Basic FPGA Tutorial

using VHDL and VIVADO to design two frequencies PWM modulator system
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>8 MODULATOR SOZIUS WRAPPER</strong></td>
<td>77</td>
</tr>
<tr>
<td>8.1 Description</td>
<td>77</td>
</tr>
<tr>
<td>8.2 Creating Module</td>
<td>78</td>
</tr>
<tr>
<td><strong>9 DESIGN IMPLEMENTATION</strong></td>
<td>87</td>
</tr>
<tr>
<td>9.1 Creating XDC File</td>
<td>87</td>
</tr>
<tr>
<td>9.1.1 Defining Timing Constraints</td>
<td>93</td>
</tr>
<tr>
<td>9.2 Implementation</td>
<td>101</td>
</tr>
<tr>
<td>9.2.1 About the Vivado Implementation Process</td>
<td>101</td>
</tr>
<tr>
<td>9.2.2 Run Implementation</td>
<td>103</td>
</tr>
<tr>
<td>9.2.3 After Implementation</td>
<td>104</td>
</tr>
<tr>
<td>9.2.4 Implementation Reports</td>
<td>106</td>
</tr>
<tr>
<td>9.3 Generate Bitstream File</td>
<td>109</td>
</tr>
<tr>
<td>9.4 Program Device</td>
<td>110</td>
</tr>
<tr>
<td>9.5 Modifications in case of using different development boards</td>
<td>118</td>
</tr>
<tr>
<td><strong>10 DEBUGGING DESIGN</strong></td>
<td>125</td>
</tr>
<tr>
<td>10.1 Inserting ILA and VIO Cores into Design</td>
<td>125</td>
</tr>
<tr>
<td>10.2 Debug a Design using Integrated Vivado Logic Analyzer</td>
<td>143</td>
</tr>
<tr>
<td><strong>11 DESIGNING WITH IPs</strong></td>
<td>153</td>
</tr>
<tr>
<td>11.1 IP Integrator</td>
<td>154</td>
</tr>
<tr>
<td>11.2 Creating Modulator IP Core with AXI4 Interface</td>
<td>174</td>
</tr>
</tbody>
</table>
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

Introduction

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

The following project is designed for:

- Designing Surface: VIVADO 2018.2
- HD Language: VHDL
- Simulator: Vivado Simulator
- Device: Sozins 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
- Designing with IPs
CHAPTER 1. INTRODUCTION

1.3 Structure of this Tutorial

This tutorial is composed of twelve 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, block diagram of one possible solution for the modulator design and a lot of basic information about the Vivado Design Suite.

- **Chapter 2: "Selector"** - In this chapter you will find all the necessary information about how to create a new project in the Vivado IDE, how to create Selector 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: "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 7: "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. Additionally, this chapter holds information about the Vivado synthesis process.

- **Chapter 8: "Modulator Sozius Wrapper"** - This chapter includes all the necessary information about the Modulator Sozius Wrapper module. This module will be used to target Sozius development board. Considering that the main component of the Sozius development board is Zynq-7000 AP SoC, in this chapter you will find all the necessary information how to use PS and PL parts of the Zynq-7000 AP SoC system for the purpose of our project.

- **Chapter 9: "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 10: "Debugging Design"** - This chapter explains the process of debugging design. In this chapter you will find the information how to instantiate ILA and VIO cores into the design and how to debug your design using integrated Vivado Logic Analyzer.

- **Chapter 11: "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.

This tutorial is accompanied by the pdf lab presentations. In total there are 16 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: "Quick Guide to Running Modulator Design on FPGA Board"** - 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 Selector Module"** - covers the information presented in the sub-chapters 2.1, 2.2, 2.4, 2.4.1 of Chapter 2: "Selector" of this tutorial.

- **Lab 4: "Selector Verification"** - covers the information presented in the sub-chapters 2.5, 2.6 of Chapter 2: "Selector" 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 PWM Module" - covers the information presented in the Chapter 6: "PWM" of this tutorial.

Lab 9: "Creating Modulator Module" - covers the information presented in the Chapter 7: "Modulator" of this tutorial.

Lab 10: "Creating Modulator Sozius Wrapper Module" - covers the information presented in the Chapter 8: "Modulator Sozius Wrapper" of this tutorial.

Lab 11: "Creating XDC File" - covers the information presented in the sub-chapter 9.1 of Chapter 9: "Design Implementation" of this tutorial.

Lab 12: "Design Implementation" - covers the information presented in the sub-chapter 7.5 of Chapter 7: "Modulator" and sub-chapters 9.2, 9.3, 9.4 of Chapter 9: "Design Implementation" of this tutorial.

Lab 13: "Debugging Design" - covers the information presented in the sub-chapter 10.1 of the Chapter 10 "Debugging Design" of this tutorial.

Lab 14: "Debug a Design using Integrated Vivado Logic Analyzer" - covers the information presented in the sub-chapter 10.2 of the Chapter 10 "Debugging Design" of this tutorial.

Lab 15: "Designing with IPs - IP Integrator" - covers the information presented in the sub-chapter 11.1 of the Chapter 11 "Designing with IPs" of this tutorial.

Lab 16: "Creating Modulator IP with AXI4 Interface" - covers the information presented in the sub-chapter 11.2 of the Chapter 11 "Designing with IPs" of this tutorial.

1.4 Objectives of this Tutorial

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.

PWM Signal

Pulse-width modulation (PWM) uses a rectangular pulse wave whose pulse width is modulated by some other signal (in our case we will use a sine wave) resulting in the variation of the average value of the waveform. Typically, PWM signals are used to either convey information over a communications channel or control the amount of power sent to a load. To learn more about PWM signals, please visit http://en.wikipedia.org/wiki/Pulse-width_modulation
Figure 1.1: Example of the PWM signal

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

1.5 One Possible Solution for the Modulator Design

One Possible Solution

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

Figure 1.2: Sine wave with 256 samples

Figure 1.2 is showing the sine wave that will be used to modulate the PWM signal.
CHAPTER 1. INTRODUCTION

Block diagram

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

![Block diagram](image)

Figure 1.3: Block diagram

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

**Selector**

This module will generate one output signal with two possible increment values for Counter module. Which increment value will be chosen depends on the position of the two-state on-board switch (sel_i).

**Counter**

This module is an up counter with user-selectable increment value. Its task will be to generate read addresses for the ROM where samples of the sine wave are stored. The step of the counting will be controlled by the Selector module, via inc_i 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 implement the Finite State Machine (FSM), that will be used to 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.
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 Figure 1.5 are shown steps in designing modules of this lab:

Design Steps

Figure 1.4: Details of PWM signal generation

Figure 1.5: Project Design Steps
• First we will create the Selector module that will provide one output signal with two possible increment values.

• 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 module that will generate PWM signal.

• At the end, we will create Modulator module where we will merge all the previously designed modules into one big design.

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".

1.6 Design Flow

On the Figure 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.

![Figure 1.7: Design Verification Steps](image)

• **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.

*Note:* In the Lab 2: "Quick Guide to Running Modulator Design on FPGA Board" 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.

### 1.8 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>
<thead>
<tr>
<th>Flow Element</th>
<th>Project Mode</th>
<th>Non-Project Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design Source File Management</td>
<td>Automatic</td>
<td>Manual</td>
</tr>
<tr>
<td>Flow Customization</td>
<td>Limited</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Reporting</td>
<td>Automatic</td>
<td>Manual</td>
</tr>
<tr>
<td>Analysis Stages</td>
<td>Designs only</td>
<td>Designs and checkpoints</td>
</tr>
</tbody>
</table>

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

Figure 1.8 shows the differences between Project and Non-Project Mode Tcl commands.
CHAPTER 1. INTRODUCTION

![Table and Diagram](image)

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.joule 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 1.2: Basic Project Mode Tcl Commands**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>create_project</td>
<td>Creates the Vivado IDE project. Arguments include project name and location, design top module name, and target part.</td>
</tr>
<tr>
<td>add_files</td>
<td>Adds source files to the project. These include Verilog (.v), VHDL (.vhdl 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.</td>
</tr>
<tr>
<td>set_property</td>
<td>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.</td>
</tr>
<tr>
<td>import_files</td>
<td>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.</td>
</tr>
<tr>
<td>launch_runs</td>
<td>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 through that process. The -to_step option is used to launch the implementation process, including bitstream generation, in incremental steps.</td>
</tr>
<tr>
<td>wait_on_run</td>
<td>Ensures the run is complete before processing the next steps in the flow.</td>
</tr>
<tr>
<td>open_run</td>
<td>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.</td>
</tr>
<tr>
<td>close_design</td>
<td>Closes the design in memory.</td>
</tr>
<tr>
<td>start_gui stop_gui</td>
<td>Invokes or closes the Vivado IDE with the current design in memory.</td>
</tr>
</tbody>
</table>

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.
Table 1.3: Basic Non-Project Mode Tcl Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>read_edif</td>
<td>Imports an EDIF or NGC netlist file into the Design Source fileset of current project.</td>
</tr>
<tr>
<td>read_verilog</td>
<td>Reads the Verilog (.v) and SystemVerilog (.sv) source files for the Non-Project Mode session.</td>
</tr>
<tr>
<td>read_vhdl</td>
<td>Reads VHDL (.vhdl or .vhdl) source files for the Non-Project Mode session.</td>
</tr>
<tr>
<td>read_ip</td>
<td>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.</td>
</tr>
<tr>
<td>read_xdc</td>
<td>Reads the .sdc or .xdc format constraints source files for the Non-Project Mode session.</td>
</tr>
<tr>
<td>set_param set_property</td>
<td>Used for multiple purposes. For example, it can be used to define design configuration, tool settings, and so forth.</td>
</tr>
<tr>
<td>link_design</td>
<td>Compiles the design for synthesis if netlist sources are used for the session.</td>
</tr>
<tr>
<td>synth_design</td>
<td>Launches Vivado synthesis with the design top module name and target part as arguments.</td>
</tr>
<tr>
<td>opt_design</td>
<td>Performs high-level design optimization.</td>
</tr>
<tr>
<td>power_opt_design</td>
<td>Performs intelligent clock gating to reduce overall system power. This is an optional step.</td>
</tr>
<tr>
<td>place_design</td>
<td>Places the design.</td>
</tr>
<tr>
<td>phys_opt_design</td>
<td>Performs physical logic optimization to improve timing or routability. This is an optional step.</td>
</tr>
<tr>
<td>route_design</td>
<td>Routes the design.</td>
</tr>
<tr>
<td>report_*</td>
<td>Runs a range of standard reports, which can be run at any stage of the design process.</td>
</tr>
<tr>
<td>write_bitstream</td>
<td>Generates a bitstream file and runs DRCs.</td>
</tr>
<tr>
<td>write_checkpoint</td>
<td>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.</td>
</tr>
<tr>
<td>start_gui stop_gui</td>
<td>Invokes or closes the Vivado IDE with the current design in memory.</td>
</tr>
</tbody>
</table>
Chapter 2

SELECTOR

2.1 Description

- **Description**
  
  - **Usage:** This module will generate one output signal with two possible increment values for Counter module. Which increment value will be chosen depends on the position of the two-state on-board switch (sel_i).
  
  - **Block diagram:**

    ![Selector block diagram](image)

    **Figure 2.1:** Selector block diagram

    - **Input ports:**
      
      - `clk_i`: input clock signal
      - `inc_i`: different increments for different frequencies
      - `sel_i`: signal made for selecting frequency

    - **Output ports:**
      
      - `inc_o`: output signal with appropriate increment value, which depends on the state of the sel_i input signal

    - **Generics:**
      
      - `num_freqs_g`: number of possible frequencies
      - `width_g`: the number of bits used to represent amplitude value

    - **File name:** `selector_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 create 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:

- Launch the Vivado software: Select Start -> All Programs -> Xilinx Design Tools -> Vivado 2018.2 -> Vivado 2018.2 and the Vivado Getting Started page will appear, see Figure 2.2.

Create Project

![Vivado Getting Started page](image)

Figure 2.2: The Vivado Getting Started page

- On the Getting Started page, choose first offered Create Project option, under the Quick Start section.

- In the Create a New Vivado Project dialog box click Next and the wizard will guide you through the creation of a new project.
- In the **Project Name** dialog box specify the name and the location of the new project and click **Next**.

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

As you can see from the figure 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.

- In the **Default Part** dialog box choose a default Xilinx part or board for your project and click **Next**.
Figure 2.6: Default Part dialog box

The main component of the Sozius development board is Zynq-7000 AP SoC, so in the Default Part dialog box select Parts option and set the filter parameters as it is shown on the Figure 2.6.

- In the New Project Summary dialog box click Finish if you are satisfied with the summary of your project.

Figure 2.7: New Project Summary dialog box

If you are not satisfied, you can go back as much as necessary to correct all the questionable issues, see Figure 2.7.

After we finished with the new project creation, in a few seconds Vivado IDE Viewing Environment will appear, see Figure 2.8.
When Vivado creates a new project, it also creates a directory with the name and at the location that we specified in the GUI (see Figure 2.4). That means that all project data will be stored in the `project_name` 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

![Vivado IDE Viewing Environment](image)

Figure 2.8: Vivado IDE Viewing Environment

Vivado IDE tool will collect all the design units in the `xil_defaultlib` library by default. Change the library from `xil_defaultlib` to `work`, because `work` library is the default place for majority of industry tools to store design units.

To change this feature click **Settings** command in the **Project Manager** and in the **Settings** dialog box, under the **General** options, change **Default library** to be `work`, see Figure 2.9.
Change the Default Library

![Settings dialog box with changed default library to "work"](image)

Figure 2.9: Settings dialog box with changed default library to "work"

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 a variety of ways using the tabs at the bottom of the Sources window: Hierarchy, Library or Compile Order, see Figure 2.10.

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 and so on, see Figure 2.10.
CHAPTER 2. SELECTOR

Figure 2.10: Vivado IDE Viewing Environment

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 have 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 Figure 2.11.
CHAPTER 2. SELECTOR

Figure 2.11: 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 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 Figure 2.10, 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:

- In the Vivado **Flow Navigator**, click the **Add Sources** command (**Project Manager** option).

  ![Add Sources command](image)

  **Figure 2.12: Add Sources command**

- In the **Add Sources** dialog box, select **Add or create design sources** option to create the design source files in the project and click **Next**.
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.

- In the Create Source File dialog box, fill the file type, file name and file location on the following way:
  
  - File type: VHDL
  - File name: selector_rtl
  - File location: Local to Project
- Click OK to create a new source file (*selector_rtl.vhd*) and add it into your project (*modulator*).

- Now your source file will appear in the Add or Create Design Sources dialog box. Click Finish.

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

Please, rename Entity name to be *selector* and Architecture name to be *rtl*.

- Specify ports for the intended module as it is also shown on the following figure.
- Click **OK** and your source file should appear under the **Design Sources** in the **Sources** view in the **Project Manager** window, see Figure 2.18.

- Double-click on the created **selector_rtl.vhd** source file to see what the tool has created for us, see Figure 2.19.
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 a 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 `selector_rtl.vhd` from the project. To remove some file from the project, do the following:

1. Select the file that you want to remove.
2. Right-click on the selected file and choose **Remove File from Project...** option, see Figure 2.20.

[Figure 2.19: Automatically generated frequency_trigger_rtl.vhd source file]
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 Figure 2.21.

![Figure 2.21: Remove Sources dialog box](image)

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

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

- **Optional**: Open "Modulator" project (**modulator.xpr**) (if it is not already opened).
Create Selector Module

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

- In the **New File** dialog box, type the name of your source file (e.g. `selector_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.

- When you click **Save**, Vivado IDE will automatically open empty `selector_rtl.vhd` source file in Vivado text editor.

- Insert the **VHDL code** and add the `selector_rtl` module header.

- 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.

- In the Vivado **Flow Navigator** click the **Add Sources** command.

![Add Sources command](image1.png)

**Figure 2.22**: Add Sources command

- In the **Add Sources** dialog box, select **Add or create design sources** option to add the design source files into the project and click **Next**.

![Add Sources dialog box - Add or create design sources option](image2.png)

**Figure 2.23**: Add Sources dialog box - Add or create design sources option

- 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.
- In the Add Source Files dialog box, browse to the project working directory and select the *selector_rtl.vhd* source file.

- Click OK and the *selector_rtl.vhd* source file should appear in the Add or Create Design Sources dialog box.
CHAPTER 2. SELECTOR

Figure 2.26: Add or Create Design Sources dialog box - with added file

- Click Finish and your source file should appear under the Design Sources in the Sources view in the Project Manager window.

Figure 2.27: Vivado IDE Viewing Environment after module creation

Note: Double-click on the selector rtl (selector 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.
**Selected 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

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

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

-- Entity defines the interface of a module
-- Generics are static, they are used at compile time
-- Ports are updated during operation and behave like signals on a schematic or
-- traces on a PCB
-- Entity is a primary design unit
entity selector is
  generic(
    num_freqs_g : integer range 1 to 64 := 2; -- number of possible frequencies
    width_g : integer range 1 to 64 := 1; -- the number of bits used to represent amplitude value
  );
  port(
    -- input clock signal
    clk_i : in std_logic;
    -- different increments for different frequencies
    inc_i : in std_logic_vector(num_freqs_g*width_g - 1 downto 0);
    -- signal made for selecting frequency
    sel_i : in std_logic_vector(0 downto 0);
    -- output signal with appropriate increment value, depends on the sel_i state
    inc_o : out std_logic_vector(width_g - 1 downto 0)
  );
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, th and maybe for which technology
architecture rtl of selector is
  signal inc_s : std_logic_vector(inc_o'range) := (others => '0'); -- clock counter
begin
  -- Defines a sequential process
  -- Counts two different values depending on the sel_i
  process (sel_i)
  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_i input port
    wait until rising_edge(clk_i);
    if (unsigned(sel_i) < (inc_i'length)) then
      inc_s <= inc_i((to_integer(unsigned(sel_i))+1)*width_g-1 downto to_integer(unsigned(sel_i))*width_g);
    else
      inc_s <= (others => '0');
    end if;
  end process;
  inc_o <= inc_s;
end architecture;
```

2.5 Creating Test Bench

**Selector Test Bench**
- **Usage:** used to verify correct operation of the Selector module defined in the `selector_rtl.vhd` file

- **Test bench internal signals:**
  - `clk_s` : input clock signal
- **sel_s**: input signal used to select output signal increment value
- **inc_s**: output signal whose increment value depends on the sel_s signal state
- **inc_i_s**: calculates two possible increments values based on design setting

**Generics:**
- **board_setting_g**: defines board specific settings
- **design_setting_g**: defines design specific settings

**File name**: selector_tb.vhd

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

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

- **Optional**: Launch Vivado IDE (if it is not already launched).
- **Optional**: Open "Modulator" project (modulator.xpr) (if it is not already opened).

**Generate Selector Test Bench File**

- In the main Vivado IDE menu, click **File -> New File...** option to open Vivado text editor.
  - In the **New File** dialog box, type the name of your test bench file (e.g. selector_tb.vhd) in the **File name** field and choose to save it into your working directory, on the same place where you saved selector_rtl.vhd source file.
  - When you click **Save**, Vivado IDE will automatically open empty selector_tb.vhd source file in Vivado text editor.
  - Insert the VHDL code and add the selector_tb module header.
  - 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.
  - In the Vivado **Flow Navigator** click the **Add Sources** command.

![Add Sources command](image)

- In the **Add Sources** dialog box, select **Add or create simulation sources** option to add the simulation source files into the project and click **Next**.
- In the **Add or Create Simulation Sources** dialog box, click the + icon and select **Add Files...** option.

- In the **Add Source Files** dialog box, browse to the project working directory and select the `selector_tb.vhd` source file.
- Click OK and the selector_tb.vhd source file should appear in the Add or Create Simulation Sources dialog box.

- Click Finish and your test bench file should appear under the Simulation Sources / sim_1 in the Sources view, in the Project Manager window.
Figure 2.33: Vivado IDE Viewing Environment with added test bench file

*Note:* Double-click on the `selector_tb.tb` 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.

**Selector test bench:**

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library work;
use work.modulator_pkg.all;
entity selector_tb is
  generic(
    board_setting_g : board_setting_t := sozius_c; -- defines board specific settings
    design_setting_g : design_setting_t := sim_setting_c -- defines design specific settings
  );
end entity;
architecture tb of selector_tb is
  -- calculates from the design setting the actual used configuration
  constant inc_c : a1integer_t := calc_inc_f(board_setting_g, design_setting_g);
  signal inc_i_s : std_logic_vector(design_setting_g.freq_hz'length*design_setting_g.nco_width - 1 downto 0) := (others => '0');
  signal clk_s : std_logic := '0';
  signal sel_s : std_logic_vector(0 downto 0) := (others => '0');
  signal inc_s : std_logic_vector(design_setting_g.nco_width - 1 downto 0) := (others => '0');
begin
  inc_i_s <= conv_int_array_to_slv_f(inc_c, design_setting_g.nco_width);
  -- input clock signal
  signal clk_s : std_logic := '0';
  -- signal used to select output signal increment value
  signal sel_s : std_logic_vector(0 downto 0) := (others => '0');
  -- signal whose increment value depends on the sel_s state
  signal inc_s : std_logic_vector(design_setting_g.nco_width - 1 downto 0) := (others => '0');
  begin
    -- converts from array of ints to std_logic_vector
    inc_i_s <= conv_int_array_to_slv_f(inc_c, design_setting_g.nco_width);
```

2018/11/29 www.so-logic.net
-- Instantiation of device under test (DUT)
-- No component definition is necessary
-- Use keyword entity, work is the library

selector_i : entity work.selector(rtl) -- selector module instance
    generic map(
        width_g => design_setting_g.nco_width
    )
    port map(
        clk_i => clk_s,
        inc_i => inc_i_s,
        sel_i => sel_s,
        inc_o => inc_s
    );

clk_s <= not (clk_s) after 10 ns;
 sel_s(0) <= '1' after 200 ns;
end architecture;

Include *modulator_pkg.vhd* Source File into the Project

As you can see from the codes above, you must include *modulator_pkg.vhd* source file into your *modulator* project.

In the *modulator_pkg.vhd* file are defined generics that will be used in these files.

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 manage 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:

**Simulate with Vivado Simulator**

- In the Sources window, under the Simulation Sources / sim_1, select selector_tb file.

- In the Flow Navigator, under the Simulation, click on the Run Simulation button.

- Choose the only offered Run Behavioral Simulation option, and your simulation will start.
CHAPTER 2. SELECTOR

Figure 2.34: Run Behavioral Simulation option

- The tool will compile the test bench file and launch the Vivado simulator, see Figure 2.35.

Figure 2.35: 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.

Simulation Results

- Correct any errors before proceeding.

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

- Assuming no errors, your simulation results should look similar to the figure below.
Figure 2.36: Simulation Results

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

- **Optional**: If you want to restart and rerun simulation for specific time, see Figure 2.37.

Figure 2.37: 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: Selector Verification**.
Chapter 3

COUNTER

3.1 Description

Description

- **Usage**: This module is an up counter with user-selectable increment value. Its task will be to generate read addresses for the ROM where samples of the sine wave are stored. The step of the counting will be controlled by the Selector module, via inc_i port, and the output of the Counter module will be an input of the Digital Sine module.

- **Block diagram**:

![Block diagram](image)

Figure 3.1: Counter block diagram

- **Input ports**:
  - clk_i : input clock signal
  - inc_i : counter increment value

- **Output ports**:
  - count_o : current counter value

- **Generics**:
  - bits_g : the number of samples in one period of the signal
  - count_max_g : threshold value for counter

- **File name**: counter_rtl.vhd

3.2 Creating Module

Generate Counter 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:**

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity counter is
generic(
  bits_g : integer range 1 to 99 := 1; -- the number of samples in one period of the signal
  count_max_g : integer -- threshold value for counter
);
port(
  clk_i : in std_logic; -- input clock signal
  inc_i : in std_logic_vector(bits_g - 1 downto 0); -- counter increment value
  count_o : out std_logic_vector(bits_g - 1 downto 0) -- current counter value
);
end entity;
architecture rtl of counter is
begin
  counter_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_i input port
    wait until rising_edge(clk_i);
    -- to_unsigned function converts from integer type to unsigned integer type
    if (count_s = to_unsigned(count_max_g, count_s'length)) then
      count_s <= (others => '0'); -- counter reset
    else
      count_s <= count_s + unsigned(inc_i); -- counting
    end if;
  end process;
  count_o <= std_logic_vector(count_s);
end architecture;
```

### 3.3 Creating Test Bench

#### Counter Test Bench

- **Usage:** used to verify correct operation of the counter module defined in the `counter_rtl.vhd` file

- **Test bench internal signals:**
  - `clk_s`: input clock signal
  - `inc_s`: counter increment value
  - `count_s`: current counter value
- **Generics:**
  - \(\text{bits}_g\): the number of samples in one period of the signal
  - \(\text{count}_\text{max}_g\): threshold value for counter

- **File name:** counter\_tb.vhd

We will now create a new simulation set (\textit{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.

**Create New Simulation Set**

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

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

![Create Simulation Set option](image1)

- In the Create Simulation Set dialog box, enter a name for the new simulation set or leave \textit{sim\_2} as a name and click OK.

![Create Simulation Set dialog box](image2)

- In the Add or Create Simulation Sources dialog box, under the new \textit{sim\_2} simulation set, use Add Files... option to add the test bench file for the Counter module.

- In the Add Source Files dialog box, browse to the project working directory and select the counter\_tb.vhd test bench file.
- Click OK and *counter_tb.vhd* source file should appear in the Add or Create Simulation Sources dialog box.

- Click Finish and your test bench file should appear under the Simulation Sources / sim_2 in the Sources view, in the Project Manager window.

![Vivado IDE Viewing Environment with created new simulation set](image)

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

**Counter test bench:**

```vhdl
library ieee;
use ieee.std_logic_1164.all;
library work;
use work.modulator_pkg.all;
entity counter_tb is
  -- Use lower values for generics to speed up simulation time
  generic(
    bits_g : integer range 1 to 99 := 4; -- the number of samples in one period of the signal
    count_max_g : integer := 12 -- threshold value for counter
  );
end entity;
architecture tb of counter_tb is
  signal clk_s : std_logic := '1'; -- input clock signal
  signal inc_s : std_logic_vector(bits_g - 1 downto 0) := X"3"; -- counter increment value
  signal count_s : std_logic_vector(bits_g - 1 downto 0) := (others => '0'); -- current counter value
begin
  -- Instantiation of device under test (DUT)
  -- No component definition is necessary
  -- Use keyword entity, work is the library
  counter_i : entity work.counter(rtl) generic map(
    bits_g => bits_g,
    count_max_g => count_max_g
  );
end architecture;
```
CHAPTER 3. COUNTER

```vhdl
port map (
    clk_i => clk_s,
    inc_i => inc_s,
    count_o => count_s
);
```

```vhdl
clk_s <= not (clk_s) after 5 ns; -- generates input clock signal
end architecture;
```

3.4 Simulating

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

**Make Active New Simulation Set**

- In the Sources window, under the Simulation Sources, select new sim_2 simulation set, right-click on it and choose Make Active option.

![Figure 3.5: Make Active option](image)

**Simulate with Vivado Simulator**

- In the Flow Navigator, under the Simulation, click Run Simulation command.

- Choose the only offered Run Behavioral Simulation option and your simulation will start.

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

- Correct any errors before proceeding.

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

**Simulation Results**

- Assuming no errors in the Vivado simulator command line, your simulation result should look similar to figure below.

www.so-logic.net 2018/11/29 49
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

Description
- 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 below.

![Figure 4.1: Sine-package description](image)

The formula for calculating the sine wave shown on the Figure 4.1.

Sine Signal Formula

\[ \sin \left( \frac{2\pi i}{N} \right) \ast (2^{\text{width}_c - 1} - 1) + 2^{\text{width}_c - 1} - 1, N = 2^{\text{depth}_c} \]

- **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:

\[ \sin\left(\frac{2\pi i}{N}\right) \] is telling us that the signal is periodic, with \(2\pi\) period; \(i\) is the current sample value (from 0 to 255) and \(N\) is the number of samples in one period of the signal.

\[ 2^{\text{width}} - 1 \] is telling us that the amplitude of the sine signal is 2047.

\[ 2^{\text{width}} - 1 \] 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:**

```vhdl
-- Make reference to libraries that are necessary for this file:
-- the first part is a symbolic name, the path is depending of the tools
-- the second part is a package name
-- the third part includes all functions from that package
-- Better for documentation would be to include only the functions that are necessary
library ieee;
use ieee.math_real.all;
use ieee.std_logic_1164.all;
use ieee.numeric_std.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 declarations
    type a1integer_t is array (natural range <>) of integer;
    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, sozius, undefined); -- enumeration type
    type has_diff_clk_t is (yes, no); -- enumeration type for differential clock buffer use
    type a1real_t is array (natural range <>) of real;

    -- defines board specific settings
    type board_setting_t 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 of the board (in Hz)
        has_diff_clk : has_diff_clk_t; -- specifies if board has differential clock or not
    end record;
    constant lx9_c : board_setting_t := (lx9, 100000000.0, no); -- Spartan-6
    constant zedboard_c : board_setting_t := (zedboard, 100000000.0, no); -- Zynq-7000
    constant ml605_c : board_setting_t := (ml605, 200000000.0, yes); -- Virtex-6
    constant kc705_c : board_setting_t := (kc705, 200000000.0, yes); -- Kintex-7
    constant microzed_c : board_setting_t := (microzed, 33333333.3, no); -- Microzed
    constant sozius_c : board_setting_t := (sozius, 50000000.0, no); -- Sozius
    constant undefined_c : board_setting_t := (undefined, 50000000.0, no); -- not defined

    -- defines design specific settings
    type design_setting_t is record
        freq_hz : a1real_t(0 to 1); -- frequencies for the PWM signal, specified in Hz
        lut_depth : integer range 0 to 31; -- the number of samples in one period of the signal
        lut_width : integer range 0 to 63; -- the number of bits used to represent amplitude value
        nco_width : integer range 0 to 63; -- number of bits used for numerically controlled oscillator
    end record;
    constant audio_setting_c : design_setting_t :=
        (freq_hz => (880.0, 440.0),
         lut_depth => 12,
         lut_width => 8,
         nco_width => 24);

end modulator_pkg;
```
-- defines led specific settings
constant led_setting_c : design_setting_t :=
( freq_hz => (1.0, 3.5),
  lut_depth => 12,
  lut_width => 16,
  nco_width => 31 );

-- defines simulation specific settings
constant sim_setting_c : design_setting_t :=
( freq_hz => (1000.0, 10000.0),
  lut_depth => 12,
  lut_width => 16,
  nco_width => 31 );

-- defines simulation specific settings
constant sim_setting1_c : design_setting_t :=
( freq_hz => (1000.0, 10000.0),
  lut_depth => 12,
  lut_width => 16,
  nco_width => 31 );

-- calculates from the design setting the actual used configuration
function calc_inc_f
( constant board_c : board_setting_t;
  constant design_c : design_setting_t )
return a1integer_t is
  variable inc_v : a1integer_t(design_c.freq_hz'range);
begin
  for i in design_c.freq_hz'range loop
    inc_v(i) := integer(round(real(2 ** design_c.nco_width) * design_c.freq_hz(i) / board_c.fclk));
  end loop;
return inc_v;
end function;

-- init_sin_f function declaration
function init_sin_f
( constant depth_c : in integer; -- is the number of samples in one period of the signal (2^8=256)
  constant width_c : in integer -- is the number of bits used to represent amplitude value (2^12=4096) )
return a1integer_t;

-- converts from array of ints to std_logic_vector
function conv_int_array_to_slv_f
( constant int_array_i : a1integer_t;
  constant width_c : integer )
return std_logic_vector;
end package;

-- in the package body will be description of the function defined before
package body modulator_pkg is

-- calculates from the design setting the actual used configuration
function calc_inc_f
( constant board_c : board_setting_t;
  constant design_c : design_setting_t )
return a1integer_t is
  variable inc_v : a1integer_t(design_c.freq_hz'range);
begin
  for i in design_c.freq_hz'range loop
    inc_v(i) := integer(round(real(2 ** design_c.nco_width) * design_c.freq_hz(i) / board_c.fclk));
  end loop;
return inc_v;
end function;

-- init_sin_f function definition
function init_sin_f
( constant depth_c : in integer;
  constant width_c : in integer )
return a1integer_t;
begin
  variable init_arr_v : a1integer_t(0 to (2 ** depth_c - 1));
  for i in 0 to ((2 ** depth_c)- 1) loop -- calculates amplitude values
    init_arr_v(i) := integer(round(sin((math_2_pi / real(2 ** depth_c)) * real(i)) * 
                                   (2 ** width_c - 1) / (2 ** depth_c - 1)) * (2 ** width_c - 1));
  end loop;
return init_arr_v;
end function;
(\text{real}(2 \times (\text{width}_c - 1) - 1.0)) \times \text{integer}(2 \times (\text{width}_c - 1) - 1);

\text{return \ init\_arr\_v;}
\end{function}

\text{-- converts from array of ints to std\_logic\_vector}
\function\ conv\_int\_array\_to\_slv\_f
\begin{func}
\text{constant int\_array\_i : all\_integer\_t;}
\text{constant width\_c : integer}
\end{func}
\text{return std\_logic\_vector is}
\begin{variable}
\text{variable out\_slv\_v : std\_logic\_vector(int\_array\_i'length\times width\_c-1 \ downto \ 0) := (others => '0');}
\text{begin}
\text{for i in int\_array\_i'range loop}
\text{out\_slv\_v((i+1)\times width\_c-1 \ downto i\times width\_c) := std\_logic\_vector(to\_unsigned(int\_array\_i(i), width\_c));}
\text{end loop;}
\text{return out\_slv\_v;}
\end{variable}
\end{function}
\end{package body}

\textit{Note}: All the information about creating the sine package, you can also find in the \textbf{Lab 6: "Creating Sine Package"}.  

\textbf{Lab 6: "Creating Sine Package"}
Chapter 5

DIGITAL SINE

5.1 Description

Description

- **Usage**: This module will generate a 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**:

![Digital Sine block diagram](image)

- **Input ports**:
  - `clk_i`: input clock signal
  - `addr_i`: address value for the sine waveform ROM

- **Output ports**:
  - `data_o`: 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:

-- 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.numeric_std.all;
library work;
use work.modulator_pkg.all;

-- Entity defines the interface of a module
-- Generics are static, they are used at compile time
-- Ports are updated during operation and behave like signals on a schematic or
-- traces on a PCB
-- Entity is a primary design unit
entity 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_i : in std_logic; -- input clock signal
    addr_i : in std_logic_vector(depth_g-1 downto 0); -- address value for the sine waveform ROM
    data_o : out std_logic_vector(width_g-1 downto 0) -- current amplitude value of the sine signal
  );
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, th and maybe for which technology
architecture rtl of sine is
  -- Between architecture and begin is declaration area for types, signals and constants
  -- Everything declared here will be visible in the whole architecture
  constant data_c : integer_t := init_sin_f(depth_g, width_g); -- returns sine amplitude value
  signal addr_s : integer range 0 to 2 ** depth_g - 1 := 0; -- amplitude counter
  signal data_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
    -- Replaces the sensitivity list
    -- Suspends evaluation until an event occurs
    -- In our case event we are waiting for is rising edge on the clk_i input port
    wait until rising_edge(clk_i);
    -- converts addr_i from std_logic_vector type to integer type
    addr_s <= to_integer(unsigned(addr_i));
    -- converts data_c from integer type to std_logic_vector type
    data_s <= std_logic_vector(to_unsigned(data_c(addr_s), width_g)); -- fetch amplitude
  end process;
  data_o <= data_s;
end architecture;

Note: All the information about creating the Digital Sine module, you can also find in the Lab 7: "Creating Digital Sine Module".
Chapter 6

PWM

6.1 Description

Description

- **Usage**: This module will generate an PWM signal modulated using the digital sine wave from the Digital Sine module. This module will implement the Finite State Machine (FSM), that will be used to 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.

- **Block diagram**:

![PWM block diagram](image)

Figure 6.1: PWM block diagram

**FSM State Diagram**

![FSM state diagram](image)

Figure 6.2: FSM state diagram
- **Input ports:**
  - `clk_i`: input clock signal
  - `value_i`: current amplitude value of the sine signal

- **Output ports:**
  - `pwm_o`: pulse width modulated signal

- **Generics:**
  - `width_g`: the number of bits used to represent amplitude value

---

### 6.2 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:**

```vhdl
-- 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.numeric_std.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 pwm is
  generic(
    width_g : integer range 1 to 99 := 12 -- the number of bits used to represent amplitude value
  );
  port(  
    clk_i : in std_logic; -- input clock signal
    value_i : in std_logic_vector(width_g-1 downto 0); -- current amplitude value of the sine signal
    pwm_o : out std_logic -- pulse width modulated signal
  );
end entity;

-- Architecture is a secondary design unit and describes the functionality of the module
-- The 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 pwm is
begin
  -- fsm_state_p models state register and next-state logic
  fsm_state_p : process
  begin
    wait until rising_edge(clk_i);
    count_s <= count_s + 1;
    pwm_o <= '0';
    case state is
    when load_st, pwm_high_st, pwm_low_st =>
  end case;
  end process
end; -- architecture
```

---

1. **Input ports:**
   - `clk_i`: input clock signal
   - `value_i`: current amplitude value of the sine signal

2. **Output ports:**
   - `pwm_o`: pulse width modulated signal

3. **Generics:**
   - `width_g`: the number of bits used to represent amplitude value

---

**File name:** `pwm_rtl.vhd`
6.3 Creating Test Bench

PWM Test Bench

- **Usage**: used to verify correct operation of the PWM module defined in the `pwm_rtl.vhd` file

- **Test bench internal signals**:
  
  - `clk_s`: input clock signal
  - `count_s`: counts samples in one period of the sine signal
  - `data_s`: current amplitude value of the sine signal
  - `pwm_s`: pwm signal
  - `sim_end_s`: end of simulation process

- **Generics**:
  
  - `board_setting_g`: defines board specific settings
  - `design_setting_g`: defines simulation specific settings

- **File name**: `pwm_tb.vhd`

Create New Simulation Set

We will now create a new simulation set (`sim_3`) 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**:

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library work;
```
use work.modulator_pkg.all;

entity pwm_tb is
  generic(
    -- defines board specific settings
    board_setting_g : board_setting_t := sozius_c;
    -- defines simulation specific settings
    design_setting_g : design_setting_t := sim_setting_c
  );
end entity;

architecture tb of pwm_tb is
  -- period of input clock signal
  constant clock_period_c : time := 1000000000.0 / board_setting_g.fclk * 1 ns;
  -- max number of samples in one period of the sine signal
  constant counter_max_c : unsigned(design_setting_g.lut_width - 1 downto 0) := (others => '1');
  signal clk_s : std_logic := '0';
  signal count_s : unsigned(design_setting_g.lut_width - 1 downto 0) := (others => '0');
  signal data_s : unsigned(design_setting_g.lut_width - 1 downto 0) := (others => '0');
  signal pwm_s : std_logic := '0';
  signal sim_end_s : boolean := false;
begin
  ramp_p : process
  begin
    wait until rising_edge(clk_s);
    if (count_s = 2 ** design_setting_g.lut_width - 1) then
      count_s <= (others => '0');
      if (data_s = counter_max_c) then
        wait for 20 us;
        data_s <= (others => '0');
        sim_end_s <= true;
      else
        data_s <= data_s + 1;
      end if;
    else
      count_s <= count_s + 1;
    end if;
  end process;

  pwm_i : entity work.pwm(rtl) port map (clk_i => clk_s,
    value_i => std_logic_vector(data_s),
    pwm_o => pwm_s);

  clk_p : process
  begin
    while not (sim_end_s) loop
      clk_s <= not (clk_s);
      wait for clock_period_c/2;
    end loop;
  end process;
end architecture;

6.4 Simulating

Simulate with Vivado Simulador

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

- You can start your simulation (see Chapter 3.4 Simulating).
- Simulate your design for 5 ms (see Chapter 2.6 Simulating – step 9.).

Simulation Results
- Assuming no errors, your simulation result should look similar to figure below.

Figure 6.3: Simulation Results

*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 7

MODULATOR

7.1 Description

Description

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

- **Block diagram:**

![Modulator block diagram](image)

Figure 7.1: Modulator block diagram

- **Input ports:**
  - `clk_i`: input clock signal
  - `inc_i`: different increments for different frequencies
  - `sel_i`: signal made for selecting frequency

- **Output ports:**
  - `pwm_o`: pulse width modulated signal

- **Generics:**
  - `board_setting_g`: defines board specific settings
• design_setting_g : defines led specific settings

- File name: modulator_rtl.vhd

7.2 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:

-- 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_textio.all;
use ieee.numeric_std.all;
library work;
use work.modulator_pkg.all;

-- Entity defines the interface of a module
-- Generics are static, they are used at compile time
-- Ports are updated during operation and behave like signals on a schematic or
-- traces on a PCB
-- Entity is a primary design unit
design_setting_g : design_setting_t
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, th and maybe for which technology
architecture rtl of modulator is

-- Between architecture and begin is declaration area for types, signals and constants
-- Everything declared here will be visible in the whole architecture
constant counter_max_c : std_logic_vector(design_setting_g.nco_width - 1 downto 0) := (others => '1');
signal addr_s : std_logic_vector(design_setting_g.nco_width - 1 downto 0) := (others => '0');
signal inc_s : std_logic_vector(design_setting_g.nco_width - 1 downto 0) := (others => '0');
signal data_s : std_logic_vector(design_setting_g.lut_width - 1 downto 0) := (others => '0');

begin

selector_i : entity work.selector(rtl) -- selector module instance
generic map(
width_g => design_setting_g.nco_width
)
port map(
clk_i => clk_i,
inc_i => inc_i,
inc_o => inc_s,
addr_o => addr_s
);

counter_i : entity work.counter(rtl) -- counter module instance
generic map(
bits_g => design_setting_g.nco_width,
count_max_g => to_integer(unsigned(counter_max_c))
)
port map (}
CHAPTER 7. MODULATOR

7.3 Creating Test Bench

Modulator Test Bench

- **Usage**: used to verify correct operation of the Modulator module defined in the `modulator_rtl.vhd` file

- **Test bench internal signals**:
  - `clk_s`: input clock signal
  - `sel_s`: signal made for selecting frequency
  - `pwm_s`: pulse width modulated signal

- **Generics**:
  - `board_setting_g`: defines board specific settings
  - `design_setting_g`: defines simulation specific settings
  - `sim_end_g`: defines duration of the simulation process

- **File name**: `modulator_tb.vhd`

Create New Simulation Set

We will now create a new simulation set (`sim_4`) 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**:

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library work;
use work.modulator_pkg.all;
entity modulator_tb is
generic;
```
-- defines board specific settings
board_setting_g : board_setting_t := sozius_c;
-- defines simulation specific settings
design_setting_g : design_setting_t := sim_setting_c;
-- defines duration of the simulation process
sim_end_g := time := 10 ns);
end entity;

architecture tb of modulator_tb is

-- period of input clock signal
custom_clock_period_c : time := 1000000000.0 / board_setting_g.fclk * 1 ns;
custom inc_c := calc_inc_f(board_setting_g, design_setting_g);
signal inc_s : std_logic_vector(design_setting_g.freq_hz'length*design_setting_g.nco_width - 1 downto 0) := (others => '0');
signal clk_s : std_logic := '1'; -- input clock signal
signal sel_s : std_logic_vector(custom_inc_c.nco_width - 1 downto 0); -- signal made for selecting frequency
signal pwm_s : std_logic := '0'; -- pulse width modulated signal
begin

-- converts from array of ints to std_logic_vector
inc_s <= conv_int_array_to_slv_f(inc_c, design_setting_g.nco_width);
modulator_i : entity work.modulator -- modulator module instance
generic map(
   design_setting_g => design_setting_g,
   board_setting_g => board_setting_g
) port map(
   clk_i => clk_s,
   inc_i => inc_s,
   sel_i => sel_s,
   pwm_o => pwm_s
);

clk_p: process -- generates input clock signal
begin
   while (now < sim_end_g) loop
      clk_s <= not (clk_s);
      wait for custom_clock_period_c/2;
   end loop;
   wait;
end process;

7.4 Simulating

Simulate with Vivado Simulator

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

- You can start your simulation (see Chapter 3.4 Simulating).

- Simulate your design for 15 ms (see Chapter 2.6 Simulating – step 9.).

Simulation Results

- Assuming no errors, your simulation result should look similar to figure below.
CHAPTER 7. MODULATOR

Figure 7.2: Simulation Results

*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"](http://example.com). 

7.5 Synthesis

7.5.1 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.

7.5.2 Run Synthesis

To synthesize your design, follow these steps:

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

- In the Vivado Flow Navigator, click Run Synthesis command (Synthesis option) and wait for task to be completed, see Figure 8.3.

*Note*: You can monitor the Synthesis progress in the bar in the upper-right corner of the Vivado IDE.
- After the synthesis is completed, the **Synthesis Completed** dialog box will appear, see Figure 8.4.

![Synthesis Completed dialog box](image)

**Figure 7.4: 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.

- Select **Open Synthesized Design** and click **OK**, see Figure 8.4.

- Make sure that **Default Layout** option is selected from the view layout pull-down menu in the main toolbar, see Figure 8.5.

![Default Layout option](image)

**Figure 7.5: Default Layout option**

### 7.5.3 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 Figure 8.6.
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 Figure 7.7.

- **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 Figure 8.8. Timing Reports can be generated at any point after synthesis.

  - Tcl command equivalent to this option is: `report_timing_summary`
CHAPTER 7. MODULÄTOR

Figure 7.8: Timing Summary Report

- **Report Clock Networks**: Generates a clock tree for the design, see Figure 8.9. 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`

Figure 7.9: Clock Networks Report

- **Report Clock Interaction**: Verifies constraint coverage on paths between clock domains. This option uses an interclock 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 7.10: 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 Figure 8.11, 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.

![Figure 7.11: DRC Report](image)

• **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 Figure 8.12. 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.

![Figure 7.12: Noise Report](image)

• **Report Utilization**: Generates a graphical version of the Utilization Report, see Figure 8.13.

![Figure 7.13: Utilization Report](image)

• **Report Power**: Provides detailed power and thermal analysis reports that can be customized for the power supply and application environment, see Figure 8.14. 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`
>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 8.5.5 Schematic View.

7.5.4 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:

- Select the Reports tab at the bottom of the IDE, see Figure 8.15.

![Figure 7.15: Reports tab](image)

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

- In the Reports tab, double-click on the Vivado Synthesis Report to open it and examine the report contents, see Figure 8.15.

*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 Figure 8.16.

Figure 7.16: Vivado Synthesis Report

- When finished, close the report.

- In the Reports tab, double-click on the Utilization Report to examine its content, see Figure 8.15.

Utilization Report - describes the amount of device resources that the synthesized design is expected to use, see Figure 8.17.
7.5.5 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:

- Select one or more logic elements in an open window, such as the Netlist window.
- In the Flow Navigator / Synthesis / Synthesized Design click the Schematic command, see Figure 8.18.
- After few seconds, **Schematic** window will show up, and your design should look similar to the design shown on the Figure 8.19.

![Modulator Schematic View](image)

**Figure 7.19: Modulator 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 displayed5 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:

- **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 objects are in another window and you want to redraw the display around those selected objects

- **Auto-fit Selection** - Automatically redrews the Schematic window around newly selected objects. This mode can be enabled or displayed
• **Toggle autohide pins for selected cells** - 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 8

MODULATOR SOZIUS WRAPPER

8.1 Description

Description

- **Usage**: This module will be used to target Sozius development board.

Sozius 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 one main VHDL model, `sozius_xz2_modulator_vio_rtl.vhd` model, which will be also the top model of the design.

About Sozius Development Board

The main component of the Sozius 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 Sozius 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.

Modulator Sozius Wrapper Structure

In our design we will program PL part of the Zynq FPGA with `sozius_xz2_modulator_vio_rtl.vhd` model.

PS part is also required to generate clock signal for the Modulator design, since the only reference clock source on the Sozius board is connected to the PS part of the Zynq FPGA.

Properly configured PS part is described in the `sozius_xz_lab_ps_bd` component. The `sozius_xz_lab_ps_bd` component will be contained in the `sozius_xz2_modulator_vio_rtl.vhd` model.
8.2 Creating Module

As we already said, in our design we will program PL part of the Zynq FPGA with `sozius_xz2_modulator_vio_rtl.vhd` model. Since existing LEDs and switches on the Sozius 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 detailed information about ILA and VIO cores you can find in the Chapter 10 "Debugging Design" of this tutorial.

**Instantiate ILA and VIO Core into the Design**

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.

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.

All these information you can also find in the Chapter 10 "Debugging Design" of this tutorial, where both flows are explained in detail.
Instantiate VIO Core into the Project

- Instantiate VIO core into our design using steps for VIO core instantiation, explained in the Sub-chapter 10.1 "Inserting ILA and VIO Cores into Design" of this tutorial. Use the same core customizations as it is explained in this sub-chapter:
  
  1. In the VIO (Virtual Input/Output) (3.0) window, enter *vio_core_name* (*vio_core*) in the Component Name field
  2. 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
  3. 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
  4. 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
  5. Click OK

After VIO core generation, your VIO core should appear in the Sources window, see Figure 8.2.

![Figure 8.2: Source tab with generated VIO core](image)

Instantiate ILA Core into the Project

ILA core will be instantiated into our design using "Netlist Insertion Debug Probing Flow", explained in the Sub-chapter 10.1 of this tutorial.

We will use *mark_debug* attribute to add debug nets (*pwm_s*, *sel_s* and *cc_count_s*) to our HDL file (*sozius_xz2_modulator_vio_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 *cc_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 that will measure the duration of the PWM pulse, see Figure 8.1. Both of these instances, plus ILA and VIO core instances will be included within *sozius_xz2_modulator_vio_rtl.vhd* VHDL model.

Complete sozius_xz2_modulator_vio_rtl.vhd Source File

- To create and add *sozius_xz2_modulator_vio_rtl.vhd* source file 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.

```vhdl
-- 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.numeric_std.all;
library work;
```

www.so-logic.net 2018/11/29 79
use work.modulator_pkg.all;
use work.sozius_components_package.all;

entity sozius_xz2_modulator_vio is
  generic(
    -- version number of PL for read back from PS
    hw_version_g : std_logic_vector(31 downto 0) := x"19980418";
    -- User defined settings for the pwm design
    board_setting_g : board_setting_t := sozius_c;
    design_setting_g : design_setting_t := led_setting_c
  );
  port(
    -- ethernet phy reset, must be high necessary for operation !!!
    pl_phy_reset_n_o : out std_logic := '1'; --! reset of ethernet
    -- ps io
    ps_ddr3_addr : in out std_logic_vector(14 downto 0);
    ps_ddr3_ba : in out std_logic_vector(3 downto 0);
    ps_ddr3_cas_n : in out std_logic;
    ps_ddr3_ck_n : in out std_logic;
    ps_ddr3_ck_p : in out std_logic;
    ps_ddr3_cke : in out std_logic;
    ps_ddr3_cs_n : in out std_logic;
    ps_ddr3_dm : in out std_logic_vector( 3 downto 0);
    ps_ddr3_dq : in out std_logic_vector(31 downto 0);
    ps_ddr3_dqs_n : in out std_logic_vector( 3 downto 0);
    ps_ddr3_dqs_p : in out std_logic_vector( 3 downto 0);
    ps_ddr3_odt : in out std_logic;
    ps_ddr3_ras_n : in out std_logic;
    ps_ddr3_reset_n : in out std_logic;
    ps_ddr3_we_n : in out std_logic;
    ps_ddr_vrn : in out std_logic;
    ps_ddr_vrp : in out std_logic;
    ps_clk_i : in out std_logic;
    ps_por_n_i : in out std_logic;
    ps_srst_n_i : in out std_logic;
    ps_phy_mdc_io : in out std_logic;
    ps_phy_mdio_io : in out std_logic;
    ps_phy_rx_clk_io : in out std_logic;
    ps_phy_rx_ctrl_io : in out std_logic;
    ps_phy_rxd_io : in out std_logic_vector(3 downto 0);
    ps_phy_tx_clk_io : in out std_logic;
    ps_phy_tx_ctrl_io : in out std_logic;
    ps_phy_txd_io : in out std_logic_vector(3 downto 0);
    ps_i2c_scl_io : in out std_logic;
    ps_i2c_sda_io : in out std_logic;
    ps_led_error_n_io : in out std_logic;
    ps_led_front_n_io : in out std_logic_vector(1 downto 0);
    ps_led_sdcard_n_io : in out std_logic;
    ps_sw0_a_io : in out std_logic;
    ps_sw0_b_io : in out std_logic;
    ps_sw1_a_io : in out std_logic;
    ps_sw1_b_io : in out std_logic;
    ps_sw2_a_io : in out std_logic;
    ps_sw2_b_io : in out std_logic;
    ps_sw3_a_io : in out std_logic;
    ps_sw3_b_io : in out std_logic;
    ps_uart_rx_io : in out std_logic;
    ps_uart_tx_io : in out std_logic;
    ps_qspi_cs_n_io : in out std_logic;
    ps_qspi_data_io : in out std_logic_vector(3 downto 0);
    ps_qspi_clk_io : in out std_logic;
    ps_sdio_clk_io : in out std_logic;
    ps_sdio_data_io : in out std_logic_vector(3 downto 0);
    ps_usb_clk_io : in out std_logic;
    ps_usb_data_io : in out std_logic_vector(7 downto 0);
    ps_usb_dir_io : in out std_logic;
    ps_usb_nxt_io : in out std_logic;
    ps_usb_stp_io : in out std_logic
  );
end entity;

architecture structural of sozius_xz2_modulator_vio is

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

attribute mark_debug : string;

constant inc_c : a1integer_t := calc_inc_f(board_setting_g, design_setting_g);

signal inc_s : std_logic_vector(design_setting_g.freq_hz'length*design_setting_g.nco_width - 1 downto 0) := (others => '0');
signal cc_count_s : std_logic_vector(31 downto 0) := (others => '0');
signal pwm_s : std_logic;
signal sel_s : std_logic_vector(0 downto 0);

attribute mark_debug of cc_count_s : signal is "true";
attribute mark_debug of pwm_s : signal is "true";
attribute mark_debug of sel_s : signal is "true";

end architecture;
-- declaration for fixed signal PL to PS
signal pl_clk0_s : std_logic;
signal pl_reset_n_s : std_logic;

-- ps signals
signal ps_mio_s : std_logic_vector(53 downto 0);

-- 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

  inc_s <= conv_int_array_to_slv_f(inc_c, design_setting_g.nco_width);

  -- modulator module instance
  modulator_i: entity work.modulator(rtl)
    generic map(
      board_setting_g => board_setting_g,
      design_setting_g => design_setting_g
    )
    port map(
      clk_i => pl_clk0_s,
      inc_i => inc_s, -- inc_c,
      sel_i => sel_s,
      pwm_o => pwm_s
    );

  -- counter for measuring the duration of the high pulse of the PWM signal
  measurement_counter_p: process
  begin
    wait until rising_edge(pl_clk0_s);
    if (pwm_s = '0') then
      cc_count_s <= (others => '0');
    else
      cc_count_s <= std_logic_vector(unsigned(cc_count_s) + 1);
    end if;
  end process;

  -- vio_core component instance
  vio_i: vio_core
  port map (  
    clk => pl_clk0_s,
    probe_in0(0) => pwm_s,
    probe_out0 => sel_s
  );

  -- instance of processor system PS
  sozius_xz_lab_ps_bd_i: component sozius_xz_lab_ps_bd
    port map (  
      ddr3_addr => ps_ddr3_addr,
      ddr3_ba => ps_ddr3_ba,
      ddr3_cas_s => ps_ddr3_cas_s,
      ddr3_ck_n => ps_ddr3_ck_n,
      ddr3_ck_p => ps_ddr3_ck_p,
      ddr3_cwe => ps_ddr3_cwe,
      ddr3_cs_n => ps_ddr3_cs_n,
      ddr3_dm => ps_ddr3_dm,
      ddr3_dq => ps_ddr3_dq,
      ddr3_dqm => ps_ddr3_dqm,
      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 => '0',
      pl_uart_1_txd => open,
      pl_spi_0_io0_i => '0',
      pl_spi_0_io0_o => open,
      pl_spi_0_io1_i => '0',
      pl_spi_0_io1_o => open,
      pl_spi_0_sck_i => '0',
      pl_spi_0_sck_o => open,
      pl_spi_0_ss1_o => open,
      pl_spi_0_ss2_o => open,
      pl_spi_0_ss_i => '0',
      pl_uart_0_rxd => '0',
      pl_uart_0_txd => open,
      pl_spi_1_io0_i => '0',
      pl_spi_1_io0_o => open,
      pl_spi_1_io1_i => '0',
      pl_spi_1_io1_o => open,
      pl_spi_1_sck_i => '0',
      pl_spi_1_sck_o => open,
      pl_spi_1_ss1_o => open,
      pl_spi_1_ss2_o => open,
      pl_spi_1_ss_i => '0',
  );
Complete sozius_components_package.vhd Source File

PS part of the Zynq FPGA is also required to generate clock signal for the Modulator Sozius design.

Properly configured PS part is described in the sozius_xz Lab_ps_bd component of the sozius_xz2_modulator_vio_rtl.vhd model.

The complete sozius_xz Lab_ps_bd component declaration you can find in the sozius_components_package.vhd package declaration.

sozius_components_package.vhd:

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arITH.all;
use ieee.std_logic_unsigned.all;

package sozius_components_package is

end architecture;
```

82 2018/11/29 www.so-logic.net
component sozius_xz_lab_ps_bd is
port (
    ps_clk0 : out std_logic;
    ps_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;
    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
);    end component;
end package;

Note: Don’t forget to add sozius_components_package.vhd package into the project!

Configure the Zynq PS Part to work on Sozius Development Board

Finally, we must configure the Zynq PS part to work on Sozius 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 Sozius board is sozius_xz_lab_ps_bd.tcl.

This Tcl script file is too long to be shown in the tutorial, so ask your instructor for details.

Execute Tcl File in the Vivado IDE

- Next step is to execute the sozius_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>/sozius_xz_lab_ps_bd.tcl

Where <path> stands for the full path to the folder where the sozius_xz_lab_ps_bd.tcl Tcl file is stored.

---

**Figure 8.3: Tcl Console window**

**Block Diagram after Vivado Tcl Script Execution**

After Vivado has finished with the Tcl script execution, a created block diagram containing Zynq PS will be visible in the Vivado IDE.

---

**Figure 8.4: Block diagram of Zynq PS configured to run on Sozius board**

**Creating XDC File**

- Now is the time to create and add constraints file for the Sozius board, sozius_xz_modulator_vio.xdc.

To create and add constraints file, please use steps from the Sub-chapter 9.1 "Creating XDC File", where it is in detail explained in paragraph "Creating a XDC File using Vivado Text Editor".

The complete sozius_xz_modulator_vio.xdc constraints file you can find in the Sub-chapter 9.1 "Creating XDC File".

**Synthesize Design**

- In the Vivado Flow Navigator, click Run Synthesis command and wait for task to be completed.

- After the synthesis is completed, choose Open Synthesized Design option in the Synthesis Completed dialog box.
CHAPTER 8. MODULATOR SOZIUS WRAPPER

Instantiate ILA Core into the Project

- 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 `cc_count_s` nets to ILA core, as it is explained in steps 20 - 33 in the Sub-chapter 10.1 "Inserting ILA and VIO Cores into Design".

Note: Pay attention to enable Capture control feature for ILA in step 31!

**Note:** All the information about Sozius development board, ILA and VIO core instantiation, defining structure of the ARM-based processor system for Sozius development board, configuring the Zynq PS part to work on Sozius development board, and many more things, you can also find in the Lab 11: "Creating Modulator Sozius Wrapper Module".
Chapter 9

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 7.5 Synthesis). After design implementation is completed successfully, we must generate bitstream file and use it to program target FPGA device.

9.1 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 Constraints** - 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.

**XDC 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 proprietary physical constraints

**XDC Constraints Properties**

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

Generating an XDC File

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 "Sozius (Zynq-7000) 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 once. 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 7.5.2 Run Synthesis.

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

- 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 Figure 9.1.

![Figure 9.1: I/O Planning option](image)

This will change the layout from the Default view to the I/O Planning view, see Figure 9.2.
The main window of the I/O Planning view displays the package view of the Sozius 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.

- In the I/O Ports tab, click Expand All option, or expand each bus separately to see all I/O Ports of your design, see Figure 9.3.

Note that all of the pins in this view have an assigned location.

Grey icons indicate unplaced ports, while yellow icons indicate placed ports. On the Figure 9.3 we can see that all I/O ports are coloured yellow, since all 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.

The same thing you can see as you drag across the package view (see Figure 9.2), 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

- To connect your logical with your physical ports, select one scalar port and find in the user guide for the socius development board to which pin location you would like to connect that scalar port.

- In the I/O Ports tab, click on the target scalar port Package Pin column and choose a pin location to connect the target port.

- Click on the target scalar port I/O Std column and change the I/O standard from default to target (usually LVCMOS18 and LVCMOS33) standard.

LVCMOS18 is a low voltage CMOS I/O standard using 1.8V 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.

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.

- Leave all the target port options unchanged, because they are default values.

Note: After assigning pin location and I/O standard for target port, we will notice that I/O Port Properties window popped up. This is the another way to change port properties, see Figure 9.4. If you want to apply some changes that you made, just click the Automatically update button.

![Figure 9.4: I/O Port Properties window](image)

- Repeat these configuration steps for the remaining ports using the pin locations and necessary I/O standards information.
All the information about assigned pin locations and I/O standards of our design you can find extracted in the `sozius_xz_modulator_vio.xdc` constraints file in the text below.

- When you are finished with the placement constraints, click `File -> Constraints -> Save As...`

- In the `Save Constraints` dialog box, type the name of the constraints file in the `File name` field, for example `modulator_sozius`, see Figure 9.5.

![Figure 9.5: Save Constraints dialog box](image)

- In the `Save Constraints As` dialog box, type the name of the constraint set in the `New Constraints set name` field, for example `modulator_sozius`, see Figure 9.6.

![Figure 9.6: Save Constraints As dialog box](image)

- Click `OK` and your `modulator_sozius` constraint set with `modulator_sozius.xdc` file should appear in the `Sources` window under the `Constraints`, see Figure 9.7.

![Figure 9.7: Created modulator_sozius constraints set](image)

As can be seen from the Figure 9.3, in our design there is a quite a lot of unassigned pins. In this case, instead of using Vivado GUI to specify pin locations, a better approach would be to create a XDC file using Vivado text editor which is explained in the following section.
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:

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

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

Creating a XDC File using Vivado Text Editor

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

- In the **New File** dialog box, type the name of your constraints file (**sozius_xz_modulator_vio.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.

- When you click **Save**, Vivado IDE will automatically open empty **sozius_xz_modulator_vio.xdc** constraints file in Vivado text editor.

- Write the constraints into the opened **sozius_xz_modulator_vio.xdc** constraints file.

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

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

- In the **Add Sources** dialog box, select **Add or create constraints** option to add the constraints file to the project and click **Next**.

![Add Sources dialog box](image)

**Figure 9.8: Add Sources dialog box - Add or create constraints option**

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

- In the **Add Constraint Files** dialog box, browse to the project working directory and select the **sozius_xz_modulator_vio.xdc** constraints file.
- Click OK and the `sozius_xz_modulator_vio.xdc` constraints file should appear in the Add or Create Constraints dialog box.

- Click Finish and your constraints file should appear under the Constraints in the Sources view.

The complete `sozius_xz_modulator_vio.xdc` constraints file you can find in the text below.

```
sozius_xz_modulator_vio.xdc constraints file:

# set properties for bitstream generation
set_property BITSTREAM.GENERAL.COMPRESS TRUE [current_design]  
set_property BITSTREAM.GENERAL.XADCPOWERDOWN ENABLE [current_design]  
set_property BITSTREAM.GENERAL.XADCHANDLER 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
# pins must be implemented !!
set_property PACKAGE_PIN V13 [get_ports pl_phy_reset_n_o]
set_property IOSTANDARD LVCMOS33 [get_ports pl_phy_reset_n_o]
```

In the constraints file you can see assigned pin locations and I/O standards for each logical port of our design. For each logical port two physical 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.

### 9.1.1 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.

In the modulator design there is no need to specify any additional timing constraints. However, in this chapter we will present and explain the procedure for defining new timing constraints for the design to be implemented.

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:

1. In the Flow Navigator, under the Synthesis Design section, select first offered Constraints Wizard command.
2. When the No Target Constraints File dialog box appear, see Figure 10.9, just click Define Target option to associate current design with constraints file.

Figure 9.9: No Target Constraints File dialog box

3. In the Define Constraints and Target dialog box, select sozius_xz_modulator_vio.xdc file as target constraints file and click OK, see Figure 9.10. In the Define Constraints and Target dialog box, you can also create new or add existing constraints file.

Figure 9.10: Define Constraints and Target dialog box

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 Figure 9.11. After reading the page, click Next to continue.
5. In the **Primary Clocks** dialog box, Timing Constraints Wizard will display all the clock sources with a missing clock definition. 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. Click **Next** to continue.

6. 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.

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.

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.

9. Next, the wizard looks at the input delays. 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
10. Click Next to continue.

11. Next, the Output Delays page of the wizard displays all the outputs that are unconstrained in the design. The page layout is very similar to the inputs page. Click Next to continue.

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.

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.

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.

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.

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

17. The Constraints Summary page is the final page of the Timing Constraints wizard, see Figure 9.12. 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.

![Figure 9.12: Constraints Summary dialog box](image-url)
Defining timing constraints using Constraints Editor

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

1. Select Window -> Timing Constraints option from the main Vivado IDE menu to open the Timing Constraints window, see Figure 9.13, or

   ![Figure 9.13: Timing Constraints option](image)

   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 Figure 9.14.

   ![Figure 9.14: Timing Constraints window](image)
CHAPTER 9. DESIGN IMPLEMENTATION

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. You can 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 50 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.

2. In the Constraints tree view window of the Timing Constraint editor, double-click on the Create Clock (1) option under the Clocks section to create a clock constraint.

3. In the Create Clock dialog box, enter clock_name (clk_i) in the Clock name field, see Figure 9.15.

4. Click the icon next to the Source objects field and Specify Clock Source Objects dialog box will appear, see Figure 9.15.

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

5. In the Specify Clock Source Objects dialog box (see Figure 9.16), do the following:
   - Ensure that Nets is selected from the Find names of type drop-down list
   - Enter clk_i in the empty search field
   - Click Find
   - In the Find results: 10 section, select clk_i
   - Click the -> icon to select clk_i

Figure 9.15: Create Clock dialog box
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 Figure 9.17:

   - Enter **20 ns** in the **Period** field in the Waveform section, because 20 ns is the period of the 50 MHz input clock signal.

   - Ensure that the **Rise at** and **Fall at** fields are set to 0 and 10 respectively, which means that the duty cycle of the input clock signal will be 50%.

   - Click **OK** to create the clock constraint.
CHAPTER 9. DESIGN IMPLEMENTATION

Figure 9.17: Create Clock dialog box after specifying the period for the clk_i

The Timing Constraints window now displays the timing constraint applied to the design, see Figure 9.18.

Figure 9.18: 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 Figure 9.18.

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

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

- If the sozius_xx_modulator_vio.xdc file is already open, click the Reload link in the banner of the sozius_xx_modulator_vio.xdc tab to reload the constraints file from disk.
- If the sozius_xx_modulator_vio.xdc file is not open, select the Sources window, Hierarchy view
- Expand Constraints folder, see Figure 9.19
- Double-click on the sozius_xx_modulator_vio.xdc file, under the constrs_1, to open the file and you should see that your timing constraints were saved to the XDC file.

![Figure 9.19: sozius_xx_modulator_vio.xdc constraints file in the Sources window](image)

9.2 Implementation

9.2.1 About the Vivado Implementation Process

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.

Vivado Design Implementation Process

The Vivado implementation process includes logical and physical transformations of the design. The implementation process consists of the following sub-processes:
CHAPTER 9. DESIGN IMPLEMENTATION

Figure 9.20: Design Implementation Process

- **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:
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

9.2.2 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:

1. In the Flow Navigator, click Run Implementation command and wait for implementation to be completed, see Figure 9.21.

![Figure 9.21: Run Implementation command](image)

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

2. After the implementation is completed, the Implementation Completed dialog box will appear, see Figure 9.22.

![Figure 9.22: Implementation Completed dialog box](image)

3. Select Open Implementation Design option in the Implementation Completed dialog box and click OK to open the implemented design.
9.2.3 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 9.23. 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):

- **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
- **Schematic**: Opens the Schematic window

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.
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.

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

3. The Clock Interaction report will display in the main Vivado IDE window, see Figure 9.25.

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.

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

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

6. The Utilization report will display at the bottom of the Vivado IDE, see Figure 9.26.

www.so-logic.net 2018/11/29 105
9.2.4 Implementation Reports

While the Flow Navigator points to the most important reports, the Reports tab contains several other useful reports, see Figure 9.27:

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 13: "Design Implementation".
Vivado Implementation Log - describes the implementation process and any issues it encountered.

IO Report - Lists every signal, its attributes and its final location, see Figure 9.28. It is always important to double-click pin assignments before implementing, because the tools can move any pin that is unassigned.

![Figure 9.28: IO Report](image)

Utilization Report - describes the amount of FPGA resources used in a text format, see Figure 9.29.

![Figure 9.29: Utilization Report](image)
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.

![Figure 9.30: Control Sets Report](image)

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 Figure 9.31.
Figure 9.31: Power Report

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

Figure 9.32: 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.

### 9.3 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:
1. In the Flow Navigator, under Program and Debug, click on the Generate Bitstream command, see Figure 9.33.

![Figure 9.33: Generate Bitstream command](image)

Note that the Generate Bitstream process will try to resynthesize and implement the design if any of process is out of date.

2. Click Yes to acknowledge running of the processes that are needed for bitstream generation.

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 13: "Design Implementation".

9.4 Program Device

After you have generated the bitstream file, the next step will be to download it into the target FPGA device. In our case it will be socius development 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:

1. For the socius development board, connect the Digilent USB JTAG cable of your socius board to the Windows machine’s USB port.

2. Ensure that the board is plugged in and powered on.

3. Make sure that the board settings are proper.

4. In the Flow Navigator, under the Program and Debug, click Open Hardware Manager command, see Figure 9.34.

![Figure 9.34: Open Hardware Manager command](image)

The another way to open the hardware manager is to select Flow -> Open Hardware Manager option from the main Vivado menu.

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 Figure 9.35.
Use the Open Target command, under the Open Hardware Manager in the Program and Device section, to open new or recent hardware targets, see Figure 9.36.

Use Tcl commands to open a connection to a hardware target.

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 Figure 9.37.

7. In the Open Hardware Target dialog box, click Next.

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 Figure 9.38.

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.
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 Figure 9.39.

![Select Hardware Target dialog box](image)

**Figure 9.39:** 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.

10. Click Next.

11. In the Open Hardware Target Summary dialog box, click Finish to connect to the hardware described in the summary window, see Figure 9.40.
Once 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 Figure 9.41.

12. You can program the hardware device right-clicking on the device in the Hardware window and selecting the Program Device... option, see Figure 9.42.

The another way to program your device is to select Program device option from the Hardware Manager view, as it is shown on the Figure 9.43.
In the **Program Device** window, click **Program** to program your device, see Figure 9.44.

![Figure 9.44: Program Device window](image)

**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.

Once 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 Figure 9.45.

![Figure 9.45: Hardware Device Properties window](image)

In case of using development boards without involving processor usage this will be the last step in programming process. In this case, after downloading your design into the targeting device, led diode on the board will start blinking. The speed of blinking will be chosen depending on the position of the two-state on-board switch sw0.

In case of using **Sozius** development board, which involves the processor usage because of the necessary clock generation, some additional steps must be done. After programming Sozius device, we have to create application project using Vivado SDK tool to initialize the processing system which will then start generating internal clock signal used by the Modulator design (connected to the clk_i port). Systems that involves processor usage are explained in detail in the "**Embedded System Design**" tutorial.
Export Hardware

- When the Sozius board is programmed, select **File -> Export -> Export Hardware...** option from the main Vivado IDE menu.

- In the **Export Hardware** dialog box, you don’t have to include bistream file, so just click **OK**.

![Export Hardware dialog box]

Figure 9.46: Export Hardware dialog box

Launch SDK

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:

- Select **File -> Launch SDK** from the main Vivado IDE menu.

- In the **Launch SDK** dialog box, make sure that both **Exported location** and **Workspace** are set to **Local to Project** and click **OK**.

![Launch SDK dialog box]

Figure 9.47: Launch SDK dialog box

SDK will be launched in a separate window, see Figure 9.48.
CHAPTER 9. DESIGN IMPLEMENTATION

To create an application project, do the following:

**Create Application Project and Run Application**

- Select **File -> New -> Application Project** and the **Application Project** dialog box will appear.

- In the **Project name** field, type a name of the new project, in our case it will be **modulator_sozius**, leave all other parameters unchanged and click **Next**.
- 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**.

![Templates dialog box](image)

**Figure 9.50: Templates dialog box**

- In the SDK **Project Explorer**, select your application project (*modulator_sozius*), right-click on it and select **Run As -> Launch on Hardware (System Debugger)** option.

- Turn back to the Vivado IDE and in the **Hardware** window of the **Hardware Manager** right-click on the FPGA device (*xc7z020_1*) and select **Refresh Device** option.

![Hardware Manager](image)

**Figure 9.51: ILA Dashboard**

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.
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 menu, see Figure 9.53.

![Figure 9.52: Close Target option](image)

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 Figure 9.54.

![Figure 9.53: Close Server option](image)

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 13: "Design Implementation"**.

### 9.5 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 **ZedBoard** development board.

Both, ZedBoard and Sozius development boards, has single-ended reference clock. ZedBoard has 100 MHz reference clock, while Sozius has 50 MHz reference clock. To accommodate your design to the new development board, do the following steps:

- Change the type of the target FPGA device,
- Change the xdc constraints file, and
- Change the necessary source codes.

1. **Change the type of the target FPGA device.**
CHAPTER 9. DESIGN IMPLEMENTATION

- In the **Project Summary** window (Settings) click on the `xc7z020clg400-1`, see Figure 9.55.

![Figure 9.54: Project Settings window](image)

- In the **Settings** dialog box, click on the icon beside **Project device** field to browse the another development board, see Figure 9.56.

![Figure 9.55: Project Settings dialog box](image)

- In the **Select Device** dialog box, choose **Boards** option, then select **ZedBoard Zynq Evaluation and Development Kit** and click **OK**, see Figure 9.57.
2. **Change the xdc constraints file.**

Create new `modulator_zedboard.xdc` constraints file, save it in your working directory and include it in the design. The content of the `modulator_zedboard.xdc` constraints file for the ZedBoard is presented in the text below:

```vhdl
set_property PACKAGE_PIN Y9 [get_ports clk_p]
set_property PACKAGE_PIN F22 [get_ports sel_i]
set_property PACKAGE_PIN T22 [get_ports pwm_o]

set_property IOSTANDARD LVCMOS33 [get_ports clk_p]
set_property IOSTANDARD LVCMOS25 [get_ports sel_i]
set_property IOSTANDARD LVCMOS33 [get_ports pwm_o]

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

The things that we changed in the xdc file:

- **Placement Constraints** - find in the User Guide for the ZedBoard development board pin locations where you would like to connect the input differential clock (clk_p, clk_n) and sel_i and pwm_o ports.
- **Timing Constraints** - change the period of the input clock signal. For ZedBoard development board, you have to change input clock period from 20 ns to 10 ns, because ZedBoard development board has 100 MHz input clock frequency.

3. **Change the source codes.**

Because we changed the target development board, from Sozius to ZedBoard, 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:

  ```vhd
  type board_type_t is (lx9, zedboard, ml605, kc705, vc707, microzed, sozius);
  ```

- Create a new constant for the new development board. Constant must be a structure of type `board_setting_t`. 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

  ```vhd
  -- place the information about the new boards here:
  constant lx9_c : board_setting_t := (lx9, 100000000.0, no);  Spartan-6
  constant zedboard_c : board_setting_t := (zedboard, 100000000.0, no);  Zynq-7000
  constant ml605_c : board_setting_t := (ml605, 200000000.0, yes);  Virtex-6
  constant kc705_c : board_setting_t := (kc705, 200000000.0, yes);  Kintex-7
  constant vc707_c : board_setting_t := (vc707, 200000000.0, yes);  Virtex-7v
  constant microzed_c : board_setting_t := (microzed, 333333333.3, no);  MicroZed
  constant sozius_c : board_setting_t := (sozius, 500000000.0, no);  Sozius
  ```

**modulator_tb.vhd:**

- Change the type of your development board. In our case it will be from `sozius_c` to `zedboard`.

  ```vhd
  -- defines board specific settings
  board_setting_g : board_setting_t := zedboard_c;
  ```

**sozius_xz2_modulator_vio_rtl.vhd:**

- Remove `sozius_xz2_modulator_vio_rtl.vhd` file from the design.

**modulator_wrapper_rtl.vhd:**

- Create new `modulator_wrapper_rtl.vhd` source file and include it into the Modulator design.

  - **Usage:** This module is necessary to support different development boards with different reference 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:**

  ```plaintext
  Figure 9.57: Modulator wrapper block diagram
  ```

  - **Input ports:**
    * `clk_p`: differential input clock signal
    * `clk_n`: differential input clock signal
* sel_i : input signal from the on-board switch, used for changing output signal frequency

- **Output ports:**
  * pwm_o : 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_setting_g : parameter that specifies major characteristics of the board that will be used to implement the modulator design
  * design_setting_g : user defined settings for the pwm design

- **File name:** modulator_wrapper_rtl.vhd

**Modulator wrapper VHDL model:**

```vhdl
-- 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.numeric_std.all;
library unisim;
use unisim.vcomponents.all;
library work;
use work.modulator_pkg.all;

-- Entity defines the interface of a module
-- Generics are static, they are used at compile time
-- Ports are updated during operation and behave like signals on a schematic or
-- traces on a PCB
-- Entity is a primary design unit
entity modulator_wrapper is
  generic(
    this_module_is_top_g : module_is_top_t := yes;
    board_setting_g : board_setting_t := zedboard_c;
    design_setting_g : design_setting_t := led_setting_c);
  port(
    clk_p : in std_logic; -- differential input clock signal
    clk_n : in std_logic; -- differential input clock signal
    sel_i : in std_logic_vector(0 downto 0); -- signal made for selecting frequency
    pwm_o : 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
  constant inc_c : a1integer_t := calc_inc_f(board_setting_g, design_setting_g);
  signal inc_s : std_logic_vector(design_setting_g.freq_hz'length*design_setting_g.nco_width - 1 downto 0) := (others => '0');
  signal clk_i_s : std_logic;
  signal sel_s : std_logic_vector(0 downto 0);
  signal pwm_s : std_logic;
begin
  -- converts from array of ints to std_logic_vector
end architecture;
```

-- 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
```vhdl
architecture rtl of modulator_wrapper is
begin
  -- Converts from array of ints to std_logic_vector
end architecture;
```
inc_s <= conv_int_array_to_slv_f(inc_c, design_setting_g.nco_width);
-- in case of MicroZed board we must enable on-board clock generator

-- 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_setting_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_i_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_setting_g.has_diff_clk = no) generate
            clk_i_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_i_s <= clk_p;
end generate not_top;

-- modulator module instance
modulator_i: entity work.modulator(rtl)
generic map(
    board_setting_g => board_setting_g,
    design_setting_g => design_setting_g
)

port map (  
    clk_i => clk_i_s,
    inc_i => inc_s,
    sel_i => sel_s,
    pwm_o => pwm_s
);
end;
Chapter 10

DEBUGGING DESIGN

In this chapter we will show how user can debug a design. We will use Vivado Logic Analyzer as an integrated Vivado analyzer.

10.1 Inserting ILA and VIO Cores into Design

Vivado Logic Analyzer

Vivado Logic Analyzer is an 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.

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 11.1 "IP Integrator".

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.
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.
**CHAPTER 10. DEBUGGING DESIGN**

Figure 10.2: Symbol of the VIO v3.0 core

*Note:* 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".

**Using the Netlist Insertion Debug Probing Flow**

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 design using the HDL instantiation debug probing flow.

**VIO Core Generation**

- In the Vivado **Flow Navigator**, under the **Project Manager**, click the **IP Catalog** command.

- 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.
- 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.

  - 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_o` signal and one output probe for `sel_i` signal.

  - In the **PROBE_IN Ports(0..0)** tab leave Probe Width of the **PROBE_IN0** Probe Port to be 1, because our `pwm_o` 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 sel_i signal is also 1 bit signal.

- Click **OK**.

- In the **Generate Output Products** window click **Generate**.
Adding Debug Nets

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 cc_count_s*]
  set_property mark_debug true [get_nets pwm_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 cc_count_s : signal is "true";
  attribute mark_debug of pwm_s : signal is "true";
  ```

  *Note:*
Verilog:

(* mark_debug *) wire cc_count_s;
(* mark_debug *) wire pwm_s;

- Right-click and select **Mark Debug** or **Unmark Debug** on Synthesis netlist.

- Use **Tcl prompt** to set the **mark_debug** attribute. For example:

  ```tcl
  set mark_debug true [get_nets cc_count_s]
  set mark_debug true [get_nets pwm_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.

We will use **mark_debug** attribute to add debug nets (**pwm_s** and **cc_count_s**) to our HDL file (**sozius_xz2_modulator_vio_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 **cc_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 that will measure the duration of the PWM pulse, see Figure 10.9. Both of these instances, plus ILA and VIO core instances will be included within **sozius_xz2_modulator_vio_rtl.vhd** VHDL model.

### Connection Between ILA, VIO Core and Modulator Module

![Figure 10.9: Modulator Sozius block diagram](image)

www.so-logic.net 2018/11/29 131
Create and Add `sozius_xz2_modulator_vio_rtl.vhd` Source File

To create and add `sozius_xz2_modulator_vio_rtl.vhd` source file 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 file you can find in the Sub-chapter 8.2 Creating Module of this tutorial.

Note: Don’t forget to set `sozius_xz2_modulator_vio_rtl.vhd` source file to be the top file!

Configure the Zynq PS Part to work on Sozius Development Board

Now, we must configure the Zynq PS part to work on Sozius development board.

This includes a number of configuration steps.

All the PS configuration steps will be done using `sozius_xz_lab_ps_bd.tcl` Tcl script file.

This Tcl script file is too long to be shown in the tutorial, so ask your instructor for details.

Execute Tcl File in the Vivado IDE

Next step is to execute the `sozius_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>/sozius_xz_lab_ps_bd.tcl
```

Where `<path>` stands for the full path to the folder where the `sozius_xz_lab_ps_bd.tcl` Tcl file is stored.

![Figure 10.10: Tcl Console window](image)

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 Figure 10.11.

Block Diagram after Vivado Tcl Script Execution
Creating XDC File

- Now is the time to create and add constraints file for the Sozius board, sozius_xz_modulator_vio.xdc.

To create and add constraints file, please use steps from the Sub-chapter 9.1 "Creating XDC File", where it is in detail explained in paragraph "Creating a XDC File using Vivado Text Editor".

The complete sozius_xz_modulator_vio.xdc constraints file you can find in the Sub-chapter 9.1 "Creating XDC File".

- In the Project Manager, click the Settings command, see Figure 10.12.

![](image)

Figure 10.12: Project Settings command

- In the Settings dialog box, select Synthesis option from the left pane.

- In the Synthesis window, change the flatten_hierarcy option from rebuilt to none as it is shown on the Figure 10.13 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.
Design Synthesis

- In the Vivado Flow Navigator, click Run Synthesis command (Synthesis option) and wait for task to be completed.

- After the synthesis is completed, choose Open Synthesized Design option in the Synthesis Completed dialog box.

- Open Debug Layout, if it is not already opened.

Figure 10.13: Project Settings dialog box

Figure 10.14: Run Synthesis command

Figure 10.15: Synthesis Completed dialog box

Figure 10.16 and 10.17 shows assigned debug nets to the VIO core and debug nets that were marked in the
sozius_xz2_modulator_vio_rtl.vhd source file with mark_debug attributes and that we will assign to the ILA core.

- Select the Netlist tab, beside Sources tab and expand Nets folder of the sozius_xz2_modulator_vio module. 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 Figure 10.18.

- In the Confirm Debug Net(s) dialog box (in case of marking new debug net), click OK, see Figure 10.19.
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:

**Set up Debug Wizard**

- In the Debug window, select Set Up Debug button to launch the wizard.

![Set Up Debug button](image)

- In the Set Up Debug dialog box, click Next to open Nets to Debug dialog box.

![Nets to Debug dialog box](image)
- In the **Nets to Debug** dialog box you will find nets that you have marked for debugging.

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 Figure 10.24.
- If you are satisfied with the debug net selection, click **OK**.

- 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.

  ![Find Nets dialog box](image)

  **Figure 10.24: Find Nets dialog box**

  ![Select Clock Domain option](image)

  **Figure 10.25: 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.

- In the **Select Clock Domain** dialog box modify clock domain as needed.

Be aware that each clock domain present in the table results in a separate ILA v6.2 core instance.
- Select the same clock domain for `pwm_s` net, because signals captured by the same ILA core must have the same clock domain.

- Ones you are satisfied with the debug net selection, click Next.

- In the **ILA Core Options** dialog box, enable **Capture control** option, leave all parameters unchanged and click Next.
CHAPTER 10. DEBUGGING DESIGN

Figure 10.28: 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 Figure 10.29.

- If you are satisfied with the results, click **Finish** to insert and connect the ILA v6.2 cores in your synthesized design netlist.

Figure 10.29: Set up Debug Summary dialog box

- The debug nets are now assigned to the ILA v6.2 debug core, what you can see in the **Debug** window, see Figure 10.30.
CHAPTER 10. DEBUGGING DESIGN

Figure 10.30: Debug window with assigned debug nets

The generated ILA core you can also find in the Netlist window, see Figure 10.31.

Figure 10.31: Netlist window with generated ILA core
- Implement your design with Run Implementation option from the Flow Navigator / Implementation (see Sub-Chapter 10.2.2 Run Implementation).

- Generate bitstream file with Generate Bitstream option from the Flow Navigator / Program and Debug (see Sub-Chapter 10.3 Generate Bitstream File).

- Program your Sozius device (see Sub-Chapter 10.4 Program Device).

In case of using Sozius development board, which involves the processor usage because of the necessary clock generation, some additional steps must be done. After programming Sozius device, we have to create application project using Vivado SDK tool to initialize the processing system which will then start generating internal clock signal used by the Modulator design (connected to the clk_i port). Systems that involves processor usage are explained in detail in the "Embedded System Design" tutorial.

- When the sozius board is programmed, select File -> Export -> Export Hardware... option from the main Vivado IDE menu.

- 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:

- Select File -> Launch SDK from the main Vivado IDE menu.

- 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:

- In the SDK, select File -> New -> Application Project and the Application Project dialog box will appear.

- In the Project name field, type a name of the new project, in our case it will be modulator_sozius, leave all other parameters unchanged and click Next.

- 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.

- In the SDK Project Explorer select your application project (modulator_sozius), right-click on it and select Run As -> Launch on Hardware (System Debugger) option.

- Turn back to the Vivado IDE and in the Hardware window of the Hardware Manager right-click on the FPGA device (xc7z020_1) 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.
10.2 Debug a Design using Integrated Vivado Logic Analyzer

Vivado Logic Analyzer

Vivado logic analyzer is an integrated logic analyzer. Once 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.

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.

Figure 10.33: Hardware window with ILA and VIO cores

The next step in design debugging process is to set up the ILA core.
ILA Dashboard

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 figure below.

![ILA Dashboard](image)

**Figure 10.34: ILA Dashboard**

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

**Add Probes to the VIO Window**

- Open the VIO dashboard by clicking the `hw_vios` tab and press blue `+` button in the middle of the VIO dashboard to add the probes.

- In the **Add Probes** window select both `pwm_s` and `sel_s` probes and click **OK**.
- In the VIO Probes window you will see two 1-bit probes, pwm_s and sel_s, see Figure 10.36. pwm_s probe is actually connected to the pwm_o output port of the Modulator module, as can be seen on the Figure 10.9. Similarly, sel_s probe is connected to the sel_i input port of the Modulator module.

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 sel_s probe from 0 to 1 and from 1 to 0, see Figure 10.37. The default sel_s value is 0.
- Turn back to the ILA dashboard by clicking the `h_ila_1` tab and in the Trigger Setup window press blue + button in the middle to add the probes.

- In the Add Probes window select only `pwm_s_1` probe and click OK, see Figure 10.38.

**Add Probes to the Trigger Setup Window**

The another way to add debug probes to the Basic Trigger Setup window is to drag and drop the probes from the Debug Probes window to the Basic Trigger 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 Trigger 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.

Set the Compare Values in the Trigger Setup Window

- 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).

![Figure 10.39: Changing the Compare Values in the Trigger Setup window](image)

As you can see from the illustration above, the Trigger 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 than)
  - \(<=\) (less than 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 previous value)
  - **Hexadecimal**
    - \(X\) : All bits corresponding to the value string character are "don’t care" values
- In the **ILA Settings** window, change the **Capture mode** to be **BASIC** in the **Capture Mode Settings** section, see Figure 10.40.

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 **ALWAYS** 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

**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

- In the **Capture Setup** window press blue + button in the middle to add the probes.

- In the **Add Probes** window select only **pwm_s_1** probe and click OK, see Figure 10.40.
CHAPTER 10. DEBUGGING DESIGN

Set the ILA Settings

![Image of Capture Setup window](image)

**Figure 10.40: Add Probes to the Capture Setup window**

Add Probes to the Capture Setup Window and Set the Compare Values

- In the Capture Setup window, leave $\text{=} \text{(equal)}$ value in the Operator cell, $[B]$ (Binary) value in the Radix cell and set the Value parameter to be $F$ (1-to-0 transition).

![Image of Compare Values](image)

**Figure 10.41: Changing the Compare Values in the Capture Setup window**

Run the ILA Core Trigger

- 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_il_1) in the Hardware window and selecting Run Trigger or Run Trigger Immediate option from the popup menu, see Figure 10.42.

![Run Trigger option](image)

**Figure 10.42: Run Trigger option**

Once the ILA core captured data has been uploaded to the Vivado IDE, it is displayed in the Waveform Viewer, see Figure 10.43.

*Note:* After triggering the ILA core, in the waveform viewer change the *cc_count_s* Waveform Style from Digital to Analog, and your captured waveform should look like as the waveform on the Figure 10.43.

**Design Analysis - Part 1**

![Captured waveform of the sine signal, when sel_s=0](image)

**Figure 10.43: Captured waveform of the sine signal, when sel_s=0**

**Change the sel_s Value**

- Turn back to the VIO Probes window and change the Value of the *sel_s* signal from 0 to 1.

- Arm the trigger ones more and after triggering the ILA core your captured waveform should look like as the waveform on the following figure.

**Design Analysis - Part 2**
Figure 10.44: Captured waveform of the sine signal, when sel_s=1

Note: By comparing the waveforms shown on Figures 10.43 and 10.44 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.

The ILA core can capture data samples when the core status is Pre-Trigger, Waiting for Trigger or Port-Trigger, see Figure 10.38. 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 \texttt{pwm\_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 Figure 10.43 or Figure 10.44.

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 15: "Debug a Design using Integrated Vivado Logic Analyzer".
Chapter 11

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.

**Vivado IP-Centric Design Flow**

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. Figure below illustrates the IP-Centric design flow.

![Figure 11.1: Vivado IP-Centric Design Flow](image)

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.

11.1 IP Integrator

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.

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.

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:

IP Integrator features

- 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

IP Integrator within your Project

In this tutorial you will instantiate a few modules and 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 Sozius development board.

The following steps describe how to use the IP Integrator within your project:
Create IP Integrator Project

- 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 Project** option.

- In the **Create a New Vivado Project** dialog box, click **Next** to confirm the new project creation.

![Create a New Vivado Project dialog box](image1)

**Figure 11.2: Create a New Vivado Project dialog box**

- 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**.

![Project Name dialog box](image2)

**Figure 11.3: Project Name dialog box**
- 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**.

![Project Type dialog box](image1)

**Figure 11.4: Project Type dialog box**

- 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.

![Add Sources dialog box](image2)

**Figure 11.5: Add Sources dialog box**

- In the **Add Constraints (optional)** dialog box, remove if there are some constraints files, and click **Next**.
- In the **Default Part** dialog box, click **Parts** option and set the following parameters as it is shown on figure below.

![Default Part dialog box](image)

**Figure 11.6: Add Constraints (optional) dialog box**

- 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.

![New Project Summary dialog box](image)

**Figure 11.7: Default Part dialog box**
After we finished with the new project creation, in a few seconds Vivado IDE will appear with the created modulator_ipi project, see Figure 11.9.

Create Block Design
- In the Flow Navigator, expand **IP Integrator** and select **Create Block Design** command.

![Create Block Design option](image)

**Figure 11.10: Create Block Design option**

- In the **Create Block Design** dialog box, specify **modulator_ipi** name of the block design in the **Design name** field and click **OK**.

![Create Block Design dialog box](image)

**Figure 11.11: Create Block Design dialog box**

The Vivado IDE will display a blank design canvas. You can quickly create complex subsystem by integrating modules and IP cores in it, see Figure 11.12.

![Vivado IDE with a blank design canvas](image)

**Figure 11.12: Vivado IDE with a blank design canvas**
The **Module Reference** feature of the Vivado IP integrator lets you quickly add a module or entity definition from a Verilog or VHDL source file directly into your block design.

While this feature does have limitations, it provides a means of quickly adding RTL modules without having to go through the process of packaging the RTL as an IP to be added through the Vivado IP catalog.

Both flows have their benefits and costs:

- The **Package IP** flow is rigorous and time consuming, but it offers a well-defined IP that can be managed through the IP catalog, used in multiple designs, and upgraded as new revisions become available.

- The **Module Reference** flow is quick, but does not offer the benefits of working through the IP catalog.

In this tutorial both flows will be used. The following section explains the usage of the Module Reference technology. The Package IP flow will be explained later, in the last sub-chapter 11.2 "Creating Modulator IP Core with AXI4 Interface".

### Add RTL Source Files to the Vivado Project

- To add HDL to the block design, first you must add the RTL source file to the Vivado project.

- Add the following source files into the `modulator_ipi` project using **Add Sources** command from the Project Manager:

  - `selector_rtl.vhd`
  - `counter_rtl.vhd`
  - `sine_rtl.vhd`
  - `pwm_rtl.vhd`
  - `modulator_rtl.vhd`
  - `modulator_pkg.vhd`

An RTL source file can define one or more modules or entities within the file. The Vivado IP Integrator can access any of the modules defined within an added source file, see Figure 11.13.

#### Figure 11.13: RTL Sources in the Sources window

### Add Modules to the Block Design

- In the block design, you can add a reference to an RTL module using the **Add Module** command from the right-click menu of the design canvas, as shown in the following figure.
- The **Add Module** dialog box displays a list of all valid modules defined in the RTL source files that you have added to the project. Select a selector module to add from the list, and click **OK** to add it to the block design, see Figure 11.15.

The **Add Module** dialog box also provides a **Hide incompatible modules** check box that is enabled by default. This hides module definitions in the loaded source files that do not meet the requirements of the Module Reference feature and, consequently, cannot be added to the block design.

You can uncheck this check box to display all RTL modules defined in the loaded source files, but you will not be able to add all modules to the block design. Examples of modules that you might see when deselecting this option include:

- Files that have syntactical errors
- Modules with missing sources
- Module definitions that contain or refer to an EDIF netlist, a DCP file, another block design, or unsupported IP

You can also add modules to an open block design by selecting the module in the Sources window and using the **Add Module to Block Design** command from the context menu, see figure 11.16.
Finally, RTL can also be dragged and dropped from the Sources view onto the block design canvas.

The IP integrator adds the selected module to the block design, and you can make connections to it just as you would with any other IP in the design. The IP displays in the block design with special markings that identify it as an RTL referenced module, see Figure 11.17.

- Right-click in the IP integrator canvas and select the **Add Module...** option to add the rest of the necessary modules (counter(counter_rt.vhd), sine(sine_rtl.vhd) and pwm(pwm_rt.vhd)). At this point, the IP Integrator canvas should look like as it is shown on the following figure.
- Double-click on the each of the modules to re-customize them.

First, double-click on the `selector_v1_0` module and the Vivado IDE will automatically open the re-customization window for the `selector_v1_0` module.

The window that will be opened is used to set up the general `selector` module parameters, see Figure 11.19.

### `selector_v1_0` Module Re-customization

- In the `selector_v1_0(1.0)` dialog box, configure the parameters on the same way as it is shown on the Figure 11.19 and click OK.

*Note:* To know how to configure the right values, open `modulator rtl.vhd` and `modulator pkg.vhd` source
files and find out how the generics are setted.

- Double-click on the rest three modules and re-customize them on the same way as it is shown on the following Figures 11.20, 11.21 and 11.22.

**counter_v1_0 Module Re-customization**

![](image1)

Figure 11.20: Counter re-customize module reference window

**sine_v1_0 Module Re-customization**

![](image2)

Figure 11.21: Sine re-customize module reference window

**pwm_v1_0 Module Re-customization**
Add and Re-Customize Slice IP

- To continue working on the "modulator_ipi" block design, you have to add \texttt{Slice(xlslice_0)} IPs from the IP Catalog.

  - In the design canvas, right-click and choose \textbf{Add IP...} option.

  - Use the \textbf{Add IP} link in the IP Integrator canvas, see Figure 11.24, or
- Click on the Add IP button in the IP Integrator sidebar menu, see Figure 11.25.

- In the IP Catalog, search for the Slice core:

- When you find it, press enter on the keyboard or simply double-click on the Slice core in the IP Catalog and the selected core will be automatically instantiated into the IP Integrator design canvas.

- Double-click on it to re-customize the core. Re-customize the Slice IP core on the same way as it is shown
CHAPTER 11. DESIGNING WITH IPS

on the following figure.

![Slice re-customization IP window](https://www.so-logic.net/2018/11/29)

Figure 11.27: Slice re-customization IP window

The **Slice** IP core is necessary to rip bits off a bus net. Often there is a need to rip some bits off a wide bus net. This IP core can be instantiated to accomplish this purpose.

**Create Necessary Ports**

After we added all the necessary modules and IPs into our design, the next step will be to connect modules and IPs between themselves. Make connections on the same way as it is shown on the following IP Integrator block diagram.

- First step will be to create new ports:

  - Select `clk_i` pin, right-click on it and select **Create port...** option.
In the Create Port dialog box, check the port name clk_i in the Port name field, leave all other parameters unchanged and click OK.

Repeat the same procedure with pwm_o, inc_i and sel_i ports, where

- pwm_o port will also be one bit wide port
- inc_i port will be 62 bits wide vector, and
- sel_i port will be 1 bit wide vector.

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 Figure 11.30.

Connect Modules and IPs

![Figure 11.30: IP Integrator design canvas with connected modules and IPs](image)

- 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 Figure 11.31, or

![Figure 11.31: Validate Design option from the main menu](image)

by clicking the design canvas and selecting **Validate Design** button from the main toolbar menu, see Figure 11.32.
- In the **Validate Design** dialog box, click **OK**, see Figure 11.33.

**Figure 11.32: Validate Design button from the main toolbar menu**

**Figure 11.33: Validate Design dialog box**

**Save Block Design**

- 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.

**Execute Tcl File**

- Execute the `sozius_xz_lab_ps_bd.tcl` Tcl file in the Vivado IDE to properly configure the PS part of the Zynq7 processing system.

Go to the Tcl console window and type the following and press enter:

```
source <path>/sozius_xz_lab_ps_bd.tcl.
```

After `sozius_xz_lab_ps_bd.tcl` Tcl file execution you can notice that Vivado IDE has created the second block designs, `sozius_xz_lab_ps_bd`, beside `modulator_ipi` block design.

Now, we should create `sozius_xz2_modulator_vio_ipi_rtl.vhd` source file where we will instantiate `sozius_xz_lab_ps_bd`, `modulator_ipi` and `vio_core` components, on the same way as it was done before in this tutorial, see Figure 8.1. The complete `sozius_xz2_modulator_vio_ipi_rtl.vhd` file you can find in the text below.

```
sozius_xz2_modulator_vio_ipi_rtl.vhd:

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

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library work;
use work.modulator_pkg.all;
use work.sozius_components_package.all;

entity sozius_xz2_modulator_vio_ipi is
  generic(
    -- version number of PL for read back from PS
    hw_version_g : std_logic_vector(31 downto 0) := x"19980418";
    -- User defined settings for the pwm design
    board_setting_g : board_setting_t := sozius_c;
    design_setting_g : design_setting_t := led_setting_c
  );
end sozius_xz2_modulator_vio_ipi;
```

170 2018/11/29 www.so-logic.net
-- ethernet phy reset, must be high necessary for operation !!!
pl_phy_reset_s_o : out std_logic := '1'; -- reset of ethernet
-- io
ps_ddr3_addr : inout std_logic_vector(14 downto 0);
ps_ddr3_bm : inout std_logic_vector(3 downto 0);
ps_ddr3_cs_n : inout std_logic;
ps_ddr3_cs_p : inout std_logic;
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_dq : inout std_logic_vector(31 downto 0);
ps_ddr3_dm : inout std_logic_vector(3 downto 0);
ps_ddr3_oe : 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 sozius_xz2_modulator_vio_ipi is
-- Between architecture and begin is declaration area for types, signals and constants
-- Everything declared here will be visible in the whole architecture
attribute mark_debug : string;
signal inc_c : a1integer_t := calc_inc_f(board_setting_g, design_setting_g);
signal inc_s : std_logic_vector(design_setting_g.freq_hz'length*design_setting_g.nco_width - 1 downto 0)
:= (others => '0');
signal cc_count_s : std_logic_vector(31 downto 0) := (others => '0');
signal pwm_s : std_logic;
signal sel_s : std_logic_vector(0 downto 0);
attribute mark_debug of cc_count_s : signal is "true";
attribute mark_debug of pwm_s : signal is "true";
attribute mark_debug of sel_s : signal is "true";
-- declaration for fixed signal PL to PS
signal pl_clk0_s : std_logic;
signal pl_reset_n_s : std_logic;
-- ps signals
signal ps_mio_s : std_logic_vector(53 downto 0);
-- vio_core component definition
component vio_core

www.so-logic.net 2018/11/29 171
CHAPTER 11. DESIGNING WITH IPS

```vhdl
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;

component modulator_ipi
port(
    clk_i : in std_logic; -- input clock signal
    inc_i : in std_logic_vector(61 downto 0);
    -- inc_i : in a1integer_t := calc_inc_f(board_setting_g, design_setting_g);
    sel_i : in std_logic_vector(0 downto 0); -- signal made for selecting frequency
    pwm_o : out std_logic -- pulse width modulated signal
);
end component;

begin
  inc_s <= conv_int_array_to_slv_f(inc_c, design_setting_g.nco_width);

  -- modulator module instance
  modulator_i: modulator_ipi
    port map(
      clk_i => pl_clk0_s,
      inc_i => inc_s, -- inc_c,
      sel_i => sel_s,
      pwm_o => pwm_s
    );

  -- counter for measuring the duration of the high pulse of the PWM signal
  measurement_counter_p: process
  begin
    wait until rising_edge(pl_clk0_s);
    if (pwm_s = '0') then
      cc_count_s <= (others => '0');
    else
      cc_count_s <= std_logic_vector(unsigned(cc_count_s) + 1);
    end if;
  end process;

  -- vio.core component instance
  vio_i: vio_core
    port map (clk => pl_clk0_s,
              probe_in0(0) => pwm_s,
              probe_out0 => sel_s);

  -- instance of processor system PS
  sozius_xz_lab_ps_bd_i: component sozius_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 => '0',
      pl_uart_1_txd => open,
      pl_spi_0_io0_i => '0',
      pl_spi_0_io0_o => open,
      pl_spi_0_io0_t => open,
      pl_spi_0_io1_i => '0',
      pl_spi_0_io1_o => open,
      pl_spi_0_io1_t => open,
      pl_spi_0_sck_i => '0',
      pl_spi_0_sck_o => open,
      pl_spi_0_sck_t => open,
      pl_spi_0_ss1_o => open,
      pl_spi_0_ss2_o => open,
      pl_spi_0_ss_i => '0',
      pl_spi_0_ss_o => open,
      pl_spi_0_ss_t => open,
      pl_iic_1_scl_i => '0',
      pl_iic_1_scl_o => open,
    );
end;
```
CHAPTER 11. DESIGNING WITH IPS

pl_iic_1_scl_t := open;
pl_iic_1_sda_i := '0';
pl_iic_1_sda_o := open;
pl_iic_1_sda_t := open;
mi_o_cdm := '1', -- pl_m_i_cdm_i;
usbind_0_port_indctl := open;
usbind_0_resr_perselect := open;
pl_clk0 := pl_clk0_s;
pl_reset_n := pl_reset_n_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_phy_rx_ctrl_io;
ps_mio_s (27) <= ps_phy_rxd_io;
ps_mio_s (26 downto 23) <= ps_phy_txd_io;
ps_mio_s (22) <= ps_phy_rts_cts_io;
ps_mio_s (21) <= ps_phy_rts_csi_io;
ps_mio_s (20 downto 17) <= ps_phy_txdio;
ps_mio_s (16) <= ps_phy_c si 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;

- Add newly created sozius_xz2_modulator_vio_ipi_rtl.vhd file with sozius_components_package.vhd file into our design and the Sources window should look the same as on the Figure 11.34.

- Set sozius_xz2_modulator_vio_ipi_rtl.vhd file to be the top module.

Instantiate VIO Core into the Design

- Instantiate VIO core into the design using IP Catalog, on the same way as it is done previously in this tutorial, see Sub-chapter 11.1 "Inserting ILA and VIO Cores into Design".

Figure 11.34: Sources window with sozius_xz2_modulator_vio_ipi design
Add Constraints File and Instantiate ILA Core into the Design

- The next step will be to add `sozius_xz_modulator_vio.xdc` constraints file.

- Synthesize your design with Run Synthesis option from the Flow Navigator / Synthesis (see Sub-chapter 7.5.2 Run Synthesis).

- After the synthesis is completed, choose Open Synthesized Design option in the Synthesis Completed dialog box.

- 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 `cc_count_s` nets to ILA core, as it is explained 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!

Design Implementation, Bitstream File Generation, Device Programming and Debugging Design

- Implement your design with Run Implementation option from the Flow Navigator / Implementation (see Sub-Chapter 9.2.2 Run Implementation).

- Generate bitstream file with Generate Bitstream option from the Flow Navigator / Program and Debug (see Sub-Chapter 9.3 Generate Bitstream File).

- Program your Sozius device (see Sub-Chapter 9.4 Program Device).

- After programming Sozius device, you should get the same results as it is explained in the Sub-chapter 10.2 "Debug a Design using Integrated Vivado Logic Analyzer".

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".

11.2 Creating Modulator IP Core with AXI4 Interface

AXI 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.

AXI4 Interface Types

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: "sel_i" "inc_i_freqhigh" and "inc_i_freqlow". The first register, "sel_i" register, will be connected to the sel_i port of the Modulator module. The next two registers, "inc_i_freqhigh" and "inc_i_freqlow" will be connected to the same "inc_i" port of the Modulator module and will be used for storing increment values for different frequencies. 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 Figure 11.35.

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.

"Create and Package IP" Wizard

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 previous figure) and allows user to add user specific logic inside this AXI enabled IP (Modulator module on the previous figure).

Wizard Configuration

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 (sel_i REGISTER in the block diagram) will be used to replace the sel_i switch from the board
- the second register (inc_i_freqhigh REGISTER in the block diagram) will be used to write inc_i_freqhigh values in it
- the third register (inc_i_freqlow REGISTER in the block diagram) will be used to write inc_i_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:

- 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 Project option.

- In the Create a New Vivado Project dialog box, click Next to confirm the new project creation.

- 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.

- 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.

- In the Default Part dialog box, ensure that the Sozius board is selected and click Next.

- 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.

Create AXI4 Peripheral using "Create and Package IP" Wizard

- 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 New IP... option.

![Create and Package New IP dialog box](Figure 11.36: Create and Package New IP option)

- In the Create and Package New IP dialog box, click Next.
- In the **Create Peripheral, Package IP or Package a Block Design** dialog box, choose to **Create a new AXI4 peripheral** and click **Next**.

- In the **Peripheral Details** dialog box, give the peripheral an appropriate name (**modulator_axi_ip**), description and location, and click **Next**.
Note: The **Display Name** you provide shows in the Vivado IP Catalog. You can have different names in the **Name** and **Display Name** fields. Any change in the **Name** filed reflects automatically in the **Display Name** filed, which is concatenated with the **Version** field.

- 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 (sel_i, inc_i_freqhigh and inc_i_freqlow), so the last one will be unused. Looking to this, we will stick with the default values and just click **Next**.

- In the last **Create Peripheral** dialog box, select **Edit IP** option and click **Finish**. Another Vivado window will open, which will allow you to modify the peripheral that we just created.
CHAPTER 11. DESIGNING WITH IPs

Figure 11.41: Create Peripheral dialog box

Identification Window

- In the Package IP - modulator_axi_ip window, in the Identification section, fill some basic information about your new modulator_axi_ip IP.

Figure 11.42: 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 11.35 from the beginning of this chapter.

Add Modulator Module Source Files

- In the Flow Navigator, click Add Sources command to add all the necessary Modulator module source files (selector_rtl.vhd, counter_rtl.vhd, modulator_pkg.vhd, sine_rtl.vhd, pwm_rtl.vhd and modulator_rtl.vhd) and after adding your Hierarchy tab should look like as it is shown on the following figure.

www.so-logic.net
Note: In the Add or Create Design Sources dialog box don’t forget to enable Copy sources into IP Directory option.

- Now is the time to modify AXI peripheral. Open the branch "modulator_axi_ip_v1_0".

- Double-click on the "modulator_axi_ip_v1_0_S00_AXI_inst" file to open it.

- In the "modulator_axi_ip_v1_0_S00_AXI.vhd" file make the following changes:
  - Add modulator_pkg package
  - In the entity declaration, add lut_depth_g, lut_width_g, and nco_width_g generics in the generic map, below the first comment line "--Users to add parameters here"
  - In the entity declaration, add pwm_o port as 1-bit output port in the port map, below the comment line "-- Users to add ports here", see Figure 11.45.
CHAPTER 11. DESIGNING WITH IPS

Modify AXI Peripheral - Part 1

- Create temporary signal, `inc_s`, necessary for bit concatenation.
- Create constant `design_setting_c`, as shown on Figure 11.46.

Modify AXI Peripheral - Part 2

- 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 shown on Figure 11.47.
Modify AXI Peripheral - Part 3

- Save the file changes.

- 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 Figure 11.48.

**Open the "modulator_axi_ip_v1_0" File**

- Now, double-click on the "modulator_axi_ip_v1_0.vhd" file to open it.

- In the "modulator_axi_ip_v1_0.vhd" file make the following changes:

  - in the entity declaration, add `lut_depth_g`, `lut_width_g` and `nco_width_g` generics in the generic map, below the first comment line "--Users to add parameters here"

  - in the entity declaration, add `pwm_o` port as 1-bit output port in the port map, below the comment line "-- Users to add ports here", see Figure 11.49.

Modify AXI Peripheral - Part 4
Now, in the `modulator_axi_ip_v1_0_S00_AXI` component declaration add `lut_depth_g`, `lut_width_g` and `nco_width_g` generics in the generic map and `pwm_o` port in the port map, see Figure 11.50.

Modify AXI Peripheral - Part 5

- In the `modulator_axi_ip_v1_0_S00_AXI` component instance assign `lut_depth_g`, `lut_width_g` and `nco_width_g` generics to their values and connect `pwm_o` port of the `modulator_axi_ip_v1_0_S00_AXI` component to the `pwm_o` port of the IP, see Figure 11.51.
Modify AXI Peripheral - Part 6

- Save the file changes.

Compatibility Window

- 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.

File Groups Window

- In the Package IP - modulator_axi_ip window, open File Groups section, and click Merge changes from File Groups Wizard link.
CHAPTER 11. DESIGNING WITH IPS

Figure 11.53: File Groups window

Customization Parameters Window

- 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 following figure.

Figure 11.54: Customization Parameters window after merging changes from Customization Parameters Wizard

*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.

Edit IP Parameters

- In the **Customization Parameters** window, unhide the **Hidden Parameters** and hide the **Customization Parameters**, because we would like to have only **lut_depth_g**, **lut_width_g** and **nco_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 **Visible in Customization GUI** option and click **OK**.

If you would like to hide some IP Parameter, just disable the **Visible in Customization GUI** option in the **Edit IP Parameter** dialog box.
CHAPTER 11. DESIGNING WITH IPS

Figure 11.55: Edit IP Parameter window

Review and Package Window

- Now, open Review and Package section and click Re-Package IP option, see Figure 11.56.

Figure 11.56: 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.

IP Catalog

- Open IP Catalog and search for modulator_axi_ip IP. When you find it, double-click on it to customize and generate the IP.
- In the `modulator_axi_ip_v1.0 (1.0)` customization window, set:

  - **Lut Depth G** to value 12,
  - **Lut Width G** to value 16 and
  - **Nco Width G** to value 31

- click **OK**, see Figure 11.58.

**Customize and Generate IP**

- In the **Generate Output Products** dialog box, click **Generate** to generate the `modulator_axi_ip_0` IP.
- In the **Sources** window expand `modulator_axi_ip_0` IP to see what the tool has created for us.

- 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.

- In the **Sources** window expand all the levels of `modulator_axi_ip_0` IP hierarchy, see Figure 11.59. You can see the structure of the `modulator_axi_ip_0` IP.

  ![Figure 11.59: Sources window with modulator_axi_ip_0 sources hierarchy](image)

**Verification**

- The final step in Modulator IP core development process is the verification of correct operation.

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.

If you want to see all the AXI4-Lite interface signals, please consult "LogiCORE IP AXI4-Lite IPIF" Product Guide for Vivado Design Suite.

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.

The AXI4-Lite interface signals are listed and described in the Table 11.1.

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>I/O</th>
<th>Initial State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AXI Global System Signals</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S_AXI_ACLK</td>
<td>I</td>
<td>-</td>
<td>AXI Clock.</td>
</tr>
<tr>
<td>S_AXI_ARSETN</td>
<td>I</td>
<td>-</td>
<td>AXI Reset, active-low.</td>
</tr>
<tr>
<td><strong>AXI Global System Signals</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S_AXI_AWMADDR[C_S_AXI_ADDR_WIDTH-1:0]</td>
<td>I</td>
<td>-</td>
<td>AXI write address. The write address bus gives the address of the write transaction.</td>
</tr>
<tr>
<td>S_AXI_AWPROT[2:0]</td>
<td>I</td>
<td>-</td>
<td>AXI write address protection signal. &quot;000&quot; value is recommended. Infrastructure IP passes Protection bits across a system.</td>
</tr>
<tr>
<td>S_AXI_AWVALID</td>
<td>I</td>
<td>-</td>
<td>Write address valid. This signal indicates that valid write address and control information are available.</td>
</tr>
</tbody>
</table>

188 2018/11/29 www.so-logic.net
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.
On the following figure AXI4-Lite single write operation timing diagram is presented. Using this diagram, we will create stimulus component in the test bench file for our design.

![AXI4-Lite single write operation timing diagram](image)

Figure 11.60: 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_ARSTEN 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 \texttt{inc\_i\_freqhigh} value (S_AXI_WDATA) in the appropriate register (2nd register, see Figure 11.35). To know what will be the address location of the "inc\_i\_freqhigh" register, we must first understand the structure of S_AXI_AWADDR signal.

**S_AXIS_AWADDR Signal**

S_AXIS_AWADDR is a 4-bit wide signal. AXI address space is byte addressable.

![S_AXIS_AWADDR signal](image)

Figure 11.61: S_AXIS_AWADDR signal

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_AXIS_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.

**Internal 32-bit Registers Address Map of the Modulator IP Core**

In our case, internal 32-bit registers address map will have the following structure:

<table>
<thead>
<tr>
<th>Internal Register Name</th>
<th>S_AXIS_AWADDR Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;sel_i&quot; register</td>
<td>&quot;0000&quot; (0)</td>
</tr>
<tr>
<td>&quot;inc_i_freqhigh&quot;    register</td>
<td>&quot;0100&quot; (4)</td>
</tr>
<tr>
<td>&quot;inc_i_freqlow&quot;     register</td>
<td>&quot;1000&quot; (8)</td>
</tr>
<tr>
<td>&quot;4. unused&quot; register</td>
<td>&quot;1100&quot; (12)</td>
</tr>
</tbody>
</table>

Table 11.2: Internal Registers Address Map of the Modulator IP Core
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 "inc_i_freqhigh" register. We should also validate this address (by setting S_AXI_AWVALID signal to '1') and write desired "inc_i_freqhigh" value in the "inc_i_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 "inc_i_freqlow" value in the "inc_i_freqlow" register by repeating the same procedure. At the end, we will repeat the same procedure once more, to write appropriate value to the "sel_i" register.

The complete test bench file for Modulator IP core with AXI4 interface is shown below.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.modulator_pkg.all;

entity modulator_axi_ip_tb is
  generic(
    -- defines board specific settings
    board_setting_g : board_setting_t := sozius_c;
    -- defines simulation specific settings
    design_setting_g : design_setting_t := sim_setting_c;
  );
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_raddr_s : std_logic_vector(3 downto 0) := (others=>'0');
  signal s00_axi_rprot_s : std_logic_vector(2 downto 0) := (others=>'0');
  signal s00_axi_rvalid_s : std_logic := '0';
  signal s00_axi_rready_s : std_logic := '0';
  -- AXI Global System 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;
  -- pulse width modulated signal
  signal pwm_out_s : std_logic;
begin
  -- period of input clock signal
  constant clock_period_c : time := 1000000000.0 / board_setting_g.fclk * 1 ns;
  constant inc_c : a1integer_t := calc_inc_f(board_setting_g, design_setting_g);
  -- 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,
    );
end;
```
-- 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_arready_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_arready_s <= '1';
  wait until falling_edge(s00_axi_aclk_s);

  -- write inc_i_freqhigh value into appropriate register
  s00_axi_awaddr_s <= "0100";
  s00_axi_awvalid_s <= '1';
  s00_axi_wdata_s <= std_logic_vector(to_unsigned(inc_c(0), 32));
  s00_axi_wvalid_s <= '1';
  s00_axi_wstrb_s <= "1111";
  s00_axi_bready_s <= '1';
  wait until s00_axi_arready_s = '1';
  wait until s00_axi_arready_s = '0';
  wait until falling_edge(s00_axi_aclk_s);
  s00_axi_awaddr_s <= "0000";
  s00_axi_awvalid_s <= '0';
  s00_axi_wdata_s <= std_logic_vector(to_unsigned(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 sel_i=0
  wait for 40 ms;

  -- write value sel_i=1 into appropriate register
  s00_axi_awaddr_s <= "0000";
  s00_axi_awvalid_s <= '1';
  s00_axi_wdata_s <= std_logic_vector(to_unsigned(1, 32));
  s00_axi_wvalid_s <= '1';
  s00_axi_wstrb_s <= "1111";
  s00_axi_bready_s <= '1';
  wait until s00_axi_arready_s = '1';
  wait until s00_axi_arready_s = '0';
  wait until falling_edge(s00_axi_aclk_s);
  s00_axi_awaddr_s <= "0000";
  s00_axi_awvalid_s <= '0';
  s00_axi_wdata_s <= std_logic_vector(to_unsigned(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);

end process;
CHAPTER 11. DESIGNING WITH IPS

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;

Simulation

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

- In the Sources window, under the Simulation Sources / sim_1, select modulator_axi_ip_tb.vhd file.

- In the Flow Navigator, under the Simulation, click on the Run Simulation button.

- Choose the only offered Run Behavioral Simulation option, and your simulation will start.

![Simulation](image)

Figure 11.62: Run Behavioral Simulation option

- The tool will compile the test bench file and launch the Vivado simulator.

**Important:** If your Vivado IDE notify an error that compiler cannot find modulator_pkg.vhd in the default library, that means that compile order in the packaged modulator_axi_ip_0 IP core is not correct. To see the problem and to correct it, please follow the next steps:

- In the Sources window, open Compile Order tab and expand Design Sources. Under the Design Sources you will find a list of VHD1 source files that are packaged in the modulator_axi_ip_0 IP core in default compile order, see Figure 11.63.

![Compile Order](image)

Figure 11.63: Default compile order in modulator_axi_ip_v1_0 IP core

Considering that this compile order is not correct, you have to manually change it in a way that you need.

- Open component.xml file. This file you can find on the following address: ip_repository -> modulator_axi_ip_1.0 -> component.xml.
When you open component.xml file, find the xilinx_vhdlbehavioralsimulation_view_files set section. In that section you will find the default simulation files compile order, see Figure 11.64.

![Figure 11.64: Default simulation files compile order in component.xml file](image)

- Change the simulation files compile order list in the same way as it is shown on the Figure 11.65 and Save the file.

![Figure 11.65: Corrected simulation files compile order in component.xml file](image)
- After we made the changes to the `modulator_axi_ip_0` IP core, we have to refresh it in the IP Catalog. In the **Project Manager** click **Refresh IP Catalog** link, see Figure 11.66.

![Figure 11.66: Refresh IP Catalog link](image)

- In the **IP Status** window, click **Upgrade Selected** button, see Figure 11.67 and the **Enable Core Container** dialog box will appear.

![Figure 11.67: Upgrade Selected option](image)

- In the **Enable Core Container** dialog box, leave **Continue with Core Container Disabled** option selected and click **OK**, see Figure 11.68.

![Figure 11.68: Enable Core Container dialog box](image)

- Before we continue with the simulation process, in the **Sources** window open **Compile Order** tab and check is the compile order now as we said it to be, see Figure 11.69.
- Run Behavioral Simulation process ones more.

**Simulation Settings**

- 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.

- 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.

- Simulate your design for 120 ms.

**Simulation Results - Part 1**

- Go to the beginning of the simulation result, zoom out few times and find the moment where s00_axi_arresetn_s signal is changing from 0 to 1.

- Your simulation results should look like as it is shown on the following figure.
From the simulation results we can see that our system works as we predicted.

Figure 11.70: Simulation results - writing to inc_i_freqhigh and inc_i_freqlow registers
- Zoom fit and then zoom in few times around 40 ms and you will see the "sel_i" register change.

- If you zoom out a few times more, you can also see the pwm frequency change, when sel_i=0 and when
sel_i=1.

Figure 11.72: Simulation Results - pwm signal frequency change as a result of the change of the sel_i register value

Note: All the information about how to create a design with AXI4 interface, how to modify existing design to adjust it to the requirements of the AXI4 interface, how to create a new AXI4 peripheral, how to connect existing design with the AXI interface, how to verify your IP with AXI4 interface, you can also find in the Lab 18: "Creating Modulator IP Core with AXI4 Interface".