See all versions of this document

Vivado Design Suite User Guide: Model-Based DSP Design Using System Generator

UG897 (v2020.2) November 18, 2020

XILINX®

XILINX

Table of Contents

Revision History 5
Supported MATLAB versions and Operating Systems 6

Chapter 1: Introduction 7
Navigating Content by Design Process 8
Xilinx DSP Block Set 8
FIR Filter Generation 9
Support for MATLAB 10
Hardware Co-Simulation 11
System Integration Platform 13
Operating System, MATLAB, and Simulator Support in System Generator 13

Chapter 2: Installation 14
Downloading 14
Using the Xilinx Installer 14
Post Installation Tasks 16

Chapter 3: Hardware Design Using System Generator 20
Design Flows Using System Generator 21
System-Level Modeling in System Generator 23
Automatic Code Generation 40
Compiling MATLAB into an FPGA 50
Importing a System Generator Design into a Bigger System 72
Configurable Subsystems and System Generator 73
Notes for Higher Performance FPGA Design 79
Using FDATool in Digital Filter Applications 84
Multiple Independent Clocks Hardware Design 88
AXI Interface 97
AXI4-Lite Slave Interface Generation 102
Tailor Fitting a Platform Based Accelerator Design in System Generator 116
Using Super Sample Rate (SSR) Blocks in System Generator 122

Chapter 4: Performing Analysis in System Generator......127
Timing Analysis in System Generator......128
Resource Analysis in System Generator......136

Chapter 5: Using Hardware Co-Simulation......146
Compiling a Model for Hardware Co-Simulation......147
Performing Standard Hardware Co-Simulation......151
Performing Burst Mode Hardware Co-Simulation......155
M-Code Access to Hardware Co-Simulation......157
Setting Up Your Hardware Board......157
Hardware Co-Simulation Blocks......162
Hardware Co-Simulation Clocking......166
Point-to-Point Ethernet Hardware Co-Simulation......168
Burst Data Transfers for Hardware Co-Simulation......173

Chapter 6: Importing HDL Modules......181
Black Box HDL Requirements and Restrictions......181
Black Box Configuration M-Function......182
Multiple Independent Clock Support on Black Box......197
HDL Co-Simulation......199

Chapter 7: Black Box Configuration Wizard......202
Using the Configuration Wizard......202
Configuration Wizard Fine Points......203

Chapter 8: System Generator Compilation Types......205
HDL Netlist Compilation......205
Hardware Co-Simulation Compilation......207
IP Catalog Compilation......208
Synthesized Checkpoint Compilation......214
Creating Your Own Custom Compilation Target......215

Chapter 9: Creating Custom Compilation Targets......216
xilinx_compilation Base Class......216
Creating a New Compilation Target......217
Base Class Properties and APIs......219
Examples of Creating Custom Compilation Targets......222

Appendix A: System Generator GUI Utilities......229

Xilinx BlockAdd... 230
Xilinx Tools > Save as blockAdd default.. 231
Xilinx BlockConnect... 232
Xilinx Tools > Terminate.. 234
Xilinx Waveform Viewer.. 237

Appendix B: Additional Resources and Legal Notices.......................... 249

Xilinx Resources... 249
Solution Centers... 249
Documentation Navigator and Design Hubs.. 249
References... 250
Please Read: Important Legal Notices... 250

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
11/18/2020 Version 2020.2	
General Updates	• Update to clarify note in Black Box HDL Requirements and Restrictions. • Clarified Choosing MATLAB for System Generator.
Added DSP58 Support	• Performing Resource Analysis. • FIR Filter Generation.
06/03/2020 Version 2020.1	
General Updates	• Updates to MATLAB® command prompts. • Note added to Accessing Resource Analysis Results. • Update Xilinx BlockAdd.

Supported MATLAB versions and Operating Systems

System Generator supports MATLAB® versions:

- 2019a
- 2019b
- 2020a
- 2020b

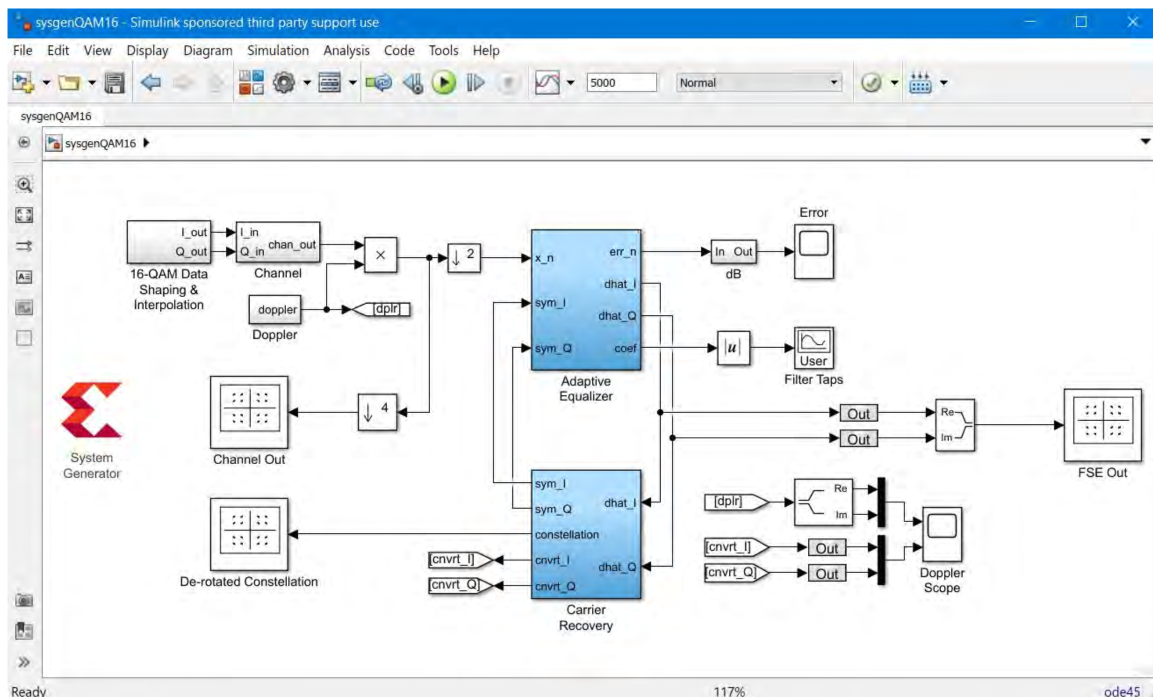
The following operating systems are supported on x86 and x86-64 processor architectures:

- **Windows 10 Pro and Enterprise (64-bit):** 10.0 1809 Update; 10.0 1903 Update; 10.0 1909 Update; 10.0 2004 Update
- **Red Hat Enterprise Workstation/Server 7 (64-bit):** 7.4 (Matlab R2019a, R2019b only), 7.5, 7.6, 7.7, 7.8
- **Ubuntu Linux (64-bit):** 16.04.5 LTS;16.04.6 LTS; 18.04.1 LTS; 18.04.2 LTS, 18.04.3 LTS; 18.04.4 LTS;
- **SUSE Enterprise Linux 12 (64-bit):** 12.4

Introduction

System Generator is a DSP design tool from Xilinx® that enables the use of the MathWorks model-based Simulink® design environment for FPGA design. Previous experience with Xilinx FPGAs or RTL design methodologies are not required when using System Generator. Designs are captured in the DSP friendly Simulink modeling environment using a Xilinx specific blockset. The System Generator design can then be imported into a Vivado® IDE project using the IP catalog.

Figure 1: **System Generator Design**



Refer to the *Vivado Design Suite Tutorial: Model-Based DSP Design Using System Generator* (UG948) for hands-on lab exercises and step-by-step instruction on how to create a System Generator for DSP model and then import that model into a Vivado IDE project.

Navigating Content by Design Process

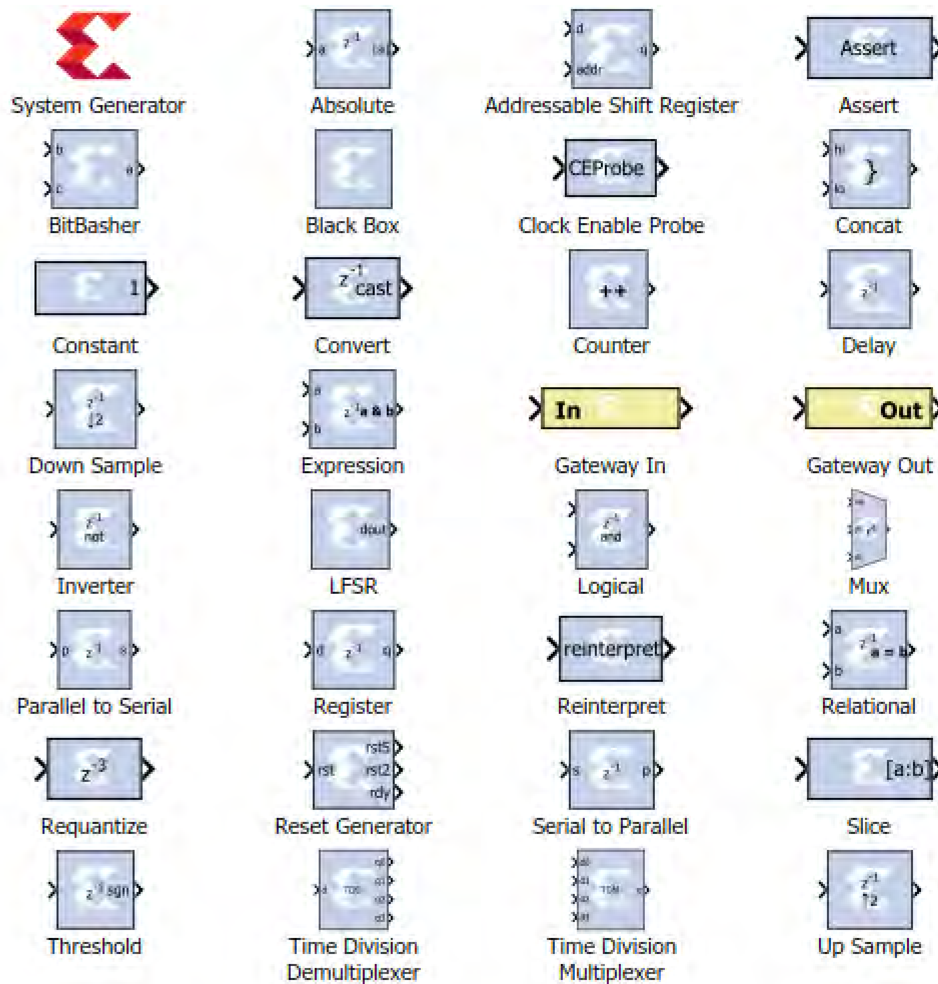
Xilinx® documentation is organized around a set of standard design processes to help you find relevant content for your current development task. This document covers the following design processes:

- **Hardware, IP, and Platform Development:** Creating the PL IP blocks for the hardware platform, creating PL kernels, subsystem functional simulation, and evaluating the Vivado® timing, resource use, and power closure. Also involves developing the hardware platform for system integration. Topics in this document that apply to this design process include:
  - System-Level Modeling in System Generator
  - Importing a System Generator Design into a Bigger System
  - Using Super Sample Rate (SSR) Blocks in System Generator
  - Chapter 5: Using Hardware Co-Simulation

Xilinx DSP Block Set

Over 130 DSP building blocks are provided in the Xilinx® DSP blockset for Simulink®. These blocks include the common DSP building blocks such as adders, multipliers and registers. Also included are a set of complex DSP building blocks such as forward error correction blocks, FFTs, filters and memories. These blocks leverage the Xilinx® IP core generators to deliver optimized results for the selected device.

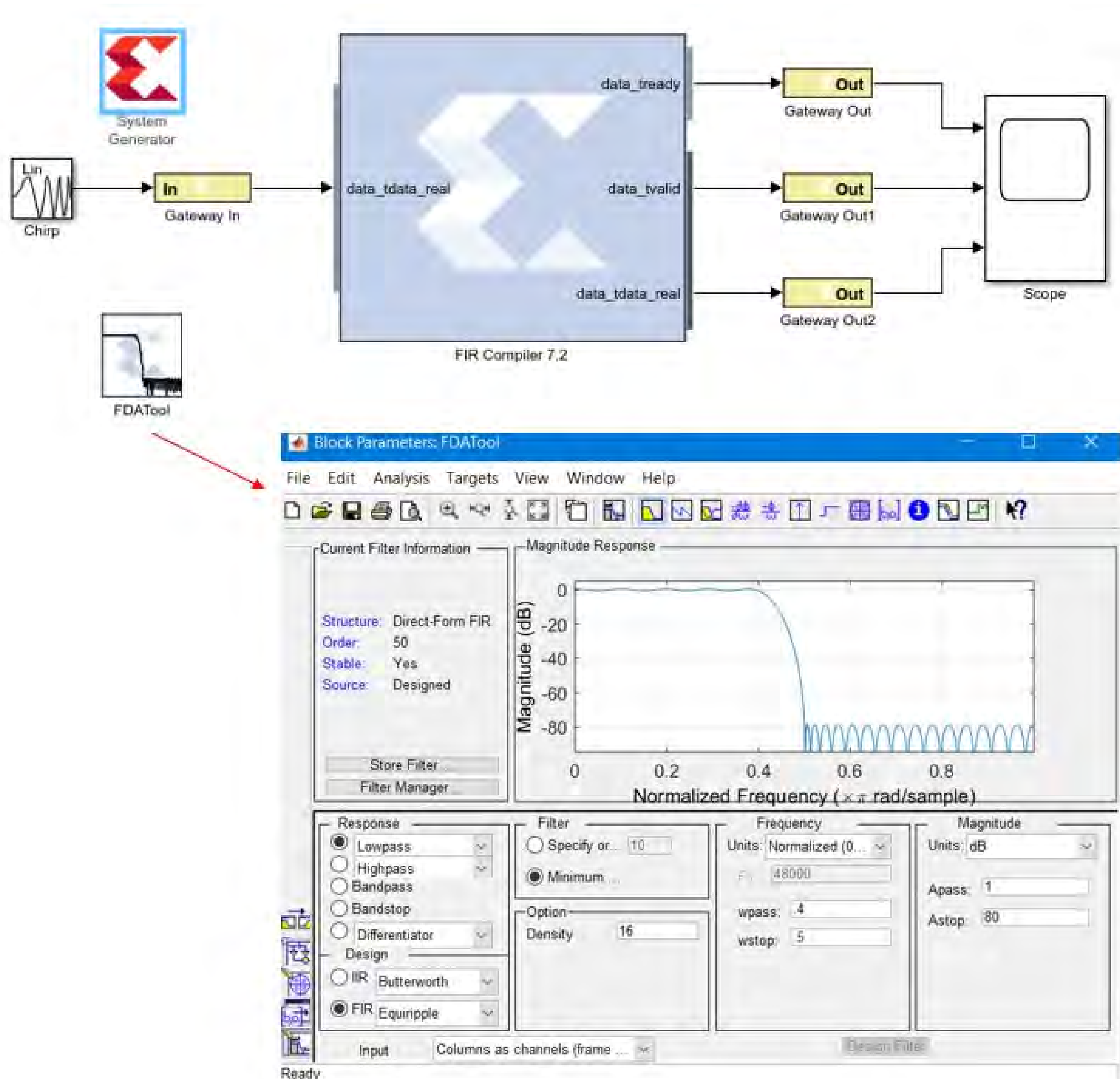
Figure 2: **Xilinx DSP Block Set**



FIR Filter Generation

System Generator includes a FIR Compiler block that targets the dedicated DSP48E1, DSP48E2, and DSP58 hardware resources in the 7 series, UltraScale™ and Versal™ devices respectively to create highly optimized implementations. Configuration options allow generation of single rate, interpolation, decimation, Hilbert, and interpolated implementations. Standard MATLAB® functions such as fir2 or the MathWorks FDA tool can be used to create coefficients for the Xilinx® FIR Compiler.

Figure 3: **FDA Tool Example**

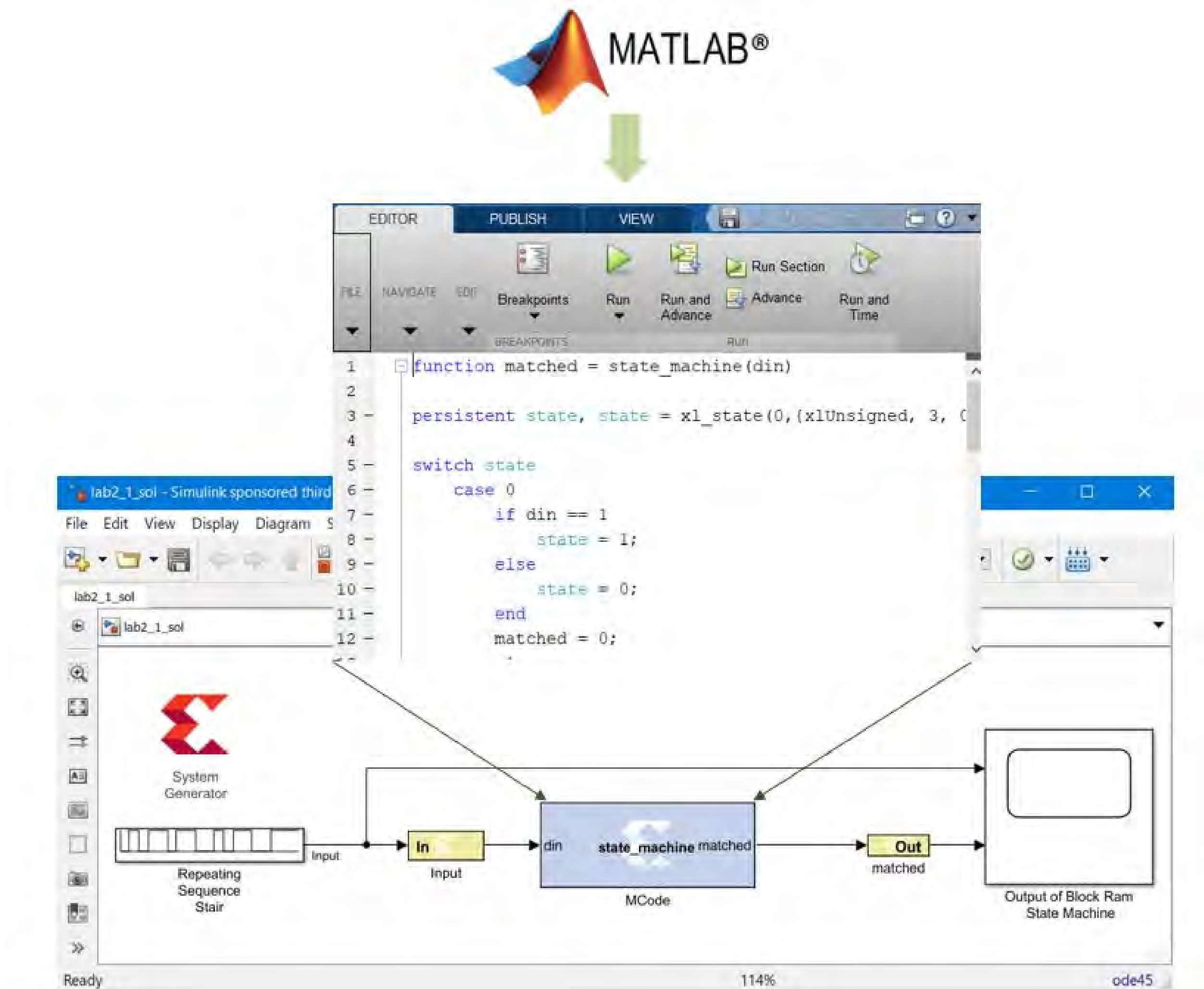


Support for MATLAB

System Generator library consists of an MCode block that allows the use of non-algorithmic MATLAB® for the modeling and implementation of simple control operations.

The MATLAB releases supported in this release of System Generator are described in Supported MATLAB versions and Operating Systems.

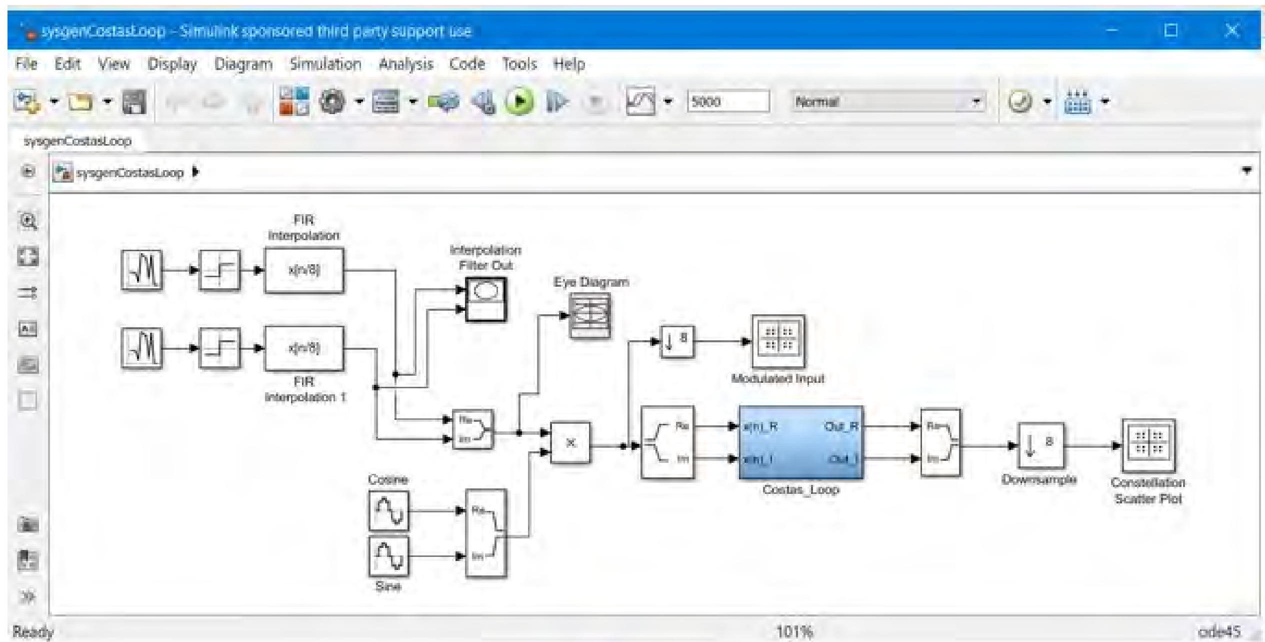
Figure 4: **MCode Block Example**



Hardware Co-Simulation

System Generator provides accelerated simulation through hardware co-simulation. System Generator will automatically create a hardware simulation token for a design captured in the Xilinx® DSP blockset that will run on supported hardware platforms. This hardware will co-simulate with the rest of the Simulink® system to provide up to a 1000x simulation performance increase.

Figure 5: **Hardware Co-Simulation**



System Integration Platform

System Generator provides a system integration platform for the design of DSP FPGAs that allows the RTL, Simulink®, MATLAB® and C/C++ components of a DSP system to come together in a single simulation and implementation environment. System Generator supports a black box block that allows RTL to be imported into Simulink and co-simulated with either ModelSim or Xilinx® Vivado® simulator, and provides a Vitis™ HLS block that allows integration and simulation of C/C++ sources.

Operating System, MATLAB, and Simulator Support in System Generator

The operating systems supported in this release of System Generator are described in the Operating Systems section of the *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing* (UG973).

The MATLAB® releases and simulation tools supported in this release of System Generator are described in the Compatible Third-Party Tools section of the *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing* (UG973).

Installation

Downloading

System Generator is part of the Vivado® Design Suite and may be downloaded from the Xilinx website. You may purchase, register, and download the System Generator software from the System Generator for DSP page on the Xilinx website.

Hardware Co-Simulation Support

If you have an FPGA development board, you may be able to take advantage of System Generator's ability to use FPGA hardware co-simulation with Simulink® simulations. The System Generator software includes support for all Xilinx® Development Boards. System Generator board support packages can be downloaded from the Boards and Kits page on the Xilinx website.

UNC Paths Not Supported

System Generator does not support UNC (Universal Naming Convention) paths. For example System Generator cannot operate on a design that is located on a shared network drive without mapping to the drive first.

Using the Xilinx Installer

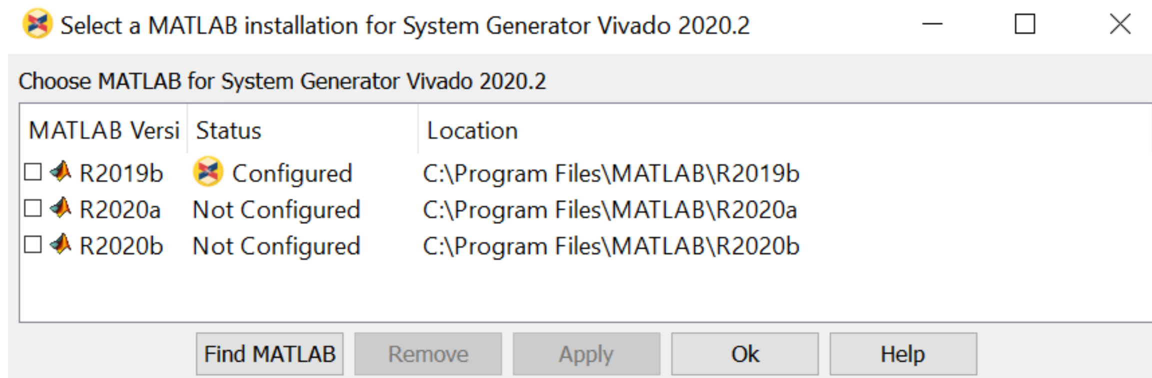
System Generator for DSP is part of the Vivado® Design Suite. You must use the Xilinx® Design Tools installer to install System Generator.

Before invoking the Xilinx Design Tools installer, it is a good idea to make sure that all instances of MATLAB® are closed. When all instances of MATLAB are closed, launch the installer and follow the directions on the screen.

Choosing MATLAB for System Generator

Windows Installations

Figure 6: **Choosing MATLAB**



This dialog box allows you to associate any supported MATLAB® installation with this version of System Generator.

Click the check box of the MATLAB installation(s) you wish to associate with this version of System Generator, select the Xilinx® Design Suite you wish to associate with, then click **Apply**. Once the Apply operation is completed, the value in the Status column changes from “Not Configured” to “Configured”.

The application lists all the available MATLAB installations. The Status field shows one of the following values:

- **Unsupported:** This version of MATLAB is not supported with this version of System Generator.
- **Not Configured:** This version of MATLAB is not yet associated with this version of System Generator. To associate this version of MATLAB with System Generator, click the check box and then click **Apply**.
- **Configured:** System Generator is now ready to be used with this version of MATLAB.

If you do not see a version of MATLAB listed, click **Find MATLAB** to browse for a valid version.

If you wish to change the MATLAB configuration, open **System Generator MATLAB Configurator 2020.2**.

If MATLAB is configured for a Design Suite, for example, the ISE Design Suite, and you wish to re-configure MATLAB for another Design Suite, for example, Vivado® IDE, you must select the Configured MATLAB version box and click **Remove** before you re-configure for Vivado IDE.

Linux Installations

Launching System Generator under Linux is handled via a shell script called `sysgen` located in the `<Vivado_install_dir>/bin` directory. Before launching this script, you must make sure the MATLAB executable can be found in the PATH environment variable. Once the MATLAB executable can be found, executing System Generator will launch the first MATLAB executable found in PATH and attach System Generator to that session of MATLAB. Also, the System Generator shell script supports all the options that MATLAB supports and can be passed as command line arguments to the System Generator script.

Post Installation Tasks

Post-Installation Tasks on Linux

After following the directions of the main Xilinx® Installation Wizard, you are ready to launch System Generator by typing: `sysgen`

This will invoke MATLAB® and dynamically add System Generator to that MATLAB session. At the top of the MATLAB command window, you will see the following:

```
Type "xlDoc" to open the Xilinx System Generator help documentation.
Type "demo blockset xilinx" to view the demos available for Xilinx System
Generator.
```

Compiling Xilinx HDL Libraries

The Xilinx tool that compiles libraries for use in ModelSim SE is named `compile_simlib`.

To compile the Xilinx HDL libraries, launch the Vivado Design Suite and then enter `compile_simlib` in the Vivado Tcl console.

Note: You can enter `compile_simlib -help` in the Vivado Tcl Console for more details on executing this Tcl command.

Example Designs Installed with System Generator

System Generator includes example models that can be accessed from the Xilinx section of the documentation available from the Help menu in the MATLAB command window (**Help → Documentation → Xilinx**), or by typing this command from the MATLAB command prompt:

```
>> demo blockset xilinx
```

Managing the System Generator Cache

System Generator incorporates a disk cache to speed up the iterative design process. The cache does this by tagging and storing files related to simulation and generation, then recalling those files during subsequent simulation and generation rather than rerunning the time consuming tools used to create those files.

Specifying Board Support in System Generator

When System Generator is installed on your system as part of a Vivado Design Suite installation, System Generator will have access to any Xilinx® development boards installed with the Vivado Design Suite.

Additional boards from Xilinx partners are available and a Board Interface file that defines a board (`board.xml`) can be downloaded from a partner website and installed as part of the Vivado Design Suite. You can also create custom Board Interface files, as detailed in Appendix A, Board Interface File, in the *Vivado Design Suite User Guide: System-Level Design Entry* (UG895). Both the Vivado Design Suite and System Generator must be configured to add partner boards and custom boards to the repository of boards available for use.

The procedure for configuring the Vivado Design Suite for use with boards is detailed in Using the Vivado Design Suite Platform Board Flow in the *Vivado Design Suite User Guide: System-Level Design Entry* (UG895). The Vivado Design Suite lets you create projects using Xilinx target design platform boards (TDP), or user-specified boards that have been added to a board repository. When you select a specific board, the Vivado Design Suite tools show information about the board, and enable additional designer assistance as part of IP customization, and for IP integrator designs.

To configure System Generator for using a partner board or custom board, you must add commands to MATLAB®'s `startup.m` file, a file you create for commands to be executed when MATLAB starts up.

To make a board available to your Simulink® models in System Generator:

1. At the MATLAB command line, enter the command `which startup.m` to determine if your MATLAB installation already has a `startup.m` file.

   The `which startup.m` command searches through the folders in the MATLAB search path to find a `startup.m` file. If there is a `startup.m` file in the search path, `which startup.m` displays the full path for the file.

2. Proceed as follows:

   - If your MATLAB installation *does* have a `startup.m` file, enter the command `edit startup.m` at the command line to open the `startup.m` file for editing.

     OR

- If your MATLAB installation *does not* have a `startup.m` file, create a `startup.m` file in a folder in the MATLAB search path and open the file for editing.

The command `path` prints a listing of the folders in the search path.

3. Enter the following commands in your `startup.m` file:

```
addpath([getenv('XILINX_VIVADO') '/scripts/sysgen/matlab']);
xilinx.environment.setBoardFileRepos({'<path1>', '<path2>', '...'}];
```

where the `addpath` command specifies the location of the `setBoardFileRepos` utility and `setBoardFileRepos` points MATLAB to the location of Board Interface files. `<path>` is the path to a folder containing a Board Interface file (`board.xml`) and files referenced by the `board.xml` file, such as `part0_pins.xml` and `preset.xml`. The `<path>` can also specify a folder with multiple subdirectories, each containing a separate Board Interface file.

For example:

```
addpath([[getenv('XILINX_VIVADO')] '/scripts/sysgen/matlab']);
xilinx.environment.setBoardFileRepos({'C:/Data/userBoards', 'C:/Data/
otherBoards'});
```

4. Close the `startup.m` file (which is in a directory in the MATLAB search path) and close System Generator.

When you open System Generator, each of the partner or custom boards is available as a target board (and target Xilinx device) for your System Generator design.

To determine what partner or custom boards are available in System Generator, enter this command in the MATLAB command window:

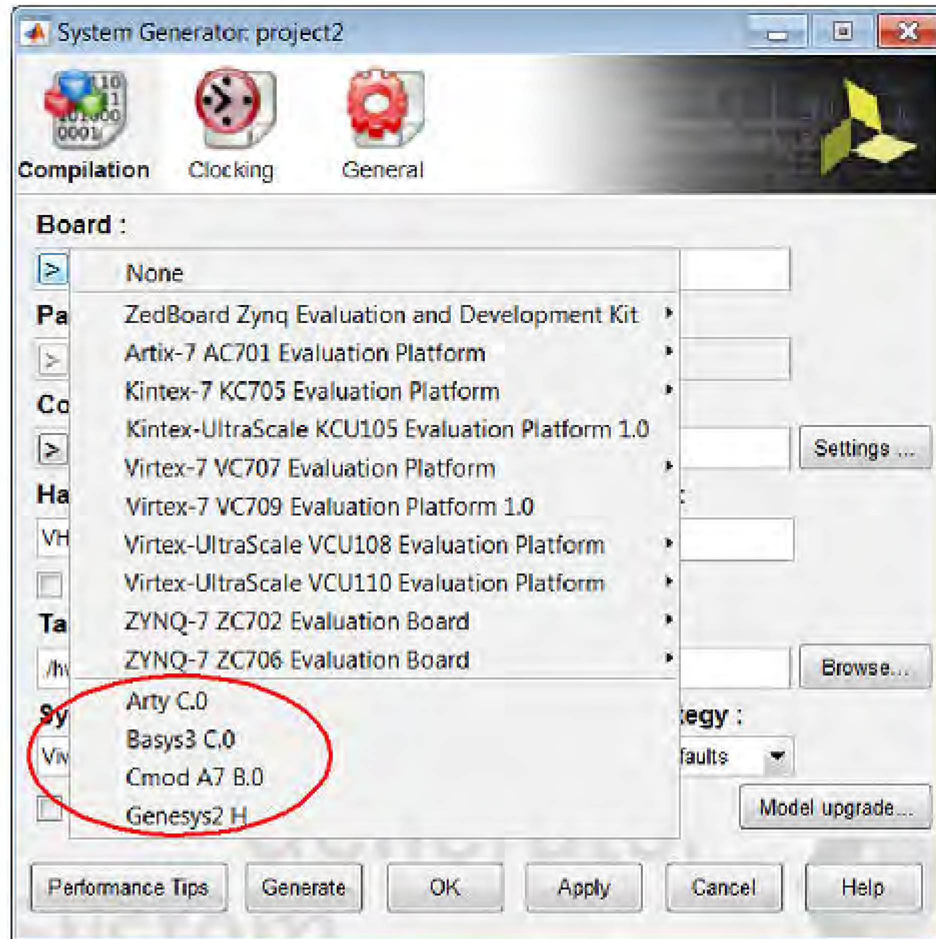
```
xilinx.environment.getBoardFiles
```

A listing of Board Interface files will display in the command window.

```
>> xilinx.environment.getBoardFiles

ans =

    'C:\Data\usrBrds\arty\C.0\board.xml'
    'C:\Data\usrBrds\basys3\C.0\board.xml'
    'C:\Data\usrBrds\cmod_a7\B.0\board.xml'
    'C:\Data\usrBrds\genesys2\H\board.xml'
```

You can also determine what partner or custom boards are available in System Generator by opening a Simulink® model and double-clicking the model's System Generator token. The added boards will appear in the System Generator token properties dialog box as a **Board** selection:

Figure 7: **Board Selection**



To add an additional board to your board repository, you can modify the `xilinx.environment.setBoardFileRepos` line in your `startup.m` file to point to the location of the new Board Interface file (`board.xml`). If you place the Board Interface file in a subdirectory under a folder already specified in the `xilinx.environment.setBoardFileRepos` line, the new board will be available the next time you open System Generator, without having to make any changes to the `startup.m` file.

XILINX®

Chapter 3

Hardware Design Using System Generator

System Generator is a system-level modeling tool that facilitates FPGA hardware design. It extends Simulink® in many ways to provide a modeling environment that is well suited to hardware design. The tool provides high-level abstractions that are automatically compiled into an FPGA at the push of a button. The tool also provides access to underlying FPGA resources through low-level abstractions, allowing the construction of highly efficient FPGA designs.

Design Flows Using System Generator	Describes several settings in which constructing designs in System Generator is useful.
System-Level Modeling in System Generator	Discusses System Generator's ability to implement device-specific hardware designs directly from a flexible, high-level, system modeling environment.
Automatic Code Generation	Discusses automatic code generation for System Generator designs.
Compiling MATLAB into an FPGA	Describes how to use a subset of the MATLAB programming language to write functions that describe state machines and arithmetic operators. Functions written in this way can be attached to blocks in System Generator and can be automatically compiled into equivalent HDL.
Importing a System Generator Design into a Bigger System	Discusses how to take the VHDL netlist from a System Generator design and synthesize it in order to embed it into a larger design. Also shows how VHDL created by System Generator can be incorporated into a simulation model of the overall system.
Configurable Subsystems and System Generator	Explains how to use configurable Subsystems in System Generator. Describes common tasks such as defining configurable Subsystems, deleting and adding blocks, and using configurable Subsystems to import compilation results into System Generator designs.
Notes for Higher Performance FPGA Design	Suggests design practices in System Generator that lead to an efficient and high-performance implementation in an FPGA.
Using FDATool in Digital Filter Applications	Demonstrates one way to specify, implement and simulate a FIR filter using the FDATool block.
Multiple Independent Clocks Hardware Design	The design can be partitioned into groups of Subsystem blocks, where each Subsystem has a common cycle period, independent of the cycle period of other Subsystems.
AXI Interface	Provides an introduction to AMBA AXI4 and draws attention to AMBA AXI4 details with respect to System Generator.
AXI4-Lite Slave Interface Generation	Describes features in System Generator that allow you to create a standard AXI4-Lite interface for a System Generator module and then export the module to the Vivado® IP catalog for later inclusion in a larger design using IP integrator.

Tailor Fitting a Platform Based Accelerator Design in System Generator	Describes how to develop an accelerator in System Generator which is part of a platform framework developed in the Vivado IP Integrator.

Design Flows Using System Generator

System Generator can be useful in many settings. Sometimes you may want to explore an algorithm without translating the design into hardware. Other times you might plan to use a System Generator design as part of something bigger. A third possibility is that a System Generator design is complete in its own right, and is to be used in FPGA hardware. This topic describes all three possibilities.

Algorithm Exploration

System Generator is particularly useful for algorithm exploration, design prototyping, and model analysis. When these are the goals, you can use the tool to flesh out an algorithm in order to get a feel for the design problems that are likely to be faced, and perhaps to estimate the cost and performance of an implementation in hardware. The work is preparatory, and there is little need to translate the design into hardware.

In this setting, you assemble key portions of the design without worrying about fine points or detailed implementation. Simulink blocks and MATLAB M-code provide stimuli for simulations, and for analyzing results. Resource estimation gives a rough idea of the cost of the design in hardware. Experiments using hardware generation can suggest the hardware speeds that are possible.

Once a promising approach has been identified, the design can be fleshed out. System Generator allows refinements to be done in steps, so some portions of the design can be made ready for implementation in hardware, while others remain high-level and abstract. System Generator's facilities for hardware co-simulation are particularly useful when portions of a design are being refined.

Implementing Part of a Larger Design

Often System Generator is used to implement a portion of a larger design. For example, System Generator is a good setting in which to implement data paths and control, but is less well suited for sophisticated external interfaces that have strict timing requirements. In this case, it may be useful to implement parts of the design using System Generator, implement other parts outside, and then combine the parts into a working whole.

A typical approach to this flow is to create an HDL wrapper that represents the entire design, and to use the System Generator portion as a component. The non-System Generator portions of the design can also be components in the wrapper, or can be instantiated directly in the wrapper.

Implementing a Complete Design

Many times, everything needed for a design is available inside System Generator. For such a design, pressing the **Generate** button instructs System Generator to translate the design into HDL, and to write the files needed to process the HDL using downstream tools. The files written include the following:

- HDL that implements the design itself.
- An HDL test bench. The test bench allows results from Simulink simulations to be compared against ones produced by a logic simulator.
- Files that allow the System Generator HDL to be used as a Vivado IDE project.

For details concerning the files that System Generator writes, see the topic Compilation Results.

Note to the DSP Engineer

System Generator extends Simulink to enable hardware design, providing high-level abstractions that can be automatically compiled into an FPGA. Although the arithmetic abstractions are suitable to Simulink (discrete time and space dynamical system simulation), System Generator also provides access to features in the underlying FPGA.

The more you know about a hardware realization (e.g., how to exploit parallelism and pipelining), the better the implementation you'll obtain. Using IP cores makes it possible to have efficient FPGA designs that include complex functions like FFTs. System Generator also makes it possible to refine a model to more accurately fit the application.

Scattered throughout the System Generator documentation are notes that explain ways in which system parameters can be used to exploit hardware capabilities.

Note to the Hardware Engineer

System Generator does not replace hardware description language (HDL)-based design, but does makes it possible to focus your attention only on the critical parts. By analogy, most DSP programmers do not program exclusively in assembler; they start in a higher-level language like C, and write assembly code only where it is required to meet performance requirements.

A good rule of thumb is this: in the parts of the design where you must manage internal hardware clocks (e.g., using DDR or phased clocking), you should implement using HDL. The less critical portions of the design can be implemented in System Generator, and then the HDL and System Generator portions can be connected. Usually, most portions of a signal processing system do not need this level of control, except at external interfaces. System Generator provides mechanisms to import HDL code into a design (see Chapter 6: Importing HDL Modules) that are of particular interest to the HDL designer.

Another aspect of System Generator that is of interest to the engineer who designs using HDL is its ability to automatically generate an HDL test bench, including test vectors. This aspect is described in the topic HDL Testbench.

Finally, the hardware co-simulation interfaces described in the topic Chapter 5: Using Hardware Co-Simulation allow you to run a design in hardware under the control of Simulink, bringing the full power of MATLAB and Simulink to bear for data analysis and visualization.

System-Level Modeling in System Generator

System Generator allows device-specific hardware designs to be constructed directly in a flexible high-level system modeling environment. In a System Generator design, signals are not just bits. They can be signed and unsigned fixed-point numbers, and changes to the design automatically translate into appropriate changes in signal types. Blocks are not just stand-ins for hardware. They respond to their surroundings, automatically adjusting the results they produce and the hardware they become.

System Generator allows designs to be composed from a variety of ingredients. Data flow models, traditional hardware description languages (VHDL and Verilog), and functions derived from the MATLAB programming language, can be used side-by-side, simulated together, and synthesized into working hardware. System Generator simulation results are bit and cycle-accurate. This means results seen in simulation exactly match the results that are seen in hardware. System Generator simulations are considerably faster than those from traditional HDL simulators, and results are easier to analyze.

System Generator Blocksets	Describes how System Generator's blocks are organized in libraries, and how the blocks can be parameterized and used.
Xilinx Commands that Facilitate Rapid Model Creation and Analysis	Introduces Xilinx commands that have been added to the Simulink pop-up menu that facilitate rapid System Generator model creation and analysis.
Signal Types	Describes the data types used by System Generator and ways in which data types can be automatically assigned by the tool.
Bit-True and Cycle-True Modeling	Specifies the relationship between the Simulink-based simulation of a System Generator model and the behavior of the hardware that can be generated from it.
Timing and Clocking	Describes how clocks are implemented in hardware, and how their implementation is controlled inside System Generator. Explains how System Generator translates a multirate Simulink model into working clock-synchronous hardware.
Synchronization Mechanisms	Describes mechanisms that can be used to synchronize data flow across the data path elements in a high-level System Generator design, and describes how control path functions can be implemented.
Block Masks and Parameter Passing	Explains how parameterized systems and Subsystems are created in Simulink.

System Generator Blocksets

A Simulink® blockset is a library of blocks that can be connected in the Simulink block editor to create functional models of a dynamical system. For system modeling, System Generator blocksets are used like other Simulink blocksets. The blocks provide abstractions of mathematical, logical, memory, and DSP functions that can be used to build sophisticated signal processing (and other) systems. There are also blocks that provide interfaces to other software tools (e.g. FDATool, ModelSim) as well as the System Generator code generation software.

System Generator blocks are *bit-accurate* and *cycle-accurate*. Bit-accurate blocks produce values in Simulink that match corresponding values produced in hardware; cycle-accurate blocks produce corresponding values at corresponding times.

Xilinx Blockset

The Xilinx® Blockset is a family of libraries that contain basic System Generator blocks. Some blocks are low-level, providing access to device-specific hardware. Others are high- level, implementing (for example) signal processing and advanced communications algorithms. For convenience, blocks with broad applicability (e.g., the Gateway I/O blocks) are members of several libraries. Every block is contained in the Index library. The libraries are described below.

Library	Description
AXI4	Blocks with interfaces that conform to the AXI4 specification
Basic Elements	Standard building blocks for digital logic
Communication	Forward error correction and modulator blocks, commonly used in digital communications systems
Control Logic	Blocks for control circuitry and state machines
Data Types	Blocks that convert data types (includes gateways)
DSP	Digital signal processing (DSP) blocks
Floating-Point	Blocks that support the Floating-Point data type
Index	Every block in the Xilinx Blockset
Math	Blocks that implement mathematical functions
Memory	Blocks that implement and access memories
Tools	“Utility” blocks, e.g. code generation (System Generator token), resource estimation, HDL co-simulation, etc.

Xilinx Reference Blockset

The Xilinx® Reference Blockset contains composite System Generator blocks that implement a wide range of functions. Blocks in this blockset are organized by function into different libraries. The libraries are described below.

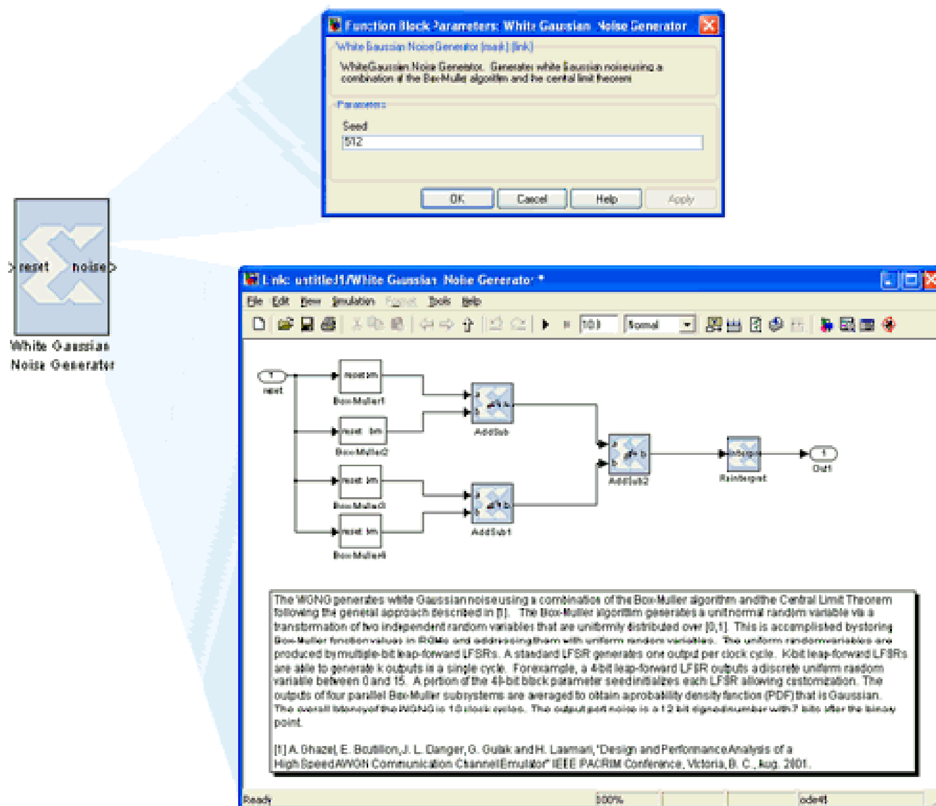
Library	Description
Communication	Blocks commonly used in digital communications systems
Control Logic	Logic Blocks used for control circuitry and state machines

Library	Description
DSP	Digital signal processing (DSP) blocks
Imaging	Image processing blocks
Math	Blocks that implement mathematical functions

Each block in this blockset is a composite, i.e., is implemented as a masked Subsystem, with parameters that configure the block.

You can use blocks from the Reference Blockset libraries as is, or as starting points when constructing designs that have similar characteristics. Each reference block has a description of its implementation and hardware resource requirements.

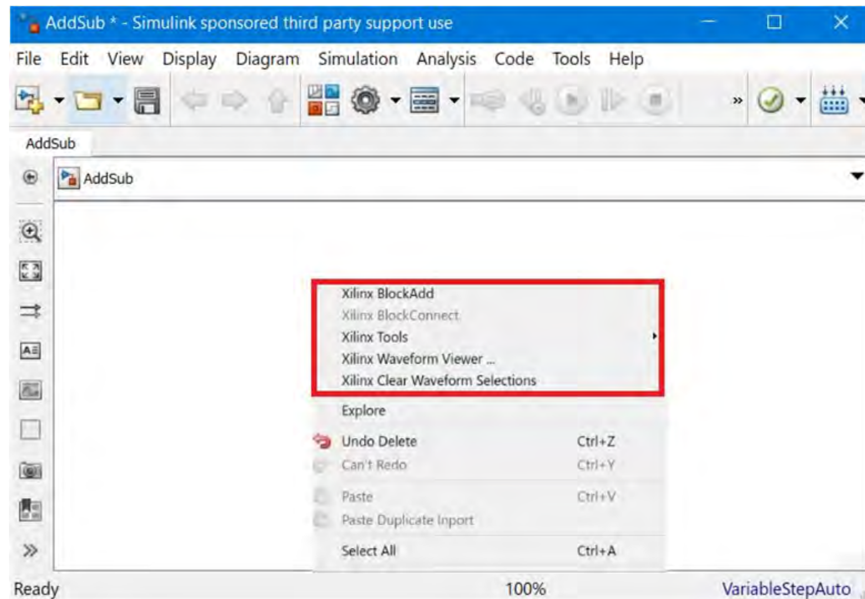
Figure 8: **Reference Blockset Library**



Xilinx Commands that Facilitate Rapid Model Creation and Analysis

Xilinx has added graphics commands to the Simulink® popup menu that will help you rapidly create and analyze your System Generator design. As shown below, you can access these commands by right-clicking the Simulink model canvas and selecting the appropriate Xilinx command:

Figure 9: **Xilinx Commands**



Details on how to use these additional Xilinx commands are provided in the topics for each individual command.

Signal Types

In order to provide bit-accurate simulation of hardware, System Generator blocks operate on Boolean, floating-point, and arbitrary precision fixed-point values. By contrast, the fundamental scalar signal type in Simulink® is double precision floating point. The connection between Xilinx blocks and non-Xilinx blocks is provided by gateway blocks. The Gateway In converts a double precision signal into a Xilinx signal, and the Gateway Out converts a Xilinx signal into double precision. Simulink® continuous time signals must be sampled by the Gateway In block.

Most Xilinx blocks are polymorphic, i.e., they can deduce appropriate output types based on their input types. When *full precision* is specified for a block in its parameters dialog box, System Generator chooses the output type to ensure no precision is lost. Sign extension and zero padding occur automatically as necessary. *User-specified precision* is usually also available. This allows you to set the output type for a block and to specify how quantization and overflow should be handled. Quantization possibilities include unbiased rounding towards plus or minus infinity, depending on sign, or truncation. Overflow options include saturation, truncation, and reporting overflow as an error.

Note: System Generator data types can be displayed by selecting **Display → Signals & Ports → Port Data Types** in Simulink. Displaying data types makes it easy to determine precision throughout a model. If, for example, the type for a port is **Fix_11_9**, then the signal is a two's complement signed 11-bit number having nine fractional bits. Similarly, if the type is **Ufix_5_3**, then the signal is an unsigned 5-bit number having three fractional bits.

In the System Generator portion of a Simulink model, every signal must be sampled. Sample times may be inherited using Simulink's propagation rules, or set explicitly in a block customization dialog box. When there are feedback loops, System Generator is sometimes unable to deduce sample periods and/or signal types, in which case the tool issues an error message. Assert blocks must be inserted into loops to address this problem. It is not necessary to add assert blocks at every point in a loop; usually it suffices to add an assert block at one point to "break" the loop.

***Note*:** Simulink can display a model by shading blocks and signals that run at different rates with different colors (**Display → Sample Time → Colors** in the Simulink pulldown menus). This is often useful in understanding multirate designs.

Floating-Point Data Type

System Generator blocks found in the Floating-Point library support the floating-point data type.

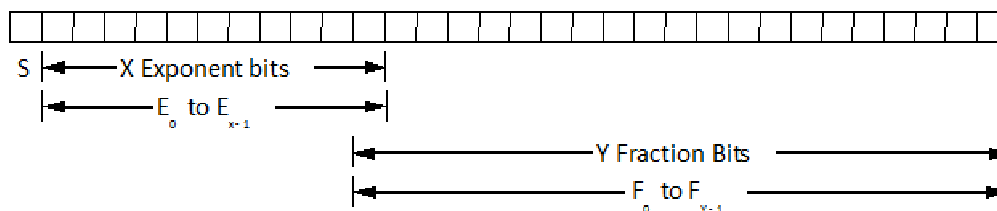
System Generator uses the Floating-Point Operator v7.1 IP core to leverage the implementation of operations such as addition/subtraction, multiplication, comparisons and data type conversion.

The floating-point data type support is in compliance with IEEE-754 Standard for Floating-Point Arithmetic. Single precision, Double precision and Custom precision floating-point data types are supported for design input, data type display and for data rate and type propagation (RTP) across the supported System Generator blocks.

IEEE-754 Standard for Floating-Point Data Type

As shown below, floating-point data is represented using one Sign bit (S), X exponent bits and Y fraction bits. The Sign bit is always the most-significant bit (MSB).

Figure 10: **Floating-Point Data**



According to the IEEE-754 standard, a floating-point value is represented and stored in the normalized form. In the normalized form the exponent value E is a biased/normalized value. The normalized exponent, E, equals the sum of the actual exponent value and the exponent bias. In the normalized form, Y-1 bits are used to store the fraction value. The F0 fraction bit is always a hidden bit and its value is assumed to be 1.

S represents the value of the sign of the number. If S is 0 then the value is a positive floating-point number; otherwise it is negative. The X bits that follow are used to store the normalized exponent value E and the last Y-1 bits are used to store the fraction/mantissa value in the normalized form.

For the given exponent width, the exponent bias is calculated using the following equation:

$$\text{Exponent_bias} = 2^{(X - 1)} - 1$$

Where X is the exponent bit width.

According to the IEEE standard, a single precision floating-point data is represented using 32 bits. The normalized exponent and fraction/mantissa are allocated 8 and 24 bits, respectively. The exponent bias for single precision is 127. Similarly, a double precision floating-point data is represented using a total of 64 bits where the exponent bit width is 11 and the fraction bit width is 53. The exponent bias value for double precision is 1023.

The normalized floating-point number in the equation form is represented as follows:

$$\text{Normalized Floating-Point Value} = (-1)^S \text{ x } F0.F1F2\ .\ FY\text{-}2FY\text{-}1 \text{ x } (2)^E$$

The actual value of exponent (E_actual) = E - Exponent_bias. Considering 1 as the value for the hidden bit F0 and the E_actual value, a floating-point number can be calculated as follows:

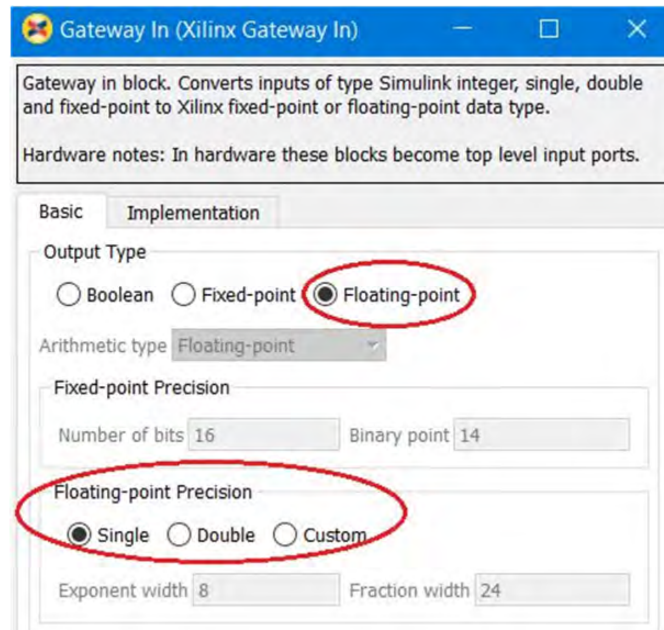
$$\text{FP_Value} = (-1)^S \text{ x } 1.F1F2\ .\ FY\text{-}2FY\text{-}1 \text{ x } (2)^{(E_actual)}$$

Floating-Point Data Representation in System Generator

The System Generator Gateway In block previously only supported the Boolean and Fixed-point data types. As shown below, the Gateway In block GUI and underlying mask parameters now support the Floating-point data type as well. You can select either a **Single**, **Double** or **Custom** precision type after specifying the floating-point data type.

For example, if Exponent width of 9 and Fraction width of 31 is specified then the floating-point data value will be stored in total 40 bits where the MSB bit will be used for sign representation, the following 9 bits will be used to store biased exponent value and the 30 LSB bits will be used to store the fractional value.

Figure 11: **Floating-point Precision**

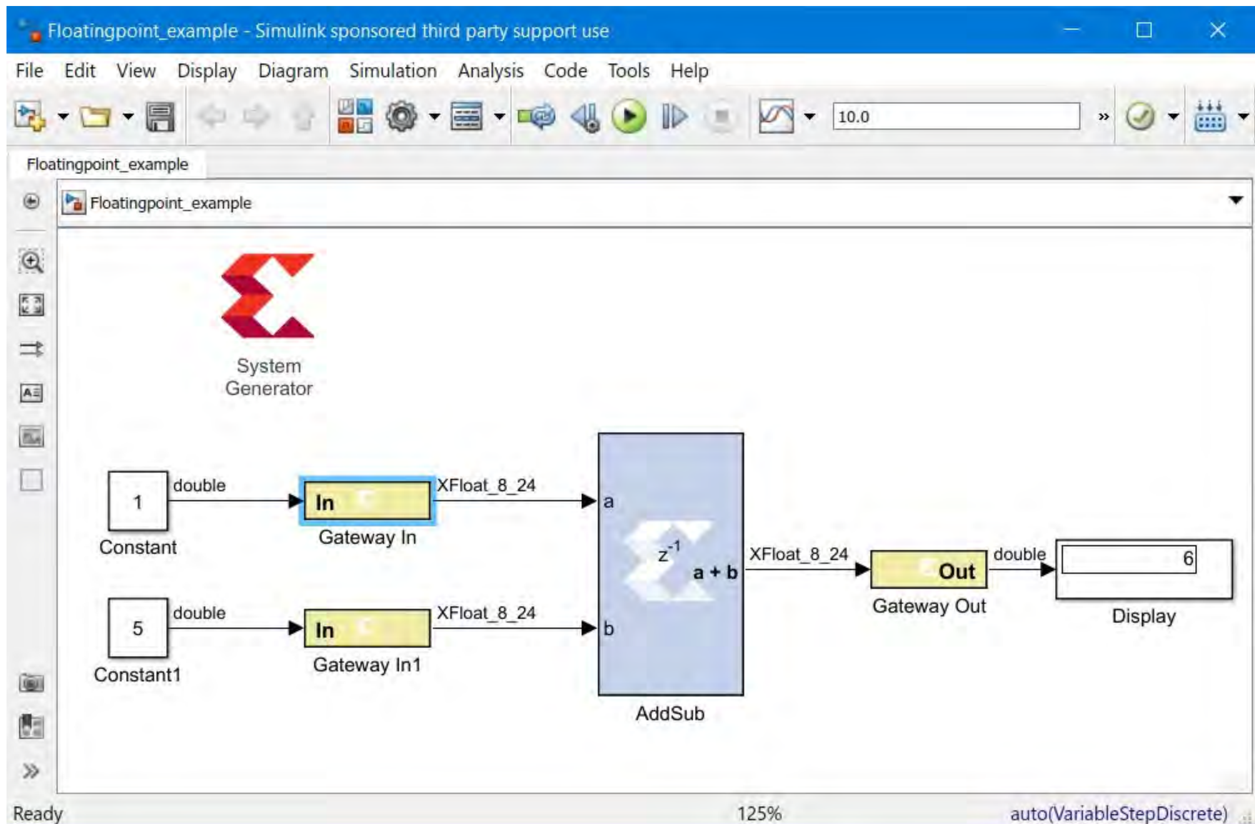


In compliance with the IEEE-754 standard, if **Single** precision is selected then the total bit width is assumed to be 32; 8 bits for the exponent and 24 bits for the fraction. Similarly when **Double** precision is selected, the total bit width is assumed to be 64 bits; 11 bits for the exponent and 53 bits for the fraction part. When **Custom** precision is selected, the **Exponent width** and **Fraction width** fields are activated and you are free to specify values for these fields (8 and 24 are the default values). The total bit width for **Custom** precision data is the summation of the number of exponent bits and the number of fraction bits. Similar to fraction bit width for **Single** precision and **Double** precision data types the fraction bit width for **Custom** precision data type must include the hidden bit F0.

Displaying the Data Type on Output Signals

As shown below, after a successful rate and type propagation, the floating-point data type is displayed on the output of each System Generator block. To display the signal data type as shown in the diagram below, you select the pulldown menu item **Display → Signals & Ports → Port Data Types**.

Figure 12: **Floating-point Data Type**



A floating-point data type is displayed using the format: `XFloat_<exponent_bit_width>_<fraction_bit_width>`. Single and Double precision data types are displayed using the string "`XFloat_8_24`" and "`XFloat_11_53`", respectively.

If for a Custom precision data type the exponent bit width 9 and the fraction bit width 31 are specified, then it will be displayed as "`XFloat_9_31`". A total of 40 bits will be used to store the floating-point data value. Because floating-point data is stored in a normalized form, the fractional value will be stored in 30 bits.

In System Generator the fixed-point data type is displayed using format `XFix_<total_data_width>_<binary_point_width>`. For example, a fixed-point data type with the data width of 40 and binary point width of 31 is displayed as `XFix_40_31`.

It is necessary to point out that in the fixed-point data type the actual number of bits used to store the fractional value is different from that used for floating-point data type. In the example above, all 31 bits are used to store the fractional bits of the fixed-point data type.

System Generator uses the exponent bit width and the fraction bit width to configure and generate an instance of the Floating-Point Operator core.

Rate and Type Propagation

During data rate and type propagation across a System Generator block that supports floating-point data, the following design rules are verified. The appropriate error is issued if one of the following violations is detected.

1. If a signal carrying floating-point data is connected to the port of a System Generator block that doesn't support the floating-point data type.
2. If the data input (both A and B data inputs, where applicable) and the data output of a System Generator block are not of the same floating-point data type. The DRC check will be made between the two inputs of a block as well as between an input and an output of the block.

   If a Custom precision floating-point data type is specified, the exponent bit width and the fraction bit width of the two ports are compared to determine that they are of the same data type.

   ***Note*:** The Convert and Relational blocks are excluded from this check. The Convert block supports Float-to-float data type conversion between two different floating-point data types. The Relational block output is always the Boolean data type because it gives a true or false result for a comparison operation.
3. If the data inputs are of the fixed-point data type and the data output is expected to be floating-point and vice versa.

   ***Note*:** The Convert and Relational blocks are excluded from this check. The Convert block supports Fixed-to-float as well as Float-to-fixed data type conversion. The Relational block output is always the Boolean data type because it gives a true or false result for a comparison operation.
4. If Custom precision is selected for the Output Type of blocks that support the floating-point data type. For example, for blocks such as AddSub, Mult, CMult, and MUX, only Full output precision is supported if the data inputs are of the floating-point data type.
5. If the Carry In port or Carry Out port is used for the AddSub block when the operation on a floating-point data type is specified.
6. If the Floating-Point Operator IP core gives an error for DRC rules defined for the IP.

AXI Signal Groups

System Generator blocks found in the AXI4 library contain interfaces that conform to the AXI4 specification. Blocks with AXI4 interfaces are drawn such that ports relating to a particular AXI4 interface are grouped and colored in similarly. This makes it easier to identify data and control signals pertaining to the same interface. Grouping similar AXI4 ports together also make it possible to use the Simulink Bus Creator and Simulink Bus Selector blocks to connect groups of signals together. More information on AXI4 can be found in the section titled AXI Interface. For more detailed information on the AMBA AXI4 specification, please refer to the Xilinx AMBA AXI4 documents found at the AMBA AXI4 Interface Protocol page on the Xilinx website.

Bit-True and Cycle-True Modeling

Simulations in System Generator are *bit-true* and *cycle-true*. To say a simulation is bit-true means that at the boundaries (i.e., interfaces between System Generator blocks and non-System Generator blocks), a value produced in simulation is bit-for-bit identical to the corresponding value produced in hardware. To say a simulation is cycle-true means that at the boundaries, corresponding values are produced at corresponding times. The boundaries of the design are the points at which System Generator gateway blocks exist. When a design is translated into hardware, Gateway In (respectively, Gateway Out) blocks become top-level input (resp., output) ports.

Timing and Clocking

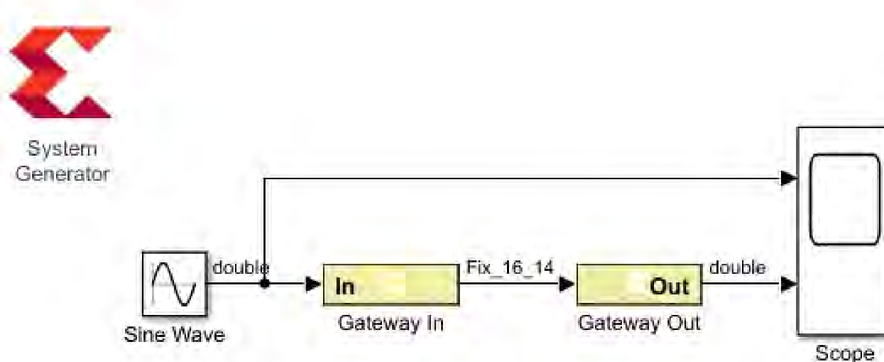
Discrete Time Systems

Designs in System Generator are discrete time systems. In other words, the signals and the blocks that produce them have associated sample rates. A block's sample rate determines how often the block is awoken (allowing its state to be updated). System Generator sets most sample rates automatically. A few blocks, however, set sample rates explicitly or implicitly.

***Note*:** For an in-depth explanation of Simulink discrete time systems and sample times, consult the Using Simulink reference manual from the MathWorks, Inc.

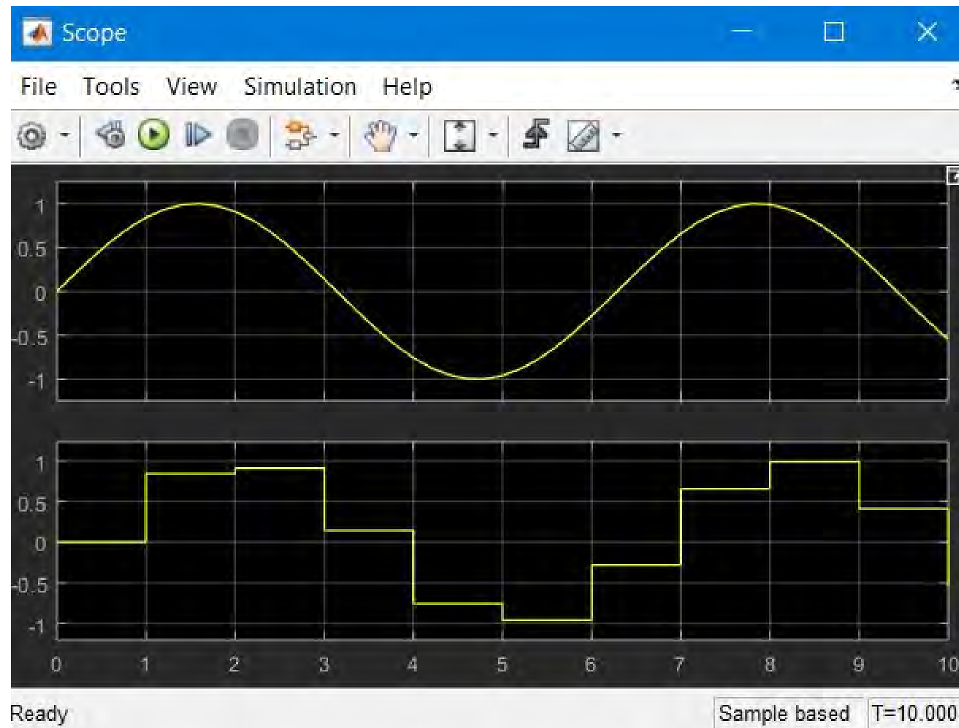
A simple System Generator model illustrates the behavior of discrete time systems. Consider the model shown below. It contains a gateway that is driven by a Simulink source (Sine Wave), and a second gateway that drives a Simulink sink (Scope).

Figure 13: **Discrete Time System**



The Gateway In block is configured with a sample period of one second. The Gateway Out block converts the Xilinx fixed-point signal back to a double (so it can analyzed in the Simulink scope), but does not alter sample rates. The scope output below shows the unaltered and sampled versions of the sine wave.

Figure 14: **Scope Output**



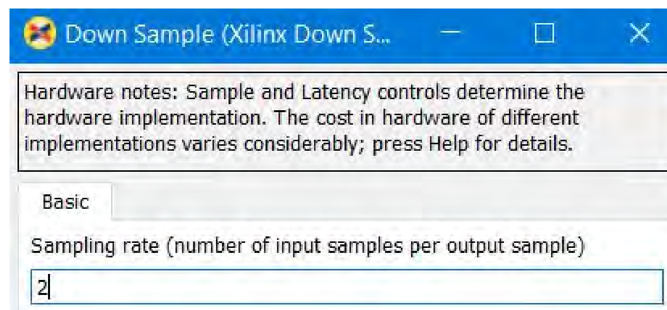
Multirate Models

System Generator supports *multirate* designs, i.e., designs having signals running at several sample rates. System Generator automatically compiles multirate models into hardware. This allows multirate designs to be implemented in a way that is both natural and straightforward in Simulink.

Rate-Changing Blocks

System Generator includes blocks that change sample rates. The most basic rate changers are the Up Sample and Down Sample blocks. As shown in the figure below, these blocks explicitly change the rate of a signal by a fixed multiple that is specified in the block's dialog box.

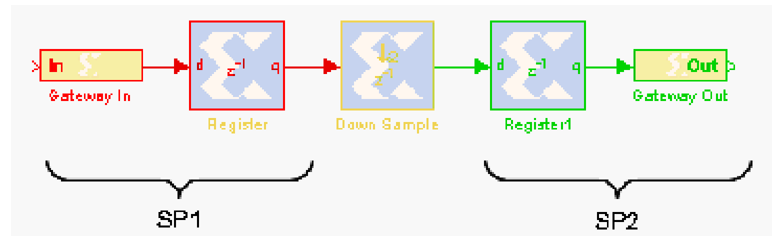
Figure 15: **Rate Change Dialog**



Other blocks (e.g., the Parallel To Serial and Serial To Parallel converters) change rates implicitly in a way determined by block parameterization.

Consider the simple multirate example below. This model has two sample periods, SP1 and SP2. The Gateway In dialog box defines the sample period SP1. The Down Sample block causes a rate change in the model, creating a new rate SP2 which is half as fast as SP1.

Figure 16: **Multirate Example**



Hardware Oversampling

Some System Generator blocks are oversampled, i.e., their internal processing is done at a rate that is faster than their data rates. In hardware, this means that the block requires more than one clock cycle to process a data sample. In Simulink such blocks do not have an observable effect on sample rates.

Although blocks that are oversampled do not cause an explicit sample rate change in Simulink, System Generator considers the internal block rate along with all other sample rates when generating clocking logic for the hardware implementation. This means that you must consider the internal processing rates of oversampled blocks when you specify the **Simulink system period** value in the System Generator token dialog box.

Asynchronous Clocking

System Generator focuses on the design of hardware that is synchronous to a single clock. It can, under some circumstances, be used to design systems that contain more than one clock. This is possible provided the design can be partitioned into individual clock domains with the exchange of information between domains being regulated by dual port memories and FIFOs. The remainder of this topic focuses exclusively on the clock-synchronous aspects of System Generator. This discussion is relevant to both single-clock and multiple-clock designs.

Synchronous Clocking

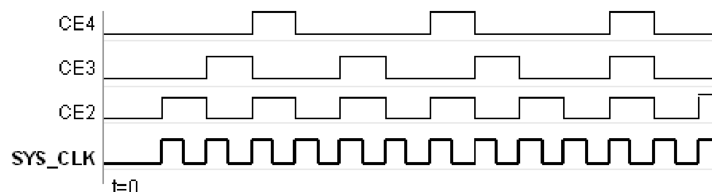
By default, System Generator creates designs with synchronous clocking, where multiple rates are realized using clock enables. When System Generator compiles a model into hardware, System Generator preserves the sample rate information of the design in such a way that corresponding portions in hardware run at appropriate rates. In hardware, System Generator generates related rates by using a single clock in conjunction with clock enables, one enable per rate. The period of each clock enable is an integer multiple of the period of the system clock.

Inside Simulink, neither clocks nor clock enables are required as explicit signals in a System Generator design. When System Generator compiles a design into hardware, it uses the sample rates in the design to deduce what clock enables are needed. To do this, it employs two user-specified values from the System Generator token: the **Simulink system period** and **FPGA clock period**. These numbers define the scaling factor between time in a Simulink simulation, and time in the actual hardware implementation. The Simulink system period must be the greatest common divisor (gcd) of the sample periods that appear in the model, and the FPGA clock period is the period, in nanoseconds, of the system clock. If p represents the Simulink system period, and c represents the FPGA system clock period, then something that takes kp units of time in Simulink takes k ticks of the system clock (hence kc nanoseconds) in hardware.

To illustrate this point, consider a model that has three Simulink sample periods 2, 3, and 4. The gcd of these sample periods is 1, and should be specified as such in the **Simulink system period** field for the model. Assume the **FPGA clock period** is specified to be 10ns. With this information, the corresponding clock enable periods can be determined in hardware.

In hardware, we refer to the clock enables corresponding to the Simulink sample periods 2, 3, and 4 as CE2, CE3, and CE4, respectively. The relationship of each clock enable period to the system clock period can be determined by dividing the corresponding Simulink sample period by the Simulink System Period value. Thus, the periods for CE2, CE3, and CE4 equal 2, 3, and 4 system clock periods, respectively. A timing diagram for the example clock enable signals is shown below:

Figure 17: **Timing Diagram**



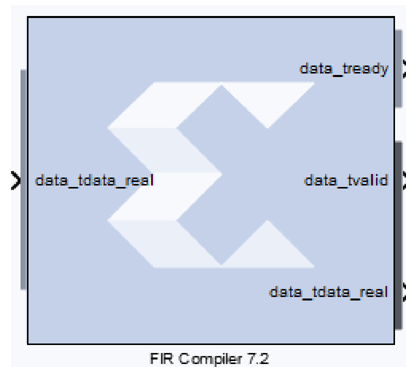
Synchronization Mechanisms

System Generator does not make implicit synchronization mechanisms available. Instead, synchronization is the responsibility of the designer, and must be done explicitly.

Valid Ports

System Generator provides several blocks (in particular, the AXI FIFO) that can be used for synchronization. Several blocks provide optional AXI signaling interfaces to denote when a sample is valid (TValid) and when the interface is ready for data (TReady). Note that the tvalid / tready ports may not be visible based on the configuration of the IP. Color association denotes a collection of ports for each interface on the block as shown below. Blocks with interfaces can be chained, affording a primitive form of flow control. Examples of such blocks with AXI interfaces include the FFT, FIR, and DDS.

Figure 18: **Block with AXI Interface**



Indeterminate Data

Indeterminate values are common in many hardware simulation environments. Often they are called "don't cares" or "Xs". In particular, values in System Generator simulations can be indeterminate. A dual port memory block, for example, can produce indeterminate results if both ports of the memory attempt to write the same address simultaneously. What actually happens in hardware depends upon effectively random implementation details that determine which port sees the clock edge first. Allowing values to become indeterminate gives the system designer greater flexibility. Continuing the example, there is nothing wrong with writing to memory in an indeterminate fashion if subsequent processing does not rely on the indeterminate result.

HDL modules that are brought into the simulation through HDL co-simulation are a common source for indeterminate data samples. System Generator presents indeterminate values to the inputs of an HDL co-simulating module as the standard logic vector 'XXX . . . XX'.

Indeterminate values that drive a Gateway Out become what are called NaNs. (NaN abbreviates "not a number".) In a Simulink scope, NaN values are not plotted. Conversely, NaNs that drive a Gateway In become indeterminate values. System Generator provides an Indeterminate Probe block that allows for the detection of indeterminate values. This probe cannot be translated into hardware.

In System Generator, any arithmetic signal can be indeterminate, but Boolean signals cannot be. If a simulation reaches a condition that would force a Boolean to become indeterminate, the simulation is halted and an error is reported. Many Xilinx blocks have control ports that only allow Boolean signals as inputs. The rule concerning indeterminate Booleans means that such blocks never see an indeterminate on a control port

A UFix_1_0 is a type that is equivalent to Boolean except for the above restriction concerning indeterminate data.

Block Masks and Parameter Passing

The same scoping and parameter passing rules that apply to ordinary Simulink blocks apply to System Generator blocks. Consequently, blocks in the Xilinx Blockset can be parameterized using MATLAB variables and expressions. This capability makes possible highly parametric designs that take advantage of the expressive and computational power of the MATLAB language.

Block Masks

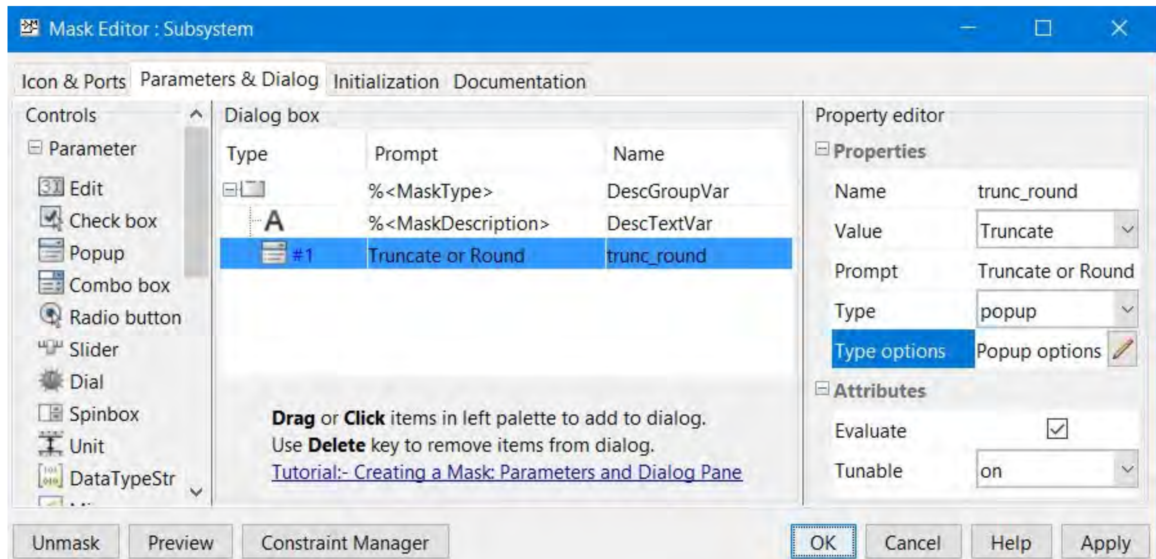
In Simulink, blocks are parameterized through a mechanism called *masking*. In essence, a block can be assigned *mask variables* whose values can be specified by a user through dialog box prompts or can be calculated in mask initialization commands. Variables are stored in a *mask workspace*. A mask workspace is local to the blocks under the mask and cannot be accessed by external blocks.

***Note*:** It is possible for a mask to access global variables and variables in the base workspace. To access a base workspace variable, use the MATLAB evalin function. For more information on the MATLAB and Simulink scoping rules, refer to the manuals titled *Using MATLAB* and *Using Simulink* from The MathWorks, Inc.

Parameter Passing

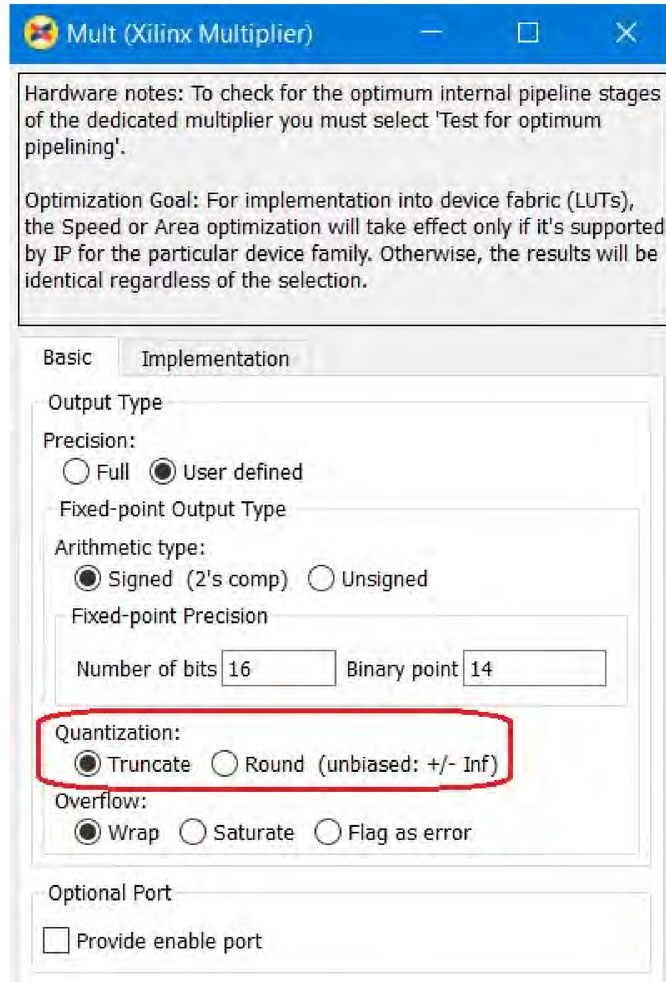
It is often desirable to pass variables to blocks inside a masked Subsystem. Doing so allows the block's configuration to be determined by parameters on the enclosing Subsystem. This technique can be applied to parameters on blocks in the Xilinx blockset whose values are set using a listbox, radio button, or checkbox. For example, when building a Subsystem that consists of a multiply and accumulate block, you can create a parameter on the Subsystem that allows you to specify whether to truncate or round the result. This parameter will be called trunc_round as shown in the figure below.

Figure 19: **Creating a Parameter**



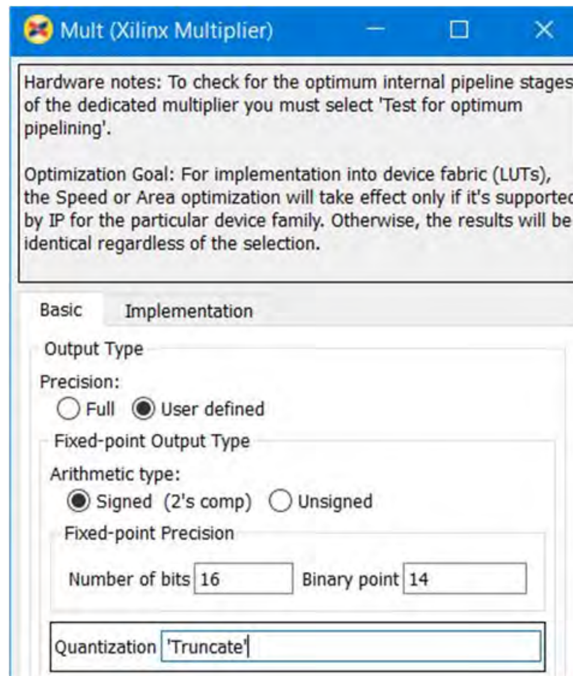
As shown below, in the parameter editing dialog for the accumulator and multiplier blocks, there are radio buttons that allow either the truncate or round option to be selected.

Figure 20: **Editing a Parameter**



In order to use a parameter rather than the radio button selection, right-click the radio button and select **Define With Expression**. A MATLAB expression can then be used as the parameter setting. In the example below, the trunc_round parameter from the Subsystem mask can be used in both the accumulator and multiply blocks so that each block will use the same setting from the mask variable on the Subsystem.

Figure 21: **Using a Parameter**



Automatic Code Generation

System Generator automatically compiles designs into low-level representations. The ways in which System Generator compiles a model can vary, and depend on settings in the System Generator token. In addition to producing HDL descriptions of hardware, the tool generates auxiliary files. Some files (e.g., project files, constraints files) assist downstream tools, while others (e.g., VHDL test bench) are used for design verification.

Compiling and Simulating Using the System Generator Token	Describes how to use the System Generator token to compile designs into equivalent low-level HDL.
Compilation Results	Describes the low-level files System Generator produces when **HDL Netlist** is selected on the System Generator token and **Generate** is pushed.
Vivado Project	Describes the example project System Generator produces when **HDL Netlist** or **IP Catalog** is selected on the System Generator token and **Generate** is pushed.
HDL Testbench	Describes the VHDL test bench that System Generator can produce.

Compiling and Simulating Using the System Generator Token

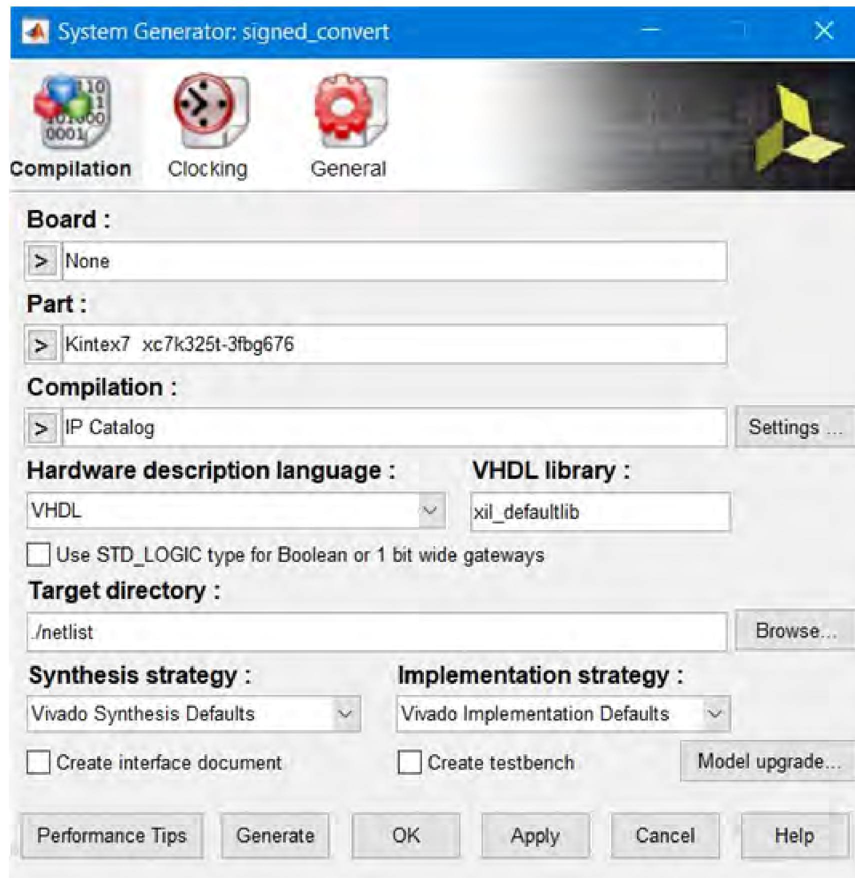
System Generator automatically compiles designs into low-level representations. Designs are compiled and simulated using the System Generator token. This topic describes how to use the block.

The System Generator token is a member of the Xilinx Blockset's Basic Elements and Tools libraries. As with all Xilinx blocks, the System Generator token can also be found in the Index library.

A design must contain at least one System Generator token, but can contain several System Generator tokens on different levels (one per level). A System Generator token that is underneath another in the hierarchy is a *slave*; one that is not a slave is a *master*. The scope of a System Generator token consists of the level of hierarchy into which it is embedded and all Subsystems below that level. Certain parameters (e.g., **Simulink System Period**) can be specified only in a master.

Once a System Generator token is added, it is possible to specify how code generation and synthesis should be handled. The System Generator Token dialog box is shown below:

Figure 22: **System Generator Token Dialog Box**



Compilation Type and the Generate Button

Pressing the **Generate** button instructs System Generator to compile a portion of the design into equivalent low-level results. The portion that is compiled is the sub-tree whose root is the Subsystem containing the block. (To compile the entire design, use a System Generator token placed at the top of the design.) The compilation type (under **Compilation**) specifies the type of result that should be produced. The possible types are:

- **IP Catalog**: Packages the design as an IP core that can be added to the Vivado IP catalog for use in another design.
- **Hardware Co-Simulation (JTAG/Point-to-point Ethernet)**: Generates HW co-simulation library block to verify an algorithm in the hardware.
- **Synthesized Checkpoint**: Creates a design checkpoint file (`synth_1.dcp`) that can then be used in any Vivado IDE project.
- **HDL Netlist**: Generates VHDL or Verilog RTL designs.

Table 1: **System Generator Token Dialog Box Controls**

Control	Description
Board	Specifies a Xilinx, Partner, or Custom board you will use to test your design. For a Partner board or a custom board to appear in the Board list, you must configure System Generator to access the board files that describe the board. Board awareness in System Generator is detailed in Specifying Board Support in System Generator. When you select a **Board**, the **Part** field displays the name of the Xilinx device on the selected **Board**, and this part name cannot be changed.
Part	Defines the Xilinx part to be used. If you have selected a **Board**, the **Part** field will display the name of the Xilinx device on the selected **Board**, and this part name cannot be changed.
Hardware description language	Specifies the language to be used for HDL netlist of the design. The possibilities are VHDL and Verilog.
VHDL library	Specifies the name of VHDL work library for code generation. The default name is xil_defaultlib.
Use STD_LOGIC type for Boolean or 1 bit wide gateways	If your design's Hardware Description Language (HDL) is VHDL, selecting this option will declare a Boolean or 1-bit port (Gateway In or Gateway Out) as a STD-LOGIC type. If this option is not selected, System Generator will interpret Boolean or 1-bit ports as vectors.
Target Directory	Defines where System Generator should write compilation results. Because System Generator and the FPGA physical design tools typically create many files, it is best to create a separate target directory, i.e., a directory other than the directory containing your Simulink model files. The directory can be an absolute path (e.g. `c:\netlist`) or a path relative to the directory containing the model (e.g. `netlist`).
Synthesis strategy	Choose a Synthesis strategy from the pre-defined strategies in the drop-down list.
Implementation strategy	Choose an Implementation strategy from the pre-defined strategies in the drop-down list.
Create interface document	When this box is checked and the Generate button is activated for netlisting, System Generator creates an HTM document that describes the design being netlisted. This document is placed in the netlist folder.
Create testbench	This instructs System Generator to create an HDL test bench. Simulating the test bench in an HDL simulator compares Simulink simulation results with ones obtained from the compiled version of the design. To construct test vectors, System Generator simulates the design in Simulink, and saves the values seen at gateways. The top HDL file for the test bench is named `<name>_tb.vhd/.v`, where `<name>` is a name derived from the portion of the design being tested and the extension is dependent on the hardware description language.
FPGA clock period	Defines the period in nanoseconds of the system clock. The value need not be an integer. The period is passed to the Xilinx implementation tools through a constraints file, where it is used as the global PERIOD constraint. Multi-cycle paths are constrained to integer multiples of this value.

Table 1: **System Generator Token Dialog Box Controls** *(cont'd)*

Control	Description
Clock pin location	Defines the pin location for the hardware clock. This information is passed to the Xilinx implementation tools through a constraints file.

Simulink System Period

You must specify a value for **Simulink system period** in the System Generator token dialog box. This value tells the underlying rate, in seconds, at which simulations of the design should run. The period must evenly divide all sample periods in the design. For example, if the design consists of blocks whose sample periods are 2, 6, and 8, then the largest acceptable sample period is 2, though other values such as 1 and 0.5 are also acceptable. Sample periods arise in three ways: some are specified explicitly, some are calculated automatically, and some arise implicitly within blocks that involve internal rate changes. For more information on how the system period setting affects the hardware clock, refer to Timing and Clocking.

Before running a simulation or compiling the design, System Generator verifies that the period evenly divides every sample period in the design.

It is possible to assemble a System Generator model that is inconsistent because its periods cannot be reconciled. (For example, certain blocks require that they run at the system rate. Driving an up-sampler with such a block produces an inconsistent model.) If, even after updating the system period, System Generator reports there are conflicts, then the model is inconsistent and must be corrected.

The period control is hierarchical. See Hierarchical Controls for details.

Block Icon Display

The options on this control affect the display of the block icons on the model. After compilation (which occurs when generating, simulating, or by pressing **Ctrl-D**) of the model various information about the block in your model can be displayed, depending on which option is chosen.

- **Default:** Basic information about port directions are shown.
- **Sample rates:** The sample rates of each port are shown like Normalized samples periods and Sample frequencies (MHz).
- **Pipeline stages:** The number of pipeline stages are shown.
- **HDL port names:** the names of the ports are shown
- **Input data types:** The input data types for each port are shown.
- **Output data types:** Output data types for each port are shown.

Hierarchical Controls

The Simulink System Period control (see the topic Simulink System Period above) on the System Generator token is hierarchical. A hierarchical control on a System Generator token applies to the portion of the design within the scope of the token, but can be overridden on other System Generator tokens deeper in the design. For example, suppose Simulink System Period is set in a System Generator token at the top of the design, but is changed in a System Generator token within a Subsystem S. Then that Subsystem will have the second period, but the rest of the design will use the period set in the top level.

Compilation Results

This topic discusses the low-level files System Generator produces when **HDL Netlist** is selected on the System Generator token and **Generate** is clicked. The files consist of HDL that implements the design. In addition, System Generator organizes the HDL files, and other hardware files into a Vivado® IDE Project. All files are written to the target directory specified on theSystem Generator token. If no test bench is requested, then the key files produced by System Generator are the following:

Table 2: **Compilation Files**

File Name or Type	Description
`<design_name>.vhd/.v`	This file contains a hierarchical structural netlist along with clock/clock enable controls
`<design_name_entity_declarations>.vhd/.v`	This file contains the entity of module definitions of System Generator blocks in the design.
`<design_name>.xpr`	This file is the Vivado IDE project file that describes all of the attributes of the Vivado IDE design.

If a test bench is requested, then, in addition to the above, System Generator produces files that allow simulation results to be compared. The comparisons are between Simulink® simulation results and corresponding results from ModelSim, or any other RTL simulator supported by Vivado® IDE such as Vivado simulator, Model, or VCS. The additional files are as follows:

Table 3: **Additional Compilation Files**

File Name or Type	Description
Various .dat files	These contain the simulation results from Simulink.
`<design_name>_tb.vhd/.v`	This is a test bench that wraps the design. When simulated, this test bench compares simulation results from the digital simulator against those produced by Simulink.

Using the System Generator Constraints File

When a design is compiled during code generation, System Generator produces *constraints* that tell downstream tools how to process the design. This enables the tools to produce a higher quality implementation, and to do so using considerably less time. Constraints supply the following:

- The period to be used for the system clock.
- The speed, with respect to the system clock, at which various portions of the design must run.
- The pin locations at which ports should be placed.
- The speed at which ports must operate.

The system clock period (i.e., the period of the fastest hardware clock in the design) can be specified in the System Generator token. System Generator writes this period to the constraints file. Downstream tools use the period as a goal when implementing the design.

Multicycle Path Constraints

Many designs consist of parts that run at different clock rates. For the fastest part, the system clock period is used. For the remaining parts, the clock period is an integer multiple of the system clock period. It is important that downstream tools know what speed each part of the design must achieve. With this information, efficiency and effectiveness of the tools are greatly increased, resulting in reduced compilation times and improved hardware realizations. The division of the design into parts, and the speed at which each part must run, are specified in the constraints file using multicycle path constraints.

IOB Timing and Placement Constraints

When translated into hardware, System Generator's Gateway In and Gateway Out blocks become input and output ports. The locations of these ports and the speeds at which they must operate can be entered in the Gateway In and Out parameter dialog boxes. Port location and speed are specified in the constraints file by IOB timing.

This topic describes how System Generator handles hardware clocks in the HDL it generates. Assume the design is named `<design>`, and `<design>` is an acceptable HDL identifier. When System Generator compiles the design, it writes a collection of HDL entities or modules, the topmost of which is named `<design>`, and is stored in a file named `<design>.vhd/.v`.

The "Clock Enables" Multirate Implementation

Clock and clock enables appear in pairs throughout the HDL. Typical clock names are *clk_1*, *clk_2*, and *clk_3*, and the names of the companion clock enables are *ce_1*, *ce_2*, and *ce_3* respectively. The name tells the rate for the clock/clock enable pair; logic driven by *clk_1* and *ce_1* runs at the system (i.e., fastest) rate, while logic driven by (say) *clk_2* and *ce_2* runs at half the system rate. Clocks and clock enables are not driven in the entity or module named <design> or any subsidiary entities; instead, they are exposed as top-level input ports

The names of the clocks and clock enables in System Generator HDL suggest that clocking is completely general, but this is not the case. To illustrate this, assume a design has clocks named clk_1 and clk_2, and companion clock enables named *ce_1* and *ce_2* respectively. You might expect that working hardware could be produced if the *ce_1* and *ce_2* signals were tied high, and *clk_2* were driven by a clock signal whose rate is half that of *clk_1*. For most System Generator designs this does not work. Instead, *clk_1* and *clk_2* must be driven by the same clock, *ce_1* must be tied high, and *ce_2* must vary at a rate half that of *clk_1* and *clk_2*.

IP Instance Caching

For compilation targets that perform Vivado synthesis to generate their output products, System Generator incorporates a disk cache to speed up the iterative design process.

With the cache enabled for your design, whenever your compilation generates an IP instance for synthesis, and the Vivado synthesis tool creates synthesis output products, the tools create an entry in the cache area.

After the cache is populated, when a new customization of the IP is created which has the exact same properties, the IP is not synthesized again; instead, the cache is referenced and the corresponding synthesis output in the cache is copied to your design's output directory. Because the IP instance is not synthesized again, and this process is repeated for every IP referenced in your design, generation of the output products is completed more quickly.

The following compilation targets invoke Vivado synthesis; these compilation targets will access the cache to synthesize IP in your design.

- Hardware Co-Simulation
- Synthesized Checkpoint

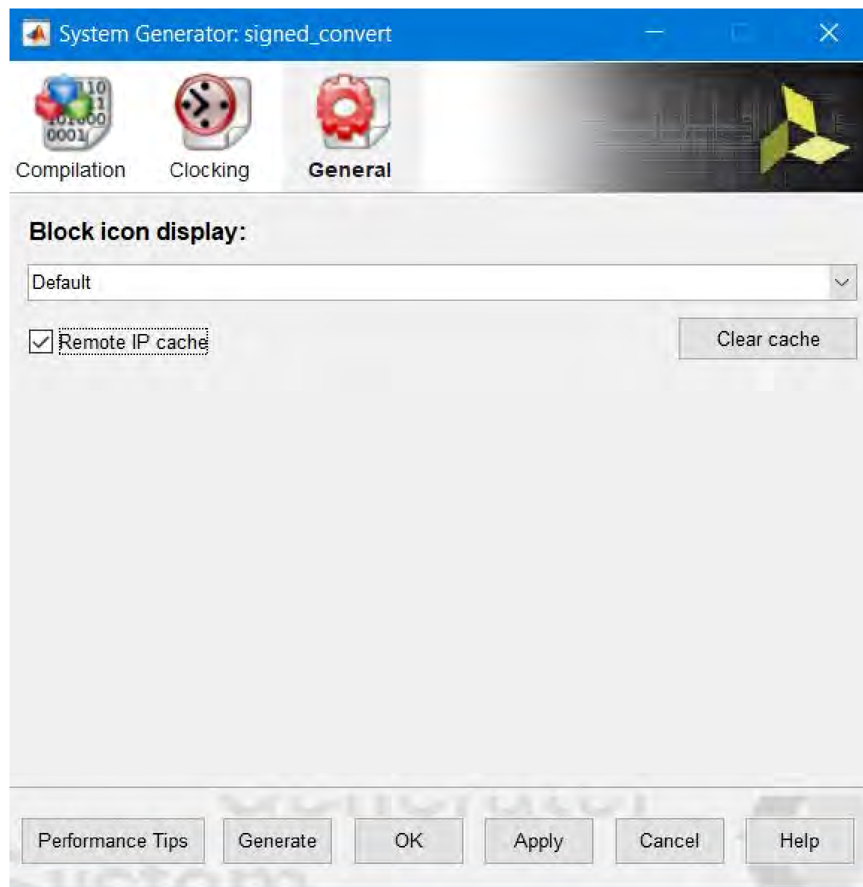
Also, when you compile your design and **Perform analysis** is selected for either **Timing** or **Resource** analysis, Vivado synthesis always runs, regardless of the compilation target. Since timing analysis or resource analysis may be performed several times for a design, enabling IP caching will improve overall performance. For a description of the **Perform analysis** compilation option, see Performing Timing Analysis or Performing Resource Analysis.

The IP cache is shared across multiple Simulink models on your system. If you reuse an IP in one design by including it in another design, and the IP is customized identically and has the same part and language settings in both Simulink models, you can gain the benefit of caching when you compile either of the designs.

You can enable IP caching for your design by selecting **Remote IP cache** in the System Generator token dialog box. The cache will then be referenced for every compilation performed afterwards.

CAUTION! *The IP Cache can grow large, depending on the number of IP present in your design.*

Figure 23: **Block Icon Display**



You can clear the cache to save disk space by clicking **Clear cache** in the System Generator token dialog box.

To find the location of the IP cache directory on your system, enter the command `xilinx.environment.getipcachepath` on the MATLAB command line. The full path to the IP cache directory will display in the MATLAB command window.

```
>> xilinx.environment.getipcachepath
ans =
C:/Users/your_id/AppData/Local/Xilinx/Sysgen/SysgenVivado/win64.o/ip
```

IP caching in System Generator is similar to IP caching in the Vivado Design Suite, described at this link in the *Vivado Design Suite User Guide: Designing with IP* (UG896). However, the IP cache for System Generator designs is in a different location than the IP cache for Vivado projects.

Vivado Project

The **HDL Netlist** and **IP Catalog** compilation targets also generate an example Vivado project, which represents an integration of the results of Code Generation.

In the case of the **HDL Netlist** compilation target, the Vivado project sets the module designed in System Generator as the top level and includes instances of IP. Also, if **Create testbench** is selected in the System Generator token, a test bench and stimulus files (`*.dat`) are also added to the project.

In the case of the **IP Catalog** compilation target, an example project is created with the following features:

- The IP generated from System Generator is already added to the IP catalog associated with the project and available for the RTL flow as well as the IP integrator-based flow.
- The design includes an RTL instantiation of IP called <ip>_0 underneath <design>_stub that indicates how to instanciate such an IP in the RTL flow
- The design includes an RTL test bench called <design>_tb that also instantiates the same IP in the RTL flow.

Note: A test bench is *not* created if **AXI4-Lite** slave interface generation is selected in a Gateway In or Gateway Out block.

- The project also includes an example IP integrator diagram with a Zynq-7000 subsystem if the part selected in this example is a Zynq-7000 SoC part. For all other parts, a MicroBlaze™-based subsystem is created.

HDL Testbench

Ordinarily, System Generator designs are bit and cycle-accurate, so Simulink simulation results exactly match those seen in hardware. There are, however, times when it is useful to compare Simulink simulation results against those obtained from an HDL simulator. In particular, this makes sense when the design contains black boxes. The **Create Testbench** checkbox in the System Generator token makes this possible.

Suppose the design is named `<design>`, and a System Generator token is placed at the top of the design. Suppose also that in the token the **Compilation** field is set to **HDL Netlist**, and the **Create Testbench** checkbox is selected. When the **Generate** button is clicked, System Generator produces the usual files for the design, and in addition writes the following:

- A file named `<design>_tb.vhd/.v` that contains a test bench HDL entity.
- Various `.dat` files that contain test vectors for use in an HDL test bench simulation.

You can perform RTL simulation using the Vivado Integrated Design Environment (IDE). For more details, refer to the document *Vivado Design Suite User Guide: Logic Simulation* (UG900).

System Generator generates the `.dat` files by saving the values that pass through gateways. In the HDL simulation, input values from the `.dat` files are stimuli, and output values are expected results. The test bench is simply a wrapper that feeds the stimuli to the HDL for the design, then compares HDL results against expected ones.

Compiling MATLAB into an FPGA

System Generator provides direct support for MATLAB through the MCode block. The MCode block applies input values to an M-function for evaluation using Xilinx's fixed-point data type. The evaluation is done once for each sample period. The block is capable of keeping internal states with the use of persistent state variables. The input ports of the block are determined by the input arguments of the specified M-function and the output ports of the block are determined by the output arguments of the M-function. The block provides a convenient way to build finite state machines, control logic, and computation heavy systems.

In order to construct an MCode block, an M-function must be written. The M-file must be in the directory of the model file that is to use the M-file or in a directory in the MATLAB path.

The following text provides examples that use the MCode block:

- Example 1 Simple Selector shows how to implement a function that returns the maximum value of its inputs;
- Example 2 Simple Arithmetic Operations shows how to implement simple arithmetic operations;
- Example 3 Complex Multiplier with Latency shows how to build a complex multiplier with latency;
- Example 4 Shift Operations shows how to implement shift operations;
- Example 5 Passing Parameters into the MCode Block shows how to pass parameters into a MCode block;
- Example 6 Optional Input Ports shows how to implement optional input ports on an MCode block;

- Example 7 Finite State Machines shows how to implement a finite state machine;
- Example 8 Parameterizable Accumulator shows how to build a parameterizable accumulator;
- Example 9 FIR Example and System Verification shows how to model FIR blocks and how to do system verification;
- Example 10 RPN Calculator shows how to model a RPN calculator – a stack machine;
- Example 11 Example of disp Function shows how to use disp function to print variable values.

Simple Selector

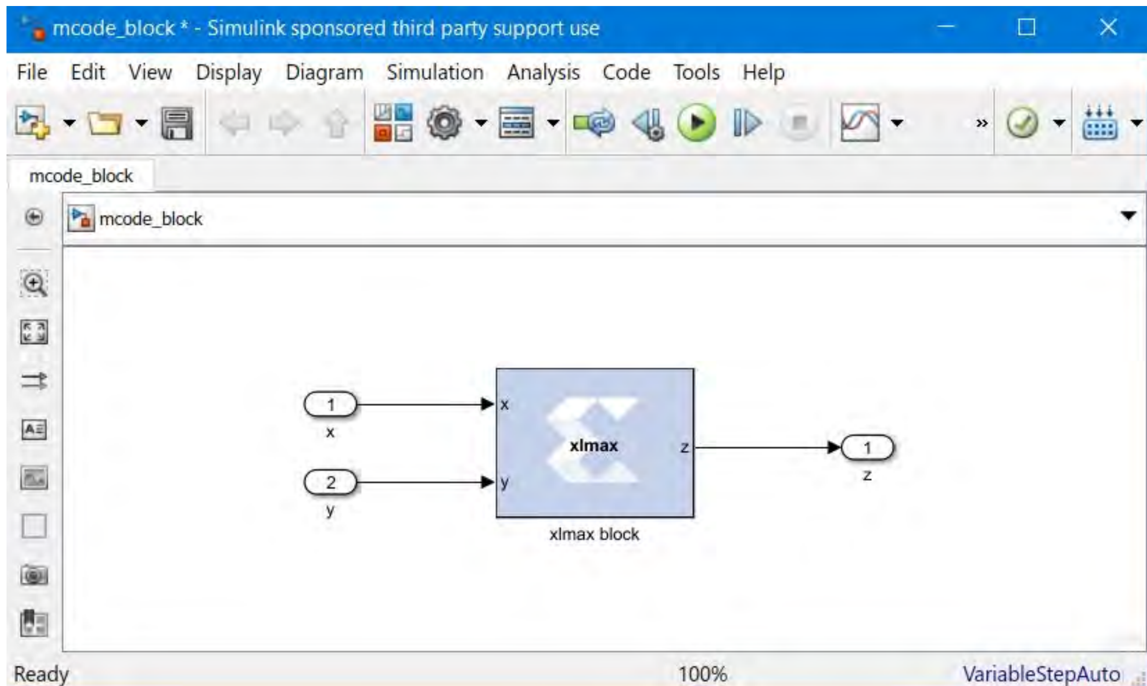
This example is a simple controller for a data path, which assigns the maximum value of two inputs to the output. The M-function is specified as the following and is saved in an M-file `xlmax.m`:

```
function z = xlmax(x, y)
  if x > y
    z = x;
  else
    z = y;
  end
```

The `xlmax.m` file should be either saved in the same directory of the model file or should be in the MATLAB path. Once the `xlmax.m` has been saved to the appropriate place, you should drag a MCode block into your model, open the block parameter dialog box, and enter `xlmax` into the **MATLAB Function** field. After clicking the **OK** button, the block has two input ports x and y, and one output port z.

The following figure shows what the block looks like after the model is compiled. You can see that the block calculates and sets the necessary fixed-point data type to the output port.

Figure 24: **Simple Selector Design**



Simple Arithmetic Operations

This example shows some simple arithmetic operations and type conversions. The following shows the `xlSimpleArith.m` file, which specifies the `xlSimpleArith` M-function.

```
function [z1, z2, z3, z4] = xlSimpleArith(a, b)
  % xlSimpleArith demonstrates some of the arithmetic operations
  % supported by the Xilinx MCode block. The function uses xfix()
  % to create Xilinx fixed-point numbers with appropriate
  % container types.%
  % You must use a xfix() to specify type, number of bits, and
  % binary point position to convert floating point values to
  % Xilinx fixed-point constants or variables.
  % By default, the xfix call uses xlTruncate
  % and xlWrap for quantization and overflow modes.
  % const1 is Ufix_8_3
  const1 = xfix({xlUnsigned, 8, 3}, 1.53);
  % const2 is Fix_10_4
  const2 = xfix({xlSigned, 10, 4, xlRound, xlWrap}, 5.687);
  z1 = a + const1;
  z2 = -b - const2;
  z3 = z1 - z2;
  % convert z3 to Fix_12_8 with saturation for overflow
  z3 = xfix({xlSigned, 12, 8, xlTruncate, xlSaturate}, z3);
  % z4 is true if both inputs are positive
  z4 = a>const1 & b>-1;
```

This M-function uses addition and subtraction operators. The MCode block calculates these operations in full precision, which means the output precision is sufficient to carry out the operation without losing information.

One thing worth discussing is the `xfix` function call. The function requires two arguments: the first for fixed-point data type precision and the second indicating the value. The precision is specified in a cell array. The first element of the precision cell array is the type value. It can be one of three different types: `xlUnsigned`, `xlSigned`, or `xlBoolean`. The second element is the number of bits of the fixed-point number. The third is the binary point position. If the element is `xlBoolean`, there is no need to specify the number of bits and binary point position. The number of bits and binary point position must be specified in pair. The fourth element is the quantization mode and the fifth element is the overflow mode. The quantization mode can be one of `xlTruncate`, `xlRound`, or `xlRoundBanker`. The overflow mode can be one of `xlWrap`, `xlSaturate`, or `xlThrowOverflow`. Quanitization mode and overflow mode must be specified as a pair. If the quantization-overflow mode pair is not specified, the `xfix` function uses `xlTruncate` and `xlWrap` for signed and unsigned numbers. The second argument of the `xfix` function can be either a double or a Xilinx fixed-point number. If a constant is an integer number, there is no need to use the `xfix` function. The Mcode block converts it to the appropriate fixed-point number automatically.

After setting the dialog box parameter **MATLAB function** to `xlSimpleArith`, the block shows two input ports a and b, and four output ports z1, z2, z3, and z4.

Figure 25: **xlSimpleArith MCode Parameter**

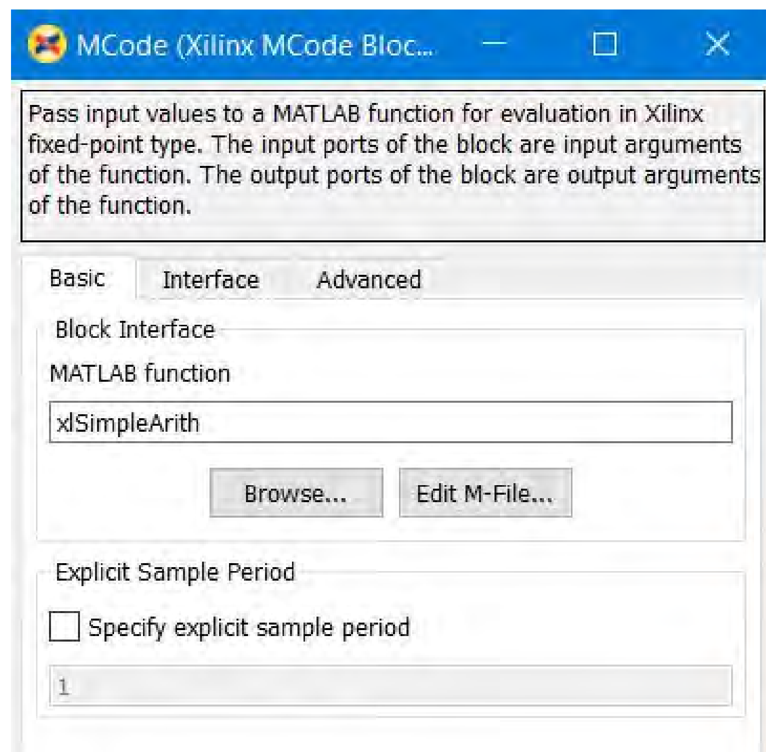
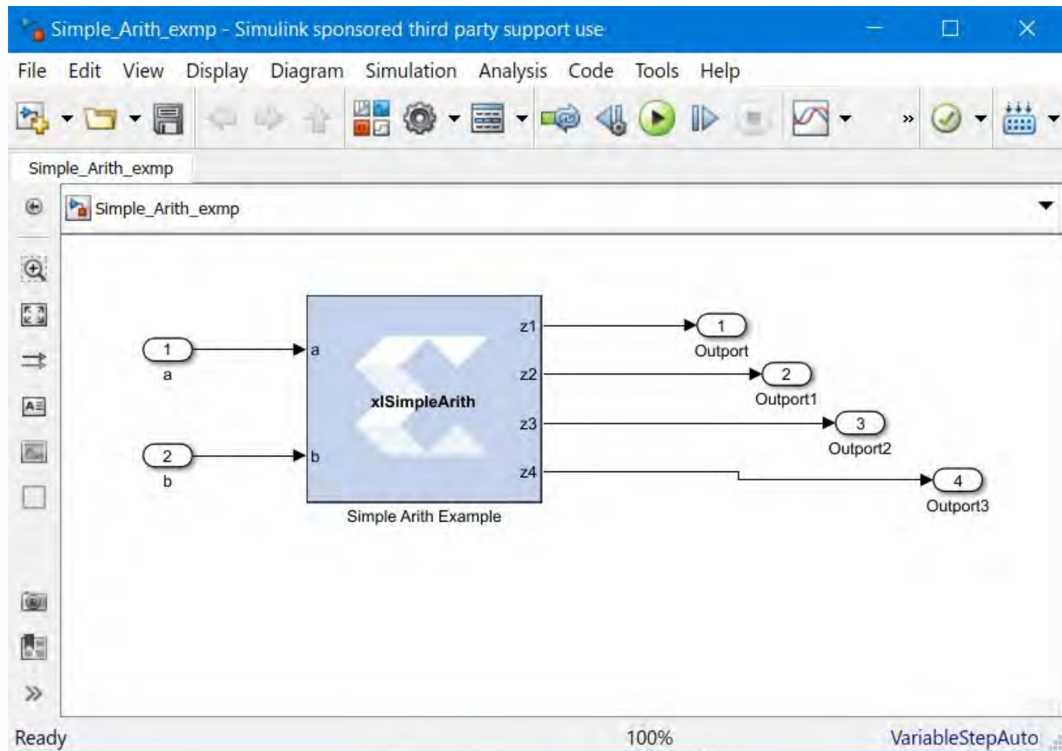


Figure 26: **xlSimpleArith Design**



M-functions using Xilinx data types and functions can be tested in the MATLAB Command Window. For example, if you type: `[z1, z2, z3, z4] = xlSimpleArith(2, 3)` in the MATLAB Command Window, you'll get the following lines:

```
UFix(9, 3): 3.500000
Fix(12, 4): -8.687500
Fix(12, 8): 7.996094
Bool: true
```

Notice that the two integer arguments (2 and 3) are converted to fixed-point numbers automatically. If you have a floating-point number as an argument, an `xfix` call is required.

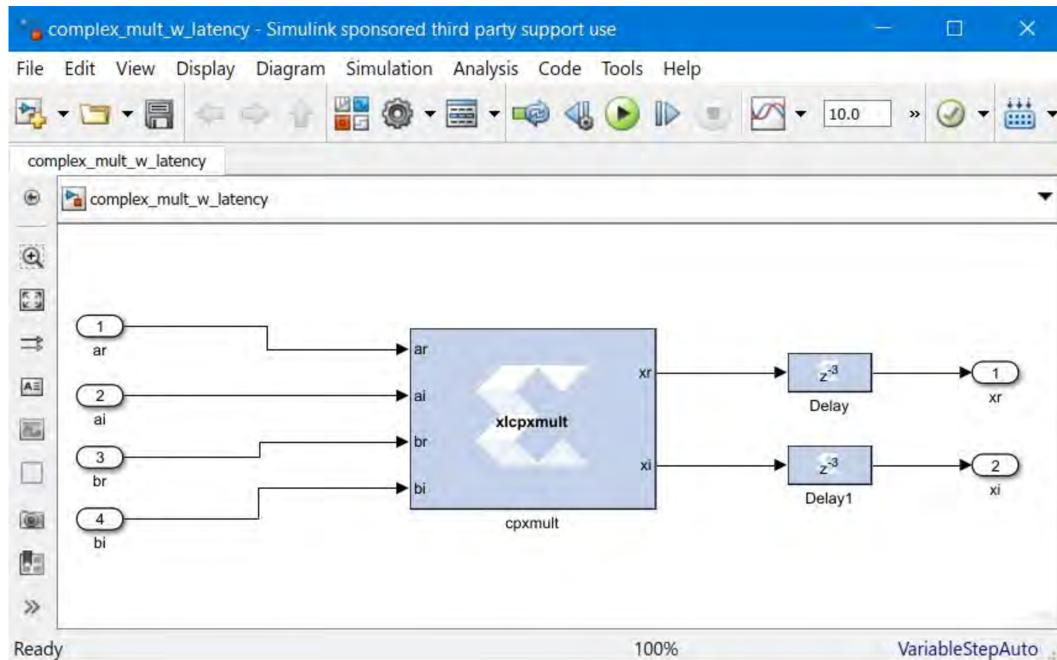
Complex Multiplier with Latency

This example shows how to create a complex number multiplier. The following shows the `xlcpxmult.m` file which specifies the xlcpxmult function.

```
function [xr, xi] = xlcpxmult(ar, ai, br, bi)
  xr = ar * br - ai * bi;
  xi = ar * bi + ai * br;
```

The following diagram shows the sub-system:

Figure 27: **Complex Multiplier Subsystem**



Two delay blocks are added after the MCode block. By selecting the option **Implement using behavioral HDL** on the Delay blocks, the downstream logic synthesis tool is able to perform the appropriate optimizations to achieve higher performance.

Shift Operations

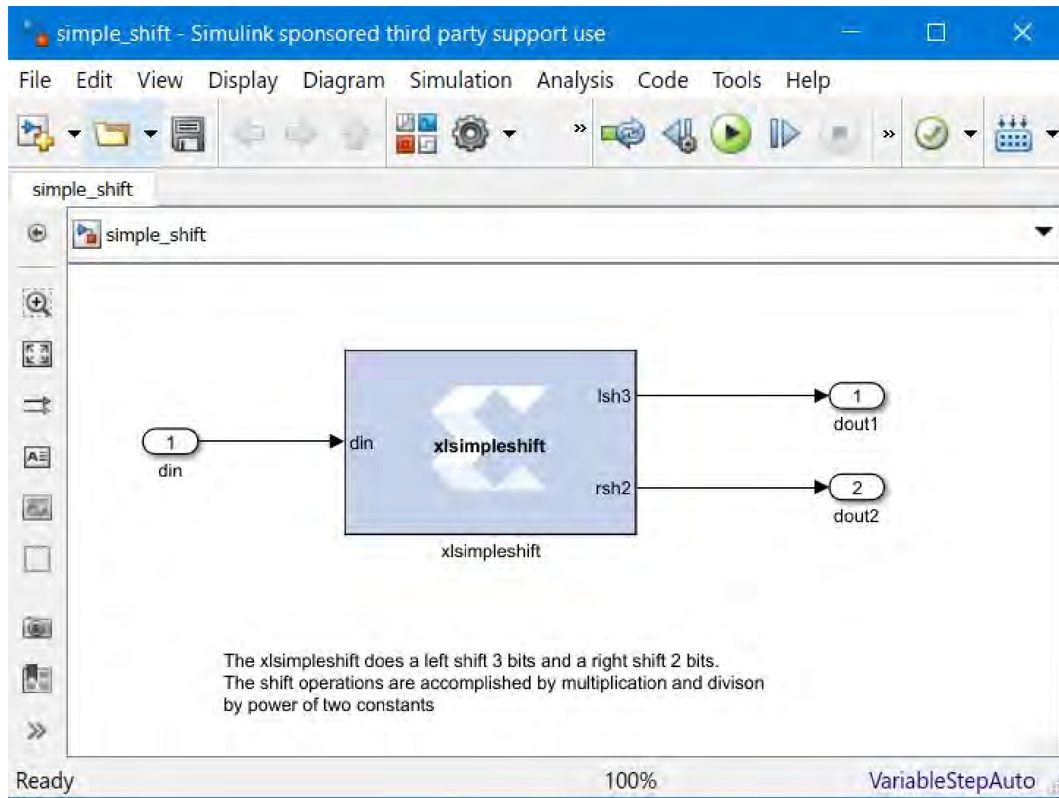
This example shows how to implement bit-shift operations using the MCode block. Shift operations are accomplished with multiplication and division by powers of two. For example, multiplying by 4 is equivalent to a 2-bit left-shift, and dividing by 8 is equivalent to a 3-bit right-shift. Shift operations are implemented by moving the binary point position and if necessary, expanding the bit width. Consequently, multiplying a Fix_8_4 number by 4 results in a Fix_8_2 number, and multiplying a Fix_8_4 number by 64 results in a Fix_10_0 number.

The following shows the `xlsimpleshift.m` file which specifies one left-shift and one right-shift:

```
function [lsh3, rsh2] = xlsimpleshift(din)
  % [lsh3, rsh2] = xlsimpleshift(din) does a left shift
  % 3 bits and a right shift 2 bits.
  % The shift operation is accomplished by
  % multiplication and division of power
  % of two constant.
  lsh3 = din * 8;
  rsh2 = din / 4;
```

The following diagram shows the sub-system after compilation:

Figure 28: **Shift Operations**



Passing Parameters into the MCode Block

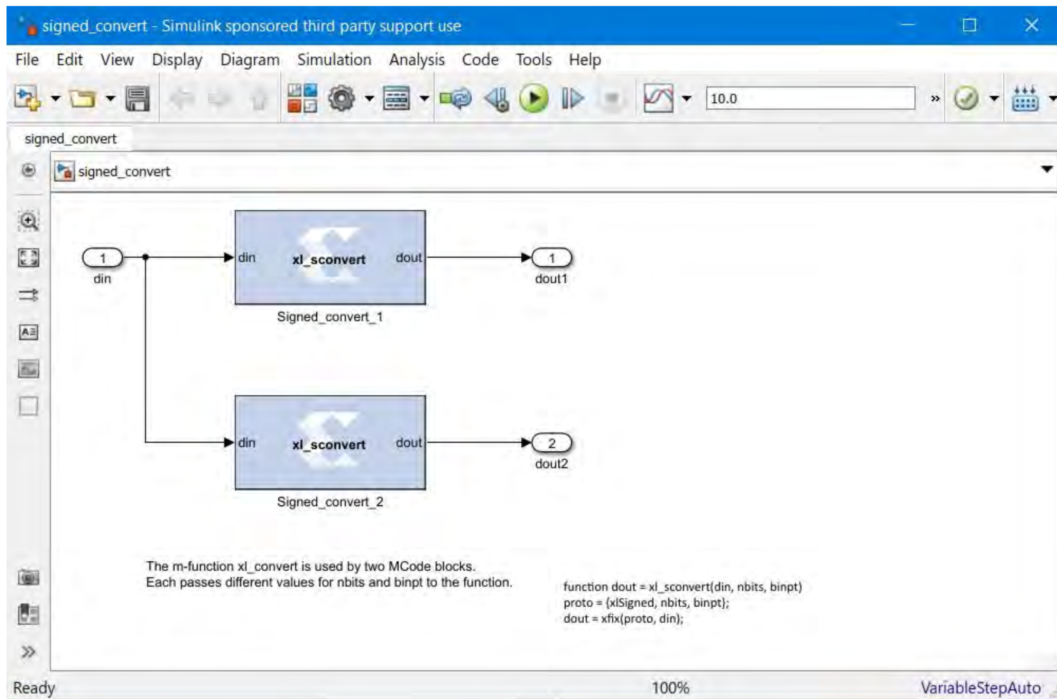
This example shows how to pass parameters into the MCode block. An input argument to an M-function can be interpreted either as an input port on the MCode block, or as a parameter internal to the block.

The following M-code defines an M-function `xl_sconvert` that is contained in file `xl_sconvert.m`:

```
function dout = xl_sconvert(din, nbits, binpt)
  proto = {xlSigned, nbits, binpt};
  dout = xfix(proto, din);
```

The following diagram shows a Subsystem containing two MCode blocks that use M-function `xl_sconvert`. The arguments `nbits` and `binpt` of the M-function are specified differently for each block by passing different parameters to the MCode blocks. The parameters passed to the MCode block labeled `signed convert 1` cause it to convert the input data from type `Fix_16_8` to `Fix_10_5` at its output. The parameters passed to the MCode block labeled `signed convert 2` causes it to convert the input data from type `Fix_16_8` to `Fix_8_4` at its output.

Figure 29: **Subsystem with Two MCode Blocks**



To pass parameters to each MCode block in the diagram above, you can click the **Edit M-File** button on the block GUI then set the values for the M-function arguments. The mask for MCode block `signed convert 1` is shown below:

Figure 30: **Masking MCode Block**

Signed_convert_1 (Xilinx M...

Pass input values to a MATLAB function for evaluation in Xilinx fixed-point type. The input ports of the block are input arguments of the function. The output ports of the block are output arguments of the function.

Basic | Interface | Advanced

Block Interface

MATLAB function

xl_sconvert

Browse... Edit M-File...

Explicit Sample Period

Specify explicit sample period

1

Signed_convert_1 (Xilinx M...

Pass input values to a MATLAB function for evaluation in Xilinx fixed-point type. The input ports of the block are input arguments of the function. The output ports of the block are output arguments of the function.

Basic | Interface | Advanced

Block Interface

Input name	Bind to value
din	
nbits	10
binpt	5

Output name	Suppress output
dout	☐

The above interface window sets the M-function argument `nbits` to be 10 and `binpt` to be 5. The mask for the MCode block `signed convert 2` is shown below:

Figure 31: **Mask for MCode Block Signed Convert 2**

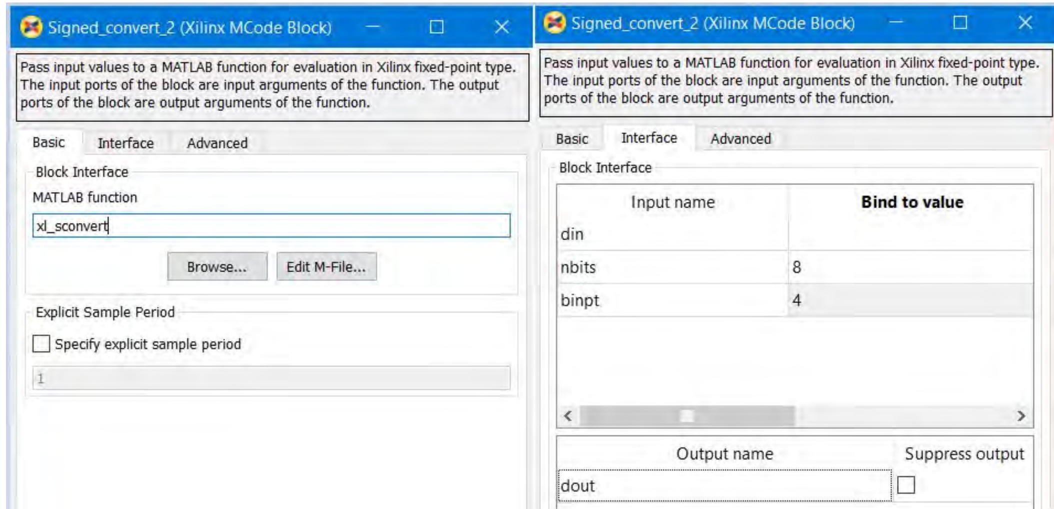
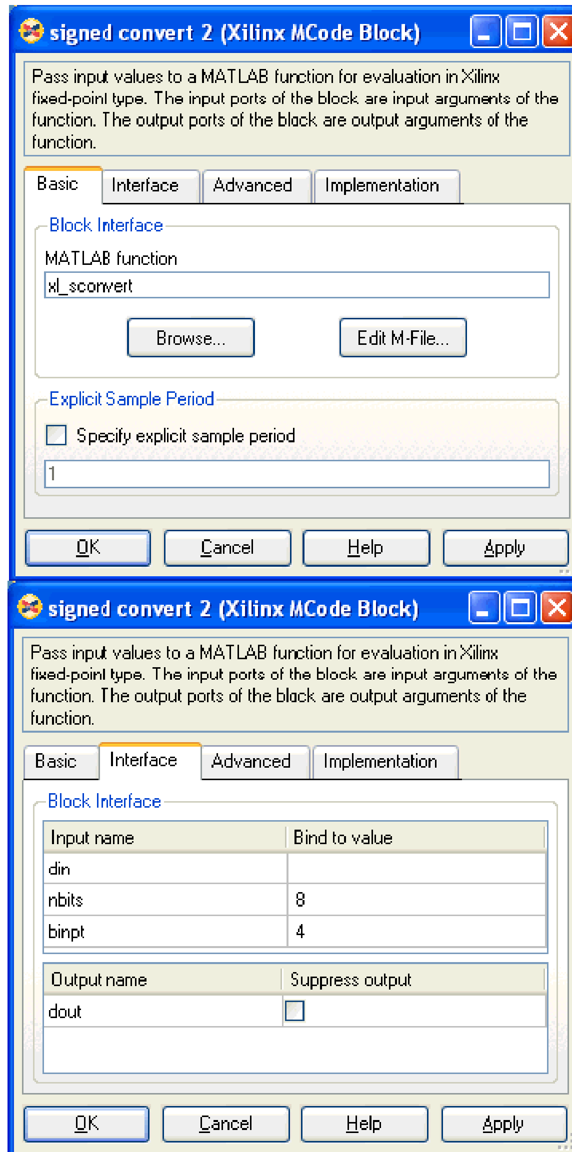


Figure 32: **Parameters**



The above interface window sets the M-function argument `nbits` to be 8 and `binpt` to be 4.

Optional Input Ports

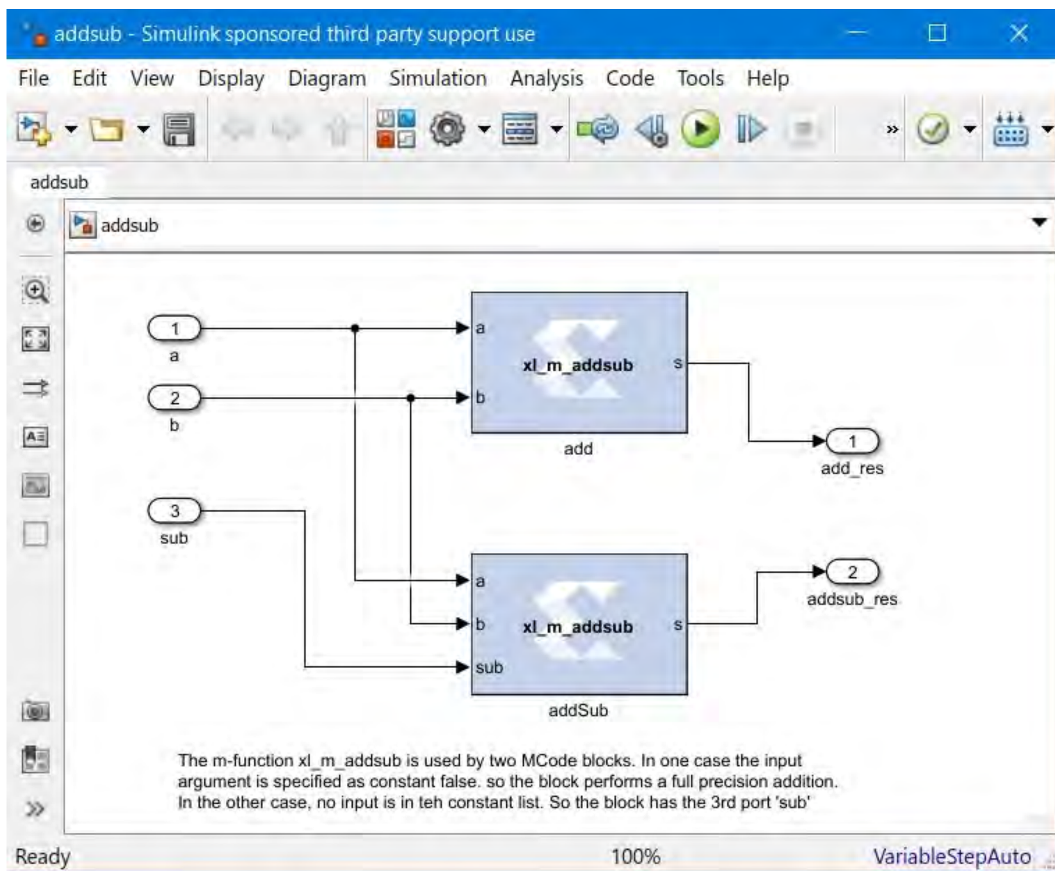
This example shows how to use the parameter passing mechanism of MCode blocks to specify whether or not to use optional input ports on MCode blocks.

The following M-code, which defines M-function `xl_m_addsub` is contained in file `xl_m_addsub.m`:

```
function s = xl_m_addsub(a, b, sub)
  if sub
    s = a - b;
  else
    s = a + b;
  end
```

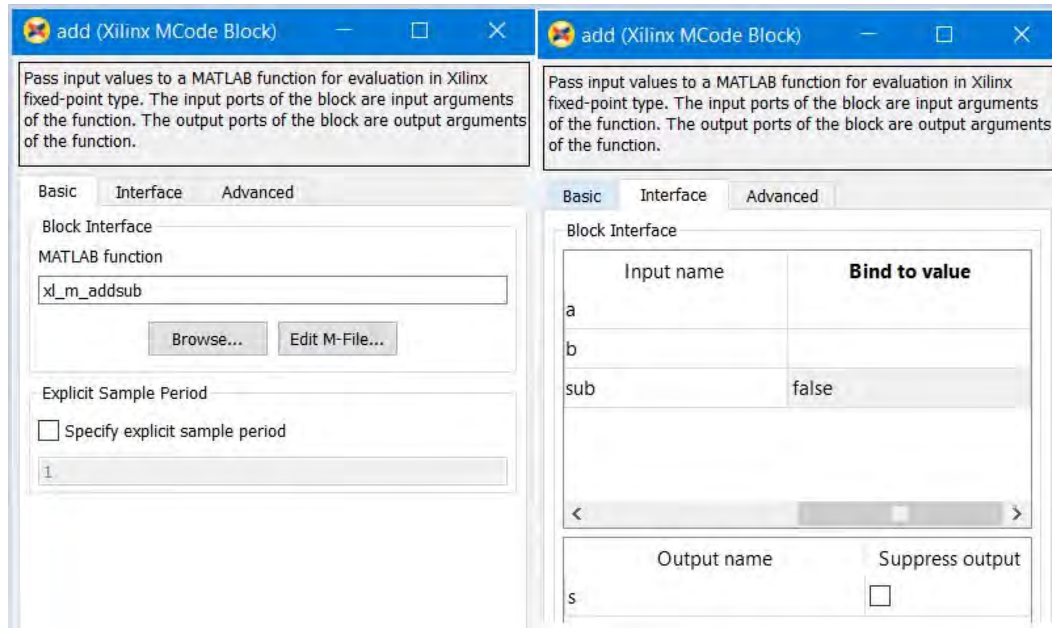
The following diagram shows a Subsystem containing two MCode blocks that use M-function `xl_m_addsub`.

Figure 33: **Two MCode Blocks Using M-Function**



The labeled add is shown in below.

Figure 34: **Block Interface Editor of the MCode Block**

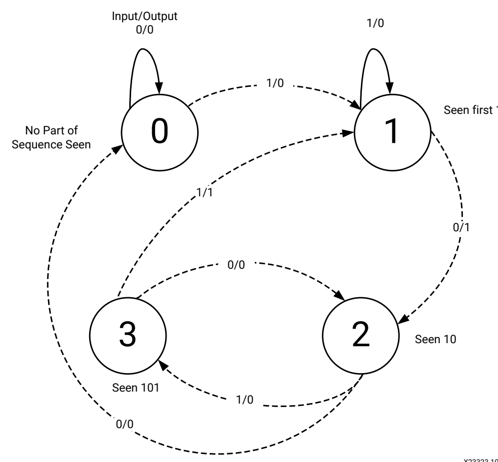


As a result, the `add` block features two input ports `a` and `b`; it performs full precision addition. Input parameter `sub` of the MCode block labeled `addsub` is not bound with any value. Consequently, the `addsub` block features three input ports: `a`, `b`, and `sub`; it performs full precision addition or subtraction based on the value of input port `sub`.

Finite State Machines

This example shows how to create a finite state machine using the MCode block with internal state variables. The state machine illustrated below detects the pattern 1011 in an input stream of bits.

Figure 35: **Finite State Machine Diagram**

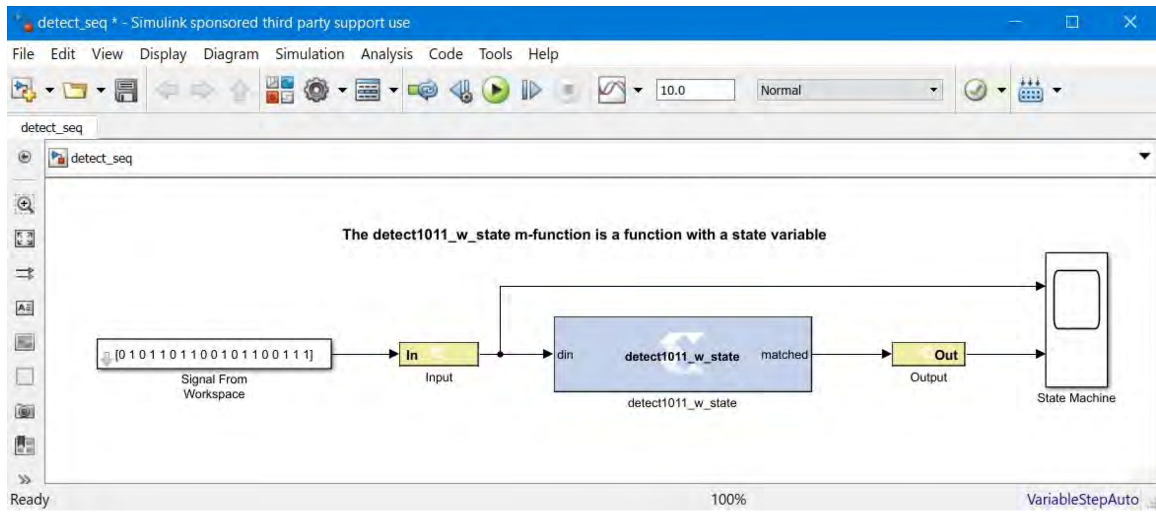


The M-function that is used by the MCode block contains a transition function, which computes the next state based on the current state and the current input. Unlike example 3 though, the M-function in this example defines persistent state variables to store the state of the finite state machine in the MCode block. The following M-code, which defines function `detect1011_w_state` is contained in file `detect1011_w_state.m`:

```
function matched = detect1011_w_state(din)
% This is the detect1011 function with states for detecting a
% pattern of 1011.

  seen_none = 0; % initial state, if input is 1, switch to seen_1
  seen_1 = 1;    % first 1 has been seen, if input is 0, switch
                 % seen_10
  seen_10 = 2;   % 10 has been detected, if input is 1, switch to
                 % seen_1011
  seen_101 = 3;  % now 101 is detected, is input is 1, 1011 is
                 % detected and the FSM switches to seen_1

  % the state is a 2-bit register
  persistent state, state = xl_state(seen_none, {xlUnsigned, 2, 0});

  % the default value of matched is false
  matched = false;

  switch state
    case seen_none
      if din==1
        state = seen_1;
      else
        state = seen_none;
      end
    case seen_1 % seen first 1
      if din==1
        state = seen_1;
      else
        state = seen_10;
      end
    case seen_10 % seen 10
      if din==1
        state = seen_101;
      else
      % no part of sequence seen, go to seen_none
      state = seen_none;
      end
    case seen_101
      if din==1
        state = seen_1;
        matched = true;
      else
        state = seen_10;
        matched = false;
      end
  end
```

The following diagram shows a state machine Subsystem containing a MCode block after compilation; the MCode block uses M-function `detect1101_w_state`.

Figure 36: **Subsystem Containing MCode Block After Compilation**



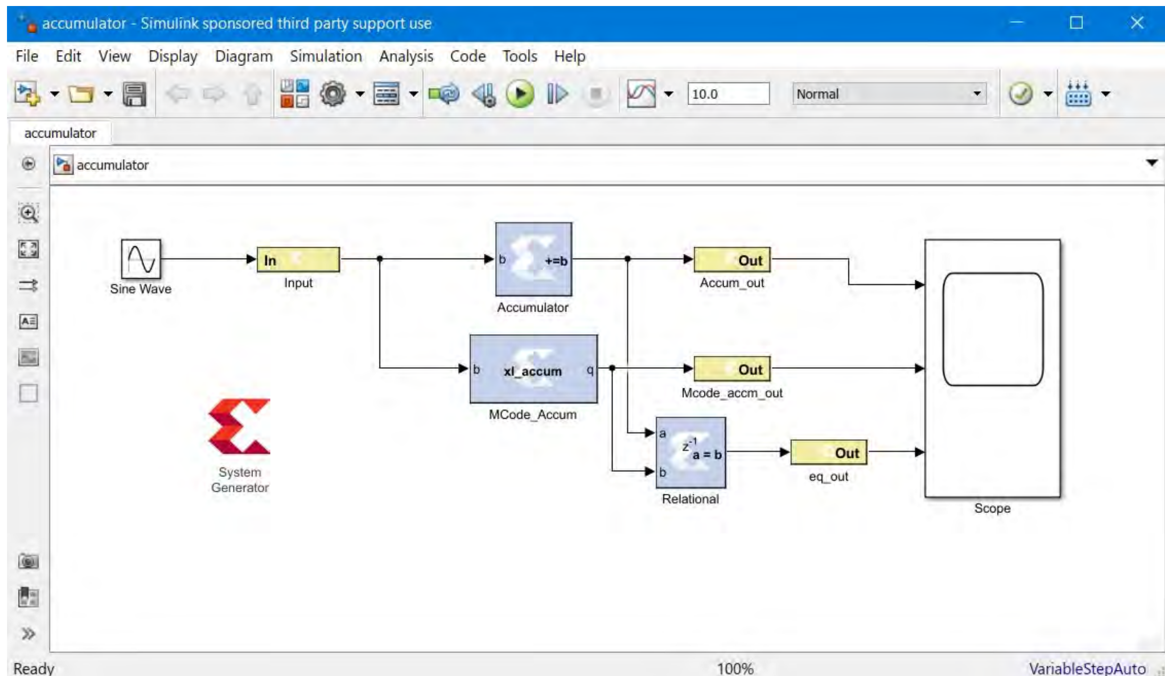
Parameterizable Accumulator

This example shows how to use the MCode block to build an accumulator using persistent state variables and parameters to provide implementation flexibility. The following M-code, which defines function `xl_accum` is contained in file `xl_accum.m`:

```
function q = xl_accum(b, rst, load, en, nbits, ov, op, feed_back_down_scale)
% q = xl_accum(b, rst, nbits, ov, op, feed_back_down_scale) is
% equivalent to our Accumulator block.
  binpt = xl_binpt(b);
  init = 0;
  precision = {xlSigned, nbits, binpt, xlTruncate, ov};
  persistent s, s = xl_state(init, precision);
  q = s;
  if rst
    if load
      % reset from the input port
      s = b;
    else
      % reset from zero
      s = init;
    end
  else
    if ~en
    else
      % if enabled, update the state
      if op==0
        s = s/feed_back_down_scale + b;
      else
        s = s/feed_back_down_scale - b;
      end
    end
  end
```

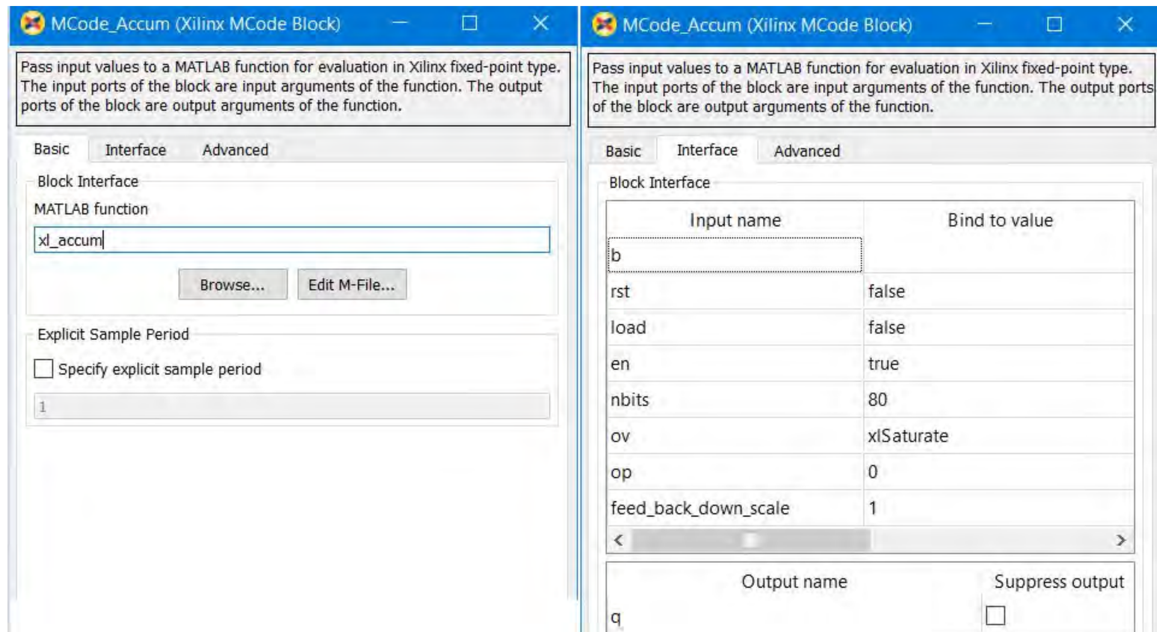
The following diagram shows a Subsystem containing the accumulator MCode block using M-function `xl_accum`. The MCode block is labeled MCode Accumulator. The Subsystem also contains the Xilinx Accumulator block, labeled Accumulator, for comparison purposes. The MCode block provides the same functionality as the Xilinx Accumulator block; however, its mask interface differs in that parameters of the MCode block are specified with a cell array in the Function Parameter Bindings parameter.

Figure 37: **MCode Accumulator**



Optional inputs `rst` and `load` of block `Accum_MCode1` are disabled in the cell array of the Function Parameter Bindings parameter. The block mask for block MCode Accumulator is shown below:

Figure 38: **Mask for MCode Accumulator**

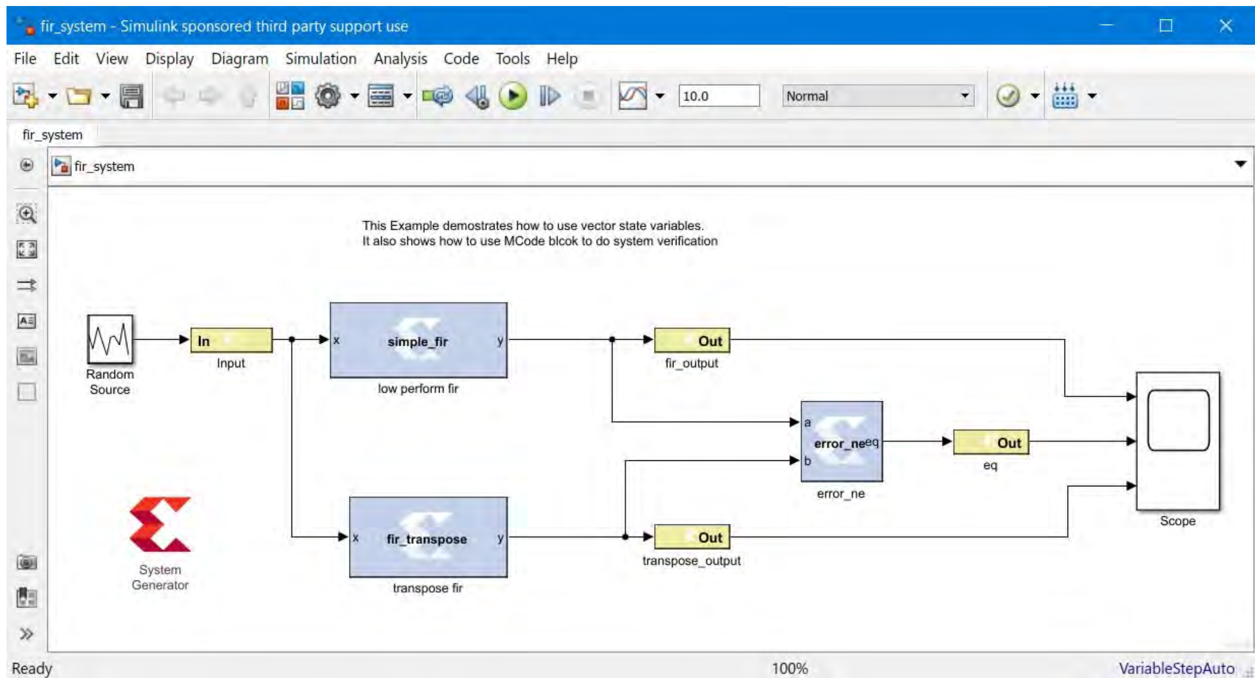


The example contains two additional accumulator Subsystems with MCode blocks using the same M-function, but different parameter settings to accomplish different accumulator implementations.

FIR Example and System Verification

This example shows how to use the MCode block to model FIRs. It also shows how to do system verification with the MCode block.

Figure 39: **FIR Example**



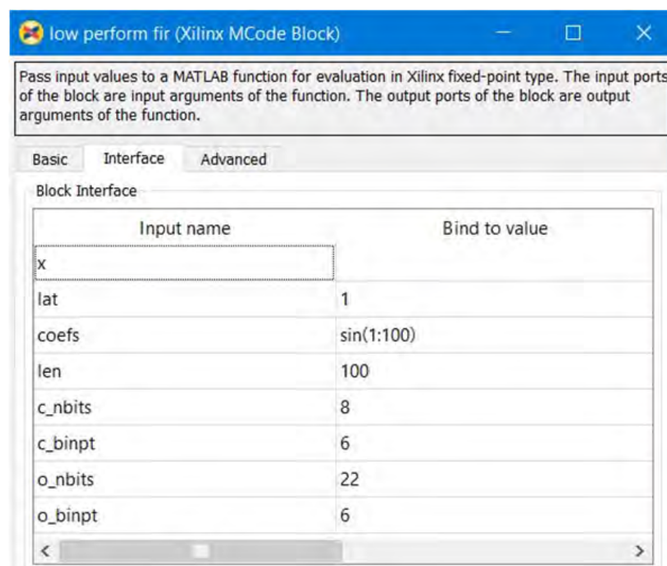
The model contains two FIR blocks. Both are modeled with the MCode block and both are synthesizable. The following are the two functions that model those two blocks.

```
function y = simple_fir(x, lat, coefs, len, c_nbits, c_binpt, o_nbits,
o_binpt)
  coef_prec = {xlSigned, c_nbits, c_binpt, xlRound, xlWrap};
  out_prec = {xlSigned, o_nbits, o_binpt};

  coefs_xfix = xfix(coef_prec, coefs);
  persistent coef_vec, coef_vec = xl_state(coefs_xfix, coef_prec);
  persistent x_line, x_line = xl_state(zeros(1, len-1), x);
  persistent p, p = xl_state(zeros(1, lat), out_prec, lat);

  sum = x * coef_vec(0);
  for idx = 1:len-1
    sum = sum + x_line(idx-1) * coef_vec(idx);
    sum = xfix(out_prec, sum);
  end
  y = p.back;
  p.push_front_pop_back(sum);
  x_line.push_front_pop_back(x);
function y = fir_transpose(x, lat, coefs, len, c_nbits, c_binpt, o_nbits,
o_binpt)
  coef_prec = {xlSigned, c_nbits, c_binpt, xlRound, xlWrap};
  out_prec = {xlSigned, o_nbits, o_binpt};
  coefs_xfix = xfix(coef_prec, coefs);
  persistent coef_vec, coef_vec = xl_state(coefs_xfix, coef_prec);
  persistent reg_line, reg_line = xl_state(zeros(1, len), out_prec);
  if lat <= 0
    error('latency must be at least 1');
  end
  lat = lat - 1;
  persistent dly,
```

```
  if lat <= 0
    y = reg_line.back;
  else
    dly = xl_state(zeros(1, lat), out_prec, lat);
    y = dly.back;
    dly.push_front_pop_back(reg_line.back);
  end
  for idx = len-1:-1:1
    reg_line(idx) = reg_line(idx - 1) + coef_vec(len - idx - 1) * x;
  end
  reg_line(0) = coef_vec(len - 1) * x;
```

The parameters are configured as following:

Figure 40: **Parameters**



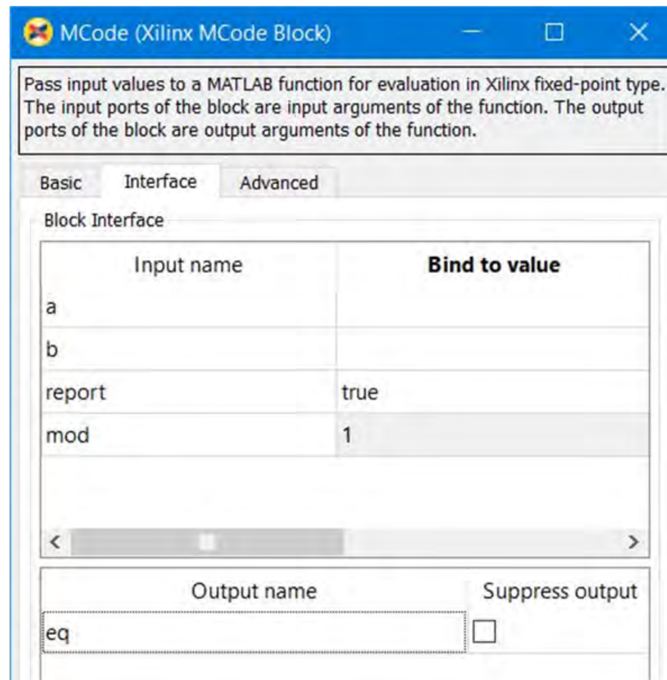
In order to verify that the functionality of two blocks is equal, we also use another MCode block to compare the outputs of two blocks. If the two outputs are not equal at any given time, the error checking block will report the error. The following function does the error checking:

```
function eq = error_ne(a, b, report, mod)
  persistent cnt, cnt = xl_state(0, {xlUnsigned, 16, 0});
  switch mod
    case 1
      eq = a==b;
    case 2
      eq = isnan(a) || isnan(b) || a == b;
    case 3
      eq = ~isnan(a) && ~isnan(b) && a == b;
    otherwise
      eq = false;
    error(['wrong value of mode ', num2str(mod)]);
  end
  if report
```

```
    if ~eq
      error(['two inputs are not equal at time ', num2str(cnt)]);
    end
  end
  cnt = cnt + 1;
```

The block is configured as following:

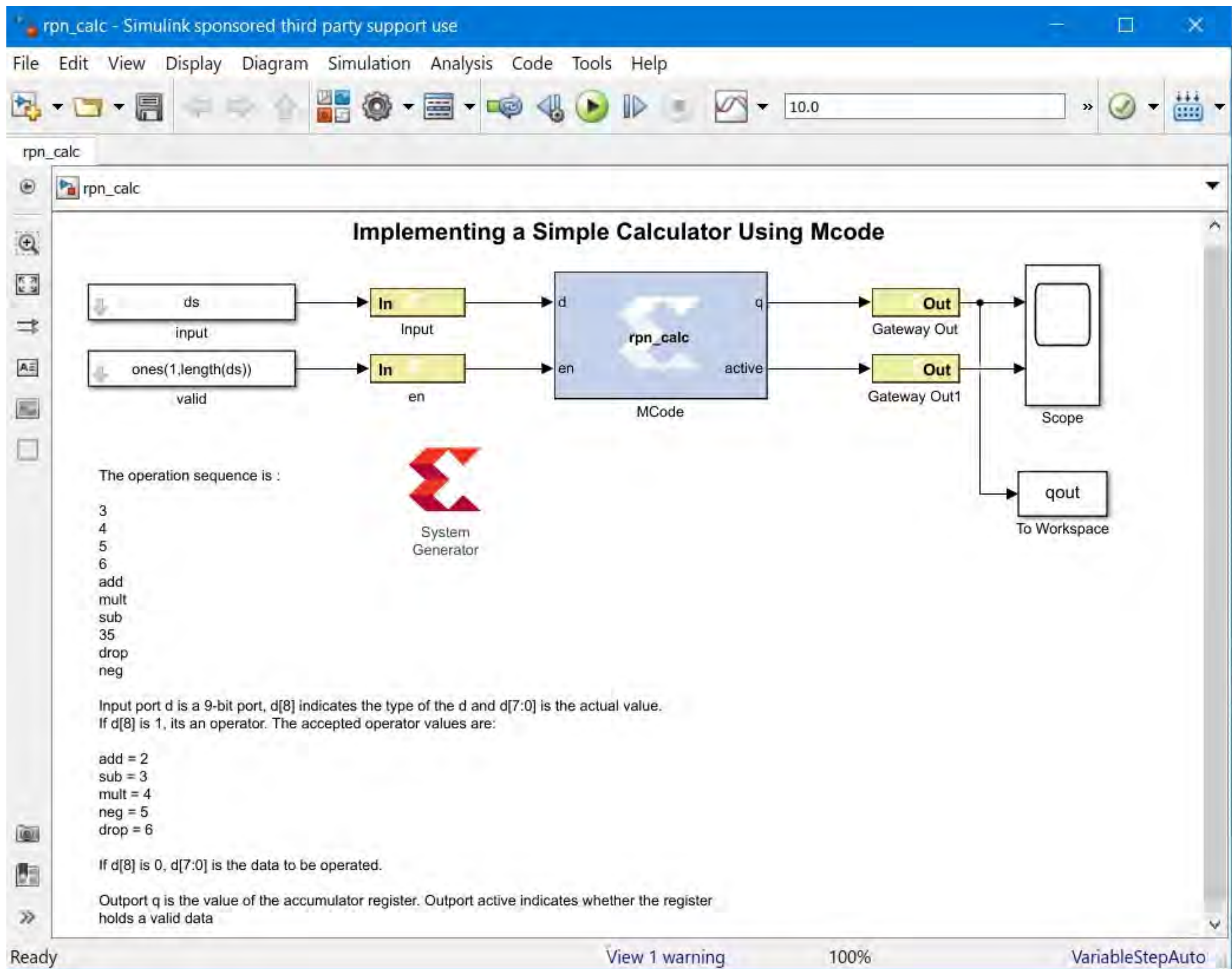
Figure 41: **Block Configuration**



RPN Calculator

This example shows how to use the MCode block to model a RPN calculator which is a stack machine. The block is synthesizable:

Figure 42: **RPN Calculator**



The following function models the RPN calculator.

```
function [q, active] = rpn_calc(d, rst, en)
  d_nbits = xl_nbits(d);
  % the first bit indicates whether it's a data or operator
  is_oper = xl_slice(d, d_nbits-1, d_nbits-1)==1;
  din = xl_force(xl_slice(d, d_nbits-2, 0), xlSigned, 0);
  % the lower 3 bits are operator
  op = xl_slice(d, 2, 0);
  % acc the the A register
  persistent acc, acc = xl_state(0, din);
  % the stack is implemented with a RAM and
  % an up-down counter
  persistent mem, mem = xl_state(zeros(1, 64), din);
  persistent acc_active, acc_active = xl_state(false, {xlBoolean});
  persistent stack_active, stack_active = xl_state(false, ...
                                                      {xlBoolean});
  stack_pt_prec = {xlUnsigned, 5, 0};
  persistent stack_pt, stack_pt = xl_state(0, {xlUnsigned, 5, 0});
  % when en is true, it's action
  OP_ADD = 2;
  OP_SUB = 3;
```

```
  OP_MULT = 4;
  OP_NEG = 5;
  OP_DROP = 6;
  q = acc;
  active = acc_active;
  if rst
    acc = 0;
    acc_active = false;
    stack_pt = 0;
  elseif en
    if ~is_oper
      % enter data, push
      if acc_active
        stack_pt = xfix(stack_pt_prec, stack_pt + 1);
        mem(stack_pt) = acc;
        stack_active = true;
      else
        acc_active = true;
      end
      acc = din;
    else
      if op == OP_NEG
        % unary op, no stack op
        acc = -acc;
      elseif stack_active
        b = mem(stack_pt);
        switch double(op)
          case OP_ADD
            acc = acc + b;
          case OP_SUB
            acc = b - acc ;
          case OP_MULT
            acc = acc * b;
          case OP_DROP
            acc = b;
        end
        stack_pt = stack_pt - 1;
      elseif acc_active
        acc_active = false;
        acc = 0;
      end
    end
  end
  stack_active = stack_pt ~= 0;
```

Example of disp Function

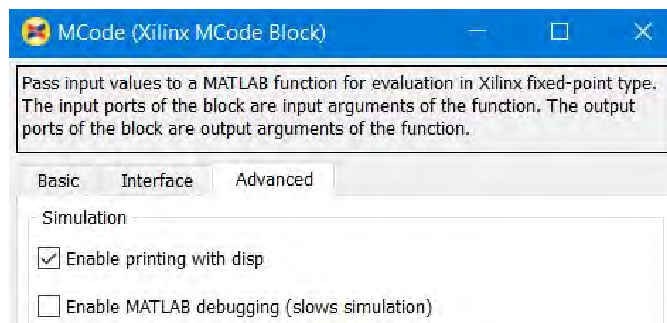
The following MCode function shows how to use the disp function to print variable values.

```
function x = testdisp(a, b)
  persistent dly, dly = xl_state(zeros(1, 8), a);
  persistent rom, rom = xl_state([3, 2, 1, 0], a);
  disp('Hello World!');
  disp(['num2str(dly) is ', num2str(dly)]);
  disp('disp(dly) is ');
  disp(dly);
  disp('disp(rom) is ');
  disp(rom);
  a2 = dly.back;
  dly.push_front_pop_back(a);
  x = a + b;
```

```
  disp(['a = ', num2str(a), ', ', ...
        'b = ', num2str(b), ', ', ...
        'x = ', num2str(x)]);
  disp(num2str(true));
  disp('disp(10) is');
  disp(10);
  disp('disp(-10) is');
  disp(-10);
  disp('disp(a) is ');
  disp(a);
  disp('disp(a == b)');
  disp(a==b);
```

Check the **Enable printing with disp** option.

Figure 43: **Enable Printing with disp**



Here are the lines that are displayed on the MATLAB console for the first simulation step.

```
mcode_block_disp/MCode (Simulink time: 0.000000, FPGA clock: 0)
Hello World!
num2str(dly) is [0.000000, 0.000000, 0.000000, 0.000000, 0.000000,
0.000000, 0.000000, 0.000000]
disp(dly) is
  type: Fix_11_7,
  maxlen: 8,
  length: 8,
  0: binary 0000.0000000, double 0.000000,
  1: binary 0000.0000000, double 0.000000,
  2: binary 0000.0000000, double 0.000000,
  3: binary 0000.0000000, double 0.000000,
  4: binary 0000.0000000, double 0.000000,
  5: binary 0000.0000000, double 0.000000,
  6: binary 0000.0000000, double 0.000000,
  7: binary 0000.0000000, double 0.000000,
disp(rom) is
  type: Fix_11_7,
  maxlen: 4,
  length: 4,
  0: binary 0011.0000000, double 3.0,
  1: binary 0010.0000000, double 2.0,
  2: binary 0001.0000000, double 1.0,
  3: binary 0000.0000000, double 0.0,
a = 0.000000, b = 0.000000, x = 0.000000
1
disp(10) is
  type: UFix_4_0, binary: 1010, double: 10.0
disp(-10) is
```

```
  type: Fix_5_0, binary: 10110, double: -10.0
disp(a) is
  type: Fix_11_7, binary: 0000.0000000, double: 0.000000
disp(a == b)
  type: Bool, binary: 1, double: 1
```

Importing a System Generator Design into a Bigger System

A System Generator design is often a sub-design that is incorporated into a larger HDL design. This topic shows how to embed two System Generator designs into a larger design and how VHDL created by System Generator can be incorporated into the simulation model of the overall system.

HDL Netlist Compilation

Selecting the **HDL Netlist** compilation target from the System Generator token instructs System Generator to generate HDL along with other related files that implement the design. In addition, System Generator produces auxiliary files that simplify downstream processing such as simulating the design using an Vivado simulator, and performing logic synthesis using Vivado synthesis. See Chapter 8: System Generator Compilation Types for more details.

Integration Design Rules

When a System Generator model is to be included into a larger design, the following two design rules must be followed.

- **Rule 1:** No Gateway or System Generator token should specify an IOB/CLK location.

Also, IOB timing constraints should be set to "none".

- **Rule 2:** If there are any I/O ports from the System Generator design that are required to be ports on the top-level design, appropriate buffers should be instantiated in the top-level HDL code.

Configurable Subsystems and System Generator

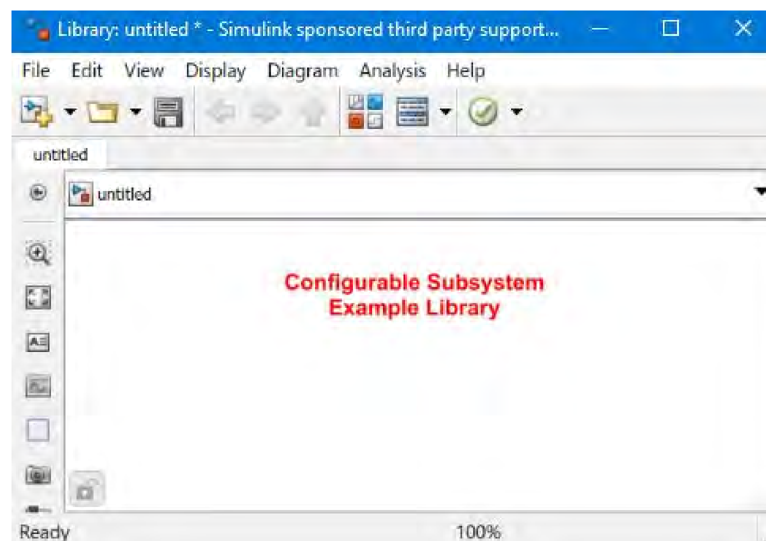
A configurable Subsystem is a kind of block that is made available as a standard part of Simulink. In effect, a configurable Subsystem is a block for which you can specify several underlying blocks. Each underlying block is a possible implementation, and you are free to choose which implementation to use. In System Generator you might, for example, specify a general-purpose FIR filter as a configurable Subsystem whose underlying blocks are specific FIR filters. Some of the underlying filters might be fast but require much hardware, while others are slow but require less hardware. Switching the choice of the underlying filter allows you to perform experiments that trade hardware cost against speed.

Defining a Configurable Subsystem

A configurable Subsystem is defined by creating a Simulink® library. The underlying blocks that implement a configurable Subsystem are organized in this library. To create such a library, do the following:

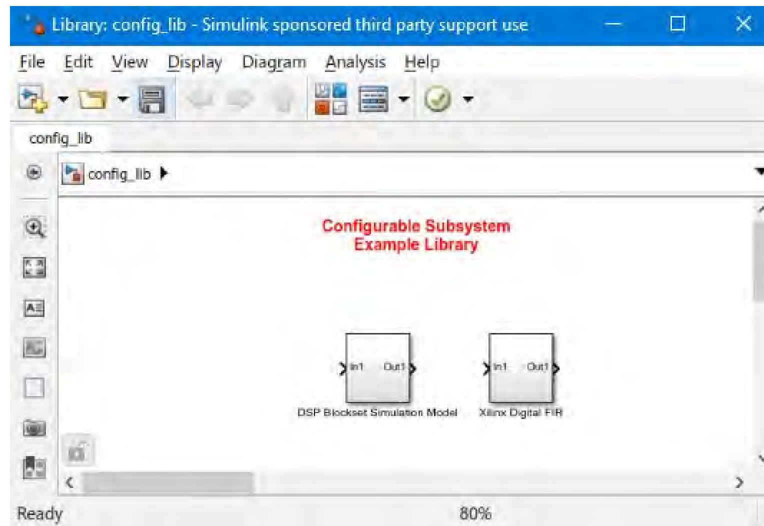
1. Make a new empty library.

Figure 44: **New Empty Library**



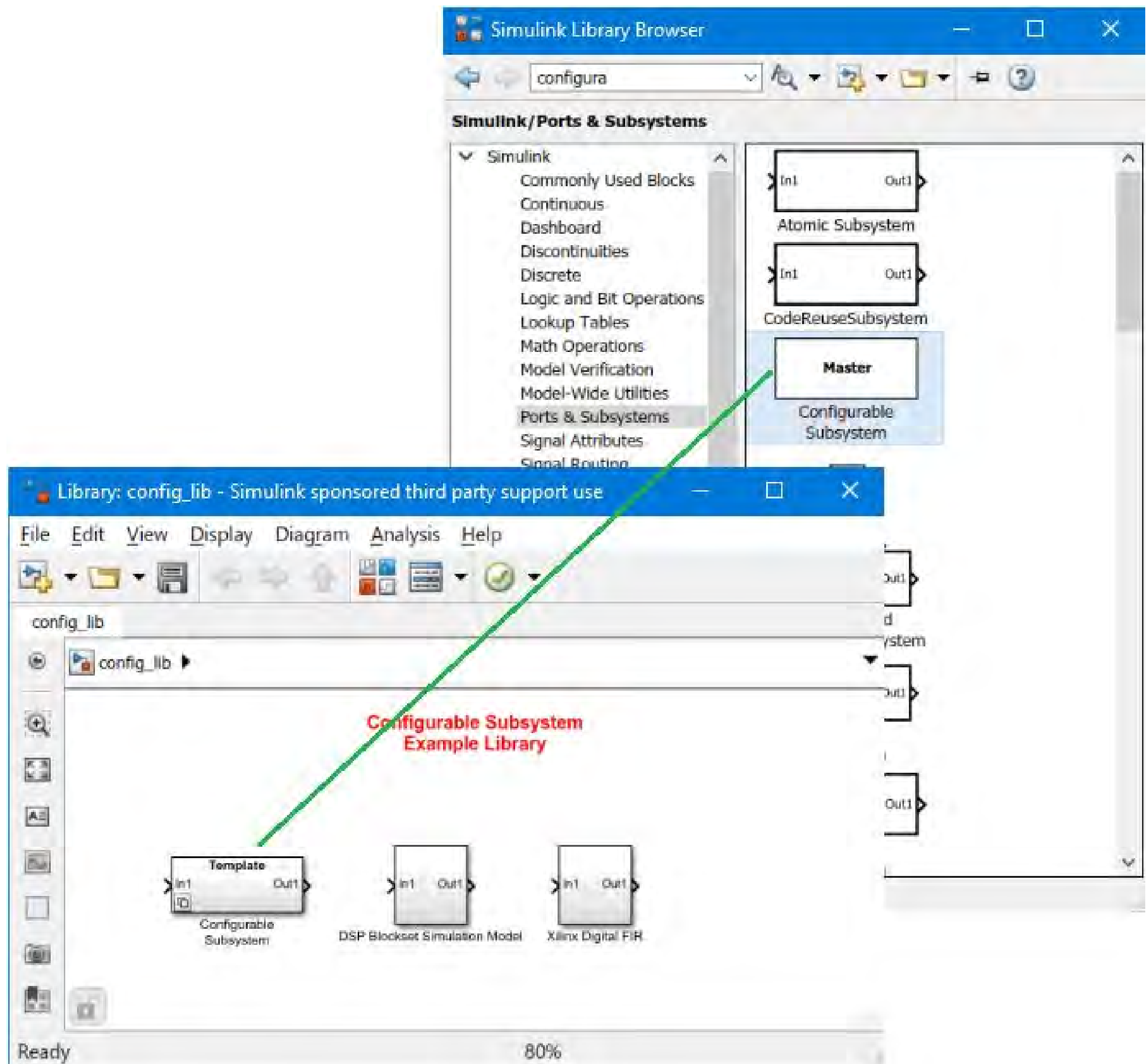
2. Add the underlying blocks to the library.

Figure 45: **Adding Underlying Blocks**



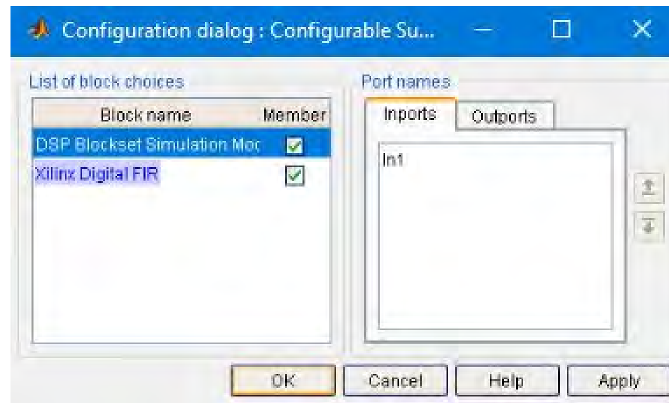
3. Drag a template block into the library. (Templates can be found in the Simulink library browser under Simulink/Ports & Subsystems/Configurable Subsystem.)

Figure 46: **Template**



4. Rename the template block if desired.
5. Save the library.
6. Double-click to open the template for the library.
7. In the template GUI, turn on each check box corresponding to a block that should be an implementation.

Figure 47: **Check Boxes**



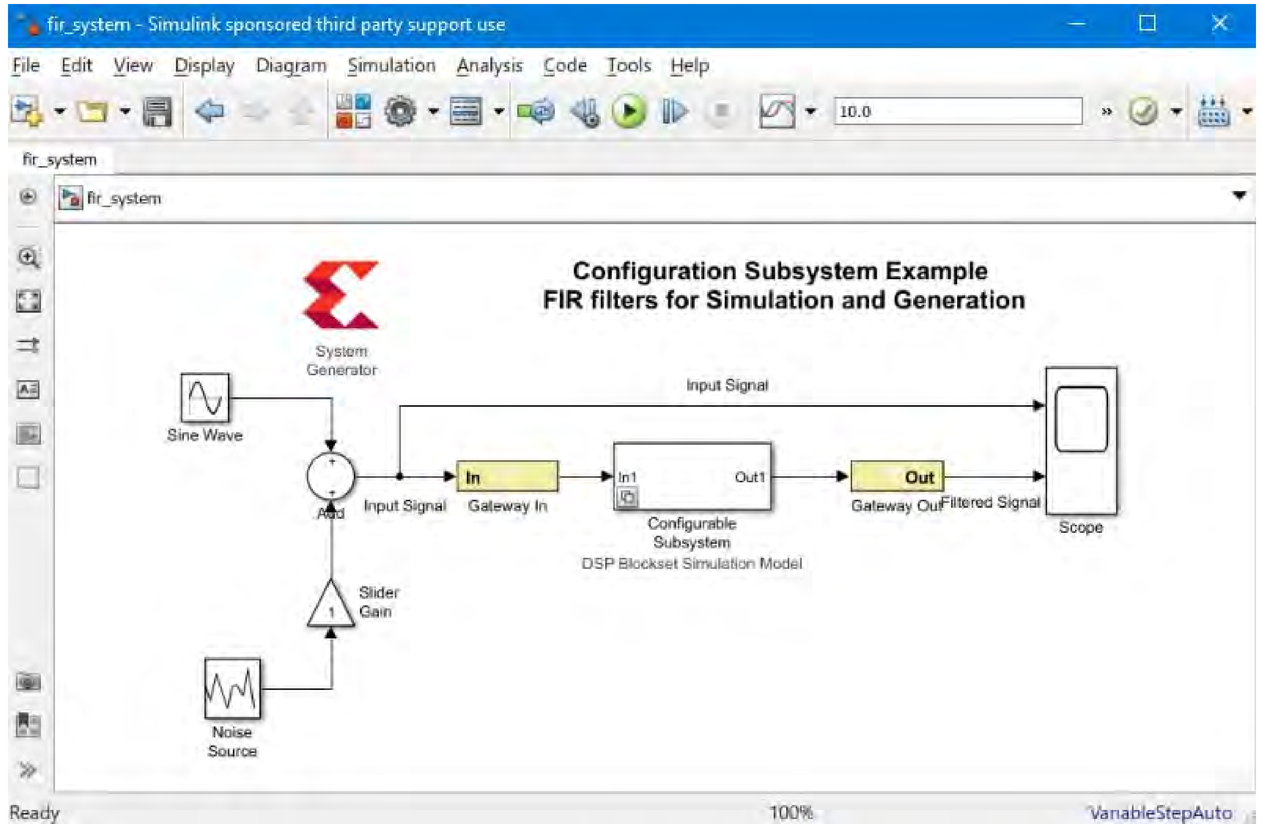
8. Press **OK**, and then save the library again.

Using a Configurable Subsystem

To use a configurable Subsystem in a design, do the following:

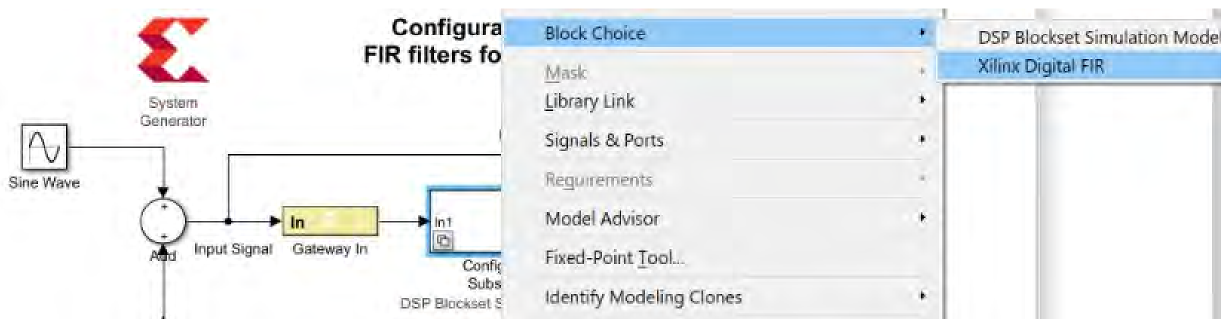
1. As described above, create the library that defines the configurable Subsystem.
2. Open the library.
3. Drag a copy of the template block from the library to the appropriate part of the design.
4. The copy becomes an instance of the configurable Subsystem.

Figure 48: **Configurable Subsystem**



5. Right-click the instance, and under **Block Choice** select the block that should be used as the underlying implementation for the instance.

Figure 49: **Block Choice**



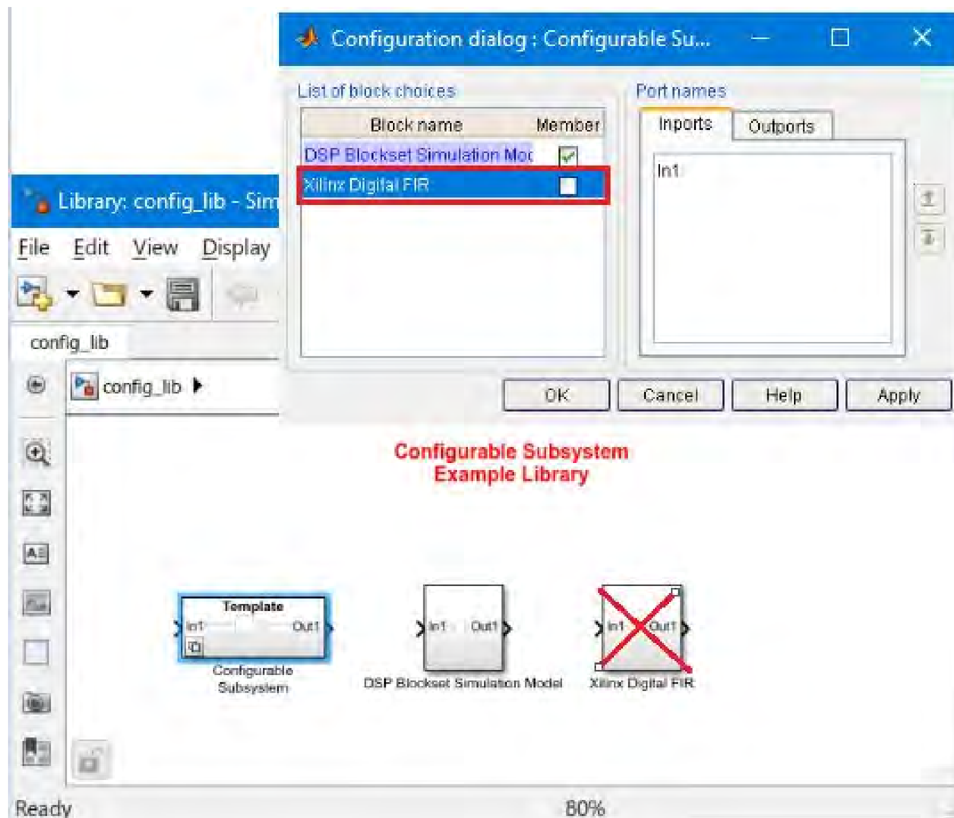
Deleting a Block from a Configurable Subsystem

To delete an underlying block from a configurable Subsystem, do the following:

1. Open and unlock the library for the Subsystem.

2. Double-click the template, and deselect the checkbox associated with the block to be deleted.
3. Press **OK**, and then delete the block.

Figure 50: **Deleting a Block**



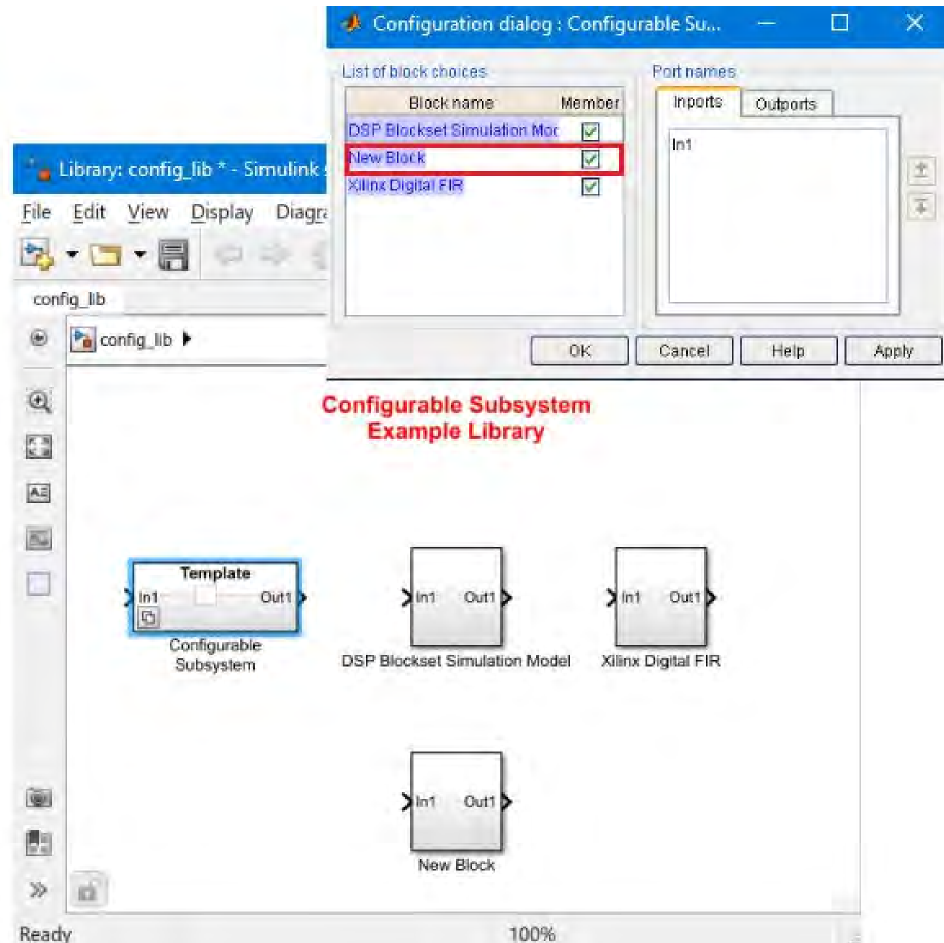
4. Save the library.
5. Compile the design by typing **Ctrl + D**.
6. If necessary, update the choice for each instance of the configurable Subsystem.

Adding a Block to a Configurable Subsystem

To add an underlying block to a configurable Subsystem, do the following:

1. Open and unlock the library for the Subsystem.
2. Drag a block into the library.
3. Double-click the template, and select the checkbox next to the added block.

Figure 51: **Add a Block**



4. Press **OK**, and then save the library.
5. Compile the design by typing **Ctrl-D**.
6. If necessary, update the choice for each instance of the configurable Subsystem.

Notes for Higher Performance FPGA Design

If you focus all your optimization efforts using the back-end implementation tools, you may not be able to achieve timing closure because of the following reasons:

- The more complex IP blocks in a System Generator design like FIR Compiler and FFT are generated under the hood. They are provided as highly-optimized netlists to the synthesis tool and the implementation tools, so further optimization may not be possible.

- System Generator netlisting produces HDL code with many instantiated primitives such as registers, BRAMs, and DSP48E1s. There is not much a synthesis tool can do to optimize these elements.

The following tips focus on what you can do in System Generator to increase the performance of your design before you start the implementation process.

- Review the Hardware Notes Included with Each Block Dialog Box
- Register the Inputs and Outputs of Your Design
- Insert Pipeline Registers
- Use Saturation Arithmetic and Rounding Only When Necessary
- Set the Data Rate Option on All Gateway Blocks
- Pipeline for Maximum Performance
- Other Things to Try

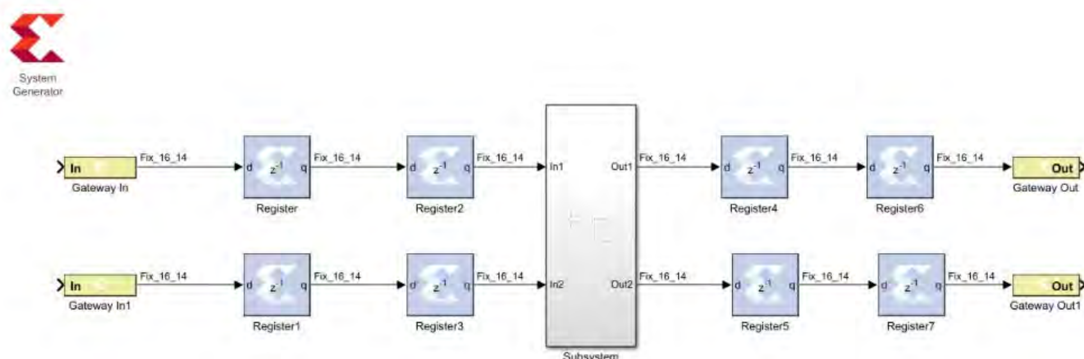
Review the Hardware Notes Included with Each Block Dialog Box

Pay close attention to the Hardware Notes included in the block dialog boxes. Many blocks in the Xilinx Blockset library have notes that explain how to achieve the most hardware efficient implementation. For example, the notes point out that the Scale block costs nothing in hardware. By contrast, the Shift block (which is sometimes used for the same purpose) can use hardware.

Register the Inputs and Outputs of Your Design

Register the inputs and outputs of your design. As shown below, this can be done by placing one or more Delay blocks with a latency 1 or Register blocks after the Gateway In and before Gateway Out blocks. Selecting any of the Register block features adds hardware.

Figure 52: **Register Inputs and Outputs**

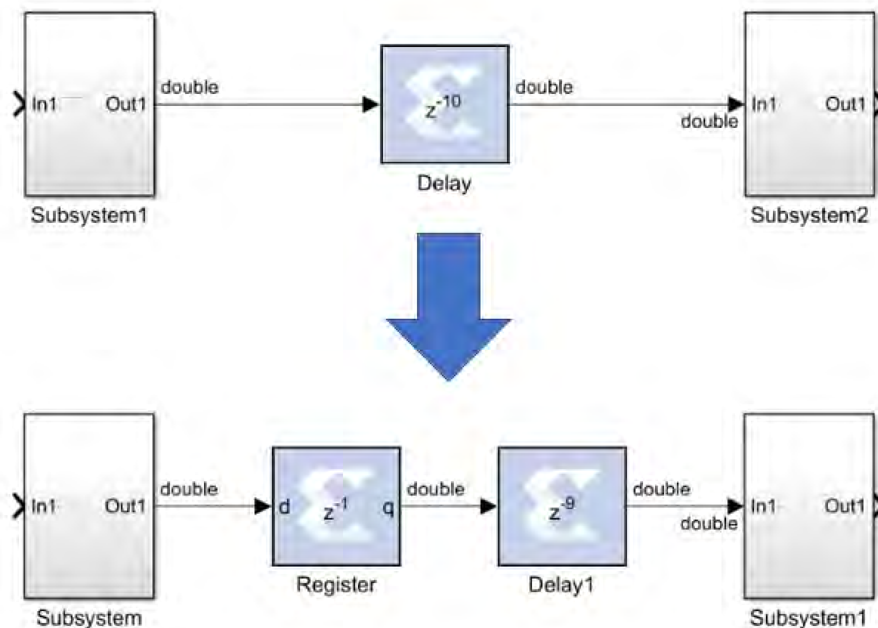


Double registering the I/Os may also be beneficial. This can be performed by instantiating two separate Register blocks, or by instantiating two Delay blocks, each having latency 1. This allows one of the registers to be packed into the IOB and the other to be placed next to the logic in the FPGA fabric. A Delay block with latency 2 does not give the same result because the block with a latency of 2 is implemented using an SRL32 and cannot be packed into an IOB.

Insert Pipeline Registers

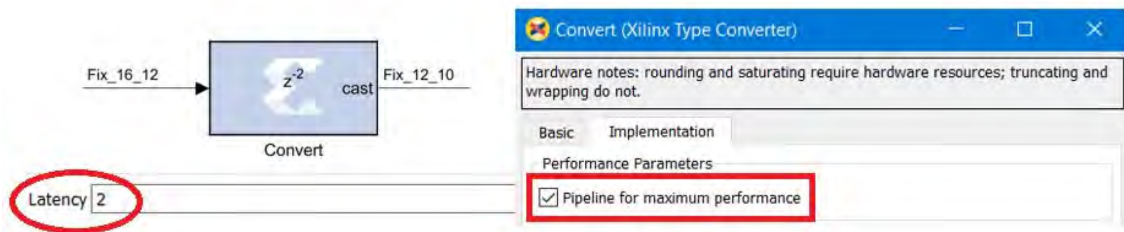
Insert pipeline registers wherever possible and reasonable. Deep pipelines are efficiently implemented with the Delay blocks since the SRL32 primitive is used. If an initial value is needed on a register, the Register block should be used. Also, if the input path of an SRL32 is failing timing, you should place a Register block before the related Delay block and reduce the latency of the Delay block by one. This allows the router more flexibility to place the Register and Delay block (SRL + Register) away from each other to maximize the margin for the routing delay of this path.

Figure 53: **Pipeline Registers**



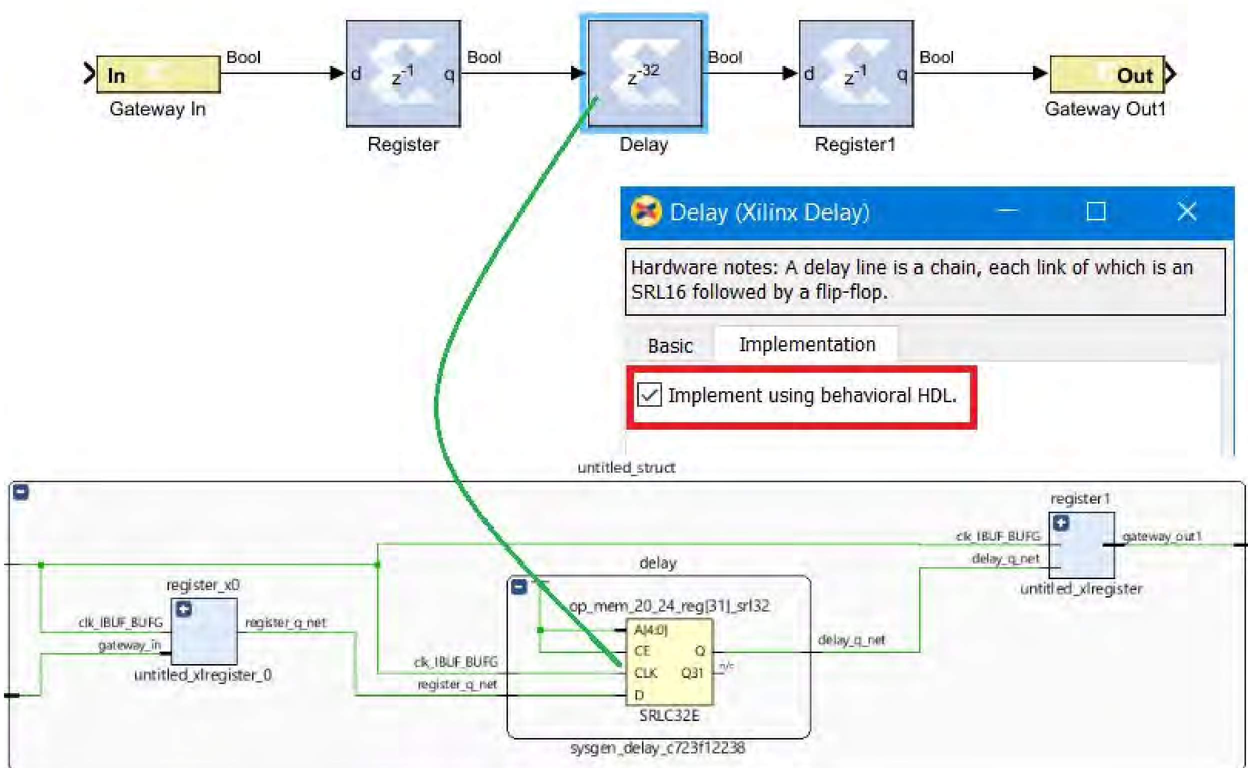
As shown in the following figure, the Convert block can be pipelined with embedded register stages to guarantee maximum performance.

Figure 54: **Convert Block**



To achieve a more efficient implementation on some Xilinx blocks, you can select the **Implement using behavioral HDL** option. As shown below, if the delay on a Delay block is 32 or greater, Xilinx synthesis infers a SRLC32E (32-bit Shift-Register) which maps into a single LUT.

Figure 55: **Implement Using Behavioral HDL**



For block RAMs (BRAMs), use the internal output register. You do this by setting the latency from 1 (the default) to 2. This enables the block RAM output register.

When you are using DSP48E1s, use the input, output and internal registers; for FIFOs, use the embedded registers option. Also, check all the high-level IP blocks for pipelining options.

Use Saturation Arithmetic and Rounding Only When Necessary

Saturation arithmetic and rounding have area and performance costs. Use only if necessary. For example a Reinterpret block doesn't cost any logic. A Convert (cast) block doesn't cost any logic if Quantization is set to Truncate and if Overflow is set to Wrap. If the data type requires the use of the Rounding and Saturation options, then pipeline the Convert block with embedded register stages. If you are using a DSP48E1, the rounding can be done within the DSP48E1.

Set the Data Rate Option on All Gateway Blocks

Select the IOB timing constraint option **Data Rate** on all **Gateway In** and **Gateway Out** blocks. When **Data Rate** is selected, the IOBs are constrained at the data rate at which the IOBs operate. The rate is determined by the **Simulink system period (sec)** field in the System Generator token and the sample rate of the Gateway relative to the other sample periods in the design.

Pipeline for Maximum Performance

For System Generator blocks that use Xilinx LogiCORE™ IP internally, the default tool behavior is to place at least one register outside of the core. For latency values greater than the optimum value of the core, the optimal pipeline registers are placed inside the core, and the remainder of the registers get pushed out.

Other Things to Try

- Change the Source Design
    - Use Additional Pipelining

      Use the Output and Pipeline registers inside block RAM and DSP48s.
    - Run Functions in Parallel

      Run functions in parallel at a slower clock rate
    - Use Retiming Techniques

      Move existing registers through combinational logic.
    - Use Hard Cores where Possible

      Use Block RAM instead of distributed RAM.
    - Use a Different Design Approach for Functions
- Avoid Over-Constraining the Design

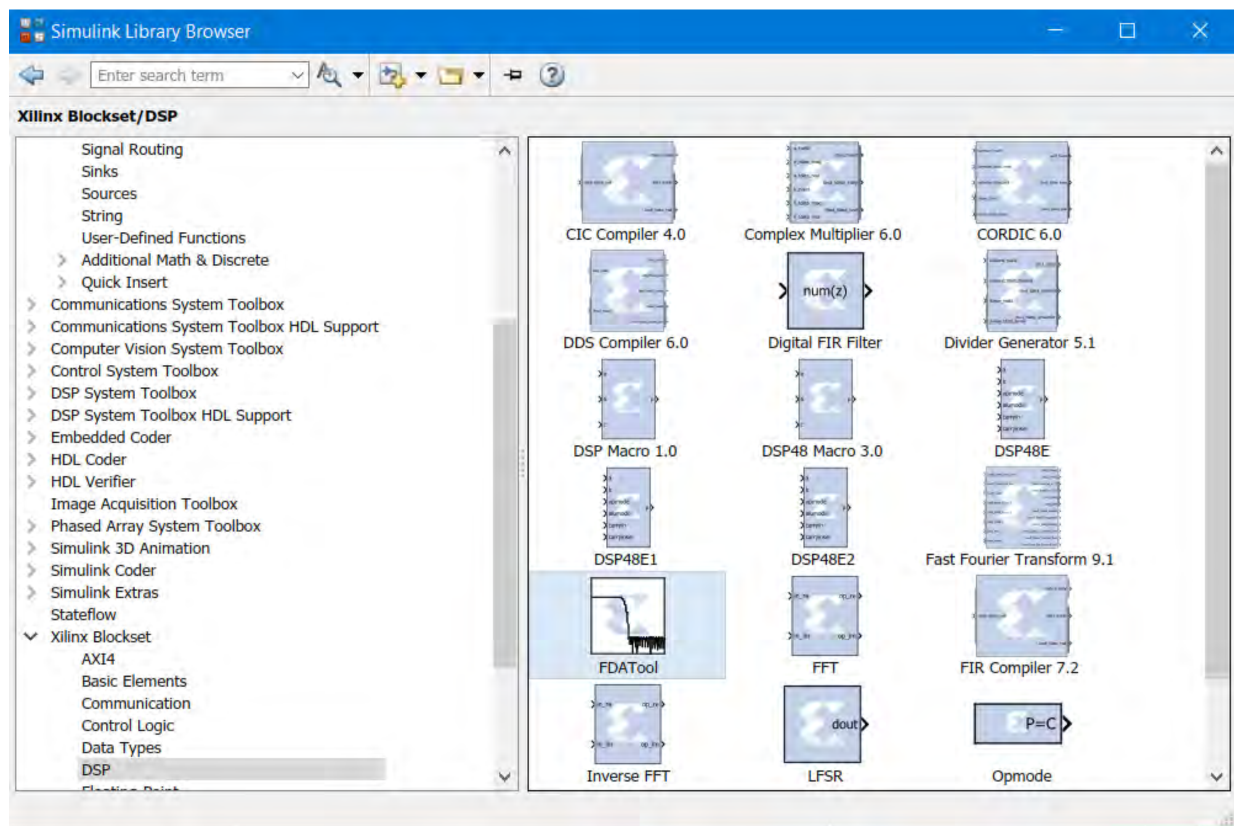
  Do not over-constrain the design and use up/down sample blocks where appropriate.

- Consider Decreasing the Frequency of Critical Design Modules
- Squeeze Out the Implementation Tools
    - Try Different Synthesis Options.
    - Floorplan Critical Modules

Using FDATool in Digital Filter Applications

The FDATool block is used to define the filter order and coefficients, and the Xilinx® Blocksets are used to implement a filter. The DSP library in the Xilinx Blockset contains the FDATool block.

Figure 56: **FDATool Block**



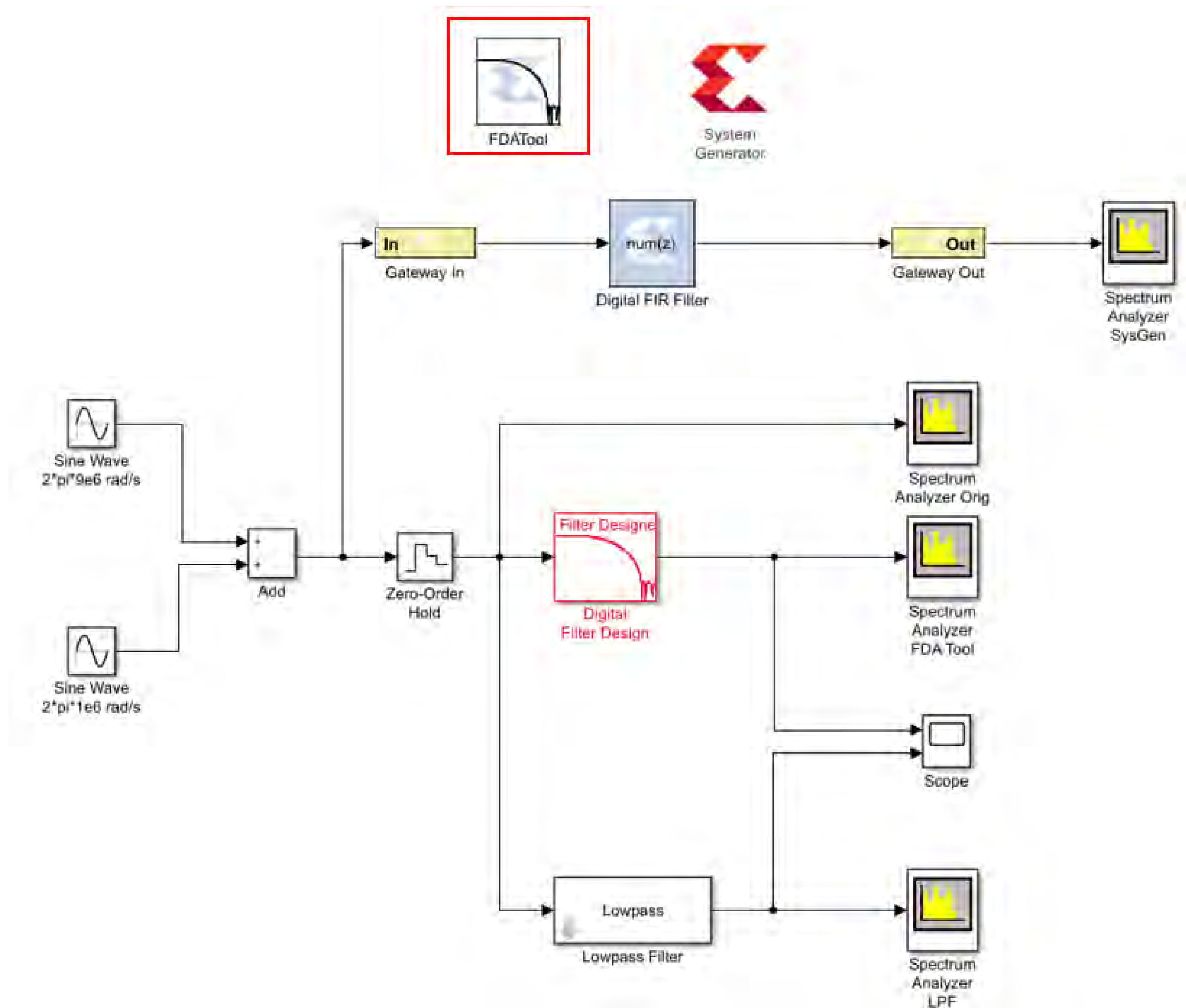
A simple System Generator model below illustrates a standard FIR filter design using the FDATool and digital FIR filter block.

The design uses two sine wave sources which are being added together and passed separately through two low-pass filters.

- The first filter is the one that could be implemented using Xilinx blockset. It is a digital low pass filter implemented using the Digital FIR filter block.
- The second filter is what is referred to as a reference filter. A low pass filter is implemented using a Direct-form FIR structure.

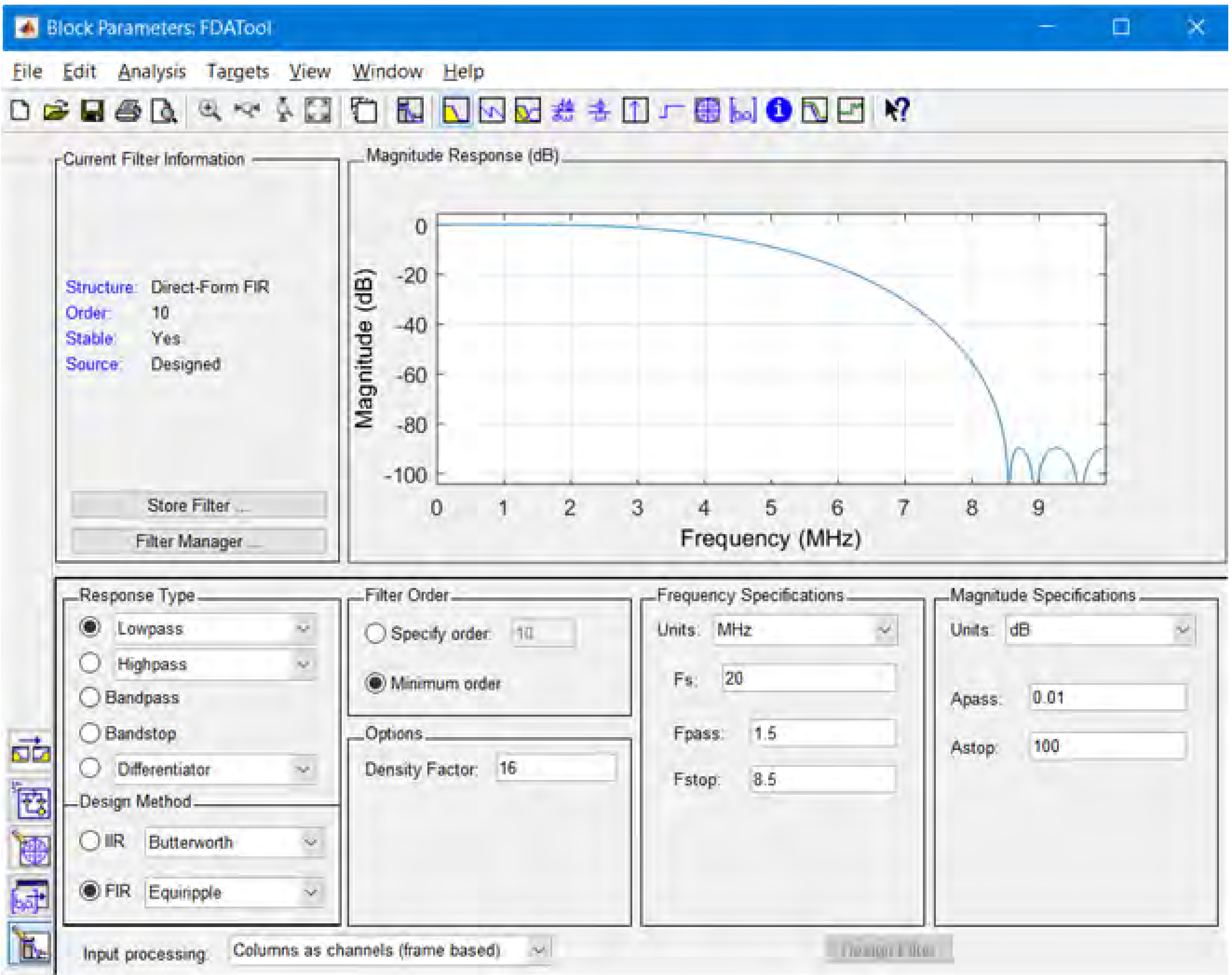
The frequency response of both filters visualized in Spectrum Analyzer block.

Figure 57: **Spectrum Analyzer Block**



The Xilinx version of the FDAtool can be used to define the coefficients of the low-pass filter to eliminate high-frequency noise. The filter configuration parameters like Response Type, Filter Order, Frequency Specification, and Magnitude Specification can be modified from the Properties Editor of the FDATool as shown below.

Figure 58: **Filter Configuration**



The Design Filter option at the bottom of the tool window allows you to find out the filter order and observe the Magnitude Response. You can also view the Phase Response, Impulse Response, Coefficients, and more by selecting the appropriate icon at the top-right of the window. You can display the filter coefficients in the MATLAB® workspace by typing the following:

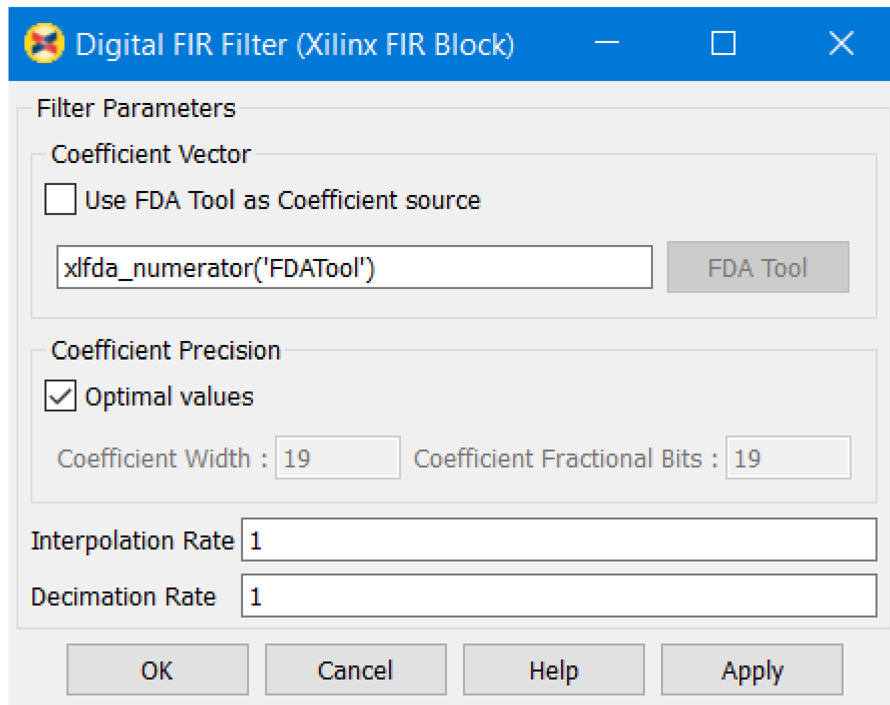
```
>> xlfda_numerator('FDATool')
```

The following functions help you find the maximum and minimum coefficient values to adequately specify the coefficient width and binary point:

```
>> max(xlfda_numerator('FDATool'))
>> min(xlfda_numerator('FDATool'))
```

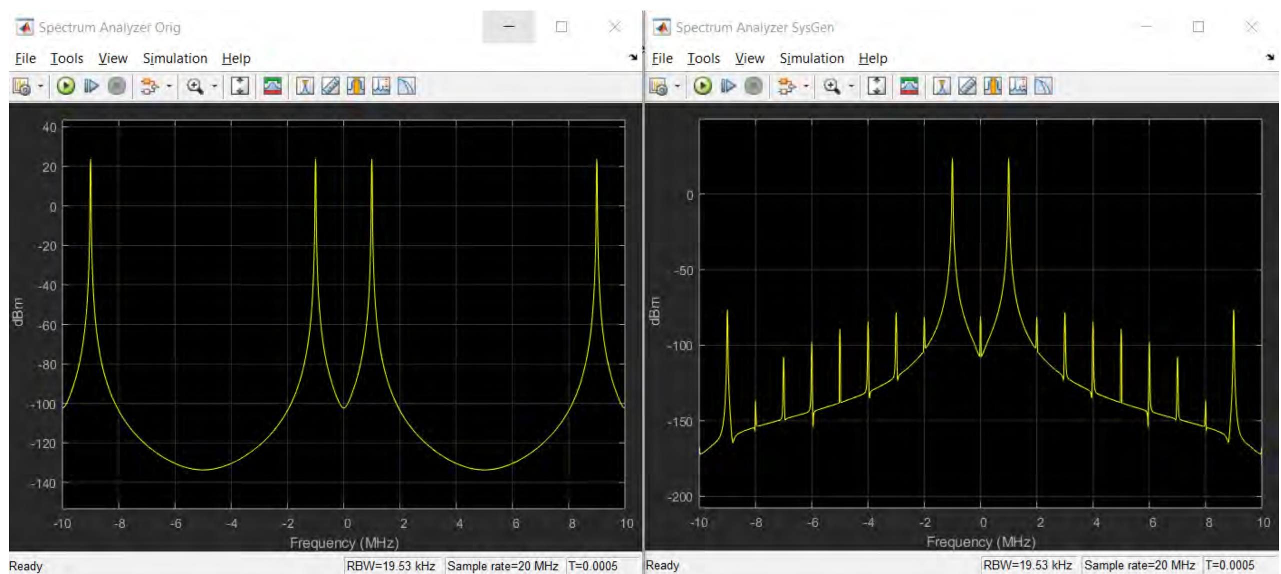
Now, the filter parameters of the FDATool instance can be associated with the Digital FIR filter instance.

Figure 59: **Digital FIR Filter**



The Xilinx Filter response can be viewed and compared with the Simulink® response using the Spectrum Analyzer.

Figure 60: **Spectrum Analyzer**



Note: The frequency response results of System Generator (right side), shown above, differs slightly with the original design (left side) due to the quantization and sampling effect inherent when a continuous time system is described in discrete time hardware.

For complete example along with steps to use the FDATool, refer to the *Vivado Design Suite Tutorial: Model-Based DSP Design Using System Generator* (UG948).

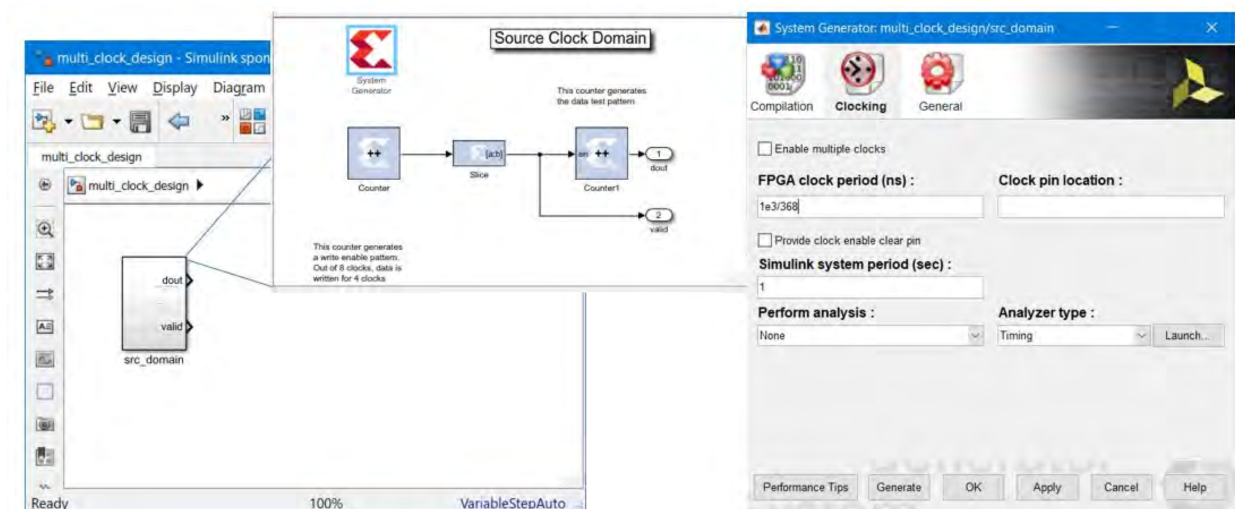
Multiple Independent Clocks Hardware Design

System Generator for DSP is a cycle accurate, high-level hardware modeling and implementation tool where the notion of a cycle is analogous to that of clock in hardware. The design can be partitioned into groups of Subsystem blocks, where each Subsystem has a common cycle period, independent of the cycle period of other Subsystems. This section details how blocks can be grouped into one cycle or clock domain and how data can be transferred between these cycle domains. In the rest of this section, the terms cycle and clock are used interchangeably.

Grouping Blocks within a Clock Domain

Blocks are grouped together in System Generator by using a Subsystem. Grouping blocks within a clock domain is no different except that a System Generator token has to be placed in the Subsystem you want to “mark” as a Clock Domain. This is shown in the figure below.

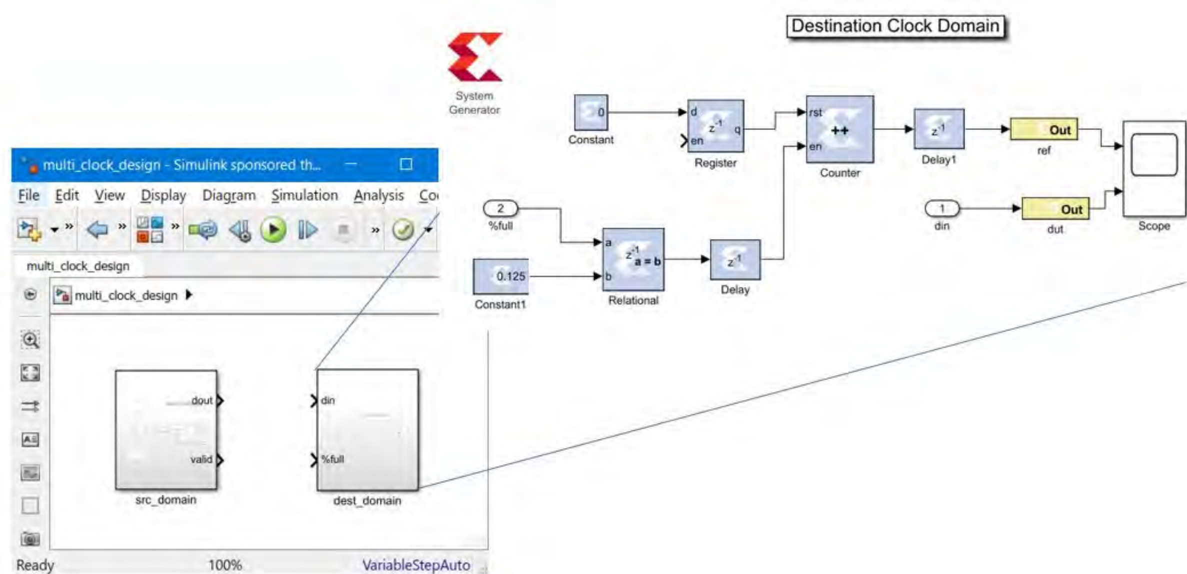
Figure 61: **Source Clock Domain**



In this figure, a clock domain Subsystem called `src_domain` has been created and a System Generator token added. Notice that the clocking tab of the System Generator token is selected. In this tab, the FPGA clock period has been set to (1000/368) ns (368 MHz) and the Simulink system period to 1. This implies that an advance of 1 Simulink second corresponds to (1000/368) ns of FPGA clock.

Similarly, another group of blocks representing another clock domain is included in a Subsystem called `dest_domain`, as shown in the figure below.

Figure 62: **Destination Clock Domain**



In this design, the `dest_domain` Subsystem is configured to run at an FPGA clock period of 1000/245 ns(245 MHz). The Simulink system period is set to 368/245. This is done because the Simulink system period of the `src_domain` Subsystem is set to 1. Hence, you normalize with the System period from the `src_domain` which is faster.

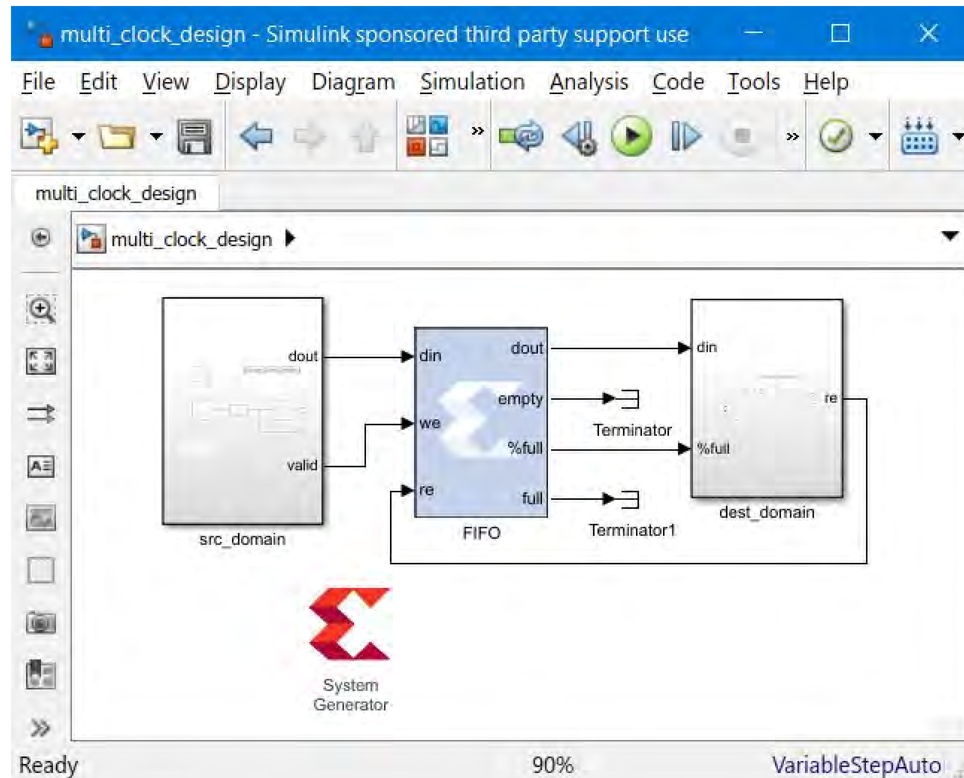
System Generator Blocks used to Create Asynchronous Clock Domains

To pass data between the `src_domain` and `dest_domain` Subsystems, you can use any one of the following logics:

1. FIFO block
2. Dual Port RAM block
3. Register block
4. Black Box block, which allows existing VHDL, Verilog, and EDIF to be brought into a design. For more information about Black Box utility, please refer to Chapter 6: Importing HDL Modules.

These blocks configure themselves to be either synchronous single clock blocks or multiple clock blocks based on their context in the design. In this design, the FIFO block is used to cross the clock domains as shown in the figure below.

Figure 63: **Cross Domain FIFO Block**

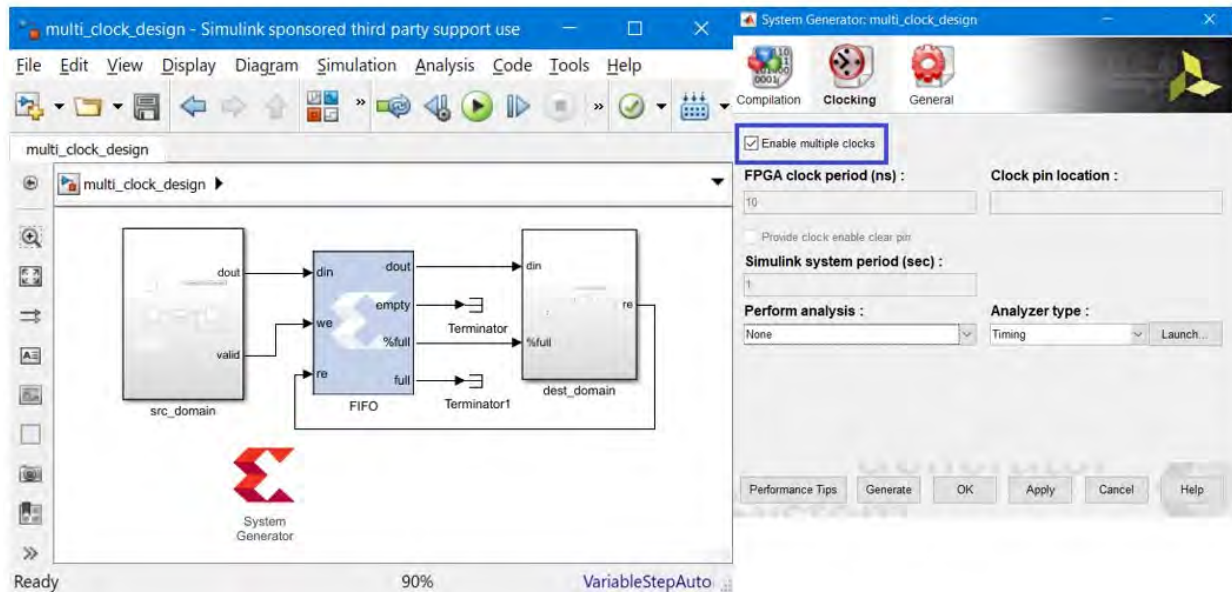


To complete the design, the FIFO block and an additional System Generator block at the top level of the design is included to enable Code Generation.

Configuring the Top-Level System Generator Token

The top-level System Generator token has to be configured to indicate that the Code Generation must proceed for a multiple clock design. This is indicated by turning on the **Enable multiple clocks** check box in the top-level System Generator token. This indicates to the Code Generation engine that the clock information for the Subsystems `src_domain` and `dest_domain` must be obtained from the System Generator tokens contained in those Subsystems. If this check box is not enabled, then the design will be treated as a single clock design where all the clock information is inherited from the top-level System Generator block.

Figure 64: **Enable Multiple Clocks**



Clock Propagation Algorithm

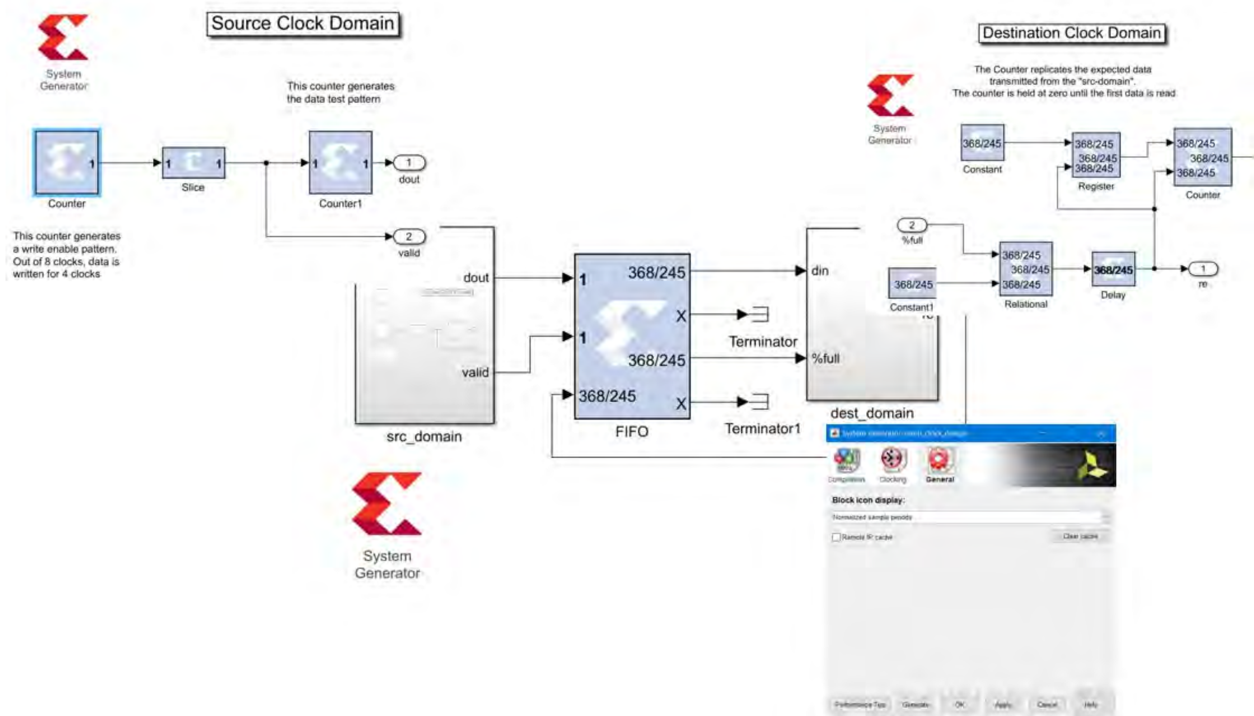
For all System Generator blocks in the `src_domain`, the clocking is governed by the System Generator token in the `src_domain` Subsystem. Similarly for the `dest_domain` Subsystem. For the FIFO block, the clocks are derived from its context in the design. Because the `we` and `din` ports are driven by signals emanating from the `src_domain` Subsystem, the `wr_clk` of the FIFO is tied to the `src_domain` clock. Because the `dout`, `full`, and `re` ports either drive or load signals from `dest_domain`, the `rd_clk` of the FIFO is tied to the `dest_domain` clock. Mixing and matching these signals across clock domains or using any other block (other than FIFO or Dual Port RAM) to cross clock domains will result in a DRC error.

Debugging Clock Propagation

The top-level System Generator token can be used to control the display of all System Generator Block Icons using the **Block icon display** control in the General Tab. From this tab, you can either select **Normalized sample periods** or **Sample frequencies** to help understand how clocks get propagated in the design.

For multiple clock designs, the behavior of **Normalized sample periods**, is that the smallest **Simulink system period** is used to normalize all the sample periods in the design.

Figure 65: **Debugging Clock Propagation**



To enable the above display, set the **Block icon display** of the top-level System Generator token to **Normalized Sample Periods** and press **Apply**.

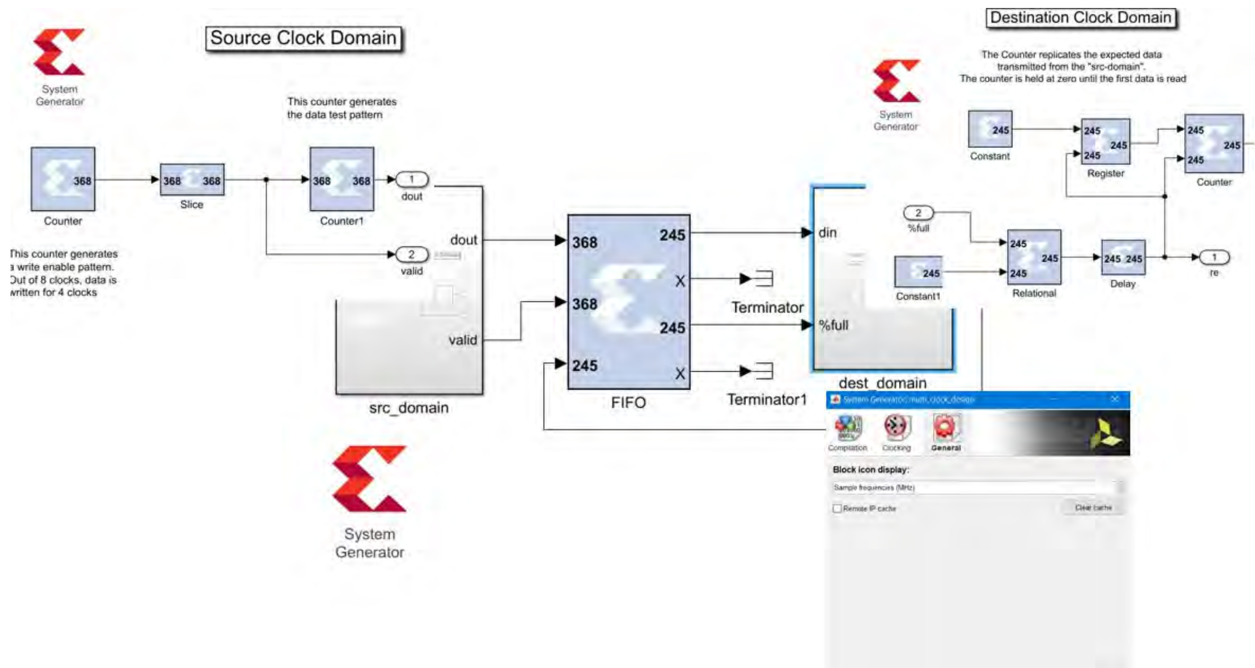
For **Sample Frequencies**, the port icon text display is the result of the following computation:

(1e6/FPGA clock period) * Simulink system period/Port sample period

where FPGA clock period is the FPGA clock period specified in ns in the domain's System Generator token, and Simulink system period is the Simulink system period in seconds specified in the domain's System Generator token.

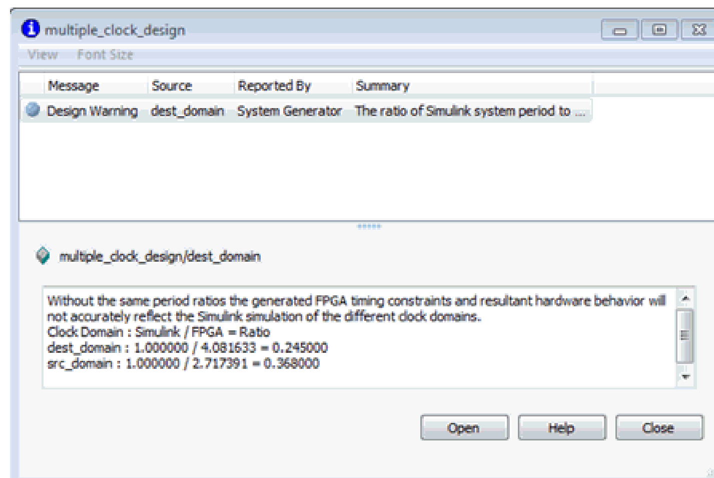
The **Sample Frequencies** can also be used to validate correct clock propagation as shown in the following figure:

Figure 66: **Sample Frequencies**



To ensure that the simulation models the hardware behavior relatively with respect to the clocks, the ratio of Simulink system period to FPGA clock period in each domain must be the same. If this relationship is not complied with the correct ratio, a warning is thrown to indicate this problem as shown in the figure below:

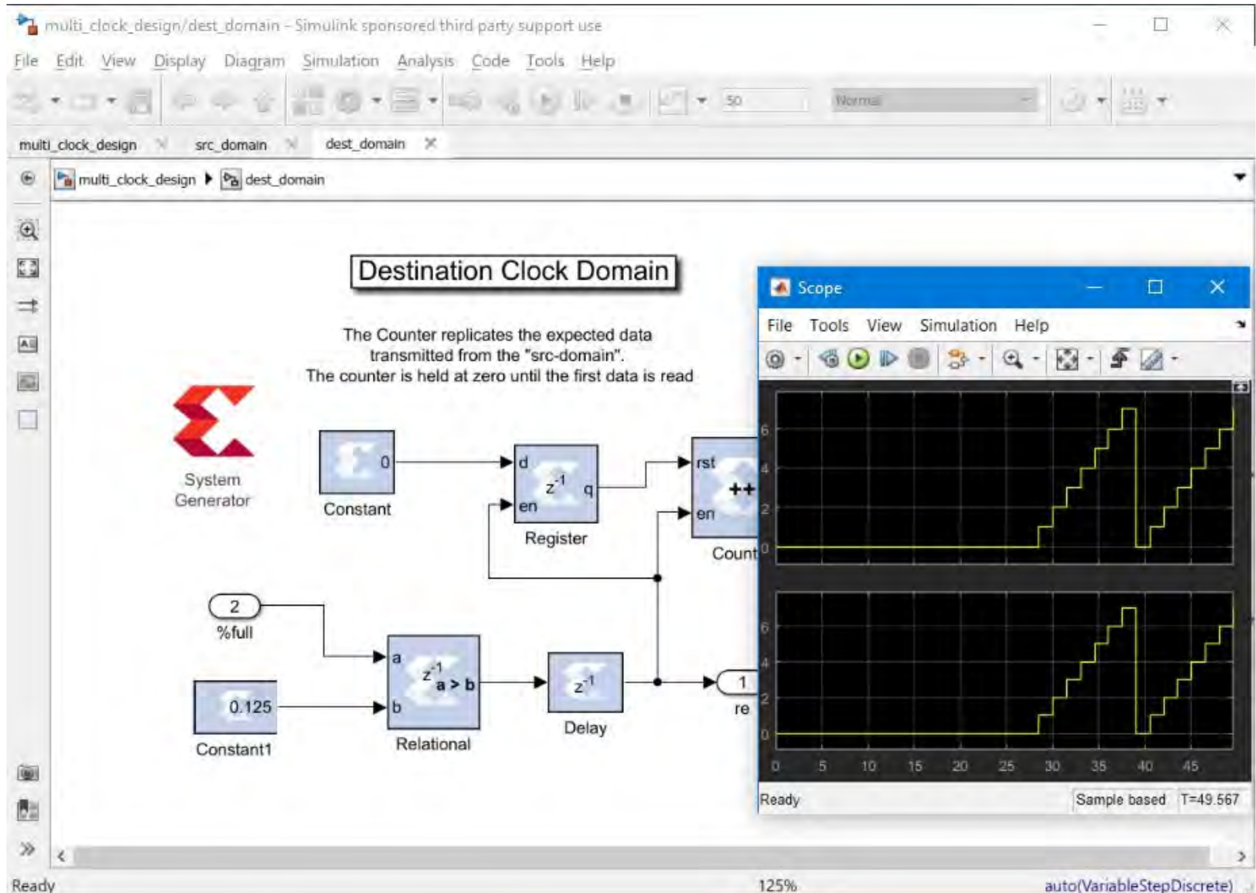
Figure 67: **Warning**



Simulation

After performing the simulation, the following results are obtained as seen in the `dest_domain` scope.

Figure 68: **Simulation**



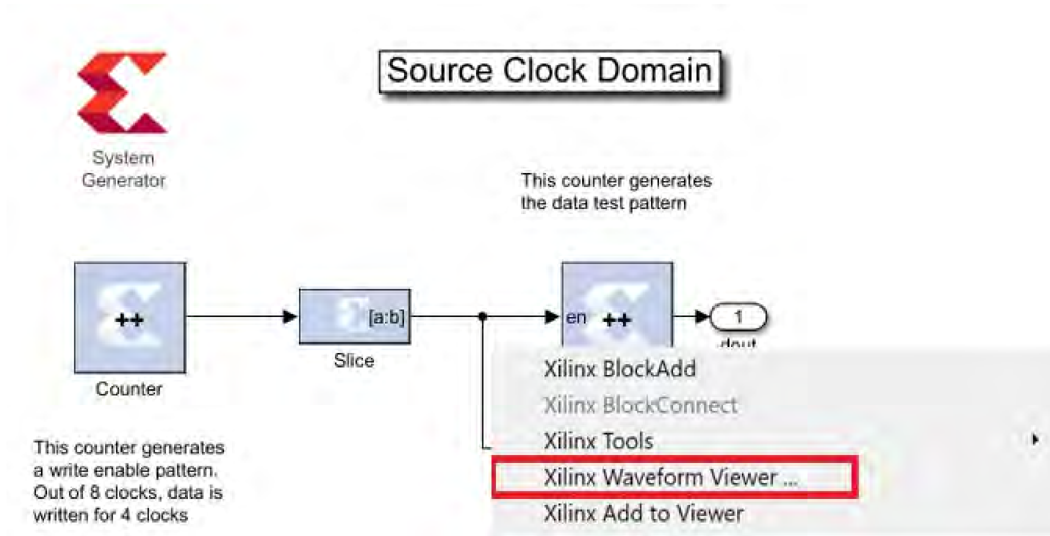
As shown above, the simulation results indicate that the data obtained is the data expected.

***Note*:** This cross-clock domain simulation behavior is NOT cycle accurate.

Debugging Multiple Clock Domain Signals

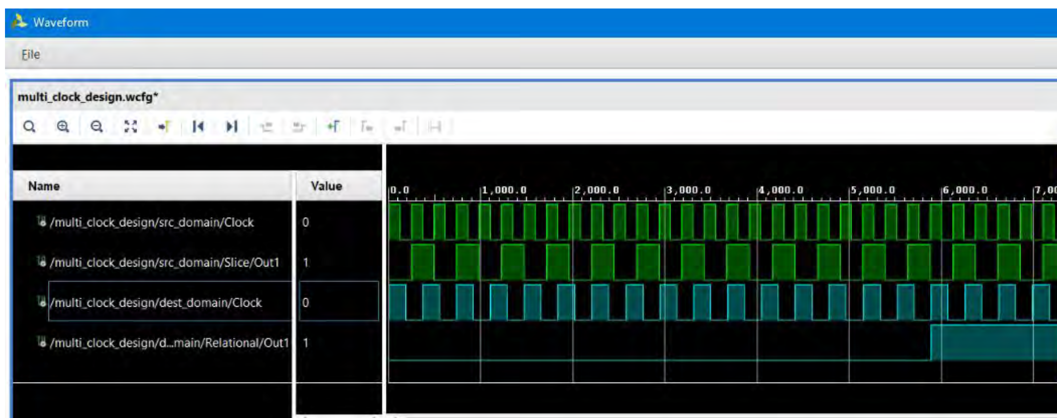
In System Generator, you can use cross probing between the signal in the Xilinx Waveform Viewer and the Simulink® diagram to aid the debugging process.

Figure 69: **Source Clock Domain**



To add a signal to the Waveform viewer, right-click the signal in the model and select **Xilinx Add To Viewer**. Simulating the design should launch the Waveform Viewer as shown below.

Figure 70: **Waveform Viewer**



All signals in same clock domain are colored similarly. In the figure above: `src_domain/Slice/Out1` and `dest_domain/Relational/Out1` are in different clock domains.

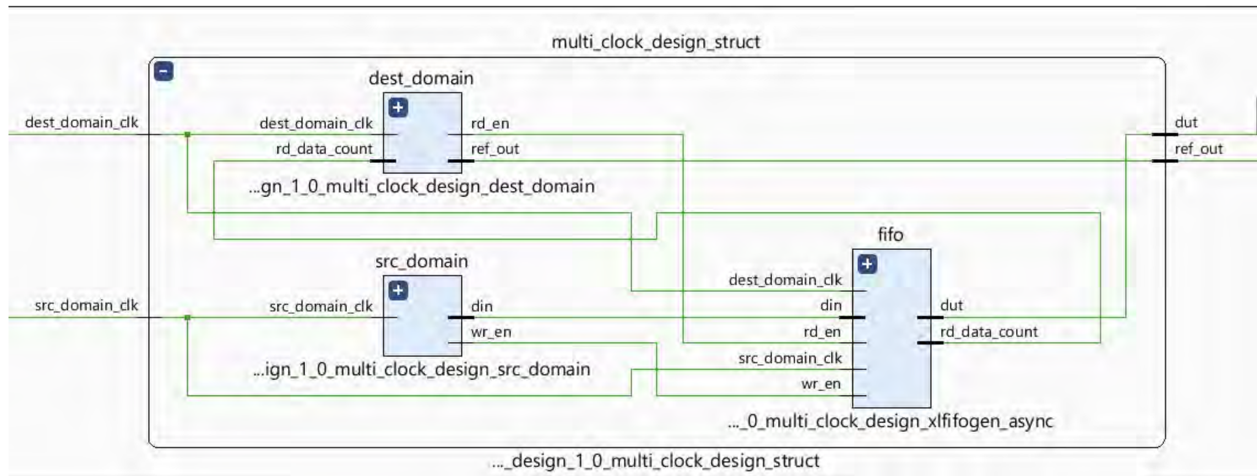
Code Generation

Code generation for a Multiple Clock design supports the following compilation targets:

- **HDL Netlist**
- **IP Catalog**
- **Synthesized Checkpoint**

A screen shot of the top-level hardware is shown in the figure below.

Figure 71: **Top-Level Hardware**



As many clock ports as there are clock domains are exposed at the top level and can be driven by a variety of Xilinx clocking constructs like MMCM, PLL etc. It is assumed that these clocks are completely asynchronous and the following period constraints are created:

```
1 create_clock -name src_domain_clk -period 2.717 [get_ports src_domain_clk]
2 create_clock -name dest_domain_clk -period 4.082 [get_ports dest_domain_clk]
```

These are the only constraints that are required because only FIFO or Dual Port RAM are allowed which have any additional clock domain constraints embedded in the IP.

Known Issues

The following are some of the known issues:

- The HWCosim Compilation Target is not supported for Multiple Clock Designs.
- Only FIFO & Dual Port RAM blocks can be in the top-level of the design when using multiple clocks.
- The behavior of blocks that aid in the crossing of Multiple clock domains is NOT cycle accurate.
- Unconnected or terminated output ports cannot be viewed in the Waveform Viewer.

AXI Interface

AMBA AXI4 (Advanced eXtensible Interface 4) is the fourth generation of the AMBA interface defined and controlled by Arm®, and has been adopted by Xilinx as the next-generation interconnect for FPGA designs. Xilinx and Arm worked closely to ensure that the AXI4 specification addresses the needs of FPGAs.

AXI is an open interface standard that is widely used by many 3rd-party IP vendors since it is public, royalty-free and an industry standard.

The AMBA AXI4 interface connections are point-to-point and come in three different flavors: AXI4, AXI4-Lite Slave, and AXI4-Stream.

- AXI4 is a memory-mapped interface which support burst transactions
- AXI4-Lite Slave is a lightweight version of AXI4 and has a non-bursting interface
- AXI4-Stream is a high-performance streaming interface for unidirectional data transfers (from master to slave) with reduced signaling requirements (compared to AXI4). AXI4-Stream supports multiple channels of data on the same set of wires.

In the following documentation, AXI4 refers to the AXI4 memory map interface, and AXI4-Lite Slave and AXI4-Stream each refer to their respective flavor of the AMBA AXI4 interface. When referring to the collection of interfaces, the term AMBA AXI4 shall be used.

The purpose of this section is to provide an introduction to AMBA AXI4 and to draw attention to AMBA AXI4 details with respect to System Generator. For more detailed information on the AMBA AXI4 specification please refer to the Xilinx AMBA-AXI4 documents found on the AMBA AXI4 Interface Protocol page on the Xilinx website.

AXI4-Stream Support in System Generator

The three most common AXI4-Stream signals are TVALID, TREADY and TDATA. Of all the AXI4-Stream signals, only TVALID is denoted as mandatory, all other signals are optional. All information-carrying signals propagate in the same direction as TVALID; only TREADY propagates in the opposite direction.

Since AXI4-Stream is a point-to-point interface, the concept of master and slave interface is pertinent to describe the direction of data flow. A master produces data and a slave consumes data.

Naming Conventions

AXI4-Stream signals are named in the following manner:

```
<Role>_<ClassName>[_<BusName>]_[<ChannelName>]<SignalName>
```

For example:

```
m_axis_tvalid
```

Here `m` denotes the Role (master), `axis` the ClassName (AXI4-Stream) and `tvalid` the SignalName.

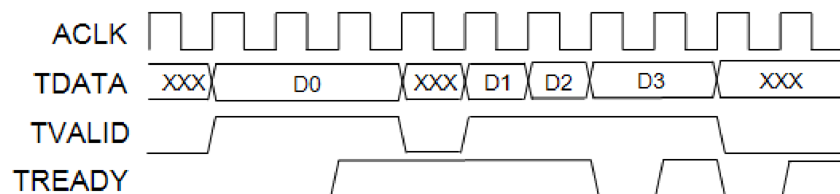
```
s_axis_control_tdata
```

Here `s` denotes the Role (slave), `axis` the ClassName, `control` the BusName which distinguishes between multiple instances of the same class on a particular IP, and `tdata` the SignalName.

Notes on TREADY/TVALID Handshaking

The TREADY/TVALID handshake is a fundamental concept in AXI to control how data is exchanged between the master and slave allowing for bidirectional flow control. TDATA, and all the other AXI4-Stream signals (TSTRB, TUSER, TLAST, TID, and TDEST) are all qualified by the TREADY/TVALID handshake. The master indicates a valid beat of data by the assertion of TVALID and must hold the data beat until TREADY is asserted. TVALID once asserted cannot be de-asserted until TREADY is asserted in response (this behavior is referred to as a “sticky” TVALID). AXI also adds the rule that TREADY can depend on TVALID, but the assertion of TVALID cannot depend on TREADY. This rule prevents circular timing loops. The timing diagram below provides an example of the TREADY/TVALID handshake.

Figure 72: **TREADY/TVALID Handshake Timing Diagram**



Handshaking Key Points

- A transfer on any given channel occurs when both TREADY and TVALID are high in the same cycle.
- TVALID once asserted, may only be de-asserted after a transfer has completed (TREADY is sampled high). Transfers may not be retracted or aborted.
- Once TVALID is asserted, no other signals in the same channel (except TREADY) may change value until the transfer completes (the cycle after TREADY is asserted).
- TREADY may be asserted before, during or after the cycle in which TVALID is asserted.
- The assertion of TVALID may not be dependent on the value of TREADY. But the assertion of TREADY may be dependent on the value of TVALID.

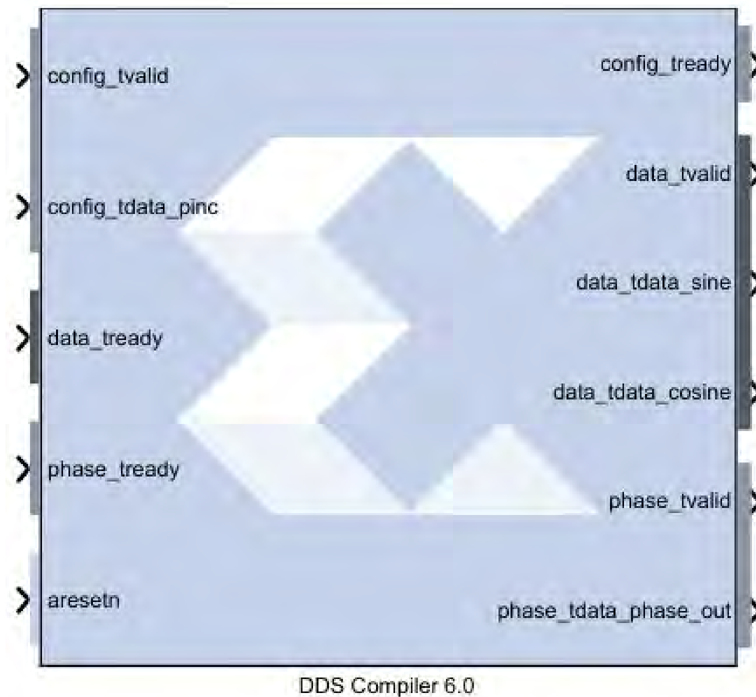
- There must be no combinatorial paths between input and output signals on both master and slave interfaces:
  - Applied to AXI4-Stream IP, this means that the TREADY slave output cannot be combinatorially generated from the TVALID slave input. A slave that can immediately accept data qualified by TVALID, should pre-assert its TREADY signal until data is received. Alternatively TREADY can be registered and driven the cycle following TVALID assertion.
  - The default design convention is that a slave should drive TREADY independently or pre-assert TREADY to minimize latency.
  - Note that combinatorial paths between input and output signals are permitted across separate AXI4-Stream channels. It is however a recommendation that multiple channels belonging to the same interface (related group of channels that operate together) should not have any combinatorial paths between input and output signals.
- For any given channel, all signals propagate from the source (typically master) to the destination (typically slave) except for TREADY. Any other information-carrying or control signals that need to propagate in the opposite direction must either be part of a separate channel ("back-channel" with separate TREADY/TVALID handshake) or be an out-of-band signal (no handshake). TREADY should not be used as a mechanism to transfer opposite direction information from a slave to a master.
- AXI4-Stream allows TREADY to be omitted which defaults its value to 1. This may limit interoperability with IP that generates TREADY. It is possible to connect an AXI4-Stream master with only forward flow control (TVALID only).

AXI4-Stream Blocks in System Generator

System Generator blocks that present an AXI4-Stream interface can be found in the Xilinx Blockset Library entitled AXI4. Blocks in this library are drawn slightly differently from regular (non AXI4-Stream) blocks.

Port Groupings

Figure 73: **DDS Compiler 6.0**



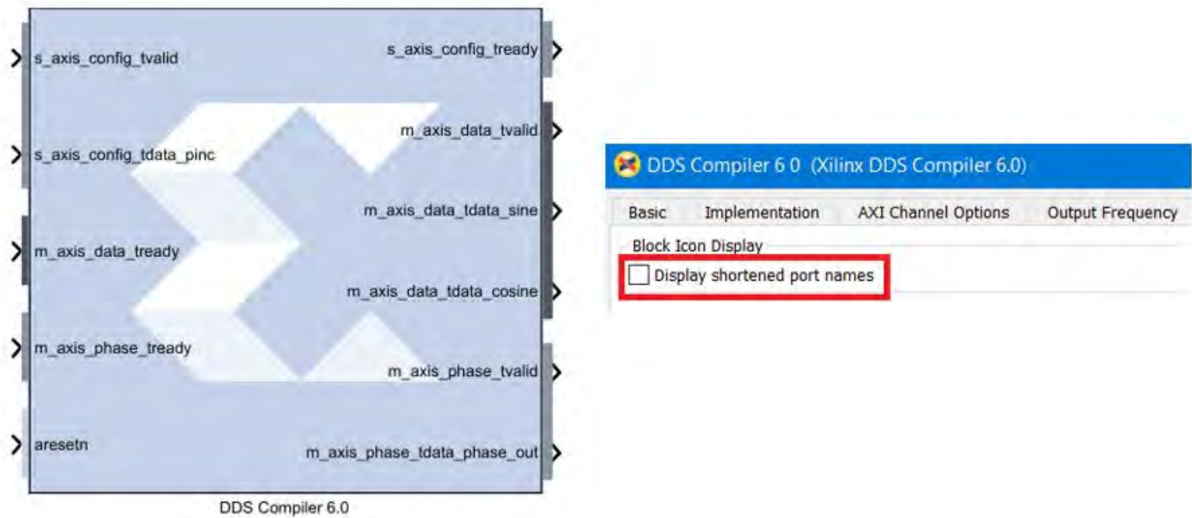
Blocks that offer AXI4-Stream interfaces have AXI4-Stream channels grouped together and color coded. For example, on the DDS compiler 6.0 block shown above, the input port `data_tready`, and the three output ports, `data_tvalid`, `data_tdata_sine`, `data_tdata_cosine` belong in the same AXI4-Stream channel. Similarly, the input port `config_tvalid`, `config_tdata_pinc` and output port `config_tready` belong in the same AXI4-Stream channel. As does `phase_tready`, `phase_tvalid`, and `phase_tdata_phase_out`.

Signals that are not part of any AXI4-Stream channels are given the same background color as the block; `aresetn` is an example.

Port Name Shortening

In the example shown below, the AXI4-Stream signal names have been shortened to improve readability on the block. Name shortening is purely cosmetic and when netlisting occurs, the full AXI4-Stream name is used. Name shorting is turned on by default; you can uncheck the **Display shortened port names** option in the block parameter dialog box to reveal the full name.

Figure 74: **DDS Compiler 6.0**



Breaking Out Multi-Channel TDATA

In AXI4-Stream, TDATA can contain multiple channels of data. In System Generator, the individual channels for TDATA are broken out. So for example, the TDATA of port `dout` below contains both real and imaginary components.

Figure 75: **Complex Multiplier 6.0**

The breaking out of multi-channel TDATA does not add additional logic to the design and is done in System Generator as a convenience to the users. The data in each broken out TDATA port is also correctly byte-aligned.

AXI4-Lite Slave Interface Generation

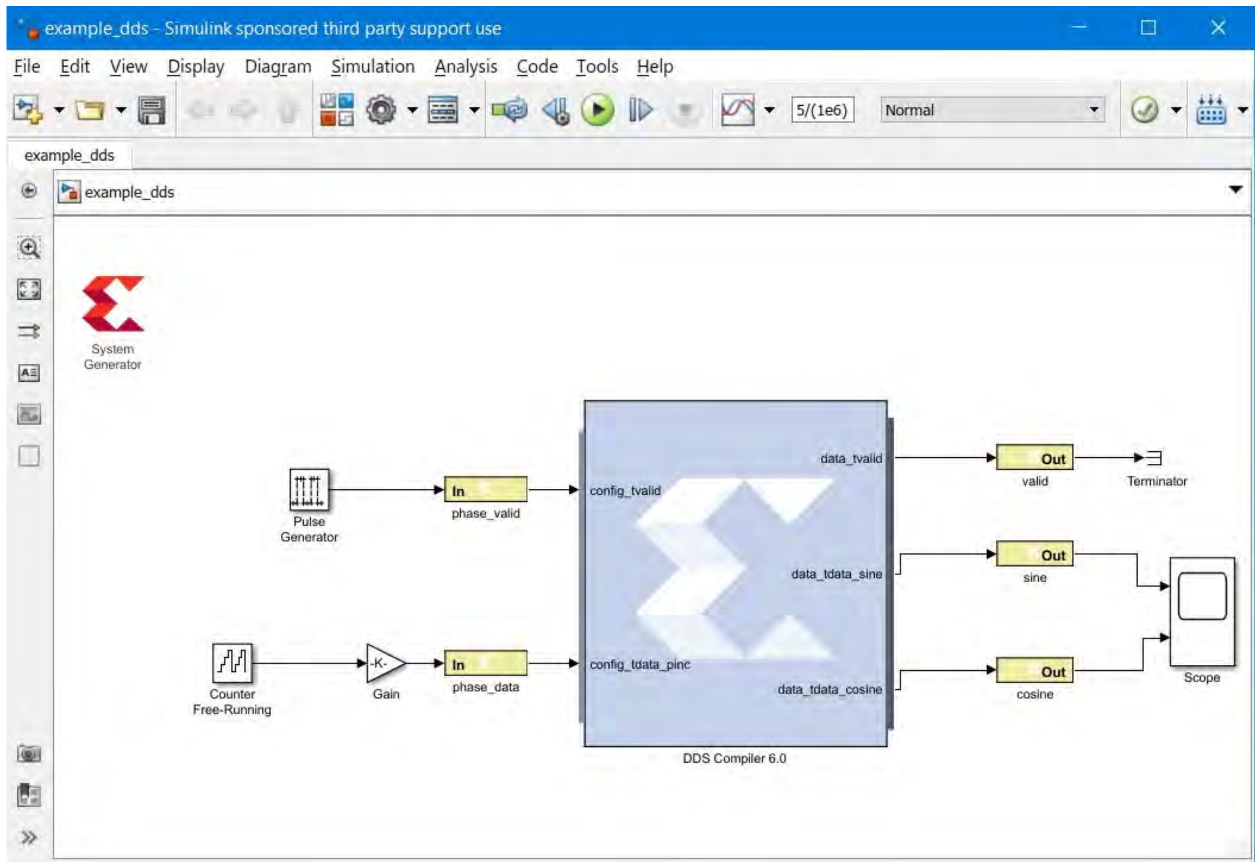
Design modules that are developed using System Generator usually form a Subsystem of a larger DSP or Video system. These System Generator modules are typically algorithmic and data path heavy modules that are best created in the visually-rich environment like MATLAB/Simulink. The larger system is typically assembled from IP from the Vivado® IP catalog. These IP typically use standard stream and control interfaces like AXI4-Lite Slave and the larger system is typically assembled using a tool like the Vivado IP integrator.

This topic describes features in System Generator that allow you to create a standard AXI4-Lite Slave interface for a System Generator module and then export the module to the Vivado® IP catalog for later inclusion in a larger design using IP integrator. System Generator also allows creation of multiple AXI4-Lite Slave interfaces across multiple clock domains.

AXI4-Lite Interface Synthesis in System Generator

Design creation and verification is exactly the same as any other System Generator design that does not include an AXI4-Lite interface. Consider the `example_dds` design shown below.

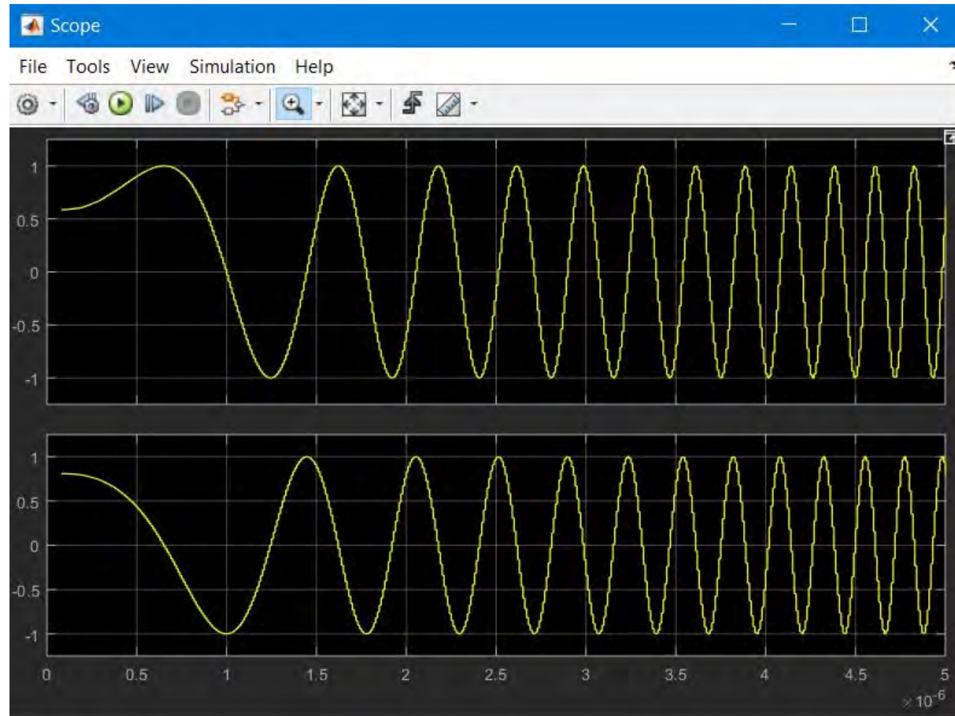
Figure 76: **Example DDS Design**



This design contains a DDS Compiler where the two input ports, `config_tvalid` and `config_tdata_pinc` are used to control the output frequency.

Following are the simulation results of this design which indicate both sine and cosine output separately.

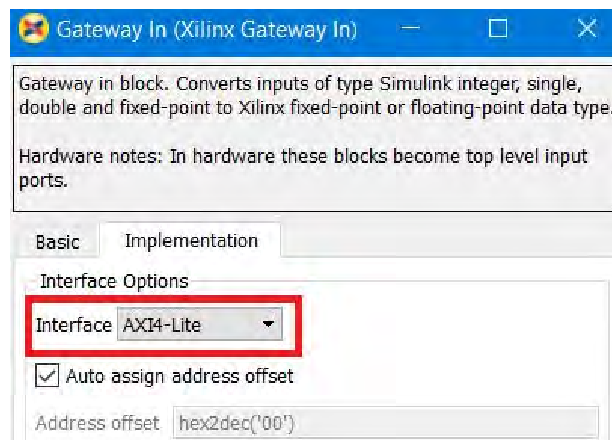
Figure 77: **Simulation Results**



Configuring the Design for an AXI4-Lite Interface

In the example_dds design, Gateway In and Gateway Out blocks mark the boundary of the Cycle and Bit accurate FPGA portion of the Simulink design. Control of the DDS Compiler frequency is accomplished by "injecting" the correct value on the signals attached to the output port of Gateway In's called `phase_valid` and `phase_data`. This is accomplished by modifying the Interface Options, as shown below for the `phase_valid` block.

Figure 78: **Interface Options**



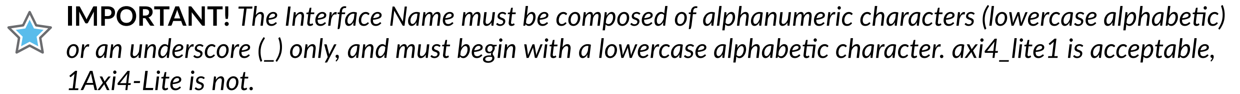
As you can see, the Interface is specified as a slave AXI4-Lite Interface in System Generator, which means that it will be transformed to a top-level AXI4-Lite interface.

The following options are also of particular interest:

Auto assign address offset (Enabled): Each Gateway is associated with a register within the AXI4-Lite Interface and this control specifies that Automatic assignment of address offsets will take place in the design based on number of different Gateway Ins mapped to the AXI4-Lite interface. Addresses are byte aligned to a 32-bit data width.

Address offset (Disabled): This option is only enabled if **Auto assign address offset** is unchecked. It allows the user to manually override of Address Offset.

Interface Name: Assigns a unique name to this interface. This name can be used to differentiate between multiple AXI4-Lite interfaces in the design.



IMPORTANT! *The Interface Name must be composed of alphanumeric characters (lowercase alphabetic) or an underscore (_) only, and must begin with a lowercase alphabetic character. axi4_lite1 is acceptable, 1Axi4-Lite is not.*

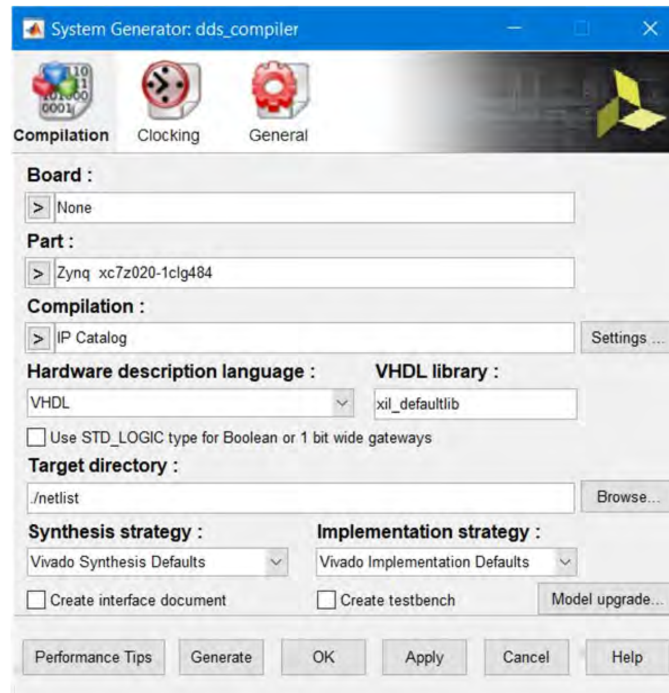
Description: The text you enter here is captured in the "Interface Documentation" that is automatically created when the design is exported to the Vivado IP catalog.

Configure the other Gateways in the design in a similar fashion.

Packaging the Design for Use in Vivado IP Integrator

When you complete the verification in System Generator, you can package the design for use in IP integrator.

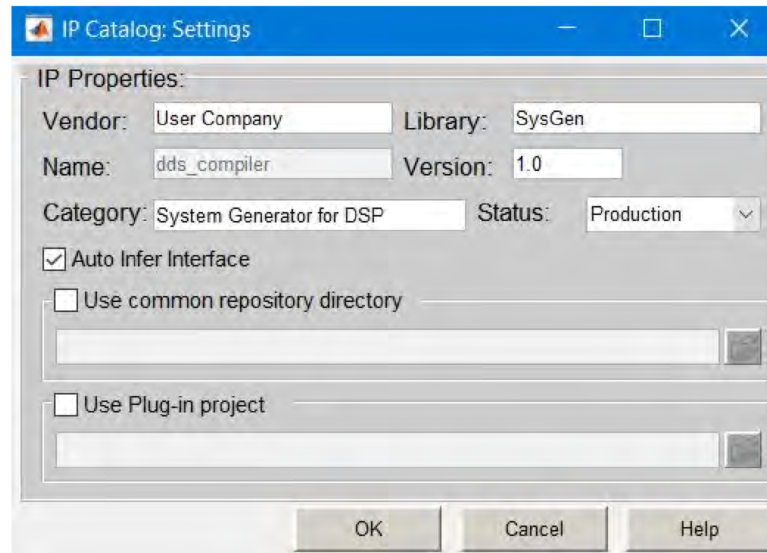
Figure 79: **System Generator Verification**



The System Generator block must first be configured to a Compilation target of **IP Catalog** (the default generation target). This compilation target will consolidate all hardware source created from System Generator (RTL + IP + Constraints) into an IP.

The part selected is the same part as that available on the Xilinx Zynq-7000 ZC702 Evaluation Board. In addition, you may also use the **Settings** button on the System Generator token to change the information that goes along with the IP. In this case, the default values shown below are used.

Figure 80: **IP Catalog Settings**



When you click the **Generate** button in System Generator token GUI, RTL+IP+Constraints generation, as well as packaging takes place.

Description of the Generated Results

Based on the System Generator settings shown above, the following folders and files are created.

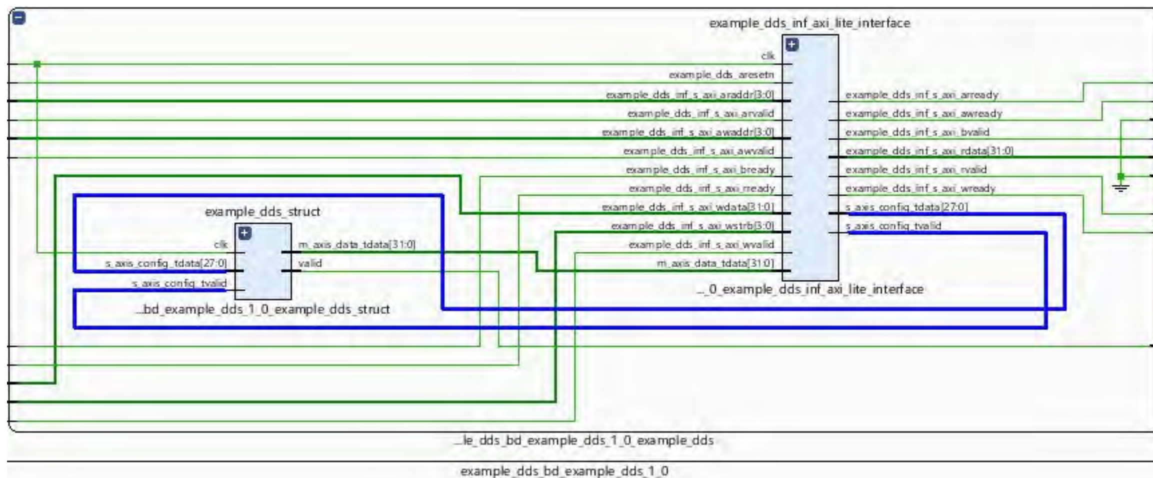
- `<target directory>/ip`: This directory contains all the IP-related hardware files, as well as the software drivers. It is this directory that you must add to the IP Catalog.
- `<target directory>/ip_catalog`: This directory contains an example Vivado IDE project called `example_dds.xpr`.

Mapping to AXI4-Lite Interfaces

Gateway Ins and Gateway Outs that are tagged as AXI4-Lite registers are mapped to different 32-bit registers within a Memory Map as shown in the Schematic below.

The schematic below is an example of mapping to a single AXI4-Lite interface, assuming all gateways have the same interface name. In a schematic with multiple AXI4-Lite interfaces, for each group of gateways having the same interface name you would see a separate AXI4-Lite Interface.

Figure 81: **Single AXI4-Lite Interface**



As you can see in the diagram, a module called `example_dds_inf_axi_lite_interface` is inserted into the design RTL, and drives the `config_tvalid` and `config_tdata` ports of the System Generator design. And at the top level, a slave AXI4-Lite Interface is exposed. It is within this module that address decoding is done and the `config_tvalid` and `config_tdata` ports are driven based on the address obtained from the processor.

The number of bits required for addressing (`s_axi_araddr` and `s_axi_awaddr`) is determined by the number of AXI4-Lite interface registers and the offset specifications of each AXI4-Lite register. Enough bits are provided during module generation to uniquely decode each register. In this example, there are two Gateways – `phase_data` and `phase_valid`. Each port is assigned an address offset of 0x0000 & 0x0004. Hence three address bits are allocated.

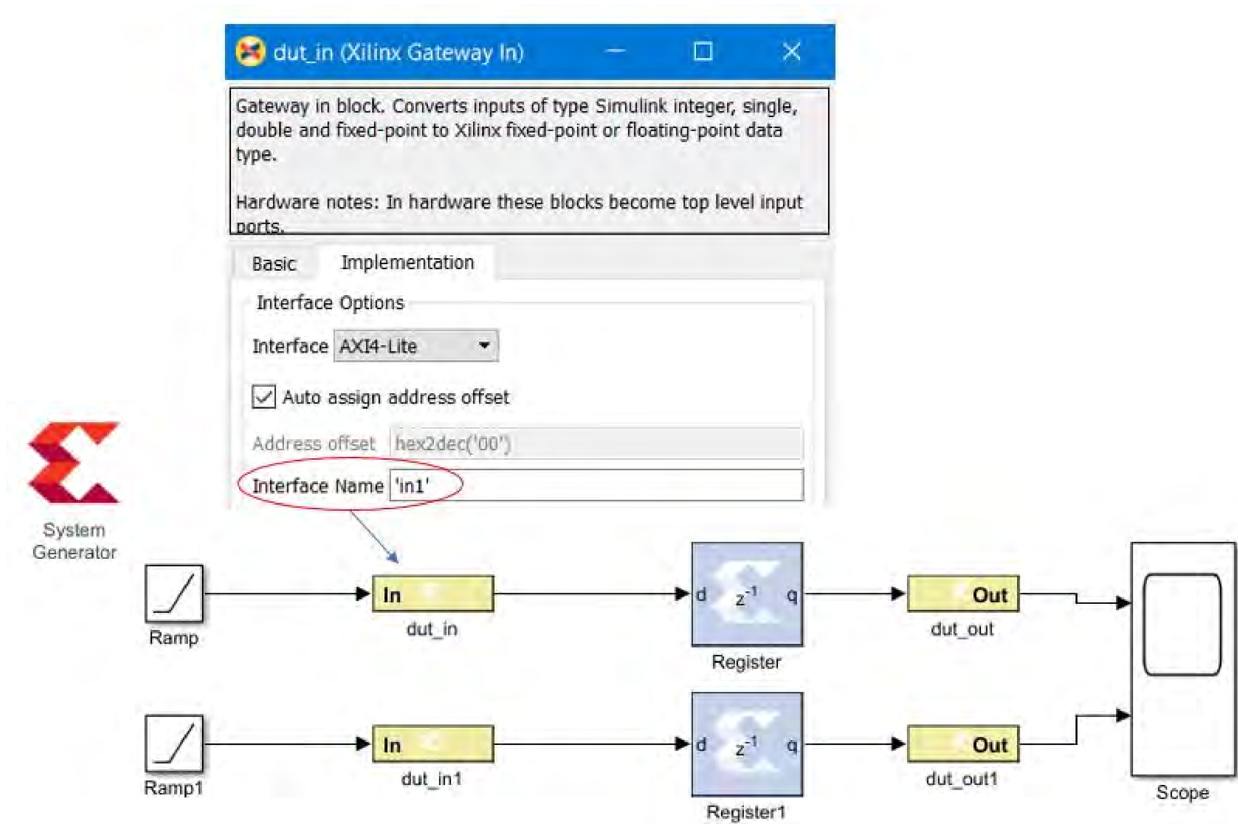
Managing Multiple AXI4-Lite Interfaces

System Generator supports creation of IP with multiple AXI4-Lite interfaces. You can group Gateway In and Gateway Out blocks into different AXI4-Lite interfaces. This feature can be used in Multiple Clock designs as well. Software drivers will also be provided.

To assign a name to an AXI4-Lite interface, use the **Interface Name** control for the Gateway In and Gateway Out blocks associated with the interface.

All Gateway Ins and Gateway Outs with the same **Interface Name** are grouped into one AXI4-Lite Interface. An **Interface Name** must begin with a lower case alphabetic character, and can only contain alphanumeric characters (lowercase alphabetic) or an underscore (_). Having the same **Interface Name** across multiple clock domains is not supported.

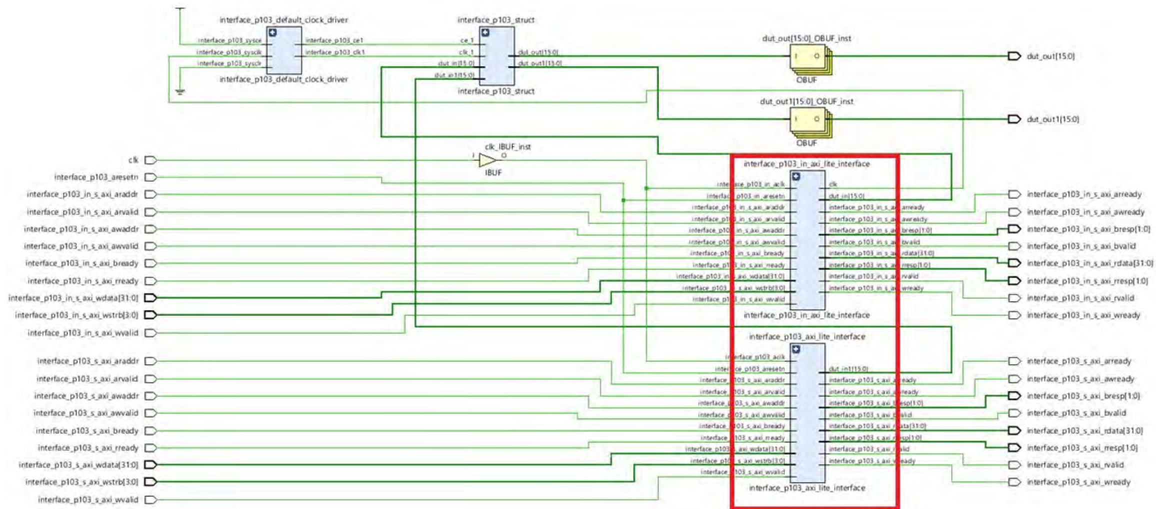
Figure 82: **Interface Name**



To generate the netlist you can use the **IP Catalog** or the **HDL Netlist** compilation target.

If you specify the **HDL Netlist** compilation target in the System Generator token, and then elaborate the design in Vivado, two AXI4-Lite Decoders will be created, as shown in red rectangle in the following figure.

Figure 83: **AXI4-Lite Decoders**

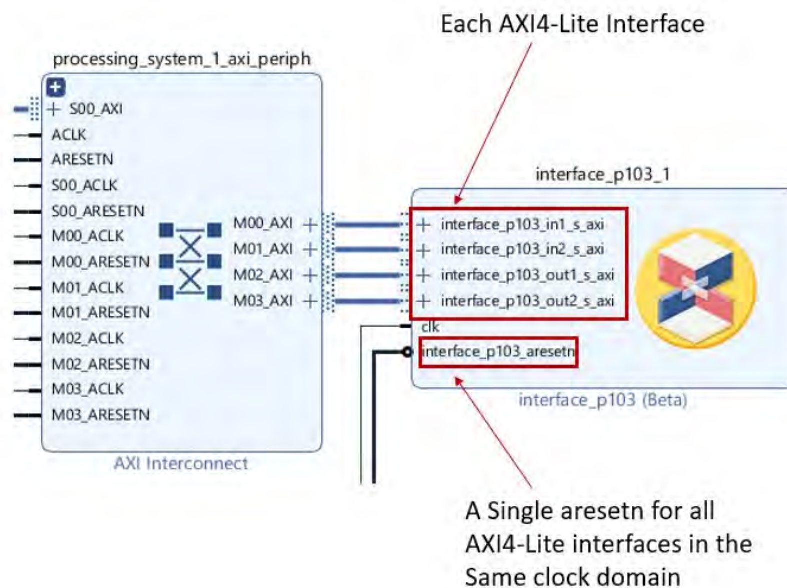


If you specify the **IP Catalog** compilation target in the System Generator token, the flow will also generate an example BD with multiple AXI4-Lite interfaces and an `aresetn` signal.

The naming convention for an interface is:

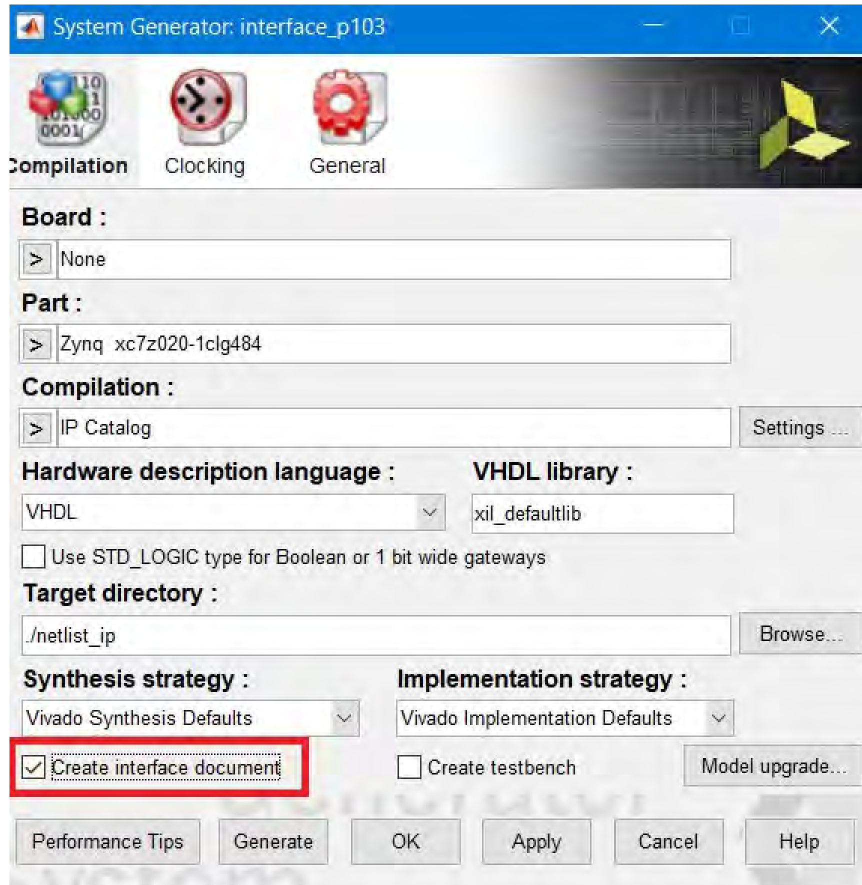
```
<clock domain name/design name>_<interface name>_s_axi
```

Figure 84: **Example BD**



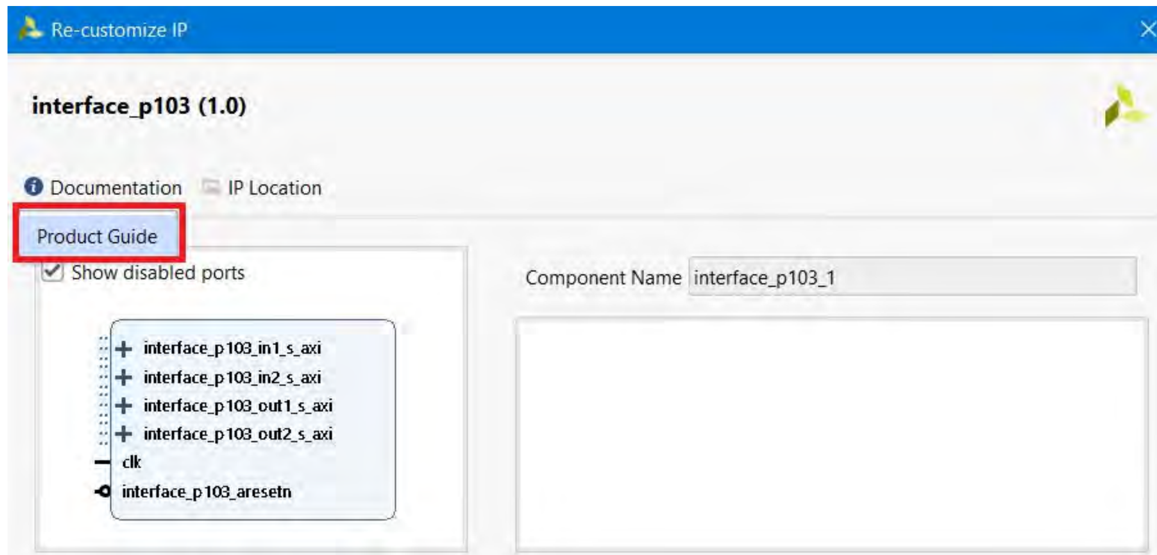
To generate a document describing the IP, select the **Create interface document** option on the System Generator Token **Compilation** tab before you perform the compilation.

Figure 85: **Create Interface Document**



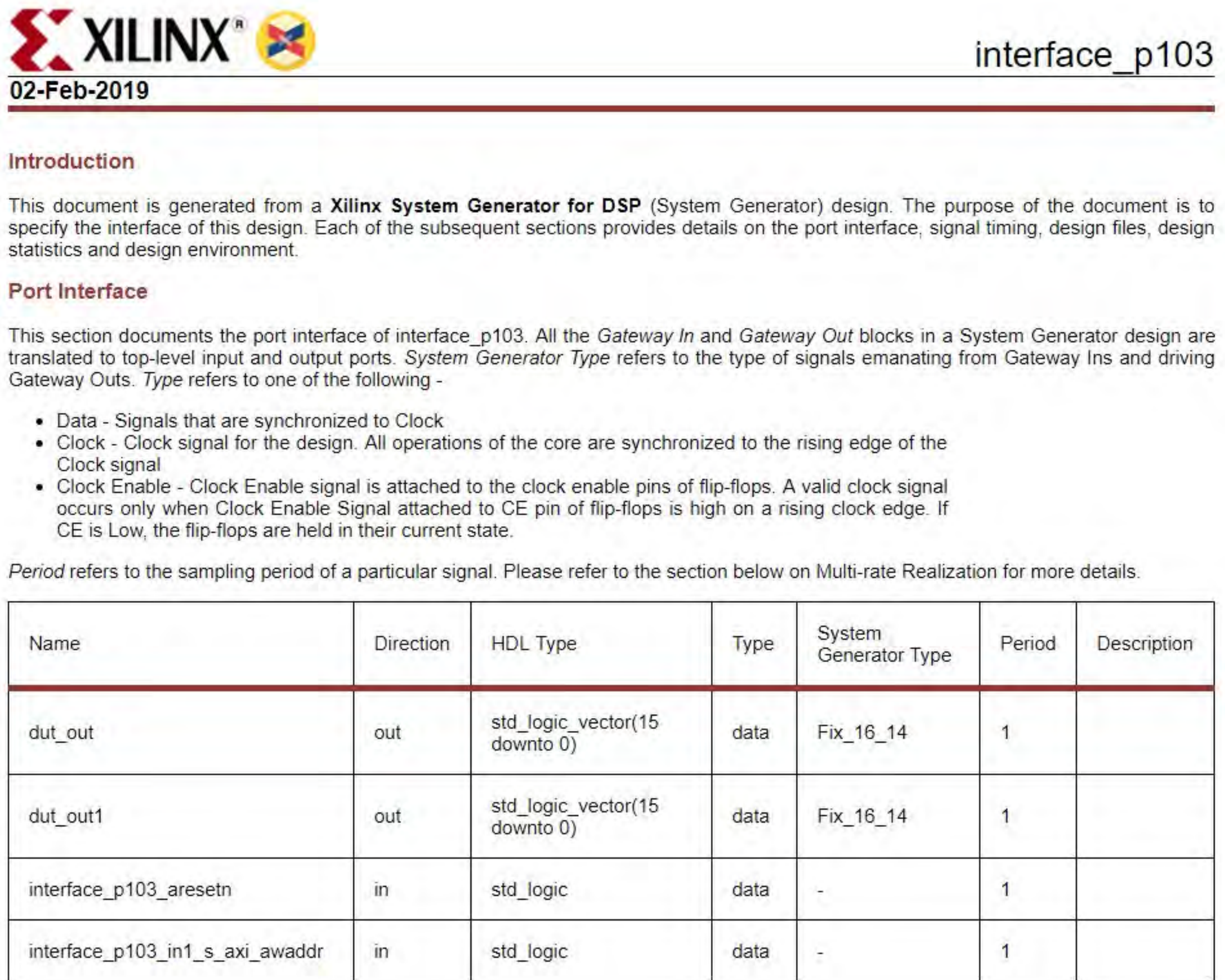
Access the document the same way you access the document for any other Vivado IP. Double-click the IP in the Vivado schematic, then select **Documentation → Product Guide**.

Figure 86: **Accessing Documents**



A document (HTML file) will open up (see example below).

Figure 87: **Sample Document**



XILINX

interface_p103

02-Feb-2019

Introduction

This document is generated from a **Xilinx System Generator for DSP** (System Generator) design. The purpose of the document is to specify the interface of this design. Each of the subsequent sections provides details on the port interface, signal timing, design files, design statistics and design environment.

Port Interface

This section documents the port interface of interface_p103. All the *Gateway In* and *Gateway Out* blocks in a System Generator design are translated to top-level input and output ports. *System Generator Type* refers to the type of signals emanating from Gateway Ins and driving Gateway Outs. *Type* refers to one of the following -

- Data - Signals that are synchronized to Clock
- Clock - Clock signal for the design. All operations of the core are synchronized to the rising edge of the Clock signal
- Clock Enable - Clock Enable signal is attached to the clock enable pins of flip-flops. A valid clock signal occurs only when Clock Enable Signal attached to CE pin of flip-flops is high on a rising clock edge. If CE is Low, the flip-flops are held in their current state.

Period refers to the sampling period of a particular signal. Please refer to the section below on Multi-rate Realization for more details.

Name	Direction	HDL Type	Type	System Generator Type	Period	Description
dut_out	out	std_logic_vector(15 downto 0)	data	Fix_16_14	1	
dut_out1	out	std_logic_vector(15 downto 0)	data	Fix_16_14	1	
interface_p103_aresetn	in	std_logic	data	-	1	
interface_p103_in1_s_axi_awaddr	in	std_logic	data	-	1	

This document contains a section on the Memory Map for the IP. If you selected **Auto assign address offset** in the Gateway In or Gateway Out port for the AXI4-Lite interfaces, you can find out the address offset the different interfaces are mapped to.

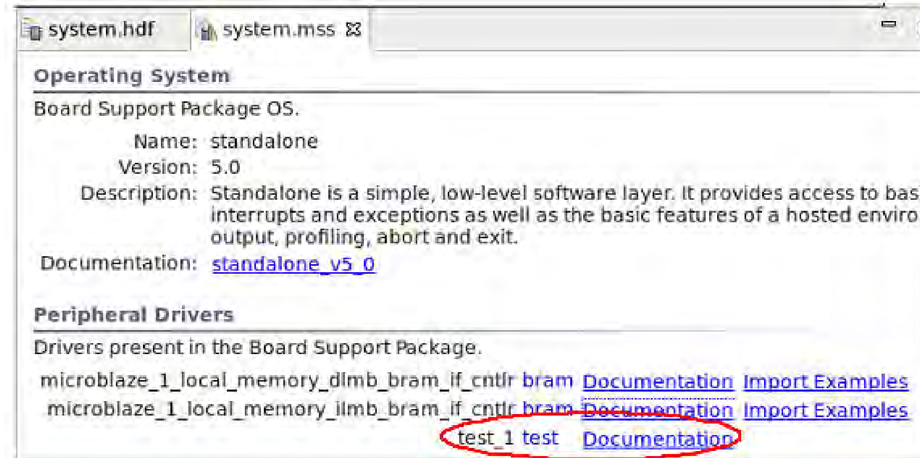
Memory Map

The table below documents the memory map for System Generator design :

Name	Type	Interface Name	Address Offset	Description
dut_in1	Fix_32_2	in	00000000	
dut_in	Fix_32_2	in	00000004	
dut_out	Fix_32_2	out	00000000	
dut_out1	Fix_32_2	out	00000004	

Software Drivers are automatically generated and packaged as well in the Vitis™ software plaform. The documentation for the software drivers can be found in the Vitis environment.

Figure 88: **Software Driver Documentation**



Address Generation

The following assumptions are made in the automatic address-generation process:

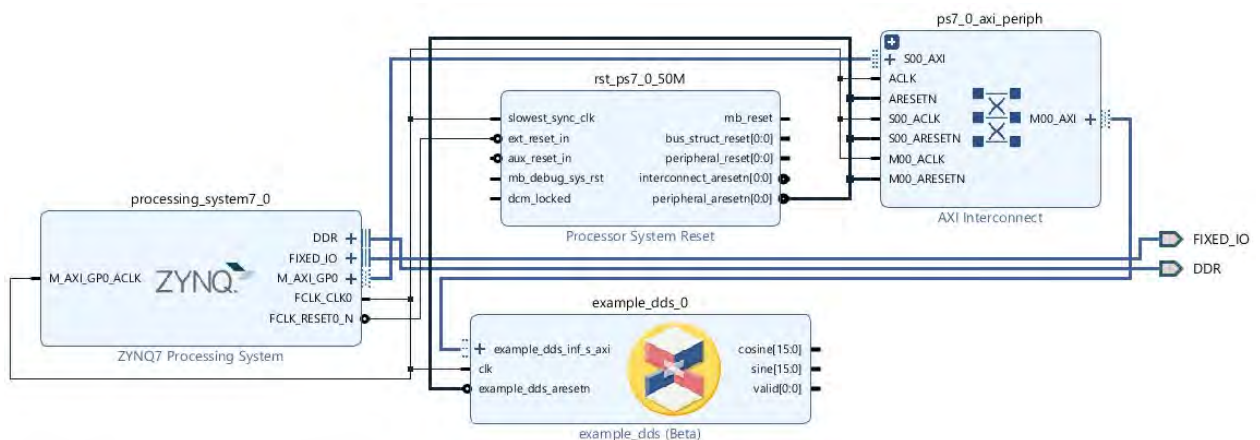
1. Each AXI4-Lite gateway is associated with a unique address offset that is aligned with a 32-bit word boundary (i.e., will be a multiple of 4).
2. Addressing begins at zero.
3. Addressing is incrementally assigned in the lexicographical order of the gateways. In the event two gateways have the same name - disambiguation will be arbitrary.
4. All AXI4-Lite gateways must be less than 32-bits wide else an error is issued.
5. If an AXI4-Lite gateway is less than 32-bits wide, then from the internal register, LSBs will be assigned into the Design Under Test (DUT).
6. The following criteria is used to manage the user-specified offset addresses:
    a. All user-specified addresses are allocated to AXI4-Lite gateways before automatic allocation.
    b. If two user-specified addresses are the same, an error is issued only during generation (otherwise it will be ignored).
    c. If the remaining AXI4-Lite gateways that are set to allocate address automatically, System Generator attempts to fill the "holes" left behind by user-specified addressing.

Features of the Vivado IDE Example Project

The Vivado® IDE example project (`example_dds.xpr`) is created to help you jump start your usage of the IP created from System Generator. This project is configured as follows:

1. The IP generated from System Generator is already added to the IP catalog associated with the project and available for the RTL flow as well as the IP integrator-based flow.
2. The design includes an RTL instantiation of IP called example_dds_0 underneath example_dds_stub that indicates how to instance such an IP in RTL flow.
3. The design includes a test bench called example_dds_tb that also instances the same IP in RTL flow.
4. The design includes an example IP integrator diagram with a Zynq®-7000 Subsystem as the part selected in this example is a Zynq®-7000 SoC part. For all other parts, a MicroBlaze-based Subsystem is created.

Figure 89: **IP Integrator Diagram**



5. If the part selected is the same as one of the supported boards, the project is set to the first board encountered with the same part setting.
6. A wrapper instancing the block design is created and set as Top.

TIP: *The interface documentation associated with the IP is accessible through the block GUI. To access this documentation, double-click the System Generator IP, and click the* ***Documentation*** *button in the GUI.*

Software Drivers

Bare-metal software drivers are created based on the address offsets assigned to the gateways. These drivers are located in the folder called `<target_directory>/ip/drivers`. `<target_directory>/ip` must be added to the Vitis™ environment search paths to use these drivers.

For each Gateway In mapped to an AXI4-Lite interface, the following two APIs are created.

```
/**
* Write to <Gateway In id> of <design name>. Assignments are LSB-justified.
*
 * @param InstancePtr is the <Gateway In id> instance to operate on.
* @param Data is value to be written to gateway <Gateway In id>.
*
* @return None.
*
* @note    <Text from Description control of the Gateway In GUI>
*
*/
void <Gateway In id>_write(example_dds *InstancePtr, u32 Data);

/**
* Read from <Gateway In id> of <design name>. Assignments are LSB-justified.
*
* @param InstancePtr is the phase_valid instance to operate on.
*
* @return u32
*
* @note    Phase Valid Port That Must Be Asserted.
*
*/
u32 <Gateway In id>_read(example_dds *InstancePtr);
```

`<Gateway In id>`: `<design_name>_<gateway_name>` where `<design_name>` is the VHDL/Verilog top-level name of the design and `<gateway_name>` is the scrubbed name of the gateway.

Gateway Outs generate a similar driver, but are read-only.

Known Issue in AXI4-Lite Interface Generation

Test Bench generation is not supported for designs that have gateways (Gateway In or Gateway Out) configured as an AXI4-Lite Interface.

Tailor Fitting a Platform Based Accelerator Design in System Generator

Platform based accelerators use a bottom-up design methodology to ease the development of larger systems. Two distinct design portions are created: the connectivity platform which connects board level interfaces to a processing system, and the differentiated logic accelerator(s) which represent the data path internal to the SoC and are controlled and/or fed by the connectivity platform design. DSP data paths or accelerators can take advantage of automation to tie into the connectivity platform and its interfaces to external devices.

To speed up creating a design in the Vivado IP Integrator in which the accelerator portion of the design will be developed in System Generator, the following procedure can be used:

1. Create a Block Diagram (BD) of your design in the Vivado IP Integrator. This will act as your connectivity platform.
2. Import the connectivity platform into System Generator.
3. Enter the accelerator portion of the design in System Generator.
4. In System Generator, compile the accelerator model using the IP Catalog flow, to create a Vivado project containing the original design (from the Vivado BD file) and the circuitry in the System Generator model.

Step 1: Create a Connectivity Platform in Vivado as an IP Integrator Block Diagram (.bd)

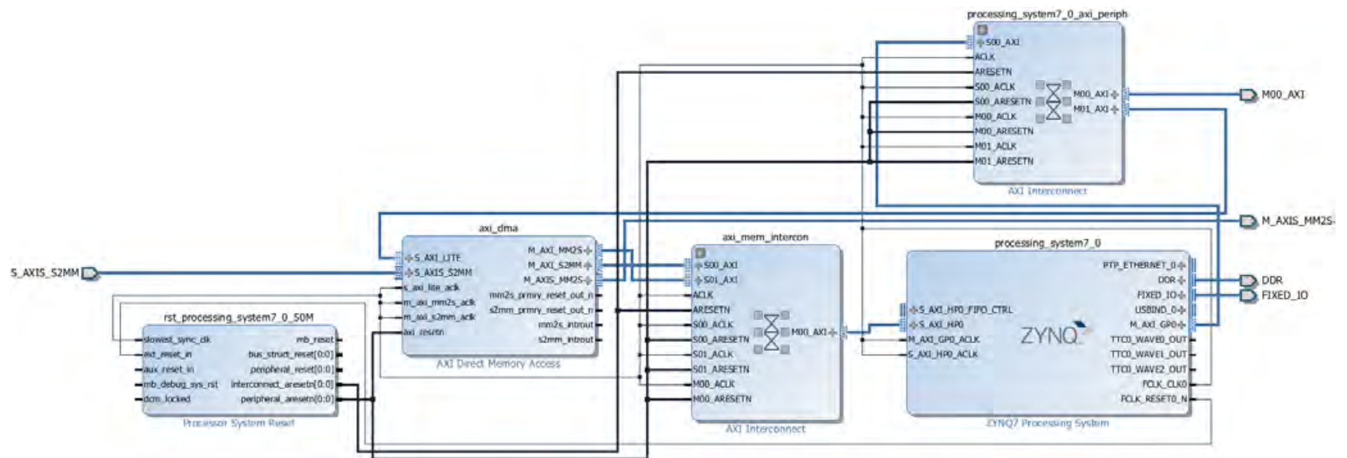
First, you must create a block diagram containing your platform design in the Vivado® IP integrator. You may use a configurable example design, a reference design, or a custom-built design as the platform based system that will contain the accelerator part of the design.

In the example below, the platform design contains a Zynq®-7000 Processing System, and AXI DMA. The connectivity platform designer intends to transfer data to and from the DDR memory using the DMA, perform DES Encryption on the data received from the DDR, and then send the encrypted data back into the DDR. The AXI4-Stream ports M_AXIS_MM2S and S_AXIS_S2MM (Data Path) are made external to the Block Diagram (BD). It shows the intent of the platform designer that these interfaces are available for System Generator to use during the System Generator BD import process. An AXI4-Lite interface, M00_AXI, is also made external, indicating that there will be a control interface on the accelerator IP.

These are requirements for the design in the IP integrator:

- This system has to be built for a specific board or part. This ensures that certain ports and interfaces have known location attributes assigned to them.
- The AXI Interfaces that you want to bring into the accelerator portion of the design have to be made external.

Figure 90: **AXI Interfaces**



Currently we support the following interfaces from the platform framework point of view:

Table 4: **Supported AXI Interfaces**

Interface	Master	Slave
AXI4	Yes	No
AXI4-Lite	Yes	No
AXI4-Stream	Yes	Yes

Step 2: Parse the BD File and Import Un-Located Ports and Interfaces into System Generator

You can now use the `xilinx.utilities.importBD` utility in System Generator to import the BD (Block Diagram) that you created in the Vivado® IP integrator.

This utility takes in the platform framework Vivado project and the name of the new model to be created in System Generator. It parses the platform design for potential System Generator ports and external interfaces (that is, interfaces whose ports do not have location attributes, based on the board connectivity and automation) and creates a sample stub in System Generator representing the accelerator portion of the design.

COMMAND USAGE:

`xilinx.utilities.importBD` takes in the platform Vivado project and the name of the new model to be created. It parses the platform for potential System Generator ports and interfaces and creates a sample stub for the user to make development easy. If the new model name is not specified an untitled model will be opened.

Inputs are: The Vivado project and the model_name (optional)

USAGE:

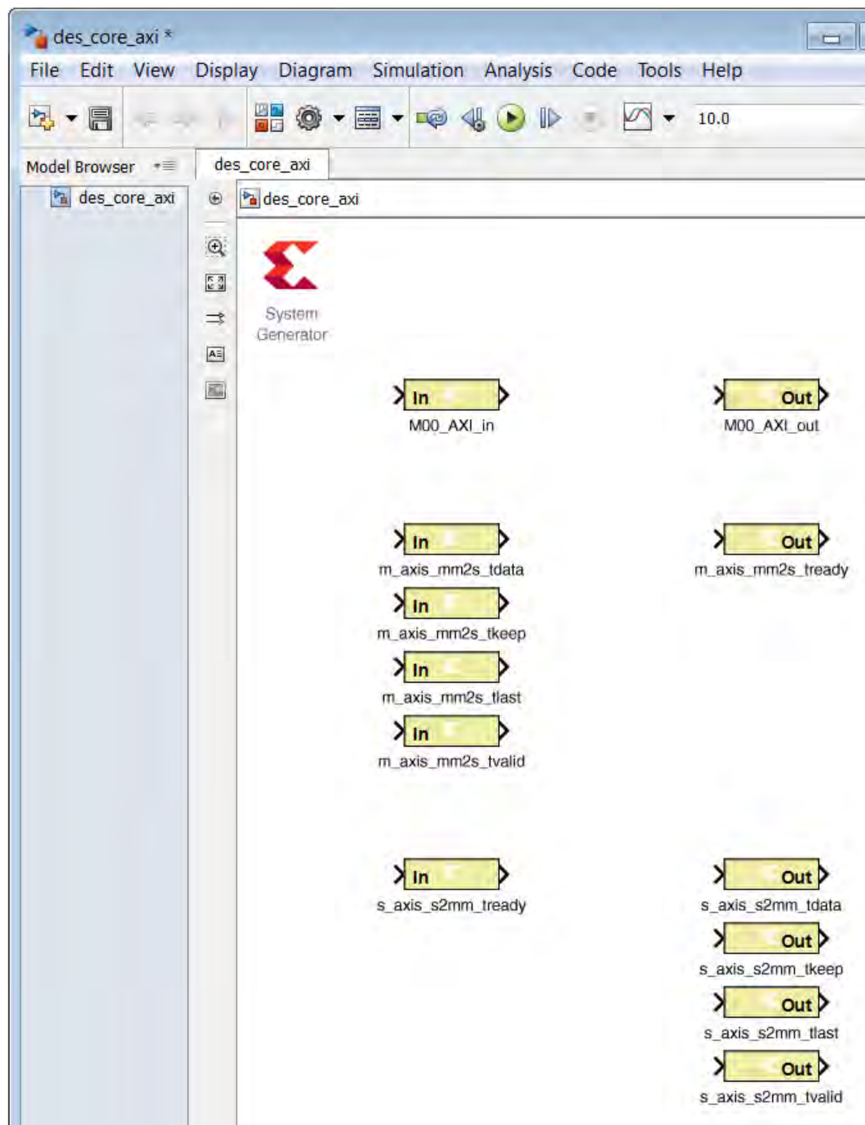
```
xilinx.utilities.importBD('<full_or_relative_path_to_vivado_project_director
y>/

<project_name>.xpr', 'mynewmodel')
```

EXAMPLES

```
xilinx.utilities.importBD('C:\test_importBD\platform.xpr', 'mynewmodel')
xilinx.utilities.importBD('C:\test_impportBD\platform.xpr')
```

In System Generator, the resulting model will look like the example below.

Figure 91: **System Generator Model**



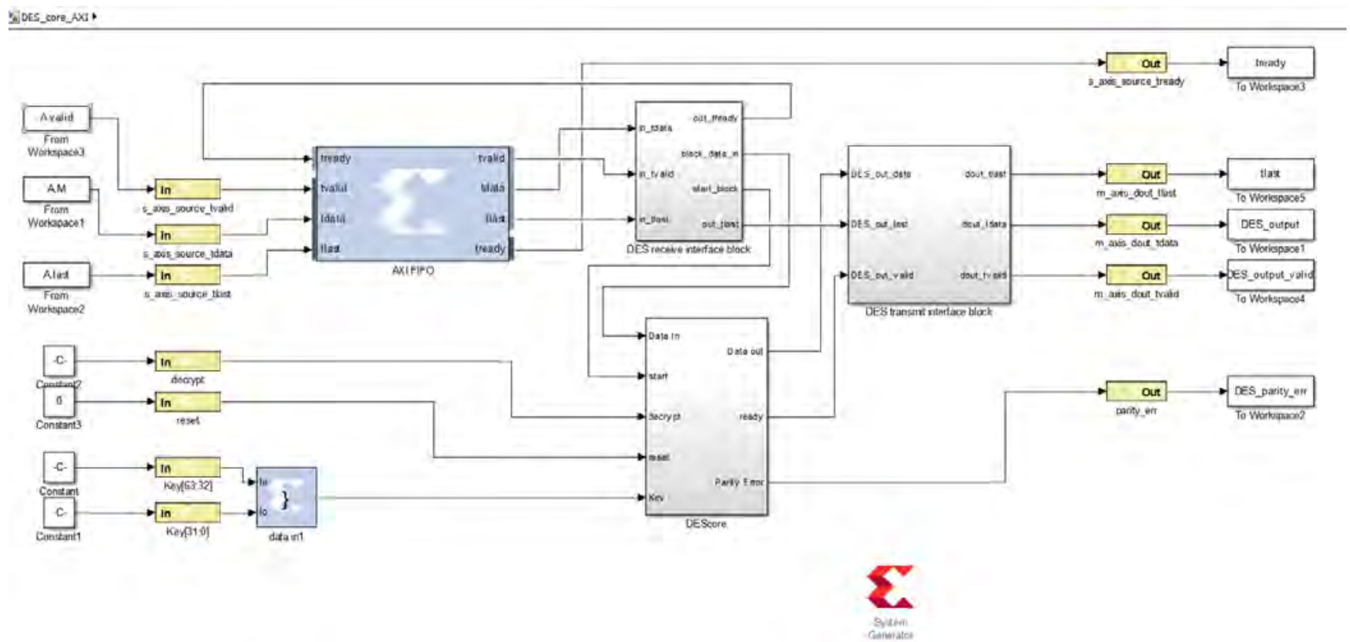
The model in System Generator will have these features:

- For each AXI4-Lite interface, a Gateway In and a Gateway Out block will appear. You can then replicate and add as many AXI4-Lite gateways as your design requires.
- For an AXI4-Stream interface, the associated TDATA, TVALID, TREADY, and other AXI4-Stream ports will appear.
- The model's System Generator token is set to a Compilation target of **IP Catalog** and the **Part** or **Board** will be set to the same Xilinx device or board as that of the Vivado project.

Step 3: In System Generator, Connect Logic to the BD Socket

At this point you can create the accelerator in System Generator. In the example below we have connected to some other logic, and renamed the gateways.

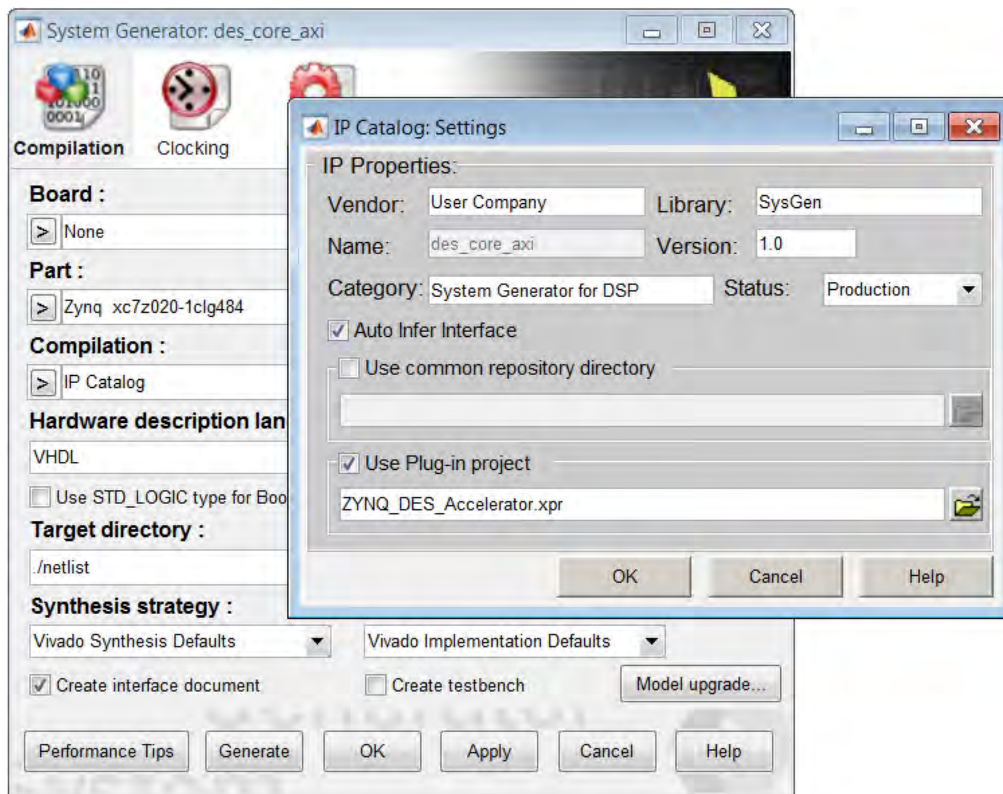
Figure 92: **Connect to BD Socket**



Step 4: Compile the Accelerator Model (IP Catalog Flow) to Create a Complete Design

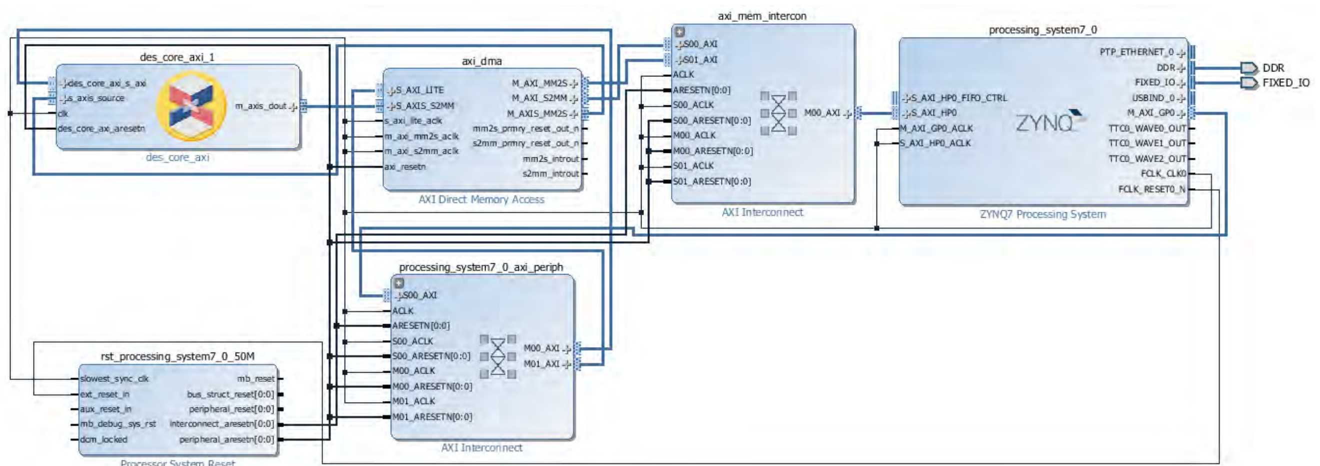
You can now use the IP catalog compilation flow to create a complete design. When you double-click the System Generator token in IP catalog flow and click the **Settings** button, the **Use Plug-in project** directory must point to the Vivado® IP integrator project from which the design was imported (see below). When you click the **Generate** button, a new Vivado project based on the original Vivado platform framework/system plus the accelerator IP created in System Generator, along with a software driver, will be created. This project will be located in an ip_catalog directory under the System Generator token's **Target directory**, and can also be placed into a common IP repository.

Figure 93: **Target Directory**



You can open this new project in Vivado to complete the implementation of your design, as in the sample below. Note the block with the System Generator symbol, which indicates a block developed in the System Generator.

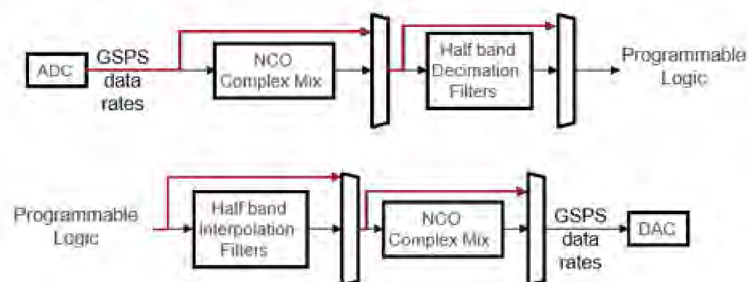
Figure 94: **New Project**



Using Super Sample Rate (SSR) Blocks in System Generator

While the Super Sample Rate (SSR) feature introduced in this section can be widely applicable to all Xilinx® devices, this section explains the motivation for it for Xilinx RFSoC devices. The integration of direct RF-sampling data converters with Xilinx's technology offers the most flexible, smallest footprint, and lowest power solution for a wide range of high performance RF applications such as Wireless communications, cable access, test & measurement, and radar. RFSoC devices provide hardened Digital Up Converters (DUC) and Digital Down Converters (DDC). NCO, Complex Mixers, and Filters are hard Macros, and filter characteristics are optimized for general Commercial applications.

Figure 95: **RFSoC Device**



Depending on what is needed, RFSoC devices can be used in two ways.

- Use the available hardened NCO & Complex Mix and Half Band Decimation/interpolation filters.
- If the sequence of the hardened blocks does not meet the design requirement, you can bypass them as shown in the figure above.

In the latter case, to meet the design requirements, you may need to implement the NCO, Complex Mixerm and DDC blocks in the fabric using System Generator for DSP. To do this, bypass the hardened blocks, and let System Generator IPs run at Programmable Logic (PL) clock frequency. When the sample rate from the ADC is in GSPS, and PL handles only the MSPS range of data, you must accept and compute multiple parallel samples every clock cycle for each data channel. The number of parallel samples is determined by calculating the ratio between the sample frequency and the Programmable Logic clock frequency, which is defined as an SSR parameter.

What is SSR?

SSR is a parameter that determines how many parallel samples to accept for every clock cycle.

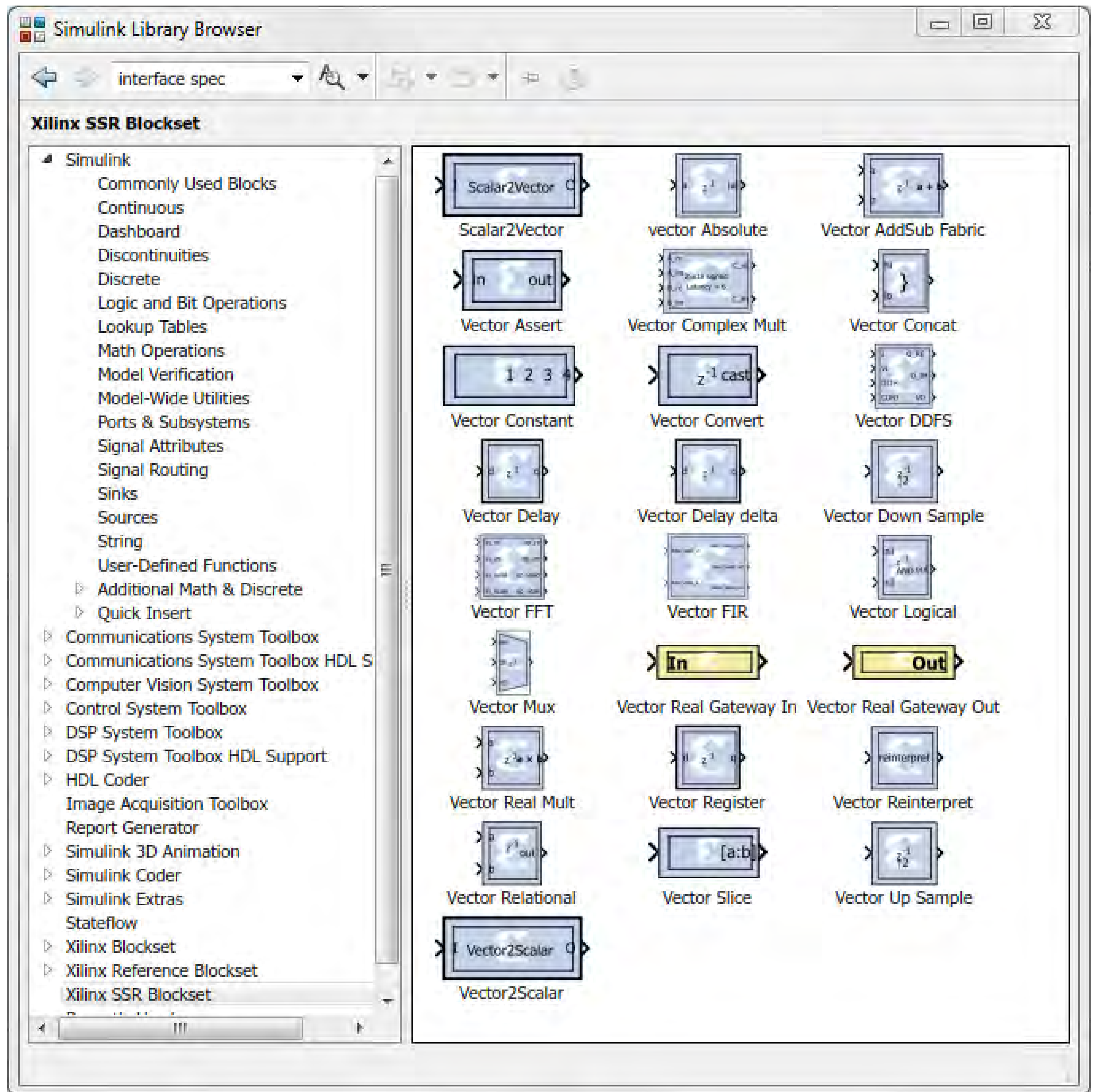
How SSR helps users?

- SSR is beneficial for users who cannot use the hardened RFSoC DUC and DDCs.
- SSR provides programmatic subsystems for NCO and Complex Mixer among many others. The user input parameters in the block mask and System Generator programmatically construct the underlying subsystem with multiple DDS blocks.
- SSR avoids manual and structural modifications to your design, which accelerates the design-cycle.

SSR Library

System Generator provides a separate set of library blocks for handling SSR. Currently, System Generator supports 25 vector blocks, which can be accessed from the MATLAB® Library Browser.

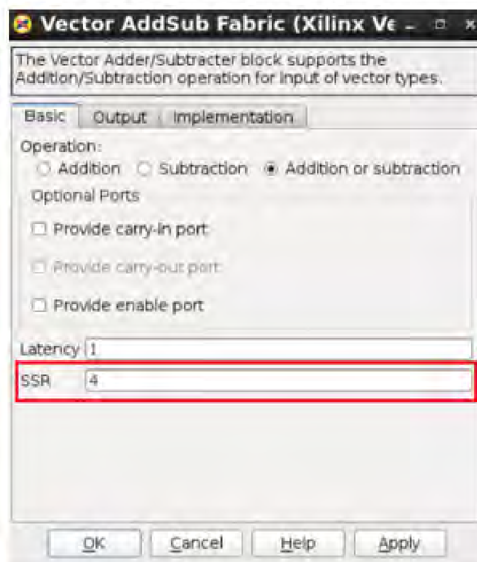
Figure 96: **SSR Block set in MATLAB**



The SSR parameter can be defined for all the blocks present in the SSR block set. When you add a block from the library, the default SSR value is 4, and the maximum SSR valiue is 256.

The SSR block set is defined in *Vivado Design Suite Reference Guide: Model-Based DSP Design Using System Generator* (UG958).

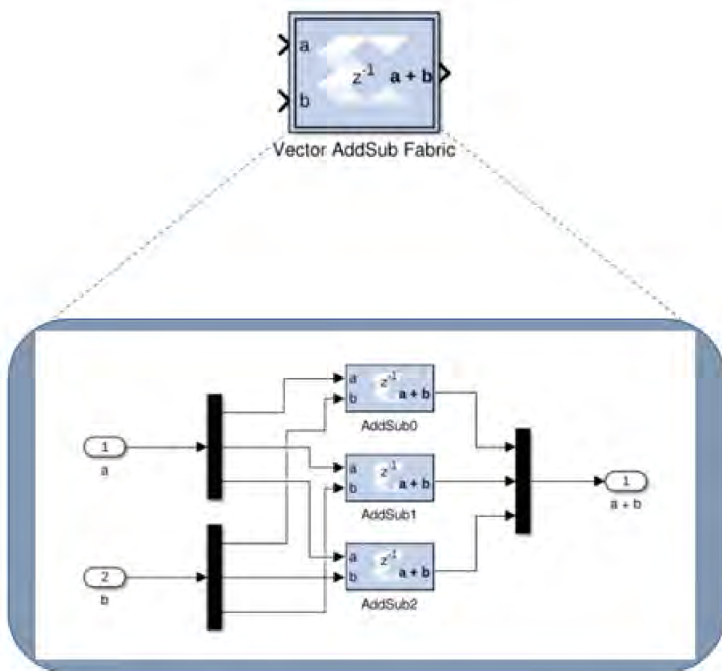
Figure 97: **Default SSR Value**



No matter what the SSR rate is, you only need to provide a limited number of signal connections as with a normal IP block. System Generator automatically takes care of all the parallel path connection internal to the SSR block, according to the SSR parameter value provided.

For example, for a Vector AddSub block, when SSR parameter is modified to 3, the internal connections are done automatically as shown below. This creates 3 parallel paths for computation and results in single output.

Figure 98: **Vector AddSub Fabric Example**



XILINX

Chapter 4

Performing Analysis in System Generator

System Generator is a bit and cycle accurate modeling tool. You can verify the functionality of your designs by simulating in Simulink®. However, to ensure that your System Generator design will work correctly when it is implemented in your target Xilinx® device, these analysis tools have been integrated into System Generator:

- **Timing Analysis:** To ensure that the HDL files generated by System Generator operate correctly in hardware, you must close timing. To help accelerate this process, timing analysis has been integrated into System Generator.
- **Resource Analysis:** To ensure that the HDL files generated by System Generator will fit into your target device, you may need to analyze the resources being used. To help accelerate this process, resource analysis has been integrated into System Generator.

Timing Analysis in System Generator	Presents an overview of timing analysis in System Generator.
Performing Timing Analysis	Describes how to perform timing analysis on your model.
Cross Probing from the Timing Analysis Results to the Model	Describes how you can cross probe from a row in the Timing Analyzer table to the Simulink model, highlighting the corresponding System Generator blocks in the path.
Accessing Existing Timing Analysis Results	Describes how to re-launch the Timing Analyzer table on pre-existing Timing Analysis results.
Recommendations For Troubleshooting Timing Violations	Describes methods to help you discover the source of timing violations in your design.
Resource Analysis in System Generator	Presents an overview of resource analysis in System Generator.
Performing Resource Analysis	Describes how to perform resource analysis on your model.
Cross Probing from the Resource Analysis Results to the Model	Describes how you can cross probe from a row in the Resource Analyzer table to the Simulink model, highlighting the corresponding block or subsystem in the design.
Accessing Existing Resource Analysis Results	Describes how to re-launch the Resource Analyzer table on pre-existing Resource Analysis results.
Recommendations for Optimizing Resource Analysis	Describes methods to help you use the Resource Analyzer to optimize resource utilization in the design.

Timing Analysis in System Generator

To ensure that the HDL files generated by System Generator work correctly in hardware, you must close timing. To help accelerate this process, timing analysis has been integrated into System Generator.

Timing analysis allows you to perform static timing analysis on the HDL files generated from System Generator, either Post-Synthesis or Post-Implementation. It also provides a mechanism to correlate the results of running the Vivado® Timing Engine on either the Post-Synthesized netlist or the Post Implementation netlist with the System Generator model in Simulink®. Thus, you do not have to leave the Simulink® modeling environment to close timing on the DSP sub-module of the design.

Invoking timing analysis on a compilation target (for example, HDL Netlist) results in a tabulated display of paths with columns showing information such as timing slack, path delay, etc. This is the Timing Analyzer table. You can sort the contents of the table using any of the column metrics such as slack, etc. Also, cross probing is enabled between the table entries and the Simulink model to accelerate finding and fixing timing failures in the model. Cross probing between the Timing Analyzer table and the Simulink model is accomplished by selecting/clicking a row in the table. The corresponding path in the model will be highlighted. The path is highlighted in red if the path corresponds to a timing violation; otherwise it is highlighted in green.

Performing Timing Analysis

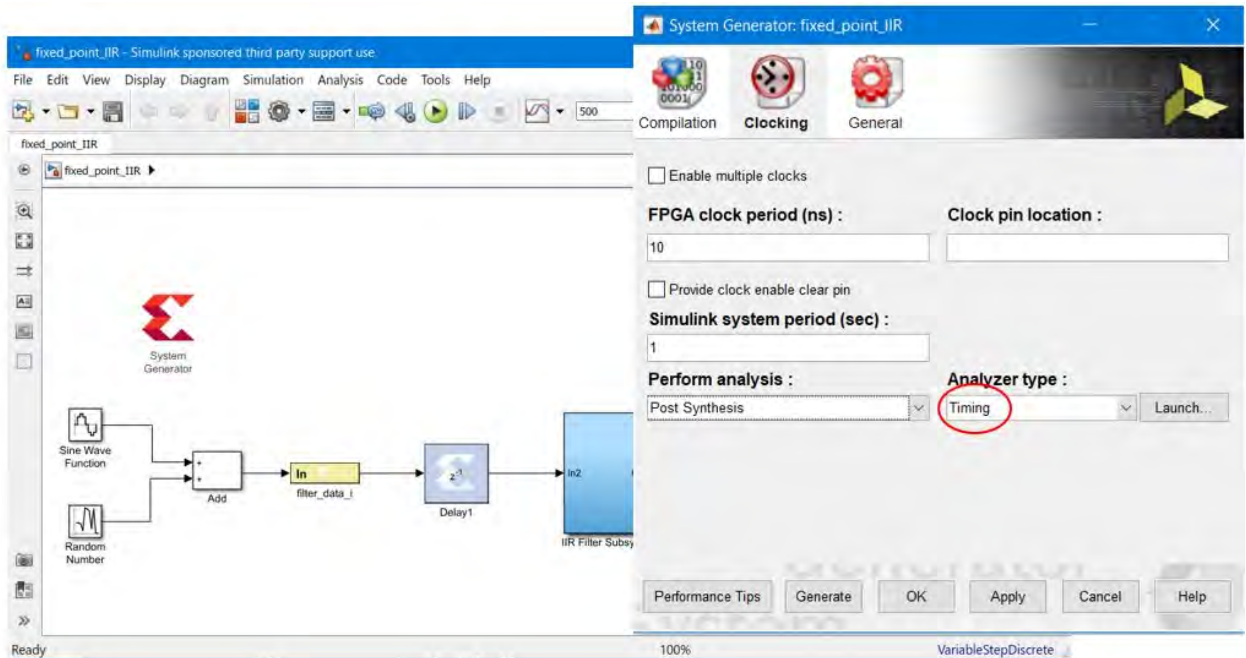
Timing analysis can be invoked whenever you generate any of the following compilation targets:

- IP catalog
- Hardware Co-Simulation
- Synthesized Checkpoint
- HDL Netlist

To perform timing analysis in System Generator:

1. Double-click the System Generator token in the Simulink model.
2. Enter the following in the System Generator token dialog box:
   - In the **Compilation** tab, specify a **Target Directory**.
   - In the **Clocking** tab, set the **Perform Analysis** field to **Post Synthesis** or **Post Implementation** based on the runtime vs. accuracy tradeoff.
   - In the **Clocking** tab, set the **Analyzer Type** field to **Timing**.

Figure 99: **Performing Timing Analysis**



3. In the System Generator token dialog box, click **Generate**.

   When you generate, the following occurs:

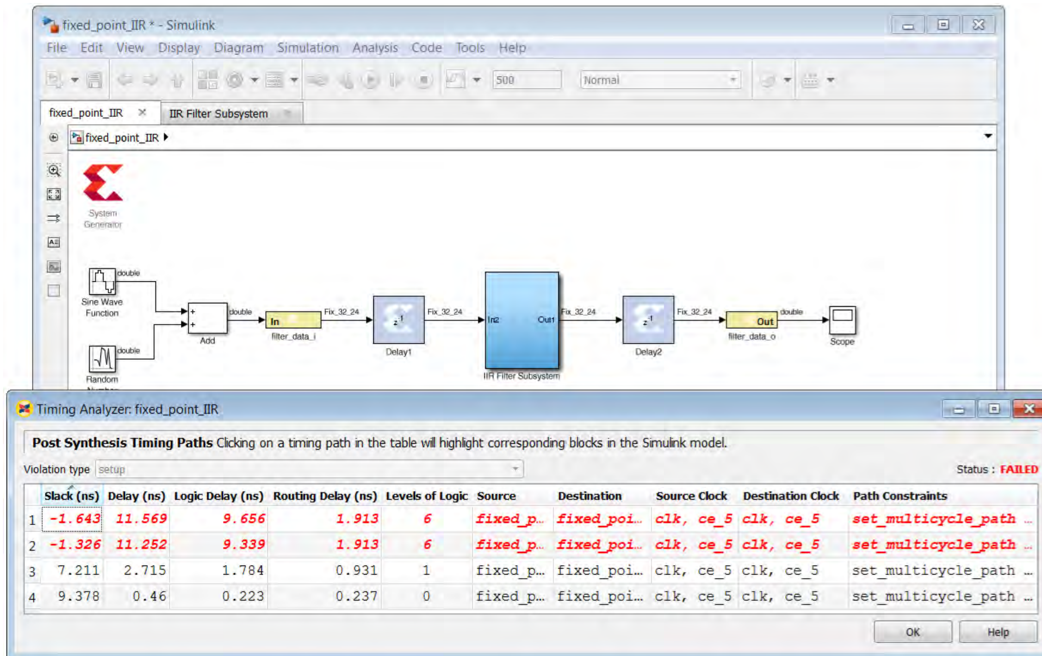
   a. System Generator generates the required files for the selected compilation target. For timing analysis System Generator invokes Vivado in the background for the design project, and passes design timing constraints to Vivado.

   b. Depending on your selection for **Perform Analysis** (**Post Synthesis** or **Post Implementation**), the design runs in Vivado through synthesis or through implementation.

   c. After the Vivado tools run is completed, timing paths information is collected and saved in a specific file format from the Vivado timing database. At the end of the timing paths data collection the Vivado project is closed and control is passed to the MATLAB®/ System Generator process.

   d. System Generator processes the timing information and displays a Timing Analyzer table with timing paths information (see below).

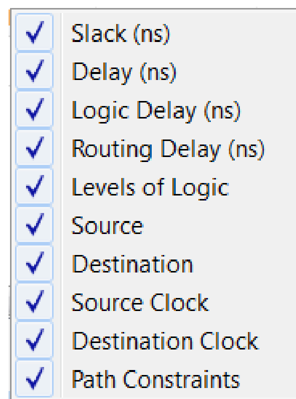
Figure 100: **Timing Analyzer Table**



In the timing analyzer table:

- Only unique paths from the Simulink model are reported.
- The 50 paths with the lowest Slack values are displayed with the worst Slack at the top, and increasing Slack below.
- Paths with timing violations have a negative Slack and display in red.
- The display order can be sorted for any column's values by clicking the column head.
- If you want to hide a column in the table, right-click any column head in the table and deselect the column to hide in the list that appears.

Figure 101: **Hide/Show Dialog**



- For a design with multiple clock cycle constraints, the Timing Analyzer can identify multicycle path constraints, and show them in the **Path Constraints** column. In that case, the **Source Clock**, and **Destination Clock** columns display clock enable signals to reflect different sampling rates.

Figure 102: **Clock Enable Signals**

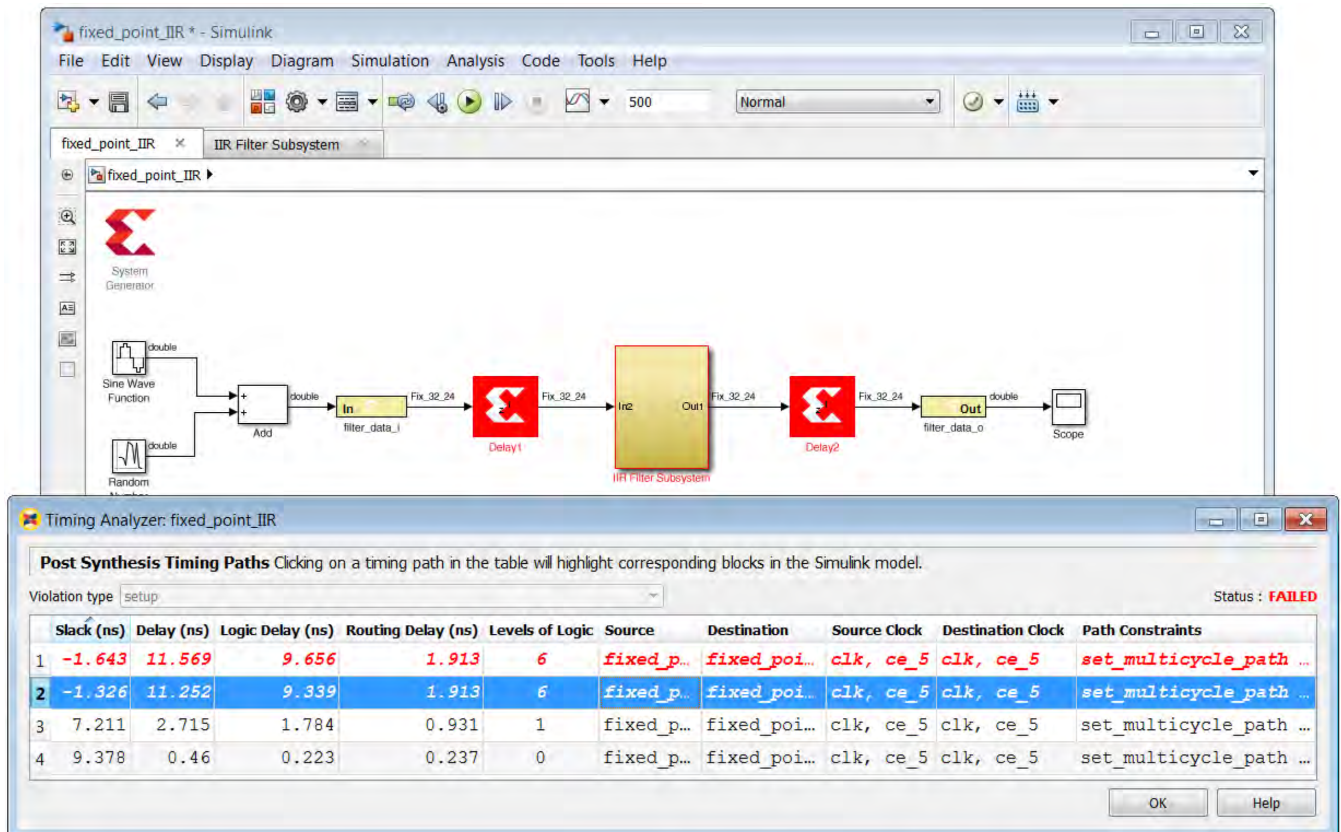
Source Clock	Destination Clock	Path Constraints
clk, ce_3	clk, ce_3	set_multicycle_path -setup 3 -hold 2
clk, ce_4	clk, ce_4	set_multicycle_path -setup 4 -hold 3
clk, ce_6	clk, ce_6	set_multicycle_path -setup 6 -hold 5
clk, ce_12	clk, ce_12	set_multicycle_path -setup 12 -hold 11
clk, ce_12	clk, ce_12	set_multicycle_path -setup 12 -hold 11
clk, ce_12	clk, ce_12	set_multicycle_path -setup 12 -hold 11

- You can cross probe from the table to the Simulink model by selecting a path in the table, which will highlight the corresponding System Generator blocks in the Simulink model. See Cross Probing from the Timing Analysis Results to the Model.

Cross Probing from the Timing Analysis Results to the Model

You can cross probe from the Timing Analyzer table to the Simulink model by clicking any path (row) in the Timing Analyzer table, which highlights the corresponding System Generator blocks in the model. This allows you to troubleshoot timing violations by analyzing the path on which they occur.

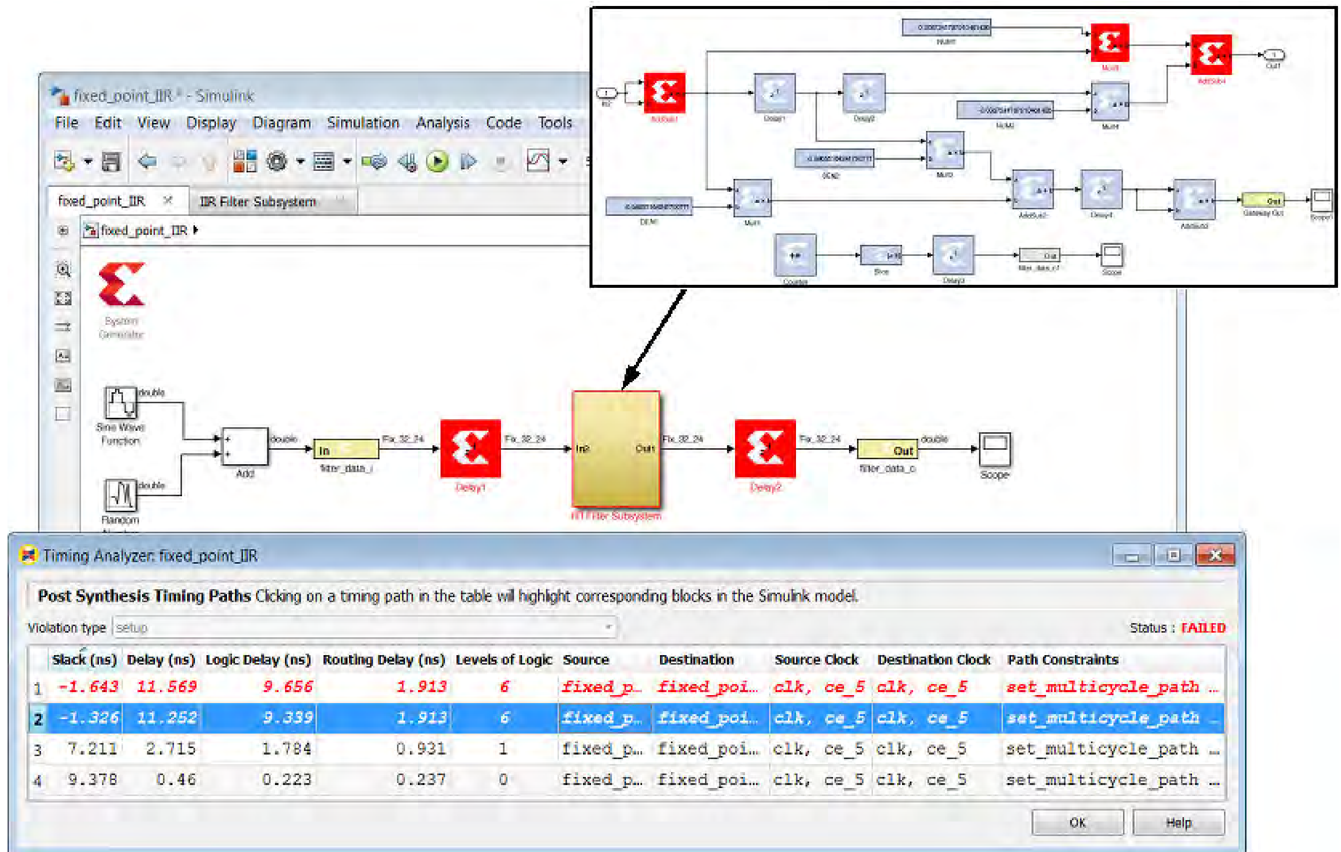
Figure 103: **Timing Analyzer Table**



When you cross probe, the following will display in the model:

- Blocks in a path with a timing violation are highlighted in red in the model, whereas blocks that belong to a path with no timing violation (that is, a path with a positive Slack value) are highlighted in green in the model.
- If blocks in a highlighted path are inside a subsystem, then the subsystem is highlighted in red so you may expand the subsystem to inspect the blocks underneath.

Figure 104: **Cross Probing**



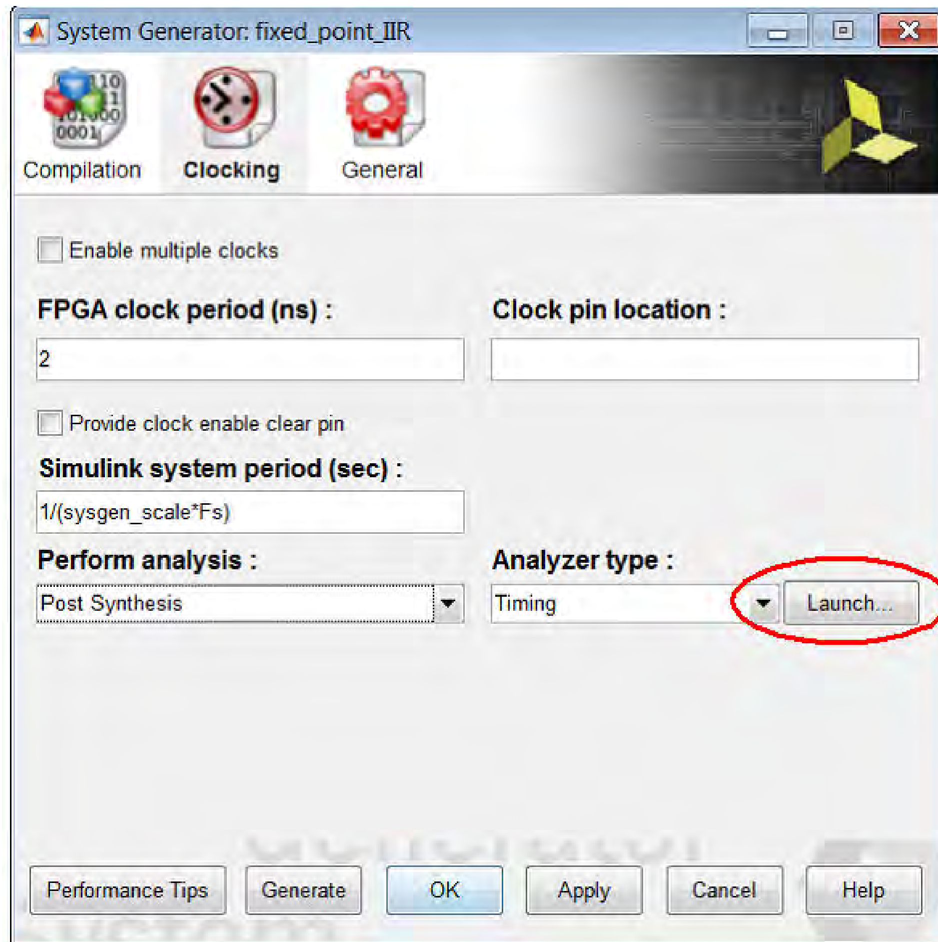
- When you select a path (row in the table) to cross probe, this normally highlights the destination block at the end of the path. That brings the subsystem containing the destination block to the front in the model. As a result, you may not be able to see the highlighted source block if the source block is in a different subsystem. If you want to see the source block, click the path in the **Source** column in the table. This will bring the subsystem containing the source block to the front of the model. Selecting the path in any other column will bring the subsystem containing the destination block to the front.

Accessing Existing Timing Analysis Results

A **Launch** button is provided under the **Clocking** tab of the System Generator token dialog box to relaunch the Timing Analyzer table using the existing timing analysis results for the model. Make sure the **Target directory** specified on the **Compilation** tab of the dialog box is readable by the Timing Analyzer, and the **Analyzer Type** field is set to **Timing**. This will only work if you already ran timing analysis on the Simulink model and haven't changed the Simulink model since the last run.

When you click the **Launch** button, the Timing Analyzer table will display the timing analysis results stored in the specified **Target directory**, regardless of the option selected for **Perform analysis** (**Post Synthesis** or **Post Implementation**).

Figure 105: **Launch Button**



You can also launch the Timing Analyzer table to display existing timing analysis results by entering this command at the MATLAB® command prompt:

```
xlAnalyzeTiming(<mdl_hdle>, <netlist_dir>)
```

where `<mdl_hdle>` is the Simulink® model handle (the handle of the top level design), and `<netlist_dir>` is the **Target directory** specified in the System Generator token dialog box.

Recommendations For Troubleshooting Timing Violations

The following are recommended for troubleshooting timing violations:

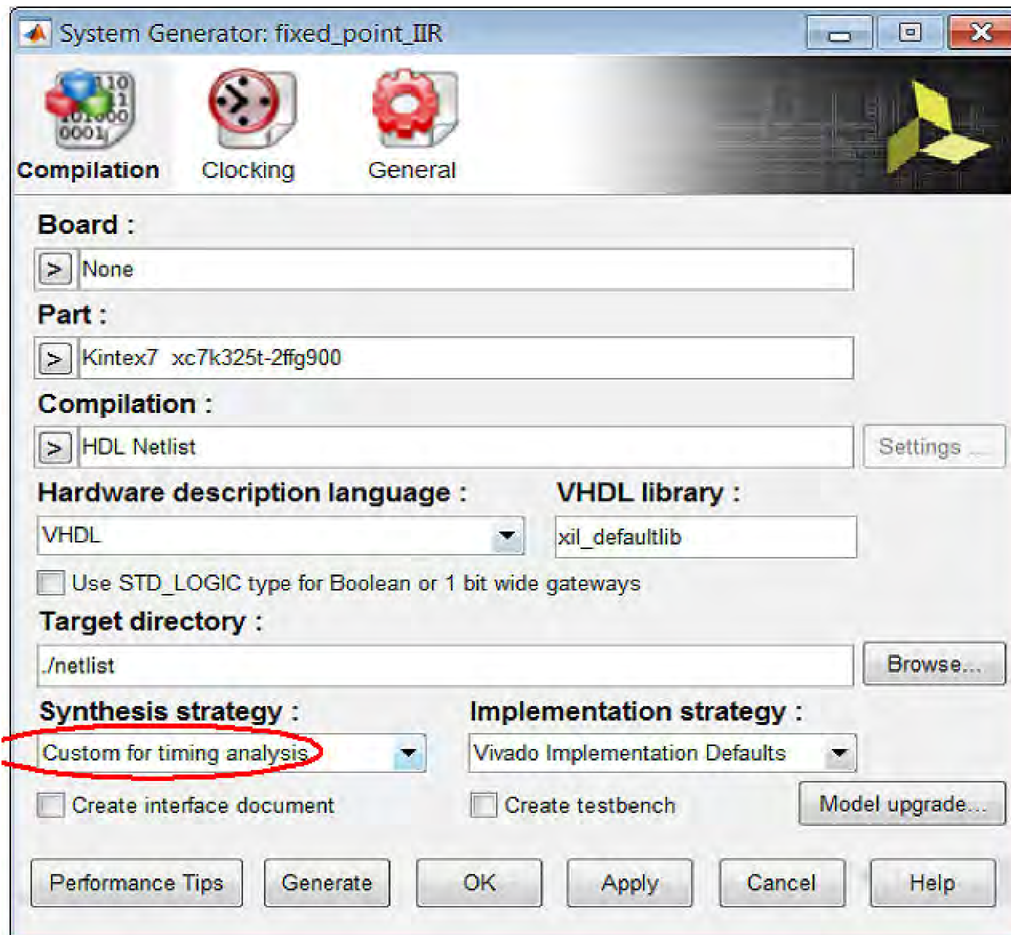
- For quicker timing analysis iterations, post-synthesis analysis is preferred over post-implementation analysis.
- After logic optimization during the Vivado Synthesis process the tool doesn't keep information about merged logic in the Vivado database. Merged and shared logic may make it difficult to accurately cross probe from Vivado timing paths to the Simulink model. Hence, it is recommended that you create a custom Vivado Synthesis strategy to control merged and shared logic.

For information about how to create a custom Synthesis strategy in Vivado, see this link in the *Vivado Design Suite User Guide: Using the Vivado IDE* (UG893).

To control merged and shared logic in the Vivado IDE, make the following changes to the default Vivado Synthesis strategy.

1. Set these Synthesis options in Vivado IDE:
   - Select the Synthesis option `-keep_equivalent_registers`.
   - Set the Synthesis option `-resource_sharing` to the value `off`.
2. Save the new Synthesis strategy and exit Vivado IDE.
3. In System Generator, select the new custom **Synthesis strategy** in the System Generator token dialog box before generating the design.

Figure 106: **Custom for Timing Analysis**



Resource Analysis in System Generator

To ensure that the HDL files generated by System Generator will fit into your target device, you may need to analyze the resources being used. To help accelerate this process, resource analysis has been integrated into System Generator.

Resource analysis allows you to determine the number of look-up tables (LUTs), registers, DSP48s (DSPs), and block RAMs (BRAMs) used by your model. The analysis is performed either Post-Synthesis or Post-Implementation and provides a mechanism to correlate the resources used in the Vivado® tools with the System Generator model in Simulink®. Thus, you do not have to leave the Simulink modeling environment to investigate and determine areas where excessive resources are being used in your design.

Invoking resource analysis on a compilation target (for example, IP catalog) results in a tabulated display of blocks, and hierarchies showing LUT, Register, DSP, and block RAM resource usage. This is the Resource Analysis table. You can sort the contents of the table using any of the column metrics such as DSPs, etc. Also, cross probing is enabled between the table entries and the Simulink model to accelerate finding and fixing excessive resource usage in the model. Cross probing between the Resource Analysis table, and the Simulink model is accomplished by selecting (clicking) a row in the table. The corresponding block, or hierarchy in the model is highlighted in yellow.

Performing Resource Analysis

Resource analysis can be performed whenever you generate any of the following compilation targets:

- IP Catalog
- Hardware Co-Simulation
- Synthesized Checkpoint
- HDL Netlist

To perform resource analysis in System Generator:

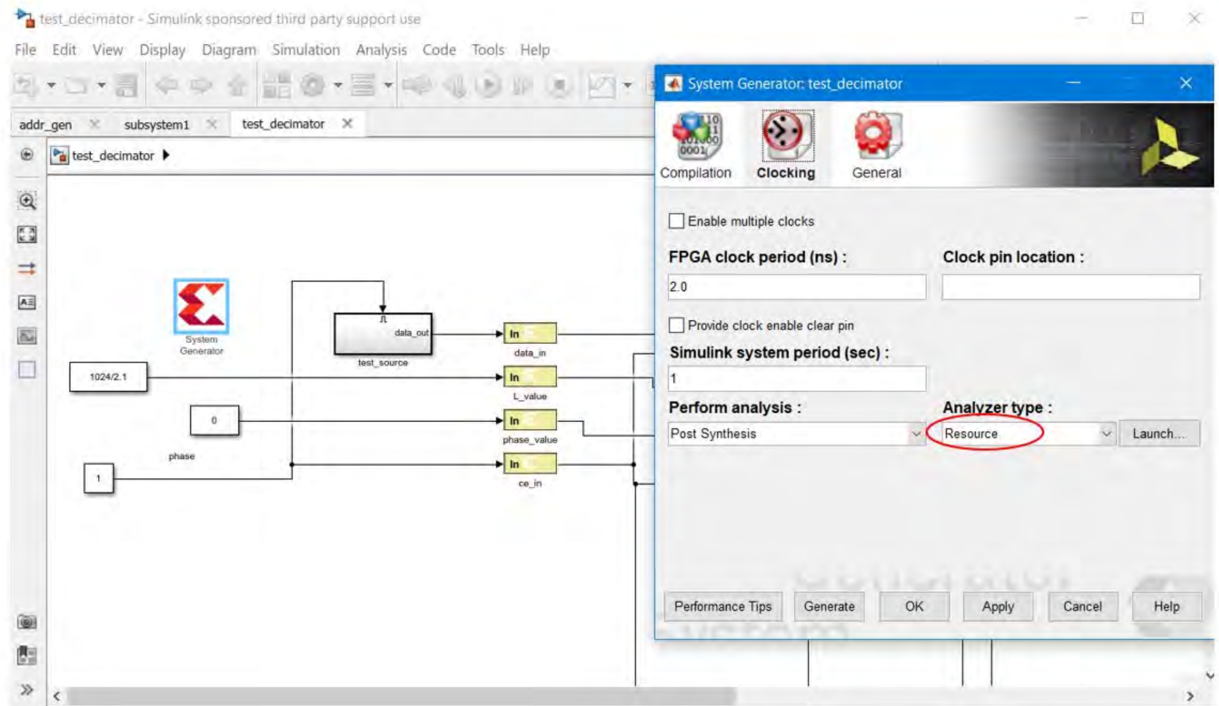
1. Double-click the System Generator token in the Simulink model.
2. Select the following in the System Generator token dialog box:
    a. In the **Compilation** tab:
        - Specify the **Part** in which your design will be implemented.

          ***Note*:** If you select a **Board** instead of a **Part**, the **Part** field will be filled in with the name of the part on the selected **Board**.
        - Select one of the **Compilation** targets.

          System Generator can perform resource analysis for any **Compilation** target you select.
        - Specify a **Target Directory**.

    b. In the **Clocking** tab:
        - Set the **Perform Analysis** field to **Post Synthesis** or **Post Implementation** based on the runtime vs. accuracy tradeoff.
        - Set the **Analyzer type** field to **Resource**.

Figure 107: **Resource Analyzer**



3. In the System Generator token dialog box, click **Generate**.

   When you generate, the following occurs:

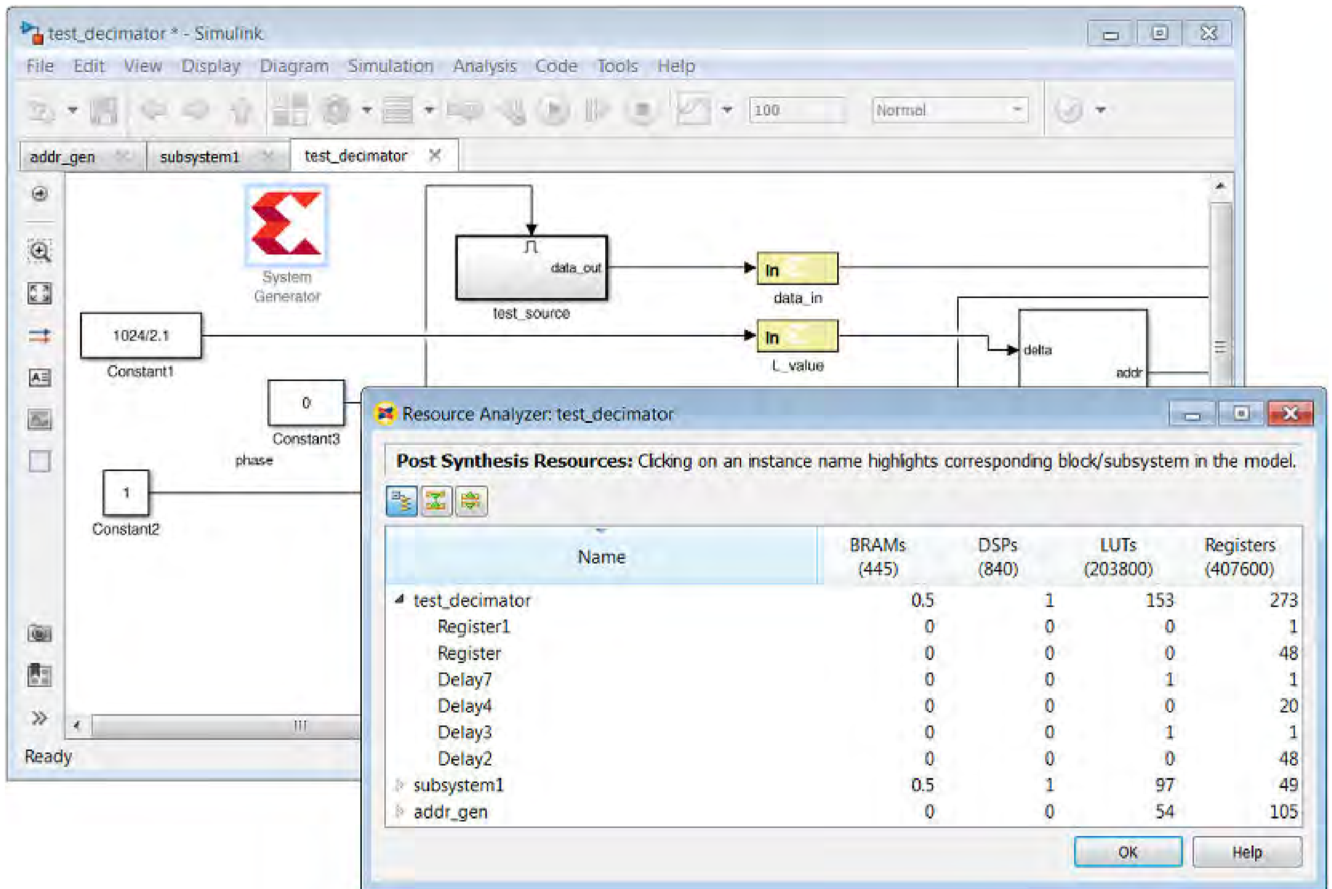
   a. System Generator generates the required files for the selected compilation target. For resource analysis System Generator invokes Vivado in the background for the design project.

   b. Depending on your selection for **Perform analysis** (**Post Synthesis** or **Post Implementation**), the design runs in Vivado through synthesis or through implementation.

   c. After the Vivado tools run is completed, resource utilization data is collected from the Vivado resource utilization database and saved in a specific file format under the target directory. At the end of the resource utilization data collection the Vivado project is closed and control is passed to the MATLAB/System Generator process.

   d. System Generator processes the resource utilization data and displays a Resource Analyzer table with resource utilization information (see below).

Figure 108: **Resource Analyzer**



In the resource analyzer table:

- The header section of the dialog box indicates the Vivado design stage after which resource utilization data was collected from Vivado. This will be either **Post Synthesis** or **Post Implementation**.
- The local toolbar contains the following commands to change the display of resource counts:
  - Hierarchical/Flat Display: Toggles the display between a hierarchical tree and a flattened list.
  - Collapse All: Collapses the design hierarchy to display only the top-level objects.
  - Expand All: Expands the design hierarchy at all levels to display resources used by each subsystem and each block in the design.
- The number shown in each column heading indicates the total number of each type of resource available in the Xilinx device for which you are targeting your design. In the example below, the design is targeting a Kintex-7 FPGA.

Figure 109: **Resource Analysis Report for Kintex-7**

Resource Analyzer: test_decimator

Post Synthesis Resources: Clicking on an instance name highlights corresponding block/subsystem in the model.

Name	BRAMs (445)	DSPs (840)	LUTs (203800)	Registers (407600)
test_decimator	0.5	1	153	273
subsystem1	0.5	1	97	49
addr_gen	0	0	54	105

- The example displays a hierarchical listing of each subsystem and block in the design, with the count of the following resource types:
  - **BRAMs:** block RAM and FIFO primitives.block RAMs (BRAMs) are counted in this way.

    Table 5: **Number of BRAMs**

Primitive Type	# BRAMs
RAMB36E	1
FIFO36E	1
RAMB18E	0.5
FIFO18E	0.5

    Variations of Primitives (for example, RAM36E1 and RAM36E2) are all counted in the same way.

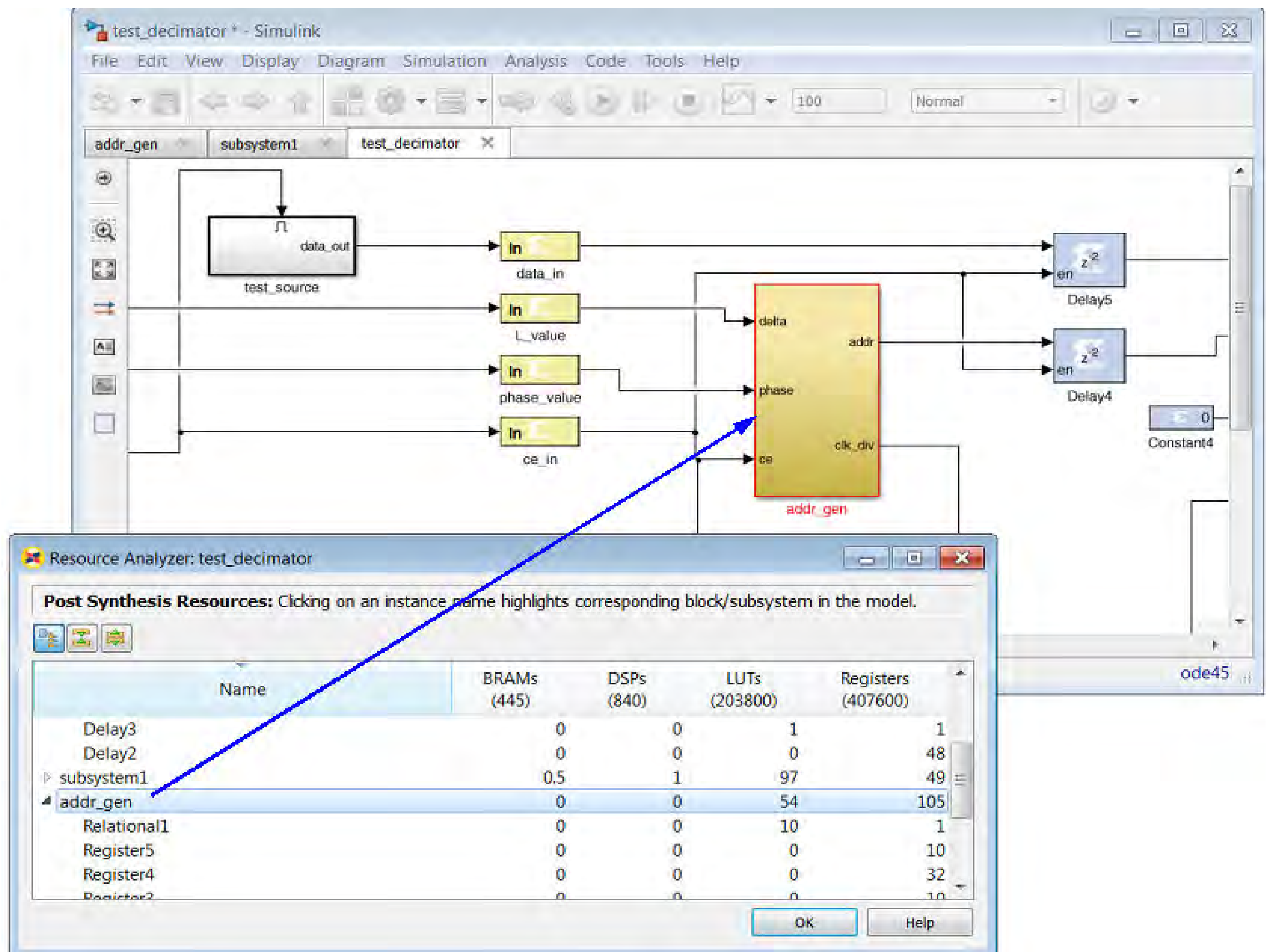
    Total BRAMs = (Number of RAMB36E) + (Number of FIFO36E) + 0.5 (Number of RAMB18E + Number of FIFO18E)

  - **DSPs:** DSP48 primitives (DSP48E, DSP48E1, DSP48E2) and DSP58
  - **Registers:** Registers and Flip-Flops. All primitive names that start with FD* (FDCE, FDPE, FDRE, FDSE, etc.) and LD* (LDCE, LDPE, etc.) are considered as **Registers**.
  - **LUTs:** All LUT types combined.
- The display order can be sorted for any column's values by clicking the column head.
- You can cross probe from the table to the Simulink model by selecting a row in the table, which will highlight the corresponding System Generator blocks in the Simulink model. See Cross Probing from the Resource Analysis Results to the Model.

Cross Probing from the Resource Analysis Results to the Model

You can cross probe from the Resource Analyzer table to the Simulink® model by clicking a block or subsystem name in the Resource Analyzer table, which highlights the corresponding System Generator block or subsystem in the model. The cross probing is useful to identify blocks and subsystems that are implemented using a particular type of resource.

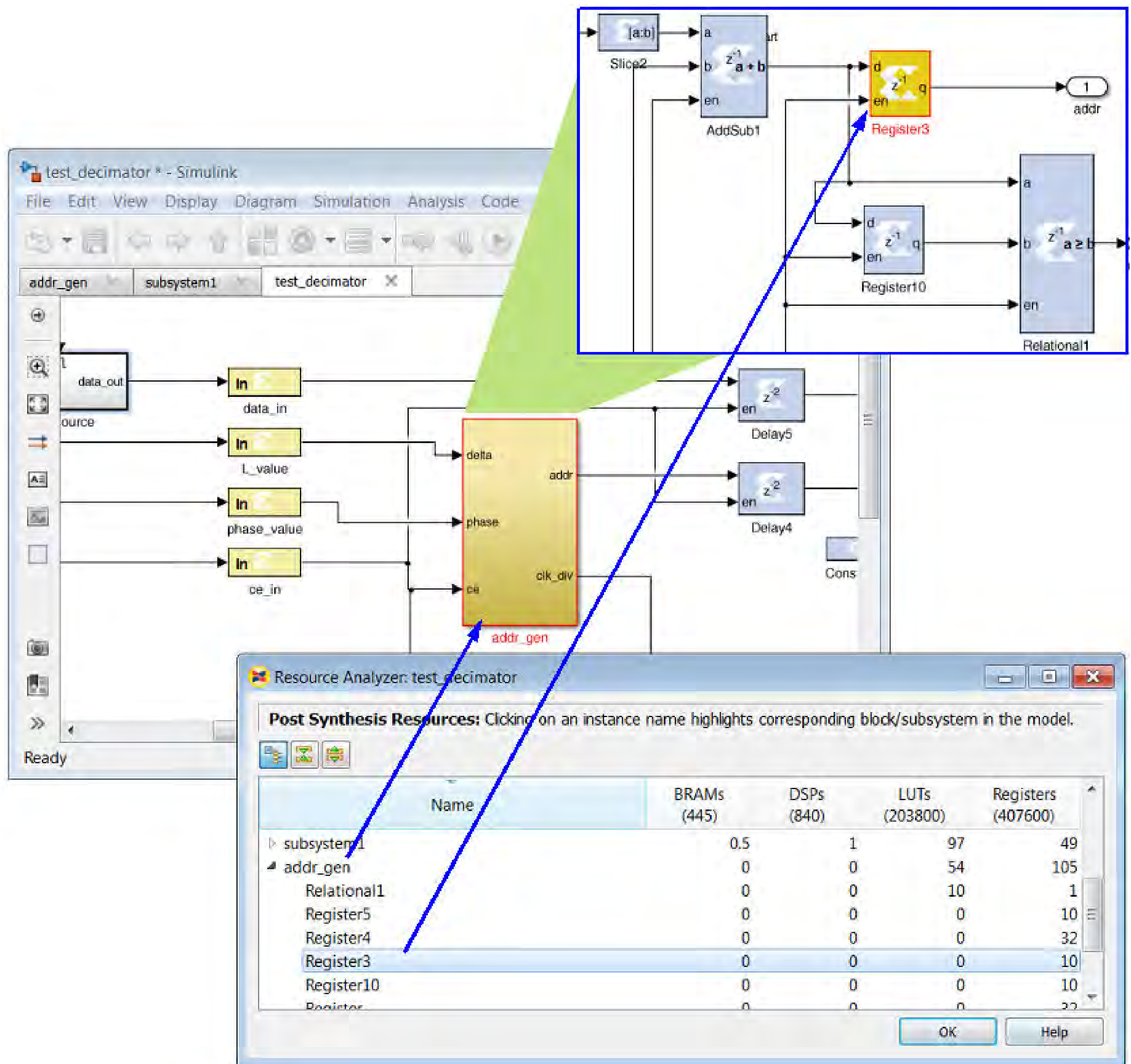
Figure 110: **Resource Analyzer**



When you cross probe, the following will display in the model:

- The block you have selected in the table will be highlighted in yellow and outlined in red.
- If the block or subsystem you have selected in the table is within an upper-level subsystem, then the parent subsystem is highlighted in red in addition to the underlying block.

Figure 111: **Selected Subsystem**

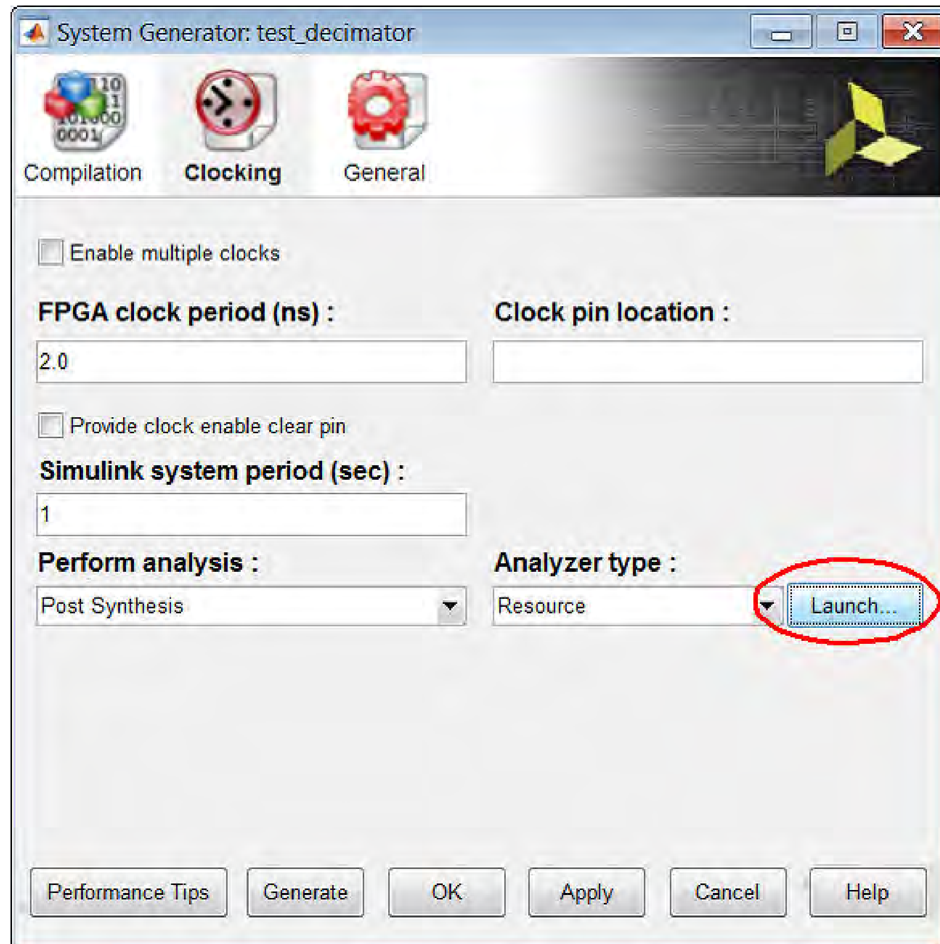


Accessing Existing Resource Analysis Results

A **Launch** button is provided under the **Clocking** tab of the System Generator token dialog box to launch the Resource Analyzer table using the existing resource utilization results for the model. Make sure the **Target directory** specified on the **Compilation** tab of the dialog box is readable by the Resource Analyzer, and the **Analyzer type** field is set to **Resource**. This will only work if you already ran analysis on the Simulink model and haven't changed the Simulink model since the last run.

When you click the **Launch** button, the Resource Analyzer table will display the resource utilization results stored in the **Target directory** specified on the **Compilation** tab, regardless of the option selected for **Perform analysis** (the **Post Synthesis** or **Post Implementation** option).

Figure 112: **Launch Button**



You can launch the Resource Analyzer table to display existing resource utilization results by entering the following command at the MATLAB® command prompt:

```
>> xlAnalyzeResource(get_param('model_name','handle'),'./netlist')
```

- `get_param('model_name','handle')` gives you the model handle.

  ***Note*:** `model_name` represents the name of the model.

- For the path to `netlist` directory, you can use an absolute path, or if you are using this API from the same directory where `netlist` directory is present, then you can use a relative path like `'./netlist'`.

Recommendations for Optimizing Resource Analysis

The following are recommended for using the Resource Analyzer to optimize resource utilization in the design:

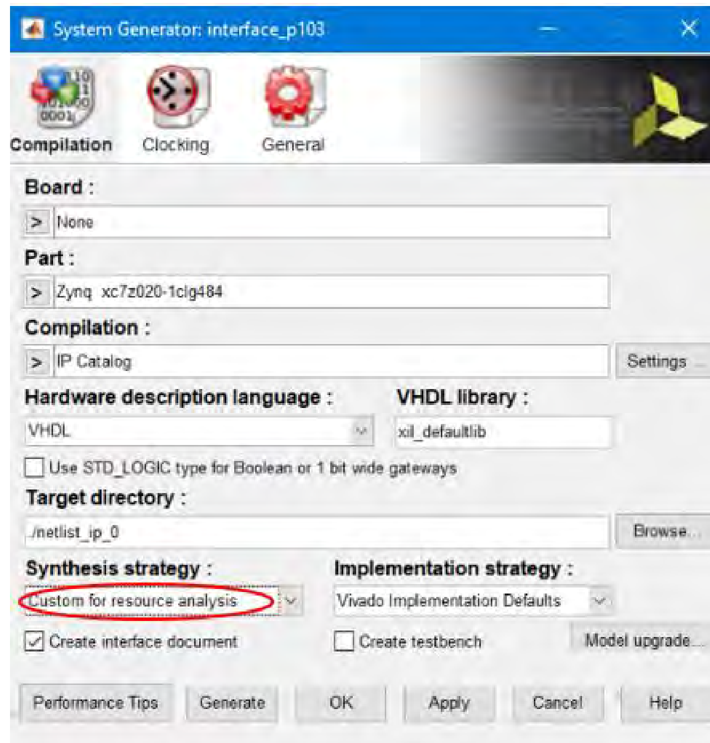
- For quicker resource analysis iterations, post-synthesis analysis is preferred over post-implementation analysis.
- After logic optimization during the Vivado Synthesis process the tool does not keep information about merged logic in the Vivado database. Merged and shared logic may make it difficult to accurately cross probe from Vivado resource data to the Simulink model. Hence, it is recommended that you create a custom Vivado Synthesis strategy to control merged and shared logic.

  For information about how to create a custom Synthesis strategy in Vivado IDE, see this link in the *Vivado Design Suite User Guide: Using the Vivado IDE* (UG893).

  To control merged and shared logic in the Vivado IDE, make the following changes to the default Vivado Synthesis strategy.

  1. In Vivado IDE:
     - Select the Synthesis option `-keep_equivalent_registers`.
     - Set the Synthesis option `-resource_sharing` to the value `off`.
  2. Save the new Synthesis strategy and exit Vivado IDE.
  3. In System Generator, select the new custom **Synthesis strategy** in the System Generator token dialog box before generating the design.

Figure 113: **Synthesis Strategy**



Using Hardware Co-Simulation

System Generator provides hardware co-simulation, making it possible to incorporate a design running in an FPGA directly into a Simulink® simulation. This allows all (or a portion) of the System Generator design that had been simulating in Simulink as sequential software to be executed in parallel on the FPGA, and can speed up simulation dramatically. Users of this flow can send larger data sets, or more test vectors, doing an exhaustive functional test of the implemented logic. This increased code coverage allows more corner cases to be verified to help identify design bugs in the logic. Data at the input to the compiled co-simulation block on the Simulink model is sent to the target FPGA, either as one transaction or a burst of transactions, executed for a given number of clock cycles in parallel, and read back to the model's co-simulation outputs.

Hardware co-simulation has two compilation types: burst or non-burst (standard). The burst mode provides much higher performance. Channels to each input of the compiled co-simulation target are opened and packets of data are sent to the open channel, followed by bursting to all of the remaining inputs. The FPGA design is executed in parallel for enough cycles to consume the data, and the target outputs are burst read in a channelized fashion. Bursting provides for less overhead to send and receive large amounts of data from the FPGA. However, burst mode is only supported through MATLAB® script-based hardware co-simulation of the Hardware Co-Simulation target and is not used within Simulink. Exhaustive data vectors can be scripted to test the functionality of the co-simulation target, and an example script is returned as part of the compilation. Non-burst mode has lower performance but allows a compiled co-simulation block to be used within Simulink in place of the original System Generator design hierarchy.

***Note*:** Hardware co-simulation does not support designs which contain multiple clocks.

Board support allows two types of physical interfaces to communicate with the co-simulation target: JTAG and Ethernet. JTAG-based communication is available for most of the JTAG aware boards that exist as a project target in the Vivado® tool suite. Boards from Xilinx partners are available and can be downloaded from the partner websites and installed as part of the Vivado Design Suite. Custom boards can also be created as detailed in Appendix A, Board Interface File, in the *Vivado Design Suite User Guide: System-Level Design Entry* (UG895). Setting up board awareness in System Generator and the minimum tags needed in the `board.xml` file are detailed in the section Specifying Board Support in System Generator.

Hardware Co-Simulation compilation targets automatically create a bitstream based on the selected communication interface and associate it to a block.

System Generator currently provides hardware co-simulation support for the following boards:

- JTAG hardware co-simulation - If a board is supported for JTAG hardware co-simulation, the **Hardware Co-Simulation** option for **Compilation** is enabled in the System Generator token dialog box when you perform the procedure described in Compiling a Model for Hardware Co-Simulation. If the **Hardware Co-Simulation** option is grayed out and disabled, you cannot perform JTAG hardware co-simulation on the board.

  This support applies to the following types of boards:

  - Xilinx boards installed as part of your Vivado Design Suite installation,
  - Partner boards, which are available and can be downloaded from the partner websites and installed as part of the Vivado Design Suite,
  - Custom boards, which can be created in the Vivado Design Suite as detailed in Appendix A, Board Interface File, in the *Vivado Design Suite User Guide: System-Level Design Entry* (UG895).

- Point-to-Point Ethernet hardware co-simulation - Point-to-Point Ethernet hardware co-simulation is currently supported for these two boards:

  - Kintex®-7 KC705 Evaluation Platform
  - Virtex®-7 VC707 Evaluation Platform

Compiling a Model for Hardware Co-Simulation

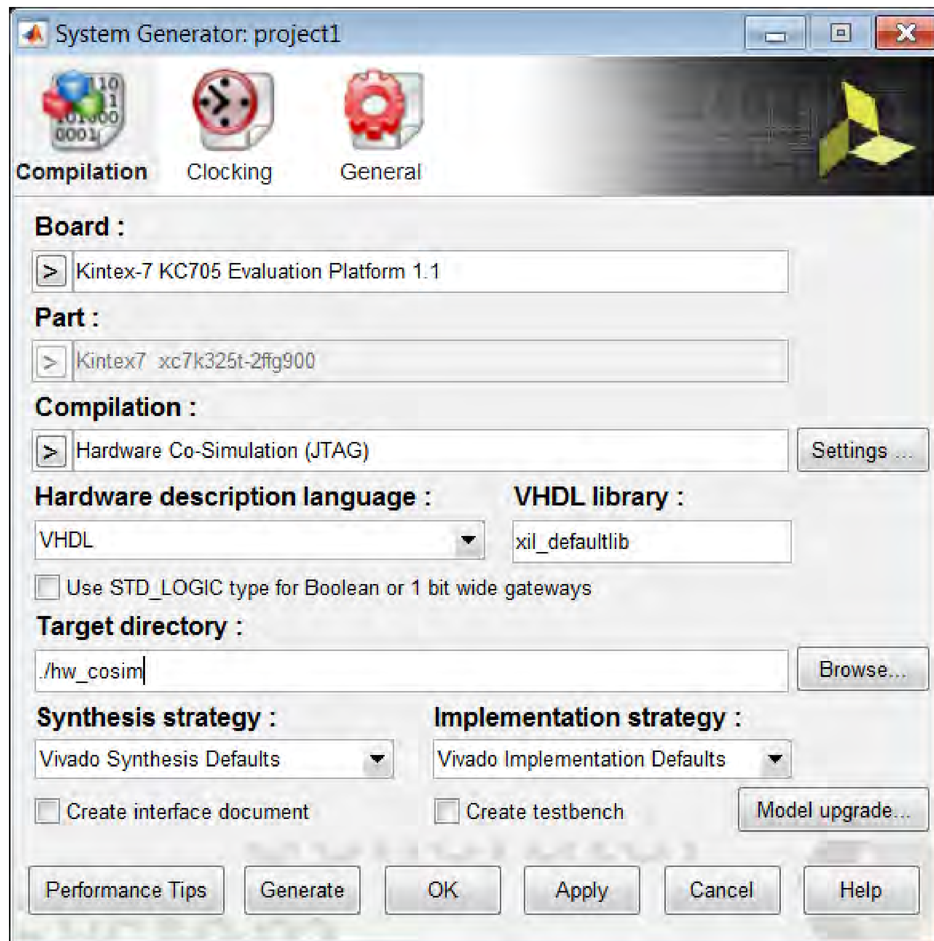
The starting point for hardware co-simulation is the System Generator model or subsystem you would like to run in hardware. A model can be co-simulated if it meets the requirements of the underlying hardware board. The model must include a System Generator token; this block defines how the model should be compiled into hardware.

For information on how to use the System Generator token, see Compiling and Simulating Using the System Generator Token.

To compile your System Generator model for hardware co-simulation, perform the following:

1. Double-click the System Generator token to open the System Generator token dialog box.

Figure 114: **System Generator Token Dialog Box**



2. In the **Compilation** tab, select a **Board** and a version of the board.

   The boards appearing in the **Board** list are:

   - All of the boards installed as part of the Vivado Design Suite.
   - Any custom boards you have created in the Vivado Design Suite.
   - Any Partner boards you have purchased and enabled in the Vivado Design Suite.

   For a Partner board or a custom board to appear in the **Board** list, you must configure System Generator to access the board files that describe the board. Board awareness in System Generator is detailed in Specifying Board Support in System Generator.

   To compile for hardware co-simulation, you must select a **Board**. You cannot set the **Board** field to **None** and select a **Part** instead of a **Board**.

   When you select a **Board**, the **Part** field displays the name of the Xilinx device on the selected **Board**, and the **Part** setting cannot be changed.

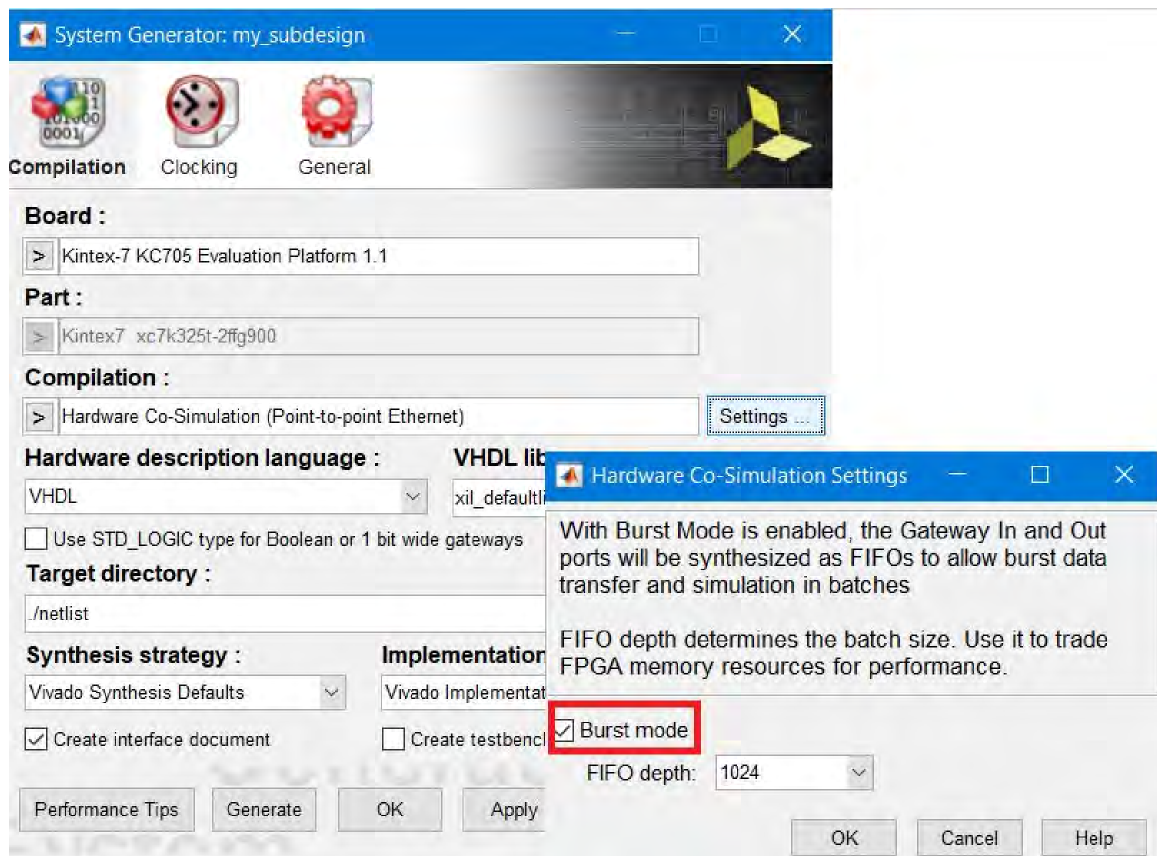
3. In the **Compilation** field, select **Hardware Co-Simulation** and, where applicable, select whether you will perform the hardware co-simulation using the **JTAG** interface or the **Point-to-Point Ethernet** interface.

   If the **Hardware Co-Simulation** option is grayed out and disabled, you cannot perform JTAG hardware co-simulation on the selected board. Currently, you can perform a Point-to-Point Ethernet hardware co-simulation on the KC705 and the VC707 boards from Xilinx.

4. *If you will use burst mode* for a faster hardware co-simulation run, click the **Settings** button next to the **Compilation** field, select **Burst mode**, and enter a **FIFO depth** for the burst mode operation. Then click **OK** to close the Hardware Co-Simulation Settings dialog box.

   For a description of the burst mode, see Burst Data Transfers for Hardware Co-Simulation.

Figure 115: **Burst Mode**



IMPORTANT! *To perform a burst mode hardware co-simulation, you must create a test bench by checking the* ***Create Testbench*** *box in the System Generator token dialog box.*

5. If you want to create a test bench as part of the compilation, select the **Create Testbench** option.

If you select **Create Testbench**, the compilation will automatically create an example test bench for you. You can also create your own test bench for hardware co-simulation (see M-Code Access to Hardware Co-Simulation).

6. Click the **Generate** button.

   The code generator produces an FPGA configuration bitstream for your design that is suitable for hardware co-simulation. System Generator not only generates the HDL and netlist files for your model during the compilation process, but it also runs the downstream tools necessary to produce an FPGA configuration file.

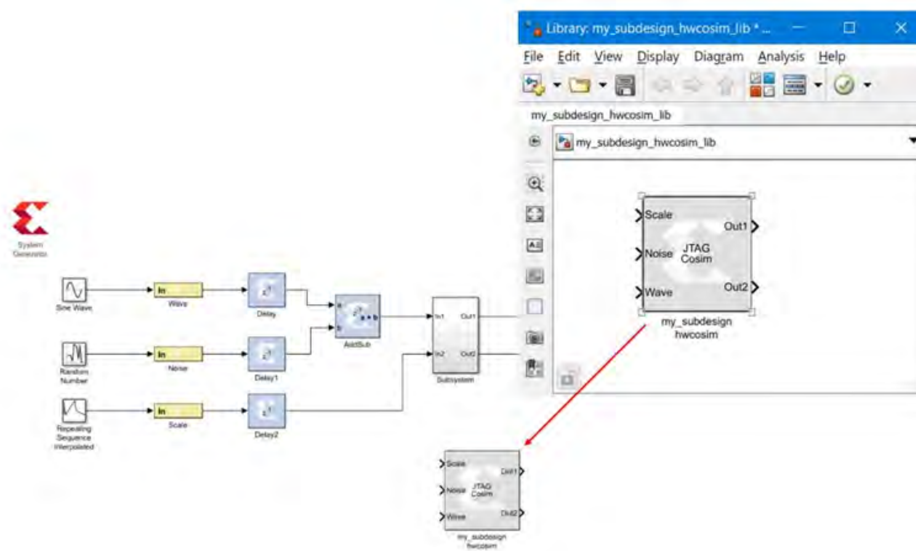
   The configuration bitstream contains the hardware associated with your model, and also contains additional interfacing logic that allows System Generator to communicate with your design using a physical interface between the board and the PC. This logic includes a memory map interface over which System Generator can read and write values to the input and output ports on your design. It also includes any board-specific circuitry that is required for the target FPGA board to function correctly.

   When the Compilation finishes the results are as follows:

   - If you *have not* selected **Burst mode** in step 4 above (standard mode), a **JTAG Cosim** or **Point-to-Point Ethernet** hardware co-simulation block will appear in a separate window. Drag (or Copy and Paste) the Hardware Cosim block into your Simulink model. The Hardware Cosim block will enable you to perform hardware co-simulation from within the Simulink window.

     For a description of the hardware co-simulation block, see Hardware Co-Simulation Blocks.

Figure 116: **Hardware Co-Simulation Library Block**



If you selected the **Create Testbench** option for compilation, an M-Code HWCosim example test bench will also be generated (see M-Code Access to Hardware Co-Simulation) by the compilation. You can use this test bench to perform hardware co-simulation, or customize this test bench to develop a test bench of your own.

- If you *have* selected **Burst mode** in step 4 above (burst mode), no hardware co-simulation block will appear. When you perform the burst mode co-simulation, you will use the MATLAB® M-code test bench placed in the target directory during compilation.
  - If you compiled the top-level design the test bench will be named:

    ```
    <design_name>_hwcosim_test.m
    ```

  - If you compiled a subsystem of the design the test bench will be named:

    ```
    <design_name>_<sub_system>_hwcosim_test.m
    ```

  The compilation has prepared the Simulink model for performing hardware co-simulation.

  To perform the hardware co-simulation, proceed as follows:

  - To perform the standard (non-burst mode) hardware co-simulation, see Performing Standard Hardware Co-Simulation.
  - To perform the burst mode hardware co-simulation, see Performing Burst Mode Hardware Co-Simulation.

Performing Standard Hardware Co-Simulation

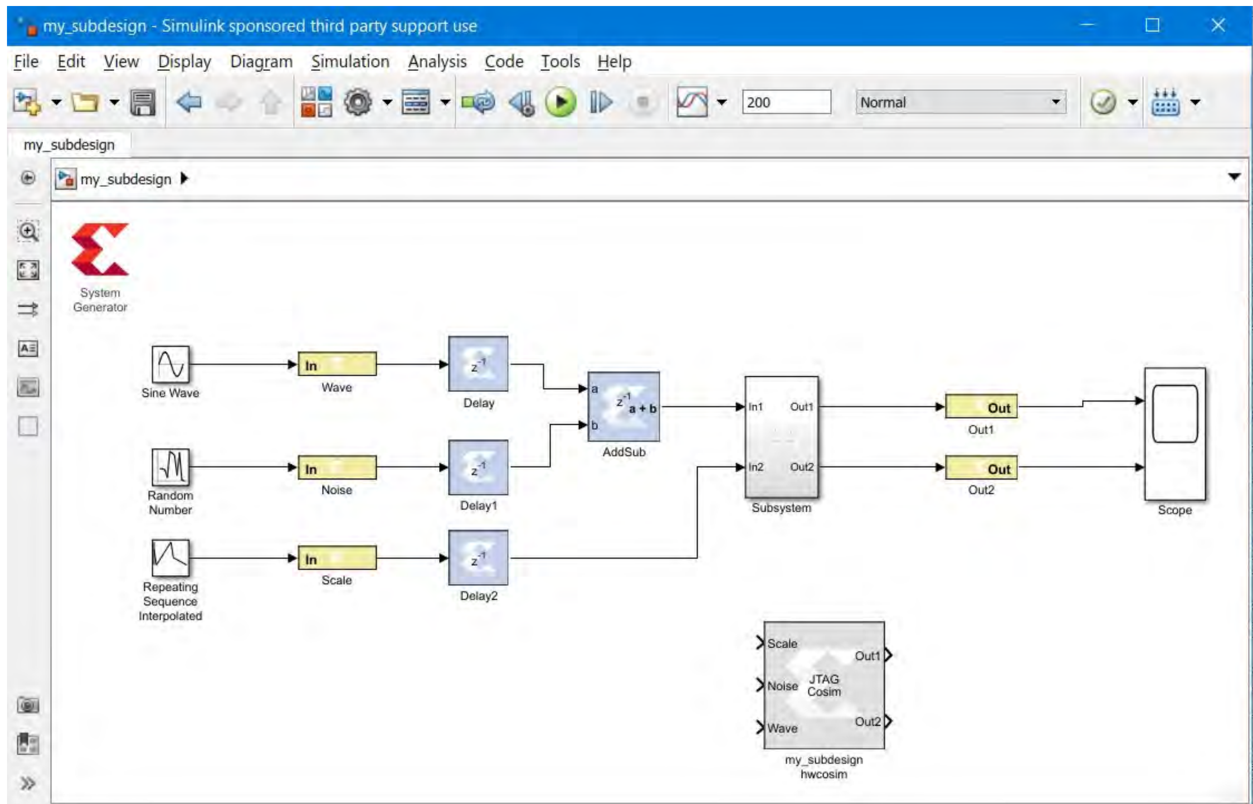
If you are performing the standard (non-burst mode) hardware co-simulation, your Simulink model will contain a JTAG or Point-to-Point Ethernet hardware co-simulation block. This block was created automatically when System Generator finished compiling your design into an FPGA bitstream (see Compiling a Model for Hardware Co-Simulation). The block is stored in a Simulink library with this file name:

```
<design_name>_hwcosim_lib.slx
```

The hardware co-simulation block was moved into your Simulink model at the end of the compilation procedure. In the following procedure, you will have to wire up this block in your Simulink model to perform hardware co-simulation.

Note: If your board contains a Zynq® SoC device, you must install the Vitis™ unified software platform with the Vivado® Design Suite to perform hardware co-simulation.

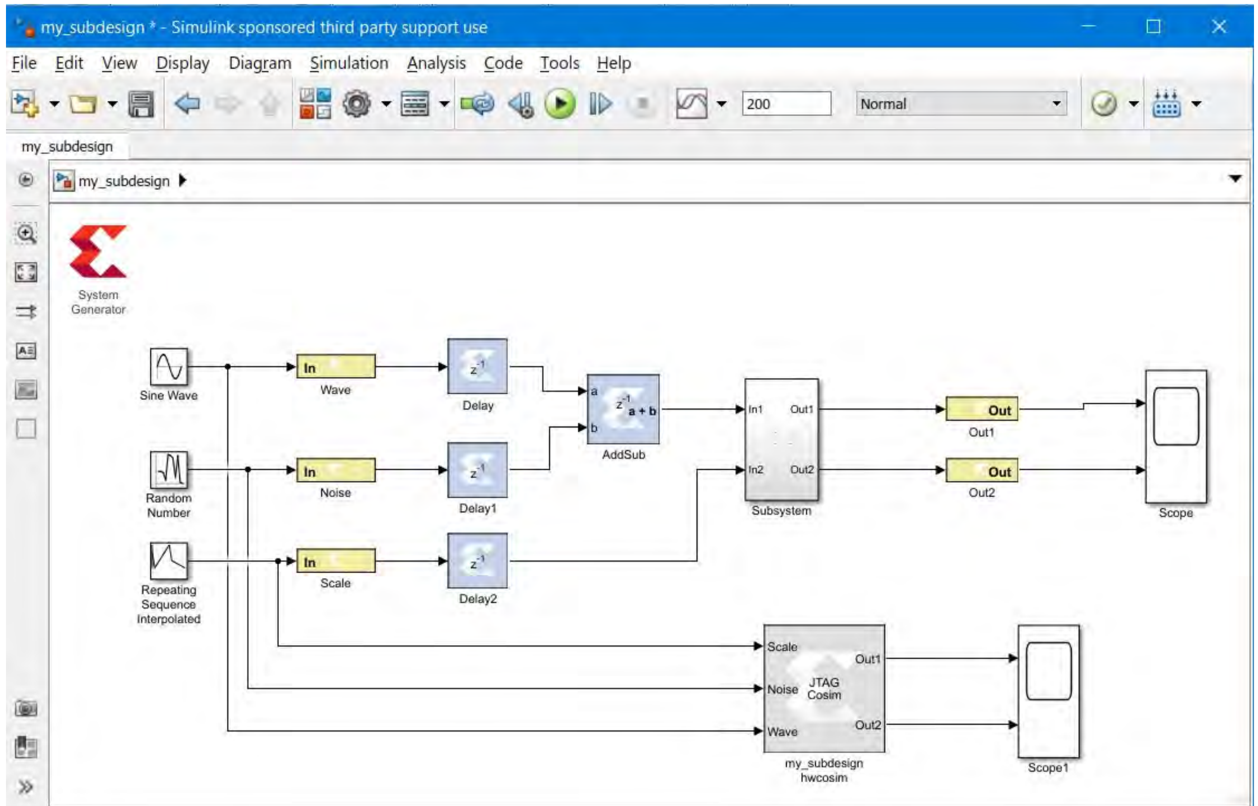
Figure 117: **Hardware Co-Simulation Block**



To perform the standard hardware co-simulation:

1. Connect the hardware co-simulation block to the Simulink blocks that supply its inputs and receive its outputs.

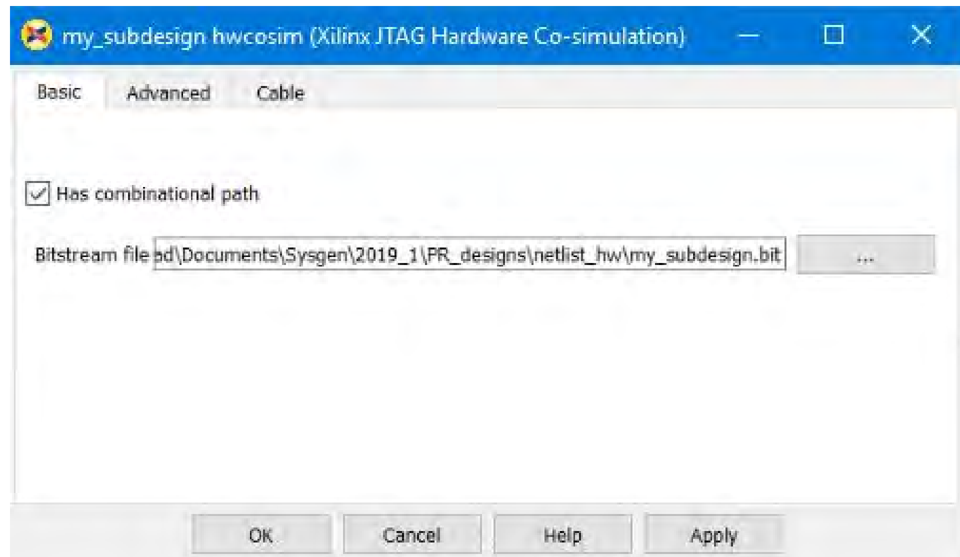
Figure 118: **Connect to Simulink Blocks**



2. Double-click the hardware co-simulation block to display the properties dialog box for the block.

   ***Note*:** There are different block properties for JTAG hardware co-simulation and for Point-to-Point Ethernet hardware co-simulation.

Figure 119: **Hardware Co-Simulation Library Block Properties**



3. Fill out the block parameters in the properties dialog box.

   The properties are described in Block Parameters for the JTAG Hardware Co-Simulation Block or Block Parameters for the Ethernet Hardware Co-Simulation Block.

4. Set up the board for performing hardware co-simulation.

   - For JTAG hardware co-simulation, you will connect a cable to the board's JTAG port.

     For a description of the setup procedure for a JTAG hardware co-simulation, using a KC705 board as an example, see Setting Up a KC705 Board for JTAG Hardware Co-Simulation.

   - For Point-to-Point Ethernet hardware co-simulation, you will connect a cable to the board's JTAG port and another cable to the board's Ethernet port. When you perform the hardware co-simulation, the Xilinx device on the board is programmed using the JTAG port, and the programmed device is then simulated using the Ethernet port.

     For a description of the board setup procedure for a Point-to-Point Ethernet hardware co-simulation, using a KC705 or VC707 board as an example, see Setting Up a KC705 Board for Point-to-Point Ethernet Hardware Co-Simulation or Setting Up a VC707 Board for Point-to-Point Ethernet Hardware Co-Simulation.

5. If you are performing Point-to-Point Ethernet hardware co-simulation:

   a. Set up the Local Area Network on the PC to allow you to perform hardware co-simulation.

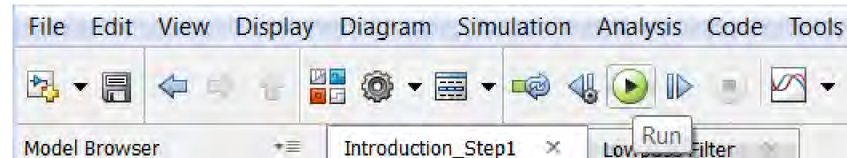
      This procedure is described in Setting Up the Local Area Network on the PC.

   b. If your PC is operating behind a firewall, disable the firewall while the hardware co-simulation runs.

c. Optionally, disable any virus protection program running on the PC while the hardware co-simulation runs.

6. In the Simulink model, simulate the model and the hardware by selecting **Simulation → Run** or clicking the **Run** simulation button.

Figure 120: **Run Button**



Running the simulation will simulate both the System Generator design (or subsystem) in your Simulink model and the Xilinx device on your target board. You can then examine the results of the two simulations and compare the results to determine if the design implemented in hardware will operate as expected.

Performing Burst Mode Hardware Co-Simulation

To perform the burst mode hardware co-simulation, you will execute the MATLAB M-code test bench that was generated automatically during compilation (see Compiling a Model for Hardware Co-Simulation).

This test bench resides in the **Target directory** specified when the design was compiled for the hardware co-simulation compilation target.

The test bench is named as follows:

- If you compiled the top-level design the test bench will be named:

```
<design_name>_hwcosim_test.m
```

- If you compiled a subsystem of the design the test bench will be named:

```
<design_name>_<sub_system>_hwcosim_test.m
```

***Note*:** If your board contains a Zynq® SoC device, you must install the Vitis™ unified software platform with the Vivado® Design Suite to perform hardware co-simulation.

To perform burst mode hardware co-simulation:

1. Set up the board for performing hardware co-simulation.
   - For JTAG hardware co-simulation, you will connect a cable to the board's JTAG port.

For a description of the setup procedure for a JTAG hardware co-simulation, using a KC705 board as an example, see Setting Up a KC705 Board for JTAG Hardware Co-Simulation.

- For Point-to-Point Ethernet hardware co-simulation, you will connect a cable to the board's JTAG port and another cable to the board's Ethernet port. When you perform the hardware co-simulation, the Xilinx device on the board is programmed using the JTAG port, and the programmed device is then simulated using the Ethernet port.

2. If you are performing Point-to-Point Ethernet hardware co-simulation:

    a. Set up the Local Area Network on the PC to allow you to perform hardware co-simulation.

       This procedure is described in Setting Up the Local Area Network on the PC.

       As part of this procedure, you can specify that the PC's Ethernet adapter can use jumbo frames (frames larger than 1500 bytes) for data transfers. Using jumbo frames can speed up the Point-to-Point Ethernet hardware co-simulation. Jumbo frames are described in Using Jumbo Frames for Point-to-Point Ethernet Hardware Co-Simulation.

    b. If your PC is operating behind a firewall, disable the firewall while the hardware co-simulation runs.

    c. Optionally, disable any virus protection program running on the PC while the hardware co-simulation runs.

3. Run the test bench script from the MATLAB console. To run the test bench script, you can open the MATLAB console, change directory to the **Target directory** and run the script by name.

   The script runs the Simulink model to determine the stimulus data driven to the Xilinx Gateway In blocks (from the other Simulink source blocks or MATLAB variables), and captures the expected output produced by the Xilinx block design (BD), and exports the data to the **Target directory** as these separate data files:

   ```
   <design_name>_<sub_system>_<port_name>.dat
   ```

   The test bench then compares actual to expected outputs.

   If the test fails this will be printed on the console, and the failing comparisons will be listed in this file:

   ```
   <design_name>_<sub_system>_hwcosim_test.result
   ```

M-Code Access to Hardware Co-Simulation

It is possible to programmatically control the hardware created through the System Generator hardware co-simulation flow using MATLAB M-code (M-Hwcosim). The M-Hwcosim interfaces allow for MATLAB objects that correspond to the hardware to be created in pure M-code, independent of the Simulink framework. These objects can then be used to read and write data into hardware. This capability is useful for providing a scripting interface to hardware co-simulation, allowing for the hardware to be used in a scripted test bench or deployed as hardware acceleration in M-code.

For more information on this subject, refer to the topic M-Code Access to Hardware Co-Simulation in the *Vivado Design Suite Reference Guide: Model-Based DSP Design Using System Generator* (UG958).

Setting Up Your Hardware Board

The first step in performing hardware co-simulation is to set up your hardware board. The hardware setup for JTAG hardware co-simulation and Point-to-point Ethernet hardware co-simulation is as follows:

- JTAG Hardware Co-Simulation - For JTAG-based hardware co-simulation, you will connect a cable to the board's JTAG port.

  Consult your board's documentation for the location of the board's JTAG port. Documentation for Xilinx boards can be downloaded from the Boards and Kits page on the Xilinx website.

  For a description of the setup procedure for a JTAG hardware co-simulation, using a KC705 board as an example, see Setting Up a KC705 Board for JTAG Hardware Co-Simulation.

- Point-to-Point Ethernet Hardware Co-Simulation - For Point-to-Point Ethernet hardware co-simulation, you will connect a cable to the board's JTAG port and another cable to the board's Ethernet port. When you perform the hardware co-simulation, the Xilinx device on the board is programmed using the JTAG port, and the programmed device is then simulated using the Ethernet port.

  Consult your board's documentation for the location of the board's JTAG and Ethernet ports. Documentation for Xilinx boards can be downloaded from the Boards and Kits page on the Xilinx website.

  For a description of the setup procedure for a Point-to-Point Ethernet hardware co-simulation, using a KC705 board or a VC707 board as an example, see Setting Up a KC705 Board for Point-to-Point Ethernet Hardware Co-Simulation or Setting Up a VC707 Board for Point-to-Point Ethernet Hardware Co-Simulation.

Setting Up a KC705 Board for JTAG Hardware Co-Simulation

The following procedure describes how to set up the hardware required to run JTAG hardware co-simulation on a KC705 board.

For detailed information about the KC705 board, see the *KC705 Evaluation Board for the Kintex-7 FPGA User Guide* (UG810).

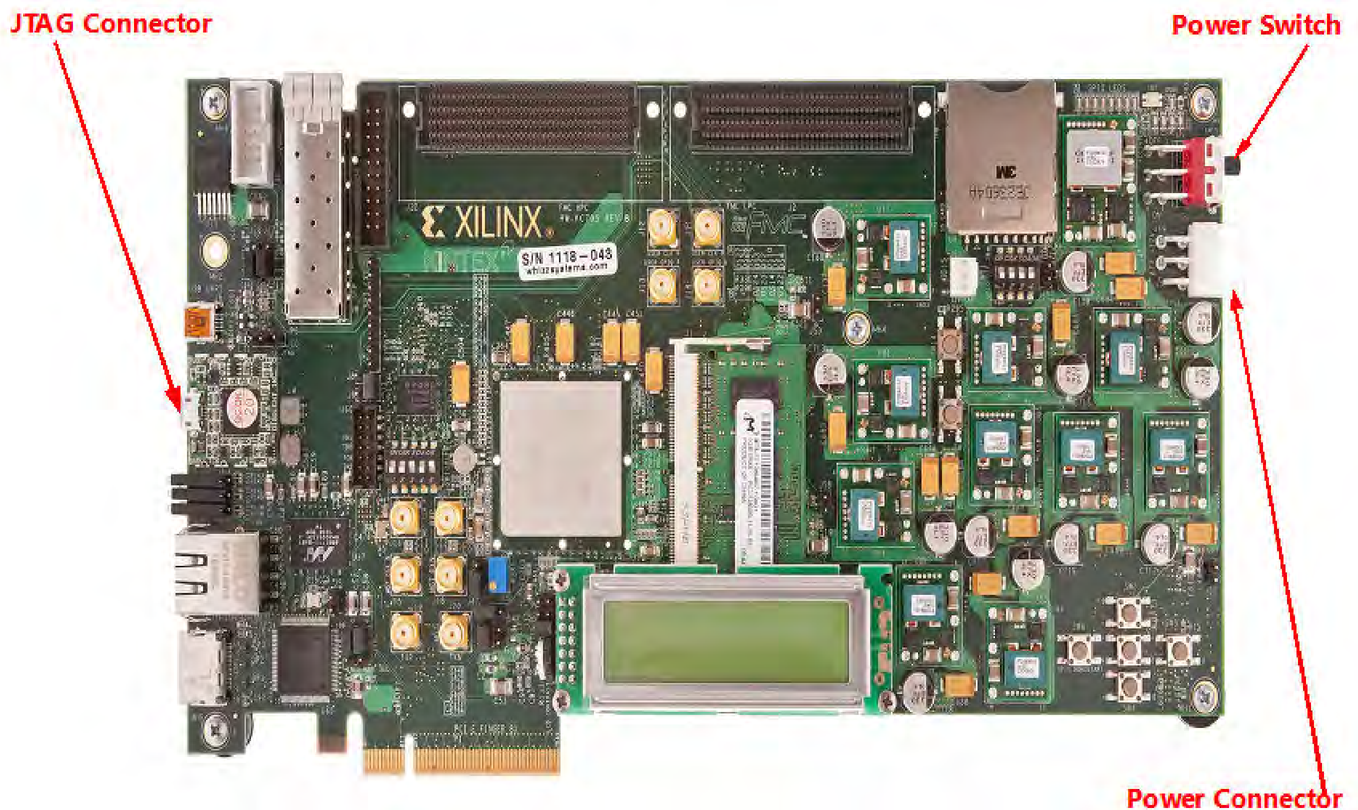
Assemble the Required Hardware

1. Xilinx Kintex-7 KC705 board which includes the following:
    a. Kintex-7 KC705 board
    b. 12V Power Supply bundled with the KC705 kit
    c. Micro USB-JTAG cable

Set Up the KC705 Board

The figure below illustrates the KC705 components of interest in this JTAG setup procedure:

Figure 121: **KC705 Board**



1. Position the KC705 board as shown above.
2. Make sure the power switch, located in the upper-right corner of the board, is in the OFF position.
3. Connect the AC power cord to the power supply brick. Plug the power supply adapter cable into the KC705 board. Plug in the power supply to AC power.
4. Connect the small end of the Micro USB-JTAG cable to the JTAG socket.
5. Connect the large end of the Micro USB-JTAG cable to a USB socket on your PC.
6. Turn the KC705 board Power switch ON.

Setting Up a KC705 Board for Point-to-Point Ethernet Hardware Co-Simulation

The following procedure describes how to install the hardware required to run an KC705 board Point-to-Point Ethernet hardware co-simulation.

For detailed information about the KC705 board, see the *KC705 Evaluation Board for the Kintex®-7 FPGA User Guide* (UG810).

Note: Point-to-Point Ethernet Hardware Co-Simulation requires full-duplex Ethernet operation, including the use of Auto-Negotiation. If you are performing Point-to-Point Ethernet Hardware Co-Simulation through a Network Interface Card (NIC) or a USB-to-Ethernet adapter, the connection will only operate under the following conditions:

- The NIC or USB-to-Ethernet adapter must be connected directly to the board.
- The NIC or USB-to-Ethernet adapter must support the IEEE 802.3ab Gigabit Ethernet standard.
- The NIC or USB-to-Ethernet adapter must support full-duplex Ethernet operation using Auto-Negotiation. Setting the speed directly without Auto-Negotiation will cause the Point-to-Point Ethernet connection to fail.

Assemble the Required Hardware

- Xilinx KC705 board
- Power Supply for the board
- Ethernet Network Interface Card (NIC) for the host PC.
- Ethernet RJ45 Male/Male Cable. (May be a Network or Crossover cable.)
- Digilent USB Cable or the Platform USB Cable to download the bitstream.

Set Up the KC705 Board

Figure 122: **KC705 Board Ethernet Connections**

To set up the KC705 board for Point-to-Point Ethernet hardware co-simulation:

1. Position the KC705 board as shown above.
2. Make sure the power switch, located in the upper-right corner of the board, is in the OFF position.
3. Connect the power cable to the right. Plug in the power supply to AC power.
4. Connect the Digilent USB cable to the top left and the other end to the host PC.
5. Connect the Ethernet cable to the KC705 board to the lower left and the other end to the host PC.
6. Turn the KC705 board Power switch ON.

Setting Up a VC707 Board for Point-to-Point Ethernet Hardware Co-Simulation

The following procedure describes how to install the hardware required to run a VC707 board Point-to-Point Ethernet hardware co-simulation.

For detailed information about the VC707 board, see the *VC707 Evaluation Board for the Virtex-7 FPGA User Guide* (UG885).

Note: Point-to-Point Ethernet Hardware Co-Simulation requires full-duplex Ethernet operation, including the use of Auto-Negotiation. If you are performing Point-to-Point Ethernet Hardware Co-Simulation through a Network Interface Card (NIC) or a USB-to-Ethernet adapter, the connection will only operate under the following conditions:

- The NIC or USB-to-Ethernet adapter must be connected directly to the board.
- The NIC or USB-to-Ethernet adapter must support the IEEE 802.3ab Gigabit Ethernet standard.
- The NIC or USB-to-Ethernet adapter must support full-duplex Ethernet operation using Auto-Negotiation. Setting the speed directly without Auto-Negotiation will cause the Point-to-Point Ethernet connection to fail.

Assemble the Required Hardware

- Xilinx® VC707 board
- Power Supply for the board
- Ethernet Network Interface Card (NIC) for the host PC.
- Ethernet RJ45 Male/Male Cable. (May be a Network or Crossover cable)
- Digilent USB Cable or Platform USB Cable to download the bitstream.

Set Up the VC707 Board

Figure 123: **VC705 Board Ethernet Connections**

To set up the VC707 board for Point-to-Point Ethernet hardware co-simulation:

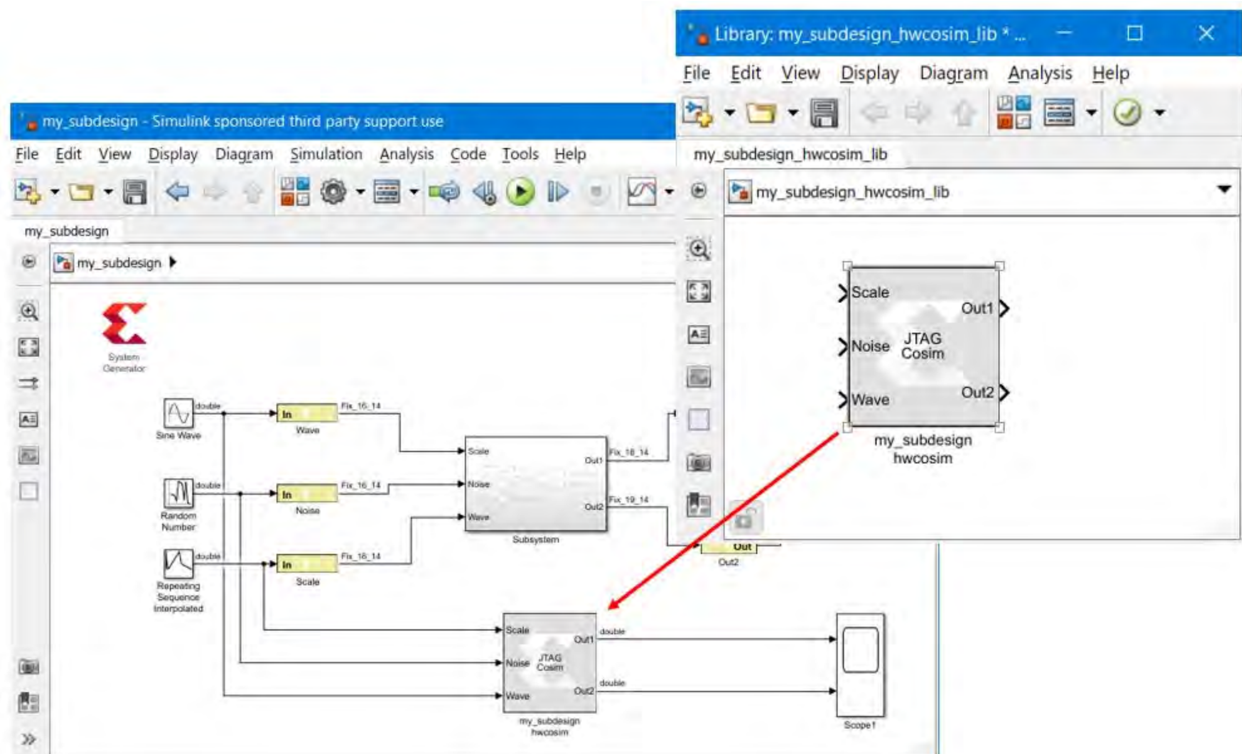
1. Position the VC707 board as shown above.

2. Make sure the power switch, located in the upper-right corner of the board, is in the OFF position.
3. Connect the power cable to the right. Plug in the power supply to AC power.
4. Connect the Digilent USB cable to the top left and the other end to the host PC.
5. Connect the Ethernet cable to the VC707 board to the lower left and the other end to the host PC.
6. Turn the VC707 board Power switch ON.

Hardware Co-Simulation Blocks

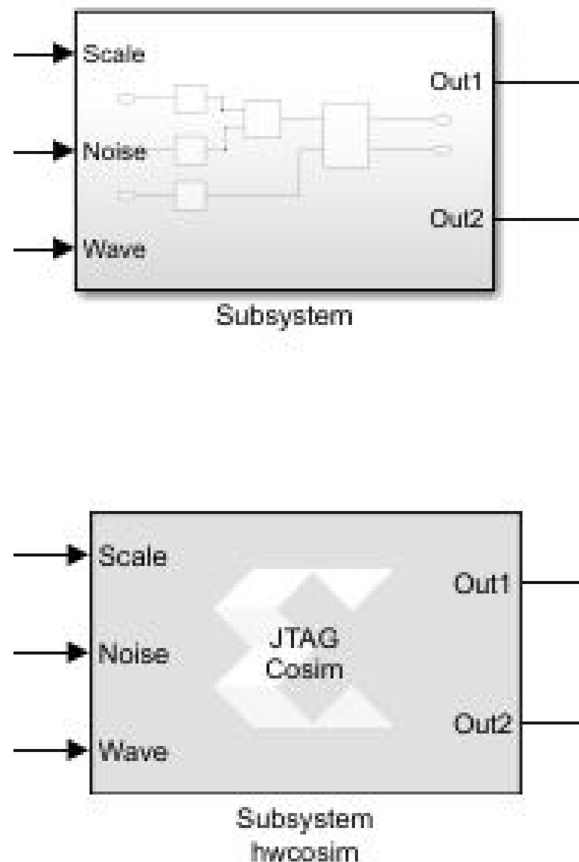
System Generator automatically creates a new hardware co-simulation block once it has finished compiling your design into an FPGA bitstream. It also creates a Simulink library to store the hardware co-simulation block. At this point, you can copy the block out of the library and use it in your System Generator design like any other Simulink or System Generator blocks.

Figure 124: **Hardware Co-Simulation Blocks**



The hardware co-simulation block assumes the external interface of the model or Subsystem from which it is derived. The port names on the hardware co-simulation block match the ports names on the original Subsystem. The port types and rates also match the original design.

Figure 125: **Port Names**



Hardware co-simulation blocks are used in a Simulink design the same way other blocks are used. During simulation, a hardware co-simulation block interacts with the underlying FPGA board, automating tasks such as device configuration, data transfers, and clocking. A hardware co-simulation block consumes and produces the same types of signals that other System Generator blocks use. When a value is written to one of the block's input ports, the block sends the corresponding data to the appropriate location in hardware. Similarly, the block retrieves data from hardware when there is an event on an output port.

hardware co-simulation blocks may be driven by Xilinx® fixed-point signal types, Simulink fixed-point signal types, or Simulink doubles. Output ports assume a signal type that is appropriate for the block they drive. If an output port connects to a System Generator block, the output port produces a Xilinx® fixed-point signal. Alternatively, the port produces a Simulink data type when the port drives a Simulink block directly.

Note: When Simulink data types are used as the block signal type, quantization of the input data is handled by rounding, and overflow is handled by saturation.

Like other System Generator blocks, hardware co-simulation blocks provide parameter dialog boxes that allow them to be configured with different settings. The parameters that a hardware co-simulation block provides depend on the FPGA board the block is implemented for (i.e. different FPGA boards provide their own customized hardware co-simulation blocks).

Block Parameters for the JTAG Hardware Co-Simulation Block

The block parameters dialog box for the JTAG hardware co-simulation block can be invoked by double-clicking the block icon in your Simulink model.

Parameters specific to the block are as follows:

Basic tab

Has combinational path: Select this if your circuit has any combinational paths. A combinational path is one in which a change propagates from input to output without any clock event. There is no latch, flip-flop, or register in the path. Enabling this option causes System Generator to read the outputs immediately after writing inputs, before clocking the design. This ensures that value changes on combinational paths extending from the hardware co-simulation block into the Simulink Model get propagated correctly.

Bitstream file: Specify the FPGA configuration bitstream. By default this field contains the path to the bitstream generated by System Generator during the last **Generate** triggered from the System Generator Token.

Advanced tab

Skip device configuration: When selected, the configuration bitstream will not be loaded into the FPGA or SoC. This option can be used if another program is configuring the device (for example, the Vivado Hardware Manager and the Vivado Logic Analyzer).

Display Part Information: This option toggles the display of the device part information string (for example, **xc7k325tffg900-2** for a Kintex device) in the center of the hardware co-simulation block.

Cable tab

Cable Settings

- **Type:** Currently, **Auto Detect** is the only setting for this parameter. System Generator will automatically detect the cable type.

Block Parameters for the Ethernet Hardware Co-Simulation Block

The block parameters dialog box for the Ethernet hardware co-simulation block can be invoked by double-clicking the block icon in your Simulink model.

Parameters specific to the block are as follows:

Basic tab

Clocking

- **Clock source:** Specifies the clocking mode (Single stepped or Free running) used to synchronize the System Generator hardware co-simulation block with its associated FPGA or SoC hardware. For a description of the two clock sources, see Clocking Modes.

Has combinational path: Select this if your circuit has any combinational paths. A combinational path is one in which a change propagates from input to output without any clock event. There is no sequential logic (latches, flip-flops, or registers) in the path. Enabling this option causes System Generator to read the outputs immediately after writing inputs, before clocking the design. This ensures that value changes on combinational paths extending from the hardware co-simulation block into the Simulink model get propagated correctly.

Bitstream file: Specify the FPGA configuration bitstream. By default this field contains the path to the bitstream generated by System Generator during the last **Generate** triggered from the System Generator token.

Advanced tab

Skip device configuration: When selected, the configuration bitstream will not be loaded into the FPGA or SoC. This option can be used if another program is configuring the device (for example, the Vivado Hardware Manager and the Vivado Logic Analyzer).

Display Part Information: This option toggles the display of the device part information string (for example, **xc7k325tffg900-2** for a Kintex device) in the center of the hardware co-simulation block.

Ethernet tab

Host Interface

Ethernet Interface: This drop-down list contains all the Ethernet interfaces detected in the host computer. Select the interface which is connected to the target board. The selected interface must be configured correctly to perform the Point-to-Point Ethernet hardware co-simulation. For a description of the host interface configuration, see Setting Up the Local Area Network on the PC.

Refresh button: the **Refresh** button gives you the ability to re-enumerate the available Ethernet interfaces. The button can be used to display Ethernet interfaces that can be hot-plugged (for example, USB-to-Ethernet adapters) or interfaces that are disabled when you open the block parameters dialog box but are enabled afterwards.

FPGA Interface

MAC Address: This is the Ethernet MAC address assigned to the target board. If left blank, the default value is da:02:03:04:05:06. This value should never be the same as the host's MAC address.

Configuration tab

Cable

- **Type:** Currently, **Auto Detect** is the only setting for this parameter. System Generator will automatically detect the cable type.

Configuration timeout (ms): Specify the timeout value for the initial Ethernet handshake after configuration.

Hardware Co-Simulation Clocking

If you are performing a standard hardware co-simulation, you will have to select a clocking mode when you configure the co-simulation block. included in your Simulink model.

Clocking Modes

There are several ways in which a System Generator hardware co-simulation block can be synchronized with its associated FPGA hardware. In single-step clock mode, the FPGA is in effect clocked from Simulink, whereas in free-running clock mode, the FPGA runs off an internal clock, and is sampled asynchronously when Simulink wakes up the hardware co-simulation block.

Single-Step Clock

In single-step clock mode, the hardware is kept in lock step with the software simulation. This is achieved by providing a single clock pulse (or some number of clock pulses if the FPGA is over-clocked with respect to the input/output rates) to the hardware for each simulation cycle. In this mode, the hardware co-simulation block is bit-true and cycle-true to the original model.

Because the hardware co-simulation block is in effect producing the clock signal for the FPGA hardware only when Simulink awakes it, the overhead associated with the rest of the Simulink model's simulation, and the communication overhead (e.g. bus latency) between Simulink and the FPGA board can significantly limit the performance achieved by the hardware. As long as the amount of computation inside the FPGA is significant with respect to the communication overhead (e.g. the amount of logic is large, or the hardware is significantly over-clocked), the hardware will provide significant simulation speed-up.

Free-Running Clock

In free-running clock mode, the hardware runs asynchronously relative to the software simulation. Unlike the single-step clock mode, where Simulink effectively generates the FPGA clock, in free-running mode, the hardware clock runs continuously inside the FPGA itself. In this mode, simulation is not bit and cycle true to the original model, because Simulink is only sampling the internal state of the hardware at the times when Simulink awakes the hardware co-simulation block. The FPGA port I/O is no longer synchronized with events in Simulink. When an event occurs on a Simulink port, the value is either read from or written to the corresponding port in hardware at that time. However, since an unknown number of clock cycles have elapsed in hardware between port events, the current state of the hardware cannot be reconciled to the original System Generator model. For many streaming applications, this is in fact highly desirable, as it allows the FPGA to work at full speed, synchronizing only periodically to Simulink.

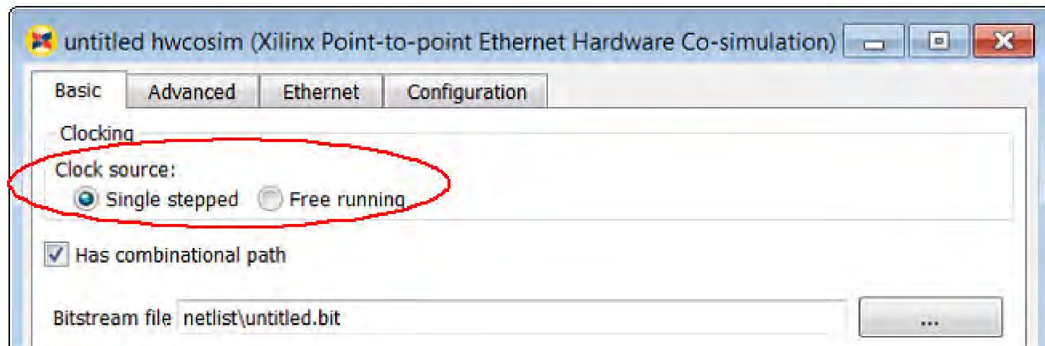
In free-running mode, you must build explicit synchronization mechanisms into the System Generator model. A simple example is a status register, exposed as an output port on the hardware co-simulation block, which is set in hardware when a condition is met. The rest of the System Generator model can poll the status register to determine the state of the hardware.

Selecting the Clock Mode

Not every hardware board supports a free-running clock. However, for those that do, the parameters dialog box for the hardware co-simulation block provides a means to select the desired clocking mode. You may change the co-simulation clocking mode before simulation starts by selecting either the **Single stepped** or **Free running** radio button for **Clock Source** in the parameters dialog box.

***Note*:** The clocking options available to a hardware co-simulation block depend on the FPGA board being used (i.e., some boards may not support a free-running clock source, in which case it is not available as a dialog box parameter).

Figure 126: **Single Stepped Button**



For a description of a way to programmatically turn on or off a free-running clock using M-Hardware Cosim, see the description of the Run operation under M-Hwcosim MATLAB Class in the *Vivado Design Suite Reference Guide: Model-Based DSP Design Using System Generator* (UG958).

Point-to-Point Ethernet Hardware Co-Simulation

The following affect hardware co-simulation performed through a Point-to-Point Ethernet interface:

- Setting Up the Local Area Network on the PC
- Point-to-Point Ethernet Hardware Co-Simulation on Linux
- Using Jumbo Frames for Point-to-Point Ethernet Hardware Co-Simulation

Setting Up the Local Area Network on the PC

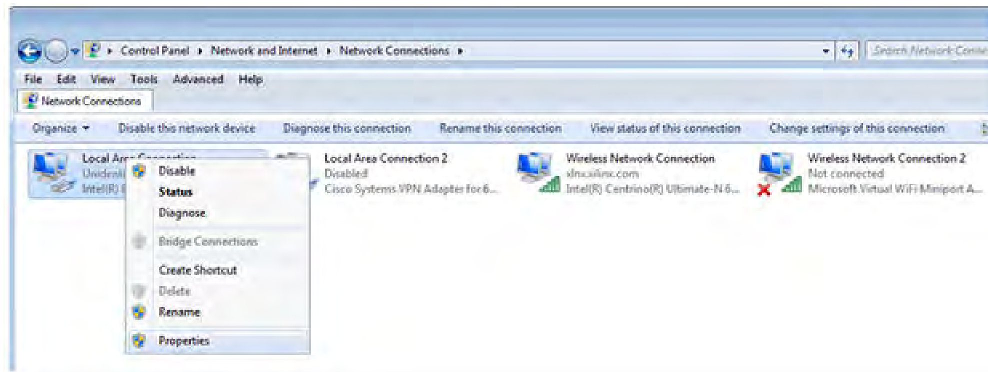
For Ethernet Point-to-Point hardware co-simulation, you are required to have a 10/100 Fast Ethernet, or a Gigabit Ethernet Adapter on your PC. To configure the settings do the following:

1. From the Windows **Start** menu, select **Control Panel**, then under **Network and Internet**, click **View network status and tasks**. On the left hand side, click **Change Adapter settings**.

Figure 127: **Change Adapter Settings**

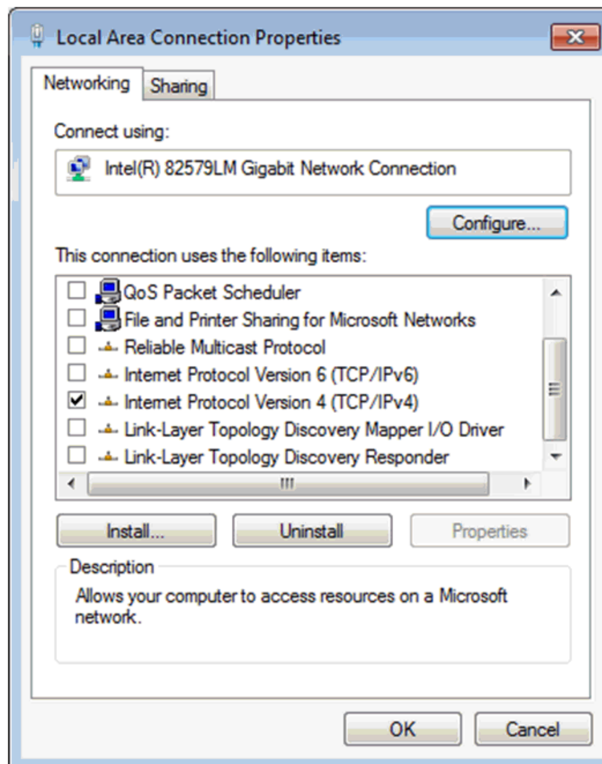
2. Right-click **Local Area Connection**, then select **Properties**.

Figure 128: **Local Area Connection Properties**



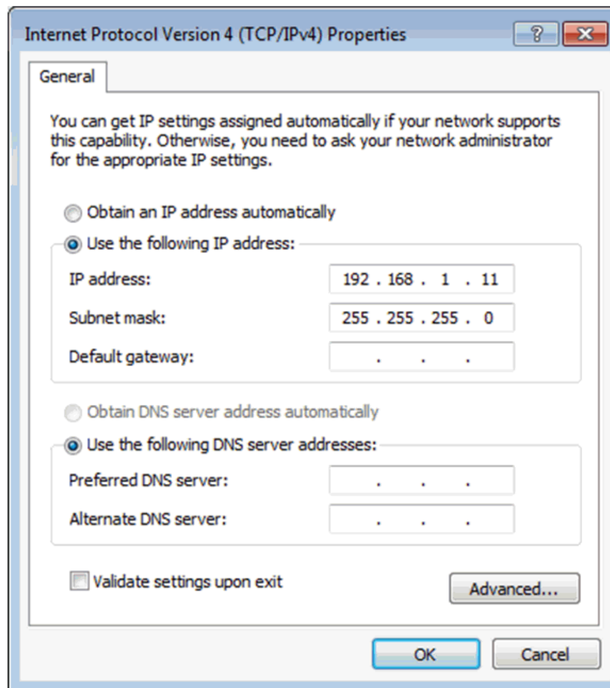
3. In the Local Area Connection Properties dialog box, enable **Internet Protocol Version 4 (TCP/IPv4)**. Disable everything else.

Figure 129: **Internet Protocol**



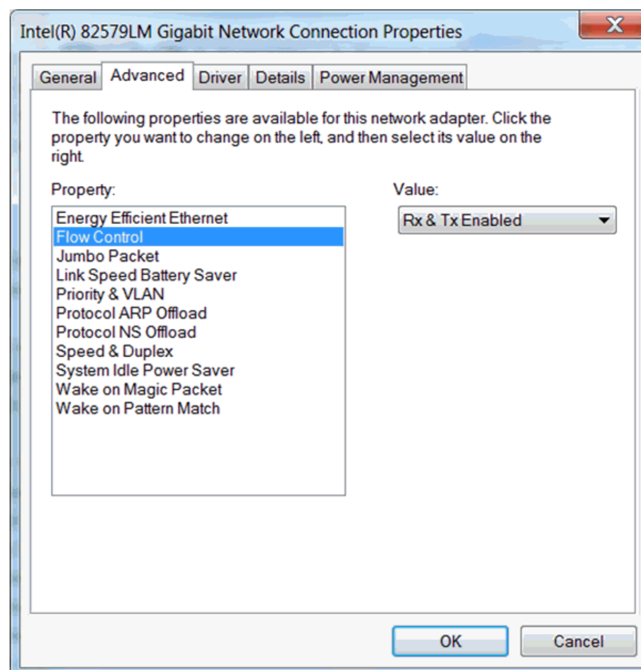
4. Select **Internet Protocol Version 4 (TCP/IPv4)**, and click **Properties**. In the Internet Protocol Version 4 (TCP/IPv4) dialog box, select **Use the following IP address**, and set the **IP Address** to 192.168.1.11, and **Subnet mask** to 255.255.255.0. Click **OK**.

Figure 130: **IP Address**



5. In the Local Area Connection Properties dialog box, click **Configure**. Click the **Advanced** tab. Click **Flow Control**. Set the **Value** to **Rx & Tx Enabled**.

Figure 131: **Flow Control**



6. If you will want to speed up the Ethernet Point-to-Point hardware co-simulation by using jumbo frames (that is, packets larger than 1500 bytes), click **Jumbo Packet** and set the **Value** to the desired frame size. See Using Jumbo Frames for Point-to-Point Ethernet Hardware Co-Simulation for a description of jumbo frames.
7. Click **OK** to close the Properties dialog box.

Using Jumbo Frames for Point-to-Point Ethernet Hardware Co-Simulation

Jumbo frames are Ethernet frames that are larger than 1500 bytes. You can speed up the data transfers needed for Point-to-Point Ethernet hardware co-simulation by specifying that the Ethernet adapter can use jumbo frames for these data transfers.

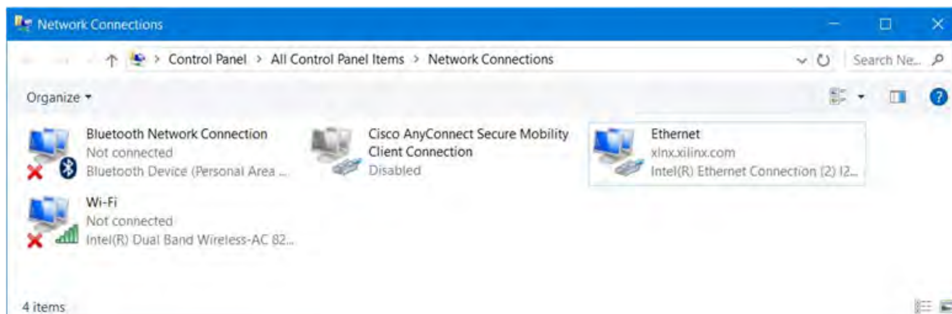
You can enable jumbo frames when you first set up the local area network for Point-to-Point Ethernet hardware co-simulation (see Setting Up the Local Area Network on the PC).

If you have already set up the local area network, you can enable the use of jumbo frames afterwards as follows:

1. In the Windows Control Panel, select **Network and Internet → Network and Sharing Center → Change Adapter Settings**.

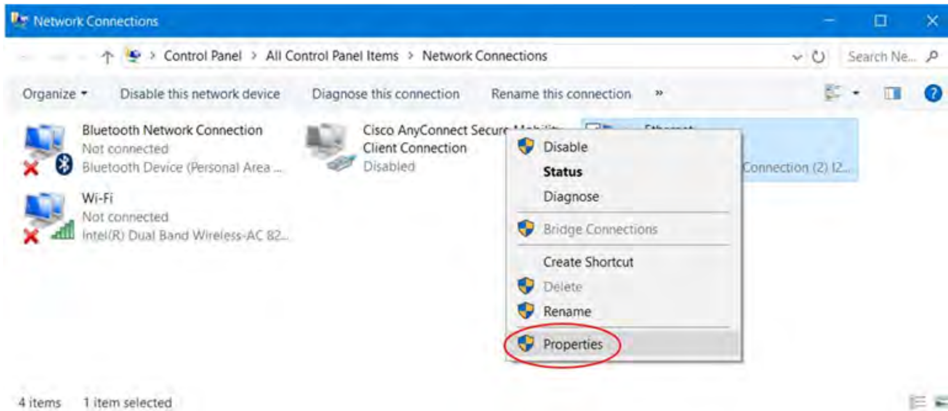
   The Network Connections folder opens.

   Figure 132: **Network Connections Folder**

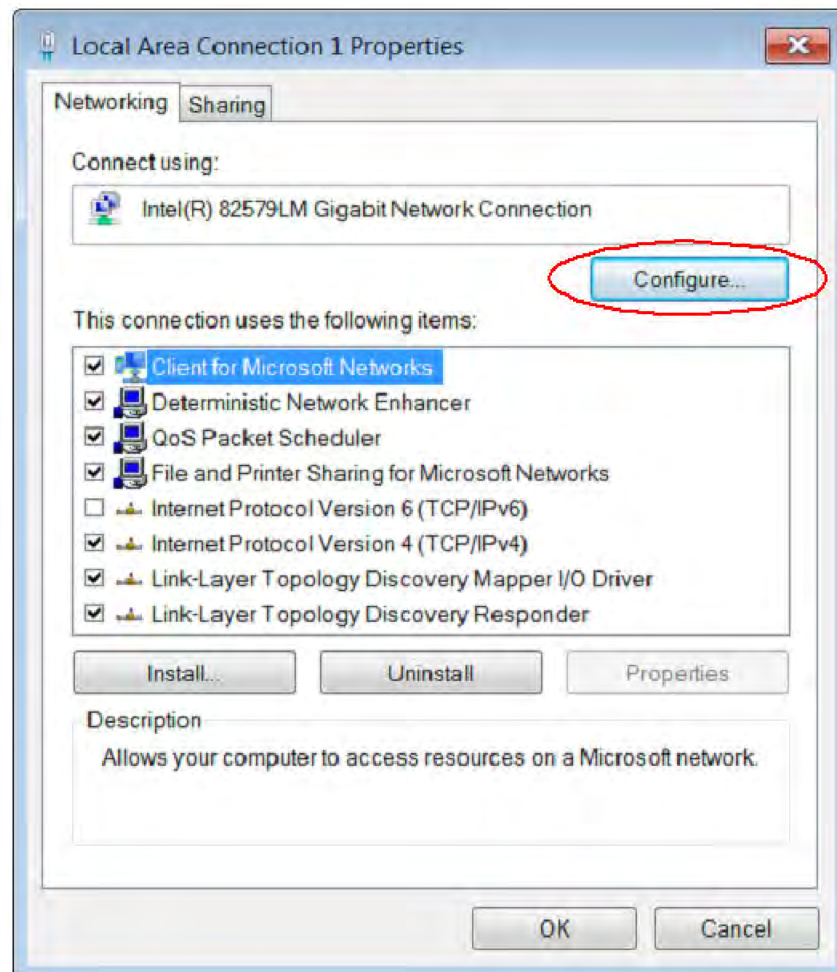
2. In the Network Connections folder, right-click the **Local Area Connection** you will use for the Ethernet hardware co-simulation and select **Properties** in the right-click menu.

Figure 133: **Properties**



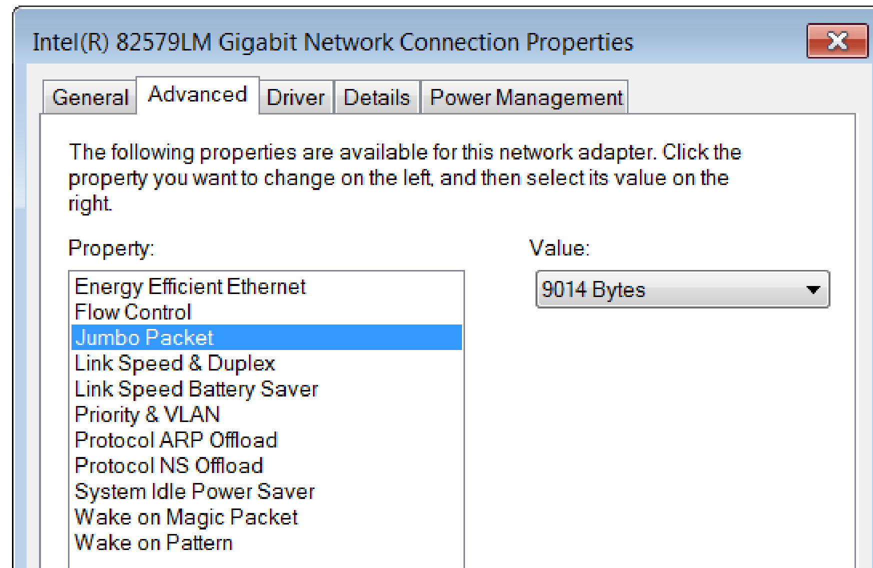
3. In the Properties dialog box for your network connection, click the **Configure** button.

Figure 134: **Configure Button**



4. In the Advanced tab of the Properties dialog box for your adapter, select the **Jumbo Packet** entry and enable using jumbo packets by setting the **Value** to the desired frame size.

Figure 135: **Jumbo Pocket**



5. Click **OK** to close the Properties dialog box for your adapter.
6. Click **OK** to close the Properties dialog box for your network connection.

Point-to-Point Ethernet Hardware Co-Simulation on Linux

To perform Point-to-Point Ethernet hardware co-simulation on Linux, you need to have sudo access on the Linux Machine. System Generator has to be launched as a sudo user. In case you do not have multiple Network Interface cards on your machine, a Network switch can be used.

Burst Data Transfers for Hardware Co-Simulation

Hardware co-simulation (HWCosim) is a methodology by which a user can offload, either partially or whole, the most compute intensive portion of a model into the actual target FPGA platform. The host system provides the stimulus to the model via the co-simulation interface (typically JTAG and/or point-to-point Ethernet) and post-processes the response. This methodology is useful for validating the correctness of the generated hardware design on the target platform itself, as well as for speeding up the simulation time during verification of the model in a hardware co-verification scenario.

MATLAB/Simulink in conjunction with System Generator for DSP currently supports two variants of HWCosim: GUI-based and MATLAB M-script based. The first is run under the control of the Simulink scheduler, and can only progress one clock cycle at a time, due to the potential for feedback loops in the model.

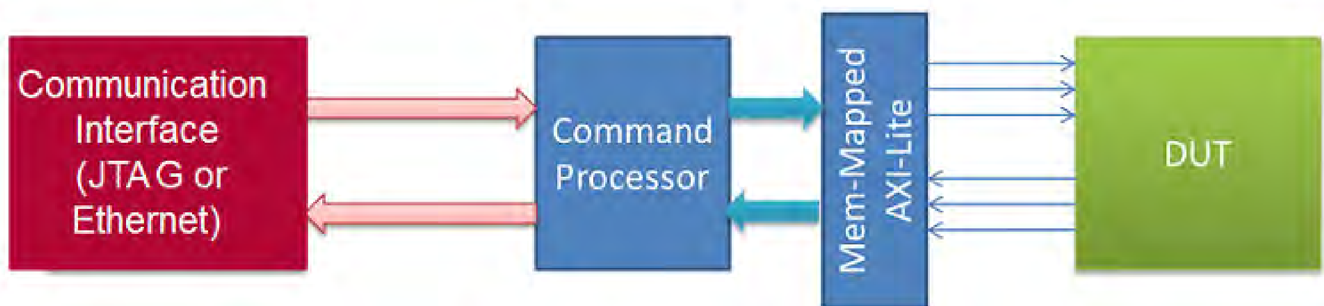
The second variant is MATLAB M-script based simulation under System Generator control (M-HWCosim), which is commonly used in testbenches produced as collateral during the bitstream generation from the System Generator token. These testbenches are typically feedback-free and come with a-priori known input that can be transferred to the device in larger batches.

Previous generations of System Generator for DSP (Vivado) implemented only a basic variant of HWCosim, which did not harness the full performance potential of the interface. Command and response packets were sent in single-cycle batches, only utilizing a small part of the available bandwidth. This leaves a lot of performance on the table which the latest version of System Generator for DSP aims to reclaim.

Hardware Co-Simulation Overview

A high-level overview of hardware co-simulation (HWCosim) is given in the figure below. At the center of it is the device under test (DUT). The DUT is typically a piece of IP that is developed and tested within a Simulink test framework providing the stimulus and receiving (and potentially evaluating) the response. In order to allow for Simulink to communicate with the DUT it needs to be embedded into the HWCosim wrapper consisting of the following components:

Figure 136: **Hardware Co-Simulation Flow**

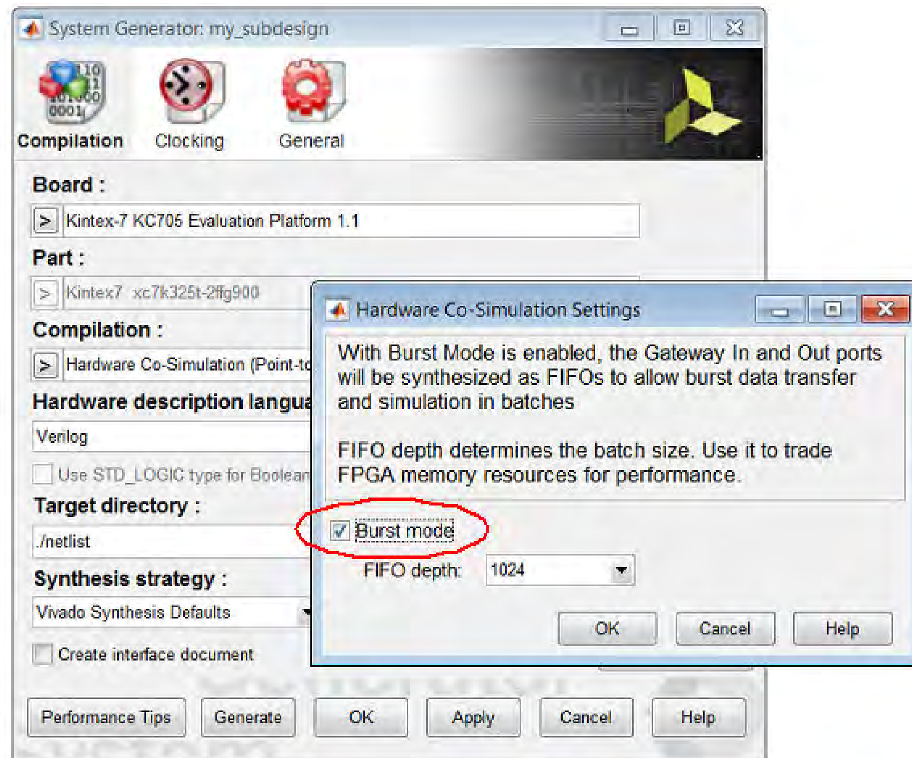


- **Communication interface (JTAG or Ethernet):** Used for communications with the host PC, receiving the command messages and sending responses.
- **Command processor:** Command messages are parsed and executed.
- **Memory-mapped AXI4-Lite register bank:** Use `write` commands to set up the stimulus data in the register map, which is driving the inputs to the DUT. Similarly, use `read` commands to query the memory-mapped DUT outputs. Finally, use a `run(x)` command to the memory-mapped clock control register to trigger exactly "x" clock pulses on the DUT's clock input. Alternatively, use `run(inf)` to start the free-running clock mode and `run(0)` to turn the clock off.

Burst Data Transfer Mode

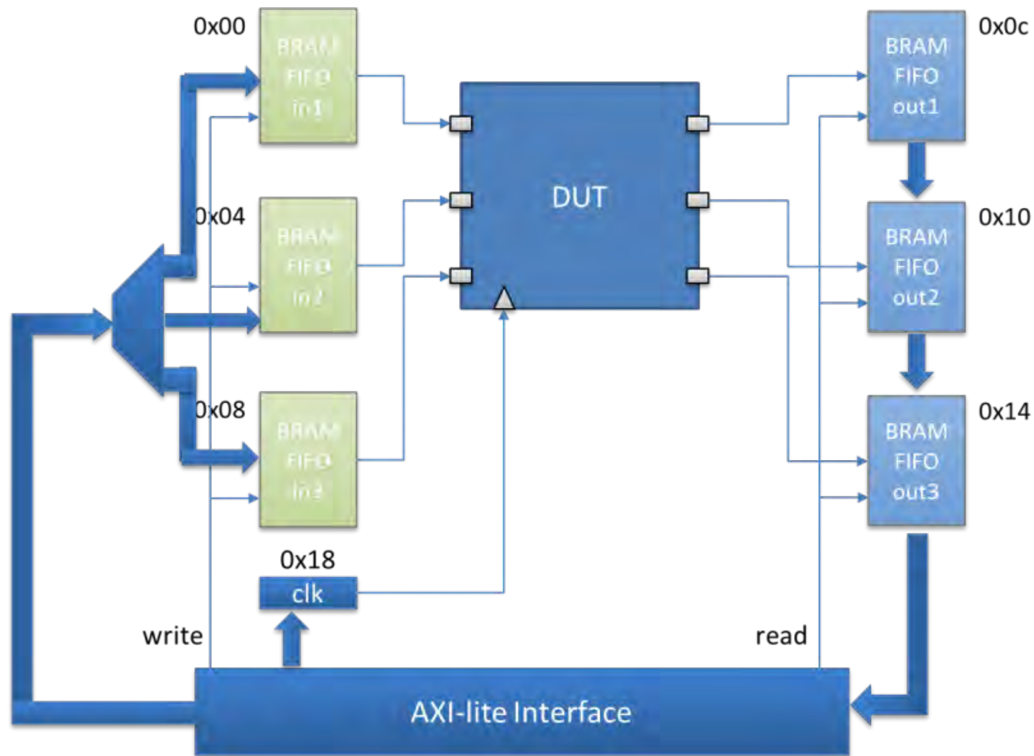
If you enable burst data transfer mode in the System Generator token (**Compilation → Settings → Burst mode**), the non-clock input and output registers will be replaced with "n"-entry FIFOs. You can select "n" (**FIFO depth**), which is useful for trading off performance versus FPGA block RAM resource use.

Figure 137: **Burst Mode**



Enabling **Burst mode** allows the M-HWCosim scheduler to "burst write" a time-sequence of "n" values into each input FIFO, run the clock for a number of cycles determined by the rate of input/output ports and the FIFO depths, and capture the resulting output in the output FIFOs. After the batch has been run, the scheduler proceeds to "burst read" the contents of the output FIFOs into a MATLAB array, where it can be checked against expected data.

Figure 138: **Burst Mode Flow**



This batch processing of time samples allows to better pack data into JTAG sequences or Point-to-Point Ethernet frames up to the maximum "jumbo" frame size, thereby significantly reducing overhead.

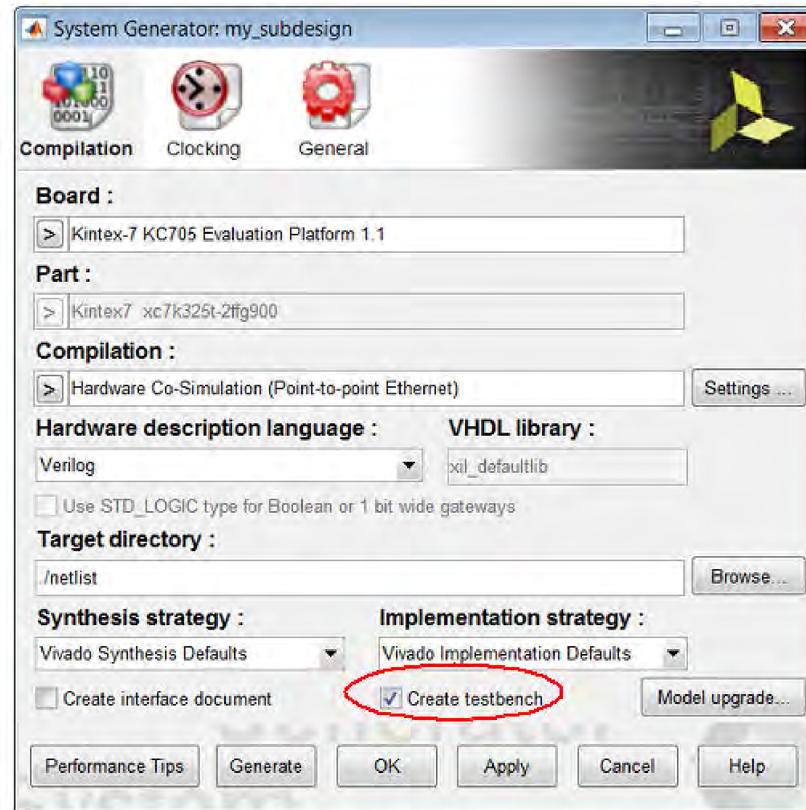
How to Use Burst Data Transfer Mode

The simplest way for you to start using burst data transfer mode is via an automatically generated test bench script. Advanced users can make use of the HWCosim API exposed via the MATLAB Hwcosim objects that are shipped with System Generator for DSP.

Automatic Testbench Generation

Testbench generation is run alongside the hardware co-simulation compilation flow. Open the System Generator token in the Simulink model and wait for the dialog box to appear. The first tab shows the **Compilation** options. A drop-down list shows the available compilation targets. After selecting one of the two hardware co-simulation flows (depending on which one is available for the selected board), the **Settings** button will be enabled and when selected it will open a secondary dialog box where burst mode and the desired FIFO depth can be chosen. After burst mode has been turned on, you can enable the automatic creation of an M-HWCosim test bench script by enabling **Create testbench** at the bottom of the **Compilation** tab.

Figure 139: **Create Test Bench**



The test generator will produce this M-script file in the Target Directory:

```
<design_name>_<sub_system>_hwcosim_test.m
```

You can run this script from the MATLAB® console. The script will also run the Simulink model to determine the stimulus data driven to the Xilinx® Gateway In blocks (from the other Simulink source blocks or MATLAB variables), while also capturing the expected output produced by the Xilinx® Block Design (BD) and exporting the data to the **Target directory** as these separate data files:

```
<design_name>_<sub_system>_<port_name>.dat.
```

To run the test bench, you can open the MATLAB console, change directory to the Target Directory, and run the script by name. If the test fails this will be printed on the console, and the failing comparisons will be listed in this file:

```
<design_name>_<sub_system>_hwcosim_test.result.
```

Burst Mode Testbench Script

The following is a test bench generated for an example design as part of the compilation flow:

TIP: *If there are multiple Ethernet adapters connected to your board, you can use M-Hwcosim to select the desired Ethernet interface for Point-to-Point Ethernet hardware co-simulation. The procedure for selecting the interface is described in Selecting the Adapter for Point-to-Point Ethernet Hardware Co-Simulation with M-Hwcosim in the Vivado Design Suite Reference Guide: Model-Based DSP Design Using System Generator (UG958).*

```
%% project3_burst_hwcosim_test
% project3_burst_hwcosim_test is an automatically generated example MCode
% function that can be used to open a hardware co-simulation (hwcosim)
target,
% load the bitstream, write data to the hwcosim target's input blocks, fetch
% the returned data, and verify that the test passed. The returned value of
% the test is the amount of time required to run the test in seconds.
% Fail / Pass is indicated as an error or displayed in the command window.

%%
% PLEASE NOTE that this file is automatically generated and gets re-created
% every time the Hardware Co-Simulation flow is run. If you modify any part
% of this script, please make sure you save it under a new name or in a
% different location.

%%
% The following sections exist in the example test function:
% Initialize Bursts
% Initialize Input Data & Golden Vectors
% Open and Simulate Target
% Release Target on Error
% Test Pass / Fail

function eta = project3_burst_hwcosim_test
eta = 0;

%%
% ncycles is the number of cycles to simulate for and should be adjusted if
% the generated testbench simulation vectors are substituted by user data.
ncycles = 10;

%%
% Initialize Input Data & Golden Vectors
% xlHwcosimTestbench is a utility function that reformats fixed-point HDL
Netlist
% testbench data vectors into a double-precision floating-point MATLAB
binary
% data array.
xlHwcosimTestbench('.','project3_burst');

%%
% The testbench data vectors are both stimulus data for each input port, as
% well as expected (golden) data for each output port, recorded during the
% Simulink simulation portion of the Hardware Co-Simulation flow.
% Data gets loaded from the data file ('<name>_<port>_hwcosim_test.dat')
% into the corresponding 'testdata_<port>' workspace variables using
% 'getfield(load('<name>_<port>_hwcosim_test.dat' ... ' commands.
%
% Alternatively, the workspace variables holding the stimulus and / or
golden
% data can be assigned other data (including dynamically generated data) to
% test the design with. If using alternative data assignment, please make
% sure to adjust the "ncycles" variable to the proper number of cycles, as
% well as to disable the "Test Pass / Fail" section if unused.
testdata_noise_x0 =
```

```
getfield(load('project3_burst_noise_x0_hwcosim_test.dat', '-mat'),
'values');
testdata_scale = getfield(load('project3_burst_scale_hwcosim_test.dat', '-
mat'), 'values');
testdata_wave = getfield(load('project3_burst_wave_hwcosim_test.dat', '-
mat'), 'values');
testdata_intout = getfield(load('project3_burst_intout_hwcosim_test.dat', '-
mat'), 'values');
testdata_sigout = getfield(load('project3_burst_sigout_hwcosim_test.dat', '-
mat'), 'values');

%%
% The 'result_<port>' workspace variables are arrays to receive the actual
results
% of a Hardware Co-Simulation read from the FPGA. They will be compared to
the
% expected (golden) data at the end of the Co-Simulation.
result_intout = zeros(size(testdata_intout));
result_sigout = zeros(size(testdata_sigout));

%%
% project3_burst.hwc is the data structure containing the Hardware Co-
Simulation
% design information returned after netlisting the Simulink / System
% Generator model.
% Hwcosim(project) instantiates and returns a handle to the API shared
library object.
project = 'project3_burst.hwc';
h = Hwcosim(project);
try
    %% Open the Hardware Co-Simulation target and co-simulate the design
    open(h);
    cosim_t_start = tic;
    h('noise_x0') = testdata_noise_x0;
    h('scale') = testdata_scale;
    h('wave') = testdata_wave;
    run(h, ncycles);
    result_intout = h('intout');
    result_sigout = h('sigout');
    eta = toc(cosim_t_start);
    % Release the handle for the Hardware Co-Simulation target
    release(h);

%% Release Target on Error
catch err
    release(h);
    rethrow(err);
    error('Error running hardware co-simulation testbench. Please refer to
hwcosim.log for
details.');
end

%% Test Pass / Fail
logfile = 'project3_burst_hwcosim_test.results';
logfd = fopen(logfile, 'w');
sim_ok = true;
sim_ok = sim_ok & xlHwcosimCheckResult(logfd, 'intout', testdata_intout,
result_intout);
sim_ok = sim_ok & xlHwcosimCheckResult(logfd, 'sigout', testdata_sigout,
result_sigout);
fclose(logfd);
if ~sim_ok
    error('Found errors in the simulation results. Please refer to
```

```
project3_burst_hwcosim_test.results for details.');
end
disp(['Hardware Co-Simulation successful. Data matches the Simulink
simulation and completed in
' num2str(eta) ' seconds.']) ;
```

This script first defines the number of cycles (`ncycles`) to run in the simulation, prepares the test bench, and loads the stimulus data and expected output into MATLAB arrays. Then it creates an Hwcosim object instance with a handle (`h`), which loads the HWCosim API shared library. Inside the try-catch block it opens the instance, initializes the FPGA, and opens a connection to it.

Once the setup phase is complete, the code between the `tic` and `toc` timing commands executes the write-run-read commands. Please note that unlike in previous versions of HWCosim, this test bench does not require a for-loop to cycle through every clock cycle. This is due to the new smart cache layer which can buffer up nearly arbitrary size write commands in host memory before issuing smaller cycles of write-run-read batches to the hardware (during execution of the user-visible `run(h, ncycles)` command).

At the end of the execution phase the HWCosim instance is released and the test bench compares actual to expected outputs.

Comments in the test bench code will help you understand the flow of the hardware co-simulation and help you develop customized test bench scripts for your design.

XILINX

Chapter 6

Importing HDL Modules

Sometimes it is important to add one or more existing HDL modules to a System Generator design. The System Generator Black Box block allows VHDL, and Verilog, to be brought into a design. The Black Box block behaves like other System Generator blocks - it is wired into the design, participates in simulations, and is compiled into hardware. When System Generator compiles a Black Box block, it automatically connects the ports of the Black Box to the rest of the design. A Black Box can be configured to support either synchronous clock designs or multiple hardware clock designs based on the context and System Generator token settings.

The Black Box Interface	
Black Box HDL Requirements and Restrictions	Details the requirements and restrictions for VHDL, Verilog, and EDIF associated with black boxes.
Chapter 7: Black Box Configuration Wizard	Describes how to use the Black Box Configuration Wizard.
Black Box Configuration M-Function	Describes how to create a black box configuration M-function.

HDL Co-Simulation	
Configuring the HDL Simulator	Explains how to configure the Vivado® simulator or ModelSim to co-simulate the HDL in the Black Box block.
Co-Simulating Multiple Black Boxes	Describes how to co-simulate several Black Box blocks in a single HDL simulator session.

Black Box HDL Requirements and Restrictions

An HDL component associated with a black box must adhere to the following System Generator requirements and restrictions:

- The entity name must not collide with any other entity name in the design.
- Bi-directional ports are supported in HDL black boxes, however they will not be displayed in the System Generator as ports; they only appear in the generated HDL after netlisting.
- For Verilog black boxes, the module and port names must follow standard HDL naming conventions.
- Any port that is a clock or clock enable must be of type std_logic. (For Verilog black boxes, ports must be of non-vector inputs, e.g., input clk.)

- Clock and clock enable ports in black box HDL should be expressed as follows: Clock and clock enables must appear as pairs (i.e., for every clock, there is a corresponding clock enable, and vice-versa). A black box may have more than one clock port and its behavior changes based on the context of the design.
    - In Synchronous single clock design context, a single clock source is used to drive each clock port. Only the clock enable rates differ.
    - In case of multiple independent hardware clock design context, two different clock sources is used to drive clock and clock enable pins.
- Each clock name (respectively, clock enable name) must contain the substring `clk`, for example `my_clk_1` and `my_ce_1`.
- The name of a clock enable must be the same as that for the corresponding clock, but with `ce` substituted for `clk`. For example, if the clock is named `src_clk_1`, then the clock enable must be named `src_ce_1`.
- Falling-edge triggered output data cannot be used.

IMPORTANT! *It is not recommended to use the black box block to import encrypted RTLs which are generated for Vivado IP. As an alternative, try to import Vivado IPs using a DCP file.*

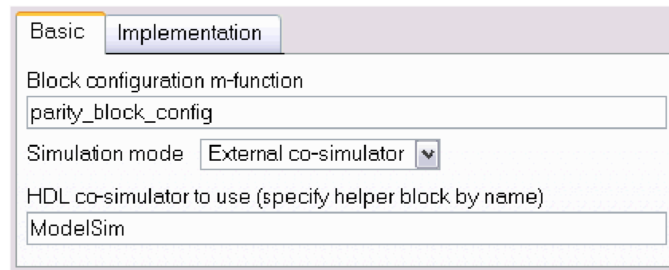
Black Box Configuration M-Function

An imported module is represented in System Generator by a Black Box block. Information about the imported module is conveyed to the black box by a configuration M-function. This function defines the interface, implementation, and the simulation behavior of the black box block it is associated with. The information a configuration M-function defines includes the following:

- Name of the top-level entity for the module
- VHDL or Verilog language selection
- Port descriptions
- Generics required by the module
- Synchronous single clock or asynchronous multiple independent clock configuration
- Clocking and sample rates
- Files associated with the module
- Whether the module has any combinational paths

The name of the configuration M-function associated with a black box is specified as a parameter in the dialog box (`parity_block_config.m`).

Figure 140: **Parameter Dialog**



Configuration M-functions use an object-based interface to specify black box information. This interface defines two objects, SysgenBlockDescriptor and SysgenPortDescriptor. When System Generator invokes a configuration M-function, it passes the function a block descriptor:

```
function sample_block_config(this_block)
```

A SysgenBlockDescriptor object provides methods for specifying information about the black box. Ports on a block descriptor are defined separately using port descriptors.

Language Selection

The black box can import VHDL and Verilog modules. SysgenBlockDescriptor provides a method, setTopLevelLanguage, that tells the black box what type of module you are importing. This method should be invoked once in the configuration M-function. The following code shows how to select between the VHDL and Verilog languages.

VHDL Module:

```
this_block.setTopLevelLanguage('VHDL');
```

Verilog Module:

```
this_block.setTopLevelLanguage('Verilog');
```

***Note*:** The Configuration Wizard automatically selects the appropriate language when it generates a configuration M-function.

Specifying the Top-Level Entity

You must tell the black box the name of the top-level entity that is associated with it. SysgenBlockDescriptor provides a method, setEntityName, which allows you to specify the name of the top-level entity.

***Note*:** Use lower case text to specify the entity name.

For example, the following code specifies a top-level entity named `foo`.

```
this_block.setEntityName('foo');
```

Note: The Configuration Wizard automatically sets the name of the top-level entity when it generates a configuration M-function.

Defining Port Blocks

The port interface of a black box is defined by the block's configuration M-function. Recall that black box ports are defined using port descriptors. A port descriptor provides methods for configuring various port attributes, including port width, data type, binary point, and sample rate.

Adding New Ports

When defining a black box port interface, it is necessary to add input and output ports to the block descriptor. These ports correspond to the ports on the module you are importing. In your model, the black box block port interface is determined by the port names that are declared on the block descriptor object. SysgenBlockDescriptor provides methods for adding input and output ports:

Adding an input port:

```
this_block.addSimulinkInport('din');
```

Adding an output port:

```
this_block.addSimulinkOutport('dout');
```

The string parameter passed to methods addSimulinkInport and addSimulinkOutport specifies the port name. These names should match the corresponding port names in the imported module.

Note: Use lower case text to specify port names.

Adding a bidirectional port:

```
config_phase = this_block.getConfigPhaseString;
if (strcmpi(config_phase,'config_netlist_interface'))
 this_block.addInoutport('bidi');
 % Rate and type info should be added here as well
end
```

Bidirectional ports are supported only during the netlisting of a design and will not appear on the System Generator diagram; they only appear in the generated HDL. As such, it is important to only add the bi-directional ports when System Generator is generating the HDL. The if-end conditional statement is guarding the execution of the code to add-in the bi-directional port.

It is also possible to define both the input and output ports using a single method call. The setSimulinkPorts method accepts two parameters. The first parameter is a cell array of strings that define the input port names for the block. The second parameter is a cell array of strings that define the output port names for the block.

Note: The Configuration Wizard automatically sets the port names when it generates a configuration M-function.

Obtaining a Port Object

Once a port has been added to a block descriptor, it is often necessary to configure individual attributes on the port. Before configuring the port, you must obtain a descriptor for the port you would like to configure. SysgenBlockDescriptor provides methods for accessing the port objects that are associated with it. For example, the following method retrieves the port named `din` on the `this_block` descriptor:

Accessing a SysgenPortDescriptor object:

```
din = this_block.port('din');
```

In the above code, an object `din` is created and assigned to the descriptor returned by the `port` function call.

SysgenBlockDescriptor also provides methods, `inport` and `outport`, that return a port object given a port index. A port index is the index of the port (in the order shown on the block interface) and is some value between 1 and the number of input/output ports on the block. These methods are useful when you need to iterate through the block's ports (e.g., for error checking).

Configuring Port Types

SysgenPortDescriptor provides methods for configuring individual ports. For example, assume port dout is unsigned, 12 bits, with binary point at position 8. The code below shows one way in which this type can be defined.

```
dout = this_block.port('dout');
dout.setWidth(12);
dout.setBinPt(8);
dout.makeUnsigned();
```

The following also works:

```
dout = this_block.port('dout');
dout.setType('Ufix_12_8');
```

The first code segment sets the port attributes using individual method calls. The second code segment defines the signal type by specifying the signal type as a string. Both code segments are functionally equivalent.

The black box supports HDL modules with 1-bit ports that are declared using either single bit port (e.g., std_logic) or vectors (e.g., std_logic_vector(0 downto 0)) notation. By default, System Generator assumes ports to be declared as vectors. You may change the default behavior using the useHDLVector method of the descriptor. Setting this method to `true` tells System Generator to interpret the port as a vector. A `false` value tells System Generator to interpret the port as single bit.

```
dout.useHDLVector(true); % std_logic_vector
dout.useHDLVector(false); % std_logic
```

Note: The Configuration Wizard automatically sets the port types when it generates a configuration M-function.

Configuring Bi-Directional Ports for Simulation

Bidirectional ports (or inout ports) are supported only during the generation of the HDL netlist, that is, bi-directional ports will not show up in the System Generator diagram. By default, bi-directional ports will be driven with 'X' during simulation. It is possible to overwrite this behavior by associating a data file to the port. Be sure to guard this code because bi-directional ports can only be added to a block during the `config_netlist_interface` phase.

```
if (strcmpi(this_block.getConfigPhaseString,'config_netlist_interface'))
  bidi_port = this_block.port('bidi');
  bidi_port.setGatewayFileName('bidi.dat');
end
```

In the above example, a text file, `bidi.dat`, is used during simulation to provide stimulation to the port. The data file should be a text file, where each line represents the signal driven on the port at each simulation cycle. For example, a 3-bit bi-directional port that is simulated for 4 cycles might have the following data file:

```
ZZZ
110
011
XXX
```

Simulation will return with an error if the specified data file cannot be found.

Configuring Port Sample Rates

The Black Box block supports ports that have different sample rates. By default, the sample rate of an output port is the sample rate inherited from the input port (or ports, if the inputs run at the same sample rate). Sometimes, it is necessary to explicitly specify the sample rate of a port (e.g., if the output port rate is different than the block's input sample rate).

Note: When the inputs to a black box have different sample rates, you must specify the sample rates of every output port.

SysgenPortDescriptor provides a method called setRate that allows you to explicitly set the rate of a port.

Note: The rate parameter passed to the setRate method is not necessarily the Simulink® sample rate that the port runs at. Instead, it is a positive integer value that defines the ratio between the desired port sample period and the Simulink® system clock period defined by the System Generator token dialog box.

Assume you have a model in which the Simulink system period value for the model is defined as 2 sec. Also assume that the example `dout` port is assigned a rate of 3 by invoking the setRate method as follows:

```
dout.setRate(3);
```

A rate of 3 means that a new sample is generated on the dout port every 3 Simulink system periods. Because the Simulink system period is 2 sec, this means the Simulink sample rate of the port is 3 x 2 = 6 sec.

Note: If your port is a non-sampled constant, you can define it in the configuration M-function using the setConstant method of SysgenPortDescriptor. You can also define a constant by passing Inf to the setRate method.

Dynamic Output Ports

A useful feature of the black box is its ability to support dynamic output port types and rates. For example, it is often necessary to set an output port width based on the width of an input port. SysgenPortDescriptor provides member variables that allow you to determine the configuration of a port. You can set the type or rate of an output port by examining these member variables on the block's input ports.

For example, you can obtain the width and rate of a port (in this case `din`) as follows:

```
input_width = this_block.port('din').width;
input_rate  = this_block.port('din').rate;
```

Note: A black box's configuration M-function is invoked at several different times when a model is compiled. The configuration function may be invoked before the data types and rates have been propagated to the black box.

The SysgenBlockDescriptor object provides Boolean member variables `inputTypesKnown` and `inputRatesKnown` that tell whether the port types and rates have been propagated to the block. If you are setting dynamic output port types or rates based on input port configurations, the configuration calls should be nested inside conditional statements that check that values of `inputTypesKnown` and `inputRatesKnown`.

The following code shows how to set the width of a dynamic output port `dout` to have the same width as input port `din`:

```
if (this_block.inputTypesKnown)
  dout.setWidth(this_block.port('din').width);
end
```

Setting dynamic rates works in a similar manner. The code below sets the sample rate of output port `dout` to be twice as slow as the sample rate of input port `din`:

```
if (this_block.inputRatesKnown)
  dout.setRate(this_block.port('din').rate*2);
end
```

Black Box Clocking

In order to import a multirate module, you must tell System Generator information about the module's clocking in the configuration M-function. System Generator treats clock and clock enables differently than other types of ports. A clock port on an imported module must always be accompanied by a clock enable port (and vice versa). In other words, clock and clock enables must be defined as a pair, and exist as a pair in the imported module. This is true for both single rate and multirate designs.

Although clock and clock enables must exist as pairs, System Generator drives all clock ports on your imported module with the FPGA system clock. The clock enable ports are driven by clock enable signals derived from the FPGA system clock.

SysgenBlockDescriptor provides a method, `addClkCEPair`, which allows you to define clock and clock enable information for a black box. This method accepts three parameters. The first parameter defines the name of the clock port (as it appears in the module). The second parameter defines the name of the clock enable port (also as it appears in the module).

The port names of a clock and clock enable pair must follow the naming conventions provided below:

- The clock port must contain the substring `clk`
- The clock enable must contain the substring `ce`
- The strings containing the substrings `clk` and `ce` must be the same (e.g. `my_clk_1` and `my_ce_1`).

The third parameter defines the rate relationship between the clock and the clock enable port. The rate parameter should not be thought of as a Simulink sample rate. Instead, this parameter tells System Generator the relationship between the clock sample period, and the desired clock enable sample period. The rate parameter is an integer value that defines the ratio between the clock rate and the corresponding clock enable rate.

For example, assume you have a clock enable port named `ce_3` that would like to have a period three times larger than the system clock period. The following function call establishes this clock enable port:

```
addClkCEPair('clk_3','ce_3',3);
```

When System Generator compiles a black box into hardware, it produces the appropriate clock enable signals for your module, and automatically wires them up to the appropriate clock enable ports.

Combinational Paths

If the module you are importing has at least one combinational path (i.e. a change on any input can effect an output port without a clock event), you must indicate this in the configuration M-function. SysgenBlockDescriptor object provides a `tagAsCombinational` method that indicates your module has a combinational path. It should be invoked as follows in the configuration M-function:

```
this_block.tagAsCombinational;
```

Specifying VHDL Generics and Verilog Parameters

You may specify a list of generics that get passed to the module when System Generator compiles the model into HDL. Values assigned to these generics can be extracted from mask parameters and from propagated port information (e.g. port width, type, and rate). This flexible means of generic assignment allows you to support highly parametric modules that are customized based on the Simulink environment surrounding the black box.

The `addGeneric` method allows you to define the generics that should be passed to your module when the design is compiled into hardware. The following code shows how to set a VHDL Integer generic, `dout_width`, to a value of 12.

```
addGeneric('dout_width','Integer','12');
```

It is also possible to set generic values based on port on propagated input port information (e.g. a generic specifying the width of a dynamic output port).

Because a black box's configuration M-function is invoked at several different times when a model is compiled, the configuration function may be invoked before the data types (or rates) have been propagated to the black box. If you are setting generic values based on input port types or rates, the `addGeneric` calls should be nested inside a conditional statement that checks the value of the `inputTypesKnown` or `inputRatesKnown` variables. For example, the width of the dout port can be set based on the value of din as follows:

```
if (this_block.inputTypesKnown)
  % set generics that depend on input port types
  this_block.addGeneric('dout_width', ...
  this_block.port('din').width);
end
```

Generic values can be configured based on mask parameters associated with a block box. SysgenBlockDescriptor provides a member variable, `blockName`, which is a string representation of the black box's name in Simulink. You may use this variable to gain access the black box associated with the particular configuration M-function. For example, assume a black box defines a parameter named `init_value`. A generic with name `init_value` can be set as follows:

```
simulink_block = this_block.blockName;
init_value = get_param(simulink_block,'init_value');
this_block.addGeneric('init_value', 'String', init_value);
```

Note: You can add your own parameters (e.g. values that specify generic values) to the black box by doing the following:

- Copy a black box into a Simulink library or model.
- Break the link on the black box.
- Add the desired parameters to the black box dialog box.

Black Box VHDL Library Support

This Black Box feature allows you to import VHDL modules that have predefined library dependencies. The following example illustrates how to do this import.

The VHDL module below is a 4-bit, Up counter with asynchronous clear (`async_counter.vhd`). It will be compiled into a library named `async_counter_lib`.

```
1   -- 4-bit, Up counter, with asynchronous clear
2      library ieee;
3      use ieee.std_logic_1164.all;
4      use ieee.std_logic_unsigned.all;
5      entity async_counter is
6      port(clk, clr : in  std_logic;
7          ce: in std_logic := '1';
8          q : out std_logic_vector(3 downto 0));
9      end async_counter;
10     architecture archi of async_counter is
11        signal tmp: std_logic_vector(3 downto 0);
12     begin
13     process (clk, clr)
14           begin
15           if (clr='1') then
16               tmp <= "0000";
17           elsif (clk'event and clk='1') then
18               tmp <= tmp + 1;
19           end if;
20     end process;
21     q <= tmp;
22     end archi;
```

The VHDL module below is a 4-bit, Up counter with synchronous clear (`sync_counter.vhd`). It will be compiled into a library named `sync_counter_lib`.

```
1   -- 4-bit, Up counter, with synchronous clear
2      library ieee;
3      use ieee.std_logic_1164.all;
4      use ieee.std_logic_unsigned.all;
5
6      entity sync_counter is
7      port(clk, clr : in  std_logic;
8          ce: in std_logic := '1';
9          q : out std_logic_vector(3 downto 0));
10     end sync_counter;
11     architecture archi of sync_counter is
12        signal tmp: std_logic_vector(3 downto 0);
13     begin
14     process (clk)
15     begin
16      if (clk'event and clk='1') then
17        if (clr='1') then
18            tmp <= "0000";
19        else
20            tmp <= tmp + 1;
21        end if;
22      end if;
23     end process;
24     q <= tmp;
25     end archi;
```

The VHDL module below is the top-level module that is used to instantiate the previous modules. This is the module that you need to point to when adding the BlackBox into you System Generator model.

```vhdl
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4  library sync_counter_lib;
5  use sync_counter_lib.all;
6  library async_counter_lib;
7  use async_counter_lib.all;
8
9
10 entity top_level is
11 port(clk, clr : in  std_logic;
12     ce: in std_logic := '1';
13     q_sync : out std_logic_vector(3 downto 0);
14     q_async : out std_logic_vector(3 downto 0)
15     );
16 end top_level;
17
18 architecture structural of top_level is
19 component async_counter
20 port (
21       clk, clr, ce: in std_logic;
22       q: out std_logic_vector(3 downto 0));
23 end component;
24
25 component sync_counter
26 port (
27       clk, clr, ce: in std_logic;
28       q: out std_logic_vector(3 downto 0));
29 end component;
30
31 begin
32 counter_0: entity async_counter_lib.async_counter
33 port map (
34    ce => ce,
35    q  => q_async,
36    clk => clk,
37    clr => clr
38  );
39 counter_1: entity sync_counter_lib.sync_counter
40 port map (
41    ce => ce,
42    q  => q_sync,
43    clk => clk,
44    clr => clr
45  );
46 end structural;
```

Define libraries using "library" and "use" clauses

The VHDL is imported by first importing the top-level entity, `top_level`, using the Black Box.

Once the file is imported, the associated Black Box Configuration M-file needs to be modified as follows:

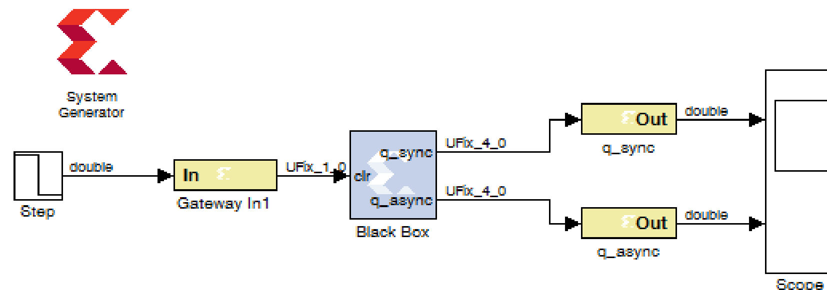
```
% Add addtional source files as needed.
%  |-------------
%  | Add files in the order in which they should be compiled.
%  | If two files "a.vhd" and "b.vhd" contain the entities
%  | entity_a and entity_b, and entity_a contains a
%  | component of type entity_b, the correct sequence of
%  | addFile() calls would be:
%  |    this_block.addFile('b.vhd');
%  |    this_block.addFile('a.vhd');
%  |-------------

%    this_block.addFile('');
%    this_block.addFile('');
this_block.addFile('top.vhd');
this_block.addFileToLibrary('async_counter.vhd','async_counter_lib');
this_block.addFileToLibrary('sync_counter.vhd','sync_counter_lib');
```

Specifying library names by using "addFileToLibrary" command

The interface function `addFileToLibrary` is used to specify a library name other than "work" and to instruct the tool to compile the associated HDL source to the specified library.

The System Generator model should look similar to the figure below.

Figure 141: **System Generator Model with Black Box Support**



The next step is to double-click the System Generator token and click the **Generate** button to generate the HDL netlist.

During the generation process, a Vivado IDE project (`.xpr`) is created and placed with the `hdl_netlist` folder under the `netlist` folder. If you double-click the Vivado IDE project and select the Libraries tab under the Source view, you will see not only a `work` library, but an `async_counter_lib` library and `sync_counter_lib` library as well.

Error Checking

It is often necessary to perform error checking on the port types, rates, and mask parameters of a black box. SysgenBlockDescriptor provides a method, setError, that allows you to specify an error message that is reported to the user. The error message that a user sees is the string parameter passed to setError.

Black Box API

SysgenBlockDescriptor Member Variables

Type	Member	Description
String	entityName	Name of the entity or module.
String	blockName	Name of the black box block.
Integer	numSimulinkInports	Number of input ports on black box.
Integer	numSimulinkOutports	Number of output ports on the black box.
Boolean	inputTypesKnown	true if all input types are defined, and false otherwise.
Boolean	inputRatesKnown	true if all input rates are defined, and false otherwise.
Array of Doubles	inputRates	Array of sample periods for the input ports (indexed as in inport(indx)). Sample period values are expressed as integer multiples of the Simulink System Period value specified by the master System Generator token
Boolean	error	true if an error has been detected, and false otherwise.
Cell Array of Strings	errorMessages	Array of all error messages for this block.

SysgenBlockDescriptor Methods

Method	Description
setTopLevelLanguage(language)	Declares language for the top-level entity (or module) of the black box. The language should be `VHDL` or `Verilog`.
setEntityName(name)	Sets name of the entity or module.
addSimulinkInport(pname)	Adds an input port to the black box. pname defines the name the port should have.
addSimulinkOutport(pname)	Adds an output port to the black box. pname defines the name the port should have.
setSimulinkPorts(in,out)	Adds input and output ports to the black box. in (respectively, out) is a cell array whose element tell the names to use for the input (resp., output) ports.
addInoutport(pname)	Adds a bidirectional port to the black box. pname defines the name the port should have. Bidirectional ports can only be added during the config_netlist_interface phase of configuration.
tagAsCombinational()	Indicate that the block has a combinational path (i.e., direct feedthrough) from an input port to an output port.

Method	Description
addClkCEPair(clkPname, cePname, rate)	Defines a clock/clock enable port pair for the block. clkPname and cePname tell the names for the clock and clock enable ports respectively. rate, a double, tells the rate at which the port pair runs. The rate must be a positive integer. Note the clock (respectively, clock enable) name must contain the substring clk (resp., ce). The names must be parallel in the sense that the clock enable name is obtained from the clock name by replacing clk with ce.
port(name)	Returns the SysgenPortDescriptor that matches the specified name.
inport(indx)	Returns the SysgenPortDescriptor that describes a given input port. indx tells the index of the port to look for, and should be between 1 and numInputPorts.
outport(indx)	Returns the SysgenPortDescriptor that describes a given output port. indx tells the index of the port to look for, and should be between 1 and numOutputPorts.
addGeneric(identifier, value)	Defines a generic (or parameter if using Verilog) for the block. identifier is a string that tells the name of the generic. value can be a double or a string. The type of the generic is inferred from value's type. If value is an integral double (e.g., 4.0), the type of the generic is set to integer. For a non-integral double, the type is set to real. When value is a string containing only zeros and ones (e.g., `0101'), the type is set to bit_vector. For any other string value, the type is set to string.
addGeneric(identifier, type, value)	Explicitly specifies the name, type, and value for a generic (or parameter, if using Verilog) for the block. All three arguments are strings. identifier tells the name, type tells the type, and value tells the value.
addFile(fn)	Adds a file name to the list of files associated to this black box, `fn` is the file name. Ordinarily, HDL files are associated to black boxes, but any sorts of files are acceptable. VHDL file names should end in `.vhd`; Verilog file names should end in `.v`. The order in which file names are added is preserved, and becomes the order in which HDL files are compiled. File names can be absolute or relative. Relative file names are interpreted with respect to the location of the .mdl or library .mdl for the design.
getDeviceFamilyName()	Gets the name of the FPGA corresponding to the black box.
getConfigPhaseString	Returns the current configuration phase as a string. A valid return string includes: config_interface, config_rate_and_type, config_post_rate_and_type, config_simulation, config_netlist_interface, and config_netlist.
setSimulatorCompilationScript(script)	Overrides the default HDL co-simulation compilation script that the black box generates. `script` tells the name of the script to use. For example, this method can be used to short-circuit the compilation phase for repeated simulations where the HDL for the black box remains unchanged.
setError(message)	Indicates that an error has occurred, and records the error message. `message` gives the error message.

SysgenPortDescriptor Member Variables

Type	Member	Description
String	name	Tells the name of the port.

Type	Member	Description
Integer	simulinkPortNumber	Tells the index of this port in Simulink®. Indexing starts with 1 (as in Simulink).
Boolean	typeKnown	True if this port's type is known, and false otherwise.
String	type	Type of the port, such as UFix_<n>_<b>, Fix_<n>_<b>, or Bool.
Boolean	isBool	True if port type is Bool, and false otherwise.
Boolean	isSigned	True if type is signed, and false otherwise.
Boolean	isConstant	True if port is constant, and false otherwise.
Integer	width	Tells the port width.
Integer	binpt	Tells the binary point position, which must be an integer in the range 0..width.
Boolean	rateKnown	True if the rate is known, and false otherwise.
Double	rate	Tells the port sample time. Rates are positive integers expressed as MATLAB® doubles. A rate can also be infinity, indicating that the port outputs a constant.

SysgenPortDescriptor Methods

Method	Description
setName(name)	Sets the HDL name to be used for this port.
setSimulinkPortNumber(num)	Sets the index associated with this port in Simulink®. num tells the index to assign. Indexing starts with 1 (as in Simulink).
setType(typeName)	Sets the type of this port to type. Type must be one of Bool, UFix_<n>_<b> , Fix_<n>_<b> , signed or unsigned. The last two choices leave the width and binary point position unchanged. XFloat_<exponent_bit_width>_fraction_bit_width> is also supported. For example: ap_return_port = this_block.port('ap_return'); ap_return_port.setType('XFloat_30_2');
setWidth(w)	Sets the width of this port to w.
setBinpt(bp)	Sets the binary point position of this port to bp.
makeBool()	Makes this port Boolean.
makeSigned()	Makes this port signed.
makeUnsigned()	Makes this port unsigned.
setConstant()	Makes this port constant

Method	Description
setGatewayFileName(filename)	Sets the dat file name that will be used in simulations and test-bench generation for this port. This function is only meant for use with bi-directional ports so that a hand written data file can be used during simulation. Setting this parameter for input or output ports is invalid and will be ignored.
setRate(rate)	Assigns the rate for this port. rate must be a positive integer expressed as a MATLAB® double or Inf for constants.
useHDLVector(s)	Tells whether a 1-bit port is represented as single-bit (ex: std_logic) or vector (ex: std_logic_vector(0 downto 0)).
HDLTypeIsVector()	Sets representation of the 1-bit port to std_logic_vector(0 downto 0).

Multiple Independent Clock Support on Black Box

Design Rule Checks on Port Connection

When a black box is used in a multiple independent hardware clock design context, design rule checks (DRCs) for its port connections must be added in the configuration M-function. This helps to avoid invalid or incorrect port connection with different clock sources. You need to ensure all port signals are connected from/to a proper clocked-subsystem interface.

The utility `checkPortsOfSameClockDomain()` should be used to specify a list of ports from a particular clock domain and to group it together. The input arguments to this application programming interface (API) are 'SysgenBlockDescriptor' objects followed by the list of port names associated with a particular clock domain.

In the example shown below, the API puts out an error check, and verifies that the four ports are connected to the same subsystem clock domain.

```
checkPortsOfSameClockDomain (<block_descriptor>, '<port_name_1>',
'<port_name_2>',
'<port_name_3>', '<port_name_4>');
```

Configuring Port Sample Rates

In multiple clock hardware designs, the clock period of the port interface should be computed using the connected "clocked subsystem domain". By default, "synchronous system clock" source is used by all the ports, but for asynchronous clock hardware designs, it is necessary to explicitly specify the clock sources of every port (e.g., if the output port clock is different than the block's input port clock).

Note: You must set the sample rate to '1.0' for all output ports of multiple independent clock black box designs; it automatically sets the output ports to the destination clock subsystem period.

SysgenPortDescriptor provides a method called `setRate` that you can use to explicitly set the rate of a port.

Example:

```
port('<port_name>').setRate(1.0)
```

Black Box Clocking

In order to import a synchronous or asynchronous black box module, you must tell System Generator information about the module's clocking in the configuration M-function. System Generator treats clock and clock enables differently than other types of ports. A clock port on an imported module must always be accompanied by a clock enable port, and vice versa. In other words, clock and clock enables must be defined as a pair, and exist as a pair in the imported module. This is true for both single synchronous clock and multiple independent clock designs.

SysgenBlockDescriptor provides a method called addClkCEPair that you can use to define clock, clock enable, and its associated clock period by using clock sub-system domain. The clock domain information is not required for synchronous single clock designs.

The first parameter defines the name of the clock port (as it appears in the module). The second parameter defines the name of the clock enable port (also as it appears in the module).

The port names of a clock and clock enable pair must follow the naming conventions provided below:

- The clock port must contain the substring `clk`.
- The clock enable must contain the substring `ce`.
- The strings containing the substrings `clk` and `ce` must be the same, such as: `my_clk_1` and `my_ce_1`.

The third parameter defines the rate relationship between the clock and the clock-enable port. The rate parameter should not be thought of as a Simulink® sample rate. Instead, this parameter tells System Generator the relationship between the clock sample period, and the desired clock enable sample period. The rate parameter is an integer value that defines the ratio between the clock rate and the corresponding clock enable rate.

For multiple independent clock designs, the fourth and fifth optional parameters are mandatory.

The fourth parameter holds a "Boolean" value, and it defines whether clock and clock enable pair is tied to ground. If you set it to `true`, both clock and clock enable would be tied to ground during simulation. Setting it to false would activate clock and clock enable rate transitions.

The fifth parameter defines the clock period for the corresponding clock-clock enable pair. The 'clockDomain' property of the black box "SysgenPortDescriptor" must be used to set the clock periods for multiple independent clock designs.

Example:

```
rate_data = this_block.port('<port_name>').rate;
clkDomain_data = this_block.port(<port_name>).clockDomain;
this_block.addClkCEPair('clk','ce',rate_data, false, clkDomain_data);
```

HDL Co-Simulation

This topic describes how a mixed language/mixed flow design that includes Xilinx® blocks, HDL modules, and a Simulink block design can be simulated in its entirety.

System Generator simulates black boxes by automatically launching an HDL simulator, generating additional HDL as needed (analogous to an HDL test bench), compiling HDL, scheduling simulation events, and handling the exchange of data between the Simulink and the HDL simulator. This is called *HDL co-simulation*.

Configuring the HDL Simulator

Black box HDL can be co-simulated with Simulink® using the System Generator interface to either the Vivado® simulator or the ModelSim simulation software from Model Technology, Inc.

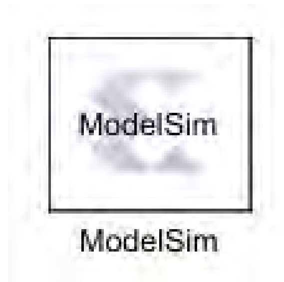
Xilinx® Simulator

To use the Xilinx simulator for co-simulating the HDL associated with the black box, select **Vivado Simulator** as the option for the **Simulation mode** parameter on the black box. The model is then ready to be simulated and the HDL co-simulation takes place automatically.

ModelSim Simulator

To use the ModelSim simulator by Model Technology, Inc., you must first add the ModelSim block that appears in the Tools library of the Xilinx Blockset to your Simulink diagram.

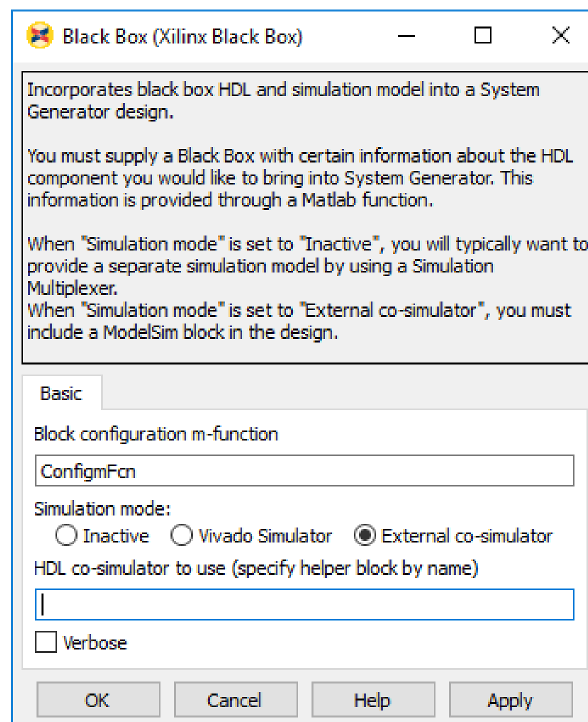
Figure 142: **ModelSim Block**



For each black box that you wish to have co-simulated using the ModelSim simulator, you need to open its block parameterization dialog and set it to use the ModelSim session represented by the black box that was just added. You do this by making the following two settings:

1. Change the Simulation Mode field from Inactive to External co-simulator.
2. Enter the name of the ModelSim block (e.g., ModelSim) in the HDL co-simulator to use field.

Figure 143: **Black Box Parameters**



The block parameter dialog for the ModelSim block includes some parameters that you can use to control various options for the ModelSim session. See the block help page for details. The model is then ready to be simulated with these options, and the HDL co-simulation takes place automatically.

Co-Simulating Multiple Black Boxes

System Generator allows many black boxes to share a common ModelSim co-simulation session. For example, many black boxes can be set to use the same ModelSim block. In this case, System Generator automatically combines all black box HDL components into a single shared top-level co-simulation component, which is transparent to the user. However, only one ModelSim simulation license is needed to co-simulate several black boxes in the Simulink® simulation.

Multiple black boxes can also be co-simulated with the Vivado simulator by selecting **Vivado Simulator** as the option for **Simulation mode** on each black box.

Black Box Configuration Wizard

System Generator provides a configuration wizard that makes it easy to associate a VHDL or Verilog module to a Black Box block. The Configuration Wizard parses the VHDL or Verilog module that you are trying to import, and automatically constructs a configuration M-function based on its findings. Then, it associates the configuration M-function it produces to the Black Box block in your model. Whether or not you can use the configuration M-function as is depends on the complexity of the HDL you are importing. Sometimes the configuration M-function must be customized by hand to specify details the configuration wizard misses. Details on the construction of the configuration M-function can be found in the Black Box Configuration M-Function topic.

Using the Configuration Wizard

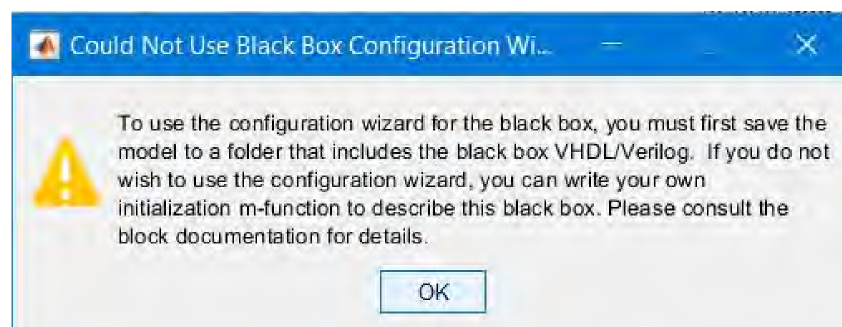
The Black Box Configuration Wizard opens automatically when a new black box block is added to a model.

***Note*:** Before running the Configuration Wizard, ensure the VHDL or Verilog you are importing meets the specified Black Box HDL Requirements and Restrictions.

For the Configuration Wizard to find your module, the model must be saved in the same directory as the module you are trying to import.

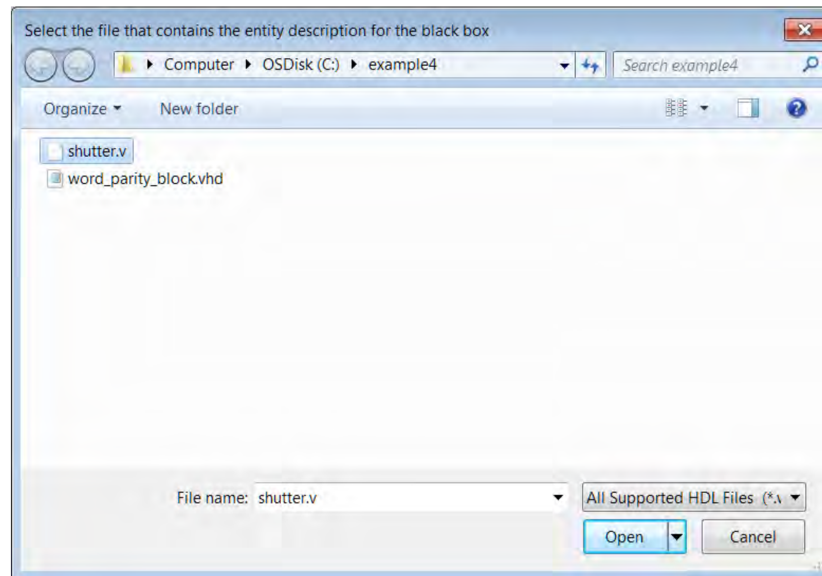
***Note*:** The wizard only searches for `.vhd` and `.v` files in the same directory as the model. If the wizard does not find any files it issues a warning and the black box is not automatically configured. The warning looks like the following:

Figure 144: **Warning**



After searching the model's directory for `.vhd` and `.v` files, the Configuration Wizard opens a new window that lists the possible files that can be imported. An example screenshot is shown below:

Figure 145: **Files to Import**



You can select the file you would like to import by selecting the file, and then pressing the **Open** button. At this point, the configuration wizard generates a configuration M-function, and associates it with the black box block.

Note: The configuration M-function is saved in the model's directory as `<module>_config.m`, where `<module>` is the name of the module that you are importing.

Configuration Wizard Fine Points

The configuration wizard automatically extracts certain information from the imported module when it is run, but some things must be specified by hand. These things are described below:

Note: The configuration function is annotated with comments that instruct you where to make these changes.

- If your model has a combinational path, you must call the tagAsCombinational method of the block's SysgenBlockDescriptor object. A multiple independent hardware clock design will not support a combinational path.
- The Configuration Wizard only knows about the top-level entity that is being imported. There are typically other files that go along with this entity. These files must be added manually in the configuration M-function by invoking the addFile method for each additional file.

- The Configuration Wizard automatically creates either a synchronous single clock black box descriptor or an asynchronous multiple clock black box descriptor.
    - In the case of single-rate black box, every port on the black box runs at the same rate. In most cases, this is acceptable. You may want to explicitly set port rates, which can result in a faster simulation time.
    - In the case of a multiple clock black box, the input port rate must be derived from the "source clock subsystem" and the output port rate must be set based on the "destination clock subsystem". In some cases, you may want to explicitly set port rates for a required configuration.

System Generator Compilation Types

There are different ways in which System Generator can compile your design into an equivalent, often lower-level, representation. The way in which a design is compiled depends on settings in the System Generator dialog box. The support of different compilation types provides you the freedom to choose a suitable representation for your design's environment. For example, an HDL Netlist or IP catalog is an appropriate target if your design is used as a component in a larger system.

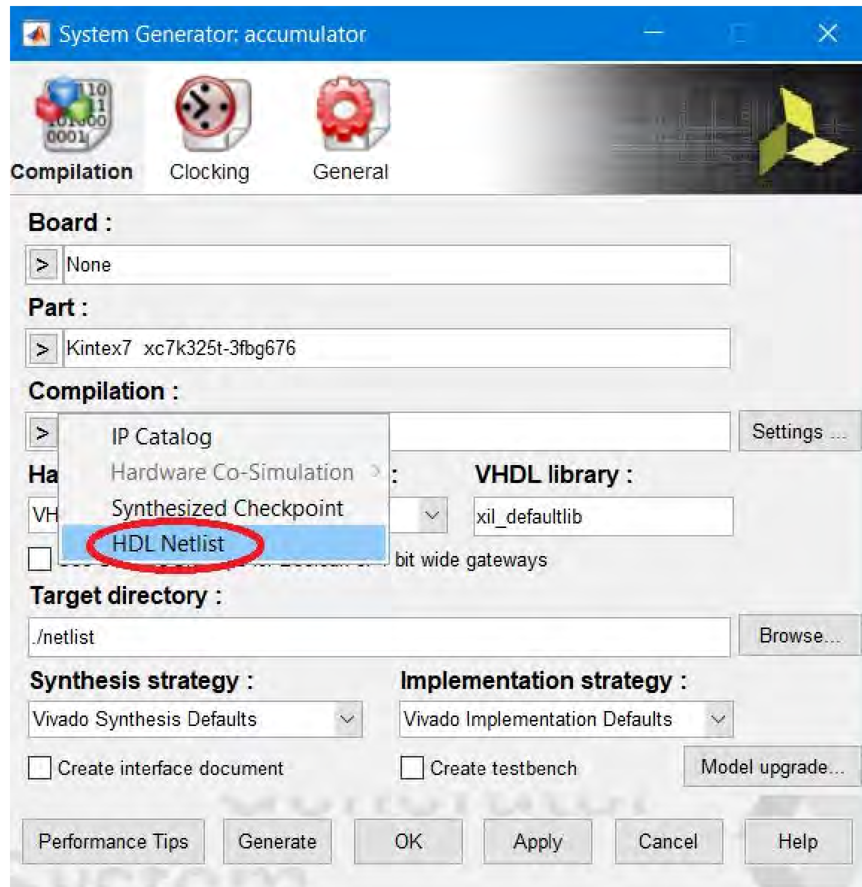
HDL Netlist Compilation	Describes how to generate HDL files that implement the design.
Hardware Co-Simulation Compilation	Describes how System Generator can be configured to compile your design into FPGA hardware that can be used by Simulink® and ModelSim.
IP Catalog Compilation	Describes how to package a System Generator design as an IP core that can be added to the Vivado® IP catalog for use in another design. System Generator uses the IP catalog compilation type as the default generation target.
Synthesized Checkpoint Compilation	Describes how to generate a synthesized checkpoint file (`synth_1.dcp`) that can be used in a Vivado integrated design environment (IDE) project.

HDL Netlist Compilation

The **HDL Netlist** compilation type produces HDL files that implement the design. More details regarding the HDL Netlist compilation flow can be found in the Compilation Results section.

As shown below, you may select **HDL Netlist** compilation by left-clicking the **Compilation** submenu control on the System Generator token dialog box, and selecting the **HDL Netlist** target.

Figure 146: **HDL Netlist**



The **Board** and **Part** fields allow you to specify the board or part for which you are targeting the **HDL Netlist** compilation. When you select a **Board**, the **Part** field automatically displays the name of the Xilinx® device on the selected **Board**, and this part name cannot be changed.

The HDL Netlist compilation can be performed for any of the boards or parts your Vivado tools support. In addition to accessing the Xilinx development boards installed as part of your Vivado installation, you can also specify Partner boards or custom boards (see Specifying Board Support in System Generator).

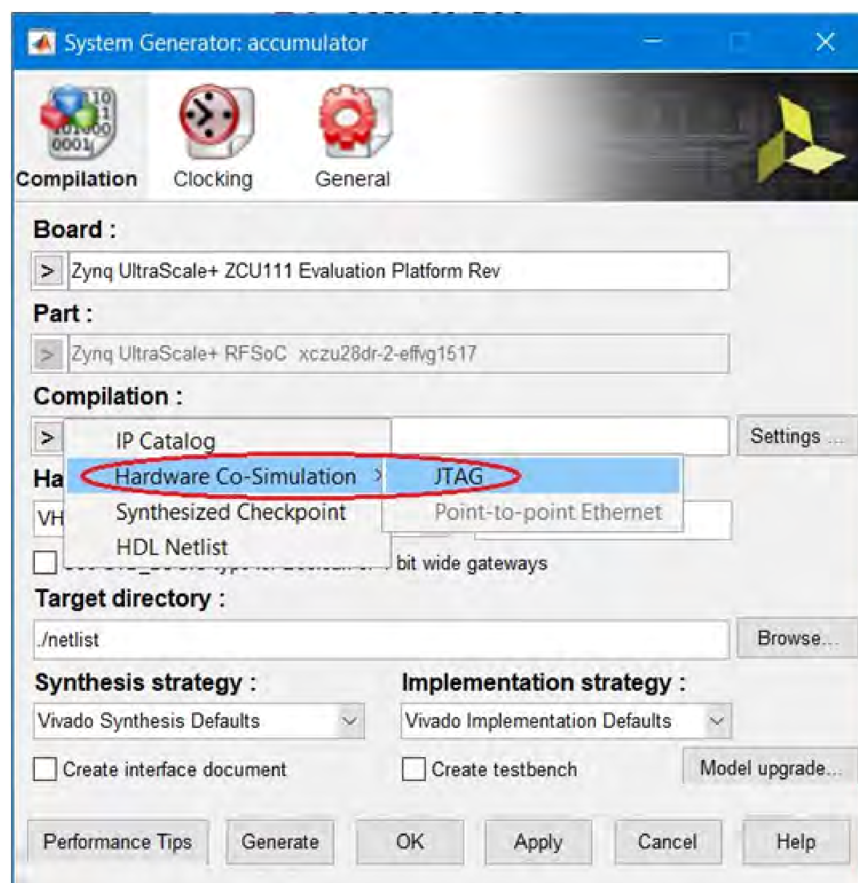
The files generated as part of an HDL Netlist compilation are placed in an `hdl_netlist` subdirectory under the directory you specified in the **Target directory** field. These files are described in the Compilation Results section.

Hardware Co-Simulation Compilation

System Generator can compile designs into FPGA hardware that can be used in the loop with Simulink® simulations. This capability is discussed in the topic Chapter 5: Using Hardware Co-Simulation.

As shown below, you may select **Hardware Co-Simulation** compilation by left-clicking the **Compilation** submenu control on the System Generator token dialog box, and selecting the **Hardware Co-Simulation** target.

Figure 147: **Hardware Co-Simulation**



The **Board** fields allows you to specify the development board you are targeting when you are performing the Hardware Co-Simulation compilation. You can only select a Board for Hardware Co-Simulation compilation - you cannot select a Part. When you select a **Board**, the **Part** field automatically displays the name of the Xilinx® device on the selected **Board**, and this part name cannot be changed.

JTAG Hardware Co-Simulation is supported for all Xilinx development boards. Point-to-Point Ethernet Hardware Co-Simulation is only supported on a KC705 or VC707 board.

The Simulink library (<design_name>_hwcosim_lib.slx) generated as part of a Hardware Co-Simulation compilation is placed in the directory you specified in the **Target directory** field. This library, and the hardware co-simulation block stored in the library, are described in Hardware Co-Simulation Blocks.

IP Catalog Compilation

System Generator uses the **IP Catalog** compilation type as the default generation target.

The IP Catalog compilation target allows you to package your System Generator design into an IP module that can be included in the Vivado IP catalog. From there, the generated IP can be instantiated into another Vivado user design as a submodule.

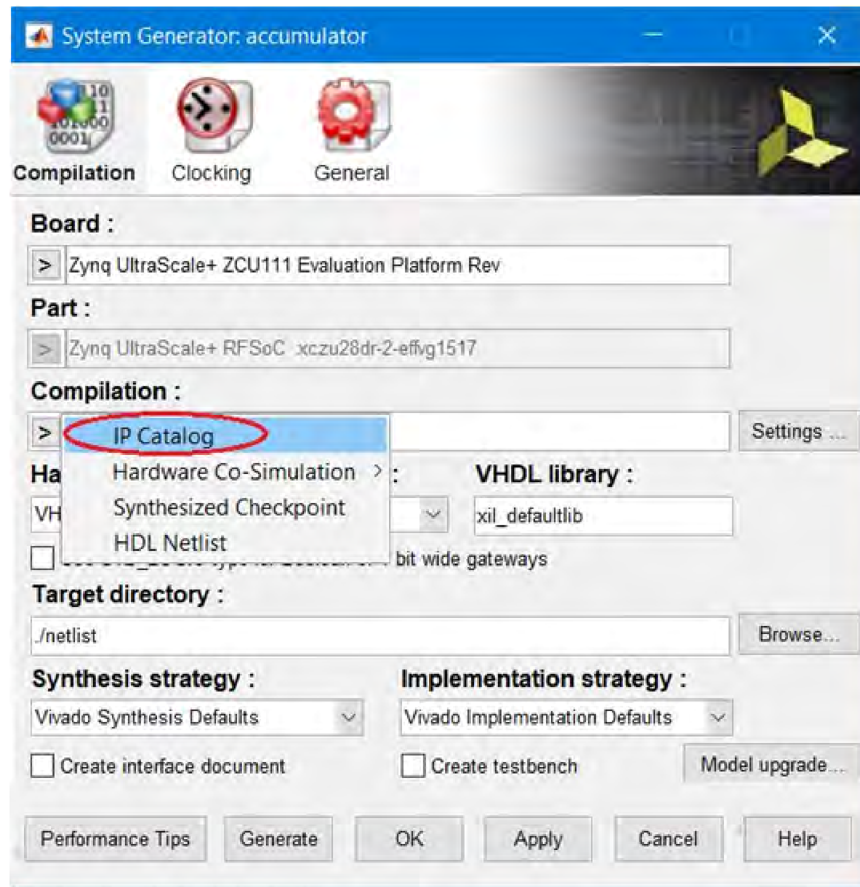
System Generator first generates an HDL NetList based on the block design. If there are Vivado IP modules in the design, all the necessary IP files are copied into a subfolder named `IP`. Finally, all the RTL design files and Vivado IP design files are included into a ZIP file that is placed in a subfolder named `ip_catalog`.

The IP Catalog Flow

In a System Generator design, double-click the System Generator token.

As shown below, under **Compilation**, click the **>** button, then select **IP Catalog**.

Figure 148: **IP Catalog**



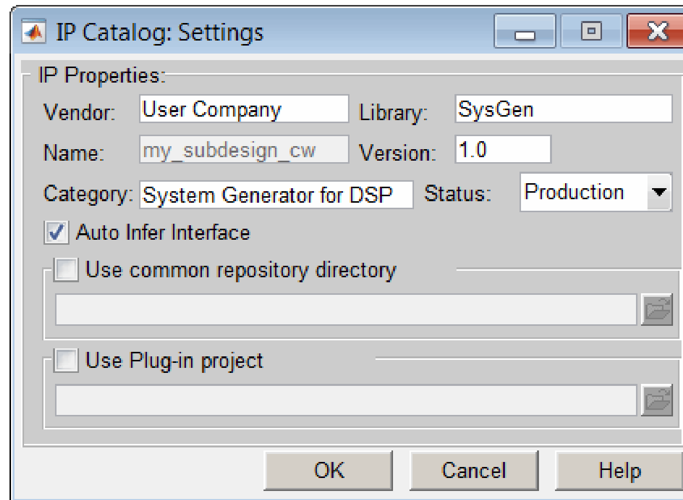
The **Board** and **Part** fields allow you to specify the board or part for which you are targeting the **IP Catalog** compilation. When you select a **Board**, the **Part** field automatically displays the name of the Xilinx device on the selected **Board**, and this part name cannot be changed.

The **IP Catalog** compilation can be performed for any of the boards or parts your Vivado tools support. In addition to accessing the Xilinx development boards installed as part of your Vivado installation, you can also specify Partner boards or custom boards (see Specifying Board Support in System Generator).

The **Target directory** field allows you to specify the location of the generated files.

The **Settings** button activates and when you click on it, a dialog box appears as shown below, allowing you to enter information about the module that will appear in the Vivado IP catalog.

Figure 149: **IP Catalog Settings**

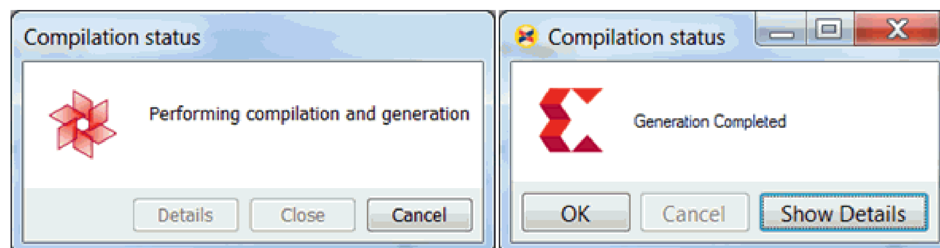


The **Use common repository directory** field allows you to specify a directory referred to as the Common Repository. In an IP catalog compilation, the IP created is copied over to this location. If a Vivado user adds this Path as User Repository in the Vivado project's IP Settings, then all IPs that a System Generator user has placed in this Common Repository will automatically be picked up by Vivado and can be used either in an IP integrator or an RTL flow.

The **Use Plug-in project** field is used to specify a Vivado project containing an IP integrator Block Diagram (BD) that has been imported into System Generator. For an example of a procedure that will need to have a Vivado project specified in this field, see Tailor Fitting a Platform Based Accelerator Design in System Generator.

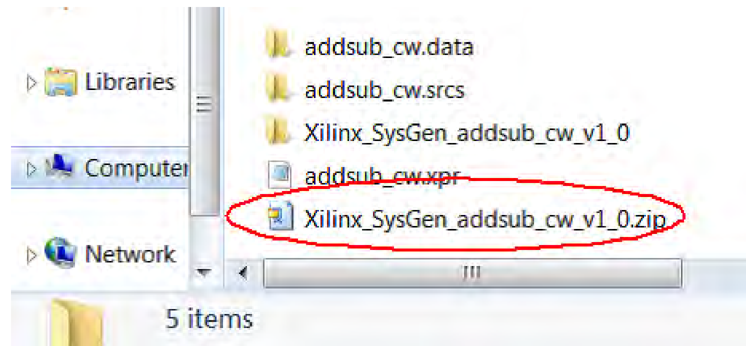
Once you click the **Generate** button, the IP catalog flow starts. As shown below, **Compilation status** windows pop up and indicate the progress of the flow. Once the IP Catalog flow is finished, it will indicate **Generation Completed**. You can then click **Show Details**, to get more detailed information.

Figure 150: **Compilation Status**



Navigate to the specified Target directory, to find a folder named `ip_catalog`. This folder contains all the necessary files to form an IP from your System Generator design. The ZIP file, circled below, contains all the files required to include the System Generator design as IP in the Vivado IP catalog.

Figure 151: **ZIP File**



Using AXI4 Interfaces

Selecting the **Auto Infer Interface** option in the IP Catalog: Settings dialog box ensures AXI4 interfaces are automatically inferred from the design Gateway In and Gateway Out ports. The **Auto Infer Interface** option groups signals into AXI4-Stream, AXI4-Lite and AXI4 interfaces based on the port names.

The **Auto Infer Interface** option will infer interfaces based on the following criteria:

- The Gateway In and Gateway Out port name suffix must exactly match the signal names in the AXI4 interface standard.
- The design must contain the minimum number of signals to be considered a valid AXI4 interface.

For example, if a design has two Gateway In ports named PortName_tdata and PortName_tvalid, and also a Gateway Out port named PortName_tready, the **Auto Infer Interface** option infers these three ports into a single AXI4-Stream port named PortName. In this example.

- The port name suffixes are exact matches for the signals in an AXI4-Stream interface (TDATA, TREADY and TVALID).
- These three signals are the minimum signals required for an AXI4-Stream interface.

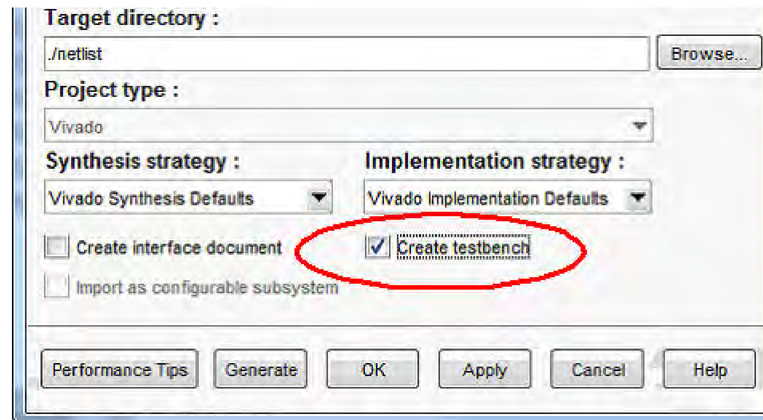
If optional AXI4 sideband signals are present, for example the TUSER signal is optional in the AXI4-Stream standard, and they are named using the same naming convention (for example, PortName_tuser) they will be grouped into the same AXI4 Interface.

For more details on AXI4 interfaces, AXI4 interface signals names and the minimum required signals for an AXI4 interface, refer to the document *Vivado Design Suite: AXI Reference Guide* (UG1037).

Including a Testbench with the IP Module

To verify the functionality of the newly generated IP, it is important to include a test bench. As shown below, if you check **Create testbench**, a test bench is automatically created when you click the **Generate** button.

Figure 152: **Create Testbench**

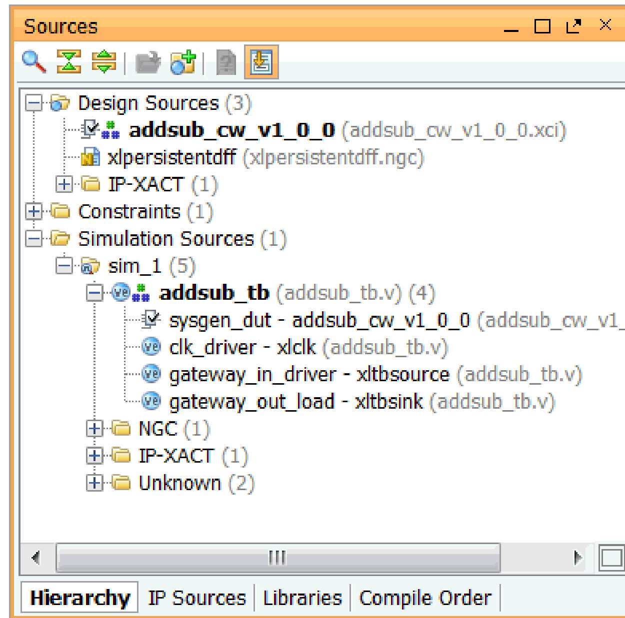


As shown below, when you include a test bench, you can verify the IP functionality by adding three more steps to the flow.

- **Step 1:** Add the new IP to the Vivado IP catalog. Refer to the document *Vivado Design Suite User Guide: Designing with IP* (UG896).
- **Step 2:** Create a new Vivado IDE project and add the IP as the top-level source.
- **Step 3:** Run simulation, synthesis and implementation to verify the functionality of the generated IP.

The following figure shows an open Vivado IDE project with the newly created IP as the top-level source.

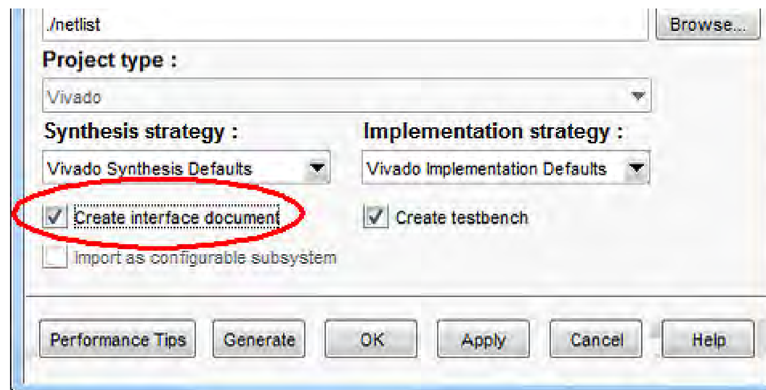
Figure 153: **New IP**



Adding an Interface Document to the IP Module

As shown below, check **Create interface document**, then click **Generate**, and System Generator generates an interface document for the IP and packages this HTML document with the IP.

Figure 154: **Create Interface Document**



You can find a new folder, `documentation`, under the `netlist` folder. Right-click the new IP in the Vivado IDE, and click **Product guide**, to open one HTML file with interface information about this IP.

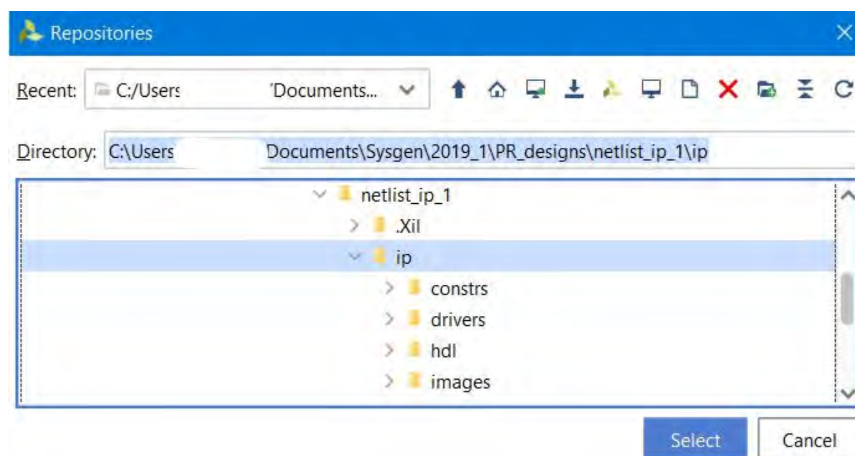
Adding the Generated IP to the Vivado IP Catalog

To use the IP generated from System Generator, you need to create a new project, or open an existing project that targets the same device as specified in System Generator for creating the IP.

***Note*:** The IP is only accessible in this project. For each new project where you use this IP, you need to perform the same steps.

Select **IP Catalog** in the Project Manager, and right-click an empty area in IP Catalog window. Select **Add Repository**, and add the directory that contains your new IP.

Figure 155: **IP Catalog**



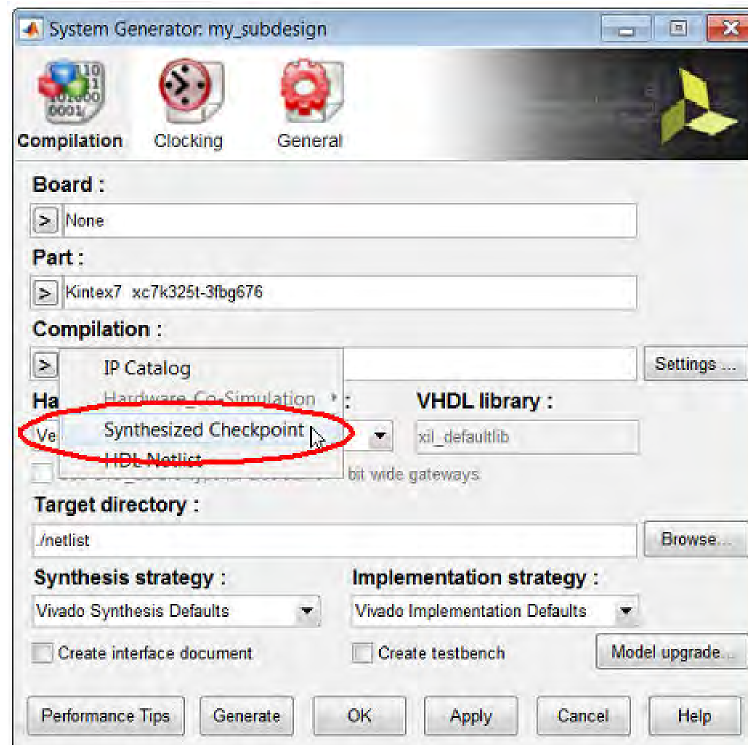
Once the IP is added to the IP catalog, you can include it in larger designs just as you would with any other IP in the IP catalog.

Synthesized Checkpoint Compilation

Vivado tools provide design checkpoint files (`.dcp`) as a mechanism to save and restore a design at key steps in the design flow. Checkpoints are merely a snapshot of a design at a specific point in the flow. A Synthesized Checkpoint is a checkpoint file that is created in the out-of-context (OOC) mode after a design has been successfully synthesized.

When you select the **Synthesized Checkpoint** compilation target (see figure below), a synthesized checkpoint target file named `<design_name>.dcp` is created, and placed in the **Target directory**. You can then use this `<design_name>.dcp` file in any Vivado IDE project.

Figure 156: **Synthesized Checkpoint**



The **Board** and **Part** fields allow you to specify the board or part for which you are targeting the Synthesized Checkpoint compilation. When you select a **Board**, the **Part** field automatically displays the name of the Xilinx device on the selected **Board**. This part name cannot be changed.

The Synthesized Checkpoint compilation can be performed for any of the boards or parts your Vivado tools support. In addition to accessing the Xilinx development boards installed as part of your Vivado installation, you can also specify partner boards or custom boards (see Specifying Board Support in System Generator).

Creating Your Own Custom Compilation Target

System Generator provides a custom compilation infrastructure to create your own custom compilation target. In addition to generating HDL from your System Generator design, you can also create a compilation target plug-in that automates steps both before and after the HDL is generated. Details about how to create a custom compilation target can be found in the topic Chapter 9: Creating Custom Compilation Targets topic.

XILINX

Chapter 9

Creating Custom Compilation Targets

System Generator provides a custom compilation infrastructure that allows you to create your own custom compilation targets. In addition to generating HDL from your System Generator design, you can also create a compilation target plug-in that automates steps both before and after the Vivado® integrated design environment (IDE) project is created. In order to create a custom compilation target, you need to be familiar with the object-oriented programming concepts in the MATLAB® environment.

xilinx_compilation Base Class

The custom compilation infrastructure provides a base class named `xilinx_compilation`. From this base class, you can then create a subclass, and use its properties and override the member functions to implement your own functionality.

Figure 157: **Base Class**

Creating a New Compilation Target

The following general procedure outlines how to create a new compilation target, and is followed by more specific examples.

Running the Helper Function

Create a new custom compilation target by running the following helper function.

```
xilinx.environment.addCompilationTarget(target_name, directory_name)
```

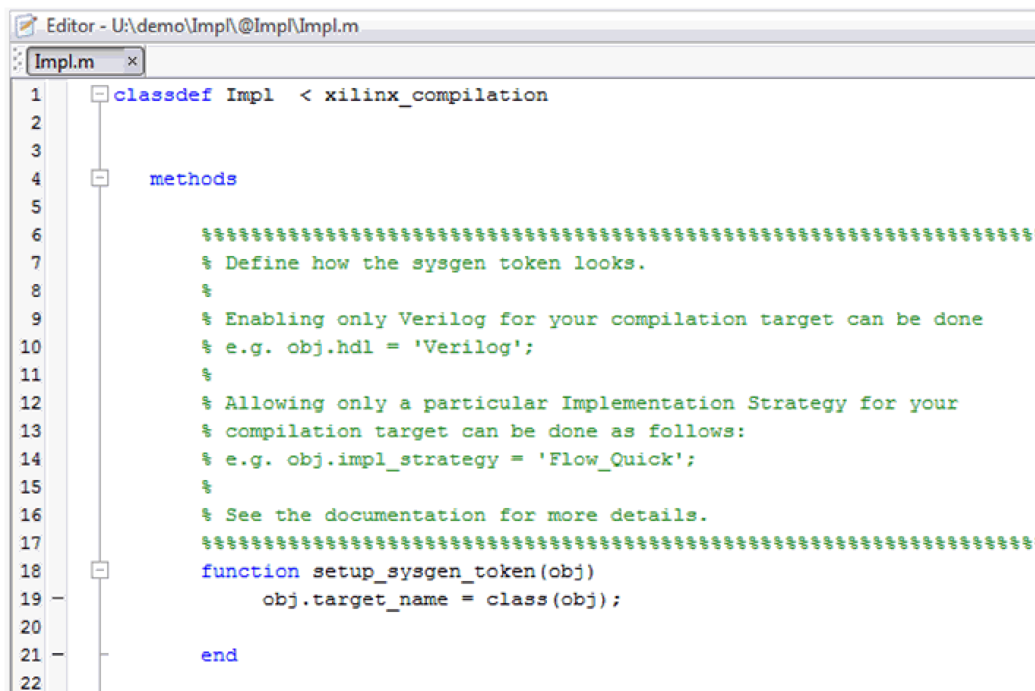
For example, consider the following command:

```
xilinx.environment.addCompilationTarget('Impl', 'U:\demo')
```

Figure 158: **Helper Function**

When you enter this command in the MATLAB Command Window as shown above, the following happens:

1. A folder is created named `Impl/@Impl` in `U:\demo`.
2. Inside the folder, a template class file `Impl` is created (`Impl.m`), which is derived from the base class `xilinx_compilation`. At this point, if no modifications are made to the file, the newly created `Impl` compilation target acts the same as the **HDL Netlist** compilation target. The content of the `Impl.m` file is shown in the following figure.



3. The helper function then adds `U:\demo\Impl` to the MATLAB path, so that the new class `Impl` can be discovered by MATLAB.

Note: Be aware that the `target_name` cannot contain spaces. After the class is created, you can add spaces to the `target_name` property of the class.

Modifying a Compilation Target

If modifications are made to a class file for a compilation target, you are required to call the following helper function. This helper function ensures that System Generator detects the new class definition.

```
>> xilinx.environment.rehashCompilationTarget
```

Adding an Existing Compilation Target

You must add the path that contains the folder with the custom compilation target. As shown below, you can use the `addpath` functionality provided by MATLAB® to do this:

```
>>addpath('U:\demo\Impl');
```

When you use `addpath`, you must provide the absolute path, not the relative path.

Saving a Custom Compilation Target

You can use the `savepath` functionality in MATLAB® to save the custom compilation target. To do the save, you may need write permission to the MATLAB installation area.

Removing a Custom Compilation Target

To remove the custom compilation target, remove the path to the target from the MATLAB® Search Path.

Base Class Properties and APIs

The xilinx_compilation base class resides in the following location:

```
<Vivado Install Path>/scripts/sysgen/matlab/@xilinx_compilation
```

System Generator Token-Related Properties and APIs

setup_sysgen_token()

This function is called to populate the System Generator token information by the Custom Compilation Infrastructure. You can use any of the following functions related to the System Generator token to set how the token looks by default when the custom target is selected. The fields, their default values and the field enablement/disablement can be set by the following System Generator token application programming interface (API) functions.

add_part(family, device, speed, package, temperature)

An example of an explicit command is `add_part('Kintex7', 'xc7k325t', '-1' , 'fbg676','')`. If the part-related APIs are not used, the end user can select any device that he wants to choose from the list.

string target_name

This is a required field that has to be set in the `setup_sysgen_token()` function.

string hdl

The default value is an empty string. Valid options are `verilog` or `vhdl`. Once a value is set to this field, this field will be disabled for further user selection.

string synth_strategy

The default value is an empty string. Once a value is set to this field, this field will be disabled for further user selection. If this API is used, the user has to make sure that the specified strategy exists. Otherwise, it will result in an error.

string impl_strategy

The default value is an empty string. Once a value is set to this field, this field will be disabled for further user selection. If this API is used, the user has to make sure that the specified strategy exists. Otherwise, it will result in an error.

string create_tb

The default value is an empty string. The valid options are `on` or `off`. Once a value is set to this field, this field will be disabled for further user selection.

string create_iface_doc

The default value is an empty string. The valid options are `on` or `off`. Once a value is set to this field, this field will be disabled for further user selection.

Vivado Project-Related Properties

top_level_module

You can use this property to set the top-level name of their choice. This parameter accepts a MATLAB® string.

Vivado IDE Project Generation-Related Functions

pre_project_creation(design_info)

This function should be called before you create the Vivado® IDE project. Before the System Generator Infrastructure creates the project, it has to know what files need to be added to the Vivado® IDE project, and what additional Tcl commands need to be run. There might be use-cases where the user wants to add some files to the project based on the top-level port interface of the System Generator design. For this purpose, a structure that describes the port interface is passed into this function called `design_info`. `design_info` is described in detail in a later section.

post_project_creation(design_info)

This function should be called at the end of Vivado IDE project creation. This is the last function to be called after the Project Generation script is run. This is a useful function for things like error parsing, generating reports, and opening the Vivado IDE project. A structure which describes the port interface is passed into this function called `design_info`. `design_info` is described in detail in a later section.

add_tcl_command(string)

This function adds the additional Tcl commands as a string. These Tcl commands will be issued after the Vivado IDE project is created. Use this command to create a bitstream once project creation occurs. The Tcl command can also be used to source a particular Tcl file. The commands are executed in the order in which they are received.

add_file(string)

This function adds user-defined files to the Vivado IDE project. This application programming interface (API) function can also be used to add XDC constraint files to the Vivado IDE project. You should make sure that the order in which add_file is called, is hierarchical in nature. The top-module file must be added last.

run_synthesis()

This function runs synthesis in the Vivado IDE project.

run_implementation()

This function runs implementation in the Vivado IDE project.

generate_bitstream()

This function generates a bitstream in the Vivado IDE project.

Design Info

`design_info` is a MATLAB® struct and its contents are shown below:

Figure 159: **Design Info**

design_info | design_info.ports | design_info.ports.gateway_in

design_info.ports.gateway_in <1x1 struct>

Field	Value	Min	Max
ArithmeticType	'xlSigned'		
BinaryPoint	14	14	14
DatFile	'adder_gateway_i...		
Direction	'in'		
IconText	'Gateway In'		
IsClock	0	0	0
Name	'gateway_in'		
Period	1	1	1
Type	'Fix_16_14'		
Width	16	16	16

design_info | design_info.ports | design_info.ports.gatew

design_info <1x1 struct>

Field	Value	Min	Max
ports	<1x1 struct>		
sim_time	10	10	10
target_dir	'/group/dspuser...		
testbench	'on'		
top_level	'adder'		

Examples of Creating Custom Compilation Targets

The following examples provide more detail on how you can create various kinds of customized targets.

Example 1: Creating an Implementation Target

1. Open a System Generator model, then open the System Generator token. This populates the token with all the available compilation targets.
2. In the MATLAB® Command Window, modify the path as per your requirements, then enter the following command:

```
xilinx.environment.addCompilationTarget('Impl', 'U:\demo')
```

This provides a template derived class for you to edit.

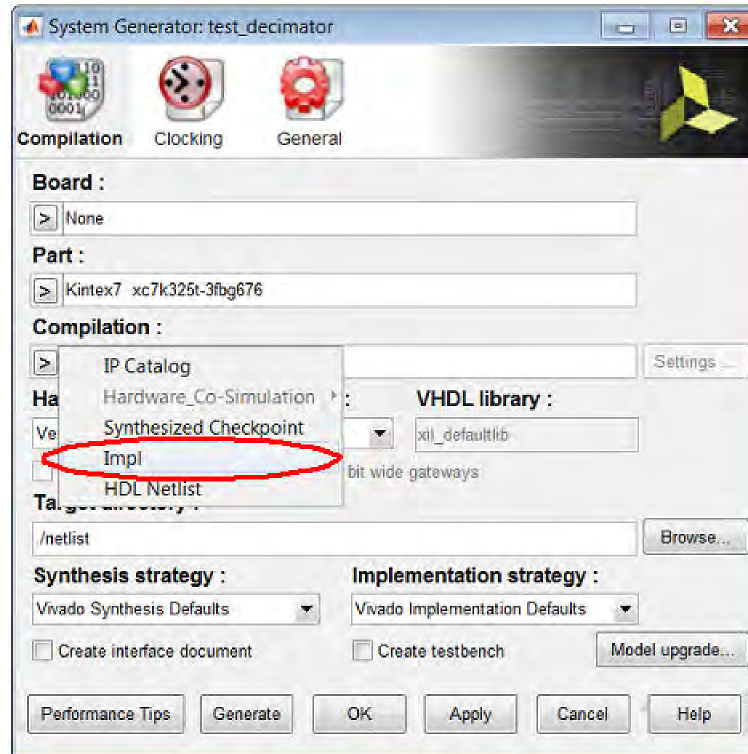
3. In the MATLAB Command Window, enter the following command:

```
xilinx.environment.rehashCompilationTarget
```

   This ensures that the new compilation target is picked up by the System Generator token.

4. Close and then re-open the System Generator token. You now see the compilation target, **Impl** on the token as shown below.

Figure 160: **Selecting Impl**



5. At this point, selecting **Impl** does not perform any customized operations on the System Generator token. It is equivalent to an HDL Netlist compilation target.

6. Open `U:\demo\Impl\@Impl\Impl.m` in the MATLAB Editor.

7. Populate the `setup_sysgen_token()` function as per the requirements. Using this approach, you can control how the System Generator token should look, including the enabled/disabled fields when the user-defined custom compilation is selected.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Define how the sysgen token looks.
%
% Enabling only Verilog for your compilation target can be done
% e.g. obj.hdl = 'Verilog';
%
% Allowing only a particular Implementation Strategy for your
% compilation target can be done as follows:
% e.g. obj.impl_strategy = 'Flow_Quick';
%
% See the documentation for more details.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function setup_sysgen_token(obj)
     obj.target_name = class(obj);
     obj.hdl = 'Verilog';
     obj.impl_strategy = 'Flow_Quick';
end
```

8. In the MATLAB Command Window, you should enter the following command:

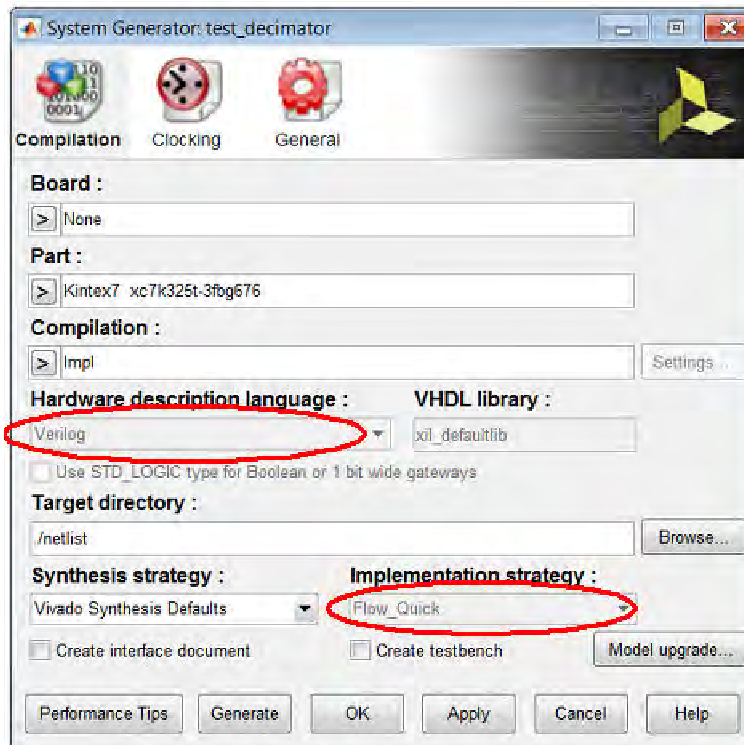
```
xilinx.environment.rehashCompilationTarget
```

   This ensures that the updated class definition of **Impl** is used.

9. Close and then re-open the System Generator token. Select **Impl** from the list of Compilation targets.

10. The System Generator token appears as follows:

Figure 161: **Selecting Verilog, and Flow Quick**



11. Observe that the **Hardware description language** field and the **Implementation strategy** field are fixed to what you set in the **Impl** class and are disabled for user modification.
12. All the user specified files and additional Tcl commands to be run are known before the Vivado® IDE project is created. The next step is to populate the `pre_project_creation()` function as indicated below:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Define how the Project should be generated. Adding tcl commands,
% files etc. should be done here.
%
% e.g. obj.add_tcl_command('launch_runs synth_1');
% e.g. obj.add_file('C:\work\myconstraints.xdc');
% e.g. obj.run_implementation()
%
% design_info is the struct that contains the information about the
% design and its interface. See documentation for more details
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function pre_project_creation(obj, design_info)
     obj.add_tcl_command('launch_runs synth_1');
     obj.add_tcl_command('wait_on_run synth_1');
     obj.run_implementation();
end
```

13. In the MATLAB Command Window, enter the following command:

    ```
    xilinx.environment.rehashCompilationTarget
    ```

    This ensures that the updated class definition of Impl is used.

14. Close and then re-open the System Generator token. Select **Impl** from the list of Compilation targets.

15. Click **Generate**. Once the process is finished, you can see the implementation results by opening up the Vivado IDE project.

Example 2: Creating a Bitstream Target

1. Open a System Generator design.

2. In the MATLAB command Window, modify the path as per your requirements, similar to the first example, and then enter the following command:

   ```
   xilinx.environment.addCompilationTarget('Bitstream', '.')
   ```

   This provides a template derived class for the users to edit. The last field corresponds to the directory which contains the `board.xml` file.

3. In the MATLAB Command Window, enter the following command:

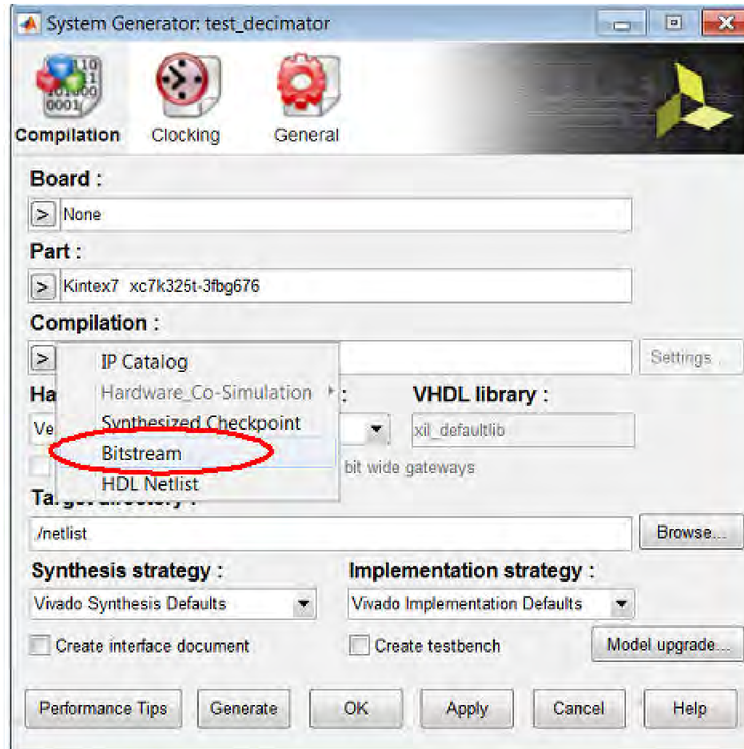
   ```
   xilinx.environment.rehashCompilationTarget
   ```

   This will ensure that the new compilation target is picked up by the System Generator token

4. Close and then re-open the System Generator token.

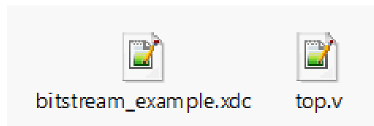
5. You will now see the compilation target **Bitstream** on the System Generator token as shown below.

Figure 162: **Bitstream**



6. Open the `Bitstream.m` created in the '`./Bitstream/@Bitstream/Bitstream.m`'.

7. Download the following two files:



8. Inside the function `pre_project_creation()`, add the following lines to do the following:

   a. Set the board as a KC705 board.

   b. Add a new top-level file (`top.v`) to use the differential clock ports of KC705.

   c. Add a new XDC file to give the location constraints for the clock, dip, and led ports.

   d. Set the newly added module `top` as the top.

   e. Run synthesis.

   f. Run implementation.

   g. Generate bitstream.

After you save the files to a location on your computer, you should give the full path to the files in the add_file API as per your path.

```
add_tcl_command(obj, 'set_property board xilinx.com:kintex7:kc705:1.1
[current_project]');
add_file(obj,
'/group/dspusers-xsj/umangp/rel/2013.4/cust_comp_test/
bitstream_example.xdc');
add_file(obj, '/group/dspusers-xsj/umangp/rel/2013.4/cust_comp_test/
top.v');
obj.top_level_module = 'top';
run_synthesis(obj);
run_implementation(obj);
generate_bitstream(obj);
```

9. In the MATLAB Command Window, enter the following command:

```
xilinx.environment.rehashCompilationTarget
```

This ensures that the new compilation target is picked up by the System Generator token

10. Close and then re-open the System Generator token.
11. Select the **Bitstream** compilation target.
12. Click the **Generate** button.
13. After the generation is complete, you can find the bit file in the following directory:

```
./<Target_directory>/Bitstream/bitstream_example.runs/impl_1/top.bit
```

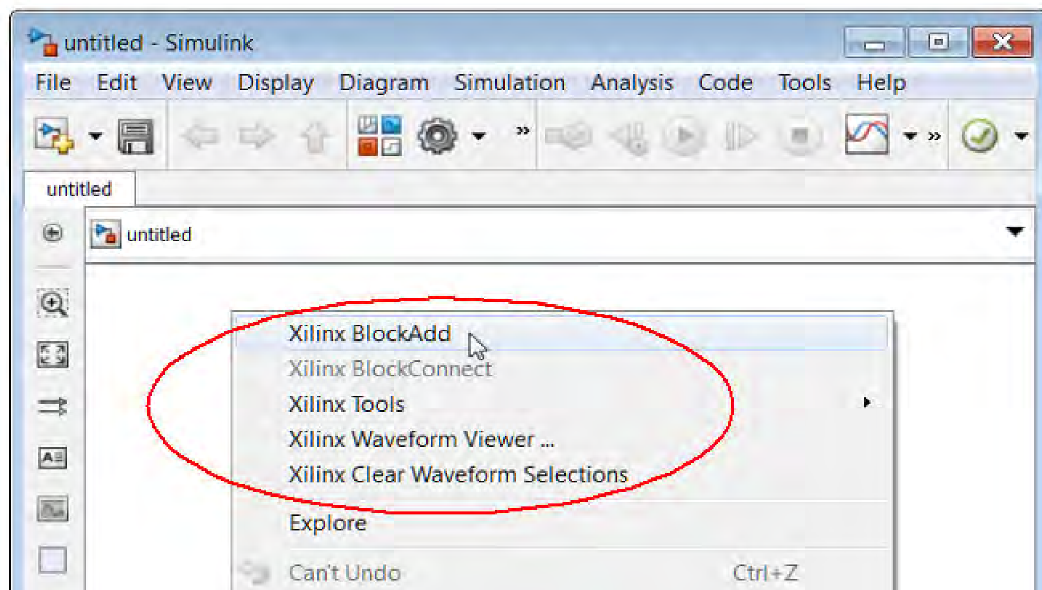
![XILINX logo]

Appendix A

System Generator GUI Utilities

Xilinx has added graphics commands to the Simulink® model popup menu that will help you rapidly create and analyze your System Generator design. As shown below, you can access these commands by right-clicking the Simulink model canvas and selecting the appropriate Xilinx command:

Figure 163: **Xilinx Commands**



A detailed description of the additional Xilinx commands is provided below.

Xilinx BlockAdd	Facilitates the rapid addition of Xilinx blocks (and a limited set of Simulink blocks) to a Simulink model.
Xilinx Tools > Save as blockAdd default	This feature allows you to pre-configure a block, then add multiple copies of the pre-configured block using the BlockAdd feature.
Xilinx BlockConnect	Facilitates the rapid connection of blocks in a Simulink model
Xilinx Tools > Terminate	Facilitates the rapid addition of Simulink terminator blocks on open output ports and/or Xilinx Constant Blocks on open input ports.

Xilinx Waveform Viewer	The Xilinx Waveform Viewer displays a waveform diagram of selected signals in your System Generator design. Waveforms can be displayed in the Waveform Viewer after running a Simulink simulation. Inputs and outputs of blocks in the Xilinx Blockset can be displayed in the Waveform Viewer.
Xilinx Clear Waveform Selections	Deletes all of the waveforms currently displayed in the Waveform Viewer, and closes the Waveform Viewer.

Xilinx BlockAdd

Facilitates the rapid addition of Xilinx® blocks (and a limited set of Simulink® blocks) to a Simulink model.

How to Invoke

Method 1

Right-click the Simulink canvas and select **Xilinx BlockAdd**.

Method 2

Execute the short cut Ctrl 1 (one).

Method 3

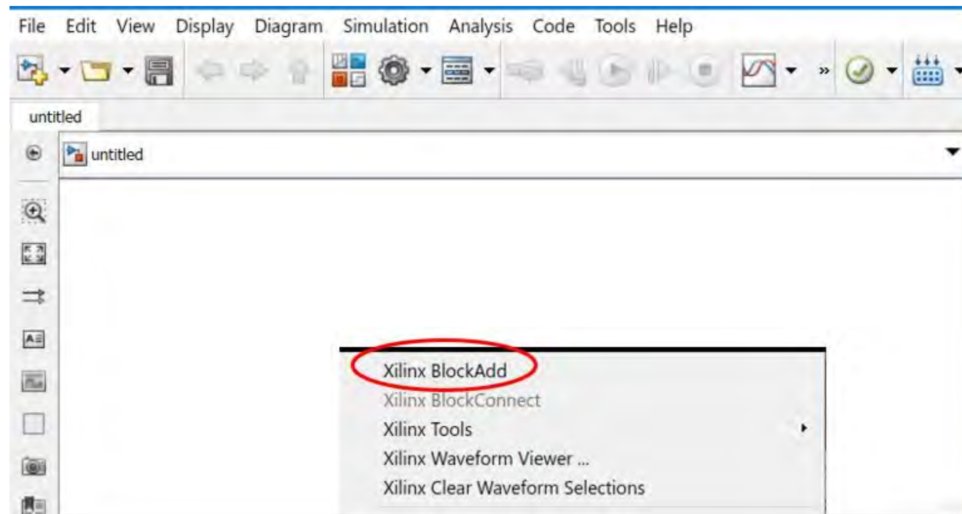
From the Simulink model pull down menu, select the following item:

Tools→Xilinx→BlockAdd

How to Use

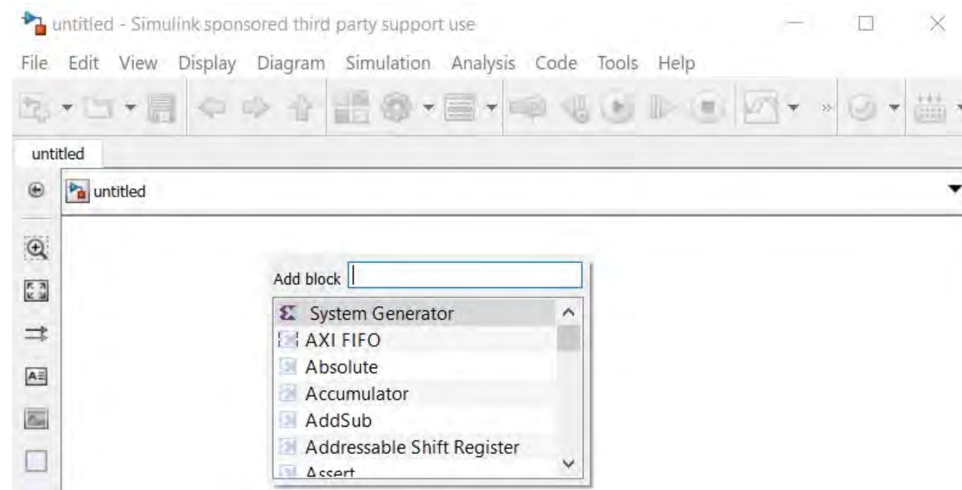
1. Right-click the Simulink canvas and select **Xilinx BlockAdd**.

Figure 164: **Xilinx BlockAdd**



2. Double-click **AddSub**.

Figure 165: **AddSub**



3. To add multiple copies of the same block, add a block, select the block, press **Ctrl + C**, then press **Ctrl + V** the required number of times.

4. To dismiss the Add block window, press **Esc**.

Xilinx Tools > Save as blockAdd default

This feature allows you to pre-configure a block, then add multiple copies of the pre-configured block using the BlockAdd feature.

How to Use

Assume you need to add multiple Gateway In blocks of type **Boolean** to a model.

1. Add one Gateway In block to the model.
2. Double click the **Gateway In** block, change the **Output type** to **Boolean** and click **OK**.
3. Select the modified **Gateway In** block, right-click and select **Xilinx Tools → Save as blockAdd default**.
4. Now, every time you add addition Gateway In blocks to the model using the BlockAdd feature, the block is of Output type Boolean.

How to Restore the Block Default

1. Select a block with pre-configured (changed) defaults.
2. Right-click and select **Xilinx Tools → Clear blockAdd defaults**.

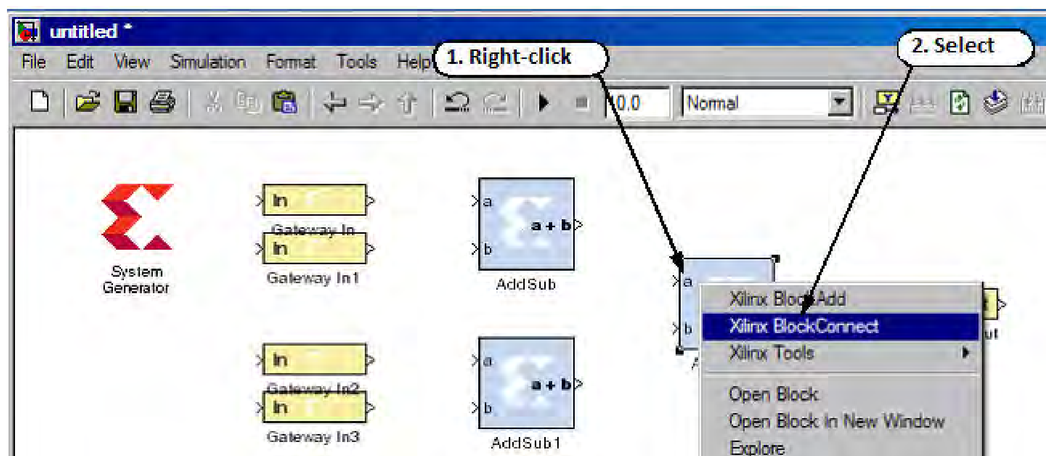
Xilinx BlockConnect

Facilitates the rapid connection of blocks in a Simulink® model.

Simple Connections

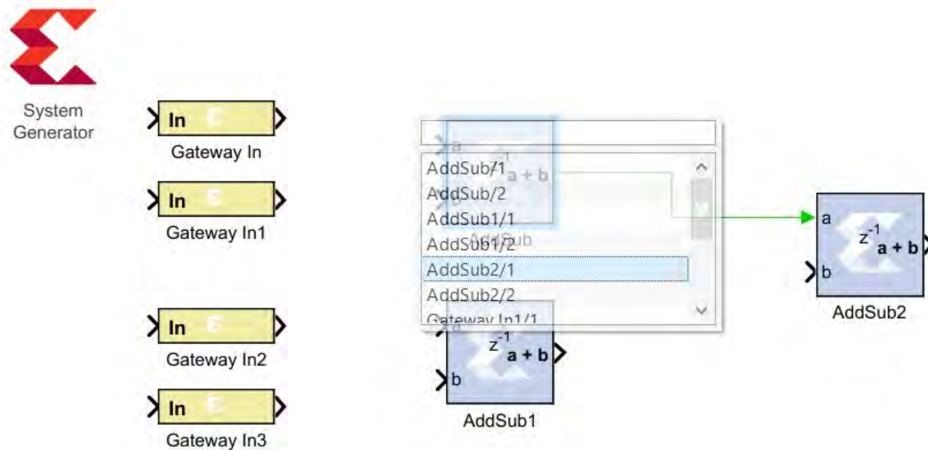
1. As shown below, select an open port of a block, right-click, keep the cursor on a particular port then right click and select **Xilinx BlockConnect** option to see the list of possible connections for that port. **Xilinx Block Connect** only shows list of possible connections for one port at a time.

Figure 166: **Xilinx BlockConnect**



2. BlockConnect proposes the nearest connection with a green line. To confirm, you can double-click the selected connection in the table. The connection then turns black. Otherwise, select another connection in the table to see if the new green line connection is correct.

Figure 167: **Connecting Blocks**

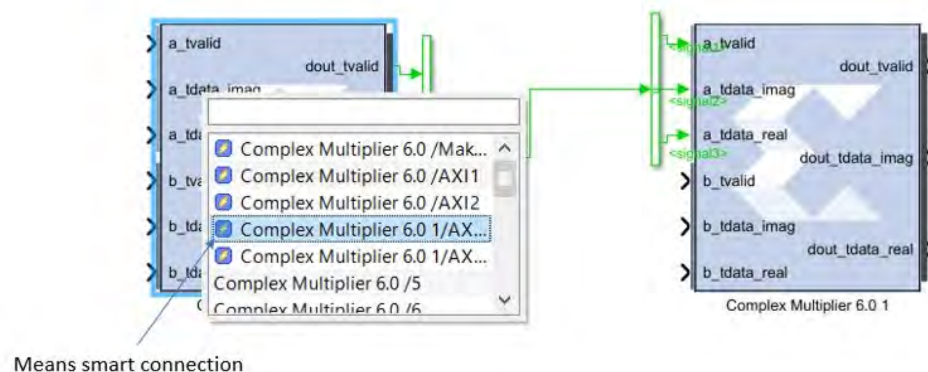


Smart Connections

As shown in the following figure, a "lightning bolt" icon indicates a "smart" connection. Smart connections have intelligence built in to help you manage the connection. For example, right-clicking a block with an AXI interface allows you to:

- Group or separate the AXI signals to or from a bus.
- Connect to other ports with the same number of AXI connections.

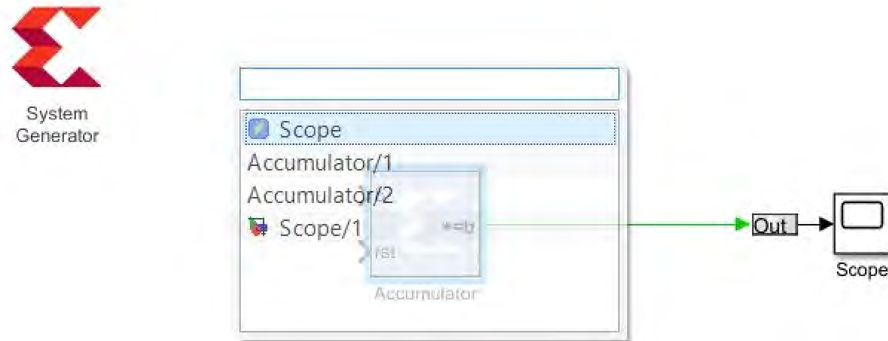
Figure 168: **Smart Connections**



No port data type checking is performed and any AXI ports with the same number of ports are allowed to connect.

In another smart connection example below, right-clicking the **Accumulator** block output, selecting **BlockConnect**, and double-clicking **Scope** creates a smart connection to the Scope block. The Gateway Out block is added automatically.

Figure 169: **Connecting Scope Block**



If a second connection is made to this Scope block, a second port is automatically added to the Scope. The driving signal name is also used to name the signal driving the scope.

Xilinx Tools > Terminate

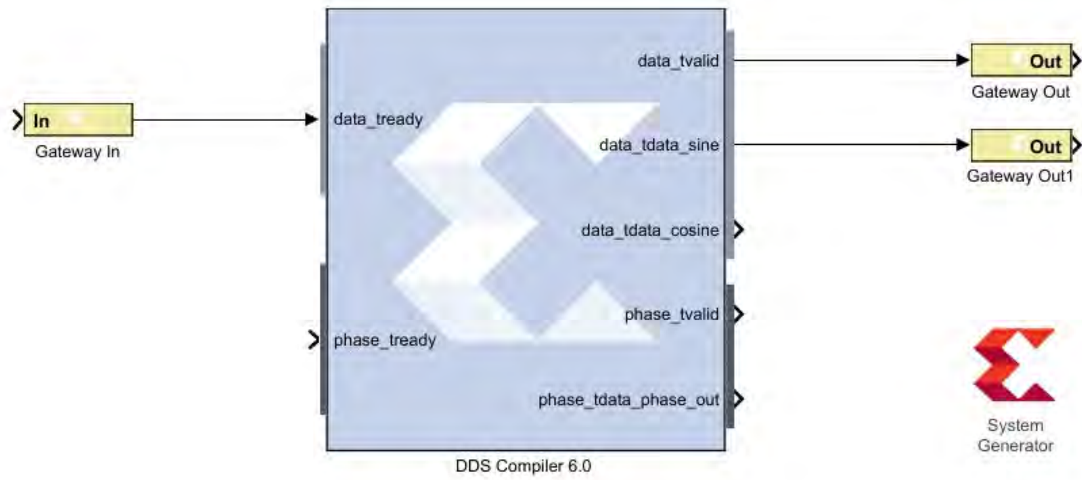
Facilitates the rapid addition of Simulink® terminator blocks on open output ports and/or Xilinx® Constant Blocks on open input ports.

How to Use

Terminating Open Outputs

Consider the following model with open input and output ports:

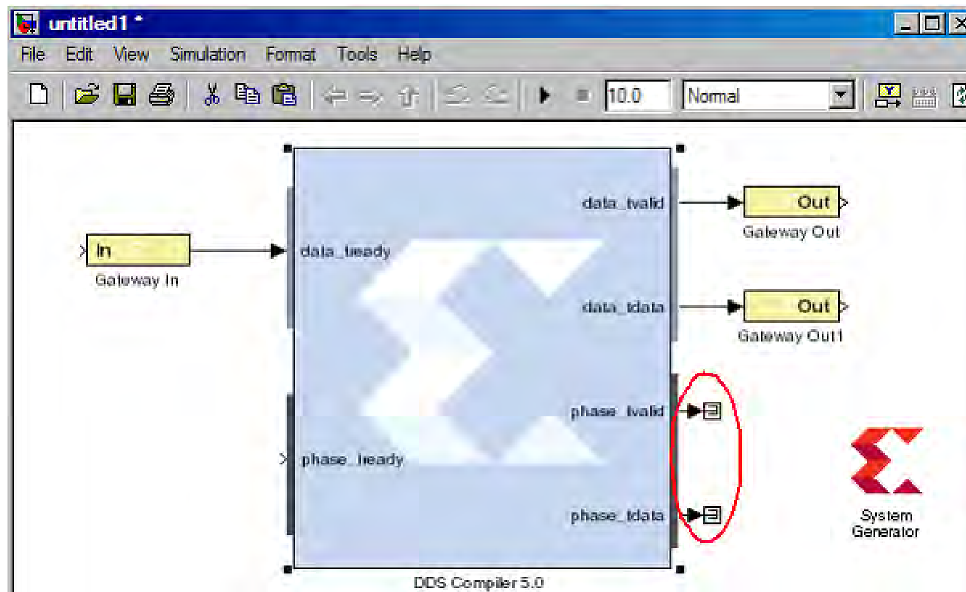
Figure 170: **Model with Open Input and Output Ports**



Right-click the DDS Compiler 5.0 block in this case and select **Xilinx Tools → Terminate → Outputs**.

The following figure illustrates the resulting terminated outputs.

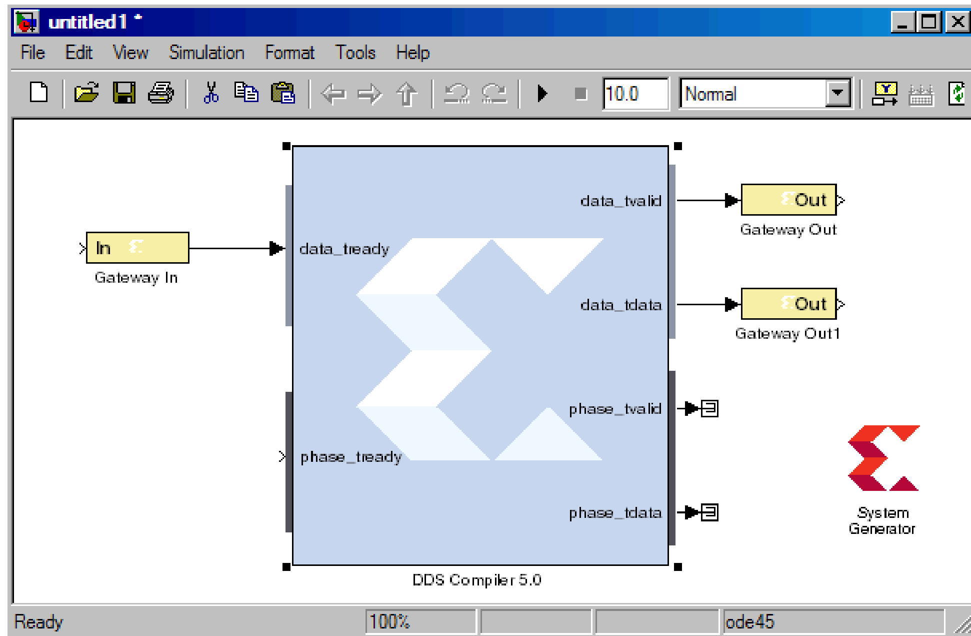
Figure 171: **Terminated Output Ports**



Terminating Open Inputs

Consider the following model with an open input port:

Figure 172: **Model with Open Input Port**

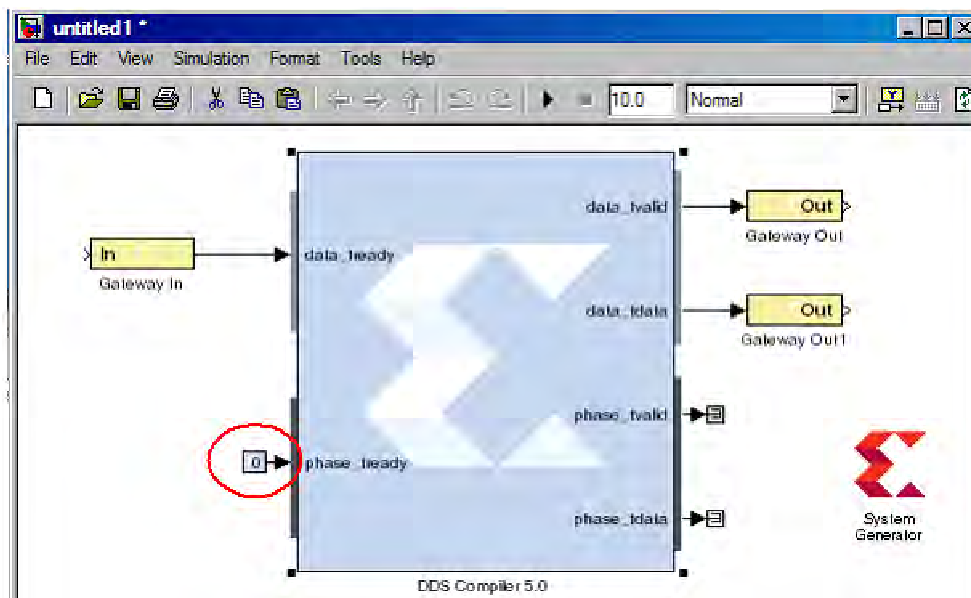


Right-click the DDS Compiler 5.0 block and select:

Xilinx Tools → Terminate → Inputs

The following figure illustrates the resulting terminated input.

Figure 173: **Terminated Input Port**



Verifying Input Port Data Type Requirements

System Generator connects each open input port to a Xilinx® Constant Block. The new Constant blocks are set to the following default values:

Type: Signed (2's comp)

Constant value: 0

Number of bits: 16

Binary point: 14

This terminate tool does not do data type checking on the input ports. If an open port requires a different data type, for example a Boolean data type, you will need to double-click the Constant block and change the Output Type to Boolean.

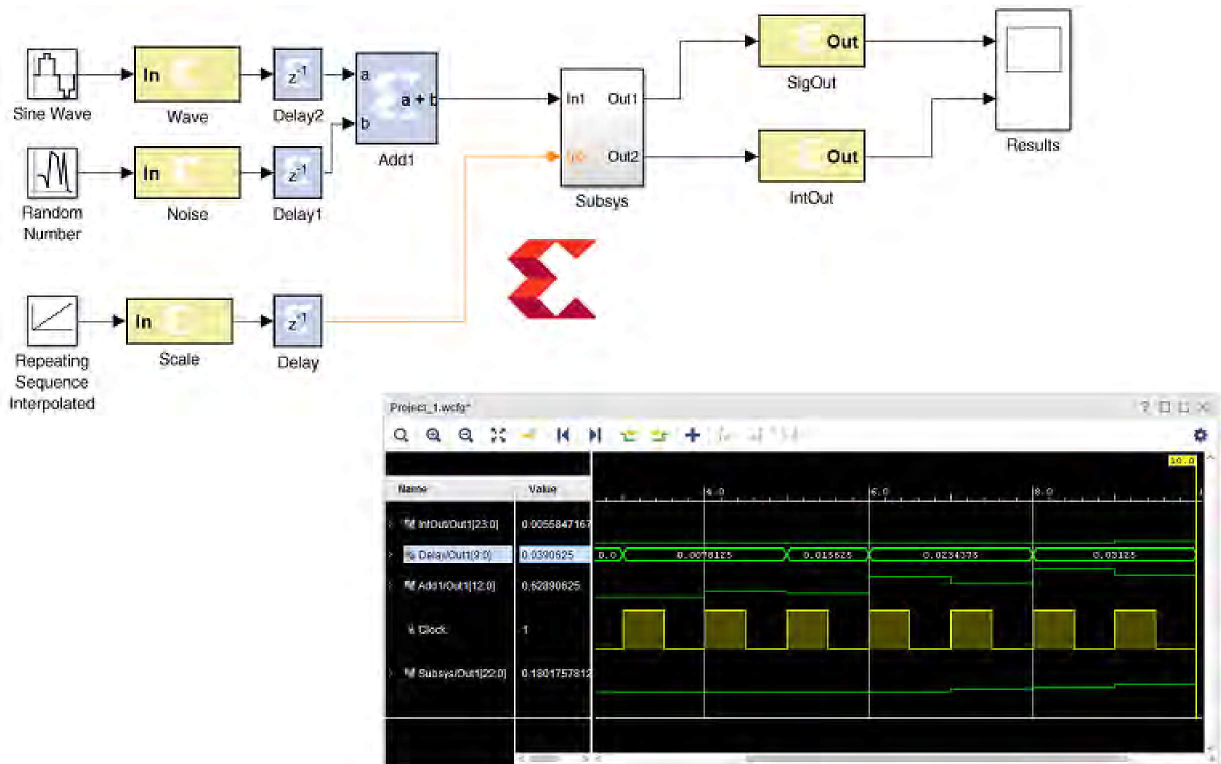
To check for data type mismatches, click the Simulink® model canvas, and enter Ctrl-D. System Generator will report on all the data type mismatches, if there are any.

Xilinx Waveform Viewer

The Xilinx® Waveform Viewer displays a waveform diagram of selected signals in your System Generator design. Waveforms can be displayed in the Waveform Viewer after running a Simulink simulation. Inputs and outputs of blocks in the Xilinx Blockset can be displayed in the Waveform Viewer.

In your design, you can select the signals that will be monitored in the Waveform Viewer. As you develop and troubleshoot your design, the waveforms for the signals you are monitoring will be updated in the Waveform Viewer each time you simulate the model.

Figure 174: **Example Design in Waveform Viewer**



The Xilinx Waveform Viewer used with System Generator is also used by other tools in the Vivado® toolset. The Waveform Viewer is used to analyze a design and debug code in the Vivado simulator and to display data captured by the Integrated Logic Analyzer (ILA) for in-system debugging.

For information on using the Waveform Viewer to develop and troubleshoot your design, see this link in the *Vivado Design Suite User Guide: Logic Simulation* (UG900).

Waveform Viewer Files

The first time you open the Waveform Viewer for your Simulink model, System Generator creates a `wavedata` directory in the directory containing your Simulink model.

Note: You will need write permission for the directory containing your Simulink model.

Data describing the display in the Waveform Viewer is stored in the following files in the `wavedata` directory:

- `<design_name>.wcfg` - This is the waveform configuration file. It contains the names of the signals you are monitoring in your design and how the waveforms for these signals will appear in the Waveform Viewer.

- `<design_name>.wdb` - This is the waveform database file. It contains the data necessary to draw the waveforms in the Waveform Viewer.

The names of the signals that are being monitored are stored in the Simulink model (SLX file). If the Simulink model cannot access the data in the `wavedata` directory (for example, if you moved the model's SLX file to a different directory and opened it in the new directory), you can display the monitored signals by opening the Waveform Viewer and simulating the design. The waveforms for the monitored signals will then appear in the Waveform Viewer.

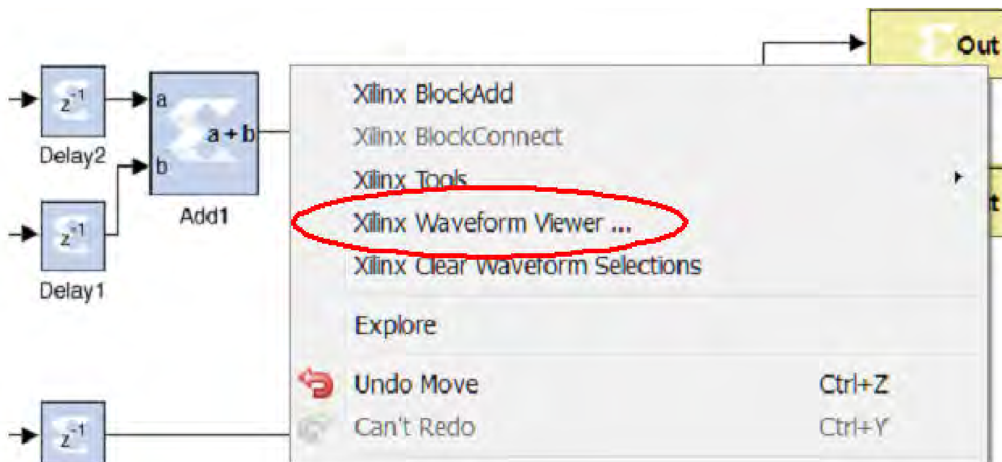
Opening the Xilinx Waveform Viewer

Youc can open the Waveform Viewer in either of the following ways:

- Opening from right-click menu:

  Right-click in your model and select **Xilinx Waveform Viewer**.

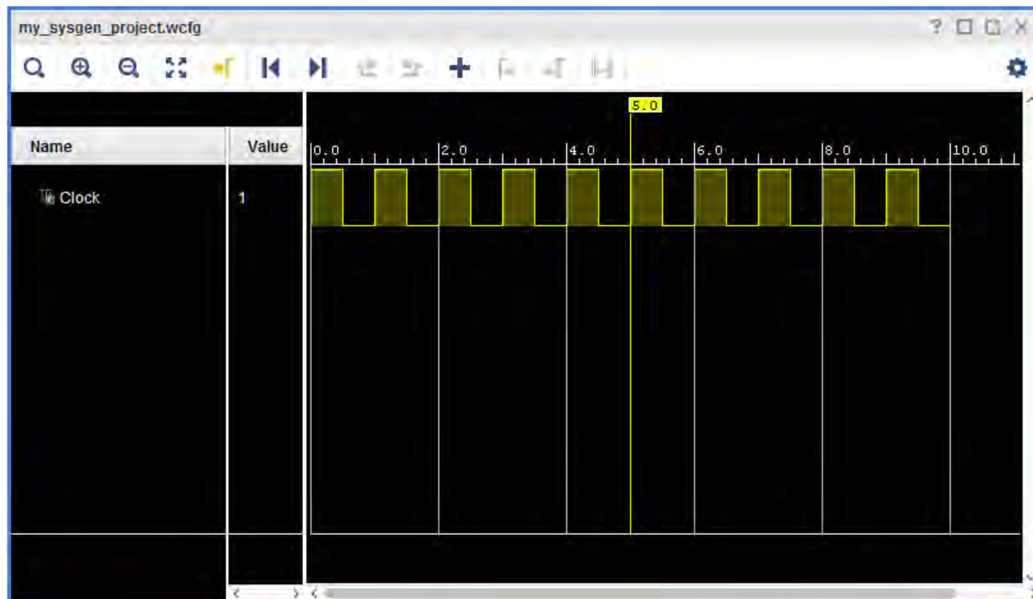
Figure 175: **Xilinx Waveform Viewer**



If you open it from the right-click menu, the Waveform Viewer opens with the following display:

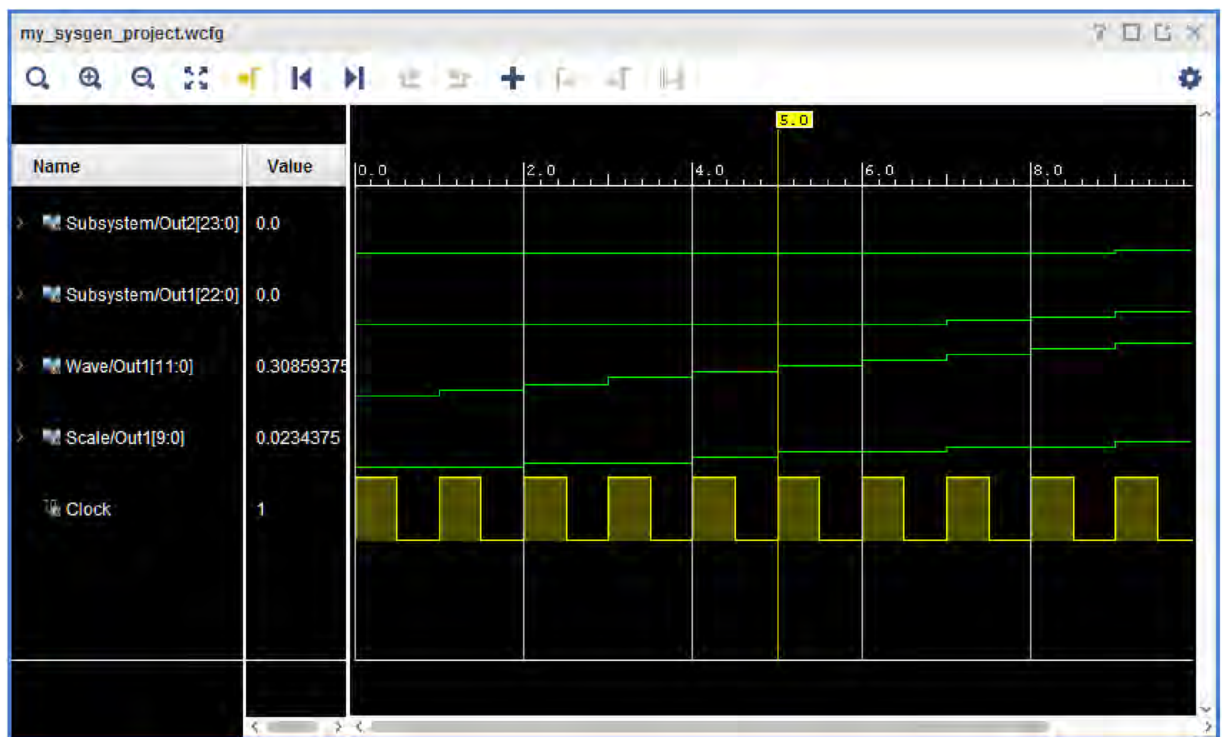
- If this is the first time you are opening the Waveform Viewer for this design, the Waveform Viewer opens displaying waveforms for the clock signals in your design, and no other waveforms. You can then add the signals in your design that you want monitored to the Waveform Viewer display (see Adding Signals to the Waveform Viewer Display).

Figure 176: **Waveform Viewer Display**



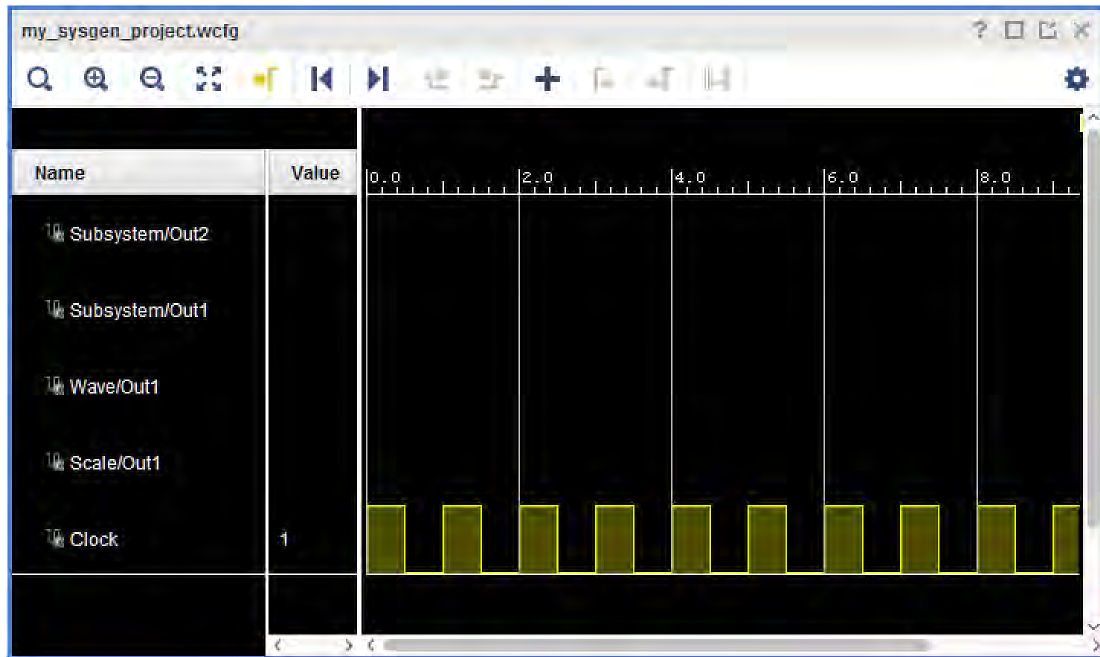
- If you have previously monitored signals in the Waveform Viewer for this design, and have saved the data, the Waveform Viewer opens displaying the signal names, and waveforms displayed when you last closed the Waveform Viewer.

Figure 177: **Signals in Waveform Viewer**



- If you have previously monitored signals in the Waveform Viewer for this design, but cannot access the saved data (for example, if you moved the model's SLX file to a different directory, and opened it in the new directory), the Waveform Viewer opens displaying the signal names for the signals monitored when you last saved the model. The Waveform Viewer will not show the waveforms for the monitored signals until you resimulate the model.

Figure 178: **Waveform Viewer New Signals**



- Opening after simulation:

  If you have previously monitored signals in the Waveform Viewer for your design, the Waveform Viewer will open automatically when you simulate your model.

Adding Signals to the Waveform Viewer Display

Inputs and outputs of blocks in the Xilinx® Blockset can be displayed in the Waveform Viewer. The data necessary to draw each signal's waveform is not stored with the design; it is generated by simulation. You can only display a signal's waveform after you have added the signal to the Waveform Viewer and then simulated the model.

To add signals to the display in the Waveform Viewer:

1. With the Waveform Viewer open, select a signal in the System Generator model.

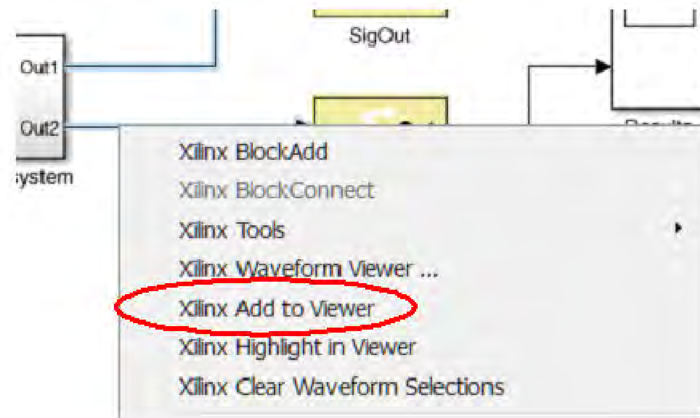
   You can also select multiple signals by using **Shift+** click to select additional signals.

   ***Note*:** For the Gateway In block, only the output signal can be displayed in the Waveform Viewer.

2. Right-click one of the selected signals in the System Generator model and select **Xilinx Add to Viewer** in the right-click menu.

   Note: If you select a signal that is currently displayed in the Waveform Viewer, the **Xilinx Add to Viewer** entry will not appear in the right-click menu.

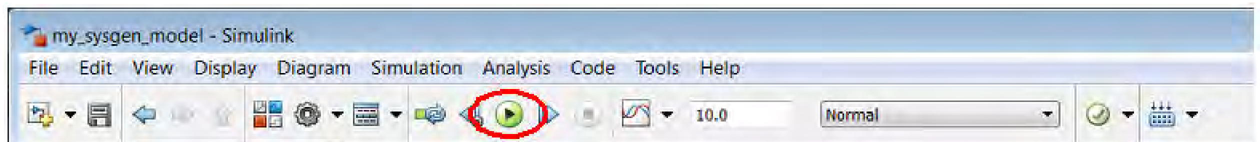
   Figure 179: **Xilinx Add to Viewer**

   The signal names of the selected signals appear in the Waveform Viewer.

3. Only the names of the added signals appear in the Waveform Viewer, because the Waveform Viewer does not have the data to draw the signal's waveform until you simulate the design.
4. Simulate the model.

   Figure 180: **Run Button**

   After the simulation is finished, the waveforms for the added signals are displayed in the Waveform Viewer.

Deleting Signals From the Waveform Viewer Display

1. In the Waveform Viewer, select the signals to be deleted.

   Use **Shift+**click or **Ctrl+**click to select multiple signal names (**Ctrl+A** to select all).
2. Right click one of the selected names and select **Delete** in the right-click menu.

   OR

   Press the **Delete** key.

The waveforms are deleted from the Waveform Viewer. Deleted waveforms are no longer monitored; if you resimulate the model the deleted waveforms will not appear in the Waveform Viewer.

Cross Probing Between the Waveform Viewer and the Model

Cross probing helps you correlate the waveforms in the viewer to the wires in the System Generator model.

You can cross probe signals between the Waveform Viewer, and the model in the following ways:

- To cross probe a signal from the Waveform Viewer to the System Generator model, select one or more signal names in the Waveform Viewer. Use **Shift+**click or **Ctrl+**click to select multiple signal names (**Ctrl+A** to select all).

  The selected signals are highlighted in orange in the System Generator model.

  To unhighlight a signal you have highlighted in the System Generator model, **Ctrl+**click the signal name in the Waveform Viewer. The signal is unhighlighted in the System Generator model.

- To cross probe a signal from the System Generator model to the Waveform Viewer:

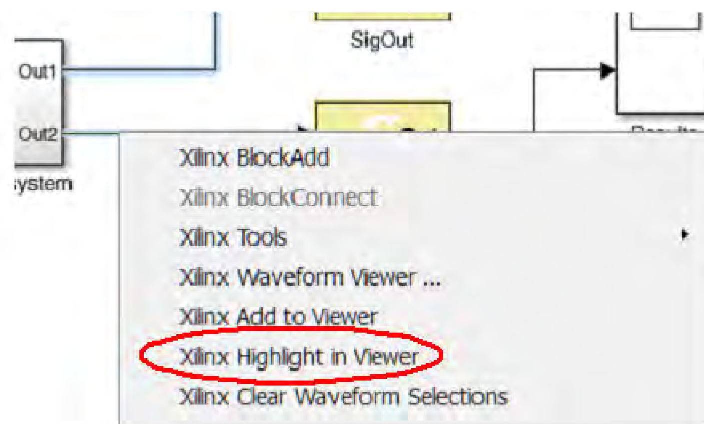
  1. With the Waveform Viewer open, select a signal in the System Generator model.

     You can also select multiple signals by using **Shift+**click to select additional signals.

  2. Right-click one of the selected signals in the System Generator model and select **Xilinx Highlight in Viewer** in the right-click menu.

     ***Note*:** If you select a signal that is not currently displayed in the Waveform Viewer, the **Xilinx Highlight in Viewer** entry will not appear in the right-click menu.

Figure 181: **Xilinx Highlight in Viewer**



3. Observe that the signal names of the selected signals are highlighted in the Waveform Viewer.

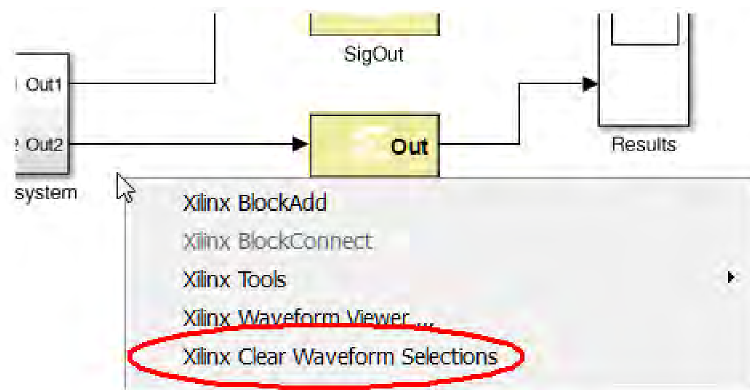
Clearing the Waveform Viewer Display

To clear the waveform display, deleting all the waveforms currently displayed in the Waveform Viewer:

1. Right-click in the System Generator model.
2. Select **Xilinx Clear Waveform Selections** in the right-click menu.

All of the signals currently displayed in the Waveform Viewer are deleted from the Waveform Viewer display, and the Waveform Viewer closes. The deleted waveforms are no longer monitored and the `wavedata` directory (which contains data describing the current display in the Waveform Viewer) is removed from the directory containing your Simulink model.

Figure 182: **Xilinx Clear Waveform Selections**



To open the Waveform Viewer again, right-click in your model and select **Xilinx Waveform Viewer** in the right-click menu. The Waveform Viewer opens displaying waveforms for the clock signals in your design, and no other waveforms.

Customizing the Display and Analyzing Waveforms

The Waveform Viewer has many tools to customize how your waveforms are displayed and to analyze the waveforms. For information on using the Waveform Viewer to develop and troubleshoot your design, see this link in the *Vivado Design Suite User Guide: Logic Simulation* (UG900).

Tips for Working in the Waveform Viewer

The following tips will help you with your waveform analysis using the System Generator model and the Waveform Viewer:

- Keep the Waveform Viewer open during a System Generator session. Do not close the Waveform Viewer between each simulation.
- If you select a group of signals in the Waveform Viewer, all of the signals in the group will be cross-probed from the Waveform Viewer to the System Generator model.
- To add multiple signals in your System Generator model to the Waveform Viewer display, you can press and hold the left mouse button and drag the mouse to draw a box around the signals, selecting them. Then right-click one of the selected signals and select **Xilinx Add to Viewer** in the right-click menu. The selected signals will be added to the Waveform Viewer display.
- When naming an output signal for a block in your System Generator model, avoid using the reserved characters shown in the table below. These are reserved characters in VHDL or Verilog. If your model does contain a signal with a reserved character, its name will be changed in the Waveform Viewer display according to the following mapping table.

Table 6: **Reserved Characters**

Reserved Character	Mapped To
(	#1
)	#2
[	#3
]	#4
.	#5
,	#6
:	#7
\	#8

Closing the Waveform Viewer

To close the Waveform Viewer, select **File→Exit**. If you have not yet saved the waveform data, you will be prompted to save the data before the Waveform Viewer closes.

How to Migrate WaveScope Signals Names from a Deprecated WaveScope Block

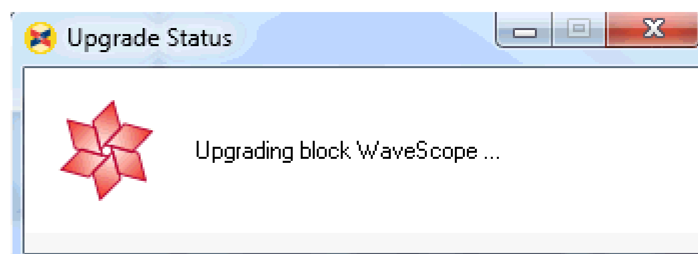
If your design includes a deprecated WaveScope block, you can migrate the existing monitor signal names from the deprecated WaveScope block to the Upgraded block as follows:

1. Right-click the WaveScope block to bring up the context menu.
2. Select **Xilinx Tools→Upgrade block**.

Figure 183: **Upgrade Block**

3. The Upgrade is performed.

Figure 184: **Upgrade Status**



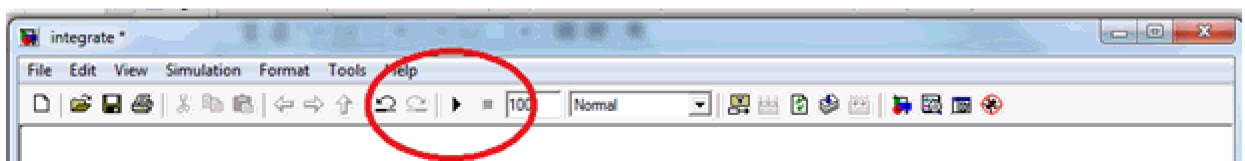
4. After the Upgrade operation is performed, the deprecated WaveScope block is removed from the model, and a summary is written in the MATLAB® console, as shown below:

Figure 185: **MATLAB Console**

```
Migrated Wavescope block signals:
    Wave/Out1
    Noise/Out1
    Mult/Out1

The following Wavescope blocks are removed:
    integrate_scope/WaveScope

A summary of Sysgen Model Upgrade has been written to ModelUpgrade_log_3_22_14_25.lo
>>
```

5. Finally, click the Simulation button to simulate the design.

Figure 186: **Simulation Button**



After the simulation finishes, the signal names from the deprecated WaveScope block are displayed in the Waveform Viewer.

XILINX

Appendix B

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

Solution Centers

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator (DocNav) provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

- From the Vivado® IDE, select **Help → Documentation and Tutorials**.
- On Windows, select **Start → All Programs → Xilinx Design Tools → DocNav**.
- At the Linux command prompt, enter `docnav`.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In DocNav, click the **Design Hubs View** tab.
- On the Xilinx website, see the Design Hubs page.

Note: For more information on DocNav, see the Documentation Navigator page on the Xilinx website.

References

- *Vivado Design Suite Reference Guide: Model-Based DSP Design Using System Generator* (UG958)
- *Vivado Design Suite Tutorial: Model-Based DSP Design Using System Generator* (UG948)
- *Vivado Design Suite User Guide: Using the Vivado IDE* (UG893)
- *Vivado Design Suite User Guide: Design Flows Overview* (UG892)
- *ISE to Vivado Design Suite Migration Guide* (UG911)
- *Vivado Design Suite User Guide: Designing with IP* (UG896)
- *Vivado Design Suite User Guide: Using Constraints* (UG903)
- *Vivado Design Suite User Guide: Using Tcl Scripting* (UG894)
- *Vivado Design Suite Tutorial: Design Flows Overview* (UG888)
- *Vivado Design Suite User Guide: System-Level Design Entry* (UG895)
- *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing* (UG973)
- *UltraFast Design Methodology Guide for Xilinx FPGAs and SoCs* (UG949)

Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at https://

www.xilinx.com/legal.htm#tos; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos.

AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

Copyright

© Copyright 2016-2020 Xilinx, Inc. Xilinx, the Xilinx logo, Alveo, Artix, Kintex, Spartan, Versal, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. AMBA, AMBA Designer, Arm, ARM1176JZ-S, CoreSight, Cortex, PrimeCell, Mali, and MPCore are trademarks of Arm Limited in the EU and other countries. All other trademarks are the property of their respective owners.