

Vivado Design Suite User Guide

Model-Based DSP Design using System Generator

UG897 (v2013.3) October 2, 2013



Notice of Disclaimer

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 the Limited Warranties which can be viewed at <http://www.xilinx.com/warranty.htm>; 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 Critical Applications: <http://www.xilinx.com/warranty.htm#critapps>.

© Copyright 2012-2013 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
10/16/12	2012.3	Initial Xilinx release.
03/20/13	2013.1	Updates added to reflect GUI changes in the product. New chapter added on migrating System Generator designs from IDS to Vivado IDE environment.
06/19/13	2013.2	Editorial improvements have been made to the text.
10/02/13	2013.3	Added a new section at the end of Chapter 4 titled "AXI4-Lite Interface Generation". New compilation types have been added to Chapter 7 and a new Chapter 8 titled "Creating Custom Compilation Targets" has been added. Black Box examples have been moved to the document titled Vivado Design Suite Tutorial: Model-Based DSP Design using System Generator (ug948).

Chapter 1: Introduction

The Xilinx DSP Block Set	8
FIR Filter Generation	9
Support for MATLAB	9
Hardware Co-Simulation	11
System Integration Platform	12

Chapter 2: Installation

Downloading	13
Hardware Co-Simulation Support	13
UNC Paths Not Supported	13
Using the Xilinx Installer	14
Choosing MATLAB for System Generator	14
Post Installation Tasks	15
Post-Installation Tasks on Linux	15
Hardware Co-Simulation Installation	15
Compiling Xilinx HDL Libraries	16
Example Designs Associated with this User Guide	16
Managing the System Generator Cache	16

Chapter 3: Migrating Designs to the Vivado IDE

Introduction	17
Upgrade Methodology	17
Preparing the Model for Migration using the ISE Environment	17
Completing the Migration Flow in the Vivado IDE	20

Chapter 4: Hardware Design using System Generator

Design Flows using System Generator	23
Algorithm Exploration	23
Implementing Part of a Larger Design	23
Implementing a Complete Design	24
Note to the DSP Engineer	24
Note to the Hardware Engineer	24
System-Level Modeling in System Generator	25
System Generator Blocksets	26
Signal Types	28
Floating-Point Data Type	29
AXI Signal Groups	33
Bit-True and Cycle-True Modeling	33
Timing and Clocking	34
Synchronization Mechanisms	37

Block Masks and Parameter Passing	38
Automatic Code Generation	41
Compiling and Simulating Using the System Generator Token	41
Caching	45
Compilation Results	45
HDL Testbench	47
Multiple Clock Island Netlisting	48
Compiling MATLAB into an FPGA	50
Simple Selector	51
Simple Arithmetic Operations	51
Complex Multiplier with Latency	54
Shift Operations	54
Passing Parameters into the MCode Block	55
Optional Input Ports	58
Finite State Machines	61
Parameterizable Accumulator	62
FIR Example and System Verification	65
RPN Calculator	68
Example of disp Function	69
Importing a System Generator Design into a Bigger System	71
HDL Netlist Compilation	71
Integration Design Rules	72
Configurable Subsystems and System Generator	72
Defining a Configurable Subsystem	72
Using a Configurable Subsystem	74
Deleting a Block from a Configurable Subsystem	75
Adding a Block to a Configurable Subsystem	76
Generating Hardware from Configurable Subsystems	77
Notes for Higher Performance FPGA Design	79
Review the Hardware Notes Included with Each Block Dialog Box	79
Register the Inputs and Outputs of Your Design	80
Insert Pipeline Registers	80
Use Saturation Arithmetic and Rounding Only When Necessary	82
Set the Data Rate Option on All Gateway Blocks	82
Other Things to Try	83
Using FDATool in Digital Filter Applications	83
Design Overview	85
Open and Generate the Coefficients for this FIR Filter	85
Parameterize the MAC-Based FIR Block	86
Generate and Assign Coefficients for the FIR Filter	87
Browse Through and Understand the Xilinx Filter Block	89
Run the Simulation	90
AXI Interface	92
Introduction	92
AXI4-Stream Support in System Generator	93
AXI-Stream Blocks in System Generator	95
AXI4-Lite Interface Generation	97
Introduction	97
AXI4-Lite Interface Synthesis in System Generator	97
Configuring the Design for an AXI4-Lite Interface	98
Packaging the Design for Use in Vivado IP Integrator	99

Description of the Generated Results	100
Mapping to a Single AXI4-Lite Interface	101
Address Generation	101
Features of the Vivado IDE Example Project	102
Software Drivers	104

Chapter 5: Using Hardware Co-Simulation

Introduction	105
M-Code Access to Hardware Co-Simulation	105
Installing Your Hardware Board	105
JTAG-Based Hardware Co-Simulation	105
Compiling a Model for Hardware Co-Simulation	106
Choosing a Compilation Target	106
Invoking the Code Generator	106
Hardware Co-Simulation Blocks	107
Hardware Co-Simulation Clocking	109
Clocking Modes	109
Selecting the Clock Mode	110
Installing a KC705 Board for JTAG Hardware Co-Simulation	110

Chapter 6: Importing HDL Modules

Black Box HDL Requirements and Restrictions	112
Black Box Configuration Wizard	113
Black Box Configuration M-Function	115
HDL Co-Simulation	130
Introduction	130
Configuring the HDL Simulator	130
Co-Simulating Multiple Black Boxes	132

Chapter 7: System Generator Compilation Types

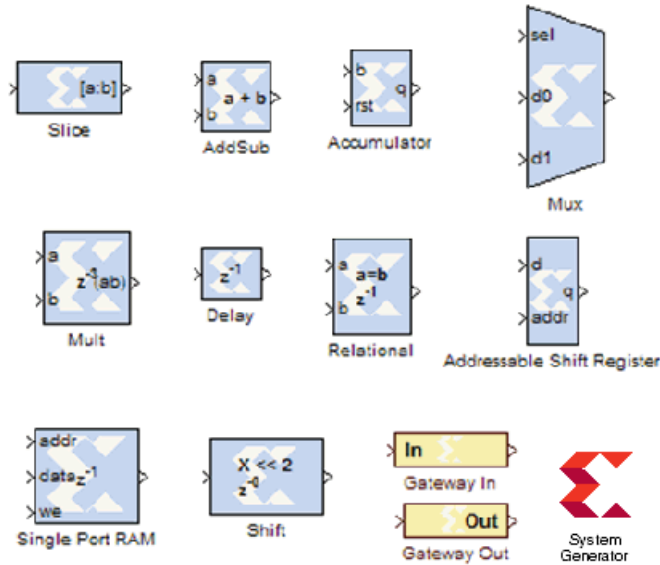
HDL Netlist Compilation	134
Hardware Co-Simulation Compilation	134
IP Catalog Compilation	134
The IP Catalog Flow	135
Including a Testbench with the IP Module	136
Adding an Interface Document to the IP Module	137
Adding the Generated IP to the Vivado IP Catalog	137
Synthesized Checkpoint Compilation	139
Creating Your Own Custom Compilation Target	139

Chapter 8: Creating Custom Compilation Targets

xilinx_compilation Base Class	140
Creating a New Compilation Target	141
Running the Helper Function	141
Creating a New Board Target	142
Modifying a Compilation Target	143
Adding an Existing Compilation Target	143
Saving a Custom Compilation Target	143
Removing a Custom Compilation Target	143
Base Class Properties and APIs	144
System Generator Token-Related Properties and APIs	144
Vivado Project-Related Properties	145
Vivado IDE Project Generation-Related Functions	145
Design Info	147
Examples of Creating Custom Compilation Targets	148
Example 1: Creating an Implementation Target	148
Example 2: Creating a Zedboard Target	151
Example 3: Creating a Bitstream Target	152
Xilinx Resources	155
Solution Centers	155
References	155

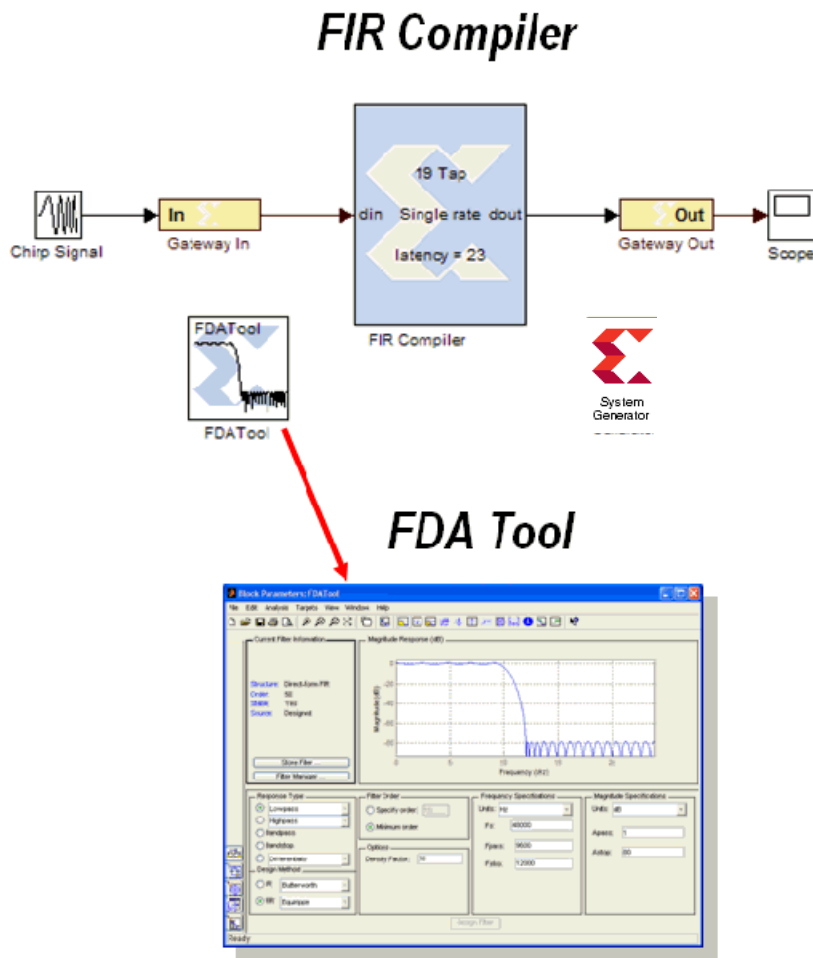
The Xilinx DSP Block Set

Over 90 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.



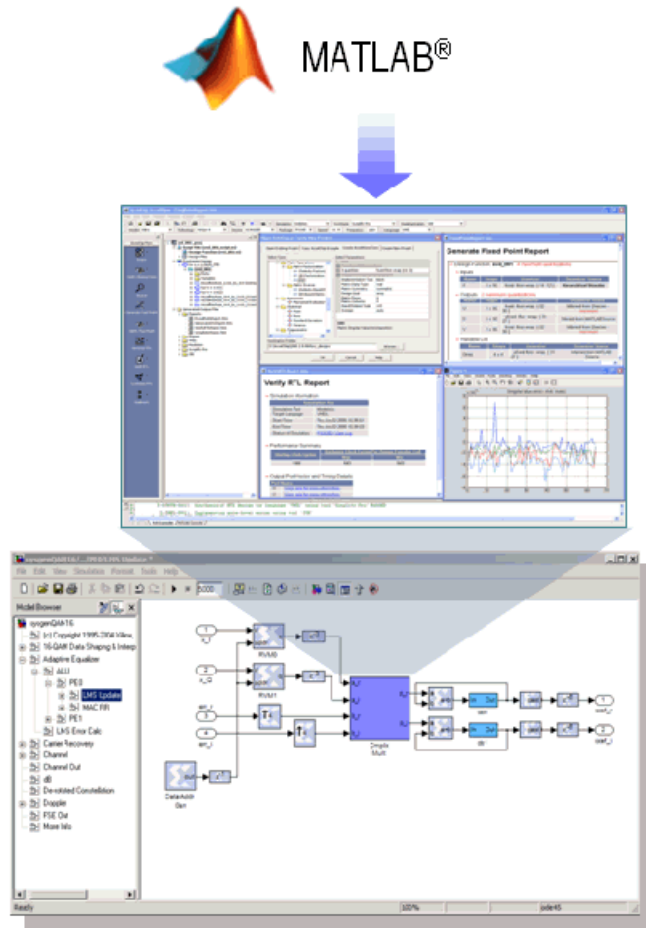
FIR Filter Generation

System Generator includes a FIR Compiler block that targets the dedicated DSP48E1 hardware resources in the 7 series devices to create highly optimized implementations. Configuration options allow generation of direct, polyphase decimation, polyphase interpolation and oversampled implementations. Standard MATLAB functions such as fir2 or the MathWorks Fdatool can be used to create coefficients for the Xilinx FIR Compiler.



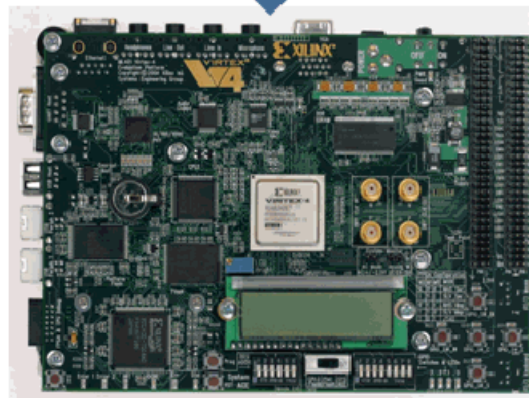
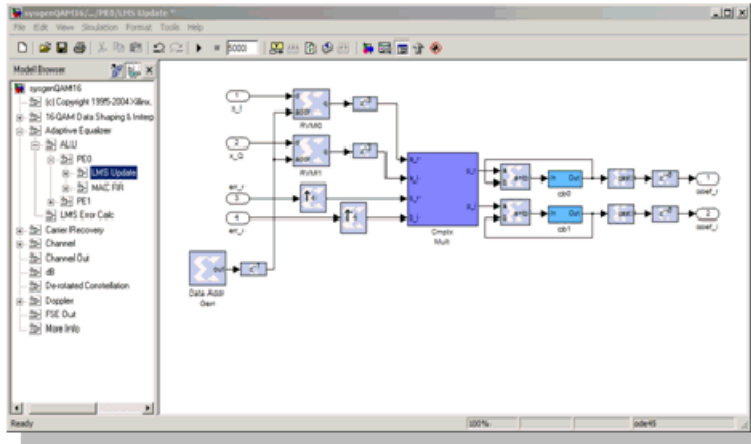
Support for MATLAB

Included in System Generator is an MCode block that allows the use of non-algorithmic MATLAB for the modeling and implementation of simple control operations.



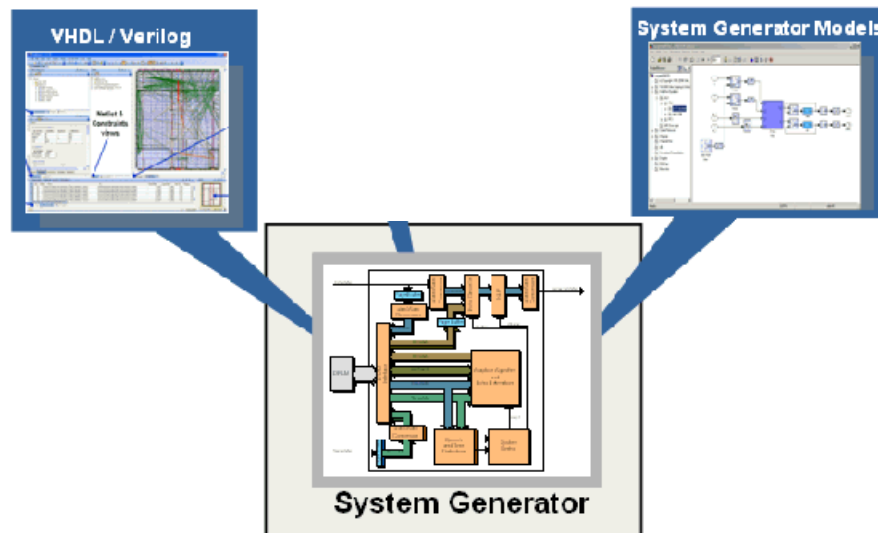
Hardware Co-Simulation

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.



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. System Generator also supports the inclusion of a MicroBlaze® embedded processor running C/C++ programs.



Installation

Downloading

System Generator is part of the Vivado™ Design Suite and may be download from the Xilinx web page. You may purchase, register, and download the System Generator software from the site at:

<http://www.xilinx.com/tools/sysgen.htm>

Note: In special circumstances, System Generator can be delivered on a CD. Please contact your Xilinx distributor if your circumstances prohibit you from downloading the software via the web.

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 the Kintex™-7 KC705 Development Board, the Virtex™-7 VC707 Development Board, and the Zynq-7000 series ZC702 and ZC706 Development Board. System Generator board support packages can be downloaded from the following URL:

http://www.xilinx.com/products/boards_kits/index.htm

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.

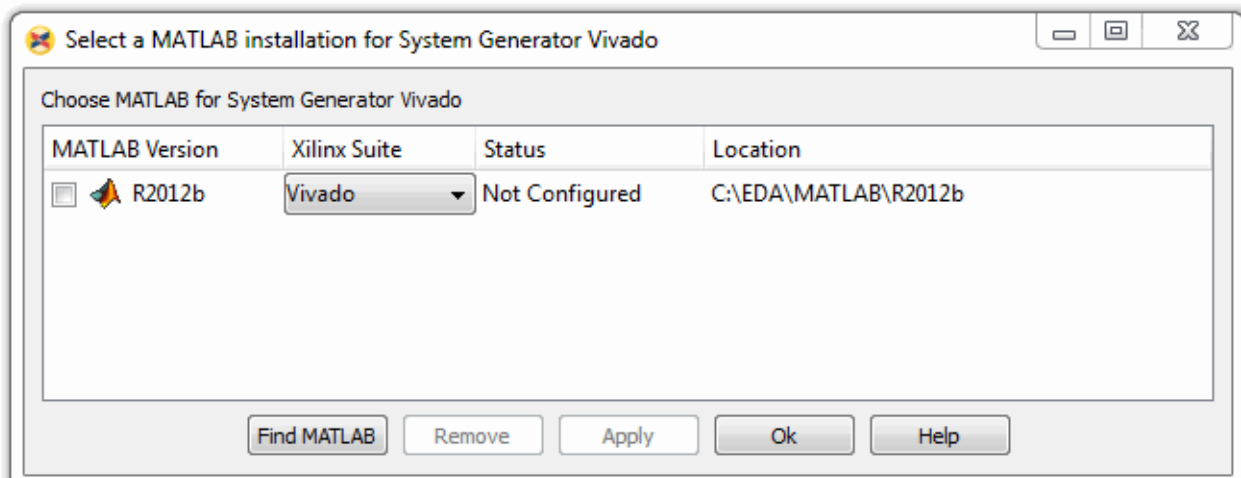
Using the Xilinx Installer

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



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 don't see a version of MATLAB listed, click **Find MATLAB** to browse for a valid version.

If you wish to change the MATLAB configuration, select the following Windows menu item:

Start > All Programs > Xilinx Design Tools > Vivado 2013.1 > System Generator > System Generator MATLAB Configurator.

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

Linux Installations

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

Note: This will invoke MATLAB and dynamically add System Generator to that MATLAB session. At the top of the MATLAB Command Window, you should see the "Installed System Generator dynamically" messages. You are now ready to run System Generator.

The following is an expected message under certain conditions. If System Generator is already installed when this script runs, you will see the following message:

```
System Generator currently found installed into matlab default path.
```

Hardware Co-Simulation Installation

This topic provides links to hardware and software installation procedures for hardware co-simulation. If you do not plan to use hardware co-simulation, you may skip this topic.

Note: If installation instructions for your particular platform are not provided here, please refer to the installation instructions that come with your Platform Kit. For instructions on how to install a Xilinx USB Cable and cable driver software on a Windows or Linux Operating System, refer to the Xilinx document titled: [USB Cable Installation Guide](#)

Compiling Xilinx HDL Libraries

If you intend to simulate System Generator designs using ModelSim, you must compile your IP (cores) libraries. This topic describes the procedure.

ModelSim SE

The Xilinx tool that compiles libraries for use in ModelSim SE is named **compplib**. The following command can, for example, be used to compile all the VHDL and Verilog libraries with ModelSim SE:

```
compplib -s mti_se -f all -l all
```

Complete instructions for running compplib can be found in the Xilinx Software Manual titled "Command Line Tool User Guide".

Example Designs Associated with this User Guide

Example Designs that are used for illustration in this document are contained in a ZIP file that may be downloaded from the Web. This ZIP file is named **ug897-example-files.zip** and is physically located near the place where the User Guide is located. This document assumes that you have downloaded the example designs to the location **C:/ug897-example-files**.

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.

Migrating Designs to the Vivado IDE

Introduction

System Generator for DSP has a new Upgrade Model feature that assist you in migrating designs previously created in the IDS environment to designs that are compatible with the Vivado Integrated Design Environment (IDE).

Requirements for migration are as follows:

- The design must target 7 series or Zynq devices
 - Before migration, the design blocks must be upgraded to the latest version found in System Generator.
 - Design blocks that are incompatible with Vivado IDE must be removed or replaced.
-

Upgrade Methodology

The recommended migration methodology involves (1) preparing the model for migration using the ISE Environment and (2) completing the migration flow using the Vivado Integrated Design Environment (IDE).

Preparing the Model for Migration using the ISE Environment

The model preparation involves (1) upgrading all blocks to the latest found in System Generator release and (2) removing blocks that are incompatible with the Vivado environment.

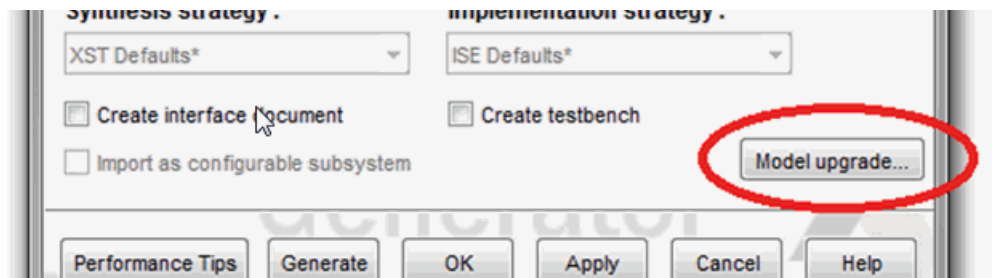
Upgrading Blocks to the Latest Found in System Generator

1. Open the System Generator model in the latest System Generator release.

The latest blocks with multiple versions are listed in the table below:

Block Name	Latest Version in ISE
CIC Compiler	CIC Compiler 3.0
CORDIC	CORDIC5.0
Complex Multiplier	Complex Multiplier 5.0
Convolution Encoder	Convolution Encoder 8.0
Divider Generator	Divider Generator 4.0
DDS Compiler	DDS Compiler 5.0
DSP48 Macro	DSP48 Macro 2.1
FIR Compiler	FIR Compiler 6.3
Fast Fourier Transform	Fast Fourier Transform 8.0
Interleaver/De-Interleaver	Interleaver/De-Interleaver 7.1
Reed-Soloman Decoder	Reed-Soloman Decoder 8.0
Reed-Soloman Encoder	Reed-Soloman Encoder 8.0
Viterbi Decoder0	Viterbi Decoder 8.0

2. Double click on the System Generator token and then click the **Model upgrade** button as shown below:



3. Observe the information in the generated Status Report, as shown in the following figure:

Upgrade Status Report

Model upgrade flow assists user in migration of old versions of Sysgen blocks to latest available versions in the System Generator design [model_upgrade](#).

Upgradation of blocks to latest version is recommended.

[Upgrade](#) the model.

model_upgrade

This Model has 2 blocks which are superceded. Necessary action is required to maintain the functionality of the design in future releases.

Upgrade Support - Specifies the update support of the block

Replace Support - Specifies the block replacement and stitching support in the Model

Block name	Block version Used	Block version Available	Upgrade support	Replace support	Perform Upgrade	Action
.../model_upgrade/Complex Multiplier 3.1	3.1	5.0	Yes	No	Upgrade	Required
.../Interleaver//De-interleaver 7.0	7.0	7.1	Yes	Yes	Upgrade	Required

- Two blocks in this the model are upgradable.
- The Interleaver/De-interleaver 7.0 block has full Replace support. When you click Upgrade in the **Perform Upgrade** column, the single block is upgraded.
- In this case, the Complex Multiplier 3.1 block does not have full Replace support because moving from the non-AXI 3.1 block to the AXI 5.0 block requires manual intervention. When you click Upgrade in the column, a sub-system work-space is create where you can manually re-connect the input/out signals to the new AXI ports.

Removing Blocks that are Incompatible with the Vivado Environment

Blocks that are incompatible with the Vivado IDE should be removed from the model. Incompatible blocks are listed below:

Block Incompatible with Vivado IDE	Action to Take
ChipScope	Continue using System Generator 14.4 or directly use the Vivado IDE for debug
Configurable Subsystem Manager	
Multiple Subsystem Generator	
Resource Estimator	Remove this block until a replacement capability is introduced in a future release
EDK Processor	Continue using System Generator 14.4 until this capability is introduced in a future release
From FIFO, To FIFO, From Register, To Register, Shared Memory, Shared Memory Read, Shared Memory Write	Continue using System Generator 14.4 until a replacement capability is introduced in a future release
PicoBlaze Instruction Display PicoBlaze Microcontroller	Continue using System Generator 14.4
VDMA Interface 5.3	Continue using System Generator 14.4 until a replacement capability is introduced in a future release
WaveScope	Use Waveform Viewer

Completing the Migration Flow in the Vivado IDE

1. Verify that the model contains only the latest 14.4/14.5 blocks and that blocks incompatible with the Vivado environment have been removed.
2. Open the prepared System Generator design in the Vivado IDE.
3. Right-click on a blank space in the model sheet and select **Tools > Upgrade model** from the pop-up menu.

The following table shows block versions that are upgraded to the latest System Generator version found in the Vivado IDE:

Latest Block in Sysgen-ISE	Latest Block in Sysgen-Vivado IDE
CIC Compiler 3.0	CIC Compiler 4.0
CORDIC5.0	CORDIC 6.0
Complex Multiplier 5.0	Complex Multiplier 6.0
Convolution Encoder 8.0	Convolution Encoder 9.0
DDS Compiler 5.0	DDS Compiler 6.0
Divider Generator 4.0	Divider Generator 5.0
DSP48 Macro 2.1	DSP48 Macro 3.0
FIR Compiler 6.3	FIR Compiler 7.1
Fast Fourier Transform 8.0	Fast Fourier Transform 9.0
Interleaver/De-Interleaver 7.1	Interleaver/De-Interleaver 8.0
Reed-Soloman Decoder 8.0	Reed-Soloman Decoder 9.0
Reed-Soloman Encoder 8.0	Reed-Soloman Encoder 9.0
Viterbi Decoder 8.0	Viterbi Decoder 9.0

4. Select **File > Save** from the pull-down menu.
5. Re-simulate the design in MATLAB to verify that it is functionally correct.
6. **Close** the design

The design migration process is now complete.

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.

AXI Interface	Provides an introduction to AMBA AXI4 and draws attention to AMBA AXI4 details with respect to System Generator.
AXI4-Lite 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.

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;
- A HDL testbench.. The testbench 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 make 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 the 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 [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 testbench, including test vectors. This aspect is described in the topic [HDL Testbench](#).

Finally, the hardware co-simulation interfaces described in the topic [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 design 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.

[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

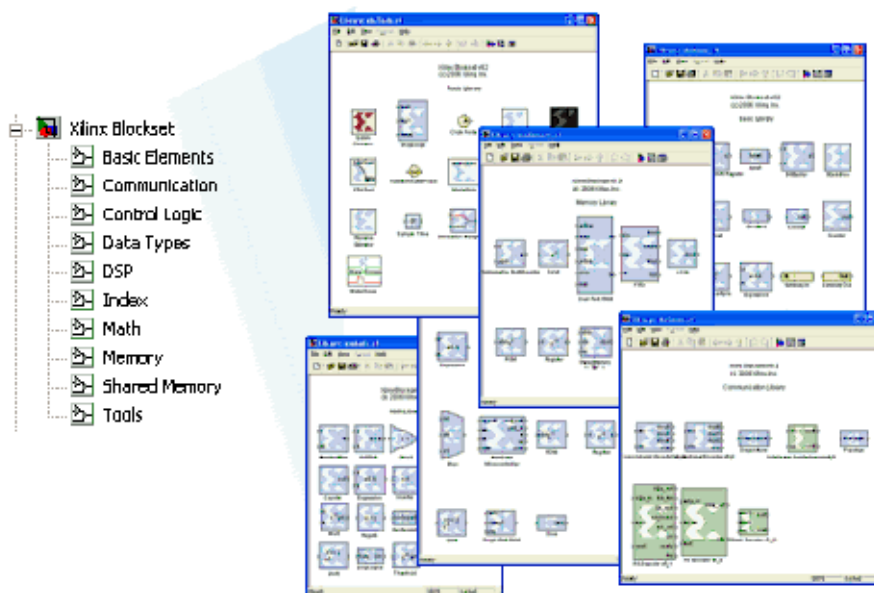
Describes mechanisms that can be used to synchronize data flow across the data path elements in a high-level System Generator design, and describes how control path functions can be implemented.

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, logic, 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.

Note: It is important that you don't name your design the same as a Xilinx block. For example, if you name your design **black box.mdl**, it may cause System Generator to issue an error message.

Library	Description
AXI4	Blocks with interfaces that conform to the AXI™4 specification
Basic Elements	ElementsStandard 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
DSP	Digital signal processing (DSP) blocks
Data Types	Blocks that convert data types (includes gateways)
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

Note: More information concerning blocks can be found in the topic [Xilinx Blockset](#).

Xilinx Reference Blockset

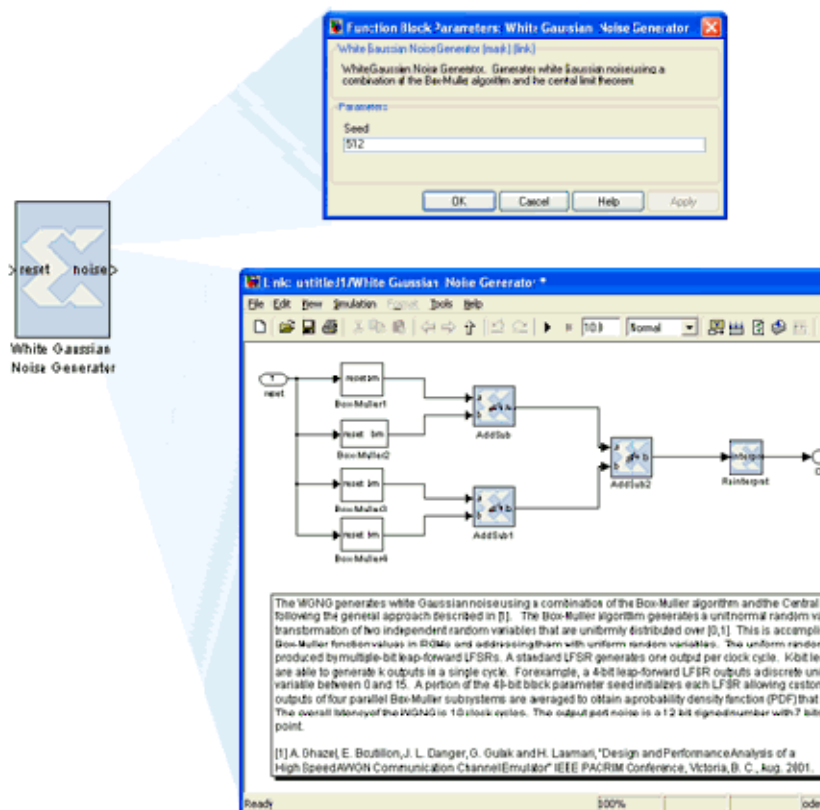
The Xilinx Reference Blockset contains composite System Generator blocks that implement a wide range of functions. Blocks in this blockset are organized by function into different libraries. The libraries are described below.

Library	Description
Communication	Blocks commonly used in digital communications systems
Control Logic	LogicBlocks used for control circuitry and state machines
DSP	Digital signal processing (DSP) blocks

Library	Description
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. Individual documentation for each block is also provided in the topic



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 are able to 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 **Format > 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 (**Format > Sample Time Colors** in the Simulink pulldown menus). This is often useful in understanding multirate designs.

Floating-Point Data Type

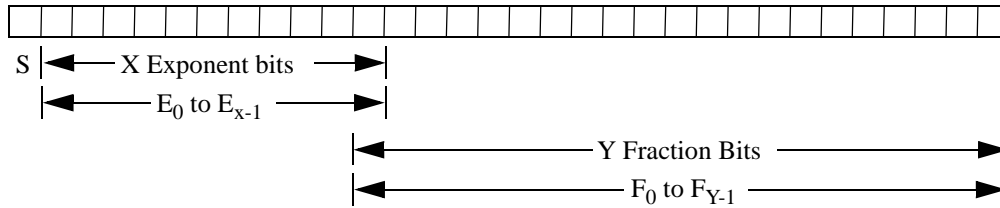
System Generator blocks found in the Floating-Point library support the floating-point data type.

System Generator uses the Floating-Point Operator v6.0 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).



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:

$$\text{Exponent_bias} = 2^{(X - 1)} - 1, \text{ where } X \text{ is the exponent bit width.}$$

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:

$$\text{Normalized Floating-Point Value} = (-1)^S \times F_0.F_1F_2 \dots F_{Y-2}F_{Y-1} \times (2)^E$$

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:

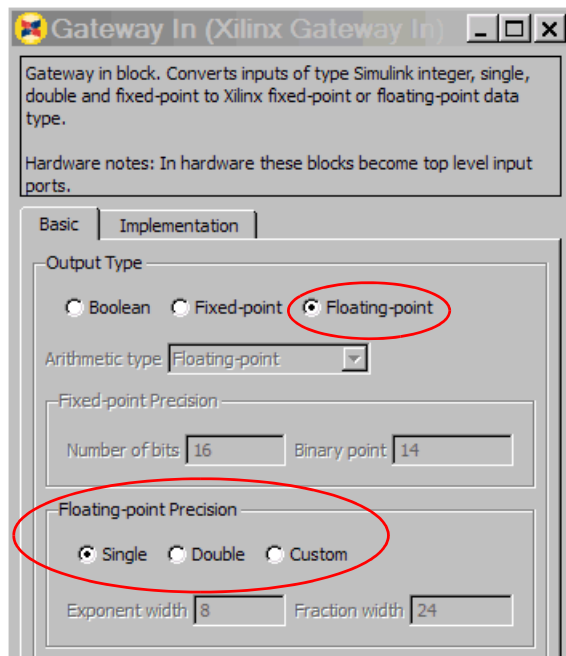
$$\text{FP_Value} = (-1)^S \times 1.F_1F_2 \dots F_{Y-2}F_{Y-1} \times (2)^{(E_{\text{actual}})}$$

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.

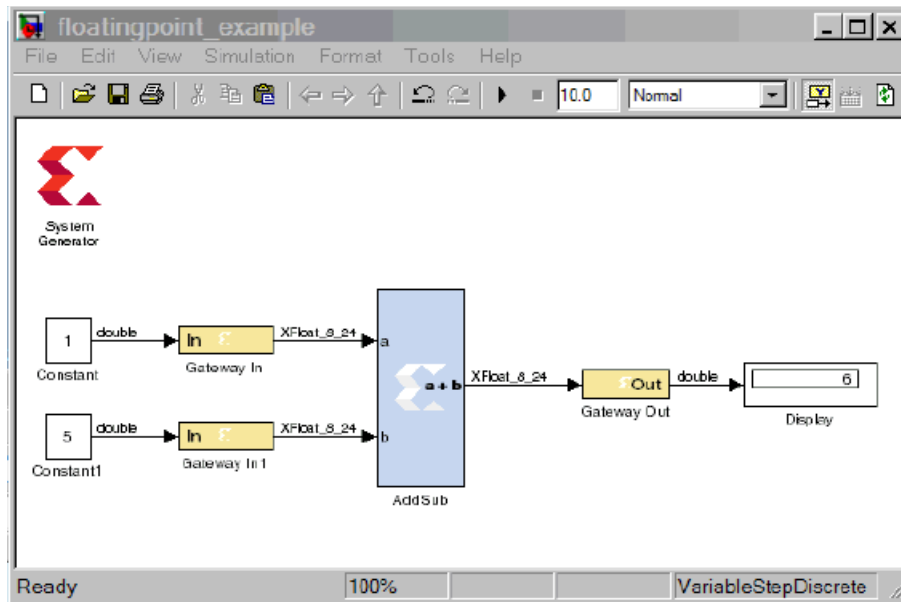
For example, if Exponent width of 9 and Fraction width of 31 is specified then the floating-point data value will be stored in total 40 bits where the MSB bit will be used for sign representation, the following 9 bits will be used to store biased exponent value and the 30 LSB bits will be used to store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



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 **Format > Port/Signal Displays > Port Data Types**.



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. Since 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 User Defined 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 AXI™ 4 specification. Blocks with AXI interfaces are drawn such that ports relating to a particular AXI interface are grouped and colored in similarly. This makes it easier to identify data and control signals pertaining to the same interface. Grouping similar AXI 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 AXI can be found in the section entitled [AXI Interface](#). For more detailed information on the AMBA AXI4 specification, please refer to the Xilinx AMBA AXI4 documents found at the following location: <http://www.xilinx.com/ipcenter/axi4>

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.

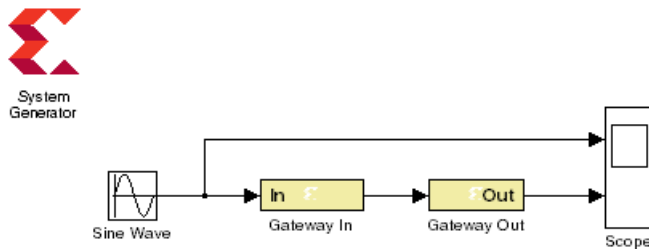
Timing and Clocking

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.

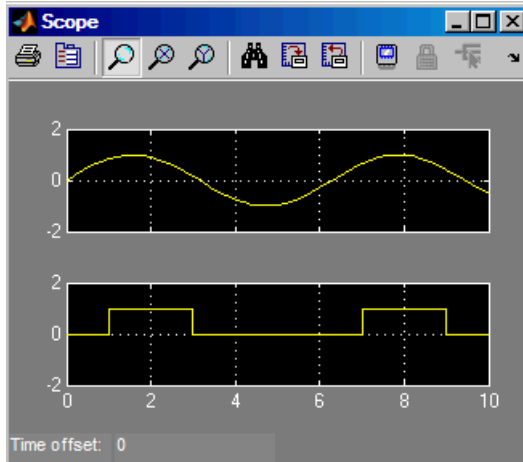
Note: For an in-depth explanation of Simulink discrete time systems and sample times, consult the Using Simulink reference manual from the MathWorks, Inc.

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).



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.



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.

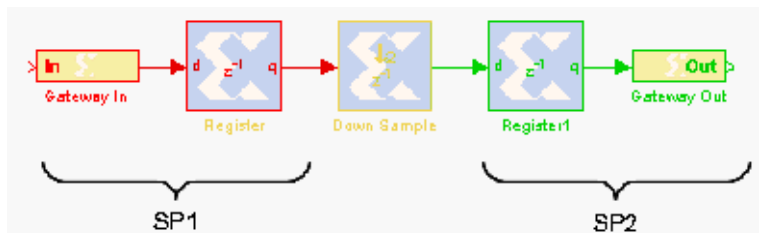
Rate-Changing Blocks

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.



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.



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.

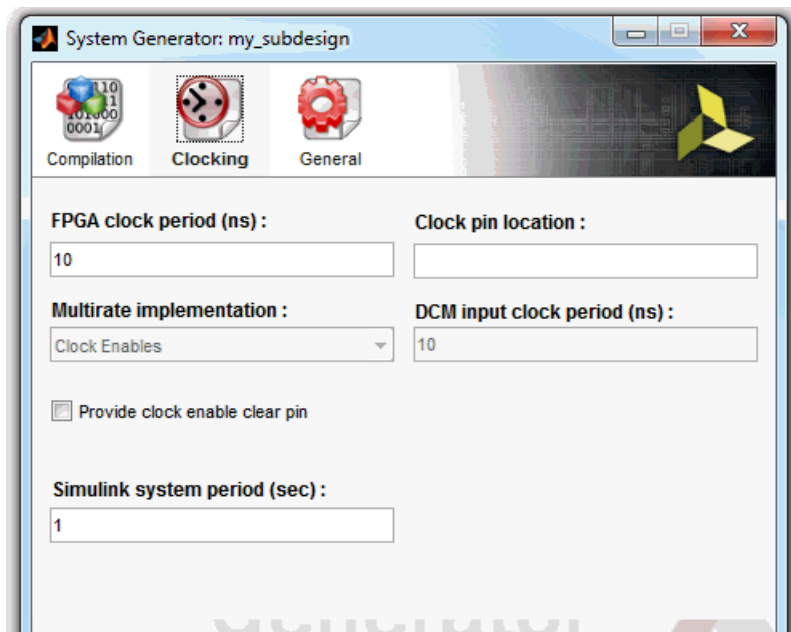
Although blocks that are oversampled do not cause an explicit sample rate change in Simulink, System Generator considers the internal block rate along with all other sample rates when generating clocking logic for the hardware implementation. This means that you must consider the internal processing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

As shown in the figure below, when you use the System Generator token to compile a design into hardware, there is one clocking option for Multirate implementation: (1) Clock Enables (the default).



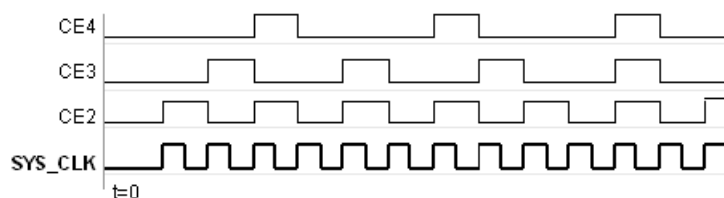
Clock Enables

When System Generator compiles a model into hardware with the Clock Enable option selected, 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 k 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:



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, a FIFO) that can be used for synchronization. Several blocks provide input (respectively, output) ports that specify when an input (resp., output) sample is valid. Such ports can be chained, affording a primitive form of flow control. Blocks with such ports include the FFT, FIR, and Viterbi.

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.

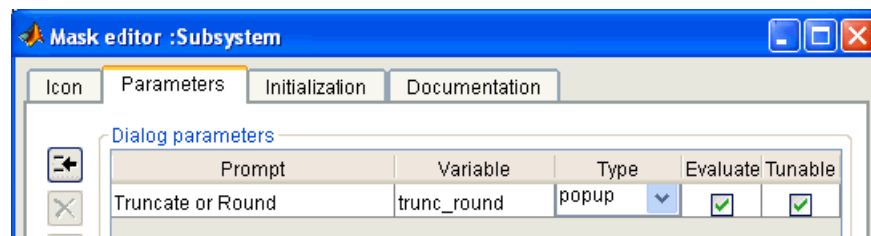
Block Masks

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.

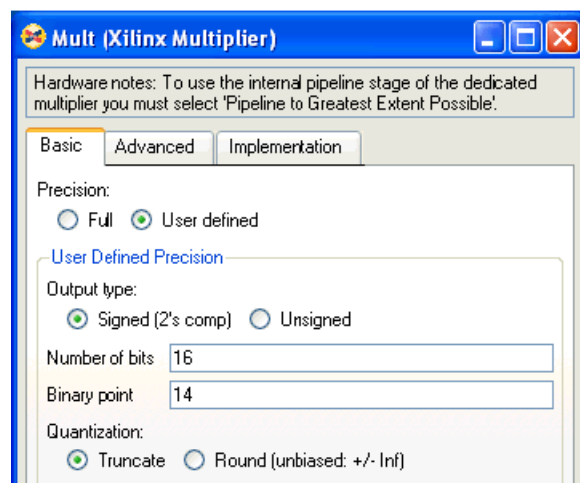
Note: It is possible for a mask to access global variables and variables in the base workspace. To access a base workspace variable, use the MATLAB evalin function. For more information on the MATLAB and Simulink scoping rules, refer to the manuals titled *Using MATLAB and Using Simulink* from *The MathWorks,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

Parameter Passing

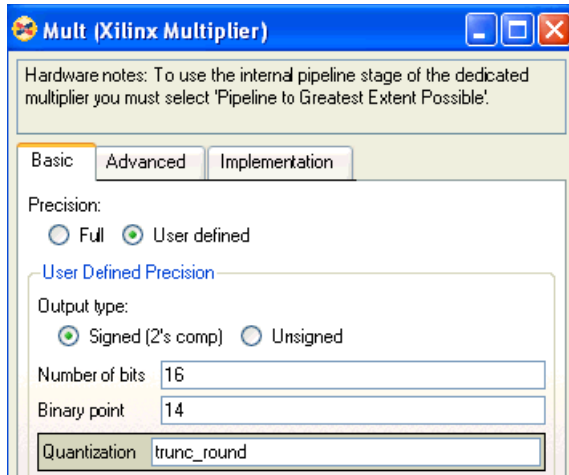
It is often desirable to pass variables to blocks inside a masked subsystem. Doing so allows the block's configuration to be determined by parameters on the enclosing subsystem. This technique can be applied to parameters on blocks in the Xilinx blockset whose values are set using a listbox, radio button, 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.



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.



In order to use a parameter rather than the radio button selection, right click on 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.



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 testbench) 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.

HDL Testbench

Describes the VHDL testbench that System Generator can produce.

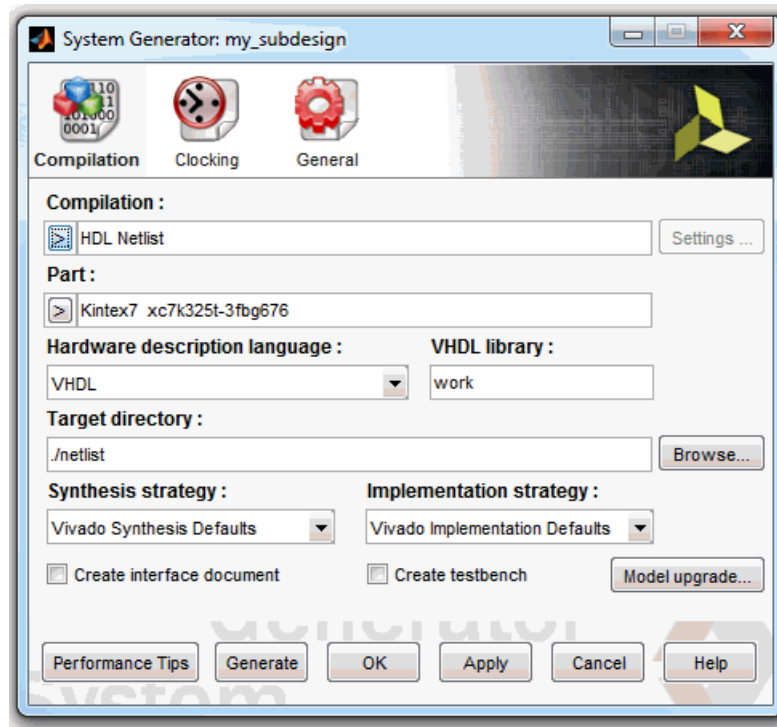
Compiling and Simulating Using the System Generator Token

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.

Before a System Generator design can be simulated or translated into hardware, the design must include a System Generator token. When creating a new design, it is a good idea to add a System Generator token immediately. 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 token's dialog box is shown below:



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

- **HDL Netlist**
- **Various varieties of hardware co-simulation**
- **IP Catalog**- packages the design as an IP core that can be added to the Vivado IP catalog for use in another design.
- **Synthesized Checkpoint** - Creates a design checkpoint file (synth_1.dcp) that can then be used in any Vivado IDE project.

Control	Description
Part	Defines the FPGA part to be used.
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).
Hardware description language	Specifies the language to be used for HDL netlist of the design. The possibilities are VHDL and Verilog.
Create testbench	This instructs System Generator to create an HDL testbench. Simulating the testbench 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 testbench 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.
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 a "documentation" subfolder under the netlist folder.
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. Multicycle paths are constrained to integer multiples of this value.
Clock pin location	Defines the pin location for the hardware clock. This information is passed to the Xilinx implementation tools through a constraints file.
DCM input clock period(ns)	Specify if different than the FPGA clock period(ns) option (system clock). The FPGA clock period (system clock) will then be derived from this hardware-defined input.

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. If a problem is found, System Generator opens a dialog box suggesting an appropriate value. Clicking the button labeled **Update** instructs System Generator to use the suggested value. To see a summary of period conflicts, click the button labeled **View Conflict Summary**. If you allow System Generator to update the period, you must restart the simulation or compilation.

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 the discussion of hierarchical controls below 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 **Control-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
- 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.

Caching

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 the files during subsequent simulation and generation rather than re-running the time consuming tools used to create these files.

To obtain the pathname to the System Generator cache, enter the following command at the MATLAB console prompt:

```
>> xilinx.environment.getcachepath
```

To clear or remove the cache, simply remove the cache directory:

```
>> rmdir(xilinx.environment.getcachepath,'s')
```

For predictable results after removing the cache, you should first close System Generator and then re-open System Generator.

Compilation Results

In 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, NGC and EDIF that implement 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 the System Generator token. If no testbench is requested, then the key files produced by System Generator are the following:

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 sysgen 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 testbench 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. The additional files are the following:

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

Using the System Generator Constraints File

When a design is compiled, 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.

See the descriptions of the lock and the block for more information. 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*.

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.

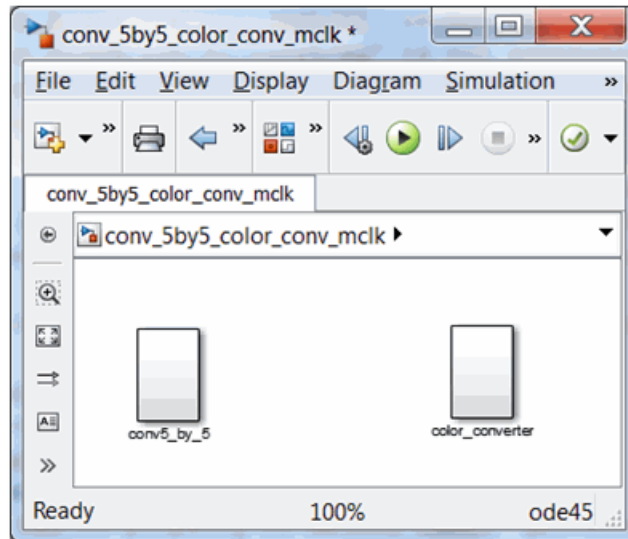
Suppose the design is named <design>, and a System Generator token is placed at the top of the design. Suppose also that in the token the **Compilation** field is set to **HDL Netlist**, and the **Create Testbench** checkbox is selected. When the **Generate** button is clicked, System Generator produces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

1. A file named <design>_tb.vhd/.v that contains a testbench HDL entity;
2. Various .dat files that contain test vectors for use in an HDL testbench simulation.
3. 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)*.

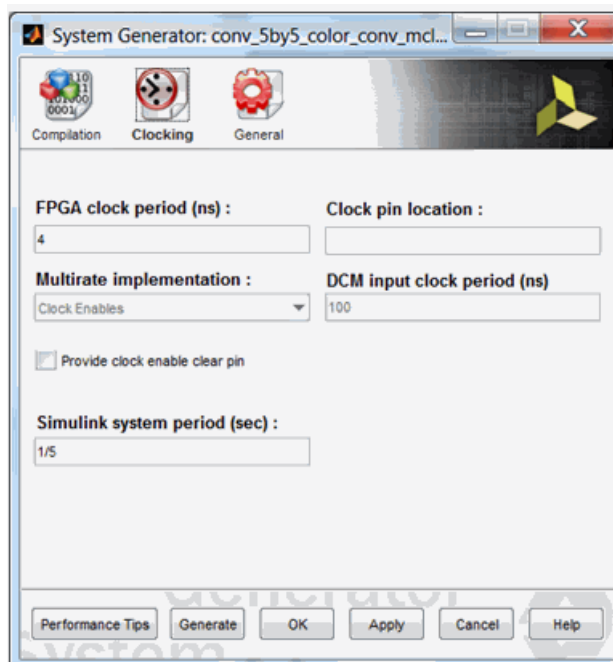
System Generator generates the .dat files by saving the values that pass through gateways. In the HDL simulation, input values from the .dat files are stimuli, and output values are expected results. The testbench is simply a wrapper that feeds the stimuli to the HDL for the design, then compares HDL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

Multiple Clock Island Netlisting

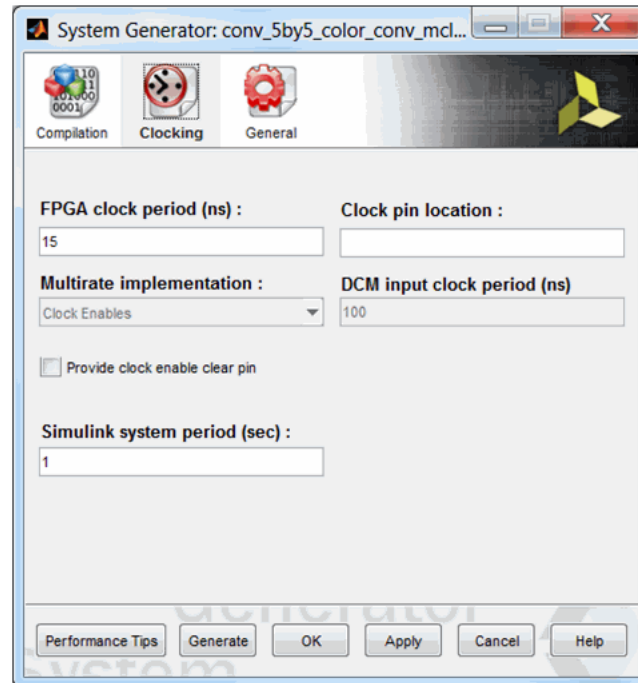
Starting with System Generator for DSP v2013.1, automatic code generation can be extended to a design containing multiple clock islands. To do this, you must partition the design into multiple subsystems at the top level as shown below.



As shown in the design above, the top level contains only subsystems where each subsystem contains a System Generator block the captures the clocking information for that subsystem. Shown below is the System Generator block underneath conv5_by_5. The clock period is set to 4ns.



Shown next is the System Generator block underneath the color_conv. The clock period is set to 15ns.



To generate such a design, the following MATLAB commands must be executed.

```
>>xilinxModelObject = xilinx.model(get_param('conv_5by5_color_conv_mclk'));
```

The above command creates a Xilinx Model Object in MATLAB where the Simulink model name 'conv_5by5_color_conv_mlk' is the name of the Simulink model.

Note: To execute this command you must ensure that the model is already opened.

Next, you have to create a structure that captures all the settings for automatic generation. An example is shown below:

```
>> settings = struct('Family', 'virtex7', ...%Device Family
'Device', 'xc7vx485t', ...%Part name
'Speed', '-1', ...%Speed Grade being used
'Package', 'ffg1157',...%Package Name being used
'SynthesisTool', 'Vivado',...%Synthesis Tool setting
'SynthesisLanguage', 'VHDL',...%Synthesis Language Setting
'SynthesisStrategy', 'Vivado Synthesis Defaults',...%Synthesis Strategy Using defaults
'ImplementationStrategy', 'Vivado Implementation Defaults',...%Implementation Strategy
'TargetDirectory', './netlist',...%Code Generation directory
'Testbench','on');...%The Test bench option
```

Finally, to generate the design, you need to execute the following command:

```
>>xilinxModelObject.generate(settings);
```

This command synchronizes the setting of all All System Generator tokens, except the clocking settings, and invokes the code generator to create one project with RTL files, IP and Constraints.

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 ten 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.

The first two examples are in the `mcode_block_tutorial.mdl` file of the `examples/mcode_block` directory in your installation of the System Generator software. Examples 3 and 4 are in the `mcode_block_tutorial2.mdl` file. Examples 5 and 6 are in the `mcode_block_tutorial3.mdl` file. Examples 7 and 8 are in the `mcode_block_tutorial4.mdl` file. Example 9 is `mcode_block_verify_fir.mdl`. Example 10 is in `mcode_block_rpn_calculator.mdl`.

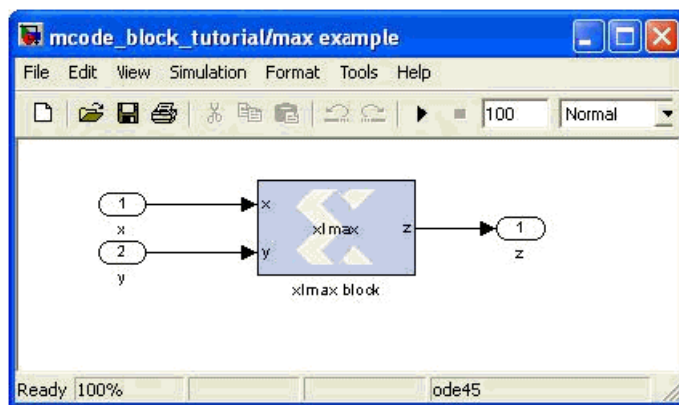
Simple Selector

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.



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()
```

```

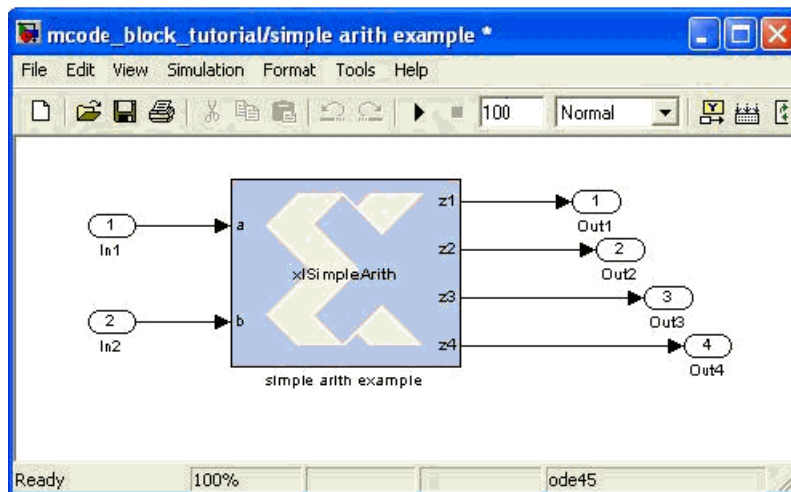
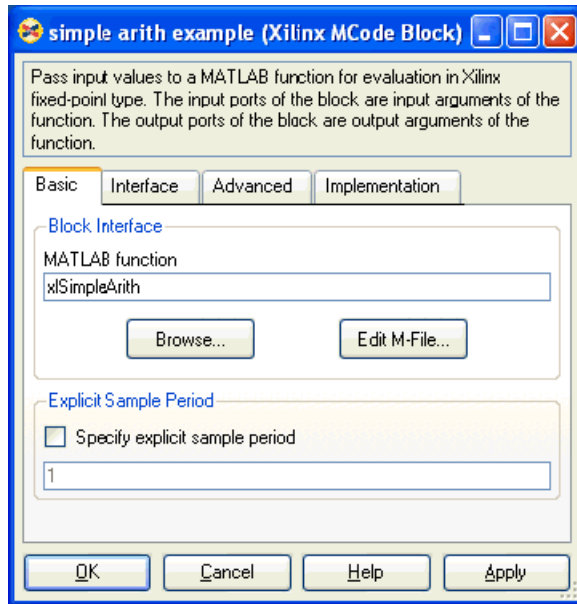
% to create Xilinx fixed-point numbers with appropriate
% container types.%
% You must use a xfix() to specify type, number of bits, and
% binary point position to convert floating point values to
% Xilinx fixed-point constants or variables.
% By default, the xfix call uses xlTruncate
% and xlWrap for 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. Quantization 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`.



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
    
```

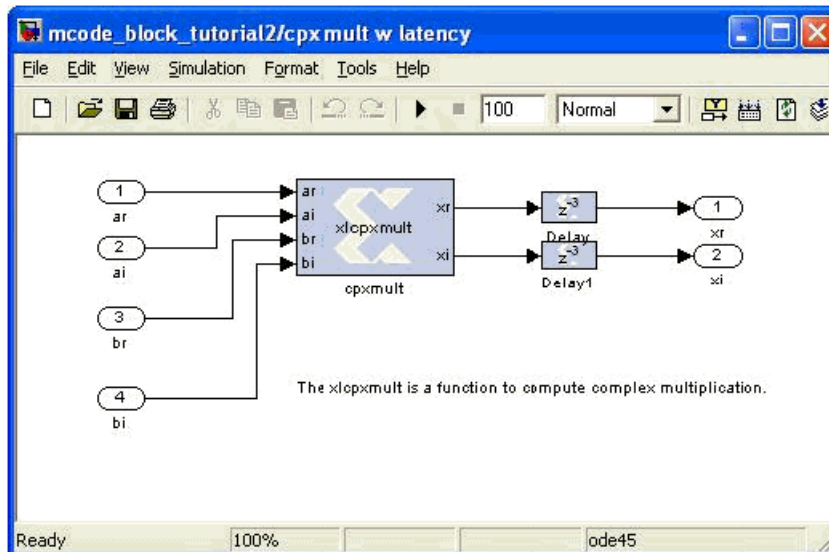
Notice that the two integer arguments (2 and 3) are converted to fixed-point numbers automatically. If you have a floating-point number as an argument, an `xfix` call is required.

Complex Multiplier with Latency

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

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

The following diagram shows the sub-system:



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:

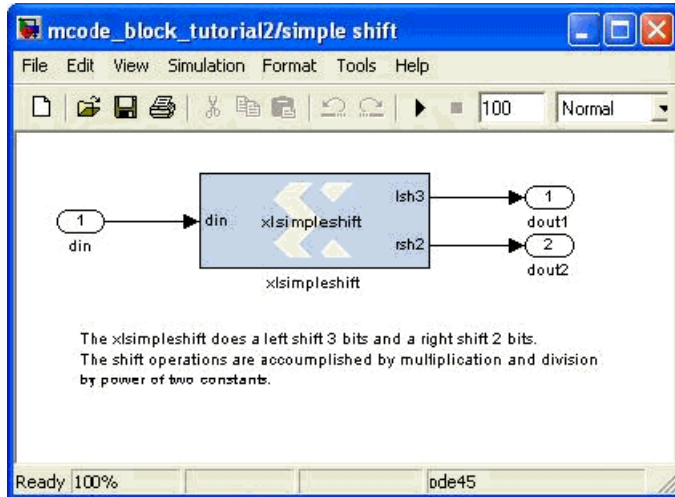
```
function [lsh3, rsh2] = xlsimpleshift(din)
    % [lsh3, rsh2] = xlsimpleshift(din) does a left shift
    % 3 bits and a right shift 2 bits.
    % The shift operation is accomplished by
    % multiplication and division of power
```

```

% of two constant.
lsh3 = din * 8;
rsh2 = din / 4;

```

The following diagram shows the sub-system after compilation:



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` is contained in file `xl_sconvert.m`:

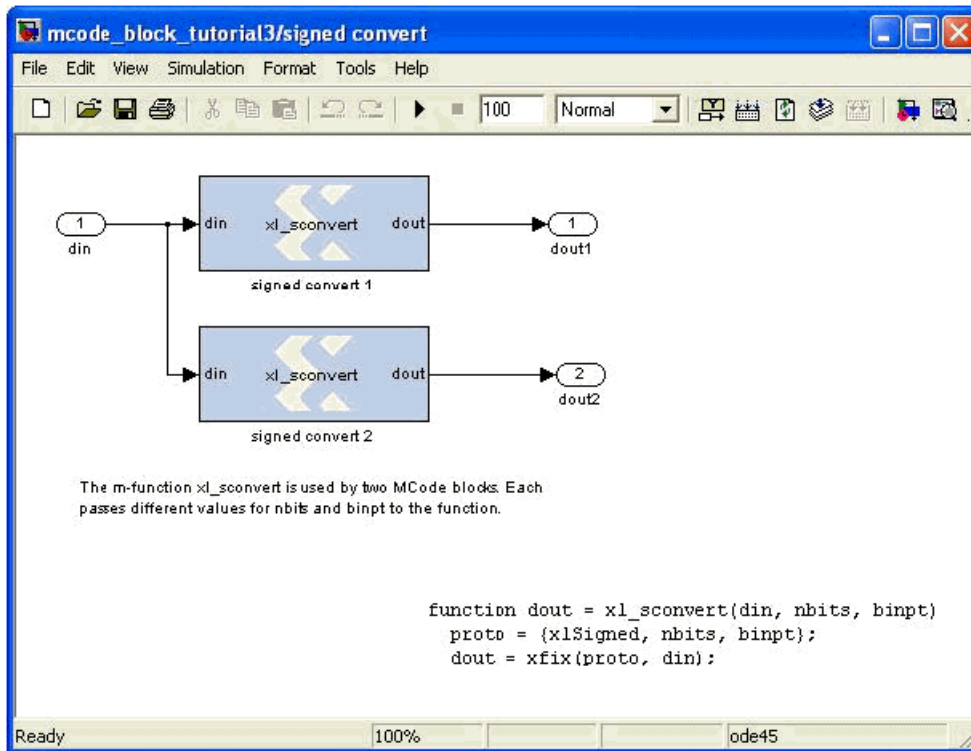
```

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

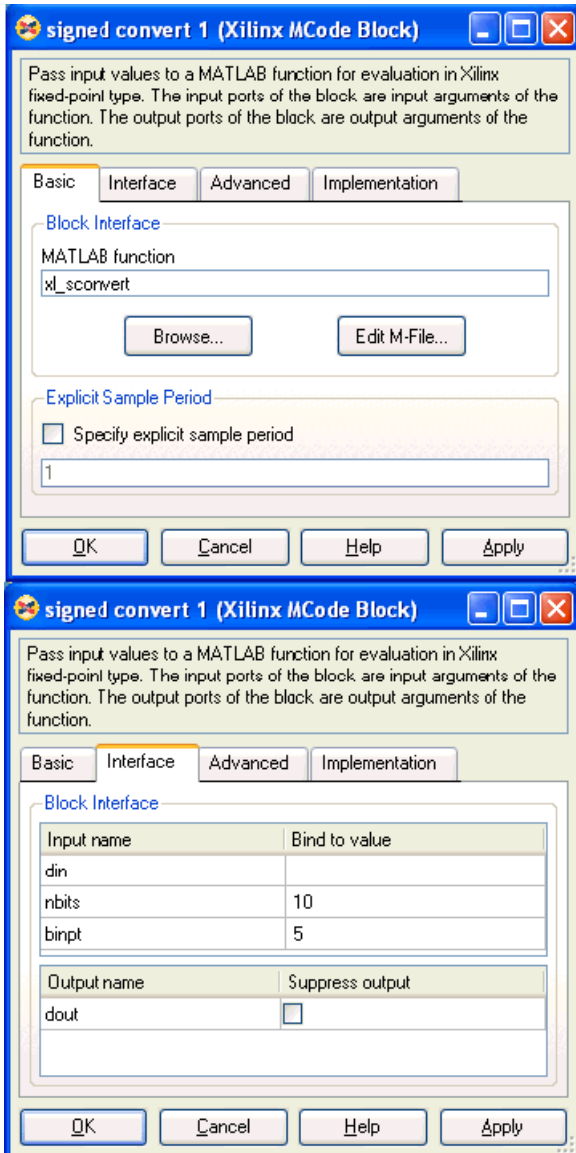
```

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

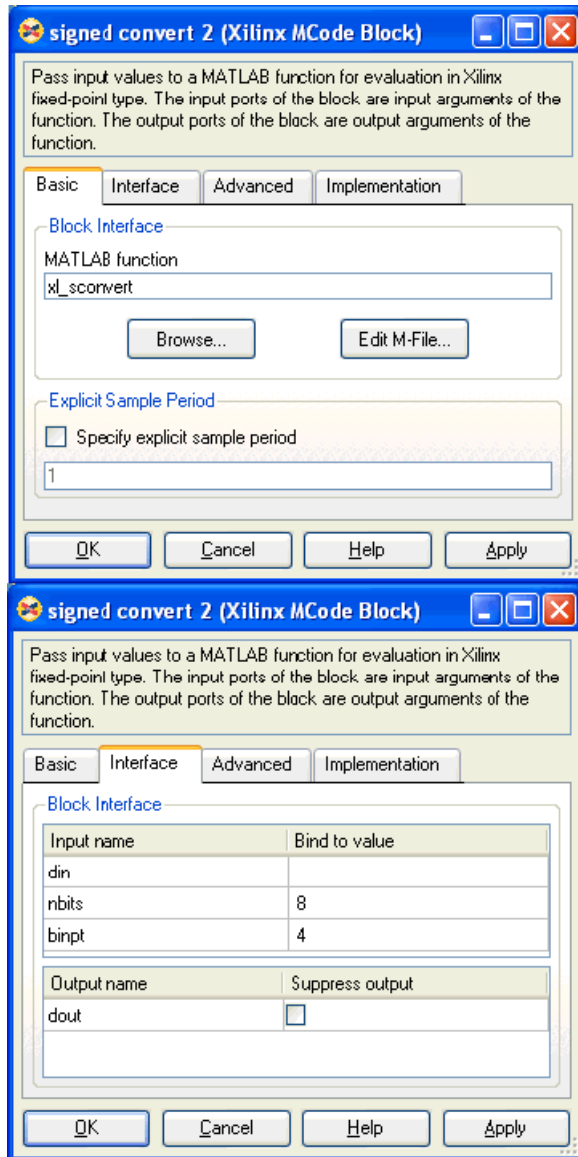
passed to the MCode block labeled signed convert2 causes it to convert the input data from type Fix_16_8 to Fix_8_4 at its output.



To pass parameters to each MCode block in the diagram above, you can click the **Edit Interface** 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:



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:



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.

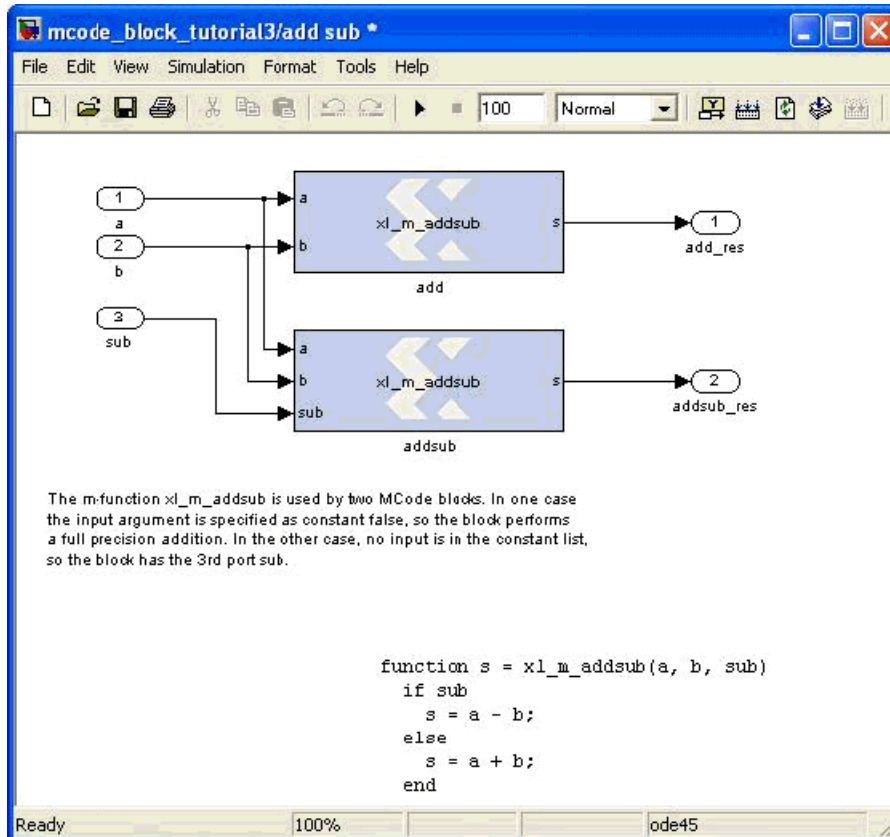
The following M-code, which defines M-function `xl_m_addsub` is contained in file `xl_m_addsub.m`:

```
function s = xl_m_addsub(a, b, sub)
```

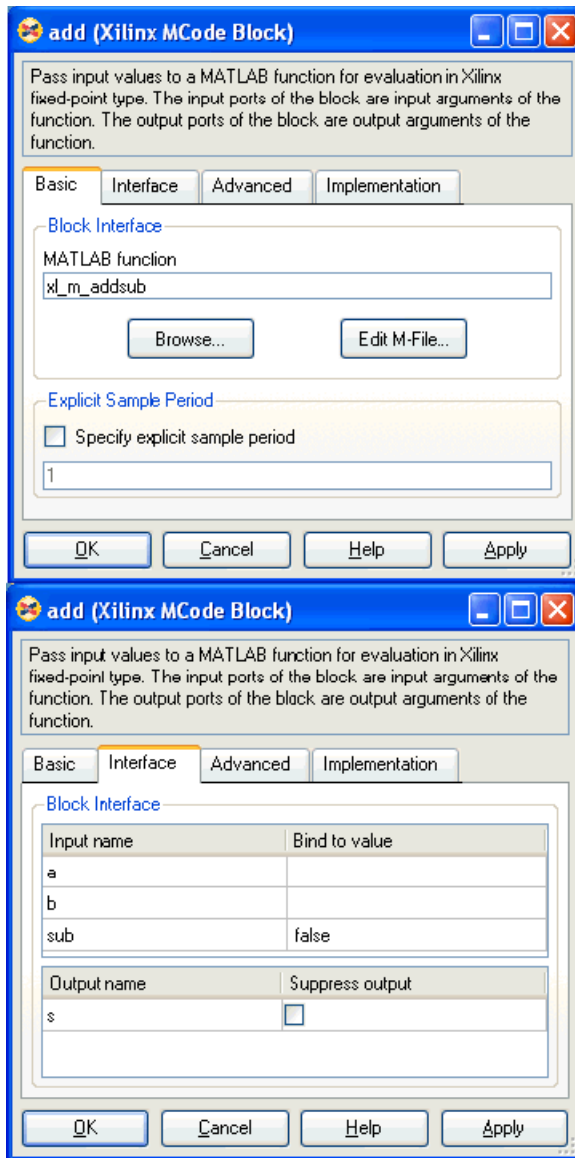
```

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.



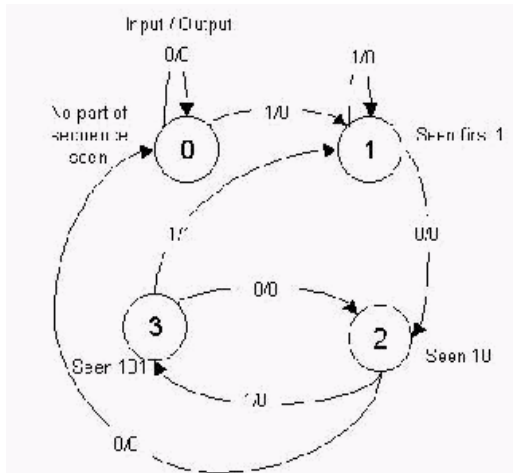
The Block Interface Editor of the MCode block labeled add is shown in below.



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.



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, if 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, {xUnsigned, 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

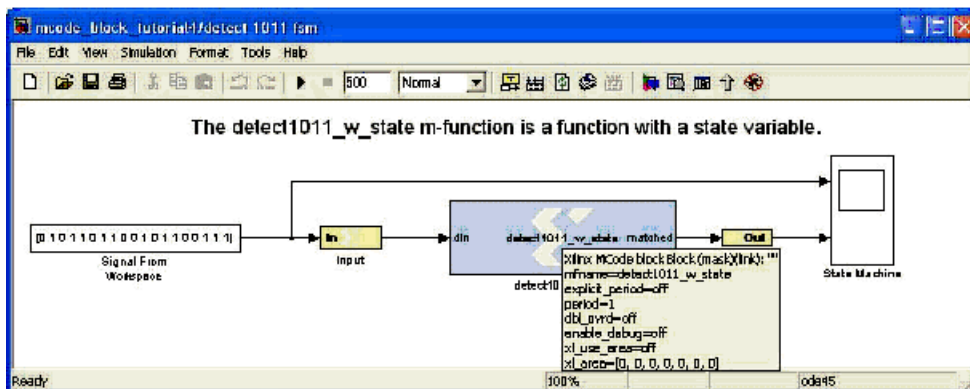
```

```

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
end

```

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



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

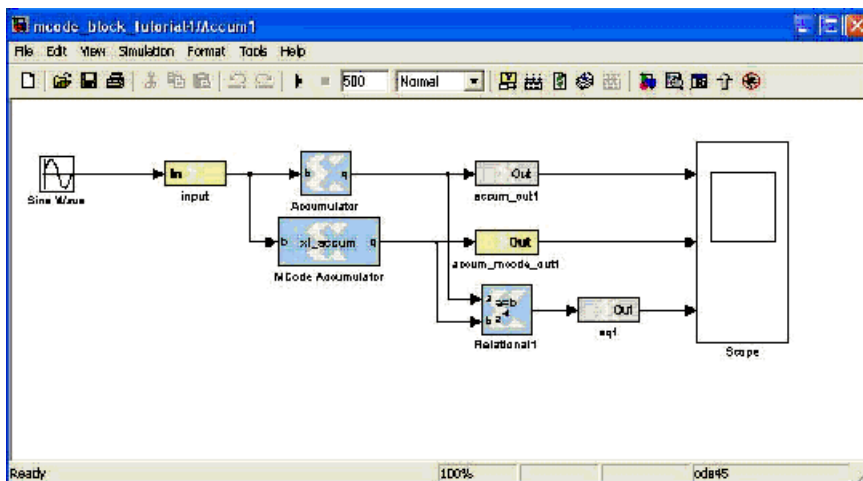
```

```

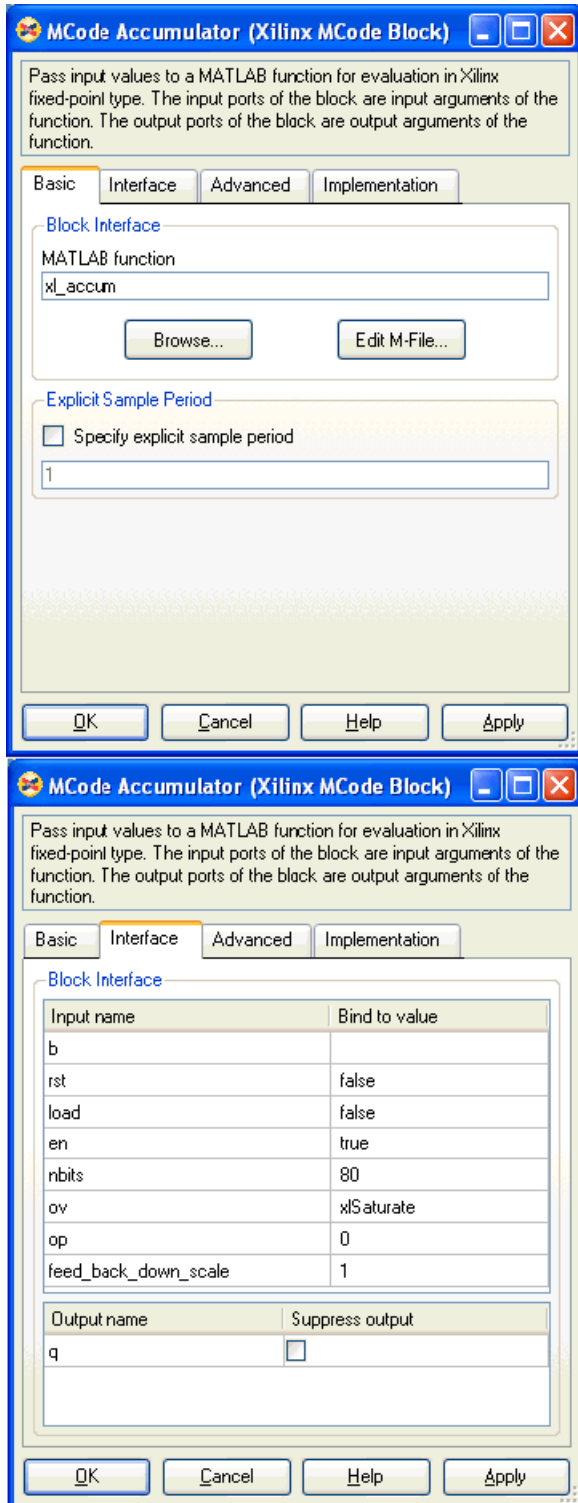
s = b;
else
% reset from zero
s = init;
end
else
if ~en
else
% if enabled, update the state
if op==0
s = s/feed_back_down_scale + b;
else
s = s/feed_back_down_scale - b;
end
end
end
end

```

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.



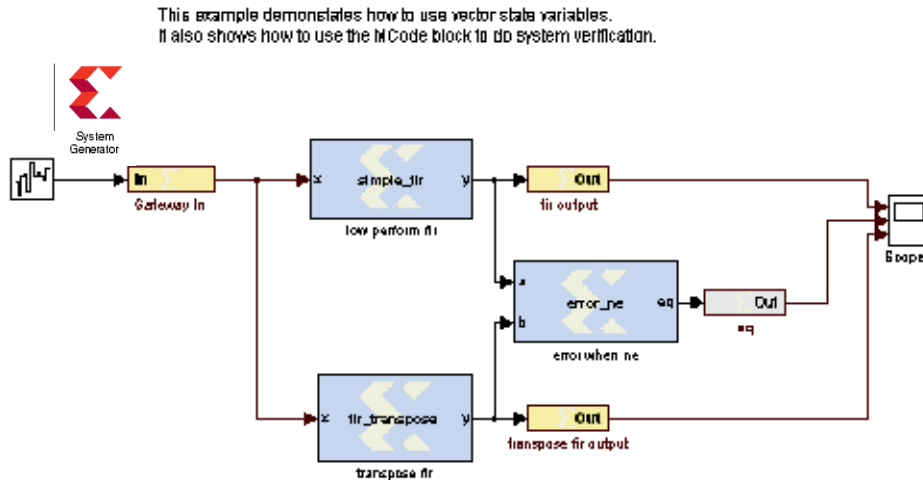
Optional inputs rst and load of block Accum_MCode1 are disabled in the cell array of the Function Parameter Bindings parameter. The block mask for block MCode Accumulator is shown below:



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.



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);
```

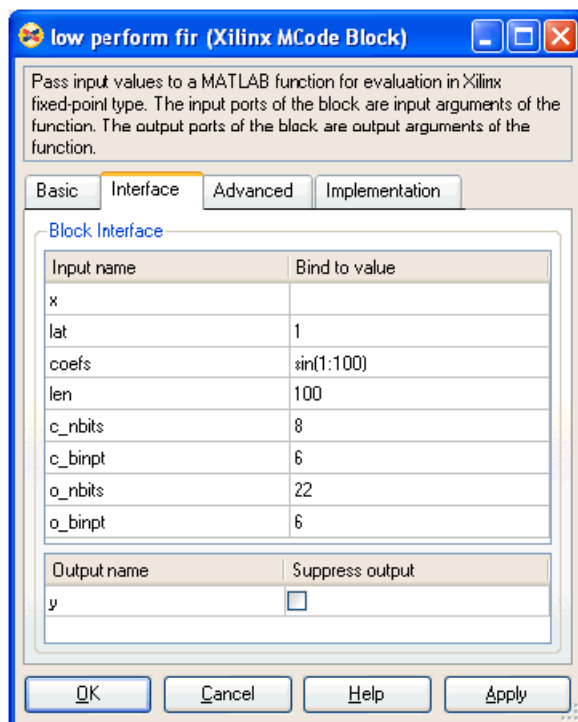
```
function y = fir_transpose(x, lat, coefs, len, c_nbits, c_binpt, o_nbits, o_binpt)
coef_prec = {xlSigned, c_nbits, c_binpt, xlRound, xlWrap};
out_prec = {xlSigned, o_nbits, o_binpt};
coefs_xfix = xfix(coef_prec, coefs);
persistent coef_vec, coef_vec = xl_state(coefs_xfix, coef_prec);
persistent reg_line, 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:



In order to verify that the functionality of two blocks are 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;

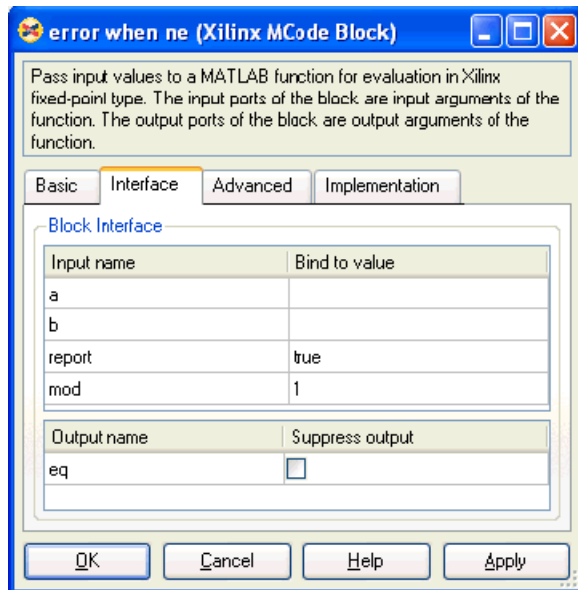
```

```

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;

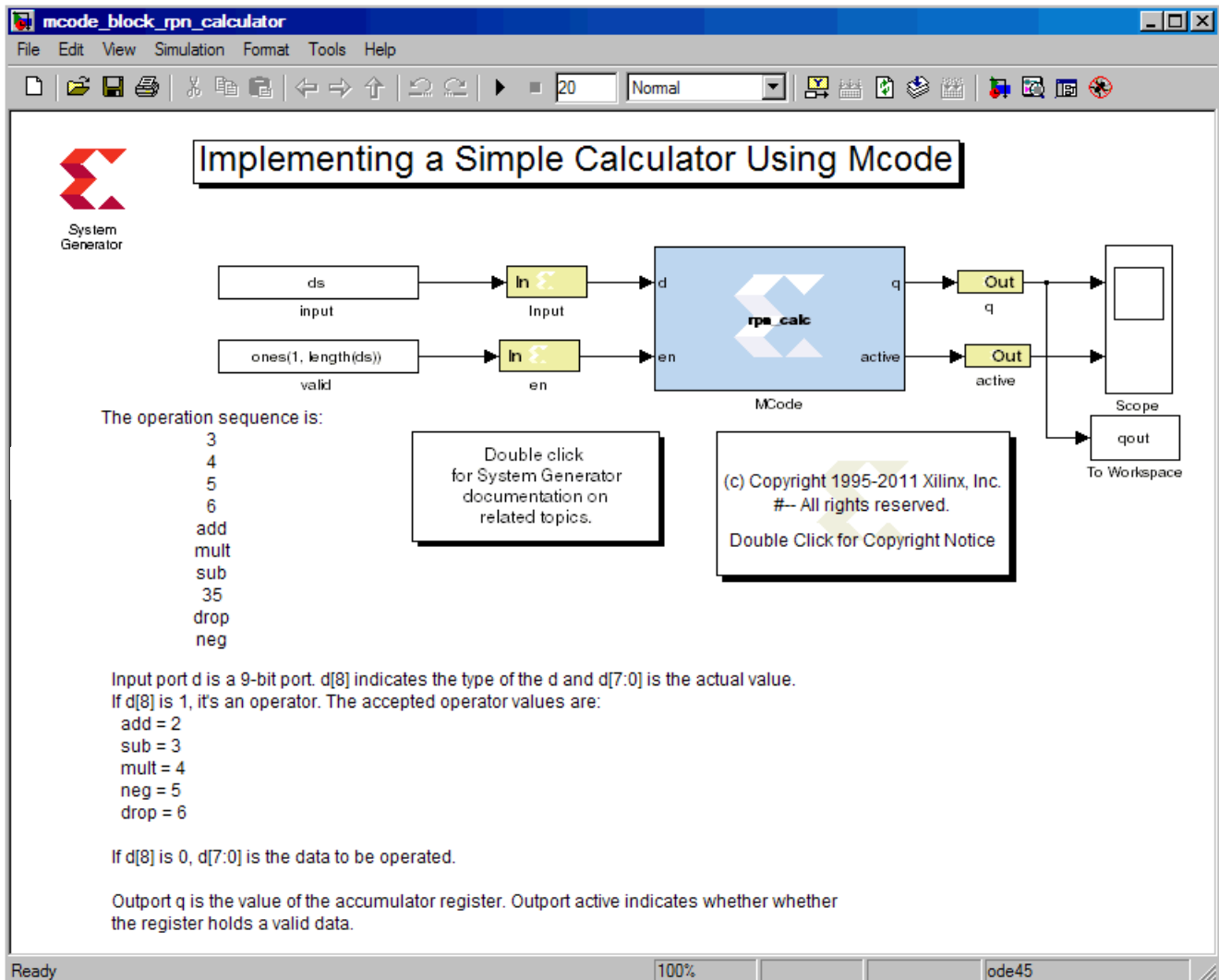
```

The block is configured as following:



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:



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), xSigned, 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.

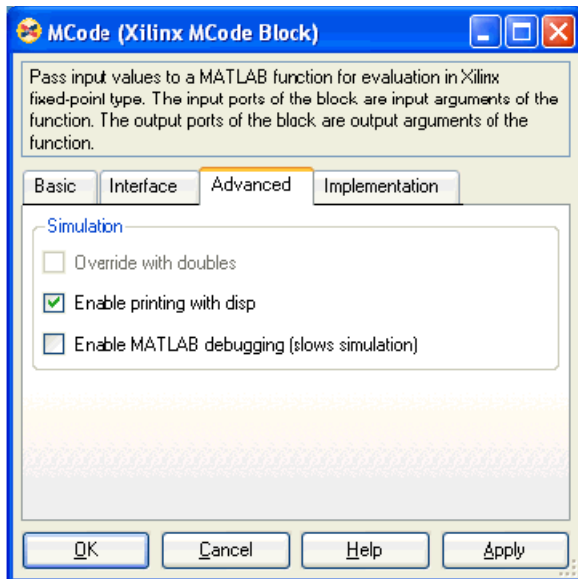
```
function x = testdisp(a, b)
```

```

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);

```

The Enable print with disp option must be checked.



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 the topic System Generator Compilation Types for more details.

The System Generator project information is encapsulated in the file <design_name>_mcw.sgp depending on which clocking option is selected. This topic shows how multiple System Generator designs can be included as sub-modules in a larger design.

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 bubbled up to the top-level design, appropriate buffers should be instantiated in the top-level HDL code.

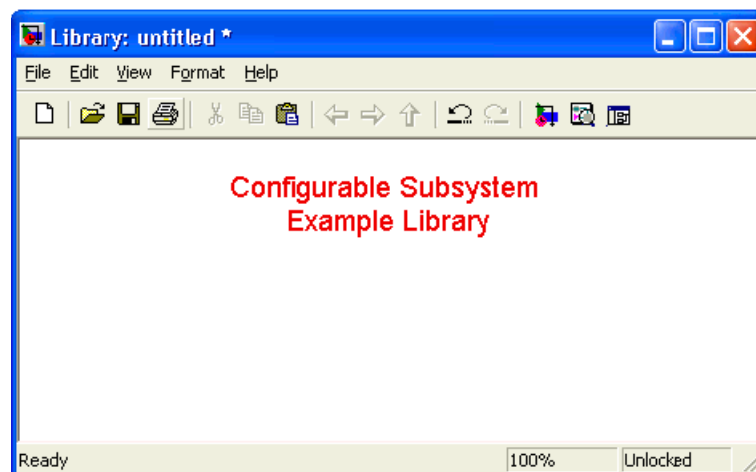
Configurable Subsystems and System Generator

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.

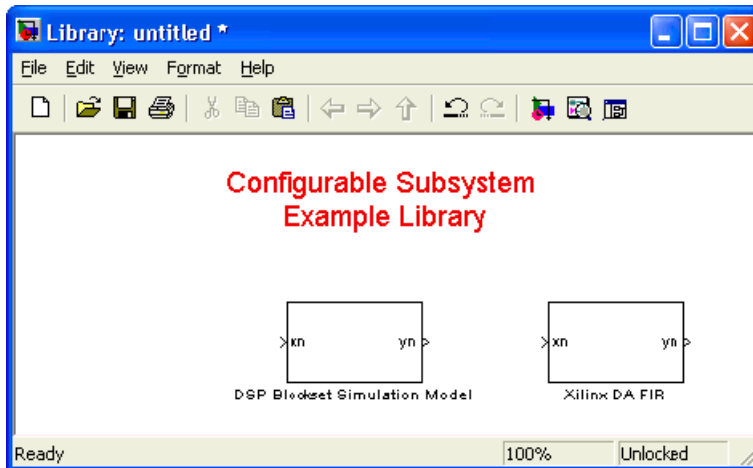
Defining a Configurable Subsystem

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:

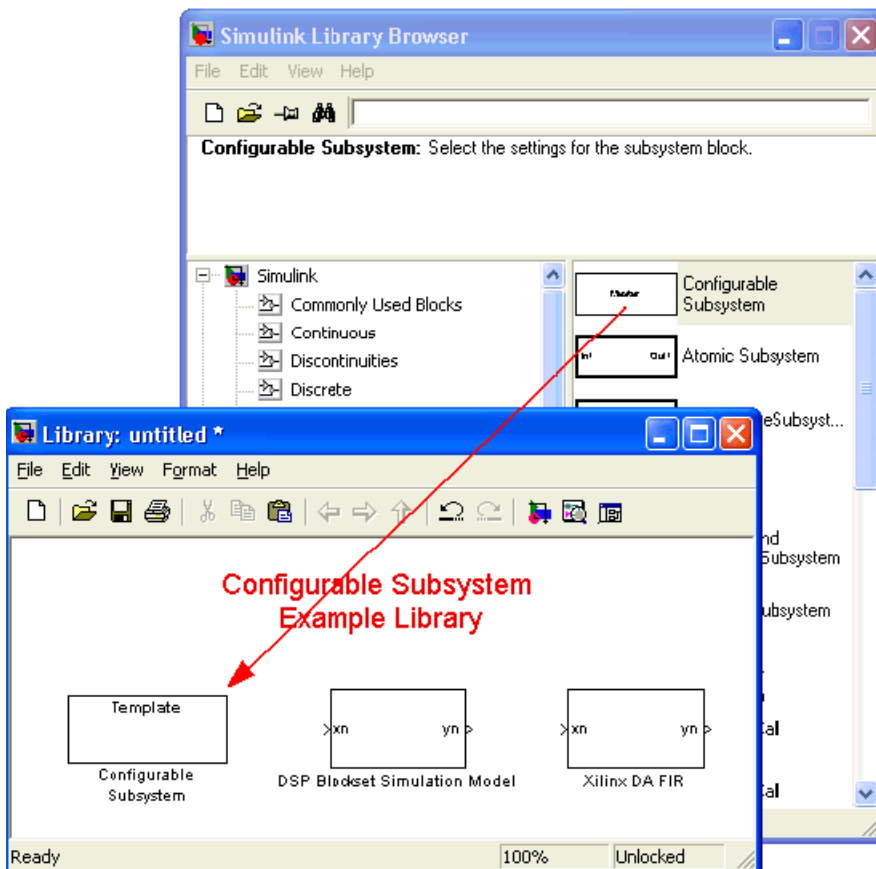
- Make a new empty library.



- Add the underlying blocks to the library.

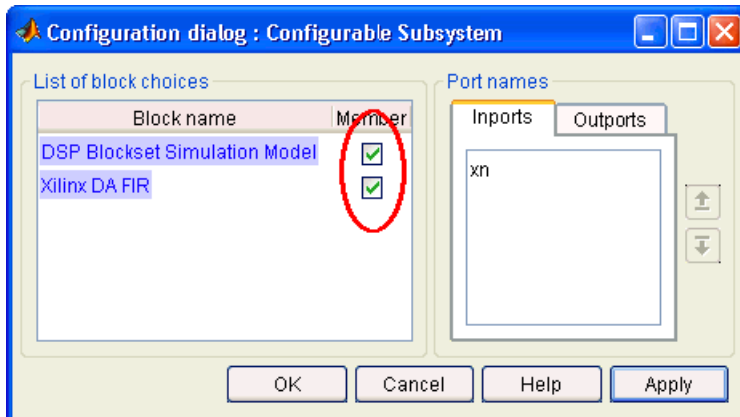


- Drag a template block into the library. (Templates can be found in the Simulink library browser under Simulink/Ports & Subsystems/Configurable Subsystem.)



- Rename the template block if desired.
- Save the library.
- Double click to open the template for the library.

- In the template GUI, turn on each checkbox corresponding to a block that should be an implementation.



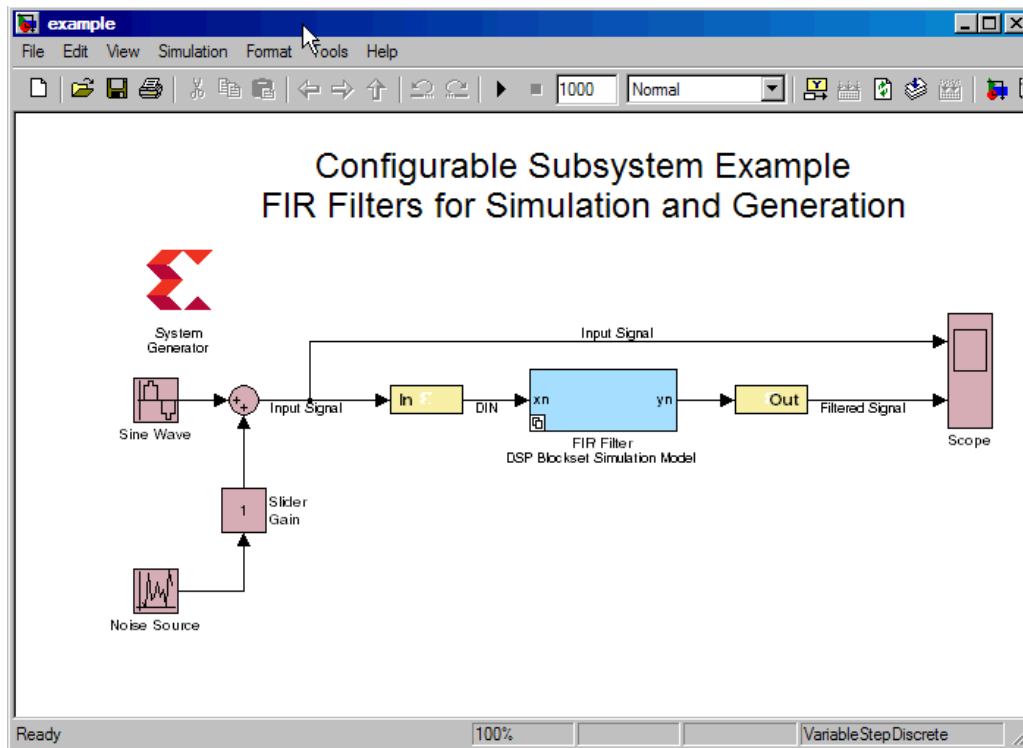
- Press **OK**, and then save the library again.

Using a Configurable Subsystem

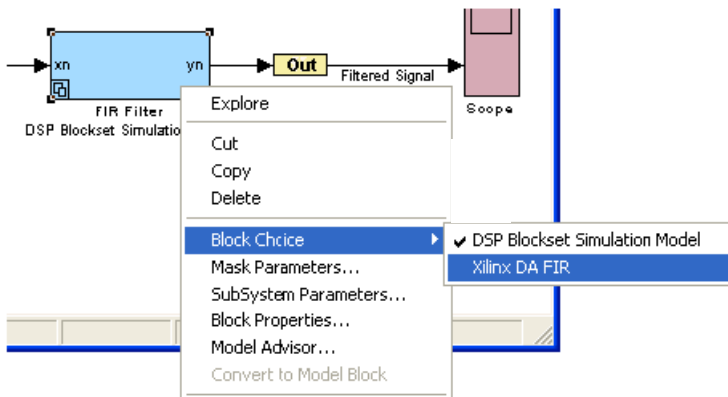
To use a configurable subsystem in a design, do the following:

- As described above, create the library that defines the configurable subsystem.
- Open the library.
- Drag a copy of the template block from the library to the appropriate part of the design.

- The copy becomes an instance of the configurable subsystem.



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

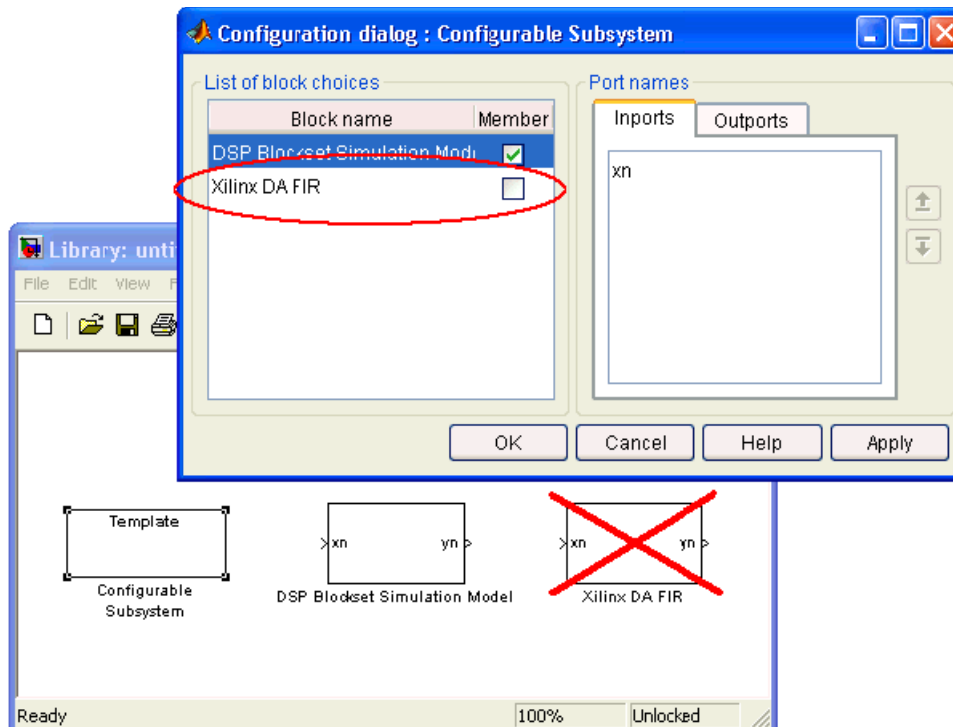


Deleting a Block from a Configurable Subsystem

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

- Open and unlock the library for the subsystem.
- Double click on the template, and turn off the checkbox associated to the block to be deleted.

- Press **OK**, and then delete the block.



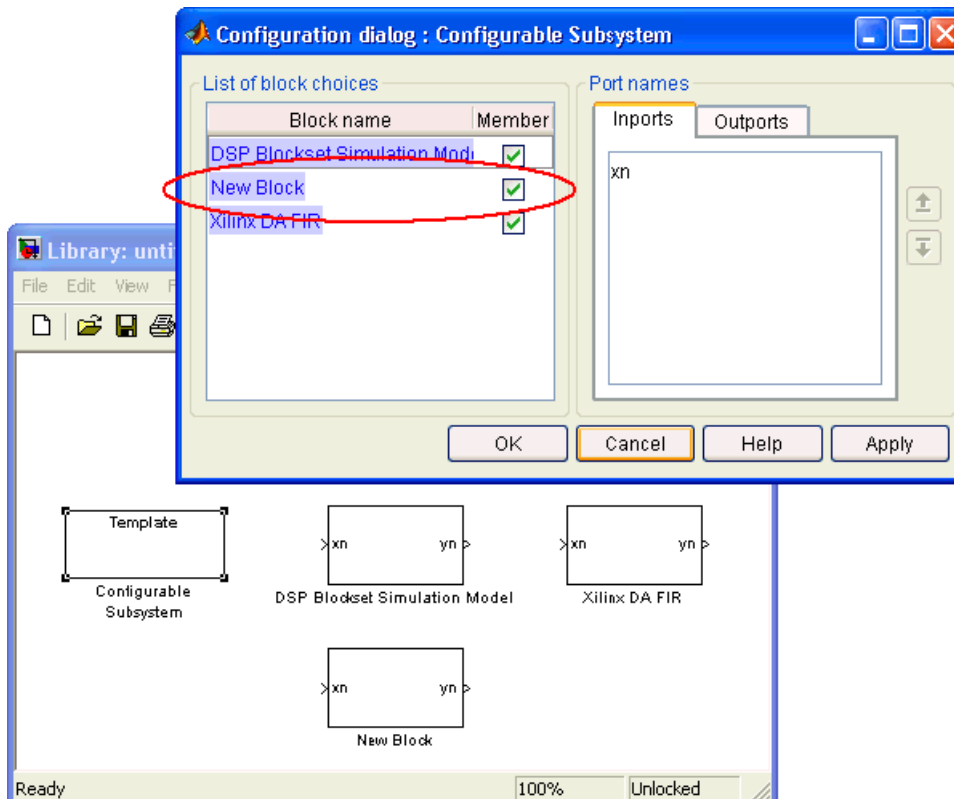
- Save the library.
- Compile the design by typing Ctrl-d.
- If necessary, update the choice for each instance of the configurable subsystem.

Adding a Block to a Configurable Subsystem

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

- Open and unlock the library for the subsystem.
- Drag a block into the library.

- Double click on the template, and turn on the checkbox next to the added block.



- Press **OK**, and then save the library.
- Compile the design by typing Ctrl-d.
- If necessary, update the choice for each instance of the configurable subsystem.

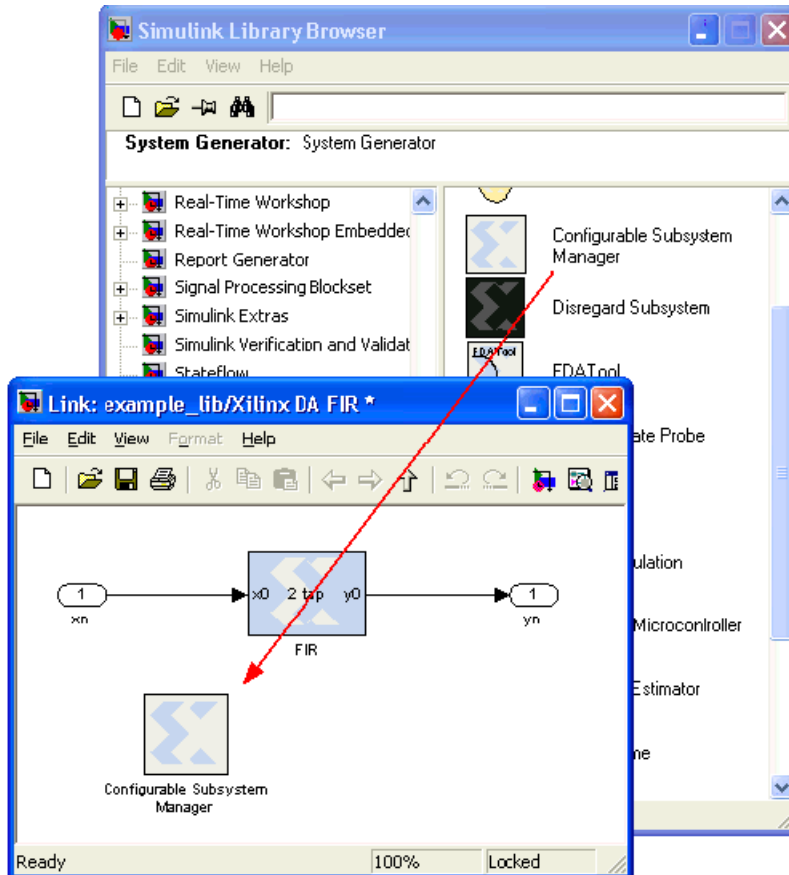
Generating Hardware from Configurable Subsystems

In System Generator, blocks both participate in simulations and produce hardware. Sometimes, for a configurable subsystem, it is worthwhile to use one underlying block for simulation, but use another for hardware generation. For example, it might make sense to use ordinary System Generator blocks to produce simulation results, but use a black box to supply the corresponding HDL. The System Generator configurable subsystem manager block makes this possible; the ordinary block choice for the configurable subsystem is used when simulating, and the block specified in the manager is used for hardware generation.

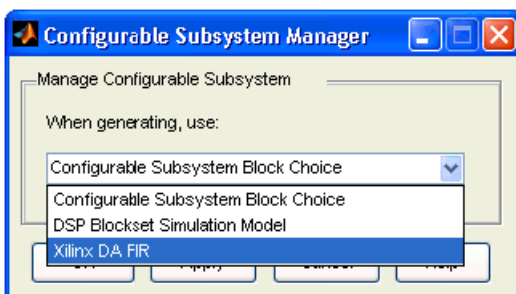
To use a configurable subsystem manager, do the following:

- Open and unlock the library for the configurable subsystem.
- Select one of the blocks in the library, and double click to open it. (Aside from the template any block will do, provided the block is itself a subsystem. If there is no such subsystem in the library, it is not possible to use a configurable subsystem manager.)

- Drag a manager block into the subsystem opened above. (The manager block can be found in Xilinx Blockset/Tools/Configurable Subsystem Manager).



- Double click to open the GUI on the manager, then select the block that should be used for hardware generation in the configurable subsystem.



- Press **OK**, then save the subsystem, and the library.

The MathWorks description of configurable subsystems can be found the following address:

<http://www.mathworks.com/access/helpdesk/help/toolbox/simulink/slref/configurablesubsystem.shtml>.

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.

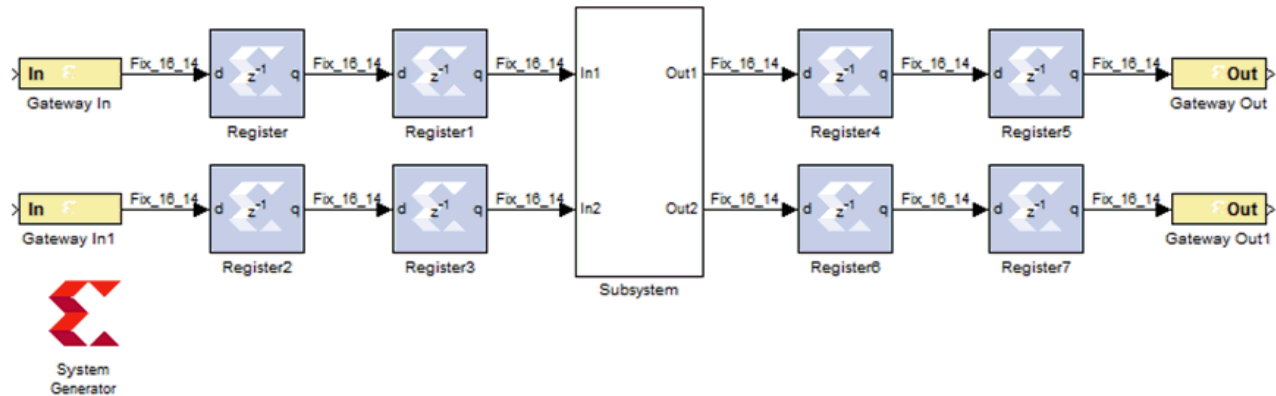
- [Review the Hardware Notes Included with Each Block Dialog Box](#)
- [Register the Inputs and Outputs of Your Design](#)
- [Insert Pipeline Registers](#)
- [Use Saturation Arithmetic and Rounding Only When Necessary](#)
- [Set the Data Rate Option on All Gateway Blocks](#)
- [Other Things to Try](#)

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.

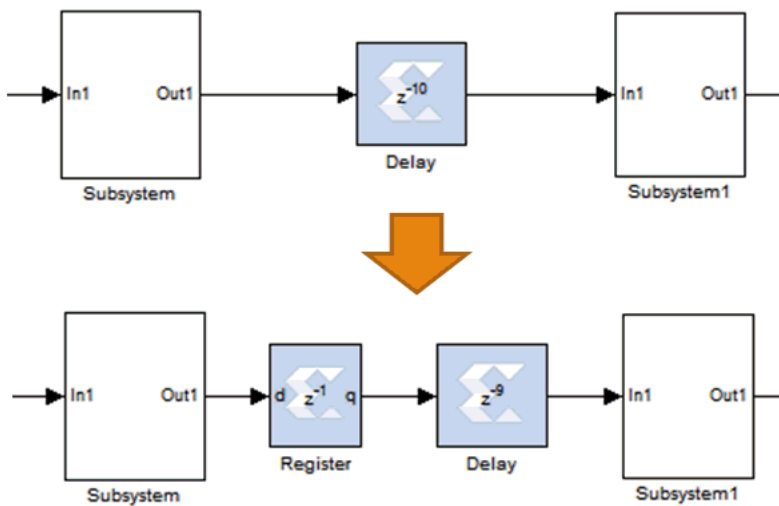


Double registering the I/Os may also be beneficial. This can be performed by instantiating two separate Register blocks, or by instantiating two Delay blocks, each having latency 1. This allows one of the registers to be packed into the IOB and the other to be placed next to the logic in the FPGA fabric. A Delay block with latency 2 does not give the same result because the block with a latency of 2 is implemented using an SRL16 and cannot be packed into an IOB.

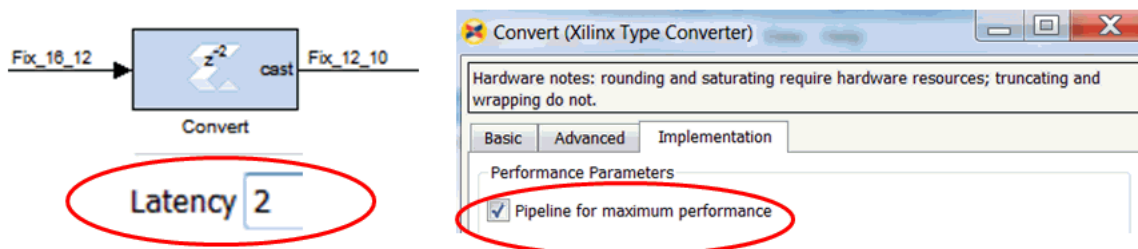
Insert Pipeline Registers

Insert pipeline registers wherever possible and reasonable. Deep pipelines are efficiently implemented with the Delay blocks since the SRL16 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 SRL16 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.

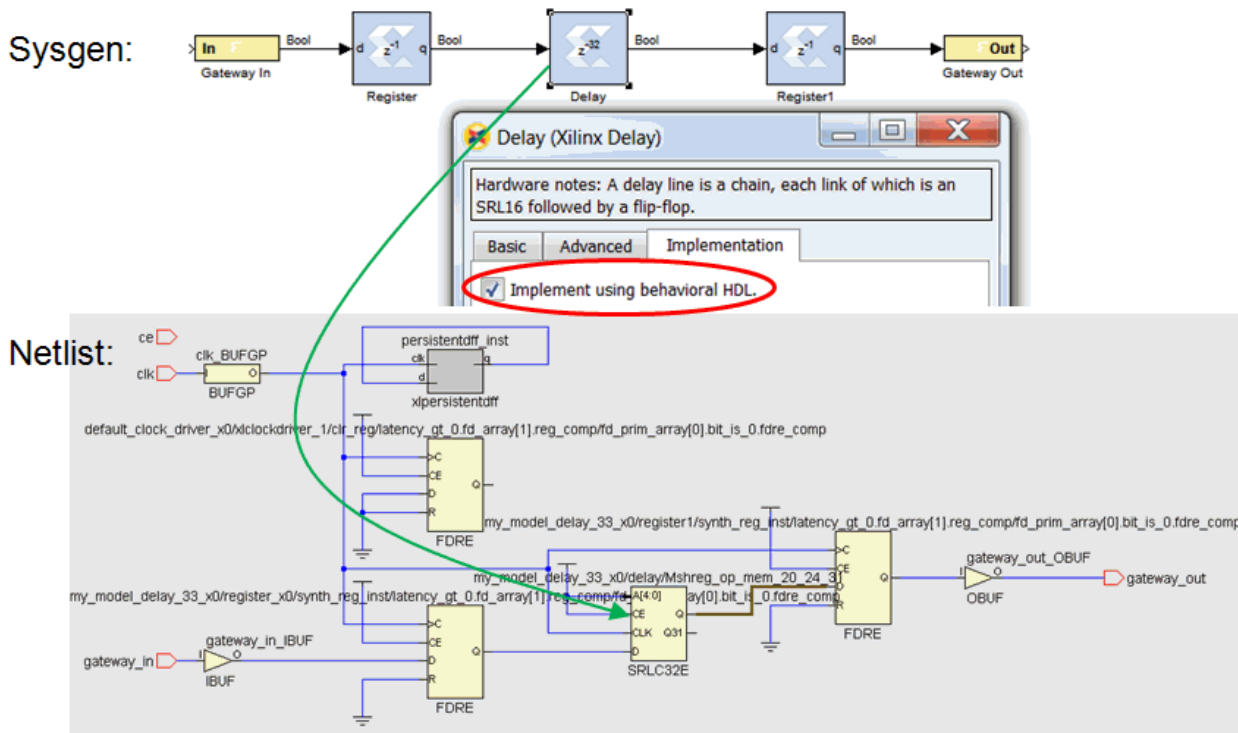


As shown below, the Convert block can be pipelined with embedded register stages to guarantee maximum performance.



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.



For BRAMS (Block RAMS), use the internal output register. You do this by setting the latency from 1 (the default) to 2. This enables the BRAM 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.

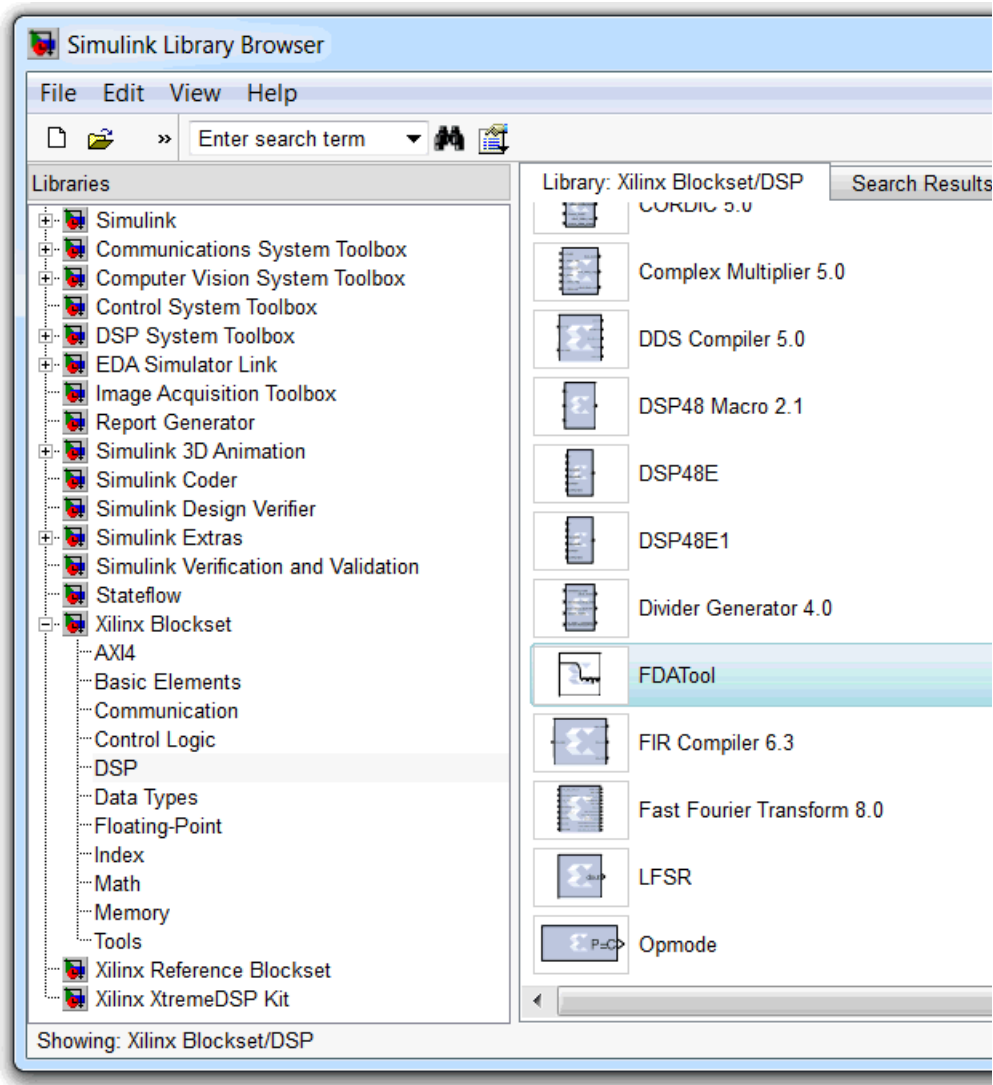
Other Things to Try

- Change the Source Design
 - Use Additional Pipelining
Use the Output and Pipeline registers inside BRAM 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
Don't 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 following example demonstrates one way of specifying, implementing, and simulating a FIR filter using the FDATool block. The FDATool block is used to define the filter order and coefficients and the Xilinx Blocksets are used to implement a MAC-based FIR filter using a

single MAC (Multiply-ACcumulate) engine. The quality of frequency response is then validated by comparing it to a double-precision Simulink filter model.



Although a single MAC engine FIR filter is used for this example, we strongly recommend that you look at the DSP Reference Library provided as a part of the Xilinx Reference Blockset. The DSP Reference Library consists of multi-MAC, as well as, multi-channel implementation examples with variations on the type of memory used.

A demo included in the System Generator demos library also shows an efficient way to implement a MAC-based interpolation filter. To see the demo, type the following in the MATLAB Command Window:

```
>> demo blockset xilinx
```

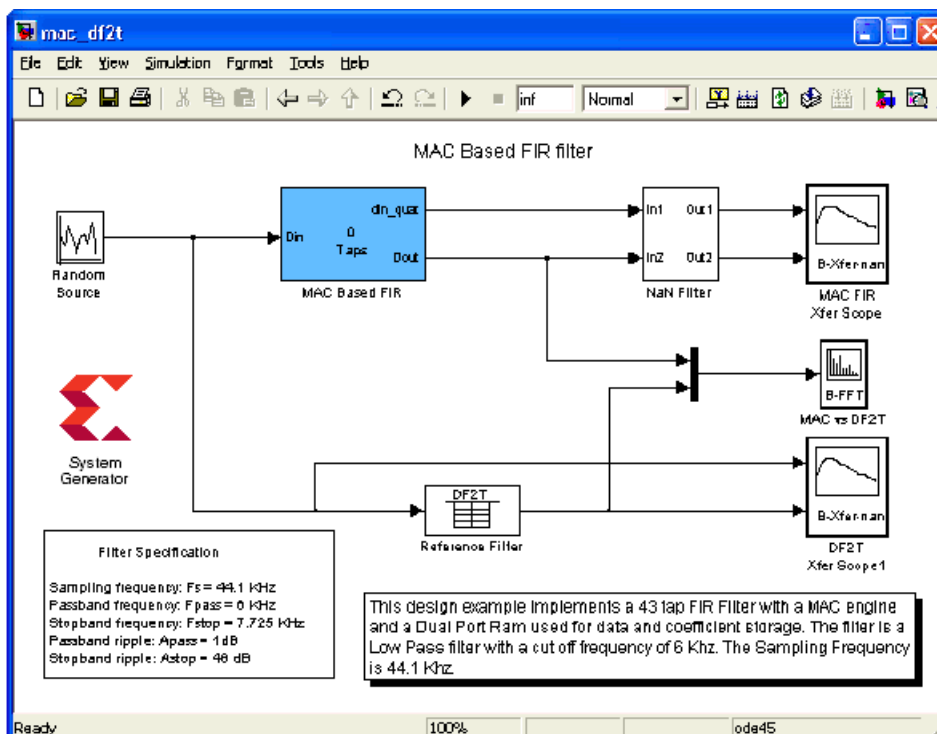
then select FIR filtering: Polyphase 1:8 filter using SRL16Es from the list of demo designs.

Design Overview

This design uses the random number source block from the DSP Blockset library to drive two different implementations of a FIR filter:

- The first filter is the one that could be implemented in a Xilinx device. It is a fixed-point FIR filter implemented with a dual-port Block memory and a single multiply-accumulator.
- The second filter is what is referred to as reference filter. It is a double-precision, direct-form II transpose filter.

The frequency response of each filter is then plotted in a transfer function scope.



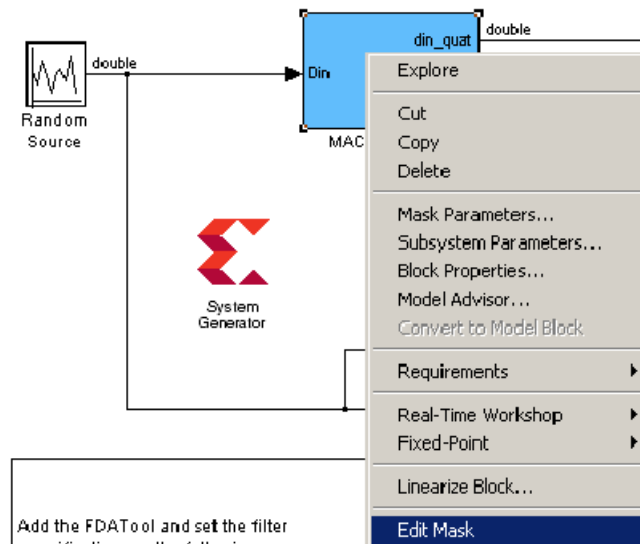
Open and Generate the Coefficients for this FIR Filter

1. From the MATLAB console window, cd into the directory C:\ug897-example-files\mac_df2t.
2. Open the design model by typing mac_df2t from your MATLAB Command Window.

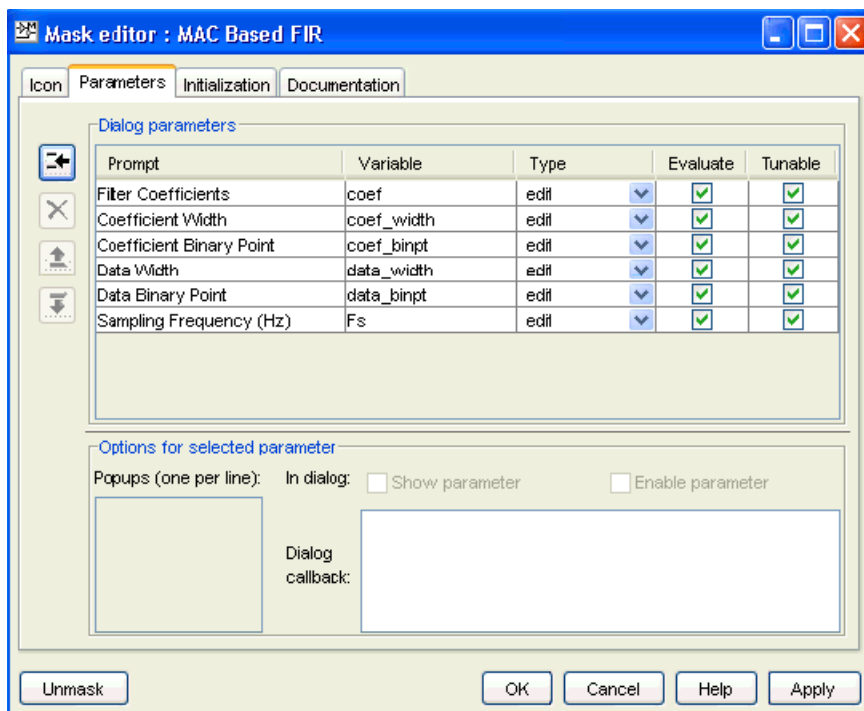
For the purpose of this exercise, the variables coef, coef_width, coef_binpt, data_width, data_binpt and Fs are not defined. You will first use these variables as mask parameters to the MAC Based FIR block and then design and assign the filter coefficients using the FDATool. The fully functional model is available in the current directory and is called mac_df2t_soln.mdl.

Parameterize the MAC-Based FIR Block

1. Right Click on the MAC-Based FIR block and select **Edit Mask** as shown in the figure below.

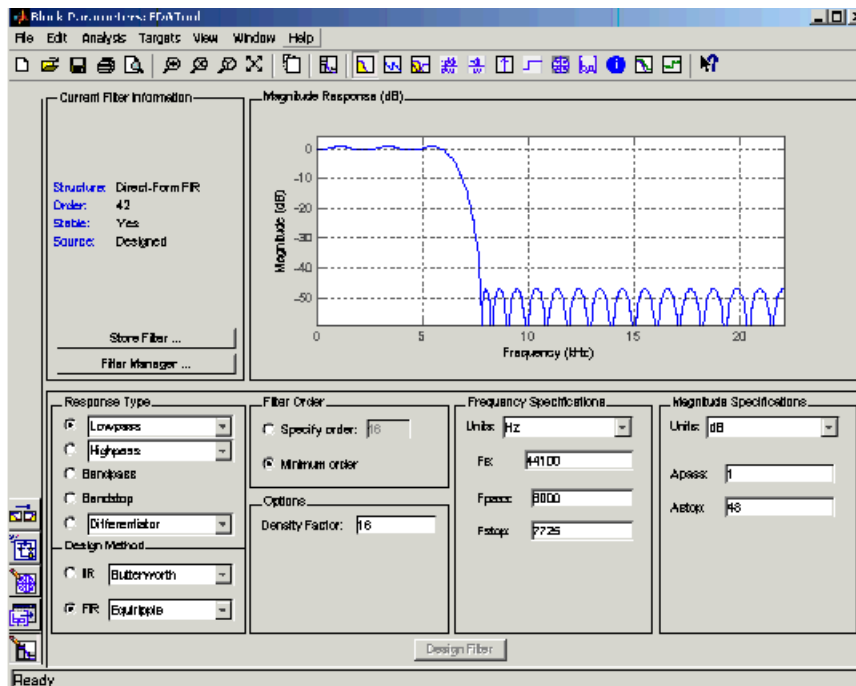


2. Double-click on the Parameters tab and add the parameters coef, data_width and data_binpt as shown below.



Generate and Assign Coefficients for the FIR Filter

1. Drag and drop the FDATool block into your model from the DSP Xilinx Blockset Library.
2. Double-click on the FDATool block and enter the following specifications in the Filter Design & Analysis Tool for a low-pass filter designed to eliminate high-frequency noise in audio systems:
 - Response Type: **Lowpass**
 - Filter Order: **Minimum order**
 - Frequency Specifications
 - Units: **Hz**
 - Fs: **44100**
 - Fpass: **6000**
 - Fstop: **7725**
 - Magnitude Specifications
 - Units: **dB**
 - Apass: **1**
 - Astop: **48**



3. Click on **Design Filter** at the bottom of the tool window 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 GUI. Based on the FDATATool, a 43-tap FIR filter (order 0-42) is required in order to meet the design specifications listed above.

The filter coefficients can be displayed in the MATLAB workspace by typing:

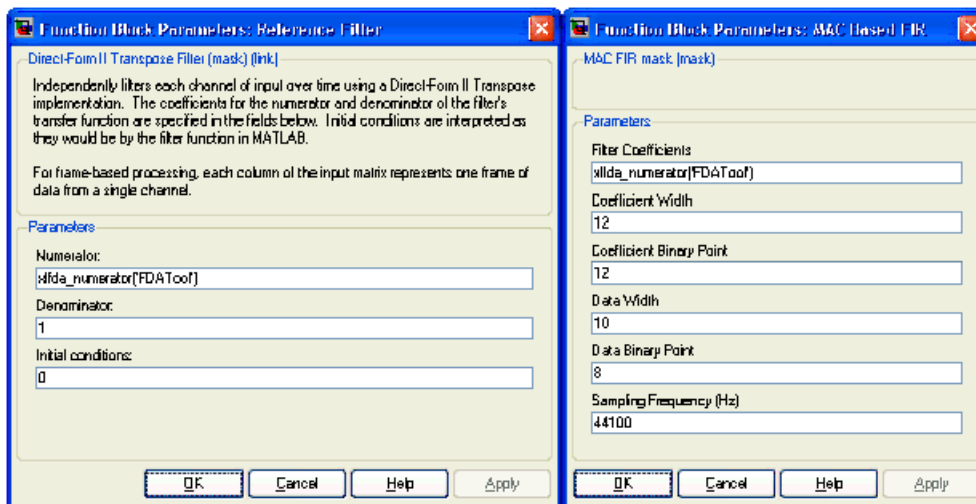
```
>> xlfda_numerator('FDATool')
```

These useful functions help you find the maximum and minimum coefficient value in order to adequately specify the coefficient width and binary point:

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

For this exercise, the coefficient type has been set to be Fix_12_12, which is a 12-bit number with the binary point to the left of the twelfth bit. The result of the max() function above shows that the largest coefficient is 0.3022, which means that the binary point may be positioned to the left of the most significant bit. How do you reason that? A Fix_12_12 number has a range of -0.5 to 0.4998, meaning the dynamic range is maximized by putting the binary point left of the most significant bit. If you moved the binary point to the right (by using a Fix_12_11 number) you would lose one bit of dynamic range because a Fix_12_11 number has a range of -1 to 0.9995, which is more than you require to represent the coefficients.

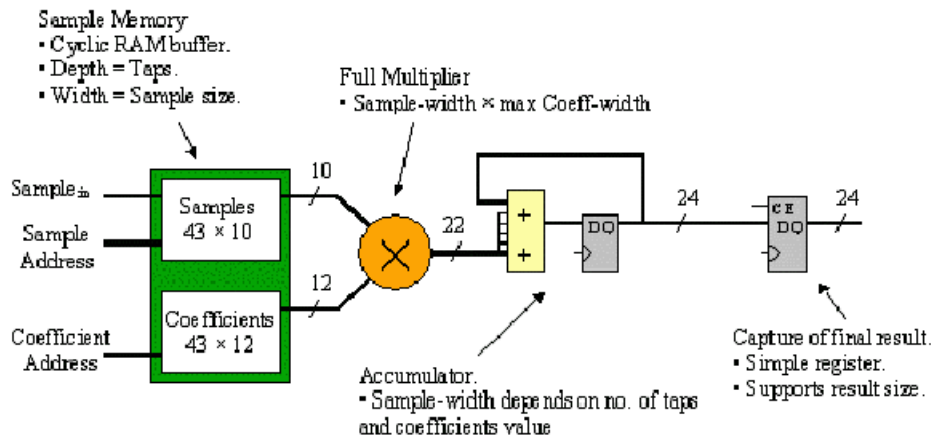
4. Click on the Reference Filter block and the MAC Based FIR block and verify the parameter values for coef, coef_width, coef_binpt, data_width, data_binpt and Fs as shown below.



Click OK on each dialog box

Browse Through and Understand the Xilinx Filter Block

The following block diagram showing how the MAC-based FIR filter has been implemented for this exercise.



At this point, the MAC filter is set up for a 10-bit signed input data (Fix_10_8), a 12-bit signed coefficient (Fix_12_12), and 43 taps. All these parameters can be modified directly from the MAC block GUI. The coefficients and data need to be stored in a memory system. For the exercise, you choose to use a dual-port memory to store the data and coefficients, with the data being captured and read out using a circular RAM buffer. The RAM is used in a mixed-mode configuration: values are written and read from port A (RAM mode), and the coefficients are only read from port B (ROM mode).

The multiplier is set up to use the embedded multiplier resource available in Xilinx 7 series devices as well as three levels of latency in order to achieve the fastest performance possible. The precision required for the multiplier and the accumulator is a function of the filter taps (coefficients) and the number of taps. Since these are fixed at design time, it is possible to tailor the hardware resources to the filter specification. The accumulator need only have sufficient precision to accumulate maximal input against the filter taps, which is calculated as follows:

$$\text{acc_nbits} = \text{ceil}(\log_2(\text{sum}(\text{abs}(\text{coef} * 2^{\text{coef_width_bp}})))) + \text{data_width} + 1;$$

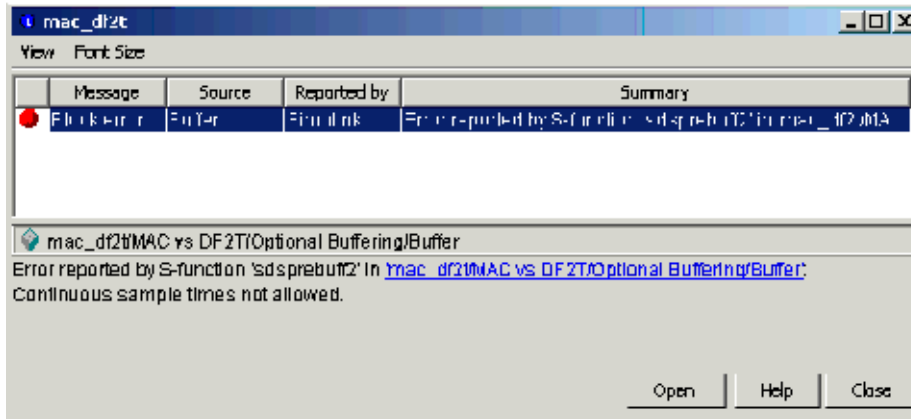
Upon reset, the accumulator re-initializes to its current input value rather than zero, which allows the MAC engine to stream data without stalling. A capture register is required for streaming operation since the MAC engine reloads its accumulator with an incoming sample after computing the last partial product for an output sample.

Finally, a downsampler reduces the capture register sample period to the output sample period. The block is configured with latency to obtain the most efficient hardware implementation. The downsampling rate is equal to the coefficient array length.

Run the Simulation

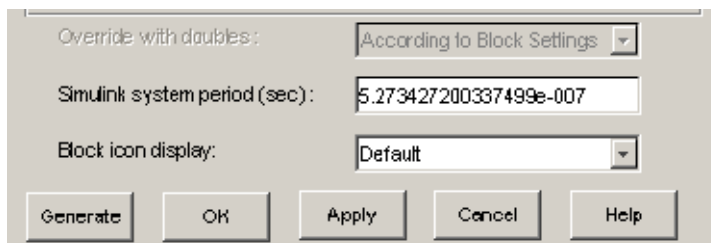
1. Change the simulation time to 0.05, then run the simulation

You should get the message shown in the figure below.



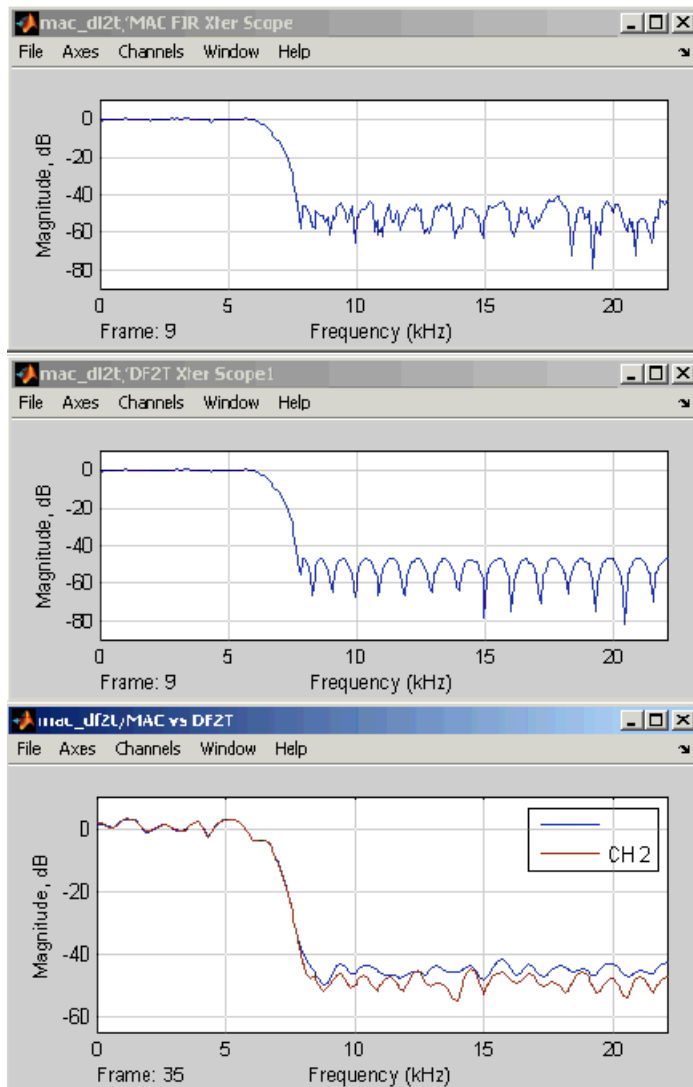
System Generator gets its input sample period from the din **Gateway In** block which has 1/Fs specified as the data input sample period. As the MAC-based FIR filter is over-sampled according to the number of taps, the System Clock Period will always be equal to 1/(Filter Taps * Fs).

2. Double click on the System Generator token and change the Simulink system period to specify the System Clock Period as $5.273427e-007 = 1/(43 * 44100)$ as shown below.



3. Run the simulation again and notice that the Xilinx implementation of the MAC-based FIR filter meets the original filter specifications and that its frequency response is almost identical to the double precision Simulink models.

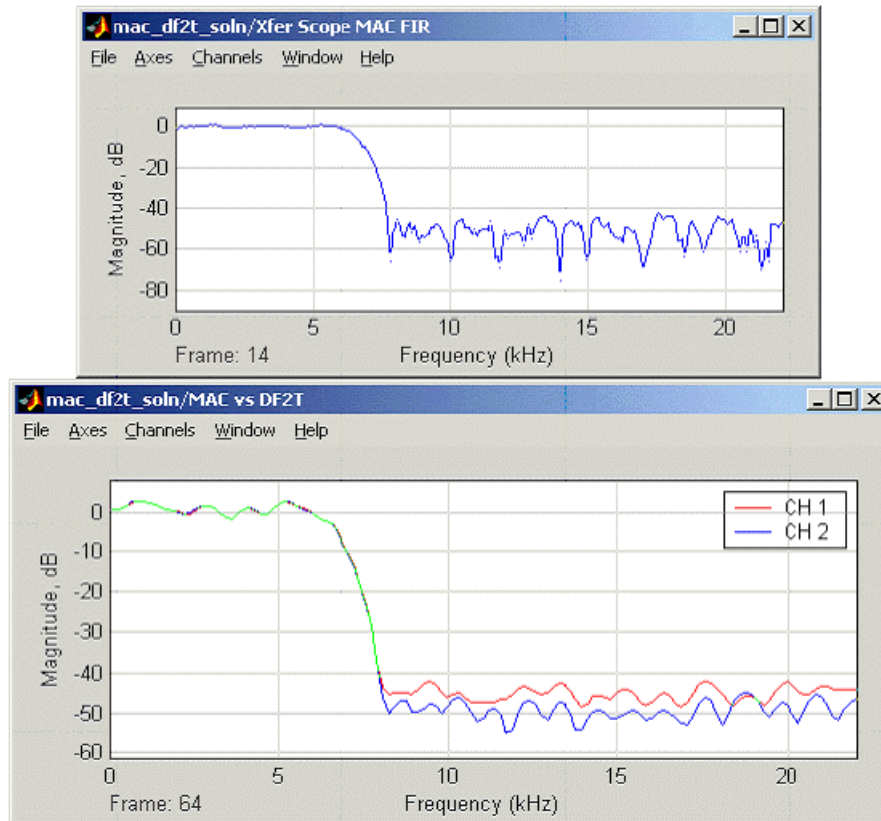
As you can see, the filter passband response measurement as well as zeros can clearly be seen. You should get similar frequency responses as shown in the following figure.



It is possible to increase or decrease the precision of the Xilinx Filter in order to reach the perfect area/performance/quality trade off required by your design specifications.

Stop the simulation and modify the coefficient width to **FIX_10_10** and the data width to **FIX_8_6** from the block GUI. Update the model (Ctrl-d) and push into the MAC engine block. You should now notice that the datapath has been automatically updated to only eighteen bits on the output of the multiplier and twenty on the output of the accumulator.

Restart the simulation and observe how the frequency response has been affected. The attenuation has indeed degraded (less than 40dB) due to the fixed-wordlength effects.



AXI Interface

Introduction

AMBA® AXI™4 (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 and AXI4-Stream.

- AXI4 is a memory-mapped interface which support burst transactions
- AXI4-Lite 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 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 in <http://www.xilinx.com/ipcenter/axi4.htm>.

AXI4-Stream Support in System Generator

The 3 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 instance:

m_axis_tvalid

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

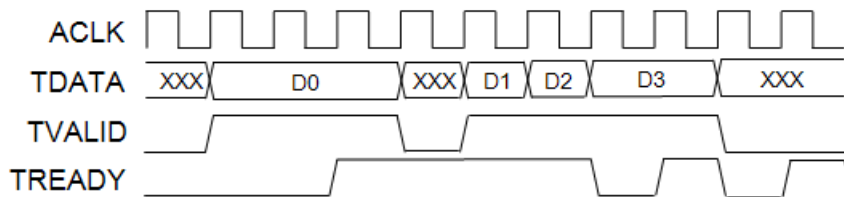
s_axis_control_tdata

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 AXI-Streaming 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.



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

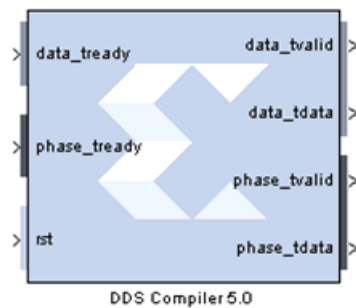
control signals that need to propagate in the opposite direction must either be part of a separate channel (“back-channel” with separate TREADY/TVALID handshake) or be an out-of-band signal (no handshake). TREADY should not be used as a mechanism to transfer opposite direction information from a slave 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)

AXI-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



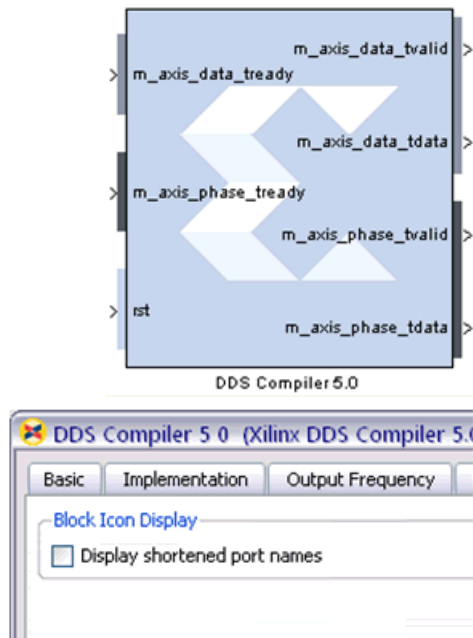
Blocks that proffer AXI4-Stream interfaces have AXI4-Stream channels grouped together and color coded. For example, on the DDS Compiler 5.0 block shown above, the top-most input port **data_tready** and the top two output ports, **data_tvalid** and **data_tdata** belong in the same AXI4-Stream channel. As does **phase_tready**, **phase_tvalid** and **phase_tdata**.

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

Port Name Shortening

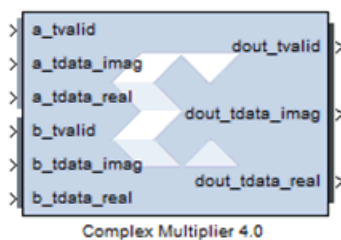
In the example shown below, the AXI4-Stream signal names have been shortened to improve readability on the block. Name shortening is purely cosmetic and when netlisting occurs, the full AXI4-Stream name is used. Name shortening is turned on by default; you can

uncheck the **Display shortened port names** option in the block parameter dialog box to reveal the full name.



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.



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 Interface Generation

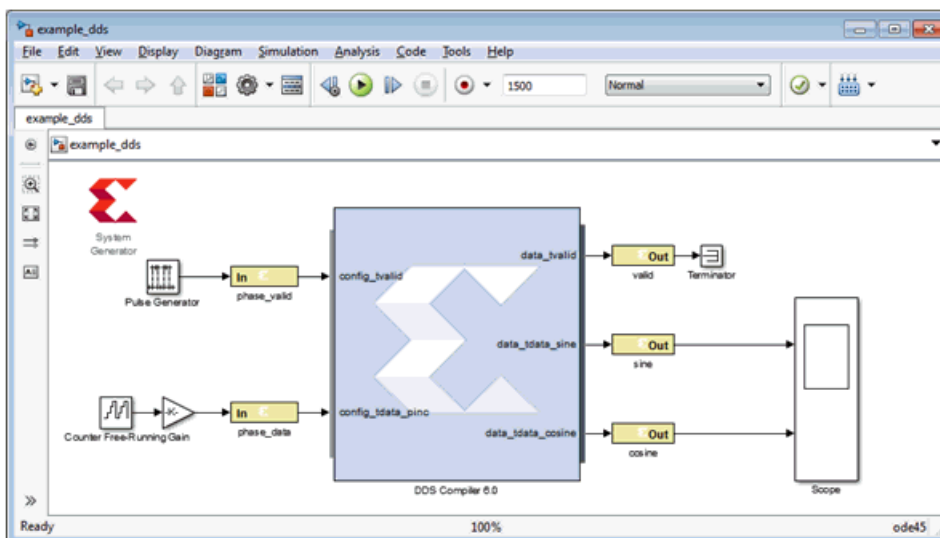
Introduction

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 & 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 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 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.

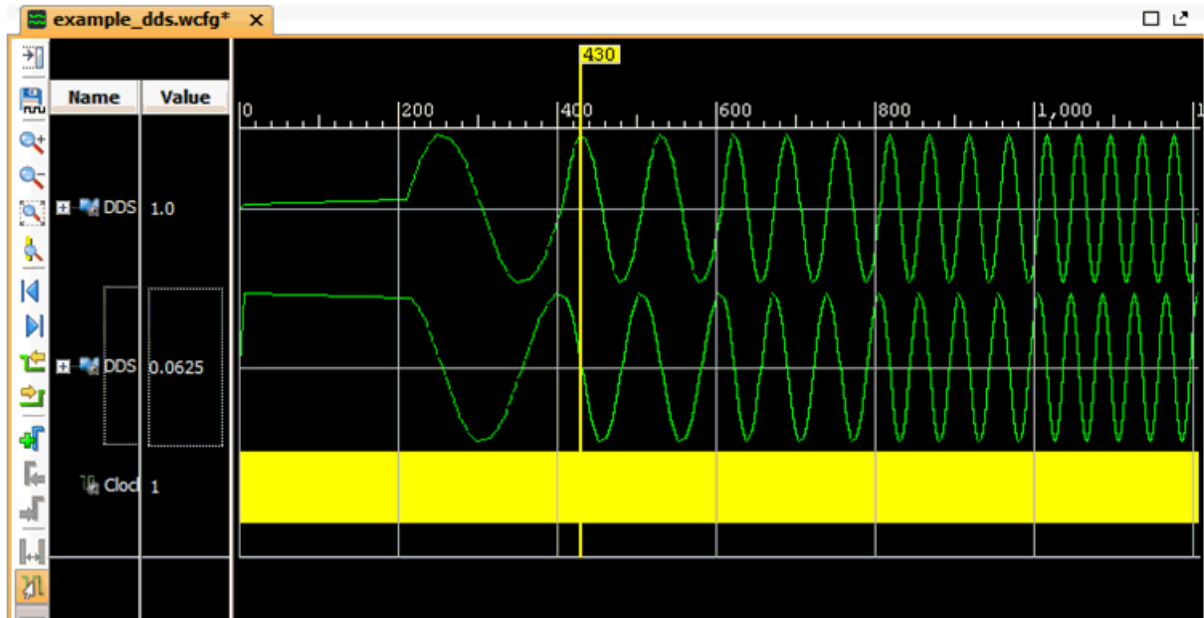
AXI4-Lite Interface Synthesis in System Generator

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.



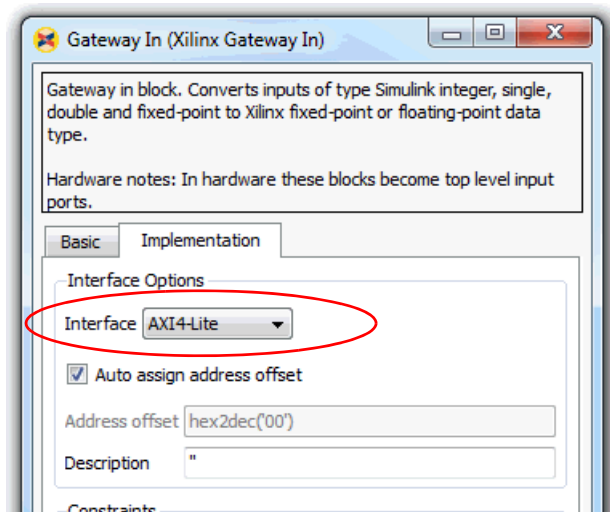
This design contains a DDS Compiler where the two input ports, **phase_valid** and **phase_data** are used to control the output frequency.

The simulation results of this design are shown below which indicate that the output frequency is increasing over time.



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.



As you can see, the Interface is specified as a slave AXI4-Lite Interface on System Generator for DSP design, 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.

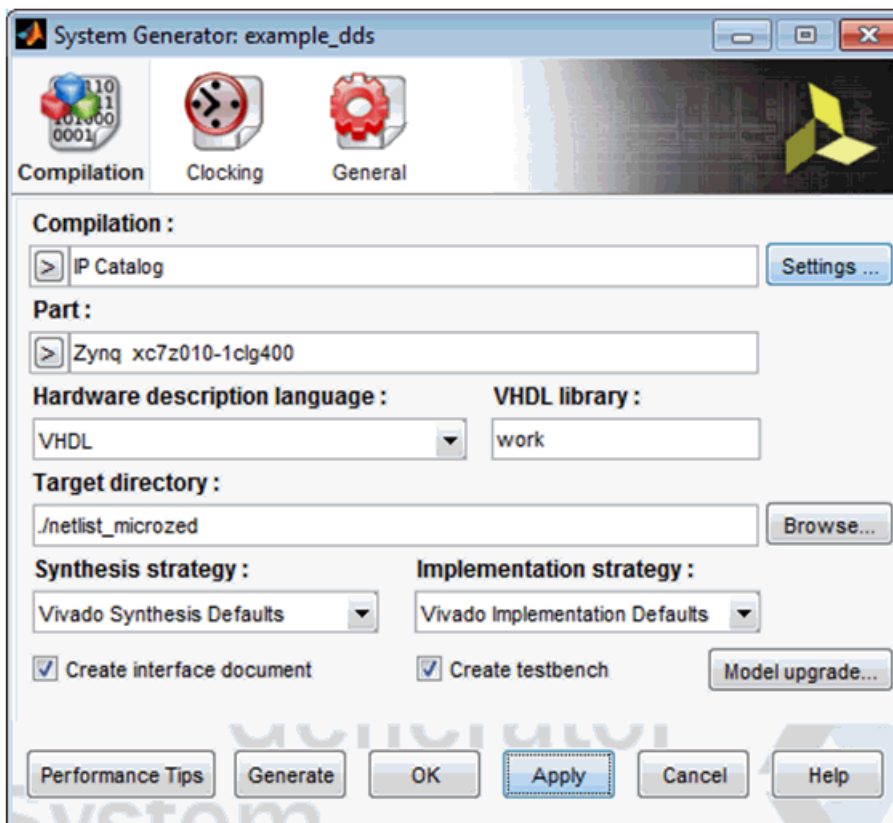
Address offset (Disabled): This option is only enabled if **Auto assign address offset** is unchecked. It allows the user the ability to manually override of Address Offset.

Description: The information entered here is captured in "Interface Documentation" that is automatically created when the design is exported for inclusion into the Vivado IP catalog.

The other Gateways in the design are also configured in a similar fashion.

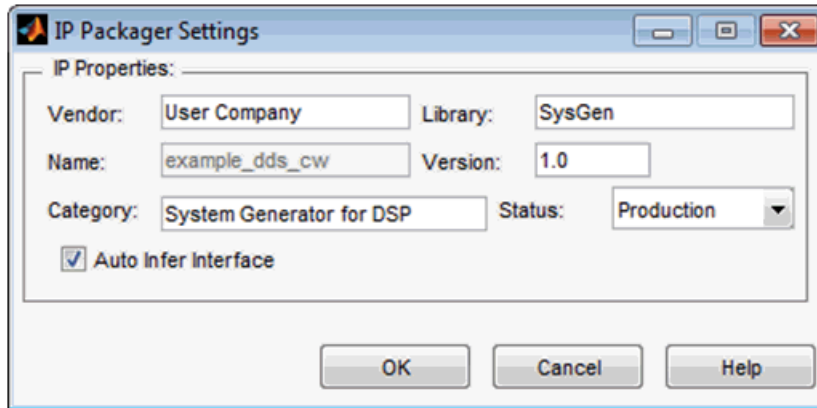
Packaging the Design for Use in Vivado IP Integrator

Now that verification in System Generator is completed, the design can be packaged for use in IP Integrator.



The System Generator block must first be configured to a Compilation target of **IP Catalog**. 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 Avnet MicroZed 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.



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

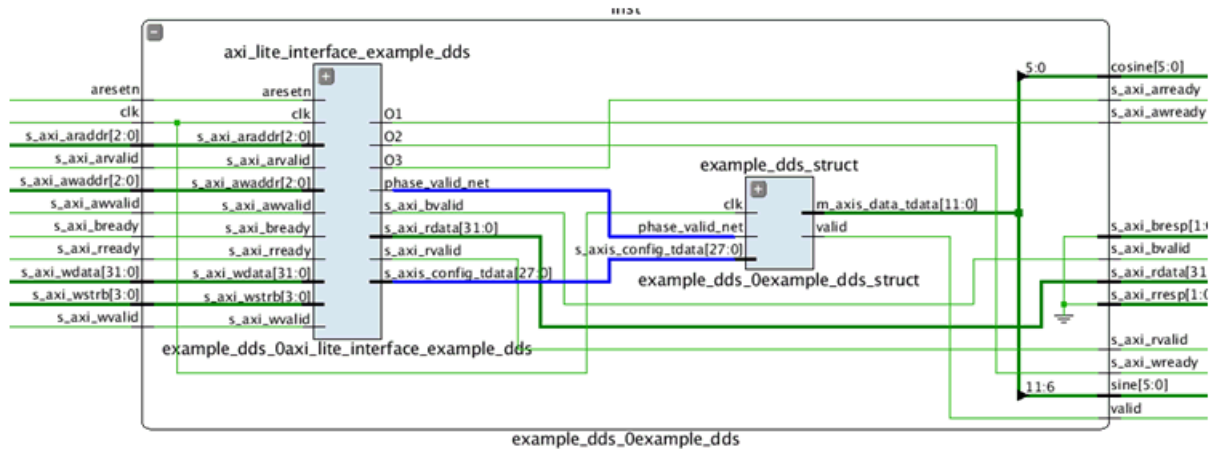
Description of the Generated Results

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

1. **<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.
2. **<target directory>/ip_catalog** : this directory contains an example Vivado IDE project called example_dds.xpr

Mapping to a Single AXI4-Lite Interface

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



As you can see in the diagram, a module called **axi_lite_interface_example_dds** is inserted into the design RTL and drives the phase_valid and phase_data 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 phase_valid or phase_data ports are driven based on the address obtained from the processor.

The number of bits required for addressing (s_axi_araddr & 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.

Address Generation

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

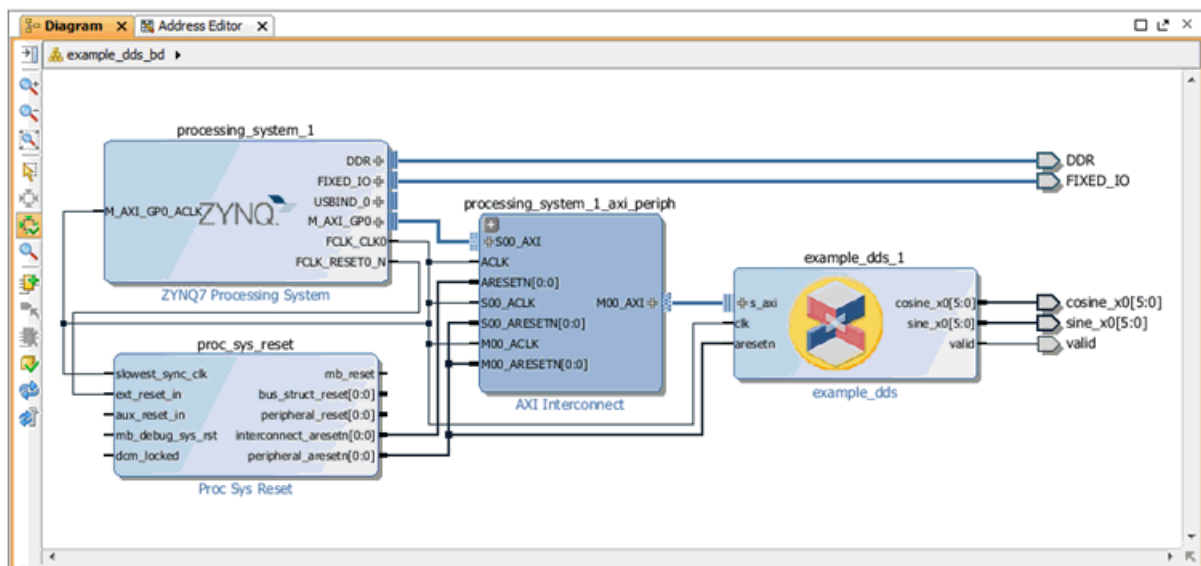
1. Each AXI4-Lite gateway is associated with a unique address offset that is aligned with a 32-bit word boundary (i.e. will be a multiple of 4)
2. Addressing begins at zero
3. Addressing is incrementally assigned in the lexicographical order of the gateways. In the event two gateways have the same name – 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 DUT (Design Under Test)

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 testbench 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 subsystem as the part selected in this example is a ZynQ part. For all other parts, a MicroBlaze-based subsystem is created.



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.

Additionally, the interface documentation associated with the IP is accessible as well through the block GUI.

To access this documentation, you double click on the System Generator IP and click on the **Documentation** button in the GUI. This opens the documentation as shown below



When you scroll to the bottom of this documentation, you'll find a section called Memory Map that is of particular interest in this flow.

Memory Map

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

Name	Type	Address Offset	Description
phase_valid	Bool	00000000	Phase Valid Port That Must Be Asserted
phase_data	UFix_28_28	00000004	Phase Configuration Port

Table 4. Memory Map

The "Description" field in this Memory Map table correlates well with the string entered in the corresponding Gateway In block.

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 SDK 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.
 *
 * @paramInstancePtr is the <Gateway In id> instance to operate on.
 * @paramData is value to be written to gateway <Gateway In id>.
 *
 * @returnNone.
 *
 * @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.
 *
 * @paramInstancePtr is the phase_valid instance to operate on.
 *
 * @returnu32
 *
 * @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.

Using Hardware Co-Simulation

Introduction

System Generator provides hardware co-simulation, making it possible to incorporate a design running in an FPGA directly into a Simulink simulation. "Hardware Co-Simulation" compilation targets automatically create a bitstream and associate it to a block. When the design is simulated in Simulink, results for the compiled portion are calculated in hardware. This allows the compiled portion to be tested in actual hardware and can speed up simulation dramatically.

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 of this subject, refer to the topic [M-Code Access to Hardware Co-Simulation](#) in the section Programmatic Access.

Installing Your Hardware Board

The first step in performing hardware co-simulation is to install and setup your hardware board. The following topics provide specific installation and setup instructions for Xilinx supported boards:

JTAG-Based Hardware Co-Simulation

[Installing a KC705 Board for JTAG Hardware Co-Simulation](#)

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, provided it meets the requirements of the underlying hardware board. This model must include a System Generator token; this block defines how the model should be compiled into hardware. The first step in the flow is to open the System Generator token dialog box and select a compilation type under **Compilation**.

For information on how to use the System Generator token, see [Compiling and Simulating Using the System Generator Token](#).

Choosing a Compilation Target

You may choose the hardware co-simulation board by selecting an appropriate compilation type in the System Generator token dialog box. Hardware co-simulation targets are organized under the **Hardware Co-Simulation** submenu in the **Compilation** dialog box field.

When a compilation target is selected, the fields on the System Generator token dialog box are automatically configured with settings appropriate for the selected compilation target. System Generator remembers the dialog box settings for each compilation target. These settings are saved when a new target is selected, and restored when the target is recalled.

Invoking the Code Generator

The code generator is invoked by pressing the **Generate** button in the System Generator token dialog box.

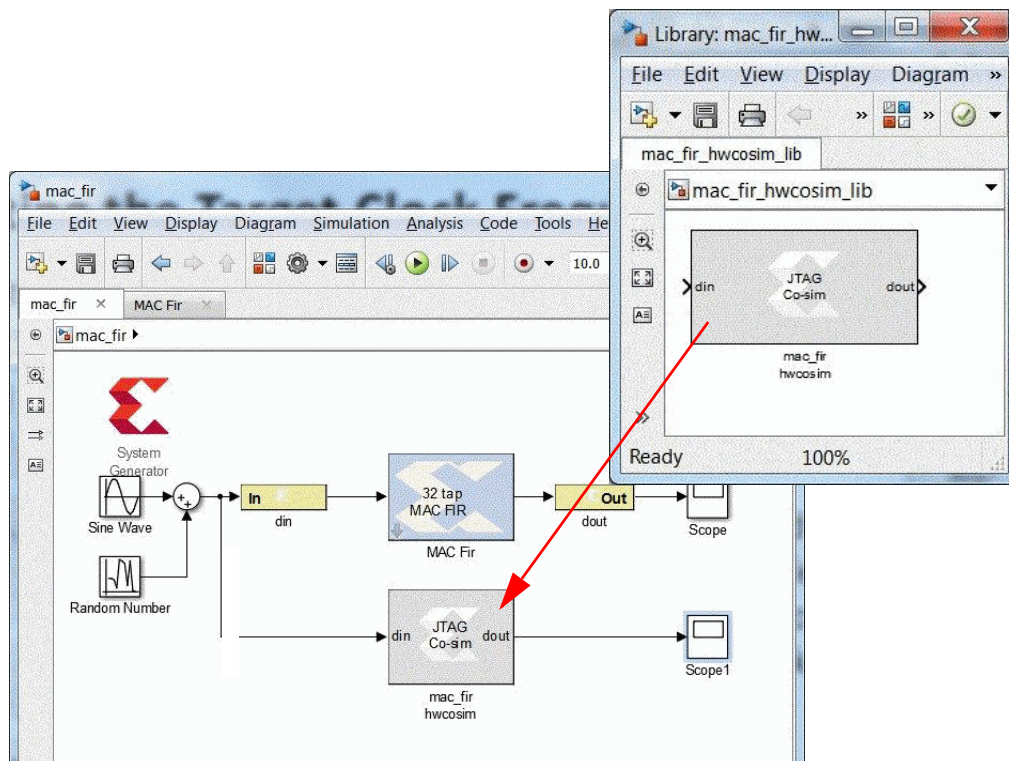
The code generator produces a FPGA configuration bitstream for your design that is suitable for hardware co-simulation. System Generator not only generates the HDL and netlist files for your model during the compilation process, but it also runs the downstream tools necessary to produce an FPGA configuration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

Note: A status dialog box will appear after you press the **Generate** button. During compilation, the status box provides a **Cancel** and **Show Details** button. Pressing the **Cancel** button will stop compilation. Pressing the **Show Details** button exposes details about each phase of compilation as it is run. It is possible to hide the compilation details by pressing the **Hide Details** button on the status dialog box.

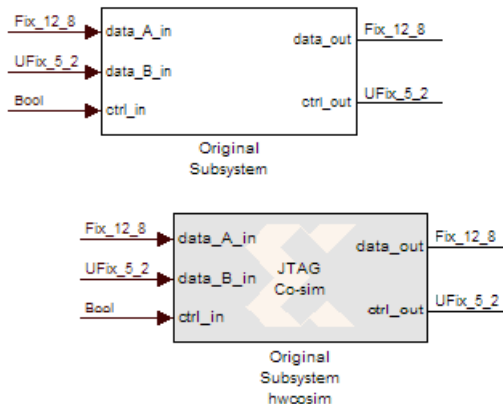
The configuration bitstream contains the hardware associated with your model, and also contains additional interfacing logic that allows System Generator to communicate with your design using a physical interface between the board and the PC. This logic includes a memory map interface over which System 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.

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. A Simulink library is also created in order 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 as you would other Simulink and System Generator 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.



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).

Hardware Co-Simulation Clocking

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 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 a general rule of thumb, 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 under the **Clocking** etch 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).

Installing a KC705 Board for JTAG Hardware Co-Simulation

The following procedure describes how to install and setup the hardware and software required to run JTAG Hardware Co-Simulation on an KC705 board.

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

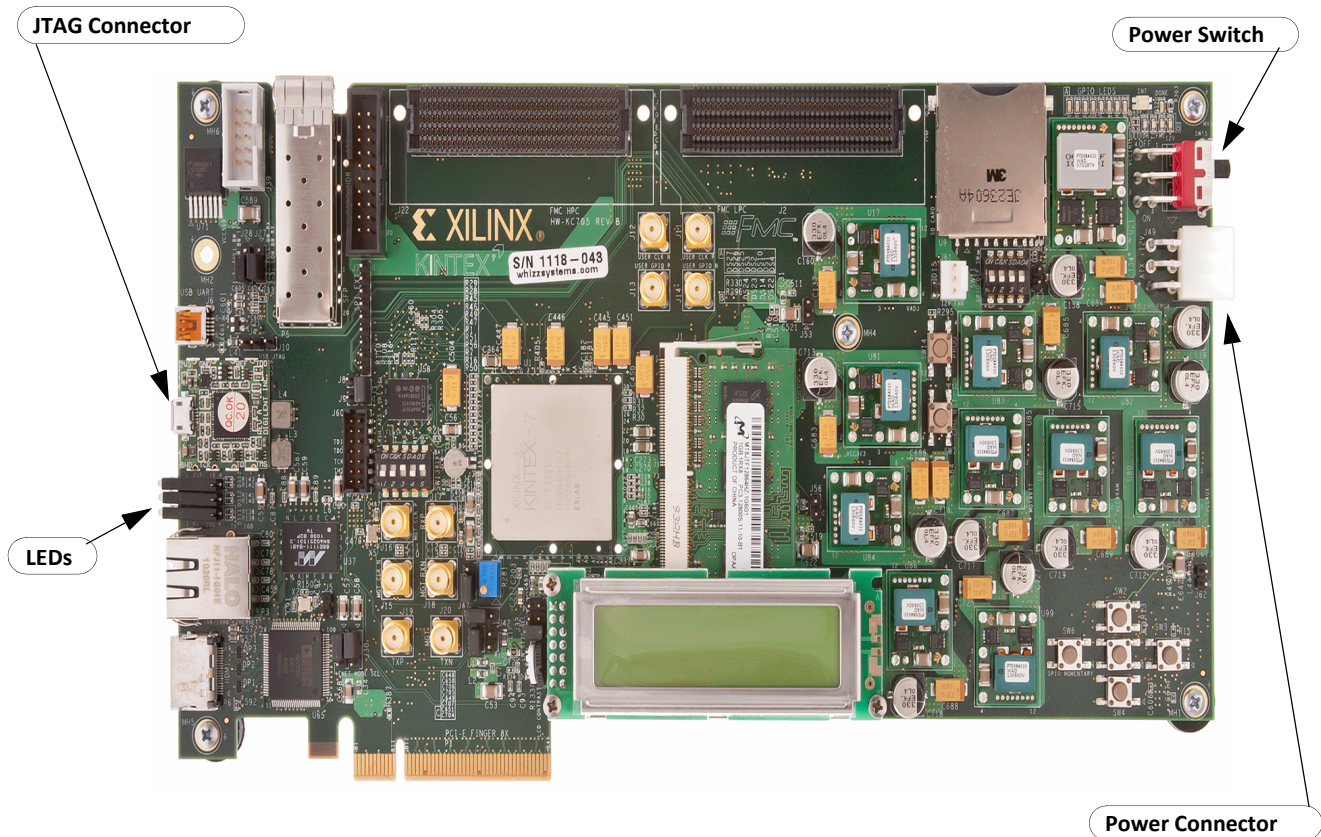
Install Vivado Design Suite Software on the Host PC

Install the Xilinx IVivado™ Design Suite software in the Host PC as described in the document:

Xilinx Design Tools: Installation and Licensing Guide

Setup the KC705 Board

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



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 small end of the Micro USB-JTAG cable to the JTAG socket.
4. Connect the large end of the Micro USB-JTAG cable to a USB socket on your PC.
5. 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.
6. Turn the KC705 board Power switch **ON**.

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, Verilog, and EDIF 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 wires the imported module and associated files into the surrounding netlist.

Table 6-1:

The Black Box Interface	
Black Box HDL Requirements and Restrictions	Details the requirements and restrictions for VHDL, Verilog, and EDIF associated with black boxes.
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 VHDL 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). Although a black box may have more than one clock port, a single clock source is used to drive each clock port. Only the clock enable rates differ.
- 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.

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. It then 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 topic [Black Box Configuration M-Function](#).

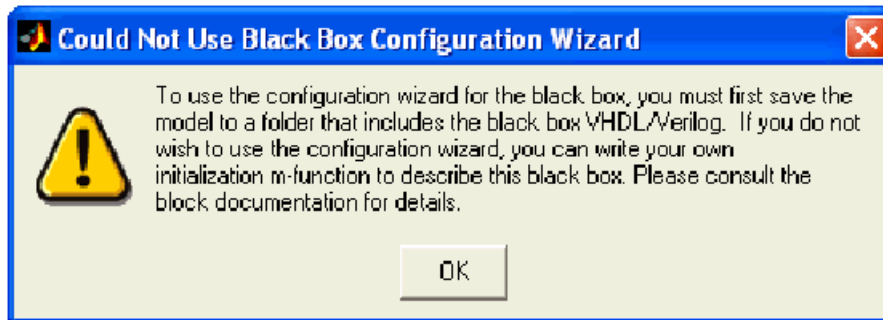
Using the Configuration Wizard

The Black Box Configuration Wizard opens automatically when a new black box block is added to a model.

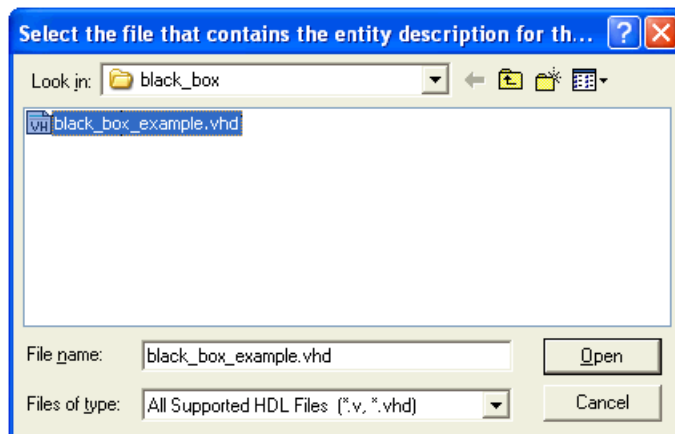
Note: Before running the Configuration Wizard, ensure the VHDL or Verilog you are importing meets the specified [Black Box HDL Requirements and Restrictions](#).

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. This means, in particular, that the model must be saved to same directory.

Note: The wizard only searches for .vhd and .v files in the same directory as the .mdl file. 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:



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:



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.

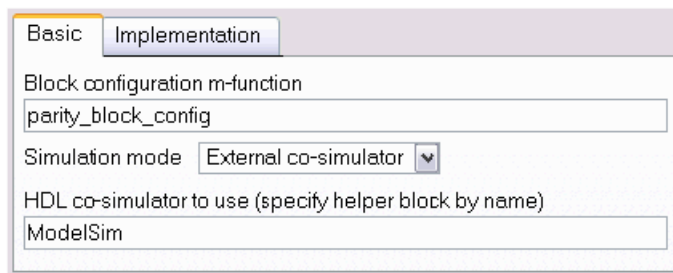
- The Configuration Wizard only knows about the top-level entity that is being imported. There are typically other files that go along with this entity. These files must be added manually in the configuration M-function by invoking the `addFile` method for each additional file.
- The Configuration Wizard creates a single-rate black box. This means that 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.

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. More specifically, 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;
- 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 black box parameters dialog box (`parity_block_config.m` in the example shown below).



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 Block Ports

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
```

Bi-directional 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

Bi-directional 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 since 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, `setRate`, which 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 of 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, 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. Since the Simulink system period is 2 sec, this means the Simulink sample rate of the port is $3 \times 2 = 6$ sec.

Note: If your port is a non-sampled constant, you may define it as so 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.

Note: 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 or 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 allow 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 your System Generator model.

```
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:

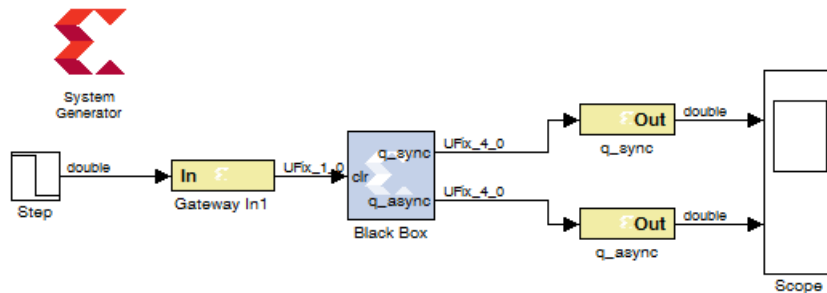
```

% Add additional 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.



The next step is to double-click on the System Generator token and click on 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 on 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, which allows you to specify an error message that is reported to the user. The string parameter passed to setError is the error message that is seen by user.

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. 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 tells the name the port should have.
addSimulinkOutport(pname)	Adds an output port to the black box. pname tells 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 bi-directional port to the black box. pname specifies the name the port should have. Bi- directional 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.
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.
output(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.

Method	Description
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 (respectively, Verilog) file names should end in .vhd (resp., .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 device corresponding to the Blackbox.
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. This method can, for example, 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.
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, e.g., UFix_<n>_, Fix_<n>_, 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>_, Fix_<n>_, 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: <pre>ap_return_port = this_block.port('ap_return'); ap_return_port.setType('XFloat_30_2');</pre>
setWidth(w)	Sets the width of this port to w.

Method	Description
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
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).

HDL Co-Simulation

Introduction

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 testbench), 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.

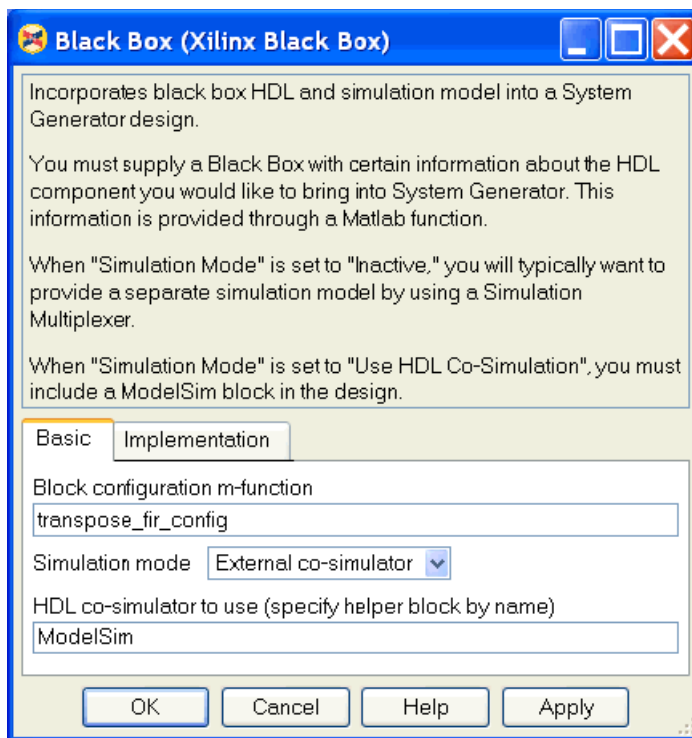
ModelSim Simulator

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.



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.



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. I.e., 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. This is transparent to the user. It does mean, however, that 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 just selecting *Vivado Simulator* as the option for *Simulation mode* on each black box.

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

System Generator uses the HDL Netlist compilation type as the default generation target.

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.

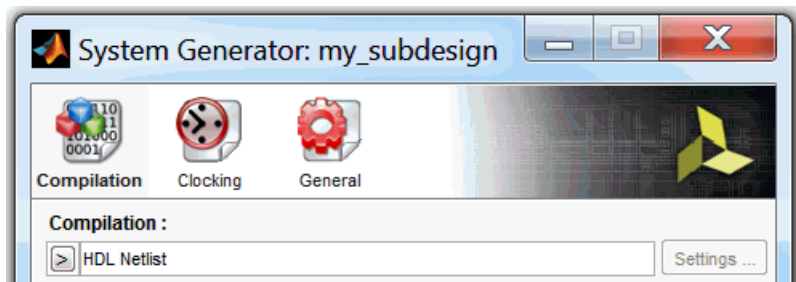
Synthesized Checkpoint Compilation

Describes how to generate a synthesized checkpoint file (synth_1.dcp) that can be used in a Vivado IDE project.

HDL Netlist Compilation

System Generator uses the **HDL Netlist** compilation type as the default generation target. More details regarding the HDL Netlist compilation flow can be found in the sub-topic titled [Compilation Results](#).

As shown below, you may select **HDL netlist** compilation by left-clicking the **Compilation** submenu control on the System Generator token dialog box, and select the **HDL Netlist** target.



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 [Using Hardware Co-Simulation](#).

You may select a hardware co-simulation target by left-clicking the **Compilation** submenu control on the System Generator dialog box, and selecting the desired hardware co-simulation platform. The list of available co-simulation platforms depends on which hardware co-simulation plugins are installed on your system.

IP Catalog Compilation

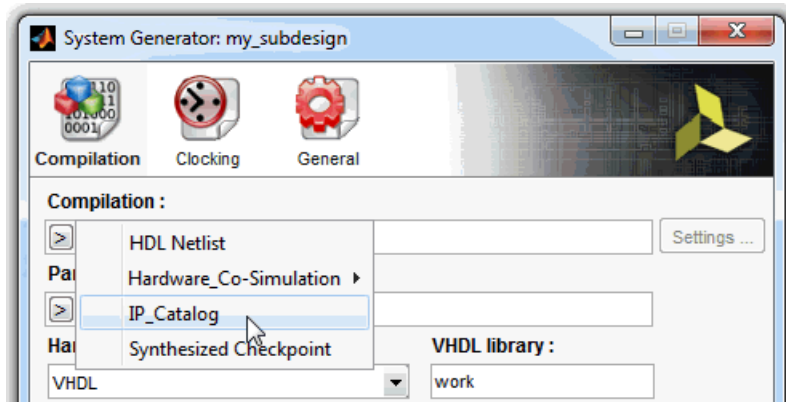
The IP Packager compilation target allows you to package your System Generator design into an IP module that can be included in the Vivado IP catalog. From there, the generated IP can be instantiated into another Vivado user design as a submodule.

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**.

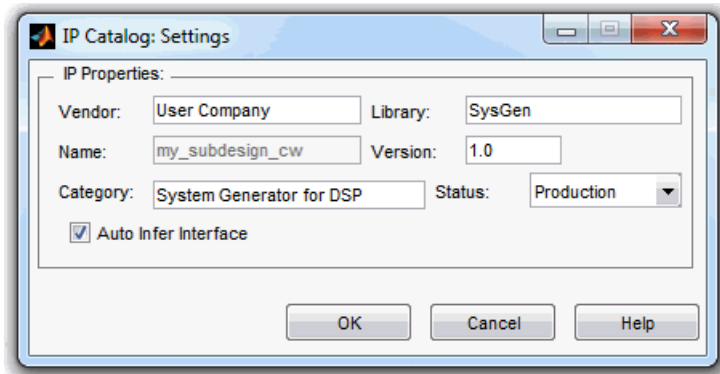
The IP Catalog Flow

In a System Generator design, double click on System Generator token.

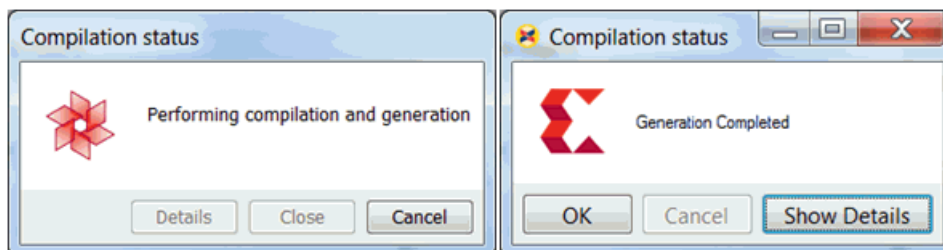
As shown below, under **Compilation:**, click on the > button, then select **IP Catalog**.



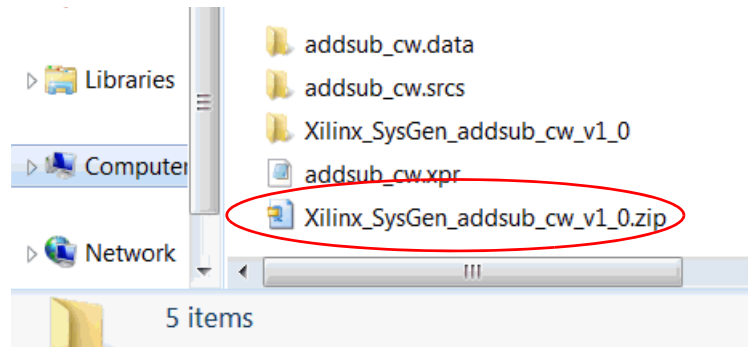
The **Settings** button activates and when you click on it, a dialog box appears as shown below that allows you to enter information about the module that will appear in the Vivado IP Catalog.



The **Target directory** field allows you to specify the location of the generated files. 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 on **Show Details**, to get more detailed information.

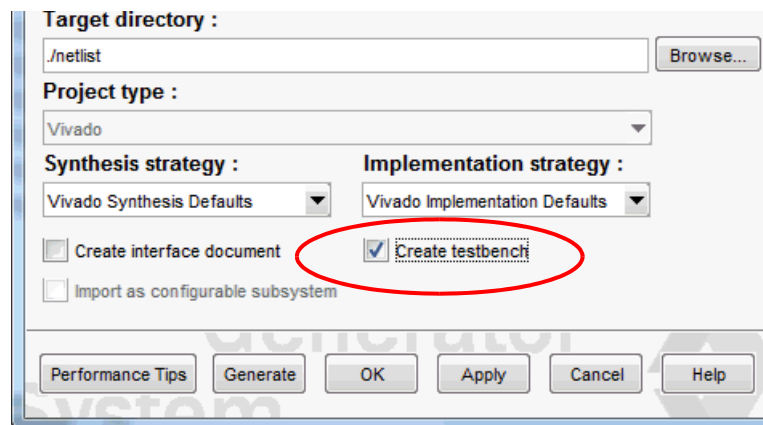


If you navigate to the specified Target directory, you'll find a folder named **ip_catalog**. This folder contains all the necessary files to form an IP from your System Generator design. The ZIP file, circled below, contains all the files required to include the System Generator design as IP in the Vivado IP catalog.



Including a Testbench with the IP Module

In order to verify the functionality of the newly generated IP, it is important to include a testbench. As shown below, if you check **Create testbench**, a test bench will automatically be created when you click the **Generate** button.



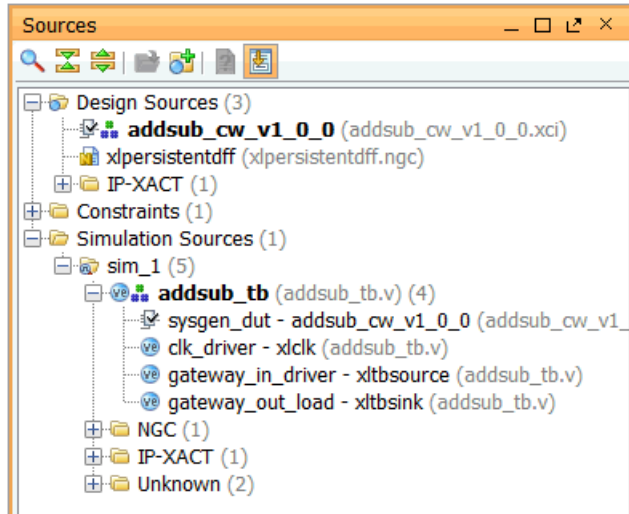
As shown below, when you include a testbench, you can verify the IP functionality by adding three more steps to the flow.

Step 1: Add the new IP to the Vivado IP catalog,

Step 2: Create a new Vivado IDE project and add the IP as the top-level source

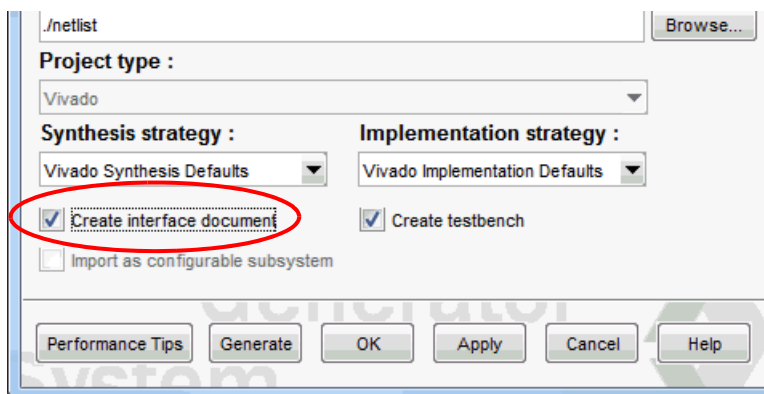
Step 3: Run simulation, synthesis and implementation to verify the functionality of the generated IP.

The following figure shows an open Vivado IDE project with the newly created IP as the top-level source.



Adding an Interface Document to the IP Module

As shown below, if you check **Create interface document**, then press **Generate**, System Generator will generate an interface document for the IP and package this HTML document with the IP.



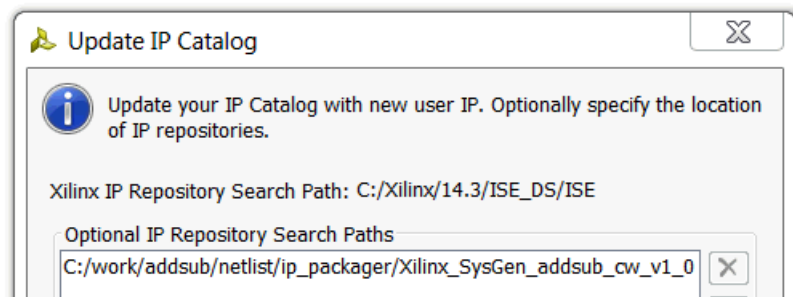
You can find a new folder **documentation** under the **netlist** folder. When you right click on the new IP in Vivado, and click **Data sheet**, one HTML file will be opened with interface information about this IP.

Adding the Generated IP to the Vivado IP Catalog

In order to use the generated IP 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 will only be accessible in this project. For each new project where you will use this IP, you need to perform the same steps.

Second, select **IP Catalog** in the “Project Manager” and right click on an empty area in IP Catalog window. Select **Update IP Catalog** and add the directory the contains your new IP.

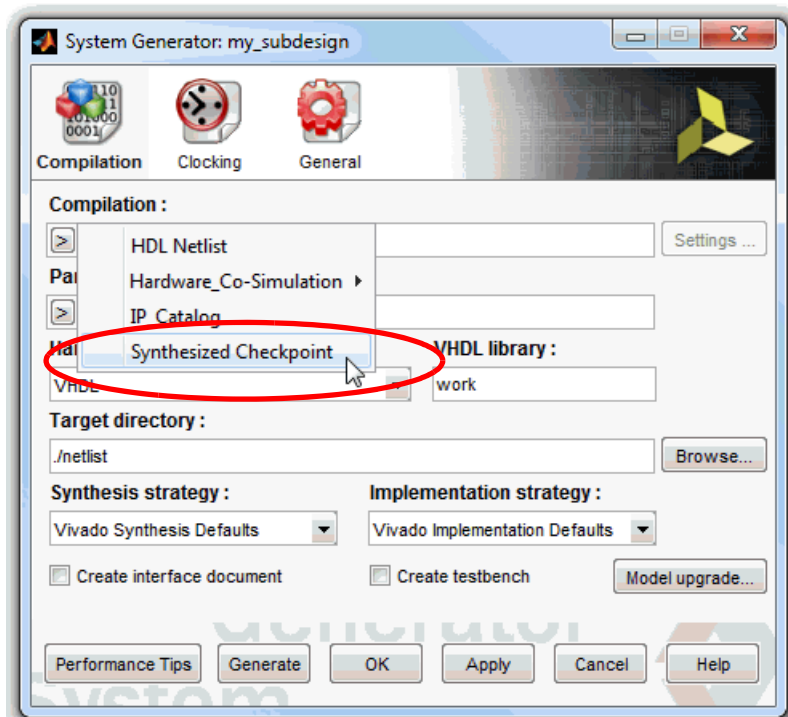


Once the IP is added to the IP Catalog, you can include it in larger designs just as you might with any other IP in the IP catalog.

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.

As shown in the figure below, when you select the Synthesized Checkpoint compilation target, a synthesized checkpoint target file named synth_1.dcp is created and placed in the Target directory.



This synth_1.dcp file can then be used in any Vivado IDE project.

Creating Your Own Custom Compilation Target

System Generator provides a custom compilation infrastructure that allows you to create your own custom compilation target. In addition to generating HDL from your System Generator design, you can 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 titled [Creating Custom Compilation Targets](#).

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 create a compilation target plug-in that automates steps both before and after the Vivado 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.



Creating a New Compilation Target

The following text outlines the general procedure for creating a new compilation target. Specific examples of creating targets follow this description.

Running the Helper Function

You 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')`



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 will act the same as the **HDL Netlist** compilation target. The content of the Impl.m file is shown in the following figure.

```

Editor - U:\demo\Impl\@Impl\Impl.m
Impl.m x
1  classdef Impl < xilinx_compilation
2
3
4  methods
5
6      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
7      % Define how the sysgen token looks.
8      %
9      % Enabling only Verilog for your compilation target can be done
10     % e.g. obj.hdl = 'Verilog';
11     %
12     % Allowing only a particular Implementation Strategy for your
13     % compilation target can be done as follows:
14     % e.g. obj.impl_strategy = 'Flow_Quick';
15     %
16     % See the documentation for more details.
17     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
18     function setup_sysgen_token(obj)
19         obj.target_name = class(obj);
20
21     end
22

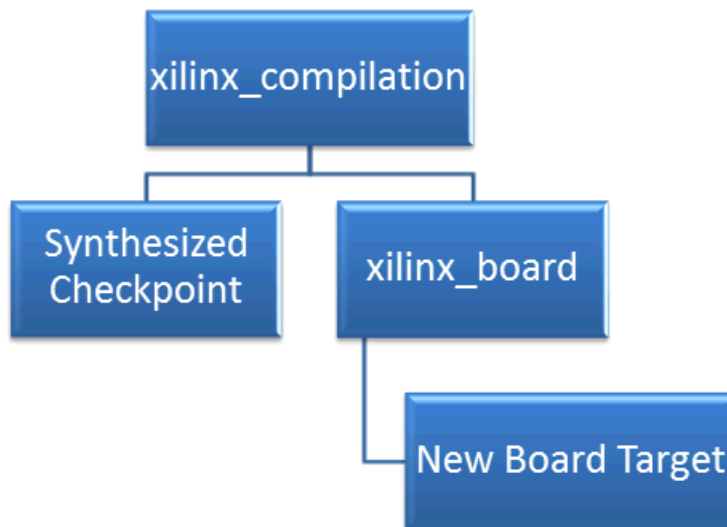
```

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.

Creating a New Board Target

You can support a new development board in System Generator using this interface.



This requires that the board information be present in the Vivado IDE data section and that it is supported in the Vivado IDE. Consider the syntax for the following `addBoard` command:

```
xilinx.environment.addBoard(target_name, directory_name, board_xml_dir)
```

When you enter the following specific command, several things happen.

```
xilinx.environment.addBoard('Zedboard', './',  
'C:\Xilinx\Vivado\2013.3\data\boards\zynq\ZED\revD')
```

1. A folder named `Zedboard/@Zedboard` is created in the current directory. Class `Zedboard` is derived from class `xilinx_board`, which is derived from the `xilinx_compilation` class.
2. Class `Zedboard` does most of the XML Parsing and makes sure that hardware co-simulation is run and the `hwcosim` block is created.

The class created by this function can be used out of the box (provided the XML file is present and correct).

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 are required to add the path which 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 need to 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

Removing the custom compilation target is done by removing the path to the target from the MATLAB Search Path.

Base Class Properties and APIs

The Base class *xilinx_compilation* 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 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 API's 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. 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. 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

Users 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 the Vivado IDE project is created. 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 which describes the port interface will be 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 will be 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. This command can be used to create a bitstream once project creation occurs. The Tcl command can also be used to source a particular Tcl file. The commands will be executed in the order in which they are received.

add_file(string)

This function adds user-defined files to the Vivado IDE project. This 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:

design_info x design_info.ports x design_info.ports.gateway_in x

design_info.ports.gateway_in <1x1 struct>

Field	Value	Min	Max
ArithmeticType	'\Signed'		
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 x design_info.ports x 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 and then enter the following command:

```
xilinx.environment.addCompilationTarget('Impl', 'U:\demo')
```

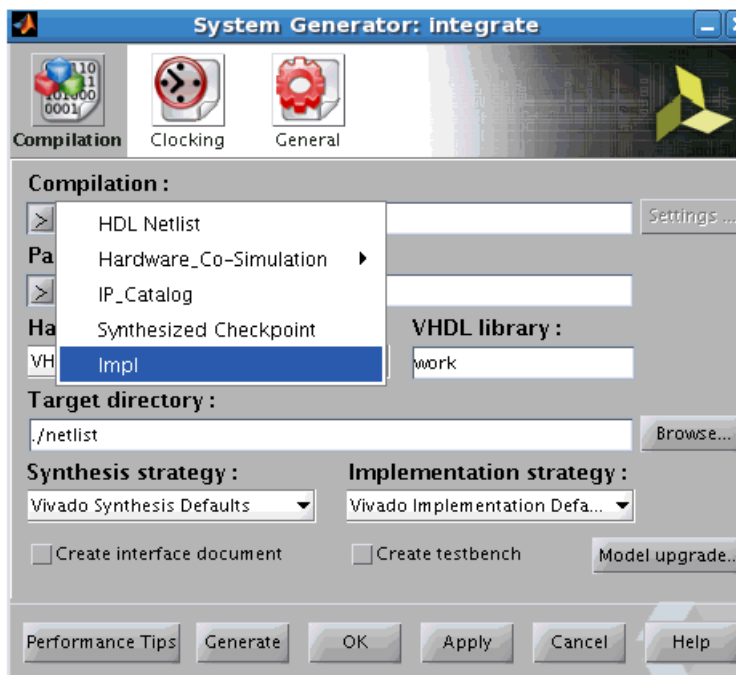
This will provide a template derived class for the users to edit.

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 will now see the compilation target **Impl** on the token as shown below.



5. At this point, selecting **Impl** will 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

```

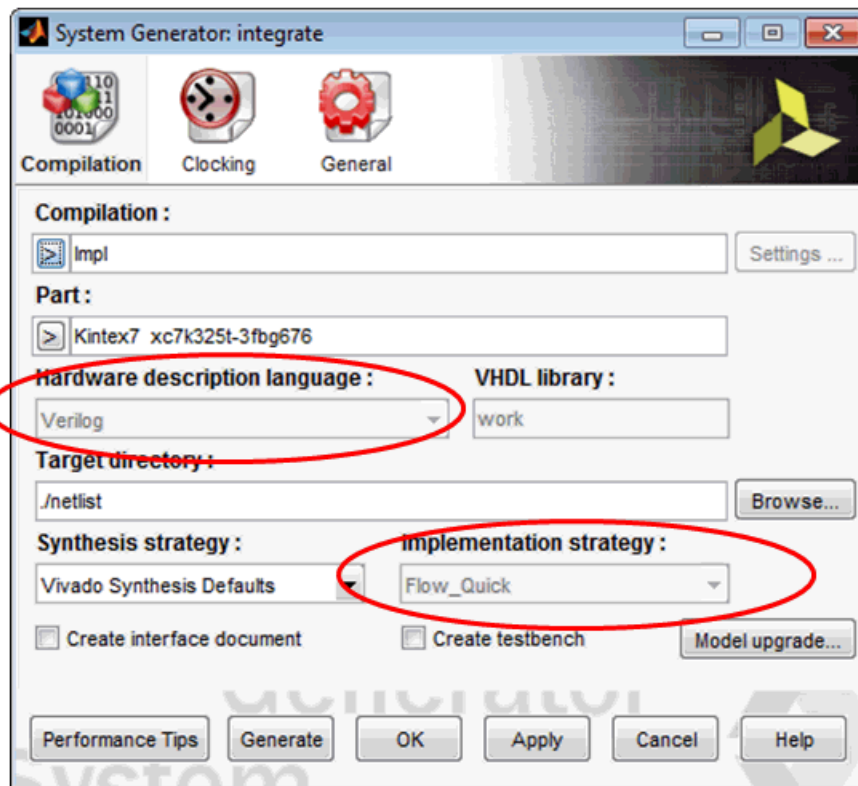
8. In the MATLAB Command Window, you should enter the following command:

xilinx.environment.rehashCompilationTarget

This will ensure 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 will appear as follows:



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 will ensure 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 on **Generate**. Once the process is finished, you can see the implementation results by opening up the Vivado IDE project.

Example 2: Creating a Zedboard Target

1. Open a System Generator design.

2. In the MATLAB Command Window, modify the path as per your machine/installation and then enter the following command:

```
xilinx.environment.addBoard ('Zedboard', 'U:\demo',  
'C:\Xilinx\Vivado\2013.3\data\boards\zynq\ZED\revD')
```

This provides a template derived class for the users to edit. The last field corresponds to the directory which contains the board.xml file.

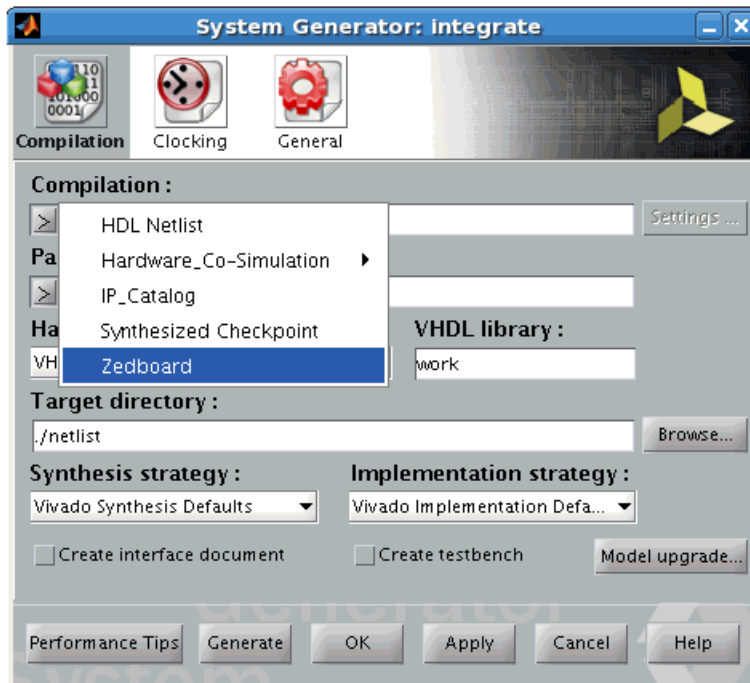
3. In the MATLAB Command Window, enter the following command:

```
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.

- You will now see the compilation target **Zedboard** on the System Generator token as shown below.



- Select **Zedboard**.

Under the Part menu, you will see the only valid available device is selected by default.

- Click **Generate**.

You will see that the Hardware co-Simulation block is generated.

You can now use this Hardware Co-Simulation block in any System Generator design with a Zedboard.

Example 3: Creating a Bitstream Target

- Open a System Generator design.
- 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', '.')
```

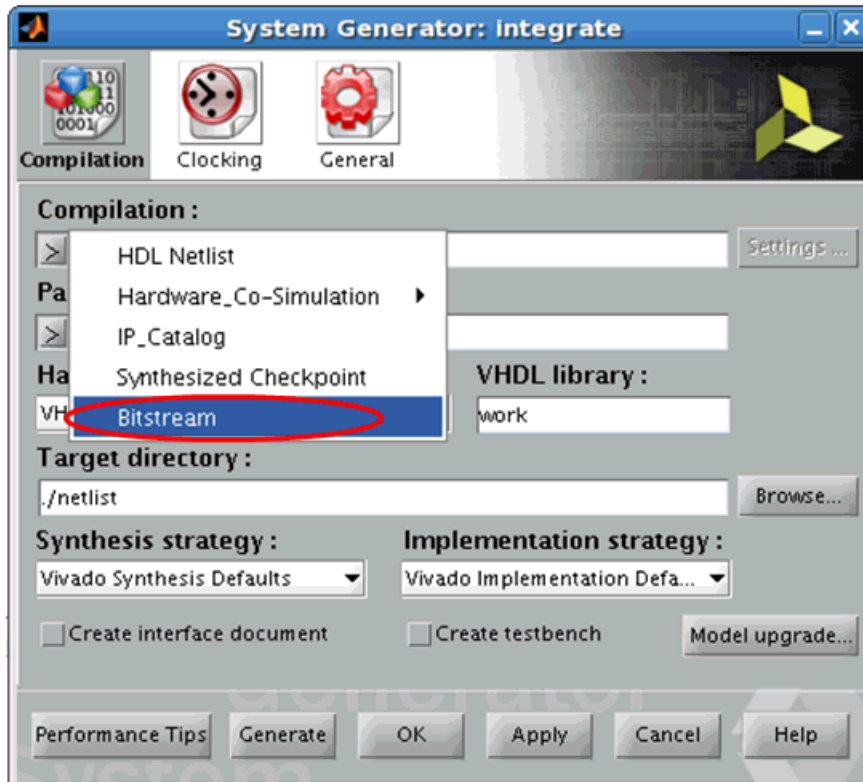
This provides a template derived class for the users to edit. The last field corresponds to the directory which contains the board.xml file.

- In the MATLAB Command Window, enter the following command:

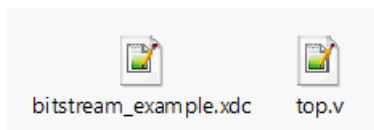
```
xilinx.environment.rehashCompilationTarget
```

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.



6. Open the Bitstream.m created in the './Bitstream/@Bitstream/Bitstream.m'
7. Download the two files below:



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/2013.3/cust_comp_test/bitstream_example.xdc');
add_file(obj, '/group/dspusers-xsj/umangp/rel/2013.3/cust_comp_test/top.v');
obj.top_level_module = 'top';
run_synthesis(obj);
run_implementation(obj);
generate_bitstream(obj);
```

8. 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

- 9. Close and then re-open the System Generator token.
- 10. Select the **Bitstream** compilation target.
- 11. Click the **Generate** button.
- 12. 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

Additional Resources

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see the Xilinx® Support website at:

www.xilinx.com/support.

For a glossary of technical terms used in Xilinx documentation, see:

www.xilinx.com/company/terms.htm.

Solution Centers

See the [Xilinx Solution Centers](#) for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips.

References

These documents provide supplemental material useful with this guide:

1. *Vivado Design Suite Reference Guide: Model-Based DSP Design Using System Generator* ([UG958](#))
2. *Vivado Design Suite Tutorial: Model-Based DSP Design Using System Generator* ([UG948](#))
3. *Vivado Design Suite User Guide: Using the Vivado IDE* ([UG893](#))
4. *Vivado Design Suite Migration Methodology Guide* ([UG911](#))
5. *Vivado Design Suite User Guide: Using Constraints* ([UG903](#))
6. *Vivado Design Suite User Guide: Design Flows Overview* ([UG892](#))
7. *Vivado Design Suite User Guide: Using Tcl Scripting* ([UG894](#))

8. *Vivado Design Suite User Guide: System-Level Design Entry* ([UG895](#))
9. *Vivado Design Suite User Guide: Designing with IP* ([UG896](#))
10. *Vivado® Design Suite User Guide: Release Notes, Installation, and Licensing* ([UG973](#))
11. Vivado Design Suite Video Tutorials (www.xilinx.com/training/vivado/index.htm)
12. Vivado Design Suite Tutorials
(www.xilinx.com/support/documentation/dt_vivado2013-3_tutorials.htm)
13. Vivado Design Suite User Guides
(www.xilinx.com/support/documentation/dt_vivado2013-3_userguides.htm)
14. Vivado Design Suite Reference Guides
(www.xilinx.com/support/documentation/dt_vivado2013-3_referenceguides.htm)
15. Vivado Design Suite Methodology Guides
(www.xilinx.com/support/documentation/dt_vivado2013-3_methodologyguides.htm)
16. *UltraFast™ Design Methodology Guide for the Vivado Design Suite* ([UG949](#))
17. *UltraFast Design Methodology Checklist* ([XTP301](#))
18. Vivado Design Suite Documentation
(www.xilinx.com/support/documentation/dt_vivado2013-3.htm)