

 See all versions  
of this document

# Vitis Model Composer User Guide

UG1483 (v2025.1) May 29, 2025



# Table of Contents

<b>Chapter 1: Overview.....</b>	<b>4</b>
Navigating Content by Design Process.....	4
Introduction.....	5
What's New and Limitations.....	7
Installation.....	7
Post-Installation Tasks.....	10
Supported MATLAB Versions and Operating Systems.....	13
Recommended Display Resolution.....	13
<b>Chapter 2: HDL Library.....</b>	<b>14</b>
Introduction.....	14
Hardware Design Using the HDL Library.....	18
Performing Analysis in Vitis Model Composer.....	110
Using Hardware Co-Simulation.....	123
Importing HDL Modules.....	157
Compilation Types for HDL Library Designs.....	180
Creating Custom Compilation Targets.....	190
AMD Waveform Viewer.....	199
<b>Chapter 3: HLS Library.....</b>	<b>208</b>
Introduction.....	208
Creating a Model Composer Design.....	209
Importing C/C++ Code as Custom Blocks.....	226
Generating Outputs.....	260
Simulating and Verifying Your Design.....	284
Select Target Device or Board.....	288
<b>Chapter 4: AI Engine Library.....</b>	<b>290</b>
Introduction.....	290
Vitis Model Composer for AI Engine Development.....	293
Creating an AI Engine Design using Vitis Model Composer.....	295
AI Engine DSPLib.....	347

Setting Signal Size to Avoid Buffer Overflow.....	349
AI Engine Designs with Feedback Loops.....	353
Simulation and Code Generation.....	354
<b>Chapter 5: Hardware Validation Flow for AI Engines and PL.....</b>	<b>382</b>
Introduction.....	382
High-Level Flow for Generating a Hardware Image.....	382
Setting up the Tool to Generate an Image File for Hardware Validation Flow.....	384
Running the Hardware Flow Outside the MATLAB Environment.....	389
Design Considerations.....	389
<b>Chapter 6: Connecting AI Engine and Non-AI Engine Blocks.....</b>	<b>391</b>
AI Engine/Programmable Logic Integration.....	391
Connecting Source and Sink Blocks.....	407
<b>Appendix A: Model Composer Utilities.....</b>	<b>410</b>
AI Engine Utilities.....	410
HDL Utilities.....	415
<b>Appendix B: Additional Resources and Legal Notices.....</b>	<b>450</b>
Finding Additional Documentation.....	450
Support Resources.....	451
References.....	451
Revision History.....	452
Please Read: Important Legal Notices.....	453

# Overview

---

## Navigating Content by Design Process

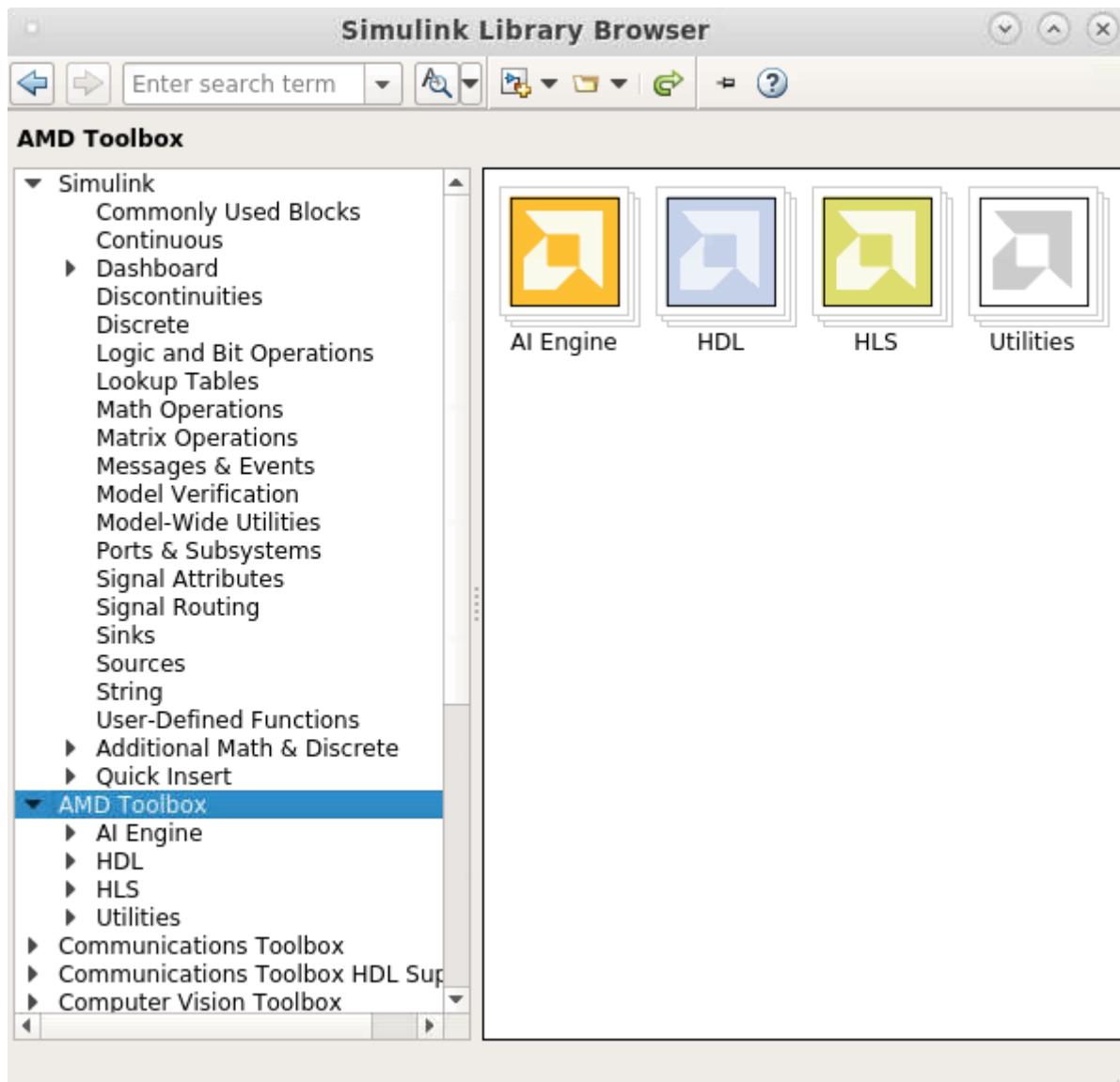
AMD Adaptive Computing documentation is organized around a set of standard design processes to help you find relevant content for your current development task. You can access the AMD Versal™ adaptive SoC design processes on the [Design Hubs](#) page. You can also use the [Design Flow Assistant](#) to better understand the design flows and find content that is specific to your intended design needs. This document covers the following design processes:

- **AI Engine Development:** Creating the AI Engine graph and kernels, library use, simulation debugging and profiling, and algorithm development. Also includes the integration of the PL and AI Engine kernels. Topics in this document that apply to this design process include:
  - [Creating an AI Engine Design using Vitis Model Composer](#)
  - [Simulation and Code Generation](#)
  - [Chapter 6: Connecting AI Engine and Non-AI Engine Blocks](#)
- **Hardware, IP, and Platform Development:** Creating the PL IP blocks for the hardware platform, creating PL kernels, functional simulation, and evaluating the AMD Vivado™ timing, resource use, and power closure. Also involves developing the hardware platform for system integration. Topics in this document that apply to this design process include:
  - [Hardware Design Using the HDL Library](#)
  - [Creating a Model Composer Design](#)
  - [Compilation Types for HDL Library Designs](#)
- **System Integration and Validation:** Integrating and validating the system functional performance, including timing, resource use, and power closure. Topics in this document that apply to this design process include:
  - [Performing Analysis in Vitis Model Composer](#)
  - [Using Hardware Co-Simulation](#)

# Introduction

AMD Vitis™ Model Composer is a model-based design tool that enables rapid design exploration within the Simulink® environment and accelerates the path to production on AMD devices through automatic code generation. This provides a library of performance-optimized blocks for design and implementation of algorithms on AMD devices using HDL, HLS, and AI Engine blocks.

Figure 1: Simulink Library Browser



You can focus on expressing algorithms using blocks from these libraries as well as custom user-imported blocks, without worrying about implementation specifics, and leverage all the capabilities of Simulink's graphical environment for algorithm design, simulation, and verification.

The AI Engine library in Vitis Model Composer contains blocks to import kernels and graphs which can be targeted to the AI Engine portion of AMD Versal™ devices. Also included is a set of complex AI Engine DSP building blocks related to FIR, FFT, DDS, and Mixers. The tool transforms your AI Engine design specification into a top level connected graph to implement AI Engine-based devices.

The HDL library in Vitis Model Composer contains DSP building blocks. These blocks include the basic element blocks such as adders, multipliers, and registers. Also included are a set of complex DSP building blocks such as FFTs, filters, and memories. These blocks leverage the AMD IP core generators to deliver optimized results for the selected device.

The HLS library in Vitis Model Composer offers predefined blocks which includes functional blocks for Math, Linear Algebra, Logic, and Bit-wise operations. The tool transforms your algorithmic specifications to production-quality implementation through automatic optimizations that extend the AMD High Level Synthesis technology.

Vitis Model Composer allows you to co-simulate a heterogeneous system consisting of AI Engines and PL. You can directly use optimized AI Engine, HLS, or HDL blocks from the Simulink Library Browser, or import code as blocks. These blocks can be connected in Simulink to perform functional simulation.

The hardware validation flow in AMD Vitis Model Composer provides a methodology to verify AI Engine and/or PL-based applications on AMD hardware. Vitis Model Composer generates a hardware image that can be run on a board to verify whether the results from hardware match the functional simulation output.

The rest of this document describes information on features and specific blocks related to the:

- **HDL Library:** Refer to [Chapter 2: HDL Library](#)
- **HLS Library:** Refer to [Chapter 3: HLS Library](#)
- **AI Engine Library:** Refer to [Chapter 4: AI Engine Library](#)
- **Hardware Validation Flow:** Refer to [Chapter 5: Hardware Validation Flow for AI Engines and PL](#).
- **Co-Simulation of AI Engine and PL:** Refer to [Chapter 6: Connecting AI Engine and Non-AI Engine Blocks](#).

---

# What's New and Limitations

System Generator - the previous standalone design environment to develop DSP algorithms and generate HDL as an output - is now part of AMD Vitis™ Model Composer. As a result of this product unification, HLS, AI Engine, and System Generator (HDL) libraries in the AMD toolbox have been merged to develop algorithms in a single MATLAB session.

The HDL Library in the AMD Toolbox contains all the library blocks previously contained in the System Generator blockset. Any existing System Generator design can be opened in Vitis Model Composer. Furthermore, you can develop new algorithms using the HDL library similarly to using the System Generator tool.

For information related to what is new for a specific release of Model Composer, refer to [What's New](#).

In addition, while Model Composer is a toolbox built onto the MathWorks Simulink environment, there are certain features of Simulink that are not supported in Model Composer. The following is a list of some of the unsupported features:

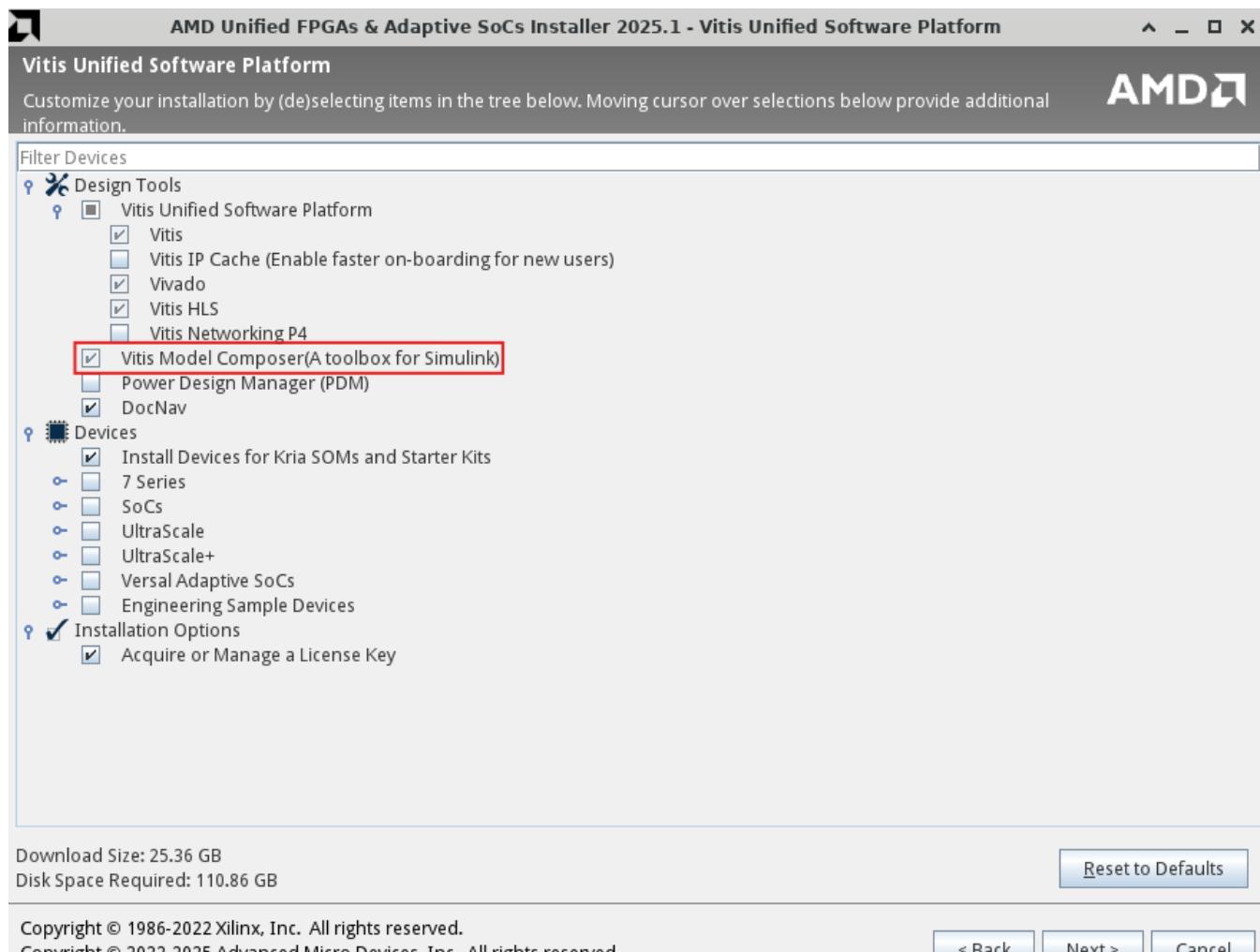
- Simulink Performance Advisor.
  - Model referencing and subsystem referencing.
  - Variant subsystems.
  - Model Composer blocks do not support Simulink fixed-point types and only support AMD fixed-point types.
  - Fixed-point designer does not integrate with Model Composer.
  - Accelerator mode and Rapid Accelerator mode.
- 

# Installation

## Downloading

AMD Vitis™ Model Composer is part of AMD Vivado™ as well as the Vitis software platform, which can be downloaded from the [website](#). The AI Engine library is available only in the Vitis software platform, it is not part of the Vivado installation.

The AMD Unified Installer for FPGAs & Adaptive SoCs automatically installs Vitis Model Composer, irrespective of whether you choose Vitis or Vivado as a product. The following figure shows the Vitis installer window with Vitis Model Composer selected.

**Figure 2: Vitis Installer**

## Launching Vitis Model Composer

You can launch the Vitis Model Composer tool directly from the desktop or from the command line. Double-click the **Vitis Model Composer** icon, or launch it from the Start menu in Windows, or use the following command from the command prompt.

```
model_composer
```



**TIP:** The command-line use of Vitis Model Composer requires that the command shell has been configured as follows. You must change directory to `<install_dir>/<version>/Model_Composer` and run the `settings64.bat` (or `.sh`) file, where `<install_dir>` is the installation folder and `<version>` is the specific version of the tool. MATLAB opens, and the HLS, HDL, and AI Engine libraries and features are overloaded onto this environment.

Vitis Model Composer supports the latest releases of MATLAB. For more information on Supported MATLAB versions and operating systems, refer to [Supported MATLAB Versions and Operating Systems](#). If you have multiple versions of MATLAB installed on your system, the first version found in your PATH is used by the tool. You can edit the PATH to move the preferred version of MATLAB to precede other versions. You can also direct the tool to open a specific version of the tool using the `-matlab` option as follows:

```
model_composer -matlab C:\Progra~1\MATLAB\R2022a
```



**TIP:** When you specify the path to the MATLAB version, do not specify the full path to the executable (`bin/MATLAB`). The string `C:\Progra~1\` is a shortcut to `C:\Program Files\` which eliminates spaces from the command path. The command-line use of the Vitis Model Composer tool requires that the command shell has been properly configured as previously discussed.

After launching the tool, you will see the following in the MATLAB command window. Use these links to access the documentation and product examples.

**Figure 3: Documentation and Examples Links**

Vitis Model Composer: [User Guide](#) [Examples and Tutorials](#)

# Post-Installation Tasks

## Configure OS Libraries (Ubuntu and RHEL9 Only)

Vitis Model Composer requires symbolic links to be created on the Ubuntu and Red Hat 9 operating systems. After installation, these symbolic links can be created by running the following command:

```
model_composer -configlib
```

This command requires `sudo` access.

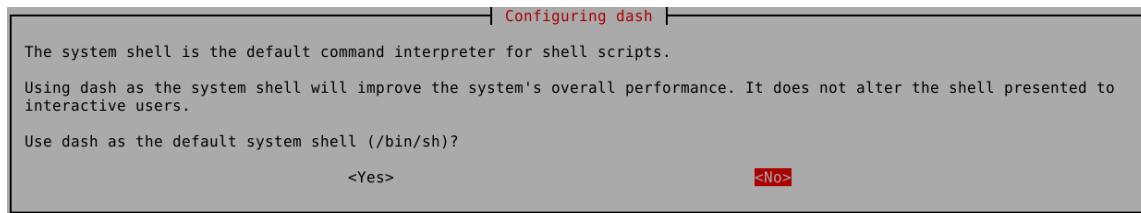
## Configure Bash Shell (Ubuntu Only)

Ubuntu 20 and 22 by default come with dash shell. To avoid any issue that you might encounter while running the downstream AI Engine flows, it is recommended to change the shell from dash to bash using the following sudo command:

```
sudo dpkg-reconfigure dash
```

When prompted whether to use dash as the default system shell, select **No**.

*Figure 4: Configure Bash Shell*



## Compiling AMD HDL Libraries

The AMD tool that compiles libraries for use in Questa SE is named `compile_simlib`.

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

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

## Managing the Model Composer Block Cache

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

## Specifying Board Support in Model Composer

When Vitis Model Composer is installed on your system as part of a Vivado Design Suite installation, Model Composer will have access to any AMD development boards installed with Vivado.

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

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

After you have configured the board for use with the Vivado Design Suite, you must do the following to make the board available in Vitis Model Composer:

1. Run the following command in the MATLAB Command Window:

```
[status, boardTable, partTable] = xmchubReloadDeviceInfo;
```

2. Check that the variable `boardTable` in the MATLAB workspace contains information about your board:

```
boardTable
```

```

>> [status, boardTable, partTable] = xmcHubPclLoadDeviceInfo;
***** Vivado v2022.2 (64-bit)
***** SR Build 3964533 on Wed Oct 5 19:12:30 MDT 2022
***** IP Build 3963802 on Wed Oct 5 23:08:30 MDT 2022
*** Copyright 1986-2022 Xilinx, Inc. All Rights Reserved.

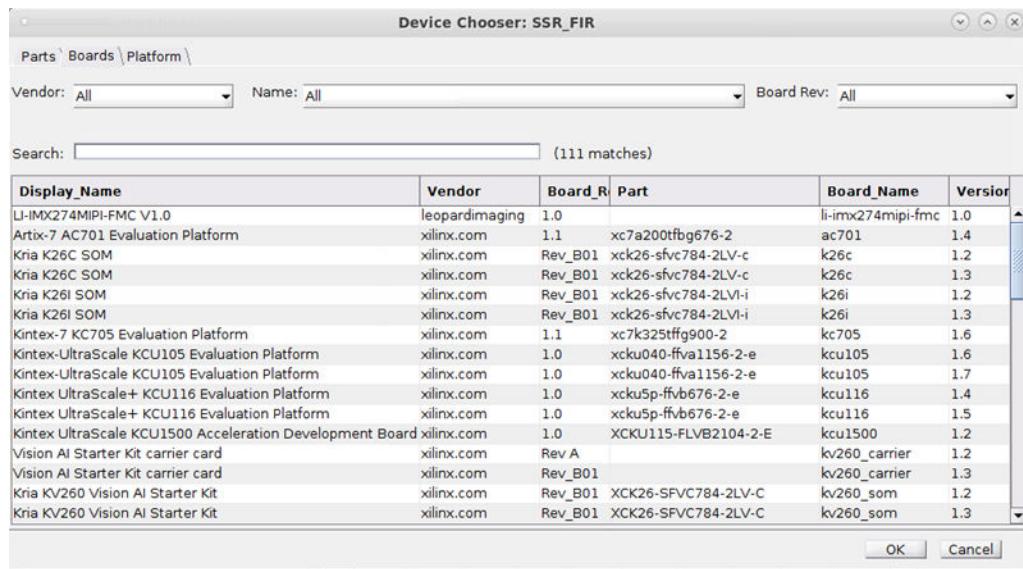
Sourcing tcl script          /Xilinx/Vivado/Vivado_init.tcl'
42 Beta devices matching pattern found, 42 enabled.
enable_beta_device: Time (s): cpu = 00:00:10 ; elapsed = 00:00:11 . Memory (MB): peak = 2183.500 ; gain = 131.508 ; free physical = 46124 ; free virtual = 62330
source loadDevices.tcl
# set_param board.repoPaths      /Vivado/2022.2/data/xhub/boards*
# load librdi_dsp_tclTasks.so
# dsp_write_board_objects -dirpath /model_composer/data -csv
# dsp_write_board_objects: Time (s): cpu = 00:00:06 ; elapsed = 00:00:06 . Memory (MB): peak = 2206.051 ; gain = 102.008 ; free physical = 46021 ; free virtual = 62228
# dsp_write_board_objects -dirpath /model_composer/data
# exit
INFO: [Common 17-206] Exiting Vivado at Thu Oct 6 15:52:05 2022...
>> boardTable
boardTable =
71xtable

| Display_Name                          | Vendor             | Board_Rev   | Part                     | Board_Name            | Version |
|---------------------------------------|--------------------|-------------|--------------------------|-----------------------|---------|
| {'LI-IMX274MIPi-FMC V1.0'}            | {'leopardimaging'} | {'1.0'}     | {'0x0 char'}             | {'li-imx274mipi-fmc'} | {'1.0'} |
| {'Artix-7 AC701 Evaluation Platform'} | {'xilinx.com'}     | {'1.1'}     | {'xc7a200tfbg676-2'}     | {'ac701'}             | {'1.4'} |
| {'Kria K26C SOM'}                     | {'xilinx.com'}     | {'Rev_B01'} | {'xck26-sfvc784-2LV-c'}  | {'k26c'}              | {'1.2'} |
| {'Kria K26C SOM'}                     | {'xilinx.com'}     | {'Rev_B01'} | {'xck26-sfvc784-2LV-c'}  | {'k26c'}              | {'1.3'} |
| {'Kria K26C SOM'}                     | {'xilinx.com'}     | {'Rev_B01'} | {'xck26-sfvc784-2LV-c'}  | {'k26c'}              | {'1.4'} |
| {'Kria K26i SOM'}                     | {'xilinx.com'}     | {'Rev_B01'} | {'xck26-sfvc784-2LV1-i'} | {'k26i'}              | {'1.2'} |
| {'Kria K26i SOM'}                     | {'xilinx.com'}     | {'Rev_B01'} | {'xck26-sfvc784-2LV1-i'} | {'k26i'}              | {'1.3'} |


```

### 3. Restart Vitis Model Composer.

The board will now be visible under the Boards tab of the Device Chooser within the Vitis Model Composer Hub block.



## Hardware Co-Simulation Support

If you have an FPGA development board, you might be able to take advantage of Vitis Model Composer's ability to use FPGA hardware co-simulation with Simulink® simulations. The Model Composer software includes support for all AMD Development Boards. Model Composer board support packages can be downloaded from the [Boards and Kits](#) page on the AMD website.

## UNC Paths Not Supported

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

---

# Supported MATLAB Versions and Operating Systems

Vitis Model Composer supports the following MATLAB versions:

- R2024a
- R2024b

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

- **Windows 11 Enterprise:** 11.0 23H2; 11.0 24H2
- **Windows 10 Pro and Enterprise:** 10.0 22H2

*Note:* AI Engine development is not supported on the Windows platform.

- **Red Hat Enterprise Workstation/Server 8:** 8.10
- **Red Hat Enterprise Workstation/Server 9:** 9.2; 9.3; 9.4; 9.5
- **Ubuntu Linux 22:** 22.04.2 LTS; 22.04.3 LTS; 22.04.4 LTS; 22.04.5 LTS
- **Ubuntu Linux 24:** 24.04 LTS; 24.04.1 LTS

---

# Recommended Display Resolution



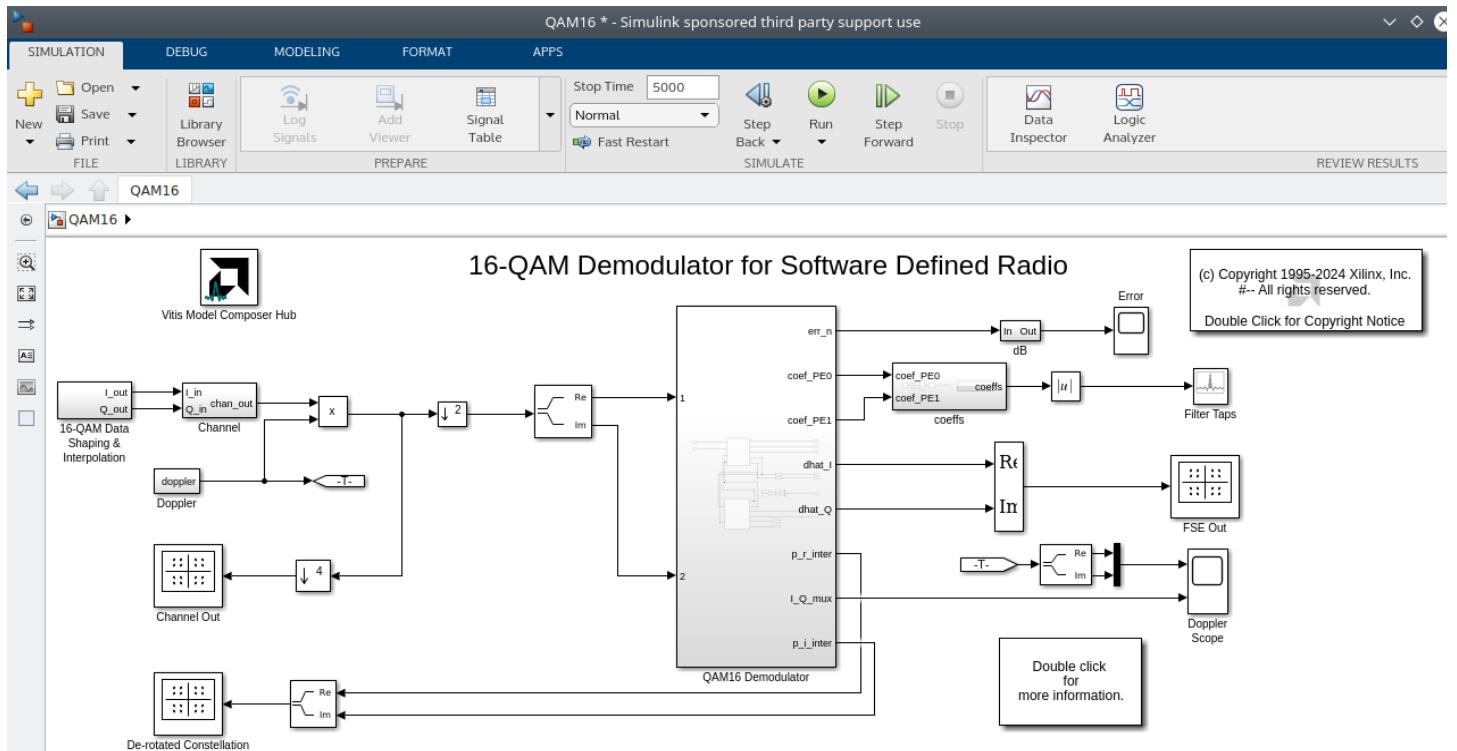
**RECOMMENDED:** The recommended display resolution for both Windows and Linux is 1920 x 1080 with scaling set at 100%. The blocks will look best with these settings.

# HDL Library

## Introduction

AMD Vitis™ Model Composer provides the HDL blockset in AMD toolbox that enables the use of the MathWorks model-based Simulink design environment for FPGA design. Previous experience with AMD FPGAs or RTL design methodologies are not required when using the Model Composer HDL blockset. Designs are captured in the DSP friendly Simulink modeling environment using the HDL blockset. The Model Composer design can then be imported into a Vivado IDE project using the IP catalog.

*Figure 5: Model Composer HDL Design*

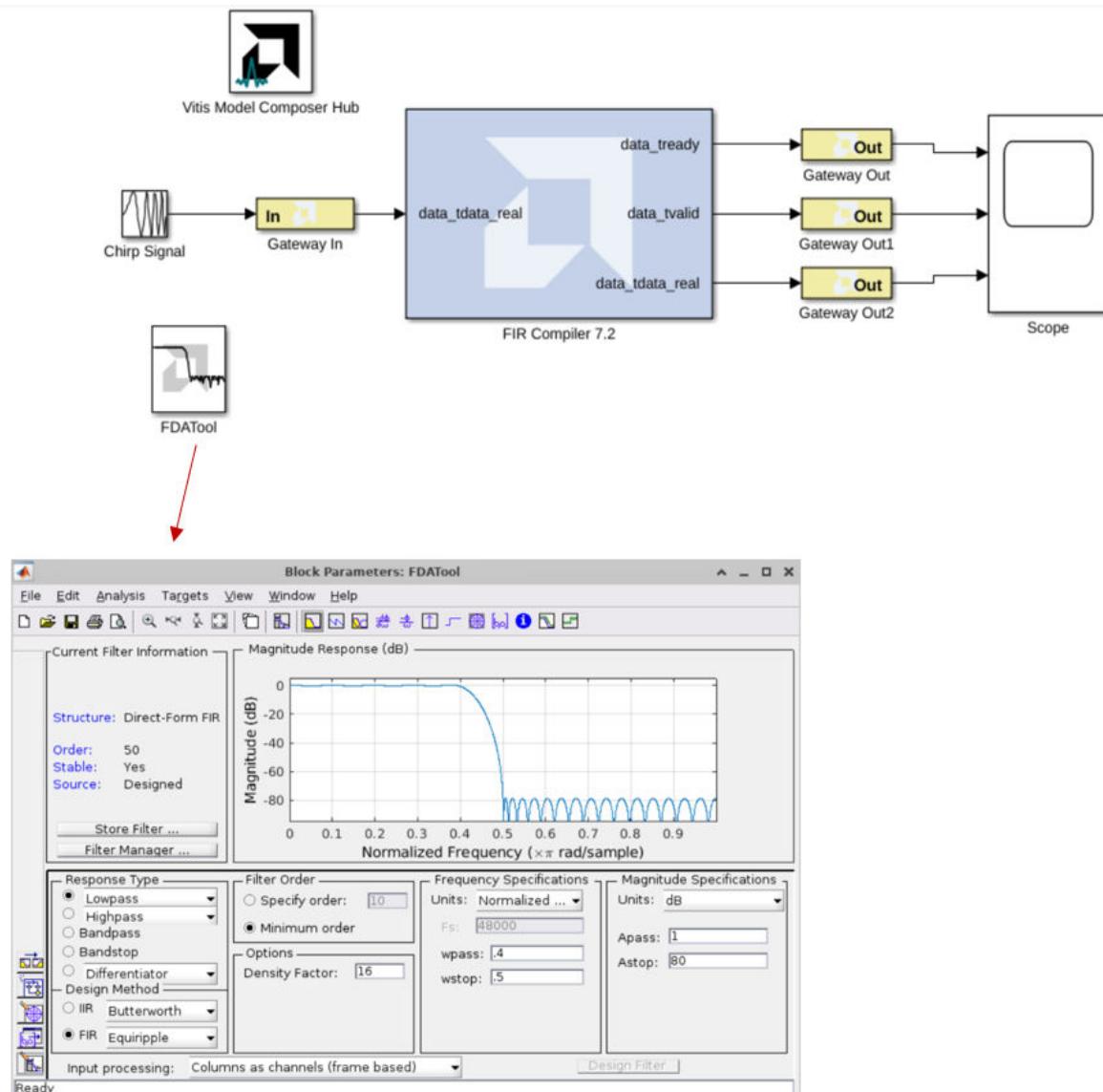


Refer to the [Vitis Model Composer Tutorials](#) for hands-on lab exercises and step-by-step instruction on how to create a model using HDL blockset and then import that model into a Vivado IDE project.

## FIR Filter Generation

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

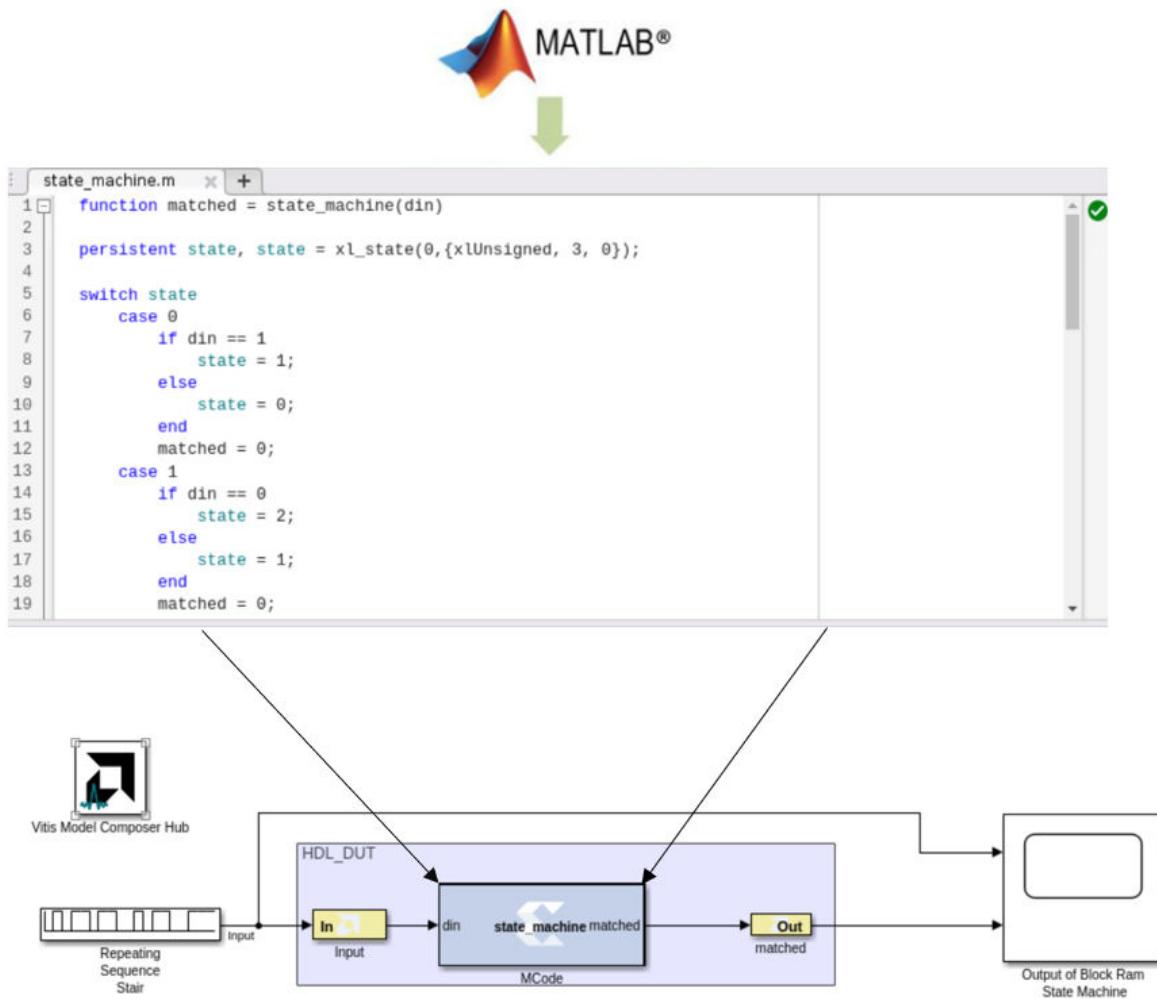
Figure 6: FDA Tool Example



## Support for MATLAB

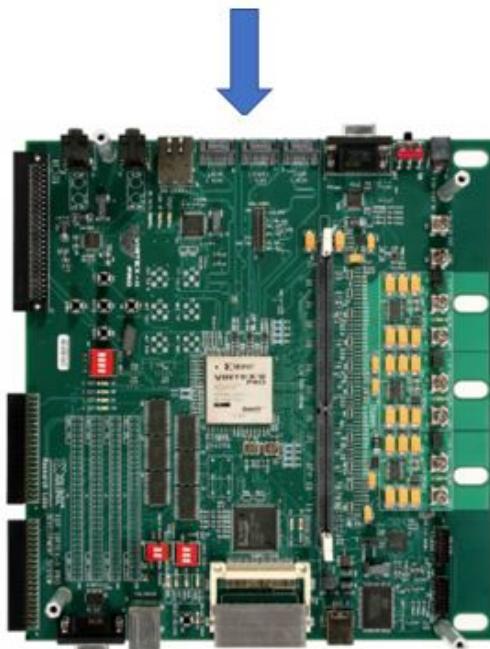
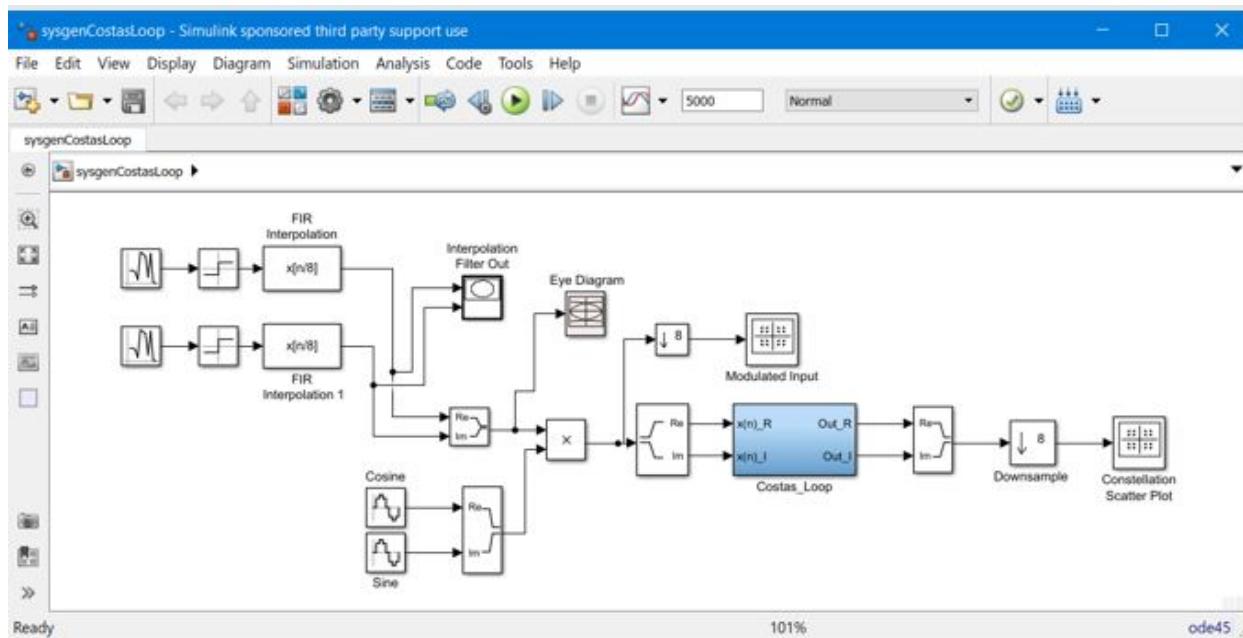
The HDL Library in Model Composer consists of an MCode block that allows the use of non-algorithmic MATLAB® for the modeling and implementation of simple control operations.

Figure 7: MCode Block Example



## Hardware Co-Simulation

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

**Figure 8: Hardware Co-Simulation**

## System Integration Platform

Model Composer 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. Model Composer supports a black box block that allows RTL to be imported into Simulink and co-simulated with either Questa or AMD Vivado™ simulator, and provides an AMD Vitis™ HLS block that allows integration and simulation of C/C++ sources.

---

## Hardware Design Using the HDL Library

Model Composer 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.

**Table 1: Hardware Design Using the HDL Library**

<a href="#">Design Flows Using Model Composer</a>	Describes several settings in which constructing designs in Model Composer is useful.
<a href="#">System-Level Modeling in Vitis Model Composer</a>	Discusses Model Composer's ability to implement device-specific hardware designs directly from a flexible, high-level, system modeling environment.
<a href="#">Automatic Code Generation</a>	Discusses automatic code generation for Model Composer designs using the HDL Library.
<a href="#">Compiling MATLAB into an FPGA</a>	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 Model Composer HDL Library and can be automatically compiled into equivalent HDL.
<a href="#">Importing a Model Composer HDL Design into a Bigger System</a>	Discusses how to take the VHDL netlist from a Vitis Model Composer design and synthesize it to embed it into a larger design. Also shows how VHDL created by Model Composer can be incorporated into a simulation model of the overall system.
<a href="#">Variant Subsystems and Vitis Model Composer</a>	Explains how to use Configurable Subsystems in Model Composer. Describes common tasks such as defining Configurable Subsystems, deleting, and adding blocks, and using Configurable Subsystems to import compilation results into Model Composer designs.
<a href="#">Notes for Higher Performance FPGA Design</a>	Suggests design practices in Model Composer that lead to an efficient and high-performance implementation in an FPGA.
<a href="#">Using the FDATool in Digital Filter Applications</a>	Demonstrates one way to specify, implement, and simulate a FIR filter using the FDATool block.
<a href="#">Multiple Independent Clocks Hardware Design</a>	The design can be partitioned into groups of Subsystem blocks, where each Subsystem has a common cycle period, independent of the cycle period of other Subsystems.

**Table 1: Hardware Design Using the HDL Library (cont'd)**

<a href="#">AXI Interface</a>	Provides an introduction to AMBA AXI4 and draws attention to AMBA AXI4 details with respect to Model Composer
<a href="#">AXI4-Lite Slave Interface Generation</a>	Describes features in Vitis Model Composer that allow you to create a standard AXI4-Lite interface for a Model Composer module and then export the module to the AMD Vivado™ IP catalog for later inclusion in a larger design using IP integrator.

## Design Flows Using Model Composer

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

### ***Algorithm Exploration***

Vitis Model Composer 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 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. Model Composer 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. Model Composer's facilities for hardware co-simulation are particularly useful when portions of a design are being refined.

### ***Implementing Part of a Larger Design***

Often Vitis Model Composer is used to implement a portion of a larger design. For example, Model Composer 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 can be useful to implement parts of the design using Model Composer, 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 Model Composer portion as a component. The non-Model Composer 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 Vitis Model Composer. For such a design, clicking the **Export** button instructs Model Composer to translate the design into HDL, and to write the files needed to process the HDL using downstream tools. The files written include the following:

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

For details concerning the files that Model Composer writes, see the topic [Compilation Results](#).

## ***Note to DSP Engineers***

Vitis Model Composer 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), Model Composer also provides access to features in the underlying FPGA.

The more you know about a hardware realization (for example, 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. Model Composer also makes it possible to refine a model to more accurately fit the application.

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

## ***Note to Hardware Engineers***

Vitis Model Composer 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 (for example, using DDR memory or phased clocking), you should implement using HDL. The less critical portions of the design can be implemented in Model Composer, and then the HDL and Model Composer portions can be connected. Usually, most portions of a signal processing system do not need this level of control, except at external interfaces. Model Composer 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 Model Composer that is of interest to engineers who design using HDL is its ability to automatically generate an HDL test bench, including test vectors. This aspect is described in the topic [HDL Testbench](#).

Finally, the hardware co-simulation interfaces described in [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 Vitis Model Composer

Vitis Model Composer allows device-specific hardware designs to be constructed directly in a flexible high-level system modeling environment. In a Model Composer 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.

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

**Table 2: System-Level Modeling in Vitis Model Composer**

<a href="#">Model Composer HDL Blocksets</a>	Describes how Vitis Model Composer's HDL blocks are organized in libraries, and how the blocks can be parameterized and used.
<a href="#">Signal Types</a>	Describes the data types used by Model Composer and ways in which data types can be automatically assigned by the tool.
<a href="#">Bit-True and Cycle-True Modeling</a>	Specifies the relationship between the Simulink-based simulation of a Vitis Model Composer model and the behavior of the hardware that can be generated from it.
<a href="#">Timing and Clocking</a>	Describes how clocks are implemented in hardware, and how their implementation is controlled inside Vitis Model Composer. Explains how Model Composer translates a multirate Simulink model into working clock-synchronous hardware.

**Table 2: System-Level Modeling in Vitis Model Composer (cont'd)**

Synchronization Mechanisms	Describes mechanisms that can be used to synchronize data flow across the data path elements in a high-level Vitis Model Composer 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.

## **Model Composer HDL 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, Model Composer HDL library blocksets are used like other Simulink blocksets. The blocks provide abstractions of mathematical, logical, memory, and DSP functions that can be used to build sophisticated signal processing (and other) systems. There are also blocks that provide interfaces to other software tools (for example, FDATool, Questa) as well as the Model Composer code generation software.

Model Composer HDL 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.

### **AMD HDL Blockset**

The AMD HDL Blockset is a family of libraries that contain basic Model Composer HDL 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. The libraries are described in the following table.

**Table 3: HDL Blockset Libraries**

Library	Description
Basic Elements	Standard building blocks for digital logic.
DSP	Digital signal processing (DSP) blocks.
Interfaces	Blocks that support connection with Simulink blocks.
Logic and Bit Operations	Blocks for performing logic and bit operations.
Memory	Blocks that implement and access memories.
Signal Routing	Blocks that support Routing signals.
Sources	Blocks that generate signal data
Tools	“Utility” blocks. For example, code generation (Vitis Model Composer Hub block), resource estimation, HDL co-simulation, etc.
User-Defined functions	Blocks that support importing custom functions.
SSR	Blocks that support Super Sample Rate algorithms with vector inputs and outputs.

## Signal Types

In order to provide bit-accurate simulation of hardware, HDL 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 gateway blocks in the Model Composer HDL library allow connection between HDL blocks in the AMD toolbox and blocks in the Simulink library. The Gateway In converts a double precision signal into an AMD signal, and the Gateway Out converts an AMD signal into double precision. Simulink® continuous time signals must be sampled by the Gateway In block.

Most HDL blocks are polymorphic (that is, they can deduce appropriate output types based on their input types). When *full precision* is specified for a block in its parameters dialog box, Model Composer 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:** Vitis Model Composer data types can be displayed by selecting **Display → Signals & Ports → Port Data Types** in Simulink. Displaying data types makes it easy to determine precision throughout a model. If, for example, the type for a port is `Fix_11_9`, then the signal is a two's complement signed 11-bit number having nine fractional bits. Similarly, if the type is `Ufix_5_3`, then the signal is an unsigned 5-bit number having three fractional bits.

In the Model Composer portion of a Simulink model, every signal must be sampled. Sample times can be inherited using Simulink's propagation rules, or set explicitly in a block customization dialog box. When there are feedback loops, Model Composer 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 (click **Display → Sample Time → Colors** in the Simulink pull-down menus). This is often useful in understanding multirate designs.

## Floating-Point Data Type

Many Model Composer HDL blocks across various libraries support the floating-point data type.

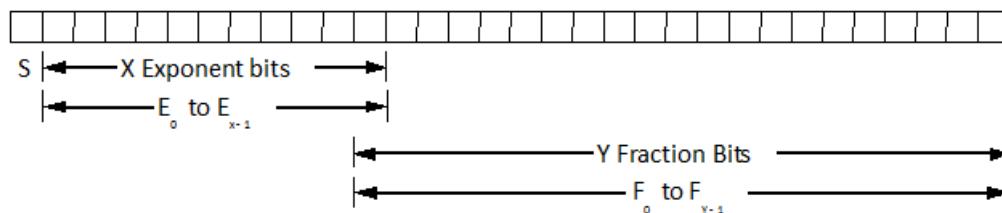
Model Composer uses the Floating-Point Operator v7.1 IP core to leverage the implementation of operations such as addition/subtraction, multiplication, comparisons, and data type conversion.

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

### IEEE-754 Standard for Floating-Point Data Type

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

Figure 9: Floating-Point Data



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

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

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

Equation 1: Exponent Bias

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

Where X is the exponent bit width.

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

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

**Equation 2: Normalized Floating Point Value**

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

The actual value of exponent ( $E_{\text{actual}}$ ) =  $E - \text{Exponent\_bias}$ . Considering 1 as the value for the hidden bit F0 and the  $E_{\text{actual}}$  value, a floating-point number can be calculated as follows:

**Equation 3: Floating Point Number Calculation**

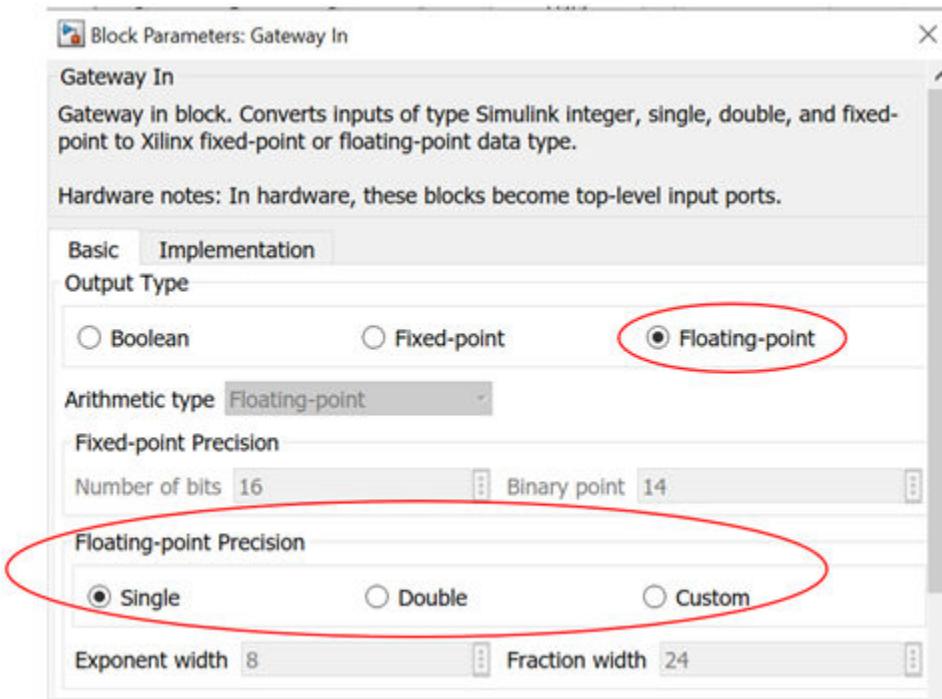
$$\text{FP\_Value} = (-1)^S \times 1.F1F2 . FY-2FY-1 \times (2)^{(E_{\text{actual}})}$$

**Floating-Point Data Representation in Model Composer**

The HDL Gateway In block supports Boolean, Fixed-point, and Floating-point data types as shown in the following figure. You can select either a Single, Double, or Custom precision type after specifying the floating-point data type.

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

*Figure 10: Floating-point Precision*

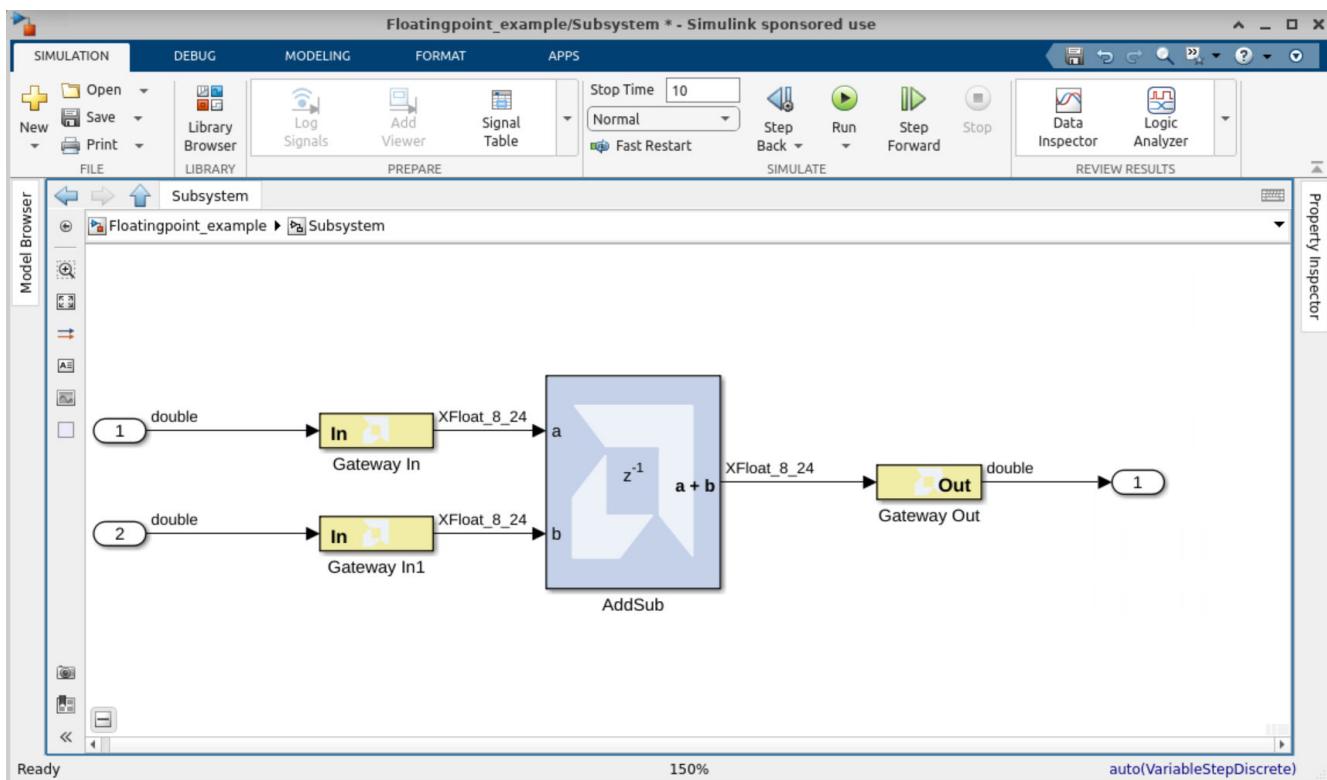


In compliance with the IEEE-754 standard, if Single precision is selected then the total bit width is assumed to be 32; 8 bits for the exponent and 24 bits for the fraction. Similarly when Double precision is selected, the total bit width is assumed to be 64 bits; 11 bits for the exponent and 53 bits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

### Displaying the Data Type on Output Signals

As shown in the following figure, after a successful rate and type propagation, the floating-point data type is displayed on the output of each HDL block. To display the signal data type as shown in the following figure, select **Debug → Diagnostics → Information Overlays → Port Data Types**.

*Figure 11: Floating-point Data Type*



A floating-point data type is displayed using the format:

`XFLOAT_<exponent_bit_width>_<fraction_bit_width>`. Single and Double precision data types are displayed using the string "XFLOAT\_8\_24" and "XFLOAT\_11\_53", respectively.

If for a Custom precision data type the exponent bit width 9 and the fraction bit width 31 are specified, then it will be displayed as "XFfloat\_9\_31". A total of 40 bits will be used to store the floating-point data value. Because floating-point data is stored in a normalized form, the fractional value will be stored in 30 bits.

In Model Composer 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.

Model Composer 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 Model Composer HDL 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 an HDL 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 an HDL block are not of the same floating-point data type. The DRC check will be made between the two inputs of a block as well as between an input and an output of the block.

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

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

3. If the data inputs are of the fixed-point data type and the data output is expected to be floating-point and vice versa.

**Note:** The Convert and Relational blocks are excluded from this check. The Convert block supports Fixed-to-float as well as Float-to-fixed data type conversion. The Relational block output is always the Boolean data type because it gives a true or false result for a comparison operation.

4. If Custom precision is selected for the Output Type of blocks that support the floating-point data type. For example, for blocks such as AddSub, Mult, CMult, and MUX, only Full output precision is supported if the data inputs are of the floating-point data type.
5. If the Carry In port or Carry Out port is used for the AddSub block when the operation on a floating-point data type is specified.

6. If the Floating-Point Operator IP core gives an error for DRC rules defined for the IP.

## ***AXI Signal Groups***

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

## ***Bit-True and Cycle-True Modeling***

Simulations in Vitis Model Composer are *bit-true* and *cycle-true*. To say a simulation is bit-true means that at the boundaries (that is, interfaces between Model Composer HDL blocks and non-HDL 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 Model Composer HDL gateway blocks exist. When a design is translated into hardware, Gateway In (respectively, Gateway Out) blocks become top-level input (resp., output) ports.

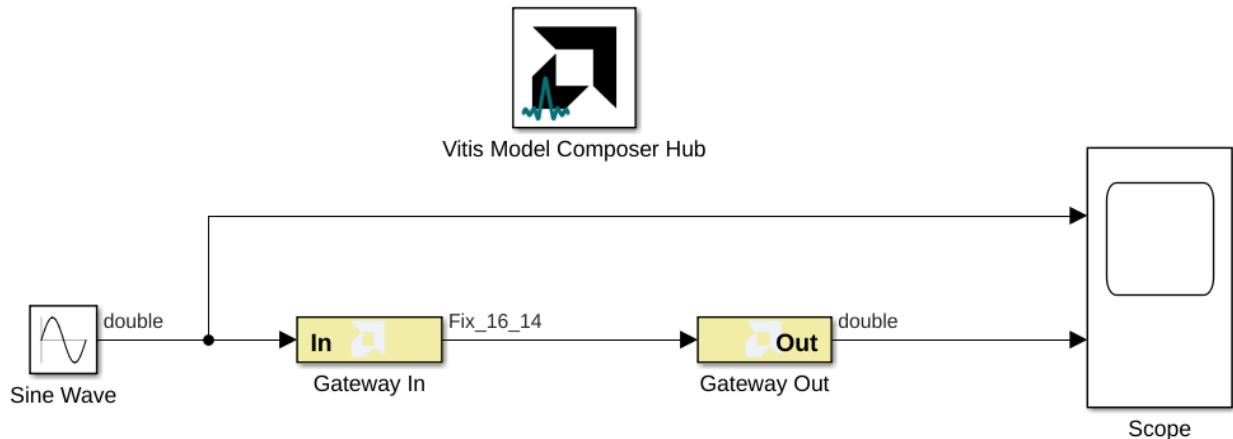
## ***Timing and Clocking***

### **Discrete Time Systems**

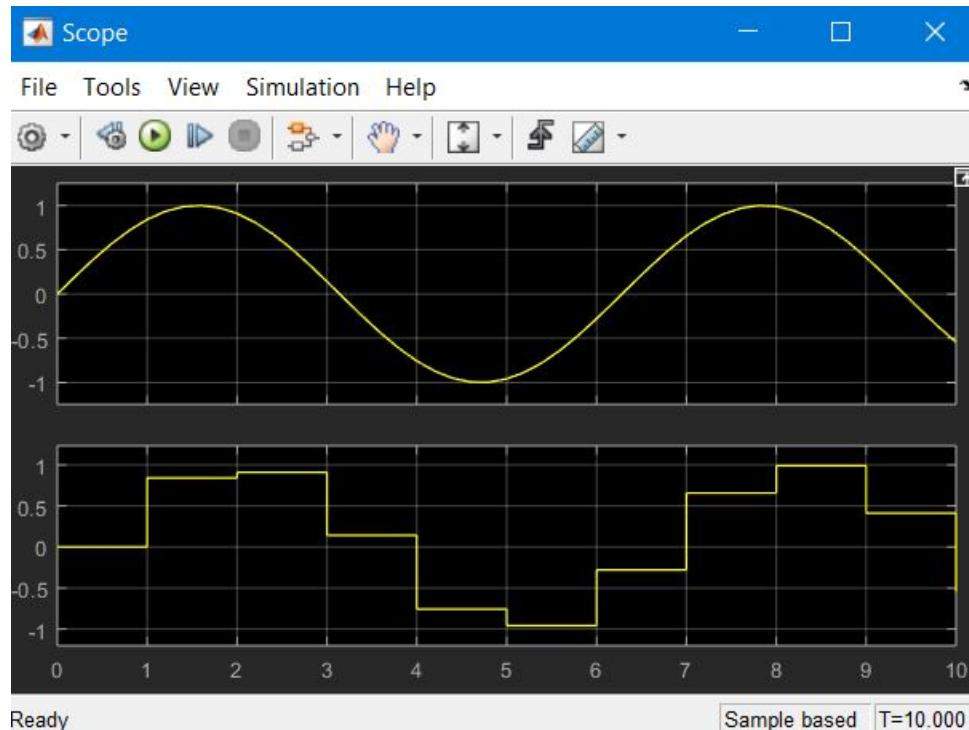
Designs in Vitis Model Composer are discrete time systems. In other words, the signals and the blocks that produce them have associated sample rates. A block's sample rate determines how often the block is awoken (allowing its state to be updated). Model Composer sets most sample rates automatically. 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 Model Composer model illustrates the behavior of discrete time systems. Consider the model shown below. It contains a gateway that is driven by a Simulink source (Sine Wave), and a second gateway that drives a Simulink sink (Scope).

**Figure 12: Discrete Time System**

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

**Figure 13: Scope Output**

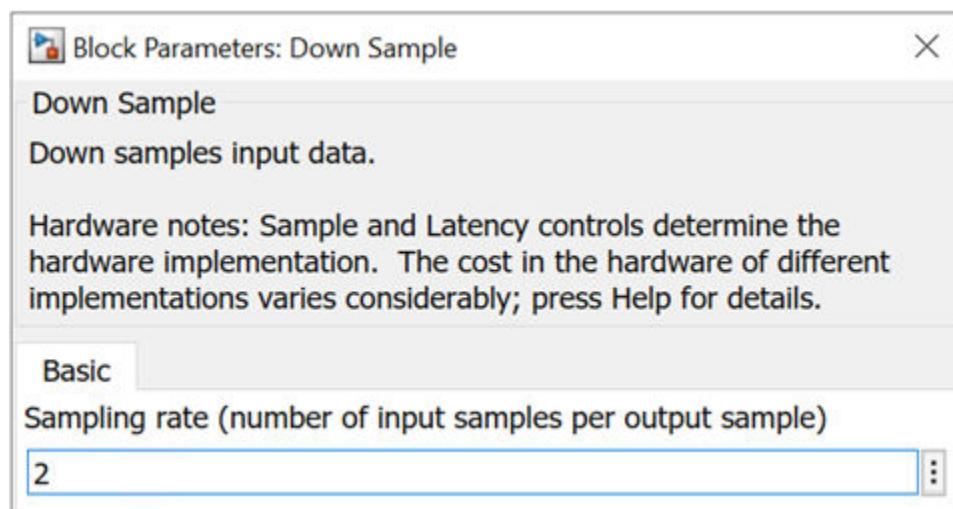
## Multirate Models

Model Composer supports *multirate* designs, that is, designs having signals running at several sample rates. Model Composer 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

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

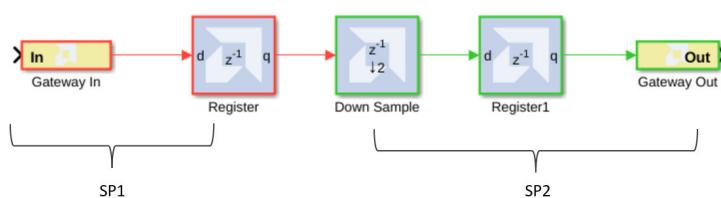
Figure 14: Rate Change Dialog



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

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

Figure 15: Multirate Example



## Hardware Oversampling

Some HDL blocks are oversampled, that is, 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, Model Composer 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 Model Composer Hub block dialog box.

## Asynchronous Clocking

Model Composer 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 Model Composer. This discussion is relevant to both single-clock and multiple-clock designs.

## Synchronous Clocking

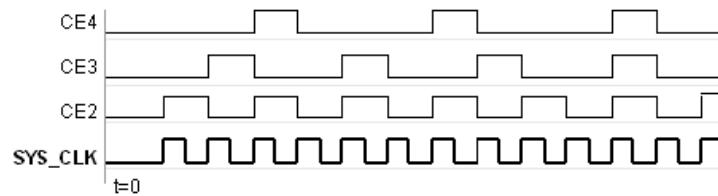
By default, Model Composer creates designs with synchronous clocking, where multiple rates are realized using clock enables. When Model Composer compiles a model into hardware, it preserves the sample rate information of the design in such a way that corresponding portions in hardware run at appropriate rates. In hardware, Model Composer 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 Model Composer design. When Model Composer 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 Model Composer Hub block: 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  $k_p$  units of time in Simulink takes  $k$  ticks of the system clock (hence  $k_c$  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 10 ns. With this information, the corresponding clock enable periods can be determined in hardware.

In hardware, 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 in the following figure.

Figure 16: Timing Diagram



## Synchronization Mechanisms

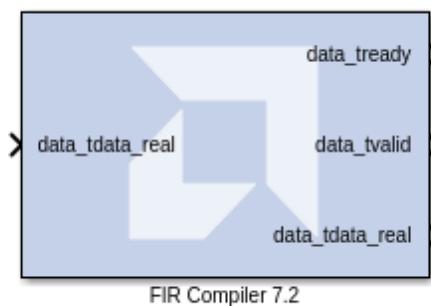
Model Composer does not make implicit synchronization mechanisms available. Instead, synchronization is the responsibility of the designer, and must be done explicitly.

### Valid Ports

Vitis Model Composer provides several blocks (in particular, the AXI FIFO) that can be used for synchronization. Several blocks provide optional AXI signaling interfaces to denote when a sample is valid (TValid) and when the interface is ready for data (TReady).

**Note:** The tvalid / tready ports might not be visible based on the configuration of the IP. Color association denotes a collection of ports for each interface on the block as shown below. Blocks with interfaces can be chained, affording a primitive form of flow control. Examples of such blocks with AXI interfaces include the FFT, FIR, and DDS.

Figure 17: Block with AXI Interface



## Indeterminate Data

Indeterminate values are common in many hardware simulation environments. Often they are called "don't cares" or "Xs." In particular, values in Model Composer 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. Model Composer 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 (Not a Number). In a Simulink scope, NaN values are not plotted. Conversely, NaNs that drive a Gateway In become indeterminate values. Model Composer provides an Indeterminate Probe block that allows for the detection of indeterminate values. This probe cannot be translated into hardware.

In Model Composer, 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 AMD 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 HDL blocks. Consequently, blocks in the AMD HDL 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

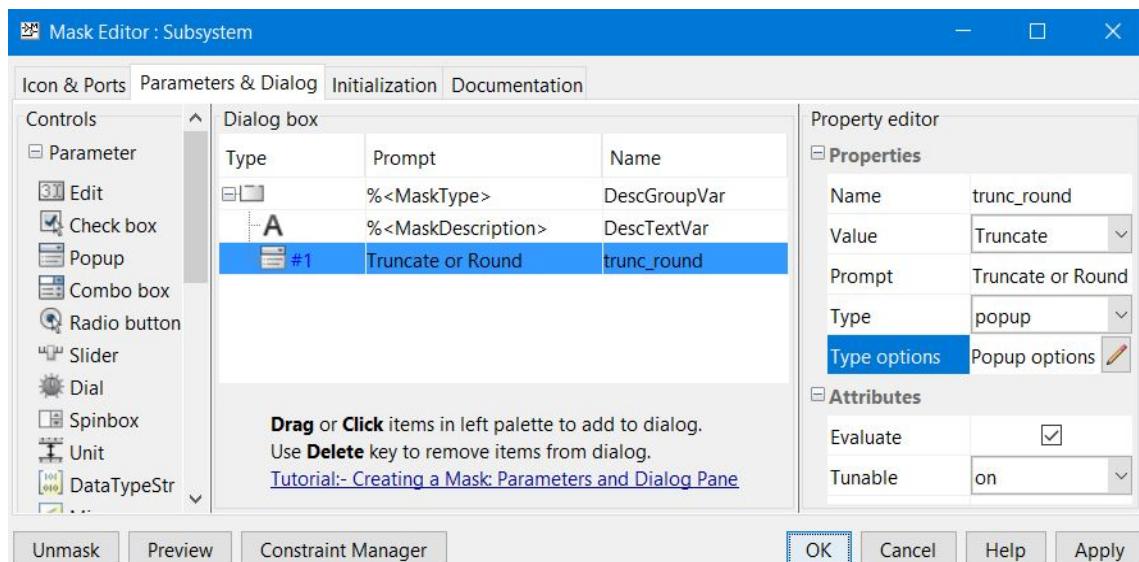
In Simulink, blocks are parameterized through a mechanism called *masking*. In essence, a block can be assigned *mask variables* whose values can be user-specified through dialog box prompts or can be calculated in mask initialization commands. Variables are stored in a *mask workspace*. A mask workspace 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.

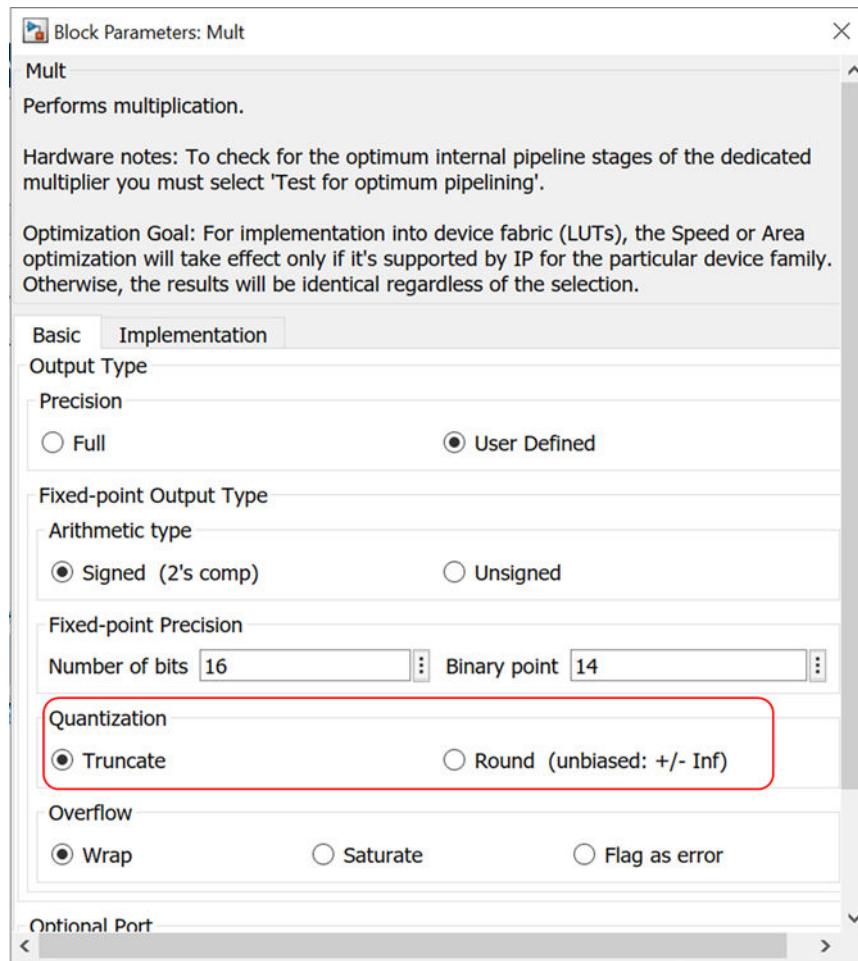
## Parameter Passing

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 AMD HDL 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 following figure.

Figure 18: Creating a Parameter



As shown in the following figure, in the Block Parameters dialog for the accumulator and multiplier blocks, there are radio buttons that allow either the Truncate or Round option to be selected.

**Figure 19: Editing a Parameter**

## Automatic Code Generation

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

**Table 4: Automatic Code Generation**

<a href="#">Compiling and Simulating Using the Model Composer Hub</a>	Describes how to use the Vitis Model Composer Hub Block to compile designs into equivalent low-level HDL.
<a href="#">Compilation Results</a>	Describes the low-level files Vitis Model Composer produces when HDL Netlist is selected on the Model Composer Hub block and Generate is pushed.
<a href="#">Vivado Project</a>	Describes the example project Vitis Model Composer produces when HDL Netlist or IP catalog is selected on the Model Composer Hub block and Generate is pushed.

**Table 4: Automatic Code Generation (cont'd)**

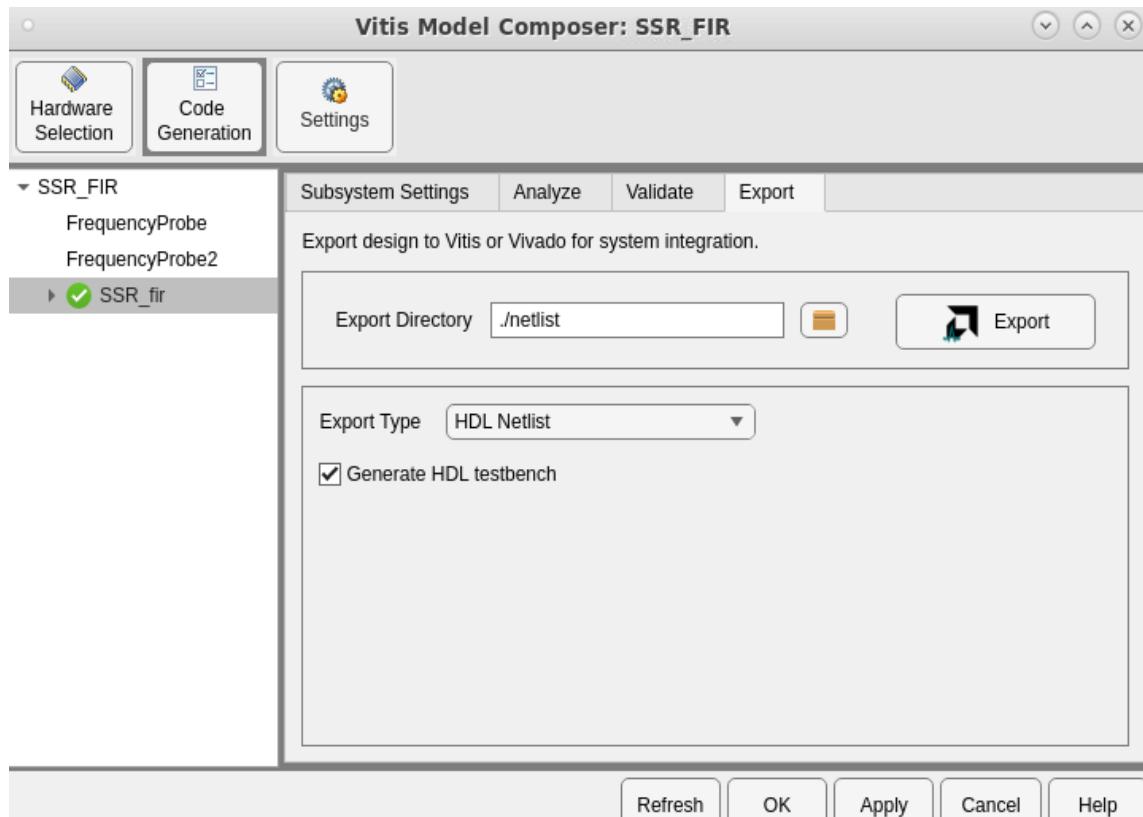
HDL Testbench	Describes the VHDL test bench that Model Composer can produce.
---------------	--

## **Compiling and Simulating Using the Model Composer Hub**

Vitis Model Composer automatically compiles designs into low-level representations. Designs are compiled and simulated using the Model Composer Hub block. This topic describes how to use the Vitis Model Composer Hub block.

A design must contain a single Vitis Model Composer Hub block at the top level of the model. This block controls code generation for all of the subsystems within a model. Some parameters, such as clock frequency, can be specified on a subsystem-by-subsystem basis.

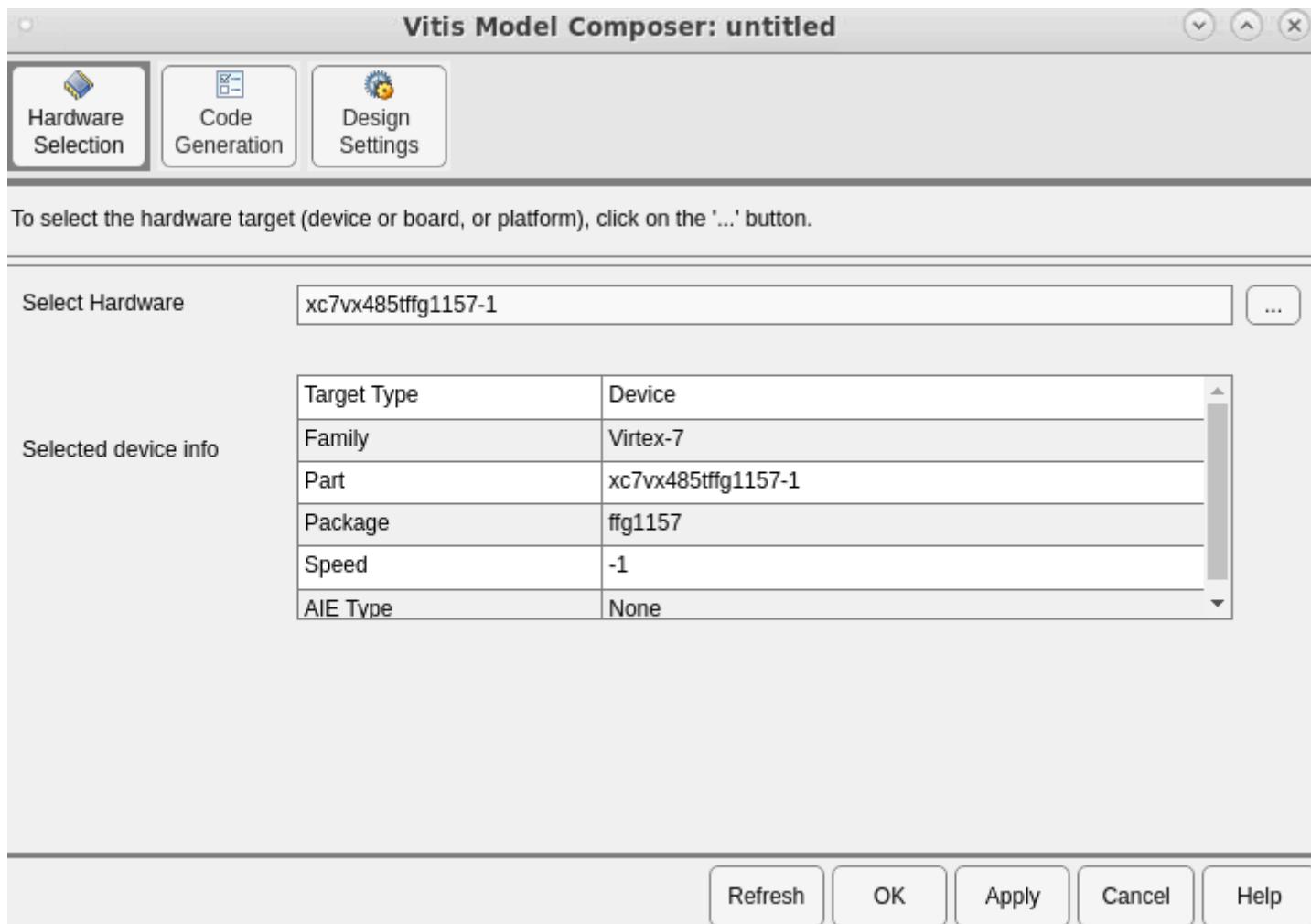
Once a Vitis Model Composer Hub block is added, it is possible to specify how code generation and synthesis should be handled. The Model Composer Hub block dialog box is shown in the following figure:

**Figure 20: Model Composer Hub Block**

### **Hardware Selection**

This tab allows you to specify an AMD, Partner, or Custom board you will use to test your design.

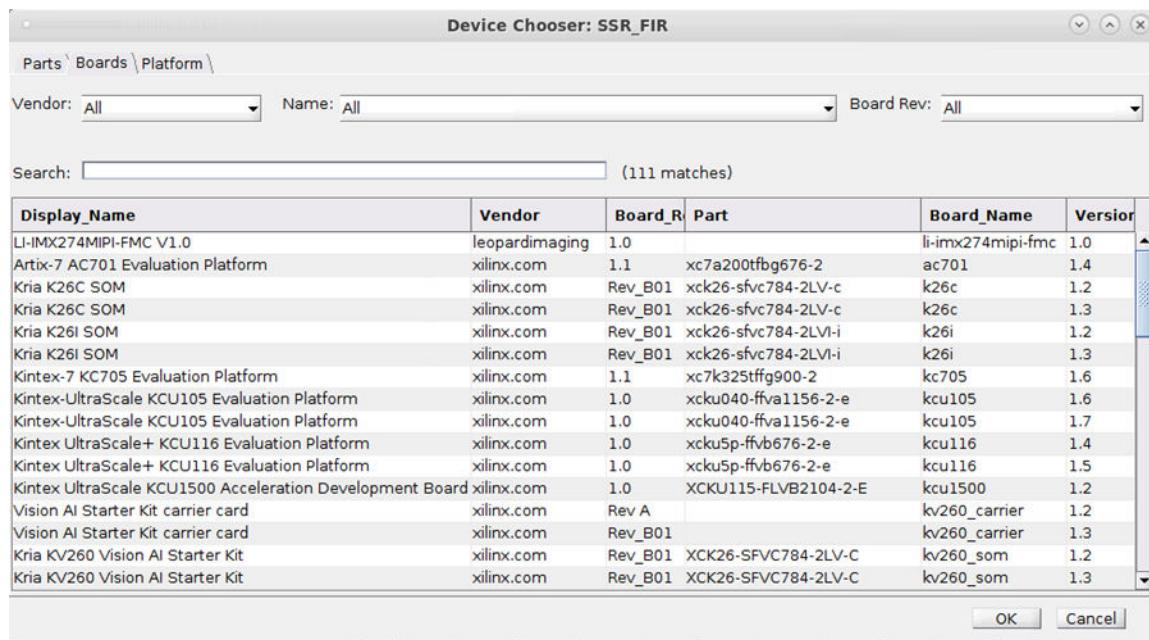
Figure 21: Hardware Selection Tab



Clicking the browse button (...) displays the Device Chooser dialog box. This allows you to select a part, board, or platform to which your design is targeted.

For a Partner board or a custom board to appear in the Board list, you must configure Model Composer to access the board files that describe the board. Board awareness in Model Composer is detailed in [Specifying Board Support in Model Composer](#).

Figure 22: Device Chooser



## Code Generation

The Code Generation tab contains, on the left side, a list of subsystems in the model. Subsystems denoted with a green check mark are ready to generate HDL, HLS, or AI Engine code. To configure code generation for a subsystem, select it from the list. Selecting a subsystem will generate code for it along with any of its child subsystems.

If a subsystem is not denoted with a green check mark, select it to view the reason why the subsystem cannot generate code, along with troubleshooting steps.

For a detailed description of the code generation settings available, refer to the relevant Model Composer Hub block section:

- [Vitis Model Composer Hub \(HDL\)](#)
- [Vitis Model Composer Hub \(HLS\)](#)
- [Vitis Model Composer Hub \(AI Engine\)](#)

## Settings

This tab allows you to specify the following settings for the entire model.

- **Treat this model as a legacy System Generator design for backward compatibility:** When you automatically upgrade a System Generator token to the Model Composer Hub block, this check box is automatically enabled. When the check box is enabled, you can use the Model Composer Hub block without requiring changes to your legacy designs. However, newer capabilities provided by the Model Composer Hub block, such as the Validate on Hardware Flow, will not be available.
- **Number of parallel AI Engine builds:** To speed up model compilation, Vitis Model Composer can build the AI Engine blocks in parallel, taking advantage of multiple cores on your machine. This value can be increased up to maximum number of cores on your machine.

## **Compilation Results**

This topic discusses the low-level files Vitis Model Composer produces when HDL Netlist is the selected Export Type and Export is clicked in the Model Composer Hub block. The files consist of HDL that implements the design. In addition, Model Composer organizes the HDL files, and other hardware files into an AMD Vivado™ IDE Project. These files are written to the folder <target directory>/ip/<design\_name>/src where <target directory> is the target directory specified on the Model Composer Hub block. If no test bench is requested, then the key files produced by Model Composer are the following:

**Table 5: Compilation Files**

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

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

**Table 6: Additional Compilation Files**

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

## Using the Constraints File

When a design is compiled during code generation, Vitis Model Composer 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 (that is, the period of the fastest hardware clock in the design) can be specified in the Model Composer Hub block. Model Composer 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, the 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, Vitis Model Composer's HDL Gateway In and Gateway Out blocks become input and output ports. The locations of these ports and the speeds at which they must operate can be entered in the Gateway In and Out parameter dialog boxes. Port location and speed are specified in the constraints file by IOB timing.

This topic describes how Model Composer handles hardware clocks in the HDL it generates. Assume the design is named <design>, and <design> is an acceptable HDL identifier. When Model Composer 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 (that is, 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 Vitis Model Composer 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 Model Composer designs this does not work. Instead, `clk_1` and `clk_2` must be driven by the same clock, `ce_1` must be tied high, and `ce_2` must vary at a rate half that of `clk_1` and `clk_2`.

## IP Instance Caching

When Vivado synthesis is invoked by Vitis Model Composer, a disk cache is used to speed up the iterative design process.

Vivado synthesis is invoked when performing Timing or Resource analysis or when exporting a Synthesized Checkpoint.

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

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

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

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

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

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

## Vivado Project

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

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

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

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

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

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

## HDL Testbench

Ordinarily, Vitis Model Composer 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 Generate HDL testbench check box in the Vitis Model Composer Hub block makes this possible.

Suppose the design is named <design>, and a Vitis Model Composer Hub block is placed at the top of the design. Suppose also that in the Hub block the Generate HDL testbench check box is selected on the Export tab. When the Export button is clicked, Model Composer produces the usual files for the design, and in addition writes the following:

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

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

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

## Compiling MATLAB into an FPGA

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

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

The following text provides examples that use the MCode block:

- Example 1 [Simple Selector](#) shows how to implement a function that returns the maximum value of its inputs;
- Example 2 [Simple Arithmetic Operations](#) shows how to implement simple arithmetic operations;
- Example 3 [Complex Multiplier with Latency](#) shows how to build a complex multiplier with latency;
- Example 4 [Shift Operations](#) shows how to implement shift operations;
- Example 5 [Passing Parameters into the MCode Block](#) shows how to pass parameters into a MCode block;
- Example 6 [Optional Input Ports](#) shows how to implement optional input ports on an MCode block;
- Example 7 [Finite State Machines](#) shows how to implement a finite state machine;
- Example 8 [Parameterizable Accumulator](#) shows how to build a parameterizable accumulator;
- Example 9 [FIR Example and System Verification](#) shows how to model FIR blocks and how to do system verification;
- Example 10 [RPN Calculator](#) shows how to model a RPN calculator – a stack machine;
- Example 11 [Example of disp Function](#) shows how to use disp function to print variable values.

## Simple Selector

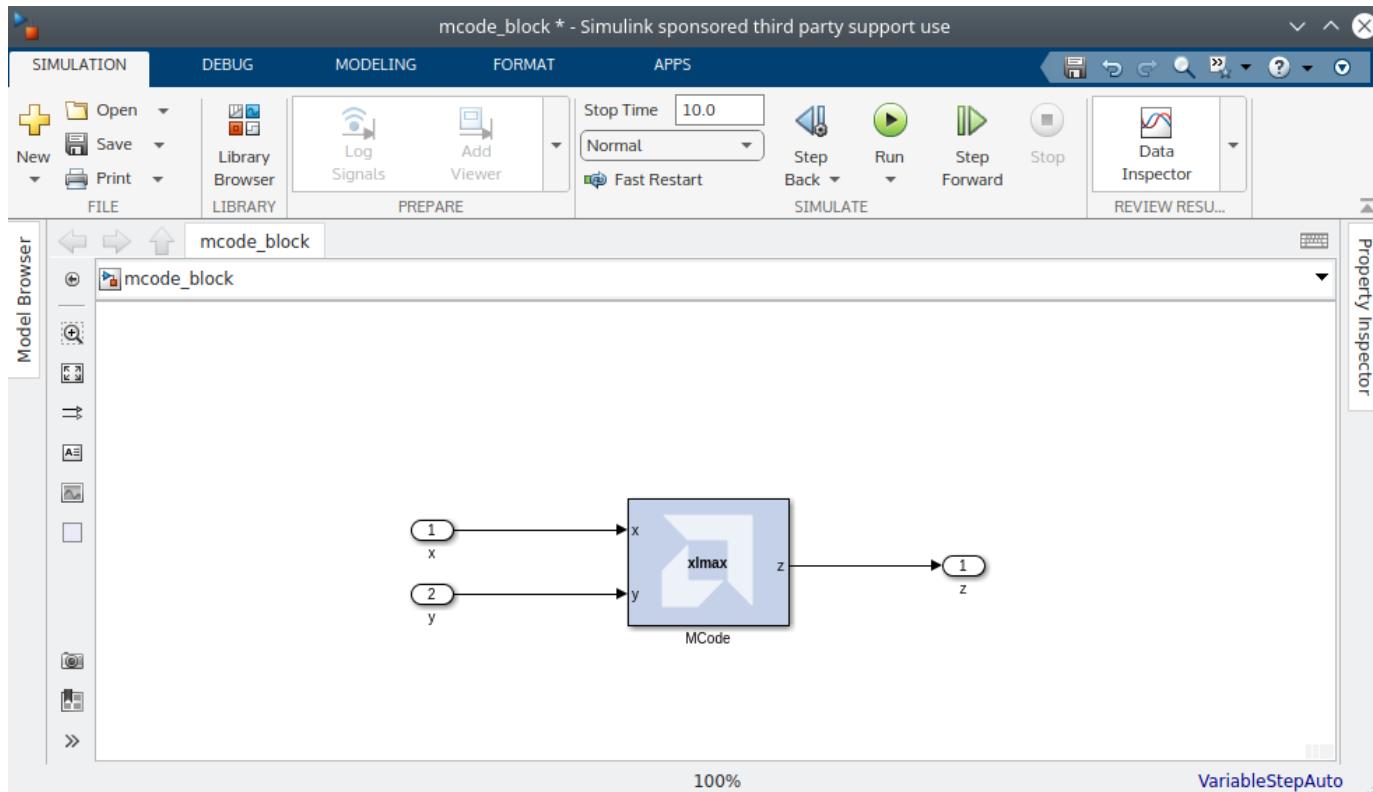
This example is a simple controller for a data path, which assigns the maximum value of two inputs to the output. The M-function is specified as the following and is saved in an M-file `x1max.m`:

```
function z = x1max(x, y)
if x > y
    z = x;
else
    z = y;
end
```

The `x1max.m` file should be either saved in the same directory of the model file or should be in the MATLAB path. Once the `x1max.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 `x1max` into the MATLAB Function field. After clicking the OK button, the block has two input ports `x` and `y`, and one output port `z`.

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

Figure 23: Simple Selector Design



## Simple Arithmetic Operations

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

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

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

One thing worth discussing is the `xfix` function call. The function requires two arguments: the first for fixed-point data type precision and the second indicating the value. The precision is specified in a cell array. The first element of the precision cell array is the type value. It can be one of three different types: `xlUnsigned`, `xlSigned`, or `xlBoolean`. The second element is the number of bits of the fixed-point number. The third is the binary point position. If the element is `xlBoolean`, there is no need to specify the number of bits and binary point position. The number of bits and binary point position must be specified in pair. The fourth element is the quantization mode and the fifth element is the overflow mode. The quantization mode can be one of `xlTruncate`, `xlRound`, or `xlRoundBanker`. The overflow mode can be one of `xlWrap`, `xlSaturate`, or `xlThrowOverflow`. 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 an AMD fixed-point number. If a constant is an integer number, there is no need to use the `xfix` function. The Mcode block converts it to the appropriate fixed-point number automatically.

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

Figure 24: xlSimpleArith MCode Parameter

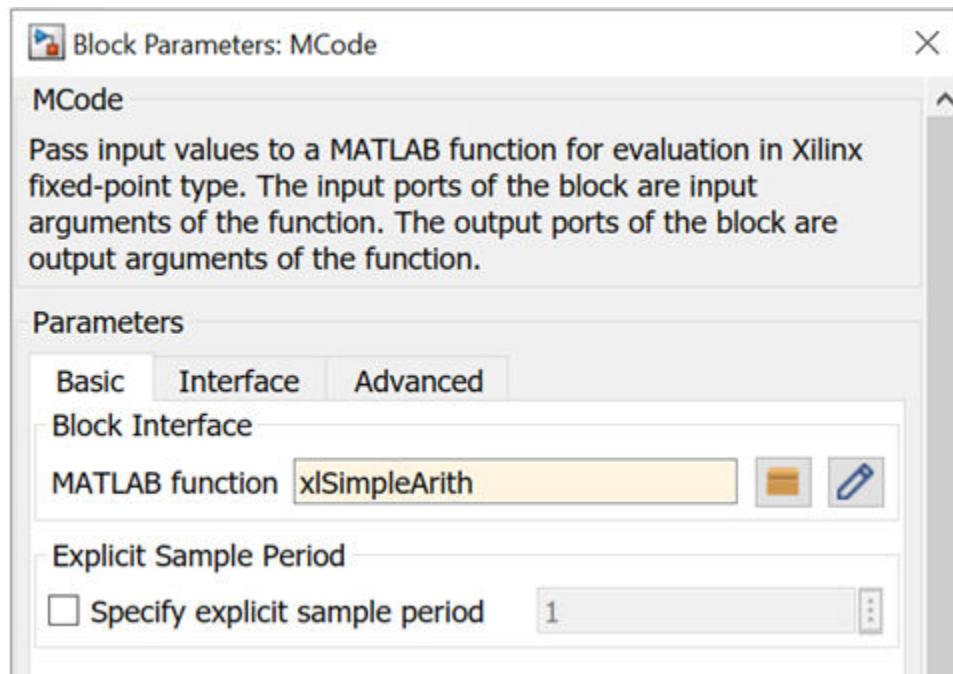
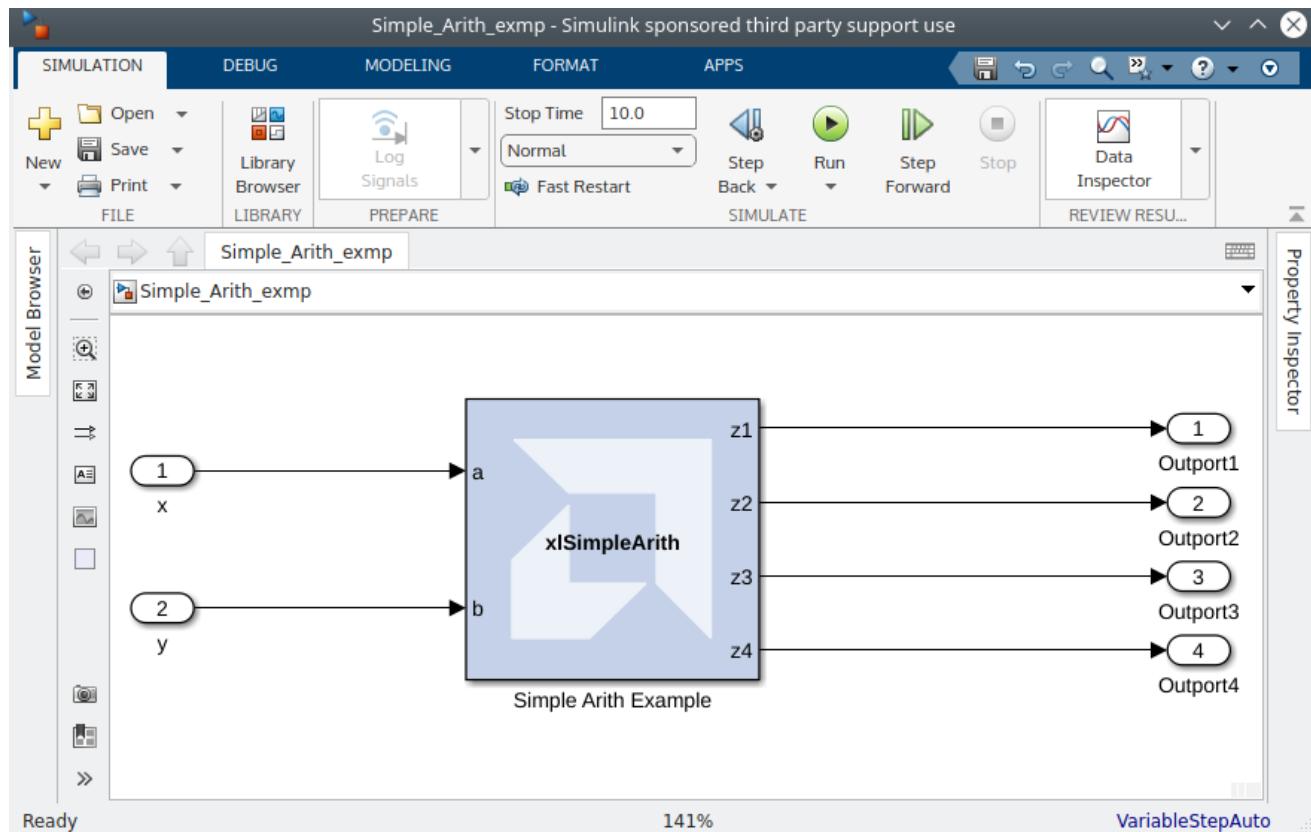


Figure 25: xlSimpleArith Design



M-functions using AMD 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
```

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

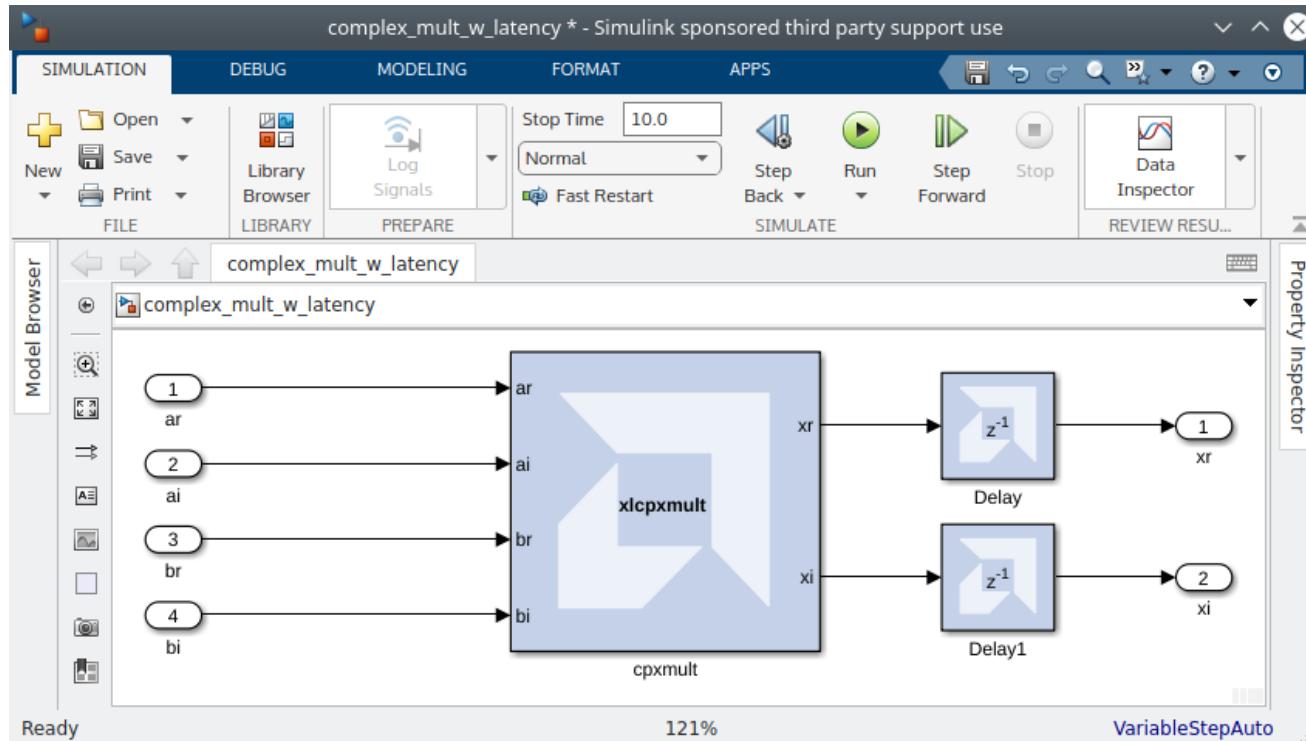
## Complex Multiplier with Latency

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

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

The following diagram shows the subsystem:

Figure 26: Complex Multiplier Subsystem



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

## Shift Operations

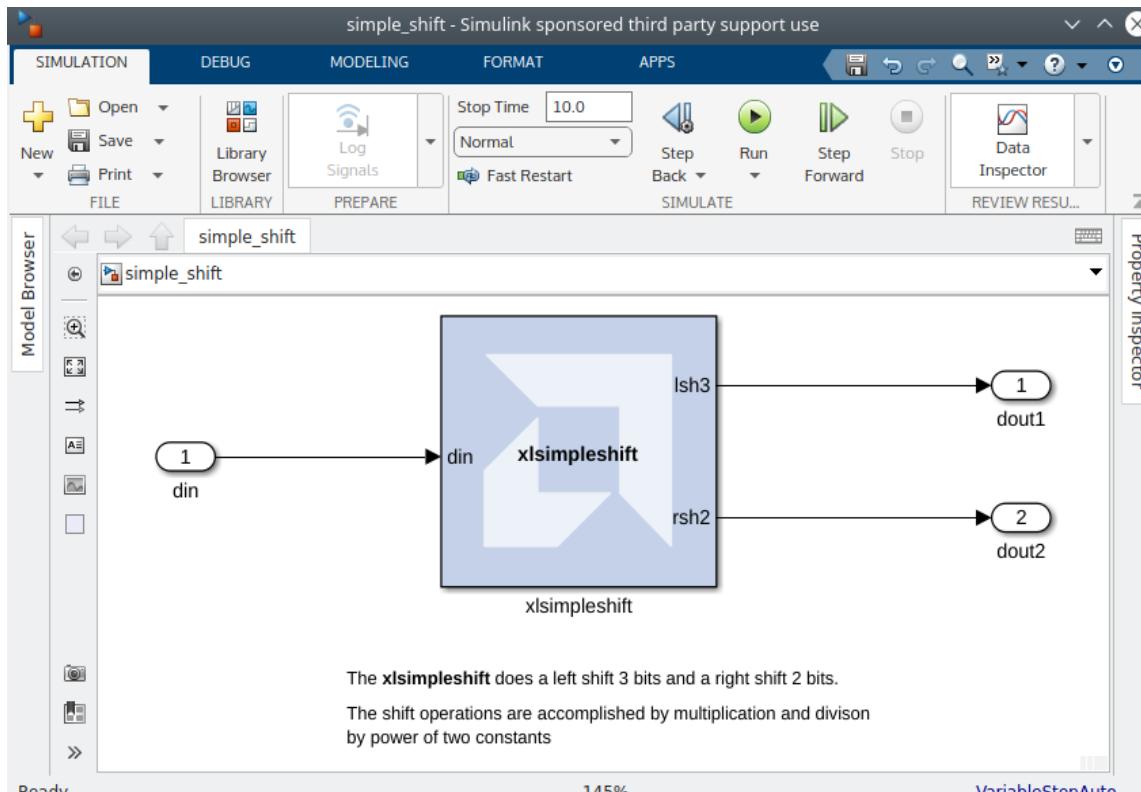
This example shows how to implement bit-shift operations using the MCode block. Shift operations are accomplished with multiplication and division by powers of two. For example, multiplying by 4 is equivalent to a 2-bit left-shift, and dividing by 8 is equivalent to a 3-bit right-shift. Shift operations are implemented by moving the binary point position and if necessary, expanding the bit width. Consequently, multiplying a Fix\_8\_4 number by 4 results in a Fix\_8\_2 number, and multiplying a Fix\_8\_4 number by 64 results in a Fix\_10\_0 number.

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

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

The following diagram shows the subsystem after compilation:

Figure 27: Shift Operations



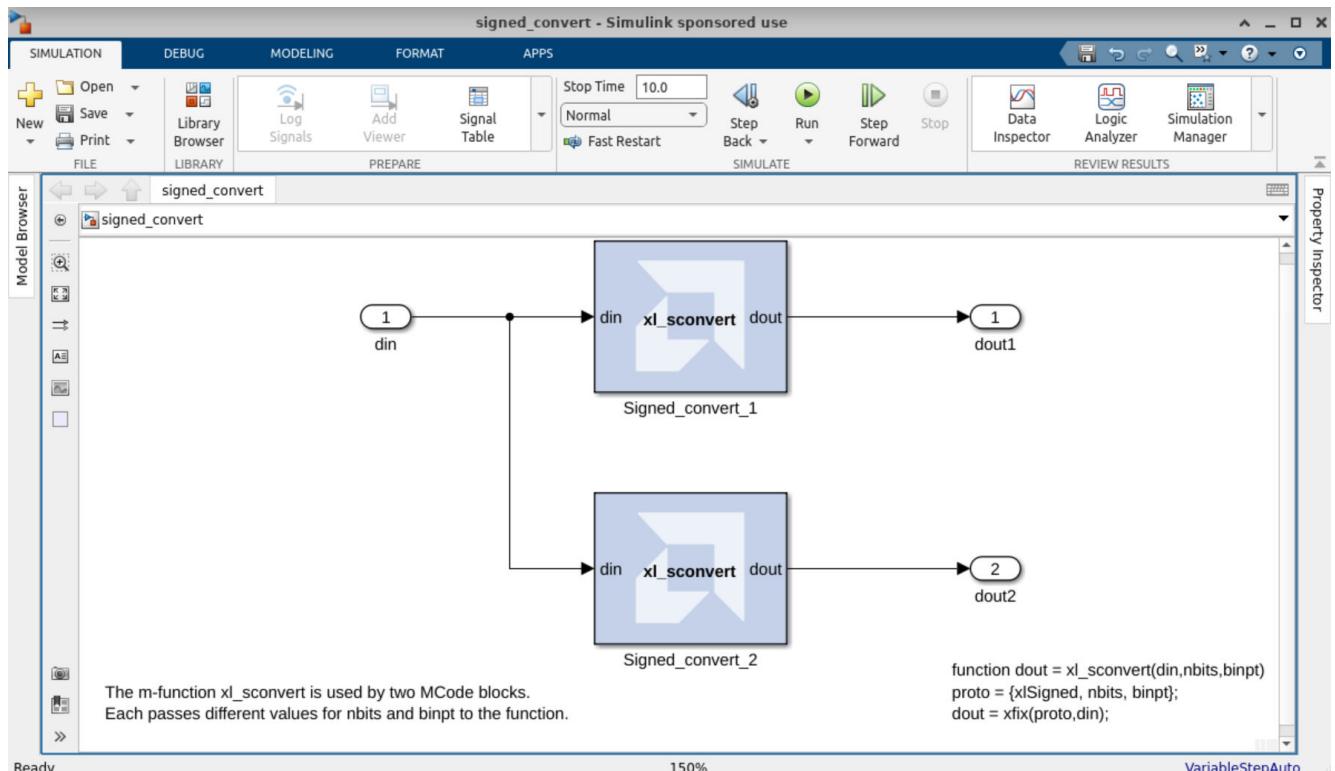
## Passing Parameters into the MCode Block

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

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

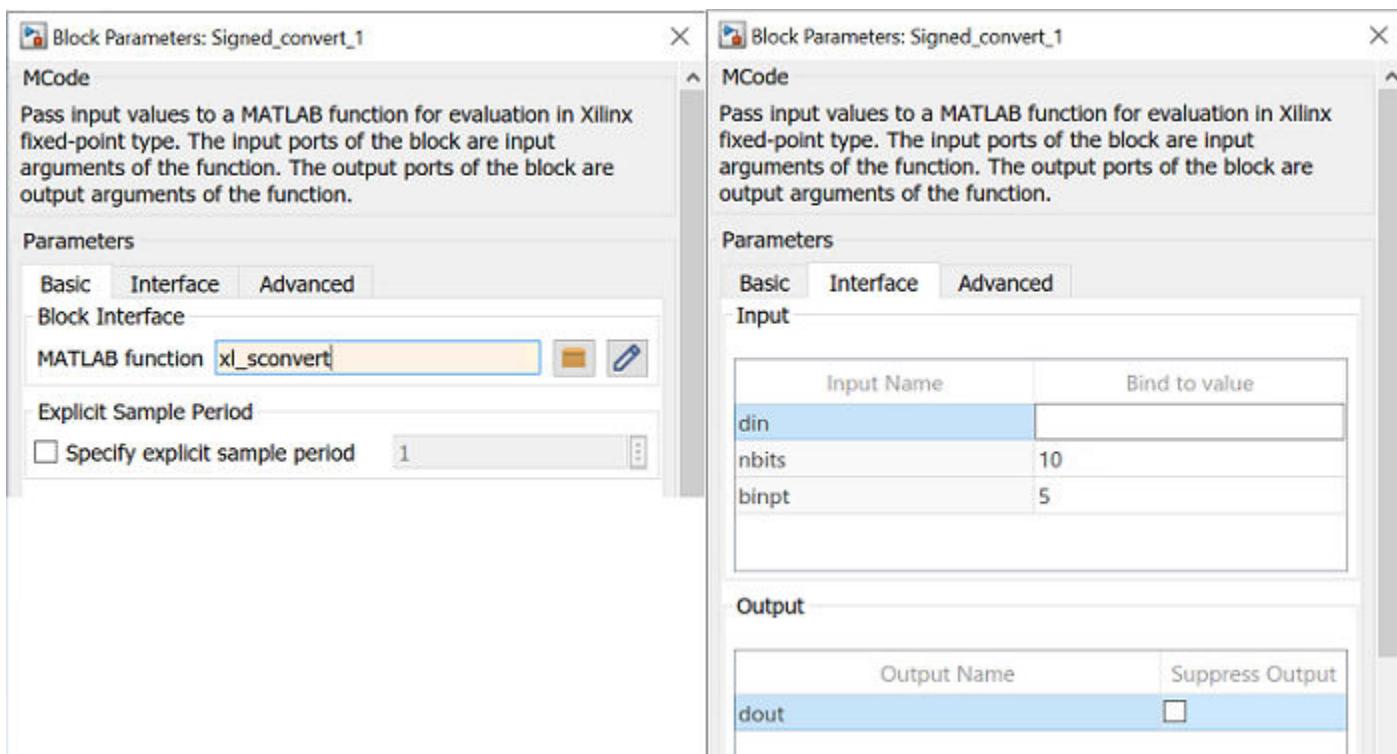
```
function dout = xl_sconvert(din, nbits, binpt)
  proto = {xlSigned, nbits, binpt};
  dout = xfix(proto, din);
```

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

**Figure 28: Subsystem with Two MCode Blocks**

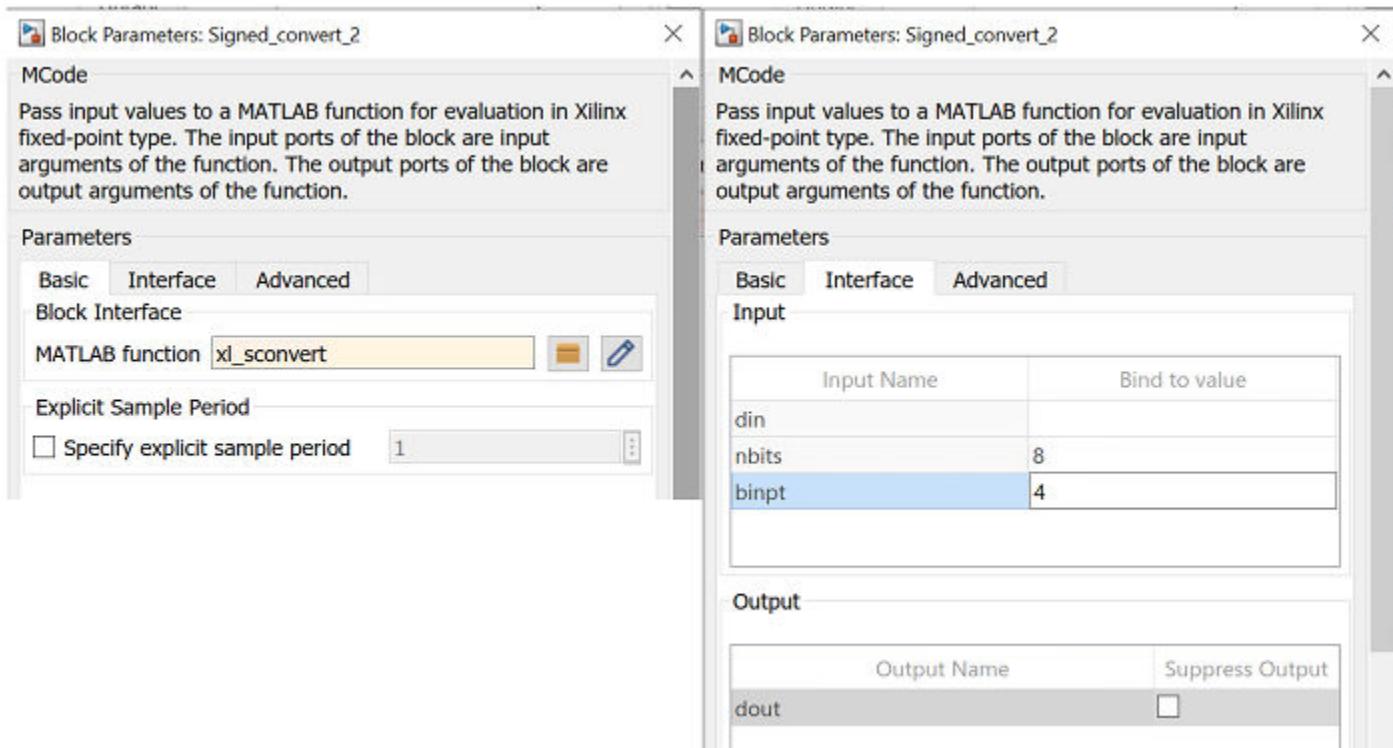
To pass parameters to each MCode block in the diagram above, you can click the **Edit M-File** button on the block GUI then set the values for the M-function arguments. The mask for MCode block `signed convert 1` is shown below:

Figure 29: Masking MCode Block



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

Figure 30: Mask for MCode Block Signed Convert 2



## Optional Input Ports

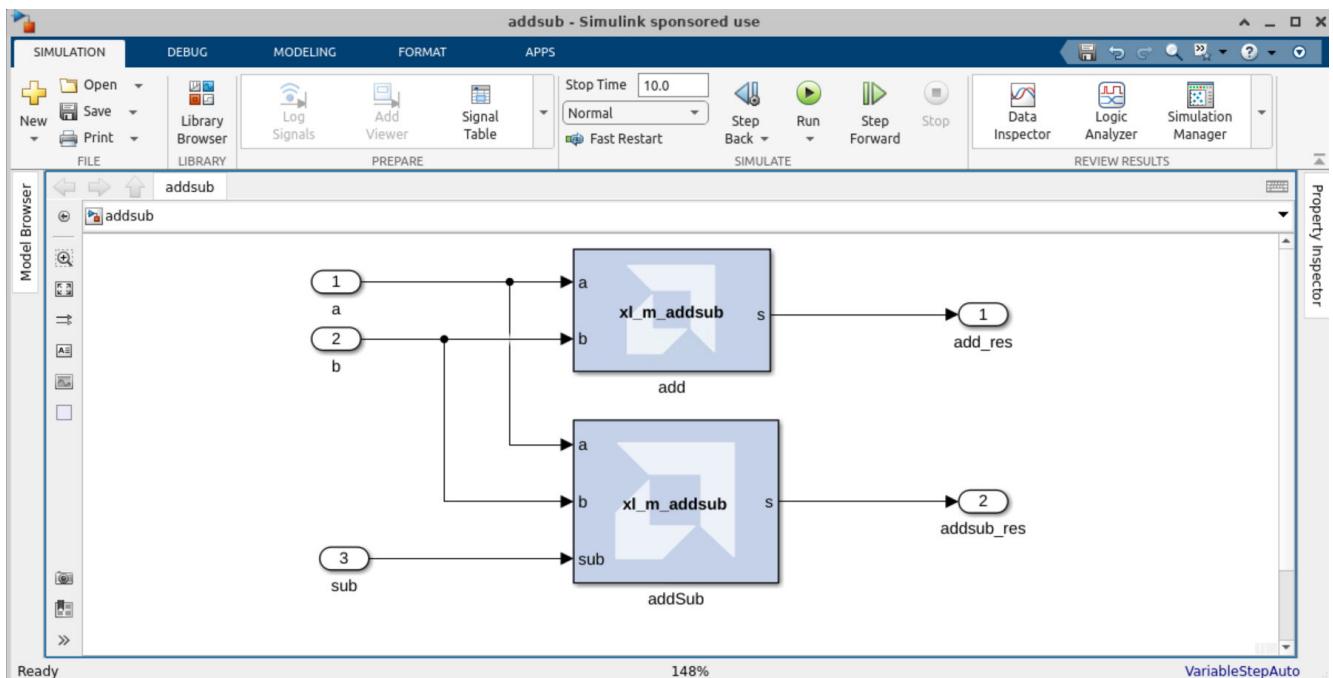
This example shows how to use the parameter passing mechanism of MCode blocks to specify whether or not to use optional input ports on MCode blocks.

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

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

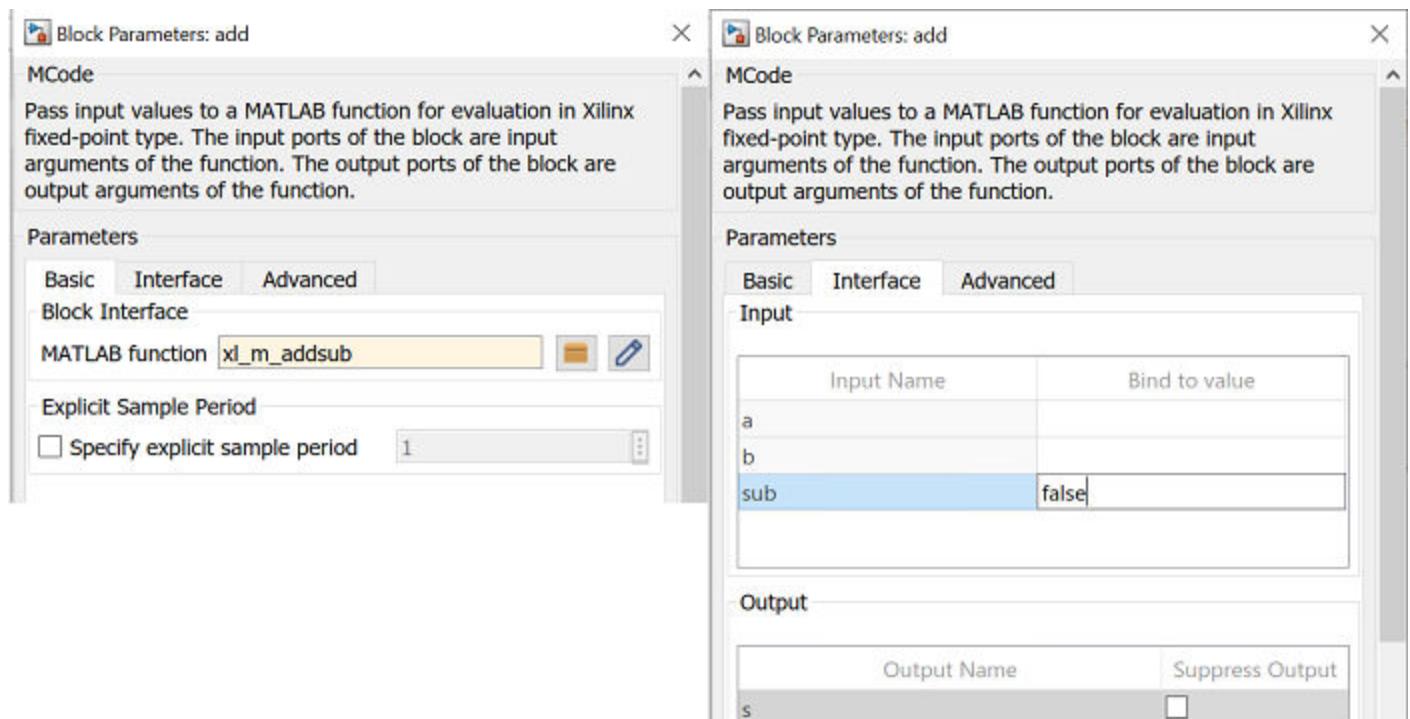
The following diagram shows a Subsystem containing two MCode blocks that use M-function `xl_m_addsub`.

Figure 31: Two MCode Blocks Using M-Function



The labeled add is shown in below.

Figure 32: Block Interface Editor of the MCode Block

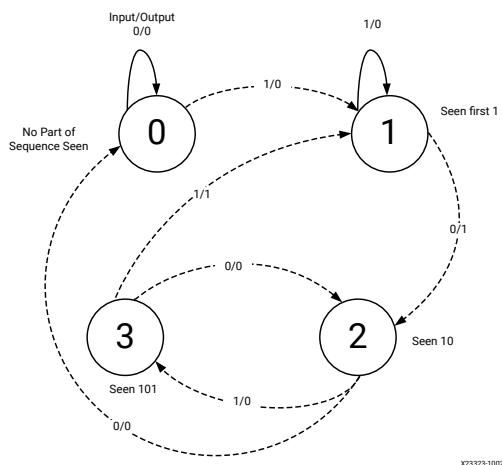


As a result, the `add` block features two input ports `a` and `b`; it performs full precision addition. Input parameter `sub` of the MCode block labeled `addsub` is not bound with any value. Consequently, the `addsub` block features three input ports: `a`, `b`, and `sub`; it performs full precision addition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

## Finite State Machines

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

**Figure 33: Finite State Machine Diagram**



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

```

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

seen_none = 0; % initial state, if input is 1, switch to seen_1
seen_1 = 1; % first 1 has been seen, if input is 0, switch
% seen_10
seen_10 = 2; % 10 has been detected, if input is 1, switch to
% seen_1011
seen_101 = 3; % now 101 is detected, 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, {xlUnsigned, 2, 0});

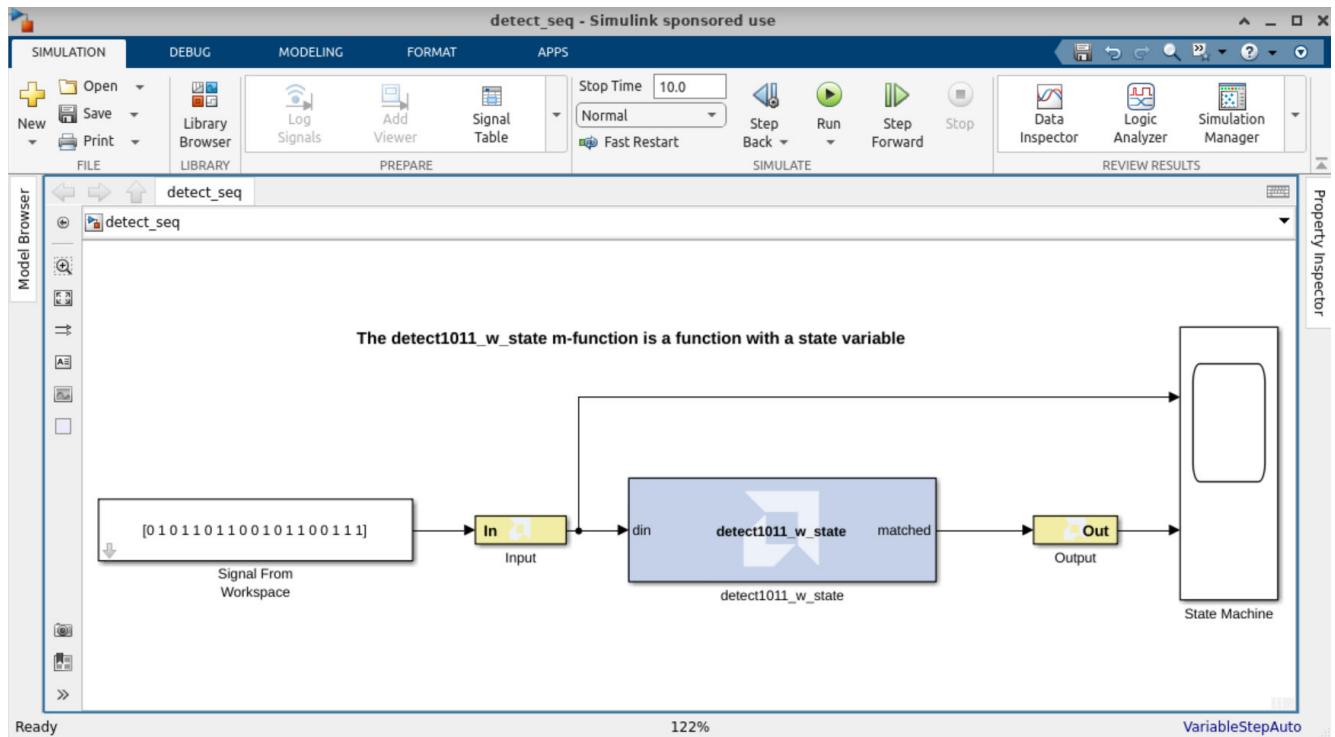
% the default value of matched is false
matched = false;

```

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

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

Figure 34: Subsystem Containing MCode Block After Compilation



## Parameterizable Accumulator

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

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

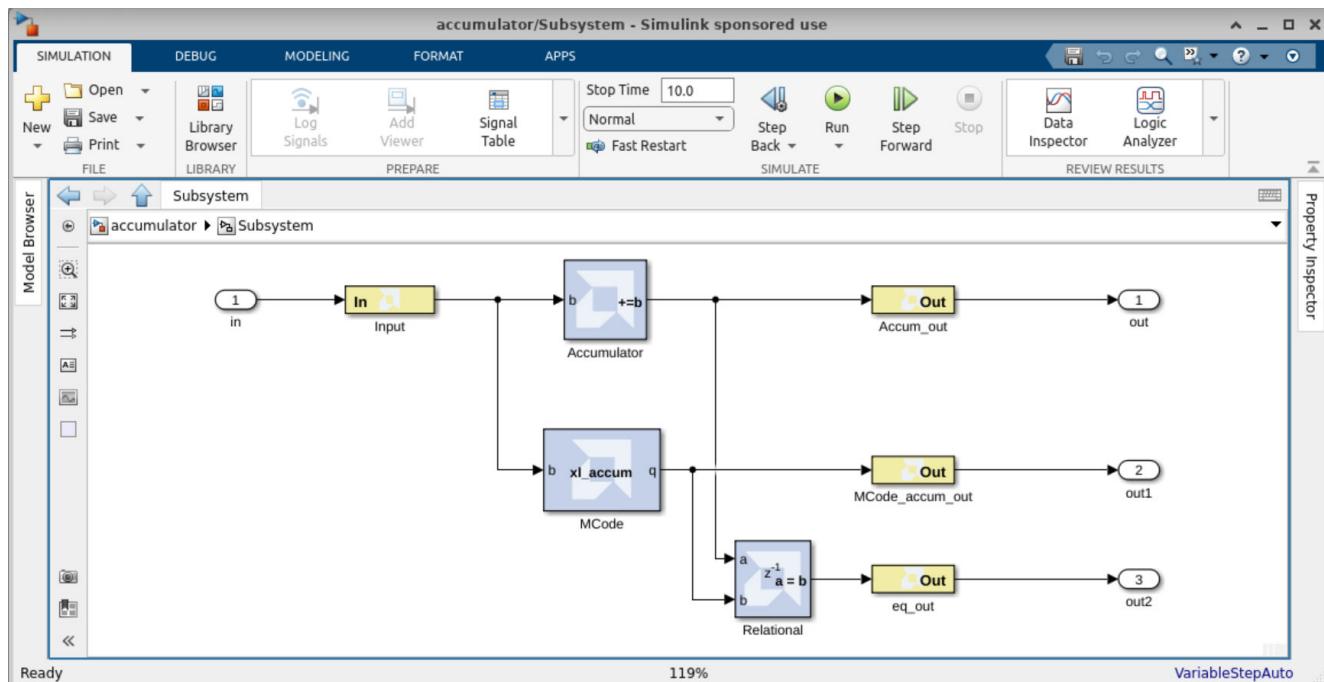
```

else
    s = s/feed_back_down_scale - b;
end
end

```

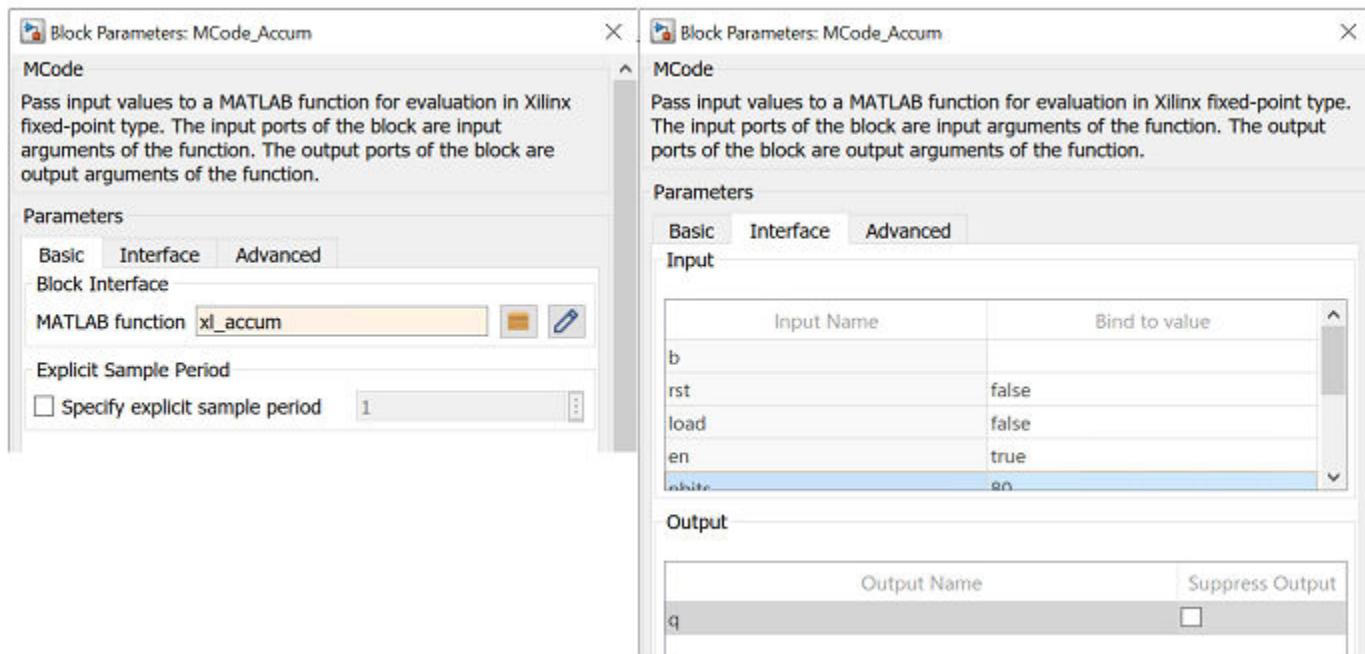
The following diagram shows a Subsystem containing the accumulator MCode block using M-function `x1_accum`. The MCode block is labeled MCode Accumulator. The Subsystem also contains the AMD Accumulator block, labeled Accumulator, for comparison purposes. The MCode block provides the same functionality as the AMD Accumulator block; however, its mask interface differs in that parameters of the MCode block are specified with a cell array in the Function Parameter Bindings parameter.

*Figure 35: MCode Accumulator*



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

Figure 36: Mask for MCode Accumulator

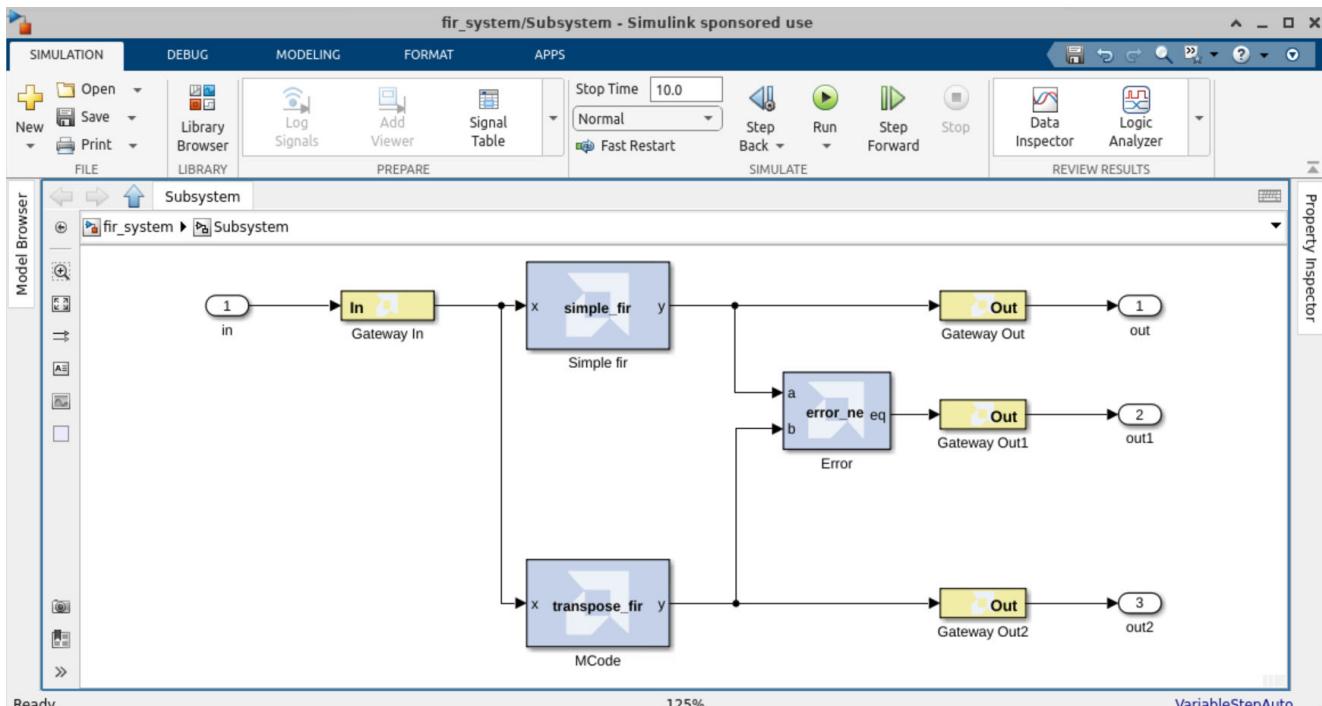


The example contains two additional accumulator Subsystems with MCode blocks using the same M-function, but different parameter settings to accomplish different accumulator implementations.

## FIR Example and System Verification

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

Figure 37: FIR Example



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

```

function y = simple_fir(x, lat, coefs, len, c_nbBits, c_binpt, o_nbBits,
o_binpt)
    coef_prec = {xlSigned, c_nbBits, c_binpt, xlRound, xlWrap};
    out_prec = {xlSigned, o_nbBits, 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_nbBits, c_binpt, o_nbBits,
o_binpt)
    coef_prec = {xlSigned, c_nbBits, c_binpt, xlRound, xlWrap};
    out_prec = {xlSigned, o_nbBits, o_binpt};
    coefs_xfix = xfix(coef_prec, coefs);
    persistent coef_vec, coef_vec = xl_state(coefs_xfix, coef_prec);
    persistent reg_line, reg_line = xl_state(zeros(1, len), out_prec);
    if lat <= 0
        error('latency must be at least 1');
    end
    lat = lat - 1;

```

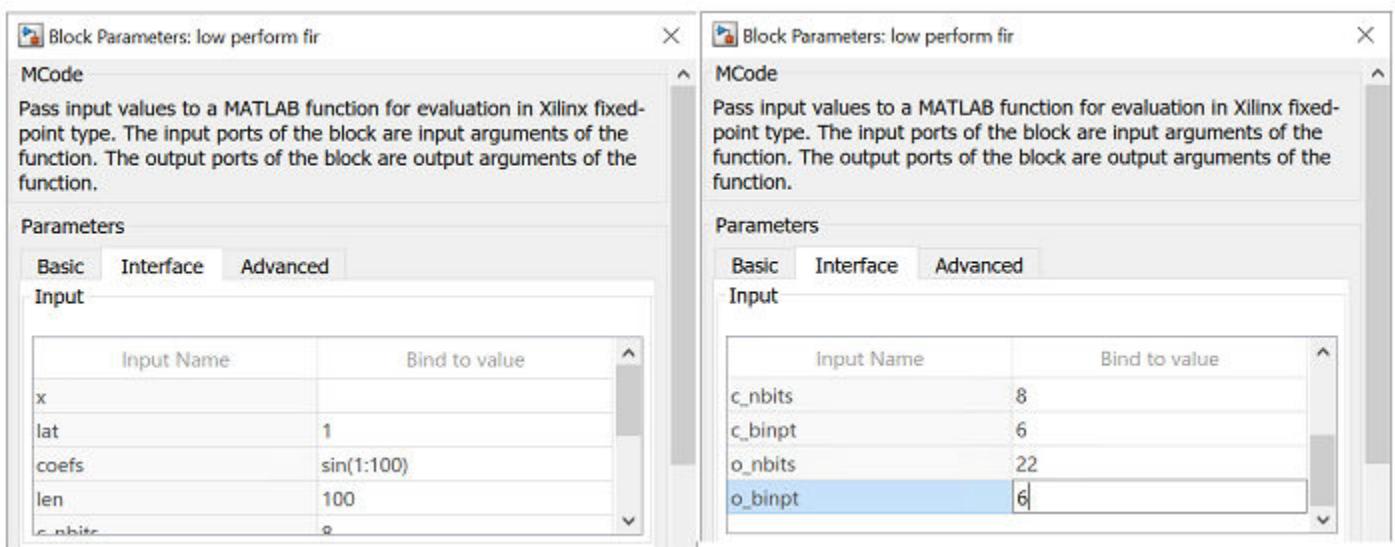
```

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

```

The parameters are configured as following:

*Figure 38: Parameters*



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

```

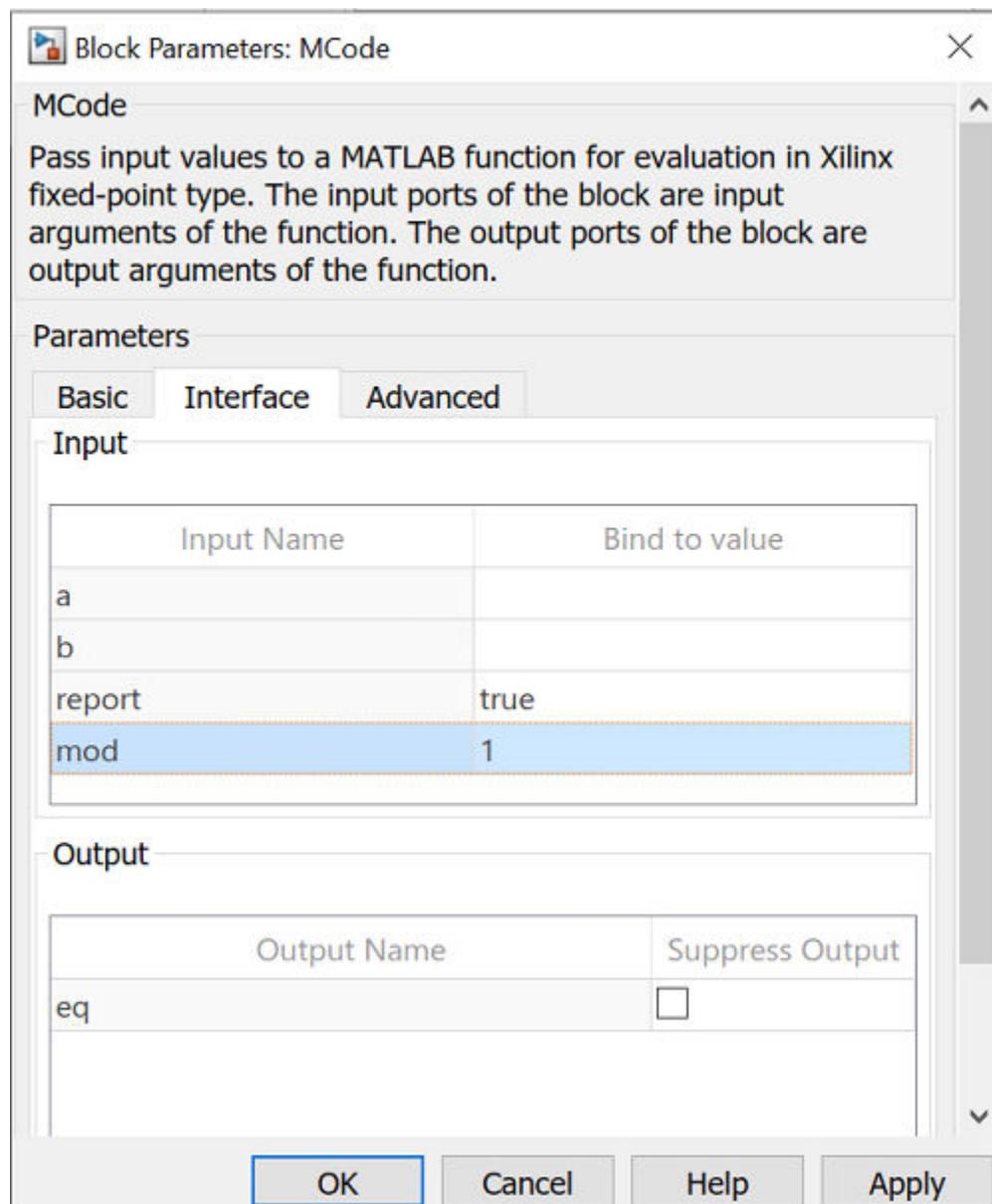
function eq = error_ne(a, b, report, mod)
    persistent cnt, cnt = xl_state(0, {xlUnsigned, 16, 0});
    switch mod
        case 1
            eq = a == b;
        case 2
            eq = isnan(a) || isnan(b) || a == b;
        case 3
            eq = ~isnan(a) && ~isnan(b) && a == b;
        otherwise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

```

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

The block is configured as following:

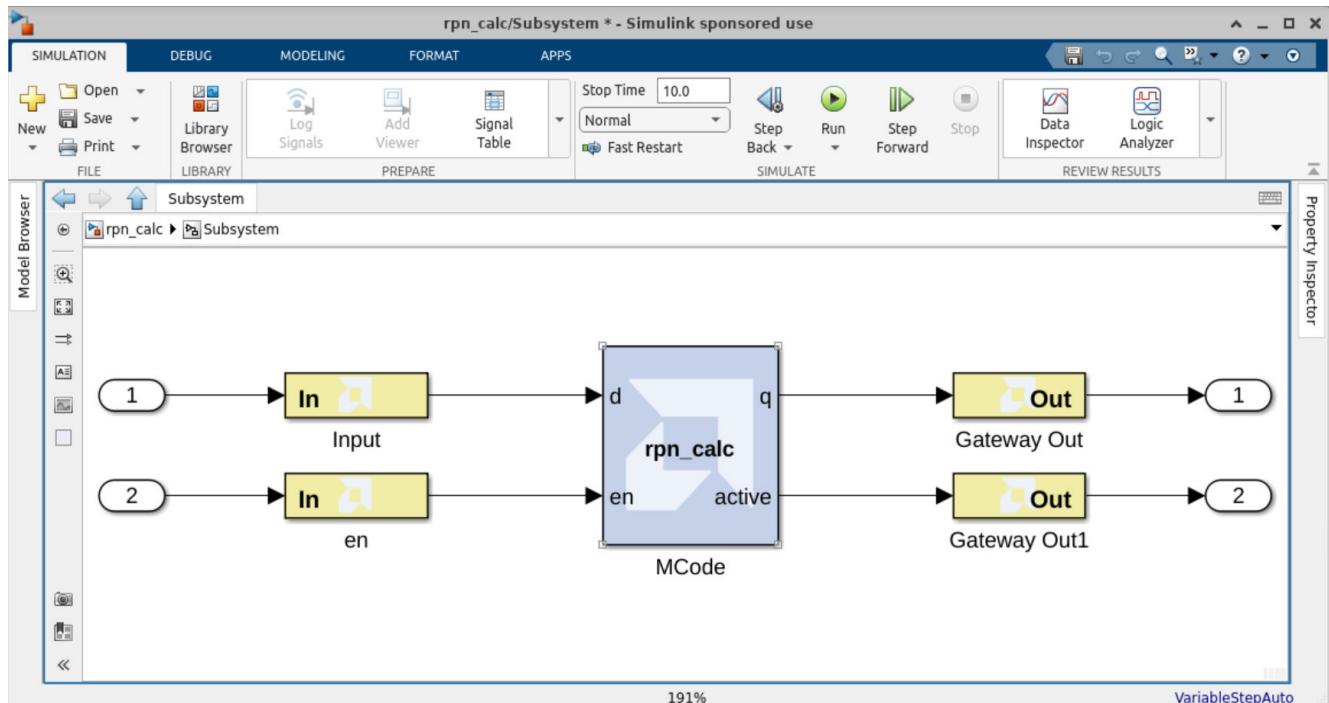
Figure 39: Block Configuration



## RPN Calculator

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

Figure 40: RPN Calculator



The following function models the RPN calculator.

```
function [q, active] = rpn_calc(d, rst, en)
d_nbBits = xl_nbBits(d);
% the first bit indicates whether it's a data or operator
is_oper = xl_slice(d, d_nbBits-1, d_nbBits-1)==1;
din = xl_force(xl_slice(d, d_nbBits-2, 0), xlSigned, 0);
% the lower 3 bits are operator
op = xl_slice(d, 2, 0);
% acc the the A register
persistent acc, acc = xl_state(0, din);
% the stack is implemented with a RAM and
% an up-down counter
persistent mem, mem = xl_state(zeros(1, 64), din);
persistent acc_active, acc_active = xl_state(false, {xlBoolean});
persistent stack_active, stack_active = xl_state(false, ...
{xlBoolean});
stack_pt_prec = {xlUnsigned, 5, 0};
persistent stack_pt, stack_pt = xl_state(0, {xlUnsigned, 5, 0});
% when en is true, it's action
OP_ADD = 2;
OP_SUB = 3;
OP_MULT = 4;
OP_NEG = 5;
OP_DROP = 6;
q = acc;
```

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

## Example of disp Function

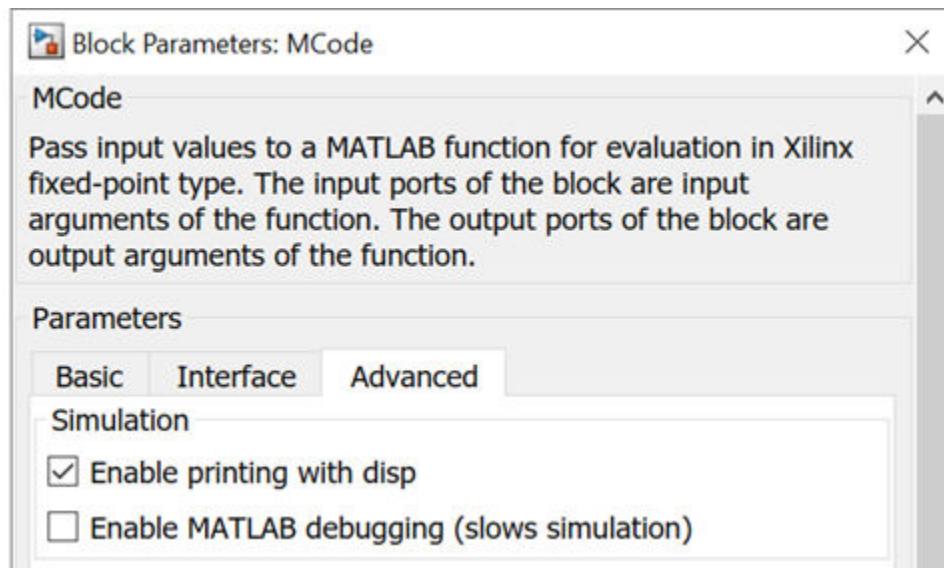
The following MCode function shows how to use the disp function to print variable values.

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

```
disp('disp(10) is');
disp(10);
disp('disp(-10) is');
disp(-10);
disp('disp(a) is ');
disp(a);
disp('disp(a == b)');
disp(a==b);
```

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

Figure 41: Enable Printing with disp



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

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

```
a = 0.000000, b = 0.000000, x = 0.000000
1
disp(10) is
    type: UFix_4_0, binary: 1010, double: 10.0
disp(-10) is
    type: Fix_5_0, binary: 10110, double: -10.0
disp(a) is
    type: Fix_11_7, binary: 0000.0000000, double: 0.000000
disp(a == b)
    type: Bool, binary: 1, double: 1
```

## Importing a Model Composer HDL Design into a Bigger System

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

### *HDL Netlist Compilation*

Selecting the HDL Netlist compilation target from the Vitis Model Composer Hub block instructs Model Composer to generate HDL along with other related files that implement the design. In addition, Vitis Model Composer produces auxiliary files that simplify downstream processing such as simulating the design using an Vivado simulator, and performing logic synthesis using Vivado synthesis. See [Compilation Types for HDL Library Designs](#) for more details.

### *Integration Design Rules*

When a Vitis Model Composer model is to be included into a larger design, the following two design rules must be followed.

- **Rule 1:** No Gateway should specify an IOB/CLK location.  
IOB timing constraints should be set to: none.
- **Rule 2:** If there are any I/O ports from the Model Composer design that are required to be ports on the top-level design, appropriate buffers should be instantiated in the top-level HDL code.

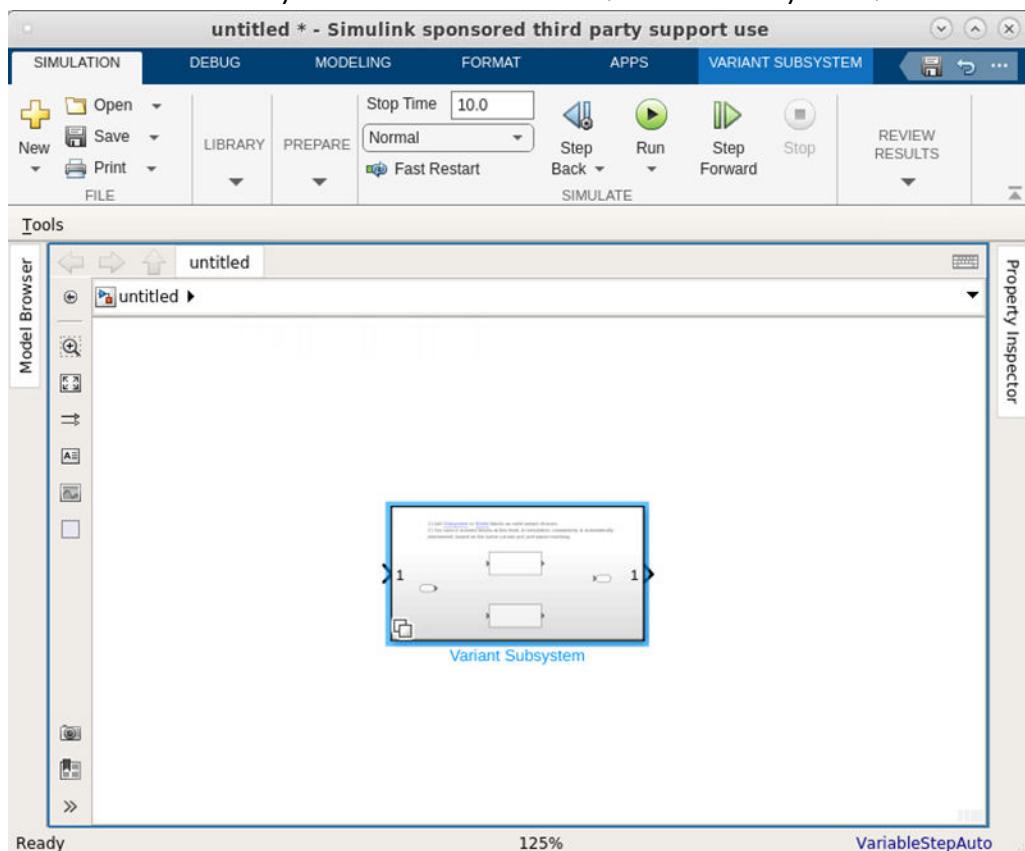
## Variant Subsystems and Vitis Model Composer

A variant Subsystem is a type of block that is made available as a standard part of Simulink. In effect, a variant 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 Vitis Model Composer you might, for example, specify a general-purpose FIR filter as a variant 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.

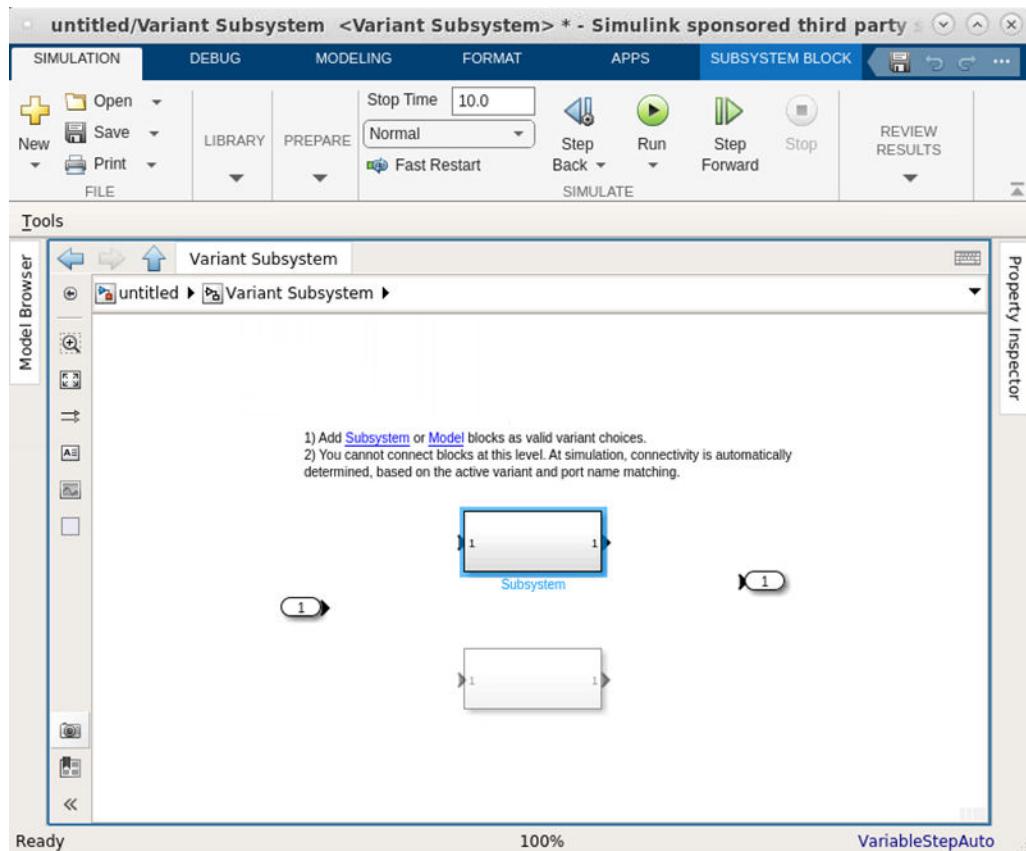
### Defining a Variant Subsystem

A variant subsystem can be created by dragging a Variant Subsystem block from the Library Browser into your model. This block will then contain underlying blocks to implement each variant. To create a variant subsystem, do the following:

1. Drag a Variant Subsystem block into your model. (The Variant Subsystem block can be found in the Simulink Library Browser under Simulink/Ports & Subsystems/Variant Subsystem.)

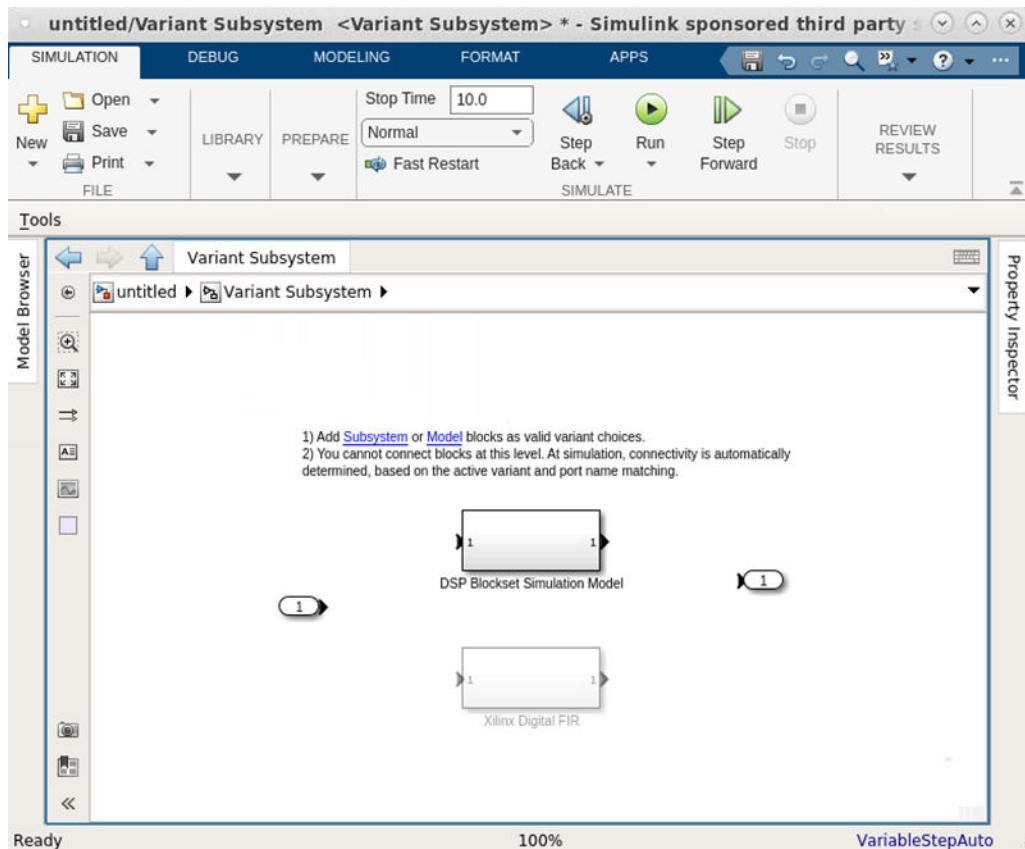


2. Double-click the Variant Subsystem block.



3. Each variant choice is defined in a Subsystem or Model block within the Variant Subsystem. You can define each variant choice's implementation by populating the empty Subsystem blocks in the model, or by copying Subsystem or Model blocks in from elsewhere. Rename each Subsystem block to describe which variant selection it represents.

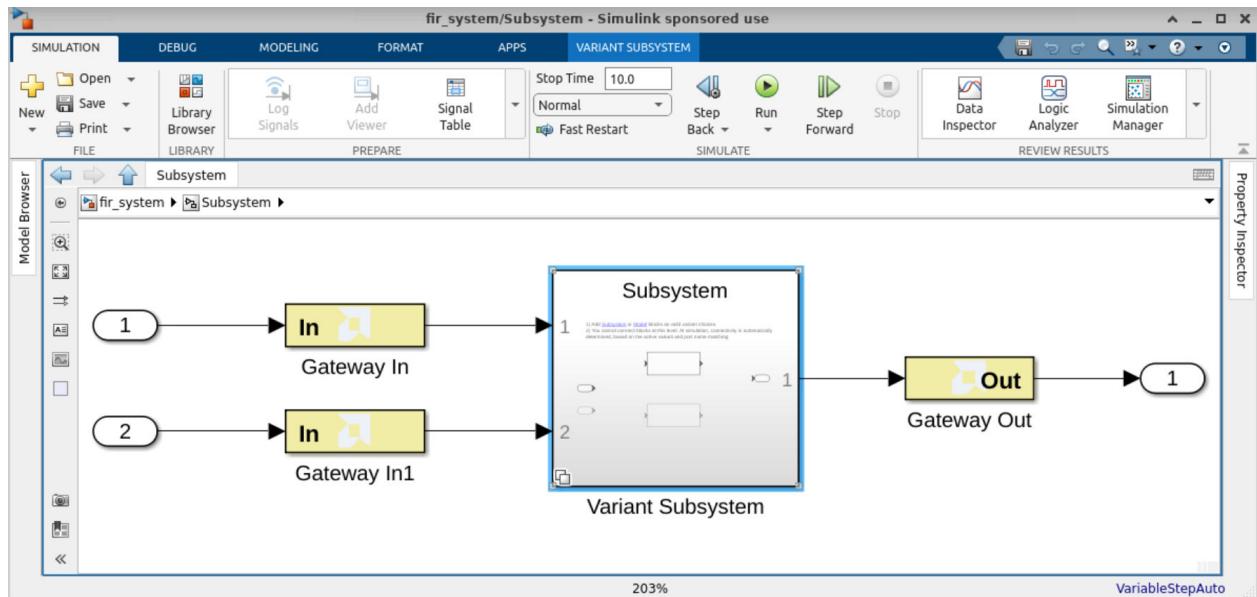
**Note:** The Variant Subsystem block inputs and outputs should not be connected; the port connectivity will be determined at runtime based on which variant is active.



## Using a Variant Subsystem

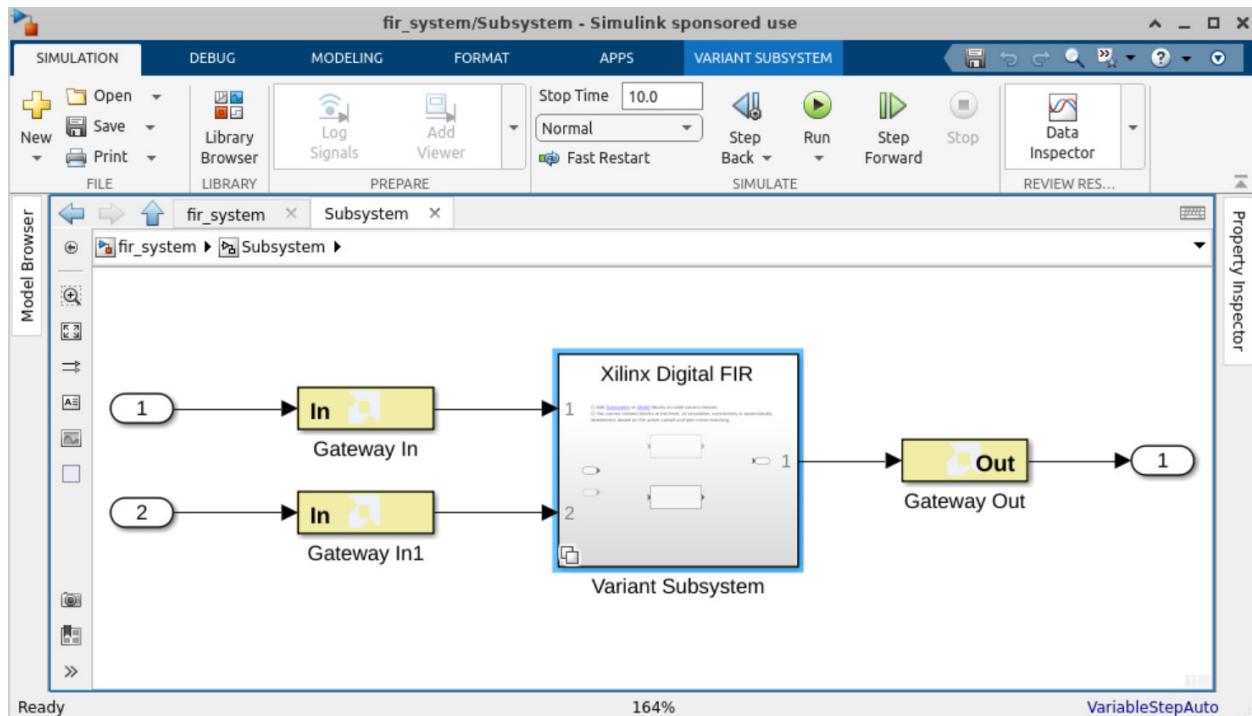
To use a Variant Subsystem in a design after its variant selections have been implemented, do the following:

1. Create the variant subsystem and insert it into the design at the appropriate location.

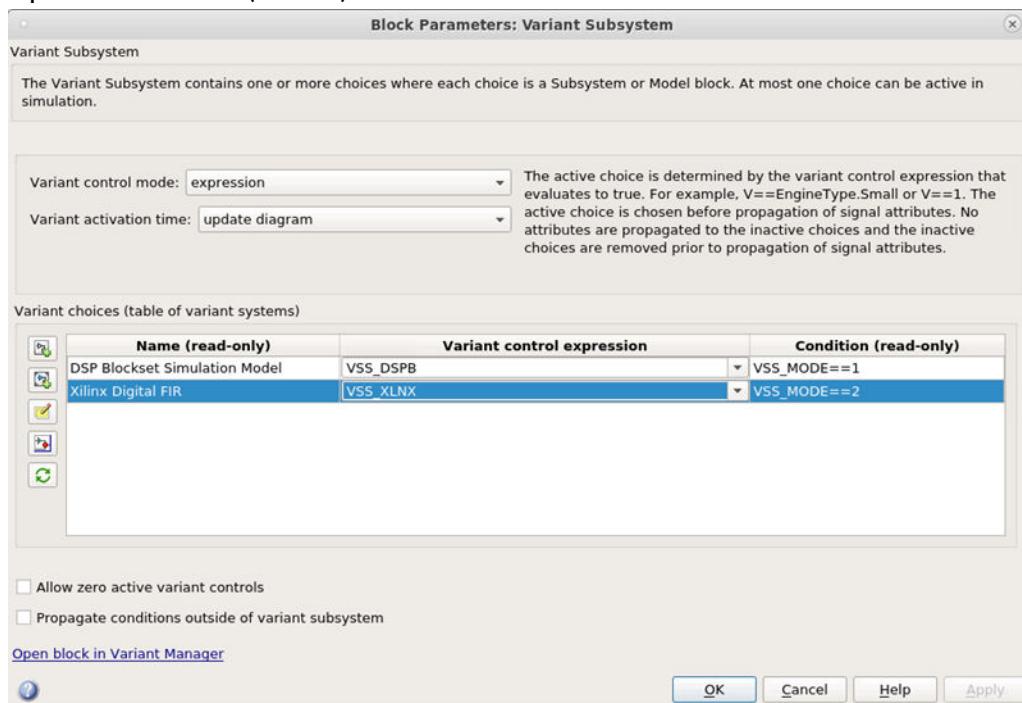


2. Right-click the variant block and select **Block Parameters**. The Block Parameters dialog box for the variant block opens.
3. To select the active Variant based on the evaluation of the Variant conditions, use the expression mode. Otherwise, select **label** mode. When you select the Variant control mode as label, the Label mode active choice option becomes available. In label mode, Variant control need not be created in the global workspace. You can select an active Variant choice from Label mode active choice options.
4. Use the options available on the Block Parameter dialog box to add variant controls and corresponding variant conditions.

The following figure shows an example configuration for expression mode:



5. To activate a variant selection, type the variant name in the MATLAB® command window. For example, type `VSS_MODE = 2.`
6. Update the model (Ctrl+D) to ensure that the variant selection has been activated.



## Notes for Higher Performance FPGA Design

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

- The more complex IP blocks in a Model Composer 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 might not be possible.
- Model Composer 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 Model Composer to increase the performance of your design before you start the implementation process.

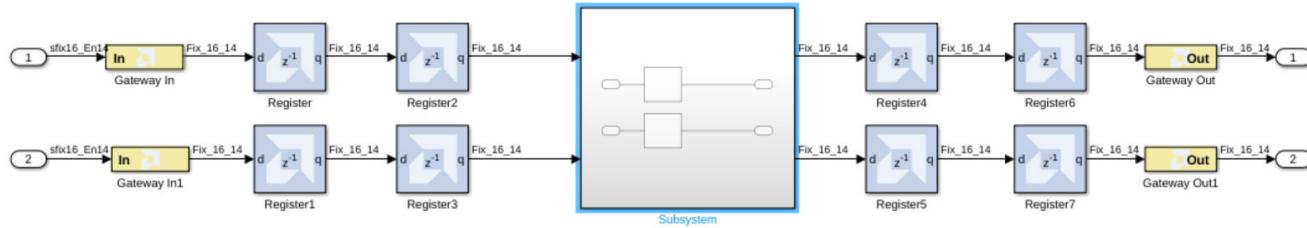
- [Review the Hardware Notes Included with Each Block Dialog Box](#)
- [Register the Inputs and Outputs of Your Design](#)
- [Insert Pipeline Registers](#)
- [Use Saturation Arithmetic and Rounding Only When Necessary](#)
- [Set the Data Rate Option on All Gateway Blocks](#)
- [Pipeline for Maximum Performance](#)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

### ***Review the Hardware Notes Included with Each Block Dialog Box***

Pay close attention to the Hardware Notes included in the block dialog boxes. Many blocks in the AMD 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 resources.

### ***Register the Inputs and Outputs of Your Design***

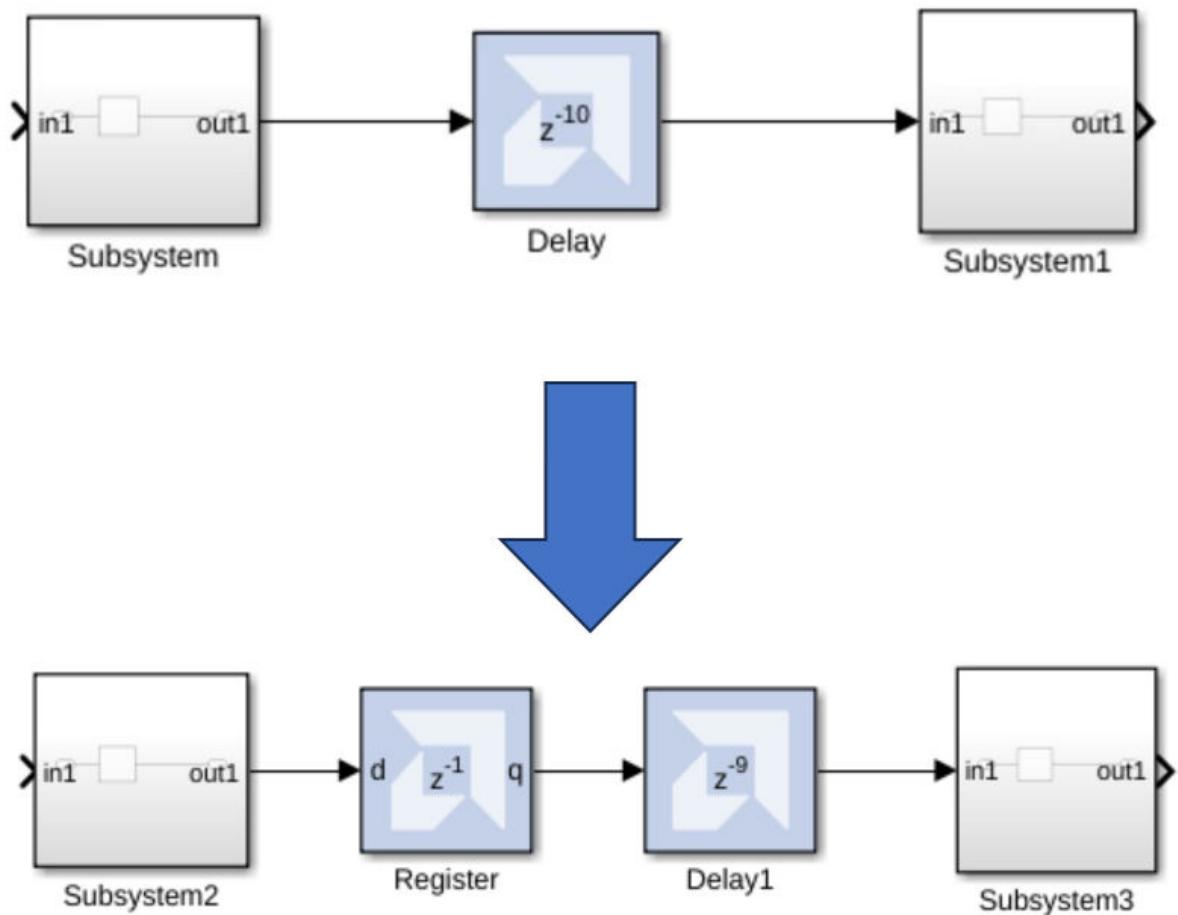
Register the inputs and outputs of your design. As shown below, this can be done by placing one or more Delay blocks with a latency 1 or Register blocks after the Gateway In and before Gateway Out blocks. Selecting any of the Register block features adds hardware.

**Figure 42: Register Inputs and Outputs**

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

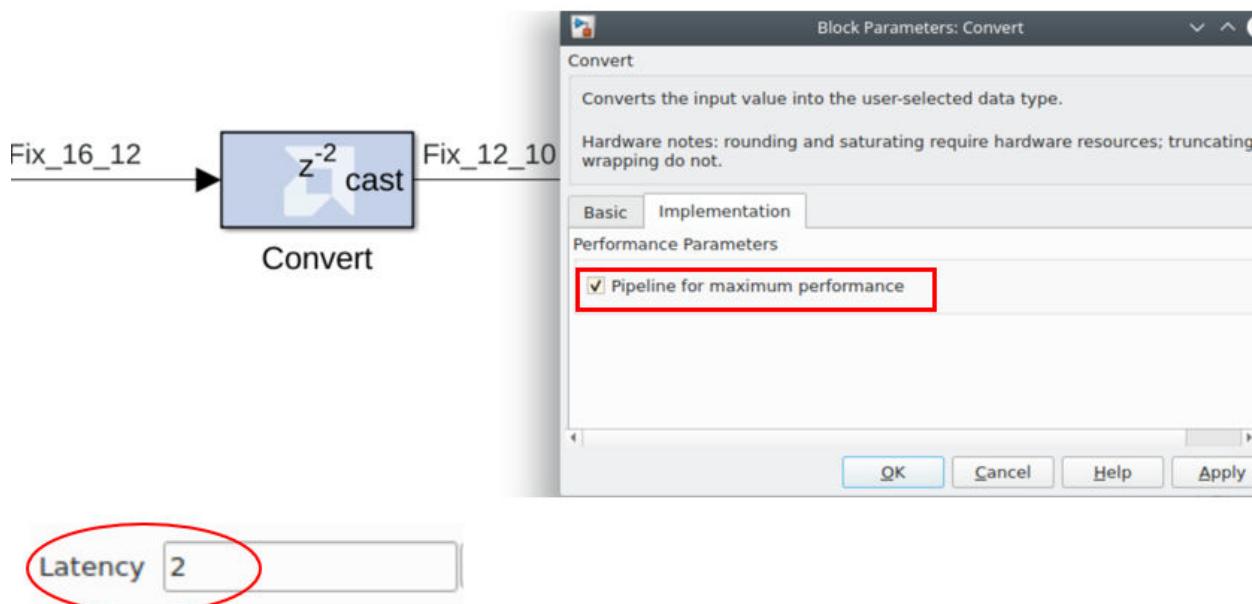
## Insert Pipeline Registers

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

**Figure 43: Pipeline Registers**

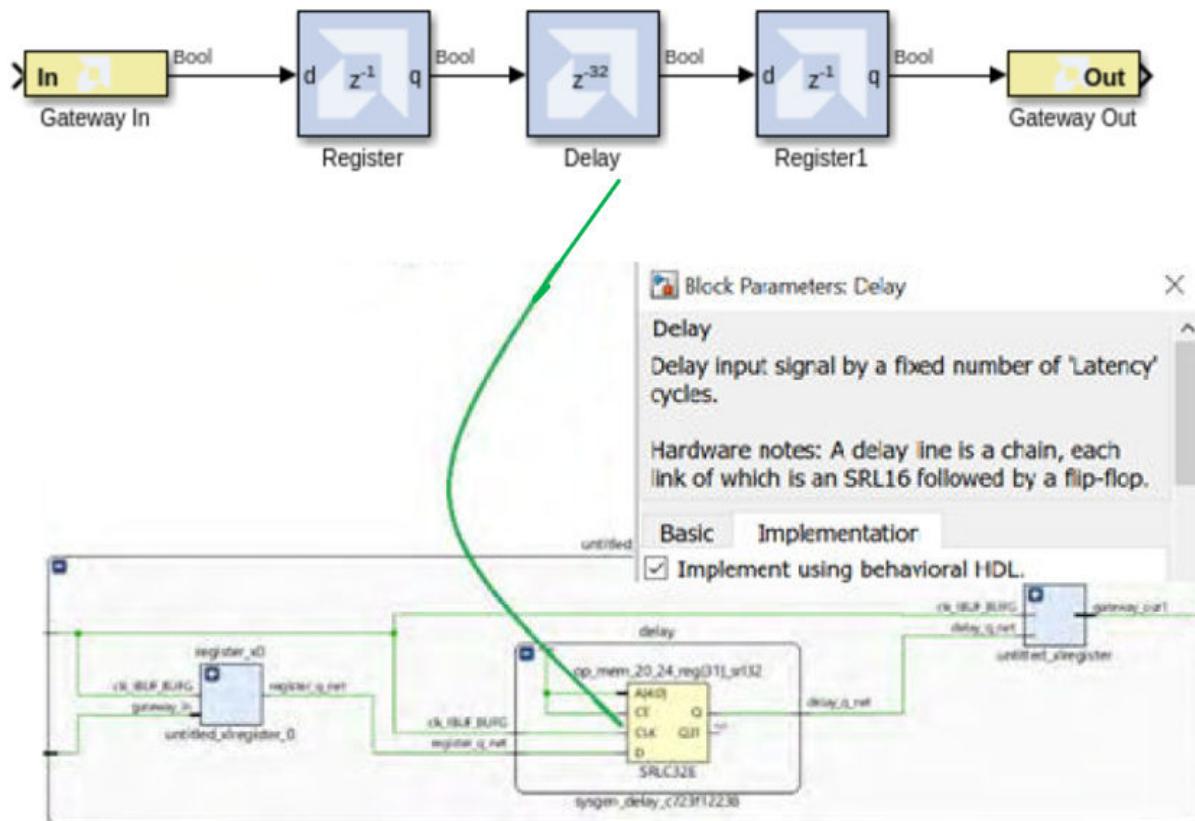
As shown in the following figure, the Convert block can be pipelined with embedded register stages to guarantee maximum performance.

Figure 44: Convert Block



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

Figure 45: Implement Using Behavioral HDL



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

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

### ***Use Saturation Arithmetic and Rounding Only When Necessary***

Saturation arithmetic and rounding have area and performance costs. Use only if necessary. For example a Reinterpret block does not cost any logic. A Convert (cast) block does not 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 Vitis Model Composer Hub block and the sample rate of the Gateway relative to the other sample periods in the design.

## ***Pipeline for Maximum Performance***

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

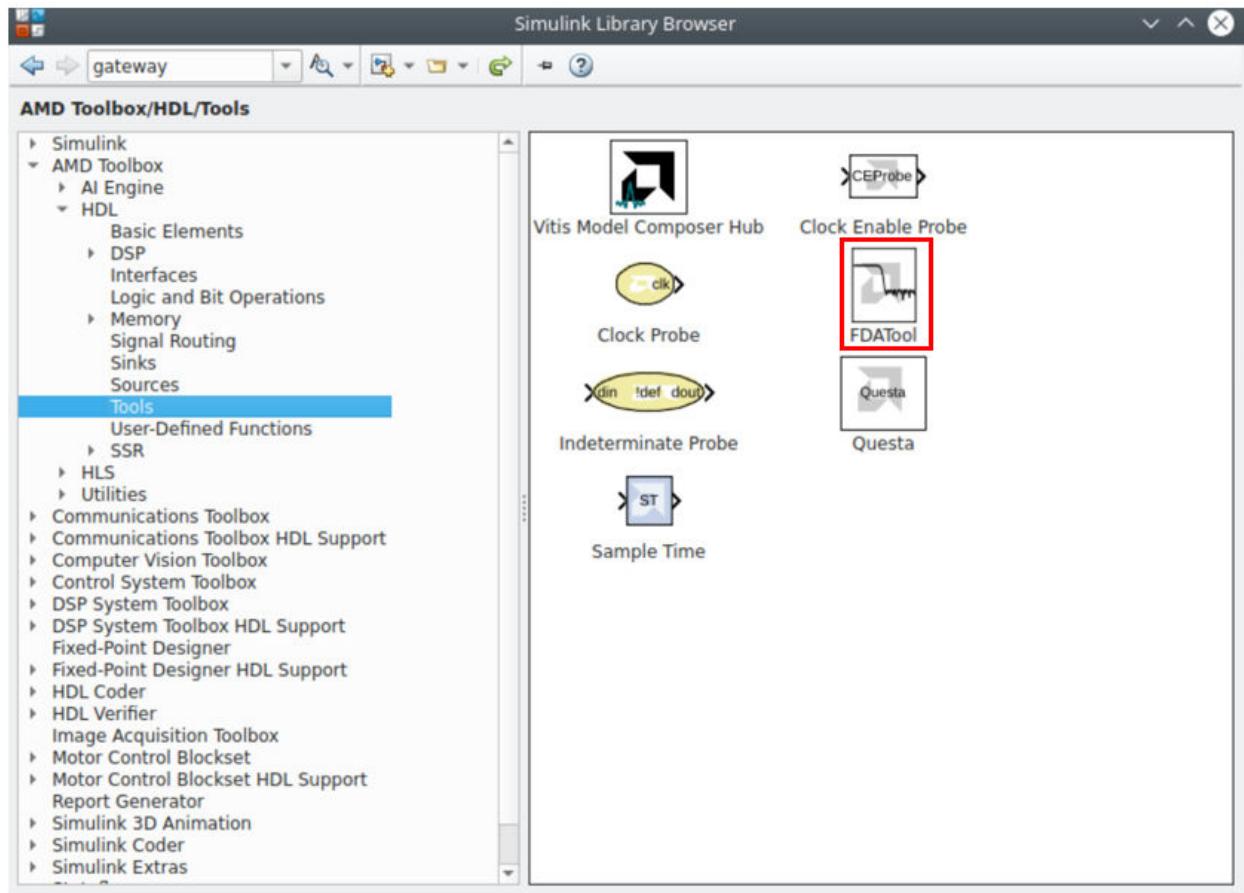
## ***Other Things to Try***

- Change the Source Design:
  - Use Additional Pipelining  
Use the Output and Pipeline registers inside block RAM and DSP48s.
  - Run Functions in Parallel  
Run functions in parallel at a slower clock rate
  - Use Retiming Techniques  
Move existing registers through combinational logic.
  - Use Hard Cores where Possible  
Use Block RAM instead of distributed RAM.
  - Use a Different Design Approach for Functions
- Avoid Over-Constraining the Design:  
Do not over-constrain the design and use up/down sample blocks where appropriate.
- Consider Decreasing the Frequency of Critical Design Modules
- Squeeze Out the Implementation Tools:
  - Try Different Synthesis Options.
  - Floorplan Critical Modules

## **Using the FDATool in Digital Filter Applications**

The FDATool block is used to define the filter order and coefficients, and the HDL blocks are used to implement a filter. The Tools library in the HDL Blockset contains the FDATool block.

Figure 46: FDATool Block

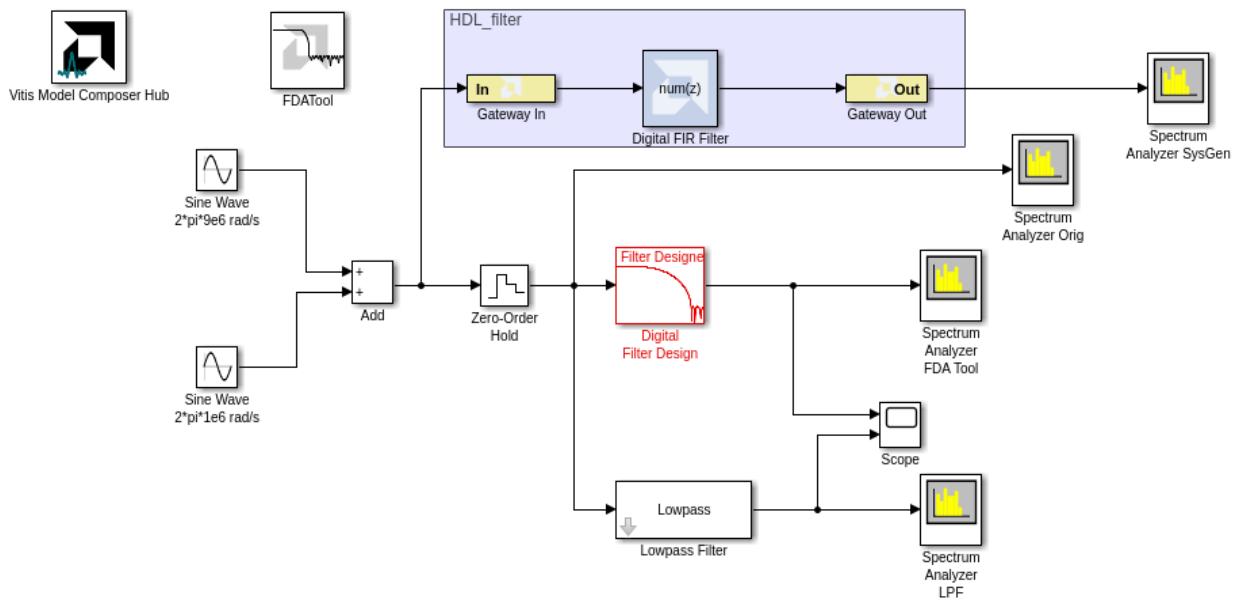


A simple Model Composer model below illustrates a standard FIR filter design using the FDATool and digital FIR filter block.

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

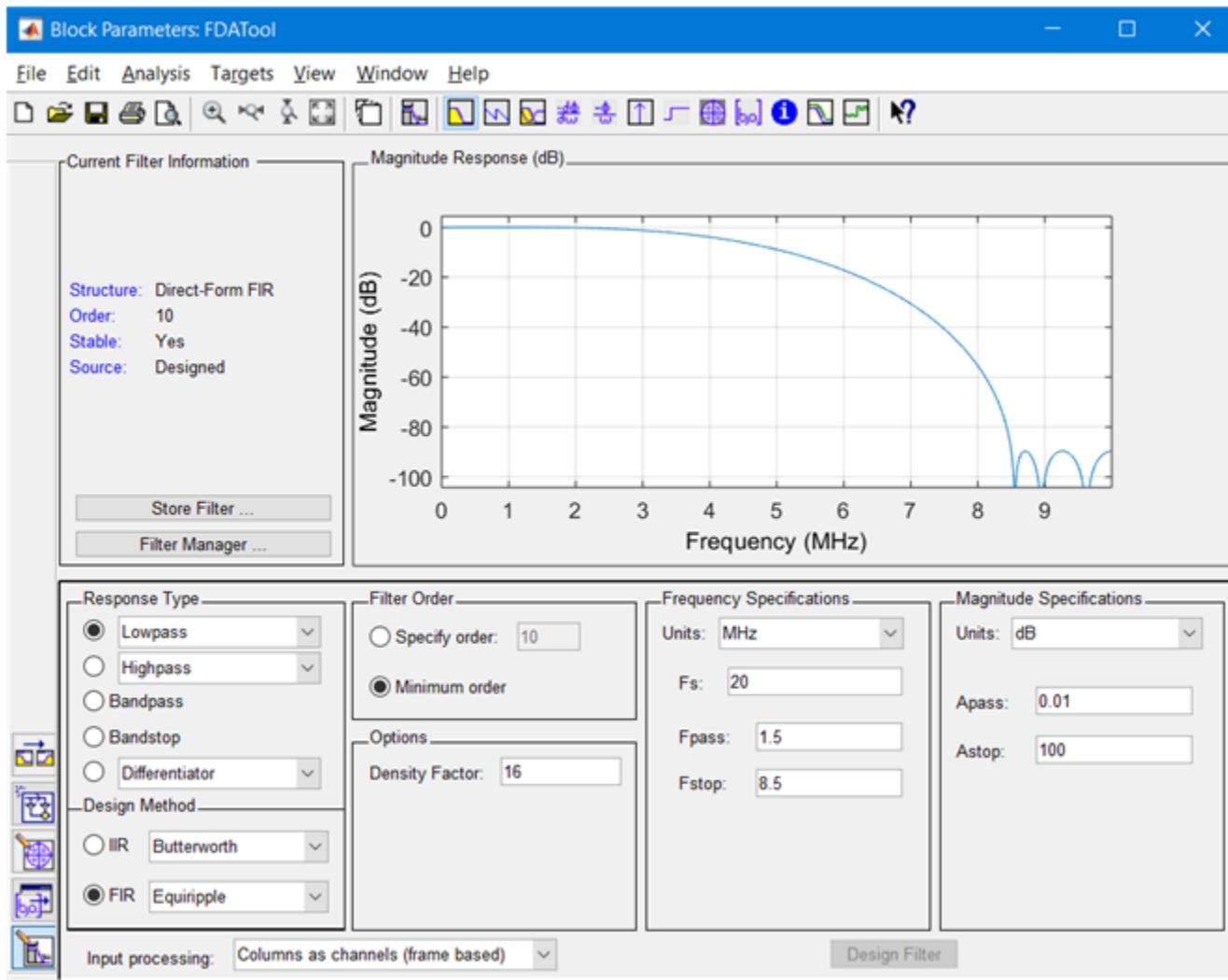
- The first filter is the one that could be implemented using the AMD HDL blockset. It is a digital low pass filter implemented using the Digital FIR filter block.
- The second filter is what is referred to as a reference filter. A low pass filter is implemented using a Direct-form FIR structure.

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

**Figure 47: Spectrum Analyzer Block**

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

Figure 48: Filter Configuration



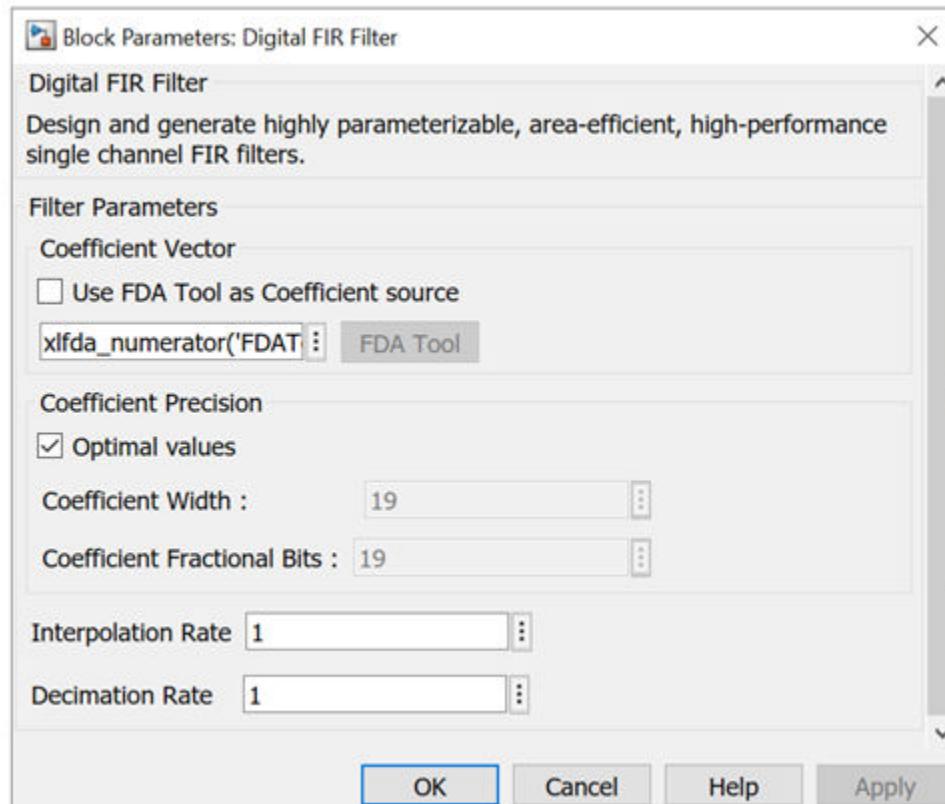
The Design Filter option at the bottom of the tool window allows you to find out the filter order and observe the Magnitude Response. You can also view the Phase Response, Impulse Response, Coefficients, and more by selecting the appropriate icon at the top-right of the window. You can display the filter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

```
>> xlfd_a_numerator('FDATool')
```

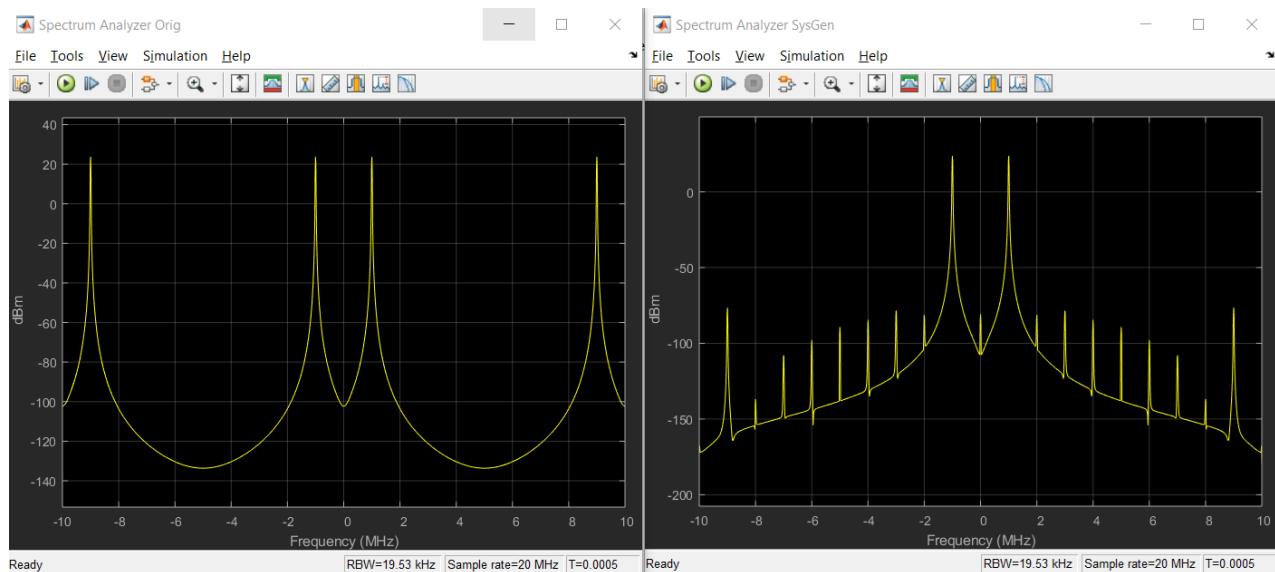
The following functions help you find the maximum and minimum coefficient values to adequately specify the coefficient width and binary point:

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

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

**Figure 49: Digital FIR Filter**

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

**Figure 50: Spectrum Analyzer**

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

For complete example along with steps to use the FDATool, refer to the [Vitis Model Composer Tutorials](#).

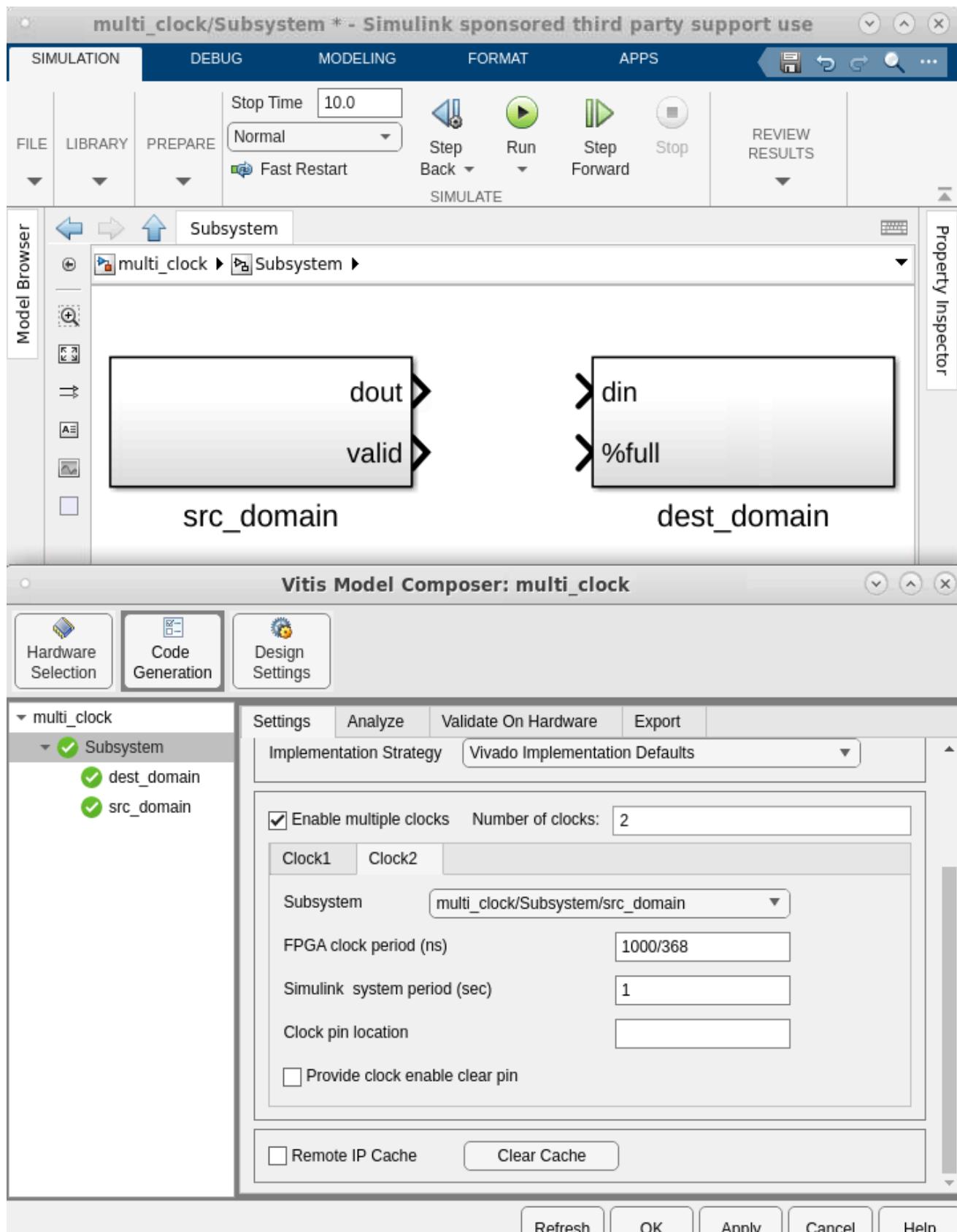
## Multiple Independent Clocks Hardware Design

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

### ***Grouping Blocks within a Clock Domain***

Blocks are grouped together in Vitis Model Composer by using a Subsystem. Subsystems can also be used to group blocks within a clock domain. Clock settings for each subsystem can then be controlled by enabling multiple clocks on the HDL Settings tab of the Vitis Model Composer Hub block.

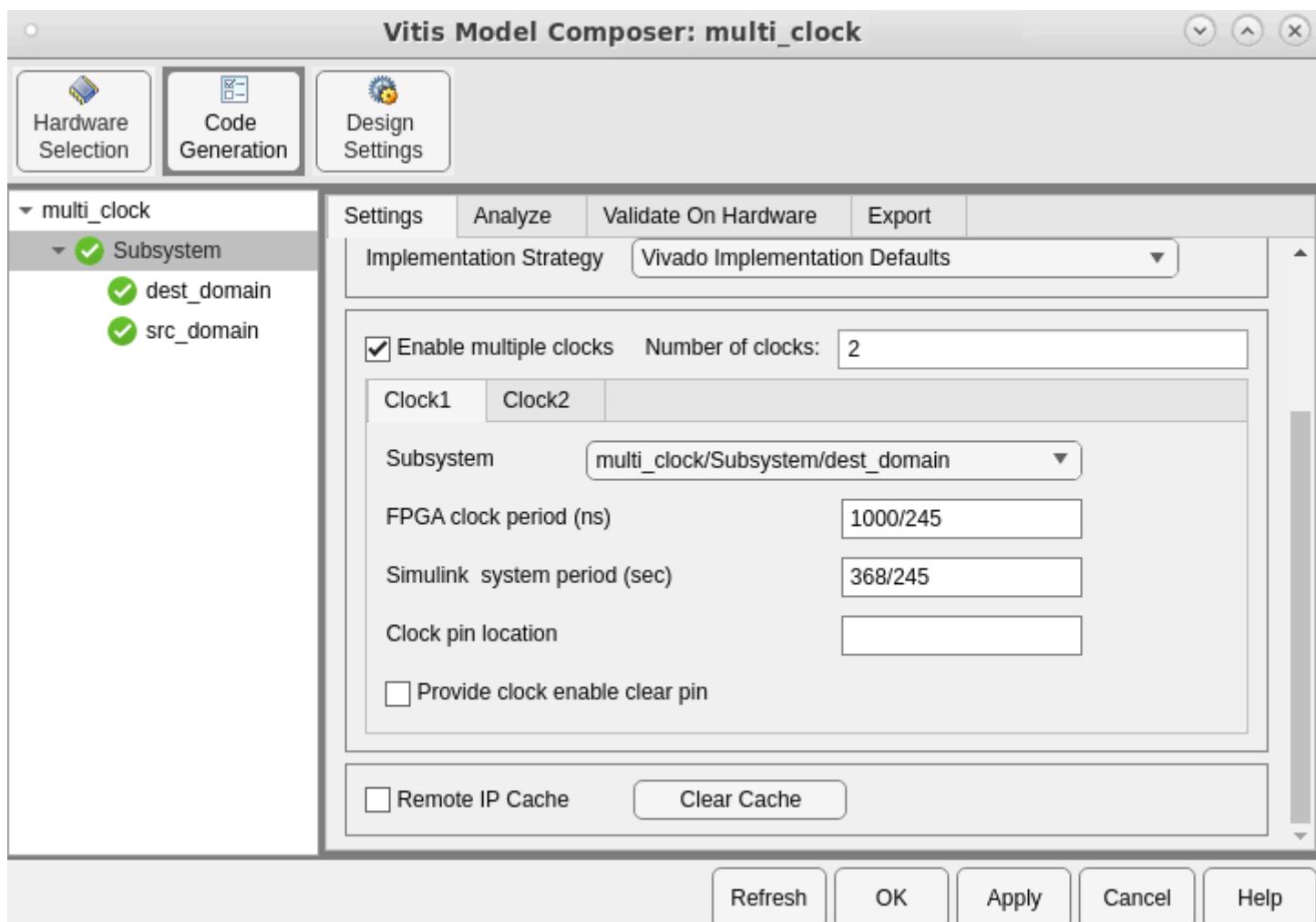
Figure 51: Source Clock Domain



In the previous figure, a clock domain Subsystem called `src_domain` has been created. On the HDL Settings tab, the FPGA clock period has been set to  $(1000/368)$  ns (368 MHz) and the Simulink system period to 1 sec. This implies that an advance of 1 Simulink second corresponds to  $(1000/368)$  ns of FPGA clock.

Similarly, another group of blocks representing another clock domain is included in a Subsystem called `dest_domain`. This Subsystem is configured to run at an FPGA clock period of  $1000/245$  ns (245 MHz). The Simulink system period is set to  $368/245$ . This is done because the Simulink system period of the `src_domain` Subsystem is set to 1. Hence, you normalize the System period from the faster `src_domain`.

Figure 52: Destination Clock Domain



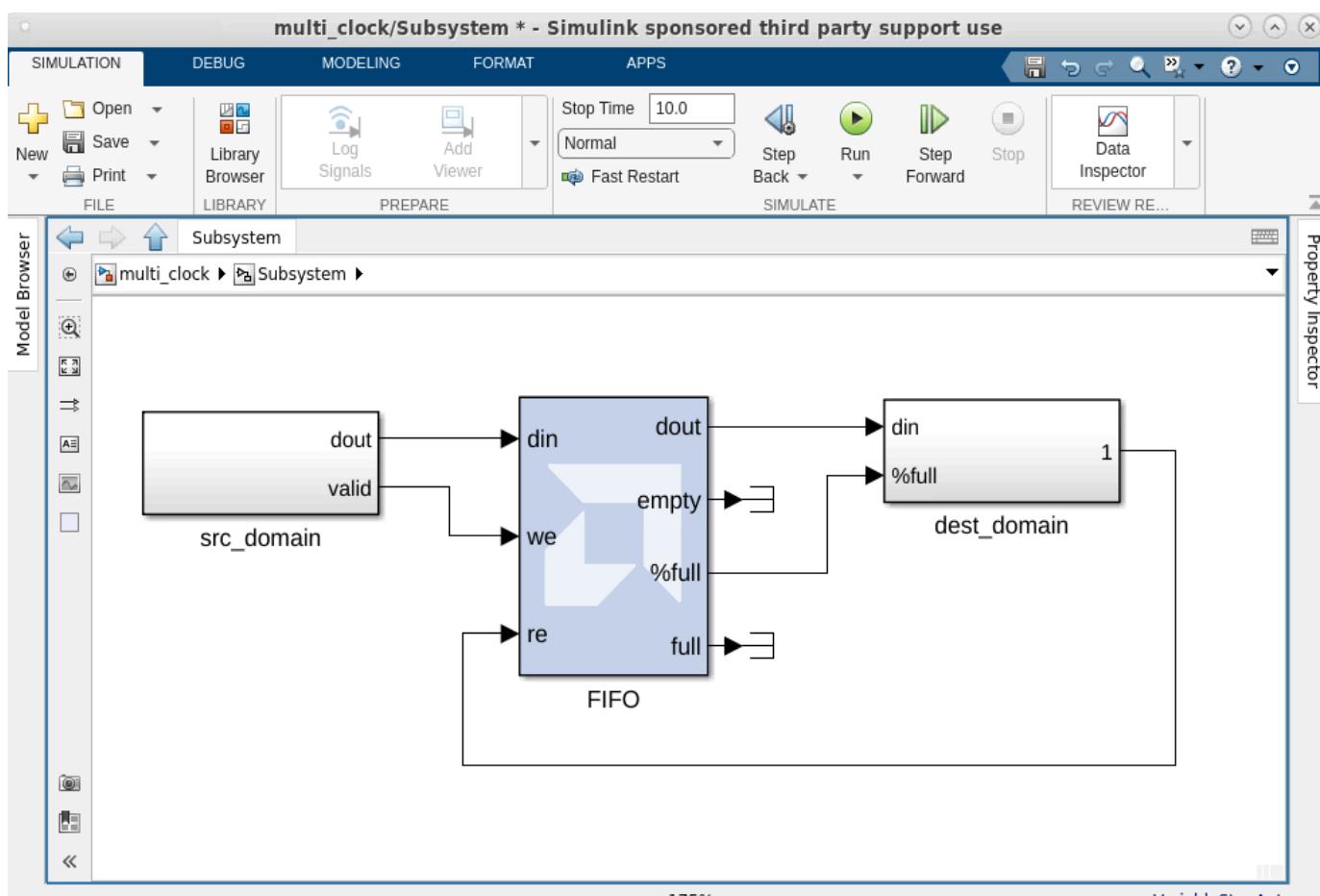
### HDL Blocks used to Create Asynchronous Clock Domains

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

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

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

Figure 53: Cross Domain FIFO Block



To complete the design, the FIFO block at the top level of the design is included to enable Code Generation.

## Configuring the Vitis Model Composer Hub Block

The Vitis Model Composer Hub block has to be configured to indicate that the Code Generation must proceed for a multiple clock design. This is indicated by turning on the Enable multiple clocks check box on the HDL Settings tab. This indicates to the Code Generation engine that the clock information for the Subsystems `src_domain` and `dest_domain` must be obtained from each clock's sub-tab. If this check box is not enabled, then the design will be treated as a single clock design.

## Clock Propagation Algorithm

For all HDL blocks in the `src_domain`, the clocking is governed by the Clock 1 tab in the Model Composer Hub block. Similarly for the `dest_domain` Subsystem, the clocking is governed by the Clock 2 tab in the Model Composer Hub block. For the FIFO block, the clocks are derived from its context in the design. Because the `we` and `din` ports are driven by signals emanating from the `src_domain` Subsystem, the `wr_clk` of the FIFO is tied to the `src_domain` clock. Because the `dout`, `full`, and `re` ports either drive or load signals from `dest_domain`, the `rd_clk` of the FIFO is tied to the `dest_domain` clock. Mixing and matching these signals across clock domains or using any other block (other than FIFO or Dual Port RAM) to cross clock domains will result in a DRC error.

## Debugging Clock Propagation

The Model Composer Hub block can be used to control the display of all HDL Block Icons using the Block Icon Display control in the HDL Analyze tab. From this tab, you can either select Normalized sample periods or Sample frequencies to help understand how clocks get propagated in the design. For multiple clock designs, the behavior of Normalized sample periods is that the smallest Simulink system period is used to normalize all the sample periods in the design.

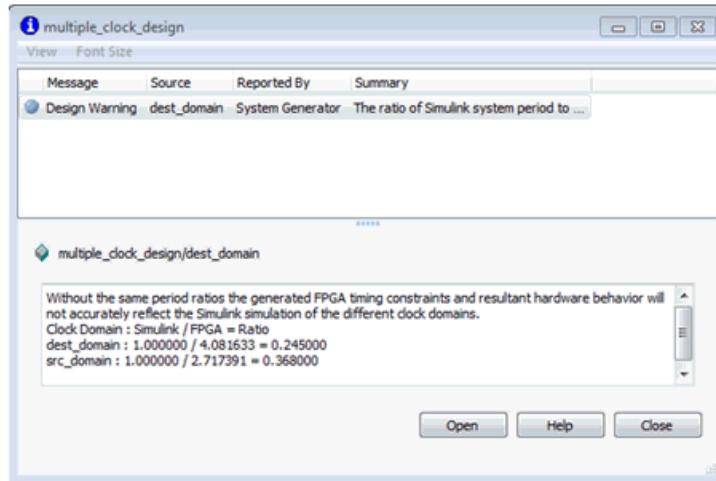
For Sample Frequencies, the port icon text display is the result of the following computation:

$$(1e6/\text{FPGA clock period}) * \text{Simulink system period}/\text{Port sample period}$$

where FPGA clock period is the FPGA clock period specified in ns in the domain's Clock Settings tab, and Simulink system period is the Simulink system period in seconds specified in the domain's HDL Clock Settings tab.

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

Figure 54: Warning

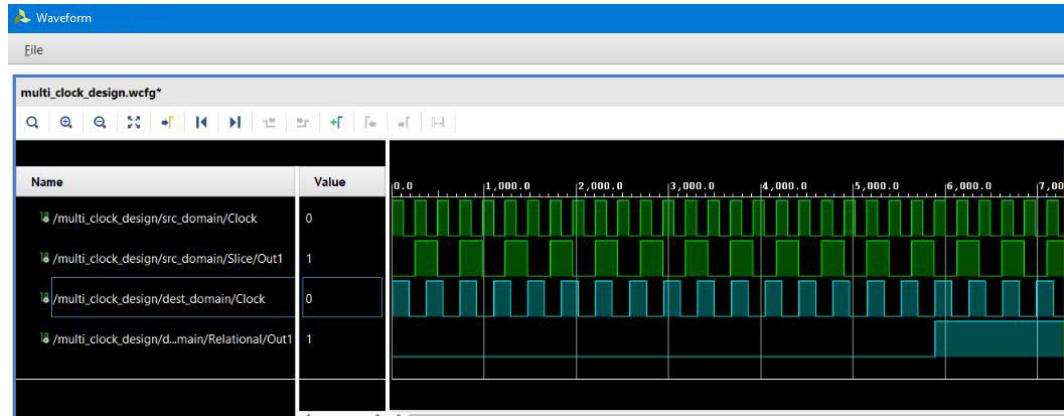


## Debugging Multiple Clock Domain Signals

In Vitis Model Composer you can use cross probing between the signal in the AMD Waveform Viewer and the Simulink diagram to aid the debugging process.

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

Figure 55: Waveform Viewer



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

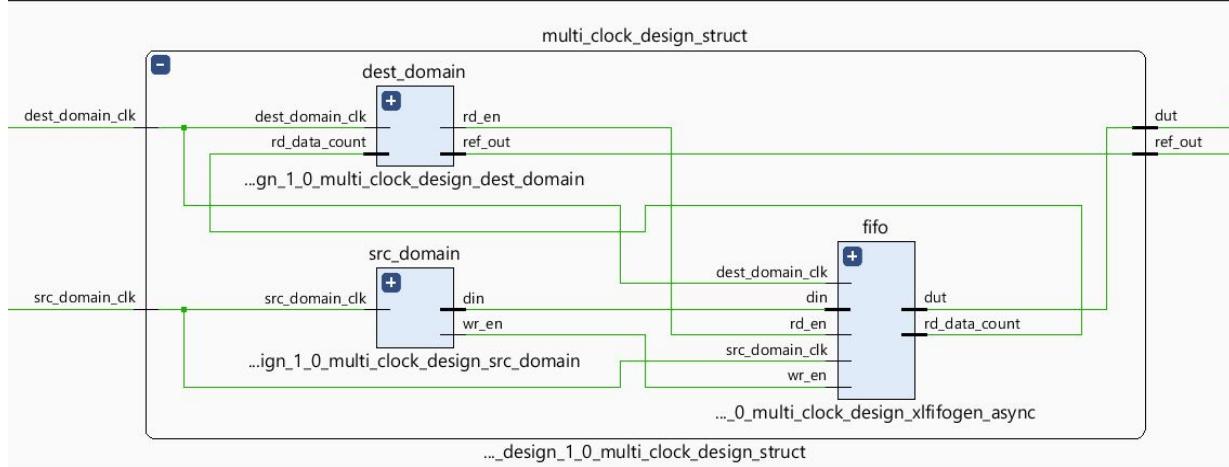
## Code Generation

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

- HDL Netlist
- IP catalog
- Synthesized Checkpoint

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

**Figure 56: Top-Level Hardware**



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

**Figure 57: Period Constraints**

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

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

## Known Issues

The following are some of the known issues:

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

## AXI Interface

AMBA AXI4 (Advanced eXtensible Interface 4) is the fourth generation of the AMBA interface defined and controlled by Arm®, and has been adopted by AMD as the next-generation interconnect for FPGA designs. AMD 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 third-party IP vendors because it is public, royalty-free, and an industry standard.

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

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

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

The purpose of this section is to provide an introduction to AMBA AXI4 and to draw attention to AMBA AXI4 details with respect to Model Composer. For more detailed information on the AMBA AXI4 specification, refer to the AMD AMBA-AXI4 documents found on the [AMBA AXI4 Interface Protocol](#) page on the AMD website.

### ***AXI4-Stream Support in Model Composer***

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

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

### **Naming Conventions**

AXI4-Stream signals are named in the following manner:

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

For example:

```
m_axis_tvalid
```

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

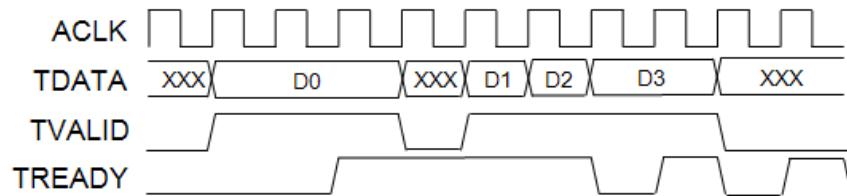
```
s_axis_control_tdata
```

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

### Notes on TREADY/TVALID Handshaking

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

Figure 58: TREADY/TVALID Handshake Timing Diagram



### Handshaking Key Points

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

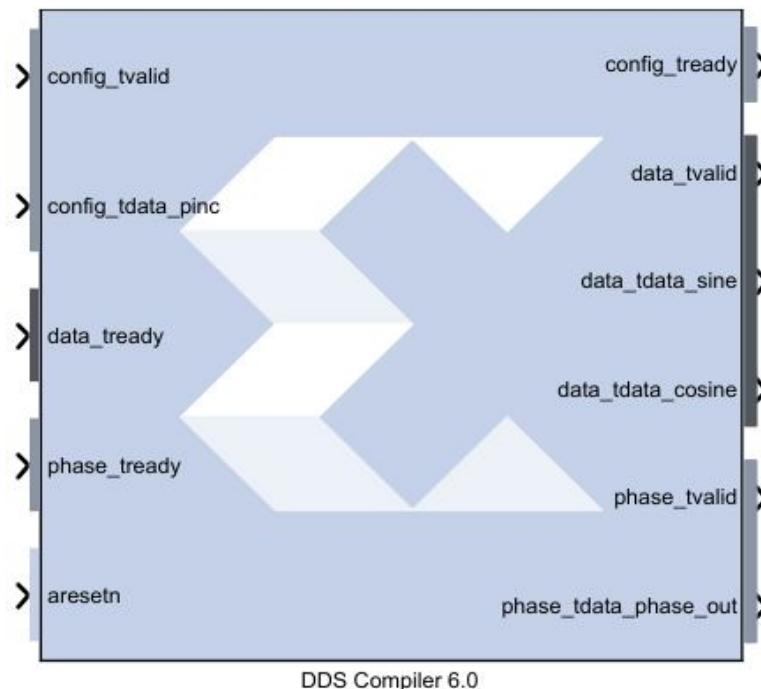
- There must be no combinatorial paths between input and output signals on both master and slave interfaces:
  - Applied to AXI4-Stream IP, this means that the TREADY slave output cannot be combinatorially generated from the TVALID slave input. A slave that can immediately accept data qualified by TVALID, 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.
  - 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 can limit interoperability with IP that generates TREADY. It is possible to connect an AXI4-Stream master with only forward flow control (TVALID only).

## ***AXI4-Stream Blocks in Model Composer***

HDL blocks that present an AXI4-Stream interface can be found in the Vitis Model Composer HDL Blockset Library entitled DSP/AXI4. Blocks in this library are drawn slightly differently from regular (non AXI4-Stream) blocks.

## Port Groupings

Figure 59: DDS Compiler 6.0



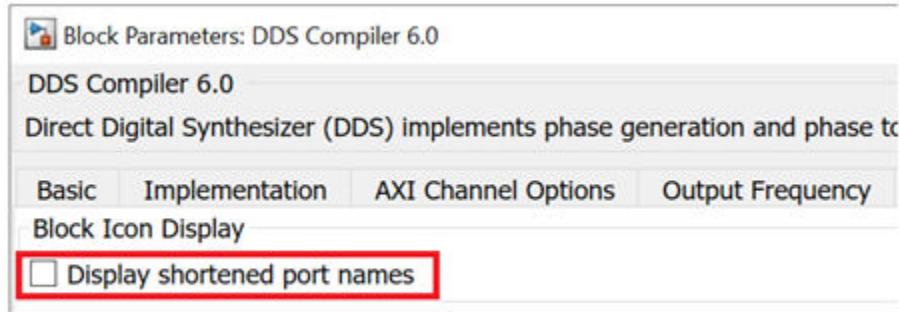
Blocks that offer AXI4-Stream interfaces have AXI4-Stream channels grouped together and color coded. For example, on the DDS compiler 6.0 block shown above, the input port `data_tready`, and the three output ports, `data_tvalid`, `data_tdata_sine`, `data_tdata_cosine` belong in the same AXI4-Stream channel.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

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

## Port Name Shortening

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

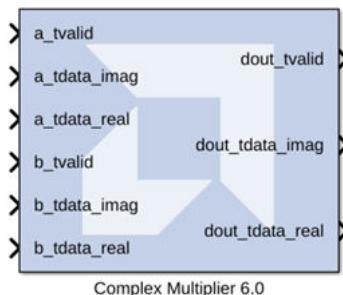
Figure 60: DDS Compiler 6.0



## Breaking Out Multi-Channel TDATA

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

Figure 61: Complex Multiplier 6.0



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

## AXI4-Lite Slave Interface Generation

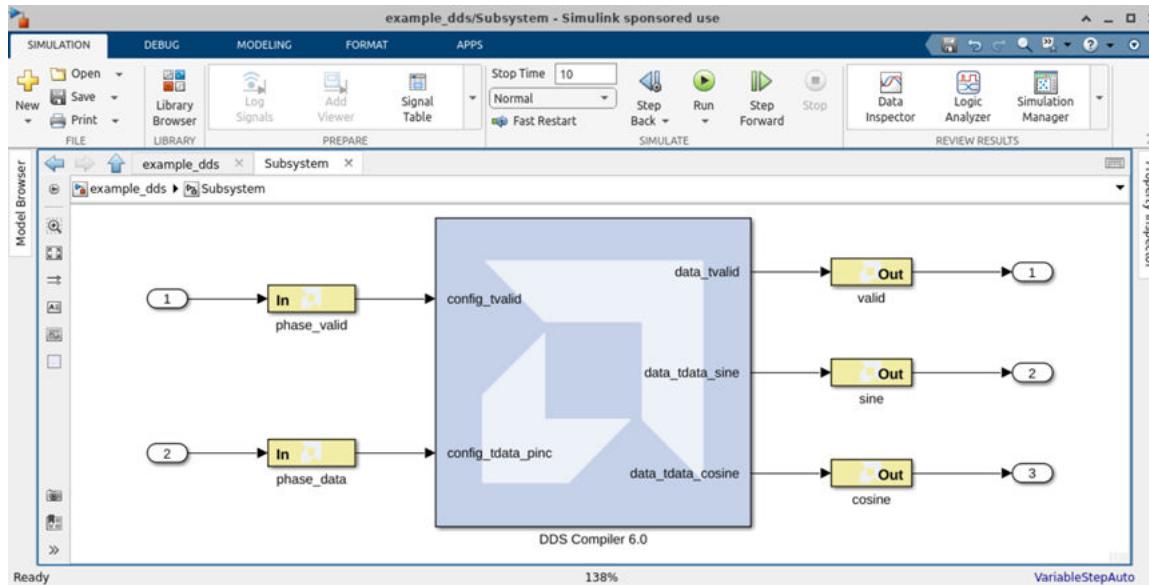
Design modules that are developed using Model Composer usually form a Subsystem of a larger DSP or Video system. These Model Composer modules are typically algorithmic and data path heavy modules that are best created in the visually-rich environment like MATLAB/Simulink. The larger system is typically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

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

## ***AXI4-Lite Interface Synthesis in Vitis Model Composer***

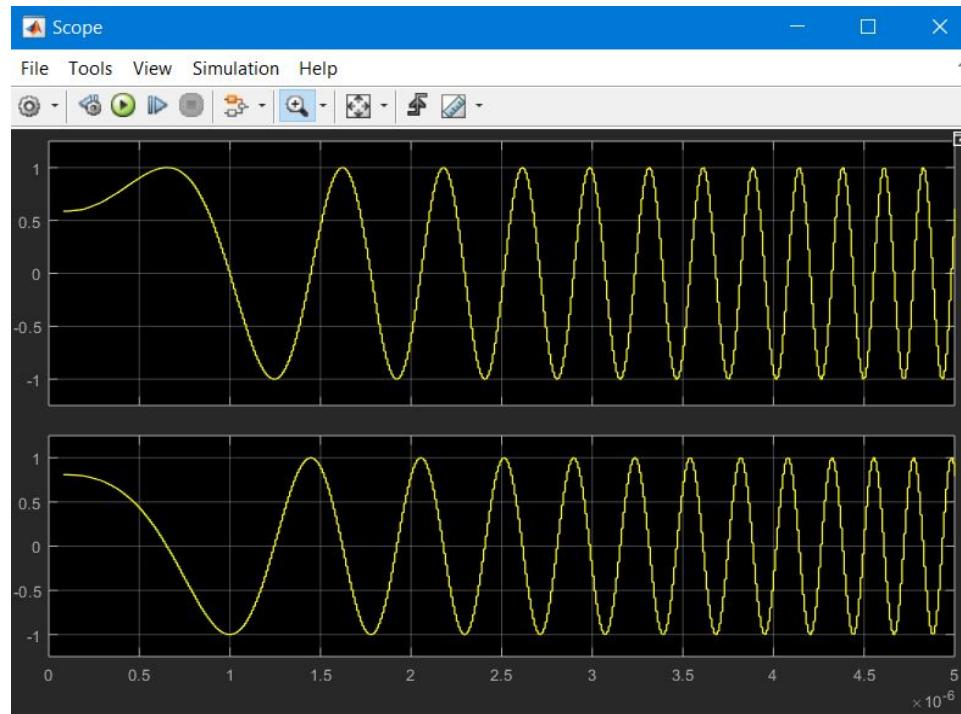
Design creation and verification is exactly the same as any other Vitis Model Composer design that does not include an AXI4-Lite interface. Consider the `example_dds` design shown below.

**Figure 62: Example DDS Design**



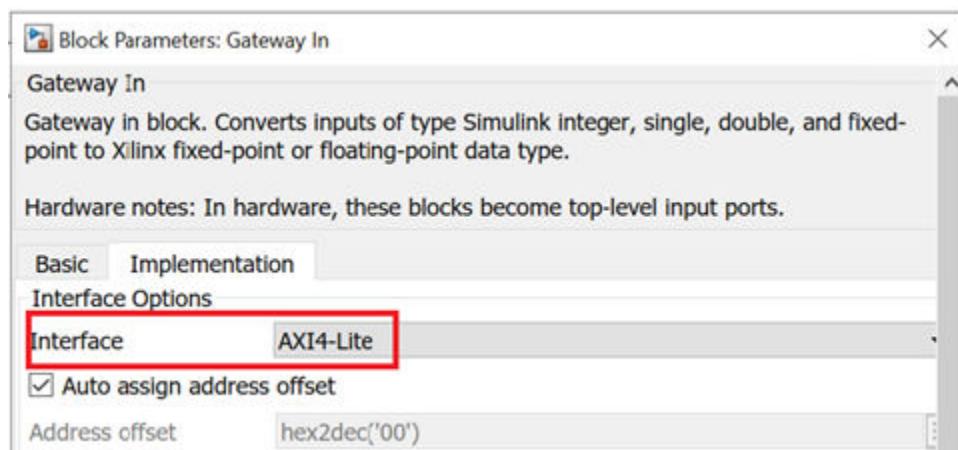
This design contains a DDS Compiler where the two input ports, `config_tvalid` and `config_tdata_pinc` are used to control the output frequency.

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

**Figure 63: Simulation Results**

## ***Configuring the Design for an AXI4-Lite Interface***

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

**Figure 64: Interface Options**

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

The following options are also of particular interest:

- **Auto assign address offset:** (Enabled) Each Gateway is associated with a register within the AXI4-Lite Interface and this control specifies that Automatic assignment of address offsets will take place in the design based on number of different Gateway Ins mapped to the AXI4-Lite interface. Addresses are byte aligned to a 32-bit data width.
- **Address offset:** (Disabled) This option is only enabled if Auto assign address offset is unchecked. It allows the user to manually override of Address Offset.
- **Interface Name:** Assigns a unique name to this interface. This name can be used to differentiate between multiple AXI4-Lite interfaces in the design.



---

**IMPORTANT!** The Interface Name must be composed of alphanumeric characters (lowercase alphabetic) or an underscore (\_) only, and must begin with a lowercase alphabetic character. 'axi4\_lite1' is acceptable, '1Ax4-Lite' is not.

---

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

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



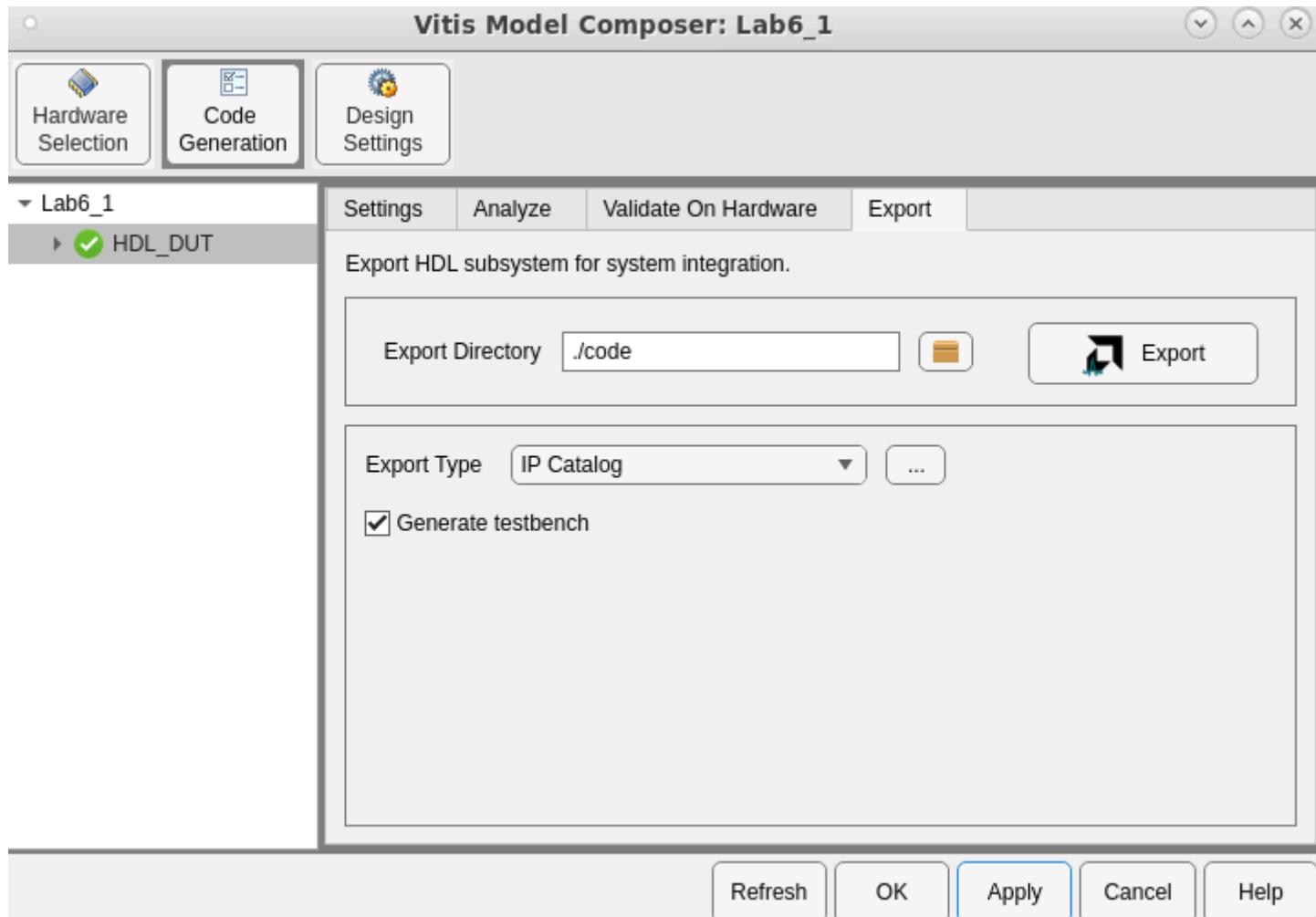
---

**TIP:** As an alternative to using individual Gateway In/Out blocks for each AXI signal, you can use the Gateway In/Out AXI4-Stream blocks in the HDL/Interfaces library.

---

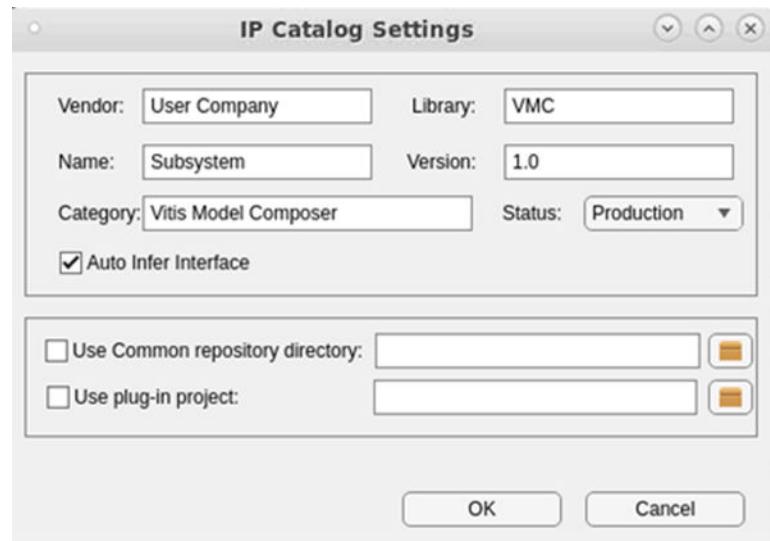
## Packaging the Design for Use in Vivado IP Integrator

When you complete the verification in Vitis Model Composer, you can package the design for use in IP integrator.

*Figure 65: Export IP*

The HDL subsystem must first be configured to use Export Type of IP Catalog. This compilation type will consolidate all hardware source created from Vitis Model Composer (RTL + IP + Constraints) into an IP. In addition, you can also use the button to the right of the Export Type drop-down menu to change the information that goes along with the IP. In this case, the default values shown below are used.

Figure 66: IP Catalog Settings



When you click the Export button in the Vitis Model Composer Hub block, the RTL code is generated and packaged along with constraints into an IP.

### Description of the Generated Results

Based on the Model Composer settings shown above, the following folders and files are created.

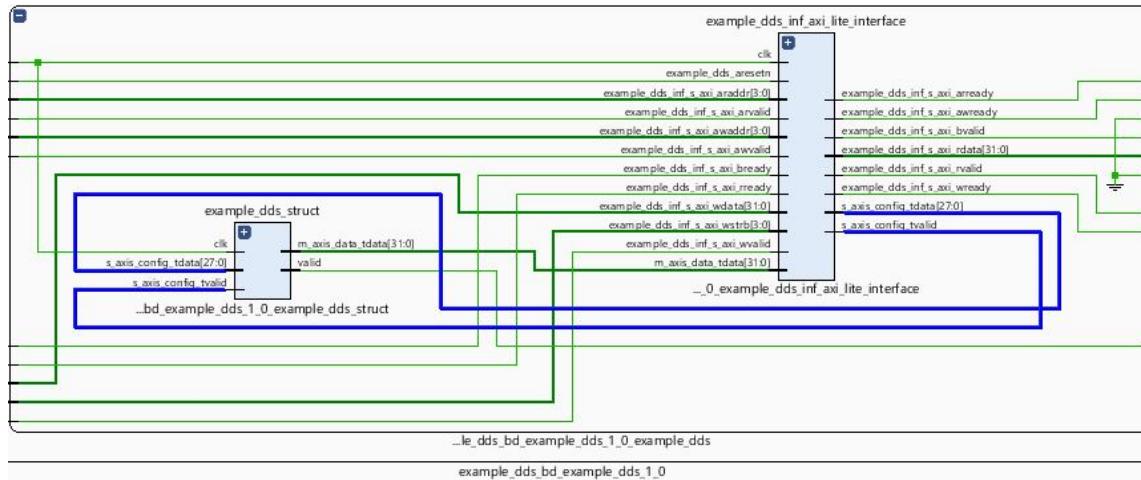
- <code directory>/ip/<Subsystem name>/src/ip: This directory contains all the IP-related hardware files, as well as the software drivers. It is this directory that you must add to the IP catalog.
- <code directory>/ip/<Subsystem name>/src/ip\_catalog: This directory contains an example Vivado IDE project called <Subsystem name>.xpr.

### Mapping to AXI4-Lite Interfaces

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

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

Figure 67: Single AXI4-Lite Interface



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

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

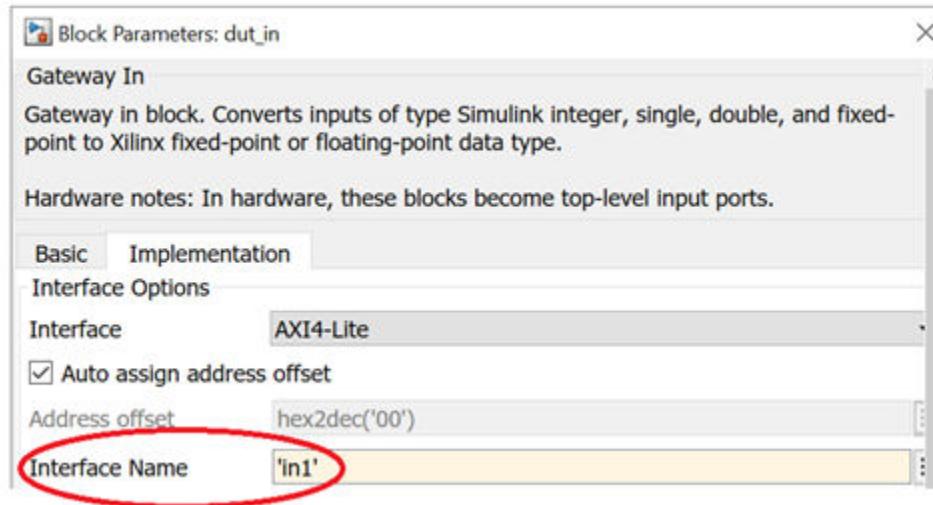
## Managing Multiple AXI4-Lite Interfaces

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

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

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

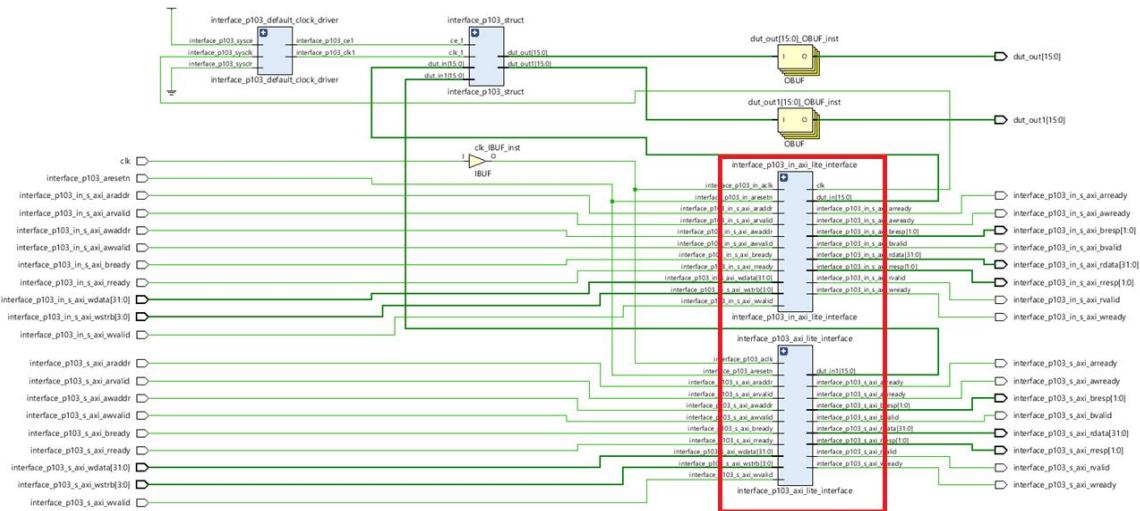
Figure 68: Interface Name



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

If you specify the HDL Netlist compilation type in the Vitis Model Composer Hub block, and then elaborate the design in Vivado, two AXI4-Lite Decoders will be created, as shown in the red rectangle in the following figure.

Figure 69: AXI4-Lite Decoders

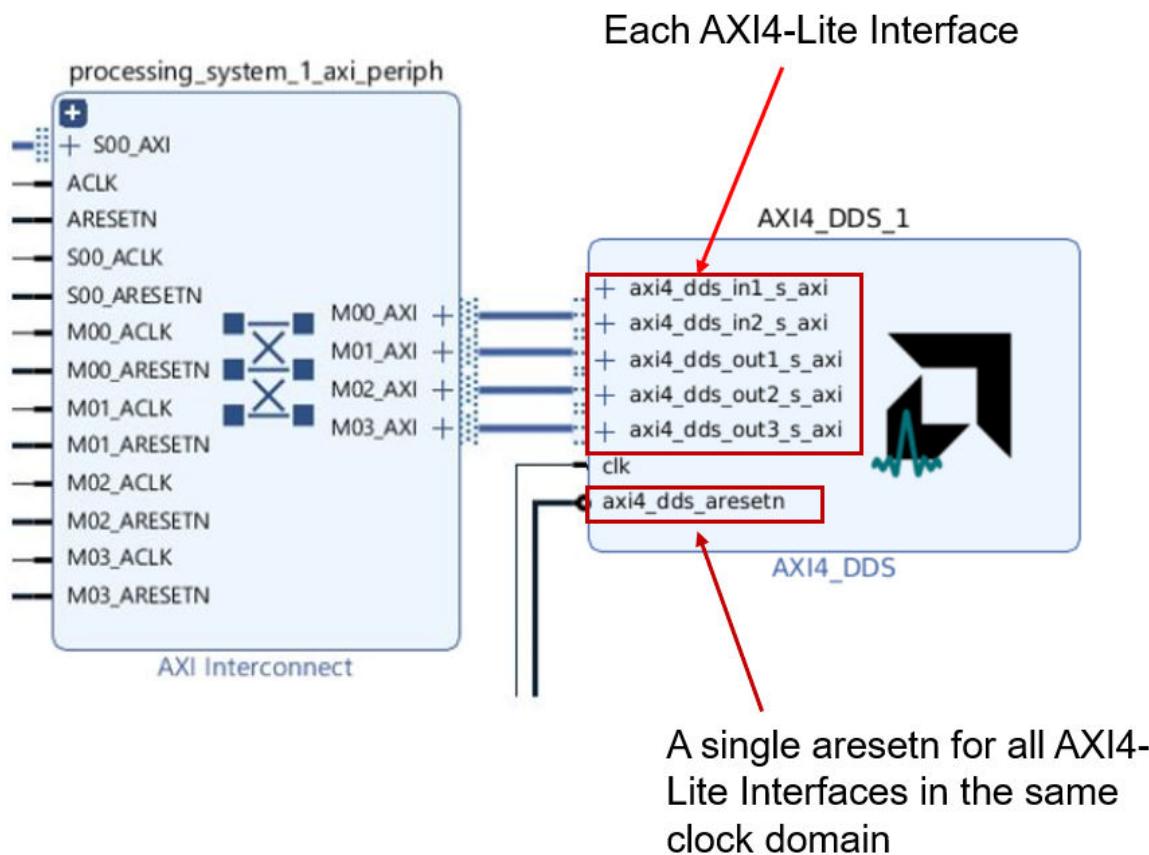


If you specify the IP Catalog compilation type in the Vitis Model Composer Hub block, the flow will also generate an example BD with multiple AXI4-Lite interfaces and an `aresetn` signal.

The naming convention for an interface is:

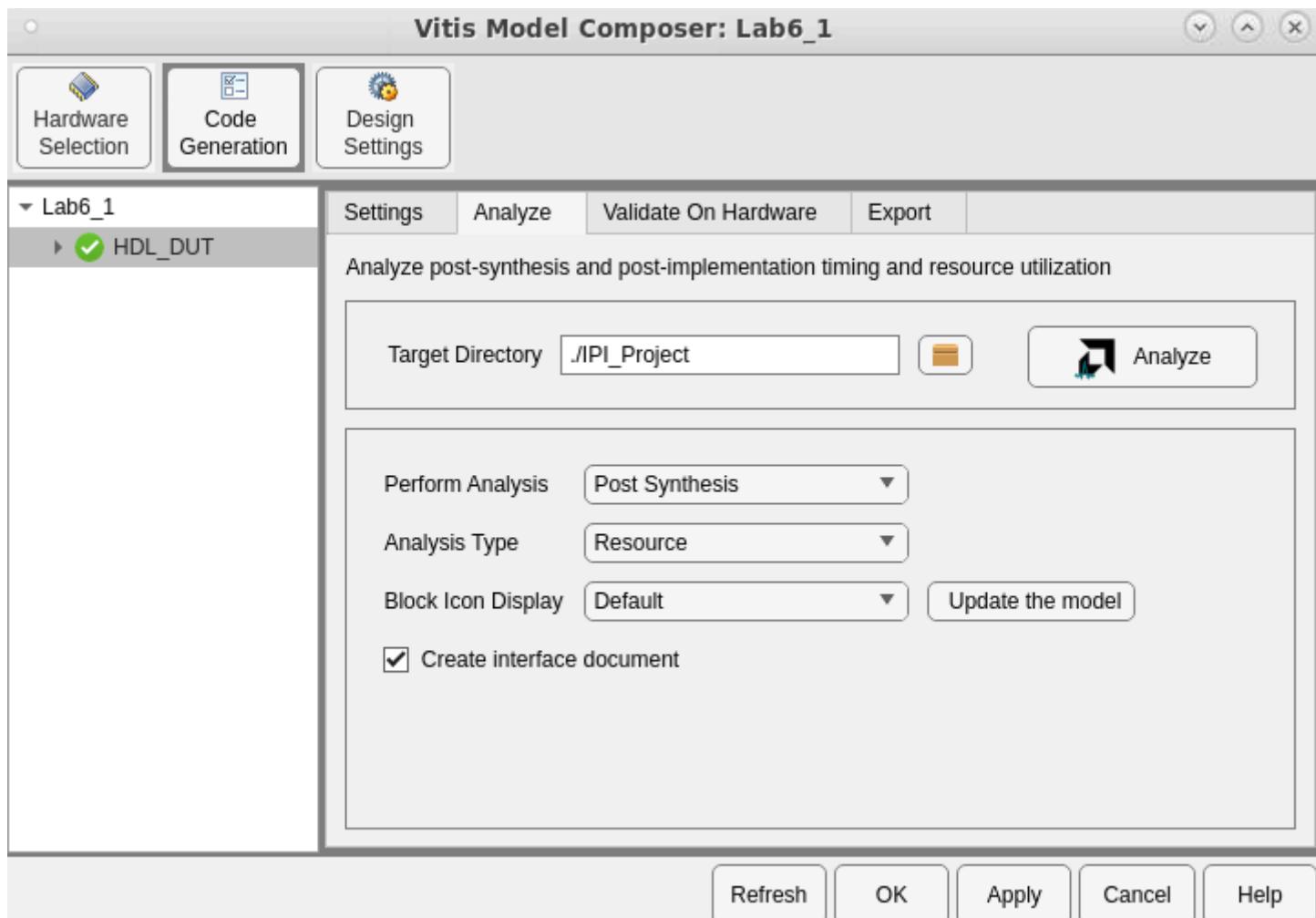
```
<clock domain name/design name>_<interface name>_s_axi
```

Figure 70: Example BD



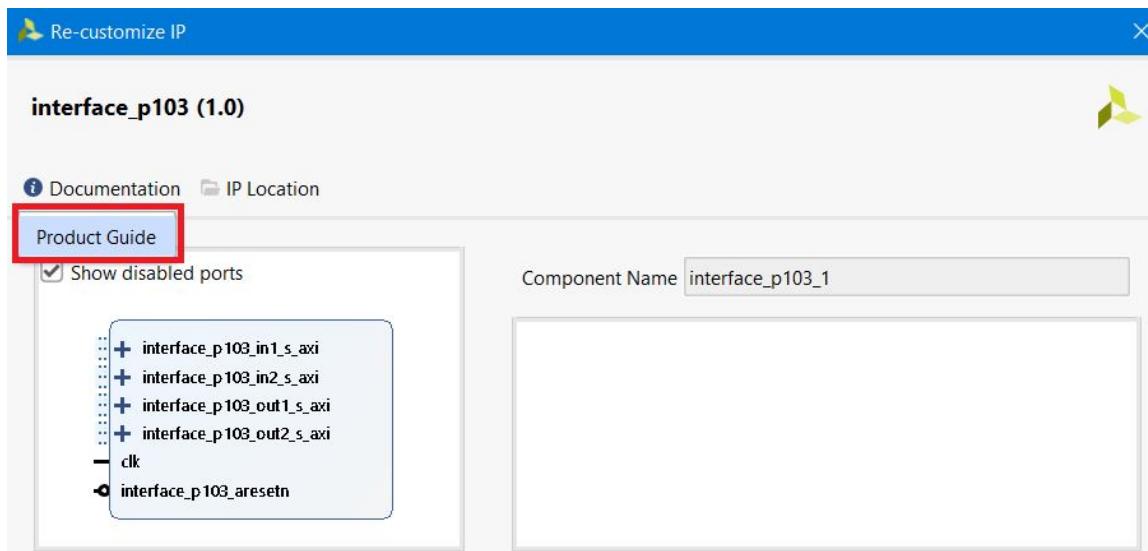
To generate a document describing the IP, select the **Create interface document** option on the Analyze tab of the Vitis Model Composer Hub block before you perform the compilation.

Figure 71: Create Interface Document



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

Figure 72: Accessing Documents



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

**Figure 73: Sample Document****02-Feb-2019****interface\_p103****Port Interface**

This section documents the port interface of interface\_p103. All the *Gateway In* and *Gateway Out* blocks in a System Generator design are translated to top-level input and output ports. *System Generator Type* refers to the type of signals emanating from *Gateway Ins* and driving *Gateway Outs*. *Type* refers to one of the following -

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

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

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

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

**Figure 74: Memory Map****Memory Map**

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

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

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

## ***Address Generation***

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

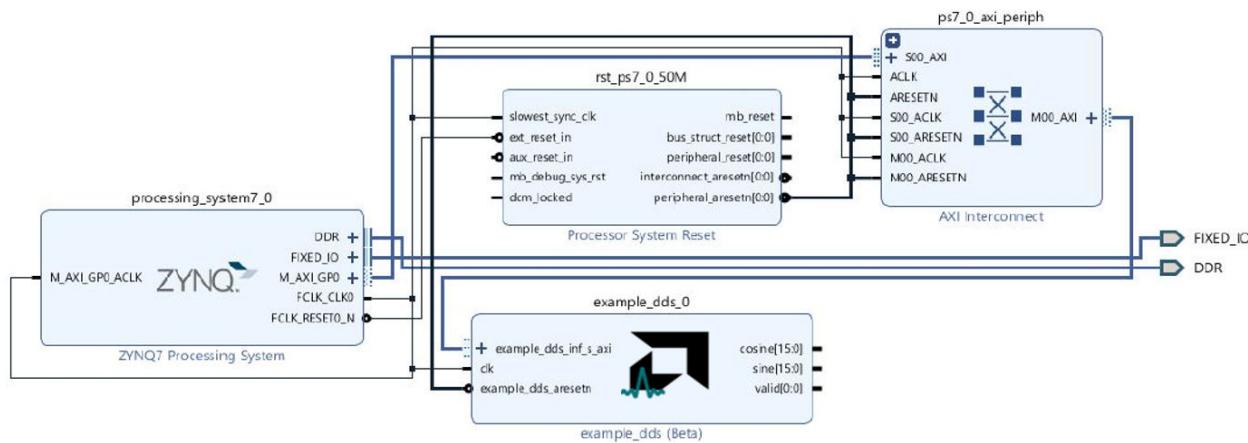
1. Each AXI4-Lite gateway is associated with a unique address offset that is aligned with a 32-bit word boundary (that is, will be a multiple of 4).
2. Addressing begins at zero.
3. Addressing is incrementally assigned in the lexicographical order of the gateways. In the event two gateways have the same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

## ***Features of the Vivado IDE Example Project***

The AMD Vivado™ IDE example project (<code directory>/ip\_catalog/example\_dds.xpr) is created to help you jump start your usage of the IP created from Vitis Model Composer.

1. The IP generated from Model Composer is already added to the IP catalog associated with the project and available for the RTL flow as well as the IP integrator-based flow.
2. The design includes an RTL instantiation of IP called example\_dds\_0 underneath example\_dds\_stub that indicates how to instance such an IP in RTL flow.
3. The design includes a test bench called example\_dds\_tb that also instances the same IP in RTL flow.
4. The design includes an example IP integrator diagram with an AMD Zynq™ 7000 subsystem as the part selected. In this example it is an AMD Zynq™ 7000 SoC part. For all other parts, a MicroBlaze-based subsystem is created.

Figure 75: IP Integrator Diagram



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



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

## Software Drivers

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

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

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

/*
 * Read from <Gateway In id> of <design name>. Assignments are LSB-justified.
 *
 * @param InstancePtr is the phase_valid instance to operate on.
 */
```

```

/*
 * @return u32
 *
 * @note      Phase Valid Port That Must Be Asserted.
 *
 */
u32 <Gateway In id>_read(example_dds *InstancePtr);

```

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

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

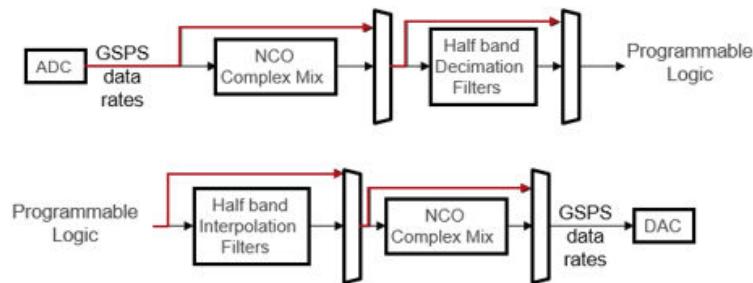
### ***Known Issue in AXI4-Lite Interface Generation***

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

## **Using Super Sample Rate (SSR) Blocks in Model Composer**

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

*Figure 76: RFSoC Device*



Depending on what is needed, RFSoCs can be used in two ways:

- Use the available hardened NCO & Complex Mix and Half Band Decimation/interpolation filters.

- If the sequence of the hardened blocks does not meet the design requirement, you can bypass them as shown in the preceding figure.

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

### What is SSR?

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

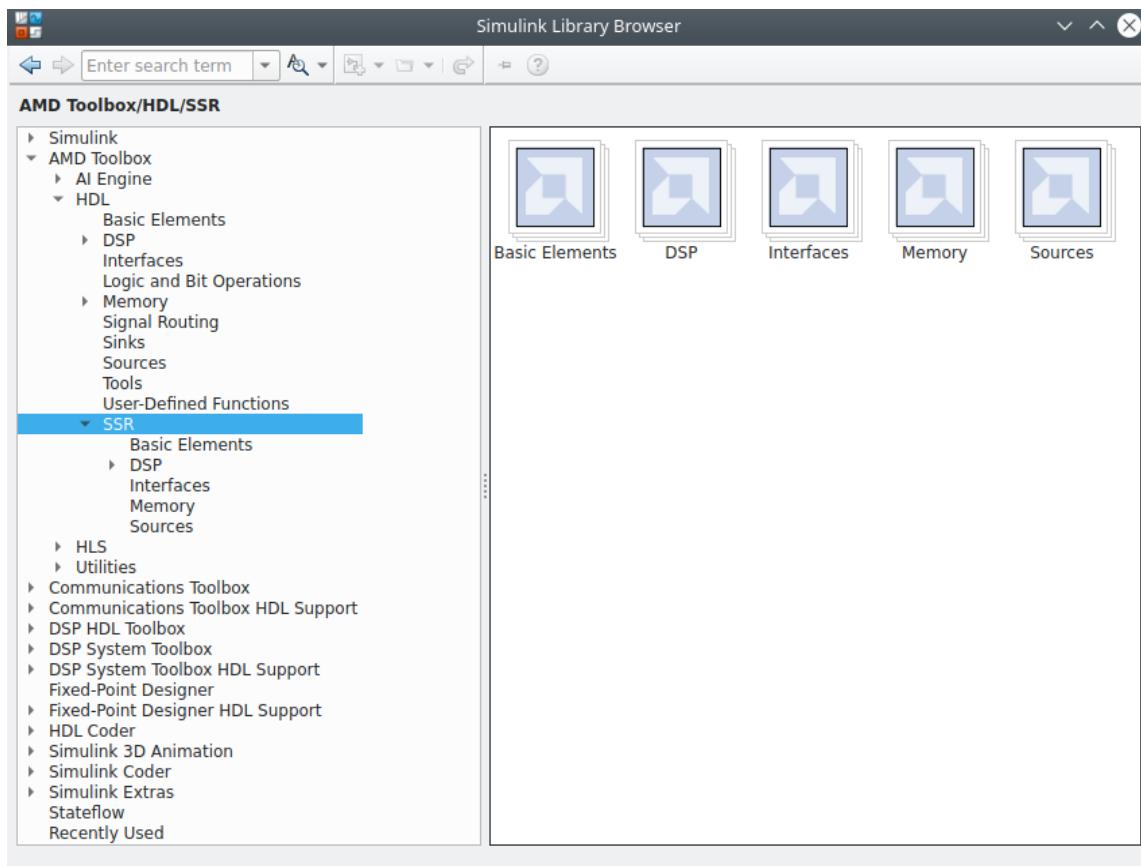
### How SSR helps users?

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

### SSR Library

Model Composer provides a separate set of library blocks for handling SSR. Currently, Model Composer supports 25 vector blocks, which can be accessed from the Simulink Library Browser.

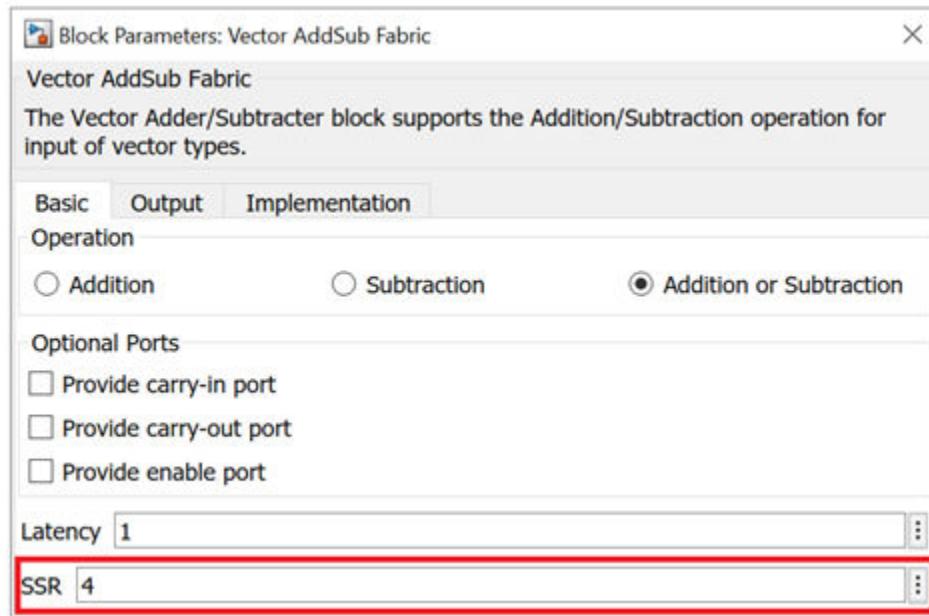
Figure 77: SSR Block set in HDL Library



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

The SSR block set is defined in the AMD SSR Blockset.

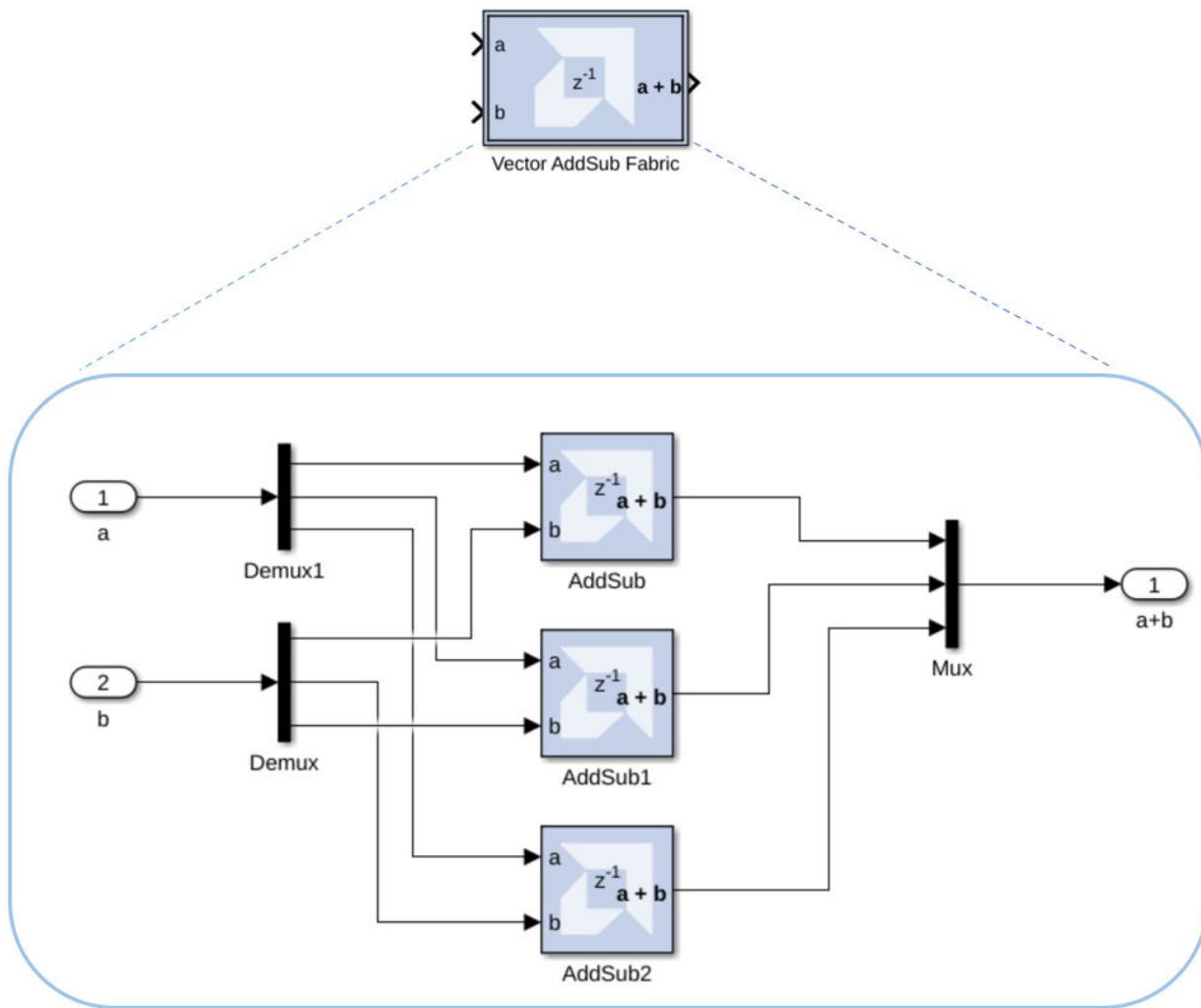
Figure 78: Default SSR Value



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

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

Figure 79: Vector AddSub Fabric Example



## Performing Analysis in Vitis Model Composer

Vitis Model Composer is a bit- and cycle-accurate modeling tool. You can verify the functionality of your designs by simulating in Simulink®. However, to ensure that your Model Composer design works correctly when it is implemented in your target AMD device, these analysis tools are integrated into Model Composer:

- **Timing Analysis:** To ensure that the HDL files generated by Model Composer operate correctly in hardware, you must close timing. To help accelerate this process, timing analysis has been integrated into Model Composer.

- Resource Analysis:** To ensure that the HDL files generated by Model Composer will fit into your target device, you might need to analyze the resources being used. To help accelerate this process, resource analysis has been integrated into Model Composer.

**Table 7: Performing Analysis in Vitis Model Composer**

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

## Timing Analysis in Vitis Model Composer

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

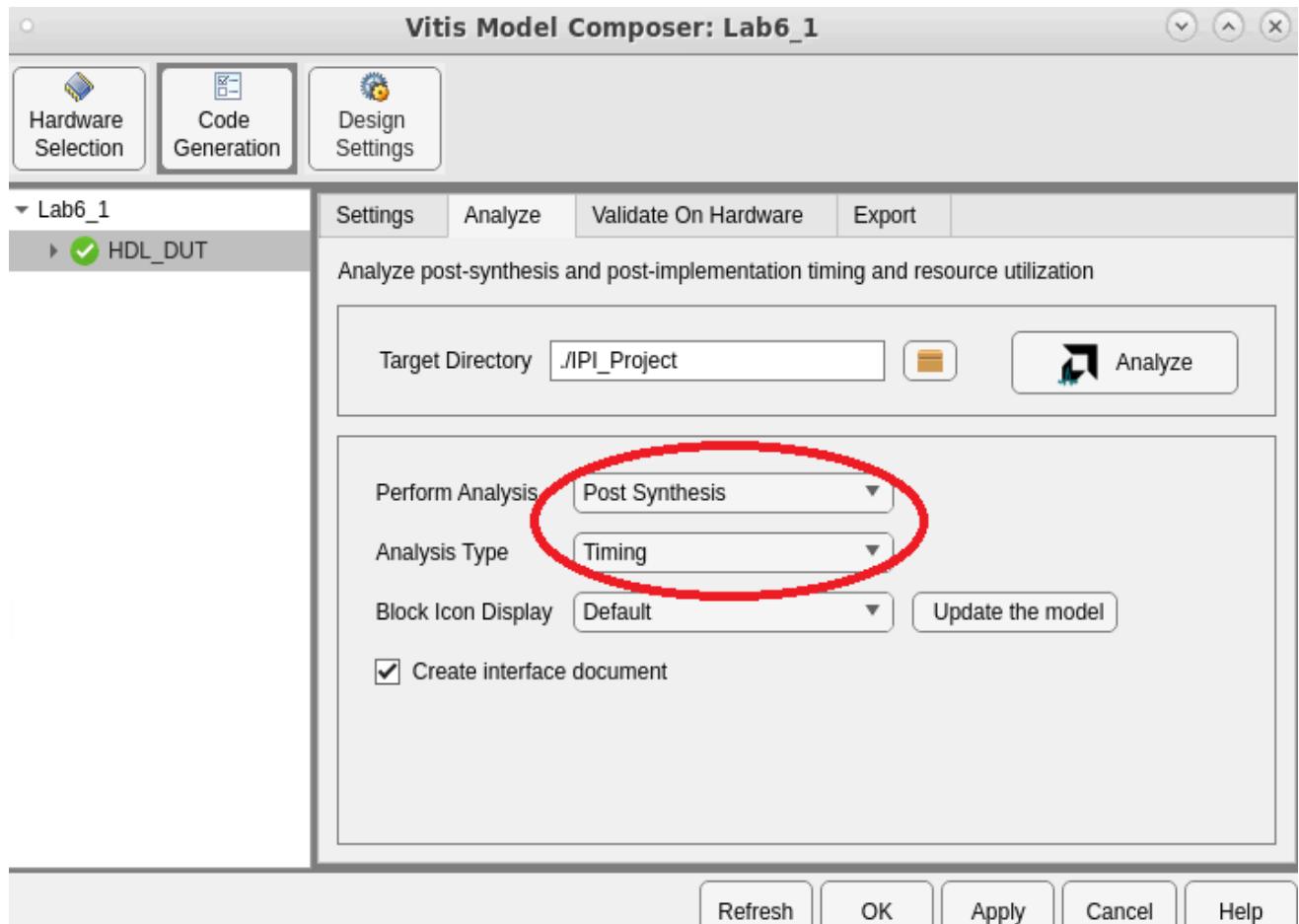
Timing analysis allows you to perform static timing analysis on the HDL files generated from Model Composer, either Post-Synthesis or Post-Implementation. It also provides a mechanism to correlate the results of running the AMD Vivado™ Timing Engine on either the Post-Synthesized netlist or the Post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

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

## ***Performing Timing Analysis***

To perform timing analysis in Vitis Model Composer:

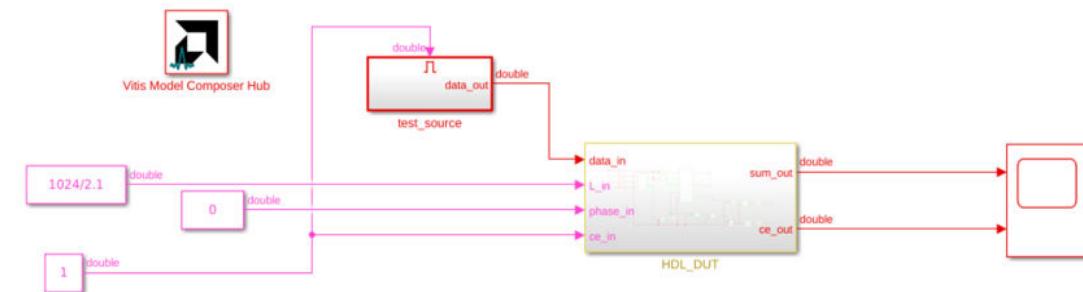
1. Double-click the **Vitis Model Composer Hub** block in the Simulink model.
2. Enter the following in the Vitis Model Composer Hub dialog box:
  - a. In the Analyze tab, set the Perform Analysis field to Post Synthesis or Post Implementation based on the runtime versus accuracy tradeoff. Post Synthesis timing will be less accurate but run faster, while Post Implementation timing will be more accurate but run slower.
  - b. Set the Analysis Type field to Timing.



3. In the Model Composer Hub dialog box, click **Analyze**.

When you generate, the following occurs:

- Model Composer generates the required files for the selected compilation target. For timing analysis Model Composer invokes Vivado in the background for the design project, and passes design timing constraints to Vivado.
- Depending on your selection for Perform Analysis (Post Synthesis or Post Implementation), the design runs in Vivado through synthesis or through implementation.
- After the Vivado tools run is completed, timing paths information is collected and saved in a specific file format from the Vivado timing database. At the end of the timing paths data collection the Vivado project is closed and control is passed to the MATLAB®/Model Composer process.
- Model Composer processes the timing information and displays a Timing Analyzer table with timing paths information (see below).



Timing Analyzer: Lab3

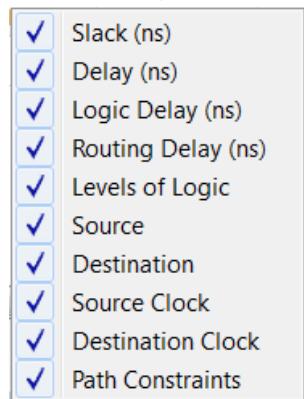
Post Synthesis Timing Paths: Clicking on an instance name highlights corresponding block/subsystem in the model

Violation type : setup Select Columns Status : FAILED

	Slack (ns)	Delay (ns)	Logic Delay (ns)	Routing Delay (ns)	Levels of Logic	Source	Destination	Source Clock	Destination Clock
1	-0.2670	1.9980	1.5640	0.4340	0	Lab3/HDL_DUT/subsystem1/coef	Lab3/HDL_DUT/subsystem1/Mult	clk	clk
2	2.2000	1.7870	1.2860	0.5010	14	Lab3/HDL_DUT/subsystem1/AddSub2	Lab3/HDL_DUT/subsystem1/AddSub2	clk	clk
3	2.6960	1.2910	1.0450	0.2460	9	Lab3/HDL_DUT/addr_gen/Register4	Lab3/HDL_DUT/addr_gen/Register	clk	clk
4	2.8120	1.1750	0.6840	0.4910	3	Lab3/HDL_DUT/addr_gen/Register	Lab3/HDL_DUT/addr_gen/AddSub1	clk	clk

In the timing analyzer table:

- Only unique paths from the Simulink model are reported.
- The 50 paths with the lowest Slack values are displayed with the worst Slack at the top, and increasing Slack below.
- Paths with timing violations have a negative Slack and display in red.
- The display order can be sorted for any column's values by clicking the column head.
- To show/hide columns, click the **Select Columns** button and select/deselect the column name as required.



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

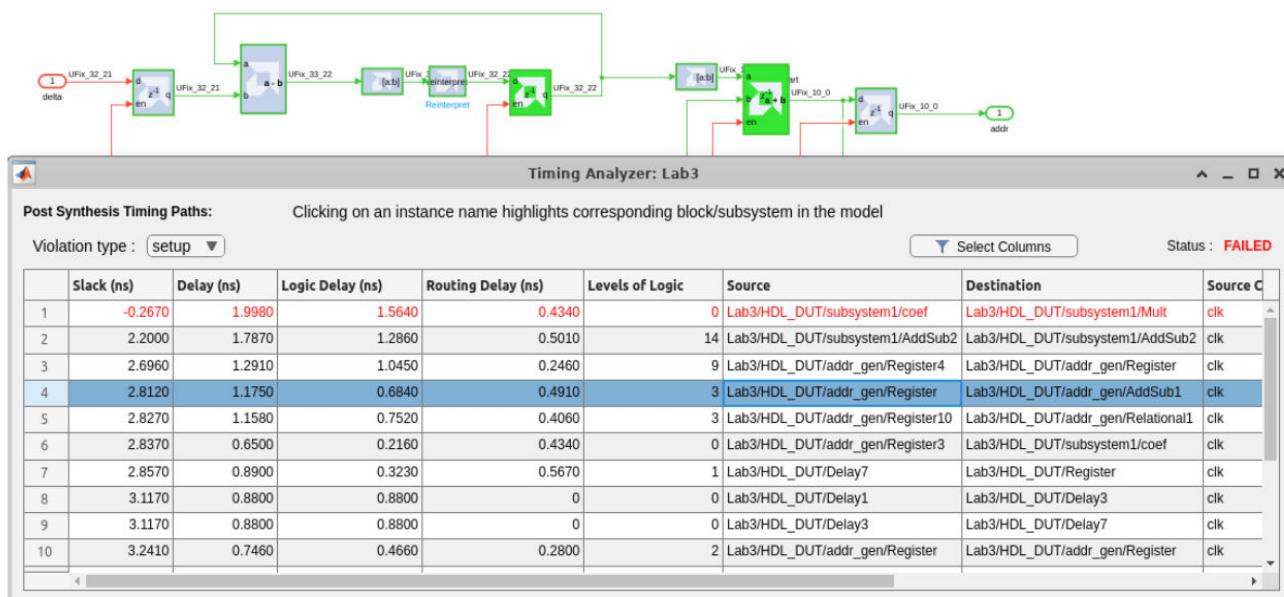
Source Clock	Destination Clock	Path Constraints
clk, ce_3	clk, ce_3	set_multicycle_path -setup 3 -hold 2
clk, ce_4	clk, ce_4	set_multicycle_path -setup 4 -hold 3
clk, ce_6	clk, ce_6	set_multicycle_path -setup 6 -hold 5
clk, ce_12	clk, ce_12	set_multicycle_path -setup 12 -hold 11
clk, ce_12	clk, ce_12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

- You can cross probe from the table to the Simulink model by selecting a path in the table, which will highlight the corresponding HDL blocks in the Simulink model. See [Cross Probing from the Timing Analysis Results to the Model](#).

## Cross Probing from the Timing Analysis Results to the Model

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

Figure 80: Timing Analyzer Table

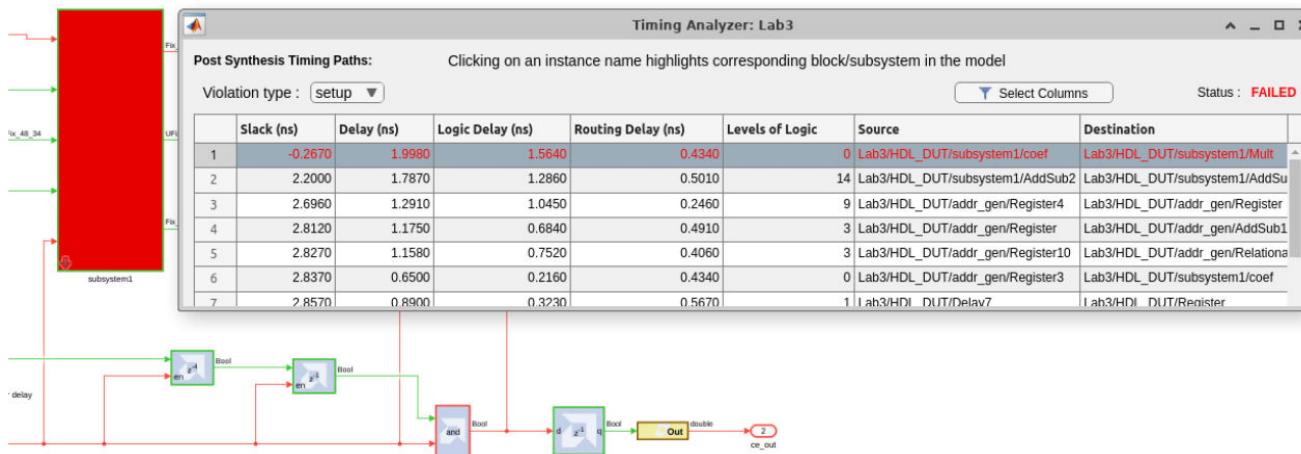


When you cross probe, the following displays in the model:

- Blocks in a path with a timing violation are highlighted in red in the model, whereas blocks that belong to a path with no timing violation (that is, a path with a positive Slack value) are highlighted in green in the model.

- If blocks in a highlighted path are inside a subsystem, then the subsystem is highlighted in red, so you can expand the subsystem to inspect the blocks underneath.

**Figure 81: Cross Probing**



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

## Accessing Existing Timing Analysis Results

To access existing timing analysis results, select **Timing** as the Analysis Type and click the **Analyze** button again. The Timing Analyzer table displays the timing analysis results stored in the specified Code Directory, regardless of the option selected for Perform Analysis (**Post Synthesis** or **Post Implementation**).

## Recommendations For Troubleshooting Timing Violations

The following are recommended for troubleshooting timing violations:

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

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

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

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

## Resource Analysis in Vitis Model Composer

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

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

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

### Performing Resource Analysis

To perform resource analysis in Vitis Model Composer:

1. Double-click the Model Composer Hub block in the Simulink model.
2. Select the following in the Model Composer Hub dialog box dialog box:
  - a. In the Hardware Selection tab:
    - i. Click the button next to the Select Hardware box to open the Device Chooser.

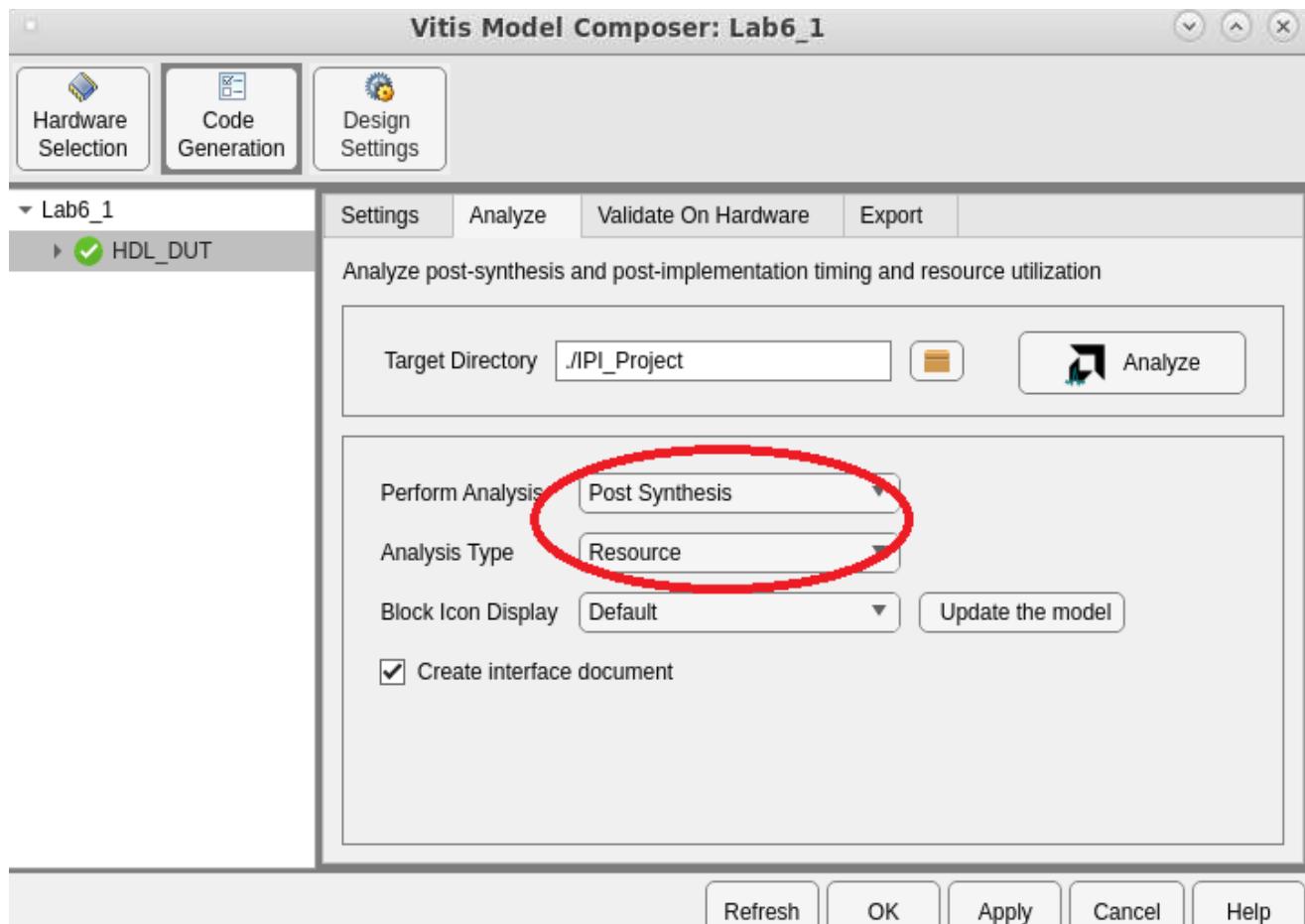
- ii. Specify the Part in which your design will be implemented.

**Note:** If you select a Board or Platform instead of a Part, the Part field will be populated with the name of the part on the selected Board or Platform.

- b. In the Analyze tab:

- i. Set the Perform Analysis field to Post Synthesis or Post Implementation based on the runtime versus accuracy tradeoff.
- ii. Set the Analysis type field to Resource.

Figure 82: Resource Analyzer



3. In the Model Composer Hub dialog box, click **Analyze**.

When you generate, the following occurs:

- a. Model Composer generates the required files for the selected compilation target. For resource analysis Model Composer invokes Vivado in the background for the design project.

- b. Depending on your selection for Perform analysis (Post Synthesis or Post Implementation), the design runs in Vivado through synthesis or through implementation.
- c. After the Vivado tools run is completed, resource utilization data is collected from the Vivado resource utilization database and saved in a specific file format under the target directory. At the end of the resource utilization data collection the Vivado project is closed and control is passed to the MATLAB/Vitis Model Composer process.
- d. Model Composer processes the resource utilization data and displays a Resource Analyzer table with resource utilization information (see below).

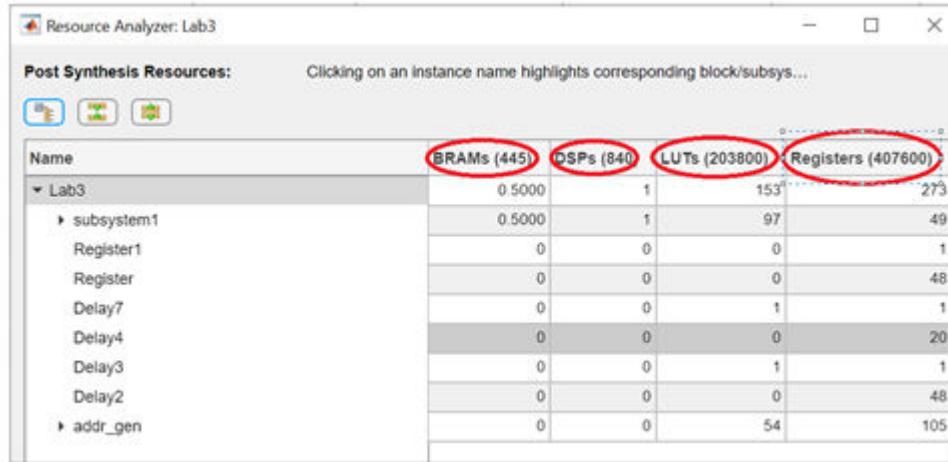
Figure 83: Resource Analyzer

Name	BRAMs (140)	DSPs (220)	LUTs (53200)	Registers (106400)
Lab6_1	2	0	569	641
DEScore	0	0	382	208
Start	0	0	0	1
round_datapath	0	0	176	80
Key Scheduler	0	0	168	58
DES_fsm	0	0	6	5
Data	0	0	32	64
DES transmit interface block	0	0	67	197
DES receive interface block	0	0	7	137
AXI FIFO	2	0	113	99

In the resource analyzer table:

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

Figure 84: Resource Analysis Report for Kintex 7



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

Table 8: Number of BRAMs

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

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

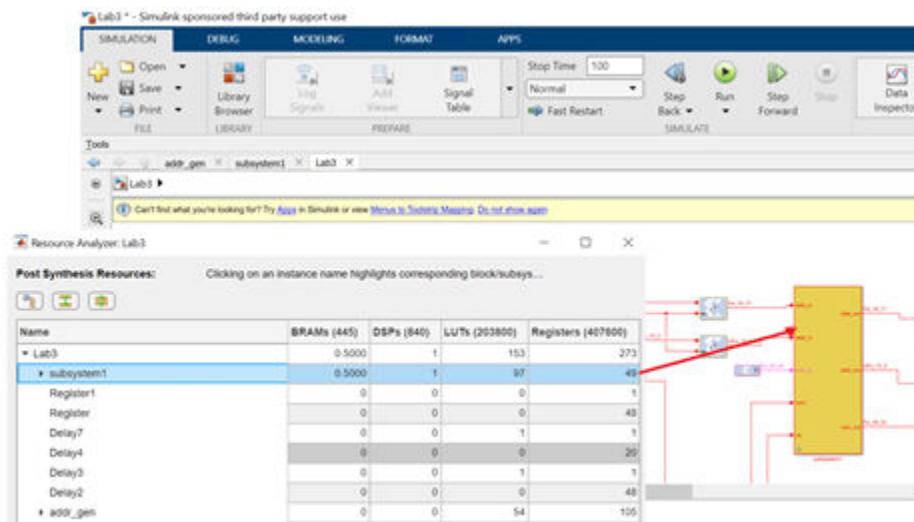
$$\text{Total BRAMs} = (\text{Number of RAMB36E}) + (\text{Number of FIFO36E}) + 0.5 \\ (\text{Number of RAMB18E} + \text{Number of FIFO18E})$$

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

## Cross Probing from the Resource Analysis Results to the Model

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

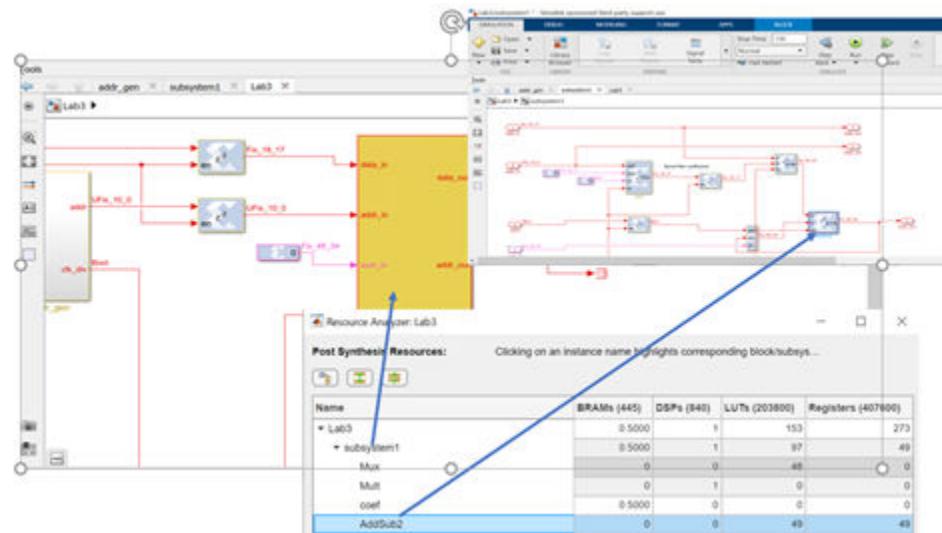
Figure 85: Resource Analyzer



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

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

Figure 86: Selected Subsystem



## Accessing Existing Resource Analysis Results

To access existing resource analysis results, select **Resource** as the Analysis Type and click the **Analyze** button again. The Resource Analyzer table displays the resource analysis results stored in the specified Code Directory, regardless of the option selected for Perform Analysis (Post Synthesis or Post Implementation).

## Recommendations for Optimizing Resource Analysis

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

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

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

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

1. In Vivado IDE:

- Select the Synthesis option `-keep_equivalent_registers`.

- . Set the Synthesis option `-resource_sharing` to the value `off`.
  - 2. Save the new Synthesis strategy and exit Vivado IDE.
  - 3. In Vitis Model Composer, select the new custom **Synthesis Strategy** in the Model Composer Hub dialog box (HDL Settings tab) before generating the design.
- 

## Using Hardware Co-Simulation

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

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

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

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

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

- If a board is supported for JTAG hardware co-simulation, the Hardware Co-Simulation option for Compilation is enabled in the Vitis Model Composer Hub block dialog box when you perform the procedure described in [Compiling a Model for Hardware Co-Simulation](#). If the Hardware Co-Simulation option is grayed out and disabled, you cannot perform JTAG hardware co-simulation on the board.

This support applies to the following types of boards:

**Table 9: Board Support**

Board Name	Display Name
zed	ZedBoard Zynq Evaluation and Development Kit
ac701	Artix 7 AC701 Evaluation Platform 1.0/1.1
kc705	Kintex 7 KC705 Evaluation Platform 1.0/1.1
kcu105	Kintex UltraScale KCU105 Evaluation Platform
vc707	Virtex 7 VC707 Evaluation Platform
vc709	Virtex 7 VC709 Evaluation Platform
vcu108	Virtex UltraScale VCU108 Evaluation Platform
zc702	ZYNQ-7 ZC702 Evaluation Board
zc706	ZYNQ-7 ZC706 Evaluation Board

- AMD boards installed as part of your Vivado installation.
- Partner boards, which are available and can be downloaded from the partner websites and installed as part of the Vivado Design Suite.
- Custom boards, which can be created in the Vivado Design Suite as detailed in this [link](#), in the *Vivado Design Suite User Guide: System-Level Design Entry (UG895)*.

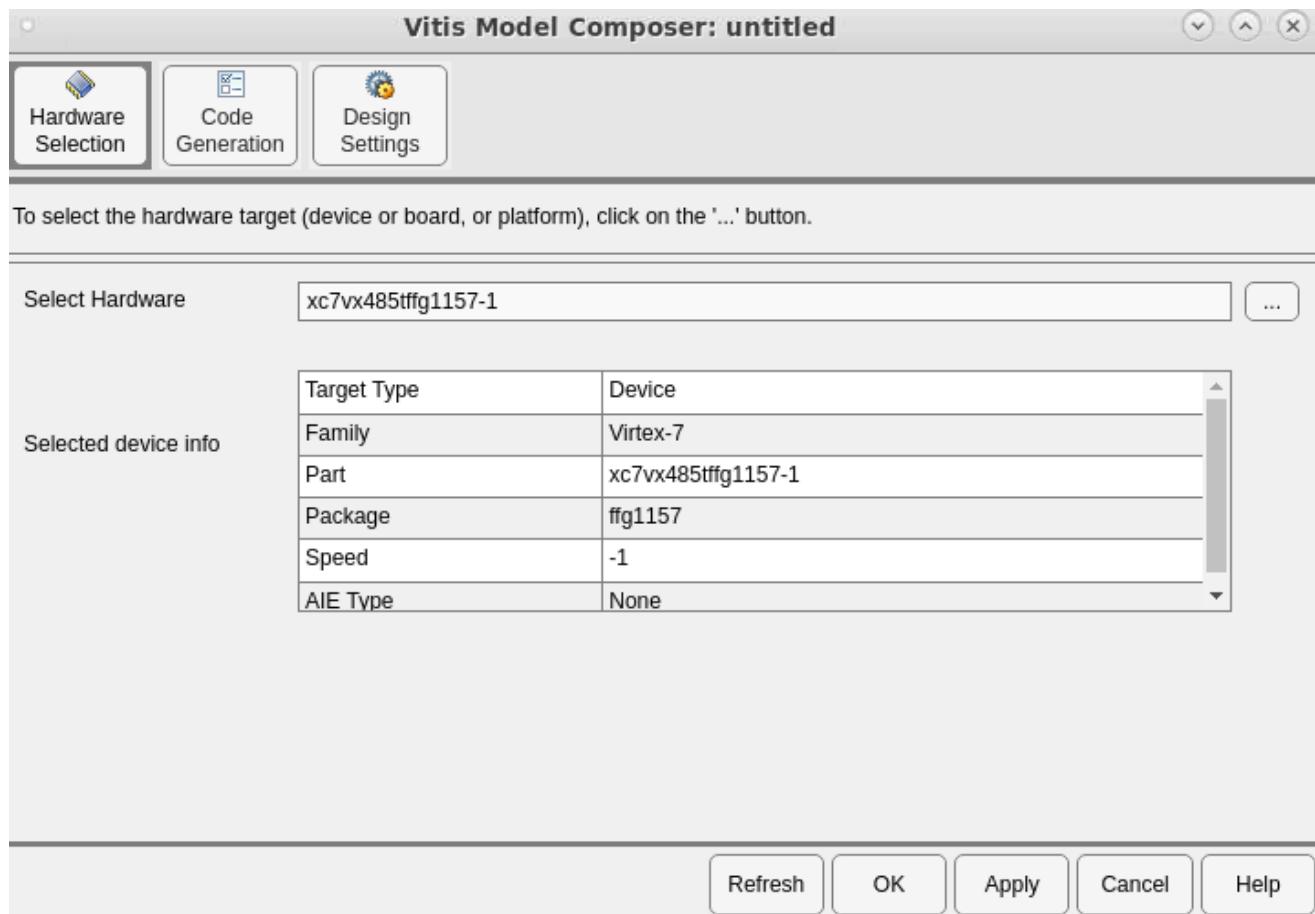
## Compiling a Model for Hardware Co-Simulation

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

For information on how to use the Model Composer Hub, see [Compiling and Simulating Using the Model Composer Hub](#).

To compile your Vitis Model Composer model for hardware co-simulation, perform the following:

1. Double-click the **Vitis Model Composer Hub** block to open the Model Composer Hub dialog box.



2. In the Hardware Selection tab, click the button next to the Select Hardware field to open the Device Chooser.
3. In the Boards tab, select a Board.

The boards displaying in the Board list are:

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

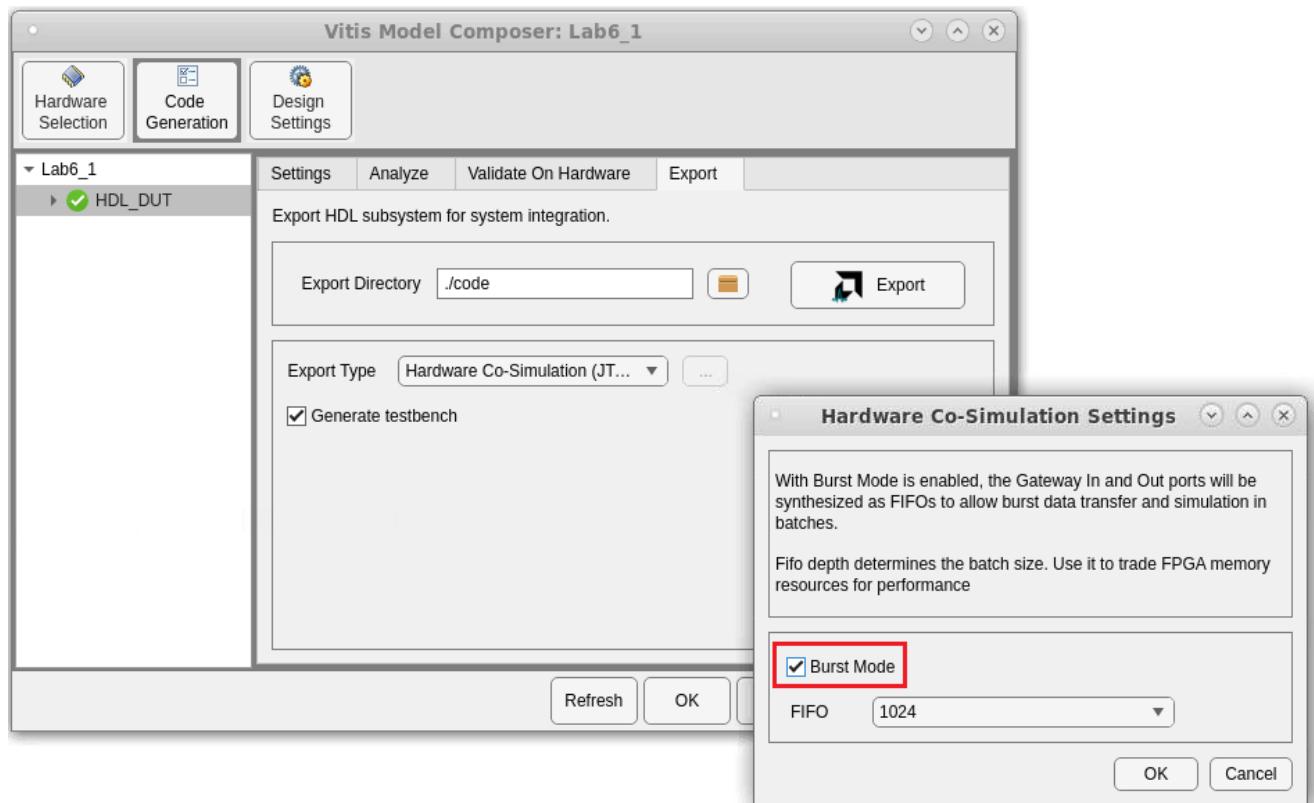
For a Partner board or a custom board to appear in the Board list, you must configure Vitis Model Composer to access the board files that describe the board. Board awareness in Model Composer is detailed in [Specifying Board Support in Model Composer](#).

To compile for hardware co-simulation, you must select a Board. You cannot select a Part or Platform.

4. Click **OK** to close the Device Chooser.
5. In the Code Generation pane, select the desired HDL subsystem from the list on the left.
6. In the Export tab, select **Hardware Co-Simulation (JTAG)** as the Export Type.

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

For a description of the burst mode, see [Burst Data Transfers for Hardware Co-Simulation](#).



**IMPORTANT!** To perform a burst mode hardware co-simulation, you must create a test bench by checking the Generate testbench box on the Export tab of the Vitis Model Composer Hub block.

8. If you want to create a test bench as part of the compilation, select **Generate testbench** on the Export tab. If you select **Generate testbench**, the compilation will automatically create an example test bench for you. You can also create your own test bench for hardware co-simulation (see [M-Code Access to Hardware Co-Simulation](#)).
9. Click the **Export** button.

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

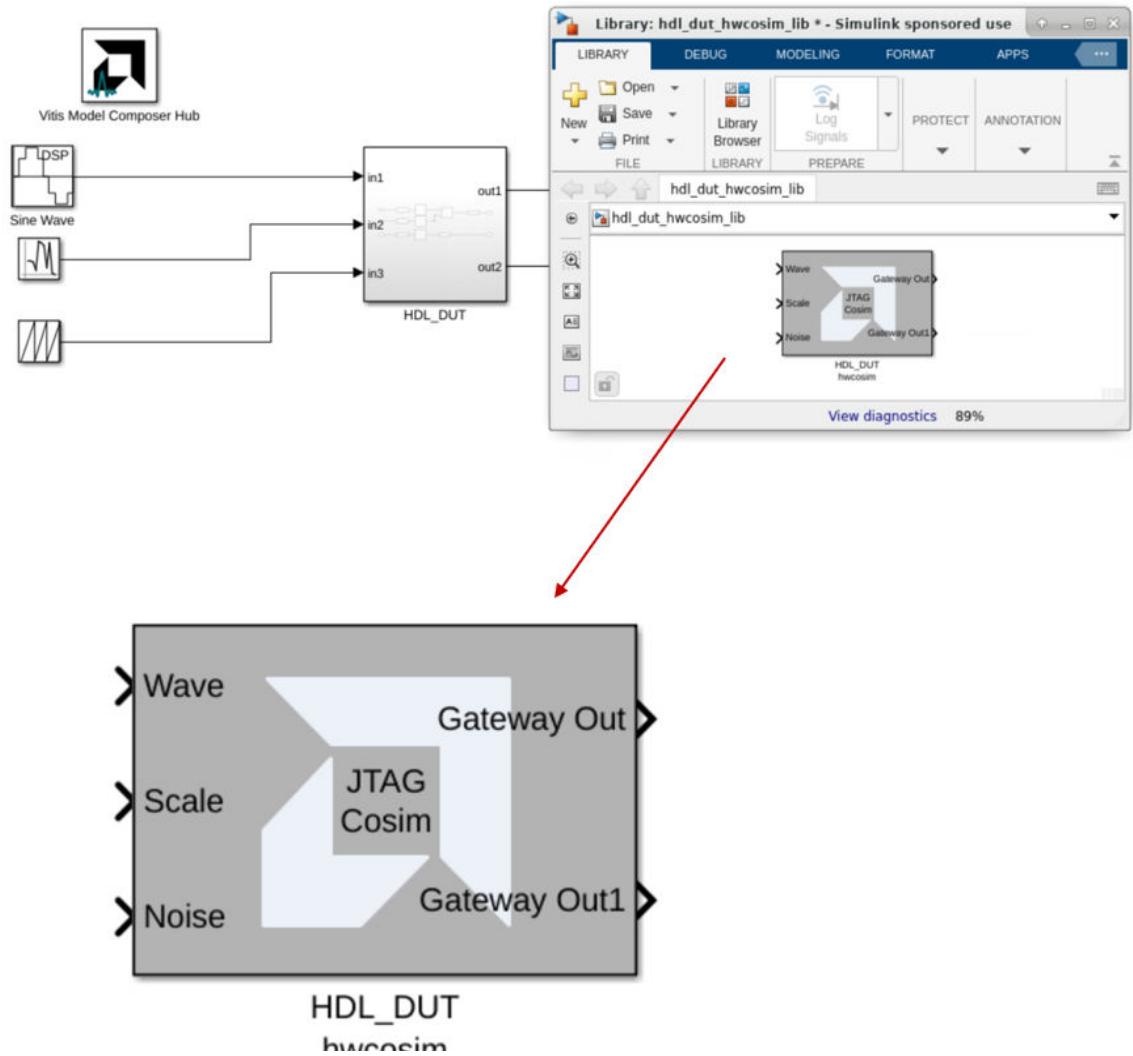
The configuration bitstream contains the hardware associated with your model, and also contains additional interfacing logic that allows Model Composer to communicate with your design using a physical interface between the board and the PC. This logic includes a memory map interface over which Model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

When the Compilation finishes the results are as follows:

- If you *have not* selected Burst mode in step 7 above (standard mode), a JTAG Cosim hardware co-simulation block will appear in a separate window. Drag (or Copy and Paste) the Hardware Cosim block into your Simulink model. The Hardware Cosim block will enable you to perform hardware co-simulation from within the Simulink window.

For a description of the hardware co-simulation block, see [Hardware Co-Simulation Blocks](#).

Figure 87: Hardware Co-Simulation Library Block



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

- If you have selected Burst mode in step 7 above (burst mode), no hardware co-simulation block will appear. When you perform the burst mode co-simulation, you will use the MATLAB® M-code test bench placed in the target directory during compilation.
  - If you compiled the top-level design the test bench will be named:

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

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

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

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

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

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

## Performing Standard Hardware Co-Simulation

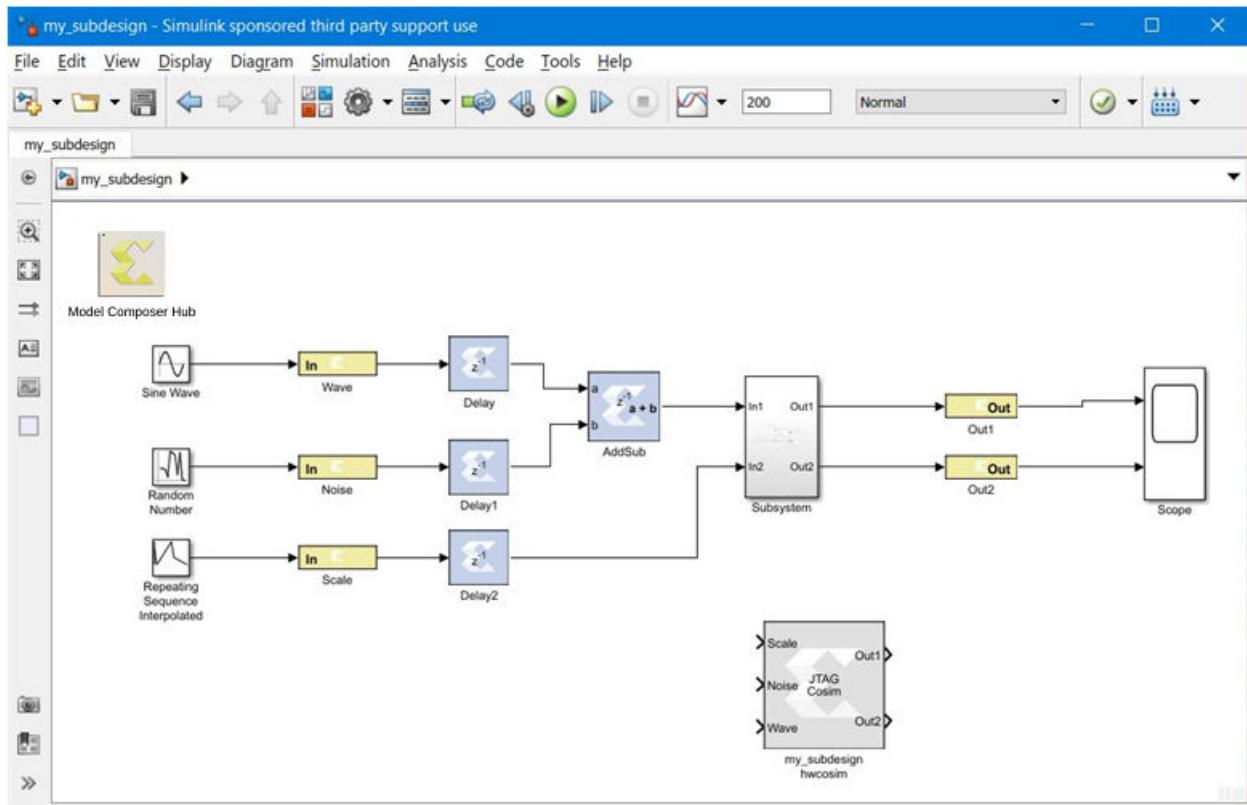
If you are performing the standard (non-burst mode) hardware co-simulation, your Simulink model will contain a JTAG hardware co-simulation block. This block was created automatically when Model Composer finished compiling your design into an FPGA bitstream (see [Compiling a Model for Hardware Co-Simulation](#)).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

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

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

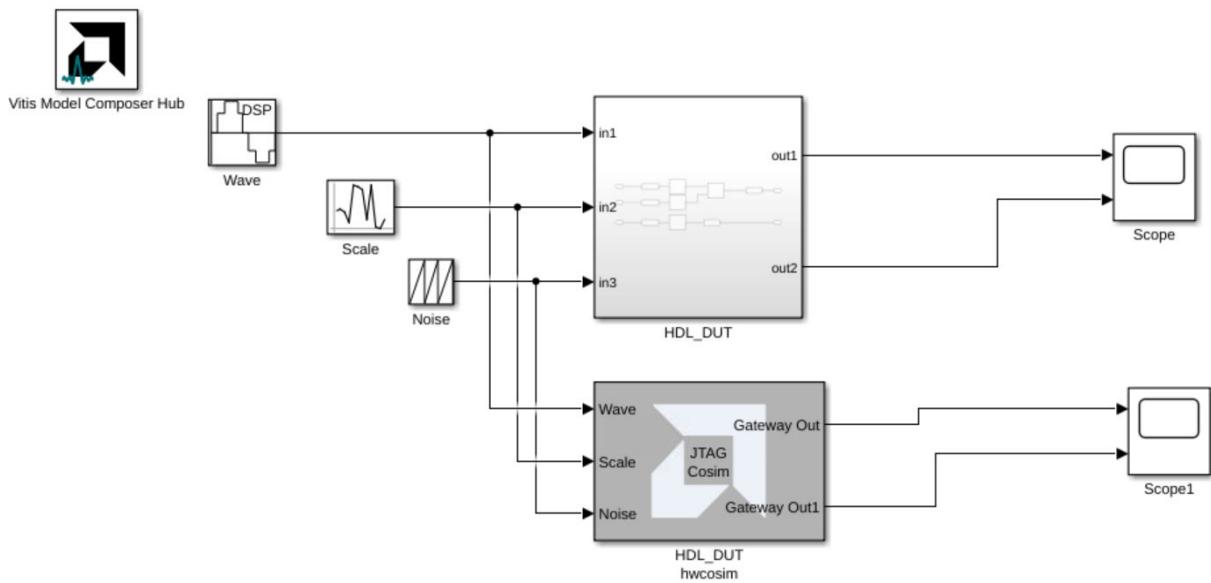
**Note:** If your board contains an AMD Zynq™ SoC device, you must install the AMD Vitis™ unified software platform with the AMD Vivado™ Design Suite to perform hardware co-simulation.

Figure 88: Hardware Co-Simulation Block

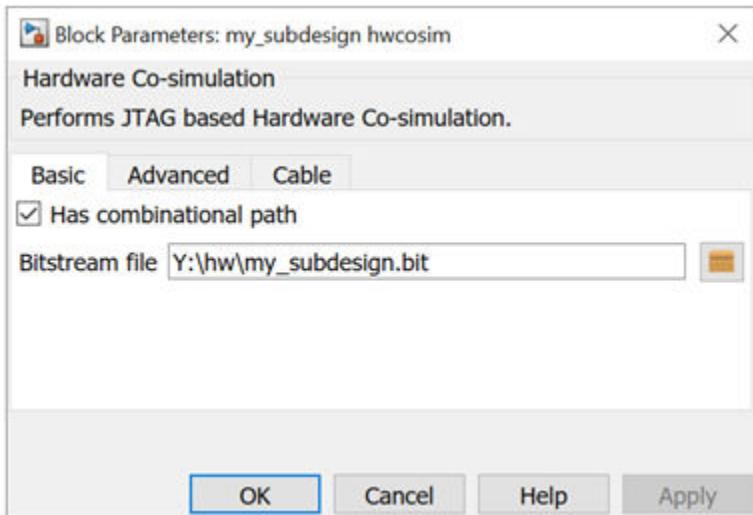


To perform the standard hardware co-simulation:

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



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



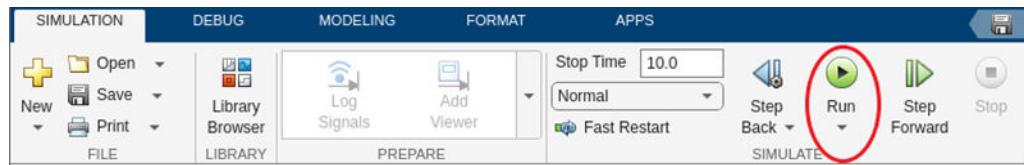
- Fill out the block parameters in the properties dialog box.

The properties are described in [Block Parameters for the JTAG Hardware Co-Simulation Block](#).

- To set up the board for performing JTAG hardware co-simulation, you should connect a cable to the board's JTAG port.

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

- In the Simulink model, simulate the model and the hardware by clicking the **Run** button on the Simulation tab.



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

## Performing Burst Mode Hardware Co-Simulation

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

This test bench resides in the `Target_directory` specified when the design was compiled for the hardware co-simulation compilation target.

The test bench is named as follows:

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

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

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

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

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

To perform burst mode hardware co-simulation, do the following:

1. Set up the board for performing JTAG hardware co-simulation.
  - Connect a cable to the board's JTAG port.  
For a description of the setup procedure for a JTAG hardware co-simulation, using a KC705 board as an example, see [Setting Up a KC705 Board for JTAG Hardware Co-Simulation](#).
2. Run the test bench script from the MATLAB console. To run the test bench script, you can open the MATLAB console, change directory to the HDL subsystem source directory (`<target_directory>/ip/<hdl_subsystem>/src`) and run the script by name.

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

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

The test bench then compares actual to expected outputs.

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

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

## M-Code Access to Hardware Co-Simulation

HDL Hardware co-simulation in Model Composer brings on-chip acceleration and verification capabilities into the Simulink simulation environment. In the typical Model Composer flow, a Model Composer model is first compiled for a hardware co-simulation platform, during which a hardware implementation (bitstream) of the design is generated and associated to a hardware co-simulation block. The block is inserted into a Simulink model and its ports are connected with appropriate source and sink blocks. The whole model is simulated while the compiled Model Composer design is executed on an FPGA.

Alternatively, it is possible to programmatically control the hardware created through the Model Composer HDL 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. Apart from supporting the scheduling semantics of a Model Composer simulation, M-Hwcosim also gives the flexibility for any arbitrary schedule to be used. This flexibility can be exploited to improve the performance of a simulation, if the user has apriori knowledge of how the design works. Additionally, the M-Hwcosim objects provide accessibility to the hardware from the MATLAB console, allowing for the hardware internal state to be introspected interactively.

### ***Compiling Hardware for Use with M-Hwcosim***

Compiling hardware for use in M-Hwcosim follows the same flow as the typical Model Composer HDL hardware co-simulation flow. You start off with a Model Composer model in Simulink, select a hardware co-simulation target in the Compiling Hardware for Use with M-Hwcosim and click **Generate**. At the end of the generation, a hardware co-simulation library is created.

Among other files in the netlist directory, you can find a bit file and an hwc file. The bit file corresponds to the FPGA implementation, and the hwc file contains information required for M-Hwcosim. Both bit file and hwc file are paired by name, for example, `mydesign_cw.bit` and `mydesign_cw.hwc`.

The hwc file specifies additional meta information for describing the design and the chosen hardware co-simulation interface. With the meta information, a hardware co-simulation instance can be instantiated using M-Hwcosim, through which you can interact with the co-simulation engine.

M-Hwcosim inherits the same concepts of ports and fixed point notations as found in the existing co-simulation block. Every design exposes its top-level ports for external access.

## M-Hwcosim Simulation Semantics

The simulation semantics for M-Hwcosim differs from that used during hardware co-simulation in a Model Composer block diagram; the M-Hwcosim simulation semantics is more flexible and is capable of emulating the simulation semantics used in the block-based hardware co-simulation.

In the block-based hardware co-simulation, a rigid simulation semantic is imposed; before advancing a clock cycle, all the input ports of the hardware co-simulation are written to. Next all the output ports are read and the clock is advanced. In M-Hwcosim the scheduling of when ports are read or written to, is left to the user. For instance it would be possible to create a program that would only write data to certain ports on every other cycle, or to only read the outputs after a certain number of clock cycles. This flexibility allows users to optimize the transfer of data for better performance.

## Data Representation

M-Hwcosim uses fixed point data types internally, while it consumes and produces double precision floating point values to external entities. All data samples passing through a port are fixed point numbers. Each sample has a preset data width and an implicit binary point position that are fixed at the compilation time. Data conversions (from double precision to fixed point) happen on the boundary of M-Hwcosim. In the current implementation, quantization of the input data is handled by rounding, and overflow is handled by saturation.

## Interfacing to Hardware from M-Code

When a model has been compiled for hardware co-simulation, the generated bitstream can be used in both a model-based Simulink flow, or in M-code executed in MATLAB. The general sequence of operations to access a bitstream in hardware typically follows the sequence described below.

1. Configure the hardware co-simulation interface.  
**Note:** The hardware co-simulation configuration is persistent and is saved in the `hwc` file. If the co-simulation interface is not changed, there is no need to re-run this step.
2. Create a M-Hwcosim instance for a particular design.
3. Open the M-Hwcosim interface.
4. Repeatedly run the following sub-steps until the simulation ends.
5. Write simulation data to input ports.
6. Read simulation data from output ports.
7. Advance the design clock by one cycle.
8. Close the M-Hwcosim interface.
9. Release the M-Hwcosim instance.

## Automatic Generation of M-Hwcosim Testbench

M-Hwcosim enables the test bench generation for hardware co-simulation. When the Create testbench option is checked in the Model Composer Hub block, the hardware co-simulation compilation flow generates an M-code script (`<design>_hwcosim_test.m`) and golden test data files (`<design>_<port>_hwcosim_test.dat`) for each gateway based on the Simulink simulation. The M-code script uses the M-Hwcosim API to implement a test bench that simulates the design in hardware and verifies the results against the golden test data. Any simulation mismatch is reported in a result file (`<design>_hwcosim_test.results`).

As shown below in the Example, the test bench code generated is easily readable and can be used as a basis for your own simulation code.

### Example

```
function multi_rates_cw_hwcosim_test
try
% Define the number of hardware cycles for the simulation.
ncycles = 10;

% Load input and output test reference data.
testdata_in2 = load('multi_rates_cw_in2_hwcosim_test.dat');
testdata_in3 = load('multi_rates_cw_in3_hwcosim_test.dat');
testdata_in7 = load('multi_rates_cw_in7_hwcosim_test.dat');
testdata_pb00 =
load('multi_rates_cw_pb00_hwcosim_test.dat');
testdata_pb01 =
load('multi_rates_cw_pb01_hwcosim_test.dat');
testdata_pb02 =
load('multi_rates_cw_pb02_hwcosim_test.dat');
testdata_pb03 =
load('multi_rates_cw_pb03_hwcosim_test.dat');
testdata_pb04 =
load('multi_rates_cw_pb04_hwcosim_test.dat');

% Pre-allocate memory for test results.
result_pb00 = zeros(size(testdata_pb00));
result_pb01 = zeros(size(testdata_pb01));
result_pb02 = zeros(size(testdata_pb02));
result_pb03 = zeros(size(testdata_pb03));
result_pb04 = zeros(size(testdata_pb04));

% Initialize sample index counter for each sample period to
be
% scheduled.
insp_2 = 1;
insp_3 = 1;
insp_7 = 1;
outsp_1 = 1;
outsp_2 = 1;
outsp_3 = 1;
outsp_7 = 1;

% Define hardware co-simulation project file.
project = 'multi_rates_cw.hwc';

% Create a hardware co-simulation instance.
```

```
h = Hwcosim(project);

% Open the co-simulation interface and configure the
hardware.

try
open(h);
catch
% If an error occurs, launch the configuration GUI for the
user
% to change interface settings, and then retry the process
again.
release(h);
drawnow;
h = Hwcosim(project);
open(h);
end

% Simulate for the specified number of cycles.
for i = 0:(ncycles-1)

% Write data to input ports based their sample period.
if mod(i, 2) == 0
h('in2') = testdata_in2(insp_2);
insp_2 = insp_2 + 1;
end
if mod(i, 3) == 0
h('in3') = testdata_in3(insp_3);
insp_3 = insp_3 + 1;
end
if mod(i, 7) == 0
h('in7') = testdata_in7(insp_7);
insp_7 = insp_7 + 1;
end

% Read data from output ports based their sample period.
result_pb00(outsp_1) = h('pb00');
result_pb04(outsp_1) = h('pb04');
outsp_1 = outsp_1 + 1;
if mod(i, 2) == 0
result_pb01(outsp_2) = h('pb01');
outsp_2 = outsp_2 + 1;
end
if mod(i, 3) == 0
result_pb02(outsp_3) = h('pb02');
outsp_3 = outsp_3 + 1;
end
if mod(i, 7) == 0
result_pb03(outsp_7) = h('pb03');
outsp_7 = outsp_7 + 1;
end

% Advance the hardware clock for one cycle.
run(h);

end

% Release the hardware co-simulation instance.
release(h);

% Check simulation result for each output port.
logfile = 'multi_rates_cw_hwcosim_test.results';
logfd = fopen(logfile, 'w');
sim_ok = true;
```

```
        sim_ok = sim_ok & check_result(logfd, 'pb00',
testdata_pb00, result_pb00);
        sim_ok = sim_ok & check_result(logfd, 'pb01',
testdata_pb01, result_pb01);
        sim_ok = sim_ok & check_result(logfd, 'pb02',
testdata_pb02, result_pb02);
        sim_ok = sim_ok & check_result(logfd, 'pb03',
testdata_pb03, result_pb03);
        sim_ok = sim_ok & check_result(logfd, 'pb04',
testdata_pb04, result_pb04);
        fclose(logfd);
        if ~sim_ok
            error('Found errors in simulation results. Please refer to
''%s'' for details.', logfile);
        end

        catch
            err = lasterr;
            try release(h); end
            error('Error running hardware co-simulation testbench. %s',
err);
        end

%-----
function ok = check_result(fd, portname, expected, actual)
ok = false;

fprintf(fd, [ '\n' repmat('=', 1, 95), '\n']);
fprintf(fd, 'Output: %s\n\n', portname);

% Check the number of data values.
nvals_expected = numel(expected);
nvals_actual = numel(actual);
if nvals_expected ~= nvals_actual
    fprintf(fd, ['The number of simulation output values (%d)
differs ' ...
    'from the number of reference values (%d).\n'], ...
    nvals_actual, nvals_expected);
    return;
end

% Check for simulation mismatches.
mismatches = find(expected ~= actual);
num_mismatches = numel(mismatches);
if num_mismatches > 0
    fprintf(fd, 'Number of simulation mismatches = %d\n',
num_mismatches);
    fprintf(fd, '\n');
    fprintf(fd, 'Simulation mismatches:\n');
    fprintf(fd, '-----\n');
    fprintf(fd, '%10s %40s %40s\n', 'Cycle', 'Expected values',
'Actual values');
    fprintf(fd, '%10d %40.16f %40.16f\n', ...
    [mismatches-1; expected(mismatches); actual(mismatches)]);
    return;
end
```

```
ok = true;
fprintf(fd, 'Simulation OK\n');
```

## Resource Management

M-Hwcosim manages resources that it holds for a hardware co-simulation instance. It releases the held resources upon the invocation of the release instruction or when MATLAB exits. However, it is recommended to perform an explicit cleanup of resources when the simulation finishes or throws an error. To allow proper cleanup in case of errors, it is suggested to enclose M-Hwcosim instructions in a MATLAB try-catch block as illustrated below.

```
try
    % M-Hwcosim instructions here
catch
    err = lasterror;
    % Release any Hwcosim instances
    try release(hwcosim_instance); end
    rethrow(err);
end
```

The following command can be used to release all hardware co-simulation instances.

```
xlHwcosim('release'); % Release all Hwcosim instances
```

## M-Hwcosim MATLAB Class

The `Hwcosim` MATLAB class provides a higher level abstraction of the hardware co-simulation engine. Each instantiated `Hwcosim` object represents a hardware co-simulation instance. It encapsulates the properties, such as the unique identifier, associated with the instance. Most of the instruction invocations take the `Hwcosim` object as an input argument. For further convenience, alternative shorthand is provided for certain operations.

*Table 10: Operation Syntax*

Actions	Syntax
Constructor	<code>h = Hwcosim(project)</code>
Destructor	<code>release(h)</code>
Open hardware	<code>open(h)</code>
Close hardware	<code>close(h)</code>
Write data	<code>write(h, 'portName', inData);</code>
Read data	<code>outData = read(h, 'portName');</code>

**Table 10: Operation Syntax (cont'd)**

Actions	Syntax
Run	<code>run(h);</code>
Port information	<code>portinfo(h);</code>
Set property	<code>set(h, 'propertyName', PropertyValue);</code>
Get property	<code>PropertyValue = get(h, 'propertyName');</code>

## Constructor

### Syntax

```
h = Hwcosim(project);
```

### Description

Creates an Hwcosim instance. An instance is a reference to the hardware co-simulation project and does not signify an explicit link to hardware; creating a Hwcosim object informs the Hwcosim engine where to locate the FPGA bitstream, it does not download the bitstream into the FPGA. The bitstream is only downloaded to the hardware after an open command is issued.

The project argument should point to the hwc file that describes the hardware co-simulation.

Creating the Hwcosim object lists all input and output ports. The example below shows the output of a call to the Hwcosim constructor, displaying the ID of the object and a list of all the input and output gateways/ports.

```
>> h = Hwcosim(p)
Model Composer HDL Hardware Co-simulation Object
  id: 30247
  imports:
    gateway_in
    gateway_in2
  outputs:
    gateway_out
```

## Destructor

### Syntax

```
release(h);
```

### Description

Releases the resources used by the Hwcosim object h. If a link to hardware is still open, release will first close the hardware.

## Open Hardware

### Syntax

```
open(h);
```

### Description

Opens the connection between the host PC and the FPGA. Before this function can be called, the hardware co-simulation interface must be configured. The argument, h, is a handle to an Hwcosim object.

## Close Hardware

### Syntax

```
close(h);
```

### Description

Closes the connection between the host PC and the FPGA. The argument, h, is a handle to an Hwcosim object.

## Write Data

### Syntax

```
h('portName') = inData; %If inData is array, results in burst write.  
h('portName') = [1 2 3 4];  
write(h, 'portName', inData);  
write(h, 'portName', [1 2 3 4]); %burst mode
```

### Description

Ports are referenced by their legalized names. Name legalization is a requirement for VHDL and Verilog synthesis, and converts names into all lower-case, replaces white space with underscores, and adds unique suffixes to avoid namespace collisions. To find out what the legalized input and output port names are, run the helper command `portinfo(h)`, or see the output of Hwcosim at the time of instance creation.

`inData` is the data to be written to the port. Normal single writes are performed if `inData` is a scalar value. If burst mode is enabled and `inData` is a  $1 \times n$  array, it will be interpreted as a timeseries and written to the port via burst data transfer.

## Read Data

### Syntax

```
outData = h('portName');
outData = h('portName', 25); %burst mode
outData = read(h, 'portName');
outData = read(h, 'portName', 25); %burst
```

### Description

Ports are referenced by their legalized names (see previous class above).

If burst mode is enabled, and depending on whether the read command has 3 or 4 parameters (2 or 3 parameters in the case of a subscript reference `h('portName', ...)`), `outData` will be assigned a scalar or a  $1 \times n$  array. If an array, the data is the result of a burst data transfer.

## Run

### Syntax

```
run(h);
run(h, n);
run(h, inf); %start free-running clock
run(h, 0); %stop free-running clock
```

### Description

When the hardware co-simulation object is configured to run in single-step mode, the `run` command is used to advance the clock. `run(h)` will advance the clock by one cycle. `run(h,n)` will advance the clock by  $n$  cycles.

The `run` command is also used to turn on (and off) free-running clock mode: `run(h, inf)` will start the free-running clock and `run(h, 0)` will stop it.

A read of an output port will need to be preceded either by a 'dummy' `run` command or by a `write`, to force a synchronization of the read cache with the hardware.

## Port Information

### Syntax

```
portinfo(h);
```

## Description

This method will return a MATLAB struct array with fields `inports` and `outports`, which themselves are struct arrays holding all input and output ports, respectively, again represented as struct arrays. The fieldnames of the individual port structs are the legalized portnames themselves, so you might obtain a cell array of input port names suitable for `Hwcosim write` commands by issuing these commands:

```
a = portinfo(h);
inports = fieldnames(a.inports);
```

You can issue a similar series of commands for output ports (`outports`).

Additional information contained in the port structs are `simulink_name`, which provides the fully hierarchical Simulink name including spaces and line breaks, `rate`, which contains the signal's rate period with respect to the DUT clock, `type`, which holds the Model Composer data type information, and, if burst mode is enabled, `fifo_depth`, indicating the maximum size of data bursts that can be sent to Hardware in a batch.

## Set Property

### Syntax

```
set(h, 'propertyName', PropertyValue);
```

### Description

The `set` method sets or changes any of the contents of the internal properties table of the `Hwcosim` instance `h`. It is required that `h` already exists before calling this method.

### Examples

```
set(h, 'booleanProperty', logical(0));
set(h, 'integerProperty', int32(12345));
set(h, 'doubleProperty', pi);
set(h, 'stringProperty', 'Rosebud!');
```

## Get Property

### Syntax

The get property method returns the value of any of the contents of the internal properties table in the Hwcosim instance `h`, referenced by the `propertyName` key. It is required that `h` already exists before calling this method. If the `propertyName` key does not exist in `h`, the method throws an exception and prints an error message.

### Examples

```
bool_val = get(h, 'booleanProperty');

int_val = get(h, 'integerProperty');

double = get(h, 'doubleProperty');

str_val = get(h, 'stringProperty');
```

## M-Hwcosim Utility Functions

### xlHwcosim

#### Syntax

```
xlHwcosim('release');
```

#### Description

When M-Hwcosim objects are created global system resources are used to register each of these objects. These objects are typically freed when a release command is called on the object. xlHwcosim provides an easy way to release all resources used by M-Hwcosim in the event of an unexpected error. The release functions for each of the objects should be used if possible because the xlHwcosim call release the resources for all instances of a particular type of object.

#### Example

```
xlHwcosim('release') %release all instances of Hwcosim objects.
```

## Setting Up Your Hardware Board

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

For JTAG-based hardware co-simulation, you will connect a cable to the board's JTAG port. Consult your board's documentation for the location of the board's JTAG port. Documentation for AMD boards can be downloaded from the [Boards and Kits](#) page on the AMD website.

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

## ***Setting Up a KC705 Board for JTAG Hardware Co-Simulation***

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

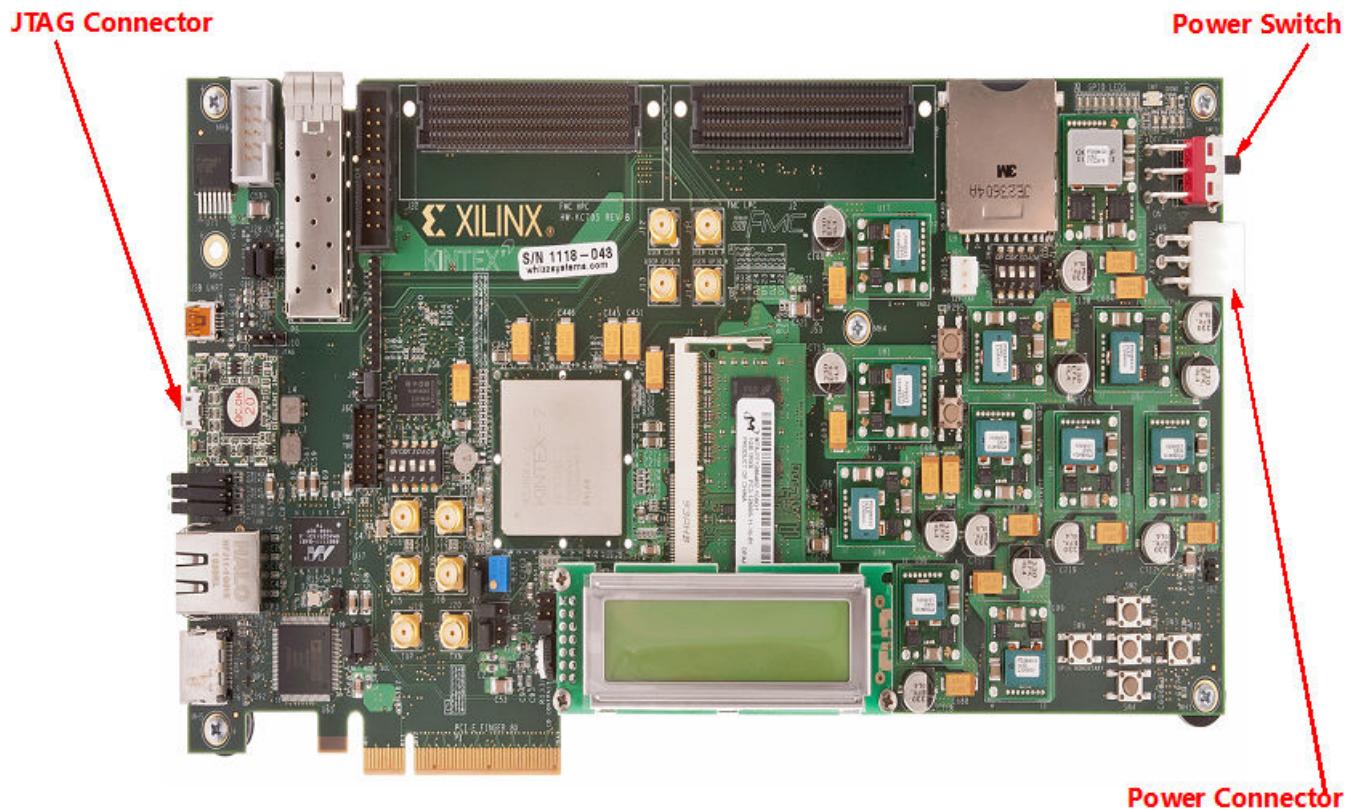
For detailed information about the KC705 board, see the *KC705 Evaluation Board for the Kintex 7 FPGA* ([UG810](#)).

### **Assemble the Required Hardware**

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

### **Set Up the KC705 Board**

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

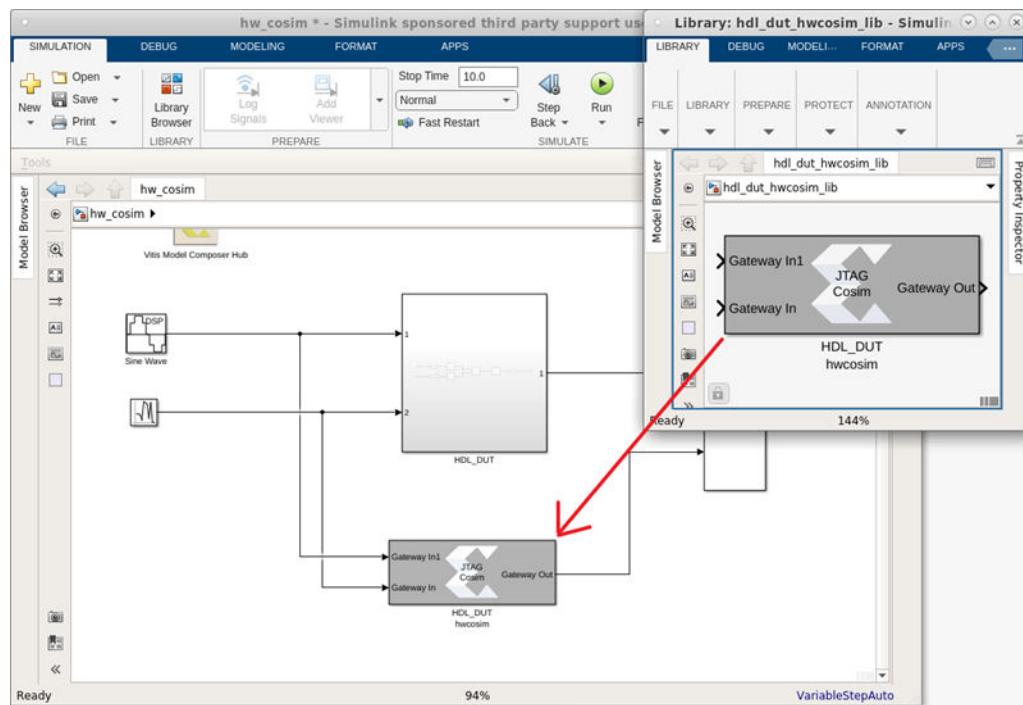
*Figure 89: KC705 Board*

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

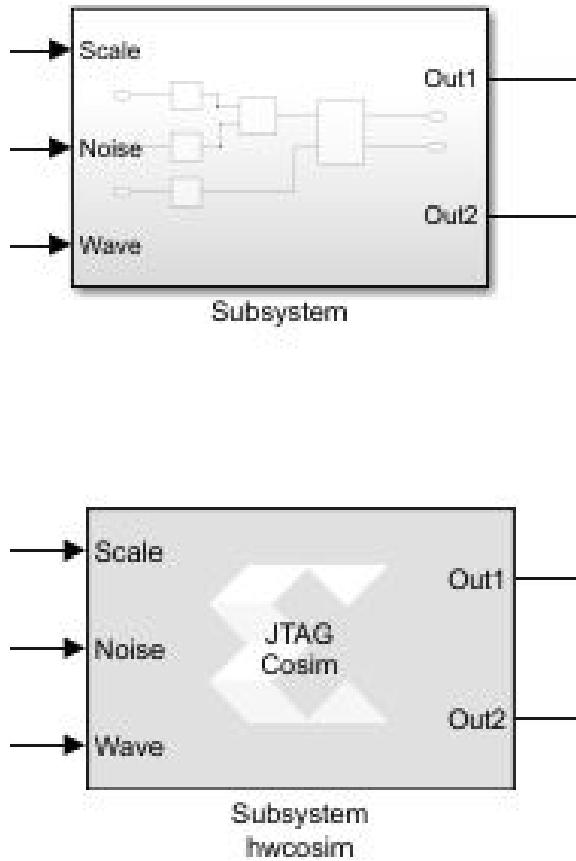
## Hardware Co-Simulation Blocks

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

Figure 90: Hardware Co-Simulation Blocks



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

**Figure 91: Port Names**

Hardware co-simulation blocks are used in a Simulink design the same way other blocks are used. During simulation, a hardware co-simulation block interacts with the underlying FPGA board, automating tasks such as device configuration, data transfers, and clocking. A hardware co-simulation block consumes and produces the same types of signals that other Model Composer HDL 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 can be driven by AMD 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 an HDL block, the output port produces an AMD 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 HDL 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 (that is, different FPGA boards provide their own customized hardware co-simulation blocks).

## ***Block Parameters for the JTAG Hardware Co-Simulation Block***

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

Parameters specific to the block are as follows:

### **Basic tab**

- **Has combinational path:** Select this if your circuit has any combinational paths. A combinational path is one in which a change propagates from input to output without any clock event. There is no latch, flip-flop, or register in the path. Enabling this option causes Vitis Model Composer to read the outputs immediately after writing inputs, before clocking the design. This ensures that value changes on combinational paths extending from the hardware co-simulation block into the Simulink Model get propagated correctly.
- **Bitstream file:** Specify the FPGA configuration bitstream. By default this field contains the path to the bitstream generated by Model Composer during the last Generate triggered from the Vitis Model Composer Hub block.

### **Advanced tab**

- **Skip device configuration:** When selected, the configuration bitstream will not be loaded into the FPGA or SoC. This option can be used if another program is configuring the device (for example, the Vivado Hardware Manager and the Vivado Logic Analyzer).
- **Display Part Information:** This option toggles the display of the device part information string (for example, xc7k325tffg900-2 for a Kintex device) in the center of the hardware co-simulation block.

### **Cable tab**

Cable Settings

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

## **Hardware Co-Simulation Clocking**

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

## Clocking Modes

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

### Single-Step Clock

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

Because the hardware co-simulation block is in effect producing the clock signal for the FPGA hardware only when Simulink awakes it, the overhead associated with the rest of the Simulink model's simulation, and the communication overhead (for example, bus latency) between Simulink and the FPGA board can significantly limit the performance achieved by the hardware. As long as the amount of computation inside the FPGA is significant with respect to the communication overhead (for example, 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, because 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 Model Composer 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 Model Composer 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 Model Composer 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 can change the co-simulation clocking mode before simulation starts by selecting either the Single stepped or Free running radio button for Clock Source in the parameters dialog box.

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

Figure 92: Single Stepped Button



For a description of a way to programmatically turn on or off a free-running clock using M-Hardware Cosim, see the description of the Run operation under in [M-Hwcosim MATLAB Class](#).

## Burst Data Transfers for Hardware Co-Simulation

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

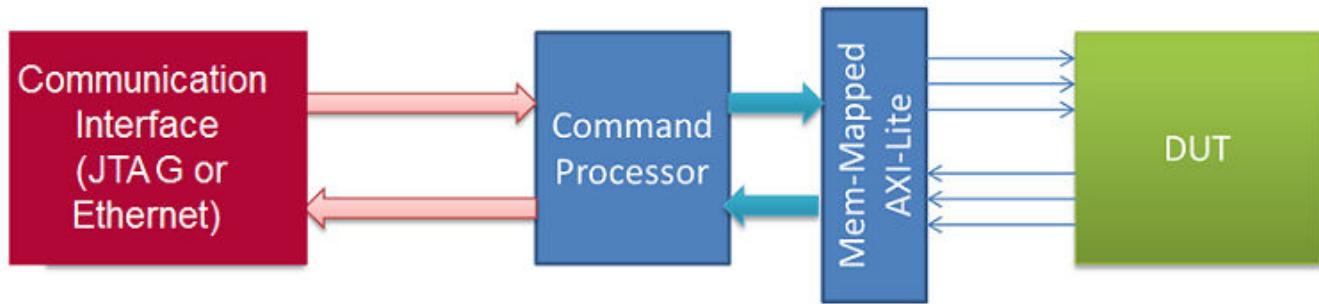
MATLAB/Simulink in conjunction with Vitis Model Composer currently supports two variants of HWCosim: GUI-based and MATLAB M-script-based. The first is run under the control of the Simulink scheduler, and can only progress one clock cycle at a time, due to the potential for feedback loops in the model.

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

## Hardware Co-Simulation Overview

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

Figure 93: Hardware Co-Simulation Flow

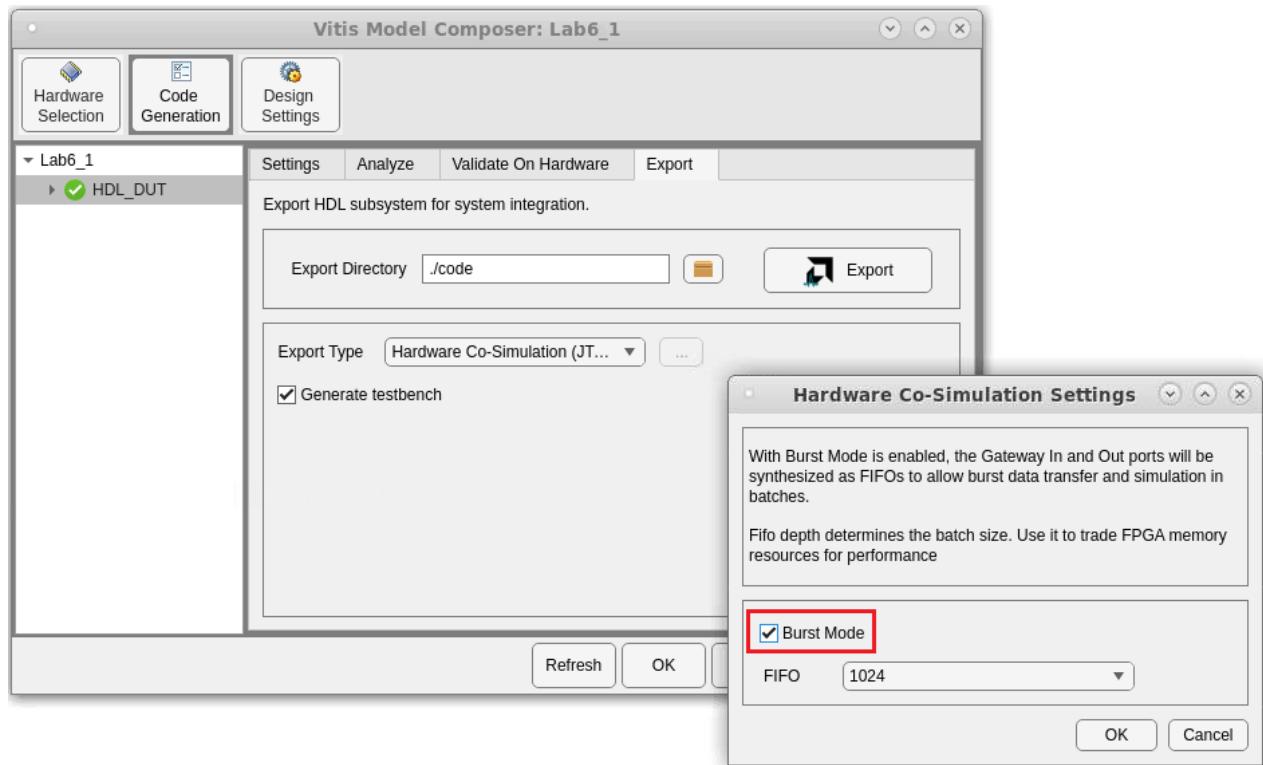


- **Communication interface (JTAG):** Used for communications with the host PC, receiving the command messages and sending responses.
- **Command processor:** Command messages are parsed and executed.
- **Memory-mapped AXI4-Lite register bank:** Use `write` commands to set up the stimulus data in the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

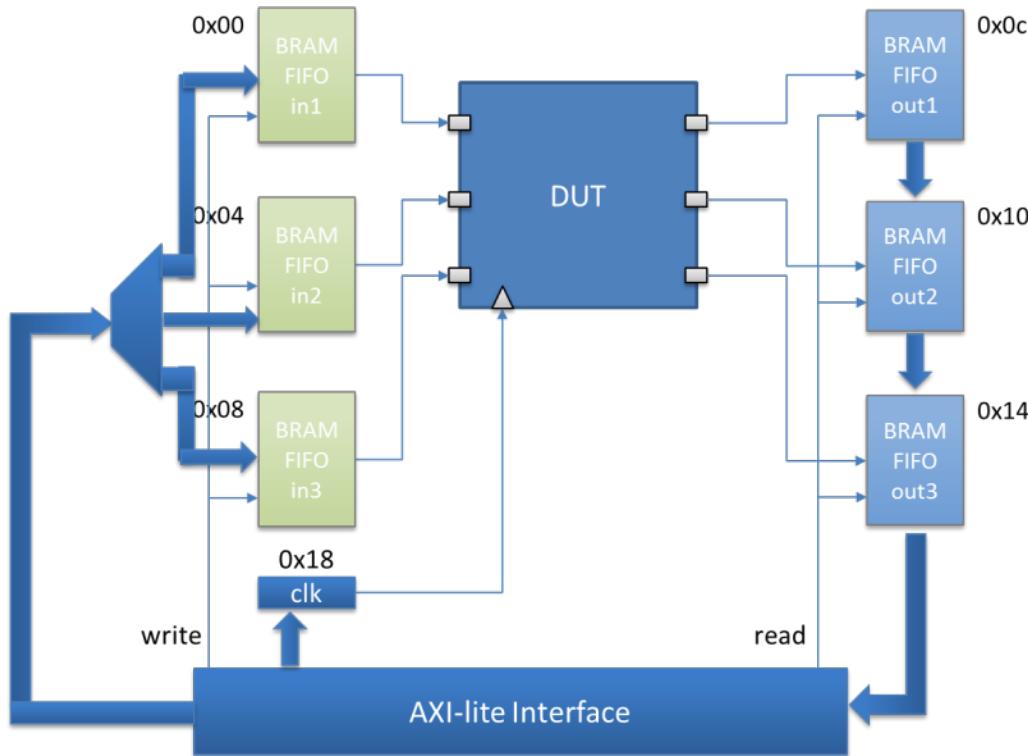
## Burst Data Transfer Mode

If you enable burst data transfer mode in the Vitis Model Composer Hub block, the non-clock input and output registers will be replaced with "n"-entry FIFOs. You can select "n" (FIFO depth), which is useful for trading off performance versus FPGA block RAM resource use.

Figure 94: Burst Mode



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

*Figure 95: Burst Mode Flow*

This batch processing of time samples allows to better pack data into JTAG sequences or thereby significantly reducing overhead.

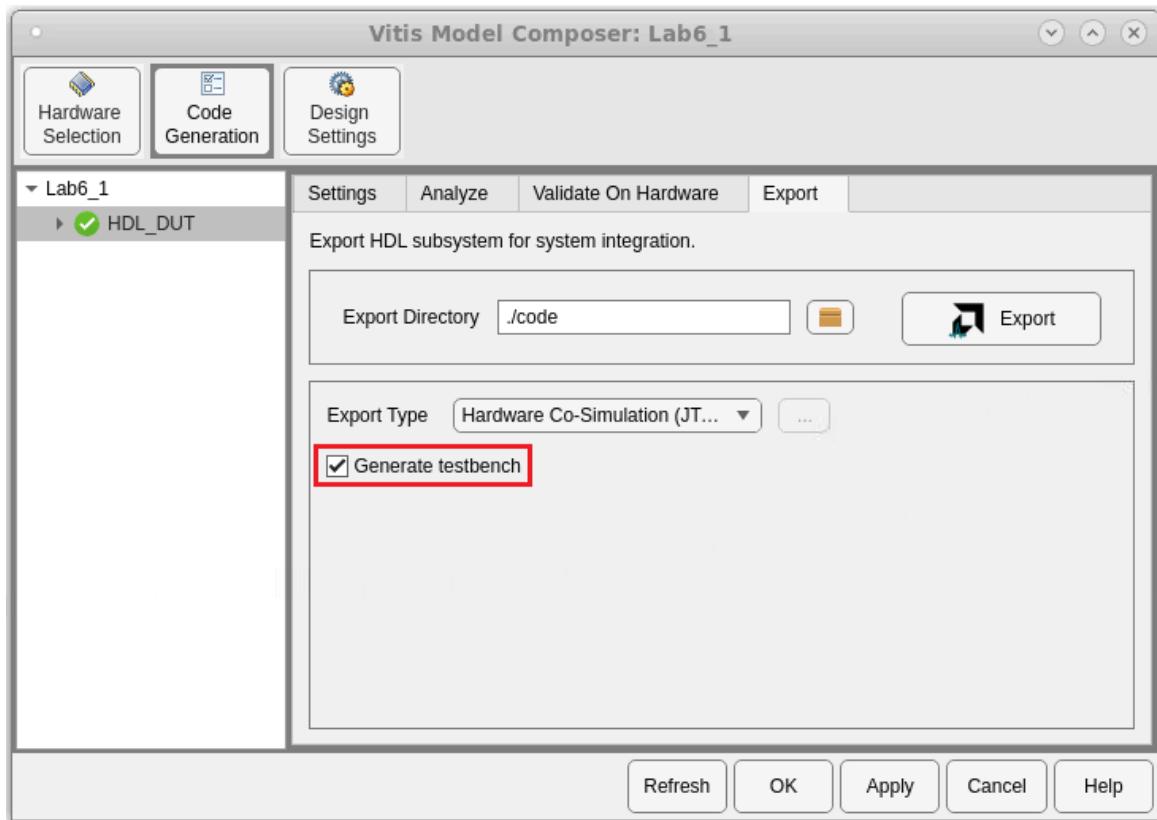
## ***How to Use Burst Data Transfer Mode***

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

### **Automatic Testbench Generation**

Testbench generation is run alongside the hardware co-simulation compilation flow. You can enable the automatic creation of an M-HWCosim test bench script by enabling **Generate testbench** at the bottom of the Export tab.

Figure 96: Create Test Bench



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

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

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

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

To run the test bench, you can open the MATLAB console, change directory to the HDL subsystem source directory (<target\_directory>/ip/<hdl\_subsystem>/src), and run the script by name. If the test fails this will be printed on the console, and the failing comparisons will be listed in this file:

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

## Burst Mode Testbench Script

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

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

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

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

function eta = project3_burst_hwcosim_test
eta = 0;

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

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

%%
% The testbench data vectors are both stimulus data for each input port, as
% well as expected (golden) data for each output port, recorded during the
% Simulink simulation portion of the Hardware Co-Simulation flow.
% Data gets loaded from the data file ('<name>_<port>_hwcosim_test.dat')
% into the corresponding 'testdata-<port>' workspace variables using
% 'getfield(load('<name>_<port>_hwcosim_test.dat' ...)' commands.
%
% Alternatively, the workspace variables holding the stimulus and / or
golden
% data can be assigned other data (including dynamically generated data) to
% test the design with. If using alternative data assignment, please make
% sure to adjust the "ncycles" variable to the proper number of cycles, as
% well as to disable the "Test Pass / Fail" section if unused.
testdata_noise_x0 =
getfield(load('project3_burst_noise_x0_hwcosim_test.dat', '-mat'),
'velues');
testdata_scale = getfield(load('project3_burst_scale_hwcosim_test.dat', '-
```

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

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

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

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

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

```

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

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

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

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

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

---

## Importing HDL Modules

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

*Table 11: Black Box Interface*

<a href="#">Black Box HDL Requirements and Restrictions</a>	Details the requirements and restrictions for VHDL, Verilog, and EDIF associated with black boxes.
<a href="#">Black Box Configuration Wizard</a>	Describes how to use the Black Box Configuration Wizard.
<a href="#">Black Box Configuration M-Function</a>	Describes how to create a black box configuration M-function.

**Table 12: HDL Co-Simulation**

<a href="#">Configuring the HDL Simulator</a>	Explains how to configure the AMD Vivado™ simulator or Questa to co-simulate the HDL in the Black Box block.
<a href="#">Co-Simulating Multiple Black Boxes</a>	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 Vitis Model Composer 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 Model Composer as ports; they only appear in the generated HDL after netlisting.
- For Verilog black boxes, the module and port names must follow standard HDL naming conventions.
- Any port that is a clock or clock enable must be of type `std_logic`. (For Verilog black boxes, ports must be of non-vector inputs, for example, `input clk`.)
- Clock and clock enable ports in black box HDL should be expressed as follows: Clock and clock enables must appear as pairs (that is, for every clock, there is a corresponding clock enable, and vice-versa). A black box can have more than one clock port and its behavior changes based on the context of the design.
  - In Synchronous single clock design context, a single clock source is used to drive each clock port. Only the clock enable rates differ.
  - In case of multiple independent hardware clock design context, two different clock sources is used to drive clock and clock enable pins.
- Each clock name (respectively, clock enable name) must contain the substring `clk`, for example `my_clk_1` and `my_ce_1`.
- The name of a clock enable must be the same as that for the corresponding clock, but with `ce` substituted for `clk`. For example, if the clock is named `src_clk_1`, then the clock enable must be named `src_ce_1`.
- Falling-edge triggered output data cannot be used.
- Fixed-point binary values with negative indices (for example, `(5 downto -3)`) are not supported.



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

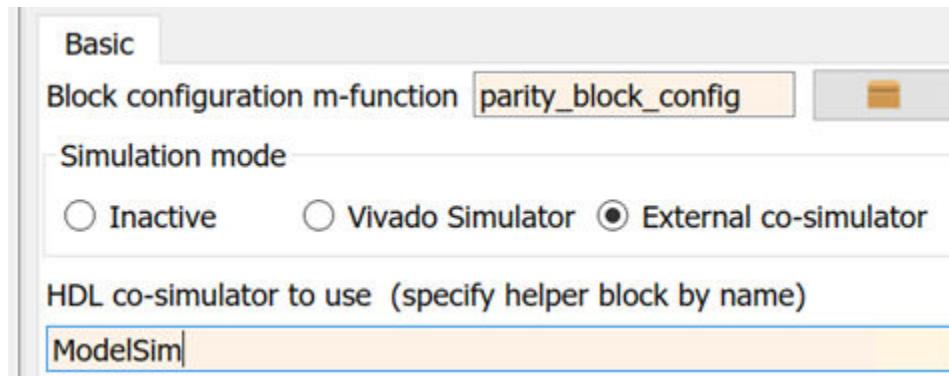
## Black Box Configuration M-Function

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

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

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

Figure 97: Parameter Dialog



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

```
function sample_block_config(this_block)
```

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

## Language Selection

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

VHDL Module:

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

Verilog Module:

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

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

## Specifying the Top-Level Entity

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

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

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

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

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

## Defining Port Blocks

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

## Adding New Ports

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

Adding an input port:

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

Adding an output port:

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

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

**Note:** Use lower case text to specify port names.

Adding a bidirectional port:

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

Bidirectional ports are supported only during the netlisting of a design and will not appear on the Model Composer diagram; they only appear in the generated HDL. As such, it is important to only add the bi-directional ports when Model Composer 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 `import` 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 (for example, 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 (for example, `std_logic`) or vectors (for example, `std_logic_vector(0 downto 0)`) notation. By default, Vitis Model Composer assumes ports to be declared as vectors. You can change the default behavior using the `useHDLVector` method of the descriptor. Setting this method to `true` tells Model Composer to interpret the port as a vector. A `false` value tells Model Composer to interpret the port as single bit.

```
dout.useHDLVector(true); % std_logic_vector
dout.useHDLVector(false); % std_logic
```

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

## Configuring Bi-Directional Ports for Simulation

Bidirectional ports (or inout ports) are supported only during the generation of the HDL netlist, that is, bi-directional ports will not show up in the Model Composer diagram. By default, bi-directional ports will be driven with 'X' during simulation. It is possible to overwrite this behavior by associating a data file to the port. Be sure to guard this code because bi-directional ports can only be added to a block during the `config_netlist_interface` phase.

```
if (strcmpi(this_block.getConfigPhaseString, 'config_netlist_interface') )  
    bidi_port = this_block.port('bidi');  
    bidi_port.setGatewayFileName('bidi.dat');  
end
```

In the above example, a text file, `bidi.dat`, is used during simulation to provide stimulation to the port. The data file should be a text file, where each line represents the signal driven on the port at each simulation cycle. For example, a 3-bit bi-directional port that is simulated for four 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 (for example., if the output port rate is different than the block's input sample rate).

**Note:** When the inputs to a black box have different sample rates, you must specify the sample rates of every output port.

`SysgenPortDescriptor` provides a method called `setRate` that allows you to explicitly set the rate of a port.

**Note:** The rate parameter passed to the `setRate` method is not necessarily the Simulink® sample rate that the port runs at. Instead, it is a positive integer value that defines the ratio between the desired port sample period and the Simulink® system clock period defined by the Model Composer Hub block dialog box.

Assume you have a model in which the Simulink system period value for the model is defined as 2 sec. Also assume that the example `dout` port is assigned a rate of 3 by invoking the `setRate` method as follows:

```
dout.setRate(3);
```

A rate of 3 means that a new sample is generated on the `dout` port every 3 Simulink system periods. Because the Simulink system period is 2 sec, this means the Simulink sample rate of the port is  $3 \times 2 = 6$  sec.

**Note:** If your port is a non-sampled constant, you can define it in the configuration M-function using the `setConstant` method of `SysgenPortDescriptor`. You can also define a constant by passing `Inf` to the `setRate` method.

## Dynamic Output Ports

A useful feature of the black box is its ability to support dynamic output port types and rates. For example, it is often necessary to set an output port width based on the width of an input port. `SysgenPortDescriptor` provides member variables that allow you to determine the configuration of a port. You can set the type or rate of an output port by examining these member variables on the block's input ports.

For example, you can obtain the width and rate of a port (in this case `din`) as follows:

```
input_width = this_block.port('din').width;
input_rate = this_block.port('din').rate;
```

**Note:** A black box's configuration M-function is invoked at several different times when a model is compiled. The configuration function can be invoked before the data types and rates are 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 Vitis Model Composer information about the module's clocking in the configuration M-function. Model Composer treats clock and clock enables differently than other types of ports. A clock port on an imported module must always be accompanied by a clock enable port (and vice versa). In other words, clock and clock enables must be defined as a pair, and exist as a pair in the imported module. This is true for both single rate and multirate designs.

Although clock and clock enables must exist as pairs, Model Composer 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 (for example, `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 Model Composer 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 Model Composer 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 (that is, 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 can specify a list of generics that get passed to the module when Vitis Model Composer compiles the model into HDL. Values assigned to these generics can be extracted from mask parameters and from propagated port information (for example, port width, type, and rate). This flexible means of generic assignment allows you to support highly parametric modules that are customized based on the Simulink environment surrounding the black box.

The `addGeneric` method allows you to define the generics that should be passed to your module when the design is compiled into hardware. The following code shows how to set a VHDL Integer generic, `dout_width`, to a value of 12.

```
addGeneric('dout_width','Integer','12');
```

It is also possible to set generic values based on port on propagated input port information (for example, 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 can be invoked before the data types (or rates) are 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 can 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 (for example, values that specify generic values) to the black box by doing the following:

- Copy a black box into a Simulink library or model.
- Break the link on the black box.
- Add the desired parameters to the black box dialog box.

## ***Black Box VHDL Library Support***

This Black Box feature allows you to import VHDL modules that have predefined library dependencies. The following example illustrates how to do this import.

The VHDL module below is a 4-bit, Up counter with asynchronous clear (`async_counter.vhd`). It will be compiled into a library named `async_counter_lib`.

**Figure 98: 4-bit, Up Counter with Asynchronous Clear**

```
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`.

Figure 99: 4-bit, Up Counter with Synchronous Clear

```
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 Black Box into your Model Composer model.

Figure 100: Top-level Module

```
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:

Figure 101: Black Box Configuration M-file

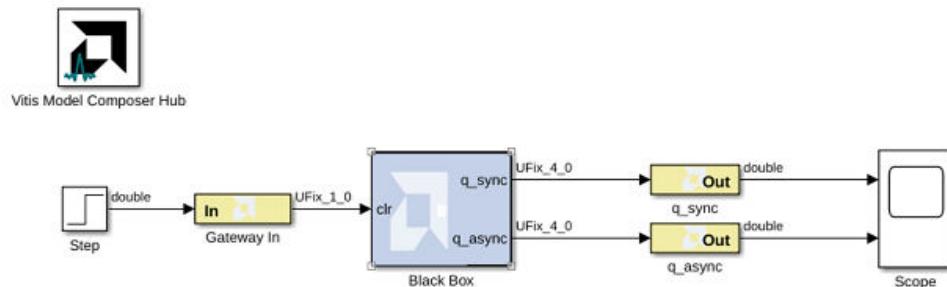
```
% Add additional source files as needed.
% |-----
% | Add files in the order in which they should be compiled.
% | If two files "a.vhd" and "b.vhd" contain the entities
% | entity_a and entity_b, and entity_a contains a
% | component of type entity_b, the correct sequence of
% | addFile() calls would be:
% |     this_block.addFile('b.vhd');
% |     this_block.addFile('a.vhd');
% | -----
Specifying library names by using  
"addFileToLibrary" command
%     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');
```



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 Model Composer model should look similar to the following figure.

Figure 102: Model Composer Model with Black Box Support



The next step is to double-click the Model Composer Hub block and click the Generate button to generate the HDL netlist.

During the generation process, a Vivado IDE project (.xpr) is created and placed with the `hdl_netlist` folder under the `netlist` folder. If you double-click the Vivado IDE project and select the Libraries tab under the Source view, you will see not only a `work` library, but an `async_counter_lib` library and `sync_counter_lib` library as well.

## Error Checking

It is often necessary to perform error checking on the port types, rates, and mask parameters of a black box. `SysgenBlockDescriptor` provides a method, `setError`, that allows you to specify an error message that is reported. The error message is the string parameter passed to `setError`.

## Black Box API

### SysgenBlockDescriptor Member Variables

Type	Member	Description
String	entityName	Name of the entity or module.
String	blockName	Name of the black box block.
Integer	numSimulinkImports	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 import(indx)). Sample period values are expressed as integer multiples of the Simulink System Period value specified by the Model Composer Hub block.
Boolean	error	true if an error has been detected, and false otherwise.
Cell Array of Strings	errorMessages	Array of all error messages for this block.

### SysgenBlockDescriptor Methods

Method	Description
setTopLevelLanguage(language)	Declares language for the top-level entity (or module) of the black box. The language should be VHDL or Verilog.
setEntityName(name)	Sets name of the entity or module.
addSimulinkImport(pname)	Adds an input port to the black box. pname defines the name the port should have.
addSimulinkOutport(pname)	Adds an output port to the black box. pname defines the name the port should have.
setSimulinkPorts(in,out)	Adds input and output ports to the black box. in (respectively, out) is a cell array whose element tell the names to use for the input (resp., output) ports.
addInoutport(pname)	Adds a bidirectional port to the black box. pname defines the name the port should have. Bidirectional ports can only be added during the config_netlist_interface phase of configuration.
tagAsCombinational()	Indicate that the block has a combinational path (that is, 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.

Method	Description
port(name)	Returns the <code>SysgenPortDescriptor</code> that matches the specified name.
inport(indx)	Returns the <code>SysgenPortDescriptor</code> that describes a given input port. <code>indx</code> tells the index of the port to look for, and should be between 1 and <code>numInputPorts</code> .
outport(indx)	Returns the <code>SysgenPortDescriptor</code> that describes a given output port. <code>indx</code> tells the index of the port to look for, and should be between 1 and <code>numOutputPorts</code> .
addGeneric(identifier, value)	Defines a generic (or parameter if using Verilog) for the block. <code>identifier</code> is a string that tells the name of the generic. <code>value</code> can be a double or a string. The type of the generic is inferred from <code>value</code> 's type. If <code>value</code> is an integral double (for example, 4.0), the type of the generic is set to integer. For a non-integral double, the type is set to real. When <code>value</code> is a string containing only zeros and ones (for example, `0101'), the type is set to <code>bit_vector</code> . 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. <code>identifier</code> tells the name, <code>type</code> tells the type, and <code>value</code> tells the value.
addFile(fn)	Adds a file name to the list of files associated to this black box, <code>fn</code> is the file name. Ordinarily, HDL files are associated to black boxes, but any sorts of files are acceptable. VHDL file names should end in <code>.vhd</code> ; Verilog file names should end in <code>.v</code> . 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 <code>.mdl</code> or library <code>.mdl</code> for the design.
getDeviceFamilyName()	Gets the name of the FPGA corresponding to the black box.
getConfigPhaseString	Returns the current configuration phase as a string. A valid return string includes: <code>config_interface</code> , <code>config_rate_and_type</code> , <code>config_post_rate_and_type</code> , <code>config_simulation</code> , <code>config_netlist_interface</code> , and <code>config_netlist</code> .
setSimulatorCompilationScript(script)	Overrides the default HDL co-simulation compilation script that the black box generates. <code>script</code> tells the name of the script to use. For example, this method can be used to short-circuit the compilation phase for repeated simulations where the HDL for the black box remains unchanged.
setError(message)	Indicates that an error has occurred, and records the error message. <code>message</code> gives the error message.
addDirectory(directoryName)	Adds the entire list of HDL files present in the specified directory to the Black Box flow. This is useful in case of the top level HDL file depends on other HDL files. This works fine in case of Simulation mode set to Vivado Simulator option.

## SysgenPortDescriptor Member Variables

Type	Member	Description
String	name	Tells the name of the port.

Type	Member	Description
Integer	simulinkPortNumber	Tells the index of this port in Simulink®. Indexing starts with 1 (as in Simulink).
Boolean	typeKnown	True if this port's type is known, and false otherwise.
String	type	Type of the port, such as UFix_<n>_<b>, Fix_<n>_<b>, or Bool.
Boolean	isBool	True if port type is Bool, and false otherwise.
Boolean	isSigned	True if type is signed, and false otherwise.
Boolean	isConstant	True if port is constant, and false otherwise.
Integer	width	Tells the port width.
Integer	binpt	Tells the binary point position, which must be an integer in the range 0..width.
Boolean	rateKnown	True if the rate is known, and false otherwise.
Double	rate	Tells the port sample time. Rates are positive integers expressed as MATLAB® doubles. A rate can also be infinity, indicating that the port outputs a constant.

## SysgenPortDescriptor Methods

Method	Description
setName(name)	Sets the HDL name to be used for this port.
setSimulinkPortNumber(num)	Sets the index associated with this port in Simulink®. num tells the index to assign. Indexing starts with 1 (as in Simulink).
setType(typeName)	Sets the type of this port to type. Type must be one of Bool, UFix_<n>_<b>, Fix_<n>_<b>, signed or unsigned. The last two choices leave the width and binary point position unchanged.  XFfloat_<exponent_bit_width>_fraction_bit_width > is also supported. For example: <pre>ap_return_port = this_block.port('ap_return'); ap_return_port.setType('XFfloat_30_2');</pre>
setWidth(w)	Sets the width of this port to w.
setBinpt(bp)	Sets the binary point position of this port to bp.
makeBool()	Makes this port Boolean.
makeSigned()	Makes this port signed.
makeUnsigned()	Makes this port unsigned.
setConstant()	Makes this port constant

Method	Description
setGatewayFileName(filename)	Sets the <code>.dat</code> 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: <code>std_logic</code> ) or vector (ex: <code>std_logic_vector(0 downto 0)</code> ).
HDLTypeIsVector()	Sets representation of the 1-bit port to <code>std_logic_vector(0 downto 0)</code> .

## Multiple Independent Clock Support on Black Box

### *Design Rule Checks on Port Connection*

When a black box is used in a multiple independent hardware clock design context, design rule checks (DRCs) for its port connections must be added in the configuration M-function. This helps to avoid invalid or incorrect port connection with different clock sources. You need to ensure all port signals are connected from/to a proper clocked-subsystem interface.

The utility `checkPortsOfSameClockDomain()` should be used to specify a list of ports from a particular clock domain and to group it together. The input arguments to this application programming interface (API) are '`SysgenBlockDescriptor`' objects followed by the list of port names associated with a particular clock domain.

In the example shown below, the API puts out an error check, and verifies that the four ports are connected to the same subsystem clock domain.

```
checkPortsOfSameClockDomain (<block_descriptor>, '<port_name_1>',  
'<port_name_2>',  
'<port_name_3>', '<port_name_4>' );
```

### *Configuring Port Sample Rates*

In multiple clock hardware designs, the clock period of the port interface should be computed using the connected "clocked subsystem domain." By default, "synchronous system clock" source is used by all the ports, but for asynchronous clock hardware designs, it is necessary to explicitly specify the clock sources of every port (for example, if the output port clock is different than the block's input port clock).

**Note:** You must set the sample rate to '1.0' for all output ports of multiple independent clock black box designs; it automatically sets the output ports to the destination clock subsystem period.

`SysgenPortDescriptor` provides a method called `setRate` that you can use to explicitly set the rate of a port.

Example:

```
port('<port_name>').setRate(1.0)
```

## ***Black Box Clocking***

In order to import a synchronous or asynchronous black box module, you must tell Model Composer information about the module's clocking in the configuration M-function. Model Composer treats clock and clock enables differently than other types of ports. A clock port on an imported module must always be accompanied by a clock enable port, and vice versa. In other words, clock and clock enables must be defined as a pair, and exist as a pair in the imported module. This is true for both single synchronous clock and multiple independent clock designs.

`SysgenBlockDescriptor` provides a method called `addClkCEPair` that you can use to define clock, clock enable, and its associated clock period by using clock sub-system domain. The clock domain information is not required for synchronous single clock designs.

The first parameter defines the name of the clock port (as it appears in the module). The second parameter defines the name of the clock enable port (also as it appears in the module).

The port names of a clock and clock enable pair must follow the naming conventions provided below:

- The clock port must contain the substring `clk`.
- The clock enable must contain the substring `ce`.
- The strings containing the substrings `clk` and `ce` must be the same, such as: `my_clk_1` and `my_ce_1`.

The third parameter defines the rate relationship between the clock and the clock-enable port. The rate parameter should not be thought of as a Simulink® sample rate. Instead, this parameter tells Model Composer the relationship between the clock sample period, and the desired clock enable sample period. The rate parameter is an integer value that defines the ratio between the clock rate and the corresponding clock enable rate.

For multiple independent clock designs, the fourth and fifth optional parameters are mandatory.

The fourth parameter holds a "Boolean" value, and it defines whether clock and clock enable pair is tied to ground. If you set it to `true`, both clock and clock enable would be tied to ground during simulation. Setting it to `false` would activate clock and clock enable rate transitions.

The fifth parameter defines the clock period for the corresponding clock-clock enable pair. The '`clockDomain`' property of the black box "`SysgenPortDescriptor`" must be used to set the clock periods for multiple independent clock designs.

Example:

```
rate_data = this_block.port('<port_name>').rate;
clkDomain_data = this_block.port(<port_name>).clockDomain;
this_block.addClkCEPair('clk',ce',rate_data, false, clkDomain_data);
```

## HDL Co-Simulation

This topic describes how a mixed language/mixed flow design that includes AMD HDL blocks, HDL modules, and a Simulink block design can be simulated in its entirety.

Model Composer simulates black boxes by automatically launching an HDL simulator, generating additional HDL as needed (analogous to an HDL test bench), compiling HDL, scheduling simulation events, and handling the exchange of data between the Simulink and the HDL simulator. This is called *HDL co-simulation*.

### **Configuring the HDL Simulator**

Black box HDL can be co-simulated with Simulink® using the Model Composer interface to either the AMD Vivado™ simulator or the Questa simulation software from Model Technology, Inc.

#### **AMD Simulator**

To use the AMD 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.

#### **Questa Simulator**

To use the Questa simulator by Model Technology, Inc., you must first add the Questa block that appears in the Tools library of the AMD HDL Blockset to your Simulink diagram.

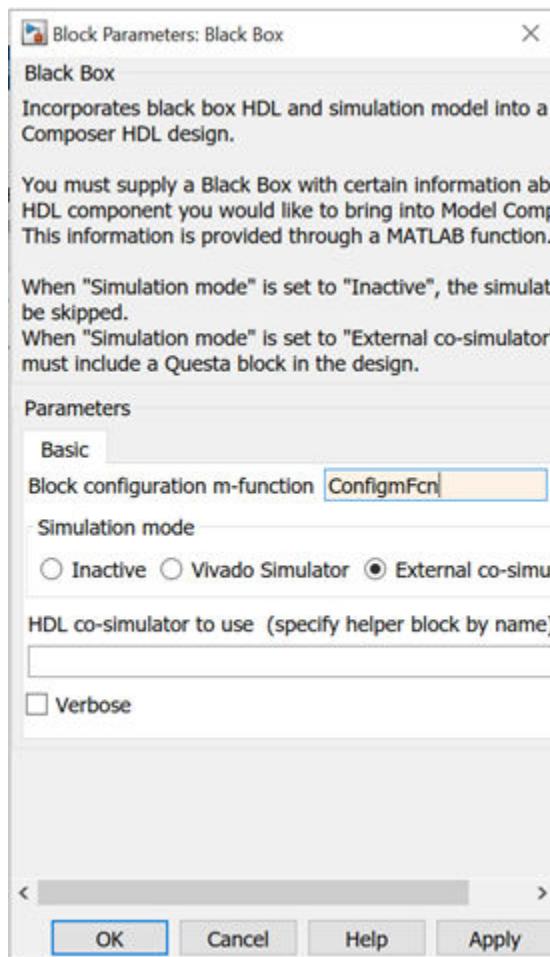
*Figure 103: Questa Block*



For each black box that you wish to have co-simulated using the Questa simulator, you need to open its block parameterization dialog and set it to use the Questa session represented by the black box that was recently 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 Questa block (for example, Questa) in the HDL co-simulator to use field.

**Figure 104: Black Box Parameters**



The block parameter dialog for the Questa block includes some parameters that you can use to control various options for the Questa 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***

Model Composer allows many black boxes to share a common Questa co-simulation session. For example, many black boxes can be set to use the same Questa block. In this case, Model Composer automatically combines all black box HDL components into a single shared top-level co-simulation component, which is transparent to the user. However, only one Questa simulation license is needed to co-simulate several black boxes in the Simulink® simulation.

Multiple black boxes can also be co-simulated with the Vivado simulator by selecting **Vivado Simulator** as the option for Simulation mode on each black box.

## ***Black Box Configuration Wizard***

Vitis Model Composer provides a configuration wizard that makes it easy to associate a VHDL or Verilog module to a Black Box block. The Configuration Wizard parses the VHDL or Verilog module that you are trying to import, and automatically constructs a configuration M-function based on its findings. Then, it associates the configuration M-function it produces to the Black Box block in your model. Whether or not you can use the configuration M-function as is depends on the complexity of the HDL you are importing. Sometimes the configuration M-function must be customized by hand to specify details the configuration wizard misses. Details on the construction of the configuration M-function can be found in the [Black Box Configuration M-Function](#) topic.

### ***Using the Configuration Wizard***

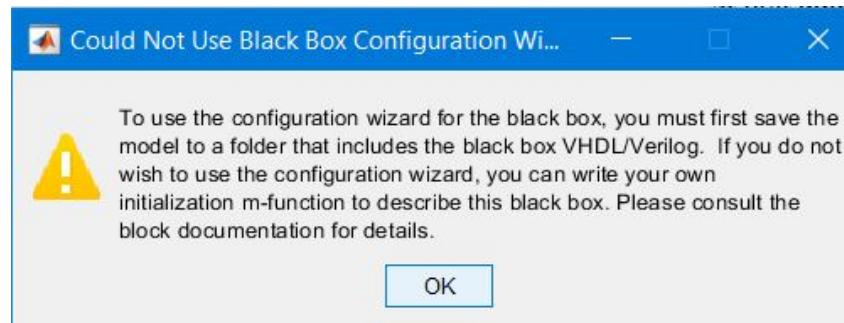
The Black Box Configuration Wizard opens automatically when a new black box block is added to a model.

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

For the Configuration Wizard to find your module, the model must be saved in the same directory as the module you are trying to import.

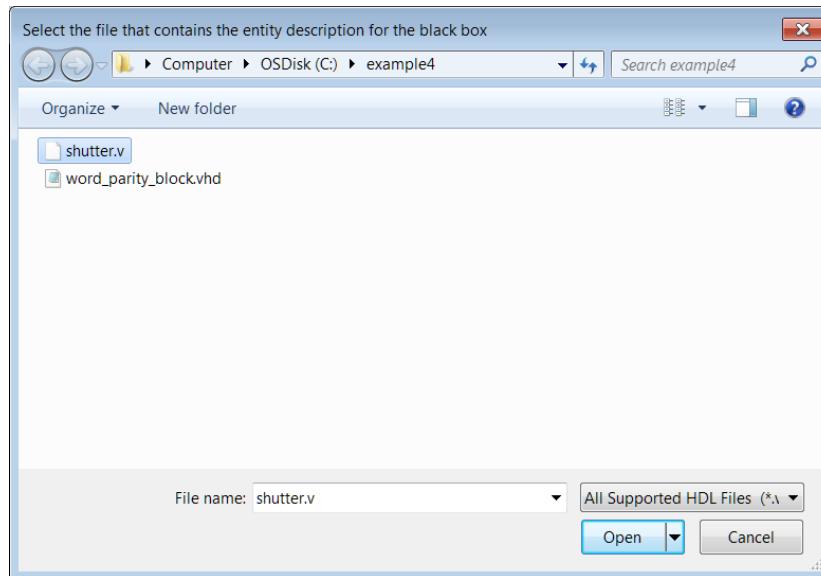
**Note:** The wizard only searches for .vhd and .v files in the same directory as the model. If the wizard does not find any files it issues a warning and the black box is not automatically configured. The warning looks like the following:

Figure 105: Warning



After searching the model's directory for .vhd and .v files, the Configuration Wizard opens a new window that lists the possible files that can be imported. An example screenshot is shown below:

Figure 106: Files to Import



You can select the file you would like to import by selecting the file, and then pressing the **Open** button. At this point, the configuration wizard generates a configuration M-function, and associates it with the black box block.

**Note:** The configuration M-function is saved in the model's directory as <module>\_config.m, where <module> is the name of the module that you are importing.

### Configuration Wizard Fine Points

The configuration wizard automatically extracts certain information from the imported module when it is run, but some things must be specified by hand. These things are described below:

**Note:** The configuration function is annotated with comments that instruct you where to make these changes.

- If your model has a combinational path, you must call the `tagAsCombinational` method of the block's `SysgenBlockDescriptor` object. A multiple independent hardware clock design will not support a combinational path.
- The Configuration Wizard only knows about the top-level entity that is being imported. There are typically other files that go along with this entity. These files must be added manually in the configuration M-function by invoking the `addFile` method for each additional file or by using the `addDirectory` method once (that is, by keeping the dependent files in a separate directory).
- The Configuration Wizard automatically creates either a synchronous single clock black box descriptor or an asynchronous multiple clock black box descriptor.
  - In the case of single-rate black box, every port on the black box runs at the same rate. In most cases, this is acceptable. You might want to explicitly set port rates, which can result in a faster simulation time.
  - In the case of a multiple clock black box, the input port rate must be derived from the "source clock subsystem" and the output port rate must be set based on the "destination clock subsystem." In some cases, you might want to explicitly set port rates for a required configuration.

---

## Compilation Types for HDL Library Designs

There are different ways in which Vitis Model Composer can compile your design into an equivalent, often lower-level, representation. The way in which a design is compiled depends on settings in the Model Composer Hub block 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.

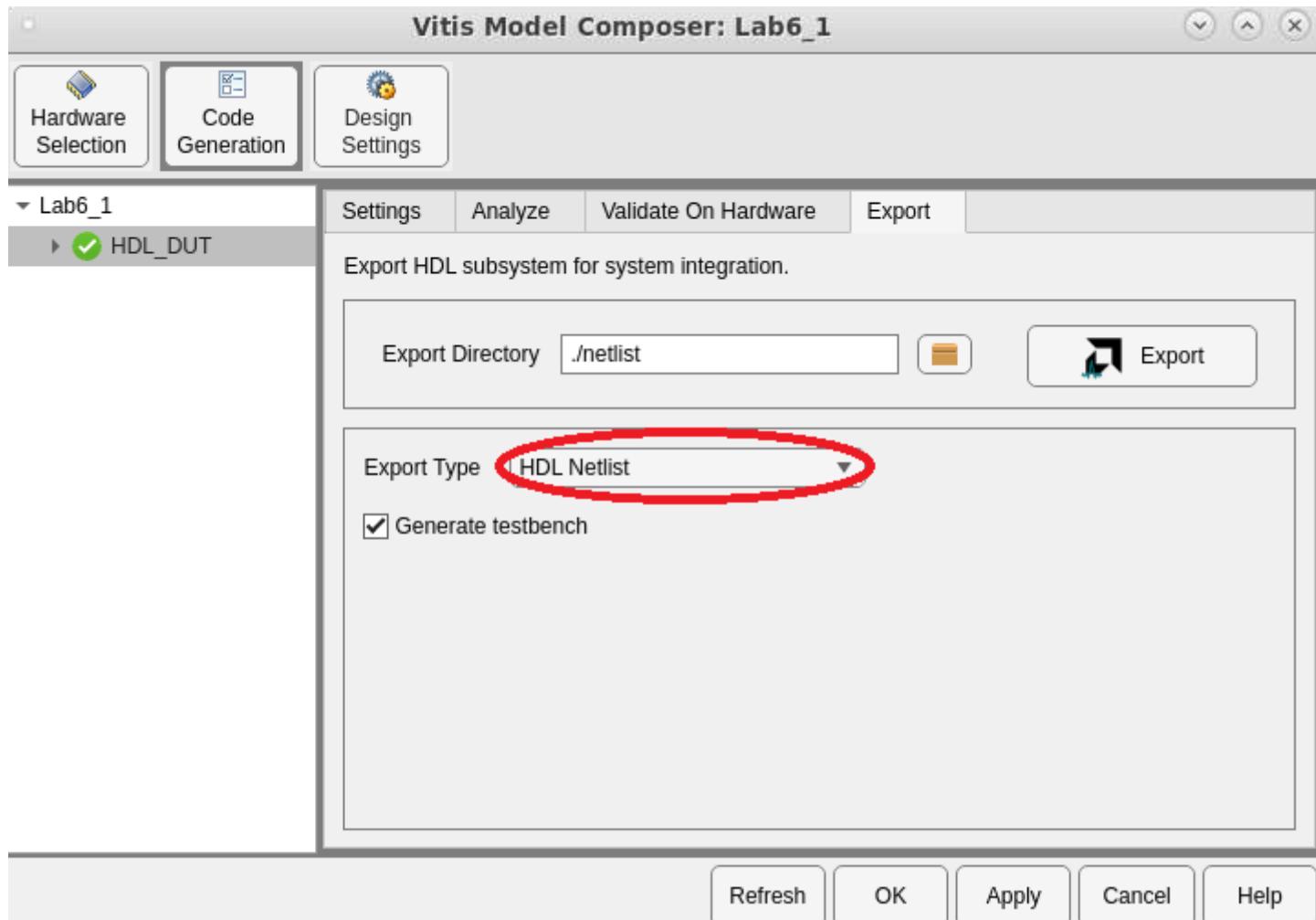
<a href="#">HDL Netlist Compilation</a>	Describes how to generate HDL files that implement the design.
<a href="#">Hardware Co-Simulation Compilation</a>	Describes how Model Composer can be configured to compile your design into FPGA hardware that can be used by Simulink® and Questa.
<a href="#">IP Catalog Compilation</a>	Describes how to package a Model Composer design as an IP core that can be added to the AMD Vivado™ IP catalog for use in another design. Model Composer uses the IP catalog compilation type as the default generation target.
<a href="#">Synthesized Checkpoint Compilation</a>	Describes how to generate a synthesized checkpoint file ( <code>synth_1.dcp</code> ) that can be used in a Vivado integrated design environment (IDE) project.

## HDL Netlist Compilation

The HDL Netlist compilation type produces HDL files that implement the design. More details regarding the HDL Netlist compilation flow can be found in the [Compilation Results](#) section.

As shown in the following figure, you can select HDL Netlist compilation by selecting it from the Export Type menu on the Export tab.

Figure 107: HDL Netlist



The HDL Netlist compilation can be performed for any of the boards or parts your Vivado tools support. In addition to accessing the AMD development boards installed as part of your Vivado installation, you can also specify Partner boards or custom boards (see [Specifying Board Support in Model Composer](#)).

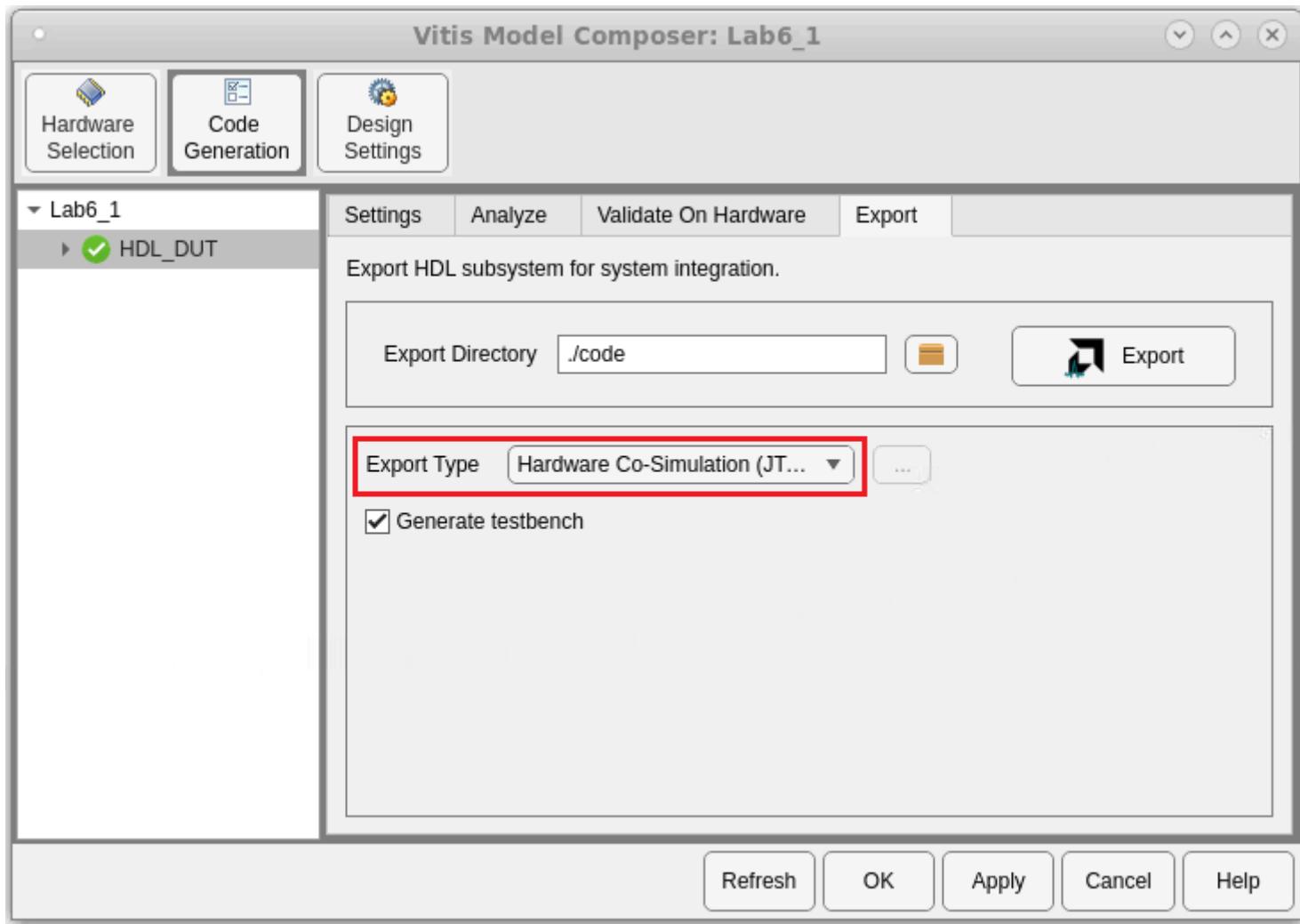
The files generated as part of an HDL Netlist compilation are placed in an `hdl_netlist` subdirectory under the `<code directory>/ip/<HDL Subsystem Name>` directory. These files are described in the [Compilation Results](#) section.

## Hardware Co-Simulation Compilation

Vitis Model Composer 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](#).

As shown in the following figure, you can select Hardware Co-Simulation compilation by selecting it from the Export Type menu on the Export tab.

Figure 108: Hardware Co-Simulation



JTAG Hardware Co-Simulation is supported for all AMD development boards.

The Simulink library (`<design_name>_hwcosim_lib.slx`) generated as part of a Hardware Co-Simulation compilation is placed in the directory you specified in the Export Directory field. This library, and the hardware co-simulation block stored in the library, are described in [Hardware Co-Simulation Blocks](#).

## IP Catalog Compilation

Vitis Model Composer uses the IP catalog compilation type as the default generation target.

The IP catalog compilation target allows you to package your Model Composer 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.

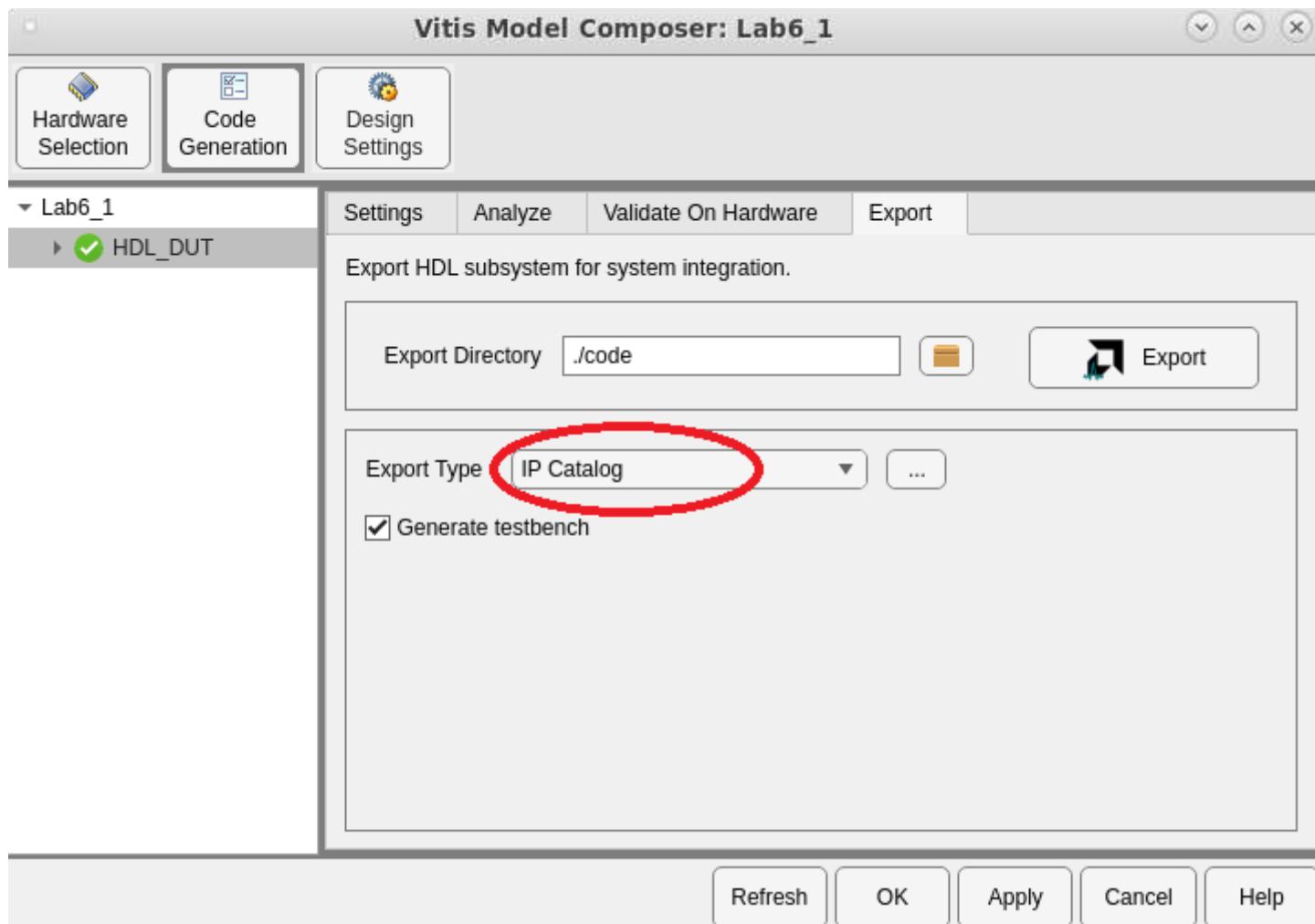
Model Composer first generates an HDL NetList based on the block design. If there are Vivado IP modules in the design, all the necessary IP files are copied into a subfolder named `IP`. Finally, all the RTL design files and Vivado IP design files are included into a ZIP file that is placed in a subfolder named `ip_catalog`.

### ***The IP Catalog Flow***

In a Vitis Model Composer design, double-click the **Vitis Model Composer Hub** block.

As shown in the following figure, you can select IP Catalog compilation by selecting it from the Export Type menu on the Export tab.

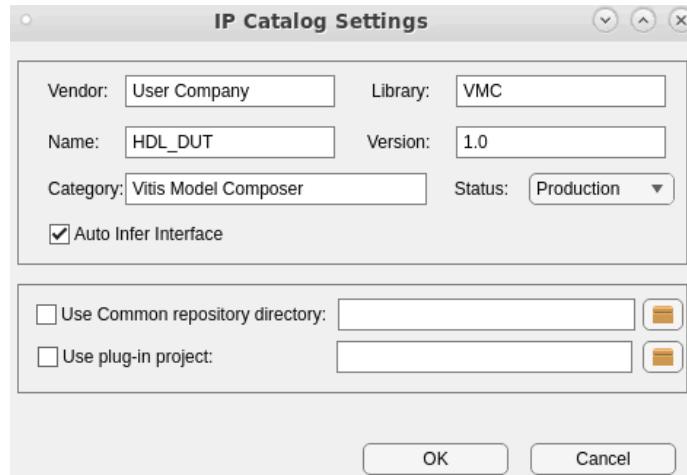
Figure 109: IP Catalog



The IP Catalog compilation can be performed for any of the boards or parts your Vivado tools support. In addition to accessing the AMD development boards installed as part of your Vivado installation, you can also specify Partner boards or custom boards (see [Specifying Board Support in Model Composer](#)).

The Export Directory field allows you to specify the location of the generated files.

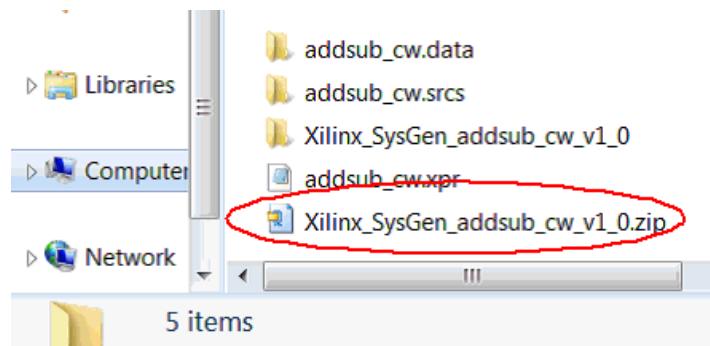
The ... to the right of the Compilation Type field allows you to enter information about the module that will appear in the Vivado IP catalog.

**Figure 110: IP Catalog Settings**

The Use common repository directory: field allows you to specify a directory referred to as the Common Repository. In an IP catalog compilation, the IP created is copied over to this location. If a Vivado user adds this Path as User Repository in the Vivado project's IP Settings, then all IPs that a Model Composer user has placed in this Common Repository are automatically picked up by Vivado and can be used either in an IP integrator or an RTL flow.

The Use Plug-in project field is used to specify a Vivado project containing an IP integrator Block Diagram (BD) that has been imported into Model Composer.

When you click the Export button, the IP catalog generation flow starts. Navigate to the specified HDL subsystem source directory (<target\_directory>/ip/<hdl\_subsystem>/src), to find a folder named ip\_catalog. This folder contains all the necessary files to form an IP from your Model Composer design. The ZIP file, circled below, contains all the files required to include the Model Composer design as IP in the Vivado IP catalog.

**Figure 111: ZIP File**

## Using AXI4 Interfaces

Selecting the Auto Infer Interface option in the IP Catalog Settings dialog box ensures AXI4 interfaces are automatically inferred from the design Gateway In and Gateway Out ports. The Auto Infer Interface option groups signals into AXI4-Stream, AXI4-Lite and AXI4 interfaces based on the port names.

The Auto Infer Interface option infers interfaces based on the following criteria:

- The Gateway In and Gateway Out port name suffix must exactly match the signal names in the AXI4 interface standard.
- The design must contain the minimum number of signals to be considered a valid AXI4 interface.

For example, if a design has two Gateway In ports named `PortName_tdata` and `PortName_tvalid`, and also a Gateway Out port named `PortName_tready`, the Auto Infer Interface option infers these three ports into a single AXI4-Stream port named `PortName`. In this example.

- The port name suffixes are exact matches for the signals in an AXI4-Stream interface (TDATA, TREADY, and TVALID).
- These three signals are the minimum signals required for an AXI4-Stream interface.

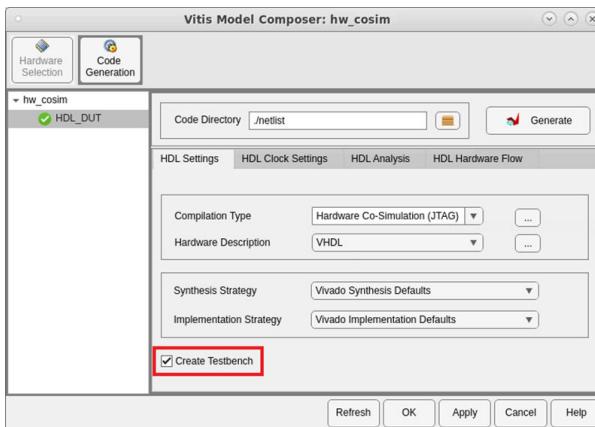
If optional AXI4 sideband signals are present, for example the TUSER signal is optional in the AXI4-Stream standard, and they are named using the same naming convention (for example, `PortName_tuser`) they are grouped into the same AXI4 Interface.

For more details on AXI4 interfaces, AXI4 interface signals names and the minimum required signals for an AXI4 interface, refer to the document *Vivado Design Suite: AXI Reference Guide* ([UG1037](#)).

## Including a Testbench with the IP Module

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

Figure 112: Create Testbench

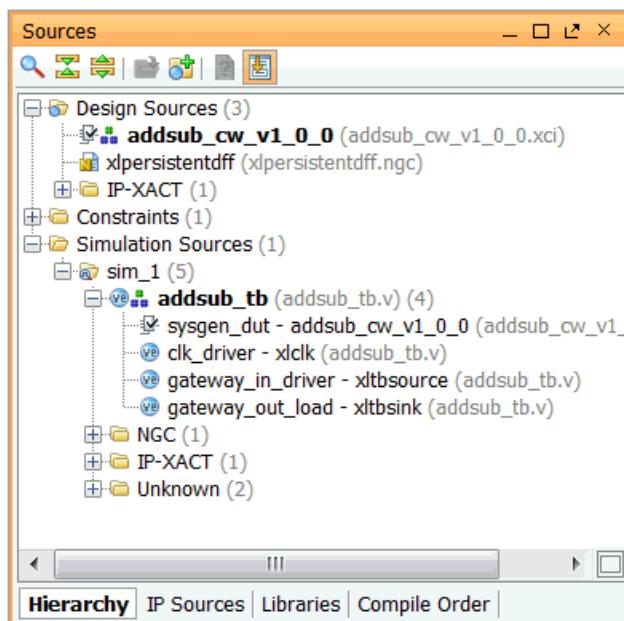


As shown below, when you include a test bench, you can verify the IP functionality by adding three more steps to the flow.

- **Step 1:** Add the new IP to the Vivado IP catalog. Refer to the document *Vivado Design Suite User Guide: Designing with IP* ([UG896](#)).
- **Step 2:** Create a new Vivado IDE project and add the IP as the top-level source.
- **Step 3:** Run simulation, synthesis, and implementation to verify the functionality of the generated IP.

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

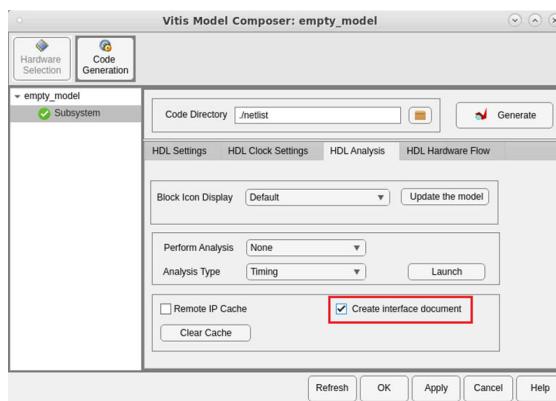
Figure 113: New IP



## ***Adding an Interface Document to the IP Module***

As shown below, select **Create interface document**, then click **Generate**, and Vitis Model Composer generates an interface document for the IP and packages this HTML document with the IP.

**Figure 114: Create Interface Document**



You can find a new folder, `documentation`, under the `netlist` folder. Right-click the new IP in the Vivado IDE, and click **Product guide**, to open one HTML file with interface information about this IP.

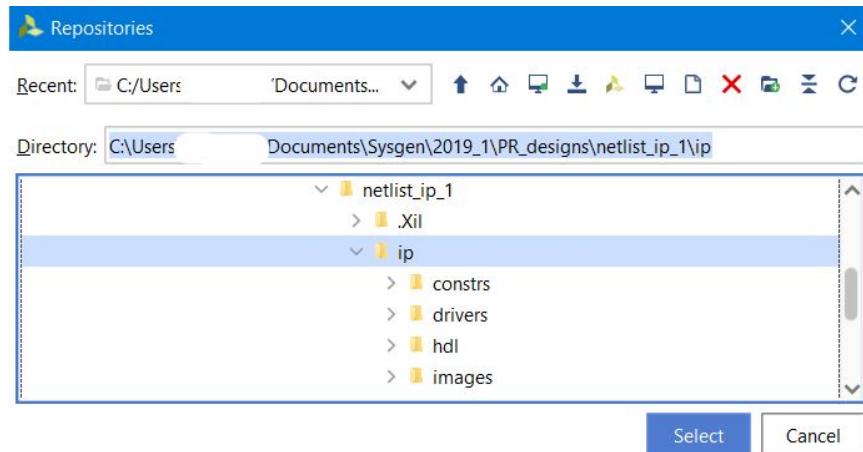
## ***Adding the Generated IP to the Vivado IP Catalog***

To use the IP generated from Vitis Model Composer, you need to create a new project, or open an existing project that targets the same device as specified in Model Composer for creating the IP.

**Note:** The IP is only accessible in this project. For each new project where you use this IP, you need to perform the same steps.

Select **IP Catalog** in the Project Manager, and right-click an empty area in IP catalog window. Select **Add Repository**, and add the directory that contains your new IP.

Figure 115: IP Catalog



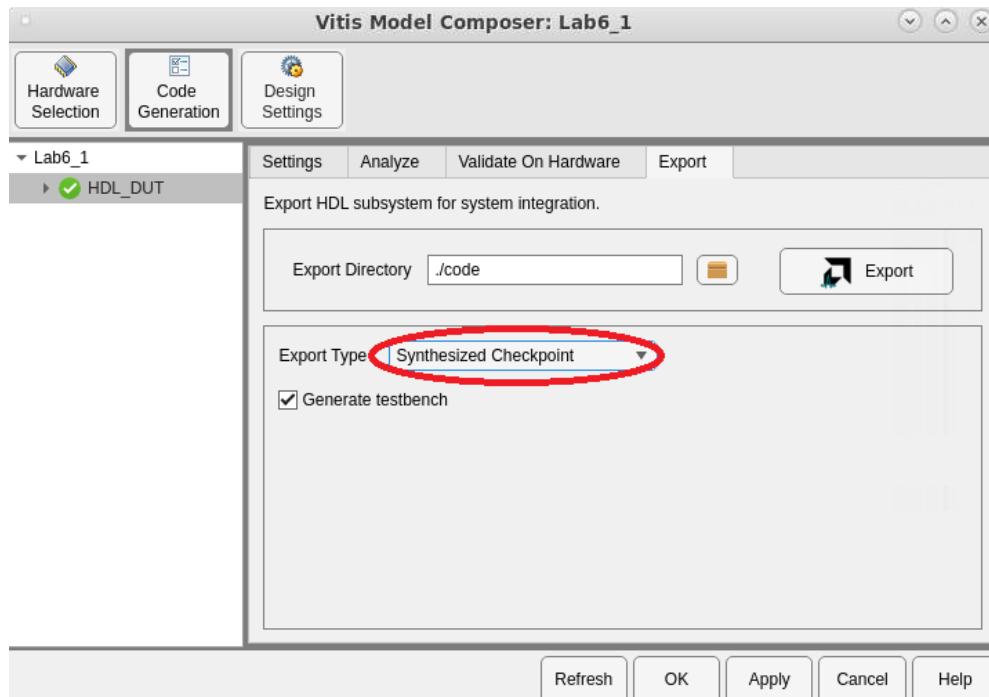
When the IP has been added to the IP catalog, you can include it in larger designs, as you would with any other IP in the IP catalog.

## Synthesized Checkpoint Compilation

Vivado tools provide design checkpoint files (.dcp) as a mechanism to save and restore a design at key steps in the design flow. Checkpoints are merely a snapshot of a design at a specific point in the flow. A Synthesized Checkpoint is a checkpoint file that is created in the out-of-context (OOC) mode after a design has been successfully synthesized.

When you select the Synthesized Checkpoint export type (see the following figure), a synthesized checkpoint target file named <design\_name>.dcp is created, and placed in the Export Directory. You can then use this <design\_name>.dcp file in any Vivado IDE project.

Figure 116: Synthesized Checkpoint



The Synthesized Checkpoint compilation can be performed for any of the boards or parts your Vivado tools support. In addition to accessing the AMD development boards installed as part of your Vivado installation, you can also specify partner boards or custom boards (see [Specifying Board Support in Model Composer](#)).

## Creating Your Own Custom Compilation Target

Vitis Model Composer provides a custom compilation infrastructure to create your own custom compilation target. In addition to generating HDL from your Model Composer design, you can also create a compilation target plug-in that automates steps both before and after the HDL is generated. Details about how to create a custom compilation target can be found in the topic [Creating Custom Compilation Targets](#).

---

## Creating Custom Compilation Targets

Model Composer provides a custom compilation infrastructure that allows you to create your own custom compilation targets. In addition to generating HDL from your Model Composer design, you can also create a compilation target plug-in that automates steps both before and after the AMD Vivado™ integrated design environment (IDE) project is created. In order to create a custom compilation target, you need to be familiar with the object-oriented programming concepts in the MATLAB® environment.

## xilinx\_compilation Base Class

The custom compilation infrastructure provides a base class named `xilinx_compilation`. From this base class, you can then create a subclass, and use its properties and override the member functions to implement your own functionality.

Figure 117: Base Class



## Creating a New Compilation Target

The following general procedure outlines how to create a new compilation target, and is followed by more specific examples.

### *Running the Helper Function*

Create a new custom compilation target by running the following helper function.

```
xilinx.environment.addCompilationTarget(target_name, directory_name)
```

For example, consider the following command:

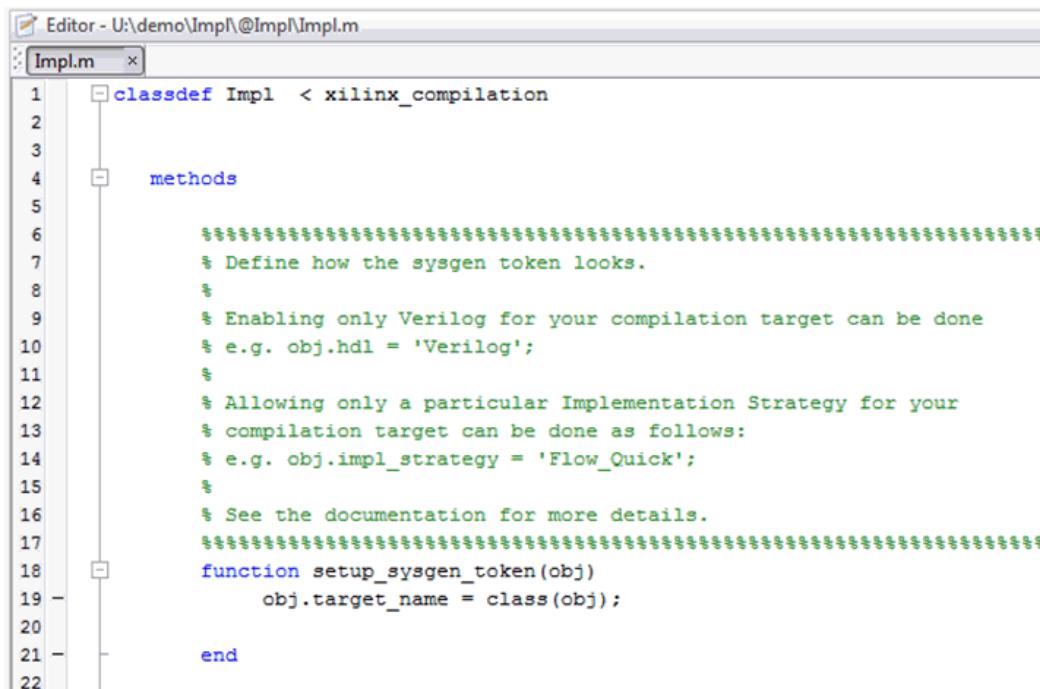
```
xilinx.environment.addCompilationTarget('Impl', 'U:\demo')
```

Figure 118: Helper Function



When you enter this command in the MATLAB Command Window as shown above, the following happens:

1. A folder is created named `Impl/@Impl` in `U:\demo`.
2. Inside the folder, a template class file `Impl` is created (`Impl.m`), which is derived from the base class `xilinx_compilation`. At this point, if no modifications are made to the file, the newly created `Impl` compilation target acts the same as the HDL Netlist compilation target. The content of the `Impl.m` file is shown in the following figure.



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

## 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 Vitis Model Composer detects the new class definition.

```
>> xilinx.environment.rehashCompilationTarget
```

## ***Adding an Existing Compilation Target***

You must add the path that contains the folder with the custom compilation target. As shown below, you can use the `addpath` functionality provided by MATLAB® to do this:

```
>>addpath('U:\demo\Impl');
```

When you use `addpath`, you must provide the absolute path, not the relative path.

## ***Saving a Custom Compilation Target***

You can use the `savepath` functionality in MATLAB® to save the custom compilation target. To do the save, you need write permission to the MATLAB installation area.

## ***Removing a Custom Compilation Target***

To remove the custom compilation target, remove the path to the target from the MATLAB® search path.

# **Base Class Properties and APIs**

The `xilinx_compilation` base class resides in the following location:

```
<install_dir>/<version>/Model_Composer/scripts/sysgen/matlab/  
@xilinx_compilation
```

## ***Vivado Project-Related Properties***

### **`top_level_module`**

You can use this property to set the top-level name of your choice. This parameter accepts a MATLAB® string.

## ***Vivado IDE Project Generation-Related Functions***

### **`pre_project_creation(design_info)`**

This function should be called before you create the AMD Vivado™ IDE project. Before the Model Composer Infrastructure creates the project, it has to know what files need to be added to the AMD 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 Model Composer design. For this purpose, a structure that describes the port interface is passed into this function called `design_info`. `design_info` is described in detail in a later section.

### **post\_project\_creation(design\_info)**

This function should be called at the end of Vivado IDE project creation. This is the last function to be called after the Project Generation script is run. This is a useful function for things like error parsing, generating reports, and opening the Vivado IDE project. A structure which describes the port interface is passed into this function called `design_info`. `design_info` is described in detail in a later section.

### **add\_tcl\_command(string)**

This function adds the additional Tcl commands as a string. These Tcl commands will be issued after the Vivado IDE project is created. Use this command to create a bitstream after project creation occurs. The Tcl command can also be used to source a particular Tcl file. The commands are executed in the order in which they are received.

### **add\_file(string)**

This function adds user-defined files to the Vivado IDE project. This application programming interface (API) function can also be used to add XDC constraint files to the Vivado IDE project. You should make sure that the order in which `add_file` is called, is hierarchical in nature. The top-module file must be added last.

### **run\_synthesis()**

This function runs synthesis in the Vivado IDE project.

### **run\_implementation()**

This function runs implementation in the Vivado IDE project.

### **generate\_bitstream()**

This function generates a bitstream in the Vivado IDE project.

## ***Design Info***

`design_info` is a MATLAB® struct and its contents are shown below:

Figure 119: Design Info

The screenshot shows two tables of design information. The top table is for a specific port named 'design\_info.ports.gateway\_in'. The bottom table is for the entire design object.

Field	Value	Min	Max
ArithmeticType	'xISigned'		
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

Field	Value	Min	Max
ports	<1x1 struct>		
sim_time	10	10	10
target_dir	'/group/dspuser...'		
testbench	'on'		
top_level	'adder'		

## ***Token-Related Properties and APIs***

### **`setup_sysgen_token()`**

This function is called to populate Model Composer Hub information by the Custom Compilation Infrastructure. You can use any of the following functions related to the Model Composer Hub 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 Model Composer Hub application programming interface (API) functions.

### **`add_part(family, device, speed, package, temperature)`**

An example of an explicit command is `add_part('Kintex7', 'xc7k325t', '-1', 'fbg676', '')`. If the part-related APIs are not used, you can select any device 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. Select 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. When a value is set in this field, the field will be disabled for further user selection. If this API is used, you need to ensure that the specified strategy exists. Otherwise, it will result in an error.

**string impl\_strategy**

The default value is an empty string. When a value is set in this field, the field will be disabled for further user selection. If this API is used, you need to ensure that the specified strategy exists. Otherwise, it will result in an error.

**string create\_tb**

The default value is an empty string. The valid options are on or off. When a value is set in this field, the field will be disabled for further user selection.

**string create\_iface\_doc**

The default value is an empty string. The valid options are on or off. When a value is set in this field, this field will be disabled for further user selection.

## 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. In the MATLAB® Command Window, enter the following command:

```
xilinx.environment.addCompilationTarget('Impl', 'demo')
```

This provides a template derived class in the `demo` folder for you to edit.

2. Open the newly created file `demo\Impl \@Impl\Impl.m` in the MATLAB Editor.
3. Populate the `setup_sysgen_token()` function as per the requirements. Using this approach, you can control how the Model Composer Hub block 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
```

4. Save `Impl.m`.
5. In the MATLAB Current Folder browser, right-click the **demo** folder and select **Add To Path → Selected Folder and Subfolders**.
6. In the MATLAB Command Window, enter the following command:

```
xilinx.environment.rehashCompilationTarget
```

This ensures that the updated class definition of `Impl` is used.

7. Open the Model Composer Hub block in your model.
8. On the Settings pane of your HDL subsystem, ensure Verilog is selected as the Target Language.
9. Under Implementation Strategy, select **Flow\_Quick**.
10. 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 in the following figure.

```
#####
% 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
```

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

```
xilinx.environment.rehashCompilationTarget
```

This ensures that the updated class definition of `Impl` is used.

12. Close and then re-open the Model Composer Hub block.
13. Select **Flow\_Quick** from the list of Implementation strategies.
14. On the **Export** tab, select `Impl` as the **Export Type**.
15. Click **Export**. When the process is finished, you can see the implementation results by opening the Vivado IDE project.

## **Example 2: Creating a Bitstream Target**

1. Open a Vitis Model Composer design.
2. In the MATLAB command Window, modify the path as per your requirements, similar to the first example, and then enter the following command:

```
xilinx.environment.addCompilationTarget('Bitstream', '.')
```

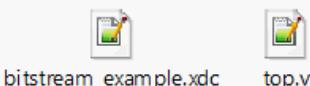
This provides a template derived class for you to edit. The last field corresponds to the directory which contains the `board.xml` file.

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

```
xilinx.environment.rehashCompilationTarget
```

This ensures that the new compilation target is picked up by the Vitis Model Composer Hub block.

4. Close and then re-open the Vitis Model Composer Hub block.
5. You will now see the export type Bitstream on the **Export** tab of the Vitis Model Composer Hub block.
6. Open the `Bitstream.m` created in the `'./Bitstream/@Bitstream/Bitstream.m'`.
7. Download the following two files:



8. Inside the function `pre_project_creation()`, add the following lines to do the following:
  - a. Set the board as a KC705 board.
  - b. Add a new top-level file (`top.v`) to use the differential clock ports of KC705.
  - c. Add a new XDC file to give the location constraints for the clock, dip, and led ports.
  - d. Set the newly added module `top` as the top.
  - e. Run synthesis.
  - f. Run implementation.

- g. Generate bitstream.

After you save the files to a location on your computer, you should give the full path to the files in the `add_file` API as per your path.

```
add_tcl_command(obj, 'set_property board xilinx.com:kintex7:kc705:1.1
[current_project]');
add_file(obj,
'/group/dspusers-xsj/umangp/rel/2013.4/cust_comp_test/
bitstream_example.xdc');
add_file(obj, '/group/dspusers-xsj/umangp/rel/2013.4/cust_comp_test/
top.v');
obj.top_level_module = 'top';
run_synthesis(obj);
run_implementation(obj);
generate_bitstream(obj);
```

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

```
xilinx.environment.rehashCompilationTarget
```

This ensures that the new compilation target is picked up by the Vitis Model Composer Hub block.

10. Close and then re-open the Vitis Model Composer Hub block.

11. Select the **Bitstream** export type.

12. Click the **Export** button.

13. After the generation is complete, you can find the bit file in the following directory:

```
./<target_directory>/ip/<hdl_subsystem>/src/Bitstream/
bitstream_example.runs/impl_1/top.bit
```

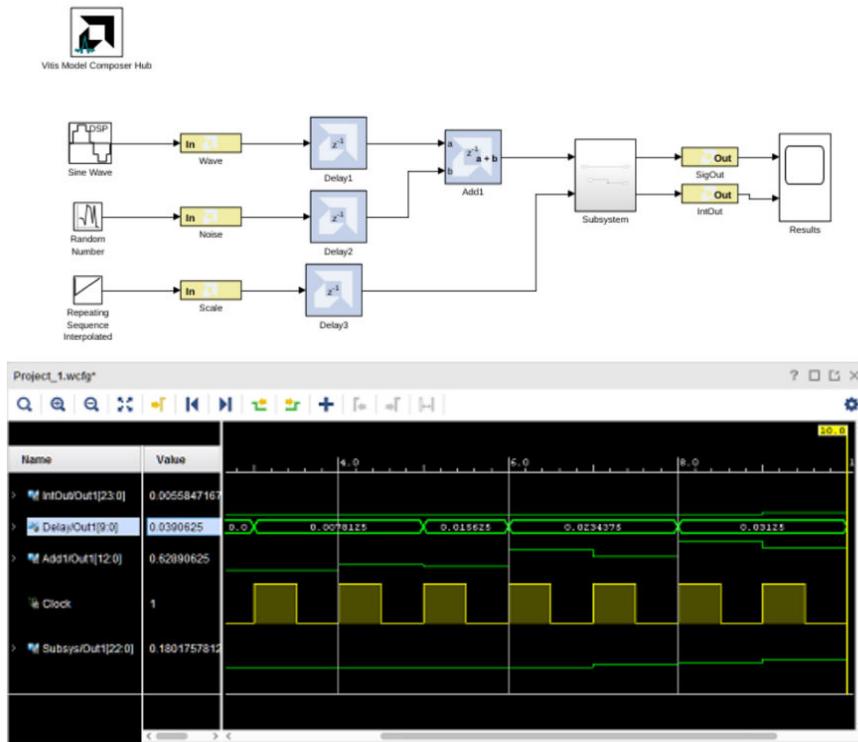
---

## AMD Waveform Viewer

The AMD Waveform Viewer displays a waveform diagram of selected signals in your Vitis Model Composer design. Waveforms can be displayed in the Waveform Viewer after running a Simulink simulation. Inputs and outputs of blocks in the HDL Blockset can be displayed in the Waveform Viewer.

In your design, you can select the signals that will be monitored in the Waveform Viewer. As you develop and troubleshoot your design, the waveforms for the signals you are monitoring will be updated in the Waveform Viewer each time you simulate the model.

Figure 120: Example Design in Waveform Viewer



The AMD Waveform Viewer used with Vitis Model Composer is also used by other tools in the AMD Vivado™ toolset. The Waveform Viewer is used to analyze a design and debug code in the Vivado simulator and to display data captured by the Integrated Logic Analyzer (ILA) for in-system debugging.

For information on using the Waveform Viewer to develop and troubleshoot your design, see this [link](#) in the *Vivado Design Suite User Guide: Logic Simulation (UG900)*.

## Waveform Viewer Files

The first time you open the Waveform Viewer for your Simulink model, Model Composer creates a `wavedata` directory in the directory containing your Simulink model.

**Note:** You will need write permission for the directory containing your Simulink model.

Data describing the display in the Waveform Viewer is stored in the following files in the `wavedata` directory:

- `<design_name>.wcfg` - This is the waveform configuration file. It contains the names of the signals you are monitoring in your design and how the waveforms for these signals will appear in the Waveform Viewer.

- <design\_name>.wdb - This is the waveform database file. It contains the data necessary to draw the waveforms in the Waveform Viewer.

The names of the signals that are being monitored are stored in the Simulink model (SLX file). If the Simulink model cannot access the data in the `wavedata` directory (for example, if you moved the model's SLX file to a different directory and opened it in the new directory), you can display the monitored signals by opening the Waveform Viewer and simulating the design. The waveforms for the monitored signals will then appear in the Waveform Viewer.

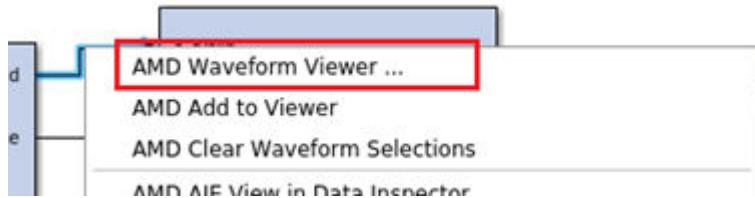
## Opening the AMD Waveform Viewer

You can open the Waveform Viewer in either of the following ways:

- Opening from right-click menu:

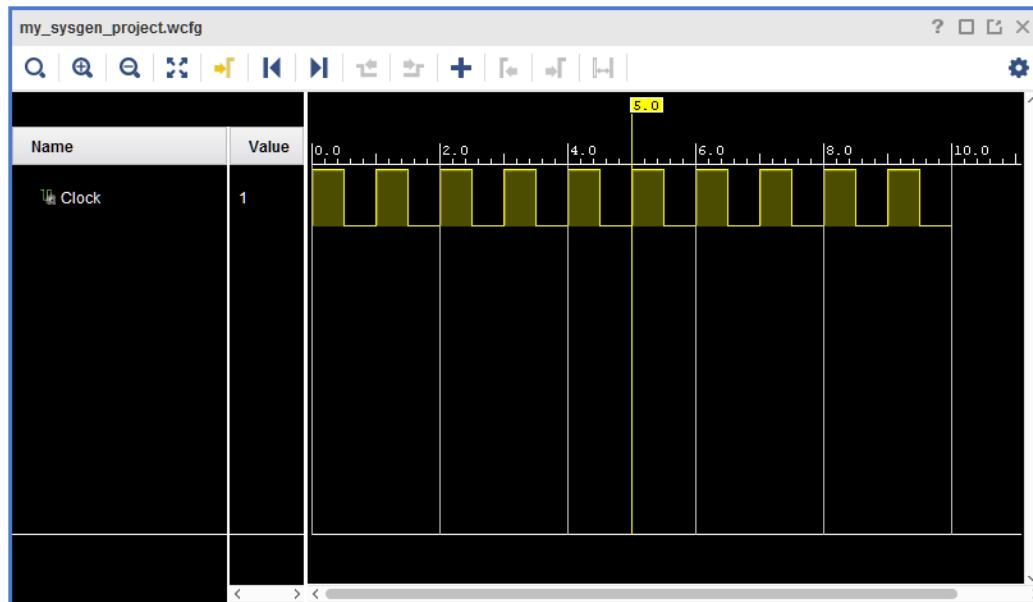
Right-click in your model and select **AMD Waveform Viewer....**

*Figure 121: AMD Waveform Viewer*

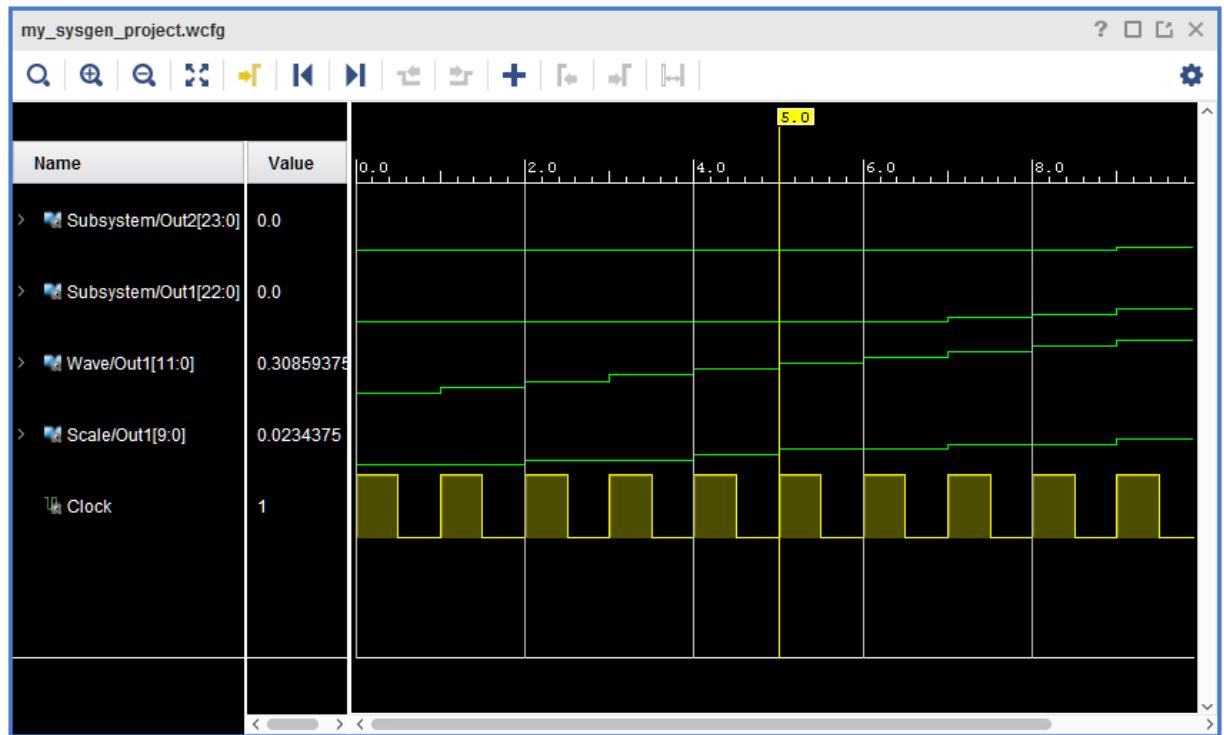


If you open it from the right-click menu, the Waveform Viewer opens with the following display:

- If this is the first time you are opening the Waveform Viewer for this design, the Waveform Viewer opens displaying waveforms for the clock signals in your design, and no other waveforms. You can then add the signals in your design that you want monitored to the Waveform Viewer display (see [Adding Signals to the Waveform Viewer Display](#)).

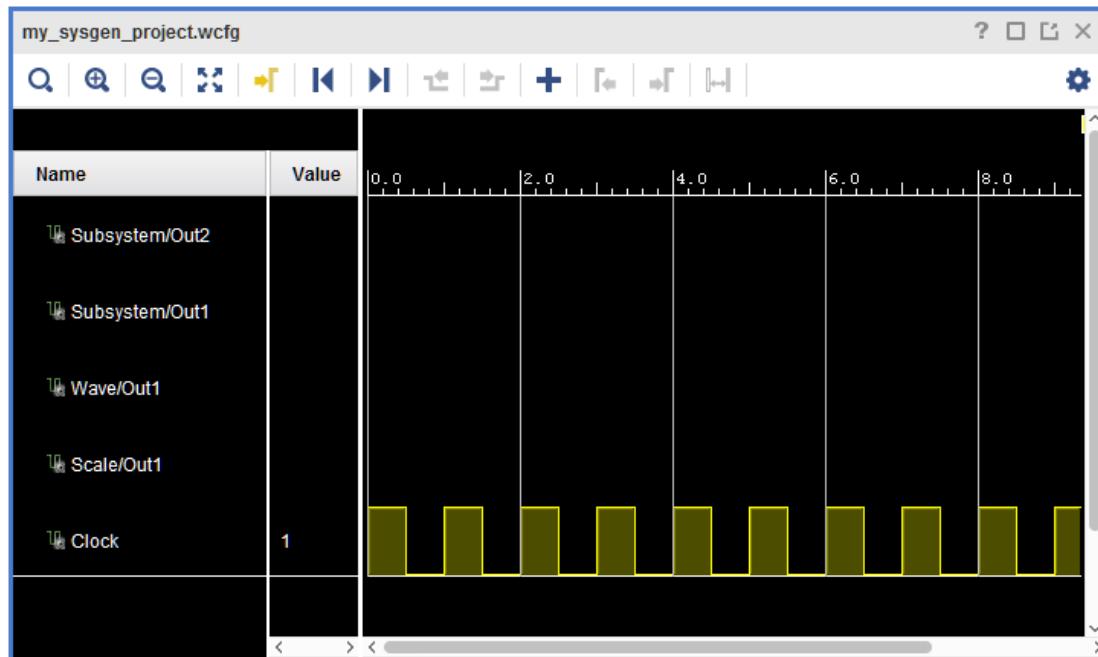
*Figure 122: Waveform Viewer Display*

- If you have previously monitored signals in the Waveform Viewer for this design, and have saved the data, the Waveform Viewer opens displaying the signal names, and waveforms displayed when you last closed the Waveform Viewer.

*Figure 123: Signals in Waveform Viewer*

- If you have previously monitored signals in the Waveform Viewer for this design, but cannot access the saved data (for example, if you moved the model's SLX file to a different directory, and opened it in the new directory), the Waveform Viewer opens displaying the signal names for the signals monitored when you last saved the model. The Waveform Viewer will not show the waveforms for the monitored signals until you re-simulate the model.

Figure 124: Waveform Viewer New Signals



- Opening after simulation:

If you have previously monitored signals in the Waveform Viewer for your design, the Waveform Viewer will open automatically when you simulate your model.

## Adding Signals to the Waveform Viewer Display

Inputs and outputs of blocks in the HDL Blockset can be displayed in the Waveform Viewer. The data necessary to draw each signal's waveform is not stored with the design; it is generated by simulation. You can only display a signal's waveform after you have added the signal to the Waveform Viewer and then simulated the model.

To add signals to the display in the Waveform Viewer:

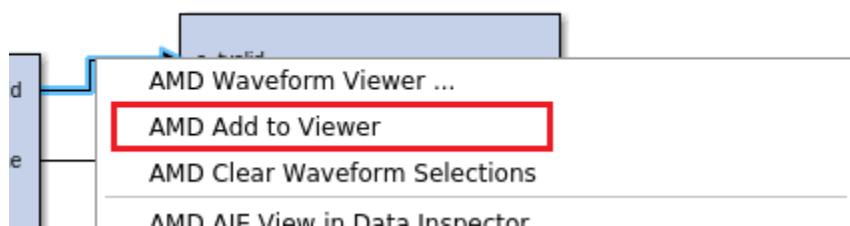
1. With the Waveform Viewer open, select a signal in the Vitis Model Composer model.

You can also select multiple signals by using Shift + click to select additional signals.

**Note:** For the Gateway In block, only the output signal can be displayed in the Waveform Viewer.

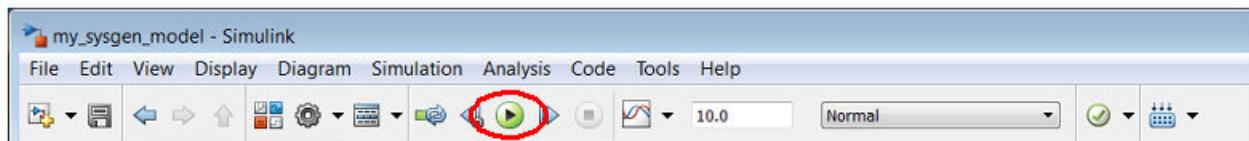
2. Right-click one of the selected signals in the Model Composer model and select **AMD Add to Viewer** in the right-click menu.

**Note:** If you select a signal that is currently displayed in the Waveform Viewer, the AMD Add to Viewer entry will not appear in the right-click menu.



The signal names of the selected signals appear in the Waveform Viewer.

3. Only the names of the added signals appear in the Waveform Viewer, because the Waveform Viewer does not have the data to draw the signal's waveform until you simulate the design.
4. Simulate the model.



After the simulation is finished, the waveforms for the added signals are displayed in the Waveform Viewer.

## Deleting Signals From the Waveform Viewer Display

1. In the Waveform Viewer, select the signals to be deleted.  
Use Shift+click or Ctrl+click to select multiple signal names (Ctrl+A to select all).
2. Right-click one of the selected names and select **Delete** in the right-click menu.

OR

Press the Delete key.

The waveforms are deleted from the Waveform Viewer. Deleted waveforms are no longer monitored; if you re-simulate the model the deleted waveforms will not appear in the Waveform Viewer.

## Cross Probing Between the Waveform Viewer and the Model

Cross probing helps you correlate the waveforms in the viewer to the wires in the Vitis Model Composer model.

You can cross probe signals between the Waveform Viewer, and the model in the following ways:

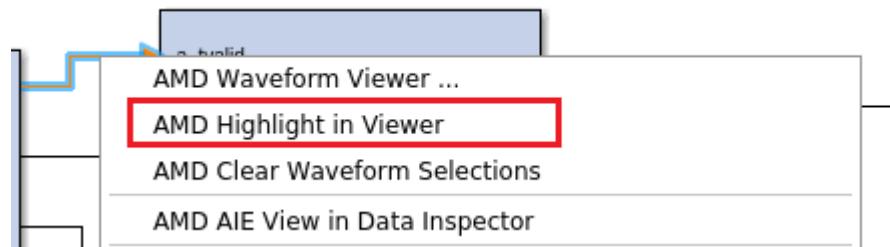
- To cross probe a signal from the Waveform Viewer to the Model Composer model, select one or more signal names in the Waveform Viewer. Use Shift+click or Ctrl+click to select multiple signal names (press **Ctrl+A** to select all).

The selected signals are highlighted in orange in the Model Composer model.

To unhighlight a signal you have highlighted in the Model Composer model, Ctrl+click the signal name in the Waveform Viewer. The signal is unhighlighted in the Model Composer model.

- To cross probe a signal from the Model Composer model to the Waveform Viewer:
  1. With the Waveform Viewer open, select a signal in the Model Composer model.  
You can also select multiple signals by using Shift+click to select additional signals.
  2. Right-click one of the selected signals in the Model Composer model and select **AMD Highlight in Viewer** in the right-click menu.

**Note:** If you select a signal that is not currently displayed in the Waveform Viewer, the AMD Highlight in Viewer entry will not appear in the right-click menu.



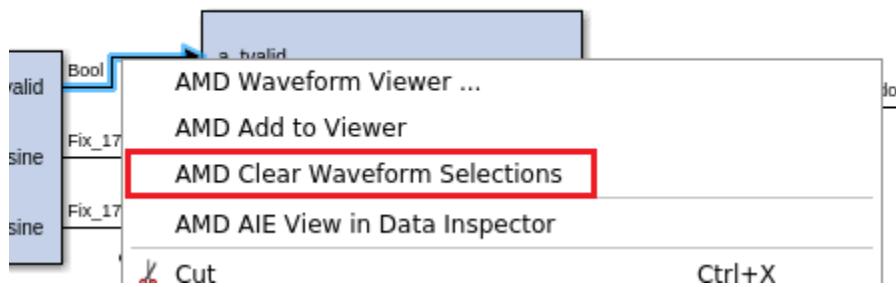
3. Observe that the signal names of the selected signals are highlighted in the Waveform Viewer.

## Clearing the Waveform Viewer Display

To clear the waveform display, deleting all the waveforms currently displayed in the Waveform Viewer:

1. Right-click in the Vitis Model Composer model.
2. Select **AMD Clear Waveform Selections** in the right-click menu.

All of the signals currently displayed in the Waveform Viewer are deleted from the Waveform Viewer display, and the Waveform Viewer closes. The deleted waveforms are no longer monitored and the waveform data are deleted from the `wavedata` directory.



To open the Waveform Viewer again, right-click in your model and select **AMD Waveform Viewer** in the right-click menu. The Waveform Viewer opens displaying waveforms for the clock signals in your design, and no other waveforms.

## Customizing the Display and Analyzing Waveforms

The Waveform Viewer has many tools to customize how your waveforms are displayed and to analyze the waveforms. For information on using the Waveform Viewer to develop and troubleshoot your design, see this [link](#) in the *Vivado Design Suite User Guide: Logic Simulation (UG900)*.

## Tips for Working in the Waveform Viewer

The following tips will help you with your waveform analysis using the Vitis Model Composer model and the Waveform Viewer:

- Keep the Waveform Viewer open during a Model Composer session. Do not close the Waveform Viewer between each simulation.
- If you select a group of signals in the Waveform Viewer, all of the signals in the group will be cross-probed from the Waveform Viewer to the Model Composer model.
- To add multiple signals in your Model Composer model to the Waveform Viewer display, you can press and hold the left mouse button and drag the mouse to draw a box around the signals, selecting them. Then right-click one of the selected signals and select **AMD Add to Viewer** in the right-click menu. The selected signals will be added to the Waveform Viewer display.
- When naming an output signal for a block in your Model Composer model, avoid using the reserved characters shown in the table below. These are reserved characters in VHDL or Verilog. If your model does contain a signal with a reserved character, its name will be changed in the Waveform Viewer display according to the following mapping table.

Table 14: Reserved Characters

Reserved Character	Mapped To
(	#1

**Table 14: Reserved Characters (cont'd)**

Reserved Character	Mapped To
)	#2
[	#3
]	#4
.	#5
,	#6
:	#7
\	#8

## Closing the Waveform Viewer

To close the Waveform Viewer, select **File**→**Exit**. If you have not yet saved the waveform data, you will be prompted to save the data before the Waveform Viewer closes.

# HLS Library

## Introduction

AMD Vitis™ Model Composer provides the HLS blockset in the AMD Toolbox. This enables you to transform your algorithmic specifications to production-quality IP implementations using automatic optimizations and leveraging the high-level synthesis technology of Vitis HLS. Using the IP integrator in Vivado, you can then integrate the IP into a platform that, for example, might include an AMD Zynq™ device, DDR3 DRAM, and a software stack running on an Arm® processor.

The HLS library in the AMD Toolbox provides optimized blocks for use within the Simulink environment. These include basic functional blocks for expressing algorithms like Math, Linear Algebra, Logic, and Bit-wise operations and others.

The HLS library contains the following categories of elements.

**Table 15: HLS Block Library**

Library	Description
Logic and Bit Operations	Blocks that perform logical operations and bit-wise operations.
Lookup Tables	Block that performs a one dimensional lookup operation with an input index.
Math Functions	Blocks that implement mathematical functions.
Ports and Subsystems	Blocks that allow creation of subsystems and input/output ports.
Relational Operations	Blocks for comparing two signals.
Signal Attributes	Blocks that perform data type conversion.
Signal Operations	Blocks that delay signals in time.
Signal Routing	Blocks that combine, decombine, and route signals.
Sinks	Blocks that receive and display signal output from other blocks.
Source	Blocks that generate or import signal data.
Tools	Blocks that control the implementation and interface of the Model.
User-Defined Functions	Block for importing an HLS Kernel as a block.

The HLS block library is compatible with the standard Simulink block library, and these blocks can be used together to create models that can be simulated in Simulink. However, only certain Simulink blocks are supported for code generation by Model Composer. The Simulink blocks compatible with output generation from Model Composer can be found in the HLS block library.

Model Composer also lets you create your own custom blocks from existing C/C++ code for use in models. Refer to [Importing C/C++ Code as Custom Blocks](#) for more information.

Your Model Composer design is bit-accurate with regard to the final implementation in hardware, though untimed. You can compile the design model into C++ code for synthesis in Vitis HLS, create HDL blocks, or create packaged IP to be used in Vivado.

To familiarize yourself with the Model Composer HLS library, the [Vitis Model Composer Tutorials](#) include labs and data to walk you through the tool.

The rest of this document discusses the following topics:

- [Creating a Model Composer model using HLS block libraries.](#)
- [Importing existing C-code into the Model Composer HLS block library for use in your models.](#)
- [Compiling the model for use in downstream design tools.](#)
- [Verifying your Model Composer model, C++, and RTL outputs.](#)

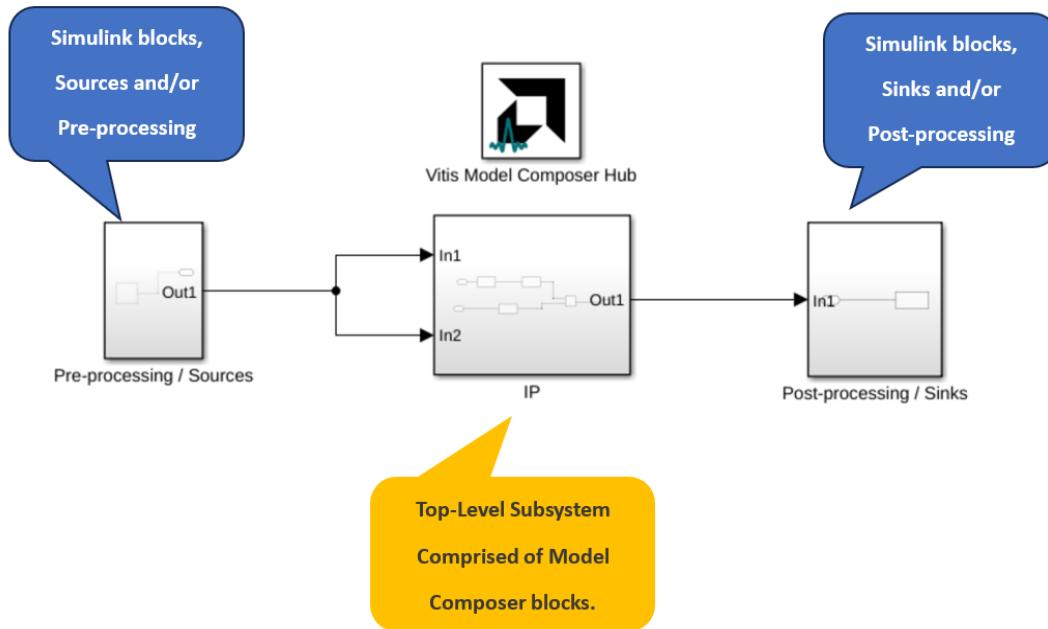
---

## Creating a Model Composer Design

As shown in the image below, a Vitis Model Composer design with HLS blocks includes the following elements:

1. Simulink® blocks that determine inputs, and provide source signals. These blocks are used in simulation of the design, but do not affect the output generated by Model Composer.
2. A top-level subsystem block, as described in [Creating a Top-Level Subsystem Module](#), that encapsulates the algorithm defined by the Model Composer model. This subsystem module can contain:
  - Blocks from the HLS library to define your algorithm, as listed in the HLS Library.
  - Custom imported functions as described in [Importing C/C++ Code as Custom Blocks](#).
  - An Interface Spec block that defines the hardware interfaces as described in [Defining the Interface Specification](#).
3. The Model Composer Hub block that controls throughput of the design, and output generation through a series of options as described in [Vitis Model Composer Hub](#).
4. The output signals, or sinks that process the output in Simulink. Again, these blocks are used during Simulink simulation as described in [Simulating and Verifying Your Design](#), but do not affect the output generated by Model Composer.

Figure 125: Elements of a Model Composer Design



## Creating a New Model

You create a new model by adding blocks from the Library Browser into the Simulink Editor. You then connect these blocks with signal lines to establish relationships between blocks. The Simulink Editor manages the connections with smart guides and smart signal routing to control the appearance of your model as you build it. You can add hierarchy to the model by encapsulating a group of blocks and signals as a subsystem within a single block. Model Composer provides a set of predefined blocks that you can combine to create a detailed model of your application.

In the Simulink start page, select **Blank Model** to open a new model.



**TIP:** You can also open an existing Vitis Model Composer template if any are defined. Model templates are starting points to reuse settings and block configurations. To learn more about templates, see [Create a Template from a Model](#) in the Simulink documentation.

The Simulink start page also lists the recent models that you have opened on the left-hand column. You can open one of these recent models if you prefer.

The blank model opens, and you will create the Model Composer model by adding blocks, specifying block parameters, and using signal lines to connect the blocks to each other.



**IMPORTANT!** HLS Library only supports one sample time for the model, and does not support multi-time systems. All HLS Library blocks inherit the sample time from the source block of the model. See [What is Sample Time](#) for more information.

To save the model select **File → Save** from the main menu. The Save As dialog box is opened, with a file browser. Navigate to the appropriate folder or location, and enter a name for the model in the File Name field. Click **Save**. The model is saved with the file extension `.slx`.

Model Composer also includes example models based on HLS library which can be accessed from the *Examples and Tutorials* section of the *Vitis Model Composer* documentation available from the Help menu in the tool, or by typing the `xmcOpenExamples` command from the MATLAB command prompt:

```
>> xmcOpenExamples
```

## Adding Blocks to a Model

You can add blocks to the current model by opening the Simulink Library Browser and dragging and dropping the block onto the design canvas of the Simulink editor. Open the Library Browser



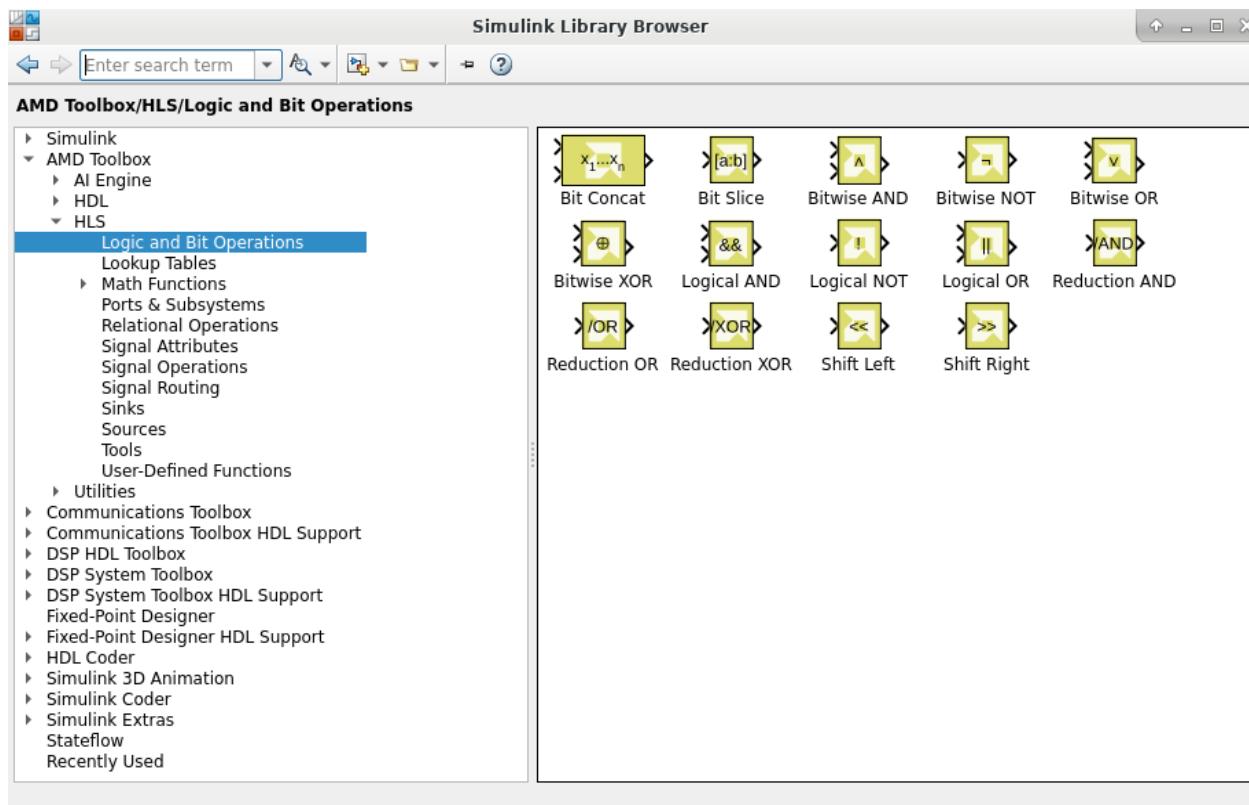
by clicking the button on the Simulation tab. You will see the standard Simulink library of blocks, as well as the HLS library in the AMD Toolbox.



**TIP:** You can also open the Library browser by typing the `sllibraryBrowser` command from the command prompt.

The HLS blocks are organized into sub-categories based on functionality. The following figure shows the HLS > Logic and Bit Operations block library in the Library Browser.

Figure 126: Library Browser



Double-clicking a block in the Library Browser opens the Block Parameter dialog box displaying the default values for the various parameters defined on the selected block. While the block is in the library, you can only view the parameters. To edit the parameters, you must add the block to the design canvas.

To get additional information about a block you can right-click a block in the Library Browser and select the **Help** command. Alternatively, you can double-click the block in the Library Browser and click the **Help** button from the block dialog box. The Help browser opens with specific information for the block.

When you drag and drop the block onto the canvas, the block is added to the model with the default parameter values defined.




---

**TIP:** You can also quickly add blocks to the current model by single-clicking on the design canvas of the Simulink Editor and typing the name of a block. Simulink displays possible matches from the libraries, and you can select and add the block of interest.

---

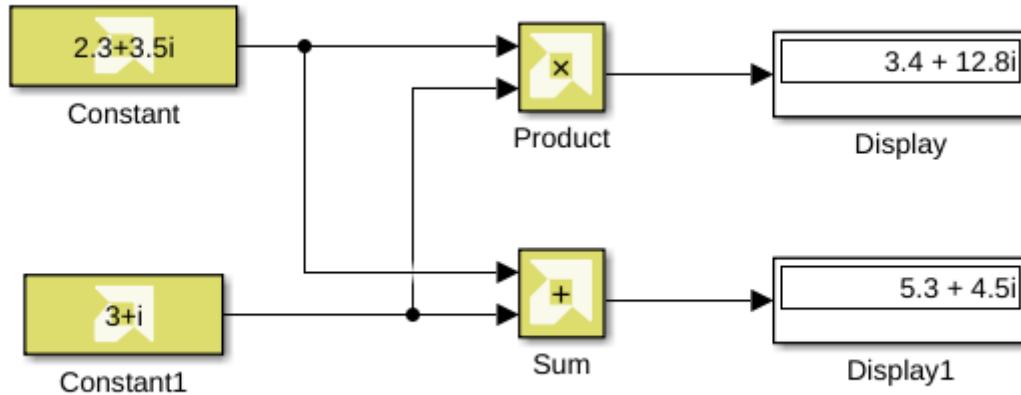
Simulink models contain both signals and parameters. Signals are represented by the lines connecting blocks. Parameters are coefficients that define key characteristics and behavior of a block.

## Connecting Blocks

You can connect the output ports on blocks to the input ports of other blocks with signal lines. Signal lines define the flow of data through the model. Signals can have several attributes:

- **Data type:** Defines the type of data carried by the signal. Values can range from integer, to floating point, to fixed point data types. See [Working with Data Types](#) for more information.
- **Signal dimension:** Defines the values as being scalar, vector, or matrices. See [Signal Dimensions](#) and [Matrices, Vectors, and Scalars](#) for more information.
- **Complexity:** Defines a value as being a complex or real number. See [Signal Values](#) for more information. The following figure shows complex numbers propagating through a model.

Figure 127: Complex Signal Values



To add a signal line, position the cursor over an input or output port of a Simulink block. The cursor changes to a cross hair (+). Left-click and drag the mouse away from the port. While holding down the mouse button, the connecting line appears as a dotted line as you move across the design canvas. The dotted line represents a signal that is not completely connected.

Release the mouse button when the cursor is over a second port to be connected. If you start with an input port, you can stop at an output port, or connect to another signal line; if you start at an output you can stop at an input. Simulink connects the ports with a signal line and an arrow indicating the direction of signal flow.

You can connect into an existing line by right-clicking and dragging the mouse. This creates a branching line connected to the existing signal line at the specified location. The branch line can connect to an input or output as appropriate to the connected signal.



**TIP:** You can also connect blocks by selecting them sequentially while holding the Ctrl key. This connects the output of the first block into the input of the second block. Keeping the Ctrl key pressed and selecting another block continues the connection chain.

Simulink updates the model when you run simulation by clicking **Run Simulation**. You can also update the model using the **Simulation** → **Update Diagram** menu command, or by typing **Ctrl+D** at any point in the design process. You will see the Data Types, Signal Dimensions and Sample Times from the source blocks propagate through the model.



**TIP:** You can use the **Display** → **Signals and Ports** menu command to enable the various data that you want displayed in your model, such as Signal Dimensions and Port Data Types.

You cannot specify sample times on HLS Library blocks, except for the Constant block from the Source library of the HLS library. Model Composer infers the sample time from the source blocks connected at the input of the model, and does not support multiple sample times in the Model Composer design.

By updating the diagram from time to time, you can see and fix potential design issues as you develop the model. This approach can make it easier to identify the sources of problems by limiting the scope to recent updates to the design. The Update Diagram is also faster than running simulation.

## Working with Data Types

Data types supported by HLS library blocks include the following:

**Table 16: Model Composer Data Types**

Name	Description
double	Double-precision floating point
single	Single-precision floating point
half*	Half-precision floating point
int8	Signed 8-bit integer
uint8	Unsigned 8-bit integer
int16	Signed 16-bit integer
uint16	Unsigned 16-bit integer
int32	Signed 32-bit integer
uint32	Unsigned 32-bit integer
fixed*	Signed and unsigned fixed point
boolean	For this data type, Simulink represents real, nonzero numeric values as TRUE (1)



**IMPORTANT!** Data types marked with '\*' are specific to Model Composer HLS Library, and are not naturally supported by Simulink. While Simulink does support fixed point data types, you must have the Fixed-Point Designer™ product installed and licensed. In addition, the fixed point data type supported by Vitis Model Composer is not compatible with the fixed point data type supported by Simulink although it uses a similar notation.

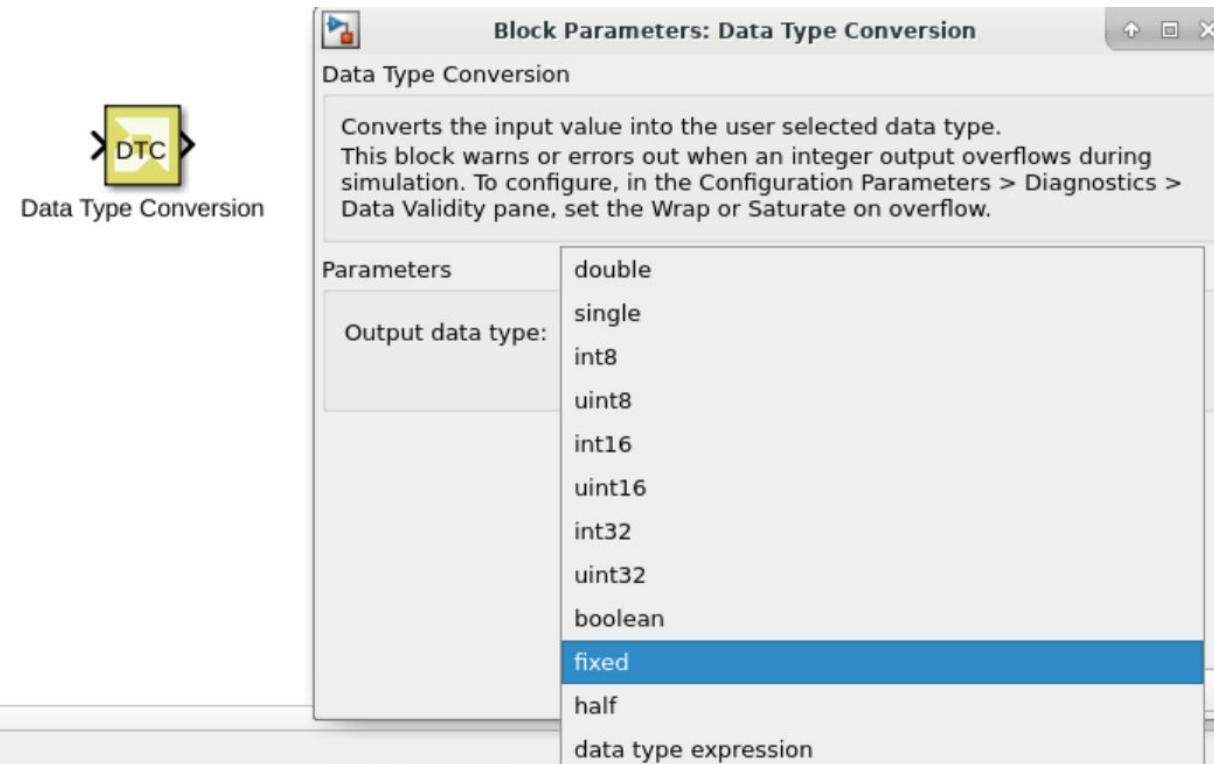
Notice in the preceding table there are some data types that are supported by Model Composer HLS Library that are not supported by default in Simulink. If you connect blocks from the HLS library, with `fixed` or `half` data types, to Simulink native blocks, you will see an error when running simulation in Simulink, or when using the Update Diagram command, or pressing `Ctrl+D`.

```
RelationalOperator does not accept signals of data type 'x_sfix16'.  
'ConstRE_or_IMpartBug/Relational Operator' only accepts numeric and  
enumerated data types.
```

This error indicates that Simulink could not cast the signal value from the Model Composer fixed data type to a double precision floating point data type.

In cases of mismatched data types, Model Composer recommends that you use a Data Type Conversion block to specify the behavior of the model, and indicate the conversion of one data type to another. The Data Type Conversion block (DTC) is found in the HLS Library under the Signal Attributes library.

Figure 128: Data Type Conversion Block



The DTC block lets you specify the Output data type, while the input data type is automatically determined by the signal connected to the input port. Using the DTC block, you can convert from single precision floating point to double precision for example, or from double precision to single precision.

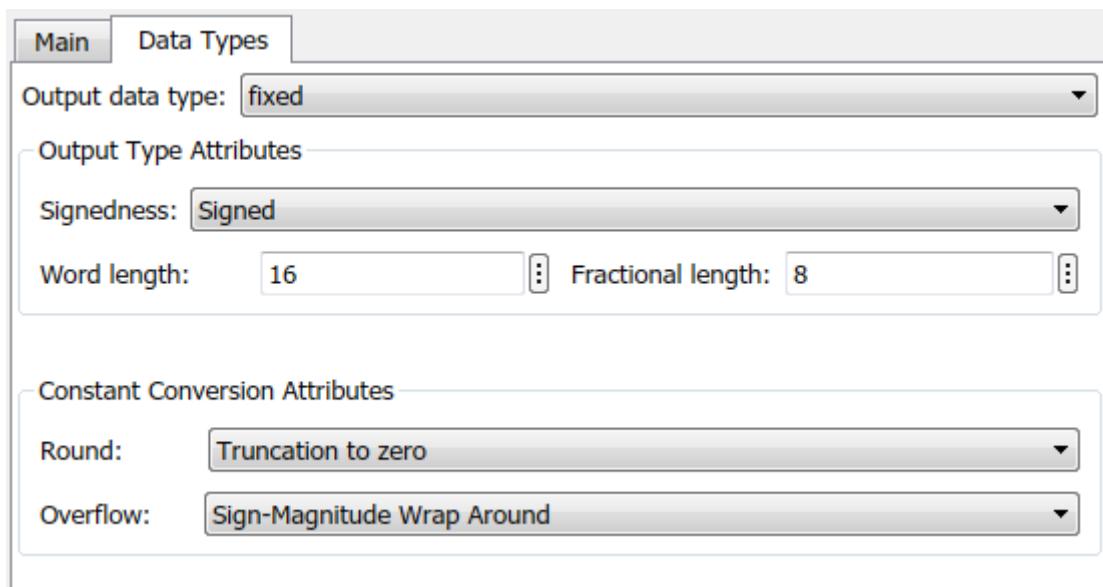


**IMPORTANT!** You must exercise caution when casting from a higher precision data type to a lower precision data type, as loss of precision can lead to rounding or truncation and loss of data can occur.

## Working with Fixed-Point Data Types

As indicated earlier, Simulink® provides support for fixed-point data types through the Fixed-Point Designer™ product. However, the format of the `fixed` data type supported by Vitis Model Composer and Simulink are not compatible.

Figure 129: Fixed-Point Data Type



The format used to display the Model Composer fixed-point data types is as follows: `x_- [ u / s ] fix[wl]_E[n][f]`

Where:

- `x_-`: Is the prefix indicating the AMD fixed data type.
- `[ u / s ]`: Represents signed or unsigned data.
- `fix`: Indicates the fixed-point data type.
- `[wl]`: Specifies the word length of the data.
- `E`: Prefix for the fractional portion of the fixed-point data type. Does not display if the fractional length is 0.

- n: Displays 'n' if the binary point is to the left of the right-most bit in the word; or displays no 'n' if the binary point is to the right of the right-most bit in the word.
- [f1]: Specifies the fractional length of the fixed-point data type, indicating the position of the binary point with respect to the right-most bit in the word.

For example, `x_sfix16_En6` represents a signed 16-bit fixed-point number, with 6-bits allocated to the right of the binary point.

Notice the fixed-point data type also lets you specify what happens in the case of data overflow, or the need to do rounding or truncation.

You must use a DTC block to convert the Model Composer fixed-point data type into the Simulink fixed-point data type. However, there is no direct conversion between Model Composer fixed-point and Simulink fixed-point data types, so you can use the following method:

1. Convert Model Composer fixed-point data type to the `double` data type using the DTC block from the HLS library in the Library Browser.
2. Convert the `double` data type to Simulink format fixed-point data type using the Simulink Data Type Conversion block from the Simulink Signal Attributes library in the Library Browser.
3. Match the signedness, word length, and fractional length between the two fixed-point data types.



---

**TIP:** Converting between the Model Composer fixed data type and the Simulink fixed data type is not recommended unless necessary for your design. You can convert from the Model Composer fixed-point data type to double, as shown in the first step, and this should be sufficient for most applications.

---

Although handling fixed-point data type in a design is more time consuming, the value of using fixed-point data types for applications targeted at implementation in an FPGA is worth the challenge. It is widely accepted that designing in floating point leads to higher power usage for the design. This is true for FPGAs where floating-point DSP blocks are hardened onto the FPGA and users must implement a floating point solution using DSP blocks and additional device resources. Floating-point implementations require more FPGA resources than an equivalent fixed-point solution. With this higher resource usage comes higher power consumption and ultimately increased overall cost of implementing the design. For more information refer to the white paper *Reduce Power and Cost by Converting from Floating Point to Fixed Point* ([WP491](#)).

## Working with Half Data Types

Vitis Model Composer also supports a `half` precision floating point data type, which is 16 bits wide instead of 32 bits, because it consumes less real estate on the target device when implemented on the FPGA. This is an important consideration when designing in Model Composer. However, Simulink does not support the `half` data type, and this can lead to errors in simulation. The solution is to use the Model Composer DTC block to convert the `half` into the `single` data type supported by Simulink, for those portions of the design that are not in the Model Composer sub-module and will not be part of the generated output.

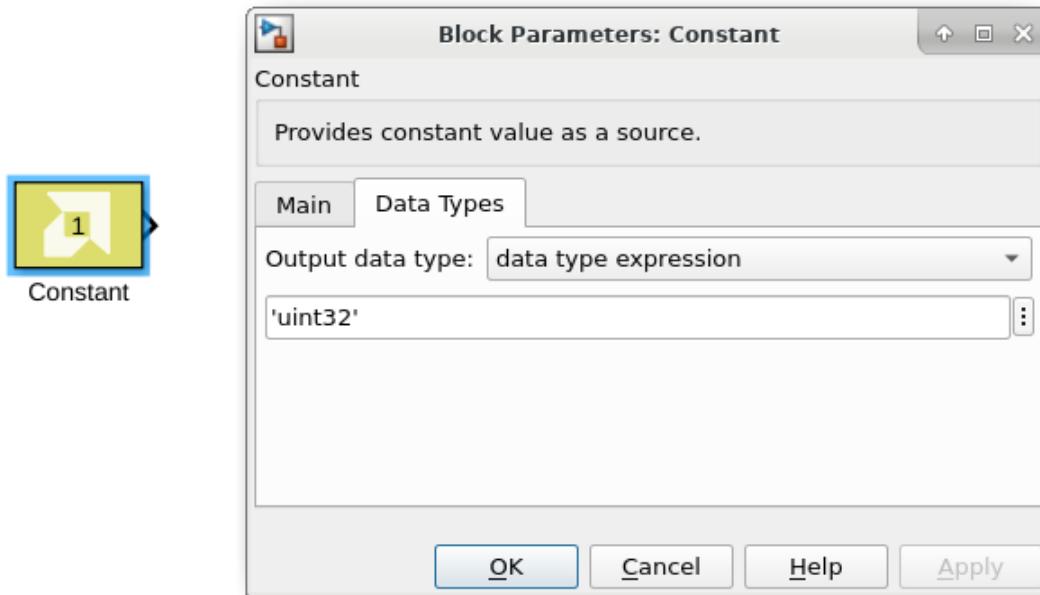
## Working with Data Type Expression

Vitis Model Composer lets you specify data types as an expression. Currently the following HLS library blocks support data type expression:

- Constant
- Data Type Conversion
- Gain
- Look-Up Table
- Reinterpret

To specify a data type expression open one of the block types that support it, and edit the data type and value. The following shows a data type expression being defined for the Constant block. The Output data type is specified as an expression, and a string is specified to indicate the data type value, in this case '`uint32`'.

Figure 130: Data Type Expression



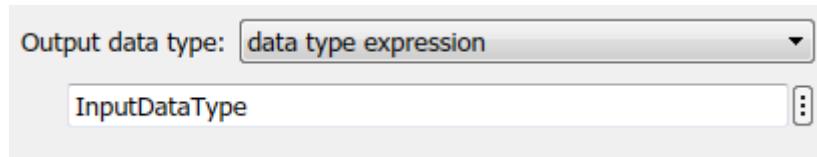
The data type value can be specified as a string representing any of the supported data types shown in the Model Composer Data Types table in the [Working with Data Types](#) section. The exception to this rule is for fixed point data types, which are not specified with the `fixed` string, but are defined according to the display format discussed under [Working with Fixed-Point Data Types](#) (for example, '`x_sfix16_En8`').

The real benefit of defining a data type as an expression is the ability to programmatically determine the data type value using a variable from the model. For instance, if you define a variable from the MATLAB® command line:

```
>> InputDataType = 'x_ufix8_En7';
```

You can use the variable in defining the data type expression.

Figure 131: Variable Data Type



You can specify variables from the MATLAB command line, or define variables within the model by selecting **Modeling** → **Design** → **Model Explorer** menu command, or simply pressing **Ctrl+H**. From the Model Explorer you can create, edit, and manage variables that the model or a block uses.



**TIP:** You can also enable the *Modeling > Design > Property Inspector* command to display the variables for currently selected objects.

## Managing Overflow

Sometimes the data type definition for a block will not support the incoming data value on a signal. In these cases, an overflow condition can occur if the value on the signal is too large or too small to be represented by the data type of the block. The two types of overflow that can occur are wrap overflow, and saturation overflow:

- **Wrap on Overflow:** This is the default overflow mechanism, and causes the bit value to wrap at the overflow point. For instance, when an arithmetic operation such as multiplying two numbers produces a result larger than the maximum value for the data type, effectively causing a wrap around.
- **Saturate on Overflow:** This is an overflow mechanism in which all operations such as addition and multiplication are limited to a fixed range between the minimum and maximum values supported by the data type. In essence, the value will reach the maximum or minimum value and stop.

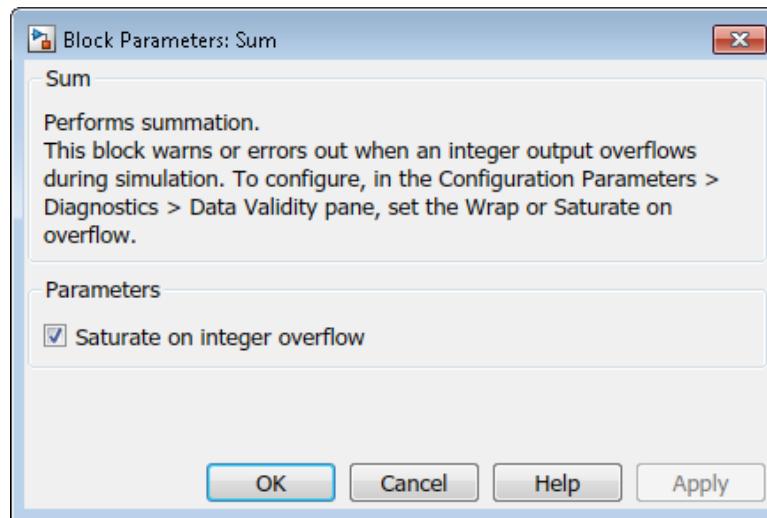
Wrapping is the default method for handling overflow, as it occurs naturally as the value overruns the data type. There is no checking required. However, the Saturate on Overflow option requires some additional logic to check the data value against the permitted maximum or minimum value to prevent wrapping. This additional logic consumes available resources on the target device.

### Saturate on Overflow

#### Saturate on Integer Overflow

Vitis Model Composer currently supports overflow detection of integer data values on signals. As previously indicated, the default overflow mechanism is to wrap on overflow. Specific blocks in the standard Simulink library of blocks and in the HLS library have an option to Saturate on integer overflow. This can be enabled on the Block Parameters dialog box. This parameter applies only if the output is an integer (int8, int16, int32, uint8, uint16, uint32). Refer to the Model Composer Block Library for information specific to a block.

Figure 132: Saturate on Integer Overflow

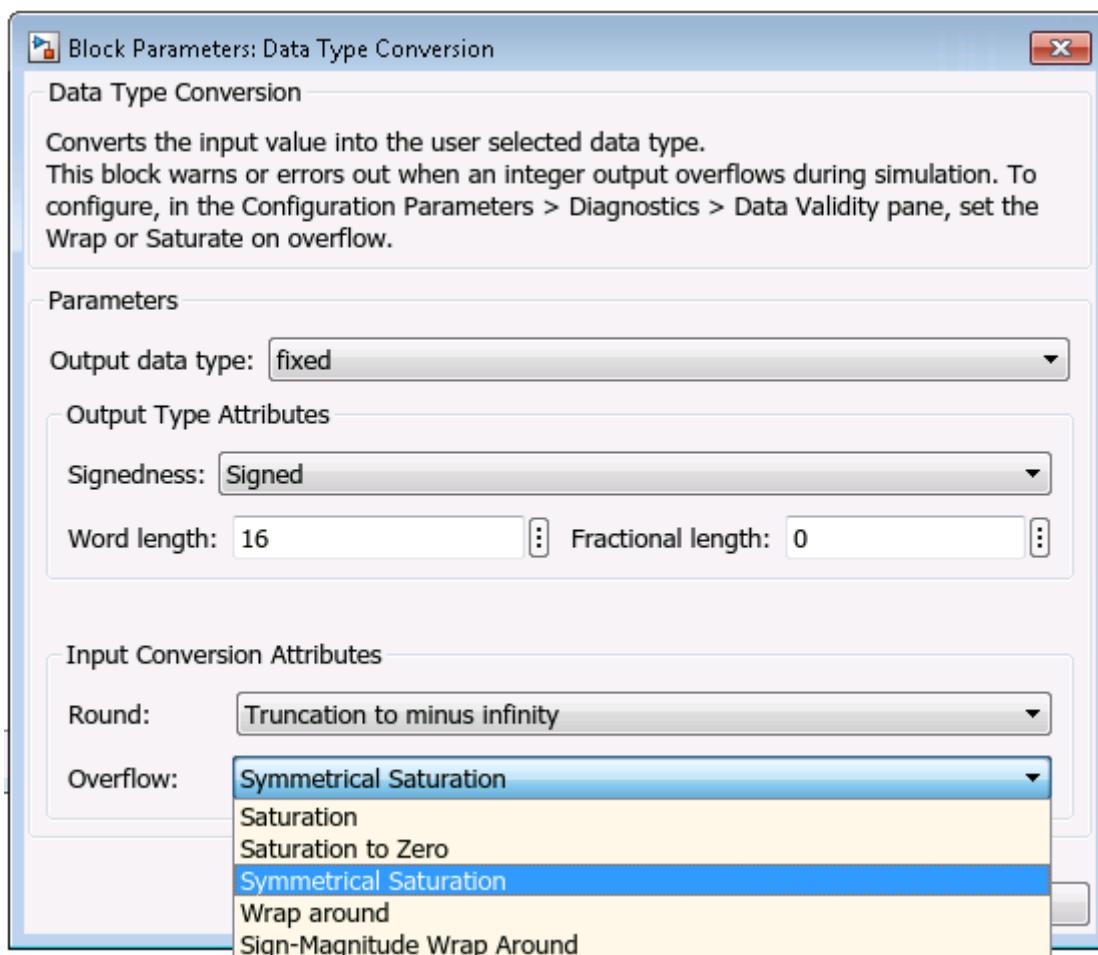


Saturate on integer overflow simply means that when the input value exceeds the range of values supported by the output, either too great or too small, the value simply sits at the max or min supported value. The value is saturated, and does not change.

### Saturate on Fixed-Point Overflow

For fixed-point data types, as supported on the Data Conversion Block (DTC) for example, the overflow modes offer more control than the Saturate on integer overflow option, as shown in the following figure.

Figure 133: Fixed-Point Overflow



A description of the different fixed-point overflow modes is provided below, with a graph to illustrate the condition.

**Table 17: Fixed-Point Overflow Modes**

Mode	Description	Image																																																
Saturation	When the input value overflows the output data type, the output value reaches saturation at the min or max value, and does not change.	<p style="text-align: center;"><b>Saturation</b></p> <table border="1"> <caption>Data for Saturation Mode</caption> <thead> <tr> <th>x</th> <th>y (Input)</th> <th>y (Output)</th> </tr> </thead> <tbody> <tr><td>-7</td><td>-5</td><td>-5</td></tr> <tr><td>-6</td><td>-4</td><td>-4</td></tr> <tr><td>-5</td><td>-3</td><td>-3</td></tr> <tr><td>-4</td><td>-2</td><td>-2</td></tr> <tr><td>-3</td><td>-1</td><td>-1</td></tr> <tr><td>-2</td><td>0</td><td>0</td></tr> <tr><td>-1</td><td>1</td><td>1</td></tr> <tr><td>0</td><td>2</td><td>2</td></tr> <tr><td>1</td><td>3</td><td>3</td></tr> <tr><td>2</td><td>3</td><td>3</td></tr> <tr><td>3</td><td>3</td><td>3</td></tr> <tr><td>4</td><td>3</td><td>3</td></tr> <tr><td>5</td><td>3</td><td>3</td></tr> <tr><td>6</td><td>3</td><td>3</td></tr> <tr><td>7</td><td>3</td><td>3</td></tr> </tbody> </table>	x	y (Input)	y (Output)	-7	-5	-5	-6	-4	-4	-5	-3	-3	-4	-2	-2	-3	-1	-1	-2	0	0	-1	1	1	0	2	2	1	3	3	2	3	3	3	3	3	4	3	3	5	3	3	6	3	3	7	3	3
x	y (Input)	y (Output)																																																
-7	-5	-5																																																
-6	-4	-4																																																
-5	-3	-3																																																
-4	-2	-2																																																
-3	-1	-1																																																
-2	0	0																																																
-1	1	1																																																
0	2	2																																																
1	3	3																																																
2	3	3																																																
3	3	3																																																
4	3	3																																																
5	3	3																																																
6	3	3																																																
7	3	3																																																
Saturation to Zero	When the input value overflows the output data type, the output value reaches saturation at the min or max value, and returns to zero.	<p style="text-align: center;"><b>Saturation to zero</b></p> <table border="1"> <caption>Data for Saturation to Zero Mode</caption> <thead> <tr> <th>x</th> <th>y (Input)</th> <th>y (Output)</th> </tr> </thead> <tbody> <tr><td>-7</td><td>-5</td><td>-5</td></tr> <tr><td>-6</td><td>-4</td><td>-4</td></tr> <tr><td>-5</td><td>-3</td><td>-3</td></tr> <tr><td>-4</td><td>-2</td><td>-2</td></tr> <tr><td>-3</td><td>-1</td><td>-1</td></tr> <tr><td>-2</td><td>0</td><td>0</td></tr> <tr><td>-1</td><td>1</td><td>1</td></tr> <tr><td>0</td><td>2</td><td>2</td></tr> <tr><td>1</td><td>3</td><td>3</td></tr> <tr><td>2</td><td>3</td><td>3</td></tr> <tr><td>3</td><td>3</td><td>3</td></tr> <tr><td>4</td><td>3</td><td>3</td></tr> <tr><td>5</td><td>3</td><td>3</td></tr> <tr><td>6</td><td>3</td><td>3</td></tr> <tr><td>7</td><td>3</td><td>3</td></tr> </tbody> </table>	x	y (Input)	y (Output)	-7	-5	-5	-6	-4	-4	-5	-3	-3	-4	-2	-2	-3	-1	-1	-2	0	0	-1	1	1	0	2	2	1	3	3	2	3	3	3	3	3	4	3	3	5	3	3	6	3	3	7	3	3
x	y (Input)	y (Output)																																																
-7	-5	-5																																																
-6	-4	-4																																																
-5	-3	-3																																																
-4	-2	-2																																																
-3	-1	-1																																																
-2	0	0																																																
-1	1	1																																																
0	2	2																																																
1	3	3																																																
2	3	3																																																
3	3	3																																																
4	3	3																																																
5	3	3																																																
6	3	3																																																
7	3	3																																																
Symmetrical Saturation	Like Saturation to Zero, except the min and max values are symmetrical, or equal in size though opposite in value.	<p style="text-align: center;"><b>Symmetrical saturation</b></p> <table border="1"> <caption>Data for Symmetrical Saturation Mode</caption> <thead> <tr> <th>x</th> <th>y (Input)</th> <th>y (Output)</th> </tr> </thead> <tbody> <tr><td>-7</td><td>-5</td><td>-5</td></tr> <tr><td>-6</td><td>-4</td><td>-4</td></tr> <tr><td>-5</td><td>-3</td><td>-3</td></tr> <tr><td>-4</td><td>-2</td><td>-2</td></tr> <tr><td>-3</td><td>-1</td><td>-1</td></tr> <tr><td>-2</td><td>0</td><td>0</td></tr> <tr><td>-1</td><td>1</td><td>1</td></tr> <tr><td>0</td><td>2</td><td>2</td></tr> <tr><td>1</td><td>3</td><td>3</td></tr> <tr><td>2</td><td>3</td><td>3</td></tr> <tr><td>3</td><td>3</td><td>3</td></tr> <tr><td>4</td><td>3</td><td>3</td></tr> <tr><td>5</td><td>3</td><td>3</td></tr> <tr><td>6</td><td>3</td><td>3</td></tr> <tr><td>7</td><td>3</td><td>3</td></tr> </tbody> </table>	x	y (Input)	y (Output)	-7	-5	-5	-6	-4	-4	-5	-3	-3	-4	-2	-2	-3	-1	-1	-2	0	0	-1	1	1	0	2	2	1	3	3	2	3	3	3	3	3	4	3	3	5	3	3	6	3	3	7	3	3
x	y (Input)	y (Output)																																																
-7	-5	-5																																																
-6	-4	-4																																																
-5	-3	-3																																																
-4	-2	-2																																																
-3	-1	-1																																																
-2	0	0																																																
-1	1	1																																																
0	2	2																																																
1	3	3																																																
2	3	3																																																
3	3	3																																																
4	3	3																																																
5	3	3																																																
6	3	3																																																
7	3	3																																																

Table 17: Fixed-Point Overflow Modes (cont'd)

Mode	Description	Image
Wrap around	When the input value exceeds the output data type, the output value is wrapped from the maximum value to the minimum value, or from the minimum value to the maximum value, thus cycling through the range of permitted values.	<p style="text-align: center;">Wrap around</p>
Sign-Magnitude Wrap Around	When the input value exceeds the output data type, the output value reaches the maximum value, and then begins decreasing to return to the minimum value. In an underflow situation, the minimum value is reached, and begins increasing to return to the maximum value.	<p style="text-align: center;">Wrap around sign magnitude</p>

## Configuring Overflow Warnings

In case of either wrap or saturate, you might want to know when overflow occurs. You can define how Simulink handles each of these overflow conditions in the model by modifying the Model Configuration Parameters. Open the Model Configuration Parameters by selecting **Modeling → Model Settings → Model Settings**, or typing `Ctrl-E`. In the Configuration Parameters dialog box, under the Diagnostics > Data Validity tab, you can specify values for the Wrap on Overflow and Saturate on Overflow fields. Each of these fields can have one of the following settings:

- **none**: Simulink takes no special action to report or handle the overflow.
- **warning**: A message will be displayed in the diagnostic viewer. The next warning for the same block will be ignored, and simulation will continue.
- **error**: An error message will be displayed in the diagnostic viewer and the simulation will be terminated.



**TIP:** Help for this dialog box can be found under [Model Configuration Parameters: Data Validity Diagnostics](#).

## Creating a Top-Level Subsystem Module

In order to generate output from the Vitis Model Composer model the top-level of the Model Composer model must contain the Vitis Model Composer Hub block, as described in [Vitis Model Composer Hub](#), as well as a subsystem that encapsulates the application design. To generate output from the subsystem that is instantiated at the top-level of the design, only HLS Library blocks and a limited set of specific Simulink® blocks can appear in the subsystem. The HLS Library blocks define the functions to be compiled for the packaged IP or compiled C++ code. The top-level design can contain other blocks and subsystem modules that serve different purposes, such as simulation, but the primary application must be completely contained within the specified subsystem.



**TIP:** The specific Simulink blocks that are supported in the Model Composer subsystem also appear in the Model Composer block library.

To create a subsystem from within a model, add one or more blocks to the model canvas, select the blocks, and turn the selected blocks into a subsystem:

1. Drag and drop blocks onto the model canvas in the Simulink Editor, as explained in [Adding Blocks to a Model](#).
2. Select one or more blocks, and right-click to use the Create Subsystem from Selection command.
3. Name the subsystem, giving it the same name you want to assign to the generated output application or IP.
4. Double-click the subsystem to open it in the Simulink Editor, and continue the design.

The Explorer bar and Model Browser in Simulink help you navigate your model:

- The Explorer bar lets you move up and down the hierarchy, or back and forth between different views in the Simulink Editor.
- The Model Browser provides a view of the Model Hierarchy, and lets you select and open different levels to quickly move through the hierarchy of the design.

# Importing C/C++ Code as Custom Blocks

## Introduction

Vitis Model Composer lets you import C or C++ code to create new blocks that can be added to a library for use in models alongside other HLS Library blocks. This feature lets you build custom block libraries for use in Model Composer.

Vitis Model Composer provides two options for importing C or C++ code as an HLS block: `xmcImportFunction` and the HLS Kernel block. These options differ in the capabilities available and in how the function interfaces are defined.

- The `xmcImportFunction` allows you to import any HLS function with any supported HLS interfaces. `xmcImportFunction` creates from the function a block that can be connected to other HLS blocks. Vitis Model Composer can simulate the created block alongside other blocks and generate RTL code for synthesis. To create an IP for integration into a larger Vivado or Vitis design, it is necessary to enclose the HLS function in a subsystem (perhaps along with other HLS blocks) and use the Interface Spec block to designate streaming ports for the subsystem.
- The HLS Kernel block allows you to import an HLS Kernel, which is an RTL component suitable for use in the Vitis embedded system development flow. HLS Kernels are proper IPs on their own. HLS Kernels are required to have AXI streaming interfaces. In Vitis Model Composer, the HLS Kernel block can be interfaced with other HLS or AI Engine blocks and run through the Hardware Validation Flow.

If you want to use Vitis Model Composer to interface the HLS design with an AI Engine design in a heterogeneous system, or if you want to validate the HLS design on hardware using Vitis Model Composer, you should use the HLS Kernel flow instead of the `xmcImportFunction` flow. The HLS Kernel flow is covered in [Chapter 6: Connecting AI Engine and Non-AI Engine Blocks](#).

The remainder of this chapter focuses on the `xmcImportFunction` flow.

## Using the `xmcImportFunction` Command

Vitis Model Composer provides the `xmcImportFunction` command, for use from the MATLAB command line, to let you specify functions defined in source and header files to import into Model Composer, and create Model Composer blocks, or block library. The `xmcImportFunction` command uses the following syntax:

```
xmcImportFunction('libName', {'funcNames'}, 'hdrFile', {'srcFiles'},  
{'srchPaths'}, 'options')
```

Where:

- `libName`: A string that specifies the name of the Model Composer HLS library that the new block is added to. The library can be new, and will be created, or can be an existing library.
- `funcNames`: Specifies a list (cell array) of one or more function names defined in the source or header files to import as a Model Composer block. An empty set, {}, imports all functions defined in the specified header file (`hdrFile`). For functions inside namespaces, full namespace prefix needs to be given. For example, to import the function 'sinf' in `hls_math`, the full function name '`hls::sinf`' needs to be specified.
- `hdrFile`: A string that specifies a header file (.h) which contains the function declarations, or definitions. This should be the full path to the header file if it is not residing inside the current working directory. For example, to import a function from `hls_math.h`, you need to specify the full path '`$XILINX_VIVADO/include/hls_math.h`'.



**IMPORTANT!** The function signature must be defined in the header file, and any Model Composer (XMC) pragmas must be specified as part of the function signature in the header file.

- `srcFiles`: Specifies a list of one or more source files to search for the function definitions. When used in a model, the header and source files, together with the main header file of the model, `headerFile`, will be compiled into the shared library for simulation, and copied into the Target Directory specified for output generation in the Model Composer Hub block, as described in [Vitis Model Composer Hub](#).
- `srchPaths`: Specifies a list of one or more search paths for header and source files. An empty set, {}, indicates no search path, in which case the code is looked for in the MATLAB current folder. Use '`$XILINX_VIVADO/include`' to include the HLS header files.

In addition, the `xmcImportFunction` command has the following options, which can follow the required arguments in any order:

- '`unlock`': Unlock an existing library if it is locked. The `xmcImportFunction` command can add blocks to an existing library, but it must be unlocked to do so.
- '`override`': Overwrite an existing block with the same name in the specified library.



**TIP:** The help for `xmcImportFunction` can be accessed from the MATLAB command line, using `help xmcImportFunction`, and provides the preceding information.

As an example, the following `simple.h` header file defines the `simple_add` function, with two double-precision floating point inputs and a pointer output. The function simply adds the two inputs and returns the sum as the output.

```
void simple_add(const double in1, const double in2, double *out) {  
    *out = in1 + in2;  
}
```

To import the `simple_add` function as a block in a Model Composer HLS library you can enter the following command at the MATLAB command prompt:

```
xmcImportFunction('SimpleLib',{'simple_add'},'simple.h',{},{})
```

Where:

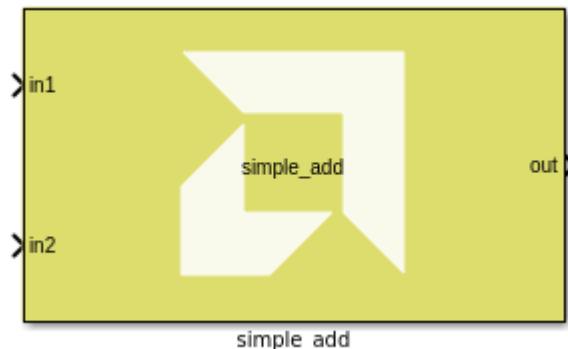
- `SimpleLib` is the name of the Model Composer HLS library to add the block to.
- `simple_add` is the function name to import.
- `simple.h` is the header file to look in.
- No C source files or search paths are specified. In this case, the function definition must be found in the specified header file, and only the MATLAB current folder will be searched for the specified files.



*TIP: Model composer will give you a warning if you attempt to import a block with the same function name as a block that is already in the specified library.*

When `xmcImportFunction` completes, the `SimpleLib` library model will open with the `simple_add` block created as shown below.

Figure 134: **simple\_add** Block



During simulation, the C/C++ code for the Library blocks will get compiled and a library file will get created and loaded. The library file will be cached to speed up initializing simulations on subsequent runs. You can change the source code underlying the library block without the need to re-import the block, or re-create the library, although the cached library file will be updated if you change the C/C++ source code.

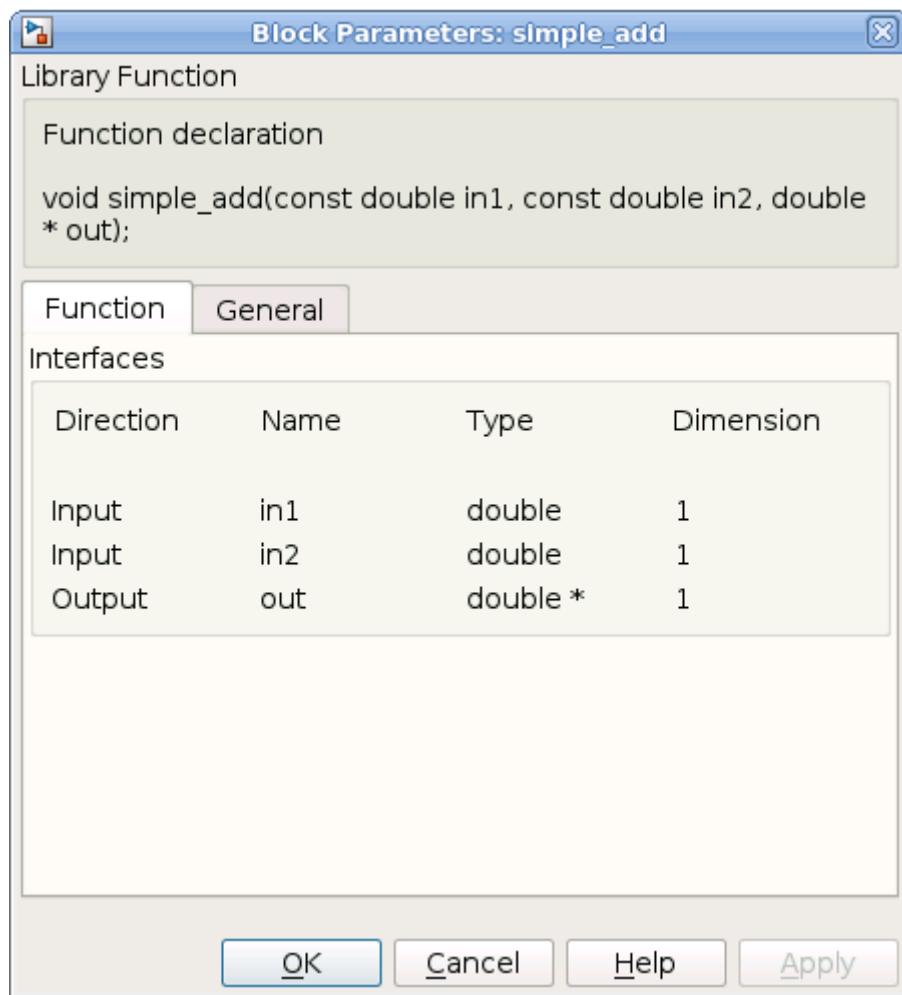
However, if you change the function signature, or the parameters to the function, then you will need to rerun the `xmcImportFunction` command to recreate the block. In this case, you will also need to use the `override` option to overwrite the existing block definition in your library.



**IMPORTANT!** You must rerun the `xmcImportFunction` command any time that you change the interface to an imported function, or the associated XMC pragmas. In this case, you should delete the block from your design and replace it with the newly generated block from the library. However, you can change the function content without the need to run `xmcImportFunction` again.

After the block symbol has been created, you can double-click on the symbol to see the parameters of the imported block. You can quickly review the parameters as shown in the following figure to ensure the function ports are properly defined.

Figure 135: **simple\_add** Block Parameters



## Importing C/C++ into Model Composer

While Vitis Model Composer lets you import C or C++ functions to create a library of blocks, it does have specific requirements for the code to be properly recognized and processed. The function source can be defined in either a header file (.h), or in a C or C++ source file (.c, .cpp), but the header file must include the function signature.

You can import functions with function arguments that are real or complex types of scalar, vectors, or matrices, as well as using all the data types supported by Model Composer HLS Library, including fixed-point data types. Model Composer also lets you define functions as templates, with template variables defined by input signals, or as customization parameters to be specified when the block is added into the model or prior to simulating the model. Using function templates in your code lets you create a Model Composer block that supports different applications, and can increase the re-usability of your block library. Refer to [Defining Blocks Using Function Templates](#) for more information.



**IMPORTANT!** The function signature must be defined in the header file, and any Model Composer (XMC) pragmas must be specified as part of the function signature in the header file.

The `xmc ImportFunction` command supports C++ functions using `std::complex<T>` or `hls::x_complex<T>` types. For more details, see the explanation in [Using Complex Types](#).

If the input of an imported function is a 1-D array, the tool can perform some automatic mappings between the input signal and the function argument. For example, if the function argument is `a[10]`, then the connected signal in Model Composer can be either a vector of size 10, or a row or column matrix of size `1x10` or `10x1`.

However, if all of the inputs and outputs of an imported function are scalar arguments, you can connect a vector signal, or a matrix signal to the input. In this case, the imported function processes each value of the vector, or matrix on the input signal as a separate value, and will combine those values into the vector, or matrix on the output signal. For example, a vector of size 10 connected to a scalar input, will have each element of the vector processed, and then returned to a vector of size 10 on the output signal.

You can import functions that do not have any inputs, and instead only generate outputs. This is known as a source block, and can have an output type of scalar, vector, complex, or matrix. You can also import source blocks with multiple outputs. The following example function has no input port, and `y` is the output:

```
#include <stdint.h>
#include <ap_fixed.h>

#pragma XMC OUTPORT y
#pragma XMC PARAMETER Limit
template <typename T>
void counter(T &y, int16_t Limit)
{
    static T count = 0;

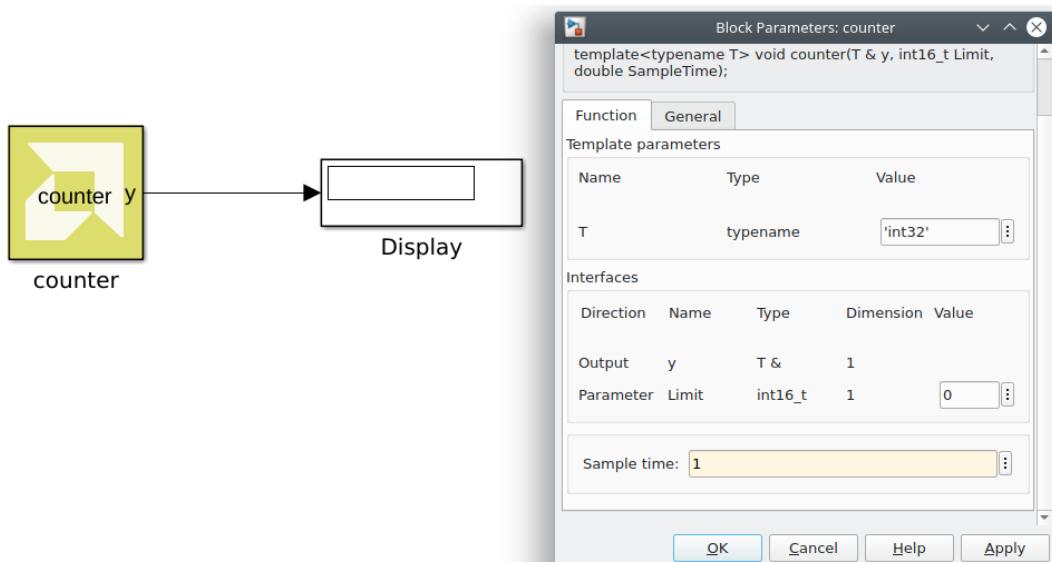
    count++;

    if (count > Limit)
        count = 0;
    y = count;
}
```



**TIP:** Because source blocks have no inputs, the `SampleTime` parameter is automatically added when the block is created with `xmcImportFunction` command, as shown in the Function declaration in the following image. The default value is -1 which means the sample time is inherited from the model. You can also explicitly specify the sample time by customizing the block when it is added to a model, as shown below.

Figure 136: Setting Sample Time for a Source Block



The direction of ports for the function arguments can be determined automatically by the `xmcImportFunction` command, or manually specified by pragma with the function signature in the header file.

- Automatically determining input and output ports:
  - The `return` value of the function is always defined as an output, unless the `return` value is `void`.
  - A formal function argument declared with the `const` qualifier is defined as an input.
  - An argument declared with a reference, a pointer type, or an array type without a `const` qualifier is defined as an output.
  - Other arguments are defined as inputs by default (for example, scalar read-by-value).
- Manually defining input and output ports:
  - You can specify which function arguments are defined as inputs and outputs by adding the `IMPORT` and `EXPORT` pragmas into the header file immediately before the function declaration.
  - `#pragma XMC IMPORT <parameter_name> [ , <parameter_name>... ]`
  - `#pragma XMC EXPORT <parameter_name> [ , <parameter_name>... ]`

In the following example `in` is automatically defined as an input due to the presence of the `const` qualifier, and `out` is defined as an output. The imported block will also have a second output due to the integer `return` value of the function.

```
int func(const int in, int &out);
```

In the following function `in` is automatically defined as an input, and `out` as an output, however, there is no `return` value.

```
void func(const in[512], int out[512]);
```

In the following example the ports are manually identified using pragmas that have been added to the source code right before the function declaration. This is the only modification to the original C++ code needed to import the function into Model Composer. In this example the pragmas specify which parameter is the input to the block and which parameter is the output of the block.

```
#pragma XMC IMPORT din
#pragma XMC EXPORT dout
void fir_sym (ap_fixed<17, 3, AP_TRN, AP_WRAP> din[100],
              ap_fixed<17, 3, AP_TRN, AP_WRAP> dout[100]);
```



**TIP:** *ap\_fixed* specifies a fixed-point number compatible with Vitis HLS.

---

Manually adding pragmas to the function signature in the header file to define the input and output parameters of the function is useful when your code does not use the `const` qualifier, and adding the `const` qualifier can require extensive editing of the source code when there is a hierarchy of functions. It also makes the designation of the inputs and outputs explicit in the code, which can make the relationship to the imported block more clear.

Some final things to consider when writing C or C++ code for importing into Model Composer:

- You should develop your source code to be portable between 32 bit and 64 bit architectures.
- Your source code can use Vitis HLS pragmas for resource and performance optimization, and Model Composer uses those pragmas but does not modify or add to them.
- If your code has static variables, the static variable will be shared across all instances of the blocks that import that function. If you do not want to share that variable across all instances you should copy and rename the function with the static variable and import a new library block using the `xmcImportFunction` command.

- If you use C (.c) source files to model the library function (as opposed to C++ (.cpp) source files), the .h header file must include an `extern "C"` declaration for the downstream tools (such as Vitis HLS) to work properly. An example of how to declare the `extern "C"` in the header files is as follows:

```
// c_function.h:  
#ifndef __cplusplus  
extern 'C' {  
#endif  
void c_function(int in, int &out);  
#ifdef __cplusplus  
}  
#endif
```

## Using Complex Types

C++ code for Vitis HLS can use `std::complex<T>` or `hls::x_complex<T>` to model complex signals for `xmclImportFunction` blocks.

The code generated by Model Composer uses `std::complex` for representing complex signals. If your imported C/C++ block function is modeled with `hls::x_complex`, when generating the output code Model Composer will automatically insert `std::complex-to-hls::x_complex` adapters to convert the complex types for the block ports.

The C++ code must also include the required header file for the complex type declaration. For the `hls::x_complex` type, the `xmcImportFunction` command must also include '`$XILINX_VIVADO/include`' search path for the `hls_x_complex.h` header file. For example, the following imports the `complex_mult` function, and specifies the needed include path:

```
xmcImportFunction('my_lib', {'complex_mult'}, 'complex_mult.h', {},  
{'$XILINX_VIVADO/include'});
```

## Example Functions Using Complex Types

```
#include "hls_x_complex.h"  
hls::x_complex<double>  
complex_mult(hls::x_complex<double> in1, hls::x_complex<double> in2)  
{ return in1 * in2; }  
  
#include <complex>  
std::complex<double>  
complex_mult2(std::complex<double> in1, std::complex<double> in2)  
{ return in1 * in2; }  
  
#include <complex>  
void  
complex_mult3(std::complex<double> in1, std::complex<double> in2,  
std::complex<double> &out1)  
{ out1.real(in1.real() * in2.real() - in1.imag() * in2.imag());  
    out1.imag(in1.real() * in2.imag() + in1.imag() * in2.real()); }
```

## Defining Blocks Using Function Templates

 **IMPORTANT!** To use template syntax, your function signature and definition should both be specified in a header file when running `xmcImportFunction`.

While it is common to write functions that accept only a predetermined data type, such as `int32`, in some cases you might want to create a block that accepts inputs of different sizes, or supports different data types, or create a block that accepts signals with different fixed-point lengths and fractional lengths. To do this you can use a function template that lets you create a block that accepts a variable signal size, data type, or data dimensions.

You can define blocks using function templates, as shown in the following example:

```
#include <stdint.h>
template <int ROWS, int COLS>
void simple_matrix_add(const int16_t in1[ROWS][COLS],
                      const int16_t in2[ROWS][COLS],
                      int16_t out[ROWS][COLS]) {
    for (int i = 0; i<ROWS; i++) {
        for (int j = 0; j<COLS; j++) {
            out[i][j] = in1[i][j] + in2[i][j];
        }
    }
}
```

The example uses the template parameters `ROWS` and `COLS`. The actual dimensions of the input and output arrays, `in1[ROWS][COLS]` for instance, are determined at simulation time by the dimensions of the input signals to the block. `ROWS` and `COLS` are template parameters used to define the dimensions of the function arguments, and also used in the body of the function, `i<ROWS` for example.

Use the command below to import the function into Model Composer:

```
xmcImportFunction('SimpleLib', {'simple_matrix_add'}, ...
    'template_example.h', {}, {}, 'unlock')
```



**TIP:** In the example above the ellipsis (...) is used to indicate a continuation of the command on the next line. Refer to [Continue Long Statements on Multiple Lines](#) in the MATLAB documentation for more information.

You can perform simple arithmetic operations using template parameters. For example, the following code multiplies the `ROWS` and `COLS` of the input matrix to define the output, as shown in the following figure.

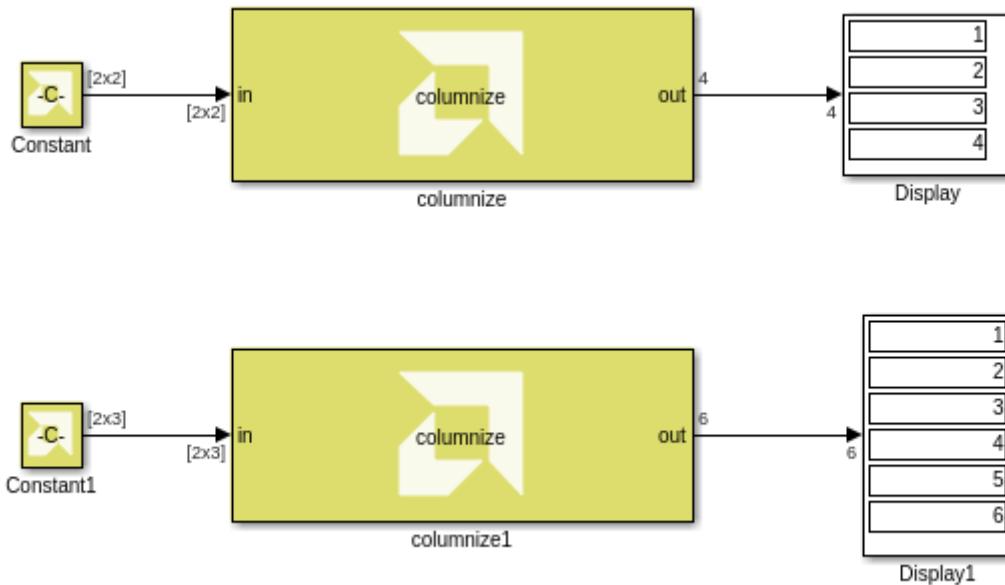
```
#include <stdint.h>
#pragma XMC IMPORT in
#pragma XMC OUTPORT out
template<int ROWS,int COLS>
void columnize(const int16_t in[ROWS][COLS], int16_t out[ROWS*COLS]) {
    for (int i = 0; i<ROWS; i++) {
```

```

        for (int j = 0; j<COLS; j++) {
            out[i*COLS+j] = in[i][j];
        }
    }
}

```

Figure 137: Columnize Function



Other simple supported operations include +, -, \*, /, %, <<, and >>, using both the template parameters and integer constants. For example:

```

template<int M, int N>
void func(const int in[M][N], int out[M*2][M*N]);

template<int ROWS, int COLS>
void func(array[2 * (ROWS + 1) + COLS + 3]);

```

You can also define a function template that uses a fixed-point data type of variable word length and integer length using function templates, as shown in the following example:

```

#include <stdint.h>
#include <ap_fixed.h>
#pragma XMC OUTPORT out
template <int WordLen, int IntLen>
void fixed_add(const ap_fixed<WordLen,IntLen> in1,
               const ap_fixed<WordLen,IntLen> in2,
               ap_fixed<WordLen+1,IntLen> &out) {
    out = in1+in2;
}

```

The example above uses the fixed point notations from Vitis HLS, which specifies the word length and the integer length. In Model Composer, as described in [Working with Data Types](#), you specify the word length and the fractional length. This requires you to use some care in connecting fixed point data types in Model Composer to the imported `fixed_add` block. For example, in the function above if `WordLen` is 16 and `IntLen` is 11, in Model Composer fixed point data type the word length is 16, and the fractional length is 5. For more information on fixed-point notation in Vitis HLS, refer to the *Vitis High-Level Synthesis User Guide* ([UG1399](#)).



**TIP:** As shown in the example above, simple arithmetic operations are also supported in the fixed point template parameter.

To import the `fixed_add` function and create a block in Model Composer, use the following command:

```
xmc ImportFunction('SimpleLib',{'fixed_add'},fixed_example.h',{},{'$XILINX_VIVADO/include'})
```

## Function Templates for Data Types

Function templates for data types are functions that can operate with generic data types. This lets you create library functions that can be adapted to support multiple data types without needing to replicate the code or block in the Vitis Model Composer HLS block library to support each type. The `xmc ImportFunction` command in Model Composer will create generic library blocks which the user of the block can connect to signals of any data types supported by the block.

The data type (`typename`) template parameters are resolved at simulation runtime, when the code and simulation wrapper are generated. The parameters are replaced during simulation by the actual data types that are specified by the signals connecting to the library block. The resolved data types can only be the types that Model Composer supports as discussed in [Working with Data Types](#).

To import a block that accepts multiple data types, you use a function template. For example:

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}
```

Or, in the case of a function with complex function arguments, the example would appear as follows:

```
#include <complex>
template <typename T>
void mult_by_two(std::complex< T > x, std::complex< T > *y)
{
    *Out = In1 * 2;
}
```

The determination of the type is made by Model Composer during simulation. The `typename` (or `class`) parameters are propagated from input signals on the block, or are customization parameters that must be defined by the user at simulation runtime.



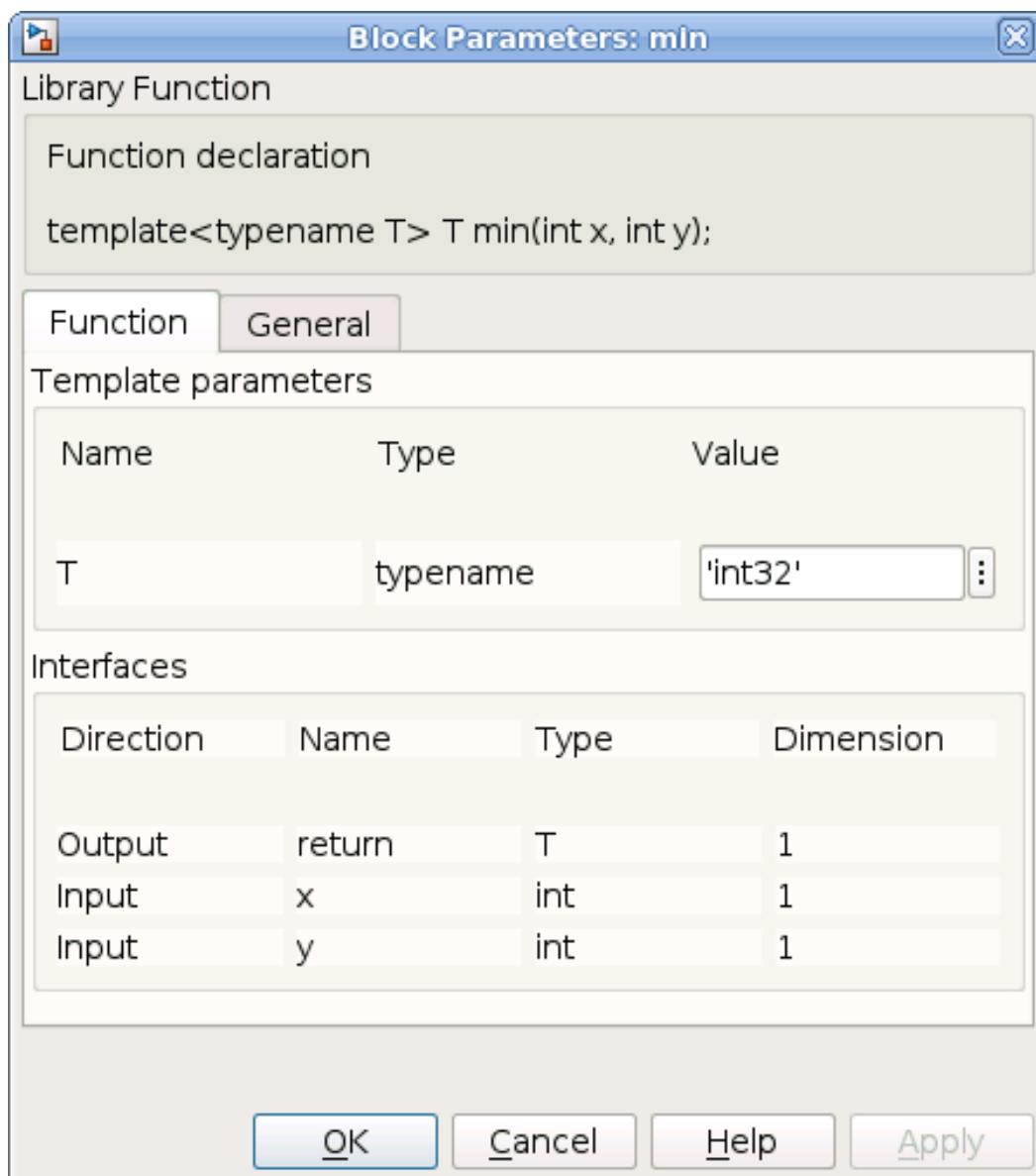
**IMPORTANT!** *The data type for a function or class cannot be propagated from an output.*

For example, the following function template specifies the parameter 'T' as a customization parameter. Because it is not associated with either input argument, 'x' or 'y', it must be specified by the user when the block is added to the model:

```
template <typename T>
T min(int x, int y) {
    return (x < y) ? x : y;
}
```

The Block Parameters dialog box for the generated Library Function block has an edit field to enter the template argument as shown in the following figure.

Figure 138: Library Function Block Parameters



In the template syntax, the data type template parameters for function or class can be specified with other template parameters. The order of specification is not important. For example:

```
template <typename T1, int ROWS, int COLS, int W, int I>
T1 func(T1 x[ROWS][COLS], ap_fixed<W, I> &y) {
...
}
```



**IMPORTANT!** In the example above, notice that the 'T1' template parameter is used to specify both the function return and the data type of the input 'x'. In this case, because it is a single template parameter, both arguments will resolve to the same data type that is propagated from the input signal to the block.

## SUPPORTED\_TYPES/UNSUPPORTED\_TYPES Pragma

When defining a data type (`typename`) template parameter (or `class`), you can also define the accepted data types for the variable by using either the SUPPORTED\_TYPES or UNSUPPORTED\_TYPES pragma as part of the function signature. This is shown in the following code example.

```
#pragma XMC IMPORT x
#pragma XMC IMPORT y
#pragma XMC SUPPORTED_TYPES T: int8, int16, int32, double, single, half
template <class T>
T max(T x, T y) {
    return (x > y) ? x : y;
}

#pragma XMC UNSUPPORTED_TYPES T: boolean
#pragma XMC IMPORT x, y
template <typename T>
T min(T x, T y) {
    return (x < y) ? x : y;
}
```

Model Composer supports an extensive list of data types as discussed in [Working with Data Types](#). To specify which of these data types the template parameter supports, you can either include the list of supported types, or the unsupported types. The SUPPORTED\_TYPES and UNSUPPORTED\_TYPES pragmas are simply two opposite views of the same thing:

- SUPPORTED\_TYPES: Specifies a template parameter name (`param`), and the list of data types that are accepted by that parameter. This implies the exclusion of all types not listed.

```
#pragma XMC SUPPORTED_TYPES param: type1, type2, ...
```

- UNSUPPORTED\_TYPES: Specifies a template parameter name (`param`), and the list of data types that are not accepted by that parameter. This implies the inclusion of all types not listed.

```
#pragma XMC UNSUPPORTED_TYPES param: type1, type2, ...
```

With the SUPPORTED\_TYPES or UNSUPPORTED\_TYPES pragma in place, Model Composer will check the type of input signal connected to the block to ensure that the data type is supported. Without the use of one of these pragmas, the data type template parameter will accept any of the data types supported by Model Composer.

## Function Template Specialization and Overloading

Specialization is supported for function templates in the `xmc ImportFunction` command. Model Composer will create the library block for the generic function template, supporting multiple data types, but the block will also include any specialized functions to be used when connected to input signals with matching data types. Both the generic function, and any specialization functions are compiled into the block DLL. For example:

```
template <typename T>
T min(T x, T y) {
    return (x < y) ? x : y;
}

template <>
bool min<bool>(bool x, bool y) {
...
}
```

In this case, Model Composer will call the specialized boolean form of the `min` function when the block is connected to boolean signals.

Overloading of a function with the same number of input/output arguments is also supported by Model Composer. For example, the following defines two forms of the function:

```
int func(int x);
float func(float x);
```

You can also overload a function template as shown below:

```
template <typename T>
int func(int x, T y);

template <typename T>
float func(float x, T y);
```



**TIP:** Overloading functions with different numbers of input/output arguments or different argument dimensions is not supported, and must be defined as separate functions.

## Defining Customization Parameters

Template parameters can be used to define the port array sizing and data type, and the parameters are defined by the input signals on the block. Additional customization parameters can also be defined, which are not defined by input signals, and thus must be defined by the user using the Block Parameter dialog box sometime before simulation runtime.

There are two methods to define customization parameters for a library function block:

- Using C/C++ [function templates](#).
- Assigning the Model Composer (XMC) [PARAMETER pragma](#) to a function argument, defining it as not connecting to an input or output of the block, but rather as a customization parameter.

There are pros and cons to both methods, which are discussed below.

## Function Templates

The first method, defines a function template that uses template parameters for customization. A template parameter defines a customization parameter in Model Composer if its value is not determined by an input signal, or derived by the function as an output. You can use customization parameters to define the values, data types, or data dimensions of output ports, or for parameters in use in the function body.



**IMPORTANT!** *The template function signature and function definition must be defined in the C/C++ header file.*

The template parameter for the function argument is defined using standard function template syntax, but the template parameter is not assigned to an input argument in the function signature. When the block is instantiated into a model, Model Composer identifies template parameters whose values are not determined by input signals, and lets the user define the values for those customization parameters. Values can be defined for customization parameters in the model any time prior to simulation.

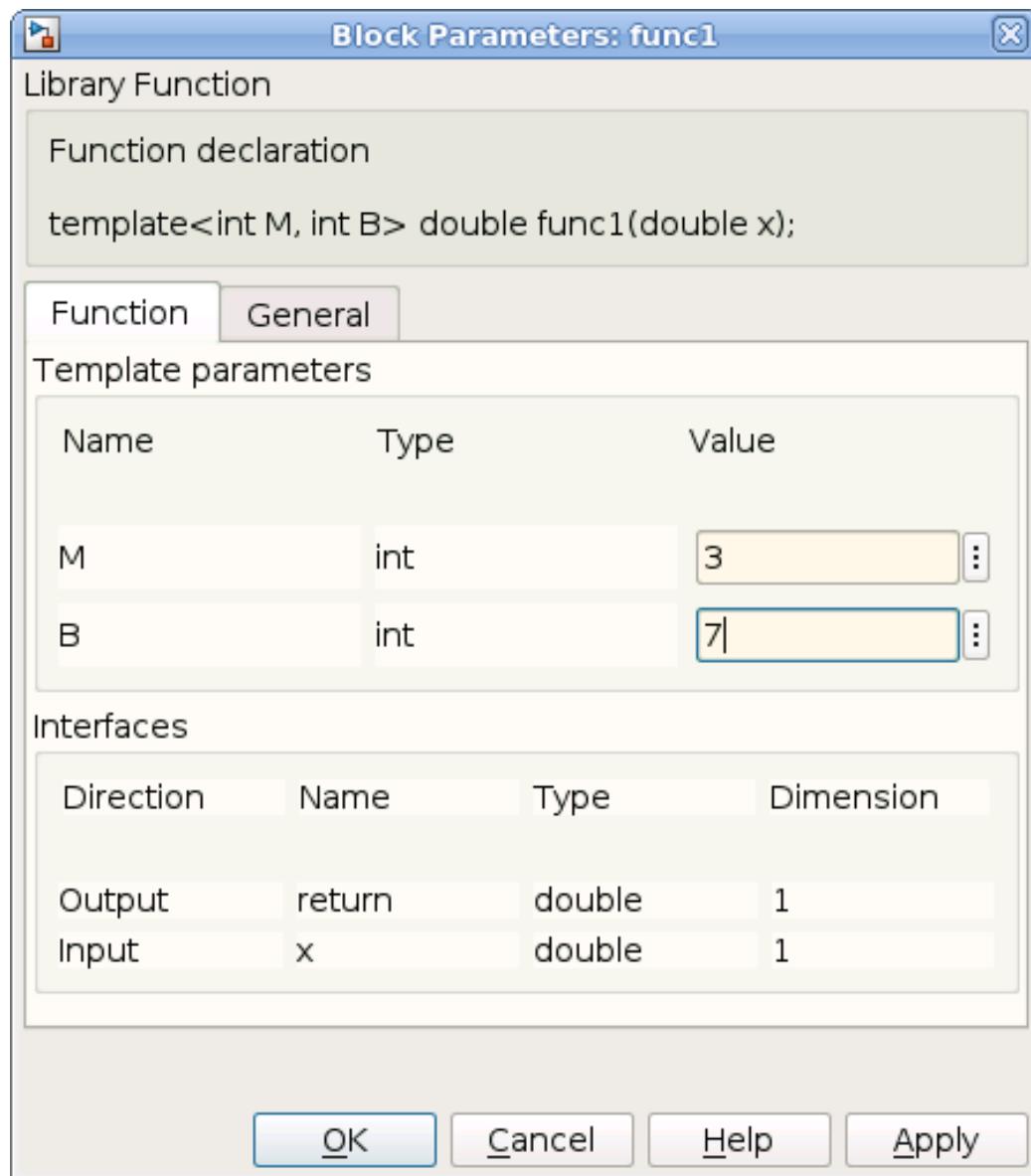
For function templates, the customization parameters can only be integer values to define the size or dimensions of a data type, or can only be scalar variables with definable data types. Model Composer defines a default value of 0 for integer parameters, and 'int32' for data type, or typename parameters.

In the function template example below, the template parameters 'M' and 'B' define customization parameters because the parameter values are not inherited from the input signal to the block. In this case, the parameters need to be customized by the user when the block is added to the model, or any time before simulation.

```
template <int M, int B>
double func1(double x) {
    return x * M + B;
}
```

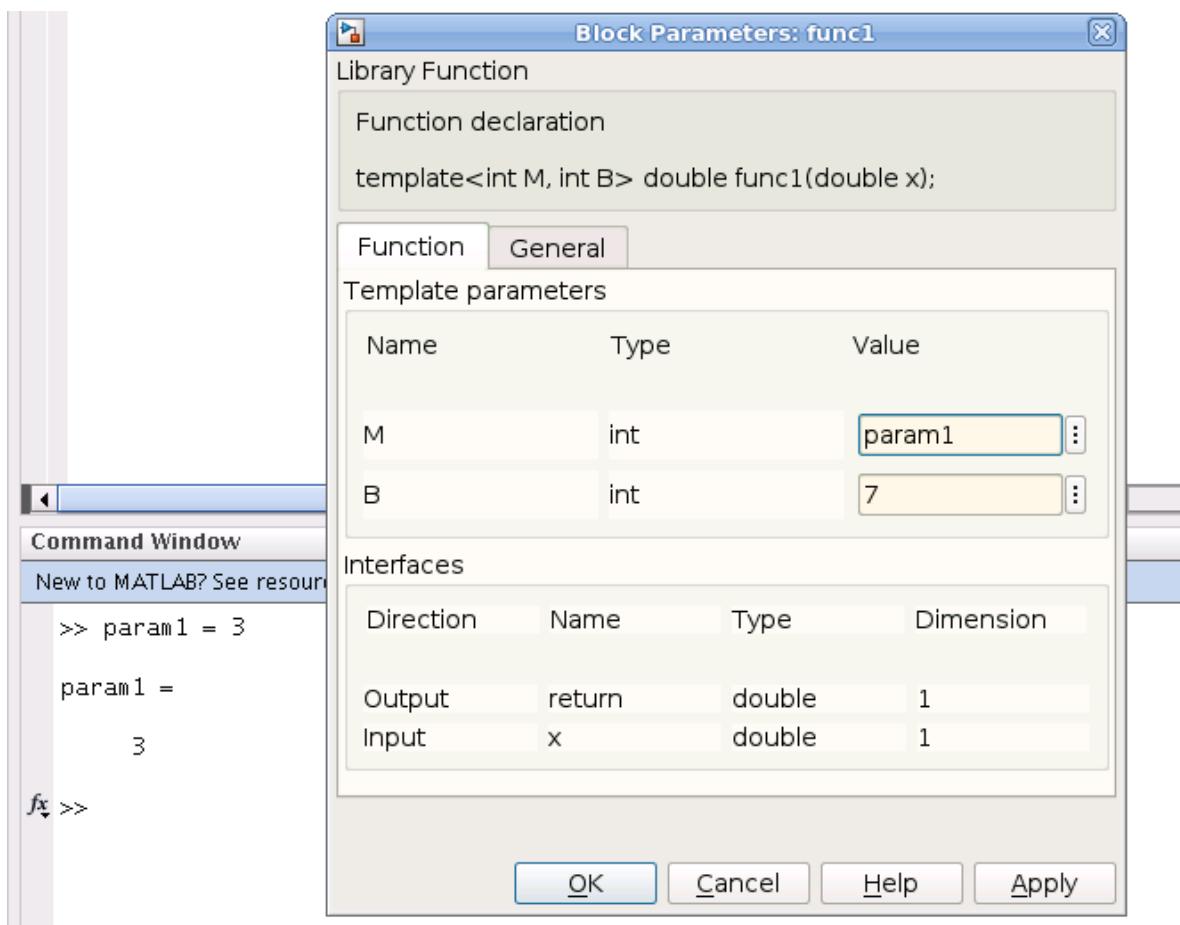
Customization parameters are displayed in the Block Parameters dialog box for the imported block as shown for the `func1` function below. Double click on a block in the model to open the Block Parameters dialog box, then enter the value for any editable parameters, such as 'M' and 'B' below.

Figure 139: Entering Parameter Values



Optionally, the user can also specify the name of a MATLAB workspace variable in the text field for the customization parameter, and have the value determined by Model Composer through the MATLAB variable. For example, the variable `param1` is defined in the MATLAB workspace, and used to define the value for 'M'.

Figure 140: Defining Parameters using Workspace Variables



## PARAMETER Pragma

The second method defines function arguments as customization parameters through the use of the Model Composer `PARAMETER` pragma.

To declare that a function argument is a customization parameter, you must add the `PARAMETER` pragma with the parameter name, or list of names, before the function signature in the header file. You can specify multiple parameters with one pragma, or have separate pragmas for each, as shown below.

```
#pragma XMC PARAMETER <name1>, <name2>
#pragma XMC PARAMETER <name3>
function declaration(<name1>, <name2>, <name3>)
```

When a function argument is declared a customization parameter by pragma, the `xmc ImportFunction` command will not create an input or output port on the block for that argument. It will be defined for use inside the function body only. When the block is added to a model, a customization field is added to the Block Parameter dialog box, and the user of the block can define values for the customization parameters.

Using the `PARAMETER` pragma on a function argument that is already driven by the input signal will be flagged as an error or a warning. In this case, the signal input propagation through the function will have higher precedence than the customization parameter.

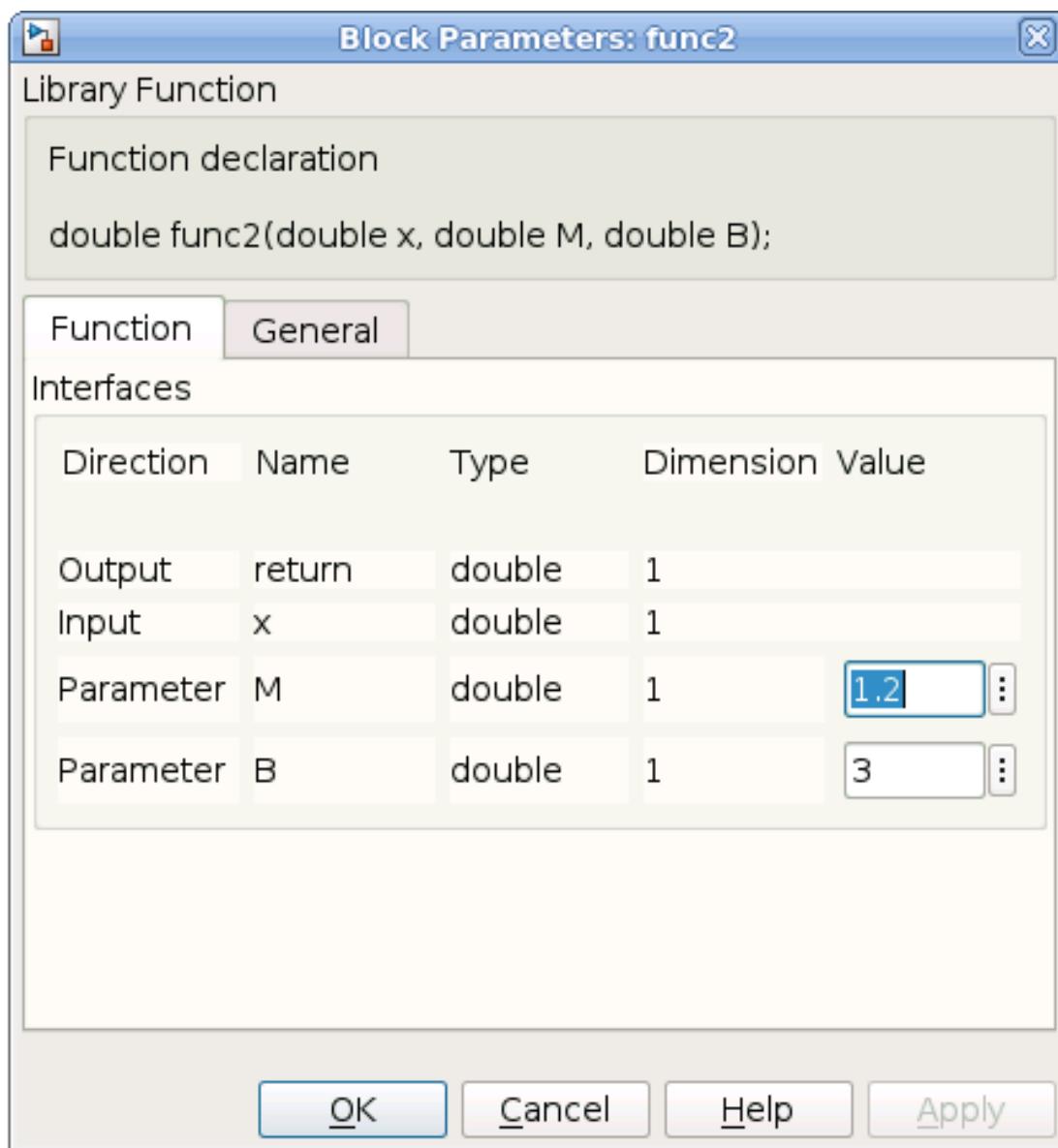
While the [function templates](#) method only supports scalar and integer type customization parameters, the `PARAMETER` pragma supports integer, floating point or fixed point data type for the parameters. The customization parameters also can be scalar, vector or a two-dimensional matrix. In addition, while the function template defines default values of 0 for integer types, and `int32` for the data type, the `PARAMETER` pragma lets you define default value for the parameters. Model Composers defines default values of 0 for all parameters that do not have user-defined defaults.

The example below uses the Model Composer `PARAMETER` pragma to define the customization parameters 'M' and 'B'.

```
#pragma XMC PARAMETER M, B
double func2(double x, double M = 1.2, double B = 3) {
    return x * M + B;
}
```

The 'M' and 'B' customization parameters also have default values assigned: `M=1.2, B=3`. The default values for the customization parameters are assigned to the arguments in the function signature, and are displayed in the Block Parameters dialog box when opened for edit, as shown below.

Figure 141: Customization Parameters with Defaults



**IMPORTANT!** If you define default values for the customization parameters of any argument, the C/C++ language requires that all arguments following that one must also have default values assigned, because the function can be called without arguments having default values. Therefore, you should add all customization parameters with default values at the end of the function argument list.

## Vector and Matrix Customization Parameters

The PARAMETER pragma method can also be used to specify customization parameters with vector and matrix dimensions, or values. In the following example the `coef` vector is defined by the pragma as a customization parameter:

```
#pragma XMC PARAMETER coef
#pragma XMC IMPORT din
#pragma XMC OUTPORT dout
#pragma XMC SUPPORTS_STREAMING
void FIR(ap_fixed<17, 3> din[100], ap_fixed<17, 3> dout[100],
ap_fixed<16, 2> coef[52]);
```

The constant array values of the customization parameter are entered in MATLAB expression format.

**Note:** commas are optional:

- Vector parameter: [val1, val2, val3, ...]
- Matrix parameter (row-major order): [val11, val12, val13, ...; val21, val22, val23, ...; ...]

## Interface Output Types and Sizes

Customization parameters can also be used to directly set the data types and dimension size for output ports whose values are not determined by inputs to the function. In the function below, the template variables define the word length and fractional length of the `ap_fixed` data type and the array size.

```
template <typename T1, int N1, int W2, int I2, int N2>
void func(const T1 in[N1], ap_fixed<W2, I2> out[N2]) {
...
}
```

The template variables 'W2', 'I2' and 'N2' define customization parameters because the values must be set by the user rather than determined from the input arguments. However, Model Composer recognizes that the template variables 'T1' and 'N1' are specified on the input port, and so the data type (`typename`) and the size of the input vector are not customization parameters, but rather get defined by the input signal on the block.

To set the data type for output ports, or arguments used in the body of the function, the `typename` specified must be one of the Model Composer supported data types, including the signed or unsigned fixed data types.

## Model Composer Supported Data Types

- **Supported Typenames:**
  - 'int8'

- 'uint8'
- 'int16'
- 'uint16'
- 'int32'
- 'uint32'
- 'double'
- 'single'
- 'x\_half"
- 'boolean'
- 'x\_sfix<n1>\_En<n2>'
- 'x\_ufix<n1>\_En<n2>'

In the example function below, while the `typename` for 'T1' is determined by the input signal, you can set the `typename` for 'T2' in the Block Parameters dialog box on the mask, when the block is added to a model, or before simulation runtime:

```
template <typename T1, int N1, typename T2, int N2>
void func(const T1 in[N1], T2 out[N2]) {
    ...
}
```

## Pragmas for `xmcImportFunction`

### **XMC SUPPORTS\_STREAMING**

The Vitis Model Composer `SUPPORTS_STREAMING` pragma indicates that the array parameters of a function support streaming data. This means that each element of the array is accessed once in a strict sequential order and indicates to Model Composer to optimize the design for streaming data. There can be no random access to the non-scalar arguments of a function to which the `SUPPORTS_STREAMING` pragma is applied.

The following example illustrates the difference between random access and sequential access. The `transform_matrix` output array of the `create_transform_matrix` function is addressed in a random order. It accesses the last row of the `transform_matrix` first, followed by the first and second row. This prevents the block from supporting streaming data.

```
void create_transform_matrix(const float angle, const float center_x,
                           const float center_y, float transform_matrix[3]
                           [3]) {
    float a = hls::cosf(angle);
    float b = hls::sinf(angle);

    transform_matrix[2][0] = 0;
    transform_matrix[2][1] = 0;
```

```
transform_matrix[2][2] = 0;  
  
transform_matrix[0][0] = a;  
transform_matrix[0][1] = b;  
transform_matrix[0][2] = (1-a)*center_x-b*center_y;  
  
transform_matrix[1][0] = -b;  
transform_matrix[1][1] = a;  
transform_matrix[1][2] = b*center_x +(1-a)*center_y;  
}
```

To change this function to support streaming data, it can be modified as depicted below to address the `transform_matrix` output array in a sequential manner:

```
#pragma XMC SUPPORTS_STREAMING  
void create_transform_matrix(const float angle, const float center_x,  
                           const float center_y, float transform_matrix[3]  
[3]) {  
    float a = hls::cosf(angle);  
    float b = hls::sinf(angle);  
  
    transform_matrix[0][0] = a;  
    transform_matrix[0][1] = b;  
    transform_matrix[0][2] = (1-a)*center_x-b*center_y;  
  
    transform_matrix[1][0] = -b;  
    transform_matrix[1][1] = a;  
    transform_matrix[1][2] = b*center_x +(1-a)*center_y;  
  
    transform_matrix[2][0] = 0;  
    transform_matrix[2][1] = 0;  
    transform_matrix[2][2] = 0;  
}
```

As shown in the preceding example, to specify that an imported function supports streaming, simply add the `SUPPORTS_STREAMING` pragma in the C or C++ header file before the function declaration:

```
#pragma XMC SUPPORTS_STREAMING
```

If the function has array arguments that are accessed sequentially, but `SUPPORTS_STREAMING` is not specified, then a subsystem using that block will not be implemented in a streaming architecture. This means the performance of the function will not be optimized.



**IMPORTANT!** If your function accesses the array arguments in random order, you must not specify the `SUPPORTS_STREAMING` pragma or an error will be returned when generating output or verifying your design.

The following is an example of a function that accesses the array arguments in a strictly sequential order, supporting the streaming of data. This function flips the rows of an input image horizontally. The function accesses the input image in a sequential order and buffers two rows of the input image in a circular buffer. Once two full rows are buffered, the function writes the buffer content to the function argument in a sequential order. As such, this function supports streaming, and uses the SUPPORTS\_STREAMING pragma to specify it.

```
#ifndef _MY_FUNCS
#define _MY_FUNCS

#include <stdint.h>

#pragma XMC IMPORT in1
#pragma XMC OUTPORT out1
#pragma XMC SUPPORTS_STREAMING
#pragma XMC BUFFER_DEPTH 4+2*WIDTH
// This function reverses each of the rows of the input image.
template<int WIDTH, int HEIGHT>
void
flip(uint8_t in1[HEIGHT][WIDTH],
      uint8_t out1[HEIGHT][WIDTH])
{
#pragma HLS dataflow

    uint8_t buf[2][WIDTH];

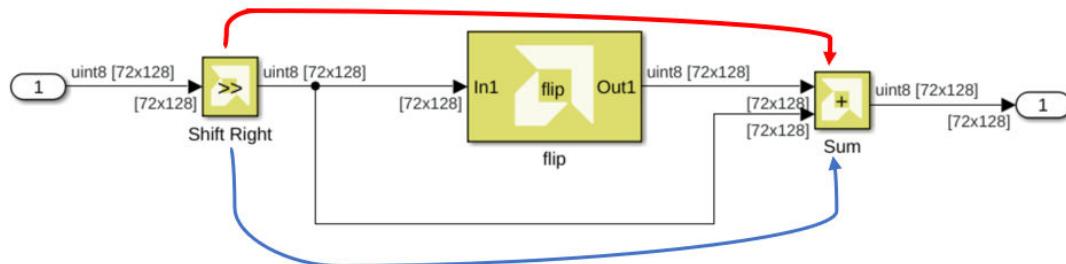
    int readBuf = 0;
    int writeBuf = 0;
    for (int row = 0; row < HEIGHT + 2; ++row) {
        for (int col = 0; col < WIDTH; ++col) {
#pragma HLS DEPENDENCE array inter false
#pragma HLS PIPELINE II=1
            if (row < HEIGHT) {
                buf[writeBuf][col] = in1[row][col];
                if (col == WIDTH-1) {
                    ++writeBuf;
                    writeBuf = (3 == writeBuf) ? 0 : writeBuf;
                }
            }
            if (row > 1) {
                out1[row - 2][col] = buf[readBuf][WIDTH- 1 - col];
                if (col == WIDTH-1) {
                    ++readBuf;
                    readBuf = (3 == readBuf) ? 0 : readBuf;
                }
            }
        }
    }
#endif
```

## XMC BUFFER\_DEPTH

The Vitis Model Composer `BUFFER_DEPTH` pragma provides information for properly sizing the buffers that connect the blocks in an implementation. These buffers are implemented as FIFOs in hardware. By default, Model Composer sets the depths of these buffers to 1. However, if your design has re-convergent paths (two paths converging into the same node) and the processing of data from the blocks of one path are not in lockstep with the processing of data from the other path, then a deadlock can occur. To avoid the deadlock the depth of one or more of the buffers on the paths can be increased to store the data. The following example illustrates this concept.

In the following diagram the `Sum` block consumes both the output signal of the `flip` block (red path), and the output of the `Shift Right` block (blue path). The `flip` block has been created with the `xmcImportFunction` command, and its source code is shown in the `flip` function previously described.

Figure 142: Buffer Depth



From the code for the `flip` block, you can see that the block needs to read 2 full rows before producing the first output. If the `BUFFER_DEPTH` pragma is not specified for the block, Model Composer sets the buffer sizes to 1 for the signals in the diagram. This results in deadlock, because the `flip` block reads 257 pixels from the input FIFO before producing the first output. However, by default, the parallel blue path feeding the second input of `Sum` has only enough storage for 1 pixel.



**TIP:** Vitis HLS provides some capability to detect deadlocks during C/RTL co-simulation. In case a deadlock is detected, the tool prints out messages showing which FIFOs are involved in the deadlock, to help identify FIFOs that might require a `BUFFER_DEPTH` of more than 1.

To change the default `BUFFER_DEPTH`, as shown in the `flip` function, place the pragma in the header file before the function declaration:

```
#pragma XMC BUFFER_DEPTH <depth>
```

Where `<depth>` specifies the buffer depth, and can be specified as a value or an expression.

By specifying `#pragma XMC BUFFER_DEPTH 4 + 2 * WIDTH` in the `flip` function, Model Composer can determine that there is an imbalance in processing among the re-convergent paths, and address this imbalance by setting the buffer depth for the second input to the `Sum` block (blue path) to match the buffer depth of the `flip` block.



**TIP:** Determining the minimal buffer depth might require a bit of trial and error because it also depends on the timing of the reads and writes into the FIFOs in the RTL code. In the `flip` function example, 256 (or  $2 * WIDTH$ ) was not sufficient `BUFFER_DEPTH`, but 260 (or  $4 + 2 * WIDTH$ ) prevented the deadlock.

## XMC THROUGHPUT\_FACTOR

The Model Composer `THROUGHPUT_FACTOR` pragma provides some control over the throughput of an `xmcImportFunction` block. You can add the `THROUGHPUT_FACTOR` pragma to your function header file, along with the `SUPPORTS_STREAMING` pragma as shown in the following example:

```
#pragma XMC THROUGHPUT_FACTOR TF_param: 1,2,4
#pragma XMC SUPPORTS_STREAMING
template<int ROWS, int COLS, int TF_param>
void DilationWrap(const uint8_t src[ROWS][COLS], uint8_t dst[ROWS][COLS])
```

The syntax of the pragma as shown in the prior example is:

```
#pragma XMC THROUGHPUT_FACTOR TF_param: 1,2,4
```

Where:

- The `TF_param` must be an `int` type template parameter, as is in the example above.
- It is optional, though recommended, to specify any specific throughput factors that are supported by the function. In the example above, `1,2,4` specifies the supported throughput factors in the pragma, expressed as positive integers, and must include the value 1. If you do not explicitly specify the throughput factors, the `TF_param` is assumed to be valid for any positive throughput factor up to the upper limit of 16 that is supported by Model Composer.

As discussed in [Controlling the Throughput of the Implementation](#), you specify the throughput factor for the model in the Model Composer Hub block. You can specify a throughput factor for the Hub block that divides evenly into one of the `THROUGHPUT_FACTOR` values on the `xmcImportFunction` block.



**IMPORTANT!** If the throughput factor of the Hub block does not match, or does not divide evenly into the `THROUGHPUT_FACTOR` specified by the `xmcImportFunction` block, then the throughput is reduced to 1 for the block function.

Note the following requirements:

- `THROUGHPUT_FACTOR` pragma must be used on Template functions.
- `THROUGHPUT_FACTOR` pragma must be used with `SUPPORTS_STREAMING` pragma.

- Only one THROUGHPUT\_FACTOR pragma can be specified for an `xmcImportFunction` block.
- The block function will be called with actual arguments that have cyclic ARRAY\_reshape directives with factor=TF (see example below). For more information on the ARRAY\_reshape pragma, refer to [HLS Pragmas](#) in the *Vitis Reference Guide* ([UG1702](#)).
- The read accesses from a non-scalar input argument of the function should be compliant with the requirements for streaming, and AMD Vitis™ HLS should be able to combine groups of TF reads into 1 read of the reshaped array.
- The write accesses into a non-scalar output argument of the function should be compliant with the requirements for streaming, and AMD Vitis™ HLS should be able to combine groups of TF writes into 1 write of the reshaped array.

The following is an example function specifying both SUPPORTS\_STREAMING and THROUGHPUT\_FACTOR pragmas:

```
#include <stdint.h>

#pragma XMC THROUGHPUT_FACTOR TF: 1, 2, 4, 8, 16
#pragma XMC SUPPORTS_STREAMING
template<int TF>
void mac(const int32_t In1[240], const int32_t In2[240], const int32_t
In3[240],
         int32_t Out1 [240])
{
    #pragma HLS ARRAY_RESHAPE variable=In1 cyclic factor=TF
    #pragma HLS ARRAY_RESHAPE variable=In2 cyclic factor=TF
    #pragma HLS ARRAY_RESHAPE variable=In3 cyclic factor=TF
    #pragma HLS ARRAY_RESHAPE variable=Out1 cyclic factor=TF

    for (uint32_t k0 = 0; k0 < 240 / TF; ++k0) {
        #pragma HLS pipeline II=1
        int32_t Product_in2m[TF];
        int32_t Sum_in2m[TF];
        int32_t Product_in1m[TF];
        int32_t Sum_outm[TF];
        for (uint32_t k1 = 0; k1 < TF; ++k1) {
            Product_in2m[k1] = In2[(k0 * TF + k1)];
        }
        for (uint32_t k1 = 0; k1 < TF; ++k1) {
            Sum_in2m[k1] = In3[(k0 * TF + k1)];
        }
        for (uint32_t k1 = 0; k1 < TF; ++k1) {
            Product_in1m[k1] = In1[(k0 * TF + k1)];
        }
        for (uint32_t k1 = 0; k1 < TF; ++k1) {
            int32_t Product_in2s;
            int32_t Sum_in2s;
            int32_t Product_in1s;
            int32_t Product_outs;
            int32_t Sum_outs;
            Product_in2s = Product_in2m[k1];
            Sum_in2s = Sum_in2m[k1];
            Product_in1s = Product_in1m[k1];
            Product_outs = Product_in1s * Product_in2s;
            Sum_outs = Product_outs + Sum_in2s;
            Sum_outm[k1] = Sum_outs;
        }
    }
}
```

```
        }
        for (uint32_t k1 = 0; k1 < TF; ++k1) {
            Out1[(k0 * TF + k1)] = Sum_outm[k1];
        }
    }
```

## Adding Your Library to Library Browser

To use the imported blocks in your library, you can simply open the Simulink model of your library, and copy blocks into a new Model Composer model. However, if you want to see your library listed in the Library Browser, and be able to drag and drop blocks from the library into new models, after running the `xmcImportFunction` command you must prepare your library using the following process.

1. Enable Library Browser parameter.
2. Save the library.
3. Create `slblocks.m` script for the library.
4. Add path to MATLAB, or add library to MATLAB path.
5. Refresh Library Browser.



**TIP:** Setting up the library using this process is only required for newly created libraries, rather than existing libraries which have already been setup.

To enable the library to be available in the Library Browser, you must turn on the `EnableLBRepository` parameter for the library. After importing a block into a new library using `xmcImportFunction`, with the library open, you must use the following command from the MATLAB command line prior to saving your library:

```
set_param(gcs, 'EnableLBRepository', 'on');
```

This parameter identifies the library as belonging to the Library Browser. However, that is just the first step. Save the library by clicking **File** → **Save** from the main menu, or by clicking the button in the toolbar.

Libraries in the Library Browser also require the presence of a script in the same directory as the library model, called `slblocks.m`, that defines the metadata associated with the library. You can create this script by copying an existing script from another library, or copying it from the MATLAB installation, or by editing the following text and saving it as `slblocks.m`:

```
function blkStruct = slblocks
    % This function adds the library to the Library Browser
    % and caches it in the browser repository

    % Specify the name of the library
```

```
Browser.Library = 'newlib';
% Specify a name to display in the library Browser
Browser.Name = 'New Library';

blkStruct.Browser = Browser;
```

Notice the `Browser.Library` specifies the name of the library model minus the `.slx` file extension, and `Browser.Name` specifies the display name that will appear in Library Browser.

**Note:** Each library should appear in a separate directory, with the `<library>.slx` file and the `slblocks.m` script for that library.

After creating the library and the `slblocks.m` script, you need to either add the library location to the MATLAB path so that MATLAB can find it, or copy the library to a folder that is already on the MATLAB path. You can type `path` at the MATLAB command prompt to see the current path that MATLAB uses. You can also add the library directory to the MATLAB path using the following commands:

```
addpath ('library_folder')
savepath
```

Where:

- `'library_folder'`
- `savepath` is a string that saves the current search path to `pathdef.m`.



**TIP:** To remove a folder from the MATLAB path you can use the following command sequence:

```
rmpath ('library_folder')
savepath
```

Finally, to view the new library in the Library Browser, from the left side of the Library Browser window right-click and select the **Refresh Library Browser** command. This will load the library into the tool. You should now be able to view the imported blocks and drag and drop them into your models.

## Debugging Imported Blocks

Vitis Model Composer provides the ability to debug C/C++ code that has been imported using the `xmcImportFunction` command or as an HLS Kernel or AI Engine block.

This feature lets you build the C/C++ function with debug information, and load it into Simulink for simulation. Once loaded, you can step into the C/C++ code of a specific imported block, and debug the function. The debugging environment lets you set debug break points in the C/C++ code, step through, and observe the intermediate results to verify the function in the context of the simulation. Debugging C/C++ code during Simulink simulation provides a natural flow. You can set desired input stimulus in Simulink, and observe the effect stepping through the code.

## Using Vitis Debugger

The Vitis IDE provides a debug environment that can be invoked directly from Vitis Model Composer.

The Vitis IDE debug environment has many features found in traditional GUI-based debug environments, such as GDB. You can add break points to the code, step over or step into specific lines of codes, loops, or functions, and examine the state of variables and force them to specific values.

To invoke the Vitis Debugger from Vitis Model Composer, do the following:

1. Open the model that contains the C/C++ source code block you would like to debug.
2. Run `vmcLaunchVitisDebugger` in the MATLAB® Command Window.

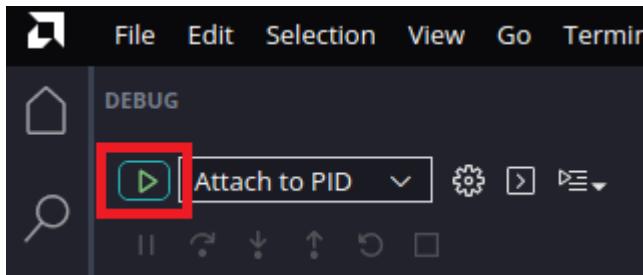
After a few moments, the Vitis IDE loads and the following message appears.

```
Opening Vitis Debugger...
*****
* Follow these steps in AMD Vitis:
* (1) On the leftmost toolbar, click on the 'Debug' button.
* (2) Inside the 'Debug' panel, select 'Attach to PID' and click on 'Start Debugging'.
* (3) Wait for the debugger to attach the process.
* (4) You can open source files in Vitis and set breakpoints at this stage.
*
* After completing the above steps, go back to your model and simulate it. *
*****
```

3. Select the **Debug** button on the Vitis IDE menu.



4. Inside the Debug panel, make sure Attach to PID is select in the drop-down menu. Click **Start Debugging**.



Wait for the debugger to finish attaching to process. When finished, the status bar at the bottom of the Vitis window turns orange and indicate clangd: idle.



5. In the Vitis Debugger, open the source file you would like to debug. (Select **File**→**Open File...**)
6. On the source code line where you would like the debugger to stop, place a breakpoint by clicking to the left of the line number. A red circle displays when the breakpoint has been successfully placed.

```
4 void SingleWindow::detect_Singl
5 {
6     count++;
7     if (count % AVERAGE ==0)
8     {
9         prevmode = false;
10    }
```

**Note:** When debugging HLS kernels on Windows, it is not possible to set breakpoints in the source code editor.

Instead, the breakpoints must be set in the `_ide/launch.json` file. Modify the `autorun` section of the file by adding a `break` statement, similar to the code snippet below. The file must be saved and the debugger restarted for the breakpoint to enable.

```
"autorun": [
    "handle SIGSEGV nostop noprint",
    "set breakpoint pending on",
    "break detectSingleWindow.cpp:82"
]
```

7. Return to Simulink and run the model.

```
4 void SingleWindow::detect_Sin
5 {
6     count++;
7     if (count % AVERAGE ==0)
8     {
9         prevmode = false;
10    }
```

You can now use the Debug controls on the left side of the screen to control program execution, view the call stack and data variables, and enable and disable breakpoints. For more information on the capabilities of the Vitis Debugger, refer to *Vitis Unified Software Platform Documentation: Embedded Software Development (UG1400)*.

When finished debugging, click **Stop** in the Debug panel.



When model execution hits the breakpoint that you set, the Vitis Debugger pauses execution on that line, which becomes highlighted.

## Using Another Debugger

### Enable Debug Mode

When using another debugger, the `xmcImportFunctionSettings` command can be used to set up the debugging tool.

Debugging the C/C++ function requires the simulation model to be built in the debug mode, instead of being built for release. To enable the debug build, use the following command:

```
xmcImportFunctionSettings('build', 'debug')
```

Refer to [xmclImportFunctionSettings Command Syntax](#) for more information on the command options. Enabling debug mode in Windows operating system causes Model Composer to return the following messages in the MATLAB® Command Window:

```
Imported C/C++ code will be built with MinGW compiler. You can use gdb to
debug your C/C++ code.
MATLAB process ID is 4656.
You can also get the process ID by typing "feature getpid" in the MATLAB
command window.
```

The information returned above can be used to launch the default debug tool, and connect to the MATLAB process, as described in the following sections.



**TIP:** You can restore the release build environment, using the 'release' value of the 'build' option:

```
xmcImportFunctionSettings('build', 'release')
```

### Launch the Debug Tool

After enabling the debug build mode, the `xmcImportFunctionSettings` command returns a link to the suggested debugging tool.

A third-party debugger is required for debugging with Model Composer. The default debugger is GDB for both the Linux and Windows operating systems.

### Setting a Breakpoint for the Imported Function

When the debugger is launched, you can set a breakpoint for the C/C++ imported function in the current model. This will break, or pause the Simulink simulation at the point it enters the imported function. This lets you perform further debugging actions, such as stepping through the function, printing variable values, or listing lines of code. Refer to the documentation for your debugging tool for more information on specific commands, and debugging techniques.



**TIP:** The following commands are provided for GDB, as it is the default debugger for Model Composer.

Setting a breakpoint uses the function name of the imported function:

```
(gdb) break <function_name>
```

Because simulation has not yet started, GDB will respond that no symbol table is loaded, and indicate that you can use the `break` command to specify break points. This simply means that you can also specify breakpoints based on the source file for the imported C/C++ function, and line number, specified as follows:

```
(gdb) break <file name>:<line num>
```

For example: `break func3_d.h:10`



**IMPORTANT!** Blocks created from function templates, as described in [Defining Blocks Using Function Templates](#), require the file name and line number to set breakpoints.

## Connecting Debug to the MATLAB Process

With the breakpoint established, you can attach GDB to the MATLAB® process by using the GDB attach command, and specifying the process ID (PID) returned by the `xmcImportFunctionSettings` command when you set up the debug build mode, as previously discussed. Use the following command:

```
(gdb) attach <PID>
```



**IMPORTANT!** After it is attached to GDB, the MATLAB process is suspended. You must use the `continue` command to have the process resume running after the `gdb` prompt is returned:

```
(gdb) continue
```

At this point you are ready to start the Simulink® simulation, and begin debugging your design.

## xmcImportFunctionSettings Command Syntax

The `xmcImportFunctionSettings` command sets options for the import function feature in Vitis Model Composer. Options are specified in the form of name/value pairs. The current options include:

- **build:** Specifies the build environment for Model Composer. Supported values include:
  - **release:** The default build mode. Generates the specified output, and lets you perform simulation.
  - **debug:** Provides integration into a third-party compiler to let you step through and observe the imported C/C++ function using standard debug features.
- **compiler:** Specifies the third-party compiler to use for debugging purposes. The supported values are:
  - **default:** Compile with GCC or MinGW compiler for debugging using GDB debugger.

**Note:** GCC or MinGW, and GDB are included with the standard installation of Model Composer.

- **blocks:** Specifies the specific block or blocks that you want to observe during the debugging process. If the option is not specified, all imported function blocks in the current design are included for debugging. You can specify one or more blocks using the following command form:

```
xmcImportFunctionSettings('blocks', {'block1','block2', ...})
```

You can also unset the selection of the blocks using the following command:

```
xmcImportFunctionSettings('blocks', {})
```

You can use the MATLAB `gcb` command to get the block path for a specific block. The blocks must be specified as a list of blocks, even if only one block is specified. For example:

```
xmcImportFunctionSettings('blocks', {'xmc_optical_flow/Lucas-Kanade'})
```



**TIP:** The settings specified by `xmcImportFunctionSettings` remain set until you exit Model Composer, or change the options using `xmcImportFunctionSettings`.

---

# Generating Outputs

## Introduction



**IMPORTANT!** To generate output from the Vitis Model Composer HLS model, only HLS library blocks and a limited set of Simulink blocks can be used in the subsystem that is instantiated at the top-level of the design. The Simulink blocks compatible with output generation in Model Composer can be found in the HLS blockset.

Model Composer automatically compiles designs into low-level representations. However, a Model Composer model requires the addition of the Model Composer Hub block to configure compilation and generate outputs. Model Composer can create two different types of output from the model, as defined by the Export Type setting of the Model Composer Hub block:

- [IP Catalog](#)
- [HLS C++ Code](#)

## Vitis Model Composer Hub

Control implementation of the model.

### Description

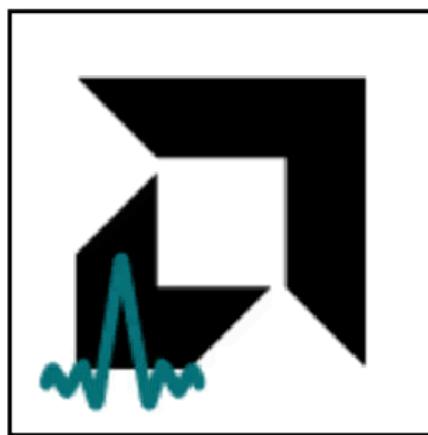
The Vitis Model Composer Hub block controls the behavior of the Vitis Model Composer tool.

You can specify the targeted design flow for the generated output, the directory path for the output, and the desired device and design clock frequency using the following tabs.

- The Hardware Selection pane helps with device or board selection.
- The Code Generation pane provides tabs to take the design through an elaboration workflow of analysis, validation on hardware, and export out of Vitis Model Composer. Different options will be displayed depending on if the subsystem is targeting HDL, HLS, or AIE.
- The Design Settings pane provides miscellaneous settings for the design.

## Library

AI Engine/Tools; HLS/Tools; Utilities/Code Generation.



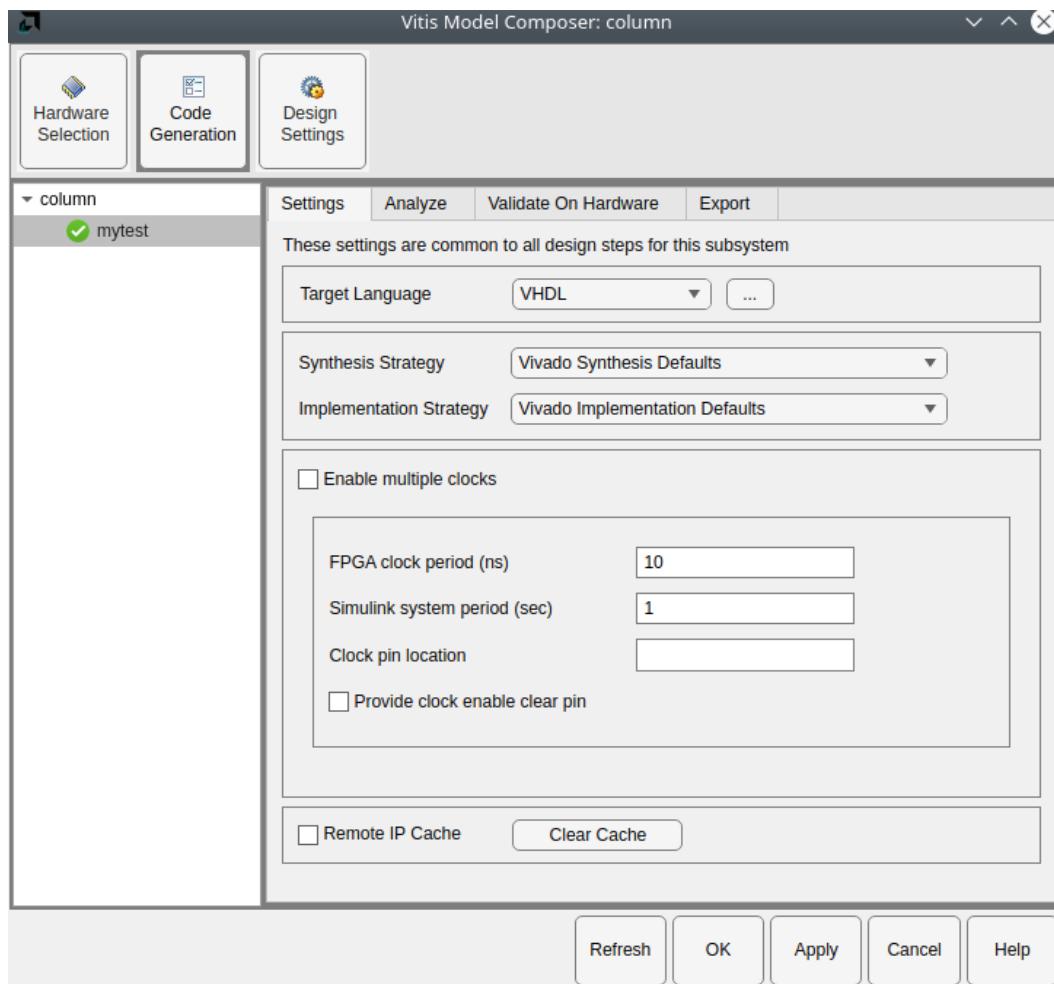
# Vitis Model Composer Hub

## Data Type Support

Data type support is not applicable to the Vitis Model Composer Hub block.

## Parameters

Figure 143: Vitis Model Composer Hub Block Parameters



The following section describes the configurable options available in each pane of the Vitis Model Composer Hub block.

- **Hardware Selection:** Clicking the browse button (...) displays the Device Chooser dialog box. This allows you to select a part, board, or platform to which your design is targeted. Vitis Model Composer obtains board and device data from the Vivado database.
- **Code Generation:** Select the subsystem name for which to generate code from the list on the left. Different settings will display depending on the type of code generation (HDL, HLS, or AI Engine) to be performed.
- **Settings (AIE):**
  - **AIE Compiler Options:** Provides the ability to pass additional command line options (debug options, execution target options etc.) to the AIE Compiler.

- **Analyze (AIE):**

- **Target Directory:** Specifies the work directory for performing the actions on this tab.
- **AIE Simulator Options:** Provides the ability to pass additional command line options to the AIE Simulator.
- **Simulation timeout (cycles):** When enabled, it specifies the number of cycles for which AIE simulation is run. The default value is 50000.
- **Collect profiling statistics and enable 'printf' for debugging:** When enabled, this option allows profiling data to be collected for analysis.
- **Collect trace data for Vitis Analyzer, viewing internal signals, and latency:** When enabled, this option collects trace data for signals within the AI Engine design to be viewed in Vitis Analyzer or the Simulation Data Inspector.
- **View AIE Simulation output and throughput:** Opens the Simulation Data Inspector to plot the outputs of the AI Engine subsystem as simulated by the AIE Simulator.
- **Open Vitis Analyzer:** Click to invoke the Vitis Analyzer tool. This option is only enabled after AI Engine Simulation has been ran at least once after enabling.

- **Settings (HDL):**

- **Target Language:** Specifies the HDL language to be used for compilation of the design. The possibilities are VHDL and Verilog.
  - **VHDL library:** Specifies the name of VHDL work library for code generation. The default name is `xil_defaultlib`.
  - **Use STD\_LOGIC type for Boolean or 1 bit wide gateways:** If your design's Hardware Description Language (HDL) is VHDL, selecting this option will declare a Boolean or 1-bit port (Gateway In or Gateway Out) as a STD-LOGIC type. If this option is not selected, Vitis Model Composer will interpret Boolean or 1-bit ports as vectors.

**Note:** When you enable this option and try to run Generate code and Run behavioral simulation in Vivado, you might see a failure during the elaboration phase.

- **Synthesis Strategy:** Choose a Synthesis strategy from the pre-defined strategies in the drop-down menu.
- **Implementation Strategy:** Choose an Implementation strategy from the pre-defined strategies in the drop-down menu.
- **Enable multiple clocks:** Must be enabled when the design has multiple clocks. This indicates to the code generation engine that the clock information for the various subsystems must be obtained from the respective clock tabs. If not enabled, then the design will be treated as a single clock design.

- **Number of clocks:** Defines the number of clocks in the design. The number of clock tabs that appear will be equivalent to the number of clocks. In each clock tab, you must select the subsystem and configure the clock settings of that subsystem.

- **FPGA clock period (ns):**

Defines the period in nanoseconds of the system clock. The value need not be an integer. The period is passed to the AMD 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.

- **Simulink system period (sec):** Defines the Simulink System Period, in units of seconds. The Simulink system period is the greatest common divisor of the sample periods that appear in the model. These sample periods are set explicitly in the block dialog boxes, inherited according to Simulink's propagation rules, or implied by a hardware oversampling rate in blocks with this option. In the final case, the implied sample time is in fact faster than the observable simulation sample time for the block in Simulink. In hardware, a block having an oversampling rate greater than one processes its inputs at a faster rate than the data. For example, a sequential multiplier block with an over-sampling rate of eight implies a (Simulink) sample period that is one eighth of the multiplier block's actual sample time in Simulink. This parameter can be modified only in a master block.

- **Clock pin location:** Defines the pin location for the hardware clock. This information is passed to the AMD implementation tools through a constraints file. This option should not be specified if the Vitis Model Composer design is to be included as part of a larger HDL design.
- **Provide clock enable clear pin:** This instructs Vitis Model Composer to provide a `ce_clr` port on the top-level clock wrapper. The `ce_clr` signal is used to reset the clock enable generation logic. The ability to reset clock enable generation logic allows designs to have dynamic control for specifying the beginning of data path sampling. This signal is important for modules that will be implemented in a DFX platform.

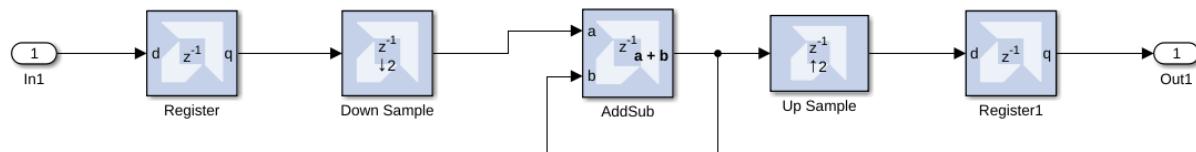
- **Analyze (HDL):**

**Note:** For more information about the tasks available on this tab, refer to [Performing Analysis in Vitis Model Composer](#).

- **Target Directory:** Specifies the work directory for performing the actions on this tab.
- **Perform Analysis:** Specifies whether an analysis (timing or resource) will or will not be performed on the Vitis Model Composer design when it is compiled. If None is selected, no timing analysis or resource analysis will be performed. If Post Synthesis is selected, the analysis will be performed after the design has been synthesized in the Vivado toolset. If Post Implementation is selected, the analysis will be performed after the design is implemented in the Vivado toolset.
- **Analysis Type:** Two selections are provided: Timing or Resource. After generation is completed, a Timing Analyzer table or Resource Analyzer table is launched.

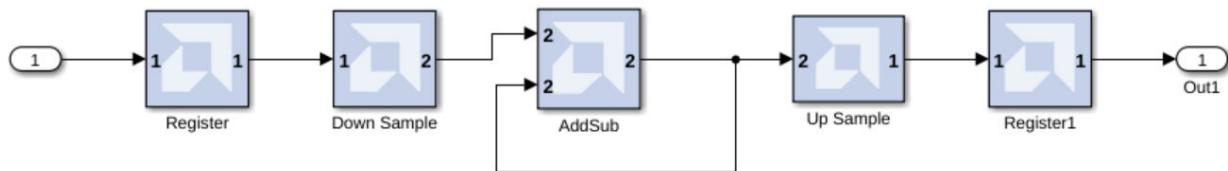
- **Block Icon Display:** Specifies the type of information to be displayed on each block icon in the model after compilation is complete. The various display options are described below.
  - **Default:** Displays the default block icon information on each block in the model. A block's default icon is derived from the `xbsIndex` library.

**Figure 144: Default Block Icon**



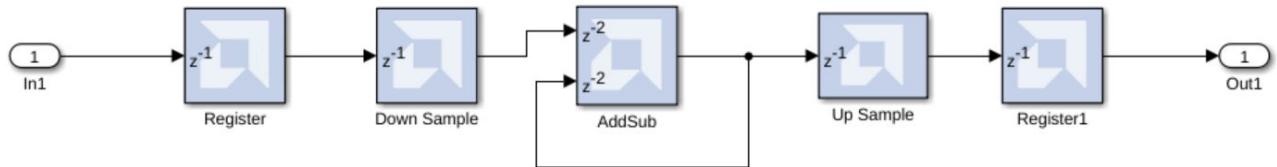
- **Normalized Sample Periods:** Displays the normalized sample periods for all the input and output ports on each block. For example, if the Simulink System Period is set to 4 and the sample period propagated to a block port is 4, then the normalized period that is displayed for the block port is 1; and if the period propagated to the block port is 8 then the sample period displayed would be 2 (a larger number indicates a slower rate).

**Figure 145: Normalized Sample Periods Icon**



- **Sample frequencies (MHz):** Displays sample frequencies for each block.
- **Pipeline stages:** Displays the latency information from the input ports of each block. The displayed pipeline stage might not be accurate for certain high-level blocks such as the FFT, RS Encoder/Decoder, Viterbi Decoder, etc. In this case the displayed pipeline information can be used to determine whether a block has a combinational path from the input to the output. For example, the Up Sample block in the following figure shows that it has a combinational path from the input to the output port.

Figure 146: Pipeline Stages



- **HDL port names:** Displays the HDL port name of each port on each block in the model.
- **Input data types:** Displays the data type of each input port on each block in the model.
- **Output data types:** Displays the data type of each output port on each block in the model.
- **Update the model:** Update the model to display the selected Block Icon type.
- **Create interface document:** When this check box is selected and the Generate button is activated for netlisting, Vitis Model Composer creates an HTM document that describes the design being netlisted. This document is placed in a documentation subfolder under the netlist folder.
- **Settings (HLS):**
  - **FPGA clock frequency:** Specifies the clock frequency in MHz for the targeted device. This frequency is passed to the downstream tool flow.
  - **Testbench stack size (MBytes):** This parameter prompts you to enter a larger stack size. When Create and run testbench is enabled, the Testbench stack size option specifies the size of the testbench stack frame during C simulation (CSIM). Occasionally, the default stack frame size of 10 MB allocate for execution of the testbench might be insufficient to run the test, due to large arrays allocated on the stack and/or deep nesting of subsystems. Typically when this happens, the test would fail with a segmentation fault and an associated error message. In such a case you can increase the size of the stack frame and rerun the test.
- **Analyze (HLS):**
  - **Target Directory:** Specifies the work directory for performing the actions on this tab.
- **Validate on Hardware:**
  - **Target Directory:** Specifies the work directory for performing the actions on this tab.
  - **HW System Type:** Choose between Baremetal or Linux hardware validation flow.
  - **Target:** Specify the target for hardware validation flow.
  - **Common SW Dir:** Provide the path to the folder containing the PetaLinux common images. This option is only enabled when a Linux HW System Type is selected.

- **Target SDK Dir:** Provide the path to the folder containing the target SDK. This option is only enabled when a Linux HW System Type is selected.
  - **Generate (BOOT.BIN/SD card image) after code generation:** When enabled, a BOOT.BIN (for baremetal HW system) or SD card image (for Linux HW system) will be generated after code generation.
  - **Export:**
    - **Export Directory:** Specifies the directory where the export products will be created.
    - **Export Type:**
      - **IP Catalog:** Export HDL or HLS subsystem as a Vivado IP. When IP Catalog is selected, the Settings (...) button brings up a dialog box that allows you to add a description of the IP that will be placed in the IP catalog.
      - **Synthesized Checkpoint:** Export HDL subsystem as a Synthesized Checkpoint for use in Vivado.
      - **HDL Netlist:** Export HDL subsystem as HDL code and netlist for use in Vivado.
      - **Vitis HLS:** Export HLS subsystem as HLS C/C++ code.
      - **Vitis Subsystem:** Export HDL, AIE, HLS or heterogeneous subsystem as a Vitis Subsystem (VSS).
  - **Graph Code:** Export AIE subsystem as AI Engine graph code.
  - **Generate testbench: HDL designs:** This instructs Vitis Model Composer to create an HDL test bench. Simulating the test bench in an HDL simulator compares Simulink simulation results with ones obtained from the compiled version of the design. To construct test vectors, Vitis Model Composer simulates the design in Simulink, and saves the values seen at gateways. The top HDL file for the test bench is named <name>\_testbench.vhd / .v, where <name> is a name derived from the portion of the design being tested.
- Note:** Testbench generation is not supported for designs that have gateways (Gateway In or Gateway Out) configured as an AXI4-Lite interface.
- AIE/HLS designs:** When enabled, Vitis Model Composer generates the test vectors while generating the code.
- **Design Settings:**

- **Treat this model as a legacy System Generator design for backward-compatibility:** Specify whether the Model Composer Hub block should treat your model as a legacy System Generator design. When you automatically upgrade a System Generator token to the Model Composer Hub block, this check box is automatically enabled. When the check box is enabled, you can use the Model Composer Hub block without requiring changes to your legacy designs. However, newer capabilities provided by the Model Composer Hub block, such as the Validate on Hardware Flow, will not be available.
- **Number of parallel AI Engine builds:** To speed up model compilation, Vitis Model Composer can build the AI Engine blocks in parallel, taking advantage of multiple cores on your machine. This value can be increased up to maximum number of cores on your machine.

## Controlling the Throughput of the Implementation

### ***Introduction***

Throughput of a system is one of its most important design criteria. For example, if you are designing a system that processes High Definition video frames (1920x1080) at 30 frames per second, the required throughput of your application would be 62,208,000 pixels per second (1920x1080x30). If you process one pixel per clock (in hardware terminology this is called an initiation interval of one, or II=1), your device needs to be clocked at over 62.2 MHz. If your requirements change, and you need to process a 4K video frame at 60 frames per second, the required throughput of the application would be 497,664,000 pixels per second (3840x2160x60), and your device needs to be clocked at over 497 MHz.

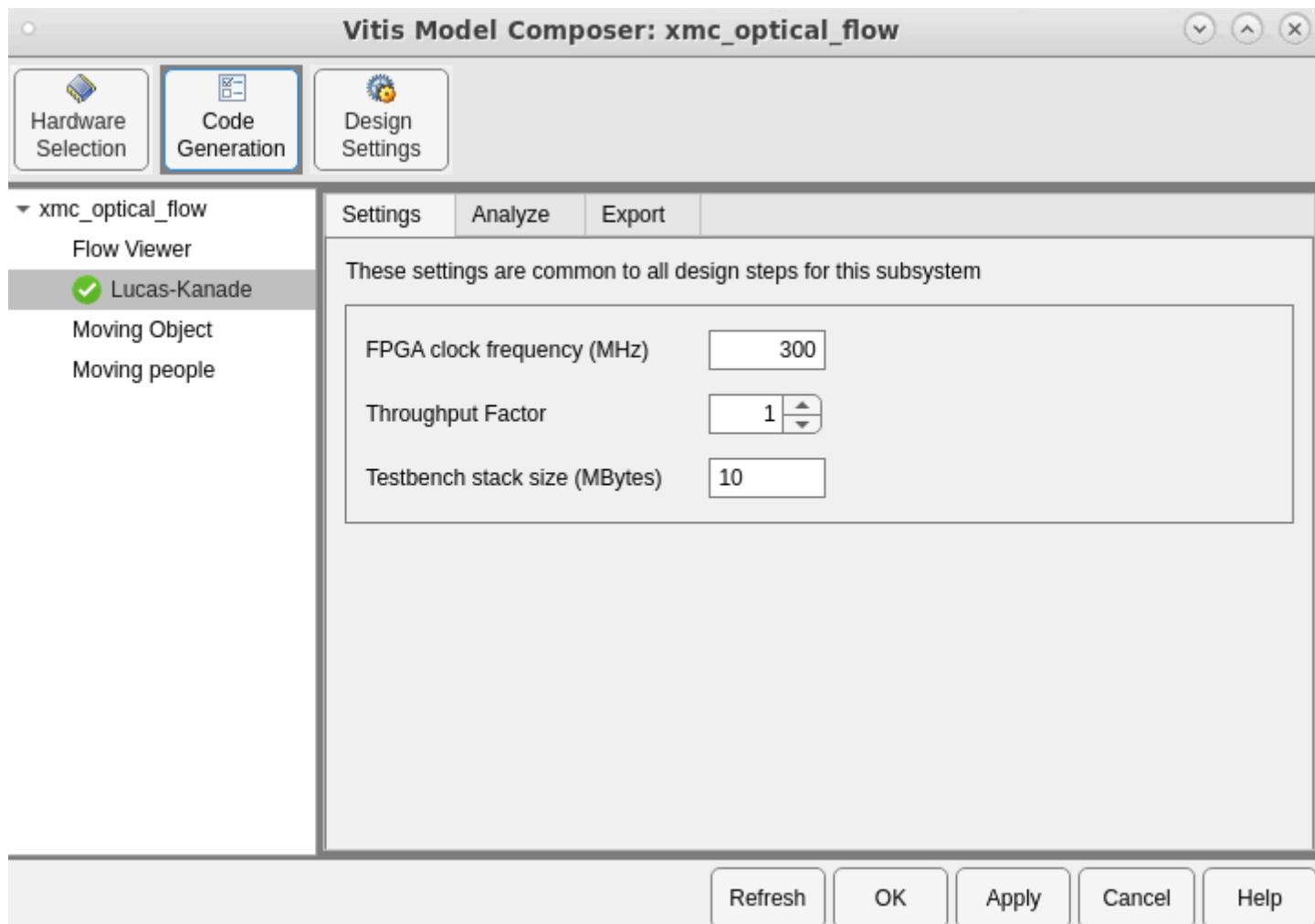
However, in practice you might not be able to achieve an initiation interval of one (II=1), therefore to achieve the desired throughput, you need to clock the device at even higher rates. In other applications, such as wireless communications, the clock frequencies needed to achieve a desired throughput could easily surpass the maximum clock frequency allowed for a device.

If you need to increase the throughput of your design, without increasing the clock frequency that your device is operating at (to operate at a clock frequency below the maximum allowed for a device, or to curtail power consumption), you can take advantage of the programmable logic nature of AMD FPGAs, and use parallelization techniques to process more samples per clock. Throughput control in Vitis Model Composer allows you to do that without making structural changes to your design in Simulink.

### ***Setting Throughput Factor from the Hub Block***

The Model Composer Hub block provides Throughput Factor to control the throughput of the generated code, or hardware design. The Throughput Factor specifies how many sample elements of the inputs are to be processed per clock cycle. By default, the Throughput Factor is set to 1. You can specify this factor by clicking the HLS Settings tab of the Vitis Model Composer Hub block, as shown below:

Figure 147: Model Composer Hub - Throughput Factor



The Throughput Factor must be between 1 and 16. Specifying a value greater than 1 will create parallel logic to process the transactions, using more resources from the device, and increasing the HLS and Vivado Synthesis runtime.

Code generation for designs with Throughput Factor > 1 imposes additional restrictions on the design. In case these restrictions are not met, Model Composer will return an error explaining the violation.

### ***Restrictions on Using Throughput Control***

A Throughput Factor of more than 1 can be achieved only if the design complies with the following restrictions:

- The Throughput Factor must be between 1 and 16.
- The subsystem must have at least one non-scalar input port.

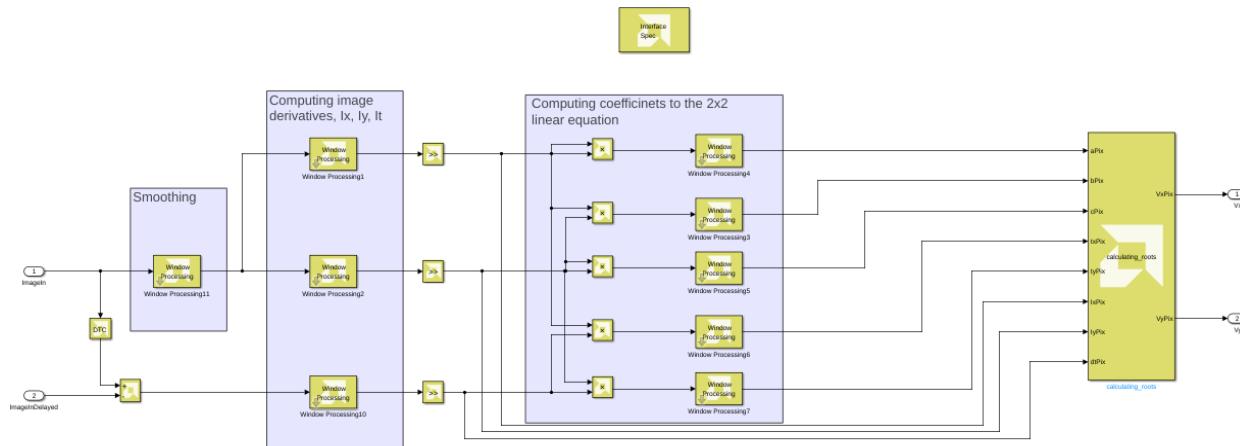
- All of the non-scalar ports of the subsystem must use either AXI4-Stream or FIFO interfaces.
- For any vector signal within the subsystem, but not inside a Window Processing block kernel, the vector length must be a multiple of the Throughput Factor.
- For any matrix signal within the subsystem, but not inside a Window Processing block kernel, the number of columns must be a multiple of the Throughput Factor.
- Except for blocks inside a Window Processing block kernel, the subsystem must not include any of the following blocks:
  - look up
  - matrix multiply, QR inverse
  - transpose, hermitian
  - sum of elements and product of elements with floating point input
  - cumulative sum, reducing min, reducing max
  - if action subsystem
- For blocks created using `xmcImportFunction`, refer to [XMC THROUGHPUT\\_FACTOR](#).

In summary, if the specified Throughput Factor is  $>1$ , and the design complies with all above mentioned restrictions, Model Composer can generate models that process samples concurrently. In cases where the design does not meet these restrictions, Model Composer will not generate an output model.

## ***Understanding Throughput Control Through an Example***

The following section demonstrates the benefits of using the Throughput Control feature with the Optical flow example design found in the list of examples for Vitis Model Composer HLS library.

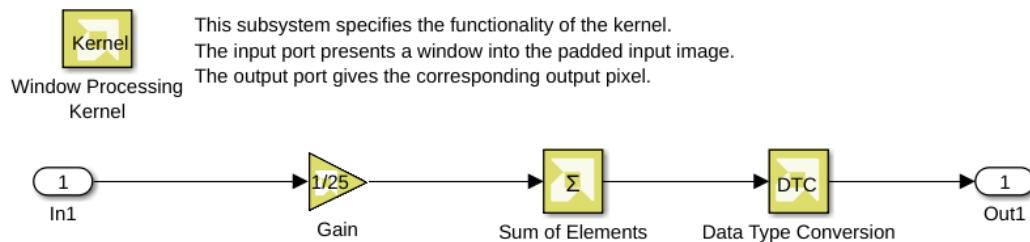
*Figure 148: Optical Flow Example*



This design uses the following blocks:

- Data Type Conversion, Subtract, Right Shift, Product
- Window processing blocks, with Gain, Sum of Elements, and Data Type Conversion.
- An Import Function block with the `calculating_roots` function.

**Figure 149: Window Processing Kernel**



All these blocks follow the element-wise application pattern, and comply with the restrictions previously discussed.

---

**IMPORTANT!** Direct use of the Sum of Elements block in subsystems using Throughput Control is restricted. In this example, the 'Sum of Elements' block is used in the Window Processing block but not directly in the top-level subsystem.

---

With the default Throughput Factor=1, Model Composer generates the code shown below:

```
void
Lucas_Kanade(hls::stream< uint8_t >& ImageIn, hls::stream< uint8_t >&
              ImageInDelayed, hls::stream< float >& Vx, hls::stream< float >& Vy)
{
    #pragma HLS INTERFACE axis port=ImageIn
    #pragma HLS INTERFACE axis port=ImageInDelayed
    #pragma HLS INTERFACE axis port=Vx
    #pragma HLS INTERFACE axis port=Vy
    #pragma HLS INTERFACE s_axilite port=return
    #pragma HLS dataflow
```

The IP reads its inputs, the image and delayed image, over AXI4-Stream. These streams will use a data width of 8 bits (1 pixel). Similarly pixels of the output image are streamed over an AXI4-Stream interface of data width 8 bits.

If you set TF=4, you get the code shown below.

```
void
Lucas_Kanade(hls::stream< xmc::MultiScalar< uint8_t, 4 > >& ImageIn,
              hls::stream< xmc::MultiScalar< uint8_t, 4 > >& ImageInDelayed,
              hls::stream< xmc::MultiScalar< float, 4 > >& Vx,
              hls::stream< xmc::MultiScalar< float, 4 > >& Vy)
{
    #pragma HLS INTERFACE axis port=ImageIn
    #pragma HLS data_pack variable=ImageIn
```

```
#pragma HLS INTERFACE axis port=ImageInDelayed
#pragma HLS data_pack variable=ImageInDelayed
#pragma HLS INTERFACE axis port=Vx
#pragma HLS data_pack variable=Vx
#pragma HLS INTERFACE axis port=Vy
#pragma HLS data_pack variable=Vy
#pragma HLS INTERFACE s_axilite port=return
#pragma HLS dataflow
```

This IP receives 4 pixels of the input image, and 4 pixels of the delayed input image, at the same time over AXI4-Stream that have data width of 32 bits. Inside the IP the logic has been duplicated so that 4 pixels are processed in parallel. The IP sends 4 pixels of the output image at a time over an AXI4-Stream, of data width 32 bits.

**Note:** `xmc::MultiScalar<T, N>` is a template struct defined in `xmcMultiScalar.h`. It is a struct that contains an array of `N` elements of type `T`.

The following table represents the Vitis HLS timing and resource estimates for optical flow design.

**Table 18: Optical Flow Design Timing/Resource Utilization Estimates**

	Throughput factor = 1	Throughput factor = 4	Throughput factor = 8
Clock Freq	300 MHz	300 MHz	300 MHz
Latency/II	41848/41834	10483/10469	5358/5344
BRAM_18k (Utilization %)	5	2	4
DSP48E (Utilization %)	2	9	19
FF (Utilization %)	8	30	59
LUT (Utilization %)	14	36	88

The second line in the table shows the initiation interval (II). At clock frequency of 300 MHz and Throughput Factors 4 and 8, the initiation interval of the design is reduced by a factor of approximately 4 and approximately 8 respectively, when compared with the initiation interval for Throughput Factor=1. Note that this comes at the cost of increasing resource utilization when the Throughput Factor increases.

For Throughput factor of one, the II is 41,848. The input to this design is a 200x200 pixel image frame and the value of II here indicates the number of clocks to process the entire frame. As such it takes slightly more than the duration of one clock cycle to process one pixel. As the Throughput Factor increases, the II to process one frame decreases, and the application processes more than one pixel per clock cycle.

## Defining the Interface Specification

Within the Simulink® environment, the inputs and outputs in your design are defined using "Inport" and "Outport" blocks. However, while moving from the software algorithm to an RTL implementation in hardware, these same input and output ports must be mapped to ports in the design interface, using a specific input-output (I/O) protocol, which typically operates with some real world delay. Part of developing your design is to specify how your design will communicate with other designs or IP in the system. You do this by specifying the interface to your design and choosing among a few standard I/O protocols.

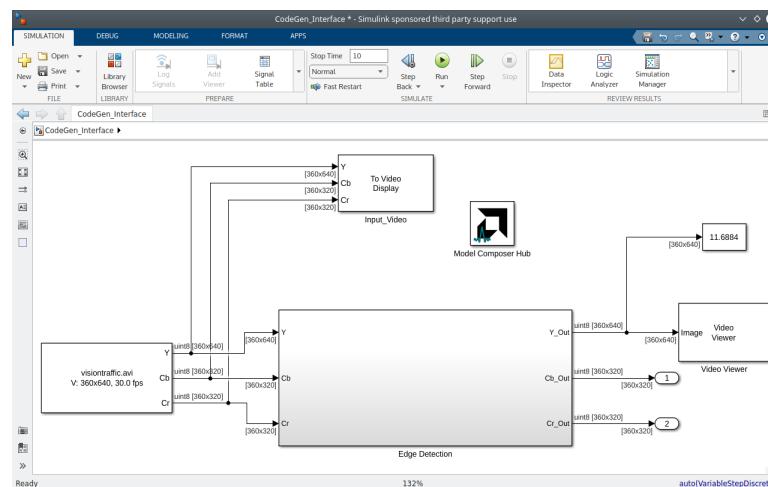
Model Composer requires the use of the Interface Specification (Interface Spec) block to define this I/O protocol.

Interface synthesis is supported only in the top-level subsystem module in the design, which Model Composer generates C++ code for. In the following figure, the Edge Detection module is the top-level subsystem module and the Interface Spec block must be instantiated inside that module.



**TIP:** Any Interface Spec blocks instantiated in other subsystems modules, or nested subsystem modules are ignored.

Figure 150: Top-Level Subsystem Module



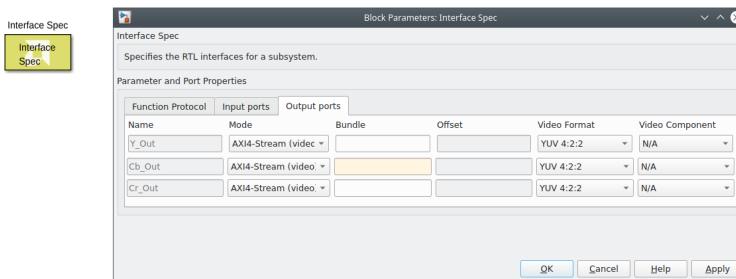
The Interface Spec block lets you control what interfaces should be used for the design. The Interface Spec affects only output code generation; it has no effect on Simulink simulation of your design. If you do not add an Interface Spec block to the subsystem module, Model Composer assigns default interfaces that might not be appropriate for the target platform or device. Therefore, it is recommended that you use the Interface Spec block to define the requirements of your subsystem module. The default function-level protocol is Handshake to specify control signals, and AXI4-Lite Slave for the function return. The default I/O protocol is AXI4-Lite Slave for scalar ports, and AXI4-Stream for non-scalar ports.

The Interface Spec block specifies how RTL ports are created from the function definition during interface synthesis. The ports in the RTL implementation are derived from the following.

- Any function-level protocol that defines control signals for the module.
- Function input and output arguments, and return values.
- Global variables accessed by the function but defined outside its scope.

**Note:** If a global variable is accessed, but all read and write operations are local to the subsystem, the resource is created in the design, and does not require the definition of a port.

Figure 151: Interface Spec Block



The Interface Spec block consists of three tabs defining the following information:

- **Function Protocol:** This is the block-level interface protocol which adds signal ports to the subsystem telling the IP when to start processing data. It is also used by the IP to indicate whether it accepts new data, or whether it has completed an operation, or whether it is idle.
- **Input Ports:** This tab automatically detects the input ports in your subsystem and lets you specify the interface protocol on those ports.
- **Output Ports:** This tab automatically detects the output ports on the subsystem module, and lets you specify the interface protocol.



**IMPORTANT!** The Interface Spec block has a current limitation of 8 input ports and 8 output ports on the subsystem module.

The Interface Specification displays and lets you configure the following features or parameters of the function or I/O port protocol.

**Table 19: Function Protocol Tab**

Attribute	Description
Mode	<p>Specifies a block-level protocol to add control signals to the subsystem module. The supported block-level protocols are:</p> <ul style="list-style-type: none"> <li>AXI4-Lite Slave: Implements the return port as an AXI4-Lite Slave interface, and adds the block-level control ports defined by the Handshake protocol. This is the default function protocol.</li> <li>Handshake: Defines a set of block-level control ports for the function to start processing input, and indicate when the design is <code>idle</code>, <code>done</code>, and <code>ready</code> for new input data.</li> <li>No block-level I/O protocol: No control ports are added to the subsystem.</li> </ul>
Bundle	Only valid with the AXI4-Lite Slave mode. Indicates that multiple ports should be grouped into the same interface. The bundle is specified by a <code>&lt;name&gt;</code> that cannot contain spaces or special characters.

**Table 20: Input/Output Port Tabs**

Attribute	Description
Name	Displays the port name, which cannot be changed from here.
Mode	<p>Specifies the port-level I/O protocols. The supported port-level protocols are:</p> <ul style="list-style-type: none"> <li>Default: Uses AXI4-Lite Slave interface for scalar ports, or AXI4-Stream interface for non-scalar ports.</li> <li>AXI4-Stream: Implements ports as an AXI4-Stream interface for high-speed streaming data.</li> <li>AXI4-Stream (video): Implements ports as an AXI4-Stream interface, with the additional assignment of <code>Video Format</code> and <code>Video Component</code> attributes.</li> <li>AXI4-Lite Slave: Implements the port as part of an AXI4-Lite Slave interface. All input or output ports with the same Bundle name are grouped into the same AXI4-Lite Slave interface.</li> <li>FIFO: Implements the port with a standard FIFO interface, combining data input or output with associated active-Low FIFO empty and full control signals.</li> </ul> <p><b>Note:</b> The FIFO interface is the most hardware-efficient approach for access to a memory element that is always sequential, that is, no random access is required. To read from non-sequential address locations, use the Block RAM interface.</p> <ul style="list-style-type: none"> <li>Constant: The data applied to the input port remains stable during the function operation, but is not a constant value that can be optimized. This allows internal optimizations to remove unnecessary registers.</li> <li>Valid Port: Implements a data port with an associated <code>valid</code> port to indicate when the data is valid for reading or writing.</li> <li>No protocol: No protocol. Neither the input or output data signals have associated control ports to indicate when data is read or written.</li> <li>Block RAM: Implements array arguments as a standard RAM interface. If you use the generated IP in Vivado IP integrator, the memory interface appears as a single port.</li> </ul>

**Table 20: Input/Output Port Tabs (cont'd)**

Attribute	Description
Bundle	Used with the AXI4-Stream (video) interfaces that have more than one color component. In this case there should be one port for each color component, and the ports should specify the same bundle <name> so they will be grouped into the same AXI4-Stream (video) interface. Also valid with the AXI4-Lite Slave mode. This parameter explicitly groups all interface ports with the same bundle <name> into the same AXI4-Lite Slave interface.
Offset	Only valid with the AXI4-Lite Slave mode. This parameter specifies an address offset associated with the port in the AXI4-Lite Slave address map. The offset is specified as a non-negative integer, with a default value of 0.
Video Format	Only valid with the AXI4-Stream (video) mode. This parameter specifies the color format for a video stream. Valid formats include: <ul style="list-style-type: none"> <li>YUV 4:2:2: Video format based on brightness (luminance) and color (chrominance), with reduced color content.</li> <li>YUV 4:4:4: Video format based on brightness (luminance) and color (chrominance), with full color content.</li> <li>RGB: Video format based on separate Red, Blue, and Green color signals.</li> <li>Mono: Specifies an audio format for your video.</li> </ul>
Video Component	Only valid with the AXI4-Stream (video) mode. This parameter specifies the color component for a video format that uses more than one color component. Valid video components include: <ul style="list-style-type: none"> <li>Y, U, V: Specifies one component of the YUV video format.</li> <li>R, G, B: Specifies one component of the RGB video format.</li> </ul>

The choice of port-level interface protocol should take into account the following considerations:

- Scalar ports can be implemented using any of the following protocols: Default, AXI4-Lite Slave, Constant, Valid Port, No protocol.
- Large array or matrix ports should use a streaming protocol such as AXI4-Stream, FIFO, or AXI4-Stream (video).
- Video signals can be transported over an AXI4-Stream (video) interface. In this case you also need to specify the video format YUV 4:2:2, YUV 4:4:4, RGB, or Mono. For video formats that have more than one color component, you also need to assign multiple ports to the same signal bundle, and you need to specify which port carries which color component. All of the ports that make up the video signal are implemented by a single AXI4-Stream interface that includes start-of-frame and end-of-line sideband signals. For more information refer to *AXI4-Stream Video IP and System Design Guide* ([UG934](#)).

## Generating Packaged IP for Vivado

Vitis Model Composer can automatically generate packaged IP for use in Vivado IP catalog.

When Model Composer generates output for the IP catalog, it first writes the C++ code as described in [Generating C++ Code](#), and then it synthesizes RTL code from the C++ code. This process begins when you set the Export Type in the Model Composer Hub block to IP Catalog, click Apply to confirm any changes, and click **Export**.

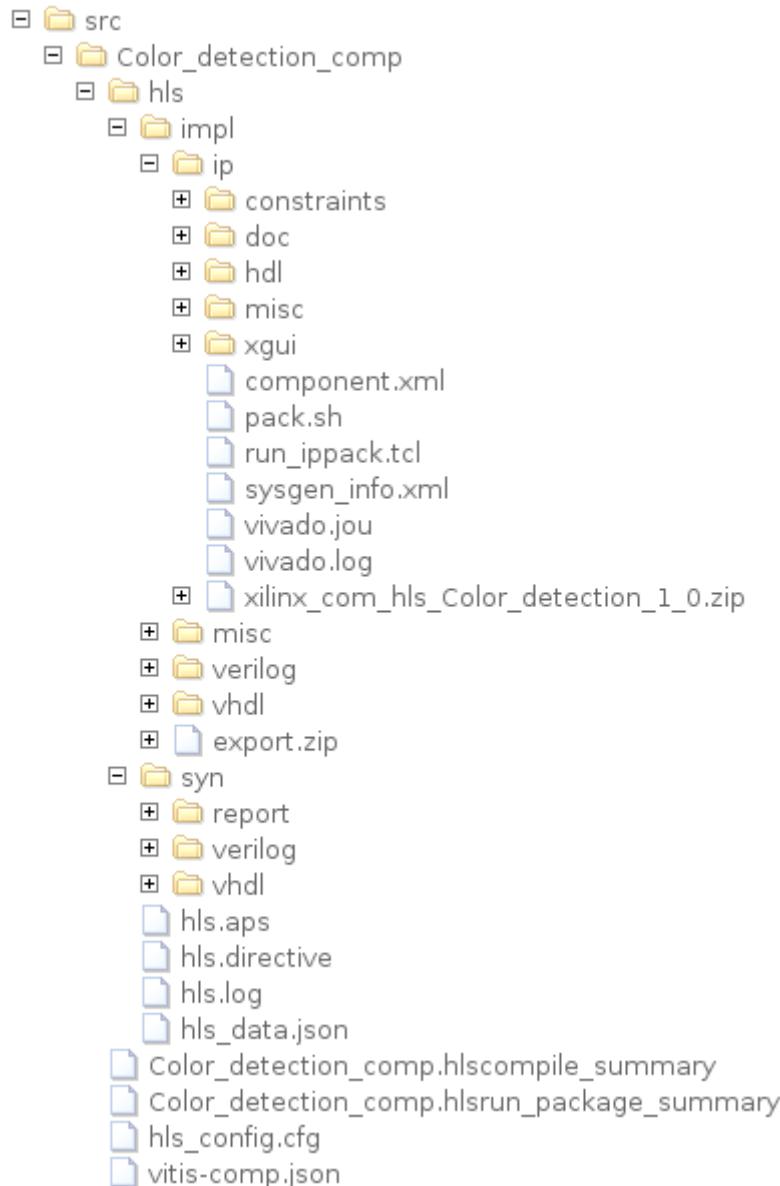
Model Composer displays a transcript window of the process.

When Vitis Model Composer has completed synthesizing the RTL, it launches Vivado to create and package the IP for the subsystem design.

Model Composer generates the following outputs from the algorithm:

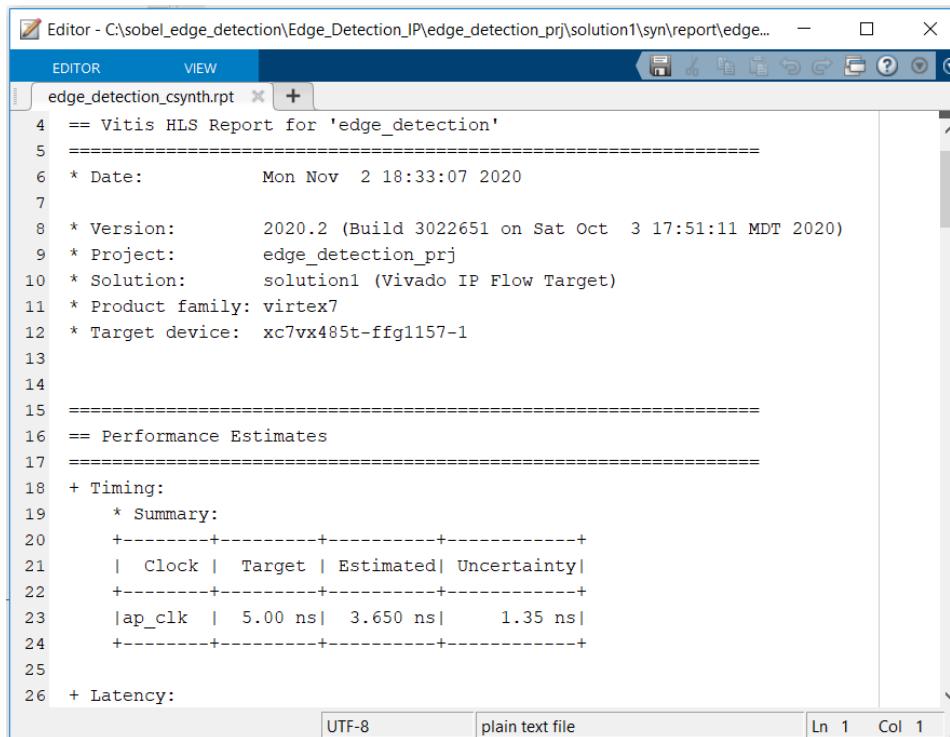
- SystemC (IEEE 1666-2006, version 2.2)
- VHDL (IEEE 1076-2000)
- Verilog (IEEE 1364-2001)
- Report files created during synthesis, C/RTL co-simulation, and IP packaging.

When Vitis Model Composer has completed generating the packaged IP, it can be found in the project directory structure as shown in the following figure. The `src` folder is contained in the `Export Directory` specified by the Model Composer Hub. The `Color_detection_comp` folder is a Vitis HLS component. For more information refer to the *Vitis High-Level Synthesis User Guide* ([UG1399](#)). The `syn` and `impl` folders store the results of synthesis and implementation. The `ip` folder contains the packaged IP to add to the Vivado Design Suite IP catalog.

**Figure 152: Packaged IP Folder**

After Model Composer has generated the packaged IP, the `.zip` file archive in the `<component_name>/hls/impl/ip` folder can be imported into the Vivado IP catalog, and used in any Vivado Design Suite design, either as RTL IP, or in the IP integrator.

The Synthesis Report can be found in the generated file `<component_name>/hls/synth/reports/<component_name>_csynth.rpt`. The Synthesis Report includes details on the estimated performance and resource utilization of the RTL design synthesized by Model Composer. You can review this report to see the estimates and review your model.

**Figure 153: Synthesis Report**

```
Editor - C:\sobel_edge_detection\Edge_Detection_IP\edge_detection_prj\solution1\syn\report\edge...
EDITOR      VIEW
edge_detection_csynth.rpt  [+]
4 == Vitis HLS Report for 'edge_detection'
5 =====
6 * Date:      Mon Nov  2 18:33:07 2020
7
8 * Version:   2020.2 (Build 3022651 on Sat Oct  3 17:51:11 MDT 2020)
9 * Project:   edge_detection_prj
10 * Solution: solution1 (Vivado IP Flow Target)
11 * Product family: virtex7
12 * Target device: xc7vx485t-ffg1157-1
13
14
15 =====
16 == Performance Estimates
17 =====
18 + Timing:
19     * Summary:
20     +-----+-----+-----+
21     | Clock | Target | Estimated| Uncertainty|
22     +-----+-----+-----+
23     |ap_clk | 5.00 ns| 3.650 ns|    1.35 ns|
24     +-----+-----+-----+
25
26 + Latency:
```

The screenshot shows a text editor window titled "Editor - C:\sobel\_edge\_detection\Edge\_Detection\_IP\edge\_detection\_prj\solution1\syn\report\edge...". The file is named "edge\_detection\_csynth.rpt". The content of the report is as follows:

```
== Vitis HLS Report for 'edge_detection'
=====
* Date:      Mon Nov  2 18:33:07 2020
*
* Version:   2020.2 (Build 3022651 on Sat Oct  3 17:51:11 MDT 2020)
* Project:   edge_detection_prj
* Solution: solution1 (Vivado IP Flow Target)
* Product family: virtex7
* Target device: xc7vx485t-ffg1157-1

=====
== Performance Estimates
=====

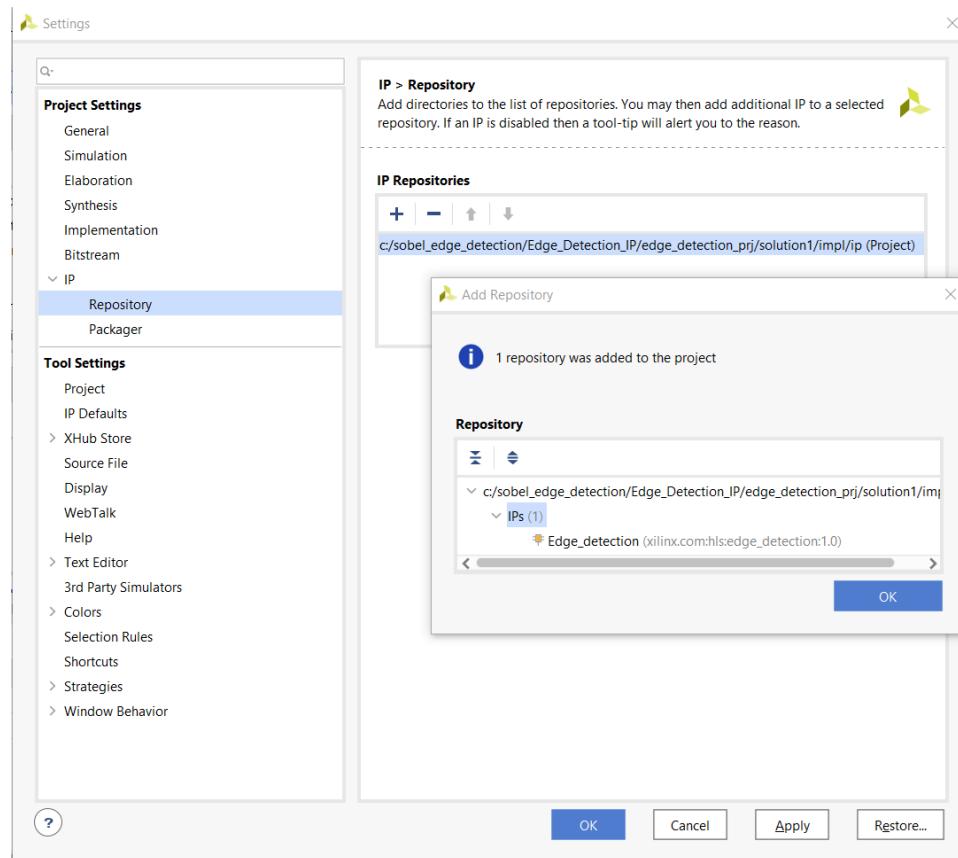
+ Timing:
    * Summary:
    +-----+-----+-----+
    | Clock | Target | Estimated| Uncertainty|
    +-----+-----+-----+
    |ap_clk | 5.00 ns| 3.650 ns|    1.35 ns|
    +-----+-----+-----+
```

The report includes sections for project details, performance estimates, and timing tables.

For Model Composer models that specify AXI4-Lite Slave interfaces through the Interface Specification block, as discussed in [Defining the Interface Specification](#), a set of software driver files is also created by Vitis HLS during the IP packaging process. These C driver files can be included in an SDK C project and used to access the AXI4-Lite Slave slave port. The software driver files are written to directory <component\_name>/hls/impl/ip/drivers and are included in the packaged IP.

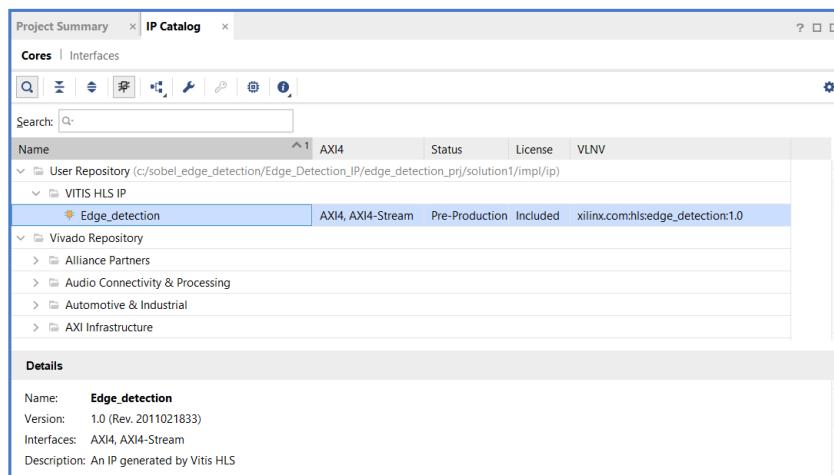
To add the IP into the Vivado IP catalog, from within the Vivado Integrated Design Environment (IDE), select **Tools**→**Settings** to open the Settings dialog box. Select the **IP**→**Repository** command, and add the Vitis HLS packaged IP as shown in the following figure.

Figure 154: Setting the IP Repository



After adding the path to the repository, the IP is added to the IP catalog as shown in the following figure. You can now use the IP in standard RTL designs, or in Vivado IP integrator block designs. For more information on working with IP and adding to the IP repository refer to the [Vivado Design Suite User Guide: Designing with IP \(UG896\)](#).

Figure 155: IP Catalog



---

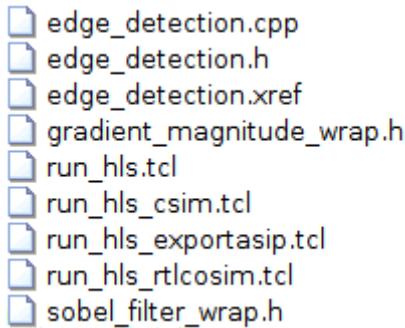
**IMPORTANT!** If you see the repository added to the IP catalog, but do not see the Vitis HLS packaged IP, it is possible the target part for the current project is not compatible with the device used when generating the Model Composer output. You can fix this by changing the part in the current project to the device specified by the Model Composer model.

---

## Generating C++ Code

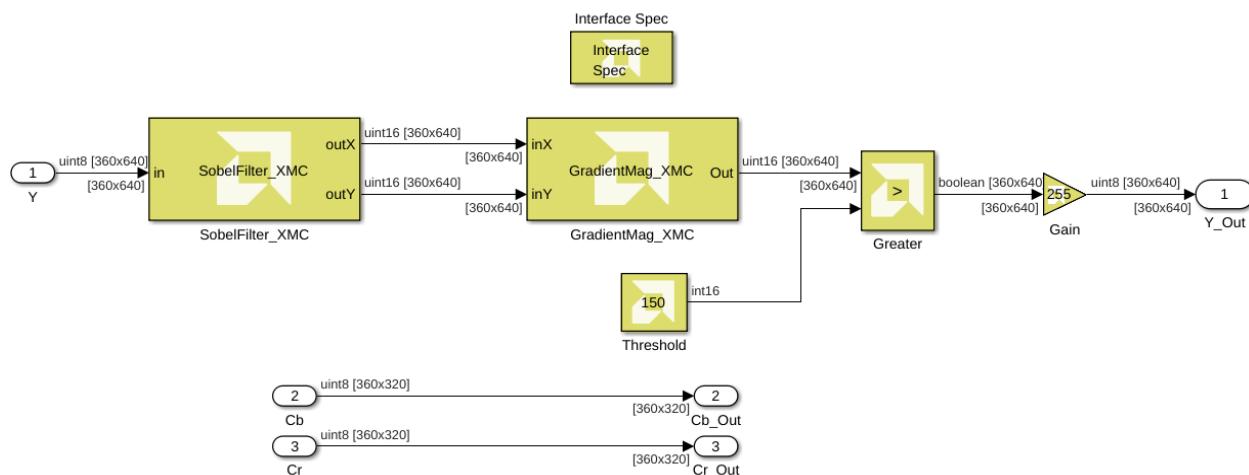
The following figure shows the HLS C++ Code output by Vitis Model Composer from the Export command. The C++ code is output either as an intermediate step when exporting a packaged IP for Vivado output, or as a specified Export Type to let you optimize the C++ code using directives or pragmas in AMD Vitis™ HLS.

Figure 156: C++ Output Files



The files generated by Model Composer reflect the contents and hierarchy of the subsystem that was compiled. In this case, the subsystem is the Edge Detection function described in the HLS section of the [Vitis Model Composer Tutorials](#). The following figure shows the contents of the Edge Detection subsystem.

Figure 157: Edge Detection Subsystem



The Edge\_Detection.cpp file specifies the following include files, which incorporate the code generated for the various Model Composer blocks used in the subsystem:

```
#include "Edge_Detection.h"
#include "GradMagnitude.h"
#include "SobelFilter.h"
```

The following shows the generated code for the Edge Detection subsystem. Notice the pragmas added to the function to specify the function protocol and the I/O port protocols for the function signature and return value. The pragmas help direct the component synthesized by Vitis HLS, and result in higher performance in the implemented RTL.

```
Edge_Detection(hls::stream< ap_axiu<16, 1, 1, 1> >& Y,
               hls::stream< ap_axiu<16, 1, 1, 1> >& Y_Out)
{
    #pragma HLS INTERFACE s_axilite port=return
    #pragma HLS INTERFACE axis bundle=image_out port=Y_Out
    #pragma HLS INTERFACE axis bundle=input_vid port=Y
    #pragma HLS dataflow
    uint8_t core_Y[360][640];
    #pragma HLS stream variable=core_Y dim=2 depth=1
    uint8_t core_Cb[360][320];
    #pragma HLS stream variable=core_Cb dim=2 depth=1
    uint8_t core_Cr[360][320];
    #pragma HLS stream variable=core_Cr dim=2 depth=1
    uint8_t core_Y_Out[360][640];
    #pragma HLS stream variable=core_Y_Out dim=2 depth=1
    uint8_t core_Cb_Out[360][320];
    #pragma HLS stream variable=core_Cb_Out dim=2 depth=1
    uint8_t core_Cr_Out[360][320];
    #pragma HLS stream variable=core_Cr_Out dim=2 depth=1
    fourier::AxiVideoStreamAdapter< uint8_t >::readStreamVf0(Y,
        reinterpret_cast< uint8_t*>(core_Y), reinterpret_cast< uint8_t*>(core_Cb),
        reinterpret_cast< uint8_t*>(core_Cr), 360, 640);
    Edge_Detection_core(core_Y, core_Cb, core_Cr, core_Y_Out, core_Cb_Out,
        core_Cr_Out);
    fourier::AxiVideoStreamAdapter< uint8_t >::writeStreamVf0(Y_Out,
        reinterpret_cast< uint8_t*>(core_Y_Out), reinterpret_cast<
        uint8_t*>(core_Cb_Out), reinterpret_cast< uint8_t*>(core_Cr_Out), 360, 640);
}
```

Finally, notice the various Tcl scripts generated in the output folder. They can be used to automate various packaging, synthesis, and simulation steps in Vitis:

- `run_hls.tcl`: This is a Tcl script that can be used to run Vitis HLS on the generated output files to create a Vitis HLS project and component. Each Vitis HLS project holds one set of C/C++ code and can contain multiple components. Each component can have different constraints and optimization directives. For more information refer to the *Vitis High-Level Synthesis User Guide* ([UG1399](#)).
- `run_hls_csim.tcl`: Same as `run_hls.tcl`, but also runs C simulation of the component.
- `run_hls_rtlcosim.tcl`: Same as `run_hls_csim.tcl`, but also synthesizes RTL code from the C++ code and runs RTL simulation of the component.

- `run_hls_exportasip.tcl`: Same as `run_hls_rtlcosim.tcl`, but also generates an IP from the component.

You can run the Tcl scripts from an AMD Vitis™ HLS command prompt as follows:

1. Open the AMD Vitis™ HLS Command Prompt:
  - On Windows, click **Start** → **All Programs** → **Xilinx Design Tools** → **Vitis HLS 2025.1** → **Vitis HLS** → **Vitis HLS 2025.1 Command Prompt**.
  - On Linux, open a new shell and source the `<install_dir>/<version>/Vitis/settings64.sh` script to configure the shell.
2. From the command prompt, change the directory to the parent folder of the Export Directory specified on the Model Composer Hub dialog box when you generated the output, as discussed at [Vitis Model Composer Hub](#).
3. Further change directory into the `src` folder containing the Tcl scripts. For example:

```
cd ip/edge_detection/src
```

4. From the command prompt, launch the `run_hls.tcl` script:

```
vitis-run --mode hls --tcl ./run_hls.tcl
```

AMD Vitis™ HLS launches to synthesize the RTL from the C++ code, generating an AMD Vitis™ HLS project, and component in the process.

## Model Composer Log File

To help you diagnose issues related to code generation, Vitis Model Composer generates a log file, `model_composer.log`, that is written to the Target or Export Directory specified on the Vitis Model Composer Hub block.

By default, Model Composer generates code for the model using a streaming micro-architecture in which blocks run concurrently via task pipelining, or dataflow. However, in some cases this streaming micro-architecture is not achievable, because the model includes an `xmc ImportFunction` block that does not support streaming for example. In this case, Model Composer generates code in which the blocks operate sequentially. The Model Composer log file includes information to help you identify when this condition occurs, and what some possible causes might be.

The log file also contains information related to which blocks are used in the Model Composer model.

If Export Type is set to IP Catalog, information related to running AMD Vitis™ HLS is provided. In these cases, more detailed information can be found in the `vitis_hls.log` file which can also be found in the directory specified in <Export Directory>/ip/<Subsystem Name>/src.

---

# Simulating and Verifying Your Design

## Introduction

Verification in Vitis Model Composer can be separated into two distinct processes:

- Verification of the algorithm in Simulink® to verify the functional correctness of the design.
- Verification of the Model Composer model, to confirm the equivalence of the simulation results in Simulink and the C++ and RTL outputs.

In high-level synthesis, running the compiled C program is referred to as C simulation. Executing the C++ algorithm generated by Model Composer simulates the function and verifies that the output from the code matches the output from the Simulink simulation to validate that the algorithm is functionally correct.

In C/RTL co-simulation, Vitis HLS uses the C test bench to simulate the C function prior to synthesis and to verify the RTL output. The verification process consists of three phases:

1. The C simulation is executed and the inputs to the top-level subsystem are saved as “input vectors.”
2. The input vectors are used in behavioral simulation of the RTL code for the top-level subsystem created by Vitis HLS. The outputs from the RTL are saved as “output vectors.”
3. The output vectors are applied to the C test bench as output from the top-level subsystem, to verify the results of the C-simulation match.

Vitis HLS uses this return value for both C simulation and C/RTL co-simulation to determine if the results are correct.

## Simulating in Simulink

Simulink can interactively simulate your model, and view the results on scopes and graphical displays. The Simulink model defines what data to input at the start of simulation, and defines what data to capture during simulation.

When defining the model, you define the input and output signals for the model. Input signals load data into the model for simulation, while output signals allow you to record simulation results. The kind of data you want to load impacts your choice of signal loading techniques. You can define input data as constant values, use source blocks, such as the Sine Wave block, import data from a spreadsheet, or use the output of a previous simulation. Refer to [Prepare Model Inputs and Outputs](#) in the Simulink documentation for more information on preparing for simulation in Simulink.

After the simulation is completed, you can analyze any logged data with MATLAB scripts and visualization tools like the Simulation Data Inspector within the MathWorks environment.

When you click the **Analyze** button in the [Vitis Model Composer Hub](#) block, Model Composer performs two tasks:

1. Automatically logs the test data, or stimulus, at the input of your design, and the simulation results as test vectors for later use as "golden" data for comparison during C/C++ and RTL co-simulation. This file is named `signals.stim`, and is added to the specified `Code Directory` when generating output.
2. Executes the generated test bench and verifies equivalence of the code using C-simulation and C/RTL co-simulation. This process is compute intensive, and can take considerable time.

 **IMPORTANT!** If the model simulation time in Simulink is long, with significant amounts of data processed, the test bench execution will take an even longer time. Model Composer will generate an error if the simulation time becomes infinite.

## Managing the HLS Block Cache

Vitis Model Composer creates a cache entry when you simulate a model with an imported block created using `xmcImportFunction`. When you simulate the model again, unless you make any changes to the function, or supporting source files, Model Composer will use the cached entry for the block to initiate the simulation faster. You can manage the simulation cache in Model Composer using the following command from the MATLAB command prompt:

```
>> xmcSimCache
```

The usage for this command is as follows.

*Table 21: xmcSimCache Command Usage*

Command	Description
<code>xmcSimCache('enable')</code>	Enable the simulation cache. This is enabled by default.
<code>xmcSimCache('disable')</code>	Disable the simulation cache.
<code>xmcSimCache('isEnabled')</code>	Returns the state of the simulation cache.
<code>xmcSimCache('setLocation', &lt;dir&gt;)</code>	Specify a directory to use for the simulation cache. For example:  <code>xmcSimCache('setLocation', 'C:/temp')</code>
<code>xmcSimCache('setDefaultLocation')</code>	Restore the simulation cache to its default location.
<code>xmcSimCache('getLocation')</code>	Return the current location of the simulation cache.
<code>xmcSimCache('clear')</code>	Clear the entire simulation cache.

**Table 21: xmcSimCache Command Usage (cont'd)**

<b>Command</b>	<b>Description</b>
<code>xmcSimCache('clear', 'release', &lt;version&gt;)</code>	Clear simulation cache entries for the specified release version. For example:  <code>xmcSimCache('clear', 'release', '2017.4')</code>
<code>xmcSimCache('clear', 'days', &lt;number&gt;)</code>	Clear simulation cache entries older than or equal to the specified number of days. For example:  <code>xmcSimCache('clear', 'days', 30)</code>
<code>xmcSimCache('clear', 'id', &lt;vals&gt;)</code>	Clear simulation cache entries by the cache ID. For example:  <code>xmcSimCache('clear', 'id', {'12345678', 'abcdefgh'})</code>

## Verifying the C++ Code

When generating the output using the Vitis Model Composer Hub block, it generates a `makefile`, the test bench, `tb.cpp`, and `signals.stim` as previously discussed. The purpose of the test bench is to apply input stimuli, generated during Simulink simulation, to the top-level function of the generated C++ or RTL code and compare that function's output against the output samples captured in the `signals.stim` file. Depending on the output generated, the verification flow runs simulation on the C++ or RTL outputs generated by Model Composer and looks for the same result as generated by Simulink.

When the Export Type on the Model Composer Hub block is HLS C++ code, the verification flow is as follows:

- The model is simulated in Simulink and the input and outputs are logged into the `signals.stim` binary file.
- Model Composer generates the C++ code and a test bench, `tb.cpp`, which contains a `main()` function.
- Model Composer launches simulation.
- It verifies that the output from the generated C++ code matches the output logged from the Simulink simulation, `signals.stim`.
- In case of a mismatch, the mismatched output signal name is reported, as well as the actual and expected values.
- The result is a Pass/Fail returned by Model Composer.

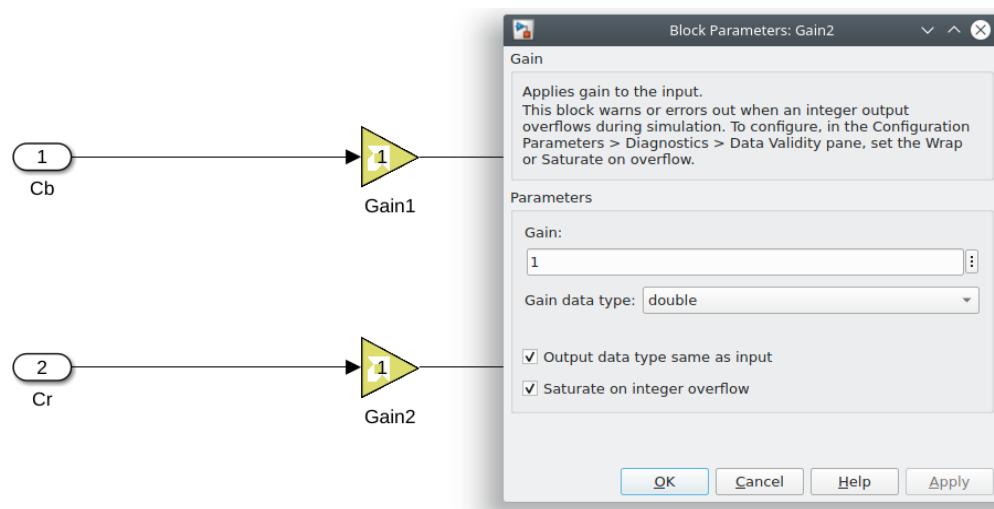


**IMPORTANT!** During simulation by Model Composer, you might receive the following error message:

Failed to find a XilinxLibrary block connected to input port.

This error simply means that there is an input in the subsystem that does not connect to a block from the HLS Library block set. This might be due to the presence of signals that you are simply passing through the subsystem for signal grouping, or improved readability. The recommended fix for this conditions is to connect the input to a Gain block from the **AMD Toolbox** → **HLS** → **Math Functions** → **Math Operations** block library, with the default value of 1, and Output data type same as input checked, as shown in the following figure.

Figure 158: Adding Gain to Unconnected Inputs



## Verifying the C/RTL Code

When the Export Type specified on the Vitis Model Composer Hub is IP Catalog, Model Composer uses C/RTL co-simulation to verify the RTL output. Again, the objective is to verify that the results of the RTL simulation match the results of the Simulink simulation. In this case, the verification flow is as follows:

- The Model Composer model is simulated in Simulink to capture the test vectors in `signals.stim`.
- Model Composer generates the C/C++ code and the C test bench, `tb.cpp`.
- Model Composer runs the C-synthesis and generates the RTL output.
- Model Composer runs the C/RTL co-simulation. This step ensures the following:
  - That the C++ code generated by Model Composer is correct by comparing with the Simulink simulation, `signals.stim`.
  - That the RTL code generated by Model Composer is correct by comparing the output stimulus from RTL with C/C++ output.



**TIP:** After verification, Model Composer packages the IP for use in Vivado.

- The result is a separate Pass/Fail returned by Model Composer for both the C-simulation and the C/RTL co-simulation. If the C-simulation fails, the process stops before the C/RTL simulation is run.
- 

# Select Target Device or Board

## Device Chooser Dialog Box

Choose the default part or platform board to use for the current project. The device resources that the design is synthesized against, and placed onto are determined by selecting the target part, board, or platform.

- **Parts tab:** Lists the available target parts for the current project.
- **Boards tab:** Lists the available target boards for the current project.
- **Platform tab:** Allows you to specify the path to an AMD Platform File (.xpfm).



**IMPORTANT!** The devices and boards that are available are determined at the time the Vitis Model Composer tool is installed. You can also add uninstalled devices or boards. To learn more refer to AMD Answer Record [60112](#).

The selection of target part or boards can be limited or filtered by specifying search patterns in one or more of the available search fields at the top of the Device Chooser dialog box.

For parts:

- **Category:** Choose devices according to Military, Automobile, or Commercial grade products.
- **Family:** Filter devices according to the available device families (such as Virtex, Kintex, or UltraScale).
- **Package:** Specify the type of package the device will have.
- **Speed:** Filter the device by a specific speed grade.
- **Temperature:** Filter the device by a specific temperature.

For boards:

- **Vendor:** Choose devices according to the available vendors of platform boards.
- **Name:** Filter devices according to the available display names.
- **Board Rev:** Specify the revision level of the board.

The target parts or boards that match the specified filters and/or search string appear in a table of results in the lower portion of the dialog box.

Choose a target part or board for the design, and click **OK**.

# AI Engine Library

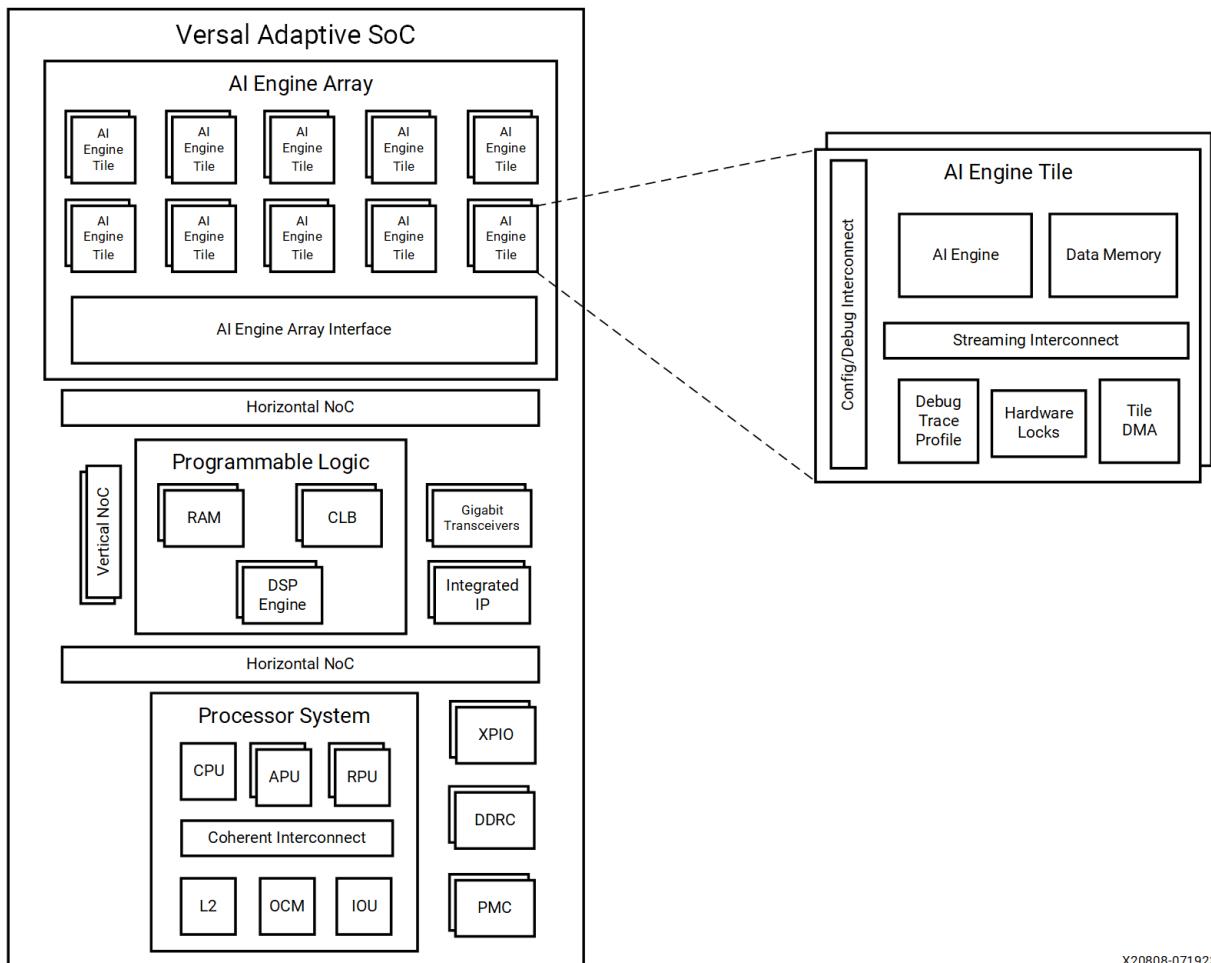
---

## Introduction

AMD Versal™ adaptive SoC devices are the industry's first adaptive compute acceleration platforms, combining adaptable processing and acceleration engines with programmable logic and configurable connectivity to enable customized, heterogeneous hardware solutions for a wide variety of applications in Data Center, Automotive, 5G Wireless, Wired, and Defense. Versal Adaptive SoCs provide transformational features like an integrated silicon host interconnect shell and AI Engines, programmable logic (PL), and a processor subsystem (PS), providing superior performance/watt over conventional FPGAs, CPUs, and GPUs.

Versal devices are built from a library of building blocks dedicated to processing, compute, acceleration, and connectivity. The following figure shows a top-level view of the Versal adaptive SoC, that contains three major architectural areas: the Arm® processing system, the programmable logic, and the AI Engine array. They form a tightly integrated heterogeneous compute platform.

Figure 159: AMD Versal Adaptive SoC Overview



## AI Engines

An AI Engine is an array of very-long instruction word (VLIW) processors with single instruction multiple data (SIMD) vector units that are highly optimized for compute-intensive applications, specifically digital signal processing (DSP), 5G wireless applications, and artificial intelligence (AI) technology such as machine learning (ML). They provide up to five times higher compute density for vector-based algorithms.

AI Engines provide multiple levels of parallelism including instruction-level and data-level parallelism:

- Instruction-level parallelism includes a scalar operation, up to two moves, two vector reads (loads), one vector write (store), and one vector instruction that can be executed—in total, a 7-way VLIW instruction per clock cycle.
- Data-level parallelism is achieved via vector-level operations where multiple sets of data can be operated on a per-clock-cycle basis.

Each AI Engine contains both a vector and scalar processor, dedicated program memory, local 32 KB data memory, and access to local memory in any of four neighboring directions (north, south, east, or west). It also has access to DMA engines and AXI4 interconnect switches to communicate via streams to other AI Engines or to the programmable logic (PL) or the DMA.

Some Versal adaptive SoCs include the AI Engine-ML (AIE-ML) that consists of an array of AIE-ML tiles, AIE-ML memory tiles, and the AIE-ML array interface consisting of the network on chip (NoC) and programmable logic (PL) tiles. Each AIE-ML tile integrates a very-long instruction word (VLIW) processor, integrated memory, and interconnects for streaming, configuration, and debug. The AIE-ML array introduced a separate functional block, the memory tile, that is used to significantly reduce PL resources (LUTs and URAMs) for ML applications.

## Programmable Logic and Processor Subsystem

Versal's programmable logic blocks and memory are architected for flexible custom-compute and data movement. The Arm Cortex®-A72 and Cortex®-R5F processors allow for complex control processing tasks.

Refer to the *Versal Adaptive SoC AI Engine Architecture Manual* ([AM009](#)) and the *Versal Adaptive SoC AIE-ML Architecture Manual* ([AM020](#)) for specific details on the AI Engine array and interfaces of each architecture.

Refer to the *Versal Architecture and Product Data Sheet: Overview* ([DS950](#)) for specific details on Compute and Acceleration Engines.

## AI Engine Kernels

An AI Engine kernel is a C/C++ program written using AI Engine vector data types and specialized intrinsic calls that target the VLIW vector processor. They are computational functions running on an AI Engine. These kernels form the fundamental building blocks of an AI Engine program which consists of dataflow graph specification. The AI Engine kernel code is compiled using the `aiecompiler` included in the AMD Vitis™ software platform core development kit. The `aiecompiler` compiles the kernels to produce an ELF file that runs on the AI Engine processors.

## AI Engine Graphs

An AI Engine program consists of a dataflow graph specification written in C++. This specification can be compiled and executed using the `aiecompiler`. A static dataflow (SDF) graph application consists of nodes and edges where nodes represent compute kernel functions and edges represent data connections. Kernels in the application can be compiled to run on the AI Engine or in the PL region of the device.

# Vitis Model Composer for AI Engine Development

AMD Vitis™ Model Composer enables the rapid simulation, exploration, and code generation of algorithms targeted for AI Engines from within the Simulink® environment. You can achieve this by importing AI Engines kernels and data-flow graphs into Model Composer as blocks and controlling the behavior of the kernels and graphs by configuring the block GUI parameters. Simulation results can be visualized by seamlessly connecting Simulink source and sink blocks with the Model Composer AI Engines blocks. Furthermore, the simulation results can be sent to the MATLAB® workspace for further analysis.

Refer to [Creating an AI Engine Design using Vitis Model Composer](#) for more information on importing AI Engine kernels and graphs as blocks.

Vitis Model Composer provides a set of AI Engine library blocks for use within the Simulink environment. These include:

- Blocks that support importing AI Engine kernel or graph code into Simulink.
- Blocks that support connection between the AI Engine and the AMD HDL blockset.
- Configurable AI Engine blocks for DSP functions such as FIRs, FFTs, Mixers, and DDS.

**Note:** For more information on specific blocks refer to the AMD Vitis™ Model Composer AI Engine Block library.

Connecting HLS kernel blocks, HDL library blocks, and AI Engine blocks, allows modeling and simulation of a heterogeneous platform which can be targeted to both programmable logic and AI Engines in AMD Versal™ Adaptive SoC devices.

Vitis Model Composer supports targeting both AIE and AIE-ML devices. Vitis Model Composer supports features specific to the AIE-ML architecture, including memory tile shared buffers and the `bfloat16` data type.

In addition to simulation, you can also use the AI Engine Model Composer Hub to generate dataflow graphs. For more details on the AI Engine Model Composer Hub block, specific to AI Engine code generation, refer to [Code Generation](#).

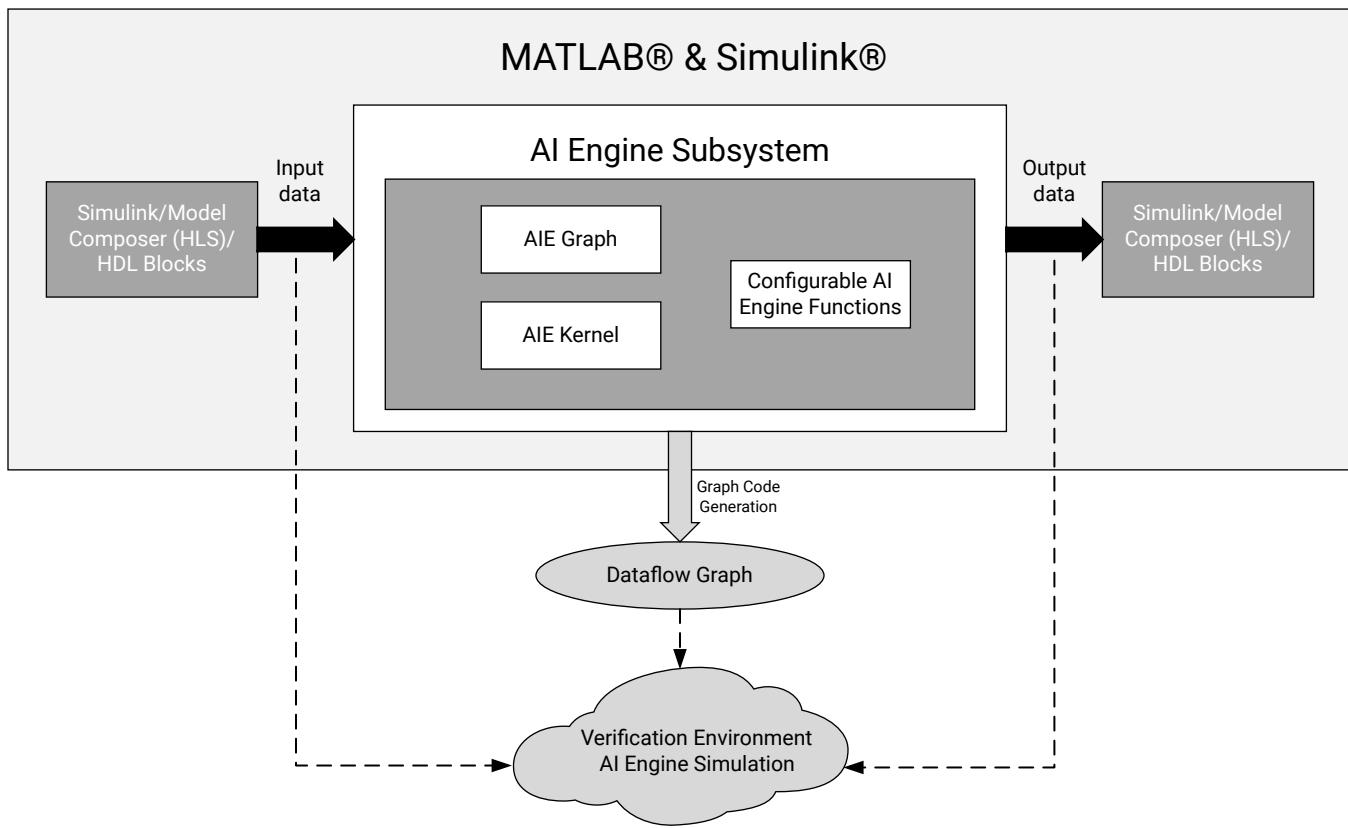
Vitis Model Composer allows you to verify the generated dataflow graph code using the AI Engine simulator. Based on verification requirements, you can choose to verify your algorithm from the Hub block.

The simulation results are compared against the reference design in the Simulink environment.

**Note:** Refer to [Simulation and Code Generation](#) for more details on simulator options and verification.

A typical AI Engine design flow is shown in the following diagram.

Figure 160: Typical AI Engine Design Flow



X25368-052521

To learn more about the AI Engine flow in Model Composer, refer to the [Vitis Model Composer Tutorials](#).

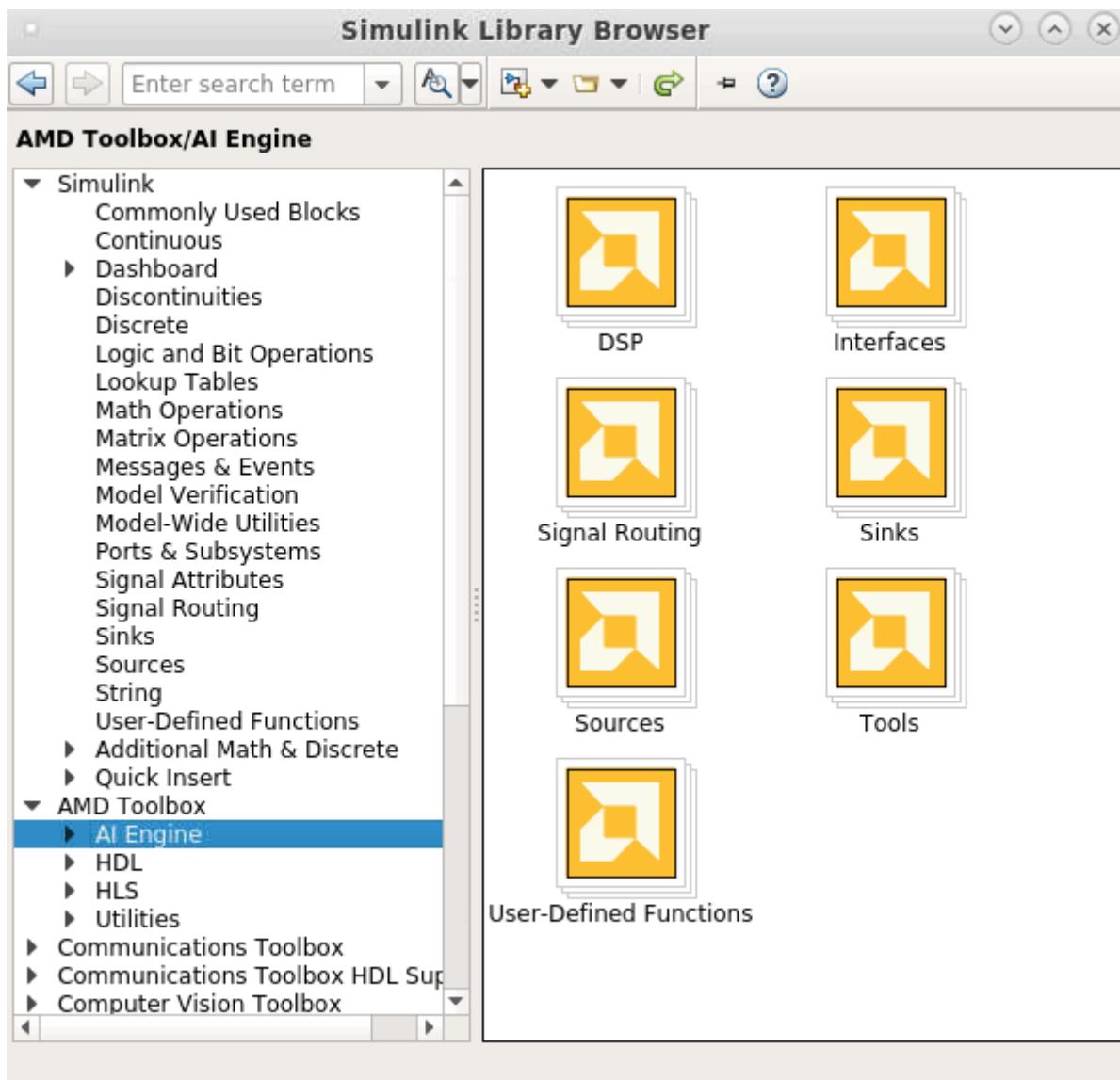
The remainder of this section discusses the following topics:

- [Creating an AI Engine Design using Vitis Model Composer](#)
- [Simulation and Code Generation](#)
- [Verification of AI Engine Code](#)

# Creating an AI Engine Design using Vitis Model Composer

As previously discussed, AI Engine kernels are functions that form the fundamental building blocks of the data-flow graph. Model Composer supports generating the AI Engine data-flow graph by importing the AI Engine kernel or sub-graph. The AI Engine library is available under AMD Toolbox in the Simulink library browser set as shown in the following figure.

Figure 161: Simulink Library Browser



## Preparing the Kernels

Kernels are declared as C/C++ functions that return void and can use special data types for arguments which are discussed in [Data Accessing Mechanisms](#). The kernels should be defined each in their own source file. This organization is recommended for reusability and faster compilation. Furthermore, the kernel source files should include all relevant header files to allow for independent compilation.



**IMPORTANT!** *It is assumed that your kernel code is already developed. If you try to import bad kernel code, for example one that causes memory corruption, you might see undesirable outcomes including MATLAB crashing.*

The topics in this section introduce some basics of AI Engine programming which are necessary to understand the Vitis Model Composer AI Engine design flow. Refer to [AI Engine Tools and Flows User Guide \(UG1076\)](#) for a high-level overview of the kernel and AI Engine programming models. Some insight is also provided on data access APIs to help you understand the configuration parameters of the AI Engine kernel blocks available in the library.

## Data Accessing Mechanisms

An AI Engine kernel can either consume or produce blocks of data, or, it can access and produce data streams in a sample-by-sample fashion. The data access APIs for both cases are described in the following sections.

### Buffer-Based Access

From the kernel perspective, an incoming block of data is called an input buffer. Input buffers are defined by the type of data contained within that buffer. The following example shows a declaration of an input buffer carrying complex integers where the real and imaginary parts are both 16 bits wide.

```
input_buffer<cint16> myInputBuffer;
```

From the kernel perspective, an outgoing block of data is called an output buffer. Again, these are defined by type. The following example shows a declaration of an output window carrying 32-bit integers.

```
output_buffer<int32> myOutputBuffer;
```

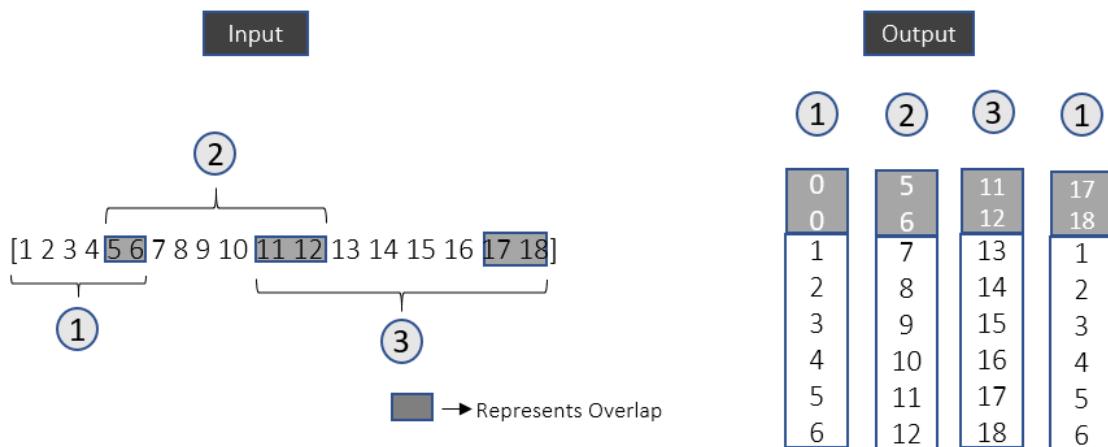
A kernel reads from its input buffers and writes to its output buffers. By default, the synchronization required to wait for an input buffer of data or provide an empty output buffer is performed before entering the kernel. There is no synchronization required within the kernel to read or write the individual elements of data. In other words, the kernel will not execute unless there is a full buffer available.

In some situations, if you are not consuming a buffer's worth of data on every invocation of a kernel, or if you are not producing a buffer's worth of data on every invocation, you can control the buffer synchronization by configuring the kernel port to be `async` in the Block Parameters dialog box of the kernel.

It is also possible to have overlap from one block of input to the next. This in general is required for certain algorithms such as filters. This overlap is referred to as 'Buffer Margin'. If a Buffer margin is specified, the kernel has access to a total number of samples equal to `buffer_size + margin_size`.

The behavior of the buffer margin can be demonstrated using the following example.

*Figure 162: Simulation*



Here, input is a vector of size 8 and this is fed to the kernel block which is configured to have a buffer size of 6 and a buffer margin of 2, as shown in the previous figure. The kernel should have access to a total of 8 samples at every invocation. During the first simulation cycle, two 0's are prepended to the first 6 new values from the input data. For the subsequent simulation cycles, the kernel receives 8 values which includes 6 new values and 2 values from the previous cycle.

## Stream-Based Access

Kernels can access data streams in a sample-by-sample fashion using data access APIs. With a stream-based access model, kernels receive an input stream or an output stream of typed data as an argument. Each access to these streams is synchronized (that is, reads stall if the data is not available in the stream and writes stall if the stream is unable to accept new data).

The following example shows a declaration of input and output streams of type `cint16`.

```
input_stream<cint16> & myInputStream;
output_stream<cint16> & myOutputStream;
```

There is also a direct stream communication channel between one AI Engine and the physically adjacent AI Engine, called a *cascade*. The cascade stream is connected within the AI Engine array in a snake-like linear fashion from AI Engine processor to processor.

The following example shows a declaration of input and output cascade streams of type `cacc48`.

```
input_cascade<cacc48> & myInputCascade;  
output_cascade<cacc48> & myOutputCascade;
```

## Runtime Parameter Specification

It is possible to modify the behavior of the AI Engine program or data-flow graph based on a dynamic condition or event using the runtime parameter. The modification could be in the data being processed, for example a modified mode of operation or a new coefficient table, or it could be in the control flow of the graph such as conditional execution or dynamically reconfiguring a graph. Either the kernels or the graphs can be defined to execute with parameters.

If an integer scalar value appears in the formal arguments of a kernel function, then that parameter becomes a runtime parameter. Runtime parameters are processed as ports alongside those created by streams and windows. Both scalar and array values can be passed as runtime parameters.

Consider the following example where a kernel function is defined with runtime parameters. Here, `select` is a scalar RTP port and `coefficients` is a vector RTP with 32 integers.

```
#ifndef FUNCTION_KERNELS_H  
#define FUNCTION_KERNELS_H  
  
void filter_with_array_param(input_buffer<cint16> & in,  
output_buffer<cint16> * out, int32 select, const int32 (&coefficients)[32]);  
  
#endif
```

Two types of RTPs are supported:

- **Synchronous Parameters (or triggering parameters):** The kernel does not execute until the runtime parameter is written by a controlling processor. Upon a write, the kernel executes once, reading the new updated value. After completion, it is blocked from executing again until the parameter is updated. This allows for a different type of execution model from the normal streaming model, and can be useful for certain updating operations where blocking synchronization is important.
- **Asynchronous Parameters:** These parameters can be changed any time by a controlling processor such as Arm®. They are read each time a kernel is invoked without any specific synchronization. These types of parameters can be used, for example, to pass new filter coefficients to a filter kernel that changes infrequently.

## Importing AI Engine Code as a Block

Vitis Model Composer allows you import the AI Engine kernel from the AI Engine library. This allows you to create a block with an interface that has input and output ports equivalent to the arguments of an AI Engine kernel function, and also gives the flexibility to configure the kernel parameters using the Block Parameters dialog box. If you have a data-flow graph instead of an AI Engine kernel function, then Model Composer also allows you to import it into the Simulink environment from where it is possible to seamlessly connect the AI Engine graph block with AI Engine kernel block to build a complete system.

In summary, the entry point for using an AI Engine block set can be a kernel or a data flow sub-graph for which Vitis Model Composer generates a block with interfaces that match the function arguments of a kernel or a graph.

### Variable-Size Signals

In Simulink, a signal whose size (the number of elements in a dimension) can change during Simulink simulation is called a variable-size signal. To understand the importance of variable-size signaling in the context of modeling an AI Engine design in Model Composer, consider a kernel that outputs even numbers from a set of input numbers.

```
void even_calc(input_buffer<int32> & in, output_stream<int32> & out) {
    int32 val;
    auto pIn = aie::begin(in);

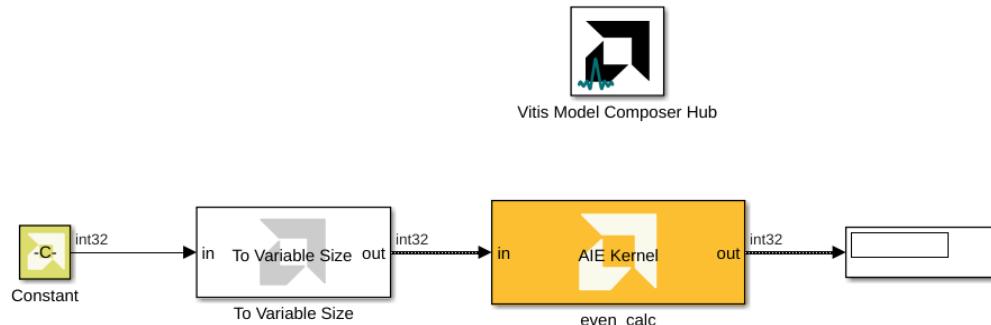
    for (unsigned i=0; i<4; i++) {
        val = *pIn++;
        if(val % 2 == 0) {

            writeincr(out, val);
        }
    }
}
```

Here, the input is a window of type `int32` and the output is a stream of similar type. If you try to model this in Simulink, you will observe that the number of data samples it produces might vary at each invocation.

Assume the input window size is 4 and the input given to the kernel is [1 -2 3 -4 5 -6 7 8 9 10 12 14 5 7 9 13].

Figure 163: Variable-Size Signals



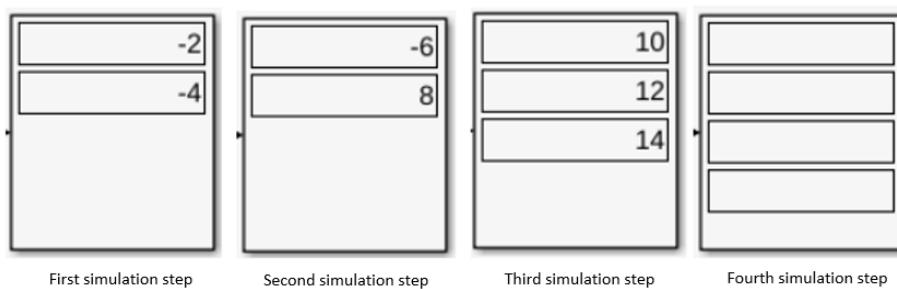
To understand why the size of the output can vary dynamically, run the simulations in steps.

During the first simulation step, the AI Engine kernel consumes four values (that is,  $1, -2, 3, -4$ ) and outputs two values ( $[-2, -4]$ ), which are the even numbers in the set  $[1, -2, 3, -4]$ . In the second simulation step, the kernel consumes the next set of window inputs (that is,  $[5, -6, 7, 8]$ ) the output is  $[-6, 8]$ . For the third set of inputs  $[9, 10, 12, 14]$ , the output at the third simulation step is  $[10, 12, 14]$ . Similarly for the final set of inputs  $[5, 7, 9, 13]$ , the output is empty because there are no even numbers to produce from the fourth input window.

In summary, the output size at the:

- First simulation step is 2.
- Second simulation step is 2.
- Third simulation step is 3.
- Fourth simulation step is 0.

Figure 164: Display Block - Simulation Steps



So, the kernel produces data samples of different sizes at every invocation and sometimes it does not produce any output. To model this behavior in Model Composer, you can use variable-size signals in Simulink. However, during a simulation, the number of dimensions cannot change. A variable-size signal always has an associated maximum signal size. In this case, it is 4.

The variable-size signal in Simulink is represented with a thicker signal line unlike the normal signal as highlighted in the previous figure. You can learn more about variable size signals at the following links:

- [MATLAB](#)
- [GitHub](#)

## ***Importing AI Engine Kernels***

Vitis Model Composer supports importing C/C++ kernels functions. Function must have `void` as the return type. It also supports importing class kernels as well as templated kernels. Model Composer provides two AI Engine library blocks to import kernel functions of different types (class-based and non-class-based kernels): AIE Kernel and AIE Class Kernel.

These are described in the following sections.

### **Non-Class-Based Kernels**

To Import a non-class-based AI Engine kernel as a block into Vitis Model Composer, you need to use an AI Engine Kernel block from the AI Engine Library as shown.

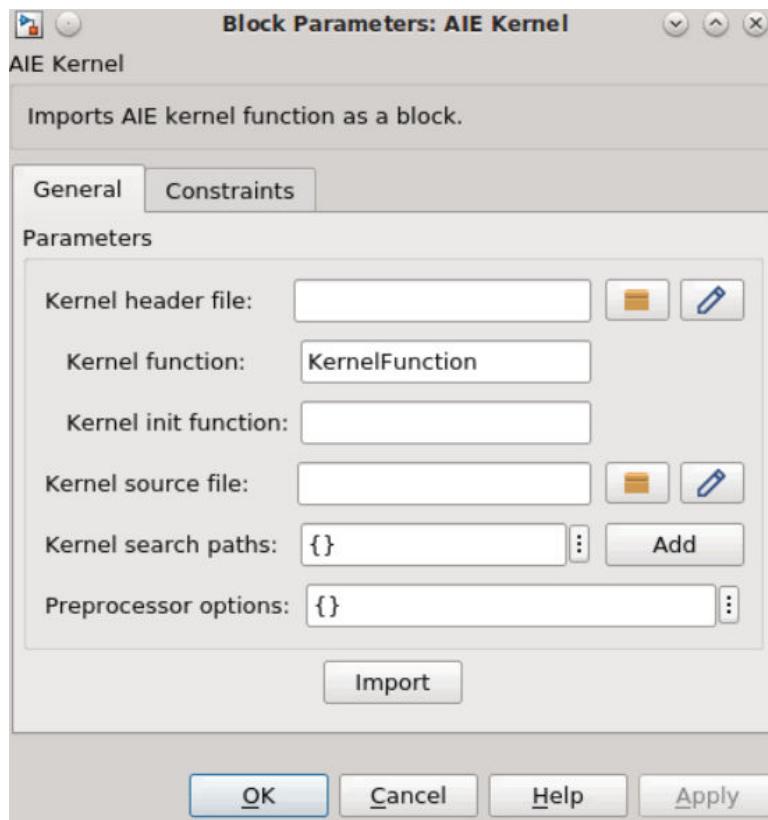
*Figure 165: AI Engine Kernel*



AIE Kernel

Double-clicking the block symbol displays the parameters of the AI Engine kernel block as shown in the following figure.

Figure 166: Block Parameters Dialog Box: AIE Kernel



The block mask parameters need to be updated to import the kernel function as a block. The following table provides details on the parameters and description for each parameter.

Table 22: AIE Kernel: Parameters

Parameter Name	Parameter Type	Criticality	Description
Kernel header file	String	Mandatory	Name of the header file that contains the kernel function declaration. The string could be just the file name, a relative path to the file, or an absolute path of the file. Use the Browse button to navigate to the file.
Kernel function	String	Mandatory	Name of the kernel function for which the block is to be created. This function should be declared in the kernel header file.
Kernel init function	String	Optional	Name of the initialization function used by the kernel function.
Kernel source file	String	Mandatory	Name of the source file that contains the kernel function definition. The string could be the file name, a relative path to the file or an absolute path of the file.
Kernel search paths	Vector of Strings	Optional	If the kernel header file or the kernel source file are not found using the value provided through Kernel header file or Kernel source file fields, respectively, then the paths provided through Kernel search paths are used to find the files.  This parameter allows the use of environment variables while specifying paths for the kernel header file and the kernel source file. The environment variable can be used in either \${ENV} or \$ENV format.

Table 22: AIE Kernel: Parameters (cont'd)

Parameter Name	Parameter Type	Criticality	Description
Preprocessor options		Optional	<p>Optional preprocessor arguments for downstream compilation with specific preprocessor options.</p> <p>The following two preprocessor option formats are accepted and multiple can be selected: <code>-D&lt;name&gt;</code> and <code>D&lt;name&gt;=&lt;definition&gt;</code> separated by a comma. That is, the optional argument must begin with <code>-D</code> and if the option <code>&lt;definition&gt;</code> value is not provided, it is assumed to be 1.</p>

The block parameter window which appears after double-clicking on the AI Engine kernel block is same irrespective of whether the kernel is Window type or Stream type. But the function configuration parameters changes for Window and Stream type. To edit the code in the MATLAB editor, click **Edit** (immediately after the browse button).

### Importing Buffer-Based Kernels

As explained in [Data Accessing Mechanisms](#), the size of the input and output data blocks for buffer-based access depends on the specified buffer size. Vitis Model Composer supports the following buffer-based input and output data types as interfaces to the AI Engine kernel block.

- `input_buffer<Type>`
- `output_buffer<Type>`

Table 23: Buffer-Based Input and Output Data Types

<Type>	Complexity	Signedness
int8	Real	Signed
int16	Real	Signed
int32	Real	Signed
int64	Real	Signed
uint8	Real	Unsigned
uint16	Real	Unsigned
uint32	Real	Unsigned
uint64	Real	Unsigned
cint16	Complex	Signed
cint32	Complex	Signed
float	Real	N/A
cfloat	Complex	N/A
bfloat16	Real	N/A

As an example, to import a simple kernel with a buffer-based interface, the following `simple.h` header file defines the `add_kernel` function with two input buffers and one output buffer of type `int16`.

## Simple.h

```
#ifndef __ADD_KERNEL_H__
#define __ADD_KERNEL_H__

#include <adf.h>
#define NUM_SAMPLES 4

using namespace adf;

void add_kernel(input_buffer<int16> & in1, input_buffer<int16> & in2,
output_buffer<int16> & outw);

#endif
```

The kernel (`simple.cc`) is defined as follows. It processes a `sum` operation on `in1` and `in2` and produces output on `outw`.

```
#include "simple.h"
void add_kernel(input_buffer<int16> & in1, input_buffer<int16> & in2,
output_buffer<int16> & outw)
{
    // Use scalar iterator to traverse data
    auto pIn1 = aie::begin(in1);
    auto pIn2 = aie::begin(in2);
    auto pOut = aie::begin(outw);

    for (unsigned i=0; i<NUM_SAMPLES; i++) {
        *pOut++ = *pIn1++ + *pIn2++;
    }
}
```



**TIP:** Although not required, the following recommendations are useful for reusability and faster compilation.

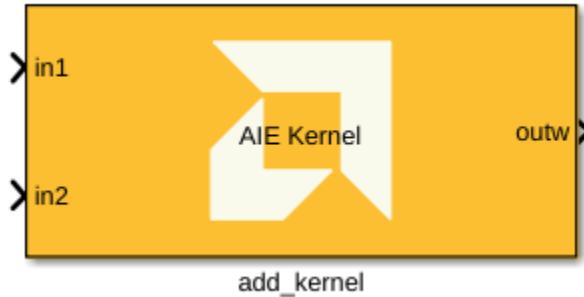
- Define each kernel in its own source file.
- Organize kernels by creating directories for header files and source files separately.
- Kernel source files should include all relevant header files to allow for independent compilation.

To import the `add_kernel` function as a block in a Model Composer design, double-click the **AIE Kernel** block and update parameters as follows:

- **Kernel header file:** `kernels/include/simple.h`
- **Kernel function:** `add_kernel`
- **Kernel Init function:** Leave empty
- **Kernel source file:** `kernels/source/simple.cc`
- **Kernel search path:** Leave empty
- **Preprocessor options:** Leave empty

When you click the **Import** button in the Block Parameters dialog box, the tool parses the function signature in the header file and updates the AI Engine kernel block GUI interface as shown in the following figure.

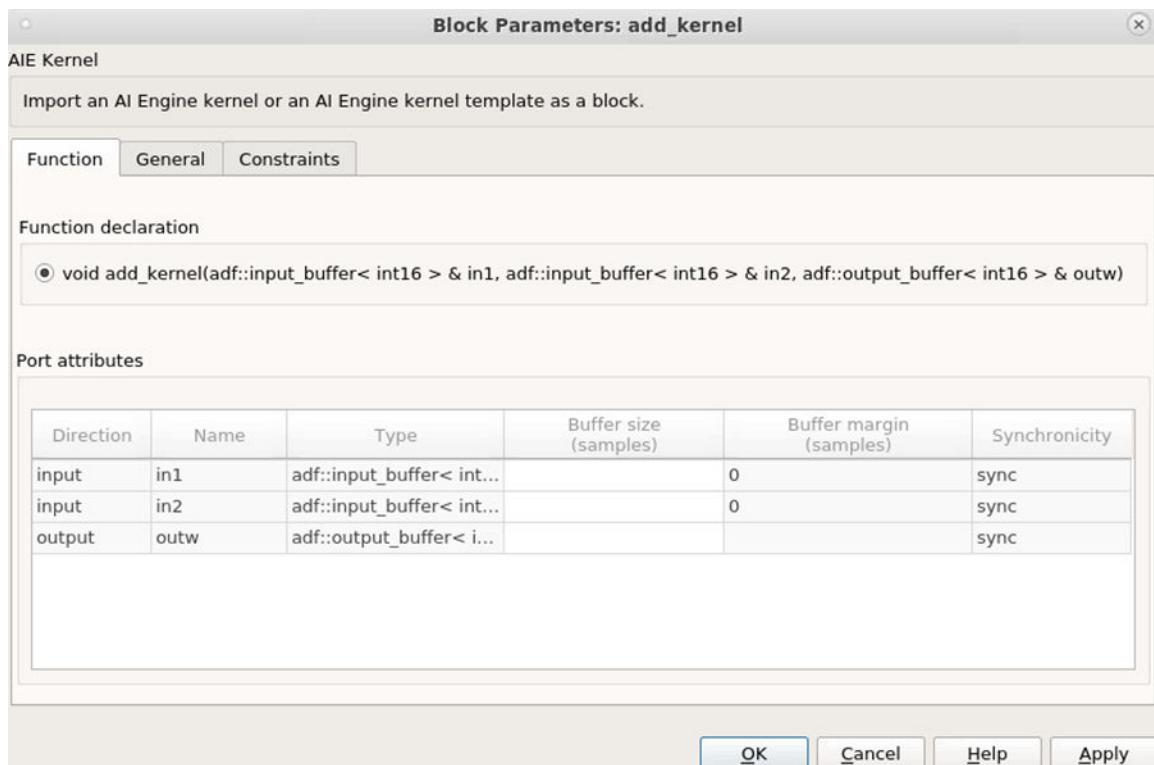
Figure 167: AIE Kernel Block (Updated)



After the AIE Kernel block is added to the Simulink editor, input and output ports are not present. But, after adding the kernel parameters in the Block Parameters dialog box, the block is updated with two input ports and one output port with block name matching the imported kernel function.

After a successful import, the Function tab displays automatically, providing user-editable configuration parameters. You can quickly review the function definition of the imported kernel function and the port names with directions.

Figure 168: Function Tab



Appropriate values should be entered in the Function tab for Buffer size and Buffer margin (see the previous figure).

### Setting the Buffer Margin Value

As explained in [Data Accessing Mechanisms](#), buffer margin is the overlapping of input data samples. Model Composer accepts Buffer margin value in terms of the number of samples. The values given in the Buffer margin fields should be multiple of 32 bytes.

For example, if your input data type is `int16` which is 2 bytes, the minimum Buffer margin value that is accepted is 16 samples ( $16 \times 2$ ). The other values that are accepted can be 32, 48, 64 samples and so on.

Another example: If your input is of type `c_int32` which is 8 bytes (real 4 bytes and 4 imaginary bytes), in this case the minimum buffer margin value that is accepted is 4 samples. The other values that are accepted can be 8, 12, 16 samples and so on.

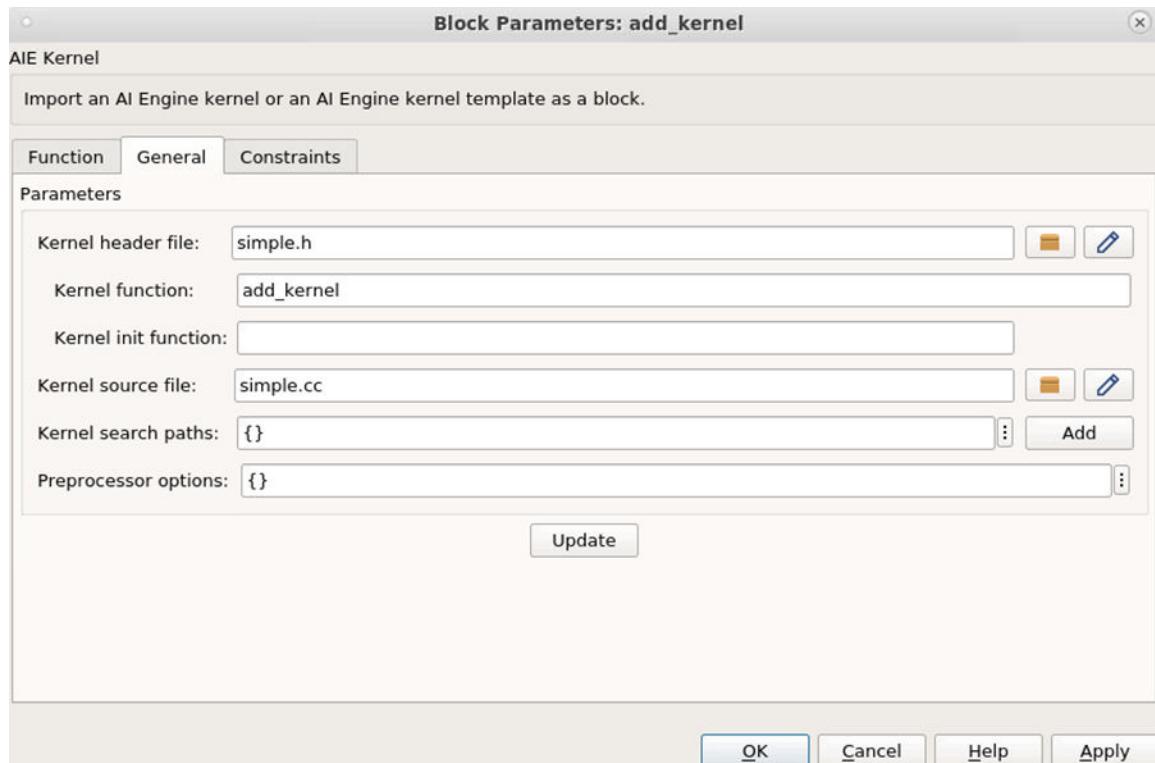
The following table provides details on the parameters and description for each parameter.

**Table 24: Buffer Port Parameters**

Parameter Name	Criticality	Description
Buffer size	Mandatory	<ul style="list-style-type: none"><li>• Buffer size is required for each port (argument) of the kernel function.</li><li>• The value represents the number of samples (elements).</li><li>• The buffer size must be a positive integer value.</li><li>• Buffer size should be a multiple of 16 bytes if no buffer margin is specified. If a margin is specified, the buffer size should be a multiple of 32 bytes.</li></ul>
Buffer margin	Mandatory	<ul style="list-style-type: none"><li>• Buffer margin is required for each input port (argument) of the kernel function.</li><li>• The value represents the number of samples (elements).</li><li>• The buffer margin must be a non-negative integer.</li><li>• The buffer margin should be a multiple of 32 bytes.</li></ul>
Synchronicity	Mandatory	<ul style="list-style-type: none"><li>• The Synchronicity value options available are <code>sync</code> and <code>async</code>.</li><li>• Port synchronicity is set to <code>sync</code> by default. You can optionally change it <code>async</code>.</li></ul>

After the successful import of kernel functions, the Import button label in the General tab changes to Update enabling further updates of block parameters. You can change the source code of the kernel function even after importing the block without requiring a re-import. However, if you change the function signature, or the parameters to the function, then you will need to click Update in the General tab to apply changes.

Figure 169: General Tab



### Importing Stream-Based Kernels

As explained in [Data Accessing Mechanisms](#), stream-based kernels access data streams in a sample-by-sample fashion. Vitis Model Composer supports the following stream-based input and output data types as interfaces to the AIE Kernel block.

- `input_stream<TYPE>`
- `output_stream<TYPE>`

Table 25: Stream-Based Input and Output Data Types

<Type>	Complexity	Signedness
int8	Real	Signed
int16	Real	Signed
int32	Real	Signed
int64	Real	Signed
uint8	Real	Unsigned
uint16	Real	Unsigned
uint32	Real	Unsigned
uint64	Real	Unsigned
cint16	Complex	Signed

**Table 25: Stream-Based Input and Output Data Types (cont'd)**

<Type>	Complexity	Signedness
cint32	Complex	Signed
float	Real	N/A
cfloat	Complex	N/A
bfloat16	Real	N/A
accfloat	Real	N/A
caccfloat	Complex	N/A

As an example, to import a simple kernel with a stream-based interface, the following `simple.h` header file declares the `simple_comp` function with one input stream and one output stream.

```
#ifndef __COMPLEX_KERNEL_H__
#define __COMPLEX_KERNEL_H__
#include <adf.h>
void simple_comp(input_stream<cint16> * in, output_stream<cint16> * out);
#endif //__COMPLEX_KERNEL_H__
```

The function is defined in `simple.cc`.

```
#include "simple.h"
void simple_comp(input_stream<cint16> * in, output_stream<cint16> * out) {
    cint16 c1, c2;

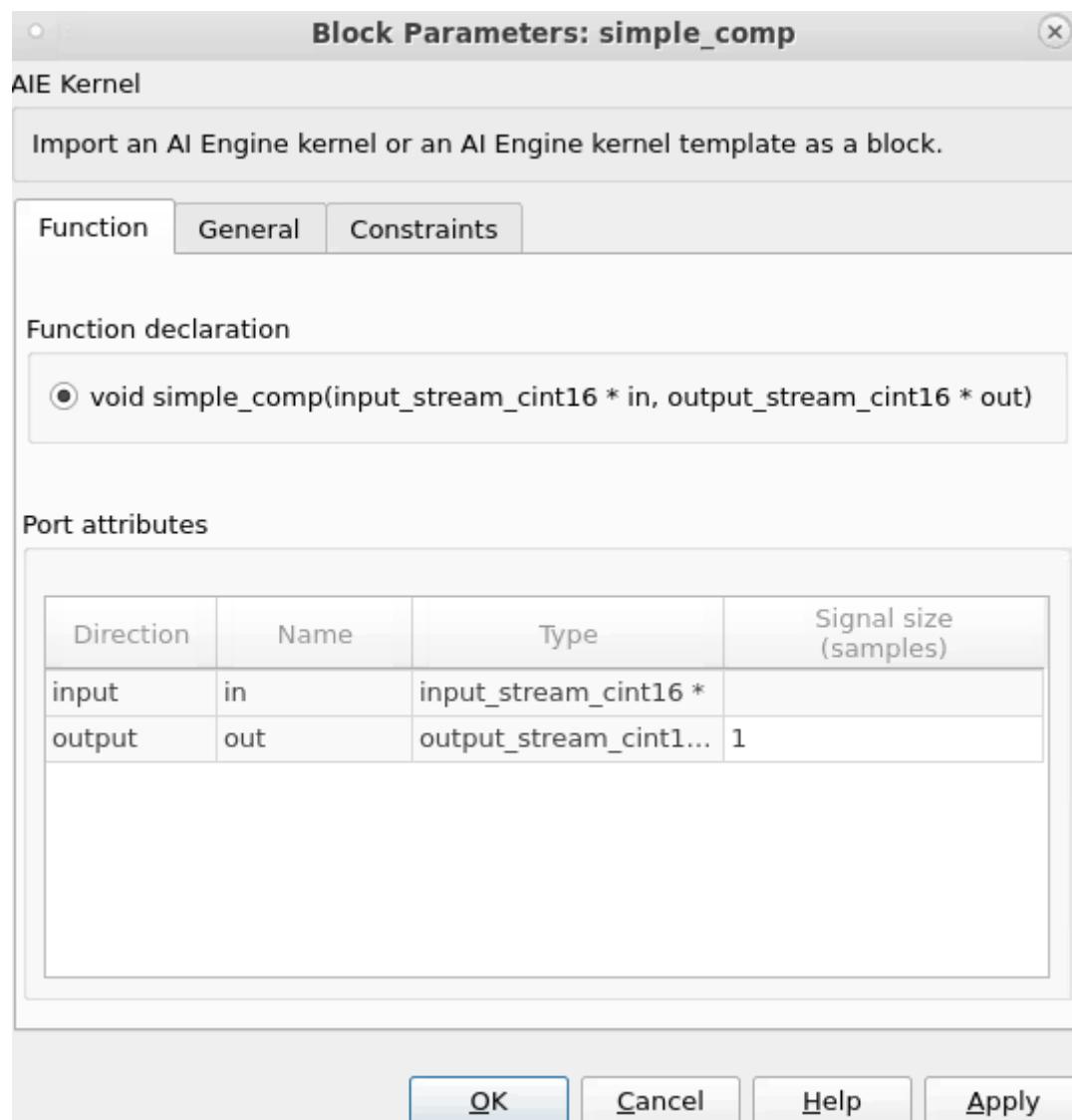
    for (unsigned i=0; i<NUM_SAMPLES; i++) {
        c1 = readincr(in);
        c2.real = c1.real+c1.imag;
        c2.imag = c1.real-c1.imag;
        writeincr(out, c2);
    }
}
```

**Note:** See the *AI Engine Tools and Flows User Guide* ([UG1076](#)) for details of the `readincr()` and `writeincr()` APIs.

Although the function arguments for buffer-based and stream-based kernels are different, the procedure for importing the stream-based kernel is the same.

After a successful import, the Function tab GUI displays automatically. You can quickly review the function definition and ports as shown in the following figure.

Figure 170: Function Tab



In order to import the kernel, the signal size of each output port must be specified. The following table provides details on this parameter.

Table 26: Stream Port Parameters

Parameter Name	Description
Signal size	This parameter represents the size of the output signal and must be set to a value greater than or equal to the maximum number of samples that are produced during any invocation of the kernel.

In the General tab, the Import button changes to Update, enabling further updates of block parameters.

Model Composer also supports cascade stream connections between two AI Engine processors.

An AI Engine kernel can use cascade stream connections as follows:

- `input_cascade<TYPE>`
- `output_cascade<TYPE>`

Different cascade data types are supported for each AI Engine architecture.

**Table 27: Cascade Stream-Based Input and Output Data Types (AIE1)**

<Type>	Complexity	Signedness
int8	Real	Signed
uint8	Real	Unsigned
int16	Real	Signed
cint16	Complex	Signed
int32	Real	Signed
cint32	Complex	Signed
float	Real	N/A
cfloat	Complex	N/A
acc48	Real	N/A
acc80	Real	N/A
cacc48	Complex	N/A
cacc80	Complex	N/A
accfloat	Real	N/A
caccfloat	Complex	N/A

**Table 28: Cascade Stream-Based Input and Output Data Types (AIE-ML)**

<Type>	Complexity	Signedness
int8	Real	Signed
uint8	Real	Unsigned
int16	Real	Signed
uint16	Real	Unsigned
cint16	Complex	Signed
int32	Real	Signed
uint32	Real	Unsigned
cint32	Complex	Signed
bfloat16	Real	N/A
acc32	Real	N/A
acc64	Real	N/A
cacc64	Complex	N/A
accfloat	Real	N/A

Table 28: Cascade Stream-Based Input and Output Data Types (AIE-ML) (cont'd)

<Type>	Complexity	Signedness
caccfloat	Complex	N/A

In Model Composer, a cascade stream port with an accumulator data type is represented as a fixed point signal (for example, `x_sfix48`) that can be either complex or real.

Consider the following example, where a cascade output stream of one kernel is connected to the cascade input stream of another kernel.

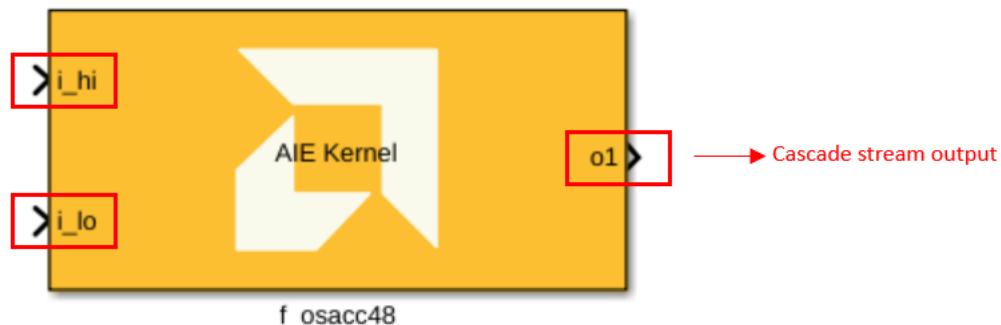
```
#ifndef __CASCADE_KERNELS_H__
#define __CASCADE_KERNELS_H__
void f_osacc48(input_buffer<int32> &i_hi,
                 input_buffer<int16> &i_lo,
                 output_cascade<acc48> *o1);
#endif
```

The kernel function `f_osacc48` has two input windows: `i_hi` and `i_lo`, and one cascade stream output: `o1`.

**Note:** This kernel function includes both window-based ports and stream-based ports.

After importing this kernel function, the AIE Kernel block is as shown in the following figure.

Figure 171: AIE Kernel after Import

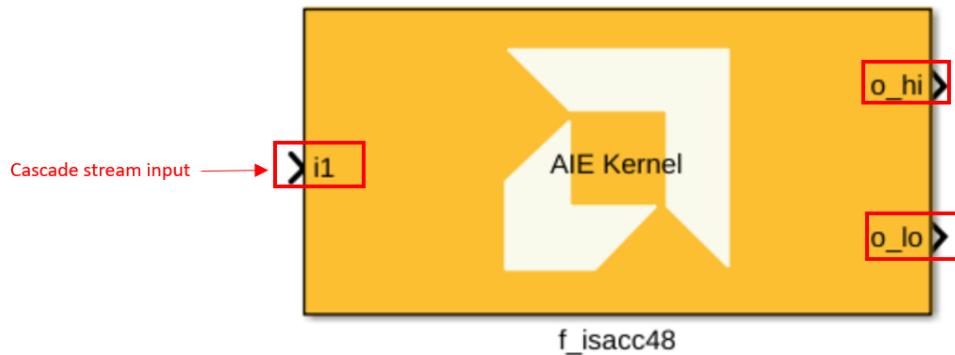


Consider another kernel function `f_isacc48`, which has one cascade stream input: `i1`, and two output windows: `o_hi` and `o_lo`.

```
#ifndef __CASCADE_KERNELS_H__
#define __CASCADE_KERNELS_H__
void f_isacc48(input_cascade<acc48> *i1,
                output_buffer<int32> &o_hi,
                output_buffer<int16> &o_lo);
#endif
```

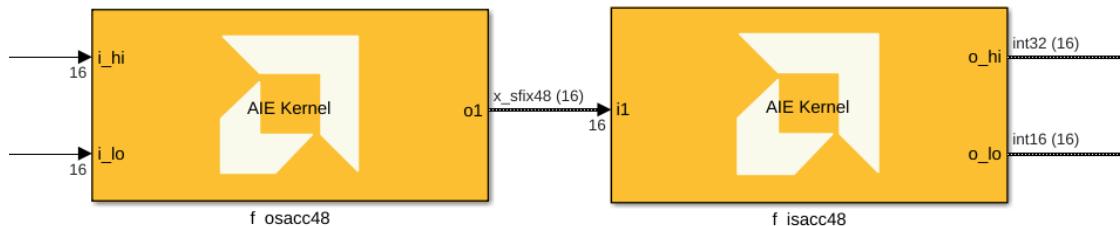
After importing the second kernel function, the AIE Kernel block is as shown in the following figure.

**Figure 172: AIE Kernel Block (Second Kernel Function)**



Now the two kernels can be connected to form a cascade connection using the cascade stream output of block `f_osacc48` and the cascade stream input of block `f_isacc48`. This is shown in the following figure.

**Figure 173: Connected Kernels (Cascade Connection)**



### Importing an AI Engine Kernel with Runtime Parameters

Vitis Model Composer supports importing AI Engine kernels with runtime parameters in kernel functions alongside window and stream types. The following table lists the scalar data types that can be passed as runtime parameters.

**Table 29: Scalar Data Types**

<Type>	Complexity	Signedness
int8	Real	Signed
int16	Real	Signed
int32	Real	Signed
int64	Real	Signed
uint8	Real	Unsigned
uint16	Real	Unsigned
uint32	Real	Unsigned
uint64	Real	Unsigned
cint16	Complex	Signed
cint32	Complex	Signed
float	Real	N/A
cfloat	Complex	N/A
bfloat16	Real	N/A

Implicit ports are inferred for each parameter (scalar and vector data types) in the function argument. The following table describes the type of port inferred for each function argument.

Formal Parameter	Port Class
T	Input
Const T	Input
T &	Inout
Const T &	Input
Const T (&) [ .. ]	Input
T (&) [ .. ]	Inout

In the following example, the `simple_rtp` function has two real-time parameters. Notice the function argument `select` which is passed by value, and argument `weight` which is passed by reference.

```
#ifndef __RTP_KERNEL_H__
#define __RTP_KERNEL_H__

void simple_rtp(input_buffer<cint16> & in, output_buffer<cint16> & outw,
int32 & weight, int32 select);

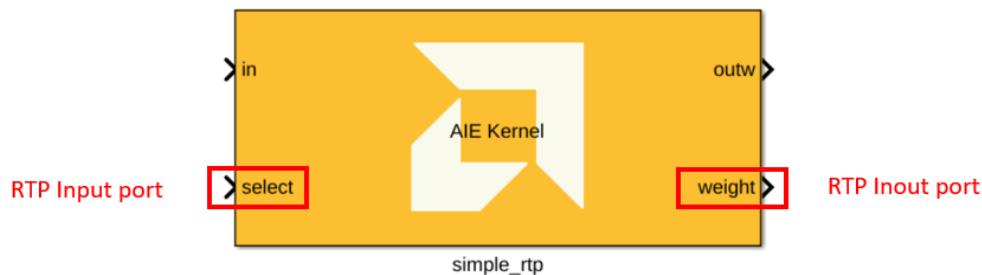
#endif //__RTP_KERNEL_H__
```

When imported for the above function, the AIE Kernel block looks as shown in the following figure. In Model Composer, the `inout` port appears as the `output` port on the AIE Kernel block.

**Note:** Vitis Model Composer ignores the `inout` RTP ports during code generation and only considers them for Simulink simulation. (that is, they will not be read from the PS).

Because RTPs are used alongside the window and stream ports, the procedure for importing the kernel function remains the same. When the above kernel function with RTPs are imported, the AIE Kernel block looks as shown in the following figure.

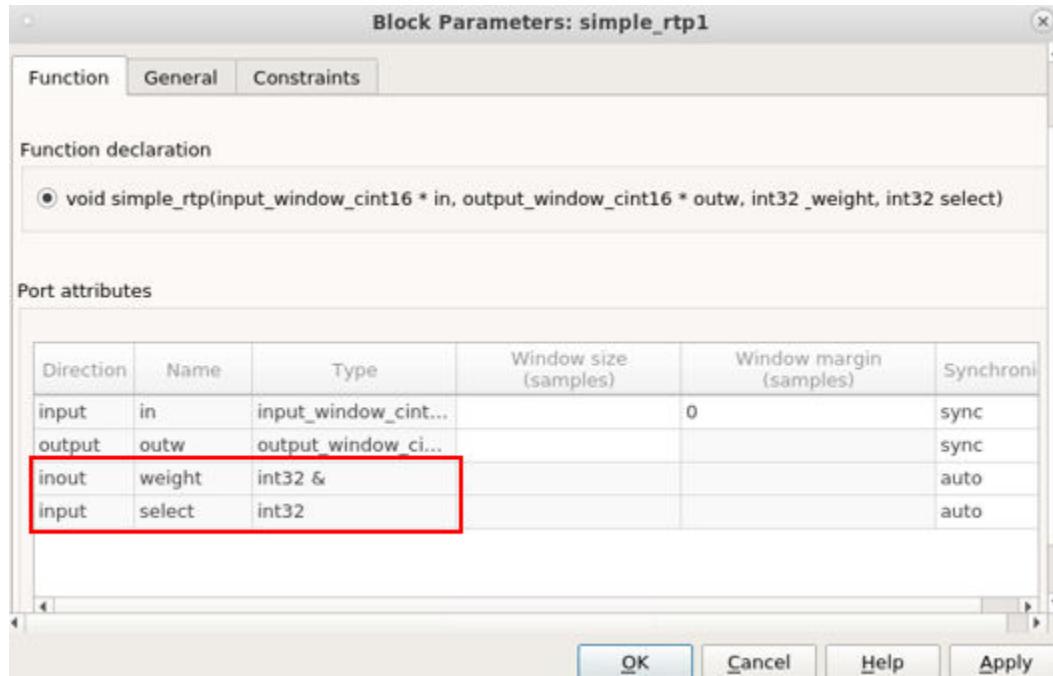
Figure 174: AIE Kernel (RTPs)



**Note:** The AIE kernel block name (`simple_rtp`) is the same as the AI Engine kernel function name.

After a successful import, the Function tab displays automatically. You can quickly review the function definition and runtime parameter ports as shown.

Figure 175: Function Tab



Port synchronicity is the only parameter that is specific to RTPs. The following table provides details about the valid synchronicity of the Destination RTP input port with respect to the Source RTP inout port. The default port synchronicity is set to 'auto'.

**Table 30: Valid Synchronicity**

Source RTP Inout Port	Destination RTP input Port
auto	async
sync	auto
sync	sync
async	async

If the source RTP inout port is set to 'auto' then the destination RTP input port should be 'async'. Similarly for other combinations. Model Composer throws an appropriate error when you try to use any combination which is not specified in the previous table.

### Importing an AI Engine Kernel with Function Template

You might require a generic function that can be used for different datatypes. Using templates, you can pass a datatype as a parameter and Vitis Model Composer supports importing an AI Engine Kernel with a function template. To do this, use the same AIE Kernel block used earlier to import the ordinary C++ functions.

As an example to import the kernel function with templates, consider below header file `kernel.h`, containing the declaration of a function template. Here, the template has typename template parameter T, and a non-type (integral) template parameter N.

`kernel.h`

```
#ifndef _AIE_TEMP_KERNELS_H_
#define _AIE_TEMP_KERNELS_H_

#include <adf.h>

using namespace adf;

template<typename T, int N>
void myFunc(input_buffer<T> &i1,
            output_buffer<T> &o1
            );

#endif // ifndef _AIE_TEMP_KERNELS_H
```

The definition of the function template is in the source file `kernel.cpp` as shown below.

`kernel.cpp`

```
#include "kernel.h"

template<typename T, int N>
void myFunc(input_buffer<T> &i1, output_buffer<T> &o1)
{
    auto pIn = aie::begin(i1);
    auto pOut = aie::begin(o1);
```

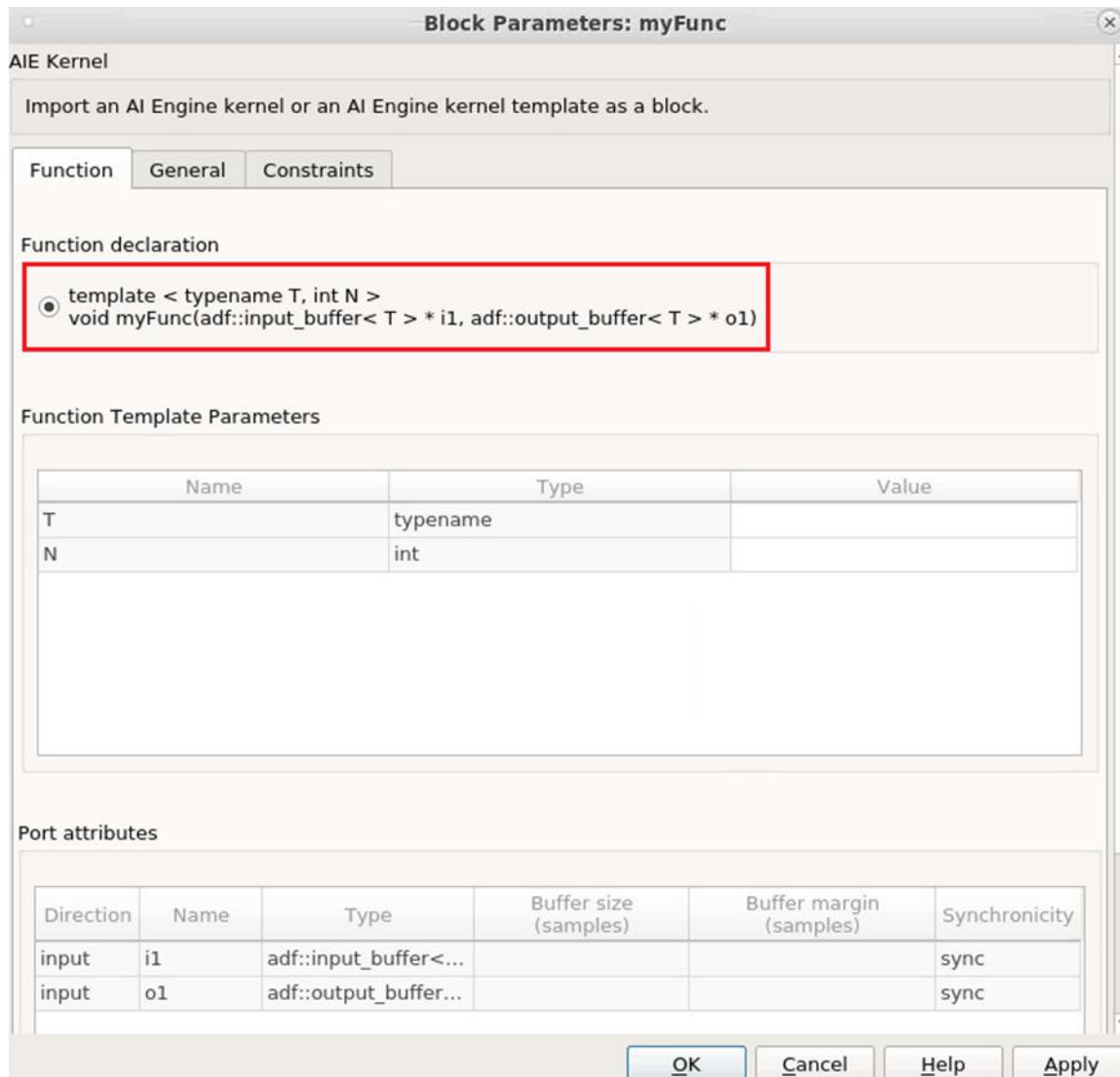
```
for(int i = 0; i < 8; i++)
{
    *pOut++ *= *pIn++;
}
```

Notice the usage of the non-type template parameter 'N' in the kernel output computation. To import the template function as a block into Model Composer, double-click the AIE Kernel block and update the parameters as follows.

- **Kernel header file:** kernel.h
- **Kernel function:** myFunc
- **Kernel Init function:** Leave empty
- **Kernel source file:** kernel.cpp
- **Kernel search path:** Leave empty
- **Preprocessor options:** Leave empty

When you click the Import button in the Block Parameters dialog box, the Function tab displays automatically. Enter the values of a template type parameter 'T' and a template non-type parameter of integral type within the Function Template Parameters section as shown in the following figure. Double-click the appropriate editable field and enter the values. You can also review the declaration of the template function in the Function declaration section.

Figure 176: AIE Kernel: Function declaration Section



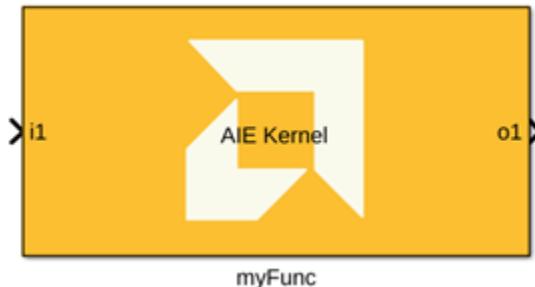
The following typenames are supported as type template parameters:

- int8, int16, int32, int64
- uint8, uint16, uint32, uint64
- bfloat16, float, double
- cint16, cint32, cfloat

Scroll down to the Port attributes section in the Function tab and enter appropriate user-editable configuration parameters as shown.

After entering the appropriate values in the Function tab, click Apply. Notice the updated interface of the AIE kernel block GUI as shown in the following figure.

Figure 177: AIE Kernel: Updated



### Template Specialization

For cases where you want to override the default template implementation to handle a particular type in a different way, Vitis Model Composer supports template specialization. Consider the following example where a function `myFunc` has one specialized version declared to implement an `int16` datatype along with a generic template function.

#### kernel.cpp

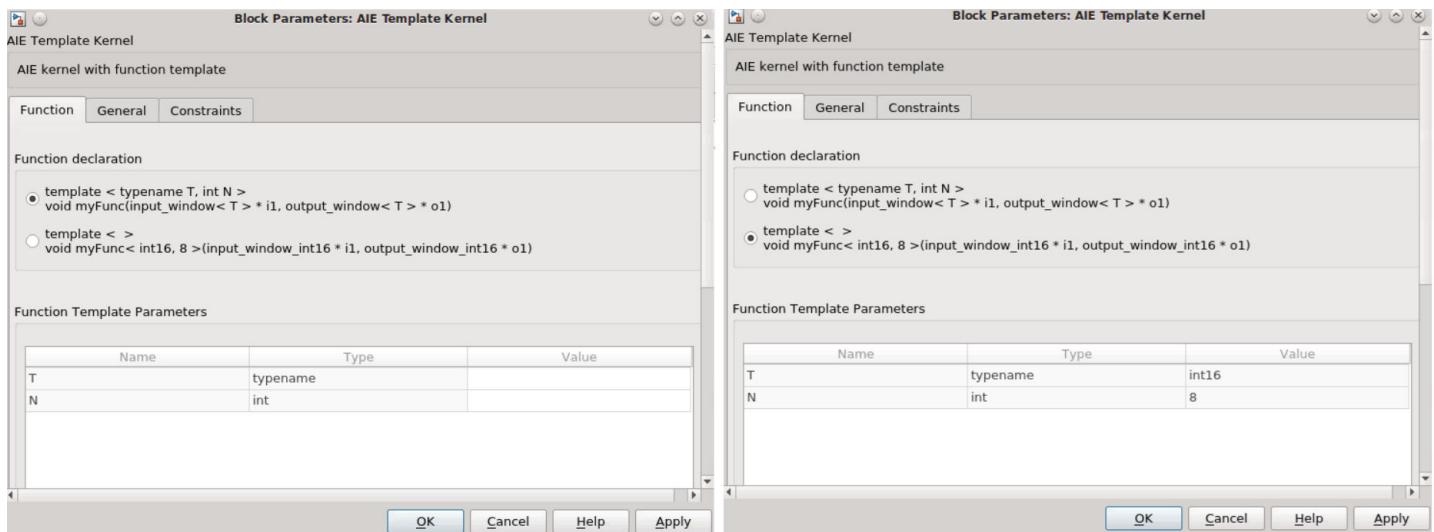
```
#ifndef _AIE_CLASS_KERNELS_H_
#define _AIE_CLASS_KERNELS_H_

#include <adf.h>
template<typename T, int N>
void myFunc(input_buffer<T> &i1,
            output_buffer<T> &o1
            );
template<>
void myFunc<int16,8>(input_buffer<int16> &i1,
                      output_buffer<int16> &o1);

#endif
```

When you try to import the kernel `myFunc` as a block into Vitis Model Composer using the AIE Template Kernel block, the Function tab in the block GUI parameter looks as shown. If you select the function variant corresponding to the base template, the Function Template Parameters table shows the values corresponding to that. If you select the specialization variant instead, the table shows the values of the template parameters of that specialization. You cannot change these values.

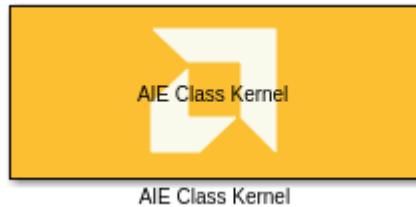
Figure 178: AIE Template Kernel: Functional declaration Options



## Class-Based Kernels

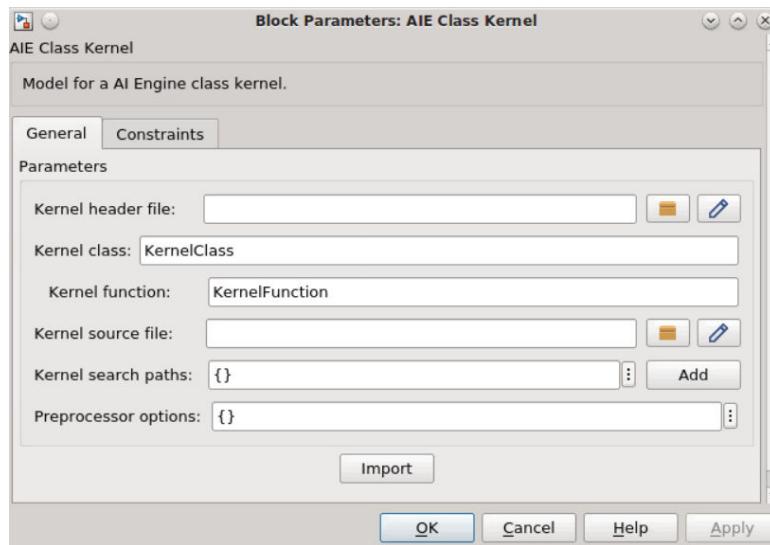
Vitis Model Composer supports importing the C++ kernel class to have constructor parameters for specifying parameter values. You need to use an AIE Class Kernel block from the AI Engine Library as shown.

Figure 179: AIE Class Kernel



Double-clicking the block symbol displays the parameters of the AI Engine class kernel block as shown in the following figure.

Figure 180: AIE Class Kernel: Block Parameters



The block mask parameters need to be updated to import the kernel function as a block. The following table provides details on the parameters and description for each parameter.

Table 31: AIE Class Kernel: Parameters

Parameter Name	Parameter Type	Criticality	Description
Kernel header file	String	Mandatory	Name of the header file that contains the kernel class and registerKernelClass method declarations. The string could be just the file name, a relative path to the file or an absolute path of the file. Use the browse button to select the file. This field does not accept environmental variables.
Kernel class	String	Mandatory	Name of the kernel class which contains member variables and kernel member function.
Kernel function	String	Mandatory	Name of the kernel member function for which the block is to be created. This function should be registered using the registerKernelClass method in kernel header file.
Kernel source file	String	Mandatory	Name of the source file that contains where the kernel member function definition and non-default constructor parameter values are specified. The string could be the file name, a relative path to the file, or an absolute path of the file. The string could be the file name, a relative path to the file or an absolute path of the file. This field does not accept environmental variables.
Kernel search paths	Vector of Strings	Optional	If the kernel header file or the kernel source file are not found using the value provided through the Kernel header file or Kernel source file fields, respectively, then the paths provided through Kernel search paths are used to find the files. This parameter allows use of environment variables while specifying paths for the kernel header file and the kernel source file. The environment variable can be used in either \${ENV} or \$ENV format.

**Table 31: AIE Class Kernel: Parameters (cont'd)**

Parameter Name	Parameter Type	Criticality	Description
Preprocessor options		Optional	<p>Optional preprocessor arguments for downstream compilation with specific preprocessor options.</p> <p>The following two preprocessor option formats are accepted and multiple can be selected: <code>-D&lt;name&gt;</code> and <code>-D&lt;name&gt;=&lt;definition&gt;</code> separated by a comma. That is, the optional argument must begin with <code>-D</code> and if the option <code>&lt;definition&gt;</code> value is not provided, it is assumed to be 1.</p>

The AIE Class Kernel block supports all the kernel functions that a normal AI Engine kernel block can support and the Block Parameters dialog box which appears after double-clicking on the AI Engine class kernel block is the same irrespective of whether the kernel member function is Window-based or Stream-based. To edit the header file or source file, you can click the Edit button (immediately after the browse button).

### Kernel Class with Default Constructor

As an example, to import the C++ class kernel with the default constructor, consider the following `simple.h` header file that defines the kernel class `simple_class` with the default constructor.

`simple.h`

```
#include "adf.h"

using namespace adf;

class simple_class
{
private:
    int16 val;
    int16 numSamples;

public:
    simple_class();
    void mulBytwo(input_buffer<int16> & in, output_buffer<int16> & out);

    static void registerKernelClass()
    {
        REGISTER_FUNCTION(simple_class::mulBytwo);
    }
};
```



**RECOMMENDED:** It is highly recommended to define the body of the kernel and class constructors in `.cpp` file.

It is necessary to register the kernel function using the `registerKernelClass()` method. More than one class kernel can be declared in a header file and each class should contain separate `registerKernelClass()` methods. Only one function can be registered per class and Vitis Model Composer can import only functions that are registered using `REGISTER_FUNCTION()`.

The Kernel function is defined in `simple.cpp` as shown below.

#### simple.cpp

```
#include "simple.h"

simple_class::Simple_class()
{
    val = 24;
    numSamples = 8;
}

void simple_class::mulBytwo(input_buffer<int16> & in, output_buffer<int16>
& out)
{
    auto pIn = aie::begin(in);
    auto pOut = aie::begin(out);

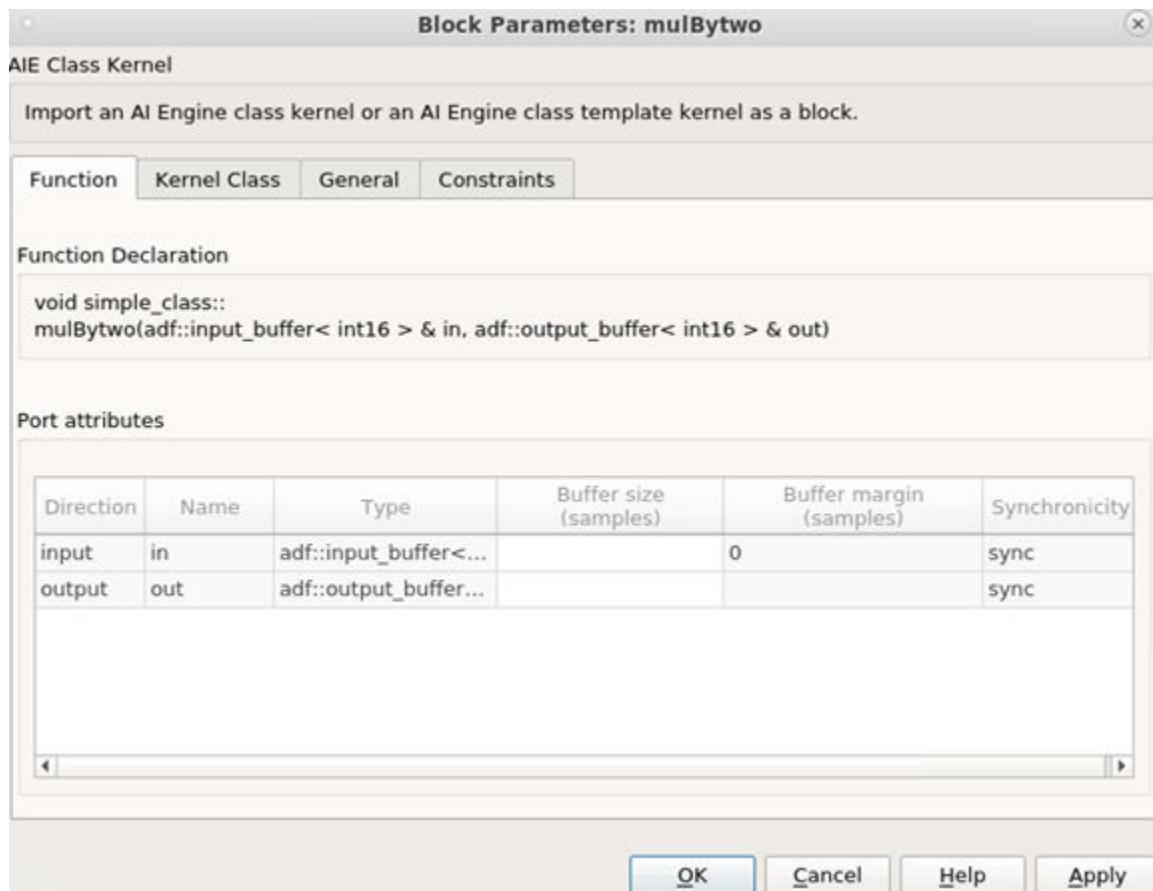
    for (int i=0; i<numSamples; i++)
    {
        *pOut++ = 2*(*pIn++) + val;
    }
}
```

To import the `mulBytwo` function as a block in a Vitis Model Composer design, double click the AIE Class Kernel block and update the parameters as follows.

- **Kernel header file:** `simple.h`
- **Kernel class:** `simple_class`
- **Kernel function:** `mulBytwo`
- **Kernel source file:** `simple.cpp`
- **Kernel search path:** Leave empty
- **Preprocessor options:** Leave empty

Click the **Import** button in the Block Parameters dialog box. After successful import, the Function tab displays. This provides user-editable configuration parameters as shown in the following figure.

Figure 181: AIE Class Kernel: Block Parameters



After entering the appropriate values in the Function tab, click **Apply** to see the updated interface of the AIE Class Kernel block GUI as shown.

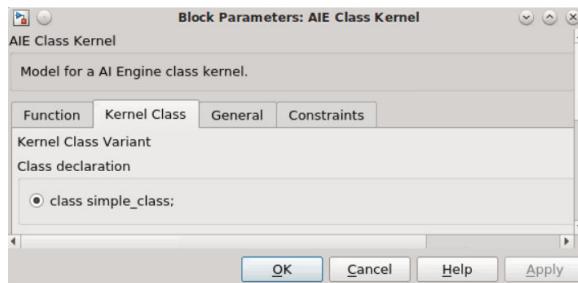
Figure 182: AIE Class Kernel after Update



You can quickly review the function declaration of the imported kernel function and the port names with directions, from the Function tab.

Click the **Kernel Class** tab to observe the class declaration as shown in the following figure.

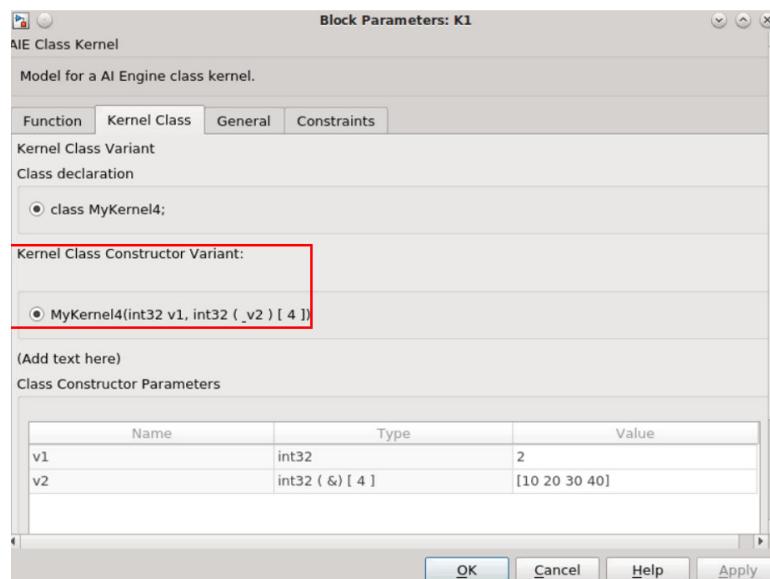
Figure 183: Kernel Class Tab: Class Declaration



### Class Kernels with Parameterized Constructors

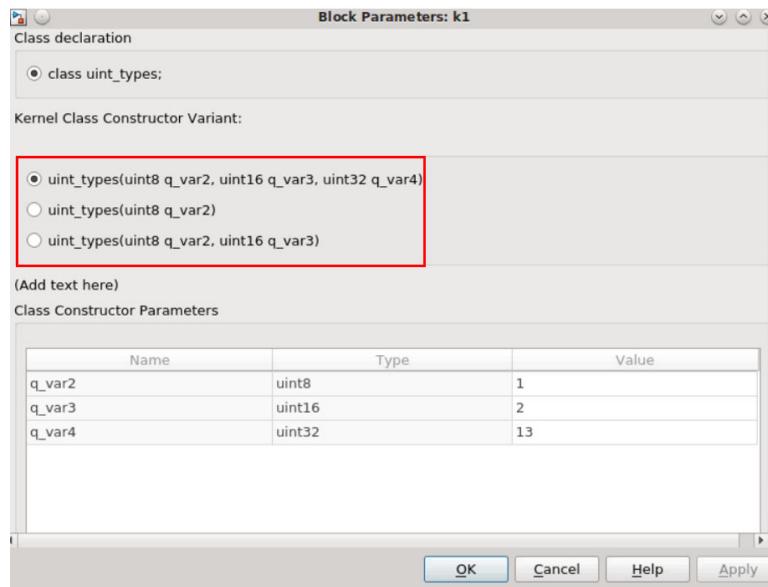
Default constructors do not take any arguments and have no parameters. However, it is possible to pass arguments to the constructors and Vitis Model Composer supports importing the class kernels with parameterized constructors. Both scalar and vector arguments can be passed to constructors. You can assign values to these parameters from the Kernel Class tab in the AI Engine Class Kernel block as shown in the following figure. You can also observe the parameterized constructor declaration in Kernel Class Constructor Variant section.

Figure 184: Kernel Class: Parameter Values



If you have multiple variants of Kernel Class Constructors, you can choose one of them and pass values to the constructor arguments accordingly.

Figure 185: Kernel Class: Multiple Values



### Constructor with Reference to an Array

Consider the following example that declares the constructor with reference to an array as argument.

#### **fir.h**

```
class FIR
{
private:
    int32 (&coeffs)[NUM_COEFFS];
    int32 tapDelayLine[NUM_COEFFECTS];
    uint32 numSamples;
public:
    FIR(int32(&coefficients)[NUM_COEFFECTS], uint32 samples);
    void filter(input_buffer<int32> & in, output_buffer<int32> & out);
    static void registerKernelClass()
    {
        REGISTER_FUNCTION(FIR::filter);
        REGISTER_PARAMETER(coeffs);
    }
};
```

#### **fir.cpp**

```
#include "fir.h"
FIR::FIR(int32(&coefficients)[NUM_COEFFECTS], uint32 samples)
: coeffs(coefficients)
{
    for (int i = 0; i < NUM_COEFFECTS; i++)
        tapDelayLine[i] = 0;
    numSamples = samples;
```

```
    }
void FIR::filter(input_buffer<int32> & in, output_buffer<int32> & out)
{
    ...
}
```

Here, member variable `coeffs` is an `int32 (&) [NUM_COEFFS]` data type. The constructor initializer `coeffs(coefficients)` initializes `coeffs` to the reference to an array allocated externally to the class object. To let the aiecompiler know that the `coeffs` member variable is intended to be allocated by the compiler, you must use `REGISTER_PARAMETER` to register an array reference member variable inside the `registerKernelClass()` method. The aiecompiler throws an appropriate error if the constructors with reference to an array are not registered.

### Kernel with Class Templates

You might require a class implementation that remains the same for all classes but the data types vary. Vitis Model Composer supports importing the kernels with class templates using the AIE Class Kernel block. Consider the following example which declares the class template in `kernel.h`.

```
kernel.h

#ifndef _AIE_CLASS_KERNELS_H_
#define _AIE_CLASS_KERNELS_H_
#include <adf.h>

template<typename T, int N>
class MyKernel {
    int m_count;
public:
    MyKernel();
    void myFunc(input_stream<T> *i1,
                output_stream<T> *o1,
                output_stream<int> *o2);

    static void registerKernelClass()
    {
        REGISTER_FUNCTION(MyKernel::myFunc);
    }
};

#endif
```

In this case, the default constructor initializer `m_count(N)` initializes `m_count` with template parameter `N` as shown in the following `kernel.cpp` code.

### kernel.cpp

```
#include "kernel.h"

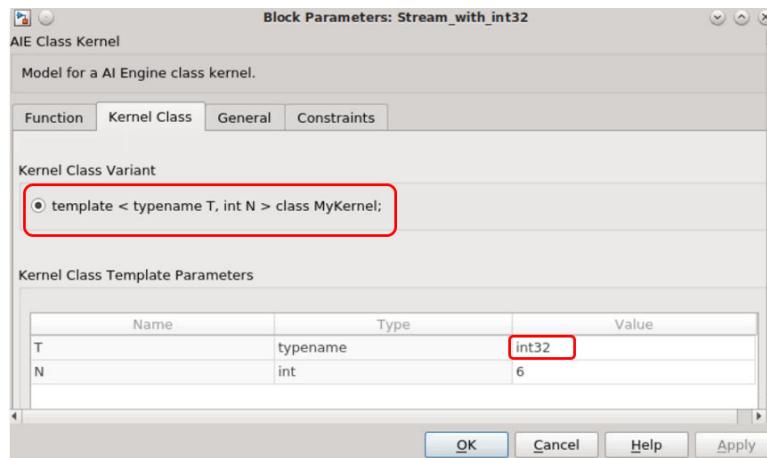
template<typename T, int N>
MyKernel<T,N>::MyKernel()
: m_count(N)
```

```
{  
}  
  
template<typename T, int N>  
void  
MyKernel<T,N>::myFunc( input_stream<T> *i1,  
                         output_stream<T> *o1,  
                         output_stream<int> *o2)  
{  
    put_ms(0, get_ss(0) * N);  
    ++m_count;  
    writeincr(o2, m_count);  
}
```

After successfully importing the kernel with class template using the AIE Class Kernel block, the Function tab displays. Here you can enter appropriate values in the user-editable configuration parameters. Click **Apply** to see the updated interface in the Block Parameters dialog box the AIE Class Kernel block.

Redirect to the Kernel Class tab in the Block Parameters dialog box to review the template class declaration from the kernel class variant. In the Kernel Class tab, you can enter the value of a template type parameter 'T' and a template non-type parameter of integral type as shown.

*Figure 186: Kernel Class Template*



### IMPORTANT!

1. Template type parameters can be any valid window, stream, or RTP datatypes.
2. Only a value that has an integral type is supported for template non-type parameters.
3. MATLAB variables can be used to specify non-type template parameters.

## Template Specialization

For cases when you need to override the default template implementation to handle a particular type in a different way, Vitis Model Composer supports template specialization. Consider the following example where a class `MyClass` has two different interfaces than the generic `MyClass`. One specialized version is declared to implement the `cint16` datatype and other version to implement the `uint32` datatype.

`template_specialization.h`

```
#include <adf.h>
template<typename T,int N>
class MyClass
{
};

template<>
class MyClass<cint16,1> {
    int m_count;
    int16 var_1;
    int16 var_2;
    int16 var_3;
    uint16 var_4;
public:
    MyClass();
    MyClass(int16 q_var1,int16 q_var2,int16 q_var3,uint16 q_var4);
    MyClass(int16 q_var1,int16 q_var2);
    MyClass(int16 q_var1,int16 q_var2,int16 q_var3);

    void func_mem(input_stream<cint16> *il,
                  output_stream<cint16> *ol,
                  output_stream<int> *o2);

    static void registerKernelClass()
    {
        REGISTER_FUNCTION(MyClass::func_mem);
    }
};

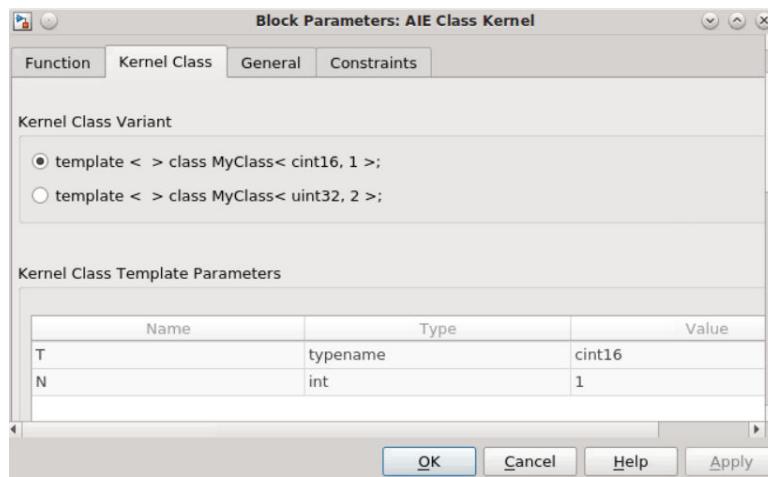
template<>
class MyClass<uint32,2> {
    int m_count;
    int16 var;
public:
    MyClass(uint32 q_var1);

    void func_mem(input_stream<uint32> *il,
                  output_stream<uint32> *ol,
                  output_stream<int> *o2);

    static void registerKernelClass()
    {
        REGISTER_FUNCTION(MyClass::func_mem);
    }
};
```

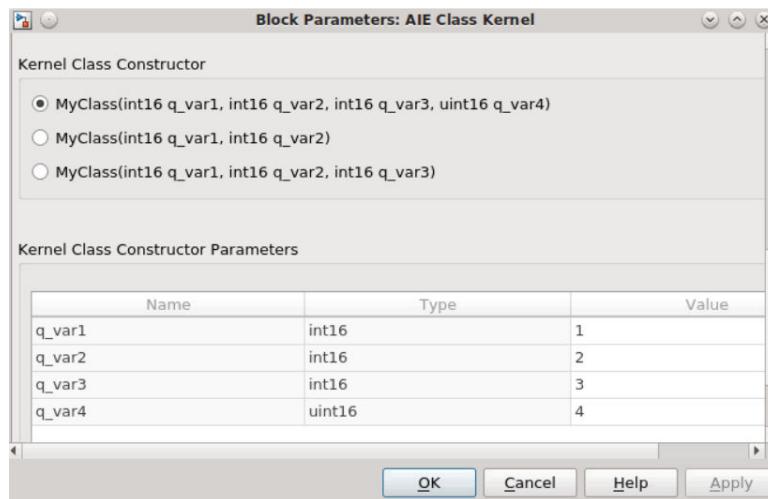
You can see that two functions are registered separately in two specialized classes. When you try to import the kernel `func_mem` as a block into Vitis Model Composer using the AIE Class kernel block, the Kernel Class tab in block GUI parameters looks as shown.

Figure 187: Class Variant



After selecting one of the Kernel Class Variants from the list, the Class Template Parameters update accordingly. The list of Kernel Class Constructors for the corresponding class variant, is updated and you can select from the list (see the following figure).

Figure 188: Kernel Class Constructors



**Note:** MATLAB variables can be used to specify the values of Kernel Class Constructor Parameters.

## Template Partial Specialization

For cases where you write a template that specializes one template parameter and still allows some parameterization, you can use the template partial specialization. Vitis Model Composer allows you to import the class kernels with partial specialization using the AIE Class Kernel block. Consider the following example where a class `class_a` is partially specialized with a non-type template parameter.

`partial_specialization.h`

```
#include <adf.h>
template<typename T,int N>
class class_a
{
};

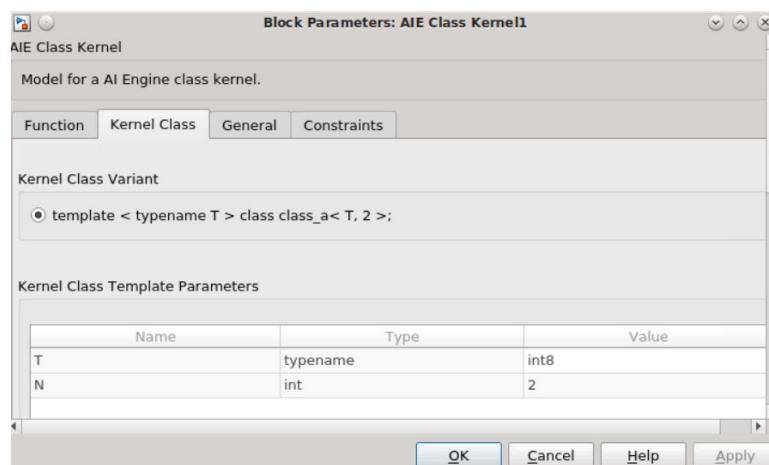
template<typename T>
class class_a<T,2> {
    int m_count;
    T var;
public:
    class_a(T q_var1);

    void func_mem(input_stream<T> *i1,
                  output_stream<T> *o1,
                  output_stream<int> *o2);

    static void registerKernelClass()
    {
        REGISTER_FUNCTION(class_a::func_mem);
    }
};
```

Notice that the function `func_mem` is registered in `registerKernelClass()` method. When you try to import the kernel function as a block into Model Composer using the AIE Class Kernel block, the Kernel Class tab in the Block Parameters dialog box looks as shown.

Figure 189: Block Parameters: AIE Class Kernel



Because the class is partially specialized with a non-type template parameter, you cannot edit the parameter 'N' from the Kernel Class Template Parameters. However, the value of the template type parameter can be modified to any valid datatype.

## Kernels with Namespaces

Vitis Model Composer supports importing kernels declared in a namespace. For both templated and non-templated kernels you need to qualify the kernel function with the namespace.

Consider the following examples where the non-templated kernel function is qualified with the namespace `ns1` and templated kernel function is qualified with the namespace `ns2`.

### Kernel.h

```
namespace ns1 {  
void myFunc_1(input_stream<int32> * restrict i1,  
              output_stream<int32> * restrict o1);  
} // namespace ns1
```

### templateKernel.h

```
namespace ns2 {  
template<typename T,int size>  
void myFunc_2(input_stream<int32> * restrict i1,  
              output_stream<int32> * restrict o1);  
} // namespace ns2
```

To import the above functions using the AIE Kernel and AIE Template Kernel blocks, the Kernel function parameter in the Block Parameters dialog box should be updated as follows:

- **Kernel function:** `ns1::myFunc_1` (Non-templated function)
- **Kernel function:** `ns2::myFunc_2` (Templated function)

If you have a class kernel declared in a namespace, then only the kernel class field should be qualified, and not the kernel function.

For example, consider the following kernel class which is qualified with the namespace `ns3`.

```
class_kernel.h  
  
namespace ns3 {  
#include "adf.h"  
  
class simple_class  
{  
private:  
    int16 val;  
    int16 numSamples;  
  
public:  
    simple_class();
```

```
void func_q(input_window_int16* in, output_window_int16* out);

static void registerKernelClass()
{
    REGISTER_FUNCTION(simple_class::func_q);
}
} ;
} // namespace ns3
```

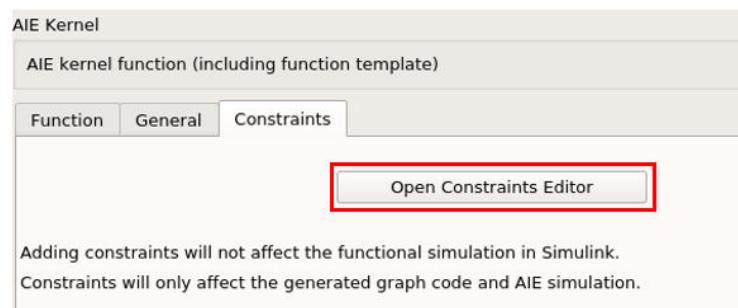
To Import the kernel function `func_q` as a block, the Kernel class and Kernel function parameters in the AIE Class Kernel block should be updated as follows:

- **Kernel class:** `ns3::simple_class`
- **Kernel function:** `func_q`

## Specifying Constraints

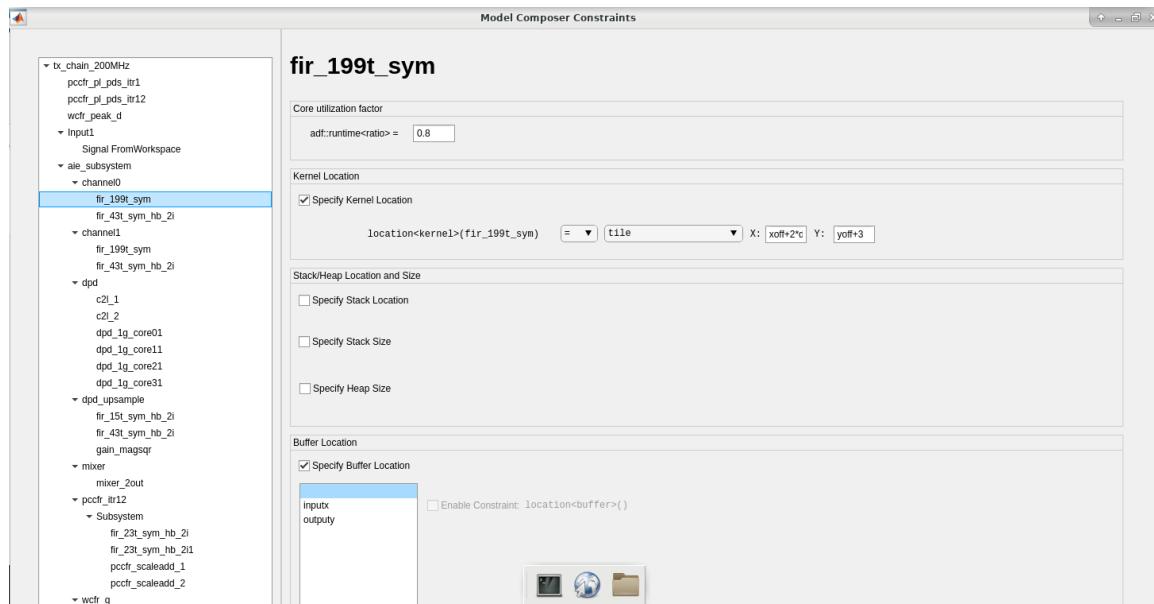
Constraints are user-defined properties for graph nodes which provide additional information to the compiler. Vitis Model Composer provides a mechanism to specify these constraints from within the kernel import or graph import blocks in the AI Engine library. During graph code generation, all these constraints automatically appear inside the graph code. The following figure shows the Constraints tab of the AIE Kernel block, highlighting the Open Constraints Editor button.

Figure 190: Constraints Tab



Clicking **Open Constraints Editor** opens the Vitis Model Composer Constraints dialog box. Here you can specify various constraints such as core utilization factor, kernel location, buffer location, stack/heap location, and size as shown in the following figure.

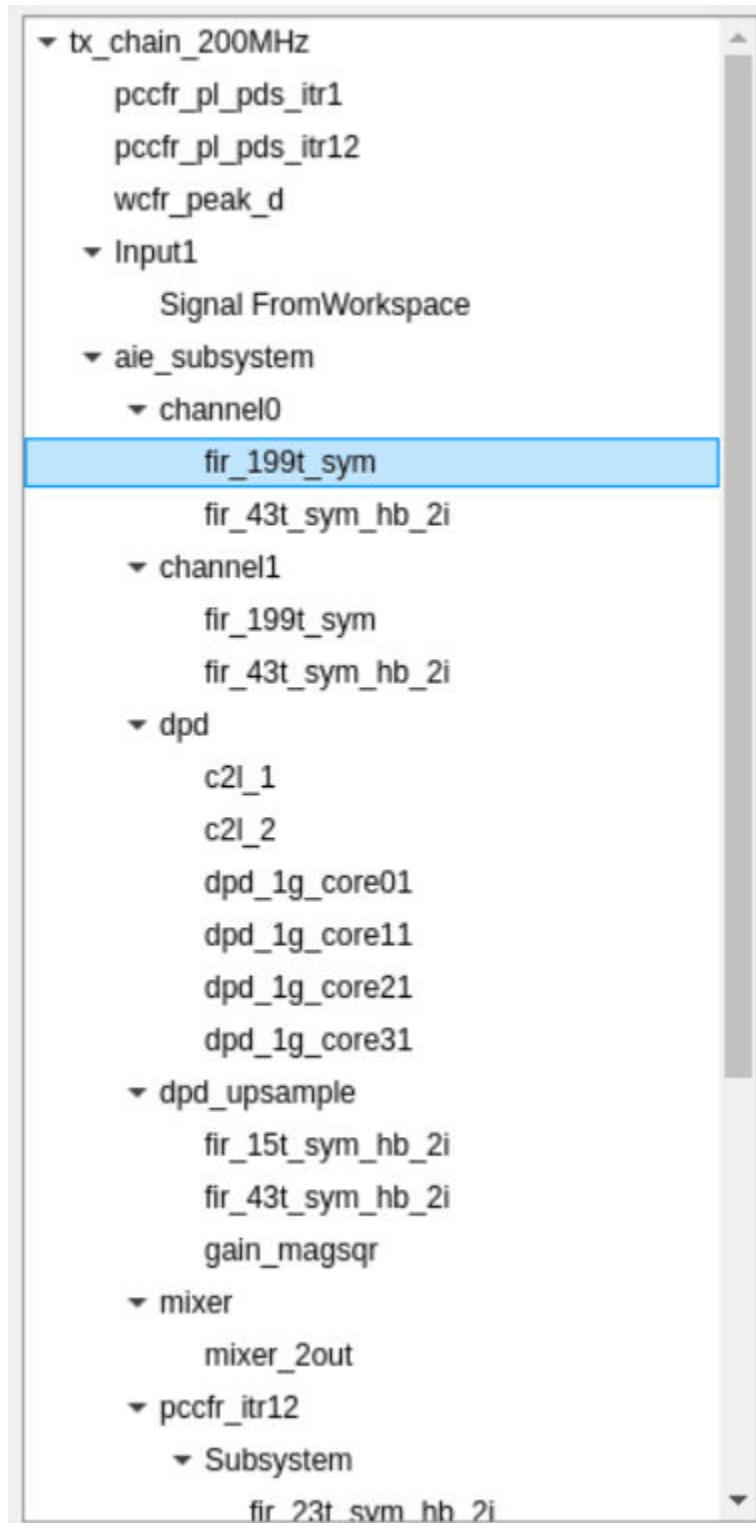
Figure 191: Model Composer Constraints Dialog Box



A similar constraints tab is available for all blocks in the AMD Toolbox/AI Engine/User-Defined Functions library and the constraint editor reflects the constraints available for that particular block. Adding these constraints will not affect Simulink simulation as they are only used for generating the graph code and AIE simulation.

When you open the constraints editor from any particular kernel/graph import block, the Model Composer Constraints dialog box allows you to specify the constraints for that particular kernel. However, you can switch between all the kernels/graphs available in your design from the navigation panel on the left side and specify the constraints accordingly as shown in the following figure. In this way, you can avoid opening each kernel/graph block separately to specify constraints.

Figure 192: Navigation Panel in Constraints Editor



The remainder of this section discusses the types of constraints that Vitis Model Composer supports.

### Core utilization factor (runtime<ratio>)

The core utilization factor ratio is specified as the ratio of the function run time compared to the cycle budget and must be between 0 and 1. The cycle budget is the number of instruction cycles a function can take to either consume data from its input or to produce a block of data on its output.

You can specify the core utilization factor in the Vitis Model Composer Constraints editor window as shown.

*Figure 193: Core utilization factor*



### Kernel Location

When building large graphs with multiple subgraphs, it is sometimes useful to control the exact mapping of kernels to AI Engines, either relative to other kernels or in an absolute sense. Specifying location constraints provides a powerful mechanism to create a robust, scalable, and predictable mapping of your graph onto the AI Engine array. It also reduces the choices for the mapper to try, which can considerably speed up the mapper.

The kernel location constraint can be specified from the Model Composer Constraints editor window as shown.

- By default, the option **Specify Kernel Location** is deselected. Select this to specify the constraint.

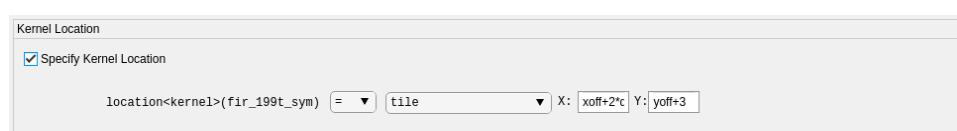
*Figure 194: Specify Kernel Location*



You can choose to either:

- Constrain a kernel to be placed on a specified AI Engine tile.

*Figure 195: Constrain Kernel*



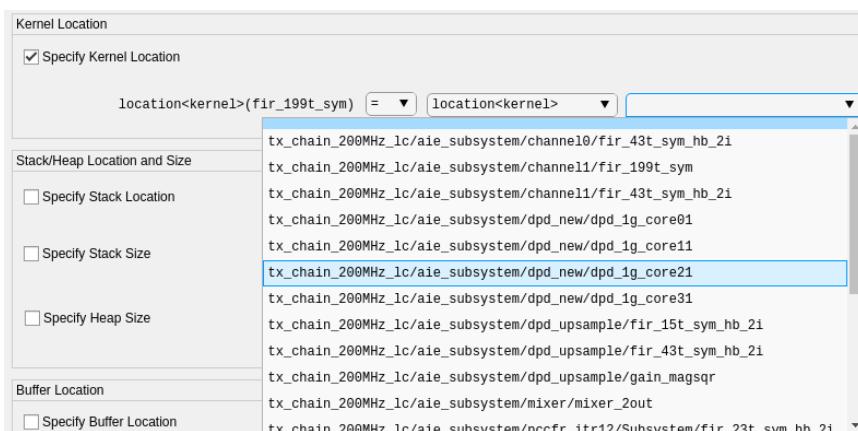


**IMPORTANT!** MATLAB variables can be used to specify the kernel locations.

Or

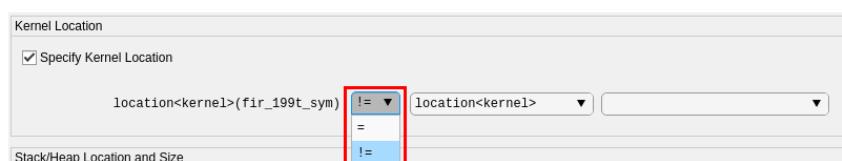
- Constrain two kernels to be placed relatively on the same AI Engine. This forces them to be sequenced in topological order and be able to share memory buffers without synchronization. As shown, you can select the kernel you want to place relatively on the AI Engine from the drop down.

**Figure 196: Constrain Two Kernels**



- Vitis Model Composer also supports specifying two kernels, say k1 and k2. These should not be mapped to the same AI Engine.

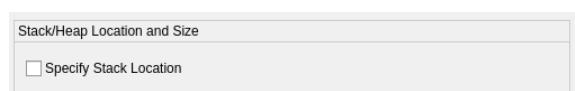
**Figure 197: Specify Two Kernels**



### Stack Location

The stack location constraint is used to specify the location of the system memory (stack and heap) of the AI Engine where the specified kernel is mapped. This provides the mechanism to constrain the location of the system memory with respect to other buffers used by that kernel. By default, the option **Specify Stack Location** is deselected. Select this to specify the constraint.

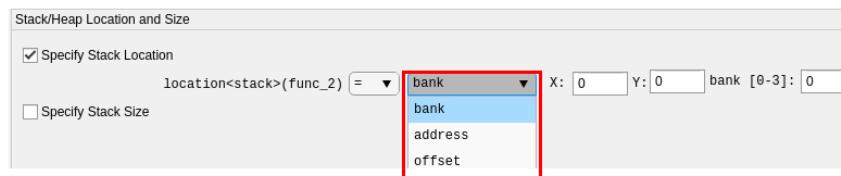
**Figure 198: Specify Stack Location**



You can specify the stack location by pointing to a:

- Specific data memory bank on an AI Engine tile. The bank ID is relative to the tile and can take values 0,1,2,3.
- Specific data memory address on an AI Engine tile. The offset address is relative to the tile starting at zero with a maximum value of 32768 (32K).
- Specific data memory address offset. The offset address is between 0 and 32768 (32K) and is relative to a tile allocated by the compiler.

Figure 199: Specify Data Memory Address Offset



Vitis Model Composer also supports specifying the stack location of the kernel where it should not be mapped to a particular bank, address, or offset.

### Stack Size

This constraint allows you to set the stack size for an individual kernel. By default the option Specify Stack Size is deselected. Select this to specify the constraint. The default value is 1024.

Figure 200: Specify Stack Size



### Heap Size

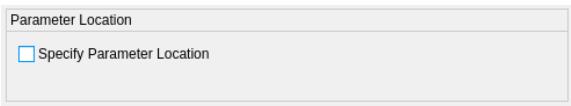
This constraint allows you to set the heap size for an individual kernel. By default the option Specify Heap Size is deselected. Select this to specify the constraint. The default value is 1024.

Figure 201: Specify Heap Size

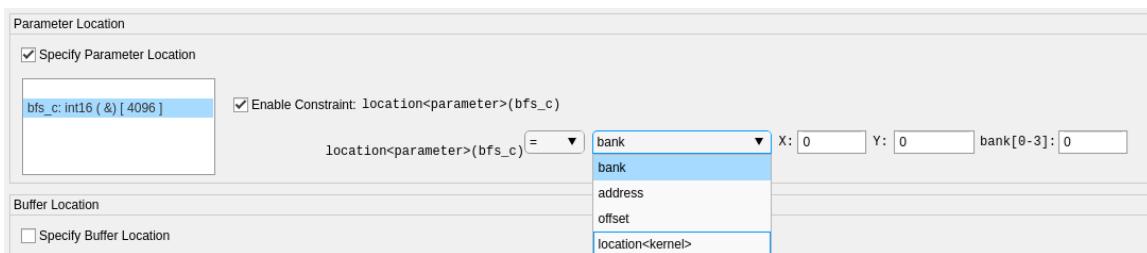


### Parameter Location

This constraint allows you to set the location of the parameter array declared within the graph.

**Figure 202: Specify Parameter Location**

By default, the option **Specify Parameter Location** is deselected. When you enable this option, the parameters that can be constrained are displayed. You can click on each individual parameter and constrain the location of a parameter lookup table to be placed on specific address or a bankid. You can also constrain the parameter location to be on the same tile as that of some other kernel. This ensures that the buffer, or parameter array can be accessed by the other kernel without requiring a DMA.

**Figure 203: Constrain Parameter Location**

### Buffer Location

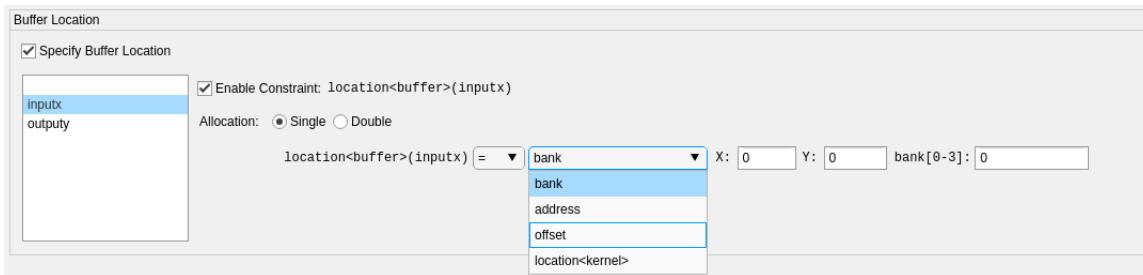
The AI Engine compiler attempts to automatically allocate buffers for windows, lookup tables, and runtime parameters in the most efficient manner possible. However, you might want to explicitly control their placement in memory. Similar to the kernels shown previously in this section, buffers inferred on a kernel port can also be constrained to be mapped to specific tiles, banks, or even address offsets using location constraints.

**Figure 204: Specify Buffer Location**

- By default, the option **Specify Buffer Location** is deselected. When you enable this option, the kernel ports that can be constrained are displayed. Buffer locations are only allowed on window kernel ports.
- You can click on each individual port and enable constraint as shown in the following figure.
- You can use the Allocation option to specify a single or double buffer constraint on a window port. By default, a window port is double buffered.
- For a single buffer allocation, you can choose to constrain the buffer location by pointing to a:
  - Specific data memory bank on an AI Engine tile. The bank ID is relative to the tile and can take values 0,1,2,3.

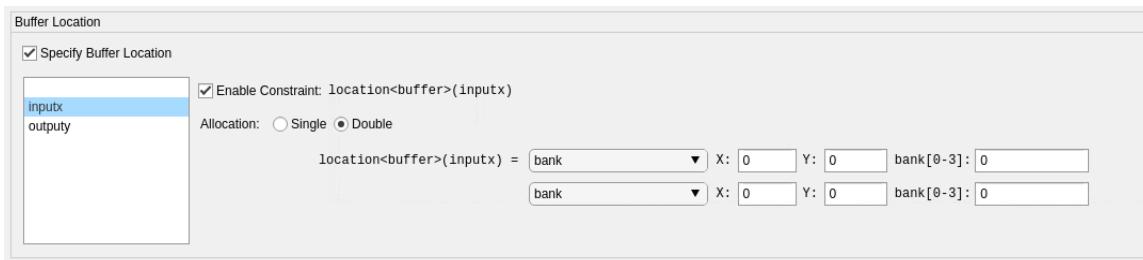
- Specific data memory address on an AI Engine tile. The offset address is relative to the tile starting at zero with a maximum value of 32768 (32K).
- Specific data memory address offset. The offset address is between 0 and 32768 (32K) and is relative to a tile allocated by the compiler.
- Specific buffer location to be on the same bank as that of one or more other port buffers. This ensures that the buffer can be accessed by other kernel without requiring a DMA.

**Figure 205: Constrain Buffer Location**



- You can constrain the location of double buffers attached to a port that are to be placed on a specific address or a bank id.

**Figure 206: Constrain Double Buffer Location**




---

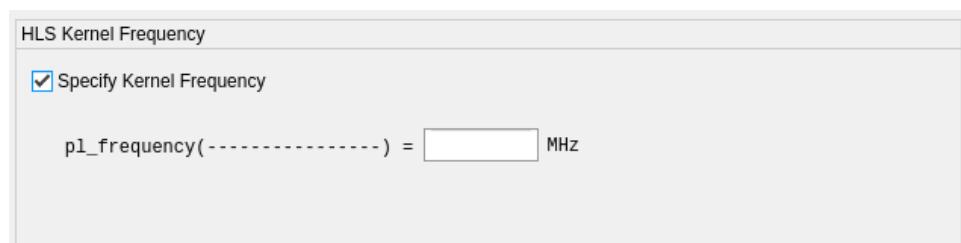
**IMPORTANT!** The non-collocation constraint (that is, specifying where the buffer should not be mapped to) is allowed only for single buffer.

---

## HLS Kernel Frequency

This constraint allows you to specify the clock frequency (in MHz) of the PL Kernel.

**Figure 207: Specify Kernel Frequency**



## Graph Bounding Box

This bounding box constraint specifies a rectangular bounding box for a graph to be placed in AI Engine tiles, between columns from `column_min` to `column_max` and rows from `row_min` to `row_max`. Multiple bounding box location constraints can be set to specify an irregular shape bounding region.

*Figure 208: Specify Bounding Box*



By default, the Specify Bounding Box option is deselected. Select this to specify the boundary location of the graph. Further, you can also enable the Specify Another Bounding Box option to specify multiple bounding regions.

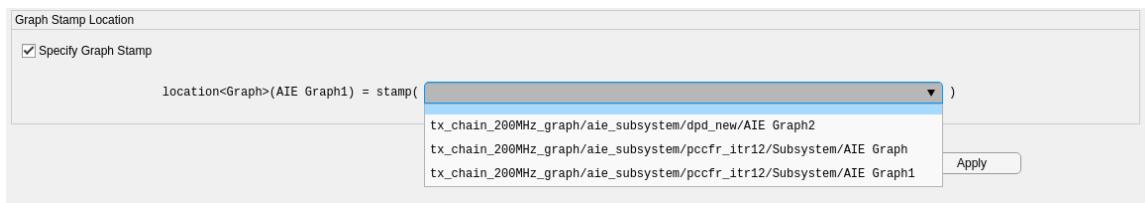
## Graph Stamp Location

The graph stamp location constraint can be used when the same graph has multiple instances that can be constrained to the same geometry in AI Engine. There are two main advantages of using this constraint:

- When the same graph is instantiated multiple times, the throughput should be same. Because of the differences in routing, throughput might not be exactly identical. However, it will be much closer when stamping is used.
- The runtime required will be significantly less because the AI Engine compiler only solves a reference graph instead of the entire design.

By default, the Specify Graph Stamp option is deselected. When selected, Model Composer allows you to select the graph from the drop down menu for which the graph (in lhs) can be stamped.

*Figure 209: Specify Graph Stamp*

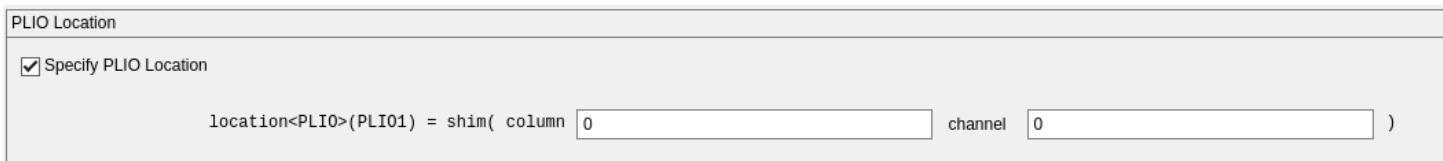


**Note:** When you copy an AI Engine block, the constraints applied on the original block might not be properly copied.

## PLIO Location

PLIOs can be constrained to a specific AI Engine array interface, which is specified by column and channel. The column and channel are zero-based. The channel is optional, and if omitted, the compiler selects the optimal channel.

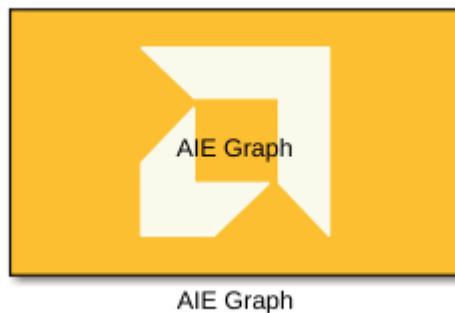
Figure 210: Specify PLIO Location



## Importing AI Engine Graphs

As discussed in [AI Engine Graphs](#), a graph is a connection of different compute kernel functions. Unlike when importing kernels, where a kernel function is imported as a block into Vitis Model Composer, in this case, graph code is imported as a block. To import the graph as block into Vitis Model Composer, you need to select the AIE Graph block from the AI Engine library (shown in the following figure).

Figure 211: AIE Graph



AIE Graph

Vitis Model Composer allows connection between the AIE Graph block and the AIE Kernel block so that the whole design can be simulated in the Simulink environment.



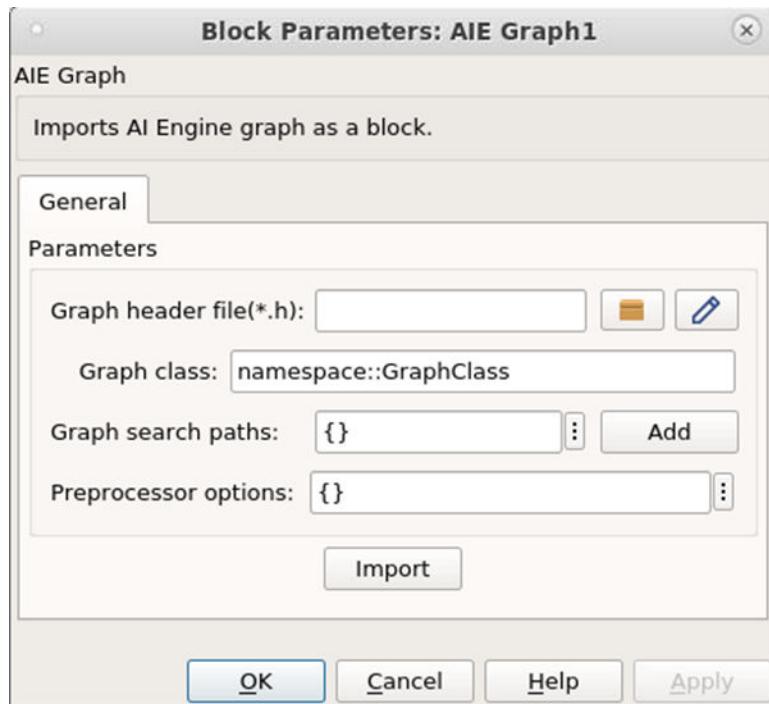
**IMPORTANT!** *It is assumed that your kernel code is already developed and that the associated kernels are properly organized.*

The AIE Graph block supports importing the AI Engine graph into Vitis Model Composer using the header file (\*.h).

## Using the Header File

To import the graph using the header file ( . h), double-click the **AIE Graph** block. Specify the graph header file, class, search path, and preprocessor options as shown in the following figure.

*Figure 212: AIE Graph Block Parameters Dialog Box*



The following table provides further details including names and descriptions of each parameter.

*Table 32: AIE Graph Block Parameters*

Parameter Name	Parameter Type	Criticality	Description
Graph application file (*.h)	String	Mandatory	Specify the file ( . h), where the application graph class is defined and the Adaptive Data Flow (ADF) header (adf.h), kernel function prototypes are included.
Graph class	String	Mandatory	Specify the name of the graph class.
Graph Search paths	Vector of strings	Mandatory	Specify search paths where header files, kernels, and other include files can be found and included for simulation. The search path \$XILINX_VITIS/adf/include (where adf.h is defined) is included by default and does not need to be specified.
Preprocessor options		Optional	Optional preprocessor arguments for downstream compilation with specific preprocessor options. The following preprocessor option formats are accepted and multiple can be selected: '-D<name>' and '-D<name>=<definition>'. That is, the optional argument must begin with the '-D' string and if the optional <definition> value is not provided, it is assumed to be 1.

```
graph.h

#ifndef __XMC_PROJ_H__
#define __XMC_PROJ_H__

#include <adf.h>
#include "simple.h"

class Proj_base : public adf::graph {
public:
    adf::kernel AIE_Kernel;

public:
    adf::input_port In1, In2;
    adf::output_port Out1, Out2;

    Proj_base() {
        // create kernel AIE_Kernel
        AIE_Kernel = adf::kernel::create(simple_comp_1);
        adf::source(AIE_Kernel) = "simple.cc";

        // create kernel constraints AIE_Kernel
        adf::runtime<ratio>( AIE_Kernel ) = 0.9;

        // create nets to specify connections
        adf::connect<adf::stream> net0 (In1, AIE_Kernel.in[0]);
        adf::connect<adf::stream> net1 (In2, AIE_Kernel.in[1]);
        adf::connect<adf::stream> net2 (AIE_Kernel.out[0], Out1);
        adf::connect<adf::stream> net3 (AIE_Kernel.out[1], Out2);
    }
};

class Proj : public adf::graph {
public:
    Proj_base mygraph;

public:
    adf::input_plio In1, In2;
    adf::output_plio Out1, Out2;

    Proj() {
        In1 = adf::input_plio::create("In1",
            adf::plio_32_bits,
            "./data/input/In1.txt");

        In2 = adf::input_plio::create("In2",
            adf::plio_32_bits,
            "./data/input/In2.txt");

        Out1 = adf::output_plio::create("Out1",
            adf::plio_32_bits,
            "Out1.txt");

        Out2 = adf::output_plio::create("Out2",
            adf::plio_32_bits,
            "Out2.txt");

        adf::connect<> (In1.out[0], mygraph.In1);
        adf::connect<> (In2.out[0], mygraph.In2);
        adf::connect<> (mygraph.Out1, Out1.in[0]);
    }
};
```

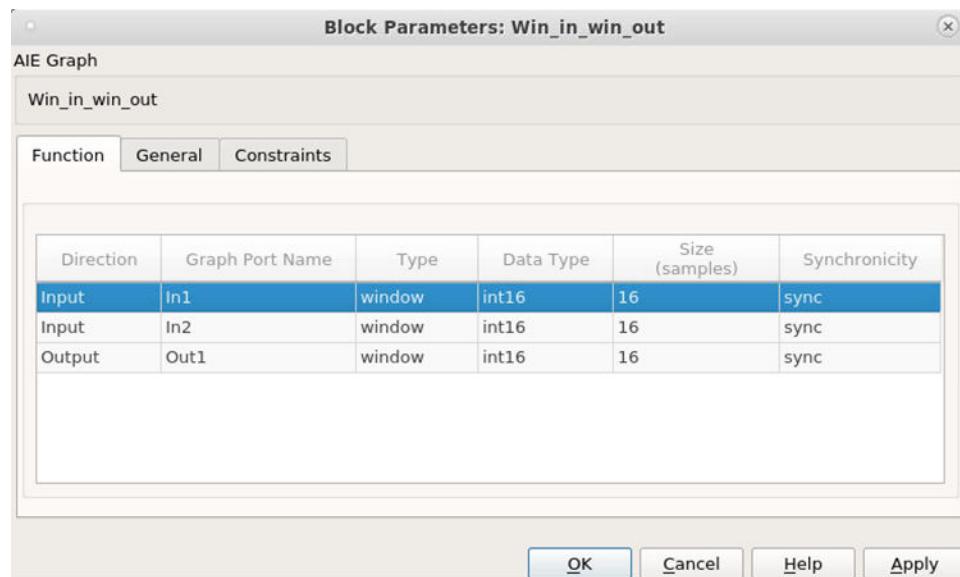
```
        adf::connect< > (mygraph.Out2, Out2.in[0]);  
    }  
};  
  
#endif // __XMC_PROJ_H__
```

**Note:** It is not allowed to import the graph class that has PLIO attributes specified. Model Composer will throw an appropriate error if you try to do so. In this case, use the `Proj_base` class to import the graph.

When the GUI parameters are updated, click the **Import** button.

The Block Parameters dialog box updates as shown in the following figure. This contains the port direction and type.

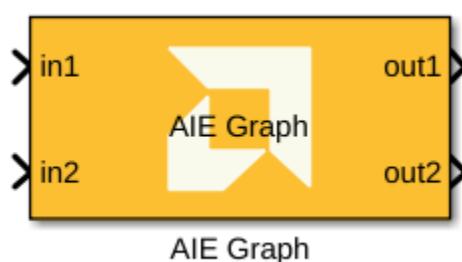
Figure 213: Block Parameters Dialog Box



Parameters such as the Graph Port Name, Data Type etc. are automatically updated from the graph code.

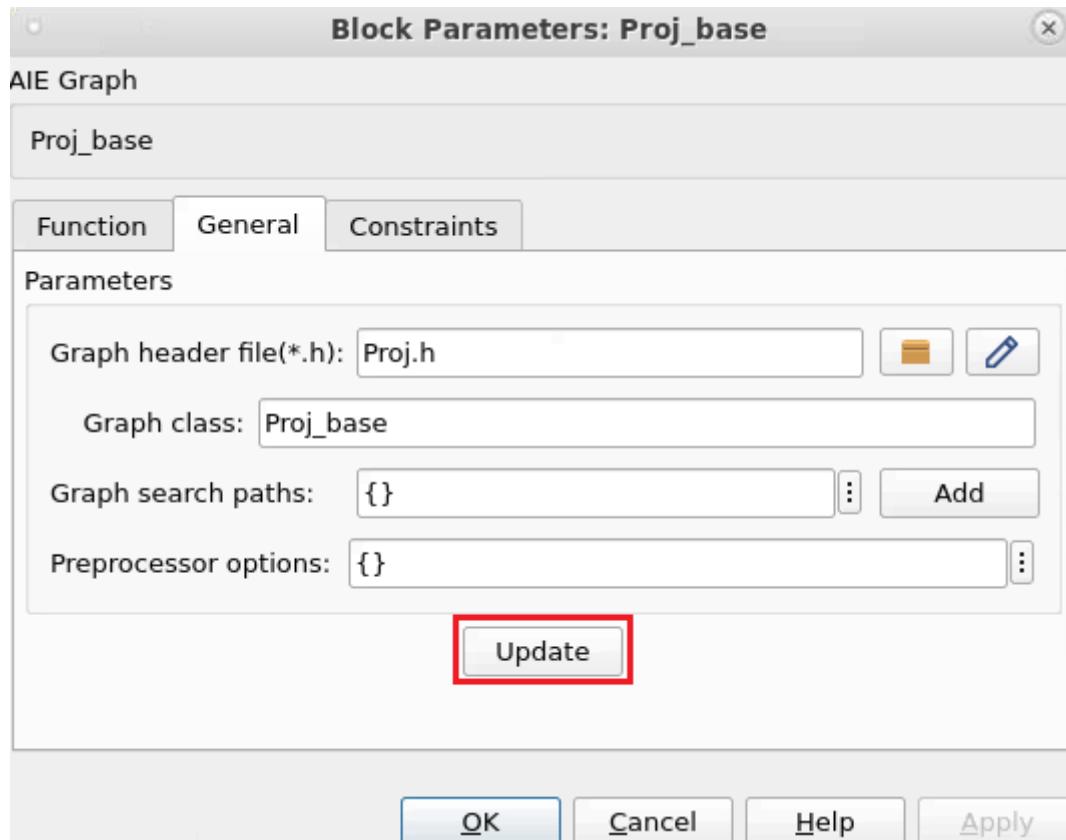
The AIE Graph block GUI interface with input and output ports is as shown in the following figure.

Figure 214: AIE Graph Block



In the General tab, the Import button changes to Update, enabling further update of block parameters.

Figure 215: Update Button



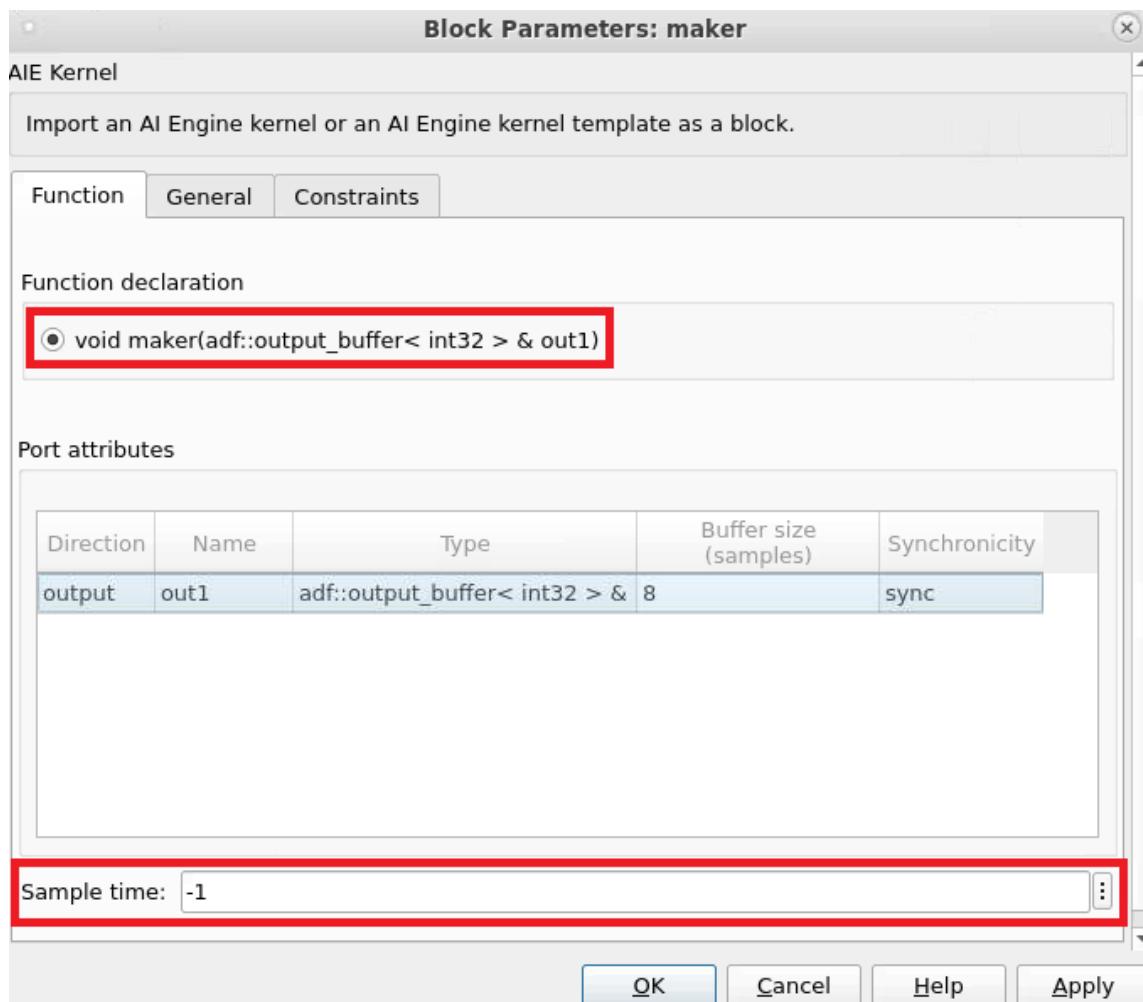
### ***Importing Kernels and Graphs as Source Blocks***

Vitis Model Composer supports importing kernels and graphs as a source blocks (i.e., user-defined functions with no input ports) into the design. These source blocks have inherited sample time and work only when the model has at least one another block with a non-inherited sample time that Simulink can use for sample time propagation. This requires you to add a block (for example a Constant block) in the model and specify a valid sample time. Alternatively, you can explicitly specify the sample time for the imported source blocks similarly to the other source blocks such as Constant, RTP Source blocks etc. from Simulink and AI Engine libraries respectively.

The AIE Kernel, AIE Class Kernel and AIE Graph blocks from the AI Engine/User-Defined Functions library supports importing the code as a source block. When you try to import the kernel into the Vitis Model Composer that has only output ports, the tool identifies that as a source block and adds the Sample time parameter to the Function tab in Block Parameters dialog box as shown in the following figure.

**Note:** The Sample time parameter is not visible when the kernel code has at least one input port.

Figure 216: Block Parameters: maker

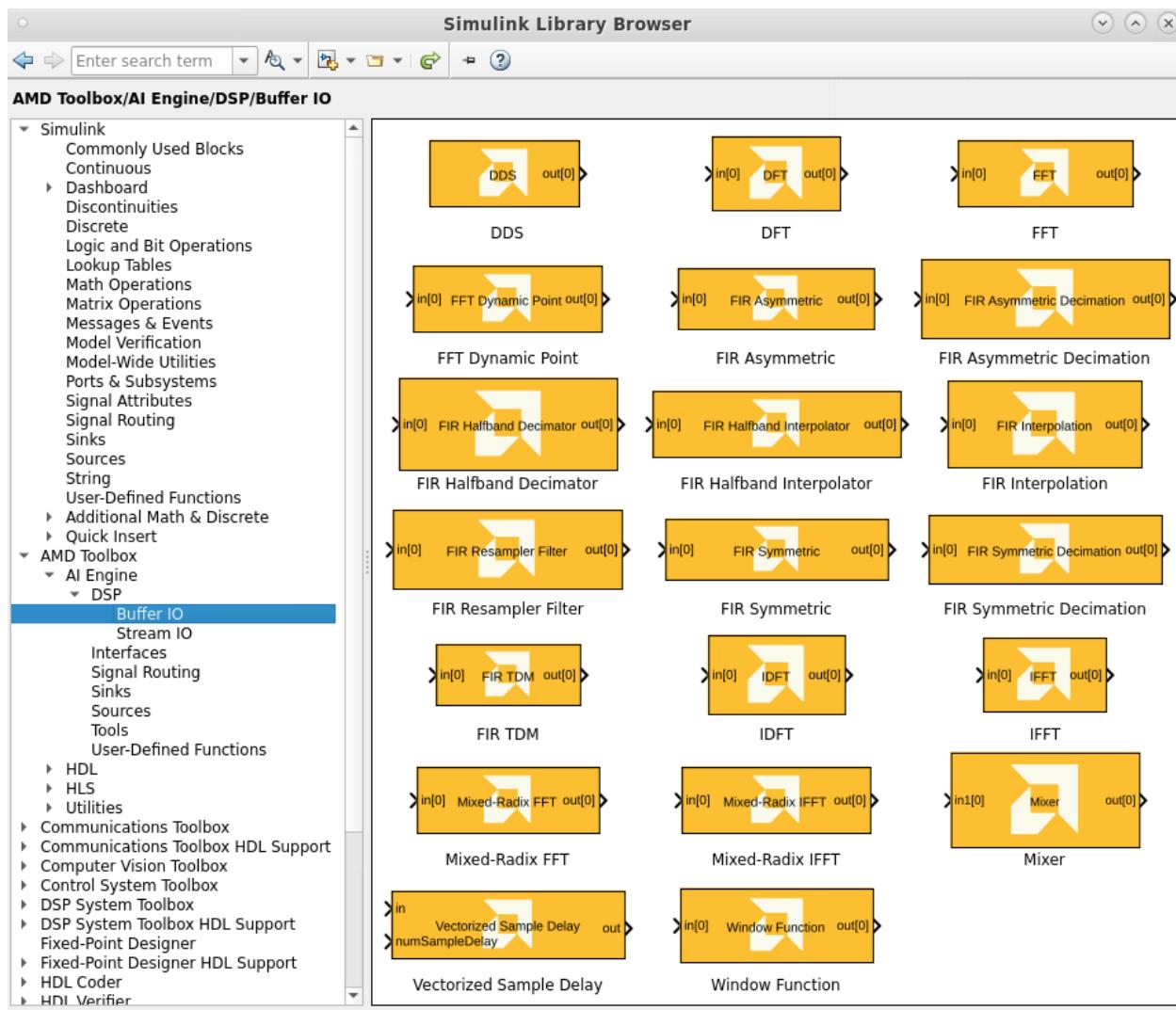


The default value of the Sample time is -1 which indicates the sample time is inherited. Valid sample time values must be positive, real scalar, or -1.

## AI Engine DSPLib

The AMD Vitis™ DSP Library (DSPLib) is a library of commonly used DSP functions optimized for AI Engines. To facilitate the use of these functions in a design, AMD Vitis™ Model Composer provides different DSPLib functions as blocks within the AMD Toolbox/AI Engine/DSP library. You can conveniently drag and drop one of these blocks into your model from the Simulink Library browser and configure the block.

Figure 217: Simulink Library Browser



Vitis Model Composer provides a copy of DSPLib functions which can be used as is. Optionally, you can download DSPLib functions from the online GitHub repository and use the MATLAB command `xmcLibraryPath` to set the path to the DSPLib library.

For more information on using the MATLAB utility, refer to [xmcLibraryPath](#).

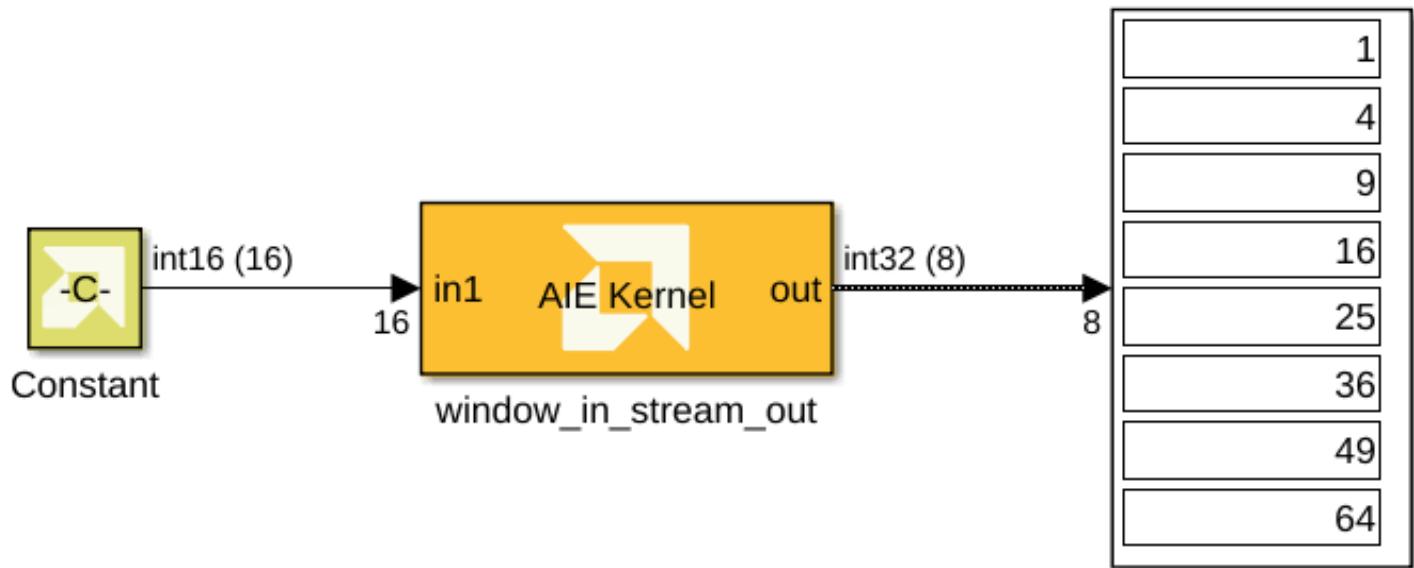
# Setting Signal Size to Avoid Buffer Overflow

The Signal Size field on the AI Engine import block masks only applies to kernels with stream or cascade outputs. Moreover, it has no implementation significance and it is only meaningful for simulation purposes in the Simulink environment. This section provides more in-depth knowledge of what Signal Size is and how to set it.

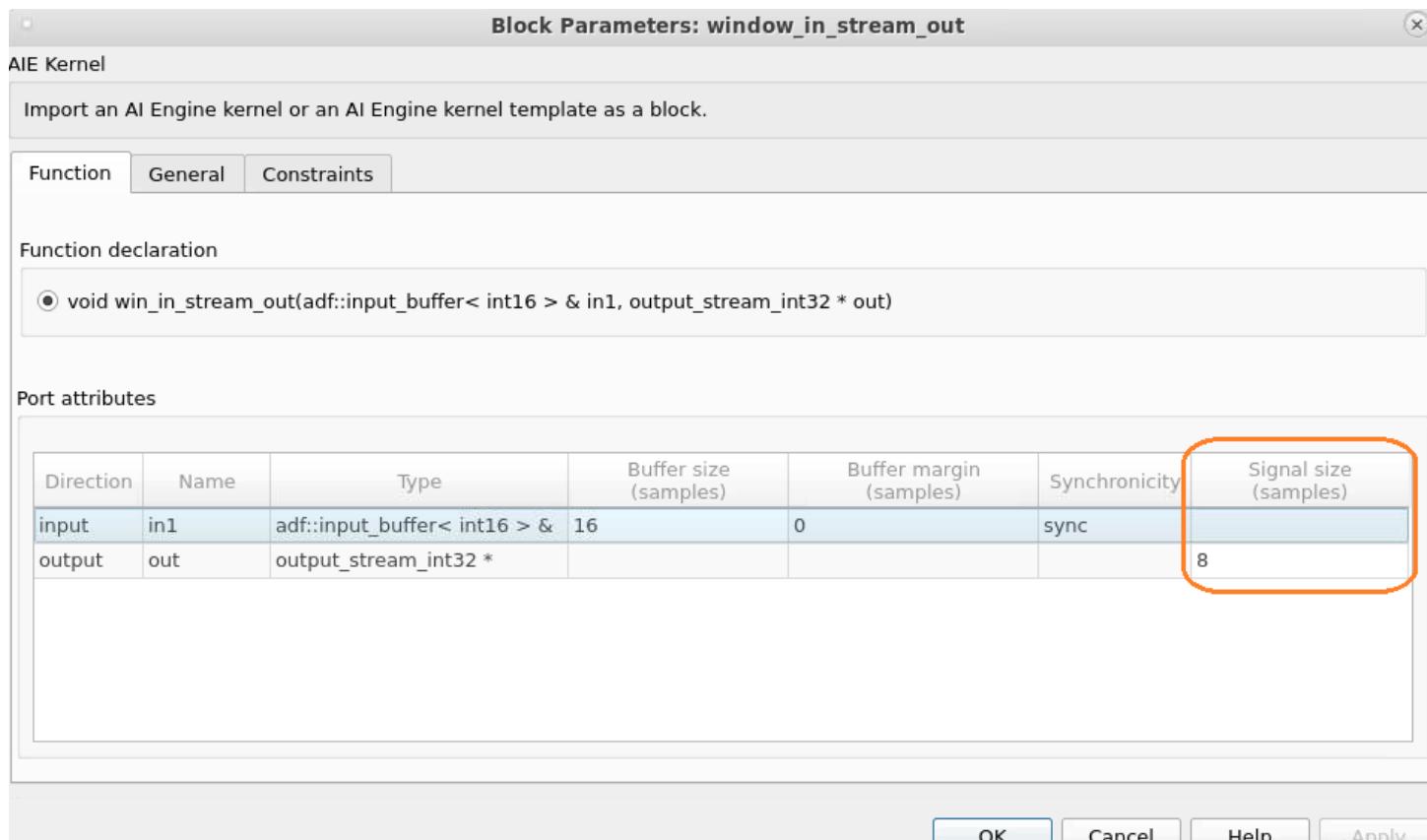
Start with a very simple kernel with buffer input and stream output. The kernel code is as follows:

```
void win_in_stream_out(input_buffer<int16> & in1, output_stream<int32> * out) {
    int16 val;
    auto pIn = aie::begin(in1);
    for (unsigned i=0; i<16; i++) {
        val = *pIn++;
        int32 squaring = val * val;
        writeincr(out, squaring);
    }
}
```

Figure 218: AIE Kernel: Window Input/Stream Output



This kernel expects a buffer of size 16 and at every invocation of this kernel, 16 output samples are generated. Import this kernel into Simulink using the AIE Kernel block. The mask for the block is shown in the following figure.

*Figure 219: Block Parameters: window\_in\_stream\_out*

Regardless of what value you set the signal size to, it does not affect the numerical output. For this example, you will generally set the signal size to 16 because every invocation of the kernel produces 16 samples. In this case, the output of this block will be a variable size signal of maximum size 16 (equal to the signal size) and each output will contain 16 samples. However, if for example you set the signal size to 32, the output of the block will be a variable size signal with a maximum size of 32, but each output will only contain 16 samples.

What if you set the signal size to a number smaller than 16, for example to 8? In this case, similar to the previous cases, the output will be a variable size signal of maximum size of 8. As mentioned previously, at each invocation of the kernel, the kernel produces 16 samples. Eight of these samples will be put out by the block. The other eight are stored in an internal buffer in the block. If you call the kernel too many times, eventually the internal buffer of the block will fill up and you will see a buffer overflow error as shown in the following figure.

**Figure 220: Buffer Overflow Error**

An error occurred while running the simulation and the simulation was terminated

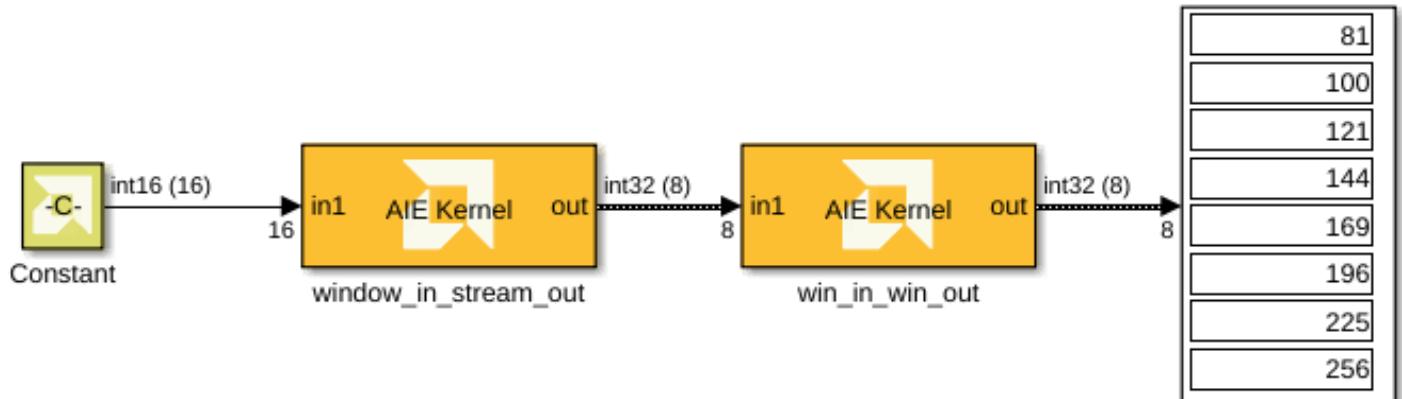
Caused by:

- Error reported by S-function 'XilinxLibrary' in '[maker test/window in stream out](#)':
   
ERROR-XMC-9003: Imminent buffer overflow on input in1. Tried to write 32 bytes but succeeded in writing only 4 bytes. For suggestions on how to resolve this issue, see the Quick Guide: [How to fix buffer overflow errors with AI Engine and HLS Kernels?](#)

Component: Simulink | Category: Block error

This is a trivial example. You might contend that there is no reason to set the signal size to anything less than 16, and that is correct. Now examine a model with two AI Engine kernels. Connect the output of the kernel previously created to another AI Engine kernel with buffer input and buffer output. The code for this second kernel is as follows:

```
void win_in_win_out(input_buffer<int32> & inw, output_buffer<int32> & outw)
{
    int32 temp;
    auto pIn = aie::begin(inw);
    auto pOut = aie::begin(outw);
    for (unsigned i=0; i<8; i++) {
        temp = *pIn++;
        *pOut++ = temp;
    }
}
```

**Figure 221: Two AIE Kernels**

This kernel requires an input buffer of size 8 and produces a buffer size of 8. Now consider two scenarios. First consider a case in which the first block has the signal size set to 16. As mentioned previously, with a signal size of 16, the buffer for the first block will not overflow. But now examine the second block more closely. The second kernel upon receiving 16 samples, will get invoked twice. Each time, it produces eight samples for a total of 16 samples. However, because the output size is 8, the block will produce eight samples and store the other eight in the internal buffer. As before, if you run this model for long enough, the buffer for the second block will overflow and simulation will stop.

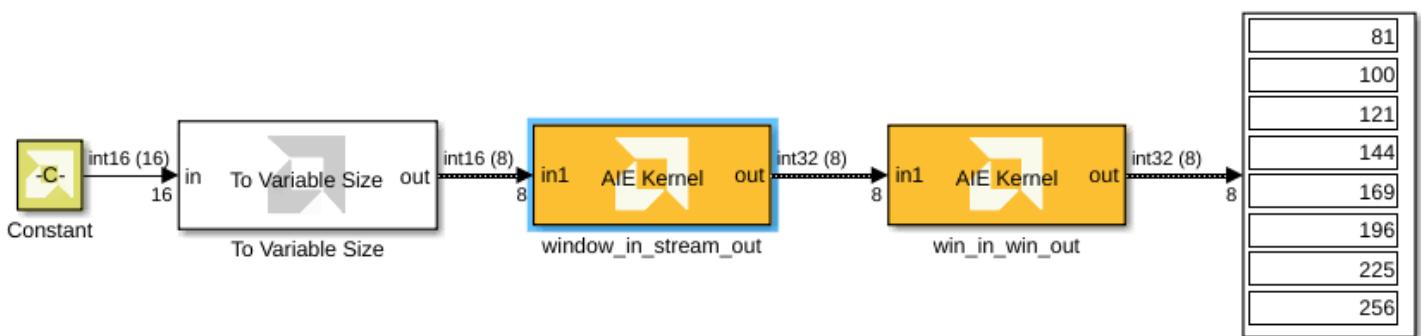
In another scenario, to avoid an overflow, you might set the signal size for the first block to 8. This will avoid an overflow in the second block. However as mentioned previously, now the buffer for the first block will overflow. So how can you get out of this situation?

The buffer overflows because you are feeding more data to the blocks than the blocks can process. If you reduce the rate, the kernels will be able to process any excess data in the buffers and as such prevent the overflow. Now look into this more carefully.

Assume the simulation has been running for a while and the first block's buffer is not empty. If you somehow stop feeding data to the first block, every time Simulink calls the first block, the kernel will not be invoked (there is no input data), but because there are samples in the buffer, the block will continue to produce samples (eight at a time) until the buffer empties out after which it will produce an empty variable size signal.

This information should help you avoid buffer overflow. Instead of stopping the input as suggested above, simply reduce the flow of the data into the first block. One way of doing this is to use a To Variable Size block from AI Engine/Tools and set the Output size on the block mask to a number smaller than the size of the input. The following figure depicts the same design shown above but with a To Variable Size block at its input.

*Figure 222: Two AIE Kernels: Buffer Block Input*



In this design, because fewer samples are being fed to the first block at any given call to the block, the buffers will not overflow. Note that the output of this model will be different from the model without the buffer block because the buffer block produces zero samples at time step zero.



**TIP:** If a block with stream output is connected to a block with buffer input, set the size of the signal for the producing block to the same size as the input window for the consuming block.



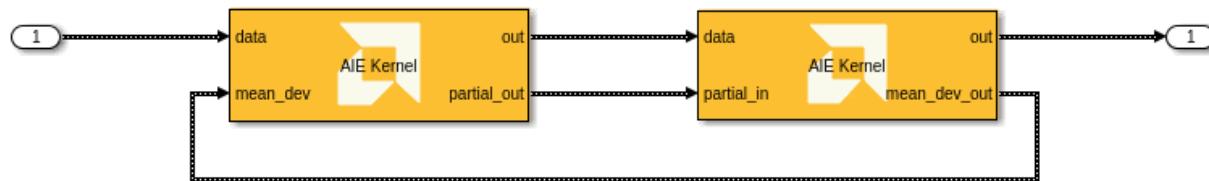
**TIP:** To avoid buffer overflow, reduce the rate you feed data to the system using a buffer block.

## AI Engine Designs with Feedback Loops

The AI Engine architecture allows you to construct designs with feedback loops; that is, when kernels are connected in a cascade, an output of one kernel can be routed to the input of an earlier kernel. This structure can result in an algebraic loop in Simulink.

Consider the following AI Engine design, where there is a feedback loop between the kernels:

Figure 223: Design with Feedback Loop



When the model is updated, this design produces a Simulink error:

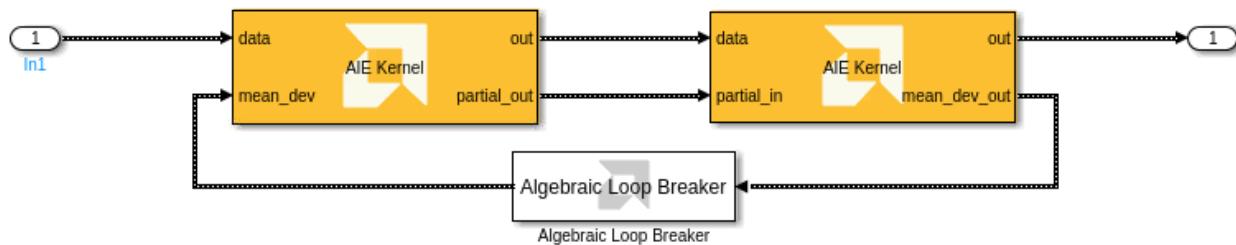
```
` Cannot solve algebraic loop involving 'block' because it consists
of blocks that cannot be assigned algebraic variables, such as
blocks with discrete-valued outputs, blocks with non-double or complex
outputs, Stateflow blocks, or nonvirtual subsystems. Consider breaking
the algebraic loop. For example, add a delay or a memory block
to the loop. To see more details about the loops use the command
Simulink.BlockDiagram.getAlgebraicLoops(bdroot)`
```

The error message suggests breaking the algebraic loop with a delay or memory block. However, these blocks will not work with variable-sized signals inside an AI Engine subsystem. To address this issue, Vitis Model Composer provides an Algebraic Loop Breaker block.

The Algebraic Loop Breaker block effectively performs a unit delay on the variable-sized signal. The initial output of this block is an empty variable size signal. On subsequent time steps, this block outputs the input signal delayed by one time step.

When the Algebraic Loop Breaker block is inserted into the feedback loop, the model updates and the design simulates as expected.

Figure 224: Design with Algebraic Loop Breaker



## Simulation and Code Generation

After a high level graphical design is created using the blocks available in the Vitis Model Composer AI Engine library, it should be simulated interactively in the Simulink environment. This process ensures the functional correctness of the design using the native Simulink functional simulator and displays the results on scopes and graphical displays. The compilation and execution times are generally short at this stage, which helps you to quickly verify the functionality and iterate over the design until the specification requirements are met. The functionally verified design can then be used to generate the dataflow graph using the Vitis Model Composer Hub block. The verification of the dataflow graph can be done using various execution targets which Vitis Model Composer supports to simulate your AI Engine application at different levels of abstraction, accuracy, and speed.

This section discusses following topics in detail:

- Running Simulink Simulation
- Code Generation
- Verifying the generated dataflow graph

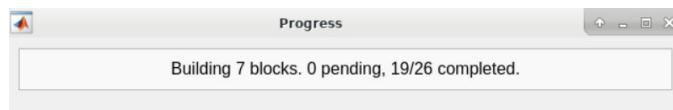
## Running Simulink Simulation

Simulation involves compiling the design and checking for any design rule violations, then executing the design to produce the outputs. You can define the inputs of the design using any Simulink source blocks and the output is analyzed either by logging the data to the workspace, or by visually viewing the results using a scope, spectrum analyzer, or display block.

Vitis Model Composer provides two MATLAB utilities `xmcVitisRead` and `xmcVitisWrite` to directly read/write data from/to the files that are formatted for AIE Simulator and/or x86 Simulator. For more information on using these utilities, refer to [AI Engine Utilities](#).

Clicking the **Simulate model** icon in the Simulink simulation tool bar, compiles all the kernels and graphs in the design. You can monitor the status from the Progress window, which displays after simulation begins (see the following figure).

Figure 225: Compilation Status



The Progress window displays only when you are compiling a design for the first time. When you simulate the model again, unless you make any changes to the imported kernel or graph, Model Composer will use the cached entry for the block to run the simulation faster. For example, assume you have three kernels (`add2`, `add3`, `add4`) in your design and you run the simulation for the first time. In that case, all the three kernels get compiled. When you change the `add3` kernel code and try to simulate again, only the changed `add3` kernel gets re-compiled and the cached entries for `add2` and `add4` are used for faster simulation.

To manage the simulation cache in Model Composer, use the commands described in [Managing the HLS Block Cache](#) from the MATLAB command prompt.

When simulation is complete, you can review the results by connecting any of the Simulink sink blocks to appropriate points in your design.

## Using Vitis Debugger

The Vitis IDE provides a debug environment that can be invoked directly from Vitis Model Composer.

The Vitis IDE debug environment has many features found in traditional GUI-based debug environments, such as GDB. You can add break points to the code, step over or step into specific lines of codes, loops, or functions, and examine the state of variables and force them to specific values.

To invoke the Vitis Debugger from Vitis Model Composer, do the following:

1. Open the model that contains the C/C++ source code block you would like to debug.
2. Run `vmcLaunchVitisDebugger` in the MATLAB® Command Window.

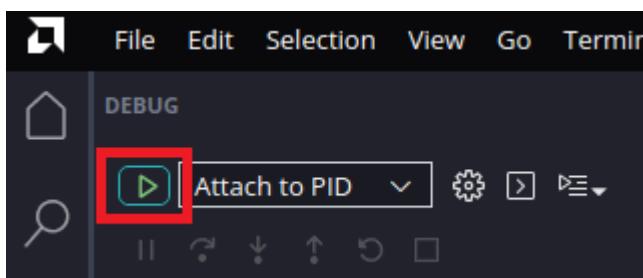
After a few moments, the Vitis IDE loads and the following message appears.

```
Opening Vitis Debugger...
*****
* Follow these steps in AMD Vitis:
* (1) On the leftmost toolbar, click on the 'Debug' button.
* (2) Inside the 'Debug' panel, select 'Attach to PID' and click on 'Start Debugging'.
* (3) Wait for the debugger to attach the process.
* (4) You can open source files in Vitis and set breakpoints at this stage.
*
* After completing the above steps, go back to your model and simulate it. *
```

3. Select the **Debug** button on the Vitis IDE menu.



4. Inside the Debug panel, make sure Attach to PID is selected in the drop-down menu. Click **Start Debugging**.



Wait for the debugger to finish attaching to process. When finished, the status bar at the bottom of the Vitis window turns orange and indicate clangd: idle.



5. In the Vitis Debugger, open the source file you would like to debug. (Select **File**→**Open File...**)
6. On the source code line where you would like the debugger to stop, place a breakpoint by clicking to the left of the line number. A red circle displays when the breakpoint has been successfully placed.

```
4 void SingleWindow::detect_Singl
5
6 {
7     count++;
8     if (count % AVERAGE ==0)
9     {
10         prevmode = false;
```

A screenshot of the Vitis Debugger code editor. A red dot is positioned at the start of line 7, indicating it is a breakpoint. The entire line of code is highlighted with a green background, and the cursor is visible at the end of line 10.

**Note:** When debugging HLS kernels on Windows, it is not possible to set breakpoints in the source code editor.

Instead, the breakpoints must be set in the `_ide/launch.json` file. Modify the `autorun` section of the file by adding a `break` statement, similar to the code snippet below. The file must be saved and the debugger restarted for the breakpoint to enable.

```
"autorun": [
    "handle SIGSEGV nostop noprint",
    "set breakpoint pending on",
    "break detectSingleWindow.cpp:82"
]
```

7. Return to Simulink and run the model.

```
4 void SingleWindow::detect_Sin
5
6 {
7     count++;
8     if (count % AVERAGE ==0)
9     {
10         prevmode = false;
```

A screenshot of the Vitis Debugger code editor. A red dot is positioned at the start of line 7, indicating it is a breakpoint. The entire line of code is highlighted with a green background, and the cursor is visible at the end of line 10.

You can now use the Debug controls on the left side of the screen to control program execution, view the call stack and data variables, and enable and disable breakpoints. For more information on the capabilities of the Vitis Debugger, refer to *Vitis Unified Software Platform Documentation: Embedded Software Development (UG1400)*.

When finished debugging, click **Stop** in the Debug panel.



When model execution hits the breakpoint that you set, the Vitis Debugger pauses execution on that line, which becomes highlighted.

## Code Generation

When the design is functionally verified in Simulink, you can generate the dataflow graph from the design. It is necessary to encapsulate the AI Engine blocks into a subsystem. To understand more about creating a top-level subsystem, refer to [Creating a Top-Level Subsystem Module](#).



**IMPORTANT!** To generate output from the AI Engine model, only blocks from the Vitis Model Composer AI Engine library and a limited set of Simulink blocks can be used in the subsystem that is instantiated at the top-level of the design. Refer to [Connecting Source and Sink Blocks](#) for more details about the blocks that are supported inside the subsystem.

### **Vitis Model Composer Hub Block for AI Engine Code Generation**

An AI Engine model in Model Composer requires the addition of the Vitis Model Composer Hub block to configure compilation and generation of AI Engine output. For more details about adding the Model Composer Hub block into the design and associated features, refer to [Vitis Model Composer Hub](#).

This section discusses only exporting an AI Engine design from Model Composer. For running and analyzing the generated AI Engine's code, refer to [Verification of AI Engine Code](#).

The Model Composer Hub block settings specific to AI Engine compilation targets are shown in the following figures.

Figure 226: Model Composer Hub AI Engine Settings

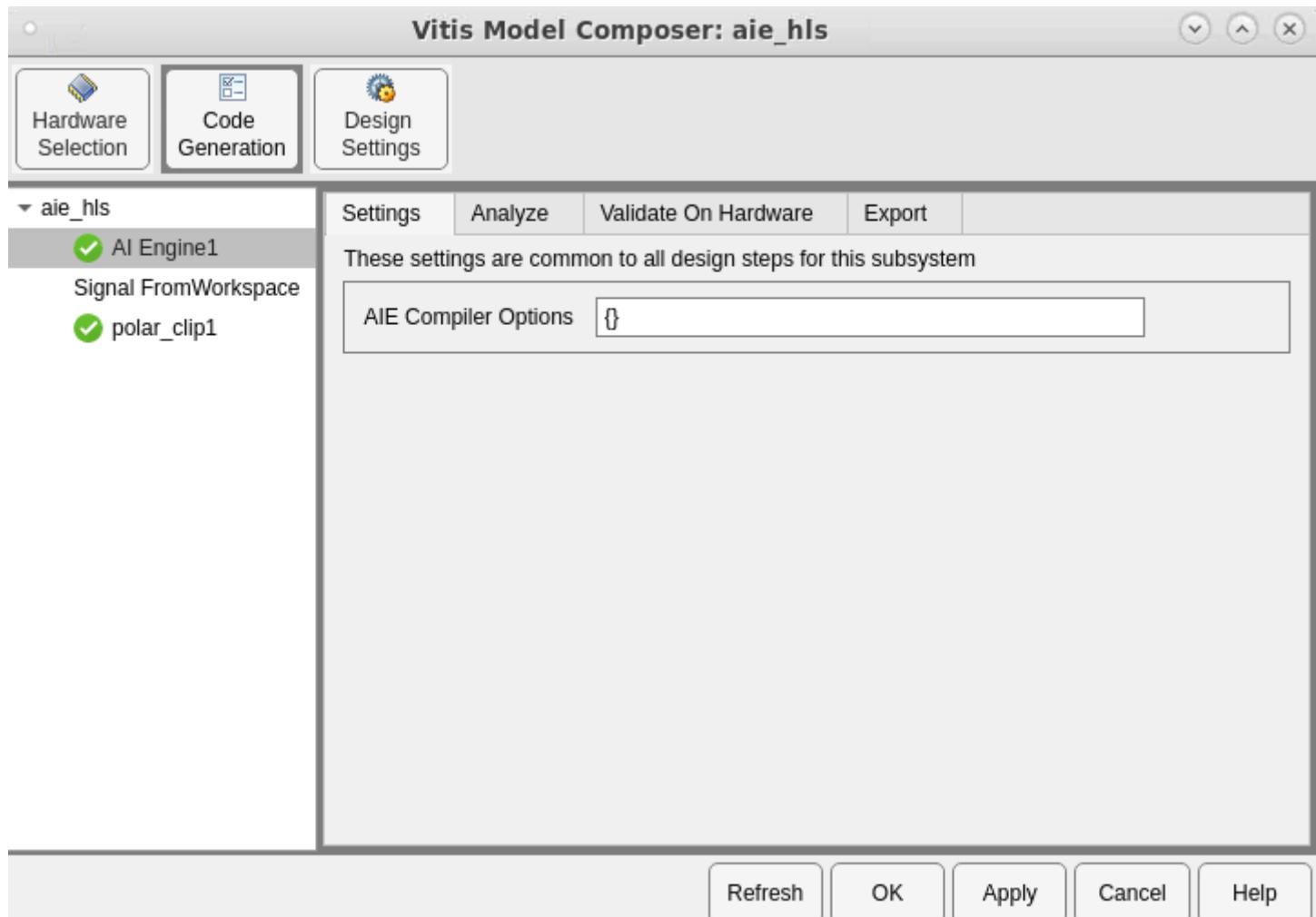
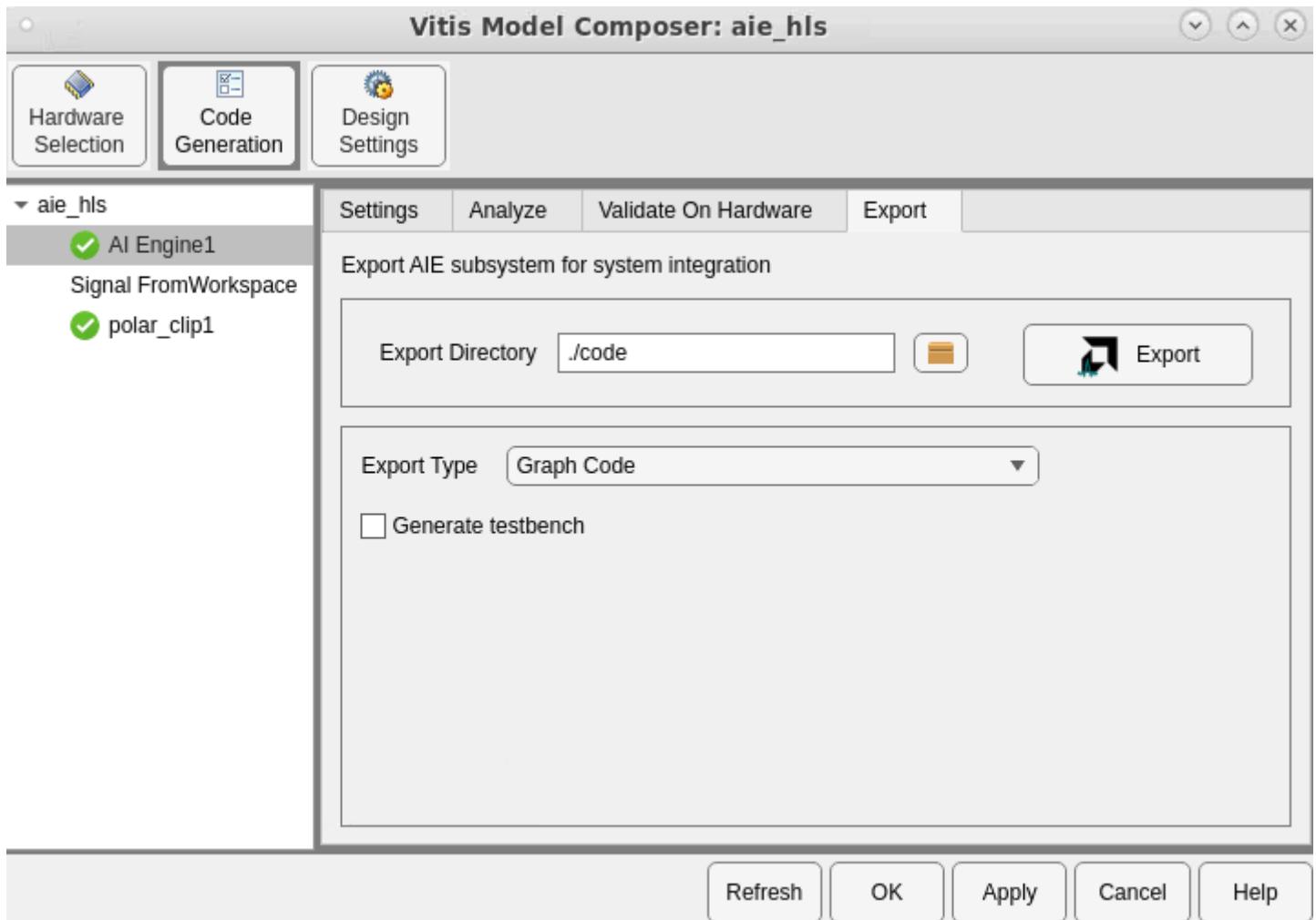


Figure 227: Model Composer Hub AI Engine Export



Select the subsystem for which to generate code from the list on the left.

On the Settings tab, you can specify a cell array of AI Engine compiler options using the AIE Compiler Options edit button. This provides a method to control the compiler debug options, execution target options, file options and so on. Examples as follows:

- To control the debug option log-levels, you can specify the string `{ '--log-level=5' }` in the AIE Compiler Options field.
- For issues related to the stack size or heap size in the downstream AI Engine flows, you can increase the size by adding `--stacksize=<int>` and `--heapsize=<int>` in the AIE Compiler options field.

On the Export tab, specify the desired **Export Type** and click **Export**.

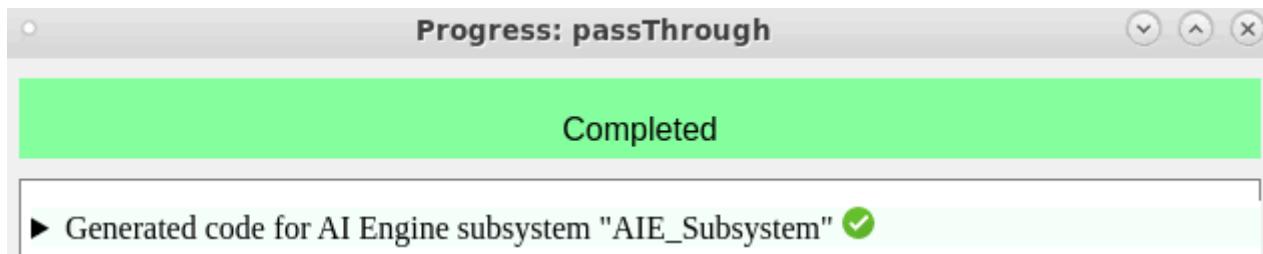


**IMPORTANT!** You can enable the **Create testbench** option to log the test data at input and output.

Model Composer displays the progress of code generation in the Progress window.

When Vitis Model Composer has completed exporting the design, it displays the status message **Completed** in the Progress window as shown in the following figure.

Figure 228: Done Code Generation



## Output Directory

A new directory gets created with the name specified in the Code Directory field in the Vitis Model Composer Hub block. There are various sub-directories in the `code` directory but the `ip/<subsystem name>/src/` directory and the `Makefile` which are highlighted in figure below, are of interest for this section. The details about other sub-directories are explained in subsequent topics.

Figure 229: Target Directory

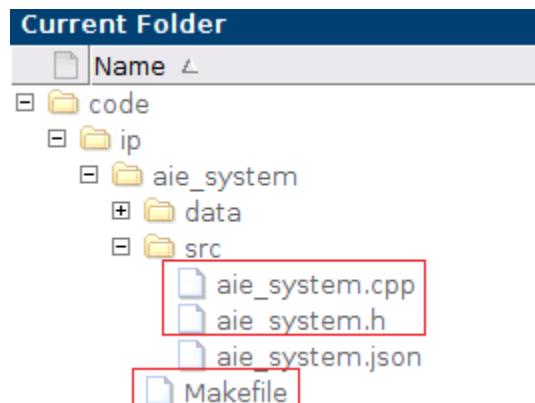
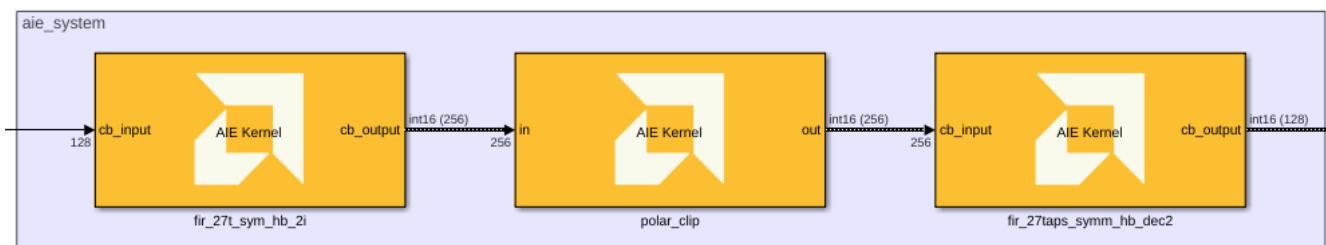


Table 33: File/Directory Descriptions

File/Directory	File/Sub-directory	Description
ip/ Subsystem_Name/src	Subsystem_Name.h	Header file that specifies the AI Engine dataflow graph. <b>Note:</b> Subsystem_Name is a unique string derived from the top-level subsystem specified in the Model Composer model.
	Subsystem_Name.cpp	Test bench to simulate the dataflow graph specified in Subsystem_Name.h.
	Makefile	This file contains the commands to compile and simulate the dataflow graph. For more details on how to use the Makefile to run simulation from command line, refer to the section <a href="#">Running Simulation using the Makefile</a> .

The files generated by Vitis Model Composer in the `src` directory reflect the contents and hierarchy of the subsystem that gets compiled. In this case, assume the subsystem is `aie_system` which is derived from the design in [Vitis Model Composer Tutorials](#). The following figure shows the interconnection of imported kernel functions as blocks, encapsulated as a subsystem.

Figure 230: Block Interconnection



The generated code for the subsystem `aie_system` is as follows.

```

aie_system.h

#ifndef __XMC_AIE_SYSTEM_H__
#define __XMC_AIE_SYSTEM_H__

#include <adf.h>
#include "kernels/inc/hb_27_2i.h"
#include "kernels/inc/polar_clip.h"
#include "kernels/inc/hb_27_2d.h"

class Aie_system_base : public adf::graph {
public:
    adf::kernel fir_27t_sym_hb_2i_0;
    adf::kernel polar_clip_0;
    adf::kernel fir_27taps_symm_hb_dec2_0;

public:
    adf::input_port In1;
    adf::output_port Out1;
}

```

```
Aie_system_base() {
    // create kernel fir_27t_sym_hb_2i_0
    fir_27t_sym_hb_2i_0 = adf::kernel::create(fir_27t_sym_hb_2i);
    adf::source(fir_27t_sym_hb_2i_0) = "kernels/src/hb_27_2i.cpp";

    // create kernel polar_clip_0
    polar_clip_0 = adf::kernel::create(polar_clip);
    adf::source(polar_clip_0) = "kernels/src/polar_clip.cpp";

    // create kernel fir_27taps_symm_hb_dec2_0
    fir_27taps_symm_hb_dec2_0 =
adf::kernel::create(fir_27taps_symm_hb_dec2);
    adf::source(fir_27taps_symm_hb_dec2_0) = "kernels/src/hb_27_2d.cpp";

    // create kernel constraints fir_27t_sym_hb_2i_0
    adf::runtime<ratio>( fir_27t_sym_hb_2i_0 ) = 0.9;

    // create kernel constraints polar_clip_0
    adf::runtime<ratio>( polar_clip_0 ) = 0.9;

    // create kernel constraints fir_27taps_symm_hb_dec2_0
    adf::runtime<ratio>( fir_27taps_symm_hb_dec2_0 ) = 0.9;

    // create nets to specify connections
    adf::connect< adf::window<512,64> > net0 (In1,
fir_27t_sym_hb_2i_0.in[0]);
    adf::connect< adf::window<1024>, adf::stream > net1
(fir_27t_sym_hb_2i_0.out[0], polar_clip_0.in[0]);
    adf::connect< adf::stream, adf::window<1024,128> > net2
(polar_clip_0.out[0], fir_27taps_symm_hb_dec2_0.in[0]);
    adf::connect< adf::window<512> > net3
(fir_27taps_symm_hb_dec2_0.out[0], Out1);
}
};

class Aie_system : public adf::graph {
public:
    Aie_system_base mygraph;

public:
    adf::input_plio In1;
    adf::output_plio Out1;

    Aie_system() {
        In1 = adf::input_plio::create("In1",
            adf::plio_32_bits,
            "./data/input/In1.txt");

        Out1 = adf::output_plio::create("Out1",
            adf::plio_32_bits,
            "Out1.txt");

        adf::connect< > (In1.out[0], mygraph.In1);
        adf::connect< > (mygraph.Out1, Out1.in[0]);
    }
};

#endif // __XMC_AIE_SYSTEM_H__
```

Vitis Model Composer automatically generates the graph header file `aie_system.h` which contains the dataflow graph corresponding to the subsystem name specified in the Hub block. The connection between the AI Engine kernel or graph as well as the configuration parameters such as window size, window margin and so on, which are specified in the AI Engine kernel block, are automatically reflected in the generated graph code.

Vitis Model Composer also generates the `aie_system.cpp` file, which is a control program that has the `main()` function defined to initialize, run, and end the simulation using the control APIs.

#### `aie_system.cpp`

```
#include "aie_system.h"

// instantiate cardano dataflow graph
Aie_system mygraph;

// initialize and run the dataflow graph
#if defined(__AIESIM__) || defined(__X86SIM__)
int main(void) {
    mygraph.init();
    mygraph.run();
    mygraph.end();
    return 0;
}
#endif
```

### Limitations

- AI Engine code cannot be generated if a top-level subsystem's `output` port is connected to a synchronous runtime parameter port.
- The value of an input runtime parameter port can only be updated once at the beginning when running the generated graph.

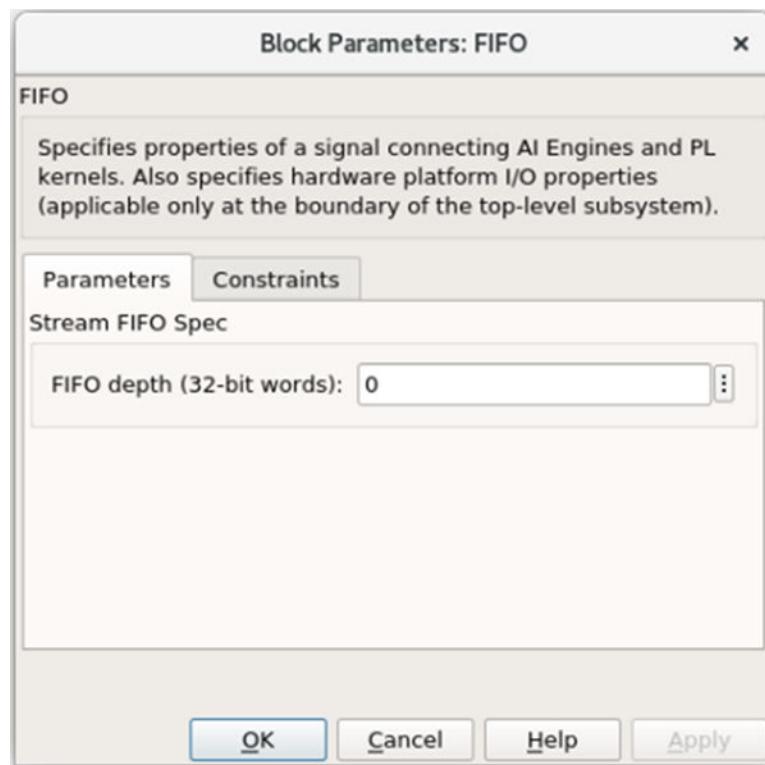
### ***Stream FIFO Depth Specification***

The AI Engine architecture uses streaming data extensively for communicating between two AI Engines, and for communicating between the AI Engine and the programmable logic (PL). This raises the potential for a resource deadlock when the data flow graph has reconvergent stream paths. If the pipeline depth of one path is longer than the other, the producer kernel can stall and might not be able to push data into the shorter path because of back pressure. At the same time, the consumer kernel is waiting to receive data on the longer path due to the lack of data. If the order of data production and consumption between two stream paths is different, a deadlock can happen even between two kernels that are directly connected with two stream paths.

Vitis Model Composer supports adding a `FIFO_DEPTH` between two AI Engine kernels or between an AI Engine and the programmable logic (PL) using the AIE FIFO block and automatically generates the graph code with `fifo_depth` constraint on a connection.

*Figure 231: AIE FIFO*

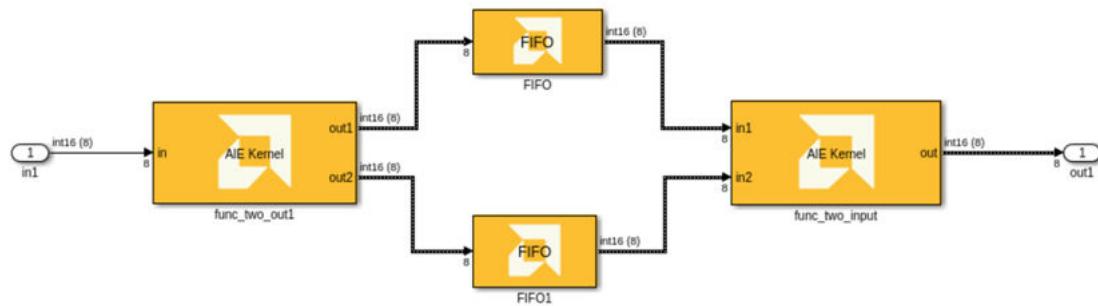
The AIE FIFO block specifies the FIFO depth on a particular path. Specifying the FIFO depth value can help avoid deadlock or stalling by creating more buffering in the paths. This block does not affect functional simulation and will only impact the generated graph code. You can set location constraints for the FIFO block using the Constraint manager. FIFOs can be either stream FIFOs, DMA FIFOs, or a combination of the two.

*Figure 232: Block Parameters: AIE FIFO*

- **Parameters:** Use this tab to specify FIFO depth.
- **Constraints:** Use this tab to set location constraints for the FIFO block using the Constraint manager.

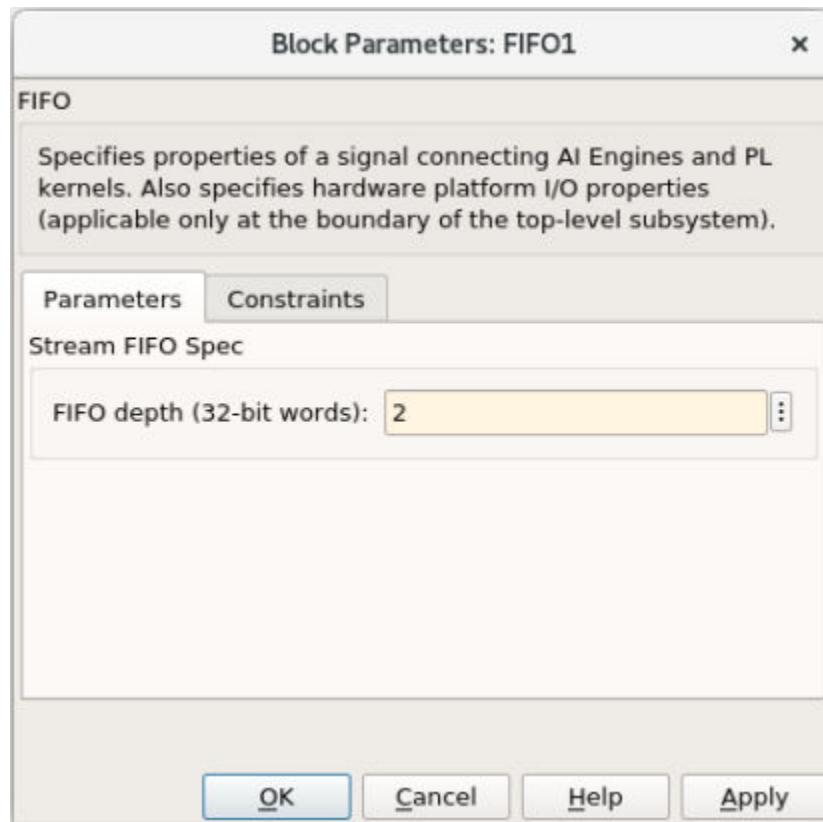
Consider the following example where the AIE FIFO blocks are connected in two stream paths between AIE Kernel blocks.

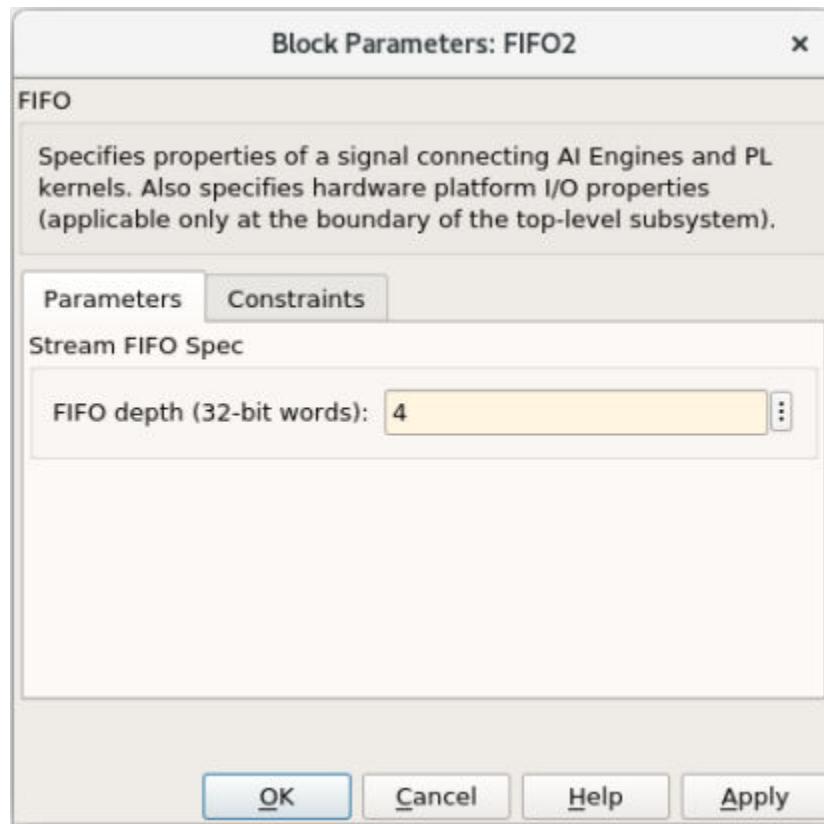
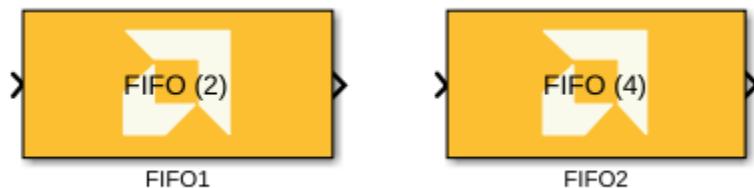
Figure 233: AIE FIFO Blocks in Two Stream Paths



The FIFO depth values are specified as 0 by default and the corresponding information is reflected on the block symbol. For example, if values 2 and 4 are specified as parameters of the two FIFO blocks, the values will update as shown in the following figures.

Figure 234: AIE FIFO



*Figure 235: AIE FIFO**Figure 236: Block Symbol Updates*

The stream FIFO values specified using the AIE Signal Spec block are automatically updated in the generated graph code (`graph.h`) with `fifo_depth` constraints as shown in the following code.

### Snippet of graph.h

```
// create nets to specify connections
adf::connect< adf::stream > net0 (In1, AIE_Kernel.in[0]);
adf::connect< adf::stream > net1 (AIE_Kernel.out[0],
AIE_Kernel1.in[0]);
adf::fifo_depth(net1) = 2;
adf::connect< adf::stream > net2 (AIE_Kernel.out[1],
AIE_Kernel1.in[1]);
adf::fifo_depth(net2) = 4;
adf::connect< adf::stream > net3 (AIE_Kernel1.out[0], Out1);
```

Notice the two `fifo_depth` constraints corresponding to the two stream paths in the design. Using this `fifo_depth` constraint on a connection is useful as it creates more buffering in paths with back pressure, and hence avoids any deadlock.



**IMPORTANT!** The AIE FIFO block can only be used in an AI Engine subsystem.

## PLIO Attributes

Typically, the AI Engine array runs at a higher clock frequency (between 1 GHz and 1.25 GHz) than the internal Programmable Logic. Within the AI Engine core the streaming data-width is 32-bit; whereas between the AI Engine interface tile and PL interface, it is 64-bit by default. To balance the throughput between AI Engine and internal programmable logic, it is desirable to pipeline enough data by choosing wider stream data paths for PL blocks. For example, to use an AI Engine array running at 32 bits / 1 GHz rate to its full potential, PL blocks can use 64 bits / 500 MHz rate or 128 bits / 250 MHz rate and so on. Such wider (> 32 bits) stream data is sequentialized automatically into 32-bit streams within the AI Engine interface tile.

## Specifying PLIOs in Model Composer Designs

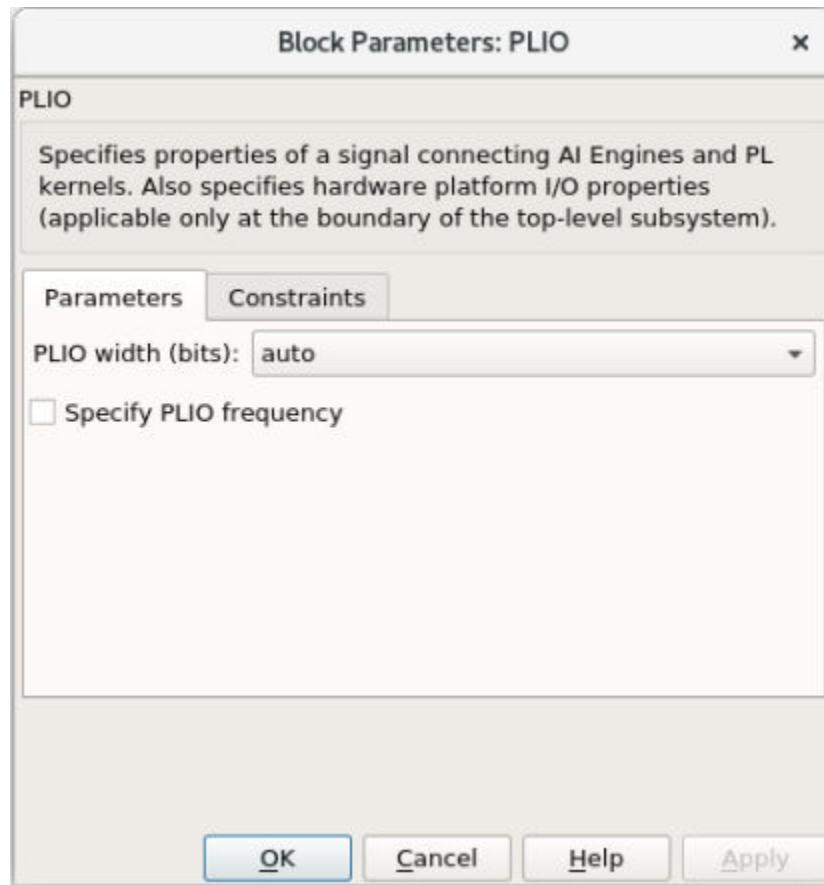
Vitis Model Composer supports specifying the PLIO width using the AIE PLIO block and making external stream connections that cross the AI Engine to PL boundary.

Figure 237: AIE Signal Spec



The AIE PLIO block supports specifying the Platform IO (PLIO) width. Specifying the PLIO width at the boundary of the AI Engine subsystem can affect the throughput of data between the AI Engine domain and the programmable logic (PL) domain. You can also set the constraints for the PLIO block using its constraint manager.

Figure 238: Block Parameters: AIE PLIO



From the Parameters tab in the AIE Signal Spec block, you can select the available PLIO width options from the drop-down menu.

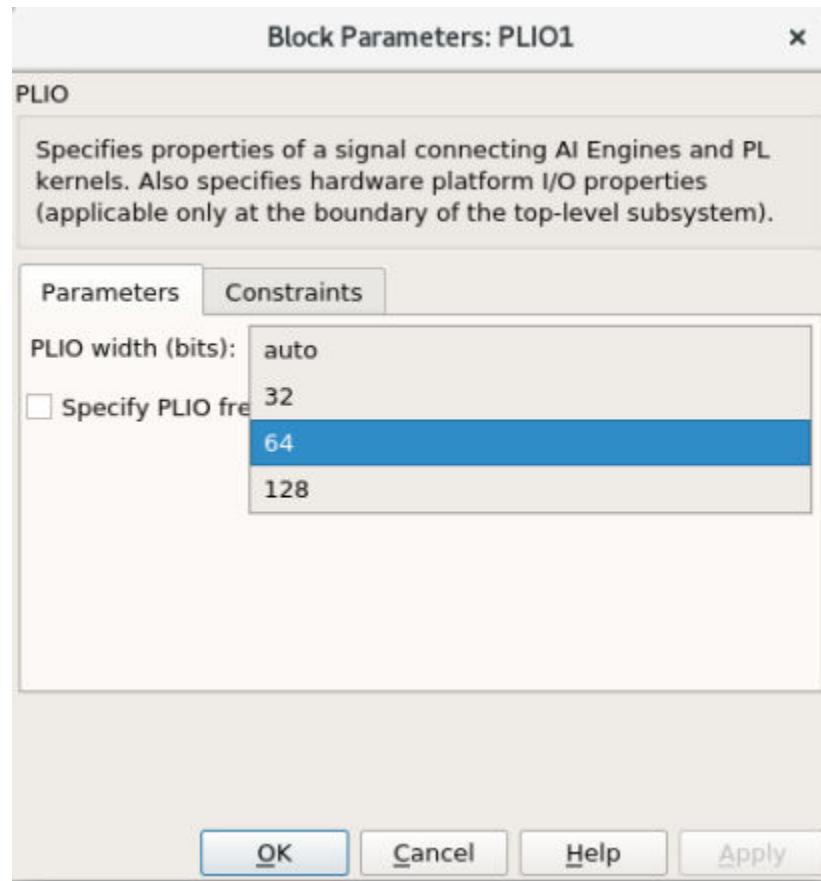
Consider the following example where an AIE Signal Spec block is connected at the boundary of the AI Engine subsystem.

Figure 239: AIE PLIO Block Connected to AI Engine Subsystem



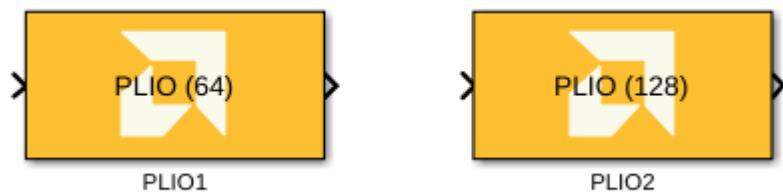
By default, the PLIO width is set to `auto`. Other available options are: 32, 64, and 128.

Figure 240: Block Parameters: AIE PLIO Width



After the PLIO width is specified as 64 and 128 for the two AIE PLIO blocks in the example, the GUI updates as follows.

Figure 241: AIE Signal Spec Blocks Updated



The PLIO width specified using the AIE PLIO block is automatically updated in the generated graph code (`graph.cpp`) with PLIO constraints as shown in the following code.

**Note:** Adding the PLIO width does not impact the Simulink simulation, only the code generation.

### Snippet of graph.h

```
adf::input_plio PL_AIE_IN;
adf::output_plio AIE_PL_OUT;

Subsystem() {
    PL_AIE_IN = adf::input_plio::create("PL_AIE_IN",
                                         adf::plio_64_bits,
                                         "./data/input/PL_AIE_IN.txt");

    AIE_PL_OUT = adf::output_plio::create("AIE_PL_OUT",
                                         adf::plio_128_bits,
                                         "AIE_PL_OUT.txt");

    adf::connect<->(PL_AIE_IN.out[0], mygraph.PL_AIE_IN);
    adf::connect<->(mygraph.AIE_PL_OUT, AIE_PL_OUT.in[0]);
}
```

The PLIO attributes are used in a program to read input from a file or write output data to a file. You can see a connection with one 64-bit PLIO attribute declared for input and one 128-bit PLIO attribute declared for output.

### Data File Layout

When simulating PLIO with data files, the data should be organized to accommodate both the width of the PL block as well as the data type of the connecting port on the AI Engine block. Vitis Model Composer automatically generates the data file that accommodates to the specified PLIO width.

For example, a data file representing a 64-bit PL interface to an AI Engine kernel expecting `cint16` should be organized as four columns per row, where each column represents a 16-bit real or imaginary value.

*Table 34: Data File Layout*

PLIO Width	AIE Kernel Data Type	Data File Layout
64-bit	<code>cint16</code>	0 0 0 0 1 1 1 1 2 2 2 2

This data file is in the output code directory.

### Specifying PLIO Frequency

The AI Engine can run at up to 1.25 GHz and can write (at most) two streams with a 32-bit data width per cycle. In contrast, an IP implemented in the PL can run at up to 500 MHz, while consuming a larger bit-width. In order to balance the throughput between the AI Engine and PL, and also ensure the processes do not create a bottleneck with respect to the total performance, it is required to match the rates between the two. Vitis Model Composer supports specifying the frequency of PL from Platform I/O tab in AIE Signal Spec block.

You can either adjust the PLIO frequency or the Width to match the rate between the AI Engine and PL. Consider an example of a 32-bit channel written to each cycle by the AI Engine at 1 GHz. In order for PL to match the rate of AI Engine, it has to consume twice the data at half the frequency or four times the data at a quarter of the frequency.

*Table 35: PLIO Frequency*

AI Engine		PL	
Frequency	Data per Cycle	Frequency	Data per Cycle
1 GHz	32 bit	500 MHz	64 bit
		256 MHz	128 bit

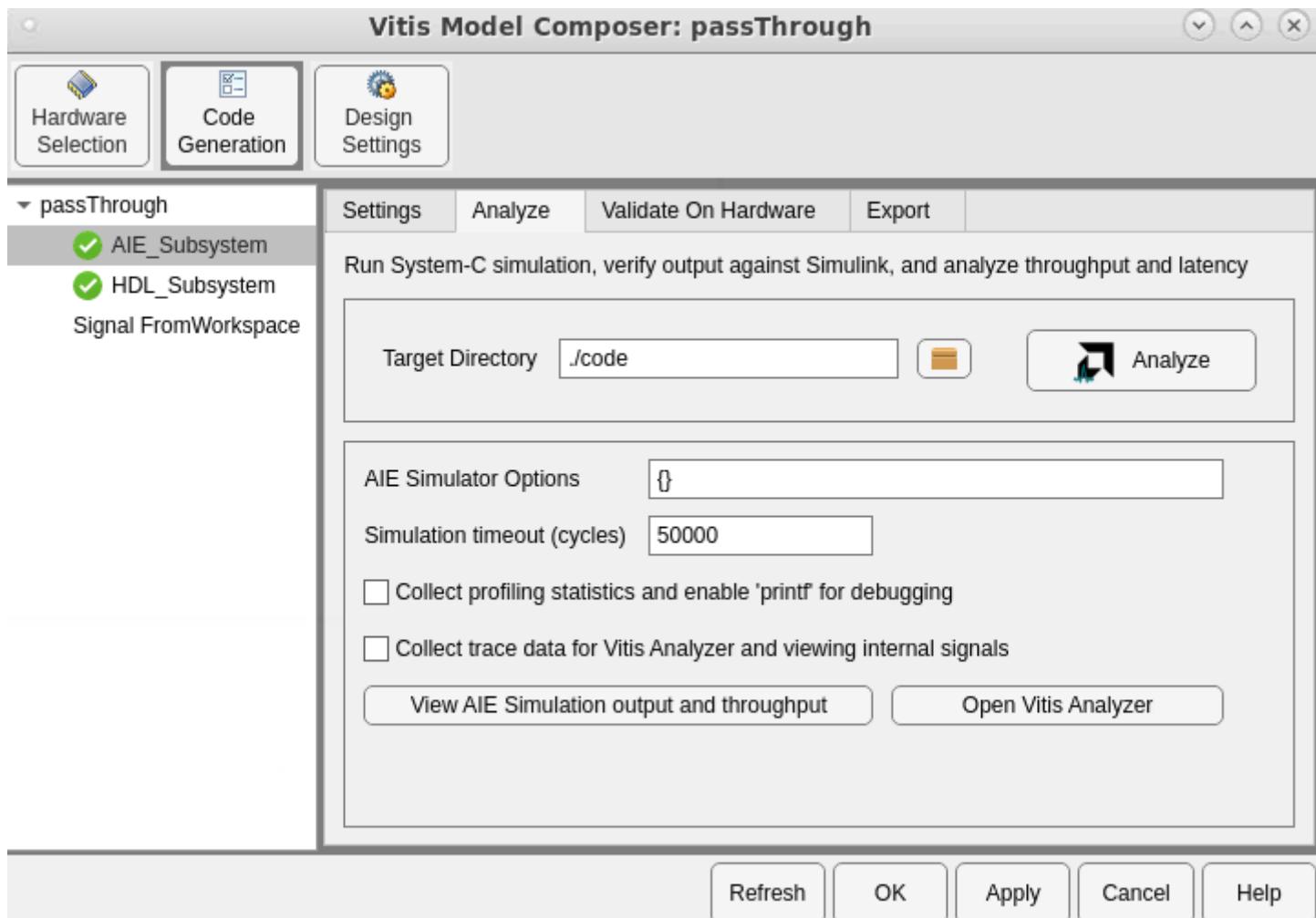
## Verification of AI Engine Code

The AMD Versal™ Adaptive SoC AIE simulator (`aiesimulator`) models the timing and resources of the AI Engine array accurately while using transaction-level, approximately timed SystemC models for NoC, DDR, PL, and PS. This allows accurate performance analysis of your AI Engine. Vitis Model Composer supports verification of the dataflow graph using this AIE simulator.

### ***Vitis Model Composer Hub Block for Verification***

To verify and analyze the AI Engine design, use the Analyze tab of the Model Composer Hub block.

Figure 242: Vitis Model Composer Hub: Testbench and Simulator Options



The **Analyze** button will generate and compile AI Engine code from the design and run the cycle-approximate AIE Simulation (SystemC).

The Simulation timeout value limits the execution to the specified number of cycles. This is necessary because of the finite amount of input data - if the timeout value is not specified, the AI Engine kernels are invoked repeatedly forever (that is, the graph runs infinitely). To avoid this situation, specify the Simulation timeout value as shown in the preceding figure.

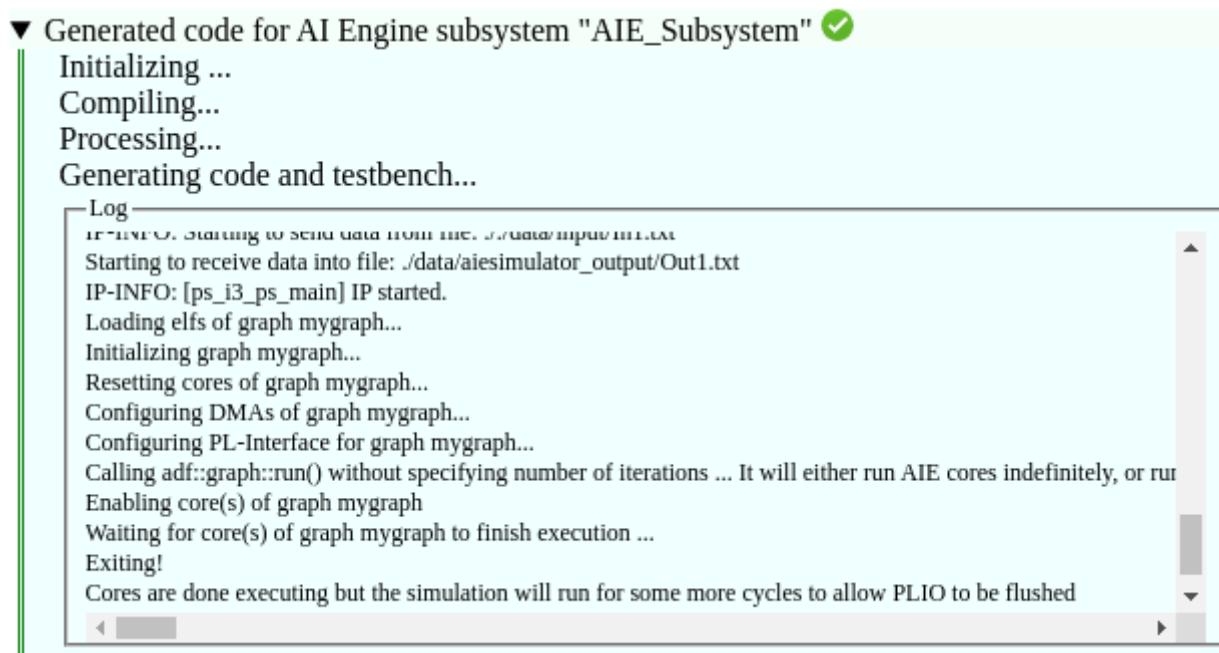
The default timeout value is set to 50,000 cycles and this value is used to terminate the simulation after the specified number of clock cycles.

AI Engine code verification and analysis advances in three phases:

1. Compiling the AI Engine graph design.
2. Running simulation using the AI Engine simulator.
3. Verifying the simulation results by comparing the output with the golden reference output.

After clicking **Analyze**, you can monitor the compilation, simulation, and verification progress of the AI Engine graph code from the Progress window (see the following figure).

Figure 243: Graph Code Progress



After successful compilation and simulation, Model Composer automatically compares the target output with the golden reference output and returns the following message in the Progress window (or in the corresponding simulation log files).

```
Comparing simulation results ...
Output data file : data/aiesimulator_output/Out1.txt.mod
reference data file : data/reference_output/Out1.txt
Simulation results MATCH.
*****
Test PASSED
Verification Complete
```

**Note:** In some scenarios, the simulator output produces fewer samples when compared with the golden output or vice-versa. In such cases, the test result will be still show as 'PASS.' This indicates that the first 'n' lines from the simulation output matches the first 'n' lines of the golden output and the results are partially matched. The `.diff` file in the corresponding simulator output captures any difference with the reference output.

In addition to this, the AI Engine compiler writes various configuration and binary files to the `work/aie` directory. For more information on the structure and contents of the directory specific to the compilation, refer to the *AI Engine Tools and Flows User Guide* ([UG1076](#)).

## Profiling Statistics and Event Tracing

You can obtain profiling data when you run your AIE simulation. Analyzing this data helps you gauge the efficiency of the kernels, the stall and active times associated with each AI Engine, and pinpoint AI Engine kernels whose performance might not be optimal. This also allows you to collect data on design latency, throughput, and bandwidth. In addition to this, you can do event trace using a formatted `printf` statement in the code for printing debug messages. To achieve this, you should enable the option Collect profiling statistics and enable 'printf' for debugging in the Analyze tab of the Vitis Model Composer Hub block.



**IMPORTANT!** Using this option generates a `run_summary` file which is written to the `aiesimulator_output` folder.



**IMPORTANT!** Enabling this option might slightly increase the overall AIE simulation time.

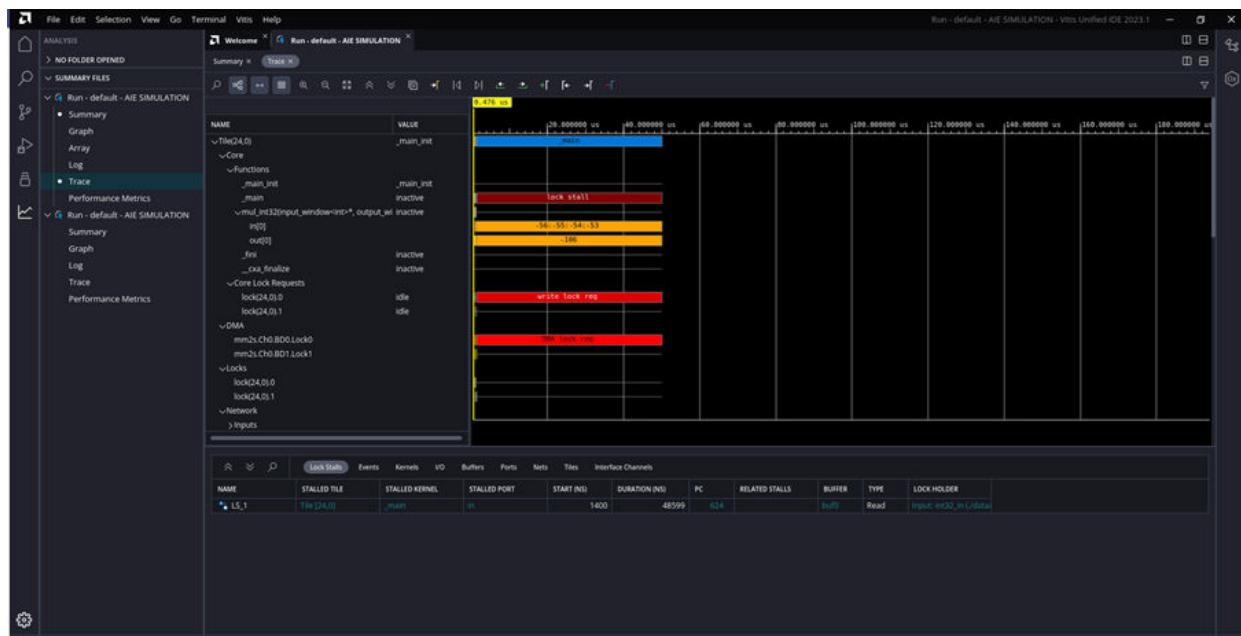
## Viewing Results in the Vitis Analyzer

The AMD Vitis™ software platform analyzer is a utility that allows you to view and analyze the reports generated while building and running the application. It is intended to let you review reports generated by both the Vitis compiler when the application is built, and the Xilinx Runtime (XRT) library when the application is run. The Vitis analyzer can be used to view reports from Vitis integrated design environment (IDE).

During the simulation of the AI Engine graph, the `aiesimulator` writes a summary of the simulation results called `default.aierun_summary`. This can be viewed in the Vitis analyzer. The summary contains a collection of reports and diagrams reflecting the state of the AI Engine application.

Vitis Model Composer integrates the Vitis analyzer utility. This can be invoked by enabling the Collect trace data for Vitis Analyzer option from within the Hub block, along with the AIE Simulation. When the simulation completes, Vitis Model Composer automatically reads the `default.aierun_summary` file which is generated during the `aiesimulator` run and invokes the Vitis Analyzer Summary window. You can navigate to the Trace window to see the trace data as shown.

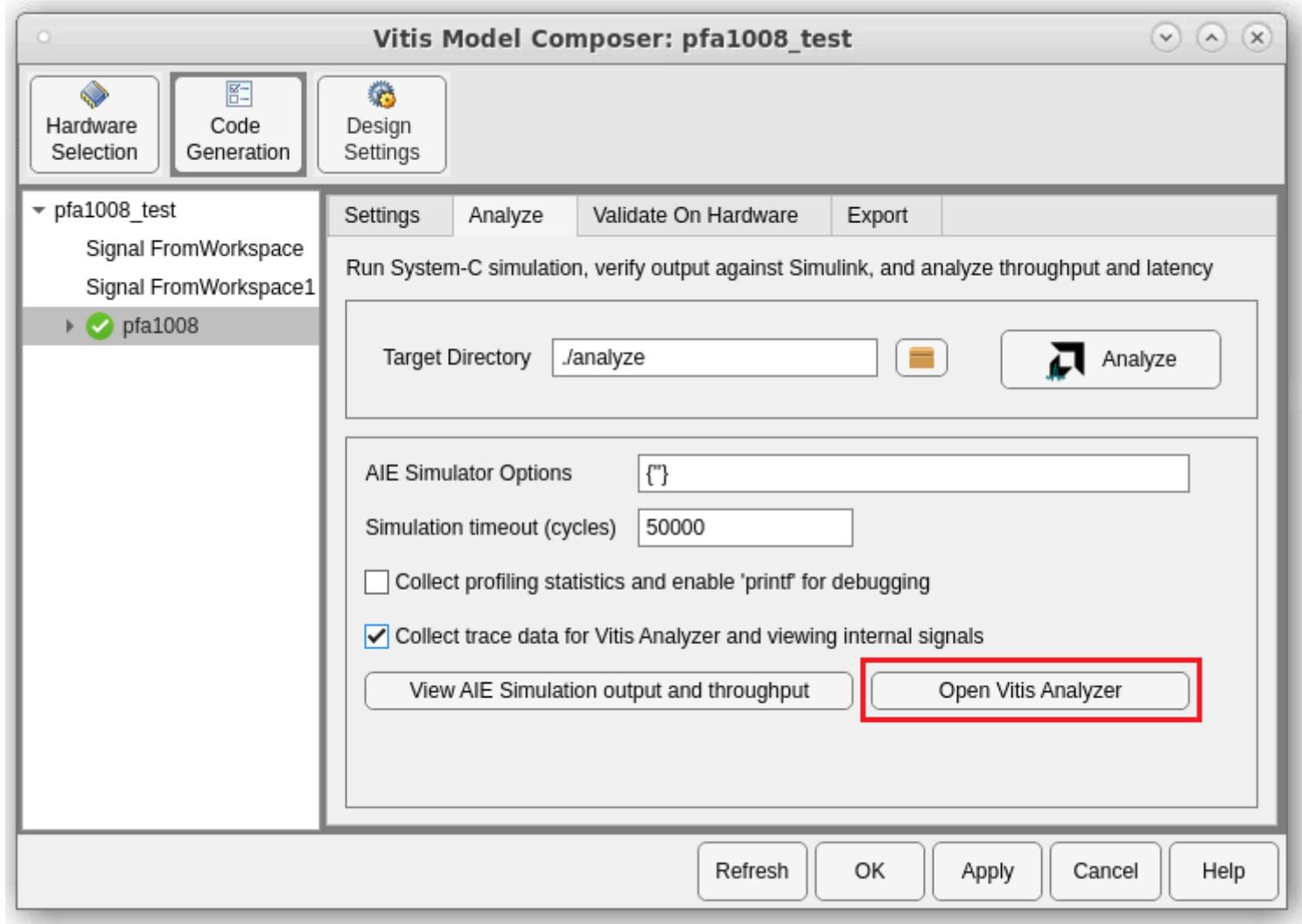
Figure 244: Vitis Analyzer Trace



From the report navigator you can view other available reports, such as Summary, Profile, Graph, Array, Log, and Performance Metrics.

**Note:** You can relaunch the Vitis Analyzer by clicking **Open Vitis Analyzer** as shown in the following figure. You can view trace data in Vitis Analyzer using this option only when the AIE Simulation has been run at least once after enabling the **Collect trace data for Vitis Analyzer** option.

Figure 245: Collect trace data for Vitis Analyzer



In Vitis Model Composer, you can launch Vitis analyzer from the MATLAB command window using the following command.

```
xmcOpenVitisAnalyzer('file')
```

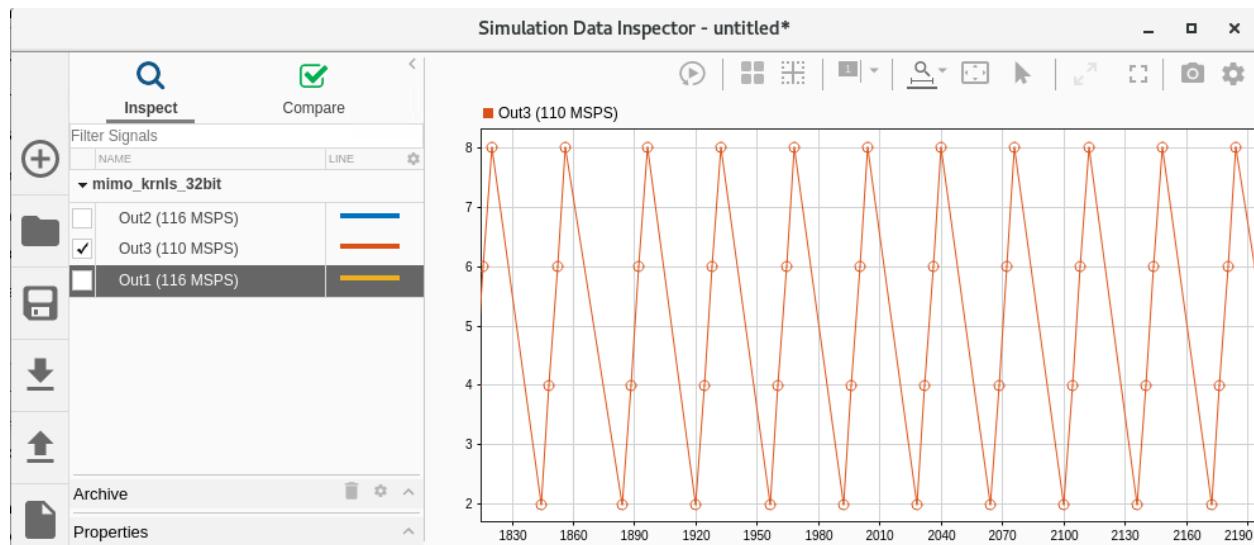
When this command is given without a <file> option, Model Composer looks up all default.aierun\_summary files in the current target directory and invokes the Vitis analyzer to display the results in the summary page.

For more information on the Vitis analyzer, refer to the Using Vitis Analyzer topic in [Vitis Unified Software Platform Documentation](#).

## Plotting AIE Simulation Output Data and Calculating Throughput

Vitis Model Composer provides the capability to log the simulation data and visualize the output of an AI Engine subsystem by integrating the Simulink 'Simulation Data Inspector' feature. It also calculates the throughput for each output port of the AI Engine subsystem. To achieve this, select Plot AIE Simulation output and estimate the throughput from the Hub block. When the AIE simulation completes, the tool automatically brings up the Simulation Data Inspector window reflecting the outputs of the AI Engine subsystem as shown in the following figure.

Figure 246: Simulation Data Inspector



The Simulation Data Inspector displays available data in the Inspect pane. To plot a signal, select the check box next to the signal. You can modify the layout and add different visualizations to analyze the simulation data. You can also get the throughput information for each port from the Inspect pane. You can use cursors to select a region of the plot for the throughput calculation. If required, you can re-open the Simulation Data Inspector from the Simulink® Editor toolbar using the Simulation Data Inspector button.

**Note:** To see the signals in the Simulation Data Inspector, the Visualization Type must be set to Time Plot.

For more information on using the Simulink® Simulation Data Inspector, visit <https://www.mathworks.com/help/simulink/ug/create-plots-with-the-simulation-data-inspector.html>.

## Calculating Latency Between AIE Ports

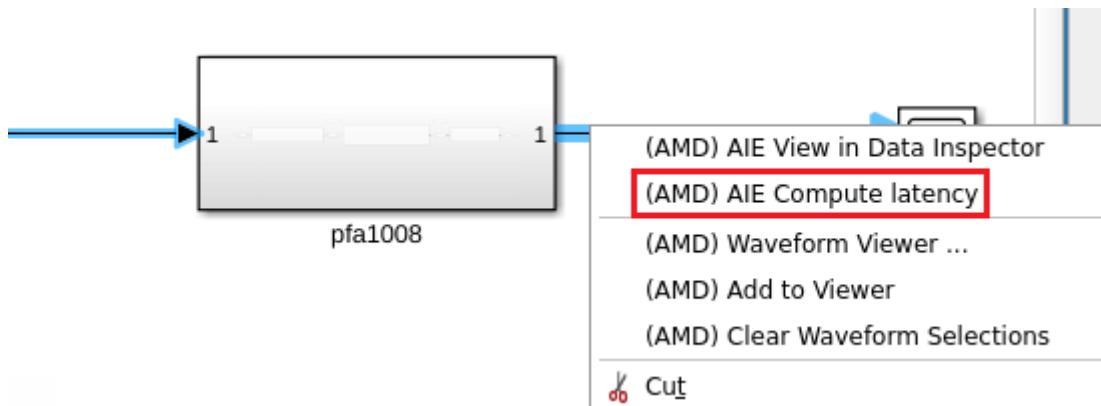
The AIE simulation is a cycle-approximate simulation, which means it can be used to estimate the latency between input and output ports of the AI Engine subsystem. This latency calculation can be displayed in Vitis Model Composer. To achieve this, run **AIE simulation** from the Analyze tab of the Model Composer Hub block with the **Collect trace data for Vitis Analyzer** option enabled. Then, do the following to select AI Engine subsystem ports and calculate the latency:

1. On the Simulink canvas, select one input to the AI Engine subsystem and one output from the AI Engine subsystem.

**Note:** To select multiple signals on the Simulink canvas, hold down the **Shift** button while selecting the signals.

2. Right-click on the selected signals and select **(AMD) AIE Compute Latency**.

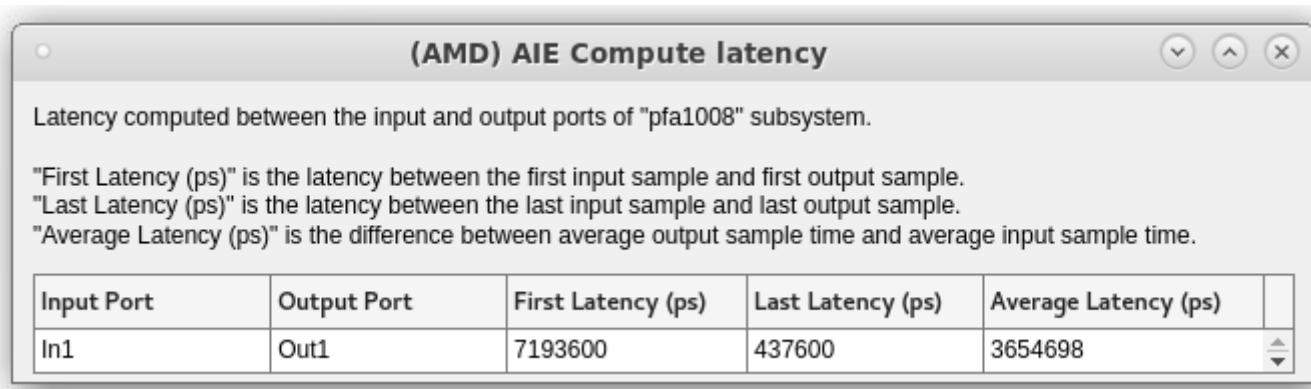
Figure 247: AIE Compute Latency



In the image above, `pfa1008` is an AI Engine subsystem.

If AIE simulation has been previously run on this design, Vitis Model Composer will display the first sample latency, last sample latency, and average latency between the two selected ports.

Figure 248: AIE Latency Display

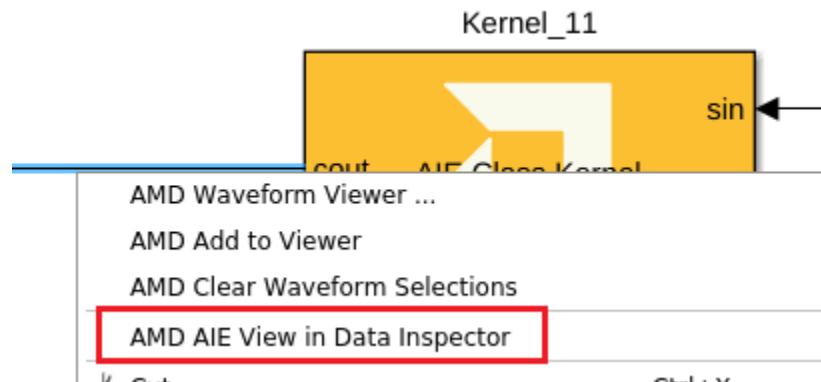


Latency can fluctuate over the course of a simulation, but it generally converges to a stable value over time.

## Plotting AIE Simulation Internal Signals

The Simulation Data Inspector can also be used to plot and calculate throughput for signals within an AI Engine subsystem. To achieve this, first run AIE Simulation with "Collect Trace Data for Vitis Analyzer and viewing internal signals" selected. Then, the signals must be selected by right-clicking and selecting **AMD AIE View in Data Inspector**.

Figure 249: Add AI Engine Internal Signals to Data Inspector



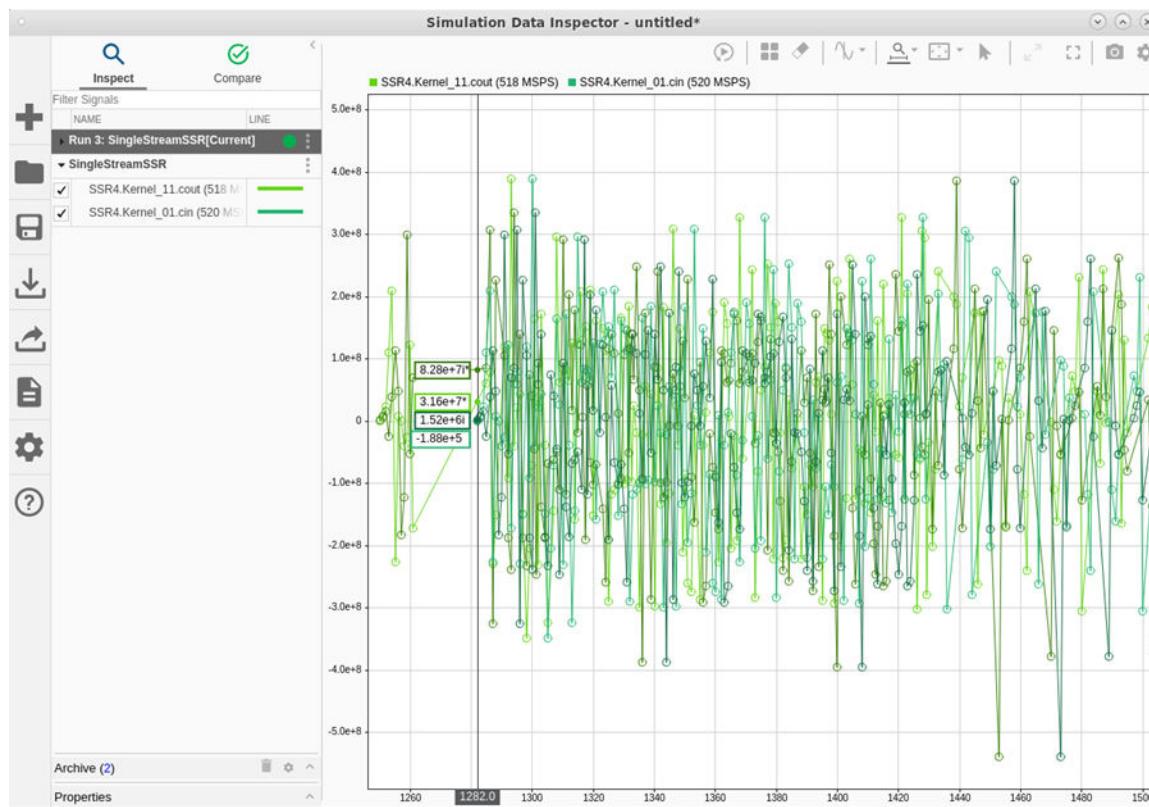
You have the option to plot the signal at its source port, destination port, or both. These signals might differ in hardware due to latency (for example, buffers) introduced in the signal.

Figure 250: Plot Source and/or Destination Ports



After making a selection, the selected signals then appear in the Simulation Data Inspector. In the following figure, "View source and destination" was selected. Note the latency between the signals at the source port and destination port.

Figure 251: Source and Destination Signals



**IMPORTANT!** Selecting the input or output of FIFO or PLIO blocks to view in the Simulation Data Inspector is not supported.

**IMPORTANT!** Viewing bfloat16 signals in the Simulation Data Inspector is not supported.

## Running Simulation using the Makefile

If you want to edit the graph code, for example, to add or modify constraints, or explore the graph code by editing the kernel configuration parameters (like window size or window margin etc.), and re-compile. Vitis Model Composer generates a Makefile allowing you to easily accomplish this by running the following command.

```
make all
```

This command compiles an AI Engine graph using the default `aiecompiler` target and runs the simulation using `aiesimulator`. It also compares the output of the simulator with the golden output. This command also launches the Vitis analyzer, based on the option selected in the Hub block.

To do this, you need to source the `settings64.csh` or `settings64.sh` from `<install_dir>/<version>/Model_Composer`.

# Hardware Validation Flow for AI Engines and PL

## Introduction

The hardware validation flow for AI Engines and PL in AMD Vitis™ Model Composer provides a methodology to verify AI Engine and PL-based applications on AMD hardware (AMD Versal™ devices). Vitis Model Composer provides the option to generate a hardware image targeting a specific platform for the Simulink® model. This hardware image can then be run on a board to verify whether the results from hardware match with the simulation output. This image can be either baremetal or Linux-based. This section covers the details of the Hardware Validation flow.

## High-Level Flow for Generating a Hardware Image

You can generate a hardware image for designs with only AI Engine subsystems, designs with both PL and AI Engine subsystems, or designs with only PL subsystems.



**IMPORTANT!** Vitis Model Composer supports integration of blocks from the AI Engine library with the HDL library to generate a hardware image. However, to integrate with HLS C/C++ code, you should use the HLS Kernel block, and not other blocks from the HLS library.

To generate the hardware image from a Simulink® design, you should have an expandable platform. This is a file with the `.xp fm` extension. For AMD hardware, such as VCK190, the platform files are shipped with Vitis. You can also create an expandable custom platform for a custom board and use it in Vitis Model Composer. For more information refer to the *Vitis Unified Software Platform Documentation: Application Acceleration Development* ([UG1393](#)).

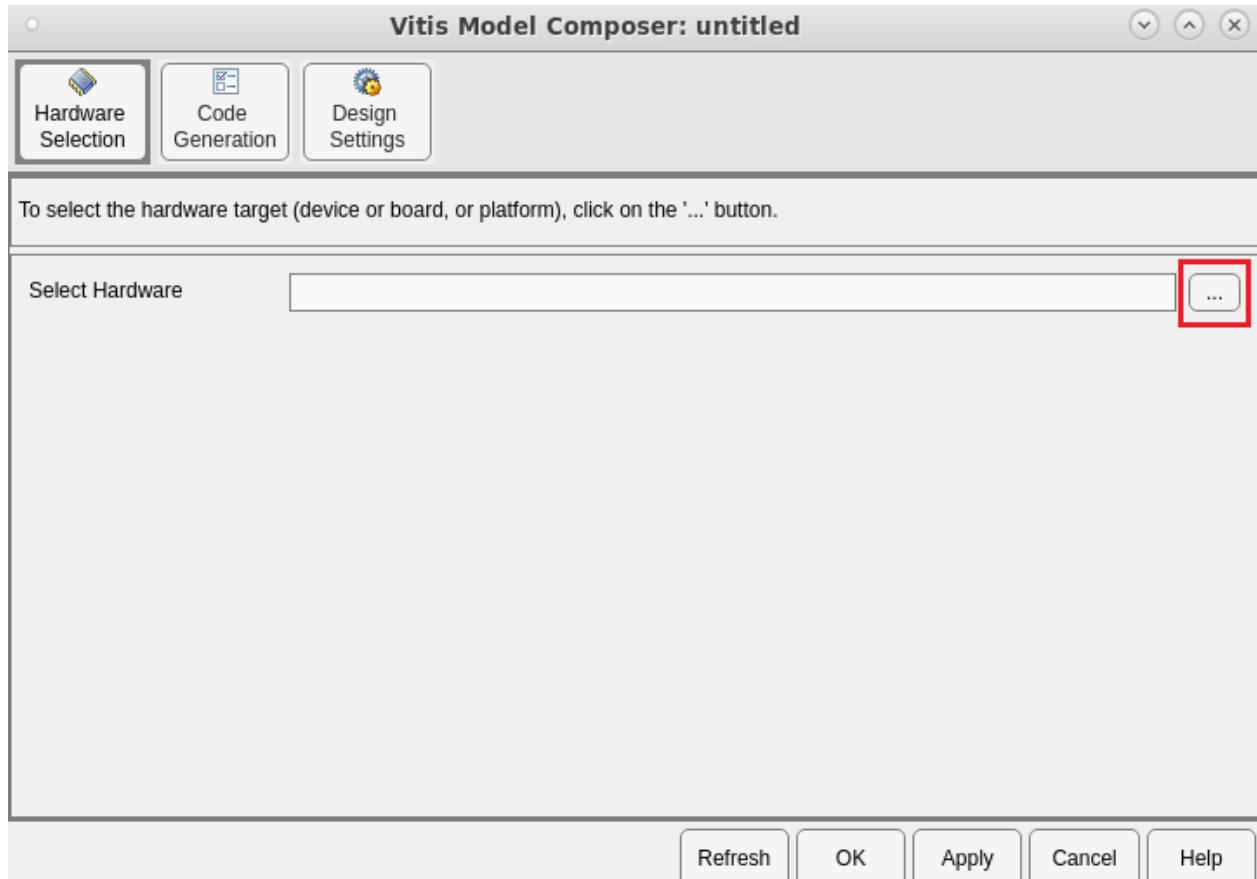
Using the Vitis Model Composer Hub block, you have a choice between creating a baremetal application or Linux-based application for either a hardware target or a hardware emulation target. In hardware emulation, AMD Vivado™ simulator is used to simulate the PL portions of the design and QEMU is used to emulate the host code for the Arm® processor. The advantage of hardware emulation is that there is no need for a hardware board. However, hardware emulation is only an emulation of the hardware and is, in general, slower than running the design in hardware.

As part of the creation of the hardware image, the input data samples that are fed into the design during simulation in Simulink are collected. Likewise, the output data samples from the design are also collected. The input and output data are packaged as part of the hardware image and downloaded to the hardware. The input data is fed into the design in hardware, the output data is collected and compared with the output data from simulation. This comparison is done by the processor in the device. Unless something is wrong, the result should match bit by bit with the result of the simulation because the simulation in Vitis Model Composer is bit-accurate. The hardware runs independently of Vitis Model Composer, and during the hardware run no information is communicated between Vitis Model Composer and the hardware.

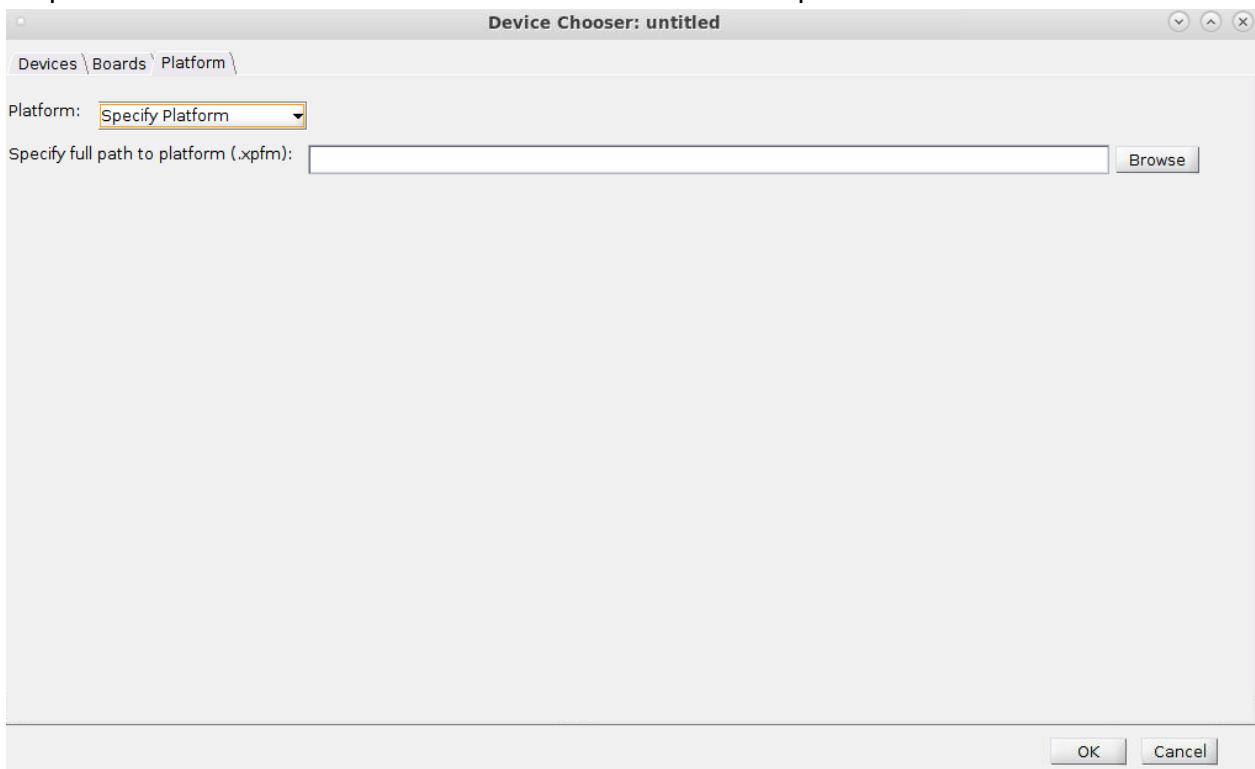
**Note:** For baremetal applications, the tool generates a `BOOT.BIN` image file. For Linux-based applications the tool generates an `sd_card.img` image file. In both cases, for hardware emulation, Vitis Model Composer triggers the emulation when image creation is complete. If you choose the hardware target, you can find the details of how to move the images to the VCK190 hardware board [here](#).

# Setting up the Tool to Generate an Image File for Hardware Validation Flow

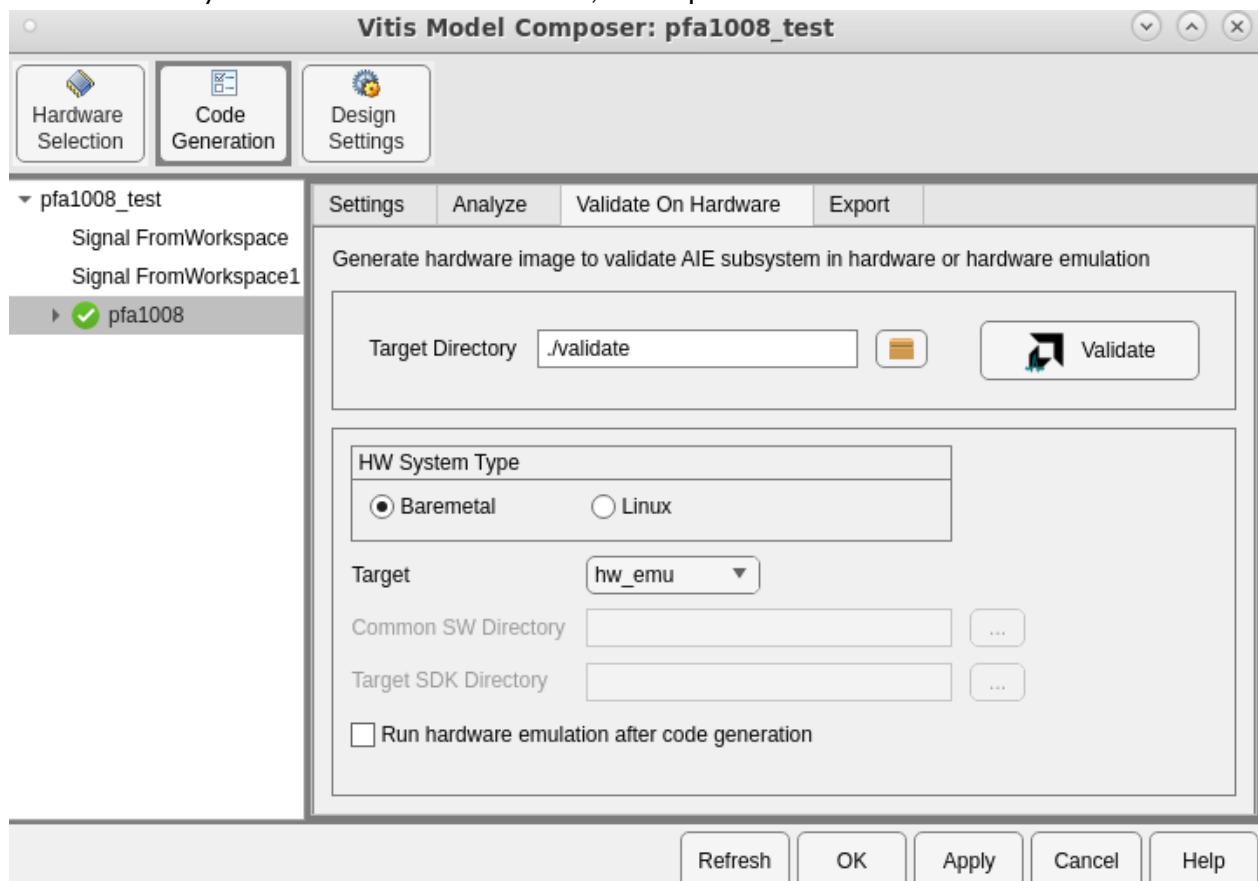
1. Choose a platform. In the Vitis Model Composer Hub block, select the **Hardware Selection** tab, and then click on the button to the right of the Select Hardware field, as shown in the following figure.



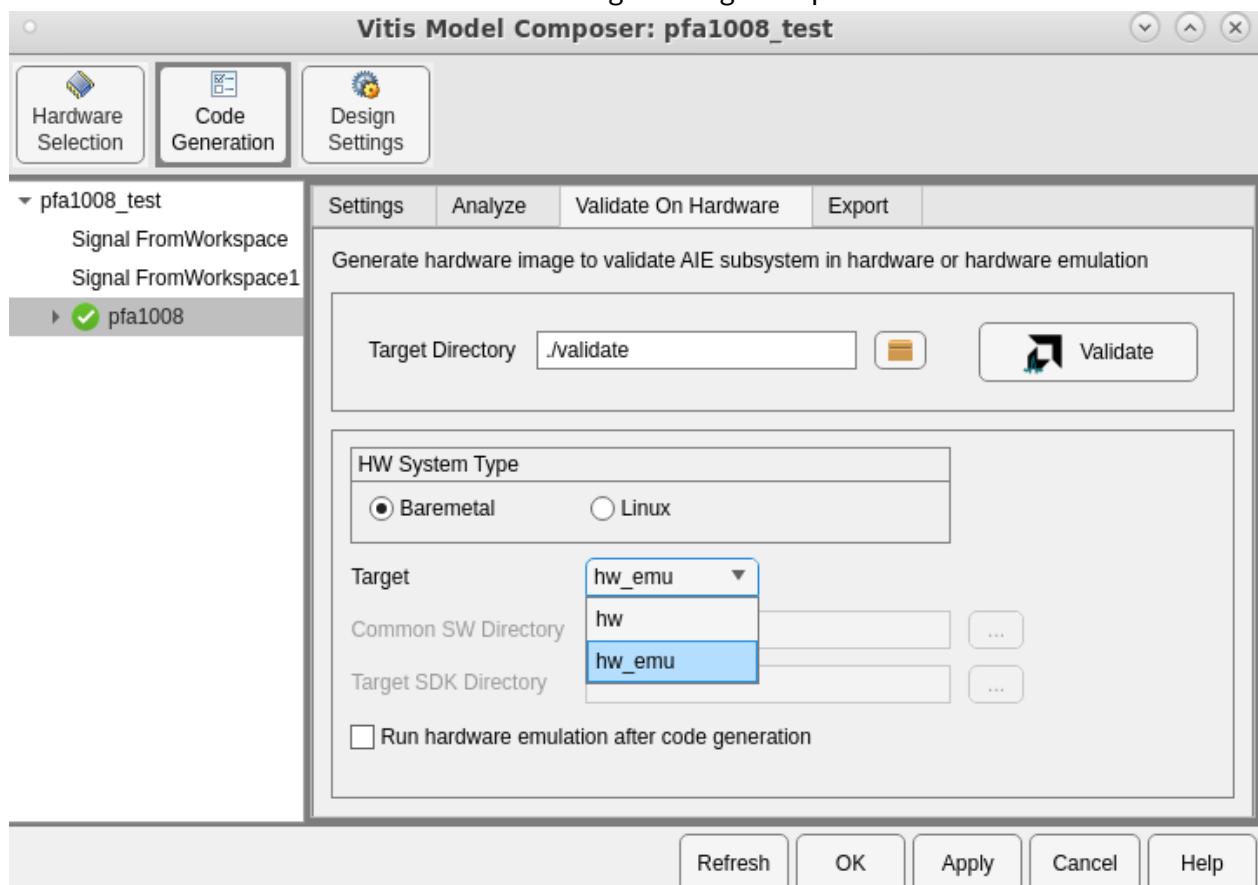
2. Select the **Platform** tab in the Device Chooser. Select **Specify Platform** in the Platform drop-down menu and use the **Browse** button to select a base platform.



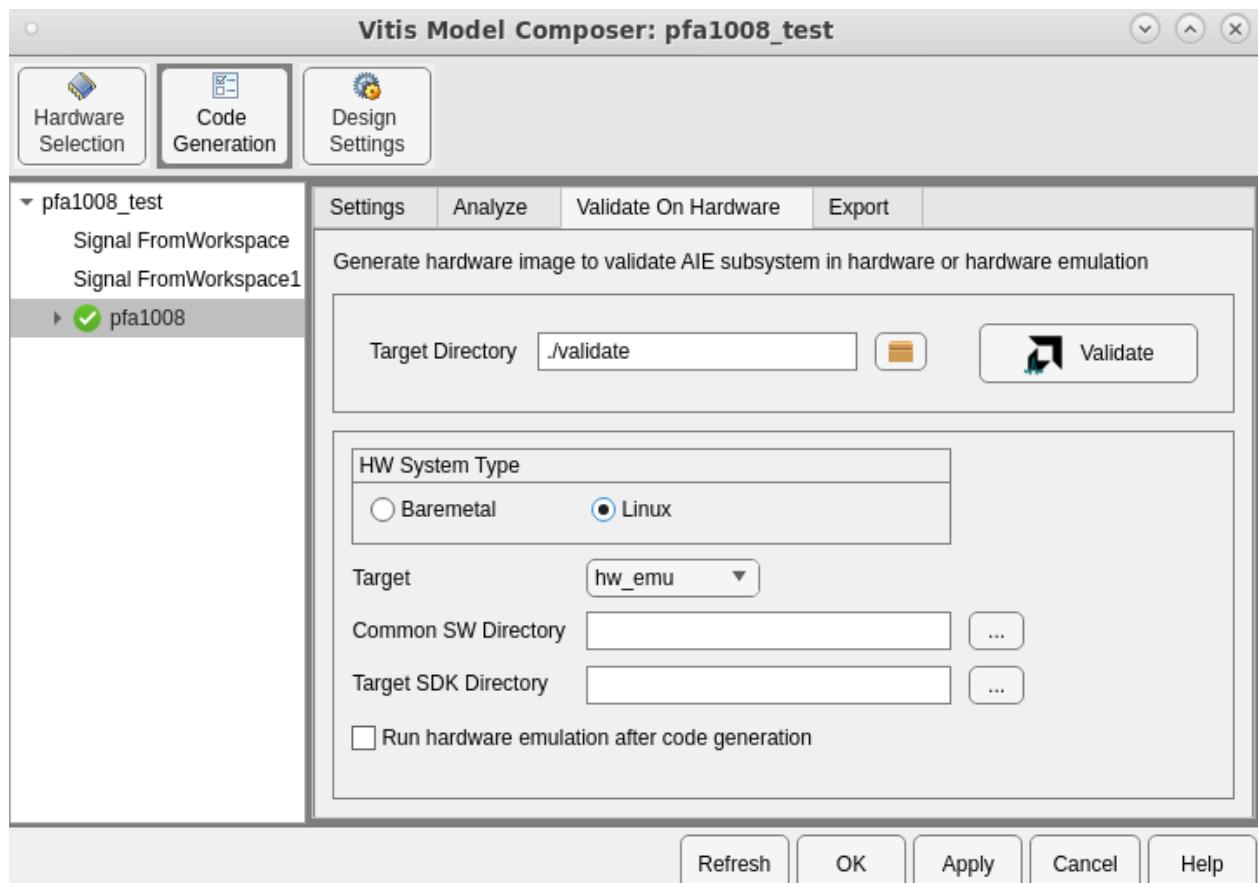
3. Select the subsystem to validate on hardware, then open the **Validate on Hardware** tab.



- From here, choose between baremetal and Linux. For each selection, you can choose between hardware or hardware emulation using the Target drop-down menu.



**Note:** For Linux applications only, further information is required (see the following figure). This is a once-only action. Use the following steps to obtain the necessary information.



1. Click [here](#) to download the Versal common image. Unzip the download and specify the directory in the Common SW Dir field.
2. Switch to bash shell and source `sdk.sh` in the common image directory. This will prompt for a target directory path for the SDK. Extracting the SDK will take about ten minutes. Specify the SDK directory in the Target SDK Dir field.
5. Click **Validate**. Depending on your settings and the complexity of your design, hardware image generation may take up to one hour. Subsequent generation can be much faster if changes to the design do not cause a change in the PL. For example if you simply increase the simulation time in Simulink (to collect more data), change the data source, or make modifications to the AI Engine kernel, the subsequent image generations will be faster.

---

# Running the Hardware Flow Outside the MATLAB Environment

When you perform any tasks in the Vitis Model Composer Hub block that involve code generation (including Analyze, Validate on Hardware, and Export), Model Composer generates Makefiles that allow you to re-run various design steps outside of Vitis Model Composer. This can allow you to run design iterations more quickly. It can also free up your MATLAB session for other work.

1. Open a Linux terminal.
  2. `cd` to the `<code-generation-directory>` directory specified in the Vitis Model Composer Hub block.
  3. From the Linux terminal, run `source setup.sh`. This opens a bash shell configured with the environment variables needed to run the tools involved in the hardware flow.
  4. To re-run the entire hardware flow, run `make all` from the bash shell.
  5. You can also re-run individual steps (targets) in the build process. Each directory in the code generation directory contains its own Makefile. To see which targets are available in a Makefile, run `make help`. Note that some targets depend on other targets; this information is contained within each Makefile.
  6. To leave the bash shell and return to the parent shell, run `exit`.
- 

## Design Considerations

- HDL and HLS subsystems can only have AXI4-Stream input and output ports. The AXI4-Stream input and output ports of the subsystem must have a bit width that is a multiple of 8 bits, up to a maximum of 128 bits.
- Ensure that there are no extra outputs from the subsystem that will not be in the hardware implementation, such as debug outputs to monitor internal signals.
- If multiple blocks are being driven by the same input signal, the signal multiplexing must occur outside the hardware subsystem, so there are subsystem inputs for each block input.
- Considerations for HLS Designs:
  - You can only use HLS Kernel blocks to import HLS C/C++ code (for PL). The blocks from the HLS library are not supported by the Hardware Validation Flow
  - The HLS Kernel should be in free-running mode. This is accomplished by including the following pragma in the HLS function: `#pragma HLS INTERFACE ap_ctrl_none port=return`

- Ensure the bit width of the HLS Kernel input or output that connects with the AI Engine matches the PLIO width of the AIE.
- Consideration for HDL Designs:
  - The HDL part of the design must not be purely combinational.

# Connecting AI Engine and Non-AI Engine Blocks

---

## AI Engine/Programmable Logic Integration

An AI Engine kernel written using specialized intrinsic and imported into Vitis Model Composer can be used as part of a larger AMD Versal™ Adaptive SoC system design. In addition to kernels operating on the AI Engines, you can specify kernels to run on the programmable logic (PL) region of the device. The PL kernels can be written using RTL or HLS C/C++ functions. The connection between AI Engine and the PL block is routed through a physical channel interface tile and conceptually the data width of the connections are 32 bits, 64 bits or 128 bits.

Model Composer allows connecting an AI Engine kernel to a HLS PL kernel only if the data types and complexities of these port matches. If the datatypes or complexities of the port of the AI Engine kernel and the port of the PL kernel do not match, an interface blocks should be used to reconcile the discrepancy.

This chapter discusses interconnecting HDL blocks or HLS C/C++ functions with AI Engine kernels:

- [Interconnecting AI Engine and HDL Blocks](#)
- [Interconnecting AI Engines and HLS Kernels](#)

## Interconnecting AI Engine and HDL Blocks

The HDL blockset in the AMD toolbox contains the common DSP building blocks such as adders, multipliers, and registers. It also includes a set of complex DSP building blocks such as FFTs, filters, and memories. Model Composer automatically compiles designs into low-level representations (i.e., RTL) which can be targeted to programmable logic. The Versal hardware allows for connecting AI Engine kernels and PL kernels. However, HDL and AI Engine domains are incompatible in at least two aspects:

- The HDL domain is cycle accurate, whereas the AI Engine domain is only bit accurate. A Model Composer HDL design might not be ready to accept input data or might not have a valid output and these are managed through `tvalid` and `tready` signals of AXI4-Streamports.

- The HDL domain accepts only scalar inputs, whereas AI Engine blocks work with variable sized vector signals.

In order to properly manage the sampling times across two domains and simulate the heterogeneous system with both PL (modeled with HDL blocks) and AI Engine, Vitis Model Composer provides interface blocks in the Utilities library to connect from AI Engine to HDL blocks and vice-versa.

- AIE to HDL - This block connects AI Engine to HDL blocks using an AXI4-Stream-like interface.
- HDL to AIE - This block connects HDL to AI Engine blocks using an AXI4-Stream-like interface.

You can find these blocks in `AMD Toolbox/Utilities/Connectors` library.

**Figure 252: Connector Blocks**



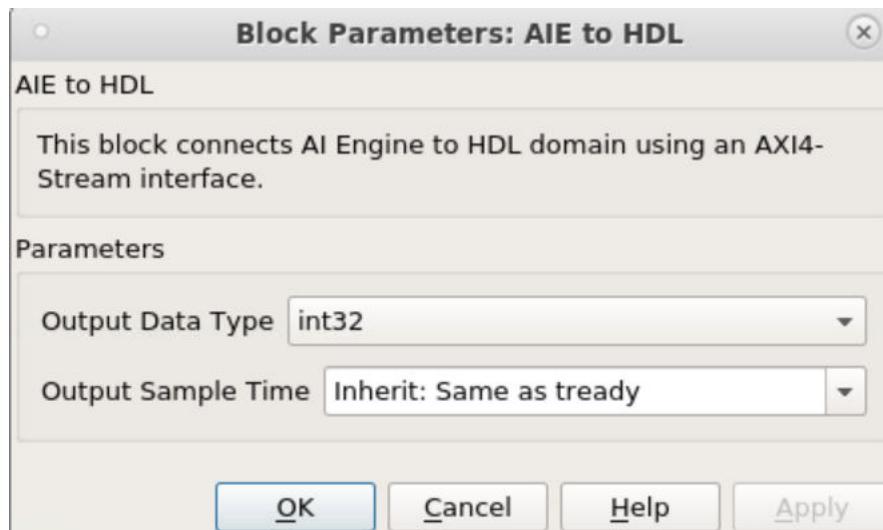
As discussed, AIE to HDL and HDL to AIE blocks have `tvalid` and `tready` ports as depicted in the previous figure. The gateway from the AI Engine to the HDL domain can accept a vector input but generates a scalar output, meaning that in Simulink, the HDL domain will run at a different rate than the AI Engine domain. For example, if the AI Engine domain is producing 16 samples at every clock, the HDL domain must run at least 16 times faster to process the data. However, in some cases, the HDL domain needs to run even faster because the HDL domain, being a cycle accurate domain, might not consume one sample per clock. For example, if the HDL domain consumes one sample every two clocks (known as an initiation interval of 2), for the earlier example, the HDL domain needs to run 32 times faster than the AI Engine domain.

The following sections discuss setting the block parameters of the AIE to HDL and HDL to AIE blocks and includes examples.

## AIE to HDL

The AIE to HDL block connects the output of AI Engine block with the input of HDL block. This block accepts variable size signal from AI Engine blocks along with the `tready` signal which indicates whether the HDL domain can accept the data. The input data type to this block is inherited from the input signal.

**Note:** If the HDL domain `tready` signal stays low for a long time, eventually the internal buffers in the AIE to HDL block will overflow and the simulation stops. The bit width of the `tdata` output of the AIE to HDL block is limited to 32, 64, and 128 according hardware functionality.

*Figure 253: AIE to HDL**Figure 254: AIE to HDL Parameters*

### Setting the AIE to HDL Block

The following image depicts the components that are needed to connect an AI Engine subsystem to an HDL design. In setting this connection, you should consider certain input design criteria and set the parameters of the blocks accordingly. These input design criteria are:

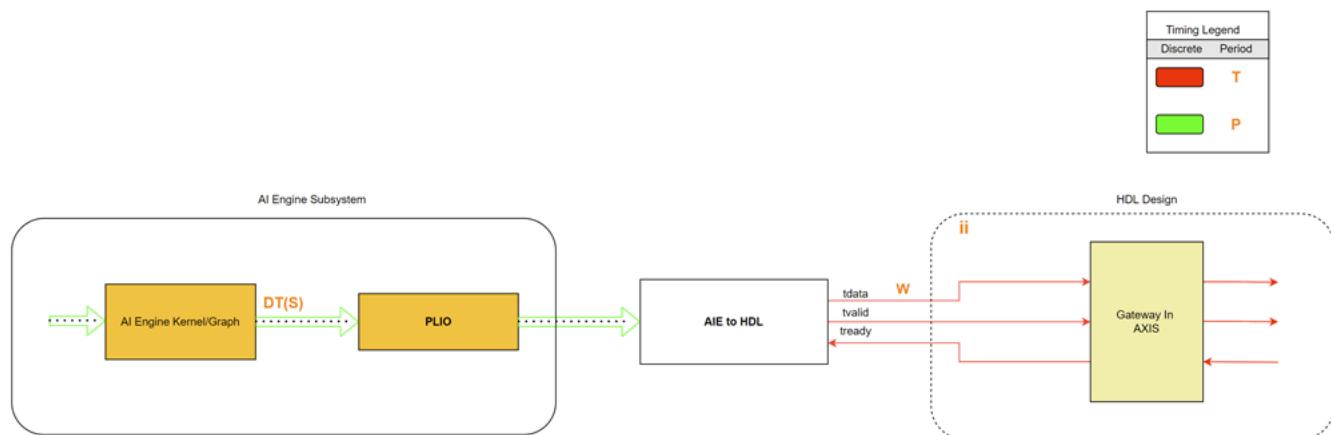
1. In the HDL design, the bit width of the `tdata` signal line ( $w$ ). This is the bit width of the data in the programmable logic.
2. HDL design sample time ( $T$ ). This sample time determines the target clock rate for which the HDL design will be clocked in hardware. For single clock designs, this will be the sample time set in the Gateway In AXIS block.

3. As mentioned previously, simulation in HDL domain is cycle-accurate. An HDL design might not be ready to accept a new sample at every cycle (the `tready` signal from the HDL design will be set to zero when the HDL design cannot accept new samples). This is referred as the initialization interval ( $II$ ) of the HDL design. For example, if an HDL design accepts a new sample every 10 cycles, the design would have an  $II$  of 10. A design that can accept a new sample at every clock cycle has an initiation interval of one.
4. The number of samples in the output of the AI Engine kernel ( $S$ ).
5. The output data type of the AI Engine kernel ( $DT$ ).

Set  $P$  to be the period of the AI Engine subsystem. Note that all the inputs and outputs signals of the AI Engine subsystem must have the same period. Later a lower limit will be determined for  $P$ .

**Note:** The PLIO block is a pass-through block and only impacts code generation.

Figure 255: Setting the AIE to HDL Block



### Step 1: Set the PLIO bit width in the PLIO Block

Set the PLIO bit width to  $W$ .

### Step 2: Set Parameters of the AIE to HDL Block

- **Output Data Type:**

Set the Output Data Type such that the output bit width is  $W$ . If  $W$  is larger than the bit width of the input, the output should be unsigned, or else the output should have the same signedness of the input. Note that the input bit width cannot be larger than  $W$ .

- **Output Sample Time:**

Set the Output Sample Time to Inherit: Same as `tready` (this is equivalent of setting this to T). Note that the bit rate into the block is:

**Equation 4: Bit Rate Input**

$$\frac{S \times (\text{DT bit width})}{P}$$

and the output bit rate of the block is:

**Figure 256: Bit Rate Output**

$$\frac{W}{T}$$

For the internal buffers of the block not to overflow, the input rate should be less than or equal to the output rate. However, the HDL design has an initialization interval of `ii`. As such,

**Figure 257: Input Rate ≤ Output Rate**

$$\text{input rate} \leq \frac{\text{output rate}}{ii}$$

or

**Figure 258: Alternate**

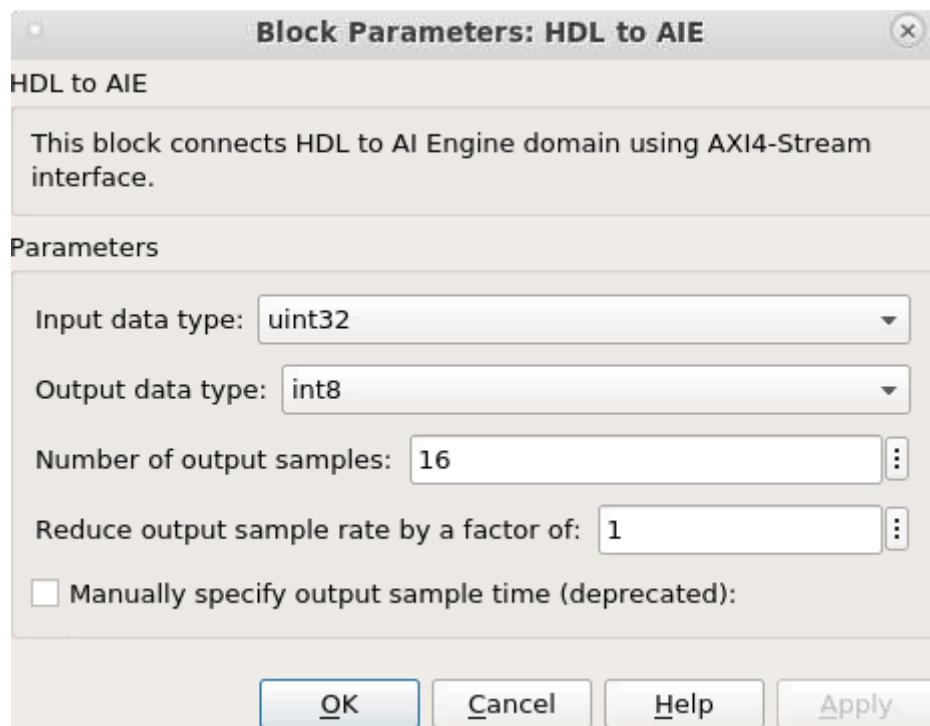
$$P \geq \frac{S \times T \times ii \times (\text{DT bit width})}{W}$$

### **Step 3: Set parameters of the Gateway In AXI Stream block**

- Set **Output data type** to `W`.
- Set **Sample Time** to `T`.

### **HDL to AIE**

The HDL to AIE block connects the output of the HDL block with the input of the AI Engine block. This block accepts `tdata` which is the primary input for the data and the `tvalid` signal that indicates the producer has valid data. Output from the HDL to AIE block is a variable size signal (`data`) to AI Engine blocks along with the `tready` signal which indicates that the block can accept a transfer. A transfer takes place when both `tvalid` and `tready` are asserted.

*Figure 259: HDL to AIE**Figure 260: HDL to AIE Parameters*

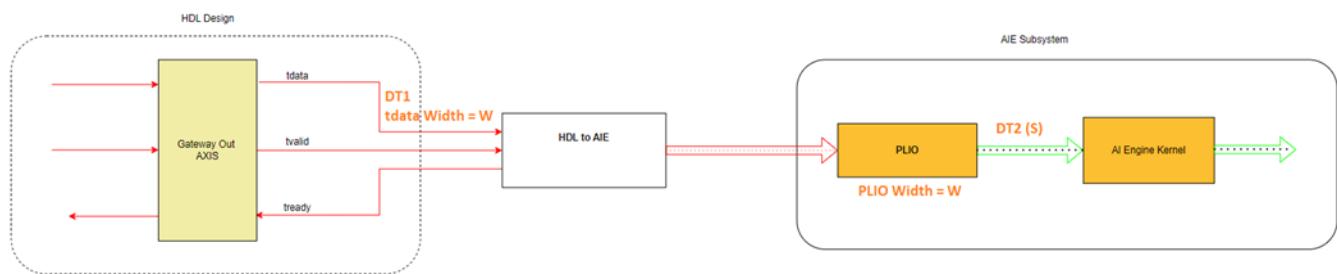
### Setting the HDL-AIE Block

The following image depicts the components that are needed to connect an HDL design to an AI Engine subsystem. In setting this connection, you should consider certain input design criteria and set the parameters of the blocks accordingly. These input design criteria are:

1. The output data type from the HDL design ( $DT_1$ ). The width ( $w$ ) of  $DT_1$  should be 32, 64, or 128 bits for an AXI-S signal.
2. The input data type to the AI Engine kernel block ( $DT_2$ ). This is determined by the AI Engine kernel.

3. The number of samples in the input to the AI Engine kernel block ( $S$ ). For an AI Engine kernel with a buffer input type this is typically the size of the input buffer. For an AI Engine kernel with a stream input, this is typically the number of samples the AI Engine kernel consumes at every invocation.
4. If the HDL design only produces a new sample every  $N$  clock cycles, the output sample rate can be reduced by an optional factor  $N$ .

Figure 261: Setting the HDL to AIE Block



Considering the five design criteria above, set the parameters of the blocks as follows:

#### Step 1: Set the PLIO bit width in the PLIO block

Set the PLIO bit width to  $W$ .

#### Step 2: Set the parameters of the HDL to AIE block

- Set Input data type to DT1.
- Set Output data type to DT2.
- Set Number of output samples to  $S$ .
- Set Reduce output sample rate by a factor of to  $N$ .

#### Step 3: Set the Gateway Out AXIS block

Set the Sample Period parameter to the same value as in the corresponding Gateway In AXI4-Stream block.

# Interconnecting AI Engines and HLS Kernels

## ***HLS Function versus HLS Kernel***

You can import an HLS function into your Vitis Model Composer design using the `xmc ImportFunction` command as described in [Importing C/C++ Code as Custom Blocks](#). You can simulate the block along with other blocks available in the Model Composer HLS library and generate HLS code from a system comprised of one or more imported HLS function blocks and other HLS blocks from the HLS library, except for HLS Kernel block. Vitis Model Composer can also import HLS kernels. This section describes HLS Kernels and how it is different from an HLS function.

## ***Behavior of HLS Functions on Blocking Calls***

An HLS function first and foremost is a function. It has a predetermined number of inputs and outputs and every time the function is invoked, it consumes the inputs and produces the predetermined number of outputs. If an HLS function imported using `xmc ImportFunction` hangs, (for example, if it has an infinite loop), Simulink will also hang, waiting indefinitely for the output from the imported block. This is because an imported HLS function using `xmc ImportFunction` runs on the same thread as Simulink. If the imported functions hangs, Simulink also hangs.

## **HLS Kernels are IPs**

When you import an HLS function into a design by itself, the HLS function will not operate as an IP with streaming ports. In Vitis Model Composer, you need to enclose the HLS function in a subsystem (perhaps along with other HLS blocks) and use the Interface Spec block to designate streaming ports for the design. You can then generate an HLS IP.

Unlike an imported HLS function, an HLS Kernel is a proper HLS IP that can be used in AMD Vitis™ HLS and be synthesized directly. The following code snippet highlights the HLS kernel code with streaming interface.

`hls_kernel.cc`

```
void hls_kernel_blk(
    hls::stream<ap_axis<64, 0, 0, 0>> & in_sample1,
    hls::stream<ap_axis<64, 0, 0, 0>> & in_sample2,
    hls::stream<ap_axis<64, 0, 0, 0>> & out0_itrl,
    hls::stream<ap_axis<64, 0, 0, 0>> & out1_itrl
)
{
    #pragma HLS PIPELINE II=1
    #pragma HLS INTERFACE ap_ctrl_none port=return
    #pragma HLS INTERFACE axis register both port=out1_itrl
    #pragma HLS INTERFACE axis register both port=out0_itrl
    #pragma HLS INTERFACE axis register both port=in_sample1
    #pragma HLS INTERFACE axis register both port=in_sample2
```

```
ap_int64 in_samp0 ; // Iteration-1: 2 complex samples concatenated to  
64-bit  
ap_int64 in_samp1 ; // Iteration-2: 2 complex samples concatenated to  
64-bit  
...
```

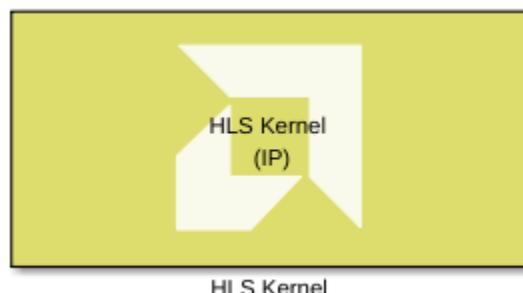
In this example, notice the function signature and also the HLS pragmas specifying the interface on the ports. This function has all the constructs required by the HLS IP. You can directly import the code above into Model Composer using the HLS Kernel block, and then simulate it.

An HLS Kernel block can accept variable size signals allowing it to connect to AI Engine blocks and also produces variable size output signals. Unlike a block that is imported using `xmcImportFunction`, the HLS Kernel block runs on a separate thread than Simulink and as such, the presence of a blocking read call in the HLS kernel code will not cause Simulink to hang when the input variable size signal is empty, instead the block will also produce an empty variable size output signal.

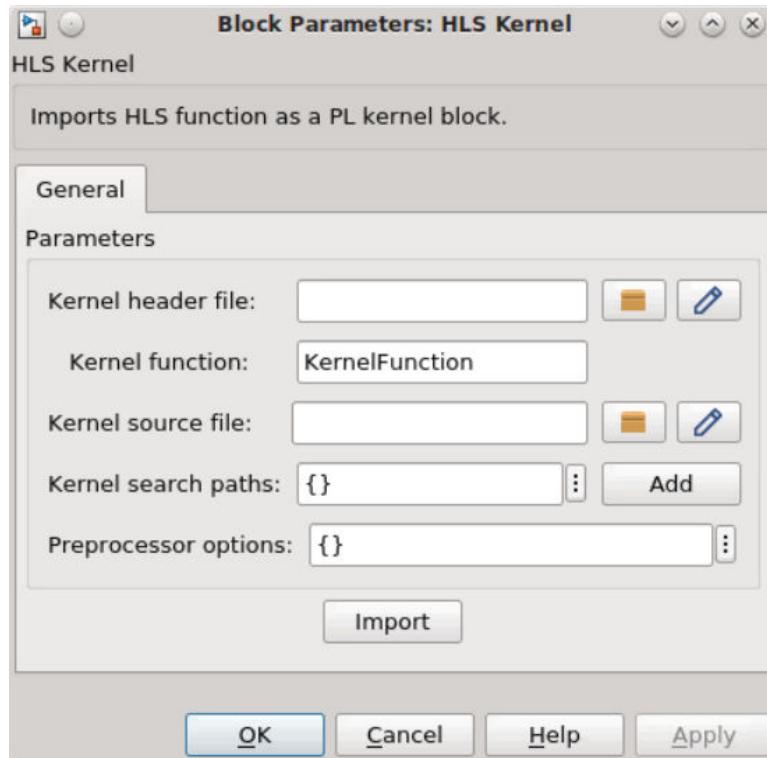
## ***Importing HLS Kernels***

To import the HLS kernel as a block into Vitis Model Composer, you need to select it from the HLS/User-Defined Functions library.

*Figure 262: HLS Kernel*



Double-click the block symbol to display the parameters of the HLS kernel block as shown in the following figure.

**Figure 263: HLS Kernel Parameters**

The block mask parameters need to be updated to import the HLS kernel as a block. The following table provides details on the parameters and descriptions for each parameter.

**Table 36: Parameters**

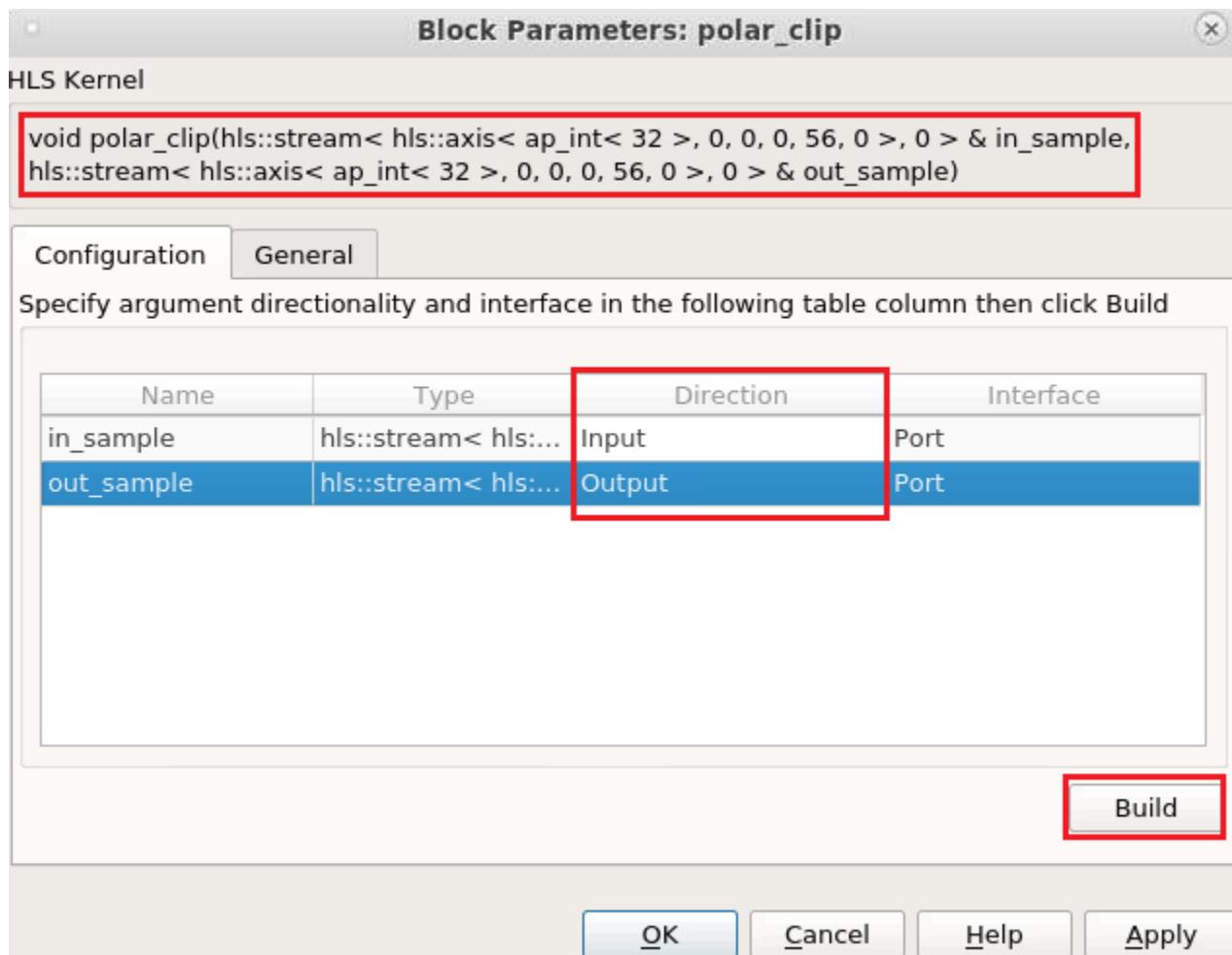
Parameter Name	Parameter Type	Criticality	Description
Kernel header file	String	Mandatory	The name of the HLS kernel header file that contains the function declaration. The string could be just the file name, a relative path to the file, or an absolute path of the file. Use the Browse button to select the file. If environment variables are used to specify the header file path, then an appropriate error is returned.
Kernel function	String	Mandatory	The name of the kernel function in C/C++ for which the HLS kernel block is to be created.
Kernel source file	String	Optional	The name of the source file that contains the kernel function implementation (definition). The string could be just the file name, a relative path to the file or the absolute path of the file. The kernel source file is optional if the header file contains both the declaration as well as the definition of the function. If environment variables are used to specify the source file path, an appropriate error is returned.

Table 36: Parameters (cont'd)

Parameter Name	Parameter Type	Criticality	Description
Kernel search paths	Vector of Strings	Optional	If the kernel header file or the kernel source file are not found using in the current folder, then the paths provided in Kernel search paths are used to locate the files. This parameter allows use of environment variables while specifying paths for the kernel header file and the kernel source file. The environment variable can be used in either \${ENV} or \$ENV format.
Preprocessor options		Optional	Optional preprocessor arguments for downstream compilation with specific preprocessor options. The following two preprocessor option formats will be accepted (multiple can be selected): -Dname and -Dname=definition. That is, the optional argument must begin with the -D string and if the option definition value is not provided, it is assumed to be 1.

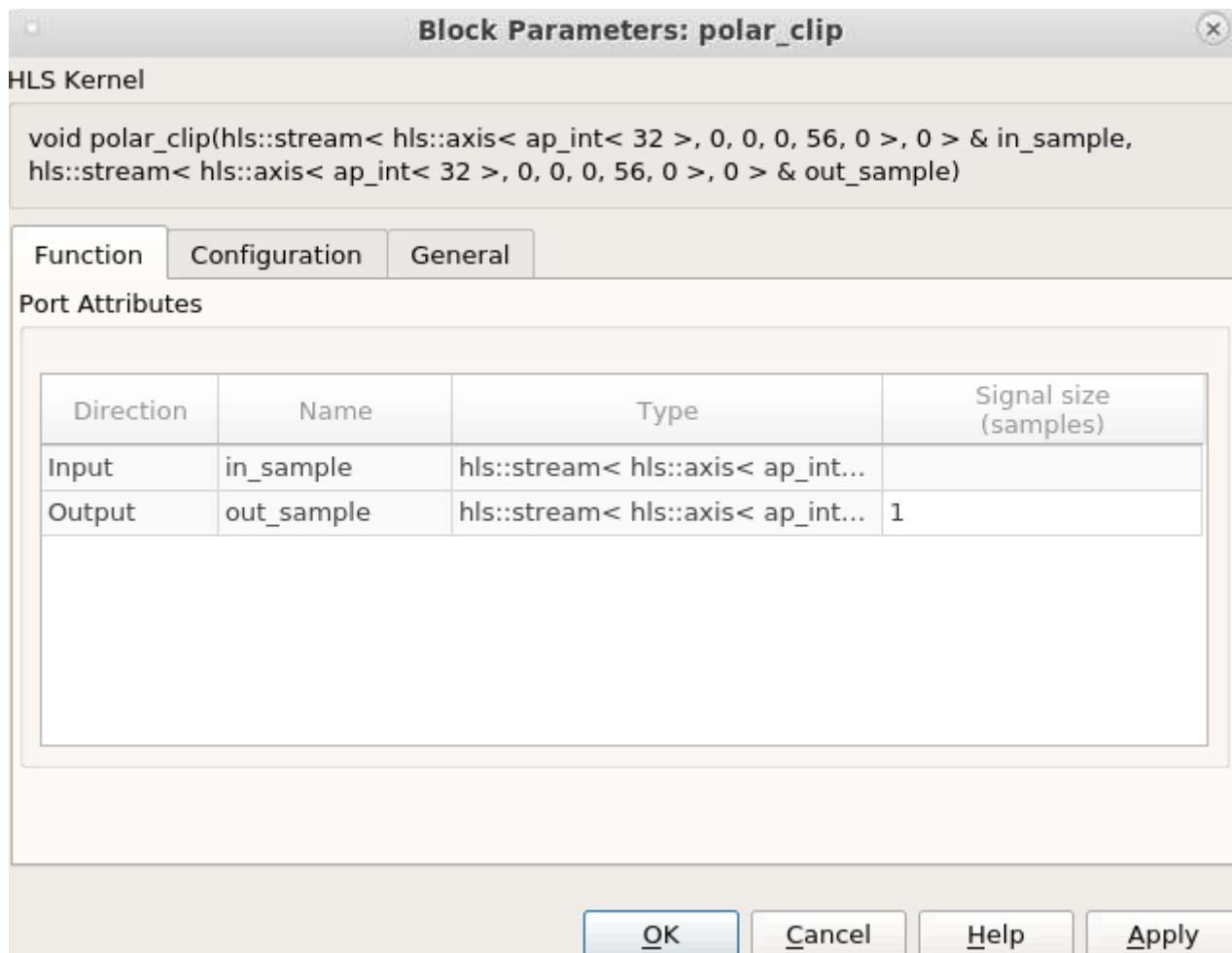
After successful import, the Configuration tab displays automatically. You can quickly review the HLS Kernel definition and specify the directionality of the ports from the drop-down as shown in following figure.

Figure 264: Kernel Definition and Ports



When you click the Build button after specifying the directionality of the ports, a new window opens as shown in the following figure. Do not forget to set the Signal size appropriately for stream signal types or else you might run into a memory overflow. For example, if at every invocation, the kernel produces 16 samples, set Signal Size to 16.

Figure 265: Kernel Definition and Ports (Post-Build)



## Importing Templated Functions

HLS kernel block supports importing a templated function as a block into Vitis Model Composer. Consider the HLS kernel with function template as depicted in the following code.

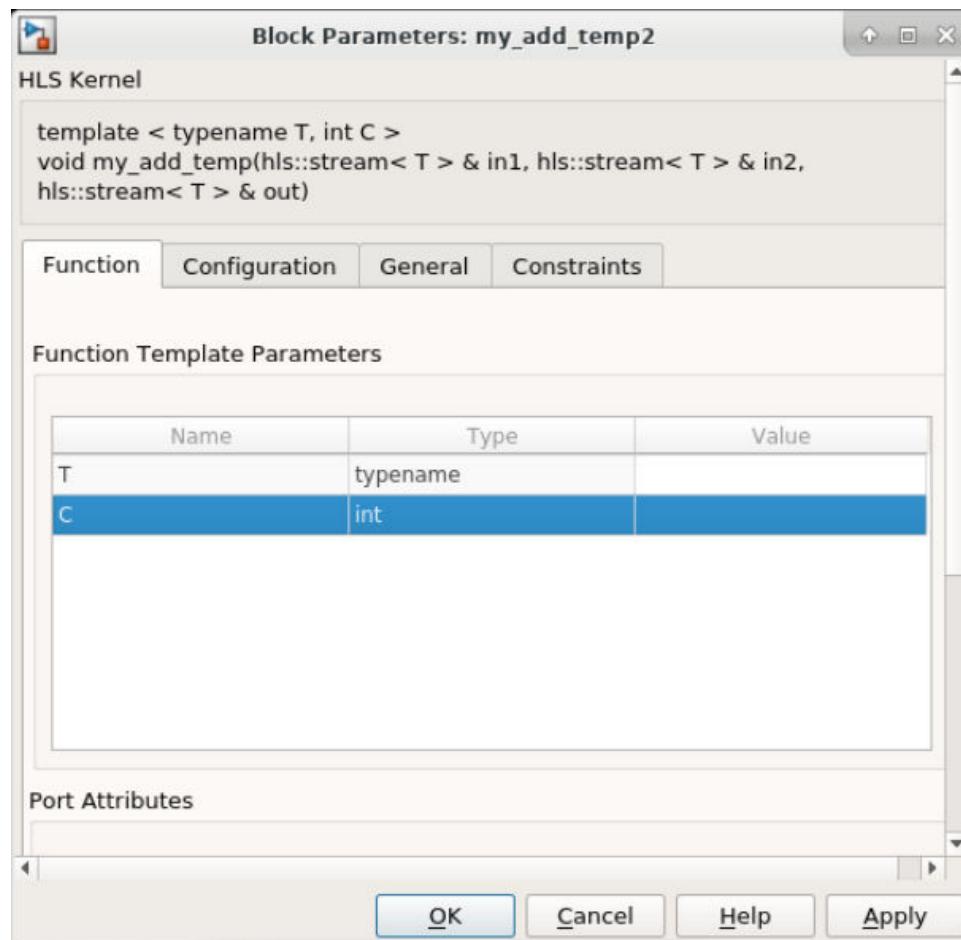
```
template <typename T, int C>
void my_add_temp(hls::stream<T> & in1, hls::stream<T> & in2, hls::stream<T> & out)
{
    T a, b, c;

    a = in1.read();
    b = in2.read();
    c = a + b * C;

    out.write(c);
}
```

After configuring the port directionalities and clicking **Build** as explained in the previous section, the following dialog box opens for setting the template values.

Figure 266: Block Parameters Dialog Box



You can enter the template values in the Function Template Parameters section, click **Apply** then click **OK**. This generates the HLS Kernel block with all the ports in the interface.

## Interconnect AI Engine and HLS Kernel Blocks

Connections between an output port of an AI Engine kernel and an input port of an HLS kernel use an AIE to HLS Kernel block. Connections between an output port of an HLS kernel and an input port of an AI Engine kernel use an HLS Kernel to AIE block. These blocks reformat the data to match the data type of the sink port. In this process no data (information) is lost; and it is simply adjusting the data type and the number of samples. For example, an interface block can reformat a signal carrying 64 int8 values to a signal carrying 16 int32 values. Use of these blocks are not mandatory if the data types between the HLS kernel block and the AI Engine blocks match.

These blocks are available in the AMD Toolbox/Utilites/Connectors library.

## AIE to HLS Kernel

The AIE to HLS Kernel block reformats a signal driven by an AI Engine Kernel block or an AI Engine kernel output port is of type `cint16` and the HLS kernel is of type graph block, so that the resulting signal matches the data type `ap_axis<64>`, you would use an AIE to HLS kernel block with parameter output type set to `uint64`. This block reads `cint16` samples from its input and then packs pairs of subsequent `cint16` samples into `uint64` samples to its output.

Figure 267: AIE to HLS Kernel Block



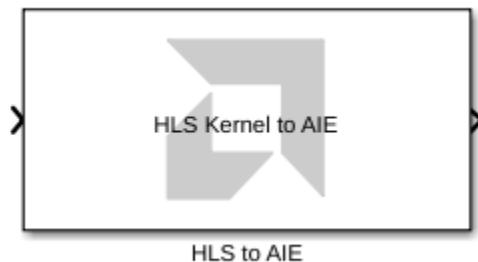
Double-click the block symbol to see the parameters of the AIE to HLS Kernel block.

- **Output Type:** Possible values are: `cint16`, `int32`, `uint32`, `cint32`, `int64`, `uint64`, `sfix128`, `ufix128`, `float`, `cfloat`.
- **Output Size:** The size of the output port. The output port is a variable sized signal whose maximum size is specified by the `OutputSize` parameter. Default Output Size is 1.

## HLS Kernel to AIE

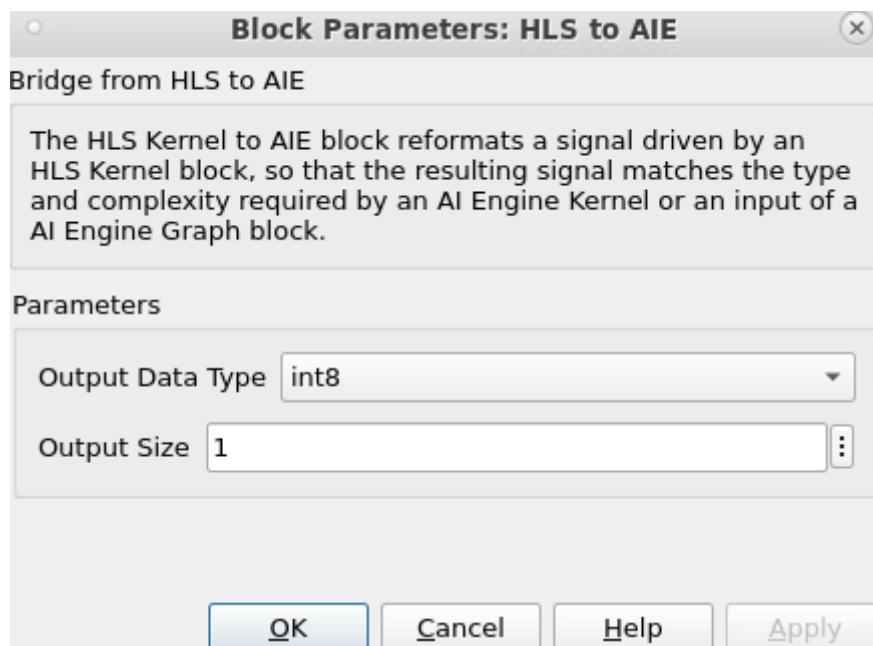
The HLS Kernel to AIE block reformats a signal driven by a port of an HLS kernel block, so that the resulting signal matches the data type and complexity required by an AI Engine kernel or an input of a AI Engine graph block. For example, if the data type of port of the HLS kernel block is `axiu<128>` and the data type of the port of the AI Engine is `uint32`, the HLS Kernel to AIE block reformats the input samples by unpacking each `axisu<128>` sample into four `uint32` samples. The output port of this block is a variable-size signal.

Figure 268: HLS Kernel to AIE Block



Double-click the block symbol to see the parameters of the HLS Kernel to AIE block.

Figure 269: Block Parameters: HLS Kernel to AIE



- **Output Data Type:** Possible values are: int8, int16, int32, int64, uint8, uint16, uint32, uint64, cint16, cint32, float, cfloat, bfloat16, cbfloat16.
- **Output Size:** The size of the output port. The output port is a variable-sized signal whose maximum size is specified by the Output Size parameter. Default size is '1'.

# Connecting Source and Sink Blocks

The AI Engine library is compatible with the standard Simulink block library, and these blocks can be used together to create models that can be simulated in Simulink. However, only certain blocks which are designed to probe at different points in the design and debug are permitted inside a subsystem during code generation. Namely, Scope, Display, To Workspace blocks etc. In addition to these, the AI Engine library provides some sink blocks that can be connected to the variable-size signal output from the AIE Kernel or AIE Graph blocks.

Table 37: Sink Blocks

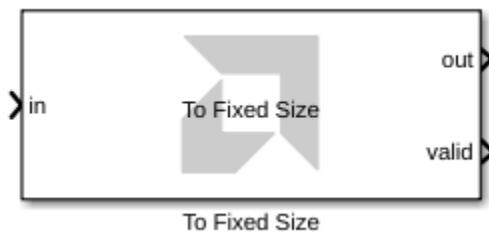
Block	Description
To Fixed Size	Converts variable-size output to fixed size.
Variable Size Signal to Workspace	Logs the variable signal to workspace.

## To Fixed Size

The output ports of the AIE Kernel and AIE Graph blocks are variable-sized (vector) signals. There is a possibility that the kernel does not produce a fixed number of output samples in each simulation step. Many Simulink blocks do not accept variable-size signals as inputs and so this limits leveraging Simulink blocks in designs that use the AIE kernel and AIE Graph blocks.

Model Composer provides the To Fixed Size block which takes a variable-sized vector input and produces a fixed sized vector output.

Figure 270: To Fixed Size



The 'To Fixed Size' block takes a variable-size input signal and produces a fixed-size output signal. There are two available modes for this block:

- **Stop simulation when input variable-size signal is not full:** In this mode, the block will stop the simulation and error out if the input variable-size signal is not full. The output signal is set to the same size as the input signal.

- **Produce a valid output signal:** In this mode, the input will be buffered until the number of samples reaches the Output Size. The buffered samples will then be transferred to the output, and the valid port is set to true. When there are not enough samples buffered, the output will be a vector of zeros, and the valid port is set to false.

### Variable Size Signal to Workspace

The output ports of AIE Kernel and AIE Graph blocks are variable-sized signals. Model Composer provides a Variable Size Signal to Workspace block in the AI Engine library to easily save the output into a workspace variable in MATLAB. The block will save only the valid samples of the input variable size signal, making analysis of the signal much easier.

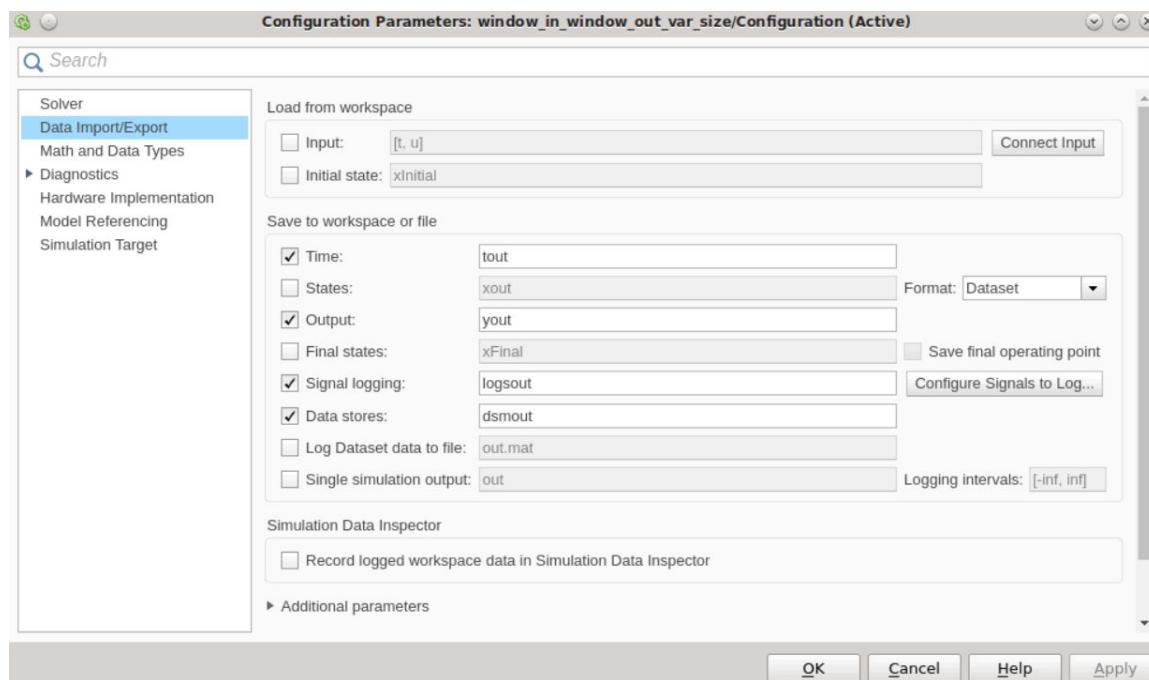
*Figure 271: Variable Size Signal to Workspace*



Within the block parameters, the default name for Variable name is set to 'simout'.

Because the Variable Size Signal to Workspace block is using the Simulink To Workspace block under the mask, settings that change the behavior of the To Workspace block will impact the Variable Size Signal to Workspace block as well. The block settings can be accessed from Model Settings (Ctrl-E).

Figure 272: Configuration Parameters



You can set the name of the output (default is `out`) from the settings window and you can also control whether the To Workspace block outputs the results in a structure called `out`. Regardless of whether the checkbox is selected or what name is chosen, this block will work in a similar way as the To workspace block.

If you toggle the Single simulation output check box from Model Settings, press Ctrl-D to refresh the text on the Variable Size Signal to Workspace block.



**IMPORTANT!** If there is an error in the simulation, the block creates an empty variable when Single simulation output is not checked from the Model Settings window. This behavior is consistent with the To Workspace block.

Connecting the AI Engine block to source blocks that generate or import signal data (Constant, Signal to Workspace block etc.) is supported.

# Model Composer Utilities

## AI Engine Utilities

xmcLibraryPath	To set the root of the DSPLib from a MATLAB® command window to a custom location in your local directory.
xmcVitisRead	Reads AMD Vitis™ data files and outputs them into MATLAB.
xmcVitisWrite	Takes an existing array and creates a file that can be read by AIEsimulator and x86simulator.

### xmcLibraryPath

`xmcLibraryPath` is a MATLAB utility that sets the root of the DSPLib from a MATLAB command window to a custom location in your local directory, using the MATLAB utility `xmcLibraryPath`.

#### Syntax

```
xmcLibraryPath( '<Arg1>', 'dsplib', <Arg2>) ;
```

'<Arg1>' should be one of the options specified in the following table.

*Table 38: Arg options*

Arg1	Description
'set'	Set the DSPLib root to the custom location.
'reset'	Reset the DSPLib root to the default location.
'get'	To get the current DSPLib root location.
'getDefault'	To get the default DSPLib root location.

#### Example 1

```
xmcLibraryPath('set', 'dsplib', '/workspace/Github_Repo/Vitis_Libraries/dsp');
```

This option sets the DSPLib to point to the code from GitHub repository.

## Example 2

```
xmcLibraryPath('reset', 'dsplib');
```

This option sets the DSPLib to the Model Composer installed directory, which is the default location.

## xmcVitisRead

xmcVitisRead is a MATLAB utility that reads data files and outputs them into MATLAB. The function supports real and complex, signed, and unsigned numbers. If the file contains timestamps, they will be added to a separate, parallel array.

### Syntax

```
A = xmcVitisRead(fileIn,dataType)
[A, TS] = xmcVitisRead(fileIn,dataType)
```

Where, the option `fileIn` is the file path and the `dataType` represents the datatype of the imported data.

The following table shows the list of datatypes and maximum possible columns allowed in the input file.

*Table 39: Data Types*

Data Types	Data Size (bits)	Maximum Possible Columns
int8	8	16
int16	16	8
int32	32	4
int64	64	2
uint8	8	16
uint16	16	8
uint32	32	4
uint64	64	2
cint16	32	4
cint32	64	2
float	32	4
cfloat	64	2

## Example

Let `Input.txt` be a file of the following format containing real numbers:

```
"T 470 ns
    1651 -17 6646 -5720
    T 472 ns
    8850 -2469 2711 7752
    T 474 ns
    -4938 -6103 -4659 -2352
    T 476 ns
    -2144 -6453 1410 5685
    T 478 ns
    -1591 1962 1190 8775"
```

The following function call imports the array as MATLAB `int16` values.

```
A = xmcVitisRead("Input.txt",'int16')
```

This is the resulting column vector:

```
A = [
    1651,
    -17,
    6646,
    -5720,
    8850,
    -2469,
    2711,
    7752,
    -4938,
    -6103,
    -4659,
    -2352,
    -2144,
    -6453,
    1410,
    5685,
    -1591,
    1962,
    1190,
    8775]
```

The following function call produces two arrays: signal and timestamp. The result is as follows.

```
[A, timestamp] = xmcVitisRead("Input.txt",'int16')

TS = [
    470,          A = [
    470,          1651,
    470,          -17,
    470,          6646,
    470,          -5270,
    472,          8850,
    472,          -2469,
    472,          2711,
    472,          7752,
    474,          -4938,
    474,          -6103,
    474,          -4659,
```

```
474,      -2352,
476,      -2144,
476,      -6453,
476,      1410,
476,      5685,
478,      -1591,
478,      1963,
478,      1190,
478]      8775]
```

 **IMPORTANT!** The function checks for the number of columns and returns an error if the number of columns exceeds the theoretical maximum for that data type.

 **IMPORTANT!** If the file contains timestamps but they are not required, the `timestamp` output can be left blank. Conversely, if the file does not have timestamps but two outputs are specified, the `timestamp` output will be an empty numeric array.

## xmcVitisWrite

This function takes an existing array and creates a file that can be read by AIEmulator and x86simulator. The input array can be of any real or complex data type.

- Complex data types are automatically detected, therefore only the data type (for example, `int16`) needs to be specified.
- The function also accepts the PLIO width and uses the data type to calculate the number of columns in the file. If the PLIO width is less than the size of the data type, the function will return an error.

### Syntax

```
xmcVitisWrite(fileName, arrayIn, dataType, widthPLIO)
```

Where:

- `fileName` is the output file name.
- `arrayIn` is the Input array.
- `dataType` is the datatype of the data.
- `widthPLIO` is the PLIO width (must be 32, 64, or 128)

The following table shows the list of datatypes and maximum possible columns in the output file, based on the PLIO width.

*Table 40: Data Types*

Data Types	Data Size (bits)	Maximum Possible Columns
int8	8	16
int16	16	8

Table 40: Data Types (cont'd)

Data Types	Data Size (bits)	Maximum Possible Columns
int32	32	4
int64	64	2
uint8	8	16
uint16	16	8
uint32	32	4
uint64	64	2
cint16	32	4
cint32	64	2
float	32	4
cfloat	64	2

### Example

Let A be an array of complex numbers of type `cfloat` and the PLIO width is set to 128.

```
A = [
      3.142 + 1.463i,
      6.288 + 3.079i,
      3.333 + 1.493i,
      3.781 + 8.781i,
      3.142 + 1.463i]
```

**Note:** The data can be either a row or column vector.

The function call is as follows.

```
xmcVitisWrite("Output.txt", A, 'cfloat', 128);
```

The file will be saved as `Output.txt` in the working directory, as specified in the function call. The file content is as follows.

```
"3.142 1.463
6.288 3.079
3.333 1.493
3.781 8.781
3.142 1.463"
```

The columns are calculated automatically from the size of the data type and the PLIO width. A complex number is divided into two columns in the file, corresponding to real and imaginary parts in that order. A file with complex values will have at least two columns, while one with real values will have at least one.

# HDL Utilities

*Table 41: HDL Utilities*

<code>xilinx.analyzer</code>	Provides the interface between the Model Composer model and AMD Vivado™ timing paths.
<code>xilinx.environment.getcachepath</code> and <code>xilinx.environment.setcachepath</code>	Used to get and set the path Model Composer uses to store the simulation cache.
<code>xilinx.resource_analyzer</code>	Enables cross-probing between the Model Composer model and Vivado resource utilization data.
<code>xlfd_a_denominator</code>	Returns the denominator of the filter object in an FDATool block.
<code>xlfd_a_numerator</code>	Returns the numerator of the filter object in an FDATool block.
<code>vmcGenerate</code>	Provides a programmatic way to invoke the Model Composer code generator.
<code>vmchub_set_param</code>	Used to set parameter values in a HDL block.
<code>xlGetReOrderedCoeff</code>	The <code>xlGetReOrderedCoeff</code> function provides the re-ordered coefficient set of a FIR Compiler block.
<code>xlOpenWaveFormData</code>	Allow you to populate saved simulation waveform data into running Waveform Viewer instance.

## xilinx.analyzer

`xilinx.analyzer` is a MATLAB® class that provides an interface between the Model Composer model and AMD Vivado™ timing paths.

The Model Composer timing analysis is supported for all compilation targets. The **Perform analysis** drop down menu under the **Clocking** tab of the Model Composer Hub block provides two options for the trade-off between total runtime vs. accuracy of the Vivado timing data. If you select either the **Post Synthesis** or the **Post Implementation** option of **Perform analysis** and click the **Generate** button, then Vivado timing paths information is collected during the netlist generation. The `xilinx.analyzer` class is used to access Vivado timing paths information. The `xilinx.analyzer` class object processes Vivado timing paths to find 50 unique paths with the worst slack value. The unique timing paths are sorted in increasing value of slack and saved in the analyzer object.

The cross-probing between Vivado timing paths and the Model Composer model is made possible using the following API functions in the `xilinx.analyzer` class.

*Table 42: xilinx.analyzer Class Functions*

Function Name	Description	Function Argument
<code>xilinx.analyzer</code>	This is a constructor of the class. A call to the <code>xilinx.analyzer</code> constructor returns object of the class.	First argument is Model Composer model name. Second argument is path to already generated netlist directory.

**Table 42: xilinx.analyzer Class Functions (cont'd)**

<b>Function Name</b>	<b>Description</b>	<b>Function Argument</b>
<code>isValid</code>	Indicates if timing analysis data is valid or not. Use this API to make sure that the <code>xilinx.analyzer</code> class construction was successful.	No argument
<code>getErrorMessage</code>	Returns an error message string if the call to the class constructor or other API function had an error.	No argument
<code>getStatus</code>	Returns 'FAILED' if any of the timing paths in the model have a violation, that is, negative slack.	No argument
<code>getVivadoStage</code>	Returns either Post Synthesis or Post Implementation. This is the Vivado design stage after which timing analysis was performed.	No argument
<code>paths</code>	Returns an array of MATLAB structures. Each structure contains data for a timing path.	A string that is equal to either 'setup' or 'hold'
<code>violations</code>	Returns an array of MATLAB structures. Each member of the array is a path structure with a timing violation.	A string that is equal to either 'setup' or 'hold'
<code>print</code>	Prints timing path information such as Slack, Path Delay, Levels of Logic, Name of Source and Destination blocks, and Source and Destination clocks.	An array of MATLAB structures for timing path data. The array can have one or more structures.
<code>highlight</code>	In the Model Composer model, highlights blocks for the timing path passed in the argument. Blocks that are already highlighted in the model will remain highlighted.	MATLAB structure for one timing path
<code>highlightOnePath</code>	In the Model Composer model, highlights blocks for the timing path passed in the argument. Before highlighting blocks for this path, the blocks that are already highlighted in the model will be unhighlighted.	MATLAB® structure for one timing path
<code>unhighlight</code>	In the Simulink® model, unhighlights all blocks currently highlighted.	No argument
<code>disp</code>	Displays a summary of timing analysis results on the MATLAB console, including the worst slack value among all timing paths.	No argument
<code>delete</code>	This is a destructor of the <code>xilinx.analyzer</code> class	No argument

**Table 43: Timing Path Data in a MATLAB Structure**

<b>Field Name</b>	<b>Description</b>
<code>Slack</code>	The double value containing timing slack for the path
<code>Delay</code>	Total Data Path delay for the path
<code>Levels_of_Logic</code>	Number of elements in Vivado design for the timing path. The number of HDL blocks in the timing path might be different from <code>Levels_of_Logic</code> .
<code>Source</code>	First HDL block in the timing path
<code>Destination</code>	Last HDL block in the timing path
<code>Source_Clock</code>	Name of the clock domain for the source block
<code>Destination_Clock</code>	Name of the clock domain for the destination block

**Table 43: Timing Path Data in a MATLAB Structure (cont'd)**

Field Name	Description
Path_Constraints	Timing constraint used for the path. For a multi-clock design, the path constraint can be a multi-clock timing constraint.
Block_Masks	Cell array where each element contains mask information for a HDL block.
Simulink_Names	Cell array where each element contains hierarchical name of a block in Model Composer model
Vivado_Names	Cell array where each element contains name of HDL block in Vivado database
Type	A timing violation type. The value is either 'setup' or 'hold'.

### xilinx.analyzer - Construct xilinx.analyzer class object

```
analyzer_object = xilinx.analyzer(<name_of_the_model> ,  
'<path_to_netlist_directory>')
```

- **Description:**

A call to xilinx.analyzer constructor returns object of the class.

The first argument is the name of the Model Composer model. The model must be open before the class constructor is called.

The second argument is an absolute or relative path to the netlist directory. You must have read permission to the netlist directory.

To access API functions of the xilinx.analyzer class use the object of the class as described below. To get more details for a specific API function type the following at the MATLAB command prompt:

```
help xilinx.analyzer.<API_function>
```

- **Example:**

```
//Construct class. Must give the model name and absolute or relative path  
to the  
//target directory  
  
>> timing_object = xilinx.analyzer('fixed_point_IIR', './  
netlist_for_timing_analysis')  
  
timing_object =  
  
Number of setup paths = 9  
Worst case setup slack = -1.6430
```

### isValid - Check validity of Vivado timing paths

- **Syntax:**

```
result = analyzer_object.isValid();
```

- **Description:** If timing analysis data is valid then the result equals '1', otherwise it is '0'. Use this API to make sure that the xilinx.analyzer class construction was successful and the timing data was valid.
- **Example:**

```
//Determine if timing analysis data is valid  
>> valid_status = timing_object.isValid()  
valid_status =  
1
```

### getErrorMessage - Get an error message

- **Syntax:**

```
result = analyzer_object.getErrorMessage();
```

- **Description:** Returns an error message string if the call to the class constructor or other API function had an error.
- **Example:**

```
//Determine if there was an error in the xilinx.analyzer constructor  
//or in any of the API functions  
>> err_msg = timing_object.getErrorMessage()  
err_msg =  
''
```

### getStatus - Timing analysis status

- **Syntax:**

```
string = analyzer_object.getStatus();
```

- **Description:** The returned string is either 'PASSED' or 'FAILED'. If any of the timing paths have a violation, that is, negative slack, then the timing analysis status is considered failed.
- **Example:**

```
//Determine if there were timing path violations in Simulink model  
>> analysis_status = timing_object.getStatus()  
analysis_status =  
FAILED
```

## getVivadoStage - Get Vivado design stage for timing analysis

- **Syntax:**

```
string = analyzer_object.getVivadoStage();
```

- **Description:** The returned string is the Vivado design stage after which timing analysis was performed and data collected in Vivado. The value is either 'Post Synthesis' or 'Post Implementation'.

- **Example:**

```
//Determine Vivado stage when timing data was collected  
>> design_stage = timing_object.getVivadoStage()  
  
design_stage =  
  
Post Synthesis
```

## paths - Access all timing paths

- **Syntax:**

```
<array_of_timing_paths_structure> =  
analyzer_object.paths('<violation_type>');
```

- **Description:** The returned value is an array of MATLAB structures. Each structure contains data for a timing path, sorted in decreasing order of timing violation, that is, in increasing order of slack value.

The argument `violation_type` is either 'setup' or 'hold' string.

- **Example:**

```
//Return an array of the timing path structures  
>> all_timing_paths = timing_object.paths('setup')  
  
all_timing_paths =  
  
1x9 struct array with fields:  
  
    Slack  
    Delay  
    Levels_of_Logic  
    Source  
    Destination  
    Source_Clock  
    Destination_Clock  
    Path_Constraints  
    Block_Masks  
    Simulink_Names  
    Vivado_Names  
    Type
```

**Note:**

There are a total of nine timing paths in this timing analysis.

You can find the data fields in each timing path as shown in Example 1 in [Additional Information](#).

**violations - Access paths with timing violations**

- **Syntax:**

```
<array_of_timing_paths_structure> =
analyzer_object.violations('<violation_type>');
```

- **Description:** The returned value is an array of MATLAB structures. Each member of the array is data for a path with a timing violation. The array elements are sorted in decreasing order of timing violation. If there are no timing violations in the design then the API function returns an empty array.

The argument `violation_type` is either 'setup' or 'hold'.

- **Example:**

```
//Return an array of timing paths with setup violations
>> violating_paths = timing_object.violations('setup')
violating_paths =
1x2 struct array with fields:
    Slack
    Delay
    Levels_of_Logic
    Source
    Destination
    Source_Clock
    Destination_Clock
    Path_Constraints
    Block_Masks
    Simulink_Names
    Vivado_Names
    Type
```

There are a total of two paths with violations in this timing analysis.

You can find the data fields in each timing path as shown in Example 3 in [Additional Information](#).

**print - Print timing path information**

- **Syntax:**

```
analyzer_object.print(<timing_path_structures>);
```

- **Description:**

Prints timing data such as Slack, Path Delay, Levels of Logic, Name of Source and Destination blocks, Source and Destination clocks, Path Constraints, etc. for the input timing path structure.

The argument is an array of MATLAB structures with one or more elements.

- **Examples:**

```
//Print timing path information for path #1

>> timing_object.print(all_timing_paths(1))
Path Num          Slack (ns)          Delay (ns)          Levels
of Logic
Source/Destination Blocks          Source Clock          Destination
Clock          Path
Constraints
  1           -1.6430
11.5690          6
fixed_point_IIR/Delay1
clk          clk
create_clock -name clk -period 2 [get_ports clk]

fixed_point_IIR/IIR Filter Subsystem/Delay4

ans =
  1

//Print timing path information for path #3

>> timing_object.print(all_timing_paths(3))
Path Num          Slack (ns)          Delay (ns)          Levels
of Logic
Source/Destination Blocks          Source Clock          Destination
Clock          Path
Constraints
  1           1.1320
0.5270          0
fixed_point_IIR/Delay1
clk          clk
create_clock -name clk -period 2 [get_ports clk]

fixed_point_IIR/Delay1

ans =
  1

//Print timing path information for path #2 from violating_paths array

>> timing_obj.print(violating_paths(2))
Path Num          Slack (ns)          Delay (ns)          Levels
of Logic
Source/Destination Blocks          Source Clock          Destination
Clock          Path
Constraints
  1           -1.3260
```

```

11.2520          6
fixed_point_IIR/Delay1
clk                  clk
create_clock -name clk -period 2 [get_ports clk]

fixed_point_IIR/Delay2

ans =
1

```

## highlight - Highlight design blocks for a timing path

- Syntax:

```
analyzer_object.highlight(<timing_path_structure>);
```

- **Description:** This API highlights HDL blocks for the timing path passed in the argument. It doesn't change the highlighting of a block from other paths, so more than one timing path can be highlighted if you use this function repeatedly.

The argument is the MATLAB structure for one timing path.

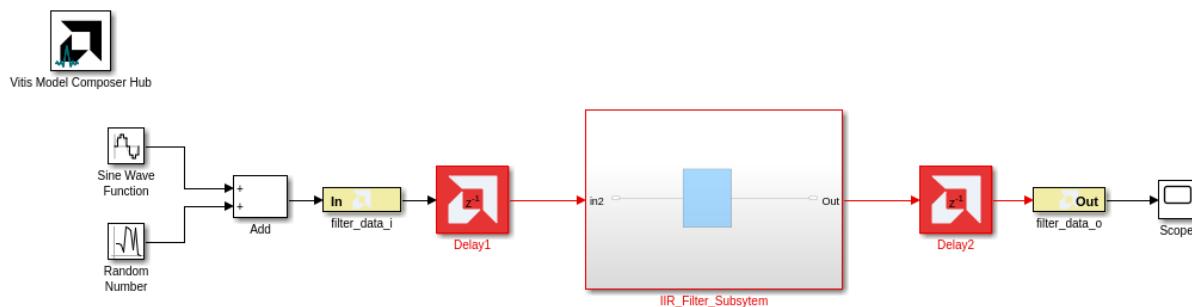
- **Example:**

```
//Highlight Simulink model blocks in the selected path
//Don't change highlighting of currently highlighted blocks in the model

>> [result, err_msg] = timing_object.highlight(all_timing_paths(1));
```

Highlighted Model Composer model blocks appear as shown below.

*Figure 273: HDL Model Blocks*



## highlightOnePath - Highlight design blocks for one timing path

- **Syntax:**

```
analyzer_object.highlightOnePath(<timing_path_structure>);
```

- **Description:**

This API highlights HDL blocks for the timing path passed in the argument. If a block from other paths is already highlighted then it will be unhighlighted first, so only one path is highlighted at a time.

The argument is the MATLAB structure for one timing path.

- **Example:**

```
//Highlight a single path in Model Composer model, and unhighlight  
//currently  
//highlighted paths  
>> [result, err_msg] = timing_object.highlightOnePath(violating_paths(2));
```

## unhighlight - Unhighlight design blocks

- **Syntax:**

```
analyzer_object.unhighlight();
```

- **Description:** This API unhighlights blocks that are already highlighted. The blocks in Model Composer model are displayed in their original colors.

- **Example:**

```
//Unhighlight any Simulink block that is currently highlighted  
>> [result, err_msg] = timing_object.unhighlight();
```

## disp - Display summary of timing analysis

- **Syntax:**

```
analyzer_object.disp();
```

- **Description:** This API displays the summary of timing paths on the MATLAB console, including the worst slack value.

- **Example:**

```
//Display a summary of timing analysis  
>> timing_object.disp()  
Number of setup paths = 9  
Worst case setup slack = -1.6430
```

## delete - Delete xilinx.analyzer class object

- **Syntax:**

```
analyzer_object.delete();
```

- **Description:** This is a destructor for the xilinx.analyzer class.

- **Example:**

```
//Delete xilinx.analyzer object, i.e., timing_object  
>> timing_object.delete();
```

## Additional Information

Accessing data fields of timing path structures:

- **Example 1: Data for timing path #1:**

```
//Return the data fields for the timing path with the worst slack  
>> all_timing_paths(1)  
  
ans =  
  
    Slack: -1.6430  
    Delay: 11.5690  
    Levels_of_Logic: 6  
        Source: 'fixed_point_IIR/Delay1'  
        Destination: 'fixed_point_IIR/IIR Filter Subsystem/Delay4'  
        Source_Clock: 'clk'  
    Destination_Clock: 'clk'  
    Path_Constraints: 'create_clock -name clk -period 2 [get_ports ...]'  
        Block_Masks: {1x5 cell}  
        Simulink_Names: {1x5 cell}  
        Vivado_Names: {1x5 cell}  
        Type: 'setup'
```

- **Example 2: Data for timing path #3:**

```
//Return the data fields for a timing path  
>> all_timing_paths(3)  
  
ans =  
  
    Slack: 1.1320  
    Delay: 0.5270  
    Levels_of_Logic: 0  
        Source: 'fixed_point_IIR/Delay1'  
        Destination: 'fixed_point_IIR/Delay1'  
        Source_Clock: 'clk'  
    Destination_Clock: 'clk'  
    Path_Constraints: 'create_clock -name clk -period 2 [get_ports ...]'  
        Block_Masks: {'fprintf('','COMMENT: begin icon graphics')...'}  
        Simulink_Names: {'fixed_point_IIR/Delay1'}  
        Vivado_Names: {'fixed_point_iir.fixed_point_iir_struct.delay1'}
```

Type: 'setup'

- **Example 3: Data for path #1 in violating\_paths array:**

```
//Return the data fields in a timing path with timing violations
>> violating_paths(1)
ans =
    Slack: -1.6430
    Delay: 11.5690
    Levels_of_Logic: 6
        Source: 'fixed_point_IIR/Delay1'
        Destination: 'fixed_point_IIR/IIR Filter Subsystem/Delay4'
        Source_Clock: 'clk'
    Destination_Clock: 'clk'
    Path_Constraints: 'create_clock -name clk -period 2 [get_ports ...]'
        Block_Masks: {1x5 cell}
        Simulink_Names: {1x5 cell}
        Vivado_Names: {1x5 cell}
        Type: 'setup'
```

## xilinx.environment.getcachepath and xilinx.environment.setcachepath

`xilinx.environment.getcachepath` is used to get the path Model Composer currently uses to store the simulation cache.

`xilinx.environment.setcachepath` is used to change the path Model Composer uses to store the simulation cache.

### Syntax

```
xilinx.environment.getcachepath
xilinx.environment.setcachepath(path)
```

### Description

When you simulate a Simulink model containing AMD IP in Model Composer, the Vivado simulator simulation data for that particular IP configuration is cached to speed up the simulation.

Model Composer establishes the simulation cache at a default location at startup, and you can determine the current path to the simulation cache with the `xilinx.environment.getcachepath` command. If you need to change the location of the simulation cache, use the `xilinx.environment.setcachepath` command. You will need to have write permission on the destination path directory. The new path will apply for the remainder of your Model Composer session.

One reason you would use `xilinx.environment.setcachepath` is to set the path if you do not have write permission on the default directory Model Composer uses for caching simulation data.

## Examples

- **Example 1: Getting the current simulation cache path:**

```
>> xilinx.environment.getcachepath  
ans =  
C:\Users\my_login\AppData\Local\Xilinx\Sysgen\SysgenVivado\win64.o
```

- **Example 2: Setting a new simulation cache path:**

```
>> xilinx.environment.setcachepath('C:\sim_cache')  
ans =  
C:\sim_cache  
>> xilinx.environment.getcachepath  
ans =  
C:\sim_cache
```

## xilinx.resource\_analyzer

`xilinx.resource_analyzer` is a MATLAB class that enables cross-probing between the Model Composer model and Vivado resource utilization data.

The Model Composer resource analysis is supported for all compilation targets. The **Perform analysis** drop down menu under the **Clocking** tab of the Model Composer Hub block provides two options for the trade-off between Vivado tools runtime vs. accuracy of the resource utilization data. If you select either the **Post Synthesis** or the **Post Implementation** option of **Perform analysis** and click the **Generate** button, then Vivado resource utilization information

is collected during the netlist generation. Once the netlist generation has completed, the `xilinx.resource_analyzer` class is used to access this Vivado resource utilization results. The `xilinx.resource_analyzer` class object processes Vivado resource utilization data to display the number of resources (BRAMs, DSPs, Registers, and LUTs) used in the Simulink model, as well as by the subsystems and low-level blocks in the model.

The cross-probing between Vivado resource utilization results and the Model Composer model is made possible through the following API functions in the `xilinx.resource_analyzer` class.

**Table 44: Functions in `xilinx.resource_analyzer` Class**

Function Name	Description	Function Argument
<code>xilinx.resource_analyzer</code>	This is a constructor of the class. A call to the <code>xilinx.resource_analyzer</code> constructor returns object of the class.	First argument is design model name. Second argument is path to already generated netlist directory.
<code>getVivadoStage</code>	Returns either Post Synthesis or Post Implementation. This is the Vivado design stage after which resource analysis was performed.	No argument
<code>getDevicePart</code>	Returns a string for device part, package, and speed grade for the device in which the design will be implemented.	No argument
<code>getDeviceResource</code>	Returns a string for the total count of the specified type of resource in the target AMD device.	(Optional) Resource type. Resource types are: BRAMs, DSPs, Registers, or LUTs.
<code>printDeviceResources</code>	Prints the total number of BRAMs, DSPs, Registers, and LUTs available on the target AMD device. The counts are printed in the MATLAB console.	No argument.
<code>getCount</code>	Returns a count for the particular resource type used by a block or subsystem.	(Optional) First argument is a Simulink handle or pathname for the block. (Optional) Second argument is Resource type. Resource types are: BRAMs, DSPs, Registers, or LUTs.
<code>print</code>	Returns a count for the particular resource type used by a block or subsystem.	(Optional) First argument is a Simulink handle or pathname for the block or subsystem. (Optional) Second argument is resource type. Resource types are: BRAMs, DSPs, Registers, or LUTs.
<code>getDistribution</code>	Returns three values: An array of MATLAB structures. Each element in the array is a structure containing the name of a block or subsystem directly under the subsystem in the argument, with a key-value pair of the resource type and number of resources used by that sub block or subsystem. A count of the resources used by the self (the block or subsystem specified in the argument). A count of the resources used by both blocks and subsystems combined.	First argument is a Simulink handle or pathname for the block or subsystem. Second argument is resource type. Resource types are: BRAMs, DSPs, Registers, or LUTs.

**Table 44: Functions in `xilinx.resource_analyzer` Class (cont'd)**

Function Name	Description	Function Argument
<code>getErrorMessage</code>	Returns an error message string if the call to the class constructor or other API function had an error.	No argument
<code>highlight</code>	In the Simulink model, highlights the specified block or subsystem with yellow color and red border.	A Simulink handle or pathname for the block to highlight.
<code>unhighlight</code>	In the Simulink model, unhighlights a block which is currently highlighted.	(Optional) A Simulink handle or pathname for the block to unhighlight.
<code>delete</code>	This is a destructor of the <code>xilinx.resource_analyzer</code> class.	No argument

**Table 45: Resource Data in a MATLAB Structure**

Field Name	Description
BRAMs	<p>Count of block RAM resources for a block or subsystem. BRAMs are counted in this way:</p> <ul style="list-style-type: none"> <li>• RAMB36E: 1 BRAM</li> <li>• RAMB36E: 1 BRAM</li> <li>• RAMB18E: 0.5 BRAM</li> <li>• FIFO18E: 0.5 BRAM</li> </ul> <p>Variations of Primitives (for example, RAM36E1 and RAM36E2) are all counted in the same way. Total BRAMs = (Number of RAMB36E) + (Number of FIFO36E) + 0.5 (Number of RAMB18E + Number of FIFO18E)</p>
DSPs	Count of DSP48 resources utilized by a block or subsystem.
Registers	Count of Flip-Flops and Latches used by the design is reported as the number of Registers utilized by the design model, a particular block, or a subsystem.
LUTs	Count of all LUT type resources used by a block or subsystem.

### **`xilinx.resource_analyzer` – Construct `xilinx.resource_analyzer` class object**

- **Syntax:**

```
resource_analyzer_obj =
xilinx.resource_analyzer('<name_of_the_model>', '<path_to_netlist_directory>');
```

- **Description:**

A call to `xilinx.resource_analyzer` constructor returns object of the class.

The first argument is the name of the Model Composer model. The model must be open before the class constructor is called.

The second argument is an absolute or relative path to the netlist directory. You must have read permission to the netlist directory.

To access API functions of the `xilinx.resource_analyzer` class use the object of the class as described below. To get more details for a specific API function type the following at the MATLAB command prompt:

```
help xilinx.resource_analyzer.<API_function>
```

- **Example:**

```
//Construct class. Must give the model name and absolute or relative path
//to the
//target directory

>> res_obj = xilinx.resource_analyzer('test_decimator', './
netlist_for_resource_analysis')

res_obj =

Resources used by: test_decimator
BRAMs => 0.5
DSPs => 1
Registers => 273
LUTs => 153
```

### getVivadoStage – Get Vivado design stage for resource analysis

- **Syntax:**

```
string = resource_analyzer_obj.getVivadoStage();
```

- **Description:** The returned string is the Vivado design stage after which resource analysis was performed and data collected in Vivado. The value is either Post Synthesis or Post Implementation.

- **Example:**

```
//Determine Vivado stage when resource data was collected

>> design_stage = res_obj.getVivadoStage()

design_stage =

Post Synthesis
```

### getDevicePart – Get target AMD device part name

- **Syntax:**

```
string = resource_analyzer_obj.getDevicePart();
```

- **Description:** Gets the name of the AMD device to which your design is targeted.

- **Example:**

```
//Get the AMD part in which you will implement your design  
>> part_name = res_obj.getDevicePart()  
part_name =  
xc7k325tfg676-3
```

## getDeviceResource – Get number of resources in target device

- **Syntax:**

```
total_resource_count =  
resource_analyzer_obj.getDeviceResource(<resource_type>);
```

- **Description:** The returned value is the total number of a particular type of resource contained in the AMD device for which you are targeting your design.

The `resource_type` can be:

- BRAMs - Block RAM and FIFO primitives
- DSPs - DSP48 primitives
- Registers - Registers and Flip-Flops
- LUTs - All LUT types combined

If no `resource_type` is provided, the command returns a MATLAB structure containing all device resources.

- **Example:**

```
//Determine the total number of Block RAMs in the AMD device  
>> total_brams = res_obj.getDeviceResource('BRAMs')  
total_brams =  
445  
  
//Determine the total number of Block RAMs, DSP blocks, Registers, and  
//LUTs in the  
//AMD device  
  
>> total_resource_count = res_obj.getDeviceResource  
total_resource_count =  
    BRAMs: '445'  
    DSPs: '840'  
Registers: '407600'  
    LUTs: '203800'
```

## printDeviceResources – Print number of resources in target device

- **Syntax:**

```
resource_analyzer_obj.printDeviceResources();
```

- **Description:** Prints the number of all types of resources in the used AMD device. The output is printed in the MATLAB console.

- **Example:**

```
//Print the number of all types of resources contained in the target AMD
device

>> res_obj.printDeviceResources()

BRAMs => 445
DSPs => 840
Registers => 407600
LUTs => 203800
```

## getCount – Get resource utilization for subsystem or block

- **Syntax:**

```
<block_resource_count> =
resource_analyzer_obj.getCount(<blockID>,<resource_type>);
```

- **Description:** The returned value is the total number of a particular type of resource used in the specified subsystem or block.

The `blockID` can be either a Simulink handle or a pathname (a hierarchical name) for the subsystem or block.

The `resource_type` can be:

- BRAMs - Block RAM and FIFO primitives
- DSPs - DSP48 primitives
- Registers - Registers and Flip-Flops
- LUTs - All LUT types combined

If no `resource_type` is provided, the command returns a MATLAB structure containing all device resources.

- **Example:**

```
// Return register resource utilization for Simulink block with pathname
// test_decimator/addr_gen

>> regs_in_block = res_obj.getCount('test_decimator/addr_gen',
'Registers')
```

```
ans =  
    105  
//Return resource utilization for the entire Simulink model  
  
>> total_resource_count = res_obj.getCount()  
  
Resources used by: test_decimator  
BRAMs => 0.5  
DSPs => 1  
Registers => 273  
LUTs => 153
```

## print - Prints all resources used by a subsystem or block

- **Syntax:**

```
resource_analyzer_obj.print(<blockID>);
```

- **Description:** Prints all resources (for all resource types: BRAMs, Registers, DSPs, and LUTs) used by a subsystem or block, in key-value pair. Resources are printed in the MATLAB console.

If you enter a `blockID` (which can be either a Simulink handle or a pathname), all resources used by the specified block or subsystem will be printed in the MATLAB console.

If no `blockID` argument is provided, all resources used by the top-level design will be printed in the MATLAB console.

- **Example:**

```
// Print resource utilization for Simulink subsystem with pathname  
// test_decimator/addr_gen  
  
>> res_obj.print('test_decimator/subsystem1')  
Resources used by: test_decimator/subsystem1  
BRAMs => 0.5  
DSPs => 1  
Registers => 49  
LUTs => 97  
  
//Print resource utilization for the entire Simulink model  
  
>> res_obj.print()  
Resources used by: test_decimator  
BRAMs => 0.5  
DSPs => 1  
Registers => 273  
LUTs => 153
```

**getDistribution – Get count of each resource type used by each block and subsystem under a specified subsystem**

- **Syntax:**

```
[<distribution_array>, <self_count>, <total_count>] =  
resource_analyzer_obj.getDistribution(<blockId>, <resource_type>)
```

- **Description:**

Returns count for the specified type of resource used by each block and subsystem directly under the subsystem passed as the argument.

The three returned values are:

- An array of MATLAB structures. Each element in the array is a structure containing the name of a block or subsystem directly under the subsystem in the argument, with a key-value pair of the resource type and number of resources used by that sub block or subsystem.
- A count of the resources used by the self (the block or subsystem specified in the argument).
- A count of the resources used by both blocks and subsystems combined.

The `blockID` can be either a Simulink handle or a pathname (a hierarchical name) for the subsystem or block. If no `blockID` is provided, then the command assumes the top-level module.

The `resource_type` can be:

- BRAMs - Block RAM and FIFO primitives
- DSPs - DSP48 primitives
- Registers - Registers and Flip-Flops
- LUTs - All LUT types combined

- **Example:**

```
// Return Register resource distribution for Simulink block with pathname  
// test_decimator. This is top level of the design  
  
>> [res_dist, self, total] = res_obj.getDistribution  
('test_decimator','Registers')  
  
res_dist =  
  
1x8 struct array with fields:  
  
    Name  
    Hier_Name  
    Count
```

```
self =  
    119  
  
total =  
    273  
//Return resource utilization for the entire Simulink model  
>> total_resource_count = res_obj.getCount()  
  
Resources used by: test_decimator  
BRAMs => 0.5  
DSPs => 1  
Registers => 273  
LUTs => 153
```

## getErrorMessage – Get an error message

- **Syntax:**

```
result = resource_analyzer_obj.getErrorMessage();
```

- **Description:**

Returns an error message string if the call to the class constructor or other API function had an error.

- **Example:**

```
//Determine if there was an error in the xilinx.resource_analyzer  
constructor  
//or in any of the API functions  
>> err_msg = res_obj.getErrorMessage()  
  
err_msg =  
    ''
```

## highlight – Highlight design subsystems and blocks

- **Syntax:**

```
resource_analyzer_obj.highlight(<blockId>)
```

- **Description:**

This API highlights blocks in the Simulink model. Highlighted blocks in the Model Composer model are displayed in yellow and outlined in red. Highlighting blocks using this command does not change the highlighting of other blocks currently highlighted, so more than one block can be highlighted if you use this function repeatedly.

When you enter a `blockID` (which can be either a Simulink handle or a pathname) for a block or subsystem, the specified block or subsystem will be highlighted in the Simulink model.

When the block/subsystem is highlighted then all parent subsystems up to the top level are also highlighted. When the top level module handle is provided as the `highlight` function argument no block is highlighted, but the Simulink model display changes to the top level, showing all blocks and subsystems at the top level.

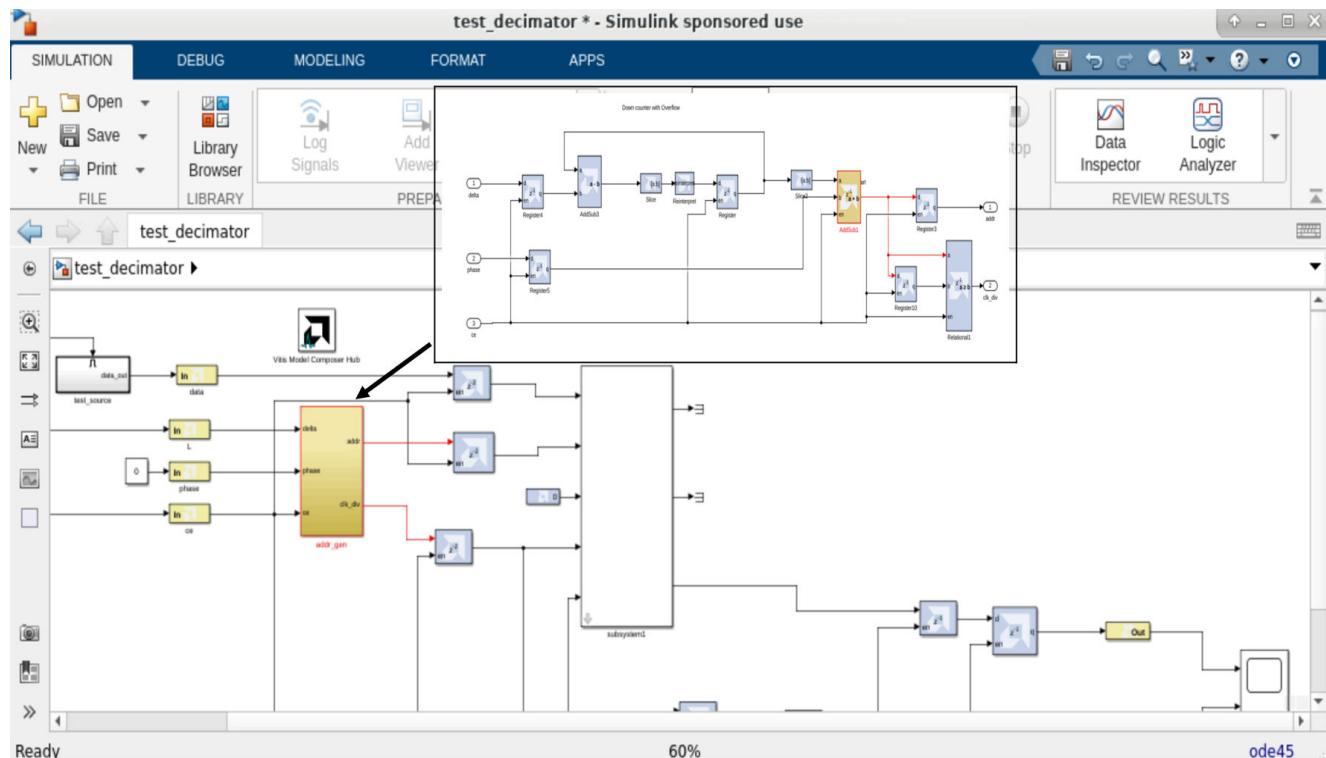
- **Example:**

```
//Highlight Simulink block with pathname fixed_point_IIR/IIR Filter
Subsystem/Mult1

>> res_obj.highlight('test_decimator/addr_gen/AddSub1')
```

Highlighted Model Composer model blocks appear as shown below.

**Figure 274: HDL Model Blocks**



### unhighlight - Unhighlight design subsystems and blocks

- **Syntax:**

```
resource_analyzer_obj.unhighlight(<blockId>)
```

- **Description:**

This API unhighlights blocks that are currently highlighted in the Simulink model. When they are unhighlighted, the blocks in the Model Composer model are displayed in their original colors.

If you enter a `blockID` (which can be either a Simulink handle or a pathname) for a block or subsystem, the specified block or subsystem will be unhighlighted in the Simulink model. When the block/subsystem is unhighlighted, all parent subsystems up to the top level are also unhighlighted.

If no `blockID` argument is provided, all currently highlighted blocks and subsystems will be unhighlighted.

- **Example:**

```
//Unhighlight Simulink block with pathname test_decimator/addr_gen/
Register4

>> res_obj.unhighlight('test_decimator/addr_gen/Register4')

//Unhighlight all Simulink blocks that are currently highlighted
>> res_obj.unhighlight();
```

## **delete - Delete xilinx.resource\_analyzer class object**

- **Syntax:**

```
resource_analyzer_obj.delete();
```

- **Description:**

This is a destructor for the `xilinx.resource_analyzer` class.

- **Example:**

```
//Delete xilinx.resource_analyzer object, i.e., res_obj
>> delete(res_obj);
```

OR

```
>> res_obj.delete();
```

## **xlfda\_denominator**

The `xlfda_denominator` function returns the denominator of the filter object stored in the AMD FDATool block.

## Syntax

```
[den] = xlfda_denominator(FDATool_name);
```

## Description

Returns the denominator of the filter object stored in the AMD FDATool block named FDATool\_name, or throws an error if the named block does not exist. The block name can be local (for example, 'FDATool'), relative (for example, '../FDATool'), or absolute (for example, 'untitled/foo/bar/FDATool').

## See Also

[xlfda\\_numerator](#)

## xlfda\_numerator

The xlfda\_numerator function returns the numerator of the filter object stored in the AMD FDATool block.

## Syntax

```
[num] = xlfda_numerator(FDATool_name);
```

## Description

Returns the numerator of the filter object stored in the AMD FDATool block named FDATool\_name, or throws an error if the named block does not exist. The block name can be local (for example, 'FDATool'), relative (for example, '../FDATool'), or absolute (for example, 'untitled/foo/bar/FDATool').

## See Also

[xlfda\\_denominator](#)

## vmcGenerate

The `vmcGenerate` function provides a programmatic way to invoke the Vitis Model Composer code generator.

## Syntax

```
status = vmcGenerate('modelName')
status = vmcGenerate('MyModel', 'MyModel_hubInfo.json')
status = vmcGenerate('MyModel', 'closeProgressWindow', 1)
[status, designInfo] = vmcGenerate('MyModel', ...)
```

## Description

`vmcGenerate` invokes the Vitis Model Composer code generator and returns a status code.

It is functionally equivalent to opening the Model Composer Hub block and clicking the Analyze, Validate, or Export buttons.

`vmcGenerate` is a blocking function that returns 0 when code generation completes successfully. A non-zero return code indicates an error during code generation.

## Example

To programmatically select the subsystem on which to operate, you can use the `vmchub_set_param` function to set the `SelectSubsystem` property. For example:

```
vmchub_set_param('modelName/Vitis Model Composer Hub', 'modelName/DUT_ss',
'SelectSubsystem', 1);
```

Then call `vmcGenerate`:

`vmcGenerate('MyModel')` generates code using the settings of the Model Composer Hub block present in 'MyModel'.

`[status, designInfo] = vmcGenerate('MyModel')` additionally gives an information struct for HDL designs. In the case of AIE and HLS this struct will be empty.

`status = vmcGenerate('MyModel', hubParams='MyModel_hubInfo.json')` uses the settings from the JSON file to generate code.

`status = vmcGenerate('MyModel', hubParams='MyModel_hubInfo.json',
targetSubsystem='MyModel/MySubsystem')` selects the subsystem for which to generate code.

`status = vmcGenerate('MyModel', closeProgressWindow=1)` closes the progress window after the code generation is done.

## See Also

[vmchub\\_get\\_param](#), [vmchub\\_set\\_param](#), [vmcGetHubParams](#), [vmcAnalyze](#), [vmcValidate](#), [vmcExport](#), Vitis Model Composer Hub block.

## vmcAnalyze

The `vmcAnalyze` function provides a programmatic way to invoke the Vitis Model Composer code generator.

### Syntax

```
status = vmcAnalyze('modelName')
status = vmcAnalyze('MyModel', 'MyModel_hubInfo.json')
status = vmcAnalyze('MyModel', 'closeProgressWindow', 1)
[status, designInfo] = vmcAnalyze('MyModel', ...)
```

### Description

`vmcAnalyze` invokes the Vitis Model Composer code generator and returns a status code.

It is functionally equivalent to opening the Model Composer Hub block and clicking the Analyze button.

`vmcAnalyze` is a blocking function that returns 0 when code generation completes successfully. A non-zero return code indicates an error during code generation.

### Example

To programmatically select the subsystem on which to operate, you can use the `vmchub_set_param` function to set the `SelectSubsystem` property. For example:

```
vmchub_set_param('modelName/Vitis Model Composer Hub', 'modelName/DUT_ss',
'SelectSubsystem', 1);
```

Then call `vmcAnalyze`:

`vmcAnalyze('MyModel')` uses the settings of the Model Composer Hub block present in 'MyModel'.

`[status, designInfo] = vmcAnalyze('MyModel')` additionally gives an information struct for HDL designs. In the case of AIE and HLS this struct will be empty.

`status = vmcAnalyze('MyModel', hubParams='MyModel_hubInfo.json')` uses the settings from the JSON file to generate code.

`status = vmcAnalyze('MyModel', hubParams='MyModel_hubInfo.json',
targetSubsystem='MyModel/MySubsystem')` selects the subsystem for which to generate code.

`status = vmcAnalyze('MyModel', closeProgressWindow=1)` closes the progress window after the code generation is done.

## See Also

[vmchub\\_get\\_param](#), [vmchub\\_set\\_param](#), [vmcGetHubParams](#), [vmcGenerate](#), [vmcValidate](#), [vmcExport](#), Vitis Model Composer Hub block.

# vmcValidate

The `vmcValidate` function provides a programmatic way to invoke the Vitis Model Composer code generator.

## Syntax

```
status = vmcValidate('modelName')
status = vmcValidate('MyModel', 'MyModel_hubInfo.json')
status = vmcValidate('MyModel', 'closeProgressWindow', 1)
[status, designInfo] = vmcValidate('MyModel', ...)
```

## Description

`vmcValidate` invokes the Vitis Model Composer code generator and returns a status code.

It is functionally equivalent to opening the Model Composer Hub block and clicking the Validate button.

`vmcValidate` is a blocking function that returns 0 when code generation completes successfully. A non-zero return code indicates an error during code generation.

## Example

To programmatically select the subsystem on which to operate, you can use the `vmchub_set_param` function to set the `SelectSubsystem` property. For example:

```
vmchub_set_param('modelName/Vitis Model Composer Hub', 'modelName/DUT_ss',
'SelectSubsystem', 1);
```

Then call `vmcValidate`:

`vmcValidate('MyModel')` uses the settings of the Model Composer Hub block present in 'MyModel'.

`[status, designInfo] = vmcValidate('MyModel')` additionally gives an information struct for HDL designs. In the case of AIE and HLS this struct will be empty.

`status = vmcValidate('MyModel', hubParams='MyModel_hubInfo.json')` uses the settings from the JSON file to generate code.

`status = vmcValidate('MyModel', hubParams='MyModel_hubInfo.json', targetSubsystem='MyModel/MySubsystem')` selects the subsystem for which to generate code.

`status = vmcValidate('MyModel', closeProgressWindow=1)` closes the progress window after the code generation is done.

## See Also

[vmchub\\_get\\_param](#), [vmchub\\_set\\_param](#), [vmcGetHubParams](#), [vmcGenerate](#), [vmcAnalyze](#), [vmcExport](#), Vitis Model Composer Hub block.

## vmcExport

The `vmcExport` function provides a programmatic way to invoke the Vitis Model Composer code generator.

### Syntax

```
status = vmcExport('modelName')
status = vmcExport('MyModel', 'MyModel_hubInfo.json')
status = vmcExport('MyModel', 'closeProgressWindow', 1)
[status, designInfo] = vmcExport('MyModel', ...)
```

### Description

`vmcExport` invokes the Vitis Model Composer code generator and returns a status code.

It is functionally equivalent to opening the Model Composer Hub block and clicking the Validate button.

`vmcExport` is a blocking function that returns 0 when code generation completes successfully. A non-zero return code indicates an error during code generation.

### Example

To programmatically select the subsystem on which to operate, you can use the `vmchub_set_param` function to set the `SelectSubsystem` property. For example:

```
vmchub_set_param('modelName/Vitis Model Composer Hub', 'modelName/DUT_ss',
'SelectSubsystem', 1);
```

Then call `vmcExport`:

`vmcExport('MyModel')` uses the settings of the Model Composer Hub block present in 'MyModel'.

[status, designInfo] = vmcExport('MyModel') additionally gives an information struct for HDL designs. In the case of AIE and HLS this struct will be empty.

status = vmcExport('MyModel', hubParams='MyModel\_hubInfo.json') uses the settings from the JSON file to export.

status = vmcExport('MyModel', hubParams='MyModel\_hubInfo.json', targetSubsystem='MyModel/MySubsystem') selects the subsystem to export.

status = vmcExport('MyModel', closeProgressWindow=1) closes the progress window after the code generation is done.

## See Also

[vmchub\\_get\\_param](#), [vmchub\\_set\\_param](#), [vmcGetHubParams](#), [vmcGenerate](#), [vmcAnalyze](#), [vmcValidate](#), Vitis Model Composer Hub block.

# vmcGetHubParams

Export Vitis Model Composer Hub block settings to a file.

## Syntax

```
vmcGetHubParams(hubBlkHdl)
vmcGetHubParams(hubBlkHdl, format='json')
```

## Description

vmcGetHubParams exports Vitis Model Composer Hub block settings to a file. The generated file contains settings for all valid subsystems in the model. The file can be used to exchange Hub block settings between models, programmatically control code generation, or track changes to Hub block settings in a source control system.

The names of parameters in the exported file are the same as the labels in the Vitis Model Composer Hub block.

The JSON file schema can be found in the hub\_schema-1.0.json file, located in the simulink folder of the Vitis Model Composer installation directory.

## Input Arguments

- **hubBlkHdl:** Path or handle to the Hub block in the model.
- **format:** Format of the generated output file. Currently 'json' is the only supported format.

## vmchub\_get\_param

Get the value of a parameter in the Vitis Model Composer Hub block.

### Syntax

```
paramValue = vmchub_get_param(hubBlock, design_or_subsystem, parameter)
```

### Description

`vmchub_get_param` returns the name or value of the specified parameter for the specified design or subsystem in the Vitis Model Composer Hub block.

`vmchub_get_param` throws an error if:

- any of the inputs are not valid, or
- if the specified design or subsystem is not selected in the Vitis Model Composer Hub block, or
- if the Vitis Model Composer Hub block configuration has not been saved at least once.

### Example

```
>> vmchub_get_param('one_krnl_one_in_one_out_32bit_2la/Vitis Model Composer Hub', 'one_krnl_one_in_one_out_32bit_2la/Subsystem1/aie_ss', 'CreateTestbench')
ans =
logical
1
```

### Input Arguments

- **hubBlock:**  
Hub block path in the model
- **design\_or\_subsystem:** Path of subsystem or design that has been selected in the hub block
- **parameter:**  
Parameter of hub block

### Output Arguments

- **paramValue:** The value of the specified parameter for the specified design or subsystem specified in the hub block.

## vmchub\_set\_param

Set the value of a parameter in the Vitis Model Composer Hub block.

## Syntax

```
vmchub_set_param(hubBlock, design_or_subsystem, parameter, paramValue)
```

## Description

`vmchub_set_param` sets the `paramValue` of the specified parameter for the specified design or subsystem in the Vitis Model Composer Hub block.

`vmchub_set_param` throws an error if:

- any of the inputs are not valid, or
- if the specified design or subsystem is not selected in the Vitis Model Composer Hub block, or
- if the Vitis Model Composer Hub block configuration has not been saved at least once.

## Example

```
>> vmchub_set_param('one_krnl_one_in_one_out_32bit_21a/Vitis Model Composer Hub', 'one_krnl_one_in_one_out_32bit_21a/Subsystem1/aie_ss', 'CreateTestbench', 1)
```

## Input Arguments

- **hubBlock:** Hub block path in the model
- **design\_or\_subsystem:** Path of subsystem or design that has been selected in the hub block
- **parameter:** Parameter of hub block
- **paramValue:** The value to be set to the specified parameter for the specified design or subsystem specified in the hub block.

# Vitis Model Composer Hub Block Parameters

Table 46: Hardware Selection Tab

Field in the App	Parameter Name	Allowed Values
Select Hardware	SelectHardware	Part, or board name or platform file

Table 47: Subsystem Selection

Field in the App	Parameter Name	Allowed Values	Notes
To select the subsystem in the left side panel	SelectSubsystem	0 or 1	Read-only parameter. Returns list of subsystems that are selected for the code generation
	TargetSubsystem		

**Table 48: Settings Tab**

<b>Field in the App</b>	<b>Parameter Name</b>	<b>Allowed Values</b>	<b>Notes</b>
Hardware Description	HardwareDescription	<ul style="list-style-type: none"> <li>Verilog</li> <li>VHDL</li> </ul>	
	VHDLLibrary	String or Char array	
	UseSTDLogic	0 or 1	
Synthesis Strategy	SynthesisStrategy	String or Char array (should be a valid synthesis strategy).	
Implementation Strategy	ImplementationStrategy	String or Char array (should be a valid implementation strategy).	
Enable Multiple clocks	EnableMultipleClocks	0 or 1	
FPGA Clock Period	FPGAClockPeriod	String or Char array (Ex: '10')	
Simulink System Period	SimulinkSystemPeriod	String or Char array (Ex: '1')	
Clock pin location	ClockPinLocation	String or Char array	
Provide Clock enable clear pin	ProvideClockEnableClearPin	0 or 1	
	ClockSettings	Struct with the fields: <ul style="list-style-type: none"> <li>subsystem</li> <li>fpgaClockPeriod</li> <li>simulinkSystemPeriod</li> <li>clockPinLocation</li> <li>provideClockEnableClearPin</li> </ul>	
FPGA Clock Frequency	FPGAClockFrequency	String or Char array (Ex: '200')	
Throughput factor	ThroughputFactor	0 to 9	
Testbench stack size	TestbenchStackSize	String or Char array (Ex: '10')	
AIE Compiler Options	AIECompilerOptions	Cell array of character vectors or an empty cell array.	

**Table 49: Analyze Tab**

<b>Field in the App</b>	<b>Parameter Name</b>	<b>Allowed Values</b>	
Block Icon Display	BlockIconDisplay	<ul style="list-style-type: none"> <li>Default</li> <li>Normalized sample periods</li> <li>Sample frequencies (MHz)</li> <li>Pipeline stages</li> <li>HDL port names</li> <li>Input data types</li> <li>Output data types</li> </ul>	

**Table 49: Analyze Tab (cont'd)**

<b>Field in the App</b>	<b>Parameter Name</b>	<b>Allowed Values</b>	
Perform Analysis	PerformAnalysis	<ul style="list-style-type: none"> <li>• None</li> <li>• Post Synthesis</li> <li>• Post Implementation</li> </ul>	
Analysis Type	AnalyzerType	<ul style="list-style-type: none"> <li>• Timing</li> <li>• Resource</li> </ul>	
Remote IP Cache	RemoteIPCache	0 or 1	
Create Interface document	CreateInterfaceDocument	0 or 1	
AIE Simulator Options	AIESimulatorOptions	Cell array of character vectors or an empty cell array.	
Simulation Timeout	SimulationTimeout	String or Char array	
Plot AIE simulation output and Estimate throughput	PlotAIESimulation	0 or 1	
Collect Profiling Statistics	CollectProfilingStats	0 or 1	
Collect Data for Vitis Analyzer	CollectDataForVitisAnalyzer	0 or 1	

**Table 50: Validate on Hardware Tab**

<b>Field in the App</b>	<b>Parameter Name</b>	<b>Allowed Values</b>
Generate Hardware Validation Code	GenerateHwValidationCode	0 or 1
Generate Hardware Image	GenerateHwImage	0 or 1
Target Directory	ValidateOnHardwareTargetDirectory	Directory
HW System Type	HwSystemType	<ul style="list-style-type: none"> <li>• BareMetal</li> <li>• Linux</li> </ul>
Target	HwTarget	<ul style="list-style-type: none"> <li>• hw</li> <li>• hw_emu</li> </ul>
Common SW Dir	HWCommonSWDir	Directory
Target SDK Dir	TargetSDKDir	Directory

**Table 51: Export Tab**

<b>Field in the App</b>	<b>Parameter Name</b>	<b>Allowed Values</b>	
Export Directory	ExportDirectory	Directory	
Export Type	ExportType	String or Char array	
Create Testbench	CreateTestbench	0 or 1	

**Table 51: Export Tab (cont'd)**

Field in the App	Parameter Name	Allowed Values	
Vendor	IPVendor	String or Char array	IP catalog Settings
Library	IPLibrary	String or Char array	
Name	IPName	String or Char array	
Version	IPVersion	String or Char array	
Category	IPCategory	String or Char array	
Status	IPStatus	<ul style="list-style-type: none"> <li>• 1 for Production</li> <li>• 2 for Beta</li> <li>• 3 for Pre-production</li> </ul>	
Auto Infer Interface	IPAutoInferInterface	0 or 1	
Use Common repository directory (Checkbox)	IPUseCommonRepoDir	0 or 1	
Edit field beside Use Common repository directory checkbox	IPCommonRepoDir	String or Char array	
Use plug-in project	IPUsePlugInProject	0 or 1	
Hardware description language (for HLS IP)	IPHwDescLang	<ul style="list-style-type: none"> <li>• Verilog</li> <li>• VHDL</li> </ul>	
Burst Mode	HwCosimBurstMode	0 or 1	Hardware Co-simulation Settings
FIFO	HwCosimFifoDepth	<ul style="list-style-type: none"> <li>• 1024</li> <li>• 2048</li> <li>• 4096</li> <li>• 8192</li> </ul>	

**Table 52: Design Settings Tab**

Field in the App	Parameter Name	Allowed Values
Treat this model as a legacy System Generator design for backward-compatibility	TreatDesignAsLegacyHDL	0 or 1
Number of parallel AI Engine builds	NumOfAIEParallelBuilds	range (1, number of cores in the system)

## xlGetReOrderedCoeff

The xlGetReOrderedCoeff function provides the re-ordered coefficient set of a FIR Compiler block.

### Syntax

```
A = xlGetReOrderedCoeff(new_coeff_set, returnType, handle)
```

## Description

**Note:** All three parameters of this function are required.

- **new\_coeff\_set:** The new coefficient set that needs to be loaded into an existing FIR Compiler. Must be supplied to the function in the original order.
- **handle:** Provides the scheme by which the coefficients are re-ordered. This can either be a handle to the FIR Compiler block in the design, or the path to a reload.txt file generated by the IP repository. If a FIR Compiler block is selected on the Simulink canvas, then the handle can be specified as gcbh.
- **returnType:** Specifies whether to return the re-ordered coefficients or the reload order only. This value can be specified as either `coeff` or `index`. A `coeff` return type modifies the required coefficient set and provide the re-arranged coefficient set that can be directly supplied to the FIR compiler block. The `index` return type provides only the coefficient address vector using the `new_coeff_set` that needs to be processed manually.

## Examples

Example 1:

If A is a row vector of coefficients, then the coefficients sorted in the reload order can be obtained as follows:

```
reload_order_coefficients = xlGetReOrderedCoeff(A, 'coeff', gcbh)
```

In this example, `reload_order_coefficients` specifies the order in which coefficients contained in A should be passed to the FIR Compiler through the reload channel.

Alternatively, the reload address vector can be obtained:

```
reload_order_coefficients = A(xlGetReOrderedCoeff(A, 'index', gcbh))
```

Example 2:

This example shows how to use an text file, generated by the IP repository, to specify the reload order.

```
reload_order_coefficients =
xlGetReOrderedCoeff(A, 'coeff', 'reload_<version>.txt')
```

## See Also

[FIR Compiler 7.2 block](#)

## xlOpenWaveFormData

Allows you to populate saved simulation waveform data into a running Waveform Viewer instance.

### Syntax

```
xlOpenWaveFormData('C:/wavedata/model_name.wdb')
```

### How to Use

1. Make sure an instance of Waveform Viewer is opened in the current Model Composer session.
2. Locate the waveform data file (`model_name.wdb`) you would like to open.

**Note:** Waveform data are saved under the `wavedata` directory.

3. Type `xlOpenWaveFormData ('C:/wavedata/model_name.wdb')` in the MATLAB console. Ensure you enter the absolute path of the waveform data file.
4. Observe the waveform data in Waveform Viewer.

### See Also

For information on using the Waveform Viewer to develop and troubleshoot your design, see *Vivado Design Suite User Guide: Logic Simulation* ([UG900](#)).

# Additional Resources and Legal Notices

## Finding Additional Documentation

### Technical Information Portal

The AMD Technical Information Portal is an online tool that provides robust search and navigation for documentation using your web browser. To access the Technical Information Portal, go to <https://docs.amd.com>.

### Documentation Navigator

Documentation Navigator (DocNav) is an installed tool that provides access to AMD Adaptive Computing documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

- From the AMD Vivado™ IDE, select **Help → Documentation and Tutorials**.
- On Windows, click the **Start** button and select **Xilinx Design Tools → DocNav**.
- At the Linux command prompt, enter `docnav`.

**Note:** For more information on DocNav, refer to the *Documentation Navigator User Guide* ([UG968](#)).

### Design Hubs

AMD Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In DocNav, click the **Design Hubs View** tab.
- Go to the [Design Hubs](#) web page.

---

# Support Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see [Support](#).

---

# References

These documents provide supplemental material useful with this guide:

1. *Introduction to FPGA Design with Vivado High-Level Synthesis* ([UG998](#))
2. *Vivado Design Suite User Guide: Logic Simulation* ([UG900](#))
3. *Vivado Design Suite User Guide: High-Level Synthesis* ([UG902](#))
4. *AXI4-Stream Video IP and System Design Guide* ([UG934](#))
5. *Vivado Design Suite User Guide: Designing with IP* ([UG896](#))
6. *Vivado Design Suite: AXI Reference Guide* ([UG1037](#))
7. *UltraFast Vivado HLS Methodology Guide* ([UG1197](#))
8. *Vitis Unified Software Platform Documentation: Application Acceleration Development* ([UG1393](#))
9. *Vitis Unified Software Platform Documentation: Embedded Software Development* ([UG1400](#))
10. *Vitis Unified Software Platform Documentation Landing Page* ([UG1416](#))
11. *Vitis Reference Guide* ([UG1702](#))
12. *Vivado Design Suite User Guide: Creating and Packaging Custom IP* ([UG1118](#))
13. *Vitis High-Level Synthesis User Guide* ([UG1399](#))
14. *AI Engine Tools and Flows User Guide* ([UG1076](#))
15. *KC705 Evaluation Board for the Kintex 7 FPGA* ([UG810](#))
16. *Vivado Design Suite User Guide: Using the Vivado IDE* ([UG893](#))
17. *Vivado Design Suite User Guide: Design Flows Overview* ([UG892](#))
18. *ISE to Vivado Design Suite Migration Guide* ([UG911](#))
19. *Vivado Design Suite User Guide: Using Constraints* ([UG903](#))
20. *Vivado Design Suite User Guide: Using Tcl Scripting* ([UG894](#))
21. *Vivado Design Suite Tutorial: Design Flows Overview* ([UG888](#))
22. *Vivado Design Suite User Guide: System-Level Design Entry* ([UG895](#))
23. *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing* ([UG973](#))

24. UltraFast Design Methodology Guide for FPGAs and SoCs ([UG949](#))
  25. AMD Vivado™ Design Suite Documentation
  26. Mathworks® Simulink® Documentation
  27. Vitis Model Composer Design Hub ([DH218](#))
  28. [Vitis Model Composer Tutorials](#)
  29. Versal Adaptive SoC AI Engine Architecture Manual ([AM009](#))
  30. Versal Adaptive SoC AIE-ML Architecture Manual ([AM020](#))
  31. Reduce Power and Cost by Converting from Floating Point to Fixed Point ([WP491](#))
  32. Versal Architecture and Product Data Sheet: Overview ([DS950](#))
- 

## Revision History

The following table shows the revision history for this document.

Section	Revision Summary
<b>05/29/2025 Version 2025.1</b>	
General updates	Updated for 2025.1 release.
<a href="#">Supported MATLAB Versions and Operating Systems</a>	Section updated
<a href="#">Installation</a>	Section updated
<b>02/03/2025 Version 2024.2</b>	
<a href="#">Setting up the Tool to Generate an Image File for Hardware Validation Flow</a>	Respin to add section
General updates	Removed sections: <ul style="list-style-type: none"><li>• Model Composer API for Programmatic Generation</li><li>• PG API Examples</li><li>• G API Error/Warning Handling and Messages</li></ul> Moved <a href="#">M-Code Access to Hardware Co-Simulation</a> .
<b>11/13/2024 Version 2024.2</b>	
<a href="#">Supported MATLAB Versions and Operating Systems</a>	Updated Operating System information.
<a href="#">Importing AI Engine Kernels</a>	Added Cascade Stream-Based Data Types.
General Updates	Replaced legacy screenshots throughout.
<b>05/30/2024 Version 2024.1</b>	
<a href="#">Chapter 2: HDL Library</a>	Updated throughout for new Hub block.
<a href="#">Chapter 3: HLS Library</a>	Updated throughout for new Hub block.
<a href="#">Chapter 4: AI Engine Library</a>	Updated throughout for new Hub block.
<a href="#">Vitis Model Composer Hub Block for AI Engine Code Generation</a>	Rewrite of section.

Section	Revision Summary
General updates	<ul style="list-style-type: none"> <li>Editorial updates throughout document.</li> <li>Replaced legacy screenshots throughout.</li> </ul>
<b>11/15/2023 Version 2023.2</b>	
<a href="#">Supported MATLAB Versions and Operating Systems</a>	Updated Operating System information
<b>10/19/2023 Version 2023.2</b>	
General Updates	Updated for 2023.2 release.
<a href="#">Data Accessing Mechanisms</a>	Updated window to buffer interface
<a href="#">Importing AI Engine Kernels</a>	Updated example code to use buffers
<a href="#">Plotting AIE Simulation Internal Signals</a>	Added new section
<a href="#">Vitis Model Composer Hub</a>	Added libadbf.a/.xo targets for hardware
<b>05/10/2023 Version 2023.1</b>	
General Updates	Updated for 2023.1 release.
<a href="#">vmcGenerate</a>	Added utility
<a href="#">Example 1: Creating an Implementation Target</a>	Updated section
<a href="#">Vitis Model Composer Hub Block Parameters</a>	Updated section
<b>01/13/2023 Version 2022.2</b>	
<a href="#">Supported MATLAB Versions and Operating Systems</a>	Updated
<b>11/08/2022 Version 2022.2</b>	
<a href="#">Vitis Model Composer Hub</a>	Added HLS Hardware Flow note
<b>10/19/2022 Version 2022.2</b>	
General Updates	Updated for 2022.2 release.
<a href="#">Automatic Code Generation</a>	Updated to use Vitis Model Composer Hub block
<a href="#">Variant Subsystems and Vitis Model Composer</a>	Replaces Configurable Subsystems and Model Composer
<a href="#">Chapter 5: Hardware Validation Flow for AI Engines and PL</a>	New chapter
AI Engine Blockset	Added Window Function and Window Function Stream blocks
<b>05/26/2022 Version 2022.1</b>	
AI Engine Blockset	Added new DSPLib Functions.
HDL Blockset	Added new Gateway In/Out AXI Stream Functions.
General Updates	Updated for release 2022.1.

---

## Please Read: Important Legal Notices

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security

vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes. THIS INFORMATION IS PROVIDED "AS IS." AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

### AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

### Copyright

© Copyright 2017-2025 Advanced Micro Devices, Inc. AMD, the AMD Arrow logo, Artix, Kintex, UltraScale, Versal, Virtex, Vitis, Vivado, Zynq, and combinations thereof are trademarks of Advanced Micro Devices, Inc. AMBA, AMBA Designer, Arm, ARM1176JZ-S, CoreSight, Cortex, PrimeCell, Mali, and MPCore are trademarks of Arm Limited in the US and/or elsewhere. MATLAB and Simulink are registered trademarks of The MathWorks, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.