AutoESL Tutorial: Introduction

UG871 (v2012.1) April 24, 2012

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
04/24/12	2012.1	Initial Xilinx release.

Table of Contents

Introduction

This guide provides an introduction to Xilinx's AutoESL 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 Field Programmable Gate Array (FPGA).

This document is designed to be used with the FIR design example in the `examples` directory located in the AutoESL installation area.

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

- Create an AutoESL project
- Perform validation of 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 implementation
- Review using the AutoESL tool with Tcl scripts

Licensing and Installation

The first steps in using the AutoESL tool are to install the software, obtain a license and configure it. Refer to the *AutoESL Installation Guide (UG869)* found in the `doc` directory in the AutoESL installation area.

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

Overview

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

Design Goals

The hardware design goals for this FIR design project are:

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

The final design should be able to process 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 from the `examples` directory in the AutoESL installation area to a local work area.

Note: PC users - The pathname 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 due to the spaces in the pathname.

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

Filename	Description
`fir.c`	The C code to be synthesized into RTL.
`fir_test.c`	The C test bench for the FIR design. It is used to validate the C algorithm is functioning correctly and will be re-used by the AutoESL tool to verify the RTL.
`fir.h`	The 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`	The data which is expected from the FIR function after normal operation

Learning Goals

This design example shows how to:

- Use the AutoESL Graphical User Interface (GUI) to create an AutoESL design project.
- Validate the C code within the AutoESL tool.
- Analyze the results of synthesis, understand the AutoESL reports and be able to use the Design Viewer analysis capability.
- Apply optimizations to improve the design.

- Verify the functionality of the RTL implementation matches that of the original C design.
- Execute logic synthesis using the ISE Design Suite from within the AutoESL GUI.

Project Entry

The AutoESL 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.

To invoke the GUI, double-click on the desktop icon **AutoESL GUI** or invoke it through the Windows menu by selecting **Start > All Programs > Xilinx AutoESL <version> > AutoESL GUI**. The AutoESL group is shown in Figure 2-1.

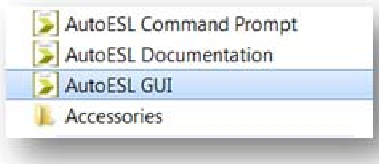
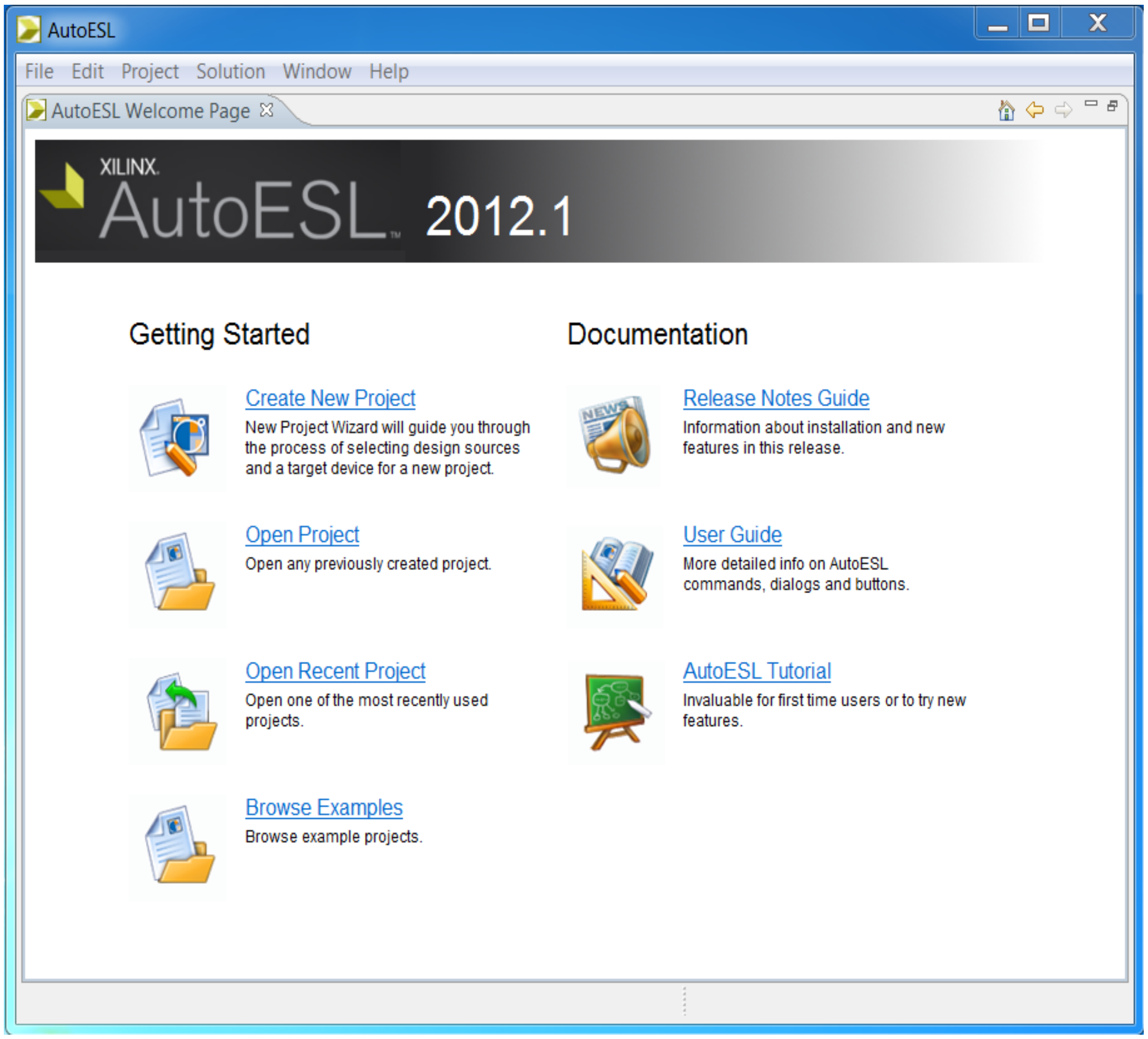


Figure 2-1: **Launching an AutoESL Shell**

AutoESL invokes with the welcome screen, which shows the primary use options for AutoESL.



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

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

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

AutoESL Tutorial
Invaluable for first time users or to try new features.

Figure 2-2: **AutoESL Welcome Screen**

The Getting Started options allow the following tasks to be performed:

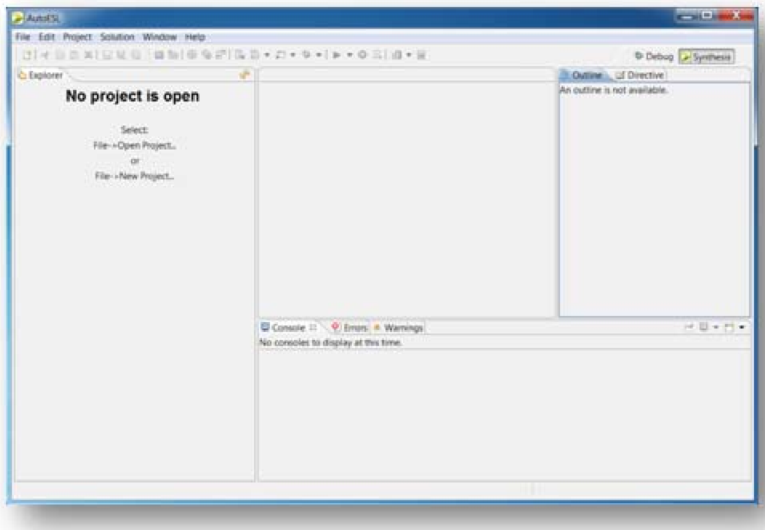
- Create New Project
  - This will launch the project setup wizard.
- Open project
  - Navigate to an existing project.
- Open Recent Project
  - Select from a list of recent projects.

The Documentation tasks available directly from the Welcome Screen shown in Figure 2-2 are:

- Browse Examples
  - Open AutoESL examples. These can also be found in the examples directory in the AutoESL installation area.
- Release Note Guide
  - Open the Release Notes for this version of software.
- User Guide
  - Open the AutoESL User Guide.
- AutoESL Tutorial
  - Select a tutorial to open.

To open these documents click the AutoESL Tutorials and the Tutorial: Introduction. Start by selecting Create New Project.

This brings up the AutoESL GUI as shown in Figure 2-3.



No project is open

Select:
File->Open Project...
or
File->New Project...

An outline is not available.

No consoles to display at this time.

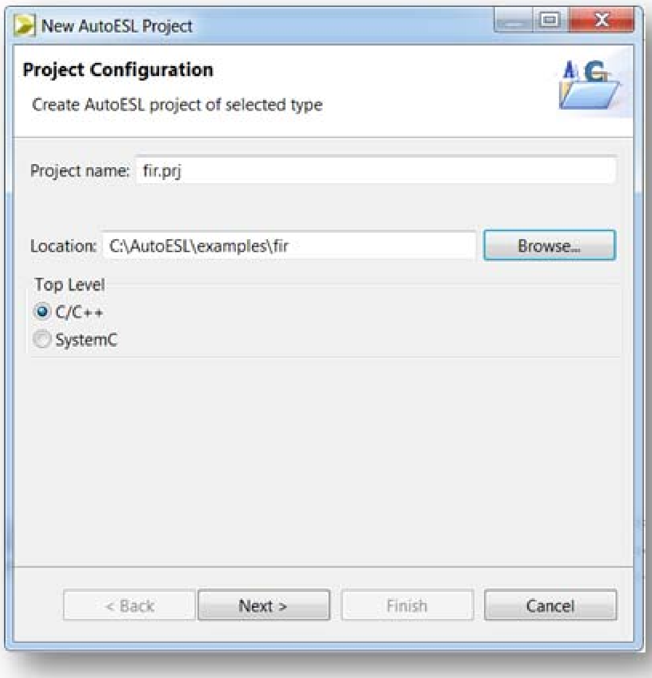
Figure 2-3: **AutoESL GUI**

Creating a New Project

When the GUI opens, you are prompted to open or create a new project.

The steps to create a new project in the `fir` directory in the local work area are:

1. In the menu bar, select **File > New Project...** to open the Project Wizard (see Figure 2-4).



Project name: fir.prj

Location: C:\AutoESL\examples\fir

Top Level

● C/C++

○ SystemC

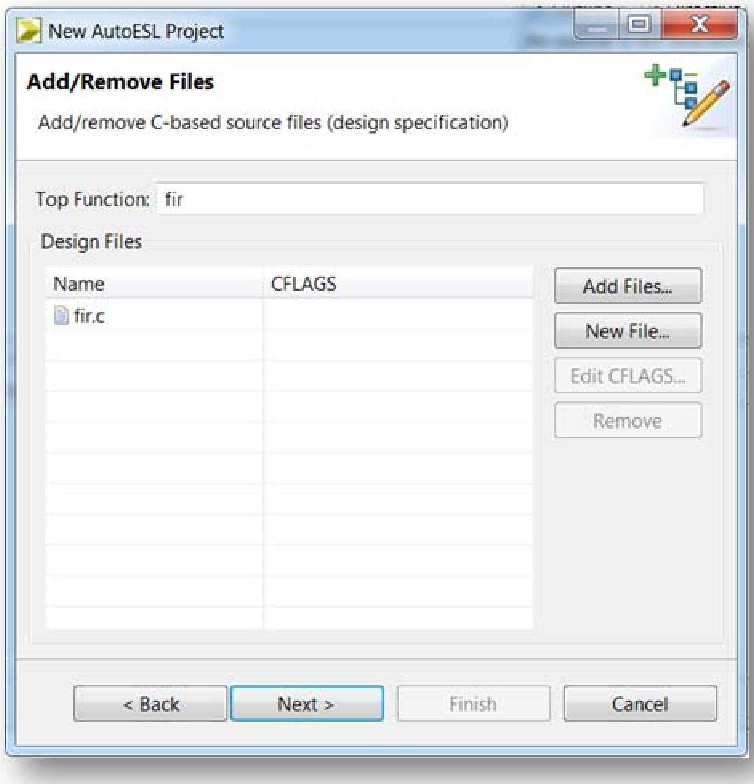
Figure 2-4: **Project Specification**

2. Enter the project name as **`fir.prj`**.
3. Browse to the location of the **`fir`** directory, select it and click **OK**.
4. Specify the top-level as **C/C++**.

   A SystemC project is only required when the top-level is a SystemC SC_MODULE.

5. Click **Next>**.

The next window prompts you for information on the design files (see Figure 2-5).



New AutoESL Project

Add/Remove Files

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

Top Function: fir

Design Files

Name	CFLAGS
fir.c	

Figure 2-5: **Project Design Files**

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

Figure 2-6 shows the window for specifying the test bench files. The test bench and all files used by the test bench, except header files, should be included. The files can be added one at a time or multiple files can be added by using the `ctrl` and `shift` key modifiers.

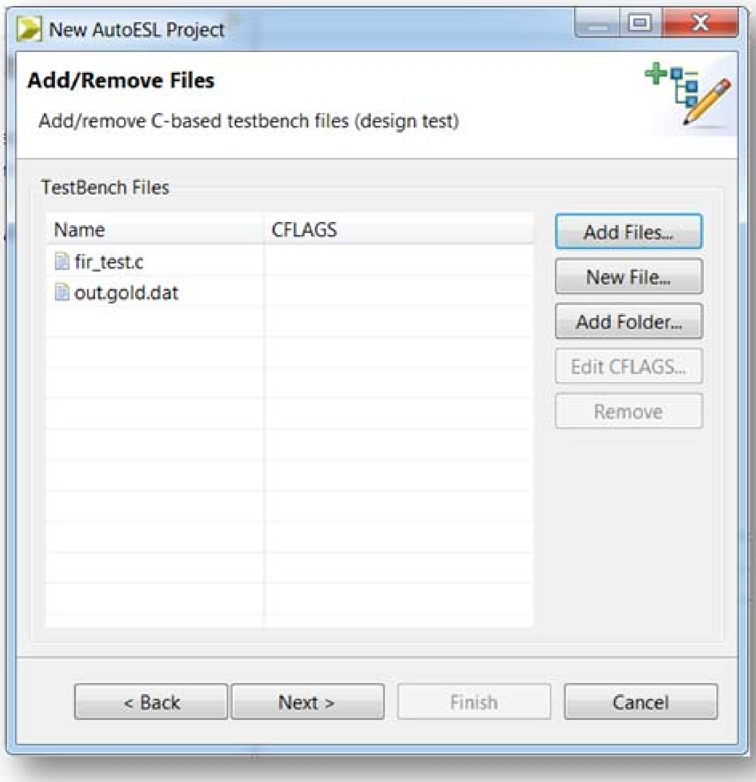


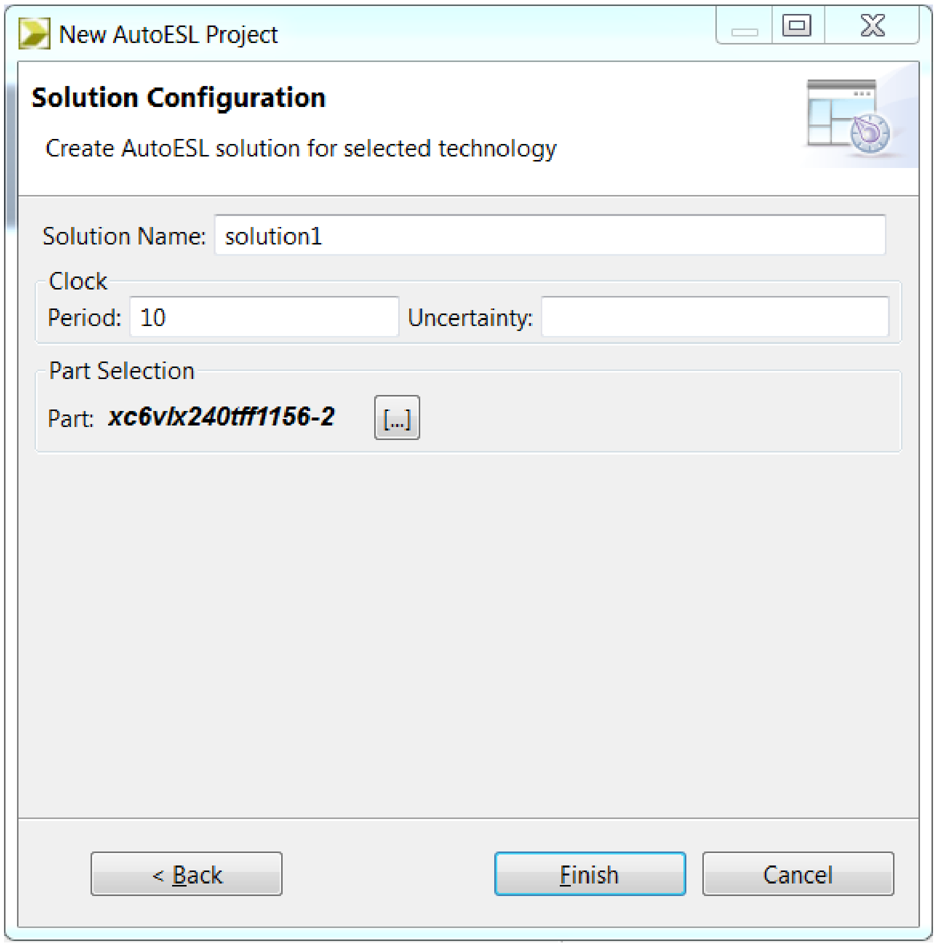
Figure 2-6: **Test Bench Files**

9. Use the **Add Files...** button to include all test bench files: `fir_test.c` and `out.gold.dat`.

10. Click **Next>**.

Failure to 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`) may cause RTL simulation to fail after synthesis due to an inability to find the data files.

The Solution Configuration window (see Figure 2-7) 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 AutoESL Project

Solution Configuration

Create AutoESL solution for selected technology

Solution Name: solution1

Clock

Period: 10 Uncertainty:

Part Selection

Part: *xc6vlx240tff1156-2*

Figure 2-7: **FIR solution**

11. Accept the default solution name (`solution1`), clock period (10ns) and clock uncertainty (12.5% of the clock period, when left blank/undefined).
12. Press 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: Virtex®-6
    - Sub-Family: Virtex-6
    - Package: ff1156
    - Speed Grade: -2
    - Temp Grade: Any

    Then select Device xc6vlx240tff1156-2 from the list of available devices.
13. Press OK to see the selection made, as shown in Figure 2-7.

The AutoESL GUI opens with the project information included, as shown in Figure 2-8.

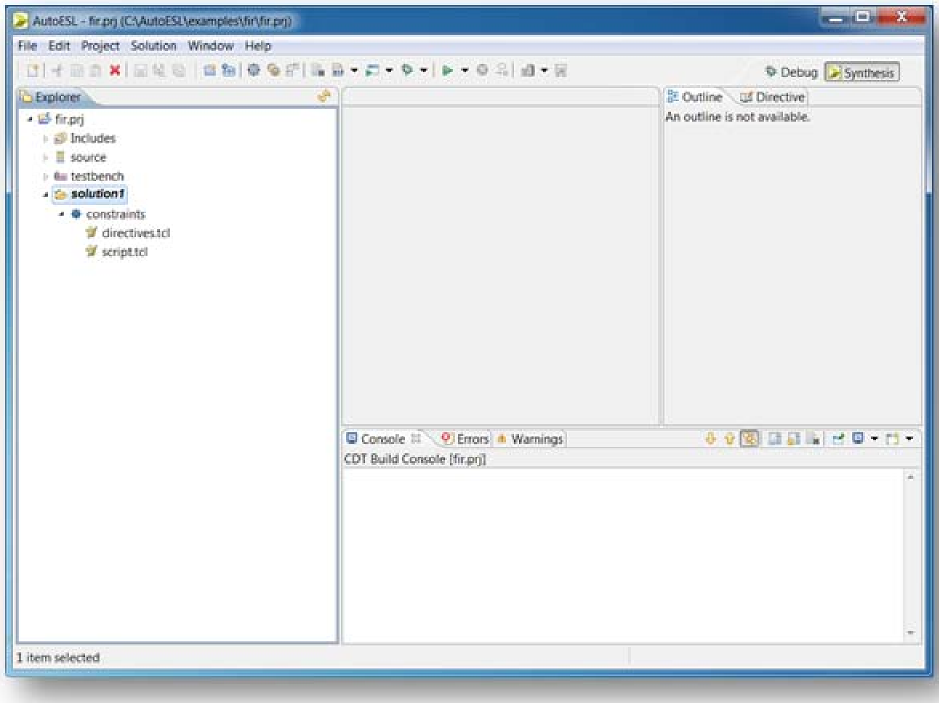


Figure 2-8: **Project GUI**

The project name can be seen on the top line of the Project Explorer pane, on the left hand side.

An AutoESL 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 2-9 and Figure 2-10.

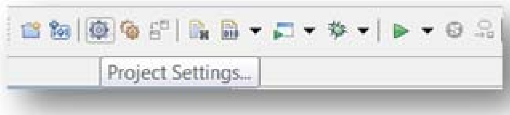


Figure 2-9: **Project Settings**

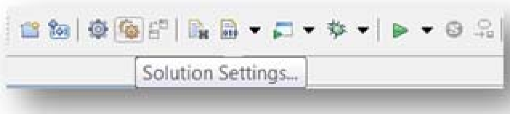


Figure 2-10: **Solution Settings**

Chapter Summary

- An AutoESL project can be setup using the project wizard.
- 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

The C design must be validated prior to synthesis to ensure it is performing correctly. This validation can be performed inside the AutoESL 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 it is self-checking and returns a value of zero to confirm the results are correct:

- 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 is produced to confirm the results are correct and the return value of the test bench `main()` function is set to 0 (zero).
- If the output is different from the golden results, a message is produced to indicate this and the return value of `main()` is set to 1 (one).

The AutoESL tool can reuse the C test bench to perform verification of the RTL.

It will confirm the successful verification of the RTL if the test bench returns a value of zero. If any other value is returned by `main()`, including no return value, it will indicate the RTL verification failed.

If the test bench has the self-checking characteristics mentioned above, the RTL results can be 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 AutoESL tool provides two types of C compilation, `Debug` and `Release`.

- Code compiled for `Debug` can be used in the AutoESL debug environment.

- Code compiled for `Release` will execute faster, since it has no debug information (but cannot be used in the debug environment).

This tutorial will demonstrate both types of C compilation.

C Validation

C simulation can be performed to validate the C algorithm by compiling the C function/design and executing it. This first example will also open the compiled C code in the AutoESL debug environment.

Figure 3-1 shows the `Build` button on the toolbar and the tool pop-up shows the current default build type is for a debug configuration.

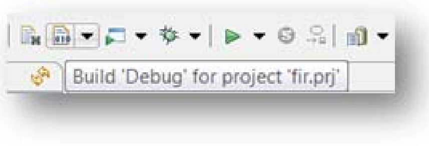
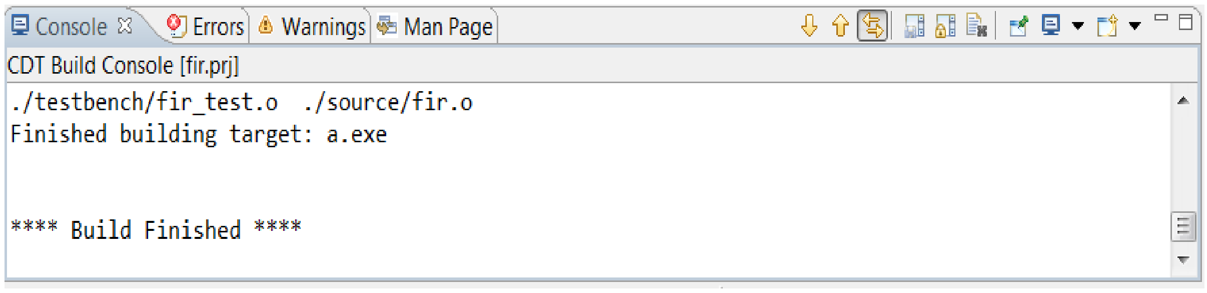


Figure 3-1: **Build (Debug) Toolbar Button**

1. Compile the design by clicking the **Build** button (Figure 3-1).

   The output of the build process is shown in the Console Pane, at the bottom of the GUI, as shown in Figure 3-2.



CDT Build Console [fir.prj]

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


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

Figure 3-2: **Build: Console Output**

The build can now be executed to validate the C function before synthesis. Figure 3-3 shows the toolbar Debug button, used to execute the build. Notice, there is a drop-down arrow on the right-hand side of the button.

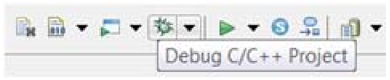


Figure 3-3: **Debug Toolbar Button**

2. Click the **drop-down arrow** on the **Debug** button and select **Debug Configurations...**, as shown in Figure 3-4.

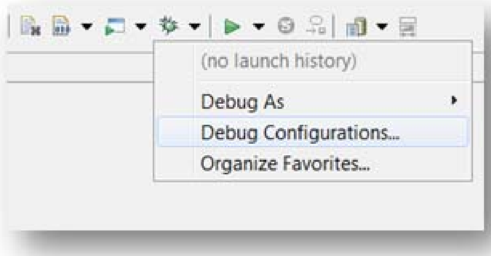
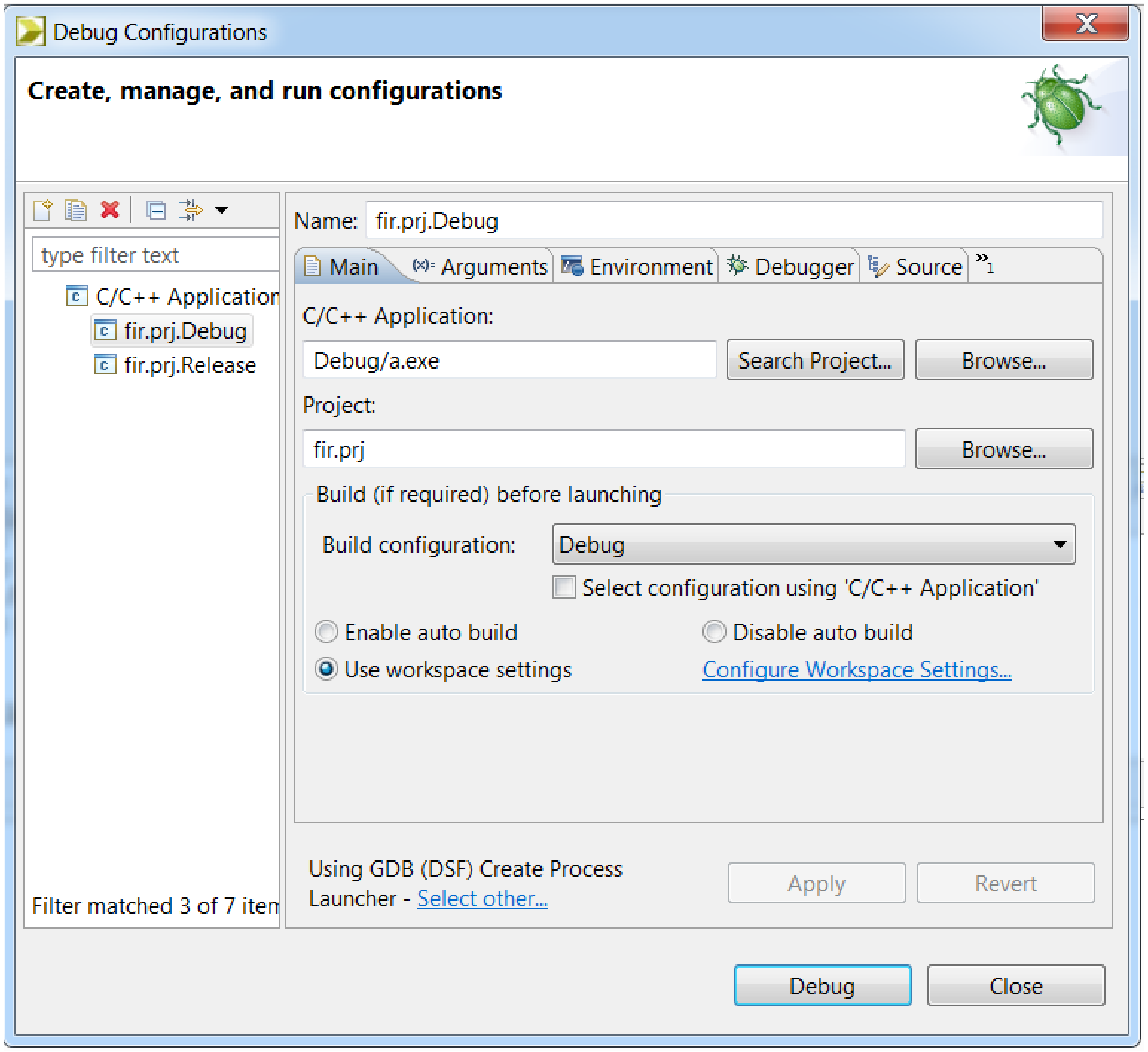


Figure 3-4: **Run Configuration**

This opens the Run Configuration dialog box shown in Figure 3-5.


Debug Configurations

Create, manage, and run configurations

type filter text

C/C++ Application
fir.prj.Debug
fir.prj.Release

Filter matched 3 of 7 items

Name: fir.prj.Debug

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

Project: fir.prj

Build (if required) before launching

Build configuration: Debug

Select configuration using 'C/C++ Application'

Enable auto build
Disable auto build
Use workspace settings [✓]
Configure Workspace Settings...

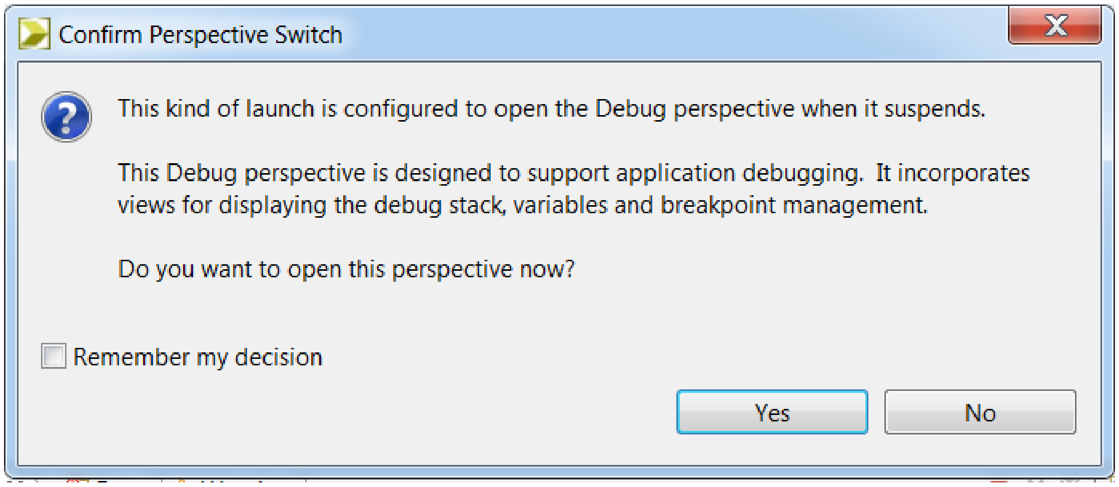
Using GDB (DSF) Create Process Launcher - Select other...

Figure 3-5: **Run Configuration: Debug**

3. Expand **C/C++ Application** and select the **`fir.prj.Debug`** configuration (see Figure 3-5).
4. Click **Debug**.

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

   The debugger opens (see Figure 3-7).

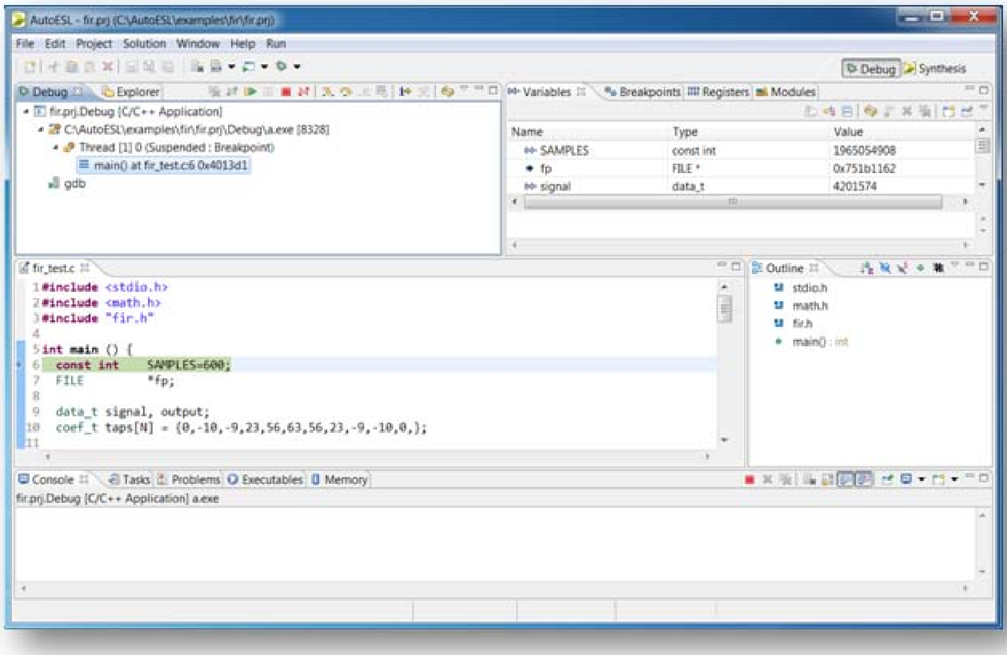


This kind of launch is configured to open the Debug perspective when it suspends.

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?

Figure 3-6: **Debug Environment**



```
fir.prj.Debug [C/C++ Application]
  C:\AutoESL\examples\fir\fir.prj\Debug\a.exe [8328]
    Thread [1] 0 (Suspended : Breakpoint)
      main() at fir_test.c:6 0x4013d1
  gdb
```

Name	Type	Value
SAMPLES	const int	1965054908
fp	FILE *	0x751b1162
signal	data_t	4201574

```
#include <stdio.h>
#include <math.h>
#include "fir.h"

int main () {
  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,};
```

```
fir.prj.Debug [C/C++ Application] a.exe
```

Figure 3-7: **Debugger**

To understand how the debugger can be used, perform the following steps:

6. To step through the code, click the **Step Into** toolbar button or use key **F5**, as shown in Figure 3-8.

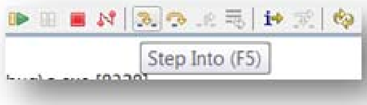
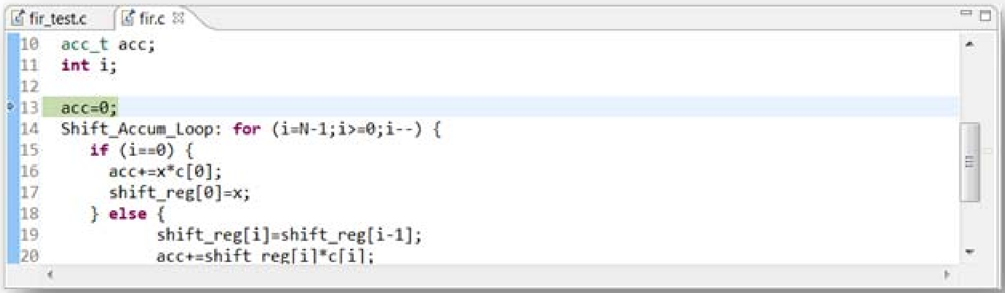


Figure 3-8: **Step Through the Code**

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

   The view shown in Figure 3-9 should be in the code window.

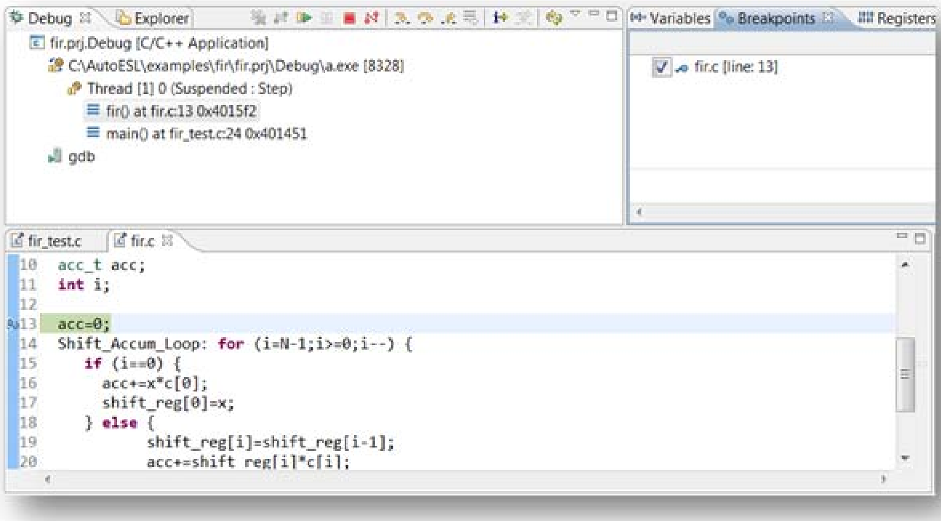


```
acc_t acc;
int i;

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];
```

Figure 3-9: **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** (small dot as shown in line 13), as shown in Figure 3-10.



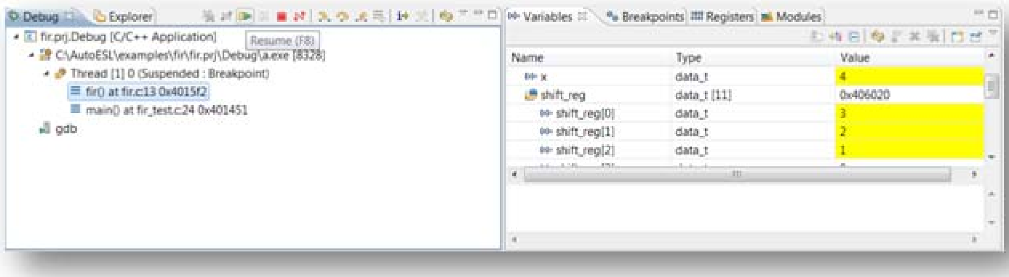
fir.prj.Debug [C/C++ Application]
C:\AutoESL\examples\fir\fir.prj\Debug\a.exe [8328]
Thread [1] 0 (Suspended : Step)
fir() at fir.c:13 0x4015f2
main() at fir_test.c:24 0x401451
gdb

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 3-10: **Adding a Breakpoint**

9. To confirm the breakpoint has been added, open the Breakpoints tab (see Figure 3-10).
10. Open the Variables tab (see Figure 3-11).



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 3-11: **Review the Operation of the C Code**

11. To execute the code until the next breakpoint (the debugger will stop each time it reaches line 13), click the **Resume** button or press key **F8** (see Figure 3-11).
12. To see the shift register update, adjust the **Variables** window to view the `shift_reg` variable (see Figure 3-11) and click the **Resume** button multiple times.
13. To end the debug session, click the **Terminate** button, as shown in Figure 3-12.

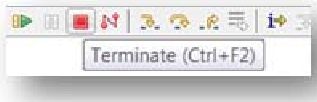


Figure 3-12: **Terminate the Debug Session**

14. Return to the Synthesis perspective by selecting **Synthesis** in the top right-hand corner of the GUI.
15. To run the design and verify the results, click the **Run** button, as shown in Figure 3-13.

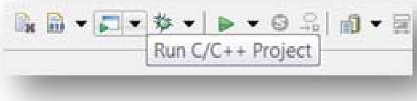
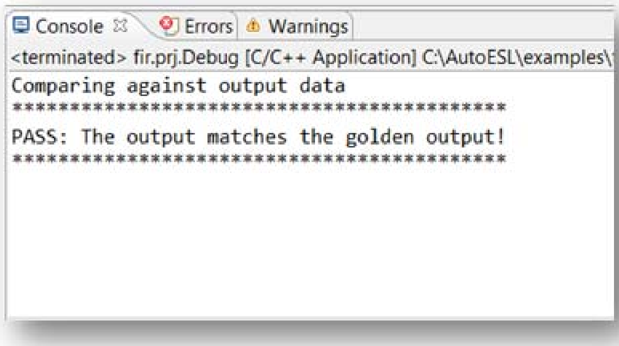


Figure 3-13: **Run the C Code**

The results are shown in the console window (see Figure 3-14), indicating the `fir` function is producing good data and operating correctly. This example assumes the golden data in the `out.gold.dat` file has already been verified as correct.



```
<terminated> fir.prj.Debug [C/C++ Application] C:\AutoESL\examples\
Comparing against output data
*******************************************
PASS: The output matches the golden output!
*******************************************
```

Figure 3-14: **C Validation Results**

Chapter Summary

- The C code should be validated before high-level synthesis to ensure it has the correct operation.
- Overall productivity is enhanced by using a test bench which can self-check the results.

- The AutoESL tool contains a C development environment that can be used to validate and debug the C design prior to synthesis.

Synthesis and Analysis

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

- Synthesis - Create an RTL implementation from the C source code.
- Simulation - Verify the RTL through co-simulation with the C test bench.
- Implementation - Generate and execute the scripts to perform logic synthesis.

Each of these steps can be executed from the toolbar as shown in Figure 4-1. Since Synthesis is the first step in this process, it is located on the left side (the tool pop-up identifies the Synthesis button).

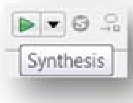


Figure 4-1: **Design Steps**

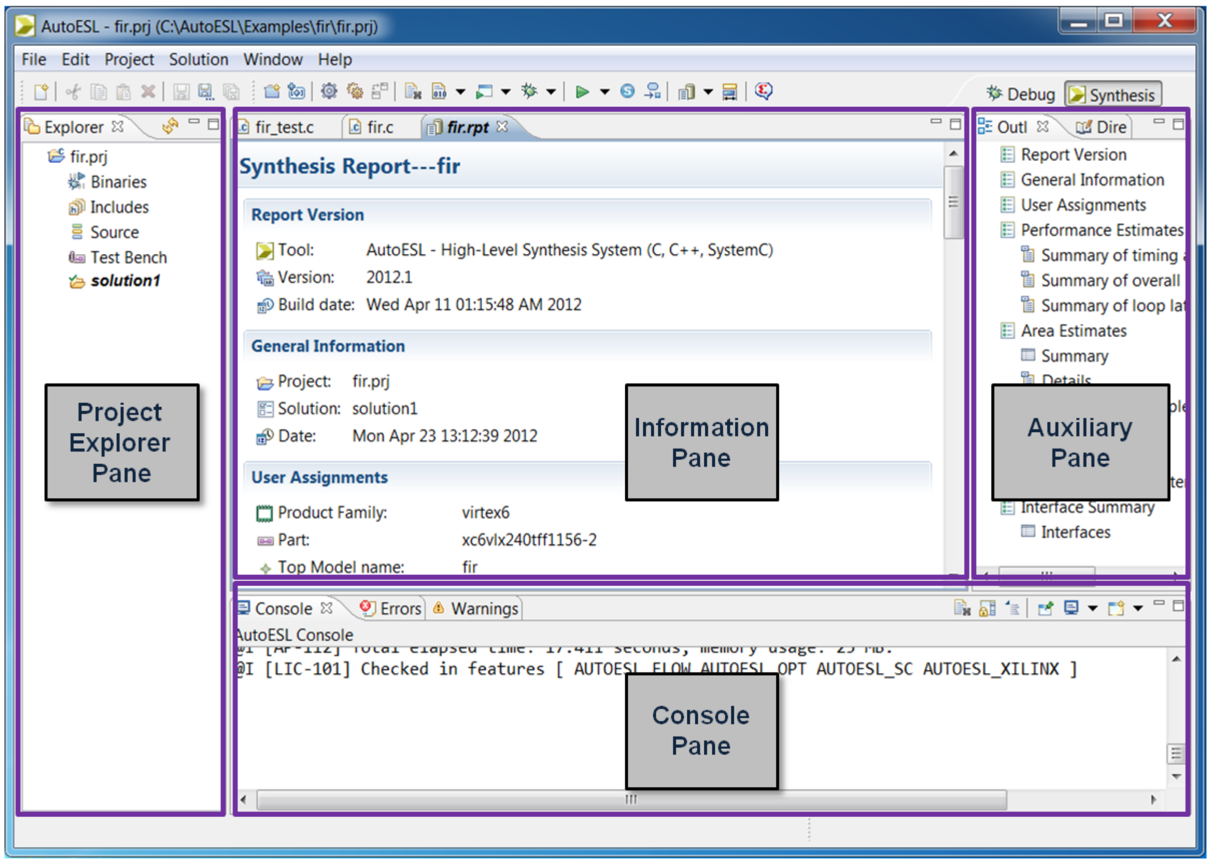
The Simulation button is to the right of the Synthesis button and the Implementation button is to the right of the Simulation button. Both simulation and implementation require synthesis completes before they can be performed and are currently grayed out in Figure 4-1.

Synthesis

The design is ready for synthesis.

Click the **Synthesis** button (see Figure 4-1).

When synthesis completes the GUI updates with the results, as shown in Figure 4-2.



Synthesis Report---fir

Report Version

Tool:	AutoESL - High-Level Synthesis System (C, C++, SystemC)
Version:	2012.1
Build date:	Wed Apr 11 01:15:48 AM 2012

General Information

Project:	fir.prj
Solution:	solution1
Date:	Mon Apr 23 13:12:39 2012

User Assignments

Product Family:	virtex6
Part:	xc6vlx240tff1156-2
Top Model name:	fir

```
@I [LIC-101] Checked in features [ AUTOESL_FLOW AUTOESL_OPT AUTOESL_SC AUTOESL_XILINX ]
```

Figure 4-2: **GUI Overview**

Now that all the window panes in the GUI are populated with data, the various panes and their functions can be fully explained.

- Project Explorer - now shows a `syn` container inside `solution1`, indicating the project has synthesis results. Expanding the `syn` container shows 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 - shows the messages produced during synthesis. Errors and warnings are shown in tabs in the Console pane.
- Information - a report on the results is automatically opened in the Information pane when synthesis completes. The Information pane also shows the contents of any files opened from the Project Explorer pane.
- Auxiliary - is cross-linked with the Information pane. Since the information pane currently shows the synthesis report, the Auxiliary pane shows an outline of this report.

TIP: Clicking on the items in the Report Outline in the Auxiliary pane, will cause the Information pane to automatically scroll to that point of the report.

Table 4-1 explains the categories in the synthesis report.

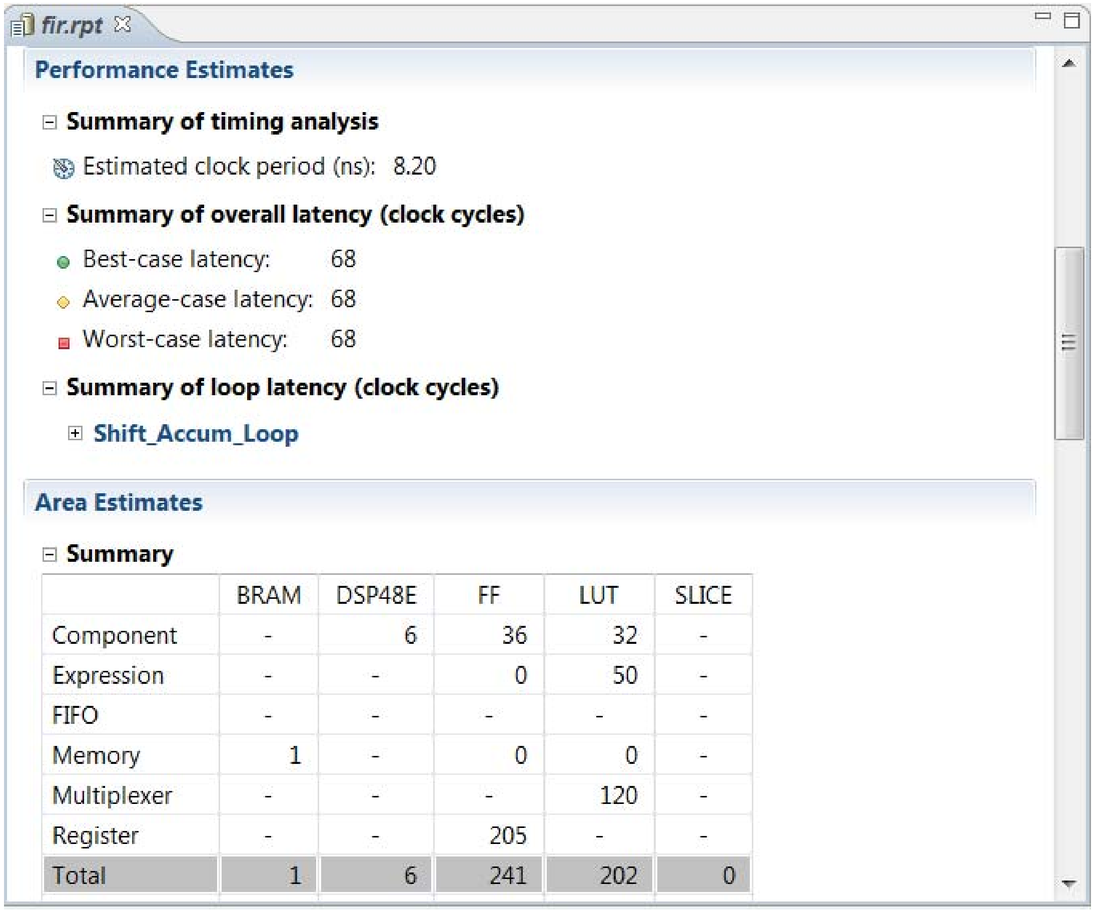
Table 4-1: **Synthesis Report Categories**

Category	Sub-Category	Description
Report Version	---	Details on the version of the AutoESL 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 will be different. If the design is pipelined this section will show the throughput. Without pipelining the throughput is the same as the latency, the next input will be 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 4-1: **Synthesis Report Categories**

Category	Sub-Category	Description
Area Estimates	Summary	This shows the resources (LUTS, Flip-Flops, DSP48s etc.) 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 inlined, 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, comparators etc. 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: port names, directions, bit-widths, etc.

A section of the report is shown in Figure 4-3.


fir.rpt

Performance Estimates

⊟ **Summary of timing analysis**

Estimated clock period (ns): 8.20

⊟ **Summary of overall latency (clock cycles)**

● Best-case latency: 68
◆ Average-case latency: 68
■ Worst-case latency: 68

⊟ **Summary of loop latency (clock cycles)**

⊞ Shift_Accum_Loop

Area Estimates

⊟ **Summary**

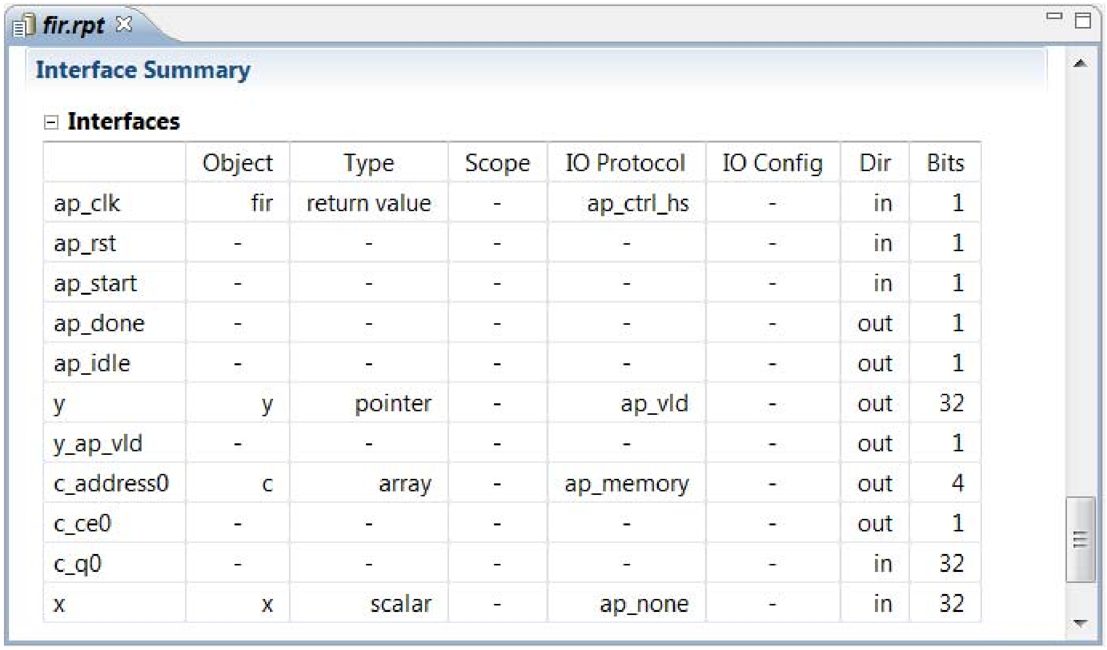
	BRAM	DSP48E	FF	LUT	SLICE
Component	-	6	36	32	-
Expression	-	-	0	50	-
FIFO	-	-	-	-	-
Memory	1	-	0	0	-
Multiplexer	-	-	-	120	-
Register	-	-	205	-	-
Total	1	6	241	202	0

Figure 4-3: **solution1 Performance and Area Summary**

This report shows the initial solution to be:

- Meeting the clock frequency of 10ns
- Taking 68 clock cycles to output data
- Using six DSP48 blocks
- Using one BRAM memory block.

Selecting Interface Summary from the Report Outline in the Auxiliary pane, or scrolling down the report in the Information pane, shows the details of the interface (see Figure 4-4).



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	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 4-4: **solution1 Performance and Area Summary**

- 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 the RTL design can be automatically verified by the `autosim` feature.
  - This IO protocol ensures the design will not start operation until input port `ap_start` is asserted (high), it will indicate 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, AutoESL determines the most appropriate RAM interface (if a dual-port will improve 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 an input valid.

Design Analysis: The Design Viewer

When synthesis has completed, the Design Viewer can be used to examine the design implementation in detail. The Design Viewer can be invoked from the AutoESL toolbar (or from the Solutions menu).

To open the Design Viewer, click on the **Design Viewer** toolbar button, as shown in Figure 4-5.

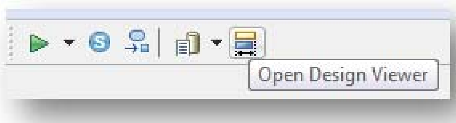
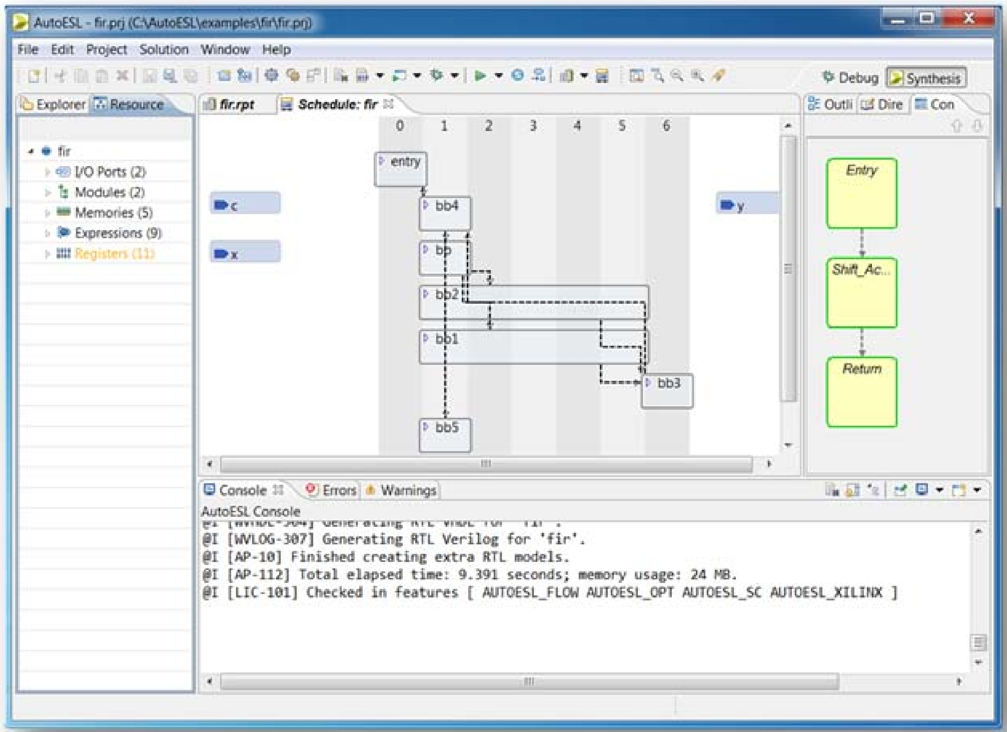


Figure 4-5: **Design Viewer Button**

The Design Viewer opens, as shown in Figure 4-6.



```
AutoESL Console
@I [WVLOG-307] Generating RTL Verilog for 'fir'.
@I [AP-10] Finished creating extra RTL models.
@I [AP-112] Total elapsed time: 9.391 seconds; memory usage: 24 MB.
@I [LIC-101] Checked in features [ AUTOESL_FLOW AUTOESL_OPT AUTOESL_SC AUTOESL_XILINX ]
```

Figure 4-6: **Design Viewer**

The Design Viewer is comprised of three panes:

- Control Flow Graph - is the closest to the software view and is the best place to begin analysis.
- Schedule Viewer - shows how the operations are scheduled in internal control steps (these are mapped to clock cycles and this view may not correlate exactly to clock cycles in all cases).
- Resource Viewer - shows how the operations in the Schedule Viewer are mapped to specific hardware resources.

In the Control Flow Graph pane, double-click on the **Shift_Accum_Loop** block and navigate down into the details of the loop, as shown in Figure 4-7.

When the `Shift_Accum_Loop` is selected, the corresponding items in the Schedule Viewer are also selected (see Figure 4-7).

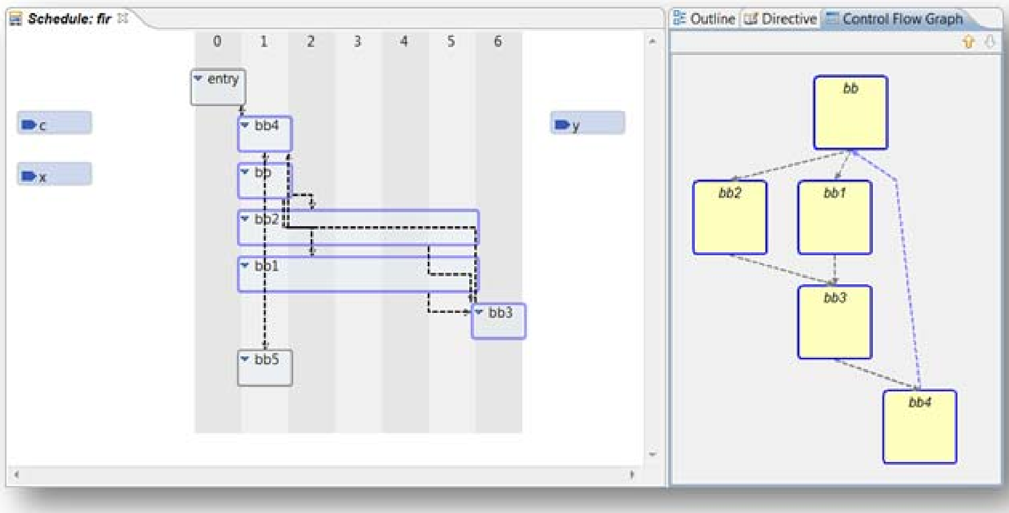


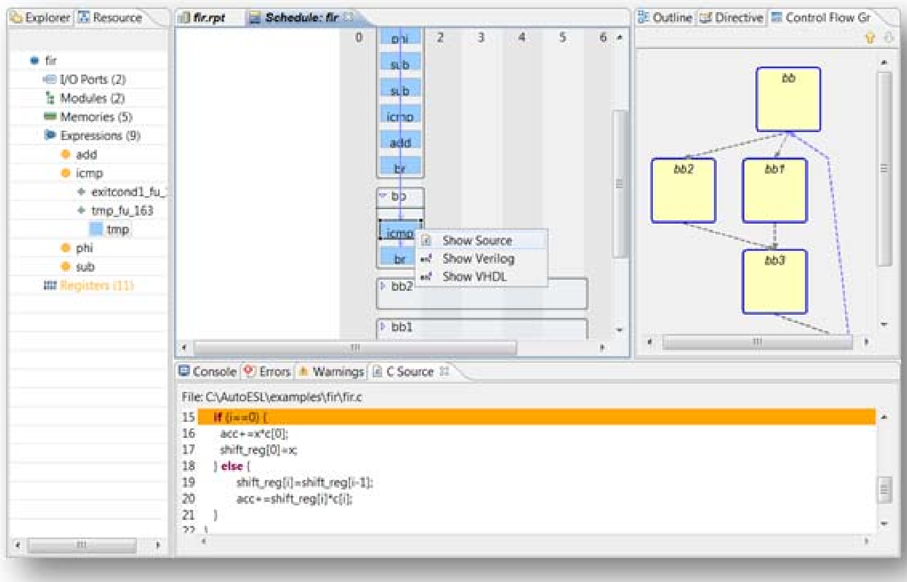
Figure 4-7: **Cross-Probing the CFG and Schedule Viewer**

Since the Control Flow Graph is closest to the software, this view shows how the design is operating.

- The `Shift_Accum_Loop` loop starts in basic block `bb`.
- It can proceed to block `bb2` or block `bb1`.
- Both blocks return control to block `bb3`.
- Before ending in block `bb4` which returns to block `bb`.

The following shows how the Design Viewer can be used 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 4-7.
2. Select the **`icmp`** operator and right-click to see the pop-up menu (see Figure 4-8).



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 4-8: **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 (see Figure 4-8).

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

Repeating the above steps with the other blocks in the design (`bb1-4`) gives a more detailed understanding of how the code in the design is implemented, allowing the initial understanding of the code to be further developed:

- The loop starts in block `bb`.
  - This is the if-condition at the start of the loop.
  - Since it is a non-conditional for-loop, the loop must be started. The exit condition will be checked at the end of the loop.
- It can proceed to 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 *AutoESL Library Guide (UG870)*.
    - Block `bb1` is the if-branch inside the for-loop and performs a single memory read, write and multiplication.
- Both blocks return control to block `bb3`.
    - This block performs the accumulation common to both branches of the if-else statement.
- Before ending 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, e.g. 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 4-9 gives a more detailed view of how the Resource Viewer shows sharing (or lack of it, in this case).

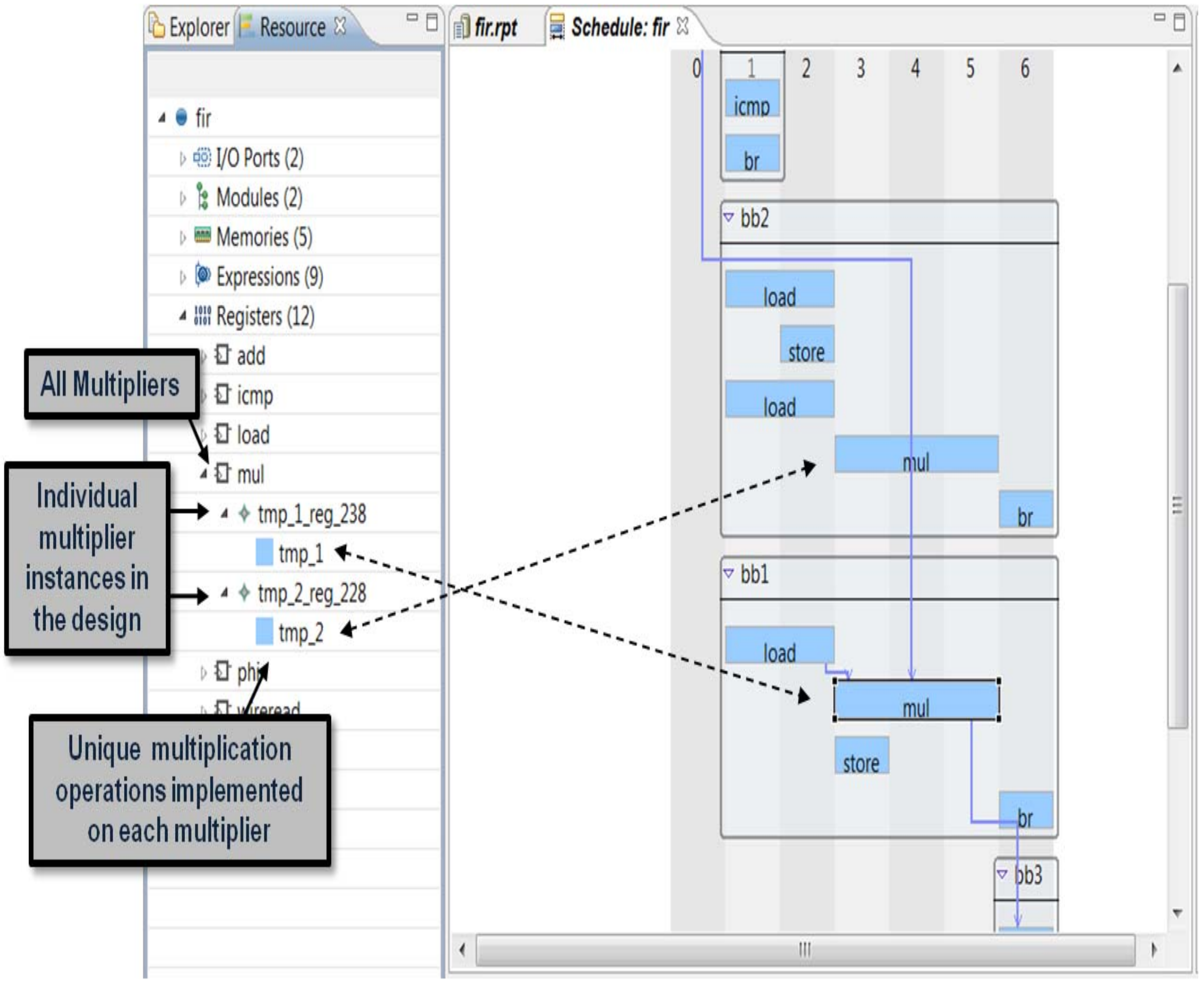


Figure 4-9: **View Sharing in the Design Viewer**

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.
- Selecting a multiplier in the Schedule Viewer will highlight which multiplier instance is used to implement it.
  - If a register is also highlighted, it indicates the output is registered.
- Expanding each multiplier in the Resource Viewer shows how many unique multiplication operations in the code (shown as blue squares) are mapped onto each hardware resource.

- Clicking on the operations (blue squares) shows 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 will show that multiple read (`load`) and write (`store`) operations are implemented on the same memory resource. This will also show 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, shows 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, since the same resources are used in every iteration of the loop, but cost cycles since 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 will require 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 and each element will be implemented by a register, allowing a shift register to be used for the implementation.
  - Once the loop is unrolled, the AutoESL tool may perform this step automatically since it is a small RAM. All optimizations performed on the design are reported in the Console. However, since 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, since 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 AutoESL tool may decide not to 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) but the only real way to be sure if they could be shared is to see the results after place and route.
  - For this example, sharing will be forced. This will demonstrate a useful technique for minimizing area.

But 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 4-3) gave an estimate that six DSP48s will be 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 the C code is using the correct bit-accurate types before proceeding to synthesis or it will result in larger and slower hardware.

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

Chapter Summary

- When synthesis completes a report on the design 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 it does what it is supposed to do (rounding and truncation are of critical importance) and validate 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 may be compared against the new results, a new solution will be created.

1. In the AutoESL GUI, select the **New Solution** button, as shown in Figure 5-1.

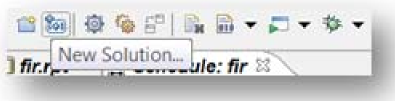


Figure 5-1: **New Solution Toolbar Button**

The New Solution dialog window opens.

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

`solution2` is created and opened.

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

Open files use up memory so if they are required keep them open, otherwise it is good practice to close them.

6. 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 if the macro BIT_ACCURATE is defined during compile or synthesis, the AutoESL 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 will result in different output data from the `fir` function. The test bench (`fir_test.c`) is also written to ensure 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 will use these new project settings and will give 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` will change if it is re-executed and the project settings have been changed.

To ensure 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, as shown in Figure 5-2.

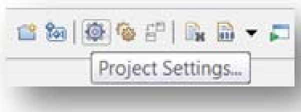


Figure 5-2: **Project Settings**

Update the setting for the C simulation:

- Define the macro BIT_ACCURATE.
- Updated the data file used by the test bench.
- Ensure the new C data types are correctly compiled.

2. In the Simulation section of the Project Settings, select **`fir_test.c`**, click the **Edit CFLAGS...** button and add **`-DBIT_ACCURATE`** to define the macro. Then 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.).

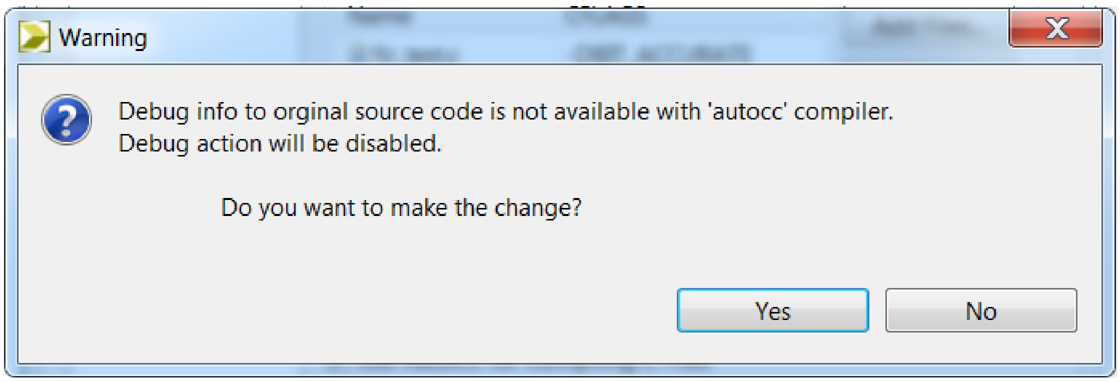
   Note: There is no need to include any AutoESL header files, such as `ap_cint.h`, using the include flag, the AutoESL include directory is automatically searched.

The next step is to use new output data in the test bench comparison.

3. In the Simulation section, select the **`out.gold.dat`** file and click the **Remove** button to remove it 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.

4. Click the **Add Files...** button and add the **`out.gold.8.dat`** file to the project.
5. Select the check box **Use AutoCC Compiler**. This opens the warning dialog box shown in Figure 5-4.


Warning

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

Do you want to make the change?

Figure 5-3: **Warning Dialog Box**

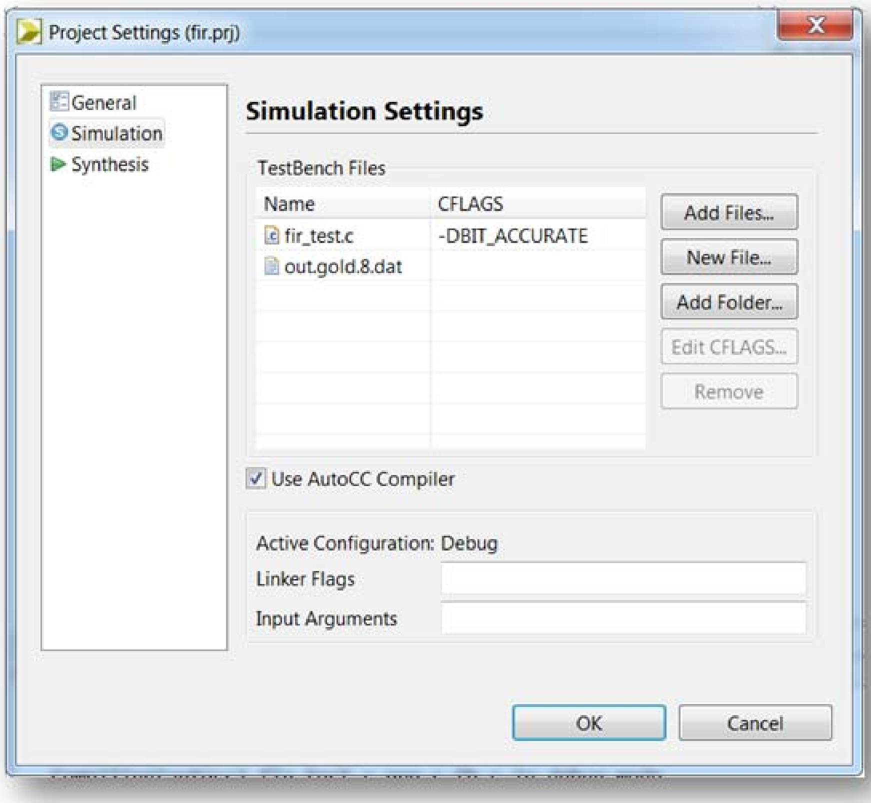
Note: Designs compiled with `AutoCC` will simulate with bit-accurate behavior but cannot be analyzed in the debug environment.

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

   The updated Simulation section is shown in Figure 5-4.

b. Click **OK**.

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



Simulation Settings

TestBench Files

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

☑ Use AutoCC Compiler

Active Configuration: Debug

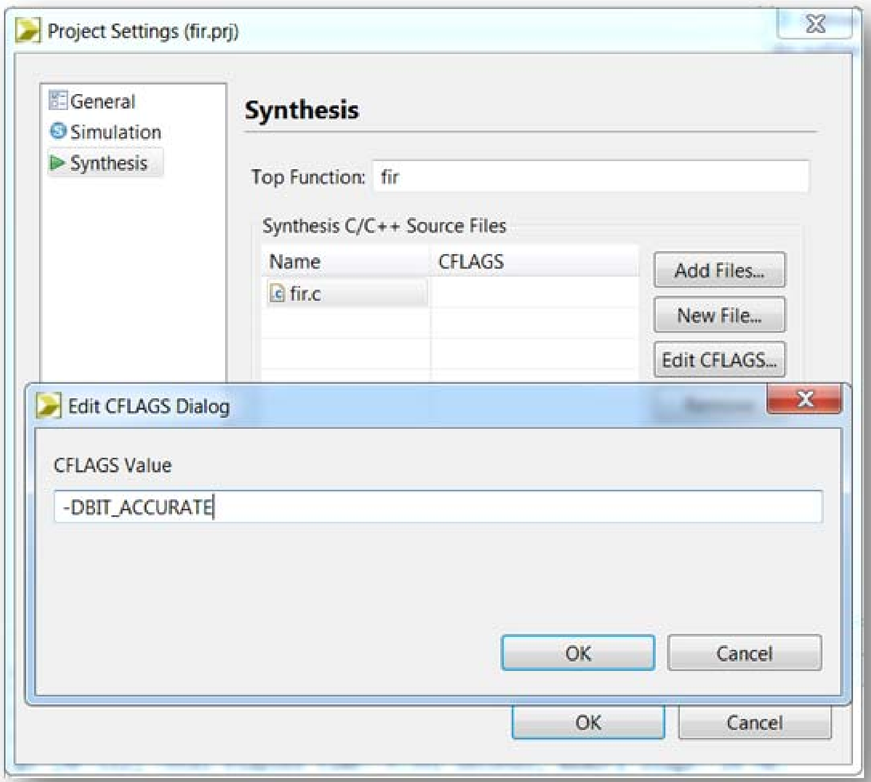
Linker Flags

Input Arguments

Figure 5-4: **Project Simulation Setttings**

6. In the Synthesis section of the project setting, select source file **`fir.c`**, select the **EDIT CFLAGS...** button and add **`-DBIT_ACCURATE`** into the CFLAGS dialogue box, as shown in Figure 5-5 and click **OK**.

   Figure 5-5 shows the settings for the CFLAGS to synthesize the design using bit-accurate types.



Top Function: fir

Synthesis C/C++ Source Files

Name	CFLAGS
fir.c	

CFLAGS Value

-DBIT_ACCURATE

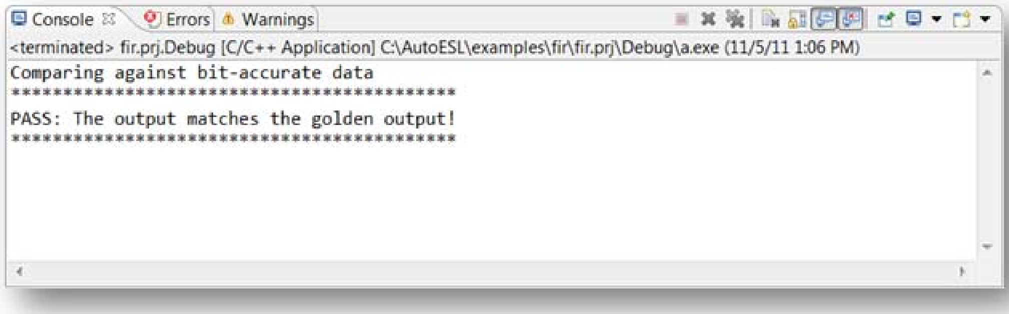
Figure 5-5: **CFLAGS Settings**

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

7. Click the **Build** toolbar button to re-compile the function.
8. Click the **Run** toolbar button to re-execute the C simulation.

   The output is shown in the console window (see Figure 5-6).

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



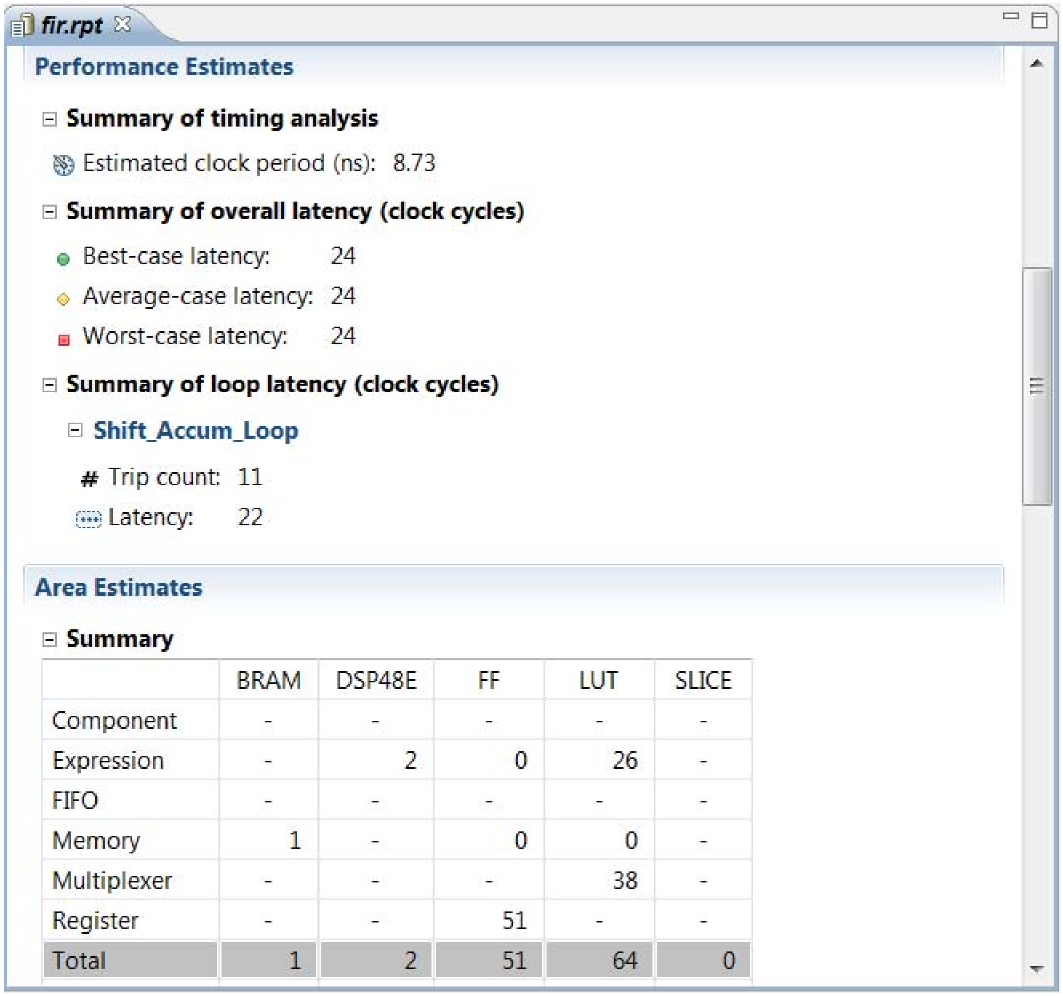
```
<terminated> fir.prj.Debug [C/C++ Application] C:\AutoESL\examples\fir\fir.prj\Debug\a.exe (11/5/11 1:06 PM)
Comparing against bit-accurate data
********************************************
PASS: The output matches the golden output!
********************************************
```

Figure 5-6: **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 now those shown in Figure 5-7, 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 5-7: **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 5-8).

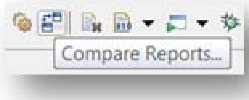
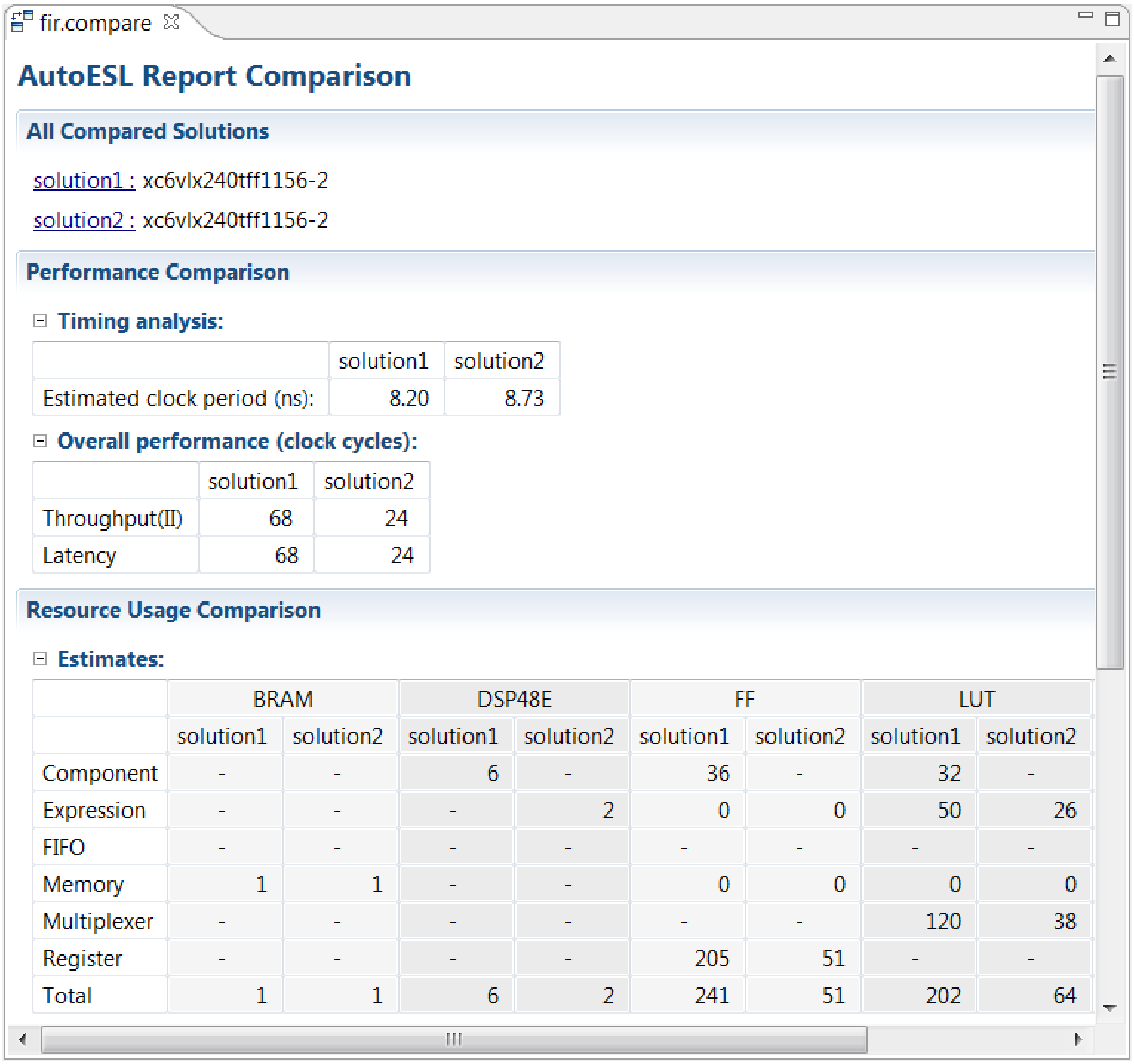


Figure 5-8: **Compare Reports Button**

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

Figure 5-9 shows the comparison of the reports for `solution1` and `solution2`.



AutoESL Report Comparison

All Compared Solutions

solution1 : xc6vlx240tff1156-2

solution2 : xc6vlx240tff1156-2

Performance Comparison

Timing analysis:

	solution1	solution2
Estimated clock period (ns):	8.20	8.73

Overall performance (clock cycles):

	solution1	solution2
Throughput(II)	68	24
Latency	68	24

Resource Usage Comparison

Estimates:

	BRAM		DSP48E		FF		LUT	
	solution1	solution2	solution1	solution2	solution1	solution2	solution1	solution2
Component	-	-	6	-	36	-	32	-
Expression	-	-	-	2	0	0	50	26
FIFO	-	-	-	-	-	-	-	-
Memory	1	1	-	-	0	0	0	0
Multiplexer	-	-	-	-	-	-	120	38
Register	-	-	-	-	205	51	-	-
Total	1	1	6	2	241	51	202	64

Figure 5-9: **solution1 vs. solution2**

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

- The number of DSP48s has been reduced to only two.
- Since a single DSP48 is being used for each multiplication instead of three, 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 subtly in the reporting. In `solution1` the multiplications were implemented as pipelined multipliers. These are implemented as sub-blocks (or components) in the RTL and thus 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.

Chapter 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:

- Update the code to use bit-accurate types.
- 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`
- Simulate the design and validate the results before synthesis.

Design Optimization

The following optimizations, discussed in an earlier chapter can now be implemented:

- The `Shift_Accum_Loop` loop should be unrolled to reduce latency.
- Array `shift_reg` should be partitioned to prevent a BRAM being used, and allow a shift register to be used.
- The input array `c` should be specified as a single-port RAM in order to guarantee a single-port RAM interface.
- The input port `x` should use 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 will be performed in `solution3`:

- The input array `c` should be specified as a single-port RAM in order to create a single-port RAM interface.
- The input port `x` should use a valid handshake.

Step 1: Creating a New Solution

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

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

   `solution3` is created and automatically opens.

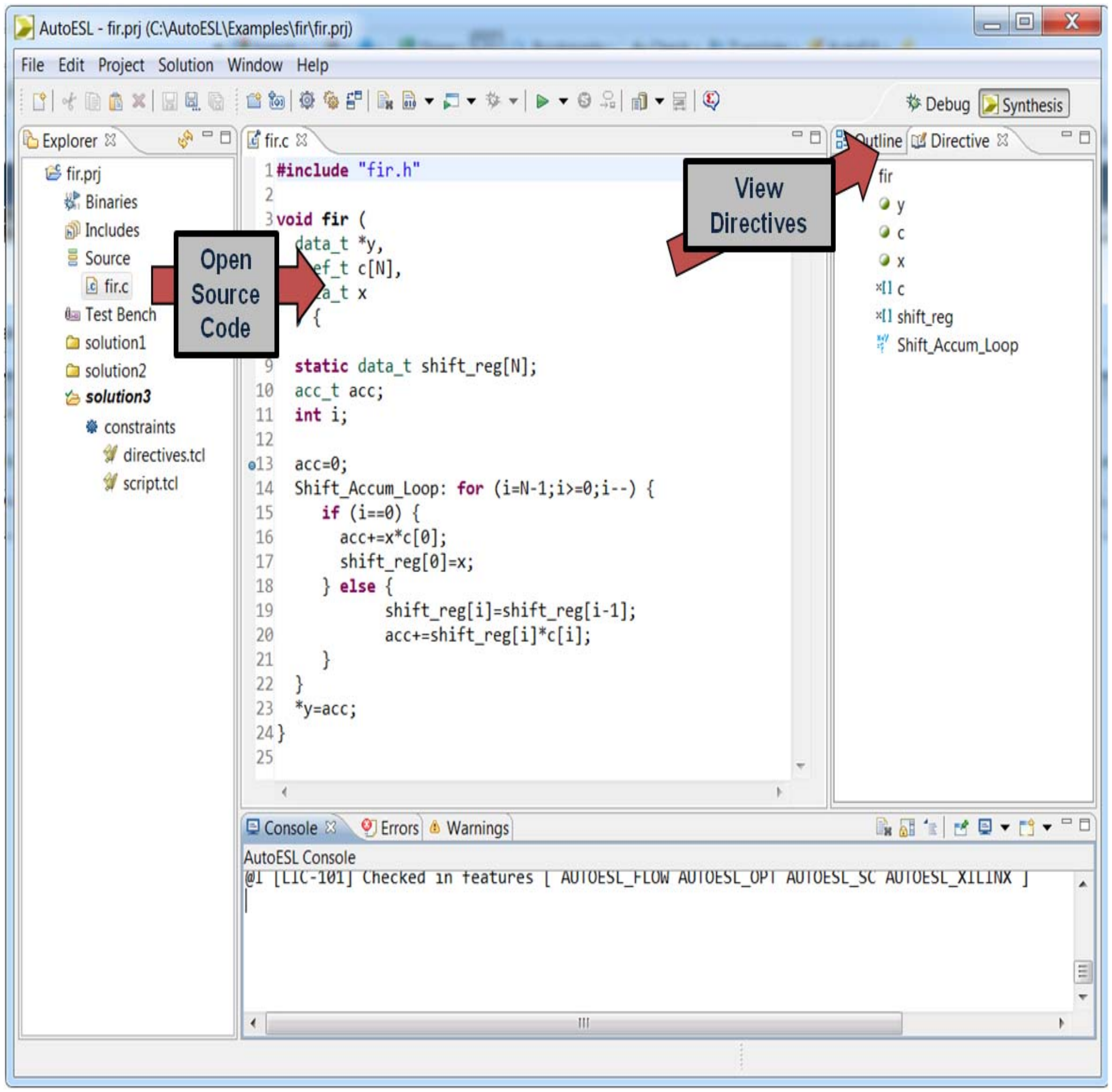
When `solution3` opens, confirm `solution3` is highlighted in bold in the Project Explorer pane, indicating it is the current active solution. Open files use memory so if they are needed keep them open, otherwise close them.

5. 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 6-1).



```
#include "fir.h"

void fir (
  data_t *y,
  coef_t c[N],
  data_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;
}
```

```
@I [LIC-101] Checked in features [ AUTOESL_FLOW AUTOESL_OPT AUTOESL_SC AUTOESL_XILINX ]
```

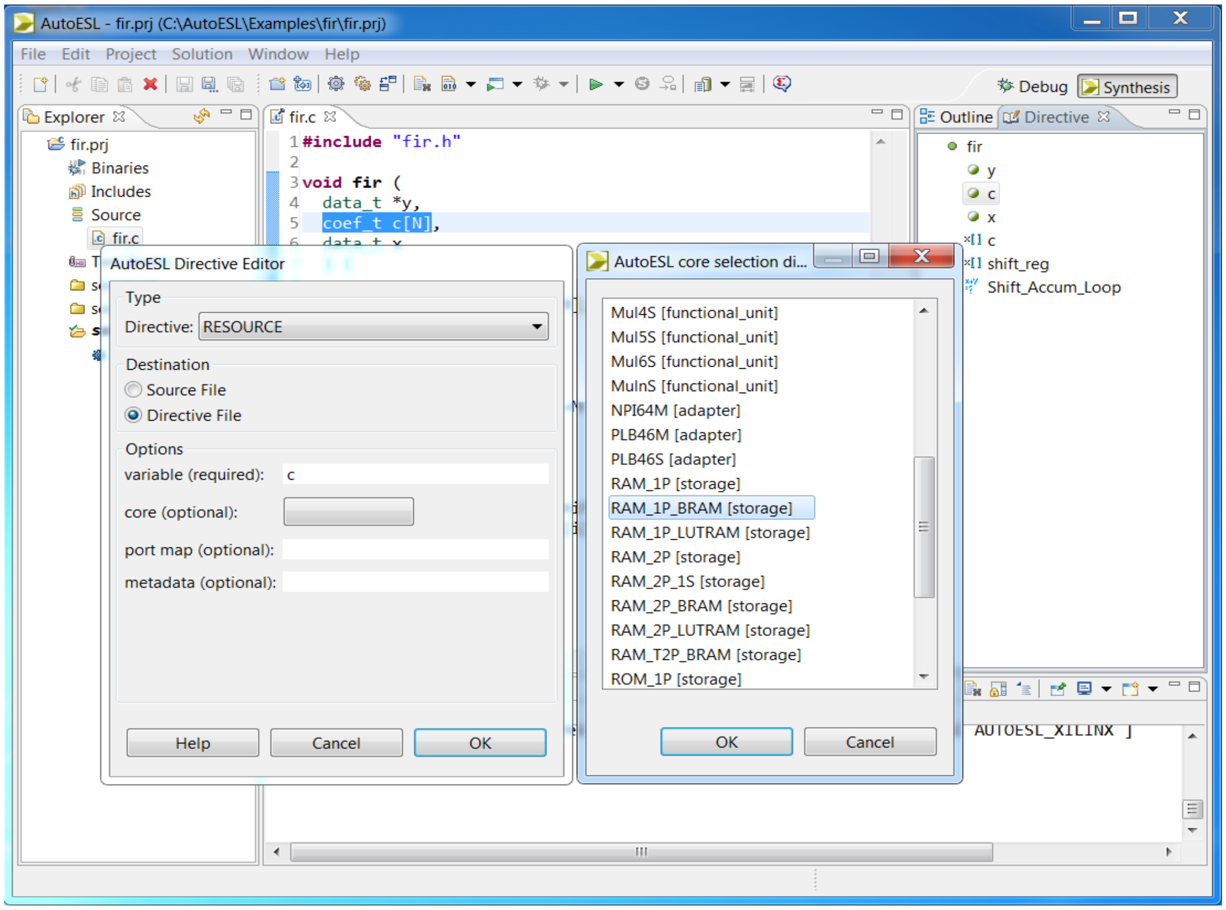
Figure 6-1: **Adding Optimization Directives**

2. Double-click on `fir.c` to open the file in the Information pane.
3. Activate the **Directive Tab** (see Figure 6-1).

The optimization directives can now be applied to the design using the Directive tab.

4. In the Directive tab, select the `c` argument/port (green dot) or the array c.
5. Right-click with the mouse and select **Insert Directives...**
6. Select **RESOURCE** from the Directive drop-down menu, click the **core box** and select **RAM_1P_BRAM**, as shown in Figure 6-2. This ensures 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,
```

Type
Directive: RESOURCE

Destination
Source File [✓ Directive File]

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

Figure 6-2: **Adding a Resource Directive**

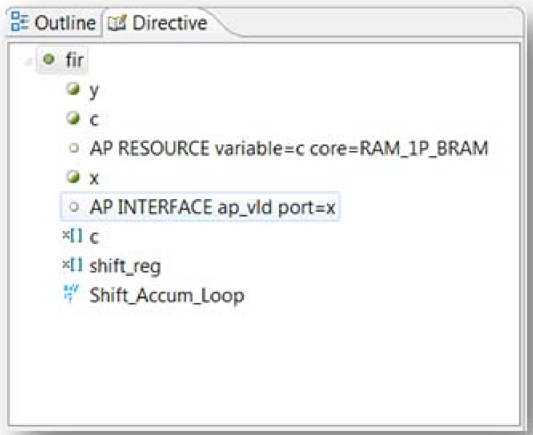
This directive informs the AutoESL tool that array `c` will be implemented as a single-port RAM. Since the array is on the function interface, this is equivalent to the RAM being "off-chip". In this case, the AutoESL tool will create the appropriate interface ports to access it.

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 AutoESL library is provided in the *AutoESL Library Guide (UG870)*.

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

8. In the Directive tab, select input port `x` (green dot).
9. Right-click with the mouse and select **Insert Directives...**
10. Select **Interface** from the Directive drop-down menu and choose **`ap_vld`** for the mode.
11. To apply the directive, click **OK**.

When complete, the Directive pane should be as shown in Figure 6-3. (Select any incorrect directive and use the mouse right-click to modify it).



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

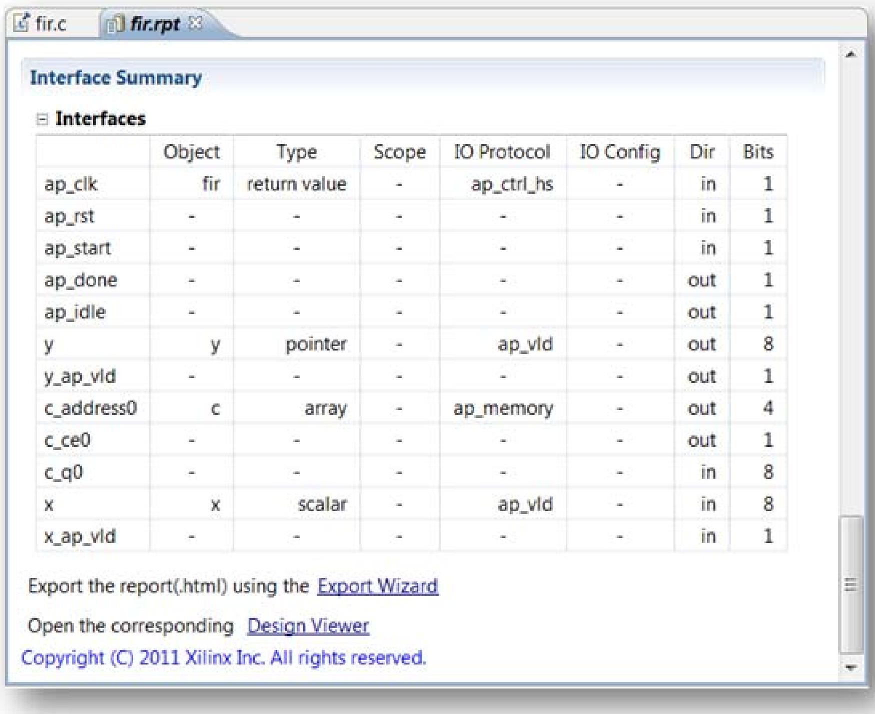
Figure 6-3: **Directive Tab solution3**

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 6-4 shows the interfaces are now correctly defined.



fir.c | fir.rpt

Interface Summary

Interfaces

	Object	Type	Scope	IO Protocol	IO Config	Dir	Bits
ap_clk	fir	return value	-	ap_ctrl_hs	-	in	1
ap_rst	-	-	-	-	-	in	1
ap_start	-	-	-	-	-	in	1
ap_done	-	-	-	-	-	out	1
ap_idle	-	-	-	-	-	out	1
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

Copyright (C) 2011 Xilinx Inc. All rights reserved.

Figure 6-4: **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 6-5.

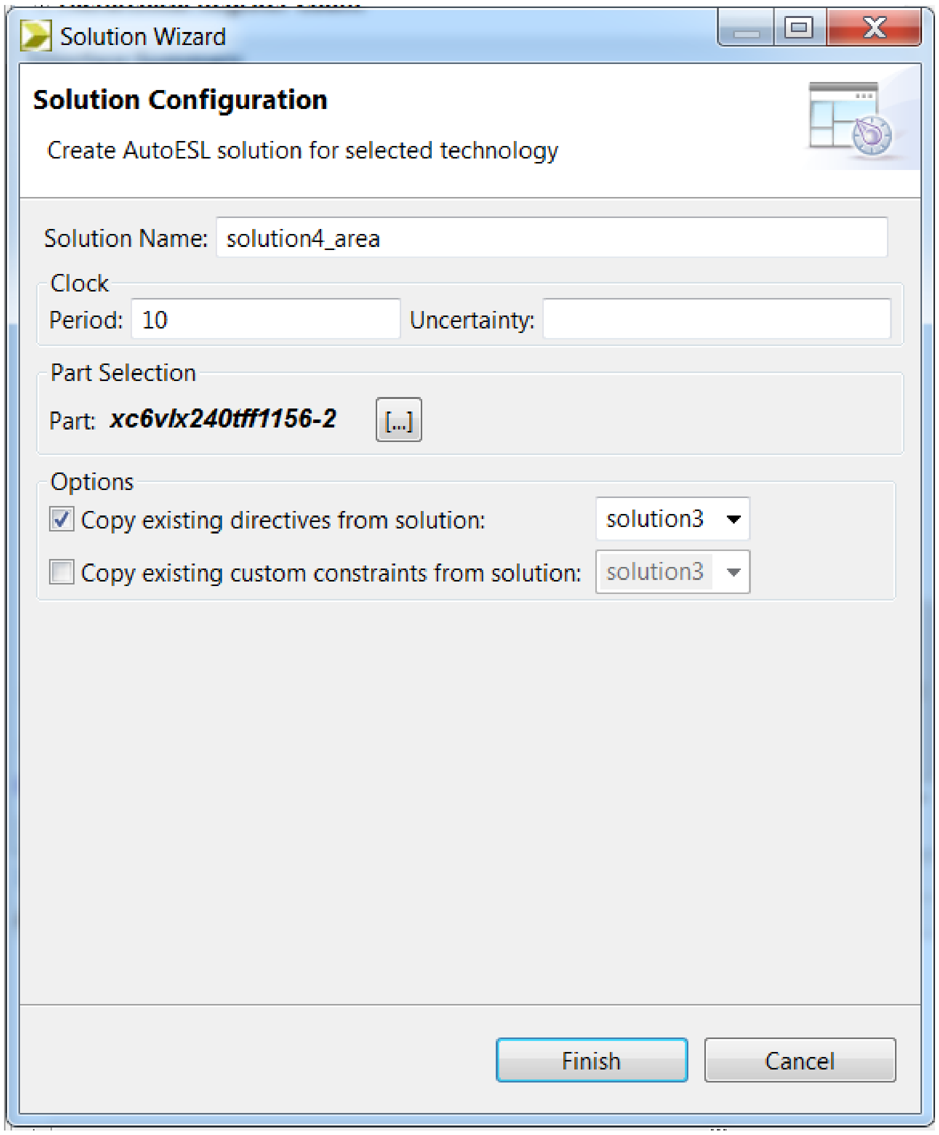
Step 1: Creating a New Solution

1. Click the **New Solution** toolbar to create a new solution.
2. Name the solution, **`solution4_area`**. The solution names will default to solution1, 2, 3 etc. but can be named anything.
3. Select the **Copy existing directives from solution:** box and select **`solution3`** from the menu.

   This copies the IO directives specified in `solution3` into `solution4_area`.

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

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



Solution Wizard

Solution Configuration

Create AutoESL solution for selected technology

Solution Name: solution4_area

Clock

Period: 10 Uncertainty:

Part Selection

Part: ***xc6vlx240tff1156-2***

Options

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

Figure 6-5: **Create solution4_area**

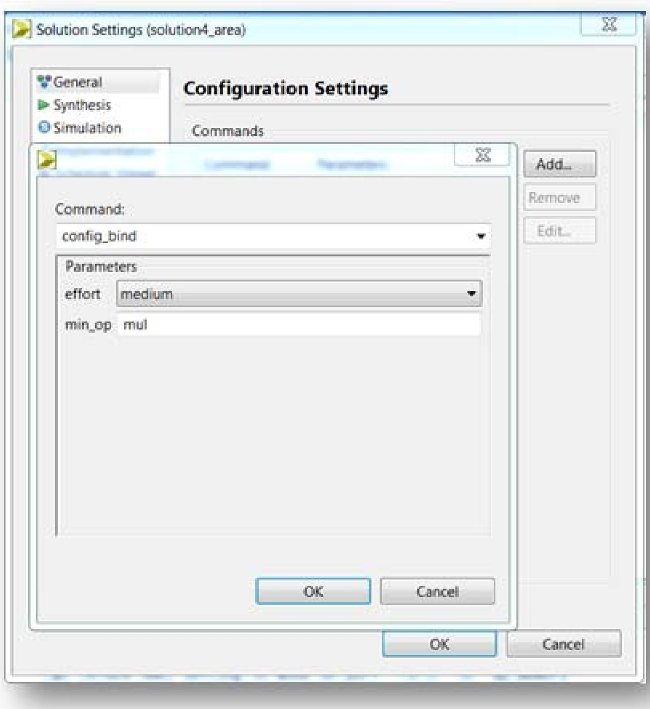
Step 2: Sharing of Multipliers

To force sharing of the multipliers, a configuration setting will be used.

1. Open the solution settings using the menu **Solution/Solution Settings...**

2. Select **General** on the left-hand side menu and click the **Add...** button to open the list of configurations.
3. Select **`config_bind`** from the drop-down menu and specify **`mul`** in the **min_op** (minimize operator) field, as shown in Figure 6-6.
4. Click **OK** to set the configuration, and 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 AutoESL to minimize the number of the specified operators (`mul` operations in this case) and overrides any `mux` delay estimation.



Command: config_bind

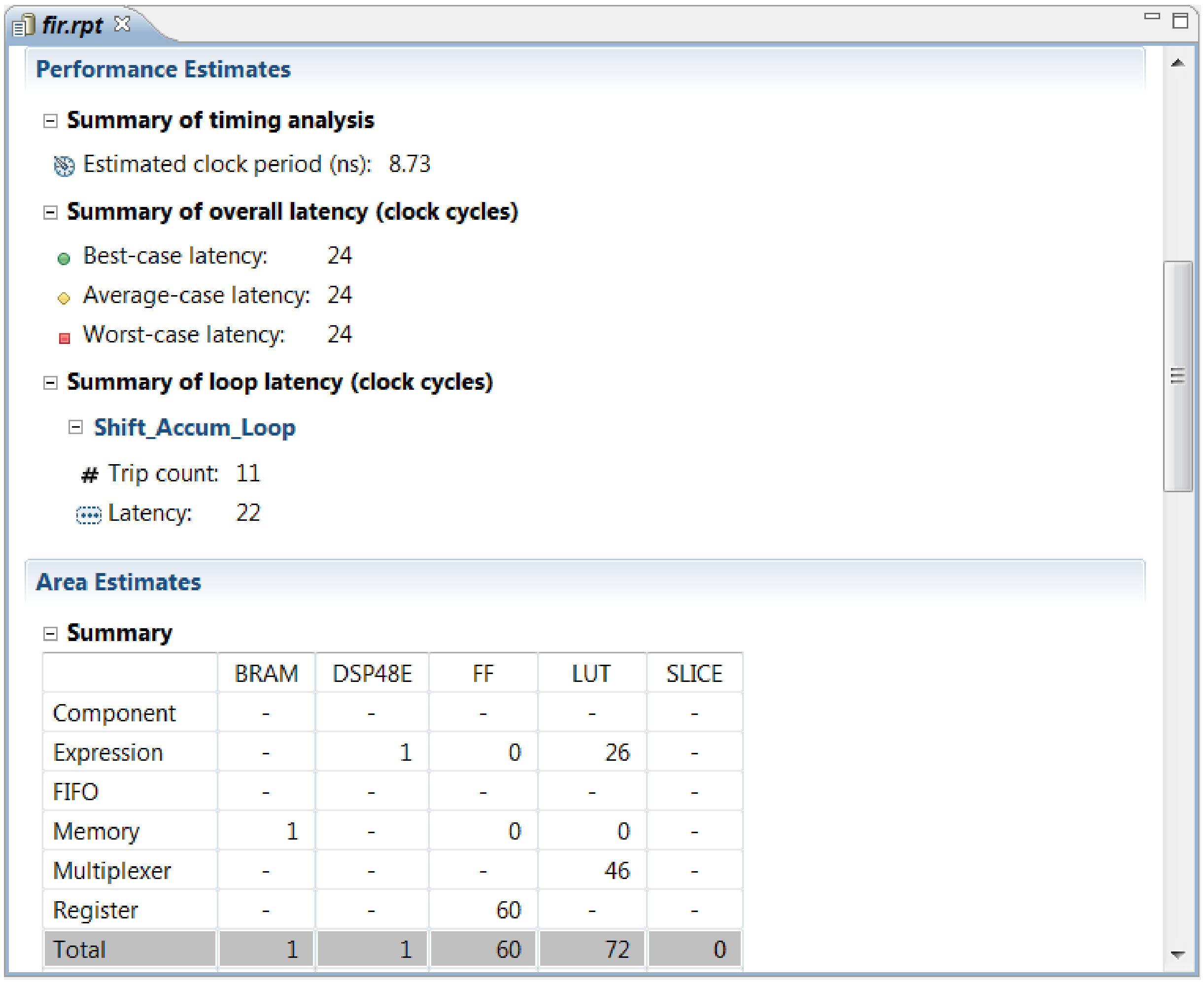
Parameters
effort: medium
min_op: mul

Figure 6-6: **Adding Custom Constraints**

Step 3: Synthesis

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

When synthesis completes, the synthesis report opens showing the configuration command was successful and only a single multiplier is now used in the design (see Figure 6-7).



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	-	1	0	26	-
FIFO	-	-	-	-	-
Memory	1	-	0	0	-
Multiplexer	-	-	-	46	-
Register	-	-	60	-	-
Total	1	1	60	72	0

Figure 6-7: **Auxiliary Pane Directives Tab**

This design is using the same hardware resources to implement every iteration of the loop. This is the smallest number of resources 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, (unrolling the loop and partitioning the memory), `solution3`, with the correct IO interface, will be used as the starting point.

Step 1: Creating a New Solution

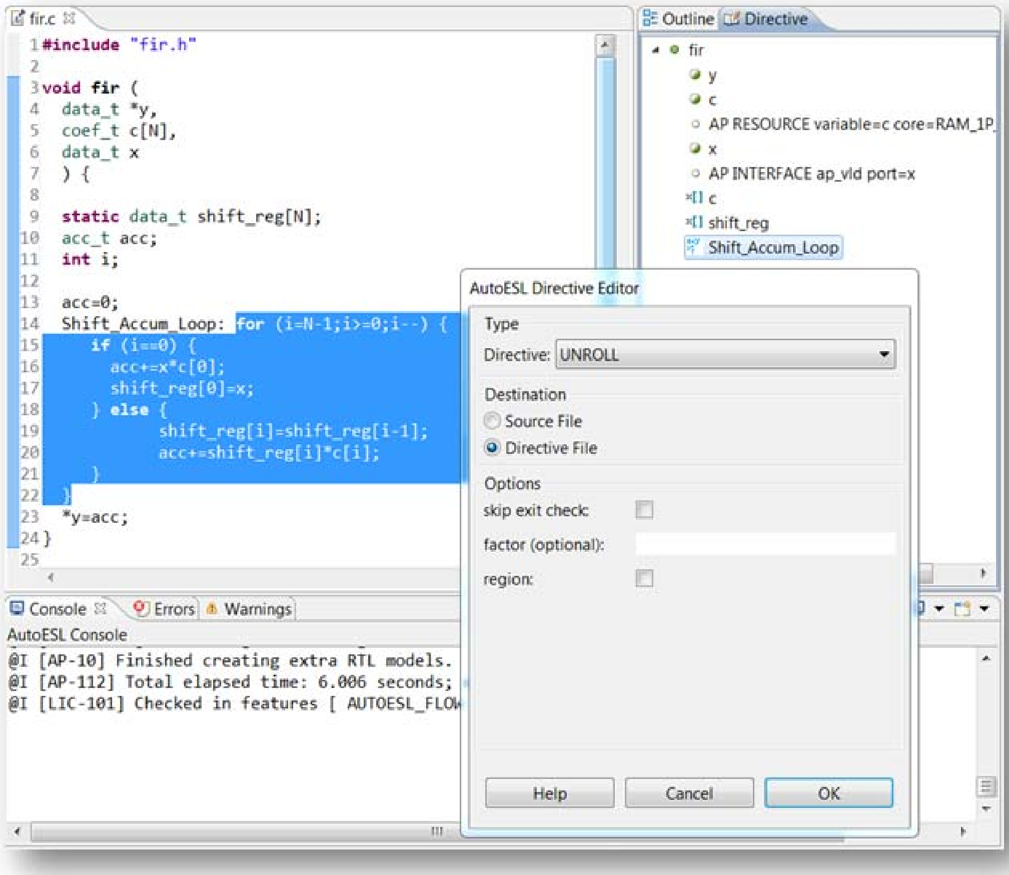
Begin by creating a new solution.

1. Click the **New Solution** toolbar to create a new solution.
2. Name the solution `solution5_throughput`.
3. Select the **Copy existing directives** from solution box and select **`solution3`** (NOT `solution4_area`) from the drop-down menu.

   This will copy the IO directives specified in `solution3` into `solution5_throughput`.
4. Close any existing tabs from previous solutions. In the Project menu, select **Close Inactive Solution Tabs**.

Step 2: Unrolling the Loop

The following steps (summarized in Figure 6-8) explain how to unroll the loop.



```
#include "fir.h"

void fir (
  data_t *y,
  coef_t c[N],
  data_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;
}
```

```
AutoESL Console
@I [AP-10] Finished creating extra RTL models.
@I [AP-112] Total elapsed time: 6.006 seconds;
@I [LIC-101] Checked in features [ AUTOESL_FLOW
```

```
fir
  y
  c
    AP RESOURCE variable=c core=RAM_1P
  x
    AP INTERFACE ap_vld port=x
  c
  shift_reg
  Shift_Accum_Loop
```

AutoESL Directive Editor
Type
Directive: UNROLL
Destination
Source File
[✓ Directive File]
Options
skip exit check:
factor (optional):
region:

Figure 6-8: **Unrolling FOR Loop**

1. In the Directive tab, select loop **`Shift_Accum_Loop`**. (Open the source code to see the Directive tab).

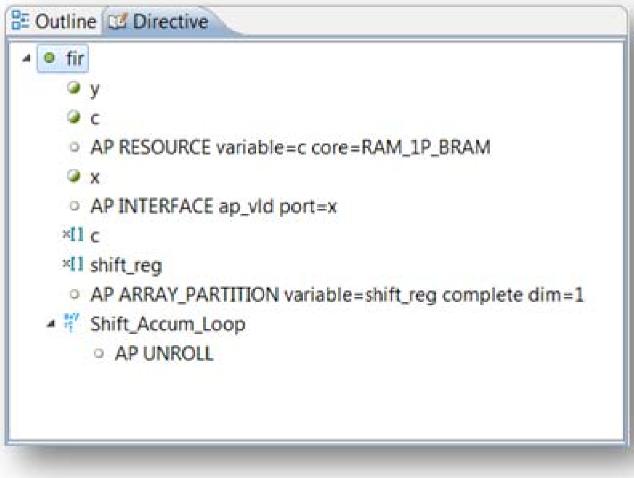
2. Right-click with the mouse and select **Insert Directives...**.
3. Select **Unroll** from the Directive drop-down menu.
4. Select **OK** to apply the directive.

Leaving the other options in the Directives window unchecked and blank ensures the loop will be fully unrolled.

Similarly, apply the directive to partition the array into individual elements, which will then be arranged as a shift-register.

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



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

Figure 6-9: **solution5_throughout Directives**

Step 3: Synthesis

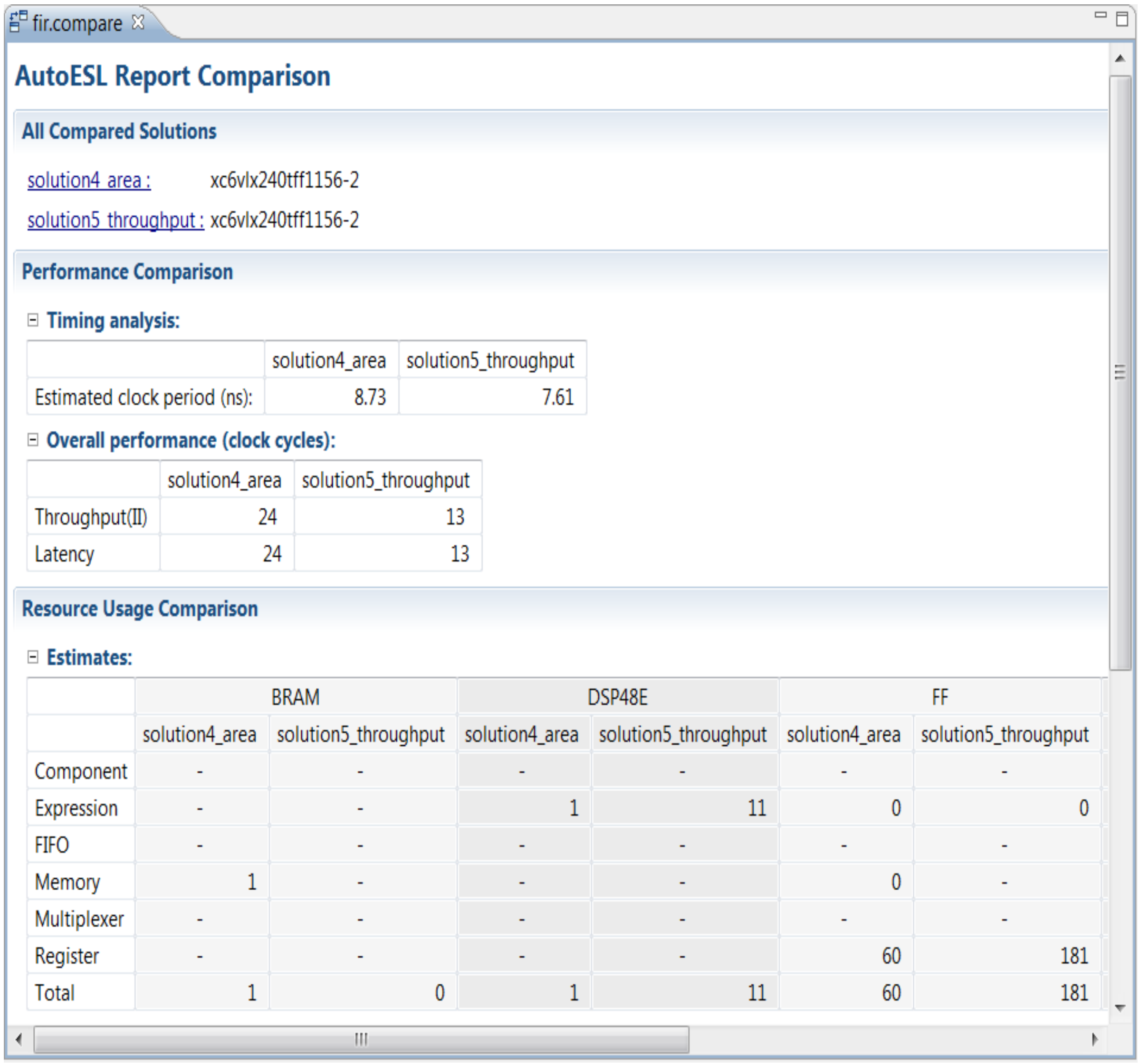
1. Click the **Synthesis** toolbar button to synthesize the design.

   When synthesis completes, the synthesis report automatically opens.

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

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

   Figure 6-10 shows the comparison of the reports from `solution4_area` and `solution5` (the LUTS are not shown in Figure 6-10 due to the wide nature of the report).



AutoESL Report Comparison

All Compared Solutions

solution4_area : xc6vlx240tff1156-2

solution5_throughput : xc6vlx240tff1156-2

Performance Comparison

Timing analysis:

	solution4_area	solution5_throughput
Estimated clock period (ns):	8.73	7.61

Overall performance (clock cycles):

	solution4_area	solution5_throughput
Throughput(II)	24	13
Latency	24	13

Resource Usage Comparison

Estimates:

	BRAM		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	181
Total	1	0	1	11	60	181

Figure 6-10: **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 does however take 24 clock cycles to complete.

The high throughput design is processing 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 over twice the number of flip-flops as the small design.

Scrolling down the report window will show 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 `solution4_area` will use much less power than `solution5_throughput` and the increase is caused by both additional registers and expressions (logic).

Chapter Summary

- Optimization directives can be added 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 Implementation

The AutoESL tool allows both RTL verification and implementation to be performed from the GUI. The verification and implementation 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 implementation are `autosim` and `autoimpl`, 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 AutoESL tool.

For RTL simulation, the AutoESL 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, (including a C co-simulation license since the original C test bench is simulated) 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 7-1.

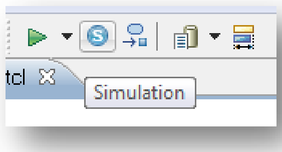
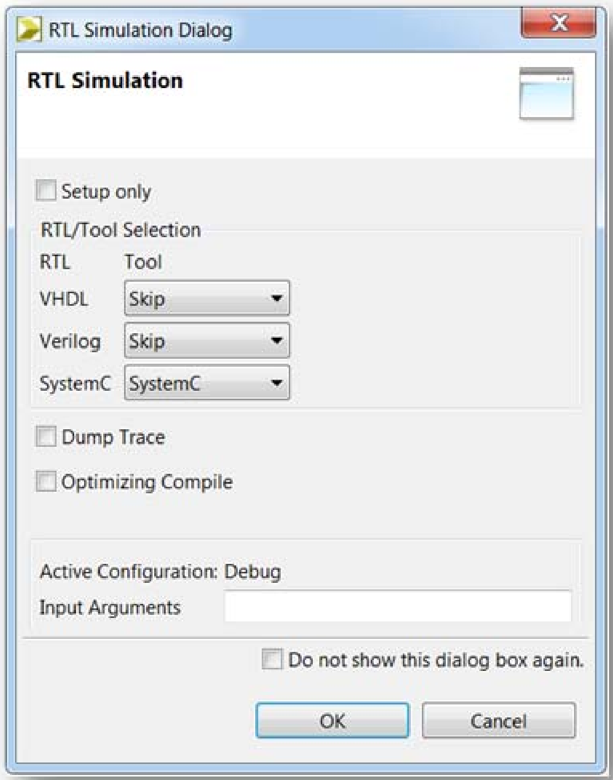


Figure 7-1: **Simulation Toolbar Button**

The co-simulation dialog opens, as shown in Figure 7-2.



RTL Simulation Dialog

RTL Simulation

Setup only

RTL/Tool Selection

RTL	Tool
VHDL	Skip
Verilog	Skip
SystemC	SystemC

Dump Trace

Optimizing Compile

Active Configuration: Debug

Input Arguments

Do not show this dialog box again.

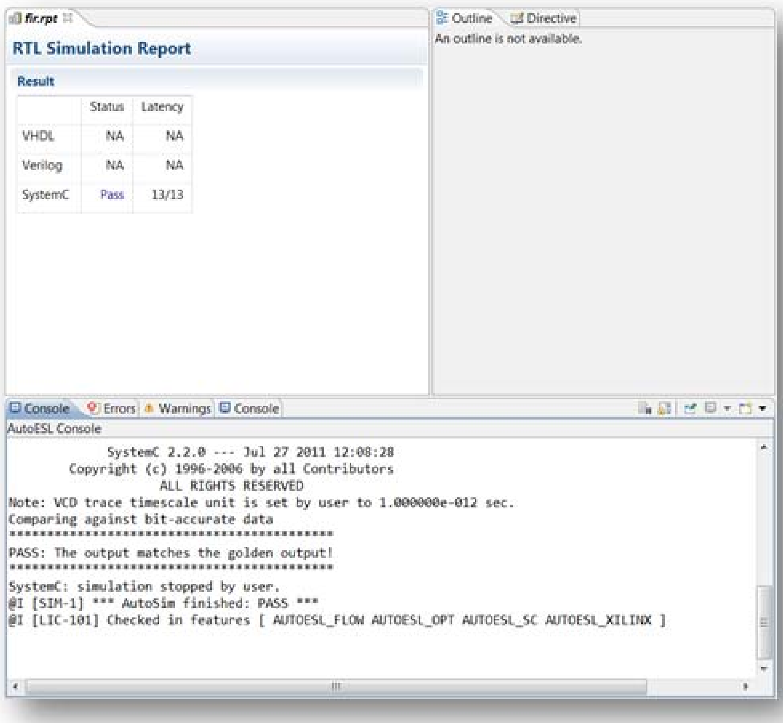
Figure 7-2: **RTL Verification Menu**

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 7-3). 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.



RTL Simulation Report

Result

	Status	Latency
VHDL	NA	NA
Verilog	NA	NA
SystemC	Pass	13/13

```
                SystemC 2.2.0 --- Jul 27 2011 12:08:28
          Copyright (c) 1996-2006 by all Contributors
                       ALL RIGHTS RESERVED
Note: VCD trace timescale unit is set by user to 1.000000e-012 sec.
Comparing against bit-accurate data
*******************************************
PASS: The output matches the golden output!
*******************************************
SystemC: simulation stopped by user.
@I [SIM-1] *** AutoSim finished: PASS ***
@I [LIC-101] Checked in features [ AUTOESL_FLOW AUTOESL_OPT AUTOESL_SC AUTOESL_XILINX ]
```

Figure 7-3: **Simulation Report**

RTL Implementation

The final step in the AutoESL flow is to pass the RTL design into logic synthesis for implementation.

To use RTL logic synthesis tools, the executable should be available in the search path. Refer to the *AutoESL Installation Guide (UG869)* for details.

1. Click the **Implementation** button in the toolbar, as shown in Figure 7-4.

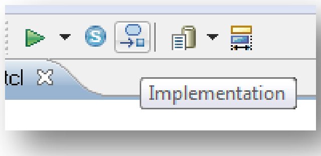
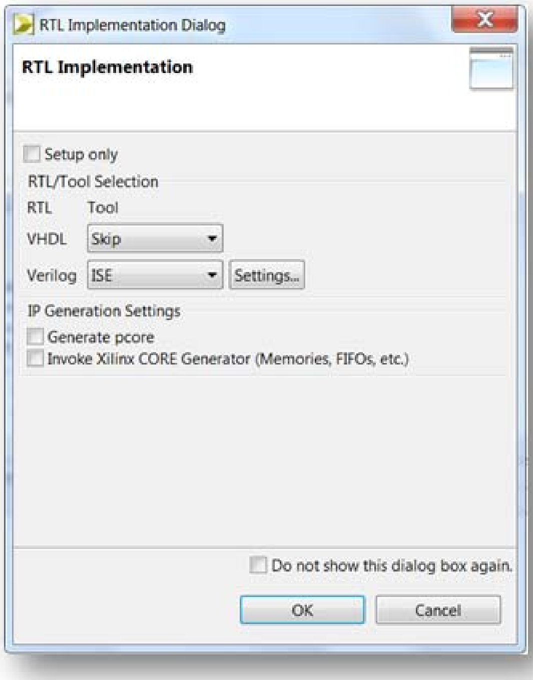


Figure 7-4: **Implementation Toolbar Button**

The implementation dialog opens, as shown in Figure 7-5.



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.

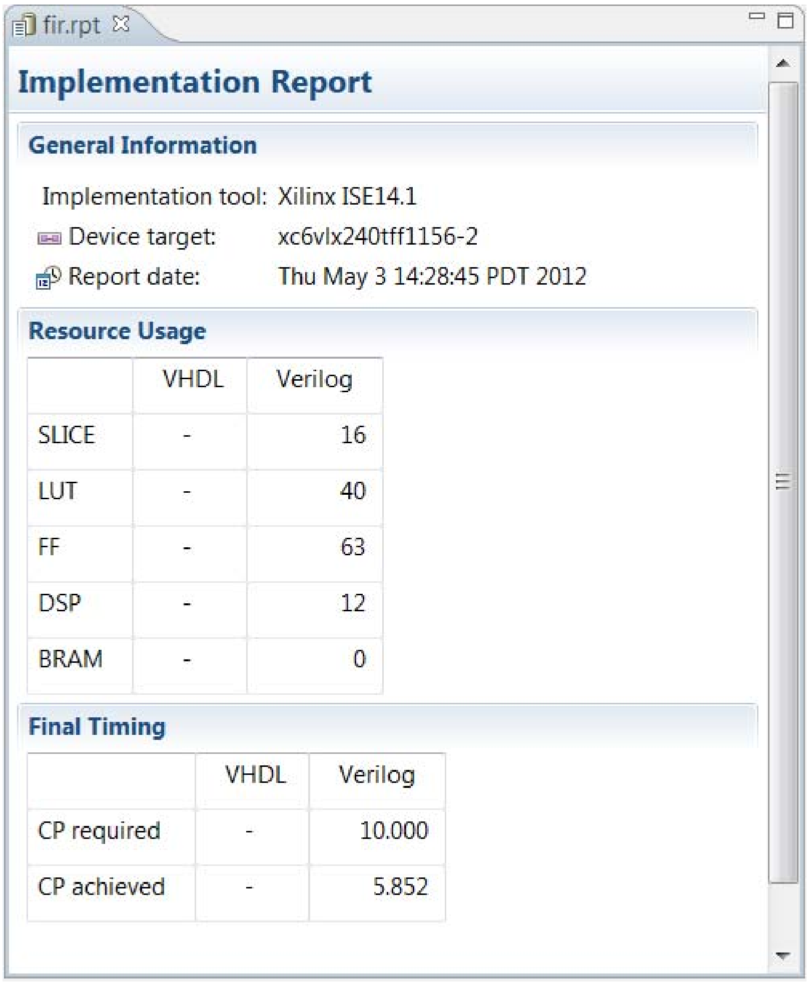
Figure 7-5: **RTL Implementation Menu**

2. For VHDL or Verilog, select **ISE** from the drop-down menu. In this example, Verilog is used, as shown in Figure 7-5.
3. Click **OK**.

   Implementation starts.

If the Setup Only option is used, the files are created in directory `fir.prj/solution5_throughout/impl/verilog` but the ISE Design Suite does not run.

When implementation completes, the implementation report automatically opens (see Figure 7-6).



fir.rpt

Implementation Report

General Information

Implementation tool: Xilinx ISE14.1

Device target: xc6vlx240tff1156-2

Report date: Thu May 3 14:28:45 PDT 2012

Resource Usage

	VHDL	Verilog
SLICE	-	16
LUT	-	40
FF	-	63
DSP	-	12
BRAM	-	0

Final Timing

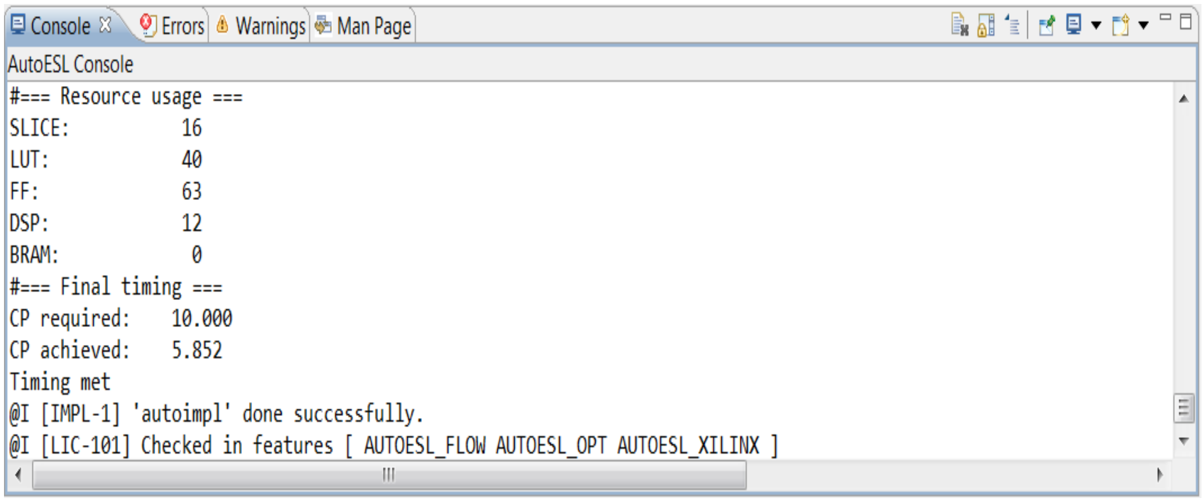
	VHDL	Verilog
CP required	-	10.000
CP achieved	-	5.852

Figure 7-6: **solution5_throughput Report**

The report shows the design is meeting timing but using more DSP48s than predicated by the AutoESL tool. The AutoESL tool will estimate the DSP48 usage based on the number of multiplications required. Logic synthesis will perform its own optimization and may produce a design which uses more or less DSP48s. In some cases, logic synthesis may implement some logic operations, increasing the number of DSP48s and reducing the number of LUTs. Logic synthesis may also be able to decompose and reduce the number of multiplications, thereby reducing the number of DSP48s.

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

Additionally, the results can be seen in the Console, as shown in Figure 7-7.



```
#=== Resource usage ===
SLICE:          16
LUT:            40
FF:             63
DSP:            12
BRAM:            0
#=== Final timing ===
CP required:    10.000
CP achieved:    5.852
Timing met
@I [IMPL-1] 'autoimpl' done successfully.
@I [LIC-101] Checked in features [ AUTOESL_FLOW AUTOESL_OPT AUTOESL_XILINX ]
```

Figure 7-7: **Implementation Summary**

4. Exit the AutoESL 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.

Chapter Summary

- The path to verification and implementation tool executables must be in the search path prior to execution from within the AutoESL tool. Refer to the *AutoESL Installation Guide (UG869)* 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 AutoESL tool using the existing C test bench.
- The design can be implemented using standard FPGA logic synthesis tools from within the AutoESL tool.

The Shell and Scripts

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

AutoESL at the Shell

The AutoESL GUI can be invoked at the Linux or DOS shell prompt.

Invoke a DOS shell from the menu by selecting **Start > All Programs > AutoESL<version> > AutoESL Command Prompt** to ensure the search paths for the AutoESL tool are already defined in the shell.

```
$ autoesl
```

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

```
$ autoesl –i
autoesl> exit
$
```

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

```
$ autoesl -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:

```
$ autoesl -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 (this is the same `script.tcl` file edited in Figure 6-6 when adding custom constraints).

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 AutoESL.
## 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 AutoESL.
## 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 which has five scripts, used to create each of the five solutions in this tutorial.

Table 8-1: **Summary of Scripts**

Filename	Solution	Description
`run1_aesl.tcl`	`solution1`	Creates the first solution, using standard implementation types.
`run2_aesl.tcl`	`solution2`	Sets the macro to use AutoESL bit-accurate types.
`run3_aesl.tcl`	`solution3`	The IO interfaces are defined.
`run4_aesl.tcl`	`solution4_area`	Uses the directives from solution3 plus the `config_bind` command to force sharing of the multipliers.
`run5_aesl.tcl`	`solution5_throughput`	Optimizations are applied to create a high-throughput version.

These scripts can be run to reproduce all the solutions in this tutorial. The project and solutions can then be opened in the GUI and analyzed.

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