj`.

   The project code is shown in Figure 2-3. Refer to *Vivado HLS Tutorial: Introduction (UG871)*.

basic.cpp

```
45 ******************************************************************************/
46 #include "basic.h"
47
48 void basic(char a, char b, char *c) {
49
50 // Disable any IO protocol on ports (set to ap_none)
51 // And register ports
52 #pragma HLS INTERFACE ap_none register port=a
53 #pragma HLS INTERFACE ap_none register port=b
54 #pragma HLS INTERFACE ap_none register port=c
55
56 // Specify all ports as an AXI4 Lite Interface
57 // - include the block level controls (associated with the return port)
58 // - use name BUS_A to ensure all are grouped into the same interface
59 #pragma HLS RESOURCE variable=a core=AXI4LiteS metadata="-bus_bundle BUS_A"
60 #pragma HLS RESOURCE variable=b core=AXI4LiteS metadata="-bus_bundle BUS_A"
61 #pragma HLS RESOURCE variable=c core=AXI4LiteS metadata="-bus_bundle BUS_A"
62 #pragma HLS RESOURCE variable=return core=AXI4LiteS metadata="-bus_bundle BUS_A"
63
64   *c = a + b;
65
66 }
67
```

Figure 2-3: **C Code**

2. Click the **Synthesis** button, shown in Figure 2-4.

Figure 2-4: **Synthesis Button**

3. Click the **Export RTL** button, shown in Figure 2-5.

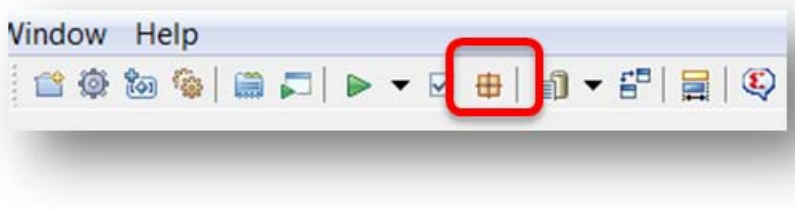


Figure 2-5: **Export RTL Button**

The RTL Implementation dialog box opens.

4. Select **OK**.

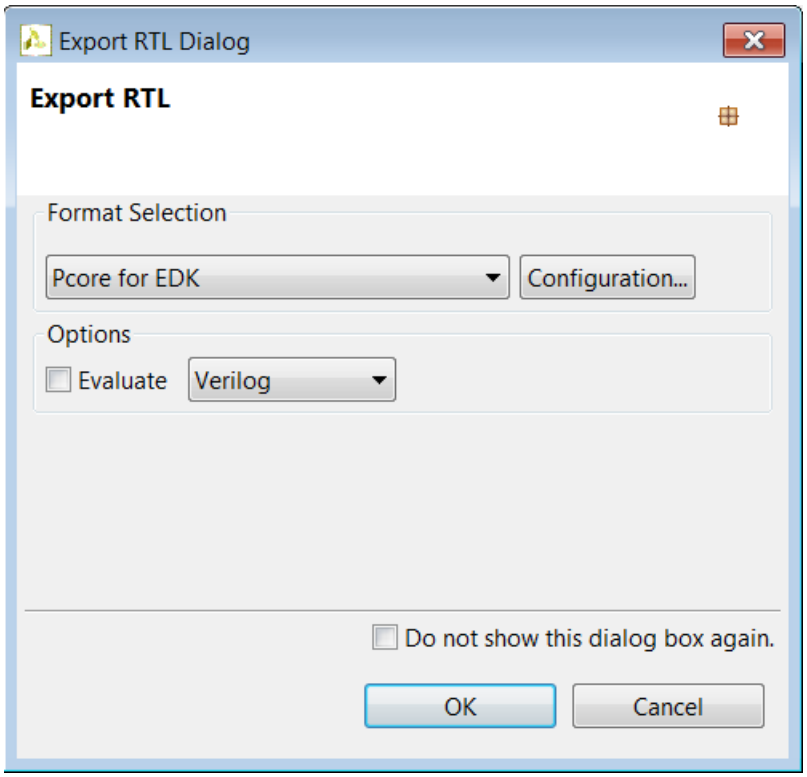


Figure 2-6: **Export RTL Dialog Box**

The generated Pcore is located in the `/impl` directory of the selected solution, as shown in Figure 2-7.

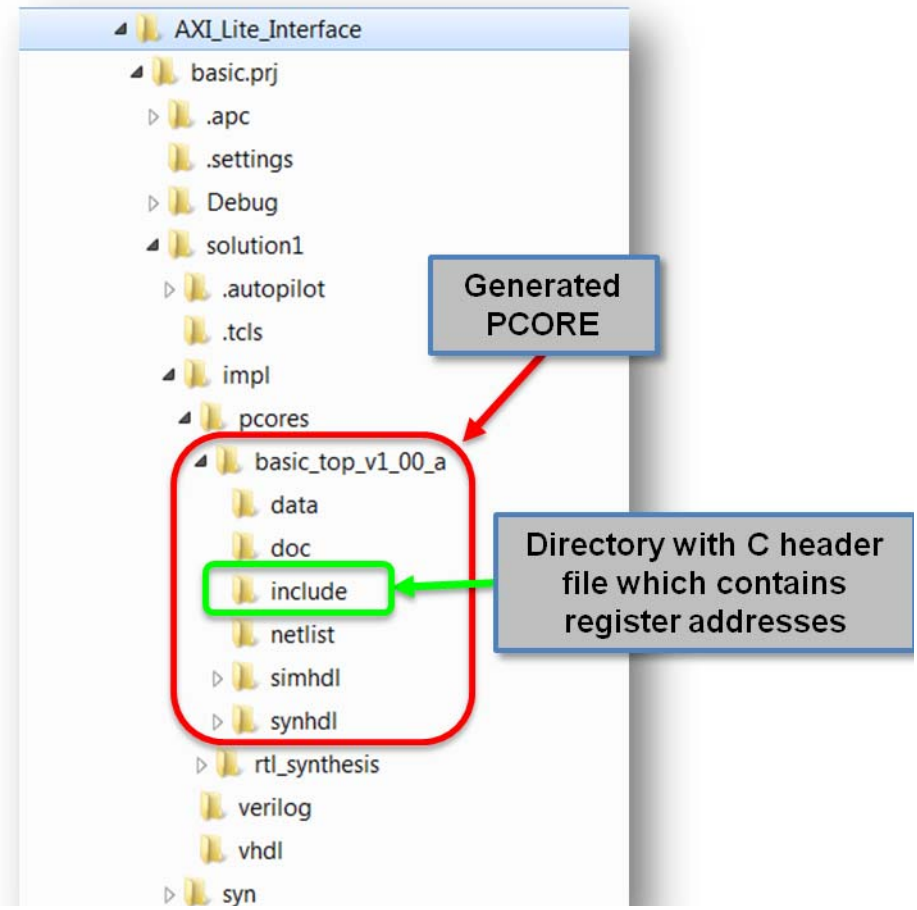


Figure 2-7: **Generated Pcore Location**

Pcore Register List

As stated in Vivado HLS Pcore Functionality, the Pcore has seven registers. Three registers represent the passed-in arguments (A, B, and C) of the C code. The other four registers represent the control register for AP control signals AP_START, AP_DONE, and AP_IDLE, and three interrupt control registers.

Table 2-1: **Pcore Registers**

Register Name	Width	R/W	Default Value	Address offset	Description
Control	3	R/W	0	0x00	Bit 0 - ap_start (Read/Write/SC) Bit 1 - ap_done (Read/COR) Bit 2 - ap_idle (Read) SC = Self Clear, COR = Clear on Read
Global Interrupt Control	1	R/W	0	0x04	Bit 0 - Enable all interrupts.
Interrupt enable Register	1	R/W	0	0x08	Bit 0 - ap_done signal.
Interrupt Status Register	1	R/W	0	0x0c	Bit 0 - ap_done signal (Read/TOW) TOW = Toggle on Write
A	8	R/W	0	0x14	Variable A.
B	8	R/W	0	0x1c	Variable B.
C	8	R/W	0	0x24	Variable C.

Integrating Generated Pcores

To integrate the generated Pcore with the MicroBlaze Processor using XPS:

1. Copy the generated Pcore from Vivado HLS directory structure to XPS directory structure, as shown in Figure 2-8.

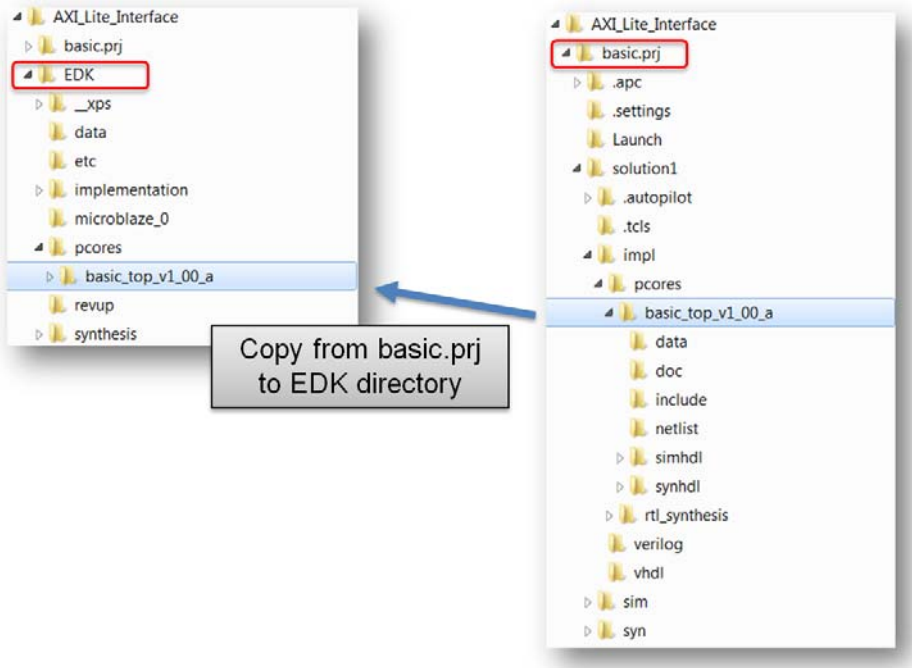


Figure 2-8: **XPS and Vivado HLS Directory Structures**

2. In XPS, add the generated Pcore from the IP catalog by clicking **Pcore**. The new Pcore appears under the `Project Local PCores` directory, as shown in Figure 2-9, page 74.
3. When the connection dialog box opens, accept the option to connect to instance `microblaze_0`.

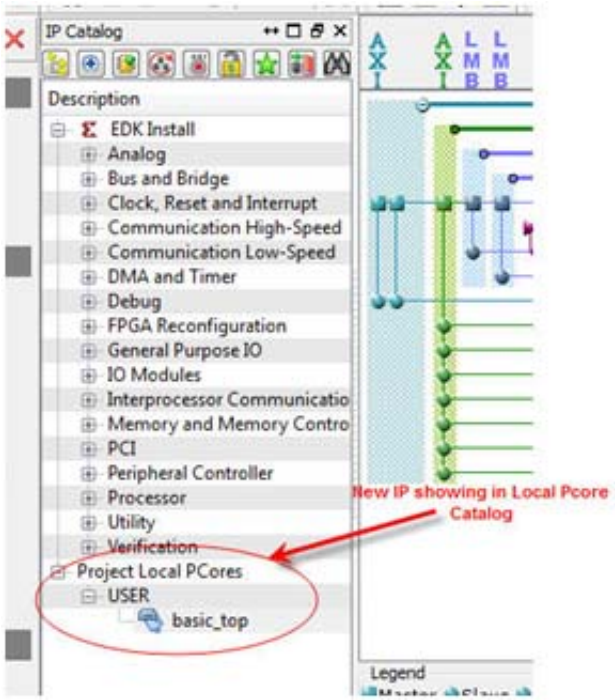


Figure 2-9: **Add Generated Pcore**

4. If **Pcore** is not listed under the `Project Local PCores` directory, then you must direct XPS to rescan the user repository.

   Select **Project > Rescan User Repositories**.

5. Change the default Reset Polarity of the generated pcore from Active Low by doing the following:

   a. Double-click the Pcore to customize it, as shown in Figure 2-10.

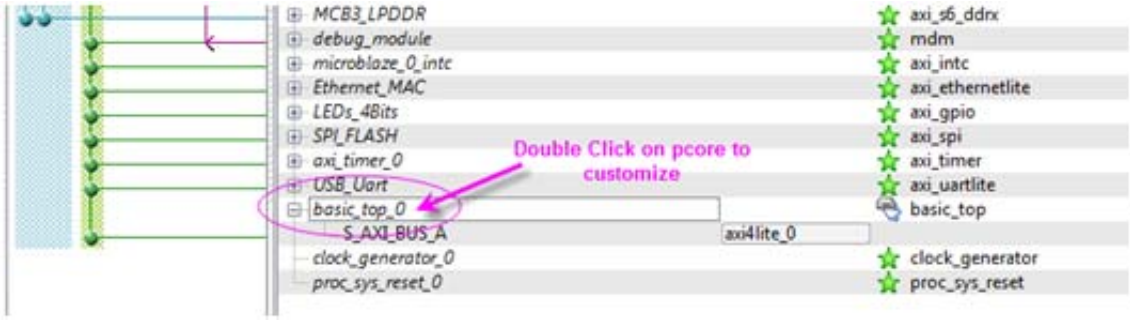


Figure 2-10: **Customize Pcore**

   b. Deselect the RESET_ACTIVE_LOW signal check box, as shown in Figure 2-11.

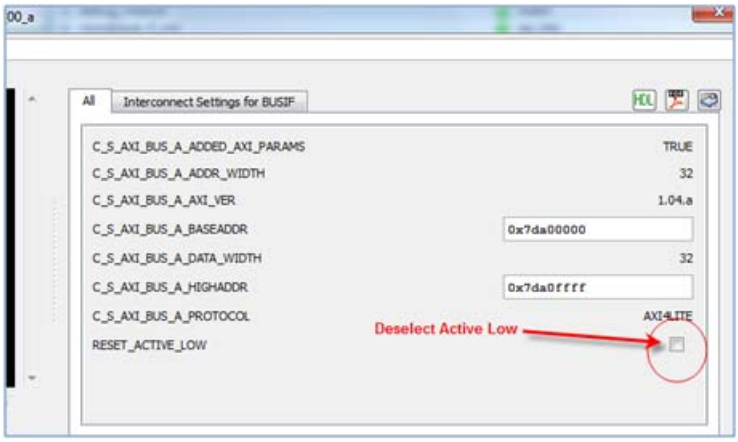


Figure 2-11: **Active Low Signal Check Box**

6. Set the Pcore base address.

   You can set the base address from different locations. One location is in the Pcore customization window, shown in Figure 2-11.

   The other location is the Addresses tab in the Assembly window, shown in Figure 2-12. This option allows you to view the full memory map for the MicroBlaze processor, which prevents memory overlap errors.

   Note: You can also automatically generate addresses in the Assembly window.

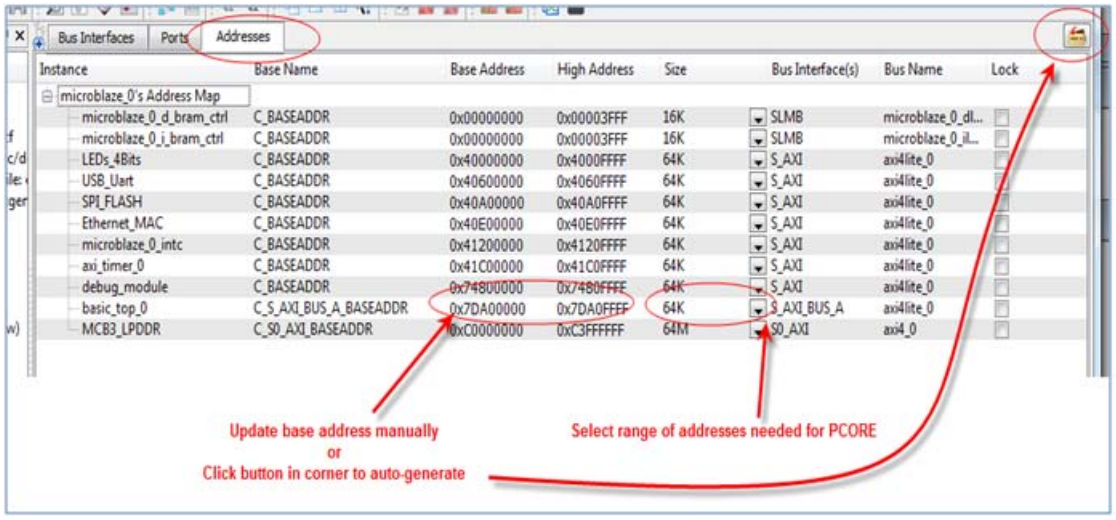


Figure 2-12: **Set Pcore Base Address**

7. Connect the Pcore to AXI Interconnect in the Bus Interfaces tab, as shown in Figure 2-13.

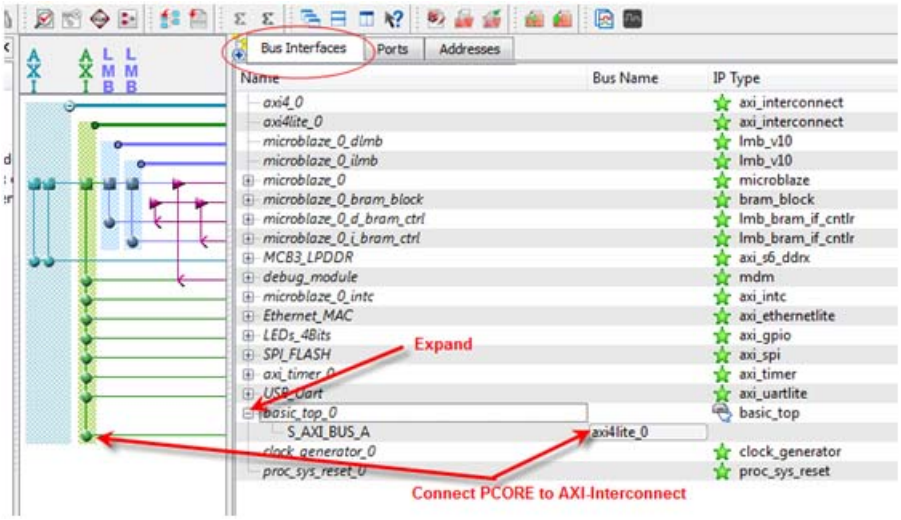


Figure 2-13: **Connect Pcore to AXI-Interconnect**

8. Connect clocks and rest signals.
9. Change to the Ports tab to see the other ports that needs connections (see Figure 2-14).

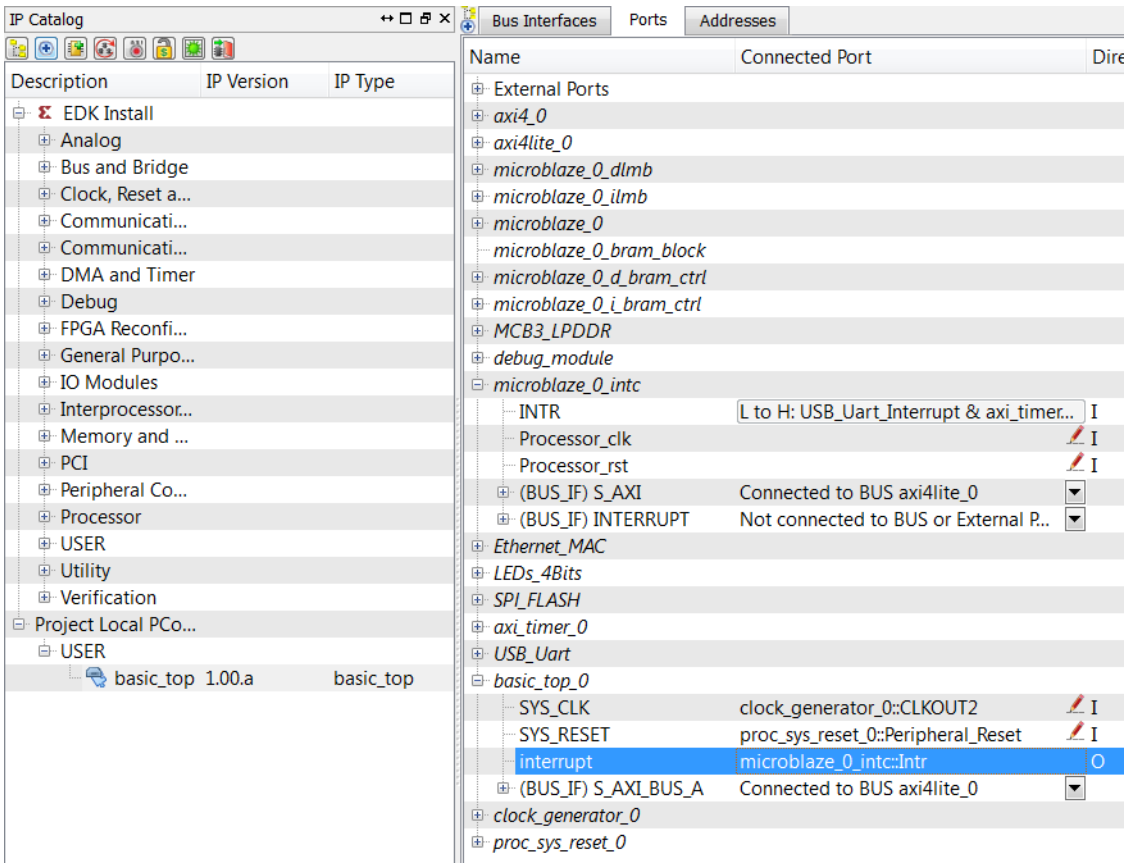


Figure 2-14: **Connect Clocks and Rest Signals**

10. As shown in Figure 2-14, on instance, basic_top_0:
    a. Connect port `SYS_CLK` to `clock_generator_0: CLKOUT2`.
    b. Connect port `SYS_RST` to `proc_sys_reset_0: Peripheral_Reset`.
    c. Add port interrupt `basic_top_0` to the list of connected interrupts.
    d. Confirm that port (BUS_IF) is connected to BUS `axi4lite_0`.

Generating the FPGA Bitstream

A software application image is needed to initialize the BRAM. The MicroBlaze processor runs the software application after reset.

1. Refer to Creating Application Software for steps on creating an ELF file. In this example, the application software has already been compiled into the `hello_world_0.elf` file.

   Figure 2-15 shows how to select the ELF file to initialize the BRAM.

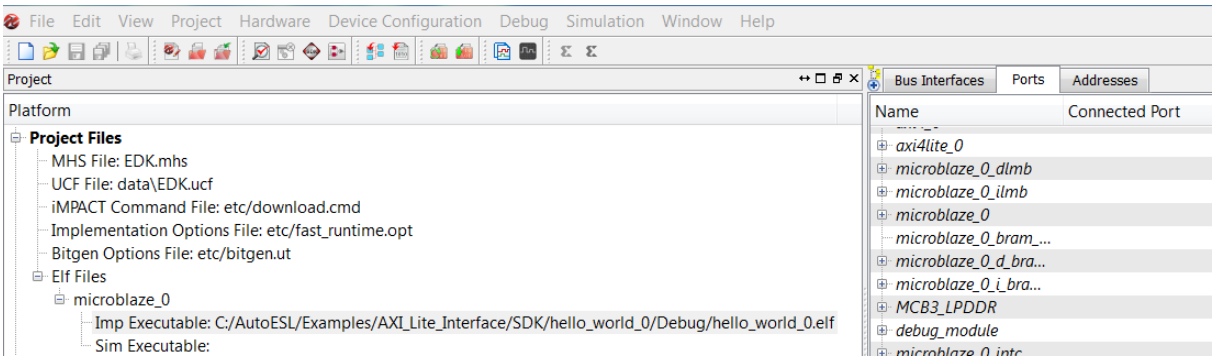


Figure 2-15: **Select ELF File**

2. Select **Device Configuration > Update Bitstream** to generate the bitstream.

   This performs two steps in serial:

   a. Generates the FPGA bitstream `<project_name>.bit` in the implementation directory.
   b. Initializes the BRAM with the ELF file selected and generates a `download.bit` file in the `/implementation` directory.

Controlling the Generated Pcore

The generated Pcore has six registers accessible by the MicroBlaze processor through the AXI-Lite interface. C code is needed to read and write with these registers, as shown in Figure 2-16.

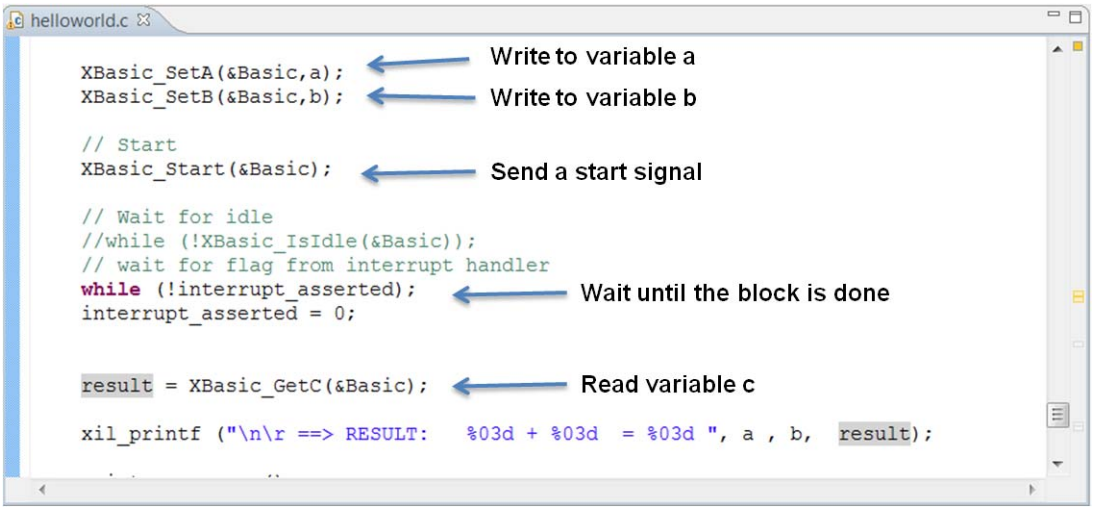


Figure 2-16: **C Code to Read and Write with Registers**

The Vivado HLS pcore provides C functions that allows the ports to be accessed. In this example, these functions are in `xbasic.c` and header file `xbasic.h`. Header file `xbasic_BUS_A.h` creates some useful macros.

Creating Application Software

The Xilinx Software Development Kit (SDK) is a software development environment to create and debug software applications. Features include project management, multiple build configurations, a feature rich C/C++ code editor, error navigation, a debugging and profiling environment, and source code version control. For more SDK information, refer to http://www.xilinx.com/tools/sdk.htm.

The steps for creating application software using SDK are:

1. Select **Project > Export Hardware Design to SDK**.

   Exporting the hardware description of the system from XPS to SDK enables it to create software application images for that system.

2. From the Export to SDK window, click **Export & Launch SDK**.

   Note: This can take several minutes to complete.

3. In the Workspace Launcher dialog box, use the **Browse** button to select a directory location for your workspace and click **OK**, as shown in Figure 2-17.

CAUTION! *Make sure that the path name does not contain spaces.*

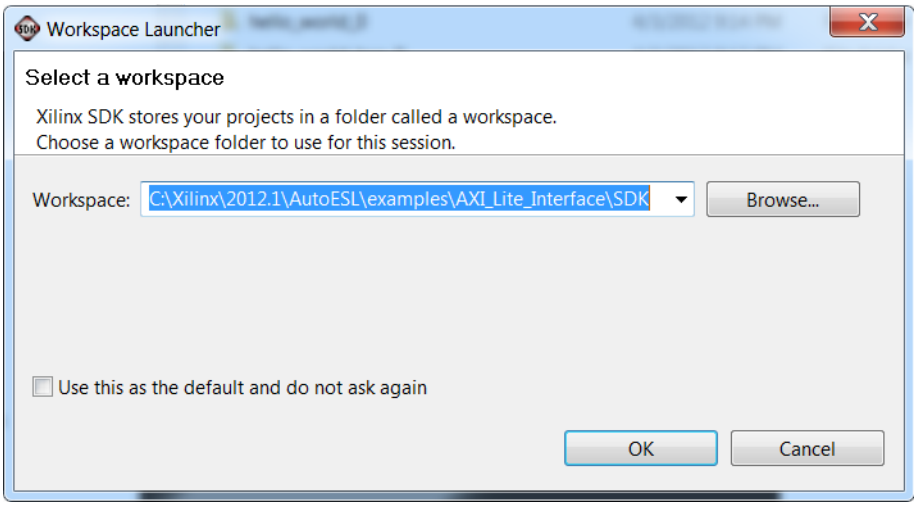


Figure 2-17: **Workspace Launcher Dialog Box**

4. To create a new C project, select **File > New > Xilinx C Project**.
5. Select the **Hello World** application as a starting point from the project templates, as shown in Figure 2-18.
6. Click **Next**.

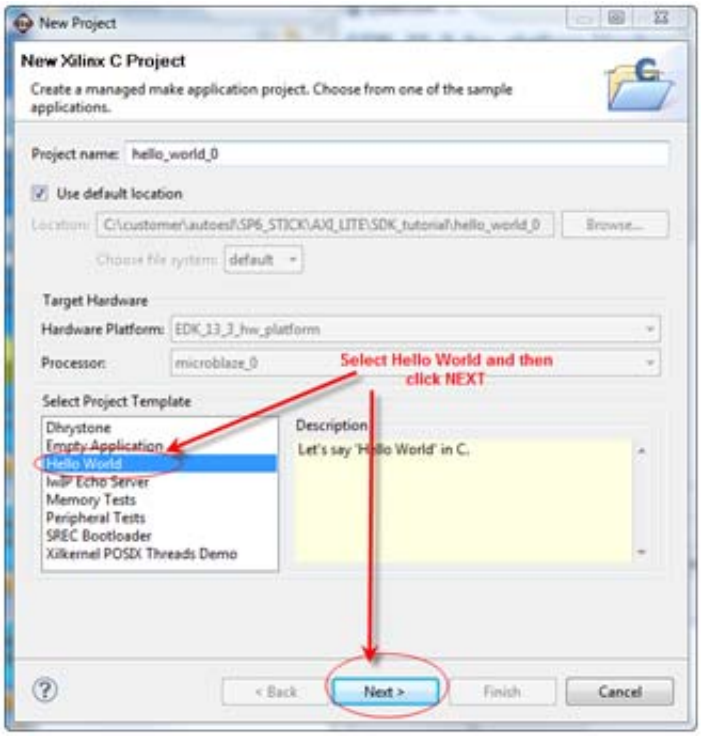


Figure 2-18: **Hello World Template**

7. Click **Finish**.

   The application automatically starts building and creating an ELF file.

   The ELF file is the compiled application and is created in the `/Debug` directory (see Figure 2-19), with the application name and the `.elf` extension. In this example, the file is `hello_world_0.elf`.

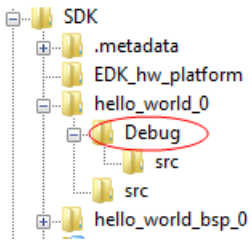


Figure 2-19: **ELF File Location**

8. Edit the `Helloworld.c` file and add code to test the generated Pcore.

   Include the C files from the Vivado HLS pcore sub-directory include (`basic_top_v1_00_a\include`), as shown in figure Figure 2-20.

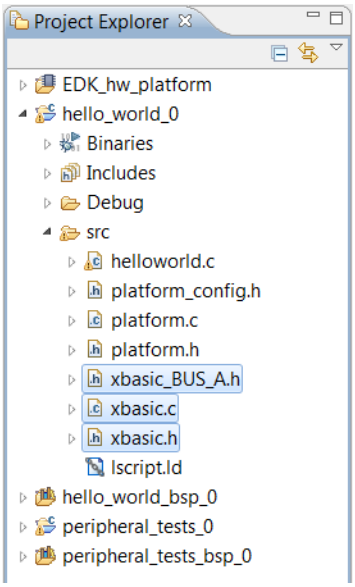


Figure 2-20: **C Files in hello_world_0/src Directory**

9. Open the `helloworld.c` for editing by double clicking on `helloworld.c` after expanding the `hello_world_0` application and the `/src` directory.

10. In the editor, change the code in `helloworld.c` to match the code below. Refer to the C code SDK\hello_world_0\src\helloword.c in the tutorial directory.

```
#include <stdio.h>
#include "platform.h"
#include "xparameters.h"
#include "xil_io.h"
#include "xstatus.h"
#include "xbasic.h"  // DM added
#include "xintc.h"
#include "xil_exception.h"
#include "xuartlite_l.h"

#define pritnf xil_printf
void print(char *str);



// BASIC Pcore SETUP
XBasic Basic;
XBasic_Config Basic_Config =
{
    0,
    XPAR_BASIC_TOP_0_S_AXI_BUS_A_BASEADDR
};

int SetupBasic(void)
{
    return XBasic_Initialize(&Basic, &Basic_Config);
}

//-------------- Setup Interrupt control -----------------------------
//
#define INTC_DEVICE_ID          XPAR_INTC_0_DEVICE_ID
#define XBASIC_INTERRUPT_ID   XPAR_MICROBLAZE_0_INTC_BASIC_TOP_0_INTERRUPT_INTR
XIntc InterruptController;  /* The instance of the Interrupt Controller */

int interrupt_count = 0;   // just for statiscs
int interrupt_asserted = 0;

void XBasic_InterruptHandler(void *InstancePtr)
{

      interrupt_count++;
      // clear the interrupt
      XBasic_InterruptClear(&Basic, 1);
      // poor man semaphore
      interrupt_asserted = 1;

}


//-----------------------------------------------------
int SetupInterrupt(void)
{
      int Status;

    // Initialize the interrupt controller driver so that it is ready to use.
```

```
    Status = XIntc_Initialize(&InterruptController, INTC_DEVICE_ID);
    if (Status != XST_SUCCESS)
    {
          return XST_FAILURE;
    }

    // Connect a device driver handler that is called when an interrupt
    // for the device occurs, the device driver handler performs the specific
    // interrupt processing for the device
    Status = XIntc_Connect
                  ( &InterruptController, XBASIC_INTERRUPT_ID,
                     (XInterruptHandler)XBasic_InterruptHandler,
                     NULL
                  );
    if (Status != XST_SUCCESS)
    {
          return XST_FAILURE;
    }

    // Start the interrupt controller such that interrupts are enabled for
    // all devices that cause interrupts, specific real mode so that
    // the timer counter can cause interrupts thru the interrupt controller.
    //
    Status = XIntc_Start(&InterruptController, XIN_REAL_MODE);
    if (Status != XST_SUCCESS)  { return XST_FAILURE; }

    // Enable the interrupt for the AESL BASIC CORE
    XIntc_Enable(&InterruptController, XBASIC_INTERRUPT_ID);



  //  Initialize the exception table.
    Xil_ExceptionInit();

    // Register the interrupt controller handler with the exception table.
    Xil_ExceptionRegisterHandler(
                XIL_EXCEPTION_ID_INT,
                (Xil_ExceptionHandler) XIntc_InterruptHandler,
                &InterruptController
                );

    // Enable non-critical exceptions.
    Xil_ExceptionEnable();

  XBasic_InterruptEnable(&Basic, 1);
  XBasic_InterruptGlobalEnable(&Basic);

    return XST_SUCCESS;
}



void print_core_regs(void )
{

    xil_printf ("\n\r   A       reg [0x%08x] ", XBasic_GetA(&Basic ) );
    xil_printf ("\n\r   B       reg [0x%08x] ", XBasic_GetB(&Basic ));
    xil_printf ("\n\r   C       reg [0x%08x] ", XBasic_GetC(&Basic ) );
    xil_printf ("\n\r   DONE    reg [0x%08x] ", XBasic_IsDone(&Basic) );
```

```
      xil_printf ("\n\r   IDLE    reg [0x%08x] ", XBasic_IsIdle(&Basic) );

      xil_printf ("\n\r   INT STATS   [%d]     ", interrupt_count );

}

int ReadInt(int size)
{
  int value=0;
  char c ='0';
  int i;
  for (i=0; i <size; i++)
  {
      c=inbyte();
    if (c==' ' )
    {
       c='0';
       outbyte(c);
    }
    else if (c=='\n')
    {
      break;
      return value;
    }
    else if (c=='\r')
    {
      break;
      return value;
    }
    else
    {
      outbyte(c);
      value=value*10+c-'0';
    }
  }

  return value;
}


int main()
{
      //init_platform();

    int a = 1000;
    int b = 1000;
    u32 result;

    // initialize AESL Pcore

    int status;
    status = XBasic_Initialize(&Basic, &Basic_Config);
    if (status != XST_SUCCESS) {
        xil_printf("\n\r ==> Basic failed.\n\r");
    } else {
        xil_printf("\n\r ==> Basic succeeded.\n\r");
    }
```

```
    // Initialize the interrupts (local then global)
    status = SetupInterrupt();
    if (status != XST_SUCCESS) {
        xil_printf("\n\r ==> SetupInterrupt failed.\n\r\n\r");
    } else {
        xil_printf("\n\r ==> SetupInterrupt succeeded.\n\r\n\r");
   }


    // Get and setup the data
    while (1)
    {
    print("\n\r ============================================================");
    print("\n\r =========== START OF AESL BASIC CORE TEST =================");
    print("\n\r ================      RESULT = A + B     ==================\n\r");

    while (a > 255)
    {
      print("\n\r --> Please enter number between (0-255) for variable A : ");
      a = ReadInt(3);
    }

    while (b > 255)
    {
      print("\n\r --> Please enter number between (0-255) for variable B : ");
       b = ReadInt(3);
     }

    XBasic_SetA(&Basic,a);
    XBasic_SetB(&Basic,b);

    // Start
    XBasic_Start(&Basic);

    // Wait for idle
    //while (!XBasic_IsIdle(&Basic));
    // wait for flag from interrupt handler
    while (!interrupt_asserted);
    interrupt_asserted = 0;


    result = XBasic_GetC(&Basic);

    xil_printf ("\n\r ==> RESULT:   %03d + %03d  = %03d ", a , b,  result);

    print_core_regs();

    print("\n\r ============================================================");
    print("\n\r ============================================================\n\r");
    // reset variable to value bigger then 255 to prompt user for new input
    a =1000;
    b =1000;

    }

    //cleanup_platform();

    return 0;
}
```

Running the Demo on the Avnet MicroBoard

Setup Requirements

For this demo, you must have the following:

- Avnet MicroBoard
- Two USB cables connected to UART and JTAG ports of the Avnet MicroBoard and to the PC, as shown in Figure 2-21

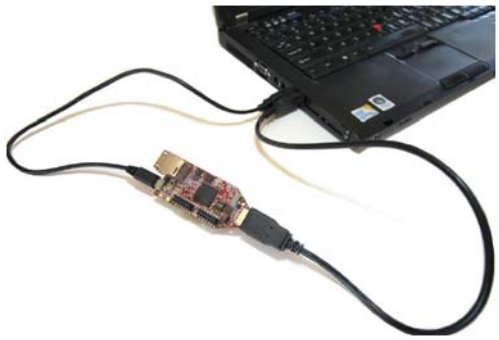


Figure 2-21: **Cable Connections**

- Hyperterminal (Tera Term) with the serial port setup shown in Figure 2-22

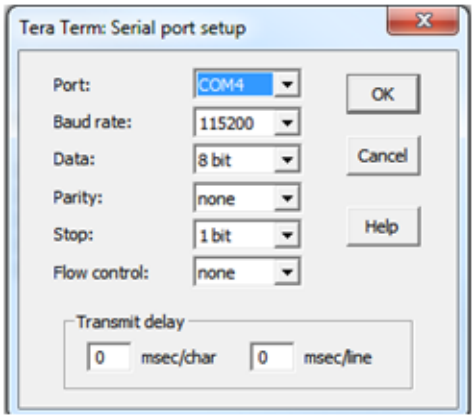


Figure 2-22: **Serial Port Set Up**

- `Download.bit` file provided with the reference design

Finding Serial Port Number on Windows 7 PC

1. Click the **Windows Start** button.
2. In the **Search programs and files** box, type **`devmgmt.msc`**.

   Windows lists `devmgmt.msc` in the search results window.
3. Select **devmgmt.msc** and when Windows asks for permission, click **Yes**.
4. When the Device Manager window appears, expand **Ports (COM & LPT)** to find the USB to UART COM port number, as shown in Figure 2-23.

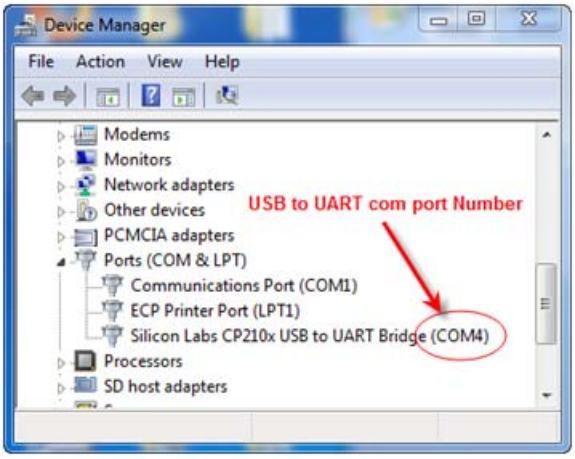


Figure 2-23: **USB to UART COM Port Number**

Running the Demo

1. Open hyperterminal (Tera Term) with the settings shown in Figure 2-22.
2. Download the bit file from XPS by selecting **Device Configuration > Download Bitstream**.
3. After the FPGA is configured, you are prompted to provide values for `A` and `B`. Figure 2-24 shows an example output of the application.

```
============================================================
========== START OF AESL BASIC CORE TEST ==================
===============      RESULT = A + B     ===================

--> Please enter number between (0-255) for variable A : 5
--> Please enter number between (0-255) for variable B : 10
==> RESULT:   005 + 010  = 015
  A       reg [0x00000005]
  B       reg [0x0000000A]
  C       reg [0x0000000F]
  START   reg [0x00000000]
  DONE    reg [0x00000000]
  IDLE    reg [0x00000001]
============================================================
============================================================
```

Figure 2-24: **Application Output**

Running Bus Functional Model Simulation

Runing bus functional model simulation on a generated Pcore requires the following steps:

1. Adding Pcores to an XPS Project
2. Adding CLOCK and RESET Connections
3. Generating the Simulation Model
4. Running the Simulation

Adding Pcores to an XPS Project

1. Start XPS.
2. In the Welcome Window, select **Create New Blank Project**.
3. Change the target device to match the FPGA on your board.
4. Unselect both check boxes under Auto Instantiate Clock/Reset, **AXI Clock Generator** and **AXI Reset Module**.
5. Click **OK**.

   The Spartan-6 LX9 part is selected in the following example.

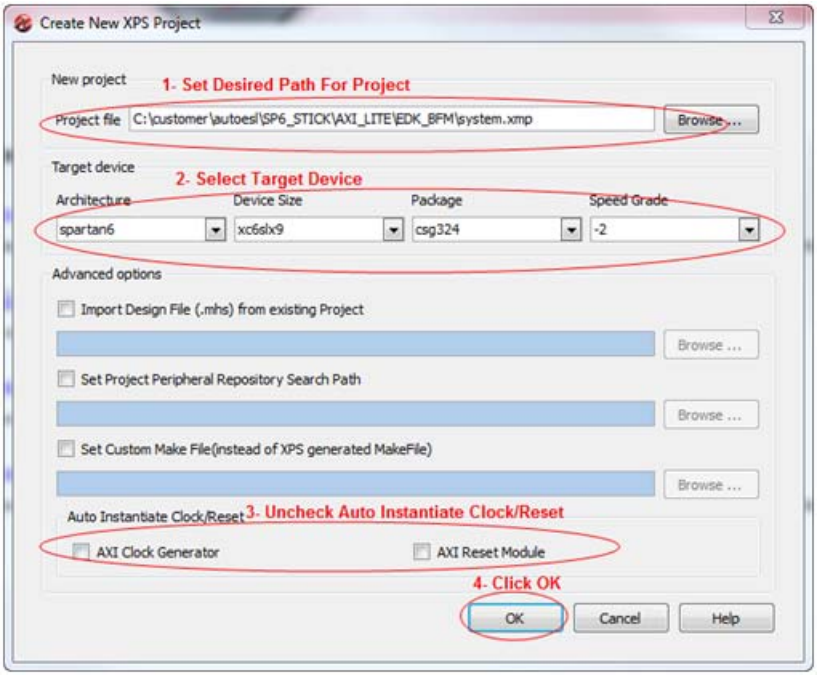


Figure 2-25: **Create New XPS Project Dialog Box**

6. Copy the generated Pcore from Vivado HLS directory structure to XPS directory structure.

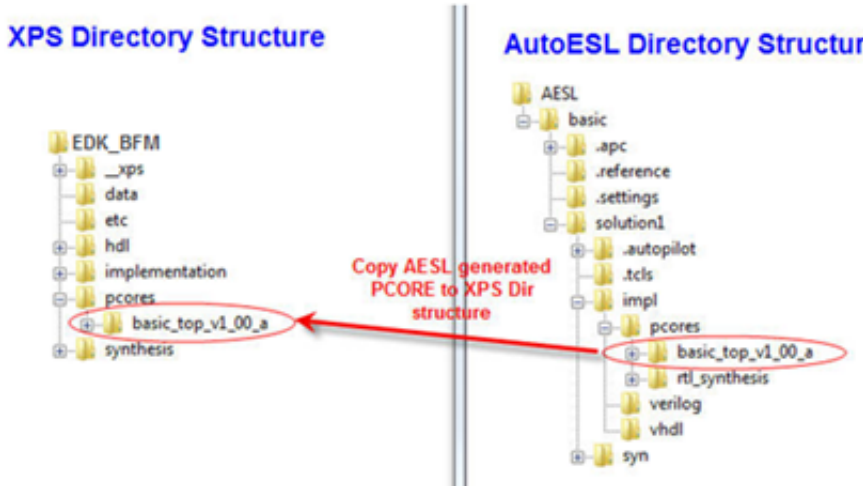


Figure 2-26: **Vivado HLS Directory Structure**

7. In the IP catalog, double-click each of the following pcores to add it into the blank XPS project:
   - AXI Interconnect
   - Basic_top
   - AXI4 Lite Master BFM
   - Pcore generated from Vivado HLS
   - Pcore used to simulate an AXI LITE Master (for example, processor)

   Note: If Pcore is not listed under the Project Local Pcores, select **Project > Rescan User Repository** to have XPS rescan the user repository.
8. Double-click **AXI Interconnect** and click **YES** when prompted to add the Pcore to your design.
9. When prompted to customize the Pcore, click **OK** in the XPS Core Config dialog box to accept the default settings.

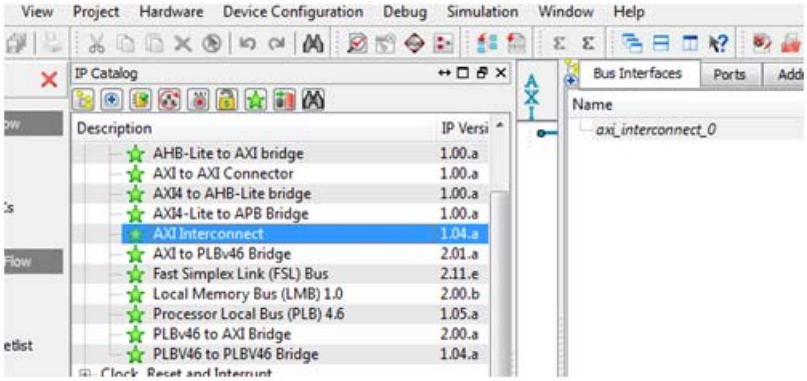


Figure 2-27: **Pcore Added to IP Catalog**

10. Double-click the AXI LIte Master BFM to add it.
11. Click **YES** when prompted to add the Pcore.
12. When prompted to customize the Pcore, rename the component instance name to **`bfm_processor`**.
13. Click **OK**.

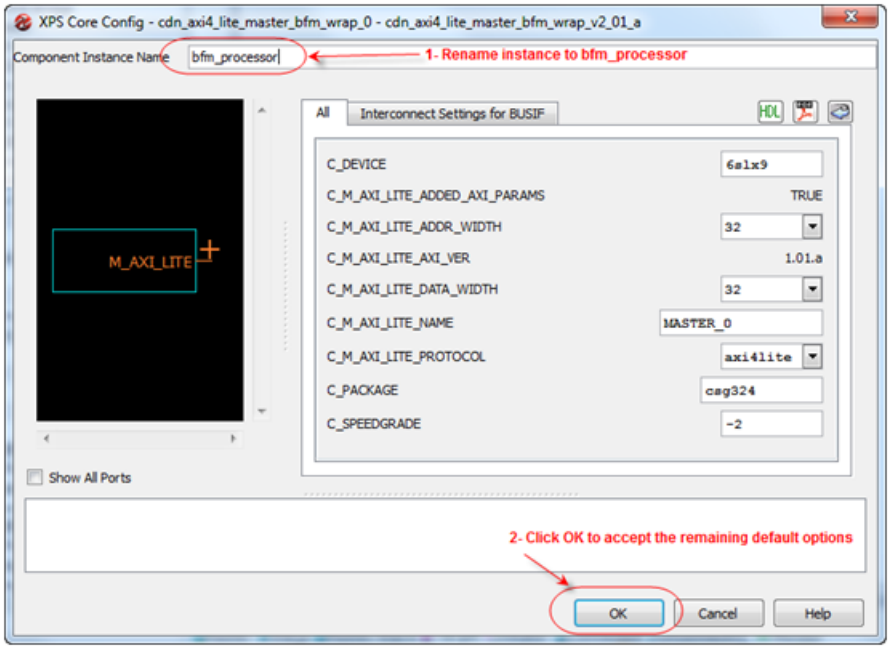


Figure 2-28: **XPS Core Config**

14. Connect `bfm_processor` to the AXI interconnect, as shown in Figure 2-29.

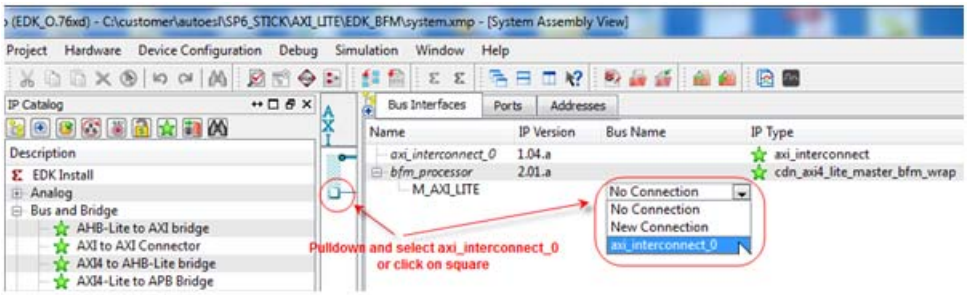


Figure 2-29: **bfm_processor Connection**

15. Double-click the `basic_top` Pcore to add it.
16. Click **YES** when prompted to add the Pcore.
17. When prompted to customize the Pcore, change `C_S_AXI_BUS_A_BASEADDR` to **`0x00000000`** and `C_S_AXI_BUS_` to **`0x00000FFF`**.
18. Click **OK**.

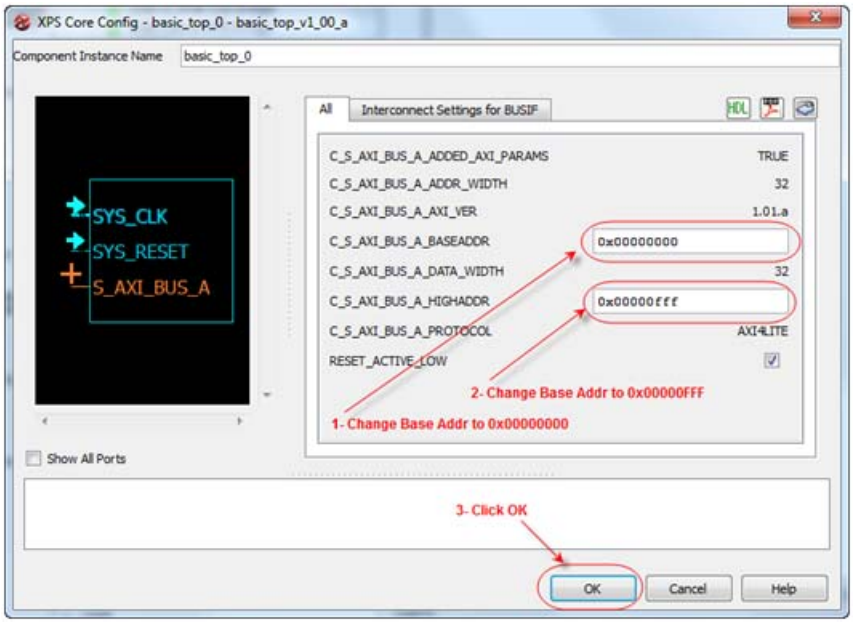


Figure 2-30: **XPS Core Config Dialog Box**

19. Connect `basic_top_0` to the AXI interconnect as shown in Figure 2-31.

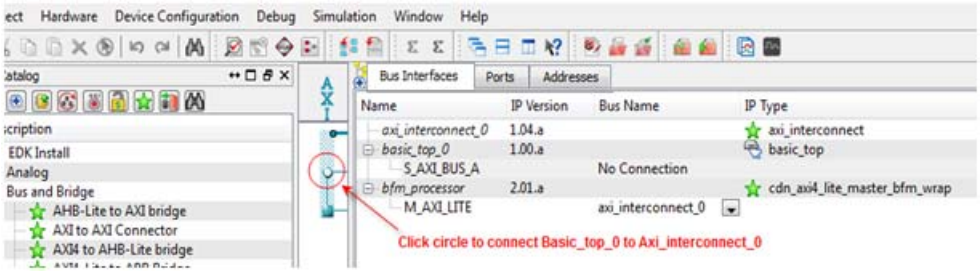


Figure 2-31: **AXI Interconnect**

Adding CLOCK and RESET Connections

The next step is to add the CLOCK and RESET connections for Pcores. To do this, you must manually edit the Microprocessor Hardware Specification (MHS) file.

1. Double-click the `system.mhs` file.

   It opens in the text editor in the XPS main window.

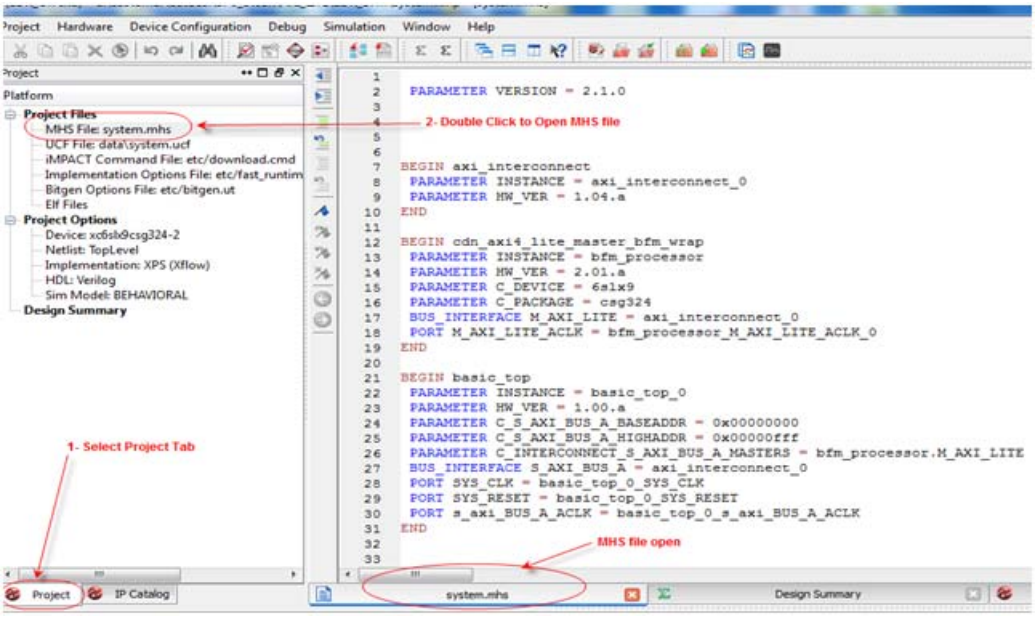


Figure 2-32: **Opening the MHS File**

2. Add the `sys_clk` and `sys_reset` external port declarations at the beginning of the MHS file.

```
PARAMETER VERSION = 2.1.0

PORT sys_clk = sys_clk, DIR = I, SIGIS = CLK, CLK_FREQ = 100000000
PORT sys_reset = sys_reset, DIR = I, SIGIS = RST
```

Figure 2-33: **External Port Declarations in MHS File**

3. Connect `sys_clk` and `sys_reset` to `axi_interconnect` in the MHS file.

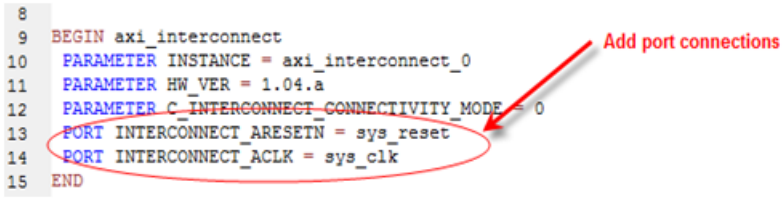


Figure 2-34: **Port Connections in MHS File**

4. Connect `sys_clk` to `bfm_processor` Pcore in MHS file.

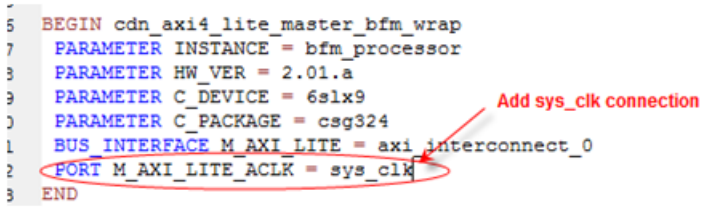


Figure 2-35: **sys_clk Connection in MHS File**

5. Connect the `sys_clk` and `sys_rest` to the Vivado HLS generated Pcore in the MHS file.

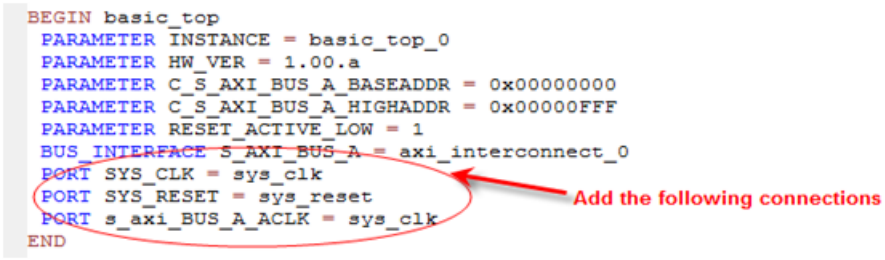


Figure 2-36: **Additional Connections in the MHS File**

Generating the Simulation Model

The next step is to generate the Simulation model.

1. Select **Project > Project Options** to set the Simulation Project option. In this example, Verilog and behavioral are selected.

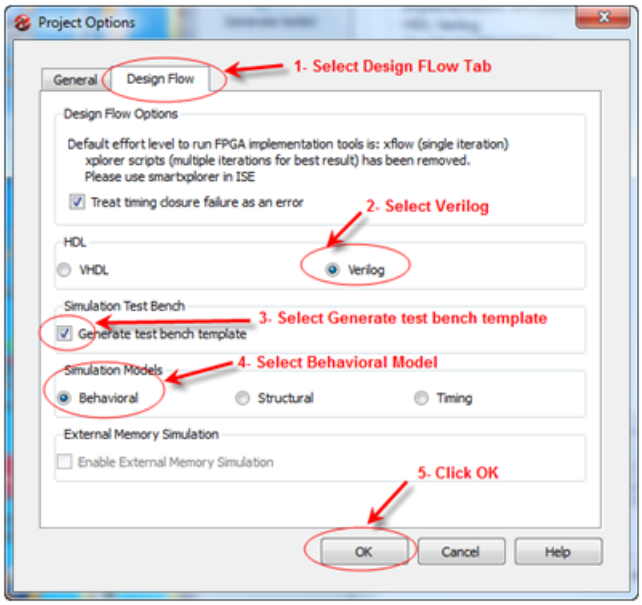


Figure 2-37: **Project Options Dialog Box**

2. In the Project Options dialog box, select the following options:
   - Design Flow
   - Verilog
   - Generate Test Bench Template
   - Behavioral model
3. Select **Simulation > Generate Simulation HDL Files** to generate the simulation file.

XPS creates the simulation directory structure to the XPS project.

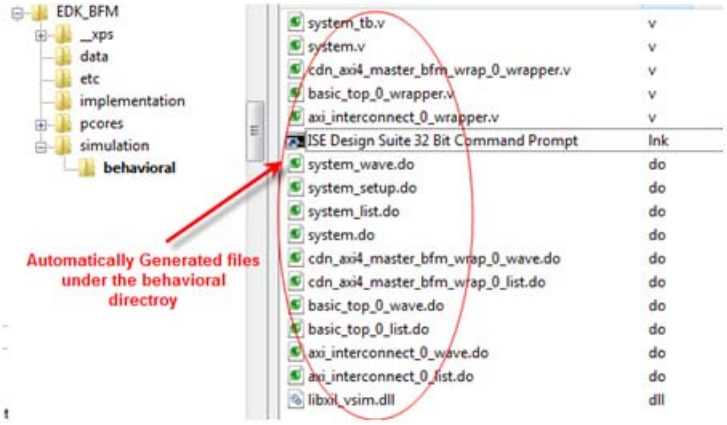


Figure 2-38: **Simulation Directory Structure**

4. Edit the `system_tb.v` file and add code to read/write the Pcore generated by Vivado HLS.

   The `system_tb.v` file is a template to which you must add code to read/write the Pcore registers.

   The BFM for the AXI4-Lite Master has predefined API for TASK to initiate transactions on the AXI4 interface. For detailed information on API, refer to *AXI Bus Functional Model (DS824)*.

   Two main tasks were added to the testbench to facilitate the reading/writing if the registers. These two tasks used a combination of the API defined in *AXI Bus Functional Model (DS824)*.

The write task is displayed in Figure 2-39.

```
task automatic SINGLE_WRITE;
   input [`ADDR_BUS_WIDTH-1 : 0] address;
   input [C_SLV_DWIDTH-1 : 0]    data;
   input [`PROT_BUS_WIDTH-1 : 0] prot;
   input [3:0]                   strobe;
   output[`RESP_BUS_WIDTH-1 : 0] response;
   begin
       $display ("  ==> AXI WRITE :  time[%t] Address[0x%08x] reg [%s] Data [0x%08x]" , $time,  address ,REG_NAME(address), data );
     fork
        dut.bfm_processor.bfm_processor.cdn_axi4_lite_master_bfm_inst.SEND_WRITE_ADDRESS(address,prot);
        dut.bfm_processor.bfm_processor.cdn_axi4_lite_master_bfm_inst.SEND_WRITE_DATA(strobe, data);
        dut.bfm_processor.bfm_processor.cdn_axi4_lite_master_bfm_inst.RECEIVE_WRITE_RESPONSE(response);
     join
     CHECK_RESPONSE_OKAY(response);
   end
endtask
```

Figure 2-39: **Write Task Code**

The read task is displayed in Figure 2-40.

```
task automatic SINGLE_READ;
   input [`ADDR_BUS_WIDTH-1 : 0] address;
   output [C_SLV_DWIDTH-1 : 0]    data;
   input [`PROT_BUS_WIDTH-1 : 0] prot;
   output[`RESP_BUS_WIDTH-1 : 0] response;
   begin
    dut.bfm_processor.bfm_processor.cdn_axi4_lite_master_bfm_inst.SEND_READ_ADDRESS(address,prot);
    dut.bfm_processor.bfm_processor.cdn_axi4_lite_master_bfm_inst.RECEIVE_READ_DATA(data,response);
    $display ("  ==> AXI READ  :  time[%t] Address[0x%08x] reg [%s] Data [0x%08x][%d]" , $time,  address ,REG_NAME(address), data,data );
    CHECK_RESPONSE_OKAY(response);
   end
endtask
```

Figure 2-40: **Read Task Code**

Testing for the Pcore is written using the above tasks. The example code is displayed in Figure 2-41.

```
    $display("---- TEST AESL PCORE                                    ");
      // write op code and make sure only 8 bit are writable
     SINGLE_WRITE(`VAR_A_ADDR,5,mtestProtection,4'b1111, response);  #20;
     SINGLE_READ(`VAR_A_ADDR,rd_data,mtestProtection,response);       #20;
     SINGLE_WRITE(`VAR_B_ADDR,10,mtestProtection,4'b1111, response);  #20;
     SINGLE_READ (`VAR_B_ADDR,rd_data,mtestProtection,response);       #50;
     $display ("----> Enable AP_START -----------");
     SINGLE_WRITE(`AP_START_ADDR,1,mtestProtection,4'b1111, response);  #50;
     SINGLE_READ (`AP_IDLE_ADDR,rd_data,mtestProtection,response);       #20;
     SINGLE_READ (`VAR_C_ADDR,rd_data,mtestProtection,response);       #20;
```

Figure 2-41: **Pcore Testing Example Code**

Refer to the `system_tb.v` file in the `/simulation` directory for the complete code.

Running the Simulation

The final step is to run the simulation.

1. Download the Cadence AXI BFM PLI library and copy it to the `/behavioral` directory.

   The library files are available online at: https://secure.xilinx.com/webreg/clickthrough.do?filename=axi_bfm_ug_examples.tar.gz

   Refer to *AXI Bus Functional Model (DS824)* for full details about the libraries. The windows library name is `libxil_vsim.dll`.

   Note: The `xil_vsim.dll` library is compiled for 32 bit systems. Therefore you must start the ModelSim simulator from a 32-bit command shell window.

   The recommended 32-bit windows shell is the one provided under the ISE 32 bit command prompt. To open this shell, select **Windows Start > Programs > Xilinx ISE Design Suite 14.1 > Accessories > ISE Design Suite 32 Bit Command Prompt**.

2. In the shell window, do the following:

   a. Change directories to the `/behavioral` directory.

   b. Run ModelSim in GUI mode and run `system_setup.do` by typing **`vsim -gui -do system_setup.do`**.

   ModelSim starts and runs the `system_setup.do` TCL script.

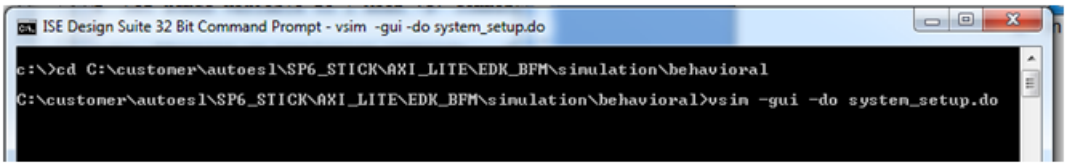


Figure 2-42: **system_setup.do TCL Script**

3. Override the template `system_tb.v` file in the `/behavioral` directory with the one in the `/simulation` directory.
4. In the ModelSim transcript window, type the following:
   - **`c`**

     This compiles the design files.
   - **`s`**

     This loads the design for simulation.
   - **`w`**

     This opens a wave window.
   - **`run -all`**

     This runs the test bench.

Figure 2-43 shows the output results on the transcript window.

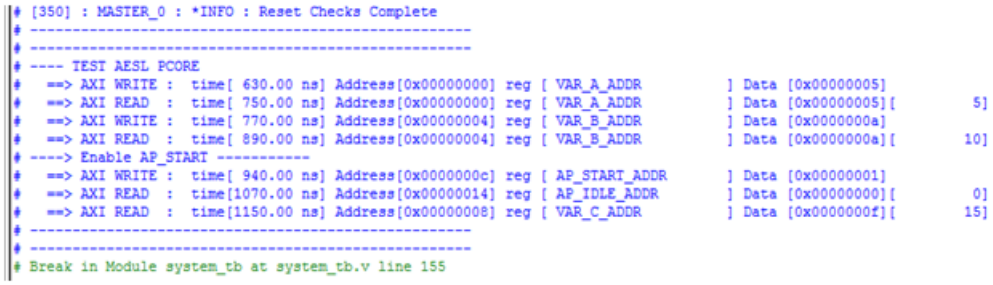


Figure 2-43: **Transcript Window**

Vivado HLS: Integrating System Generator

Introduction

One of the features in Vivado HLS 2012.2 is the ability to export RTL designs targeted to 7-series devices into Xilinx System Generator environment. This tutorial describes the steps in taking a design from Vivado HLS into System Generator.

Software Application for Vivado HLS

A Vivado HLS design project is made of 2 software components: a testbench and the code which will be transformed into hardware by the tool. The software directory included with this tutorial contains the software files for the example:

- Test bench file **`fir_test.cpp`**
- Design File **`fir.cpp`**
- A header file **`fir.h`** used with the test bench and the design files.

The header file **`filter.h`** is shown in Example 1.1.

```
#include <stdio.h>
#include <stdlib.h>
#include <hls_stream.h>

#define TAPS 21
#define RUN_LENGTH 100

int fir_hw(hls::stream<int> &input_val, hls::stream<int> &output_val);
```

Example 1.1 FIR Header File

The testbench file **`fir_test.cpp`** is shown in Example 1.2. This file follows the recommended Vivado HLS approach of separating the testbench code from the code

targeted for hardware implementation. This allows a simple way of exercising the same hardware function with different testbenches and code reuse in other hardware projects.

The testbench file is self-checking file. Vivado HLS requires the testbench to issue a return value of 0 if the functionality is correct and any non-zero value if there is an error. In this test bench, a version of the filter called **fir_sw** is ran in the test bench and it's results compared to those of function **fir_hw**. Function **fir_hw** will be synthesized to RTL: the test bench will confirm both functions produce the same results before and after synthesis.

By checking the output of the software implementation against the hardware implementation of function FIR, you can be certain that the generated hardware is correct. Another approach to generating self-checking testbenches is to have known good data files containing the expected result of the hardware function.

```
#include "fir.h"

int fir_sw(hls::stream<int> &input_val, hls::stream<int> &output_val)
{
  int I;
  static short shift_reg[TAPS] = {0};
  const short coeff[TAPS] = {6,0,-4,-3,5,6,-6,-13,7,44,64,44,7,-13,
                             -6,6,5,-3,-4,0,6};

  for(i=0; i < RUN_LENGTH; i++){
    int sample;
    sample = input_val.read();

    //Shift Register
    for(int j=0; j < TAPS-1; j++){
      shift_reg[j] = shift_reg[j+1];
    }
    shift_reg[TAPS-1] = sample;

    //Filter Operation
    int acc = 0;
    for(int k=0; k < TAPS; k++){
      acc += shift_reg[k] * coeff[k];
    }
    output_val.write(acc);
  }
}

int main()
{
  hls::stream<int> input_sw;
  hls::stream<int> input_hw;
  hls::stream<int> output_hw;
  hls::stream<int> output_sw;

  //Write the input values
  for(int i = 0; i < RUN_LENGTH; i++){
    input_sw.write(i);
    input_hw.write(i);
  }
```

```
  //Call to software model of FIR
   fir_sw(input_sw, output_sw);
  //Call to hardware model of FIR
  fir_hw(input_hw, output_hw);

  for(int k=0; k < RUN_LENGTH; k++){
    int sw, hw;
    sw = output_sw.read();
    hw = output_hw.read();
    if(sw != hw){
      printf("ERROR: k = %d sw = %d hw = %d\n",k,sw,hw);
          return 1;
    }
  }
  printf("Success! both SW and HW models match.\n");
  return 0;
}
```

Example 1.2 FIR Testbench Code

The version of the FIR function, **fir_hw**, which will be exported to a System Generator design, is shown in Example 1.3. This code is the same code as the software version of FIR. This design uses the hls::stream class to implement a streaming data type. See the *Vivado Design Suite User Guide: High-Level Synthesis (UG902)* for more details on using steaming interfaces.

An optimization directive for the **fir_hw** function is embedded into the source code as a pragma: HLS PIPELINE II=1 rewind. This optimization will ensure that in the RTL implementation, each iteration of the for-loop will be implemented to operate in a pipelined manner with 1 clock cycle (II=1) between iterations: iteration 1 will start, and one clock cycle later iteration 2 will start (even though iteration 1 has not finished). The rewind option ensures this iteration rate can be performed by the entire function.

```
#include "fir.h"

int fir_hw(hls::stream<int> &input_val, hls::stream<int> &output_val)
{
  int I;
  static short shift_reg[TAPS] = {0};
  const short coeff[TAPS] = {6,0,-4,-3,5,6,-6,-13,7,44,64,44,7,-13,
                             -6,6,5,-3,-4,0,6};

  for(i=0; i < RUN_LENGTH; i++){
#pragma HLS PIPELINE II=1 rewind

    int sample;
    sample = input_val.read();

    //Shift Register
    for(int j=0; j < TAPS-1; j++){
      shift_reg[j] = shift_reg[j+1];
    }
    shift_reg[TAPS-1] = sample;

    //Filter Operation
```

```
    int acc = 0;
    for(int k=0; k < TAPS; k++){
      acc += shift_reg[k] * coeff[k];
    }
    output_val.write(acc);
  }
}
```

Example 1.3 FIR Code for Hardware Generation

Create a Project in Vivado HLS for the FIR Application

The following steps will demonstrate how to run Vivado HLS and create the FIR application as a hardware block for a System Generator based design.

To invoke Vivado HLS, through the Windows menu: **Start** > **All Programs** > **Xilinx Design Tools** > **Vivado** > **Vivado HLS**.

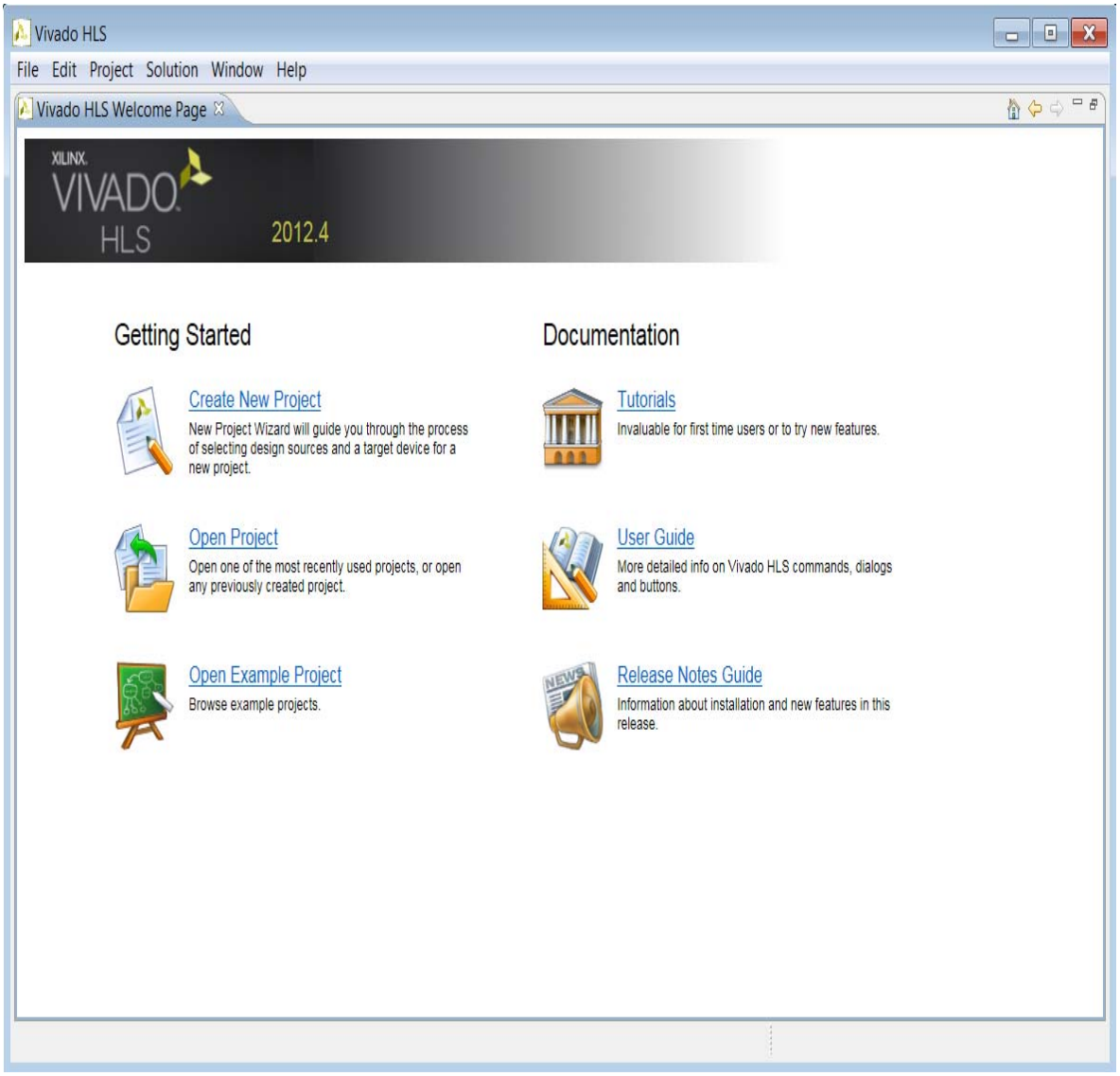


Figure 3-1: Vivado HLS Welcome Screen

1. Click the **Create New Project** button on the GUI toolbar.
2. Set the project name to **`fir_prj`** (Figure 3-2).
3. **Browse** to the location of the fir_sys_gen directory to set the location for the project (Figure 3-2).

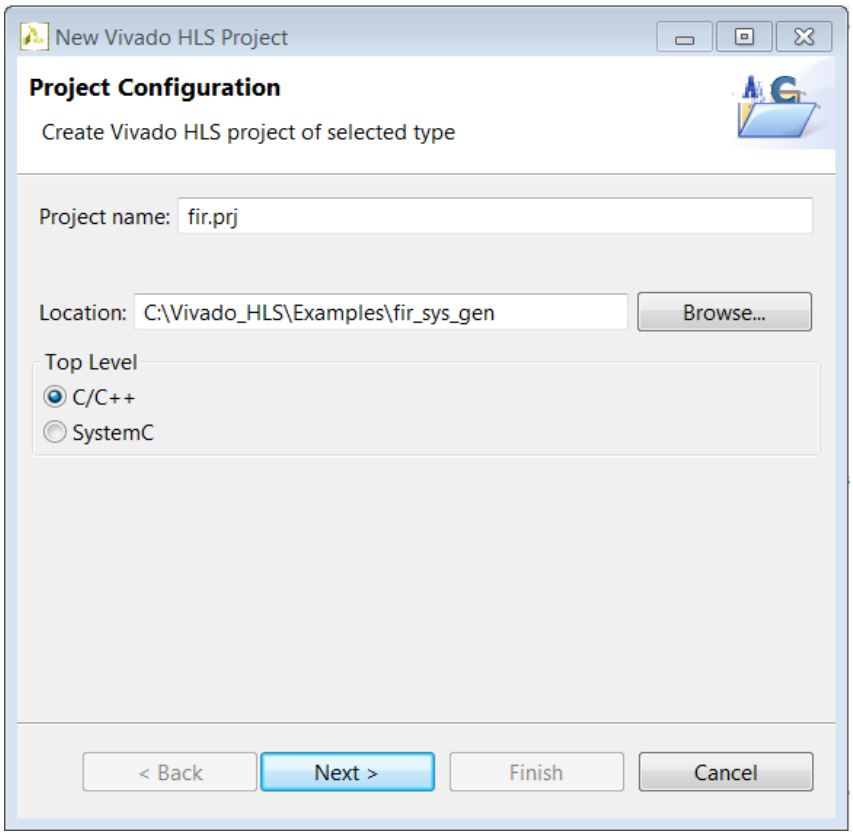


Figure 3-2: Project Configuration

4. Click **Next** to set the top function to **`fir_hw`** and add the **`fir.cpp`** (Figure 3-3).

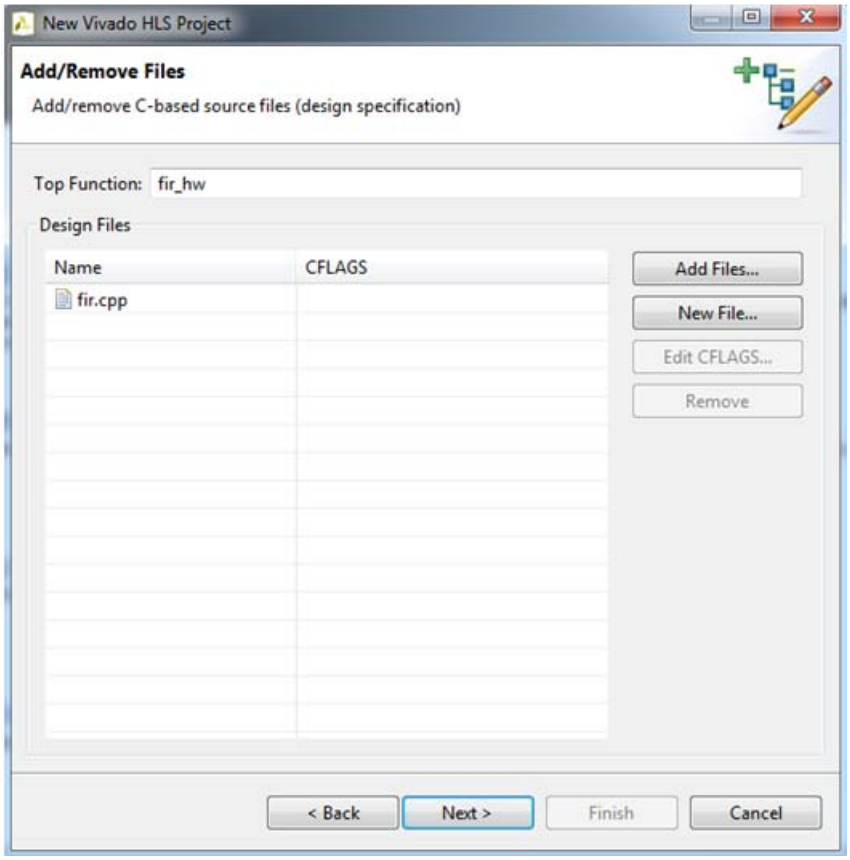


Figure 3-3: Add Files for Hardware Synthesis

5. Click **Next** to add the file **`fir_test.cpp`** (Figure 3-4).

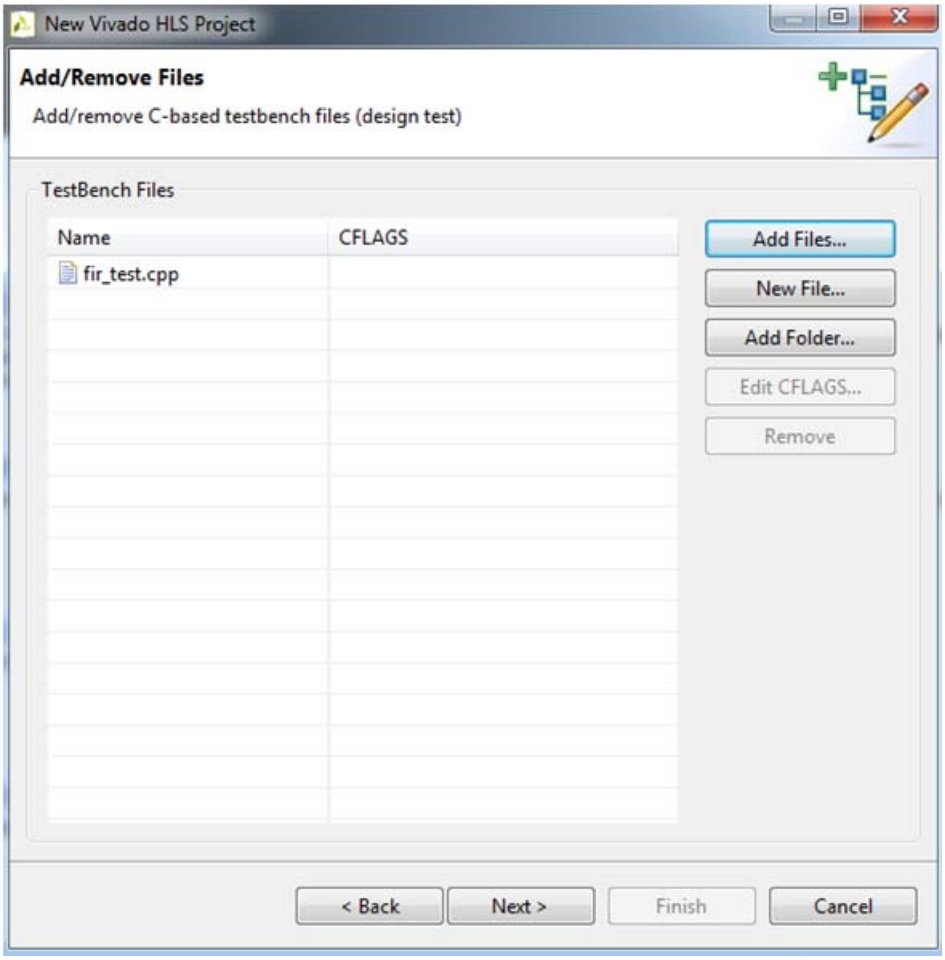


Figure 3-4: Add Testbench Files

6. Set the clock period to 10 (Figure 3-5).
7. Click on **Part Selection** to set the FPGA target (Figure 3-6).

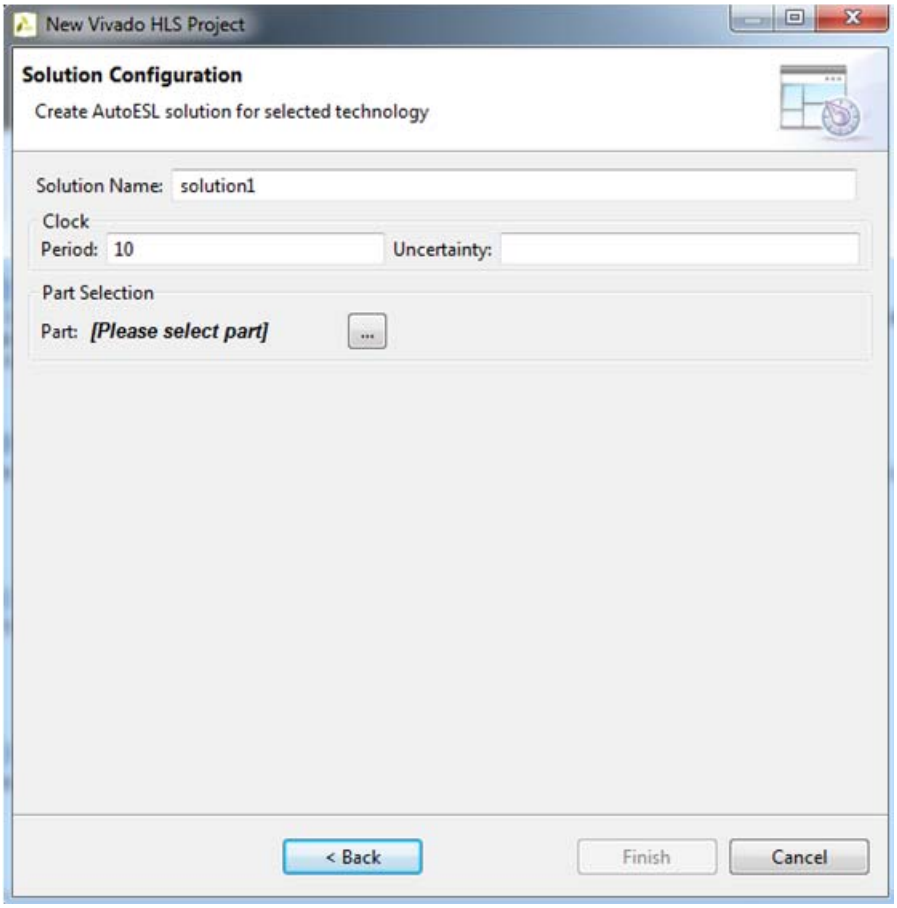


Figure 3-5: Solution Configuration

8. Set the part to **`xc7k420tffg1156-1`** and click **OK**(Figure 3-7).

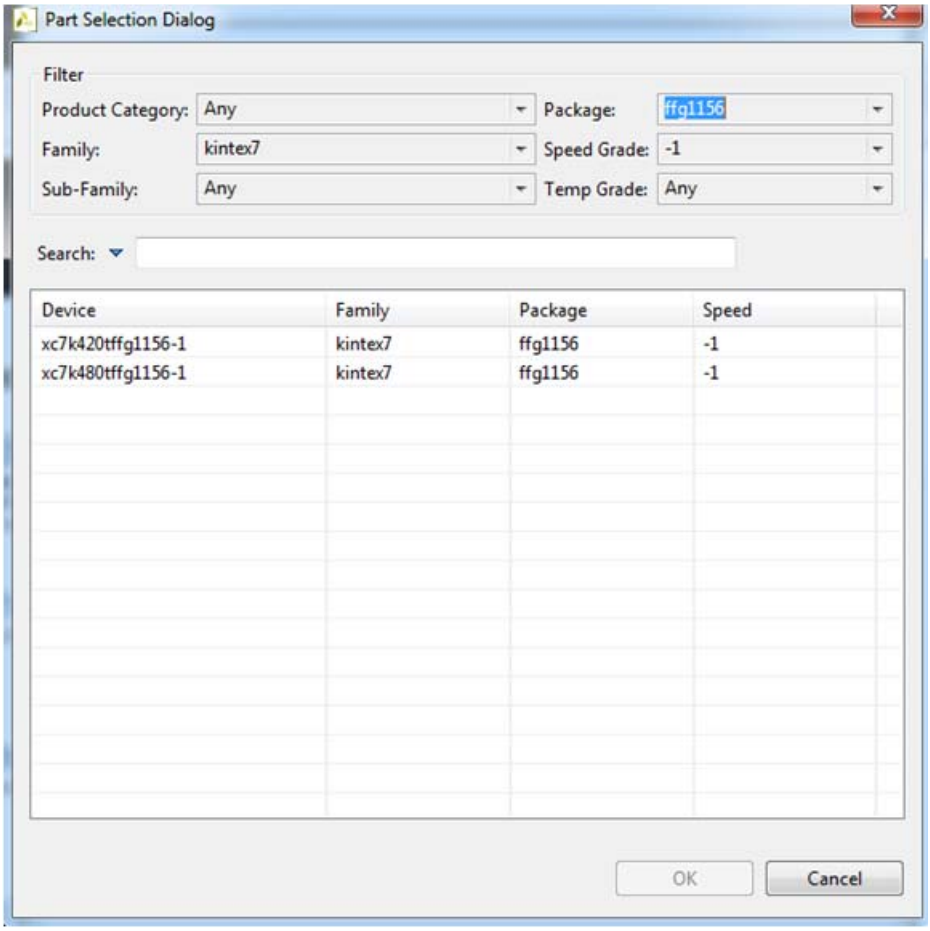


Figure 3-6: Part Selection

9. Click **Finish**, the Vivado HLS GUI should look like Figure 3-7.

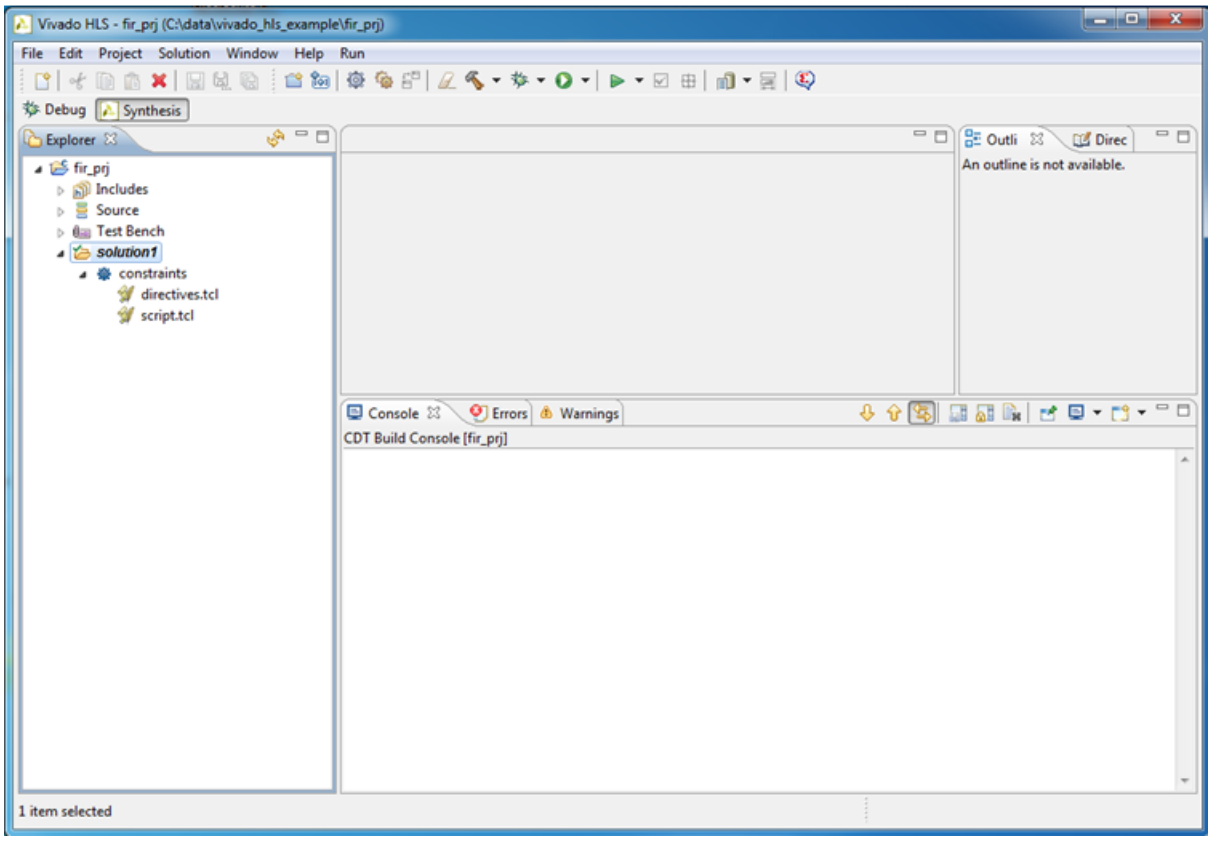


Figure 3-7: Vivado HLS after Project Creation

Create an RTL design

The first step in generating a hardware block with Vivado HLS is to check the correctness of the C code. This can be done within Vivado HLS using the software build and run commands.

The steps to verify C code works correctly are as follows:

1. Click the **Run C Simulation** icon on the GUI toolbar (Figure 3-8).

Figure 3-8: Run C Simulation Icon

2. When the C simulation dialog box opens, select **OK** without selecting any of the options (Figure 3-9).

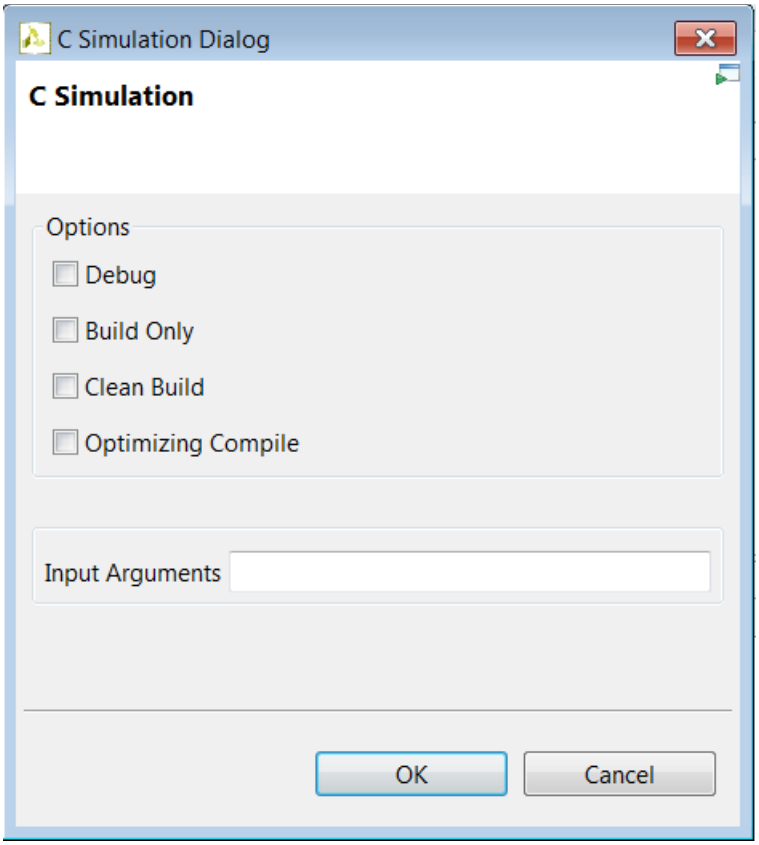


Figure 3-9: C Simulation Dialog Box

3. When the C simulation completes, the console should look like Figure 3-10.

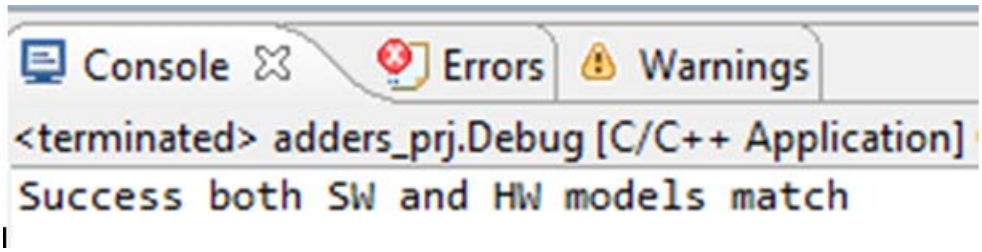


Figure 3-10: Expected Console Output

Once the C code is known to be correct, it is time to generate the module for System Generator. The following steps describe how to accomplish this task.

4. Click the **Synthesis** icon on the GUI toolbar (Figure 3-11).

Figure 3-11: Synthesis Icon

Once synthesis complete, the report file will open automatically. The report can be review to ensure the design meets the desired area and performance. If the desired performance has been achieved, the design can be exported to the System Generator environment.

5. Click the **RTL Export** icon on the GUI toolbar (Figure 3-12).

Figure 3-12: RTL Export Icon

6. When the Export RTL menu opens, select **System Generator for DSP (Vivado)** from the drop-down menu (Figure 3-13).
7. Click **OK**.

At this step, logic synthesis can be run to evaluate if the timing and area estimates reported by Vivado HLS will be met after RTL synthesis. This step is not performed in this example.

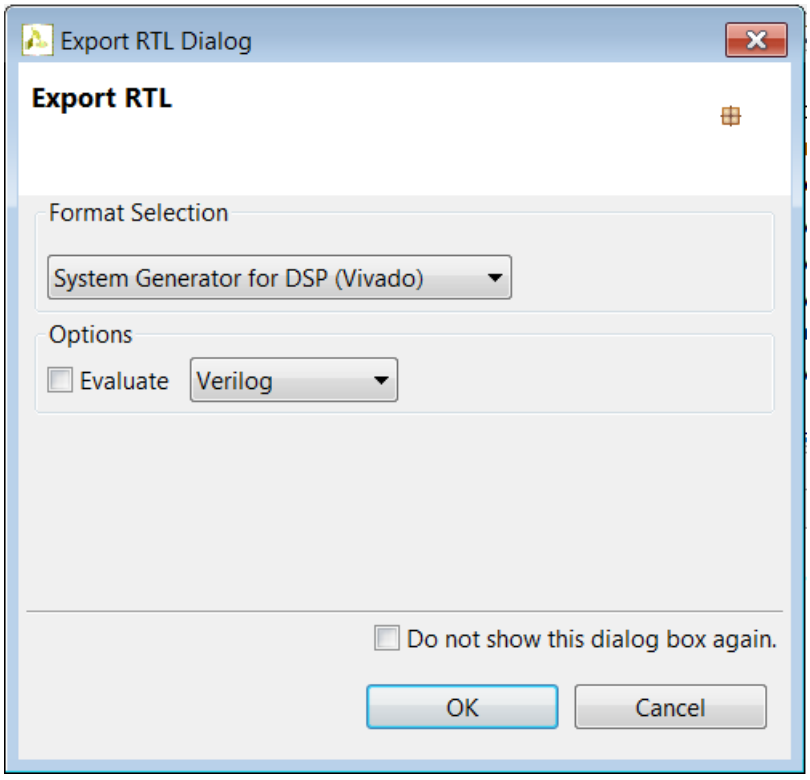


Figure 3-13: RTL Export Dialog

8. Check the console for successful execution (Figure 3-14).

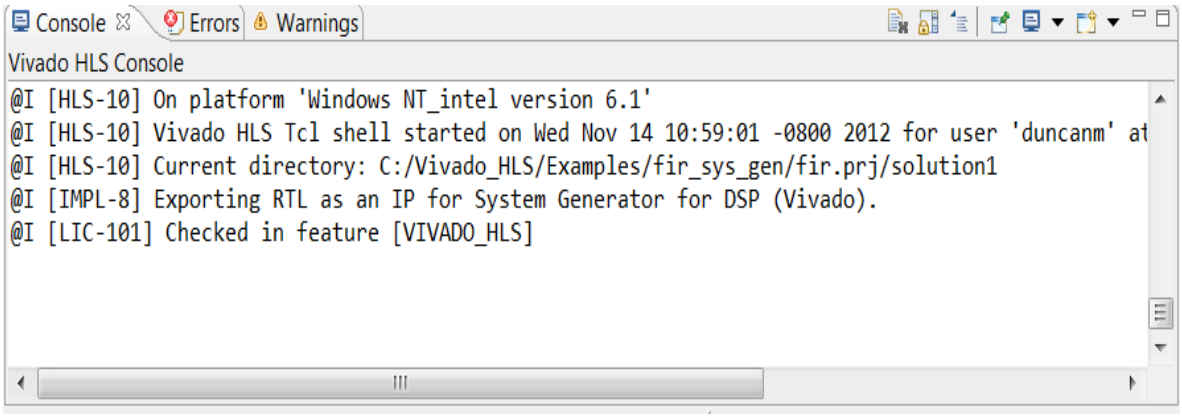


Figure 3-14: Successful Execution of RTL Export

The package for System Generator will be available in the `solution/impl` directory. A sysgen directory containing simulation and implementation models of the synthesized C/C++ function will be created in this location.

Note: If RTL synthesis was run to evaluate the design, the results of RTL synthesis are not included in the export package. They are only provided as an evaluation check, not part of the IP, and hence they are stored in the `solution/impl/<HDL>` directory (verilog or VHDL, depending on the selection made in Figure 3-13). The RTL IP should be re-synthesized with the complete design to obtain the final results (after the entire design is placed and routed).

Import the Design into System Generator

Open the file **`fir_sysgen.mdl`** file in MatLab. This shows the design shown in Figure 3-15.

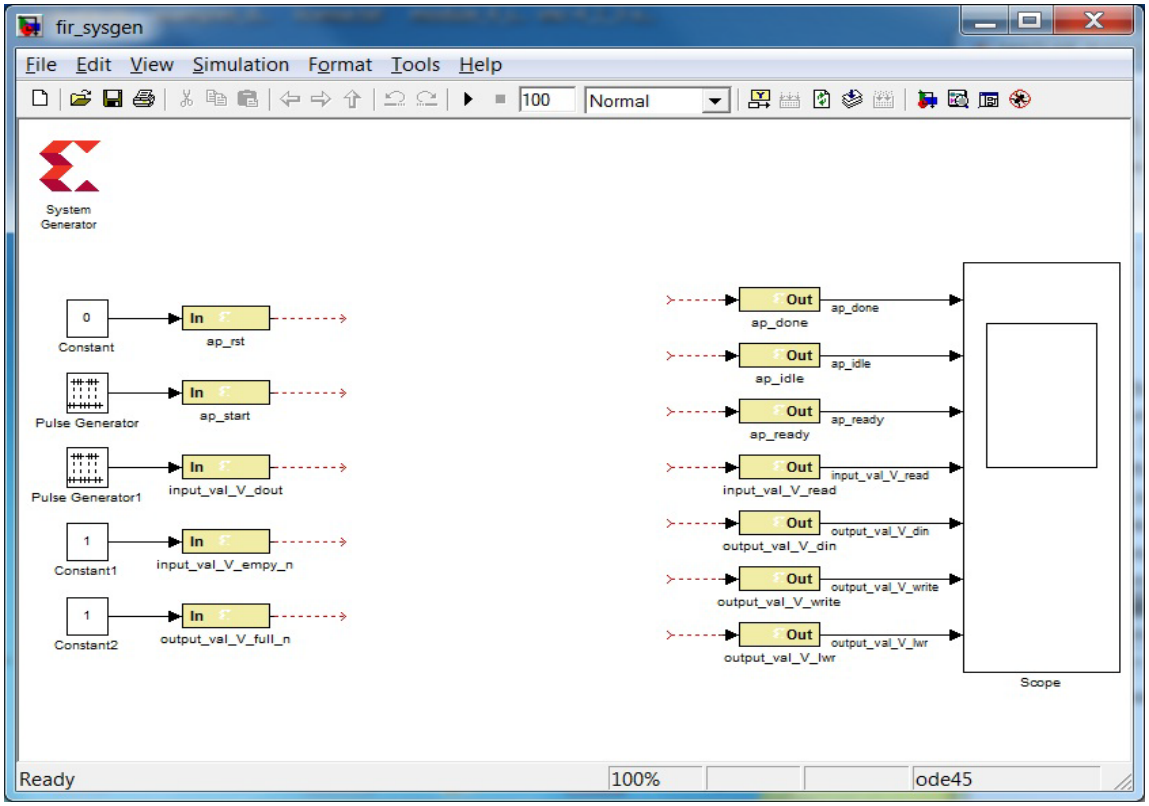


Figure 3-15: Initial fir_sysgen Design

The RTL IP created by Vivado HLS can now be imported into this initial design.

1. Right-click and select the option **XilinxBlockAdd** to instantiate new Vivado HLS block.
2. Scroll down the list in dialog box and select **Vivado HLS** or partially type the name Vivado HLS, as shown in Figure 3-16.
3. Select **Vivado HLS** to insatiate the initial block.

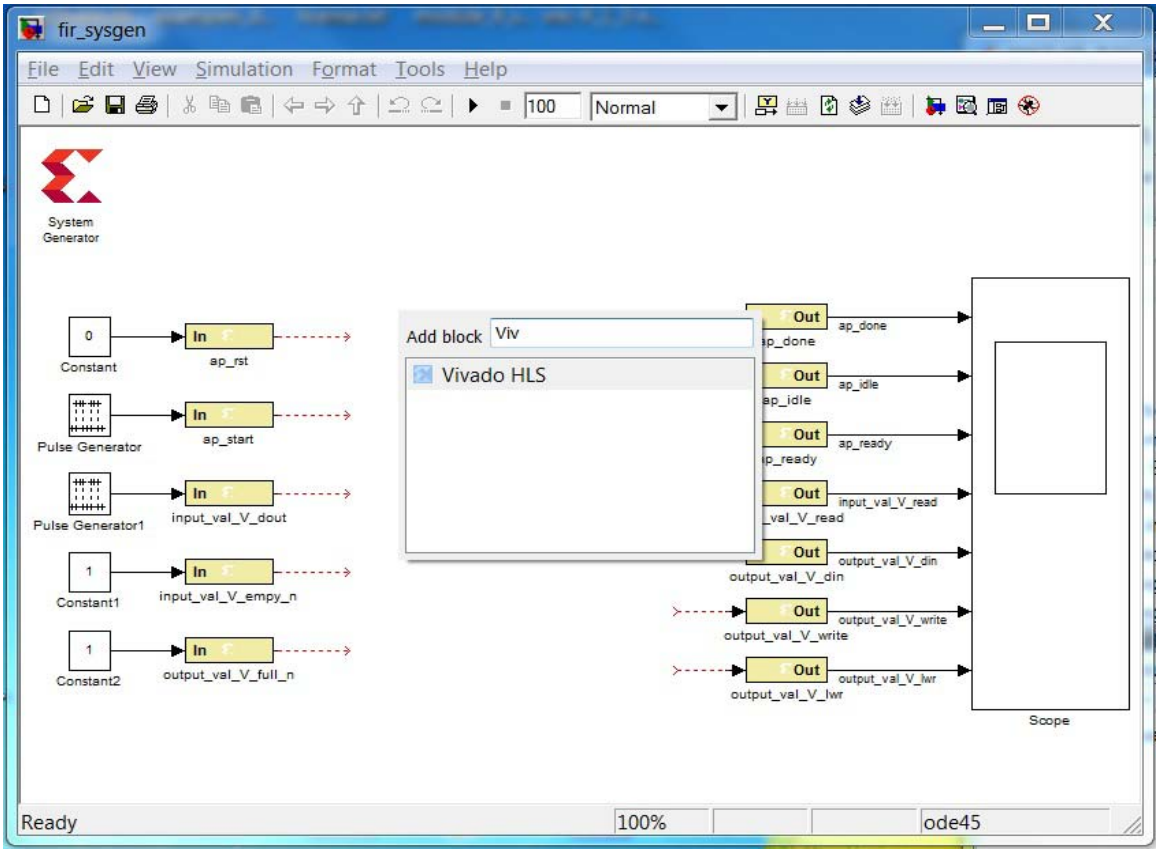


Figure 3-16: Instantiating an Vivado HLS Block

The next step is to import the RTL IP from the Vivado HLS project solution directory.

4. Double-click on the newly instantiated Vivado HLS block to open the Block Parameters dialog box.

Browse to the `solution` directory where the Vivado HLS block was exported (Figure 3-17).

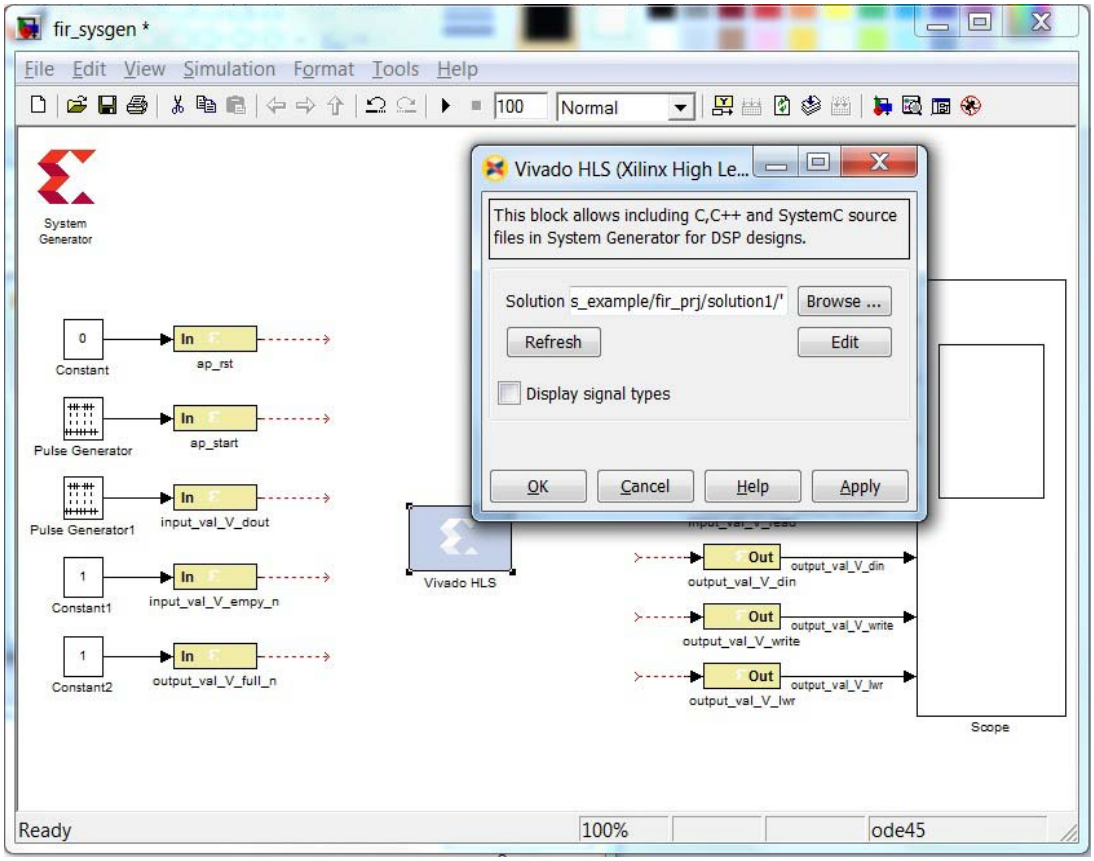


Figure 3-17: Importing Vivado HLS IP

Connect the ports on the IP to the design ports to obtain the results shown in Figure 3-18.

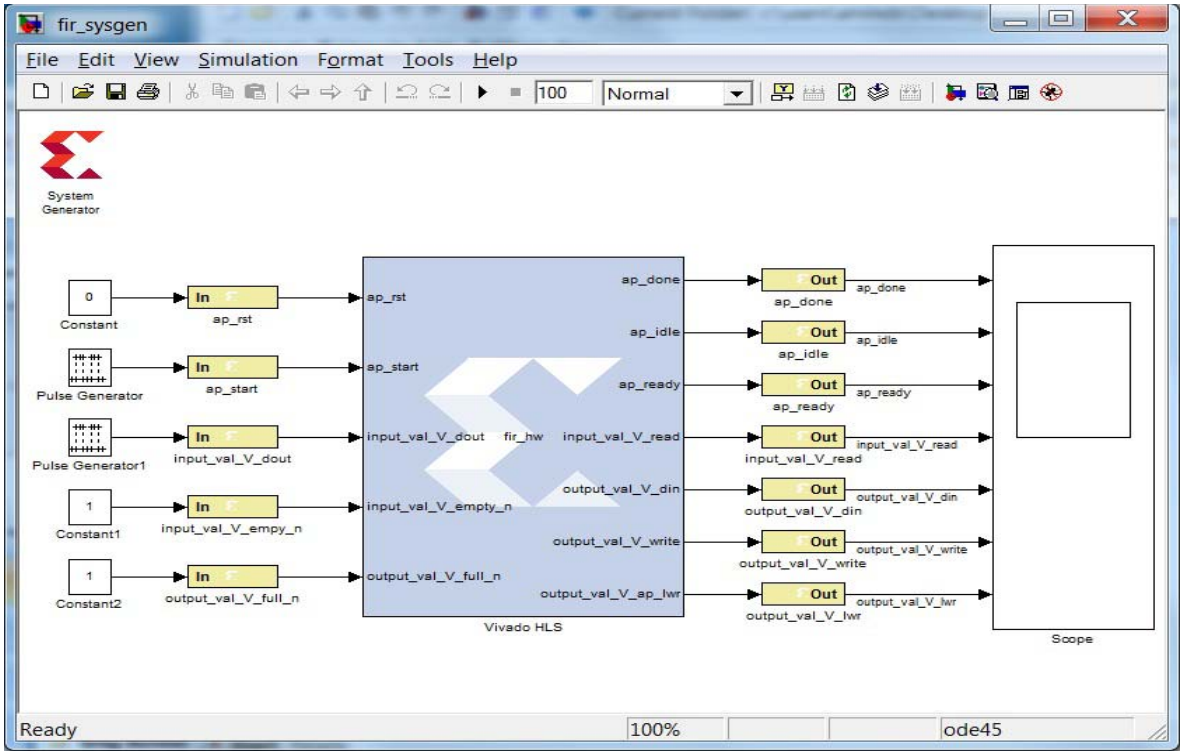


Figure 3-18: Final Design

Using AXI4 Interfaces

Introduction

This tutorial shows how to add AXI4 interfaces to a Vivado HLS design. The tutorial uses the example of connecting to the ARM processor on a Zynq device.

By the end of this tutorial, you will be able to:

- Create a basic Zynq system
- Instantiate an HLS generated block in a Zynq system
- Debug the communication between HLS generated IP and ARM
- Connect two HLS IPs using AXI4-streaming
- Use the AXI-DMA core for data transfers to external memory from HLS IP

Before starting on the exercises in this guide, please copy all materials to a directory named autoesl_axi on your computer.

Required Hardware

The exercises in this guide require the following board from Xilinx:

- Zynq-7 ZC702'

Exercise 1: Creating a basic Zynq System

This exercise focuses on the creation of a basic Zynq system in which the ARM processor is used. There are no FPGA fabric accelerators in this exercise. The purpose of this exercise is to help familiarize you with the Zynq design flow.

Creating a PlanAhead Project

To create a Zynq system, start PlanAhead in either 32 or 64-bit version depending on your computer. To invoke PlanAhead, through the Windows menu: Start ' All Programs ' Xilinx Design Tools ' ISE Design Suite 14.1 ' PlanAhead. The PlanAhead group is shown in Figure 4-1.

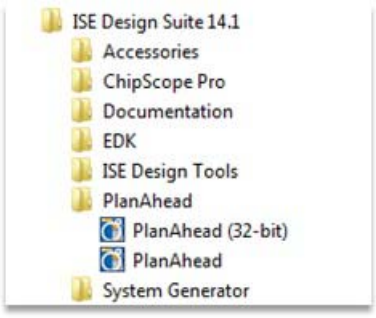


Figure 4-1: Launching PlanAhead

This brings up PlanAhead GUI as shown in Figure 4-2.

Figure 4-2: PlanAhead GUI

As can be seen when the GUI opens, the first step is to open or create a new project. Start by creating a new project in the autoesl_axi directory in the local work area.

1. Click Create New Project to open the Project Wizard (Figure 4-3), and press next.
2. Enter the project name as exercise1 (Figure 4-4), and press next.
3. Change the project location to a suitable location on your system

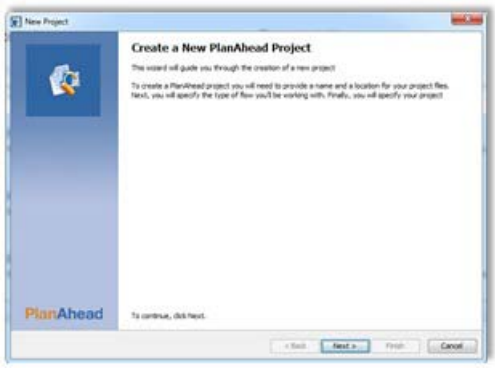


Figure 4-3: Project Wizard

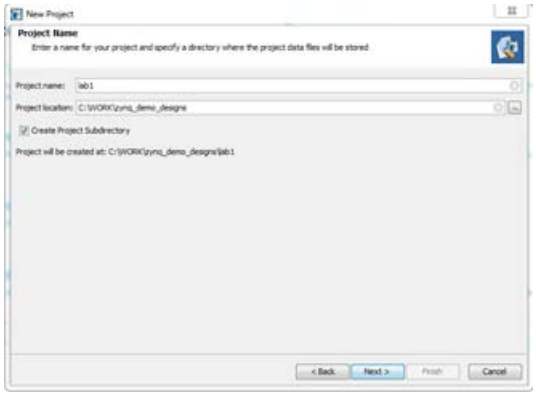


Figure 4-4: Project Name Definition

4. Select RTL project (Figure 4-5) and click next.
5. Click next to skip adding sources at this time.
6. Click next to skip adding existing IP at this time.

7. Click next to skip adding constraints at this time.

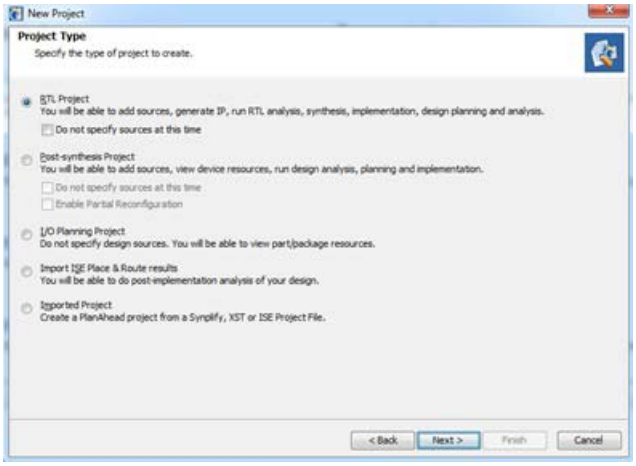


Figure 4-5: Select RTL Project

8. In the default part dialog, click on boards (Figure 4-6)
9. Set family to Zynq-7000
10. Select the Zynq-7 ZC702 Evaluation Board, and click next
11. Review the information in the project summary (Figure 4-7), and click finish.

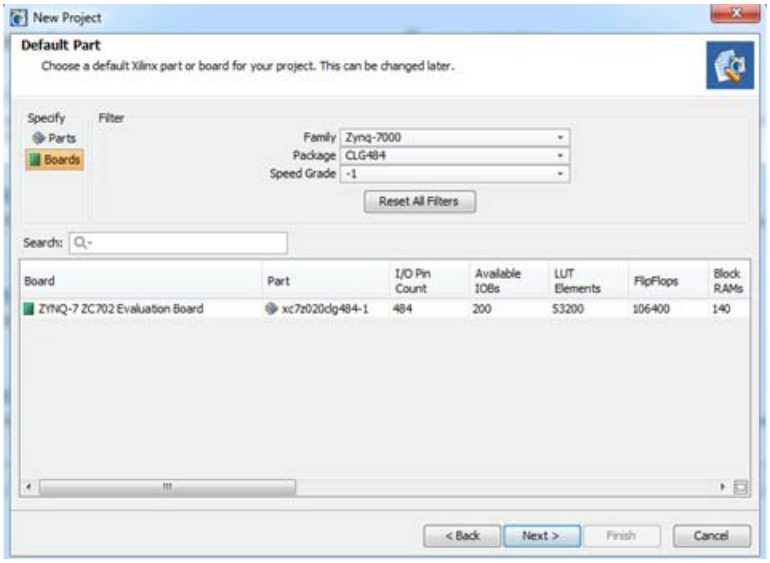


Figure 4-6: Part Selection

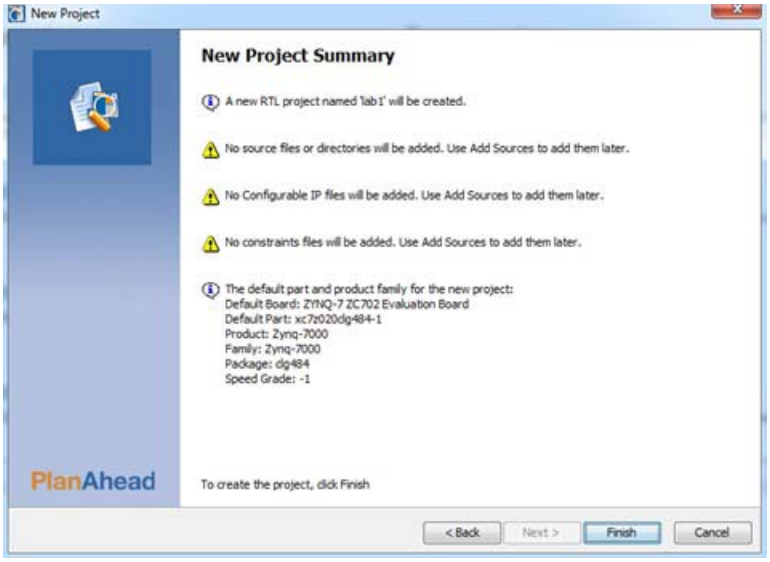


Figure 4-7: New Project Summary

Once a project has been successfully created, the PlanAhead GUI will open with an empty project (Figure 4-8). The first thing to do is to add an embedded subsystem for the ARM processor.

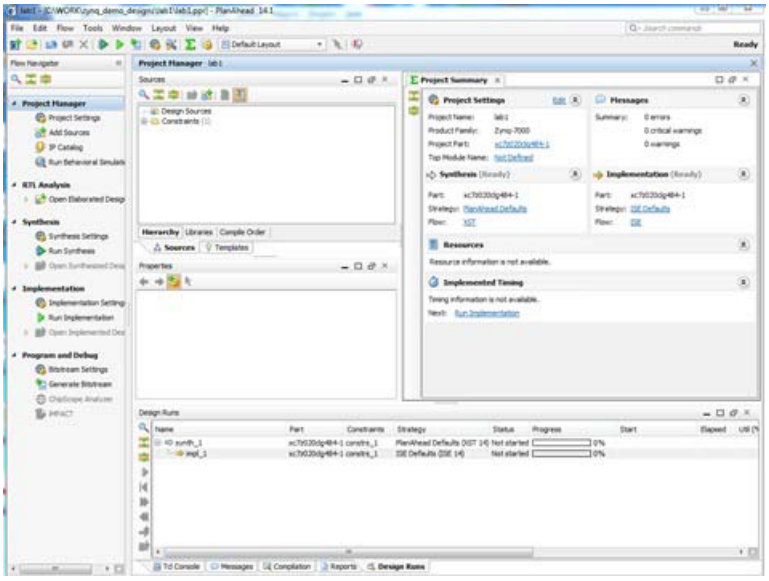


Figure 4-8: PlanAhead GUI with empty

Creating a Processor Sub-System

The following steps describe how to create a processor sub-system for a PlanAhead project.

12. Click Add Sources
13. Select add or create embedded sources (Figure 4-9) and click next.
14. Click Create Sub-Design (Figure 4-10)
15. Set module name to system and click ok
16. Click Finish

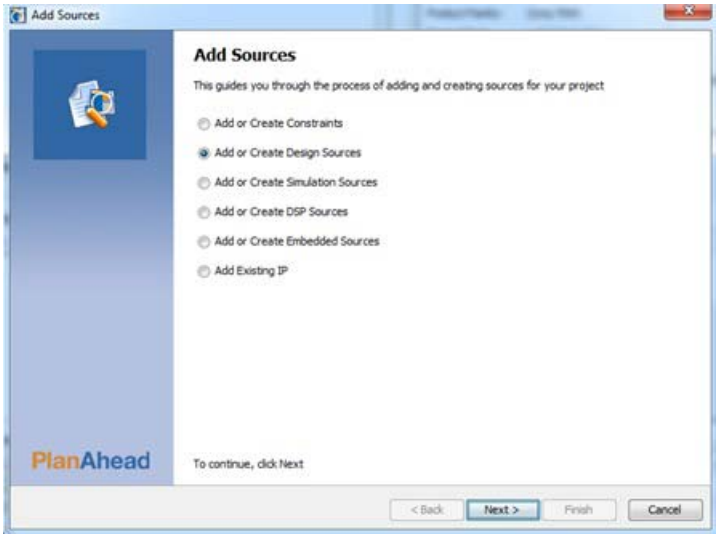


Figure 4-9: Add Sources

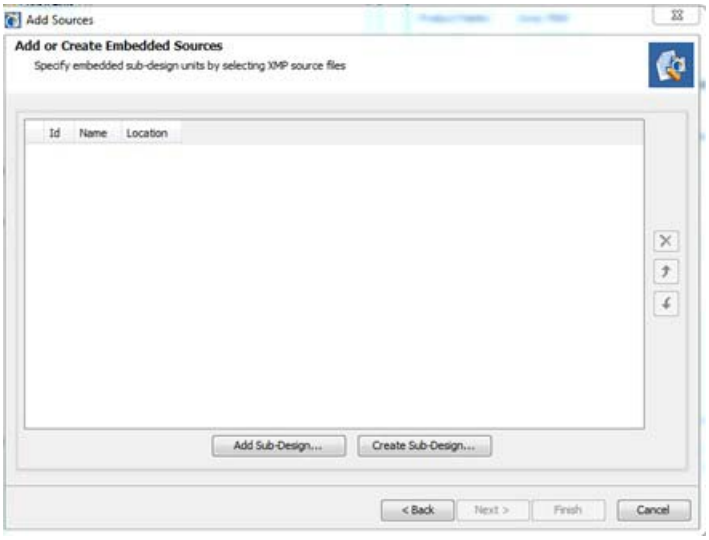


Figure 4-10: Create Sub-Design

Once the sub-system is created, the system will start Xilinx Platform Studio (XPS) to configure the ARM processor sub-system. XPS will ask to create a processor subsystem, click yes. The screen from XPS should look like Figure 4-11.

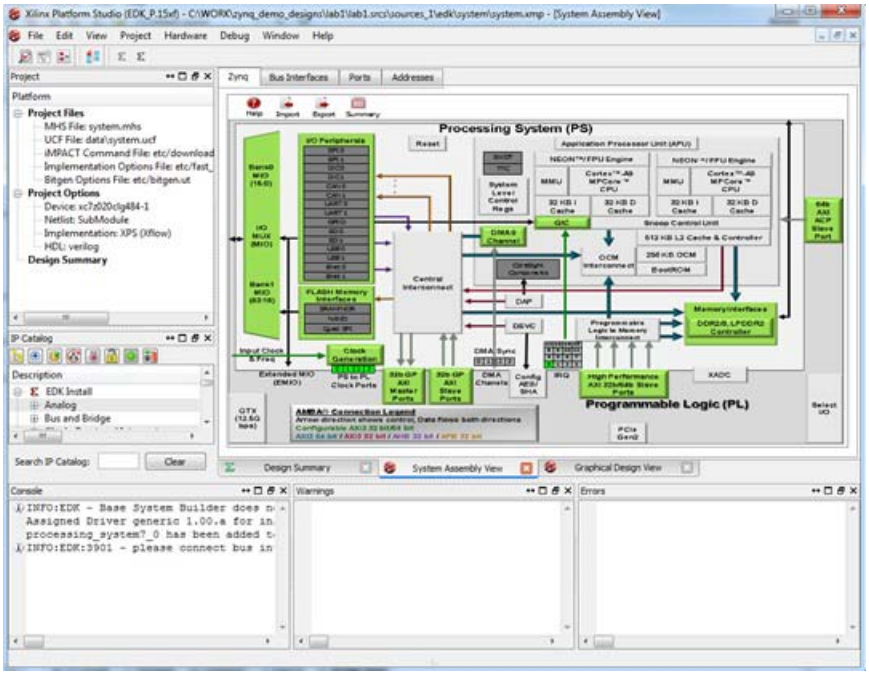


Figure 4-11: XPS Main Screen

17. Click Import in the Zynq tab (Figure 4-12)
18. Select the ZC702 Template (Figure 4-13) and click ok
19. Click yes on dialog box to update Zynq MIO and MHS files.
20. Close XPS
21. Click yes on message to close XPS. This will return you to PlanAhead

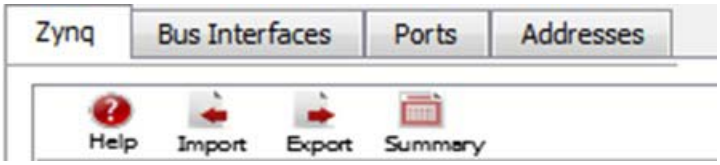


Figure 4-12: Zynq Configuration Tab

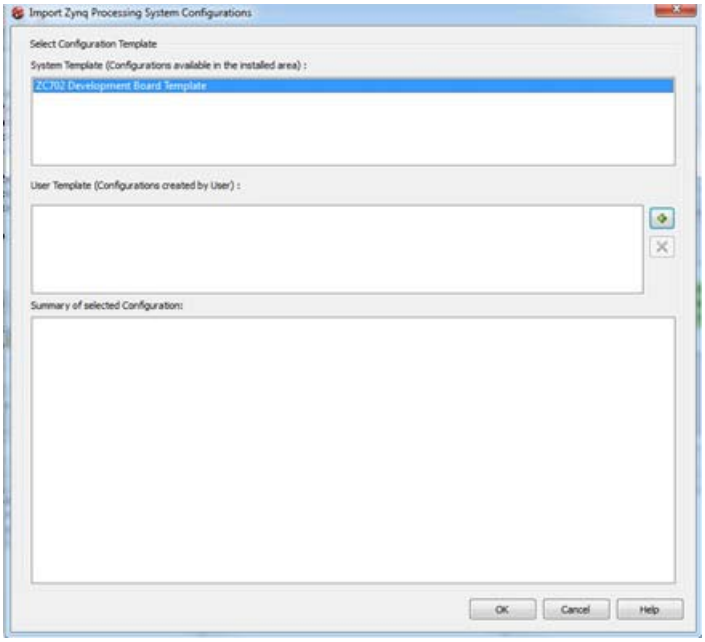


Figure 4-13: Configuration Template Selection

Designs in PlanAhead require a top level RTL file for system generation. Since the processor component of this design is a sub-system, a top level wrapper is needed to connect the sub-system to the pins on the FPGA. PlanAhead enables the automatic generation of top level RTL files in the following steps.

22. Click on system in the sources tab to select it.

23. Right click on system and select Create Top HDL (Figure 4-14)

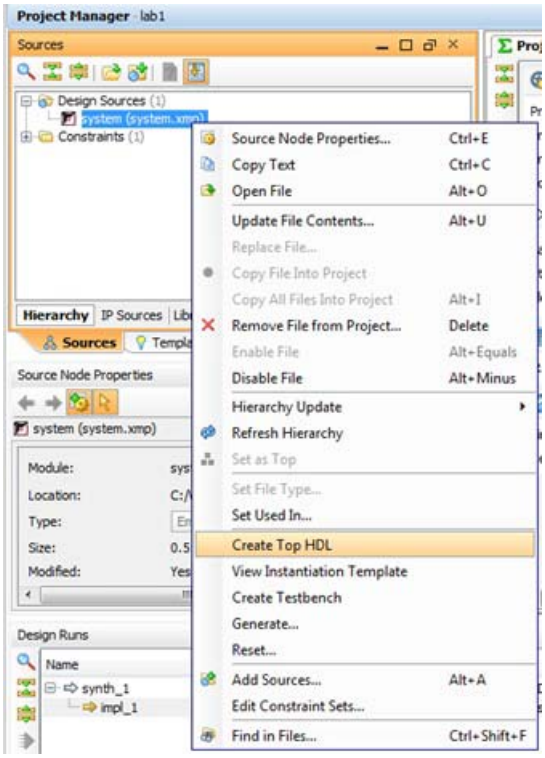


Figure 4-14: Creating Top HDL

Exporting a Hardware Platform to Software Development

The generation of the top level RTL file completes the creation of the hardware platform. The following steps describe how to export the hardware platform into the embedded software development environment Xilinx SDK. The steps to transfer the hardware design to the software development environment are

24. Click on File.
25. Click on Export.
26. Click on Export Hardware Figure 4-15.
27. In the Export Hardware GUI select Export Hardware (Figure 4-16)

28. Select Launch SDK and click OK (Figure 4-16)

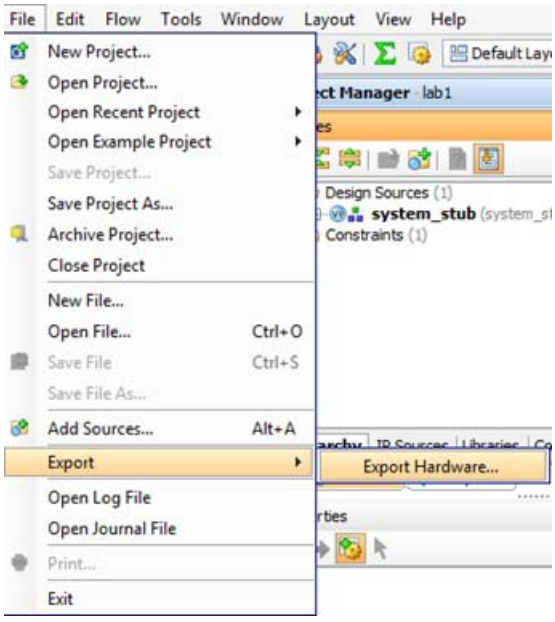


Figure 4-15: Export Hardware to Software Development

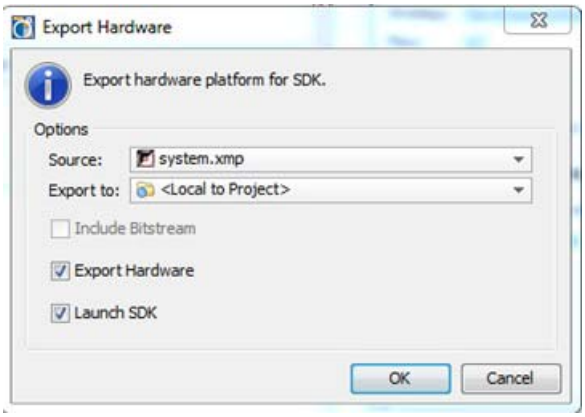


Figure 4-16: Export Hardware GUI

After step 1.28 completes, Xilinx SDK should be open and the GUI should look like Figure 4-17.

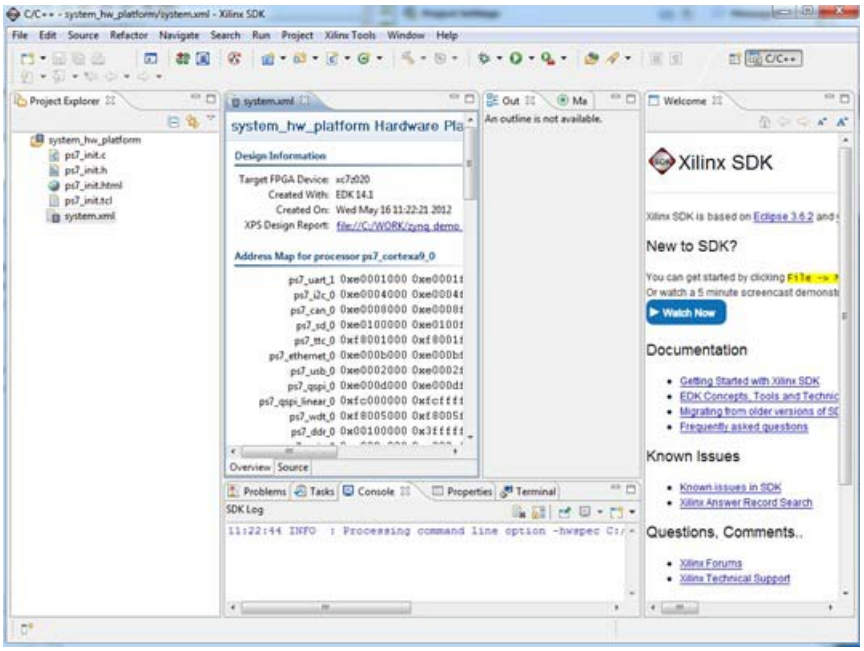


Figure 4-17: Xilinx SDK GUI

Creating a Basic Software Application

Xilinx SDK is the embedded software design tool for creating applications for the hardware platform defined in Xilinx Platform Studio. The following steps show how to create an initial "hello world" application running on the ARM processor.

29. Create a new project by selecting File > New > Xilinx C Project (Figure 4-18)
30. Select Hello World from the project templates (Figure 4-19) and click next
31. Select create new board support package and click finish (Figure 4-20)

By creating a new board support package, SDK is taking the hardware platform information from XPS and putting together the minimum header and library files to access the capabilities of the hardware from software. This is a required step for the first software

project using a given hardware platform from XPS. The board support package can be shared across all software applications, which target the same hardware system.

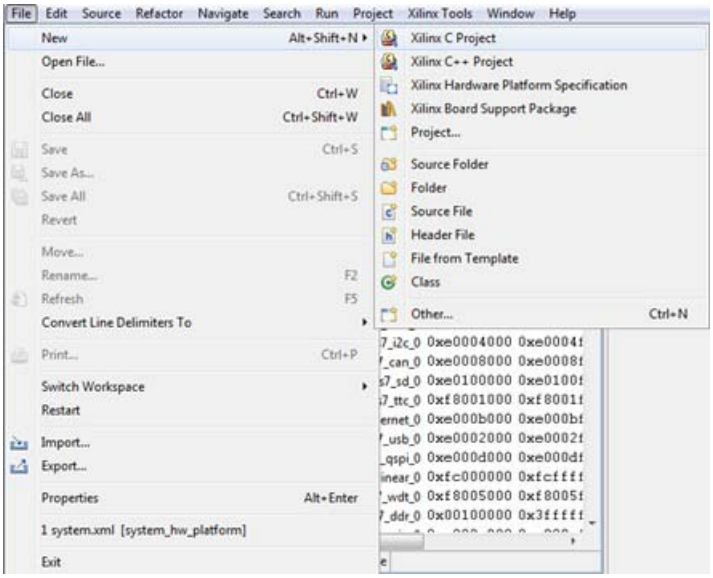


Figure 4-18: Creating a New Project

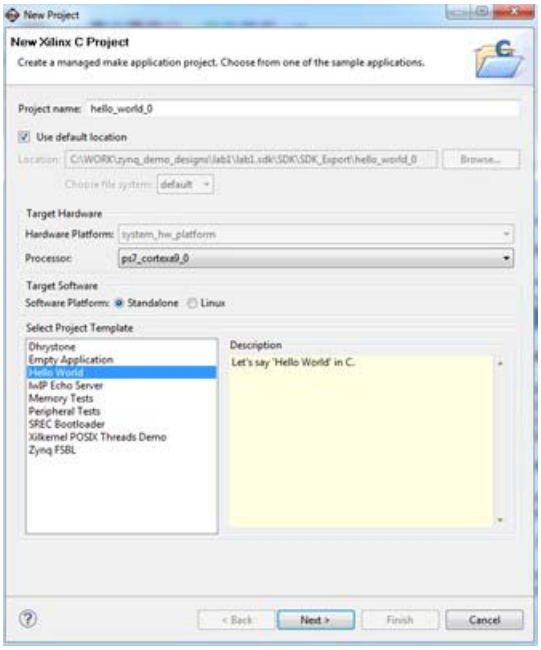


Figure 4-19: New Project GUI

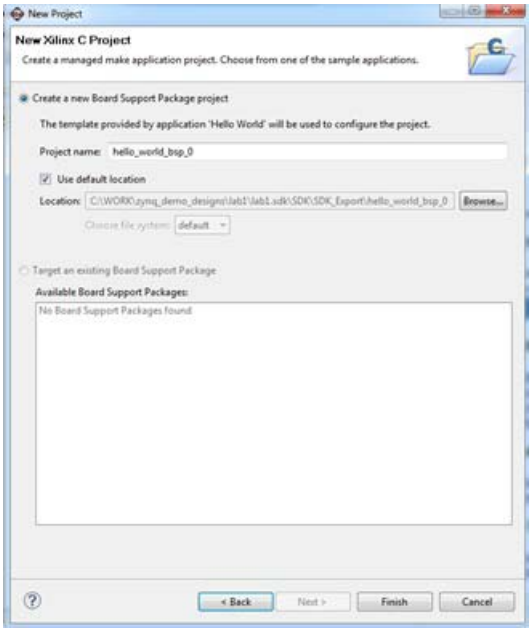


Figure 4-20: Create New Board Support Package

The "Hello World" application is shown in Figure 4-21. This code has 3 parts: initialization, body of the application, and shutting down the system. The initialization function init_platform activates the caches for the ARM processor and enables the UART for output. In this case the body of the application is a simple print statement. The shutdown part of the program disables the cache for the processor. The application will be automatically compiled and processor executable will be generated.

```
int main()
{
  // Initialization
  init_platform();

  // Application Body
  print("Hello World\n\r");

  //Program Shutdown
  cleanup_platform();
  Return 0;
}
```

Figure 4-21: **Hello World Program for Zynq**

In order to view the result of the program running on the processor, a terminal connection must be created. At this point the user can choose between either the built-in terminal program in SDK or their own terminal emulation program. The following steps describe how to configure the terminal in SDK.

32. Select Terminal 1 from the bottom panel of the SDK GUI (Figure 4-22)

33. Click the settings button for the terminal (Figure 4-23)
34. Configure the terminal as shown in Figure 4-24 and click ok. The COM port number is computer dependent.

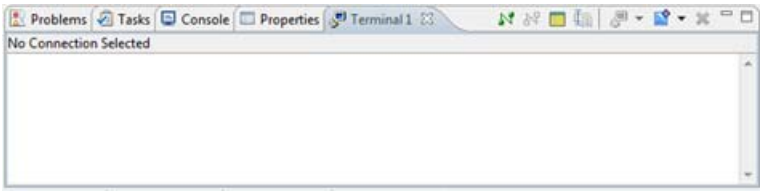


Figure 4-22: Terminal Emulation Tab

Figure 4-23: Terminal Settings Button

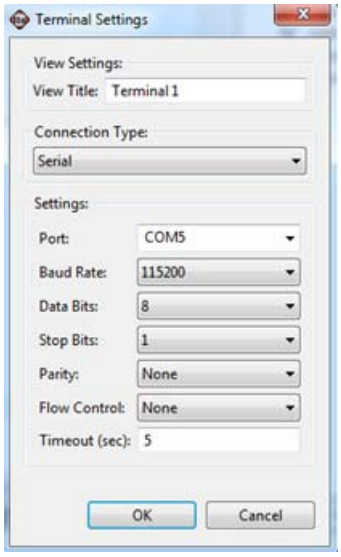


Figure 4-24: Terminal Settings

Running a Software Application on the Board

Running an application from SDK on the board involves creating a Run Configuration. The following steps describe how to set up a Run configuration and execute the code on the board.

35. Right click on the software project (hello_world_0)
36. Click on Run As

37. Click Run Configurations (Figure 4-25)
38. In the Run Configurations screen, double click on Xilinx C/C++ ELF (Figure 4-26)
39. Accept the defaults (Figure 4-27)
40. Click Run

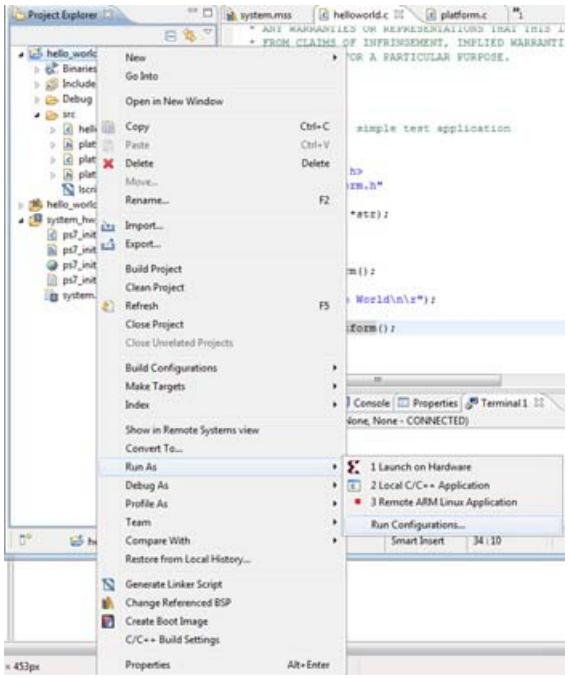


Figure 4-25: Run Configurations

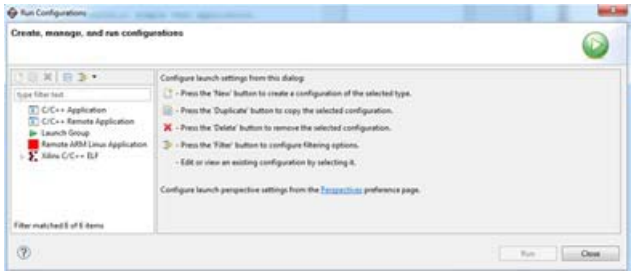


Figure 4-26: Run Configurations

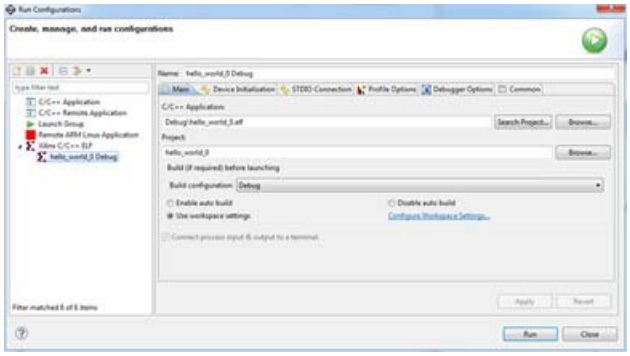


Figure 4-27: Run Configurations Defaults

If everything worked correctly, the Console tab should have the same information messages as Figure 4-28. Also, the Terminal Tab should be the same as Figure 4-29.

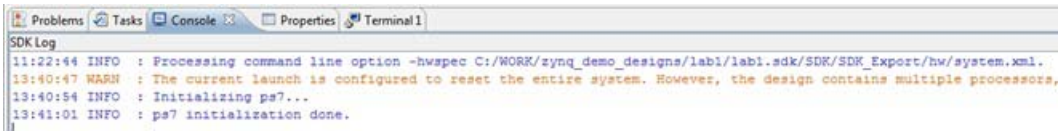


Figure 4-28: Console Output on Successful Execution of Hello World

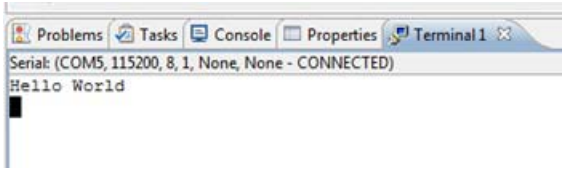


Figure 4-29: Terminal Output for Hello World

Exercise 2: Instantiating High-Level Syntheis Generated IP in a Zynq System

This exercise takes the basic Zynq system developed in Exercise 1 and adds an HLS generated IP block. For this exercise, the I/O of the design will be based on AXI4-Lite to communicate with the processor. A block diagram of the system in this exercise is shown in Figure 4-30.

Figure 4-30: Exercise 2 System

Software Application for AutoESL

An HLS design project is made of 2 software components: a testbench and the code which will be transformed into hardware by the tool. The "exercise2/autoesl_code" directory included with these exercises contains the software files for this project: adders_test.cpp and adders.cpp

The header file adders.h is shown in Figure 4-31. The testbench file adders_test.cpp is shown in Figure 4-32. This file follows the recommended HLS approach of separating the testbench code from the code targeted for hardware implementation. This allows a simple way of exercising the same hardware function with different testbenches and code reuse in other hardware projects. The other thing to notice is that the testbench file is self-checking. HLS requires the testbench to issue a return value of 0 if the functionality is correct and any non-zero value if there is an error. By checking the output of the software implementation against the hardware implementation of adders, the user can be certain that the generated hardware is correct. Another approach to generating checking testbenches is to have known good data files containing the expected result of the hardware function.

The version of adders, which will be implemented in hardware, is shown in Figure 4-33. This code is the same code as the software version of adders and adds the interface definition macros. For this example, the I/O of the generated hardware module will connect to the AXI4-Lite interface and communicate with the processor. In addition to the interfaces specified in the C code, this design will generate an interrupt signal for the ARM processor in Zynq. The interrupt will be issued every time the hardware implementation of adders has completed the function.

```
#include <stdio.h>
#include <stdlib.h>

int adders_hw(int in1, int in2, int *sum);
```

Figure 4-31: **Adders Header File**

```
#include "adders.h"

int adders_sw(int in1, int in2, int *sum)
{
  int temp;
  *sum = in1 + in2 + *sum;
  temp = in1 + in2;
  return temp;
}

int main()
{
  int in1;
  int in2;
  int sum_sw, sum_hw;
  int return_sw, return_hw;

  in1 = 5;
  in2 = 20;
  sum_sw = sum_hw = 50;

  //Call to software model of adders
  return_sw = adders_sw(in1, in2, &sum_sw);

  //Call to hardware model of adders
  return_hw = adders_hw(in1, in2, &sum_hw);

  if((return_sw == return_hw) && (sum_sw == sum_hw)){
    printf("Success both SW and HW models match\n");
    return 0;
  }else{
    printf("Error: return_sw = %d return_hw = %d
      sum_sw = %d  sum_hw = %d\n",
                    return_sw,return_hw,sum_sw,sum_hw);
    return 1;
}
}
```

Figure 4-32: **Adders Testbench Code**

```
#include "adders.h"
#include "ap_interfaces.h"

int adders_hw(int in1, int in2, int *sum){
AP_INTERFACE(in1,ap_none);
AP_INTERFACE(in2,ap_none);
AP_INTERFACE(sum,ap_none);
AP_BUS_AXI4_LITE(in1,ADDER_CONTROL);
AP_BUS_AXI4_LITE(in2,ADDER_CONTROL);
AP_BUS_AXI4_LITE(sum,ADDER_CONTROL);
AP_CONTROL_BUS_AXI(ADDER_CONTROL);
```

```
int temp;

  *sum = in1 + in2 + *sum;
  temp = in1 + in2;

  return temp;
}
```

Figure 4-33: **Adders Code for Hardware Generation**

Create a Project in High-Level Synthesis for the Adders Application

The following steps will demonstrate how to run HLS and create the adders application as a hardware peripheral for the ARM processor.

To invoke HLS, through the Windows menu: **Start** > **All Programs** > **Xilinx Design Tools** > **Vivado 2012.4** > **Vivado HLS GUI**. The Vivado HLS group is shown in Figure 4-34. The Vivado HLS GUI is shown in Figure 4-35.

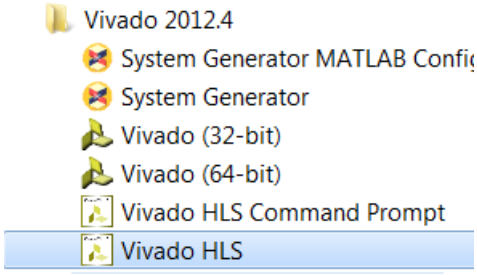


Figure 4-34: **Launching the Vivado HLS GUI**

Figure 4-35: Vivado HLS GUI

1. Click Create New Project
2. Set the project name to adders_prj (Figure 4-36)
3. Set the location the location of exercise 2 and click next
4. Set the top function to adders_hw and add the file adders.cpp (Figure 4-37) and click next.

5. Add the file adders_test.cpp and click next (Figure 4-38)

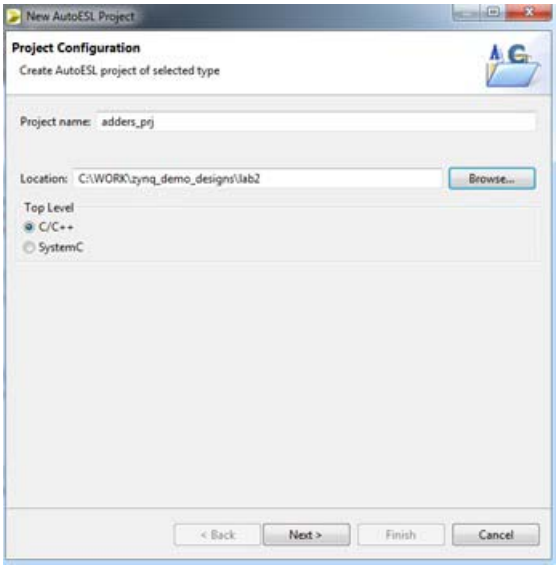


Figure 4-36: Project Configuration

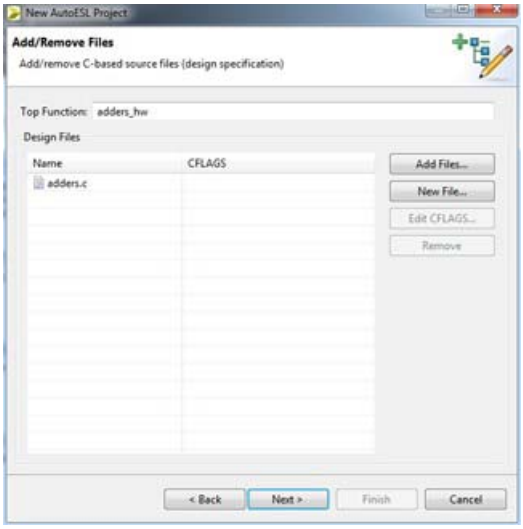


Figure 4-37: Add Files for Hardware Synthesis

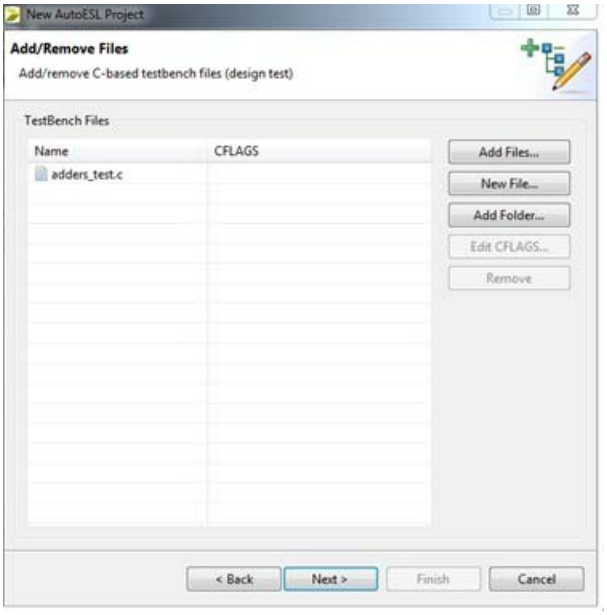


Figure 4-38: Add Testbench Files

6. Set the clock period to 5 (Figure 4-39)
7. Click on part selection to set the FPGA target (Figure 4-39)
8. Set the pact to xc7z020clg484-1 and click ok (Figure 4-40)
9. Click finish the HLS GUI should look like Figure 4-41

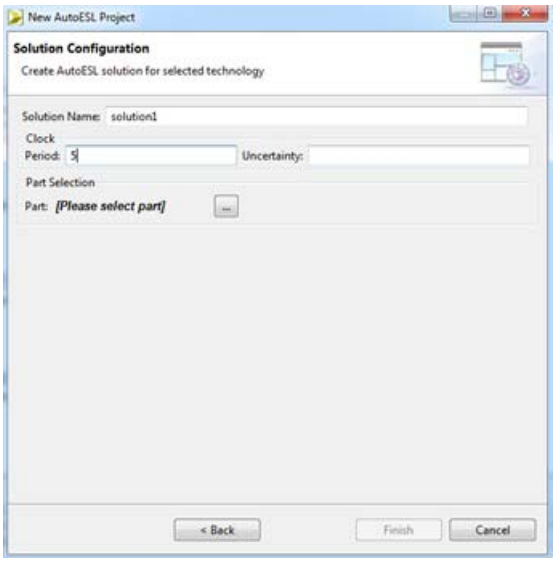


Figure 4-39: Solution Configuration

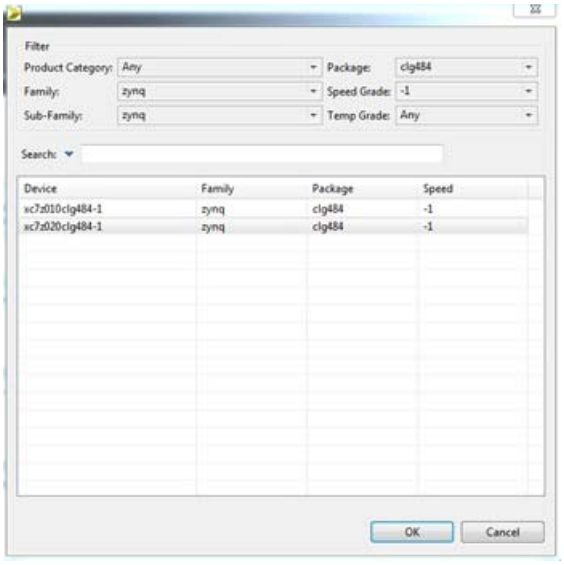


Figure 4-40: Part Selection

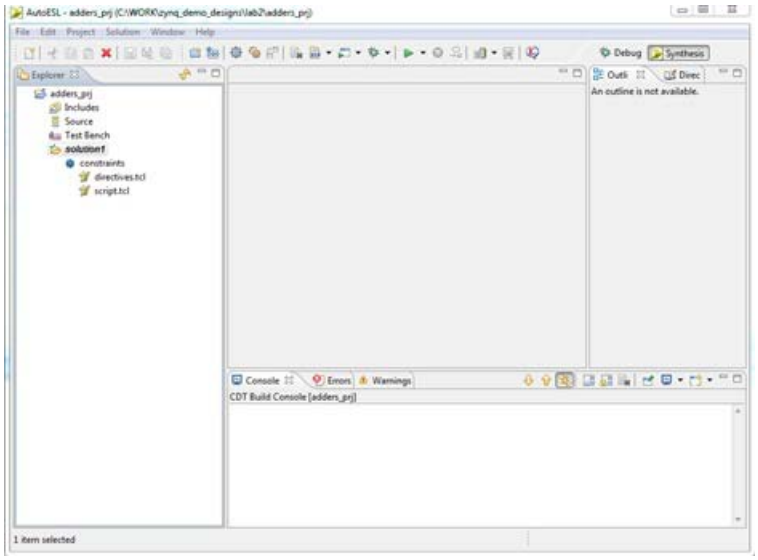


Figure 4-41: HLS GUI after Project Creation

The first step in generating a hardware block with HLS is to check the correctness of the C code. This can be done within HLS using the software build and run commands. The steps to verify C code works correctly are

10. Click the software build icon (Figure 4-42)
11. Click the software run icon (Figure 4-43)

12. Select adders_prj.Debug and click ok. Console should look like Figure 4-44

Figure 4-42: Build Icon

Figure 4-43: Run Icon

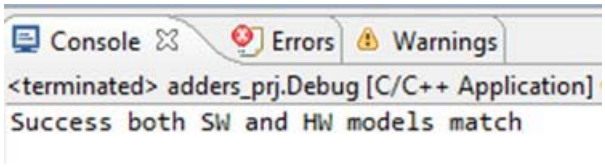


Figure 4-44: Expected Console Output

Once the C code is known to be correct, it is time to generate the hardware peripheral core for the processor. The following steps describe how to accomplish this task.

13. Click the synthesis icon (Figure 4-45)
14. Click the implementation icon (Figure 4-46)
15. Select setup only and generate pcore (Figure 4-47)
16. Click ok
17. Check the console for successful execution (Figure 4-48)
18. If step 17 was successful, exit HLS

Figure 4-45: Synthesis Icon

Figure 4-46: Implementation Icon

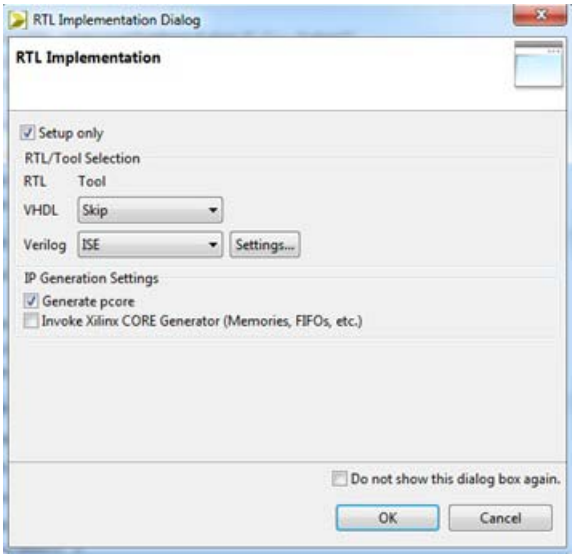


Figure 4-47: RTL Implementation Dialog

```
@I [IMPL-8] Starting RTL implementation using ISE ...
@I [IMPL-3] 'autoimpl' file generation finished.
@I [IMPL-1] 'autoimpl' done successfully.
@I [LIC-101] Checked in features [ AUTOESL_FLOW AUTOESL_OPT AUTOESL_XILINX ]
```

Figure 4-48: Successful Execution of Implementation Stage

Connecting the High-Level Synthesis Block into the Processor Sub-System

19. Create a basic Zynq processor sub-system. The steps to create the system are in exercise 1 steps 1.1 to 1.19. Do not continue Exercise 1 after step 1.19

Before continuing work in XPS, the HLS pcore should be placed in the local repository of Exercise 2. The steps to place the HLS peripheral in the local repository for the processor sub-system are

20. Copy the HLS pcore directory adders_hw_top_1_00_a. This directory is located in exercise2 > adders_prj > solution1 > impl > pcores
21. Place adders_hw_top_1_00_a into the pcore directory of the sub-system project. The pcore directory is located in exercise2 > exercise2_hw_prj > exercise2_hw_prj.srcs > sources_1 > edk > system > pcores

After step 2.21, the sub-system user repository must be rescanned to recognize the HLS peripheral. This step makes the core generated in HLS available for this hardware project. The steps to rescan the user repository are

22. Return to the XPS screen.
23. Click Project
24. Click Rescan User Repositories (Figure 4-49)
25. Check the IP Catalog for the HLS core. It will be listed in Project Local PCores > USER (Figure 4-50)

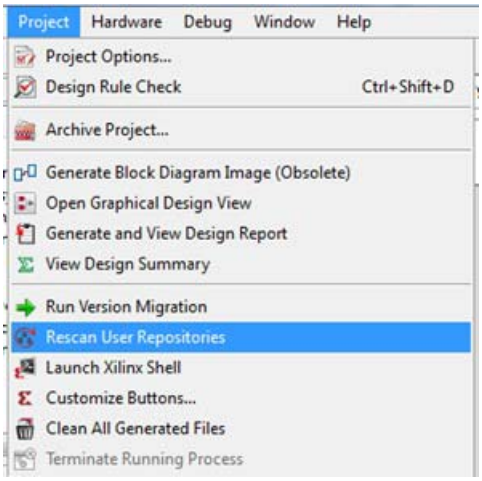


Figure 4-49: Rescan User Repositories

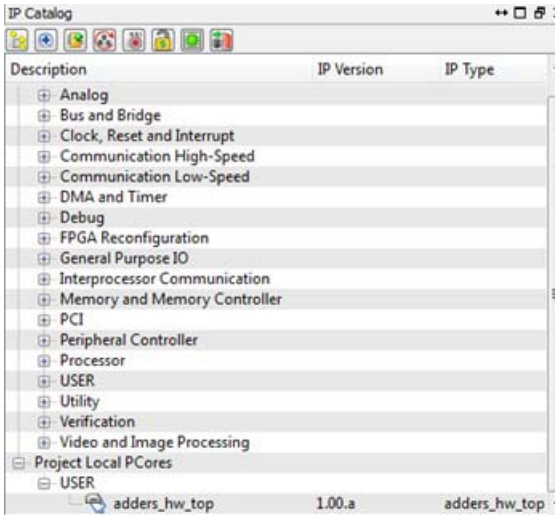


Figure 4-50: Project Local PCores

Once the HLS IP is recognized by XPS, you are ready to add the peripheral to the system. The steps for adding the hardware block to the sub-system are

26. Double click on adders_hw_top from the IP Catalog (Figure 4-50)
27. Click Yes when asked if you want to add adders_hw_top to the design.
28. Click Ok to accept default values
29. Select processor instance to processing_system7_0 and click Ok (Figure 4-51)
30. The resulting Bus Interfaces Tab should look like Figure 4-52

Figure 4-51: Instantiate and Connect Dialog

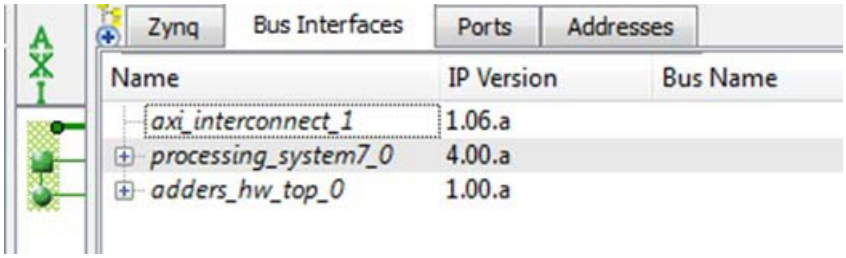


Figure 4-52: Bus Interfaces after Adding HLS Block

Connecting the HLS generated IP to the AXI4-Lite bus interface does not complete the hardware connection. For the adders IP block, three more ports must be connected: SYS_CLK, SYS_RESET and interrupt. At this time, HLS generates two clock signals for a pcore ACLK for the AXI4 interface and SYS_CLK for the accelerator part. Both of these clock signals must be connected to the same clock source. Before connecting the clock signals, the ports tab in XPS should look like Figure 4-53.

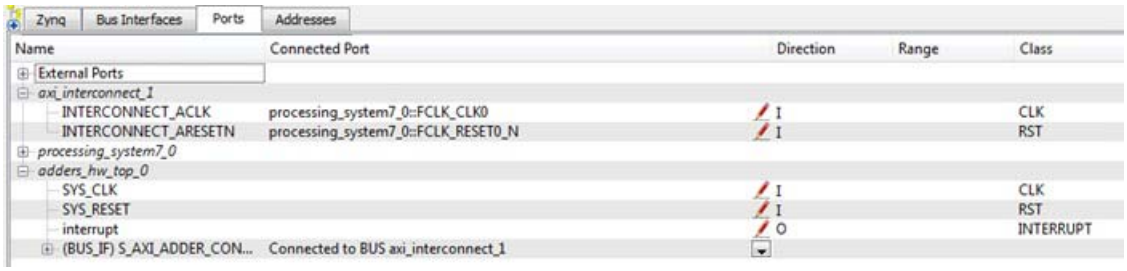


Figure 4-53: Ports Tab before Signal Connection

The final connections for the HLS generated IP are described in the following steps.

31. Click on the pencil next to SYS_CLK in adders_hw_top_0.
32. Select processing_system7_0 in the left box (Figure 4-53)
33. Select FCLK_CLK0 in the right box (Figure 4-53)
34. Click the pencil next to SYS_RESET
35. Select processing_system7_0 in the left box (Figure 4-54)
36. Select FCLK_RESET0_N in the right box (Figure 4-55)

Figure 4-54: Setting SYS_CLK Signal

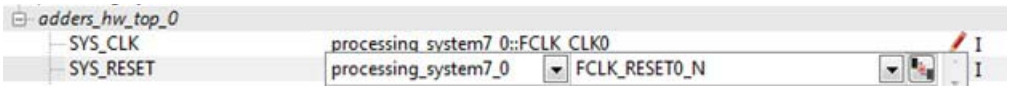


Figure 4-55: Setting SYS_RESET Signal

37. Click on the interrupt signal. This will open the Interrupt Connection Dialog (Figure 4-56)
38. Click the right arrow to connect the interrupt (Figure 4-57)
39. Click OK
40. Check that the Ports tab looks like Figure 4-58
41. Close XPS

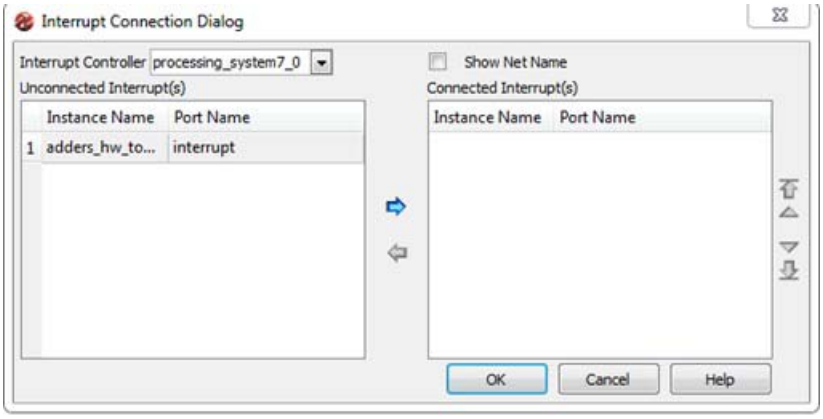


Figure 4-56: Interrupt Connection Dialog

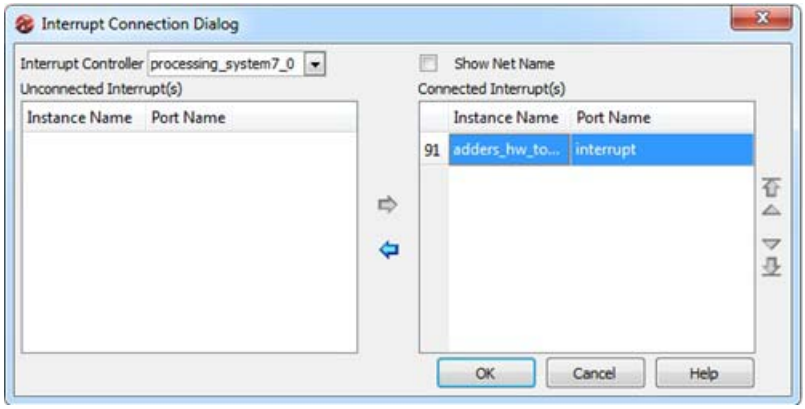


Figure 4-57: Connected Interrupt Signal

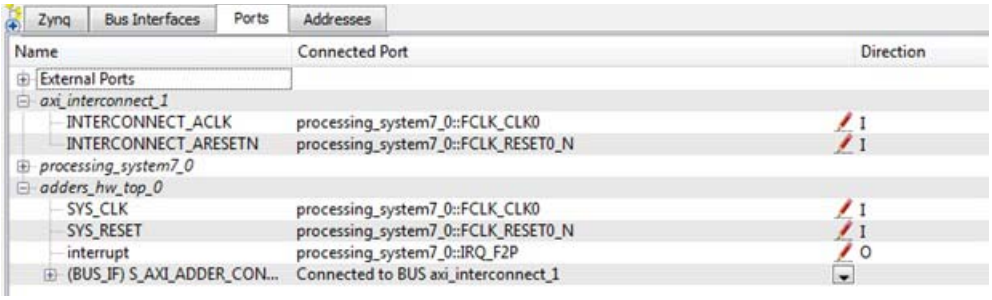


Figure 4-58: Complete Ports Tab

42. Create a top level RTL module in PlanAhead. Instructions for this step are in Exercise 1 steps 1.27 to 1.28
43. Click "Generate Bitstream" Figure 4-59. This will create the necessary FPGA fabric programming file.

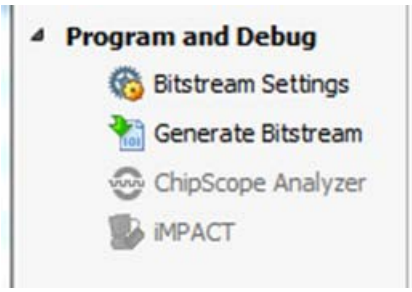


Figure 4-59: Generate Bitstream

At the end of bitstream generation, a message window will be displayed to indicate success (Figure 4-60).

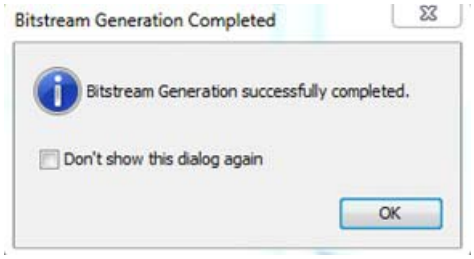


Figure 4-60: Bitstream Generation Complete Dialog

44. Export the hardware platform to SDK. In the export hardware dialog, make sure to select the bitstream (Figure 4-61). Instructions on how to export a hardware platform to SDK are in Exercise 1 steps 1.27 to 1.33

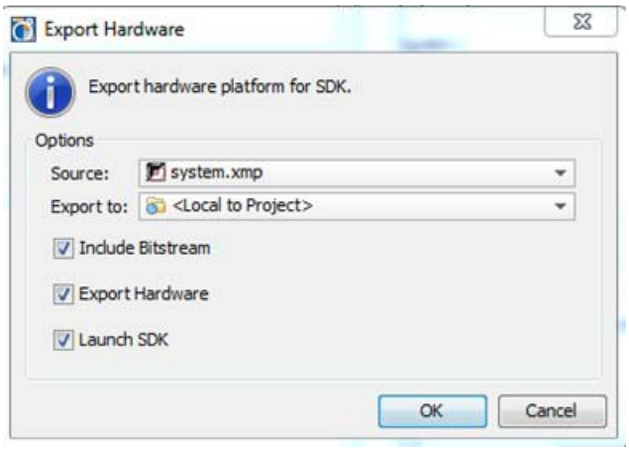


Figure 4-61: Export Hardware Configuration

Creating the Software Application for the Adders Design

45. Create a basic hello_world application in SDK. The instructions on how to create this application are in Exercise 1 steps 1.29 to 1.31
46. Click the FPGA program icon Figure 4-62
47. Accept default values and click program in the programming dialog (Figure 4-63)

Figure 4-62: FPGA Program Icon

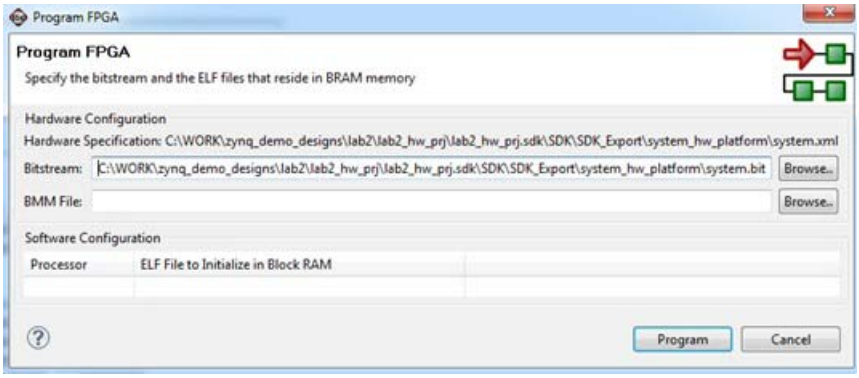


Figure 4-63: FPGA Programming Dialog

After programming the FPGA, check that the Hello World application runs correctly on the system. Instructions on how to run a software application Zynq from XPS are in steps 1.35 to 1.40 of Exercise 1.

The first step in transforming the Hello World application into a program using the adders block in the FPGA fabric is to include the HLS generated drivers into the software project. The steps to add the software drivers are

48. Right click on src in the hello_world_0 project (Figure 4-64)
49. Select import (Figure 4-65)
50. Select General
51. Select File System and click next (Figure 4-66)
52. Select the include directory of the HLS generated pcore. This will be in exercise2_hw_prj\exercise2_hw_prj.srcs\sources_1\edk\system\pcores\adders_hw_top_v1_00_a\include

53. Select all files and click finish (Figure 4-66)

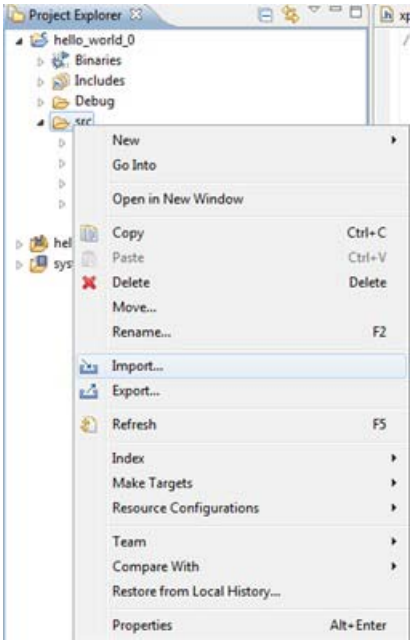


Figure 4-64: File Import

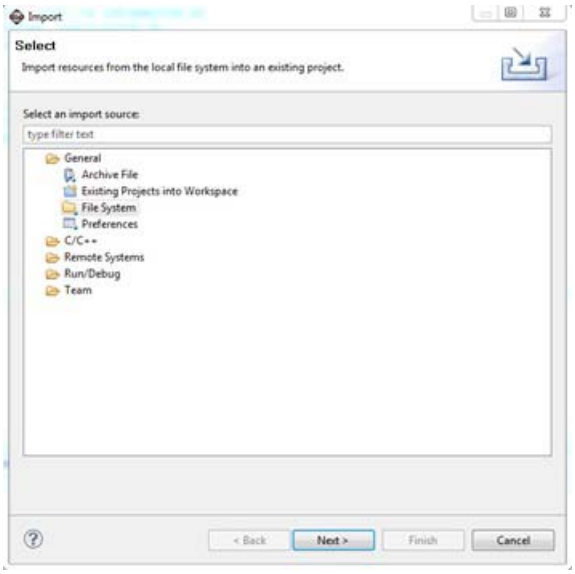


Figure 4-65: Select Import Source Type

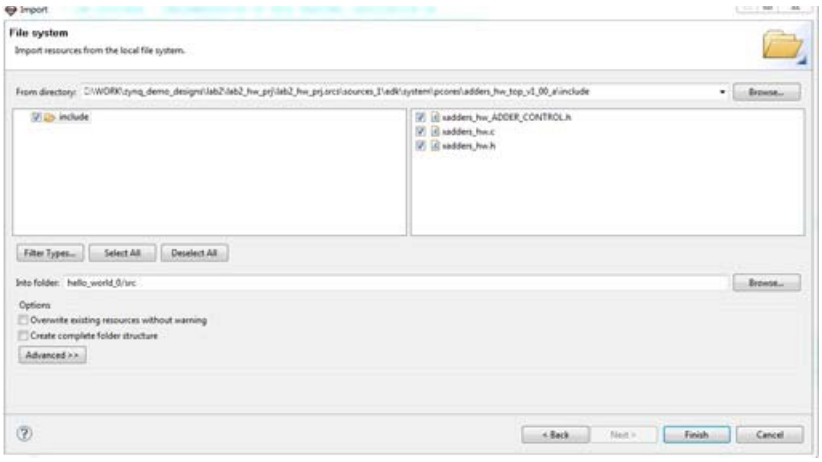


Figure 4-66: Selection of HLS Driver Files

Now that the driver files for the HLS generated IP have been added to the SDK project it is time to start modifying the Hello World application. In the include files of the hello_world.c template file, the following includes need to be added:

```
#include "xparameters.h"
#include "xscugic.h"
#include "xadders_hw.h"
```

Figure 4-67: **Required Header Files**

The purpose of each header file is

Table 4-1: **Header File Purpose**

File	Purpose
xparameters.h	Processor peripheral header file
xscugic.h	Processor interrupt controller header file
xadders_hw.h	HLS driver header file

In the "xparameter.h" file, the user needs to locate the base address assigned to the HLS IP. The file is located in hello_world_bsp_0 > ps7_cortexa9_0 > include (Figure 4-67).

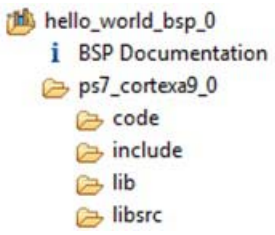


Figure 4-68: Location of xparameters.h

The required parameters for the HLS IP are shown in Figure 4-68.

```
#include "xparameters_ps.h"

#define STDIN_BASEADDRESS 0xE0001000
#define STDOUT_BASEADDRESS 0xE0001000

/******************************************************************/


/* Definitions for peripheral ADDERS_HW_TOP_0 */
#define XPAR_ADDERS_HW_TOP_0_S_AXI_ADDER_CONTROL_BASEADDR 0x73A00000
#define XPAR_ADDERS_HW_TOP_0_S_AXI_ADDER_CONTROL_HIGHADDR 0x73A0FFFF
```

Figure 4-69: xparameters.h

After the include files, the following declarations are required in the global memory space of the program:

```
extern XAdders_hw adder;
volatile static int RunAdder = 0;
volatile static int NewResult = 0;
XScuGic ScuGic;
XAdders_hw adder;
Xadders_hw_Config adder_config = {
  0,
  XPAR_ADDERS_HW_TOP_0_S_AXI_ADDER_CONTROL_BASEADDR
};
```

Figure 4-70: **Global Declarations for Interrupt Controller and HLS IP**

The value of RunAdder is used later in the interrupt service routine to determine if the processor should automatically restart the hardware module or not. The NewResult value is used to inform the processor of a result from the hardware module. In addition to these variables, there is a definition of a structure for the interrupt controller "ScuGic" and for the adder module in the FPGA fabric. The adder module uses the variable "adder_config" to set the configuration of a specific instance of the adder module. The define value in adder_config is obtained from the xparameters.h file in Figure 4-69. The adder instance is completed by the "adder" variable which contains state and configuration information.

Before the main function is modified to use the adder in the FPGA fabric, a set of helper functions are required to set up the software infrastructure for this design. These functions are examples of what is required in a typical application using both the processor and the FPGA fabric.

```
int AdderSetup(){
  return XAdders_hw_Initialize(&adder,&adder_config):
}
```

Figure 4-71: **Adder Setup**

The AdderSetup function in Figure 4-71 sets the address map data on the adder instance for the processor to access the core.

```
void AdderStart(void *InstancePtr){
  XAdders_hw *pAdder = (XAdders_hw *)InstancePtr;
  // Enable the Local IP Interrupt
  XAdders_hw_InterruptEnable(pAdder,1);
  // Enable the Global IP Interrupt
  XAdders_hw_InterruptGlobalEnable(pAdder);
  // Start the Adder Core
  XAdders_hw_Start(pAdder);
}
```

Figure 4-72: **AdderStart Function**

The AdderStart function enables the interrupt sources in the HLS generated IP and also starts the module. The enable sequence for the HLS IP is to first enable the local interrupt followed by the global interrupt enable. At this point HLS IP only has the ap_done signal as the only interrupt source. The API has been designed with the flexibility to grow and accommodate multiple interrupt sources from within the HLS generated IP. These additional sources will be available in future versions of HLS. Once the local interrupt and global interrupts are enabled, the XAdders_hw_Start function can be called to initialize the module. In terms of hardware, this last function call is the same as applying a value of 1 to the ap_start signal.

For every HLS hardware module, the processor requires an interrupt service routine. Figure 4-73 shows an example of a typical interrupt service routine for an HLS module.

```
void AdderIsr(void *InstancePtr){
  XAdders_hw *pAdder = (XAdders_hw *)InstancePtr;
  //Disable Global Interrupt
  XAdders_hw_InterruptGlobalDisable(pAdder);
  //Disable Local Interrupt
  XAdders_hw_InterruptDisable(pAdder, 0xffffffff);
  //Clear the local interrupt
  XAdders_hw_InterruptClear(pAdder,1);
  //Set the NewResult flag
  NewResult = 1;
  //Check if the core should be restarted
  if(RunAdder){
    AdderStart(pAdder);
  }
}
```

Figure 4-73: **Adder Interrupt Service Routine**

The AdderIsr has to be registered in the processor exception table to be run. Figure 4-73 shows how to setup interrupts on the ARM and enable the interrupt service routine for the adder core. This is the final infrastructure function required for a hardware module to be used by the ARM processor.

For this exercise, Figure 4-74 shows an example program that provides input to the adder core in the FPGA fabric and then displays the results on the Terminal screen. The example program has the following sections:

```
  "Platform initialization
  "Adder core setup
  "Processor interrupt setup
  "Setting configuration values for the adder core
  "Starting the adder core
  "Waiting for core to finish
  "Displaying results on the terminal
int SetupInterrupt()
{
  int result;
  //Search for interrupt configuration table
  XScuGic_Config *pCfg = XScuGic_LookupConfig(XPAR_SCUGIC_SINGLE_DEVICE_ID);
  if (pCfg == NULL){
    print("Interrupt Configuration Lookup Failed\n\r");
    return XST_FAILURE;
  }
  // Initialize the interrupt controller
  result = XScuGic_CfgInitialize(&ScuGic,pCfg,pCfg->CpuBaseAddress);
  if(result != XST_SUCCESS){
    return result;
  }
  // self test the interrupt controller
  result = XScuGic_SelfTest(&ScuGic);
  if(result != XST_SUCCESS){
    return result;
  }

  // Initialize the exception handler
  Xil_ExceptionInit();

  // Register the exception handler

Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,(Xil_ExceptionHandler)XScuGic_Int
erruptHandler,&ScuGic);

  //Enable the exception handler
  Xil_ExceptionEnable();

  // Connect the Adder ISR to the exception table
  result =
XScuGic_Connect(&ScuGic,XPAR_FABRIC_ADDERS_HW_TOP_0_INTERRUPT_INTR,(Xil_InterruptHa
ndler)AdderIsr,&adder);
  if(result != XST_SUCCESS){
    return result;
  }
  XScuGic_Enable(&ScuGic,XPAR_FABRIC_ADDERS_HW_TOP_0_INTERRUPT_INTR);
  return XST_SUCCESS;
}
```

Figure 4-73: **Processor Interrupt Initialization Routine**

```
int main(){
  int sum_result, adder_return;
```

```
  int status;

  //Initialize software platform
  init_platform();

  //Setup the Adder core
  status = AdderSetup();
  if(status != XST_SUCCESS){
    print("Adder Setup Failed\n\r);
}

//Set up the Interrupt
status = SetupInterrupt();
if(status != XST_SUCCESS){
  print("Interrupt Setup Failed\n\r");
}

//Set configuration values for the adder core
XAdders_hw_SetIn1(&adder,10);
XAdders_hw_SetIn2(&adder,5);
XAdders_hw_SetSum_i(&adder,20);

//Start the adder core
AdderStart(&adder);

//Wait for adder core to finish
while(!NewResult);

//Get result values from the adder core
adder_return = XAdders_hw_GetReturn(&adder);
sum_result = XAdders_hw_GetSum_o(&adder);

//Display results on the terminal window
printf("adder_return = %d sum_result = %d\n\r",adder_return,sum_result);

//Disable the system
cleanup_platform();
return 0;
}
```

Figure 4-74: **Example Main Program for Adder Core**

Compile and execute the software application on the board. The steps for setting up the terminal and running the application on the board are in Exercise 1 steps 1.35 to 1.40. The results of the program should look like Figure 4-75.

```
adder_return = 15  sum_result = 35
```

Figure 4-75: **Results of Adder Application**

Exercise 3: Debugging the Communication between High-Level Synthesis IP and ARM

This exercise is based on the hardware system developed in Exercise 2 to demonstrate how the communication between the processor and the core in the FPGA fabric can be debugged using the AXI monitor IP. The system for this exercise is shown in Figure 4-76. To start this exercise, create a exercise3 folder in your working directory.

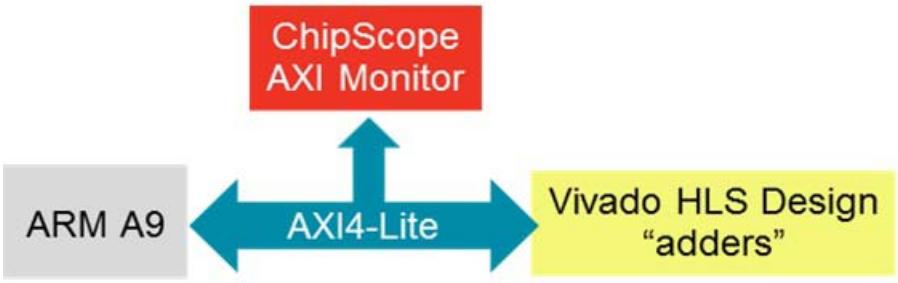


Figure 4-76: Exercise 3 Hardware System

Adding the AXI Monitor IP

1. Start PlanAhead and recreate the system of exercise2 for exercise3. This will give you a clean working environment to test the AXI monitor. Follow steps 1.1 to 1.19 in Exercise 1. The result should look like Figure 4-77.
2. Follow steps 2.19 to 2.40 in Exercise 2.
3. Click on **Debug** > **Debug Configuration** to enable debugging of the AXI interface between the processor and the core (Figure 4-78).

4. In the Debug Configuration screen click on Add ChipScope Peripheral (Figure 4-79).

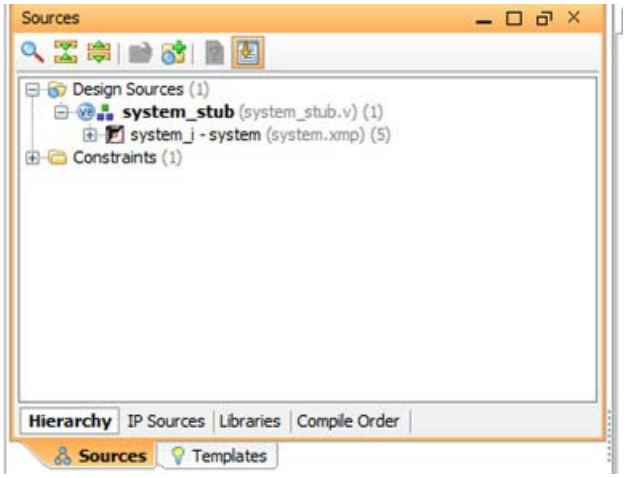


Figure 4-77: XPS Sources Window

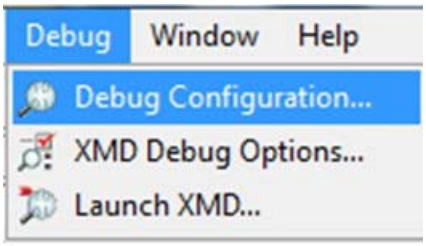


Figure 4-78: Debug Configuration

Figure 4-79: Debug Configuration - Add ChipcScope Peripheral

5. Select adding AXI Monitor and click Ok (Figure 4-80)
6. Set the bus to adders_hw_top_0.S_AXI_ADDER_CONTROL in the drop box to select which bus signals to monitor (Figure 4-81)
7. Click Ok

8. The Bus Interfaces tab in XPS should look like Figure 4-82

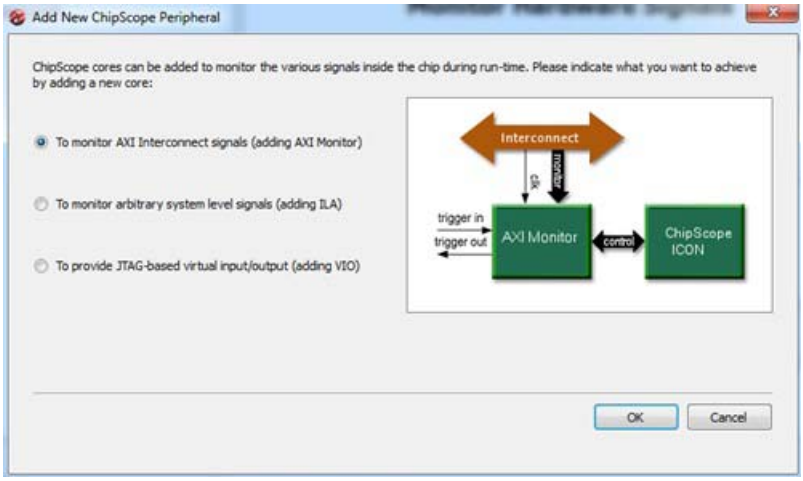


Figure 4-80: Add AXI Monitor

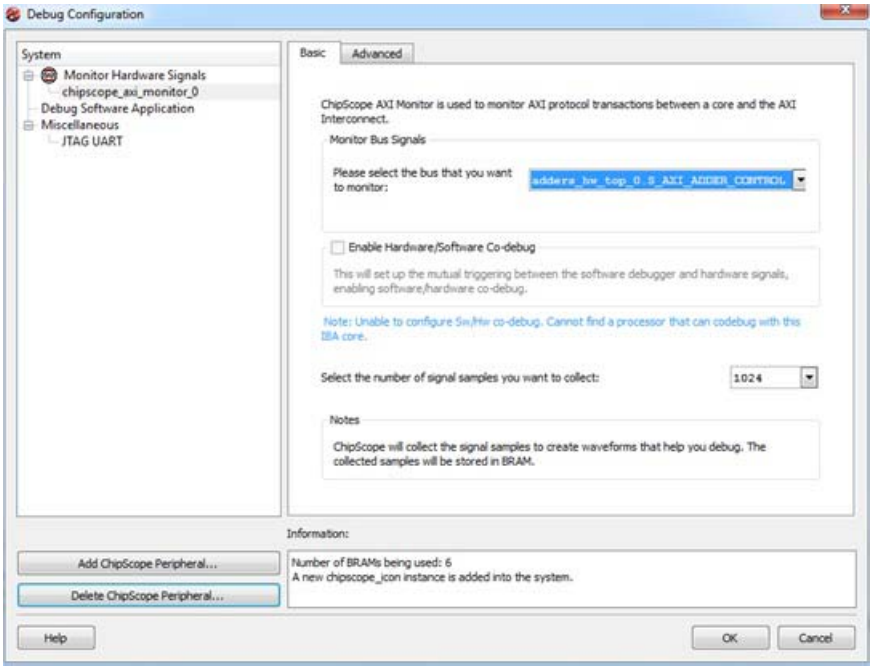


Figure 4-81: Monitor Bus Configuration

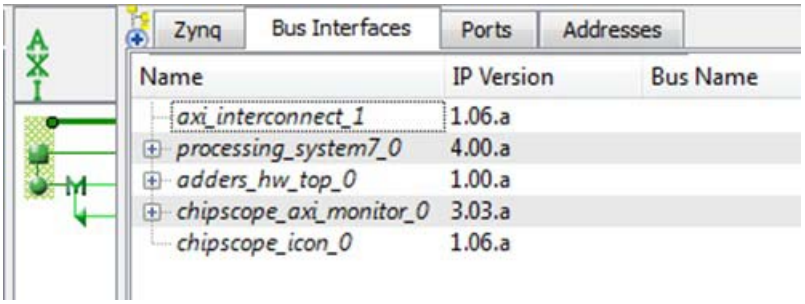


Figure 4-82: Bus Interfaces Tab

9. Close XPS
10. Create a top level RTL file - steps 1.22 and 1.23 in Exercise 1
11. Generate a bitstream for the system Figure 2.30.
12. Export the hardware system and bitstream to SDK - steps 1.24 to 1.28 in Exercise 1
13. Create the basic hello_world application - steps 1.24 to 1.31 in Exercise 1
14. Program the FPGA - steps 2.46 and 2.47 of Exercise 2

Setting the Software Application for Debugging

The software application for this exercise will reuse the software developed in Exercise 2. For this case, import the entire hello_world application from Exercise 2. The steps are

15. Right-click on hello_world_0 project folder and select import
16. Select **General** > **File System** (Figure 4-83)
17. Select the location for the files in the exercise2 directory, the files are in exercise2\exercise2_hw_prj\exercise2_hw_prj.sdk\SDK\SDK_Export\hello_world_0\src
18. Select the following files for import: helloworld.c, xadders_hw.c, xadders_hw.h, xadders_hw_ADDER_CONTROL.h
19. Click overwrite existing resources and click Finish (Figure 4-84)
20. After the import process, there will be a conflict between the original helloworld.c and the version from Exercise2. Delete the helloworld.c in hello_world_0\src

21. Run the application on the board. The results to the terminal should be the same as in Exercise 2.

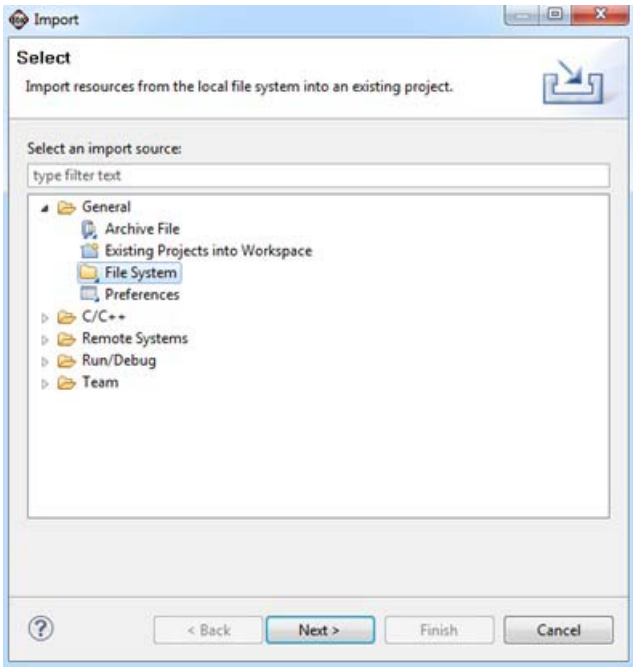


Figure 4-83: File System Import

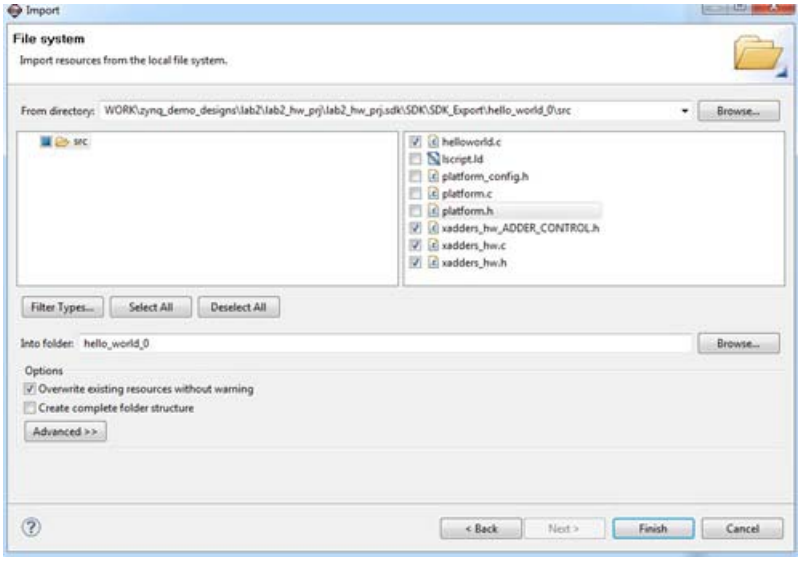


Figure 4-84: File Import Selection

Debugging the connection between the processor and the Adder core will combine the use of the software source-level debugger (part of Xilinx SDK) and ChipScope Pro. Chipscope is a logic analyzer tool that will capture traces of transactions between the processor and

accelerator. One way of making the transactions easier to view in Chipscope is to pause the execution of the software application after each interesting interaction. The simplest way to pause the application is to run the application in the debugger and set breakpoints at points of interest in the source code. When encountered during a debug session the application will pause before the execution of the statement(s) at the breakpoint.

For this Exercise, breakpoints will be set in the following functions and hardware signal activity will be captured and viewed in ChipScope

```
"AdderStart()
"AdderIsr()
```

A debugger breakpoint can be set at a statement in the source code in several ways: by placing the editor cursor anywhere on the line containing the statement and then selecting the "Run > Toggle Breakpoint" menu item; by pressing Ctrl+Shift+B on the keyboard; by double-clicking in the left-hand margin directly adjacent to the line of interest.

To run an application under the control of the debugger make sure it is selected in the "Project Explorer" pane then start it by selecting the "Run > Debug" menu item, pressing the "F11" key or clicking the "Debug" icon on the toolbar. This will cause the SDK GUI to switch into the Debug perspective and by default execution will pause at the application entry point, i.e. before the first statement in main().

Execution can be advanced to subsequent breakpoints by selecting the "Run > Resume" menu item, pressing the "F8 key" or clicking the "Resume" toolbar icon.

22. Set breakpoints at the lines shown in Figure 4-85
23. Start a debug session for the "hello_world_0" application
24. Click OK on "Reset Status" dialog that appears
25. Click YES to accept the change to the Debug perspective

26. The debug session should now be active and look similar to Figure 4-86

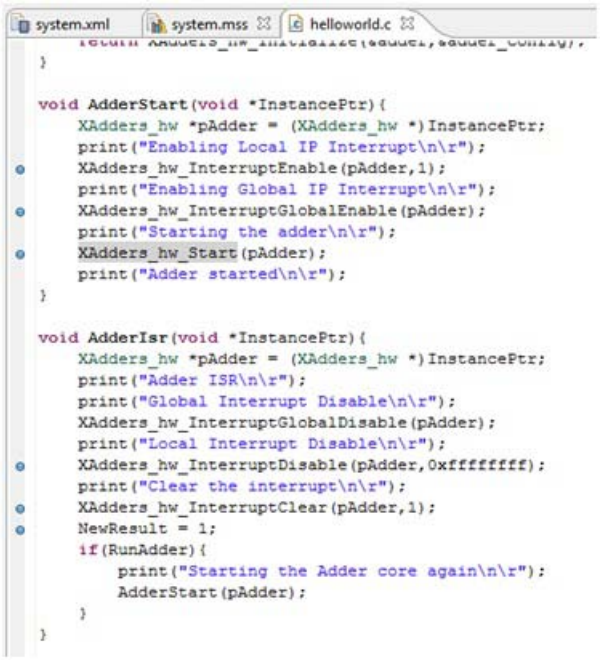


Figure 4-85: Breakpoint Locations

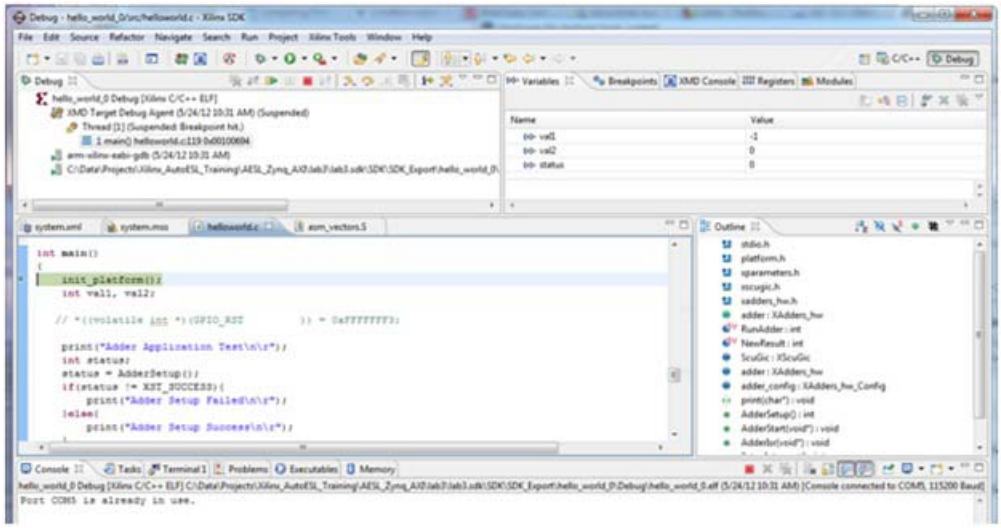


Figure 4-86: Active Debug Session

Using ChipScope Pro

The changes to the software project presented above are the only changes needed to easily view the processor to adder core transactions in ChipScope. To invoke ChipScope, through the Windows menu: Start ' All Programs ' Xilinx Design Tools ' ChipScope Pro ' ChipScope

64-bit ' Analyzer. The ChipScope group is shown in Figure 4-87. Also, the choice of 64-bit or 32-bit versions of the tool depends on the host computer. The main ChipScope screen is shown in Figure 4-88.

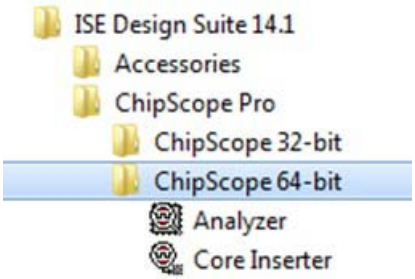


Figure 4-87: ChipScope Group

Figure 4-88: Main ChipScope Screen

27. Click the JTAG chain icon (Figure 4-89) to connect ChipScope to the hardware system
28. A successful execution of the previous step will show 2 devices in the Zynq JTAG chain: ARM_DAP and XC7Z020 (Figure 4-90)
29. Click OK
30. The basic setup is complete and ChipScope should look like Figure 4-91

Figure 4-89: JTAG Chain Icon

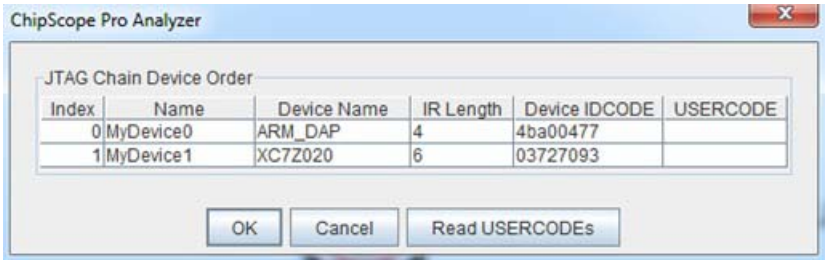


Figure 4-90: JTAG Chain Devices

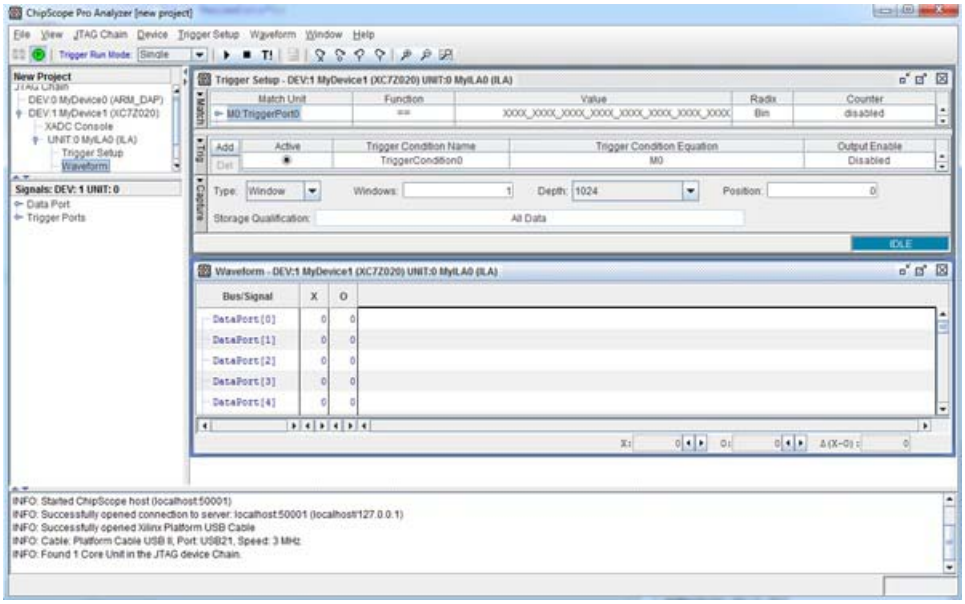


Figure 4-91: ChipScope Screen after JTAG Chain Detection

At this stage, ChipScope has detected the AXI monitor in the FPGA fabric and is ready to capture signal traces. The one thing missing is the signal names in order to be able to understand the information displayed by the tool. The signal name information is stored in a *.cdc file, which was automatically created during the creation of the hardware platform. The steps to import this file are

31. Click **File** > **Import** (Figure 4-92).
32. In the signal import dialog, click select new file.
33. Set the file to chipscope_axi_monitor_0.cdc . This file is in exercise3\exercise3_prj\exercise3_prj.srcs\sources_1\edk\system\implementation\chipscope_axi_monitor_0_wrapper.
34. The import dialog should look like Figure 4-93. Click Ok

35. ChipScope should look like Figure 4-94

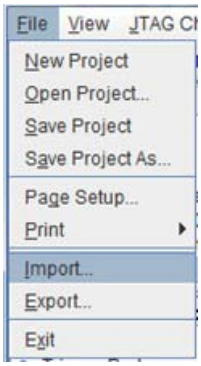


Figure 4-92: File Import

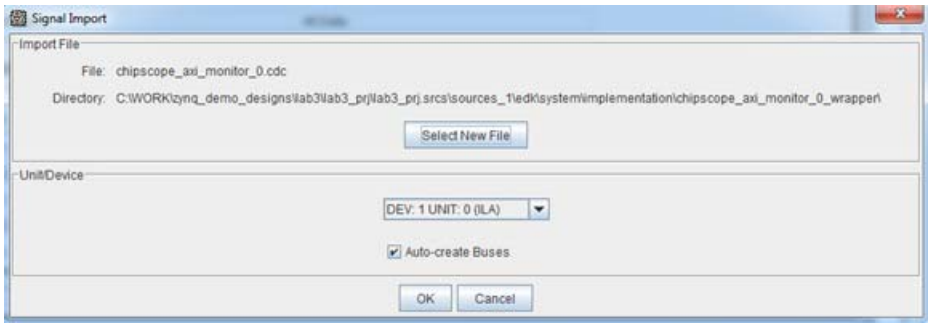


Figure 4-93: Signal Import Dialog

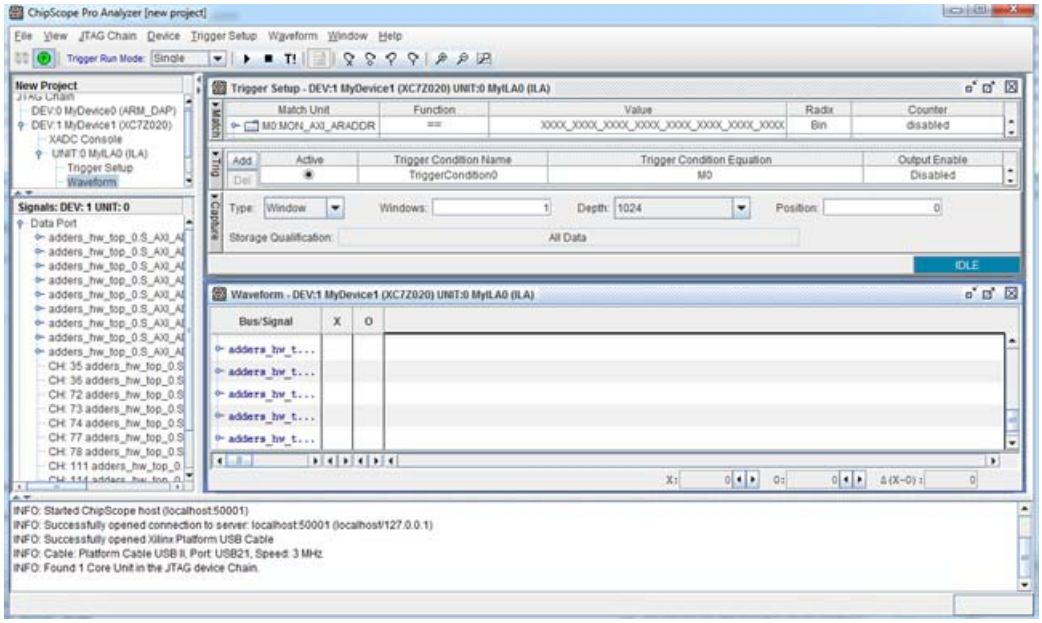


Figure 4-94: ChipScope after Signal Configuration

The next step in using ChipScope is to limit the amount of data captured by using a trigger. In this system, the address base address of the adder core will be used to trigger data capture.

36. In Trigger Setup, click on radix and change it from Bin to Hex for M0 and M2 (Figure 4-95)
37. Change the value of M0 and M2 to 73AX_XXXX, this sets the range to capture all transactions to the adder core. Please check your xparameters.h file in SDK for the correct value (Figure 4-96)

38. Change the Trigger Run Mode from Single to Repetitive (Figure 4-97)

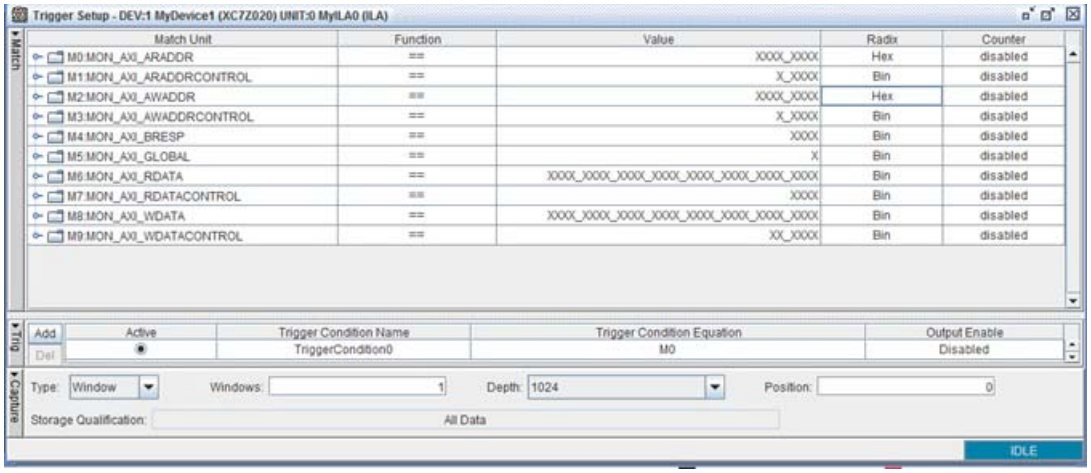


Figure 4-95: Change the Radix from Binary to Hex

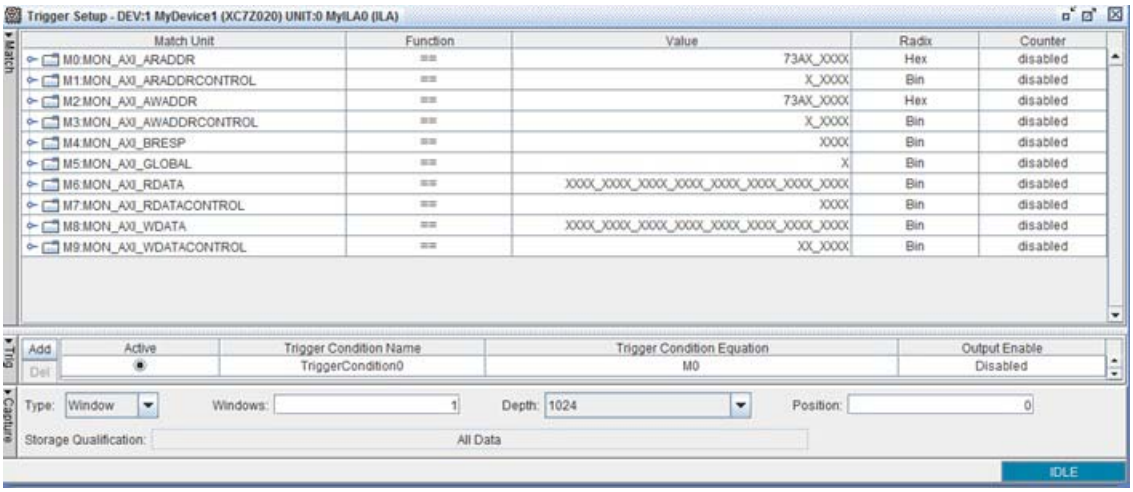


Figure 4-96: Trigger Settings for M0 and M1

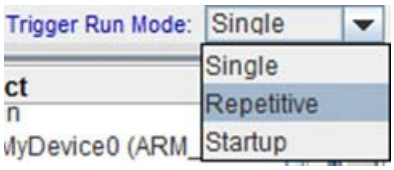


Figure 4-97: Trigger Run Mode

39. Return to the SDK debugger session and "Resume" to the 1st breakpoint by pressing the F8 key

40. The debug session should look like Figure 4-98

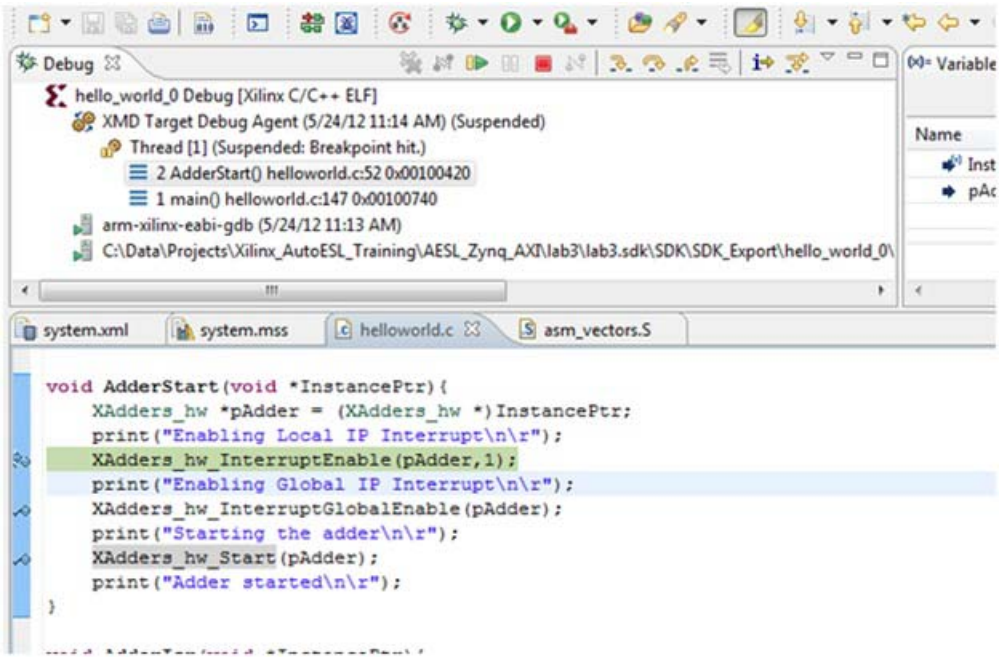


Figure 4-98: Application Paused at 1st Breakpoint

41. Click the Trigger Arm button (Figure 4-99)
42. The first trigger capture should look like the waveform in Figure 4-100

Figure 4-99: Trigger Arm Button

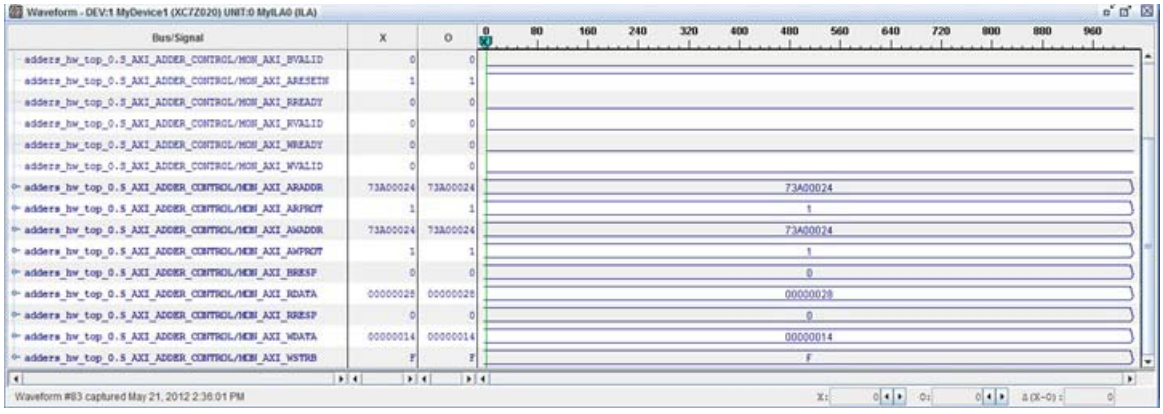


Figure 4-100: Initial Trigger Capture

43. Press F8 in the SDK debug session to continue to the next breakpoint

44. The trigger capture to set the local IP interrupt should look like Figure 3.26
45. Continue stepping through the program by pressing the F8 key and watch the transaction traces in ChipScope.

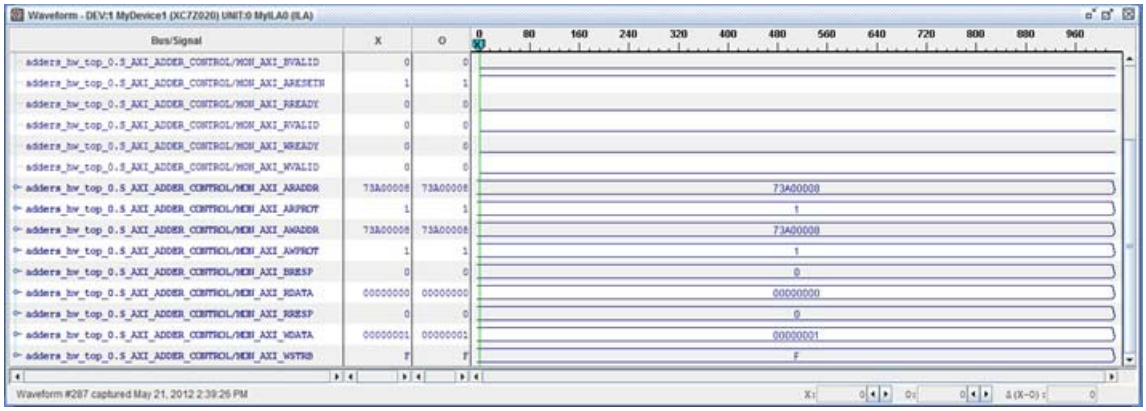


Figure 4-101: Local IP Interrupt Setup ChipScope Waveform

Exercise 4: Connecting 2 High-Level Synthesis IPs with AXI4-streams

This exercise shows how to connect two HLS generated IP blocks with AXI4-streams. AXI4-stream is a communication standard for point-to-point data transfer without the need of addressing or external bus masters. This protocol allows both cores to dynamically synchronize on data using a producer-consumer model. Depending on the implementation of AXI4-stream used, the communication channel can be built as wires or with storage to account for data rate mismatches at each end. The hardware system for Exercise 4 is shown in Figure 4-102.

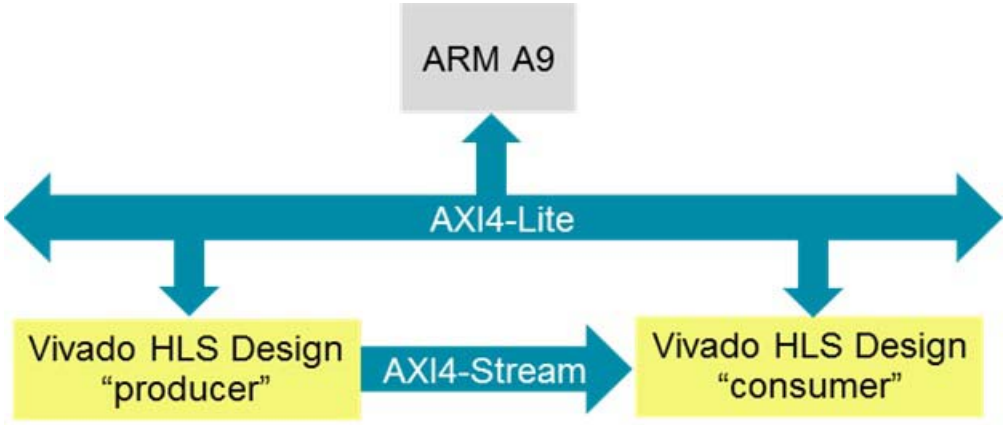


Figure 4-102: Exercise 4 System

High-Level Synthesis Software Projects

For this exercise, two IP blocks will be generated with HLS. The instructions on how to run HLS are described in Exercise 2 steps 2.1 to 2.18. The code for the two hardware blocks is shown in the following figures.

```
#ifndef STRM_TEST_H_
#define STRM_TEST_H_

#define MAX_STRM_LEN 1024

typedef unsigned short strm_param_t;
typedef short strm_data_t;
typedef int data_out_t;

void strm_producer(strm_data_t strm_in[MAX_STRM_LEN], strm_param_t strm_len);
void strm_consumer(data_out_t *dout, strm_data_t strm_in[MAX_STRM_LEN], strm_param_t
strm_len);

#endif // STRM_TEST_H_
```

Figure 4-103: **Figure 4.2 Header File for 2 IP Example**

```
#include <iostream>
#include "strm_test.h"

using namespace std;

int main(void)
{
   unsigned err_cnt = 0;
   data_out_t hw_result, expected = 0;
   strm_data_t strm_array[MAX_STRM_LEN];
   strm_param_t strm_len = 42;

   // Generate expected result
   for (int i = 0; i < strm_len; i++) {
      expected += i + 1;
   }

   strm_producer(strm_array, strm_len);
   strm_consumer(&hw_result, strm_array, strm_len);

   // Test result
   if (hw_result != expected) {
      cout << "!!! ERROR";
      err_cnt++;
   } else {
      cout << "*** Test Passed";
   }
   cout <<  " - expectcted: " << expected;
   cout << "  Got: " << hw_result << endl;
   return err_cnt;
}
```

Figure 4-104: **Figure 4.3 Exercise 4 Testbench**

```
#include "strm_test.h"
#include "ap_interfaces.h"

void strm_producer(strm_data_t strm_out[MAX_STRM_LEN], strm_param_t strm_len)
{
  AP_INTERFACE(strm_len,ap_none);
  AP_BUS_AXI4_LITE(strm_len,CONTROL_BUS);
  AP_CONTROL_BUS_AXI(CONTROL_BUS);
  AP_BUS_AXI_STREAM(strm_out,OUTPUT_STREAM);

   for (int i = 0; i < strm_len; i++) {
#pragma AP PIPELINE
      strm_out[i] = i + 1;
   }
}
```

Figure 4-105: **Figure 4.4 Stream Producer Block**

```
#include "strm_test.h"
#include "ap_interfaces.h"
```

```
void strm_consumer(data_out_t *dout, strm_data_t strm_in[MAX_STRM_LEN], strm_param_t
strm_len)
{
//AP_INTERFACE(dout,none);
//AP_INTERFACE(strm_len,none)
  AP_INTERFACE(dout,ap_none);
  AP_INTERFACE(strm_len,ap_none);
  AP_BUS_AXI4_LITE(dout,CONTROL_BUS);
  AP_BUS_AXI4_LITE(strm_len,CONTROL_BUS);
  AP_CONTROL_BUS_AXI(CONTROL_BUS);
  AP_BUS_AXI_STREAM(strm_in,INPUT_STREAM);
   data_out_t accum = 0;

   for (int i = 0; i < strm_len; i++) {
#pragma AP PIPELINE
      accum += strm_in[i];
   }
   *dout = accum;
}
```

Figure 4-106: **Figure 4.5 Stream Consumer Block**

1. Create an HLS implementation of the strm_producer function.
2. Create an HLS implementation for the strm_consumer function.

Create the Hardware Platform

3. Start PlanAhead and create a basic Zynq embedded system. Follow steps 1.1 to 1.18 in Exercise 1.
4. Add the strm_consumer and strm_producer blocks to the hardware design. Steps 2.19 to 2.40 in Exercise 2.
5. After adding both the adders and loopback cores to the hardware system, the Bus Interfaces tab should look like Figure 4.6

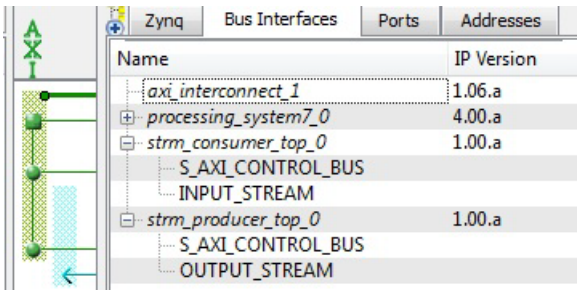


Figure 4-107: Bus Interface Tab

6. Connect OUTPUT_STREAM from the strm_producer to INPUT_STREAM of the strm_consumer

7. The Bus Interfaces tab should look like Figure 4.7

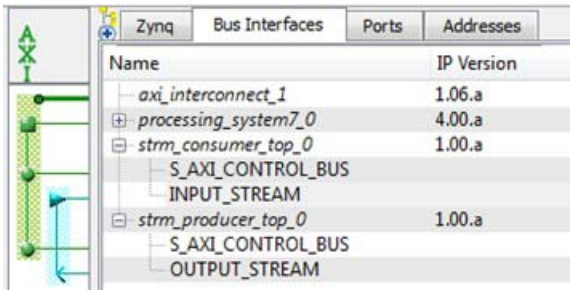


Figure 4-108: Bus Interface Tab with Stream Connections

8. In the Ports Tab, connect the clock, reset and interrupt signals for the strm_producer and strm_consumer cores. The steps for this task are in Exercise 2. Steps number 2.32 to 2.36.
9. Connect the interrupts of the strm_producer and strm_consumer cores. The steps for this task are in Exercise 2. Steps 2.37 to 2.39.
10. The Ports tab should look like Figure 4.8
11. Click the address tab and check for unmapped addresses. If unmapped addresses are listed, click the address generation button (Figure 4.9)

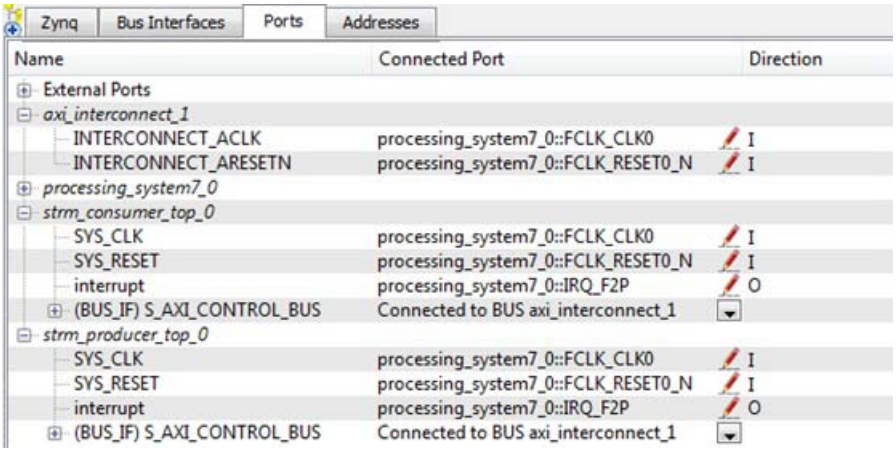


Figure 4-109: Complete Ports Tab

Figure 4-110: Address Generation Button

12. Exit XPS
13. Create top level RTL module in PlanAhead

14. Generate bitstream
15. Export hardware platform to SDK

Configuring the Software Platform

16. Create a basic hello_world application for the system. The steps for this task are in Exercise 2 steps 1.28 to 1.30
17. Program the FPGA. Steps 2.42 to 2.43 from Exercise 2.
18. Execute the default hello_world application on the board.

For this example with 2 HLS generated IP blocks, there are significant changes in the software application, which prevent a direct copy from a previous exercise like in the case of Exercise 3. The steps to change the software platform are

19. Import the HLS driver files for the strm_producer core. For reference, view steps 2.44 to 2.49 in Exercise 2.
20. Import the HLS driver files for the strm_consumer core.
21. Add the include files in Figure 4.10 to helloworld.c

```
#include "xparameters.h"
#include "xscugic.h"
#include "xstrm_producer.h"
#include "xstrm_consumer.h"
```

Figure 4-111: **List of Include Files**

22. Add the global declarations in Figure 4.11 to the program.
23. From the helloworld.c file in Exercise 2, copy the SetupInterrupt function.
24. Create support functions for the strm_producer core:

```
ProducerSetup() (Figure 4.10), ProducerStart() (Figure 4.11), ProducerIsr() (Figure
4.12)
```

25. Create support functions for the strm_consumer core:

```
ConsumerSetup() (Figure 4.13), ConsumerStart() (Figure 4.14), ConsumerIsr() (Figure
4.15).
```

26. Setup the interrupts. Example code is shown in Figure 4.16
27. Create a main function for the ARM. Example code is shown in Figure 4.17

//Declarations for the hardware cores

```
extern XStrm_producer producer;
extern XStrm_consumer consumer;
volatile static int RunProducer = 0;
volatile static int RunConsumer = 0;
```

```
volatile static int ConsumerResult = 0;

XStrm_producer producer;
XStrm_consumer consumer;

XStrm_producer_Config producer_config={
  0,
  XPAR_STRM_PRODUCER_TOP_0_S_AXI_CONTROL_BUS_BASEADDR
};
XStrm_consumer_Config consumer_config={
  1,
  XPAR_STRM_CONSUMER_TOP_0_S_AXI_CONTROL_BUS_BASEADDR
};

//Interrupt Controller Instance
XScuGic ScuGic;
```

Figure 4-112: **Global Declarations for the HLS IP Cores**

```
int ProducerSetup(){
  return XStrm_producer_Initialize(&producer,&producer_config);
}
```

Figure 4-113: **Producer Setup Function**

```
void ProducerStart(void *InstancePtr){
  XStrm_producer *pProducer = (XStrm_producer *)InstancePtr;
  XStrm_producer_InterruptEnable(pProducer,1);
  XStrm_producer_InterruptGlobalEnable(pProducer);
  XStrm_producer_Start(pProducer);
}
```

Figure 4-114: **Producer Start Function**

```
void ProducerIsr(void *InstancePtr){
  XStrm_producer *pProducer = (XStrm_producer *)InstancePtr;


  //Disable the global interrupt from the producer
  XStrm_producer_InterruptGlobalDisable(pProducer);

  XStrm_producer_InterruptDisable(pProducer,0xffffffff);

  // clear the local interrupt
  XStrm_producer_InterruptClear(pProducer,1);

  ProducerDone = 1;
  // restart the core if it should run again
  if(RunProducer){
    ProducerStart(pProducer);
  }
}
```

Figure 4-115: **Producer ISR Function**

```
int ConsumerSetup(){
  return XStrm_consumer_Initialize(&consumer,&consumer_config);
}
```

Figure 4-116: **Consumer Setup Function**

```
void ConsumerStart(void *InstancePtr){
  XStrm_consumer *pConsumer = (XStrm_consumer *)InstancePtr;
  XStrm_consumer_InterruptEnable(pConsumer,1);
  XStrm_consumer_InterruptGlobalEnable(pConsumer);
  XStrm_consumer_Start(pConsumer);
}
```

Figure 4-117: **Consumer Start Function**

```
void ConsumerIsr(void *InstancePtr){
  XStrm_consumer *pConsumer = (XStrm_consumer *)InstancePtr;

  //Disable the global interrupt from the producer
  XStrm_consumer_InterruptGlobalDisable(pConsumer);
  //Disable the local interrupt
  XStrm_consumer_InterruptDisable(pConsumer,0xffffffff);

  // clear the local interrupt
  XStrm_consumer_InterruptClear(pConsumer,1);

  ConsumerResult = 1;
  // restart the core if it should run again
  if(RunConsumer){
    ConsumerStart(pConsumer);
  }
}
```

Figure 4-118: **Consumer ISR Function**

```
int SetupInterrupt()
{
  //This functions sets up the interrupt on the ARM
  int result;
  XScuGic_Config *pCfg =
                        XScuGic_LookupConfig(XPAR_SCUGIC_SINGLE_DEVICE_ID);
  if (pCfg == NULL){
    print("Interrupt Configuration Lookup Failed\n\r");
    return XST_FAILURE;
  }
  result = XScuGic_CfgInitialize(&ScuGic,pCfg,pCfg->CpuBaseAddress);
  if(result != XST_SUCCESS){
    return result;
  }
  // self test
  result = XScuGic_SelfTest(&ScuGic);
  if(result != XST_SUCCESS){
    return result;
  }
  // Initialize the exception handler
  Xil_ExceptionInit();
  // Register the exception handler
```

```
  Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,(Xil_ExceptionHandler)
                                    XScuGic_InterruptHandler,&ScuGic);
  //Enable the exception handler
  Xil_ExceptionEnable();
  // Connect the Producer ISR to the exception table
  result =
       XScuGic_Connect(&ScuGic,XPAR_FABRIC_STRM_PRODUCER_TOP_0_INTERRUPT_INTR,
                       (Xil_InterruptHandler)ProducerIsr,&producer);
  if(result != XST_SUCCESS){
    return result;
  }
  //Connect the Consumer ISR to the exception table
  result =
       XScuGic_Connect(&ScuGic,XPAR_FABRIC_STRM_CONSUMER_TOP_0_INTERRUPT_INTR,
                        (Xil_InterruptHandler)ConsumerIsr,&consumer);
  if(result != XST_SUCCESS){
      return result;
  }
       //Enable the interrupts for the Producer and Consumer
XScuGic_Enable(&ScuGic,XPAR_FABRIC_STRM_PRODUCER_TOP_0_INTERRUPT_INTR);
  XScuGic_Enable(&ScuGic,XPAR_FABRIC_STRM_CONSUMER_TOP_0_INTERRUPT_INTR);
  return XST_SUCCESS;
}
```

Figure 4-119: **Interrupt Setup Function**

```
int main()
{
    Init_platform();

    print("AutoESL Consumer Producer Example\n\r");
    int length;
    int status;
    int result;
    length = 50;
    printf("Length of stream = %d\n\r",length);

    status = ProducerSetup();
    if(status != XST_SUCCESS){
       print("Producer setup failed\n\r");
    }
    status = ConsumerSetup();
    if(status != XST_SUCCESS){
        print("Consumer setup failed\n\r");
    }
    //Setup the interrupt
       status = SetupInterrupt();
       if(status != XST_SUCCESS){
           print("Interrupt setup failed\n\r");
       }

    XStrm_consumer_SetStrm_len(&consumer,length);
    XStrm_producer_SetStrm_len(&producer,length);

    ProducerStart(&producer);
    ConsumerStart(&consumer);

    while(!ProducerDone) print("waiting for producer to finish\n\r");
```

```
    while(!ConsumerResult) print("waiting for consumer to finish\n\r");

    result = XStrm_consumer_GetDout(&consumer);
    printf("Consumer result = %d\n\r",result);
    print("Finished\n\r");

    cleanup_platform();

    return 0;
}
```

Figure 4-120: **Example Main Application**

Exercise 5: DMA Transfer to High-Level Synthesis IP

This exercise shows how an HLS generated IP block can communicate with external memory by using DMA transfers. The HLS block connects to the DMA controller using AXI4-streams. This approach avoids addressing logic and the pipelining limitations associated with the current implementation of AXI4 in HLS. Burst formatting, address generation and scheduling of the memory transaction is left to the AXI DMA IP.

This exercise uses the ACP port to connect the AXI DMA to the L2 cache of the ARM processor. The techniques shown in this experiment are directly applicable to connections using the HP ports for connection to DDR. From the view of the HLS core, the memory interface through the DMA is the same regardless of whether the memory is DDR or L2 cache. It is a system architecture decision to share data between the processor and the HLS core by either using DDR or the L2 cache. L2 cache will offer a lower latency in communication than the DDR. In contrast, the DDR provides the ability to transfer a lot more data than the L2 cache.

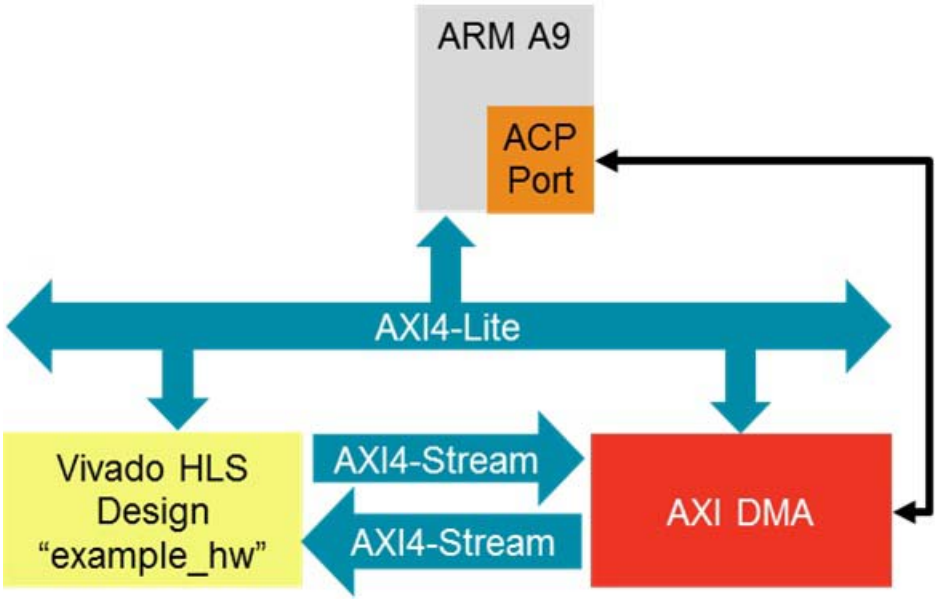


Figure 4-121: Exercise 5 System

High-Level Synthesis Software Projects

For this exercise, a single HLS block will be connected to the AXI DMA. The code for the HLS project is shown in Figure 5.2 to Figure 5.4. Create this software project in HLS and generate a PCore for Zynq. The instructions on how to run HLS are described in Exercise 2 steps 2.1 to 2.18.

The code in Figure 5.2 shows the usage of the AXI_VAL datatype. This is a user defined datatype, which expresses the side channel information associated with AXI4-stream. In

HLS, any side channel information that is not part of the protocol handshake must be expressed in the C/C++ code and used in some way. This means that while HLS will abstract the TREADY and TVALID signals all other signals in the AXI4-stream must be part of the user code. Also, keep in mind that besides the TDATA, TREADY and TVALID signals, all other AXI4-stream signals are optional. The use of side channel signals depends on the blocks the HLS AXI4-stream interface is connecting to.

```
#include <stdio.h>
#include <stdlib.h>
#include "ap_interfaces.h"
#include "ap_axi_sdata.h"
typedef ap_mm2s<32,1,1,1> AXI_VAL;
void example_hw(AXI_VAL A[256], AXI_VAL B[256], int val1, int val2);
```

Figure 4-122: **example.h Header File**

```
#include "example.h"

//Software implementation of the function in hardware
void example_sw(AXI_VAL A[256], AXI_VAL B[256], int val1, int val2){
  int i;
  for(i=0; i < 256; i++){
    B[i].data = A[i].data.to_int() + val1 - val2;
}
}
int main(){
  int i;
  int val1, val2;
  AXI_VAL A[256];
  AXI_VAL res_hw[256], res_sw[256];

  //Set the values of the input stream A
  for(i=0; i < 256; i++){
     A[i].data = i;
}

//Set the additional parameters of the function
  val1 = 10;
  val2 = 5;

  //Call the software implementation
  example_sw(A,res_sw,val1,val2);
  //Call the hardware implementation
  example_hw(A,res_hw,val1,val2);

  //Compare the software and hardware results for correctness
  for(i=0; i < 256; i++){
    if(res_sw[i].data.to_int() != res_hw[i].data.to_int()) {
      printf("Error: software and hardware result mismatch\n");
      //A return value of 1 marks an error
      return 1;
    }
  }

  Printf("Success! software and hardware match\n");
```

```
  //A return value of 0 marks success
  return 0;
}
```

Figure 4-123: **Exercise 5 Testbench**

```
#include "example.h"

void example_hw(AXI_VAL A[256], B[256], int val1, int val2){

//Define AutoESL native interface behavior for val1 and val2
AP_INTERFACE(val1,none);
AP_INTERFACE(val2,none);

//Map val1 and val2 from a native AutoESL interface to AXI4-lite
AP_BUS_AXI4_LITE(val1,CONTROL_BUS);
AP_BUS_AXI4_LITE(val2,CONTROL_BUS);

//Map the control of the function to AXI4-lite
AP_CONTROL_BUS_AXI(CONTROL_BUS);

//Create an AXI4-stream interface for arrays A and B
AP_BUS_AXI_STREAMD(A,INPUT_STREAM);
AP_BUS_AXI_STREAMD(B,OUTPUT_STREAM);

  int i;
  AXI_VAL input_value, result;

  for(i = 0; i < 256; i++){
    input_value = A[i];
    result.data = input_value.data.to_int() + val1 - val2;
    result.keep = 15;
    if(I == 255)
      result.last = 1;
    else
      result.last = 0;
    B[i] = result;
  }
}
```

Figure 4-124: **Exercise5 Application Code**

1. Create an HLS implementation of the example_hw function in Figure 4-124.

Create the Hardware Platform

2. Start PlanAhead and create a basic Zynq system in the exercise5 folder. Follow steps 1.1 to 1.18 in Exercise 1.
3. Copy the pcore created for example_hw into the pcore directory of the Zynq system. For reference, please look at step 2.20 in Exercise 2
4. Click on the green box for 64b AXI ACP port in the Zynq tab (Figure 4-125)

5. Click Enable S_AXI_ACP interface in the XPS Core Config window and click ok. (Figure 4-126)

Figure 4-125: ACP Port

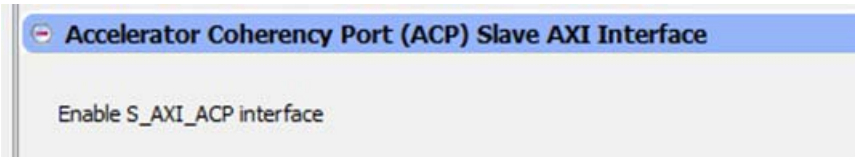


Figure 4-126: Enable ACP Interface

6. Double click example_hw core to add it to the design. This core should be avai lable in the USER section of the IP Catalog.
7. Click yes when asked to confirm adding the example_hw core to the design
8. In the Core Config screen, change the following settings:

   C_INPUT_STREAM PROTOCOL =

   GENERIC to **XIL_AXI_STREAM_ETH_DATA**

   C_OUTPUT_STREAM_PROTOCOL =

   GENERIC to **XIL_AXI_STREAM_ETH_DATA**

9. Click ok
10. Select the processor instance to make the connections, click ok

11. In the IP Catalog, double click on AXI Interconnect to add it to the system (Figure 4-127)

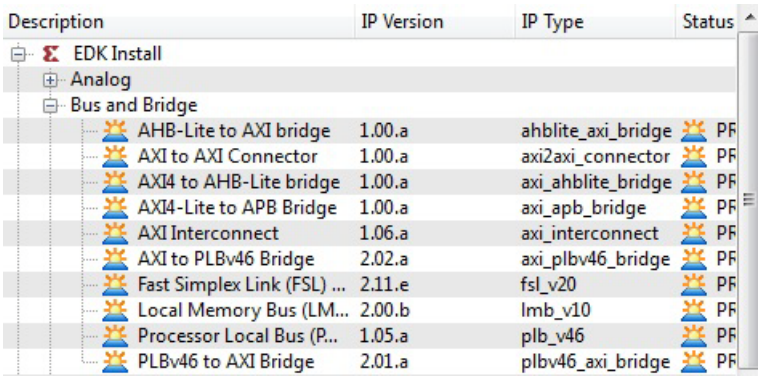


Figure 4-127: AXI Interconnect Selection

12. Click yes to confirm adding the axi_interconnect to the design.
13. Click ok in the Core Config screen (accept all defaults)
14. In the IP Catalog, double click on the AXI DMA Engine v5.00.a to add it to the system (Figure 4-128)

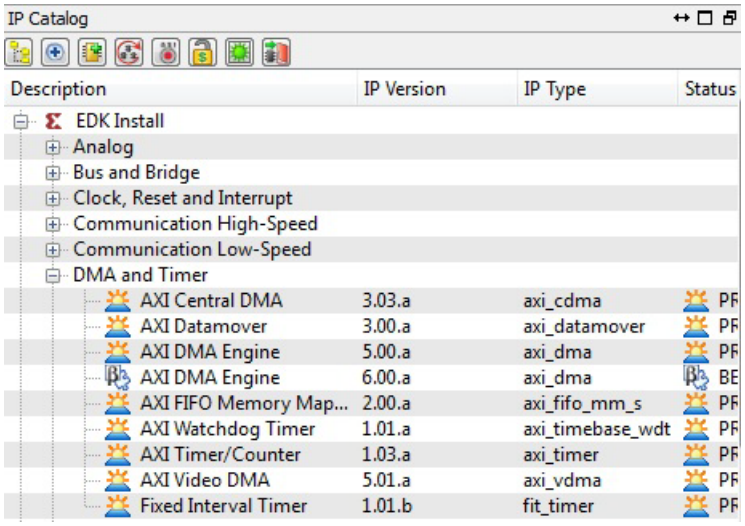


Figure 4-128: AXI DMA Selection

15. Click yes to confirm adding the axi_dma to the design.
16. In the Core Config screen:
    a. click to enable Include MM2S Data Realignment Engine
    b. click to enable Include S2MM Data Realignment Engine
    c. click to disable Include Scatter Gather Engine
17. Set the Maximum Memory Map Burst Size for MM2s to 256

18. Set the Maximum Memory Map Burst Size for S2MM to 256
19. The Core Config screen should look like Figure 4-129
20. Click ok

Figure 4-129: AXI DMA Core Configuration

21. Click yes to confirm adding the axi_dma to the design.
22. Select "user will make necessary connections" and click ok
23. The Bus Interfaces tab should look like Figure 4-130

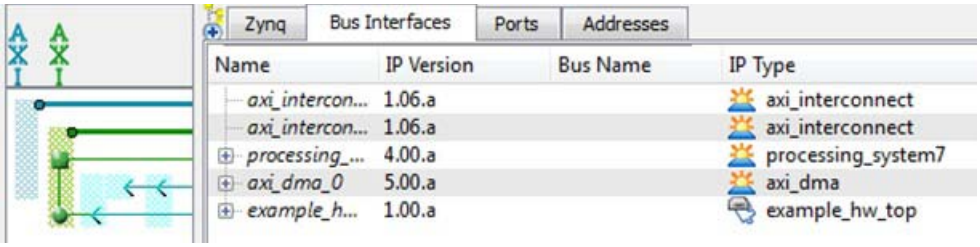


Figure 4-130: Bus Interfaces Tab

24. Click the + sign and expand all cores in the bus interface tab
25. Connect the S_AXI_LITE from axi_dma_0 to M_AXI_GP0
26. Connect M_AXI_MM2S to axi_interconnect_0 (dma engine)
27. Connect M_AXI_S2MM to axi_interconnect_0 (dma engine)
28. Connect S_AXI_ACP to axi_interconnect_0 (processor)
29. Connect S_AXIS_S2MM to OUTPUT_STREAM

30. Connect M_AXIS_MM2S to INPUT_STREAM
31. The Bus Interfaces tab should look like Figure 4-131

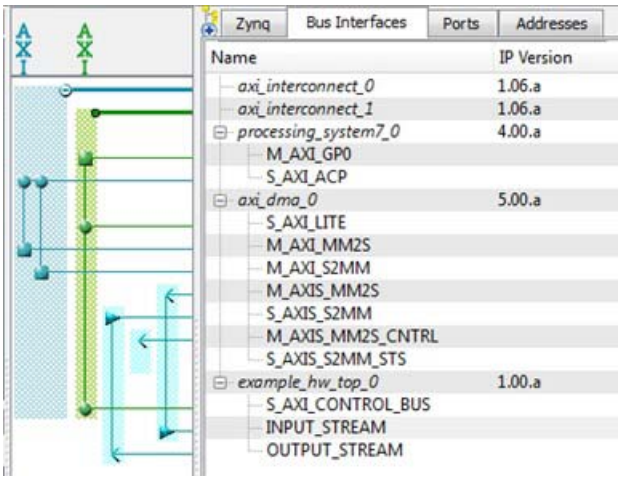


Figure 4-131: Bus Interfaces Tab after Configuration

32. Click Addresses
33. If anything appears as an unmapped address, click the address generation button (Figure 4-132)
34. Address tab should look like Figure 4-133

Figure 4-132: Address Generator Button

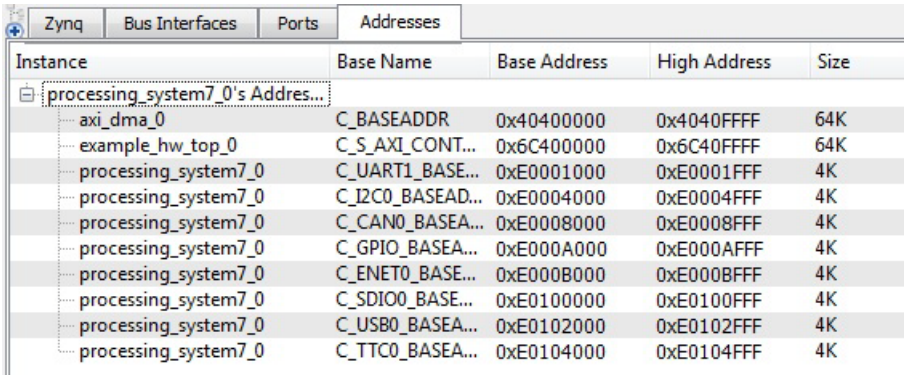


Figure 4-133: Address Generation Tab

The following steps detail which ports must be connected in the ports tab to complete the hardware system design. For reference, the instructions on how to create port connections are in steps 2.32 to 2.39 of Exercise 2.

35. Connect ports from axi_interconnect_0

```
INTERCONNECT_ACLK to processing_system7::FCLK_CLK0
INTERCONNECT_ARESETN to processing_system7::FCLK_RESET0_N
```

36. Connect port from processing_system7

```
S_AXI_ACP_ACLK to processing_system7::FCLK_CLK0
```

37. Connect ports from axi_dma_0 to processing_system7::FCLK_CLK0

```
s_axi_lite_aclk, m_axi_mm2s_aclk, m_axi_s2mm_aclk
```

38. Connect ports from example_hw_top_0

```
SYS_CLK to processing_system7::FCLK_CLK0
SYS_REST to processing_system7::FCLK_RESET0_N
```

39. Connect the interrupt from example_hw_top_0
40. Close and exit XPS
41. Create top level RTL module in PlanAhead
42. Generate a bitstream
43. Export the hardware platform to XPS and launch XPS

Software Application for ARM

The ARM software application for Exercise 5 is an extension of the software applications created for the previous experiments. It has the same structure of the previous applications and the same infrastructure functions for the HLS generated IP. In addition to the existing software, this application includes the routines to handle the DMA engine.

44. Create the basic hello_world application
45. Program the FPGA
46. Run hello_world on the board to test the system is working correctly
47. Import the software driver files for the example_hw HLS IP block
48. Modify the include section of hello_world to look like Figure 4-134

```
#include <stdio.h>
#include "platform.h"
#include "xparameters.h"
#include "xscugic.h"
#include "xaxidma.h"
#include "xexample_hw.h"
```

Figure 4-134: **Include Files for ARM Application**

49. Create the support functions for the HLS IP

```
ExampleHwSetup(), ExampleHwStart(), ExampleHwIsr(),
```

50. Create the function to setup the interrupt: SetupInterrupt()

The functions in steps 5.49 and 5.50 were developed in Exercise 2. Different versions of these functions are available in exercises 2 to 4. Before the DMA engine can be utilized, it must also be initialized and configured properly. An example on how to initialize the DMA is shown in Figure 4-135

```
int init_dma(){
  XAxiDma_Config *CfgPtr;
  int status;

  CfgPtr = XAxiDma_LookupConfig(XPAR_AXI_DMA_0_DEVICE_ID);
  if(!CfgPtr){
    print("Error looking for AXI DMA config\n\r");
    return XST_FAILURE;
  }
  status = XAxiDma_CfgInitialize(&AxiDma,CfgPtr);
  if(status != XST_SUCCESS){
    print("Error initializing DMA\n\r");
    return XST_FAILURE;
  }
  //check for scatter gather mode
  if(XAxiDma_HasSg(&AxiDma)){
    print("Error DMA configured in SG mode\n\r");
    return XST_FAILURE;
  }
  //disable the interrupts from the DMA
  XAxiDma_IntrDisable(&AxiDma,XAXIDMA_IRQ_ALL_MASK,XAXIDMA_DEVICE_TO_DMA);
  XAxiDma_IntrDisable(&AxiDma,XAXIDMA_IRQ_ALL_MASK,XAXIDMA_DMA_TO_DEVICE);

  return XST_SUCCESS;
}
```

Figure 4-135: **Example Code to Initialize the DMA**

The DMA engine is capable of generating interrupts at the end of each transaction. For this Exercise, the interrupts are disabled. The processor will know that the computation on the fabric is complete by waiting on the HLS IP interrupt and polling the status of the DMA engine.

The main function in this Exercise uses the self-checking concept used in HLS testbenches to compare the results generated by the processor and by the FPGA fabric for the same function. The processor implementation of the FPGA fabric functionality is shown in Figure 4-136.

```
void example_sw(int A[256], int B[256], int val1, int val2){
  int i;
  for(i = 0; i < 256; i++){
    B[i] = A[i] + val1 - val2;
  }
}
```

Figure 4-136: **Processor Implementation of FPGA Computation**

The main function for the processor is shown in Figure 4-135.

51. Create the DMA initialization function - example in Figure 4-135
52. Create the software version of the hardware function Figure 4-136
53. Modify the main function to use the DMA Figure 4-137
54. Run the application on the board

```
int main(){
  int i;
  int status;
  int A[256];
  int val1, val2;
  int res_hw[256], int res_sw[256];

  val1 = 10;
  val2 = 5;

  init_platform();
print("AutoESL and DMA example\n");
for(i=0; i < 256; i++) A[i] = i;

//Setup the AutoESL Block
status = ExampleHwSetup();
if(status != XST_SUCCESS){
          print("Error: example setup failed\n");
      return XST_FAILURE;
  }
  status = SetupInterrup();
if(status != XST_SUCCESS){
          print("Error: interrupt setup failed\n");
      return XST_FAILURE;
  }
  XExample_hw_SetVal1(&example,val1);
  XExample_hw_SetVal2(&example,val2);
  ExampleHwStart();

  status = init_dma();
if(status != XST_SUCCESS){
          print("Error: DMA setup failed\n");
      return XST_FAILURE;
  }

  //flush the cache
  Unsigned int dma_size = 256 * sizeof(unsigned int);
  Xil_DCacheFlushRange((unsigned int)A,dma_size);
  Xil_DCacheFlushRange((unsigned int)res_hw,dma_size);

  //transfer A to the AutoESL block
  status = XAxiDma_SimpleTransfer(&AxiDma,(unsigned int)A, dma_size,
               XAXIDMA_DMA_TO_DEVICE);
if(status != XST_SUCCESS){
          print("Error: DMA transfer to AutoESL block failed\n");
```

```
      return XST_FAILURE;
  }

  //get results from the AutoESL block
  status = XAxiDma_SimpleTransfer(&AxiDma,(unsigned int)res_hw,dma_size,
                XAXIDMA_DEVICE_TO_DMA);



if(status != XST_SUCCESS){
          print("Error: DMA transfer from AutoESL block failed\n");
      return XST_FAILURE;
  }

  //poll the DMA engine to verify transfers are complete
  while((XAxiDma_Busy(&AxiDma,XAXIDMA_DEVICE_TO_DMA)) ||
       (XAxiDma_Busy(&AxiDma,XAXIDMA_DMA_TO_DEVICE)))

  //wait on the result from the AutoESL block
     while(!ResultExample) print("Waiting for core to finish\n\r");

  //call the software version of the function
example_sw(A, res_sw, val1, val2);

  //Compare the results from sw and hw
  for(I = 0; I < 256; i++){
    if(res_sw[i] != res_hw[i]){
      print("ERROR: results mismatch\n");
      return 1;
    }
  }
  print("SW and HW results match!\n");
       cleanup_platform();

       return 0;
}
```

Figure 4-137: **Main Function for ARM**

XILINX

Appendix A

Additional Resources

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see the Xilinx Support website at:

www.xilinx.com/support

For a glossary of technical terms used in Xilinx documentation, see:

www.xilinx.com/company/terms.htm

Solution Centers

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

References

- Vivado Design Suite 2012.4 Documentation
  http://www.xilinx.com/support/documentation/dt_vivado_vivado2012-4.htm