Vivado Design Suite Tutorial

High-Level Synthesis

UG871 (v2012.2) August 20, 2012

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
8/20/12	1.0	Initial Xilinx release of the Vivado Design Suite Tutorial: High-Level Synthesis.

Table of Contents

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.

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 execute 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 > Vivado <version> > Vivado HLS GUI**. The Vivado HLS group is shown in Figure 3-1.

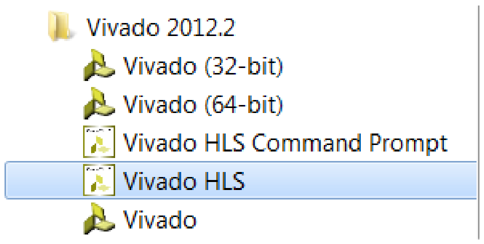
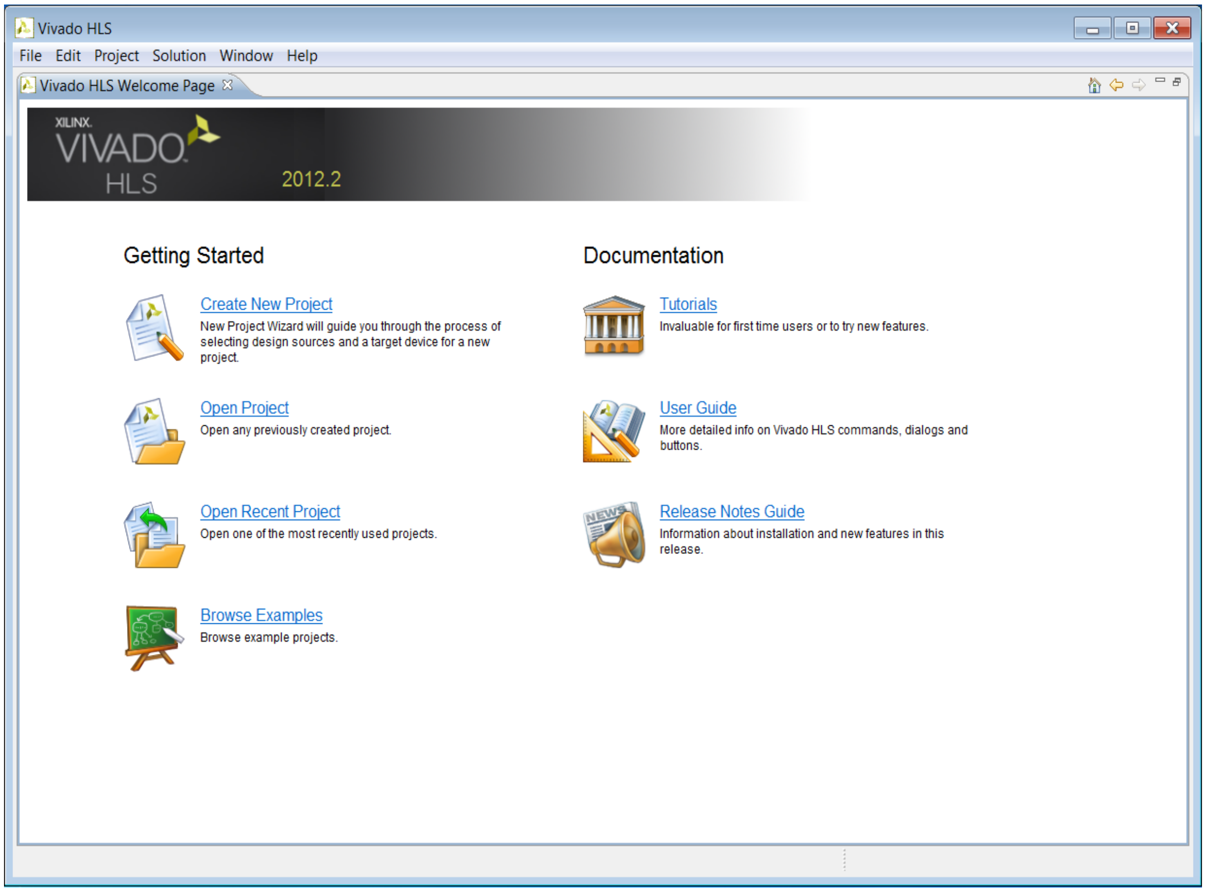


Figure 1-1: **Launching the Vivado HLS GUI**

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



Getting Started

Create New Project
New Project Wizard will guide you through the process of selecting design sources and a target device for a new project.

Open Project
Open any previously created project.

Open Recent Project
Open one of the most recently used projects.

Browse Examples
Browse example projects.

Documentation

Tutorials
Invaluable for first time users or to try new features.

User Guide
More detailed info on Vivado HLS commands, dialogs and buttons.

Release Notes Guide
Information about installation and new features in this release.

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

The Getting Started options are:

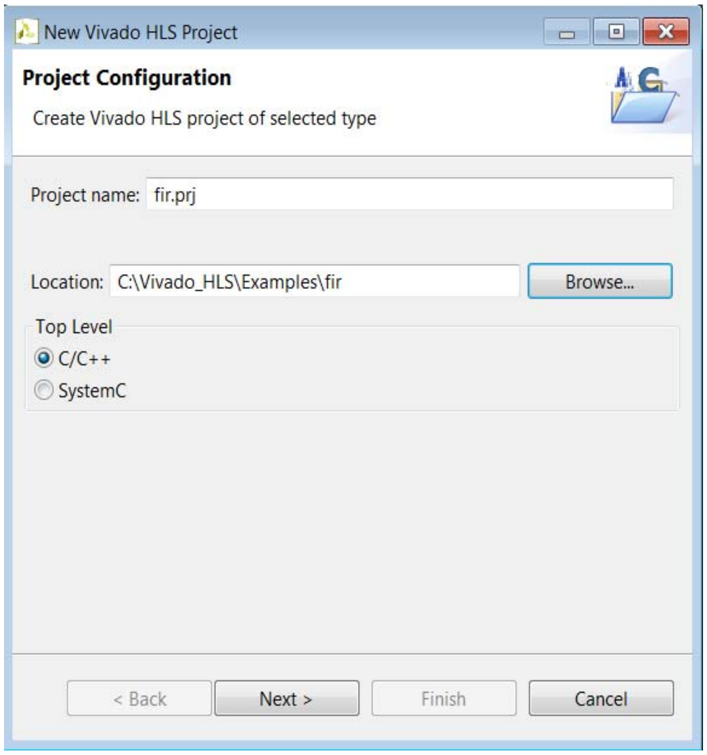
- **Create New Project**: Launches the project setup wizard.
- **Open Project**: Opens a window for you to navigate to an existing project.
- **Open Recent Project**: Gives you a list of recent projects, from which you can select one to open.
- **Browse Examples**: Open Vivado HLS examples. These can also be found in the examples directory in the Vivado HLS installation area.

The Documentation options are:

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

Creating a New Project

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



Project name: fir.prj

Location: C:\Vivado_HLS\Examples\fir

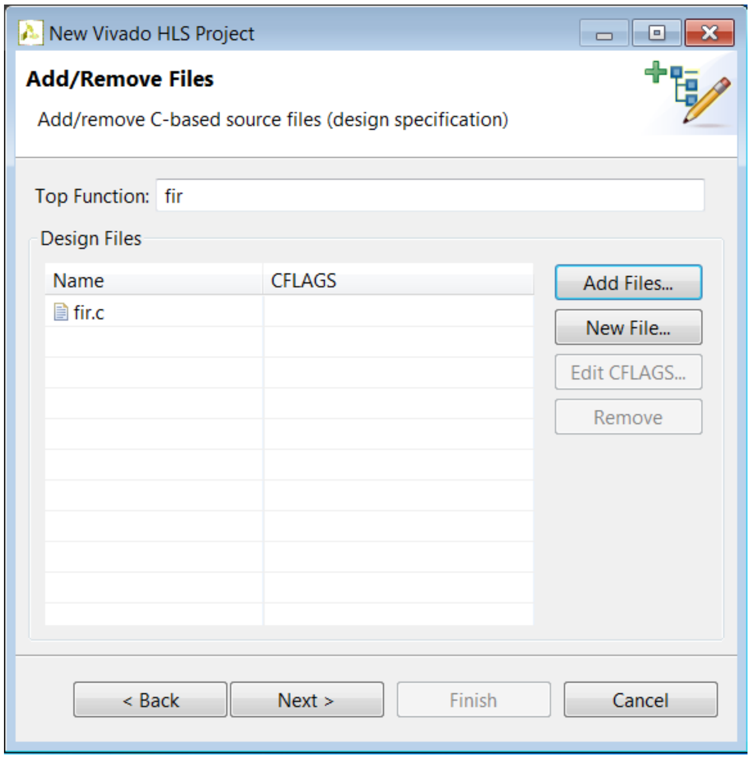
Top Level
(● C/C++) ○ SystemC

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 3-4).



New Vivado HLS Project

Add/Remove Files

Add/remove C-based source files (design specification)

Top Function: fir

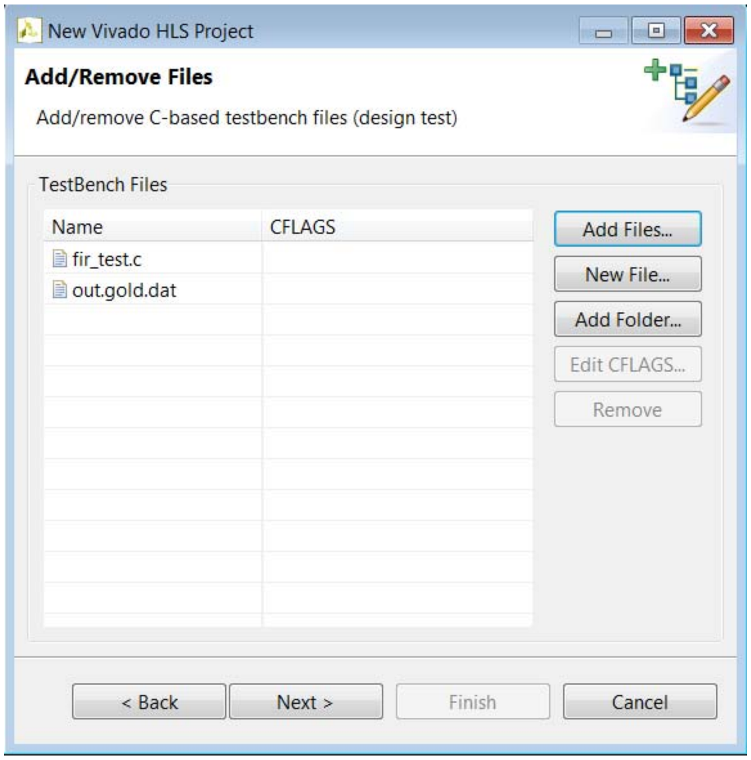
Design Files

Name	CFLAGS
fir.c	

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 3-5 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.



Add/Remove Files

Add/remove C-based testbench files (design test)

TestBench Files

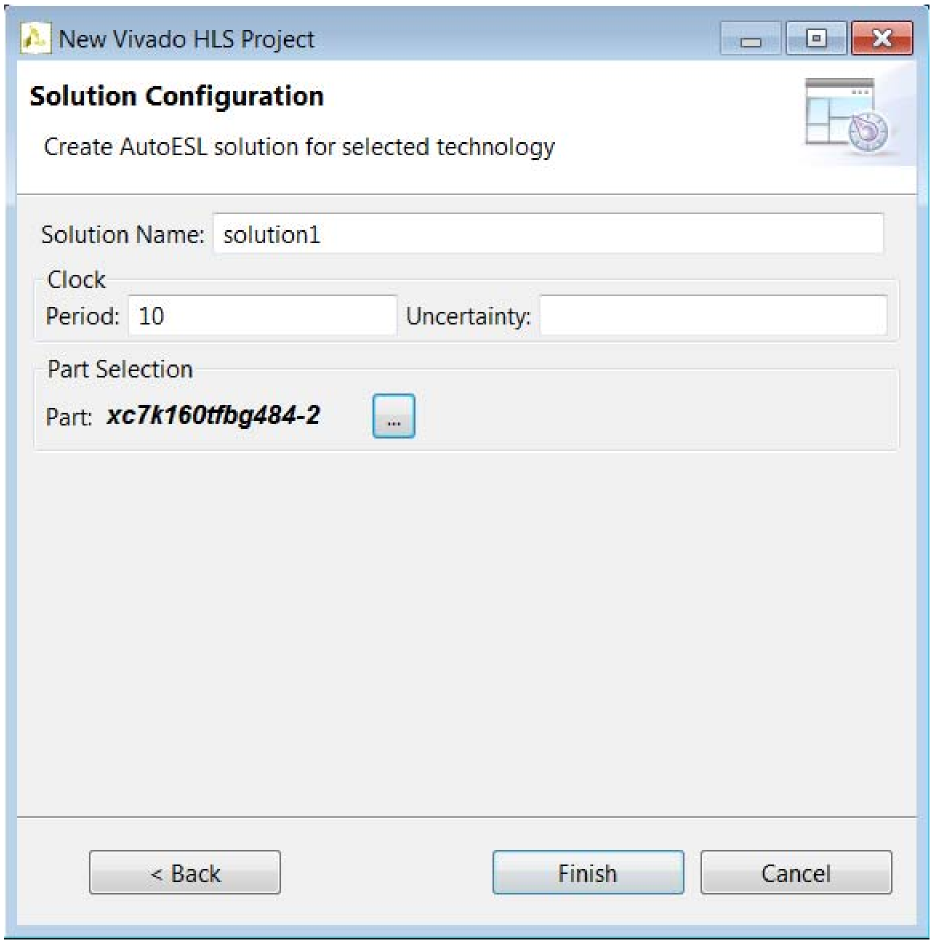
Name	CFLAGS
fir_test.c	
out.gold.dat	

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 3-6) 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.


New Vivado HLS Project

Solution Configuration

Create AutoESL solution for selected technology

Solution Name: solution1

Clock

Period: 10 Uncertainty:

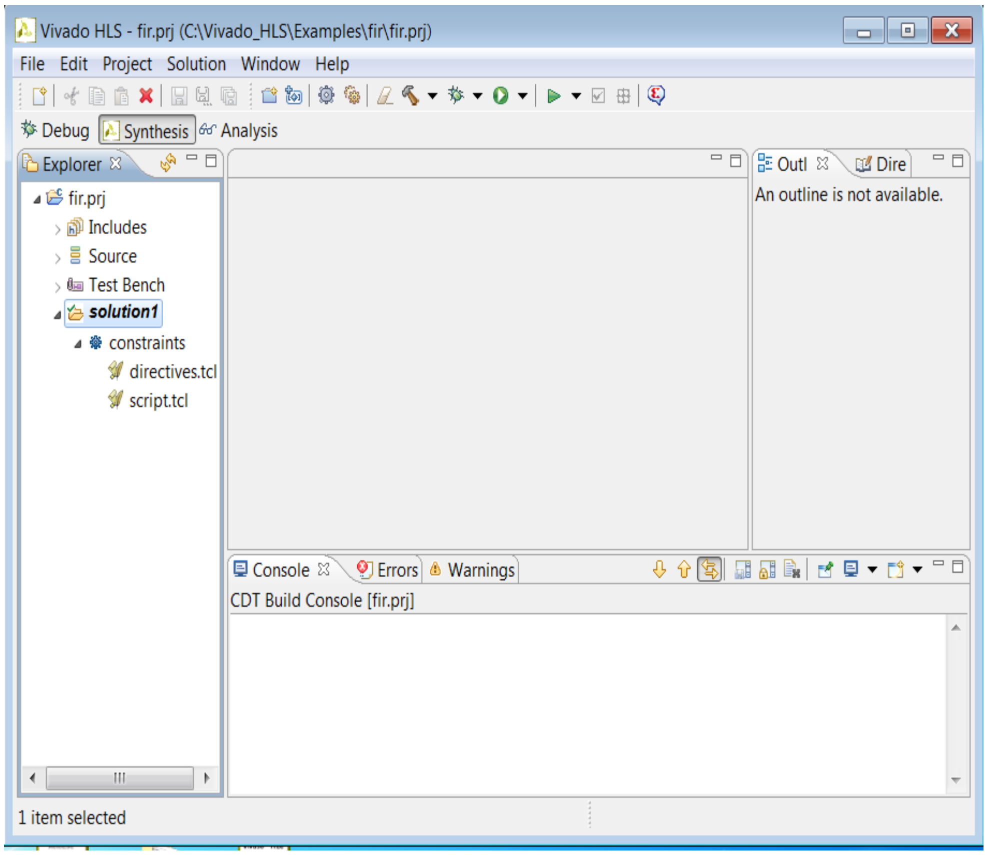
Part Selection

Part: **xc7k160tfbg484-2**

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 3-6.

The Vivado HLS GUI opens with the project information included, as shown in Figure 3-7.



An outline is not available.

CDT Build Console [fir.prj]

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 3-8 and Figure 3-9.

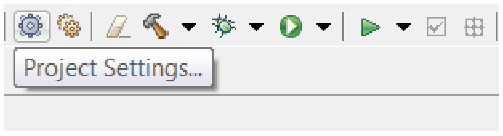


Figure 1-8: **Project Settings**

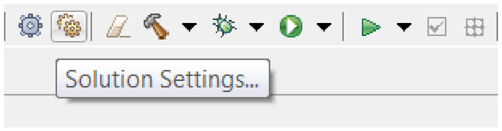


Figure 1-9: **Solution Settings**

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 two types of C compilation: `Debug` and `Release`.

- Code compiled for `Debug` can be used in the Vivado HLS debug environment.
- Code compiled for `Release` executes faster, because it has no debug information. However, it cannot be used in the debug environment.

This tutorial demonstrates both types of C compilation.

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 3-10 shows the `Build` button on the toolbar and the tool pop-up shows that the current default build type is for a debug configuration.

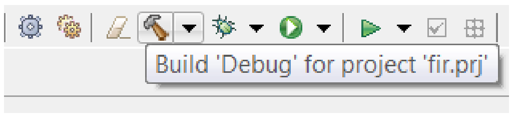
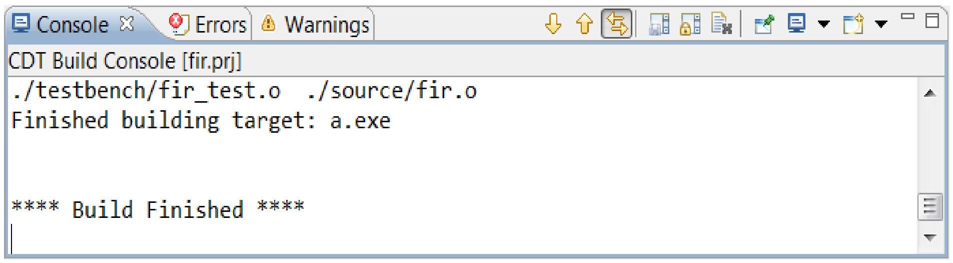


Figure 1-10: **Build (Debug) Toolbar Button**

1. Click the **Build** button, shown in Figure 3-10, to compile the design.

   The output of the build process is shown in the Console Pane at the bottom of the GUI, as shown in Figure 3-11.



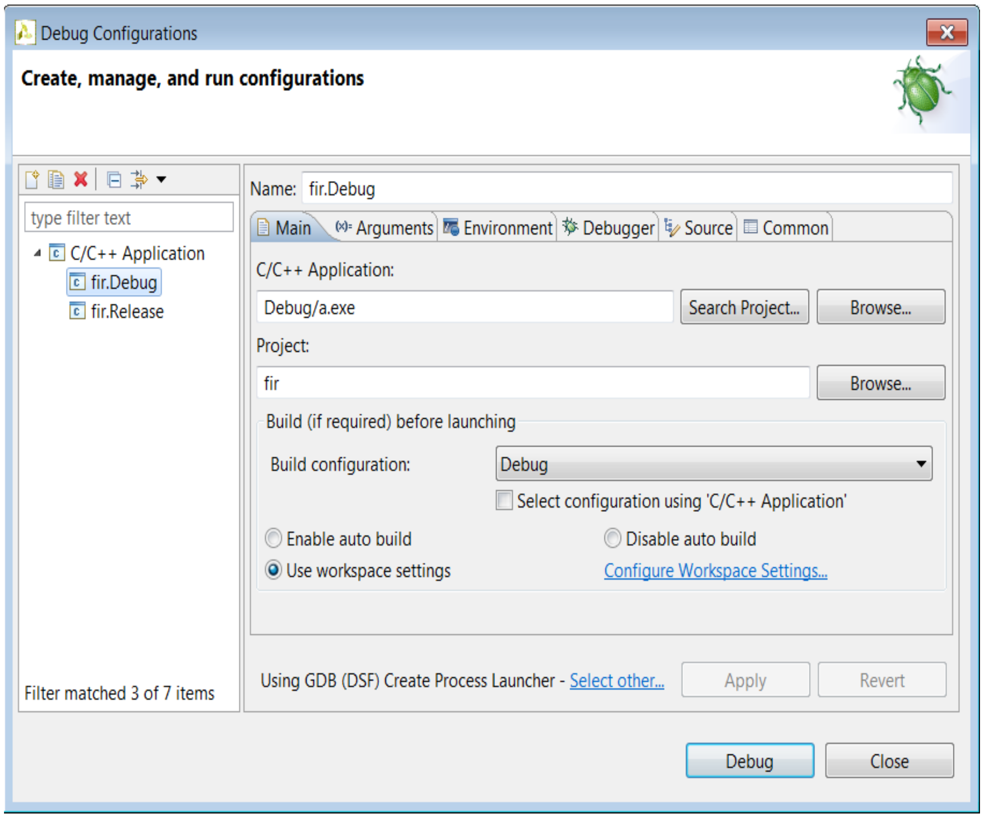
```
./testbench/fir_test.o  ./source/fir.o
Finished building target: a.exe


**** Build Finished ****
```

Figure 1-11: **Build: Console Output**

   You can now execute the build to validate the C function before synthesis.

2. Click the drop-down arrow next to the **Debug** button (shown in Figure 3-10) and select **Debug Configurations**.

   This opens the Run Configuration dialog box shown in Figure 3-12.



Name: fir.Debug

C/C++ Application: Debug/a.exe

Project: fir

Build (if required) before launching

Build configuration: Debug

Select configuration using 'C/C++ Application' [ ]

Enable auto build [ ] Disable auto build [ ]

Use workspace settings [✓]

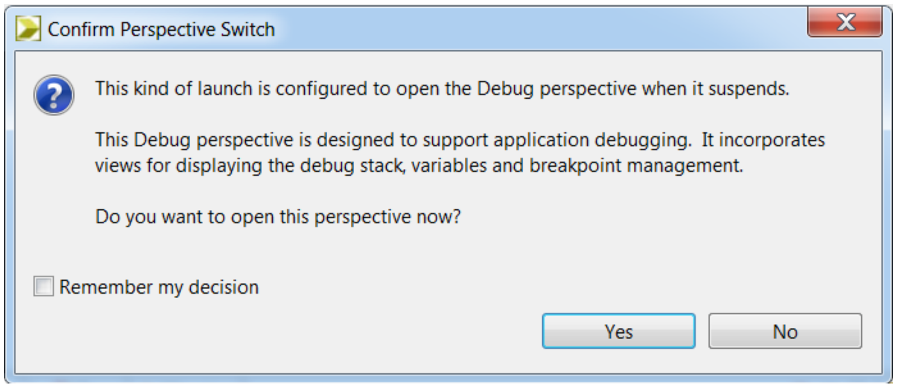
Using GDB (DSF) Create Process Launcher

Figure 1-12: **Run Configuration: Debug**

3. Expand **C/C++ Application** and select the `fir.prj.Debug` configuration (see Figure 3-12).
4. Click **Debug**.

   The build executes and you are prompted to move to the debug environment.
5. Select **Yes**.

   The debugger opens (see Figure 3-14).


Confirm Perspective Switch

This kind of launch is configured to open the Debug perspective when it suspends.

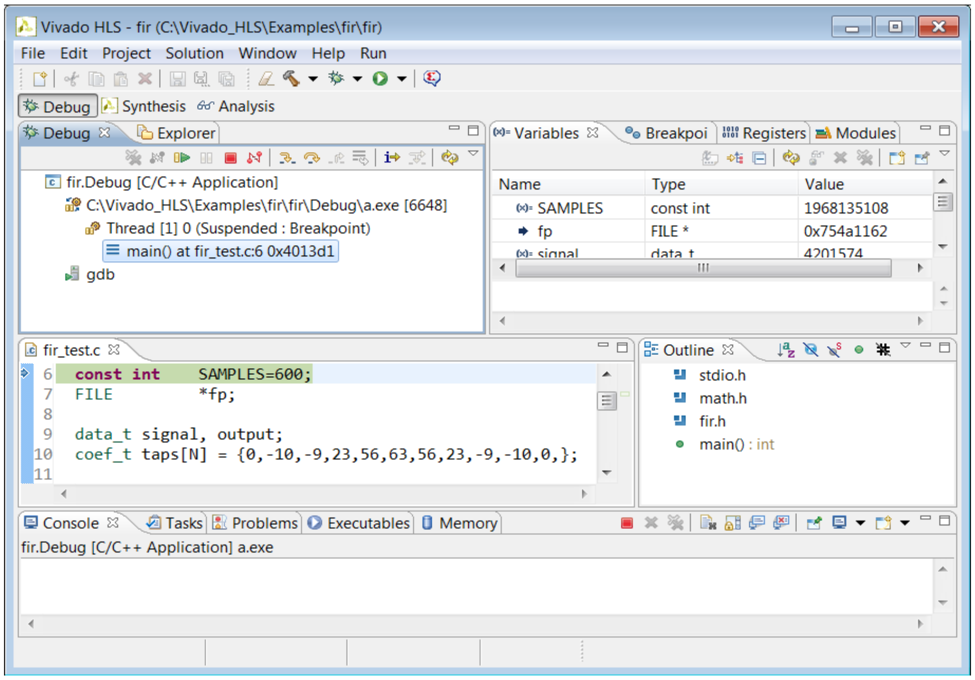
This Debug perspective is designed to support application debugging. It incorporates views for displaying the debug stack, variables and breakpoint management.

Do you want to open this perspective now?

Remember my decision

Yes No

Figure 1-13: **Debug Environment**


Vivado HLS - fir (C:\Vivado_HLS\Examples\fir\fir)

File Edit Project Solution Window Help Run

Debug Synthesis Analysis

fir.Debug [C/C++ Application]
C:\Vivado_HLS\Examples\fir\fir\Debug\a.exe [6648]
Thread [1] 0 (Suspended : Breakpoint)
main() at fir_test.c:6 0x4013d1
gdb

Name	Type	Value
SAMPLES	const int	1968135108
fp	FILE *	0x754a1162
signal	data_t	4201574

```
const int     SAMPLES=600;
FILE          *fp;

data_t signal, output;
coef_t taps[N] = {0,-10,-9,23,56,63,56,23,-9,-10,0,};

```

Outline: stdio.h, math.h, fir.h, main() : int

Console: fir.Debug [C/C++ Application] a.exe

Figure 1-14: **Debugger Window**

The following steps describe using the debugger.

6. Step through the code by clicking the **Step Into** toolbar button, as shown in Figure 3-15.

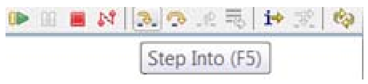
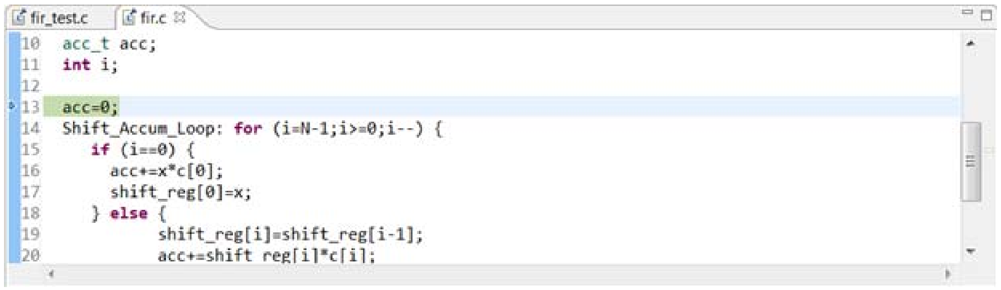


Figure 1-15: **Step Into Button**

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

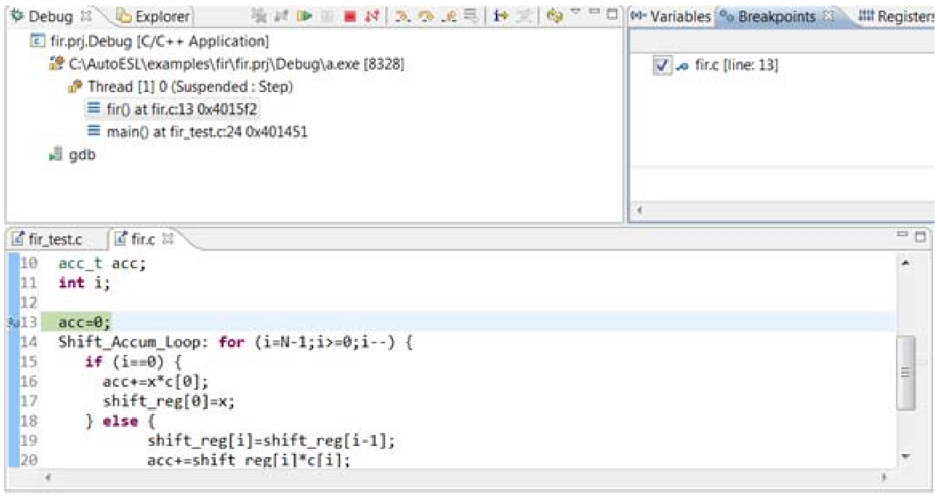
   The code window looks like Figure 3-16.



```
10  acc_t acc;
11  int i;
12
13  acc=0;
14  Shift_Accum_Loop: for (i=N-1;i>=0;i--) {
15     if (i==0) {
16       acc+=x*c[0];
17       shift_reg[0]=x;
18     } else {
19            shift_reg[i]=shift_reg[i-1];
20            acc+=shift_reg[i]*c[i];
```

Figure 1-16: **Debug in the FIR Design**

8. To add a breakpoint, in the left-hand margin of the fir.c tab, double-click on line **13**. A breakpoint indication mark appears to the left of the line number, as shown in Figure 3-17.



```
fir.prj.Debug [C/C++ Application]
  C:\AutoESL\examples\fir\fir.prj\Debug\a.exe [8328]
    Thread [1] 0 (Suspended : Step)
      fir() at fir.c:13 0x4015f2
      main() at fir_test.c:24 0x401451
  gdb
```

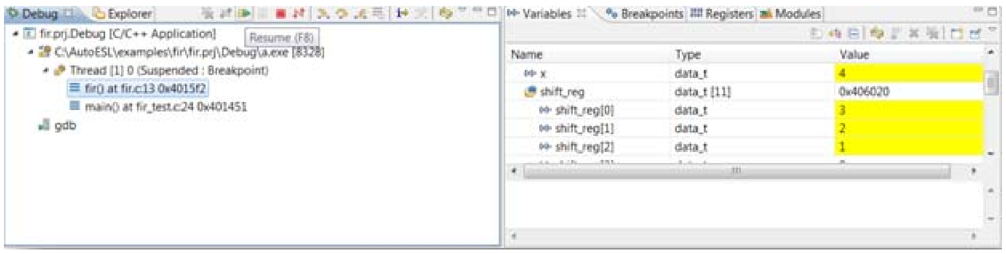
```
fir.c [line: 13]
```

```
10  acc_t acc;
11  int i;
12
13  acc=0;
14  Shift_Accum_Loop: for (i=N-1;i>=0;i--) {
15     if (i==0) {
16       acc+=x*c[0];
17       shift_reg[0]=x;
18     } else {
19            shift_reg[i]=shift_reg[i-1];
20            acc+=shift_reg[i]*c[i];
```

Figure 1-17: **Adding a Breakpoint**

9. To confirm the breakpoint has been added, open the Breakpoints tab, shown in Figure 3-17.

10. Open the Variables tab, shown in Figure 3-18.



```
fir.prj.Debug [C/C++ Application]
  C:\AutoESL\examples\fir\fir.prj\Debug\a.exe [8328]
    Thread [1] 0 (Suspended : Breakpoint)
      fir() at fir.c:13 0x4015f2
      main() at fir_test.c:24 0x401451
  gdb
```

Name	Type	Value
x	data_t	4
shift_reg	data_t [11]	0x406020
shift_reg[0]	data_t	3
shift_reg[1]	data_t	2
shift_reg[2]	data_t	1

Figure 1-18: **Review the Operation of the C Code**

11. Click the **Resume** button to execute the code until the next breakpoint.

    The debugger stops each time it reaches line 13.

12. Adjust the Variables window to view the `shift_reg` variable. This updates the shift register.
13. Click the **Resume** button multiple times.
14. Click the **Terminate** button, shown in Figure 3-19, to end the debug session.

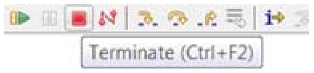


Figure 1-19: **Terminate Button**

15. Click **Synthesis** to return to the Synthesis perspective as shown in Figure 3-20.

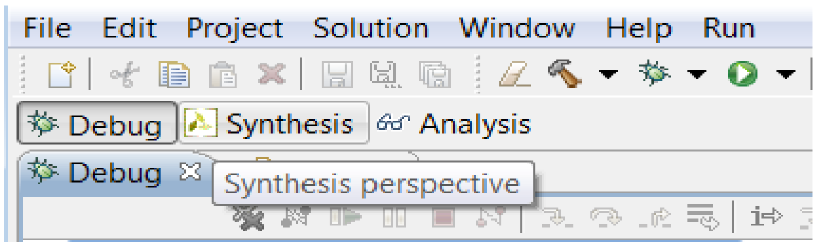


Figure 1-20: **Synthesis Perspective**

16. Click the **Run** button, shown in Figure 3-21, to run the design and verify the results.

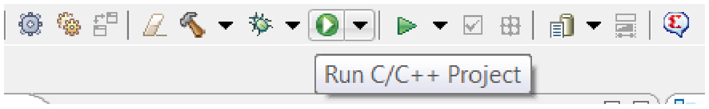
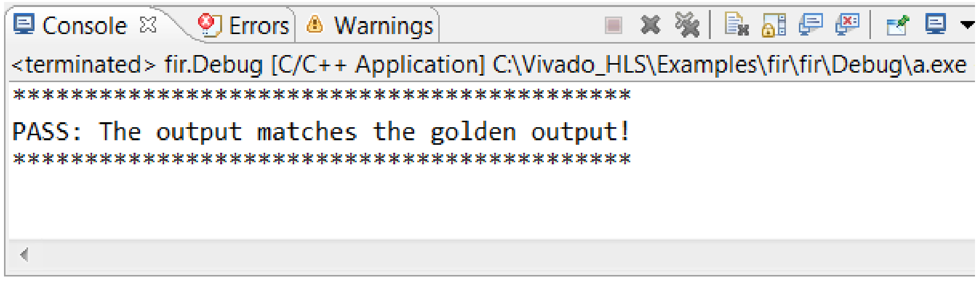


Figure 1-21: **Run C/C++ Project Button**

The results are shown in the console window (see Figure 3-22), 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.



```
<terminated> fir.Debug [C/C++ Application] C:\Vivado_HLS\Examples\fir\fir\Debug\a.exe
*******************************************
PASS: The output matches the golden output!
*******************************************
```

Figure 1-22: **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 execute each of these steps from the toolbar as shown in Figure 3-23. Because Synthesis is the first step in this process, the **Synthesis** button is located on the left side.

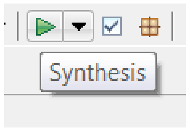


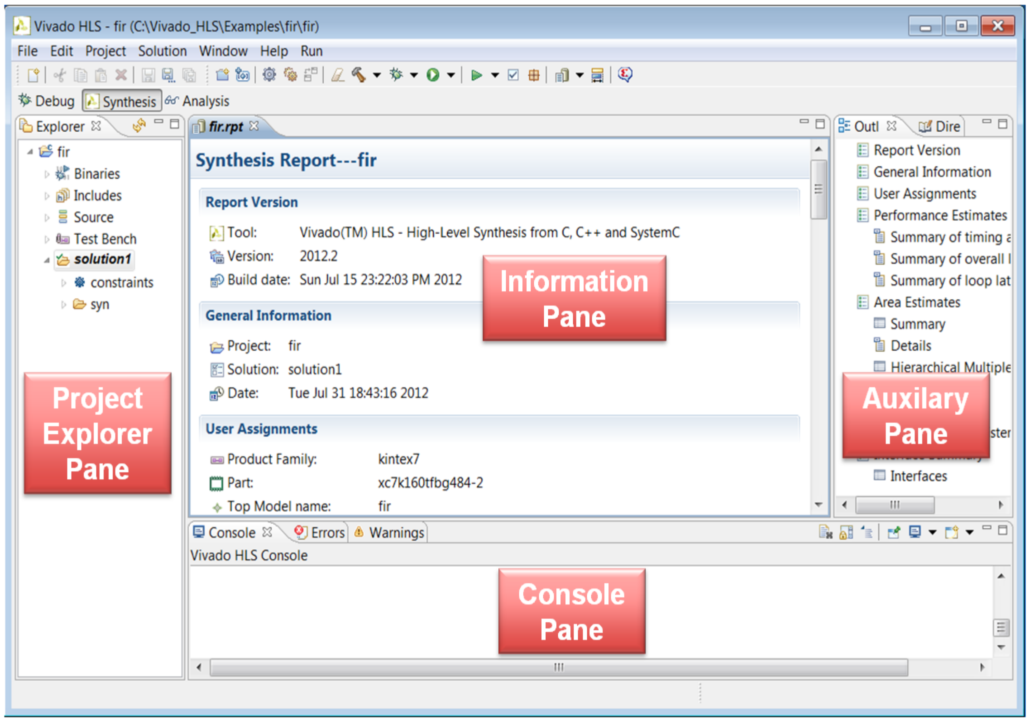
Figure 1-23: **Design Steps**

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 3-23.

Synthesis

Your design is now ready for synthesis. Click the **Synthesis** button, as shown in Figure 3-23.

When synthesis completes, the GUI updates with the results, as shown in Figure 3-24.



Synthesis Report---fir

Report Version

Tool: Vivado(TM) HLS - High-Level Synthesis from C, C++ and SystemC
Version: 2012.2
Build date: Sun Jul 15 23:22:03 PM 2012

General Information

Project: fir
Solution: solution1
Date: Tue Jul 31 18:43:16 2012

User Assignments

Product Family:	kintex7
Part:	xc7k160tfbg484-2
Top Model name:	fir

Figure 1-24: **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.

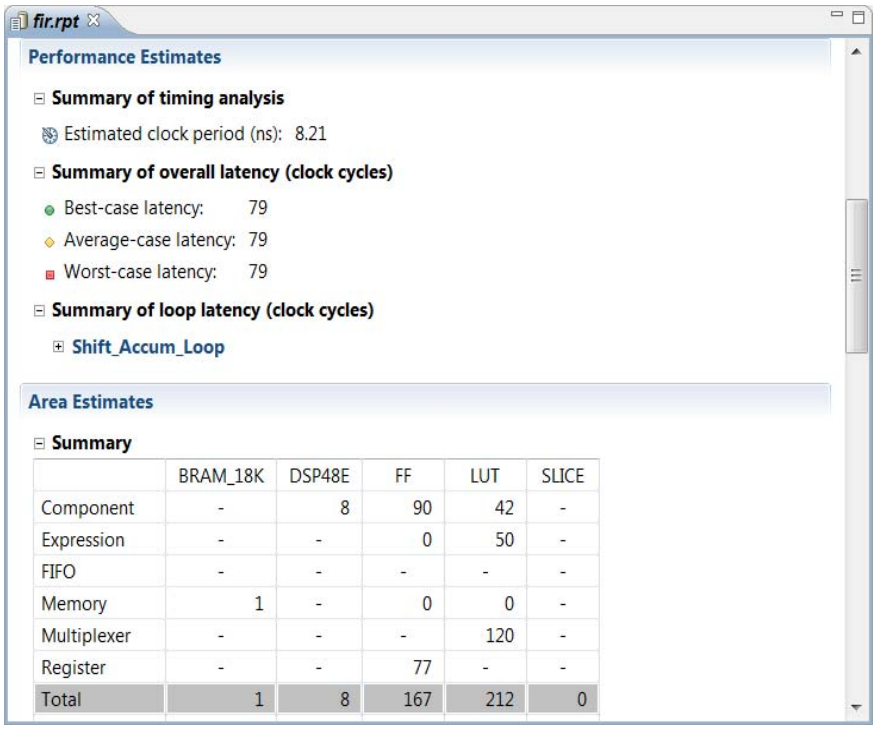
Table 1-2: **Synthesis Report Categories**

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 executed.
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.
	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 3-25.


fir.rpt

Performance Estimates

⊟ Summary of timing analysis

Estimated clock period (ns): 8.21

⊟ Summary of overall latency (clock cycles)

● Best-case latency: 79

◆ Average-case latency: 79

■ Worst-case latency: 79

⊟ Summary of loop latency (clock cycles)

⊞ Shift_Accum_Loop

Area Estimates

⊟ Summary

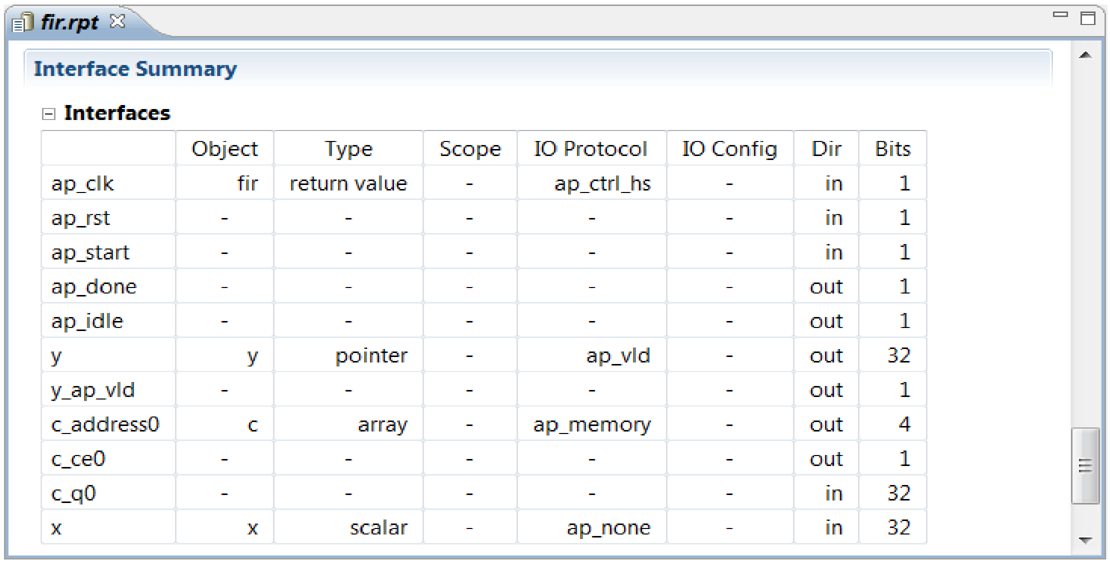
	BRAM_18K	DSP48E	FF	LUT	SLICE
Component	-	8	90	42	-
Expression	-	-	0	50	-
FIFO	-	-	-	-	-
Memory	1	-	0	0	-
Multiplexer	-	-	-	120	-
Register	-	-	77	-	-
Total	1	8	167	212	0

Figure 1-25: **solution1 Performance and Area Summary**

This report shows the initial solution to be:

- Meeting the clock frequency of 10ns
- Taking 79 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 3-26.



	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
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 1-26: **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), it indicates completion and its idle state by asserting `ap_done` and `ap_idle`, respectively.
- 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 3-27.

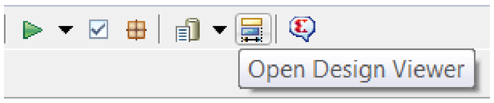
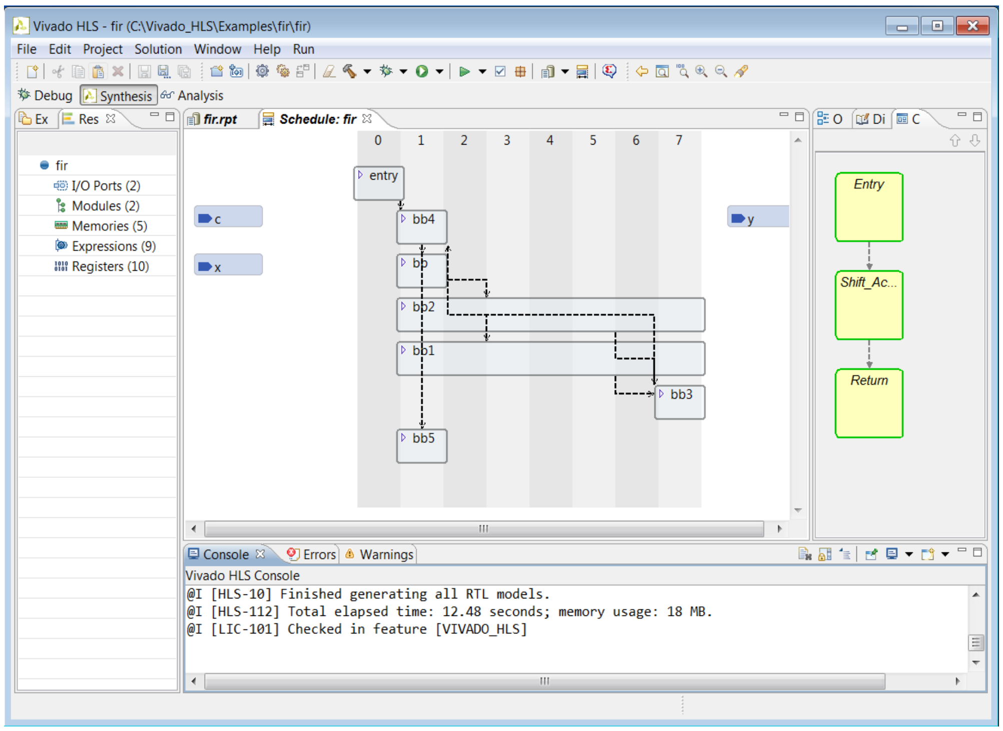


Figure 1-27: **Design Viewer Button**

The Design Viewer opens, as shown in Figure 3-28.



Vivado HLS Console

```
@I [HLS-10] Finished generating all RTL models.
@I [HLS-112] Total elapsed time: 12.48 seconds; memory usage: 18 MB.
@I [LIC-101] Checked in feature [VIVADO_HLS]
```

Figure 1-28: **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 3-29.

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

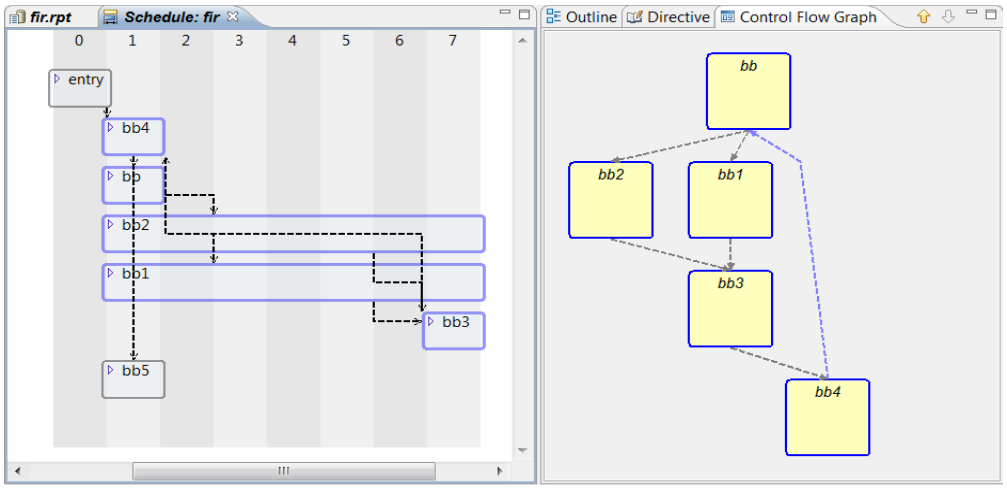


Figure 1-29: **Cross-Probing the CFG and Schedule Viewer**

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`.

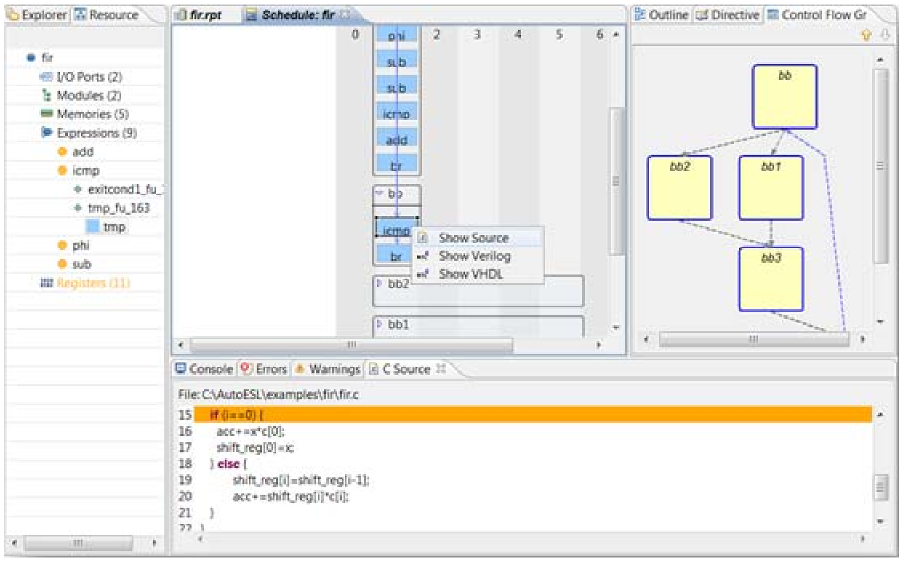
b. The loop proceeds to either block `bb2` or block `bb1`.

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 3-29.
2. Select the **`icmp`** operator and right-click to see the pop-up menu (see Figure 3-30).



File: C:\AutoESL\examples\fir\fir.c

```
  if (i==0) {
    acc+=x*c[0];
    shift_reg[0]=x;
  } else {
      shift_reg[i]=shift_reg[i-1];
      acc+=shift_reg[i]*c[i];
  }
}
```

Figure 1-30: **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 3-31 shows a more detailed view of how the Resource Viewer shows sharing (or lack of it, in this case).

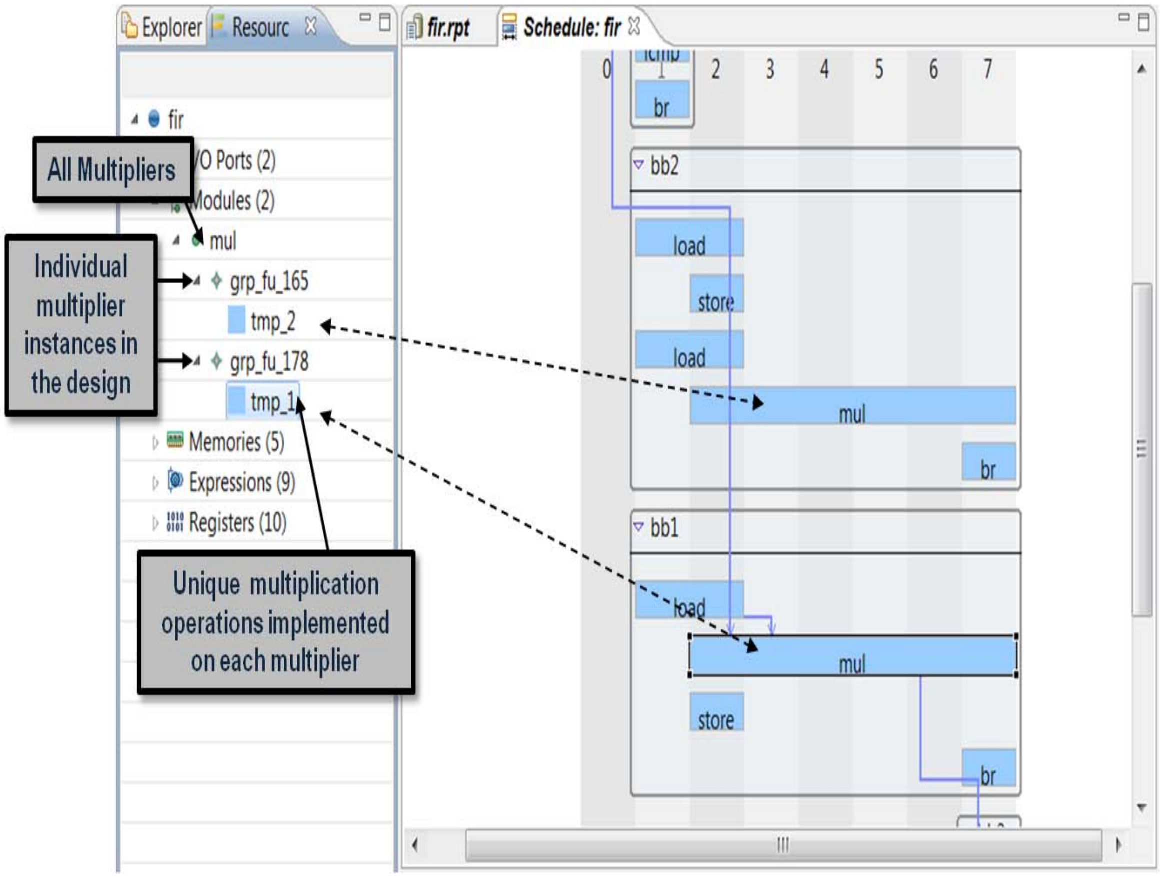


Figure 1-31: **View Sharing in the Design Viewer**

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 executed 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 3-25) 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 3-32.

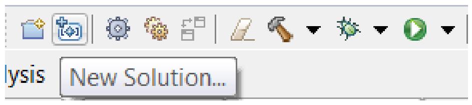


Figure 1-32: **New Solution Toolbar Button**

   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 3-33.

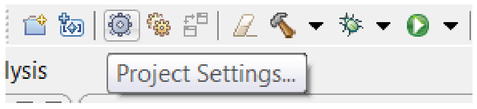


Figure 1-33: **Project Settings Button**

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**.

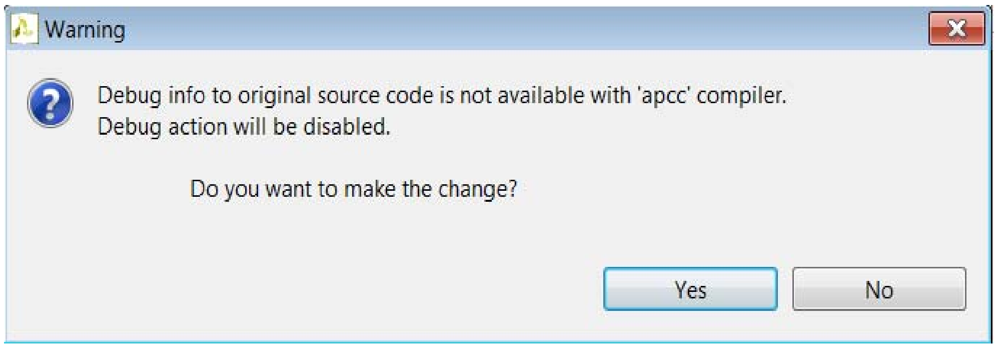
   The CFLAGS section is used to define any options required to compile the C program. This example uses the compiler option **`-D`**; however, all `gcc` options are supported in the CFLAGS section (**`-I`**<*include path*> etc.).

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.
   e. Select the **Use AutoCC Compiler** check box.

      The warning dialog box opens, as shown in Figure 3-34.


Warning

Debug info to original source code is not available with 'apcc' compiler.
Debug action will be disabled.

Do you want to make the change?

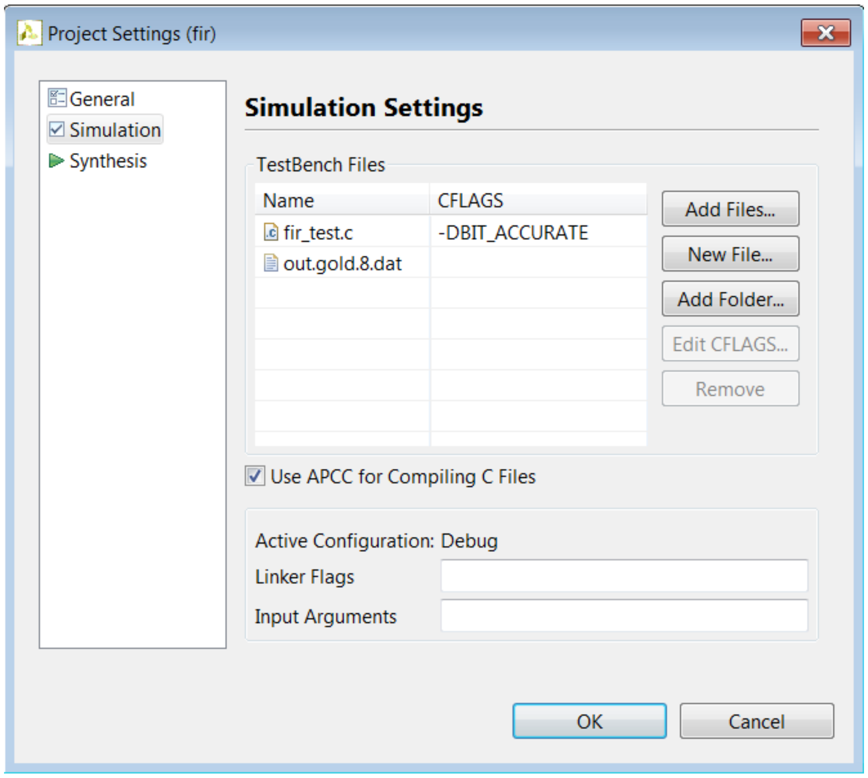
Figure 1-34: **Warning Dialog Box**

Note: Designs compiled with `AutoCC` simulate with bit-accurate behavior but cannot be analyzed in the debug environment.

   f. Click **Yes** to accept this warning.

      The updated Simulation section is shown in Figure 3-35.
   g. Click **OK**.

The types used to define bit-accurate behavior in a C function require special handling and must be compiled using the Vivado HLS C compiler `AutoCC`. This is not required for bit-accurate C++ and SystemC types, only bit-accurate C types.


Project Settings (fir)

General
Simulation
Synthesis

Simulation Settings

TestBench Files

Name	CFLAGS
fir_test.c	-DBIT_ACCURATE
out.gold.8.dat	

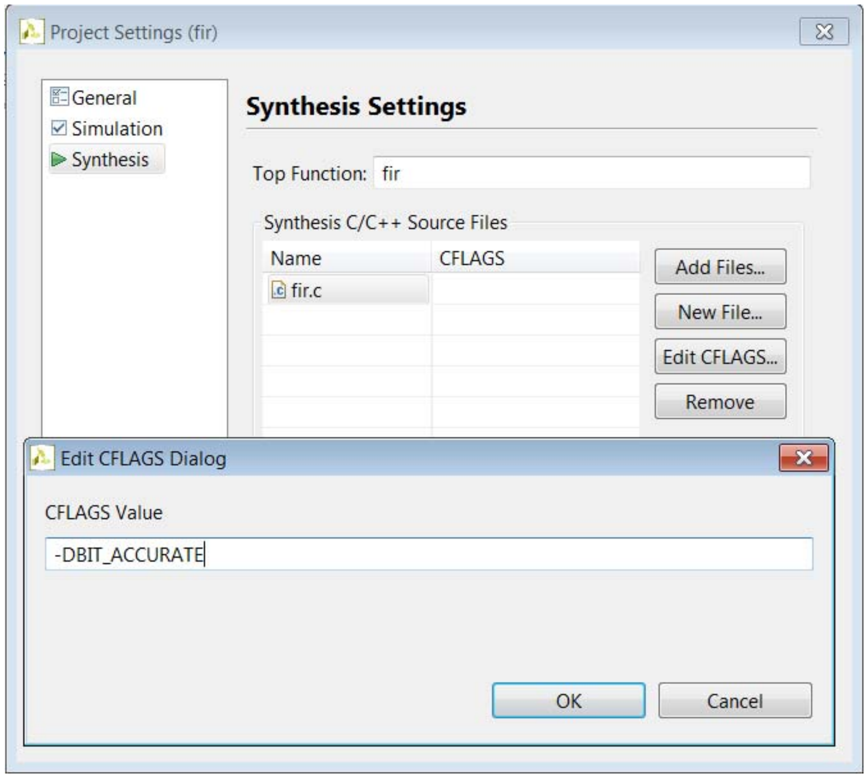
[✓] Use APCC for Compiling C Files

Active Configuration: Debug
Linker Flags
Input Arguments

Figure 1-35: **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 3-36 shows the settings for the CFLAGS to synthesize the design using bit-accurate types.


Project Settings (fir)

- General
- Simulation
- Synthesis

Synthesis Settings

Top Function: fir

Synthesis C/C++ Source Files

Name	CFLAGS
fir.c	

Edit CFLAGS Dialog

CFLAGS Value

-DBIT_ACCURATE

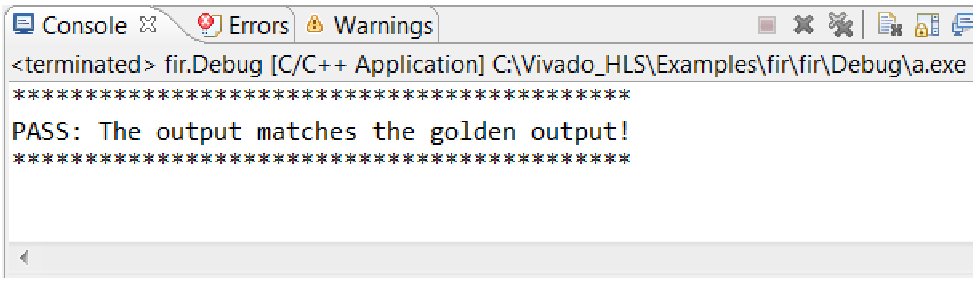
Figure 1-36: **CFLAGS Settings Dialog Box**

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

5. Click the **Build** toolbar button to recompile the function.
6. Click the **Run** toolbar button to re-execute the C simulation.

   The output displays in the console window. See Figure 3-37.

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



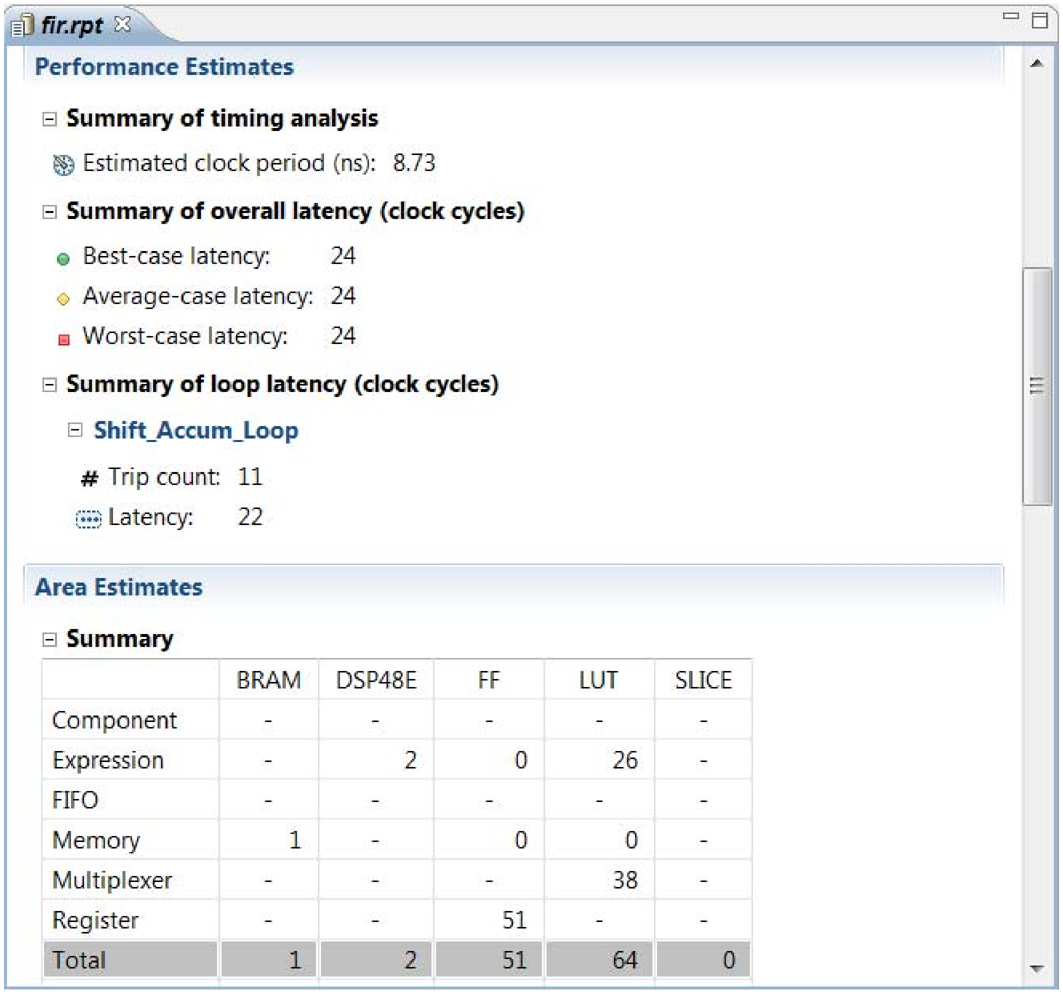
```
<terminated> fir.Debug [C/C++ Application] C:\Vivado_HLS\Examples\fir\fir\Debug\a.exe
*******************************************
PASS: The output matches the golden output!
*******************************************
```

Figure 1-37: **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 3-38, where only two DSP48s are used and the estimated clock frequency is now faster.


fir.rpt

Performance Estimates

Summary of timing analysis

Estimated clock period (ns): 8.73

Summary of overall latency (clock cycles)

Best-case latency:	24
Average-case latency:	24
Worst-case latency:	24

Summary of loop latency (clock cycles)

Shift_Accum_Loop

Trip count:	11
Latency:	22

Area Estimates

Summary

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

Figure 1-38: **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 3-39).

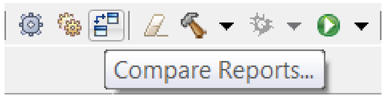
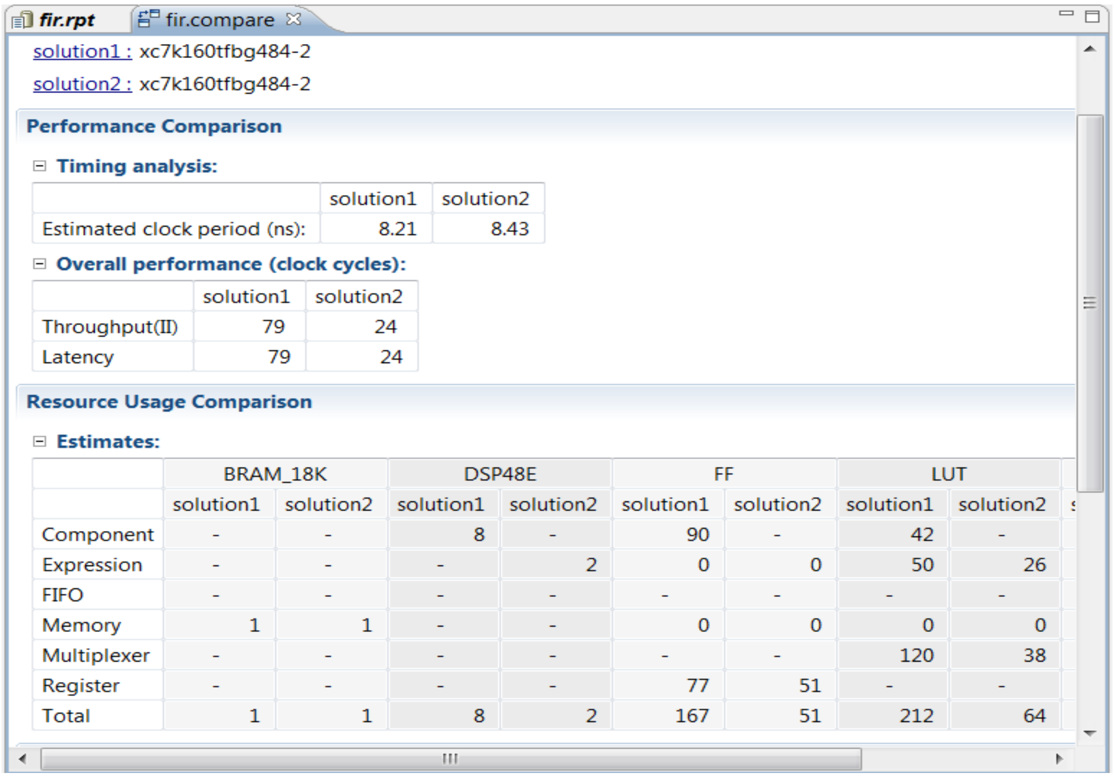


Figure 1-39: **Compare Reports Button**

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

Figure 3-40 shows the comparison of the reports for solution1 and solution2.



fir.rpt | fir.compare

solution1 : xc7k160tfbg484-2
solution2 : xc7k160tfbg484-2

Performance Comparison

Timing analysis:

	solution1	solution2
Estimated clock period (ns):	8.21	8.43

Overall performance (clock cycles):

	solution1	solution2
Throughput(II)	79	24
Latency	79	24

Resource Usage Comparison

Estimates:

	BRAM_18K		DSP48E		FF		LUT	
	solution1	solution2	solution1	solution2	solution1	solution2	solution1	solution2
Component	-	-	8	-	90	-	42	-
Expression	-	-	-	2	0	0	50	26
FIFO	-	-	-	-	-	-	-	-
Memory	1	1	-	-	0	0	0	0
Multiplexer	-	-	-	-	-	-	120	38
Register	-	-	-	-	77	51	-	-
Total	1	1	8	2	167	51	212	64

Figure 1-40: **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.

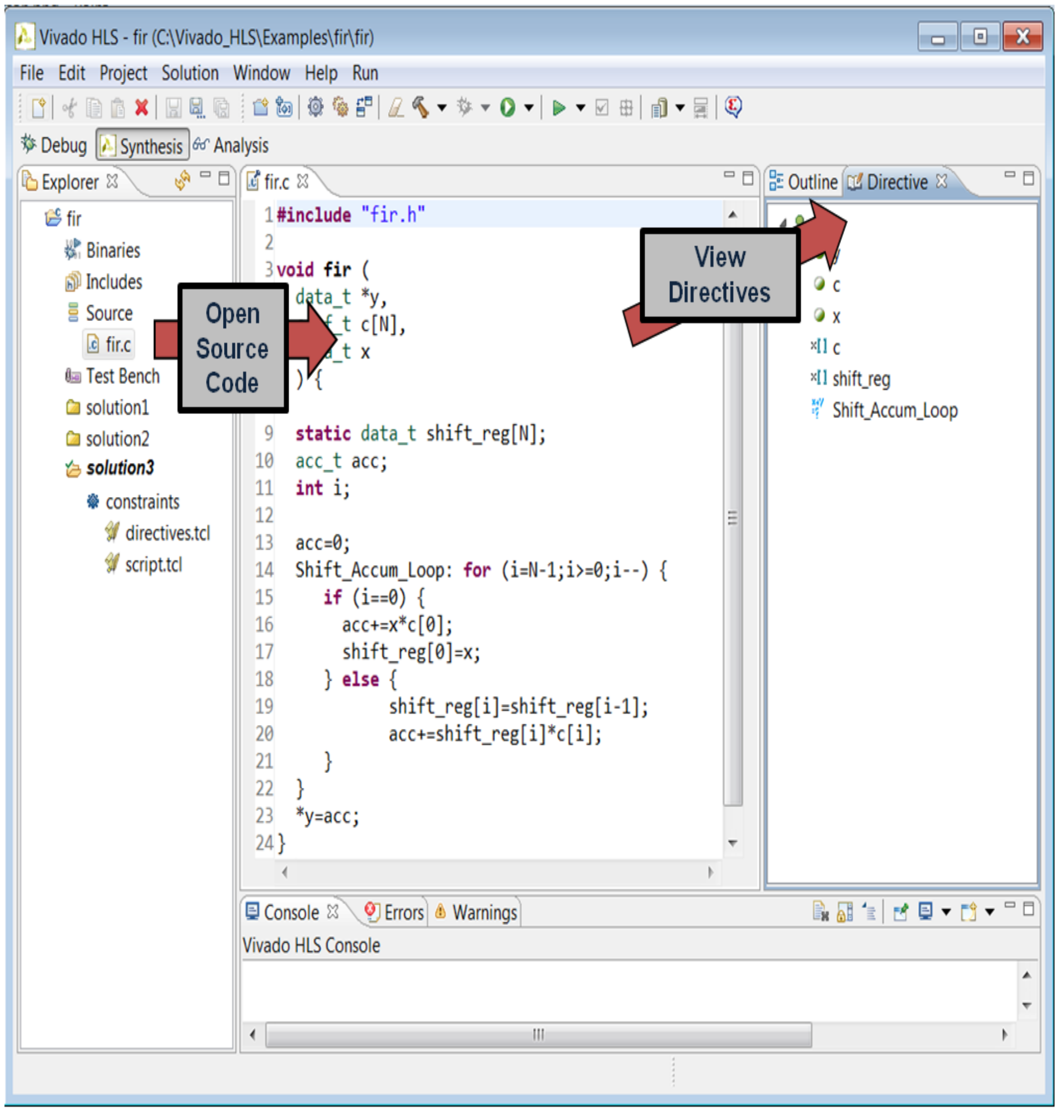
   When `solution3` opens, confirm that `solution3` is highlighted in bold in the Project Explorer pane, indicating that it is the current active solution.

   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 3-41).



```
#include "fir.h"

void fir (
  data_t *y,
     _t c[N],
     _t x
  ) {

  static data_t shift_reg[N];
  acc_t acc;
  int i;

  acc=0;
  Shift_Accum_Loop: for (i=N-1;i>=0;i--) {
     if (i==0) {
       acc+=x*c[0];
       shift_reg[0]=x;
     } else {
          shift_reg[i]=shift_reg[i-1];
          acc+=shift_reg[i]*c[i];
     }
  }
  *y=acc;
}
```

Figure 1-41: **Adding Optimization Directives**

2. Double-click **`fir.c`** to open the file in the Information pane.
3. Click the **Directive Tab** (see Figure 3-41).

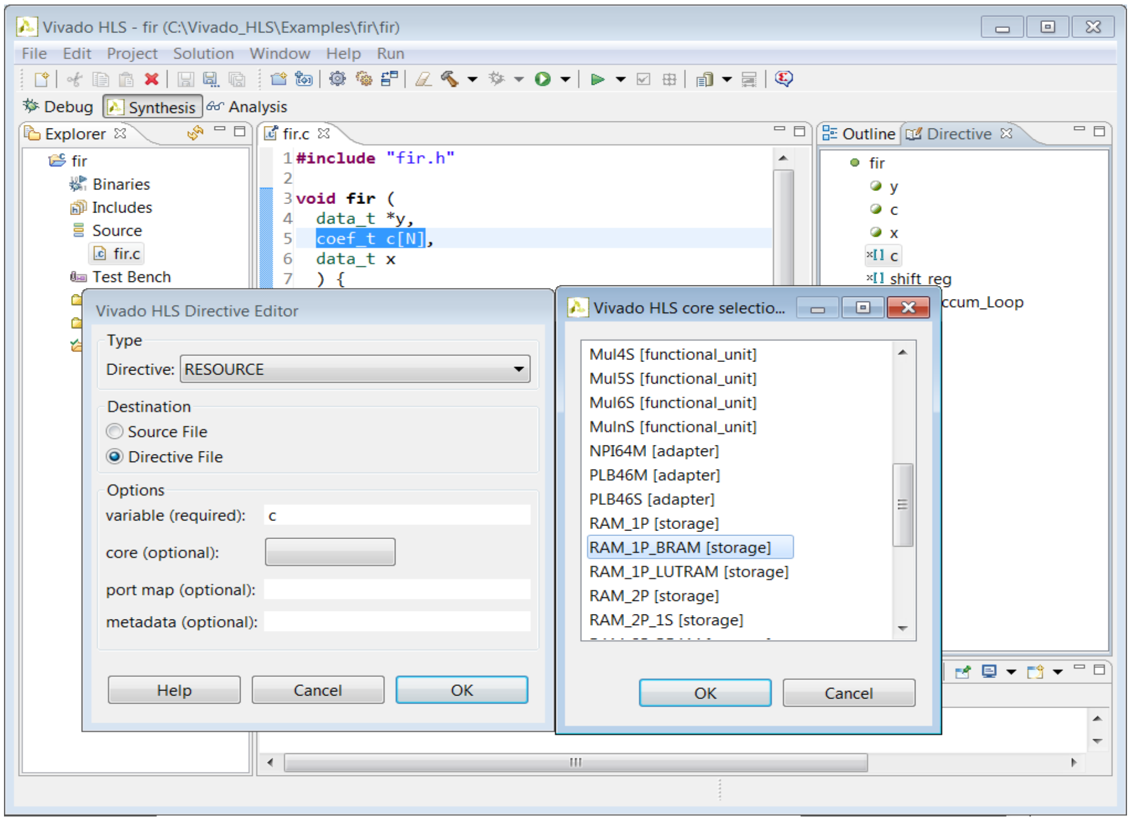
   You can now apply the optimization directives to the design.

4. In the Directive tab, select the **`c`** argument/port (green dot) or the array `c`.
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 3-42.

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

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



```
#include "fir.h"

void fir (
  data_t *y,
  coef_t c[N],
  data_t x
  ) {
```

Vivado HLS Directive Editor

Type
Directive: RESOURCE

Destination
Source File [✓ Directive File]

Options
variable (required): c
core (optional):
port map (optional):
metadata (optional):

Vivado HLS core selectio...

```
Mul4S [functional_unit]
Mul5S [functional_unit]
Mul6S [functional_unit]
MulnS [functional_unit]
NPI64M [adapter]
PLB46M [adapter]
PLB46S [adapter]
RAM_1P [storage]
RAM_1P_BRAM [storage]
RAM_1P_LUTRAM [storage]
RAM_2P [storage]
RAM_2P_1S [storage]
```

Figure 1-42: **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.

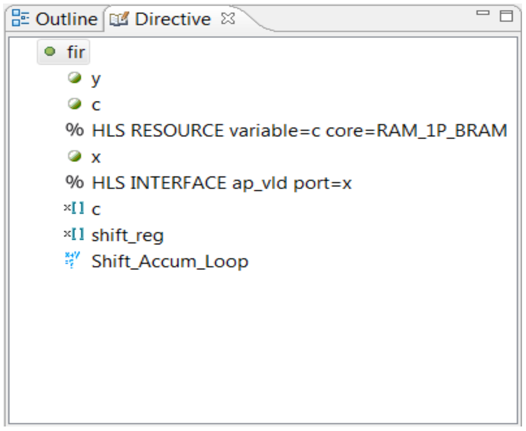
The interface ports created (the number of address ports) are determined by pins on the RAM_1P_BRAM core. A complete description of the cores in the Vivado HLS library is provided in the *Vivado Design Suite User Guide: High-Level Synthesis (UG902) > High-Level Synthesis Operator and Core Guide chapter.*

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 3-43. Select any incorrect directive and use the mouse right-click to modify it.



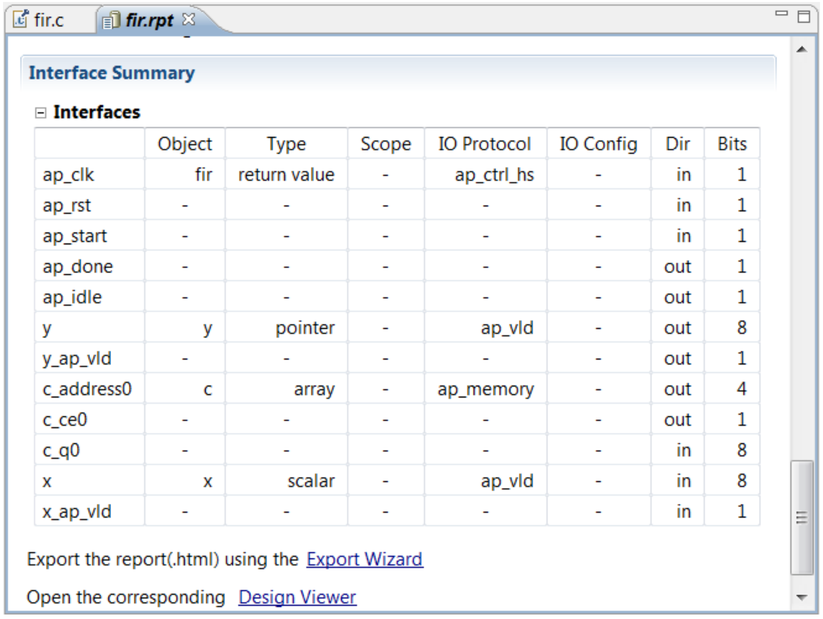
```
fir
  y
  c
  % HLS RESOURCE variable=c core=RAM_1P_BRAM
  x
  % HLS INTERFACE ap_vld port=x
  c
  shift_reg
  Shift_Accum_Loop
```

Figure 1-43: **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 3-44 shows the interfaces are now correctly defined.



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
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

Export the report(.html) using the Export Wizard

Open the corresponding Design Viewer

Figure 1-44: **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

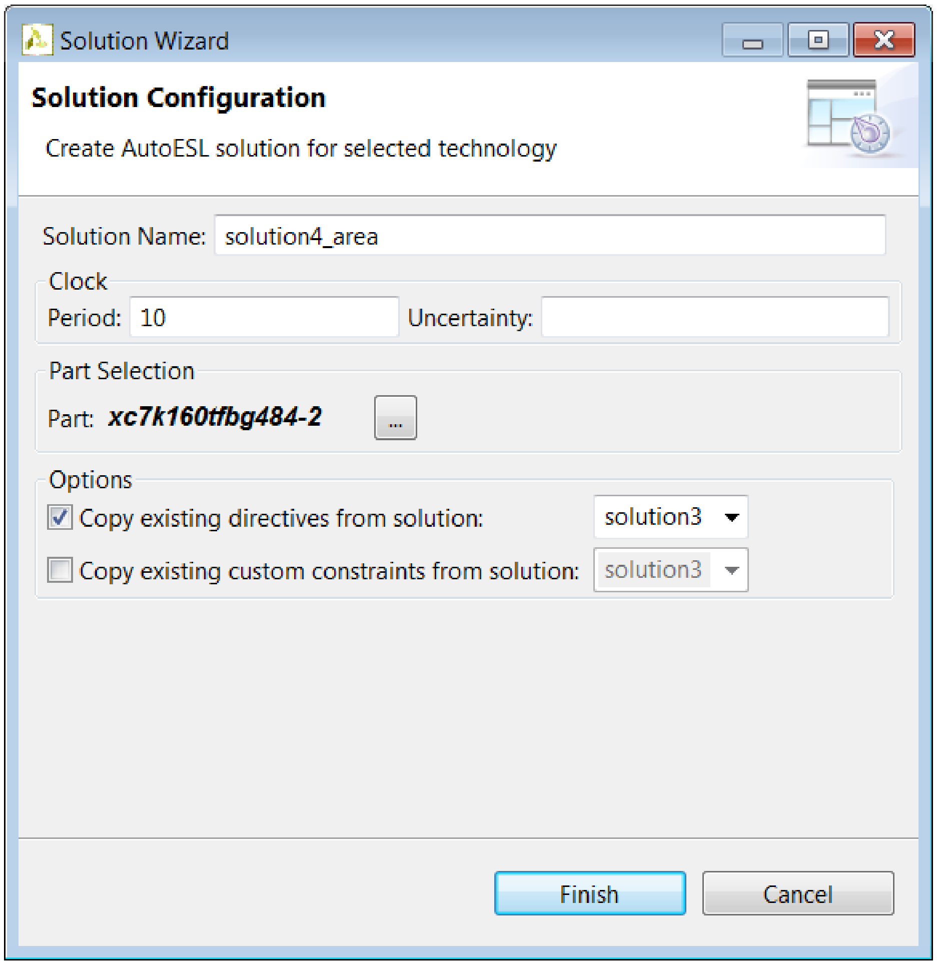
The design in `solution3` represents the starting point for further optimizations. Begin by creating a new solution, as shown in Figure 3-45.

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. Select the **Copy existing directives from solution** check box and select **`solution3`** from the menu.

   The IO directives specified in `solution3` copy into `solution4_area`.
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.



Solution Name: solution4_area

Clock
Period: 10 Uncertainty:

Part Selection
Part: xc7k160tfbg484-2

Options
[✓] Copy existing directives from solution: solution3
[ ] Copy existing custom constraints from solution: solution3

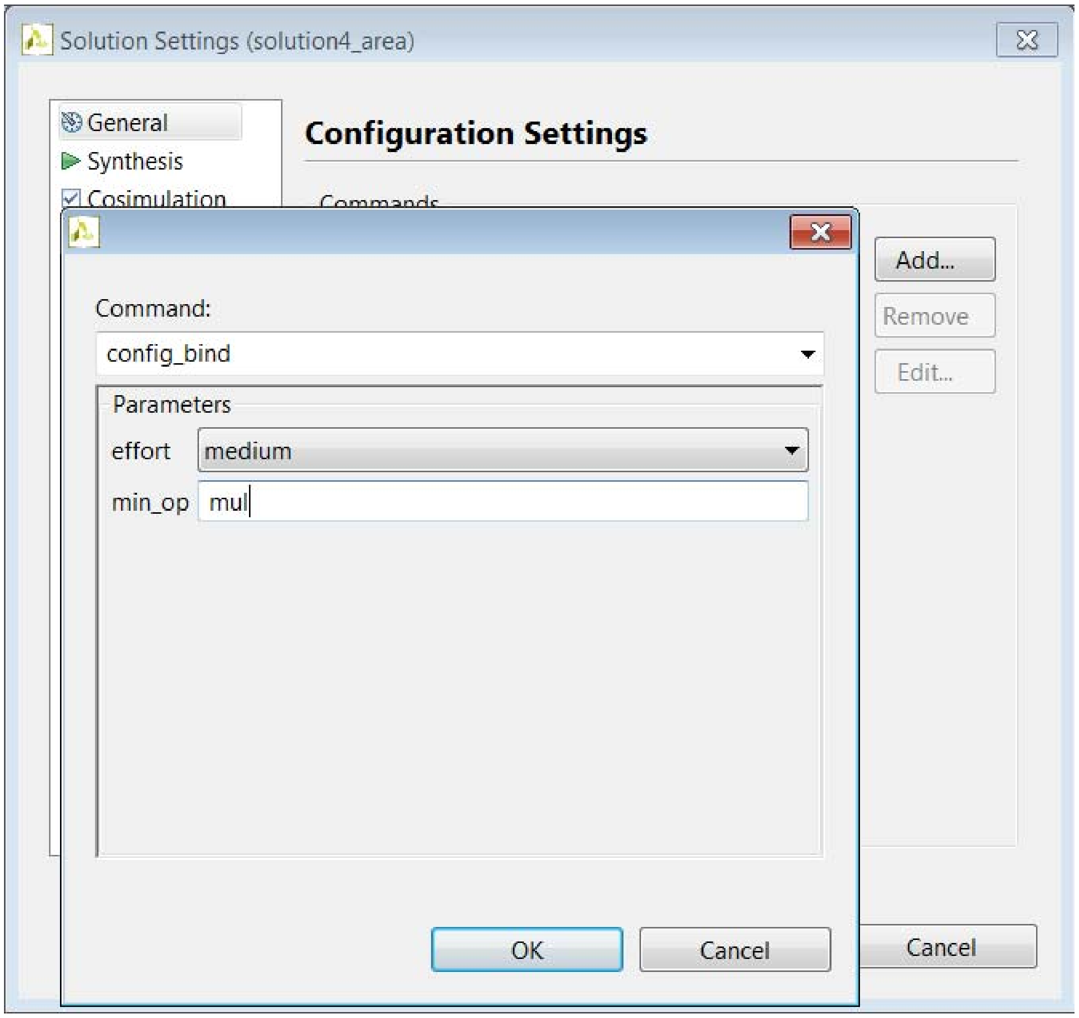
Figure 1-45: **Create solution4_area**

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 1-46.
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.



Command: config_bind

Parameters

effort: medium

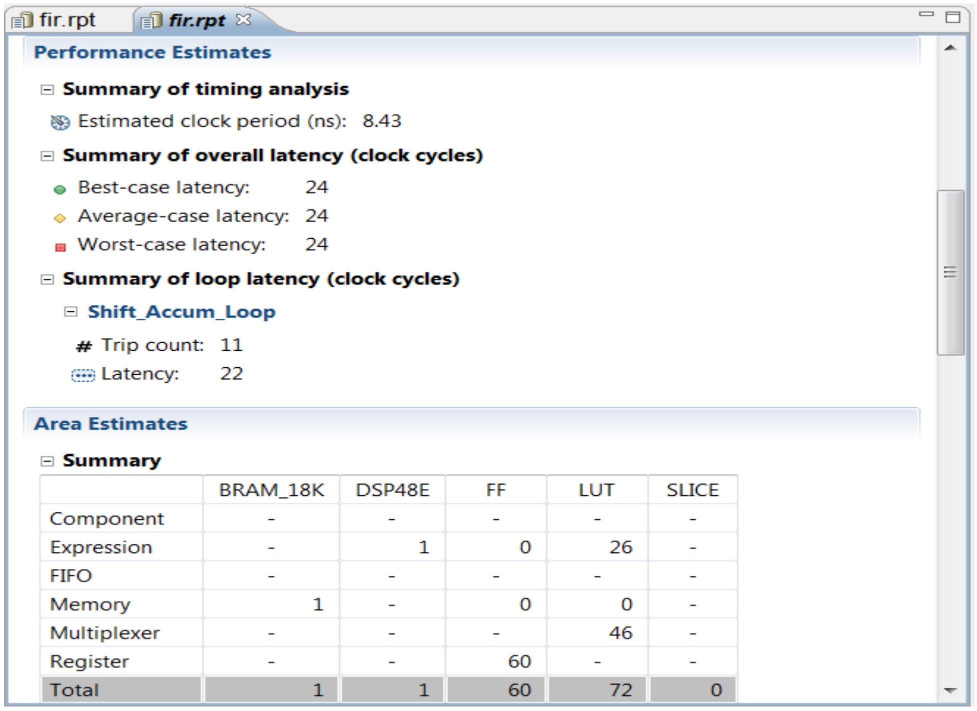
min_op: mul

Figure 1-46: **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 3-47.



Performance Estimates

Summary of timing analysis

Estimated clock period (ns): 8.43

Summary of overall latency (clock cycles)

Best-case latency:	24
Average-case latency:	24
Worst-case latency:	24

Summary of loop latency (clock cycles)

Shift_Accum_Loop

Trip count:	11
Latency:	22

Area Estimates

Summary

	BRAM_18K	DSP48E	FF	LUT	SLICE
Component	-	-	-	-	-
Expression	-	1	0	26	-
FIFO	-	-	-	-	-
Memory	1	-	0	0	-
Multiplexer	-	-	-	46	-
Register	-	-	60	-	-
Total	1	1	60	72	0

Figure 1-47: **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.

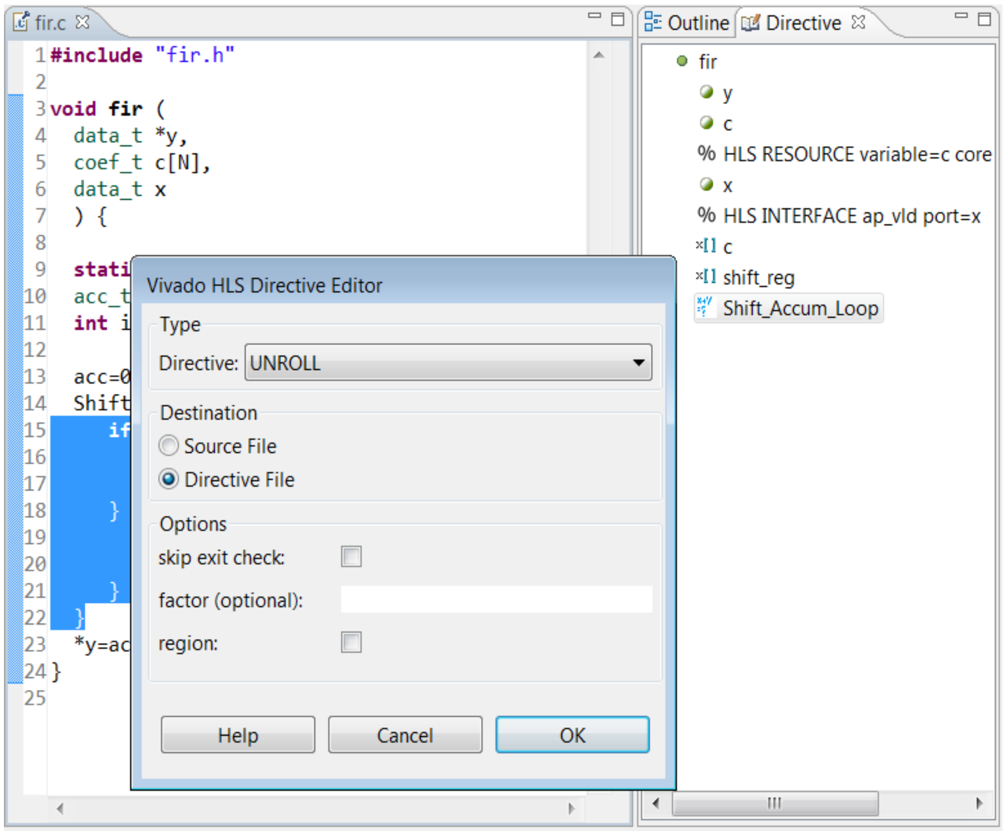
1. Click the **New Solution** button to create a new solution.
2. Name the solution `solution5_throughput`.
3. Select the **Copy existing directives from solution** check box.
4. Select **`solution3`** from the drop-down menu.

   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 3-48, explain how to unroll the loop.



```
#include "fir.h"

void fir (
  data_t *y,
  coef_t c[N],
  data_t x
  ) {

  stati
  acc_t
  int i

  acc=0
  Shift
    if


    }


    }
  }
  *y=ac
}
```

```
fir
  y
  c
  % HLS RESOURCE variable=c core
  x
  % HLS INTERFACE ap_vld port=x
  c
  shift_reg
  Shift_Accum_Loop
```

Vivado HLS Directive Editor

Type
Directive: UNROLL

Destination
Source File [✓ Directive File]

Options
skip exit check: ☐
factor (optional):
region: ☐

Figure 1-48: **Unrolling FOR Loop**

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

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

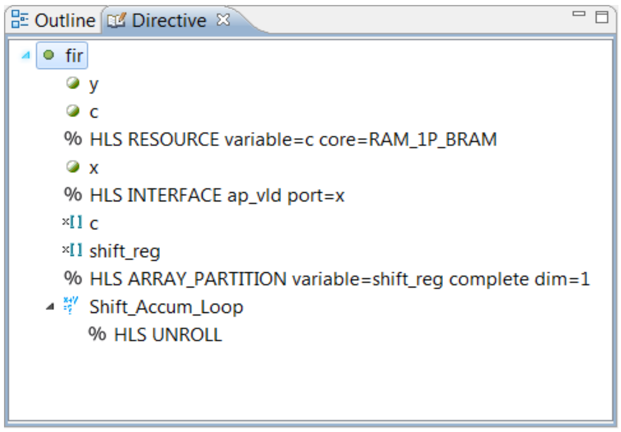
2. Right-click and select **Insert Directives**.
3. From the Directive drop-down menu, select **Unroll**.
4. Select **OK** to apply the directive.

   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 3-49.



```
fir
    y
    c
    % HLS RESOURCE variable=c core=RAM_1P_BRAM
    x
    % HLS INTERFACE ap_vld port=x
    c
    shift_reg
    % HLS ARRAY_PARTITION variable=shift_reg complete dim=1
    Shift_Accum_Loop
        % HLS UNROLL
```

Figure 1-49: **solution5_throughout Directives**

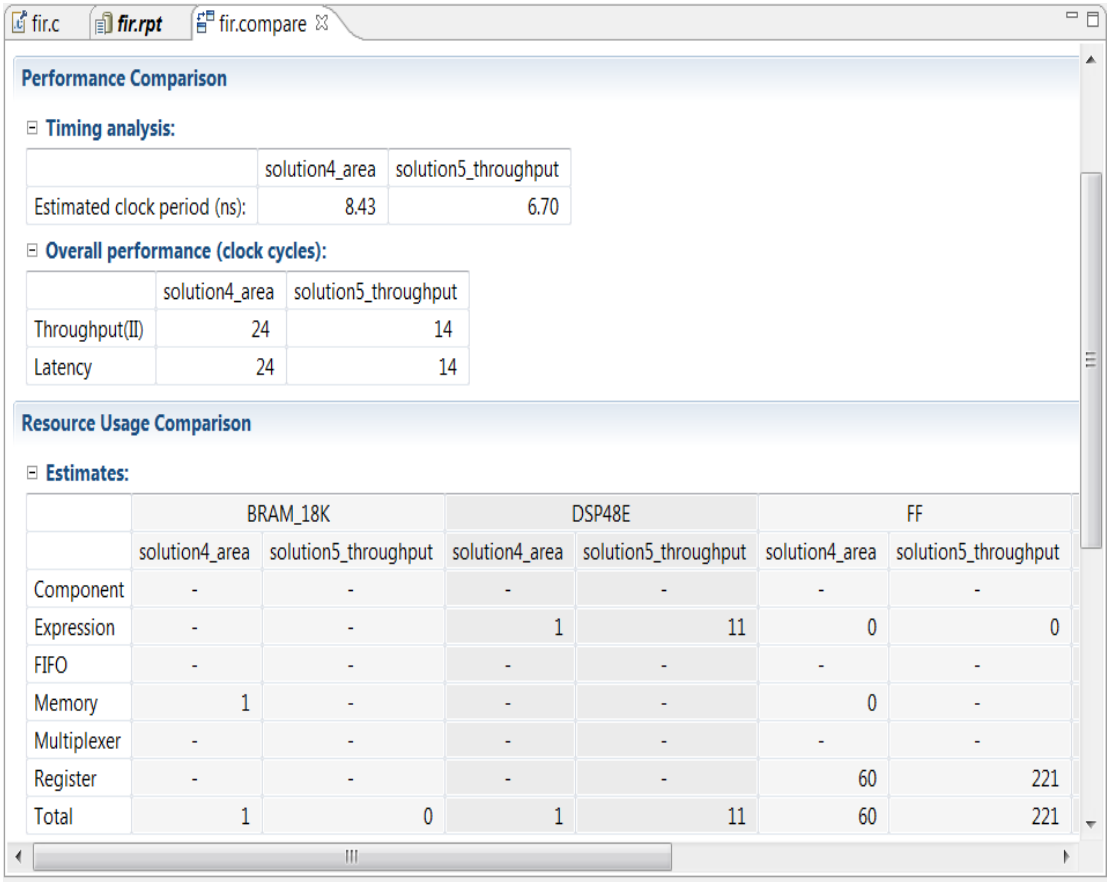
Step 3: Synthesis

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 3-50 shows the comparison of the reports from `solution4_area` and `solution5` (the LUTS are not shown in Figure 3-50 due to the wide nature of the report).



fir.c | fir.rpt | fir.compare

Performance Comparison

Timing analysis:

	solution4_area	solution5_throughput
Estimated clock period (ns):	8.43	6.70

Overall performance (clock cycles):

	solution4_area	solution5_throughput
Throughput(II)	24	14
Latency	24	14

Resource Usage Comparison

Estimates:

	BRAM_18K		DSP48E		FF	
	solution4_area	solution5_throughput	solution4_area	solution5_throughput	solution4_area	solution5_throughput
Component	-	-	-	-	-	-
Expression	-	-	1	11	0	0
FIFO	-	-	-	-	-	-
Memory	1	-	-	-	0	-
Multiplexer	-	-	-	-	-	-
Register	-	-	-	-	60	221
Total	1	0	1	11	60	221

Figure 1-50: **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 3-51.

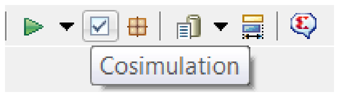
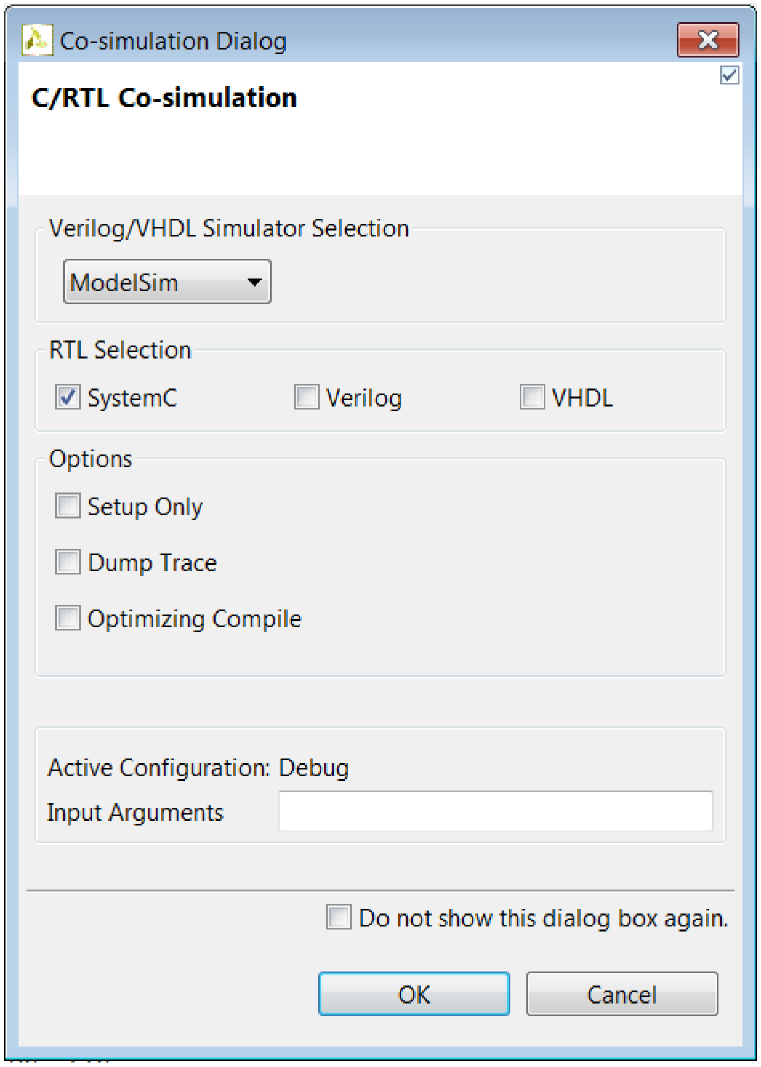


Figure 1-51: **Simulation Toolbar Button**

The co-simulation dialog opens, as shown in Figure 3-52.


Co-simulation Dialog

C/RTL Co-simulation

Verilog/VHDL Simulator Selection: ModelSim

RTL Selection: [✓ SystemC] Verilog VHDL

Options: Setup Only, Dump Trace, Optimizing Compile

Active Configuration: Debug

Input Arguments

Do not show this dialog box again.

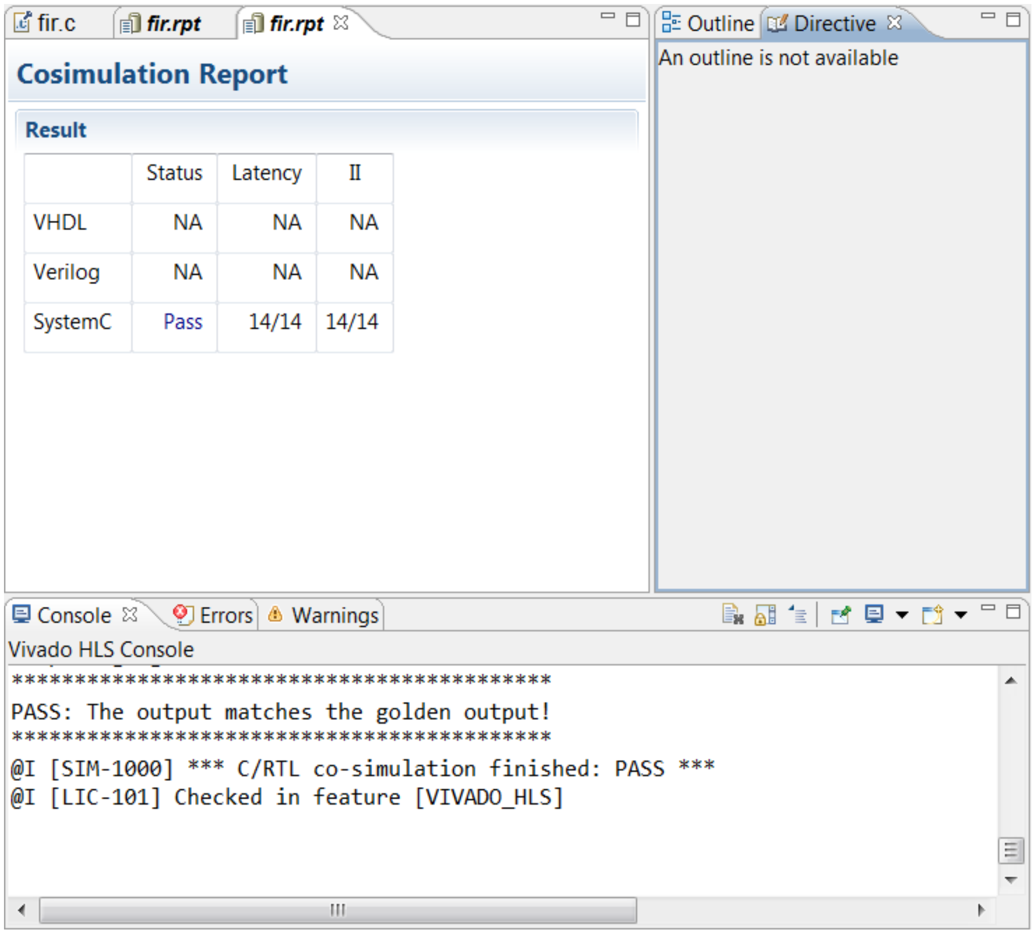
Figure 1-52: **RTL Verification Menu**

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 3-53). 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.



Cosimulation Report

Result

	Status	Latency	II
VHDL	NA	NA	NA
Verilog	NA	NA	NA
SystemC	Pass	14/14	14/14

```
*******************************************
PASS: The output matches the golden output!
*******************************************
@I [SIM-1000] *** C/RTL co-simulation finished: PASS ***
@I [LIC-101] Checked in feature [VIVADO_HLS]
```

Figure 1-53: **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 3-54.

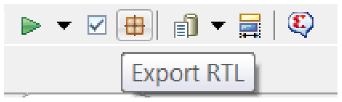
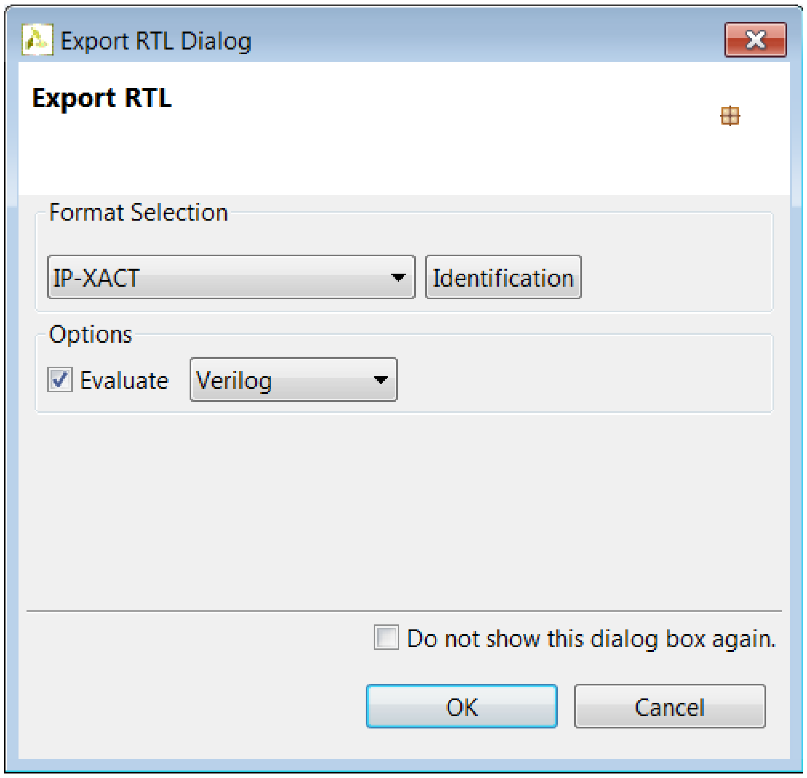


Figure 1-54: **Export RTL Toolbar Button**

The Export RTL dialog opens, as shown in Figure 3-55.


Export RTL Dialog

Export RTL

Format Selection

IP-XACT | Identification

Options

☑ Evaluate | Verilog

☐ Do not show this dialog box again.

OK | Cancel

Figure 1-55: **Export RTL Menu**

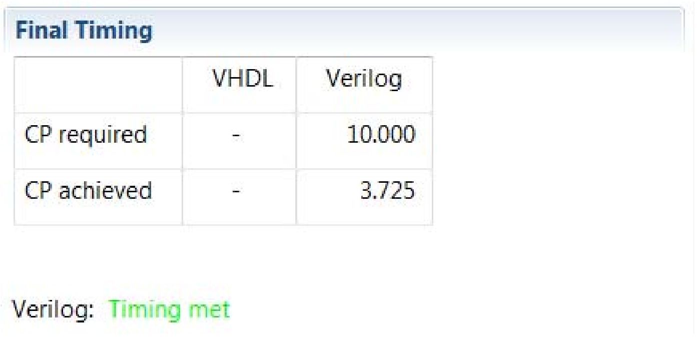
2. In this example, the design will be exported to IP-XACT format. Refer to the *Vivado Design Suite User Guide: High-Level Synthesis (UG902)* for an explanation of all export formats and how to import them into the appropriate Xilinx design tool.
3. In this example RTL synthesis will be performed: select the evaluate option. For VHDL or Verilog, select from the drop-down menu. In this example, Verilog is used, as shown in Figure 3-55.
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 3-56).



Final Timing

	VHDL	Verilog
CP required	-	10.000
CP achieved	-	3.725

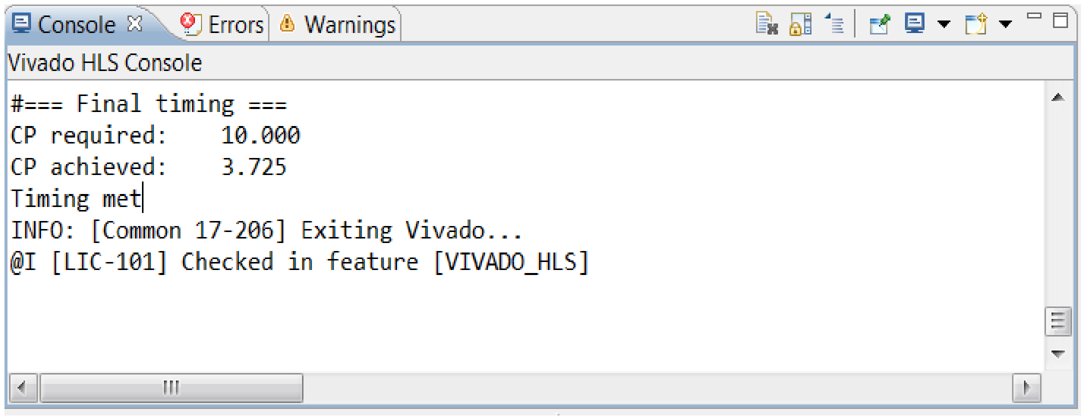
Verilog: Timing met

Figure 1-56: **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 3-57.



```
#=== Final timing ===
CP required:    10.000
CP achieved:    3.725
Timing met
INFO: [Common 17-206] Exiting Vivado...
@I [LIC-101] Checked in feature [VIVADO_HLS]
```

Figure 1-57: **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 2012.2 > 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_file fir.c -cflags "-DBIT_ACCURATE"
add_file -tb out.gold.8.dat
add_file -tb fir_test.c -cflags "   -DBIT_ACCURATE"
open_solution "solution5_throughput"
set_part  {xc6vlx240tff1156-2}
create_clock -period 10

source "./fir.prj/solution5_throughput/directives.tcl"
elaborate
autosyn
```

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_file fir.c -cflags "-DBIT_ACCURATE"
add_file -tb out.gold.8.dat
add_file -tb fir_test.c -cflags "   -DBIT_ACCURATE"
open_solution "solution5_throughput"
set_part  {xc6vlx240tff1156-2}
create_clock -period 10

source "./directives.tcl"
elaborate
```

```
autosyn
```

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.

Chapter 2

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 3-1.

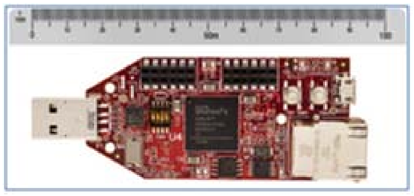


Figure 2-1: **Avnet Spartan-6 LX9 MicroBoard**

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 3-2, page 61 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

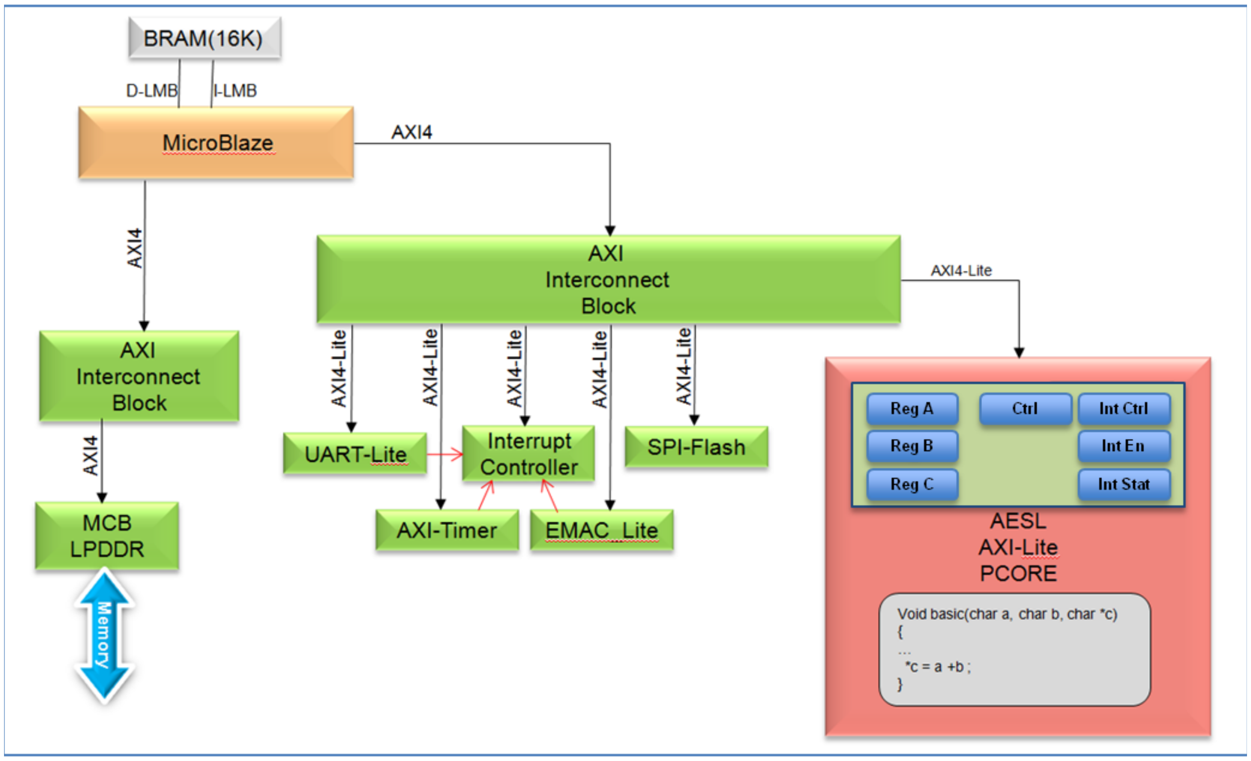


Figure 2-2: **Block Diagram**

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 3, Vivado HLS: Introduction Tutorial for details about how to create an Vivado HLS project.

Note: Figure 3-3, page 63 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.

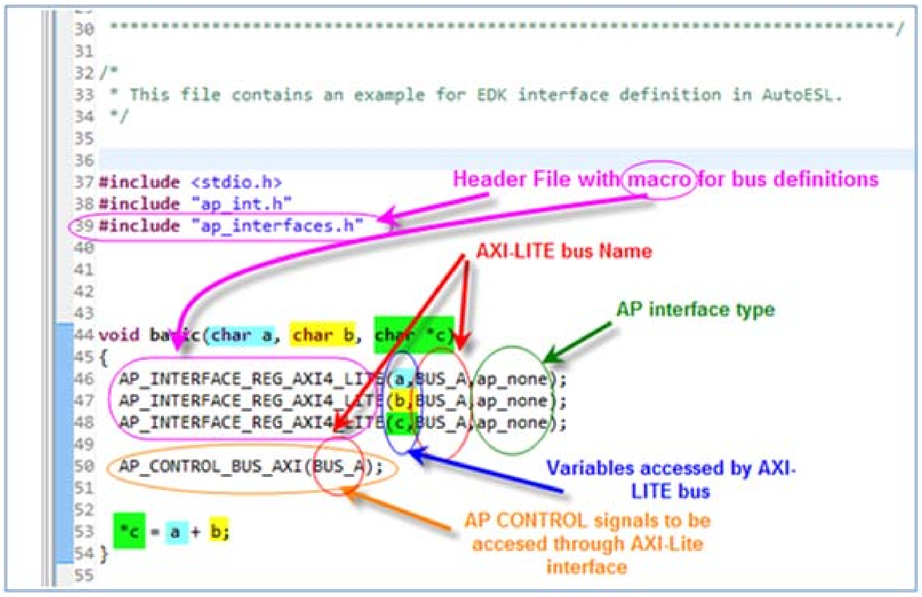
Table 3-1, page 66 describes all the registers created by Vivado HLS for the generated Pcore.

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 3-3. Refer to *Vivado HLS Tutorial: Introduction (UG871),* located in the `/doc` directory where the Vivado HLS tool is installed.



```
******************************************************************************/

/*
 * This file contains an example for EDK interface definition in AutoESL.
 */


#include <stdio.h>
#include "ap_int.h"
#include "ap_interfaces.h"




void basic(char a, char b, char *c)
{
  AP_INTERFACE_REG_AXI4_LITE(a,BUS_A,ap_none);
  AP_INTERFACE_REG_AXI4_LITE(b,BUS_A,ap_none);
  AP_INTERFACE_REG_AXI4_LITE(c,BUS_A,ap_none);

  AP_CONTROL_BUS_AXI(BUS_A);


  *c = a + b;
}

```

Header File with macro for bus definitions
AXI-LITE bus Name
AP interface type
Variables accessed by AXI-LITE bus
AP CONTROL signals to be accesed through AXI-Lite interface

Figure 2-3: **C Code**

2. Click the **Synthesis** button, shown in Figure 3-4.

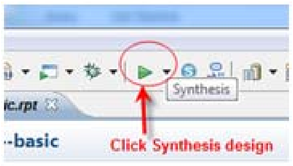


Figure 2-4: **Synthesis Button**

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

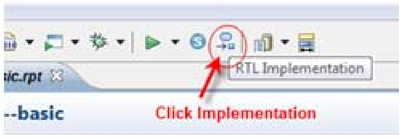
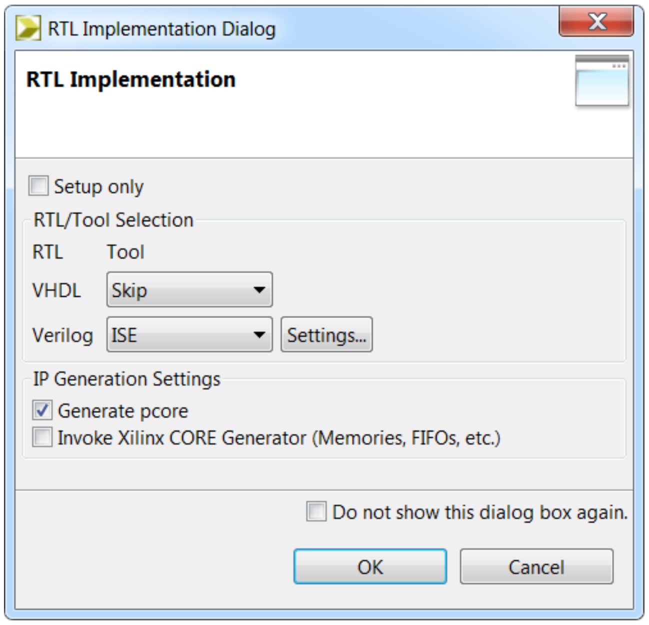


Figure 2-5: **Export RTL Button**

The RTL Implementation dialog box opens, shown in Figure 3-6.


RTL Implementation Dialog

RTL Implementation

☐ Setup only

RTL/Tool Selection

RTL	Tool
VHDL	Skip
Verilog	ISE

Settings...

IP Generation Settings

☑ Generate pcore

☐ Invoke Xilinx CORE Generator (Memories, FIFOs, etc.)

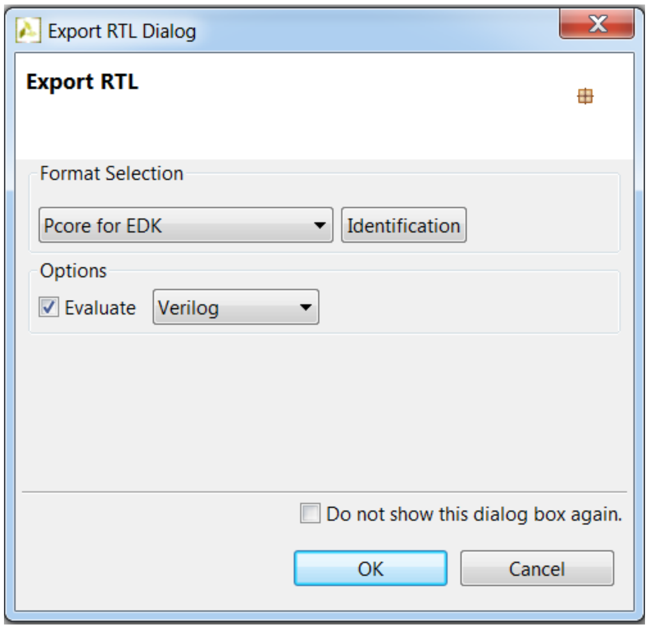
☐ Do not show this dialog box again.

OK Cancel

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

4. Select the **Generate pcore** check box.

   In addition to creating the Pcore, these settings execute ISE for RTL synthesis. The ISE executable must be in the Windows search path for ISE to launch; refer to the *Xilinx Design Tools: Installation and Licensing Guide (UG978)*.


Export RTL Dialog

Export RTL

Format Selection

Pcore for EDK | Identification

Options

☑ Evaluate | Verilog

☐ Do not show this dialog box again.

OK Cancel

Figure 2-7: **Export RTL Dialog Box**

The generated Pcore is located in the /`impl` directory of the selected solution, as shown in Figure 3-8.

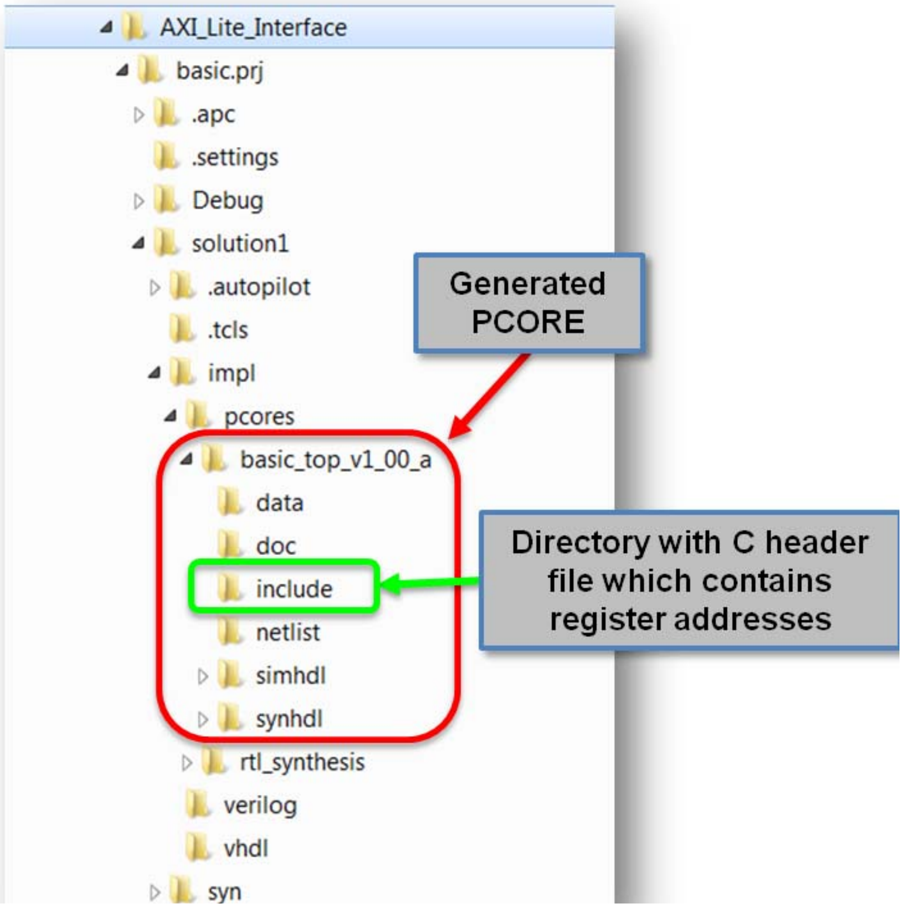


Figure 2-8: **Generated Pcore Location**

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 3-9.

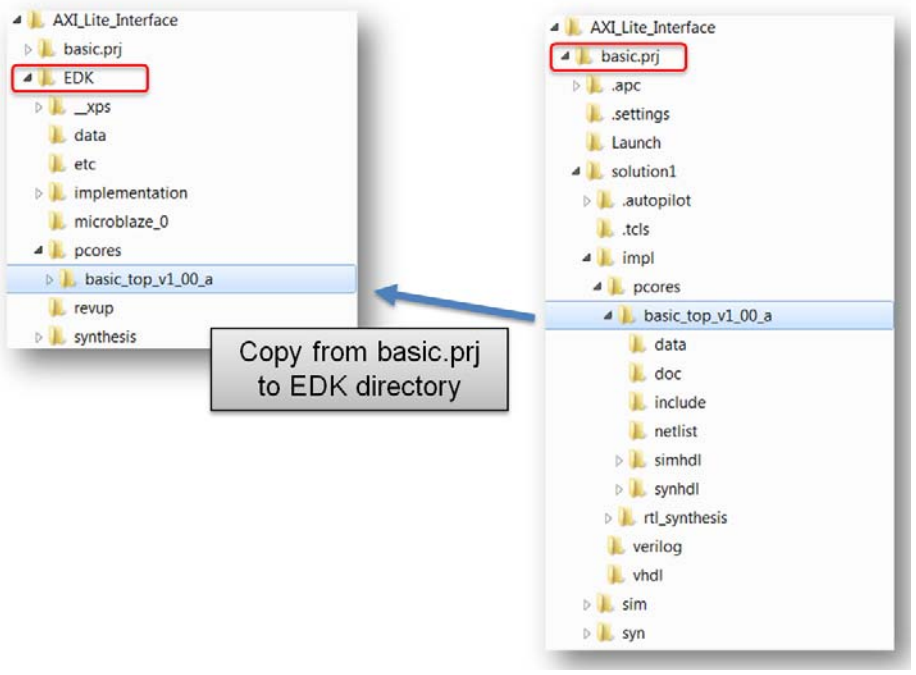


Figure 2-9: **XPS and Vivado HLS Directory Structures**

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 3-10, page 68.
3. When the connection dialog box opens, accept the option to connect to instance `microblaze_0`.

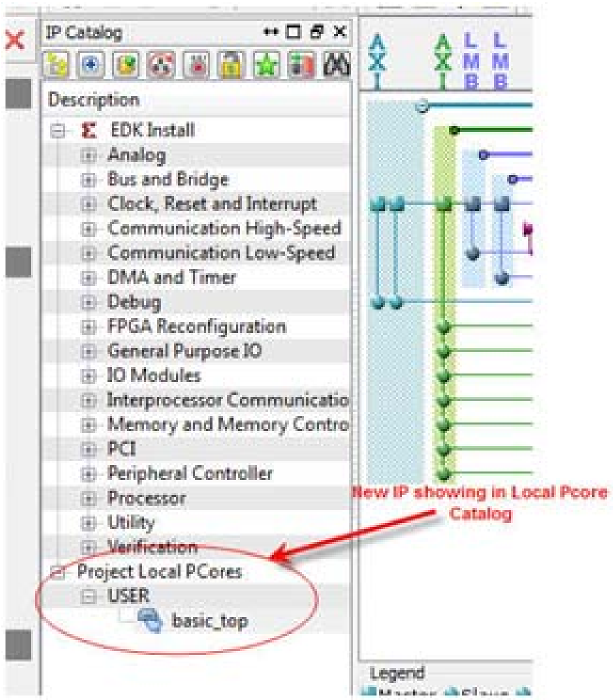


Figure 2-10: **Add Generated Pcore**

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 3-11.

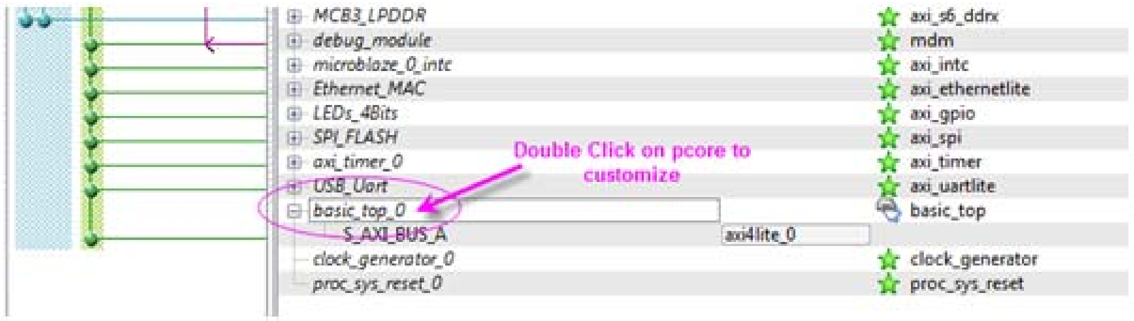
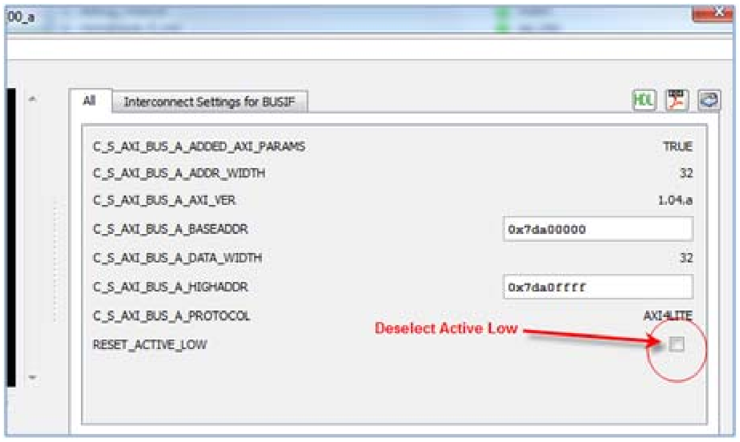


Figure 2-11: **Customize Pcore**

   b. Deselect the RESET_ACTIVE_LOW signal check box, as shown in Figure 3-12.



C_S_AXI_BUS_A_ADDED_AXI_PARAMS	TRUE
C_S_AXI_BUS_A_ADDR_WIDTH	32
C_S_AXI_BUS_A_AXI_VER	1.04.a
C_S_AXI_BUS_A_BASEADDR	0x7da00000
C_S_AXI_BUS_A_DATA_WIDTH	32
C_S_AXI_BUS_A_HIGHADDR	0x7da0ffff
C_S_AXI_BUS_A_PROTOCOL	AXI4LITE
RESET_ACTIVE_LOW	☐

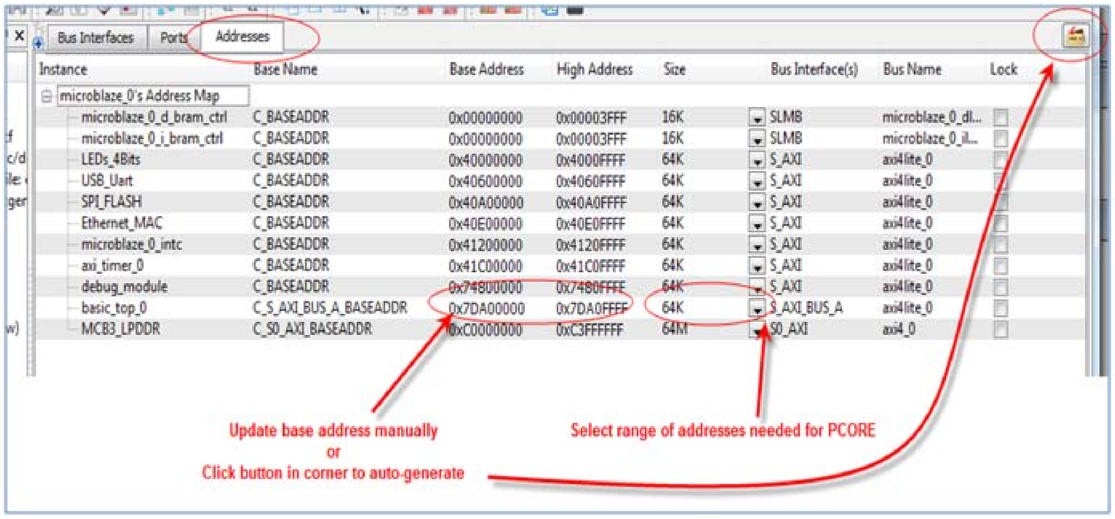
Figure 2-12: **Active Low Signal Check Box**

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 3-12.

   The other location is the Addresses tab in the Assembly window, shown in Figure 3-13. This option allows you to view the full memory map for the MicroBlaze processor, which prevents memory overlap errors.

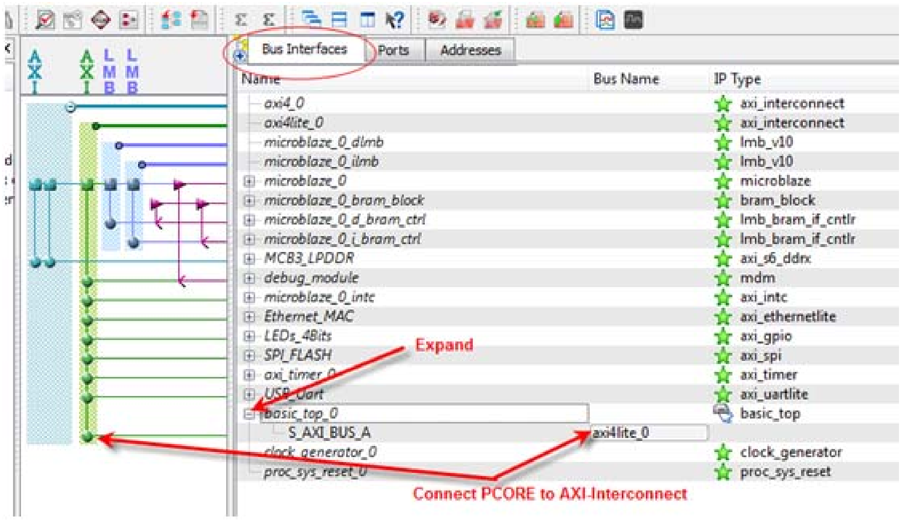
   Note: You can also automatically generate addresses in the Assembly window.


Bus Interfaces
Ports
Addresses
Update base address manually
or
Click button in corner to auto-generate
Select range of addresses needed for PCORE

Instance	Base Name	Base Address	High Address	Size	Bus Interface(s)	Bus Name	Lock
microblaze_0's Address Map							
microblaze_0_d_bram_ctrl	C_BASEADDR	0x00000000	0x00003FFF	16K	SLMB	microblaze_0_dl...	☐
microblaze_0_i_bram_ctrl	C_BASEADDR	0x00000000	0x00003FFF	16K	SLMB	microblaze_0_il...	☐
LEDs_4Bits	C_BASEADDR	0x40000000	0x4000FFFF	64K	S_AXI	axi4lite_0	☐
USB_Uart	C_BASEADDR	0x40600000	0x4060FFFF	64K	S_AXI	axi4lite_0	☐
SPI_FLASH	C_BASEADDR	0x40A00000	0x40A0FFFF	64K	S_AXI	axi4lite_0	☐
Ethernet_MAC	C_BASEADDR	0x40E00000	0x40E0FFFF	64K	S_AXI	axi4lite_0	☐
microblaze_0_intc	C_BASEADDR	0x41200000	0x4120FFFF	64K	S_AXI	axi4lite_0	☐
axi_timer_0	C_BASEADDR	0x41C00000	0x41C0FFFF	64K	S_AXI	axi4lite_0	☐
debug_module	C_BASEADDR	0x74800000	0x7480FFFF	64K	S_AXI	axi4lite_0	☐
basic_top_0	C_S_AXI_BUS_A_BASEADDR	0x7DA00000	0x7DA0FFFF	64K	S_AXI_BUS_A	axi4lite_0	☐
MCB3_LPDDR	C_S0_AXI_BASEADDR	0xC0000000	0xC3FFFFFF	64M	S0_AXI	axi4_0	☐

Figure 2-13: **Set Pcore Base Address**

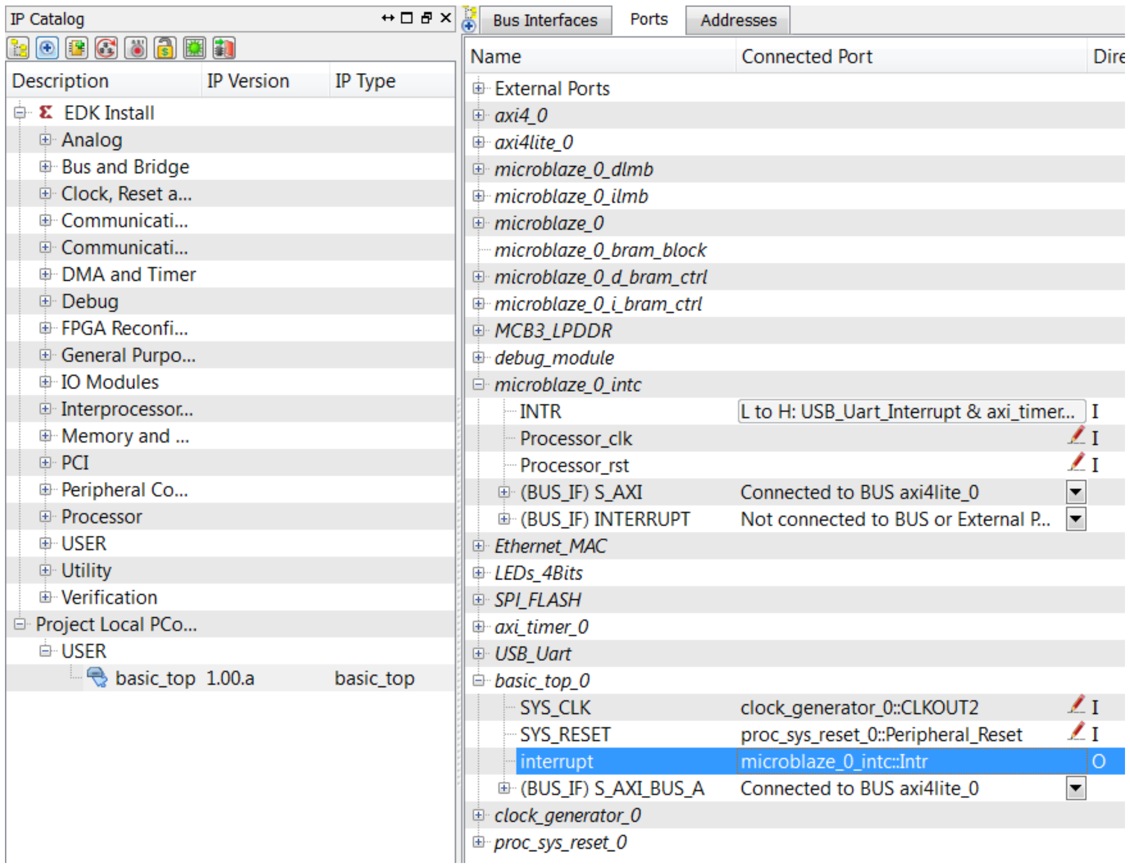
7. Connect the Pcore to AXI Interconnect in the Bus Interfaces tab, as shown in Figure 3-14.



Name	Bus Name	IP Type
axi4_0		axi_interconnect
axi4lite_0		axi_interconnect
microblaze_0_dlmb		lmb_v10
microblaze_0_ilmb		lmb_v10
microblaze_0		microblaze
microblaze_0_bram_block		bram_block
microblaze_0_d_bram_ctrl		lmb_bram_if_cntlr
microblaze_0_i_bram_ctrl		lmb_bram_if_cntlr
MCB3_LPDDR		axi_s6_ddrx
debug_module		mdm
microblaze_0_intc		axi_intc
Ethernet_MAC		axi_ethernetlite
LEDs_4Bits		axi_gpio
SPI_FLASH		axi_spi
axi_timer_0		axi_timer
USB_Uart		axi_uartlite
basic_top_0		basic_top
S_AXI_BUS_A	axi4lite_0	
clock_generator_0		clock_generator
proc_sys_reset_0		proc_sys_reset

Figure 2-14: **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 3-15).



Description	IP Version	IP Type
EDK Install		
Analog		
Bus and Bridge		
Clock, Reset a...		
Communicati...		
Communicati...		
DMA and Timer		
Debug		
FPGA Reconfi...		
General Purpo...		
IO Modules		
Interprocessor...		
Memory and ...		
PCI		
Peripheral Co...		
Processor		
USER		
Utility		
Verification		
Project Local PCo...		
USER		
basic_top	1.00.a	basic_top

Name	Connected Port	Dire
External Ports		
axi4_0		
axi4lite_0		
microblaze_0_dlmb		
microblaze_0_ilmb		
microblaze_0		
microblaze_0_bram_block		
microblaze_0_d_bram_ctrl		
microblaze_0_i_bram_ctrl		
MCB3_LPDDR		
debug_module		
microblaze_0_intc		
INTR	L to H: USB_Uart_Interrupt & axi_timer...	I
Processor_clk		I
Processor_rst		I
(BUS_IF) S_AXI	Connected to BUS axi4lite_0	
(BUS_IF) INTERRUPT	Not connected to BUS or External P...	
Ethernet_MAC		
LEDs_4Bits		
SPI_FLASH		
axi_timer_0		
USB_Uart		
basic_top_0		
SYS_CLK	clock_generator_0::CLKOUT2	I
SYS_RESET	proc_sys_reset_0::Peripheral_Reset	I
interrupt	microblaze_0_intc::Intr	O
(BUS_IF) S_AXI_BUS_A	Connected to BUS axi4lite_0	
clock_generator_0		
proc_sys_reset_0		

Figure 2-15: **Connect Clocks and Rest Signals**

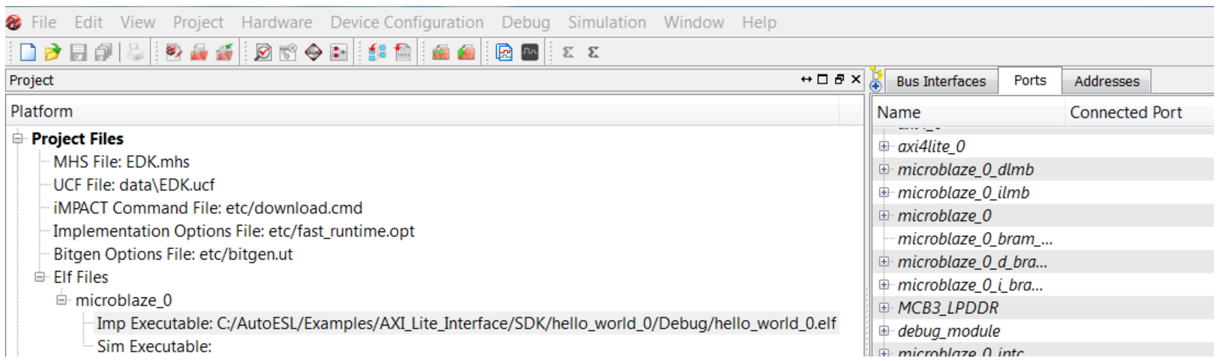
10. As shown in Figure 3-15, on instance, basic_top_0:
    a. Connect port `SYS_CLK` to `clock_generator_0: CLKOUT2`.
    b. Connect port `SYS_RST` to `proc_sys_reset_0: Peripheral_Reset`.
    c. Add port interrupt `basic_top_0` to the list of connected interrupts.
    d. Confirm that port (BUS_IF) is 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 3-16 shows how to select the ELF file to initialize the BRAM.



Project Files
MHS File: EDK.mhs
UCF File: data\EDK.ucf
iMPACT Command File: etc/download.cmd
Implementation Options File: etc/fast_runtime.opt
Bitgen Options File: etc/bitgen.ut
Elf Files
microblaze_0
Imp Executable: C:/AutoESL/Examples/AXI_Lite_Interface/SDK/hello_world_0/Debug/hello_world_0.elf
Sim Executable:

axi4lite_0
microblaze_0_dlmb
microblaze_0_ilmb
microblaze_0
microblaze_0_bram_...
microblaze_0_d_bra...
microblaze_0_i_bra...
MCB3_LPDDR
debug_module

Figure 2-16: **Select ELF File**

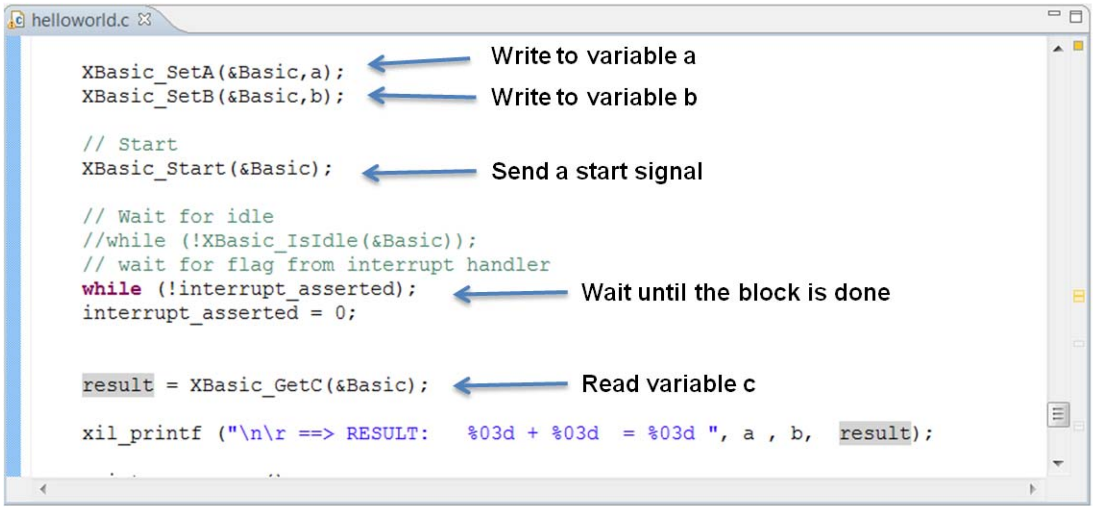
2. Select **Device Configuration > Update Bitstream** to generate the bitstream.

   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 3-17.



```
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);
```

Figure 2-17: **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**.

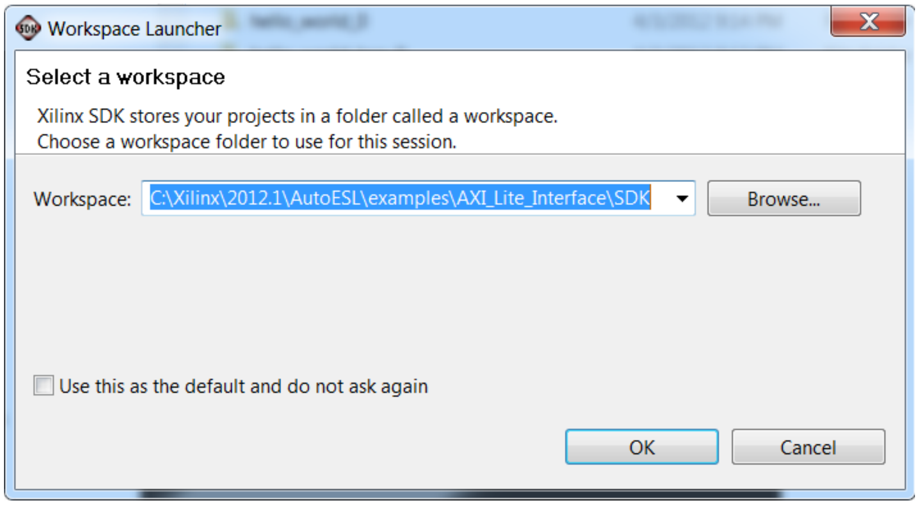
   Exporting the hardware description of the system from XPS to SDK enables it to create software application images for that system.

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 3-18.

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


Workspace Launcher

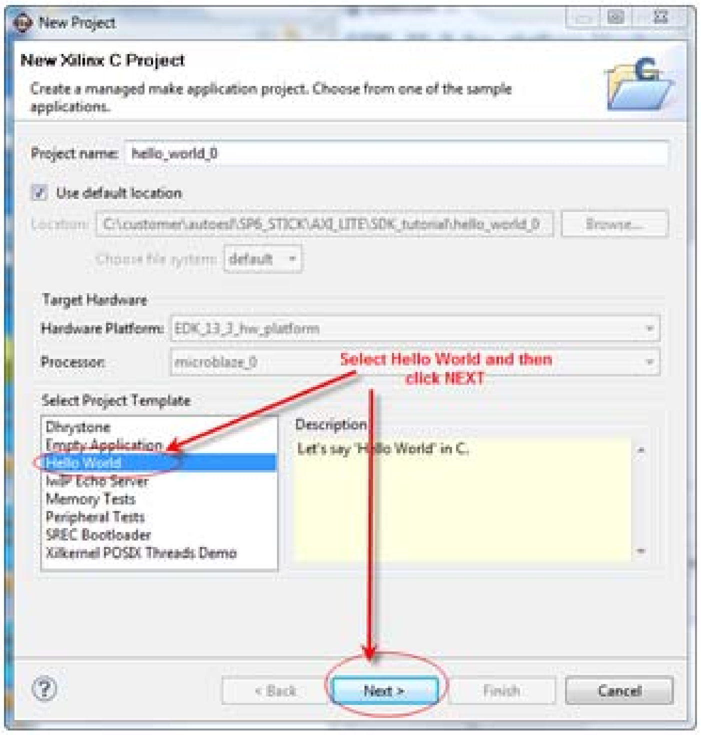
Select a workspace

Xilinx SDK stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

Workspace: C:\Xilinx\2012.1\AutoESL\examples\AXI_Lite_Interface\SDK

Figure 2-18: **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 3-19.
6. Click **Next**.


New Project

New Xilinx C Project

Create a managed make application project. Choose from one of the sample applications.

Project name: hello_world_0

Use default location [✓]

Location: C:\customer\autoesl\SP6_STICK\AXI_LITE\SDK_tutorial\hello_world_0

Choose file system: default

Target Hardware

Hardware Platform: EDK_13_3_hw_platform

Processor: microblaze_0

Select Project Template

Dhrystone
Empty Application
[✓ Hello World]
lwIP Echo Server
Memory Tests
Peripheral Tests
SREC Bootloader
Xilkernel POSIX Threads Demo

Description

Let's say 'Hello World' in C.

Figure 2-19: **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 3-20), with the application name and the `.elf` extension. In this example, the file is `hello_world_0.elf`.

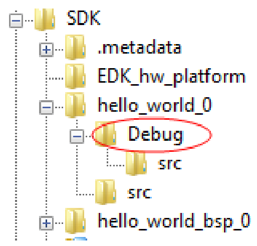


Figure 2-20: **ELF File Location**

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 3-21.

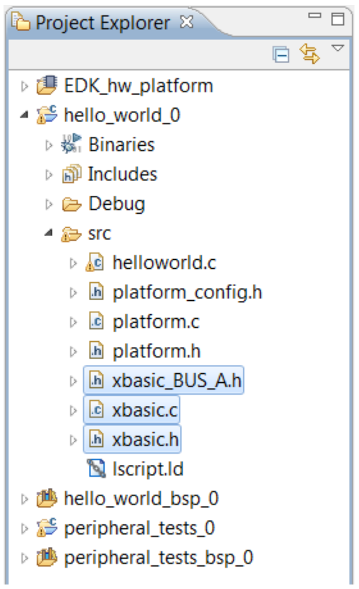


Figure 2-21: **C Files in hello_world_0/src Directory**

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 3-22

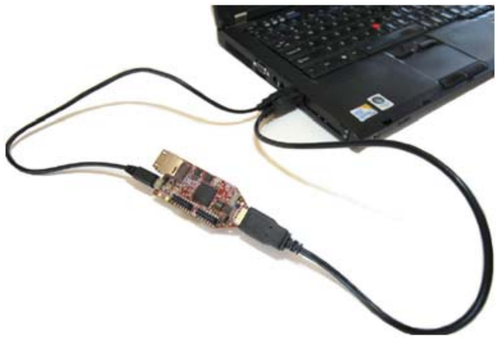
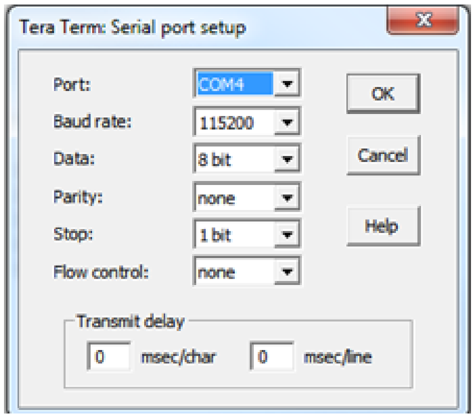


Figure 2-22: **Cable Connections**

- Hyperterminal (Tera Term) with the serial port setup shown in Figure 3-23



Tera Term: Serial port setup

Port:	COM4
Baud rate:	115200
Data:	8 bit
Parity:	none
Stop:	1 bit
Flow control:	none

Transmit delay

0 msec/char 0 msec/line

Figure 2-23: **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 3-24.

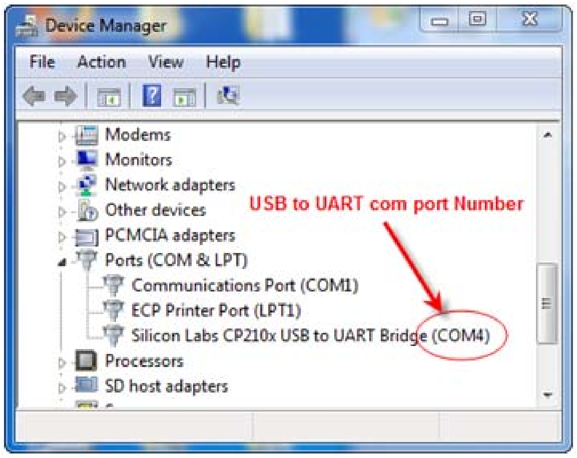
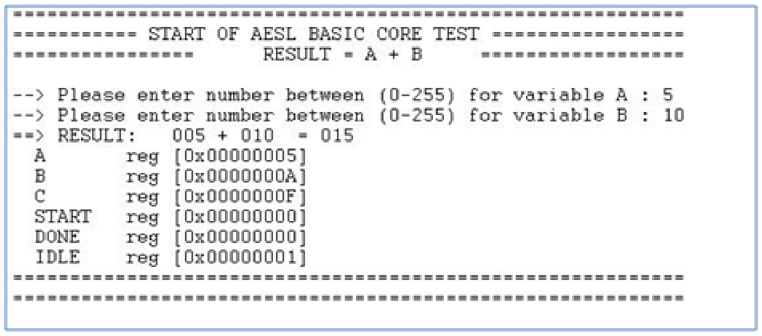


Figure 2-24: **USB to UART COM Port Number**

Running the Demo

1. Open hyperterminal (Tera Term) with the settings shown in Figure 3-23.
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 3-25 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 2-25: **Application Output**

Running Bus Functional Model Simulation

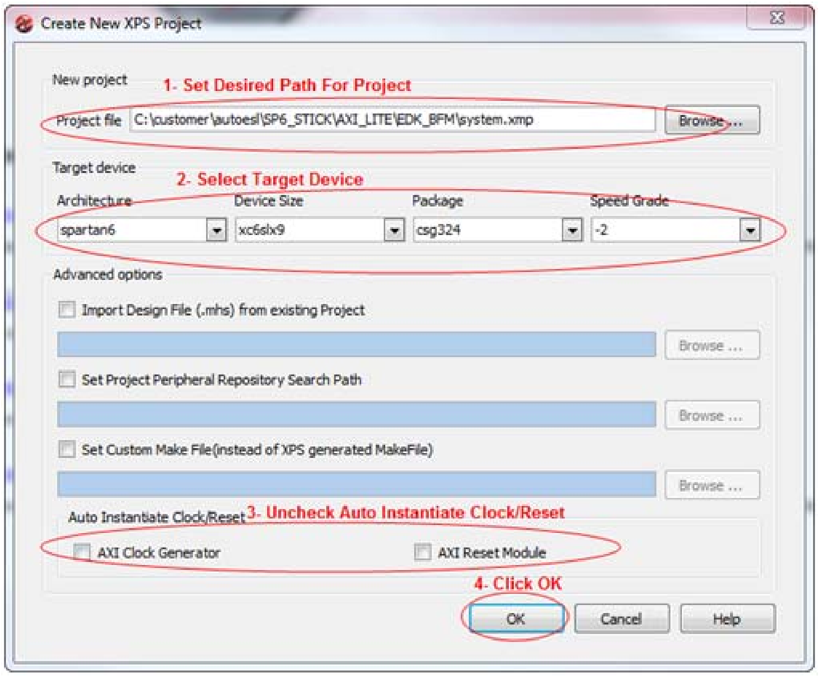
Runing bus functional model simulation on a generated Pcore requires the following steps:

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.



Create New XPS Project

New project

1- Set Desired Path For Project

Project file: C:\customer\autoesl\SP6_STICK\AXI_LITE\EDK_BFM\system.xmp

Target device

2- Select Target Device

Architecture	Device Size	Package	Speed Grade
spartan6	xc6slx9	csg324	-2

Advanced options

☐ Import Design File (.mhs) from existing Project

☐ Set Project Peripheral Repository Search Path

☐ Set Custom Make File(instead of XPS generated MakeFile)

Auto Instantiate Clock/Reset

3- Uncheck Auto Instantiate Clock/Reset

☐ AXI Clock Generator ☐ AXI Reset Module

4- Click OK

Figure 2-26: **Create New XPS Project Dialog Box**

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

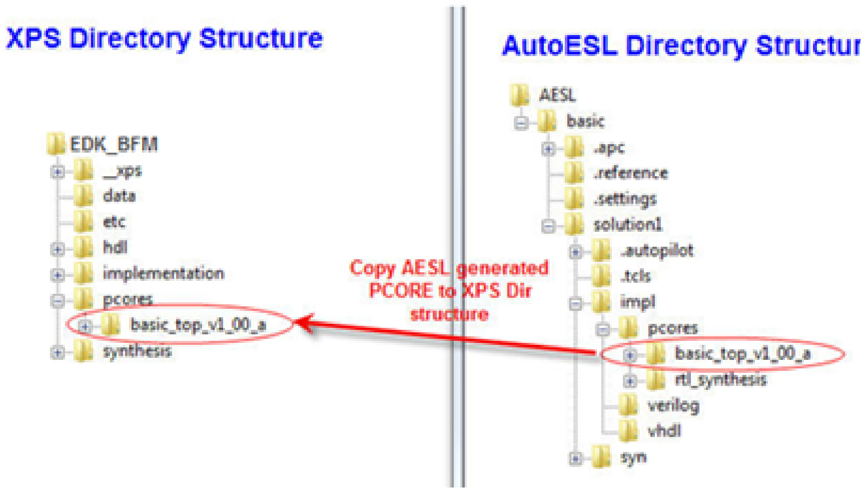
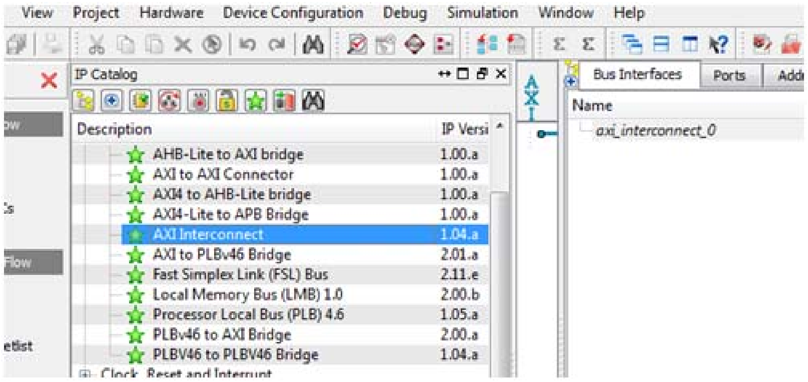


Figure 2-27: **Vivado HLS Directory Structure**

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.



Description	IP Versi
AHB-Lite to AXI bridge	1.00.a
AXI to AXI Connector	1.00.a
AXI4 to AHB-Lite bridge	1.00.a
AXI4-Lite to APB Bridge	1.00.a
AXI Interconnect	1.04.a
AXI to PLBv46 Bridge	2.01.a
Fast Simplex Link (FSL) Bus	2.11.e
Local Memory Bus (LMB) 1.0	2.00.b
Processor Local Bus (PLB) 4.6	1.05.a
PLBv46 to AXI Bridge	2.00.a
PLBV46 to PLBV46 Bridge	1.04.a
Clock, Reset and Interrupt	

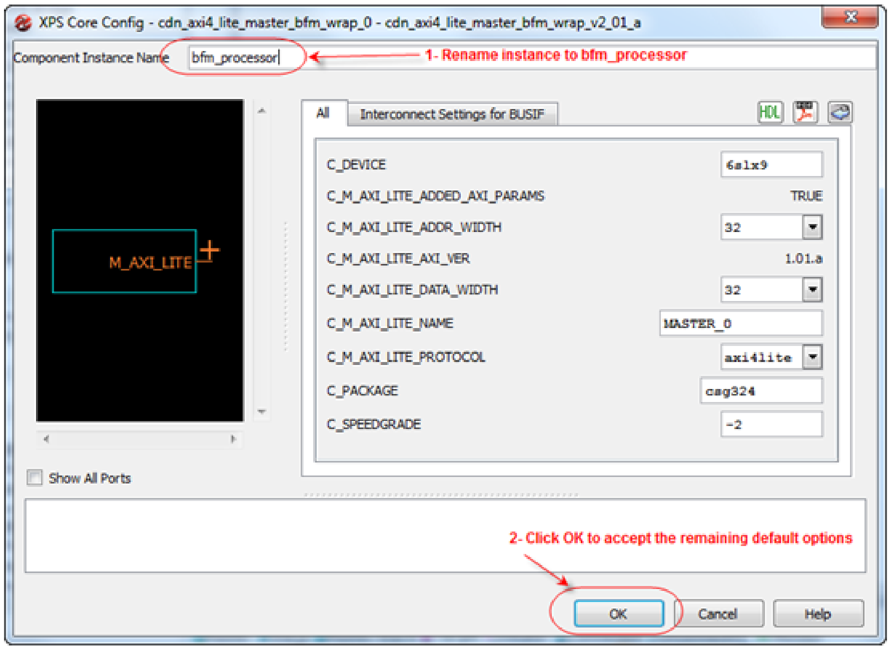
Figure 2-28: **Pcore Added to IP Catalog**

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**.

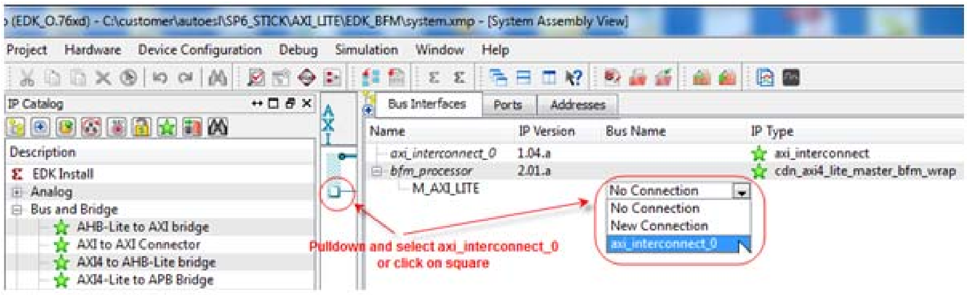

XPS Core Config - cdn_axi4_lite_master_bfm_wrap_0 - cdn_axi4_lite_master_bfm_wrap_v2_01_a
1- Rename instance to bfm_processor
All
Interconnect Settings for BUSIF
M_AXI_LITE
Show All Ports
2- Click OK to accept the remaining default options
OK
Cancel
Help

Component Instance Name: bfm_processor

C_DEVICE	6slx9
C_M_AXI_LITE_ADDED_AXI_PARAMS	TRUE
C_M_AXI_LITE_ADDR_WIDTH	32
C_M_AXI_LITE_AXI_VER	1.01.a
C_M_AXI_LITE_DATA_WIDTH	32
C_M_AXI_LITE_NAME	MASTER_0
C_M_AXI_LITE_PROTOCOL	axi4lite
C_PACKAGE	csg324
C_SPEEDGRADE	-2

Figure 2-29: **XPS Core Config**

14. Connect `bfm_processor` to the AXI interconnect, as shown in Figure 3-30.



Name	IP Version	Bus Name	IP Type
axi_interconnect_0	1.04.a		axi_interconnect
bfm_processor	2.01.a		cdn_axi4_lite_master_bfm_wrap
M_AXI_LITE		No Connection	

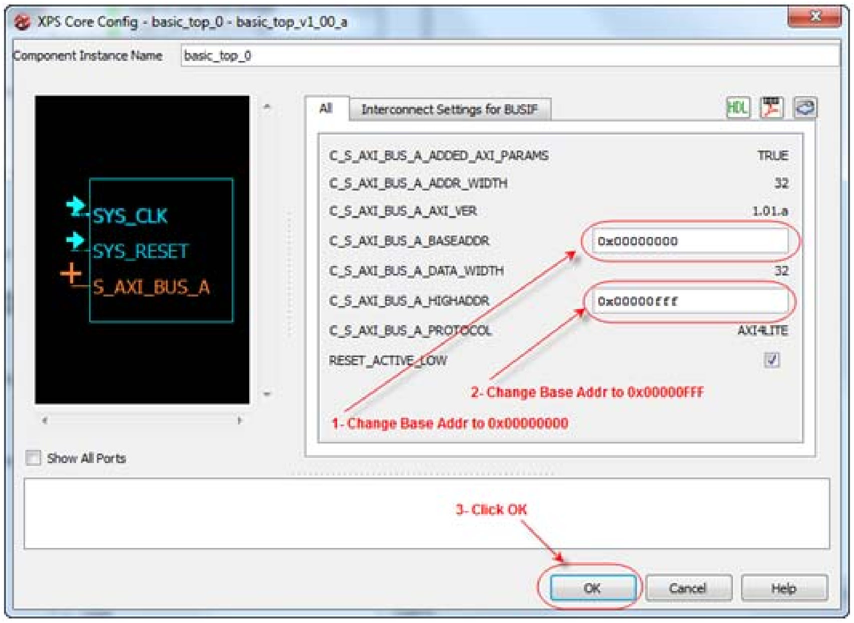
Figure 2-30: **bfm_processor Connection**

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**.

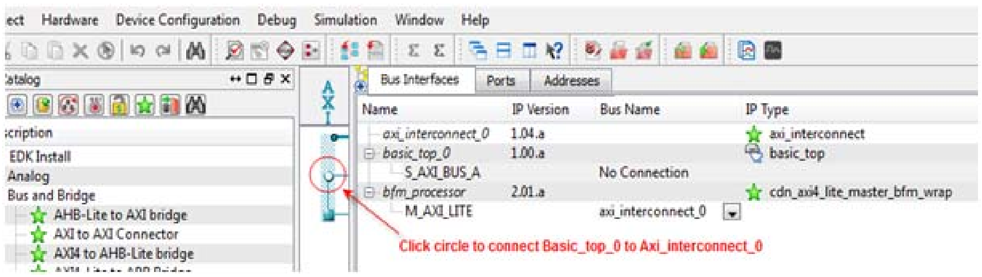

XPS Core Config - basic_top_0 - basic_top_v1_00_a
SYS_CLK
SYS_RESET
S_AXI_BUS_A
Show All Ports
All
Interconnect Settings for BUSIF
2- Change Base Addr to 0x00000FFF
1- Change Base Addr to 0x00000000
3- Click OK
OK
Cancel
Help

Component Instance Name: basic_top_0

C_S_AXI_BUS_A_ADDED_AXI_PARAMS	TRUE
C_S_AXI_BUS_A_ADDR_WIDTH	32
C_S_AXI_BUS_A_AXI_VER	1.01.a
C_S_AXI_BUS_A_BASEADDR	0x00000000
C_S_AXI_BUS_A_DATA_WIDTH	32
C_S_AXI_BUS_A_HIGHADDR	0x00000fff
C_S_AXI_BUS_A_PROTOCOL	AXI4LITE
RESET_ACTIVE_LOW	✓

Figure 2-31: **XPS Core Config Dialog Box**

19. Connect `basic_top_0` to the AXI interconnect as shown in Figure 3-32.



Name	IP Version	Bus Name	IP Type
axi_interconnect_0	1.04.a		axi_interconnect
basic_top_0	1.00.a		basic_top
S_AXI_BUS_A		No Connection	
bfm_processor	2.01.a		cdn_axi4_lite_master_bfm_wrap
M_AXI_LITE		axi_interconnect_0	

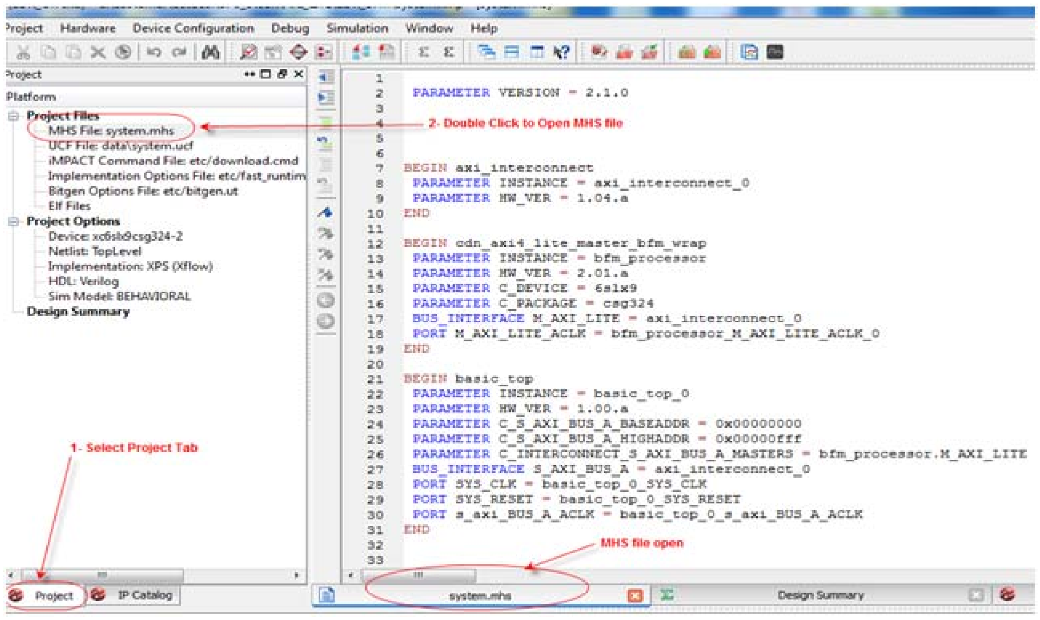
Figure 2-32: **AXI Interconnect**

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.



```

 PARAMETER VERSION = 2.1.0



BEGIN axi_interconnect
 PARAMETER INSTANCE = axi_interconnect_0
 PARAMETER HW_VER = 1.04.a
END

BEGIN cdn_axi4_lite_master_bfm_wrap
 PARAMETER INSTANCE = bfm_processor
 PARAMETER HW_VER = 2.01.a
 PARAMETER C_DEVICE = 6slx9
 PARAMETER C_PACKAGE = csg324
 BUS_INTERFACE M_AXI_LITE = axi_interconnect_0
 PORT M_AXI_LITE_ACLK = bfm_processor_M_AXI_LITE_ACLK_0
END

BEGIN basic_top
 PARAMETER INSTANCE = basic_top_0
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_S_AXI_BUS_A_BASEADDR = 0x00000000
 PARAMETER C_S_AXI_BUS_A_HIGHADDR = 0x00000fff
 PARAMETER C_INTERCONNECT_S_AXI_BUS_A_MASTERS = bfm_processor.M_AXI_LITE
 BUS_INTERFACE S_AXI_BUS_A = axi_interconnect_0
 PORT SYS_CLK = basic_top_0_SYS_CLK
 PORT SYS_RESET = basic_top_0_SYS_RESET
 PORT s_axi_BUS_A_ACLK = basic_top_0_s_axi_BUS_A_ACLK
END


```

Project
Platform
Project Files
MHS File: system.mhs
UCF File: data\system.ucf
iMPACT Command File: etc/download.cmd
Implementation Options File: etc/fast_runtim
Bitgen Options File: etc/bitgen.ut
Elf Files
Project Options
Device: xc6slx9csg324-2
Netlist: TopLevel
Implementation: XPS (Xflow)
HDL: Verilog
Sim Model: BEHAVIORAL
Design Summary

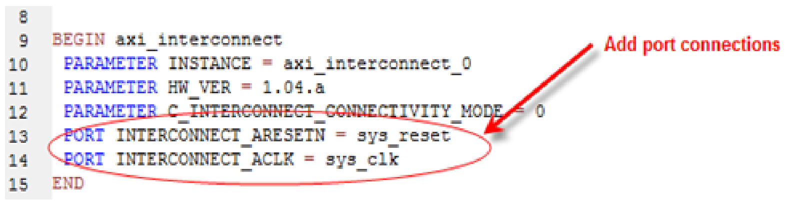
Figure 2-33: **Opening the MHS File**

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 2-34: **External Port Declarations in MHS File**

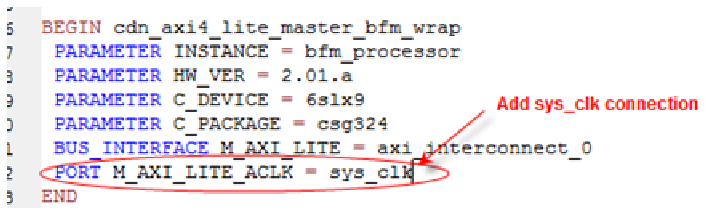
3. Connect `sys_clk` and `sys_reset` to `axi_interconnect` in the MHS file.



```
BEGIN axi_interconnect
 PARAMETER INSTANCE = axi_interconnect_0
 PARAMETER HW_VER = 1.04.a
 PARAMETER C_INTERCONNECT_CONNECTIVITY_MODE = 0
 PORT INTERCONNECT_ARESETN = sys_reset
 PORT INTERCONNECT_ACLK = sys_clk
END
```

Figure 2-35: **Port Connections in MHS File**

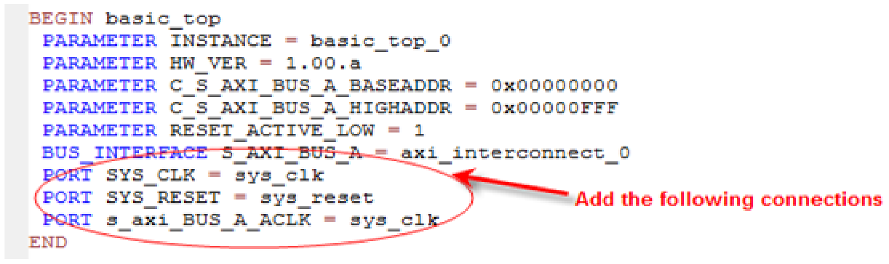
4. Connect `sys_clk` to `bfm_processor` Pcore in MHS file.



```
BEGIN cdn_axi4_lite_master_bfm_wrap
 PARAMETER INSTANCE = bfm_processor
 PARAMETER HW_VER = 2.01.a
 PARAMETER C_DEVICE = 6slx9
 PARAMETER C_PACKAGE = csg324
 BUS_INTERFACE M_AXI_LITE = axi_interconnect_0
 PORT M_AXI_LITE_ACLK = sys_clk
END
```

Figure 2-36: **sys_clk Connection in MHS File**

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



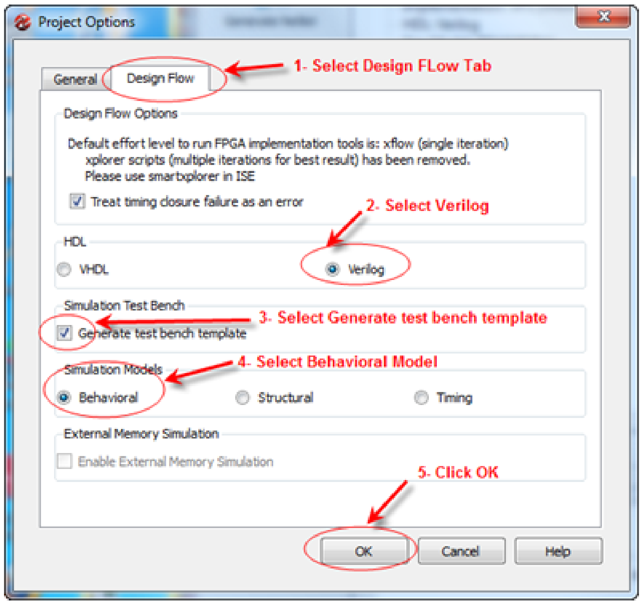
```
BEGIN basic_top
 PARAMETER INSTANCE = basic_top_0
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_S_AXI_BUS_A_BASEADDR = 0x00000000
 PARAMETER C_S_AXI_BUS_A_HIGHADDR = 0x00000FFF
 PARAMETER RESET_ACTIVE_LOW = 1
 BUS_INTERFACE S_AXI_BUS_A = axi_interconnect_0
 PORT SYS_CLK = sys_clk
 PORT SYS_RESET = sys_reset
 PORT s_axi_BUS_A_ACLK = sys_clk
END
```

Figure 2-37: **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.


Project Options
General
Design Flow
1- Select Design FLow Tab
2- Select Verilog
3- Select Generate test bench template
4- Select Behavioral Model
5- Click OK
OK
Cancel
Help

Design Flow Options

Default effort level to run FPGA implementation tools is: xflow (single iteration)
xplorer scripts (multiple iterations for best result) has been removed.
Please use smartxplorer in ISE

[✓] Treat timing closure failure as an error

HDL

VHDL [✓ Verilog]

Simulation Test Bench

[✓] Generate test bench template

Simulation Models

[✓ Behavioral] Structural Timing

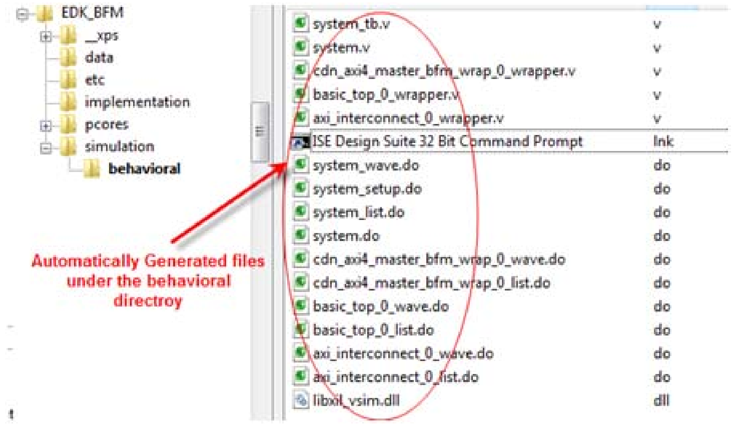
External Memory Simulation

[ ] Enable External Memory Simulation

Figure 2-38: **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.


EDK_BFM
_xps
data
etc
implementation
pcores
simulation
behavioral
Automatically Generated files under the behavioral directroy

system_tb.v	v
system.v	v
cdn_axi4_master_bfm_wrap_0_wrapper.v	v
basic_top_0_wrapper.v	v
axi_interconnect_0_wrapper.v	v
ISE Design Suite 32 Bit Command Prompt	lnk
system_wave.do	do
system_setup.do	do
system_list.do	do
system.do	do
cdn_axi4_master_bfm_wrap_0_wave.do	do
cdn_axi4_master_bfm_wrap_0_list.do	do
basic_top_0_wave.do	do
basic_top_0_list.do	do
axi_interconnect_0_wave.do	do
axi_interconnect_0_list.do	do
libxil_vsim.dll	dll

Figure 2-39: **Simulation Directory Structure**

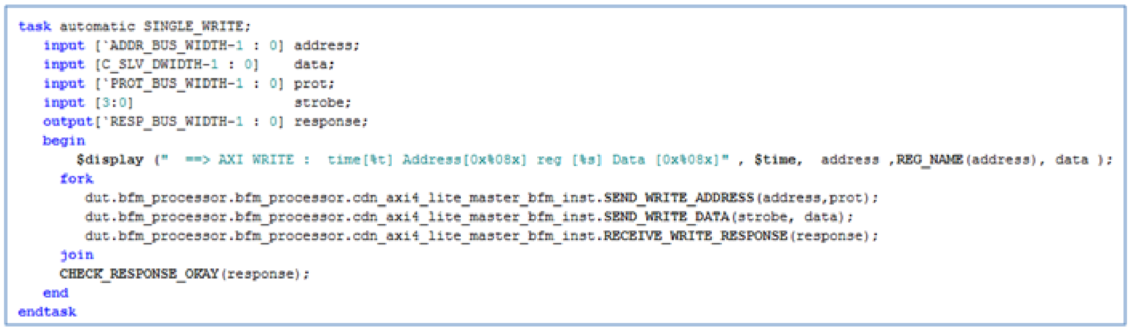
4. Edit the `system_tb.v` file and add code to read/write the Pcore generated by Vivado HLS.

   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)*.

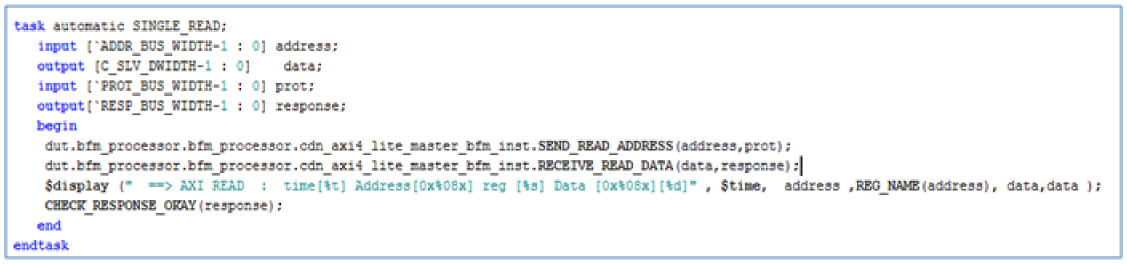
The write task is displayed in Figure 3-40.



```
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 2-40: **Write Task Code**

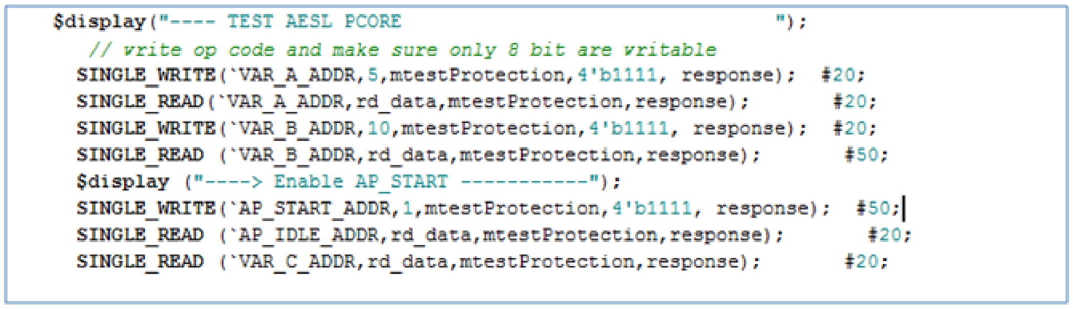
The read task is displayed in Figure 3-41.



```
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 2-41: **Read Task Code**

Testing for the Pcore is written using the above tasks. The example code is displayed in Figure 3-42.



```
    $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 2-42: **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.

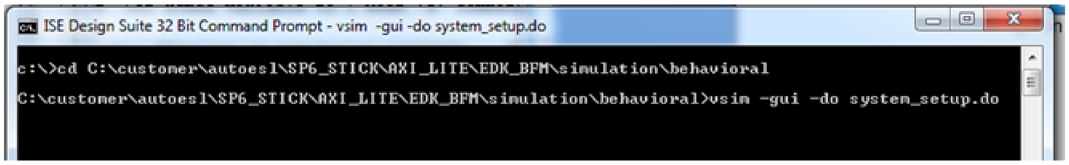
   The recommended 32-bit windows shell is the one provided under the ISE 32 bit command prompt. To open this shell, select **Windows Start > Programs > Xilinx ISE Design Suite 14.1 > Accessories > ISE Design Suite 32 Bit Command Prompt**.

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.


ISE Design Suite 32 Bit Command Prompt - vsim -gui -do system_setup.do

```
c:\>cd C:\customer\autoesl\SP6_STICK\AXI_LITE\EDK_BFM\simulation\behavioral
C:\customer\autoesl\SP6_STICK\AXI_LITE\EDK_BFM\simulation\behavioral>vsim -gui -do system_setup.do
```

Figure 2-43: **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.

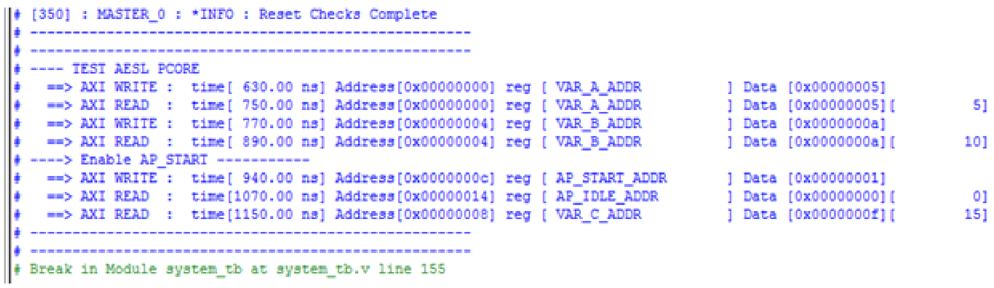
   - **`w`**

     This opens a wave window.

   - **`run -all`**

     This runs the test bench.

Figure 3-44 shows the output results on the transcript window.



```
# [350] : MASTER_0 : *INFO : Reset Checks Complete
# ----------------------------------------------------
# ----------------------------------------------------
# ---- TEST AESL PCORE
#   ==> AXI WRITE :  time[ 630.00 ns] Address[0x00000000] reg [ VAR_A_ADDR         ] Data [0x00000005]
#   ==> AXI READ  :  time[ 750.00 ns] Address[0x00000000] reg [ VAR_A_ADDR         ] Data [0x00000005][         5]
#   ==> AXI WRITE :  time[ 770.00 ns] Address[0x00000004] reg [ VAR_B_ADDR         ] Data [0x0000000a]
#   ==> AXI READ  :  time[ 890.00 ns] Address[0x00000004] reg [ VAR_B_ADDR         ] Data [0x0000000a][        10]
# ----> Enable AP_START -----------
#   ==> AXI WRITE :  time[ 940.00 ns] Address[0x0000000c] reg [ AP_START_ADDR      ] Data [0x00000001]
#   ==> AXI READ  :  time[1070.00 ns] Address[0x00000014] reg [ AP_IDLE_ADDR       ] Data [0x00000000][         0]
#   ==> AXI READ  :  time[1150.00 ns] Address[0x00000008] reg [ VAR_C_ADDR         ] Data [0x0000000f][        15]
# ----------------------------------------------------
# ----------------------------------------------------
# Break in Module system_tb at system_tb.v line 155
```

Figure 2-44: **Transcript Window**

Chapter 3

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 executed 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
```

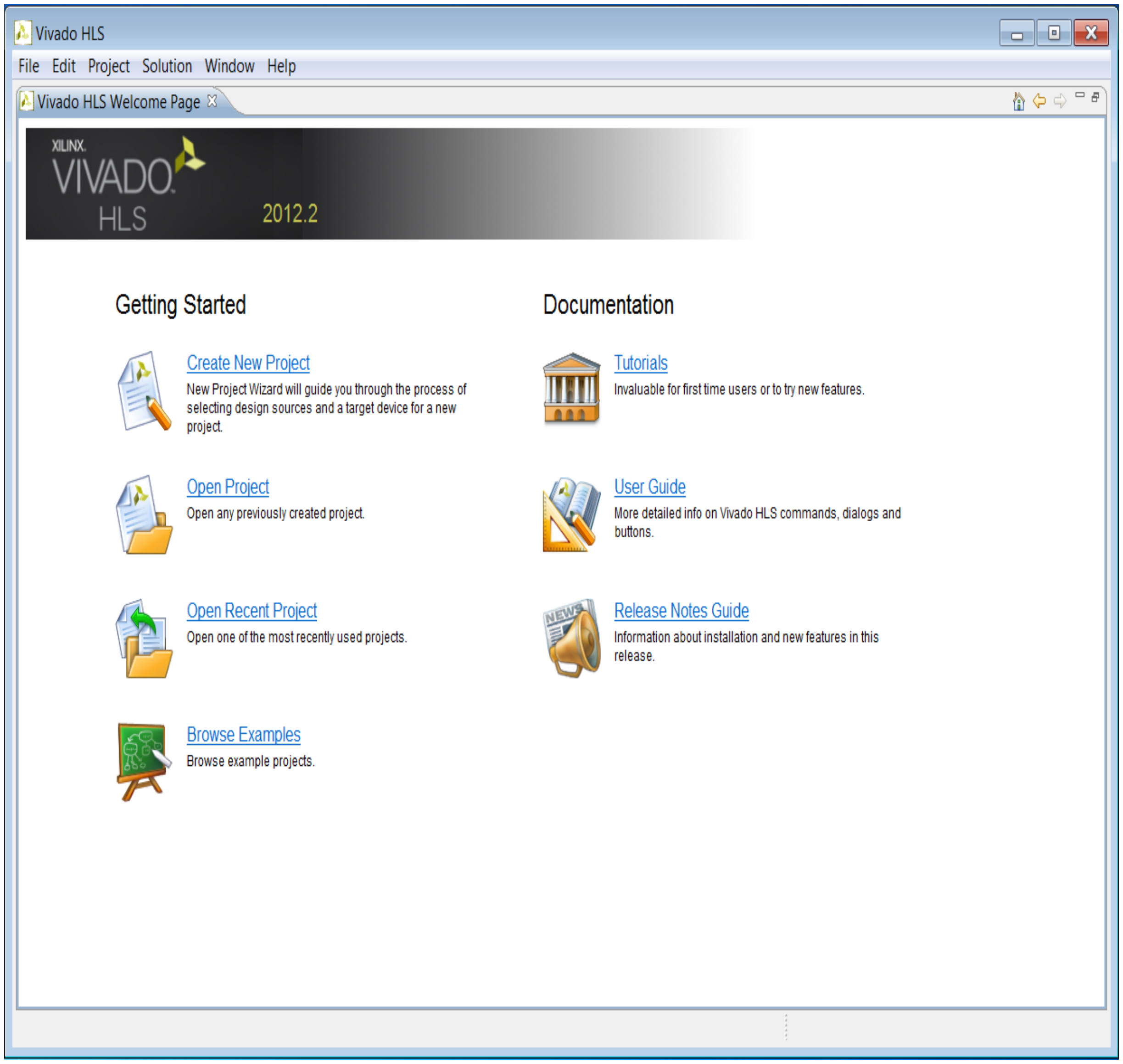
```
        int acc = 0;
        for(int k=0; k < TAPS; k++){
          acc += shift_reg[k] * coeff[k];
        }
        output_val.write(acc);
      }
    }
```

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**.



Getting Started

Create New Project
New Project Wizard will guide you through the process of selecting design sources and a target device for a new project.

Open Project
Open any previously created project.

Open Recent Project
Open one of the most recently used projects.

Browse Examples
Browse example projects.

Documentation

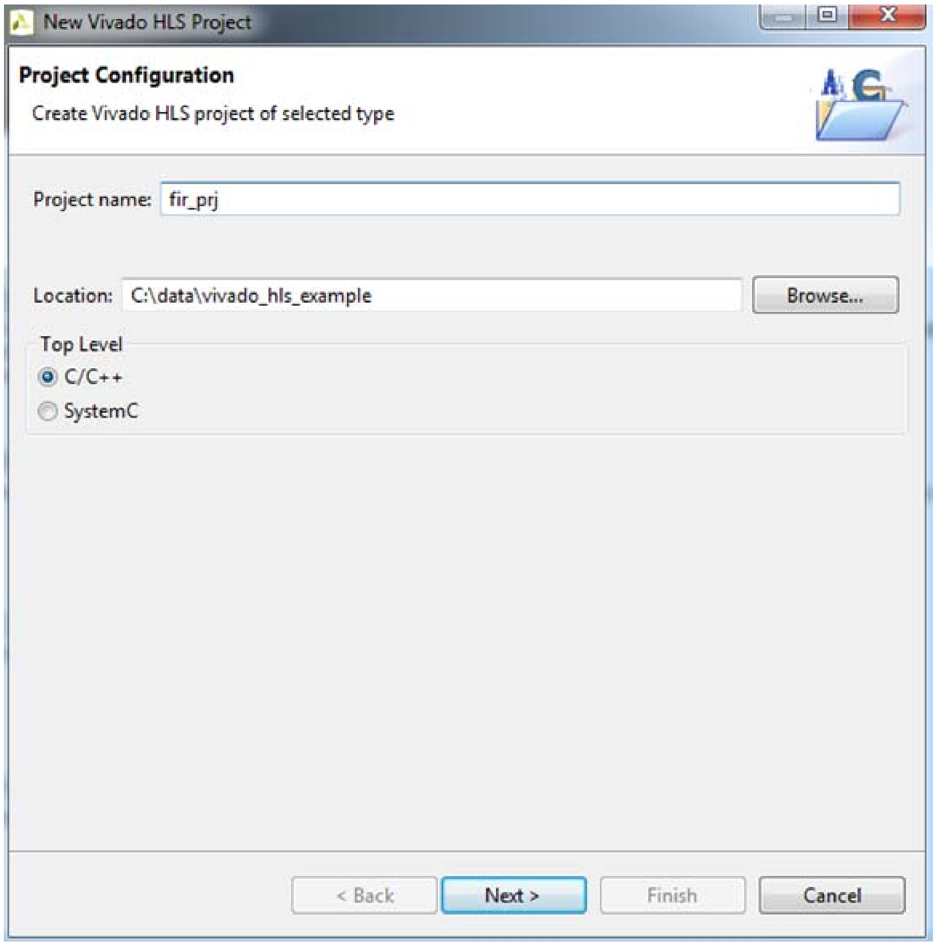
Tutorials
Invaluable for first time users or to try new features.

User Guide
More detailed info on Vivado HLS commands, dialogs and buttons.

Release Notes Guide
Information about installation and new features in this release.

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. Click **Next** to set the location for the project (Figure 3-2)



Project Configuration

Create Vivado HLS project of selected type

Project name: fir_prj

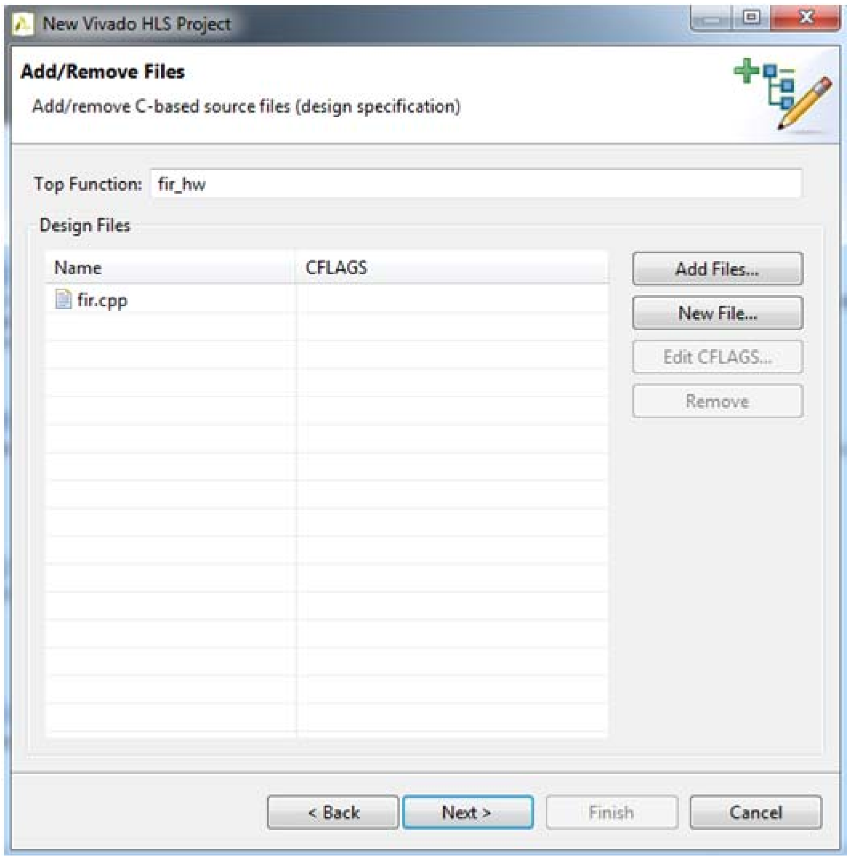
Location: C:\data\vivado_hls_example

Top Level

(● C/C++) ○ SystemC

Figure 3-2: Project Configuration

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



New Vivado HLS Project

Add/Remove Files

Add/remove C-based source files (design specification)

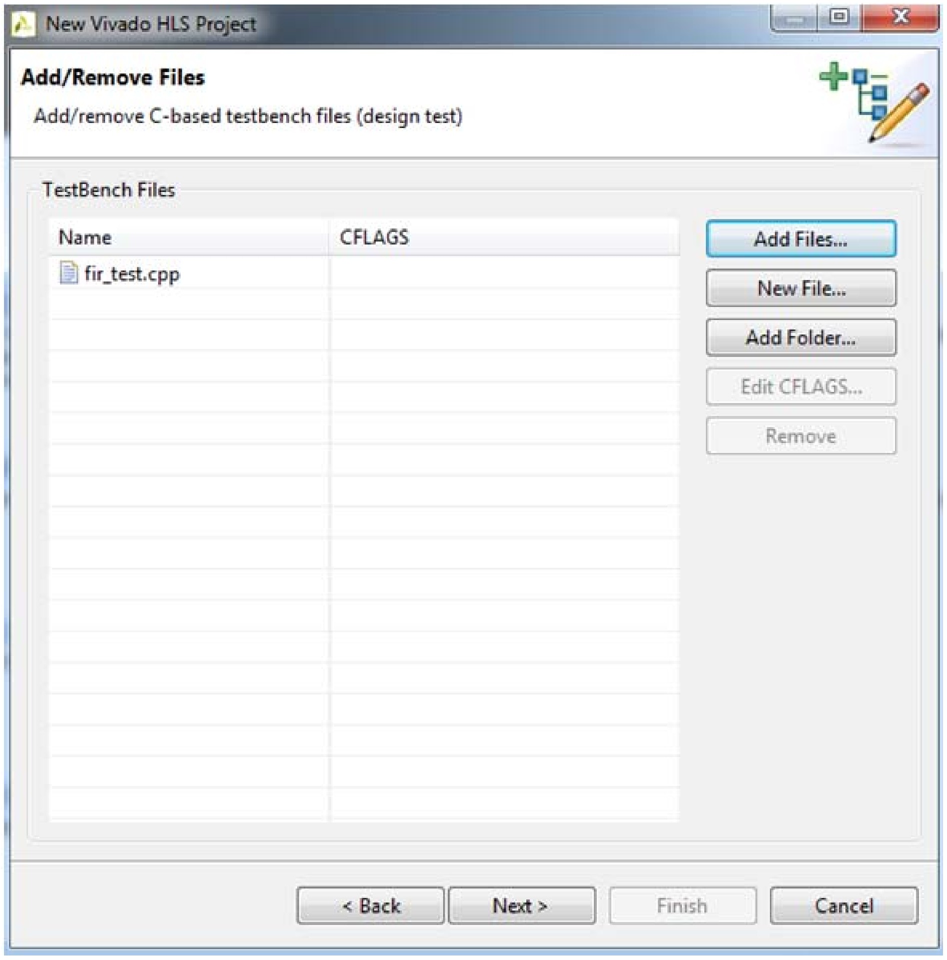
Top Function: fir_hw

Design Files

Name	CFLAGS
fir.cpp	

Figure 3-3: Add Files for Hardware Synthesis

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


New Vivado HLS Project

Add/Remove Files

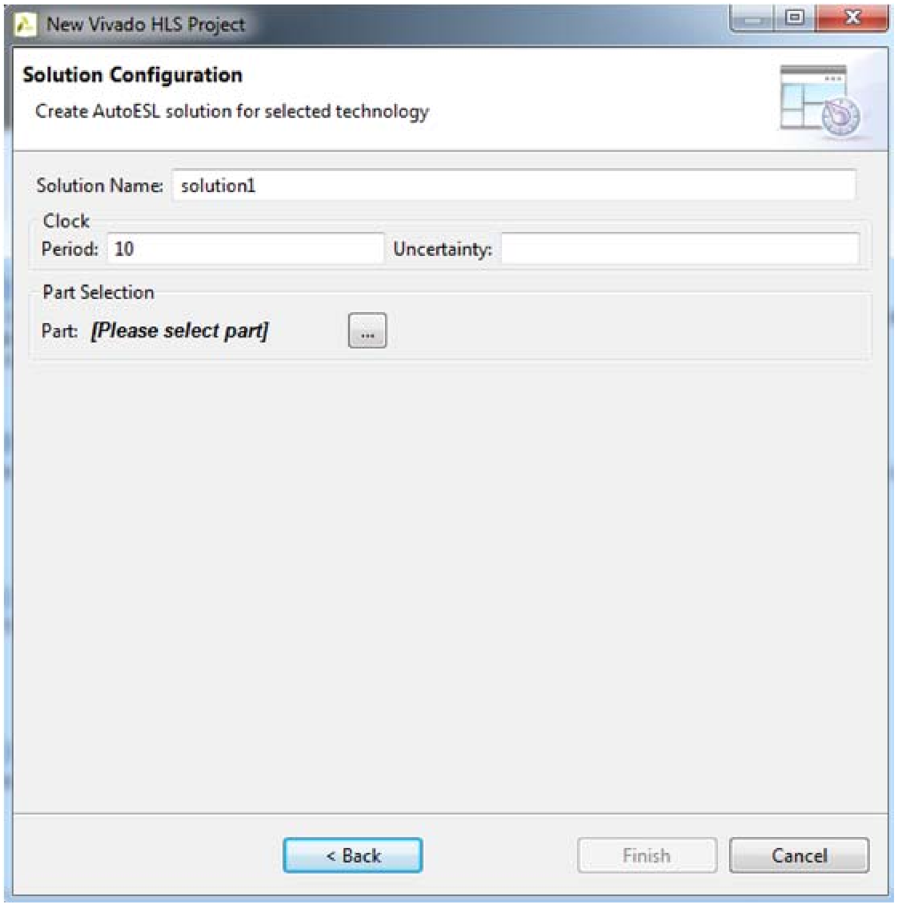
Add/remove C-based testbench files (design test)

TestBench Files

Name	CFLAGS
fir_test.cpp	

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).



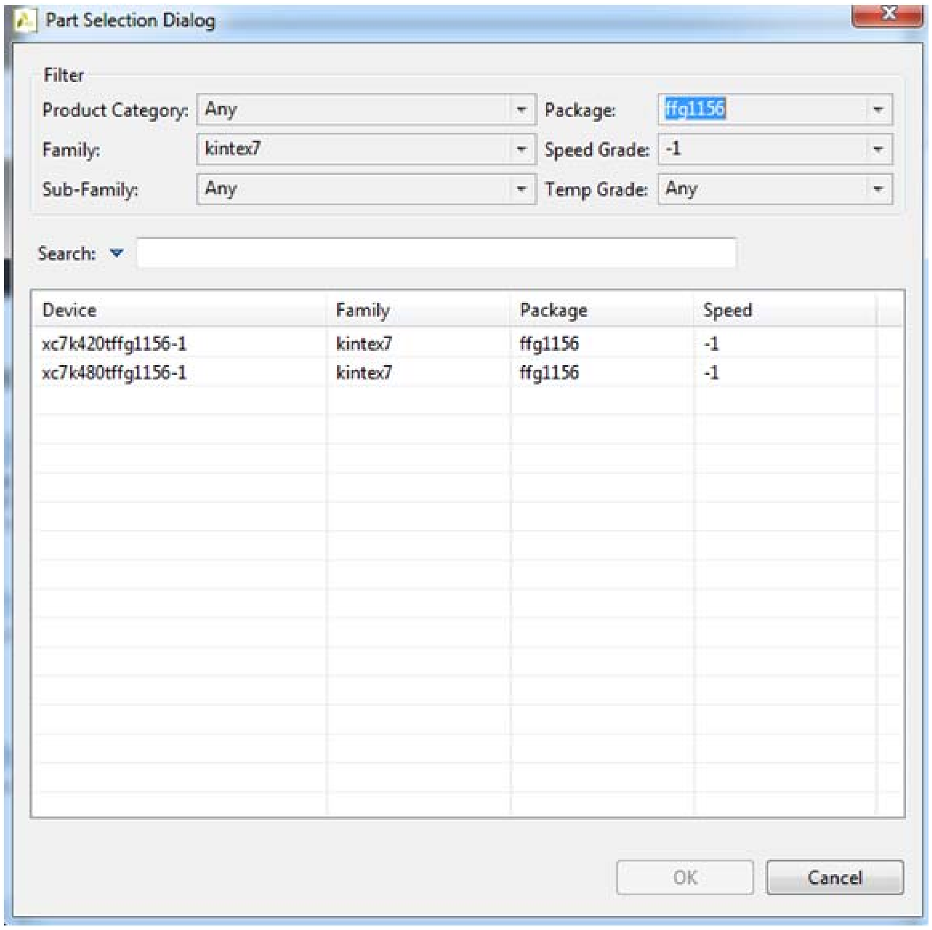
Solution Name: solution1

Clock
Period: 10
Uncertainty:

Part Selection
Part: *[Please select part]*

Figure 3-5: Solution Configuration

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


Part Selection Dialog

Filter

Product Category: Any
Package: ffg1156
Family: kintex7
Speed Grade: -1
Sub-Family: Any
Temp Grade: Any

Search:

Device	Family	Package	Speed
xc7k420tffg1156-1	kintex7	ffg1156	-1
xc7k480tffg1156-1	kintex7	ffg1156	-1

Figure 3-6: Part Selection

9. Click **Finish**, the Vivado HLS GUI should look like Figure 3-11

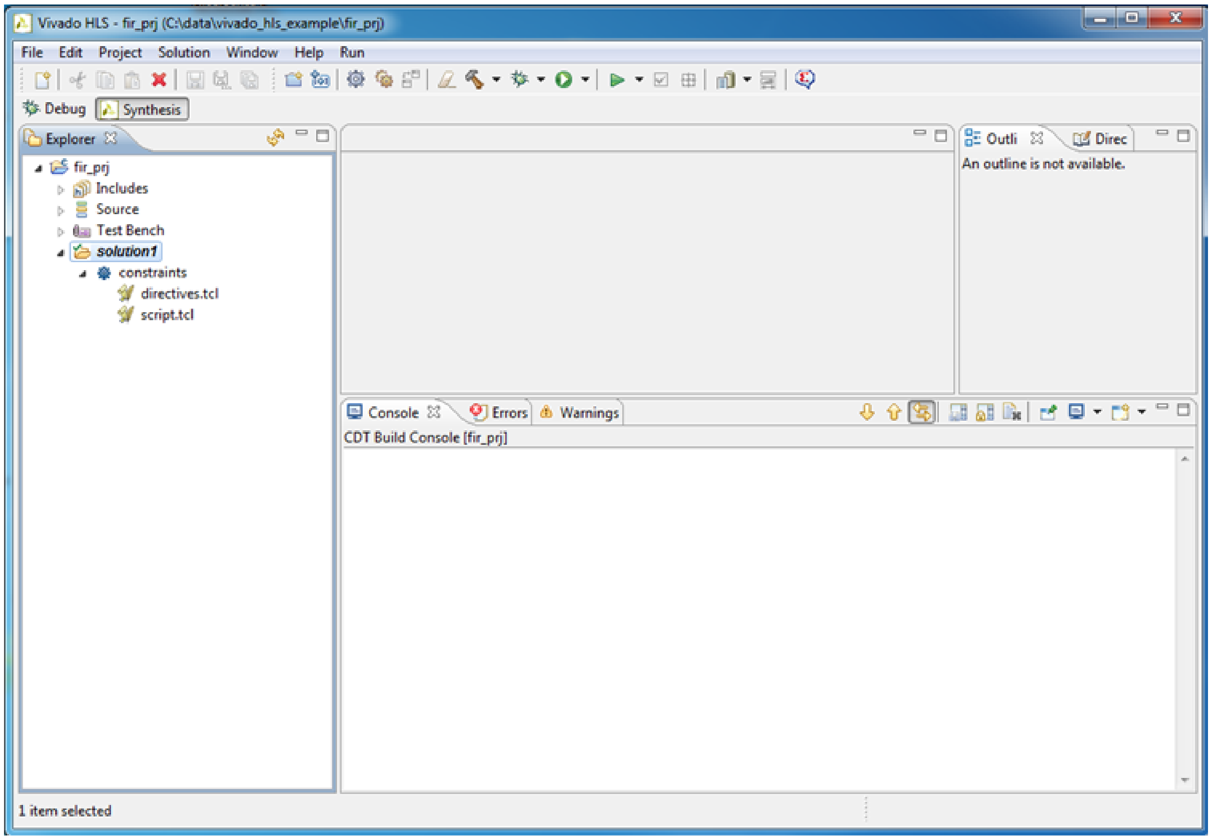

Vivado HLS - fir_prj (C:\data\vivado_hls_example\fir_prj)
File Edit Project Solution Window Help Run
Debug
Synthesis
Explorer
fir_prj
Includes
Source
Test Bench
solution1
constraints
directives.tcl
script.tcl
Outli
Direc
An outline is not available.
Console
Errors
Warnings
CDT Build Console [fir_prj]
1 item selected

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 software **Build** icon on the GUI toolbar (Figure 3-8)

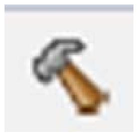


Figure 3-8: Build Icon

2. Click the software **Run** icon on the GUI toolbar (Figure 3-9)

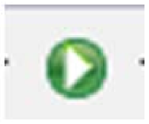
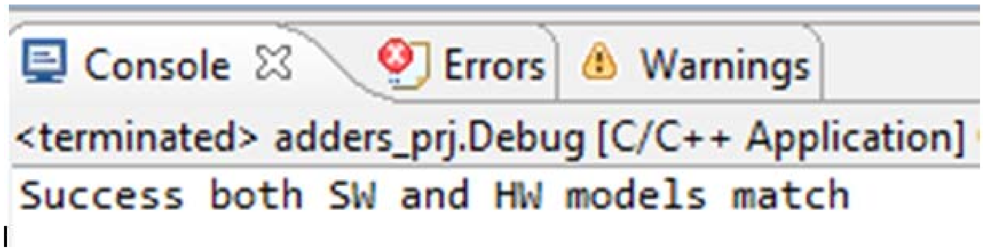


Figure 3-9: Run Icon

3. Select fir_prj.Debug and click ok. Console should look like Figure 3-10.



```
<terminated> adders_prj.Debug [C/C++ Application]
Success both SW and HW models match
```

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.

Note: All designs exported to the System Generator environment must have a global clock-enable signal. If the design does not have this implemented in the RTL, the Export RTL will process with halt with an error.

4. Click **Solution** > **Solution Settings** (Figure 3-11).

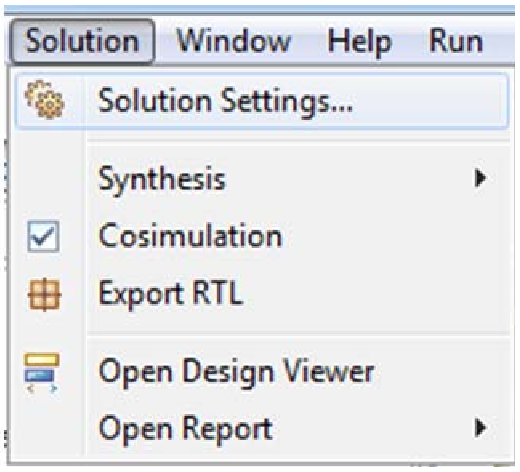


Figure 3-11: Solution Settings

5. Select **General** and then click **Add** (Figure 3-12).

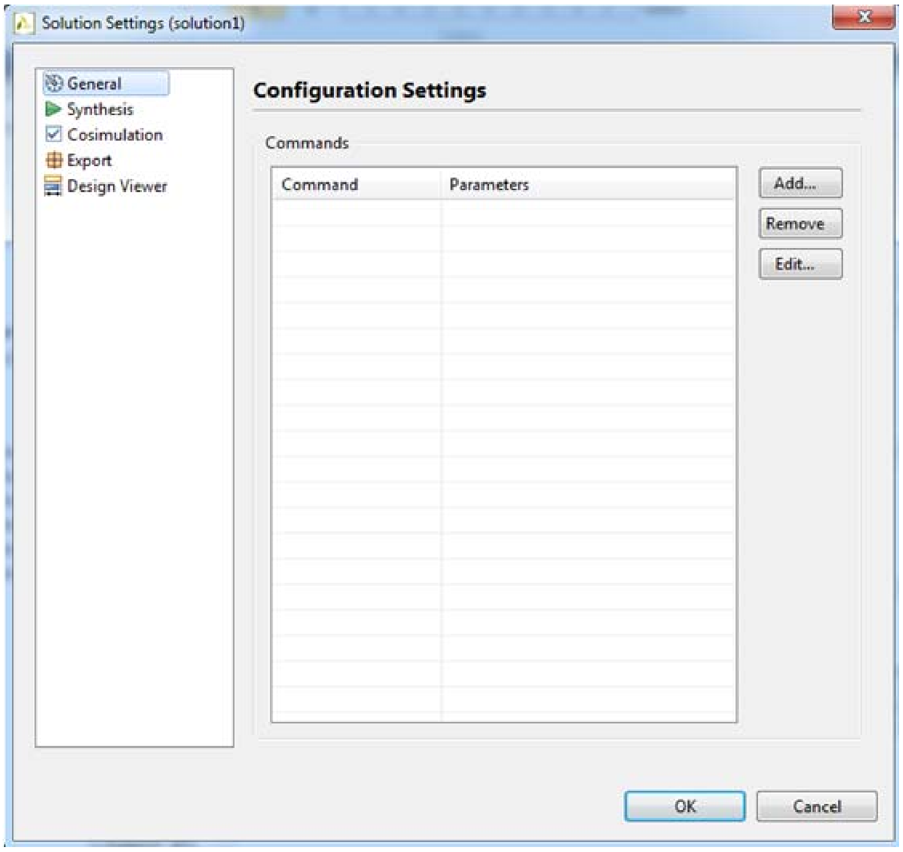


Figure 3-12: Configuration Settings

6. Set **`config_interface`** as the command (Figure 3-13).
7. Click **Clock Enable** (Figure 3-13) and then click **OK**.

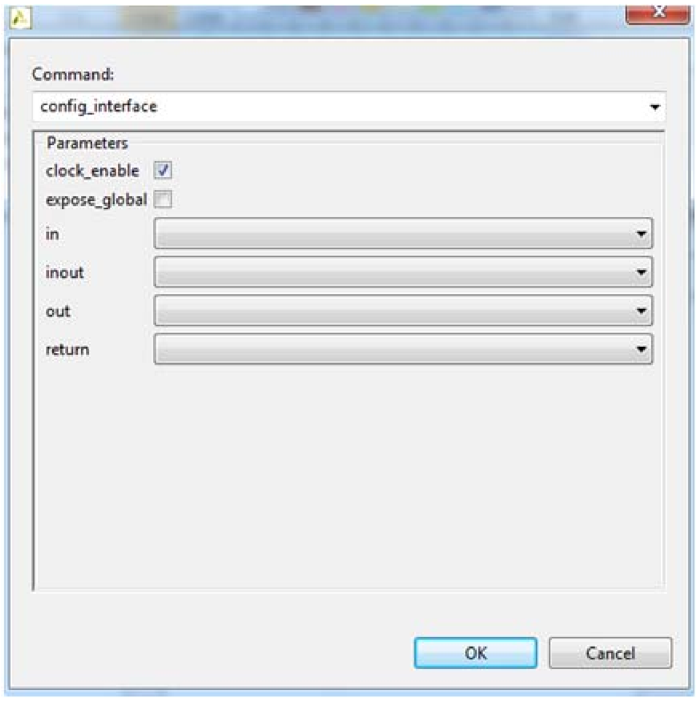

Command:
config_interface
Parameters
clock_enable ☑
expose_global ☐
in
inout
out
return

Figure 3-13: Command Configuration

The next step is to synthesize the C code into an RTL design.

8. Click the **Synthesis** icon on the GUI toolbar (Figure 3-14).

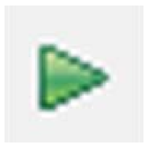


Figure 3-14: Synthesis Icon

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.

9. Click the **RTL Export** icon on the GUI toolbar (Figure 3-15)

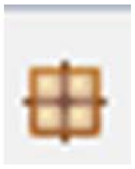
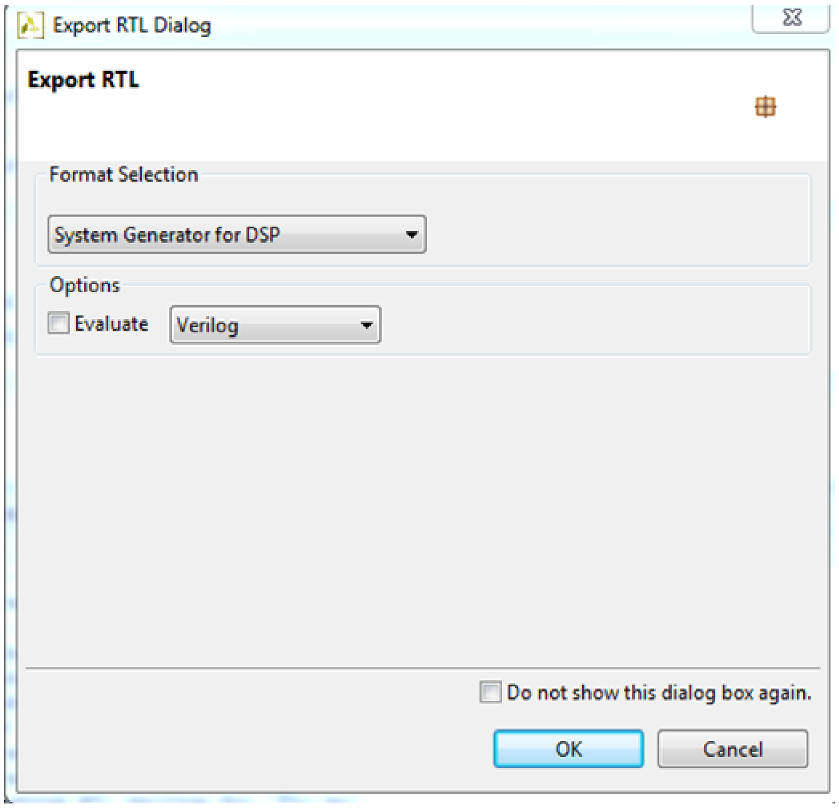


Figure 3-15: RTL Export Icon

10. When the Export RTL menu opens, select **System Generator for DSP** from the drop-down menu (Figure 3-16).
11. Click **OK**.

At this step, logic synthesis can be executed to evaluate if the timing and area estimates reported by Vivado HLS will be met after RTL synthesis. To perform this step, the path to the RTL synthesis executable must be in the system search path. This step is not performed in this example.



Export RTL Dialog

Export RTL

Format Selection

System Generator for DSP

Options

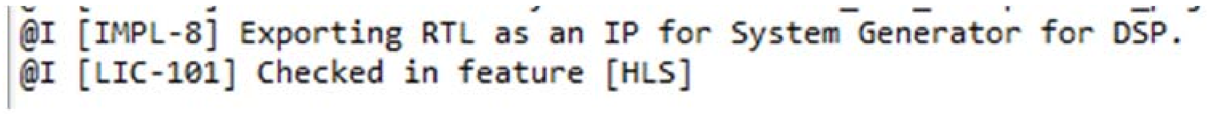
☐ Evaluate Verilog

☐ Do not show this dialog box again.

OK Cancel

Figure 3-16: RTL Export Dialog

12. Check the console for successful execution (Figure 3-17).



```
@I [IMPL-8] Exporting RTL as an IP for System Generator for DSP.
@I [LIC-101] Checked in feature [HLS]
```

Figure 3-17: 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 execute 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 3-16). 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 3-18.

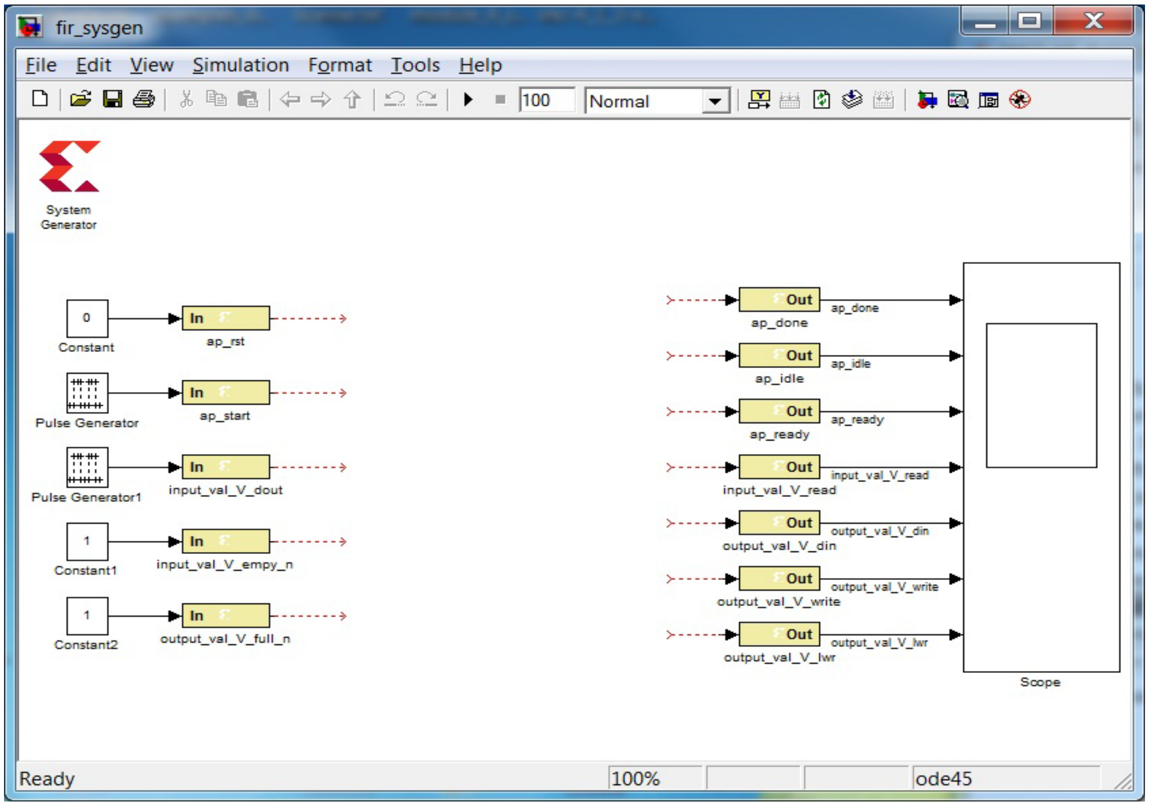


Figure 3-18: Initial fir_sysgen Design

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 3-19.
3. Select **Vivado HLS** to insatiate the initial block.

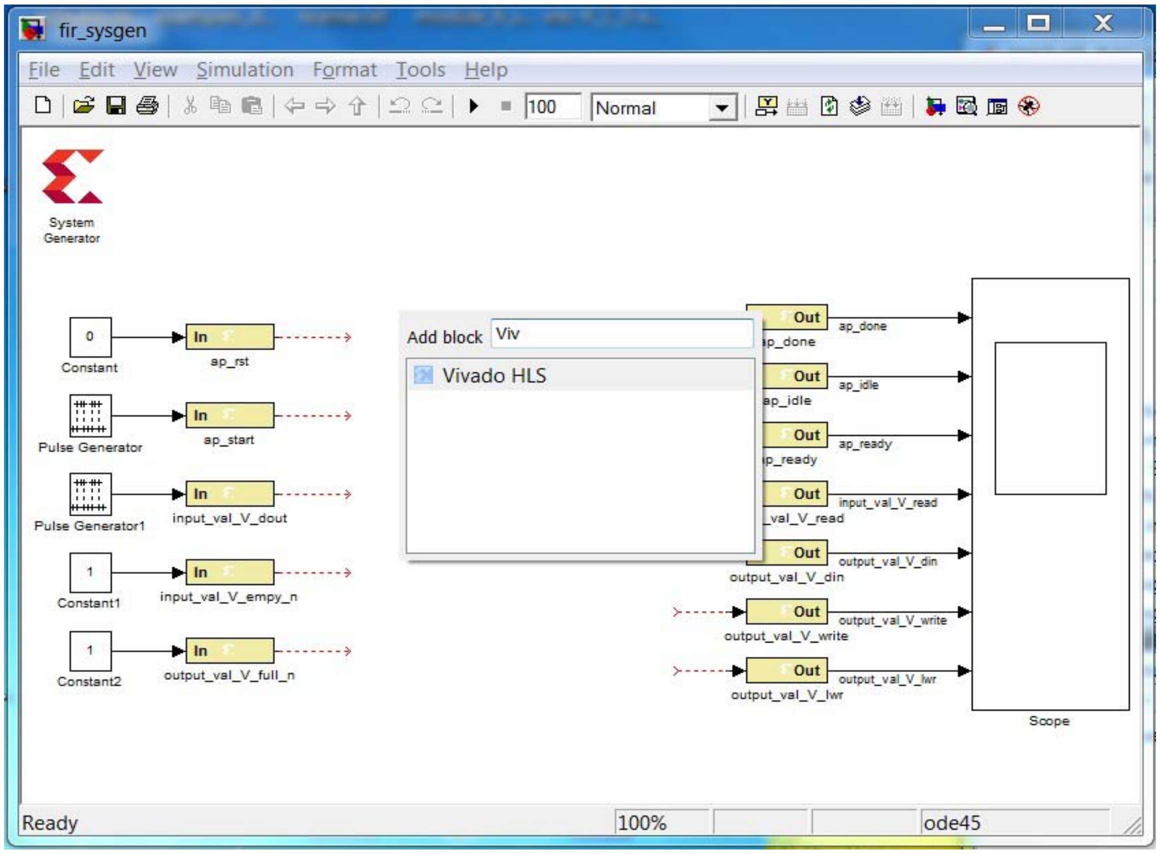
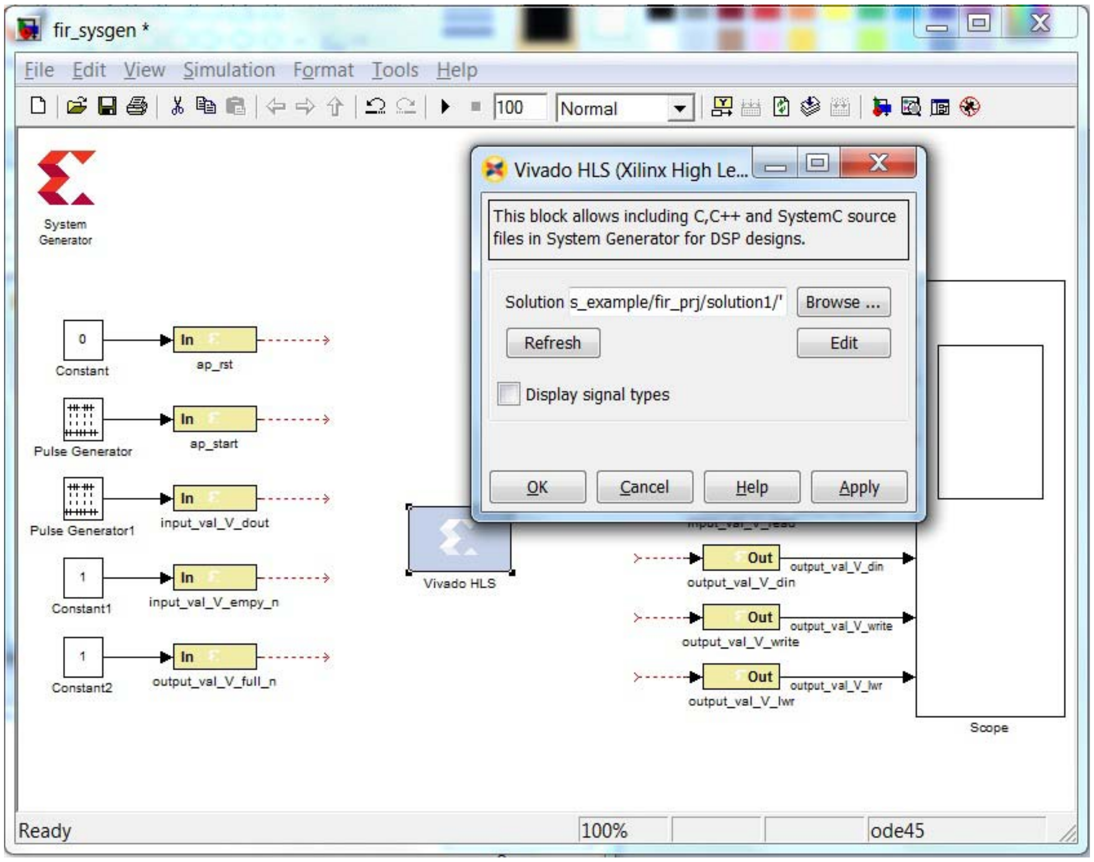


Figure 3-19: Instantiating an Vivado HLS Block

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 3-20).



This block allows including C,C++ and SystemC source files in System Generator for DSP designs.

Solution s_example/fir_prj/solution1/'

Display signal types

Figure 3-20: Importing Vivado HLS IP

Connect the ports on the IP to the design ports to obtain the results shown in Figure 3-21.

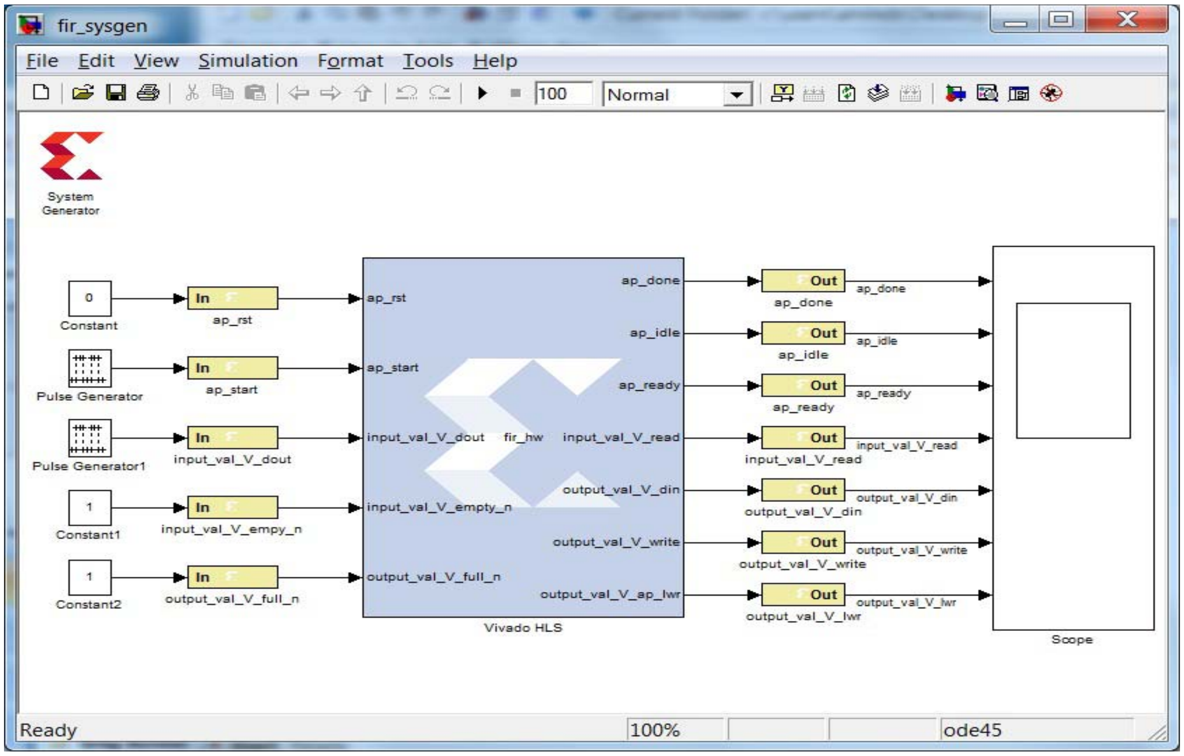


Figure 3-21: Final Design

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 2012.2 Documentation (http://www.xilinx.com/support/documentation/dt_vivado_vivado2012-2.htm)