Basic FPGA Tutorial

using VHDL and VIVADO to design two frequencies PWM modulator system

January 4, 2017

Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

1.1 Motivation

Basic FPGA Tutorial is a document made for beginners who are entering the FPGA world. This tutorial explains, step by step, the procedure of designing a simple digital system using VHDL language 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 Integrated Design Environment (IDE)
- Learn the various types of projects that can be created with the New Project Creation Wizard
- Create and add design source files with the Vivado IDE
- Synthesize and implement the design in the Vivado IDE
- Simulate a design using integrated Vivado Simulator
- Run your design on the target development board
- Debug a design in hardware using Vivado Logic Analyzer and Oscilloscope
- Designing with IPs

The following project is designed for:

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

1.3 Structure of this Tutorial

This tutorial is composed of eight chapters. The content of each chapter is explained in the text below:

- **Chapter 1: "Introduction"** - In this chapter you will find what is the purpose of this tutorial, explanation what is the PWM signal, frequency calculations, block diagram of one possible solution for the modulator design and a lot of basic information about the Vivado Design Suite.

- **Chapter 2: "Frequency Trigger"** - In this chapter you will find all the necessary information about how to create a new project in the Vivado IDE, how to create Frequency Trigger module as constituent part of the Modulator design, how to generate its test bench file and how to simulate it with the integrated Vivado simulator.
- **Chapter 3: "Counter"** - This chapter explains how to create Counter module, how to create its test bench file and how to simulate it with Vivado simulator.
- **Chapter 4: "Sine Package"** - This chapter holds the information how to create Sine package as one universal package that will be used in almost all modules of the Modulator design.
- **Chapter 5: "Digital Sine"** - This chapter explains how to create Digital Sine module, how to create its test bench file and how to simulate it with Vivado simulator.
- **Chapter 6: "Digital Sine Top"** - In this chapter you will find all the necessary information about how to create Digital Sine Top module which combines Frequency Trigger, Counter, Sine package and Digital Sine modules into one larger module. You will also find information about how to create its test bench file and how to simulate it with Vivado simulator. Additionally, this chapter holds information about the Vivado synthesis process.
- **Chapter 7: "PWM"** - This chapter explains how to create PWM module. This module will generate an PWM signal modulated using the digital sine wave from the Digital Sine module. In this chapter you will find how to create its FSM state diagram, its test bench file and how to simulate it with Vivado simulator.
- **Chapter 8: "Modulator"** - This chapter includes all the necessary information about the Modulator module, as the top module of our design. In this chapter you will find information how to create Modulator module and its test bench file and how to simulate it with Vivado simulator.
- **Chapter 9: "Modulator Wrapper"** - This chapter includes all the necessary information about the Modulator Wrapper module. In this chapter you will find information how to create a wrapper for the Modulator module that enables easy portability of the Modulator design between different development boards with different types of reference clock sources.
- **Chapter 10: "Design Implementation"** - This is a large chapter and includes all the information about the design implementation process steps. In this chapter you will learn how to create XDC file, how to implement your design, how to generate bitstream file and how to program your device. Here you will also find information about the necessary modifications in case of using different development boards.
- **Chapter 11: "Debugging Design"** - This chapter explains how you can debug your design first using internal Vivado Logic Analyzer and then using Oscilloscope as external analyzer. In this chapter you will also find what are the differences between "HDL Instantiation Debug Probing Flow" and "Netlist Insertion Debug Probing Flow".
- **Chapter 12: "Modulator Design Targeting Socius Development Board"** - This chapter will show you how to define the structure of the ARM-based processor system for socius development board, that will be used as a part of the solution for PWM signal generation.
- **Chapter 13: "Debugging with IPs"** - This chapter explains how you can create Modulator design using your own IPs, with the help of the Vivado IP Packager and IP Integrator tools, how you can debug IP integrated designs and how you can create new Modulator IP core with AXI4 interface in it.
- **Chapter 14: "Appendix"** - This chapter contains explanations about various features of the Xilinx Vivado tool that are not covered in any of the chapters.

This tutorial is accompanied by the .odp labs presentations. In total there are 19 labs. Correlation between labs and this tutorial document is the following:

- **Lab 1: "Introduction"** - covers the information presented in the **Chapter 1: "Introduction"** of this tutorial.
- **Lab 2: "Using the Vivado Tool"** - presents the overview of design development using Xilinx Vivado Design Suite and VHDL modelling language. Therefore, this lab covers information located throughout the whole tutorial document.
- **Lab 3: "Creating Frequency Trigger Module"** - covers the information presented in the **sub-chapters 2.2, 2.4, 2.4.1** of **Chapter 2: "Frequency Trigger"** of this tutorial.
- **Lab 4: "Frequency Trigger Verification"** - covers the information presented in the **sub-chapters 2.5, 2.6** of **Chapter 2: "Frequency Trigger"** of this tutorial.
- **Lab 5: "Creating Counter Module"** - covers the information presented in the **Chapter 3: "Counter"** of this tutorial.

- **Lab 6: "Creating Sine Package"** - covers the information presented in the **Chapter 4: "Sine Package"** of this tutorial.
- **Lab 7: "Creating Digital Sine Module"** - covers the information presented in the **Chapter 5: "Digital Sine"** of this tutorial.
- **Lab 8: "Creating Digital Sine Top Module"** - covers the information presented in the **Chapter 6: "Digital Sine Top"** of this tutorial.
- **Lab 9: "Creating PWM Module"** - covers the information presented in the **Chapter 7: "PWM"** of this tutorial.
- **Lab 10: "Creating Modulator Module"** - covers the information presented in the **Chapter 8: "Modulator"** of this tutorial.
- **Lab 11: "Creating XDC File"** - covers the information presented in the **sub-chapter 10.1** of **Chapter 10: "Design Implementation"** of this tutorial.
- **Lab 12: "Design Implementation"** - covers the information presented in the **sub-chapter 6.5** of **Chapter 6: "Digital Sine Top"** and **sub-chapters 10.2, 10.3, 10.4** of **Chapter 10: "Design Implementation"** of this tutorial.
- **Lab 13: "Vivado Logic Analyzer"** - covers the information presented in the **sub-chapter 11.1** of the **Chapter 11 "Debugging Design"** of this tutorial.
- **Lab 14: "Debug a Design using Integrated Vivado Logic Analyzer"** - covers the information presented in the **sub- chapter 11.2** of the **Chapter 11 "Debugging Design"** of this tutorial.
- **Lab 15: "Oscilloscope"** - covers the information presented in the **sub-chapter 11.3** of the **Chapter 11 "Debugging Design"** of this tutorial.
- **Lab 16: "Modulator Design Targeting Socius Development Board"** - covers the information presented in the **Chapter 12 "Modulator Design Targeting Socius Development Board"** of this tutorial.
- **Lab 17: "Designing with IPs - IP Packager"** - covers the information presented in the **sub-chapter 13.1** of the **Chapter 13 "Designing with IPs"** of this tutorial.
- **Lab 18: "Designing with IPs - IP Integrator"** - covers the information presented in the **sub-chapter 13.2** of the **Chapter 13 " Designing with IPs"** of this tutorial.
- **Lab 19: "Debugging IP Integrated Designs"** - covers the information presented in the **sub-chapter 13.3** of the **Chapter 13 " Designing with IPs"** of this tutorial.
- **Lab 20: "Creating Modulator IP Core with AXI4 Interface"** - covers the information presented in the **sub-chapter 13.4** of the **Chapter 13 " Designing with IPs"** of this tutorial.

1.4 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`.

Illustration 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.

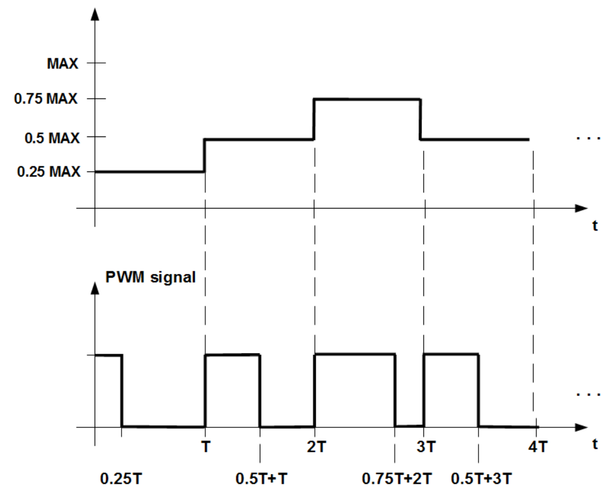


Figure 1.1: Example of the PWM signal

1.5 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.

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

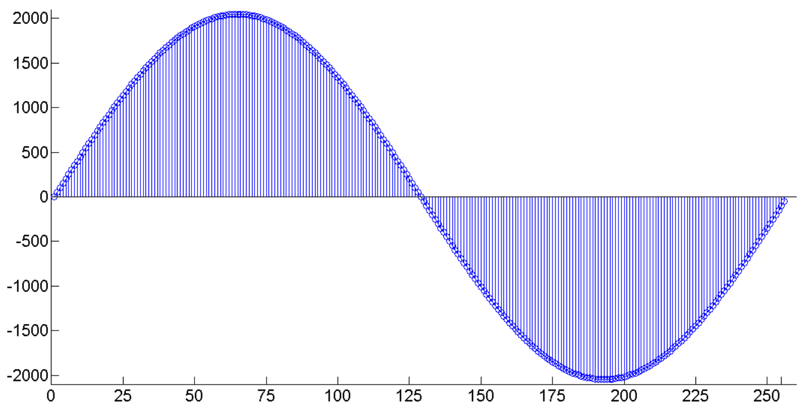


Figure 1.2: Sine wave with 256 samples

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 IEEE Math library and will be stored in an ROM memory.

Note: All of these information, such as what is the purpose of this tutorial, explanation what is the PWM signal, frequency calculations and block diagram as one possible solution for the modulator design, are illustrated in the **Lab 1: "Introduction"**.

Block diagram

Block diagram on the Illustration 1.3 shows the structure of one possible system that can be used to generate required PWM signals.

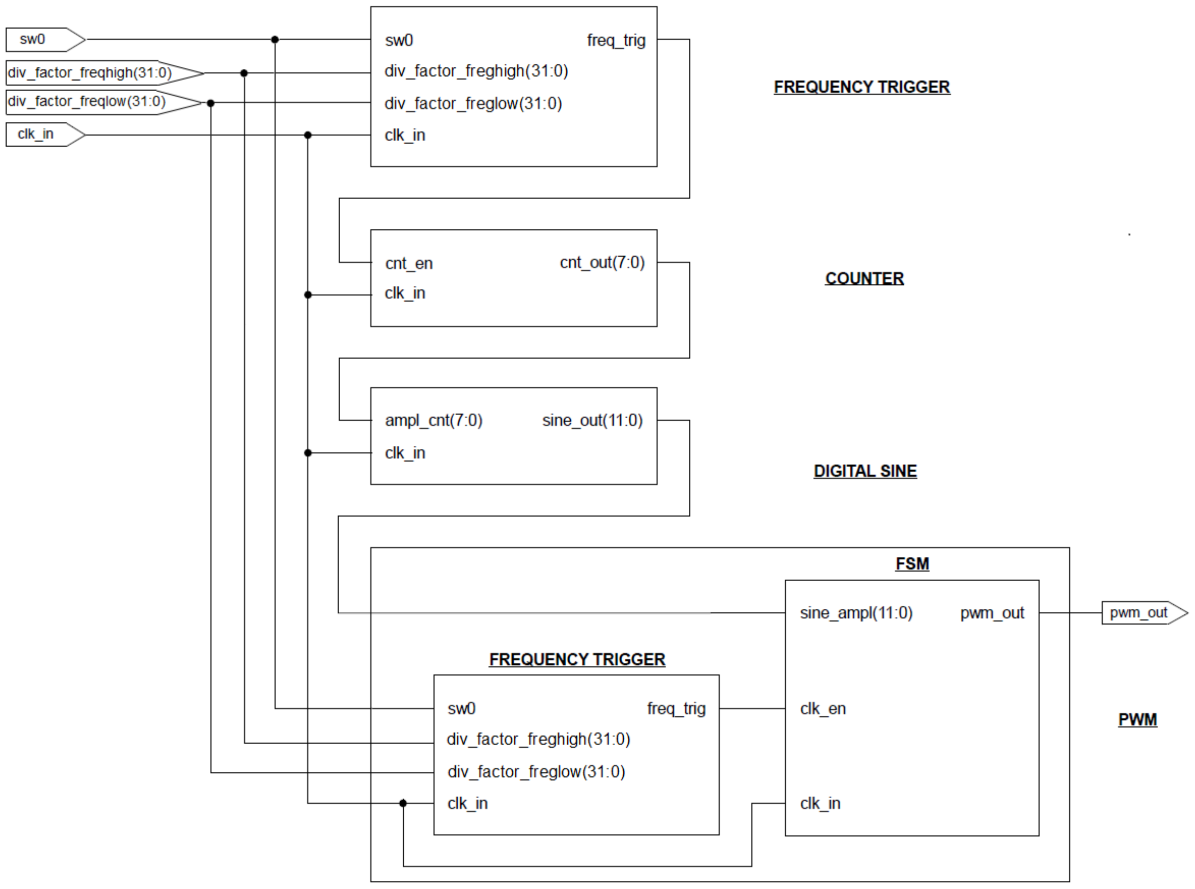


Figure 1.3: Block diagram

Let us briefly explain each module shown on the Illustration 1.3:

Frequency Trigger

This module will generate one output signal with two possible frequencies calculated in the Tables 1.1 and 1.2, one with 256 Hz and the second one with 896 Hz. Which frequency will be chosen depends on the position of the two-state on-board switch (sw0).

Counter

This module will be an universal (generic) counter. It's task will be to generate read addresses for the ROM where samples of the sine wave are stored. The speed of the counting will be controlled by the Frequency Trigger module, via freg_trig port, and the output of the Counter module will be an input of the Digital Sine module.

Digital Sine

This module will generate an digital representation of an analog (sine) signal with desired frequency. It will use the counter values as addresses to fetch the next value of the sine wave from the ROM.

In our case we will make an VHDL package with a parametrized sine signal. 2^8=256 unsigned amplitude values during one sine-period that will be stored into an ROM array.

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

Note: Don't forget to include the Sine package in the code of the Digital Sine module !

PWM

This module will generate an PWM signal modulated using the digital sine wave from the Digital Sine module. This module will be composed of two independent modules. One will be the Frequency Trigger, for generating two different frequencies and the second one will be the Finite State Machine (FSM), for generating the PWM signal.

Frequency Trigger - output from this module will be used to control the frequency at which FSM module works. As we have already said, in PWM signal information is represented as duty cycle value in each period of the signal. Since our digital sine signal can have 4096 possible values, there will also be 4096 different duty cycle values. This means that PWM's FSM must operate at frequency that is 4096 times higher than the one used by the Digital Sine module.

FSM - is necessary to generate the PWM signal. It will generate the PWM signal with correct duty cycle for each period based on the current amplitude value of digital sine signal, that is stored in the ROM.

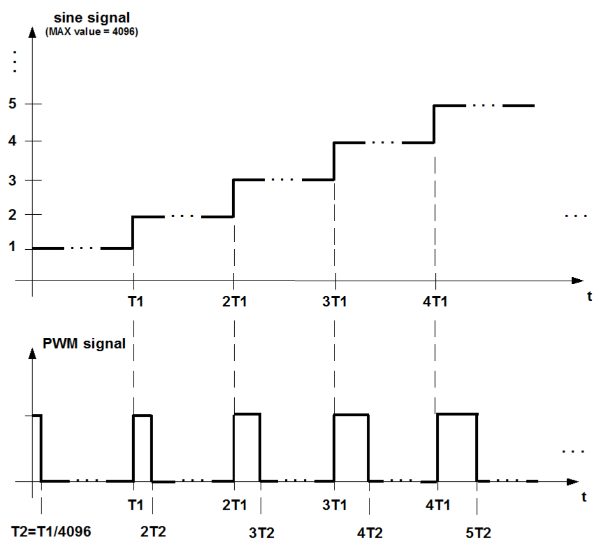


Figure 1.4: Details of PWM signal generation

Design steps

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

On the Illustration 1.5 are shown steps in designing modules of this lab:

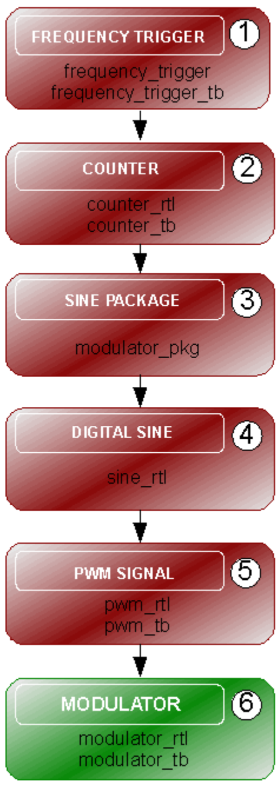


Figure 1.5: Project Design Steps

- First we will create the Frequency Trigger module that will provide one output signal with two possible frequencies.
- Then, we will create the Counter module, that will generate read addresses for the ROM where samples of the sine wave will be stored.
- Then, we will create an VHDL package with a parametrized sine signal.
- After that, we will create the Digital Sine module, where we will generate an digital representation of an analog (sine) signal and where we will include the Sine package.
- After that, we will create PWM signal with the PWM module.
- At the end, we will create Modulator module where we will merge all the previously designed modules into one big design.

Note: In the **Lab 2: "Using the Vivado Tool"** is illustrated the structure and the interface of this project, which modules we will have in our design and what will be our design steps.

1.6 Design Flow

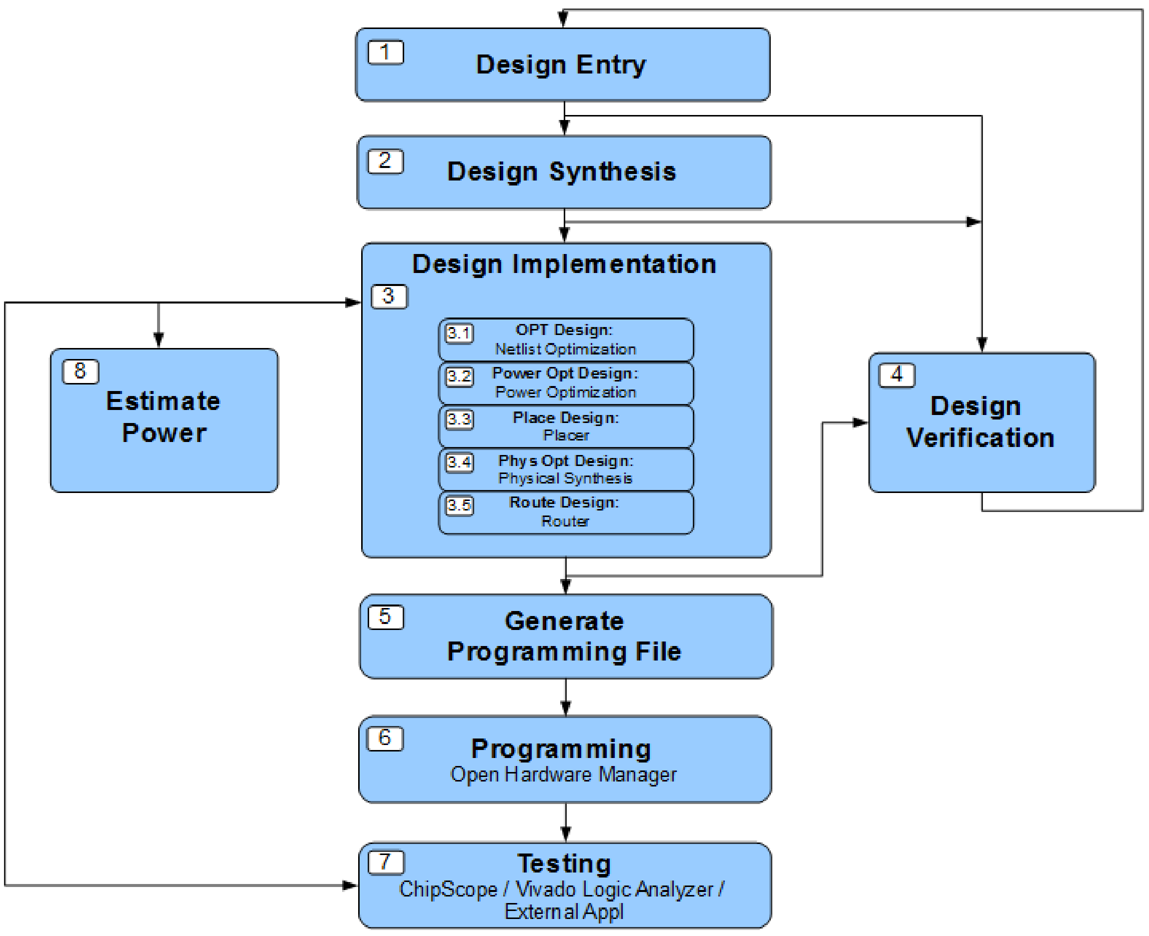

1 Design Entry
2 Design Synthesis
Design Implementation
3
3.1 OPT Design: Netlist Optimization
3.2 Power Opt Design: Power Optimization
3.3 Place Design: Placer
3.4 Phys Opt Design: Physical Synthesis
3.5 Route Design: Router
4 Design Verification
5 Generate Programming File
6 Programming Open Hardware Manager
7 Testing ChipScope / Vivado Logic Analyzer / External Appl
8 Estimate Power

Figure 1.6: Design Flow

On the Illustration 1.6 is presented the simplified Vivado Design flow.

- ***Design Entry*** **-** the first step in creating a new design is to specify it's structure and functionality. This can be done either by writing an HDL model using some text editor or drawing a schematic diagram using schematic editor.

- ***Design Synthesis*** - next step in the design process is to transform design specification (RTL design specification) into a more suitable representation (gate-level representation) that can be further processed in the later stages in the design flow. This representation is called the netlist. Prior to netlist creation synthesis tool checks the model syntax and analyse the hierarchy of your design which ensures that your design is optimized for the design architecture you have selected.

  Vivado synthesis enables you to configure, launch and monitor synthesis run. The Vivado IDE displays the synthesis result and creates report files. You can launch multiple synthesis runs also, simultaneously or serially.

- ***Design Implementation***

  Implementation step maps netlist produced by the synthesis tool onto particular device's internal structure.

  Vivado implementation includes all steps necessary to place and route the netlist onto the FPGA device resources, while meeting the design's logical, physical and timing constraints.

  Vivado implementation enables you to configure, launch and monitor implementation runs. The Vivado IDE displays the implementation result and creates report files. You can launch multiple implementation runs also, simultaneously or serially.

- ***Design Verification*** - is very important step in design process. A verification is comprised of seeking out problems in the HDL implementation in order to make it compliant with the design specification. A verification process reduces to extensive simulation of the HDL code. Design Verification is usually performed using two approaches: Simulation and Static Timing Analysis.

  There are two types of simulation:

  – ***Functional (Behavioral) Simulation*** - enables you to simulate or verify a code syntax and functional capabilities of your design. This type of simulation tests your design decisions before the design is implemented and allows you to make any necessary changes early in the design process. In functional (behavioral) simulation no timing information is provided.

  – ***Timing Simulation*** - allows you to check does the implemented design meet all functional and timing requirements and behaves as you expected. The timing simulation uses the detailed information about the signal delays as they pass through various logic and memory components and travel over connecting wires. Using this information it is possible to accurately simulate the behaviour of the implemented design. This type of simulation is performed after the design has been placed and routed for the target PLD, because accurate signal delay information can now be estimated. A process of relating accurate timing information with simulation model of the implemented design is called *Back-Annotation*.

  – ***Static Timing Analysis*** - helps you to perform a detailed timing analysis on routed FPGA design. This analysis can be useful in evaluating timing performance of the logic paths, especially if your design doesn't meet timing requirements. This method doesn't require any type of simulation.

- ***Generate Programming File*** - this option runs Xilinx bitstream generation program, to create a bitstream file that can be downloaded to the device.

- ***Programming*** - Vivado Design Suite offers **Open Hardware Manager** option that uses the native in-system device programming capabilities that are built into the Vivado IDE. Hardware manager uses the output from the Generate Programming File process to configure your target device.

- ***Testing*** - after configuring your device, you can debug your FPGA design using Vivado Logic Analyzer or some external logic analyzer.

- ***Estimate Power*** - after implementation, you can use the analyzer for estimation and power analysis. With this tool you can estimate power, based on the logic and routing resources of the actual design.

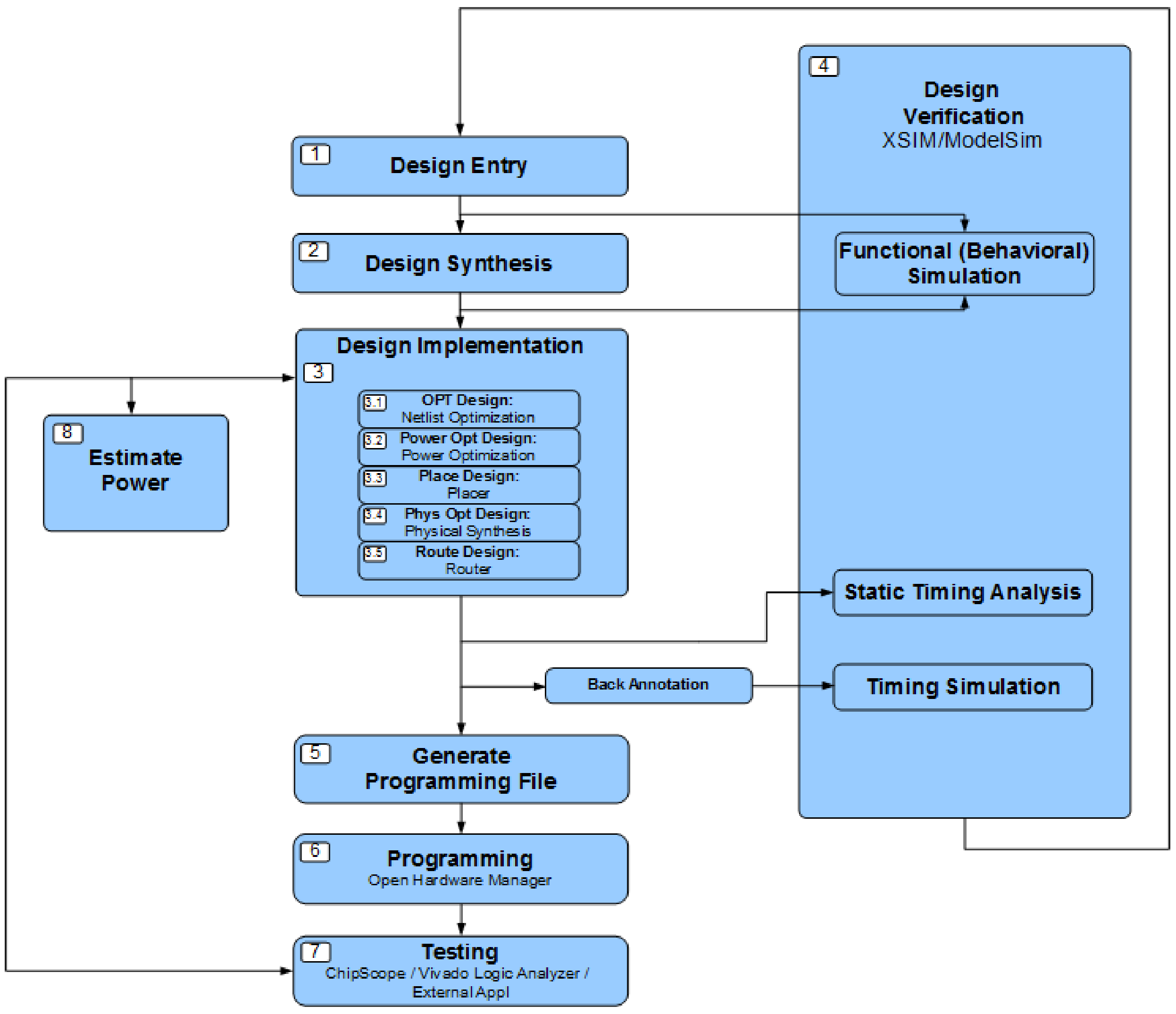

1 Design Entry
2 Design Synthesis
Design Implementation
3
3.1 OPT Design: Netlist Optimization
3.2 Power Opt Design: Power Optimization
3.3 Place Design: Placer
3.4 Phys Opt Design: Physical Synthesis
3.5 Route Design: Router
4 Design Verification XSIM/ModelSim
Functional (Behavioral) Simulation
Static Timing Analysis
Back Annotation
Timing Simulation
5 Generate Programming File
6 Programming Open Hardware Manager
7 Testing ChipScope / Vivado Logic Analyzer / External Appl
8 Estimate Power

Figure 1.7: Design Verification Steps

Note : In the Lab 2: "Using the Vivado Tool" you can also find a short description about each step from the Vivado Design Flow.

1.7 Vivado Design Suite and it's Use Modes

The Vivado Design Suite is a entirely new tool suite, designed to improve overall productivity of designing, integrating and implementing with the Xilinx 7 Series, Zynq-7000 All Programmable (AP) SoC, and UltraScale device families. The entire ISE Design Suite flow is replaced by the new Vivado Design Suite tools. It replaces all of the ISE Design Suite point tools, such as Project Navigator, Xilinx Synthesis Technology (XST), Implementation, CORE Generator tool, Timing Constraints Editor, ISE Simulator (ISim), ChipScope Analyzer, Xilinx Power Analyzer, FPGA Editor, PlanAhead design tool, and Smart-Xplorer. All of these capabilities are now built directly into the Vivado Design Suite and leverage a shared scalable data model.

Important: The Vivado IDE supports designs that target 7 Series devices, Zynq-7000 All Programmable (AP) SoC, and UltraScale devices.

Built on the shared scalable data model of the Vivado Design Suite, the entire design process can be executed in memory without having to write or translate any intermediate file formats (like it was in the ISE Design Suite flow). This accelerates runtimes, debug, and implementation while reducing memory requirements.

All of the Vivado Design Suite tools are written with a native Tool Command Language (Tcl) interface. All of the commands and options available in the Vivado IDE are accessible through Tcl. A Tcl script can contain Tcl commands covering the entire design synthesis and implementation flow, including all necessary reports generated for design analysis at any point in the design flow.

You can interact with the Vivado Design Suite using:

- GUI-based commands in the Vivado IDE
- Tcl commands entered in the Tcl Console in the Vivado IDE, in the Vivado Design Tcl shell outside the Vivado IDE, or saved to a Tcl script file that is run either in the Vivado IDE or in the Vivado Design Suite Tcl shell
- A mix of GUI-based and Tcl commands

The Vivado Design Suite supports the following industry design standards:

- Tcl
- AXI4, IP-XACT
- Synopsys design constraints (SDC)
- Verilog, VHDL, System Verilog
- SystemC, C, C++

The entire solution is, as we already said, native Tcl based, with support for SDC and Xilinx design constraints (XDC) formats. Broad Verilog, VHDL, and SystemVerilog support for synthesis enables easier FPGA adoption. Using standard IP interconnect protocol, such as AXI4 and IP-XACT, enables faster and easier system-level design integration.

There are two design flow modes in the Vivado Design Suite:

- ***Project Based Mode*** - You can run this mode in the Vivado IDE. In the Project Based Mode you create a project in the Vivado IDE, and the Vivado IDE automatically saves the state of the design, generates reports and messaging, and manages source files. A runs infrastructure is used to manage the automated synthesis and implementation process and to track run status. The entire design flow can be run with a single click within the Vivado IDE. The Vivado GUI provides high levels of automation, project management, and easy-of-use features.
- ***Non-Project Batch Mode*** - You can run this mode using Tcl commands or scripts. In the Non-Project Batch Mode you have full control of the design flow, but the Vivado tools do not automatically manage source files or report the design states. When working in Non-Project Batch Mode, sources are accessed from their current locations and the design is compiled through the flow memory. Each design step is run individually using Tcl commands. You can save design checkpoints and create reports at any stage of the design process using Tcl commands. You are viewing the active design in memory, so any changes are automatically passed forward in the flow.

Recommendation: Project Based Mode is the easiest way to get acquainted with the Vivado tool behaviour and Xilinx recommendations.

1.7.1 Differences between Project and Non-Project Mode

Some of the Project Mode features, such as source file and results management, saving design and tool configuration, design status and IP integration are not available in Non-Project Mode.

In Project Mode, the Vivado IDE tracks the history of the design and stores design information. Because, many features are automated, you have less control using this mode.

In Non-Project Mode, each action is executed using a Tcl command. All of the processing is done in memory, so no files or reports are generated automatically. Each time you compile the design, you must define all of the sources, set all tool and design configuration parameters, launch all implementation commands, and specify report files to generate. Because, the project is not created on disk, source files remain in their original locations and run output is only created where you specify. The flow provides you with all of the power of Tcl commands and full control over the entire design process.

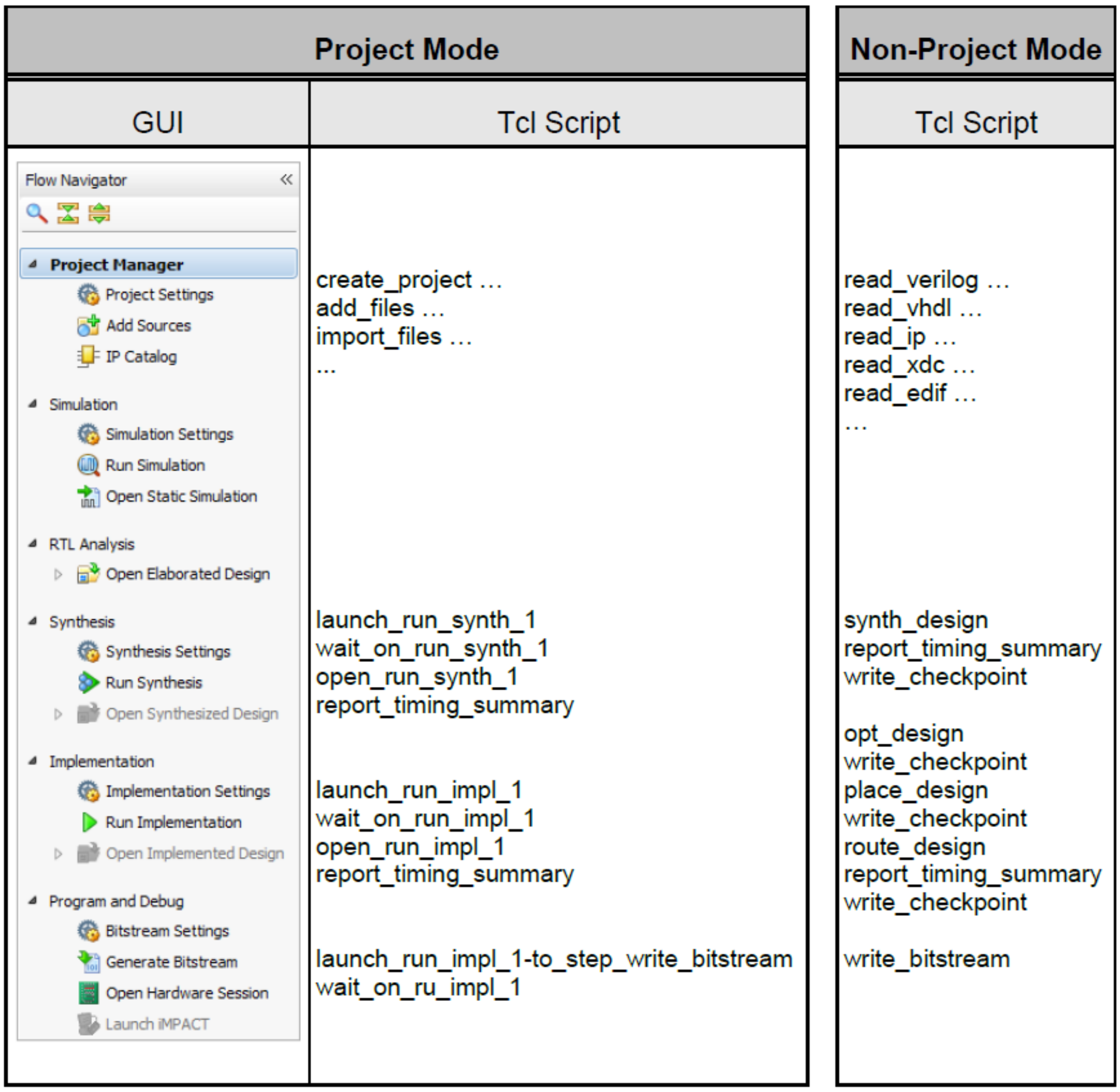
The following table highlights the feature differences between Project and Non-Project Mode.

Table 1.3: Project VS. Non-Project Mode Features

Flow Element	Project Mode	Non-Project Mode
Design Source File Management	Automatic	Manual
Flow Navigation	Guided	Manual
Flow Customization	Limited	Unlimited
Reporting	Automatic	Manual
Analysis Stages	Designs only	Designs and checkpoints

Note : Both these flows can be fully scripted and run in batch mode (no GUI).

Illustration 1.8 shows the differences between Project and Non-Project Mode Tcl commands.



Project Mode		Non-Project Mode
GUI	Tcl Script	Tcl Script
Flow Navigator Project Manager Project Settings Add Sources IP Catalog	create_project … add_files … import_files … …	read_verilog … read_vhdl … read_ip … read_xdc … read_edif … …
Simulation Simulation Settings Run Simulation Open Static Simulation		
RTL Analysis Open Elaborated Design		
Synthesis Synthesis Settings Run Synthesis Open Synthesized Design	launch_run_synth_1 wait_on_run_synth_1 open_run_synth_1 report_timing_summary	synth_design report_timing_summary write_checkpoint
Implementation Implementation Settings Run Implementation Open Implemented Design	launch_run_impl_1 wait_on_run_impl_1 open_run_impl_1 report_timing_summary	opt_design write_checkpoint place_design write_checkpoint route_design report_timing_summary write_checkpoint
Program and Debug Bitstream Settings Generate Bitstream Open Hardware Session Launch iMPACT	launch_run_impl_1-to_step_write_bitstream wait_on_ru_impl_1	write_bitstream

Figure 1.8: Project and Non-Project Mode Commands

Tcl commands depending on the mode you would like to use. The resulting Tcl scripts are different for each mode.

Some commands can be used in either mode, such as reporting commands. In some cases Tcl commands are specific to either Project and Non- Project Mode. Commands that are specific to one mode *must not be mixed* when creating scripts.

Project Mode includes GUI operations, which results in a Tcl command being executed in most cases. The Tcl commands appear in the Vivado IDE Tcl Console and are also captured in the ***vivado.jou*** file. **Journal** and **log** files provide a complete record of the Vivado IDE commands that are executed so the designer can construct scripts. You can use those files to develop scripts for use with either mode.

Journal file **(** ***vivado.jou*****)** - contains just the Tcl commands executed by the Vivado IDE. To open the journal file, select **File** -> **Open Journal File** option from the GUI

Log file **(*****vivado.log*****)** - contains all messages produced by the Vivado IDE, including Tcl commands and results, info/warning, error messages, etc. To open the log file, select **File ->** **Open Log File** option from the GUI

When we compare Vivado Project and Non-Project Modes there is one more difference, handling of **design checkpoints**. Design checkpoints enable you to take a snapshot of your design in its current state. The current netlist, constraints, and implementation results are stored in the design checkpoint.

Using design checkpoints, you can:

- restore your design if needed
- perform design analysis
- define constraints

You can write design checkpoints at different points in the flow. It is important to write design checkpoints after critical steps for design analysis and constraints definition.

When you use the Vivado IDE and the project infrastructure, you are automatically getting built-in checkpoints done for you. If you want finer control between each of the commands, you can manually write checkpoints at each stage in the Tcl non-project batch mode.

Important: With the exception of generating a bitstream, design checkpoints are not intended for use as starting points to continue the design process. They are merely snapshots of the design in its current state.

Following is the associated Tcl command:

- Tcl command: *write_checkpoint <file_name>*
- Tcl command: *read_checkpoint <file_name>*

In the Tables 1.4 and 1.5 are shown the basic Project and Non-Project Mode Tcl commands that control project creation, implementation and reporting.

Table 1.4: Basic Project Mode Tcl Commands

Command	Description
create_project	Creates the Vivado IDE project. Arguments include project name and location, design top module name, and target part.
add_files	Adds source files to the project. These include Verilog (.v), VHDL (.vhd or .vhdl), SystemVerilog (.sv), IP (.xco or xci), XDC constraints (.xdc or .sdc), embedded processor sub-systems from XPS (.xmp), and System Generator modules (.mdl). Individual files or entire directory trees can be scanned for legal sources and automatically added to the project.
set_property	Used for multiple purposes in the Vivado IDE. For projects, it can be used to define VHDL libraries for sources, simulation-only sources, target constraints files, tool settings, and so forth.
import_files	Imports the specified files into the current file set, effectively adding them into the project infrastructure. It is also used to define XDC files into constraints sets.
launch_runs launch_runs -to_step	Starts either synthesis or implementation and bitstream generation. This command encompasses the individual implementation commands as well as the standard reports generated after the run completes. It is used to launch all the steps of the synthesis or implementation process in a single command, and to track the tools progress trough that process. The -to_step option is used to launch the implementation process, including bitstream generation, in incremental steps.
wait_on_run	Ensures the run is complete before processing the next steps in the flow.
open_run	Opens either the synthesized design or implemented design for reporting analysis. A design must be opened before information can be queried using Tcl for reports, analysis, and so forth.
close_design	Closes the design in memory.
start_gui stop_gui	Invokes or closes the Vivado IDE with the current design in memory.

Table 1.5: Basic Non-Project Mode Tcl Commands

Command	Description
read_edif	Imports an EDIF or NGC netlist file into the Design Source fileset of current project.
read_verilog	Reads the Verilog (.v) and SystemVerilog (.sv) source files for the Non-Project Mode session.
read_vhdl	Reads VHDL (.vhd or .vhdl) source files for the Non-Project Mode session.
read_ip	Reads existing IP (.xco or .xci) project files for the Non-Project Mode session. The .ngc netlist is used from the .xco IP project. For .xci IP, the RTL is used for compilation or the netlist is used if it exists.
read_xdc	Reads the .sdc or .xdc format constraints source files for the Non- Project Mode session.
set_param set_property	Used for multiple purposes. For example, it can be used to define design configuration, tool settings, and so forth.
link_design	Compiles the design for synthesis if netlist sources are used for the session.
synth_design	Launches Vivado synthesis with the design top module name and target part as arguments.

opt_design	Performs high-level design optimization.
power_opt_design	Performs intelligent clock gating to reduce overall system power. This is an optional step.
place_design	Places the design.
phys_opt_design	Performs physical logic optimization to improve timing or routability. This is an optional step.
route_design	Routes the design.
report_$*$	Runs a range of standard reports, which can be run at any stage of the design process.
write_bitstream	Generates a bitstream file and runs DRCs.
write_checkpoint read_checkpoint	Save the design at any point in the flow. A design checkpoint consists of the netlist and constraints with any optimizations at that point in the flow as well as implementation results.
start_gui stop_gui	Invokes or closes the Vivado IDE with the current design in memory.

As we already said, both flows can be run using Tcl commands. You can use Tcl scripts to run the entire design flow. If you prefer to work directly with Tcl, you can interact with your design using Tcl commands.

Chapter 2

FREQUENCY TRIGGER

2.1 Description

- ***Usage* :** This module will generate one output signal with two possible frequencies, one with 256 Hz and the second one with 896 Hz. Which frequency will be chosen depends on the position of the two-state on-board switch (sw0).
- ***Block diagram:***

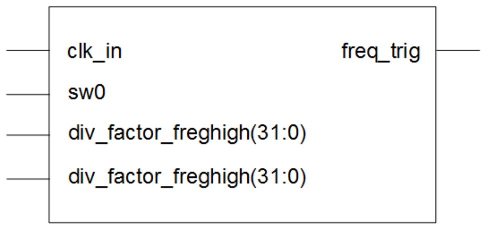


Figure 2.1: Frequency Trigger block diagram

- ***Input ports***:
  - **clk_in** : input clock signal
  - **sw0** : input signal from the on-board switch, used for changing output signal frequency
  - **div_factor_freqhigh** : input clock division factor when sw0 = ’1’
  - **div_factor_freqlow**: input clock division factor when sw0 = ’0’
- ***Output ports***:
  - **freq_trig** : output signal which frequency depends on the state of the sw0 input signal (256 Hz or 896 Hz)
- ***File name***: frequency_trigger_rtl.vhd

2.2 Creating a New Project

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

To create a new project:

Step 1. Launch the **Vivado** software: Select **Start ->** **All Programs ->** **Xilinx Design Tools ->** **Vivado 2016.4 ->** **Vivado 2016.4** and the Vivado **Getting Started** page will appear, see Illustration 2.2

As you can see from the illustration below, the **Getting Started** page contains links to create new or open an existing projects, and to view documentation.

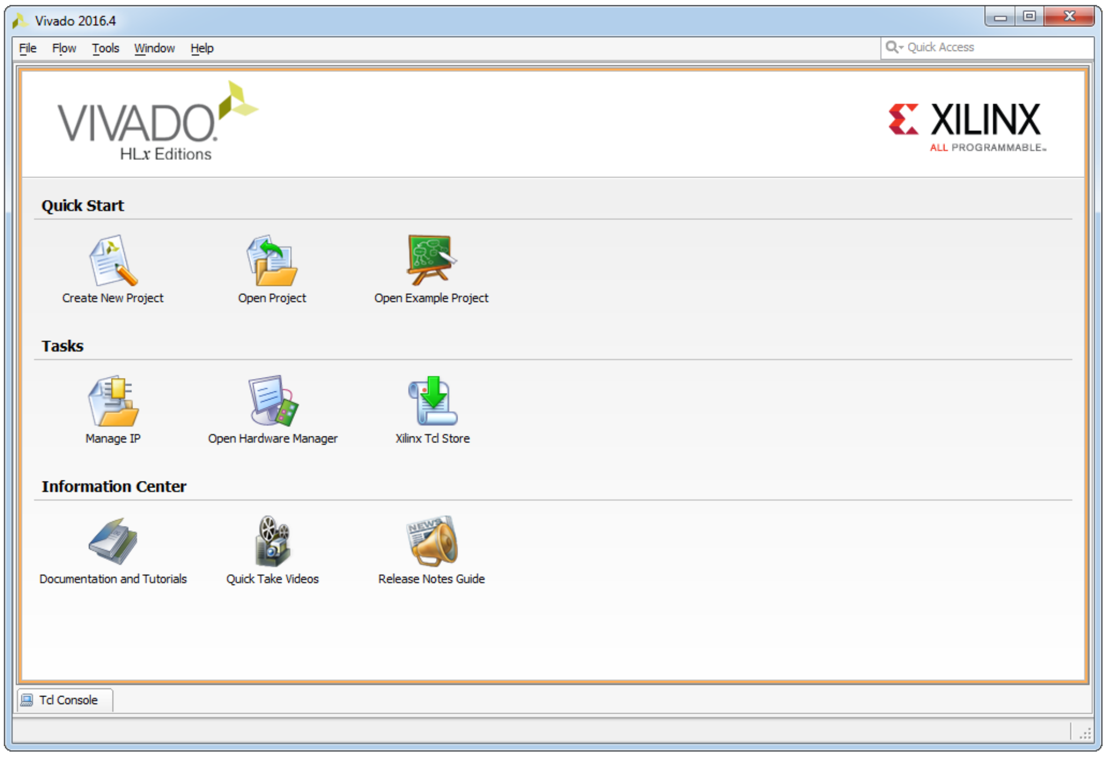
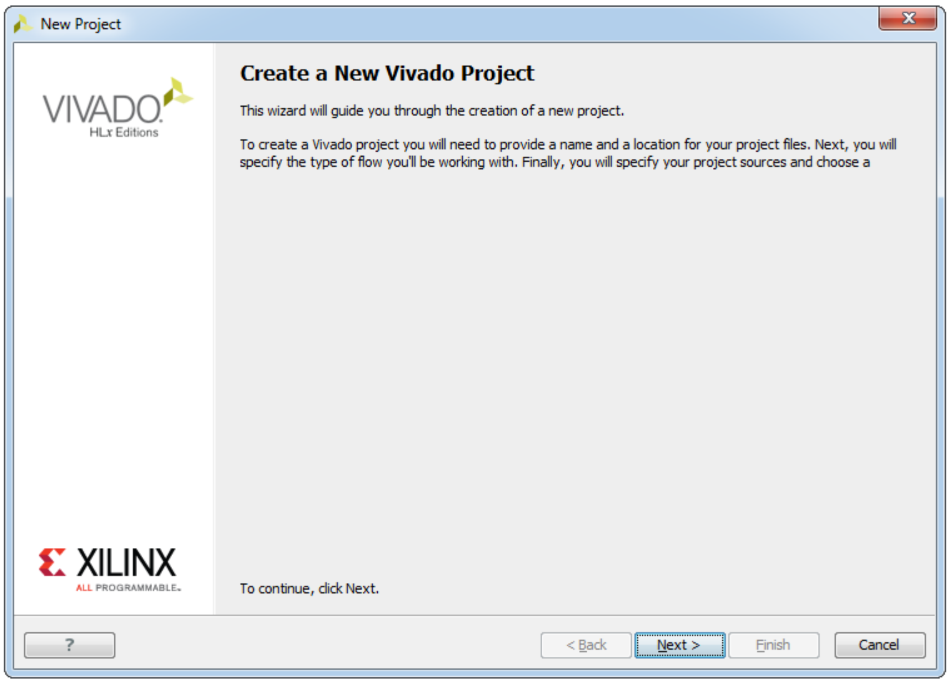


Figure 2.2: The Vivado Getting Started page

Step 2. On the **Getting Started** page, choose **Create New Project** option

Step 3. In the **Create a New Vivado Project** dialog box click **Next** and the wizard will guide you through the creation of a new project, see Illustration 2.3


New Project

VIVADO HLx Editions

Create a New Vivado Project

This wizard will guide you through the creation of a new project.

To create a Vivado project you will need to provide a name and a location for your project files. Next, you will specify the type of flow you'll be working with. Finally, you will specify your project sources and choose a

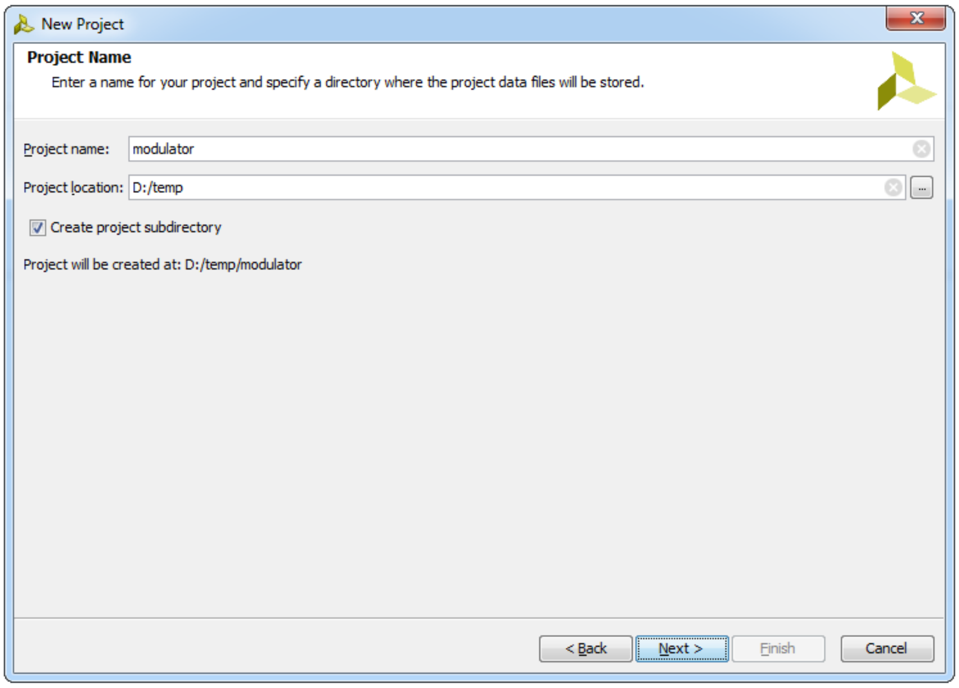
XILINX ALL PROGRAMMABLE.

To continue, click Next.

Figure 2.3: Create a New Vivado Project dialog box

Step 4. In the **Project Name** dialog box specify the name and the location of the new project:

- In the **Project name** field type **modulator** as the name of our project
- In the **Project location** field specify the location where our project data will be stored
- Leave **Create project subdirectory** option enabled, see Illustration 2.4



New Project

Project Name

Enter a name for your project and specify a directory where the project data files will be stored.

Project name: modulator

Project location: D:/temp

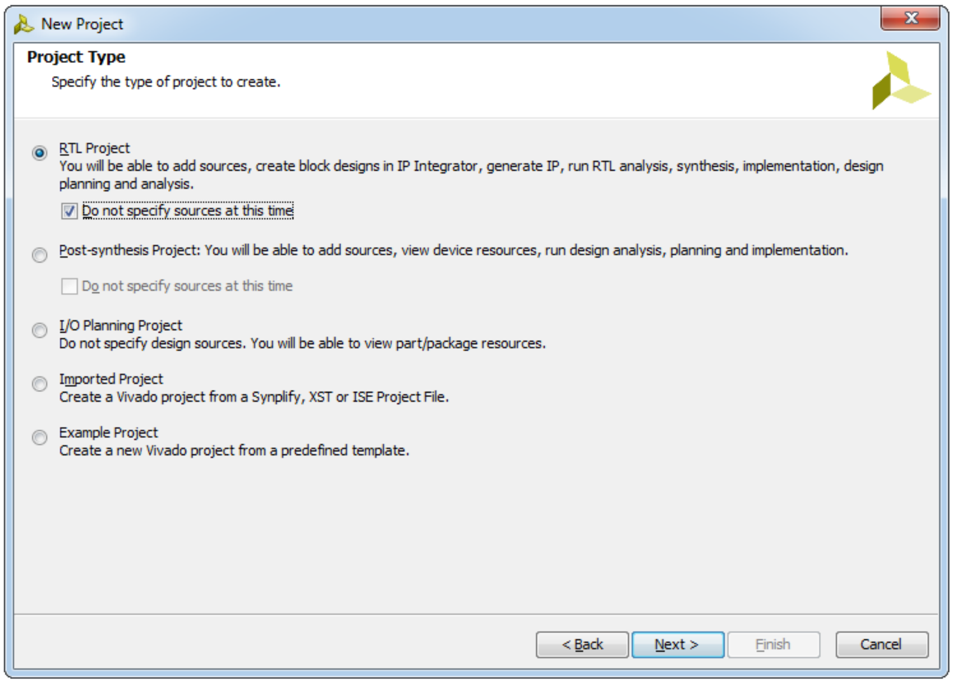
[✓] Create project subdirectory

Project will be created at: D:/temp/modulator

Figure 2.4: Project Name dialog box

Step 5. Click **Next**

Step 6. In the **Project Type** dialog box specify the type of project you want to create. In our case we will choose **RTL Project** option. Select **Do not specify sources at this time** also, see Illustration 2.5



New Project

Project Type

Specify the type of project to create.

[●] RTL Project
You will be able to add sources, create block designs in IP Integrator, generate IP, run RTL analysis, synthesis, implementation, design planning and analysis.

[✓] Do not specify sources at this time

[ ] Post-synthesis Project: You will be able to add sources, view device resources, run design analysis, planning and implementation.

[ ] Do not specify sources at this time

[ ] I/O Planning Project
Do not specify design sources. You will be able to view part/package resources.

[ ] Imported Project
Create a Vivado project from a Synplify, XST or ISE Project File.

[ ] Example Project
Create a new Vivado project from a predefined template.

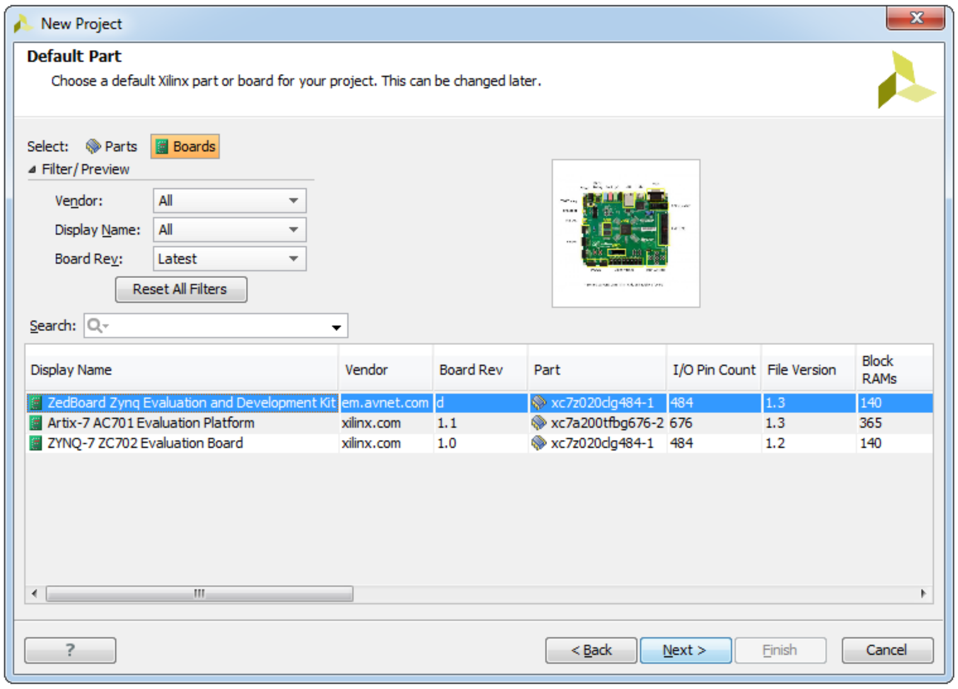
Figure 2.5: Project Type dialog box

As you can see from the Illustration above, four different types of the project can be created:

- *RTL Project* - The RTL Project environment enables you to add RTL source files and constraints, configure IP with the Vivado IP catalog, create IP subsystems with the Vivado IP integrator, synthesize and implement the design, and perform design planning and analysis.
- *Post-synthesis Project* - This type of project enables you to import third-party netlists, implement the design, and perform design planning and analysis.
- *I/O Planning Project* - With this type of project you can create an empty project for use with early I/O planning and device exploration prior to having RTL sources.
- *Imported Project* - This type of project enables you to import existing project sources from the ISE Design Suite, Xilinx Synthesis Technology (XST), or Synopsys Synplify.
- *Configure an Example Embedded Evaluation Board Design* - This type of project enables you to target the example Zynq-7000 or MicroBlaze embedded designs to the available Xilinx evaluation boards.

Step 7. Click **Next**

Step 8. In the **Default Part** 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 Illustration 2.6


New Project

Default Part

Choose a default Xilinx part or board for your project. This can be changed later.

Select: Parts Boards

Filter/ Preview

Vendor: All

Display Name: All

Board Rev: Latest

Reset All Filters

Search:

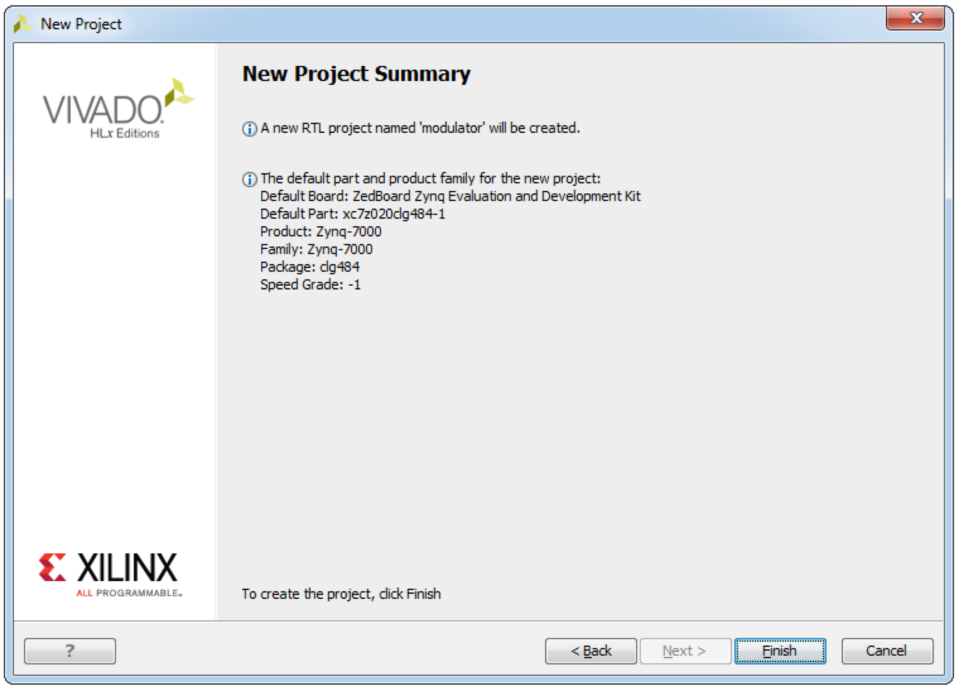
Display Name	Vendor	Board Rev	Part	I/O Pin Count	File Version	Block RAMs
ZedBoard Zynq Evaluation and Development Kit	em.avnet.com	d	xc7z020clg484-1	484	1.3	140
Artix-7 AC701 Evaluation Platform	xilinx.com	1.1	xc7a200tfbg676-2	676	1.3	365
ZYNQ-7 ZC702 Evaluation Board	xilinx.com	1.0	xc7z020clg484-1	484	1.2	140

Figure 2.6: Default Part dialog box

Step 9. Select **Zedboard Zynq Evaluation and Development Kit** as it is shown on the illustration above

Step 10. Click **Next**

Step 11. In the **New Project Summary** dialog box click **Finish** if you are satisfied with the summary of your project. If you are not satisfied, you can go back as much as necessary to correct all the questionable issues, see Illustration 2.7



New Project Summary

A new RTL project named 'modulator' will be created.

The default part and product family for the new project:
Default Board: ZedBoard Zynq Evaluation and Development Kit
Default Part: xc7z020clg484-1
Product: Zynq-7000
Family: Zynq-7000
Package: clg484
Speed Grade: -1

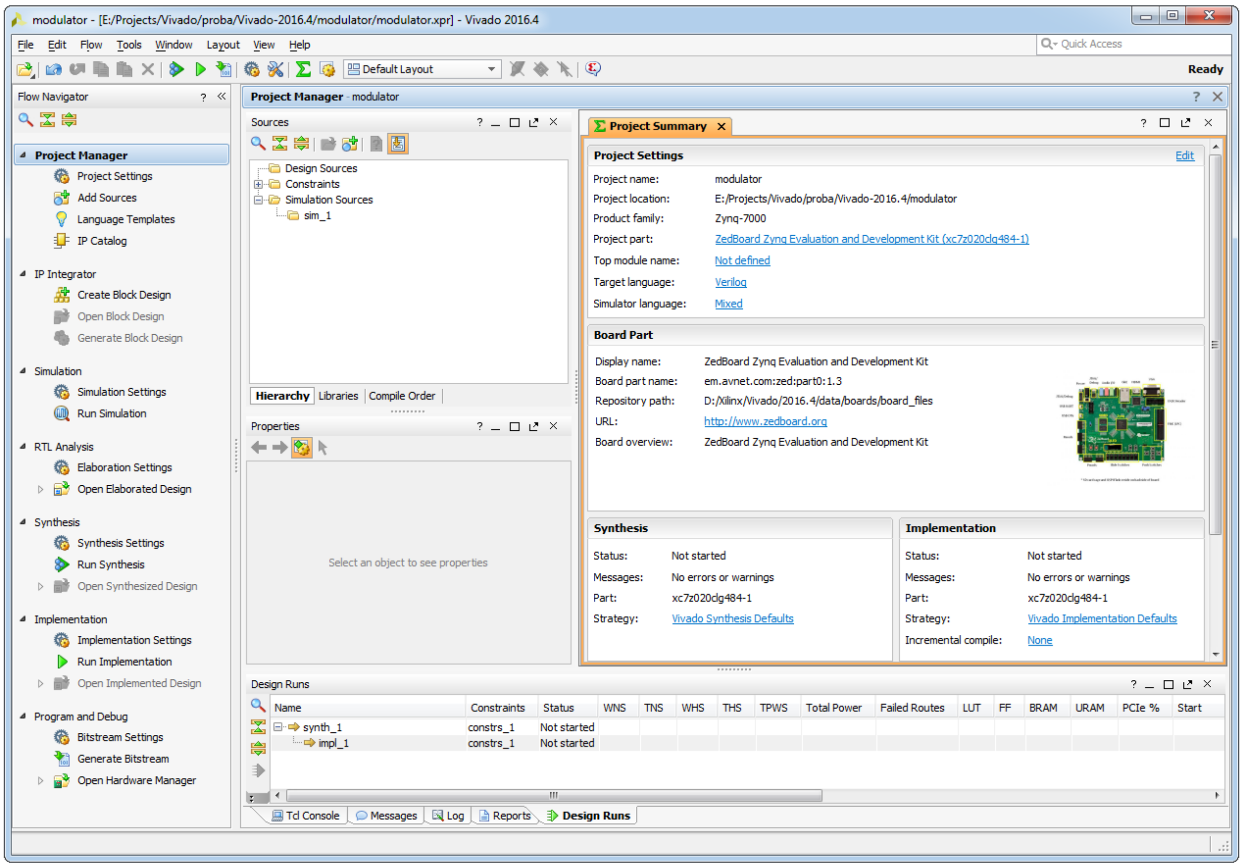
To create the project, click Finish

Figure 2.7: New Project Summary dialog box

After we finished with the new project creation, in a few seconds **Vivado IDE Viewing Environment** will appear, see Illustration 2.8.

When Vivado creates new project, it also creates a directory with the name and at the location that we specified in the GUI (see Illustration 2.4). That means that the all project data will be stored in the *project_name* (**modulator**) directory containing the following:

- *project_name.xpr* file - object that is selected to open a project (Vivado IDE project file)
- *project_name.runs* directory - contains all run data
- *project_name.srcs* directory - contains all imported local HDL source files, netlists, and XDC files
- *project_name.data* directory - stores floorplan and netlist data
- *project_name.sim* directory - contains all simulation data



Project Settings

Project name:	modulator
Project location:	E:/Projects/Vivado/proba/Vivado-2016.4/modulator
Product family:	Zynq-7000
Project part:	ZedBoard Zynq Evaluation and Development Kit (xc7z020clg484-1)
Top module name:	Not defined
Target language:	Verilog
Simulator language:	Mixed

Board Part

Display name:	ZedBoard Zynq Evaluation and Development Kit
Board part name:	em.avnet.com:zed:part0:1.3
Repository path:	D:/Xilinx/Vivado/2016.4/data/boards/board_files
URL:	http://www.zedboard.org
Board overview:	ZedBoard Zynq Evaluation and Development Kit

Synthesis

Status:	Not started
Messages:	No errors or warnings
Part:	xc7z020clg484-1
Strategy:	Vivado Synthesis Defaults

Implementation

Status:	Not started
Messages:	No errors or warnings
Part:	xc7z020clg484-1
Strategy:	Vivado Implementation Defaults
Incremental compile:	None

Design Runs

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAM	URAM	PCIe %	Start
synth_1	constrs_1	Not started													
impl_1	constrs_1	Not started													

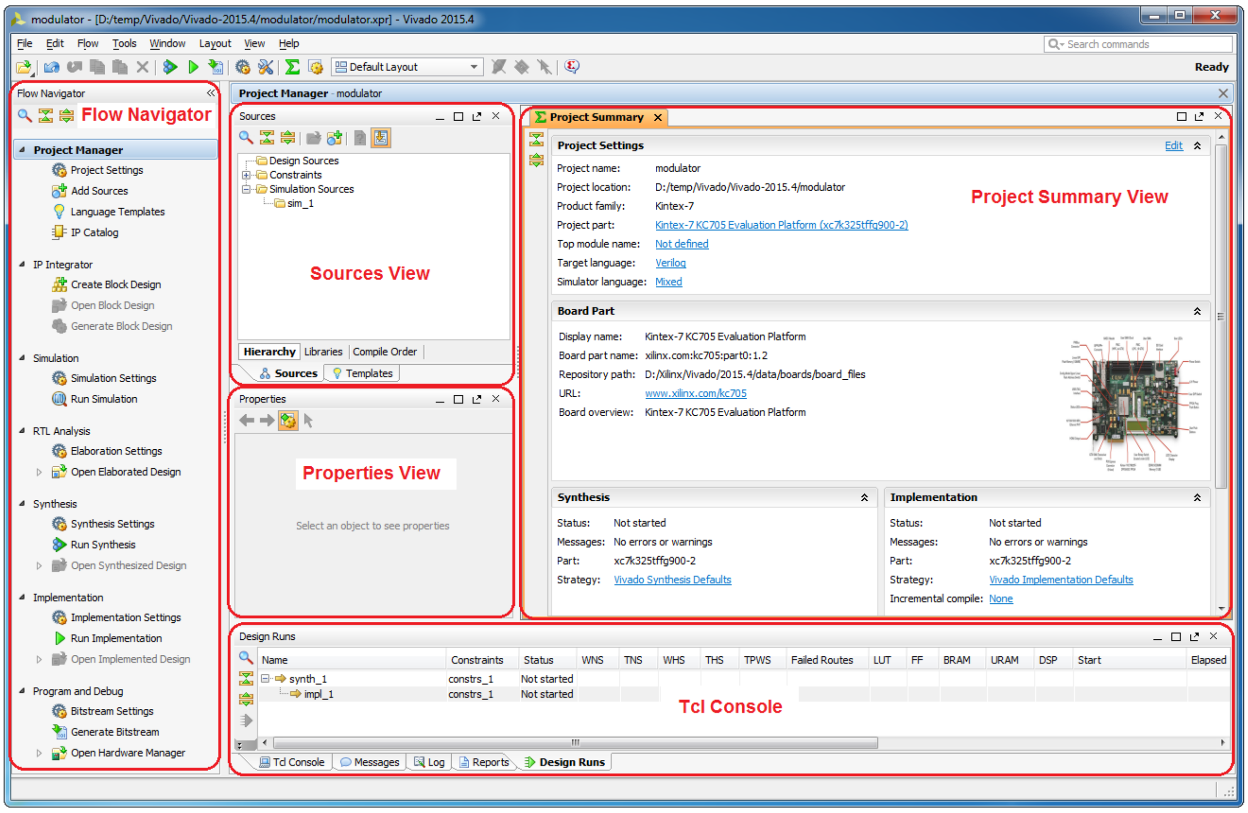
Figure 2.8: Vivado IDE Viewing Environment

2.3 Vivado Integrated Design Environment

The Vivado IDE can be used for a variety of purposes at various stages in the design flow and is very helpful at detecting design problems early in the design flow.

The Vivado IDE allows different file types to be added as design sources, including Verilog, VHDL, EDIF, NGC format cores, SDC, XDC, and TCL constraints files, and simulation test benches. These files can be stored in variety of ways using the tabs at the bottom of the Sources window: **Hierarchy**, **Library** or **Compile Order** , see Illustration 2.9.

By default, after launching, the Vivado IDE opens the Default Layout. Each docked window in the Vivado IDE is called a view, so you can find Sources View, Properties View, Project Summary View ans so on, see Illustration 2.9.



Project Settings

Project name:	modulator
Project location:	D:/temp/Vivado/Vivado-2015.4/modulator
Product family:	Kintex-7
Project part:	Kintex-7 KC705 Evaluation Platform (xc7k325tffg900-2)
Top module name:	Not defined
Target language:	Verilog
Simulator language:	Mixed

Board Part

Display name:	Kintex-7 KC705 Evaluation Platform
Board part name:	xilinx.com:kc705:part0:1.2
Repository path:	D:/Xilinx/Vivado/2015.4/data/boards/board_files
URL:	www.xilinx.com/kc705
Board overview:	Kintex-7 KC705 Evaluation Platform

Synthesis

Status:	Not started
Messages:	No errors or warnings
Part:	xc7k325tffg900-2
Strategy:	Vivado Synthesis Defaults

Implementation

Status:	Not started
Messages:	No errors or warnings
Part:	xc7k325tffg900-2
Strategy:	Vivado Implementation Defaults
Incremental compile:	None

Design Runs

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Failed Routes	LUT	FF	BRAM	URAM	DSP	Start	Elapsed
synth_1	constrs_1	Not started													
impl_1	constrs_1	Not started													

Figure 2.9: Vivado IDE Default Layout

Flow Navigator

The vertical toolbar present on the left side of the Vivado IDE is the **Flow Navigator**. The Flow Navigator provides control over the major design process tasks, such as project configuration, synthesis, implementation and bitstream creation.

Sources View

The **Sources** view displays the list of source files that has been added in the project.

- The **Design Sources** folder helps you keep track of VHDL and Verilog design source files and libraries.

- The **Constraints** folder helps you keep track of the constraints files.

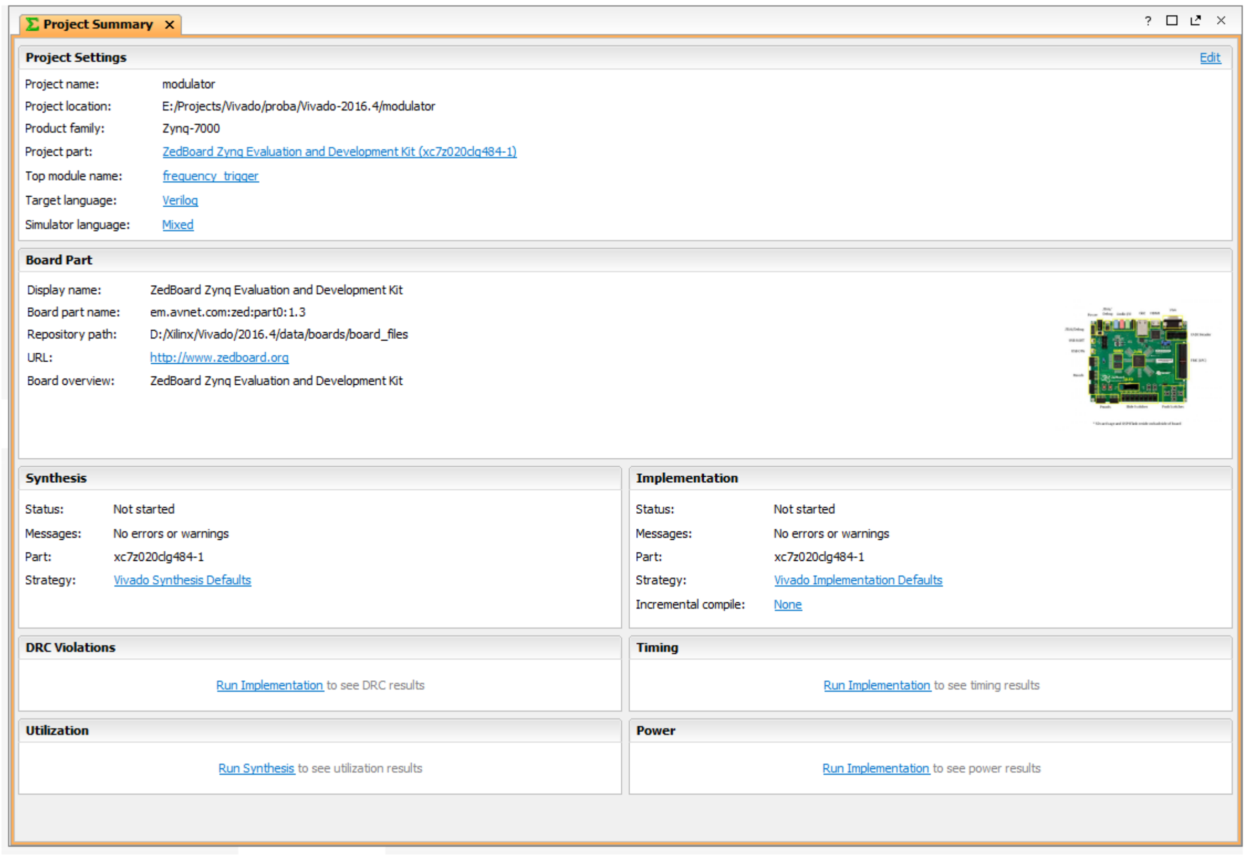
- The **Simulation Sources** folder helps keep track of VHDL and Verilog simulation sources source files and libraries.

Notice that the design hierarchy is displayed as default.

- In the **Libraries** tab, sources are grouped by file type, while the **Compile Order** tab shows the file order used for synthesis.

Project Summary View

The **Project Summary** view provides a brief overview of the status of different processes executed in the Vivado IDE, see Illustration 2.10.



Project Summary

Project Settings Edit

Project name:	modulator
Project location:	E:/Projects/Vivado/proba/Vivado-2016.4/modulator
Product family:	Zynq-7000
Project part:	ZedBoard Zynq Evaluation and Development Kit (xc7z020clg484-1)
Top module name:	frequency_trigger
Target language:	Verilog
Simulator language:	Mixed

Board Part

Display name:	ZedBoard Zynq Evaluation and Development Kit
Board part name:	em.avnet.com:zed:part0:1.3
Repository path:	D:/Xilinx/Vivado/2016.4/data/boards/board_files
URL:	http://www.zedboard.org
Board overview:	ZedBoard Zynq Evaluation and Development Kit

Synthesis

Status:	Not started
Messages:	No errors or warnings
Part:	xc7z020clg484-1
Strategy:	Vivado Synthesis Defaults

Implementation

Status:	Not started
Messages:	No errors or warnings
Part:	xc7z020clg484-1
Strategy:	Vivado Implementation Defaults
Incremental compile:	None

DRC Violations

Run Implementation to see DRC results

Timing

Run Implementation to see timing results

Utilization

Run Synthesis to see utilization results

Power

Run Implementation to see power results

Figure 2.10: Project Summary View

The **Project Settings** panel displays the project name, product family, project part, and top module name. Clicking a link in this panel you will open the Project Settings dialog box.

- The **Messages** panel summarizes the number of errors and warnings encountered during the design process.

- The **Synthesis** panel summarizes the state of synthesis in the active run. The synthesis panel also shows the target part and the strategy applied in the run.

- The **Implementation** panel summarizes the state of implementation in the active run. The Implementation panel also shows the target part and the strategy applied in the run.

Tcl Console

Below the Project Summary view, see Illustration 2.9, is the **Tcl Console** which echoes the Tcl commands as operations are performed. It also provides a means to control the design flow using Tcl commands.

2.4 Creating Module

To create a new module, follow the steps:

Step 1. In the Vivado **Flow Navigator**, click the **Add Sources** command (**Project Manager** option), see Illustration 2.11

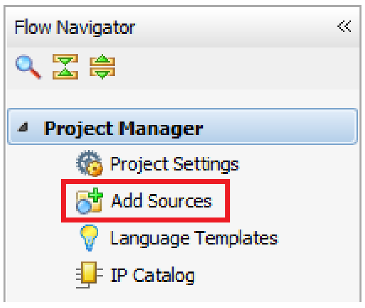
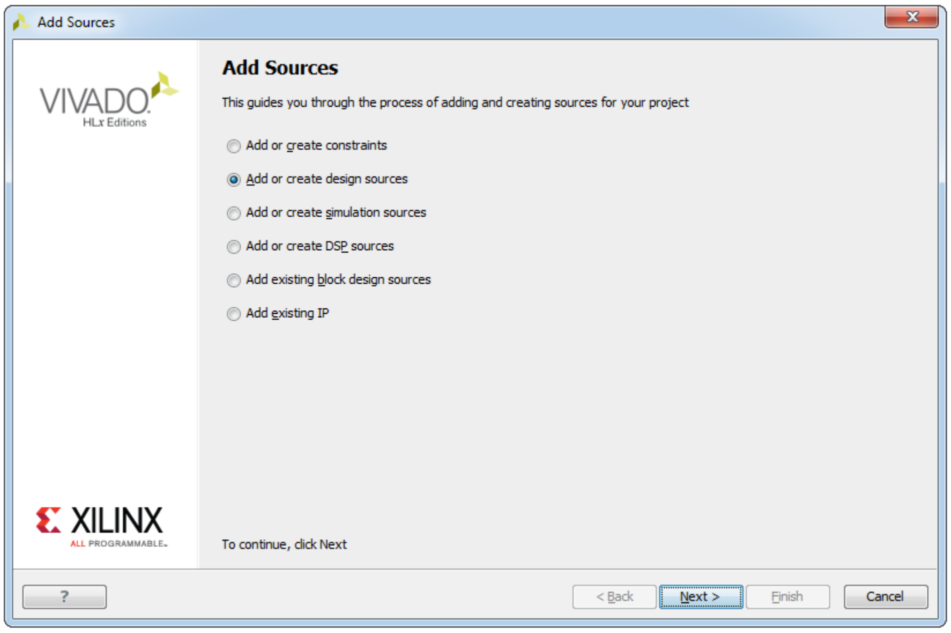


Figure 2.11: Add Sources command

Step 2. In the **Add Sources** dialog box, select **Add or create design sources** option to create the design source files in the project, see Illustration 2.12



Add Sources

This guides you through the process of adding and creating sources for your project

- Add or create constraints
- [✓ Add or create design sources]
- Add or create simulation sources
- Add or create DSP sources
- Add existing block design sources
- Add existing IP

To continue, click Next

Figure 2.12: Add Sources dialog box

Step 3. Click **Next**

Step 4. In the **Add or Create Design Sources** dialog box, click the + icon and select **Create File...** option to create a new file in the project, or just click **Create File** button, see Illustration 2.13

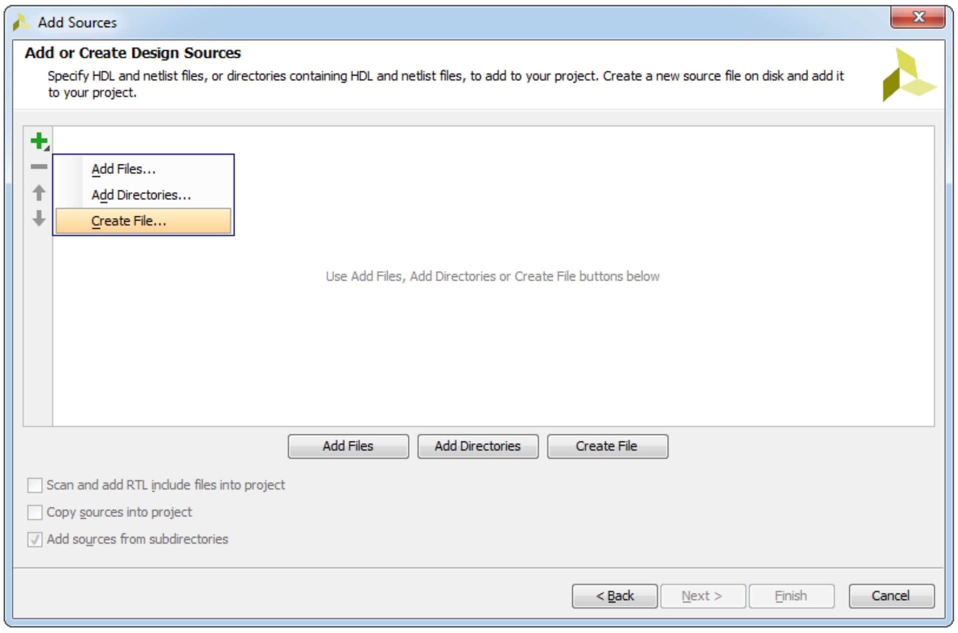

Add Sources
Add or Create Design Sources
Specify HDL and netlist files, or directories containing HDL and netlist files, to add to your project. Create a new source file on disk and add it to your project.
Add Files...
Add Directories...
Create File...
Use Add Files, Add Directories or Create File buttons below
Add Files
Add Directories
Create File
Scan and add RTL include files into project
Copy sources into project
Add sources from subdirectories
< Back
Next >
Finish
Cancel

Figure 2.13: Add or Create Design Sources dialog box - Create File option

Step 5. In the **Create Source File** dialog box, fill the file type, file name and file location on the following way, see Illustration 2.14:

- File type: **VHDL**
- File name: **frequency_trigger_rtl**
- File location: **Local to Project**

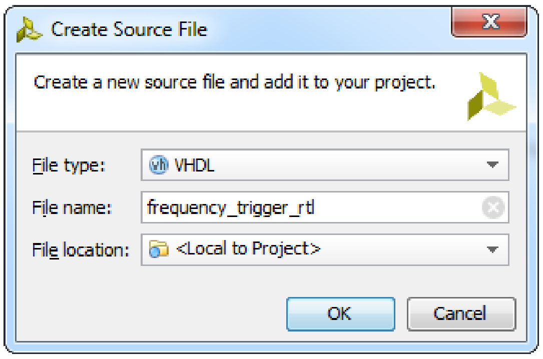
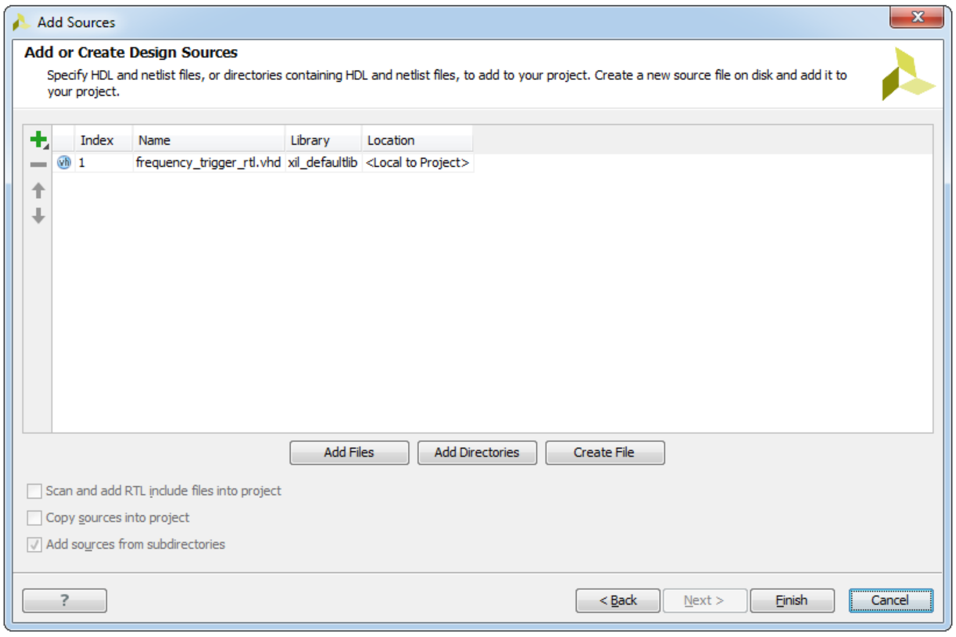

Create Source File
Create a new source file and add it to your project.
File type: VHDL
File name: frequency_trigger_rtl
File location: <Local to Project>
OK
Cancel

Figure 2.14: Create Source File dialog box

Step 6. Click **OK** to create a new source file (***frequency_trigger_rtl.vhd***) and add it into your project (***modulator***)

Step 7. Now your source file will appear in the **Add or Create Design Sources** dialog box, see Illustration 2.15. Click **Finish**


Add Sources

Add or Create Design Sources

Specify HDL and netlist files, or directories containing HDL and netlist files, to add to your project. Create a new source file on disk and add it to your project.

Index	Name	Library	Location
1	frequency_trigger_rtl.vhd	xil_defaultlib	<Local to Project>

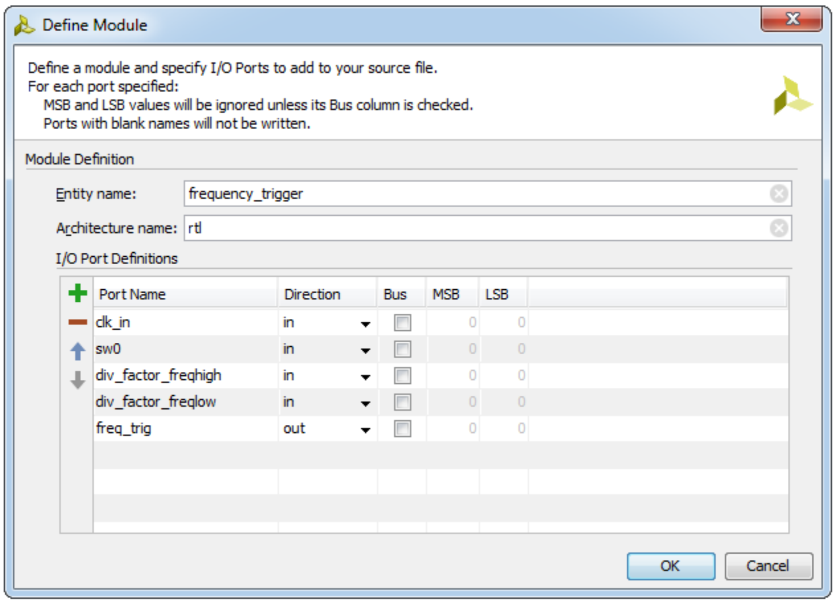
- [ ] Scan and add RTL include files into project
- [ ] Copy sources into project
- [✓] Add sources from subdirectories

Figure 2.15: Add or Create Design Sources dialog box with created file

Step 8. In the **Define Module** dialog box, Vivado IDE will automatically create **Entity name** (***frequency_trigger_rtl***) and **Architecture name** (***Behavioral***).

Please, rename **Entity name** to be ***frequency_trigger*** and **Architecture name** to be **rtl**, see Illustration 2.16

Step 9. Specify ports for the intended module as it is also shown on the Illustration 2.16


Define Module

Define a module and specify I/O Ports to add to your source file.
For each port specified:
MSB and LSB values will be ignored unless its Bus column is checked.
Ports with blank names will not be written.

Module Definition

Entity name: frequency_trigger

Architecture name: rtl

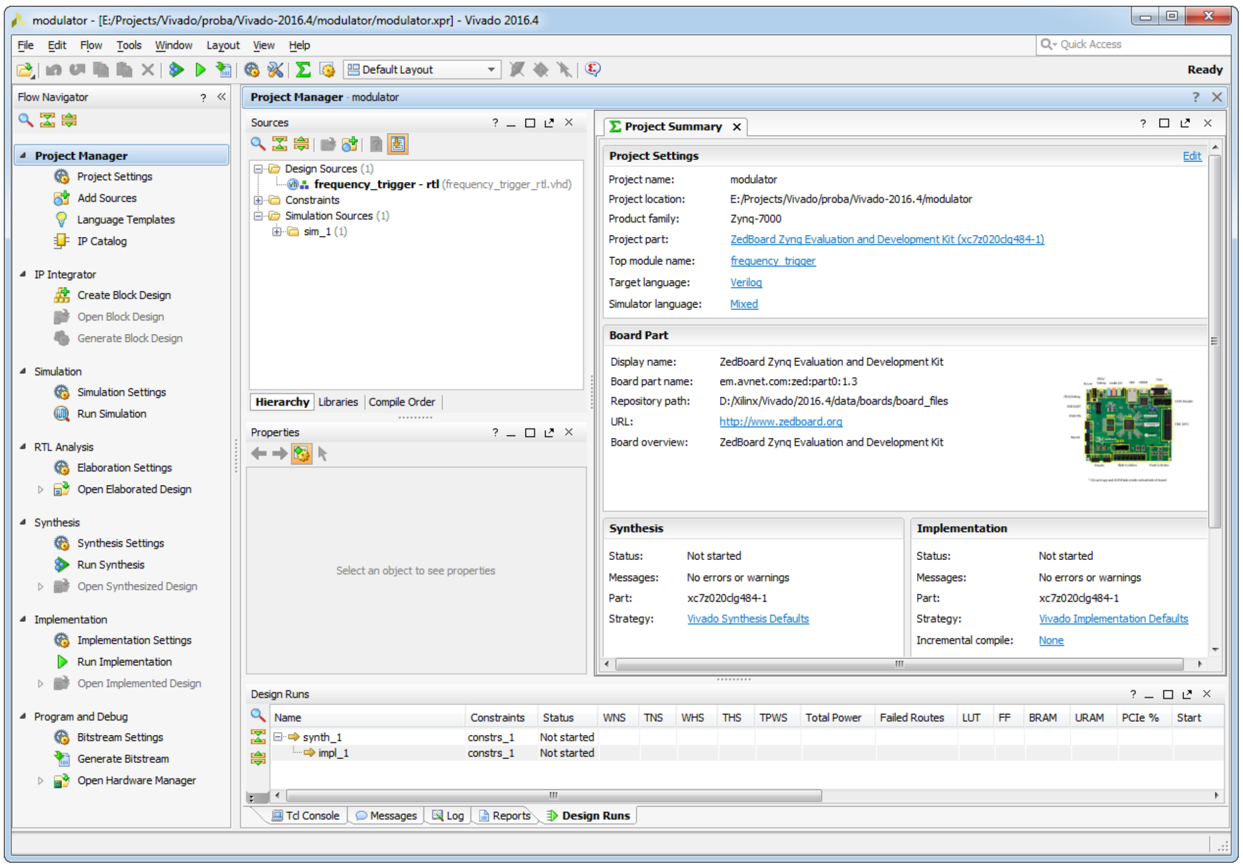
I/O Port Definitions

Port Name	Direction	Bus	MSB	LSB
clk_in	in	☐	0	0
sw0	in	☐	0	0
div_factor_freqhigh	in	☐	0	0
div_factor_freqlow	in	☐	0	0
freq_trig	out	☐	0	0

Figure 2.16: Define Module dialog box

Step 10. Click **OK** and your source file should appear under the **Design Sources** in the **Sources** view in the **Project Manager** window, see Illustration 2.17

Step 11. Double-click on the created ***frequency_trigger_rtl.vhd*** source file to see what the tool has created for us, see Illustration 2.18



Project Settings Edit

Project name:	modulator
Project location:	E:/Projects/Vivado/proba/Vivado-2016.4/modulator
Product family:	Zynq-7000
Project part:	ZedBoard Zynq Evaluation and Development Kit (xc7z020clg484-1)
Top module name:	frequency_trigger
Target language:	Verilog
Simulator language:	Mixed

Board Part

Display name:	ZedBoard Zynq Evaluation and Development Kit
Board part name:	em.avnet.com:zed:part0:1.3
Repository path:	D:/Xilinx/Vivado/2016.4/data/boards/board_files
URL:	http://www.zedboard.org
Board overview:	ZedBoard Zynq Evaluation and Development Kit

Synthesis

Status:	Not started
Messages:	No errors or warnings
Part:	xc7z020clg484-1
Strategy:	Vivado Synthesis Defaults

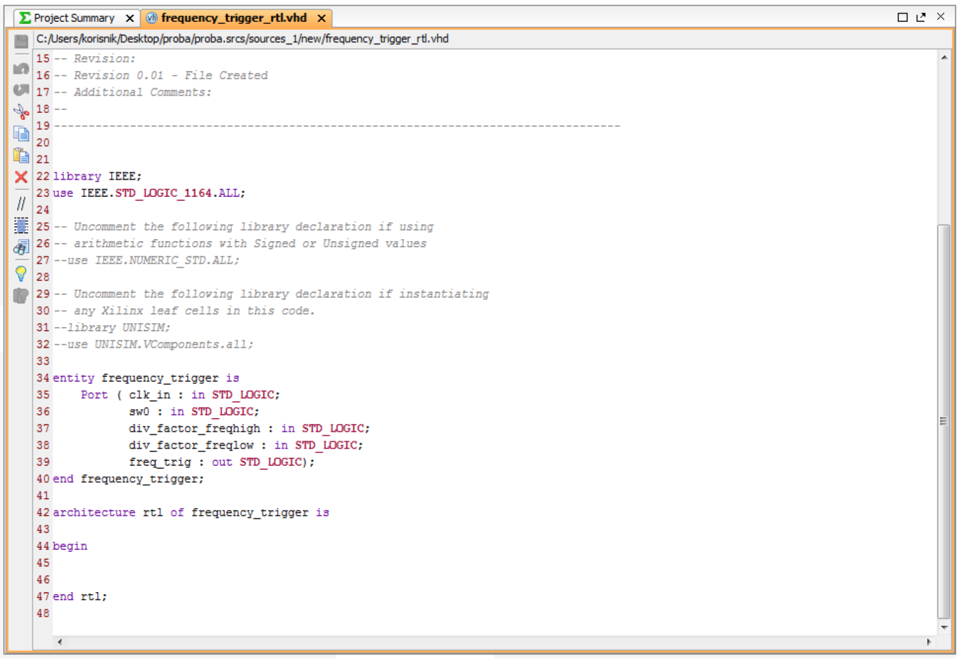
Implementation

Status:	Not started
Messages:	No errors or warnings
Part:	xc7z020clg484-1
Strategy:	Vivado Implementation Defaults
Incremental compile:	None

Design Runs

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAM	URAM	PCIe %	Start
synth_1	constrs_1	Not started													
impl_1	constrs_1	Not started													

Figure 2.17: Vivado IDE Viewing Environment after module creation



C:/Users/korisnik/Desktop/proba/proba.srcs/sources_1/new/frequency_trigger_rtl.vhd

```vhdl
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity frequency_trigger is
    Port ( clk_in : in STD_LOGIC;
           sw0 : in STD_LOGIC;
           div_factor_freqhigh : in STD_LOGIC;
           div_factor_freqlow : in STD_LOGIC;
           freq_trig : out STD_LOGIC);
end frequency_trigger;

architecture rtl of frequency_trigger is

begin


end rtl;
```

Figure 2.18: Automatically generated frequency_trigger_rtl.vhd source file

As we can see from the illustration above, the tool automatically creates a default header and the entity declaration based on the data that you entered.

Vivado editor is a powerful text editor with syntax highlighting for VHDL and Verilog HDLs. You can use Vivado editor to complete your VHDL/Verilog model of your design.

Important: The automatically generated code is not very handsome and clear, and the recommendation is to modify it. Here are the steps for modifying:

- create a complete module header as comment
- set all text to lower case
- remove all end descriptions (for example: *rtl* next to *end*) and all comments
- set all in/outputs in alphabetical order and comment them

Note: As you can see there are a lot of things for modifying. For better designs, our recommendation is not to use the GUI (Graphical User Interface) for module creation. Instead of that, create a module in an text editor, rename it to *module_name.vhd* and add it into your project.

Before we explain how to create a module using Vivado text editor, don't forget to remove *frequency_trigger_rtl.vhd* from the project. To remove some file from the project, do the following:

Step 1. Select the file that you want to remove

Step 2. Right-click on the selected file and choose **Remove File from Project...** option, see Illustration 2.19

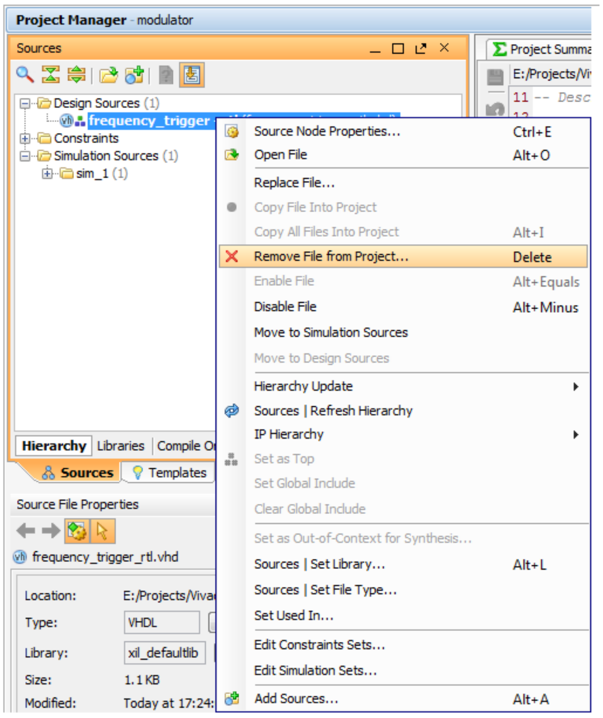
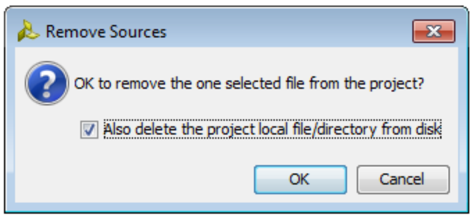


Figure 2.19: Remove File from Project option

Step 3. In the **Remove Sources** dialog box enable **Also delete the project local file/directory from disk** option, click **OK** and the file will be removed from the project, see Illustration 2.20


Remove Sources

OK to remove the one selected file from the project?

[✓] Also delete the project local file/directory from disk

Figure 2.20: Remove Sources dialog box

Note: Information about how to create the Frequency Trigger module, you can also find in the **Lab 3: "Creating Frequency Trigger Module".**

2.4.1 Creating a Module Using Vivado Text Editor

Design reuse is a common way of increasing a designer's productivity. It includes reusing a design modules that have been previously created and used within some other design. Since these modules are already created we need a way to add them to current project. This can be done using Add File option within Add Sources command. To illustrate how this can be accomplished, following procedure is presented. In this example we will first create VHDL model using Vivado text editor and save it as .vhd source file. Next we will add this source file to our project.

Here are the steps for creating a module using Vivado text editor:

Step 1. **Optional:** Launch **Vivado IDE** (if it is not already launched)

Step 2. **Optional:** Open "Modulator" project (**modulator.xpr**) (if it is not already opened)

Step 3. In the main Vivado IDE menu, click **File -> New File...** option to open Vivado text editor

Step 4 In the **New File** dialog box, type the name of your source file (e.g. ***frequency_trigger_rtl.vhd***) in the **File name** field and choose to save it into your working directory

Note: You can create new folder under your working directory, intended for storing source files.

Step 5. When you click **Save** , Vivado IDE will automatically open empty ***frequency_trigger_rtl.vhd*** source file in Vivado text editor

Step 6. Insert the **VHDL code** and add the *frequency_trigger_rtl* module header

Step 7. When you finish with module creation, click **File -> Save File** option from the main Vivado IDE menu, or just click Ctrl + S to save it

Step 8. In the Vivado **Flow Navigator** click the **Add Sources** command, see Illustration 2.21

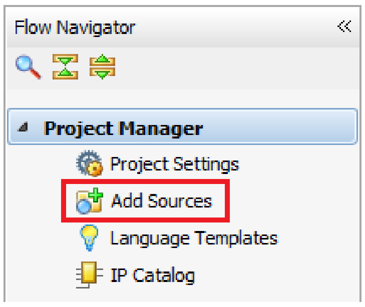
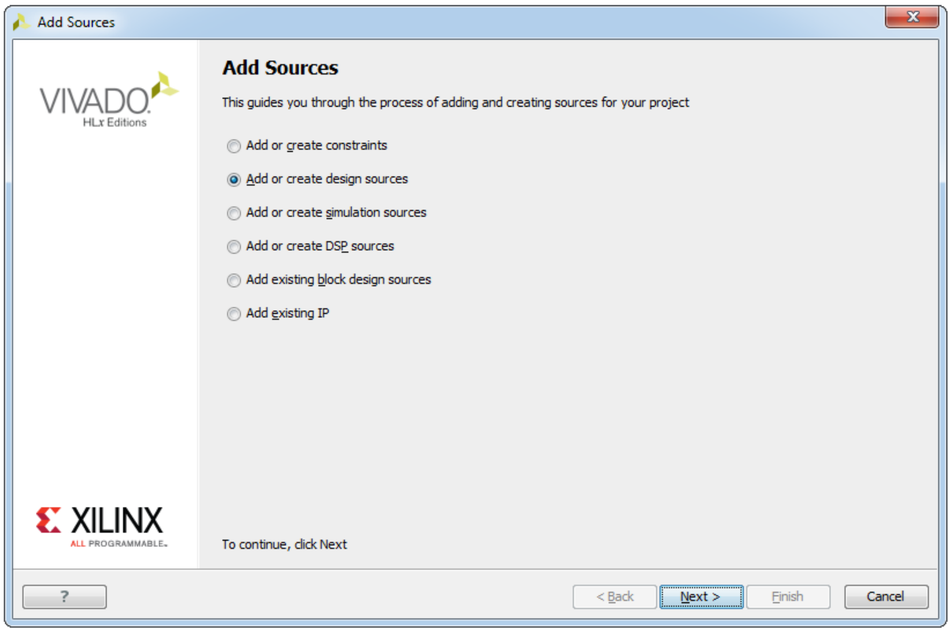


Figure 2.21: Add Sources command

Step 9. In the **Add Sources** dialog box, select **Add or create design sources** option to add the design source files into the project, see Illustration 2.22


Add Sources

Add Sources

This guides you through the process of adding and creating sources for your project

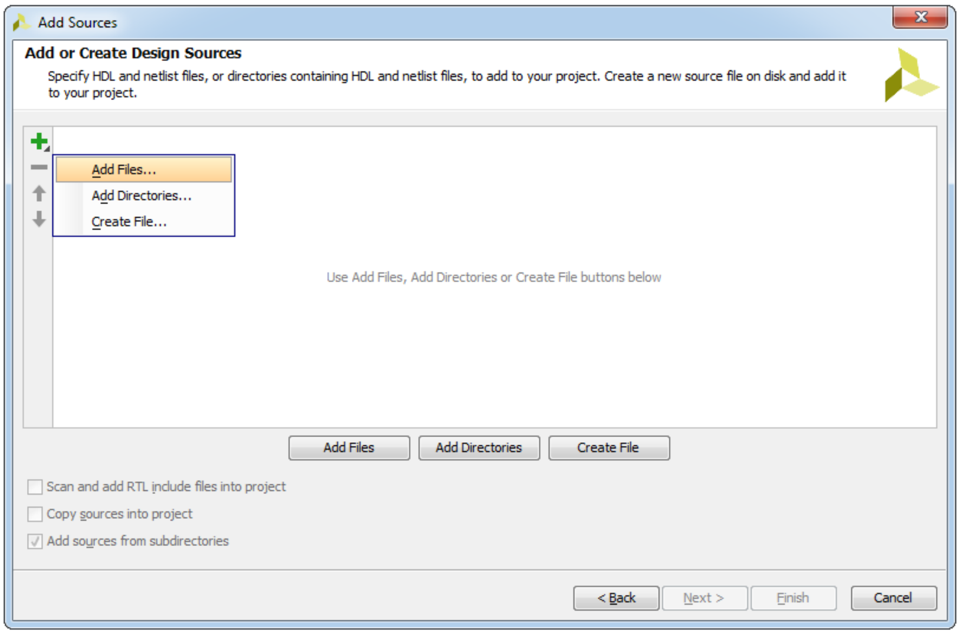
- ○ Add or create constraints
- ● Add or create design sources
- ○ Add or create simulation sources
- ○ Add or create DSP sources
- ○ Add existing block design sources
- ○ Add existing IP

To continue, click Next

Figure 2.22: Add Sources dialog box - Add or create design sources option

Step 10. Click **Next**

Step 11. In the **Add or Create Design Sources** dialog box, click the + icon and select **Add Files...** option to include the existing source files into the project, or just click **Add Files** button, see Illustration 2.23


Add Sources

Add or Create Design Sources

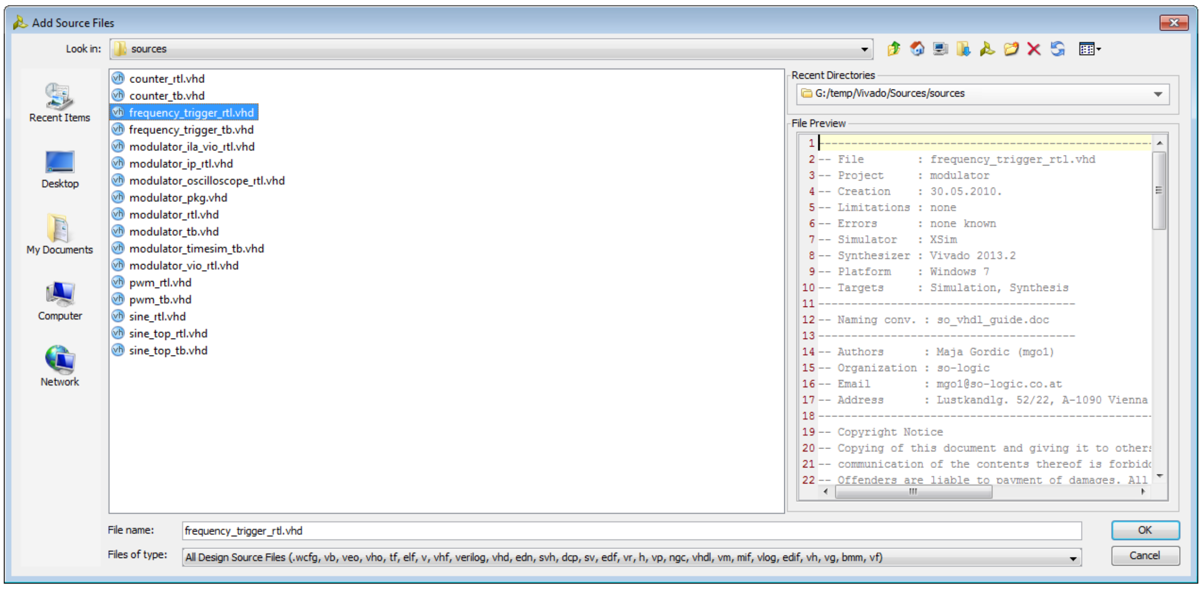
Specify HDL and netlist files, or directories containing HDL and netlist files, to add to your project. Create a new source file on disk and add it to your project.

Use Add Files, Add Directories or Create File buttons below

- ☐ Scan and add RTL include files into project
- ☐ Copy sources into project
- ☑ Add sources from subdirectories

Figure 2.23: Add or Create Design Sources dialog box - Add Files option

Step 12. In the **Add Source Files** dialog box, browse to the project working directory and select the ***frequency_trigger_-rtl.vhd*** source file, see Illustration 2.24



counter_rtl.vhd
counter_tb.vhd
frequency_trigger_rtl.vhd
frequency_trigger_tb.vhd
modulator_ila_vio_rtl.vhd
modulator_ip_rtl.vhd
modulator_oscilloscope_rtl.vhd
modulator_pkg.vhd
modulator_rtl.vhd
modulator_tb.vhd
modulator_timesim_tb.vhd
modulator_vio_rtl.vhd
pwm_rtl.vhd
pwm_tb.vhd
sine_rtl.vhd
sine_top_rtl.vhd
sine_top_tb.vhd

Recent Directories: G:/temp/Vivado/Sources/sources

File Preview

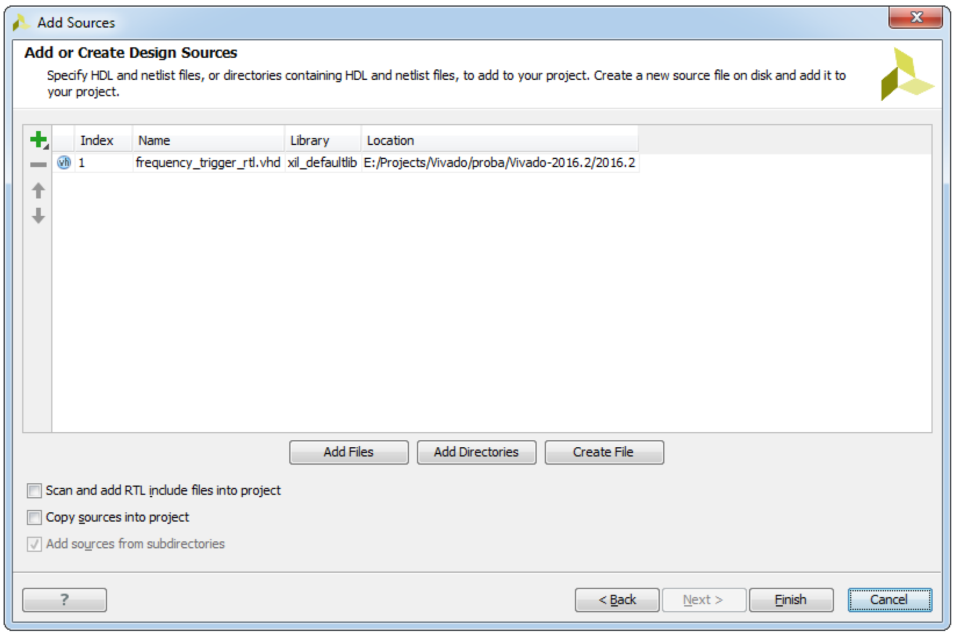
```
-------------------------------------------------------
-- File         : frequency_trigger_rtl.vhd
-- Project      : modulator
-- Creation     : 30.05.2010.
-- Limitations : none
-- Errors       : none known
-- Simulator    : XSim
-- Synthesizer : Vivado 2013.2
-- Platform     : Windows 7
-- Targets      : Simulation, Synthesis
---------------------------------------
-- Naming conv. : so_vhdl_guide.doc
---------------------------------------
-- Authors       : Maja Gordic (mgo1)
-- Organization : so-logic
-- Email         : mgo1@so-logic.co.at
-- Address       : Lustkandlg. 52/22, A-1090 Vienna
-------------------------------------------------------
-- Copyright Notice
-- Copying of this document and giving it to others
-- communication of the contents thereof is forbidd
-- Offenders are liable to payment of damages. All
```

File name: frequency_trigger_rtl.vhd

Files of type: All Design Source Files (.wcfg, vb, veo, vho, tf, elf, v, vhf, verilog, vhd, edn, svh, dcp, sv, edf, vr, h, vp, ngc, vhdl, vm, mif, vlog, edif, vh, vg, bmm, vf)

Figure 2.24: Add Source Files dialog box

Step 13. Click **OK** and the ***frequency_trigger_rtl.vhd*** source file should appear in the **Add or Create Design Sources** dialog box, as it is shown on the Illustration 2.25



Add or Create Design Sources

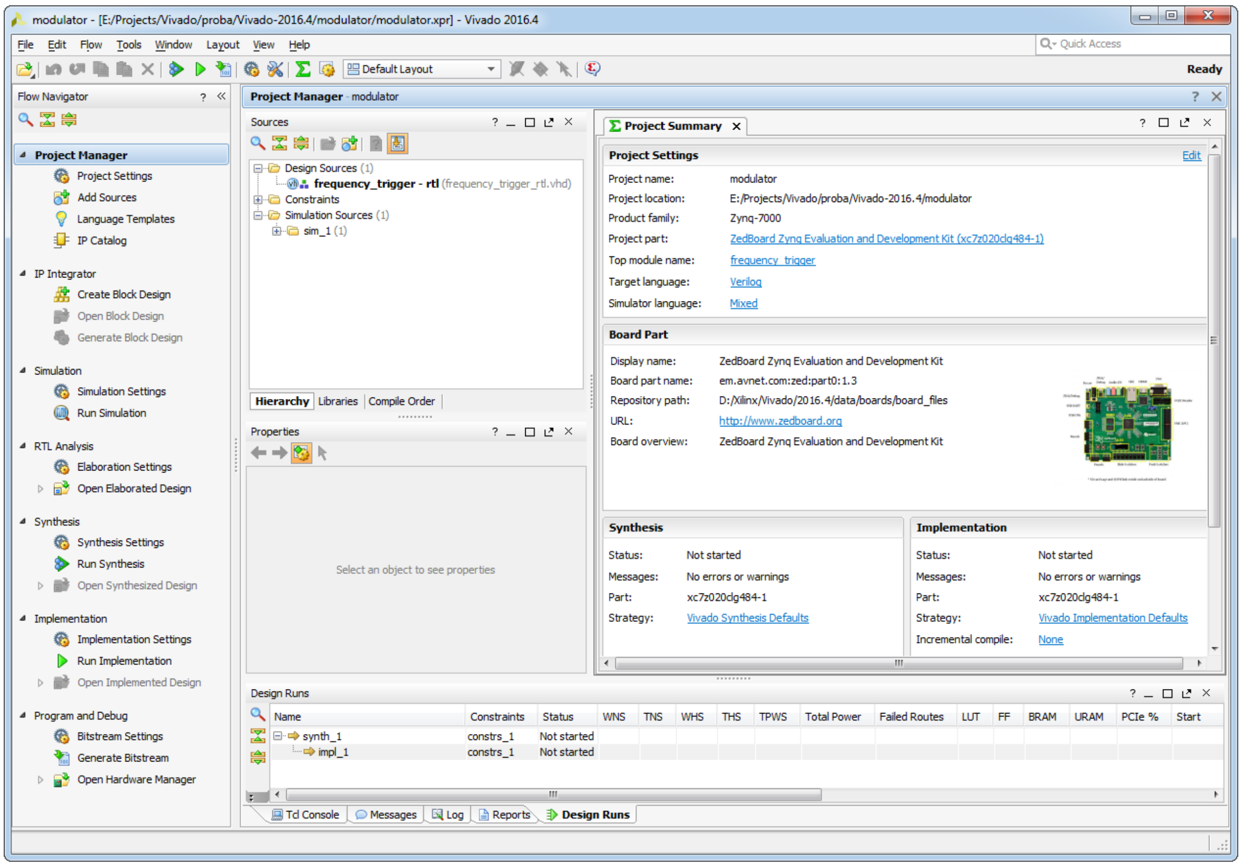
Specify HDL and netlist files, or directories containing HDL and netlist files, to add to your project. Create a new source file on disk and add it to your project.

Index	Name	Library	Location
1	frequency_trigger_rtl.vhd	xil_defaultlib	E:/Projects/Vivado/proba/Vivado-2016.2/2016.2

☐ Scan and add RTL include files into project
☐ Copy sources into project
☑ Add sources from subdirectories

Figure 2.25: Add or Create Design Sources dialog box - with added file

Step 14. Click **Finish** and your source file should appear under the **Design Sources** in the **Sources** view in the **Project Manager** window, see Illustration 2.26



modulator - [E:/Projects/Vivado/proba/Vivado-2016.4/modulator/modulator.xpr] - Vivado 2016.4

Project Manager - modulator

Sources
- Design Sources (1)
  - frequency_trigger - rtl (frequency_trigger_rtl.vhd)
- Constraints
- Simulation Sources (1)
  - sim_1 (1)

Project Summary

Project Settings

Project name:	modulator
Project location:	E:/Projects/Vivado/proba/Vivado-2016.4/modulator
Product family:	Zynq-7000
Project part:	ZedBoard Zynq Evaluation and Development Kit (xc7z020clg484-1)
Top module name:	frequency_trigger
Target language:	Verilog
Simulator language:	Mixed

Board Part

Display name:	ZedBoard Zynq Evaluation and Development Kit
Board part name:	em.avnet.com:zed:part0:1.3
Repository path:	D:/Xilinx/Vivado/2016.4/data/boards/board_files
URL:	http://www.zedboard.org
Board overview:	ZedBoard Zynq Evaluation and Development Kit

Synthesis

Status:	Not started
Messages:	No errors or warnings
Part:	xc7z020clg484-1
Strategy:	Vivado Synthesis Defaults

Implementation

Status:	Not started
Messages:	No errors or warnings
Part:	xc7z020clg484-1
Strategy:	Vivado Implementation Defaults
Incremental compile:	None

Design Runs

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAM	URAM	PCIe %	Start
synth_1	constrs_1	Not started													
impl_1	constrs_1	Not started													

Figure 2.26: Vivado IDE Viewing Environment with added source file

Note: Double-click on the **frequency_trigger - rtl (frequency_trigger_rtl.vhd)** source file in the **Sources** view and your source file should appear in the Vivado editor on the right side of the Vivado IDE. Using Vivado editor you can further modify this source file, if needed.

Frequency Trigger VHDL model:

```
-- 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;

-- 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 frequency_trigger is
    port(
        -- input clock signal
        clk_in              : in std_logic;
        -- signal made for selecting frequency
        sw0                 : in std_logic;
        -- input clock division factor when sw0 = '1'
        div_factor_freqhigh : in std_logic_vector(31 downto 0);
        -- input clock division factor when sw0 = '0'
        div_factor_freqlow  : in std_logic_vector(31 downto 0);
        -- output signal which frequency depends on the sw0 state
        freq_trig           : out std_logic
        );
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 frequency_trigger is

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

    signal freq_cnt_s : integer := 0; -- clock counter

begin

-- Defines a sequential process
-- Counts to different values depending on the sw0

    freq_ce_p : process
    begin
        -- replaces the sensitivity list
        -- suspends evaluation until an event occurs
        -- in our case event we are waiting for is rising edge on the clk_in input port
        wait until rising_edge(clk_in);
        freq_trig <= '0';              -- default assignment
        freq_cnt_s <= freq_cnt_s + 1; -- counting

        if (sw0 = '0') then
            if (freq_cnt_s >= div_factor_freqlow - 1) then
                freq_trig <= '1';
                freq_cnt_s <= 0; -- reset
            end if;
        else
            if (freq_cnt_s >= div_factor_freqhigh - 1) then
                freq_trig <= '1';
                freq_cnt_s <= 0; -- reset
            end if;
        end if;
    end process;
end;
```

2.5 Creating Test Bench

- ***Usage:*** used to verify correct operation of the frequency_trigger module defined in the frequency_trigger_rtl.vhd file
- ***Test bench internal signals:***
  - **clk_in_s**: input clock signal
  - **sw0_s**: input signal used to select output signal frequency
  - **freq_trig_s:** output signal which frequency depends of the sw0_s signal state
- ***Generics:***
  - **div_factor_freqhigh_g:** input clock division factor when sw0 = '1'
  - **div_factor_freqlow_g:** input clock division factor when sw0 = '0'
- ***File name:*** frequency_trigger_tb.vhd

We are creating a test bench to verify the correctness of a design or model.

To create and add an test bench file into the project, do the similar steps as for creating a module using Vivado text editor:

Step 1. **Optional:** Launch **Vivado IDE** (if it is not already launched)

Step 2. **Optional:** Open "Modulator" project (**modulator.xpr**) (if it is not already opened)

Step 3. In the main Vivado IDE menu, click **File -**> **New File...** option to open Vivado text editor

Step 4. In the **New File** dialog box, type the name of your test bench file (e.g. ***frequency_trigger_tb.vhd***) in the **File name** field and choose to save it into your working directory, on the same place where you saved *frequency_trigger_rtl.vhd* source file

Step 5. When you click **Save** , Vivado IDE will automatically open empty ***frequency_trigger_tb.vhd*** source file in Vivado text editor

Step 6. Insert the **VHDL code** and add the frequency_trigger_tb module header

Step 7. When you finish with the test bench creation, click **File -**> **Save File** option from the main Vivado IDE menu, or just click Ctrl + S to save it

Step 8. In the Vivado **Flow Navigator** click the **Add Sources** command, see Illustration 2.27

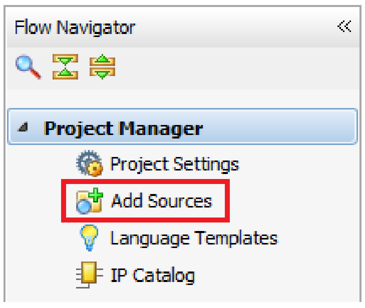
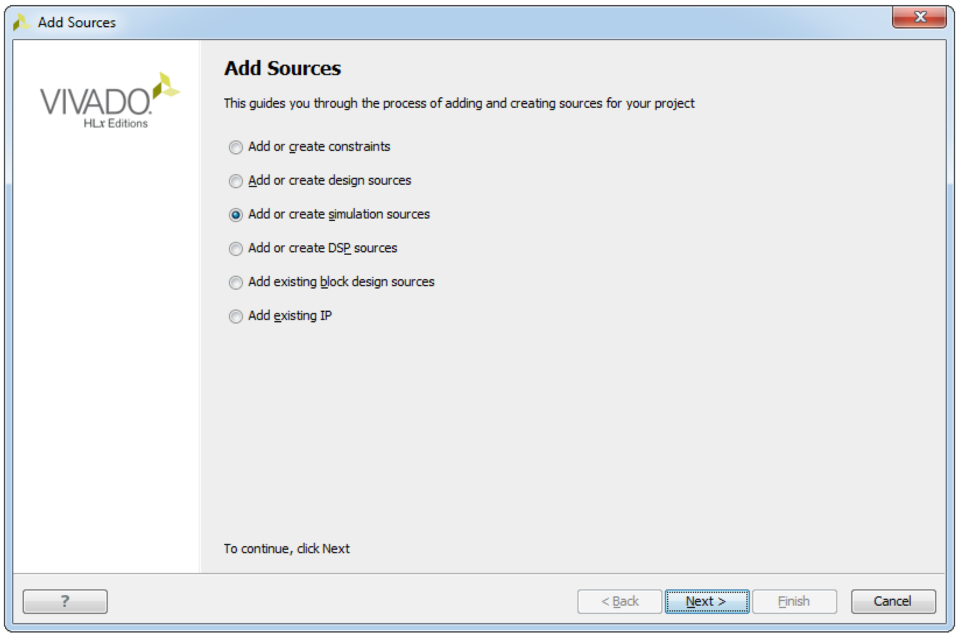


Figure 2.27: Add Sources command

Step 9. In the **Add Sources** dialog box, select **Add or create simulation sources** option to add the simulation source files into the project, see Illustration 2.28


Add Sources

Add Sources

This guides you through the process of adding and creating sources for your project

- ( ) Add or create constraints
- ( ) Add or create design sources
- (●) Add or create simulation sources
- ( ) Add or create DSP sources
- ( ) Add existing block design sources
- ( ) Add existing IP

To continue, click Next

Figure 2.28: Add Sources dialog box - Add or create simulation sources option

Step 10. Click **Next**

Step 11. In the **Add or Create Simulation Sources** dialog box, click the + icon and select **Add Files...** option, see Illustration 2.29

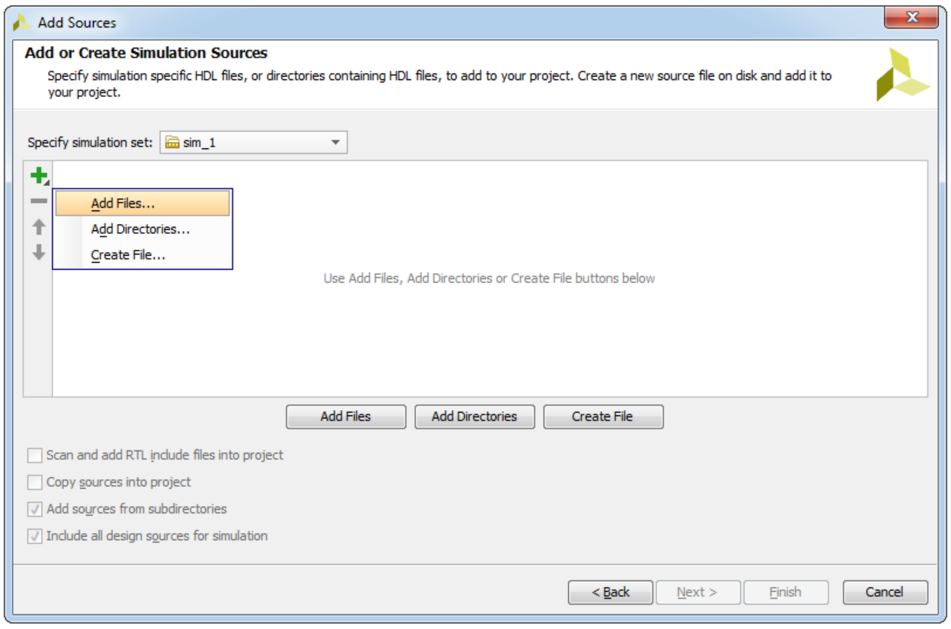
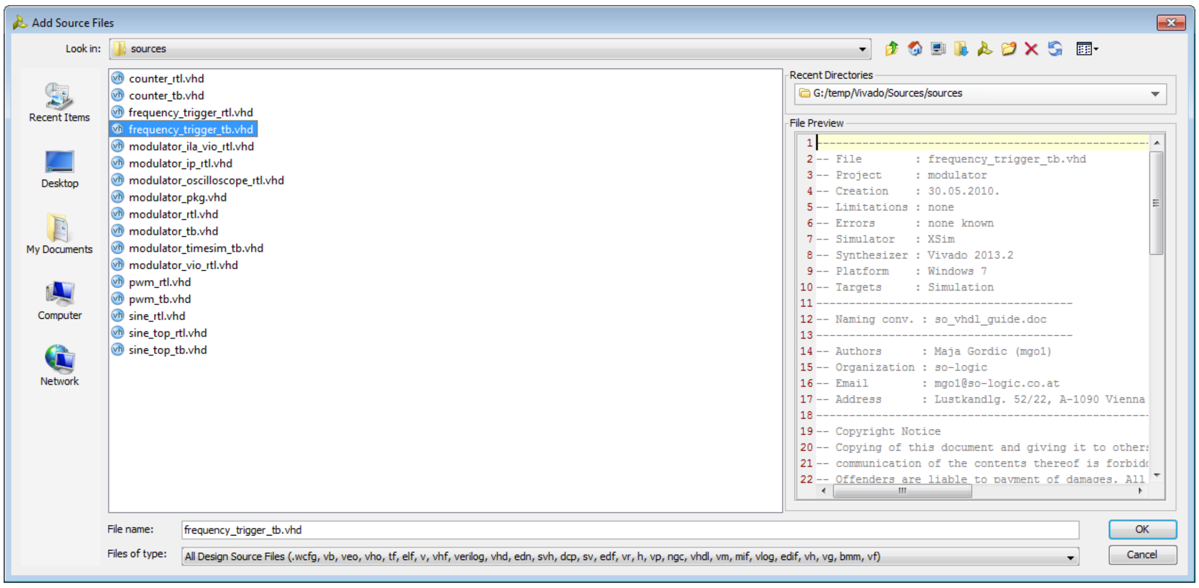

Add Sources
Add or Create Simulation Sources
Specify simulation specific HDL files, or directories containing HDL files, to add to your project. Create a new source file on disk and add it to your project.
Specify simulation set: sim_1
Add Files...
Add Directories...
Create File...
Use Add Files, Add Directories or Create File buttons below
Add Files
Add Directories
Create File
Scan and add RTL include files into project
Copy sources into project
Add sources from subdirectories
Include all design sources for simulation
< Back
Next >
Finish
Cancel

Figure 2.29: Add or Create Simulation Sources dialog box

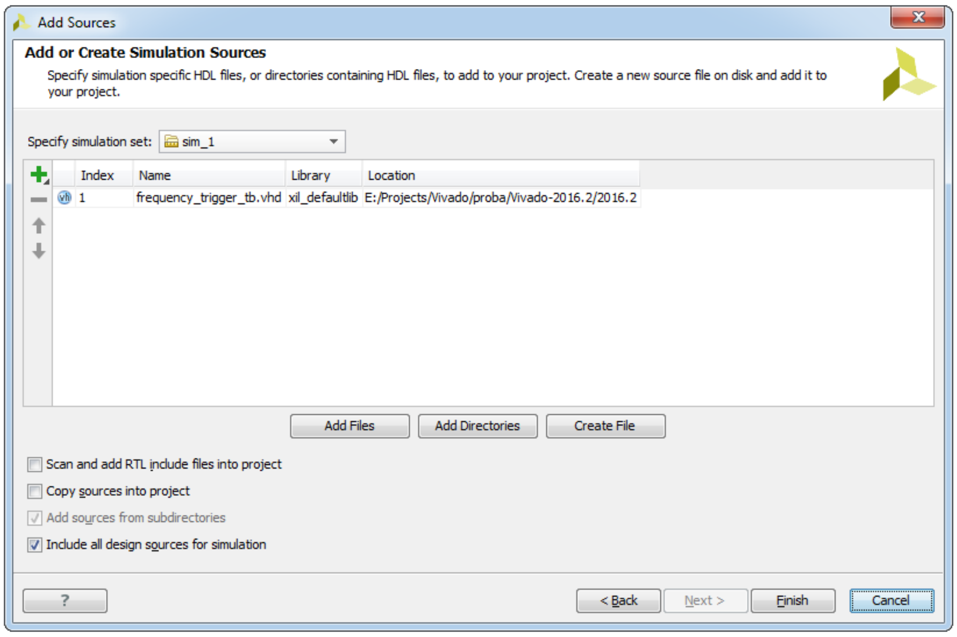
Step 12. In the **Add Source Files** dialog box, browse to the project working directory and select the ***frequency_trigger_-tb.vhd*** source file, see Illustration 2.30



```
----------------------------------------------------
-- File         : frequency_trigger_tb.vhd
-- Project      : modulator
-- Creation     : 30.05.2010.
-- Limitations  : none
-- Errors       : none known
-- Simulator    : XSim
-- Synthesizer  : Vivado 2013.2
-- Platform     : Windows 7
-- Targets      : Simulation
---------------------------------------
-- Naming conv. : so_vhdl_guide.doc
---------------------------------------
-- Authors       : Maja Gordic (mgo1)
-- Organization  : so-logic
-- Email         : mgo1@so-logic.co.at
-- Address       : Lustkandlg. 52/22, A-1090 Vienna
----------------------------------------------------
-- Copyright Notice
-- Copying of this document and giving it to others
-- communication of the contents thereof is forbidd
-- Offenders are liable to payment of damages. All
```

Figure 2.30: Add Source Files dialog box

Step 13. Click **OK** and the ***frequency_trigger_tb.vhd*** source file should appear in the **Add or Create Simulation Sources** dialog box, as it is shown on the Illustration 2.31



Add or Create Simulation Sources

Specify simulation specific HDL files, or directories containing HDL files, to add to your project. Create a new source file on disk and add it to your project.

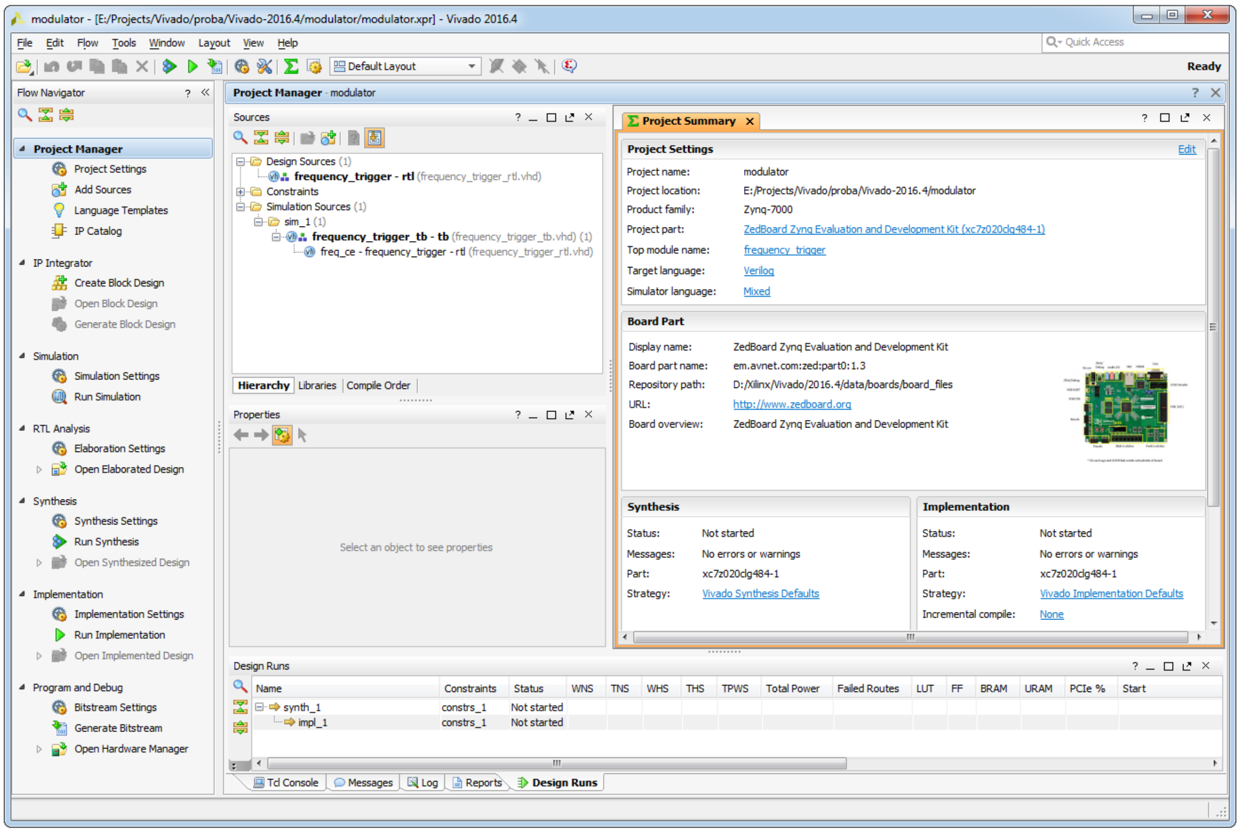
Specify simulation set: sim_1

Index	Name	Library	Location
1	frequency_trigger_tb.vhd	xil_defaultlib	E:/Projects/Vivado/proba/Vivado-2016.2/2016.2

- [ ] Scan and add RTL include files into project
- [ ] Copy sources into project
- [✓] Add sources from subdirectories
- [✓] Include all design sources for simulation

Figure 2.31: Add or Create Simulation Sources dialog box - with added file

Step 14. Click **Finish** and your test bench file should appear under the **Simulation Sources** / **sim_1** in the **Sources** view, in the **Project Manager** window, see Illustration 2.32



Sources:

```
Design Sources (1)
  frequency_trigger - rtl (frequency_trigger_rtl.vhd)
Constraints
Simulation Sources (1)
  sim_1 (1)
    frequency_trigger_tb - tb (frequency_trigger_tb.vhd) (1)
      freq_ce - frequency_trigger - rtl (frequency_trigger_rtl.vhd)
```

Project Settings

Project name:	modulator
Project location:	E:/Projects/Vivado/proba/Vivado-2016.4/modulator
Product family:	Zynq-7000
Project part:	ZedBoard Zynq Evaluation and Development Kit (xc7z020clg484-1)
Top module name:	frequency_trigger
Target language:	Verilog
Simulator language:	Mixed

Board Part

Display name:	ZedBoard Zynq Evaluation and Development Kit
Board part name:	em.avnet.com:zed:part0:1.3
Repository path:	D:/Xilinx/Vivado/2016.4/data/boards/board_files
URL:	http://www.zedboard.org
Board overview:	ZedBoard Zynq Evaluation and Development Kit

Synthesis

Status:	Not started
Messages:	No errors or warnings
Part:	xc7z020clg484-1
Strategy:	Vivado Synthesis Defaults

Implementation

Status:	Not started
Messages:	No errors or warnings
Part:	xc7z020clg484-1
Strategy:	Vivado Implementation Defaults
Incremental compile:	None

Design Runs

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAM	URAM	PCIe %	Start
synth_1	constrs_1	Not started													
impl_1	constrs_1	Not started													

Figure 2.32: Vivado IDE Viewing Environment with added test bench file

Note: Double-click on the **frequency_trigger_tb - tb (frequency_trigger_tb.vhd)** source file in the **Sources** view and your test bench file should appear in the Vivado text editor on the right side of the Vivado IDE.

Frequency Trigger test bench:

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

    -- include user defined modulator_pkg package where are important related
    -- declarations that serve a common purpose
    use work.modulator_pkg.all;

entity frequency_trigger_tb is

-- use lower values for generics to speed up simulation time
generic(
    div_factor_freqhigh_g : integer := 2; -- input clock division factor when sw0 = '1' (an example)
    div_factor_freqlow_g  : integer := 4  -- input clock division factor when sw0 = '0' (an example)
    );
end entity;

architecture tb of frequency_trigger_tb is

    signal clk_in_s          : std_logic := '1'; -- input clock signal
    signal freq_trig_s       : std_logic := '1'; -- signal which frequency depends on the sw0 state
    signal sw0_s             : std_logic := '0'; -- signal for selecting frequency

begin

    -- instantiation of device under test (DUT)
    -- no component definition is necessary
    -- use keyword entity, work is the library

    freq_ce : entity work.frequency_trigger (rtl)
        port map(
            clk_in              => clk_in_s,
            sw0                 => sw0_s,
            div_factor_freqhigh => conv_std_logic_vector(div_factor_freqhigh_g, 32),
            div_factor_freqlow  => conv_std_logic_vector(div_factor_freqlow_g, 32),
            freq_trig           => freq_trig_s
            );

    clk_in_s <= not (clk_in_s) after per_c/2; -- generates 50 MHz input clock signal;
    sw0_s <= '1' after 200 ns;
end;
```

Note: As you can see from the code above, you must include ***modulator_pkg.vhd*** source file into your **modulator** project. In the *modulator_pkg.vhd* file is defined per_c constant that will be used in this test bench. This package will be explained in detail later, in **Chapter 4. SINE PACKAGE**, where you can also find the whole modulator_pkg.vhd source code.

To include ***modulator_pkg.vhd*** source file into your **modulator** project, use **Add Sources** option from the Flow Navigator and repeat steps from the **Sub-chapter 2.4.1. Creating a Module Using Vivado Text Editor** for adding design sources.

2.6 Simulating with Vivado Simulator

Simulation is a process of emulating the real design behavior in a software environment. Simulation helps verify the functionality of a design by injecting stimulus and observing the design outputs. Simulators interpret HDL code into circuit functionality and display logical results.

The Vivado IDE is integrated with the Xilinx Vivado logic simulation environment. The Vivado IDE enables you to add and mange simulation test benches in the project. You can configure simulation options and create and manage various simulation source sets. You can launch behavioral simulation prior to synthesis using RTL sources and launch timing simulation using post-implementation simulation model, that will be generated by the Vivado IDE tool after completing the design implementation process.

After you have entered the code for the input stimulus in order to perform simulation, follow the next steps:

Step 1. In the **Sources** window, under the **Simulation Sources** / **sim_1** , select **frequency_trigger_tb - tb** file

Step 2. In the **Flow Navigator**, under the **Simulation**, click on the **Run Simulation** button

Step 3. Choose the only offered **Run Behavioral Simulation** option, see Illustration 2.33, and your simulation will start

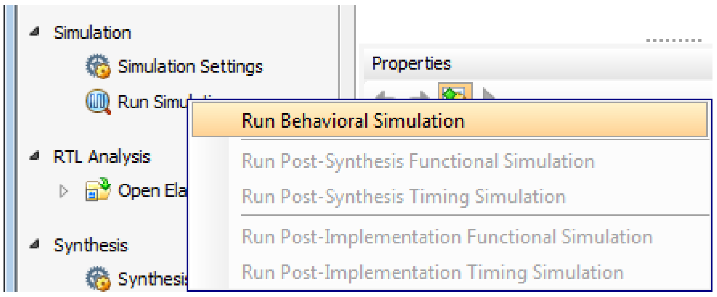
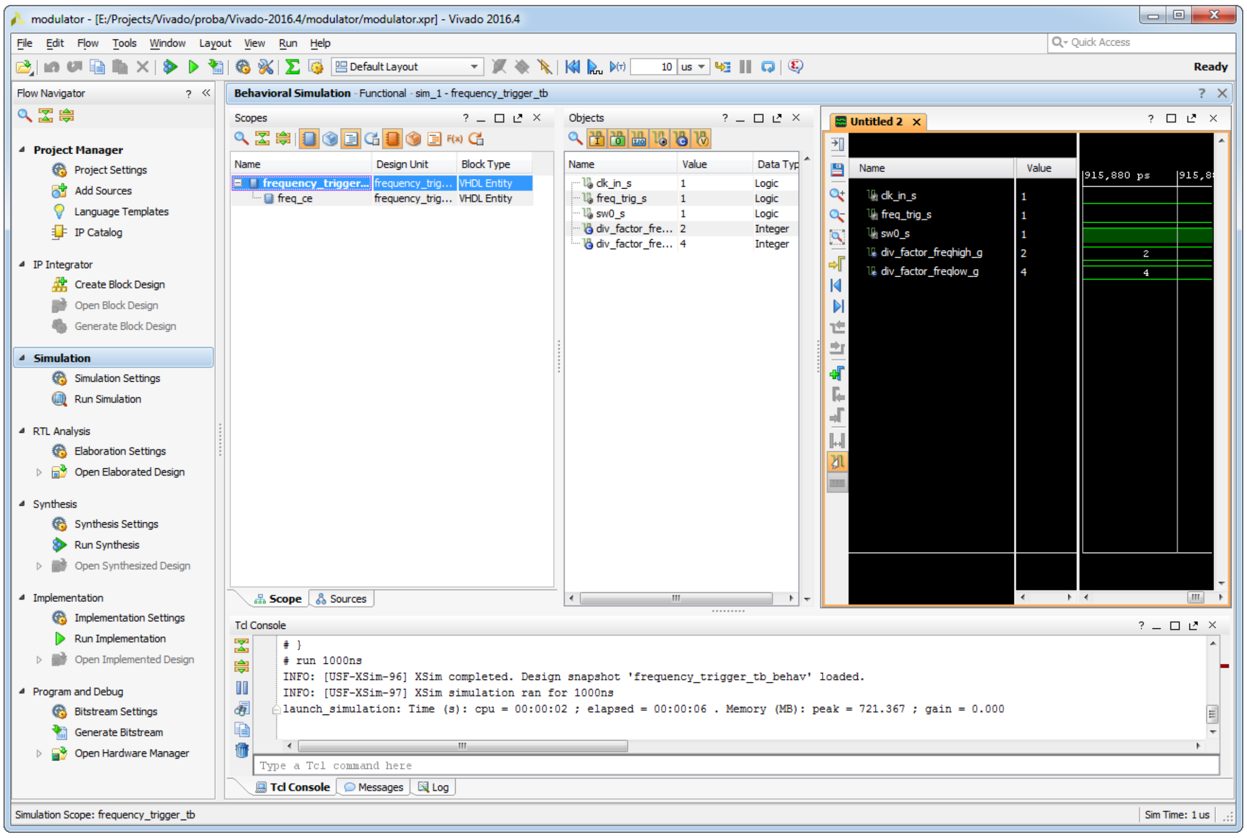

Simulation
Simulation Settings
Run Simu
RTL Analysis
Open Ela
Synthesis
Synthesis
Properties
Run Behavioral Simulation
Run Post-Synthesis Functional Simulation
Run Post-Synthesis Timing Simulation
Run Post-Implementation Functional Simulation
Run Post-Implementation Timing Simulation

Figure 2.33: Run Behavioral Simulation option

Step 4. The tool will compile the test bench file and launch the Vivado simulator, see Illustration 2.34


modulator - [E:/Projects/Vivado/proba/Vivado-2016.4/modulator/modulator.xpr] - Vivado 2016.4
Flow Navigator
Behavioral Simulation - Functional - sim_1 - frequency_trigger_tb
Scopes
Objects
Untitled 2
Tcl Console
Simulation Scope: frequency_trigger_tb
Sim Time: 1 us

```
# }
# run 1000ns
INFO: [USF-XSim-96] XSim completed. Design snapshot 'frequency_trigger_tb_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:02 ; elapsed = 00:00:06 . Memory (MB): peak = 721.367 ; gain = 0.000
```

Figure 2.34: Vivado IDE Viewing Environment - after simulation process

Note: By default, Untitled Waveform viewer will appear displaying only the signals at the top level of the test bench.

Step 5. Correct any errors before proceeding

Step 6. Double-click on the **Untitled 1** file or click on the Maximize button in the right upper corner of the waveform viewer

Step 7. Assuming no errors, your simulation result should look similar to the **Illustration 2.35**.

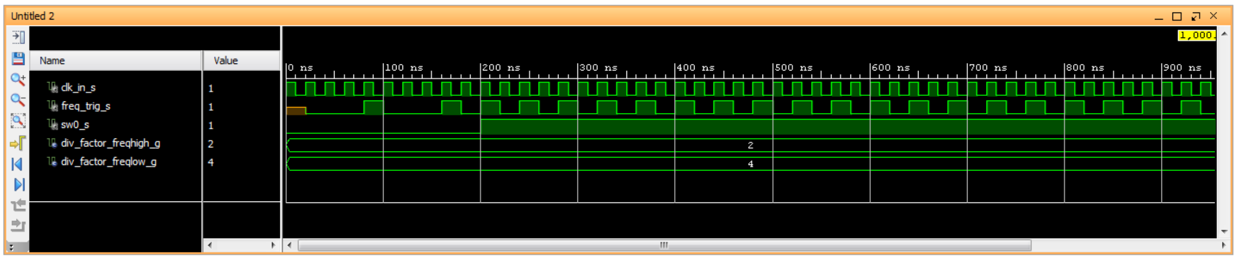


Figure 2.35: Simulation Results

Step 8. **Optional:** If you want to insert further internal signals from your simulated file, click on the desired file in the **Scopes** window and drag-and-drop the signals from the **Objects** window into the waveform window. Now you have to restart and rerun your simulation.

Step 9. **Optional:** If you want to restart and rerun simulation for specific time, see Illustration 2.36.

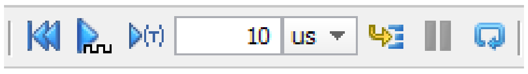


Figure 2.36: Vivado Simulator Simulation Controls

Vivado Simulator Simulation Controls has the following buttons that the user can use to control the simulation process:

- **Restart** - restarts the simulation from "time 0"
- **Run All** - run the simulation until there are no more events
- **Run for specified time** - runs the simulation for the specified amount of time
- **Step** - runs the simulation until the next breakable line
- **Break** - stops the running simulation at the next breakable line
- **Relaunch** - relaunch current Vivado simulator

Note: Information about creating a Frequency Trigger test bench file and simulating a design using Vivado simulator, you can also find in the **Lab 4:"Frequency Trigger Verification"**.

Chapter 3

COUNTER

3.1 Description

- ***Usage:*** This module will be an universal (generic) counter. It's task will be to generate read addresses for the ROM where samples of the sine wave are stored. The speed of the counting will be controlled by the Frequency Trigger module, via freg_trig port, and the output of the Counter module will be an input of the Digital Sine module.
- ***Block diagram:***

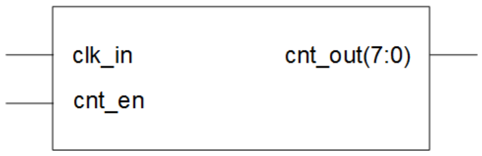


Figure 3.1: Counter block diagram

- ***Input ports:***
  - **clk_in:** input clock signal
  - **cnt_en:** counter enable
- ***Output ports:***
  - **cnt_out** : current counter value
- ***Generics:***
  - **cnt_value_g** : threshold value for counter
  - **depth_g** : the number of samples in one period of the signal
- ***File name:*** counter_rtl.vhd

3.2 Creating Module

As we already said, for better designs, our recommendation is not to use the GUI for module creation. Instead of that, create a module in Vivado text editor, name it to *module_name.vhd* and add it into your project.

All the steps for creating a new module using Vivado text editor or adding existing module are explained in **Sub-chapter 2.4.1 Creating a Module Using Vivado Text Editor**.

Counter VHDL model:

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

entity counter is
    generic(
        cnt_value_g : integer; -- threshold value for counter
        depth_g     : integer  -- the number of samples in one period of the signal
        );

    port(
        clk_in  : in std_logic;                               -- input clock signal
        cnt_en  : in std_logic;                               -- counter enable
        cnt_out : out std_logic_vector (depth_g - 1 downto 0) -- current counter value
        );
end entity;

architecture rtl of counter is

    signal cnt_out_s : std_logic_vector (depth_g - 1 downto 0) := (others => '0'); -- current counter value

begin

-- Defines a sequential process
-- This will be universal (generic) counter

     counter_p: process
     begin
        wait until rising_edge(clk_in);
            if (cnt_en = '1') then
                -- conv_std_logic_vector function converts integer type to std_logic_vector type
                if (cnt_out_s = conv_std_logic_vector (cnt_value_g, depth_g)) then
                    cnt_out_s <= (others => '0'); -- counter reset
                else
                    cnt_out_s <= cnt_out_s + 1;   -- counter
                end if;
            end if;
    end process;

    cnt_out <= cnt_out_s;

end;
```

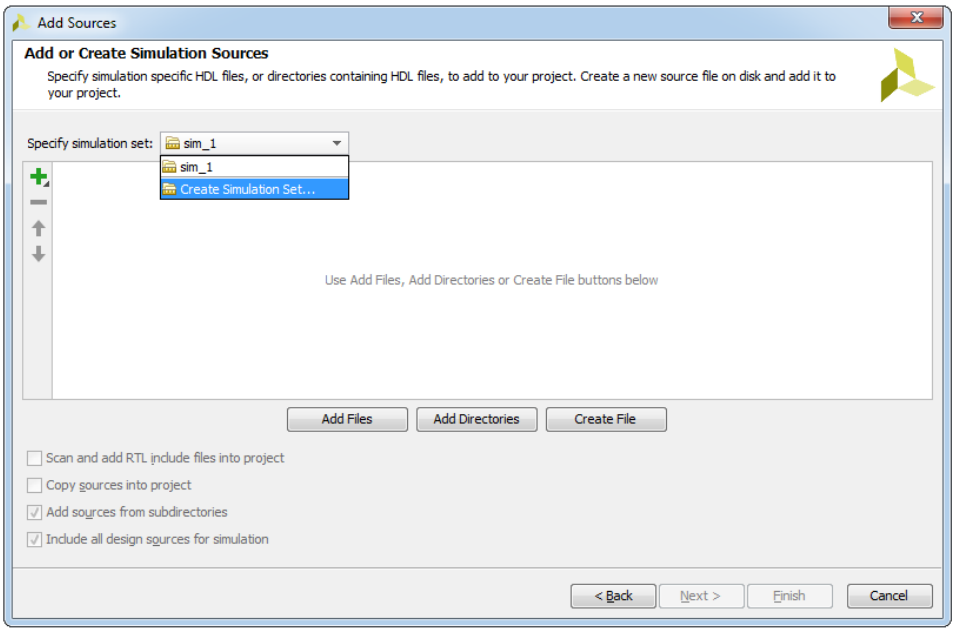
3.3 Creating Test Bench

- ***Usage:*** used to verify correct operation of the counter module defined in the counter_rtl.vhd file
- ***Test bench internal signals:***
  - **clk_in_s:** input clock signal
  - **cnt_en_s:** counter enable
  - **cnt_out_s:** current counter value
- ***Generics:***
  - **cnt_value_g:** threshold value for counter
  - **depth_g**: the number of samples in one period of the signal
- ***File name:*** counter_tb.vhd

We will now create a new simulation set (**sim_2**) with the test bench file for the Counter module **(counter_tb.vhd)** in it. We will use the similar steps as for creating test bench file for the Frequency Trigger module, explained in **Chapter 2.5 Creating Test Bench**:

Step 1. Repeat steps **1 - 10** from the **Chapter 2.5 Creating Test Bench**

Step 2. In the **Add or Create Simulation Sources** dialog box, click on the **Specify simulation set** drop-down list and choose **Create Simulation Set...** option, see Illustration 3.2


Add Sources

Add or Create Simulation Sources

Specify simulation specific HDL files, or directories containing HDL files, to add to your project. Create a new source file on disk and add it to your project.

Specify simulation set: sim_1

sim_1
Create Simulation Set...

Use Add Files, Add Directories or Create File buttons below

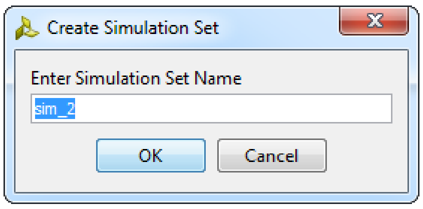
Add Files | Add Directories | Create File

☐ Scan and add RTL include files into project
☐ Copy sources into project
☑ Add sources from subdirectories
☑ Include all design sources for simulation

< Back | Next > | Finish | Cancel

Figure 3.2: Create Simulation Set option

Step 3. In the **Create Simulation Set** dialog box, enter a name for the new simulation set or leave sim_2 as a name and click **OK**, see Illustration 3.3


Create Simulation Set

Enter Simulation Set Name

sim_2

OK | Cancel

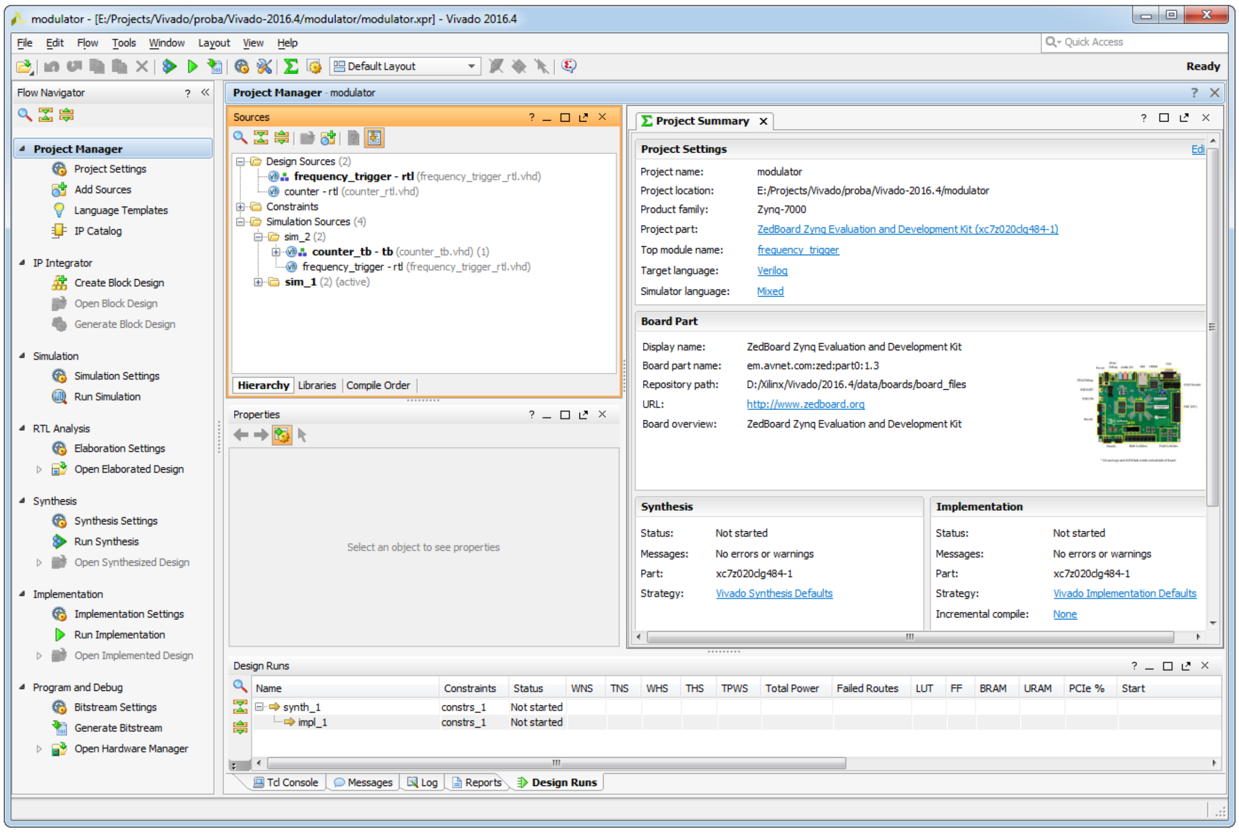
Figure 3.3: Create Simulation Set dialog box

Step 4. In the **Add or Create Simulation Sources** dialog box, under the new **sim_2** simulation set, use **Add Files...** option to add the test bench file for the Counter module

Step 5. In the **Add Source Files** dialog box, browse to the project working directory and select the ***counter_tb.vhd*** test bench file

Step 6. Click **OK** and ***counter_tb.vhd*** source file should appear in the **Add or Create Simulation Sources** dialog box

Step 7. Click **Finish** and your test bench file should appear under the **Simulation Sources** / **sim_2** in the **Sources** view, in the **Project Manager** window, see Illustration 3.4



Design Sources (2)
frequency_trigger - rtl (frequency_trigger_rtl.vhd)
counter - rtl (counter_rtl.vhd)
Constraints
Simulation Sources (4)
sim_2 (2)
counter_tb - tb (counter_tb.vhd) (1)
frequency_trigger - rtl (frequency_trigger_rtl.vhd)
sim_1 (2) (active)

Project Settings

Project name:	modulator
Project location:	E:/Projects/Vivado/proba/Vivado-2016.4/modulator
Product family:	Zynq-7000
Project part:	ZedBoard Zynq Evaluation and Development Kit (xc7z020clg484-1)
Top module name:	frequency_trigger
Target language:	Verilog
Simulator language:	Mixed

Board Part

Display name:	ZedBoard Zynq Evaluation and Development Kit
Board part name:	em.avnet.com:zed:part0:1.3
Repository path:	D:/Xilinx/Vivado/2016.4/data/boards/board_files
URL:	http://www.zedboard.org
Board overview:	ZedBoard Zynq Evaluation and Development Kit

Synthesis

Status:	Not started
Messages:	No errors or warnings
Part:	xc7z020clg484-1
Strategy:	Vivado Synthesis Defaults

Implementation

Status:	Not started
Messages:	No errors or warnings
Part:	xc7z020clg484-1
Strategy:	Vivado Implementation Defaults
Incremental compile:	None

Design Runs

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAM	URAM	PCIe %	Start
synth_1	constrs_1	Not started													
impl_1	constrs_1	Not started													

Figure 3.4: Vivado IDE Viewing Environment with created new simulation set

Counter test bench:

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

    use work.modulator_pkg.all;

entity counter_tb is

    -- Use lower values for generics to speed up simulation time
    generic(
        cnt_value_g : integer := 4; -- threshold value for counter
        depth_g     : integer := 3  -- the number of samples in one period of the signal
    );

end entity;

architecture tb of counter_tb is

    signal clk_in_s  : std_logic := ’1’;                                        -- input clock signal
    signal cnt_en_s  : std_logic := ’0’;                                        -- counter enable
    signal cnt_out_s : std_logic_vector (depth_g - 1 downto 0) := (others => ’0’); -- current counter value

begin

    counter : entity work.counter(rtl) -- counter instance
        generic map(
            cnt_value_g => cnt_value_g,
            depth_g     => depth_g
            )

        port map (
            clk_in  => clk_in_s,
            cnt_en  => cnt_en_s,
            cnt_out => cnt_out_s
            );

    clk_in_s <= not (clk_in_s) after per_c/2; -- generates 50 MHz input clock signal
    cnt_en_s <= ’1’ after 100 ns, ’0’ after 120 ns, ’1’ after 160 ns, ’0’ after 180 ns, ’1’ after 220 ns,
                ’0’ after 240 ns, ’1’ after 320 ns, ’0’ after 340 ns, ’1’ after 420 ns, ’0’ after 440 ns;

end;
```

3.4 Simulating

After you have entered the code for the input stimulus in order to perform simulation, follow the next steps:

Step 1. In the **Sources** window, under the **Simulation Sources**, select new **sim_2** simulation set, right-click on it and choose **Make Active** option, see Illustration 3.5

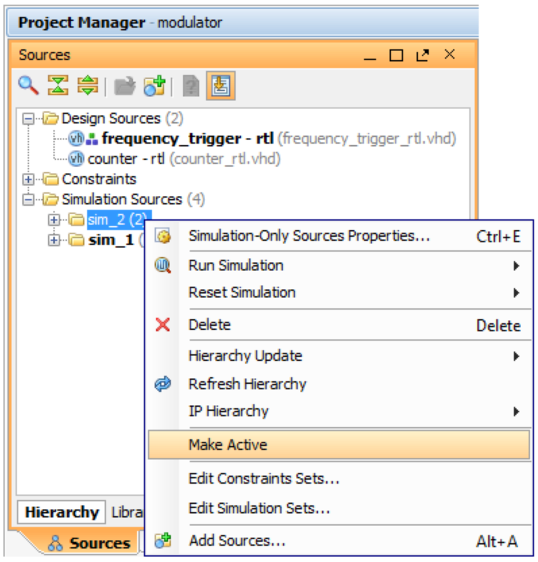


Figure 3.5: Make Active option

Step 2. In the **Flow Navigator**, under the **Simulation**, click **Run Simulation** command

Step 3. Choose the only offered **Run Behavioral Simulation** option and your simulation will start

Step 4. The tool will compile the test bench and launch the Vivado simulator

Step 5. Correct any errors before proceeding

Step 6. Double-click on the **Untitled 1** file or click on the Maximize button in the right upper corner of the waveform viewer

Step 7. Assuming no errors in the Vivado simulator command line, your simulation result should look similar to **Illustration 3.6**

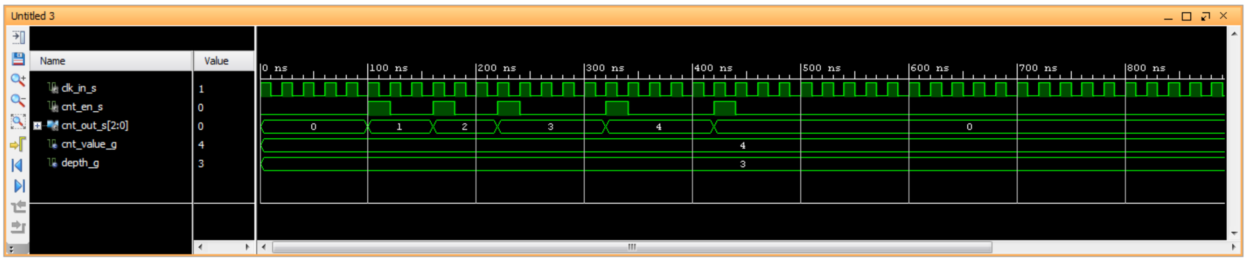


Figure 3.6: Simulation Results

Note: All the information about creating the Counter module, generating its test bench file and simulating the Counter design, you can also find in the **Lab 5: "Creating Counter Module"**.

Chapter 4

SINE PACKAGE

4.1 Description

- ***Usage:*** In our case we will make an VHDL package with a parametrized sine signal. Total of $2^8 = 256$ unsigned amplitude values during one sine-period will be stored into an ROM array.

  In order to simplify the generation of the PWM signal, we will use the sine wave signal that is shifted upwards. The value of this shift will be selected in a way to make all values of the sine signal positive. This is illustrated on the Illustration 4.1.

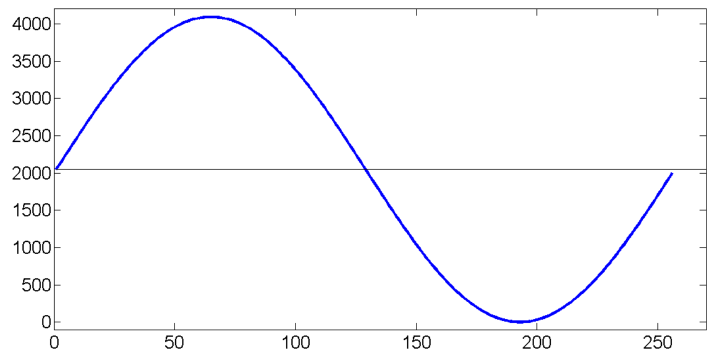


Figure 4.1: Sine-package description

The formula for calculating the sine wave shown on the Illustration 4.1 is:

$$sin(\frac{2\pi * i}{N}) * (2^{width_c - 1} - 1) + 2^{width_c - 1} - 1, N = 2^{depth_c}$$

depth_c - is the number of samples in one period of the signal ($2^8 = 256$)

width_c - is the number of bits used to represent amplitude value ($2^{12} = 4096$)

This formula is defining the nature of the desired sine signal:

- $sin(\frac{2\pi * i}{N})$ - is telling us that the signal is periodic, with 2π period; *i* - is the current sample value (from 0 to 255) and N is the number of samples in one period of the signal
- $*(2^{width_c - 1} - 1)$ - is telling us that the amplitude of the sine signal is 2047
- $+2^{width_c - 1} - 1$ - is telling us that the DC value of the sine signal is 2047, which means that the whole sine signal is shifted up
- ***File name:*** modulator_pkg.vhd

4.2 Creating Module

To create a Sine-package module, use steps for creating modules, **Sub-chapter 2.4.1 Creating a Module Using Vivado Text Editor**.

Sine package VHDL model:

```
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; -- number of samples in one period of the signal (2^8=256)
        constant width_c : in integer  -- number of bits used to represent amplitude value (2^12=4096)
        )
    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;
```

Note: All the information about creating the sine package, you can also find in the **Lab 6: "Creating Sine Package"**.

Chapter 5

DIGITAL SINE

5.1 Description

- ***Usage:*** This module will generate an digital representation of an analog (sine) signal with desired frequency. It will use the counter values as addresses to fetch the next value of the sine wave from the ROM.

  Note: Don't forget to include the Sine package in the code of the Digital Sine module!

- ***Block diagram***:

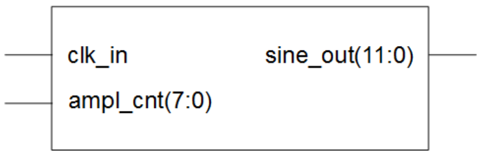


Figure 5.1: Digital Sine block diagram

- ***Input ports***:
  - **clk_in** : input clock signal
  - **ampl_cnt** : address value for the sine waveform ROM
- ***Output ports:***
  - **sine_out** : current amplitude value of the sine signal
- ***Generics:***
  - **depth_g** : the number of samples in one period of the signal
  - **width_g**: the number of bits used to represent amplitude value
- ***File name***: sine_rtl.vhd

5.2 Creating Module

To create Digital Sine module, use steps for creating modules, **Sub-chapter 2.4.1 Creating a Module Using Vivado Text Editor** .

Digital Sine VHDL model:

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

    use work.modulator_pkg.all;
```

```
entity sine is
    generic(
        depth_g : integer range 1 to 99 := 8; -- the number of samples in one period of the signal
        width_g : integer range 1 to 99 := 12 -- the number of bits used to represent amplitude value
        );

    port(
        clk_in   : in std_logic;                            -- input clock signal
        ampl_cnt : in std_logic_vector(depth_g-1 downto 0); -- address value for the sine waveform ROM
        sine_out : out std_logic_vector(width_g-1 downto 0) -- current amplitude value of the sine signal
    );
end entity;

architecture rtl of sine is

    constant sin_ampl_c : vector_t_arr := init_sin_f(depth_g, width_g);    -- returns sine amplitude value

    signal ampl_cnt_s : integer range 0 to 255 := 0;                       -- amplitude counter
    signal sine_s : std_logic_vector(width_g-1 downto 0) := (others=>'0'); -- sine signal

begin

-- Defines a sequential process
-- Fetches amplitude values and frequency -> generates sine

    sine_p : process
    begin

        wait until rising_edge(clk_in);
        -- converts ampl_cnt from std_logic_vector type to integer type
        ampl_cnt_s <= conv_integer(ampl_cnt);
        -- converts sin_ampl_c from integer type to std_logic_vector type
        sine_s <= conv_std_logic_vector(sin_ampl_c(ampl_cnt_s), width_g); -- fetch amplitude

    end process;

    sine_out <= sine_s;

end;
```

Note: All the information about creating the Digital Sine module, you can also find in the **Lab 7: "Creating Digital Sine Module"** .

Chapter 6

DIGITAL SINE TOP

6.1 Description

- ***Usage:*** This module will merge Frequency Trigger, Counter, Sine package and Digital Sine module into one Digital Sine Top module (Drawings 6.1 and 6.2). It will have four input ports: one will be used for input clock signal (clk_in), the second one will be used for changing output signal frequency (sw0) and the last two ports (div_factor_freqhigh and div_factor_freqlow) will be used for specifying input clock division factors. The only output port will represent the current amplitude value of the desired sine signal.

- ***Block diagram:***

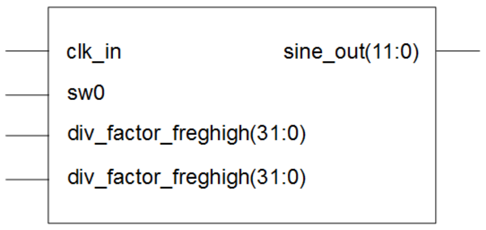


Figure 6.1: Digital Sine Top block diagram

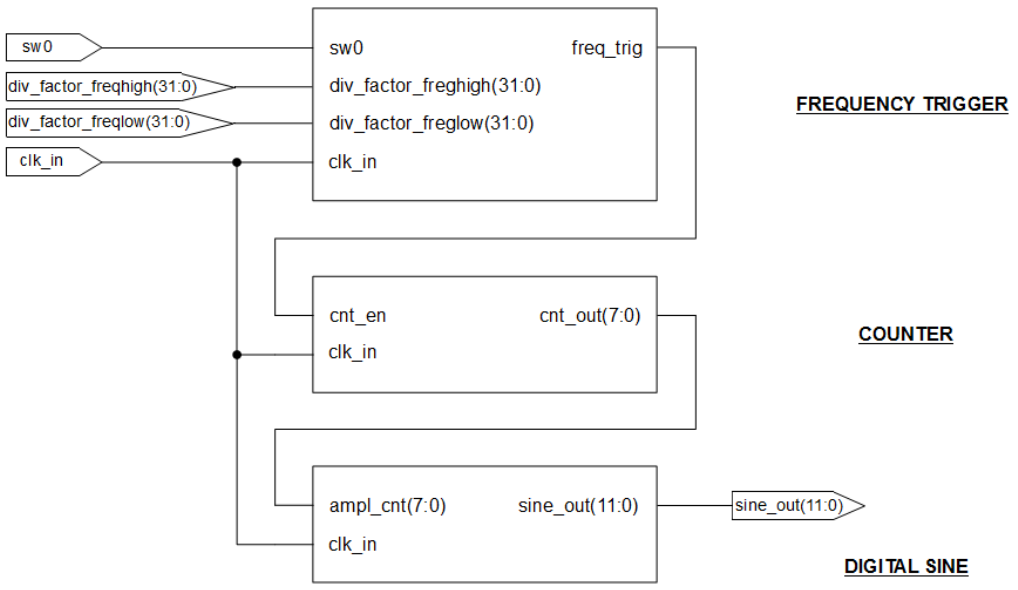


Figure 6.2: Digital Sine Top detailed block diagram

- ***Input ports:***
  - **clk_in**: input clock signal
  - **sw0**: input signal from the on-board switch, used for changing output signal frequency
  - **div_factor_freqhigh**: input clock division factor when sw0 = ’1’
  - **div_factor_freqlow** : input clock division factor when sw0 = ’0’
- ***Output ports:***
  - **sine_out**: current amplitude value of the sine signal
- ***Generics:***
  - **cntampl_value_g** : threshold value for counter, it’s value should be equal to (2^depth)-1
  - **depth_g**: the number of samples in one period of the signal
  - **width_g**: the number of bits used to represent amplitude value
- ***File name:*** sine_top_rtl.vhd

6.2 Creating Module

To create Digital Sine Top module, use steps for creating modules, **Sub-chapter 2.4.1 Creating a Module Using Vivado Text Editor** .

Digital Sine Top VHDL model:

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

    use work.modulator_pkg.all;

entity sine_top is
    generic(
        cntampl_value_g : integer := 255;             -- threshold value for counter,
                                                      -- it’s value should be equal to (2^depth)-1
        depth_g         : integer range 1 to 99 := 8; -- the number of samples in one period of the signal
        width_g         : integer range 1 to 99 := 12 -- the number of bits used to represent amplitude
                                                      -- value
        );

    port(
        clk_in              : in std_logic;                       -- input clock signal
        sw0                 : in std_logic;                       -- signal used for selecting frequency
        div_factor_freqhigh : in std_logic_vector(31 downto 0); -- threshold value for high frequency
        div_factor_freqlow  : in std_logic_vector(31 downto 0); -- threshold value for low frequency
        sine_out            : out std_logic_vector(width_g-1 downto 0) -- current amplitude value of the
                                                                       -- sine signal
         );
end entity;

architecture rtl of sine_top is

    signal ampl_cnt_s  : std_logic_vector(depth_g-1 downto 0) := (others=>’0’); -- amplitude counter
    signal freq_trig_s : std_logic := ’0’;

begin

    -- frequency trigger module instance
    freq_ce : entity work.frequency_trigger(rtl)
        port map(
            clk_in              => clk_in,              -- input clock signal
            sw0                 => sw0,                 -- signal used for selecting frequency
            div_factor_freqhigh => div_factor_freqhigh, -- input clock division factor when sw0 = ’1’
            div_factor_freqlow  => div_factor_freqlow,  -- input clock division factor when sw0 = ’0’
            freq_trig           => freq_trig_s  -- output signal which frequency depends of the sw0 state
            );

    -- counter module instance
    counterampl : entity work.counter(rtl)
        generic map(
            cnt_value_g => cntampl_value_g, -- threshold value for counter
            depth_g     => depth_g          -- the number of samples in one period of the signal
            )

        port map(
            clk_in  => clk_in,      -- input clock signal
            cnt_en  => freq_trig_s, -- counter enable
            cnt_out => ampl_cnt_s   -- current counter value
            );

    -- digital sine module instance
    sine : entity work.sine(rtl)
        generic map(
            depth_g => depth_g, -- the number of samples in one period of the signal
            width_g => width_g  -- the number of bits used to represent amplitude value
            )

        port map(
            clk_in   => clk_in,     -- input clock signal
            ampl_cnt => ampl_cnt_s, -- address value for the sine waveform ROM
            sine_out => sine_out    -- current amplitude value of the sine signal
            );

end;
```

6.3 Creating Test Bench

- ***Usage:*** used to verify correct operation of the sine_top module defined in the sine_top_rtl.vhd file
- ***Test bench internal signals:***
  - **clk_in_s**: input clock signal
  - **sw0_s**: input signal from the on-board switch, used for changing output signal frequency
  - **sine_out_s**: current amplitude value of the sine signal
- ***Generics:***
  - **cntampl_value_g**: threshold value for counter
  - **depth_g**: the number of samples in one period of the signal
  - **width_g**: the number of bits used to represent amplitude value
  - **div_factor_freqhigh_g**: threshold value for high frequency
  - **div_factor_freqlow_g**: threshold value for low frequency
- ***File name:*** sine_top_tb.vhd

We will now create a new simulation set (**sim_3**) with the test bench file for the Digital Sine Top module **(sine_top_tb.vhd)** in it. We will use the steps explained in the **Sub-chapter 3.3 Creating Test Bench**.

Digital Sine Top test bench:

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

    use work.modulator_pkg.all;

entity sine_top_tb is
    -- Use lower values for div_factor_freqhigh_g and div_factor_freqlow_g generics to speed up simulation
    -- time
    generic(
        cntampl_value_g : integer := 255;       -- threshold value for counter,
                                                -- it's value should be equal to (2^depth)-1
        depth_g : integer range 1 to 99 := 8;   -- the number of samples in one period of the signal
        width_g : integer range 1 to 99 := 12   -- the number of bits used to represent amplitude value
        div_factor_freqhigh_g : integer := 55;  -- threshold value for high frequency
        div_factor_freqlow_g  : integer := 195; -- threshold value for low frequency
        );
end entity;

architecture tb of sine_top_tb is

    signal clk_in_s   : std_logic := '0';      -- input clock signal
    signal sw0_s      : std_logic := '0';      -- signal used for selecting frequency
    signal sine_out_s : std_logic_vector(width_g-1 downto 0) := (others=>'0');
                                               -- current amplitude value of the sine signal

begin
    -- sine_top module instance
    dut : entity work.sine_top
```

```
        generic map(
            cntampl_value_g => cntampl_value_g,
            depth_g         => depth_g,
            width_g         => width_g
            )

        port map(
            clk_in              => clk_in_s,
            sw0                 => sw0_s,
            div_factor_freqhigh => conv_std_logic_vector(div_factor_freqhigh_g, 32),
            div_factor_freqlow  => conv_std_logic_vector(div_factor_freqlow_g, 32),
            sine_out            => sine_out_s
            );

    clk_in_s <= not (clk_in_s) after per_c/2; -- 50 MHz input clock signal
    sw0_s    <= '0', '1' after 1 ms;

end;
```

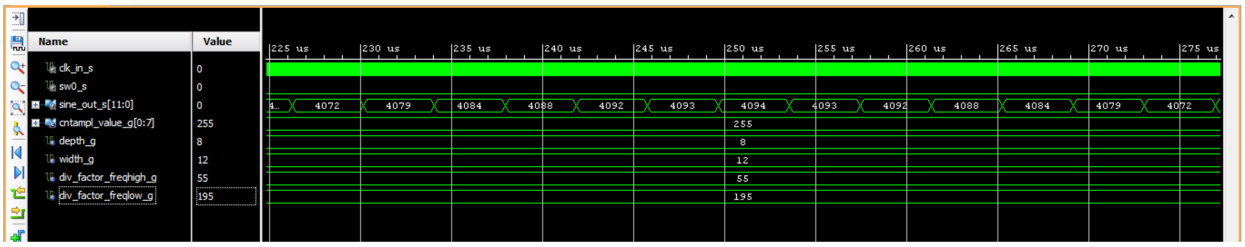
6.4 Simulating

After you have entered the code for the input stimulus in order to perform simulation:

Step 1. You can start your simulation (see **Chapter 3.4 Simulating**)

Step 2. Simulate your design for **4 ms** (see **Chapter 2.6 Simulating – step 9.**)

Step 3. Assuming no errors, your simulation result should look similar to **Illustration 6.3**.



Name	Value
clk_in_s	0
sw0_s	0
sine_out_s[11:0]	0
cntampl_value_g[0:7]	255
depth_g	8
width_g	12
div_factor_freqhigh_g	55
div_factor_freqlow_g	195

Figure 6.3: Simulation Results

As you can see from the illustration above, Vivado simulator presented sine signal, *sine_out_s,* in digital form. This is default Vivado simulator waveform style. If you would like to see if this signal really has a shape of sine signal, Vivado simulator gives you possibility to change the waveform style from digital to analog. To change the waveform style in Vivado simulator, please do the following:

1. Select the *sine_out_s* signal

2. Right-click on it and choose **Waveform Style ->** **Analog** , see Illustration 6.4

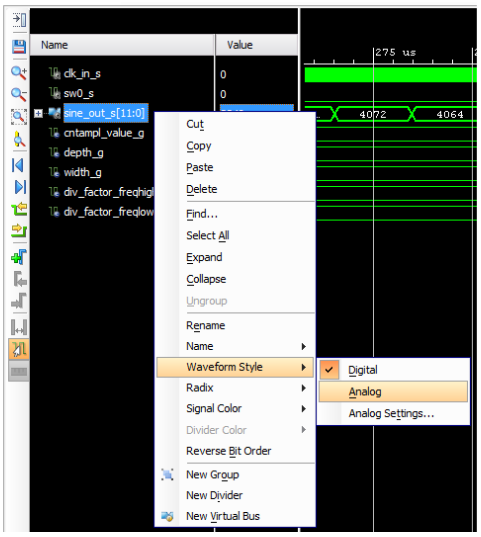


Figure 6.4: Waveform Style -> Analog option

When you change waveform style from digital to analog, Vivado simulator will automatically change sine signal perspective. Now, *sine_out_s* signal should have a shape of sine signal, as it is shown on the Illustration 6.5.

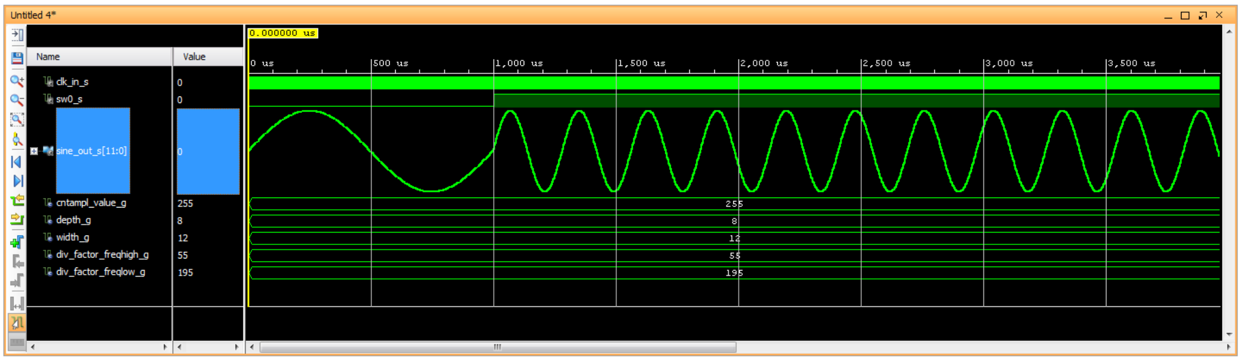


Figure 6.5: Simulation results with analog sine signal representation

In the Illustrations 6.3 and 6.5 and in the *sine_top_tb.vhd* source file you can also notice that we have changed ***div_factor-_freqhigh_g*** and ***div_facto_freqlow_g*** values from initial **196608** and **57344** values to **55** and **195** values, respectively. This is done, because we wanted to speed up the simulation process, in this example ∼1000 times, while retaining the same functionality. This is a way to speed up the simulation process without compromising functional behavioral of the system that is being simulated. This is the reason why we need only 4 ms to simulate our design, instead of 4000 ms which would take a 1000 times longer to complete.

Note: Information about creating the Digital Sine Top module, generating its test bench file and simulating the Digital Sine Top design, you can also find in the **Lab 8: "Creating Digital Sine Top Module"** .

6.5 Synthesis

6.5.1 Description

Synthesis is the process of transforming an RTL-specified design into a gate-level representation. It checks code syntax and analyse the hierarchy of your design. This ensures that your design is optimized for the design architecture that you have selected (e.g. Number of Flip-Flops, LUTs, Clock- and IO-Buffers).

Vivado IDE synthesis is timing-driven and optimized for memory usage and performance. Support for SystemVerilog as well as mixed VHDL and Verilog languages is included.

There are two ways to setup and run synthesis:

- **Use Project Mode** (which we will use in this tutorial)
- **Use Non-Project Mode** - applying the *synth_design* Tool Command Language (Tcl) command and controlling your own design files.

6.5.2 Run Synthesis

To synthesize your design, follow these steps:

Step 1. Before you run synthesis process, set **Digital Sine Top** module to be the top module. To do that, in the **Sources** window, under **Design Sources**, select *synthesizable module* (**sine_top - rtl**), right-click on it and choose **Set as Top** option

Step 2. In the Vivado **Flow Navigator**, click **Run Synthesis** command (**Synthesis** option) and wait for task to be completed, see Illustration 6.6

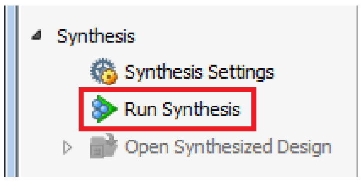
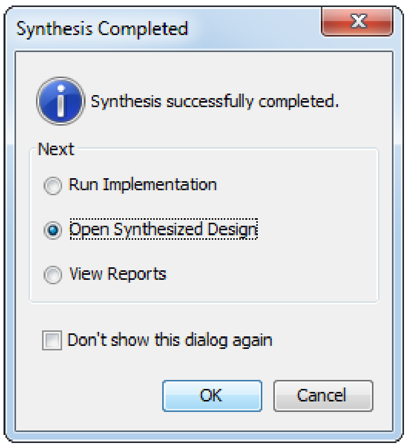


Figure 6.6: Run Synthesis command

Note: You can monitor the Synthesis progress in the bar in the upper-right corner of the Vivado IDE.

Step 3. After the synthesis is completed, the **Synthesis Completed** dialog box will appear, see Illustration 6.7


Synthesis Completed

Synthesis successfully completed.

Next

( ) Run Implementation

(•) Open Synthesized Design

( ) View Reports

☐ Don't show this dialog again

OK Cancel

Figure 6.7: Synthesis Completed dialog box

In the Synthesis Completed dialog box you can select one of the following options:

- ***Run Implementation***: which launches implementation with the current Implementation Project Settings.
- ***Open Synthesized Design***: which opens the synthesized netlist, the active constraint set, and the target device into Synthesized Design environment, so you can perform I/O pin planning, design analysis, and floorplanning.
- ***View Reports***: which opens the Reports window, so you can view reports.

Step 4. Select **Open Synthesized Design** and click **OK**, see Illustration 6.7

Step 5. Make sure that **Default Layout** option is selected from the view layout pull-down menu in the main toolbar, see Illustration 6.8

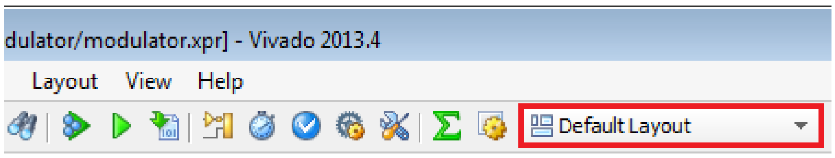


Figure 6.8: Default Layout option

6.5.3 After Synthesis

After you have synthesized your project (or opened a project that only contains netlists) the Flow Navigator changes and now includes: Constraints Wizard, Edit Timing Constraints, Set Up Debug, Report Timing Summary, Report Clock Networks, Report Clock Interaction, Report DRC, Report Noise, Report Utilization, Report Power and Schematic options, see Illustration 6.9

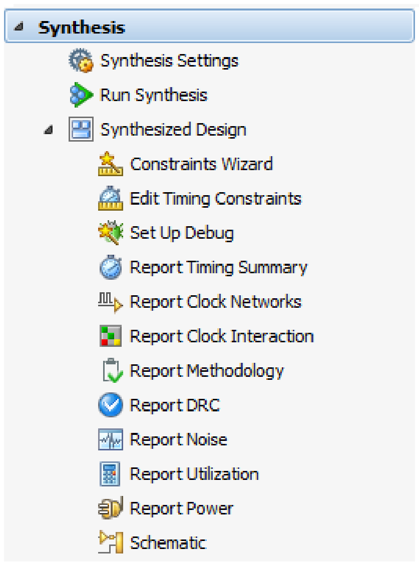


Figure 6.9: Synthesized Design options

Flow Navigator is optimized to provide quick access to the options most frequently used after synthesis:

- ***Constraints Wizard***: The Vivado IDE provides Timing Constraints wizard to walk you through the process of creating and validating timing constraints for the design. The wizard identifies clocks and logic constructs in the design and provides an interface to enter and validate the timing constraints in the design. It is only available in the synthesized and implemented designs.

- ***Edit Timing Constraints***: Open the Constraint Viewer (formerly called the Constraints Editor). The Timing Constraints window appears in the main window of the Vivado IDE, see Illustration 6.10.

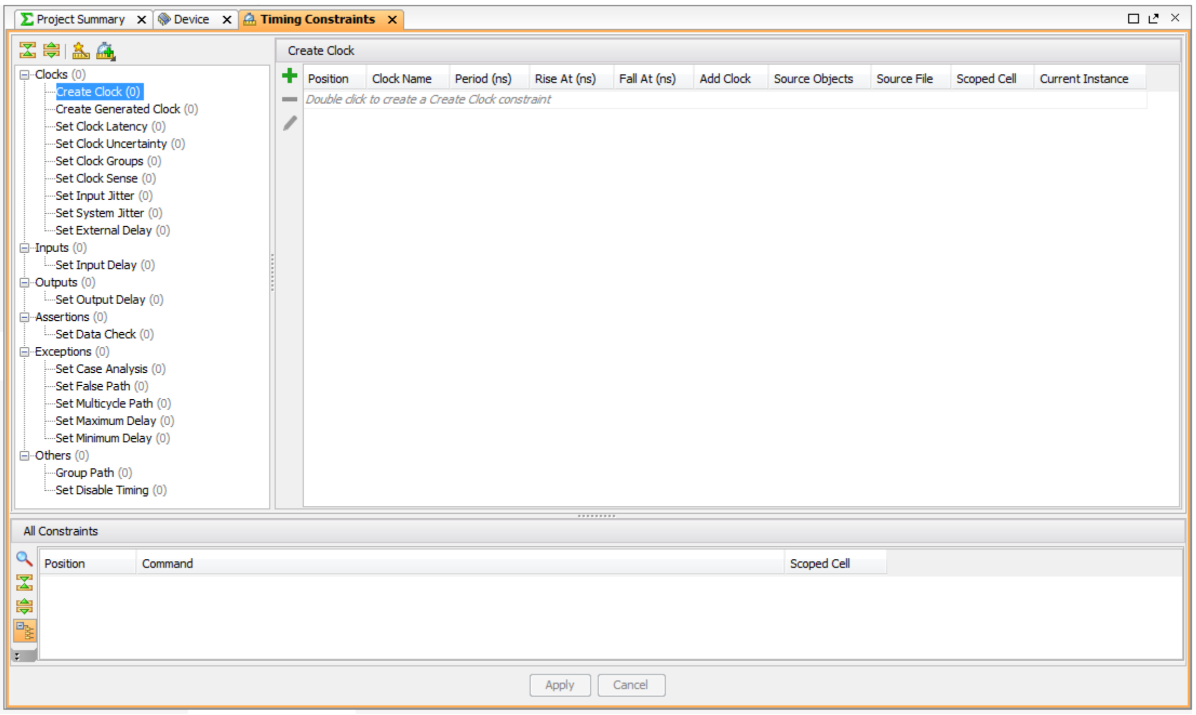
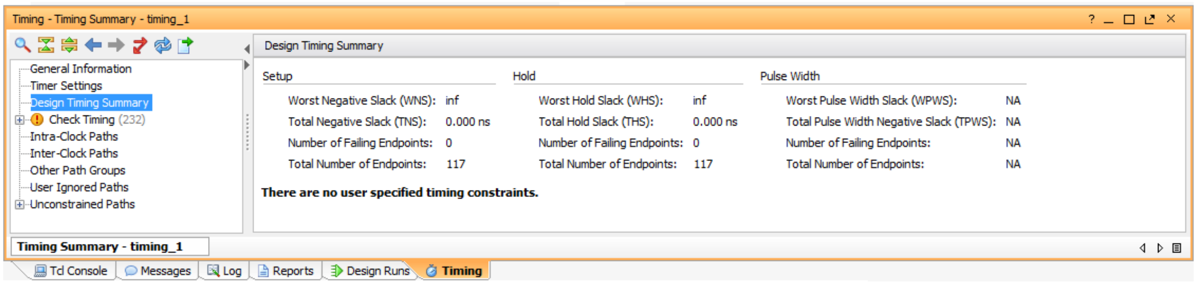


Figure 6.10: Timing Constraints window

- ***Set Up Debug***: The Vivado IDE provides Set up Debug wizard to help guide you through the process of automatically creating the debug cores and assigning the debug nets to the inputs of the cores.

- ***Report Timing Summary***: Generate a default timing report (using estimated timing information), see Illustration 6.11. Timing Reports can be generated at any point after synthesis.

  – *Tcl command* equivalent to this option is: ***report_timing_summary***



Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	inf	Worst Hold Slack (WHS):	inf	Worst Pulse Width Slack (WPWS):	NA
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	NA
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	NA
Total Number of Endpoints:	117	Total Number of Endpoints:	117	Total Number of Endpoints:	NA

There are no user specified timing constraints.

Figure 6.11: Timing Summary Report

- ***Report Clock Networks***: Generates a clock tree for the design, see Illustration 6.12. This option creates a tree view of all the logical clock trees found in the design, annotated with existing and missing clock definitions and the roots of these trees.

  – *Tcl command* equivalent for this option will be: ***report_clock_network***

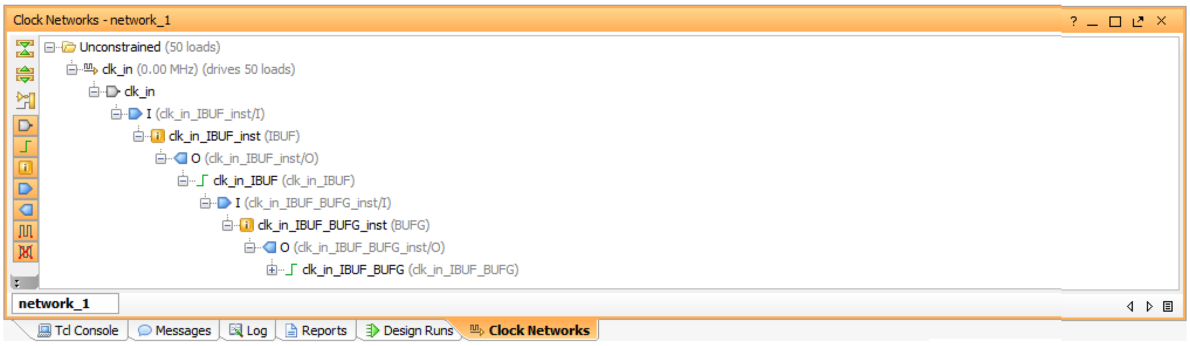

Clock Networks - network_1
Unconstrained (50 loads)
clk_in (0.00 MHz) (drives 50 loads)
clk_in
I (clk_in_IBUF_inst/I)
clk_in_IBUF_inst (IBUF)
O (clk_in_IBUF_inst/O)
clk_in_IBUF (clk_in_IBUF)
I (clk_in_IBUF_BUFG_inst/I)
clk_in_IBUF_BUFG_inst (BUFG)
O (clk_in_IBUF_BUFG_inst/O)
clk_in_IBUF_BUFG (clk_in_IBUF_BUFG)
network_1
Tcl Console
Messages
Log
Reports
Design Runs
Clock Networks

Figure 6.12: Clock Networks Report

- ***Report Clock Interaction***: Verifies constraint coverage on paths between clock domains. This option uses an inter-clock path matrix to show clock relationships and group paths. This report is helpful to tell us if timing is asynchronous (in case that we didn't include synchronization circuitry) and if paths are constrained (in case that we didn't add timing constraints to cover paths between unrelated clock domains). Green squares confirm that paths between the two clock domains are constrained.

  – *Tcl command* equivalent to this option is: ***report_clock_interaction***

- ***Report Methodology***: The Vivado Design Suite provides automated methodology checks based on the UltraFast Design Methodology Guide for the Vivado Design Suite using the Report Methodology command. You can generate a methodology report on an opened, elaborated, synthesized, or implemented design. Running the methodology report allows you to find design issues early during the elaboration stage prior to synthesis, which saves time in the design process.

  – *Tcl command* equivalent to this option is: ***report_methodology -name*** <***results_name***>

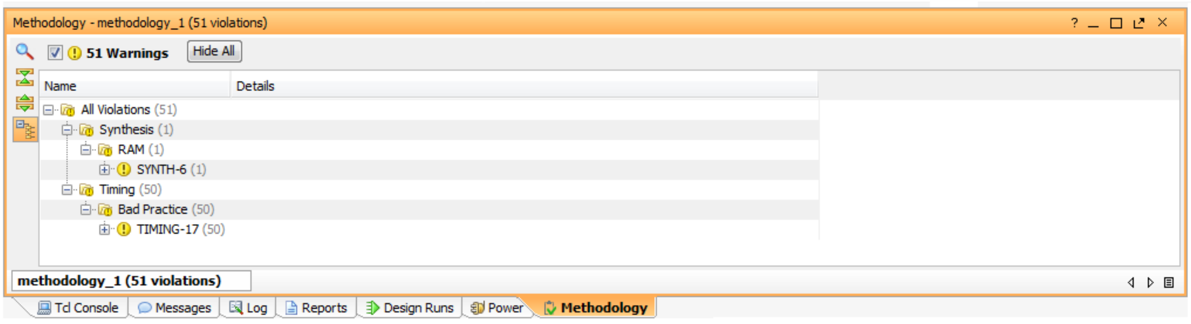
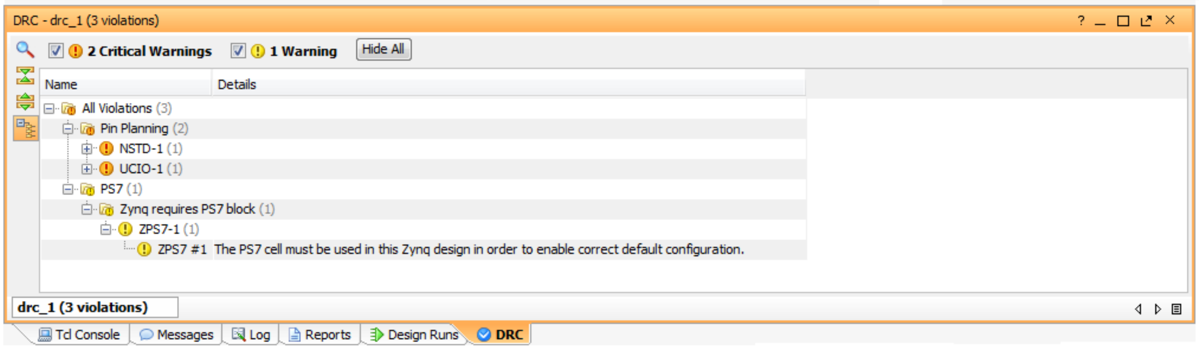


Figure 6.13: Report Methodology

- ***Report DRC***: Performs design rule check on the entire design. DRCs performed early in the design flow allow for correction before a full implementation. We can select which DRCs we would like to run, see Illustration 6.14, or we can select to run all. Objects listed in the violations are cross-selectable with HDL sources. Any problems will open a DRC window at the bottom of the Vivado GUI. If you would like to see the final sign-off DRC, run the implementation process.



DRC - drc_1 (3 violations)

2 Critical Warnings | 1 Warning | Hide All

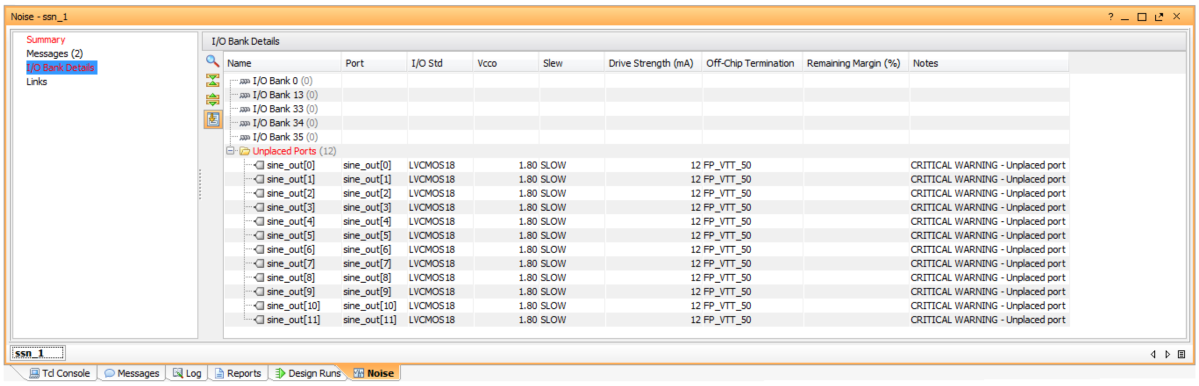
Name	Details
All Violations (3)	
Pin Planning (2)	
NSTD-1 (1)	
UCIO-1 (1)	
PS7 (1)	
Zynq requires PS7 block (1)	
ZPS7-1 (1)	
ZPS7 #1	The PS7 cell must be used in this Zynq design in order to enable correct default configuration.

drc_1 (3 violations)

Tcl Console | Messages | Log | Reports | Design Runs | DRC

Figure 6.14: DRC Report

- ***Report Noise***: Performs an SSN analysis of output and bidirectional pins in the design. This report is looking a gauge the number of pins, I/O standard, and drive strength on a bank-by-bank basis, see Illustration 6.15. Banks that are exceed, what is recommended, will be flagged in the Summary tab. SSN analysis can only be done on output and bidirectional ports.



Noise - ssn_1

Summary
Messages (2)
I/O Bank Details
Links

I/O Bank Details

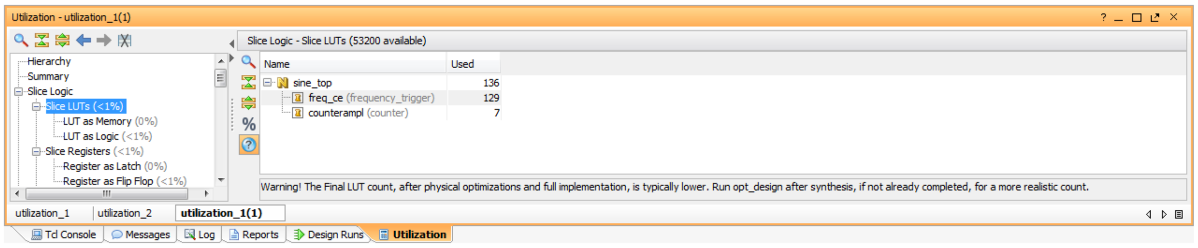
Name	Port	I/O Std	Vcco	Slew	Drive Strength (mA)	Off-Chip Termination	Remaining Margin (%)	Notes
I/O Bank 0 (0)								
I/O Bank 13 (0)								
I/O Bank 33 (0)								
I/O Bank 34 (0)								
I/O Bank 35 (0)								
Unplaced Ports (12)								
sine_out[0]	sine_out[0]	LVCMOS18	1.80	SLOW	12	FP_VTT_50		CRITICAL WARNING - Unplaced port
sine_out[1]	sine_out[1]	LVCMOS18	1.80	SLOW	12	FP_VTT_50		CRITICAL WARNING - Unplaced port
sine_out[2]	sine_out[2]	LVCMOS18	1.80	SLOW	12	FP_VTT_50		CRITICAL WARNING - Unplaced port
sine_out[3]	sine_out[3]	LVCMOS18	1.80	SLOW	12	FP_VTT_50		CRITICAL WARNING - Unplaced port
sine_out[4]	sine_out[4]	LVCMOS18	1.80	SLOW	12	FP_VTT_50		CRITICAL WARNING - Unplaced port
sine_out[5]	sine_out[5]	LVCMOS18	1.80	SLOW	12	FP_VTT_50		CRITICAL WARNING - Unplaced port
sine_out[6]	sine_out[6]	LVCMOS18	1.80	SLOW	12	FP_VTT_50		CRITICAL WARNING - Unplaced port
sine_out[7]	sine_out[7]	LVCMOS18	1.80	SLOW	12	FP_VTT_50		CRITICAL WARNING - Unplaced port
sine_out[8]	sine_out[8]	LVCMOS18	1.80	SLOW	12	FP_VTT_50		CRITICAL WARNING - Unplaced port
sine_out[9]	sine_out[9]	LVCMOS18	1.80	SLOW	12	FP_VTT_50		CRITICAL WARNING - Unplaced port
sine_out[10]	sine_out[10]	LVCMOS18	1.80	SLOW	12	FP_VTT_50		CRITICAL WARNING - Unplaced port
sine_out[11]	sine_out[11]	LVCMOS18	1.80	SLOW	12	FP_VTT_50		CRITICAL WARNING - Unplaced port

ssn_1

Tcl Console | Messages | Log | Reports | Design Runs | Noise

Figure 6.15: Noise Report

- ***Report Utilization***: Generates a graphical version of the Utilization Report, see Illustration 6.16.



Utilization - utilization_1(1)

Hierarchy
Summary
Slice Logic
Slice LUTs (<1%)
LUT as Memory (0%)
LUT as Logic (<1%)
Slice Registers (<1%)
Register as Latch (0%)
Register as Flip Flop (<1%)

Slice Logic - Slice LUTs (53200 available)

Name	Used
sine_top	136
freq_ce (frequency_trigger)	129
counterampl (counter)	7

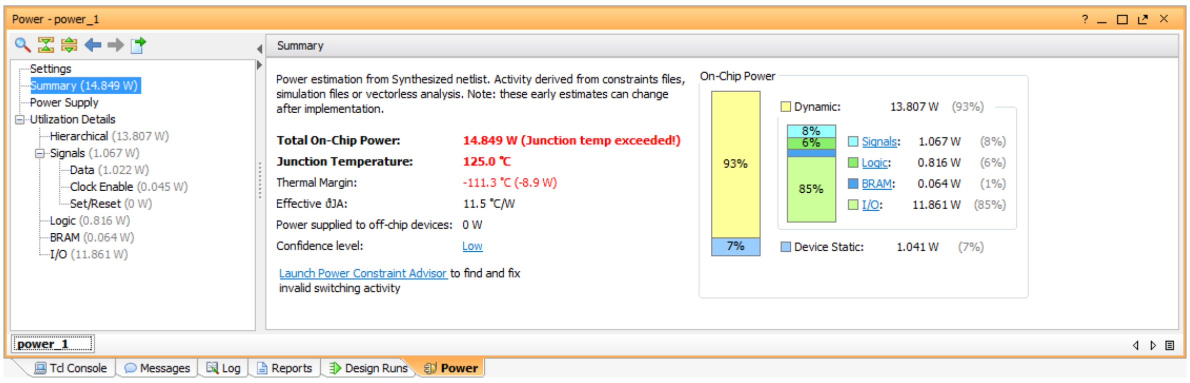
Warning! The Final LUT count, after physical optimizations and full implementation, is typically lower. Run opt_design after synthesis, if not already completed, for a more realistic count.

utilization_1 | utilization_2 | utilization_1(1)

Tcl Console | Messages | Log | Reports | Design Runs | Utilization

Figure 6.16: Utilization Report

- ***Report Power***: Provides detailed power and thermal analysis reports that can be customized for the power supply and application environment, see Illustration 6.17. This report estimates power at every stage after synthesis process. Perform also what-if analysis by varying switching activity.
  - *Tcl command* equivalent to this option is: ***report_power***



Power - power_1

- Settings
- Summary (14.849 W)
- Power Supply
- Utilization Details
  - Hierarchical (13.807 W)
  - Signals (1.067 W)
    - Data (1.022 W)
    - Clock Enable (0.045 W)
    - Set/Reset (0 W)
  - Logic (0.816 W)
  - BRAM (0.064 W)
  - I/O (11.861 W)

Summary

Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

Total On-Chip Power:	**14.849 W (Junction temp exceeded!)**
Junction Temperature:	**125.0 ℃**
Thermal Margin:	-111.3 ℃ (-8.9 W)
Effective ϑJA:	11.5 ℃/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

Dynamic:	13.807 W	(93%)
Signals:	1.067 W	(8%)
Logic:	0.816 W	(6%)
BRAM:	0.064 W	(1%)
I/O:	11.861 W	(85%)
Device Static:	1.041 W	(7%)

Figure 6.17: Power Report

- ***Schematic***: Opens the Schematic window. In the schematic window, you can view design interconnect, hierarchy structure, or trace signal paths for the elaborated design, synthesized design, or implemented design. The Schematic View is explained in detail in the **Sub-chapter 6.5.5 Schematic View**

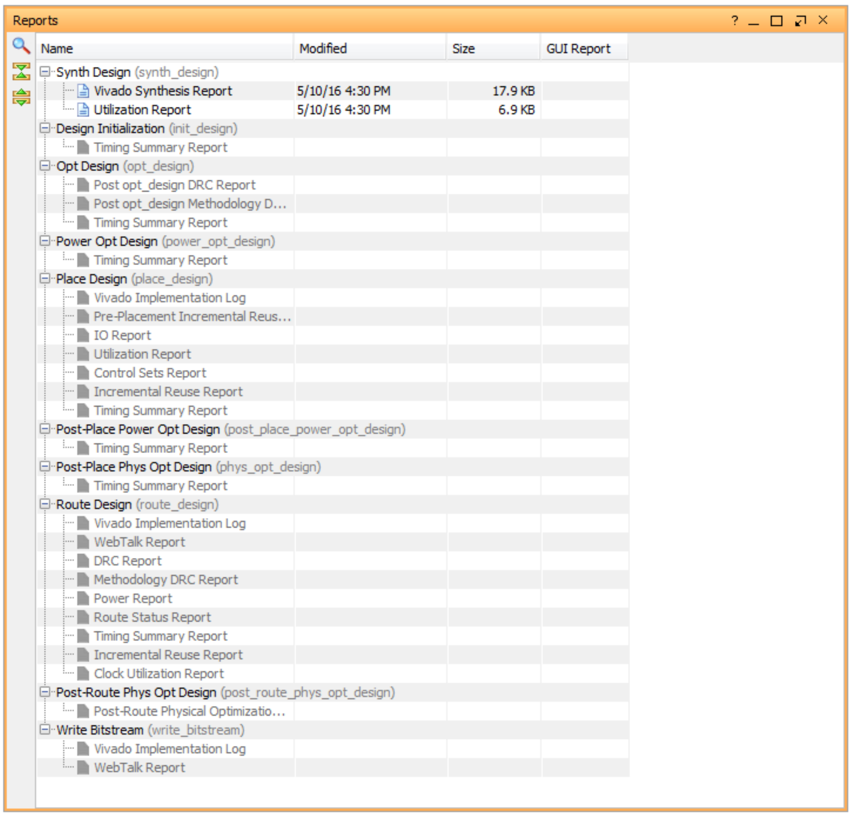
6.5.4 Synthesis Reports

After synthesis completes, you can view the reports, and open, analyze, and use the synthesis design. The reports window contains a list of reports provided by various synthesis and implementation tools in the Vivado IDE.

Open the **Reports** view to explore the reports generated during synthesis process.

To view Synthesis Report:

Step 1. Select the **Reports** tab at the bottom of the IDE, see Illustration 6.18



Reports

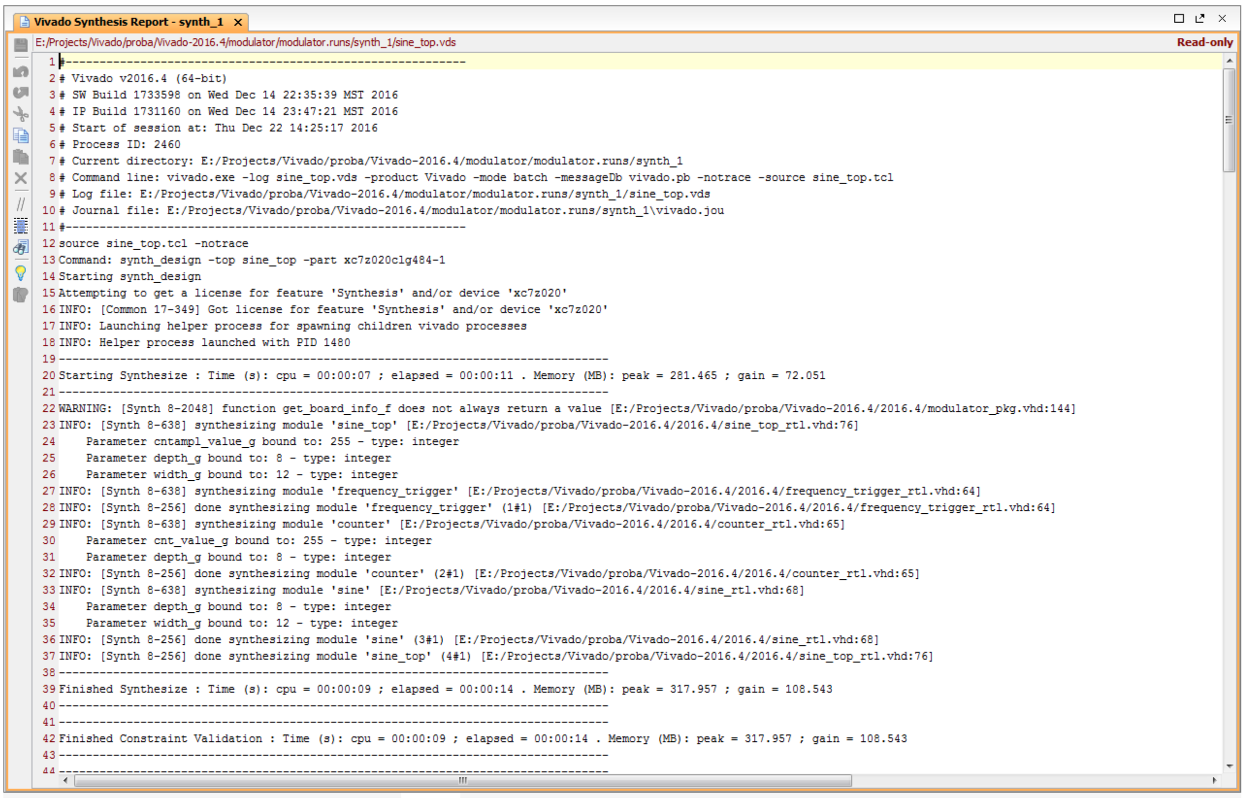
Name	Modified	Size	GUI Report
Synth Design (synth_design)			
Vivado Synthesis Report	5/10/16 4:30 PM	17.9 KB	
Utilization Report	5/10/16 4:30 PM	6.9 KB	
Design Initialization (init_design)			
Timing Summary Report			
Opt Design (opt_design)			
Post opt_design DRC Report			
Post opt_design Methodology D...			
Timing Summary Report			
Power Opt Design (power_opt_design)			
Timing Summary Report			
Place Design (place_design)			
Vivado Implementation Log			
Pre-Placement Incremental Reus...			
IO Report			
Utilization Report			
Control Sets Report			
Incremental Reuse Report			
Timing Summary Report			
Post-Place Power Opt Design (post_place_power_opt_design)			
Timing Summary Report			
Post-Place Phys Opt Design (phys_opt_design)			
Timing Summary Report			
Route Design (route_design)			
Vivado Implementation Log			
WebTalk Report			
DRC Report			
Methodology DRC Report			
Power Report			
Route Status Report			
Timing Summary Report			
Incremental Reuse Report			
Clock Utilization Report			
Post-Route Phys Opt Design (post_route_phys_opt_design)			
Post-Route Physical Optimizatio...			
Write Bitstream (write_bitstream)			
Vivado Implementation Log			
WebTalk Report			

Figure 6.18: Reports tab

Note: If this tab is not shown, select from the main menu **Windows -> Reports**

Step 2. In the **Reports** tab, double-click on the **Vivado Synthesis Report** to open it and examine the report contents, see Illustration 6.18

Vivado Synthesis Report - is a detailed resource that describes the synthesis process. It describes source file recognition, IP attributes, RTL synthesis, logic optimization, primitive inference, technology mapping, and cell usage, see Illustration 6.19.



Vivado Synthesis Report - synth_1

E:/Projects/Vivado/proba/Vivado-2016.4/modulator/modulator.runs/synth_1/sine_top.vds

Read-only

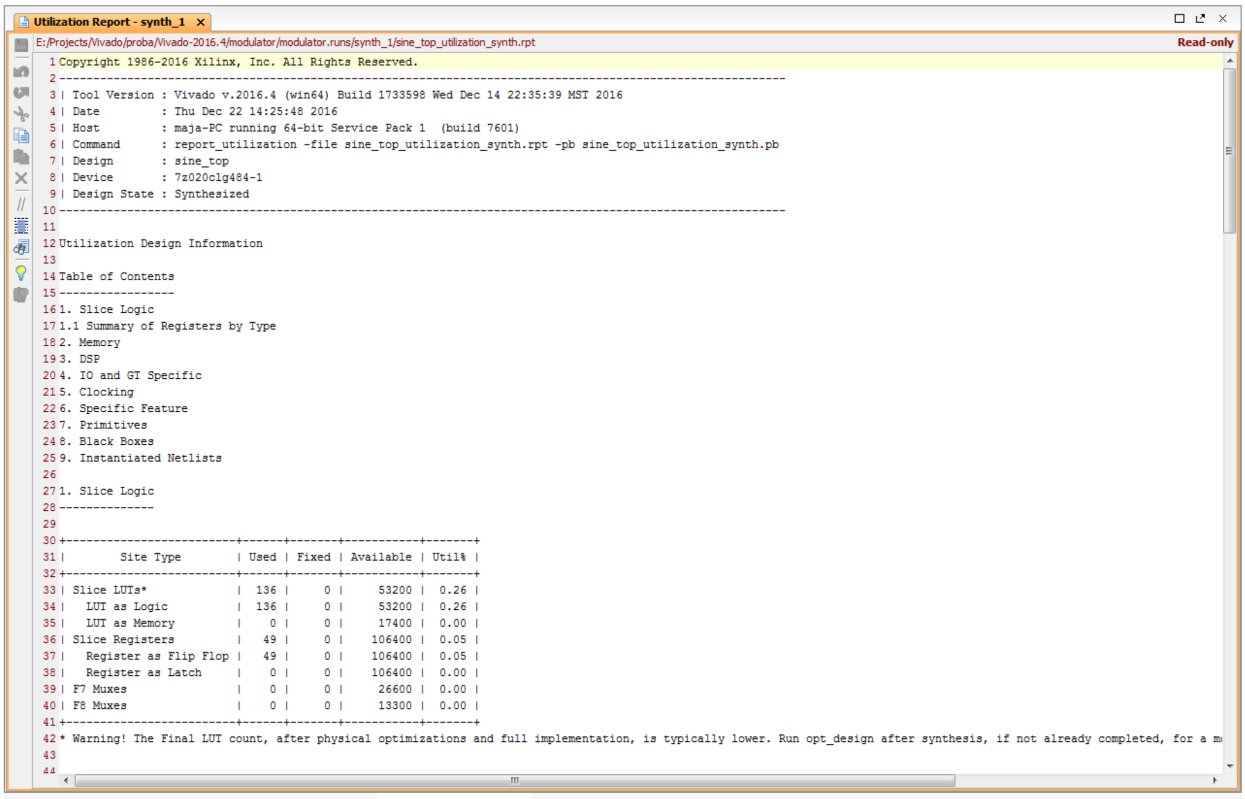
```
#------------------------------------------------------------
# Vivado v2016.4 (64-bit)
# SW Build 1733598 on Wed Dec 14 22:35:39 MST 2016
# IP Build 1731160 on Wed Dec 14 23:47:21 MST 2016
# Start of session at: Thu Dec 22 14:25:17 2016
# Process ID: 2460
# Current directory: E:/Projects/Vivado/proba/Vivado-2016.4/modulator/modulator.runs/synth_1
# Command line: vivado.exe -log sine_top.vds -product Vivado -mode batch -messageDb vivado.pb -notrace -source sine_top.tcl
# Log file: E:/Projects/Vivado/proba/Vivado-2016.4/modulator/modulator.runs/synth_1/sine_top.vds
# Journal file: E:/Projects/Vivado/proba/Vivado-2016.4/modulator/modulator.runs/synth_1\vivado.jou
#------------------------------------------------------------
source sine_top.tcl -notrace
Command: synth_design -top sine_top -part xc7z020clg484-1
Starting synth_design
Attempting to get a license for feature 'Synthesis' and/or device 'xc7z020'
INFO: [Common 17-349] Got license for feature 'Synthesis' and/or device 'xc7z020'
INFO: Launching helper process for spawning children vivado processes
INFO: Helper process launched with PID 1480
---------------------------------------------------------------------------------
Starting Synthesize : Time (s): cpu = 00:00:07 ; elapsed = 00:00:11 . Memory (MB): peak = 281.465 ; gain = 72.051
---------------------------------------------------------------------------------
WARNING: [Synth 8-2048] function get_board_info_f does not always return a value [E:/Projects/Vivado/proba/Vivado-2016.4/2016.4/modulator_pkg.vhd:144]
INFO: [Synth 8-638] synthesizing module 'sine_top' [E:/Projects/Vivado/proba/Vivado-2016.4/2016.4/sine_top_rtl.vhd:76]
    Parameter cntampl_value_g bound to: 255 - type: integer
    Parameter depth_g bound to: 8 - type: integer
    Parameter width_g bound to: 12 - type: integer
INFO: [Synth 8-638] synthesizing module 'frequency_trigger' [E:/Projects/Vivado/proba/Vivado-2016.4/2016.4/frequency_trigger_rtl.vhd:64]
INFO: [Synth 8-256] done synthesizing module 'frequency_trigger' (1#1) [E:/Projects/Vivado/proba/Vivado-2016.4/2016.4/frequency_trigger_rtl.vhd:64]
INFO: [Synth 8-638] synthesizing module 'counter' [E:/Projects/Vivado/proba/Vivado-2016.4/2016.4/counter_rtl.vhd:65]
    Parameter cnt_value_g bound to: 255 - type: integer
    Parameter depth_g bound to: 8 - type: integer
INFO: [Synth 8-256] done synthesizing module 'counter' (2#1) [E:/Projects/Vivado/proba/Vivado-2016.4/2016.4/counter_rtl.vhd:65]
INFO: [Synth 8-638] synthesizing module 'sine' [E:/Projects/Vivado/proba/Vivado-2016.4/2016.4/sine_rtl.vhd:68]
    Parameter depth_g bound to: 8 - type: integer
    Parameter width_g bound to: 12 - type: integer
INFO: [Synth 8-256] done synthesizing module 'sine' (3#1) [E:/Projects/Vivado/proba/Vivado-2016.4/2016.4/sine_rtl.vhd:68]
INFO: [Synth 8-256] done synthesizing module 'sine_top' (4#1) [E:/Projects/Vivado/proba/Vivado-2016.4/2016.4/sine_top_rtl.vhd:76]
---------------------------------------------------------------------------------
Finished Synthesize : Time (s): cpu = 00:00:09 ; elapsed = 00:00:14 . Memory (MB): peak = 317.957 ; gain = 108.543
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
Finished Constraint Validation : Time (s): cpu = 00:00:09 ; elapsed = 00:00:14 . Memory (MB): peak = 317.957 ; gain = 108.543
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
```

Figure 6.19: Vivado Synthesis Report

Step 3. When finished, close the report

Step 4. In the **Reports** tab, double-click on the **Utilization Report** to examine its content, see Illustration 6.18

Utilization Report - describes the amount of device resources that the synthesized design is expected to use, see Illustration 6.20


Utilization Report - synth_1
E:/Projects/Vivado/proba/Vivado-2016.4/modulator/modulator.runs/synth_1/sine_top_utilization_synth.rpt
Read-only

```
Copyright 1986-2016 Xilinx, Inc. All Rights Reserved.
------------------------------------------------------------------------------------------------------------
| Tool Version : Vivado v.2016.4 (win64) Build 1733598 Wed Dec 14 22:35:39 MST 2016
| Date         : Thu Dec 22 14:25:48 2016
| Host         : maja-PC running 64-bit Service Pack 1  (build 7601)
| Command      : report_utilization -file sine_top_utilization_synth.rpt -pb sine_top_utilization_synth.pb
| Design       : sine_top
| Device       : 7z020clg484-1
| Design State : Synthesized
------------------------------------------------------------------------------------------------------------

Utilization Design Information

Table of Contents
-----------------
1. Slice Logic
1.1 Summary of Registers by Type
2. Memory
3. DSP
4. IO and GT Specific
5. Clocking
6. Specific Feature
7. Primitives
8. Black Boxes
9. Instantiated Netlists

1. Slice Logic
--------------

+-------------------------+------+-------+-----------+-------+
|        Site Type        | Used | Fixed | Available | Util% |
+-------------------------+------+-------+-----------+-------+
| Slice LUTs*             |  136 |     0 |     53200 |  0.26 |
|   LUT as Logic          |  136 |     0 |     53200 |  0.26 |
|   LUT as Memory         |    0 |     0 |     17400 |  0.00 |
| Slice Registers         |   49 |     0 |    106400 |  0.05 |
|   Register as Flip Flop |   49 |     0 |    106400 |  0.05 |
|   Register as Latch     |    0 |     0 |    106400 |  0.00 |
| F7 Muxes                |    0 |     0 |     26600 |  0.00 |
| F8 Muxes                |    0 |     0 |     13300 |  0.00 |
+-------------------------+------+-------+-----------+-------+
* Warning! The Final LUT count, after physical optimizations and full implementation, is typically lower. Run opt_design after synthesis, if not already completed, for a m
```

Figure 6.20: Utilization Report

6.5.5 Schematic View

The Schematic view allows selective expansion and exploration of the logical design. You can generate schematic view for any level of the logical or physical hierarchy. You can select a logic element in an open window, such as primitive or net in the Netlist window, and use the Schematic command in the popup menu to create a Schematic window for the selected object. An elaborated design always opens with a Schematic window of the top-level of the design. In the Schematic window, you can view design interconnect, hierarchy structure, or trace signal paths for the elaborated design, synthesized design, or implemented design.

To create a schematic view, do the following steps:

Step 1. Select one or more logic elements in an open window, such as the Netlist window

Step 2. In the **Flow Navigator** / **Synthesis** / **Synthesized Design** click the **Schematic** command, see Illustration 6.21

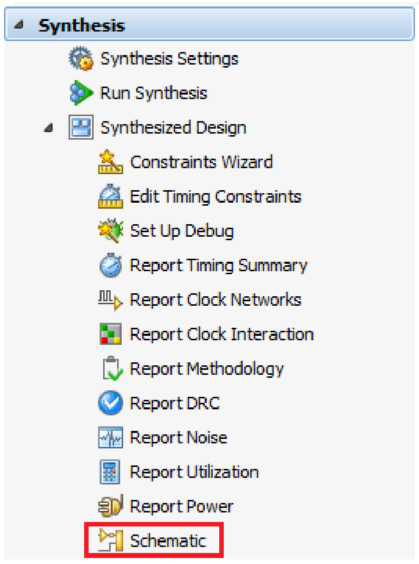


Figure 6.21: Schematic command

Step 3. After few seconds, **Schematic** window will show up, and your design should look similar to the design shown on the Illustration 6.22

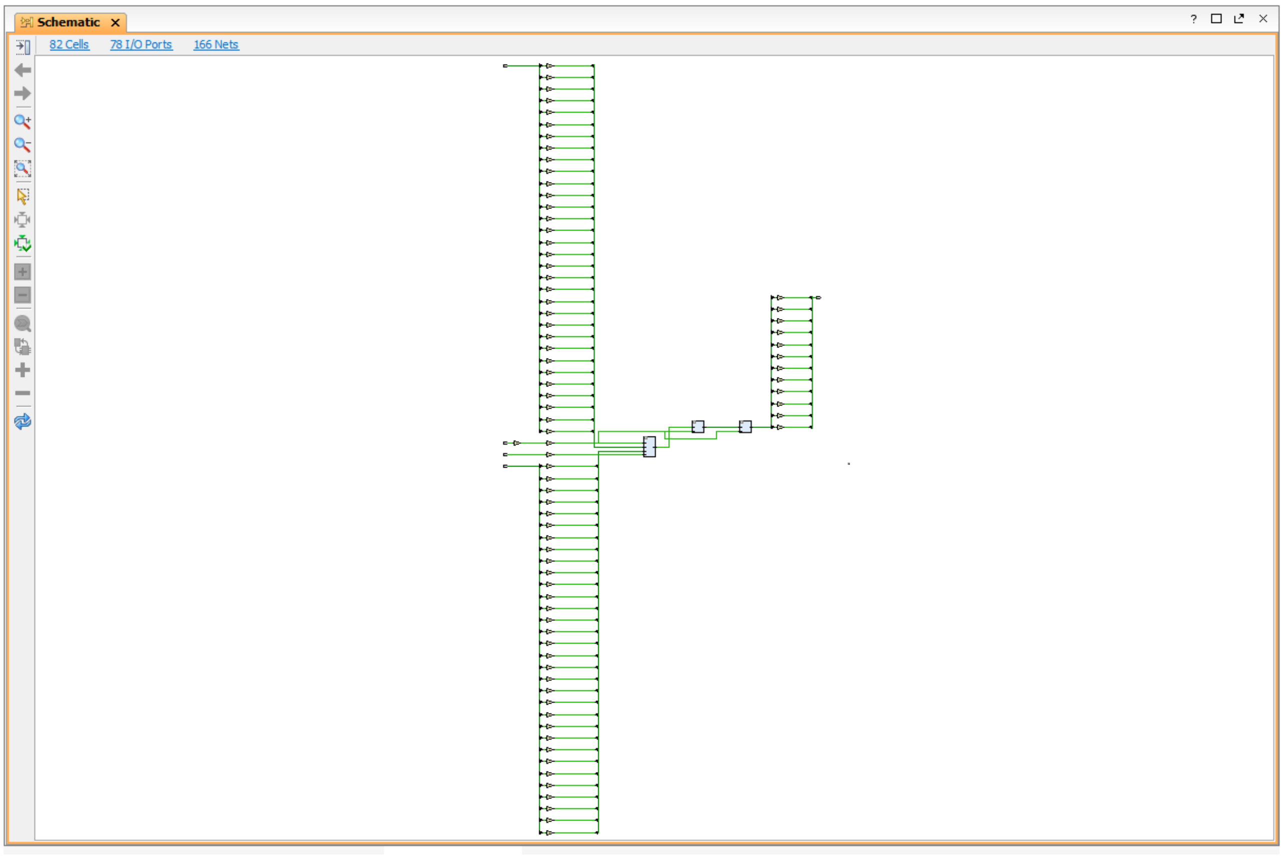


Figure 6.22: Sine-Top Schematic View

The Schematic window displays the selected logic cells or nets. If only one cell is selected, schematic symbol for that module will be displayed.

In the Schematic window, you can find and view objects as follows:

- The links as the top of the schematic sheet, labelled **Cells** , **I/O Ports**, and **Nets**, open a searchable list in the Find

Results window, making it easier to find specific items in the schematic.

- When you select objects in the schematic window, those objects are also selected in all other windows. If you opened an implemented design, the cells and nets display in the Device window.

Schematic Window Toolbar Commands

The local toolbar contains the following commands:

- **Schematic Options** - Configures the display of the Schematic window
- **Previous Position** - Resets the Schematic window to display the prior zoom, coordinates and logic content
- **Next Position** - Returns the Schematic window to display the original zoom, coordinates and logic content after Previous Position is used
- **Zoom In** - Zooms in the Schematic window (Ctrl + Equals)
- **Zoom Out** - Zooms out the Schematic window (Ctrl + Minus)
- **Zoom Fit** - Zooms out to fit the whole schematic into the display area (Ctrl + 0)
- **Select Area** - Selects the objects in the specified rectangular area
- **Fit Selection** - Redraws the Schematic window to display the currently selected objects. This is useful when selecting objects are in another window and you want to redraw the display around those selected objects
- **Autofit Selection** - Automatically redraws the Schematic window around newly selected objects. This mode can be enabled or displayed
- **Expand all logic inside selected cell** - Expands a hierarchical module from the symbol view to the logic view.

  Note: Hierarchical modules can also be expanded directly from the schematic by clicking the plus (+) icon on the schematic symbol
- **Collapse all logic inside selected cell** - Collapses a hierarchical module from the logic view to the symbol view.

  Note: An expanded hierarchical block can also be collapsed directly from the schematic by clicking the minus (-) icon on the hierarchical block
- **Magnify**: Displays a detailed popup view of the selected bus pin

  Note: Alternatively, you can press Ctrl and double-click a bus pin.
- **Toggle autohide pins for selected cell** - Toggles the pin display on selected hierarchical modules. Higher levels of the hierarchy display as concentric rectangles without pins, when a Schematic window is generated. In most cases, the lack of pins makes the Schematic window more readable. However, you can display the pins for selected cells as needed
- **Add selected elements to schematic** - Recreates the Schematic window when the newly selected elements added to the existing schematic
- **Remove selected elements from the schematic** - Recreates the Schematic window with the currently selected elements removed from the existing schematic
- **Regenerate Schematic** - Redraws the active Schematic window

Chapter 7

PWM

7.1 Description

- ***Usage***: This module will generate an PWM signal modulated using the digital sine wave from the Digital Sine module. This module will be composed of two independent modules. One will be the Frequency Trigger, for generating two different frequencies and the second one will be the Finite State Machine (FSM), for generating the PWM signal.

  Frequency Trigger module is the same module explained as in the Chapter 2. FREQUENCY TRIGGER. We need a second Frequency Trigger module in our design, because this module will generate *freq_trig* signal with 2^{width} higher frequency than the *freq_trig* signal of the first Frequency Trigger module. This is important for proper PWM signal generation.

  FSM module will generate the PWM signal. It will generate the PWM signal with correct duty cycle for each period based on the current amplitude value of digital sine signal, that is stored in the ROM. State diagram of the FSM is shown on the Figure 7.2.

- ***Block diagram:***

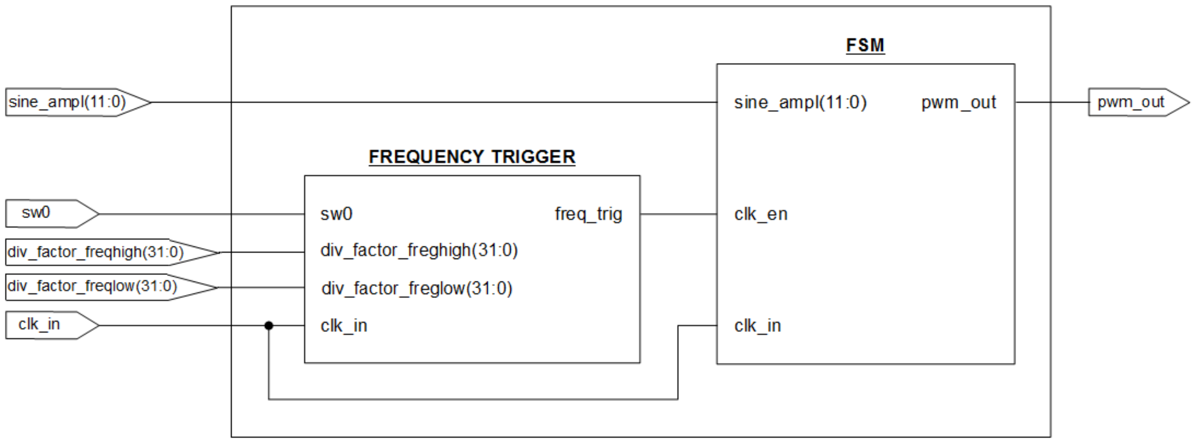


Figure 7.1: PWM block diagram

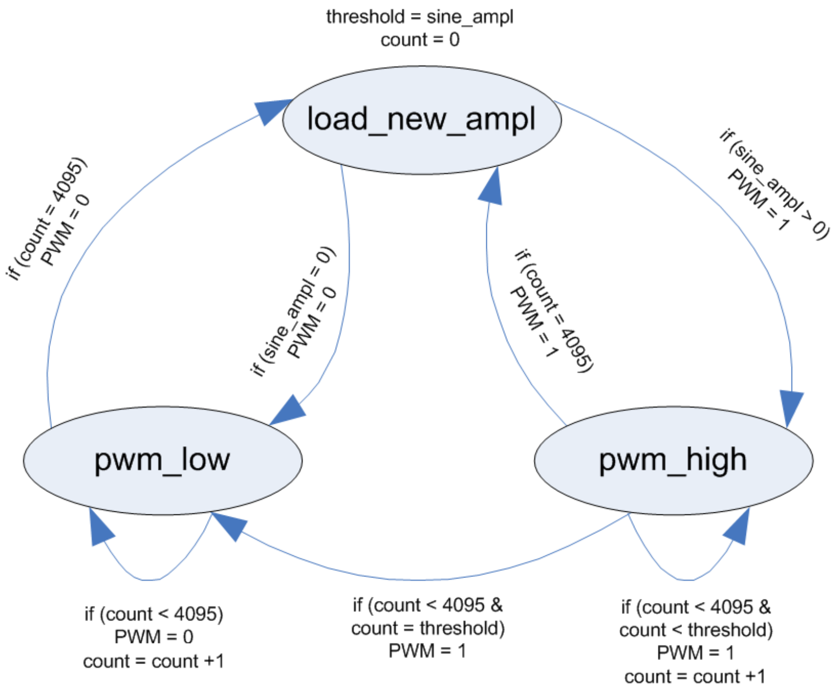


Figure 7.2: FSM state diagram

- ***Input ports:***
  - **clk_in:** input clock signal
  - **sw0**: input signal from the on-board switch, used for changing output signal frequency
  - **sine_ampl**: current amplitude value of the sine signal
  - **div_factor_freqhigh**: input clock division when sw0 = ’1’
  - **div_factor_freqlow**: input clock division when sw0 = ’0’
- ***Output ports:***
  - **pwm_out**: pulse width modulated signal
- ***Generics:***
  - **width_g**: the number of bits used to represent amplitude value
- ***File name:*** pwm_rtl.vhd

7.2 Creating Module

To create PWM module, use steps for creating modules, **Sub-chapter 2.4.1 Creating a Module Using Vivado Text Editor**.

PWM VHDL model:

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

entity pwm is
    generic(
        width_g : integer range 1 to 99 := 12; -- the number of bits used to represent amplitude value
        );

    port(
        clk_in              : in std_logic;                           -- input clock signal
        sw0                 : in std_logic;                           -- signal made for selecting frequency
        sine_ampl           : in std_logic_vector(width_g-1 downto 0); -- current amplitude value of the
                                                                         -- sine signal
        div_factor_freqhigh : in std_logic_vector(31 downto 0);      -- input clock division when sw0 = '1'
        div_factor_freqlow  : in std_logic_vector(31 downto 0);   -- input clock division when sw0 = '0'
        pwm_out             : out std_logic                       -- pulse width modulated signal
        );
end entity;


architecture rtl of pwm is

    type state_t is (load_new_ampl, pwm_high, pwm_low);   -- states load_new_ampl, pwm_high, pwm_low
    signal state: state_t ;

    signal ce_s : std_logic := '0';                       -- clock enable signal for the fsm

begin

-- Defines a sequential process
-- process1 and process2 will constitute two-process model of the FSM (Finite State Machine)

    -- process1 models state register and next-state logic
    process1_p : process (clk_in)

        -- threshold_v is variable that is telling us when pwm signal should be changed from 1 to 0
        -- integer range 0 to 4095 (in our case)
        variable threshold_v : integer range 0 to ((2**width_g)-1) := 0;
        -- count_v is variable that counts the number of elapsed cycles
        -- when count_v reaches threshold_v value it is time to change pwm signal from 1 to 0
        -- integer range 0 to 4095 (in our case)
        variable count_v     : integer range 0 to ((2**width_g)-1) := 0;

    begin
        if (clk_in = '1' and clk_in'event) then
            if (ce_s = '1') then
                case state is

                    -- in load_new_ampl state we are loading new amplitude value of the sine signal
                    when load_new_ampl =>
                        -- set the threshold_v value to the current value of the sine signal
                        threshold_v := conv_integer (sine_ampl);
                        count_v := 0;            -- default assignment

                        -- if current amplitude of the sine signal is greater than zero, there
                        -- will be a pulse on the PWM signal in the current period
                        -- (PWM will be 1 for a period of time)
                        if (sine_ampl > 0) then
                            state <= pwm_high;

                        -- if current amplitude value is equal to zero, there will be no pulse
                        -- on the PWM signal in the current period (PWM will always be 0)
                        elsif (sine_ampl = 0) then
                            state <= pwm_low;
                        end if;


                    -- when we are in pwm_high state, PWM = 1
                    when pwm_high =>
                        count_v := count_v + 1;   -- increment counter

                        -- while counter value is less than threshold_v, we stay in pwm_high state
                        if (count_v < ((2**width_g)-1) and count_v < threshold_v) then
                            state <= pwm_high;

                        -- if one period of the PWM signal has elapsed we go to load_new_ampl state
                        elsif (count_v = ((2**width_g)-1)) then
                            state <= load_new_ampl;

                        -- if count_v is equal to threshold_v, we go to pwm_low state
                        elsif (count_v < ((2**width_g)-1) and count_v = threshold_v) then
                            state <= pwm_low;
                        end if;


                    -- when we are in pwm_low state, PWM = 0
                    when pwm_low =>
                        count_v := count_v + 1;   -- increment counter

                        -- while counter value is less than 4095, we stay in pwm_low state
                        if (count_v < ((2**width_g)-1)) then
                            state <= pwm_low;

                        -- if count_v is equal to 4095, we go to load_new_ampl state
                        -- to load a new amplitude value of the sine signal
                        elsif (count_v = ((2**width_g)-1)) then
                            state <= load_new_ampl;
                        end if;
                end case;
            end if;
        end if;
    end process process1_p;

    -- process2 models output logic (logic that generates pwm signal)
    process2_p : process (state)
    begin
        case state is
            when load_new_ampl => pwm_out <= '0';
            when pwm_high      => pwm_out <= '1';
            when pwm_low       => pwm_out <= '0';
        end case;
    end process process2_p;


    fsm_ce: entity work.frequency_trigger(rtl)   -- frequency trigger module instance
        port map (
            clk_in              => clk_in,
            sw0                 => sw0,
            div_factor_freqhigh => div_factor_freqhigh,
            div_factor_freqlow  => div_factor_freqlow,
            freq_trig           => ce_s
        );
end;
```

7.3 Creating Test Bench

- ***Usage:*** used to verify correct operation of the PWM module defined in the pwm_rtl.vhd file
- ***Test bench internal signals:***
  - **clk_in_s**: input clock signal
  - **sw0_s**: input signal from the on-board switch, used for changing output signal frequency
  - **sine_out_s**: current amplitude value of the sine signal
  - **pwm_s**: pwm signal
- ***Generics:***
  - **cntampl_value_g:** threshold value for counter, it's value should be equal to $(2^{depth} - 1)$
  - **depth_g:** the number of samples in one period of the signal
  - **width_g:** the number of bits used to represent amplitude value
- ***File name:*** pwm_tb.vhd

We will now create a new simulation set (**sim_4**) with the test bench file for the PWM module **(pwm_tb.vhd)** in it. We will use the steps explained in the **Sub-chapter 3.3 Creating Test Bench**.

PWM test bench:

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

    use work.modulator_pkg.all;

entity pwm_tb is
    generic(
        cntampl_value_g : integer := 255;             -- threshold value for counter,
                                                       -- it's value should be equal to (2^depth)-1
        depth_g         : integer range 1 to 99 := 8; -- the number of samples in one period of the signal
        width_g         : integer range 1 to 99 := 12 -- the number of bits used to represent amplitude
                                                       -- value
        );
end entity;

architecture tb of pwm_tb is

    signal clk_in_s   : std_logic := '0';             -- input clock signal
    signal sine_out_s : std_logic_vector(width_g-1 downto 0) := (others=>'0');
                                                       -- current amplitude value of the sine signal
    signal sw0_s      : std_logic := '0';             -- signal made for selecting frequency
    signal pwm_s      : std_logic := '0';             -- pwm signal

begin
    dut1 : entity work.sine_top    -- sine_top module instance
        generic map(
            cntampl_value_g => cntampl_value_g,
            depth_g         => depth_g,
            width_g         => width_g
            )
        port map(
            clk_in              => clk_in_s,
            sw0                 => sw0_s,
            div_factor_freqhigh => conv_std_logic_vector(1*(2**width_g), 32), -- 1*4096=4096
            div_factor_freqlow  => conv_std_logic_vector(2*(2**width_g), 32), -- 2*4096=8192
            sine_out            => sine_out_s
            );

    dut2 : entity work.pwm          -- pwm module instance
        generic map(
            width_g => width_g
            )

        port map(
            clk_in => clk_in_s,
            sw0                 => sw0_s,
            sine_ampl           => sine_out_s,
            div_factor_freqhigh => conv_std_logic_vector(1, 32),
            div_factor_freqlow  => conv_std_logic_vector(2, 32),
            pwm_out             => pwm_s
            );

    clk_in_s <= not (clk_in_s) after per_c/2; -- input clock signal
    sw0_s    <= '0', '1' after 1 ms;

end;
```

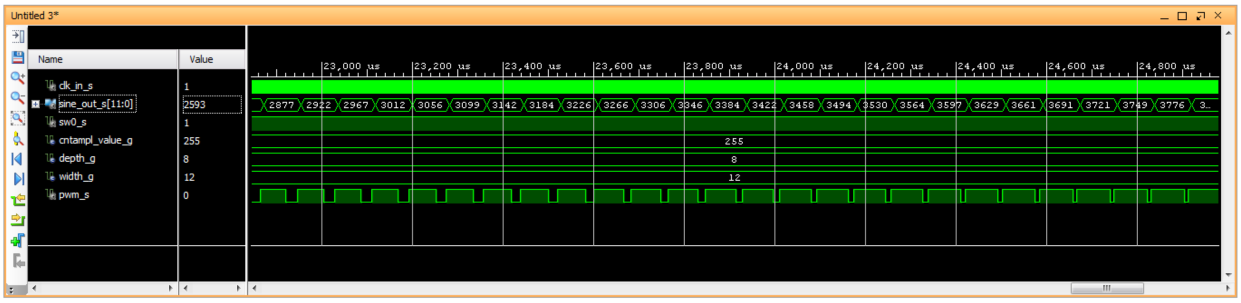
7.4 Simulating

After you have entered the code for the input stimulus in order to perform simulation:

Step 1. You can start your simulation (see **Chapter 3.4 Simulating**)

Step 2. Simulate your design for **25 ms** (see **Chapter 2.6 Simulating – step 9.**)

Step 3. Assuming no errors, your simulation result should look similar to **Illustration 7.3** .



Name	Value
clk_in_s	1
sine_out_s[11:0]	2593
sw0_s	1
cntampl_value_g	255
depth_g	8
width_g	12
pwm_s	0

Figure 7.3: Simulation Results

In this example we have also decreased ***div_factor_freqhigh*** and ***div_factor_freqlow*** values, in the dut1 instance, 10 times to shorten the duration of the simulation process. We done this on the same way like in the **Digital Sine Top test bench** file.

Note: All the information about creating the PWM module, its FSM state diagram, generating the PWM test bench file and simulating the PWM design, you can also find in the **Lab 9: "Creating PWM Module"** .

Chapter 8

MODULATOR

8.1 Description

- ***Usage***: This module will merge all the previously designed modules.
- ***Block diagram:***

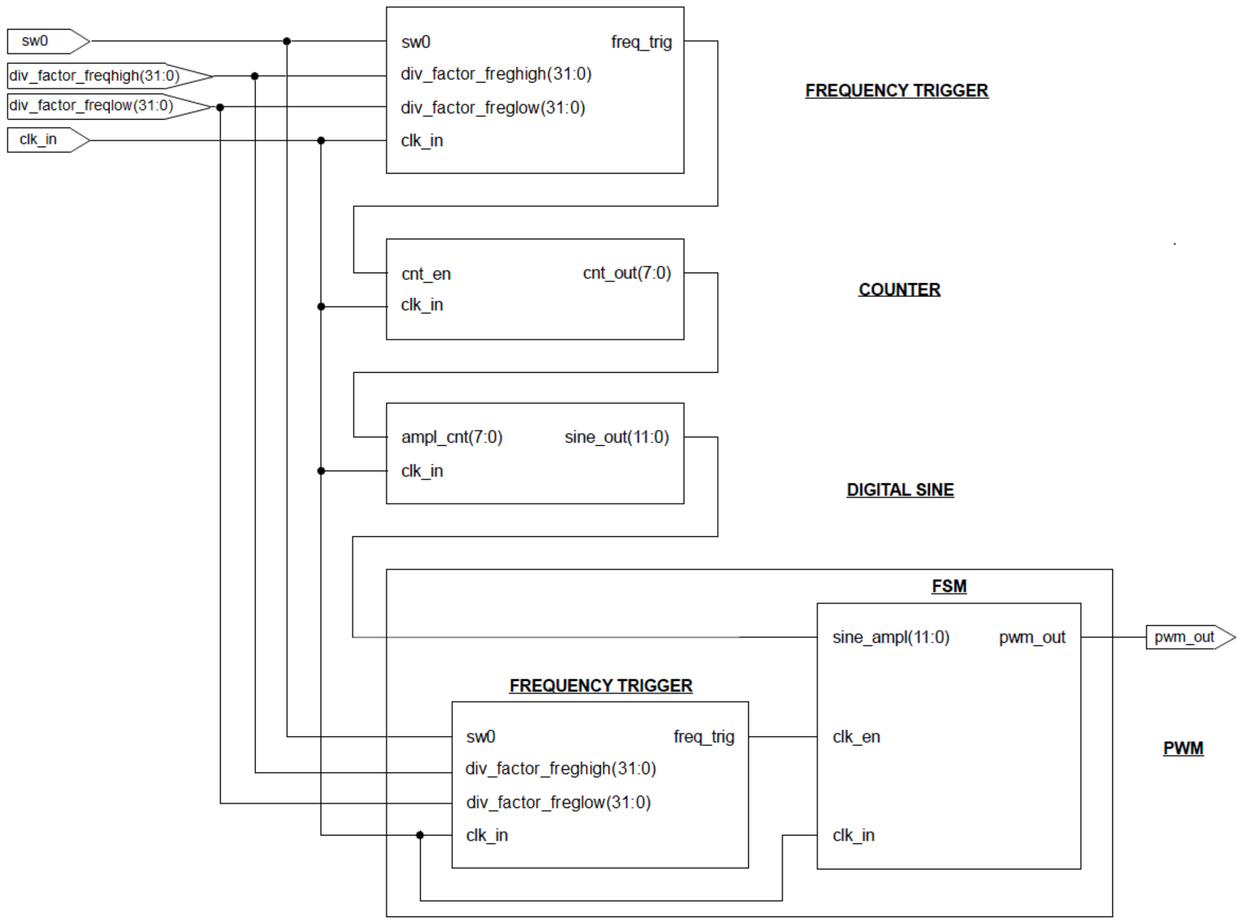


Figure 8.1: Modulator block diagram

- ***Input ports:***
  - **clk_in**: input clock signal
  - **sw0**: input signal from the on-board switch, used for changing output signal frequency

  - **div_factor_freqhigh**: input clock division when sw0 = ’1’
  - **div_factor_freqlow**: input clock division when sw0 = ’0’

- ***Output ports:***
  - **pwm_out**: pulse width modulated signal
- ***Generics:***
  - **design_setting_g**: user defined settings for the pwm design
- ***File name:*** modulator_rtl.vhd

8.2 Creating Module

To create Modulator module use steps for creating modules, **Sub-chapter 2.4.1 Creating a Module Using Vivado Text Editor** .

Modulator VHDL model:

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

    use work.modulator_pkg.all;

library unisim;
    use unisim.vcomponents.all;

entity modulator is
    generic(
    -- User defined settings for the pwm design
    design_setting_g : design_setting_t_rec := design_setting_c
    );

    port(
        clk_in              : in std_logic;                     -- input clock signal
        sw0                 : in std_logic;                     -- signal made for selecting frequency
        div_factor_freqhigh : in std_logic_vector(31 downto 0); -- input clock division when sw0 = ’1’
        div_factor_freqlow  : in std_logic_vector(31 downto 0); -- input clock division when sw0 = ’0’
        pwm_out             : out std_logic                     -- pulse width modulated signal
        );

end entity;

architecture rtl of modulator is

-- amplitude counter
signal ampl_cnt_s  : std_logic_vector(design_setting_g.depth-1 downto 0);
-- current amplitude value of the sine signal
signal sine_ampl_s : std_logic_vector(design_setting_g.width-1 downto 0);
-- signal which frequency depends on the sw0 state
signal freq_trig_s : std_logic := ’0’;

begin

freq_ce : entity work.frequency_trigger(rtl)    -- frequency trigger module instance
    port map(
        clk_in              => clk_in,
        sw0                 => sw0,
        div_factor_freqhigh => div_factor_freqhigh,
        div_factor_freqlow  => div_factor_freqlow,
        freq_trig           => freq_trig_s
        );

counterampl : entity work.counter(rtl)          -- counter module instance
    generic map(
        cnt_value_g => design_setting_g.cntampl_value,
        depth_g     => design_setting_g.depth
        )

    port map (
        clk_in  => clk_in,
        cnt_en  => freq_trig_s,
        cnt_out => ampl_cnt_s
        );

sine : entity work.sine(rtl)                    -- digital sine module instance
    generic map(
        depth_g => design_setting_g.depth,
        width_g => design_setting_g.width
        )

    port map(
        ampl_cnt => ampl_cnt_s,
        clk_in   => clk_in,
        sine_out => sine_ampl_s
        );

pwmmodule : entity work.pwm (rtl)               -- pwm module instance
    generic map (
        width_g => design_setting_g.width
        )

    port map (
        clk_in              => clk_in,
        sw0                 => sw0,
        sine_ampl           => sine_ampl_s,
        div_factor_freqhigh => conv_std_logic_vector(conv_integer(div_factor_freqhigh)/(2**design_setting_g
      .width), 32),
        div_factor_freqlow  => conv_std_logic_vector(conv_integer(div_factor_freqlow)/(2**design_setting_g.
      width), 32),
        pwm_out             => pwm_out
        );

end;
```

8.3 Creating Test Bench

- ***Usage:*** used to verify correct operation of the Modulator module defined in the modulator_rtl.vhd file
- ***Test bench internal signals:***
  - **clk_in_s**: input clock signal
  - **sw0_s**: input signal from the on-board switch, used for changing output signal frequency
  - **pwm_s**: pulse width modulated signal
- ***Generics:***
  - **board_name_g**: 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
  - **design_setting_g**: user defined settings for the pwm design
- ***File name:*** modulator_tb.vhd

We will now create a new simulation set (**sim_5**) with the test bench file for the Modulator module **(modulator_tb.vhd)** in it. We will use the steps explained in the **Sub-chapter 3.3 Creating Test Bench**.

Modulator test bench:

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

    use work.modulator_pkg.all;


entity modulator_tb is
    generic(
        -- 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

        );
end entity;

architecture tb of modulator_tb is
```

```
    signal clk_in_s : std_logic := '1'; -- input clock signal
    signal sw0_s    : std_logic := '1'; -- signal made for selecting frequency
    signal pwm_s    : std_logic := '0'; -- pulse width modulated signal

    -- period of input clock signal
    constant clock_period_c : time := 1000000000.0 / get_board_info_f(board_name_g).fclk * 1ns;

    -- constant created to short the duration of the simulation process 10 times
    constant design_setting1_c : design_setting_t_rec := (255, 10.0, 35.0, 8, 12);

    -- c1_c = fclk/(2^depth*2^width)                   - c1_c = c1_c = 95.3674, fclk = 100 MHz
    constant c1_c : real :=
       get_board_info_f(board_name_g).fclk/(real((2**design_setting1_c.depth)*(2**design_setting1_c.width)));
     -- div_factor_freqhigh_c = (c1_c/f_high)*2^width  - threshold value of frequency a = 110592
    constant div_factor_freqhigh_c : integer :=
       integer(c1_c/design_setting1_c.f_high)*(2**design_setting1_c.width);
    -- div_factor_freqlow_c  = (c1_c/f_low)*2^width    - threshold value of frequency b = 389120
    constant div_factor_freqlow_c  : integer :=
       integer(c1_c/design_setting1_c.f_low)*(2**design_setting1_c.width);


begin

    pwmmodulator : entity work.modulator   -- modulator module instance
        generic map(
            design_setting_g => design_setting1_c
            )

        port map(
            clk_in              => clk_in_s,
            sw0                 => sw0_s,
            div_factor_freqhigh => conv_std_logic_vector(div_factor_freqhigh_c, 32),
            div_factor_freqlow  => conv_std_logic_vector(div_factor_freqlow_c, 32),
            pwm_out             => pwm_s
            );

    clk_in_s <= not (clk_in_s) after clock_period_c/2; -- generates input clock signal
    sw0_s   <= '1', '0' after 25 us;

end;
```

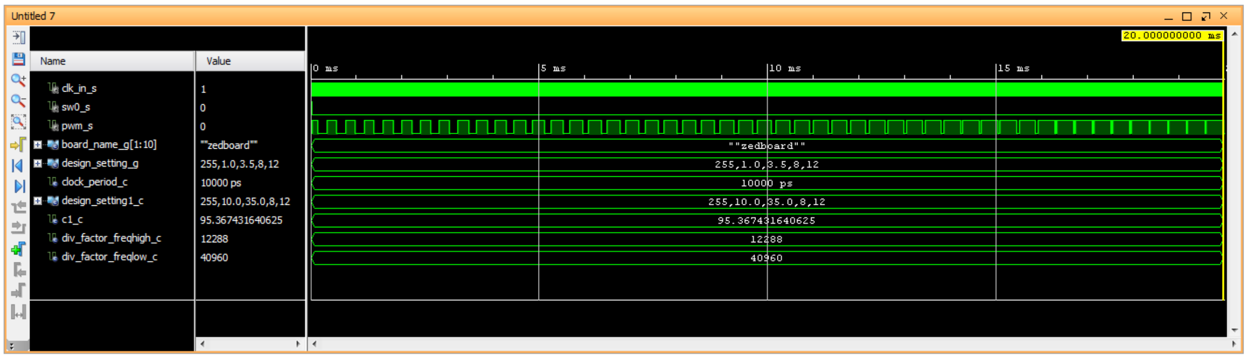
8.4 Simulating

After you have entered the code for the input stimulus in order to perform simulation:

Step 1. You can start your simulation (see **Chapter 3.4 Simulating**)

Step 2. Simulate your design for **20 ms (see Chapter 2.6 Simulating - step 9.)**

Step 3. Assuming no errors, your simulation result should look similar to **Illustration 8.2**



```
Name                      Value
clk_in_s                  1
sw0_s                     0
pwm_s                     0
board_name_g[1:10]        ""zedboard""
design_setting_g          255,1.0,3.5,8,12
clock_period_c            10000 ps
design_setting1_c         255,10.0,35.0,8,12
c1_c                      95.367431640625
div_factor_freqhigh_c     12288
div_factor_freqlow_c      40960
```

Figure 8.2: Simulation Results

In this example we have also shortened the duration of the simulation process by defining the new ***design_setting1_c*** constant in the *modulator_tb.vhd* file. As you can see from the *modulator_tb.vhd* source code we shortened the duration of the simulation process 10 times, so the simulation should now lasts **20 ms** instead of **200 ms** .

Note: All the information about creating the Modulator module, generating its test bench file and simulating the Modulator design, you can also find in the **Lab 10: "Creating Modulator Module"** .

Chapter 9

MODULATOR WRAPPER

9.1 Description

- ***Usage***: This module is necessary to support different development boards with different referent clock types (single-ended and differential clocks). In this module we will instantiate Modulator module and, if needed, differential input clock buffer. Differential input clock buffer will be instantiated if the target development board has reference clock source with differential output.
- ***Block diagram***:

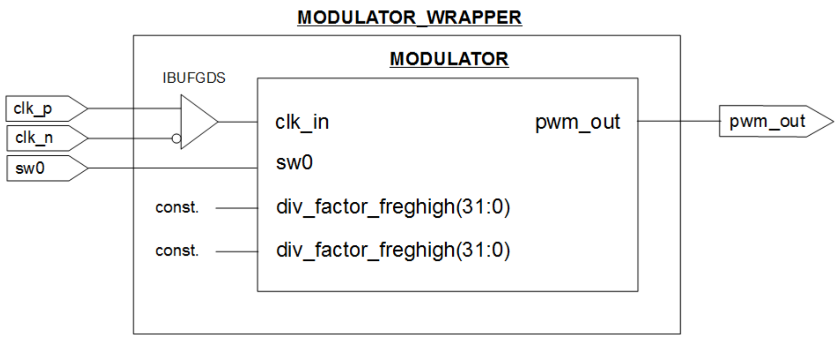


Figure 9.1: Modulator wrapper block diagram

- ***Input ports***:
  - **clk_p:** differential input clock signal
  - **clk_n:** differential input clock signal
  - **sw0:** input signal from the on-board switch, used for changing output signal frequency
- ***Output ports***:
  - **pwm_out:** pulse width modulated signal
- ***Generics***:
  - **this_module_is_top_g:** if some module is top, it needs to implement the differential clk buffer, otherwise this variable will be overwritten by a upper hierarchy layer
  - **board_name_g:** 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
  - **design_setting_g:** user defined settings for the pwm design
- ***File name***: modulator_wrapper_rtl.vhd

9.2 Creating Module

To create Modulator wrapper module use steps for creating modules, **Sub-chapter 2.4.1 Creating a Module Using Vivado Text Editor** .

Modulator wrapper VHDL model:

```
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 modulator_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 rtl of modulator_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;

    -- c1_c = fclk/(2^depth*2^width)                    - c1_c = 95.3674, fclk = 100 MHz
    constant c1_c : real :=
       get_board_info_f(board_name_g).fclk/(real((2**design_setting_g.depth)*(2**design_setting_g.width)));
     -- div_factor_freqhigh_c = (c1_c/f_high)*2^width  - threshold value of frequency a = 110592
    constant div_factor_freqhigh_c : integer :=
       integer(c1_c/design_setting_g.f_high)*(2**design_setting_g.width);
    -- div_factor_freqlow_c  = (c1_c/f_low)*2^width    - threshold value of frequency b = 389120
    constant div_factor_freqlow_c  : integer :=
       integer(c1_c/design_setting_g.f_low)*(2**design_setting_g.width);

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;


    pwmmodulator : entity work.modulator   -- modulator module instance
        generic map(
            design_setting_g => design_setting_g
            )

        port map(
            clk_in               => clk_in_s,
            sw0                  => sw0,
            div_factor_freqhigh => conv_std_logic_vector(div_factor_freqhigh_c, 32),
            div_factor_freqlow  => conv_std_logic_vector(div_factor_freqlow_c, 32),
            pwm_out              => pwm_out
            );

end;
```

Note: All the information about creating the Modulator Wrapper module, you can also find in the **Lab 10: "Creating Modulator Module"**.

Chapter 10

DESIGN IMPLEMENTATION

When we have all the necessary design files for our design, we can implement targeting FPGA design. First we should create XDC constraints file where we will define placement and timing constraints for our design. Then, we should synthesize and implement our design (synthesis process is explained in the Sub-chapter **6.5 Synthesis**). After design implementation is completed successfully, we must generate bitstream file and use it to program target FPGA device.

10.1 Creating XDC File

The Vivado IDE software allows you to specify different types of constraints to help improve your design performance. Each type of constraint serves a different purpose and is recommended under different circumstances. Following are some of the most commonly used types of constraints:

- **Timing Constrains** - are typically specified globally but can also be specified for individual paths. Global constraints include period constraints for each clock, setup times for each input, and clock-to-out constraints for each output. You can enter timing constraints using the option for the timing constraints creation in the Flow Navigator. This creates a text-based Xilinx Design Constraints (XDC) file.
- **Placement Constraints** - for FPGA designs, you can specify placement constraints for each type of logic element, such as BRAMs, DSPs, LUTs, FFs, I/Os, IOBs, and global buffers. Individual logic gates, such as AND and OR gates, are mapped into CLB function generators before the constraints are read and cannot be constrained.
- **Synthesis Constraints** - Synthesis constraints instruct the synthesis tool to perform specific operations. When using "Vivado Synthesis" for synthesis, synthesis constraints control how "Vivado Synthesis" processes and implements FPGA resources, such as state machines, multiplexers, and multipliers, during the HDL synthesis and low level optimization steps. Synthesis constraints also allow control of register duplication and fanout control during global timing optimization.

Important: The Vivado IDE doesn’t support use of User Constraints File (UCF). UCF constraints are replaced with Xilinx Design Constraints (XDC). The tool supports XDC, which is based on the industry-standard Synopsys Design Constraints (SDC).

There are key differences between XDC and UCF constraints. XDC constraints are based on the standard Synopsys Design Constraints (SDC) format. SDC has been in use and evolving for more than 20 years, making it the most popular and proven format for describing design constraints.

XDC constraints are combination of:

- Industry standard SDC, and
- Xilinx propriety physical constraints

XDC constraints have the following properties:

- There are not simple strings, but are commands that follow the Tcl semantic

- They can be interpreted like any other Tcl command by the Vivado Tcl interpreter
- They are read and parsed sequentially the same as other Tcl commands

You can enter XDC constraints in several ways, at different points in the flow:

- Store the constraints in one or more XDC files
- Generate the constraints with Tcl script

There are two different ways of generating an XDC File:

- using Vivado GUI (I/O Planning view)
- using Text Editor

Creating a XDC File using the Vivado GUI (I/O Planning view):

In this step, you will be using the I/O Planning View to place the unplaced pins in the design. In order to assign pins to the FPGA, you will determine the proper pin assignments by using the *"ZedBoard Hardware User's Guide"*. This user guide contains the pin details and a reference master XDC file specifying the location and the I/O standards to be used while selecting a pin for the design.

In order to apply the constraints to the design, the design has to be synthesized at least ones. Therefore, you will start the constraints file creation by synthesizing the design and opening the synthesized design. To synthesize your design, follow the steps explained in the **Sub-chapter 6.5.2 Run Synthesis**.

To create a XDC file using the Vivado IDE GUI, do the following:

Step 1. Change the layout from the **Default Layout** to **I/O Planning** view, in the layout pull-down menu in the main toolbar, to identify pins that don't have an assigned location, see Illustration 10.1

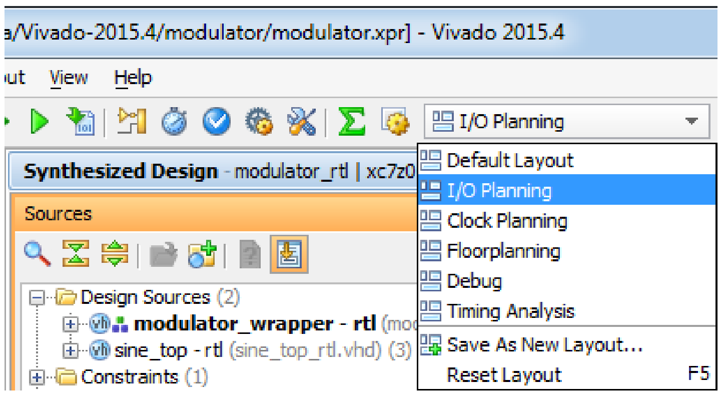


Figure 10.1: I/O Planning option

This will change the layout from the Default view to the I/O Planning view, see Illustration 10.2.

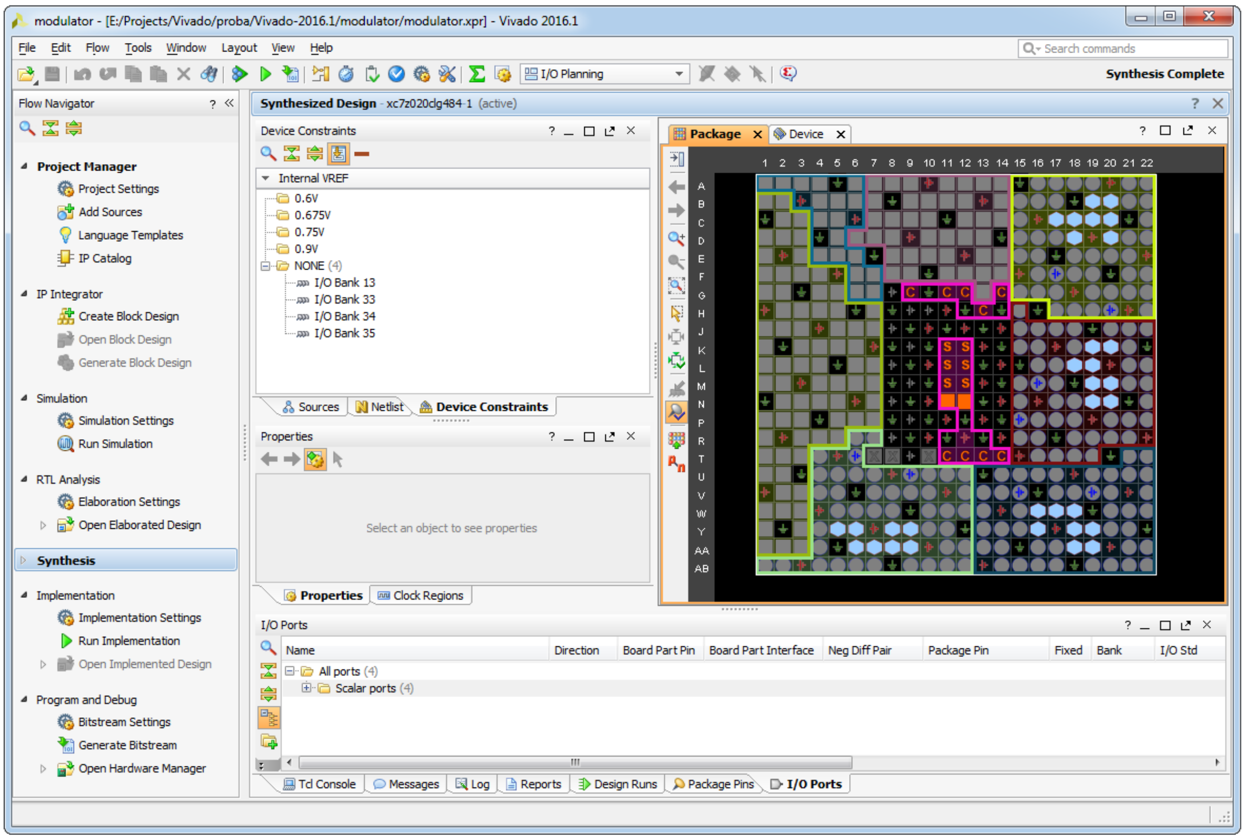
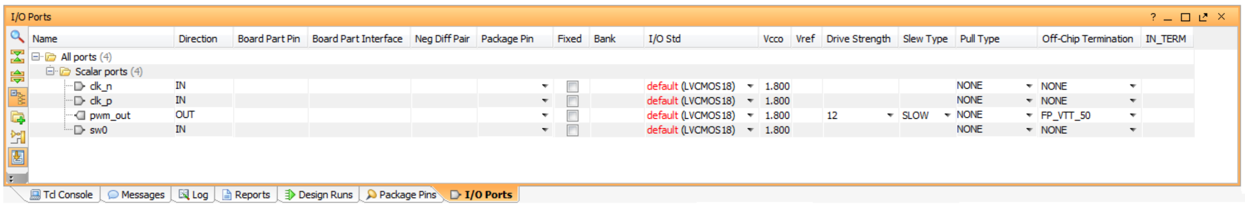


Figure 10.2: I/O Planning View

The main window of the I/O Planning view displays the package view of the ZedBoard device. Below the Package view, two additional tabs are populated. One tab displays the list of I/O ports of the design and the second tab displays the list of package pins on the device package.

Step 2. In the I/O Ports tab, click **Expand All** option, or just expand **Scalar ports**, which shows all I/O Ports of your design, see Illustration 10.3


I/O Ports

Name	Direction	Board Part Pin	Board Part Interface	Neg Diff Pair	Package Pin	Fixed	Bank	I/O Std	Vcco	Vref	Drive Strength	Slew Type	Pull Type	Off-Chip Termination	IN_TERM
All ports (4)															
Scalar ports (4)															
clk_n	IN					☐		default (LVCMOS18)	1.800				NONE	NONE	
clk_p	IN					☐		default (LVCMOS18)	1.800				NONE	NONE	
pwm_out	OUT					☐		default (LVCMOS18)	1.800		12	SLOW	NONE	FP_VTT_50	
sw0	IN					☐		default (LVCMOS18)	1.800				NONE	NONE	

Figure 10.3: I/O Ports tab

Note that none of the pins in this view have an assigned location.

Grey icons indicate unplaced ports, while yellow icons indicate placed ports. On the Illustration 10.3 we can see that all I/O ports are coloured grey, since none of them has been placed to a specific pin location. After we assign a pin location to each of the I/O ports they will be coloured yellow, as can be seen on the Illustration 10.5.

Step 3. To connect your logical with your physical ports, select one scalar port (for example **pwm_out**) and find in the user guide for the ZedBoard evaluation board to which pin location you would like to connect your pwm_out port. In our design we should connect pwm_out port with one of the LED diodes that are physically present on the ZedBoard evaluation board. If you open ZedBoard user guide you can find that the FPGA pin location of the LD0 diode is **T22** and that the I/O standard that must be used is **LVCMOS33**.

LVCMOS33 is a low voltage CMOS I/O standard using 3.3V power supply voltage. For more information about this I/-O standard, please refer to the *"JEDEC Standard JESD8C.01, Interface Standard for Nominal 3 V/3.3 V Supply Digital*

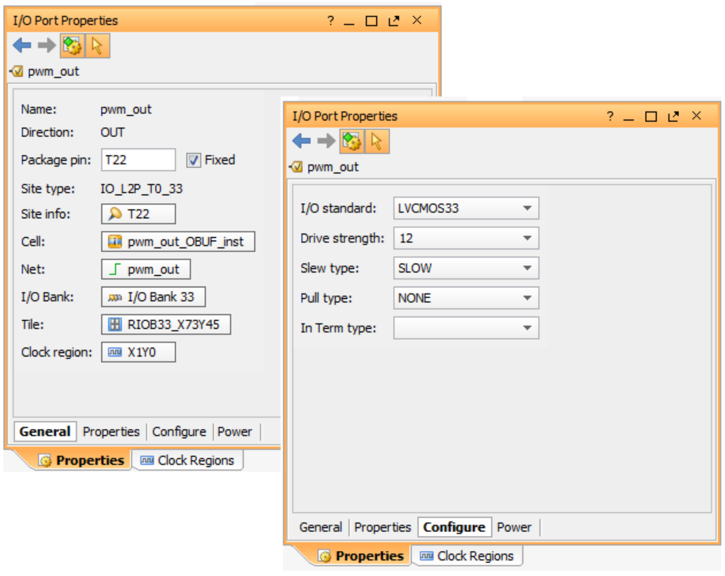
Integrated Circuits standard.

Step 4. In the **I/O Ports** tab, click on the pwm_out's **Package Pin** column and choose **T22** as a pin location to connect the pwm_out port

Step 5. Click on the pwm_out's **I/O Std** column and change the I/O standard from default LVCMOS18 to **LVCMOS33**

Step 6. Leave all the other pwm_out's options unchanged, because they are default values

Note: After assigning pin location and I/O standard for pwm_out port, we can notice that **I/O Port Properties** window popped up. This is the another way to change port properties, see Illustration 10.4. If you want to apply some changes that you made, just click the **Automatically update** button.



Name: pwm_out
Direction: OUT
Package pin: T22 ☑ Fixed
Site type: IO_L2P_T0_33
Site info: T22
Cell: pwm_out_OBUF_inst
Net: pwm_out
I/O Bank: I/O Bank 33
Tile: RIOB33_X73Y45
Clock region: X1Y0

I/O standard: LVCMOS33
Drive strength: 12
Slew type: SLOW
Pull type: NONE
In Term type:

Figure 10.4: I/O Port Properties window

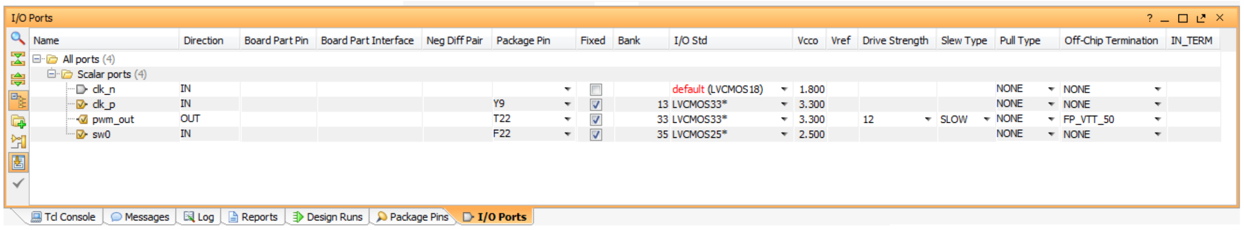
Step 7. Repeat these configuration steps for the remaining ports using the pin locations and necessary I/O standards information shown below:

- **clk_p** - pin location: Y9, I/O standard: LVCMOS33
- **sw0** - pin location: F22, I/O standard: LVCMOS25

Note: All this information has been extracted from the user guide for the ZedBoard evaluation board.

LVCMOS25 is a low voltage CMOS I/O standard using 2.5V power supply voltage. For more information about this I/O standard, please refer to the *"JEDEC Standard JESD8-5A.01, 2.5 V ± 0.2 V (Normal Range) and 1.8 V – 2.7 V (Wide Range) Power Supply Voltage and Interface Standard for Nonterminated Digital Integrated Circuits"* standard.

Note: After all modifications, **I/O Ports** tab should look like as it is shown on the Illustration 10.5.



Name	Direction	Board Part Pin	Board Part Interface	Neg Diff Pair	Package Pin	Fixed	Bank	I/O Std	Vcco	Vref	Drive Strength	Slew Type	Pull Type	Off-Chip Termination	IN_TERM
All ports (4)															
Scalar ports (4)															
clk_n	IN					☐		default (LVCMOS18)	1.800				NONE	NONE	
clk_p	IN				Y9	☑	13	LVCMOS33*	3.300				NONE	NONE	
pwm_out	OUT				T22	☑	33	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50	
sw0	IN				F22	☑	35	LVCMOS25*	2.500				NONE	NONE	

Figure 10.5: I/O Ports tab with assigned pin locations and I/O standards

Note that **clk_n** port doesn't have assigned pin location and I/O standard. This is because **clk_n** port is the differential input pair of **clk_p** port and our target ZedBoard evaluation board doesn't have differential reference clock signal.

As pins or banks are selected, the corresponding pins or banks become highlighted in the other views. This makes it easier to see that the pins assigned in each bank meet the I/O banking rules and the grouped appropriately.

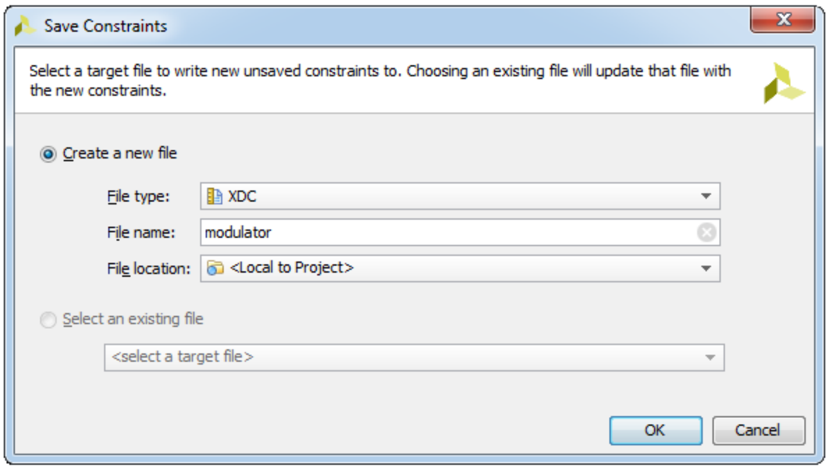
As you drag across the package view, yellow icons indicate assigned pins, grey icons indicate unassigned pins and both displayed indicates assigned I/O banks.

In the Package view you can also notice that:

- the coloured areas between the pins display the I/O banks
- the clock pins are shown as grey hexagons
- the clock-capable pins are shown as blue hexagons
- the power pins (VCC) are shown as red squares
- the ground pins (GND) are shown as green squares

Step 8. When you are finished with the placement constraints, click **File** / **Save Constraints As...**

Step 9. In the **Save Constraints** dialog box, type the name of the constraints file in the **File name** field. In our case, the name will be **modulator**, see Illustration 10.6



Save Constraints

Select a target file to write new unsaved constraints to. Choosing an existing file will update that file with the new constraints.

(●) Create a new file

File type: XDC

File name: modulator

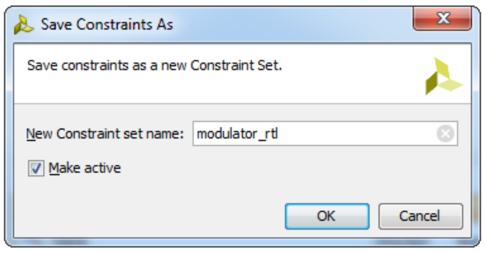
File location: <Local to Project>

( ) Select an existing file

<select a target file>

Figure 10.6: Save Constraints dialog box

Step 10. In the **Save Constraints As** dialog box, type the name of the constraint set in the **New Constraints set name** field. In our case, the name will be **modulator_rtl**, see Illustration 10.7



Save Constraints As

Save constraints as a new Constraint Set.

New Constraint set name: modulator_rtl

[✓] Make active

Figure 10.7: Save Constraints As dialog box

Step 11. Click **OK** and your **modulator_rtl** constraint set with ***modulator.xdc*** file should appear in the **Sources** window under the **Constraints**, see Illustration 10.8

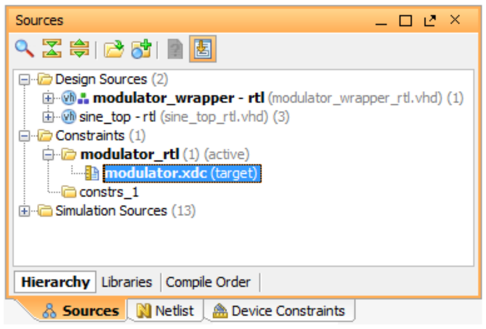
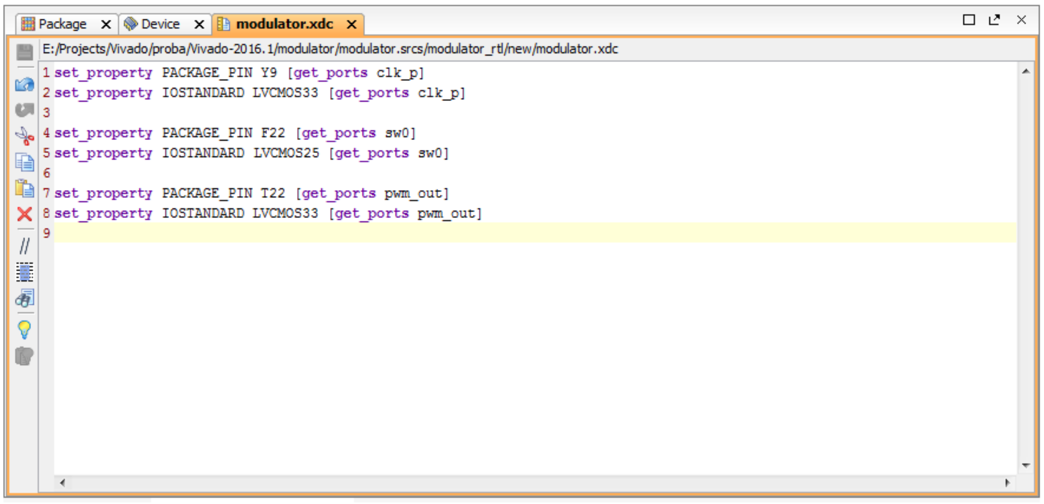


Figure 10.8: Created modulator_rtl constraints set

Step 12. Double-click on the **modulator.xdc** file to open it, see Illustration 10.9



E:/Projects/Vivado/proba/Vivado-2016.1/modulator/modulator.srcs/modulator_rtl/new/modulator.xdc

```
set_property PACKAGE_PIN Y9 [get_ports clk_p]
set_property IOSTANDARD LVCMOS33 [get_ports clk_p]

set_property PACKAGE_PIN F22 [get_ports sw0]
set_property IOSTANDARD LVCMOS25 [get_ports sw0]

set_property PACKAGE_PIN T22 [get_ports pwm_out]
set_property IOSTANDARD LVCMOS33 [get_ports pwm_out]
```

Figure 10.9: modulator.xdc file with physical constraints

In the **modulator.xdc** constraints file you can see assigned pin locations and I/O standards for each logical port of our design. For each logical port two constraints are necessary:

- First constraint connects selected logical port (by using *get_ports* Tcl command) with specified pin location (by setting the PACKAGE_PIN property, using *set_property* Tcl command).
- Second constraint sets the I/O standard that should be used for selected logical port by setting the IOSTANDARD property, using *set_property* Tcl command.

As you can see from the code above, there is a quite a lot of difference between XDC and UCF file formats. The fundamental differences between UCF and XDC files and the migrations from one format to another will be explained in detail in the **Sub-chapter 10.1.2 Migrating UCF Constraints to XDC**.

Creating a XDC File using Vivado Text Editor:

The another way to create a XDC constraints file is using Vivado text editor. The steps will be similar like in **Sub-chapter 2.4.1 Creating a Module Using Vivado Text Editor**.

Here are the steps for creating XDC file using Vivado text editor:

Step 1. **Optional:** Launch **Vivado IDE** (if it is not already launched)

Step 2. **Optional:** Open "Modulator" project (**modulator.xpr**) (if it is not already opened)

Step 3. In the main Vivado IDE menu, click **File -> New File...** option to open Vivado text editor

Step 4. In the **New File** dialog box, type the name of your constraints file (***modulator.xdc***) in the **File name** field and choose to save it into your working directory, on the same place where you saved the rest of your source files

Step 5. When you click **Save**, Vivado IDE will automatically open empty ***modulator.xdc*** source file in Vivado text editor

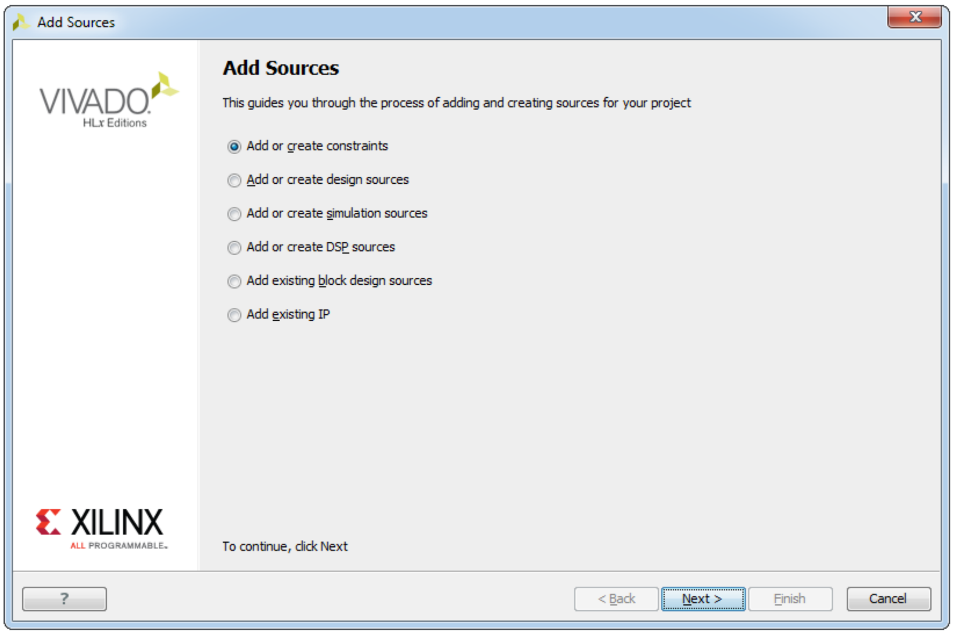
Step 6. Write the **constraints** into the opened ***modulator.xdc*** constraints file, see Illustration 10.9

Note: How to write XCD constraints file will be in detail explained in the **Sub-chapter 10.1.2 Migrating UCF Constraints to XDC**.

Step 7. When you finish with constraints file creation, click **File** -> **Save File** option from the main Vivado IDE menu, or just click Ctrl + S to save it

Step 8. In the Vivado **Flow Navigator**, click the **Add Sources** command

Step 9. In the **Add Sources** dialog box, select **Add or create constraints** option to add the constraints file to the project, see Illustration 10.10



Add Sources

This guides you through the process of adding and creating sources for your project

- [✓ Add or create constraints]
- Add or create design sources
- Add or create simulation sources
- Add or create DSP sources
- Add existing block design sources
- Add existing IP

To continue, click Next

Figure 10.10: Add Sources dialog box - Add or create constraints option

Step 10. Click **Next**

Step 11. In the **Add or Create Constraints** dialog box, click the "+" icon and select **Add Files...** option

Step 12. In the **Add Constraint Files** dialog box, browse to the project working directory and select the ***modulator.xdc*** constraints file

Step 13. Click **OK** and the ***modulator.xdc*** constraints file should appear in the **Add or Create Constraints** dialog box

Step 14. Click **Finish** and your constraints file should appear under the **Constraints** in the **Sources** view, see Illustration 10.8

10.1.1 Defining Timing Constraints

Prior to implementation, there are physical and timing constraints that need to be defined. In the previous steps we have defined physical constraints. Now, it's time to define timing constraints also.

To define timing constraints you can choose between two approaches:

- using **Constraints Wizard** , or
- using **Constraints Editor**

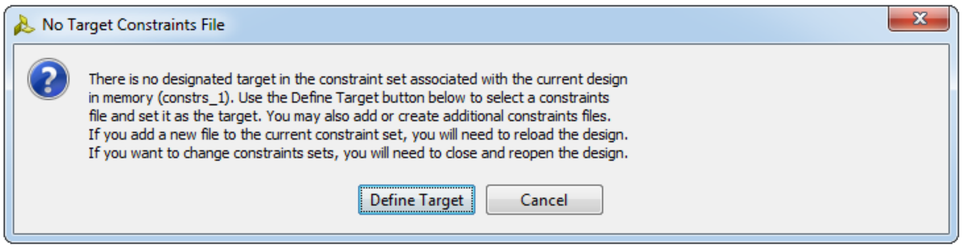
Defining timing constraints using Constraints Wizard

As we already explained, the Vivado IDE provides Timing Constraints wizard to walk you through the process of creating and validating timing constraints for the design. The Timing Constraints wizard analyzes the gate level netlist and finds missing constraints. It is only available in the synthesized and implemented designs.

To define timing constraints using Constraints Wizard, follow the next steps:

Step 1. In the Flow Navigator, under the Synthesis Design section, select first offered **Constraints Wizard** command

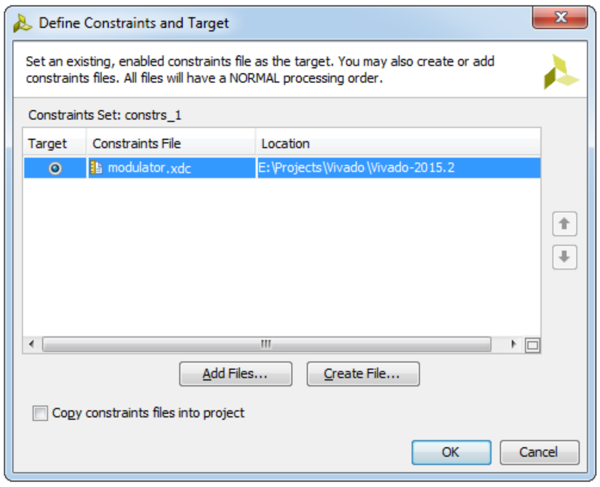
Step 2. When the **No Target Constraints File** dialog box appear, see Illustration 10.11, just click **Define Target** option to associate current design with constraints file


No Target Constraints File

There is no designated target in the constraint set associated with the current design in memory (constrs_1). Use the Define Target button below to select a constraints file and set it as the target. You may also add or create additional constraints files. If you add a new file to the current constraint set, you will need to reload the design. If you want to change constraints sets, you will need to close and reopen the design.

Figure 10.11: No Target Constraints File dialog box

Step 3. In the **Define Constraints and Target** dialog box, select ***modulator.xdc*** file as target constraints file and click **OK**, see Illustration 10.12. In the Define Constraints and Target dialog box, you can also create new or add existing constraints file.


Define Constraints and Target

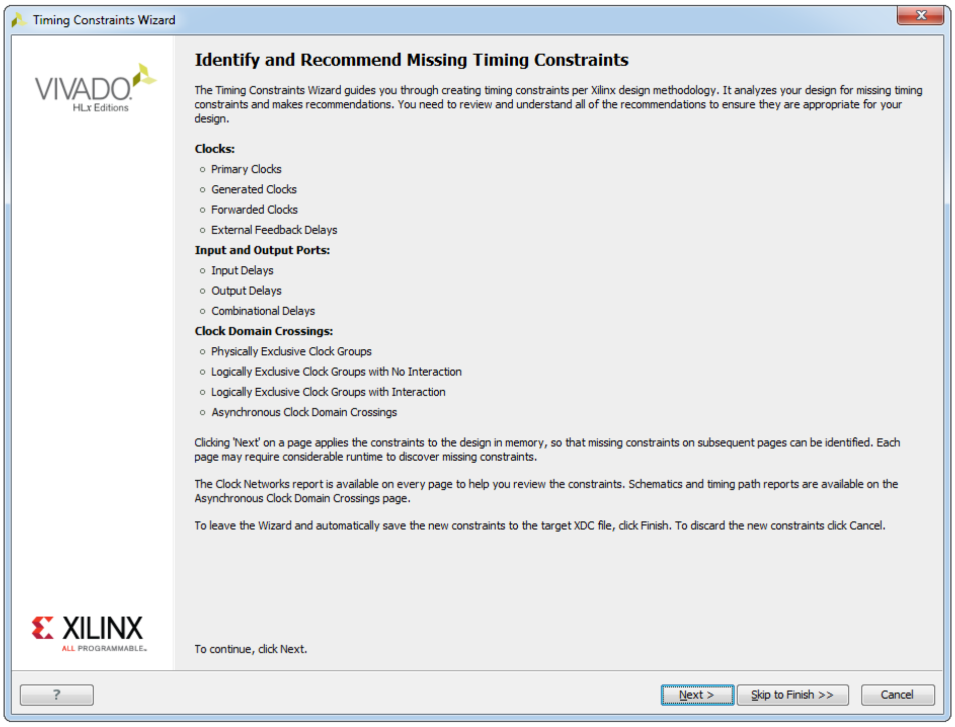
Set an existing, enabled constraints file as the target. You may also create or add constraints files. All files will have a NORMAL processing order.

Constraints Set: constrs_1

Target	Constraints File	Location
●	modulator.xdc	E:\Projects\Vivado\Vivado-2015.2

Figure 10.12: Define Constraints and Target dialog box

Step 4. In the Flow Navigator, click ones more **Constraints Wizard** command to open the introduction page. This page describes the types of constraints that the wizard will create: Clocks, Input and Output Ports, and Clock Domain Crossings, see Illustration 10.13. After reading the page, click **Next** to continue.



Identify and Recommend Missing Timing Constraints

The Timing Constraints Wizard guides you through creating timing constraints per Xilinx design methodology. It analyzes your design for missing timing constraints and makes recommendations. You need to review and understand all of the recommendations to ensure they are appropriate for your design.

Clocks:
- Primary Clocks
- Generated Clocks
- Forwarded Clocks
- External Feedback Delays

Input and Output Ports:
- Input Delays
- Output Delays
- Combinational Delays

Clock Domain Crossings:
- Physically Exclusive Clock Groups
- Logically Exclusive Clock Groups with No Interaction
- Logically Exclusive Clock Groups with Interaction
- Asynchronous Clock Domain Crossings

Clicking 'Next' on a page applies the constraints to the design in memory, so that missing constraints on subsequent pages can be identified. Each page may require considerable runtime to discover missing constraints.

The Clock Networks report is available on every page to help you review the constraints. Schematics and timing path reports are available on the Asynchronous Clock Domain Crossings page.

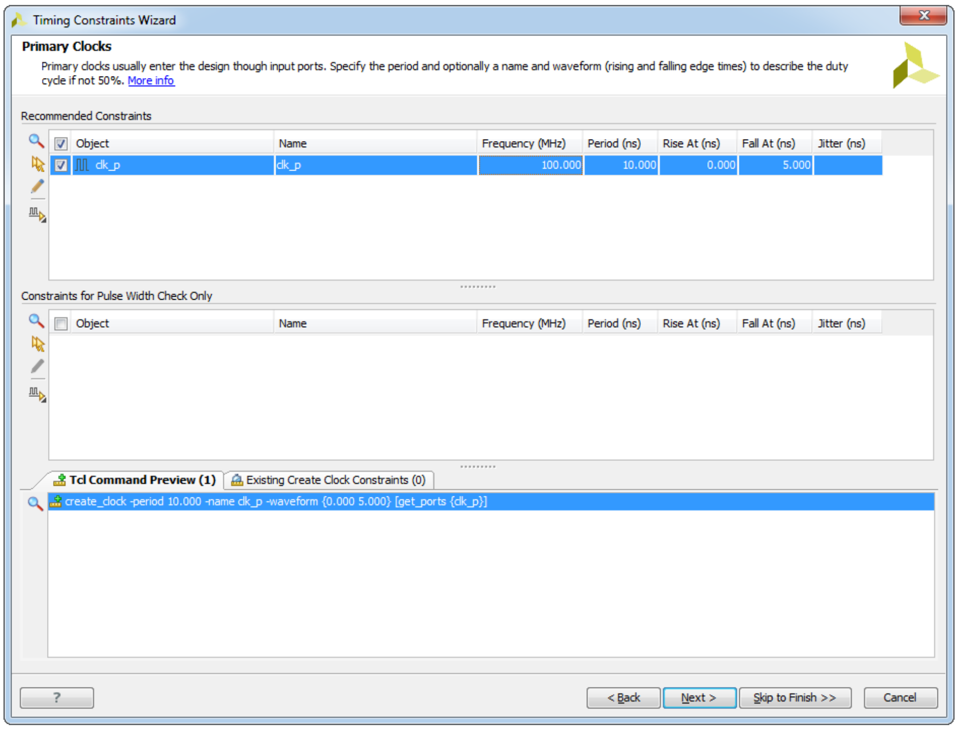
To leave the Wizard and automatically save the new constraints to the target XDC file, click Finish. To discard the new constraints click Cancel.

To continue, click Next.

Figure 10.13: Identify and Recommend Missing Timing Constraints dialog box

Step 5. In the **Primary Clocks** dialog box, Timing Constraints Wizard will display all the clock sources with a missing clock definition. Specify **100 MHz** frequency for the **clk_p** clock and wizard will automatically calculate values for Period (ns), Rise At (ns), Fall At (ns) and Jitter (ns) fields, see Illustration 10.14. Click **Next** to continue.

Each row of the wizard is a missing constraint. If you would prefer not to enter the constraint, you can uncheck the box next to the constraint. If you would like more information about how the wizard finds these missing constraints, there is a **Reference** button in the lower left-hand corner of the wizard. The reference pages are context specific and contain more information about the topologies the wizard is looking for and an explanation as to why the constraint is being suggested.



Timing Constraints Wizard

Primary Clocks

Primary clocks usually enter the design though input ports. Specify the period and optionally a name and waveform (rising and falling edge times) to describe the duty cycle if not 50%. More info

Recommended Constraints

✓ Object	Name	Frequency (MHz)	Period (ns)	Rise At (ns)	Fall At (ns)	Jitter (ns)
✓ clk_p	clk_p	100.000	10.000	0.000	5.000	

Constraints for Pulse Width Check Only

Object	Name	Frequency (MHz)	Period (ns)	Rise At (ns)	Fall At (ns)	Jitter (ns)

Tcl Command Preview (1) | Existing Create Clock Constraints (0)

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

Figure 10.14: Primary Clocks dialog box

Step 6. The primary clock constraints have been added to the design. Next, the wizard looks for unconstrained generated clocks. Generated clocks are derived from primary clocks in the FPGA fabric. In our design, the wizard determined that there are no unconstrained generated clocks. In the **Generated Clocks** dialog box, click **Next** to continue.

Step 7. Next, the wizard looks for forwarded clocks. A forwarded clock is a generated clock on a primary output port of the FPGA. These are commonly used for source synchronous buses when the capture clock travels with the data. The wizard has also determined that there are no unconstrained forwarded clocks in our design. In the **Forwarded Clocks** dialog box, click **Next** to continue.

Step 8. Next, the wizard looks for external feedback delays. MMCM or PLL feedback delay outside the FPGA is used to compute the clock delay compensation in the timing reports. The wizard did not find any unconstrained MMCM external feedback delay in our design. In the **External Feedback Delays** dialog box, click **Next** to continue.

Step 9. Next, the wizard looks at the input delays. Illustration 10.15 shows the **Input Delays** page of the Timing Constraints wizard. There are three sections on the page.

- First section shows all the input ports that are missing input delay constraints in the design. In this table you select the timing template you would like to use to constraints the input.

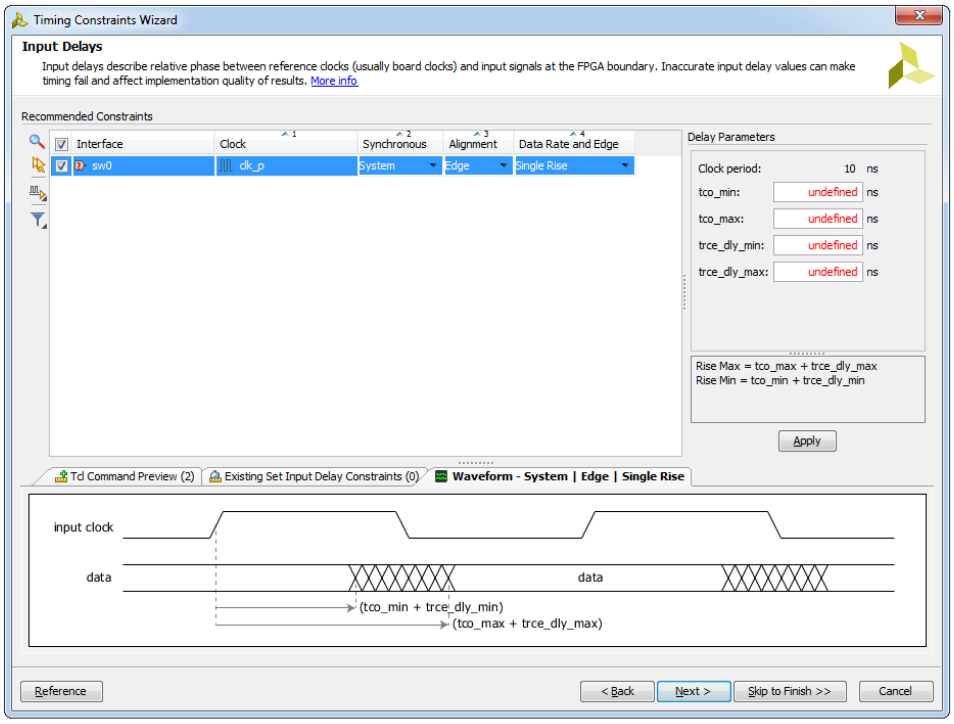
- In the second section you provide the delay values for the template. This section will change depending on the template chosen in the first section.

- In the third section there are three tabs:

  - **Tcl Command Preview** - previews the Tcl commands that will be used to constrain the design

  - **Existing Set Input Delay Constraints** - shows input delay constraints that exist in the design

  - **Waveform** - displays the waveform associated with the template



Timing Constraints Wizard

Input Delays

Input delays describe relative phase between reference clocks (usually board clocks) and input signals at the FPGA boundary. Inaccurate input delay values can make timing fail and affect implementation quality of results. More info

Recommended Constraints

	Interface	Clock	Synchronous	Alignment	Data Rate and Edge
☑	sw0	clk_p	System	Edge	Single Rise

Delay Parameters

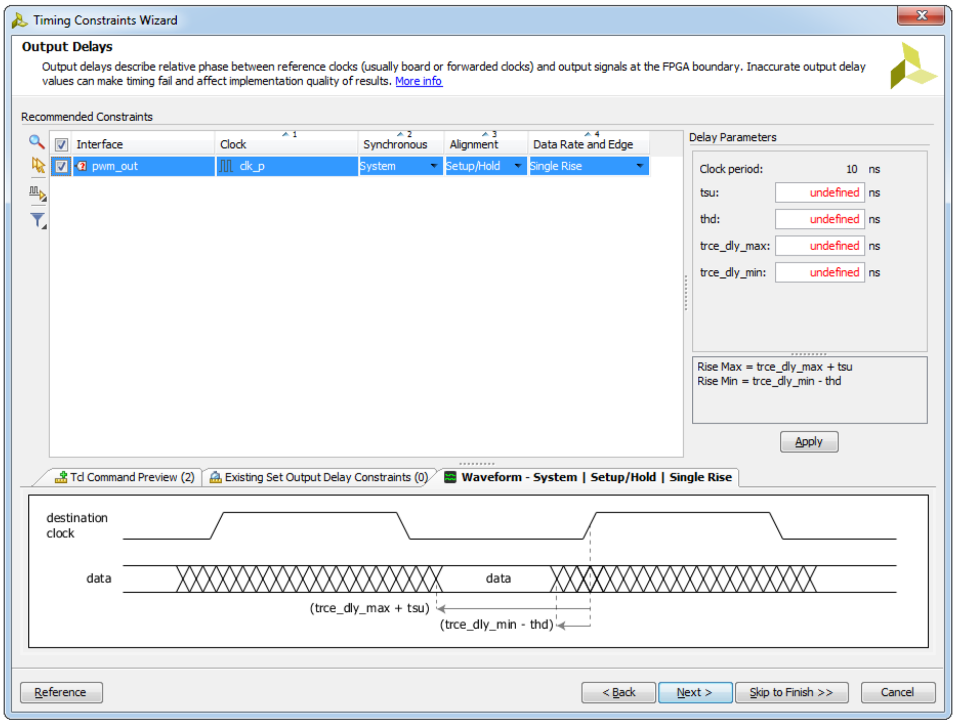
Clock period:	10	ns
tco_min:	undefined	ns
tco_max:	undefined	ns
trce_dly_min:	undefined	ns
trce_dly_max:	undefined	ns

Rise Max = tco_max + trce_dly_max
Rise Min = tco_min + trce_dly_min

Figure 10.15: Input Delays dialog box

Step 10. Uncheck the ***sw0*** input port in the first section of the Input Delays dialog box, because we don't need delay period for this input port. When you successfully finished with all input constraint values, click **Next**

Step 11. Next, the **Output Delays** page of the wizard displays all the outputs that are unconstrained in the design, see Illustration 10.16. The page layout is very similar to the inputs page. Uncheck the ***pwm_out*** output port in the first section of the Output Delays dialog box, because we don't need delay period for this output port also. When you successfully finished with all output constraint values, click **Next**



Timing Constraints Wizard

Output Delays

Output delays describe relative phase between reference clocks (usually board or forwarded clocks) and output signals at the FPGA boundary. Inaccurate output delay values can make timing fail and affect implementation quality of results. More info

Recommended Constraints

	Interface	Clock	Synchronous	Alignment	Data Rate and Edge
☑	pwm_out	clk_p	System	Setup/Hold	Single Rise

Delay Parameters

Clock period:	10	ns
tsu:	undefined	ns
thd:	undefined	ns
trce_dly_max:	undefined	ns
trce_dly_min:	undefined	ns

Rise Max = trce_dly_max + tsu
Rise Min = trce_dly_min - thd

Figure 10.16: Output Delays dialog box

Step 12. The wizard now looks for any unconstrained combinational paths through the design. A combinational path is a path that traverses the FPGA without being captured by any sequential elements. Our design doesn't contain any combinational paths. In the **Combinational Delays** dialog box, click **Next** to continue.

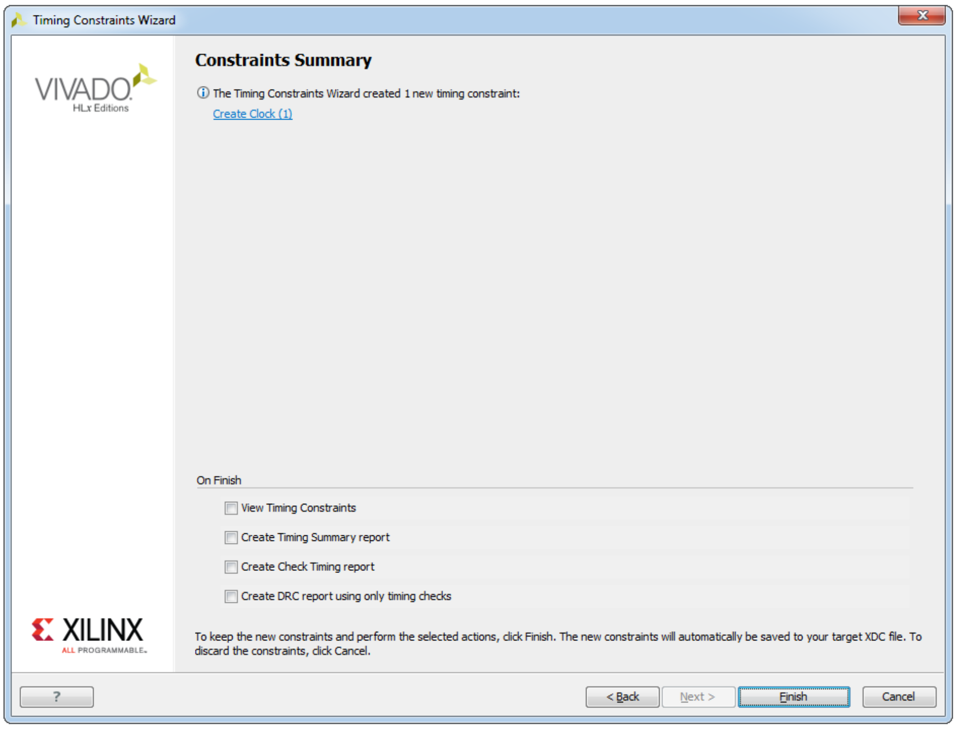
Step 13. Physically exclusive clock groups are clocks that do not exit in the design at the same time. There are no unconstrained physically exclusive clock groups in our design. In the **Physically Exclusive Clock Groups** dialog box, click **Next** to continue.

Step 14. Logically exclusive clocks with no interaction are clocks that are active at the same time except on shared clock tree sections. Then these clocks do not have logical paths between each other and outside the shared sections, they are logically exclusive. There are no unconstrained logically exclusive clock groups with no interaction in our design. In the **Logically Exclusive Clock Groups with No Interaction** dialog box, click **Next** to continue.

Step 15. Logically exclusive clocks with interaction are clocks that are active at the same time except on shared clock tree sections. When these clocks have logical paths between each other, only the clocks limited to the shared clock tree sections are logically exclusive and are therefore constrained differently than the logically exclusive clock with no interaction. There are no unconstrained logically exclusive clock groups with interaction in our design. In the **Logically Exclusive Clock Groups with Interaction** dialog box, click **Next** to continue.

Step 16. The **Asynchronous Clock Domain Crossings** page recommends constraints for safe clock domain crossings. Our design does not contain any unconstrained clock domain crossings. Click **Next** to continue.

Step 17. The **Constraints Summary** page is the final page of the Timing Constraints wizard, see Illustration 10.17. All the constraints that were generated by the wizard can be viewed by clicking the links. If you would like to run any reports once the wizard is finished, you can select them using the check boxes in the wizard. Click **Finish** to complete the Timing Constraints wizard.



Constraints Summary

The Timing Constraints Wizard created 1 new timing constraint:

Create Clock (1)

On Finish

- [ ] View Timing Constraints
- [ ] Create Timing Summary report
- [ ] Create Check Timing report
- [ ] Create DRC report using only timing checks

To keep the new constraints and perform the selected actions, click Finish. The new constraints will automatically be saved to your target XDC file. To discard the constraints, click Cancel.

Figure 10.17: Constraints Summary dialog box

Defining timing constraints using Constraints Editor

To define timing constraints using Constraints Editor, follow the next steps:

Step 1. Select **Window -> Timing Constraints** option from the main Vivado IDE menu to open the Timing Constraints window, see Illustration 10.18, or

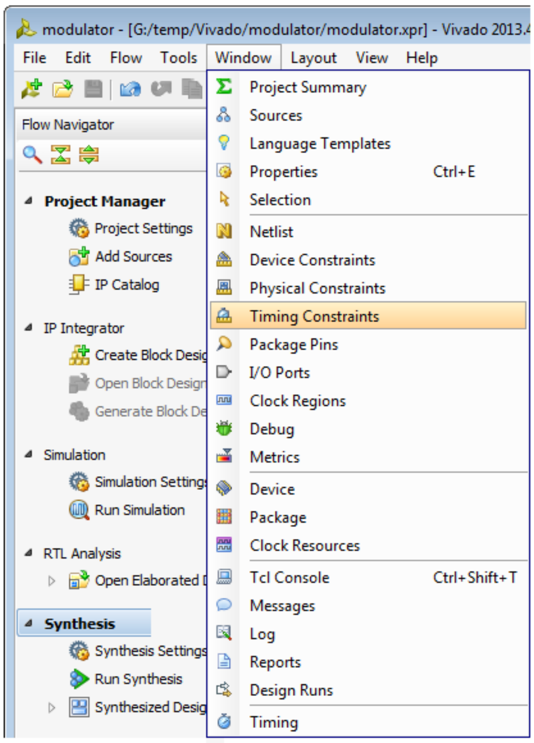


Figure 10.18: Timing Constraints option

select in the Flow Navigator, under the Synthesis Design section, second offered **Edit Timing Constraints** command

The **Timing Constraints** window appears in the main window of the Vivado IDE, see Illustration 10.19

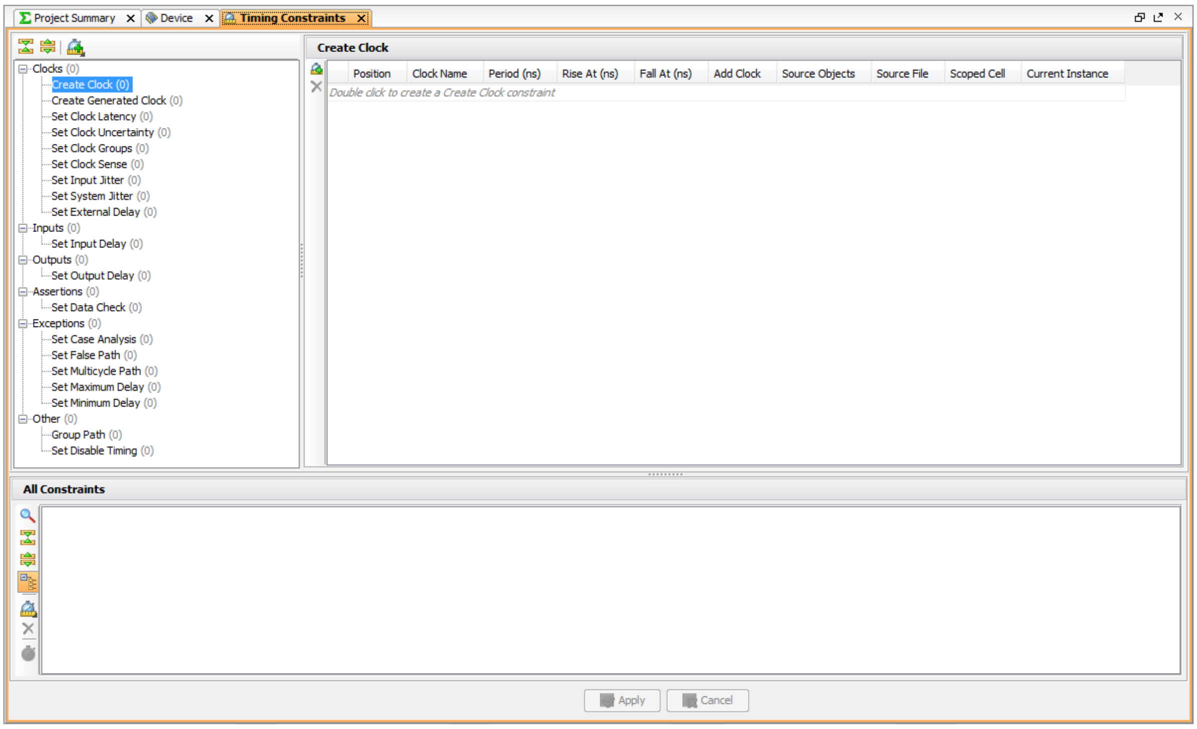


Figure 10.19: Timing Constraints window

There are three sections in the Timing Constraints window:

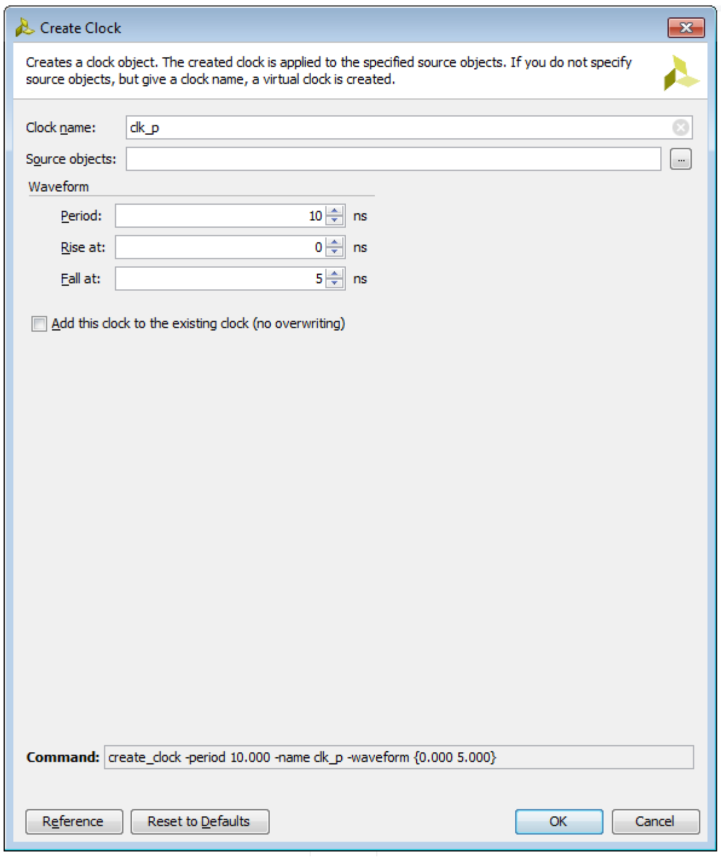
- **Constraints tree view** - displays standard timing constraints, grouped by category. Double-clicking a constraint in this section opens a new window to help you define the selected constraint.
- **Constraints Spreadsheet** - displays timing constraints of the type currently selected in the Constraints tree view. If you prefer, you can use this to directly define or edit constraints instead of using Constraints wizard.
- **All Constraints** - displays all the timing constraints that currently exist in the design

The Timing Constraints wizard identifies missing clocks, I/O delays, and clock domain crossings exceptions, but it doesn't handle general timing exceptions. We will use the timing constraints editor to create the exceptions that exist in the design.

Define the primary clock constraint by creating a clock object with a specified period. The modulator design has a 100 MHz clock supplied through differential clock input ports on the FPGA. First define the primary clock object for the design and then define a PERIOD constraint for the clock object.

Step 2. In the **Constraints tree view** window of the Timing Constraint editor, double-click on the **Create Clock (0)** option under the Clocks (0) section to create a clock constraint

Step 3. In the **Create Clock** dialog box, enter *clock_name* (**clk_p**) in the **Clock Name** field, see Illustration 10.20


Create Clock

Creates a clock object. The created clock is applied to the specified source objects. If you do not specify source objects, but give a clock name, a virtual clock is created.

Clock name: clk_p

Source objects:

Waveform

Period: 10 ns

Rise at: 0 ns

Fall at: 5 ns

Add this clock to the existing clock (no overwriting)

Command: create_clock -period 10.000 -name clk_p -waveform {0.000 5.000}

Figure 10.20: Create Clock dialog box

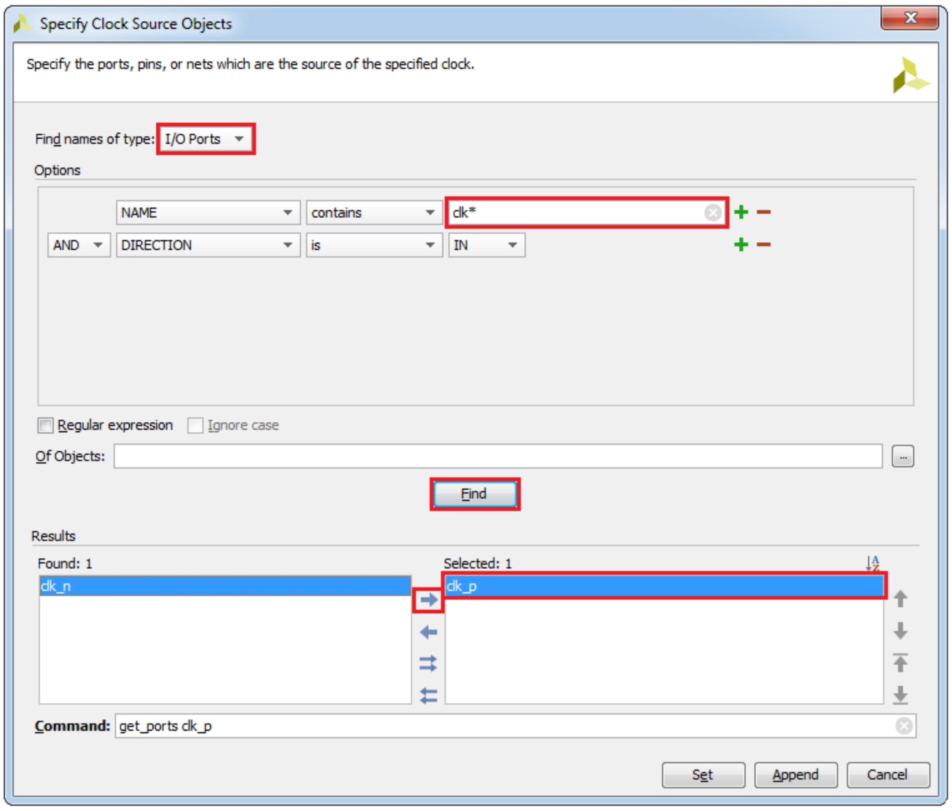
Step 4. Click the icon next to the **Source objects** field and **Specify Clock Source Objects** dialog box will appear, see Illustration 10.20

Note: This step is important to associate the clock input port to the clock definition.

Step 5. In the **Specify Clock Source Objects** dialog box (see Illustration 10.21), do the following:

- Ensure that **I/O Ports** is selected from the **Find names of type** drop-down list
- Enter **clk** in the empty search field
- Click **Find**
- In the **Find results: 2** section, select **clk_p**

- Click the -> icon to select **clk_p**
- Click **Set**


Specify Clock Source Objects
Specify the ports, pins, or nets which are the source of the specified clock.
Options
Regular expression
Ignore case
Results
Find
Set
Append
Cancel

Find names of type: I/O Ports

	NAME	contains	clk*
AND	DIRECTION	is	IN

Of Objects:

Found: 1
clk_n

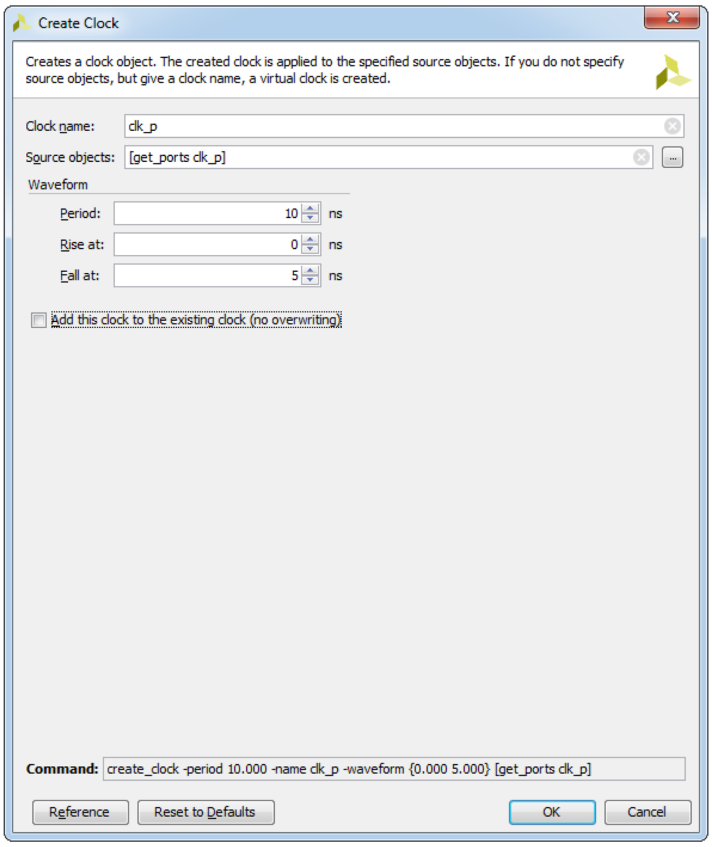
Selected: 1
clk_p

Command: get_ports clk_p

Figure 10.21: Specify Clock Source Objects dialog box

Step 6. In the **Create Clock** dialog box, specify the period by setting the period property of the clock. In this step, you will describe the period property and review the waveform details of the clock objects, see Illustration 10.22:

- Enter **10 ns** in the **Period** field in the Waveform section, because 10 ns is the period of the 100 MHz input clock signal
- Ensure that the **Rise at** and **Fall at** fields are set to **0** and **5** respectively, which means that the duty cycle of the input clock signal will be 50%.
- Click **OK** to create the clock constraint



Create Clock

Creates a clock object. The created clock is applied to the specified source objects. If you do not specify source objects, but give a clock name, a virtual clock is created.

Clock name: clk_p

Source objects: [get_ports clk_p]

Waveform

Period: 10 ns

Rise at: 0 ns

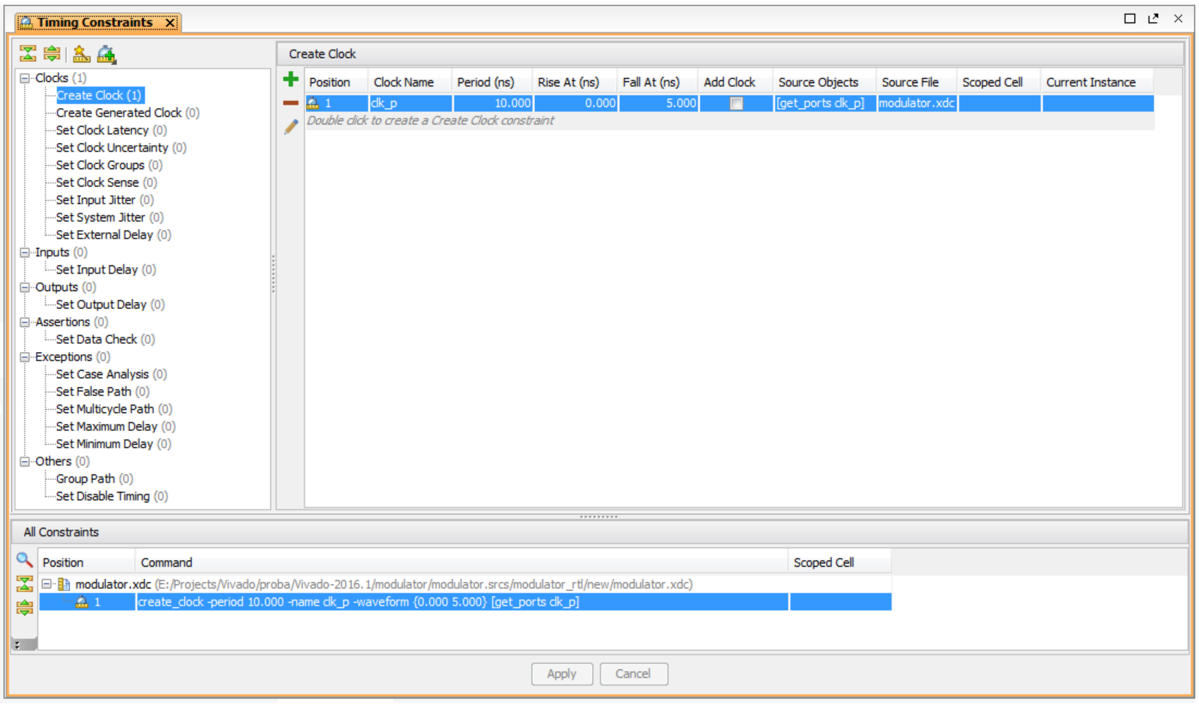
Fall at: 5 ns

☐ Add this clock to the existing clock (no overwriting)

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

Figure 10.22: Create Clock dialog box after specifying the period for the clk_p

The Timing Constraints window now displays the timing constraint applied to the design, see Illustration 10.23



- Clocks (1)
  - Create Clock (1)
  - Create Generated Clock (0)
  - Set Clock Latency (0)
  - Set Clock Uncertainty (0)
  - Set Clock Groups (0)
  - Set Clock Sense (0)
  - Set Input Jitter (0)
  - Set System Jitter (0)
  - Set External Delay (0)
- Inputs (0)
  - Set Input Delay (0)
- Outputs (0)
  - Set Output Delay (0)
- Assertions (0)
  - Set Data Check (0)
- Exceptions (0)
  - Set Case Analysis (0)
  - Set False Path (0)
  - Set Multicycle Path (0)
  - Set Maximum Delay (0)
  - Set Minimum Delay (0)
- Others (0)
  - Group Path (0)
  - Set Disable Timing (0)

Create Clock

Position	Clock Name	Period (ns)	Rise At (ns)	Fall At (ns)	Add Clock	Source Objects	Source File	Scoped Cell	Current Instance
1	clk_p	10.000	0.000	5.000	☐	[get_ports clk_p]	modulator.xdc		

Double click to create a Create Clock constraint

All Constraints

Position	Command	Scoped Cell
modulator.xdc (E:/Projects/Vivado/proba/Vivado-2016.1/modulator/modulator.srcs/modulator_rtl/new/modulator.xdc)		
1	create_clock -period 10.000 -name clk_p -waveform {0.000 5.000} [get_ports clk_p]	

Figure 10.23: Timing Constraints window with the create_clock constraint

Notice that the create_clock XDC command for the created clock is also displayed in the All Constraints view of the Timing Constraints window.

All the timing constraints that have been run are applied to the design that is loaded in the memory. The applied constraints can be saved by writing them to the XDC file. All the timing constraints applied to the design are available in the **All Constraints** view of the Timing Constraints window, see Illustration 10.23.

Step 7. To save your timing constraints to the **modulator.xdc** constraints file, select **File -> Save Constraints** command from the main menu

If you want to verify that the timing constraints have been applied to the **modulator.xdc** file, do the following:

1. If the **modulator.xdc** file is already open, click the **Reload** link in the banner of the *modulator.xdc* tab to reload the constraints file from disk.

2. If the **modulator.xdc** file is not open, select the **Sources** window, **Hierarchy** view

3. Expand **Constraints** folder

4. Double-click on the **modulator.xdc** file, under the **modulator_rtl**, to open the file and you should see that your timing constraints were saved to the XDC file, see Illustrations 10.24 and 10.25

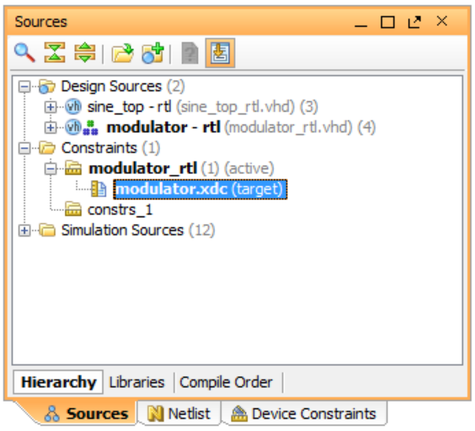
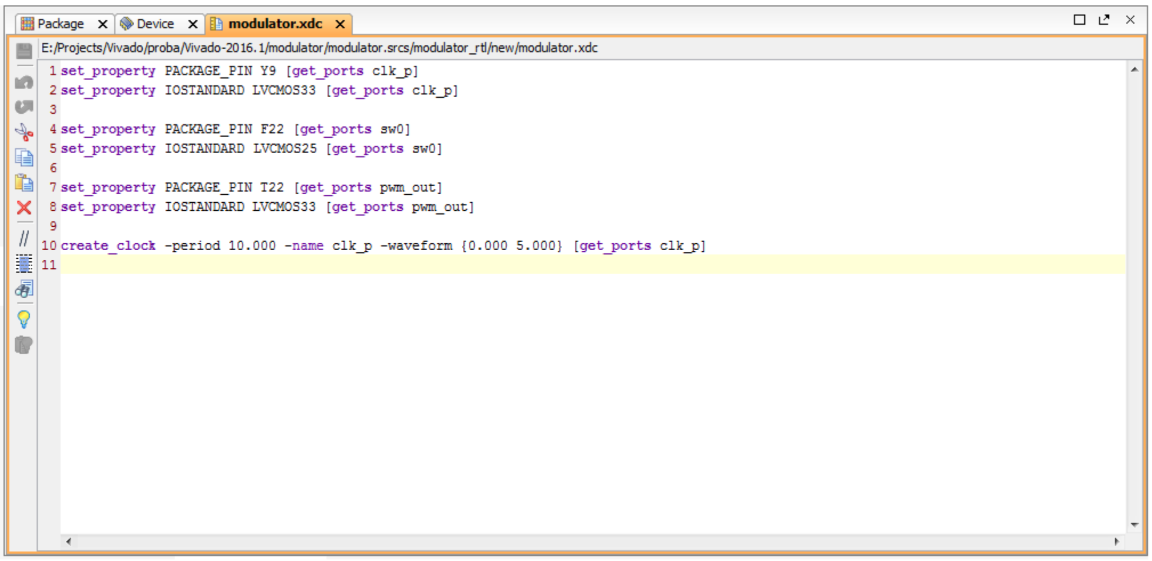


Figure 10.24: modulator.xdc constraints file in the Sources window



E:/Projects/Vivado/proba/Vivado-2016.1/modulator/modulator.srcs/modulator_rtl/new/modulator.xdc

```
set_property PACKAGE_PIN Y9 [get_ports clk_p]
set_property IOSTANDARD LVCMOS33 [get_ports clk_p]

set_property PACKAGE_PIN F22 [get_ports sw0]
set_property IOSTANDARD LVCMOS25 [get_ports sw0]

set_property PACKAGE_PIN T22 [get_ports pwm_out]
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]
```

Figure 10.25: modulator.xdc file with physical and timing constraints

In the **modulator.xdc** file you will see four blocks of commands, see Illustration 10.25. First three blocks (first six lines) are the Physical Constraints and the last line is the Timing Constraint.

10.1.2 Migrating UCF Constraints to XDC

As we already said, the Vivado IDE doesn't support the UCF constraints used in the ISE Design Suite. You must migrate the design with UCF constraints to the XDC format.

If you are familiar with the UCF file, it won't be hard for you to understand how to convert existing UCF file to XDC as a starting point for creating XDC constraints.

As with UCF, XDC consists of:

- Timing constraints
- Physical constraints

The fundamental differences between XDC and UCF constraints are:

- XDC is sequential language, with clear precedence rules
- UCF constraints are typically applied to nets, for which XDC constraints are typically applied to pins, ports, and cell objects
- UCF PERIOD constraints and XDC *create_clock* command are not always equivalent and can lead to different timing results
- UCF by default doesn't time between asynchronous clock groups, while in XDC, all clocks are considered related and timed unless otherwise constrained (*set_clock_groups*)
- In XDC, multiple clocks can exist on the same object

The Table 10.1 shows the main mapping between UCF constraints to XDC commands.

Table 10.1 UCF to XDC Mapping

UCF	XDC
TIMESPEC PERIOD	*create_clock*, *create_generated_clock*
OFFSET = IN <x> BEFORE <clk>	*set_input_delay*
OFFSET = OUT <x> BEFORE <clk>	*set_output_delay*
FROM:TO "TS _"*2*	*set_multicycle_path*
FROM:TO	*set_max_delay*
TIG	*set_false_path*
NET "clk_p" LOC = AD12	set_property LOC AD12 [get_ports clk_p]
NET "clk_p" IOSTANDARD = LVDS	set_property IOSTANDARD LVDS [get_ports clk_p]

According to the Table 10.1, our UCF file will migrate to the XDC in the following way:

ucf constraints:

```
NET "clk_p"   LOC = "Y9"  | IOSTANDARD = LVCMOS33;
NET "sw0"     LOC = "F22" | IOSTANDARD = LVCMOS25;
NET "pwm_out" LOC = "T22" | IOSTANDARD = LVCMOS33 | SLEW = SLOW | DRIVE = 12;

NET "clk_p" TNM_NET = clk_p;
TIMESPEC TS_clk_p = PERIOD "clk_in" 10 ns HIGH 50%;
```

xdc constraints:

```
set_property PACKAGE_PIN Y9 [get_ports clk_p]
set_property PACKAGE_PIN F22 [get_ports sw0]
set_property PACKAGE_PIN 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: Information about the types of constraints, how to generate XDC constraints file, differences between UCF and XDC constraints and how to migrate from UCF to XDC constraints file, you can also find in the **Lab 11: "Creating XDC File"**.

10.2 Implementation

10.2.1 About the Vivado Implementation Process

The Vivado Design Suite enables implementation of UltraScale FPGA and Xilinx 7 Series FPGA designs from the variety of design sources, including RTL designs, netlist designs and IP centric design flows.

Vivado implementation process includes all steps necessary to place and route the netlist onto the FPGA device resources, while meeting the design's logical, physical, and timing constraints.

The Vivado implementation is a timing-driven flow. It supports industry standard Synopsys Design Constraints (SDC) commands to specify design requirements and restrictions, as well as additional commands in the Xilinx Design Constraints (XDC) format.

The Vivado implementation process includes logical and physical transformations of the design. The implementation process consists of the following sub-processes:

- **Opt Design: Netlist Optimization**

  Optimizes the logical design to make it easier to fit onto the target Xilinx device:

  - Ensures optimal netlist for placement
  - Optional in non-project batch flow (but recommended)
  - Automatically enables in the project-based flow

  Because this is the first view of the assembled design (RTL and IP blocks), the design can usually be further optimized. The *opt_design* command is the next step and performs logic trimming, removing cells with no loads, propagating constant inputs, and combining LUTs for example LUTs in series that can be combined into fewer LUTs.

- **Power Opt Design: Power Optimization**

  Optimizes design elements to reduce the power demands of the target Xilinx device:

  - Disabled in project-based flow (can be set with implementation settings in GUI)
  - Power optimization includes a fine-grained clock gating solution that can reduce dynamic power by up to 30%
  - Intelligent clock gating optimizations are automatically performed on the entire design and will generate no changes to the existing logic or clocks
  - Algorithm performs analysis on all portions of the design

  Note: This step is optional.

- **Place Design: Placer**

  Places the design onto the target Xilinx device:

  - Project-based flow (included in implementation stage)
  - Non-project batch flow (place_design)
  - Can use an input XDEF as a starting point for placement

- **Phys Opt Design: Physical Synthesis**

  Optimizes design timing by replicating drivers of high-fanout nets to distribute the loads:

  - Post-placement timing-driven optimization (replicates and places drivers of high fanout nets with negative slack)
  - More features coming in future releases (register retiming)
  - Available in all flows and can be de-activated in the GUI
  - phys_opt_design (run between place_design and route_design)

  Note: This step is optional.

- **Route Design: Router**

  Routes the design onto the target Xilinx device:

  - Project-based flow (included in implementation stage)
  - Non-project batch flow (route_design)

  - Router reporting (report_route_status command)
  - Check route status of individual nets

The Vivado Design Suite includes a variety of design flows, and supports an array of design sources. In order to generate a bitstream that can be downloaded onto the FPGA device, the design must pass through implementation process.

Implementation is a series of steps that takes the logical netlist and maps it into the physical array of the target Xilinx device. These steps include:

- Logic optimization
- Placement of logic cells
- Routing of connections between cells

10.2.2 Run Implementation

Now we will run implementation process from the Flow Navigator, which will trigger synthesis followed by implementation in one step.

To run the implementation process, please do the following:

Step 1. In the **Flow Navigator**, click **Run Implementation** command and wait for implementation to be completed, see Illustration 10.26

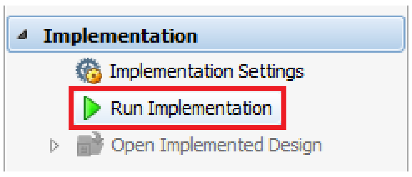
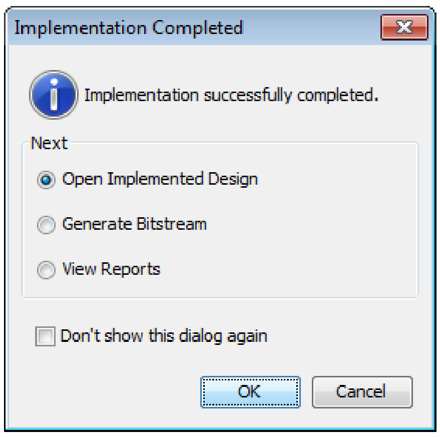


Figure 10.26: Run Implementation command

Note: You can monitor the Implementation progress in the bar in the upperright corner of the Vivado IDE.

Step 2. After the implementation is completed, the **Implementation Completed** dialog box will appear, see Illustration 10.27



Implementation Completed

Implementation successfully completed.

Next

(●) Open Implemented Design

( ) Generate Bitstream

( ) View Reports

[ ] Don't show this dialog again

Figure 10.27: Implementation Completed dialog box

Step 3. Select **Open Implementation Design** option in the **Implementation Completed** dialog box and click **OK** to open the implemented design

10.2.3 After Implementation

After implementation process:

- Sources and Netlist tabs do not change. Now as each resource is selected, it will show the exact placement of the resource on the die (Instance Properties view will show specific details about the resource).
- Timing results have to be generated with the Report Timing Summary
- As each path is selected, the placement of the logic and its connections is shown in the Device view. This is the cross-probing feature that helps with static timing analysis.

After you have implemented the design (or opened a project that only contains an implemented design), the Flow Navigator changes again, see Illustration 10.28. Flow Navigator is optimized to provide quick access to the options most frequently used after implementation (note that most of these reports are the same, except with true-timing information):

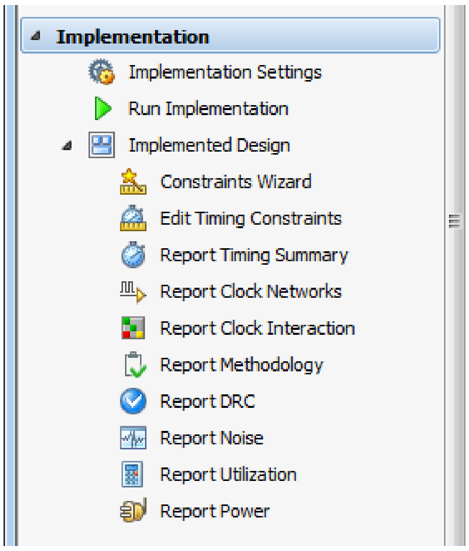
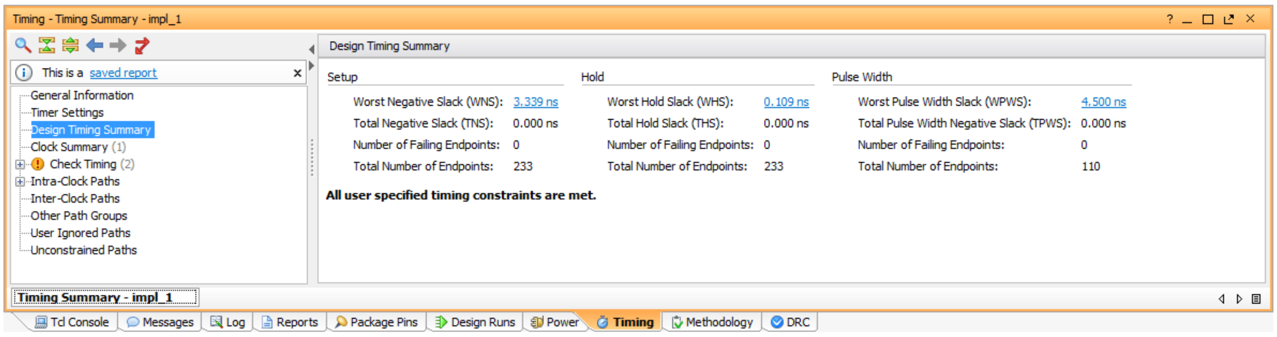


Figure 10.28: Implemented Design options

- ***Constraints Wizard***: Open the Timing Constraints wizard
- ***Edit Timing Constraints***: Open the Constraints viewer
- ***Report Timing Summary***: Generates a default timing report (using true timing information)
- ***Report Clock Networks***: Generates a clock tree for the design
- ***Report Clock Interaction***: Verifies constraint coverage on paths between clock domains
- ***Report Methodology***: Performs automated methodology checks and allows you to find design issues early in the design process
- ***Report DRC***: Performs design rule check on the entire design
- ***Report Noise***: Performs an SSO analysis of output and bidirectional pins in your design
- ***Report Utilization***: Generates a graphic version of the Utilization Report
- ***Report Power***: Provides detailed power analysis reports

Note that the Report Timing Summary is the most important default report because at this point what most designers are concerned about is meeting their timing objectives and only after completing an implementation does the designer know if they can actually do that.



Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	3.339 ns	Worst Hold Slack (WHS):	0.109 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	233	Total Number of Endpoints:	233	Total Number of Endpoints:	110

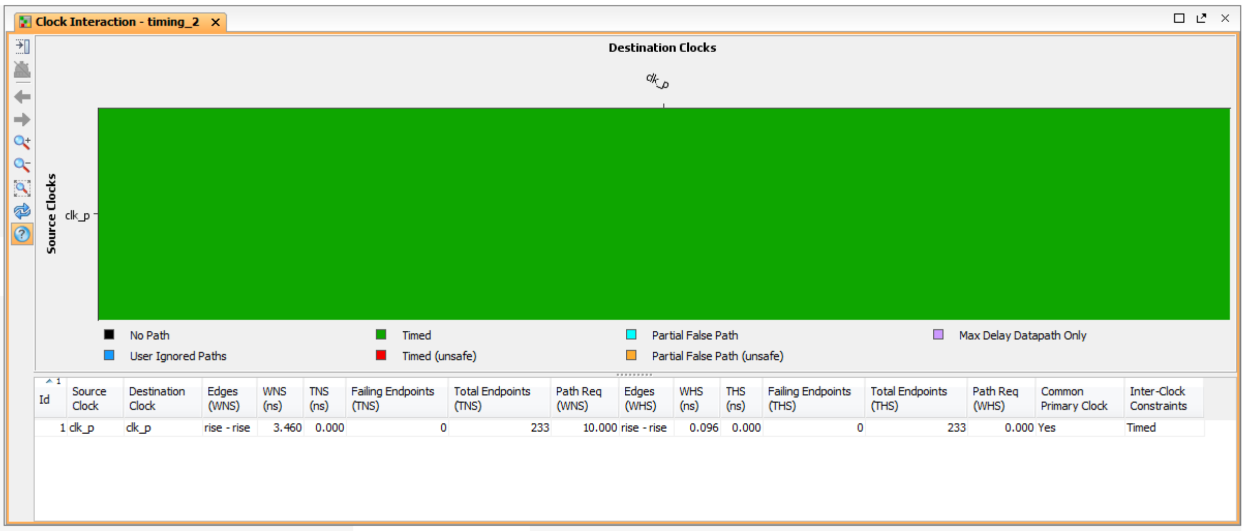
All user specified timing constraints are met.

Figure 10.29: Report Timing Summary tab

Step 1. To view the clock interaction of the design, expand **Implemented Design**, under the **Implementation** in the Flow Navigator, and select **Report Clock Interaction** command

Step 2. In the **Report Clock Interaction** dialog box, type the name of the results in the **Results name** field and click **OK**

Step 3. The **Clock Interaction** report will display in the main Vivado IDE window, see Illustration 10.30



Id	Source Clock	Destination Clock	Edges (WNS)	WNS (ns)	TNS (ns)	Failing Endpoints (TNS)	Total Endpoints (TNS)	Path Req (WNS)	Edges (WHS)	WHS (ns)	THS (ns)	Failing Endpoints (THS)	Total Endpoints (THS)	Path Req (WHS)	Common Primary Clock	Inter-Clock Constraints
1	clk_p	clk_p	rise - rise	3.460	0.000	0	233	10.000	rise - rise	0.096	0.000	0	233	0.000	Yes	Timed

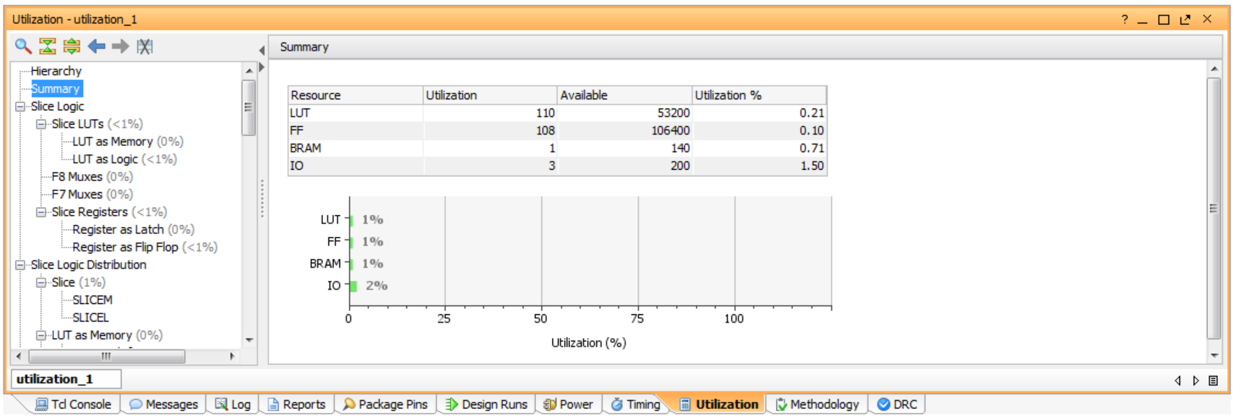
Figure 10.30: Report Clock Interaction tab

This report is helpful to tell us if timing is asynchronous (in case that we didn't include synchronization circuitry) and if paths are constrained (in case that we didn't add timing constraints to cover paths between unrelated clock domains). Green squares confirm that paths between the two clock domains are constrained.

Step 4. To view the resource utilization of the design, expand **Implemented Design**, under the **Implementation** in the Flow Navigator, and select **Report Utilization** command

Step 5. In the **Report Utilization** dialog box, type the name of the results in the **Results name** field and click **OK**

Step 6. The **Utilization** report will display at the bottom of the Vivado IDE, see Illustration 10.31



Summary

Resource	Utilization	Available	Utilization %
LUT	110	53200	0.21
FF	108	106400	0.10
BRAM	1	140	0.71
IO	3	200	1.50

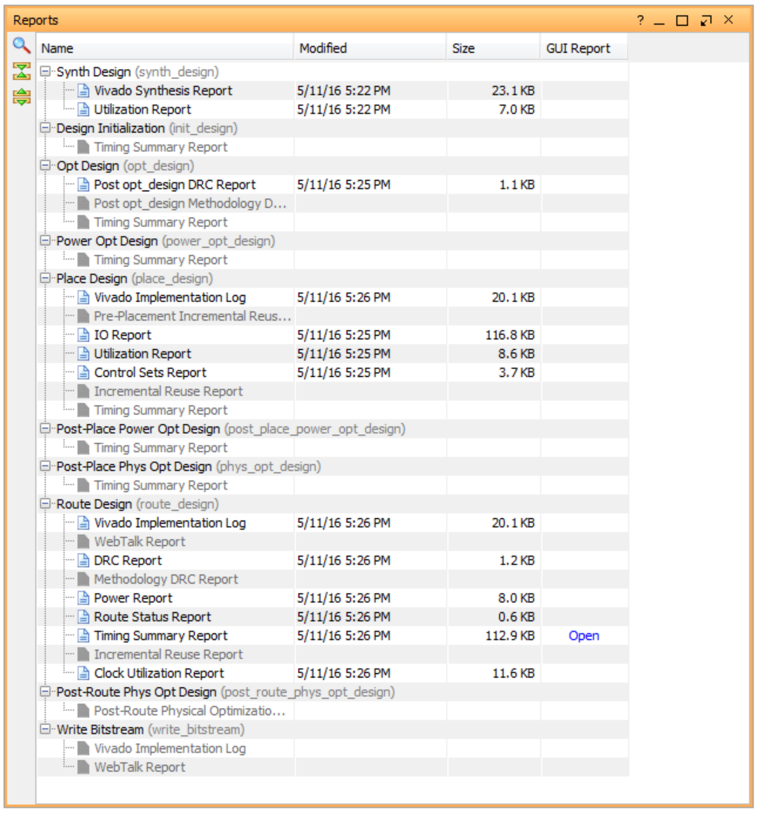
Figure 10.31: Utilization Report tab

Note: You can maximize the utilization report and explore the results.

Note: Information about the Vivado Implementation Process, you can also find in the **Lab 12: "Design Implementation"** .

10.2.4 Implementation Reports

While the Flow Navigator points to the most important reports, the **Reports** tab contains several other useful reports, see Illustration 10.32:

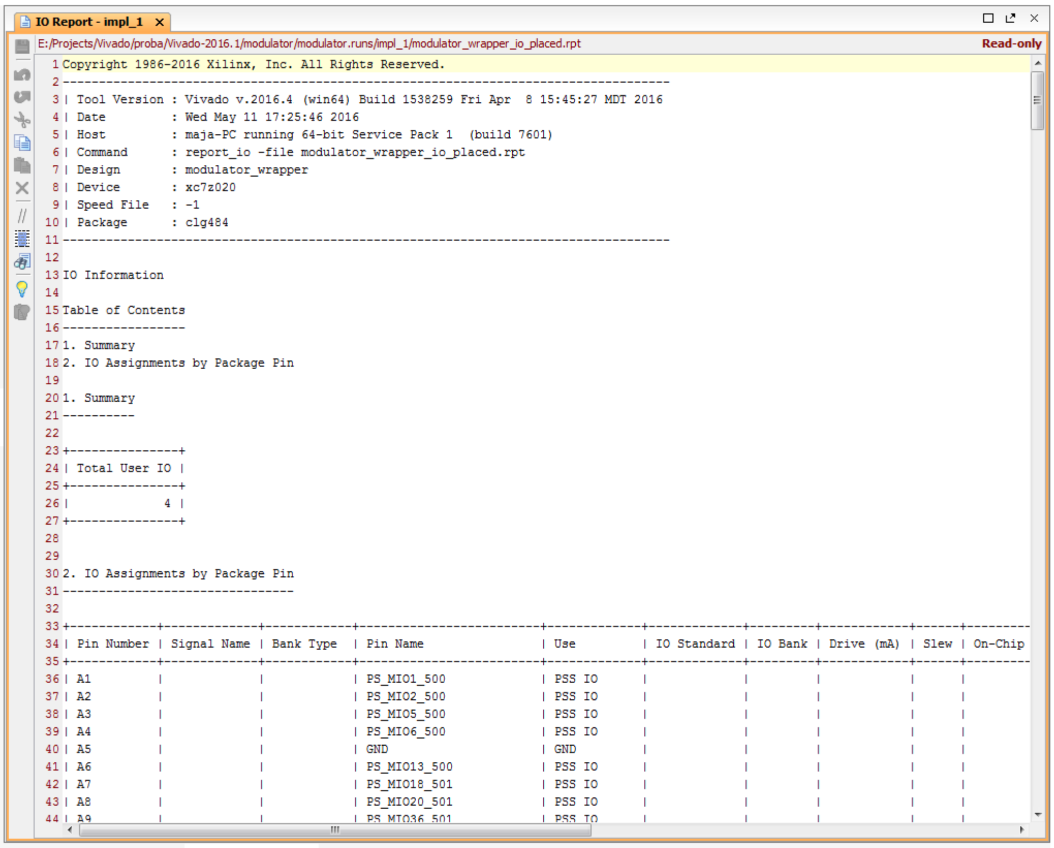

Reports

Name	Modified	Size	GUI Report
Synth Design (synth_design)			
Vivado Synthesis Report	5/11/16 5:22 PM	23.1 KB	
Utilization Report	5/11/16 5:22 PM	7.0 KB	
Design Initialization (init_design)			
Timing Summary Report			
Opt Design (opt_design)			
Post opt_design DRC Report	5/11/16 5:25 PM	1.1 KB	
Post opt_design Methodology D...			
Timing Summary Report			
Power Opt Design (power_opt_design)			
Timing Summary Report			
Place Design (place_design)			
Vivado Implementation Log	5/11/16 5:26 PM	20.1 KB	
Pre-Placement Incremental Reus...			
IO Report	5/11/16 5:25 PM	116.8 KB	
Utilization Report	5/11/16 5:25 PM	8.6 KB	
Control Sets Report	5/11/16 5:25 PM	3.7 KB	
Incremental Reuse Report			
Timing Summary Report			
Post-Place Power Opt Design (post_place_power_opt_design)			
Timing Summary Report			
Post-Place Phys Opt Design (phys_opt_design)			
Timing Summary Report			
Route Design (route_design)			
Vivado Implementation Log	5/11/16 5:26 PM	20.1 KB	
WebTalk Report			
DRC Report	5/11/16 5:26 PM	1.2 KB	
Methodology DRC Report			
Power Report	5/11/16 5:26 PM	8.0 KB	
Route Status Report	5/11/16 5:26 PM	0.6 KB	
Timing Summary Report	5/11/16 5:26 PM	112.9 KB	Open
Incremental Reuse Report			
Clock Utilization Report	5/11/16 5:26 PM	11.6 KB	
Post-Route Phys Opt Design (post_route_phys_opt_design)			
Post-Route Physical Optimizatio...			
Write Bitstream (write_bitstream)			
Vivado Implementation Log			
WebTalk Report			

Figure 10.32: Reports tab

Vivado Implementation Log - describes the implementation process and any issues it encountered

IO Report - Lists every signal, its attributes and its final location, see Illustration 10.33. It is always important to double-click pin assignments before implementing, because the tools can move any pin that is unassigned.



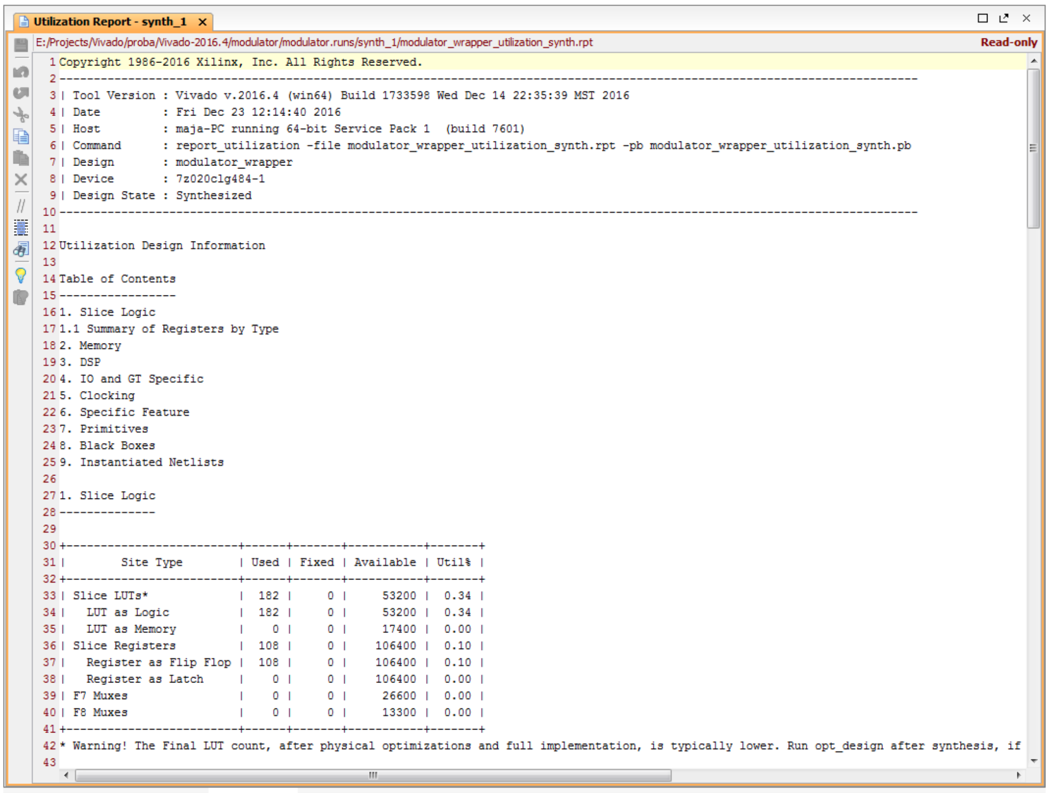
IO Report - impl_1

E:/Projects/Vivado/proba/Vivado-2016.1/modulator/modulator.runs/impl_1/modulator_wrapper_io_placed.rpt   Read-only

```
Copyright 1986-2016 Xilinx, Inc. All Rights Reserved.
--------------------------------------------------------------------------------
| Tool Version : Vivado v.2016.4 (win64) Build 1538259 Fri Apr  8 15:45:27 MDT 2016
| Date         : Wed May 11 17:25:46 2016
| Host         : maja-PC running 64-bit Service Pack 1  (build 7601)
| Command      : report_io -file modulator_wrapper_io_placed.rpt
| Design       : modulator_wrapper
| Device       : xc7z020
| Speed File   : -1
| Package      : clg484
--------------------------------------------------------------------------------

IO Information

Table of Contents
-----------------
1. Summary
2. IO Assignments by Package Pin

1. Summary
----------

+---------------+
| Total User IO |
+---------------+
|             4 |
+---------------+


2. IO Assignments by Package Pin
--------------------------------

+------------+-------------+------------+-------------------------+-------------+-------------+---------+------------+------+---------
| Pin Number | Signal Name | Bank Type  | Pin Name                | Use         | IO Standard | IO Bank | Drive (mA) | Slew | On-Chip
+------------+-------------+------------+-------------------------+-------------+-------------+---------+------------+------+---------
| A1         |             |            | PS_MIO1_500             | PSS IO      |             |         |            |      |
| A2         |             |            | PS_MIO2_500             | PSS IO      |             |         |            |      |
| A3         |             |            | PS_MIO5_500             | PSS IO      |             |         |            |      |
| A4         |             |            | PS_MIO6_500             | PSS IO      |             |         |            |      |
| A5         |             |            | GND                     | GND         |             |         |            |      |
| A6         |             |            | PS_MIO13_500            | PSS IO      |             |         |            |      |
| A7         |             |            | PS_MIO18_501            | PSS IO      |             |         |            |      |
| A8         |             |            | PS_MIO20_501            | PSS IO      |             |         |            |      |
| A9         |             |            | PS_MIO36_501            | PSS IO      |             |         |            |      |
```

Figure 10.33: IO Report

Utilization Report - describes the amount of FPGA resources used in a text format, see Illustration 10.34



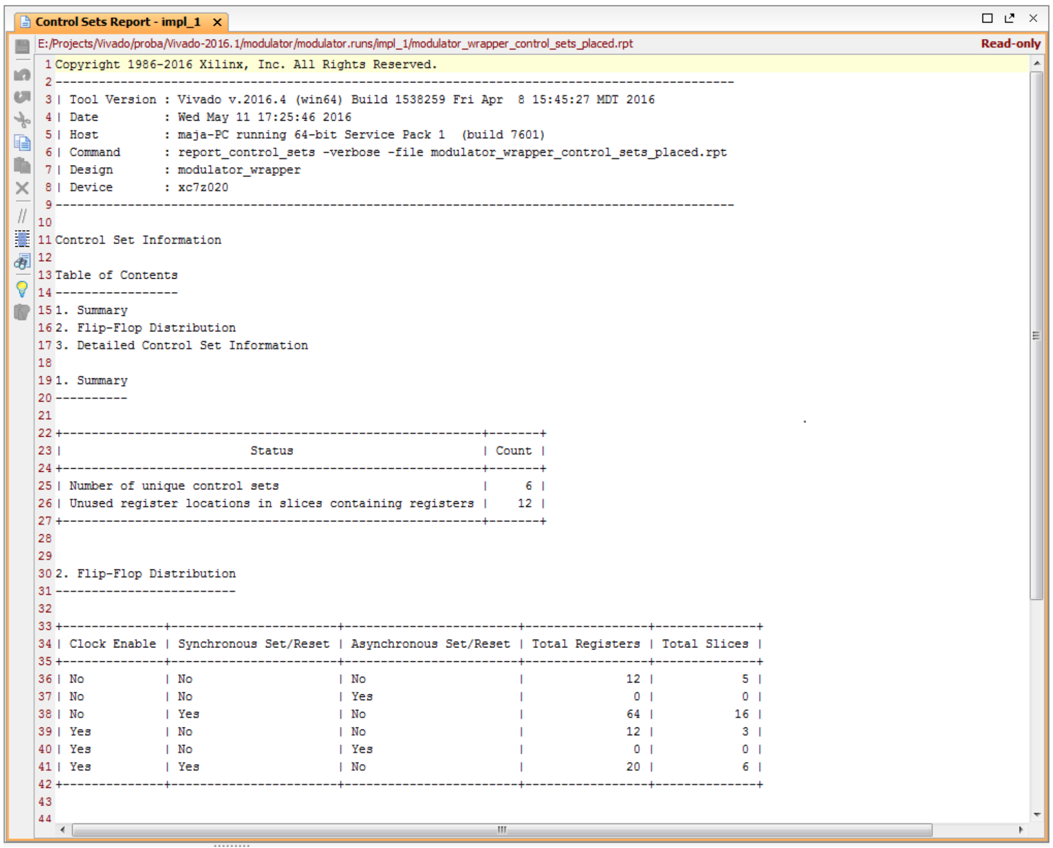
Utilization Report - synth_1

E:/Projects/Vivado/proba/Vivado-2016.4/modulator/modulator.runs/synth_1/modulator_wrapper_utilization_synth.rpt   Read-only

```
Copyright 1986-2016 Xilinx, Inc. All Rights Reserved.
------------------------------------------------------------------------------------------------------------------
| Tool Version : Vivado v.2016.4 (win64) Build 1733598 Wed Dec 14 22:35:39 MST 2016
| Date         : Fri Dec 23 12:14:40 2016
| Host         : maja-PC running 64-bit Service Pack 1  (build 7601)
| Command      : report_utilization -file modulator_wrapper_utilization_synth.rpt -pb modulator_wrapper_utilization_synth.pb
| Design       : modulator_wrapper
| Device       : 7z020clg484-1
| Design State : Synthesized
------------------------------------------------------------------------------------------------------------------

Utilization Design Information

Table of Contents
-----------------
1. Slice Logic
1.1 Summary of Registers by Type
2. Memory
3. DSP
4. IO and GT Specific
5. Clocking
6. Specific Feature
7. Primitives
8. Black Boxes
9. Instantiated Netlists

1. Slice Logic
--------------


+-------------------------+------+-------+-----------+-------+
|        Site Type        | Used | Fixed | Available | Util% |
+-------------------------+------+-------+-----------+-------+
| Slice LUTs*             |  182 |     0 |     53200 |  0.34 |
|   LUT as Logic          |  182 |     0 |     53200 |  0.34 |
|   LUT as Memory         |    0 |     0 |     17400 |  0.00 |
| Slice Registers         |  108 |     0 |    106400 |  0.10 |
|   Register as Flip Flop |  108 |     0 |    106400 |  0.10 |
|   Register as Latch     |    0 |     0 |    106400 |  0.00 |
| F7 Muxes                |    0 |     0 |     26600 |  0.00 |
| F8 Muxes                |    0 |     0 |     13300 |  0.00 |
+-------------------------+------+-------+-----------+-------+
* Warning! The Final LUT count, after physical optimizations and full implementation, is typically lower. Run opt_design after synthesis, if
```

Figure 10.34: Utilization Report

Control Sets Report - describes the number of unique control sets in the design Ideally this number will be as small as possible. Number of control sets describes how control signals were grouped. Control signals include clocks, clock enables, set, and reset signals. How the tools group them into slices and CLBs will dictate the density of the design in the FPGA.



Control Sets Report - impl_1

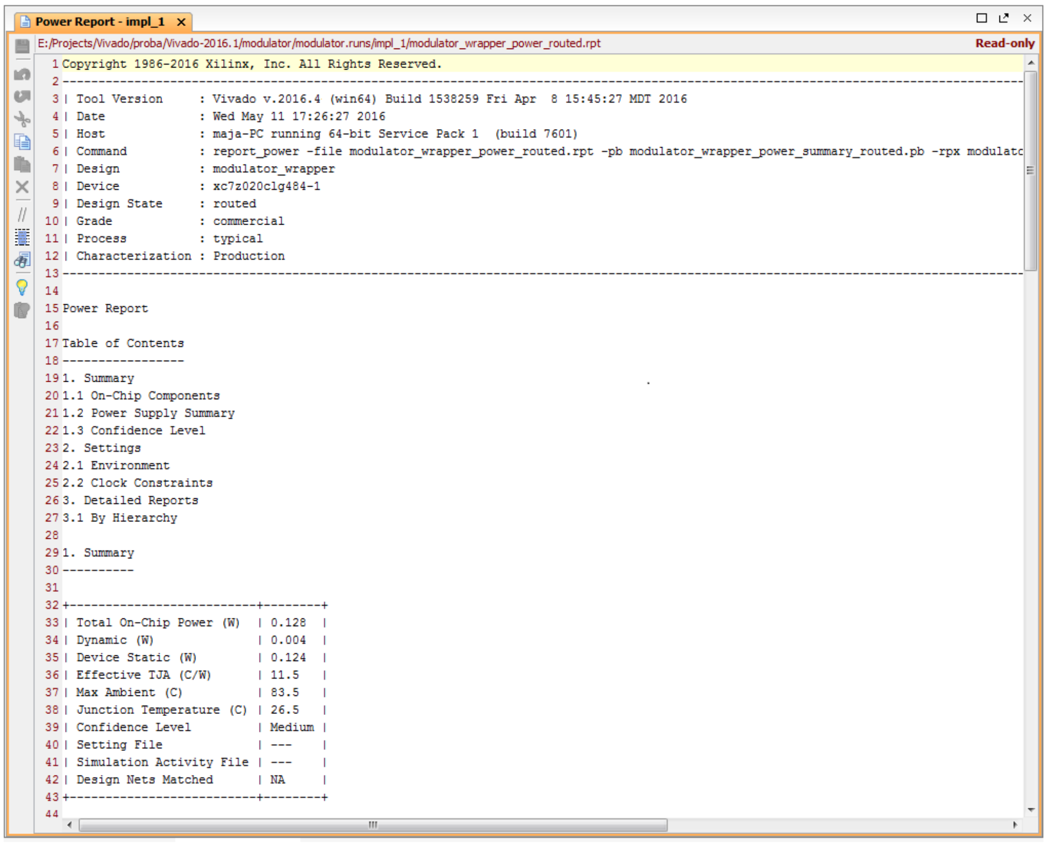
E:/Projects/Vivado/proba/Vivado-2016.1/modulator/modulator.runs/impl_1/modulator_wrapper_control_sets_placed.rpt Read-only

```
Copyright 1986-2016 Xilinx, Inc. All Rights Reserved.
----------------------------------------------------------------------------------------------
| Tool Version : Vivado v.2016.4 (win64) Build 1538259 Fri Apr  8 15:45:27 MDT 2016
| Date         : Wed May 11 17:25:46 2016
| Host         : maja-PC running 64-bit Service Pack 1  (build 7601)
| Command      : report_control_sets -verbose -file modulator_wrapper_control_sets_placed.rpt
| Design       : modulator_wrapper
| Device       : xc7z020
----------------------------------------------------------------------------------------------

Control Set Information

Table of Contents
-----------------
1. Summary
2. Flip-Flop Distribution
3. Detailed Control Set Information

1. Summary
----------

+-----------------------------------------------------------+-------+
|                          Status                           | Count |
+-----------------------------------------------------------+-------+
| Number of unique control sets                             |     6 |
| Unused register locations in slices containing registers |    12 |
+-----------------------------------------------------------+-------+


2. Flip-Flop Distribution
-------------------------

+--------------+-----------------------+------------------------+-----------------+--------------+
| Clock Enable | Synchronous Set/Reset | Asynchronous Set/Reset | Total Registers | Total Slices |
+--------------+-----------------------+------------------------+-----------------+--------------+
| No           | No                    | No                     |              12 |            5 |
| No           | No                    | Yes                    |               0 |            0 |
| No           | Yes                   | No                     |              64 |           16 |
| Yes          | No                    | No                     |              12 |            3 |
| Yes          | No                    | Yes                    |               0 |            0 |
| Yes          | Yes                   | No                     |              20 |            6 |
+--------------+-----------------------+------------------------+-----------------+--------------+

```

Figure 10.35: Control Sets Report

DRC Report - Lists the DRC routing checks that were completed

Power Report - describes the operating conditions and the estimated power consumption of your device, see Illustration 10.36

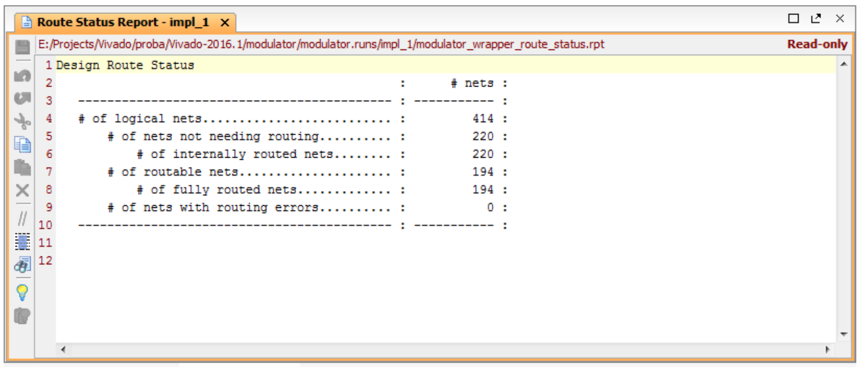

Power Report - impl_1

E:/Projects/Vivado/proba/Vivado-2016.1/modulator/modulator.runs/impl_1/modulator_wrapper_power_routed.rpt Read-only

```
Copyright 1986-2016 Xilinx, Inc. All Rights Reserved.
------------------------------------------------------------------------------------------------------------------------------
| Tool Version     : Vivado v.2016.4 (win64) Build 1538259 Fri Apr  8 15:45:27 MDT 2016
| Date             : Wed May 11 17:26:27 2016
| Host             : maja-PC running 64-bit Service Pack 1  (build 7601)
| Command          : report_power -file modulator_wrapper_power_routed.rpt -pb modulator_wrapper_power_summary_routed.pb -rpx modulatc
| Design           : modulator_wrapper
| Device           : xc7z020clg484-1
| Design State     : routed
| Grade            : commercial
| Process          : typical
| Characterization : Production
------------------------------------------------------------------------------------------------------------------------------

Power Report

Table of Contents
-----------------
1. Summary
1.1 On-Chip Components
1.2 Power Supply Summary
1.3 Confidence Level
2. Settings
2.1 Environment
2.2 Clock Constraints
3. Detailed Reports
3.1 By Hierarchy

1. Summary
----------

+--------------------------+--------+
| Total On-Chip Power (W)  | 0.128  |
| Dynamic (W)              | 0.004  |
| Device Static (W)        | 0.124  |
| Effective TJA (C/W)      | 11.5   |
| Max Ambient (C)          | 83.5   |
| Junction Temperature (C) | 26.5   |
| Confidence Level         | Medium |
| Setting File             | ---    |
| Simulation Activity File | ---    |
| Design Nets Matched      | NA     |
+--------------------------+--------+

```

Figure 10.36: Power Report

Route Status Report - reports lists any nets that could not be routed


Route Status Report - impl_1

E:/Projects/Vivado/proba/Vivado-2016.1/modulator/modulator.runs/impl_1/modulator_wrapper_route_status.rpt Read-only

```
Design Route Status
                                               :      # nets :
   ------------------------------------------- : ----------- :
   # of logical nets.......................... :         414 :
       # of nets not needing routing.......... :         220 :
           # of internally routed nets........ :         220 :
       # of routable nets..................... :         194 :
           # of fully routed nets............. :         194 :
       # of nets with routing errors.......... :           0 :
   ------------------------------------------- : ----------- :


```

Figure 10.37: Route Status Report

Timing Summary Report - identifies the default timing for the finished design (with true timing information)

The benefit of automatically generating these reports is that it encourages designers to read more about their design.

10.2.5 Run Post-Implementation Simulation

Simulation can be applied at several points in the design flow. It is one of the first steps after design entry and one of the last steps after implementation as part of the verifying the end functionality and performance of the design.

Simulation is an iterative process. It might need to be repeated until both the design functionality and the timing are met.

On the Illustration 10.38 is shown the simulation flow of a typical design.

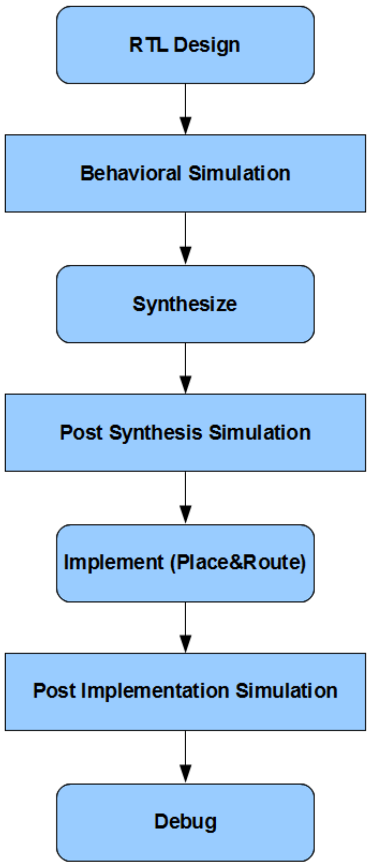


Figure 10.38: Simulation Flow

10.2.6 Run Post-Implementation Timing Simulation

You can perform functional or timing simulation after implementation process. Timing simulation is the closest emulation to actually downloading a design to a device. It allows you to ensure that the implemented design meets functional and timing requirements and has the expected behavior in the design.

To run post-implementation timing simulation, we must first create test bench for that type of simulation. We can use existing **modulator_tb.vhd** test bench file to create new **modulator_wrapper_timesim_tb.vhd** test bench file:

- change the entity name from ***modulator_tb*** to ***modulator_wrapper_timesim_tb***
- change the architecture name from ***modulator_tb*** to ***modulator_wrapper_timesim_tb*** also
- remove **clk_in_s** input clock signal and create two input clock differential signals: **clk_p_s** and **clk_n_s**
- remove **design_setting1_c** constant
- remove all the constants related to the div_factor_freqhigh and div_factor_freqlow value calculations: **c1_c**, **div_-factor_freqhigh_c**, **div_factor_freqlow_c**
- instead of Modulator module instance (**pwmmodulator**), instantiate Modulator Wrapper module (**modulatorwrap-per**) instance
- remove all the generics from the Modulator Wrapper module instance (**modulatorwrapper**) as it is shown in the code bellow

This last step is necessary because during the Synthesis process all the generics are being replaced by the values supplied by the designer. This means that the design that will be implemented will have no generics. Therefore, when we generate a Post-Implementation Timing Simulation model it can't contain any generics since they don't exist any more.

Modulator wrapper test bench file for the timing simulation:

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

    use work.modulator_pkg.all;


entity modulator_wrapper_timesim_tb is
    generic(
        -- 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
        );
end entity;

architecture tb of modulator_wrapper_timesim_tb is

    signal clk_p_s : std_logic := '1'; -- differential input clock signal
    signal clk_n_s : std_logic := '0'; -- differential input clock signal
    signal sw0_s   : std_logic := '1'; -- signal made for selecting frequency
    signal pwm_s   : std_logic := '0'; -- pulse width modulated signal

    -- period of input clock signal
    constant clock_period_c : time := 1000000000.0 / get_board_info_f(board_name_g).fclk * 1ns;

begin

    modulatorwrapper : entity work.modulator_wrapper   -- modulator_wrapper module instance
        port map(
            clk_p   => clk_p_s,
            clk_n   => clk_n_s,
            sw0     => sw0_s,
            pwm_out => pwm_s
            );

    clk_p_s <= not (clk_p_s) after clock_period_c/2; -- generates input clock signal
    clk_n_s <= not (clk_n_s) after clock_period_c/2; -- generates input clock signal
    sw0_s   <= '1', '0' after 25 us;

end;
```

After we have created a new test bench file (**modulator_wrapper_timesim_tb.vhd**) we must include it in our design:

Step 1. In the **Flow Navigator**, under the **Project Manager** , click **Add Sources** command

Step 2. In the **Add Sources** dialog box, choose **Add or create simulation sources** option and click **Next**

Step 3. In the **Add or Create Simulation Sources** dialog box, click on the **Specify simulation set** drop-down list and choose **Create Simulation Set...** option

Step 4. In the **Create Simulation Set** dialog box, enter a name for the new simulation set or leave **sim_6** as a name and click **OK**

Step 5. In the **Add or Create Simulation Sources** dialog box, under the new **sim_6** simulation set, click "+" icon and select **Add Files...** option

Step 6. In the **Add Source Files** dialog box, select **modulator_wrapper_timesim_tb.vhd** source file and click **OK**

Step 7. In the **Add or Create Simulation Sources** dialog box, click **Finish** and your new test bench file should appear in the **Sources** window, under the **Simulation Sources** / **sim_6**

Step 8. In the **Sources** window, select new **sim_6** simulation set, right-click on it and choose **Make Active** option

Step 9. Select the **modulator_wrapper_timesim_tb - tb (modulator_wrapper_timesm_tb.vhd)** file, right-click on it and choose **Set as Top** option

After we have added a new **modulator_wrapper_timesim_tb.vhd** test bench file into the design, we can start post-implementation timing simulation:

Step 1. In the **Flow Navigator**, click on the **Run Simulation** command and choose **Run Post-Implementation Timing Simulation** option, see Illustration 10.39

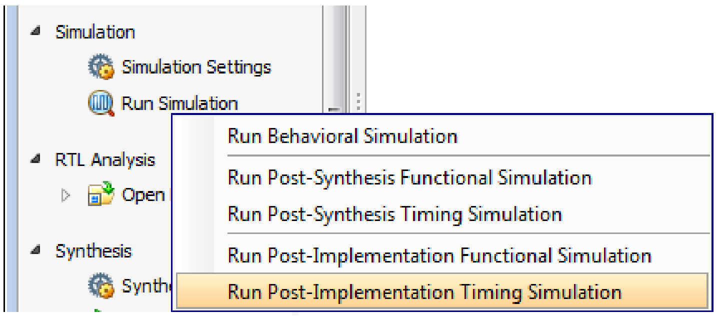

Simulation
Simulation Settings
Run Simulation
RTL Analysis
Open
Synthesis
Synth
Run Behavioral Simulation
Run Post-Synthesis Functional Simulation
Run Post-Synthesis Timing Simulation
Run Post-Implementation Functional Simulation
Run Post-Implementation Timing Simulation

Figure 10.39: Run Post-Implementation Timing Simulation option

Note: If your Vivado IDE notify an error that compiler cannot find package (**modulator_pkg.vhd**), that means that Vivado simulator has not included package automatically. Here are the steps to correct this problem:

Step 1. First step will be to add **modulator_pkg.vhd** file into the **sim_6** simulation set. To do that, click the **Add Sources** command

Step 2. In the **Add Sources** dialog box, choose **Add or create simulation sources** option and click **Next**

Step 3. In the **Add or Create Simulation Sources** dialog box, choose **sim_6** as simulation set from the **Specify simulation set** drop-down list, click the "+" icon and select **Add Files...** option

Step 4. In the **Add Source Files** dialog box, choose **modulator_pkg.vhd** file and click **OK**

Step 5. In the **Add or Create Simulation Sources** dialog box, click **Finish** and your **modulator_pkg.vhd** source file will be added under the **sim_6** simulation set

Step 6. To see where is added **modulator_pkg.vhd** source file (because it is not visible in the **Sources** view, in the **Hierarchy** tab, under the **sim_6** simulation set), click on the **Libraries** tab and expand **sim_6** simulation set, see Illustration 10.40

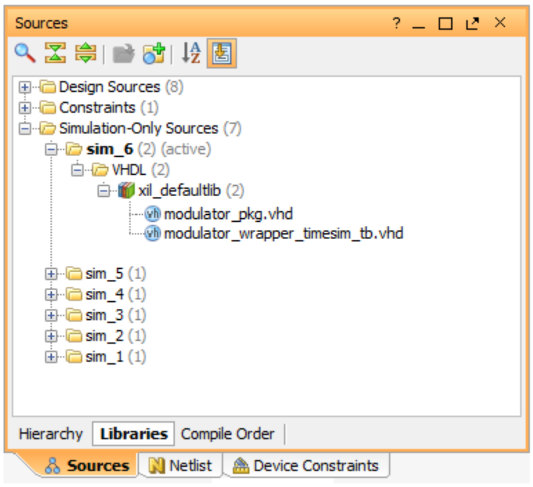
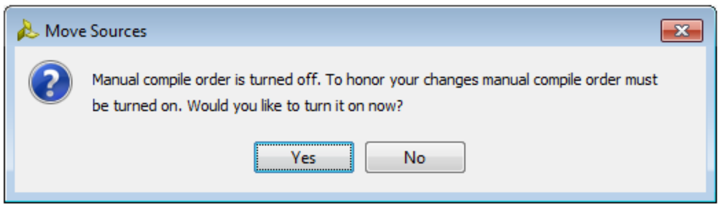


Figure 10.40: Libraries tab with added modulator_pkg.vhd file

As you can see from the picture above, **modulator_pkg.vhd** source file is now located in the library **xil_defaultlib**, as it should be.

Note: If you would like to see real compile order of your source files, open the **Compile Order** tab, beside the Libraries tab. If you are not satisfied with the automatically generated compile order of your source files, you can change it in the following way:

- To manually move some file from one place to another, **Manual Compile Order** option must be turned on. Before start moving process, tool will ask you would you like to turn on the Manual Compile Order option, see Illustration 10.41


Move Sources

Manual compile order is turned off. To honor your changes manual compile order must be turned on. Would you like to turn it on now?

Figure 10.41: Move Sources dialog box - Manual compile order

- Click Yes and the selected source file will be moved to the place of your choice

Step 7. After all those modifications, we can turn back into the **Hierarchy** tab, select **modulator_wrapper_timesim_tb.-vhd** simulation model and start ones more our post-implementation timing simulation

After implementation, the simulation information is much more complete, so you can get a better perspective on how the functionality of your design is meeting your requirements.

After you select a post-implementation functional simulation, the functional netlist is generated and the UNISIM libraries are used for simulation.

After you select a post-implementation timing simulation, the timing netlist and the SDF file are generated.

Step 8. Simulate your design for **200 ms** (see **Chapter 2.6 Simulating with Vivado Simulator – step 9.**)

Step 9. Assuming no errors, your simulation result should look similar to **Illustration 10.42**

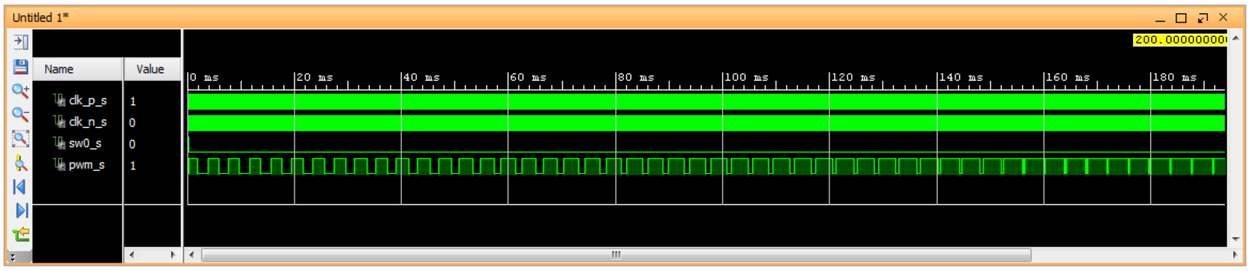


Figure 10.42: Timing Simulation Results

As you can see the results of timing simulation (waveform of the pwm_s signal) look identical to the results of functional simulation. This means that the desired functionality is preserved after all implementation steps have been performed. What is identical is the desired functionality (the shape of the pwm_s signal), but the timing properties of the pwm_s signal simulated using functional and timing simulation are significantly different, as can be seen from the following Illustrations (10.43 and 10.44).

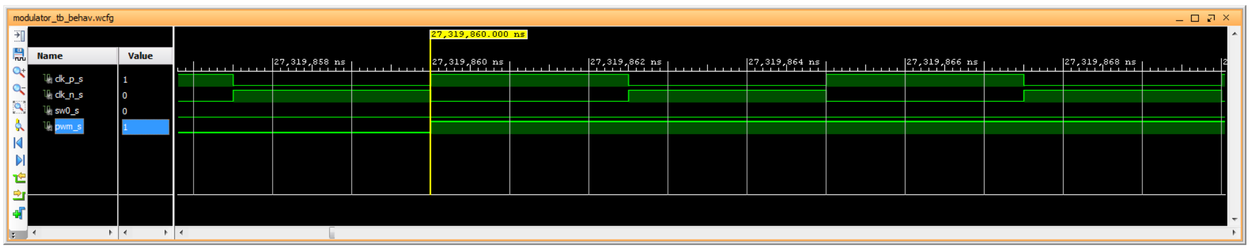


Figure 10.43: Functional Simulation Results

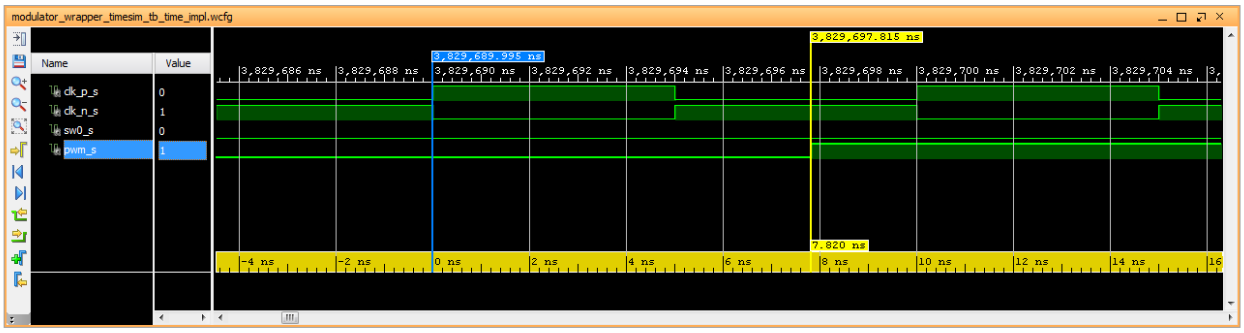


Figure 10.44: Timing Simulation Results (with signal delays)

Illustration 10.44 shows how big is the pwm_s signal delay related to the rising edge of the input clock signal (clk_p_s). This signal delay is illustrated with two markers (yellow and blue) and it amounts **7820 ns**.

Note: You can see that timing simulation lasts much longer than functional simulations. This is the reason why timing simulation is not often used in practice.

10.3 Generate Bitstream File

You can run the bitstream file generation process after your design has been completely routed for FPGAs. The bitstream file generation process produces a bitstream for Xilinx device configuration. After the design is completely routed, you must configure the device to execute the desired function.

To generate the bitstream file:

Step 1. In the **Flow Navigator**, under **Program and Debug**, click on the **Generate Bitstream** command, see Illustration 10.45

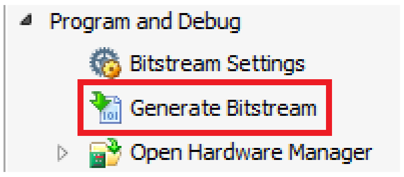


Figure 10.45: Generate Bitstream command

Note that the **Generate Bitstream** process will try to resynthesize and implement the design if any of process is out of date.

Step 2. Click **Yes** to acknowledge running of the processes that are needed for bitstream generation

Step 3. Click **Cancel** in the **Bitstream Generation Completed** dialog box

Note: Information about how to generate bitstream file, you can also find in the **Lab 12: "Design Implementation"**.

10.4 Program Device

After you have generated the bitstream file, the next step will bi to download it into the target FPGA device. In our case it will be ZedBoard evaluation board.

The Vivado tool offers **Open Hardware Manager** to use the native in-system device programming capabilities that are built into the Vivado IDE.

The Vivado IDE tool includes functionality that allows you to connect to your hardware, containing one or more FPGA devices, to program them and debug your design on the real hardware. Connecting to hardware can be done either from the Vivado IDE GUI or by using Tcl commands. In both cases, the procedure is the same:

Step 1. For the ZedBoard evaluation board, connect the Digilent USB JTAG cable of your ZedBoard board to the Windows machine's USB port

Step 2. Ensure that the board is plugged in and powered on

Step 3. Make sure that the board settings are proper

Step 4. In the **Flow Navigator**, under the **Program and Debug**, click **Open Hardware Manager** command, see Illustration 10.46

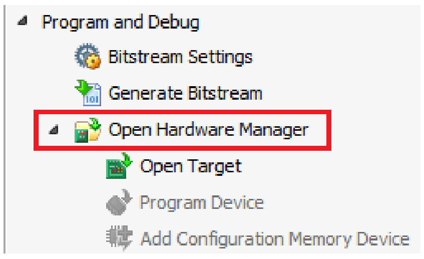


Figure 10.46: Open Hardware Manager command

The another way to open the hardware manager is to select **Flow ->** **Open Hardware Manager** option from the main Vivado menu

Step 5. The next step in opening a hardware target is connecting to the hardware server that is managing the connection to the hardware target. You can do this on three ways:

- Use the **Open target** selection in the **Hardware Manager** view, to open a recent or a new hardware targets, see Illustration 10.47

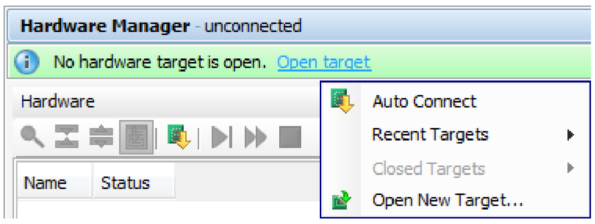


Figure 10.47: Hardware Manager view

- Use the **Open Target** command, under the **Open Hardware Manager** in the **Program and Device** section, to open new or recent hardware targets, see Illustration 10.48

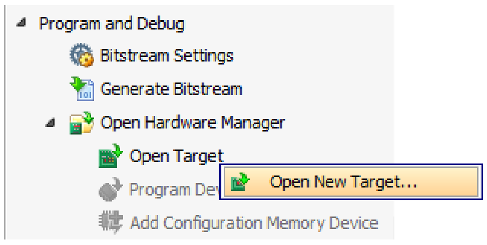
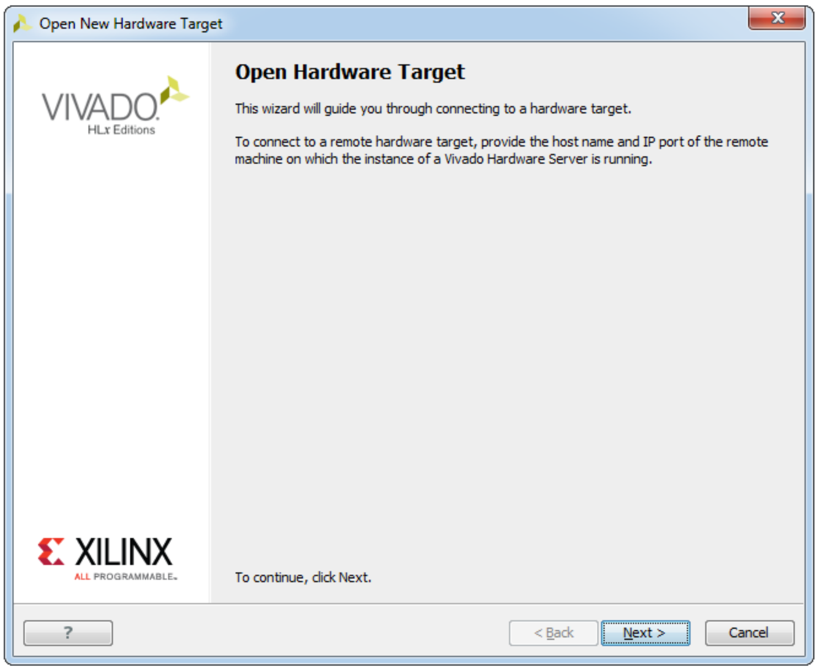


Figure 10.48: Open Target command

- Use Tcl commands to open a connection to a hardware target

Step 6. Click on the **Open New Target** command. The **Open New Hardware Target** wizard provides an interactive way for you to connect to a hardware server and target, see Illustration 10.49


Open New Hardware Target

Open Hardware Target

This wizard will guide you through connecting to a hardware target.

To connect to a remote hardware target, provide the host name and IP port of the remote machine on which the instance of a Vivado Hardware Server is running.

To continue, click Next.

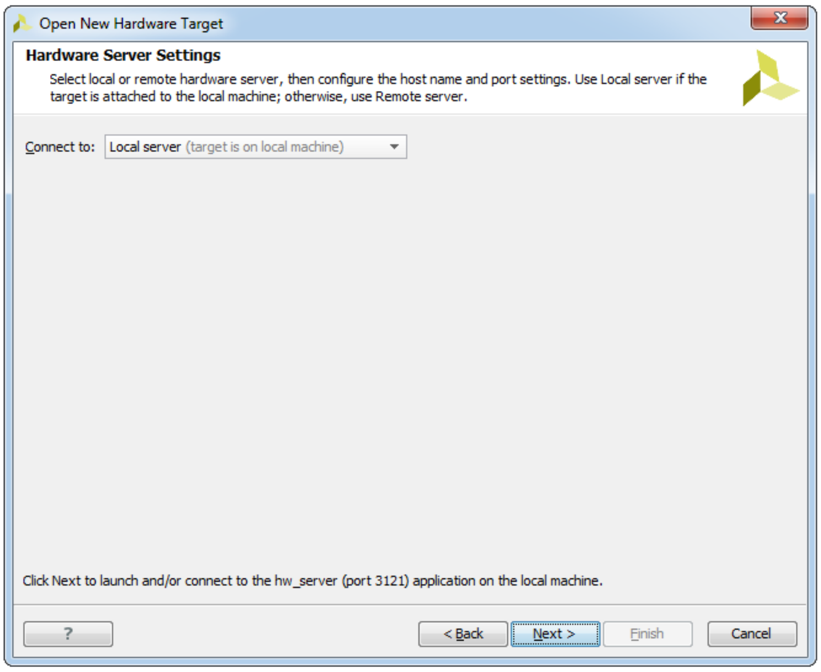
Figure 10.49: Open Hardware Target dialog box

Step 7. In the **Open Hardware Target** dialog box, click **Next**

Step 8. In the **Hardware Server Settings** dialog box, specify or select a local or remote server, depending on what machine your hardware target is connected to. Leave the default **Local server** and click **Next** , see Illustration 10.50

Local server: Use this setting if your hardware target is connected to the same machine on which you are running the Vivado IDE. The Vivado software automatically starts the Vivado hardware server (*hw_server*) application on the local machine.

Remote server: Use this setting if your hardware target is connected to a different machine on which you are running the Vivado IDE. Specify the host name or IP address of the remote machine and the port number for the hardware server (*hw_server*) application that is running on that machine.


Open New Hardware Target

Hardware Server Settings

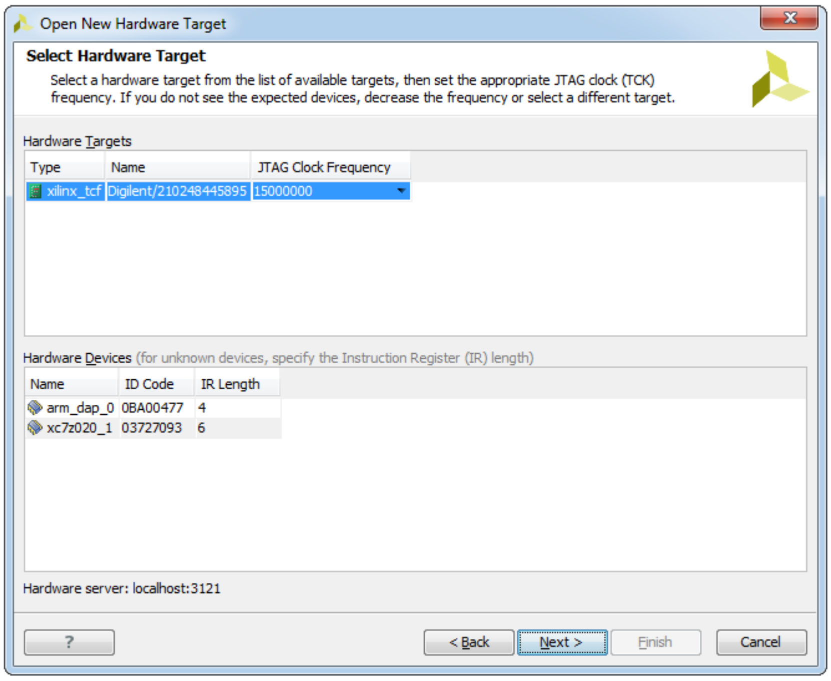
Select local or remote hardware server, then configure the host name and port settings. Use Local server if the target is attached to the local machine; otherwise, use Remote server.

Connect to: Local server (target is on local machine)

Click Next to launch and/or connect to the hw_server (port 3121) application on the local machine.

Figure 10.50: Hardware Server Settings dialog box

Step 9. In the **Select Hardware Target** dialog box, select the appropriate hardware target from the list of targets that are managed by the hardware server. Note that when you select a target, you will see the various hardware devices that are available on the hardware target, see Illustration 10.51



Select Hardware Target

Select a hardware target from the list of available targets, then set the appropriate JTAG clock (TCK) frequency. If you do not see the expected devices, decrease the frequency or select a different target.

Hardware Targets

Type	Name	JTAG Clock Frequency
xilinx_tcf	Digilent/210248445895	15000000

Hardware Devices (for unknown devices, specify the Instruction Register (IR) length)

Name	ID Code	IR Length
arm_dap_0	0BA00477	4
xc7z020_1	03727093	6

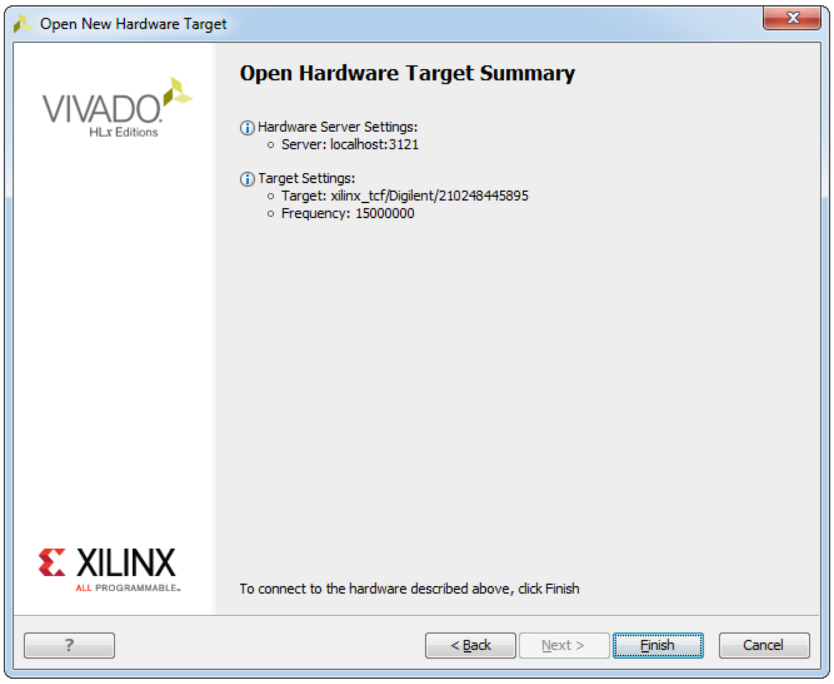
Hardware server: localhost:3121

Figure 10.51: Select Hardware Target dialog box

Note: If one or more of the devices is unknown to Vivado tool, you can provide the instruction register (IR) length directly in the **Hardware Devices** table of the Open Hardware Target wizard, see Illustration 10.51

Step 10. Click **Next**

Step 11. In the **Open Hardware Target Summary** dialog box, click **Finish** to connect to the hardware described in the summary window, see Illustration 10.52



Open Hardware Target Summary

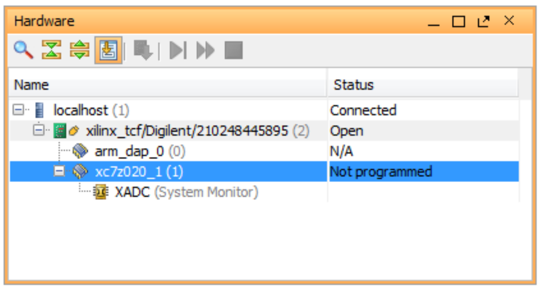
Hardware Server Settings:
- Server: localhost:3121

Target Settings:
- Target: xilinx_tcf/Digilent/210248445895
- Frequency: 15000000

To connect to the hardware described above, click Finish

Figure 10.52: Open Hardware Target Summary dialog box

Ones you finish opening a connection to a hardware target, the **Hardware** window is populated with the hardware server, hardware target, and various hardware devices for the open target, see Illustration 10.53



Name	Status
localhost (1)	Connected
xilinx_tcf/Digilent/210248445895 (2)	Open
arm_dap_0 (0)	N/A
xc7z020_1 (1)	Not programmed
XADC (System Monitor)	

Figure 10.53: Hardware view after opening a connection to the hardware target

Step 12. You can program the hardware device right-clicking on the device in the **Hardware** window and selecting the **Program Device...** option, see Illustration 10.54

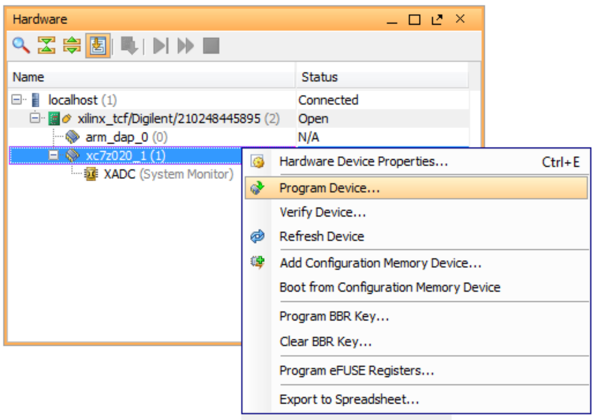


Figure 10.54: Program Device option

The another way to program your device is to select **Program device** option from the **Hardware Manager** view and choose the target device (**xc7z020_1**), as it is shown on the Illustration 10.55

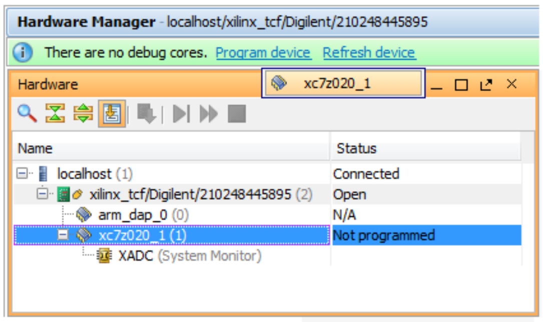
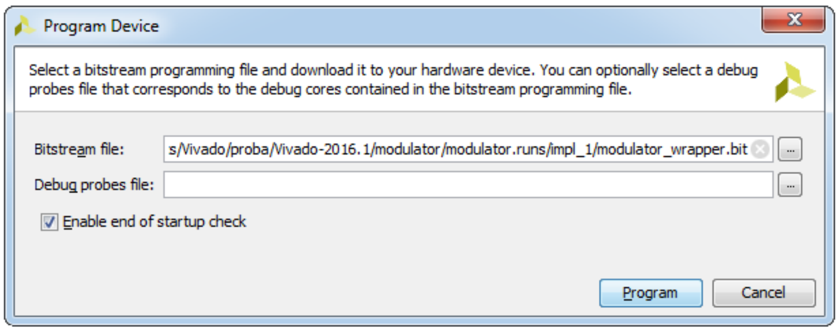

Hardware Manager - localhost/xilinx_tcf/Digilent/210248445895
There are no debug cores. Program device Refresh device
Hardware
xc7z020_1
Name
Status
localhost (1)
Connected
xilinx_tcf/Digilent/210248445895 (2)
Open
arm_dap_0 (0)
N/A
xc7z020_1 (1)
Not programmed
XADC (System Monitor)

Figure 10.55: Program device option from the Hardware Manager view

In the **Program Device** window, click **Program** to program your device, see Illustration 10.56


Program Device

Select a bitstream programming file and download it to your hardware device. You can optionally select a debug probes file that corresponds to the debug cores contained in the bitstream programming file.

Bitstream file: s/Vivado/proba/Vivado-2016.1/modulator/modulator.runs/impl_1/modulator_wrapper.bit

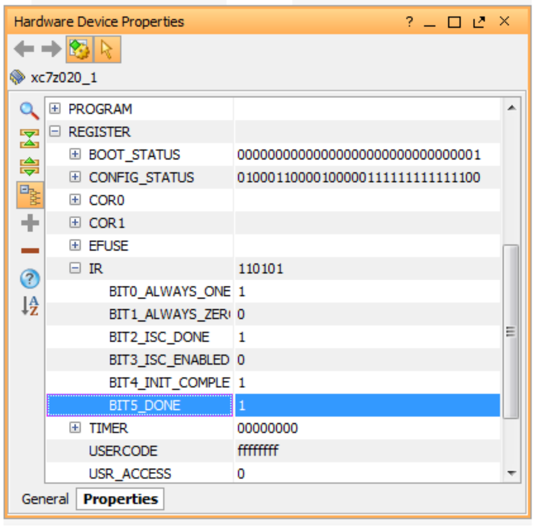
Debug probes file:

☑ Enable end of startup check

Figure 10.56: Program Device window

Note: As a convenience, Vivado IDE automatically uses .bit file for the current implemented design as the values for the programming file property of the first matching device in the open hardware target.

Ones the progress dialog box has indicated that the programming is 100% complete, you can check that the hardware device has been programmed successfully by examining the DONE status in the **Hardware Device Properties** view, see Illustration 10.57


Hardware Device Properties

xc7z020_1

PROGRAM	
REGISTER	
BOOT_STATUS	00000000000000000000000000000001
CONFIG_STATUS	01000110000100000111111111111100
COR0	
COR1	
EFUSE	
IR	110101
BIT0_ALWAYS_ONE	1
BIT1_ALWAYS_ZER	0
BIT2_ISC_DONE	1
BIT3_ISC_ENABLED	0
BIT4_INIT_COMPLE	1
BIT5_DONE	1
TIMER	00000000
USERCODE	ffffffff
USR_ACCESS	0

Figure 10.57: Hardware Device Properties window

After downloading your design into the ZedBoard device, led diode on the board will start blinking. The speed of blinking will be chosen depends on the position of the two-state on-board switch sw0.

If you want to close a hardware target, right-click on the hardware target in the **Hardware** window and select **Close Target** option from the popup manu, see Illustration 10.58

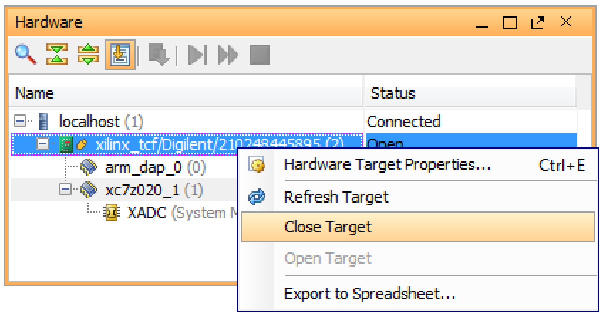


Figure 10.58: Close Target option

If you want to close a connection to the hardware server, right-click on the hardware server in the **Hardware** window and select **Close Server** option from the popup menu, see Illustration 10.59

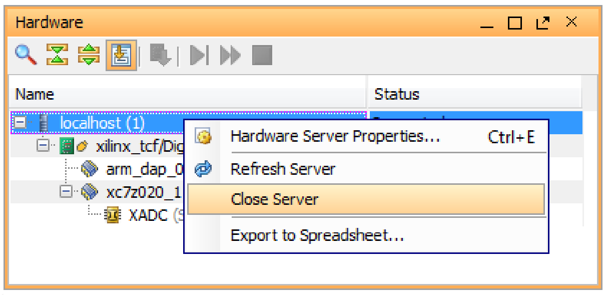


Figure 10.59: Close Server option

Assuming no errors occurs, you can test your design with a Vivado logic analyzer or an oscilloscope.

Note: Information about how to program an FPGA device, you can also find in the **Lab 12: "Design Implementation"**.

10.5 Modifications in case of using different development boards

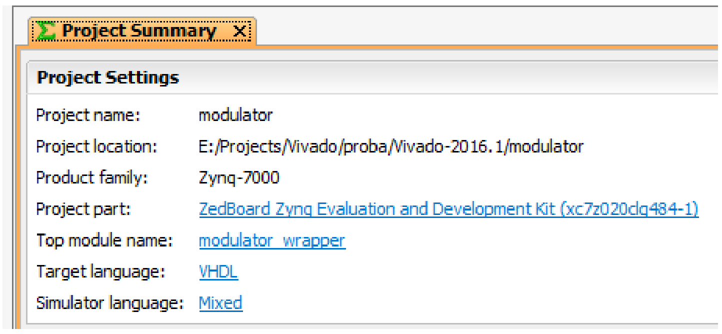
In case of using some other development board, some small modifications to accommodate your design to the new development board, must be done.

These modifications will be illustrated in case of using **Virtex-7 (VC707)** development board.

Difference between ZedBoard and Virtex-7 development board is that ZedBoard has single-ended reference clock, while Virtex-7 has differential reference clock. The other difference between these two boards is the frequency of the reference clock. ZedBoard has 100 MHz reference clock, while Virtex-7 has 200 MHz reference clock.

Step 1. Change the type of the target FPGA device

- In the **Project Summary** window (**Project Settings**) click on the **Project part: ZedBoard Zynq Evaluation and Development Kit (xc7z020clg484-1)**, see Illustration 10.60



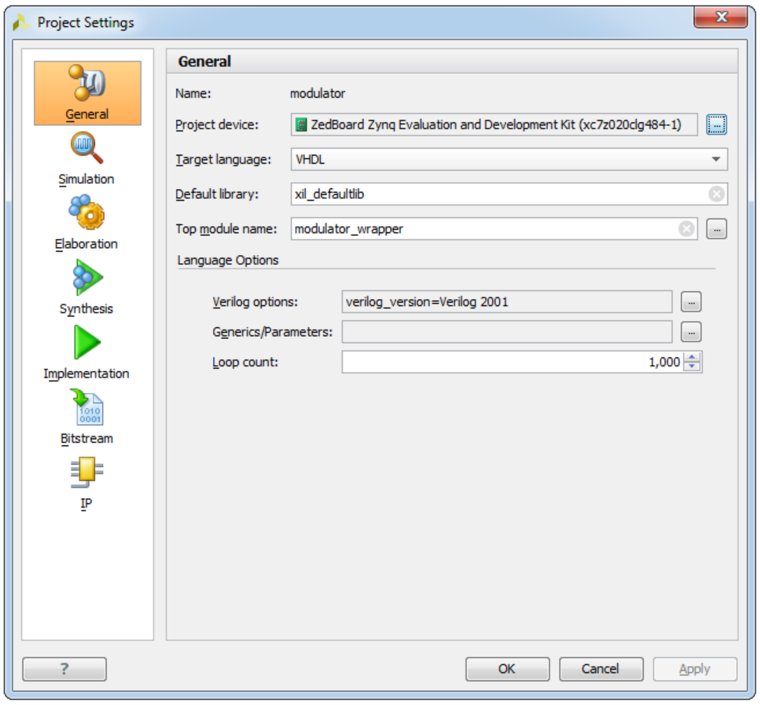
Project Summary

Project Settings

Project name:	modulator
Project location:	E:/Projects/Vivado/proba/Vivado-2016.1/modulator
Product family:	Zynq-7000
Project part:	ZedBoard Zynq Evaluation and Development Kit (xc7z020clg484-1)
Top module name:	modulator_wrapper
Target language:	VHDL
Simulator language:	Mixed

Figure 10.60: Project Settings window

- In the **Project Settings** dialog box, click on the icon beside **Project device** field to browse the another development board, see Illustration 10.61



General

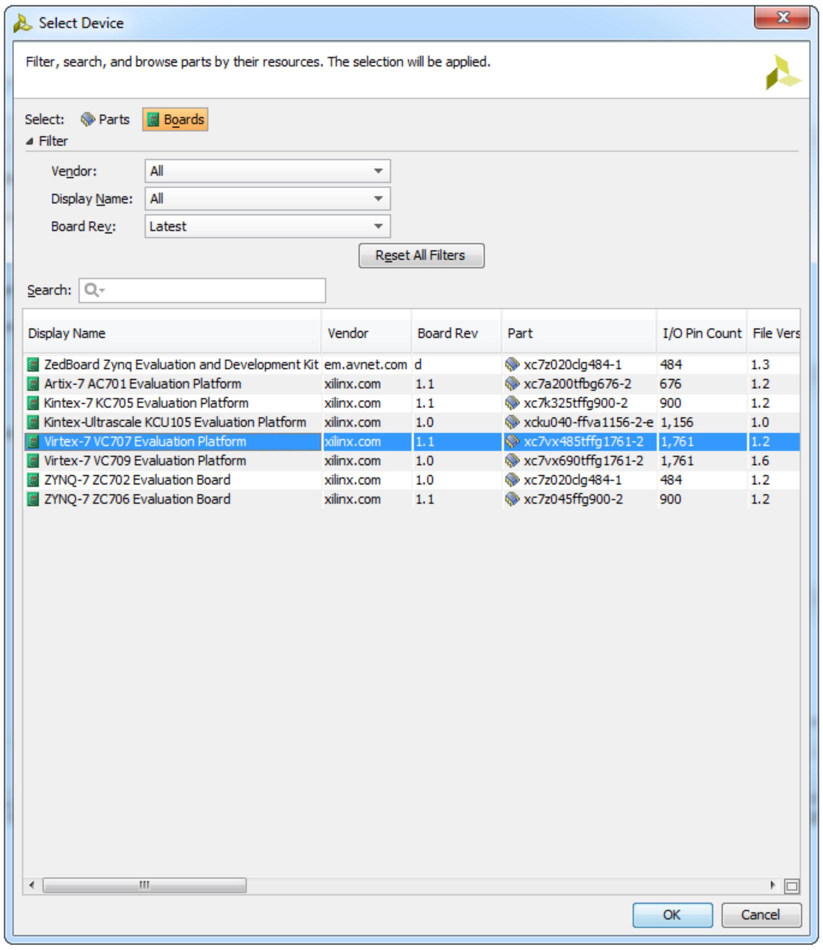
Name:	modulator
Project device:	ZedBoard Zynq Evaluation and Development Kit (xc7z020clg484-1)
Target language:	VHDL
Default library:	xil_defaultlib
Top module name:	modulator_wrapper

Language Options

Verilog options:	verilog_version=Verilog 2001
Generics/Parameters:	
Loop count:	1,000

Figure 10.61: Project Settings dialog box

In the **Select Device** dialog box, select **Virtex-7 VC707 Evaluation Platform** and click **OK**, see Illustration 10.62



Select Device

Filter, search, and browse parts by their resources. The selection will be applied.

Select: Parts [✓ Boards]

Filter

Vendor: All

Display Name: All

Board Rev: Latest

Reset All Filters

Search:

Display Name	Vendor	Board Rev	Part	I/O Pin Count	File Vers
ZedBoard Zynq Evaluation and Development Kit	em.avnet.com	d	xc7z020clg484-1	484	1.3
Artix-7 AC701 Evaluation Platform	xilinx.com	1.1	xc7a200tfbg676-2	676	1.2
Kintex-7 KC705 Evaluation Platform	xilinx.com	1.1	xc7k325tffg900-2	900	1.2
Kintex-Ultrascale KCU105 Evaluation Platform	xilinx.com	1.0	xcku040-ffva1156-2-e	1,156	1.0
Virtex-7 VC707 Evaluation Platform	xilinx.com	1.1	xc7vx485tffg1761-2	1,761	1.2
Virtex-7 VC709 Evaluation Platform	xilinx.com	1.0	xc7vx690tffg1761-2	1,761	1.6
ZYNQ-7 ZC702 Evaluation Board	xilinx.com	1.0	xc7z020clg484-1	484	1.2
ZYNQ-7 ZC706 Evaluation Board	xilinx.com	1.1	xc7z045ffg900-2	900	1.2

OK Cancel

Figure 10.62: Select Device dialog box

Step 2. Change the xdc constraints file

Open the **modulator.xdc** file from your working directory and make the following changes:

```
set_property LOC E19 [get_ports clk_p];
set_property LOC E18 [get_ports clk_n];
set_property LOC AV30 [get_ports sw0];
set_property LOC AM39 [get_ports pwm_out];

set_property IOSTANDARD LVDS [get_ports clk_p];
set_property IOSTANDARD LVDS [get_ports clk_n];
set_property IOSTANDARD LVCMOS18 [get_ports sw0];
set_property IOSTANDARD LVCMOS18 [get_ports pwm_out];

create_clock -period 5.000 -name clk_p -waveform {0.000 2.500} [get_ports clk_p]
```

The things that we changed in the xdc file:

- *Placement Constraints* - find in the User Guide for the Virtex-7 (VC707) development board pin locations where you would like to connect the input differential clock (clk_p, clk_n) and the sw0 and pwm_out ports.
- *Timing Constraints* - change the period of the input clock signal. For Virtex-7 (VC707) development board, you have to change input clock period from 10 ns to 5 ns, because Virtex-7 (VC707) development board has 200 MHz input clock frequency.

Step 3. Change the source codes

Because we changed the target development board, from ZedBoard to Virtex-7 (VC707), we must accommodate the whole system to the new parameters.

Changes that must be done are listed below.

If you want to add some other development board that is not on the list of the available development boards in our design, please open the **modulator_pkg.vhd** source file and add the desired development board information.

modulator_pkg.vhd:

- Add the name of the new development board in the **board_type_t** type declaration:

```
type board_type_t is (lx9, zedboard, ml605, kc705, vc707, microzed, socius);
```

- Create a new constant for the new development board. Constant must be a structure of type **board_setting_t_rec**. In that structure you must declare the following parameters:
  - the name of the new development board defined in the **board_type_t** type declaration
  - the frequency of the input clock signal in **MHz**
  - is the input clock differential (yes) or not (no), using a **has_diff_clk_t** type field

```
-- 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 vc707_c    : board_setting_t_rec := (vc707, 200000000.0, yes);   -- Virtex-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
```

modulator_wrapper_rtl.vhd and ***modulator_tb.vhd***:

- Change the type of your development board. In our case it will be from **zedboard** to **vc707**.

```
-- Parameter that specifies major characteristics of the board that will be used
-- to implement the modulator design
-- Possible choices: """lx9""", """zedboard""", """ml605""", """kc705""", """vc707""", """microzed""",
       """socius"""
-- Adjust the modulator_pkg.vhd file to add more
board_name_g : string := """vc707""";
```

Chapter 11

DEBUGGING DESIGN

In this chapter we will show how user can debug a design. We will use two types of analyzers, Vivado Logic Analyzer as an integrated Vivado analyzer and oscilloscope as an external debugging device.

11.1 Inserting ILA and VIO Cores into Design

In this chapter you will learn how to debug your FPGA design by inserting an Integrated Logic Analyzer (ILA) core and Virtual Input/Output (VIO) core using the Vivado IDE. You will take advantage of integrated Vivado logic analyzer functions to debug and discover some potential root causes of your design.

There are two flows (methods) supported in the Vivado Debug Probing:

1. HDL Instantiation Debug Probing Flow
2. Using the Netlist Insertion Debug Probing Flow

This chapter will illustrate "Using the Netlist Insertion Debug Probing Flow" between Vivado logic analyzer, ILA 6.2, VIO 3.0 and Vivado IDE. Details about how to use the "HDL Instantiation Debug Probing Flow" can be found in the **Chapter 14 "Appendix"**.

LogiCORE IP Integrated Logic Analyzer (ILA) v6.2 core

The LogiCORE IP Integrated Logic Analyzer (ILA) core is a customizable logic analyzer core that can be used to monitor the internal signals of a design. The ILA core includes many advanced features of modern logic analyzers, including boolean trigger equations, and edge transition triggers. Because the ILA core is synchronous to the design being monitored, all design clock constraints that are applied to your design are also applied to the components of the ILA core.

ILA core general features are:

- user-selectable number of probe ports and probe_width
- multiple probe ports, which can be combined into a single trigger condition
- AXI interface on ILA IP core to debug AXI IP cores in a system

The following illustration is a symbol of the ILA v6.2 core.

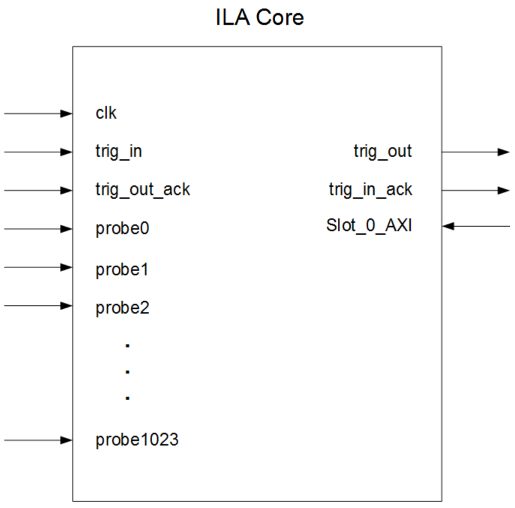


Figure 11.1: Symbol of the ILA v6.2 core

Signals in the FPGA design are connected to ILA core clock and probe inputs. These signals, attached to the probe inputs, are sampled at design speed and stored using on-chip block RAM (BRAM). The core parameters specify the number of probes, trace sample depth, and the width for each probe input. Communication with the ILA core is conducted using an auto-instantiated debug core hub that connects to the JTAG interface of the FPGA.

Note: If you want to read and learn more about the ILA v6.2 core, please refer to *"LogiCORE IP Integrated Logic Analyzer (ILA) v6.2 Product Guide".*

LogiCORE IP Virtual Input/Output (VIO) v3.0 core

The LogiCORE IP Virtual Input/Output (VIO) core is a customizable core that can both monitor and drive internal FPGA signals in real time. The number of width of the input and output ports are customizable in size to interface with the FPGA design. Because the VIO core is synchronous to the design being monitored and/or driven, all design clock constraints that are applied to your design are also applied to the components inside the VIO core. Run time interaction with this core requires the use of the Vivado logic analyzer feature. Unlike the ILA core, no on-chip or off-chip RAM is required.

VIO core general features are:

- provides virtual LEDs and other status indicators through input ports
- includes optional activity detectors on input ports to detect rising and falling transitions between samples
- provides virtual buttons and other controls indicators through output ports
- includes custom output initialization that allows you to specify the value of the VIO core outputs immediately following device configuration and start-up
- run time reset of the VIO core to initial values

The following illustration is a symbol of the VIO v3.0 core.

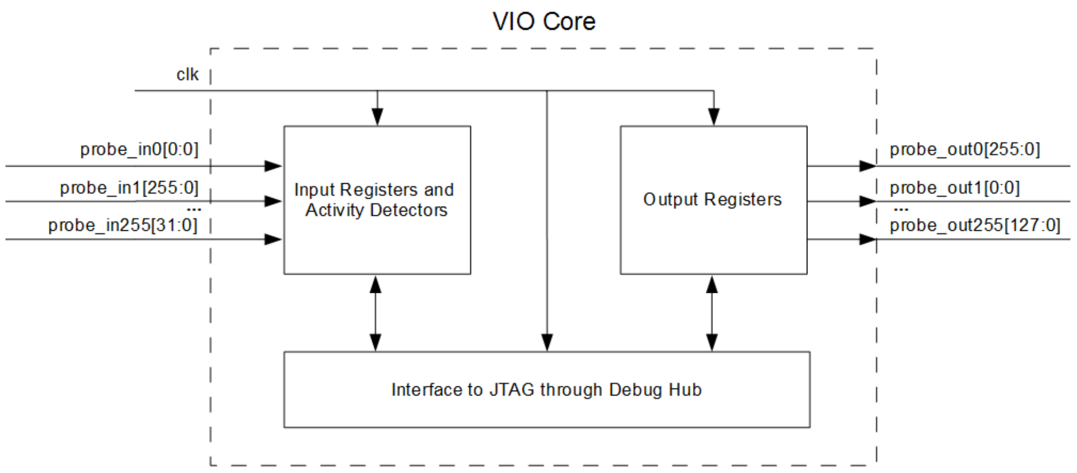


Figure 11.2: Symbol of the VIO v3.0 core

Note: If you want to read and learn more about the VIO v3.0 core, please refer to *"LogiCORE IP Virtual Input/Output (VIO) v3.0 Product Guide"*.

Insertion of debug cores in the Vivado tool is presented in a layered approach to address different needs of the diverse group of Vivado users:

- The highest level is a simple wizard that creates and configures Integrated Logic Analyzer (ILA) cores automatically based on the selected set of nets to debug

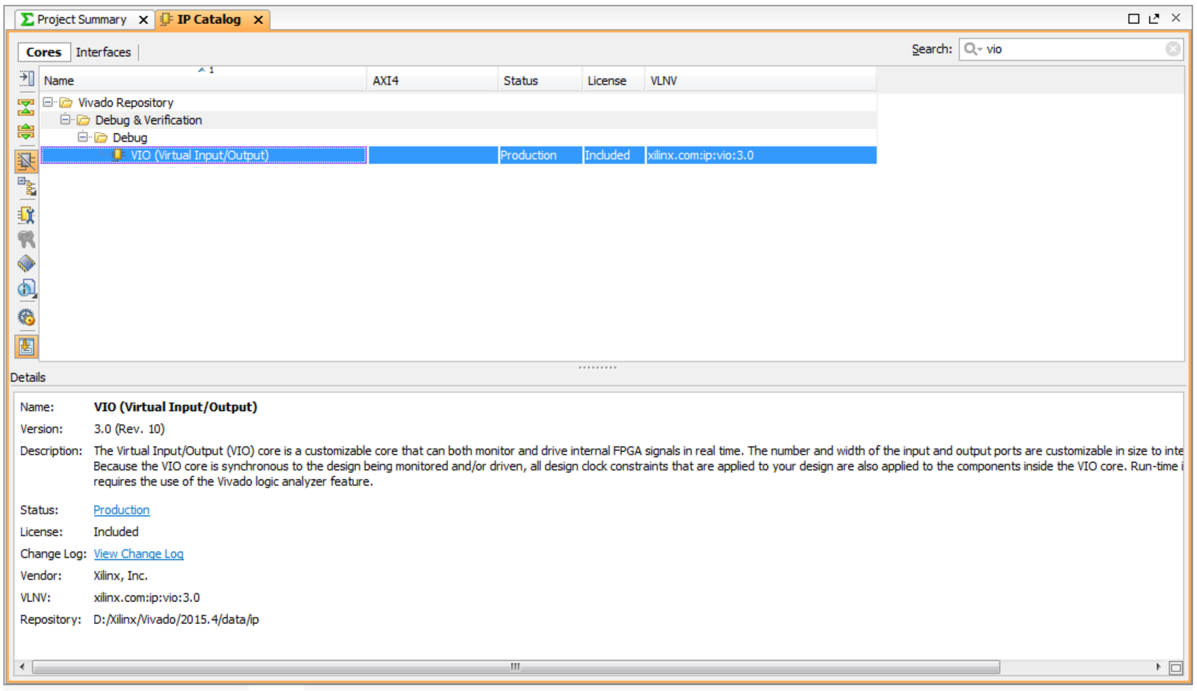
- The next level is the main Debug window allowing control over individual debug cores, ports and their properties

- The lowest level is the set of Tcl debug commands that you can enter manually or replay as a script

Netlist insertion debug probing flow can be used to insert ILA cores only. If you need the VIO core, like in our design, it must be inserted using the following steps:

Step 1. In the Vivado **Flow Navigator**, under the **Project Manager**, click the **IP Catalog** command

Step 2. In the **IP Catalog** window, in the **Search** field, search for the **VIO (Virtual Input/Output)** IP core. After you selected the VIO core, in the **Details** window, under the main IP Catalog window, you will find all the necessary information about selected IP core, see Illustration 11.3

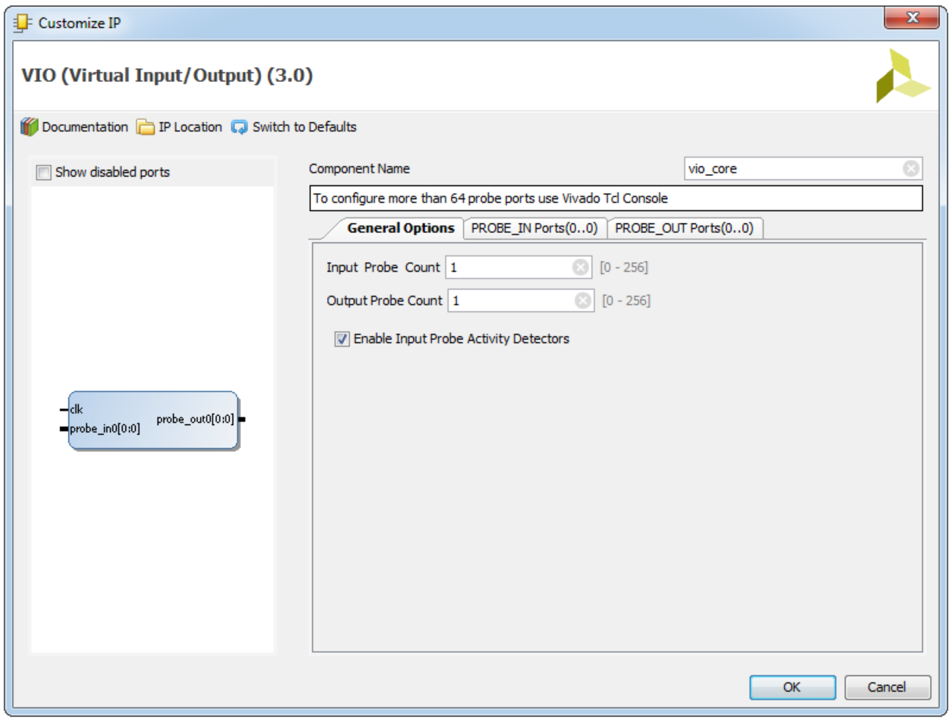


Name: VIO (Virtual Input/Output)
Version: 3.0 (Rev. 10)
Description: The Virtual Input/Output (VIO) core is a customizable core that can both monitor and drive internal FPGA signals in real time. The number and width of the input and output ports are customizable in size to inte
Because the VIO core is synchronous to the design being monitored and/or driven, all design clock constraints that are applied to your design are also applied to the components inside the VIO core. Run-time i
requires the use of the Vivado logic analyzer feature.
Status: Production
License: Included
Change Log: View Change Log
Vendor: Xilinx, Inc.
VLNV: xilinx.com:ip:vio:3.0
Repository: D:/Xilinx/Vivado/2015.4/data/ip

Figure 11.3: IP Catalog window with selected VIO core

Step 3. Double-click on the **VIO (Virtual Input/Output)** IP core and Vivado IDE will create a new skeleton source for your VIO core

The window that will be opened is used to set up the general VIO core parameters, see Illustration 11.4



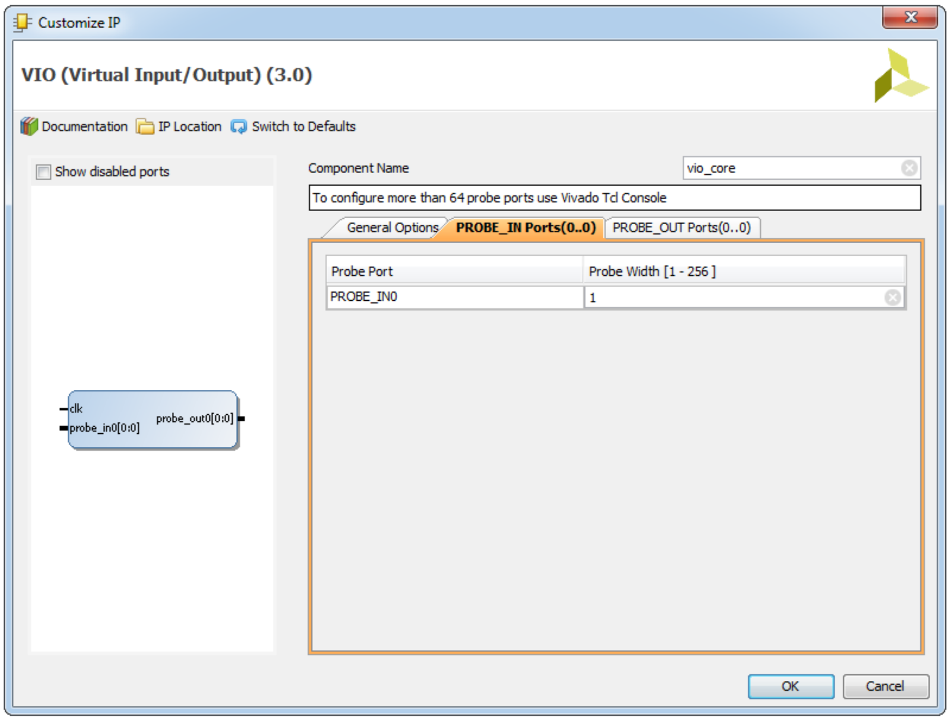
Component Name: vio_core
To configure more than 64 probe ports use Vivado Tcl Console
Input Probe Count: 1 [0 - 256]
Output Probe Count: 1 [0 - 256]
[✓] Enable Input Probe Activity Detectors

Figure 11.4: VIO core configuration window - General Options

Step 4. In the **VIO (Virtual Input/Output) (3.0)** window, enter *vio_core_name* (**vio_core**) in the **Component Name** field

Step 5. In the **General Options** tab, leave **Input Probe Count** to be **1** and **Output Probe Count** also to be **1**, because we will need one input probe for pwm_out signal and one output probe for sw0 signal, see Illustration 11.4

Step 6. In the **PROBE_IN Ports(0..0)** tab leave Probe Width of the **PROBE_IN0** Probe Port to be 1, because our pwm_out signal is 1 bit signal, see Illustration 11.5



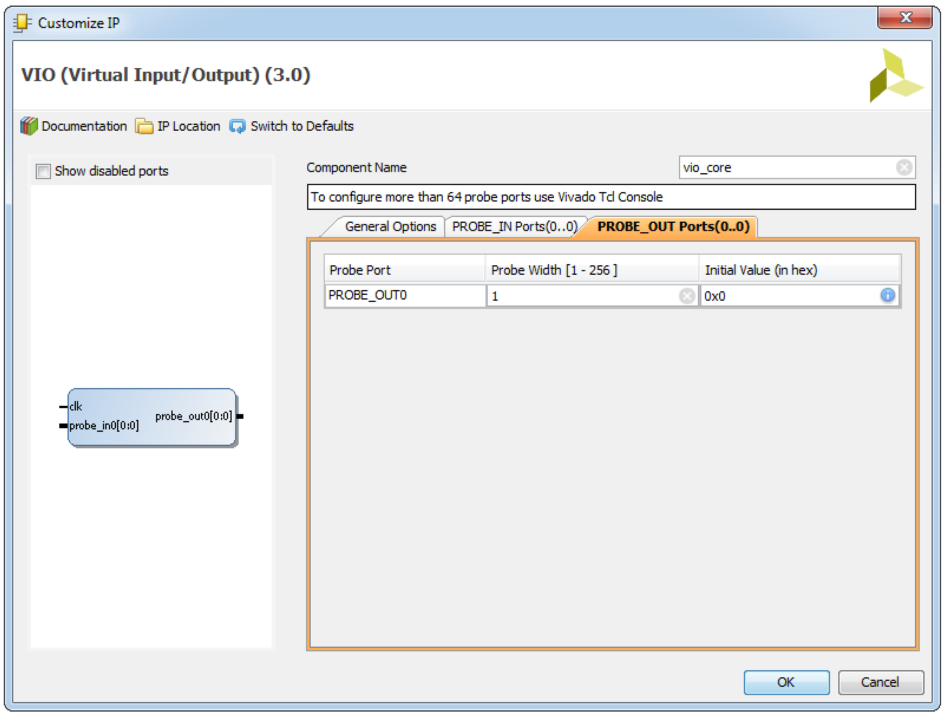
Component Name: vio_core

To configure more than 64 probe ports use Vivado Tcl Console

Probe Port	Probe Width [1 - 256]
PROBE_IN0	1

Figure 11.5: VIO core configuration window - PROBE_IN Ports(0..0) tab

Step 7. In the **PROBE_OUT Ports(0..0)** tab, leave Probe Width of the **PROBE_OUT0** Probe Port to be 1, because our sw0 signal is also 1 bit signal, see Illustration 11.6



Component Name: vio_core

To configure more than 64 probe ports use Vivado Tcl Console

Probe Port	Probe Width [1 - 256]	Initial Value (in hex)
PROBE_OUT0	1	0x0

Figure 11.6: VIO core configuration window - PROBE_OUT Ports(0..0) tab

Step 8. Click **OK**

Step 9. In the **Generate Output Products** window click **Generate**, see Illustration 11.7

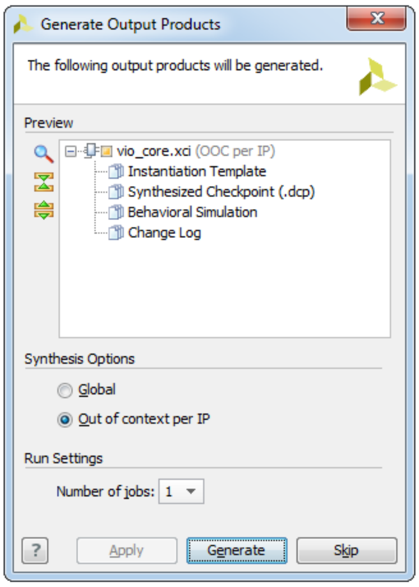


Figure 11.7: Generate Output Products window for VIO core

Note: After VIO core generation, your VIO core should appear in the Sources window, see Illustration 11.8

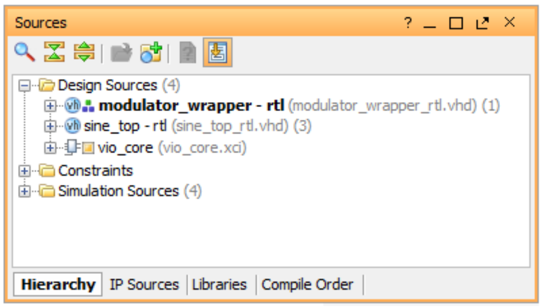


Figure 11.8: Sources tab with generated VIO core

The first step in inserting the ILA core into our design is to add debug nets to the project. Following are some of the methods how to add debug nets using the Vivado IDE:

- Add **mark_debug** attribute to the target **XDC** file

```
set_property mark_debug true [get_nets sine_ampl_s*]
set_property mark_debug true [get_nets freq_trig_s*]
```

Note: Use these attributes in synthesized design only! Do not use them with pre-synthesis or elaborated design nets.

- Add **mark_debug** attribute to **HDL** files

```
VHDL:

attribute mark_debug : string;
attribute keep: string;
attribute mark_debug of sine_ampl_s : signal is "true";
attribute mark_debug of freq_trig_s : signal is "true";

Verilog:

(* mark_debug *) wire sine_ampl_s;
(* mark_debug *) wire freq_trig_s;
```

- Right-click and select **Mark Debug** or **Unmark Debug** on Synthesis netlist
- Use **Tcl prompt** to set the **mark_debug** attribute. For example:

```
set mark_debug true [get_nets sine_ampl_s*]
set mark_debug true [get_nets freq_trig_s*]
```

This applies the *mark_debug* on the current, open netlist.

In this tutorial we will use only the second method of adding debug nets. The following steps will show you how to add debug nets to your HDL file (**modulator_rtl.vhd**) and how to synthesize your design using Vivado IDE.

Step 10. Open the existing **modulator_rtl.vhd** source file and add the following code lines into the architecture of the modulator design:

```
attribute  mark_debug : string;
attribute  keep : string;

attribute  mark_debug of sine_ampl_s : signal is "true";
attribute  mark_debug of freq_trig_s : signal is "true";
```

Now, your **modulator_rtl.vhd** source file should look like the code bellow:

```
...
architecture rtl of modulator is

    attribute mark_debug : string;
    attribute keep : string;

    -- amplitude counter
    signal ampl_cnt_s : std_logic_vector(design_setting_g.depth-1 downto 0);
    -- current amplitude value of the sine signal
    signal sine_ampl_s : std_logic_vector(design_setting_g.width-1 downto 0);
    -- signal which frequency depends on the sw0 state
    signal freq_trig_s : std_logic := '0';

    attribute mark_debug of sine_ampl_s : signal is "true";
    attribute mark_debug of freq_trig_s : signal is "true";

begin
...
```

Step 11. **Save** the **modulator_rtl.vhd** source file with new changes

After configuring and generating the VIO core, we should make a new module (**modulator_vio_rtl.vhd**) where we will connect the existing design (modulator_rtl.vhd) with the VIO core (see Figure 11.9).

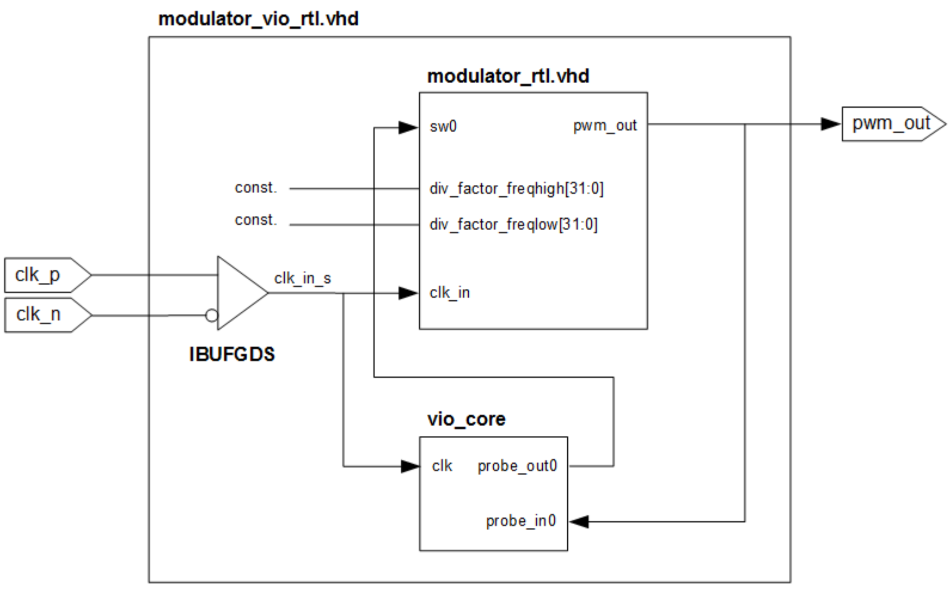


Figure 11.9: Connection between VIO core and Modulator module

As you can see from the picture above (Figure 11.9), we have to connect only Modulator module with the VIO core, because ILA core will be inserted later, in the design netlist.

To create a ***modulator_vio_rtl.vhd*** module, use steps for creating modules, **Chapter 2.4.1 Creating a Module Using Vivado Text Editor** .

modulator_vio_rtl.vhd:

```
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 modulator_vio 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
        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 rtl of modulator_vio is

    signal clk_in_s : std_logic;
    signal pwm_s    : std_logic_vector (0 downto 0);
    signal sw0_s    : std_logic_vector (0 downto 0);

    -- c1_c = fclk/(2^depth*2^width)                   - c1_c = 95.3674, fclk = 100 MHz
    constant c1_c : real := get_board_info_f(board_name_g).fclk/(real((2**design_setting_g.depth)*(2**
      design_setting_g.width)));
     -- div_factor_freqhigh_c = (c1_c/f_high)*2^width  - threshold value of frequency a = 110592
    constant div_factor_freqhigh_c : integer := integer(c1_c/design_setting_g.f_high)*(2**design_setting_g.
      width);
    -- div_factor_freqlow_c  = (c1_c/f_low)*2^width    - threshold value of frequency b = 389120
    constant div_factor_freqlow_c  : integer := integer(c1_c/design_setting_g.f_low)*(2**design_setting_g.
      width);


    -- vio_core component definition
    component vio_core
        port (
            clk        : in std_logic;
            probe_in0  : in std_logic_vector (0 downto 0);
            probe_out0 : out std_logic_vector (0 downto 0)
                );
    end component;


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

    pwm_out <= pwm_s (0);

    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;


    -- modulator module instance
    modulator: entity work.modulator(rtl)
        generic map(
            design_setting_g => design_setting_g
            )

        port map(
            clk_in              => clk_in_s,
            sw0                 => sw0_s(0),
            div_factor_freqhigh => conv_std_logic_vector(div_factor_freqhigh_c, 32),
            div_factor_freqlow  => conv_std_logic_vector(div_factor_freqlow_c, 32),
            pwm_out             => pwm_s(0)
            );

    -- vio_core component instance
    vio: vio_core
        port map (
            clk         => clk_in_s,
            probe_in0   => pwm_s,
            probe_out0 => sw0_s
            );

end;
```

After we made a new VHDL module (**modulator_vio_rtl.vhd**), we must also modify the **modulator_rtl.xdc** file, because we don't have any more *sw0* port. The new content of the xdc file is shown in the code below.

modulator_vio.xdc file:

```
set_property PACKAGE_PIN Y9 [get_ports clk_p]
set_property PACKAGE_PIN T22 [get_ports pwm_out]

set_property IOSTANDARD LVCMOS33 [get_ports clk_p]
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]
```

After finishing with the modifications, we must return to the Vivado IDE and do the following:

Step 12. Remove ***modulator_wrapper_rtl.vhd*** source file from the design

Step 13. Add ***modulator_vio_rtl.vhd*** and ***modulator_vio.xdc*** files in the Modulator design with **Add Sources** option:

- ***modulator_vio_rtl.vhd*** as Design Source file, and
- ***modulator_vio.xdc*** as Constraints file

Step 14. Remove the old ***modulator.xdc*** file from the design

Step 15. In the **Sources** window, right-click on the ***modulator_vio_rtl.vhd*** file and select **Set as Top** option

Step 16. In **Project Manager**, click the **Project Settings** command, see Illustration 11.10

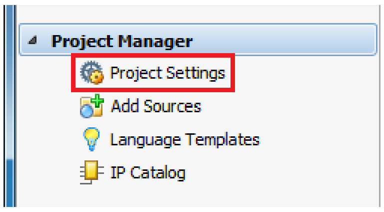


Figure 11.10: Project Settings command

Step 17. In the **Project Settings** dialog box, select **Synthesis** option from the left pane

Step 18. In the **Synthesis** window, change the **flatten_hierarchy** option from **rebuilt** to **none** as it is shown on the Illustration 11.11 and click **OK**

The reason for changing this setting to **none** is to prevent the synthesis tool from performing any boundary optimization for this tutorial.

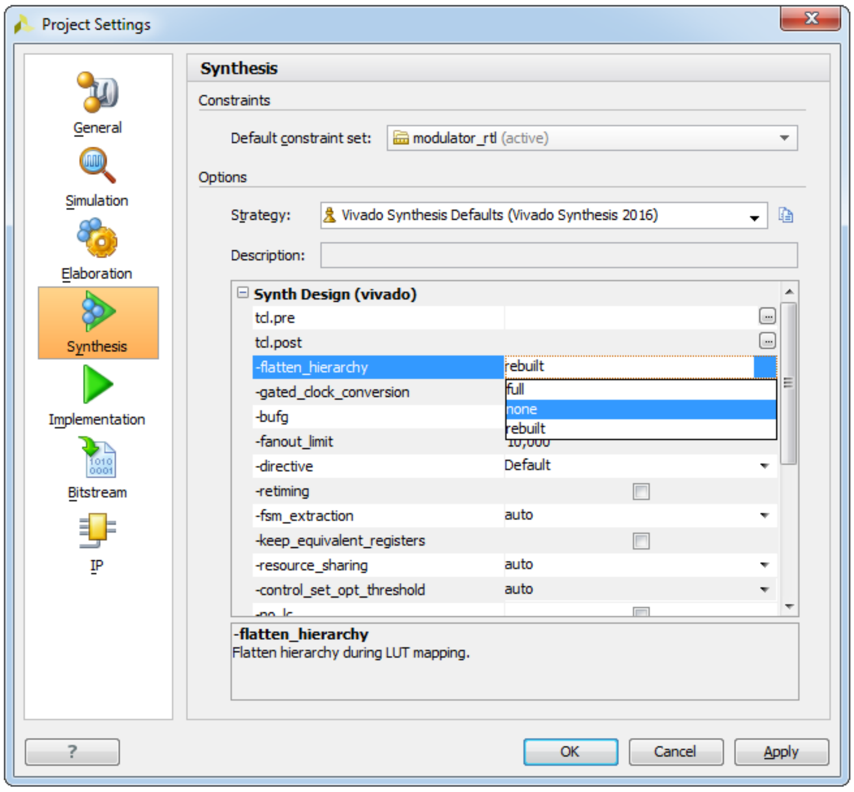


Figure 11.11: Project Settings dialog box

Step 19. In the Vivado **Flow Navigator**, click **Run Synthesis** command (**Synthesis** option) and wait for task to be completed, see Illustration 11.12

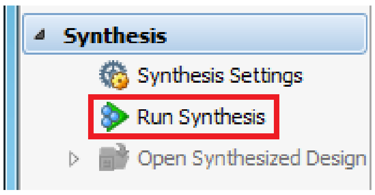
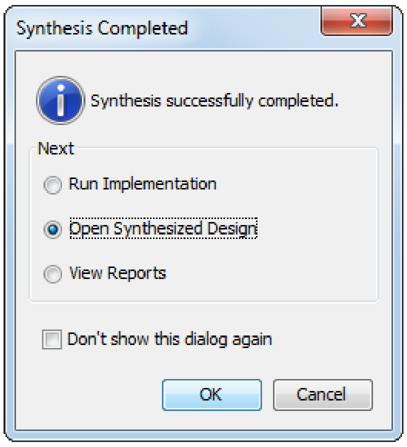


Figure 11.12: Run Synthesis command

Step 20. After the synthesis is completed, choose **Open Synthesized Design** option in the **Synthesis Completed** dialog box, see Illustration 11.13


Synthesis Completed

Synthesis successfully completed.

Next

( ) Run Implementation

(●) Open Synthesized Design

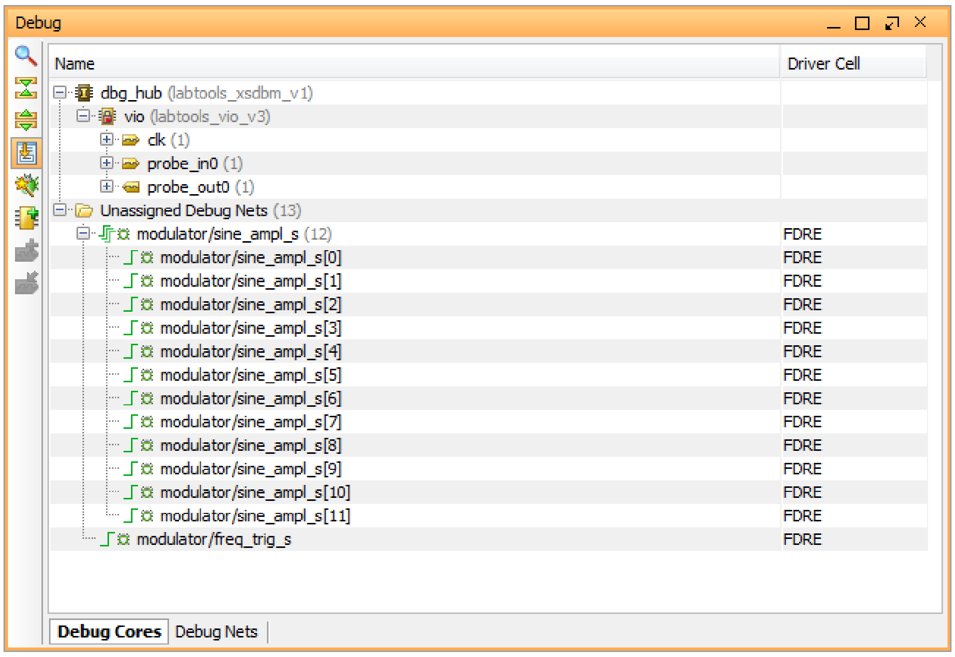
( ) View Reports

☐ Don't show this dialog again

Figure 11.13: Open Synthesized Design option

Step 21. Open **Debug Layout**, if it is not already opened

Step 22. In the **Debug** window, expand **dbg_hub** and **Unassigned Debug Nets** folders, if they are not already expanded. Illustration 11.14 shows assigned debug nets to the VIO core and debug nets that were marked in the **modulator_rtl.vhd** source file with ***mark_debug*** attributes and that we will assign to the ILA core.


Debug

Name	Driver Cell
dbg_hub (labtools_xsdbm_v1)	
vio (labtools_vio_v3)	
clk (1)	
probe_in0 (1)	
probe_out0 (1)	
Unassigned Debug Nets (13)	
modulator/sine_ampl_s (12)	FDRE
modulator/sine_ampl_s[0]	FDRE
modulator/sine_ampl_s[1]	FDRE
modulator/sine_ampl_s[2]	FDRE
modulator/sine_ampl_s[3]	FDRE
modulator/sine_ampl_s[4]	FDRE
modulator/sine_ampl_s[5]	FDRE
modulator/sine_ampl_s[6]	FDRE
modulator/sine_ampl_s[7]	FDRE
modulator/sine_ampl_s[8]	FDRE
modulator/sine_ampl_s[9]	FDRE
modulator/sine_ampl_s[10]	FDRE
modulator/sine_ampl_s[11]	FDRE
modulator/freq_trig_s	FDRE

Figure 11.14: Debug tab with unassigned debug nets

Step 23. Select the **Netlist** tab, beside **Sources** tab and expand **Nets** folders of the **modulator_vio** and **modulator** module, see Illustration 11.15

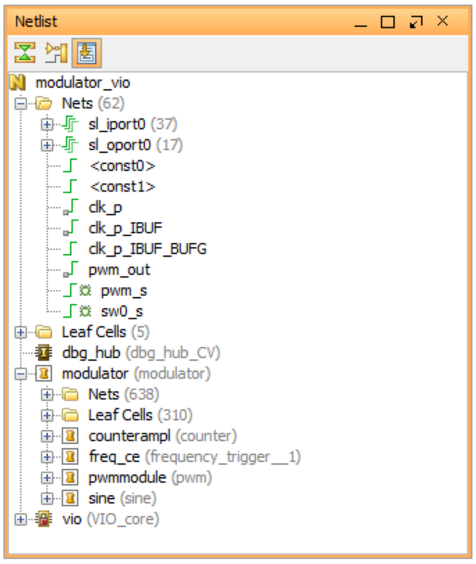


Figure 11.15: Netlist window with expanded Nets folders

In the expanded **Nets** folders you will find nets that exist in our design. Nets that we marked with ***mark_debug*** attributes are designated with green bug sign. These nets will be used to verify and debug our design.

If you are not satisfied with the marked nets and you want to mark some new or unmark some existing net, you have an opportunity to do that from the **Netlist** window in the following way:

- Select the net, right-click on it, and choose **Mark Debug** or **Unmark Debug** option, see Illustration 11.16

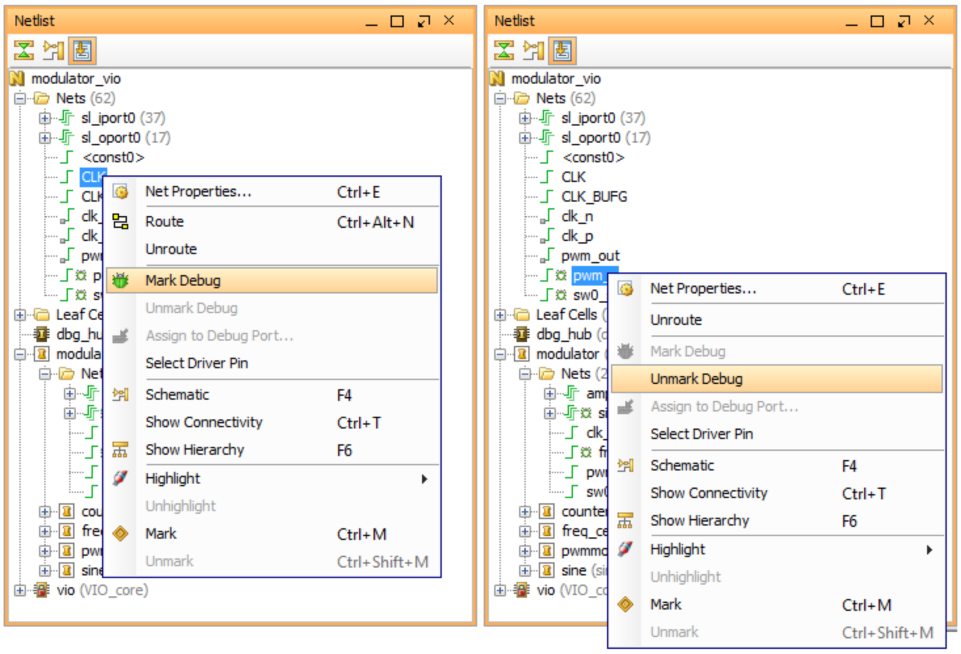
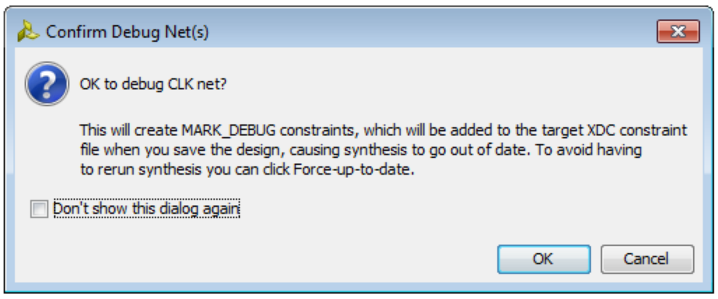


Figure 11.16: Mark and Unmark Debug option

- In the **Confirm Debug Net(s)** dialog box (in case of marking new debug net), click **OK**, see Illustration 11.17



Confirm Debug Net(s)

OK to debug CLK net?

This will create MARK_DEBUG constraints, which will be added to the target XDC constraint file when you save the design, causing synthesis to go out of date. To avoid having to rerun synthesis you can click Force-up-to-date.

Don't show this dialog again

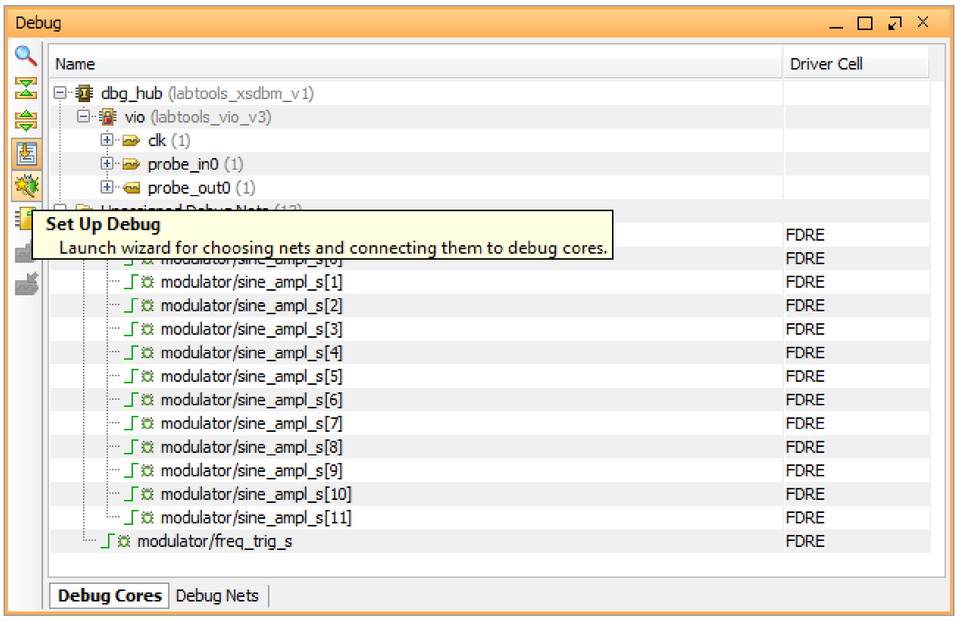
OK Cancel

Figure 11.17: Confirm Debug Net(s) dialog box

The next step after marking nets for debugging is to assign them to debug cores. The Vivado IDE provides **Set Up Debug** wizard to help guide you through the process of automatically creating the debug cores and assigning the debug nets to the inputs of the cores.

To use the **Set Up Debug** wizard to insert the debug cores, do the following:

Step 24. In the **Debug** window, select **Set Up Debug** button to launch the wizard, see Illustration 11.18



Debug

Name	Driver Cell
dbg_hub (labtools_xsdbm_v1)	
vio (labtools_vio_v3)	
clk (1)	
probe_in0 (1)	
probe_out0 (1)	
modulator/sine_ampl_s[1]	FDRE
modulator/sine_ampl_s[2]	FDRE
modulator/sine_ampl_s[3]	FDRE
modulator/sine_ampl_s[4]	FDRE
modulator/sine_ampl_s[5]	FDRE
modulator/sine_ampl_s[6]	FDRE
modulator/sine_ampl_s[7]	FDRE
modulator/sine_ampl_s[8]	FDRE
modulator/sine_ampl_s[9]	FDRE
modulator/sine_ampl_s[10]	FDRE
modulator/sine_ampl_s[11]	FDRE
modulator/freq_trig_s	FDRE

Set Up Debug
Launch wizard for choosing nets and connecting them to debug cores.

Debug Cores Debug Nets

Figure 11.18: Set Up Debug button

The another way to launch this wizard is to select **Tools** -> **Set up Debug...** option from the Vivado IDE main menu, see Illustration 11.19

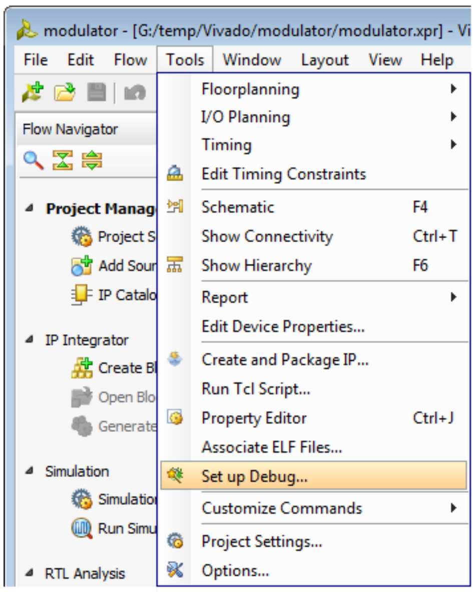
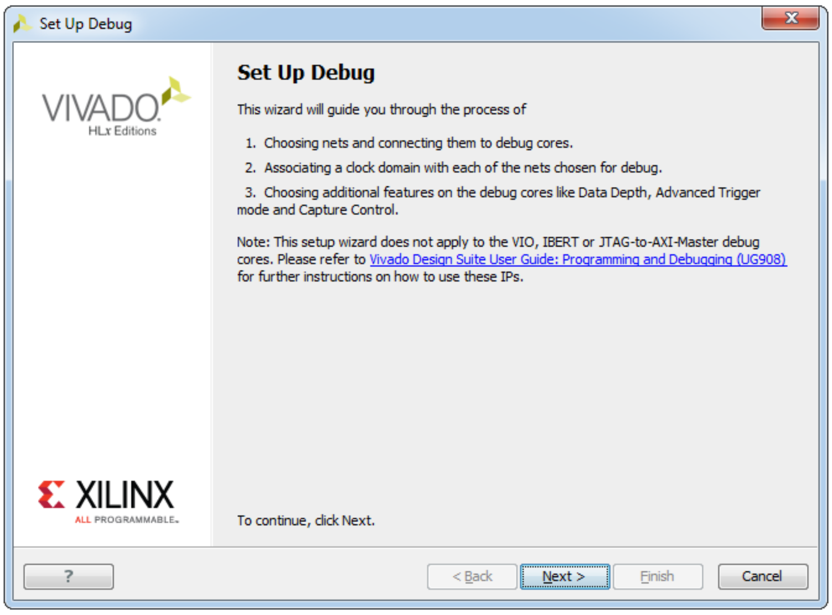


Figure 11.19: Tools -> Set up Debug option

Step 25. In the **Set Up Debug** dialog box, click **Next** to open **Nets to Debug** dialog box, see Illustration 11.20



Set Up Debug

This wizard will guide you through the process of

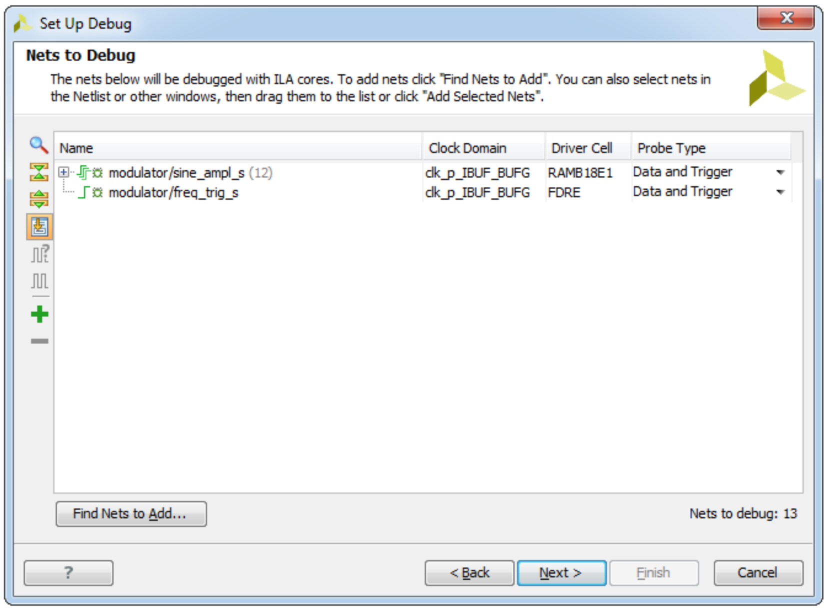
1. Choosing nets and connecting them to debug cores.
2. Associating a clock domain with each of the nets chosen for debug.
3. Choosing additional features on the debug cores like Data Depth, Advanced Trigger mode and Capture Control.

Note: This setup wizard does not apply to the VIO, IBERT or JTAG-to-AXI-Master debug cores. Please refer to Vivado Design Suite User Guide: Programming and Debugging (UG908) for further instructions on how to use these IPs.

To continue, click Next.

Figure 11.20: Set Up Debug dialog box

Step 26. In the **Nets to Debug** dialog box you will find nets that you have marked for debugging, see Illustration 11.21



Set Up Debug

Nets to Debug

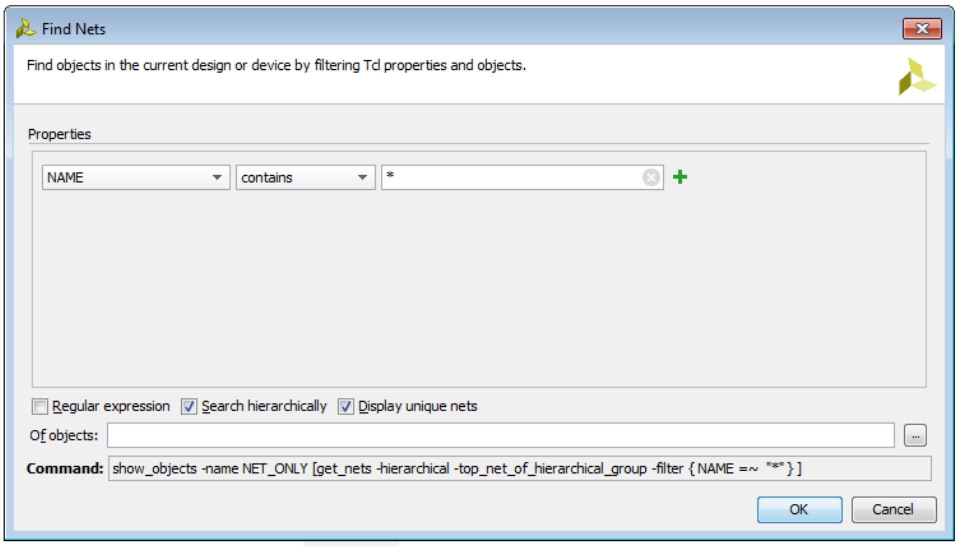
The nets below will be debugged with ILA cores. To add nets click "Find Nets to Add". You can also select nets in the Netlist or other windows, then drag them to the list or click "Add Selected Nets".

Name	Clock Domain	Driver Cell	Probe Type
modulator/sine_ampl_s (12)	clk_p_IBUF_BUFG	RAMB18E1	Data and Trigger
modulator/freq_trig_s	clk_p_IBUF_BUFG	FDRE	Data and Trigger

Find Nets to Add... Nets to debug: 13

Figure 11.21: Nets to Debug dialog box

In the **Nets to Debug** dialog box, you have also an opportunity to add more nets or remove existing nets from the table. Click **Find Nets to Add...** button to open **Find Nets** dialog box, see Illustration 11.22



Find Nets

Find objects in the current design or device by filtering Tcl properties and objects.

Properties

NAME contains *

☐ Regular expression ☑ Search hierarchically ☑ Display unique nets

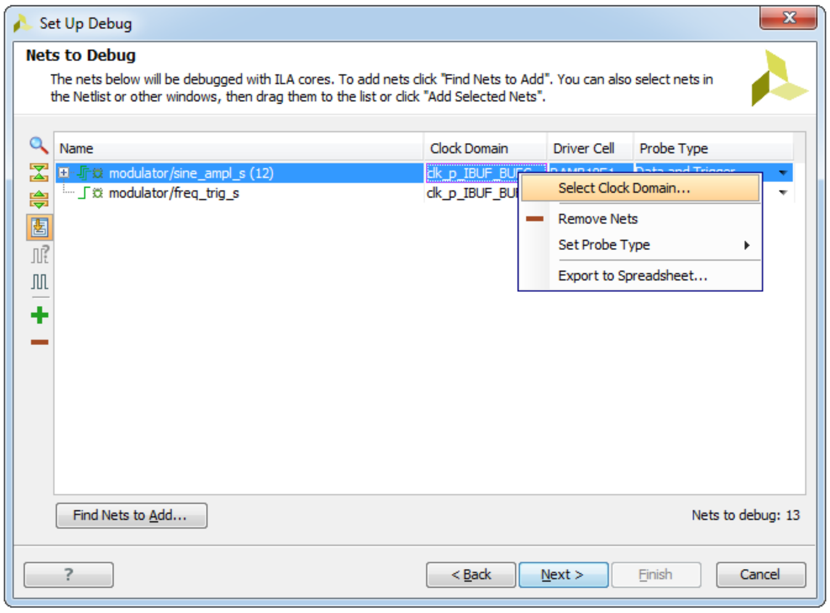
Of objects:

Command: show_objects -name NET_ONLY [get_nets -hierarchical -top_net_of_hierarchical_group -filter { NAME =~ "*" }]

Figure 11.22: Find Nets dialog box

Step 27. If you are satisfied with the debug net selection, click **OK**

Step 28. In the **Nets to Debug** dialog box, select target debug net, right-click on it and choose **Select Clock Domain...** option to change the clock domain that will be used to sample value on the net, see Illustration 11.23


Set Up Debug

Nets to Debug

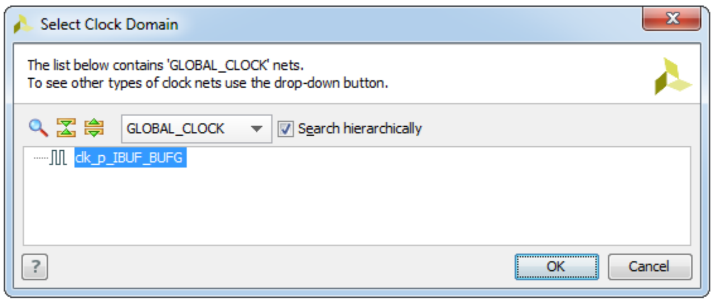
The nets below will be debugged with ILA cores. To add nets click "Find Nets to Add". You can also select nets in the Netlist or other windows, then drag them to the list or click "Add Selected Nets".

Name	Clock Domain	Driver Cell	Probe Type
modulator/sine_ampl_s (12)	clk_p_IBUF_BUFG		
modulator/freq_trig_s	clk_p_IBUF_BU		

Figure 11.23: Select Clock Domain option

Note: The **Set Up Debug** wizard attempts to automatically select the appropriate clock domain for the debug net by searching the path for synchronous elements.

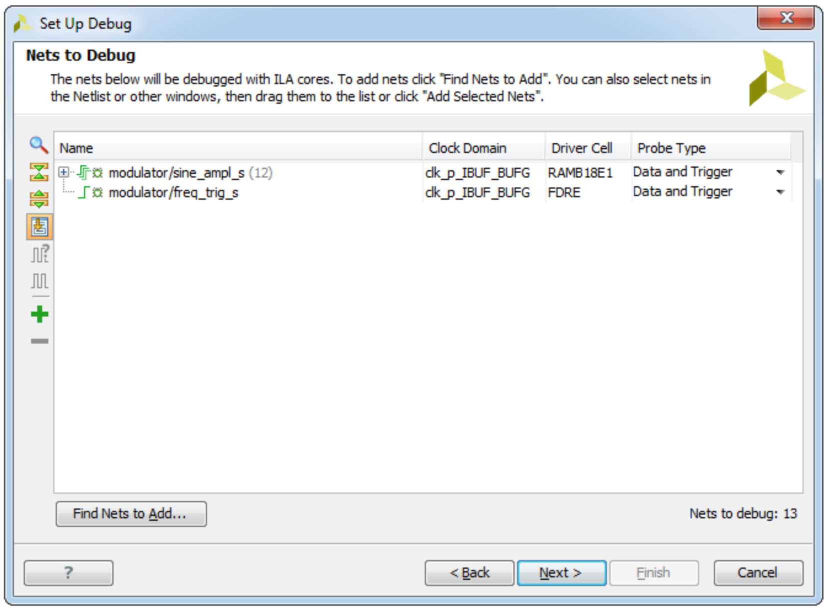
Step 29. In the **Select Clock Domain** dialog box modify clock domain as needed, see Illustration 11.24. Be aware that each clock domain present in the table results in a separate ILA v6.2 core instance.


Select Clock Domain

The list below contains 'GLOBAL_CLOCK' nets.
To see other types of clock nets use the drop-down button.

Figure 11.24: Select Clock Domain dialog box

Step 30. Select the same clock domain for ***freq_trig_s*** net, because signals captured by the same ILA core must have the same clock domain, Illustration 11.25


Set Up Debug

Nets to Debug

The nets below will be debugged with ILA cores. To add nets click "Find Nets to Add". You can also select nets in the Netlist or other windows, then drag them to the list or click "Add Selected Nets".

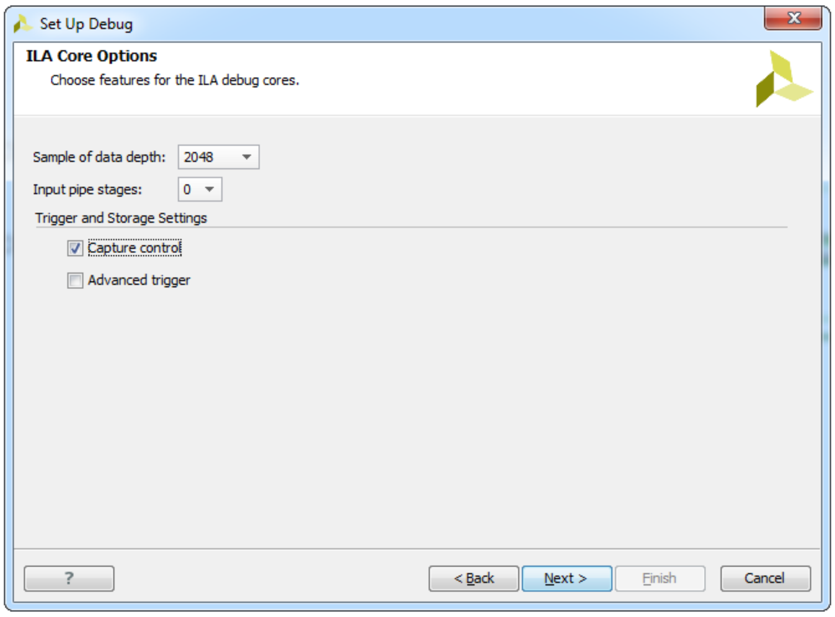
Name	Clock Domain	Driver Cell	Probe Type
modulator/sine_ampl_s (12)	clk_p_IBUF_BUFG	RAMB18E1	Data and Trigger
modulator/freq_trig_s	clk_p_IBUF_BUFG	FDRE	Data and Trigger

Nets to debug: 13

Figure 11.25: Nets to Debug dialog box - with specified clock domains

Step 31. Ones you are satisfied with the debug net selection, click **Next**

Step 32. In the **ILA Core Options** dialog box, set **Sample of data depth** option to **2048** value, enable **Capture control** option, leave all parameters unchanged and click **Next**, see Illustration 11.26


Set Up Debug

ILA Core Options

Choose features for the ILA debug cores.

Sample of data depth: 2048

Input pipe stages: 0

Trigger and Storage Settings

[✓] Capture control

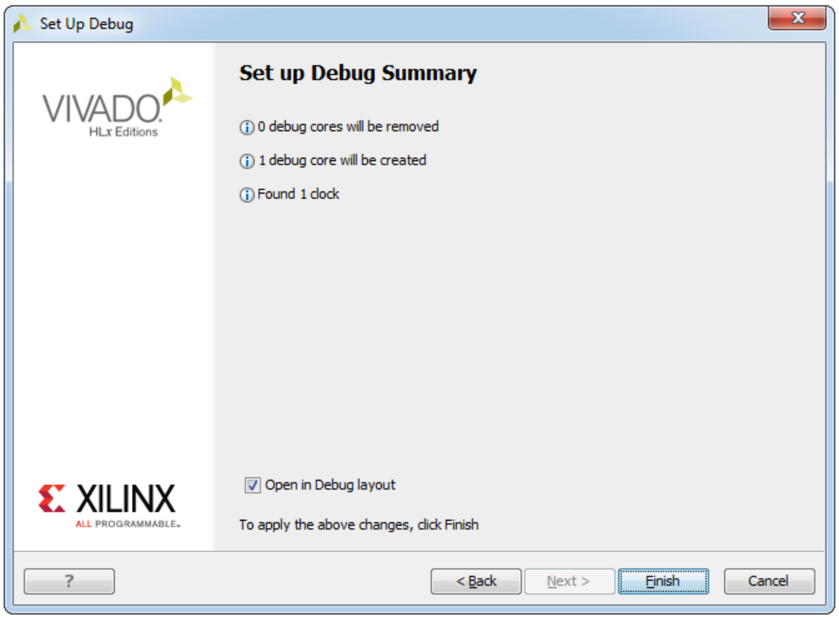
[ ] Advanced trigger

Figure 11.26: ILA Core Options dialog box

Important: **The Set Up Debug wizard inserts one ILA core per clock domain!**

The nets that were selected for debug are assigned automatically to the probe ports of the inserted ILA v6.2 cores. The last wizard screen shows the core creation summary displaying the number of clocks found and ILA cores to be created and/or removed, see Illustration 11.38

Step 33. If you are satisfied with the results, click **Finish** to insert and connect the ILA v6.2 cores in your synthesized design netlist, see Illustration 11.27


Set Up Debug

Set up Debug Summary

ⓘ 0 debug cores will be removed

ⓘ 1 debug core will be created

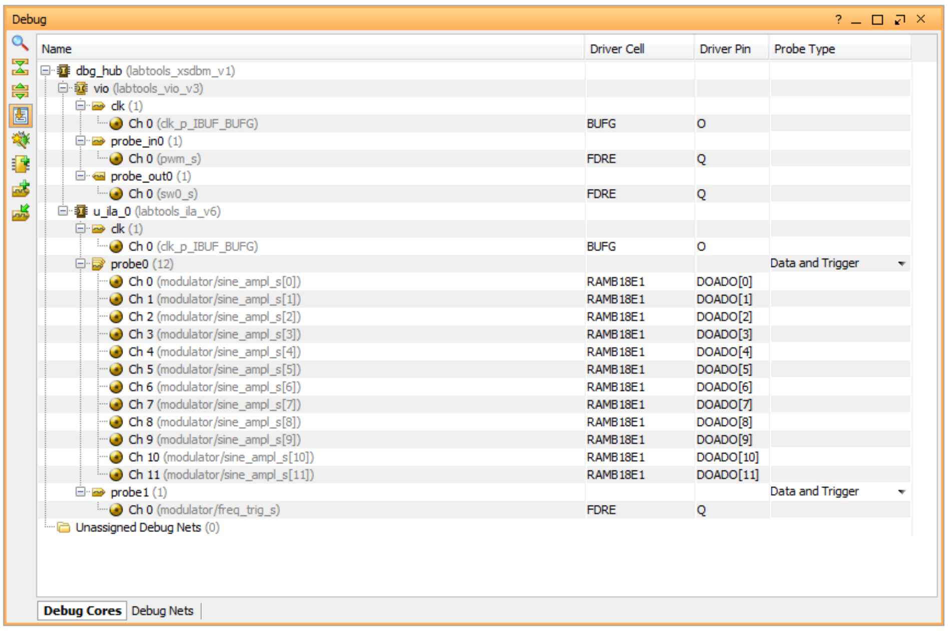
ⓘ Found 1 clock

[✓] Open in Debug layout

To apply the above changes, click Finish

Figure 11.27: Set up Debug Summary dialog box

Step 34. The debug nets are now assigned to the ILA v6.2 debug core, what you can see in the **Debug** window, see Illustration 11.28


Debug

Name	Driver Cell	Driver Pin	Probe Type
dbg_hub (labtools_xsdbm_v1)			
vio (labtools_vio_v3)			
clk (1)			
Ch 0 (clk_p_IBUF_BUFG)	BUFG	O	
probe_in0 (1)			
Ch 0 (pwm_s)	FDRE	Q	
probe_out0 (1)			
Ch 0 (sw0_s)	FDRE	Q	
u_ila_0 (labtools_ila_v6)			
clk (1)			
Ch 0 (clk_p_IBUF_BUFG)	BUFG	O	
probe0 (12)			Data and Trigger
Ch 0 (modulator/sine_ampl_s[0])	RAMB18E1	DOADO[0]	
Ch 1 (modulator/sine_ampl_s[1])	RAMB18E1	DOADO[1]	
Ch 2 (modulator/sine_ampl_s[2])	RAMB18E1	DOADO[2]	
Ch 3 (modulator/sine_ampl_s[3])	RAMB18E1	DOADO[3]	
Ch 4 (modulator/sine_ampl_s[4])	RAMB18E1	DOADO[4]	
Ch 5 (modulator/sine_ampl_s[5])	RAMB18E1	DOADO[5]	
Ch 6 (modulator/sine_ampl_s[6])	RAMB18E1	DOADO[6]	
Ch 7 (modulator/sine_ampl_s[7])	RAMB18E1	DOADO[7]	
Ch 8 (modulator/sine_ampl_s[8])	RAMB18E1	DOADO[8]	
Ch 9 (modulator/sine_ampl_s[9])	RAMB18E1	DOADO[9]	
Ch 10 (modulator/sine_ampl_s[10])	RAMB18E1	DOADO[10]	
Ch 11 (modulator/sine_ampl_s[11])	RAMB18E1	DOADO[11]	
probe1 (1)			Data and Trigger
Ch 0 (modulator/freq_trig_s)	FDRE	Q	
Unassigned Debug Nets (0)			

Figure 11.28: Debug window with assigned debug nets

The generated ILA core you can also find in the Netlist window, see Illustration 11.29

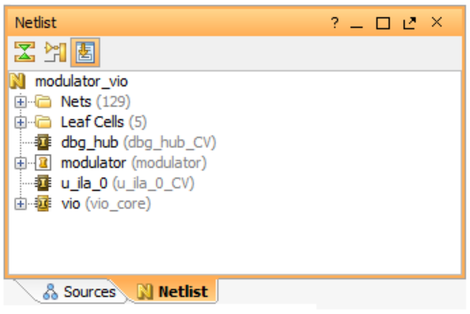


Figure 11.29: Netlist window with generated ILA core

Step 35. Implement your design with **Run Implementation** option from the **Flow Navigator** / **Implementation** (see **Sub-- Chapter 10.2.2 Run Implementation**)

Step 36. Generate bitstream file with **Generate Bitstream** option from the **Flow Navigator** / **Program and Debug** (see **Sub-Chapter 10.3 Generate Bitstream File**)

Step 37. Program your ZedBoard device (see **Sub-Chapter 10.4 Program Device**)

Note: All the information about the Vivado Netlist Instantiation Debug Probing Flow, such as its design flow and cores, how to generate, configure and instantiate some of them, as well as how to connect them with your existing design, you can also find in the **Lab 13: "Vivado Logic Analyzer"** .

11.2 Debug a Design using Integrated Vivado Logic Analyzer

Ones you have the debug cores in your design, you can use the run time logic analyzer features to debug the design in hardware. The Vivado logic analyzer feature is used to interact with new ILA, VIO, and JTAG-to-AXI Master debug cores that are in your design.

To access the Vivado logic analyzer feature:

Step 1. In the Vivado **Flow Navigator**, click the **Open Hardware Manager** command in the **Program and Debug** section, see Illustration 11.30

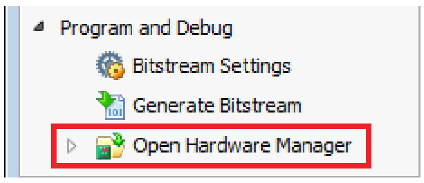
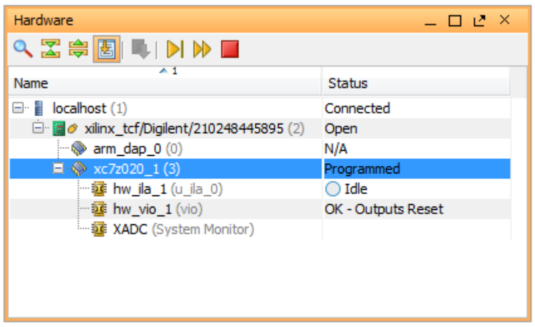


Figure 11.30: Open Hardware Manager command

Step 2. Repeat steps from the **Chapter 10.4 Program Device** to program your FPGA device with the .bit file

Step 3. After programming the FPGA device with the .bit file that contains the ILA v6.2 and VIO v3.0 cores, the **Hardware** window now shows the ILA and VIO cores that were detected after scanning the device, see Illustration 11.31

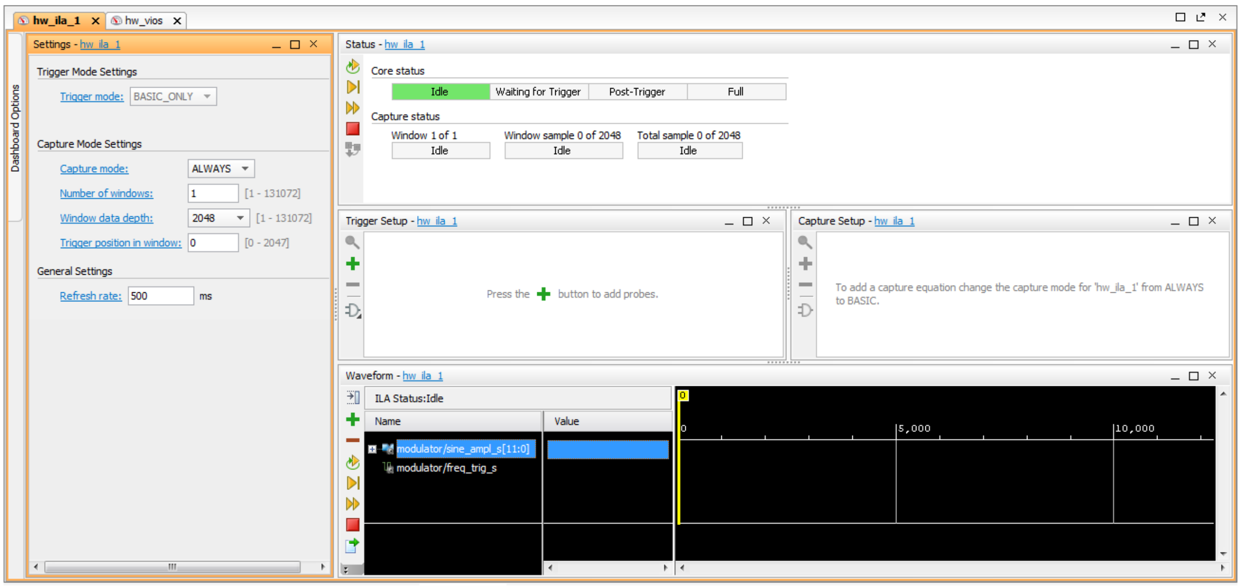


Hardware

Name	Status
localhost (1)	Connected
xilinx_tcf/Digilent/210248445895 (2)	Open
arm_dap_0 (0)	N/A
xc7z020_1 (3)	Programmed
hw_ila_1 (u_ila_0)	Idle
hw_vio_1 (vio)	OK - Outputs Reset
XADC (System Monitor)	

Figure 11.31: Hardware window showing the ILA and VIO debug cores

Step 4. The next step in design debugging process is to set up the ILA core. When the debug cores are detected upon refreshing a hardware device, the default dashboard for each debug core is automatically opened. The default **ILA Dashboard** can be seen on the Illustration 11.32



hw_ila_1 | hw_vios

Settings - hw_ila_1

Trigger Mode Settings

Trigger mode: BASIC_ONLY

Capture Mode Settings

Capture mode:	ALWAYS	
Number of windows:	1	[1 - 131072]
Window data depth:	2048	[1 - 131072]
Trigger position in window:	0	[0 - 2047]

General Settings

Refresh rate: 500 ms

Status - hw_ila_1

Core status

Idle	Waiting for Trigger	Post-Trigger	Full

Capture status

Window 1 of 1	Window sample 0 of 2048	Total sample 0 of 2048
Idle	Idle	Idle

Trigger Setup - hw_ila_1

Press the + button to add probes.

Capture Setup - hw_ila_1

To add a capture equation change the capture mode for 'hw_ila_1' from ALWAYS to BASIC.

Waveform - hw_ila_1

ILA Status:Idle

Name	Value
modulator/sine_ampl_s[11:0]	
modulator/freq_trig_s	

Figure 11.32: ILA Properties window

Every default dashboard contains windows relevant to the debug core the dashboard is created for. The default dashboard created for the ILA debug core contains five windows, as can be seen on the previous illustration:

- **Settings** window
- **Status** window
- **Trigger Setup** window
- **Capture Setup** window
- **Waveform** window

As you can see from the illustration above, **ILA Dashboard** is the central location for all status and control information of the ILA core. You can use the ILA Dashboard to interact with the ILA core in several ways:

- Use BASIC and ADVANCED trigger modes to trigger on various events in hardware
- Use ALLWAYS and BASIC capture modes to control filtering of the data to be captured
- Set the data depth of the ILA capture window

- Set the trigger position to any sample within the capture window
- Monitor the trigger and capture status of the ILA debug core

Step 5. In the **ILA Settings** window, under the **Capture Mode Settings**, configure the following parameters:

- set **Capture mode** to **BASIC**
- leave **Window data depth** on the **2048** value, and
- set **Trigger position in window** to **1000**

***Capture mode* -** selects what condition is evaluated before each sample is captured:

- ALWAYS: store a data sample during a given clock cycle regardless of any capture conditions
- BASIC: store a data sample during a given clock cycle only if the capture condition evaluates "true"

Data Depth - sets the data depth of the ILA core captured data buffer. You can set the data depth to any power of two from 1 to the maximum data depth.

Trigger Position - sets the position of the trigger mark in the captured data buffer. You can set the trigger position to any sample number in the captured data buffer. For instance, in the case of a captured data buffer that is 1024 sample deep:

- sample number 0 corresponds to the first (left- most) sample in the captured data buffer
- sample number 1023 corresponds to the last (right-most) sample in the captured data buffer
- sample numbers 511 and 512 correspond to the two "center" samples in the captured data buffer

Step 6. The next step will be to decide what ILA debug probes you want to participate in the trigger condition. Open **Debug Probes** window by clicking **Window ->** **Debug Probes** option from the main Vivado IDE menu to see all the probes corresponding to the ILA core.

Step 7. Go to the **Debug Probes** window, select the desired ILA debug probes (in our case it will be only the **freq_trig_s** debug probe), right-click on it and choose **Add Probes to Basic Capture Setup** option, see Illustration 11.33.

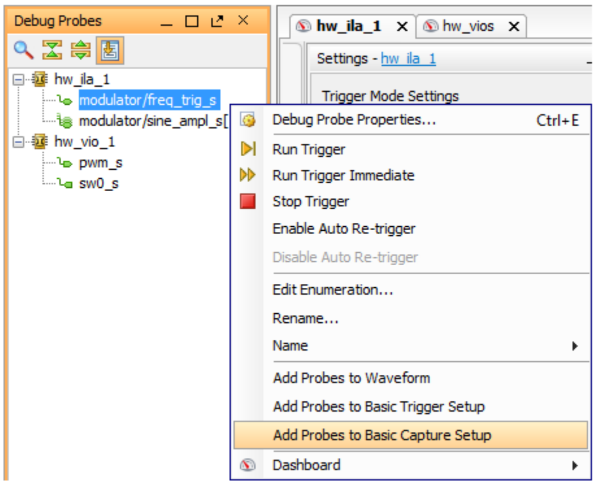


Figure 11.33: Add Probes to Basic Capture Setup option

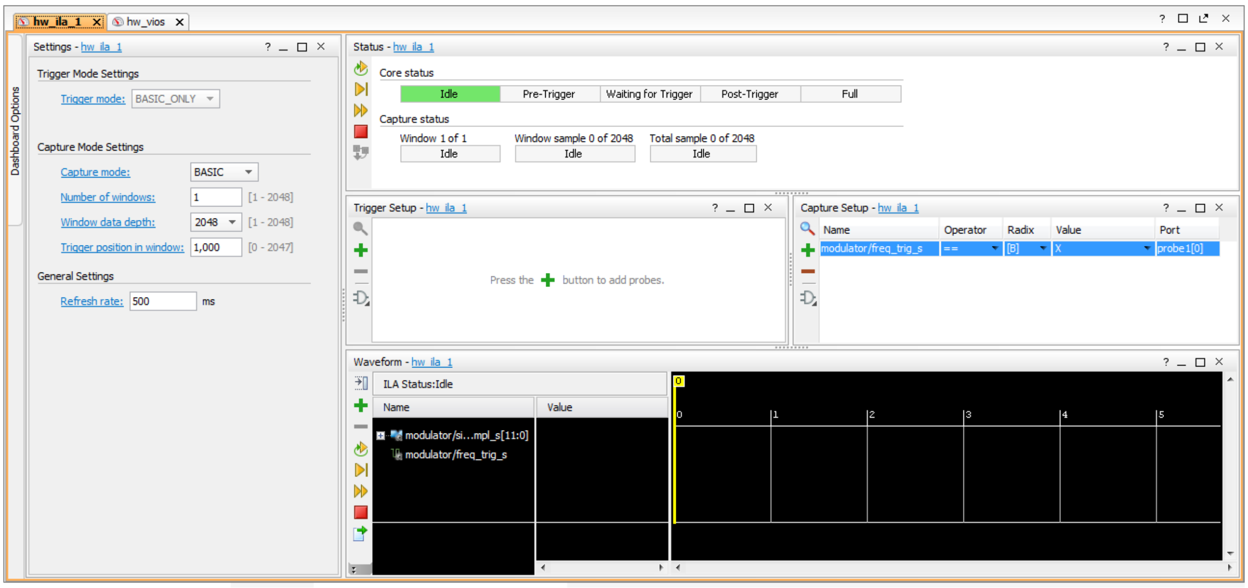
The another way to add debug probes to the **Basic Capture Setup** window is to drag and drop the probes from the Debug Probes window to the Basic Capture Setup window.

Important: Only probes that are in the **Basic Trigger Setup** or **Basic Capture Setup** window participate in the trigger condition. Any probes that are not in the window are set to "don't care" values and are not used as part of the trigger condition.

Note: If you want to remove probes from the **Basic Capture Setup** window, select the probe, right-click on it and choose **Remove** option.

The **Debug Probes** window contains information about the nets that you probed in your design using the ILA and/or VIO cores. This debug probe information is extracted from your design and stored in a data file that typically has an .ltx file extension. Normally, the ILA probe file is automatically created during implementation process. This file is automatically associated with the FPGA hardware device if the probes file is called *debug_nets.ltx* and is found in the same directory as the bitstream file that is associated with the device.

Step 8. Now, when the ILA debug probe **freq_trig_s** is in the **Basic Capture Setup** window, see Illustration 11.34, we can create trigger condition and debug probe compare values.


Settings - hw_ila_1
Trigger Mode Settings
Trigger mode: BASIC_ONLY
Capture Mode Settings
Capture mode: BASIC
Number of windows: 1 [1 - 2048]
Window data depth: 2048 [1 - 2048]
Trigger position in window: 1,000 [0 - 2047]
General Settings
Refresh rate: 500 ms

Status - hw_ila_1
Core status: Idle | Pre-Trigger | Waiting for Trigger | Post-Trigger | Full
Capture status
Window 1 of 1: Idle
Window sample 0 of 2048: Idle
Total sample 0 of 2048: Idle

Trigger Setup - hw_ila_1
Press the button to add probes.

Capture Setup - hw_ila_1

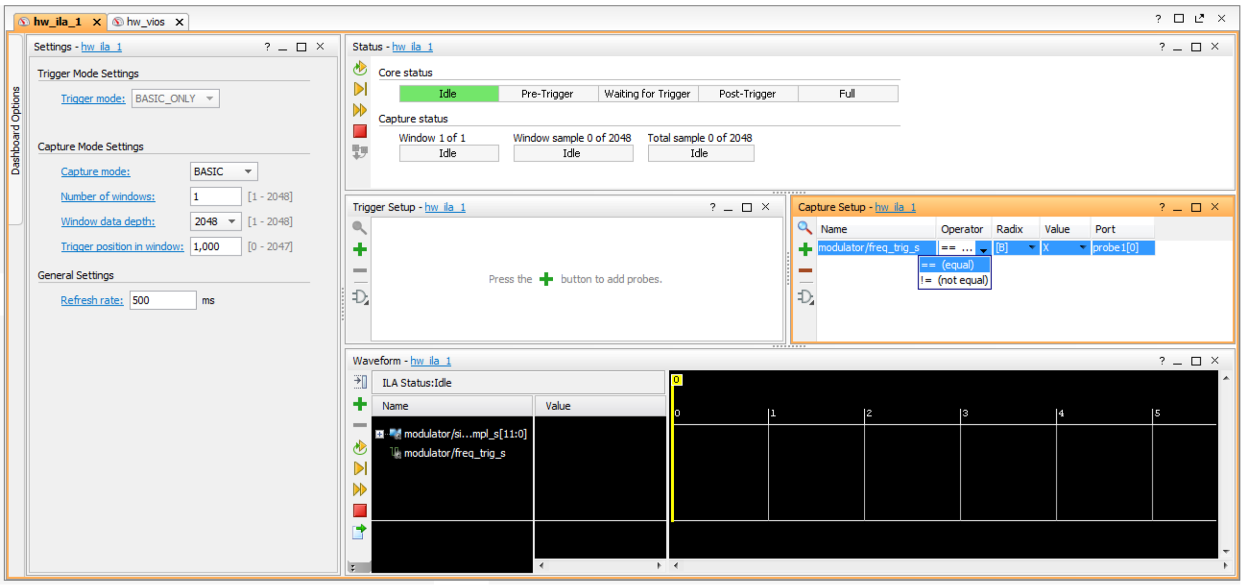
Name	Operator	Radix	Value	Port
modulator/freq_trig_s	==	[B]	X	probe1[0]

Waveform - hw_ila_1
ILA Status:Idle

Name	Value
modulator/si...mpl_s[11:0]	
modulator/freq_trig_s	

Figure 11.34: Basic Capture Setup window with the freq_trig_s debug probe

Step 9. In the **Basic Capture Setup** window, select the **Operator** cell in for a given ILA debug probe (freq_trig_s) to open the **Operator** dialog box. Select **== (equal)** option, as it is shown on the Illustration 11.35.


Settings - hw_ila_1
Trigger Mode Settings
Trigger mode: BASIC_ONLY
Capture Mode Settings
Capture mode: BASIC
Number of windows: 1 [1 - 2048]
Window data depth: 2048 [1 - 2048]
Trigger position in window: 1,000 [0 - 2047]
General Settings
Refresh rate: 500 ms

Status - hw_ila_1
Core status: Idle | Pre-Trigger | Waiting for Trigger | Post-Trigger | Full
Capture status
Window 1 of 1: Idle
Window sample 0 of 2048: Idle
Total sample 0 of 2048: Idle

Trigger Setup - hw_ila_1
Press the button to add probes.

Capture Setup - hw_ila_1

Name	Operator	Radix	Value	Port
modulator/freq_trig_s	== ...	[B]	X	probe1[0]

== (equal)
!= (not equal)

Waveform - hw_ila_1
ILA Status:Idle

Name	Value
modulator/si...mpl_s[11:0]	
modulator/freq_trig_s	

Figure 11.35: ILA probe Operator dialog box

The ILA probe trigger comparators are used to detect specific equality or inequality conditions on the probe inputs to the

ILA core. The trigger condition is the result of a Boolean "AND", "OR", "NAND", or "NOR" calculation of each of the ILA probe trigger comparator results.

Step 10. Repeat the same procedure with the **Radix** and **Value** cells and set its parameters on the following way:

- Radix: **[B] (Binary)**
- Value: **R (0-to-1 transition)**

As you can see from the illustration above, the **Basic Capture Setup** window contains three fields that you can configure:

- ***Operator***: This is the comparison operator that you can set to the following values:
  - **==** (equal)
  - **!=** (not equal)
  - $<$ (less then)
  - $<$**=** (less then or equal)
  - $>$ (greater than)
  - $>$**=** (greater than or equal)
- ***Radix***: This is the radix or base of the Value that you can set to the following values:
  - **[B]** Binary
  - **[H]** Hexadecimal
  - **[O]** Octal
  - **[A]** ASCII
  - **[U]** Unsigned Decimal
  - **[S]** signed Decimal
- ***Value***: This is the comparison value that will be compared (using the Operator) with the real-time on the nets(s) in the design that are connected to the probe input of the ILA debug core. Depending on the radix settings, the Value string is as follows:
  - ***Binary***
    - **0** : logical zero
    - **1** : logical one
    - **X** : don't care
    - **R** : rising or low-to-high transition
    - **F** : falling or high-to-low transition
    - **B** : either low- to-high or high-to-low transitions
    - **N** : no transition (current sample value is the same as the previousvalue)
  - ***Hexadecimal***
    - **X** : All bits corresponding to the value string character are "don't care" values
    - **0-9** : Values 0 through 9
    - **A-F** : values 10 through 15
  - ***Octal***
    - **X** : All bits corresponding to the value string character are "don't care" values
    - **0-7** : Values 0 through 7
  - ***ASCII***
    - Any string made up of ASCII characters
  - ***Unsigned Decimal***
    - Any non-negative integer value
  - ***Signed Decimal***
    - Any integer value

Step 11. After we set all the ILA core parameters, we can run or arming the ILA core trigger. We can run or arm the ILA core trigger in two different modes:

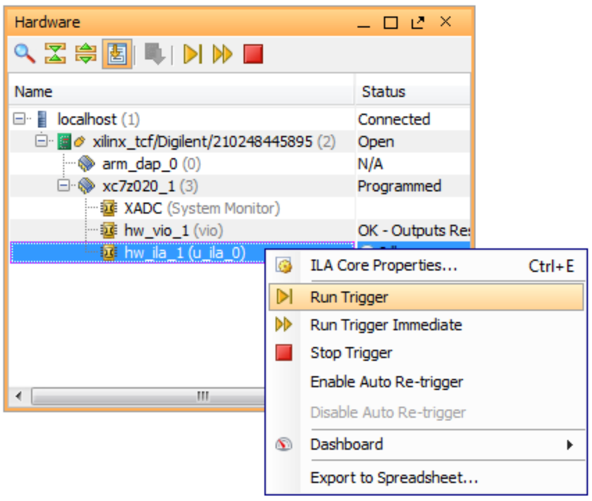
- ***Run Trigger mode*** - arms the ILA core to detect the trigger event that is defined by the ILA core trigger condition and probe compare values

  To run this mode, click the **Run Trigger** button in the **Hardware** or **Debug Probes** window.

- ***Run Trigger Immediate mode*** – arms the ILA core to trigger immediately regardless of the settings of the ILA core trigger condition and probe compare values. This command is useful for capturing any values present at the probe inputs of the ILA core.

  To run this mode, click the **Run Trigger Immediate** button in the **Hardware** or **Debug Probes** window.

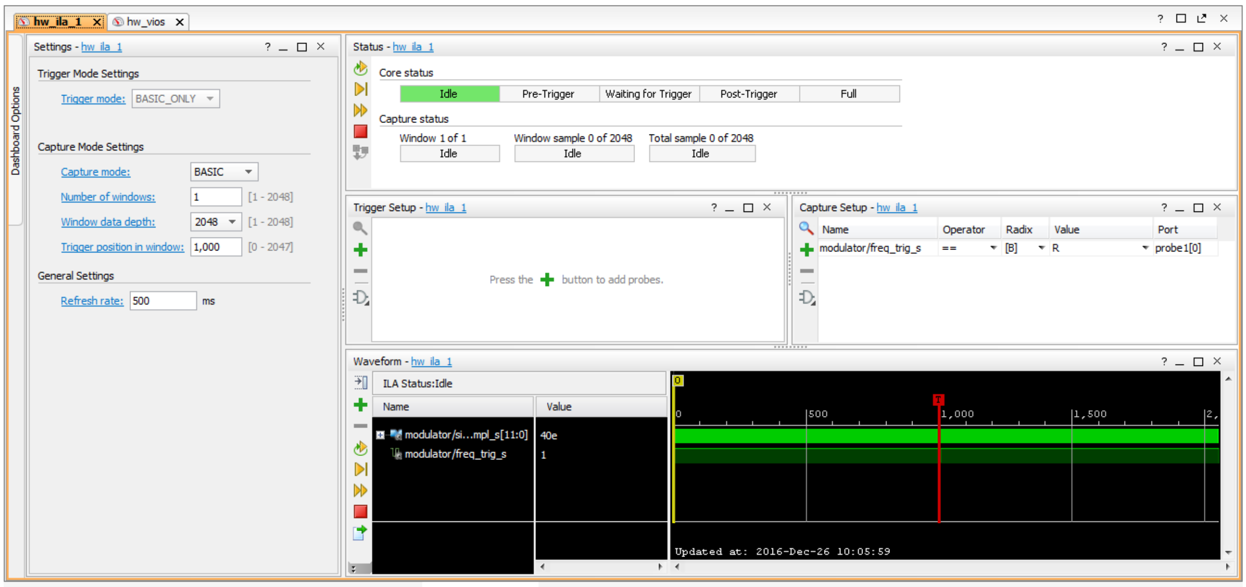
You can also arm the trigger by selecting and right-clicking on the ILA core (***hw_ila_1***) in the **Hardware** window and selecting **Run Trigger** or **Run Trigger Immediate** option from the popup menu, see Illustration 11.36



Name	Status
localhost (1)	Connected
xilinx_tcf/Digilent/210248445895 (2)	Open
arm_dap_0 (0)	N/A
xc7z020_1 (3)	Programmed
XADC (System Monitor)	
hw_vio_1 (vio)	OK - Outputs Re
hw_ila_1 (u_ila_0)	

Figure 11.36: Run Trigger option

Step 12. Once the ILA core captured data has been uploaded to the Vivado IDE, it is displayed in the **Waveform Viewer**, see Illustration 11.37



Settings - hw_ila_1

Trigger Mode Settings

Trigger mode: BASIC_ONLY

Capture Mode Settings

Capture mode: BASIC

Number of windows: 1 [1 - 2048]

Window data depth: 2048 [1 - 2048]

Trigger position in window: 1,000 [0 - 2047]

General Settings

Refresh rate: 500 ms

Status - hw_ila_1

Core status: Idle | Pre-Trigger | Waiting for Trigger | Post-Trigger | Full

Capture status

Window 1 of 1	Window sample 0 of 2048	Total sample 0 of 2048
Idle	Idle	Idle

Trigger Setup - hw_ila_1

Press the button to add probes.

Capture Setup - hw_ila_1

Name	Operator	Radix	Value	Port
modulator/freq_trig_s	==	[B]	R	probe1[0]

Waveform - hw_ila_1

ILA Status:Idle

Name	Value
modulator/si...mpl_s[11:0]	40e
modulator/freq_trig_s	1

```
Updated at: 2016-Dec-26 10:05:59
```

Figure 11.37: Content of the waveform window after trigger has been detected

Step 13. In the waveform window, select **sine_ampl_s[11:0]** probe port, right-click on it and select **Radix** -> **Unsigned Decimal** option to convert binary value to unsigned decimal

Now, when you click **Zoom Fit** option your waveform window should look the same as it is shown on the Illustration 11.38, where you can see debug probes and trigger position that we specified.

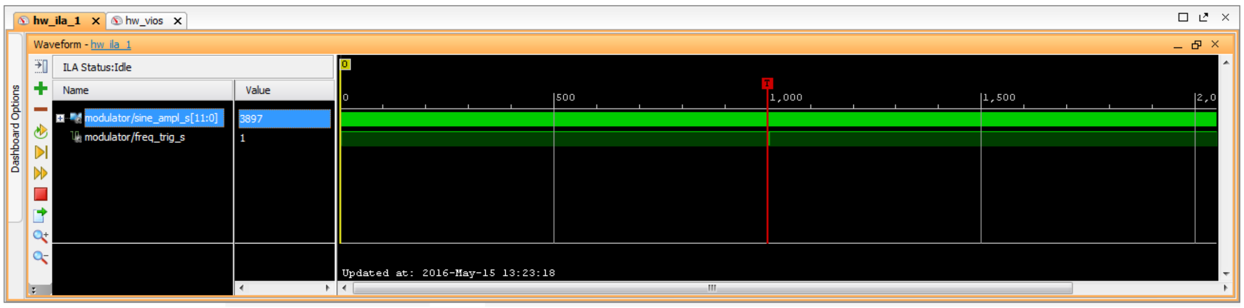


Figure 11.38: Waveform window with debug probes and specified trigger position

Step 14. Zoom In few times and you can see the first results, see Illustration 11.39

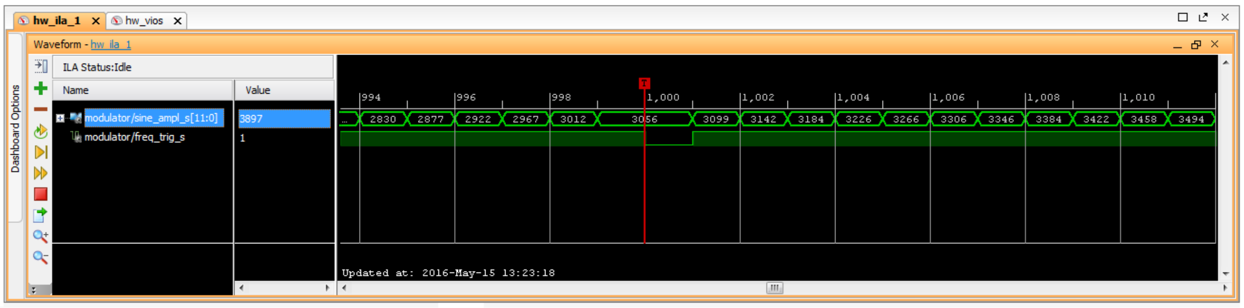


Figure 11.39: Zoomed in results in the waveform window

If you compare results obtained by the Vivado logic analyzer (Illustration 11.39) with the results obtained by the behavioral simulation of the PWM module (Illustration 11.40), you can see that the signals **sine_ampl_s** and **sine_out_s** have identical waveforms. This means that the implemented Modulator design in the FPGA is behaving in the same way as it was predicted by the simulation.

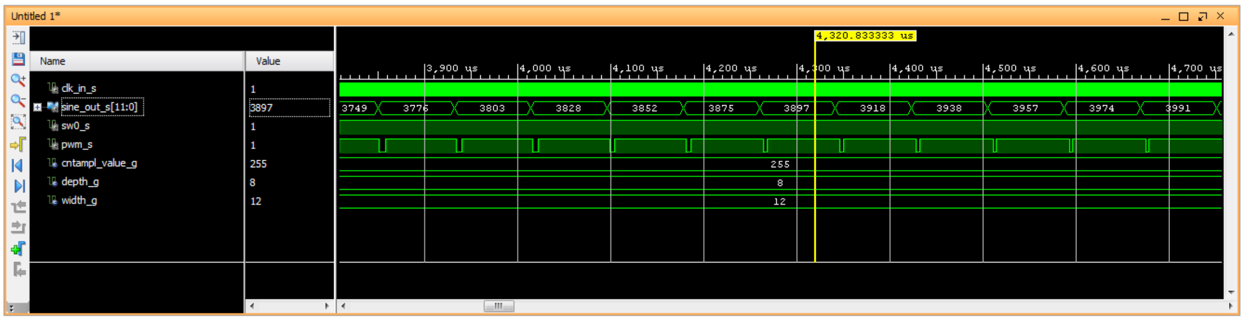


Figure 11.40: Results of the behavioral simulation of the PWM module

Note: To get results of the behavioral simulation of the PWM module, repeat steps from the **Sub-chapter 7.4 Simulating**.

If you would like to compare more result values from the Vivado logic analyzer with the results from the behavioral simulation of the PWM module, run the ILA core trigger as much as you need.

The ILA core can capture data samples when the core status is Pre-Trigger, Waiting for Trigger or Port-Trigger, see Illustration 11.35. As we already said, Capture mode selects what condition is evaluated before each sample is captured. Basic

Capture mode stores a data sample during a given clock cycle only if the capture condition evaluates "true". We used ***freq_trig_s*** signal to do the signal capturing.

Capture condition is a Boolean combination of events that is detected by match unit comparators that are attached to the trigger ports of the core. Only when this combination is detected, data will be stored in the ILA's buffer.

To be able to capture at least one period of the sine signal and to store it in the ILA buffer, we have to use capture condition feature. After triggering the ILA core, in the waveform viewer change the Waveform Style from Digital to Analog and your captured waveform should look like as the waveform on the Illustration 11.41

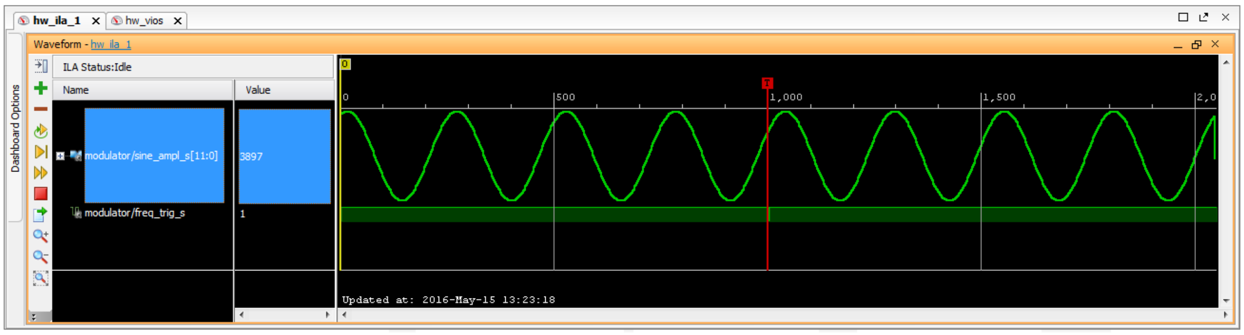


Figure 11.41: Captured waveform of the sine signal

From the illustration above we can see that data depth that we have selected for the ILA buffer is too big for this example. We can decrease the ILA buffer data depth from 131072 to 1024 and speed up the process of signal capturing. After decreasing ILA buffer data depth, your captured waveform of the sine signal should look like as the waveform on the Illustration 11.42.

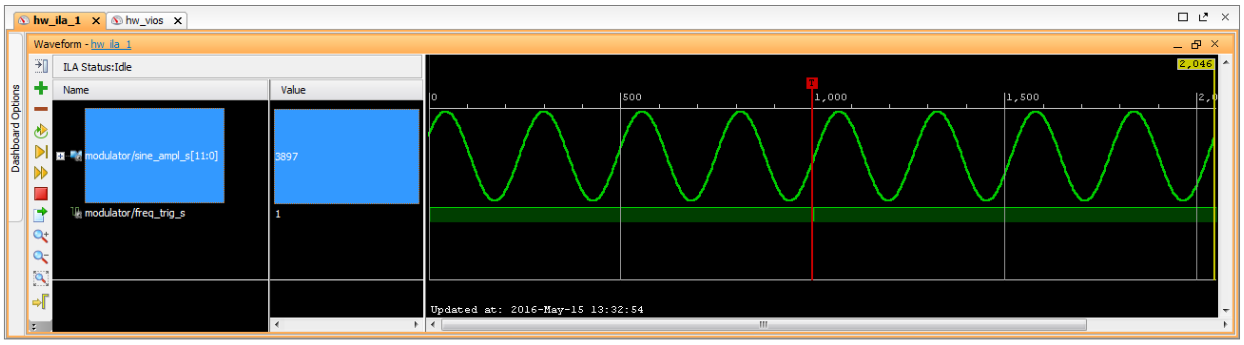


Figure 11.42: Captured waveform of the sine signal with 2048 ILA buffer data depth

Step 15. Go back to the **Debug Probes** window, select ***hw_vio_1***, right-click on it and choose **Add Probes to VIO Window** option, see Illustration 11.43

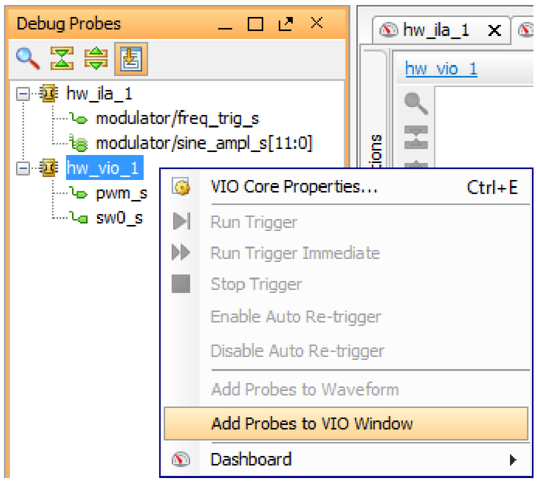
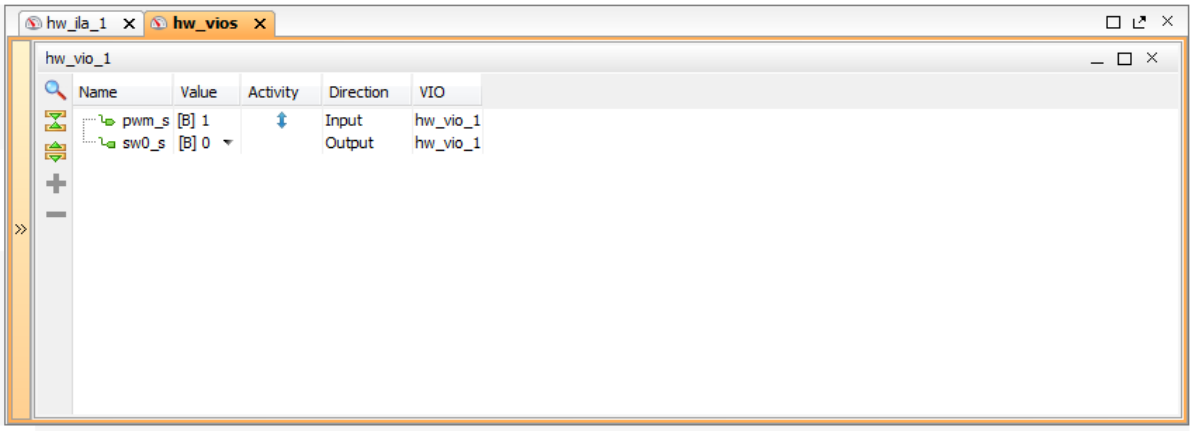


Figure 11.43: Add Probes to VIO Window option

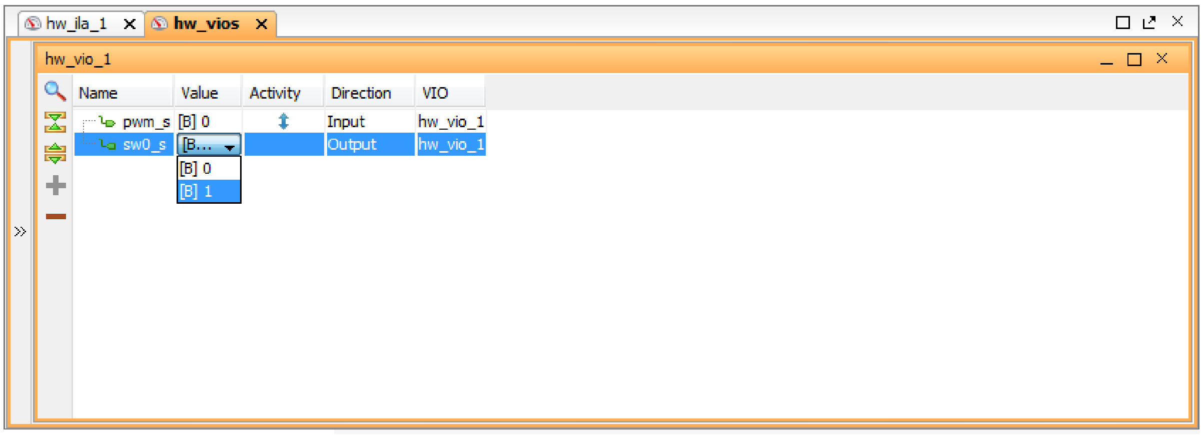
Step 15. In the **VIO Probes** window you will see two 1-bit probes, ***pwm_s*** and ***sw0_s***, see Illustration 11.45. ***pwm_s*** probe is actually connected to the ***pwm_out*** output port of the Modulator module, as can be seen on the Figure 11.9 and from the modulator_ila_vio_rtl.vhd source code. Similarly, ***sw0_s*** probe is connected to the **sw0** input port of the Modulator module.



Name	Value	Activity	Direction	VIO
pwm_s	[B] 1	⬍	Input	hw_vio_1
sw0_s	[B] 0		Output	hw_vio_1

Figure 11.44: VIO Probes window

In the **VIO Probes** window, you can observe the rate of change of the ***pwm_s*** signal. You can change the frequency of the ***pwm_s*** signal by changing the value of the ***sw0_s*** probe from 0 to 1 and from 1 to 0, see Illustration 11.45. The change in frequency of the ***pwm_s*** signal can be also observed on the development board. Now, ***sw0_s*** probe has taken the role of the switch sw0, present on the development board.



Name	Value	Activity	Direction	VIO
pwm_s	[B] 0	↕	Input	hw_vio_1
sw0_s	[B...		Output	hw_vio_1

Figure 11.45: Changing the sw0_s value

Note: All the information about debugging the design using the Vivado Logic Analyzer, such as how to configure and run it and how to analyze your design using this tool, you can also find in the **Lab 14: "Debug a Design using Integrated Vivado Logic Analyzer".**

11.3 Oscilloscope

An oscilloscope is a type of electronic instrument that creates a two- dimensional graph of one or more electrical potential differences. Typically horizontal, or x-axis, represents function of time and vertical, or y-axis, represents voltage.

To see the pwm signal on the oscilloscope, follow these steps:

Step 1. Connect the USB Connector to the Starter Kit Board Connector and to the PC

Step 2. Connect the oscilloscope's probe to some expansion connector on the Starter Kit Board (see Illustration 11.46)

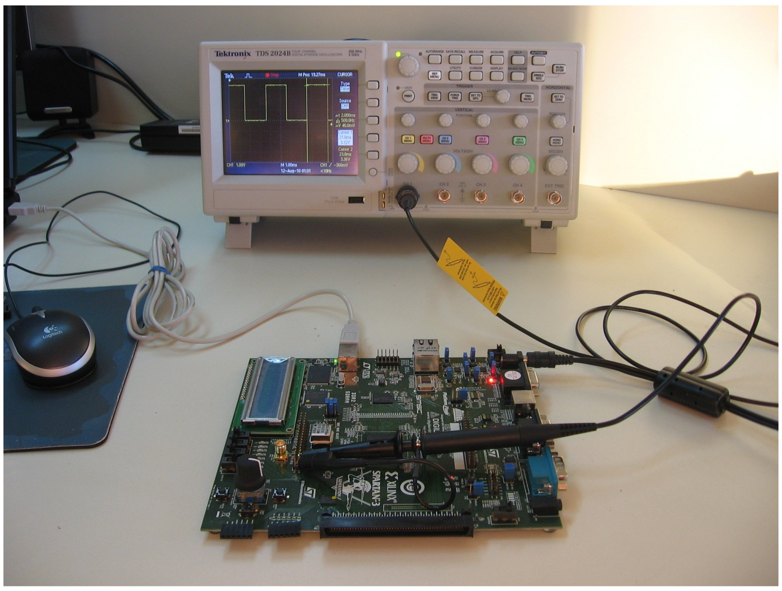


Figure 11.46: Using oscilloscope for viewing PWM signal

Step 3. Power on the ZedBoard development board

Step 4. In the ***modulator_wrapper_rtl.vhd*** file made the following modifications:

- add a new ***pwm_osc*** output port in the modulator entity declaration:

```
pwm_osc: out std_logic;
```

- in the architecture add a new temporary signal declaration:

```
signal temp_out_s: std_logic;
```

- in the port map of the pwm module (***pwmmodule***) connect the pwm_out port with the temp_out_s signal:

```
pwm_out => temp_out_s;
```

- at the and of the architecture connect the pwm_out and pwm_osc ports with the temp_out_s signal:

```
pwm_out <= temp_out_s;
pwm_osc <= temp_out_s;
```

Now, the **modulator_oscilloscope_rtl.vhd** source file should look like the code below.

modulator_oscilloscope_rtl.vhd:

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

    use work.modulator_pkg.all;

library unisim;
    use unisim.vcomponents.all;


entity modulator_oscilloscope 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
        pwm_osc : out std_logic    -- pulse width modulated signal for the oscilloscope
--        clk_en  : out std_logic    -- clock enable port used only for MicroZed board
        );
end entity;


architecture rtl of modulator_oscilloscope is

    -- input clock signal
    signal clk_in_s : std_logic;

    -- temporary signal
    signal temp_out_s  : std_logic;

    -- c1_c = fclk/(2^depth*2^width)                   - c1_c = 95.3674, fclk = 100 MHz
    constant c1_c : real :=
       get_board_info_f(board_name_g).fclk/(real((2**design_setting_g.depth)*(2**design_setting_g.width)));
     -- div_factor_freqhigh_c = (c1_c/f_high)*2^width  - threshold value of frequency a = 110592
    constant div_factor_freqhigh_c : integer :=
       integer(c1_c/design_setting_g.f_high)*(2**design_setting_g.width);
    -- div_factor_freqlow_c  = (c1_c/f_low)*2^width    - threshold value of frequency b = 389120
    constant div_factor_freqlow_c  : integer :=
       integer(c1_c/design_setting_g.f_low)*(2**design_setting_g.width);

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 (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 (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;



    pwmmodulator : entity work.modulator   -- modulator module instance
        generic map(
            design_setting_g => design_setting_g
            )

        port map(
            clk_in              => clk_in_s,
            sw0                 => sw0,
            div_factor_freqhigh => conv_std_logic_vector(div_factor_freqhigh_c, 32),
            div_factor_freqlow  => conv_std_logic_vector(div_factor_freqlow_c, 32),
            pwm_out             => temp_out_s
            );

        pwm_out <= temp_out_s;
        pwm_osc <= temp_out_s;

end;
```

Step 5. In the XDC file add location of the *pwm_osc* port. Location of the *pwm_osc* port should be chosen in such way to allow easy access for the oscilloscope's probe

Step 6. Return to the Flow Navigator and synthesize your design with **Run Synthesis** option from the **Flow Navigator** / **Synthesis** (see **Sub-chapter 6.5.2 Run Synthesis**)

Step 7. Implement your design with **Run Implementation** option from the **Flow Navigator** / **Implementation** (see **Sub--Chapter 10.2.2 Run Implementation**)

Step 8. Generate bitstream file with **Generate Bitstream** option from the **Flow Navigator** / **Program and Debug** (see **Sub-Chapter 10.3 Generate Bitstream File**)

Step 10. Program your ZedBoard device (see **Sub-Chapter 10.4 Program Device**)

Step 4. Configure the oscilloscope, and if your oscilloscope's settings are correct, you should see a pwm_out signal on the display, see Illustration 11.47

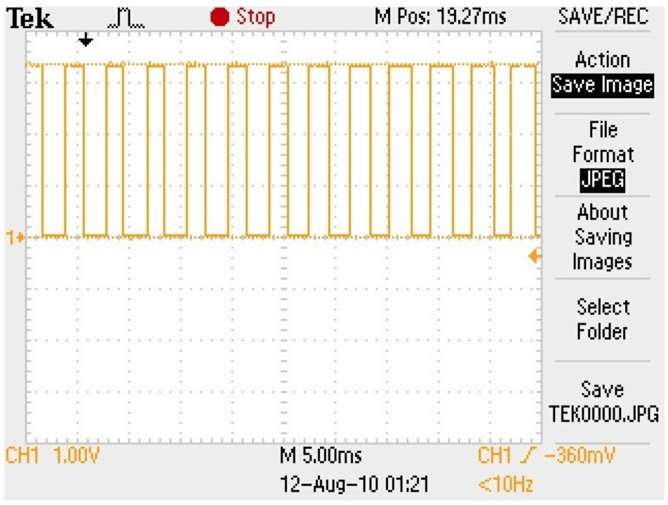


Figure 11.47: PWM signal measured by oscilloscope

Note: All the information about the Oscilloscope, how to use it and how to analyze your design on it, you can also find in the **Lab 15: "Oscilloscope"** .

Chapter 12

MODULATOR DESIGN TARGETING SOCIUS DEVELOPMENT BOARD

12.1 Description

- ***Usage***: This module will be used to target **socius** development board. Socius development board is a small, portable electronic device that can be easily powered, developed by the "so-logic" company. This module will be composed of two separate VHDL models:

  – ***modulator_socius_rtl.vhd*** model and

  – ***modulator_socius_clk_rtl.vhd*** model which will be the top model of the design

  The main component of the socius development board is ***Zynq-7000 AP SoC***. The Zynq-7000 family is based on the Xilinx All Programmable SoC (AP SoC) architecture. The Zynq-7000 AP SoC is composed of two major functional blocks: **Processing System (PS)** and **Programmable Logic (PL)**. Since existing LEDs and switches on the socius board are connected to the PS part of the Zynq FPGA, it would require programming PS part of the Zynq FPGA, which is not topic of this tutorial. It is the main topic in the *"Basic Embedded System Design"* tutorial.

  In our design we will program PL part of the Zynq FPGA with ***modulator_socius_rtl.vhd*** model. PS part is also required to generate clock signal for the Modulator design, since the only reference clock source on the socius board is connected to the PS part of the Zynq FPGA. Properly configured PS part is described in the ***socius_xz_lab_ps_-bd*** component. Both of these components, ***modulator_socius*** and ***socius_xz_lab_ps_bd***, will be contained in the ***modulator_socius_clk_rtl.vhd*** model, see block diagram below.

- ***Block diagram***:

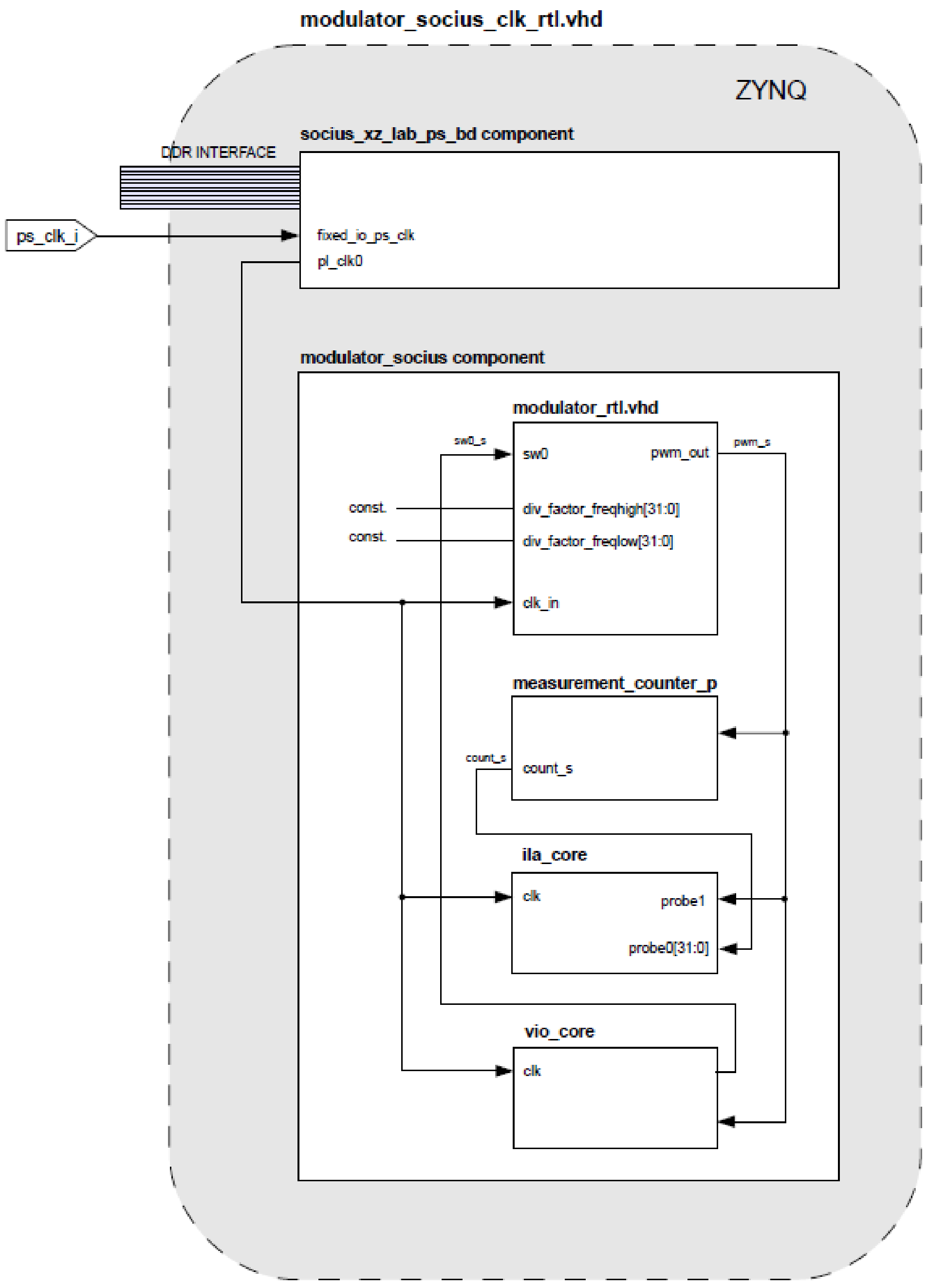


Figure 12.1: Modulator block diagram for socius development board

- ***Input ports***:
  - **ps_clk_i:** input clock signal from socius development board
- ***File name***: modulator_socius_clk_rtl.vhd

12.2 Creating Project

Our first step will be to create new project. The following steps describe how to create ARM-based hardware platform for socius development board:

Step 1. Launch the Vivado software:

Select **Start** -> **All Programs** -> **Xilinx Design Tools** -> **Vivado 2016.4** -> **Vivado 2016.4** and the Vivado **Getting Started** page will appear

Step 2. On the **Getting Started** page, choose **Create New Project** option

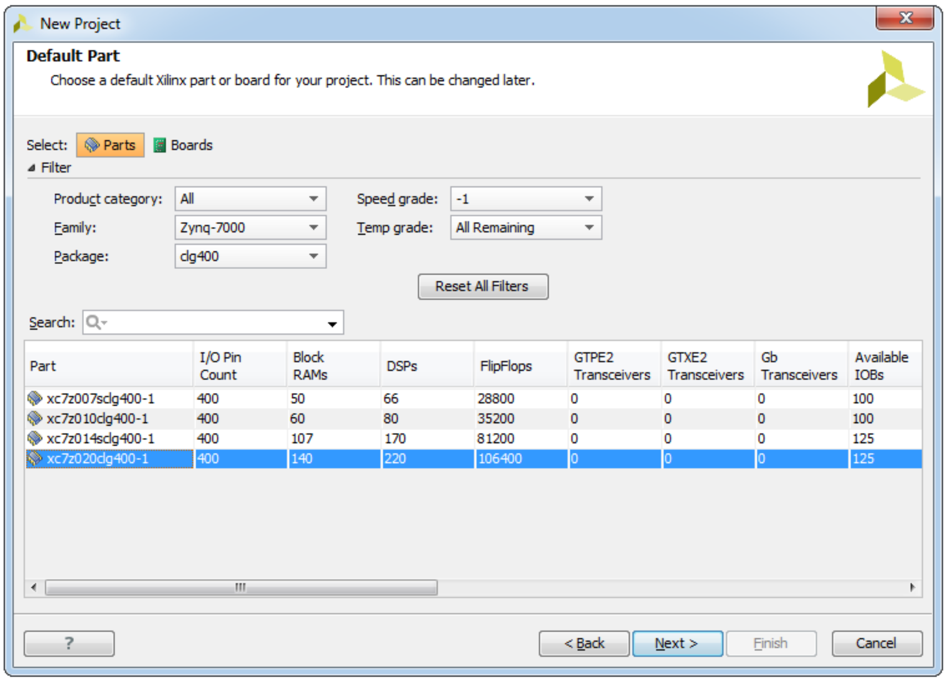
Step 3. In the **Create a New Vivado Project** dialog box, click **Next** and the wizard will guide you through the process of a new project creation

Step 4. In the **Project Name** dialog box specify the name and the location of the new project:

- In the **Project name** field type **modulator_socius** as the name of the project
- In the **Project location** field specify the location where project data will be stored
- Leave **Create project subdirectory** option enabled and
- Click **Next**

Step 5. In the **Project Type** dialog box choose **RTL Project** option, select **Do not specify sources at this time** and click **Next**

Step 6. In the **Default Part** dialog box select **Parts** option and set the following parameters as it is shown on the Illustration 12.2



Default Part

Choose a default Xilinx part or board for your project. This can be changed later.

Product category: All
Family: Zynq-7000
Package: clg400
Speed grade: -1
Temp grade: All Remaining
Search:

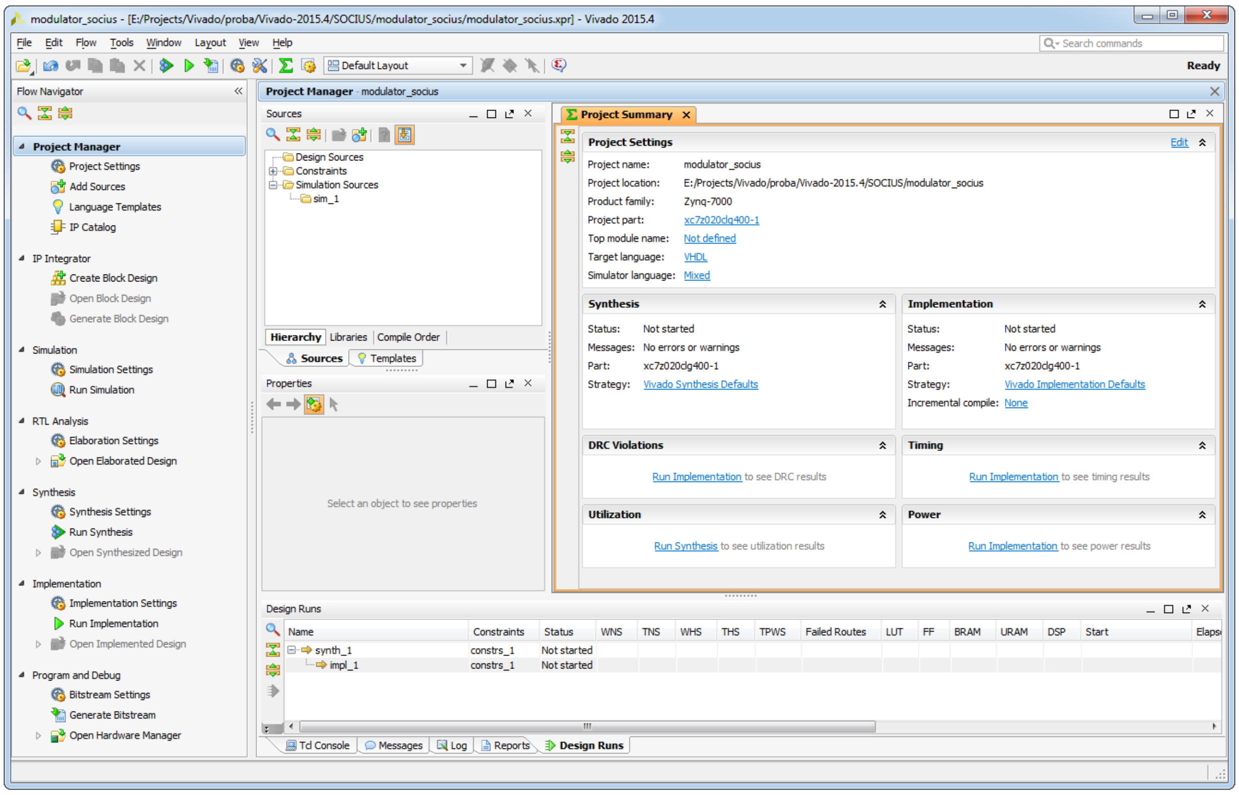
Part	I/O Pin Count	Block RAMs	DSPs	FlipFlops	GTPE2 Transceivers	GTXE2 Transceivers	Gb Transceivers	Available IOBs
xc7z007sclg400-1	400	50	66	28800	0	0	0	100
xc7z010clg400-1	400	60	80	35200	0	0	0	100
xc7z014sclg400-1	400	107	170	81200	0	0	0	125
xc7z020clg400-1	400	140	220	106400	0	0	0	125

Figure 12.2: Default Part dialog box

Step 7. Click **Next**

Step 8. In the **New Project Summary** dialog box click **Finish** if you are satisfied with the summary of your project. If you are not satisfied, you can go back as much as necessary to correct all the questionable issues.

After we finished with the new project creation, in a few seconds Vivado IDE Viewing Environment will appear, see Illustration 12.3.



Project Settings

Project name: modulator_socius
Project location: E:/Projects/Vivado/proba/Vivado-2015.4/SOCIUS/modulator_socius
Product family: Zynq-7000
Project part: xc7z020clg400-1
Top module name: Not defined
Target language: VHDL
Simulator language: Mixed

Synthesis

Status: Not started
Messages: No errors or warnings
Part: xc7z020clg400-1
Strategy: Vivado Synthesis Defaults

Implementation

Status: Not started
Messages: No errors or warnings
Part: xc7z020clg400-1
Strategy: Vivado Implementation Defaults
Incremental compile: None

DRC Violations

Run Implementation to see DRC results

Timing

Run Implementation to see timing results

Utilization

Run Synthesis to see utilization results

Power

Run Implementation to see power results

Design Runs

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Failed Routes	LUT	FF	BRAM	URAM	DSP	Start	Elaps
synth_1	constrs_1	Not started													
impl_1	constrs_1	Not started													

Figure 12.3: Vivado IDE Viewing Environment with created modulator_socius project

12.3 Creating Module

As we already said, in our design we will program PL part of the Zynq FPGA with ***modulator_socius_rtl.vhd*** model. Since existing LEDs and switches on the socius board are connected to the PS part of the Zynq FPGA, we have to instantiate Integrated Logic Analyzer (ILA) and Virtual Input/Output (VIO) cores into our design. All the information about ILA and VIO cores you can find in the Chapter 11 "Debugging Design" of this tutorial.

Both, ILA and VIO cores will be instantiated into our design, where VIO core will be instantiated using the "HDL Instantiation Debug Probing Flow" and ILA core using the "Netlist Insertion Debug Probing Flow", because netlist insertion debug probing flow can be used to insert ILA cores only. All these information you can also find in the Chapter 11 "Debugging Design" of this tutorial where both flows are explained in detail. ILA core will be used to monitor PWM signal width change and VIO core will be used to replace on-board switch used for changing output signal frequency.

Step 1. Instantiate VIO core into our design using steps for VIO core instantiation, explained in the Sub-chapter 11.1 "Inserting ILA and VIO Cores into Design" of this tutorial. Use the same core customizations as it is explained in this sub-chapter:

- In the **VIO (Virtual Input/Output) (3.0)** window, enter *vio_core_name* (**vio_core**) in the **Component Name** field
- In the **General Options** tab, leave **Input Probe Count** to be 1 and **Output Probe Count** also to be 1, because we will need one input probe for pwm_out signal and one output probe for sw0 signal
- In the **PROBE_IN Ports(0..0)** tab leave Probe Width of the **PROBE_IN0** Probe Port to be 1, because our pwm_out signal is 1 bit signal
- In the **PROBE_OUT Ports(0..0)** tab, leave Probe Width of the **PROBE_OUT0** Probe Port to be 1, because our sw0 signal is also 1 bit signal
- Click **OK**

After VIO core generation, your VIO core should appear in the Sources window, see Illustration 12.4

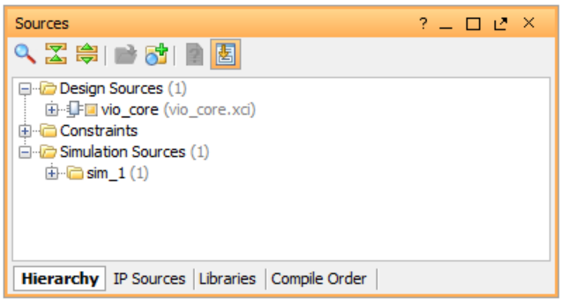


Figure 12.4: Source tab with generated VIO core

ILA core will be instantiated into our design using "Netlist Insertion Debug Probing Flow", explained in the Sub-chapter 11.1 of this tutorial. We will use **mark_debug** attribute to add debug nets (**pwm_s** and **count_s**) to our HDL file (***modulator_-socius_rtl.vhd***). As we already said ILA core will be used to monitor PWM signal width change, where *pwm_s* signal will represent PWM signal and *count_s* will measure the duration of the high pulse of the PWM signal.

In our design despite ILA and VIO cores, we will also have to instantiate Modulator module and counter which will measure the duration of the PWM pulse, see Figure 12.1. Both of these instances, plus ILA and VIO core instances will be included within ***modulator_socius_rtl.vhd*** VHDL model.

Step 2. To include all the necessary Modulator module source files (*frequency_trigger_rt.vhd, counter_rtl.vhd, modulator_-pkg.vhd, sine_rtl.vhd, sine_top_rtl.vhd, pwm_rtl.vhd* and *modulator_rtl.vhd*) into our design, in the Flow Navigator, use **Add Sources** command to add the files and after adding your Sources window should look like as it is shown on the Illustration 12.5.

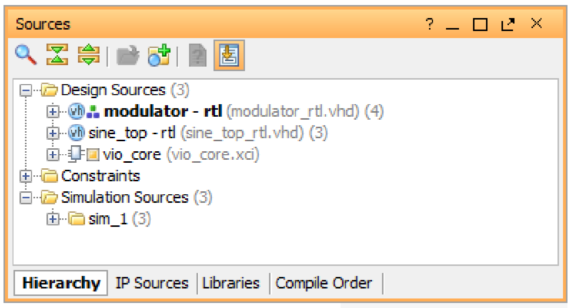


Figure 12.5: Source tab with generated VIO core and Modulator module

Step 3. To create and add ***modulator_socius_rtl.vhd*** and ***modulator_socius_clk_rtl.vhd*** source files use steps for creating modules, explained in **Sub-chapter 2.4.1 Creating a Module Using Vivado Text Editor** of this tutorial. Content of the source files you can find in the text below.

modulator_socius_rtl.vhd VHDL model:

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

library work;
    use work.modulator_pkg.all;

entity modulator_socius is
    generic(
        -- User defined settings for the pwm design
        board_setting_g  : board_setting_t_rec  := socius_c;
        design_setting_g : design_setting_t_rec := design_setting_c
        );

    port(
        clk_in : in std_logic
        );
end entity;

architecture structural of modulator_socius is
```

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

    -- MODULATOR SECTION STARTS! --
    attribute mark_debug : string;
    attribute keep : string;

    signal pwm_s   : std_logic_vector (0 downto 0);
    signal sw0_s   : std_logic_vector (0 downto 0);
    signal count_s : std_logic_vector (31 downto 0) := (others => '0');

    attribute mark_debug of pwm_s   : signal is "true";
    attribute mark_debug of count_s : signal is "true";

    constant c1_c                  : real := board_setting_g.fclk/(real((2**design_setting_g.depth)*(2**
      design_setting_g.width)));
    constant div_factor_freqhigh_c : integer := integer(c1_c/design_setting_g.f_high)*(2**design_setting_g.
      width);
    constant div_factor_freqlow_c  : integer := integer(c1_c/design_setting_g.f_low)*(2**design_setting_g.
      width);

    -- vio_core component definition
    component vio_core
        port (
            clk        : in std_logic;
            probe_in0  : in std_logic_vector (0 downto 0);
            probe_out0 : out std_logic_vector (0 downto 0)
        );
    end component;


begin

    -- modulator module instance
    modulator_i: entity work.modulator(rtl)
        generic map(
            design_setting_g => design_setting_g
            )
        port map(
            clk_in              => clk_in,
            sw0                 => sw0_s(0),
            div_factor_freqhigh => conv_std_logic_vector(div_factor_freqhigh_c, 32),
            div_factor_freqlow  => conv_std_logic_vector(div_factor_freqlow_c, 32),
            pwm_out             => pwm_s(0)
            );

    -- vio_core component instance
    vio_i: vio_core
        port map (
            clk        => clk_in,
            probe_in0  => pwm_s,
            probe_out0 => sw0_s
            );

    -- Counter for measuring the duration of the high pulse of the PWM signal
    measurement_counter_p: process
    begin
        wait until  rising_edge(clk_in);
        if (pwm_s(0) = '0') then
            count_s <= (others => '0');
        else
            count_s <= count_s + 1;
        end if;
    end process;

end architecture;
```

PS part of the Zynq FPGA is also required to generate clock signal for the Modulator_socius design. Properly configured PS part is described in the ***socius_xz_lab_ps_bd*** component of the ***modulator_socius_clk_rtl.vhd*** VHDL model. The complete ***modulator_socius_clk_rtl.vhd*** VHDL model you can find in the text below:

modulator_socius_clk_rtl.vhd VHDL model:

```
library ieee;
    use ieee.std_logic_1164.all;

entity socius_clk_top is
    port(
        -- expansion top slot
        pl_io_t_io_p_io        : inout std_logic_vector (18 downto 0);
        pl_io_t_io_n_io        : inout std_logic_vector (18 downto 0);
        -- expansion main slot
        pl_io_m_io_p_io        : inout std_logic_vector (18 downto 0);
        pl_io_m_io_n_io        : inout std_logic_vector (18 downto 0);
        -- expansion bottom slot
        pl_io_b_io_p_io        : inout std_logic_vector (18 downto 0);
        pl_io_b_io_n_io        : inout std_logic_vector (18 downto 0);
        -- ps io
        ps_ddr3_addr           : inout std_logic_vector(14 downto 0);
        ps_ddr3_ba             : inout std_logic_vector(2 downto 0);
        ps_ddr3_cas_n          : inout std_logic;
        ps_ddr3_ck_n           : inout std_logic;
        ps_ddr3_ck_p           : inout std_logic;
        ps_ddr3_cke            : inout std_logic;
        ps_ddr3_cs_n           : inout std_logic;
        ps_ddr3_dm             : inout std_logic_vector( 3 downto 0);
        ps_ddr3_dq             : inout std_logic_vector(31 downto 0);
        ps_ddr3_dqs_n          : inout std_logic_vector( 3 downto 0);
        ps_ddr3_dqs_p          : inout std_logic_vector( 3 downto 0);
        ps_ddr3_odt            : inout std_logic;
        ps_ddr3_ras_n          : inout std_logic;
        ps_ddr3_reset_n        : inout std_logic;
        ps_ddr3_we_n           : inout std_logic;
        ps_ddr_vrn             : inout std_logic;
        ps_ddr_vrp             : inout std_logic;
        ps_clk_i               : inout std_logic;
        ps_por_n_i             : inout std_logic;
        ps_srst_n_i            : inout std_logic;
        ps_phy_mdc_io          : inout std_logic;
        ps_phy_mdio_io         : inout std_logic;
        ps_phy_rx_clk_io       : inout std_logic;
        ps_phy_rx_ctrl_io      : inout std_logic;
        ps_phy_rxd_io          : inout std_logic_vector(3 downto 0);
        ps_phy_tx_clk_io       : inout std_logic;
        ps_phy_tx_ctrl_io      : inout std_logic;
        ps_phy_txd_io          : inout std_logic_vector(3 downto 0);
        ps_i2c_scl_io          : inout std_logic;
        ps_i2c_sda_io          : inout std_logic;
        ps_led_error_n_io      : inout std_logic;
        ps_led_front_n_io      : inout std_logic_vector(1 downto 0);
        ps_led_sdcard_n_io     : inout std_logic;
        ps_sw0_a_io            : inout std_logic;
        ps_sw0_b_io            : inout std_logic;
        ps_sw1_a_io            : inout std_logic;
        ps_sw1_b_io            : inout std_logic;
        ps_sw2_a_io            : inout std_logic;
        ps_sw2_b_io            : inout std_logic;
        ps_sw3_a_io            : inout std_logic;
        ps_sw3_b_io            : inout std_logic;
        ps_uart_rx_io          : inout std_logic;
        ps_uart_tx_io          : inout std_logic;
        ps_qspi_cs_n_io        : inout std_logic;
        ps_qspi_data_io        : inout std_logic_vector(3 downto 0);
        ps_qspi_clk_io         : inout std_logic;
        ps_sdio_clk_io         : inout std_logic;
        ps_sdio_cmd_io         : inout std_logic;
        ps_sdio_data_io        : inout std_logic_vector(3 downto 0);
        ps_usb_clk_io          : inout std_logic;
        ps_usb_data_io         : inout std_logic_vector(7 downto 0);
        ps_usb_dir_io          : inout std_logic;
        ps_usb_nxt_io          : inout std_logic;
        ps_usb_stp_io          : inout std_logic
      );
end entity;

architecture structural of socius_clk_top is

component socius_xz_lab_ps_bd is
  port (
    pl_clk0                 : out STD_LOGIC;
    pl_clk1                 : out STD_LOGIC;
    pl_clk2                 : out STD_LOGIC;
    pl_clk3                 : out STD_LOGIC;
    pl_int_bot              : in STD_LOGIC_VECTOR ( 0 to 0 );
    pl_int_mid              : in STD_LOGIC_VECTOR ( 0 to 0 );
    pl_int_soc              : in STD_LOGIC_VECTOR ( 0 to 0 );
    pl_int_top              : in STD_LOGIC_VECTOR ( 0 to 0 );
    pl_reset_n              : out STD_LOGIC;
    ddr3_cas_n              : inout STD_LOGIC;
    ddr3_cke                : inout STD_LOGIC;
    ddr3_ck_n               : inout STD_LOGIC;
    ddr3_ck_p               : inout STD_LOGIC;
    ddr3_cs_n               : inout STD_LOGIC;
    ddr3_reset_n            : inout STD_LOGIC;
    ddr3_odt                : inout STD_LOGIC;
    ddr3_ras_n              : inout STD_LOGIC;
    ddr3_we_n               : inout STD_LOGIC;
    ddr3_ba                 : inout STD_LOGIC_VECTOR ( 2 downto 0 );
    ddr3_addr               : inout STD_LOGIC_VECTOR ( 14 downto 0 );
    ddr3_dm                 : inout STD_LOGIC_VECTOR ( 3 downto 0 );
    ddr3_dq                 : inout STD_LOGIC_VECTOR ( 31 downto 0 );
    ddr3_dqs_n              : inout STD_LOGIC_VECTOR ( 3 downto 0 );
    ddr3_dqs_p              : inout STD_LOGIC_VECTOR ( 3 downto 0 );
    fixed_io_mio            : inout STD_LOGIC_VECTOR ( 53 downto 0 );
    fixed_io_ddr_vrn        : inout STD_LOGIC;
    fixed_io_ddr_vrp        : inout STD_LOGIC;
    fixed_io_ps_srstb       : inout STD_LOGIC;
    fixed_io_ps_clk         : inout STD_LOGIC;
    fixed_io_ps_porb        : inout STD_LOGIC;
    sdio_0_cdn              : in STD_LOGIC;
    usbind_0_port_indctl    : out STD_LOGIC_VECTOR ( 1 downto 0 );
    usbind_0_vbus_pwrselect : out STD_LOGIC;
    usbind_0_vbus_pwrfault  : in STD_LOGIC;
    pl_iic_1_sda_i          : in STD_LOGIC;
    pl_iic_1_sda_o          : out STD_LOGIC;
    pl_iic_1_sda_t          : out STD_LOGIC;
    pl_iic_1_scl_i          : in STD_LOGIC;
    pl_iic_1_scl_o          : out STD_LOGIC;
    pl_iic_1_scl_t          : out STD_LOGIC;
    pl_spi_0_sck_i          : in STD_LOGIC;
    pl_spi_0_sck_o          : out STD_LOGIC;
    pl_spi_0_sck_t          : out STD_LOGIC;
    pl_spi_0_io0_i          : in STD_LOGIC;
    pl_spi_0_io0_o          : out STD_LOGIC;
    pl_spi_0_io0_t          : out STD_LOGIC;
    pl_spi_0_io1_i          : in STD_LOGIC;
    pl_spi_0_io1_o          : out STD_LOGIC;
    pl_spi_0_io1_t          : out STD_LOGIC;
    pl_spi_0_ss_i           : in STD_LOGIC;
    pl_spi_0_ss_o           : out STD_LOGIC;
    pl_spi_0_ss1_o          : out STD_LOGIC;
    pl_spi_0_ss2_o          : out STD_LOGIC;
    pl_spi_0_ss_t           : out STD_LOGIC;
    pl_uart_1_txd           : out STD_LOGIC;
    pl_uart_1_rxd           : in STD_LOGIC;
    pl_bram_bot_addr        : out STD_LOGIC_VECTOR ( 15 downto 0 );
    pl_bram_bot_clk         : out STD_LOGIC;
    pl_bram_bot_din         : out STD_LOGIC_VECTOR ( 31 downto 0 );
    pl_bram_bot_dout        : in STD_LOGIC_VECTOR ( 31 downto 0 );
    pl_bram_bot_en          : out STD_LOGIC;
    pl_bram_bot_rst         : out STD_LOGIC;
    pl_bram_bot_we          : out STD_LOGIC_VECTOR ( 3 downto 0 );
    pl_bram_mid_addr        : out STD_LOGIC_VECTOR ( 15 downto 0 );
    pl_bram_mid_clk         : out STD_LOGIC;
    pl_bram_mid_din         : out STD_LOGIC_VECTOR ( 31 downto 0 );
    pl_bram_mid_dout        : in STD_LOGIC_VECTOR ( 31 downto 0 );
    pl_bram_mid_en          : out STD_LOGIC;
    pl_bram_mid_rst         : out STD_LOGIC;
    pl_bram_mid_we          : out STD_LOGIC_VECTOR ( 3 downto 0 );
    pl_bram_soc_addr        : out STD_LOGIC_VECTOR ( 15 downto 0 );
    pl_bram_soc_clk         : out STD_LOGIC;
    pl_bram_soc_din         : out STD_LOGIC_VECTOR ( 31 downto 0 );
    pl_bram_soc_dout        : in STD_LOGIC_VECTOR ( 31 downto 0 );
    pl_bram_soc_en          : out STD_LOGIC;
    pl_bram_soc_rst         : out STD_LOGIC;
    pl_bram_soc_we          : out STD_LOGIC_VECTOR ( 3 downto 0 );
    pl_bram_top_addr        : out STD_LOGIC_VECTOR ( 15 downto 0 );
    pl_bram_top_clk         : out STD_LOGIC;
    pl_bram_top_din         : out STD_LOGIC_VECTOR ( 31 downto 0 );
    pl_bram_top_dout        : in STD_LOGIC_VECTOR ( 31 downto 0 );
    pl_bram_top_en          : out STD_LOGIC;
    pl_bram_top_rst         : out STD_LOGIC;
    pl_bram_top_we          : out STD_LOGIC_VECTOR ( 3 downto 0 )
  );
  end component socius_xz_lab_ps_bd;

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

    --bram register interface soc
    signal pl_bram_soc_addr_s     : std_logic_vector (15 downto 0);
    signal pl_bram_soc_din_s      : std_logic_vector (31 downto 0);
    signal pl_bram_soc_dout_s     : std_logic_vector (31 downto 0);
    signal pl_bram_soc_en_s       : std_logic;
    signal pl_bram_soc_rst_s      : std_logic;
    signal pl_bram_soc_we_s       : std_logic_vector ( 3 downto 0);
    --bram register interface mid
    signal pl_bram_mid_addr_s     : std_logic_vector (15 downto 0);
    signal pl_bram_mid_din_s      : std_logic_vector (31 downto 0);
    signal pl_bram_mid_dout_s     : std_logic_vector (31 downto 0);
    signal pl_bram_mid_en_s       : std_logic;
    signal pl_bram_mid_rst_s      : std_logic;
    signal pl_bram_mid_we_s       : std_logic_vector ( 3 downto 0);
    --bram register interface top
    signal pl_bram_top_addr_s     : std_logic_vector (15 downto 0);
    signal pl_bram_top_din_s      : std_logic_vector (31 downto 0);
    signal pl_bram_top_dout_s     : std_logic_vector (31 downto 0);
    signal pl_bram_top_en_s       : std_logic;
    signal pl_bram_top_rst_s      : std_logic;
    signal pl_bram_top_we_s       : std_logic_vector ( 3 downto 0);
    --bram register interface bot
    signal pl_bram_bot_addr_s     : std_logic_vector (15 downto 0);
    signal pl_bram_bot_din_s      : std_logic_vector (31 downto 0);
    signal pl_bram_bot_dout_s     : std_logic_vector (31 downto 0);
    signal pl_bram_bot_en_s       : std_logic;
    signal pl_bram_bot_rst_s      : std_logic;
    signal pl_bram_bot_we_s       : std_logic_vector ( 3 downto 0);
    -- declaration for fixed signal PL to PS
    signal pl_clk0_s           : std_logic;
    signal pl_clk1_s           : std_logic;
    signal pl_clk2_s           : std_logic;
    signal pl_clk3_s           : std_logic;
    signal pl_reset_n_s        : std_logic;

    -- ps signals
    signal ps_mio_s            : std_logic_vector(53 downto 0);

    --uart, i2c, spi signals
    signal uart_rxd_s   : std_logic;
    signal uart_txd_s   : std_logic;
    signal spi_io0_i_s  : std_logic;
    signal spi_io0_o_s  : std_logic;
    signal spi_io0_t_s  : std_logic;
    signal spi_io1_i_s  : std_logic;
    signal spi_io1_o_s  : std_logic;
    signal spi_io1_t_s  : std_logic;
    signal spi_sck_i_s  : std_logic;
    signal spi_sck_o_s  : std_logic;
    signal spi_sck_t_s  : std_logic;
    signal spi_ss1_o_s  : std_logic;
    signal spi_ss2_o_s  : std_logic;
    signal spi_ss_i_s   : std_logic;
    signal spi_ss_o_s   : std_logic;
    signal spi_ss_t_s   : std_logic;
    signal iic_scl_i_s  : std_logic;
    signal iic_scl_o_s  : std_logic;
    signal iic_scl_t_s  : std_logic;
    signal iic_sda_i_s  : std_logic;
    signal iic_sda_o_s  : std_logic;
    signal iic_sda_t_s  : std_logic;

    --interrupt signals to ps
    signal pl_int_soc_s : std_logic;
    signal pl_int_top_s : std_logic;
    signal pl_int_mid_s : std_logic;
    signal pl_int_bot_s : std_logic;

begin



    -- modulator module instance
    modulator_i: entity work.modulator_socius(structural)
        port map(
            clk_in                  => pl_clk0_s
            );

    -- instance of processor system PS
    socius_xz_lab_ps_bd_i: component socius_xz_lab_ps_bd
        port map (
            ddr3_addr               => ps_ddr3_addr,
            ddr3_ba                 => ps_ddr3_ba,
            ddr3_cas_n              => ps_ddr3_cas_n,
            ddr3_ck_n               => ps_ddr3_ck_n,
            ddr3_ck_p               => ps_ddr3_ck_p,
            ddr3_cke                => ps_ddr3_cke,
            ddr3_cs_n               => ps_ddr3_cs_n,
            ddr3_dm                 => ps_ddr3_dm,
            ddr3_dq                 => ps_ddr3_dq,
            ddr3_dqs_n              => ps_ddr3_dqs_n,
            ddr3_dqs_p              => ps_ddr3_dqs_p,
            ddr3_odt                => ps_ddr3_odt,
            ddr3_ras_n              => ps_ddr3_ras_n,
            ddr3_reset_n            => ps_ddr3_reset_n,
            ddr3_we_n               => ps_ddr3_we_n,
            fixed_io_ddr_vrn        => ps_ddr_vrn,
            fixed_io_ddr_vrp        => ps_ddr_vrp,
            fixed_io_mio            => ps_mio_s,
            fixed_io_ps_clk         => ps_clk_i,
            fixed_io_ps_porb        => ps_por_n_i,
            fixed_io_ps_srstb       => ps_srst_n_i,
            pl_uart_1_rxd           => uart_rxd_s,
            pl_uart_1_txd           => uart_txd_s,
            pl_spi_0_io0_i          => spi_io0_i_s,
            pl_spi_0_io0_o          => spi_io0_o_s,
            pl_spi_0_io0_t          => spi_io0_t_s,
            pl_spi_0_io1_i          => spi_io1_i_s,
            pl_spi_0_io1_o          => spi_io1_o_s,
            pl_spi_0_io1_t          => spi_io1_t_s,
            pl_spi_0_sck_i          => spi_sck_i_s,
            pl_spi_0_sck_o          => spi_sck_o_s,
            pl_spi_0_sck_t          => spi_sck_t_s,
            pl_spi_0_ss1_o          => spi_ss1_o_s,
            pl_spi_0_ss2_o          => spi_ss2_o_s,
            pl_spi_0_ss_i           => spi_ss_i_s,
            pl_spi_0_ss_o           => spi_ss_o_s,
            pl_spi_0_ss_t           => spi_ss_t_s,
            pl_iic_1_scl_i          => iic_scl_i_s,
            pl_iic_1_scl_o          => iic_scl_o_s,
            pl_iic_1_scl_t          => iic_scl_t_s,
            pl_iic_1_sda_i          => iic_sda_i_s,
            pl_iic_1_sda_o          => iic_sda_o_s,
            pl_iic_1_sda_t          => iic_sda_t_s,
            sdio_0_cdn              => '1', -- pl_sd_cd_n_i,
            usbind_0_port_indctl    => open,
            usbind_0_vbus_pwrfault  => '1', -- pl_usb_fault_n_i,
            usbind_0_vbus_pwrselect => open,
            pl_bram_bot_addr        => pl_bram_bot_addr_s,
            pl_bram_bot_clk         => open,
            pl_bram_bot_din         => pl_bram_bot_din_s,
            pl_bram_bot_dout        => pl_bram_bot_dout_s,
            pl_bram_bot_en          => pl_bram_bot_en_s,
            pl_bram_bot_rst         => pl_bram_bot_rst_s,
            pl_bram_bot_we          => pl_bram_bot_we_s,
            pl_bram_mid_addr        => pl_bram_mid_addr_s,
            pl_bram_mid_clk         => open,
            pl_bram_mid_din         => pl_bram_mid_din_s,
            pl_bram_mid_dout        => pl_bram_mid_dout_s,
            pl_bram_mid_en          => pl_bram_mid_en_s,
            pl_bram_mid_rst         => pl_bram_mid_rst_s,
            pl_bram_mid_we          => pl_bram_mid_we_s,
            pl_bram_soc_addr        => pl_bram_soc_addr_s,
            pl_bram_soc_clk         => open,
            pl_bram_soc_din         => pl_bram_soc_din_s,
            pl_bram_soc_dout        => pl_bram_soc_dout_s,
            pl_bram_soc_en          => pl_bram_soc_en_s,
            pl_bram_soc_rst         => pl_bram_soc_rst_s,
            pl_bram_soc_we          => pl_bram_soc_we_s,
            pl_bram_top_addr        => pl_bram_top_addr_s,
            pl_bram_top_clk         => open,
            pl_bram_top_din         => pl_bram_top_din_s,
            pl_bram_top_dout        => pl_bram_top_dout_s,
            pl_bram_top_en          => pl_bram_top_en_s,
            pl_bram_top_rst         => pl_bram_top_rst_s,
            pl_bram_top_we          => pl_bram_top_we_s,
            pl_clk0                 => pl_clk0_s,
            pl_clk1                 => pl_clk1_s,
            pl_clk2                 => pl_clk2_s,
            pl_clk3                 => pl_clk3_s,
            pl_reset_n              => pl_reset_n_s,
            pl_int_soc(0)           => pl_int_soc_s,
            pl_int_top(0)           => pl_int_top_s,
            pl_int_mid(0)           => pl_int_mid_s,
            pl_int_bot(0)           => pl_int_bot_s
            );

            -- assignment of MIO to board names

            ps_mio_s (53)           <= ps_phy_mdio_io;
            ps_mio_s (52)           <= ps_phy_mdc_io;
            ps_mio_s (51)           <= ps_uart_tx_io;
            ps_mio_s (50)           <= ps_uart_rx_io;
            ps_mio_s (49)           <= ps_led_error_n_io;
            ps_mio_s (48 downto 47) <= ps_led_front_n_io(1 downto 0);
            ps_mio_s (46)           <= ps_led_sdcard_n_io;
            ps_mio_s (45 downto 42) <= ps_sdio_data_io;
            ps_mio_s (41)           <= ps_sdio_cmd_io;
            ps_mio_s (40)           <= ps_sdio_clk_io;
            ps_mio_s (39)           <= ps_usb_data_io(7);
            ps_mio_s (38)           <= ps_usb_data_io(6);
            ps_mio_s (37)           <= ps_usb_data_io(5);
            ps_mio_s (36)           <= ps_usb_clk_io;
            ps_mio_s (35)           <= ps_usb_data_io(3);
            ps_mio_s (34)           <= ps_usb_data_io(2);
            ps_mio_s (33)           <= ps_usb_data_io(1);
            ps_mio_s (32)           <= ps_usb_data_io(0);
            ps_mio_s (31)           <= ps_usb_nxt_io;
            ps_mio_s (30)           <= ps_usb_stp_io;
            ps_mio_s (29)           <= ps_usb_dir_io;
            ps_mio_s (28)           <= ps_usb_data_io(4);
            ps_mio_s (27)           <= ps_phy_rx_ctrl_io;
            ps_mio_s (26 downto 23) <= ps_phy_rxd_io;
            ps_mio_s (22)           <= ps_phy_rx_clk_io;
            ps_mio_s (21)           <= ps_phy_tx_ctrl_io;
            ps_mio_s (20 downto 17) <= ps_phy_txd_io;
            ps_mio_s (16)           <= ps_phy_tx_clk_io;
            ps_mio_s (15)           <= ps_i2c_sda_io;
            ps_mio_s (14)           <= ps_i2c_scl_io;
            ps_mio_s (13)           <= ps_sw3_b_io;
            ps_mio_s (12)           <= ps_sw3_a_io;
            ps_mio_s (11)           <= ps_sw2_b_io;
            ps_mio_s (10)           <= ps_sw2_a_io;
            ps_mio_s (9)            <= ps_sw1_b_io;
            ps_mio_s (8)            <= ps_sw1_a_io;
            ps_mio_s (7)            <= ps_sw0_b_io;
            ps_mio_s (6)            <= ps_qspi_clk_io;
            ps_mio_s (5 downto 2)   <= ps_qspi_data_io;
            ps_mio_s (1)            <= ps_qspi_cs_n_io;
            ps_mio_s (0)            <= ps_sw0_a_io;
```

```
end architecture;
```

Note: Don't forget to set ***modulator_socius_clk_rtl.vhd*** source file to be the top file!

Step 4. Now is the time to create and add constraints file for the socius board, ***modulator_socius.xdc***. To create and add constraints file, please use steps from the Sub-chapter 10.1 "Creating XDC File", where it is in detail explained in paragraph "Creating a XDC File using Vivado Text Editor". The complete ***modulator_socius.xdc*** constraints file you can find in the text below.

modulator_socius.xdc constraints file:

```
# set properties for bitstream genration
set_property BITSTREAM.GENERAL.COMPRESS TRUE [current_design]
#set_property BITSTREAM.GENERAL.XADCENHANCEDLINEARITY ON [current_design]
#set_property BITSTREAM.GENERAL.XADCPOWERDOWN ENABLE [current_design]

# set configuration bank voltages
set_property CFGBVS VCCO [current_design]
set_property CONFIG_VOLTAGE 3.3 [current_design]

# set condition for power analyzer
set_operating_conditions -ambient_temp 50
set_operating_conditions -board small
set_operating_conditions -airflow 250
set_operating_conditions -heatsink low
set_operating_conditions -board_layers 12to15

# unrelate clock domains in PL for clocks genrated in PS f
#set_false_path -from [get_clocks clk_fpga_1] -to [get_clocks clk_fpga_0]
#set_false_path -from [get_clocks clk_fpga_0] -to [get_clocks clk_fpga_1]
#set_clock_groups -asynchronous -group clk_fpga_0 -group clk_fpga_1

# only for power designs
#set_property C_CLK_INPUT_FREQ_HZ 300000000 [get_debug_cores dbg_hub]
#set_property C_ENABLE_CLK_DIVIDER false [get_debug_cores dbg_hub]
#set_property C_USER_SCAN_CHAIN 1 [get_debug_cores dbg_hub]
#connect_debug_port dbg_hub/clk [get_nets pl_clk3]

# Push flip flops to IOBs
#set_property IOB true [get_cells -hier *io_i_s_reg*]
#set_property IOB true [get_cells -hier *io_o_reg*]
#set_property IOB true [get_cells -hier *io_t_reg*]

# PL pins with fixed functionality for xz1 and  xz2

set_property PACKAGE_PIN M14 [get_ports pl_b35_m14_io]
set_property IOSTANDARD LVCMOS33 [get_ports pl_b35_m14_io]
set_output_delay -clock [get_clocks clk_fpga_0] -max 1.000 [get_ports pl_b35_m14_io]
set_output_delay -clock [get_clocks clk_fpga_0] -min 0.500 [get_ports pl_b35_m14_io]

set_property PACKAGE_PIN M15 [get_ports pl_b35_m15_io]
set_property IOSTANDARD LVCMOS33 [get_ports pl_b35_m15_io]
set_output_delay -clock [get_clocks clk_fpga_0] -max 1.000 [get_ports pl_b35_m15_io]
set_output_delay -clock [get_clocks clk_fpga_0] -min 0.500 [get_ports pl_b35_m15_io]

set_property PACKAGE_PIN T19 [get_ports pl_hsw_good_i]
set_property IOSTANDARD LVCMOS33 [get_ports pl_hsw_good_i]
set_input_delay -clock [get_clocks clk_fpga_0] -max 5.000 [get_ports pl_hsw_good_i]
set_input_delay -clock [get_clocks clk_fpga_0] -min 4.500 [get_ports pl_hsw_good_i]

set_property PACKAGE_PIN V13 [get_ports pl_phy_reset_n_o]
set_property IOSTANDARD LVCMOS33 [get_ports pl_phy_reset_n_o]

set_property PACKAGE_PIN T15 [get_ports pl_sd_cd_n_i]
set_property IOSTANDARD LVCMOS33 [get_ports pl_sd_cd_n_i]

set_property PACKAGE_PIN J15 [get_ports pl_pwm_fan_o]
set_property IOSTANDARD LVCMOS33 [get_ports pl_pwm_fan_o]

set_property PACKAGE_PIN R19 [get_ports pl_pwr_en_i]
set_property IOSTANDARD LVCMOS33 [get_ports pl_pwr_en_i]
set_input_delay -clock [get_clocks clk_fpga_0] -max 5.000 [get_ports pl_pwr_en_i]
set_input_delay -clock [get_clocks clk_fpga_0] -min 4.500 [get_ports pl_pwr_en_i]

set_property PACKAGE_PIN G14 [get_ports pl_rtc_out_i]
set_property IOSTANDARD LVCMOS33 [get_ports pl_rtc_out_i]
set_input_delay -clock [get_clocks clk_fpga_0] -max 5.000 [get_ports pl_rtc_out_i]
set_input_delay -clock [get_clocks clk_fpga_0] -min 4.500 [get_ports pl_rtc_out_i]

set_property PACKAGE_PIN U13 [get_ports pl_usb_reset_n_o]
set_property IOSTANDARD LVCMOS33 [get_ports pl_usb_reset_n_o]
set_output_delay -clock [get_clocks clk_fpga_0] -min 1.000 [get_ports pl_usb_reset_n_o]
set_output_delay -clock [get_clocks clk_fpga_0] -max 0.500 [get_ports pl_usb_reset_n_o]

set_property PACKAGE_PIN T14 [get_ports pl_usb_fault_n_i]
set_property IOSTANDARD LVCMOS33 [get_ports pl_usb_fault_n_i]
```

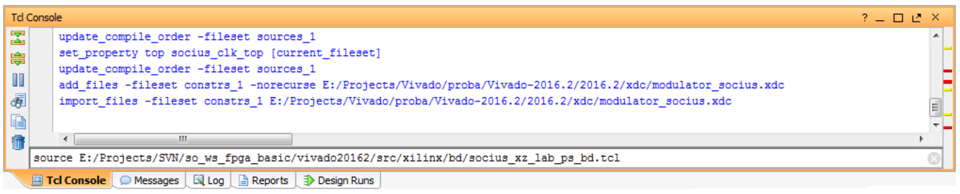
```
set_input_delay -clock [get_clocks clk_fpga_0] -max 5.000 [get_ports pl_usb_fault_n_i]
set_input_delay -clock [get_clocks clk_fpga_0] -min 4.500 [get_ports pl_usb_fault_n_i]

#set_property PACKAGE_PIN M14 [get_ports pl_b35_m14_io]
#set_property IOSTANDARD LVCMOS33 [get_ports pl_b35_m14_io]
#set_output_delay -clock [get_clocks clk_fpga_0] -max 1.000 [get_ports pl_b35_m14_io]
#set_output_delay -clock [get_clocks clk_fpga_0] -min 0.500 [get_ports pl_b35_m14_io]

#set_property PACKAGE_PIN M15 [get_ports pl_b35_m15_io]
#set_property IOSTANDARD LVCMOS33 [get_ports pl_b35_m15_io]
#set_output_delay -clock [get_clocks clk_fpga_0] -max 1.000 [get_ports pl_b35_m15_io]
#set_output_delay -clock [get_clocks clk_fpga_0] -min 0.500 [get_ports pl_b35_m15_io]

#set_property PACKAGE_PIN T19 [get_ports pl_hsw_good_i]
#set_property IOSTANDARD LVCMOS33 [get_ports pl_hsw_good_i]
#set_input_delay -clock [get_clocks clk_fpga_0] -max 5.000 [get_ports pl_hsw_good_i]
#set_input_delay -clock [get_clocks clk_fpga_0] -min 4.500 [get_ports pl_hsw_good_i]

#set_property PACKAGE_PIN V13 [get_ports pl_phy_reset_n_o]
#set_property IOSTANDARD LVCMOS33 [get_ports pl_phy_reset_n_o]
#set_output_delay -clock [get_clocks clk_fpga_1] -max 1.000 [get_ports pl_phy_reset_n_o]
#set_output_delay -clock [get_clocks clk_fpga_1] -min 0.500 [get_ports pl_phy_reset_n_o]

#set_property PACKAGE_PIN T15 [get_ports pl_sd_cd_n_i]
#set_property IOSTANDARD LVCMOS33 [get_ports pl_sd_cd_n_i]

#set_property PACKAGE_PIN J15 [get_ports pl_pwm_fan_o]
#set_property IOSTANDARD LVCMOS33 [get_ports pl_pwm_fan_o]
#set_output_delay -clock [get_clocks clk_fpga_1] -max 1.000 [get_ports pl_pwm_fan_o]
#set_output_delay -clock [get_clocks clk_fpga_1] -min 0.500 [get_ports pl_pwm_fan_o]

#set_property PACKAGE_PIN R19 [get_ports pl_pwr_en_i]
#set_property IOSTANDARD LVCMOS33 [get_ports pl_pwr_en_i]
#set_input_delay -clock [get_clocks clk_fpga_0] -max 5.000 [get_ports pl_pwr_en_i]
#set_input_delay -clock [get_clocks clk_fpga_0] -min 4.500 [get_ports pl_pwr_en_i]

#set_property PACKAGE_PIN G14 [get_ports pl_rtc_out_i]
#set_property IOSTANDARD LVCMOS33 [get_ports pl_rtc_out_i]
#set_input_delay -clock [get_clocks clk_fpga_0] -max 5.000 [get_ports pl_rtc_out_i]
#set_input_delay -clock [get_clocks clk_fpga_0] -min 4.500 [get_ports pl_rtc_out_i]

#set_property PACKAGE_PIN U13 [get_ports pl_usb_reset_n_o]
#set_property IOSTANDARD LVCMOS33 [get_ports pl_usb_reset_n_o]
#set_output_delay -clock [get_clocks clk_fpga_0] -min 1.000 [get_ports pl_usb_reset_n_o]
#set_output_delay -clock [get_clocks clk_fpga_0] -max 0.500 [get_ports pl_usb_reset_n_o]

#set_property PACKAGE_PIN T14 [get_ports pl_usb_fault_n_i]
#set_property IOSTANDARD LVCMOS33 [get_ports pl_usb_fault_n_i]
#set_input_delay -clock [get_clocks clk_fpga_0] -max 5.000 [get_ports pl_usb_fault_n_i]
#set_input_delay -clock [get_clocks clk_fpga_0] -min 4.500 [get_ports pl_usb_fault_n_i]
```

Finally, we must configure the Zynq PS part to work on socius development board. This includes a number of configuration steps. All the PS configuration steps can be done using the Vivado GUI, by creating a block design. However, since this task includes a lot of manual settings of the Zynq PS, a better approach would be to do this manual configuration only once and then to create a Tcl script file that can be used in all future configurations of the Zynq PS part. The Tcl script that should be used to correctly configure Zynq PS to work on socius board is ***socius_xz_lab_ps_bd.tcl***. This Tcl script file is too long to be shown in the tutorial, so ask your instructor for details.

Step 5. Next step is to execute the ***socius_xz_lab_ps_bd.tcl*** Tcl file in the Vivado IDE. Go to the Tcl console window and type the following and press enter:

```
source <path>/socius_xz_lab_ps_bd.tcl
```

Where <path> stands for the full path to the folder where the ***socius_xz_lab_ps_bd.tcl*** Tcl file is stored.



```
update_compile_order -fileset sources_1
set_property top socius_clk_top [current_fileset]
update_compile_order -fileset sources_1
add_files -fileset constrs_1 -norecurse E:/Projects/Vivado/proba/Vivado-2016.2/2016.2/xdc/modulator_socius.xdc
import_files -fileset constrs_1 E:/Projects/Vivado/proba/Vivado-2016.2/2016.2/xdc/modulator_socius.xdc
```

```
source E:/Projects/SVN/so_ws_fpga_basic/vivado20162/src/xilinx/bd/socius_xz_lab_ps_bd.tcl
```

Figure 12.6: Tcl Console window

After Vivado has finished with the Tcl script execution, a created block diagram containing Zynq PS will be visible in the Vivado IDE, as shown on the Illustration 12.7.

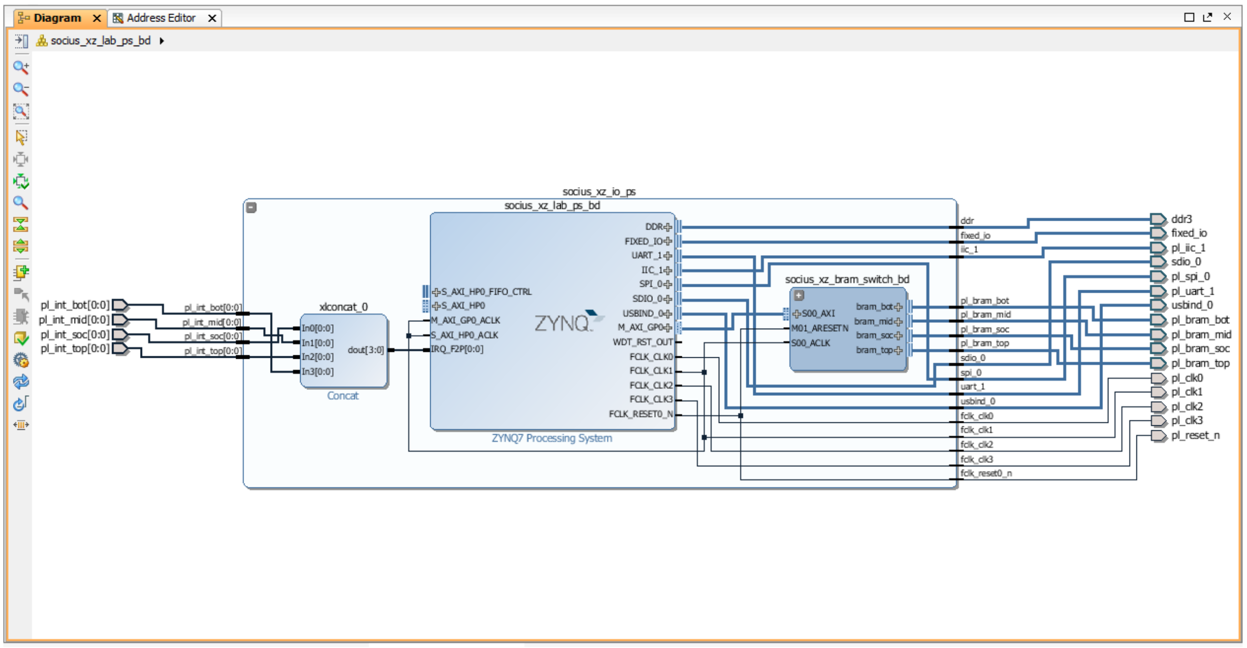


Figure 12.7: Block diagram of Zynq PS configured to run on socius board

Step 6. In the Vivado **Flow Navigator**, click **Run Synthesis** command and wait for task to be completed

Step 7. After the synthesis is completed, choose **Open Synthesized Design** option in the **Synthesis Completed** dialog box

Step 8. Open **Debug Layout** (if it is not already opened) and in the **Debug** window, select **Set Up Debug** button to launch the **Set Up Debug** wizard. In the **Set Up Debug** wizard add **pwm_s** and **count_s** nets to ILA core, as it is explained in steps **23** - **32** in the Sub-chapter 11.1 "Inserting ILA and VIO Cores into Design".

Note: Pay attention to enable **Capture control** feature for ILA in step 31!

Step 9. Implement your design with **Run Implementation** option from the **Flow Navigator** / **Implementation** (see **Sub--Chapter 10.2.2 Run Implementation**)

Step 10. Generate bitstream file with **Generate Bitstream** option from the **Flow Navigator** / **Program and Debug** (see **Sub-Chapter 10.3 Generate Bitstream File**)

Step 11. Program your **socius** board (see **Sub-Chapter 10.4 Program Device**)

Step 12. When the socius board is programmed, select **File** -> **Export** -> **Export Hardware...** option from the main Vivado IDE menu

Step 13. In the **Export Hardware** dialog box, you don't have to include bistream file, so just click **OK**

In order to get the internal FPGA clock running, we must run some application on the processing system. In order to do this, following steps must be performed:

Step 14. Select **File** -> **Launch SDK** from the main Vivado IDE menu

Step 15. In the **Launch SDK** dialog box, make sure that both **Exported location** and **Workspace** are set to **Local to Project** and click **OK**

SDK will be launched in a separate window.

To create an application project, do the following:

Step 16. Select **File** -> **New** -> **Application Project** and the **Application Project** dialog box will appear

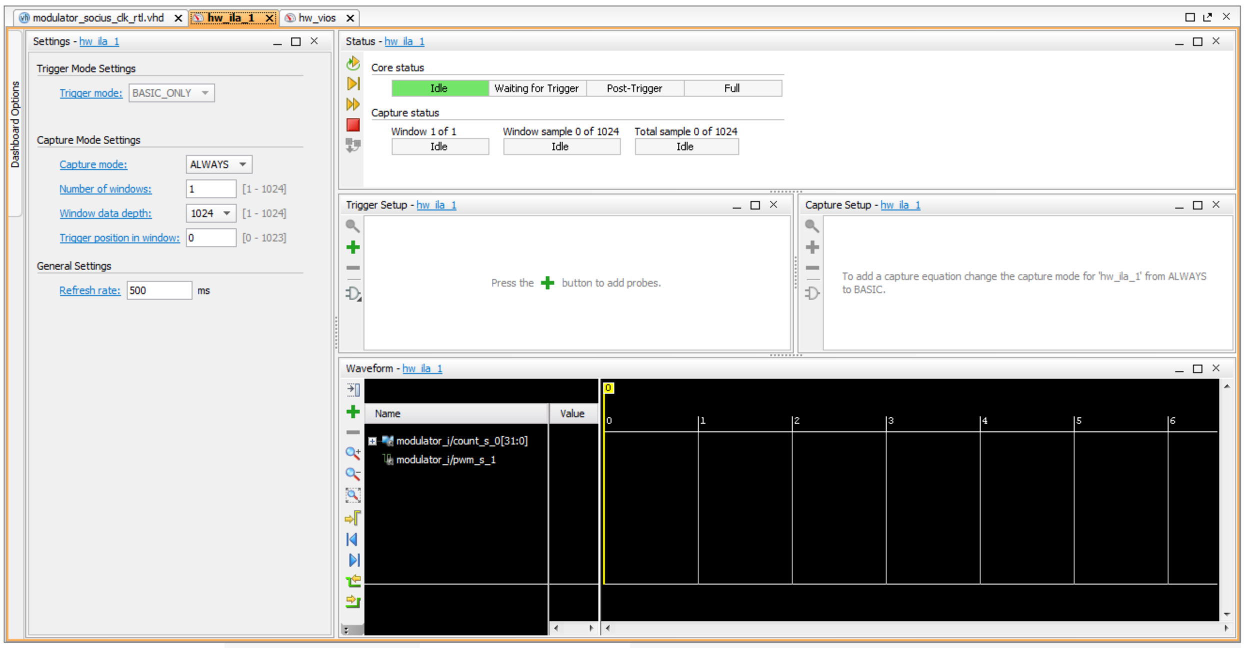
Step 17. In the **Project name** field, type a name of the new project, in our case it will be ***modulator_socius*** and click **Next**

Step 18. In the **Templates** dialog box, choose one of the available templates to generate a fully-functioning application project. You can choose **Hello World** template and click **Finish**.

Step 19. In the **Project Explorer** select your application project (***modulator_socius***), right-click on it and select **Run As** -> **Launch on Hardware (System Debugger)** option

Step 20. Turn back to the Vivado IDE and in the **Hardware** window of the **Hardware Manager** right-click on the FPGA device (xc7z020) and select **Refresh Device** option

After refreshing the FPGA device the Hardware window now shows the ILA and VIO cores that were detected after scanning the device and default dashboard for each debug core is automatically opened. The default ILA dashboard can be seen on the Illustration 12.8.



Settings - hw_ila_1

Trigger Mode Settings

Trigger mode: BASIC_ONLY

Capture Mode Settings

Capture mode: ALWAYS

Number of windows: 1 [1 - 1024]

Window data depth: 1024 [1 - 1024]

Trigger position in window: 0 [0 - 1023]

General Settings

Refresh rate: 500 ms

Core status: Idle | Waiting for Trigger | Post-Trigger | Full

Capture status: Window 1 of 1 Idle | Window sample 0 of 1024 Idle | Total sample 0 of 1024 Idle

Press the button to add probes.

To add a capture equation change the capture mode for 'hw_ila_1' from ALWAYS to BASIC.

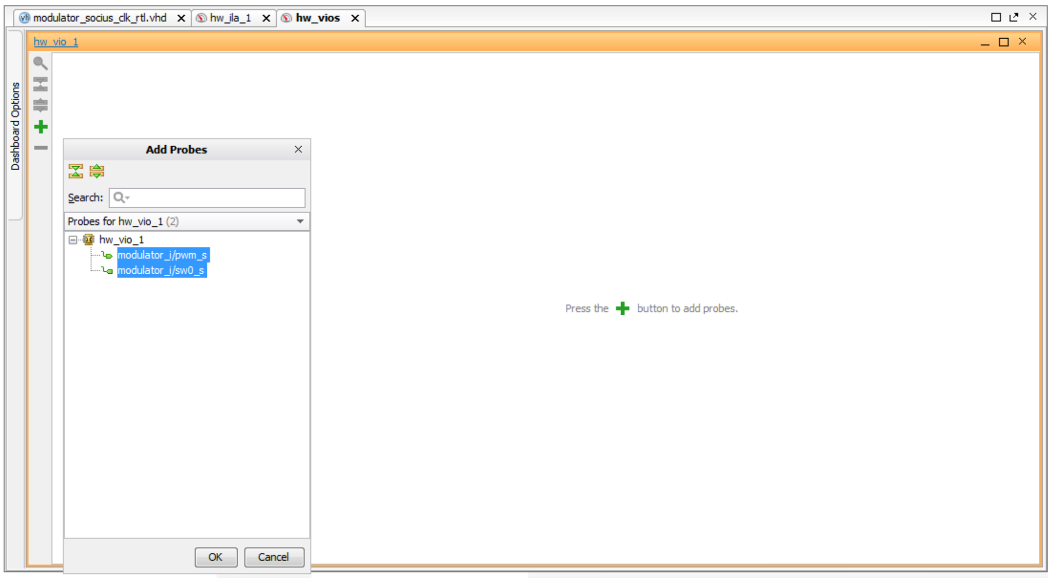
modulator_i/count_s_0[31:0]

modulator_i/pwm_s_1

Figure 12.8: ILA Dashboard

Step 21. Open the VIO dashboard by clicking the ***hw_vios*** tab and press green + button in the middle of the VIO dashboard to add the probes

Step 22. In the **Add Probes** window select both ***pwm_s*** and ***sw0_s*** probes and click **OK**, see Illustration 12.9



Probes for hw_vio_1 (2)

hw_vio_1

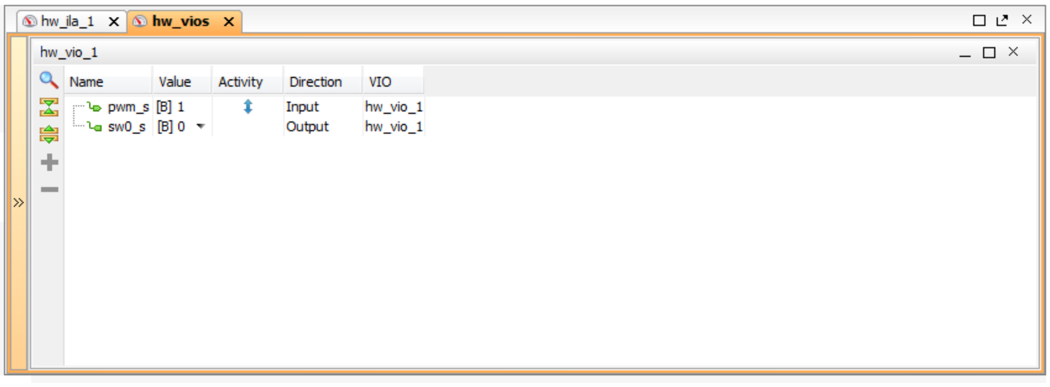
modulator_i/pwm_s

modulator_i/sw0_s

Press the button to add probes.

Figure 12.9: Add Probes to the VIO window

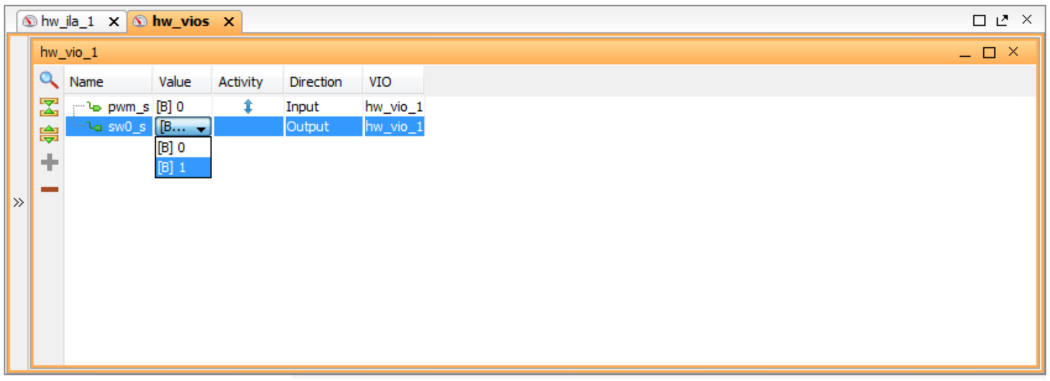
Step 23. In the **VIO Probes** window you will see two 1-bit probes, **pwm_s** and ***sw0_s***, see Illustration 12.10. ***pwm_s*** probe is actually connected to the ***pwm_out*** output port of the Modulator module, as can be seen on the Figure 11.9 and from the modulator_vio_rtl.vhd source code. Similarly, ***sw0_s*** probe is connected to the ***sw0*** input port of the Modulator module.



Name	Value	Activity	Direction	VIO
pwm_s	[B] 1	↕	Input	hw_vio_1
sw0_s	[B] 0		Output	hw_vio_1

Figure 12.10: VIO Probes window

In the VIO Probes window, you can observe the rate of change of the ***pwm_s*** signal. You can change the frequency of the ***pwm_s*** signal by changing the value of the ***sw0_s*** probe from 0 to 1 and from 1 to 0, see Illustration 12.11. The default ***sw0_s*** value is 0.



Name	Value	Activity	Direction	VIO
pwm_s	[B] 0	↕	Input	hw_vio_1
sw0_s	[B...		Output	hw_vio_1

Figure 12.11: Changing the sw0_s value

Step 24. Turn back to the ILA dashboard by clicking the ***hw_ila_1*** tab and in the **Trigger Setup** window press green + button in the middle to add the probes

Step 25. In the **Add Probes** window select only ***pwm_s_1*** probe and click **OK**, see Illustration 12.12

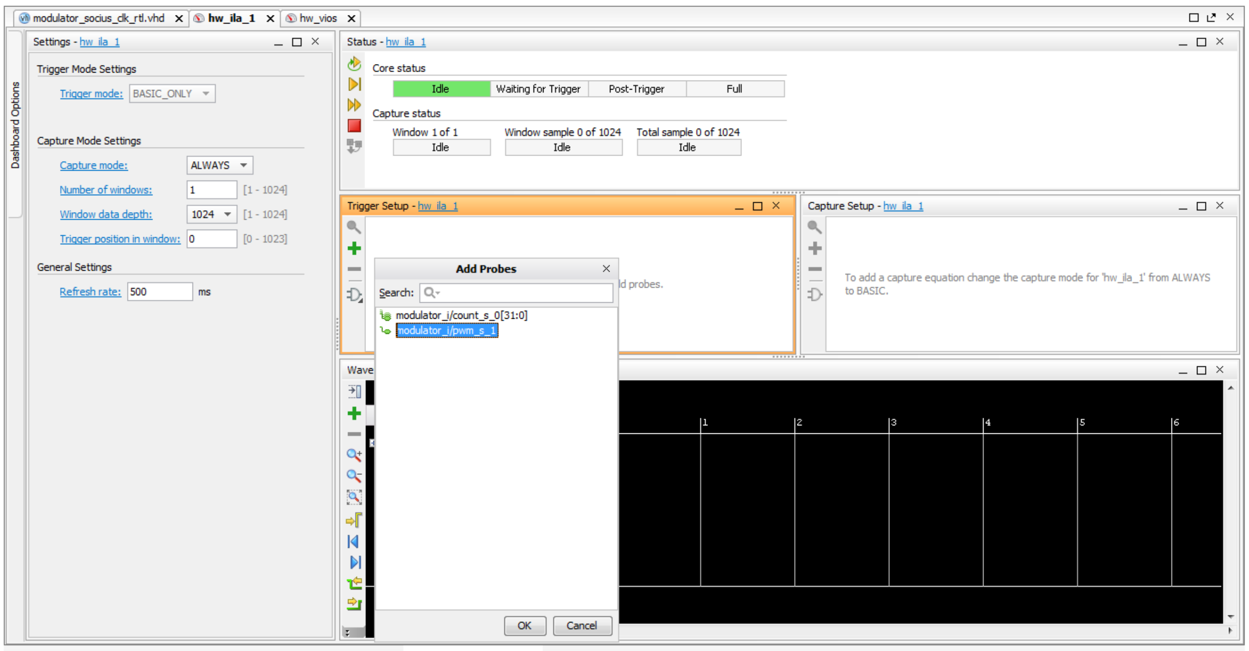
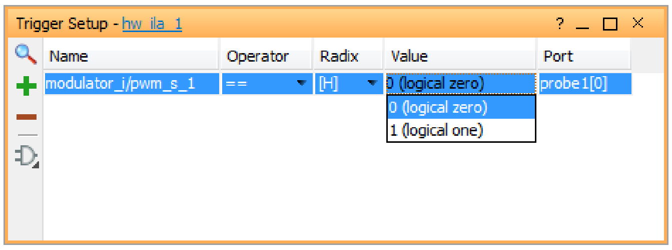


Figure 12.12: Add Probes to the Trigger Setup window

Step 26. Now, when the ILA debug probe ***pwm_s_1*** is in the **Trigger Setup** window, we can create trigger conditions and debug probe compare values. In the **Trigger Setup** window, leave **== (equal)** value in the **Operator** cell, **[H] (Hexadecimal)** value in the **Radix** cell and set the **Value** parameter to be **0 (logical zero)**, as it is shown on the Illustration 12.13.



Name	Operator	Radix	Value	Port
modulator_i/pwm_s_1	==	[H]	0 (logical zero)	probe1[0]

0 (logical zero)
1 (logical one)

Figure 12.13: Changing the Compare Values in the Trigger Setup window

Step 27. In the main ILA Properties window, change the **Capture mode** to be **BASIC** in the **Capture Mode Settings** section

Step 28. In the **Capture Setup** window press green + button in the middle to add the probes

Step 29. In the **Add Probes** window select only ***pwm_s_1*** probe and click **OK**, see Illustration 12.14

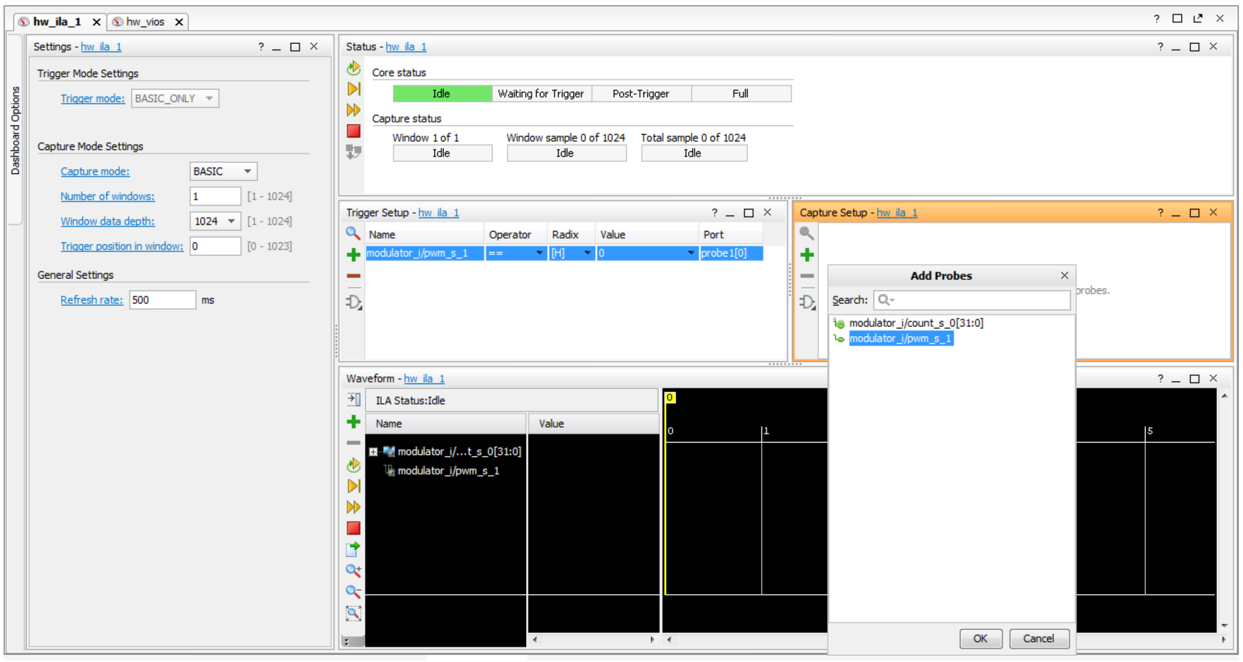
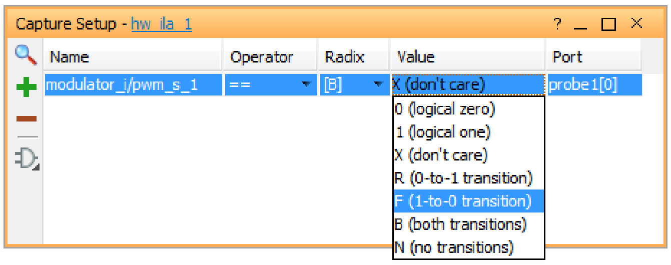


Figure 12.14: Add Probes to the Capture Setup window

Step 30. In the **Capture Setup** window, leave **== (equal)** value in the **Operator** cell, **[B] (Binary)** value in the **Radix** cell and set the **Value** parameter to be **F (1-to-0 transition)**, as it is shown on the Illustration 12.15.



Name	Operator	Radix	Value	Port
modulator_i/pwm_s_1	==	[B]	X (don't care)	probe1[0]

0 (logical zero)
1 (logical one)
X (don't care)
R (0-to-1 transition)
F (1-to-0 transition)
B (both transitions)
N (no transitions)

Figure 12.15: Changing the Compare Values in the Capture Setup window

Step 31. After we set all the ILA core parameters, we can run the ILA core trigger unit by pressing the **Run Trigger** button.

Once the ILA core captured data has been uploaded to the Vivado IDE, it is displayed in the **Waveform Viewer**, see Illustration 12.16.

Note: After triggering the ILA core, in the waveform viewer change the *count_s_0* **Waveform Style** from **Digital** to **Analog**, and your captured waveform should look like as the waveform on the Illustration 12.16.

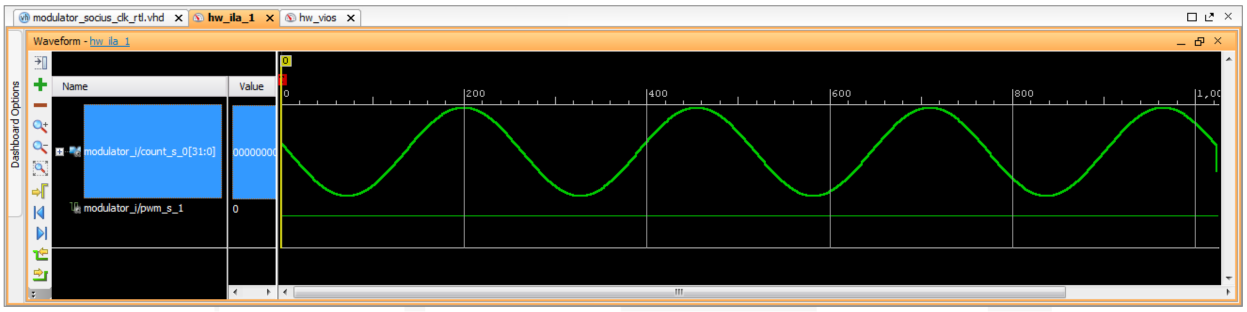


Figure 12.16: Captured waveform of the sine signal, when sw0=0

Step 32. Turn back to the VIO Probes window and change the **Value** of the ***sw0_s*** signal from **0** to **1**, see Illustration 12.11

Step 33. Arm the trigger ones more and after triggering the ILA core your captured waveform should look like as the waveform on the Illustration 12.17

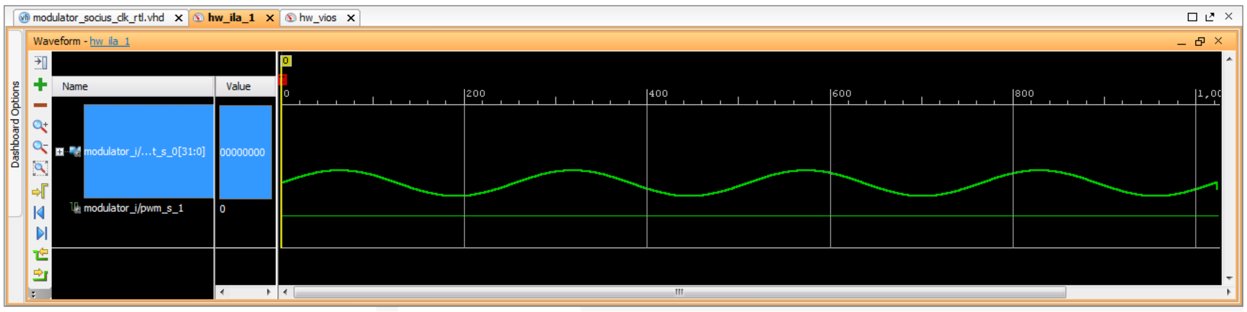


Figure 12.17: Captured waveform of the sine signal, when sw0=1

Note: By comparing the waveforms shown on Illustrations 12.16 and 12.17 we can observe that they differ in the amplitude value. This is expected since the waveforms actually represent the width of the PWM pulse generated by the modulator module. Since the frequencies of two generated PWM signals differ (one has a frequency of 1 Hz and the other of 3.5 Hz) and the PWM pulse width measurement module always uses the same frequency for measuring the duration of the PWM pulse, when the PWM frequency increases the duration of the PWM pulse will decrease, therefore decreasing the amplitude of the output signal of the PWM pulse width measurement module.

Chapter 13

DESIGNING WITH IPs

This chapter will guide you through the process of IP core creation, customization and integration into your design. Vivado Design Suite offers **IP Packager** and **IP Integrator** tool to help you with the process of designing with IP.

The Vivado Design Suite provides multiple ways to use IP in a design. The Vivado IDE provides an IP-Centric design flow that enables you to add IP modules to your project from various design sources. IP-Centric design flow helps you quickly turn design and algorithms into reusable IP. Illustration 13.1 illustrates the IP-Centric design flow.

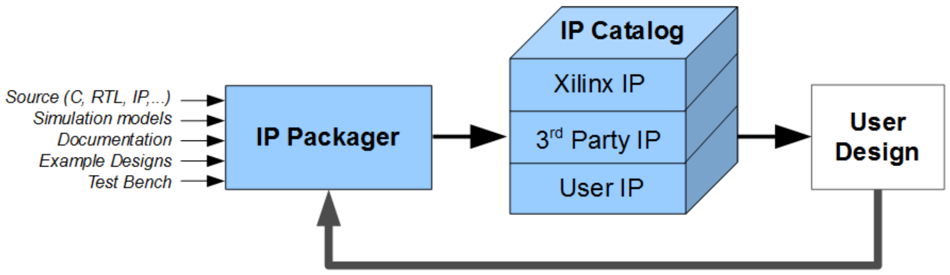


Figure 13.1: Vivado IP-Centric Design Flow

You can customize and add an IP into the project using the IP Catalog from the Vivado IDE. In the IP Catalog you can add the following:

- Modules from System Generator for DSP designs (MATLAB/Simulink algorithms) and Vivado High-Level Synthesis designs (C/C++ algorithms)
- Third party IP
- User designs packaged using IP Packager

The available methods to work with IP in a design are:

- Use the Managed IP Flow to customize IP and generate output products, including a Synthesized Design Checkpoint (DCP)
- Use IP in either Project or Non-Project modes by referencing the created Xilinx Core Instance (XCI) file, which is a recommended method for large projects with many team members
- Create and add IP within a Vivado Project. Access the IP Catalog in a project to create and add IP to design. Store the IP either inside the project or save it externally to the project, which is the recommended method for projects with small team sizes
- Create and customize IP and generate output products in a non- project script flow, including generation of a Synthesized Design Checkpoint (DCP)

In this tutorial we will show you how to create and add user designs in the IP Catalog, packaged using the **IP Packager** tool and how you can instantiate your IP into the project using IP Catalog or **IP Integrator** tools.

13.1 IP Packager

The Vivado IP Packager is a tool designed on the IEEE IP-XACT standard. It provides any Vivado user the ability to package a design at any stage of the design flow and prepare it for use in the Vivado environment. The IP user can then instantiate IP into their design either by using the IP Catalog or IP Integrator. The Illustration 13.2 shows the flow of the IP packaging and IP usage, using the IP Catalog.

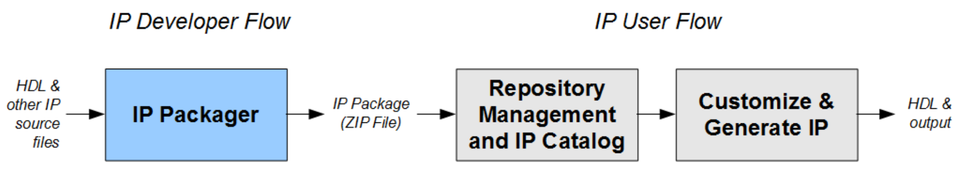


Figure 13.2: IP Packaging and Usage Flow

As you can see from the illustration above, the IP developer uses the IP Packager to package HDL and other IP source files and create an archive (*zip file*). The packaged IP is then given to the user and added to the IP Catalog. When the IP is in the IP Catalog, a user can select the IP and create a customization for their design.

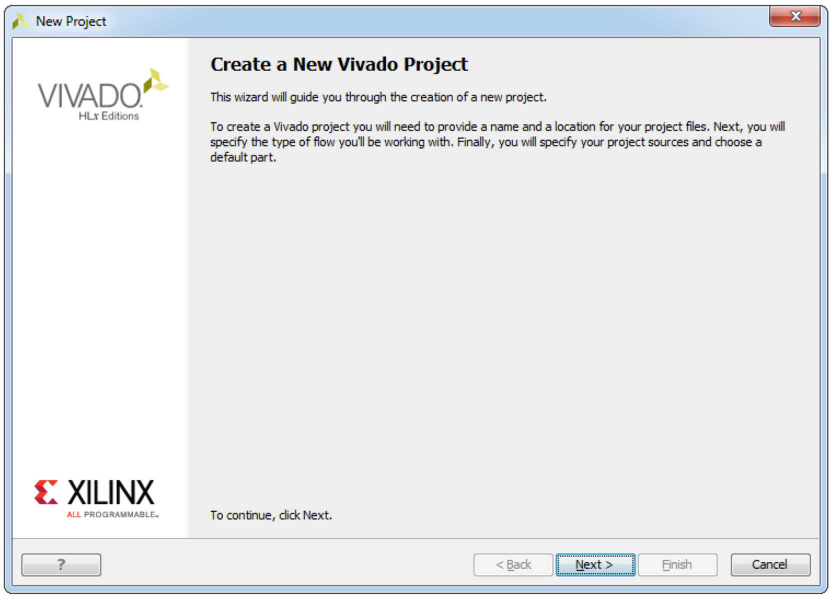
The Vivado IDE contains a **Create and Package IP** wizard that helps and guides you step-by-step through the IP creation and packaging steps. The **Create and Package IP** wizard offers the following functions:

- Create IP using source files and information from a project
- Create IP from a specified directory
- Create a template AXI4 peripheral that includes the HDL, drivers, a test application, a Bus Functional Model (BFM), and an example template

The following steps describe how to use the **Package IP** wizard to package IP. You can use the IP Packager within your existing Vivado project or you can create a new Vivado project for IP you want to package.

Step 1. Close the existing Modulator project with the **File ->** **Close Project** option from the main Vivado IDE menu and in the Vivado **Getting Started** page choose **Create New Project** option

Step 2. In the **Create a New Vivado Project** dialog box, click **Next**, see Illustration 13.3



Create a New Vivado Project

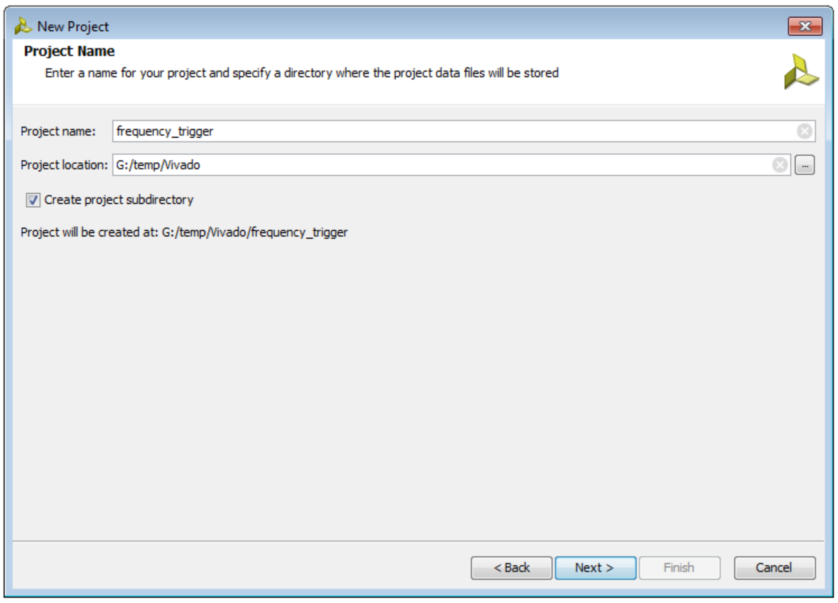
This wizard will guide you through the creation of a new project.

To create a Vivado project you will need to provide a name and a location for your project files. Next, you will specify the type of flow you'll be working with. Finally, you will specify your project sources and choose a default part.

To continue, click Next.

Figure 13.3: Create a New Vivado Project dialog box

Step 3. In the **Project Name** dialog box, enter a name of a new project and specify directory where the project data files will be stored. Name the project **frequency_trigger**, verify the project location and click **Next**, see Illustration 13.4


New Project

Project Name

Enter a name for your project and specify a directory where the project data files will be stored

Project name: frequency_trigger

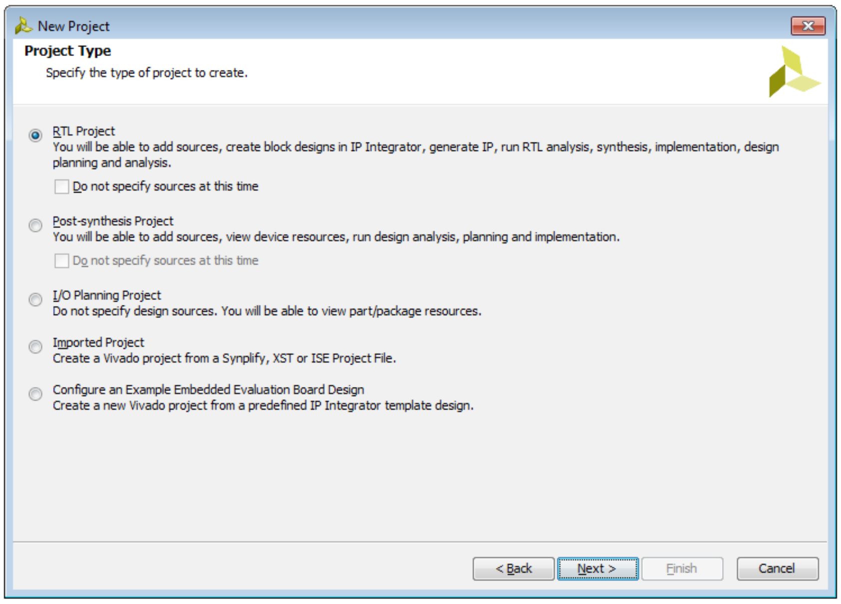
Project location: G:/temp/Vivado

☑ Create project subdirectory

Project will be created at: G:/temp/Vivado/frequency_trigger

Figure 13.4: Project Name dialog box

Step 4. In the **Project Type** dialog box, verify that the **RTL Project** is selected and the **Do not specify sources at this time** option is unchecked and click **Next**, see Illustration 13.5


New Project

Project Type

Specify the type of project to create.

● RTL Project
You will be able to add sources, create block designs in IP Integrator, generate IP, run RTL analysis, synthesis, implementation, design planning and analysis.
☐ Do not specify sources at this time

○ Post-synthesis Project
You will be able to add sources, view device resources, run design analysis, planning and implementation.
☐ Do not specify sources at this time

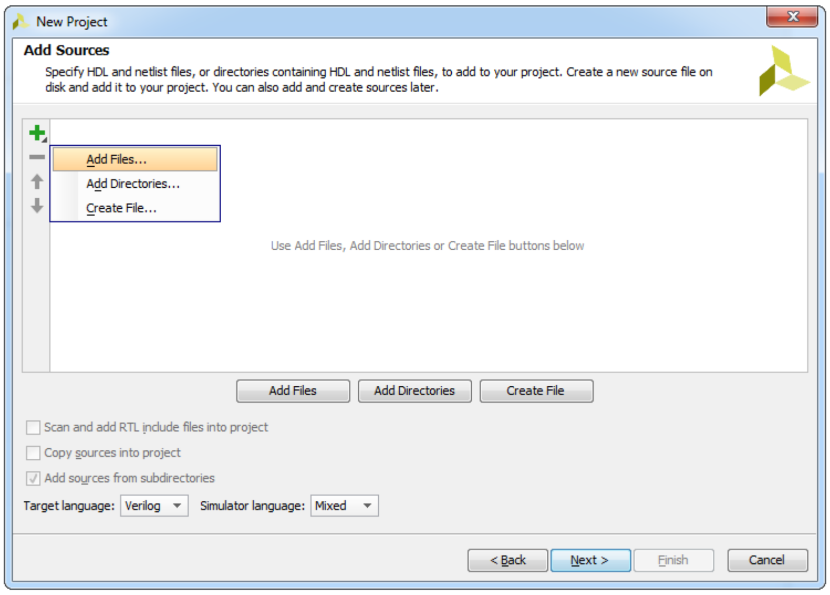
○ I/O Planning Project
Do not specify design sources. You will be able to view part/package resources.

○ Imported Project
Create a Vivado project from a Synplify, XST or ISE Project File.

○ Configure an Example Embedded Evaluation Board Design
Create a new Vivado project from a predefined IP Integrator template design.

Figure 13.5: Project Type dialog box

Step 5. In the **Add Sources** dialog box, click + icon and choose **Add Files...** option to add HDL and Netlist files to your project, see Illustration 13.6



Add Sources

Specify HDL and netlist files, or directories containing HDL and netlist files, to add to your project. Create a new source file on disk and add it to your project. You can also add and create sources later.

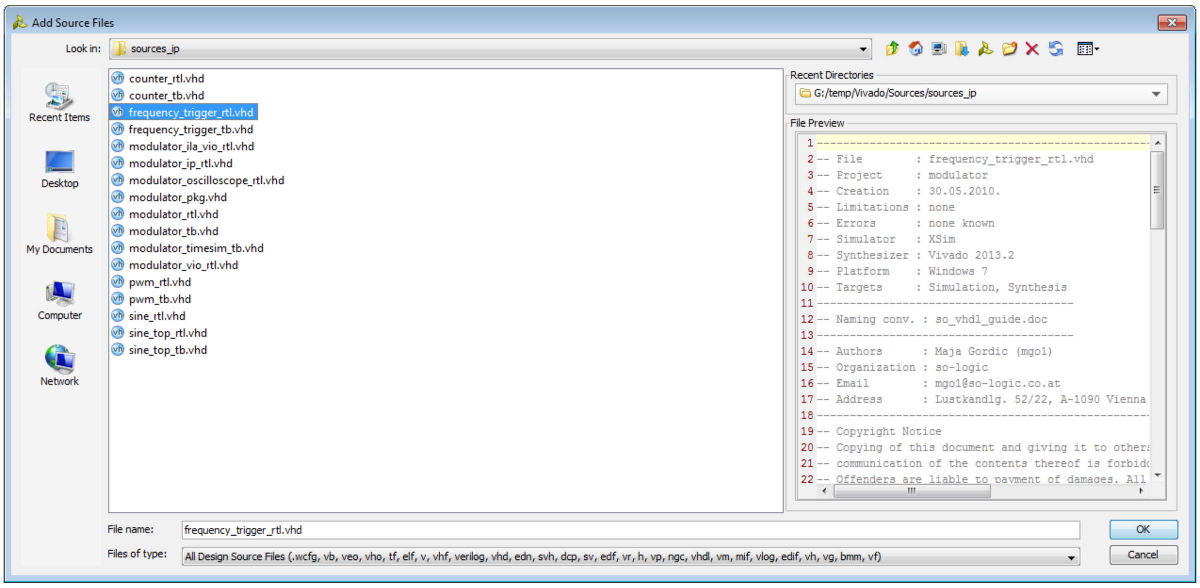
Use Add Files, Add Directories or Create File buttons below

- [ ] Scan and add RTL include files into project
- [ ] Copy sources into project
- [x] Add sources from subdirectories

Target language: Verilog Simulator language: Mixed

Figure 13.6: Add Sources dialog box

Step 6. In the **Add Source Files** dialog box, select ***frequency_trigger_rtl.vhd*** source file and click **OK**, see Illustration 13.7



Look in: sources_ip

counter_rtl.vhd
counter_tb.vhd
frequency_trigger_rtl.vhd
frequency_trigger_tb.vhd
modulator_ila_vio_rtl.vhd
modulator_ip_rtl.vhd
modulator_oscilloscope_rtl.vhd
modulator_pkg.vhd
modulator_rtl.vhd
modulator_tb.vhd
modulator_timesim_tb.vhd
modulator_vio_rtl.vhd
pwm_rtl.vhd
pwm_tb.vhd
sine_rtl.vhd
sine_top_rtl.vhd
sine_top_tb.vhd

Recent Directories: G:/temp/Vivado/Sources/sources_ip

File Preview

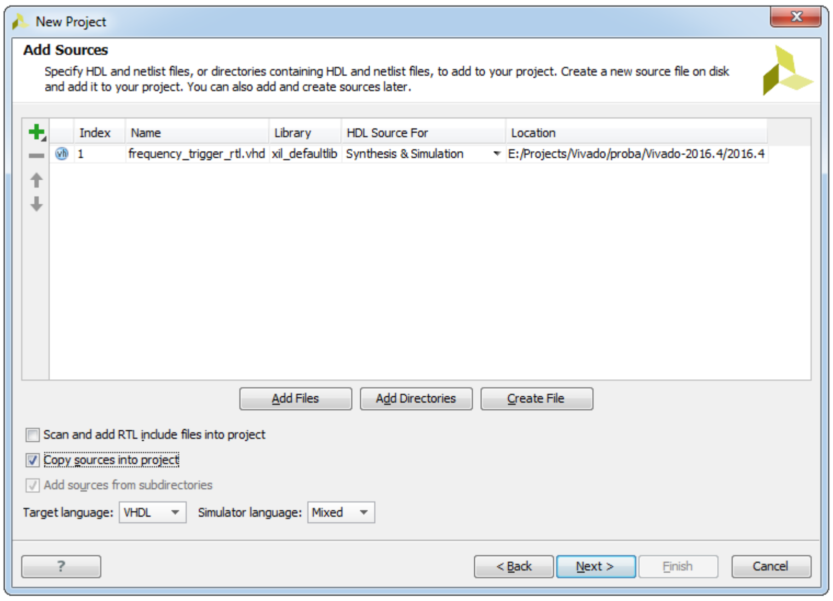
```
--------------------------------------------------------------
-- File        : frequency_trigger_rtl.vhd
-- Project     : modulator
-- Creation    : 30.05.2010.
-- Limitations : none
-- Errors      : none known
-- Simulator   : XSim
-- Synthesizer : Vivado 2013.2
-- Platform    : Windows 7
-- Targets     : Simulation, Synthesis
---------------------------------------
-- Naming conv. : so_vhdl_guide.doc
---------------------------------------
-- Authors      : Maja Gordic (mgo1)
-- Organization : so-logic
-- Email        : mgo1@so-logic.co.at
-- Address      : Lustkandlg. 52/22, A-1090 Vienna
--------------------------------------------------------------
-- Copyright Notice
-- Copying of this document and giving it to others
-- communication of the contents thereof is forbidd
-- Offenders are liable to payment of damages. All
```

File name: frequency_trigger_rtl.vhd

Files of type: All Design Source Files (.wcfg, vb, veo, vho, tf, elf, v, vhf, verilog, vhd, edn, svh, dcp, sv, edf, vr, h, vp, ngc, vhdl, vm, mif, vlog, edif, vh, vg, bmm, vf)

Figure 13.7: Add Source Files dialog box

Step 7. In the **Add Sources** dialog box, select **VHDL** as the target language and ensure that you select **Copy sources into project** option, because Xilinx strongly recommends the source files are present within the project, see Illustration 13.8



Add Sources

Specify HDL and netlist files, or directories containing HDL and netlist files, to add to your project. Create a new source file on disk and add it to your project. You can also add and create sources later.

Index	Name	Library	HDL Source For	Location
1	frequency_trigger_rtl.vhd	xil_defaultlib	Synthesis & Simulation	E:/Projects/Vivado/proba/Vivado-2016.4/2016.4

☐ Scan and add RTL include files into project

☑ Copy sources into project

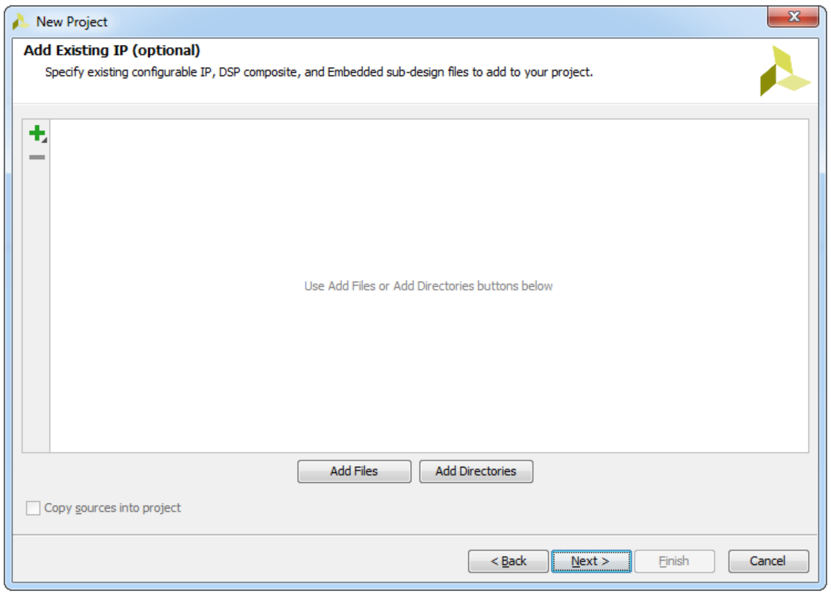
☑ Add sources from subdirectories

Target language: VHDL Simulator language: Mixed

Figure 13.8: Add Sources dilaog box with added source file

Step 8. Click **Next**

Step 9. In the **Add Existing IP (optional)** dialog box, click **Next**



Add Existing IP (optional)

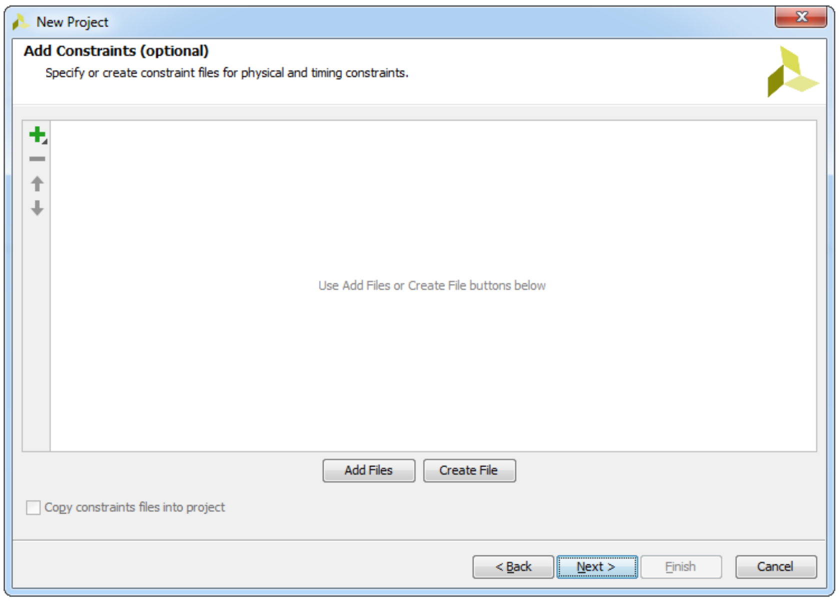
Specify existing configurable IP, DSP composite, and Embedded sub-design files to add to your project.

Use Add Files or Add Directories buttons below

☐ Copy sources into project

Figure 13.9: Add Existing IP (optional) dialog box

Step 10. In the **Add Constraints (optional)** dialog box, remove if there are some constraints files, and click **Next**, see Illustration 13.10



Add Constraints (optional)

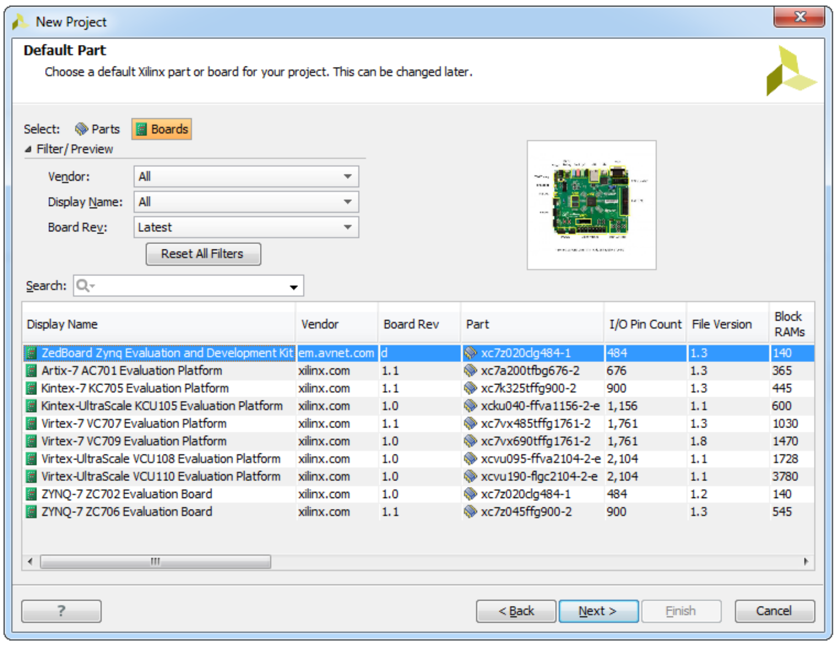
Specify or create constraint files for physical and timing constraints.

Use Add Files or Create File buttons below

Copy constraints files into project

Figure 13.10: Add Constraints (optional) dialog box

Step 11. In the **Default Part** dialog box, ensure that the **ZedBoard Zynq Evaluation and Development Kit** is selected and click **Next**, see Illustration 13.11



Default Part

Choose a default Xilinx part or board for your project. This can be changed later.

Select: Parts Boards

Filter/ Preview

Vendor: All

Display Name: All

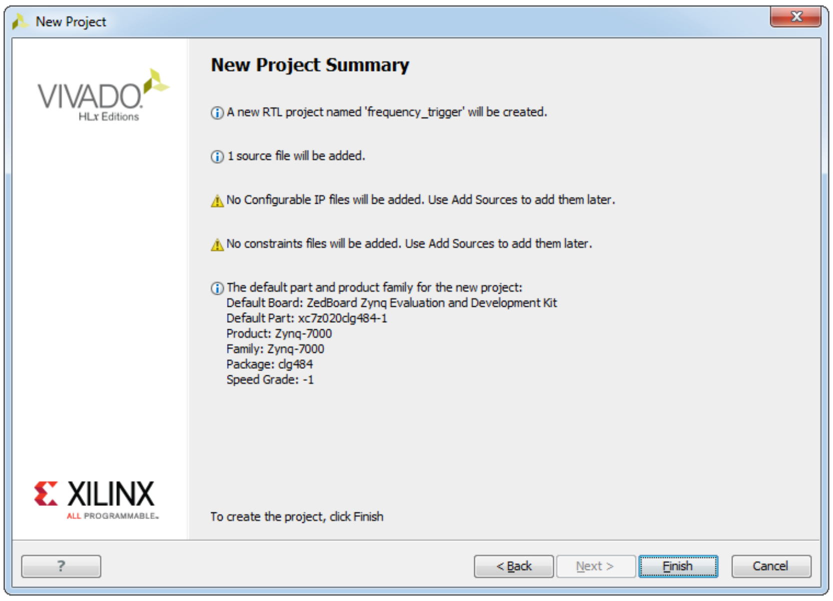
Board Rev: Latest

Search:

Display Name	Vendor	Board Rev	Part	I/O Pin Count	File Version	Block RAMs
ZedBoard Zynq Evaluation and Development Kit	em.avnet.com	d	xc7z020clg484-1	484	1.3	140
Artix-7 AC701 Evaluation Platform	xilinx.com	1.1	xc7a200tfbg676-2	676	1.3	365
Kintex-7 KC705 Evaluation Platform	xilinx.com	1.1	xc7k325tffg900-2	900	1.3	445
Kintex-UltraScale KCU105 Evaluation Platform	xilinx.com	1.0	xcku040-ffva1156-2-e	1,156	1.1	600
Virtex-7 VC707 Evaluation Platform	xilinx.com	1.1	xc7vx485tffg1761-2	1,761	1.3	1030
Virtex-7 VC709 Evaluation Platform	xilinx.com	1.0	xc7vx690tffg1761-2	1,761	1.8	1470
Virtex-UltraScale VCU108 Evaluation Platform	xilinx.com	1.0	xcvu095-ffva2104-2-e	2,104	1.1	1728
Virtex-UltraScale VCU110 Evaluation Platform	xilinx.com	1.0	xcvu190-flgc2104-2-e	2,104	1.1	3780
ZYNQ-7 ZC702 Evaluation Board	xilinx.com	1.0	xc7z020clg484-1	484	1.2	140
ZYNQ-7 ZC706 Evaluation Board	xilinx.com	1.1	xc7z045ffg900-2	900	1.3	545

Figure 13.11: Default Part dialog box

Step 12. In the **New Project Summary** dialog box, click **Finish** if you are satisfied with the summary of your project or go back as much as necessary to correct all the questionable issues, see Illustration 13.12



New Project Summary

A new RTL project named 'frequency_trigger' will be created.

1 source file will be added.

No Configurable IP files will be added. Use Add Sources to add them later.

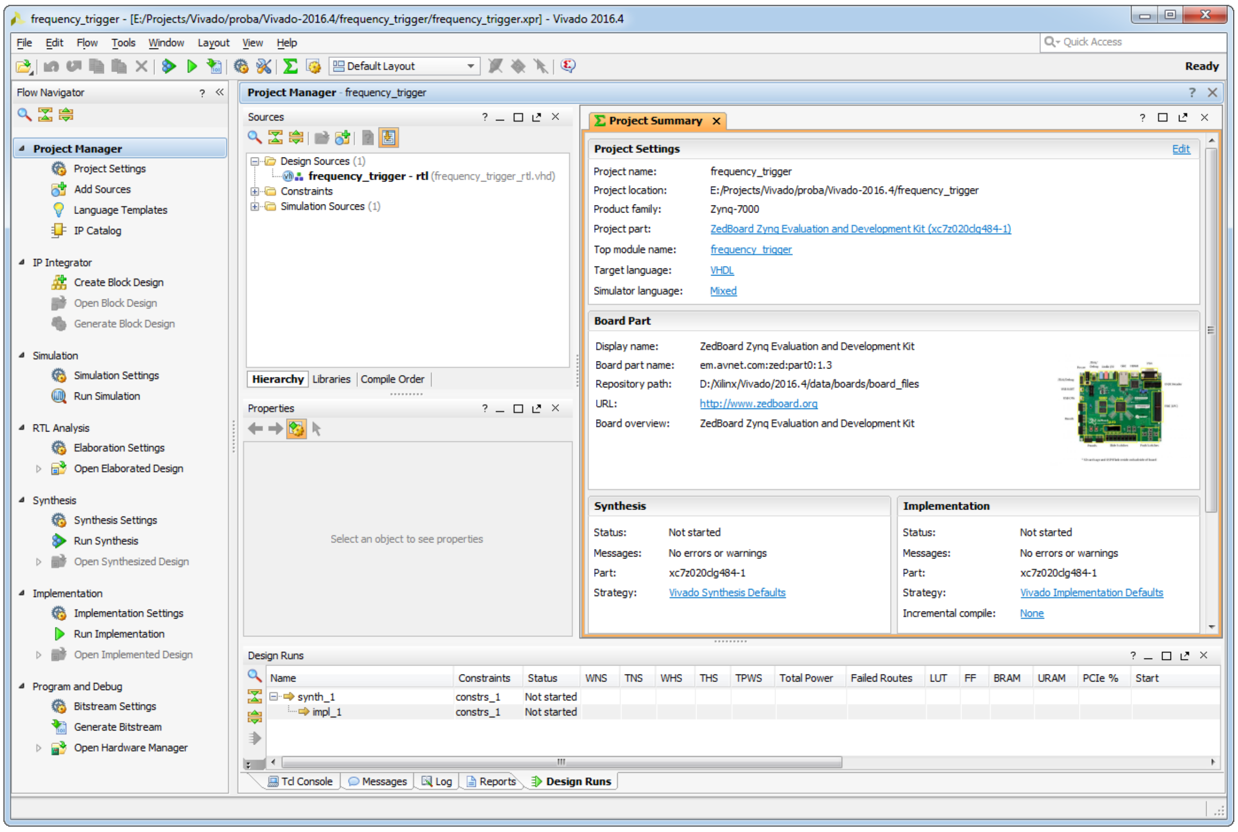
No constraints files will be added. Use Add Sources to add them later.

The default part and product family for the new project:
Default Board: ZedBoard Zynq Evaluation and Development Kit
Default Part: xc7z020clg484-1
Product: Zynq-7000
Family: Zynq-7000
Package: clg484
Speed Grade: -1

To create the project, click Finish

Figure 13.12: New Project Summary dialog box

After we finished with the new project creation, in a few seconds Vivado IDE will appear with the created ***frequency_trigger*** project, see Illustration 13.13



Project Settings

Project name: frequency_trigger
Project location: E:/Projects/Vivado/proba/Vivado-2016.4/frequency_trigger
Product family: Zynq-7000
Project part: ZedBoard Zynq Evaluation and Development Kit (xc7z020clg484-1)
Top module name: frequency_trigger
Target language: VHDL
Simulator language: Mixed

Board Part

Display name: ZedBoard Zynq Evaluation and Development Kit
Board part name: em.avnet.com:zed:part0:1.3
Repository path: D:/Xilinx/Vivado/2016.4/data/boards/board_files
URL: http://www.zedboard.org
Board overview: ZedBoard Zynq Evaluation and Development Kit

Synthesis

Status: Not started
Messages: No errors or warnings
Part: xc7z020clg484-1
Strategy: Vivado Synthesis Defaults

Implementation

Status: Not started
Messages: No errors or warnings
Part: xc7z020clg484-1
Strategy: Vivado Implementation Defaults
Incremental compile: None

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAM	URAM	PCIe %	Start
synth_1	constrs_1	Not started													
impl_1	constrs_1	Not started													

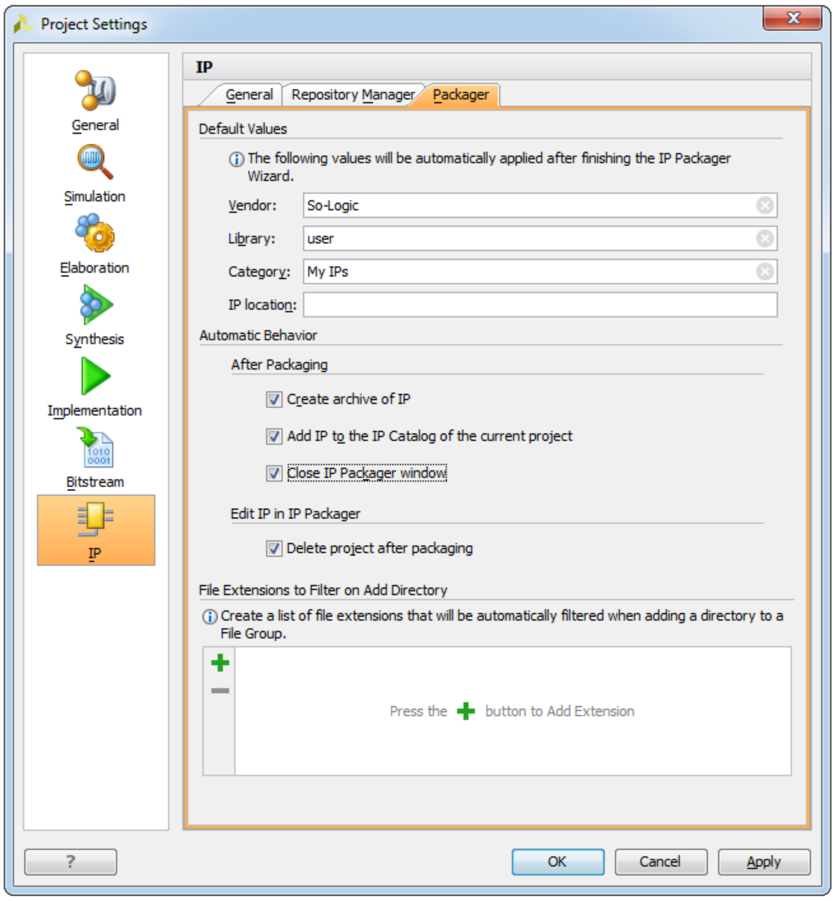
Figure 13.13: Created new frequency_trigger project

Step 13. In the Vivado **Flow Navigator**, under the **Project Manager**, click **Project Settings** command and choose **IP** from the left pane, see Illustration 13.14. Global IP project settings are available to help you be more productive when customizing IP.

Step 14. In the **IP** window, select **Packager** tab and fill the fields as it is shown on the Illustration 13.14.

Packager sets default values for packaging new IP, including vendor, library and taxonomy. This category also allows you to set the default behavior when opening the IP Packager and allows you to specify file extension to be filtered automatically. If necessary, you can change the default values for packaging IP during the IP packaging process.

Note: Ensure that the **Create archive of IP** option is enabled in the **After Packaging** section to deliver an archive file.



IP

Default Values

The following values will be automatically applied after finishing the IP Packager Wizard.

Vendor: So-Logic

Library: user

Category: My IPs

IP location:

Automatic Behavior

After Packaging

[✓] Create archive of IP

[✓] Add IP to the IP Catalog of the current project

[✓] Close IP Packager window

Edit IP in IP Packager

[✓] Delete project after packaging

File Extensions to Filter on Add Directory

Create a list of file extensions that will be automatically filtered when adding a directory to a File Group.

Figure 13.14: Packager window with configured settings that will be applied after packaging process

Our next step will be to package ***frequency_trigger*** project. To package a Vivado project as IP, do the following:

Step 15. In the main Vivado IDE menu, select **Tools** -> **Create and Package IP...** option, see Illustration 13.15

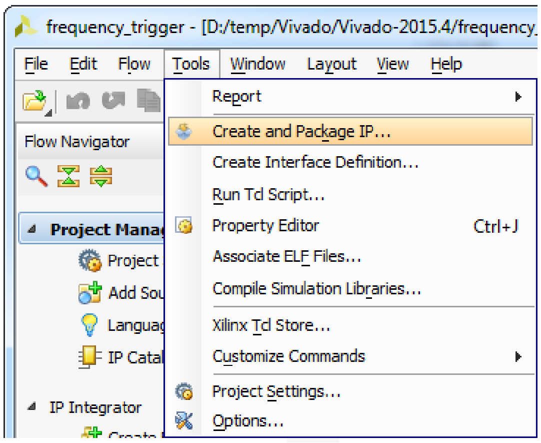
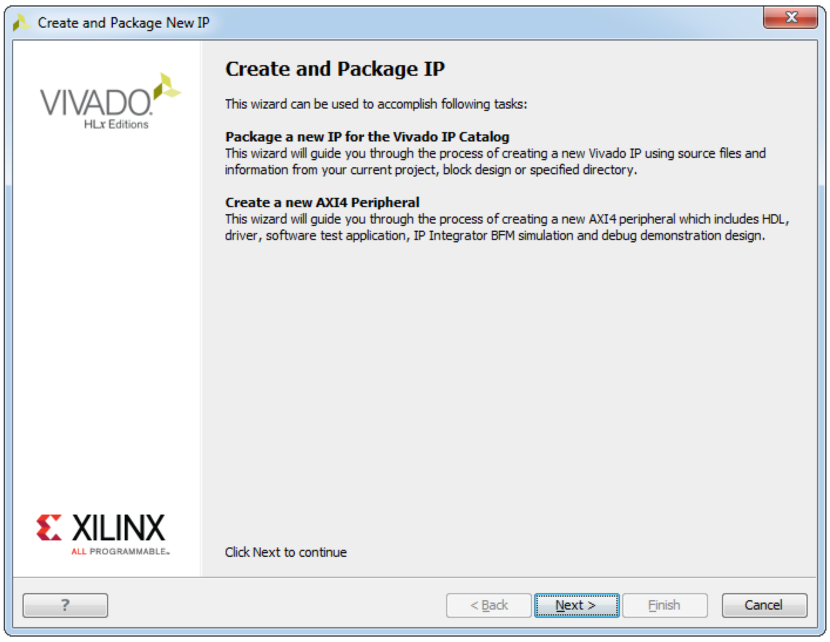


Figure 13.15: Create and Package IP option

Step 16. In the **Create and Package IP** dialog box, click **Next**, see Illustration 13.16


Create and Package New IP

Create and Package IP

This wizard can be used to accomplish following tasks:

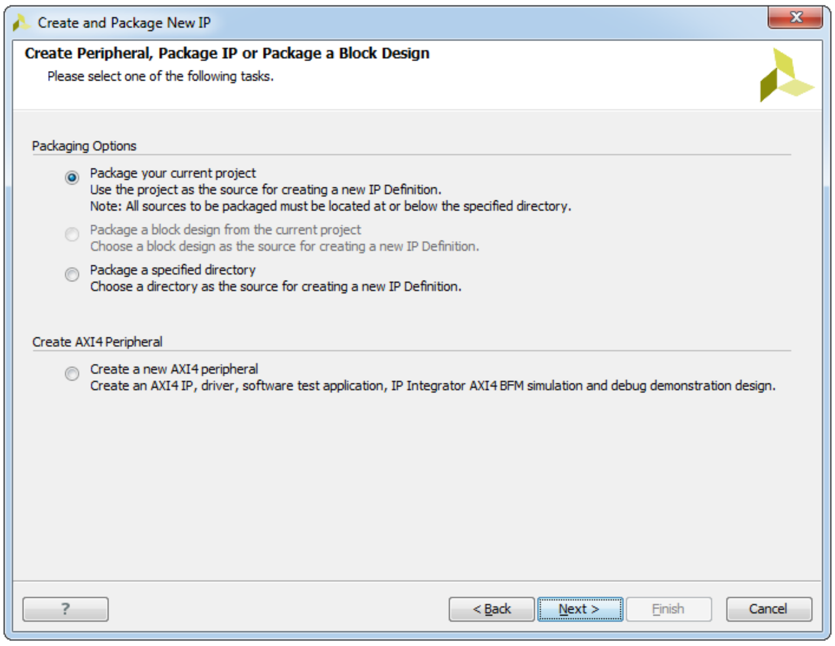
Package a new IP for the Vivado IP Catalog
This wizard will guide you through the process of creating a new Vivado IP using source files and information from your current project, block design or specified directory.

Create a new AXI4 Peripheral
This wizard will guide you through the process of creating a new AXI4 peripheral which includes HDL, driver, software test application, IP Integrator BFM simulation and debug demonstration design.

Click Next to continue

Figure 13.16: Create and Package IP dialog box

Step 17. In the **Choose Create Peripheral or Package IP** dialog box, choose **Package your current project** option and click **Next**, see Illustration 13.17


Create and Package New IP

Create Peripheral, Package IP or Package a Block Design
Please select one of the following tasks.

Packaging Options

● Package your current project
Use the project as the source for creating a new IP Definition.
Note: All sources to be packaged must be located at or below the specified directory.

○ Package a block design from the current project
Choose a block design as the source for creating a new IP Definition.

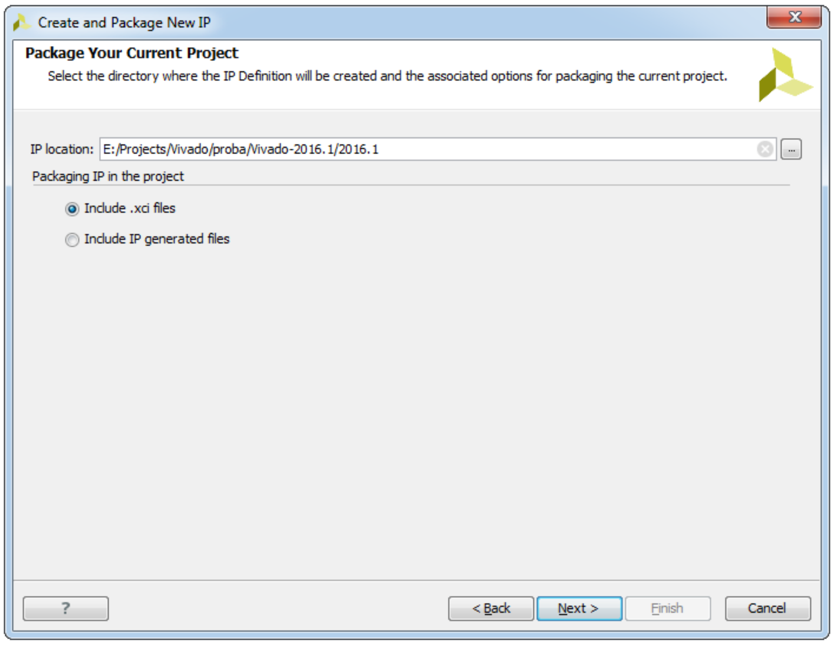
○ Package a specified directory
Choose a directory as the source for creating a new IP Definition.

Create AXI4 Peripheral

○ Create a new AXI4 peripheral
Create an AXI4 IP, driver, software test application, IP Integrator AXI4 BFM simulation and debug demonstration design.

Figure 13.17: Choose Create Peripheral or Package IP dialog box

Step 18. In the **Package Your Current Project** dialog box, choose **IP Location** and type of the **Packaging IP in the project**, see Illustration 13.18



Package Your Current Project

Select the directory where the IP Definition will be created and the associated options for packaging the current project.

IP location: E:/Projects/Vivado/proba/Vivado-2016.1/2016.1

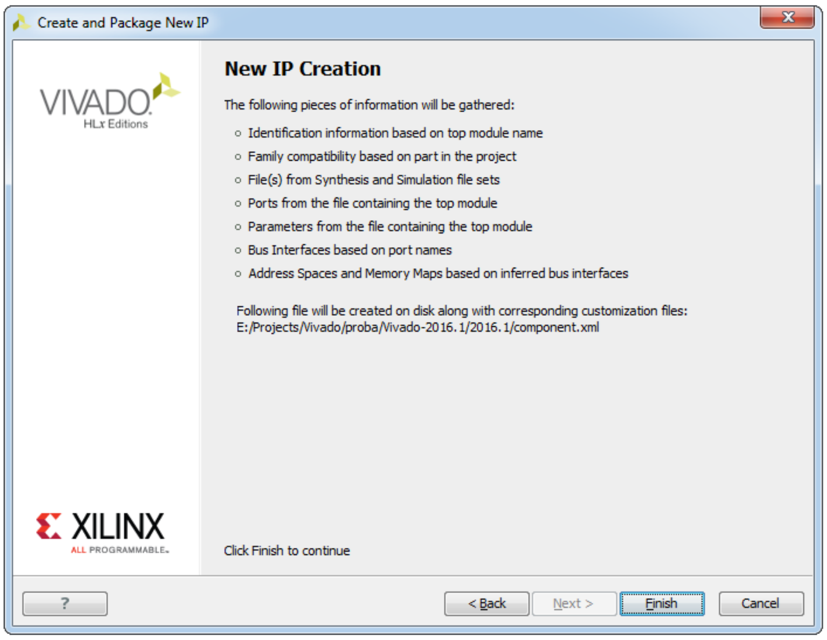
Packaging IP in the project

- [✓ Include .xci files]
- Include IP generated files

Figure 13.18: Package Your Current Project dialog box

- **IP Location**: The directory in which the tool creates the IP Definition. The default is the project sources directory.
- **Packaging IP in the project**:
  - **Include *.xci* files**: If the project you are packaging includes subcores, package only the IP customization XCI file. By deciding to include the XCI files, the IP Packager packages only the XCI file of the IP customization. This creates a subcore reference to the parent IP and allows the packaged XCI file to be managed by the Vivado tool. The advantage is that the IP can easily be upgraded to the latest release by using the Vivado IP Upgrade methodology.
  - **Include IP generated files**: Packages the generated RTL and XDC sources of the IP customization.

Step 19. In the **New IP Creation** dialog box, click **Finish**, see Illustration 13.19.



New IP Creation

The following pieces of information will be gathered:

- Identification information based on top module name
- Family compatibility based on part in the project
- File(s) from Synthesis and Simulation file sets
- Ports from the file containing the top module
- Parameters from the file containing the top module
- Bus Interfaces based on port names
- Address Spaces and Memory Maps based on inferred bus interfaces

Following file will be created on disk along with corresponding customization files:
E:/Projects/Vivado/proba/Vivado-2016.1/2016.1/component.xml

Click Finish to continue

Figure 13.19: New IP Creation dialog box

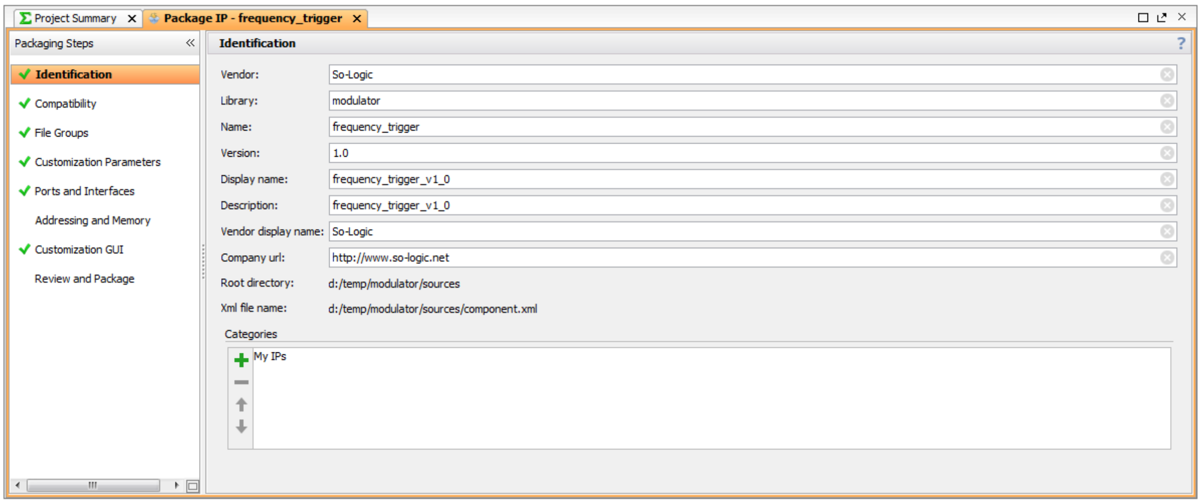
If you have selected either **Package your current project** or **Package a specified directory** option, the **New IP Creation** dialog box opens automatically to summarize the information the wizard gathered about the project, and creates a basic IP package in a staging area as shown on the illustration above.

Step 20. In the **Package IP** dialog box, click **OK** and **Package IP - frequency_trigger** window will automatically appear on the right side of the Vivado IDE, see Illustration 13.20

Review the IP Packaging steps in the Package IP page:

- **Identification**: Information used to identify your IP
- **Compatibility**: Configure the parts and/or families of Xilinx devices that are compatible with your IP
- **File Groups**: Individual files for your IP are grouped into specific file groups
- **Customization Parameters**: Specify the parameters to customize your IP
- **Ports and Interfaces**: Top-level ports and interfaces for your IP
- **Addressing and Memory**: Specify the memory-maps or address spaces
- **Customization GUI**: Configure the parameters that appear on each page of the Customization GUI
- **Review and Package**: Summary of the IP and repackaging

Step 21. In the **Package IP - frequency_trigger** window, in the **Identification** section, fill in fields as it is shown on the Illustration 13.20



Vendor: So-Logic
Library: modulator
Name: frequency_trigger
Version: 1.0
Display name: frequency_trigger_v1_0
Description: frequency_trigger_v1_0
Vendor display name: So-Logic
Company url: http://www.so-logic.net
Root directory: d:/temp/modulator/sources
Xml file name: d:/temp/modulator/sources/component.xml

Figure 13.20: Identification window

As you can see from the illustration above, Package IP wizard automatically choose **MyIPs** category, as the default category to store packaged IP.

The **Categories** option allows the IP designer to select various categories to help classify the new IP Definition. When IP definition is added to the IP Catalog, the IP will be listed under the specified categories.

Step 22. After we finished with the IP **Identification**, select the **Review and Package** option in the **Package IP** window and check the specified project directory folder to make sure that the new archive file was added, see Illustration 13.21.

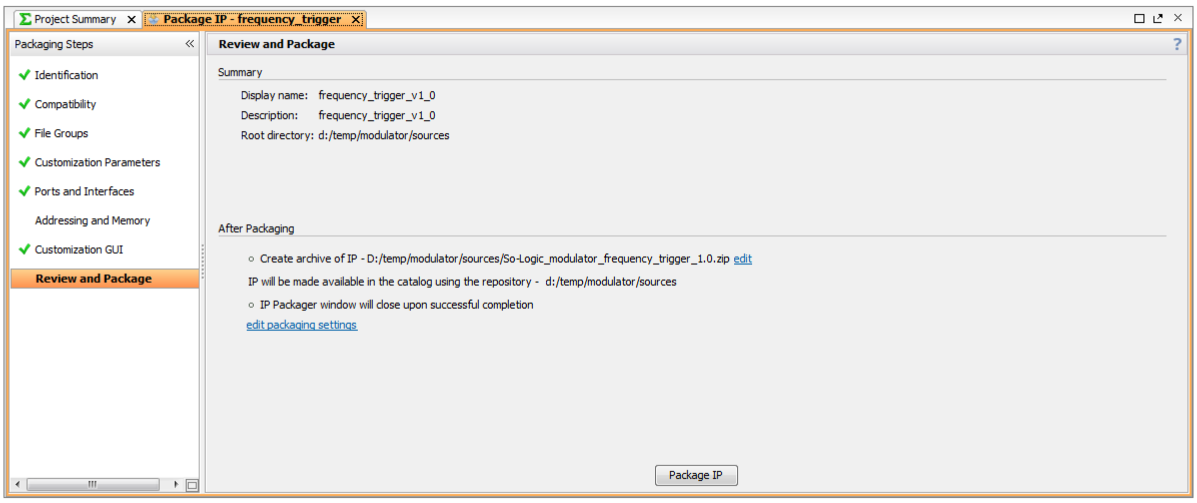
The default naming convention for the archive is:

<vendor>_<library>_<name>_<version>.zip

In our case, the name of the zip file should be:

So-Logic_modulator_frequency_trigger_1.0.zip

The user can change the default name and location of the archive by selecting the **edit** link next to the **Create archive of IP** name in the **After Packaging** selection, see Illustration 13.21.



Review and Package

Summary

Display name: frequency_trigger_v1_0
Description: frequency_trigger_v1_0
Root directory: d:/temp/modulator/sources

After Packaging

- Create archive of IP - D:/temp/modulator/sources/So-Logic_modulator_frequency_trigger_1.0.zip edit

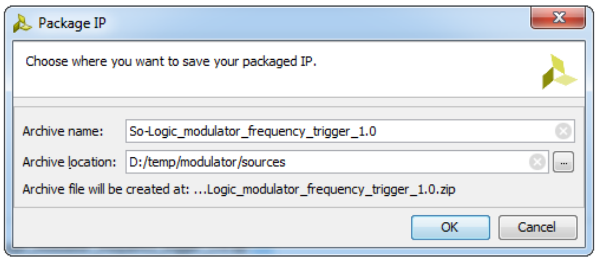
IP will be made available in the catalog using the repository - d:/temp/modulator/sources

- IP Packager window will close upon successful completion

edit packaging settings

Figure 13.21: Review and Package window

Step 23. Click **edit** link next to the **Create archive of IP** name in the **After Packaging** selection to change the name and the location of the archive, see Illustration 13.22



Choose where you want to save your packaged IP.

Archive name: So-Logic_modulator_frequency_trigger_1.0
Archive location: D:/temp/modulator/sources
Archive file will be created at: ...Logic_modulator_frequency_trigger_1.0.zip

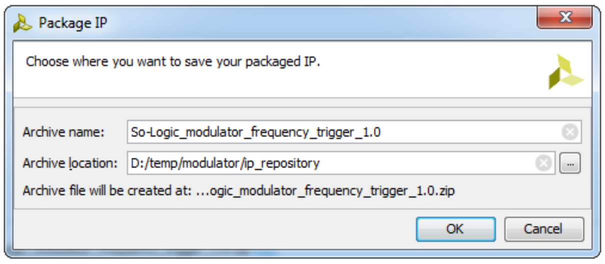
Figure 13.22: Package IP dialog box

Step 24. In the **Package IP** dialog box, change the **Archive name** to be:

So-Logic_modulator_frequency_trigger_1.0.zip

Step 25. Before you change the **Archive location**, create a new folder, ***ip_repository***, in the same folder where the ***frequency_trigger*** project was created. This new folder will be a place where we will keep all IPs (.zip files) that we will create.

Step 26. In the **Package IP** dialog box, change the **Archive location** to the new ***ip_repository*** folder, see Illustration 13.23

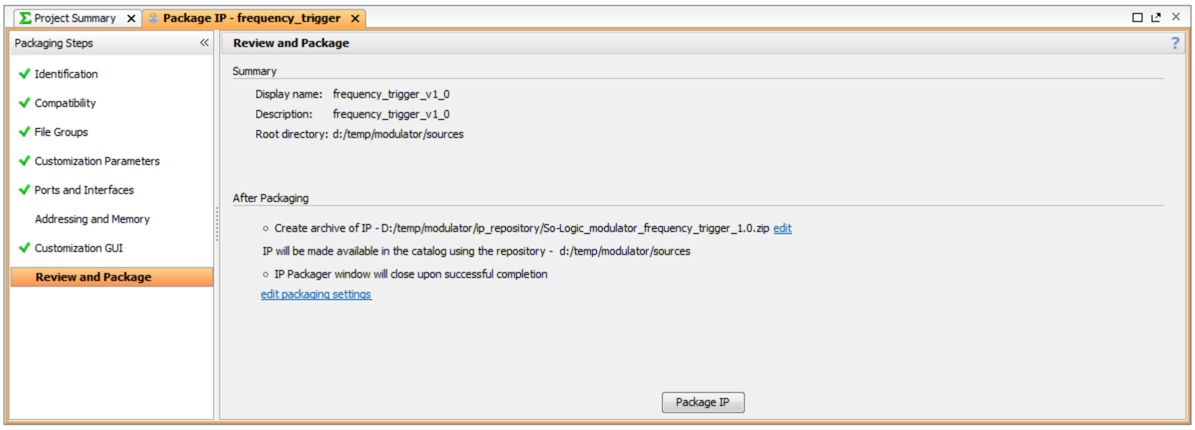


Choose where you want to save your packaged IP.

Archive name: So-Logic_modulator_frequency_trigger_1.0
Archive location: D:/temp/modulator/ip_repository
Archive file will be created at: ...ogic_modulator_frequency_trigger_1.0.zip

Figure 13.23: Package IP dialog box with selected new archive location

Step 27. Click **OK** and you should see all the modifications that we made in the **After Packaging** sector of the **Review and Package** window, see Illustration 13.24



Review and Package

Summary

Display name: frequency_trigger_v1_0
Description: frequency_trigger_v1_0
Root directory: d:/temp/modulator/sources

After Packaging

- Create archive of IP - D:/temp/modulator/ip_repository/So-Logic_modulator_frequency_trigger_1.0.zip edit

IP will be made available in the catalog using the repository - d:/temp/modulator/sources

- IP Packager window will close upon successful completion

edit packaging settings

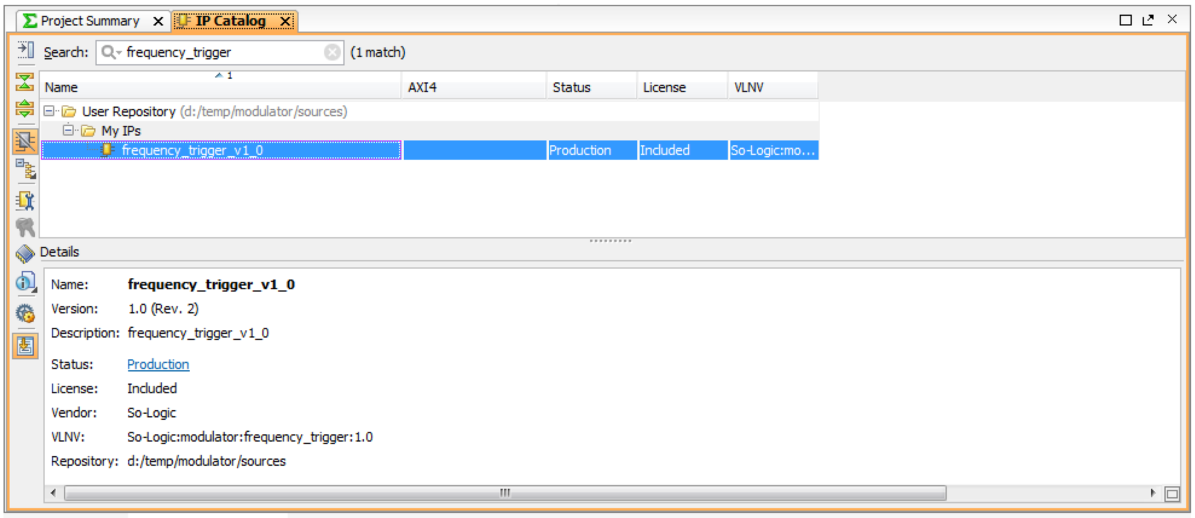
Figure 13.24: Review and Package window with new archive information

Step 28. If you are satisfied with the Package IP information, click the **Package IP** button at the bottom of the **Review and Package** window to finish with the ***frequency_trigger*** IP packaging process

Step 29. In the **Flow Navigator**, under the **Project Manager**, click **IP Catalog** command to verify the presence of our ***frequency_trigger*** IP in the IP Catalog

Step 30. In the **IP Catalog**, search for the ***frequency_trigger_v1_0*** IP, see Illustration 13.25

If you select the ***frequency_trigger_v1_0*** IP, all the data that we entered in the process of the IP creation should appear in the **Details** window, see Illustration 13.25.



Name	AXI4	Status	License	VLNV
User Repository (d:/temp/modulator/sources)				
My IPs				
frequency_trigger_v1_0		Production	Included	So-Logic:mo...

Name: frequency_trigger_v1_0
Version: 1.0 (Rev. 2)
Description: frequency_trigger_v1_0
Status: Production
License: Included
Vendor: So-Logic
VLNV: So-Logic:modulator:frequency_trigger:1.0
Repository: d:/temp/modulator/sources

Figure 13.25: frequency_trigger IP in the IP Catalog

Now, when you know the procedure for IP creation, repeat the steps (1-29) to create the rest of the IPs (*counter*, *digital_sine* and *pwm*), necessary for the Modulator project, with the following exceptions:

Counter IP:

- Name the project **"counter"** when you start new project creation
- In the project creation process, in the **Add Source Files** dialog box, choose ***counter_rtl.vhd*** source file and click **OK**
- In the **Packager IP** wizard, in the **Review and Package** window, click **edit** link next to the **Create archive of IP** name in the **After Packaging** selection to change the name and the location of the archive:
  - Change the **Archive name** to be: ***So-Logic_modulator_counter_1.0.zip***

- Change the **Archive location** to the new ***ip_repository*** folder

Digital Sine IP:

- Name the project **"digital_sine"** when you start new project creation
- In the project creation process, in the **Add Source Files** dialog box, choose ***sine_rtl.vhd*** and ***modulator_pkg.vhd*** source files and click **OK**
- In the **Packager IP** wizard, in the **Review and Package** window, click **edit** link next to the **Create archive of IP** name in the **After Packaging** selection to change the name and the location of the archive:
  - Change the **Archive name** to be:

    So-Logic_modulator_digital_sine_1.0.zip
  - Change the **Archive location** to the new ***ip_repository*** folder

Pwm IP:

- Name the project **"pwm"** when you start new project creation
- In the project creation process, in the **Add Source Files** dialog box, choose ***pwm_rtl.vhd*** and ***frequency_trigger-_rtl.vhd*** source files and click **OK**
- In the **Packager IP** wizard, in the **Review and Package** window, click **edit** link next to the **Create archive of IP** name in the **After Packaging** selection to change the name and the location of the archive:
  - Change the **Archive name** to be: ***So-Logic_modulator_pwm_1.0.zip***
  - Change the **Archive location** to the new ***ip_repository*** folder

Now, when all IPs are created, it's time to create a new project, ***modulator_ip***, where we will instantiate these IPs.

Step 31. Create new Vivado project, ***modulator_ip***, without adding any source file

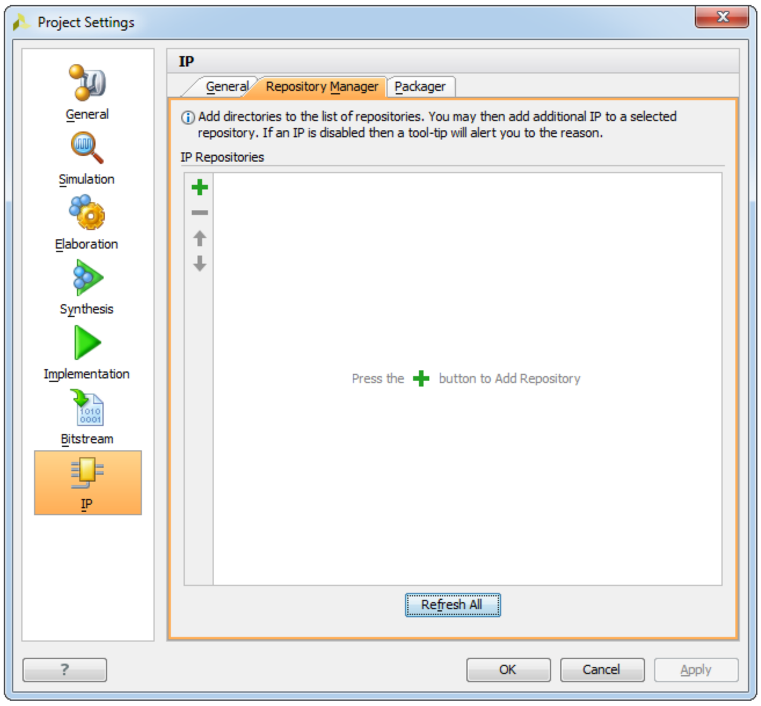
The following steps will show you how to add packaged IP to the IP Catalog:

Step 32. Open ***ip_repository*** folder with packaged IPs (.zip files) and extract each IP separately

Step 33. Then, In the **Flow Navigator**, under the **Project Manager** , click **Project Settings** command and choose **IP** from the left pane

Step 34. In the **IP** window, select **Repository Manager** tab, see Illustration 13.26

Repository Manager lets you add or remove user repositories and establish precedence between repositories.



IP

Add directories to the list of repositories. You may then add additional IP to a selected repository. If an IP is disabled then a tool-tip will alert you to the reason.

IP Repositories

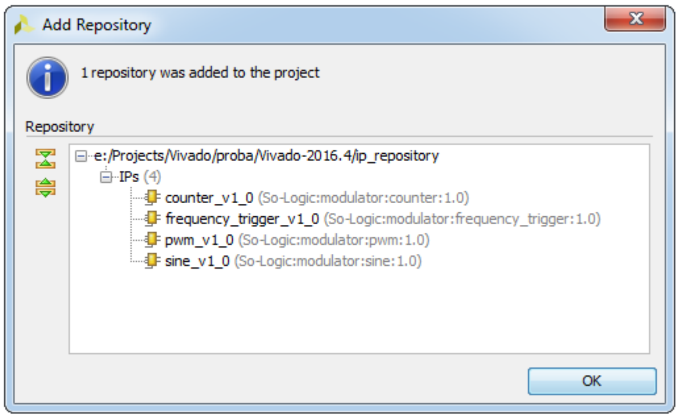
Press the + button to Add Repository

Figure 13.26: Repository Manager window

Step 35. In the **Repository Manager** window, click + icon to add the desired repository, see Illustration 13.26

Step 36. In the **IP Repositories** window, choose ***ip_repository*** folder and click **Select**

Step 37. In the **Add Repository** dialog box, click **OK** to add the selected repository (*ip_repository* with 4 IPs) to the project, see Illustration 13.27

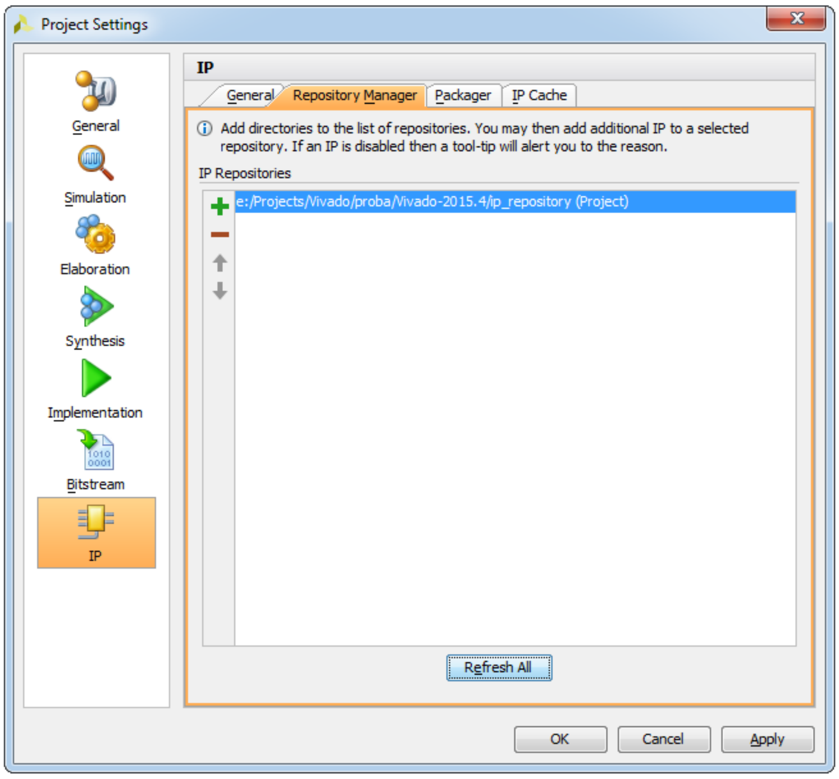


1 repository was added to the project

```
e:/Projects/Vivado/proba/Vivado-2016.4/ip_repository
  IPs (4)
    counter_v1_0 (So-Logic:modulator:counter:1.0)
    frequency_trigger_v1_0 (So-Logic:modulator:frequency_trigger:1.0)
    pwm_v1_0 (So-Logic:modulator:pwm:1.0)
    sine_v1_0 (So-Logic:modulator:sine:1.0)
```

Figure 13.27: Add Repository dialog box

Step 38. In the **Repository Manager** window, when ***ip_repository*** is added to the **IP Repositories** section, click **OK**, see Illustration 13.28



IP

Add directories to the list of repositories. You may then add additional IP to a selected repository. If an IP is disabled then a tool-tip will alert you to the reason.

IP Repositories

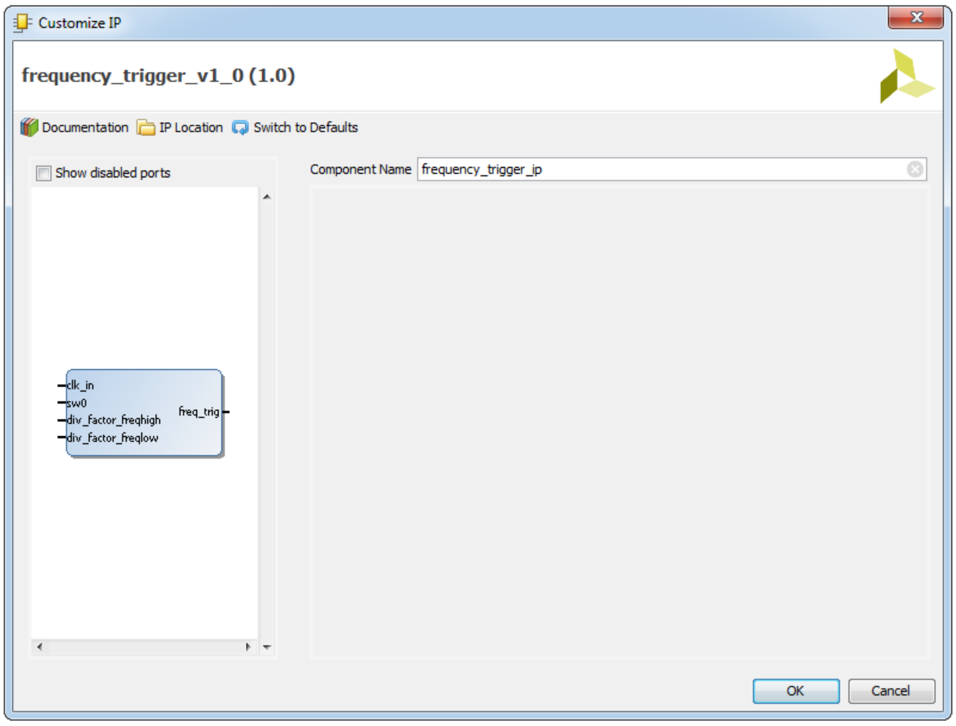
e:/Projects/Vivado/proba/Vivado-2015.4/ip_repository (Project)

Figure 13.28: Repository Manager with selected ip_repository

Step 39. In the **Flow Navigator**, under the **Project Manager**, click **IP Catalog** command to verify the presence of the previously created IPs in the IP Catalog.

Step 40. Double-click on the **frequency_trigger_v1_0** IP core and Vivado IDE will create a new skeleton source for your IP

The window that will be opened is used to set up the general **frequency_trigger** core parameters, see Illustration 13.29

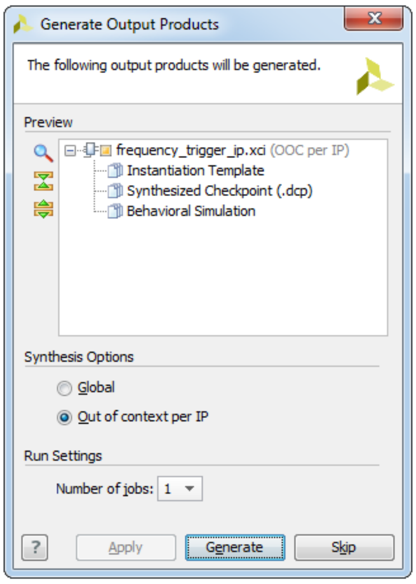


Component Name: frequency_trigger_ip

Figure 13.29: frequency_trigger IP configuration window

Step 41. In the **frequency_trigger_v1_0 (1.0)** dialog box, change the **Component Name** to be ***frequency_trigger_ip*** and click **OK**

Step 42. In the **Generate Output Products** dialog box, click **Generate**, see Illustration 13.30



The following output products will be generated.

```
frequency_trigger_ip.xci (OOC per IP)
    Instantiation Template
    Synthesized Checkpoint (.dcp)
    Behavioral Simulation
```

Number of jobs: 1

Figure 13.30: Generate Output Products window for frequency_trigger_ip core

Note: After **frequency_trigger_ip** core generation, your **frequency_trigger_ip** core should appear in the **Sources** window, see Illustration 13.31

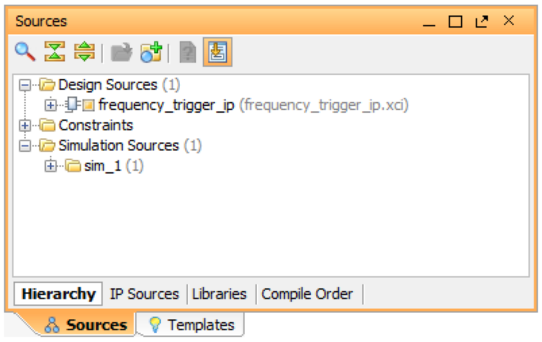
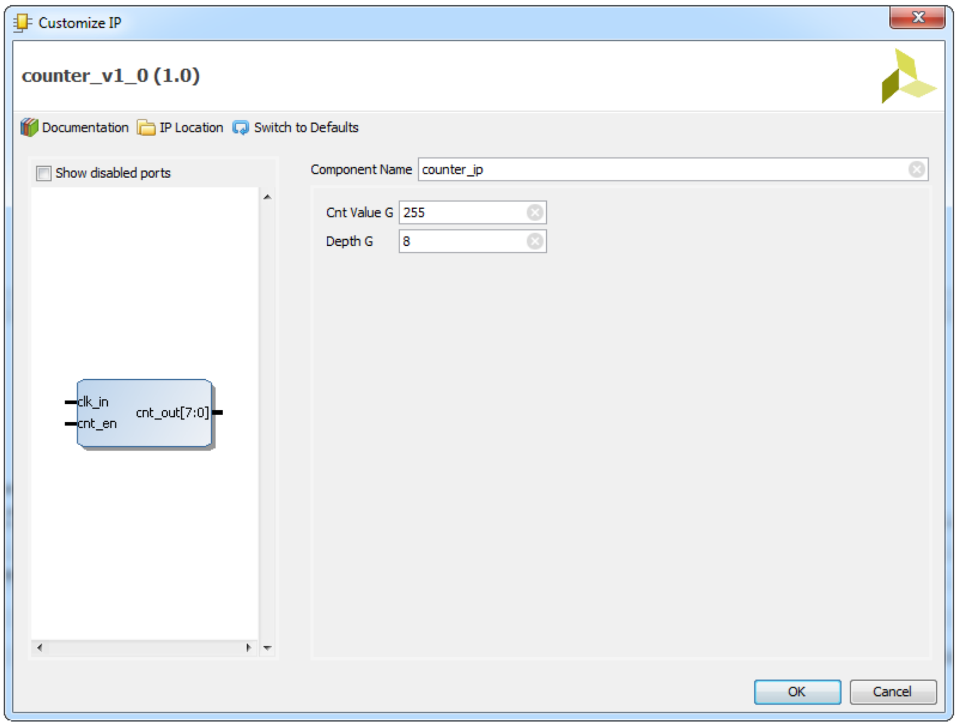


Figure 13.31: Sources window with generated frequency_trigger_ip IP

After we generate **frequency_trigger_ip** IP, we should repeat the same procedure for the **counter_v1_0** IP:

Step 43. In the **IP Catalog**, double-click on the **counter_v1_0** IP core and Vivado IDE will create a new skeleton source for the counter_v1_0 IP

The window that will be opened is used to set up the general **counter** core parameters, see Illustration 13.32



Component Name: counter_ip
Cnt Value G: 255
Depth G: 8

Figure 13.32: counter IP configuration window

Step 44. In the **counter _v1_0 (1.0)** dialog box, change the **Component Name** to be ***counter_ip*** and configure the rest of the parameters:

- **Cnt Value G** to be **255**
- **Depth G** to be **8**

Note: To know how to configure the right values, open the ***modulator_rtl.vhd*** source file and find out how it is done in the original design:

```
counterampl : entity work.counter(rtl)      -- counter module instance
    generic map(
        cnt_value_g => design_setting_g.cntampl_value,
        depth_g     => design_setting_g.depth
        )
```

Step 45. Click **OK**

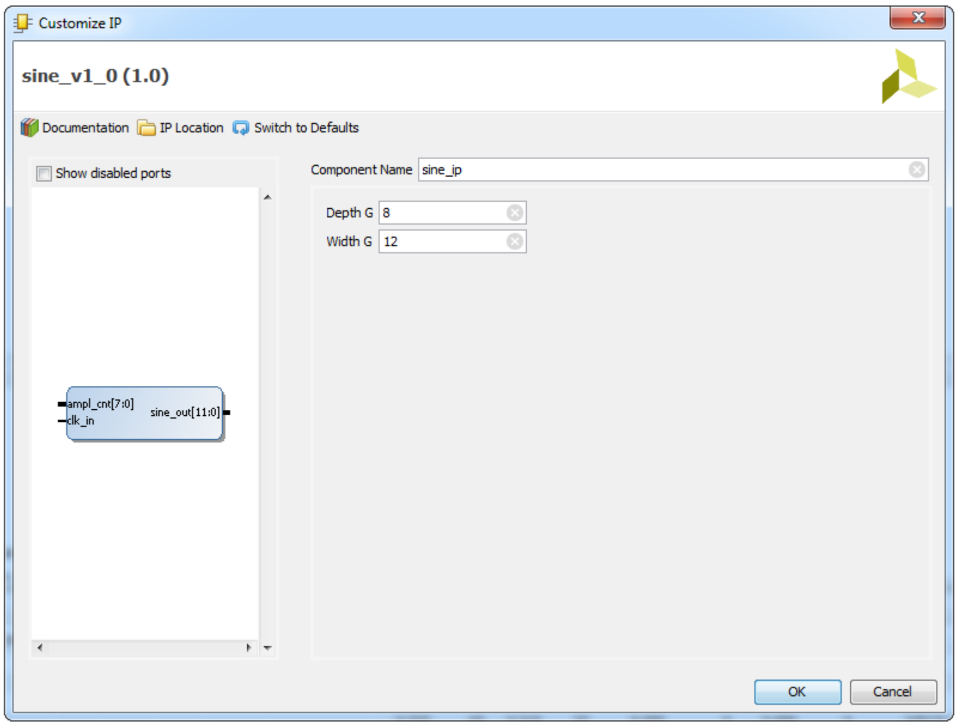
Step 46. In the **Generate Output Products** dialog box, click **Generate**

Note: After **counter_ip** core generation, your **counter_ip** core should appear in the Sources window.

After we generate **frequency_trigger_ip** and **counter_ip** IPs, we should repeat the same procedure for the **sine_v1_0** IP:

Step 47. In the **IP Catalog**, double-click on the **sine_v1_0** IP core and Vivado IDE will create a new skeleton source for the sine_v1_0 IP

The window that will be opened is used to set up the general **sine** core parameters, see Illustration 13.33



Component Name sine_ip

Depth G 8

Width G 12

Figure 13.33: sine IP configuration window

Step 48. In the **sine _v1_0 (1.0)** dialog box, change the **Component Name** to be ***sine_ip*** and configure the rest of the parameters:

- **Depth G** to be **8**
- **Width G** to be **12**

Note: To know how to configure the right values, open the ***modulator_rtl.vhd*** source file and find out how it is done in the original design:

```
sine : entity work.sine(rtl)     -- digital sine module instance
    generic map (
        depth_g => design_setting_g.depth,
        width_g => design_setting_g.width
        )
```

Step 49. Click **OK**

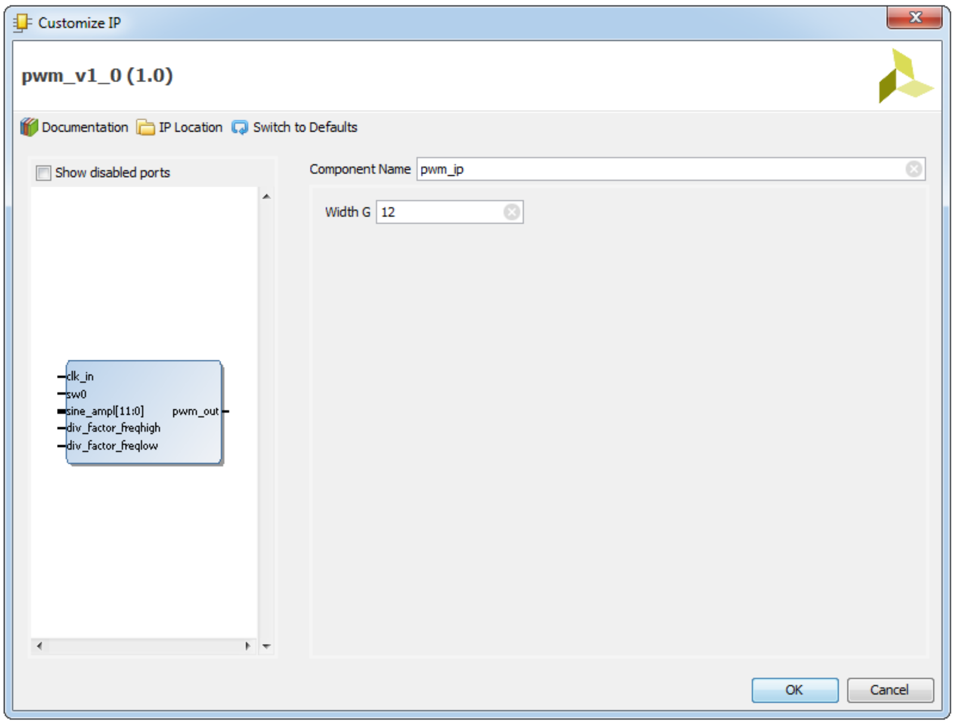
Step 50. In the **Generate Output Products** dialog box, click **Generate**

Note: After **sine_ip** core generation, your **sine_ip** core should appear in the Sources window.

After we generate **frequency_trigger_ip**, **counter_ip** and **sine_ip** IPs, we should repeat the same procedure for the last **pwm_v1_0** IP:

Step 51. In the **IP Catalog**, double-click on the **pwm_v1_0** IP core and Vivado IDE will create a new skeleton source for the pwm_v1_0 IP

The window that will be opened is used to set up the general **pwm** core parameters, see Illustration 13.34



Component Name pwm_ip

Width G 12

Figure 13.34: pwm IP configuration window

Step 52. In the **pwm _v1_0 (1.0)** dialog box, change the **Component Name** to be **pwm_ip** and configure the rest of the parameters:

- **Width G** to be **12**

Note: To know how to configure the right values, open the ***modulator_rtl.vhd*** source file and find out how it is done in the original design:

```
pwmmodule : entity work.pwm (rtl)    -- pwm module instance
    generic map (
        width_g => design_setting_g.width,
        )
```

Step 53. Click **OK**

Step 54. In the **Generate Output Products** dialog box, click **Generate**

Note: After **pwm_ip** core generation, all the generated cores should appear in the **Sources** window, see Illustration 13.35

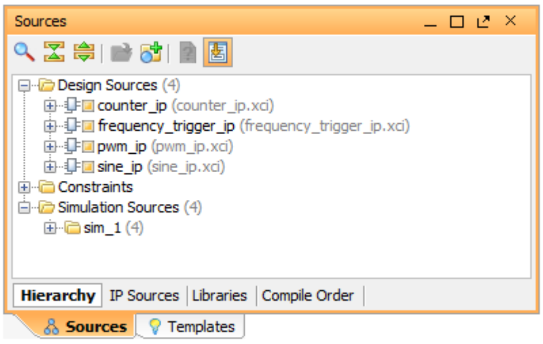


Figure 13.35: Sources window with all four generated IPs

After configuring and generating all four necessary IPs (**frequency_trigger_ip**, **counter_ip**, **sine_ip** and **pwm_ip**), we will create a top-level VHDL module, **modulator_ip_rtl.vhd**, where we will connect these IPs, see Figure 13.36.

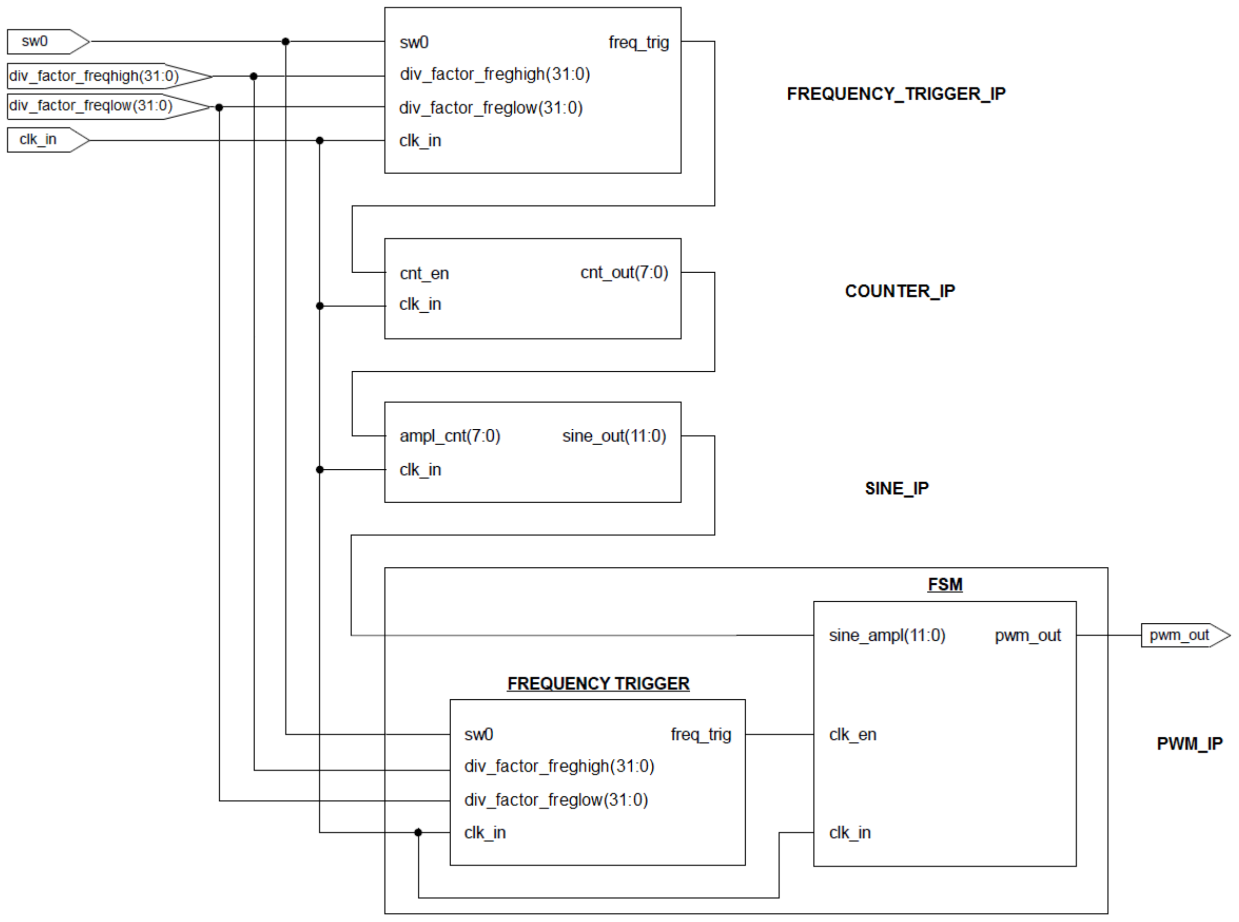


Figure 13.36: Connection between generated IPs

To create a module, use steps for creating modules, **Chapter 2.4.1 Creating a Module Using Vivado Text Editor**.

modulator_ip_rtl.vhd:

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

    use work.modulator_pkg.all;

library unisim;
    use unisim.vcomponents.all;


entity modulator_ip is
    generic (
        -- User defined settings for the pwm design
        design_setting_g : design_setting_t_rec := design_setting_c
    );

    port (
        clk_in              : in std_logic;                      -- input clock signal
        sw0                 : in std_logic;                      -- signal made for selecting frequency
        div_factor_freqhigh : in std_logic_vector (31 downto 0); -- input clock division when sw0 = '1'
        div_factor_freqlow  : in std_logic_vector (31 downto 0); -- input clock division when sw0 = '0'
        pwm_out             : out std_logic                      -- pulse width modulated signal
    );
end entity;


architecture rtl of modulator_ip is

    signal ampl_cnt_s    : std_logic_vector (7 downto 0);
    signal sine_ampl_s  : std_logic_vector (11 downto 0);
    signal freq_trig_s : std_logic;

    -- frequency_trigger_ip component definition
    component frequency_trigger_ip
        port (
            clk_in              : in std_logic;
            sw0                 : in std_logic;
            div_factor_freqhigh : in std_logic_vector (31 downto 0);
            div_factor_freqlow  : in std_logic_vector (31 downto 0);
            freq_trig           : out std_logic
        );
    end component;

    -- counter_ip component definition
    component counter_ip
        port (
            clk_in  : in std_logic;
            cnt_en  : in std_logic;
            cnt_out : out std_logic_vector (7 downto 0)
        );
    end component;

    -- sine_ip component definition
    component sine_ip
        port (
            clk_in   : in std_logic;
            ampl_cnt : in std_logic_vector (7 downto 0);
            sine_out : out std_logic_vector (11 downto 0)
        );
    end component;

    -- pwm_ip component definition
    component pwm_ip
        port (
            clk_in              : in std_logic;
            sw0                 : in std_logic;
            sine_ampl           : in std_logic_vector (11 downto 0);
            div_factor_freqhigh : in std_logic_vector (31 downto 0);
            div_factor_freqlow  : in std_logic_vector (31 downto 0);
            pwm_out             : out std_logic
        );
    end component;


begin

    -- frequency_trigger_ip component instance
    freq_trig: frequency_trigger_ip
        port map (
            clk_in              => clk_in,
            sw0                 => sw0,
            div_factor_freqhigh => div_factor_freqhigh,
            div_factor_freqlow  => div_factor_freqlow,
            freq_trig           => freq_trig_s
        );

    -- counter_ip component instance
    counter: counter_ip
        port map (
            clk_in  => clk_in,
            cnt_en  => freq_trig_s,
            cnt_out => ampl_cnt_s
        );

    -- sine_ip component instance
    sine: sine_ip
        port map (
            clk_in   => clk_in,
            ampl_cnt => ampl_cnt_s,
            sine_out => sine_ampl_s
        );

    -- pwm_ip component instance
    pwm: pwm_ip
        port map (
            clk_in              => clk_in,
            sw0                 => sw0,
            sine_ampl           => sine_ampl_s,
            div_factor_freqhigh => conv_std_logic_vector(conv_integer(div_factor_freqhigh)/(2**
      design_setting_g.width), 32),
            div_factor_freqlow  => conv_std_logic_vector(conv_integer(div_factor_freqlow)/(2**
      design_setting_g.width), 32),
            pwm_out             => pwm_out
        );
end;
```

After we finished with the ***modulator_ip_rtl.vhd*** module creation, we should create new ***modulator_ip_wrapper_rtl.vhd*** module in the same way as it was done for the Modulator module example, see **Chapter 9. MODULATOR WRAPPER**.

The block diagram and source code of the Modulator IP wrapper is shown in the text below.

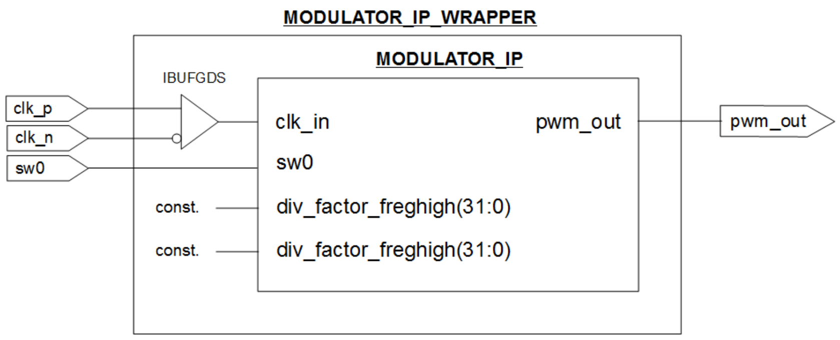


Figure 13.37: Modulator IP wrapper block diagram

Modulator IP wrapper VHDL model:

```
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 modulator_ip_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 rtl of modulator_ip_wrapper is

    -- input clock signal
    signal clk_in_s : std_logic;

    -- c1_c = fclk/(2^depth*2^width)                   - c1_c = 95.3674, fclk = 100 MHz
    constant c1_c : real :=
       get_board_info_f(board_name_g).fclk/(real((2**design_setting_g.depth)*(2**design_setting_g.width)));
     -- div_factor_freqhigh_c = (c1_c/f_high)*2^width  - threshold value of frequency a = 110592
    constant div_factor_freqhigh_c : integer :=
       integer(c1_c/design_setting_g.f_high)*(2**design_setting_g.width);
    -- div_factor_freqlow_c  = (c1_c/f_low)*2^width    - threshold value of frequency b = 389120
    constant div_factor_freqlow_c  : integer :=
       integer(c1_c/design_setting_g.f_low)*(2**design_setting_g.width);

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;


    modulatorip : entity work.modulator_ip   -- modulator_ip module instance
        generic map(
            design_setting_g => design_setting_g
            )

        port map(
            clk_in              => clk_in_s,
            sw0                 => sw0,
            div_factor_freqhigh => conv_std_logic_vector(div_factor_freqhigh_c, 32),
            div_factor_freqlow  => conv_std_logic_vector(div_factor_freqlow_c, 32),
            pwm_out             => pwm_out
            );

end;
```

After we finished with the ***modulator_ip_rtl.vhd*** and ***modulator_ip_wrapper_rtl.vhd*** module creation, we should return to the Vivado IDE and do the following:

Step 55. Add ***modulator_ip_rtl.vhd***, ***modulator_ip_wrapper_rtl.vhd*** and ***modulator.xdc*** files in the "modulator_ip" project with Flow Navigator **Add Sources** option. We should also add ***modulator_pkg.vhd*** source file.

- ***modulator_ip_rtl.vhd***, ***modulator_ip_wrapper_rtl.vhd*** and ***modulator_pkg.vhd*** as Design Source file, and
- ***modulator.xdc*** as Constraints file

Step 56. Synthesize your design with **Run Synthesis** option from the **Flow Navigator** / **Synthesis** (see **Sub-chapter 6.5.2 Run Synthesis**)

Step 57. Implement your design with **Run Implementation** option from the **Flow Navigator** / **Implementation** (see **Sub--Chapter 10.2.2 Run Implementation**)

Step 58. Generate bitstream file with **Generate Bitstream** option from the **Flow Navigator** / **Program and Debug** (see **Sub-Chapter 10.3 Generate Bitstream File**)

Step 59. Program your ZedBoard device (see **Sub-Chapter 10.4 Program Device**)

Note: All the information about designing with IPs, like how to create and package an IP, how to add it to the IP Catalog, how to customize and generate packaged IP, you can also find in the **Lab 16: "Designing with IPs - IP Packager"** .

13.2 IP Integrator

To accelerate the creation of highly integrated and complex designs, Vivado Design Suite is delivered with IP Integrator (IPI) which provides a new graphical and Tcl-based IP- and system-centric design development flow.

Rapid development of smarter systems requires levels of automation that go beyond RTL-level design. The Vivado IP Integrator accelerates IP- and system-centric design implementation by providing the following:

- Seamless inclusion of IPI sub-systems into the overall design
- Rapid capture and packing of IPI designs for reuse
- Tcl scripting and graphical design
- Rapid simulation and cross-probing between multiple design views
- Support for processor or processor-less designs
- Integration of algorithmic and RTL-level IP
- Combination of DSP, video, analog, embedded, connectivity and logic
- Matches typical designer flows
- Easy to reuse complex sub-systems
- DRCs on complex interface level connections during design assembly
- Recognition and correction of common design errors
- Automatic IP parameter propagation to interconnected IP
- System-level optimizations

The Xilinx Vivado Design Suite IP Integrator feature lets you create complex system designs by instantiating and interconnecting IP cores from the Vivado IP Catalog onto a design canvas.

You can create designs interactively through the IP Integrator design canvas GUI, or using a Tcl programming interface. You will typically construct design at the AXI interface level for greater productivity, but you may also manipulate designs at the port level for more precise design control.

In this tutorial you will instantiate a few IPs in the IP Integrator tool and then stitch them up to create an IP sub-system design. While working on this tutorial, you will be introduced to the IP Integrator GUI, run design rule checks (DRC) on your design, and then integrate the design in a top-level design in the Vivado Design Suite. Finally, you will run synthesis and implementation process, generate bitstream file and run your design on the ZedBoard development board.

The following steps describe how to use the **IP Integrator** within your project:

Step 1. Close the existing **modulator_ip** project with the **File -> Close Project** option from the main Vivado IDE menu and in the Vivado **Getting Started** page choose **Create New Project** option

Step 2. In the **Create a New Vivado Project** dialog box, click **Next** to confirm the new project creation

Step 3. In the **Project Name** dialog box, enter a name of a new project and specify directory where the project data files will be stored. Name the project ***modulator_ipi***, verify the project location, ensure that **Create project subdirectory** is checked and click **Next**

Step 4. In the **Project Type** dialog box, verify that the **RTL Project** is selected and the **Do not specify sources at this time** option is unchecked and click **Next**

Step 5. In the **Add Sources** dialog box, ensure that the **Target language** is set to **VHDL** and click **Next**. You can add sources later, under the design canvas in the Vivado IP Integrator to create a subsystem design.

Step 6. In the **Add Existing IP (optional)** dialog box, click **Next**

Step 7. In the **Add Constraints (optional)** dialog box, click **Next**

Step 8. In the **Default Part** dialog box, ensure that the **ZedBoard Zynq Evaluation and Development Kit** is selected and click **Next**

Step 9. In the **New Project Summary** dialog box, review the project summary and click **Finish** if you are satisfied with the summary of your project or go back as much as necessary to correct all the questionable issues

The new project, ***modulator_ipi***, will be automatically opened in the Vivado IDE.

Step 10. In the **Flow Navigator**, expand **IP Integrator** and select **Create Block Design** command, see Illustration 13.38

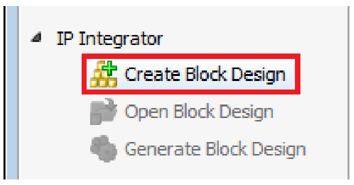
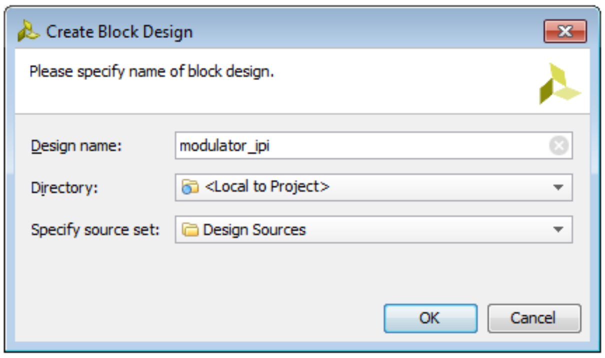


Figure 13.38: Create Block Design option

Step 11. In the **Create Block Design** dialog box, specify ***modulator_ipi*** name of the block design in the **Design name** field and click **OK**, see Illustration 13.39



Please specify name of block design.

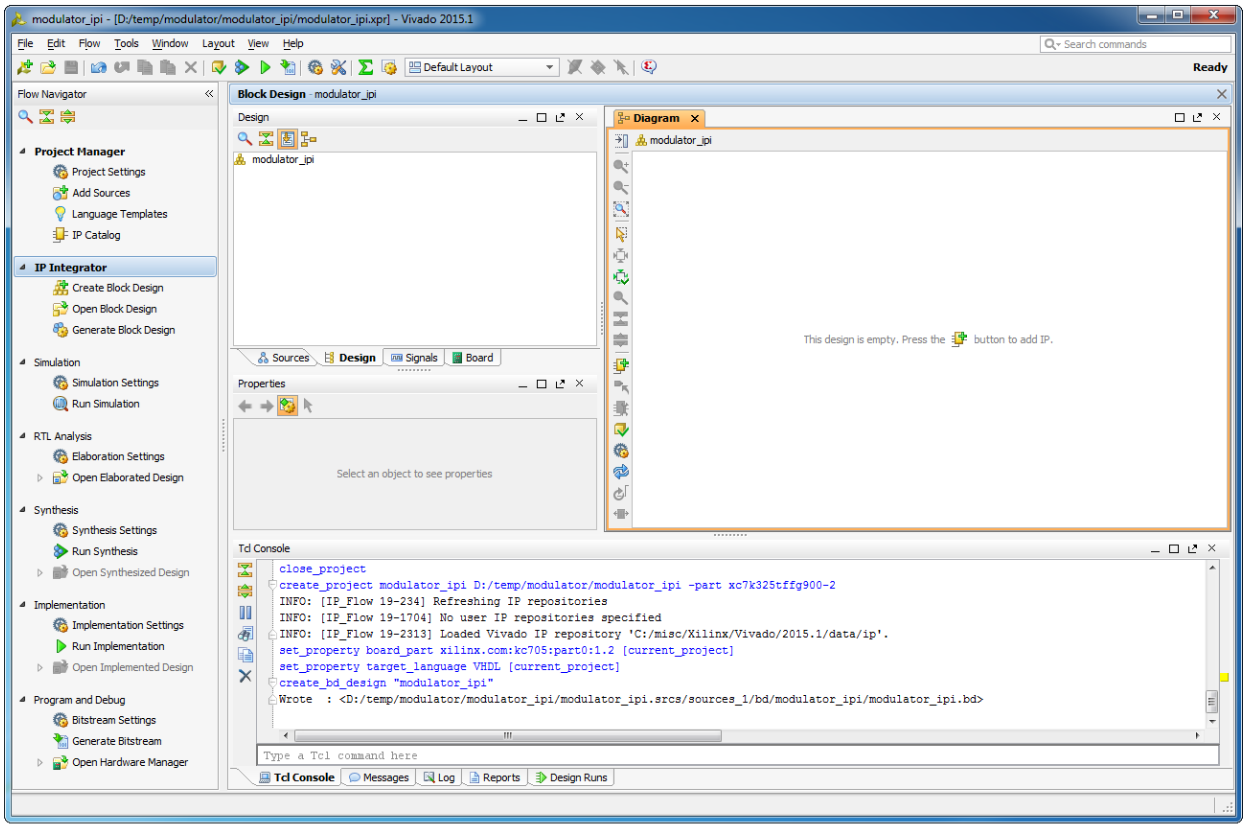
Design name: modulator_ipi

Directory: <Local to Project>

Specify source set: Design Sources

Figure 13.39: Create Block Design dialog box

The Vivado IDE will display a blank design canvas. You can quickly create complex subsystem by integrating IP cores in it, see Illustration 13.40



```
close_project
create_project modulator_ipi D:/temp/modulator/modulator_ipi -part xc7k325tffg900-2
INFO: [IP_Flow 19-234] Refreshing IP repositories
INFO: [IP_Flow 19-1704] No user IP repositories specified
INFO: [IP_Flow 19-2313] Loaded Vivado IP repository 'C:/misc/Xilinx/Vivado/2015.1/data/ip'.
set_property board_part xilinx.com:kc705:part0:1.2 [current_project]
set_property target_language VHDL [current_project]
create_bd_design "modulator_ipi"
Wrote  : <D:/temp/modulator/modulator_ipi/modulator_ipi.srcs/sources_1/bd/modulator_ipi/modulator_ipi.bd>
```

Figure 13.40: Vivado IDE with a blank design canvas

Step 12. To add our previously packaged IPs (***frequency_trigger_v1_0***, ***counter_v1_0***, ***sine_v1_0*** and ***pwm_v1_0***) to the IP Catalog, please repeat the steps **32 - 38** from the **Sub-chapter 13.1 IP Packager**.

Step 13. The ***modulator_ipi*** design is empty. To get started, add IPs from the IP Catalog. You can do that on three ways:

- In the design canvas, right-click and choose **Add IP...** option, see Illustration 13.41, or

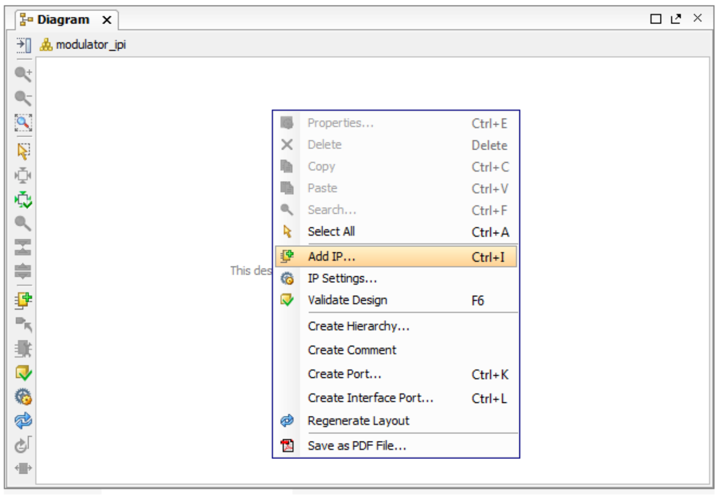


Figure 13.41: Add IP option

- Use the **Add IP** link in the IP Integrator canvas, see Illustration 13.42, or

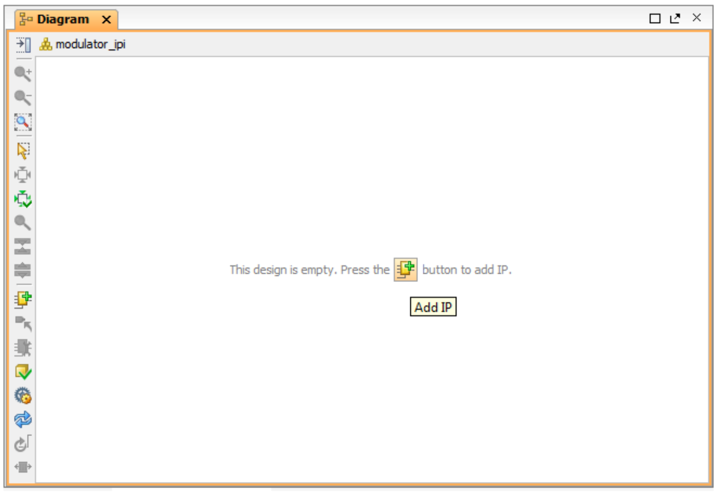


Figure 13.42: Add IP link

- Click on the **Add IP** button in the IP Integrator sidebar menu, see Illustration 13.43

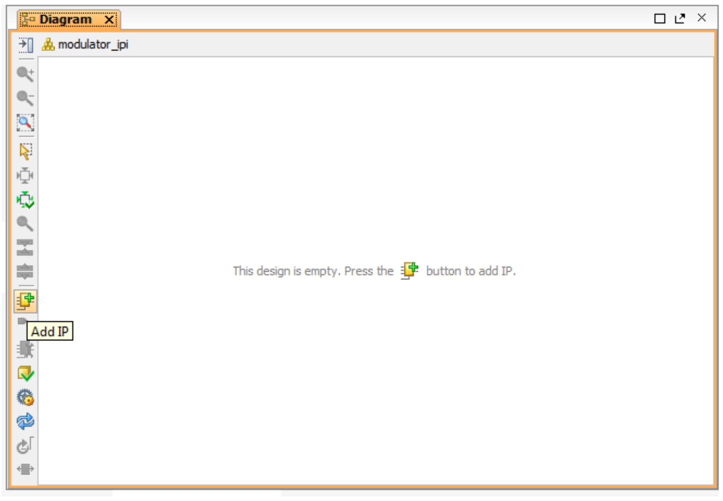


Figure 13.43: Add IP button

Step 14. In the **IP Catalog**, search for the ***frequency_trigger_v1_0*** core, see Illustration 13.44

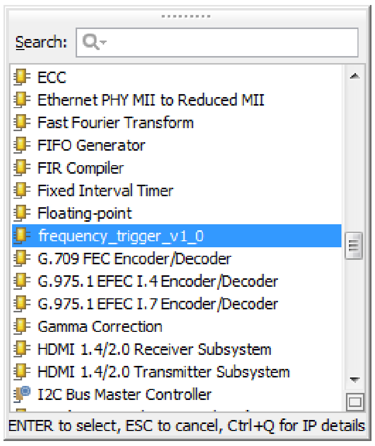

Search:
ECC
Ethernet PHY MII to Reduced MII
Fast Fourier Transform
FIFO Generator
FIR Compiler
Fixed Interval Timer
Floating-point
frequency_trigger_v1_0
G.709 FEC Encoder/Decoder
G.975.1 EFEC I.4 Encoder/Decoder
G.975.1 EFEC I.7 Encoder/Decoder
Gamma Correction
HDMI 1.4/2.0 Receiver Subsystem
HDMI 1.4/2.0 Transmitter Subsystem
I2C Bus Master Controller
ENTER to select, ESC to cancel, Ctrl+Q for IP details

Figure 13.44: frequency_trigger_v1_0 core in the IP Catalog

Step 15. When you find it, press enter on the keyboard or simply double- click on the ***frequency_trigger_v1_0*** core in the IP Catalog and the selected core will be automatically instantiated into the IP Integrator design canvas, see Illustration 13.45

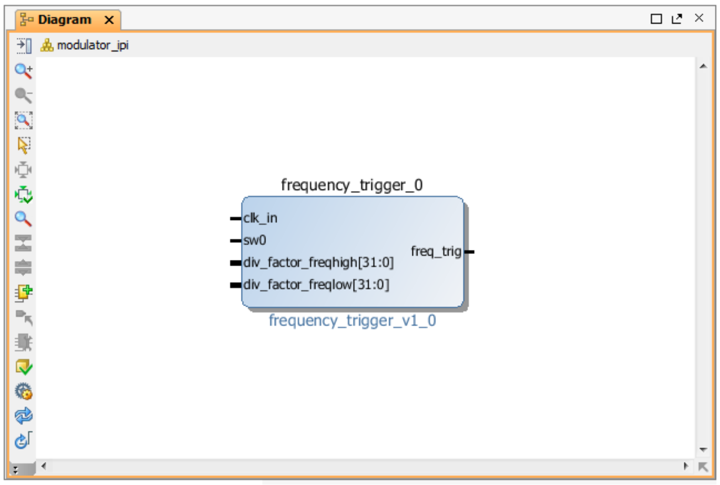


Figure 13.45: Automatically instantiated frequency_trigger_v1_0 core in the IP Integrator design canvas

Step 16. Right-click in the IP integrator canvas and select the **Add IP...** option to add the rest of the necessary IPs (***counter_v1_0***, ***sine_v1_0*** and ***pwm_v1_0***). At this point, the IP Integrator canvas should look like as it is shown on the Illustration 13.46

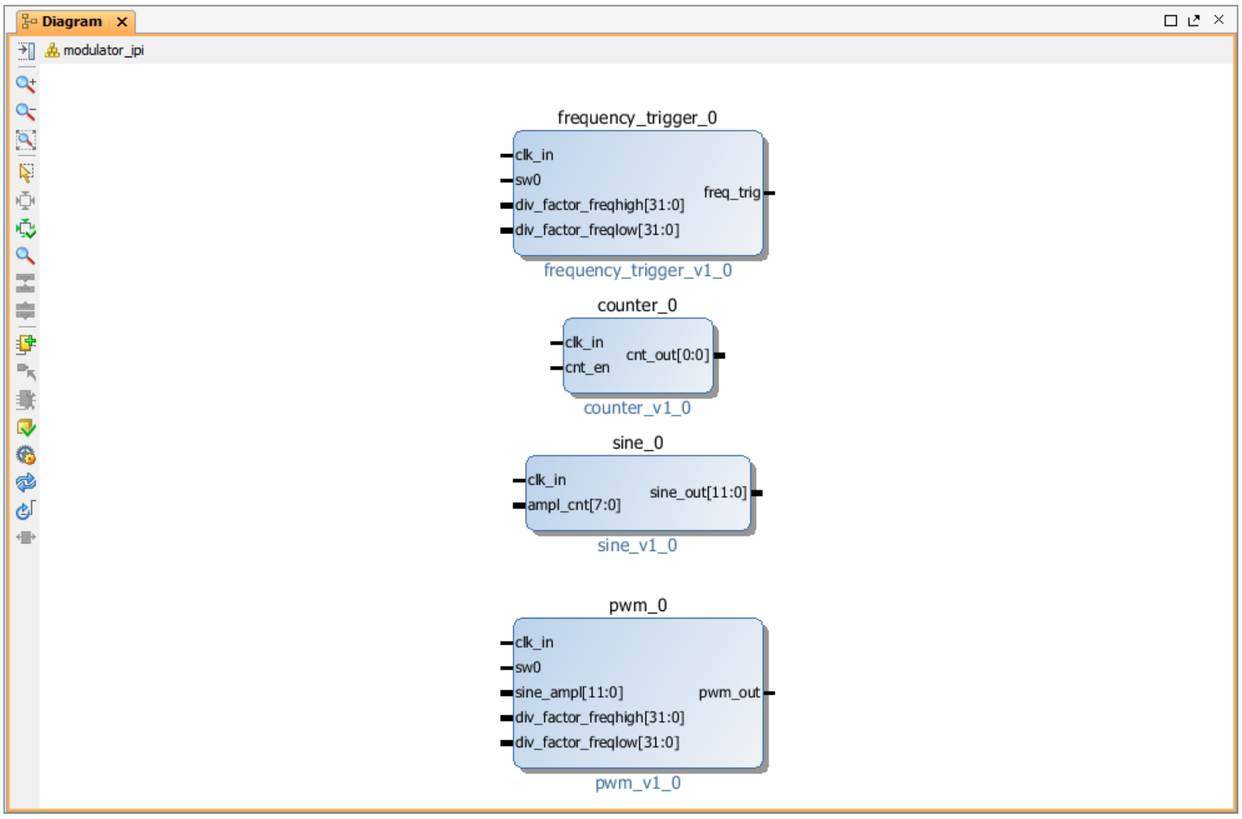
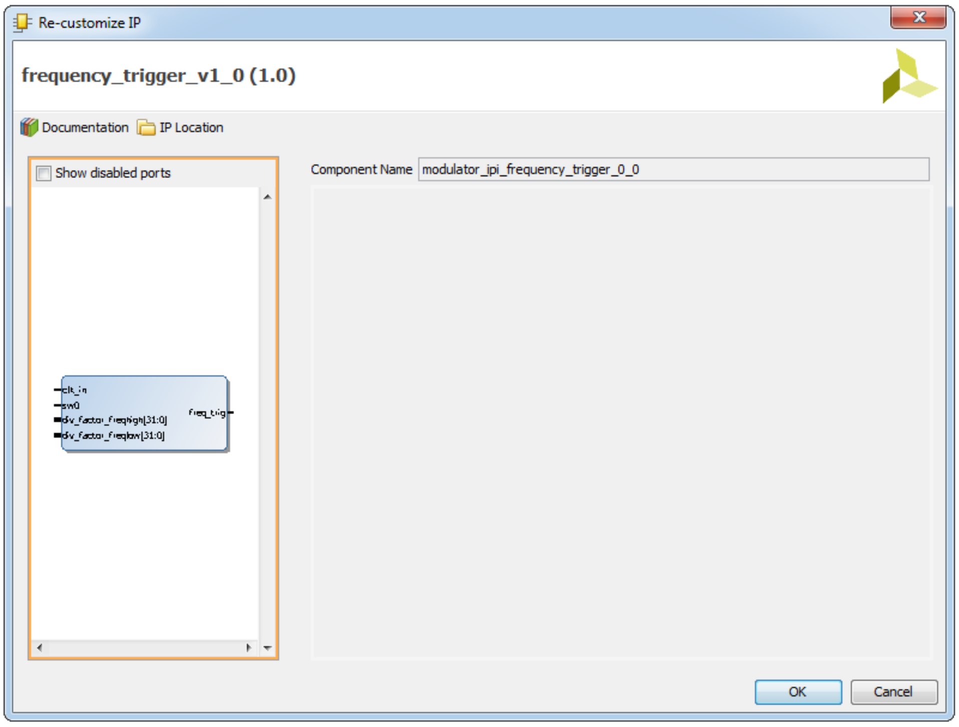


Figure 13.46: IP Integrator design canvas with all four instantiated IPs

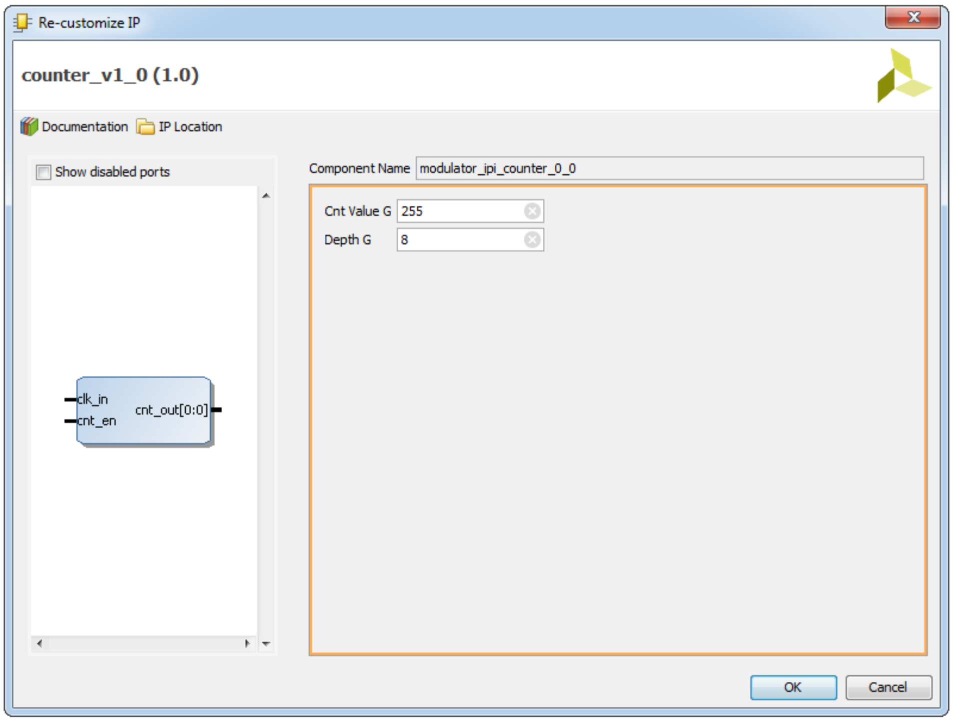
Step 17. Double-click on the each of the IP cores to re-customize it. Re- customize IPs on the same way as it is done in the previous **Sub-chapter 13.1 IP Packager** (steps: **41**, **44**, **48** and **52**), see Illustrations 13.47, 13.48, 13.49 and 13.50



frequency_trigger_v1_0 (1.0)

Component Name: modulator_ipi_frequency_trigger_0_0

Figure 13.47: frequency_trigger_v1_0 re-customization dialog box



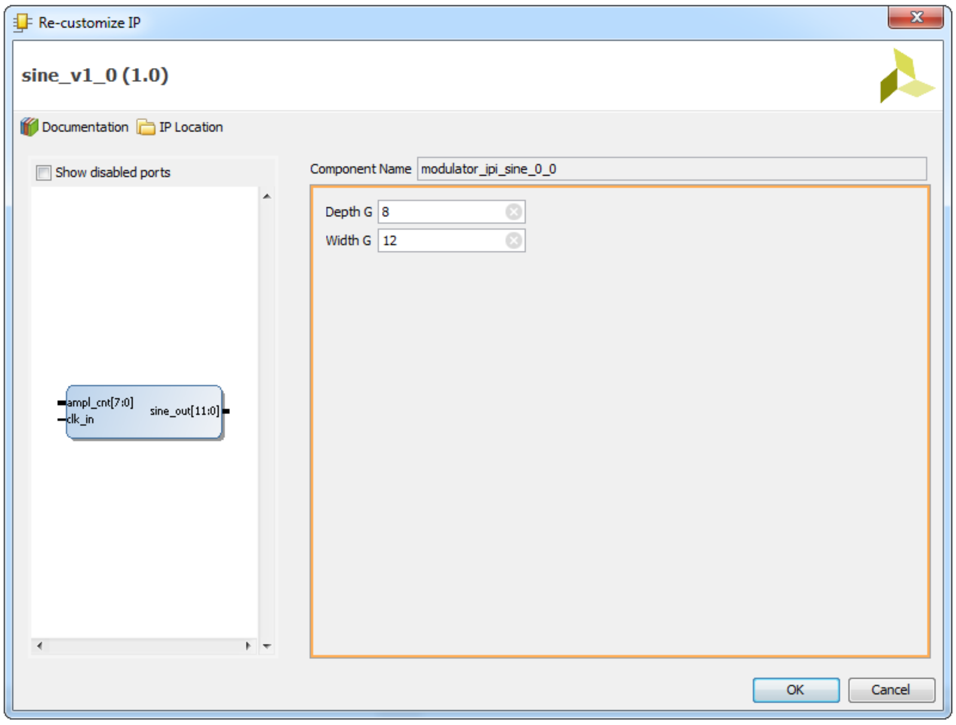
counter_v1_0 (1.0)

Component Name: modulator_ipi_counter_0_0

Cnt Value G: 255

Depth G: 8

Figure 13.48: counter_v1_0 re-customization dialog box


Re-customize IP

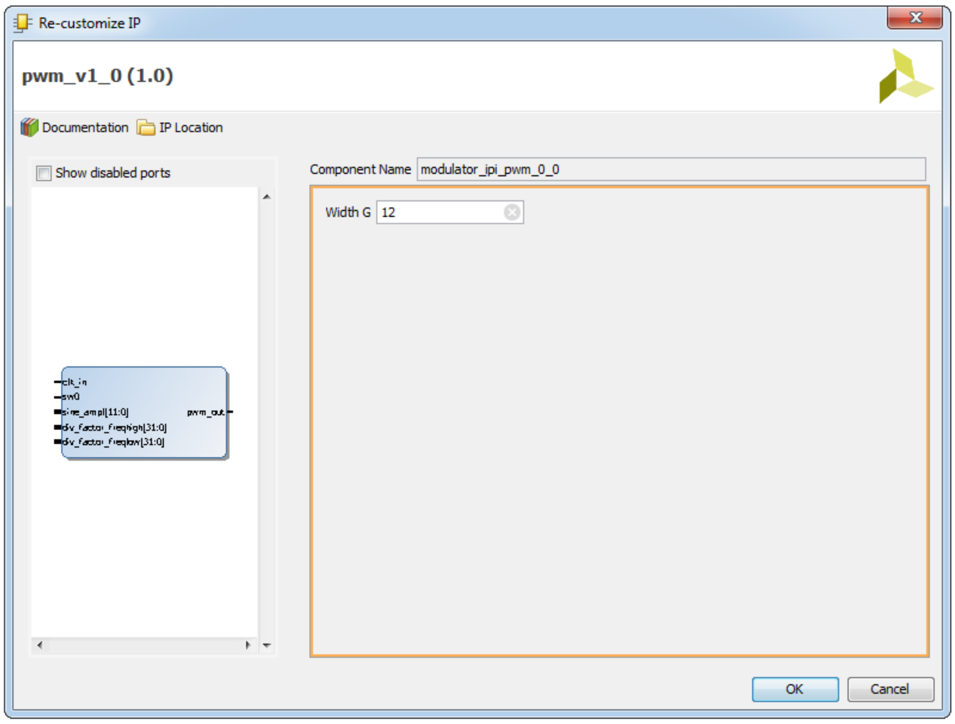
sine_v1_0 (1.0)

Component Name modulator_ipi_sine_0_0

Depth G 8

Width G 12

Figure 13.49: sine_v1_0 re-customization dialog box


Re-customize IP

pwm_v1_0 (1.0)

Component Name modulator_ipi_pwm_0_0

Width G 12

Figure 13.50: pwm_v1_0 re-customization dialog box

Step 18. After we re-customize all four IPs, the IP Integrator canvas should look like as it is shown on the Illustration 13.51

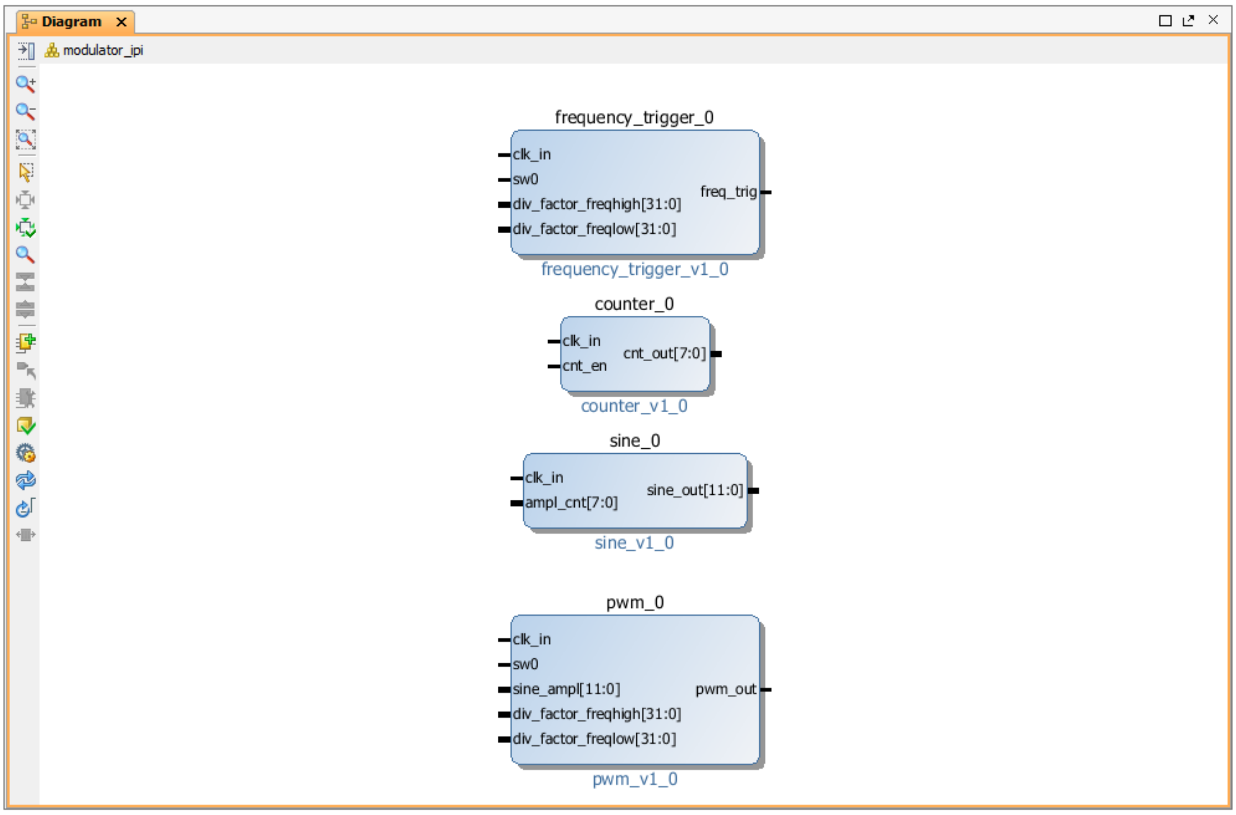


Figure 13.51: IP Integrator design canvas with all four re-customized IPs

Step 19. The last IP necessary for our design is the **Constant** IP. Add **Constant** IP four times into the block design. Two **Constant** IP instances will be connected to the ***div_factor_freqhigh(31:0)*** and ***div_factor_freqlow(31:0)*** ports of the **frequency_trigger_v1_0** module and remaining two instances to the ***div_factor_freqhigh(31:0)*** and ***div_factor_-freqlow(31:0)*** ports of the **pwm_v1_0** module, see Illustration 13.52.

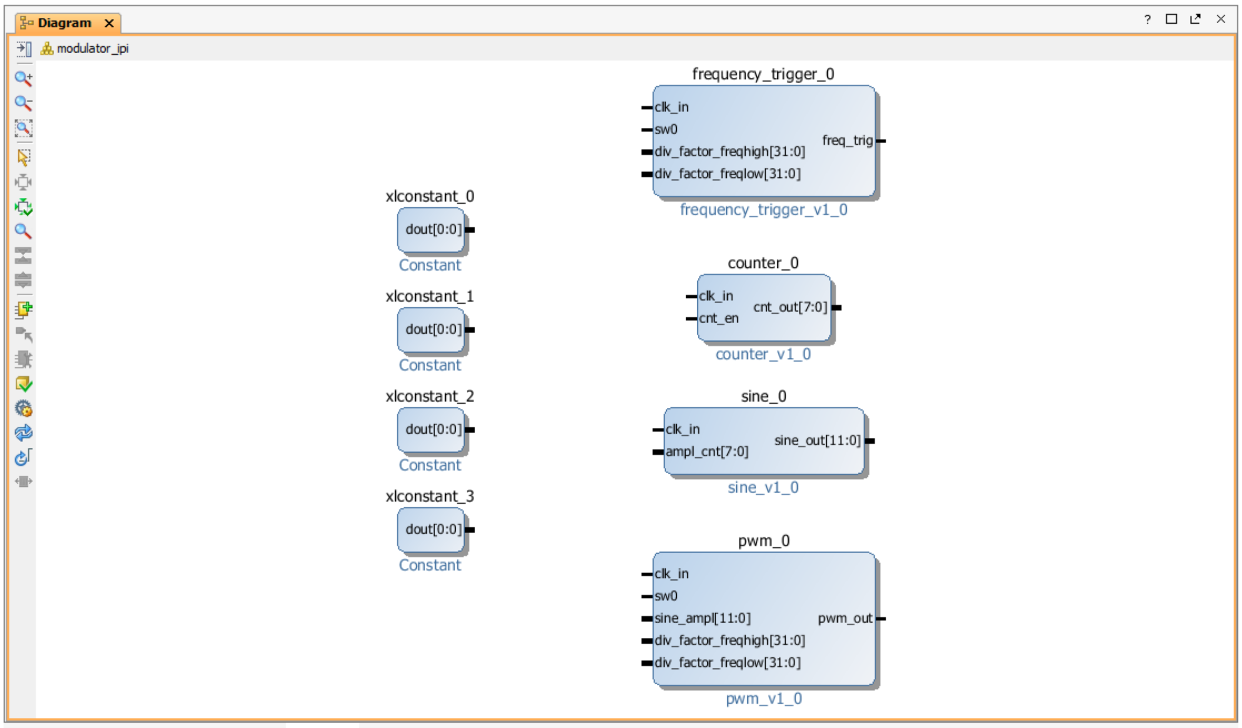
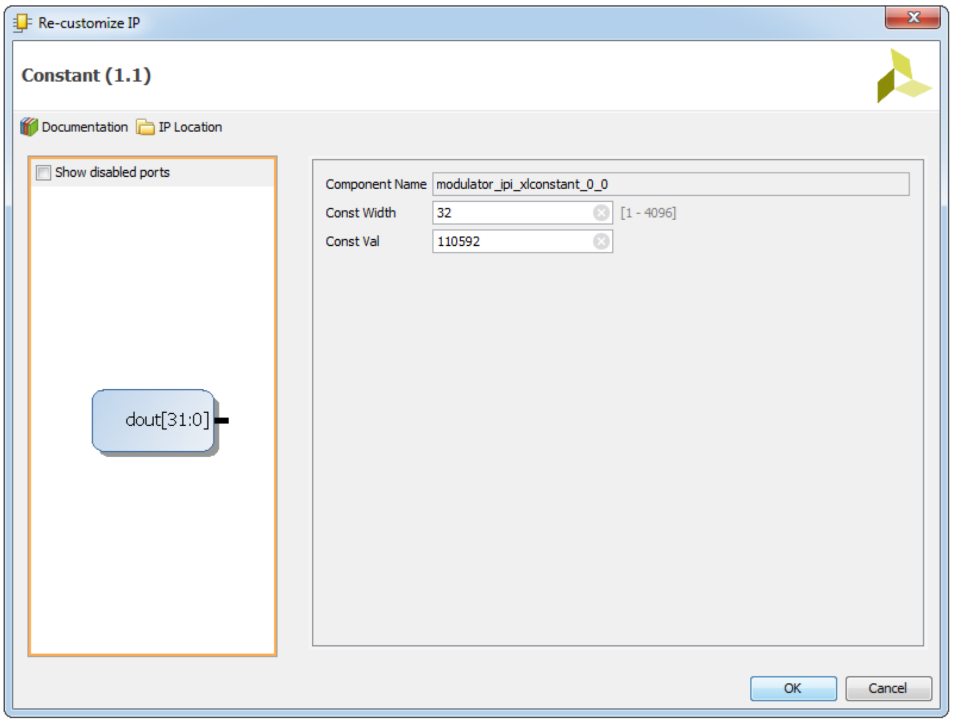


Figure 13.52: IP Integrator design canvas with instantiated Constant IPs

Step 20. Double-click on the first **Constant (xlconstant_0)** block and set the **Const Width** value to **32** and **Const Value** value to **110592**, see Illustration 13.53

- **Const Width** to **32** - because ***div_factor_freqhigh*** port that we would like to connect to is 32-bit wide

- **Const Value** to **110592** - because 110592 is the number that divides frequency of the input clock signal (100 MHz) to the required frequency, see Table 1.2


Re-customize IP

Constant (1.1)

Documentation | IP Location

Show disabled ports

Component Name: modulator_ipi_xlconstant_0_0
Const Width: 32 [1 - 4096]
Const Val: 110592

Figure 13.53: Constant block re-customization dialog box

Step 21. Do the same procedure with the second **Constant (xlconstant_1)** IP block. Set the **Const Width** value to **32** and **Const Value** value to **389120**

Step 22. In the third **Constant (xlconstant_2)** IP block, set the **Const Width** value to **32** and **Const Value** value to **27**

- **Const Value** to **27** (110592/4096=27) - because PWM module must operate at 2^{width} $(2^{12} = 4096)$ higher frequency then the Sine module. This is required in order to generate correct pwm signal, as described earlier

Step 23. In the forth **Constant (xlconstant_3)** IP block, set the **Const Width** value to **32** and **Const Value** value to **95** (389120/4096=95)

After we added all necessary IPs for our design, the next step will be to connect IPs between themselves. Make connections on the same way as it is shown on the . Here are the steps how to make these connections:

Step 24. First step will be to create new ports:

- Select **clk_in** pin, right-click on it and select **Create port...** option, see Illustration 13.54

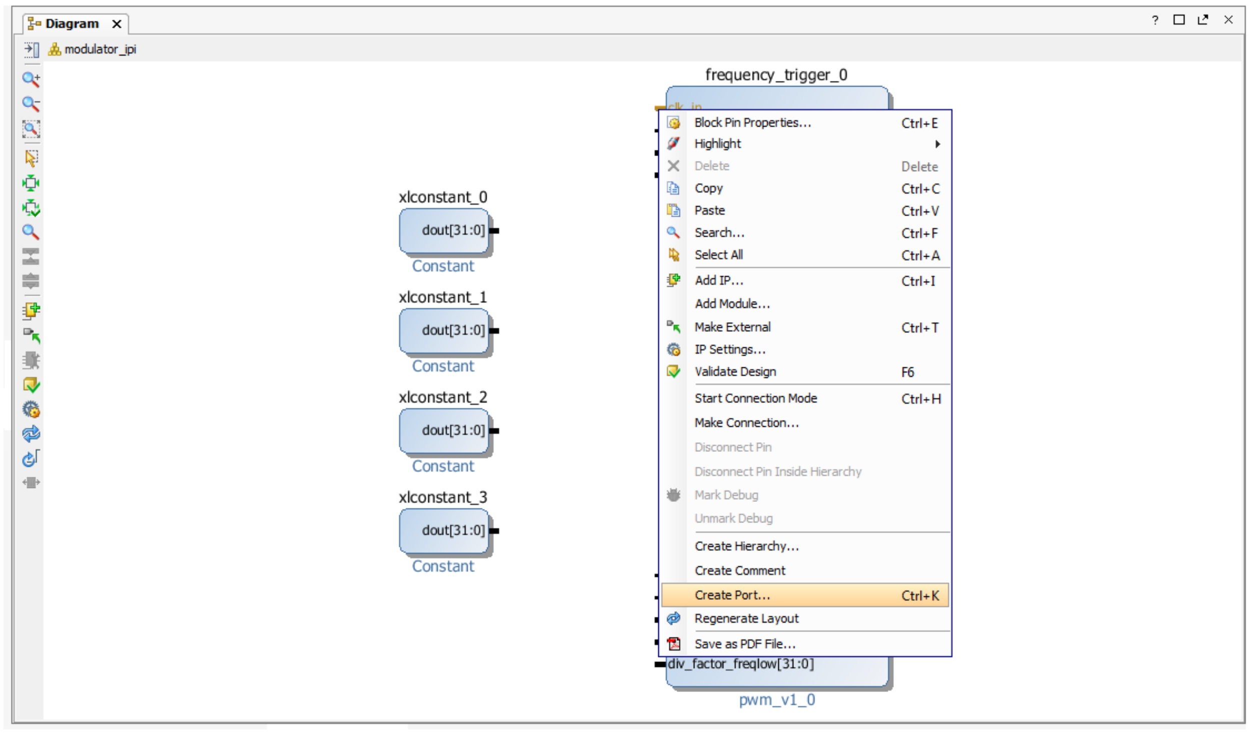
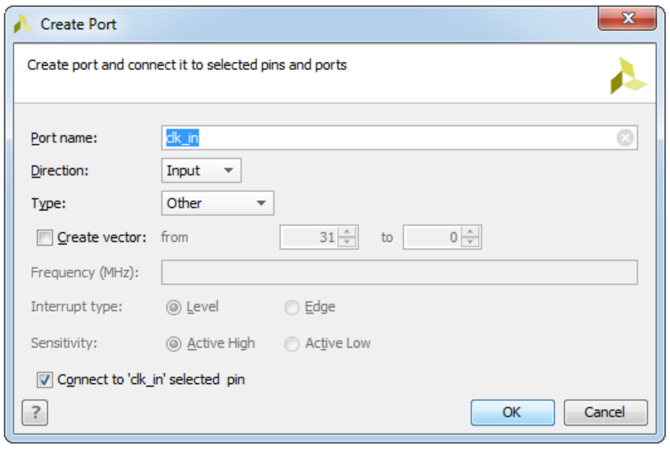


Figure 13.54: Create Port option

- In the **Create Port** dialog box, check is the port name **clk_in** in the **Port name** field, leave all other parameters unchanged and click **OK**, see Illustration 13.55


Create Port

Create port and connect it to selected pins and ports

Port name: clk_in

Direction: Input

Type: Other

Create vector: from 31 to 0

Frequency (MHz):

Interrupt type: Level Edge

Sensitivity: Active High Active Low

Connect to 'clk_in' selected pin

Figure 13.55: Create Port dialog box

- Repeat the same procedure for **sw0** and **pwm_out** pins. After these modifications, the IP Integrator design canvas should look like as it is shown on the Illustration 13.56

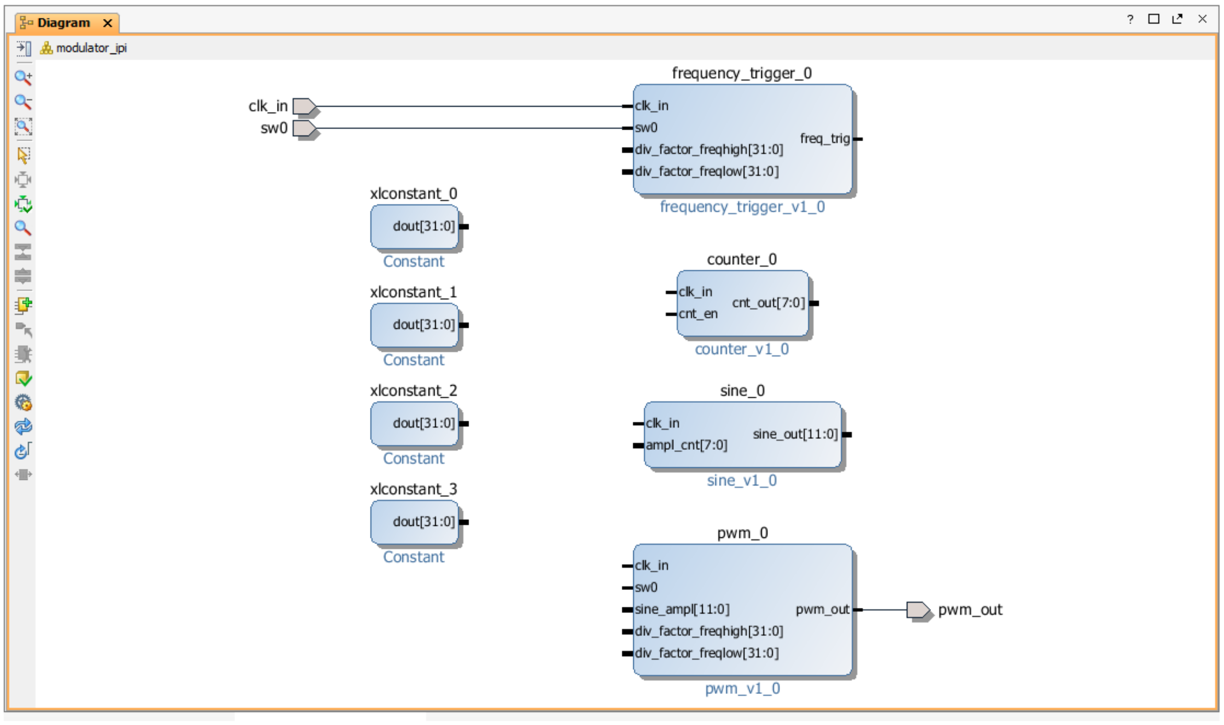


Figure 13.56: IP Integrator design canvas with new ports

Step 25. Next step will be to connect the IPs:

Place the cursor on top of the desired pin and you can notice that the cursor changes into a pencil indicating that a connection can be made from that pin. Clicking the left mouse button a connection starts. Click and drag the cursor from one pin to another. You must press and hold down the left mouse button while dragging the connection from one pin to another. As you drag the connection wire, a green checkmark appears on the port indicating that a valid connection can be made between these points. The Vivado IP Integrator highlights all possible connections points in the subsystem design as you interactively wire the pins and ports. Release the left mouse button and Vivado IP integrator makes connection between desired ports. Repeat this procedure until all the pins become associated, see Illustration 13.57

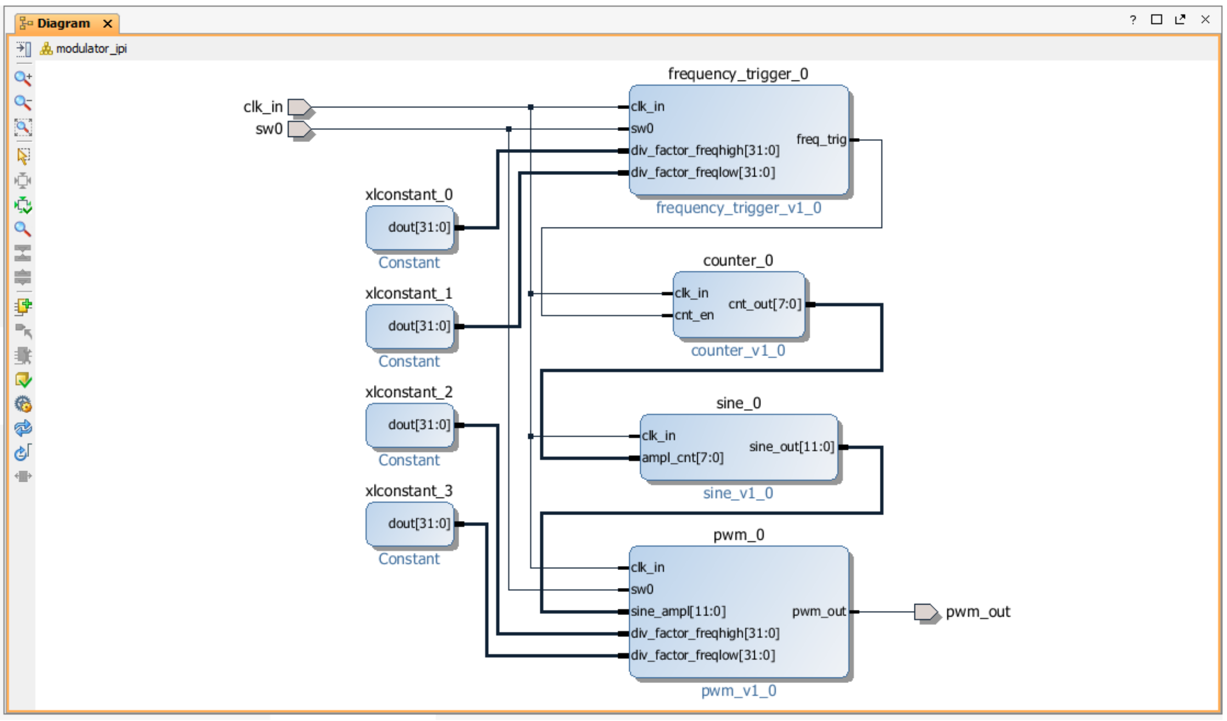


Figure 13.57: IP Integrator design canvas with connected IPs

Step 26. From the sidebar menu of the design canvas, run the IP subsystem design rule checks by clicking the **Validate Design** button

Alternatively, you can do the same by selecting **Tools -**> **Validate Design** from the main menu, see Illustration 13.58, or

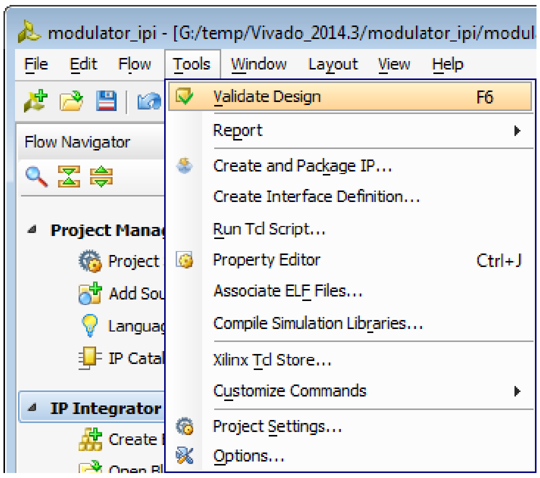


Figure 13.58: Validate Design option from the main menu

by clicking the design canvas and selecting **Validate Design** button from the main toolbar menu, see Illustration 13.59

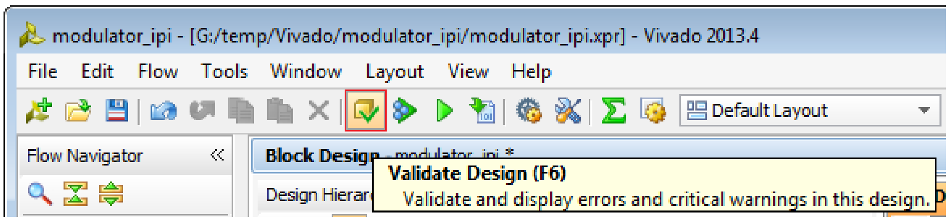
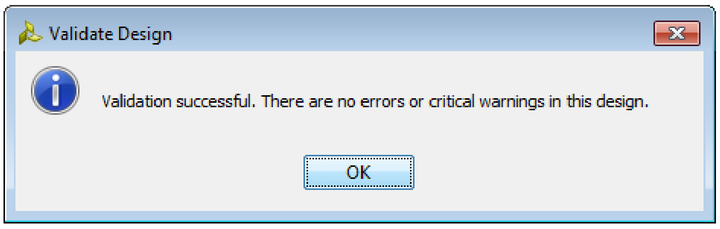

modulator_ipi - [G:/temp/Vivado/modulator_ipi/modulator_ipi.xpr] - Vivado 2013.4
File Edit Flow Tools Window Layout View Help
Default Layout
Flow Navigator
Block Design - modulator_ipi *
Design Hierar
Validate Design (F6)
Validate and display errors and critical warnings in this design.

Figure 13.59: Validate Design button from the main toolbar menu

Step 27. In the **Validate Design** dialog box, click **OK**, see Illustration 13.60



Validation successful. There are no errors or critical warnings in this design.

Figure 13.60: Validate Design dialog box

Step 28. At this point, you should save the IP integrator design. Use the **File ->** **Save Block Design** command from the main menu to save the design.

Step 29. In the **Sources** window, select **modulator_ipi**, right- click on it and choose **Create HDL Wrapper...** option, see Illustration 13.61

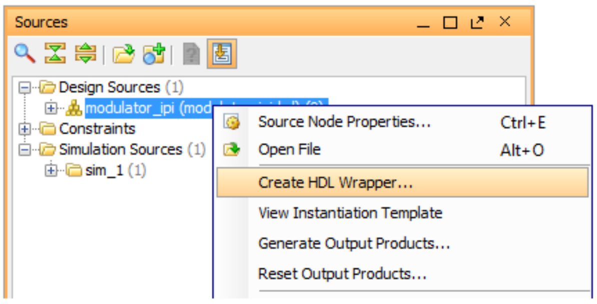
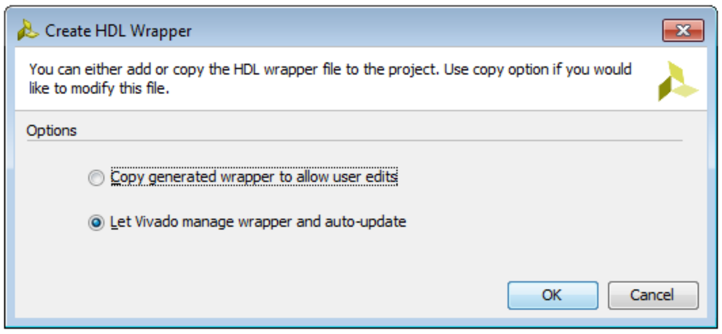


Figure 13.61: Create HDL Wrapper option

Step 30. In the **Create HDL Wrapper** dialog box, select **Let Vivado manage wrapper and auto-update** option and click **OK**, see Illustration 13.62



You can either add or copy the HDL wrapper file to the project. Use copy option if you would like to modify this file.

○ Copy generated wrapper to allow user edits

● Let Vivado manage wrapper and auto-update

Figure 13.62: Create HDL Wrapper dialog box

Step 31. After the HDL wrapper is generated, you should see it in the **Sources** window, see Illustration 13.63

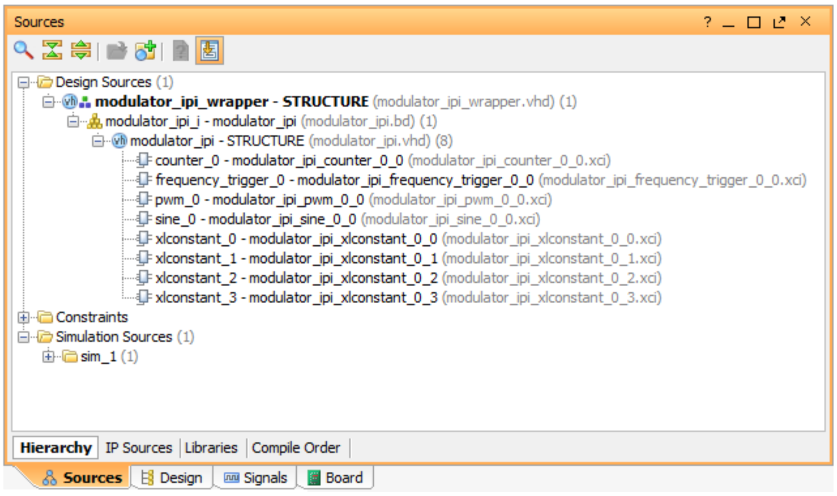

Sources
Design Sources (1)
modulator_ipi_wrapper - STRUCTURE (modulator_ipi_wrapper.vhd) (1)
modulator_ipi_i - modulator_ipi (modulator_ipi.bd) (1)
modulator_ipi - STRUCTURE (modulator_ipi.vhd) (8)
counter_0 - modulator_ipi_counter_0_0 (modulator_ipi_counter_0_0.xci)
frequency_trigger_0 - modulator_ipi_frequency_trigger_0_0 (modulator_ipi_frequency_trigger_0_0.xci)
pwm_0 - modulator_ipi_pwm_0_0 (modulator_ipi_pwm_0_0.xci)
sine_0 - modulator_ipi_sine_0_0 (modulator_ipi_sine_0_0.xci)
xlconstant_0 - modulator_ipi_xlconstant_0_0 (modulator_ipi_xlconstant_0_0.xci)
xlconstant_1 - modulator_ipi_xlconstant_0_1 (modulator_ipi_xlconstant_0_1.xci)
xlconstant_2 - modulator_ipi_xlconstant_0_2 (modulator_ipi_xlconstant_0_2.xci)
xlconstant_3 - modulator_ipi_xlconstant_0_3 (modulator_ipi_xlconstant_0_3.xci)
Constraints
Simulation Sources (1)
sim_1 (1)

Figure 13.63: Sources window with generated modulator_ipi HDL wrapper

Step 32. The last step in our design will be to crate and add ***modulator_ipi_rtl.xdc*** constraints file. The content of the ***modulator_ipi_rtl.xdc*** constraints file is shown in the text below:

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

set_property IOSTANDARD LVCMOS33 [get_ports clk_in];
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]
```

Step 33. Add ***modulator_ipi_rtl.xdc*** file in the ***modulator_ipi*** project as constraints file, see Illustration 13.64

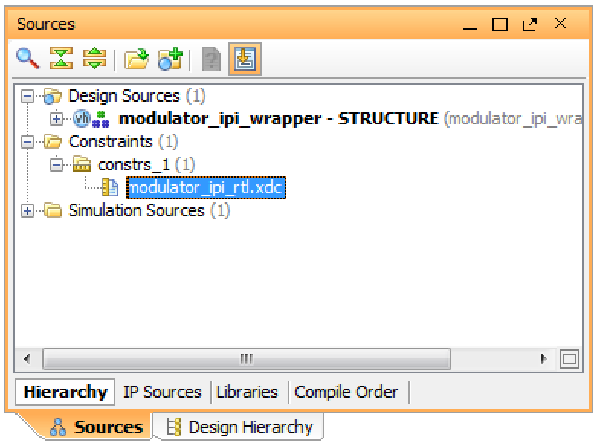

Sources
Design Sources (1)
modulator_ipi_wrapper - STRUCTURE (modulator_ipi_wra
Constraints (1)
constrs_1 (1)
modulator_ipi_rtl.xdc
Simulation Sources (1)

Figure 13.64: Sources window with added modulator_ipi_rtl.xdc constraints file

Step 34. Synthesize your design with **Run Synthesis** option from the **Flow Navigator** / **Synthesis** (see **Sub-chapter 6.5.2 Run Synthesis**)

Step 35. Implement your design with **Run Implementation** option from the **Flow Navigator** / **Implementation** (see **Sub--Chapter 10.2.2 Run Implementation**)

Step 36. Generate bitstream file with **Generate Bitstream** option from the **Flow Navigator** / **Program and Debug** (see **Sub-Chapter 10.3 Generate Bitstream File**)

4***Step 37***. Program your ZedBoard device (see **Sub-Chapter 10.4 Program Device**)

Note: All the information about how to design with IPs using Vivado IP Integrator tool, how to create complex system design by instantiating and interconnecting IP cores from the Vivado IP Catalog onto a design canvas, you can also find in the **Lab 17: "Designing with IPs - IP Integrator"** .

13.3 Debugging IP Integrated Designs

In-system debugging allows you to debug your design in real-time on your target hardware. IP Integrator provides ways to instrument your design for debugging, which will be explained in this sub-chapter. In the earlier sub-chapters we have explained that Vivado IDE has two different flows for debugging. One is the **HDL Instantiation Debug Probing Flow** and the other one is **Using the Netlist Insertion Debug Probing Flow**. Choosing the flow depends on your preferences and types of nets/signals that you are interested in debugging. In this tutorial we will explain both flows on the same, Modulator IP integrated design.

Details about how to debug your IP Integrator design using the "HDL Instantiation Debug Probing Flow" can be found in the **Chapter 14 "Appendix"**.

Using the Netlist Insertion Flow in IP Integrator

As shown in the **Sub-chapter 11.1 Inserting ILA and VIO Cores into Design**, in this flow you will mark nets that you are interested in analyzing in the block design. Marking nets for debug in the block design offers more control in terms of identifying debug signals during coding and enabling/disabling debugging later in the flow.

To start debugging process using the Netlist Insertion Flow in IP Integrator tool, please do the following:

Step 1. Right-click on the ***modulator_ipi*** block design canvas and select **Add IP...** option

Step 2. In the IP Catalog, search for VIO core, select it and double- click on it to instantiate the VIO core in the IP Integrator canvas

Step 3. In case of VIO core, use default configuration settings

Step 4. Remove **sw0** port from the IP Integrated canvas and connect the VIO core with the rest of the IPs in the same way as it is shown on the **Figure 11.9**, see Illustration 13.65

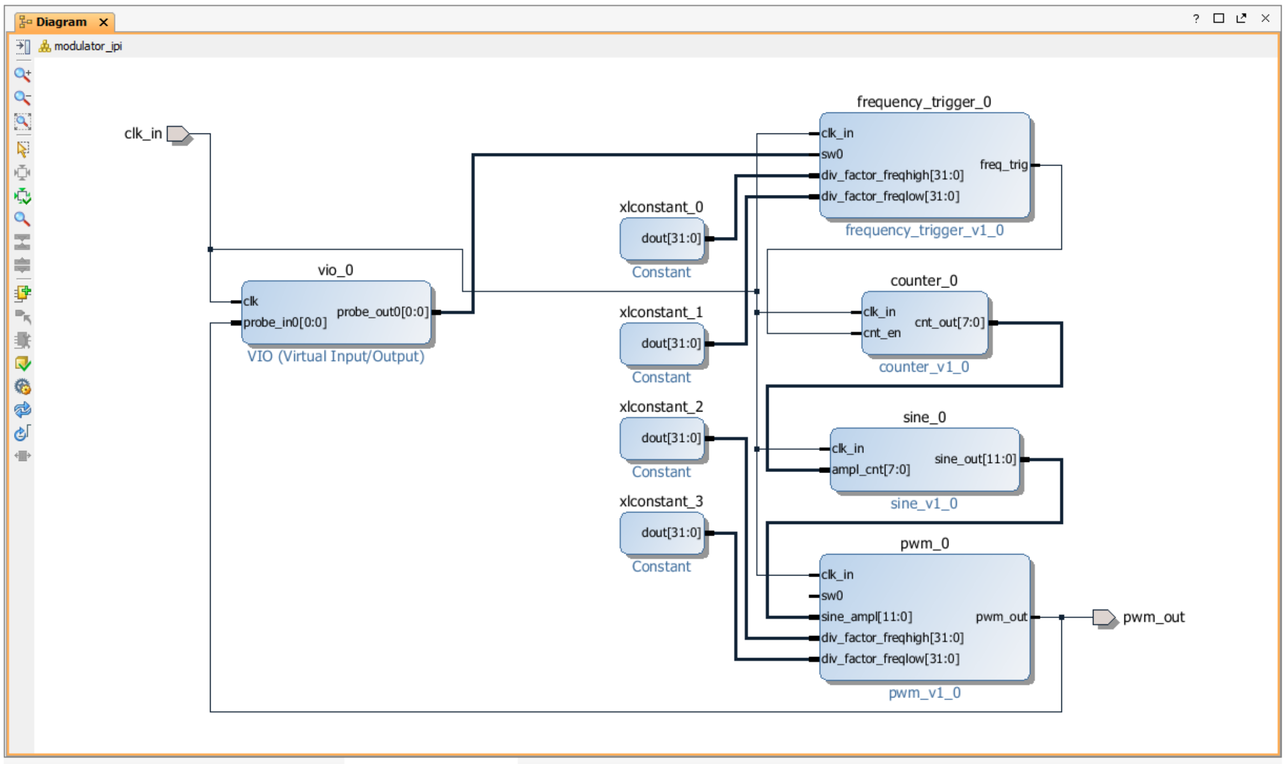


Figure 13.65: IP Integrator design canvas with connected VIO core

Step 5. The next step will be to mark nets for debug

Nets can be marked for debug in the block design by highlighting them, right-clicking and selecting **Debug**, see Illustration 13.66

Step 6. Mark ***sine_ampl_s*** and ***freq_trig_s*** nets for debug

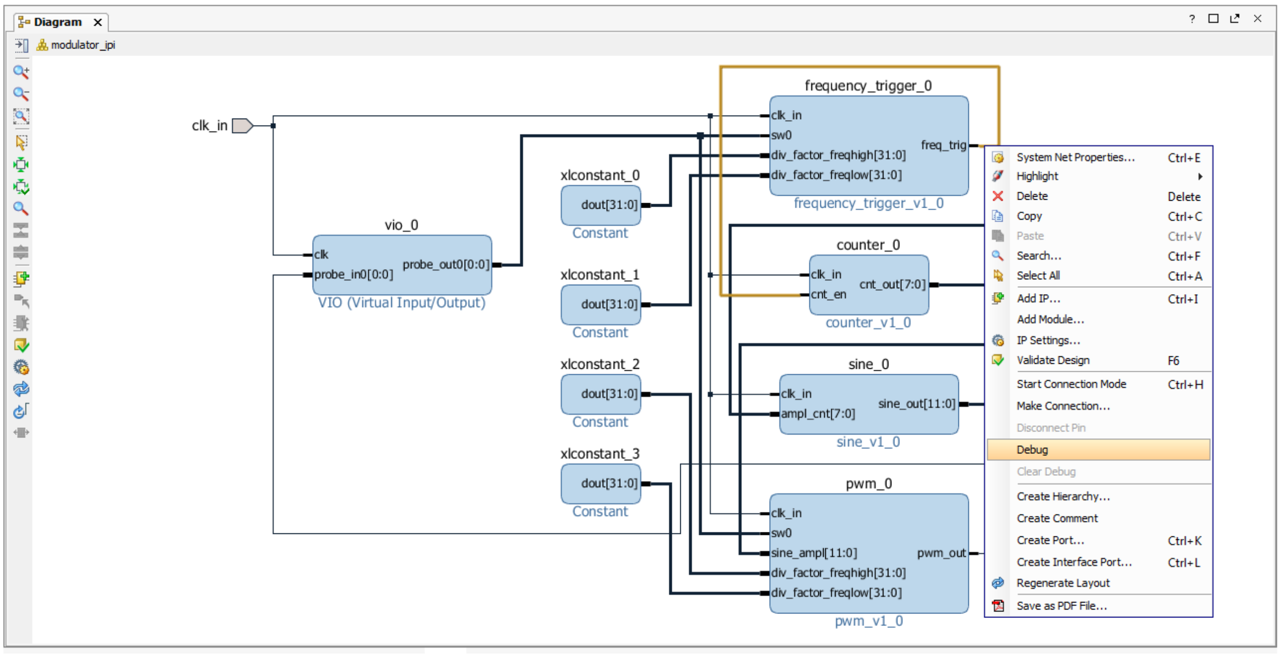


Figure 13.66: Debug option

The nets that have been marked for debug will show a small bug icon placed on top of the net in the block design. Likewise, a bug icon can be seen placed on the nets to be debugged in the Design Hierarchy window as well.

Step 7. Generate output products by clicking on the **Generate Block Design** command or by highlighting the block design in the sources window, right-clicking and selecting **Generate Output Products** option, see Illustration 13.67

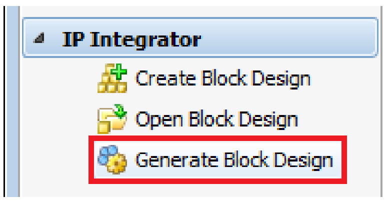
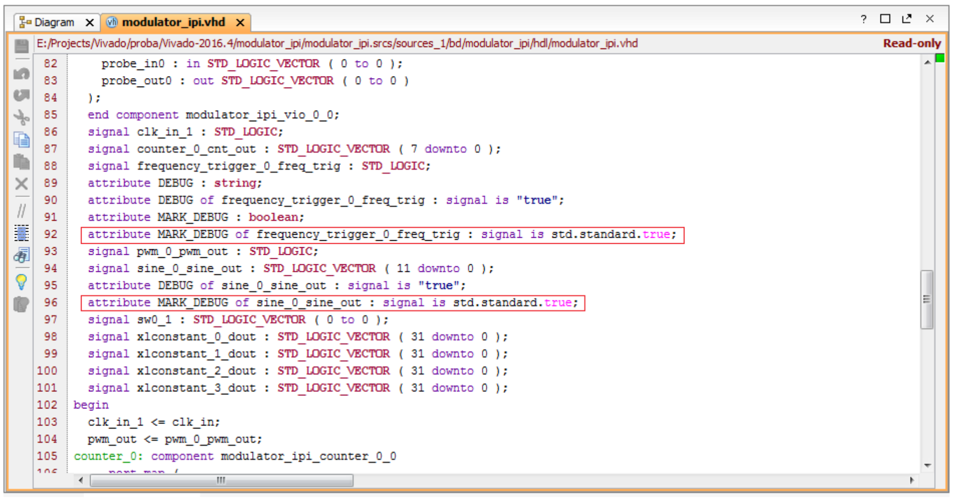


Figure 13.67: Generate Block Design command

Note: Generate outputs needed for synthesis, simulation and implementation processes.

Step 8. In the **Generate Output Products** dialog box, click **Generate**

Step 9. Marking the nets for debug places the **MARK_DEBUG attribute** on the net which can be seen in the generated top-level HDL file. This is important because prevents the Vivado tools from optimizing and renaming the nets, see Illustration 13.68



E:/Projects/Vivado/proba/Vivado-2016.4/modulator_ipi/modulator_ipi.srcs/sources_1/bd/modulator_ipi/hdl/modulator_ipi.vhd Read-only

```
    probe_in0 : in STD_LOGIC_VECTOR ( 0 to 0 );
    probe_out0 : out STD_LOGIC_VECTOR ( 0 to 0 )
  );
  end component modulator_ipi_vio_0_0;
  signal clk_in_1 : STD_LOGIC;
  signal counter_0_cnt_out : STD_LOGIC_VECTOR ( 7 downto 0 );
  signal frequency_trigger_0_freq_trig : STD_LOGIC;
  attribute DEBUG : string;
  attribute DEBUG of frequency_trigger_0_freq_trig : signal is "true";
  attribute MARK_DEBUG : boolean;
  attribute MARK_DEBUG of frequency_trigger_0_freq_trig : signal is std.standard.true;
  signal pwm_0_pwm_out : STD_LOGIC;
  signal sine_0_sine_out : STD_LOGIC_VECTOR ( 11 downto 0 );
  attribute DEBUG of sine_0_sine_out : signal is "true";
  attribute MARK_DEBUG of sine_0_sine_out : signal is std.standard.true;
  signal sw0_1 : STD_LOGIC_VECTOR ( 0 to 0 );
  signal xlconstant_0_dout : STD_LOGIC_VECTOR ( 31 downto 0 );
  signal xlconstant_1_dout : STD_LOGIC_VECTOR ( 31 downto 0 );
  signal xlconstant_2_dout : STD_LOGIC_VECTOR ( 31 downto 0 );
  signal xlconstant_3_dout : STD_LOGIC_VECTOR ( 31 downto 0 );
begin
  clk_in_1 <= clk_in;
  pwm_out <= pwm_0_pwm_out;
counter_0: component modulator_ipi_counter_0_0
```

Figure 13.68: MARK_DEBUG attributes in the generated HDL file

Step 10. Remove ***modulator_ipi_rtl.xdc*** constraints file from the design and add new ***modulator_ila_vio_rtl.xdc*** constraints file which doesn't contain sw0 port constraint

Step 11. The next step is to synthesize the design by clicking on the **Run Synthesis** command from the Flow Navigator, under the Synthesis drop-down list

Step 12. In the **Synthesis Completed** dialog box, select **Open Synthesized Design** option and click **OK**

Step 13. The **Schematic** and the **Debug** window opens

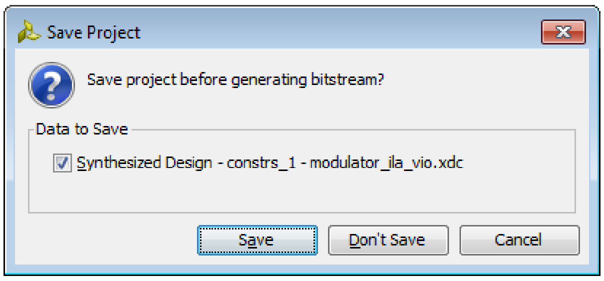
Step 14. In the **Debug** window, click on the **Set up Debug** icon to launch **Set up Debug** wizard to guide you through the process of automatically creating the debug cores and assigning the debug nets to the inputs of the cores

Step 15. Please refer to the **Sub-chapter 11.1 Using the Inserting ILA and VIO Cores into Design** and repeat steps **24** - **32** where is in detail explained how to use **Set up Debug** wizard, how to choose nets and how to connect them to debug cores

Note: Pay attention that maybe some marked debug probes in the **Nets to Debug** dialog box (step 26) would not have specified clock domain. In that case open **Select Clock Domain** dialog box, choose **ALL_CLOCK** instead of default **GLOBAL_CLOCK** nets type, select **clk_in_IBUF** as a new clock domain and click **OK**. Repeat the same procedure for the both (*sine_ampl_s* and *freq_trig_s*) debug nets.

Step 16. You are now ready to implement your design and generate a bitstream file. You can immediately click on the **Generate Bitstream** command in the Flow Navigator, under the Program and Debug drop-down list

Step 17. Since, you have made changes to the netlist by inserting an ILA core, a dialog box with a question should the design be saved prior to generating bitstream file will pop up, see Illustration 13.69


Save Project

Save project before generating bitstream?

Data to Save

☑ Synthesized Design - constrs_1 - modulator_ila_vio.xdc

Figure 13.69: Save Project dialog box

Step 18. Click **Save** in the **Save Project** dialog box

The benefit of saving the project is that if the signals marked for debug remain the same in the original block design, then there is no need to insert the ILA core after synthesis manually as these constraints will take care of it.

Step 19. Program your ZedBoard device (see **Sub-Chapter 10.4 Program Device**)

Step 20. After programming your design, you should get the same results as we presented in the **Sub-chapter 11.2 Debug a Design using Integrated Vivado Logic Analyzer** of this tutorial.

Note: All the information about how to debug your IP integrated design using the Netlist Insertion Flow, you can also find in the **Lab 18: "Debugging IP Integrated Designs"**.

13.4 Creating Modulator IP Core with AXI4 Interface

Advanced eXtensible Interface (AXI) is a standard ARM communication protocol. Xilinx adopted the AXI protocol for IP cores beginning with Spartan-6 and Virtex-6 families and continues to use it with new 7 Series and Zynq-7000 families.

AXI is part of ARM AMBA, a family of micro controller buses. The first version of AXI was first included in AMBA 3.0. AMBA 4.0 includes the second version of AXI, AXI4, which we are using now in our designs.

There are three types of AXI4 interfaces:

- AXI4-Full - for high-performance memory-mapped requirements
- AXI4-Lite - for simple, low-throughput memory-mapped communication
- AXI4-Stream - for high-speed streaming data

In the Vivado IDE you can access Xilinx IP with an AXI4 interface directly from the Vivado IP Catalog and instantiate that IP directly into an RTL design. In the IP Catalog, the AXI4 column shows IP with AXI4 interfaces that are supported and displays the which interfaces are supported by the IP interface.

To integrate our Modulator design in some processor-based system, we need to have AXI interface in our design. In order to show how to work with AXI interface we will add three internal registers: "div_factor_freqhigh", "div_factor_freqlow" and "sw0". The first two registers, "div_factor_freqhigh" and "div_factor_freqlow" will be connected to the div_factor_freqhigh and div_factor_freqlow ports of the Modulator module and will be used for storing division factor values. The third register, "sw0" register, will be connected to the sw0 port of the Modulator module. With this configuration we can change the content of these three registers through AXI interface and easily change the frequency of the pwm signal generation. Block diagram of the new Modulator design with AXI interface is presented on the Illustration 13.70.

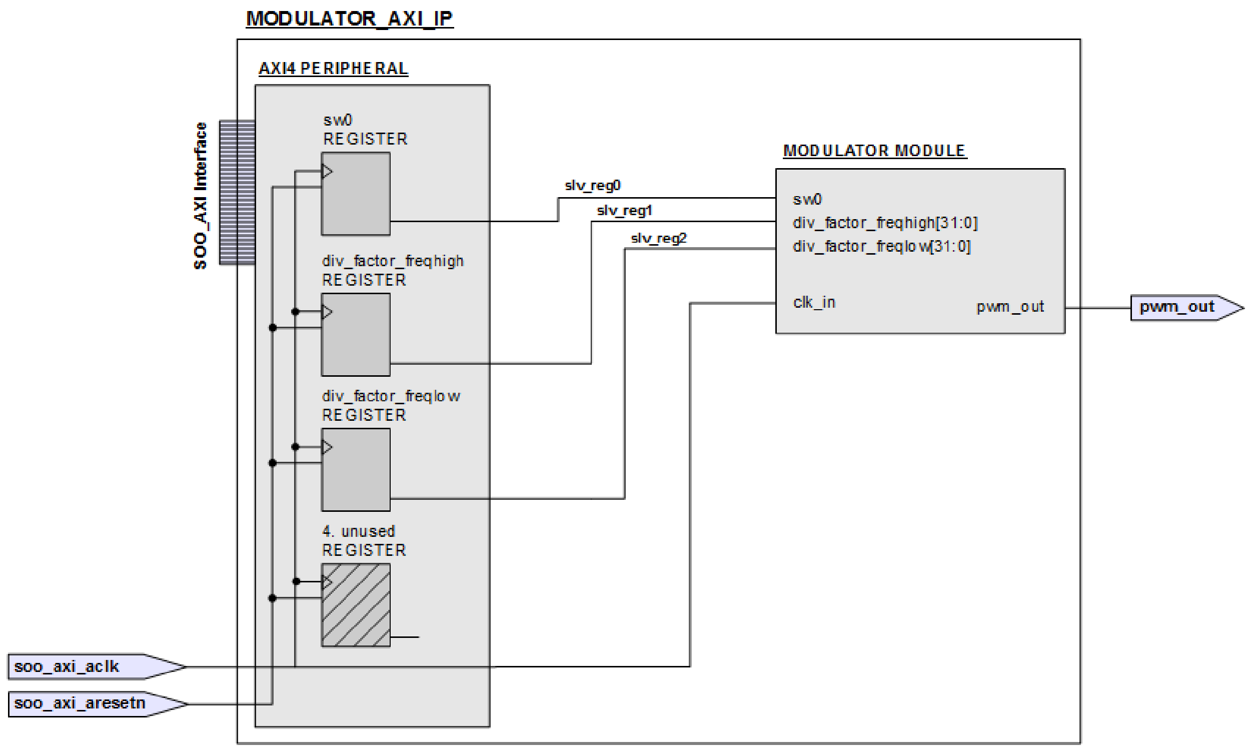


Figure 13.70: Modulator design with AXI interface

From the illustration above we can see that we should create a new Modulator module (for example ***modulator_axi***) with integrated AXI interface and instantiated modulator module (***modulator_rtl.vhd***). At the end we should package this new module as a new IP, e.g. ***modulator_axi_ip***.

The Vivado IDE provides a way to create a new AXI4 peripheral through ***Create and Package IP*** wizard. This wizard takes you through all the required steps and settings necessary for creation of an IP with selected AXI interface (Full, Lite or Stream). This wizard automatically creates interface logic for selected AXI interface type (AXI peripheral block on the Illustration 13.70) and allows user to add user specific logic inside this AXI enabled IP (Modulator module on the Illustration 13.70). In our example, we will configure wizard to create an AXI IP with one AXI-Lite interface. Within AXI peripheral block we will create four 32-bit configuration registers:

- the first register (**sw0 REGISTER** in the block diagram) will be used to replace the sw0 switch from the board
- the second register (**div_factor_freqhigh REGISTER** in the block diagram) will be used to write div_factor_freqhigh values in it
- the third register (**div_factor_freqlow REGISTER** in the block diagram) will be used to write div_factor_freqlow values in it
- the fourth register (**4. unused REGISTER** in the block diagram) will not be used. This register will be generated automatically by the wizard because the minimum number of AXI registers that must be generated is four.

The first step in creating a new ***modulator_axi*** design will be to create a new project:

Step 1. Close the existing **modulator_ipi** project with the **File -> Close Project** option from the main Vivado IDE menu and in the Vivado **Getting Started** page choose **Create New Project** option

Step 2. In the **Create a New Vivado Project** dialog box, click **Next** to confirm the new project creation

Step 3. In the **Project Name** dialog box, enter a name of a new project and specify directory where the project data files will be stored. Name the project ***modulator_axi***, verify the project location, ensure that **Create project subdirectory** is checked and click **Next**

Step 4. In the **Project Type** dialog box, verify that the **RTL Project** is selected and the **Do not specify sources at this time** option is checked and click **Next**

Step 5. In the **Default Part** dialog box, ensure that the **ZedBoard Zynq Evaluation and Development Kit** is selected and click **Next**

Step 6. In the **New Project Summary** dialog box, review the project summary and click **Finish** if you are satisfied with the summary of your project or go back as much as necessary to correct all the questionable issues

The new project, **modulator_axi**, will be automatically opened in the Vivado IDE.

Step 7. To create AXI4 peripheral and to integrate it into our design we will use ***Create and Package IP*** wizard to guide us through all the required steps and settings. In the Vivado IDE main menu, select **Tools -> Create and Package IP...** option, see Illustration 13.71

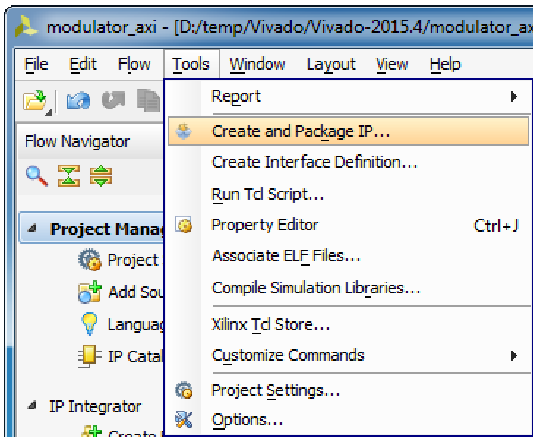
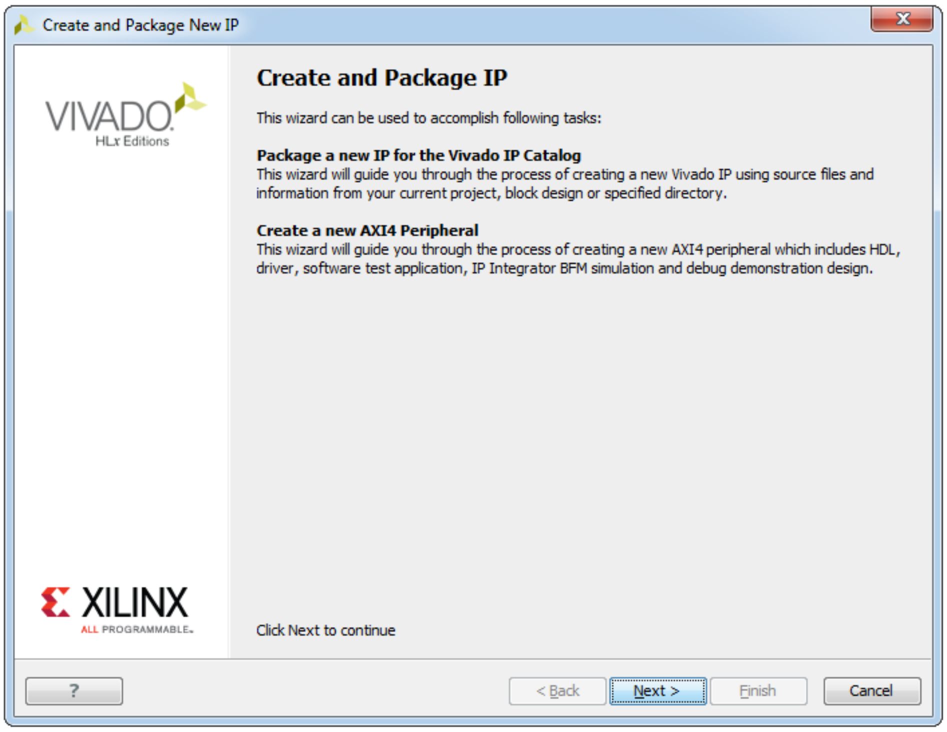


Figure 13.71: Create and Package IP... option

Step 8. In the **Create and Package IP** dialog box, click **Next**



Create and Package New IP

Create and Package IP

This wizard can be used to accomplish following tasks:

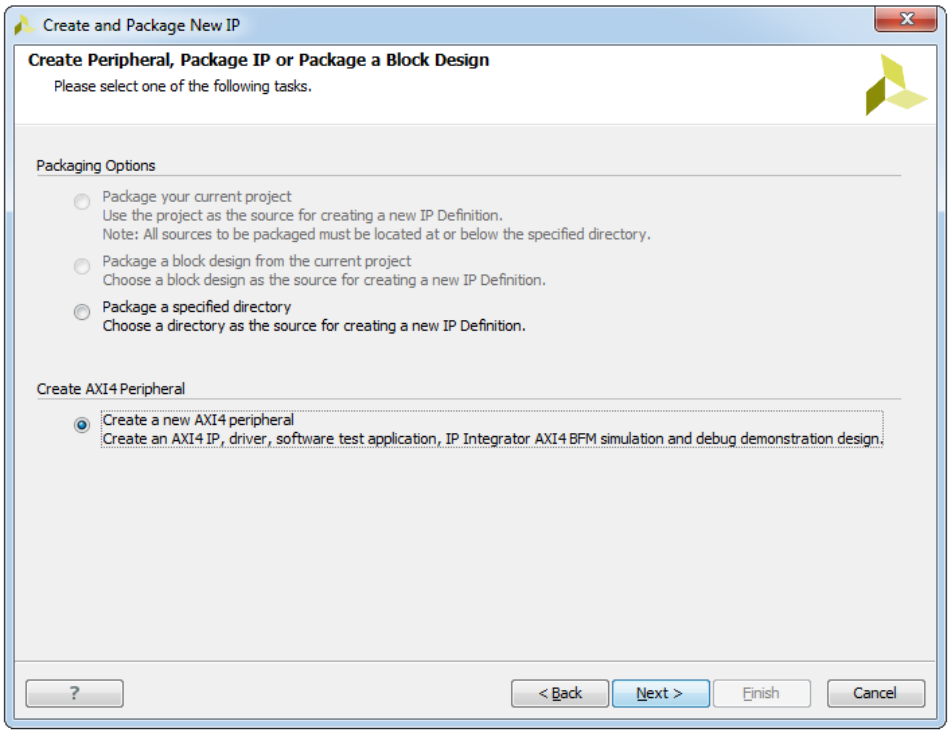
Package a new IP for the Vivado IP Catalog
This wizard will guide you through the process of creating a new Vivado IP using source files and information from your current project, block design or specified directory.

Create a new AXI4 Peripheral
This wizard will guide you through the process of creating a new AXI4 peripheral which includes HDL, driver, software test application, IP Integrator BFM simulation and debug demonstration design.

Click Next to continue

Figure 13.72: Create and Package IP dialog box

Step 9. In the **Create Peripheral, Package IP or Package a Block Design** dialog box, choose to **Create a new AXI4 peripheral** and click **Next**, see Illustration 13.73



Create and Package New IP

Create Peripheral, Package IP or Package a Block Design
Please select one of the following tasks.

Packaging Options

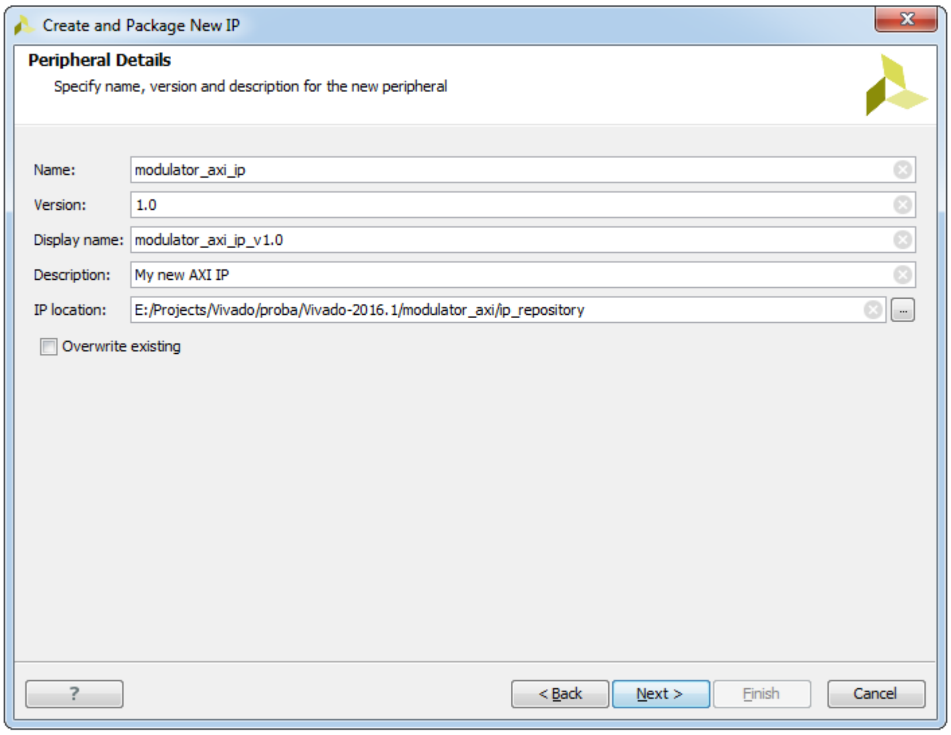
- Package your current project
  Use the project as the source for creating a new IP Definition.
  Note: All sources to be packaged must be located at or below the specified directory.
- Package a block design from the current project
  Choose a block design as the source for creating a new IP Definition.
- Package a specified directory
  Choose a directory as the source for creating a new IP Definition.

Create AXI4 Peripheral

- [✓ Create a new AXI4 peripheral]
  Create an AXI4 IP, driver, software test application, IP Integrator AXI4 BFM simulation and debug demonstration design.

Figure 13.73: Choose Create Peripheral or Package IP dialog box

Step 10. In the **Peripheral Details** dialog box, give the peripheral an appropriate name (***modulator_axi_ip***), description and location, and click **Next**



Peripheral Details

Specify name, version and description for the new peripheral

Name: modulator_axi_ip

Version: 1.0

Display name: modulator_axi_ip_v1.0

Description: My new AXI IP

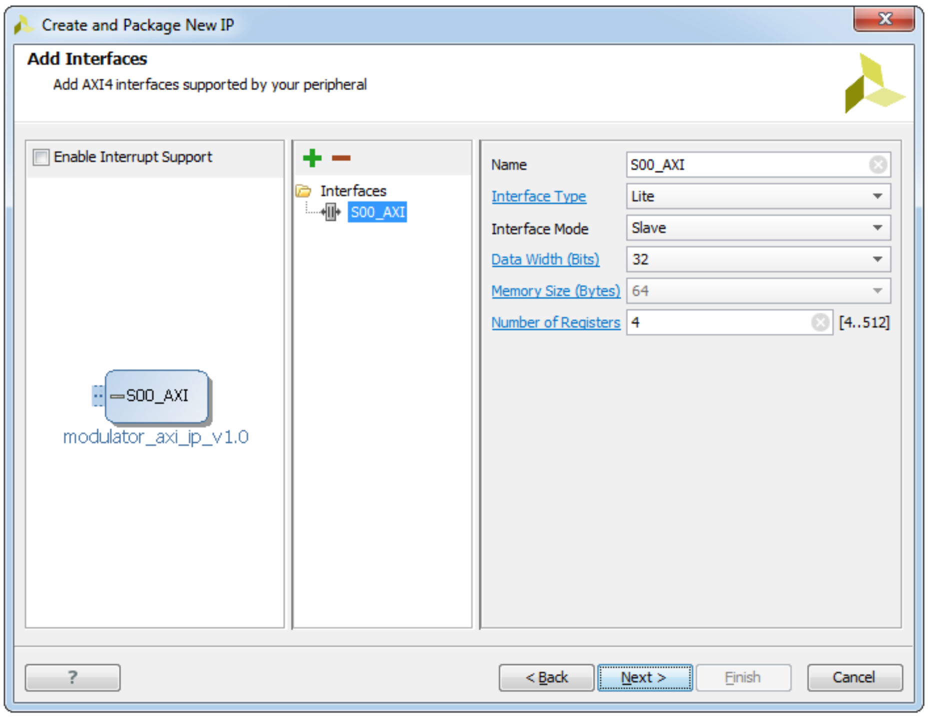
IP location: E:/Projects/Vivado/proba/Vivado-2016.1/modulator_axi/ip_repository

☐ Overwrite existing

Figure 13.74: Peripheral Details dialog box

Note: The **Display Name** you provide shows in the Vivado IP Catalog. You can have different names in the **Name** and **Display Nameb** fields. Any change in the **Name** filed reflects automatically in the **Display Name** filed, which is concatenated with the **Version** field.

Step 11. In the **Add Interfaces** dialog box, we can configure AXI interface. We will use AXI **Lite** interface, it will be **Slave** to the PS, and we will use the minimum number of **4 32-bit** registers of the offered 512 registers. In our design we need only three registers (sw0, div_factor_freqhigh and div_factor_freqlow), so the last one will be unused. Looking to this, we will stick with the default values and just click **Next**


Create and Package New IP

Add Interfaces

Add AXI4 interfaces supported by your peripheral

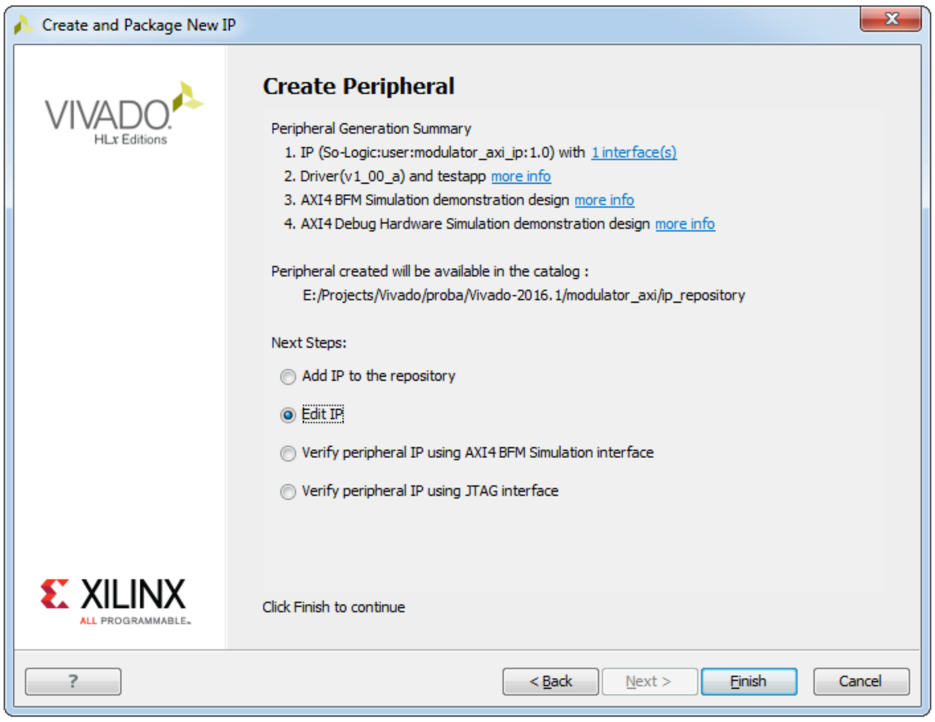
Enable Interrupt Support

Interfaces
S00_AXI

Name	S00_AXI
Interface Type	Lite
Interface Mode	Slave
Data Width (Bits)	32
Memory Size (Bytes)	64
Number of Registers	4 [4..512]

Figure 13.75: Add Interfaces dialog box

Step 12. In the last **Crate Peripheral** dialog box, select **Edit IP** option and click **Finish**, see Illustration 13.76. Another Vivado window will open, which will allow you to modify the peripheral that we just created, see Illustration 13.77.


Create and Package New IP

Create Peripheral

Peripheral Generation Summary

1. IP (So-Logic:user:modulator_axi_ip:1.0) with 1 interface(s)
2. Driver(v1_00_a) and testapp more info
3. AXI4 BFM Simulation demonstration design more info
4. AXI4 Debug Hardware Simulation demonstration design more info

Peripheral created will be available in the catalog :
E:/Projects/Vivado/proba/Vivado-2016.1/modulator_axi/ip_repository

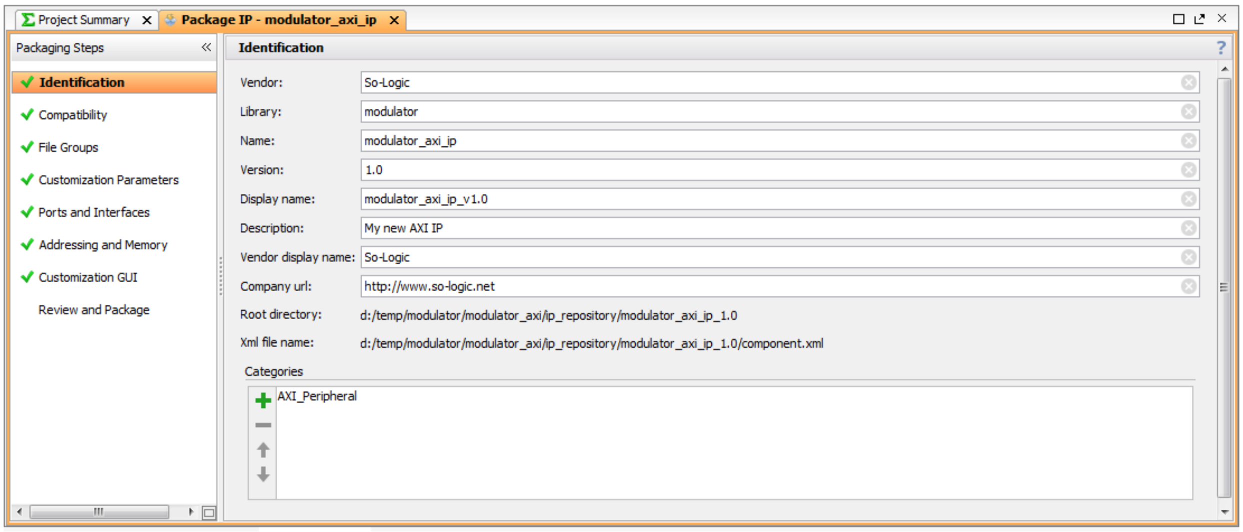
Next Steps:

- Add IP to the repository
- [✓ Edit IP]
- Verify peripheral IP using AXI4 BFM Simulation interface
- Verify peripheral IP using JTAG interface

Click Finish to continue

Figure 13.76: Create Peripheral dialog box

Step 13. In the **Package IP - modulator_axi_ip** window, in the **Identification** section, fill some basic information about your new ***modulator_axi_ip*** IP, see Illustration 13.77


Project Summary | Package IP - modulator_axi_ip

Packaging Steps: Identification, Compatibility, File Groups, Customization Parameters, Ports and Interfaces, Addressing and Memory, Customization GUI, Review and Package

Identification

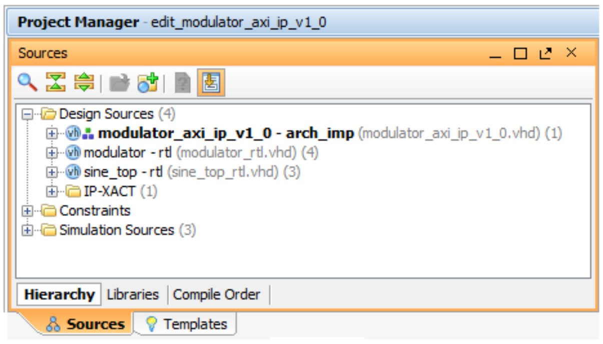
Vendor: So-Logic
Library: modulator
Name: modulator_axi_ip
Version: 1.0
Display name: modulator_axi_ip_v1.0
Description: My new AXI IP
Vendor display name: So-Logic
Company url: http://www.so-logic.net
Root directory: d:/temp/modulator/modulator_axi/ip_repository/modulator_axi_ip_1.0
Xml file name: d:/temp/modulator/modulator_axi/ip_repository/modulator_axi_ip_1.0/component.xml
Categories: AXI_Peripheral

Figure 13.77: Identification window

At this point, the peripheral that has been generated by Vivado is an AXI Lite slave, that contains 4x32-bit read/write registers. What we want is to add our Modulator module to the ***modulator_axi_ip*** IP and connect it with the three AXI registers, see block diagram on the Figure 13.70 from the beginning of this chapter.

Step 14. In the Flow Navigator, click **Add Sources** command to add all the necessary Modulator module source files (*frequency_trigger_rt.vhd, counter_rtl.vhd, modulator_pkg.vhd, sine_rtl.vhd, sine_top_rtl.vhd, pwm_rtl.vhd and modulator-_rtl.vhd*) and after adding your Hierarchy tab should look like as it is shown on the Illustration 13.78

Note: In the **Add or Create Design Sources** dialog box don't forget to enable **Copy sources into IP Directory** option.


Project Manager - edit_modulator_axi_ip_v1_0

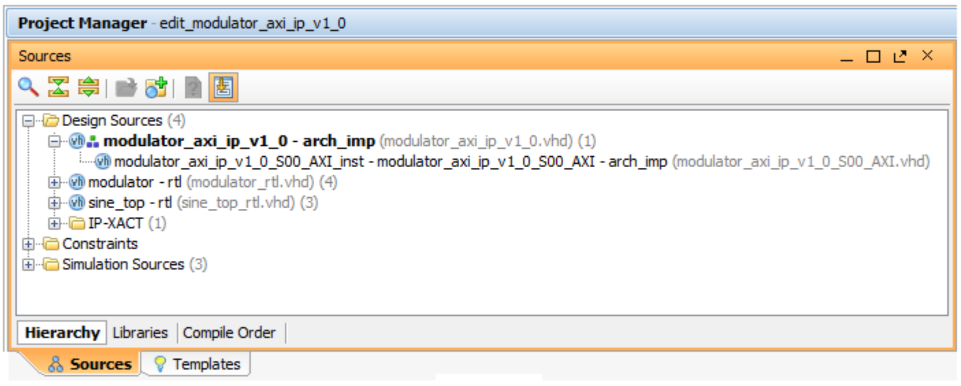
Sources

```
Design Sources (4)
    modulator_axi_ip_v1_0 - arch_imp (modulator_axi_ip_v1_0.vhd) (1)
    modulator - rtl (modulator_rtl.vhd) (4)
    sine_top - rtl (sine_top_rtl.vhd) (3)
    IP-XACT (1)
Constraints
Simulation Sources (3)
```

Hierarchy | Libraries | Compile Order
Sources | Templates

Figure 13.78: Hierarchy tab after adding all the necessary source files in the IP

Step 15. Now is the time to modify AXI peripheral. Open the branch ***"modulator_axi_ip_v1_0 - arch_imp"***, see Illustration 13.79



Project Manager - edit_modulator_axi_ip_v1_0

Sources

```
Design Sources (4)
  modulator_axi_ip_v1_0 - arch_imp (modulator_axi_ip_v1_0.vhd) (1)
    modulator_axi_ip_v1_0_S00_AXI_inst - modulator_axi_ip_v1_0_S00_AXI - arch_imp (modulator_axi_ip_v1_0_S00_AXI.vhd)
  modulator - rtl (modulator_rtl.vhd) (4)
  sine_top - rtl (sine_top_rtl.vhd) (3)
  IP-XACT (1)
Constraints
Simulation Sources (3)
```

Hierarchy | Libraries | Compile Order

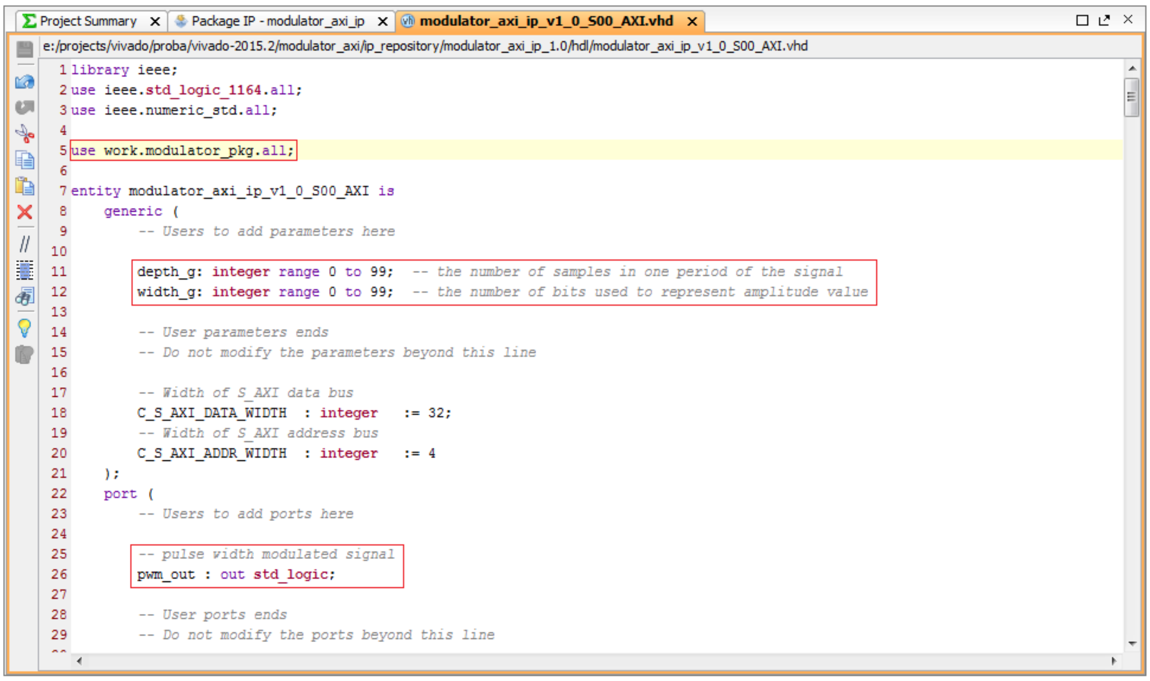
Sources | Templates

Figure 13.79: Hierarchy tab with opened modulator_axi_ip_v1_0 - arch_imp branch

Step 16. Double-click on the ***"modulator_axi_ip_v1_0_S00_AXI_inst"*** file to open it

Step 17. In the ***"modulator_axi_ip_v1_0_S00_AXI.vhd"*** file make the following changes:

- add ***modulator_pkg*** package
- in the entity declaration, add ***depth_g*** and ***width_g*** generics in the generic map, below the first comment line **"-- Users to add parameters here"**
- in the entity declaration, add ***pwm_out*** port as 1-bit output port in the port map, below the comment line **"-- Users to add ports here"**, see Illustration 13.80
- create constant ***design_setting_c***, as it is shown on the Illustration 13.80

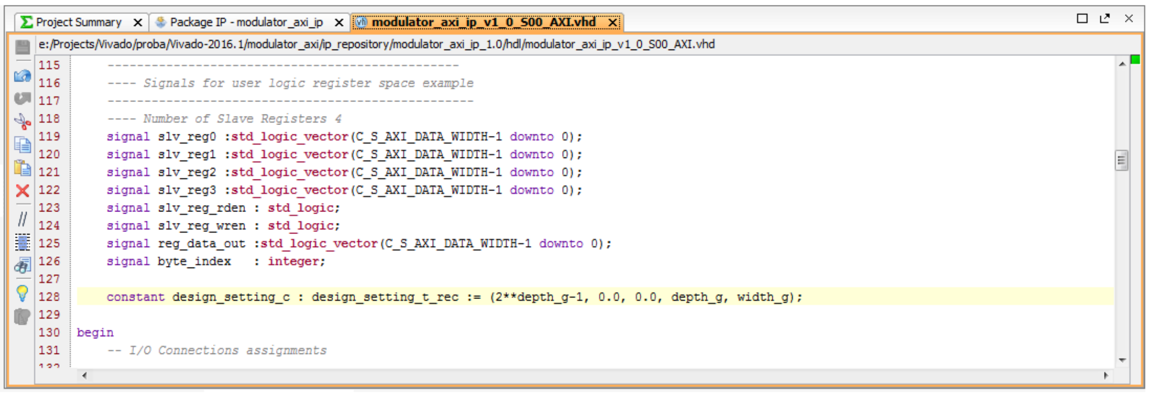


Project Summary | Package IP - modulator_axi_ip | modulator_axi_ip_v1_0_S00_AXI.vhd

e:/projects/vivado/proba/vivado-2015.2/modulator_axi/ip_repository/modulator_axi_ip_1.0/hdl/modulator_axi_ip_v1_0_S00_AXI.vhd

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

use work.modulator_pkg.all;

entity modulator_axi_ip_v1_0_S00_AXI is
    generic (
        -- Users to add parameters here

        depth_g: integer range 0 to 99;  -- the number of samples in one period of the signal
        width_g: integer range 0 to 99;  -- the number of bits used to represent amplitude value

        -- User parameters ends
        -- Do not modify the parameters beyond this line

        -- Width of S_AXI data bus
        C_S_AXI_DATA_WIDTH  : integer   := 32;
        -- Width of S_AXI address bus
        C_S_AXI_ADDR_WIDTH  : integer   := 4
    );
    port (
        -- Users to add ports here

        -- pulse width modulated signal
        pwm_out : out std_logic;

        -- User ports ends
        -- Do not modify the ports beyond this line
```

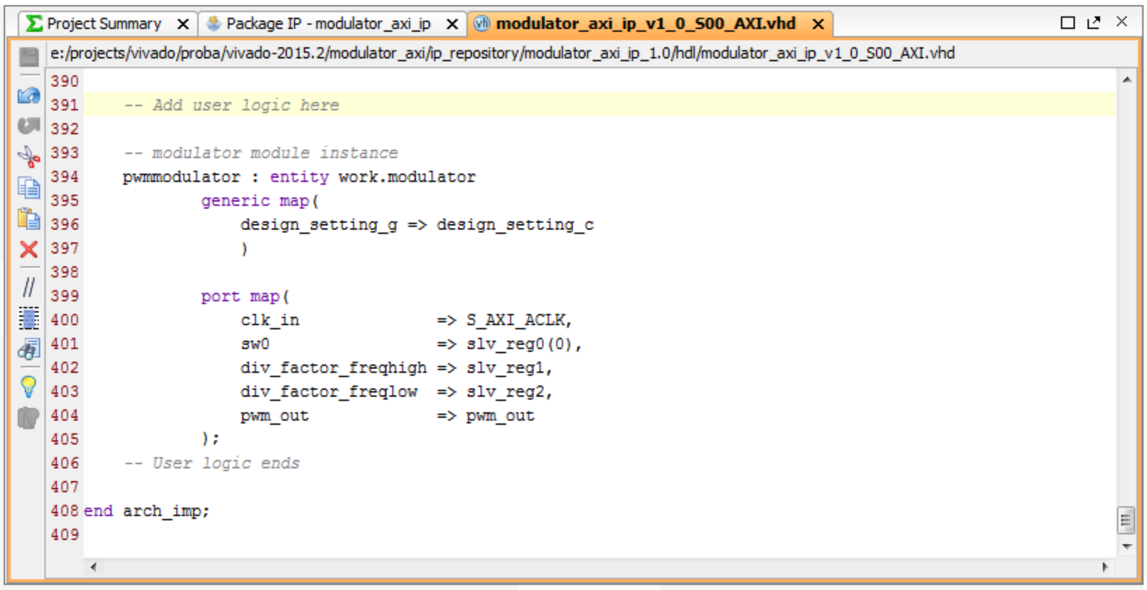
Figure 13.80: Modified modulator_axi_ip_v1_0_S00_AXI.vhd file - part 1


Project Summary
Package IP - modulator_axi_ip
modulator_axi_ip_v1_0_S00_AXI.vhd
e:/Projects/Vivado/proba/Vivado-2016.1/modulator_axi/ip_repository/modulator_axi_ip_1.0/hdl/modulator_axi_ip_v1_0_S00_AXI.vhd

```
    ------------------------------------------------
    ---- Signals for user logic register space example
    --------------------------------------------------
    ---- Number of Slave Registers 4
    signal slv_reg0 :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
    signal slv_reg1 :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
    signal slv_reg2 :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
    signal slv_reg3 :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
    signal slv_reg_rden : std_logic;
    signal slv_reg_wren : std_logic;
    signal reg_data_out :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
    signal byte_index   : integer;

    constant design_setting_c : design_setting_t_rec := (2**depth_g-1, 0.0, 0.0, depth_g, width_g);

begin
    -- I/O Connections assignments
```

Figure 13.81: Modified modulator_axi_ip_v1_0_S00_AXI.vhd file - part 2

Step 18. Now, at the end of this source code find the comment **"-- Add user logic here"** and below this comment instantiate Modulator module. Connect Modulator module ports to the AXI peripheral on the same way as it is shown on the Illustration 13.82

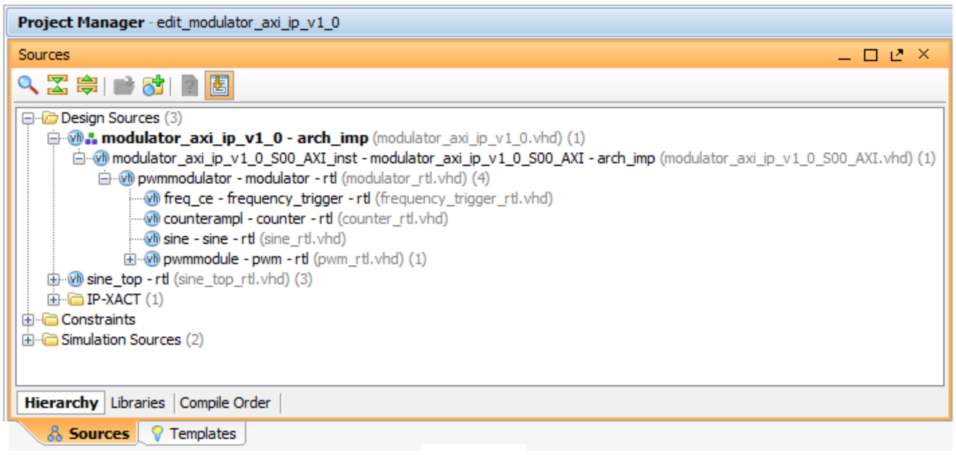

Project Summary
Package IP - modulator_axi_ip
modulator_axi_ip_v1_0_S00_AXI.vhd
e:/projects/vivado/proba/vivado-2015.2/modulator_axi/ip_repository/modulator_axi_ip_1.0/hdl/modulator_axi_ip_v1_0_S00_AXI.vhd

```

    -- Add user logic here

    -- modulator module instance
   pwmmodulator : entity work.modulator
           generic map(
               design_setting_g => design_setting_c
               )

           port map(
               clk_in              => S_AXI_ACLK,
               sw0                 => slv_reg0(0),
               div_factor_freqhigh => slv_reg1,
               div_factor_freqlow  => slv_reg2,
               pwm_out             => pwm_out
           );
    -- User logic ends

end arch_imp;

```

Figure 13.82: modulator_axi_ip_v1_0_S00_AXI.vhd file with instantiated Modulator module

Step 19. Save the file

Step 20. You should notice that the ***modulator_rtl.vhd*** source file has been integrated into the hierarchy, because we have instantiated it within the AXI peripheral, see Illustration 13.83



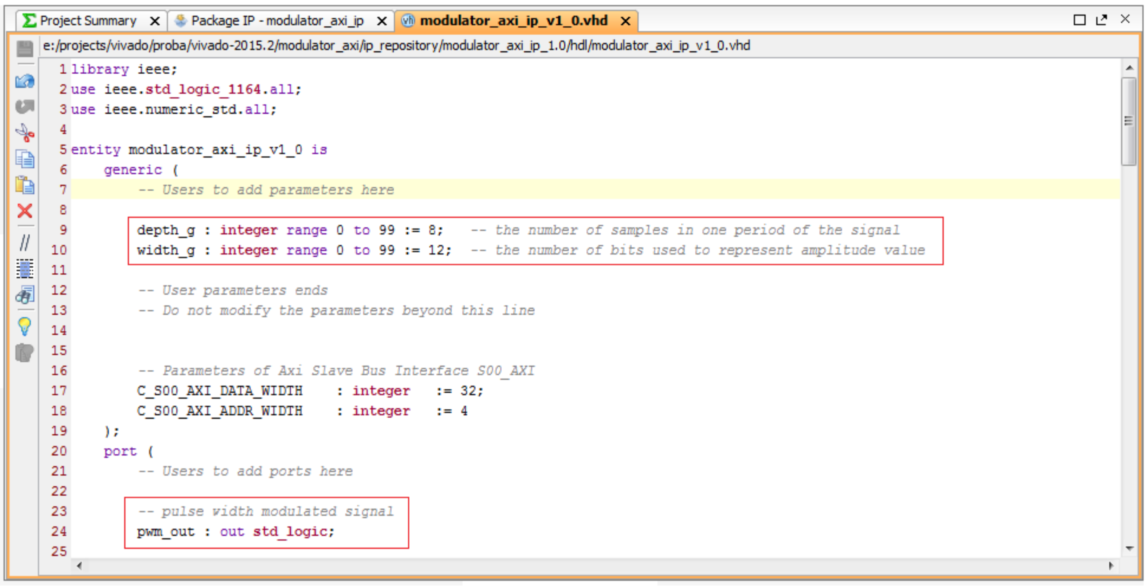
```
Project Manager - edit_modulator_axi_ip_v1_0
Sources
Design Sources (3)
  modulator_axi_ip_v1_0 - arch_imp (modulator_axi_ip_v1_0.vhd) (1)
    modulator_axi_ip_v1_0_S00_AXI_inst - modulator_axi_ip_v1_0_S00_AXI - arch_imp (modulator_axi_ip_v1_0_S00_AXI.vhd) (1)
      pwmmodulator - modulator - rtl (modulator_rtl.vhd) (4)
        freq_ce - frequency_trigger - rtl (frequency_trigger_rtl.vhd)
        counterampl - counter - rtl (counter_rtl.vhd)
        sine - sine - rtl (sine_rtl.vhd)
        pwmmodule - pwm - rtl (pwm_rtl.vhd) (1)
  sine_top - rtl (sine_top_rtl.vhd) (3)
  IP-XACT (1)
Constraints
Simulation Sources (2)
Hierarchy | Libraries | Compile Order
Sources | Templates
```

Figure 13.83: Hierarchy window with integrated Modulator module within AXI peripheral

Step 21. Now, double-click on the ***"modulator_axi_ip_v1_0 - arch_imp"*** file to open it

Step 22. In the ***"modulator_axi_ip_v1_0.vhd"*** file make the following changes:

- in the entity declaration, add ***depth_g*** and ***width_g*** generics in the generic map, below the first comment line **"-- Users to add parameters here"**
- in the entity declaration, add ***pwm_out*** port as 1-bit output port in the port map, below the comment line **"-- Users to add ports here"**, see Illustration 13.84



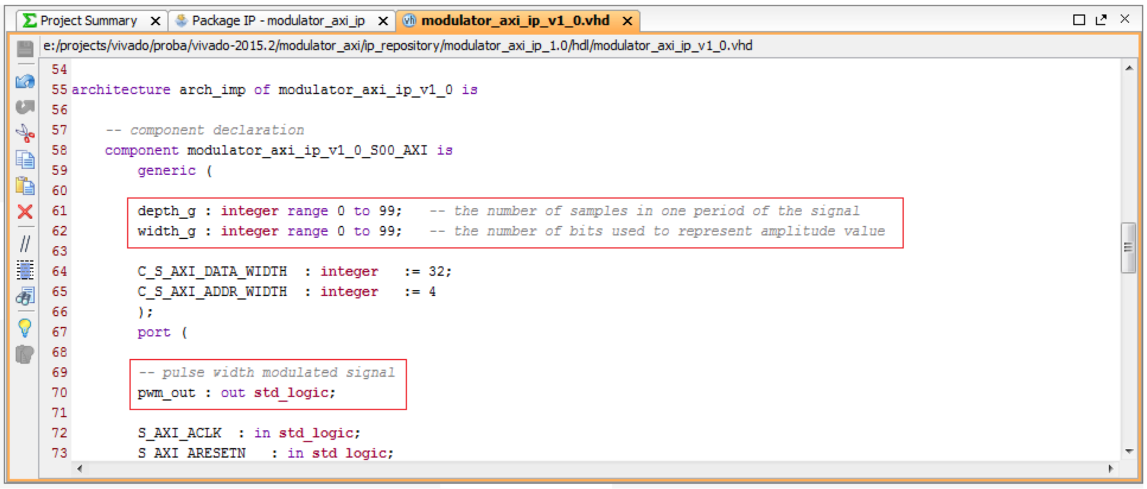
Project Summary | Package IP - modulator_axi_ip | modulator_axi_ip_v1_0.vhd

e:/projects/vivado/proba/vivado-2015.2/modulator_axi/ip_repository/modulator_axi_ip_1.0/hdl/modulator_axi_ip_v1_0.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity modulator_axi_ip_v1_0 is
    generic (
        -- Users to add parameters here

        depth_g : integer range 0 to 99 := 8;    -- the number of samples in one period of the signal
        width_g : integer range 0 to 99 := 12;   -- the number of bits used to represent amplitude value

        -- User parameters ends
        -- Do not modify the parameters beyond this line


        -- Parameters of Axi Slave Bus Interface S00_AXI
        C_S00_AXI_DATA_WIDTH    : integer   := 32;
        C_S00_AXI_ADDR_WIDTH    : integer   := 4
    );
    port (
        -- Users to add ports here

        -- pulse width modulated signal
        pwm_out : out std_logic;

```

Figure 13.84: Modified modulator_axi_ip_v1_0.vhd source file - part 1

Step 23. Now, in the *modulator_axi_ip_v1_0_S00_AXI* component declaration add ***depth_g*** and ***width_g*** generics in the generic map and ***pwm_out*** port in the port map, see Illustration 13.85



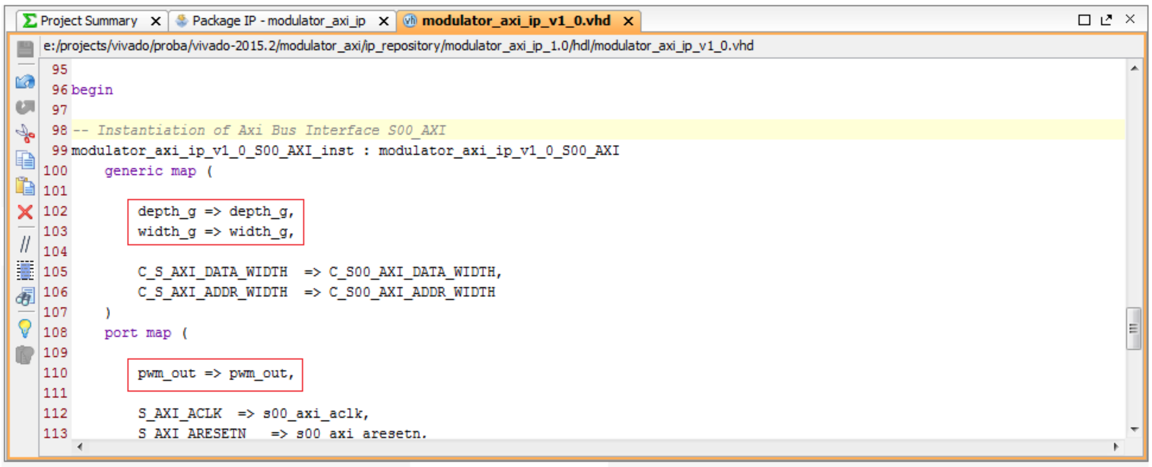
Project Summary | Package IP - modulator_axi_ip | modulator_axi_ip_v1_0.vhd

e:/projects/vivado/proba/vivado-2015.2/modulator_axi/ip_repository/modulator_axi_ip_1.0/hdl/modulator_axi_ip_v1_0.vhd

```

architecture arch_imp of modulator_axi_ip_v1_0 is

    -- component declaration
    component modulator_axi_ip_v1_0_S00_AXI is
        generic (

        depth_g : integer range 0 to 99;   -- the number of samples in one period of the signal
        width_g : integer range 0 to 99;   -- the number of bits used to represent amplitude value

        C_S_AXI_DATA_WIDTH  : integer   := 32;
        C_S_AXI_ADDR_WIDTH  : integer   := 4
        );
        port (

        -- pulse width modulated signal
        pwm_out : out std_logic;

        S_AXI_ACLK  : in std_logic;
        S_AXI_ARESETN   : in std_logic;
```

Figure 13.85: Modified modulator_axi_ip_v1_0.vhd source file - part 2

Step 24. In the *modulator_axi_ip_v1_0_S00_AXI* component instance assign ***depth_g*** and ***width_g*** generics to their values and connect ***pwm_out*** port of the *modulator_axi_ip_v1_0_S00_AXI* component to the pwm_out port of the IP, see Illustration 13.86



Project Summary | Package IP - modulator_axi_ip | modulator_axi_ip_v1_0.vhd

e:/projects/vivado/proba/vivado-2015.2/modulator_axi/ip_repository/modulator_axi_ip_1.0/hdl/modulator_axi_ip_v1_0.vhd

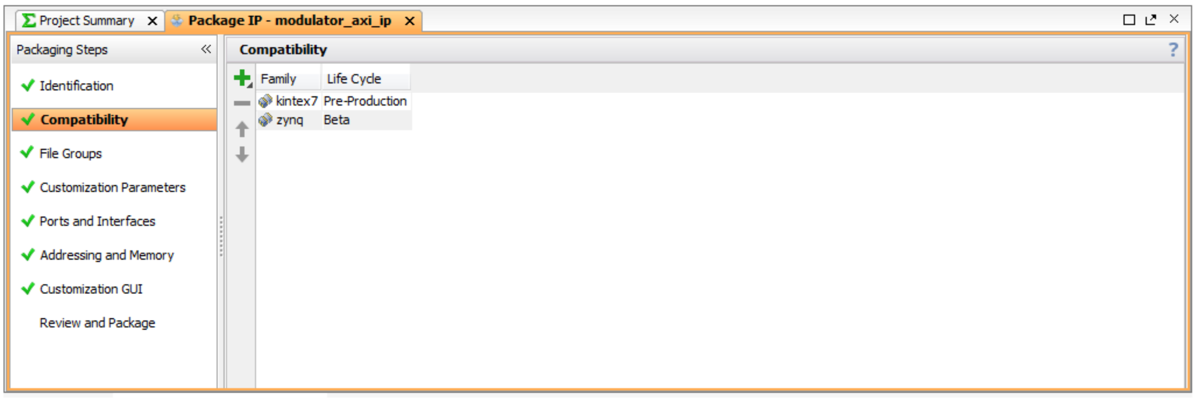
```

begin

-- Instantiation of Axi Bus Interface S00_AXI
modulator_axi_ip_v1_0_S00_AXI_inst : modulator_axi_ip_v1_0_S00_AXI
    generic map (

        depth_g => depth_g,
        width_g => width_g,

        C_S_AXI_DATA_WIDTH  => C_S00_AXI_DATA_WIDTH,
        C_S_AXI_ADDR_WIDTH  => C_S00_AXI_ADDR_WIDTH
    )
    port map (

        pwm_out => pwm_out,

        S_AXI_ACLK  => s00_axi_aclk,
        S_AXI_ARESETN   => s00_axi_aresetn,
```

Figure 13.86: Modified modulator_axi_ip_v1_0.vhd source file - part 3

Step 25. Save the file

Step 26. In the **Package IP - modulator_axi_ip** window, open **Compatibility** section and click "+" icon to add the family with whom you want your packaged IP core to be compatible. Beside Zynq family we will also add Kintex-7 family, see Illustration 13.87.

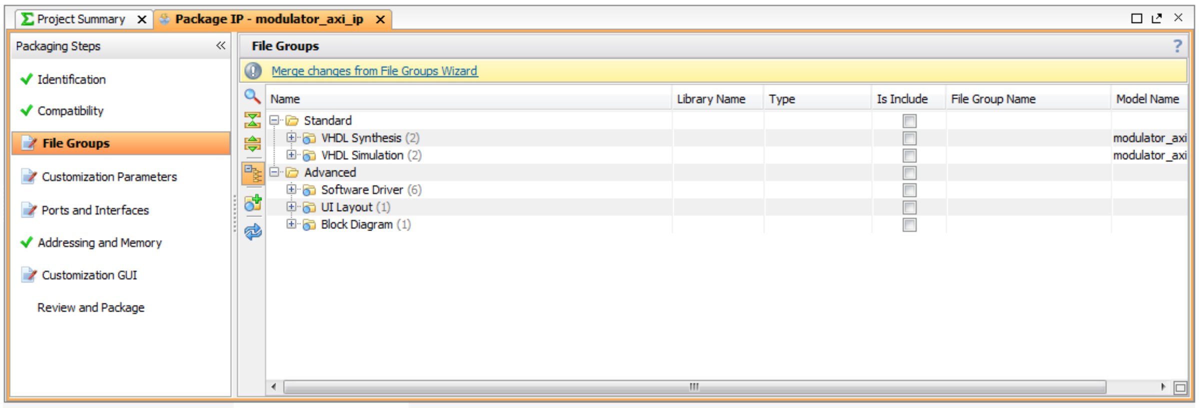
Zynq-7000 family is also used in *"Embedded System Design Tutorial"*, when illustrating how to build an embedded system around ARM processor. Since this packaged IP core will be used in ARM-based embedded system we must make it compatible with Zynq-7000 family.



Family	Life Cycle
kintex7	Pre-Production
zynq	Beta

Figure 13.87: Compatibility window

Step 27. In the **Package IP - modulator_axi_ip** window, open **File Groups** section, and click **Merge changes from File Groups Wizard** link, see Illustration 13.88

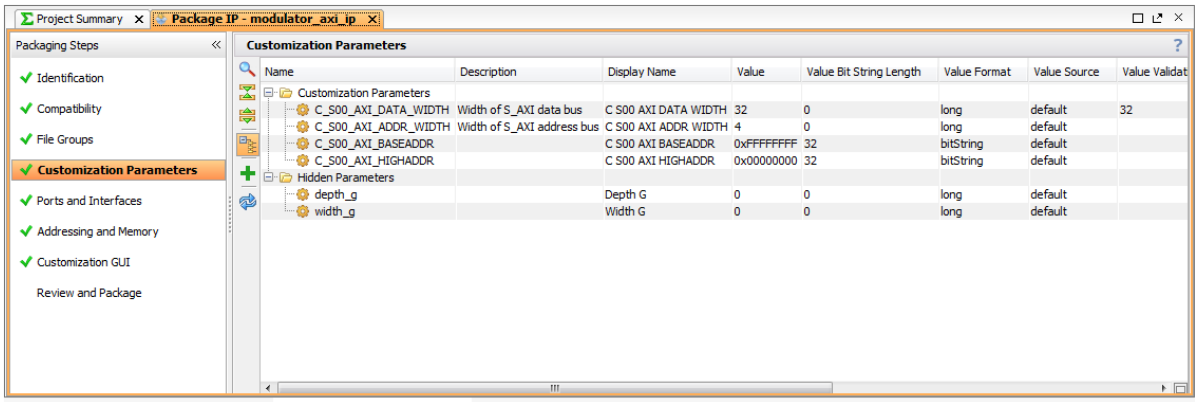


Name	Library Name	Type	Is Include	File Group Name	Model Name
Standard			☐		
VHDL Synthesis (2)			☐		modulator_axi
VHDL Simulation (2)			☐		modulator_axi
Advanced			☐		
Software Driver (6)			☐		
UI Layout (1)			☐		
Block Diagram (1)			☐		

Figure 13.88: File Groups window

Step 28. In the **Package IP - modulator_axi_ip** window, open **Customization Parameters** section, and click **Merge changes from Customization Parameters Wizard** link. After merging changes from Customization Parameters Wizard, Customization Parameters window should look like as it is show on the Illustration 13.89.

Note: After this step, you should get a green tick not only in **Customization Parameters** section, but also in **Ports and Interfaces** and **Customization GUI** sections.



Name	Description	Display Name	Value	Value Bit String Length	Value Format	Value Source	Value Validat
Customization Parameters							
C_S00_AXI_DATA_WIDTH	Width of S_AXI data bus	C S00 AXI DATA WIDTH	32	0	long	default	32
C_S00_AXI_ADDR_WIDTH	Width of S_AXI address bus	C S00 AXI ADDR WIDTH	4	0	long	default	
C_S00_AXI_BASEADDR		C S00 AXI BASEADDR	0xFFFFFFFF	32	bitString	default	
C_S00_AXI_HIGHADDR		C S00 AXI HIGHADDR	0x00000000	32	bitString	default	
Hidden Parameters							
depth_g		Depth G	0	0	long	default	
width_g		Width G	0	0	long	default	

Figure 13.89: Customization Parameters window after merging changes from Customization Parameters Wizard

Step 29. In the **Customization Parameters** window, unhide the **Hidden Parameters** and hide the **Customization Param-**

eters, because we would like to have only *depth_g* and *width_g* visible in the ***modulator_axi_ip_v1.0*** IP Customization GUI.

If you would like to unhide some IP Parameter, select it, right-click on it, choose **Edit Parameter...** option and in the **Edit IP Parameter** dialog box enable **Visable in Customization GUI** option and click **OK**, see Illustration 13.90.

If you would like to hide some IP Parameter, just disable the **Visable in Customization GUI** option in the **Edit IP Parameter** dialog box.

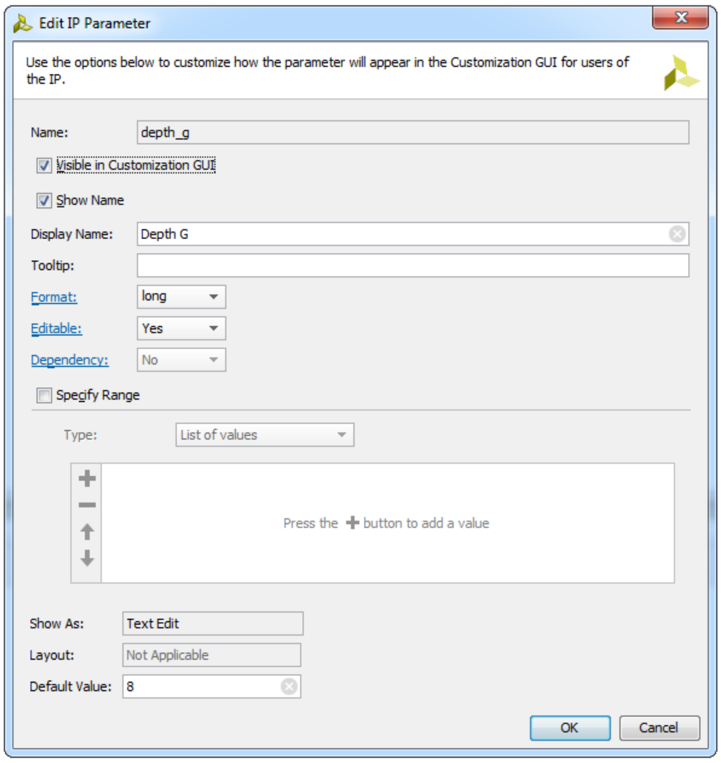


Figure 13.90: Edit IP Parameter window

Step 30. Now, open **Review and Package** section and click **Re- Package IP** option, see Illustration 13.91

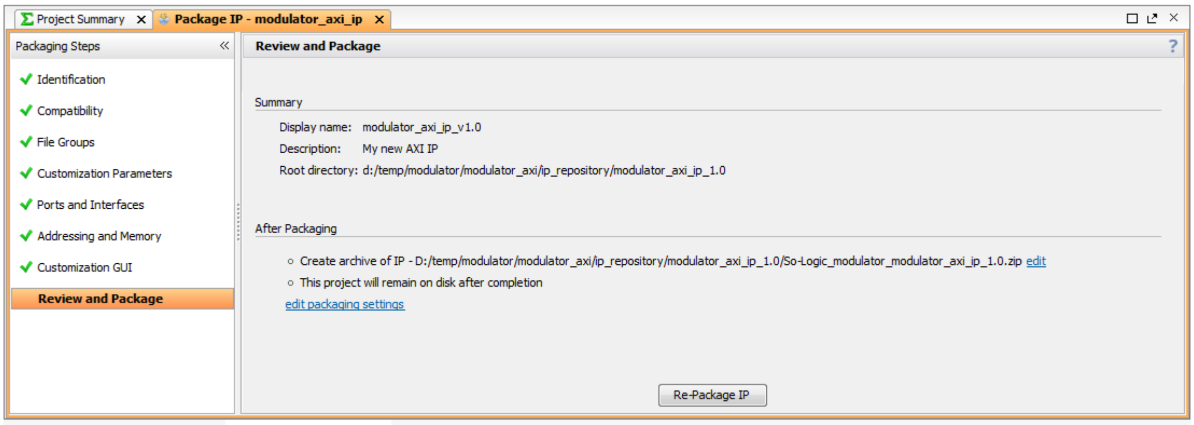
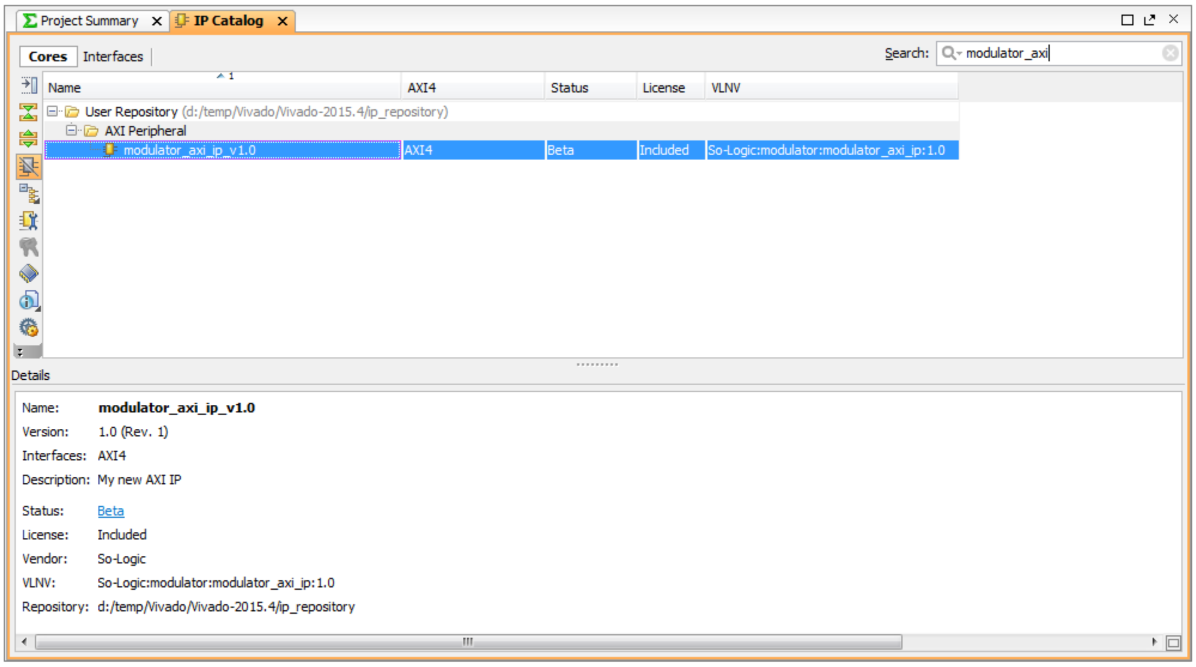

Project Summary
Package IP - modulator_axi_ip
Packaging Steps
Identification
Compatibility
File Groups
Customization Parameters
Ports and Interfaces
Addressing and Memory
Customization GUI
Review and Package
Review and Package
Summary
Display name: modulator_axi_ip_v1.0
Description: My new AXI IP
Root directory: d:/temp/modulator/modulator_axi/ip_repository/modulator_axi_ip_1.0
After Packaging
Create archive of IP - D:/temp/modulator/modulator_axi/ip_repository/modulator_axi_ip_1.0/So-Logic_modulator_modulator_axi_ip_1.0.zip edit
This project will remain on disk after completion
edit packaging settings
Re-Package IP

Figure 13.91: Review and Package window

The new AXI peripheral with instantiated Modulator module in it will be packaged and the Vivado window for the peripheral should be automatically closed. We should now be able to find our ***modulator_axi_ip*** IP in the IP Catalog.

Step 31. Open **IP Catalog** and search for ***modulator_axi_ip*** IP, see Illustration 13.92. When you find it, double-click on it to customize and generate the IP.

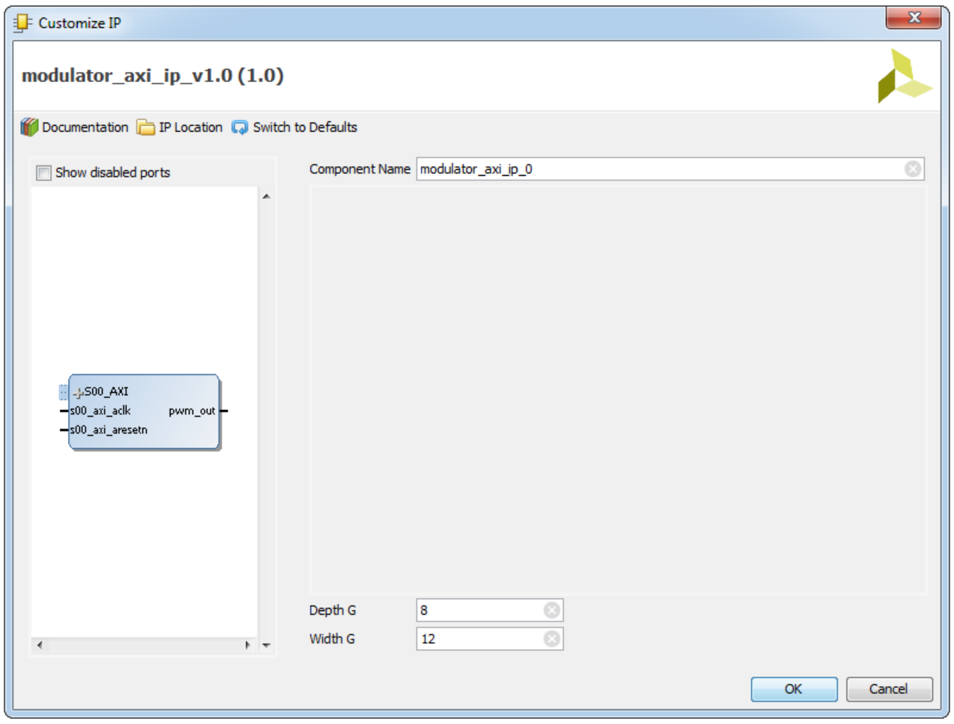


Name	AXI4	Status	License	VLNV
User Repository (d:/temp/Vivado/Vivado-2015.4/ip_repository)				
AXI Peripheral				
modulator_axi_ip_v1.0	AXI4	Beta	Included	So-Logic:modulator:modulator_axi_ip:1.0

Name:	**modulator_axi_ip_v1.0**
Version:	1.0 (Rev. 1)
Interfaces:	AXI4
Description:	My new AXI IP
Status:	Beta
License:	Included
Vendor:	So-Logic
VLNV:	So-Logic:modulator:modulator_axi_ip:1.0
Repository:	d:/temp/Vivado/Vivado-2015.4/ip_repository

Figure 13.92: IP Catalog with modulator_axi_ip IP

Step 32. In the **modulator_axi_ip_v1.0 (1.0)** customization window, check is **Depth G** set to **8** and **Width G** to **12** and if it is, click **OK**, see Illustration 13.93



Component Name: modulator_axi_ip_0

Depth G: 8

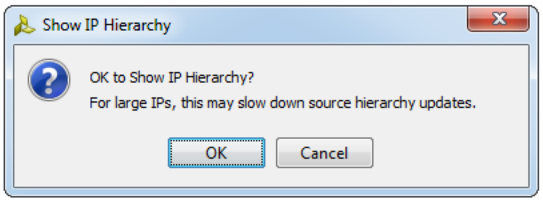
Width G: 12

Figure 13.93: Customize IP - modulator_axi_ip_v1.0

Step 33. In the **Generate Output Products** dialog box, click **Generate** to generate the ***modulator_axi_ip_0*** IP

Step 34. In the **Sources** window expand ***modulator_axi_ip_0*** IP to see what the tool has created for us

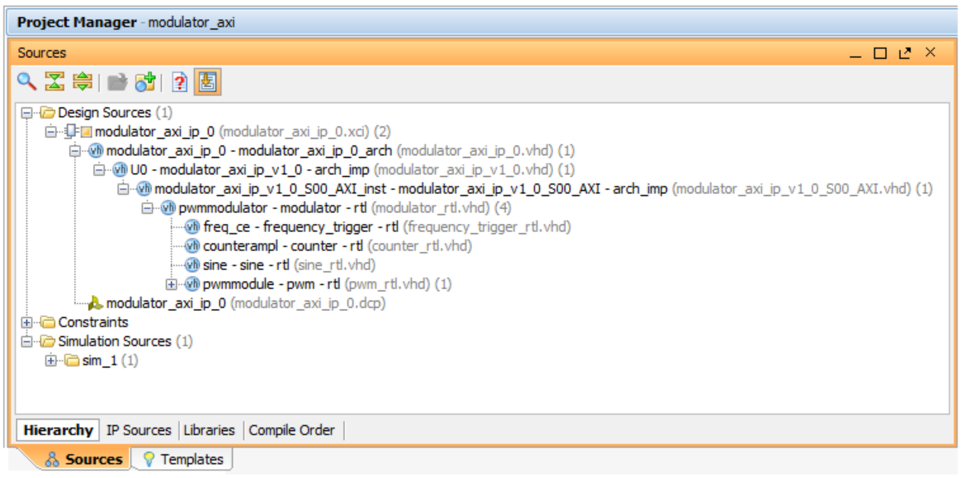
Step 35. When you try to expand ***modulator_axi_ip_0*** IP, **Show IP Hierarchy** dialog box will appear. Click **OK** to open the ***modulator_axi_ip_0*** IP hierarchy, see Illustration 13.94


Show IP Hierarchy

OK to Show IP Hierarchy?
For large IPs, this may slow down source hierarchy updates.

Figure 13.94: Show IP Hierarchy dialog box

Step 36. In the **Sources** window expand all the levels of ***modulator_axi_ip_0*** IP hierarchy, see Illustration 13.95. You can see the structure of the ***modulator_axi_ip_0*** IP.


Project Manager - modulator_axi

Sources

```
Design Sources (1)
  modulator_axi_ip_0 (modulator_axi_ip_0.xci) (2)
    modulator_axi_ip_0 - modulator_axi_ip_0_arch (modulator_axi_ip_0.vhd) (1)
      U0 - modulator_axi_ip_v1_0 - arch_imp (modulator_axi_ip_v1_0.vhd) (1)
        modulator_axi_ip_v1_0_S00_AXI_inst - modulator_axi_ip_v1_0_S00_AXI - arch_imp (modulator_axi_ip_v1_0_S00_AXI.vhd) (1)
          pwmmodulator - modulator - rtl (modulator_rtl.vhd) (4)
            freq_ce - frequency_trigger - rtl (frequency_trigger_rtl.vhd)
            counterampl - counter - rtl (counter_rtl.vhd)
            sine - sine - rtl (sine_rtl.vhd)
            pwmmodule - pwm - rtl (pwm_rtl.vhd) (1)
    modulator_axi_ip_0 (modulator_axi_ip_0.dcp)
Constraints
Simulation Sources (1)
  sim_1 (1)
```

Figure 13.95: Sources window with modulator_axi_ip_0 sources hierarchy

Step 37. At the end, we must verify our Modulator IP core with AXI4 interface

To write appropriate test bench file for our new Modulator IP core with AXI4 interface, we must first get acquainted with AXI4-Lite interface signals. The AXI4-Lite interface signals are listed and described in the Table 12.1.

Table 12.1: AXI4_Lite Interface Signals Descriptions

Signal Name	I/O	Initial State	Description
AXI Global System Signals			
S_AXI_ACLK	I	-	AXI Clock.
S_AXI_ARESETN	I	-	AXI Reset, active-low.
AXI Write Address Channel Signals			
S_AXI_AWADDR[C_S_A-XI_ADDR_WIDTH-1:0]	I	-	AXI write address. The write address bus gives the address of the write transaction.
S_AXI_AWPROT[2:0]	I	-	AXI write address protection signal. "000" value is recommended. Infrastructure IP passes Protection bits across a system.

S_AXI_AWVALID	I	-	Write address valid. This signal indicates that valid write address and control information are available.
S_AXI_AWREADY	O	0	Write address ready. This signal indicates that the slave is ready to accept an address and associated control signals.
AXI Write Data Channel Signals			
S_AXI_WDATA[C_S_AXI-_ DATA_WIDTH-1:0]	I	-	Write data.
S_AXI_WSTRB[C_S_AXI-_DATA_WIDTH/8-1:0]	I	-	Write strobes. This signal indicates which byte lanes to update in memory.
S_AXI_WVALID	I	-	Write valid. This signal indicates that valid write data and strobes are available.
S_AXI_WREADY	O	0	Write ready. This signal indicates that the slave can accept the write data.
AXI Write Response Channel Signals			
S_AXI_BRESP[1:0]	O	0	Write response. This signal indicates the status of the write transaction: "00" = OKEY, "10" = SLVERR
S_AXI_BVALID	O	0	Write response. This signal indicates the a valid write response is available.
S_AXI_BREADY	I	-	Response ready. This signal indicates that the master can accept the response information.
AXI Read Address Channel Signals			
S_AXI_ARADDR[C_S_A-XI_ ADDR_WIDTH-1:0]	I	-	Read address. The read address bus gives the address of a read transaction.
S_AXI_ARPROT[2:0]	I	-	AXI read address protection signal. "000" value is recommended. Infrastructure IP passes Protection bits across a system.

S_AXI_ARVALID	I	-	Read address valid. When High, this signal indicates that the read address and control information is valid and remains stable until the address acknowledgement signal, S_AXI_ARREADY, is High.
S_AXI_ARREADY	O	0	Read address ready. This signal indicates that the slave is ready to accept an address and associated control signals.
AXI Read Data Channel Signals			
S_AXI_RDATA[C_S_AXI-_DATA_WIDTH-1:0]	O	0	Read data.
S_AXI_RRESP[1:0]	O	0	Read response. This signal indicates the status of the read transfer.
S_AXI_RVALID	O	0	Read valid. This signal indicates that the required read data is available and the read transfer can complete.
S_AXI_RREADY	I	-	Read ready. This signal indicates that the master can accept the read data and response information.

In this table only one part of the AXI4-Lite interface signals is presented, relevant to our design. If you want to see the rest of the AXI4-Lite interface signals, please consult ***"LogiCORE IP AXI4-Lite IPIF"*** Product Guide for Vivado Design Suite. In this document you will find all the necessary information how to create a test bench file for Modulator module with AXI4-Lite interface.

Considering that we have four 32-bit registers in our design, our test bench task will be to change the content of these registers through AXI4-Lite interface and, by doing so, to change the frequency of the generated pwm signal.

On the Illustration 13.96 AXI4-Lite single write operation timing diagram is presented. Using to this diagram, we will create stimulus component in the test bench file for our design.

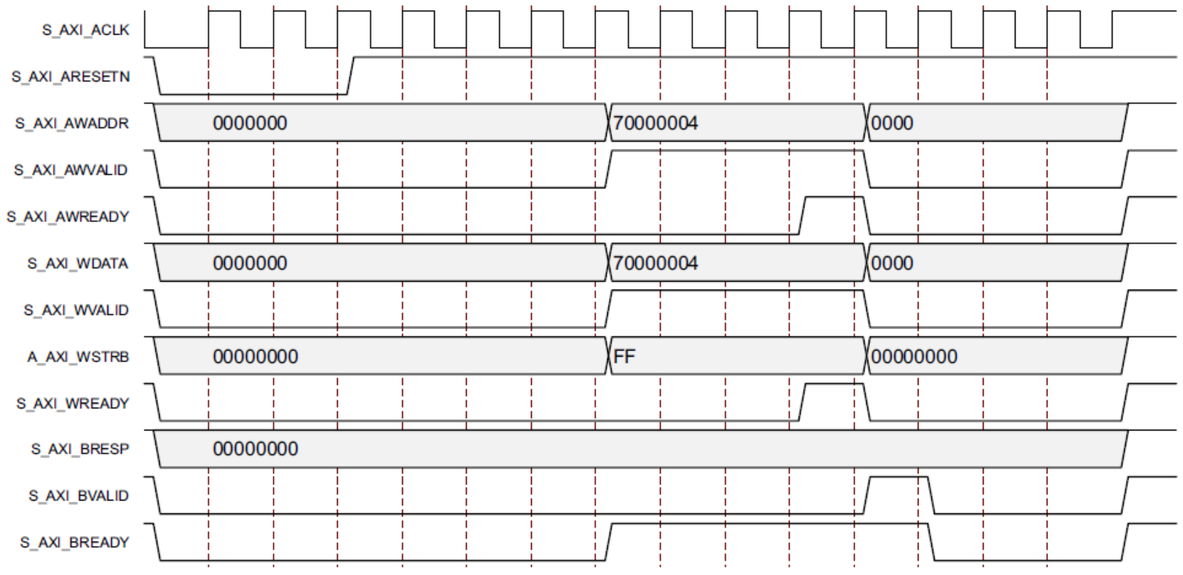


Figure 13.96: AXI4-Lite single write operation timing diagram

From the illustration above we can see that we must first generate AXI-Lite input clock signal (S_AXI_ACLK). After that, the important thing is to reset AXI4-Lite interface (by setting S_AXI_ARESETN signal to value '0'). In our case, reset will be 10 clock cycles wide. Considering that the reset is low-level sensitive, we will set it to '0' and wait for 10 falling edges of the AXI-Lite clock signal. After that, we will release the reset signal, setting it to '1'. From that moment, we will wait for the next falling edge of the AXI-Lite clock signal and write *div_factor_freqhigh* value (S_AXI_WDATA) in the appropriate register (2nd register, see Figure 13.70). To know what will be the address location of the "div_factor_freqhigh" register, we must first understand the structure of S_AXI_AWADDR signal.

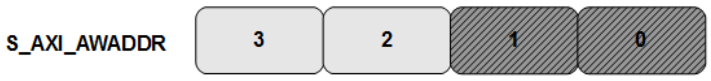


Figure 13.97: S_AXI_AWADDR signal

S_AXI_AWADDR is a 4-bit wide signal. AXI address space is byte addressable. Since we are using 32-bit registers, their addresses must be aligned on 32-bit word address boundaries. This means that values of two least significant bits (bits 0 and 1) of S_AXI_AWADDR signal are not relevant when we are addressing 32-bit registers and can have arbitrary values. On the other hand two most significant bits (bits 2 and 3) are used to select desired 32-bit register. In our case, internal 32-bit registers address map will have the following structure:

Table 12.2: Internal Registers Address Map of the Modulator IP Core

Internal Register Name	S_AXI_AWADDR Value
"sw0" register	"0000" (0)
"div_factor_freqhigh" register	"0100" (4)
"div_facator_freqlow" register	"1000" (8)
"4. unused" register	"1100" (12)

Now when we know the structure of the internal registers address space, we will assign "0100" value to the S_AXI_AWADDR signal since it is the address location of the "div_factor_freqhigh" register. We should also validate this address (by setting S_AXI_AWVALID signal to '1') and write desired *div_factor_freqhigh* value in the "div_factor_freqhigh" register (by setting S_AXI_WDATA to appropriate value). After that we should validate that the write data is valid (setting S_AXI_WVALID to '1') and that all four bytes of write data should be written in the selected internal register (setting S_AXI_WSTRB to "1111"). When S_AXI_WSTRB = "1111" that means that we would like to write data using all four byte lanes. We should also activate S_AXI_BREADY signal, because this signal indicates that master can accept a write response. After the first data write, we will wait for S_AXI_AWREADY signal to be first '1' and then '0' after one clock cycle, and then we will deactivate AXI Write Address Channel and AXI Write Data Channel signals, completing one write transaction on the AXI bus. Next we will write *div_factor_freqlow* value in the "div_factor_freqlow" register by repeating the same procedure. At the end, we will repeat the same procedure once more, to write appropriate value to the "sw0" register.

The complete test bench file for Modulator IP core with AXI4 interface is shown below.

modulator_axi_ip_tb.vhd:

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

    use work.modulator_pkg.all;

entity modulator_axi_ip_tb is
end entity;


architecture tb of modulator_axi_ip_tb is

    -- AXI Write Address Channel Signals
    signal s00_axi_awaddr_s  : std_logic_vector(3 downto 0) := (others=>'0');
    signal s00_axi_awprot_s  : std_logic_vector(2 downto 0) := (others=>'0');
    signal s00_axi_awvalid_s : std_logic := '0';
    signal s00_axi_awready_s : std_logic;
    -- AXI Write Data Channel Signals
    signal s00_axi_wdata_s   : std_logic_vector(31 downto 0):= (others=>'0');
    signal s00_axi_wstrb_s   : std_logic_vector(3 downto 0) := (others=>'0');
    signal s00_axi_wvalid_s  : std_logic := '0';
    signal s00_axi_wready_s  : std_logic;
    -- AXI Write Response Channel Signals
    signal s00_axi_bresp_s   : std_logic_vector(1 downto 0);
    signal s00_axi_bvalid_s  : std_logic;
    signal s00_axi_bready_s  : std_logic := '0';
    -- AXI Read Address Channel Signals
    signal s00_axi_araddr_s  : std_logic_vector(3 downto 0) := (others=>'0');
    signal s00_axi_arprot_s  : std_logic_vector(2 downto 0) := (others=>'0');
    signal s00_axi_arvalid_s : std_logic := '0';
    signal s00_axi_arready_s : std_logic;
    -- AXI Read Data Channel Signals
    signal s00_axi_rdata_s   : std_logic_vector(31 downto 0);
    signal s00_axi_rresp_s   : std_logic_vector(1 downto 0);
    signal s00_axi_rvalid_s  : std_logic;
    signal s00_axi_rready_s  : std_logic := '0';
    -- AXI Global System Signals
    signal s00_axi_aclk_s    : std_logic := '0';
    signal s00_axi_aresetn_s : std_logic := '1';

    -- pulse width modulated signal
    signal pwm_out_s         : std_logic;

    -- 100 MHz
    constant clock_frequency_c : real := 100000000.0;

    -- period of AXI-lite input clock signal
    constant clock_period_c : time := 1000000000.0 / clock_frequency_c * 1ns;

    -- constant created to short the duration of the simulation process 10 times
    constant design_setting1_c : design_setting_t_rec := (255, 10.0, 35.0, 8, 12);

    -- c1_c = fclk/(2^depth*2^width)                   - c1_c = 95.3674, fclk = 100 MHz
    constant c1_c : real := clock_frequency_c/(real((2**design_setting1_c.depth)*(2**design_setting1_c.
      width)));
     -- div_factor_freqhigh_c = (c1_c/f_high)*2^width  - threshold value of frequency a = 110592
    constant div_factor_freqhigh_c : integer := integer(c1_c/design_setting1_c.f_high)*(2**
      design_setting1_c.width);
    -- div_factor_freqlow_c  = (c1_c/f_low)*2^width    - threshold value of frequency b = 389120
    constant div_factor_freqlow_c  : integer := integer(c1_c/design_setting1_c.f_low)*(2**design_setting1_c
      .width);

begin

    -- modulator_axi_ip IP instance
    axi: entity work.modulator_axi_ip_0
        port map(
            s00_axi_awaddr  => s00_axi_awaddr_s,
            s00_axi_awprot  => s00_axi_awprot_s,
            s00_axi_awvalid => s00_axi_awvalid_s,
            s00_axi_awready => s00_axi_awready_s,
            s00_axi_wdata   => s00_axi_wdata_s,
            s00_axi_wstrb   => s00_axi_wstrb_s,
            s00_axi_wvalid  => s00_axi_wvalid_s,
            s00_axi_wready  => s00_axi_wready_s,
            s00_axi_bresp   => s00_axi_bresp_s,
            s00_axi_bvalid  => s00_axi_bvalid_s,
            s00_axi_bready  => s00_axi_bready_s,
            s00_axi_araddr  => s00_axi_araddr_s,
            s00_axi_arprot  => s00_axi_arprot_s,
            s00_axi_arvalid => s00_axi_arvalid_s,
            s00_axi_arready => s00_axi_arready_s,
            s00_axi_rdata   => s00_axi_rdata_s,
            s00_axi_rresp   => s00_axi_rresp_s,
            s00_axi_rvalid  => s00_axi_rvalid_s,
            s00_axi_rready  => s00_axi_rready_s,
            s00_axi_aclk    => s00_axi_aclk_s,
            s00_axi_aresetn => s00_axi_aresetn_s,
            pwm_out         => pwm_out_s
        );

    -- generates AXI-lite input clock signal
    s00_axi_aclk_s  <= not (s00_axi_aclk_s) after clock_period_c/2;

    stimulus_generator_p : process
    begin
        -- reset AXI-lite interface. Reset will be 10 clock cycles wide
        s00_axi_aresetn_s <= '0';
        -- wait for 10 falling edges of AXI-lite clock signal
        for i in 1 to 10 loop
            wait until falling_edge(s00_axi_aclk_s);
        end loop;
        -- release reset
        s00_axi_aresetn_s <= '1';
        wait until falling_edge(s00_axi_aclk_s);

        -- write div_factor_freqhigh value into appropriate register
        s00_axi_awaddr_s <= "0100";
        s00_axi_awvalid_s <= '1';
        s00_axi_wdata_s <= conv_std_logic_vector(div_factor_freqhigh_c, 32);
        s00_axi_wvalid_s <= '1';
        s00_axi_wstrb_s <= "1111";
        s00_axi_bready_s <= '1';
        wait until s00_axi_awready_s = '1';
        wait until s00_axi_awready_s = '0';
        wait until falling_edge(s00_axi_aclk_s);
        s00_axi_awaddr_s <= "0000";
        s00_axi_awvalid_s <= '0';
        s00_axi_wdata_s <= conv_std_logic_vector(0, 32);
        s00_axi_wvalid_s <= '0';
        s00_axi_wstrb_s <= "0000";
        wait until s00_axi_bvalid_s = '0';
        wait until falling_edge(s00_axi_aclk_s);
        s00_axi_bready_s <= '0';
        wait until falling_edge(s00_axi_aclk_s);


        -- write div_factor_freqlow value into appropriate register
        s00_axi_awaddr_s <= "1000";
        s00_axi_awvalid_s <= '1';
        s00_axi_wdata_s <= conv_std_logic_vector(div_factor_freqlow_c, 32);
        s00_axi_wvalid_s <= '1';
        s00_axi_wstrb_s <= "1111";
        s00_axi_bready_s <= '1';
        wait until s00_axi_awready_s = '1';
        wait until s00_axi_awready_s = '0';
        wait until falling_edge(s00_axi_aclk_s);
        s00_axi_awaddr_s <= "0000";
        s00_axi_awvalid_s <= '0';
        s00_axi_wdata_s <= conv_std_logic_vector(0, 32);
        s00_axi_wvalid_s <= '0';
        s00_axi_wstrb_s <= "0000";
        wait until s00_axi_bvalid_s = '0';
        wait until falling_edge(s00_axi_aclk_s);
        s00_axi_bready_s <= '0';
        wait until falling_edge(s00_axi_aclk_s);

        -- we are waiting for one period of pwm signal when sw0=0
        wait for 100 ms;

        -- write value sw0=1 into appropriate register
        s00_axi_awaddr_s <= "0000";
        s00_axi_awvalid_s <= '1';
        s00_axi_wdata_s <= conv_std_logic_vector(1, 32);
        s00_axi_wvalid_s <= '1';
        s00_axi_wstrb_s <= "1111";
        s00_axi_bready_s <= '1';
        wait until s00_axi_awready_s = '1';
        wait until s00_axi_awready_s = '0';
        wait until falling_edge(s00_axi_aclk_s);
        s00_axi_awaddr_s <= "0000";
        s00_axi_awvalid_s <= '0';
        s00_axi_wdata_s <= conv_std_logic_vector(0, 32);
        s00_axi_wvalid_s <= '0';
        s00_axi_wstrb_s <= "0000";
        wait until s00_axi_bvalid_s = '0';
        wait until falling_edge(s00_axi_aclk_s);
        s00_axi_bready_s <= '0';
        wait until falling_edge(s00_axi_aclk_s);

        wait;
    end process;

end;
```

After you have entered the code for the input stimulus in order to perform simulation, follow the next steps:

Step 1. In the **Sources** window, under the **Simulation Sources** / **sim_1**, select **modulator_axi_ip_tb - tb** file

Step 2. In the **Flow Navigator**, under the **Simulation**, click on the **Run Simulation** button

Step 3. Choose the only offered **Run Behavioral Simulation** option, see Illustration 13.98, and your simulation will start

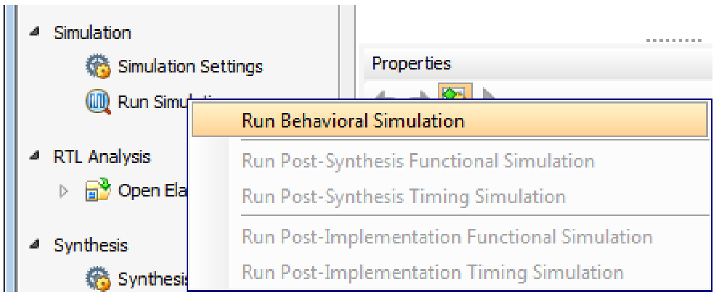

Simulation
Simulation Settings
Run Simulation
RTL Analysis
Open Ela
Synthesis
Synthesis
Properties
Run Behavioral Simulation
Run Post-Synthesis Functional Simulation
Run Post-Synthesis Timing Simulation
Run Post-Implementation Functional Simulation
Run Post-Implementation Timing Simulation

Figure 13.98: Run Behavioral Simulation option

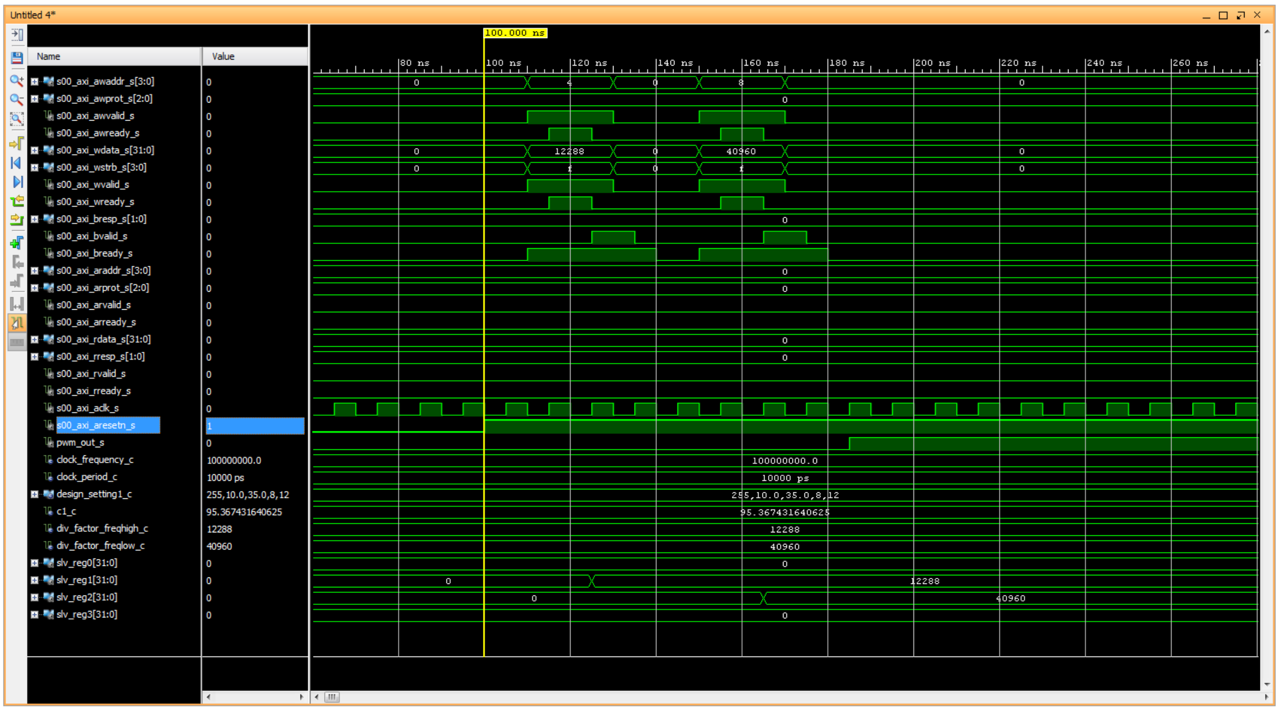
Step 4. The tool will compile the test bench file and launch the Vivado simulator

Step 5. In the Vivado simulator, open **Scopes** window and expand **modulator_axi_ip_tb** -> **axi** -> **U0** design units and select **modulator_axi_ip_v1_0_S00_AXI_inst** design unit

Step 6. In the Vivado **Objects** window select our four registers **slv_reg0[31:0]**, **slv_reg1[31:0]**, **slv_reg2[31:0]** and **slv_-reg3[31:0]** and move them to waveform window

Step 7. Simulate your design for **120 ms**

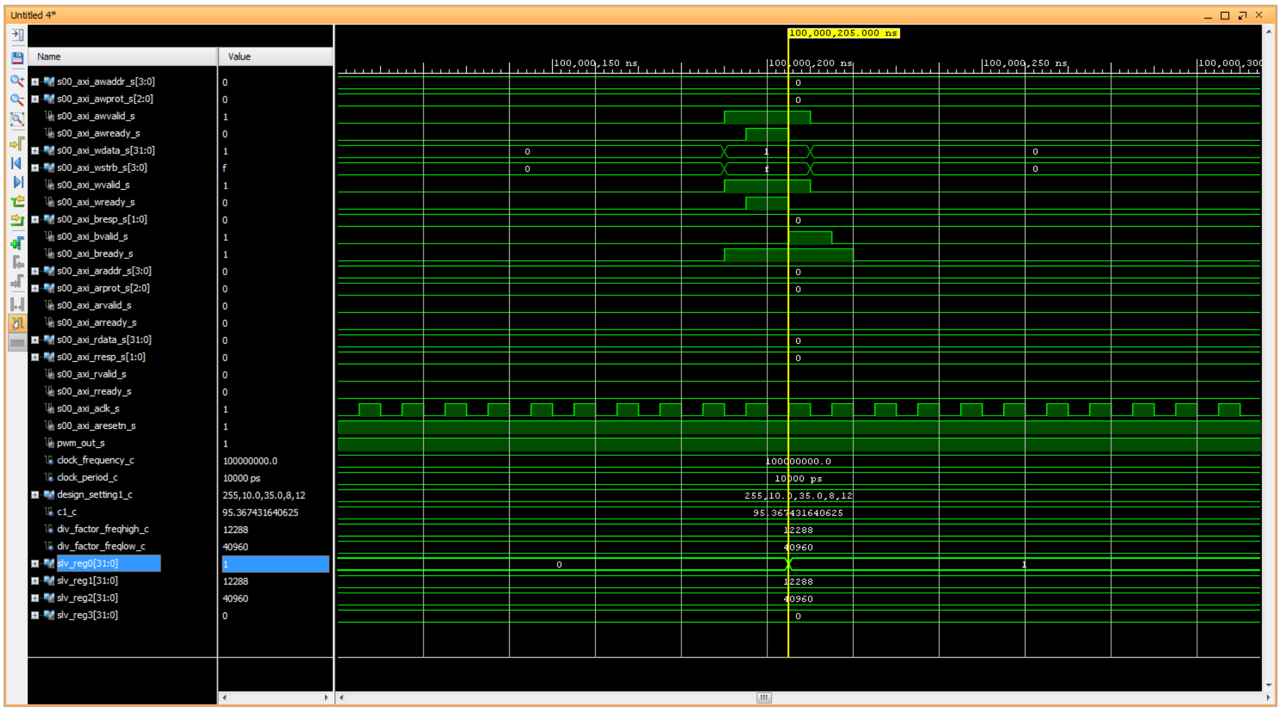
Step 8. Go to the beginning of the simulation result, zoom out few times and find the moment where ***s00_axi_aresetn_s*** signal is changing from **0** to **1**. Your simulation results should look like as it is shown on the Illustration 13.99. From the simulation results we can see that our system works as we predicted.



Name	Value
s00_axi_awaddr_s[3:0]	0
s00_axi_awprot_s[2:0]	0
s00_axi_awvalid_s	0
s00_axi_awready_s	0
s00_axi_wdata_s[31:0]	0
s00_axi_wstrb_s[3:0]	0
s00_axi_wvalid_s	0
s00_axi_wready_s	0
s00_axi_bresp_s[1:0]	0
s00_axi_bvalid_s	0
s00_axi_bready_s	0
s00_axi_araddr_s[3:0]	0
s00_axi_arprot_s[2:0]	0
s00_axi_arvalid_s	0
s00_axi_arready_s	0
s00_axi_rdata_s[31:0]	0
s00_axi_rresp_s[1:0]	0
s00_axi_rvalid_s	0
s00_axi_rready_s	0
s00_axi_aclk_s	0
s00_axi_aresetn_s	1
pwm_out_s	0
clock_frequency_c	100000000.0
clock_period_c	10000 ps
design_setting1_c	255,10.0,35.0,8,12
c1_c	95.367431640625
div_factor_freqhigh_c	12288
div_factor_freqlow_c	40960
slv_reg0[31:0]	0
slv_reg1[31:0]	0
slv_reg2[31:0]	0
slv_reg3[31:0]	0

Figure 13.99: Simulation results - writing to div_factor_freqhigh and div_factor_freqlow registers

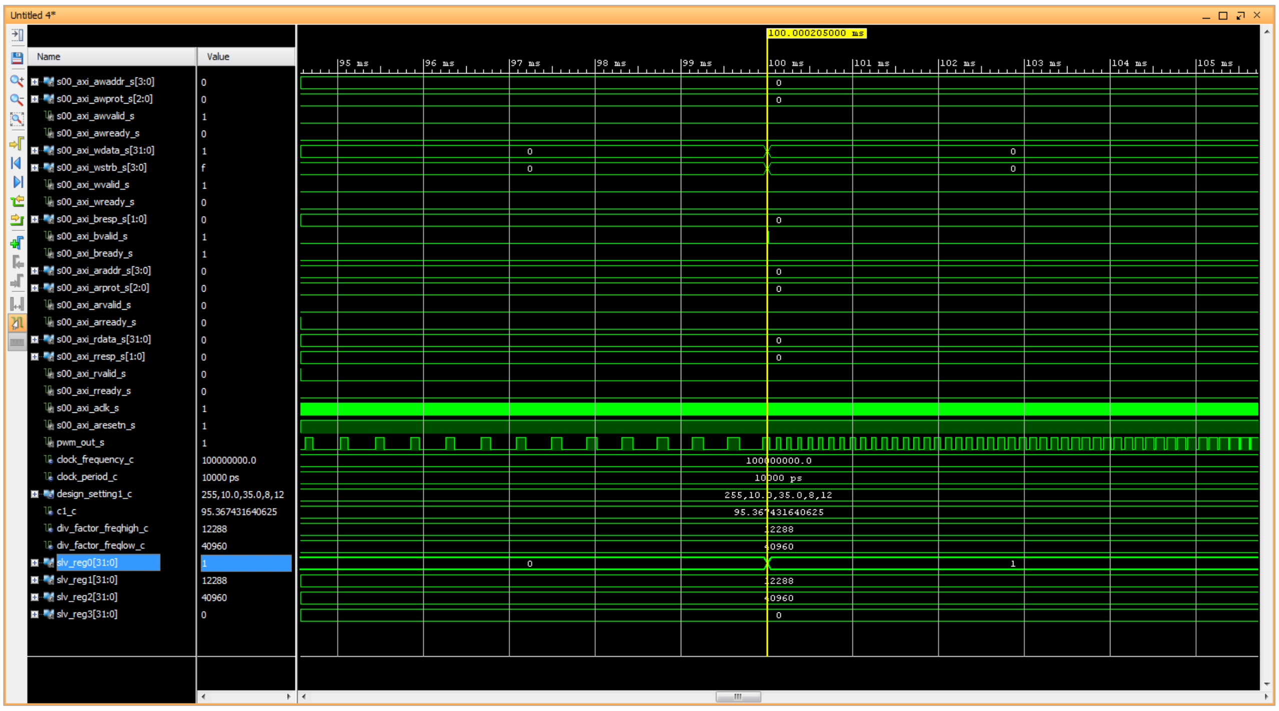
Step 9. Zoom fit and then zoom in few times around 100 ms and you will see the "sw0" register change, see Illustration 13.100



Name	Value
s00_axi_awaddr_s[3:0]	0
s00_axi_awprot_s[2:0]	0
s00_axi_awvalid_s	1
s00_axi_awready_s	0
s00_axi_wdata_s[31:0]	1
s00_axi_wstrb_s[3:0]	f
s00_axi_wvalid_s	1
s00_axi_wready_s	0
s00_axi_bresp_s[1:0]	0
s00_axi_bvalid_s	1
s00_axi_bready_s	1
s00_axi_araddr_s[3:0]	0
s00_axi_arprot_s[2:0]	0
s00_axi_arvalid_s	0
s00_axi_arready_s	0
s00_axi_rdata_s[31:0]	0
s00_axi_rresp_s[1:0]	0
s00_axi_rvalid_s	0
s00_axi_rready_s	0
s00_axi_aclk_s	1
s00_axi_aresetn_s	1
pwm_out_s	1
clock_frequency_c	100000000.0
clock_period_c	10000 ps
design_setting1_c	255,10.0,35.0,8,12
c1_c	95.367431640625
div_factor_freqhigh_c	12288
div_factor_freqlow_c	40960
slv_reg0[31:0]	1
slv_reg1[31:0]	12288
slv_reg2[31:0]	40960
slv_reg3[31:0]	0

Figure 13.100: Simulation Results - changing the value of sw0 register

Step 10. If you zoom out a few times more, you can also see the pwm frequency change, when sw0=0 and when sw0=1, see Illustration 13.101



Name	Value
s00_axi_awaddr_s[3:0]	0
s00_axi_awprot_s[2:0]	0
s00_axi_awvalid_s	1
s00_axi_awready_s	0
s00_axi_wdata_s[31:0]	1
s00_axi_wstrb_s[3:0]	f
s00_axi_wvalid_s	1
s00_axi_wready_s	0
s00_axi_bresp_s[1:0]	0
s00_axi_bvalid_s	1
s00_axi_bready_s	1
s00_axi_araddr_s[3:0]	0
s00_axi_arprot_s[2:0]	0
s00_axi_arvalid_s	0
s00_axi_arready_s	0
s00_axi_rdata_s[31:0]	0
s00_axi_rresp_s[1:0]	0
s00_axi_rvalid_s	0
s00_axi_rready_s	0
s00_axi_aclk_s	1
s00_axi_aresetn_s	1
pwm_out_s	1
clock_frequency_c	100000000.0
clock_period_c	10000 ps
design_setting1_c	255,10.0,35.0,8,12
c1_c	95.367431640625
div_factor_freqhigh_c	12288
div_factor_freqlow_c	40960
slv_reg0[31:0]	1
slv_reg1[31:0]	12288
slv_reg2[31:0]	40960
slv_reg3[31:0]	0

Figure 13.101: Simulation Results - pwm signal frequency change as a result of the change of the sw0 register value

Chapter 14

APPENDIX

14.1 HDL Instantiation Debug Probing Flow

Vivado Logic Analyzer is a integrated logic analyzer. In this chapter you will learn how to debug your FPGA design by inserting an Integrated Logic Analyzer (ILA) core and Virtual Input/Output (VIO) core using the Vivado IDE. You will take advantage of integrated Vivado logic analyzer functions to debug and discover some potential root causes of your design.

This chapter will illustrate overall integration flows between Vivado logic analyzer, ILA 6.2, VIO 3.0 and Vivado IDE. There are two flows (methods) supported in the Vivado Debug Probing:

1. HDL Instantiation Debug Probing Flow

2. Using the Netlist Insertion Debug Probing Flow

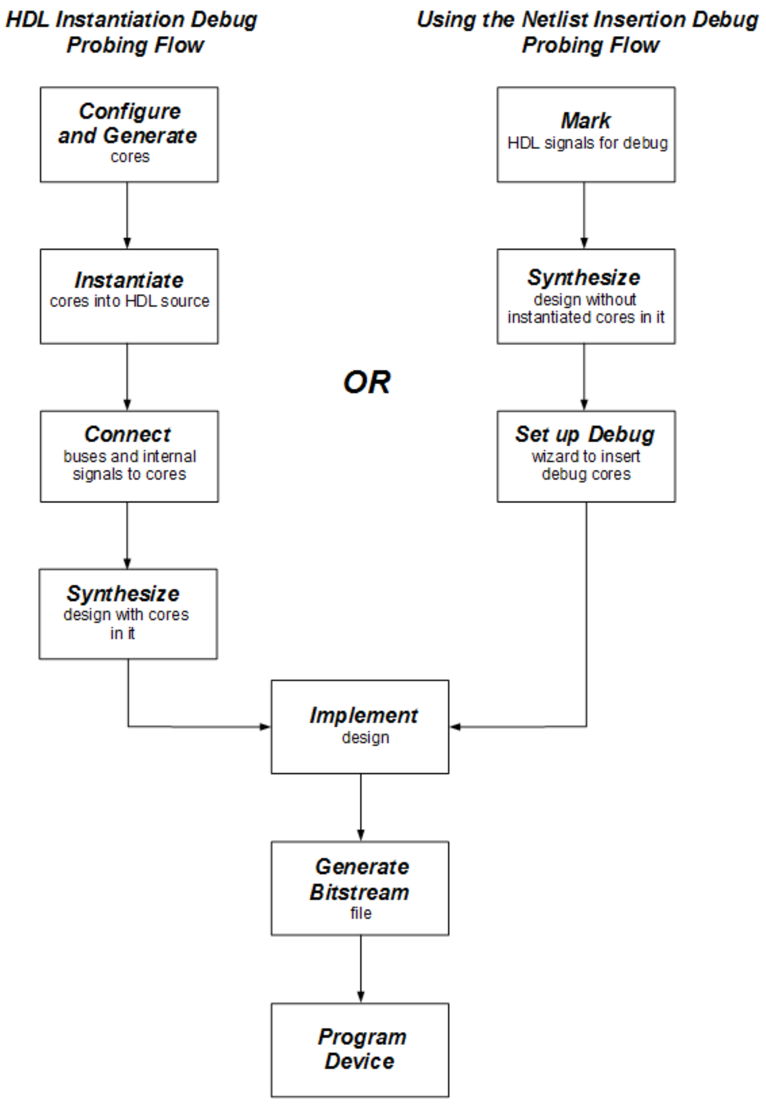

HDL Instantiation Debug Probing Flow
Configure and Generate cores
Instantiate cores into HDL source
Connect buses and internal signals to cores
Synthesize design with cores in it
Using the Netlist Insertion Debug Probing Flow
Mark HDL signals for debug
Synthesize design without instantiated cores in it
Set up Debug wizard to insert debug cores
OR
Implement design
Generate Bitstream file
Program Device

Figure 14.1: Vivado Logic Analyzer Design Flow

As we already said, the HDL instantiation flow is one of the two flows supported in the Vivado Debug Probing. The HDL instantiation debug probing flow involves the manual customization, instantiation, and connection of various debug core components directly in the HDL design source. Debug cores that are supported in this flow, in the Vivado tool, are:

- Integrated Logic Analyzer (ILA) core v6.2
- Virtual Input/Output (VIO) core v3.0
- Integrated Bit Error Ratio Tester (IBERT) core v3.0
- JTAG to AXI Master core v1.1

LogiCORE IP Integrated Logic Analyzer (ILA) v6.2 core

The LogiCORE IP Integrated Logic Analyzer (ILA) core is a customizable logic analyzer core that can be used to monitor the internal signals of a design. The ILA core includes many advanced features of modern logic analyzers, including boolean trigger equations, and edge transition triggers. Because the ILA core is synchronous to the design being monitored, all design clock constraints that are applied to your design are also applied to the components of the ILA core.

ILA core general features are:

- user-selectable number of probe ports and probe_width
- multiple probe ports, which can be combined into a single trigger condition

- AXI interface on ILA IP core to debug AXI IP cores in a system

The following illustration is a symbol of the ILA v6.2 core.

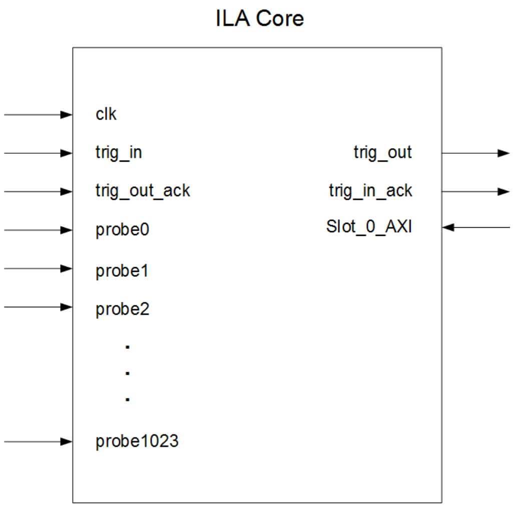


Figure 14.2: Symbol of the ILA v6.2 core

Signals in the FPGA design are connected to ILA core clock and probe inputs. These signals, attached to the probe inputs, are sampled at design speed and stored using on-chip block RAM (BRAM). The core parameters specify the number of probes, trace sample depth, and the width for each probe input. Communication with the ILA core is conducted using an auto-instantiated debug core hub that connects to the JTAG interface of the FPGA.

Note: If you want to read and learn more about the ILA v6.2 core, please refer to *"LogiCORE IP Integrated Logic Analyzer (ILA) v6.2 Product Guide".*

LogiCORE IP Virtual Input/Output (VIO) v3.0 core

The LogiCORE IP Virtual Input/Output (VIO) core is a customizable core that can both monitor and drive internal FPGA signals in real time. The number of width of the input and output ports are customizable in size to interface with the FPGA design. Because the VIO core is synchronous to the design being monitored and/or driven, all design clock constraints that are applied to your design are also applied to the components inside the VIO core. Run time interaction with this core requires the use of the Vivado logic analyzer feature. Unlike the ILA core, no on-chip or off-chip RAM is required.

VIO core general features are:

- provides virtual LEDs and other status indicators through input ports

- includes optional activity detectors on input ports to detect rising and falling transitions between samples

- provides virtual buttons and other controls indicators through output ports

- includes custom output initialization that allows you to specify the value of the VIO core outputs immediately following device configuration and start-up

- run time reset of the VIO core to initial values

The following illustration is a symbol of the VIO v3.0 core.

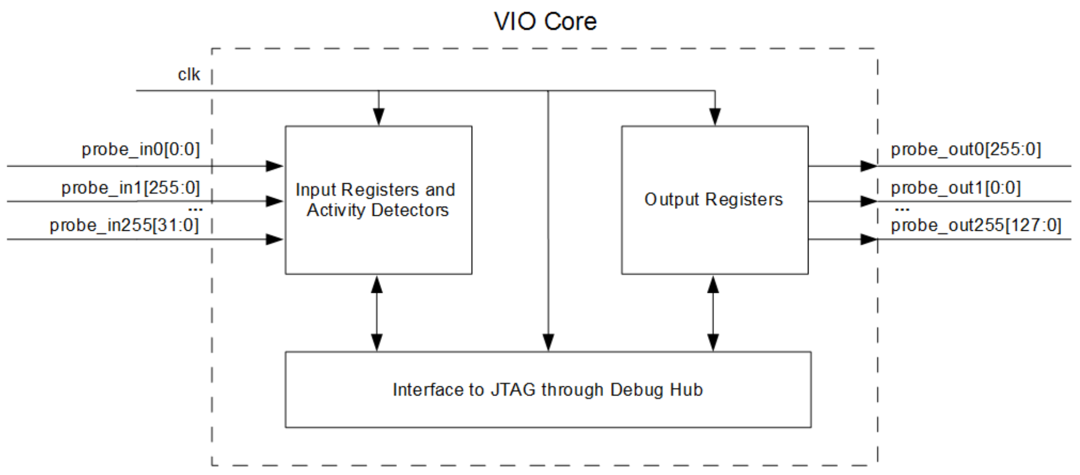


Figure 14.3: Symbol of the VIO v3.0 core

Note: If you want to read and learn more about the VIO v3.0 core, please refer to *"LogiCORE IP Virtual Input/Output (VIO) v3.0 Product Guide".*

LogiCORE IP Integrated Bit Error Ratio Tester (IBERT) for 7 Series GTX Transceivers v3.0 core

The customizable LogiCORE IP Integrated Bit Error Ratio Tester (IBERT) core for 7 Series FPGA GTX transceivers is designed for evaluating and monitoring the GTX transceivers. This core includes pattern generators and checkers that are implemented in FPGA logic, and access to ports and the dynamic reconfiguration port attributes of the GTX transceivers. Communication logic is also included to allow the design to be run time accessible through JTAG.

IBERT core general features are:

- provides a communication path between the Vivado serial I/O analyzer feature and the IBERT core
- provides a user-selectable number of 7 series FPGA GTX transceivers
- transceivers can be customized for the desired line rate, reference clock rate, reference clock source, and data path width
- requires a system clock that can be sources from a pin or one of the enabled GTX transceivers

Note: If you want to read and learn more about the IBERT v3.0 core, please refer to *"LogiCORE IP Integrated Bit Error Ratio Tester (IBERT) for 7 Series GTX Transceivers v3.0 Product Guide".*

LogiCORE IP JTAG to AXI Master v1.1 core

The LogiCORE JTAG to AXI Master IP core is a customizable core that can generate the AXI transactions and drive the AXI signals internal to FPGA in the system. The AXI bus interface protocol can be selected using a parameter in the IP customization Vivado IDE. The width of AXI data bus is customizable. This IP can drive AXI4-Lite or AXI4 Memory Mapped Slave through an AXI4 interconnect. Run time interaction with this core requires the use of the Vivado logic analyzer feature.

JTAG to AXI Master core general features are:

- provides AXI4 master interface
- option to set AXI4 and AXI4-Lite interfaces
- user selectable AXI data width - 32 to 64
- user selectable AXI ID width up to four bits
- Vivado logic analyzer Tcl Console interface to interact with hardware
- support AXI4 and Lite transactions

The following illustration shows an AXI system that uses the JTAG to AXI Master core as an AXI Master.

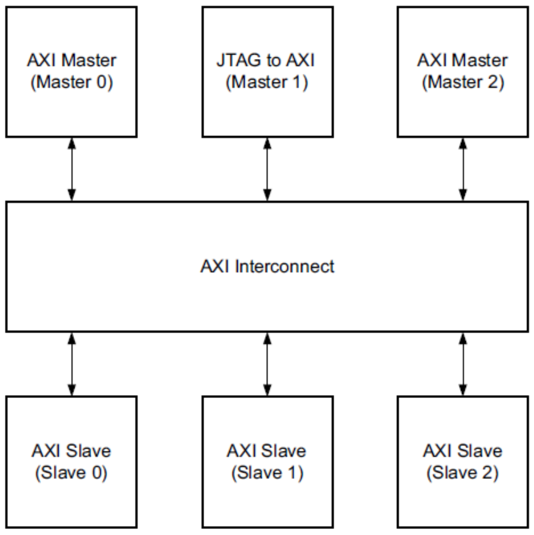


Figure 14.4: JTAG to AXI Master System

The JTAG to AXI Master core can communicate to all the downstream slaves and can coexist with the other AXI Master in this system.

Note: If you want to read and learn more about the JTAG to AXI Master v1.1 core, please refer to *"LogiCORE IP JTAG to AXI Master v1.1 Product Guide"*.

***Important*: The *IBERT IP core* and *JTAG to AXI Master IP core* won't be used in this tutorial!**

Using the HDL Instantiation Debug Probing Flow, you will generate an ILA v6.2 and VIO v3.0 IP cores using the Vivado IP Catalog and instantiate the core in a design manually as you would with any other IP core.

Step 1. Before you start ILA and VIO core generation, you must first create a new project (***modulator_ila_vio***) for **Zed-Board Zynq Evaluation and Development Kit** board

Step 2. Add Modulator design source files into the project (*frequency_trigger_rtl.vhd*, *counter_rtl.vhd*, *modulator_pkg.vhd*, *sine_rtl.vhd*, *sine_top_rtl.vhd*, *pwm_rtl.vhd*, *modulator_rtl.vhd*, *modulator_wrapper_rtl.vhd*) using **Add Sources** command from the **Flow Navigator**

Step 3. Select ***modulator_wrapper_rtl.vhd*** source file, right-click on it and select **Set as Top** option

ILA Core Generation

To configure and generate the ILA core, use the following steps:

Step 1. In the Vivado **Flow Navigator**, under the **Project Manager** , click the **IP Catalog** command, see Illustration 14.5

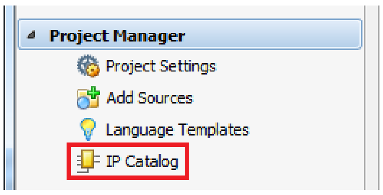
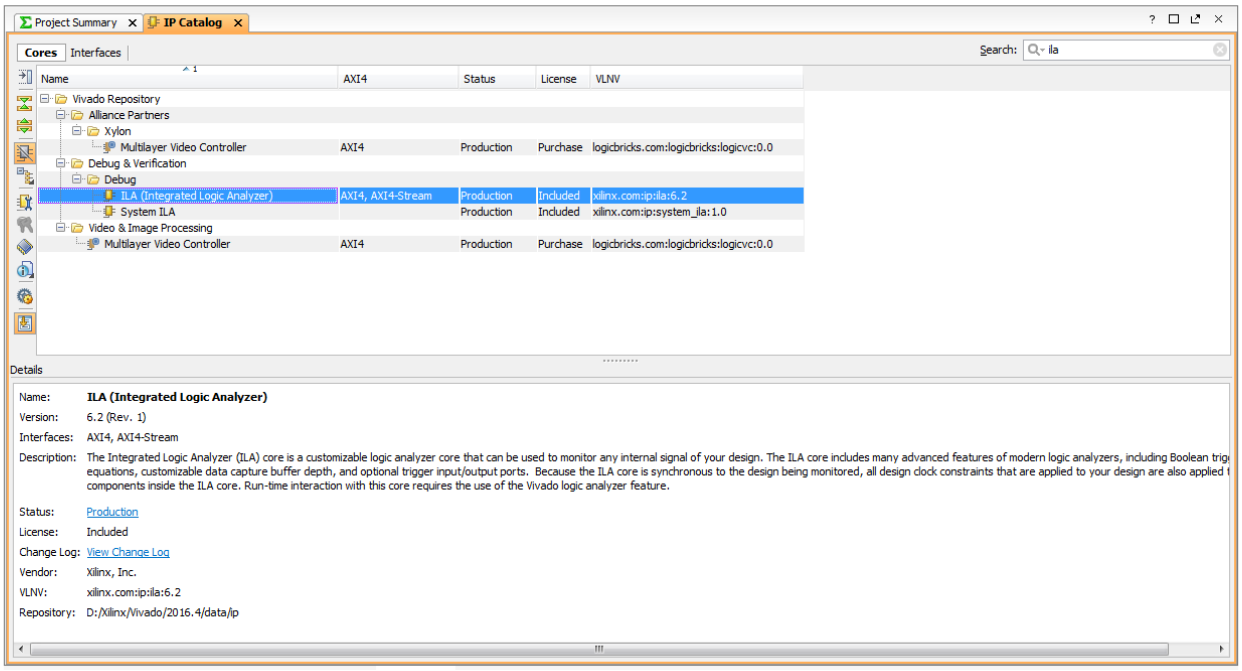


Figure 14.5: IP Catalog command

Step 2. In the **IP Catalog** window, in the **Search** field, search for the **ILA (Integrated Logic Analyzer)** IP core. After you selected the ILA core, in the **Details** window, under the main IP Catalog window, you will find all the necessary information about the selected IP core, see Illustration 14.6



Name	AXI4	Status	License	VLNV
Vivado Repository				
Alliance Partners				
Xylon				
Multilayer Video Controller	AXI4	Production	Purchase	logicbricks.com:logicbricks:logicvc:0.0
Debug & Verification				
Debug				
ILA (Integrated Logic Analyzer)	AXI4, AXI4-Stream	Production	Included	xilinx.com:ip:ila:6.2
System ILA		Production	Included	xilinx.com:ip:system_ila:1.0
Video & Image Processing				
Multilayer Video Controller	AXI4	Production	Purchase	logicbricks.com:logicbricks:logicvc:0.0

Search: ila

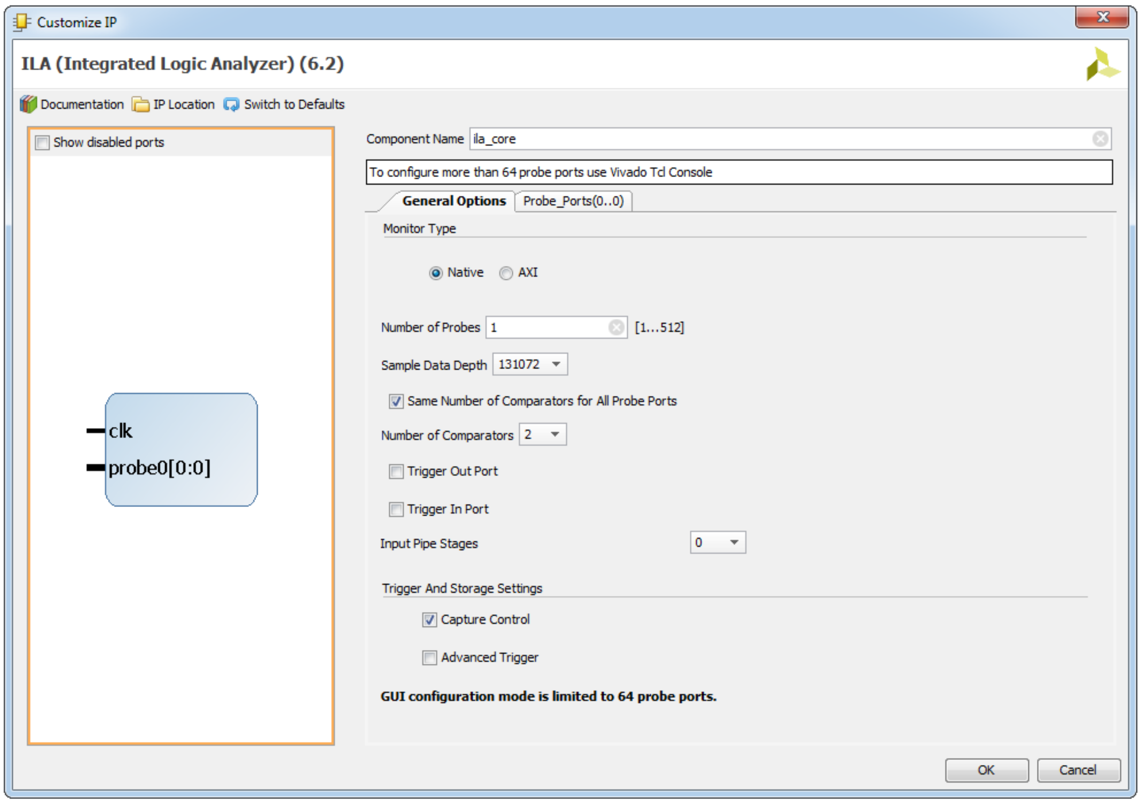
Details

Name:	**ILA (Integrated Logic Analyzer)**
Version:	6.2 (Rev. 1)
Interfaces:	AXI4, AXI4-Stream
Description:	The Integrated Logic Analyzer (ILA) core is a customizable logic analyzer core that can be used to monitor any internal signal of your design. The ILA core includes many advanced features of modern logic analyzers, including Boolean trig equations, customizable data capture buffer depth, and optional trigger input/output ports. Because the ILA core is synchronous to the design being monitored, all design clock constraints that are applied to your design are also applied t components inside the ILA core. Run-time interaction with this core requires the use of the Vivado logic analyzer feature.
Status:	Production
License:	Included
Change Log:	View Change Log
Vendor:	Xilinx, Inc.
VLNV:	xilinx.com:ip:ila:6.2
Repository:	D:/Xilinx/Vivado/2016.4/data/ip

Figure 14.6: IP Catalog window with selected ILA core

Step 3. Double-click on the **ILA (Integrated Logic Analyzer)** IP core and Vivado IDE will create a new skeleton source for your ILA core

The window that will be opened is used to set up the general ILA core parameters, see Illustration 14.7



Customize IP

ILA (Integrated Logic Analyzer) (6.2)

Component Name: ila_core

To configure more than 64 probe ports use Vivado Tcl Console

General Options

Monitor Type: [✓ Native] AXI

Number of Probes: 1 [1...512]

Sample Data Depth: 131072

[✓] Same Number of Comparators for All Probe Ports

Number of Comparators: 2

[ ] Trigger Out Port

[ ] Trigger In Port

Input Pipe Stages: 0

Trigger And Storage Settings

[✓] Capture Control

[ ] Advanced Trigger

GUI configuration mode is limited to 64 probe ports.

Figure 14.7: ILA core configuration window - General Options tab

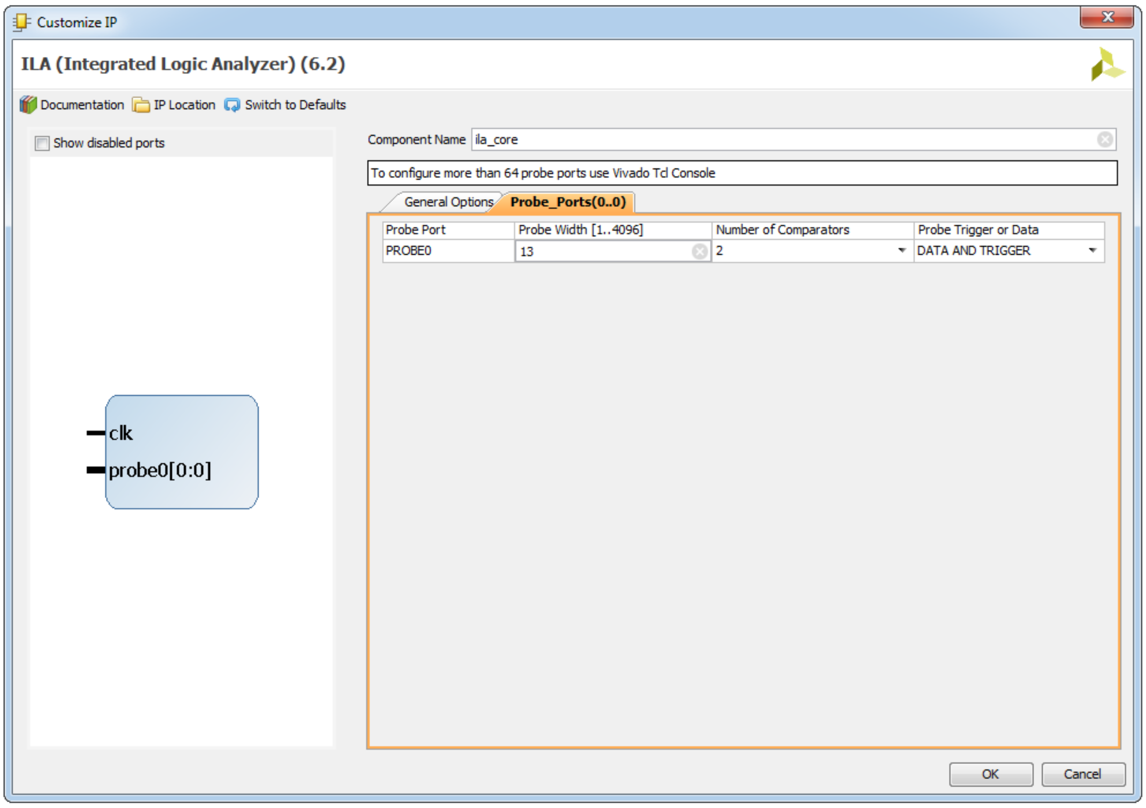
Step 4. In the **ILA (Integrated Logic Analyzer) (6.2)** window, enter *ila_core_name* (**ila_core**) in the **Component Name**

field

Step 5. In the **General Options** tab, select **Native** Monitor Type, choose maximum value for **Sample Data Depth (131072)**, enable **Capture Control** option and leave all the other parameters unchanged, see Illustration 14.7

Step 6. Select **Probe_Ports(0..0)** tab and change the **Probe Width [1..4096]** of the **PROBE0** probe port from **1** to **13**, see Illustration 14.8

We configured the probe width of the PROBE0 probe port to 13, because the width of the **sine_ampl_s** signal, that we want to see in the Vivado Logic Analyzer, is 12 bits and the width of the **freq_trig_s** signal is 1 bit.



Component Name: ila_core

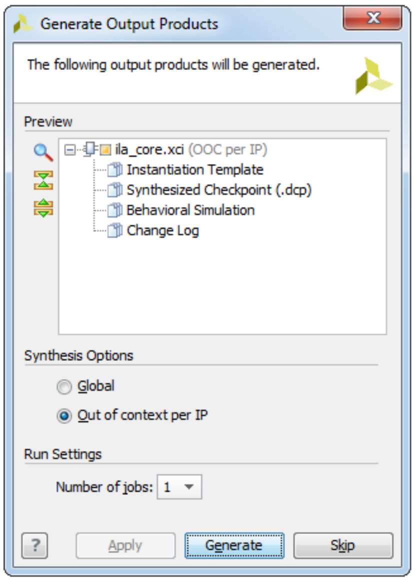
To configure more than 64 probe ports use Vivado Tcl Console

Probe Port	Probe Width [1..4096]	Number of Comparators	Probe Trigger or Data
PROBE0	13	2	DATA AND TRIGGER

Figure 14.8: ILA core configuration window - Probe_Ports(0..0) tab

Step 7. Click **OK**

Step 8. In the **Generate Output Products** window click **Generate**, see Illustration 14.9


Generate Output Products

The following output products will be generated.

Preview

- ila_core.xci (OOC per IP)
  - Instantiation Template
  - Synthesized Checkpoint (.dcp)
  - Behavioral Simulation
  - Change Log

Synthesis Options

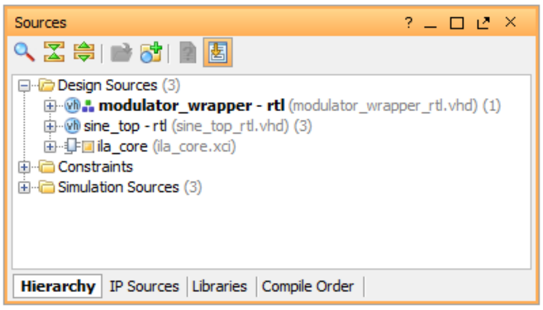
Global [✓ Out of context per IP]

Run Settings

Number of jobs: 1

Figure 14.9: Generate Output Products window for ILA core

Note: After ILA core generation, your ILA core should appear in the Sources window, see Illustration 14.10

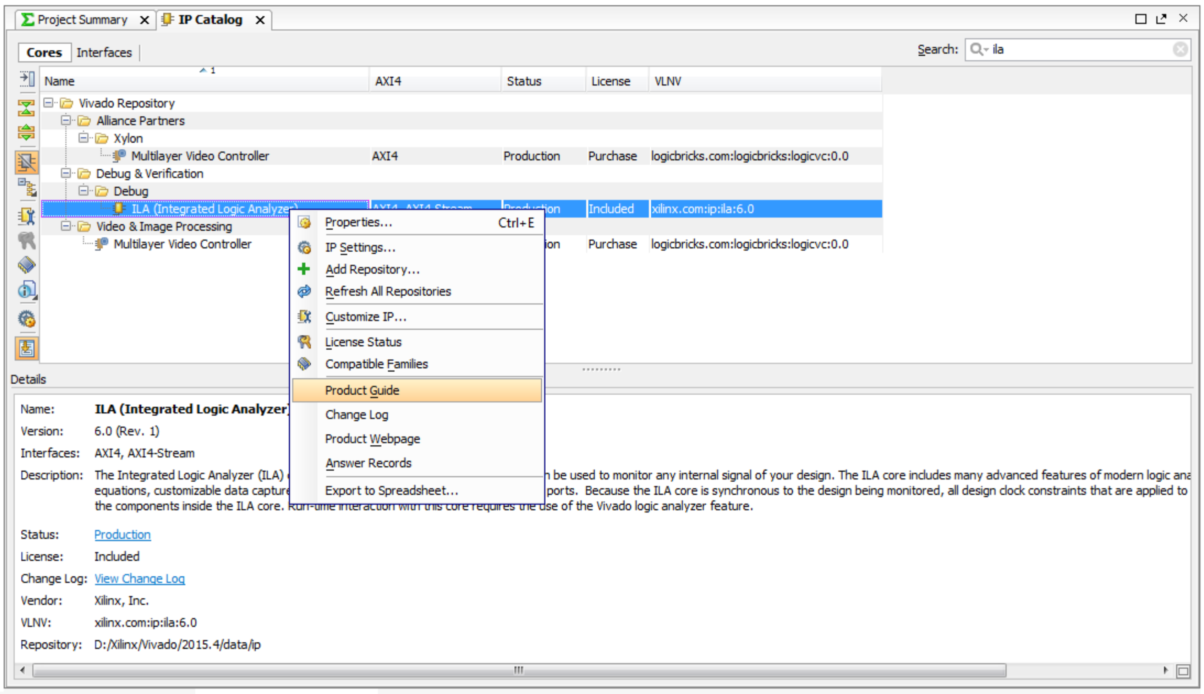

Sources

- Design Sources (3)
  - **modulator_wrapper - rtl** (modulator_wrapper_rtl.vhd) (1)
  - sine_top - rtl (sine_top_rtl.vhd) (3)
  - ila_core (ila_core.xci)
- Constraints
- Simulation Sources (3)

Figure 14.10: Sources tab with generated ILA core

Note: If you want to find product guide of the selected IP core

- right-click on the selected IP core in the IP Catalog window and choose **Product Guide** option, see Illustration 14.11. This option will open for you Xilinx web page for the selected IP core



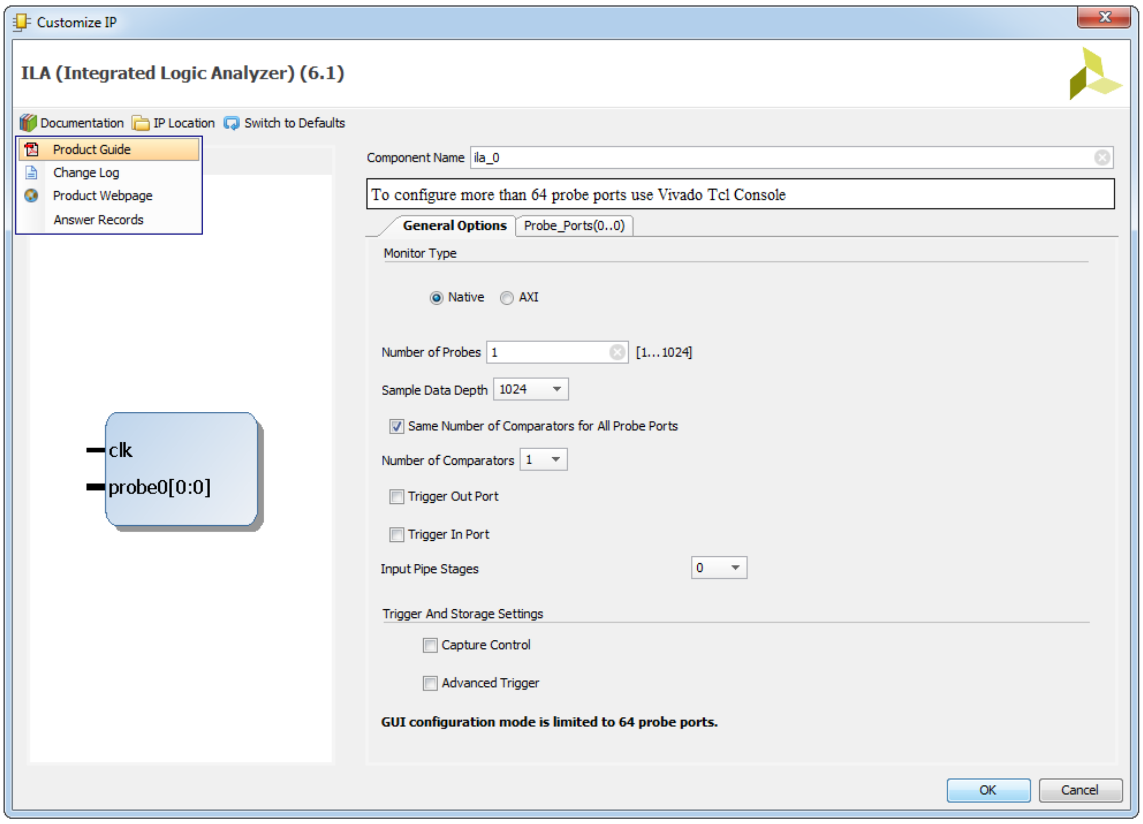
Name	AXI4	Status	License	VLNV
Vivado Repository				
Alliance Partners				
Xylon				
Multilayer Video Controller	AXI4	Production	Purchase	logicbricks.com:logicbricks:logicvc:0.0
Debug & Verification				
Debug				
ILA (Integrated Logic Analyzer)	AXI4, AXI4-Stream	Production	Included	xilinx.com:ip:ila:6.0
Video & Image Processing				
Multilayer Video Controller		Production	Purchase	logicbricks.com:logicbricks:logicvc:0.0

Details

Name:	ILA (Integrated Logic Analyzer)
Version:	6.0 (Rev. 1)
Interfaces:	AXI4, AXI4-Stream
Description:	The Integrated Logic Analyzer (ILA) ... be used to monitor any internal signal of your design. The ILA core includes many advanced features of modern logic ana... equations, customizable data capture ... ports. Because the ILA core is synchronous to the design being monitored, all design clock constraints that are applied to the components inside the ILA core. Run-time interaction with this core requires the use of the Vivado logic analyzer feature.
Status:	Production
License:	Included
Change Log:	View Change Log
Vendor:	Xilinx, Inc.
VLNV:	xilinx.com:ip:ila:6.0
Repository:	D:/Xilinx/Vivado/2015.4/data/ip

Figure 14.11: Product Guide option

- the another way is to double-click on the selected IP core in the IP Catalog window and in the main window of the selected IP core, click **Documentation** button and choose **Product Guide** option, see Illustration 14.12. This option will also open for you Xilinx web page for the selected IP core.



ILA (Integrated Logic Analyzer) (6.1)

Component Name: ila_0

To configure more than 64 probe ports use Vivado Tcl Console

Monitor Type: [✓ Native] AXI

Number of Probes: 1 [1...1024]

Sample Data Depth: 1024

[✓] Same Number of Comparators for All Probe Ports

Number of Comparators: 1

[ ] Trigger Out Port

[ ] Trigger In Port

Input Pipe Stages: 0

Trigger And Storage Settings

[ ] Capture Control

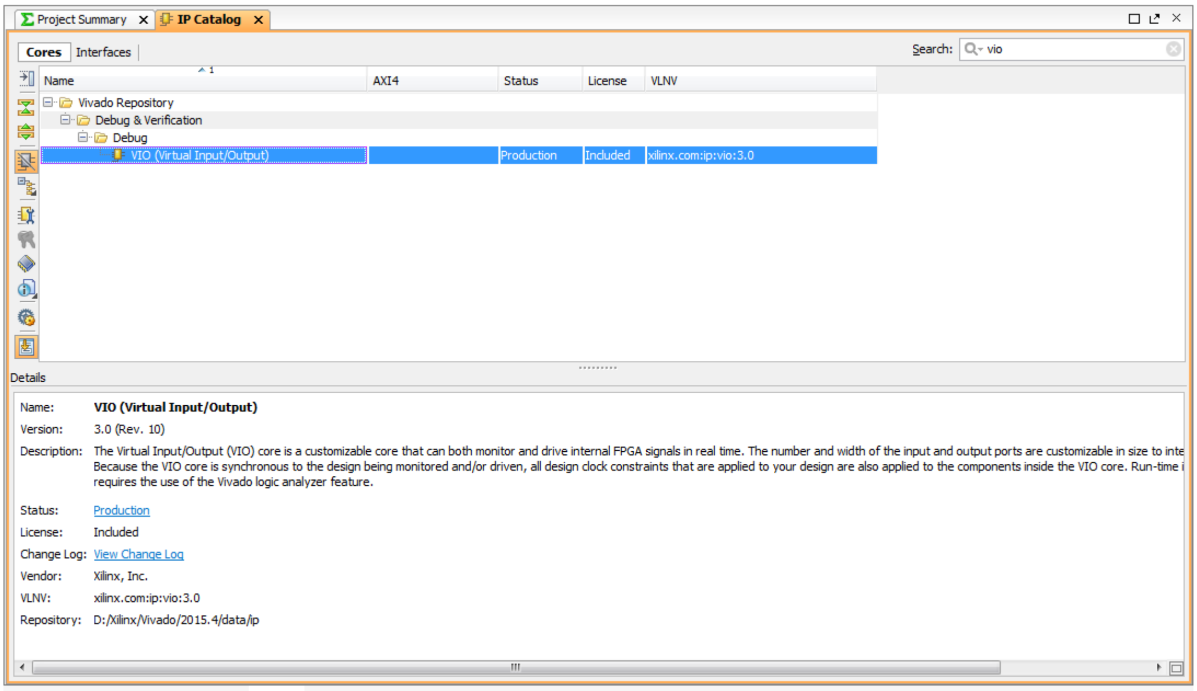
[ ] Advanced Trigger

GUI configuration mode is limited to 64 probe ports.

Figure 14.12: Documentation / Product Guide option

To configure and generate the VIO core, use the following steps:

Step 1. In the **IP Catalog** window, in the **Search** field, search for the **VIO (Virtual Input/Output)** IP core. After you selected the VIO core, in the **Details** window, under the main IP Catalog window, you will find all the necessary information about selected IP core, see Illustration 14.13



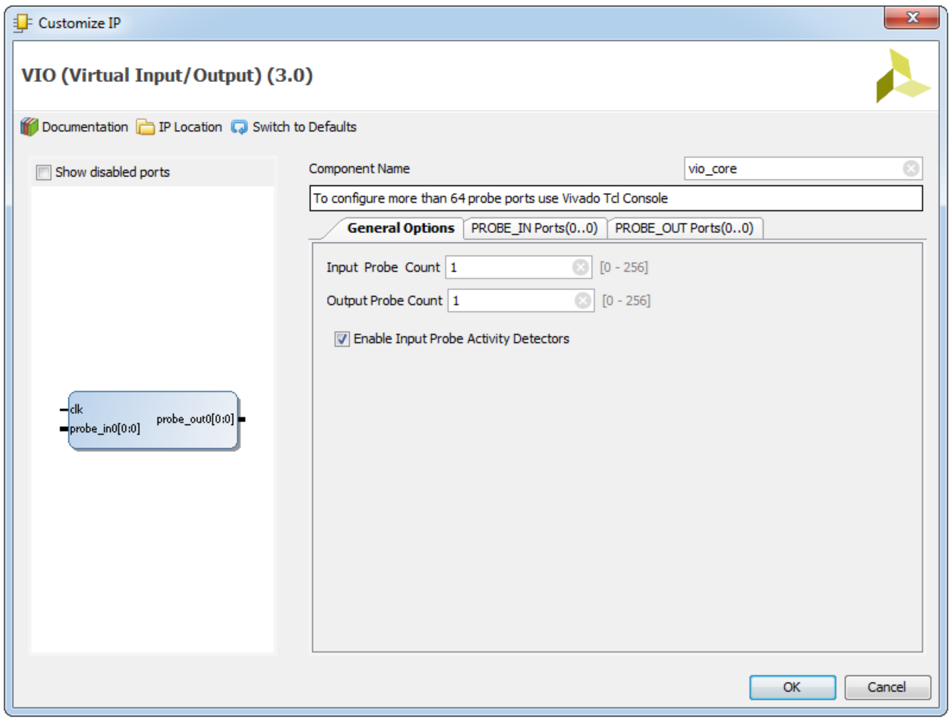
Name	AXI4	Status	License	VLNV
Vivado Repository				
Debug & Verification				
Debug				
VIO (Virtual Input/Output)		Production	Included	xilinx.com:ip:vio:3.0

Name:	**VIO (Virtual Input/Output)**
Version:	3.0 (Rev. 10)
Description:	The Virtual Input/Output (VIO) core is a customizable core that can both monitor and drive internal FPGA signals in real time. The number and width of the input and output ports are customizable in size to inte Because the VIO core is synchronous to the design being monitored and/or driven, all design clock constraints that are applied to your design are also applied to the components inside the VIO core. Run-time i requires the use of the Vivado logic analyzer feature.
Status:	Production
License:	Included
Change Log:	View Change Log
Vendor:	Xilinx, Inc.
VLNV:	xilinx.com:ip:vio:3.0
Repository:	D:/Xilinx/Vivado/2015.4/data/ip

Figure 14.13: IP Catalog window with selected VIO core

Step 2. Double-click on the **VIO (Virtual Input/Output)** IP core and Vivado IDE will create a new skeleton source for your VIO core

The window that will be opened is used to set up the general VIO core parameters, see Illustration 14.14



VIO (Virtual Input/Output) (3.0)

Show disabled ports

Component Name: vio_core

To configure more than 64 probe ports use Vivado Tcl Console

General Options

Input Probe Count: 1 [0 - 256]

Output Probe Count: 1 [0 - 256]

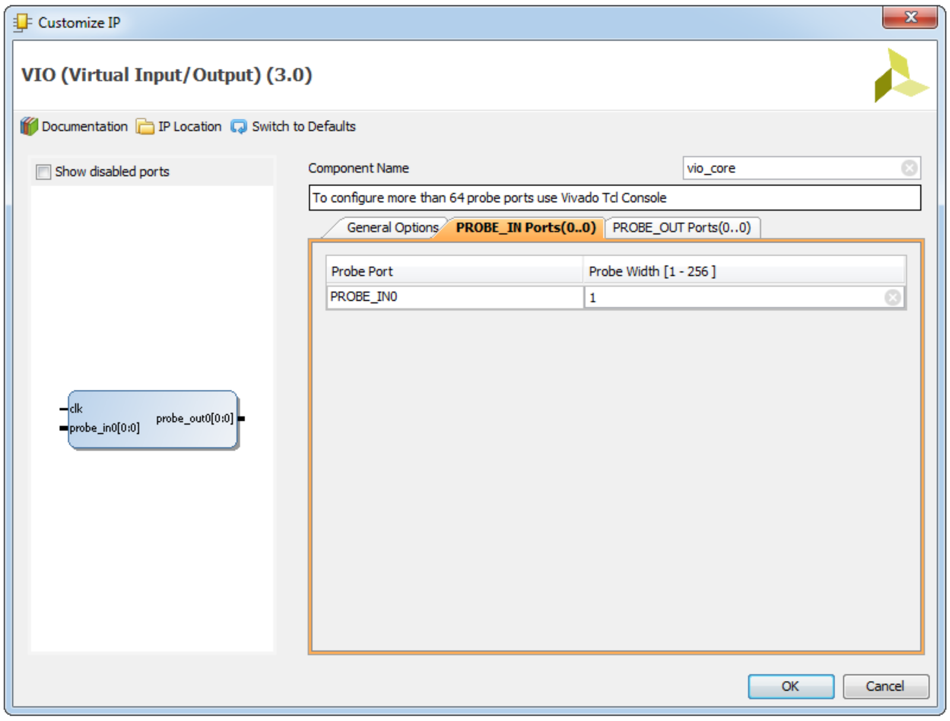
[✓] Enable Input Probe Activity Detectors

Figure 14.14: VIO core configuration window - General Options

Step 3. In the **VIO (Virtual Input/Output) (3.0)** window, enter *vio_core_name* (**vio_core**) in the **Component Name** field

Step 4. In the **General Options** tab, leave **Input Probe Count** to be **1** and **Output Probe Count** also to be **1**, because we will need one input probe for pwm_out signal and one output probe for sw0 signal, see Illustration 14.14

Step 5. In the **PROBE_IN Ports(0..0)** tab leave Probe Width of the **PROBE_IN0** Probe Port to be 1, because our pwm_out signal is 1 bit signal, see Illustration 14.15



VIO (Virtual Input/Output) (3.0)

Show disabled ports

Component Name: vio_core

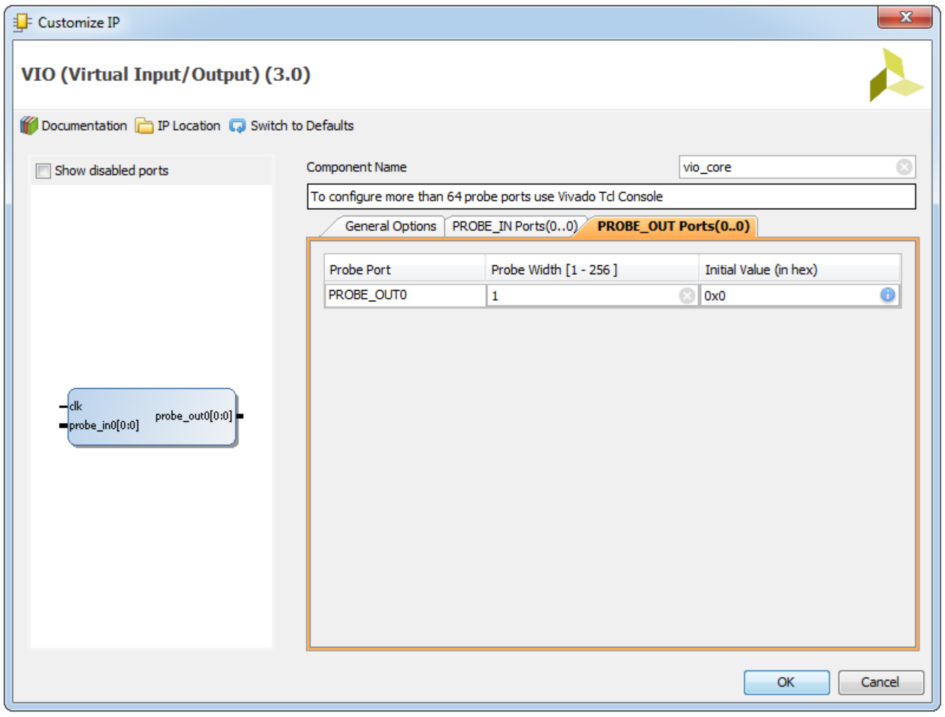
To configure more than 64 probe ports use Vivado Tcl Console

PROBE_IN Ports(0..0)

Probe Port	Probe Width [1 - 256]
PROBE_IN0	1

Figure 14.15: VIO core configuration window - PROBE_IN Ports(0..0) tab

Step 6. In the **PROBE_OUT Ports(0..0)** tab, leave Probe Width of the **PROBE_OUT0** Probe Port to be 1, because our sw0 signal is also 1 bit signal, see Illustration 14.16


Customize IP

VIO (Virtual Input/Output) (3.0)

Component Name: vio_core

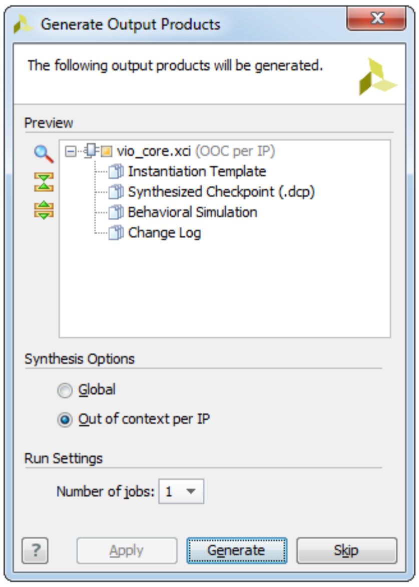
To configure more than 64 probe ports use Vivado Tcl Console

Probe Port	Probe Width [1 - 256]	Initial Value (in hex)
PROBE_OUT0	1	0x0

Figure 14.16: VIO core configuration window - PROBE_OUT Ports(0..0) tab

Step 7. Click **OK**

Step 8. In the **Generate Output Products** window click **Generate**, see Illustration 14.17


Generate Output Products

The following output products will be generated.

Preview

```
vio_core.xci (OOC per IP)
    Instantiation Template
    Synthesized Checkpoint (.dcp)
    Behavioral Simulation
    Change Log
```

Synthesis Options

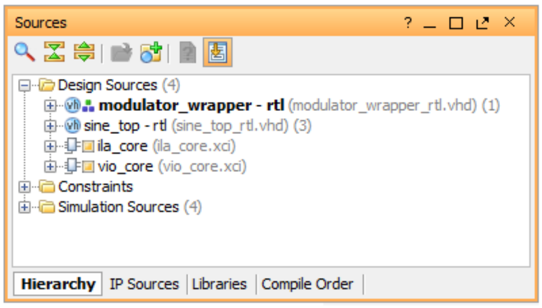
Global [✓ Out of context per IP]

Run Settings

Number of jobs: 1

Figure 14.17: Generate Output Products window for VIO core

Note: After VIO core generation, your VIO core should appear in the Sources window, see Illustration 14.18



Design Sources (4)
modulator_wrapper - rtl (modulator_wrapper_rtl.vhd) (1)
sine_top - rtl (sine_top_rtl.vhd) (3)
ila_core (ila_core.xci)
vio_core (vio_core.xci)
Constraints
Simulation Sources (4)

Figure 14.18: Source tab with generated VIO core

ILA and VIO Core Instantiation

After configuring and generating ILA and VIO cores, we should make a new module (**modulator_ila_vio_rtl.vhd**) where we will connect the existing design (**modulator_rtl.vhd**) with **ILA** and **VIO** cores (see Figure 14.19). By doing so, for the *sw0* port control it wont be necessary to use switch on the development board. Instead, we will use one of the VIO core's outputs to control the *sw0* port. This will enable us to change the state of the *sw0* port inside the Vivado Logic Analyzer.

To create the new **modulator_ila_vio_rtl.vhd** file, you can use existing **modulator_wrapper_rtl.vhd** file, making the following changes:

- remove *sw0* port from the port map
- create internal signals *pwm_s* ,*sw0_s* and *debug_data_s* as std_logic_vectors
- declare ILA and VIO core components
- in the Modulator module instance connect:
  - *sw0* port to the 0'th bit of the *sw0_s* signal
  - *pwm_out* port to the 0'th bit of the *pwm_out_s* signal
  - *debug_data* port with the vdebug_data_s signal
- instantiate and connect ILA and VIO cores as it is shown on the Figure 14.19.

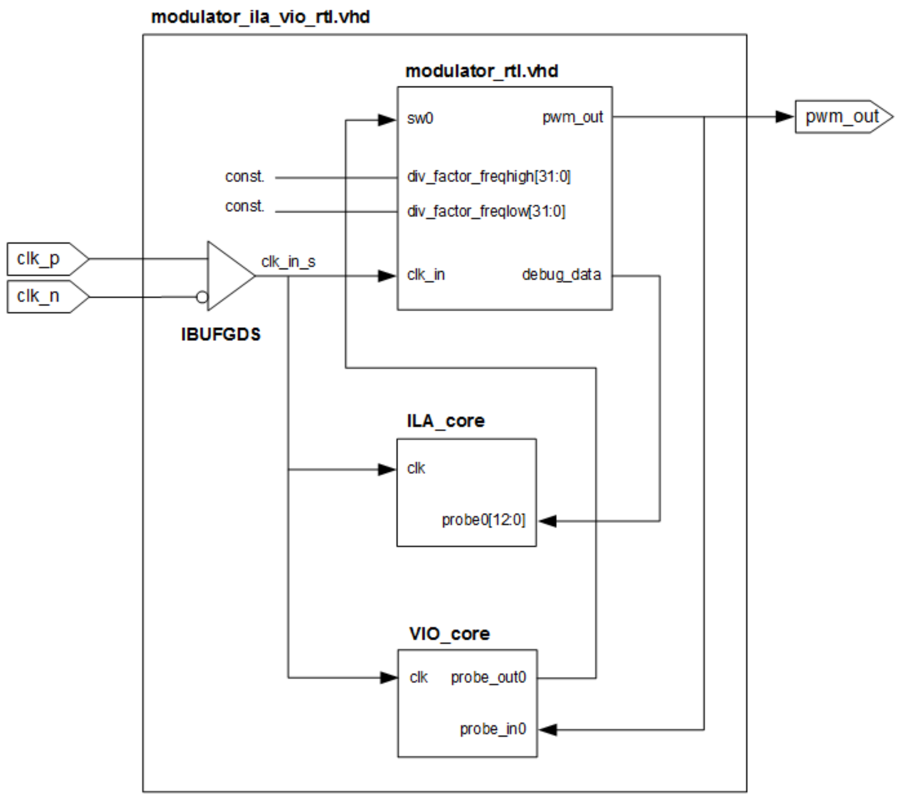


Figure 14.19: Connection between the ILA core, VIO core and Modulator module

Now we will create an VHDL module (**modulator_ila_vio_rtl.vhd**) that will make connection between the ILA core, VIO core and Modulator module (**modulator_rtl.vhd**).

To create a module, use steps for creating modules, **Chapter 2.4.1 Creating a Module Using Vivado Text Editor** .

To help you to correctly instantiate the ILA and VIO cores into your design, Xilinx tools always provide an instantiation template stored in the ∗.vho file in case of VHDL language or ∗.veo file in case of Verilog language usage.

In our case ILA core instantiation template file is located in the following folder:

modulator/modulator.srcs/sources_1/ip/ILA_core/ILA_core.vho

Similarly, VIO core instantiation template file is located in the following folder:

modulator/modulator.srcs/sources_1/ip/VIO_core/VIO_core.vho

One possible way to implement the **modulator_ila_vio_rtl.vhd** module is shown below.

modulator_ila_vio_rtl.vhd **:**

```
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 modulator_ila_vio 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
        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 rtl of modulator_ila_vio is

    signal clk_in_s     : std_logic;
    signal pwm_s        : std_logic_vector (0 downto 0);
    signal sw0_s        : std_logic_vector (0 downto 0);
    signal debug_data_s : std_logic_vector (12 downto 0);

    -- c1_c = fclk/(2^depth*2^width)                     - c1_c = 95.3674, fclk = 100 MHz
    constant c1_c : real :=
       get_board_info_f(board_name_g).fclk/(real((2**design_setting_g.depth)*(2**design_setting_g.width)));
     -- div_factor_freqhigh_c = (c1_c/f_high)*2^width  - threshold value of frequency a = 110592
    constant div_factor_freqhigh_c : integer :=
       integer(c1_c/design_setting_g.f_high)*(2**design_setting_g.width);
    -- div_factor_freqlow_c  = (c1_c/f_low)*2^width    - threshold value of frequency b = 389120
    constant div_factor_freqlow_c  : integer :=
       integer(c1_c/design_setting_g.f_low)*(2**design_setting_g.width);

    -- ila_core component definition
    component ila_core
        port (
            clk    : in std_logic;
            probe0 : in std_logic_vector (12 downto 0)
            );
    end component;

    -- vio_core component definition
    component vio_core
        port (
            clk        : in std_logic;
            probe_in0  : in std_logic_vector (0 downto 0);
            probe_out0 : out std_logic_vector (0 downto 0)
            );
    end component;

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

    pwm_out <= pwm_s (0);

    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;


    -- modulator module instance
    modulator: entity work.modulator(rtl)
        generic map(
            design_setting_g => design_setting_g
            )

        port map(
            clk_in              => clk_in_s,
            sw0                 => sw0_s(0),
            div_factor_freqhigh => conv_std_logic_vector(div_factor_freqhigh_c, 32),
            div_factor_freqlow  => conv_std_logic_vector(div_factor_freqlow_c, 32),
            pwm_out             => pwm_s(0),
            debug_data          => debug_data_s
            );

    -- ila_core component instance
    ila: ila_core
        port map (
            clk    => clk_in_s,
            probe0 => debug_data_s
            );

    -- vio_core component instance
    vio: vio_core
        port map (
            clk         => clk_in_s,
            probe_out0  => sw0_s,
            probe_in0   => pwm_s
            );

end;
```

As you can see from the picture above (Figure 14.19), we have added the **debug_data** output port to the Modulator module (modulator_rtl.vhd) and connected it to the **PROBE0** input port of the ILA core. This is important, because we will connect the internal **sine_ampl_s** and **freq_trig_s** signals from the Modulator module to the **debug_data** port, and use them as the trigger/data signals for the ILA core.

As we can also see from the picture above (Figure 14.19), we have connected the **sw0** input port of the Modulator module to the **PROBE_OUT0** output port of the VIO core. This is also important, because now we don't need any physical switch from the development board to drive the **sw0** input port. Now, this is done using an synchronous output port from the VIO core (**sw0_s** signal). This signal can be controlled within the Vivado Logic Analyzer tool to change the value of the **sw0** input port, and by doing so the frequency of the **pwm_out** signal.

To make these modifications, **modulator_rtl.vhd** source code must be modified as it is shown bellow:

Modulator VHDL model:

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

    use work.modulator_pkg.all;

library unisim;
    use unisim.vcomponents.all;

entity modulator is
    generic(
        -- User defined settings for the pwm design
        design_setting_g : design_setting_t_rec := design_setting_c
        );

    port(
        clk_in              : in std_logic;                       -- input clock signal
        sw0                 : in std_logic;                       -- signal made for selecting frequency
        div_factor_freqhigh : in std_logic_vector(31 downto 0); -- input clock division when sw0 = '1'
        div_factor_freqlow  : in std_logic_vector(31 downto 0); -- input clock division when sw0 = '0'
        pwm_out             : out std_logic;                      -- pulse width modulated signal
        debug_data          : out std_logic_vector (12 downto 0)
        );
end entity;


architecture rtl of modulator is

    -- amplitude counter
    signal ampl_cnt_s  : std_logic_vector(design_setting_g.depth-1 downto 0);
    -- current amplitude value of the sine signal
    signal sine_ampl_s : std_logic_vector(design_setting_g.width-1 downto 0);
    -- signal which frequency depends on the sw0 state
    signal freq_trig_s : std_logic := '0';


begin

    freq_ce : entity work.frequency_trigger(rtl)   -- frequency trigger module instance
        port map(
            clk_in              => clk_in,
            sw0                 => sw0,
            div_factor_freqhigh => div_factor_freqhigh,
            div_factor_freqlow  => div_factor_freqlow,
            freq_trig           => freq_trig_s
            );
    counterampl : entity work.counter(rtl)   -- counter module instance
        generic map(
            cnt_value_g => design_setting_g.cntampl_value,
            depth_g     => design_setting_g.depth
            )

        port map (
            clk_in  => clk_in,
            cnt_en  => freq_trig_s,
            cnt_out => ampl_cnt_s
            );

    sine : entity work.sine(rtl)   -- digital sine module instance
        generic map(
            depth_g => design_setting_g.depth,
            width_g => design_setting_g.width
            )

        port map(
            ampl_cnt => ampl_cnt_s,
            clk_in   => clk_in,
            sine_out => sine_ampl_s
            );

    pwmmodule : entity work.pwm (rtl)   -- pwm module instance
        generic map (
            width_g => design_setting_g.width
            )

        port map (
            clk_in              => clk_in,
            sw0                 => sw0,
            sine_ampl           => sine_ampl_s,
            div_factor_freqhigh => conv_std_logic_vector(conv_integer(div_factor_freqhigh)/(2**
      design_setting_g.width), 32),
            div_factor_freqlow  => conv_std_logic_vector(conv_integer(div_factor_freqlow)/(2**
      design_setting_g.width), 32),
            pwm_out             => pwm_out
            );

        debug_data (11 downto 0) <= sine_ampl_s;
        debug_data (12) <= freq_trig_s;

end;
```

After we made a new VHDL module (**modulator_ila_vio_rtl.vhd**), we must also modify the **modulator_rtl.xdc** file, because we don't have any more *sw0* port. The new content of the xdc file is shown in the code below.

modulator_ila_vio.xdc file:

```
set_property PACKAGE_PIN Y9 [get_ports clk_p]
set_property PACKAGE_PIN T22 [get_ports pwm_out]

set_property IOSTANDARD LVCMOS33 [get_ports clk_p]
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]
```

After finishing with the modifications, we must return to the Vivado IDE and do the following:

Step 1. Remove ***modulator_wrapper_rtl.vhd*** source file from the design

Step 2. Add ***modulator_ila_vio_rtl.vhd*** and ***modulator_ila_vio.xdc*** files in the modulator design with **Add Sources** option:

- ***modulator_ila_vio_rtl.vhd*** as Design Source file, and
- ***modulator_ila_vio.xdc*** as Constraints file

Step 3. Remove the old ***modulator.xdc*** file from the design

Step 4. In the **Sources** window, right-click on the ***modulator_ila_vio_rtl.vhd*** file and select **Set as Top** option

Step 5. Made the necessary changes in the ***modulator_rtl.vhd*** source file as it is explained in the text above

Step 6. Synthesize your design with **Run Synthesis** option from the **Flow Navigator / Synthesis** (see **Sub-chapter 6.5.2 Run Synthesis**)

Step 7. Implement your design with **Run Implementation** option from the **Flow Navigator / Implementation** (see **Sub--Chapter 10.2.2 Run Implementation**)

Step 8. Generate bitstream file with **Generate Bitstream** option from the **Flow Navigator** / **Program and Debug** (see **Sub-Chapter 10.3 Generate Bitstream File**)

Step 9. Program your ZedBoard device (see **Sub-Chapter 10.4 Program Device**)

14.2 Using the HDL Instantiation Debug Probing Flow in IP Integrator

As shown in the previous chapter *"HDL Instantiation Debug Probing Flow"*, we will instantiate an Integrated Logic Analyzer (ILA) and Virtual Input/Output (VIO) cores into our IP integrator design and connect nets in the same way as it is shown on the **Figure 11.9**.

To start debugging process using the HDL Instantiation Flow in IP Integrator tool, please do the following:

Step 1. Create new project ***modulator_ipi_hdl***

Step 2. In the ***modulator_ipi_hdl*** project create new ***modulator_ipi_hdl*** block design

Step 3. Add previously packaged IPs (*frequency_trigger_v1_0, counter_v1_0, sine_v1_0* and *pwm_v1_0*) to the IP Catalog by repeating the steps **32 - 38** from the **Sub-chapter 13.1 IP Packager**.

Step 4. Add all four IPs (***frequency_trigger_v1_0, counter_v1_0, sine_v1_0*** and ***pwm_v1_0***) to the ***modulator_ipi_hdl*** block design

Step 5. Customize the IPs on the same way as it is explained in the **Sub-chapter 13.2 IP Integrator**, step **17**

Step 6 Add four **Constant** IP blocks to the ***modulator_ipi_hdl*** block design and customize them on the same way as it is explained in the **Sub-chapter 13.2 IP Integrator**, steps **22 - 25**

Step 7. Add also **ILA** and **VIO** IPs to the ***modulator_ipi_hdl*** block design

Step 8. Leave **VIO** core as it is

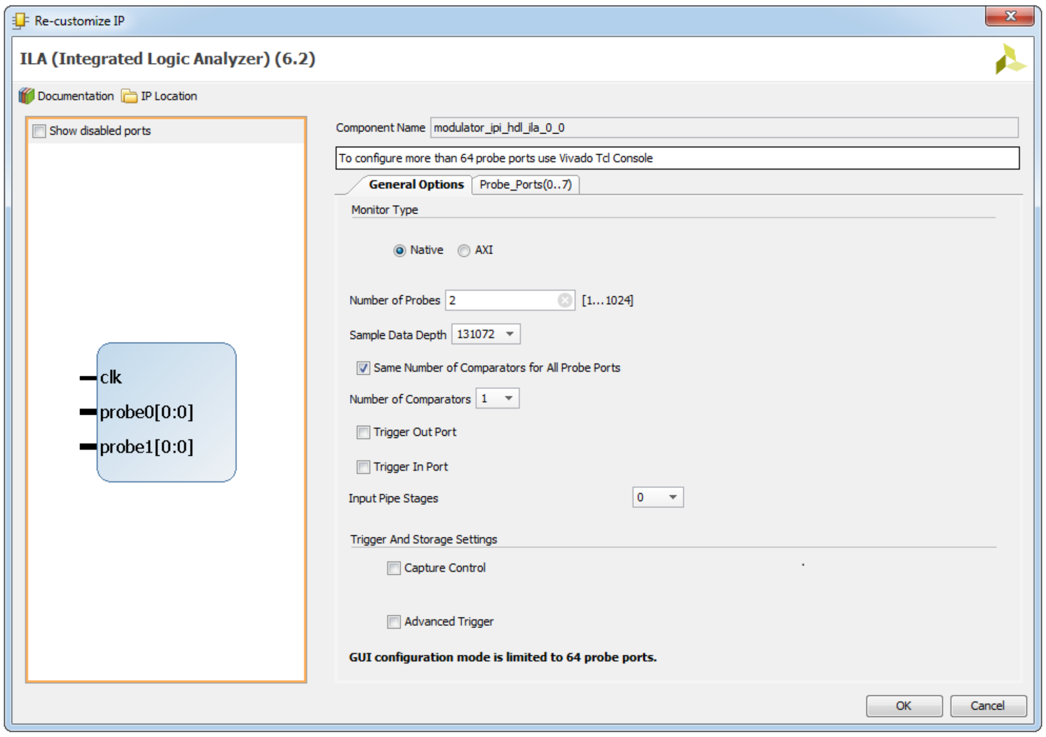
Step 9. In the IP Integrator canvas, double-click on the ILA core to re- configure it

Step 10. In the **General Options** tab of the ILA core, re-configure the following parameters:

- Set **Monitor Type** to be **Native**, instead of AXI

- Set **Number of Probes** to **2**

- Set **Sample Data Depth** to be on the maximum **131072** samples


Re-customize IP

ILA (Integrated Logic Analyzer) (6.2)

Component Name: modulator_ipi_hdl_ila_0_0

To configure more than 64 probe ports use Vivado Tcl Console

General Options | Probe_Ports(0..7)

Monitor Type: [✓ Native] AXI

Number of Probes: 2 [1...1024]

Sample Data Depth: 131072

[✓] Same Number of Comparators for All Probe Ports

Number of Comparators: 1

[ ] Trigger Out Port

[ ] Trigger In Port

Input Pipe Stages: 0

Trigger And Storage Settings

[ ] Capture Control

[ ] Advanced Trigger

GUI configuration mode is limited to 64 probe ports.

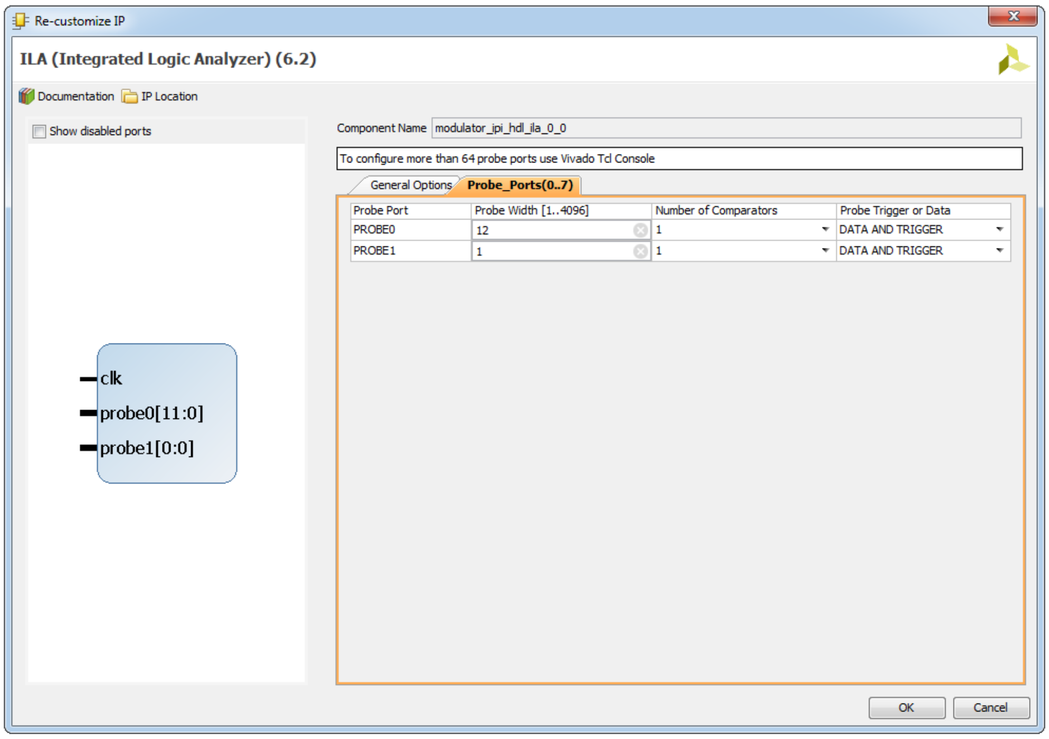
Figure 14.20: ILA core configuration window - General Options tab

Step 11. In the **Probe_Ports(0..7)** tab of the ILA core, re- configure the following parameters:

- Set the **Probe Width [1..4096]** of the **PROBE0** probe port to be **12**

- Leave the **Probe Width [1..4096]** of the **PROBE1** probe port to be **1**

We configured the probe width of the **PROBE0** probe port to **12** and **PROBE1** to **1**, because the width of the **sine_ampl_s** signal, that we want to see in the Vivado Logic Analyzer, is 12 bits and the width of the **freq_trig_s** signal is 1 bit.



Re-customize IP

ILA (Integrated Logic Analyzer) (6.2)

Documentation IP Location

Show disabled ports

Component Name modulator_ipi_hdl_ila_0_0

To configure more than 64 probe ports use Vivado Tcl Console

General Options | Probe_Ports(0..7)

Probe Port	Probe Width [1..4096]	Number of Comparators	Probe Trigger or Data
PROBE0	12	1	DATA AND TRIGGER
PROBE1	1	1	DATA AND TRIGGER

Figure 14.21: ILA core configuration window - Probe_Ports(0..7) tab

Step 12. In case of VIO core, use default configuration settings

Step 13. Connect the ILA core with the rest of the IPs in the same way as it is shown on the **Figure 14.19**

Step 14. Remove **sw0** port from the IP Integrated canvas and connect the VIO core with the rest of the IPs in the same way as it is shown on the **Figure 14.19**, see Illustration 14.22

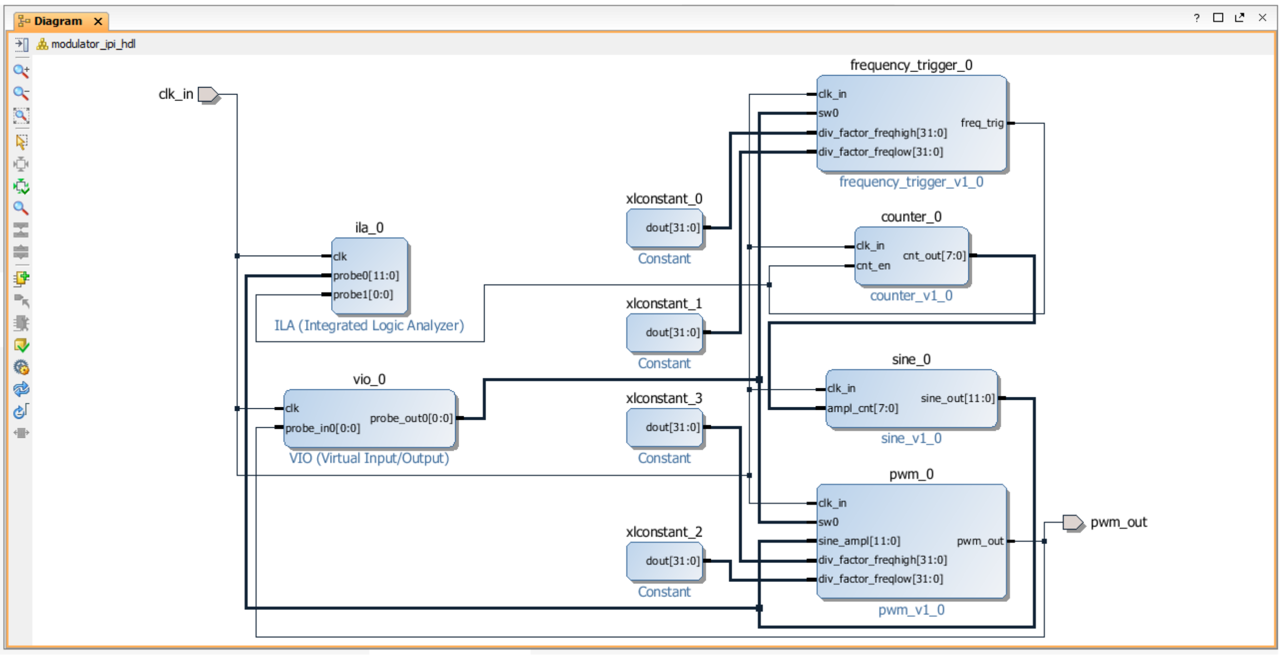


Figure 14.22: IP Integrator design canvas with connected ILA and VIO cores

Step 15. Create and add ***modulator_ipi_hdl.xdc*** constraints file to the project. The content of the ***modulator_ipi_hdl.xdc*** constraints file is shown in the text below:

```
set_property PACKAGE_PIN Y9 [get_ports clk_in]
set_property PACKAGE_PIN T22 [get_ports pwm_out]

set_property IOSTANDARD LVCMOS33 [get_ports clk_in]
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]
```

Step 16. Validate your design by selecting **Tools ->** **Validate Design** from the main menu

Step 17. Select ***modulator_ipi_hdl***, right-click on it and choose **Create HDL Wrapper...** option

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

Step 19. Implement your design with **Run Implementation** option from the **Flow Navigator / Implementation** (see **Sub--Chapter 10.2.2 Run Implementation**)

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

Step 21. Program your ZedBoard device (see **Sub-Chapter 10.4 Program Device**)

Step 22. After programming your design, you should get the same results as we presented in the **Sub-chapter 11.2 Debug a Design using Integrated Vivado Logic Analyzer** of this tutorial.