Basic HLS Tutorial

using C++ language and Vivado Design Suite to design two frequencies PWM modulator system

January 25, 2017

Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

1.1 Motivation

Basic HLS Tutorial is a document made for beginners who are entering the world of embedded system design using FPGAs. This tutorial explains, step by step, the procedure of designing a simple digital system using C/C++/SystemC languages and Xilinx Vivado Design Suite.

1.2 Purpose of this Tutorial

This tutorial is made to introduce you how to **create**, **simulate** and **test** an project and run it on your development board.

After completing this tutorial, you will be able to:

- Launch and navigate the Vivado High-Level Synthesis (HLS) tool
- Create a project using New Project Creation Wizard
- Develop a C algorithm for your design
- Verify a C algorithm of your design
- Synthesize a C algorithm into an RTL implementation (High-Level Synthesis)
- Generate reports and analyze the design
- Verify the RTL implementation
- Package the RTL implementation

The following project is designed for:

- Designing Surface: **VIVADO 2016.4**
- HD Language: **C**
- Device: **ZedBoard Zynq Evaluation and Development Kit**

1.3 Objectives of this Tutorial

In this tutorial a **PWM** signal modulated using the sine wave with two **different frequencies** (1 Hz and 3.5 Hz) will be created. Frequency that will be chosen depends on the position of the two-state on-board switch (sw0).

PWM Signal

Pulse-width modulation (PWM) uses a rectangular pulse wave whose pulse width is modulated by some other signal (in our case we will use a sine wave) resulting in the variation of the average value of the waveform. Typically, PWM signals

are used to either convey information over a communications channel or control the amount of power sent to a load. To learn more about PWM signals, please visit http://en.wikipedia.org/wiki/Pulse-width_modulation.

Figure 1.1. illustrates the principle of pulse-width modulation. In this picture an arbitrary signal is used to modulate the PWM signal, but in our case sine wave signal will be used.

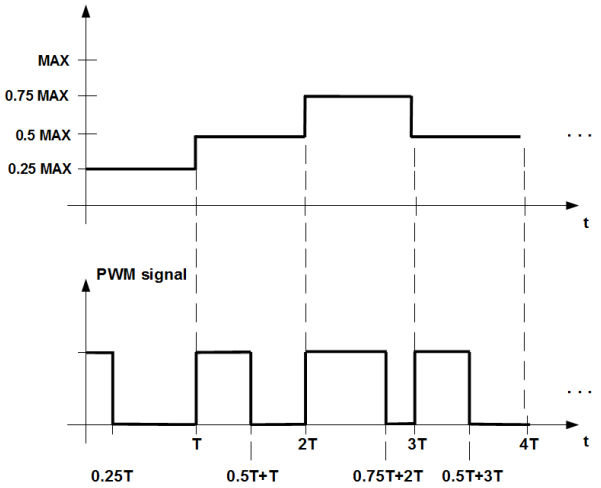


Figure 1.1: Example of the PWM signal

1.4 One Possible Solution for the Modulator Design

Considering that we are working with digital systems and signals, our task will be to generate an digital representation of an analog (sine) signal with two frequencies: 1 Hz and 3.5 Hz.

Figure 1.2 is showing the sine wave that will be used to modulate the PWM signal.

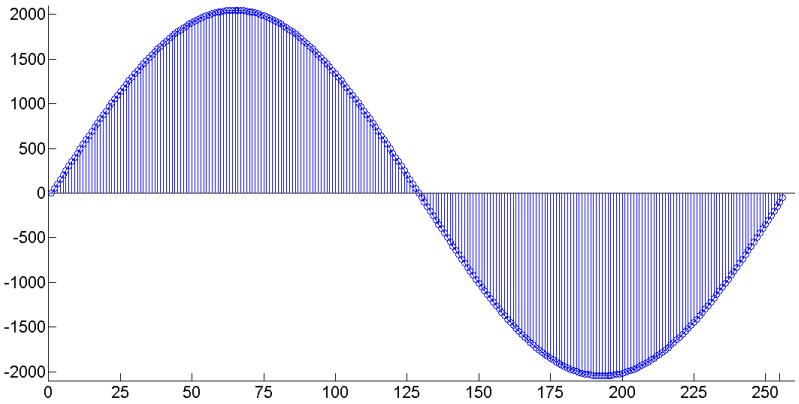


Figure 1.2: Sine wave with 256 samples

One period of the sine wave is represented with 256 (2^8) samples, where each sample can take one of 4096 (2^12) possible values. Since the sine wave is a periodic signal, we only need to store samples of one period of the signal.

Note: Pay attention that all of sine signals with the same amplitude, regardless their frequency, look the same during the one period of a signal. The only thing that is different between those sine signals is duration of a signal period. This means that the sample rate of those signals is different.

Considering that the whole system will be clocked with the 100 MHz input signal, which is available on the target development board, to get 1 Hz and 3.5 Hz frequencies (which is much smaller than 100 MHz) we should divide input clock frequency with integer value N.

In the Tables 1.1 and 1.2 are shown parameters that are necessary for generating sine signals with 1 Hz and 3.5 Hz frequencies.

Table 1.1: Sine signal with the frequency of 1 Hz

Division Factor Steps	Calculation	Explanation
T=1 s	T=1/1 Hz=1 s	T is the period of the signal
f1=256	f1=256∗1 Hz=256 Hz (or read in time: 1 s/256)	f1 is the frequency of reading whole period (T) with 256 samples
N1=390625	N1=100 MHz/256 Hz=390625	N1 is the number which divides frequency of the input clock signal (100 MHz) to the required frequency for the digital sine module
N2=95	N2=390625/4096=95.3674	N2 is the number which divides frequency of the input clock signal (100 MHz) to the required frequency for the PWM's FSM module
N1=389120	N1=95∗4096=389120	This is new calculation, because N1 must be divisible with 4096

Table 1.2: Sine signal with the frequency of 3.5 Hz

Division Factor Steps	Calculation	Explanation
T=0.286 s	T=1/3.5 Hz=0.286 s	T is the period of the signal
f2=896 Hz	f2=256∗3.5 Hz=896 Hz (or read in time: 0.286 s/256)	f2 is the frequency of reading whole period (T)
N1=111607.1429	N1=100 MHz/896 Hz=111607.1428571	N1 is the number which divides frequency of the input clock signal (100 MHz) to the required frequency
N2=27	N2=111607.1428571/4096=27.2478	N2 is the number which divides frequency of the input clock signal (100 MHz) to the required frequency for the PWM's FSM module
N1=110592	N1=27∗4096=110592	This is new calculation, because N1 must be divisible with 4096

Now, it is obvious that the sine wave can be generated by reading sample values of one period, that are stored in one table, with appropriate speed. In our case the values will be generated using the sine function from the C numerics library (math.h) and will be stored in an array.

1.5 About HLS

The Xilinx Vivado High-Level Synthesis (HLS) is a tool that transforms a C specification into a register transfer level (RTL) implementation that you can synthesize into a Xilinx field programmable gate array (FPGA).

You can write C specifications in C, C++, SystemC, or as an Open Computing Language (OpenCL) API C kernel, and the FPGA provides a massively parallel architecture with benefits in performance, cost, and power over traditional processors.

By targeting an FPGA as the execution fabric, HLS enables a software engineer to optimize code for throughout, power, and latency without the need to address the performance bottleneck of a single memory space and limited computational resources. This allows the implementation of computationally intensive software algorithms into actual products, not just functionality demonstrators.

High-level synthesis bridges hardware and software domains, providing the following primary benefits:

- Improved productivity for hardware designers

  Hardware designers can work at a higher level of abstraction while creating high-performance hardware.

- Improved system performance for software designers

  Software developers can accelerate the computationally intensive parts of their algorithms on a new compilation target, the FPGA.

Using a high-level synthesis design methodology allows you to:

- Develop algorithms at the C-level

  Work at a level that is abstract from the implementation details, which consume development time.

- Verify at the C-level

  Validate the functional correctness of the design more quickly than with traditional hardware description languages.

- Control the C synthesis process through optimization directives

  Create specific high-performance hardware implementations.

- Create multiple implementations from the C source code using optimization directives

  Explore the design space, which increases the likelihood of finding an optimal implementation.

- Create readable and portable C source code

  Retarget the C source into different devices as well as incorporate the C source into new projects.

HLS Phases

High-level synthesis includes the following phases:

- ***Scheduling***

  Determines which operations occur during each clock cycle based on:

  – Length of the clock cycle or clock frequency
  – Time it takes for the operation to complete, as defined by the target device
  – User-specified optimization directives

  If the clock period is longer or a faster FPGA is targeted, more operations are completed within a single clock cycle, and all operations might complete in one clock cycle. Conversely, if the clock period is shorter or a slower FPGA is targeted, high-level synthesis automatically schedules the operations over more clock cycles, and some operations might need to be implemented as multicycle resources.

- ***Binding***

  Determines which hardware resource implements each scheduled operation. To implement the optimal solution, high-level synthesis uses information about the target device.

- ***Control logic extraction***

  Extracts the control logic to create a finite state machine (FSM) that sequences the operations in the RTL design.

1.6 Design Steps

This tutorial will be realized step by step with the idea to explain the whole procedure of designing an digital system, using Vivado HLS tool.

- First, we will develop algorithm at the C-level.

  Work at a level that is abstract from the implementation details, which consume development time.

- Then we will verify the algorithm at the C-level.

  Validate the functional correctness of the design more quickly than with traditional hardware description languages.

- After that, we will synthesize the C algorithm into an RTL implementation.

  Using Vivado HLS tool we will automatically create an RTL implementation of our C algorithm. Vivado HLS will automatically create data path and control path modules required to implement our algorithm in hardware.

- Then, we will generate comprehensive reports and analyze the design.

  After synthesis, Vivado HLS automatically creates synthesis reports to help you understand the performance of the implementation.

- Then, we verify the RTL implementation.

  You can use it to verify that the RTL is functionally identical to the original C code.

- At the end, package the RTL implementation into a selection of IP formats.

Using Vivado HLS, you can export the RTL and package the final RTL output files as IP.

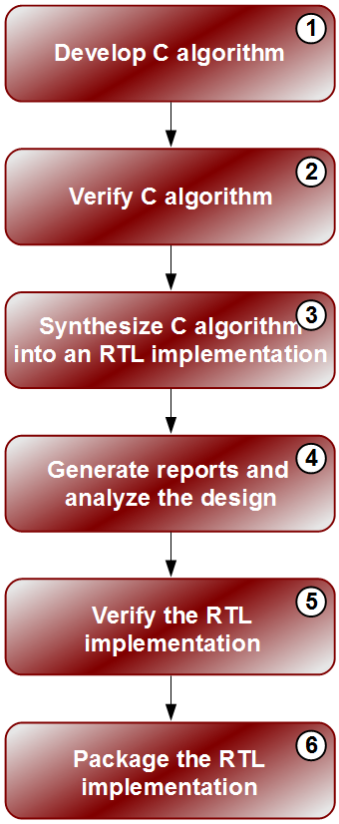


Figure 1.3: Design Steps

1.7 Vivado HLS Design Flow

The Xilinx Vivado HLS tool synthesizes a C function into an IP block that you can integrate into a hardware system. It is tightly integrated with the rest of the Xilinx design tools and provides comprehensive language support and features for creating the optimal implementation for your C algorithm.

The following Figure shows an overview of the Vivado HLS design flow.

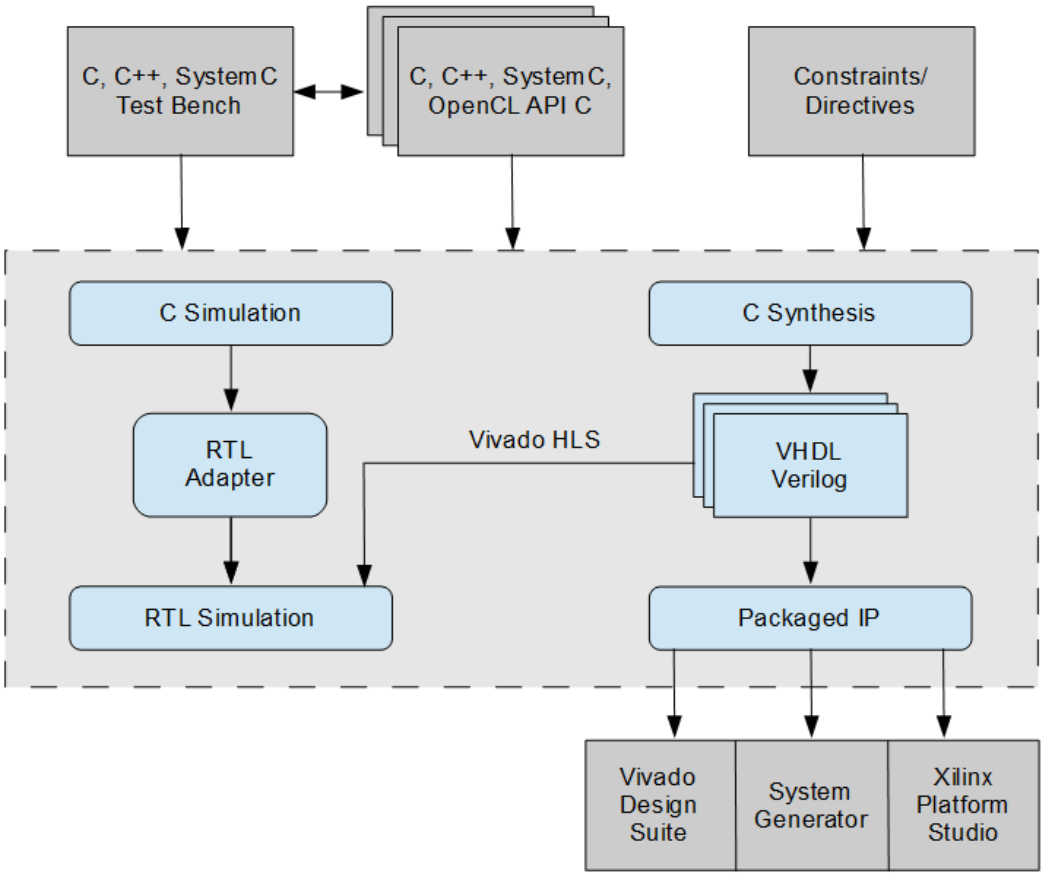


Figure 1.4: Vivado HLS Design Flow

Inputs and Outputs

Following are the **inputs** to Vivado HLS:

- C function written in C, C++, SystemC, or an OpenCL API C kernel

  This is the primary input to Vivado HLS. The function can contain a hierarchy of sub-functions.

- Constraints

  Constraints are required and include the clock period, clock uncertainty, and FPGA target. The clock uncertainty defaults to 12.5% of the clock period if not specified.

- Directives

  Directives are optional and direct the synthesis process to implement a specific behavior or optimization.

- C test bench and any associated files

  Vivado HLS uses the C test bench to simulate the C function prior to synthesis and to verify the RTL output using C/RTL Cosimulation.

You can add the C input files, directives, and constraints to a Vivado HLS project interactively using the Vivado HLS graphical user interface (GUI) or using Tcl commands at the command prompt. You can also create a Tcl file and execute the commands in batch mode.

Following are the **outputs** from Vivado HLS:

- RTL implementation files in hardware description language (HDL) formats

  This is the primary output from Vivado HLS. Using Vivado synthesis, you can synthesize the RTL into a gate-level implementation and an FPGA bitstream file. The RTL is available in the following industry standard formats:

  – VHDL (IEEE 1076-2000)

  - Verilog (IEEE 1364-2001)

  Vivado HLS packages the implementation files as an IP block for use with other tools in the Xilinx design flow. Using logic synthesis, you can synthesize the packaged IP into an FPGA bitstream.

- Report files

  This output is the result of synthesis, C/RTL co-simulation, and IP packaging.

Test Bench, Language Support, and C Libraries

In any C program, the top-level function is called *main()*. In the Vivado HLS design flow, you can specify any sub-function below main() as the top-level function for synthesis. You cannot synthesize the top-level function main(). Following are additional rules:

- Only one function is allowed as the top-level function for synthesis.
- Any sub-functions in the hierarchy under the top-level function for synthesis are also synthesized.
- If you want to synthesize functions that are not in the hierarchy under the top-level function for synthesis, you must merge the functions into a single top-level function for synthesis.
- The verification flow for OpenCL API C kernels requires special handling in the Vivado HLS flow.

Test Bench

When using the Vivado HLS design flow, it is time consuming to synthesize a functionally incorrect C function and then analyze the implementation details to determine why the function does not perform as expected. To improve productivity, use a test bench to validate that the C function is functionally correct prior to synthesis.

The C test bench includes the function main() and any sub-functions that are not in the hierarchy under the top-level function for synthesis. These functions verify that the top-level function for synthesis is functionally correct by providing stimuli to the function for synthesis and by consuming its output.

Vivado HLS uses the test bench to compile and execute the C simulation. During the compilation process, you can select the Launch Debugger option to open a full C-debug environment, which enables you to analyze the C simulation.

Note: Because Vivado HLS uses the test bench to both verify the C function prior to synthesis and to automatically verify the RTL output, using a test bench is highly recommended.

Language Support

Vivado HLS supports the following standards for C compilation/simulation:

- ANSI-C (GCC 4.6)
- C++ (G++ 4.6)
- OpenCL API (1.0 embedded profile)
- SystemC (IEEE 1666-2006, version 2.2)

C, C++, and SystemC Language Constructs

Vivado HLS supports many C, C++, and SystemC language constructs and all native data types for each language, including float and double types. However, synthesis is not supported for some constructs, including:

- Dynamic memory allocation

  An FPGA has a fixed set of resources, and the dynamic creation and freeing of memory resources is not supported.

- Operating system (OS) operations

  All data to and from the FPGA must be read from the input ports or written to output ports. OS operations, such as file read/write or OS queries like time and date, are not supported. Instead, the C test bench can perform these operations and pass the data into the function for synthesis as function arguments.

OpenCL API C Language Constructs

Vivado HLS supports the OpenCL API C language constructs and built-in functions from the OpenCL API C 1.0 embedded profile.

C Libraries

C libraries contain functions and constructs that are optimized for implementation in an FPGA. Using these libraries helps to ensure high quality of results (QoR), that is, the final output is a high-performance design that makes optimal use of the resources. Because the libraries are provided in C, C++, OpenCL API C, or SystemC, you can incorporate the libraries into the C function and simulate them to verify the functional correctness before synthesis.

Vivado HLS provides the following C libraries to extend the standard C languages:

- Arbitrary precision data types
- Half-precision (16-bit) floating-point data types
- Math operations
- Video functions
- Xilinx IP functions, including fast fourier transform (FFT) and finite impulse response (FIR)
- FPGA resource functions to help maximize the use of shift register LUT (SRL) resources

C Libraries Example

C libraries ensure a higher QoR than standard C types. Standard C types are based on 8-bit boundaries (8-bit, 16-bit, 32-bit, 64-bit). However, when targeting a hardware platform, it is often more efficient to use data types of a specific width.

For example, a design with a filter function for a communications protocol requires 10-bit input data and 18-bit output data to satisfy the data transmission requirements. Using standard C data types, the input data must be at least 16-bits and the output data must be at least 32-bits. In the final hardware, this creates a datapath between the input and output that is wider than necessary, uses more resources, has longer delays (for example, a 32-bit by 32-bit multiplication takes longer than an 18-bit by 18-bit multiplication), and requires more clock cycles to complete.

Using an arbitrary precision data type in this design instead, you can specify the exact bit-sizes to be specified in the C code prior to synthesis, simulate the updated C code, and verify the quality of the output using C simulation prior to synthesis. Arbitrary precision data types are provided for C and C++ and allow you to model data types of any width from 1 to 1024-bit. For example, you can model some C++ types up to 32768 bits.

Note: Arbitrary precision types are only required on the function boundaries, because Vivado HLS optimizes the internal logic and removes data bits and logic that do not fanout to the output ports.

Synthesis, Optimization, and Analysis

Vivado HLS is project based. Each project holds one set of C code and can contain multiple solutions. Each solution can have different constraints and optimization directives. You can analyze and compare the results from each solution in the Vivado HLS GUI.

Following are the synthesis, optimization, and analysis steps in the Vivado HLS design process:

1. Create a project with an initial solution.
2. Verify the C simulation executes without error.
3. Run synthesis to obtain a set of results.
4. Analyze the results.

After analyzing the results, you can create a new solution for the project with different constraints and optimization directives and synthesize the new solution. You can repeat this process until the design has the desired performance characteristics. Using multiple solutions allows you to proceed with development while still retaining the previous results.

Optimization

Using Vivado HLS, you can apply different optimization directives to the design, including:

- Instruct a task to execute in a pipeline, allowing the next execution of the task to begin before the current execution is complete.
- Specify a latency for the completion of functions, loops, and regions.
- Specify a limit on the number of resources used.
- Override the inherent or implied dependencies in the code and permit specified operations. For example, if it is acceptable to discard or ignore the initial data values, such as in a video stream, allow a memory read before write if it results in better performance.
- Select the I/O protocol to ensure the final design can be connected to other hardware blocks with the same I/O protocol.

Note: Vivado HLS automatically determines the I/O protocol used by any sub-functions. You cannot control these ports except to specify whether the port is registered.

You can use the Vivado HLS GUI to place optimization directives directly into the source code. Alternatively, you can use Tcl commands to apply optimization directives.

Analysis

When synthesis completes, Vivado HLS automatically creates synthesis reports to help you understand the performance of the implementation. In the Vivado HLS GUI, the Analysis Perspective includes the Performance tab, which allows you to interactively analyze the results in detail.

Figure 1.5: Example of performance tab

The Performance tab shows the following for each state:

- C0: The first state includes read operations on ports a, b, and c and the addition operation.
- C1 and C2: The design enters a loop and checks the loop increment counter and exit condition. The design then reads data into variable x, which requires two clock cycles. Two clock cycles are required, because the design is accessing a block RAM, requiring an address in one cycle and a data read in the next.
- C3: The design performs the calculations and writes output to port y. Then, the loop returns to the start.

OpenCL API C Kernel Synthesis

For OpenCL API C kernels, Vivado HLS always synthesizes logic for the entire work group. You cannot apply the standard Vivado HLS interface directives to an OpenCL API C kernel.

The following OpenCL API C kernel code shows a vector addition design where two arrays of data are summed into a third. The required size of the work group is 16, that is, this kernel must execute a minium of 16 times to produce a valid result.

```
#include <clc.h>

// For VHLS OpenCL C kernels, the full work group is synthesized
__kernel void __attribute__ ((reqd_work_group_size(16, 1, 1)))
vadd(__global int* a,
     __global int* b,
```

```
     __global int* c)
{
    int idx = get_global_id(0);
    c[idx] = a[idx] + b[idx];
}
```

Vivado HLS synthesizes this design into hardware that performs the following:

- 16 reads from interface a and b
- 16 additions and 16 writes to output interface c

RTL Verification

If you added a C test bench to the project, you can use it to verify that the RTL is functionally identical to the original C. The C test bench verifies the output from the top-level function for synthesis and returns zero to the top-level function main() if the RTL is functionally identical. Vivado HLS uses this return value for both C simulation and C/RTL co-simulation to determine if the results are correct. If the C test bench returns a non-zero value, Vivado HLS reports that the simulation failed.

Important: Even if the output data is correct and valid, Vivado HLS reports a simulation failure if the test bench does not return the value zero to function *main()*.

Vivado HLS automatically creates the infrastructure to perform the C/RTL co-simulation and automatically executes the simulation using one of the following supported RTL simulators:

- Vivado Simulator (XSim)
- ModelSim simulator
- VCS
- NCSim
- Riviera

If you select Verilog or VHDL HDL for simulation, Vivado HLS uses the HDL simulator you specify. The Xilinx design tools include Vivado Simulator. Third-party HDL simulators require a license from the third-party vendor. The VCS, NCSim, and Riviera HDL simulators are only supported on the Linux operating system.

RTL Export

Using Vivado HLS, you can export the RTL and package the final RTL output files as IP in any of the following Xilinx IP formats:

- Vivado IP Catalog

  Import into the Vivado IP catalog for use in the Vivado Design Suite.
- Pcore for Embedded Development Kit (EDK)

  Import into Xilinx Platform Studio (XPS).
- Synthesized Checkpoint (.dcp)

  Import directly into the Vivado Design Suite the same way you import any Vivado Design Suite checkpoint.

  Note: The synthesized checkpoint format invokes logic synthesis and compiles the RTL implementation into a gatelevel implementation, which is included in the IP package.

For all IP formats except the synthesized checkpoint, you can optionally execute logic synthesis from within Vivado HLS to evaluate the results of RTL synthesis. This optional step allows you to confirm the estimates provided by Vivado HLS for timing and area before handing off the IP package. These gate-level results are not included in the packaged IP.

Note: Vivado HLS estimates the timing and area resources based on built-in libraries for each FPGA. When you use logic synthesis to compile the RTL into a gate-level implementation, perform physical placement of the gates in the FPGA, and perform routing of the inter-connections between gates, logic synthesis might make additional optimizations that change the Vivado HLS estimates.

Chapter 2

DEVELOPING CUSTOM IP CORE USING HLS

In the previous chapter, we have defined the structure of the microprocessor based system that will be used as a part of the solution of PWM signal generation. In this chapter, we will explain how to generate this system using Vivado HLS tool.

2.1 Create a New Project

The first step in creating a new HLS design will be to create a new project. We will crate a new project using the Vivado HLS New Project wizard. The New Project wizard will create an APP project file for us. It will be place where Vivado HLS will organize our design files and save the design status whenever the processes are run.

To create a new project, follow these steps:

Step 1. Launch the **Vivado HLS** software:

Select **Start ->** **All Programs ->** **Xilinx Design Tools ->** **Vivado 2016.4 ->** **Vivado HLS ->** **Vivado HLS 2016.4** and the **Vivado HLS Welcome Page** page will appear, see Figure 2.1.

Figure 2.1: The Vivado HLS Welcome Page

As can be seen from the Figure above, the **HLS Welcome** page contains a lot of usable Quick Start options:

- **Create New Project** - Launch the project setup wizard.
- **Open Project** - Navigate to an existing project or select from a list of recent projects.
- **Open Example Project** - Open Vivado HLS examples.
- **Tutorials** - Opens the *"Vivado Design Suite Tutorial: High-Level Synthesis"* (UG871).
- **User Guide** - Opens this document, the *"Vivado Design Suite User Guide: High-Level Synthesis"* (UG902).
- **Release Notes Guide** - Opens the *"Vivado Design Suite User Guide: Release Notes, Installation, and Licensing"* (UG973) for the latest software version.

If any projects were previously opened, they will be shown in the **Recent Projects** pane, otherwise this window is not shown in the Welcome screen.

Step 2. In the **Vivado HLS Welcome Page** page, choose **Create New Project** option to open the Project wizard.

Step 3. In the **Project Configuration** dialog box specify the name and the location of the new project:

- In the **Project name** field type **modulator** as the name of the new project
- In the **Location** field click **Browse** button to specify the location where project data will be stored, see Figure 2.2.

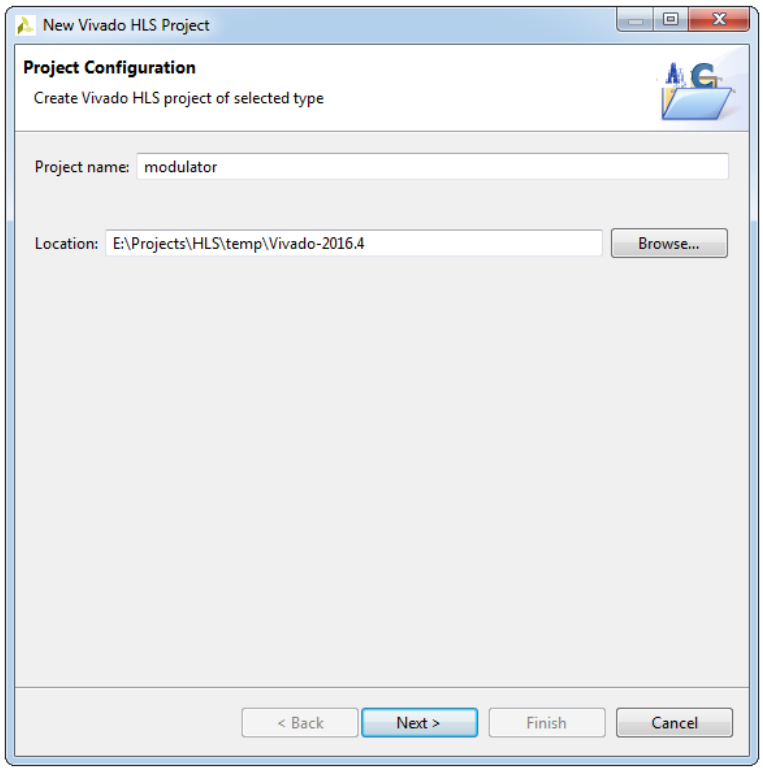


Figure 2.2: Project Configuration dialog box

Note: This step is not required when the project is specified as SystemC, because Vivado HLS automatically identifies the top-level functions.

Step 4. Click **Next**.

Step 5. In the **Add/Remove Files** dialog box, specify the C-based design files:

- Specify **modulator** as the top-level function in the **Top Function** field, see Figure 2.3

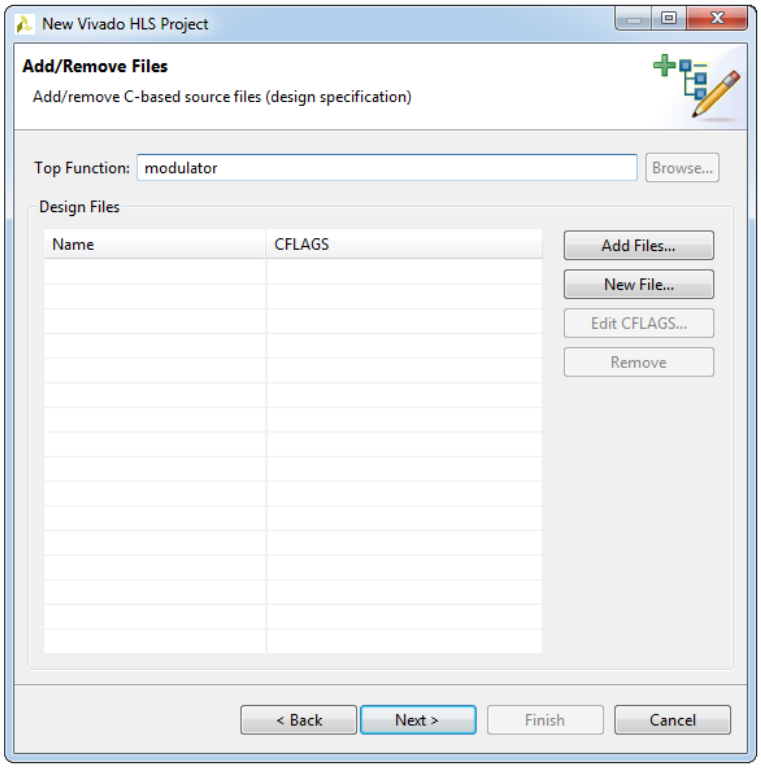


Figure 2.3: Add/Remove Files dialog box

- Click **New File...** button and in the **Save As** dialog box specify ***modulator.cpp*** as a new file name in the **File name** filed and click **Save**, see Figure 2.4

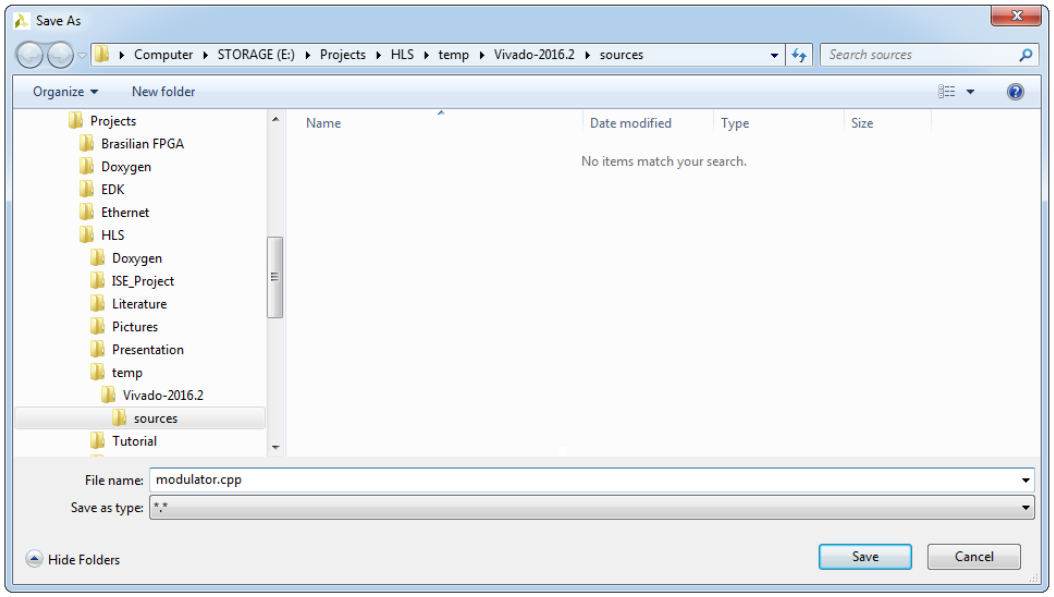


Figure 2.4: Save As dialog box

- After adding new ***modulator.cpp*** C++ file, it should appear as a part of the **Design Files** section, as it is shown on the Figure 2.5

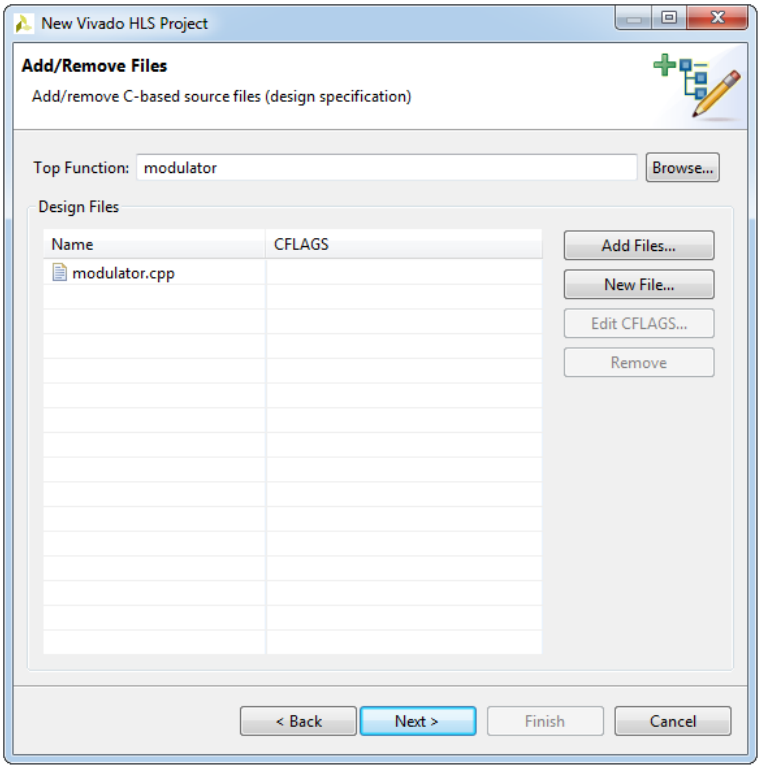


Figure 2.5: Add/Remove Files dialog box with added C++ file

- Click **Next**.

Note: You can use the **Add Files** button to add the existing source code files to the project.

Important: Do not add header files (with the .h suffix) to the project using the **Add Files** button (or with the associated *add_files* Tcl command).

In this example there is only one C design file (**modulator.c**). When there are multiple C files to be synthesized, you must add all of them to the project at this stage. Any header files that exist in the local directory are automatically included in the project. If the header resides in a different location, use the **Edit CFLAGS...** button to add the standard gcc/g++ search path information (for example, *-I<path_to_header_file_dir>*).

Step 6. In the second **Add/Remove Files** dialog box, specify the C-based testbench files:

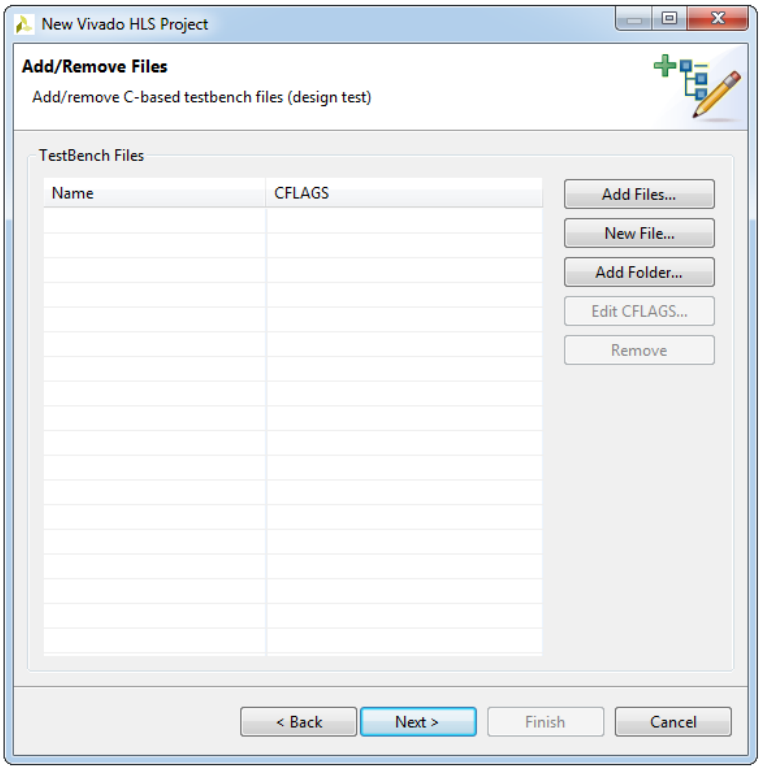


Figure 2.6: Add/Remove Files dialog box

- Click **New File...** button and in the **Save As** dialog box specify ***modulator_tb.cpp*** as a new testbench file name in the **File name** filed and click **Save**, see Figure 2.7

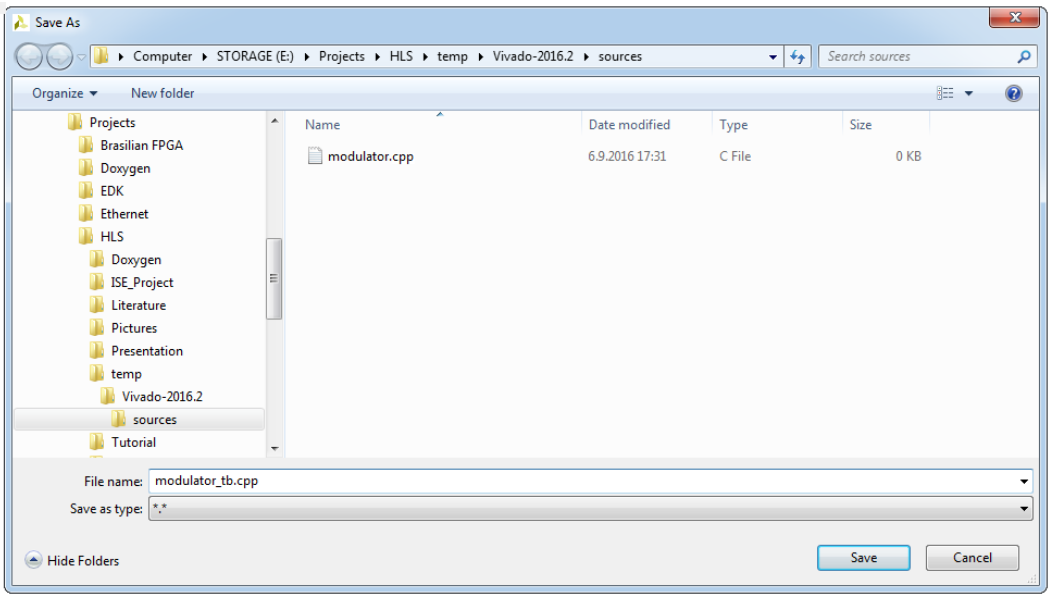


Figure 2.7: Save As dialog box with testbench file

- After adding the new ***modulator_tb.cpp*** testbench file, it should appear as a part of the **TestBench Files** section, as it is shown on the Figure 2.8

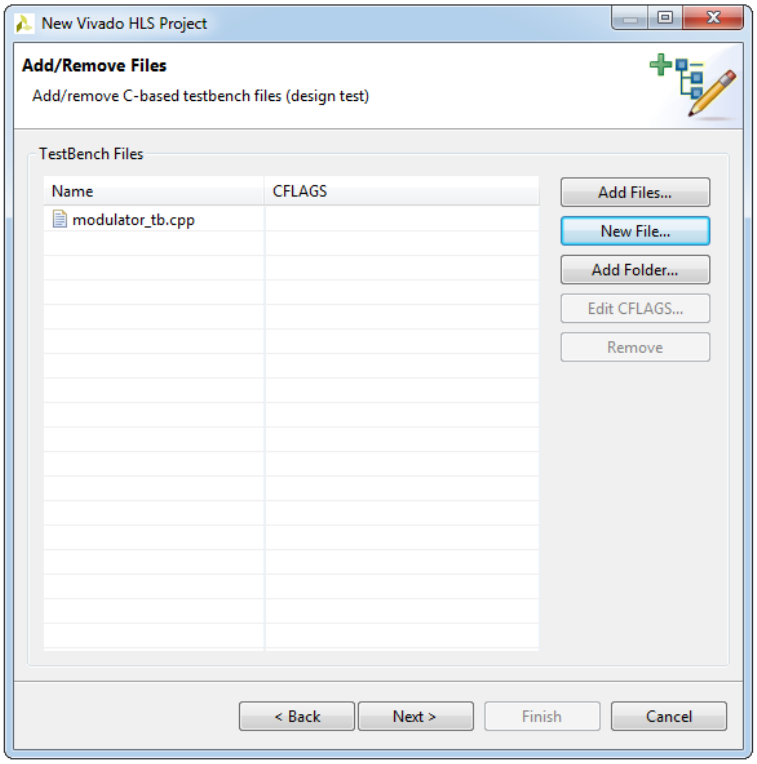


Figure 2.8: Add/Remove TestBench Files dialog box with added testbench file

- Click **Next**.

Note: The testbench 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.

Note: For SystemC designs with header files associated with the test bench but not the design file, you must use the **Add Files** button to add the header files to the project.

In most of the example designs provided with Vivado HLS, the test bench is in a separate file from the design. Having the test bench and the function to be synthesized in separate files keeps a clean separation between the process of simulation and synthesis. If the test bench is in the same file as the function to be synthesized, the file should be added as a source file and a test bench file.

As with the C source files, click the **Add Files** button to add the C test bench and the **Edit CFLAGS** button to include any C compiler options.

If the test bench files exist in a directory, the entire directory might be added to the project, rather than the individual files, using the **Add Folders** button.

Both C simulation (and RTL cosimulation) execute in subdirectories of the solution.

If you do not include all the files used by the test bench (for example, data files read by the test bench), C and RTL simulation might fail due to an inability to find the data files.

The **Solution Configuration** window (shown on the Figure 2.9) specifies the technical specifications of the first solution.

A project can have multiple solutions, each using a different target technology, package, constraints, and/or synthesis directives.

Step 7. In the **Solution Configuration** dialog box accept the default solution name (**solution1**), clock period (**10 ns**), and blank clock uncertainty (defaults to 12.5% of the clock period, when it is left blank then it is undefined), see Figure 2.9.

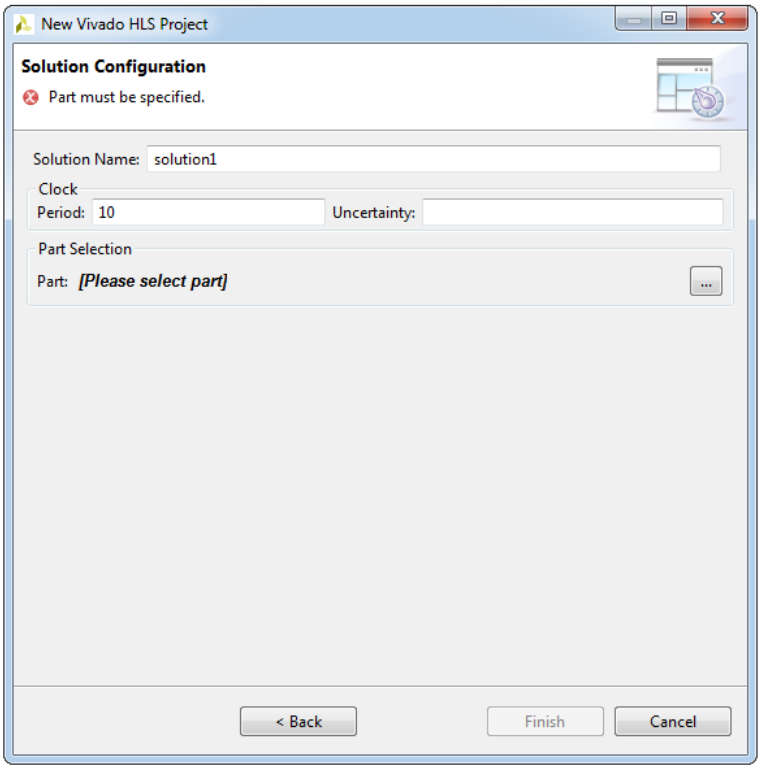


Figure 2.9: Solution Configuration dialog box

The the **Solution Configuration** dialog box allows you to specify the details of the first solution:

- **Solution Name**: Vivado HLS provides the initial default name solution1, but you can specify any name for the solution.
- **Clock Period**: The clock period specified in units of ns or a frequency value specified with the MHz suffix (for example, 100 MHz).
- **Uncertainty**: The clock period used for synthesis is the clock period minus the clock uncertainty. Vivado HLS uses internal models to estimate the delay of the operations for each FPGA. The clock uncertainty value provides a controllable margin to account for any increases in net delays due to RTL logic synthesis, place, and route. If not specified in nanoseconds (ns) or a percentage, the clock uncertainty defaults to 12.5% of the clock period.
- **Part**: Click to select the appropriate technology, as shown in the following figure.

Step 8. In the **Solution Configuration** dialog box click the part selection button to open the part selection window.

You can use the filter to reduce the number of device in the device list. If the target is a board, specify boards in the top-left corner and the device list is replaced by a list of the supported boards (and Vivado HLS automatically selects the correct target device).

Step 9. In the **Device Selection Dialog** dialog box choose a default Xilinx part or board for your project. Select **Boards** to choose the default board for the project and a list of evaluation boards will be displayed, see Figure 2.10.

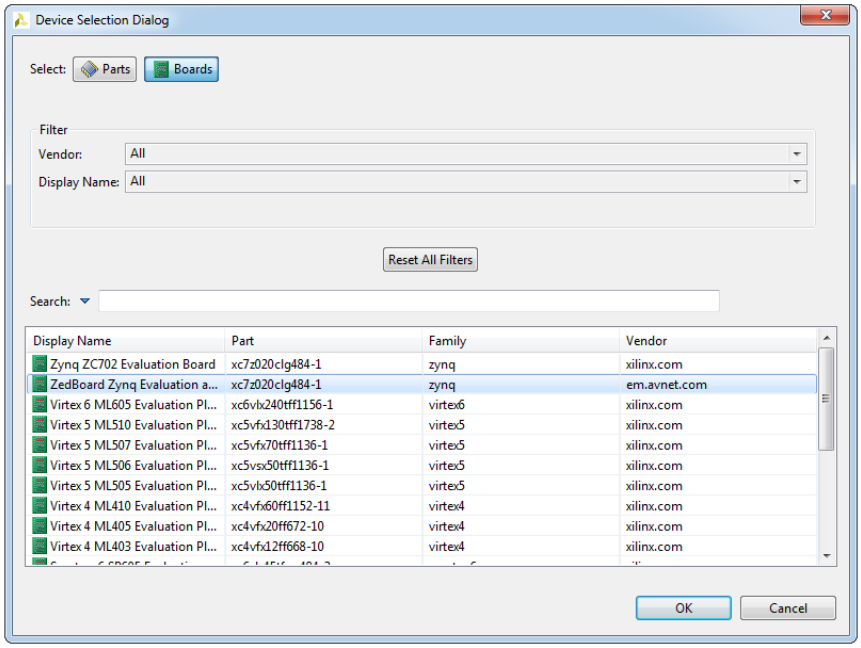


Figure 2.10: Device Selection Dialog dialog box

Step 10. Select **ZedBoard Zynq Evaluation and Development Kit** as it is shown on the Figure above and click **OK**.

In the **Solution Configuration** dialog box, the selected part name now appears under the **Part Selection** heading, see Figure 2.11.

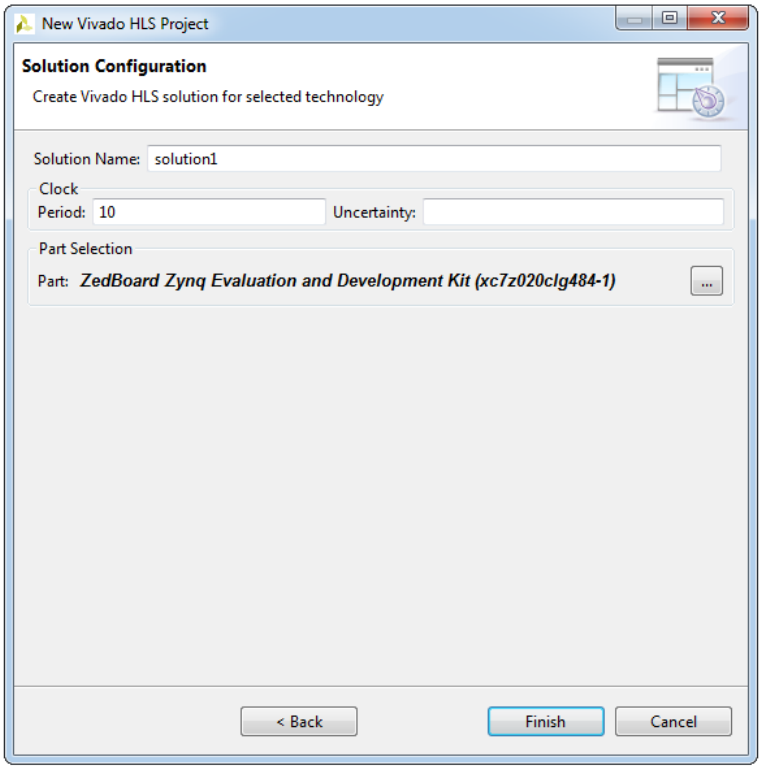


Figure 2.11: Solution Configuration dialog box with selected board

Step 11. In the **Solution Configuration** dialog box, click **Finish** to open the created Vivado HLS project, see Figure 2.12.

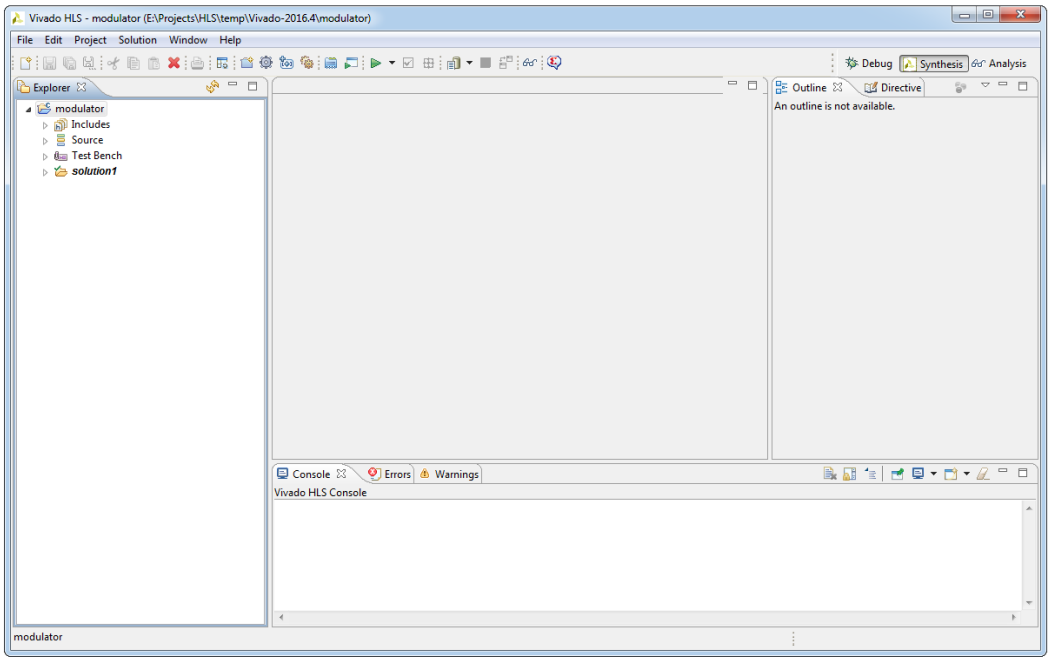


Figure 2.12: Vivado HLS Project

After we finished with the new project creation, in a few seconds Vivado HLS project will appear, see Figure 2.12.

When Vivado HLS creates a new project, it also creates a directory with the name and at the location that we specified in the GUI (see Figure 2.2). That means that the all project data will be stored in the project_name (**modulator**) directory.

In the Vivado HLS project you can notice the following:

- The project name appears on the top line of the Explorer window
- A Vivado HLS project arranges information in a hierarchical form
- The project holds information on the design source, test bench, and solutions
- The solution holds information on the target technology, design directives, and results
- There can be multiple solutions within a project, and each solution is an implementation of the same source code.

Note: At any time, you can change project or solution settings using the corresponding Project Settings and/or Solution Settings buttons in the toolbar.

The Vivado HLS GUI consists of four panes:

- **Explorer Pane**

  Shows the project hierarchy. As you proceed through the validation, synthesis, verification, and IP packaging steps, sub-folders with the results of each step are created automatically inside the solution directory (named *csim*, *syn*, *sim*, and *impl* respectively).

  When you create new solutions, they appear inside the project hierarchy alongside solution1.

- **Information Pane**

  Shows the contents of any files opened from the Explorer pane. When operations complete, the report file opens automatically in this pane.

- **Auxiliary Pane**

  Cross-links with the Information pane. The information shown in this pane dynamically adjusts, depending on the file open in the Information pane.

- **Console Pane**

  Shows the messages produced when Vivado HLS runs. Errors and warnings appear in Console pane tabs.

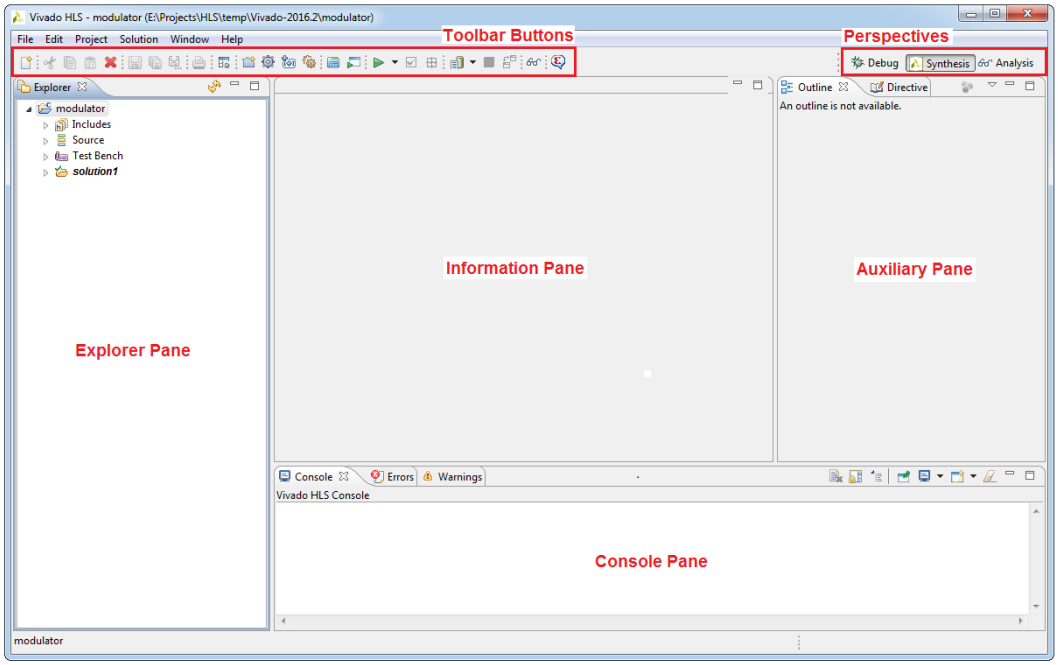


Figure 2.13: Vivado HLS GUI

In the Vivado HLS GUI you can also find:

- **Toolbar Buttons**

  You can perform the most common operations using the Toolbar buttons.

  When you hold the cursor over the button, a popup tool tip opens, explaining the function. Each button also has an associated menu item available from the pull-down menus.

- **Perspectives**

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

  - **Synthesis Perspective**
    The default perspective allows you to synthesize designs, run simulations, and package the IP.
  - **Debug Perspective**
    Includes panes associated with debugging the C code. You can open the Debug Perspective after the C code compiles (unless you use the Optimizing Compile mode as this disables debug information).
  - **Analysis Perspective**
    Windows in this perspective are configured to support analysis of synthesis results. You can use the Analysis Perspective only after synthesis completes.

2.2 Develop C Algorithm

The first step within an HLS project is to develop a C algorithm for your design. In this tutorial the actual algorithm will be written in C++ programming language.

As it is already explained in the previous sub-chapter, with the ***modulator*** project creation we have already created two empty C++ files, ***modulator.cpp*** and ***modulator_tb.cpp***. Now it is time to write their content, as well as the content of the ***modulator.h*** header file that will be stored in the same directory where these two files are saved.

The content of these three files can be found in the text below.

modulator.cpp

```
#include "ap_int.h"
#include "math.h"
#include "modulator.h"

// function that calculates sine wave samples value
void init_sine_table(ap_uint<width> *sine)
{
    float temp;

    init_sine: for (int i = 0; i < sine_samples; i ++)
        // sin (2*pi*i / N) * (2^(width-1) - 1) + 2^(width-1) - 1, N = 2^depth
        sine[i] = (ap_uint<width>)(sin(2*3.14*i/sine_samples)*(sine_ampl/2.0-1.0)+sine_ampl/2.0-1.0);

}

// pwm generator
void modulator(
   ap_uint<1> sw0,       // switch used for selecting frequency
   ap_uint<1> *pwm_out)  // pointer to pwm output
{
    static ap_uint<depth> counter = 0;           // counter for sine wave sample counting
    static ap_uint<width> sine[sine_samples];    // samples of the sine wave signal

    // sine table initialization
    init_sine_table(sine);

    // hold pwm_out high for specified number of clock cycles
    onloop: for (ap_uint<20> j = 0; j < (ap_uint<20>)(period[sw0]*sine[counter]); j++)
    {
         pwm_out = 1;
    }

    // hold pwm_out low for specified number of clock cycles
    offloop: for (ap_uint<20> j = 0; j < (ap_uint<20>)(period[sw0]*(sine_ampl - sine[counter])); j++)
    {
         pwm_out = 0;
    }

    counter++;
}
```

modulator_tb.cpp

```
#include <iostream>
#include "ap_int.h"
#include "modulator.h"

using namespace std;

ap_uint<1> pwm_out;    // pulse width modulated signal

int main(int argc, char **argv)
{
    for (int i = 0; i < 256; i ++)
        modulator(0, &pwm_out);

    for (int i = 0; i < 256; i ++)
        modulator(1, &pwm_out);

    return 0;
}
```

modulator.h

```
#ifndef __PWM_H__
#define __PWM_H__

#include "ap_int.h"
#include <cmath>
using namespace std;

#define depth               8          // the number of bits used to represent sample count of sine wave
#define width               12         // the number of bits used to represent amplitude value

#define sine_samples        256        // maximum number of samples in one period of the signal
#define sine_ampl           4096       // maximum amplitude value of the sine wave

#define refclk_frequency    100000000  // reference clock frequency (100 MHz)

#define freq_low            1          // first frequency for the PWM signal, specified in Hz
#define freq_high           3.5        // second frequency for the PWM signal, specified in Hz

// minimum duration of high value of pwm signal for two different frequencies
const float period[2] = {(float)(refclk_frequency/(sine_ampl*sine_samples*freq_low)),
                         (float)(refclk_frequency/(sine_ampl*sine_samples*freq_high))};
```

```
// Prototype of top level function for C-synthesis
void modulator(
    ap_uint<1> sw0,          // switch used for selecting frequency
    ap_uint<1> *pwm_out);    // pointer to pwm output

#endif
```

To add the content of the ***modulator.cpp*** and ***modulator_tb.cpp*** files, do the following steps:

Step 1. In the Vivado HLS **Explorer** pane expand **Source** folder and double-click on the ***modulator.cpp*** C++ file to open it, see Figure 2.14.

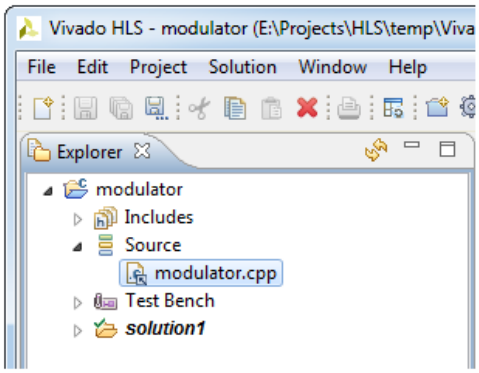


Figure 2.14: Source folder with modulator.cpp file

Step 2. In the opened ***modulator.cpp*** file copy the content of the file from the text above and click **Save** button.

Step 3. Repeat the same procedure for the ***modulator_tb.cpp*** testbench file. Therefore, in the Vivado HLS **Explorer** pane expand **Test Bench** folder and double-click on the ***modulator_tb.cpp*** file to open it, see Figure 2.15.

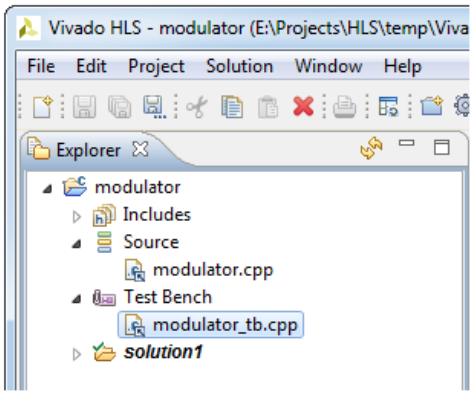


Figure 2.15: Test Bench folder with modulator_tb.cpp file

Step 4. In the opened ***modulator_tb.cpp*** file copy the content of the file from the text above and click **Save** button.

Step 5. For the ***modulator.h*** header file creation it is necessary to write it in an text editor and save it in the same folder where the rest of the files are stored. By doing so, ***modulator.h*** header file will be automatically included in the project and you should find it in the **Includes** folder of the **Explorer** pane. The content of the ***modulator.h*** header file you can also find in the text above.

2.3 Verify C Algorithm

The second step within an HLS project is to confirm that the C code is correct. This process is called ***C Validation*** or ***C Simulation***.

Verification in the Vivado HLS flow can be separated into two distinct processes:

1. Pre-synthesis validation that validates the C program correctly implements the required functionality.

2. Post-synthesis verification that verifies the RTL is correct.

Both processes are referred to as simulation: C simulation and C/RTL co-simulation.

Before synthesis, the function to be synthesized should be validated with a test bench using C simulation. A C test bench includes a top-level function *main()* and the function to be synthesized. It might include other functions. An ideal test bench has the following attributes:

- The test bench is self-checking and verifies the results from the function to be synthesized are correct.

- If the results are correct the test bench returns a value of 0 to *main()*. Otherwise, the test bench should return any non-zero values.

Vivado HLS synthesizes an OpenCL API C kernel. To simulate an OpenCL API C kernel, you must use a standard C test bench. You cannot use the OpenCL API C host code as the C test bench.

Step 1. Click the **Run C Simulation** toolbar button (Figure 2.16) to open the **C Simulation** dialog box, shown in the Figure 2.17.

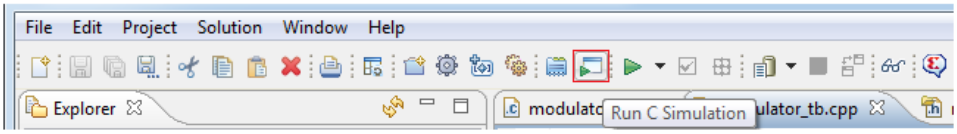


Figure 2.16: Run C Simulation button

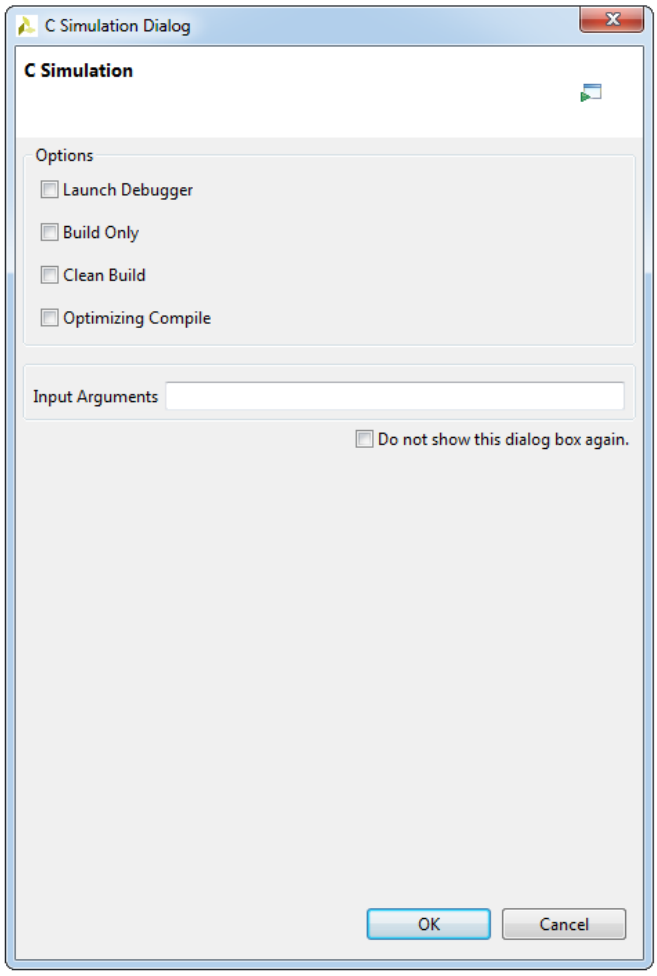


Figure 2.17: C Simulation dialog box

The another way to open the **C Simulation** dialog box is to choose **Project ->** **Run C Simulation** option from the main HLS toolbar menu.

In the **C Simulation** dialog box you can find the following options:

- **Launch Debugger** - This option compiles the C code and automatically opens the debug perspective. From within the debug perspective the Synthesis perspective button (top left) can be used to return to the synthesis perspective.
- **Build Only** - This option compiles the C code, but does not run the simulation. Details on executing the C simulation are covered in *"Reviewing the Output of C Simulation"* document.
- **Clean Build** - This option remove any existing executable and object files from the project before compiling the code.
- **Optimized Compile** - By default the design is compiled with debug information, allowing the compilation to be analyzed in the debug perspective. This option uses a higher level of optimization effort when compiling the design but removes all information required by the debugger. This increases the compile time but should reduce the simulation run time.

Step 2. In the **C Simulation** dialog box, just click **OK**.

If no option is selected in the **C Simulation** dialog box, the C code is compiled and the C simulation is automatically executed. The results are shown on the Figure 2.18. When the C code is simulated successfully, the Console window displays a message.

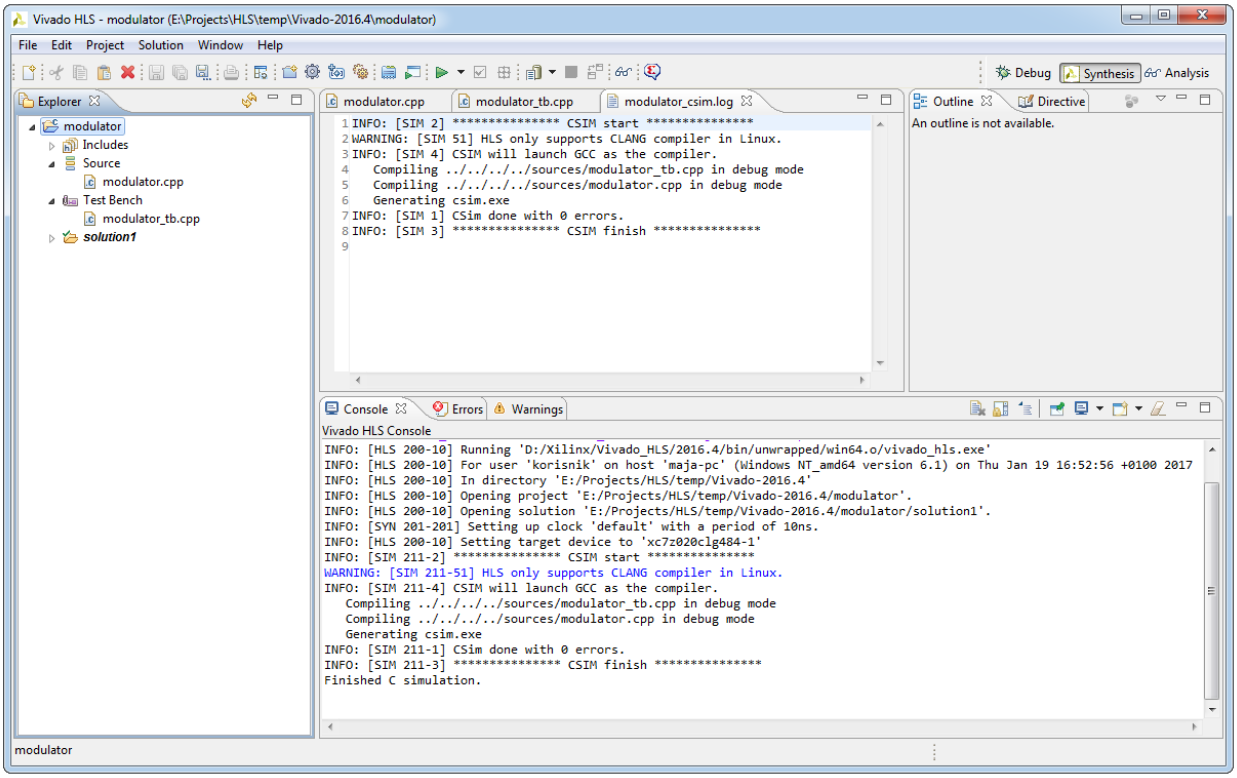


Figure 2.18: Console window showing message about successful simulation

The design is now ready for synthesis.

Note: If the C simulation ever fails, select the **Launch Debugger** option in the **C Simulation** dialog box, compile the design, and automatically switch to the Debug perspective. There you can use a C debugger to fix any problems.

2.3.1 C Simulation Output Files

When C simulation completes, a folder ***csim*** is created inside the ***solution1*** folder, see Figure 2.19.

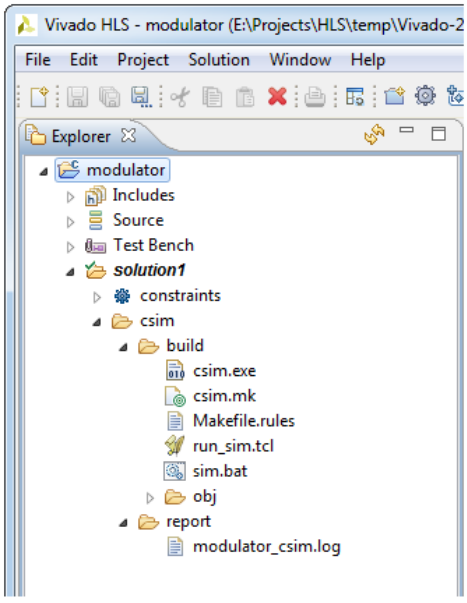


Figure 2.19: Explorer window with C Simulation Output Files

The folder ***csim/build*** is the primary location for all files related to the C simulation:

- Any files read by the test bench are copied to this folder
- The C executable file *csim.exe* is created and run in this folder
- Any files written by the test bench are created in this folder.

If the **Build Only** option is selected in the C Simulation dialog box, the file ***csim.exe*** is created in this folder, but the file is not executed. The C simulation is run manually by executing this file from a command shell. On Windows the Vivado HLS command shell is available through the start menu.

The folder ***csim/report*** contains a log file of the C simulation.

The next step in the Vivado HLS design flow is to execute synthesis.

2.4 Synthesize C Algorithm into an RTL Implementation (High-Level Synthesis)

In this step, you synthesize the C design into an RTL design and review the synthesis report.

Step 1. Click the **Run C Synthesis** toolbar button (Figure 2.20) or use the **Solution ->** **Run C Synthesis ->** **Active Solution** option form the main Vivado HLS menu to synthesize the design to an RTL implementation.

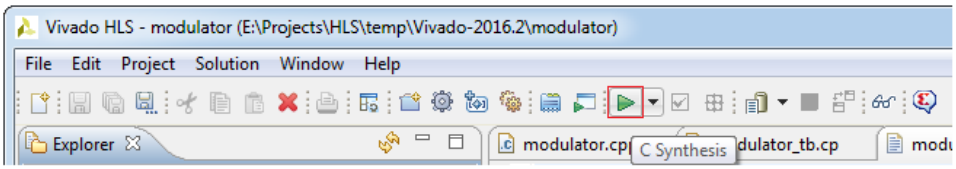


Figure 2.20: Run C Synthesis button

During the synthesis process messages are echoed to the console window. The message include information messages showing how the synthesis process is proceeding. The messages also provide details on the synthesis process.

When synthesis completes, the synthesis report for the top-level function opens automatically in the Information pane as shown in the following figure.

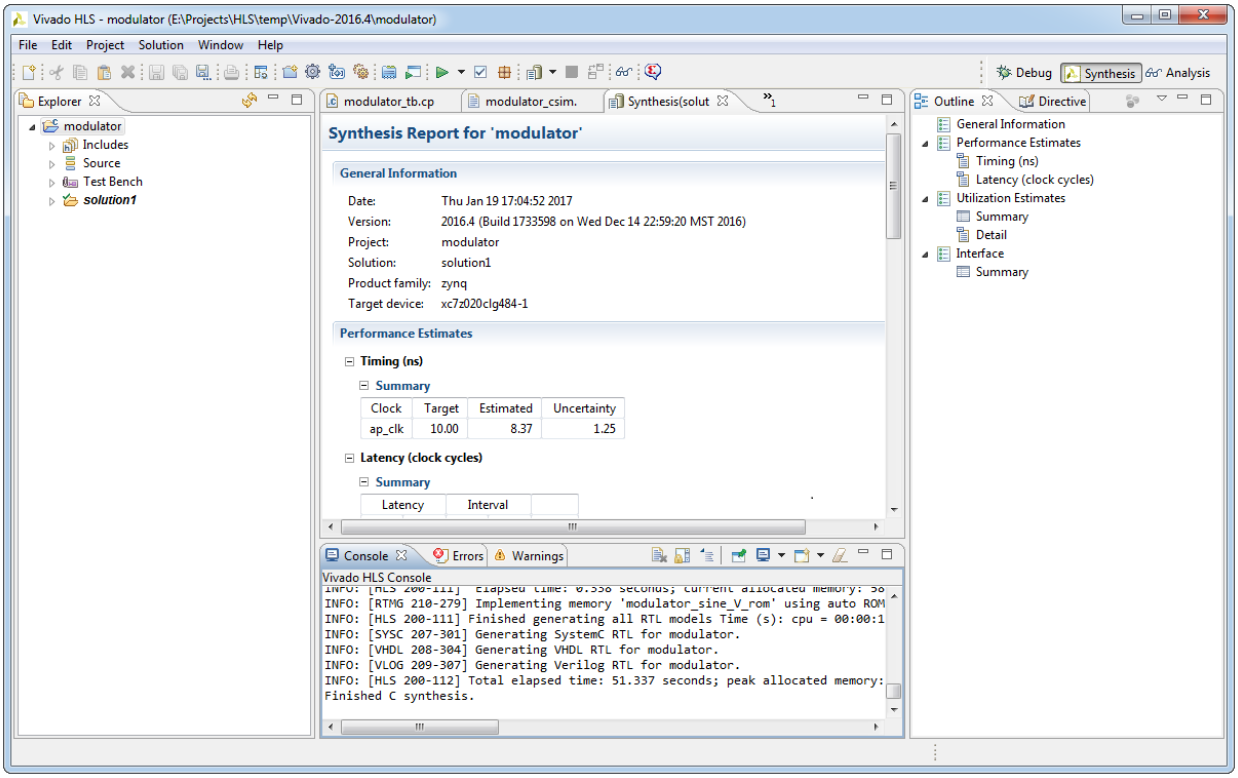


Figure 2.21: Information pane with synthesis report

The synthesis report provides details on both the performance and area of the RTL design. The **Outline** tab on the right-hand side can be used to navigate through the report. In this sub-chapter will be explained only certain report categories which are important for the current stage of design development.

The detail explanation of all synthesis report categories is presented in the Table 2.1 of sub-chapter *2.4.2 C Synthesis Results*.

Step 2. In the **Outline** tab click **Performance Estimates** option, see Figure 2.22.

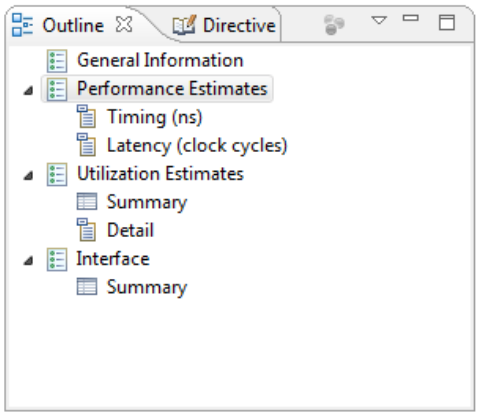


Figure 2.22: Outline tab with selected Performance Estimates option

In the **Performance Estimates** pane, expand **Timing (ns)**/**Summary** and you can see that the clock period is set to 10 ns, see Figure 2.23. Vivado HLS targets a clock period of **Clock Target** minus **Clock Uncertainty** (10.00 - 1.25 = 8.75 ns in this example).

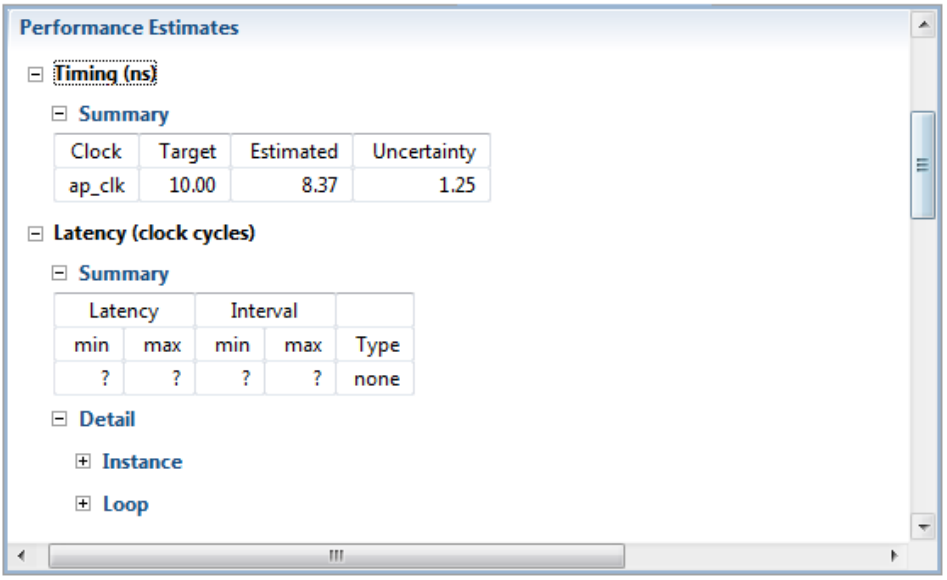


Figure 2.23: Performance Estimates report - Timing Summary

The clock uncertainty ensures there is some timing margin available for the (at this stage) unknown net delays due to place and routing.

The estimated clock period (worst-case delay) is 8.37 ns, which meets the 8.75 ns timing requirement.

In the **Performance Estimates** pane, expand **Latency (clock cycles)**/**Summary** and you can see:

- The design has a latency of ? clock cycles: it takes ? clocks to output the results.
- The interval is ? clock cycles: the next set of inputs is read after ? clocks. This is one cycle after the final output is written. This indicates the design is not pipelined. The next execution of this function (or next transaction) can only start when the current transaction completes.

Note: In our design Vivado HLS can't calculate latency values.

In the **Performance Estimates** pane, expand **Latency (clock cycles)**/**Detail** and you can see:

- There are no sub-blocks in this design. Expanding the **Instance** section shows no sub-modules in the hierarchy.
- Expanding the **Loop** section you can see that all the latency delay is due to the RTL logic synthesized from the loops named *onloop* and *offloop*. This logic executes ? times (Trip Count). Each execution requires 1 clock cycle (Iteration Latency), for a total of ? clock cycles, to execute all iterations of the logic synthesized from this loop (Latency).

  As we already said, in our design Vivado HLS can't calculate latency values.

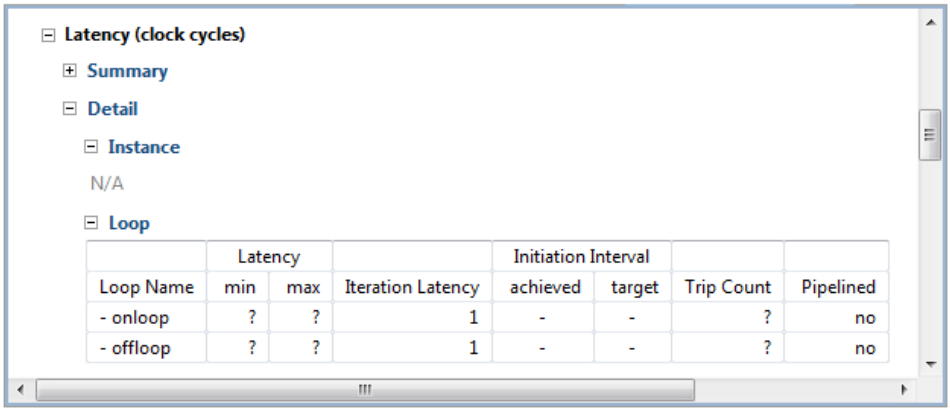
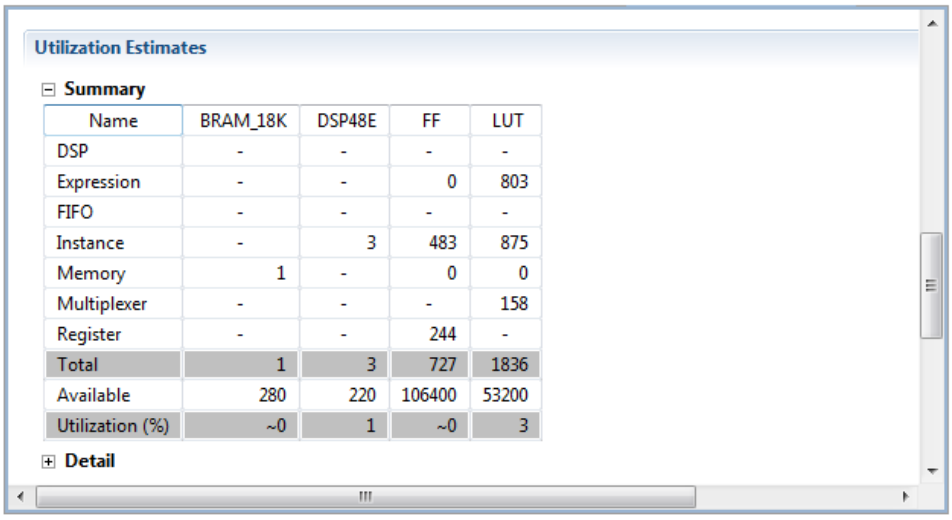


Figure 2.24: Performance Estimates report - Loop Latency Detail

Step 3. In the **Outline** tab click **Utilization Estimates** option, see Figure 2.22.



Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	803
FIFO	-	-	-	-
Instance	-	3	483	875
Memory	1	-	0	0
Multiplexer	-	-	-	158
Register	-	-	244	-
Total	1	3	727	1836
Available	280	220	106400	53200
Utilization (%)	~0	1	~0	3

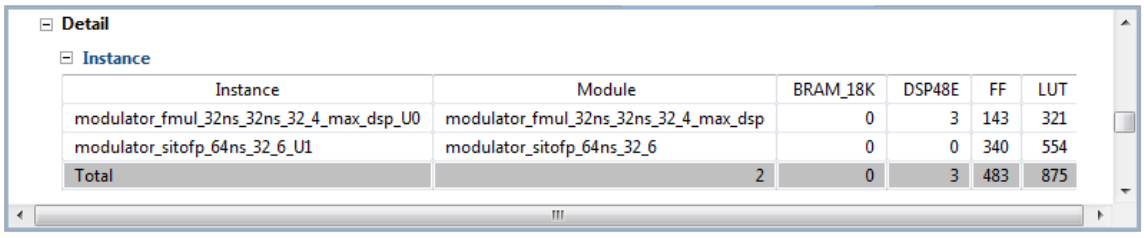
Detail

Figure 2.25: Utilization Estimates report - Summary

In the **Utilization Estimates** pane, under the **Summary** section, you can see:

- The design uses 1 BRAM_18K memory, 3 DSP48E, 727 flip-flops and 1836 LUTs. At this stage, the device resource numbers are estimates.
- The resource utilization numbers are estimates because RTL synthesis might be able to perform additional optimizations, and these figures might change after RTL synthesis.

In the **Utilization Estimates** pane, expand **Detail**/**Instance** section and you will see:



Detail

Instance

Instance	Module	BRAM_18K	DSP48E	FF	LUT
modulator_fmul_32ns_32ns_32_4_max_dsp_U0	modulator_fmul_32ns_32ns_32_4_max_dsp	0	3	143	321
modulator_sitofp_64ns_32_6_U1	modulator_sitofp_64ns_32_6	0	0	340	554
Total	2	0	3	483	875

Figure 2.26: Utilization Estimates report - Detail Instance

- The resources specified here are used by the sub-blocks instantiated at this level of the hierarchy. Although our design does not have any hierarchy, Vivado HLS introduced it when performing multiplication of floating point value and unsigned integer value (see lines 28 and 34 in ***modulator.cpp*** source code). There are two instances created by Vivado HLS:
- ***modulator_fmul_32ns_32ns_32_4_max_dsp_U0*** - used for single precision floating point multiplication and
- ***modulator_sitofp_64ns_32_6_U1*** - used for converting integer value to floating point value.

For each instance Vivado HLS reports how many resources are necessary to implement it (number of BRAMs, DSPs, FFs, LUTs).

Step 4. In the **Outline** tab click **Interface** option, see Figure 2.22.

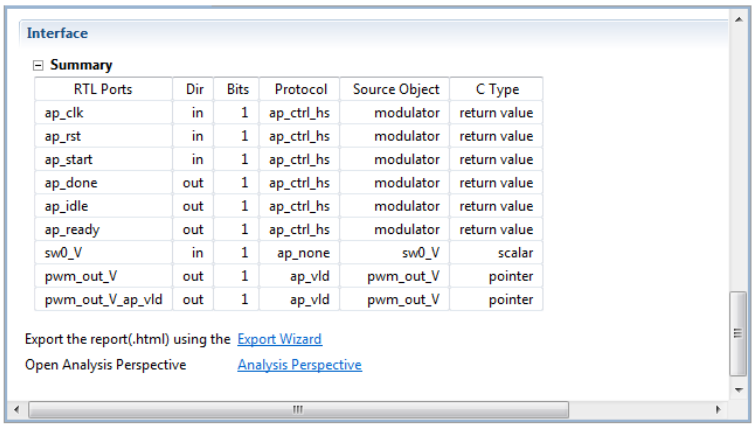


Figure 2.27: Interface report - Summary

The **Interface** report shows the ports and I/O protocols created by interface synthesis:

- The design has a clock and reset port (*ap_clk* and *ap_rst*). These are associated with the Source Object *modulator:* the design itself.
- There are additional ports associated with the design as indicated by Source Object *modulator*. Synthesis has automatically added some block level control ports: *ap_start*, *ap_done*, *ap_idle*, and *ap_ready*.
- The *Interface Synthesis* tutorial provides more information about these ports.
- Scalar input argument *sw0_V* is implemented as a data port with no I/O protocol (*ap_none*).
- Finally, the function outputs *pwm_out_V* and *pwm_out_V_ap_vld* are 1-bit data ports with an associated output valid signal indicator *pwm_out_V*.

2.4.1 C Synthesis Output Files

When synthesis completes, the folder ***syn*** is now available in the ***solution1*** folder.

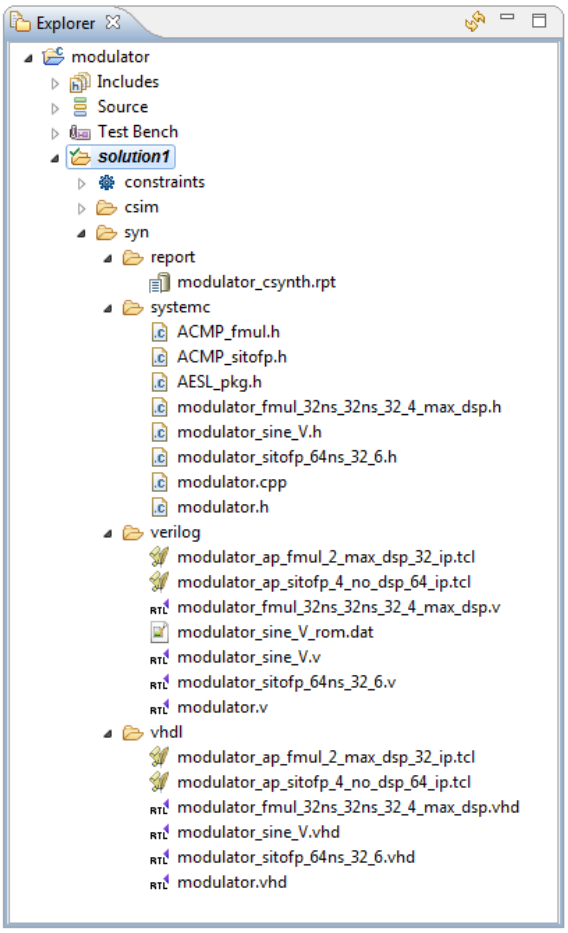


Figure 2.28: Explorer window with C Synthesis Output Files

The ***syn*** folder contains 4 sub-folders. A ***report*** folder and one folder for each of the RTL output formats.

The ***report*** folder contains a report file for the top-level function and one for every sub-function in the design: provided the function was not inlined using the INLINE directive or inlined automatically by Vivado HLS. The report for the top-level function provides details on the entire design.

The ***verilog***, ***vhdl***, and ***systemc*** folders contain the output RTL files. Figure 2.28 shows all four folders expanded. The top-level file has the same name as the top-level function for synthesis. In the C design there is one RTL file for each function (not inlined). There might be additional RTL files to implement sub-blocks (block RAM, pipelined multipliers, etc).

Important: Xilinx does not recommend using these files for RTL synthesis. Instead, Xilinx recommends using the packaged IP output files discussed later in this design flow.

In cases where Vivado HLS uses Xilinx IP in the design, such as with floating point designs, the RTL directory includes a script to create the IP during RTL synthesis. If the files in the *syn* folder are used for RTL synthesis, it is your responsibility to correctly use any script files present in those folders. If the package IP is used, this process is performed automatically by the design Xilinx tools.

2.4.2 C Synthesis Results

The two primary features provided to analyze the RTL design are:

1. Synthesis reports
2. Analysis Perspective

In addition, if you are more comfortable working in an RTL environment, Vivado HLS creates two projects during the IP packaging process:

1. Vivado Design Suite project
2. Vivado IP Integrator project

Synthesis Reports

When synthesis completes, the synthesis report for the top-level function opens automatically in the information pane (Figure 2.21). The report provides details on both the performance and area of the RTL design. The Outline tab on the right-hand side can be used to navigate through the report.

The following table explains the categories in the synthesis report.

Table 2.1: Synthesis Report Category

Category	Desription
General Information	Details on when the results were generated, the version of the software used, the project name, the solution name, and the technology details.
Performance Estimates -> Timing	The target clock frequency, clock uncertainty, and the estimate of the fastest achievable clock frequency.
Performance Estimates -> Latency -> Summary	Reports the latency and initiation interval for this block and any sub-blocks instantiated in this block. Each sub-function called at this level in the C source is an instance in this RTL block, unless it was inlined. The latency is the number of cycles it takes to produce the output. The initiation interval is the number of clock cycles before new inputs can be applied. In the absence of any PIPELINE directives, the latency is one cycle less than the initiation interval (the next input is read when the final output is written).
Performance Estimates -> Latency -> Detail	The latency and initiation interval for the instances (sub-functions) and loops in this block. If any loops contain sub-loops, the loop hierarchy is shown. The min and max latency values indicate the latency to execute all iterations of the loop. The presence of conditional branches in the code might make the min and max different. The Iteration Latency is the latency for a single iteration of the loop. If the loop has a variable latency, the latency values cannot be determined and are shown as a question mark (?). See the text after this table. Any specified target initiation interval is shown beside the actual initiation interval achieved. The tripcount shows the total number of loop iterations.
Utilization Estimates -> Summary	This part of the report shows the resources (LUTS, Flip-Flops, DSP48s) used to implement the design.
Utilization Estimates -> Details -> Instance	The resources specified here are used by the sub-blocks instantiated at this level of the hierarchy. If the design only has no RTL hierarchy, there are no instances reported. If any instances are present, clicking on the name of the instance opens the synthesis report for that instance.
Utilization Estimates -> Details -> Memory	The resources listed here are those used in the implementation of memories at this level of the hierarchy. Vivado HLS reports a single-port BRAM as using one bank of memory and reports a dual-port BRAM as using two banks of memory.

Utilization Estimates -> Details -> FIFO	The resources listed here are those used in the implementation of any FIFOs implemented at this level of the hierarchy.
Utilization Estimates -> Details -> Shift Register	A summary of all shift registers mapped into Xilinx SRL components. Additional mapping into SRL components can occur during RTL synthesis.
Utilization Estimates -> Details -> Expressions	This category shows the resources used by any expressions such as multipliers, adders, and comparators at the current level of hierarchy. The bit-widths of the input ports to the expressions are shown.
Utilization Estimates -> Details -> Multiplexors	This section of the report shows the resources used to implement multiplexors at this level of hierarchy. The input widths of the multiplexors are shown.
Utilization Estimates -> Details -> Register	A list of all registers at this level of hierarchy is shown here. The report includes the register bit-widths.
Interface Summary -> Interface	This section shows how the function arguments have been synthesized into RTL ports. The RTL port names are grouped with their protocol and source object:these are the RTL ports created when that source object is synthesized with the stated I/O protocol.

Certain Xilinx devices use stacked silicon interconnect (SSI) technology. In these devices, the total available resources are divided over multiple super logic regions (SLRs). When you select an SSI technology device as the target technology, the utilization report includes details on both the SLR usage and the total device usage.

Important: When using SSI technology devices, it is important to ensure that the logic created by Vivado HLS fits within a single SLR. For information on using SSI technology devices.

A common issue for new users of Vivado HLS is seeing a synthesis report similar to the following figure. The latency values are all shown as a “?” (question mark).

Vivado HLS performs analysis to determine the number of iteration of each loop. If the loop iteration limit is a variable, Vivado HLS cannot determine the maximum upper limit.

If the latency or throughput of the design is dependent on a loop with a variable index, Vivado HLS reports the latency of the loop as being unknown (represented in the reports by a question mark “?”).

The TRIPCOUNT directive can be applied to the loop to manually specify the number of loop iterations and ensure the report contains useful numbers. The *-max* option tells Vivado HLS the maximum number of iterations that the loop iterates over, the *-min* option specifies the minimum number of iterations performed and the *-avg* option specifies an average tripcount.

Note: The TRIPCOUNT directive does not impact the results of synthesis.

The tripcount values are used only for reporting, to ensure the reports generated by Vivado HLS show meaningful ranges for latency and interval. This also allows a meaningful comparison between different solutions.

If the C assert macro is used in the code, Vivado HLS can use it to both determine the loop limits automatically and create hardware that is exactly sized to these limits.

Analysis Perspective

In addition to the synthesis report, you can use the Analysis Perspective to analyze the results. To open the Analysis Perspective, click the Analysis button as shown in the following figure.

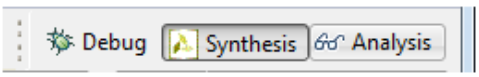


Figure 2.29: Analysis Perspective Button

The **Analysis Perspective** provides both a tabular and graphical view of the design performance and resources and supports cross-referencing between both views. The following figure shows the default window configuration when the Analysis Perspective is first opened.

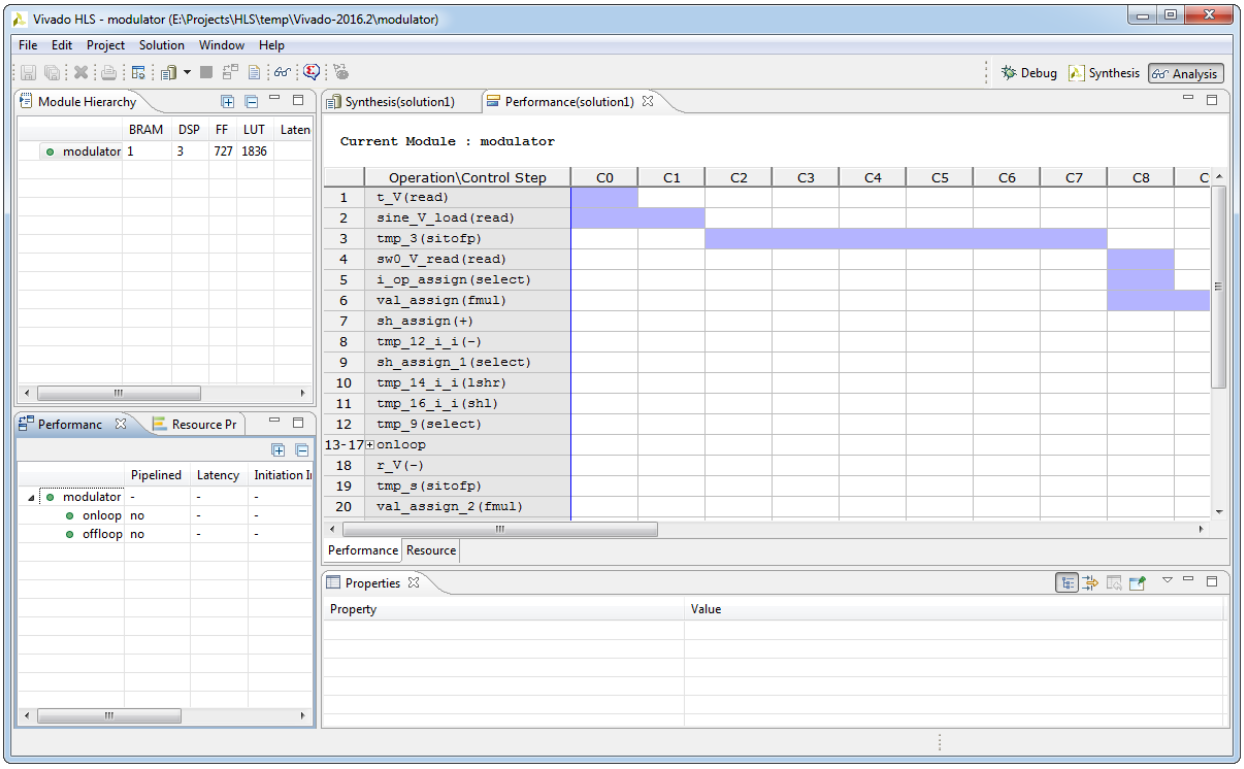


Figure 2.30: Default Analysis Perspective in the Vivado HLS GUI

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

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

The **Performance Profile** pane provides details on the performance of the block currently selected in the Module Hierarchy pane, in this case, the *modulator* block highlighted in the Module Hierarchy pane.

- The performance of the block is a function of the sub-blocks it contains and any logic within this level of hierarchy. The Performance Profile pane shows items at this level of hierarchy that contribute to the overall performance.
- Performance is measured in terms of latency and the initiation interval. This pane also includes details on whether the block was pipelined or not.
- In this example, you can see that two loops (*onloop* and *offloop*) are implemented as logic at this level of hierarchy.

The **Schedule View** pane shows how the operations in this particular block are scheduled into clock cycles. The default view is the **Performance** view.

- The left-hand column lists the resources.
  - Sub-blocks are green.
  - Operations resulting from loops in the source are coloured yellow.
  - Standard operations are purple.
- The **modulator** has three main parts:
  - A call to the **init_sine_table** function which initializes *sine* array,
  - A loop called **onloop**, and
  - A loop called **offloop**.

- The top row lists the control states in the design. Control states are the internal states used by Vivado HLS to schedule operations into clock cycles. There is a close correlation between the control states and the final states in the RTL FSM, but there is no one-to-one mapping.

The following figure shows that you can select an operation and right-click the mouse (**Goto Source** option) to open the associated variable in the source code view. You can see that the write operation is implementing the writing of data into the buf array from the input array variable.

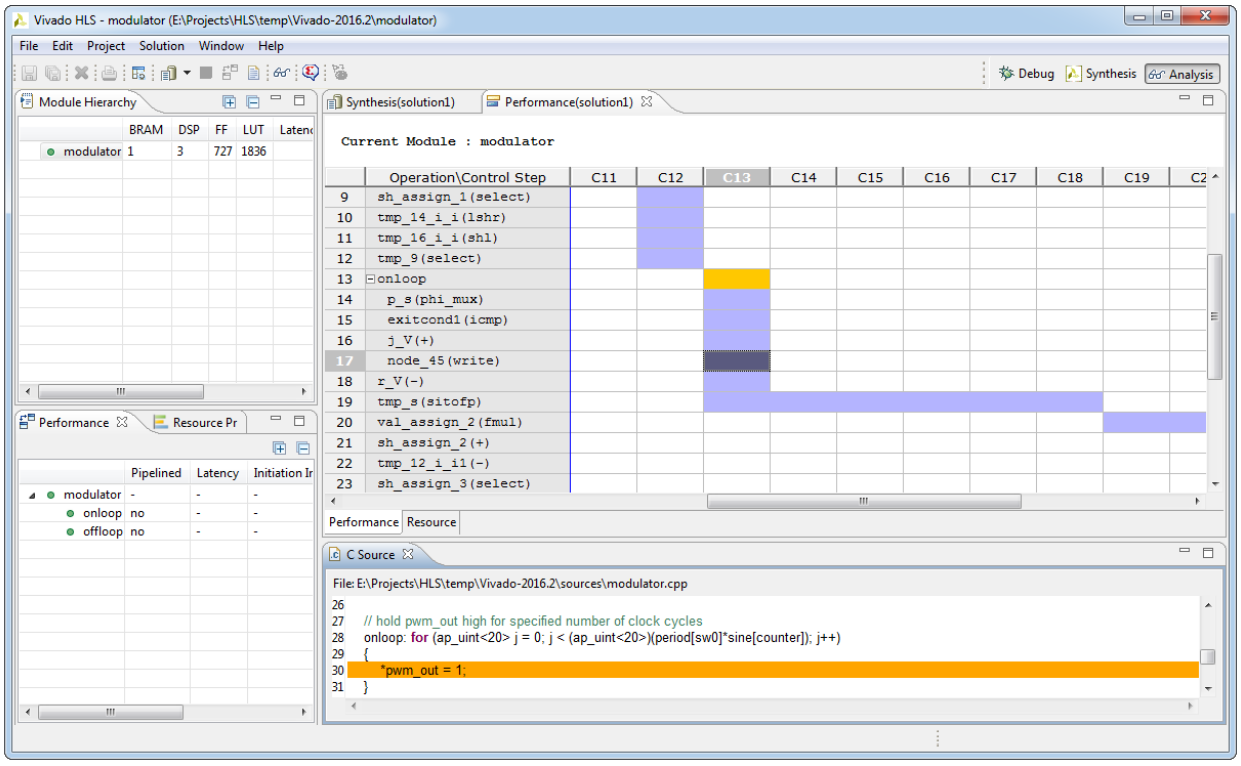


Figure 2.31: C Source Code Correlation

The Analysis Perspective also allows you to analyze resource usage. The following figure shows the **Resource profile** and the **Resource** panes.

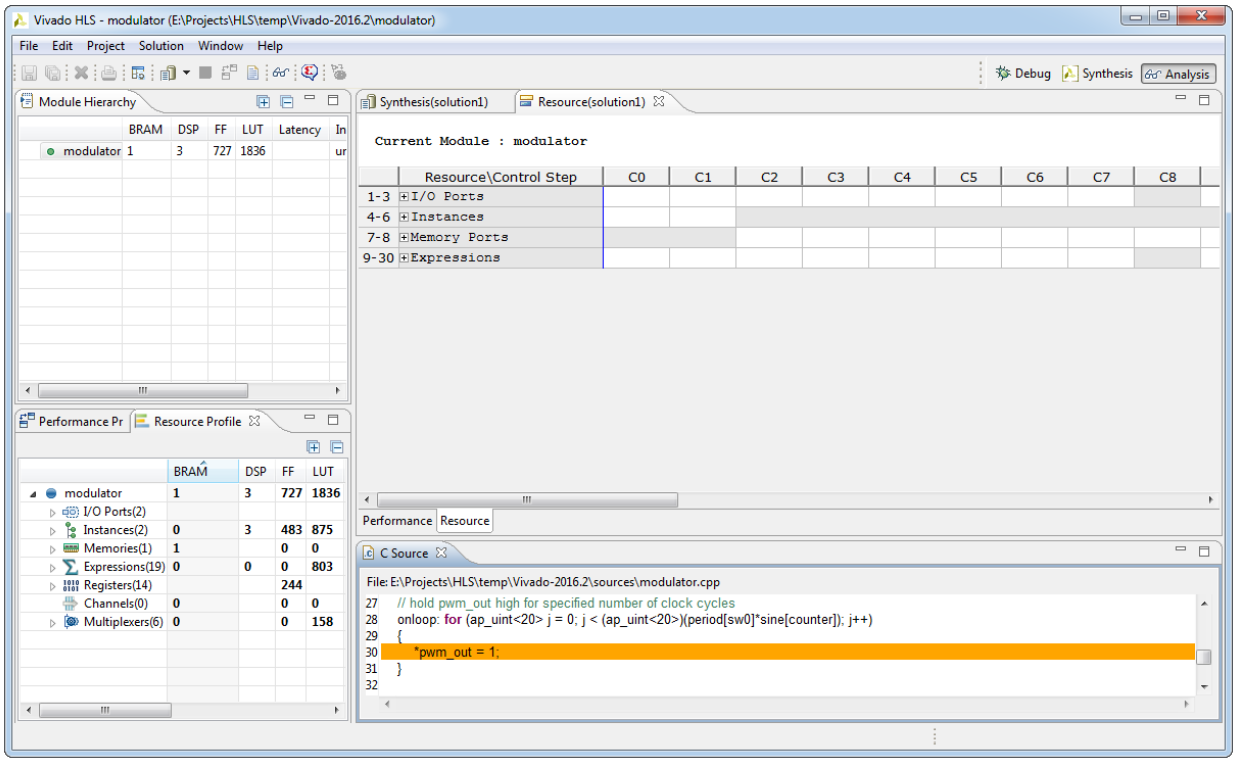


Figure 2.32: Analysis Perspective with Resource Profile

The **Resource Profile** pane shows the resources used at this level of hierarchy. In this example, you can see that all of the DSP resources are used by the two instances (***modulator_fmul_32ns_32ns_32_4_max_dsp_U0*** and ***modulator_sitofp_64ns_32_6_U1***): blocks that are instantiated inside this block, see Figure 2.33.

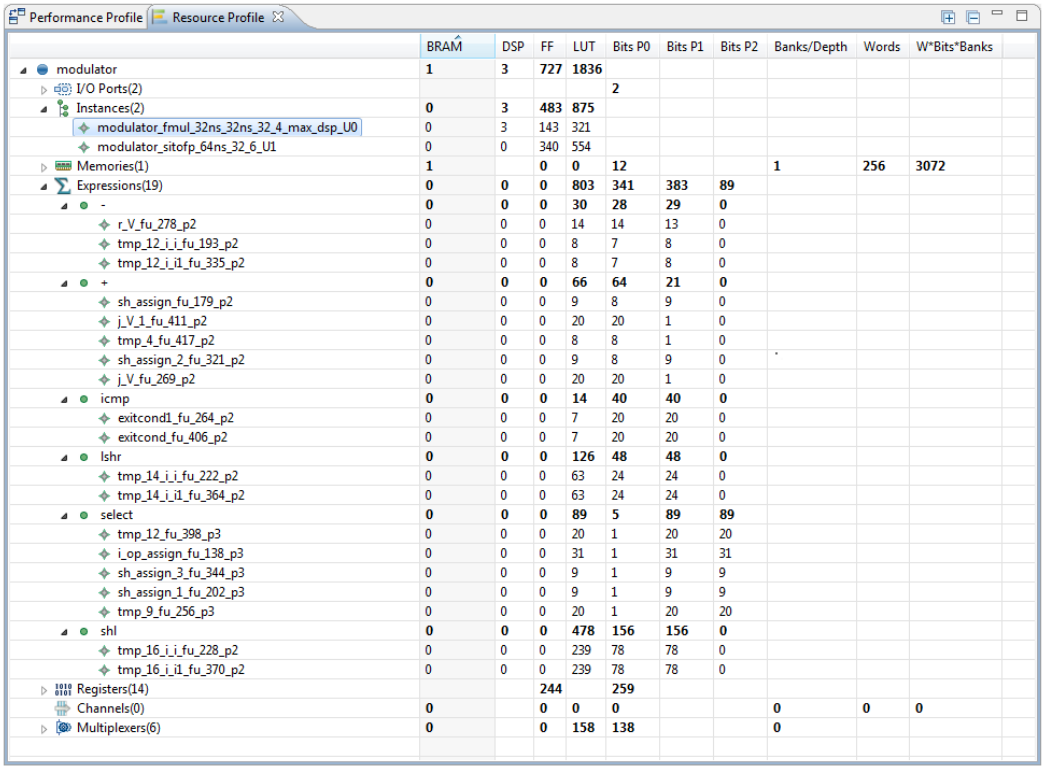


Figure 2.33: Resource Profile pane - Instances and Expressions sections

You can see by expanding the **Expressions** section that the resources at this level of hierarchy are used to implement 3 subtractors, 5 adders, 2 comparators, 2 shift right operators, 5 select operators and 2 shift left operators.

The **Resource** pane shows the control state of the operations used, see Figure 2.34. In this example, all the adder operations are associated with a different adder resource. There is no sharing of the adders. More than one add operation on each horizontal line indicates the same resource is used multiple times in different states or clock cycles.

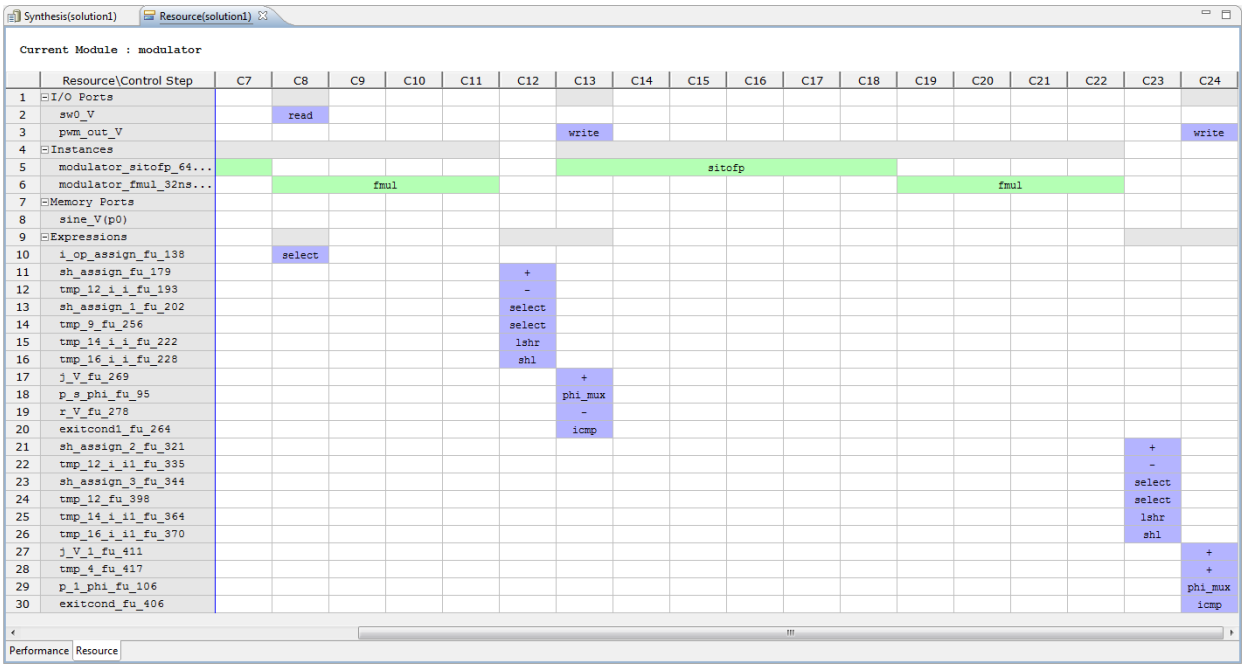


Figure 2.34: Resource pane

The Analysis Perspective is a highly interactive feature. More information on the Analysis Perspective can be found in the *Design Analysis* section of the Vivado Design Suite Tutorial, *"High-Level Synthesis (UG871)"*.

Note: Even if a Tcl flow is used to create designs, the project can still be opened in the GUI and the Analysis Perspective used to analyze the design.

Use the Synthesis perspective button to return to the synthesis view.

Generally after design analysis you can create a new solution to apply optimization directives. Using a new solution for this allows the different solutions to be compared.

2.4.3 Clock, Reset, and RTL Output

The most typical use of Vivado HLS is to create an initial design, then perform optimizations to meet the desired area and performance goals. Solutions offer a convenient way to ensure the results from earlier synthesis runs can be both preserved and compared.

Step 1. In the Vivado HLS main toolbar press **New Solution** button to open the new **Solution Configuration** dialog box, see Figure 2.35.

Figure 2.35: New Solution button

The another way to open **Solution Configuration** dialog box is to use **Project ->** **New Solution** option from the main Vivado HLS menu, see Figure 2.36.

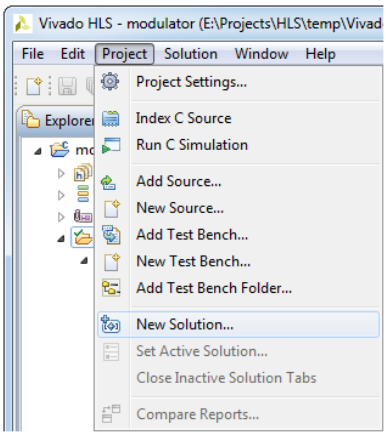


Figure 2.36: New Solution option

The **Solution Wizard** has the same options as the final window in the **New Project** wizard (Figure 2.11) plus an additional option that allow any directives and customs constraints applied to an existing solution to be conveniently copied to the new solution, where they can be modified or removed.

Step 2. In the **Solution Configuration** dialog box, leave all parameters unchanged and click **Finish**, as it is shown on the Figure 2.37.

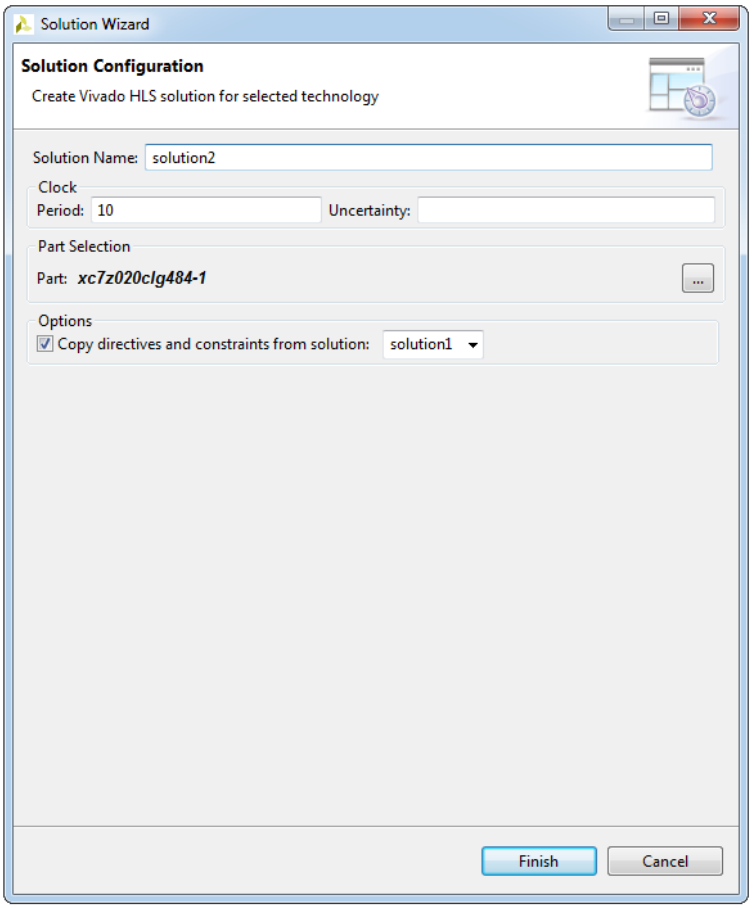


Figure 2.37: Solution Configuration dialog box

After the new solution has been created, optimization directives can be added (or modified if they were copied from the previous solution). The next section explains how directives can be added to solutions. Custom constraints are applied using the configuration options.

2.4.4 Applying Optimization Directives

The first step in adding optimization directives is to open the source code in the **Information** pane. As shown in the following figure, expand the **Source** container located at the top of the **Explorer** pane, and double-click the source file (***modulator.cpp***) to open it for editing in the **Information** pane.

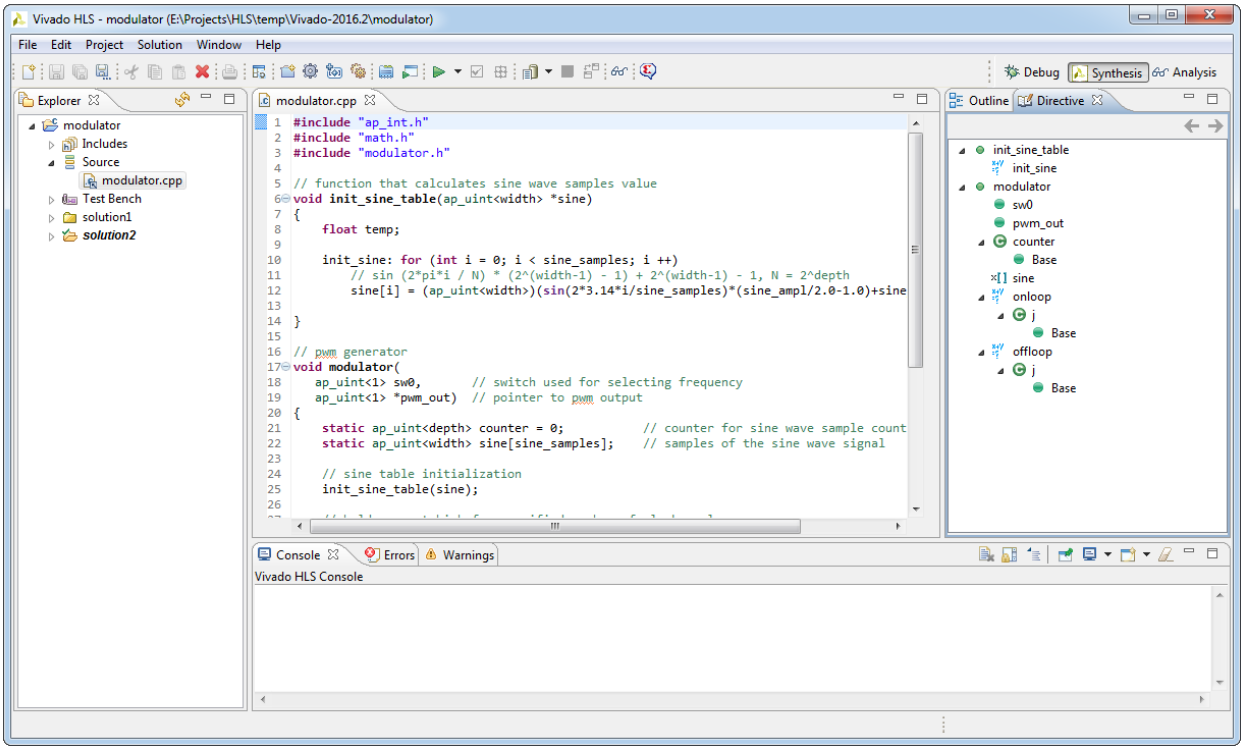


Figure 2.38: Information pane with opened source code

With the source code active in the **Information** pane, select the **Directive** tab on the right to display and modify directives for the file. The **Directive** tab contains all the objects and scopes in the currently opened source code to which you can apply directives.

Note: To apply directives to objects in other C files, you must open the file and make it active in the **Information** pane.

Although you can select objects in the Vivado HLS GUI and apply directives. Vivado HLS applies all directives to the scope that contains the object. For example, you can apply an INTERFACE directive to an interface object in the Vivado HLS GUI. Vivado HLS applies the directive to the top-level function (scope), and the interface port (object) is identified in the directive. In the following example, port *data_in* on function *foo* is specified as an AXI4-Lite interface:

```
set_directive_interface -mode s_axilite "foo" adata_in
```

You can apply optimization directives to the following objects and scopes:

- **Interfaces**

  When you apply directives to an interface, Vivado HLS applies the directive to the top-level function, because the top-level function is the scope that contains the interface.

- **Functions**

  When you apply directives to functions, Vivado HLS applies the directive to all objects within the scope of the function. The effect of any directive stops at the next level of function hierarchy. The only exception is a directive that supports or uses a recursive option, such as the PIPELINE directive that recursively unrolls all loops in the hierarchy.

- **Loops**

  When you apply directives to loops, Vivado HLS applies the directive to all objects within the scope of the loop. For example, if you apply a LOOP_MERGE directive to a loop, Vivado HLS applies the directive to any sub-loops within the loop but not to the loop itself.

Note: The loop to which the directive is applied is not merged with siblings at the same level of hierarchy.

- **Arrays**

  When you apply directives to arrays, Vivado HLS applies the directive to the scope that contains the array.

- **Regions**

  When you apply directives to regions, Vivado HLS applies the directive to the entire scope of the region. A region is any area enclosed within two braces. For example:

```
{
the scope between these braces is a region
}
```

Note: You can apply directives to a region in the same way you apply directives to functions and loops.

Step 1. To apply a directive, select an object in the **Directive** tab (in our case, ***sw0***), right-click on it and choose **Insert Directive...** option to open the **Vivado HLS Directives Editor** dialog box, see Figure 2.39.

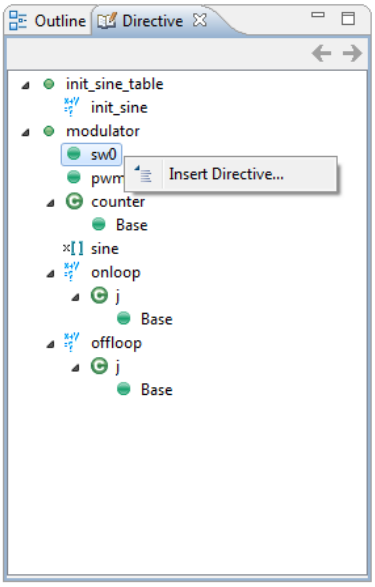


Figure 2.39: Insert Directive option

Step 2. In the **Vivado HLS Directives Editor** dialog box click on the **Directive** drop-down menu and select the appropriate directive, see Figure 2.40.

The drop-down menu shows only directives that you can add to the selected object or scope. For example, if you select an array object, the drop-down menu does not show the PIPELINE directive, because an array cannot be pipelined.

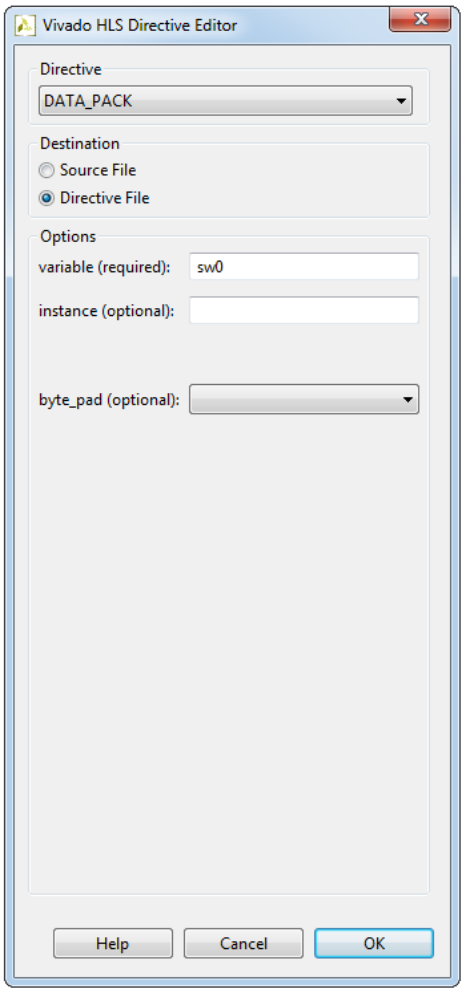


Figure 2.40: Vivado HLS Directives Editor dialog box

In the **Vivado HLS Directive Editor** dialog box, you can specify either of the following **Destination** settings:

- **Source File** - Vivado HLS inserts the directive directly into the C source file as a pragma.
- **Directive File** - Vivado HLS inserts the directive as a Tcl command into the file *directives.tcl* in the solution directory.

The following table describes the advantages and disadvantages of both approaches.

Table 2.2: Tcl Commands vs Pragmas

Directive Format	Advantages	Disadvantages
Directives file (Tcl Command)	Each solution has independent directives. This approach is ideal for design exploration. If any solution is re-synthesized, only the directives specified in that solution are applied.	If the C source files are transferred to a third-party or archived, the *directives.tcl* file must be included. The *directives.tcl* file is required if the results are to be re-created.
Source Code (Pragma)	The optimization directives are embedded into the C source code. Ideal when the C sources files are shipped to a third-party as C IP. No other files are required to recreate the same results. Useful approach for directives that are unlikely to change, such as TRIPCOUNT and INTERFACE.	If the optimization directives are embedded in the code, they are automatically applied to every solution when re-synthesized.

Step 3. In the **Vivado HLS Directive Editor** dialog box:

- choose **INTERFACE** as a directive for *sw0* input port in the **Directive** drop-down list
- leave selected **Directive File** as a **Destination**
- choose **ap_none** I/O protocol as a **mode (optional)** option in the **Options** section
- leave all other parameters unchanged and
- click **OK**, see Figure 2.41.

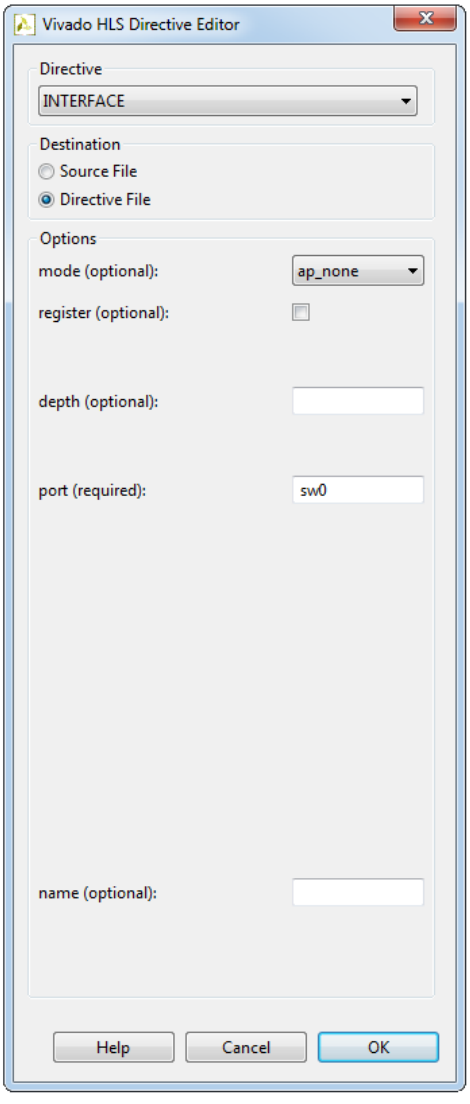


Figure 2.41: Vivado HLS Directives Editor dialog box with necessary settings

Step 4. Apply the same directive with the same settings to the ***pwm_out*** output port and the **Directive** tab with applied directives to selected ports looks as it is shown on the Figure 2.42.

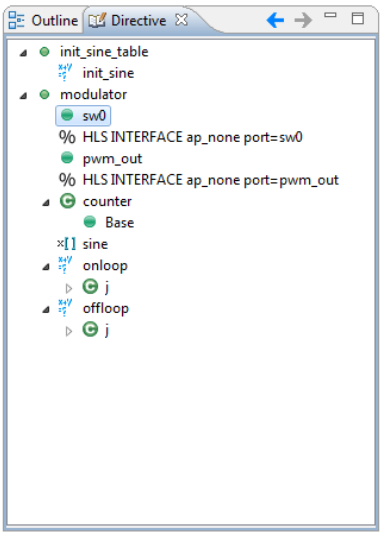


Figure 2.42: Directive tab with applied directives

Step 5. After having applied all necessary directives, run **C Synthesis** process by pressing the **C Synthesis** button (green arrow), shown on the Figure 2.20.

In the following table is presented the complete list of all optimization directives provided by Vivado HLS.

Table 2.3: Vivado HLS Optimization Directives

Directive Format	Advantages
ALLOCATION	Specify a limit for the number of operations, cores or functions used. This can force the sharing or hardware resources and may increase latency.
ARRAY_MAP	Combines multiple smaller arrays into a single large array to help reduce block RAM resources.
ARRAY_PARTITION	Partitions large arrays into multiple smaller arrays or into individual registers, to improve access to data and remove block RAM bottlenecks.
ARRAY_RESHAPE	Reshape an array from one with many elements to one with greater word-width. Useful for improving block RAM accesses without using more block RAM.
DATA_PACK	Packs the data fields of a struct into a single scalar with a wider word width.
DATAFLOW	Enables task level pipelining, allowing functions and loops to execute concurrently. Used to minimize interval.
DEPENDENCE	Used to provide additional information that can overcome loop-carry dependencies and allow loops to be pipelined (or pipelined with lower intervals).
EXPRESSION_BALANCE	Allows automatic expression balancing to be turned off.
FUNCTION_INSTANTIATE	Allows different instances of the same function to be locally optimized.
INLINE	Inlines a function, removing all function hierarchy. Used to enable logic optimization across function boundaries and improve latency/interval by reducing function call overhead.

INTERFACE	Specifies how RTL ports are created from the function description.
LATENCY	Allows a minimum and maximum latency constraint to be specified.
LOOP_FLATTEN	Allows nested loops to be collapsed into a single loop with improved latency.
LOOP_MERGE	Merge consecutive loops to reduce overall latency, increase sharing and improve logic optimization.
LOOP_TRIPCOUNT	Used for loops which have variables bounds. Provides an estimate for the loop iteration count. This has no impact on synthesis, only on reporting.
OCCURRENCE	Used when pipelining functions or loops, to specify that the code in a location is executed at a lesser rate than the code in the enclosing function or loop.
PIPELINE	Reduces the initiation interval by allowing the concurrent execution of operations within a loop or function.
PROTOCOL	This commands specifies a region of the code to be a protocol region. A protocol region can be used to manually specify an interface protocol.
RESET	This directive is used to add or remove reset on a specific state variable (global or static).
RESOURCE	Specify that a specific library resource (core) is used to implement a variable (array, arithmetic operation or function argument) in the RTL.
STREAM	Specifies that a specific array is to be implemented as a FIFO or RAM memory channel during dataflow optimization.
UNROLL	Unroll for-loops to create multiple independent operations rather than a single collection of operations.

Applying Optimization Directives to Global Variables

Directives can only be applied to scopes or objects within a scope. As such, they cannot be directly applied to global variables which are declared outside the scope of any function.

To apply a directive to a global variable, apply the directive to the scope (function, loop or region) where the global variable is used. Open the directives tab on a scope were the variable is used, apply the directive and enter the variable name manually in Directives Editor.

Applying Optimization Directives to Class Objects

Optimization directives can be also applied to objects or scopes defined in a class. The difference is typically that classes are defined in a header file. Use one of the following actions to open the header file:

- From the **Explorer** pane, open the **Includes** folder, navigate to the header file, and double-click the file to open it.
- From within the C source, place the cursor over the header file (the *#include* statement), to open hold down the **Ctrl** key, and click the header file.

The directives tab is then populated with the objects in the header file and directives can be applied.

Important: Care should be taken when applying directives as pragmas to a header file. The file might be used by other people or used in other projects. Any directives added as a pragma are applied each time the header file is included in a design.

Applying Optimization Directives to Templates

To apply optimization directives manually on templates when using Tcl commands, specify the template arguments and class when referring to class methods. For example, given the following C++ code:

```
template <uint32 SIZE, uint32 RATE>
void DES10<SIZE,RATE>::calcRUN() {...}
```

The following Tcl command is used to specify the INLINE directive on the function:

```
set_directive_inline DES10<SIZE,RATE>::calcRUN
```

The following section outlines the various optimizations and techniques you can use to direct Vivado HLS to produce a micro-architecture that satisfies the desired performance and area goals.

2.4.4.1 Clock, Reset, and RTL Output

Clock Frequency

For C and C++ designs only a single clock is supported. The same clock is applied to all functions in the design.

For SystemC designs, each SC_MODULE may be specified with a different clock. To specify multiple clocks in a SystemC design, use the -name option of the *create_clock* command to create multiple named clocks and use the CLOCK directive or pragma to specify which function contains the SC_MODULE to be synthesized with the specified clock. Each SC_MODULE can only be synthesized using a single clock. Clocks may be distributed through functions, such as when multiple clocks are connected from the top-level ports to individual blocks, but each SC_MODULE can only be sensitive to a single clock.

The clock period, in ns, is set in the **Solution ->** **Solution Settings...** (main Vivado HLS menu option). Vivado HLS uses the concept of a clock uncertainty to provide a user defined timing margin. Using the clock frequency and device target information Vivado HLS estimates the timing of operations in the design but it cannot know the final component placement and net routing: these operations are performed by logic synthesis of the output RTL. As such, Vivado HLS cannot know the exact delays.

To calculate the clock period used for synthesis, Vivado HLS subtracts the clock uncertainty from the clock period, as shown in the following figure.

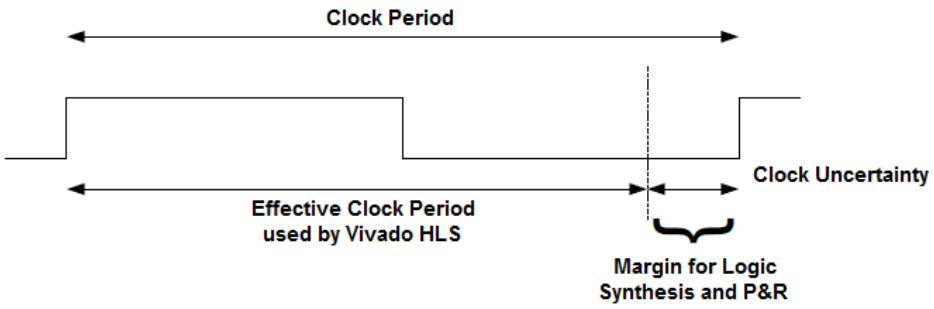


Figure 2.43: Clock Period and Margin

This provides a user specified margin to ensure downstream processes, such as logic synthesis and place & route, have enough timing margin to complete their operations. If the FPGA device is mostly utilized the placement of cells and routing of nets to connect the cells might not be ideal and might result in a design with larger than expected timing delays. For a situation such as this, an increased timing margin ensures Vivado HLS does not create a design with too much logic packed into each clock cycle and allows RTL synthesis to satisfy timing in cases with less than ideal placement and routing options.

By default, the clock uncertainty is 12.5% of the cycle time. The value can be explicitly specified beside the clock period.

Vivado HLS aims to satisfy all constraints: timing, throughput, latency. However, if a constraints cannot be satisfied, Vivado HLS always outputs an RTL design.

If the timing constraints inferred by the clock period cannot be met Vivado HLS issues message SCHED-644, as shown below, and creates a design with the best achievable performance.

```
@W [SCHED-644] Max operation delay (<operation_name> 2.39ns) exceeds the effective cycle time
```

Even if Vivado HLS cannot satisfy the timing requirements for a particular path, it still achieves timing on all other paths. This behavior allows you to evaluate if higher optimization levels or special handling of those failing paths by downstream logic syntheses can pull-in and ultimately satisfy the timing.

Important: It is important to review the constraint report after synthesis to determine if all constraints is met. The fact that Vivado HLS produces an output design does not guarantee the design meets all performance constraints. Review the *"Performance Estimates"* section of the design report.

The option *relax_ii_for_timing* of the *config_schedule* command can be used to change the default timing behavior. When this option is specified, Vivado HLS automatically relaxes the II for any pipeline directive when it detects a path is failing to meet the clock period. This option only applies to cases where the PIPELINE directive is specified without an II value (and an II=1 is implied). If the II value is explicitly specified in the PIPELINE directive, the *relax_ii_for_timing* option has no effect.

A design report is generated for each function in the hierarchy when synthesis completes and can be viewed in the solution reports folder. The worse case timing for the entire design is reported as the worst case in each function report. There is no need to review every report in the hierarchy.

If the timing violations are too severe to be further optimized and corrected by downstream processes, review the techniques for specifying an exact latency and specifying exact implementation cores before considering a faster target technology.

Reset

Typically the most important aspect of RTL configuration is selecting the reset behavior. When discussing reset behavior it is important to understand the difference between initialization and reset.

Initialization Behavior

In C, variables defined with the static qualifier and those defined in the global scope, are by default initialized to zero. Optionally, these variables may be assigned a specific initial value. For these type of variables, the initial value in the C code is assigned at compile time (at time zero) and never again. In both cases, the same initial value is implemented in the RTL.

- During RTL simulation the variables are initialized with the same values as the C code.
- The same variables are initialized in the bitstream used to program the FPGA. When the device powers up, the variables will start in their initialized state.

The variables start with the same initial state as the C code. However, there is no way to force a return to this initial state. To return to their initial state the variables must be implemented with a reset.

Controlling the Reset Behavior

The reset port is used in an FPGA to return the registers and block RAM connected to the reset port to an initial value any time the reset signal is applied. The presence and behavior of the RTL reset port is controlled using the *config_rtl* configuration.

To access the *config_rtl* configuration:

- In the Vivado HLD **Explorer** pane, select **Solution2**, right-click on it and choose **Solution Settings...** option,
- In the **Solution Settings (solution2)** dialog box, select **General** option and click **Add...** button to open **RTL Configurations** dialog box,
- In the **RTL Configurations** dialog box click the **Command** drop down list and choose **config_rtl** command,
- Leave all other settings unchanged and click **OK**,
- In the **Solution Settings (solution2)** dialog box, clok **OK**.

Important: In our design, we do not need to use reset port, so this *config_rtl* configuration is not needless for our design!

The reset settings include the ability to set the polarity of the reset and whether the reset is synchronous or asynchronous but more importantly it controls, through the reset option, which registers are reset when the reset signal is applied.

Important: When AXI4 interfaces are used on a design the reset polarity is automatically changed to active-Low irrespective of the setting in the *config_rtl* configuration. This is required by the AXI4 standard.

The reset option has four settings:

- **none** - No reset is added to the design.
- **control** - This is the default and ensures all control registers are reset. Control registers are those used in state machines and to generate I/O protocol signals. This setting ensures the design can immediately start its operation state.

- **state** - This option adds a reset to control registers (as in the control setting) plus any registers or memories derived from static and global variables in the C code. This setting ensures static and global variable initialized in the C code are reset to their initialized value after the reset is applied.
- **all** - This adds a reset to all registers and memories in the design.

Finer grain control over reset is provided through the RESET directive. If a variable is a static or global, the RESET directive is used to explicitly add a reset, or the variable can be removed from those being reset by using the RESET directive's *off* option. This can be particularly useful when static or global arrays are present in the design.

Initializing and Resetting Arrays

Arrays are often defined as static variables, which implies all elements be initialized to zero, and arrays are typically implemented as block RAM. When reset options *state* or *all* are used, it forces all arrays implemented as block RAM to be returned to their initialized state after reset. This may result in two very undesirable attributes in the RTL design:

- Unlike a power-up initialization, an explicit reset requires the RTL design iterate through each address in the block RAM to set the value: this can take many clock cycles if N is large and require more area resources to implement.
- A reset is added to every array in the design.

To prevent placing reset logic onto every such block RAM and incurring the cycle overhead to reset all elements in the RAM:

- Use the default control reset mode and use the RESET directive to specify individual static or global variables to be reset.
- Alternatively, use reset mode state and remove the reset from specific static or global variables using the off option to the RESET directive.

RTL Output

Various characteristics of the RTL output by Vivado HLS can be controlled using the *config_rtl* configuration:

- Specify the type of FSM encoding used in the RTL state machines.
- Add an arbitrary comment string, such as a copyright notice, to all RTL files using the *-header* option.
- Specify a unique name with the prefix option which is added to all RTL output file names.
- Force the RTL ports to use lower case names.

The default FSM coding is style is *onehot*. Other possible options are *auto*, *binary*, and *gray*. If you select auto, Vivado HLS implements the style of encoding using the onehot default, but Vivado Design Suite might extract and re-implement the FSM style during logic synthesis. If you select any other encoding style (*binary*, *onehot*, *gray*), the encoding style *cannot* be re-optimized by Xilinx logic synthesis tools.

The names of the RTL output files are derived from the name of the top-level function for synthesis. If different RTL blocks are created from the same top-level function, the RTL files will have the same name and cannot be combined in the same RTL project. The *prefix* option allows RTL files generated from the same top-level function (and which by default have the same name as the top-level function) to be easily combined in the same directory. The *lower_case_name* option ensures the only lower case names are used in the output RTL. This option ensures the IO protocol ports created by Vivado HLS, such as those for AXI interfaces, are specified as *s_axis_<port>_tdata* in the final RTL rather than the default port name of *s_axis_<port>_TDATA*.

2.4.4.2 Optimizing for Throughput

Use the following optimizations to improve throughput or reduce the initiation interval.

Task Pipelining

Pipelining allows operations to happen concurrently. The task does not have to complete all operations before it begin the next operation. Pipelining is applied to functions and loops. The throughput improvements in function pipelining are shown in the following figure.

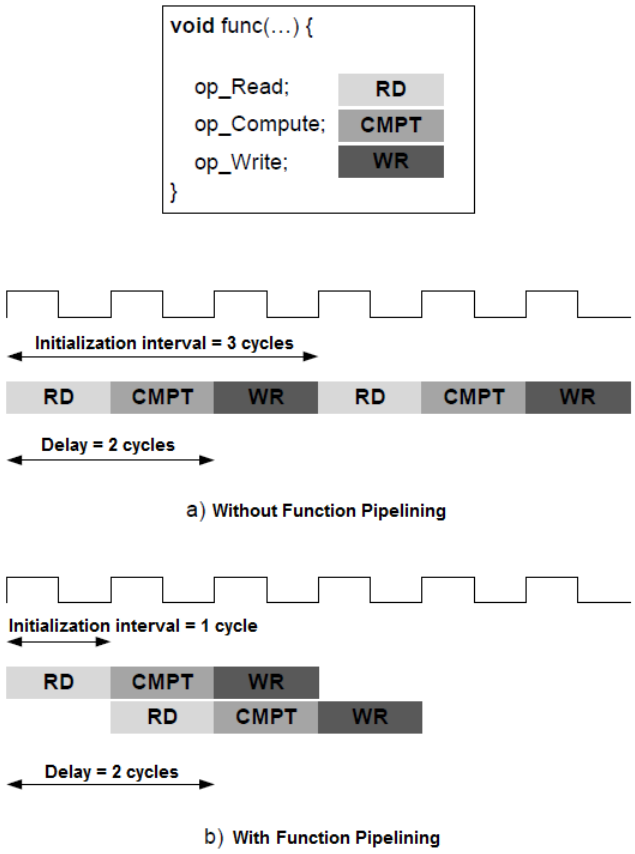


Figure 2.44: Function Pipelining Behavior

Without pipelining the function reads an input every 3 clock cycles and outputs a value every 2 clock cycles. The function has an Initiation Interval (II) of 3 and a latency of 2. With pipelining, a new input is read every cycle (II=1) with no change to the output latency or resources used.

Loop pipelining allows the operations in a loop to be implemented in a concurrent manner as shown in the following figure. In this figure, (a) shows the default sequential operation where there are 3 clock cycles between each input read (II=3), and it requires 8 clock cycles before the last output write is performed.

In the pipelined version of the loop shown in (b), a new input sample is read every cycle (II=1) and the final output is written after only 4 clock cycles: substantially improving both the II and latency while using the same hardware resources.

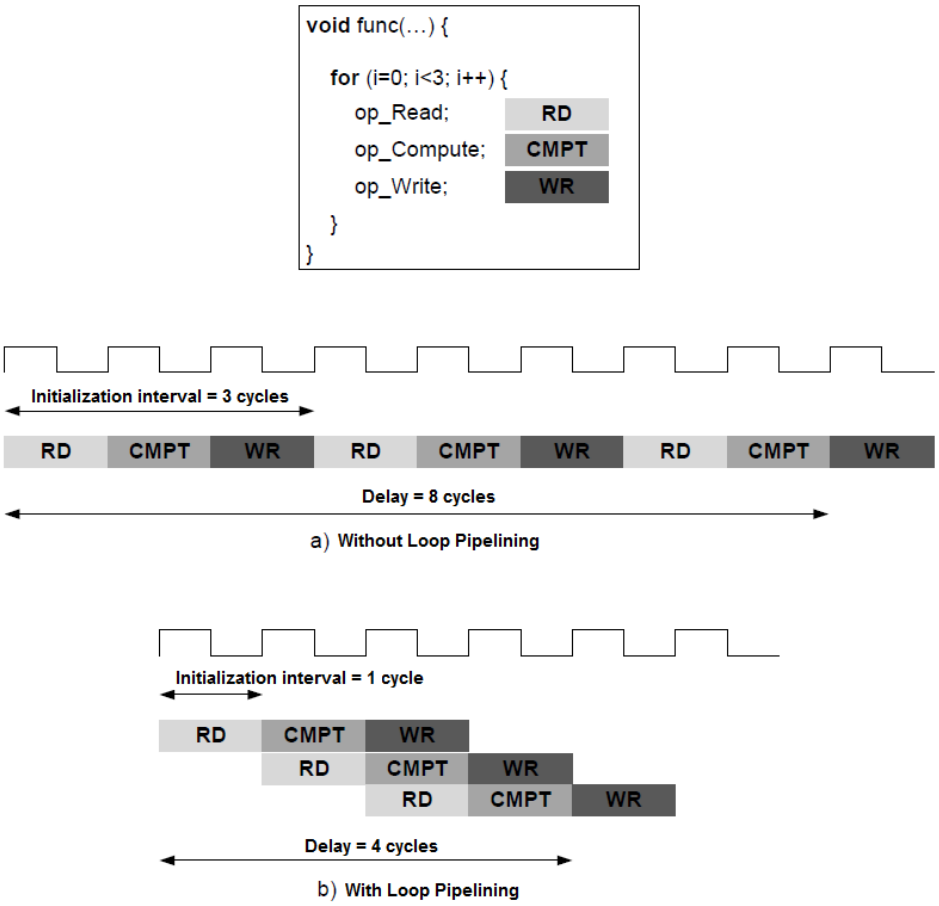


Figure 2.45: Loop Pipelining Behavior

Tasks are pipelined using the PIPELINE directive. The initiation interval defaults to 1 if not specified but may be explicitly specified.

Pipelining is applied to the specified task not to the hierarchy below: all loops in the hierarchy below are automatically unrolled. Any sub-functions in the hierarchy below the specified task must be pipelined individually. If the sub-functions are pipelined, the pipelined tasks above it can take advantage of the pipeline performance. Conversely, any sub-function below the pipelined task that is not pipelined, may be the limiting factor in the performance of the pipeline.

There is a difference in how pipelined functions and loops behave:

- In the case of functions, the pipeline runs forever and never ends.
- In the case of loops, the pipeline executes until all iterations of the loop are completed.

Partitioning Arrays to Improve Pipelining

Pipelining increases the throughput of the system, but sometimes existing data interface do not have sufficient data throughput to transmit all the necessary data to the data processing system. In this case pipelining system works under their possibilities and pipelining effects of the limited. This issue is typically caused by arrays. Arrays are implemented as block RAM which only has a maximum of two data ports. This can limit the throughput of a read/write (or load/store) intensive algorithm. The bandwidth can be improved by splitting the array (a single block RAM resource) into multiple smaller arrays (multiple block RAMs), effectively increasing the number of ports.

Arrays are partitioned using the ARRAY_PARTITION directive. Vivado HLS provides three types of array partitioning, as shown in the following figure. The three styles of partitioning are:

- **block** - The original array is split into equally sized blocks of consecutive elements of the original array.
- **cyclic** - The original array is split into equally sized blocks interleaving the elements of the original array.

- **complete** - The default operation is to split the array into its individual elements. This corresponds to resolving a memory into registers.

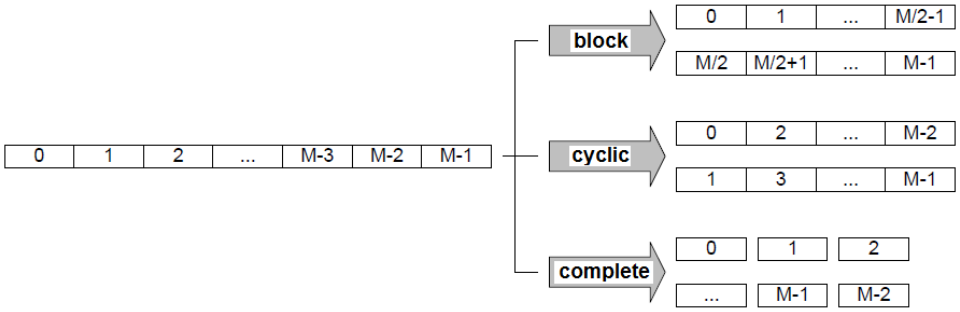


Figure 2.46: Array Partitioning

For block and cyclic partitioning the factor option specifies the number of arrays that are created. In the preceding figure, a factor of 2 is used, that is, the array is divided into two smaller arrays. If the number of elements in the array is not an integer multiple of the factor, the final array has fewer elements.

When partitioning multi-dimensional arrays, the *dimension* parameter is used to specify which dimension is partitioned. The following code shows how the *dimension* parameter is used to partition the following example code:

```
void example (...) {
    int my_array[10][6][4];
    ...
}
```

The example demonstrates how partitioning dimension 3 results in 4 separate arrays and partitioning dimension 1 results in 10 separate arrays. If zero is specified as the dimension, all dimensions are partitioned.

```
my_array[10][6][4] -> ARRAY_PARTITION, mode=compete, partition dimension = 3 -> my_array_0[10][6]
                                                                                my_array_1[10][6]
                                                                                my_array_2[10][6]
                                                                                my_array_3[10][6]

my_array[10][6][4] -> ARRAY_PARTITION, mode=compete, partition dimension = 1 -> my_array_0[6][4]
                                                                                my_array_1[6][4]
                                                                                my_array_2[6][4]
                                                                                my_array_3[6][4]
                                                                                my_array_4[6][4]
                                                                                my_array_5[6][4]
                                                                                my_array_6[6][4]
                                                                                my_array_7[6][4]
                                                                                my_array_8[6][4]
                                                                                my_array_9[6][4]

my_array[10][6][4] -> ARRAY_PARTITION, mode=compete, partition dimension = 0 -> 10x6x4=240 registers
```

The *config_array_partition* configuration determines how arrays are automatically partitioned based on the number of elements. This configuration is accessed through the Vivado HLS menu **Solution ->** **Solution Settings ->** **General ->** **Add ->** ***config_array_partition***.

The partition thresholds can be adjusted and partitioning can be fully automated with the *throughput_driven* option. When the *throughput_driven* option is selected Vivado HLS automatically partitions arrays to achieve the specified throughput.

Loop Unrolling to Improve Pipelining

By default loops are kept rolled in Vivado HLS. That is to say that the loops are treated as a single entity: all operations in the loop are implemented using the same hardware resources for iteration of the loop.

Vivado HLS provides the ability to unroll or partially unroll for-loops using the UNROLL directive.

The following figure shows both the powerful advantages of loop unrolling and the implications that must be considered when unrolling loops. This example assumes the arrays *a[i]*, *b[i]* and *c[i]* are mapped to block RAMs. This example shows how easy it is to create many different implementations by the simple application of loop unrolling.

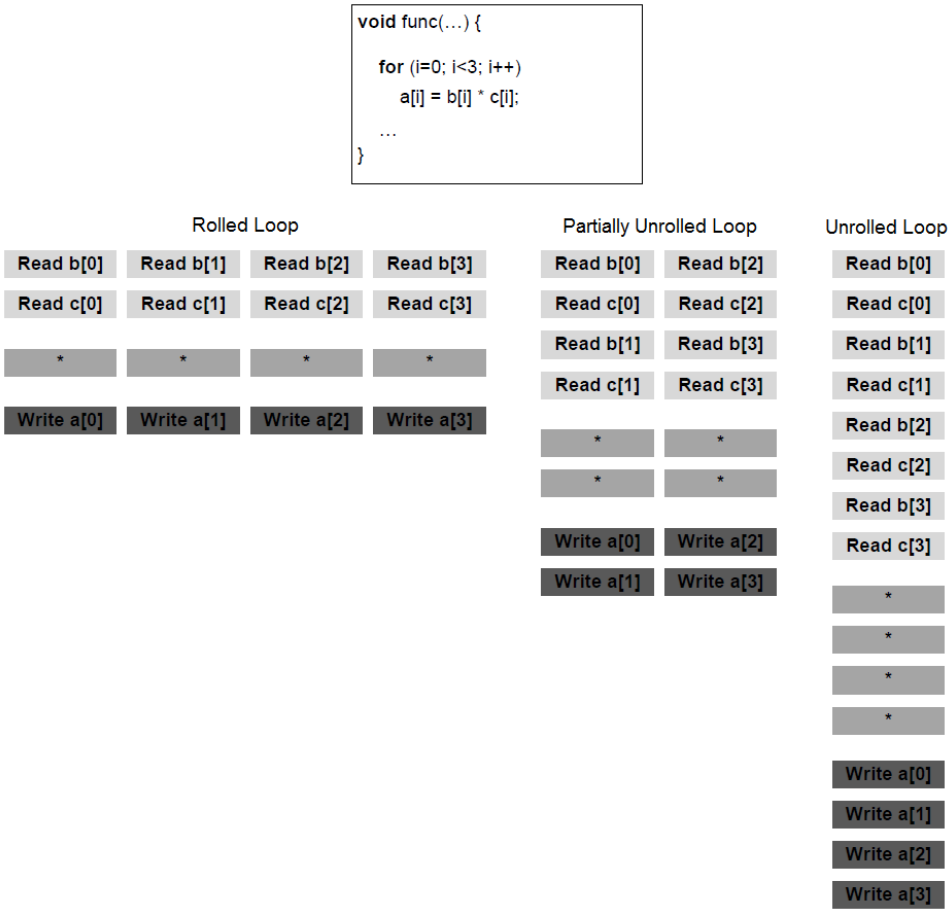


Figure 2.47: Loop Unrolling Details

- **Rolled Loop** - When the loop is rolled, each iteration is performed in a separate clock cycle. This implementation takes four clock cycles, only requires one multiplier and each block RAM can be a single-port block RAM.
- **Partially Unrolled Loop** - In this example, the loop is partially unrolled by a factor of 2. This implementation required two multipliers and dual-port RAMs to support two reads or writes to each RAM in the same clock cycle. This implementation does however only take 2 clock cycles to complete: half the initiation interval and half the latency of the rolled loop version.
- **Unrolled Loop** - In the fully unrolled version all loop operation can be performed in a single clock cycle. This implementation however requires four multipliers. More importantly, this implementation requires the ability to perform 4 reads and 4 write operations in the same clock cycle. Because a block RAM only has a maximum of two ports, this implementation requires the arrays be partitioned.

To perform loop unrolling, you can apply the UNROLL directives to individual loops in the design. Alternatively, you can apply the UNROLL directive to a function, which unrolls all loops within the scope of the function.

If a loop is completely unrolled, all operations will be performed in parallel: if data dependencies allow. If operations in one iteration of the loop require the result from a previous iteration, they cannot execute in parallel but will execute as soon as the data is available. A completely unrolled loop will mean multiple copies of the logic in the loop body.

Partial loop unrolling does not require the unroll factor to be an integer multiple of the maximum iteration count. Vivado HLS adds an exit checks to ensure partially unrolled loops are functionally identical to the original loop. For example, given the following code:

```
for(int i = 0; i < N; i++) {
    a[i] = b[i] + c[i];
}
```

Loop unrolling by a factor of 2 effectively transforms the code to look like the following example where the *break* construct is used to ensure the functionality remains the same:

```
for(int i = 0; i < N; i++) {
    a[i] = b[i] + c[i];
    if(i+1>=N) break;
        a[i+1]=b[i+1]+c[i+1];
}
```

Because N is a variable, Vivado HLS may not be able to determine its maximum value (it could be driven from an input port). If you know the unrolling factor, 2 in this case, is an integer factor of the maximum iteration count N, the *skip_exit_check* option removes the exit check and associated logic. The effect of unrolling can now be represented as:

```
for(int i = 0; i < N; i ++) {
    a[i] = b[i] + c[i];
    a[i+1] = b[i+1] + c[i+1];
}
```

This helps minimize the area and simplify the control logic.

2.4.4.3 Optimizing for Latency

In order to reduce delays in the data processing (latency) within RTL system, that is the result of the HLS synthesis using Vivad HLS tool, it is necessary to use the following optimization directives:

- Latency Constraints
- Loop Merging
- Loop Flattening

Latency Constraints

Vivado HLS supports the use of a latency constraint on any scope. Latency constraints are specified using the LATENCY directive.

When a maximum and/or minimum LATENCY constraint is placed on a scope, Vivado HLS tries to ensure all operations in the function complete within the range of clock cycles specified.

The LATENCY directive applied to a loop specifies the required latency for a single iteration of the loop. It specifies the latency for the loop body, as the following examples shows:

```
for (int i=0; i<N; i++) {
    #pragma HLS latency max=10
    ..Loop Body...
}
```

This example contains LATENCY directive which specifies that the maximum duration of the body loop execution is not greater than 10 cycles clock signal.

If the intention is to limit the total latency of all loop iterations, the latency directive should be applied to a region that encompasses the entire loop, as in this example:

```
Region_Loop: {
    #pragma HLS latency max=10
    for (int i=0; i<N; i++)
    {
        ..Loop Body...
    }
}
```

In this case, even if the loop is unrolled, the latency directive sets a maximum limit on all loop operations.

If Vivado HLS cannot meet a maximum latency constraint it relaxes the latency constraint and tries to achieve the best possible result.

If a minimum latency constraint is set and Vivado HLS can produce a design with a lower latency than the minimum required it inserts dummy clock cycles to meet the minimum latency.

Loop Merging

All rolled loops imply and create at least one state in the design FSM. When there are multiple sequential loops it can create additional unnecessary clock cycles and prevent further optimizations.

The following figure shows a simple example where a seemingly intuitive coding style has a negative impact on the performance of the RTL design.

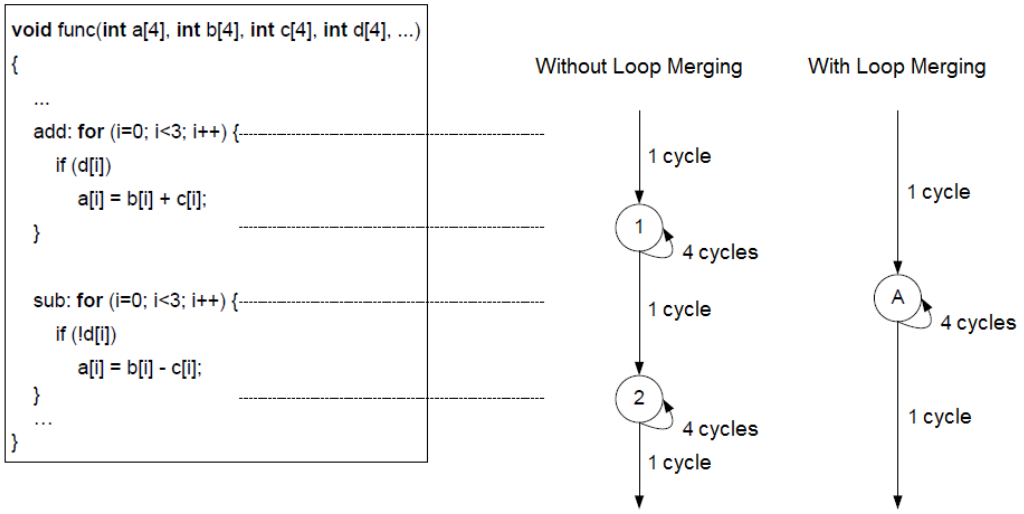


Figure 2.48: Loop Directives

On the Figure 2.48, "Without Loop Merging" shows how, by default, each rolled loop in the design creates at least one state in the FSM. Moving between those states costs clock cycles: assuming each loop iteration requires one clock cycle, it take a total of 11 cycles to execute both loops:

- 1 clock cycle to enter the *add* loop.
- 4 clock cycles to execute the *add* loop.
- 1 clock cycle to exit *add* and enter *sub*.
- 4 clock cycles to execute the *sub* loop.
- 1 clock cycle to exit the *sub* loop.
- For a total of 11 clock cycles.

In this simple example it is obvious that an else branch in the ADD loop would also solve the issue but in a more complex example it may be less obvious and the more intuitive coding style may have greater advantages.

The LOOP_MERGE optimization directive is used to automatically merge loops. The LOOP_MERGE directive will seek so to merge all loops within the scope it is placed. In the above example, merging the loops creates a control structure similar to that shown in (B) in the preceding figure, which requires only 6 clocks to complete.

Merging loops allows the logic within the loops to be optimized together. In the example above, using a dual-port block RAM allows the add and subtraction operations to be performed in parallel.

Loop Flattening

In a similar manner to the consecutive loops discussed in the previous section, it requires additional clock cycles to move between rolled nested loops. It requires one clock cycle to move from an outer loop to an inner loop and from an inner loop to an outer loop.

The following example illustrates how, if no care is taken one may spend an additional 200 clock cycles to these processes when executing external loop.

```
void func {int a, int b, int c, int d}
{
    ...
    outer_loop: while(j<100) {
        inner_loop: while(i<6) {    // 1 cycle to enter inner
            ...
            LOOP_BODY
```

```
            ...
        }                           // 1 cycle to exit inner
    }
    ...
}
```

Vivado HLS provides the set_directive_loop_flatten command to allow labeled perfect and semi-perfect nested loops to be flattened, removing the need to re-code for optimal hardware performance and reducing the number of cycles it takes to perform the operations in the loop.

- **Perfect loop nest** - only the innermost loop has loop body content, there is no logic specified between the loop statements and all the loop bounds are constant.
- **Semi-perfect loop nest** - only the innermost loop has loop body content, there is no logic specified between the loop statements but the outermost loop bound can be a variable.

For imperfect loop nests, where the inner loop has variables bounds or the loop body is not exclusively inside the inner loop, designers should try to restructure the code, or unroll the loops in the loop body to create a perfect loop nest.

2.4.4.4 Optimizing for Area

In order to reduce hardware resources needed to implement the RTL system which generates in HLS process using HSL Vivado tools, it is necessary to use the following optimization directives:

- Bit-Width Narrowing
- Function Inlining
- Array Mapping
- Array Reshaping
- Resource Allocation

Bit-Width Narrowing

The bit-widths of the variables in the C function directly impact the size of the storage elements and operators used in the RTL implementation. If a variables only requires 12-bits but is specified as an integer type (32-bit) it will result in larger and slower 32-bit operators being used, reducing the number of operations that can be performed in a clock cycle and potentially increasing initiation interval and latency.

- Use the appropriate precision for the data types.
- Confirm the size of any arrays that are to be implemented as RAMs or registers. The area impact of any over-sized elements is wasteful in hardware resources.
- Pay special attention to multiplications, divisions, modulus or other complex arithmetic operations. If these variables are larger than they need to be, they negatively impact both area and performance.

Function Inlining

Function inlining removes the function hierarchy. A function is inlined using the INLINE directive.

Inlining a function may improve area by allowing the components within the function to be better shared or optimized with the logic in the calling function. This type of function inlining is also performed automatically by Vivado HLS. Small functions are automatically inlined.

Inlining allows functions sharing to be better controlled. For functions to be shared they must be used within the same level of hierarchy. In this code example, function *top* calls f1 twice and function fsub.

```
fsub (int p, int q)
{
    int q1 = q + 10;
    f1(p1,q);    // the third instance of f1 function
    ...
}
```

```
void top {int a, int b, int c, int d}
{
    ...
    f1(a,b);    // the first instance of f1 function
    f1(a,c);    // the second instance of f1 function
    fsub(a,d);
    ...
}
```

Inlining function *fsub* and using the ALLOCATION directive to specify only 1 instance of function *fsub* is used, results in a design which only has one instance of function *fsub*: one-third the area of the example above.

```
fsub (int p, int q)
{
    #pragma HLS INLINE
    int q1 = q + 10;
    f1(p1,q);
    ...
}

void top {int a, int b, int c, int d}
{
    #pragma HLS ALLOCATION instances=f1 limit=1 function
    ...
    f1(a,b);
    f1(a,c);
    fsub(a,d);
    ...
}
```

The INLINE directive optionally allows all functions below the specified function to be recursively inlined by using the *recursive* option. If the *recursive* option is used on the top-level function, all function hierarchy in the design is removed.

The INLINE *off* option can optionally be applied to functions to prevent them being inlined. This option/em may be used to prevent Vivado HLS from automatically inlining a function.

The INLINE directive is a powerful way to substantially modify the structure of the code without actually performing any modifications to the source code and provides a very powerful method for architectural exploration.

Array Mapping

When there are many small arrays in the C Code, mapping them into a single larger array typically reduces the number of block RAM required.

Each array is mapped into a block RAM. The basic block RAM unit provide in an FPGA is 18K. If many small arrays do not use the full 18K, a better use of the block RAM resources is map many of the small arrays into a larger array. If a block RAM is larger than 18K, they are automatically mapped into multiple 18K units. In the synthesis report, review **Utilization Report ->** **Details ->** **Memory** for a complete understanding of the block RAMs in your design.

The ARRAY_MAP directive supports two ways of mapping small arrays into a larger one:

- **Horizontal mapping** - corresponds to creating a new array by concatenating the original arrays. Physically, this gets implemented as a single array with more elements.
- **Vertical mapping** - corresponds to creating a new array by concatenating the original words in the array. Physically, this gets implemented by a single array with a larger bit-width.

Horizontal Array Mapping

The following code example has two arrays that would result in two RAM components.

```
void func (...) {
    int8  array1[M];
    int12 array2[N];
    ...

    loop_1: for (i=0; i<M; i++) {
        array1[i] = ...;
        array2[i] = ...;
        ...
    }
    ...
}
```

Arrays *array1* and *array2* can be combined into a single array, specified as *array3* in the following example:

```
void func (...) {
    int8  array1[M];
    int12 array2[N];

    #pragma HLS ARRAY_MAP variable=array1 instance=array3 horizontal
    #pragma HLS ARRAY_MAP variable=array2 instance=array3 horizontal
    ...

    loop_1: for (i=0; i<M; i++) {
        array1[i] = ...;
        array2[i] = ...;
        ...
    }
    ...
}
```

In this example, the ARRAY_MAP directive transforms the arrays as shown in the following figure.

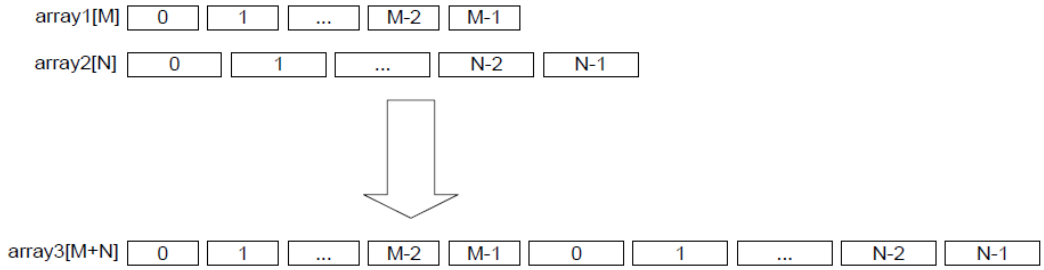


Figure 2.49: Horizontal Array Mapping

When using horizontal mapping, the smaller arrays are mapped into a larger array. The mapping starts at location 0 in the larger array and follows in the order the commands are specified. In the Vivado HLS GUI, this is based on the order the arrays are specified using the menu commands. In the Tcl environment, this is based on the order the commands are issued.

When you use the horizontal mapping shown in Figure 2.50, the implementation in the block RAM appears as shown in the following figure.

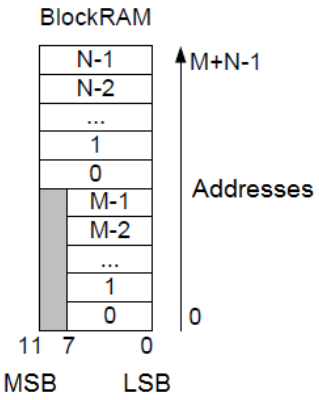


Figure 2.50: Memory for Horizontal Mapping

Vertical Array Mapping

In vertical mapping, arrays are concatenated by to produce an array with higher bit-widths.Vertical mapping is applied using the vertical option to the INLINE directive. The following figure shows how the same example as before transformed when vertical mapping mode is applied.

```
void func (...) {
    int8  array1[M];
    int12 array2[N];

    #pragma HLS ARRAY_MAP variable=array2 instance=array3 vertical
```

```
    #pragma HLS ARRAY_MAP variable=array1 instance=array3 vertical
    ...

    loop_1: for (i=0;i<M;i++) {
        array1[i] = ...;
        array2[i] = ...;
        ...
    }
    ...
}
```

The structure of the *array3* array, which is the result of vertical mapping *array1* and *array2* arrays is shown on the Figure 2.51.

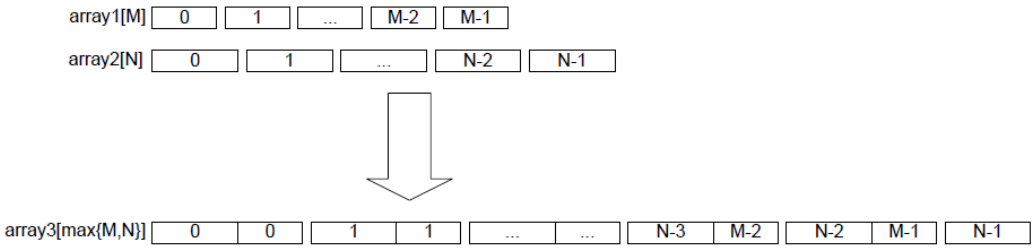


Figure 2.51: Vertical Array Mapping

In vertical mapping, the arrays are concatenated in the order specified by the command, with the first arrays starting at the LSB and the last array specified ending at the MSB. After vertical mapping the newly formed array, is implemented in a single block RAM component as shown in the following figure.

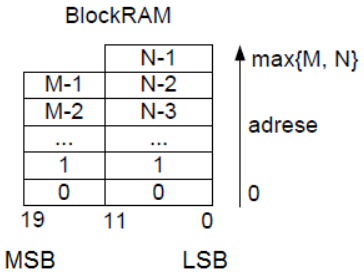


Figure 2.52: Memory for Vertical Mapping

Array Reshaping

The ARRAY_RESHAPE directive combines ARRAY_PARTITIONING with the vertical mode of ARRAY_MAP and is used to reduce the number of block RAM while still allowing the beneficial attributes of partitioning: parallel access to the data.

Given the following example code:

```
void func (...) {
    int array1[N];
    int array2[N];
    int array3[N];

    #pragma HLS ARRAY_RESHAPE variable=array1 block factor=2 dim=1
    #pragma HLS ARRAY_RESHAPE variable=array2 cycle factor=2 dim=1
    #pragma HLS ARRAY_RESHAPE variable=array3 complete dim=1
    ...
}
```

The ARRAY_RESHAPE directive transforms the arrays into the form shown in the following figure.

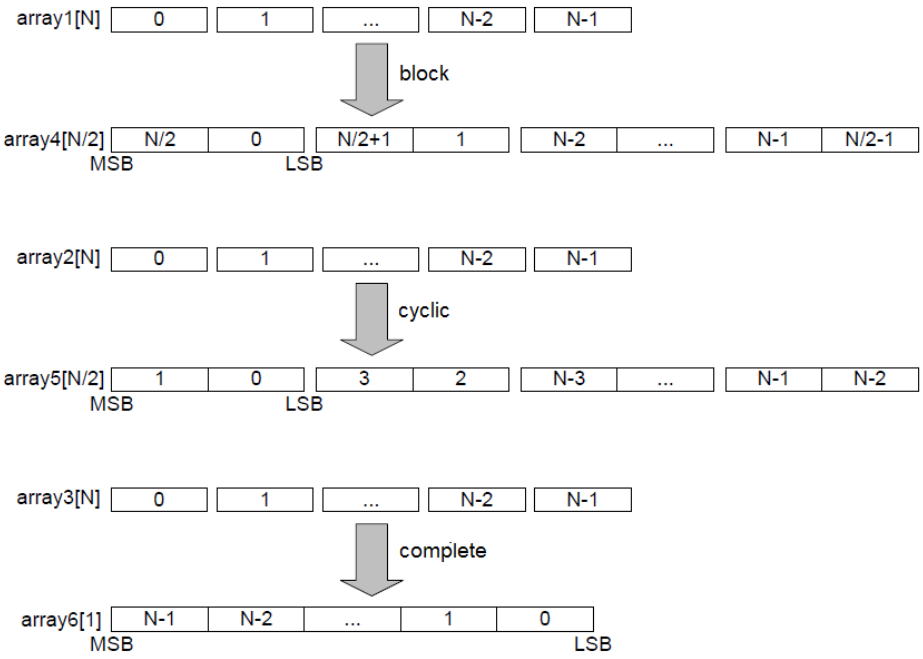


Figure 2.53: Array Reshaping

The ARRAY_RESHAPE directive allows more data to be accessed in a single clock cycle. In cases where more data can be accessed in a single clock cycle, Vivado HLS may automatically unroll any loops consuming this data, if doing so will improve the throughput. The loop can be fully or partially unrolled to create enough hardware to consume the additional data in a single clock cycle. This feature is controlled using the *config_unroll* command and the option *tripcount_threshold*. In the following example, any loops with a tripcount of less than 16 will be automatically unrolled if doing so improves the throughput.

```
config_unroll -tripcount_threshold 16
```

Resource Allocation

During synthesis Vivado HLS performs the following basic tasks:

- First, elaborates the C, C++ or SystemC source code into an internal database containing operators.

  The operators represent operations in the C code such as additions, multiplications, array reads, and writes.

- Then, maps the operators on to cores which implement the hardware operations.

  Cores are the specific hardware components used to create the design (such as adders, multipliers, pipelined multipliers, and block RAM).

Control is provided over each of these steps, allowing you to control the hardware implementation at a fine level of granularity.

Limiting the Number of Operators

Explicitly limiting the number of operators to reduce area may be required in some cases: the default operation of Vivado HLS is to first maximize performance. Limiting the number of operators in a design is a useful technique to reduce the area: it helps reduce area by forcing sharing of the operations.

The ALLOCATION directive allows you to limit how many operators, or cores or functions are used in a design. For example, if a design called foo has 317 multiplications but the FPGA only has 256 multiplier resources (DSP48s). The ALLOCATION directive shown below directs Vivado HLS to create a design with maximum of 256 multiplication (mul) operators:

```
int32 mac_unit (int16 d[317]) {
    static int32 mac;
    int i;
    #pragma HLS ALLOCATION instances=mul limit=256 operation
    for (i=0; i<300; i++) {
```

```
        #pragma HLS UNROLL
        mac += mac * d[i];
    }
    rerun mac;
}
```

You can use the *type* option to specify if the ALLOCATION directives limits operations, cores, or functions. The following table lists all the operations that can be controlled using the ALLOCATION directive.

Table 2.4: Vivado HLS Operators

Operator	Description
add	Integer Addition
ashr	Arithmetic Shift-Right
dadd	Double-precision floating point addition
dcmp	Double -precision floating point comparison
ddiv	Double -precision floating point division
dmul	Double -precision floating point multiplication
drecip	Double -precision floating point reciprocal
drem	Double -precision floating point remainder
drsqrt	Double -precision floating point reciprocal square root
dsub	Double -precision floating point subtraction
dsqrt	Double -precision floating point square root
fadd	Single-precision floating point addition
fcmp	Single-precision floating point comparison
fdiv	Single-precision floating point division
fmul	Single-precision floating point multiplication
frecip	Single-precision floating point reciprocal
frem	Single-precision floating point remainder
frsqrt	Single-precision floating point reciprocal square root
fsub	Single-precision floating point subtraction
fsqrt	Single-precision floating point square root
icmp	Integer Compare
lshr	Logical Shift-Right
mul	Multiplication
sdiv	Signed Divider
shl	Shift-Left
srem	Signed Remainder
sub	Subtraction
udiv	Unsigned Division
urem	Unsigned Remainder

Controlling the Hardware Cores

When synthesis is performed, Vivado HLS uses the timing constraints specified by the clock, the delays specified by the target device together with any directives specified by you, to determine which core is used to implement the operators. For example, to implement a multiplier operation Vivado HLS could use the combinational multiplier core or use a pipeline multiplier core.

The cores which are mapped to operators during synthesis can be limited in the same manner as the operators. Instead of limiting the total number of multiplication operations, you can choose to limit the number of combinational multiplier cores, forcing any remaining multiplications to be performed using pipelined multipliers (or vice versa). This is performed by specifying the ALLOCATION directive *type* option to be *core*.

The RESOURCE directive is used to explicitly specify which core to use for specific operations. In the following example, a 2-stage pipelined multiplier is specified to implement the multiplication for variable The following command informs Vivado HLS to use a 2-stage pipelined multiplier for variable c. It is left to Vivado HLS which core to use for variable d.

```
int func (int a, int b) {
    int c, d;

    #pragma HLS RESOURCE variable=c latency=2
    c = a*b;
    d = a*c;

    return d;
}
```

In the following example, the RESOURCE directives specify that the add operation for variable temp and is implemented

using the AddSub_DSP core. This ensures that the operation is implemented using a DSP48 primitive in the final design - by default, add operations are implemented using LUTs.

```
void apint_arith(int16 inA, int16 inB, int17 *out1) {
    int17 temp;
    #pragma HLS RESOURCE variable=temp core=AddSub_DSP
    temp = inB + inA;
     out1 = temp;
}
```

The following table lists the cores used to implement standard RTL logic operations (such as add, multiply, and compare).

Table 2.5: Functional Cores

Core	Description
AddSub	This core is used to implement both adders and subtractors.
AddSubnS	N-stage pipelined adder or subtractor. Vivado HLS determines how many pipeline stages are required.
AddSub_DSP	This core ensures that the add or sub operation is implemented using a DSP48 (Using the adder or subtractor inside the DSP48).
DivnS	N-stage pipelined divider.
DSP48	Multiplications with bit-widths that allow implementation in a single DSP48 macrocell. This can include pipelined multiplications and multiplications grouped with a pre-adder, post-adder, or both. This core can only be pipelined with a maximum latency of 4. Values above 4 saturate at 4.
Mul	Combinational multiplier with bit-widths that exceed the size of a standard DSP48 macrocell. ***Note***: Multipliers that can be implemented with a single DSP48 macrocell are mapped to the DSP48 core.
MulnS	N-stage pipelined multiplier with bit-widths that exceed the size of a standard DSP48 macrocell. ***Note***: Multipliers that can be implemented with a single DSP48 macrocell are mapped to the DSP48 core.
Mul_LUT	Multiplier implemented with LUTs.

The following table lists the cores used to implement storage elements, such as registers or memories.

Table 2.6: Storage Cores

Core	Description
FIFO	A FIFO. Vivado HLS determines whether to implement this in the RTL with a block RAM or as distributed RAM.
FIFO_ BRAM	A FIFO implemented with a block RAM.
FIFO_LUTRAM	A FIFO implemented as distributed RAM.
FIFO_SRL	A FIFO implemented as with an SRL.
RAM_1P	A single-port RAM. Vivado HLS determines whether to implement this in the RTL with a block RAM or as distributed RAM.
RAM_1P_BRAM	A single-port RAM implemented with a block RAM.
RAM_1P_LUTRAM	A single-port RAM implemented as distributed RAM.
RAM_2P	A dual-port RAM that allows read operations on one port and both read and write operations on the other port. Vivado HLS determines whether to implement this in the RTL with a block RAM or as distributed RAM.

RAM_2P_BRAM	A dual-port RAM implemented with a block RAM that allows read operations on one port and both read and write operations on the other port.
RAM_2P_LUTRAM	A dual-port RAM implemented as distributed RAM that allows read operations on one port and both read and write operations on the other port.
RAM_S2P_BRAM	A dual-port RAM implemented with a block RAM that allows read operations on one port and write operations on the other port.
RAM_S2P_LUTRAM	A dual-port RAM implemented as distributed RAM that allows read operations on one port and write operations on the other port.
RAM_T2P_BRAM	A true dual-port RAM with support for both read and write on both ports implemented with a block RAM.
ROM_1P	A single-port ROM. Vivado HLS determines whether to implement this in the RTL with a block RAM or with LUTs.
ROM_1P_BRAM	A single-port ROM. Vivado HLS determines whether to implement this in the RTL with a block RAM or with LUTs.
ROM_nP_BRAM	A multi-port ROM implemented with a block RAM. Vivado HLS automatically determines the number of ports.
ROM_1P_LUTRAM	A single-port ROM implemented with distributed RAM.
ROM_nP_LUTRAM	A multi-port ROM implemented with distributed RAM. Vivado HLS automatically determines the number of ports.
ROM_2P	A dual-port ROM. Vivado HLS determines whether to implement this in the RTL with a block RAM or as distributed ROM.
ROM_2P_BRAM	A dual-port ROM implemented with a block RAM.
ROM_2P_LUTRAM	A dual-port ROM implemented as distributed ROM.
XPM_MEMORY	Specifies the array is to be implemented with an UltraRAM. This core is only usable with devices supporting UltraRAM blocks

The RESOURCE directives uses the assigned variable as the target for the resource. If the assignment specifies multiple identical operators, the code must be modified to ensure there is a single variable for each operator to be controlled.

2.5 Verify the RTL Implementation

Post-synthesis verification is automated through the C/RTL co-simulation feature which reuses the pre-synthesis C test bench to perform verification on the output RTL.

C/RTL co-simulation uses the C test bench to automatically verify the RTL design. The verification process consists of three phases:

1. The C simulation is executed and the inputs to the top-level function, or the Device-Under-Test (DUT), are saved as “input vectors”.
2. The “input vectors” are used in an RTL simulation using the RTL created by Vivado HLS. The outputs from the RTL are save as “output vectors”.
3. The “output vectors” from the RTL simulation are applied to C test bench, after the function for synthesis, to verify the results are correct. The C test bench performs the verification of the results.

The following messages are output by Vivado HLS to show the progress of the verification.

C simulation:

```
[SIM-14] Instrumenting C test bench (wrapc)
[SIM-302] Generating test vectors(wrapc)
```

At this stage, since the C simulation was executed, any messages written by the C test bench will be output in console window or log file.

RTL simulation:

```
[SIM-333] Generating C post check test bench
[SIM-12] Generating RTL test bench
[SIM-323] Starting Verilog simulation (Issued when Verilog is the RTL verified)
[SIM-322] Starting VHDL simulation (Issued when VHDL is the RTL verified)
```

At this stage, any messages from the RTL simulation are output in console window or log file.

C test bench results checking:

```
[SIM-316] Starting C post checking
[SIM-1000] C/RTL co-simulation finished: PASS (If test bench returns a 0)
[SIM-4] C/RTL co-simulation finished: FAIL (If the test bench returns non-zero)
```

The following Figure 2.54 shows the RTL verification flow.

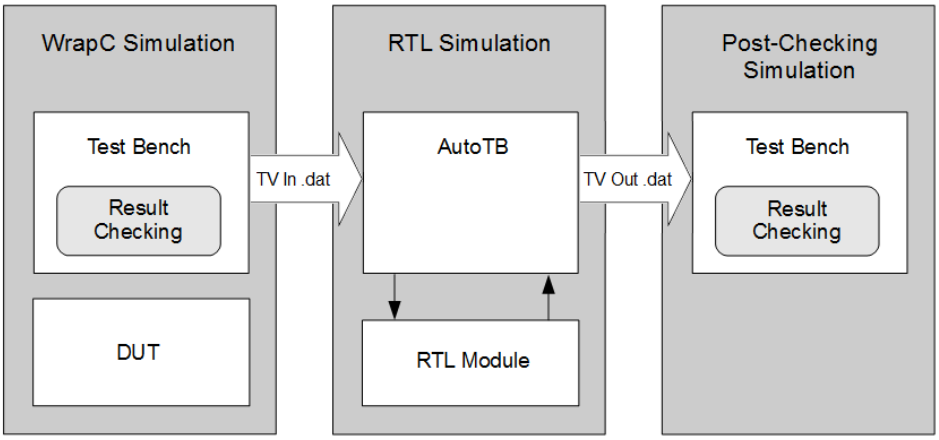


Figure 2.54: RTL Verification Flow

The following is required to use C/RTL co-simulation feature successfully:

- The test bench must be self-checking and return a value of 0 if the test passes or returns a non-zero value if the test fails.
- The correct interface synthesis options must be selected.
- Any 3rd-party simulators must be available in the search path.
- Any arrays or structs on the design interface cannot use the optimization directives or combinations of optimization directives.

To verify the RTL design produces the same results as the original C code, use a self-checking test bench to execute the verification. The following code example shows the important features of a self-checking test bench:

```
int main () {
    int ret=0;
    ...
    // Execute (DUT) Function
    ...

    // Write the output results to a file
    ...

    // Check the results
```

```
    ret = system("diff --brief -w output.dat output.golden.dat");

    if (ret != 0) {
        printf("Test failed !!!\n");
        ret=1;
    }
    else {
        printf("Test passed !\n");
    }
    ...
    return ret;
}
```

This self-checking test bench compares the results against known good results in the *output.golden.dat* file.

In the Vivado HLS design flow, the *return* value to function *main()* indicates the following:

- Zero: Results are correct.
- Non-zero value: Results are incorrect

  Note: The test bench can return any non-zero value. A complex test bench can return different values depending on the type of difference or failure. If the test bench returns a non-zero value after C simulation or C/RTL co-simulation, Vivado HLS reports an error and simulation fails.

  Constrain the return value to an 8-bit range for portability and safety, because the system environment interprets the return value of the *main()* function.

If the test bench does not check the results but returns zero, Vivado HLS indicates that the simulation test passed even though the results were not actually checked.

After ensuring that the preceding requirements are met, you can use C/RTL co-simulation to verify the RTL design using Verilog or VHDL. The default simulation language is Verilog, but you can also specify VHDL. While the default simulator is Vivado Simulator (XSim), you can use any of the following simulators to run C/RTL co-simulation:

- Vivado Simulator (XSim)
- ModelSim simulator
- VCS simulator (Linux only)
- NC-Sim simulator (Linux only)
- Riviera simulator (PC only)

2.5.1 Using C/RTL Co-Simulation

To perform C/RTL co-simulation from the GUI:

Step 1. In the main Vivado HLS toolbar menu, click the **C/RTL Cosimulation** button, see Figure 2.55. This option opens the simulation wizard window shown on the Figure 2.56.

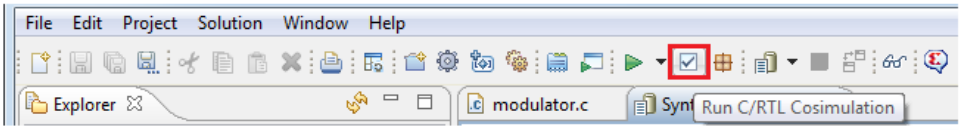


Figure 2.55: C/RTL CoSimulation toolbar button

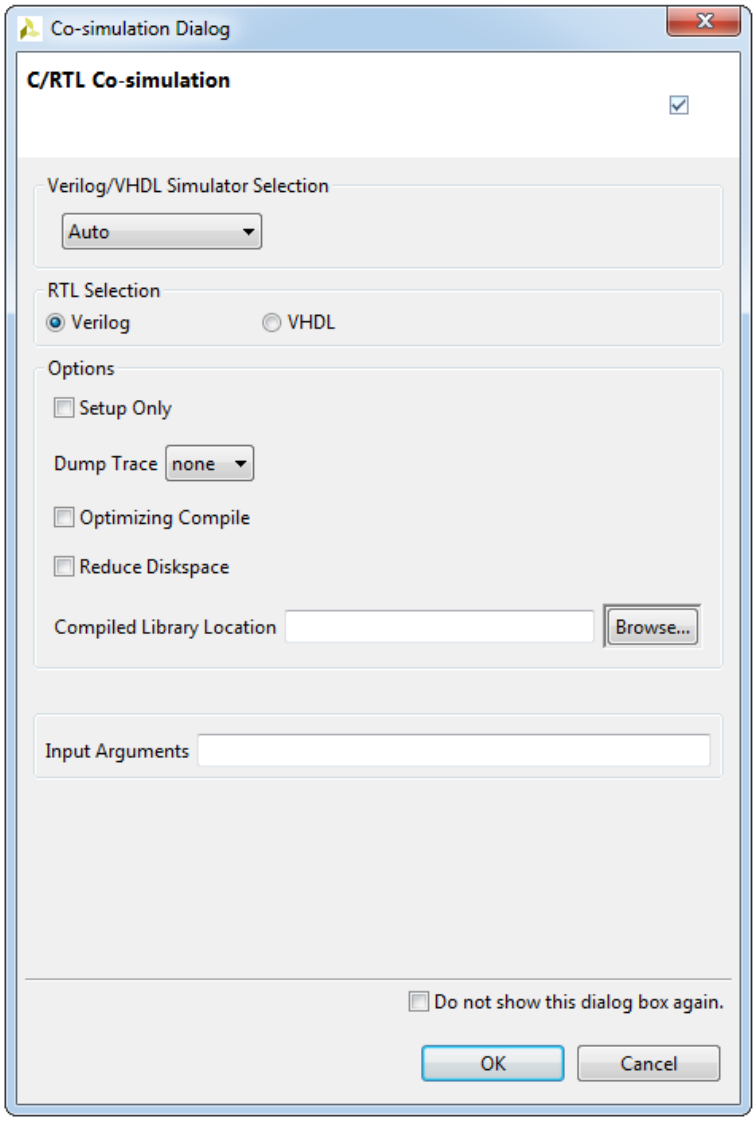


Figure 2.56: C/RTL Co-simulation dialog box

Step 2. In the **C/RTL Co-simulation** dialog box set the following parameters:

- choose **Vivado Simulator** in the **Verilog/VHDL Simulation Section** drop down list
- select **VHDL** in the **RTL Selection** section, and
- choose **all** in the **Dump Trace** drop down list, see Figure 2.57.

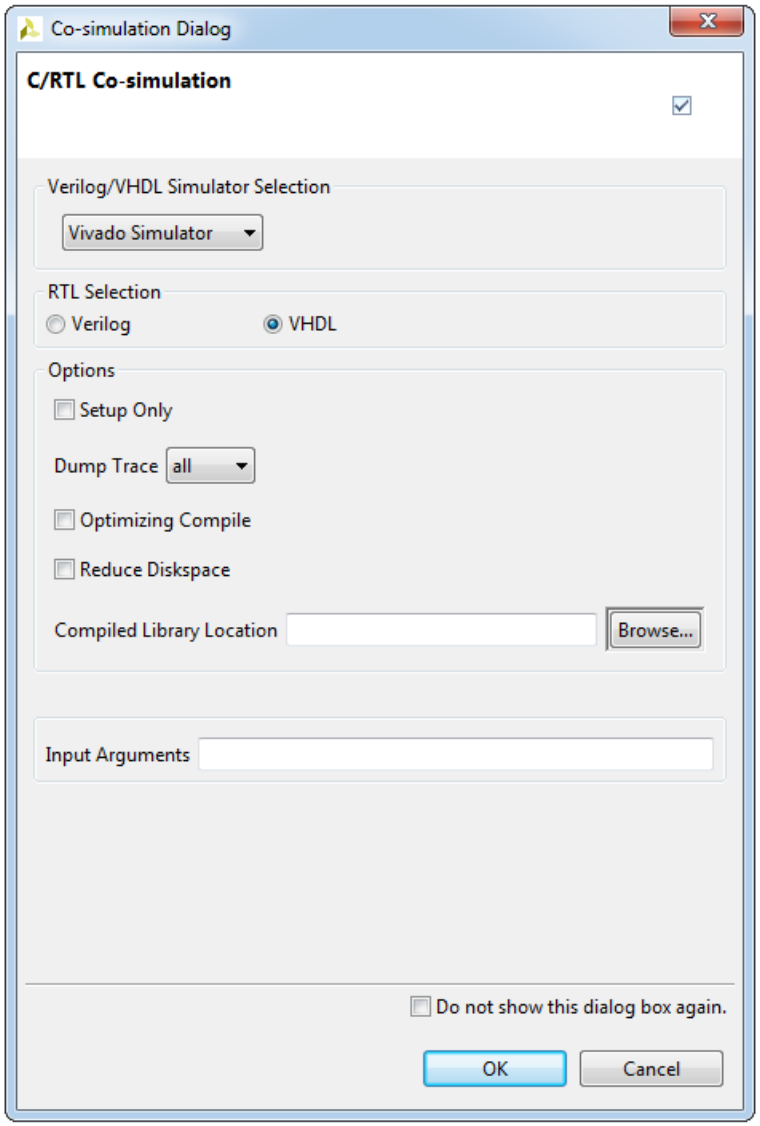


Figure 2.57: C/RTL Co-simulation dialog box with set parameters

Step 3. Leave all other parameters unchanged and click **OK**.

As can be seen from the previous figure, in the **C/RTL Co-simulation** dialog box there is an **Options** section where can be found the following options:

- **Setup Only** - This option creates all the files (wrappers, adapters, and scripts) required to run the simulation but does not execute the simulator. The simulation can be run in the command shell from within the appropriate RTL simulation folder *<solution_name>/sim/<RTL>*.
- **Dump Trace** - This option generates a trace file for every function, which is saved to the *<solution>/sim/<RTL>* folder. The drop-down menu allows you to select which signals are saved to the trace file. You can choose to trace all signals in the design, trace just the top-level ports, or trace no signals. For details on using the trace file, see the documentation for the selected RTL simulator.
- **Optimizing Compile** - This option ensures a high level of optimization is used to compile the C test bench. Using this option increases the compile time but the simulation executes faster.
- **Reduce Disk Space** - The flow shown on the Figure 2.46 in saves the results for all transactions before executing RTL simulation. In some cases, this can result in large data files. The reduce_diskspace option can be used to execute one transaction at a time and reduce the amount of disk space required for the file. If the function is executed N times in the C test bench, the reduce_diskspace option ensure N separate RTL simulations are performed. This causes the simulation to run slower.

- **Compiled Library Location** - This option specifies the location of the compiled library for a third-party RTL simulator.

  Note: If you are simulating with a third-party RTL simulator and the design uses IP, you must use an RTL simulation model for the IP before performing RTL simulation. To create or obtain the RTL simulation model, contact your IP provider.

- **Input Arguments** - This option allows the specification of any arguments required by the test bench.

Pressing the **OK** button in the **C/RTL Co-simulation** dialog box, the co-simulation process begins. Co-simulation flow can be traced within Vivado HLS Console window.

Vivado HLS executes the RTL simulation in the project sub-directory: *<SOLUTION>/sim/<RTL>*, where

- SOLUTION is the name of the solution.

- RTL is the RTL type chosen for simulation.

Any files written by the C test bench during co-simulation and any trace files generated by the simulator are written to this directory.

2.5.2 Analyzing RTL Simulations

Optionally, you can review the waveform from C/RTL cosimulation using the **Open Wave Viewer...** toolbar button, see Figure 2.58.

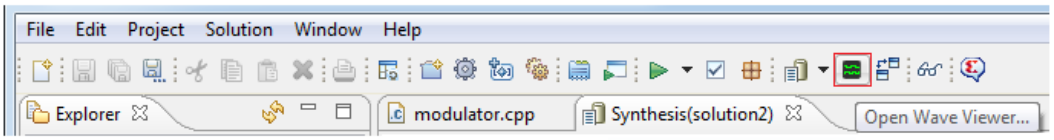


Figure 2.58: Open Wave Viewer toolbar button

To view RTL waveforms, you must select the following options before executing C/RTL cosimulation:

- **Verilog/VHDL Simulator Selection** - Select **Vivado Simulator**.

  For Xilinx 7 series and later devices, you can alternatively select **Auto**.

- **Dump Trace** - Select all or port.

When C/RTL cosimulation completes, the **Open Wave Viewer** toolbar button opens the RTL waveforms in the Vivado IDE, see Figure 2.59.

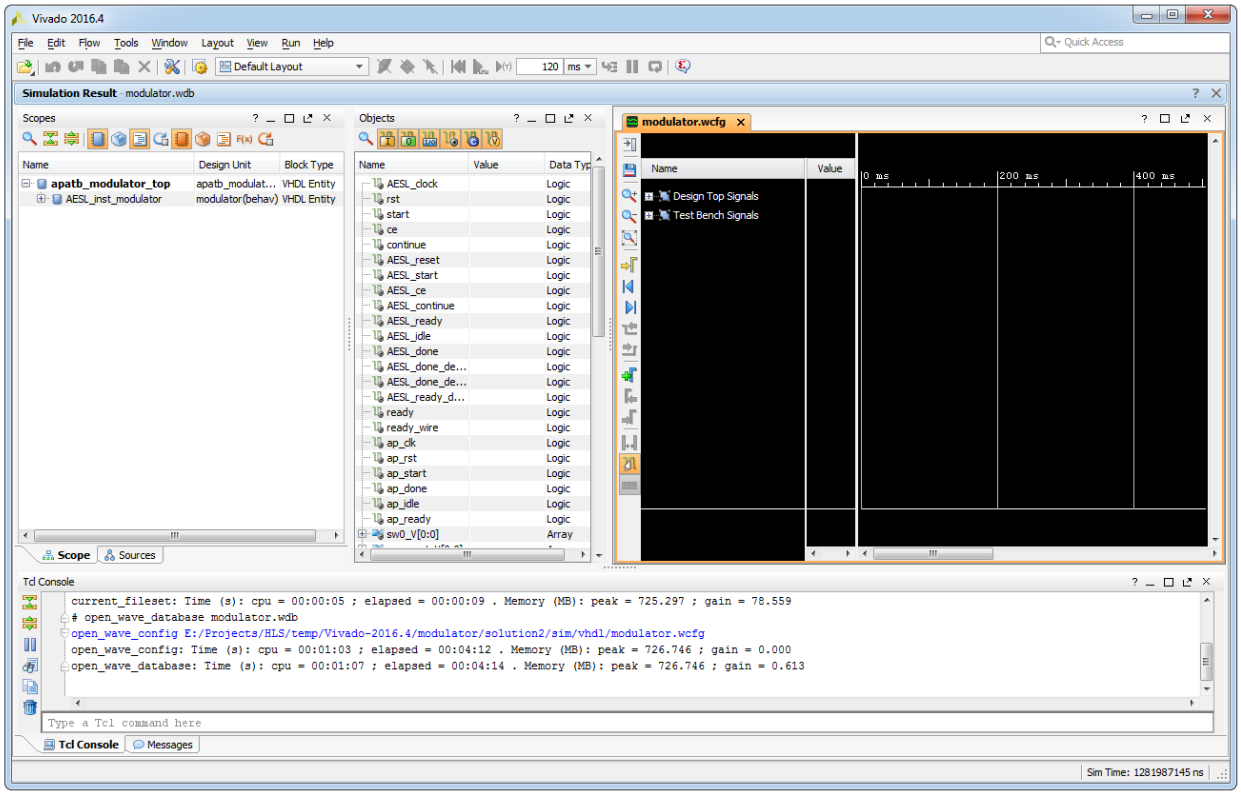


Figure 2.59: Waveform Viewer window opened in Vivado IDE

Note: When you open the Vivado IDE using this method, you can only use the waveform analysis features, such as zoom, pan, and waveform radix.

In the **Waveform Viewer** window expand **Design Top Signals** folder and then find **sw0_V[0:0]** port (in the **C Inputs ->** **sw0(wire)** folder) and **pwm_out_V[0:0]** port (in the **C Outputs ->** **pwm_out(wire)** folder) and expand them also, see Figure 2.60. Zoom in few times around spot where **sw0_V[0:0]** port changes its value from 0 to 1 and you will see the PWM signal period change. You can also notice the change of the duty cycle of the PWM signal, as it is being modulated by the sine wave. When sw0_V[0:0]=0 the period of the PWM signal is 3.5 times longer then in case when sw0_V[0:0]=1, as it was expected.

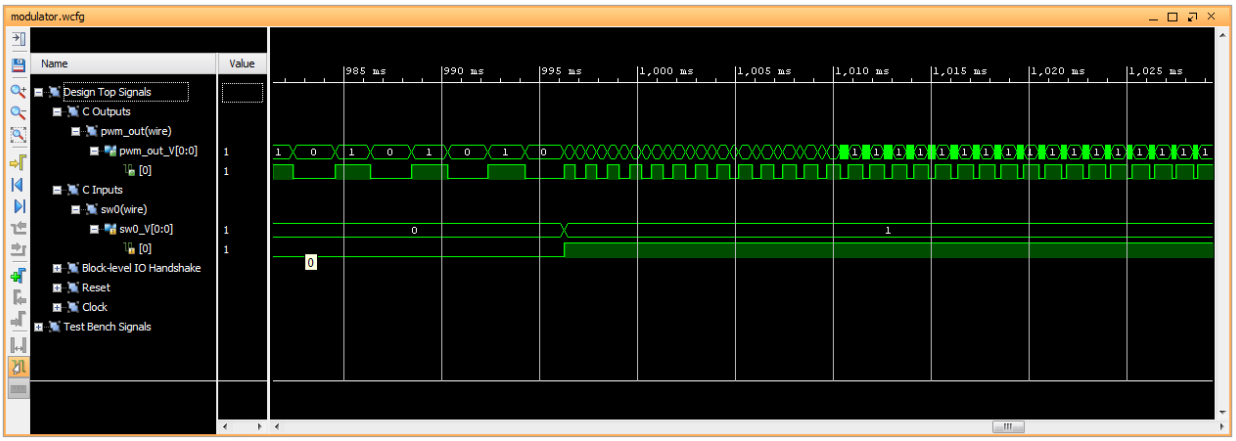


Figure 2.60: Waveform Viewer window with cosimulation results

2.6 Package the RTL Implementation

The final step in the Vivado HLS flow is to export the RTL design as a block of Intellectual Property (IP) which can be used by other tools in the Xilinx design flow. The RTL design can be packaged into the following output formats:

- IP Catalog formatted IP for use with the Vivado Design Suite
- System Generator for DSP IP for use with Vivado System Generator for DSP
- Synthesized Checkpoint (.dcp)

You can only export designs targeted to 7 series devices, Zynq-7000 AP SoC, and UltraScale devices to the Vivado Design Suite design flows.

In addition to the packaged output formats, the RTL files are available as standalone files (not part of a packaged format) in the *verilog* and *vhdl* directories located within the implementation directory *<project_name>/<solution_name>/impl*.

When Vivado HLS reports on the results of synthesis, it provides an estimation of the results expected after RTL synthesis: the expected clock frequency, the expected number of registers, LUTs and block RAMs. These results are estimations because Vivado HLS cannot know what exact optimizations RTL synthesis performs or what the actual routing delays will be, and hence cannot know the final area and timing values.

Before exporting a design, you have the opportunity to execute logic synthesis and confirm the accuracy of the estimates. The evaluate option invokes RTL synthesis during the export process and synthesizes the RTL design to gates.

Note: The RTL synthesis option is provided to confirm the reported estimates. In most cases, these RTL results are not included in the packaged IP.

For most export formats, the RTL synthesis is executed in the *verilog* or *vhdl* directories, but the results of RTL synthesis are not included in the packaged IP.

2.6.1 Packaging IP using IP Catalog Format

Upon completion of synthesis and RTL verification:

Step 1. Open the **Export RTL** dialog box by clicking the **Export RTL** toolbar button or choosing the **Solution ->** **Export RTL** option from the main Vivado HLS menu, see Figure 2.61.

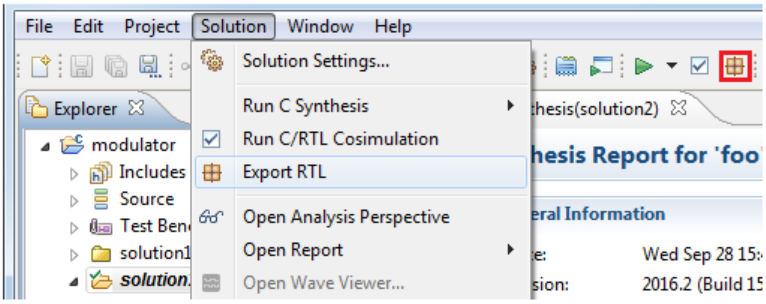


Figure 2.61: Export RTL option

Step 2. In the **Export RTL** dialog box choose **IP Catalog** option from the **Format Selection** drop down list, see Figure 2.62.

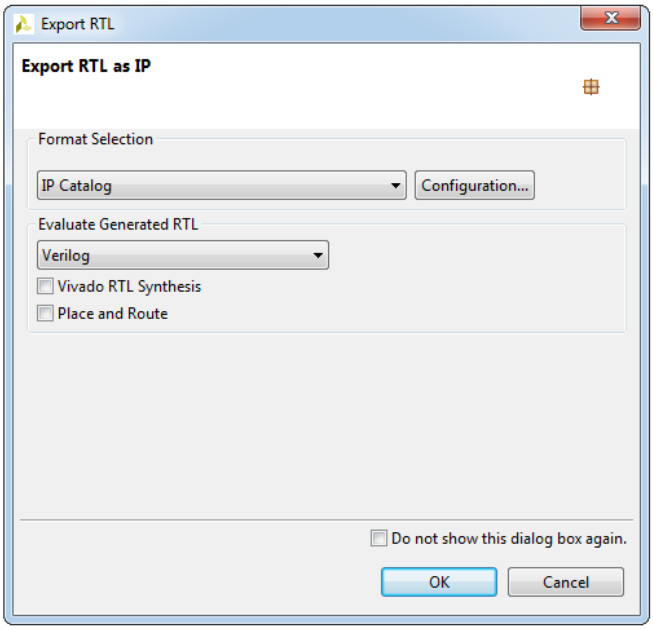


Figure 2.62: Export RTL dialog box

In the **Format Selection** drop down list you can choose between **IP Catalog**, **System Generator for DSP** or **Synthesized Checkpoint (.dcp)** format options in which RTL model will be exported. Depending of the chosen format, by clicking the **Configuration...** button, it is possible to set the additional parameters, see Illustration 2.63.

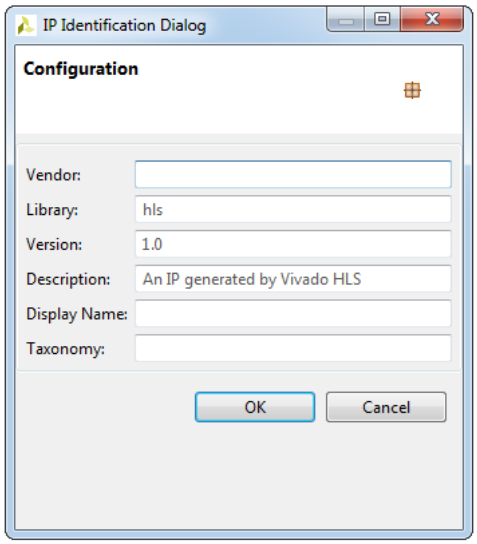


Figure 2.63: Configuration dialog box

The **Configuration** options allow the following identification tags to be embedded in the exported package. These fields can be used to help identify the packaged RTL inside the Vivado IP Catalog.

The configuration information is used to differentiate between multiple instances of the same design when the design is loaded into the IP Catalog. For example, if an implementation is packaged for the IP Catalog and then a new solution is created and packaged as IP, the new solution by default has the same name and configuration information. If the new solution is also added to the IP Catalog, the IP Catalog will identify it as an updated version of the same IP and the last version added to the IP Catalog will be used.

An alternative method is to use the *prefix* option in the *config_rtl* configuration to rename the output design and files with a unique prefix.

Step 3. In the **Configuration** dialog box provide the following configuration setting:

- **Vendor**: so-logic
- **Library**: hls
- **Version**: 1.0
- **Description**: An IP generated by Vivado HLS
- **Display Name**: hls_modulator_v1.0

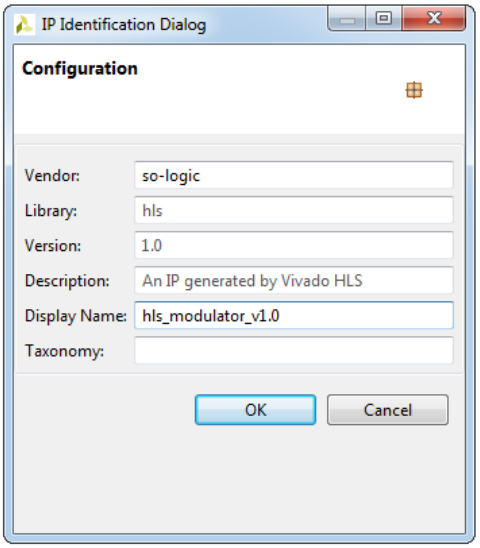


Figure 2.64: Configuration dialog box in case IP Catalog format

Step 4. In the **Configuration** dialog box click **OK**.

Step 5. In the **Export RTL** dialog box also click **OK**.

When you press OK button in the **Export RTL** dialog box, Vivado HLS will start exporting RTL model into chosen format.

After the packaging process is complete, the.zip file archive in directory *<project_name>/<solution_name>/impl/ip* can be imported into the Vivado IP Catalog and used in any Vivado design (RTL or IP Integrator).

Important: In this tutorial we will use only exporting IP to **IP Catalog**!

If you choose **System Generator for DSP** format option, this package will be written to the <project_name>/<solution_name>/impl/sysgen directory and will contain everything necessary to import the design to System Generator.

A Vivado HLS generated System Generator package may be imported into System Generator using the following steps:

1. Inside the System Generator design, right-click and use option XilinxBlockAdd to instantiate new block.
2. Scroll down the list in dialog box and select Vivado HLS.
3. Double-click on the newly instantiated Vivado HLS block to open the Block Parameters dialog box.
4. Browse to the solution directory where the Vivado HLS block was exported. Using the example, *<project_name>/<solution_name>/impl/sysgen*, browse to the *<project_name>/<solution_name>* directory and select apply.

Chapter 3

USING DEVELOPED IP CORE IN VIVADO DESIGN SUITE

After the Modulator design is packaged into IP Catalog format, the next step will be to download the packaged design into the target FPGA device. In our case it will be ZedBoard evaluation board.

Step 1. Close the **Vivado HLS** tool and open **Vivado IDE** tool.

Step 2. In the Vivado IDE tool create a new project, ***modulator_hls***, targeting the ZedBoard evaluation board and save it in the same directory where the Vivado HLS ***modulator*** project is saved. For details how to create Vivado project, please look at the *Chapter 2.2 Creating a New Project* in the *Basic FPGA Tutorial*.

Step 3. In the Vivado IDE click **Project Settings** command from the **Project Manager** section to open the **Project Settings** dialog box.

Step 4. In the **Project Settings** dialog box open **General** section and change **Target language** to be **VHDL** instead of **Verilog**.

Step 5. In the **Project Settings** dialog box open **IP** section and select **Repository Manager** tab.

Step 6. In the **Repository Manager** tab click "+" button to add the desired IP core into the project.

Step 7. In the **IP Repositories** dialog box find the HLS *modulator/solution2/impl/ip* folder, where is the required ***so-logic-_hls_modulator_1_0*** IP core stored, select it and click **Select**.

Step 8. In the **Add Repository** dialog box click **OK** to add the selected IP core to the **Repository Manager**.

Step 9. In the **Project Settings** dialog box, just click **OK** and the required ***so-logic_hls_modulator_1_0*** IP core should appear in the IP Catalog of your project.

Note: For more details how to include packaged IP core to the IP Catalog of your Vivado project, please look at the *Chapter 13.1 IP Packager*, steps 33 - 39, in the *Basic FPGA Tutorial*.

Step 10. In the **Flow Navigator**, under the **Project Manager**, click **IP Catalog** command to verify the presence of the previously created IP in the IP Catalog. In the **Search** field type the name of the IP core (in our case ***hls_modulator_v1.-0***).

Step 11. Double-click on the ***hls_modulator_v1.0*** IP core and Vivado IDE will automatically create a new skeleton source for your IP.

The window that will be opened is used to set up the general ***hls_modulator_v1.0*** IP core parameters, see Figure 3.1.

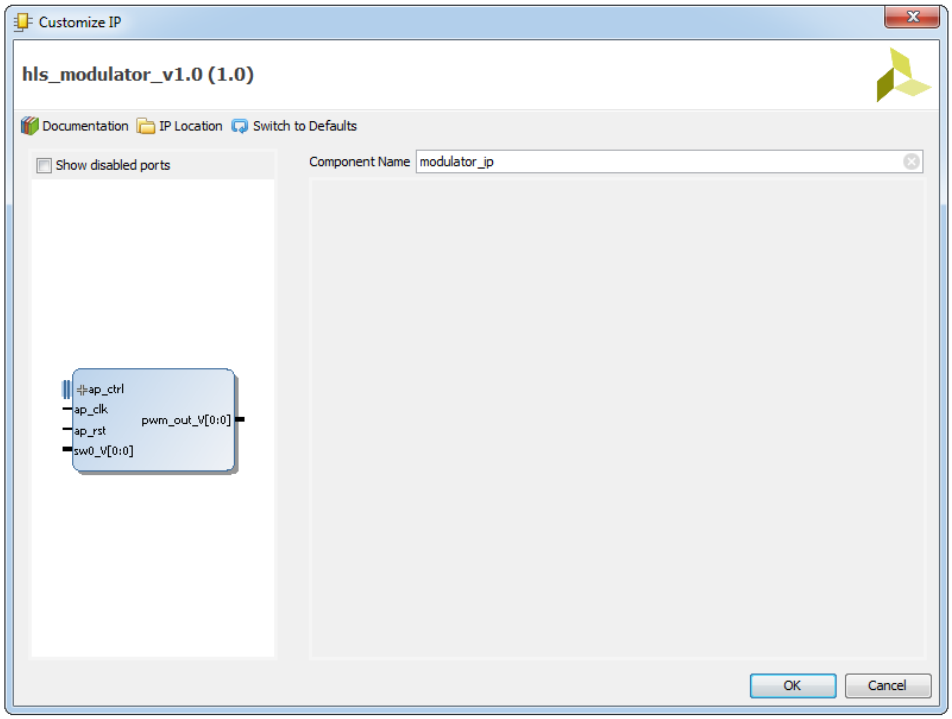


Figure 3.1: Customize IP dialog box for hls_modulator_v1.0(1.0) IP core

Step 12. In the ***hls_modulator_v1.0 (1.0)*** customization window, change the **Component Name** to be ***modulator_ip***, as it is shown on the Figure 3.1 and click **OK**.

Step 13. In the **Generate Output Products** dialog box, click **Generate**.

Step 14. In the second **Generate Output Products** dialog box, click **OK**.

Note: After ***modulator_ip*** IP core generation, your ***modulator_ip*** IP core should appear in the **Sources** window, see Figure 3.2.

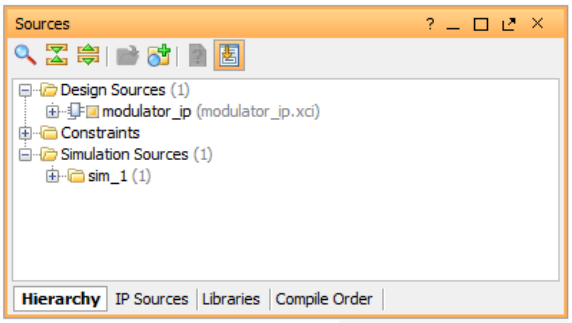


Figure 3.2: Sources window with generated hls_modulator_v1.0(1.0) IP core

After configuring and generating ***modulator_ip*** IP core, we will create a top-level VHDL module, **modulator_hls_wrapper.vhd**, where we will connect the ***modulator_ip*** IP core with the rest of the Modulator design and **modulator_pkg.vhd** VHDL package file holding necessary type definitions used in top-level VHDL module.

To create these modules, use steps for creating modules, explained in the *Chapter 2.4.1 Creating a Module Using Vivado Text Editor*, in the *Basic FPGA Tutorial*.

The content of the **modulator_hls_wrapper.vhd** and **modulator_pkg.vhd** VHDL files is presented in the text below.

modulator_hls_wrapper.vhd:

```
-- Make reference to libraries that are necessary for this file:
```

```
-- the first part is a symbolic name, the path is defined depending of the tools
-- the second part is a package name
-- the third part includes all functions from that package
-- Better for documentation would be to include only the functions that are necessary

library ieee;
    use ieee.std_logic_1164.all;
    use ieee.std_logic_arith.all;
    use ieee.std_logic_unsigned.all;

library unisim;
    use unisim.vcomponents.all;

    use work.modulator_pkg.all;

-- Entity defines the interface of a module
-- Generics are static, they are used at compile time
-- Ports are updated during operation and behave like signals on a schematic or
-- traces on a PCB
-- Entity is a primary design unit

entity modulator_hls_wrapper is
    generic(
        -- If some module is top, it needs to implement the differential clk buffer,
        -- otherwise this variable will be overwritten by a upper hierarchy layer
        this_module_is_top_g : module_is_top_t := yes;

        -- Parameter that specifies major characteristics of the board that will be used
        -- to implement the modulator design
        -- Possible choices: """lx9""", """zedboard""", """ml605""", """kc705""", """microzed""", ""socius"
      ""
        -- Adjust the modulator_pkg.vhd file to add more
        board_name_g : string := """zedboard""";

        -- User defined settings for the pwm design
        design_setting_g : design_setting_t_rec := design_setting_c

        );

    port(
        clk_p   : in std_logic;    -- differential input clock signal
        clk_n   : in std_logic;    -- differential input clock signal
        sw0     : in std_logic;    -- signal made for selecting frequency
        pwm_out : out std_logic    -- pulse width modulated signal
--        clk_en  : out std_logic    -- clock enable port used only for MicroZed board
        );

end entity;

-- Architecture is a secondary design unit and describes the functionality of the module
-- One entity can have multiple architectures for different families,
-- technologies or different levels of description
-- The name should represent the level of description like
-- structural, rtl, tb and maybe for which technology

architecture rtl of modulator_hls_wrapper is

-- Between architecture and begin is declaration area for types, signals and constants
-- Everything declared here will be visible in the whole architecture

    -- input clock signal
    signal clk_in_s : std_logic;

    signal sw_s : std_logic_vector(0 downto 0);
    signal pwm_s: std_logic_vector(0 downto 0);
begin

    -- in case of MicroZed board we must enable on-board clock generator
--    clk_en <= '1';

    -- if module is top, it has to generate the differential clock buffer in case
    -- of a differential clock, otherwise it will get a single ended clock signal
    -- from the higher hierarchy

    clk_buf_if_top : if (this_module_is_top_g = yes) generate

        clk_buf : if (get_board_info_f(board_name_g).has_diff_clk = yes) generate

            ibufgds_inst : ibufgds
                generic map(
                    ibuf_low_pwr => true,
                    -- low power (true) vs. performance (false) setting for referenced I/O standards
                    iostandard => "default"
                )

                port map (
                    o => clk_in_s, -- clock buffer output
                    i => clk_p,    -- diff_p clock buffer input
                    ib => clk_n    -- diff_n clock buffer input
                );
        end generate clk_buf;
```

```
        no_clk_buf : if (get_board_info_f(board_name_g).has_diff_clk = no) generate
            clk_in_s <= clk_p;
        end generate no_clk_buf;

    end generate clk_buf_if_top;

    not_top : if (this_module_is_top_g = no) generate
        clk_in_s <= clk_p;
    end generate not_top;

    sw_s(0) <= sw0;
    pwm_out <= pwm_s(0);
    pwmmodulator : entity work.modulator_ip   -- HLS generated PWM modulator module instance
        port map(
            ap_clk    => clk_in_s,
            ap_rst    => '0',
            ap_start  => '1',
            ap_done   => open,
            ap_idle   => open,
            ap_ready  => open,
            sw0_V     => sw_s,
            pwm_out_V => pwm_s
            );

end;
```

modulator_pkg.vhd:

```
-- Make reference to libraries that are necessary for this file:
-- the first part is a symbolic name, the path is defined depending of the tools
-- the second part is a package name
-- the third part includes all functions from that package
-- Better for documentation would be to include only the functions that are necessary

library ieee;
    use ieee.math_real.all;
    use ieee.std_logic_1164.all;
    use ieee.std_logic_arith.all;
    use ieee.std_logic_unsigned.all;

-- VHDL package is a way of grouping related declarations that serve a common purpose
-- Each VHDL package contains package declaration and package body
-- Package declaration:

package modulator_pkg is

    type module_is_top_t is (yes, no); -- only the top module can instantiate a diff clk buffer
    type board_type_t    is (lx9, zedboard, ml605, kc705, microzed, socius);
    type has_diff_clk_t  is (yes, no);

    type board_setting_t_rec is record
        board_name   : board_type_t;  -- specifies the name of the board that we are using
        fclk         : real;          -- specifies the reference clock frequency that is presented on the
      board (in Hz)
        has_diff_clk : has_diff_clk_t;-- specifies if board has differential clock or not
    end record board_setting_t_rec;

    -- place the information about the new boards here:
    constant lx9_c      : board_setting_t_rec := (lx9, 100000000.0, no);      -- Spartan-6
    constant zedboard_c : board_setting_t_rec := (zedboard, 100000000.0, no); -- Zynq-7000
    constant ml605_c    : board_setting_t_rec := (ml605, 200000000.0, yes);   -- Virtex-6
    constant kc705_c    : board_setting_t_rec := (kc705, 200000000.0, yes);   -- Kintex-7
    constant microzed_c : board_setting_t_rec := (microzed, 33333333.3, no);  -- MicroZed
    constant socius_c   : board_setting_t_rec := (socius, 50000000.0, no);    -- Socius

    -- array holding information about supported boards
    type board_info_t_arr is array (1 to 6) of board_setting_t_rec;
    constant board_info_c : board_info_t_arr := (lx9_c, zedboard_c, ml605_c, kc705_c, microzed_c, socius_c)
      ;

    type vector_t_arr is array (natural range <>) of integer;

    constant per_c : time := 20 ns; -- clock period (T=1/50 MHz), that is used in almost all test benches


    type design_setting_t_rec is record
        cntampl_value : integer;       -- counter amplitude border,
                                       -- it's value should be equal to (2^depth)-1
        f_low : real;                  -- first frequency for the PWM signal, specified in Hz
        f_high: real;                  -- second frequency for the PWM signal, specified in Hz
        depth : integer range 0 to 99; -- the number of samples in one period of the signal
        width : integer range 0 to 99; -- the number of bits used to represent amplitude value
    end record design_setting_t_rec;

    constant design_setting_c : design_setting_t_rec := (255, 1.0, 3.5, 8, 12);

    -- init_sin_f function declaration
    function init_sin_f
```

```
        (
        constant depth_c : in integer; -- is the number of samples in one period of the signal (2^8=256)
        constant width_c : in integer  -- is the number of bits used to represent amplitude value (2^12=409
      6)
        )
    return vector_t_arr;

    -- function that returns the information about the selected development board
    function get_board_info_f
        (
        constant board_name_c : in string
        )
    return board_setting_t_rec;
end;

-- In the package body will be calculated sine signal
-- Package body:
package body modulator_pkg is

    -- init_sin_f function definition
    function init_sin_f
        (
        constant depth_c : in integer;
        constant width_c : in integer
        )
    return vector_t_arr is

        variable init_arr_v : vector_t_arr(0 to (2 ** depth_c - 1));

    begin

        for i in 0 to ((2 ** depth_c)- 1) loop   -- calculate amplitude values
            init_arr_v(i) := integer(round(sin((math_2_pi / real(2 ** depth_c))*real(i)) * (real(2 ** (
      width_c - 1)) - 1.0)))
                                              + integer(2 ** (width_c - 1) - 1);
                          -- sin (2*pi*i / N) * (2width_c-1 - 1) + 2width_c-1 - 1, N = 2depth_c
        end loop;

        return init_arr_v;

    end;

    -- function that returns the information about the selected development board
    function get_board_info_f
        (
        constant board_name_c : in string
        )
    return board_setting_t_rec is

    begin
        for i in 1 to board_info_c'length loop
            -- if supplied board name equals some of supported boards, return board information for that
      board
            if (board_type_t'image(board_info_c(i).board_name) = board_name_c(2 to board_name_c'length-1))
      then
                return board_info_c(i);
            end if;
        end loop;
    end;
end;
```

Step 15. Now is the time to create constraints file for the ZedBoard evaluation platform, **modulator_zedboard.xdc**. Open Vivado text editor, copy your constraints code in it or write directly in it and save the constraints file in your working directory. The complete **modulator_zedboard.xdc** source file you can find in the text below.

modulator_zedboard.xdc:

```
set_property LOC Y9  [get_ports clk_p];
set_property LOC F22 [get_ports sw0];
set_property LOC T22 [get_ports pwm_out];

set_property IOSTANDARD LVCMOS33 [get_ports clk_p];
set_property IOSTANDARD LVCMOS25 [get_ports sw0];
set_property IOSTANDARD LVCMOS33 [get_ports pwm_out];

create_clock -period 10.000 -name clk_p -waveform {0.000 5.000} [get_ports clk_p]
```

Note: If you want to read more about XDC constraints, please look at the *Sub-chapter 10.1 Creating XDC File*, in the *Basic FPGA Tutorial*.

After we finished with the ***modulator_hls_wrapper.vhd***, ***modulator_pkg.vhd*** and ***modulator_zedboard.xdc*** files creation, we should return to the Vivado IDE and do the following:

Step 16. Add ***modulator_hls_wrapper.vhd***, ***modulator_pkg.vhd*** and ***modulator_zedboard.xdc*** files in the "modulator_hls" project with Flow Navigator **Add Sources** option:

- ***modulator_hls_wrapper.vhd*** and ***modulator_pkg.vhd*** as Design Source files, and
- ***modulator_zedboard.xdc*** as Constraints file.

Step 17. Make sure that ***modulator_hls_wrapper.vhd*** is a top-level file. If it is not, select it, right-click on it and choose **Set as Top** option.

Note: If you do not know how to add source and constraints files to the project, please see *Chapter 2.4.1 Creating a Module Using Vivado Text Editor* for VHDL files and *Sub-chapter 10.1 Creating XDC File* for XDC constraints file, in the *Basic FPGA Tutorial*.

Step 18. Synthesize your design with **Run Synthesis** option from the **Flow Navigator** / **Synthesis** (see *Sub-chapter 6.5.2 Run Synthesis*, in the *Basic FPGA Tutorial*).

Step 19. Implement your design with **Run Implementation** option from the **Flow Navigator** / **Implementation** (see *Sub-chapter 10.2.2 Run Implementation*, in the *Basic FPGA Tutorial*).

Step 20. Generate bitstream file with **Generate Bitstream** option from the **Flow Navigator** / **Program and Debug** (see *Sub-Chapter 10.3 Generate Bitstream File*, in the *Basic FPGA Tutorial*).

Step 21. Program your ZedBoard device (see *Sub-Chapter 10.4 Program Device*, in the *Basic FPGA Tutorial*).