

Xilinx **System Generator for DSP** ***Version 9.1***

User's Guide

February 16, 2007



XILINX
SYSTEM
GENERATOR™
For DSP





Xilinx is disclosing this Document and Intellectual Property (hereinafter "the Design") to you for use in the development of designs to operate on, or interface with Xilinx FPGAs. Except as stated herein, none of the Design may be copied, reproduced, distributed, republished, downloaded, displayed, posted, or transmitted in any form or by any means including, but not limited to, electronic, mechanical, photocopying, recording, or otherwise, without the prior written consent of Xilinx. Any unauthorized use of the Design may violate copyright laws, trademark laws, the laws of privacy and publicity, and communications regulations and statutes.

Xilinx does not assume any liability arising out of the application or use of the Design; nor does Xilinx convey any license under its patents, copyrights, or any rights of others. You are responsible for obtaining any rights you may require for your use or implementation of the Design. Xilinx reserves the right to make changes, at any time, to the Design as deemed desirable in the sole discretion of Xilinx. Xilinx assumes no obligation to correct any errors contained herein or to advise you of any correction if such be made. Xilinx will not assume any liability for the accuracy or correctness of any engineering or technical support or assistance provided to you in connection with the Design.

THE DESIGN IS PROVIDED "AS IS" WITH ALL FAULTS, AND THE ENTIRE RISK AS TO ITS FUNCTION AND IMPLEMENTATION IS WITH YOU. YOU ACKNOWLEDGE AND AGREE THAT YOU HAVE NOT RELIED ON ANY ORAL OR WRITTEN INFORMATION OR ADVICE, WHETHER GIVEN BY XILINX, OR ITS AGENTS OR EMPLOYEES. XILINX MAKES NO OTHER WARRANTIES, WHETHER EXPRESS, IMPLIED, OR STATUTORY, REGARDING THE DESIGN, INCLUDING ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE, AND NONINFRINGEMENT OF THIRD-PARTY RIGHTS.

IN NO EVENT WILL XILINX BE LIABLE FOR ANY CONSEQUENTIAL, INDIRECT, EXEMPLARY, SPECIAL, OR INCIDENTAL DAMAGES, INCLUDING ANY LOST DATA AND LOST PROFITS, ARISING FROM OR RELATING TO YOUR USE OF THE DESIGN, EVEN IF YOU HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. THE TOTAL CUMULATIVE LIABILITY OF XILINX IN CONNECTION WITH YOUR USE OF THE DESIGN, WHETHER IN CONTRACT OR TORT OR OTHERWISE, WILL IN NO EVENT EXCEED THE AMOUNT OF FEES PAID BY YOU TO XILINX HEREUNDER FOR USE OF THE DESIGN. YOU ACKNOWLEDGE THAT THE FEES, IF ANY, REFLECT THE ALLOCATION OF RISK SET FORTH IN THIS AGREEMENT AND THAT XILINX WOULD NOT MAKE AVAILABLE THE DESIGN TO YOU WITHOUT THESE LIMITATIONS OF LIABILITY.

The Design is not designed or intended for use in the development of on-line control equipment in hazardous environments requiring failsafe controls, such as in the operation of nuclear facilities, aircraft navigation or communications systems, air traffic control, life support, or weapons systems ("High-Risk Applications"). Xilinx specifically disclaims any express or implied warranties of fitness for such High-Risk Applications. You represent that use of the Design in such High-Risk Applications is fully at your risk.

2000 - 2006 Xilinx, Inc. All rights reserved. XILINX, the Xilinx logo, and other designated brands included herein are trademarks of Xilinx, Inc. All other trademarks are the property of their respective owners.

Release Information

Release Notes

System Generator 9.1.0.0 Enhancements

New Technologies Supported

- Virtex™-5 SXT
- Spartan™-3A

EDK Projects with Bidirectional I/O

System Generator now supports importing processor based systems, developed in Xilinx Platform Studio™, that use peripherals with bidirectional I/Os such as System Ace. This support includes the ability to create hardware co-simulation tokens for those peripherals.

Hardware co-simulation support for the ML506 development platform

- Supports both Ethernet point to point and network-based interfaces
- Includes the Virtex™-5 SX50T device
- Supports both single step and free running clocking

Xilinx DSP Blockset Enhancements

FFT Enhancements

- Supports the LogicCore FFT 4.1 Compiler
- Support for Virtex™-5
- Addition of Radix-2-lite architecture offering lowest yet resource solution
- Removal of latching behavior for START and UNLOAD signals

Interleaver/Deinterleaver Enhancements

- Supports LogicCore Interleaver/Deinterleaver 5.0
- Support added for Virtex™-5 and Spartan™-3A

Reed-Solomon Decoder Enhancements

- Supports Reed-Solomon Decoder 6.0
- Support added for Virtex™-5 and Spartan™-3A
- Variable check symbol input added (R_IN)

Reed-Solomon Encoder Enhancements

- Supports LogicCore Reed-Solomon Encoder 6.0
- Provides full support for high-performance implementations targeting Virtex 5

Viterbi Decoder Enhancements

- Supports LogicCore Viterbi Decoder 6.0
- Support added for Virtex™-5 and Spartan™-3A, including optimized area and speed architectures

- for the LUT6 fabric
- Virtex-5 specific enhancements
- New speed option for the parallel configuration of the core (Virtex-5 only)
- General Enhancements
- BRAM utilization reduced by 50% for parallel and multi-channel configurations
- Traceback length for standard traceback can now be any value between 12 and 128; previously the length had to be divisible by 6. (Reduced latency traceback length still must be divisible by 6)
- Core now uses Standard Traceback instead of Reduced Latency Traceback for multi-channel configurations
- The BER count output is no longer scaled by a factor of 8
- Internal puncturing support removed from the core. Erasure pins offer more flexible rate management

Toolflow and Integration

System Generator 9.1 is compatible with the following tools

Tool	Version
The Mathworks MATLAB®, Simulink	2006a, 2006b
Xilinx® ISIM	9.1.02i
Xilinx® AccelDSP	9.1.00i
Xilinx® XST	9.1.02i
Xilinx® ChipScope Pro	9.1.02i
Xilinx® EDK	9.1.00i
Xilinx® CoreGen	IP Update 9.1i_IP1
Mentor Graphics ModelSim® SE	6.1f
Synplicity Synplify Pro®	8.6.2 (Requires a floating license)

Migrating designs Created in Previous Versions of Software

You must update System Generator version 7.1, or earlier, models to 9.1. A brief *Conversion Guide* is provided that explains the process in detail. To update a model you run a MATLAB command that invokes a conversion script.

Please be advised that the conversion script does not automatically save an old version of your model as it updates the design nor save a new version of your model after conversion. You can either make a back up copy of your model before running the conversion script, or you can save the updated model with a new name.

Some models may require some manual modification after running the conversion script. The script will point out any necessary manual changes.

Upgrading a Xilinx System Generator Model

Upgrading v2.x and prior models

If you are upgrading from versions of System Generator earlier than **v3.1**, you must obtain System Generator v7.x and update your models to v7.x before you can update them to v8.2.

Upgrading v3.x, v6.x and v7.x models

This document describes the process of updating a Xilinx System Generator v3.x, v6.x or v7.x model to work with v8.2. **Note that any reference to v6.x or v3.x in this document can be used interchangeably with v7.x.** System Generator v8.2 provides an automatic model conversion script called `xlUpdateModel`. This script is a MATLAB function that updates a v7.x model to v8.2.

The following are the steps for updating a v7.x model to v8.2:

1. Save a backup copy of your v7.1 model and user defined libraries that your model uses.
2. Run the `xlUpdateModel` conversion script on any libraries first and then your model.
3. Read the report produced by the conversion script and follow the instructions.
4. Check that your model runs under v8.2.

These steps are described in greater detail below.

1. Run the `xlUpdateModel` conversion script.

From the MATLAB console, `cd` into the directory containing your model. If the name of your model is "designName.mdl", type `xlUpdateModel('designName')`.

The `xlUpdateModel` conversion script performs the following:

- Updates each block in your v7.x design to a corresponding v8.2 block with equivalent settings.
- Writes a report explaining all of the changes that were made. This report enumerates changes you may need to make by hand to complete the conversion.

In most cases, `xlUpdateModel` produces an equivalent v8.2 model. However, there are a few constructs that may require you to edit your model. It is important that you read the report the script produces and follow the remaining steps in this Conversion Guide.

2. Read the report produced by `xlUpdateModel` and follow the instructions.

The script writes a report when you run it. It is important that you read this report. If the report contains the issues listed below, manual intervention will be required to complete the conversion.

Xilinx System Generator v7.x models containing removed blocks

The following blocks have been removed from System Generator:

- CIC
- Clear Quantization Error
- Digital Up Converter
- J.83 Modulator

- Quantization Error
- Sync

Xilinx System Generator v7.x models that contain deprecated blocks

The following blocks still exist in System Generator, but have been deprecated:

- DDSv4.0

Xilinx System Generator v7.x models utilizing explicit sample periods

The explicit sample period fields have been removed from most non-source blocks in System Generator v8.2. Source blocks (e.g., Counter block) continue to allow the specification of explicit sample periods. When updating models containing feedback loops, Assert blocks must typically be added by hand after `xlUpdateModel` has been run. This is necessary in order to help System Generator determine appropriate rates and types for the path. The following error message is an indication that an Assert block is required:

**“The data rates could not be established for the feedback paths through this block.
You may need to add Assert blocks to instruct the system”**

In such a case, you should augment each feedback loop with an Assert block, and specify rates and types explicitly on this block.

The update script will annotate the converted model wherever the v7.1 model asserted an explicit period. In the converted model, you will most often not need to insert Assert blocks. To find out where you need them, try to update the diagram (the Update Diagram control is under the Edit menu). If rates do not resolve, you will need to insert one or more Assert blocks.

The update script can be configured to automatically insert Assert blocks immediately following blocks configured with an explicit sample period setting. To use this option, run:

```
xlUpdateModel(mdlname, 'assert')
```

3. Verify that your model runs under System Generator v8.2.

If you have followed the instructions in Steps 1 and 2, your model should run with System Generator v8.2. Open the model with System Generator v8.2 and run it.

Installing System Generator

Downloading

System Generator is only available via download from the Xilinx web page¹. You may purchase, register, and download the System Generator software from the site at:

http://www.xilinx.com/products/software/sysgen/sg_intro.htm

Hardware Co-Simulation Support

If you have an FPGA development board, you may be able to take advantage of System Generator's ability to use FPGA hardware in the loop with Simulink simulations. The System Generator software includes support for the XtremeDSP Development Kit, the MicroBlaze Multimedia Demonstration boards, the MVI hardware platform, the ML402 Virtex-4 platform, and the Spartan-3 development platform. Additional System Generator board support packages provide support for additional hardware co-simulation platforms. System Generator board support packages can be downloaded from the following URL:

http://www.xilinx.com/technology/dsp/thirdparty_devboards.htm#hardware

¹ In special circumstances, System Generator can be delivered on a CD. Please contact your Xilinx distributor if your circumstances prohibit you from downloading the software via the web.

Installing

Software Prerequisites

You must have the following software installed before installing System Generator.

- One of the following versions of MATLAB from The MathWorks Inc.:
 - MATLAB v7.1.0/Simulink v6.3 (R14.3) Service Pack 3
 - MATLAB v7.2/Simulink v6.4 (R2006a)

Note: MATLAB must be installed in a directory with no spaces (e.g., C:\MATLAB72).

- Xilinx ISE Foundation version v8.2.01 (8.2 Service Pack 1) or later, along with the full version of the ISE Simulator.
- Xilinx ISE 8.2 CORE Generator IP Update 1 (8.2i_IP1)

Some features in System Generator require the following software to be installed:

- A logic synthesis compiler. System Generator is fully compatible with Xilinx XST (included in the ISE Foundation bundle), and Synplify Pro v8.5 or v8.6.1 from Synplicity, Inc.
- A hardware description language (HDL) simulator is required only for co-simulating HDL modules within Simulink using System Generator. System Generator HDL co-simulation interfaces are compatible with Xilinx ISE Simulator, ModelSim Xilinx Edition MXE (an option with ISE Foundation), and ModelSim PE or SE, version v6.1b, from Model Technology Inc.
- The Xilinx Embedded Development Kit version 8.2 is required if the EDK Processor block is used.
- Xilinx ChipScope Pro 8.2i Service Pack 1 Release is required for the ChipScope block.

Note: The Microsoft Windows environment variable \$XILINX must be set and point to your ISE software installation directory.

ISE software service packs may be downloaded from the 8.2i Software Updates Center:

http://www.xilinx.com/xlnx/xil_sw_updates_home.jsp

Using the System Generator Installer

Download SysgenInstall_v8_2.exe (or SysgenEval_v8_2.exe) installer from the Xilinx web site. Before executing the installer, please make sure and close all instances of MATLAB. Please follow the instructions on the screen from this point.

Hardware Co-Simulation Installation

This section describes the hardware and software installation procedures for hardware co-simulation. If you do not plan to use hardware co-simulation, you may skip this section.

Installing Hardware

Before installing hardware co-simulation software, you should ensure the FPGA development board is installed correctly by following the vendor's installation instructions. If you have a board that supports JTAG hardware co-simulation, there are additional hardware installation requirements:

- The power supply and all other required cables must be attached correctly to your development board.
- A Parallel Cable IV or Platform Cable USB programmable cable must be connected to your machine. If you are using a Parallel Cable IV, the cable must connect to the ECP printer port (LPT1 only) on your machine. A Platform Cable USB connects to a standard USB port on your machine.
- The programming cable header must be connected to the JTAG port on your FPGA development board using either the flying leads or ribbon cable connectors.

The following links provide support and purchasing information for the Parallel Cable IV and Platform Cable USB:

- [Parallel Cable IV](#)
- [Platform Cable USB](#)

Installing Software

The System Generator software includes hardware co-simulation support for the XtremeDSP Development Kit, the ML402 development kit, and MicroBlaze Multimedia Demonstration board. You can add support for new FPGA development boards by installing additional System Generator plugins. These plugins are distributed as zip files that configure System Generator to support your board.

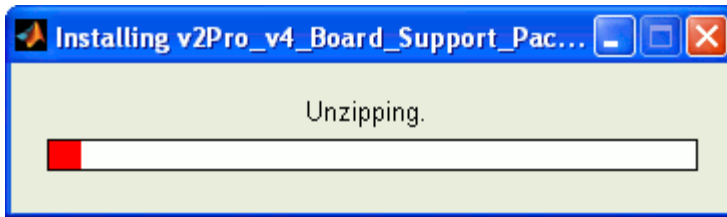
System Generator provides an installer to install your development board plugins. A System Generator plugin can be installed by following the steps provided below:

1. Download the plugin to a temporary directory.
2. From the MATLAB command window, change directories to the temporary directory where the plugin is saved.
3. From the MATLAB command window, type

```
>> xlInstallPlugin('myplugin.zip')
```

Note: `myplugin.zip` is the name of the plugin file you are installing.

This installs the plugin. A status bar is displayed to show the progress of the installer.



Once the plugin is installed, you can choose the development board as a compilation target in the System Generator block dialog box.

Compiling Xilinx HDL Libraries

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

ModelSim (PE or EE/SE)

The Xilinx tool that compiles libraries for use in ModelSim PE or EE/SE is named `compxlib`. The following command can, for example, be used to compile all the VHDL and Verilog libraries with ModelSim SE:

```
compxlib -s mti_se -f all -l all
```

Complete instructions for running `compxlib` can be found on section *Simulating Your Design*, subsection *Compiling Xilinx Simulation Libraries* on the ISE Software Manual, which is available online at:

<http://toolbox.xilinx.com/docsan/xilinx82/books/docs/sim/sim.pdf>

MXE Libraries

If you plan to use ModelSim XE (Xilinx Edition), download the MXE precompiled libraries from the Xilinx web site. The latest libraries are located at

http://www.xilinx.com/xlnx/xil_sw_updates_home.jsp?update=mx_libs

Unzip these MXE libraries into the directory in which you have MXE installed, e.g., `c:/Modeltech_XE/`. This is the location where MXE expects to find your Xilinx compiled libraries, so it is not necessary to change your `modelsim.ini` file. This file should point to the correct installed location.

Examples and Demos

System Generator Examples

Numerous examples are used to illustrate System Generator features and functions in the System Generator User's Guide. These examples are found in the <path_to_sysgen>/examples directory and are listed in the table below. In addition to these examples, System Generator also includes demonstration models which can be run from the demo page.

If you are using the Matlab help browser you can open and run the examples directly from this page. To run an example, click on the link. Matlab will change directories to the example directory and open the example model.

Black Box Examples

Topic	Description
Importing a VHDL Module (Open text)	A tutorial showing how to use the black box to import VHDL into a System Generator design and how to use Modelsim to co-simulate the VHDL module. (Open this model)
Simulating Several Black Boxes Simultaneously (Open text)	Shows how black boxes can co-simulate simultaneously, using only one ModelSim license. (Open this model)
Dynamic Black Boxes (Open text)	A tutorial showing how to parameterize the black box. (Open this model)
Importing a Verilog Module (Open text)	A tutorial showing how to use the black box to import Verilog into a System Generator design and how to use ModelSim to co-simulate the Verilog module. (Open this model)
Importing a Xilinx COREGEN Module (Open text)	A tutorial showing how to import a COREGEN module as a black box. (Open this model)

ChipScope Examples

Topic	Description
Using ChipScope Pro Analyzer for Real-time Hardware Debugging (Open text)	This tutorial demonstrates how to connect and use the Xilinx Debug Tool called ChipScope Pro within Xilinx System Generator. The integration of ChipScope Pro in the System Generator flow allows real-time debugging at system speed. (Open this model)

DSP Examples

Topic	Description
DSP48 (Open text)	Simple example demonstrating the use of the DSP48 block with the Constant block used to provide the DSP48 instruction. (Open this model)
DSP48 Macro (Open text)	Simple example demonstrating how to use a DSP48 Macro block to implement a Complex Multiplier. (Open this model)
35-Bit Multiplier using DSP48 and Constant block (Open text)	This design demonstrates the use of the DSP48 and Constant block in implementing 35 by 35-bit multipliers at different sample rates. Three multipliers implementations are shown at 1, 2, and 4 clocks per sample. (Open this model)
35-bit Multiplier using DSP48 Macro block (Open text)	This design demonstrates the use of the DSP48 Macro block in implementing a 35 by 35 Multiplier. (Open this model)
FIR filter examples using DSP48 block (Open text)	This design demonstrates the use of the DSP48 and Constant block in FIR filter implementation. The design includes sets of parallel, semi-parallel and sequential FIR filter using Type 1 and Type 2 architectures. Each filter implements a 16-tap dsp48 based FIR filters. (Open this model)
DSP48 based dynamic shifter (Open text)	This design demonstrates the use of the DSP48 block in implementing a 35-bit signed right shift using 2 DSP48s. (Open this model)
Synthesizable FIR filter for Virtex4 (Open text)	This design demonstrates how to use System Generator to implement a synthesizable FIR filter which maps efficiently to the Virtex4 architecture. (Open this model)
FIR filter using the DSP48 Macro block as a multiply accumulate function (Open text)	This design demonstrates the use of the DSP48 Macro block when implementing a sequential FIR filter. (Open this model)
MAC FIR filter	This design example implements a 43 tap FIR Filter with a MAC engine and a Dual Port Ram used for data and coefficient storage. (Open this model)
Complex FIR filter	This example demonstrates a complex FIR filter built out of blocks from the System Generator and Simulink library. (Open this model)

MCode Examples

Topic	Description
Simple Selector (Open text)	This example shows how to implement a function that returns the maximum value of its inputs. (Open this model)
Simple Arithmetic Operations (Open text)	This example shows how to implement simple arithmetic operations. (Open this model)
Complex Multiplier with Latency (Open text)	This example shows how to build a complex multiplier with latency. (Open this model)
Shift Operations (Open text)	This example shows how to implement shift operations. (Open this model)
Passing Parameters	This example shows how to pass parameters into a MCode block.

into the MCode Block (Open text)	(Open this model)
Optional Input Ports (Open text)	This example shows how to implement optional input ports on an MCode block. (Open this model)
Finite State Machines (Open text)	This example shows how to implement a finite state machine. (Open this model)
Parameterizable Accumulator (Open text)	This example shows how to build a parameterizable accumulator. (Open this model)
FIR Blocks and Verification (Open text)	This example shows how to model FIR blocks and how to do system verification. (Open this model)
RPN Calculator (Open text)	This example shows how to model a RPN calculator – a stack machine. (Open this model)
Example of disp function (Open text)	This example shows how to use the disp function. (Open this model)

Processor Examples

Topic	Description
Designing and Exporting MicroBlaze Processor Peripherals (Open text)	Demonstrates how to how to design a peripheral (pcore) in System Generator for a MicroBlaze processor, and how to export it into Xilinx Platform Studio (XPS). An RGB to gray-scale color space converter is designed and exported as a pcore using the Export to EDK compilation target. (Open this model)
Designing and Simulating MicroBlaze Processor Systems (Open text)	Demonstrates how to design and import a MicroBlaze processor system created using Xilinx Platform Studio into System Generator. A DSP48 block is used as a co-processor to MicroBlaze. (Open this model)
Software Based Simulation Modes (Open text)	Demonstrate the two software based simulation modes offered by the EDK Processor block by showing the design and simulation of an LMS filtering application. (Open this model)
Designing PicoBlaze Microcontroller Applications (Open text)	Demonstrates how to implement a PicoBlaze program in System Generator. The example programs the PicoBlaze to alter the output frequency of a Direct Digital Synthesizer (DDS) during an interrupt. (Open this model)

Shared Memory Examples

Topic	Description
Simulation across various models (Open text)	Illustrates shared memories communicating across Simulink models. (Open this model)
Host PC Shared Memory access (Open text)	Developer studio project to communicate with a shared memory. (Open this model)
High Speed Video Processing using	Discussion of a high-speed co-simulation buffering interface followed by an example in which the interface is used to support

Hardware Co-simulation (Open text)	real-time processing of a video stream using a 5x5 filter kernel. (Open this model)
High speed I/O Buffering (Open text)	Illustrates high speed Shared Memory I/O Buffering Interface for Hardware Co-simulation. (Open this model)
Mex-function interface (Open text)	Illustrates the use of a mex-function as an interface to a shared memory. (Open this model)
Multiple Cycle-True Islands for Distinct Clocks (Open text)	An example using two asynchronous clocks. (Open this model)
Shared Memory, To FIFO, To Register, To Register, From Register	Demonstrates use of shared memories, FIFOs and registers to pass information. (Open this model)
Frame-based Acceleration using Hardware Co-simulation (Open text)	Explains how to use frame or vector-based transfers to further accelerate simulations using FPGA hardware in the loop. (Open this model)

Timing Analysis Examples

Topic	Description
Using the System Generator Timing Analysis Tool to Meet Timing Requirements (Open text)	Explains how to use the System Generator Timing Analysis tool to meet timing requirements of System Generator designs. Also touches on techniques that may be used when a design does not meet timing. (Open this model)

Miscellaneous Examples

Topic	Description
Importing a System Generator Design into a Bigger System (Open text)	Discusses how to take the VHDL netlist from a System Generator design and synthesize it in order to embed it into a larger design. Also shows how VHDL created by System Generator can be incorporated into simulation model of the overall system. (Open this model)
Configurable Subsystems and System Generator (Open text)	Illustrates the use of Configurable Subsystems for Simulation and Generation. (Open this model)
Integrator	This example uses an integrator to illustrate error analysis capability. (Open this model)
Block RAM Based State Machines	Demonstrates use of Mealy State Machine block from the reference library. (Open this model)

System Generator Demos

The Xilinx System Generator for DSP provides the capability to model and implement high-performance DSP systems in field-programmable gate arrays (FPGAs) using Simulink. The Xilinx Blockset contains bit and cycle-true models of arithmetic and logic functions, memories, and DSP functions for digital filtering, spectral analysis, and digital communications. System Generator converts a Simulink model of Xilinx blocks into an efficient hardware implementation that combines synthesizable VHDL and intellectual property blocks that have been hand-crafted to run efficiently in FPGAs.

Included with the tool are numerous demonstration designs that highlight key features and tool capabilities, as well as general good design practices using real-world design applications. These designs may be accessed from the System Generator demo page.

A simple digital filter implements the following difference equation:

where a is a parameter. The System Generator model for this single-input, single-output system follows directly from this specification, as shown in Figure 1. (Click to [Open this model](#))



18

input and output of the filter. It can be readily seen that the filter largely eliminates the high frequency component of the input signal.

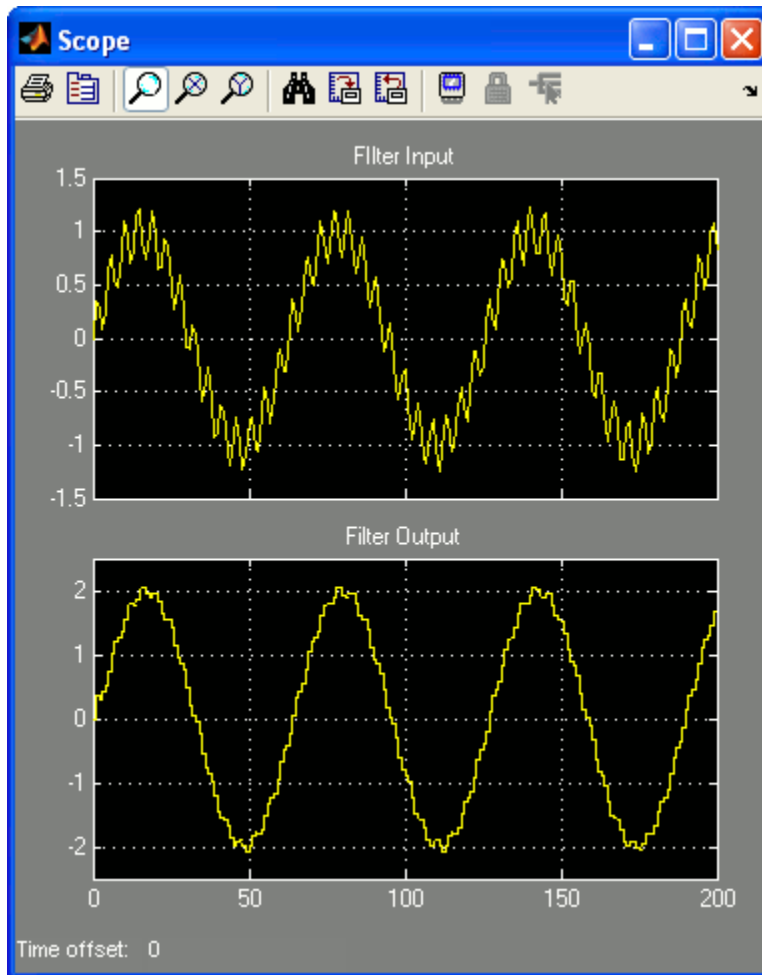


Figure 2: Fir Filter Scope

To construct this model, start MATLAB and open the Simulink library browser by typing `simulink` in the command window.

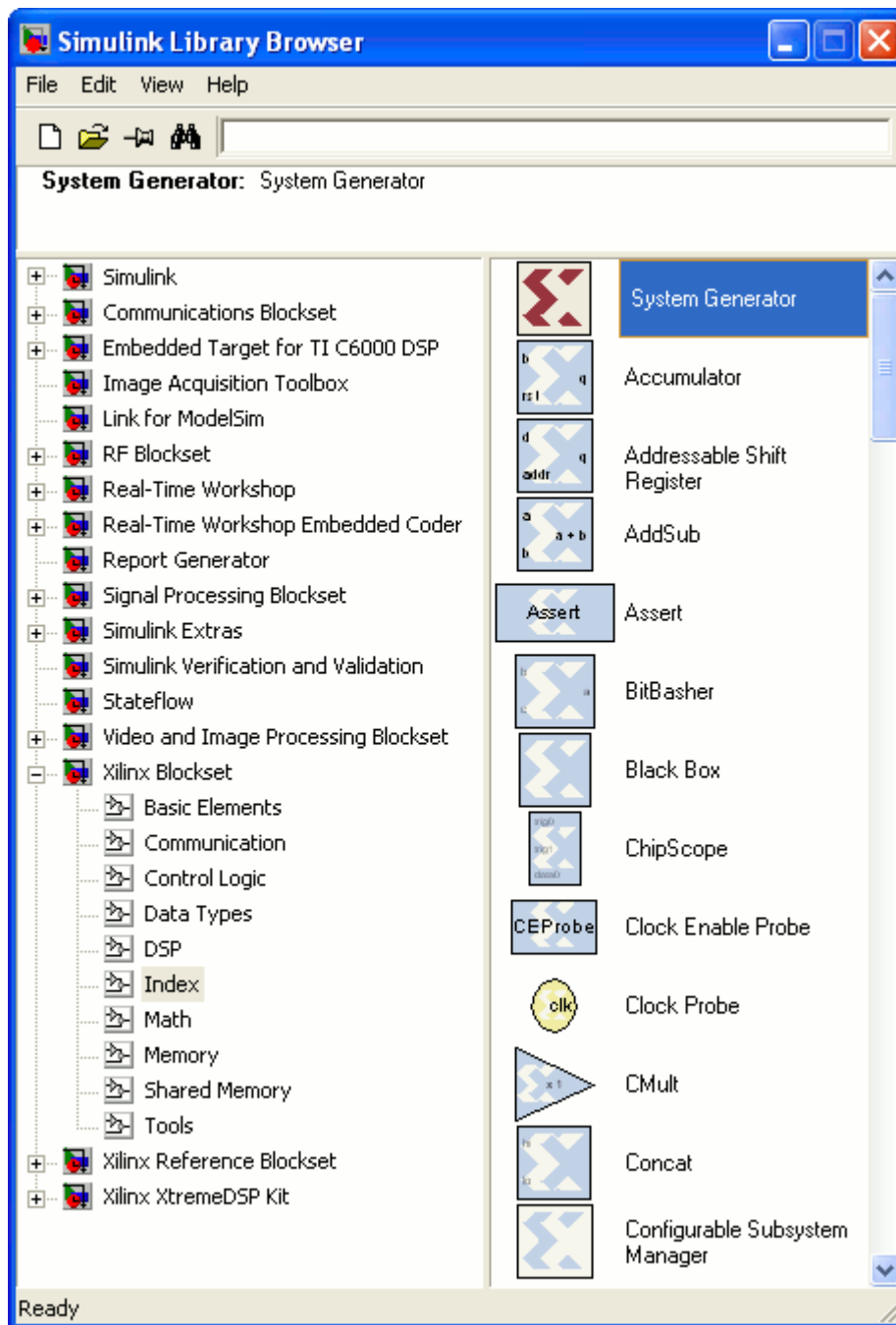


Figure 3: Simulink Library Browser

The library taxonomy tree shows installed libraries, and may differ from that shown in Figure 3. Open a new design by clicking on the “**blank page**” icon near the upper left corner of the library browser, or by selecting **File->New->Model** in the pull-down menu. Construct the model by dragging block icons from the library browser into the block diagram, and by connecting signals between ports by mouse dragging between the desired source and sink ports.

To generate a hardware description of the digital filter, double-click on the System Generator block to open its dialog box (Figure 4). This graphical interface is used to control simulation and code generation. Selecting **HDL Netlist** as the compilation target instructs System Generator to generate a hardware description of the model (in this case, VHDL language) for the specified FPGA (in this case,

a Virtex-II xc2v1000 device), into the directory specified as the **Target Directory**. Assuming you haven't made any errors in constructing your model, clicking the **Generate** button invokes the code generator and creates the hardware description.

During code generation, any non-Xilinx blocks are discarded. If the Simulink model is called `simple_fir.mdl`, then the corresponding top level VHDL entity is in a file named `simple_fir.vhd` in the target directory.

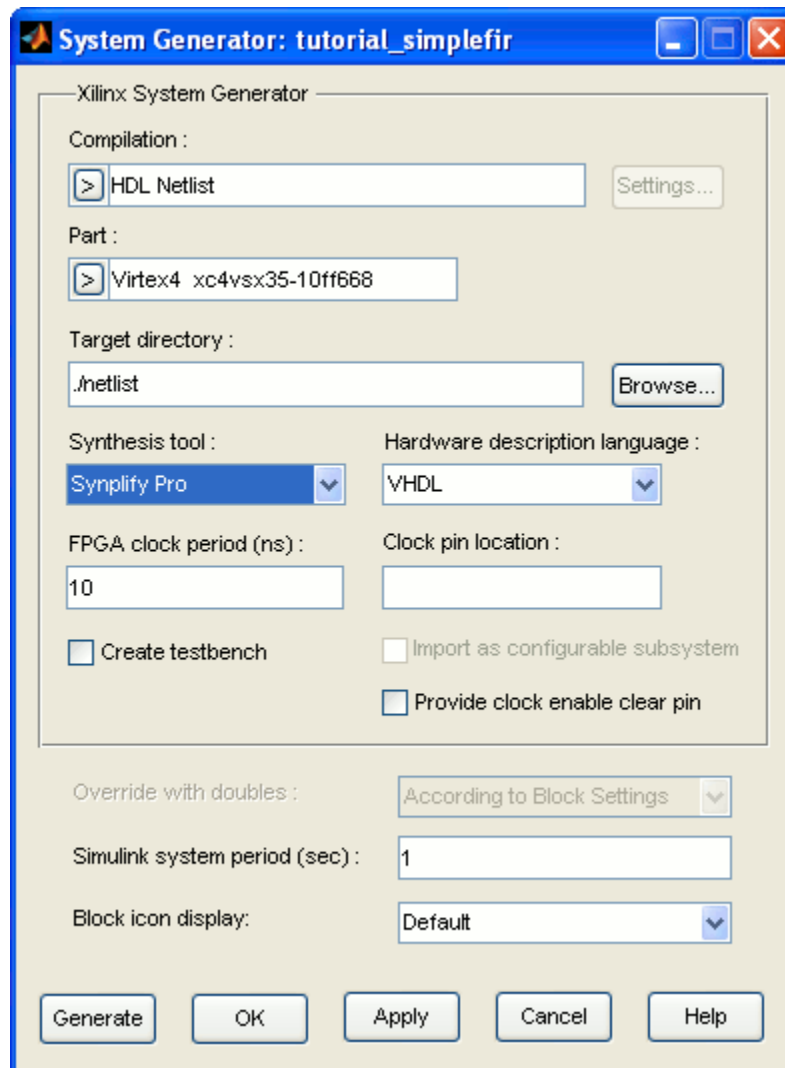


Figure 4: System Generator Dialog Box

Some further explanation is in order. System Generator provides two Simulink simulation libraries, the “Xilinx Blockset” and “Xilinx Reference Blockset”. Each contains blocks that can be used to construct models in Simulink. These blocks provide arithmetic, logic, memory, and signal processing abstractions suitable for implementing DSP systems in an FPGA. In addition to these simulation libraries, System Generator provides implementations for each function, and code generation software for translating subsystems comprised of Xilinx blocks into hardware descriptions of the system model. The simulation models are bit-accurate to the hardware description. They are also clock cycle accurate to the hardware at sample rates observable in Simulink. Consequently, the detailed hardware behavior can be understood from the Simulink simulation.

A System Generator model is constructed of blocks from the Xilinx blocksets, each of which

contributes to the hardware description created by the code generator. Like the testbench in the filter example, models typically involve other Simulink blocks for providing stimuli to the System Generator design and for analyzing or visualizing the output of the design. One of the primary advantages of designing hardware using System Generator is the rich simulation framework Simulink provides to stimulate, debug, and analyze the executable model.

Every System Generator model must contain a “System Generator” block, which is found in the “Xilinx Blockset/Basic Elements” library. This block is used to configure and invoke the code generator, and to resolve the relative time scale between Simulink simulation and the clock signal running in the FPGA hardware. The “FPGA clock period” defines the clock frequency in hardware, and the “Simulink System Period” defines the “Simulink time” that corresponds to this clock frequency. As shown in Figure 4, one second in Simulink corresponds to ten nanoseconds, i.e., a 100MHz clock. It is often expedient to normalize sample times in Simulink to integers for convenience. The software reports a Simulink error if a model does not contain a System Generator block.

Each realizable signal (i.e., translatable into hardware) must be sampled and be either a fixed-point or Boolean type. Simulink allows the modeling of continuous time signals (denoted by a sample time of zero), but System Generator does not support this abstraction, since its goal is to model realizable digital systems. Data types and sample rates are propagated through a System Generator model under normal Simulink propagation rules. This means that any source to a System Generator subsystem must be sampled and quantized. Signals passing from Simulink blocks to Xilinx blocks must go through Xilinx gateways, which enforce these constraints. Getting back to the model, double-clicking on the “gateway in” block exposes the controls by which the signal type can be defined (Figure 5).

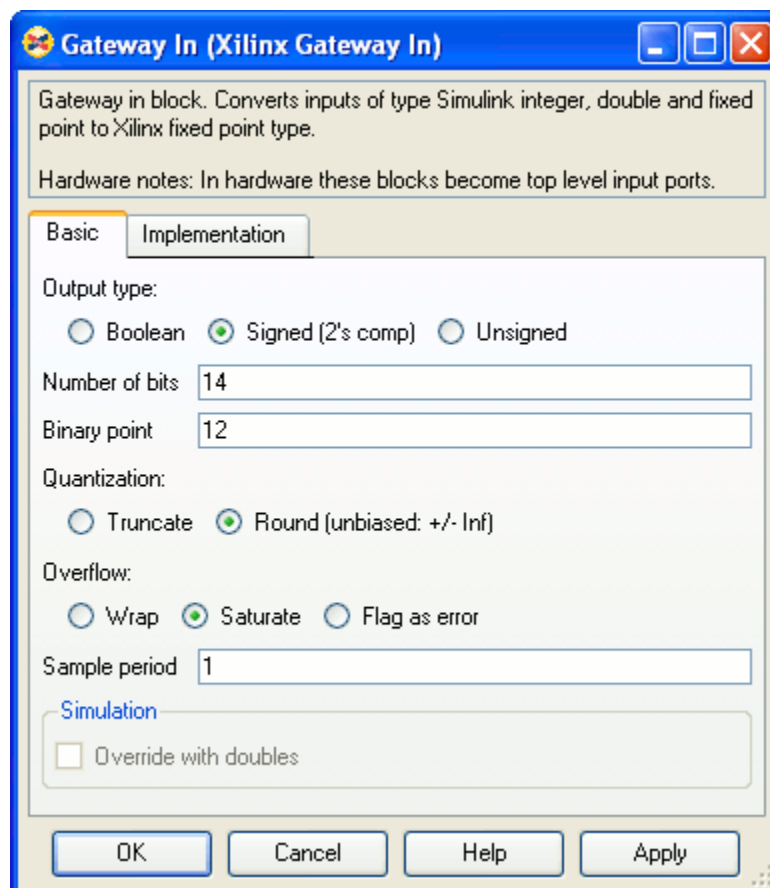


Figure 5: Gateway In Dialog Box

An output data type can be boolean, unsigned, or signed two's complement, with a user-defined numerical precision. Figure 5 shows a gateway that coerces inputs to be a 14-bit signed two's complement value, with 12 fractional bits. The sample time must be a positive number, indicating the rate at which the input signal is sampled.

In addition to coercing input type, gateways (and indeed most Xilinx blocks) also affect how hardware is generated. A gateway is ordinarily converted into a port on the top level entity. This port can bound to a particular input-output block (IOB) on an FPGA.

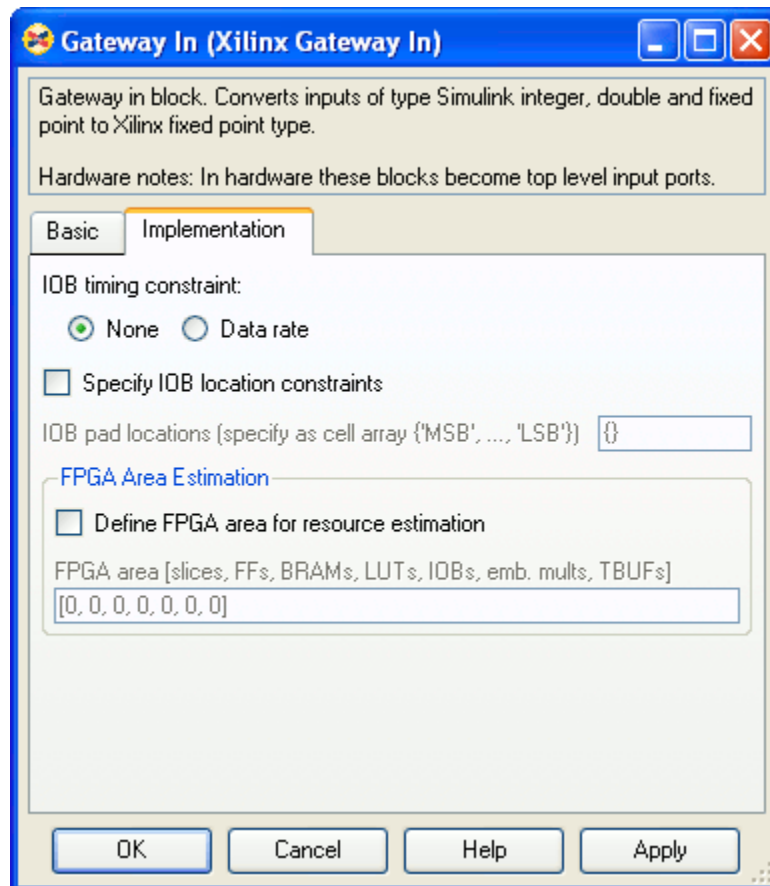


Figure 6 : Gateway Implementation Tab

An FPGA provides considerable flexibility in defining the arithmetic precision throughout a computation. System Generator blocks support arbitrary-precision fixed-point arithmetic; for example a 37-bit wide data path can be constructed if required. The amount of FPGA hardware used is directly related to the data width, so it is best (for reasons of cost, circuit speed, and power dissipation) to request only the precision required for a particular application.

Subsystems and Mask Variables

In programming, functions are often used to encapsulate a computation so that it can be used repeatedly without having to duplicate code wherever it is required. In addition, a function can insulate its caller from having to worry about the details of its implementation. Functions can also be used to organize a complex algorithm into manageable components.

In Simulink, subsystems play a similar role, providing a means of using hierarchy to manage system complexity, as well as to reuse components. For example, in DSP applications, a multiply-accumulate (MAC) function is a commonly occurring function. The System Generator libraries do not provide this as a primitive block, but one can be easily constructed using a multiplier and accumulator block as shown in **Figure 1**. (Click to [Open this model](#))

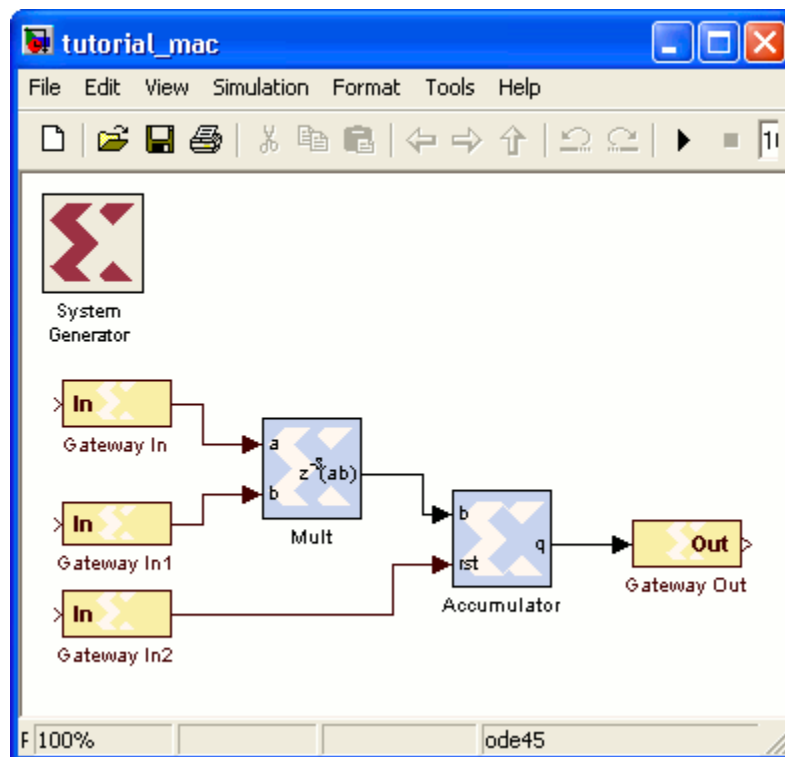


Figure 1: A multiply-accumulate (MAC) function.

Left-clicking a bounding box around the multiplier and accumulator blocks selects them in the editor. Typing Ctrl-g creates a new Simulink subsystem containing the selected blocks, and inserts this subsystem in **place**. This subsystem can be copied by right-clicking and dragging the mouse in the block editor. Double-clicking on the subsystem opens it in the editor. Renaming the input and output ports with meaningful names (e.g., the port names of the block ports they drive) makes the subsystem easier to reuse.

Functions become far more useful reusable components when they have parameters (i.e., arguments) that control their behavior. Similarly, it is often desirable to define parameters on a subsystem that control its behavior. In Simulink, this is done by “masking” a subsystem (i.e., creating an interface that provides a “face” or mask to a “caller”). It is then possible to define mask parameters that can be used within the scope of the subsystem, much as variables can be used in a block-scoped programming language. This is accomplished by right-clicking on the subsystem as shown in

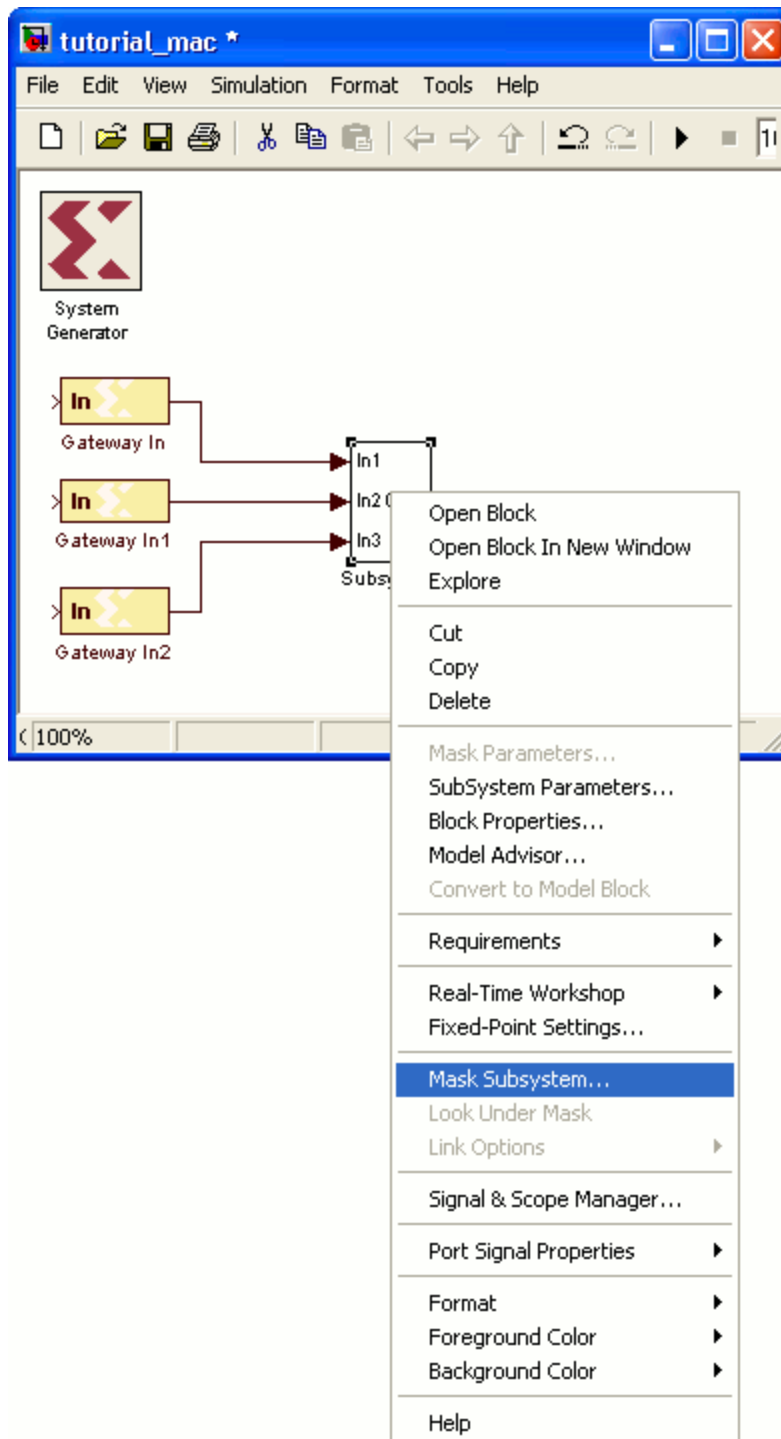


Figure 2, and selecting **Mask Subsystem** from the pull-down menu.

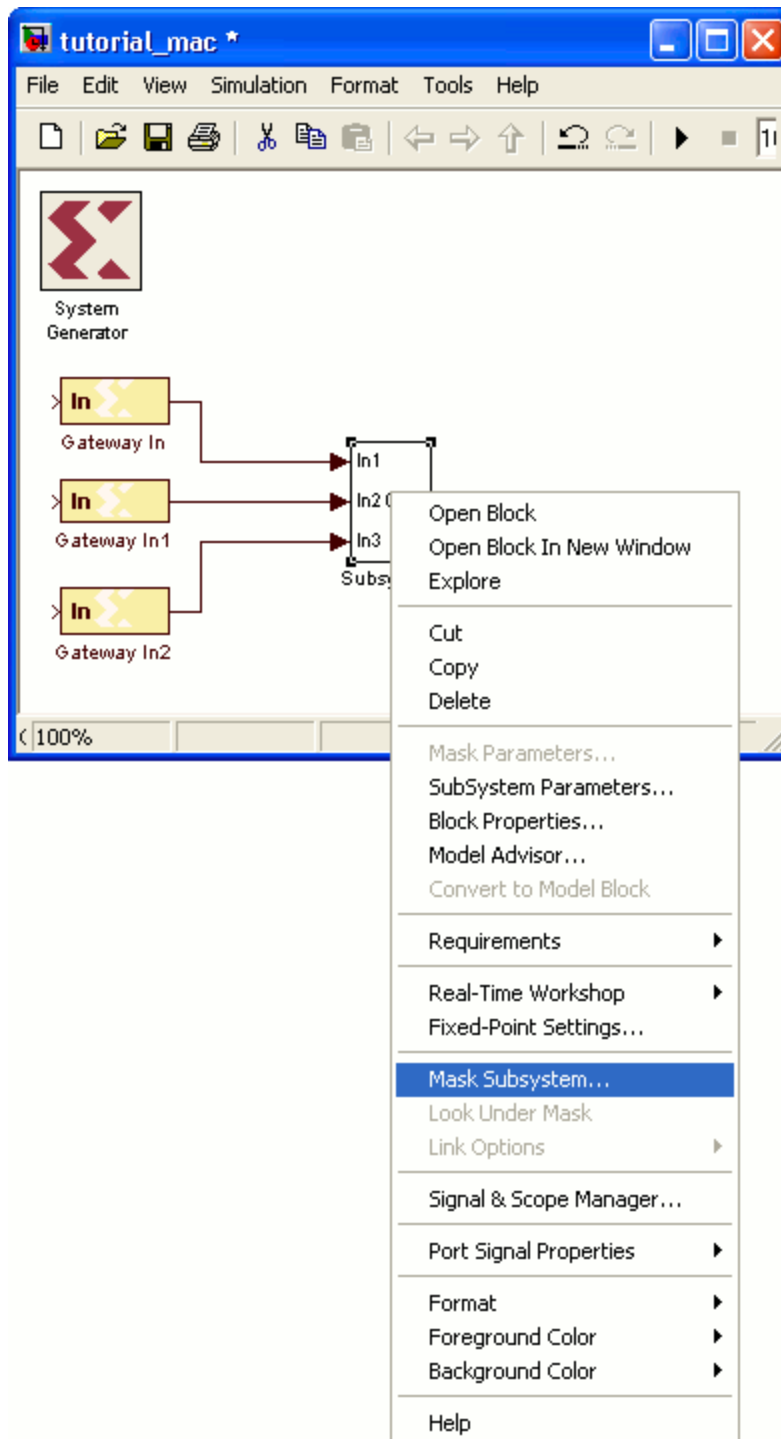


Figure 2: Masking a subsystem.

Once a subsystem has been masked, double-clicking on it opens up a Mask Dialog Box that provides definitions for its mask parameters. Simulink provides a mask editor for adding mask parameters and binding graphical controls to their customization. Right-clicking on the masked subsystem and selecting **Edit Mask** from the pull-down menu opens up the mask editor. In the case of our MAC engine, it is useful to define mask parameters to specify the input and output types of the multiplier and the accumulator (**Figure 3**).

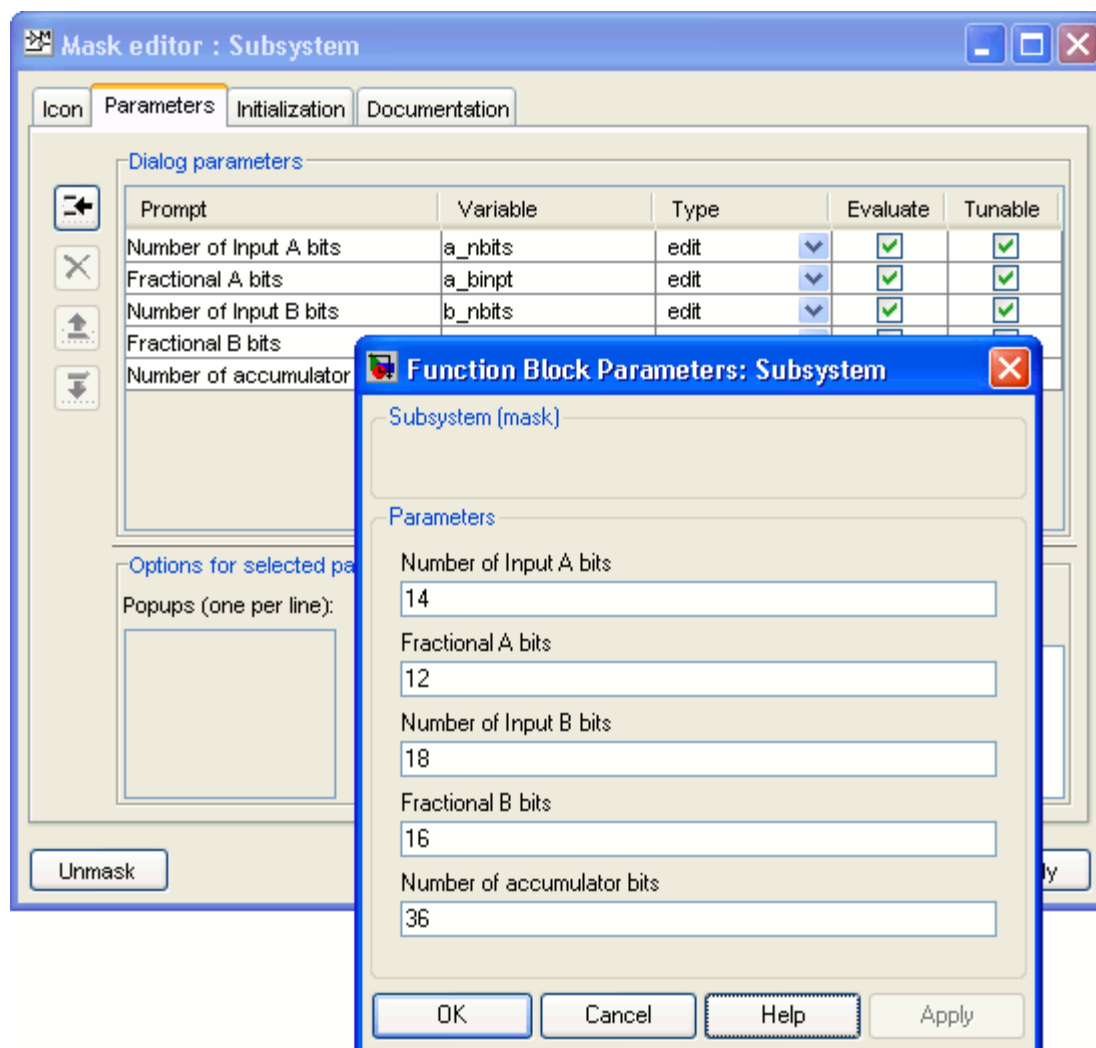


Figure 3: MAC engine mask

If there are sensible default values for the mask parameters, they can be entered in the mask Dialog Box. As shown, the precision of the A input is 14 bits with 12 bits of fraction, the B input is 18 bits with 16 bits of fraction, and the accumulator is defined to have 36 bits.

The mask parameters for the MAC subsystem can be used to control the precision of the multiplier and accumulator blocks inside the subsystem, so that the blocks automatically change precision when the MAC parameters are changed. To enter the subsystem in the block editor, right-click and select **Look Under Mask** from the pull-down menu.

Parameters do not have to be initialized with constant values. Any parameter that is customized using a Dialog Box can be defined by a MATLAB expression (including function calls). For example, in the multiplier block, the precision can be set to be $a\_nbits + b\_nbits$ wide with $a\_binpt + b\_binpt$ fractional bits. This ensures that a minimal number of FPGA resources are used, while avoiding arithmetic overflow. The number of bits in the accumulator should be set to be $accum\_nbits$.

Defining individual block parameters in terms of a subsystem's mask parameters allows the programmer to define broadly parametric models in Simulink. Block customization becomes a combination of user specification (via dialog boxes), data type propagation (via Simulink's propagation rules), and function application. For example, controlling the arithmetic precision by

parameters allows our MAC subsystem to “right-size” itself in response to the context in which it is instantiated. The implications of having access to the MATLAB interpreter during model customization should not be underestimated; the simple MAC example barely scratches the surface of what can be done.

Before moving on, it is worth presenting another way in which MATLAB can be used to customize a System Generator model. We have seen that Simulink diagrams have scoping rules for mask variables that are similar to those of a block-scoped programming language, and that block parameters can be defined in terms of subsystem masks. In fact, there is also a “global” scope that is also available for setting block parameters; this is the MATLAB workspace.

Any variable that has been defined in the MATLAB workspace (e.g., by typing directly in the command shell or by executing a MATLAB script) can also be used (subsystem masks can shadow this). Often during the design phase of an algorithm, it is desirable to iterate through a range of parameters. By defining workspace variables (say, in a MATLAB script) and binding block parameters to these variables, it is possible to achieve sweeping parameter changes with local modifications of a single script.

Simulink provides a mechanism to bind a MATLAB script to a model so that the script is executed every time the model is opened. This can ensure that every block parameter is well defined. As an example, suppose the MATLAB script `mac_init.m` contains the following variable definitions:

```
a_nbits = 14;  
a_binpt = 12;  
b_nbits = 18;  
b_binpt = 16;  
accum_nbits = 36;
```

This script can be bound to the MAC model by opening the top level model (not the MAC subsystem), typing the following command in the MATLAB command window, and then saving the model.

```
set_param(gcs, 'PreloadFcn', 'mac_init');
```

Every model has many parameters. One of these is the “pre-load” function that Simulink executes before the model is opened. The `set_param` command assigns the script `mac_init.m` to this model parameter (`gcs` is a function that returns the current system).

It is now possible to double click on the MAC subsystem and change the default values of the mask parameters to be `a_nbits`, `a_binpt`, `b_nbits`, `b_binpt`, `accum_nbits` rather than constants. Note the mask parameters shadow the variables defined in the MATLAB workspace.

Sample Rate Changing Blocks

It is desirable to adjust sample rates throughout a design so that every subsystem is clocked only at a rate necessary to compute its input-output relation. Doing so leads to more efficient use of resources, and to implementations that require less power. Significant resource savings can be accomplished by time-division multiplexing a data path to service multiple data streams that operate at a lower frequency than the processor. Polyphase decompositions of digital filters are commonly used to reduce workload (and power dissipation).

In System Generator, rate changes are effected through rate-changing blocks.

- Up Sample, Down Sample
- Assert
- Time Division Multiplexer, Time Division Demultiplexer
- Serial to Parallel, Parallel to Serial
- Dual-Port memory, Register, Addressable Shift Register
- FIR Filter (multi-rate modes)
- FIR Compiler v1_0 (multi-rate modes)

The Sysgen application provides labeling to denote parts of a model that run at different rates (i.e., each portion of a model that runs at a particular rate has the label in the diagram). This can be useful for a quick visualization of the rates in your system. To turn on this feature, select the Sysgen block and then modify the **Block Icon Display** to **Sample Rates** (See **Figure 1: Sample Rates**). After the next simulation run (**Ctrl-t**) the ports of blocks will be labeled with related rate information (See **Figure 2**).

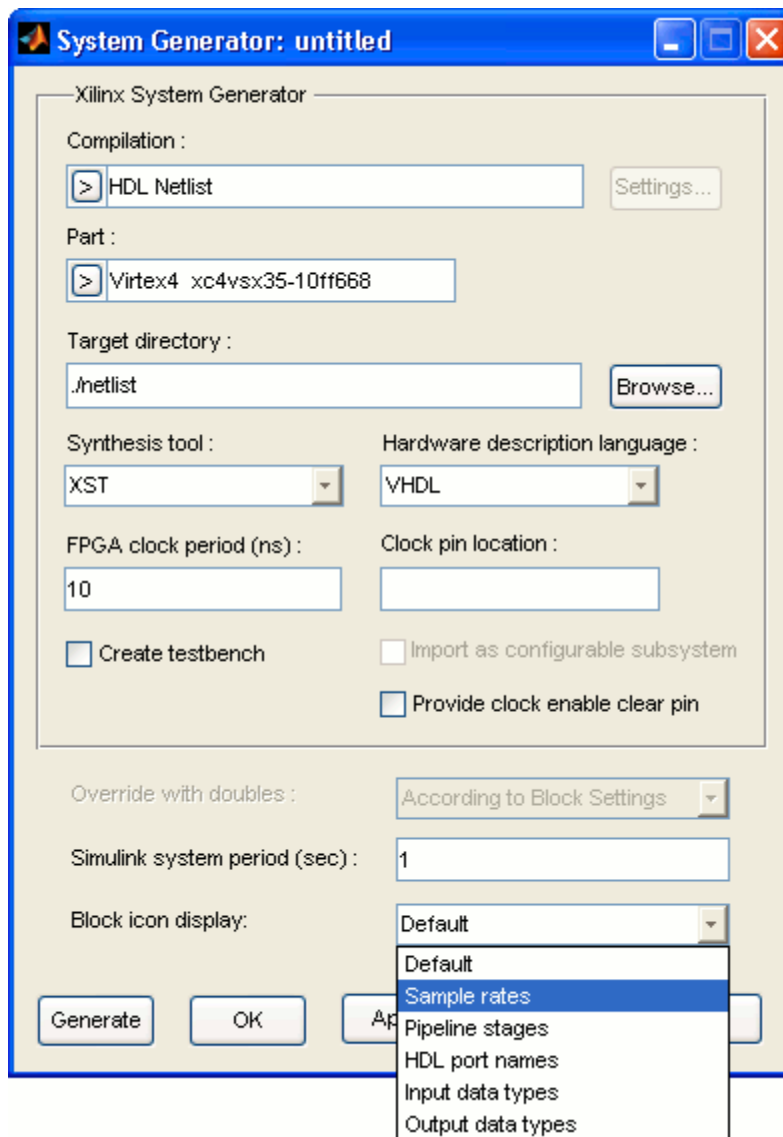


Figure 1: Sample Rates

The up-sample and down-sample blocks are used for rate conversion and rate matching. As an example, the down-sample block can be used to construct a decimating commutator that breaks a data stream into lower rate polyphase streams. A one-to-two commutator, shown in **Figure 2**, is constructed using two down-samplers and a delay block. In this model, the System Generator “Sample Time” blocks are used to extract the sample periods both before and after the down-sampler to show the difference in sample times.

Click on this link ([Commutator](#)) to open a simulink model containing a fully synthesizable commutator.

The down-sampler block has mask parameters for controlling the down-sampling factor. It also has a phase-selection parameter with two choices: the first or last sample of an output frame. When translating the block into a hardware description, the latter choice is considerably more efficient; it requires only a single register. Selecting the first sample of a frame (as shown in the example in **Figure 2**) results in an additional multiplexer in the hardware, which not only requires additional logic resources, but also introduces a combinational path into the circuit that can in some circumstances

lower the achievable clock rate in the system.

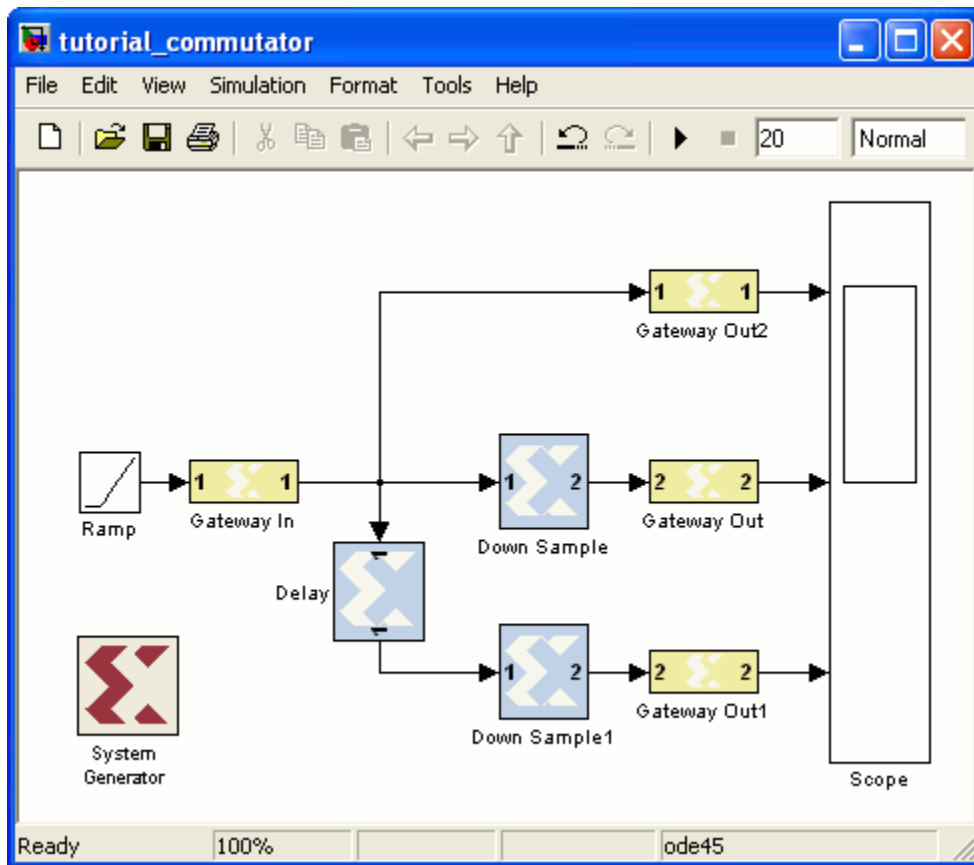


Figure 2: A one-to-two polyphase commutator.

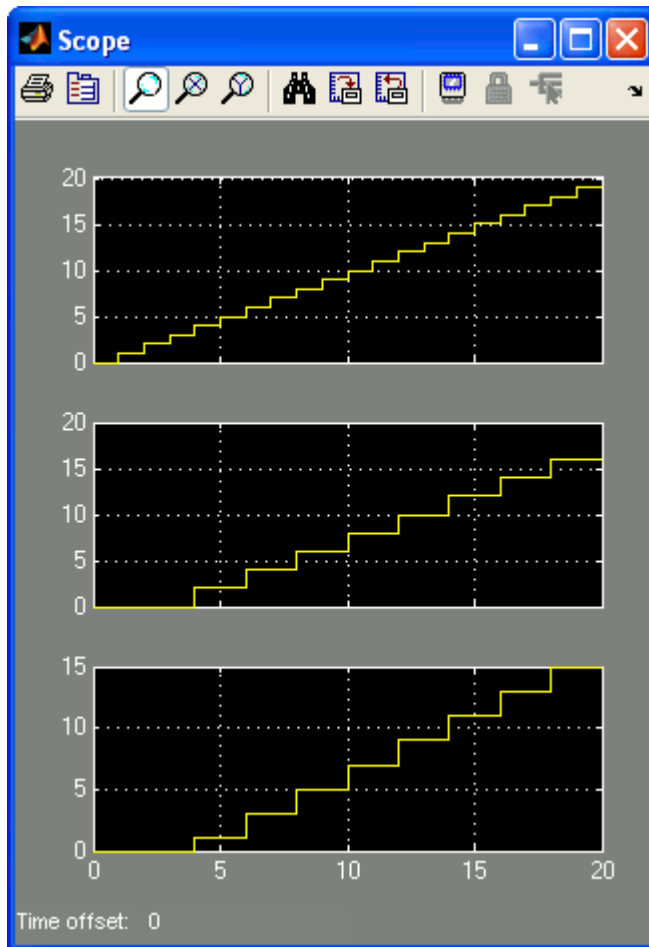


Figure 3: Scope of the one-to-two polyphase commutator.

The up-sample block is used for increasing a sample rate by an integer factor. It has two modes. The first is for use in an interpolator, where zeros are inserted between successive input samples ($L-1$ zeros are inserted for an up-sample by L interpolator). The second mode simply copies input samples ($L-1$ copies are inserted for an up-sample by L rate change). The latter mode is useful because many System Generator blocks require all inputs to run at the same rate. Both modes are demonstrated in the model shown in **Figure 4**.

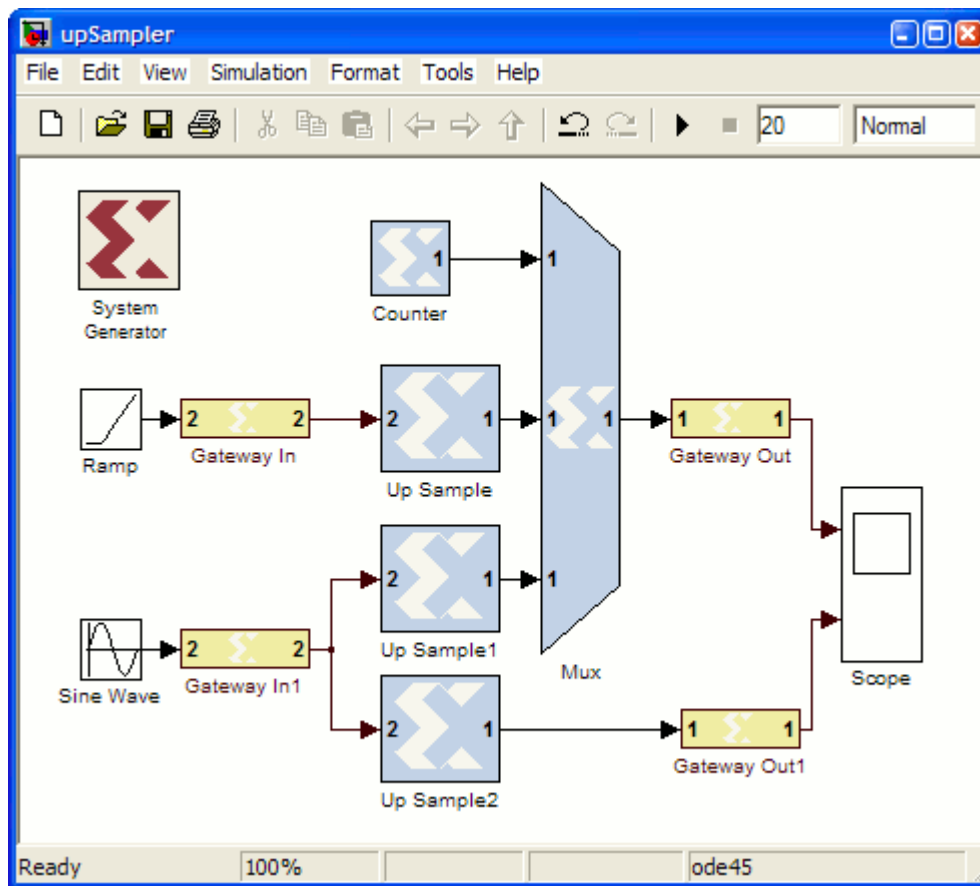


Figure 4: Interleaving and interpolating up samplers

In this model, each up-sampler doubles its input rate. The first two up-samplers copy their input samples so that all inputs to the multiplexer (whose select line is driven by a one-bit counter running at the up-sampled rate) are the same. The effect is to interleave the two data streams into a single stream running at twice the rate. The third up-sampler demonstrates the interpolation mode, with a zero inserted between each input sample.

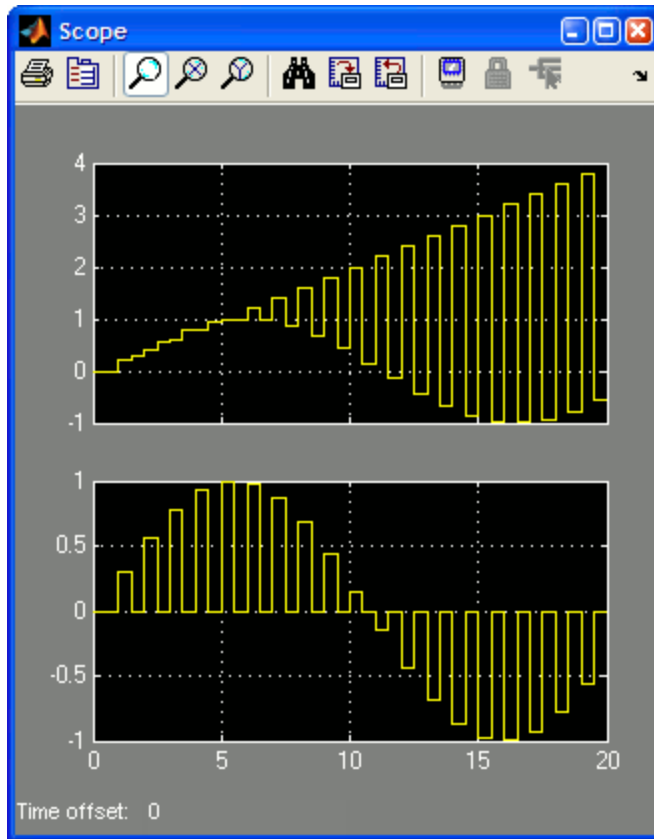


Figure 5: Scope of the Interleaving and interpolating up samplers

Branching Constructs

Block diagrams are a natural way to describe many computations, especially data path intensive systems. System Generator provides a wide variety of blocks for constructing FPGA circuits, ranging from arithmetic operators and memories, to DSP functions like FIR filters, FFTs, and error correcting codes. However, there are some functions that are better described using a language other than a Simulink block diagram. For example, multi-way branching constructs like “switch” and “if then else” statements are often more conveniently expressed in MATLAB code or a hardware description language (HDL) than they are in a block diagram.

System Generator provides several mechanisms in addition to the basic logic blocks to facilitate building “control-oriented” branching functions. The first is a Black Box block that can be used to import an HDL module written in VHDL or Verilog into Simulink (of course, the HDL module is not restricted to being a control function). The second mechanism is an m-code block that imports a MATLAB function into a System Generator model. Yet another mechanism for implementing sophisticated control functions is an 8-bit microcontroller that has been integrated into the Xilinx Blockset.

Each of these mechanisms is described in detail later in the User Guide. In this section, we use the MCode block to implement a finite state machine, a commonly used class of control functions that involve branching constructs. The MCode block accepts as a parameter a combinational function written in the MATLAB language. The function can include arbitrarily nested branching statements (switch, if, then, else), conditional operators, variable assignment, and a limited set of arithmetic operators.

Consider the following specification of a finite state machine that describes the simplified behavior of an electric car, with transition function shown in **Figure 1**.

Inputs:	Accelerator, brake
Outputs:	Velocity
States:	Stopped, Moving

In the transition diagram, the label $a'+b/0$ on the arc from the stopped state back to itself is interpreted to mean that if the accelerator is not pressed or the brake is applied, the state of the machine remains stopped and the velocity is zero.

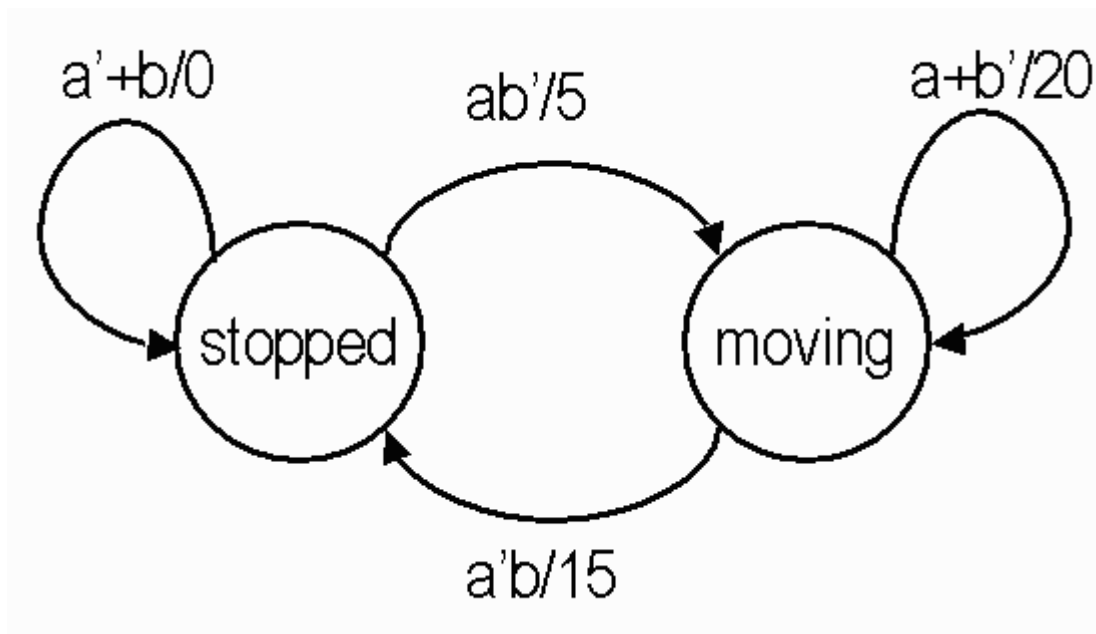


Figure 1: Finite state machine.

Here is the transition function in m-code, which of course can be developed and executed using the MATLAB interpreter:

```

function [n,v] = fsm(a,b)
    stop = 0, moving = 1; % symbolic constants
    proto = {xlUnsigned,1,0};
    persistent s, s= xl_state(stop,proto);
    switch double(s)
        case stop
            if a & ~b, n = 1, v = 5;
            else, n = 0, v = 0;
            end
        case moving
            if a | ~b, n = 1, v = 20;
            else, n = 0, v = 15;
            end
        otherwise, n = 0, v = 0;
    s = n
end

```

In MATLAB, the input arguments can be fixed-point values (System Generator provides a MATLAB fixed-point type) or double precision. When the function is imported into the m-code block, the types are inferred from the block inputs. Output types are inferred to be the minimal fixed-point type based on usage, or can be specified explicitly via constructors based on MATLAB objects.

This style of transition function leads to a classical Huffman state machine implementation (**Figure 2**).

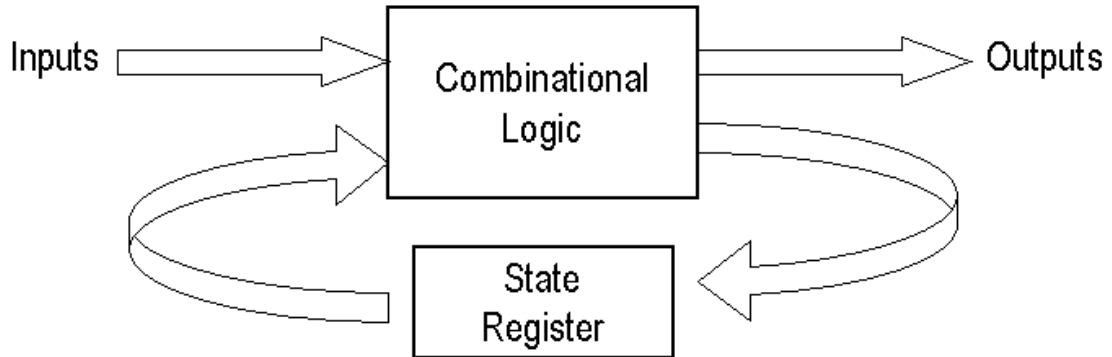


Figure 2: A Huffman finite state machine.

The System Generator model looks similar to the Huffman machine. The m-function is specified as a parameter to the m-code block, which customizes its icon to match the inputs and outputs as shown in Figure 3. The state register is implemented within the mcode persistent variable.

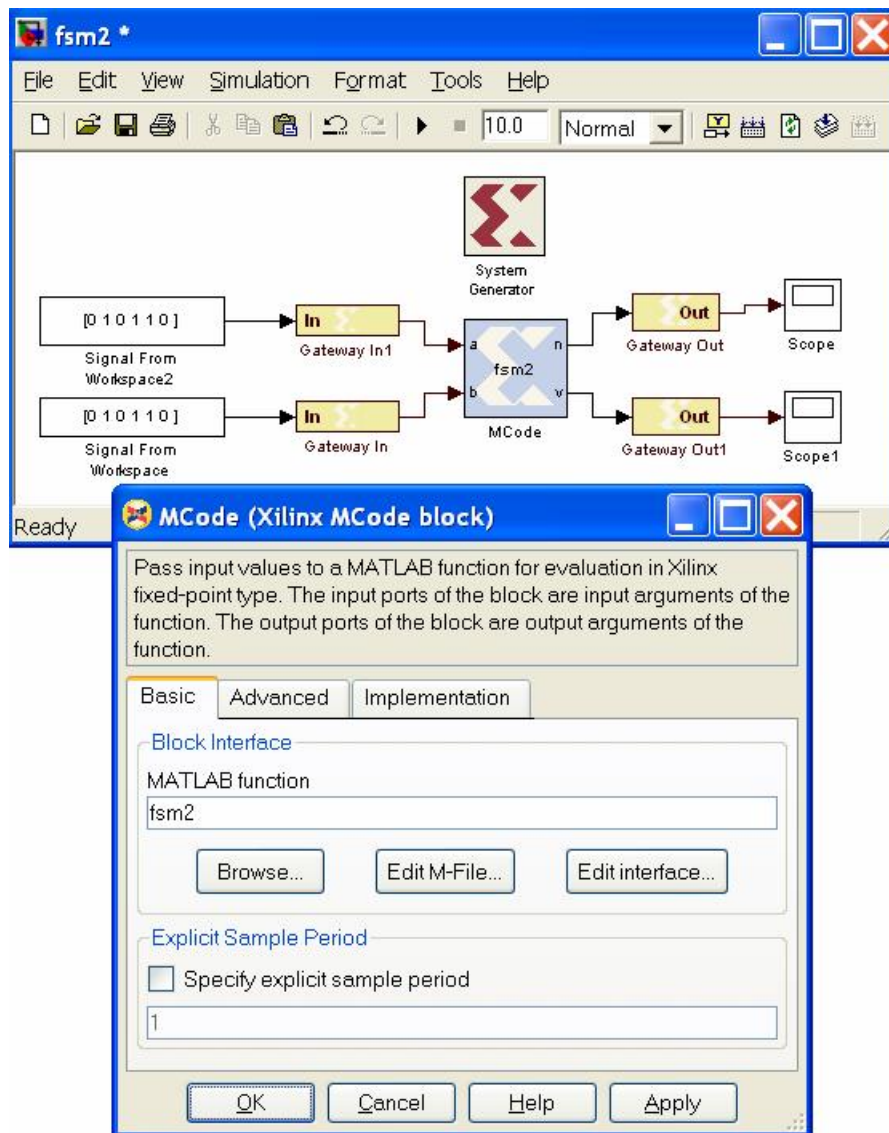


Figure 3: M-code based finite state machine.

During code generation, the m-function is translated into an equivalent VHDL module. In our example, the two-state machine is encoded with a one-bit register. For a one-hot state encoding, the register width would be two bits, and the m-code would be changed to encode states into the values 1 and 2.

Hardware Co-Simulation

System Generator blocksets allow you to construct bit-accurate and cycle-accurate models of an FPGA circuit in Simulink. Nevertheless, it is universally true that a hardware engineer wants to see the design running in hardware. System Generator provides hardware co-simulation interfaces that make it possible to incorporate an FPGA directly into a Simulink simulation. The code generator has “Hardware Co-simulation” compilation targets (analogous to the HDL Netlist target) that automatically create a bitstream. After creating the bitstream, System Generator automatically incorporates an FPGA hardware platform configured with this bitstream back into Simulink as a run-time block. When the design is simulated in Simulink, results for the compiled portion are calculated in hardware. This allows the compiled portion to be tested in actual hardware, and can speed up simulation dramatically. This process and other aspects of hardware co-simulation are the subject of Chapter 4 of this User Guide.

At this point, we have touched upon some of the mechanics of building FPGA realizations of digital systems using System Generator. It is possible to build surprisingly useful, high performance systems using the ideas we have touched upon. And it is probably a good idea to take the time to put this assertion to the test before continuing with this document.

We have avoided nearly all detail, and of course, understanding the details is required to take full advantage of the software. The remainder of this User Guide attempts to fill in what has been left out in this brief introduction. We have not described the blocksets, but they are better explored than explained. Details for each of the blocks are available in the on-line help (i.e., “Help” button on the Dialog Boxes) and in the reference material in Chapter 8.

System Generator provides demonstration examples of varying complexity that can be used as starting points for further exploration (the Xilinx Reference Blockset is another good place to start). Each design is self-documented, and indicates ways in which you can build efficient hardware using Simulink as a programming language. To run these demos, type demo in the MATLAB command window and navigate to the Xilinx Blockset designs.

Hardware Design Using System Generator

A Brief Introduction to FPGAs

A field programmable gate array (FPGA) is a general-purpose integrated circuit that is “programmed” by the designer rather than the device manufacturer. Unlike an application-specific integrated circuit (ASIC), which can perform a similar function in an electronic system, an FPGA can be reprogrammed, even after it has been deployed into a system.

A FPGA is programmed by downloading a configuration program called a *bitstream* into static on-chip random-access memory. Much like the object code for a microprocessor, this bitstream is the product of compilation tools that translate the high level abstractions produced by a designer into something equivalent but low level and executable. Xilinx System Generator pioneered the idea of compiling an FPGA program from a high level Simulink model.

An FPGA provides the user with a two-dimensional array of configurable resources that can implement a wide range of arithmetic and logic functions. These resources include dedicated DSP blocks, multipliers, dual port memories, lookup tables (LUTs), registers, tri-state buffers, multiplexers, and digital clock managers. In addition, Xilinx FPGAs contain sophisticated I/O mechanisms that can handle a wide range of bandwidth and voltage requirements. The Virtex-4 and Virtex-II Pro family FPGAs include embedded microcontrollers (IBM PowerPC 405), and multi-gigabit serial transceivers. The compute and I/O resources are linked under the control of the bitstream by a programmable interconnect architecture that allows them to be wired together into systems.

FPGAs are high performance data processing devices. FPGA DSP performance is derived from the ability they provide to construct highly parallel architectures for processing data. In contrast with a microprocessor or DSP processor, where performance is tied to the clock rate at which the processor can run, FPGA performance is tied to the amount of parallelism that can be brought to bear in the algorithms making up a signal processing system. A combination of increasingly high system clock rates (current system frequencies of 100-200 MHz are common today) and a highly distributed memory architecture gives the system designer an ability to exploit parallelism in DSP (and other) applications that operate on data streams. For example, the raw memory bandwidth of a large FPGA running at a clock rate of 150 MHz can be hundreds of terabytes per second.

There are many DSP applications (e.g., digital up/down converters) that can be implemented only in custom integrated circuits (ICs) or in an FPGA; a von Neumann processor lacks both the compute capability and the memory bandwidth required. Advantages of using an FPGA include significantly lower non-recurring engineering costs than those associated with a custom IC (FPGAs are commercial off-the-shelf devices), shorter time to market, and the configurability of an FPGA, which allows a design to be modified, even after deployment in an end application.

When working in System Generator, it is important to keep in mind that an FPGA has many degrees of freedom in implementing signal processing functions. You have, for example, the freedom to define data path widths throughout your system and to employ many individual data processors (e.g., multiply-accumulate engines), depending on system requirements. System Generator provides abstractions that allow you to design for an FPGA largely by thinking about the algorithm you want to

implement. However, the more you know about the underlying FPGA, the more likely you are to exploit the unique capabilities an FPGA provides in achieving high performance.

In the remainder of this section, we provide a brief introduction to some of the logic resources available in the FPGA, so that you gain some appreciation for the abstractions provided in System Generator.

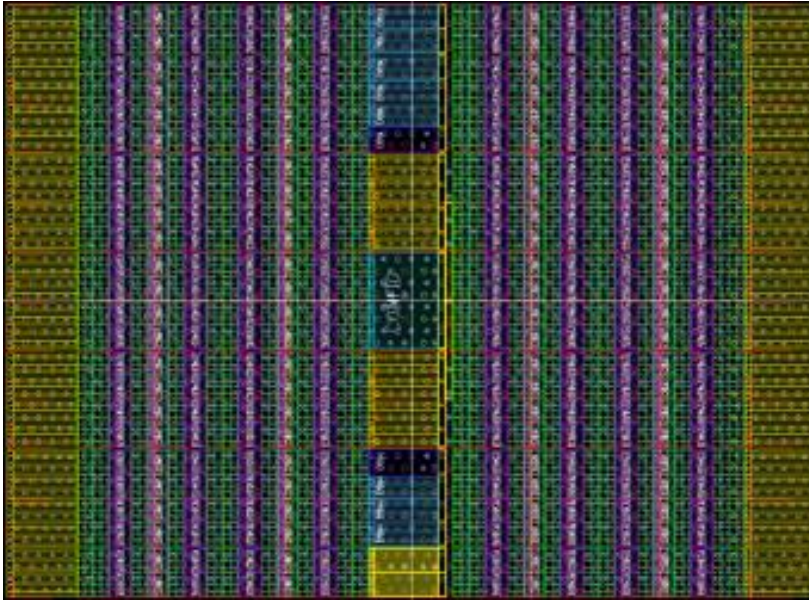


Figure 1: Xilinx Virtex-4 SX FPGA

Figure 1 shows a physical view of a Virtex-4 FPGA. To a signal processing engineer, an FPGA can be thought of as a 2-D array of logic slices striped with columns of hard macro blocks (block memory and arithmetic blocks) suitable for implementing DSP functions, embedded within a configurable interconnect mesh. In a Virtex-4 FPGA, the DSP blocks (Figure 2) can run in excess of 450 MHz, and are pitch-matched to dual port memory blocks (BRAMs) whose ports can be configured to a wide range of word sizes (18 Kb total per BRAM). The Virtex-4 SX55 device contains 512 such DSP blocks and BRAMs. In System Generator, you can access all of these resources through arithmetic and logic abstractions to build very high performance digital filters, FFTs, and other arithmetic and signal processing functions.

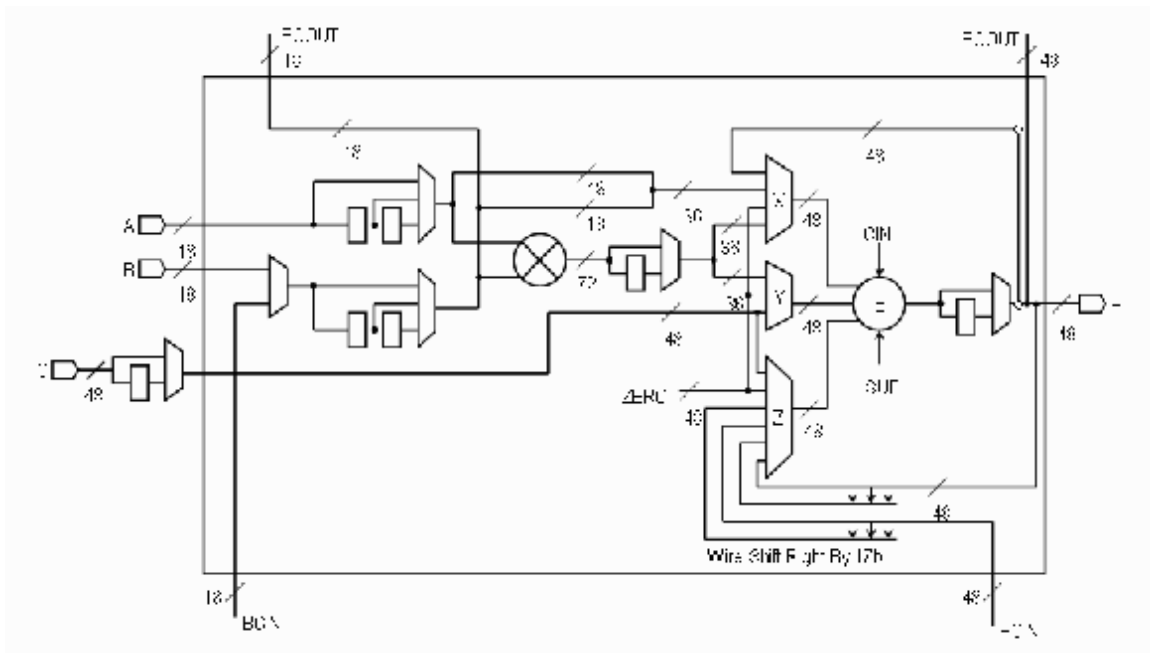


Figure 2: Virtex-4 DSP48 Block

While the multiply-accumulate function supported by a Virtex-4 DSP block is familiar to a signal processing engineer, it is instructive to take a closer look at the Virtex family logic slice (Figure 2), which is the fundamental unit of the logic fabric array.

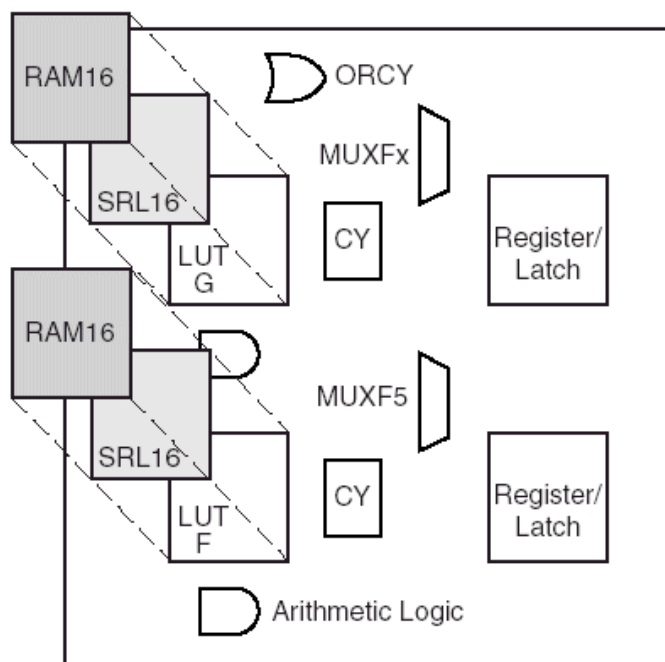


Figure 3: Virtex Family FPGA Logic Slice

Each logic slice contains two 4-input lookup tables (LUTs), two configurable D-flip flops, multiplexers, dedicated carry logic, and gates used for creating slice based multipliers. Each LUT can implement an arbitrary 4-input Boolean function. Coupled with dedicated logic for implementing fast carry circuits, the LUTs can also be used to build fast adder/subtractors and multipliers of essentially

any word size. In addition to implementing Boolean functions, each LUT can also be configured as a 16x1 bit RAM or as a shift register (SRL16). An SRL16 shift register is a synchronously clocked 16x1 bit delay line with a dynamically addressable tap point.

In System Generator, these different memory options are represented with higher level abstractions. Instead of providing a D-flip flop primitive, System Generator provides a register of arbitrary size. There are two blocks that provide abstractions of arbitrary width, arbitrary depth delay lines that map directly onto the SRL16 configuration. The *delay* block can be used for pipeline balancing, and can also be used as storage for time division multiplexed (TDM) data streams. The *addressable shift register* (ASR) block, with function depicted in Figure 2, provides an arbitrary width, arbitrary depth tapped delay line. This block is of particular interest to the signal processing engineer, since it can be used to implement tapped delay lines as well as sweeping through TDM data streams.

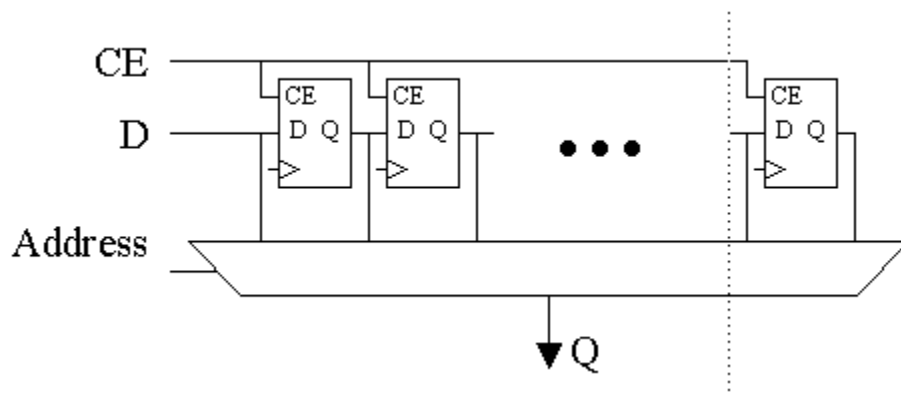


Figure 4: System Generator Addressable Shift Register

Although random access memories can be constructed either out of the BRAM or LUT (RAM16x1) primitives, doing so can require considerable care to ensure most efficient mappings, and considerable clerical attention to detail to correctly assemble the primitives into larger structures. System Generator removes the need for such tasks.

For example, the *dual port RAM* (DPRAM) block shown in Figure 4 maps efficiently onto as many BRAM or RAM16x1 components on the device as are necessary to implement the desired memory. As can be seen from the mask dialog box for the DPRAM, the interface allows the user to specify a type of memory (BRAM or RAM16x1), depth (data width is inferred from the Simulink signal driving a particular input port), initial memory contents, and other characteristics.

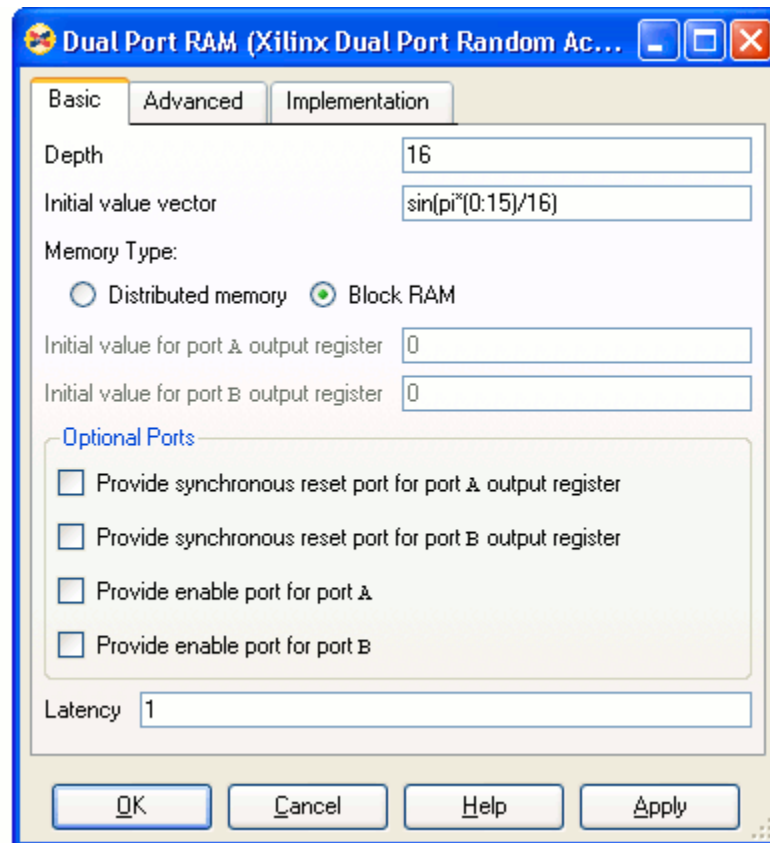


Figure 5: System Generator Dual Port RAM

In general, System Generator maps abstractions onto device primitives efficiently, freeing the user from worrying about interconnections between the primitives. System Generator employs libraries of intellectual property (IP) when appropriate to provide efficient implementations of functions in the block libraries. In this way, a user does not always have to have detailed knowledge of the underlying FPGA details. However, when it makes sense to implement an algorithm using basic functions (e.g., adder, register, memory), System Generator allows you to exploit your FPGA knowledge while reducing the clerical tasks of managing all signals explicitly.

System Generator library blocks and the mapping from Simulink to hardware are described in detail in subsequent sections of this User Guide and in the Reference Guide. There is a wealth of detailed information about FPGAs that can be found at <http://support.xilinx.com>, including data books, application notes, white papers, and technical articles.

Note to the DSP Engineer

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

The more a DSP engineer knows about a hardware realization (e.g., how to exploit parallelism and pipelining), the better the implementation he or she obtains. Using IP cores makes it possible to have efficient FPGA designs that including complex functions like FFTs. System Generator also makes it possible to refine a model as befits the application.

Scattered throughout the User's Guide and block reference guide are notes that explain ways in which

system parameters can be used to exploit hardware capabilities.

Note to the Hardware Engineer

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

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

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

Finally, hardware co-simulation interfaces described in Using FPGA Hardware in the Loop allow the hardware engineer to run a design in hardware under control of Simulink, bringing the full power of MATLAB and Simulink to bear for data analysis and visualization.

Design Flows using System Generator

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

Algorithm Exploration

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

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

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

Implementing Part of a Larger Design

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

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

Implementing a Complete Design

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

- HDL that implements the design itself;
- A *clock wrapper* that encloses the design. This clock wrapper produces the clock and clock enable signals that the design needs.
- A HDL *testbench* that encloses the clock wrapper. The testbench allows results from Simulink simulations to be compared against ones produced by ModelSim.
- Project files and scripts that allow various synthesis tools (XST and Synplify) to operate on

System Generator HDL

- Files that allow the System Generator HDL to be used as a project in Project Navigator.

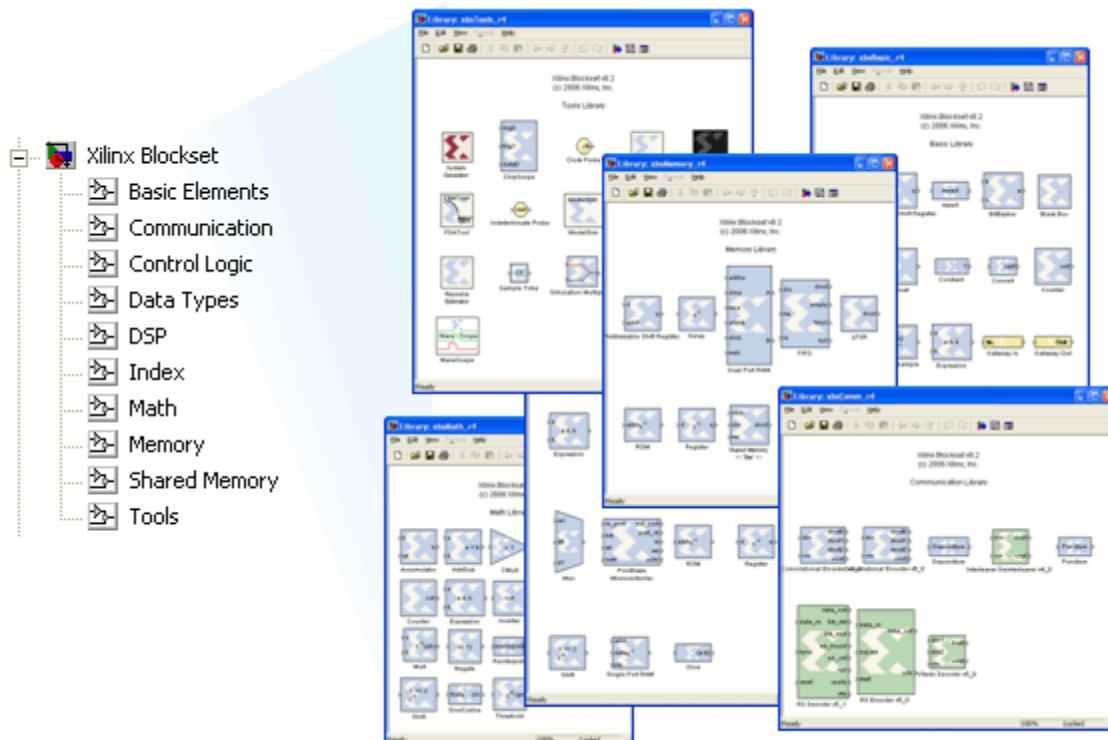
For details concerning the files that System Generator writes, see Compilation Results.

System Level Modeling in System Generator

System Generator Blocksets

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

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



Xilinx Blockset

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

Library	Description

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

Note: More information concerning blocks can be found in the System Generator Block Reference Pages.

Xilinx Reference Blockset

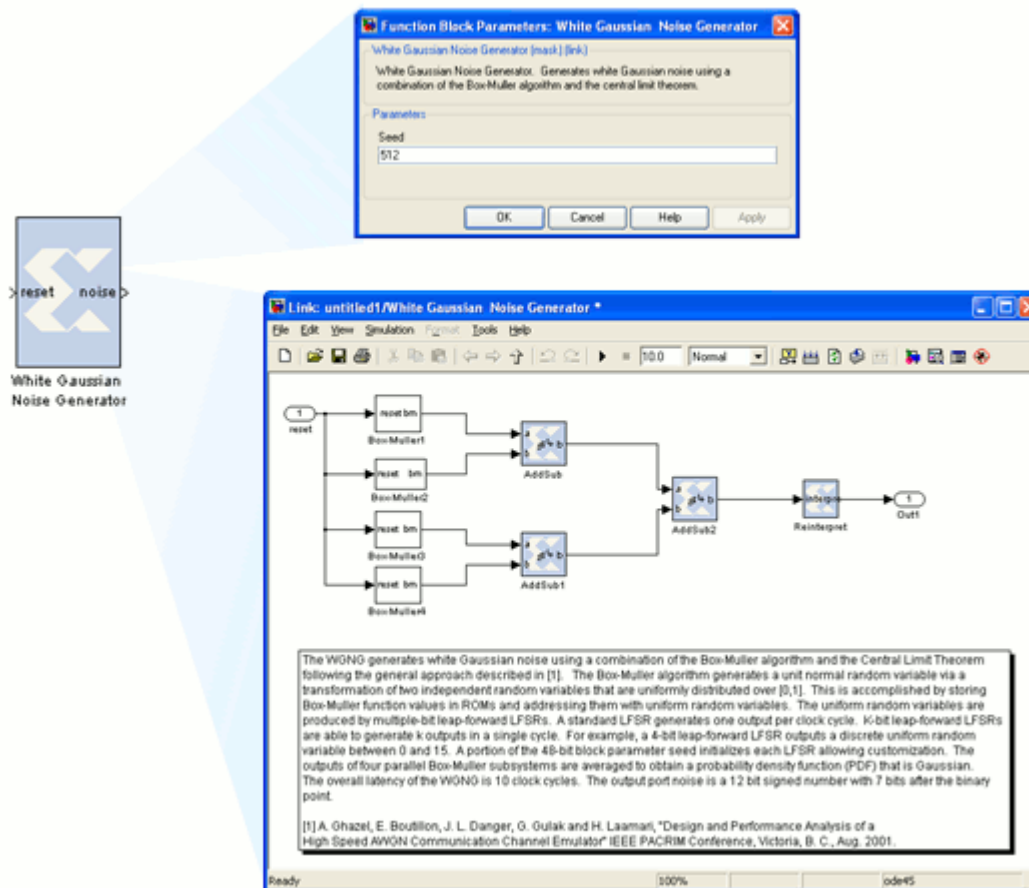
The Xilinx Reference Blockset contains composite System Generator blocks that implement a wide range of functions. Blocks in this blockset are organized by function into different libraries. The libraries are described below.

Library	Description
Communication	Blocks commonly used in digital communications systems.
Control Logic	Blocks used for control circuitry and state machines.
DSP	Digital signal processing (DSP) blocks.
Imaging	Image processing blocks.
Math	Blocks that implement mathematical functions.

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

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

System Generator Reference Design Library.



Signal Types

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

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

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

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

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

Bit-True and Cycle-True Modeling

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

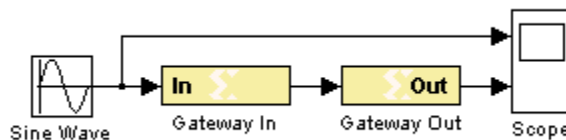
Timing and Clocking

Discrete Time Systems

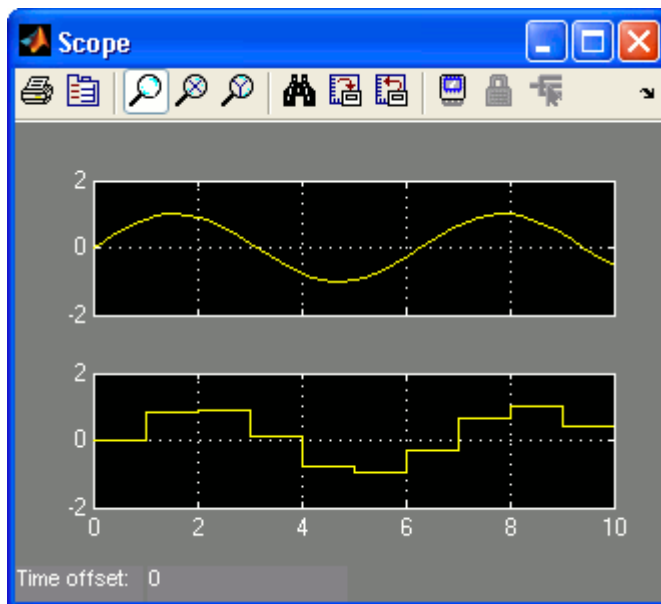
Designs in System Generator are discrete time systems. In other words, the signals and the blocks that produce them have associated sample rates. A block's sample rate determines how often the block is awoken (allowing its state to be updated). System Generator sets most sample rates automatically. A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

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

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



The Gateway In block is configured with a sample period of one second. The Gateway Out block converts the Xilinx fixed-point signal back to a double (so it can 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.

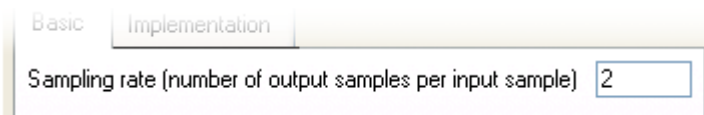


Multirate Models

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

Rate Changing Blocks

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



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

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

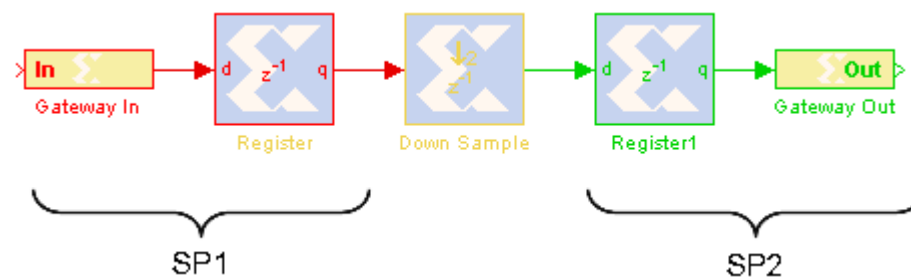


Figure 1: Simple System Generator model with 2 sample periods.

Hardware Oversampling

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

One block that can be oversampled is the DAFIR FIR filter. An oversampled DAFIR processes samples serially, thus running at a higher rate, but using less hardware.

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

Asynchronous Clocking

System Generator focuses on the design of hardware that is synchronous to a single clock. It can, under some circumstances, be used to design systems that contain more than one clock. This is possible provided the design can be partitioned into individual clock domains with the exchange of information between domains being regulated by dual port memories and FIFOs. System Generator fully supports such multi-clock designs, including the ability to simulate them in Simulink and to generate complete hardware descriptions. Details discussed in *Generating Multiple Cycle-true Islands for Distinct Clocks*. The remainder of this section focuses exclusively on the clock-synchronous aspects of System Generator. This discussion is relevant to both single-clock and multiple-clock designs.

Synchronous Clocking

When System Generator 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, System Generator generates related rates by using a single clock in conjunction with clock enables, one enable per rate. The period of each clock enable is an integer multiple of the period of the system clock.

Inside Simulink, neither clocks nor clock enables are required as explicit signals in a System Generator design. When System Generator compiles a design into hardware, it uses the sample rates in the design to deduce what clock enables are needed. To do this, it employs two user-specified values from the System Generator 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 kp units of time in Simulink takes k ticks of the system clock (hence kc nanoseconds) in hardware.

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

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

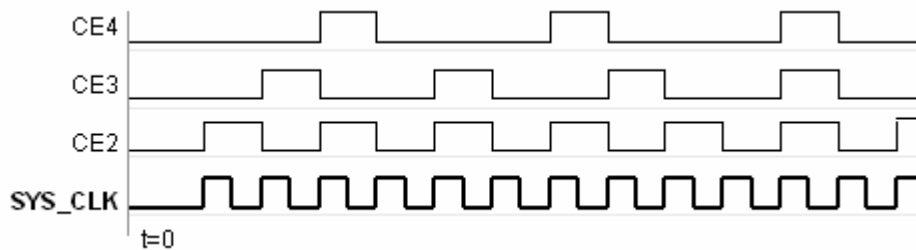


Figure 2: Example timing diagram showing clock enables for a System Generator design with three sample periods, CE2, CE3, and CE4.

Synchronization Mechanisms

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

Valid Ports

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

Indeterminate Data

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

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

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

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

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

Block Masks and Parameter Passing

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

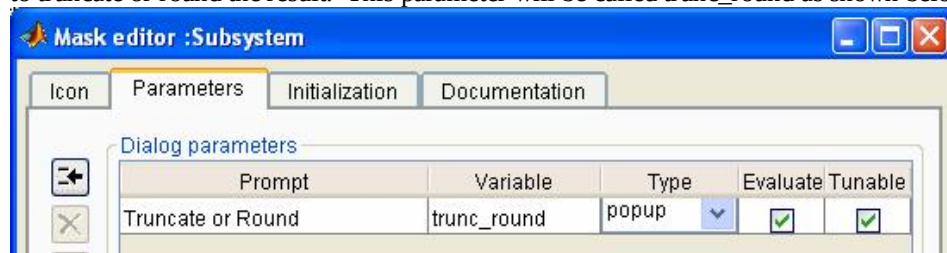
Block Masks

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

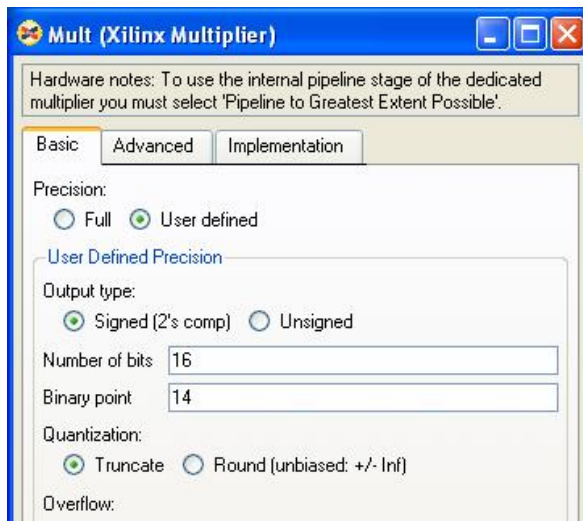
Note: It is possible for a mask to access global variables and variables in the base workspace. To access a base workspace variable, use the MATLAB `evalin` function. For more information on the MATLAB and Simulink scoping rules, refer to the *Using MATLAB* and *Using Simulink* manuals from The Mathworks,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

Parameter Passing

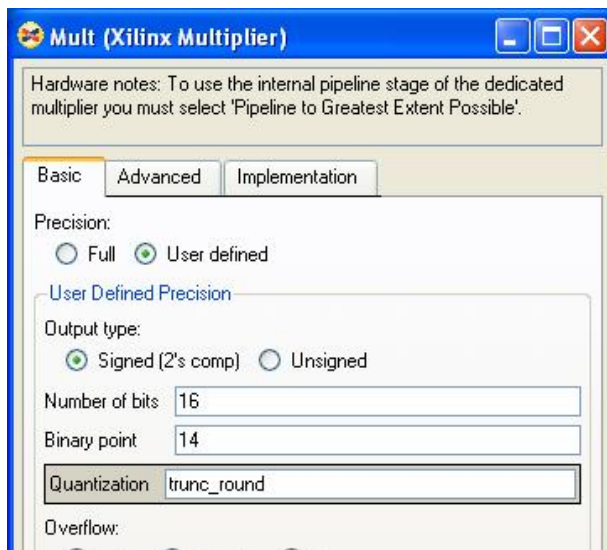
It is often desirable to pass variables to blocks inside a masked subsystem. Doing so allows the block's configuration to be determined by parameters on the enclosing subsystem. This technique can be applied to parameters on blocks in the Xilinx blockset whose values are set using a listbox, radio button,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



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



In order to use a parameter rather than the radio button selection, right click on the radio button and select: “Define With Expression”. A Matlab expression can then be used as the parameter setting. In this example, the `trunc_round` parameter from the subsystem mask can be used in both the accumulator and multiply blocks so that each block will use the same setting from the mask variable on the subsystem.



Resource Estimation

System Generator supplies tools that estimate the FPGA hardware resources needed to implement a design. Estimates include numbers of slices, lookup tables, flip-flops, block memories, embedded multipliers, I/O blocks and tristate buffers. These estimates make it easy to determine how design choices affect hardware requirements. To estimate the resources needed for a subsystem, drag a Resource Estimator block into the subsystem, double-click on the estimator, and press the **Estimate** button.

Automatic Code Generation

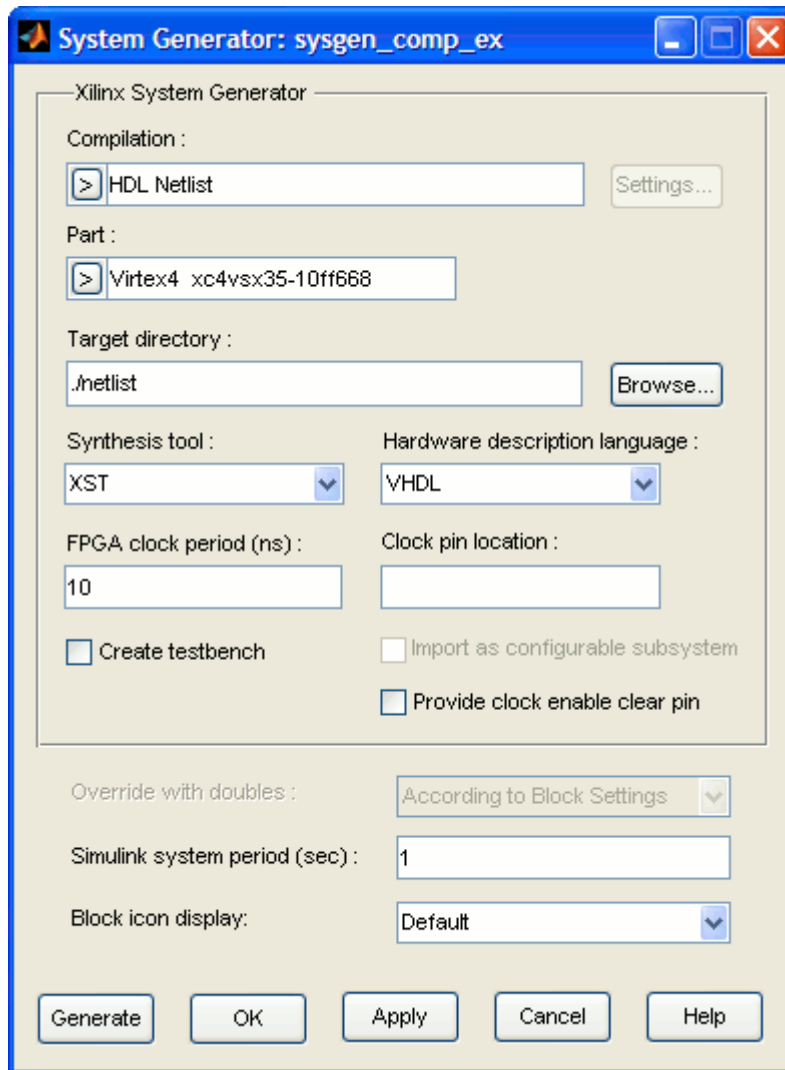
Compiling and Simulating Using the System Generator Block

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

Before a System Generator design can be simulated or translated into hardware, the design must include a System Generator block. When creating a new design, it is a good idea to add a System Generator block immediately. The System Generator block is a member of the Xilinx Blockset's Basic Elements and Tools libraries. As with all Xilinx blocks, the System Generator block can also be found in the Index library.

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

Once a System Generator block is added, it is possible to specify how code generation and simulation should be handled. The block's dialog box is shown below:



Compilation Type and the Generate Button

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

- two types of Netlists, **HDL Netlist** and **NGC Netlist**,
- **Bitstream**—produces an FPGA configuration bitstream that is ready to run in a hardware FPGA platform
- **EDK Export Tool**—for exporting to the Xilinx Embedded Development Kit
- various varieties of hardware co-simulation, and
- **Timing Analysis**—a report on the timing of the design.

HDL Netlist is the type used most often. In this case, the result is a collection of HDL and EDIF files, and a few auxiliary files that simplify downstream processing. The collection is ready to be processed by a synthesis tool (e.g., XST), and then fed to the Xilinx physical design tools (i.e., ngdbuild, map, par, and bitgen) to produce a configuration bitstream for a Xilinx FPGA. The files that are produced

are described in more detail in Compilation Results.

NGC Netlist is similar to HDL Netlist but the resulting files are NGC files instead of HDL files.

When the type is a variety of hardware co-simulation, then System Generator produces an FPGA configuration bitstream that is ready to run in a hardware FPGA platform. The particular platform depends on the variety chosen. For example, when the variety is **Hardware Co-simulation->XtremeDSP Development Kit->PCI and USB**, then the bitstream is suitable for the XtremeDSP board (available for separate purchase from Xilinx). System Generator also produces a *hardware co-simulation block* to which the bitstream is associated. This block is able to participate in Simulink simulations. It is functionally equivalent to the portion of the design from which it was derived, but is implemented by its bitstream. In a simulation, the block delivers the same results as those produced by the portion, but the results are calculated in working hardware.

Note: It is possible to customize the list of compilation types. See the section Hardware Co-Simulation Installation for details.

The remaining compilation parameters are described in the table below. Some are available only when the compilation type is **HDL Netlist**. For example, the clock pin location cannot be chosen for a hardware co-simulation compilation because it is fixed in each hardware FPGA platform.

Control	Description
Part	Defines the FPGA part to be used.
Target Directory	Defines where System Generator should write compilation results. Because System Generator and the FPGA physical design tools typically create many files, it is best to create a separate target directory, i.e., a directory other than the directory containing your Simulink model files. The directory can be an absolute path (e.g. c:\netlist) or a path relative to the directory containing the model (e.g. netlist).
Synthesis Tool	Specifies the tool to be used to synthesize the design. The possibilities are Synplicity's Synplify Pro™, Synplify, and Xilinx's XST.
Hardware Description Language	Specifies the language to be used for HDL netlist of the design. The possibilities are VHDL and Verilog.

FPGA Clock Period	Defines the period in nanoseconds of the hardware clock. The value need not be an integer. The period is passed to the Xilinx implementation tools through a constraints file, where it is used as the global PERIOD constraint. Multicycle paths are constrained to integer multiples of this value.
Clock Pin Location	Defines the pin location for the hardware clock. This information is passed to the Xilinx implementation tools through a constraints file.
Create Testbench	This instructs System Generator to create an HDL testbench. Simulating the testbench in an HDL simulator compares Simulink simulation results with ones obtained from the compiled version of the design. To construct test vectors, System Generator simulates the design in Simulink, and saves the values seen at gateways. The top HDL file for the testbench is named <name>_testbench.vhd/.v, where <name> is a name derived from the portion of the design being tested and the extension is dependent on the hardware description language.
Import as Configurable Subsystem	Tells System Generator to do two things: 1) Construct a block to which the results of compilation are associated, and 2) Construct a configurable subsystem consisting of block and the original subsystem from which the block was derived. See Configurable Subsystems and System Generator for details.

Override with Doubles

The **Override with Doubles** control is disabled in System Generator 8.1 and 8.2.

Simulink System Period

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

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

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

The period control is hierarchical; see the discussion of hierarchical controls below for details.

Block Icon Display

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

- Default—basic information about port directions are shown
- Sample rates—the sample rates of each port are shown
- Pipeline stages—the number of pipeline stages are shown
- HDL port names—the names of the ports are shown
- Input data types—the input data types for each port are shown
- Output data types—output data types for each port are shown

Hierarchical Controls

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

Compilation Results

In this section we discuss the low level files System Generator produces when **HDL Netlist** is selected on the System Generator block and **Generate** is pushed. The files consist of HDL, NGC and EDIF that implement the design. In addition, System Generator produces auxiliary files that simplify downstream processing, e.g., bringing the design into Project Navigator, simulating using ModelSim, and synthesizing using various synthesis tools. All files are written to the target directory specified on the System Generator block. If no testbench is requested, then the key files produced by System Generator are the following:

File Name or Type	Description
<design>.vhd/.v	This contains most of the HDL for the design.
<design>_cw.vhd/.v	This is a HDL wrapper for <design>_files.vhd/.v. It drives clocks and clock enables.
.edn and .ngc files	Besides writing HDL, System Generator runs CORE Generator (coregen) to implement portions of the design. Coregen writes EDIF files whose names typically look something like multiplier_virtex2_6_0_83438798287b830b.edn. Other required files may be supplied as .ngc files.
globals	This file consists of key/value pairs that describe the design. The file is organized as a Perl hash table so that the keys and values can be made available to Perl scripts using Perl evals.
<design>_cw.xcf (or .ncf)	This contains timing and port location constraints. These are used by the Xilinx synthesis tool XST and the Xilinx implementation tools. If the synthesis tool is set to something other than XST, then the suffix is changed to .ncf.
<design>_cw.ise	This allows the HDL and EDIF to be brought into the Xilinx project management tool Project Navigator.
hdlFiles	This contains the full list of HDL files written by System Generator. The files are listed in the usual HDL dependency order.
synplify_<design>.prj, or xst_<design>.prj	These files allow the design to be compiled by the synthesis tool you specified
vcom.do	This script can be used in ModelSim to compile the HDL for a behavioral simulation of the design.

If a testbench is requested, then, in addition to the above, System Generator produces files that allow simulation results to be compared. The comparisons are between Simulink simulation results and corresponding results from ModelSim. The additional files are the following.

File Name or Type	Description

Various .dat files	These contain the simulation results from Simulink.
<design>_tb.vhd/.v	This is a testbench that wraps the design. When simulated in ModelSim, this testbench compares simulation results from Simulink against those produced by ModelSim.
vsim.do	This script can be used in ModelSim to run a testbench simulation.
pn_behavioral.do, pn_postmap.do, pn_postpar.do, pn_posttranslate.do	These files allow various ModelSim simulations to be started inside Project Navigator.

Using the System Generator Constraints File

When a design is compiled, System Generator produces a *constraints file* that tells 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 file format depends on the synthesis tool that is specified in the System Generator block. When XST is selected, the file is written in the “xcf” format; for Synplify and Synplify Pro, the “ncf” format is used. The file name ends with .xcf or .ncf, as appropriate.

System Clock Period

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

Multicycle Path Constraints

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

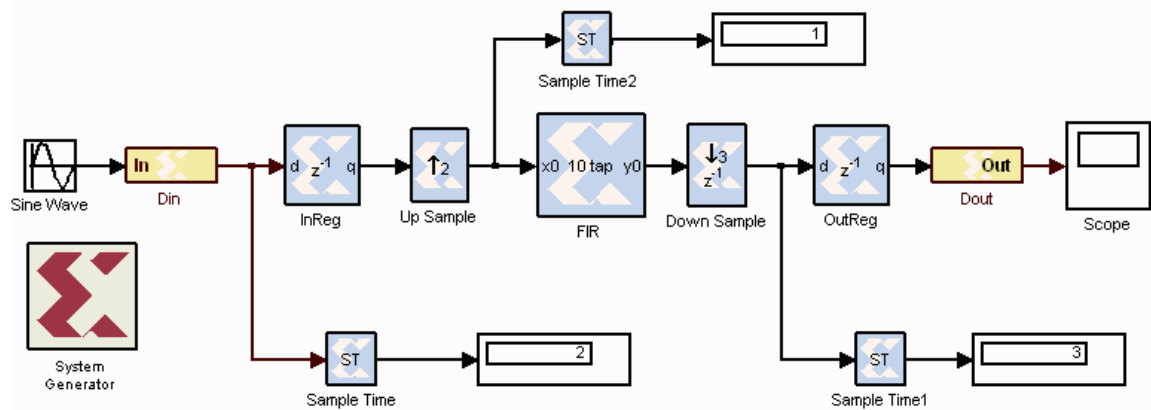
IOB Timing and Placement Constraints

When translated into hardware, System Generator’s Gateway In and Gateway Out blocks become input and output ports. The locations of these ports and the speeds at which they must operate can be entered in the Gateway In and Out parameter dialog boxes.

See the descriptions of the Gateway In block and the Gateway Out block in the Data Types library section for more information. Port location and speed are specified in the constraints file by IOB timing.

Constraints Example

Below we show a small multirate design and the constraints System Generator produces for it.



The up sampler doubles the rate, and the down sampler divides the rate by three. Assume the system clock period is 10 ns. Then the clock periods are 10 ns for the FIR, 20 ns for the input register, and 30 ns for the output register. Below we describe the constraints that convey this information.

The lines that indicate the system clock period is 10 ns are the following:

```
# Global period constraint
NET "clk" TNM_NET = "clk_392b7670";
TIMESPEC "TS_clk_392b7670" = PERIOD "clk_392b7670" 10.0 ns HIGH
50 %;
```

To build timing constraints, the blocks in the design are partitioned into timing groups. Two blocks are in the same timing group if and only if they run at the same sample rate. In this design there are three timing groups, corresponding to the three rates. The nature of constraints dictates that no name is needed for the fastest group. The remaining groups are named *ce_2_392b7670_group* and *ce_3_392b7670_group*; they correspond to periods 20 ns and 30 ns respectively.

The FIR runs at the system (i.e., fastest) rate and therefore is constrained using the global period constraint shown above. The logic used to generate clocks always runs at the system rate and is also constrained to the system rate.

The *ce_2_392b7670_group* consists of the blocks that operate at half the system rate, i.e., the input register and the up sampler. Every block in the group is driven by the clock enable net named *ce2_sysgen*. The constraints that define the group are the following:

```
# ce_2_392b7670_group and inner group constraint
Net "ce_2_sg_x0*" TNM_NET = "ce_2_392b7670_group";
TIMESPEC "TS_ce_2_392b7670_group_to_ce_2_392b7670_group" = FROM
"ce_2_392b7670_group" TO "ce_2_392b7670_group" 20.0 ns;
```

Note: A wildcard character is added to the net name to constrain any additional copies of this net that may be generated when clock enable logic is replicated. The maximum fanout of a clock enable net can be controlled in the synthesis tool.

The *ce_3_392b7670_group* operates at one third the system rate. It contains the down sampler and the output register, and is defined in a similar manner to the *ce2_group*.

```
# ce_3_392b7670_group and inner group constraint
Net "ce_3_sg_x0*" TNM_NET = "ce_3_392b7670_group";
TIMESPEC "TS_ce_3_392b7670_group_to_ce_3_392b7670_group" = FROM
"ce_3_392b7670_group" TO "ce_3_392b7670_group" 30.0 ns;
```

Group to group constraints establish relative speeds. Here are the constraints that relate the speeds of *ce_2_392b7670_group* and *ce_3_392b7670_group*:

```
# Group-to-group constraints
TIMESPEC "TS_ce_2_392b7670_group_to_ce_3_392b7670_group" = FROM
"ce_2_392b7670_group" TO "ce_3_392b7670_group" 20.0 ns;
TIMESPEC "TS_ce_3_392b7670_group_to_ce_2_392b7670_group" = FROM
"ce_3_392b7670_group" TO "ce_2_392b7670_group" 20.0 ns;
```

Port timing requirements can be set in the parameter dialog boxes for gateways. These requirements are translated into port constraints such as those shown below. In this example, the 3-bit *din* input is constrained to operate at its gateway's sample rate (corresponding to a period of 20 ns). The "FAST" attributes indicate the ports should be implemented using hardware that reduces delay. The reduction comes at a cost of increased noise and power consumption.

```
# Offset in constraints
NET "din(0)" OFFSET = IN : 20.0 : BEFORE "clk";
NET "din(0)" FAST;
NET "din(1)" OFFSET = IN : 20.0 : BEFORE "clk";
NET "din(1)" FAST;
NET "din(2)" OFFSET = IN : 20.0 : BEFORE "clk";
NET "din(2)" FAST;
```

Selecting **Specify IOB Location Constraints** for a gateway allows port locations to be specified. The locations must be entered as a cell array of strings in the box labeled **IOB Pad Locations**. Locations are package-specific; in this example a Virtex-E 2000 in a FG680 package is used. The location constraints for the *din* bus are provided in the dialog box as "{ 'D35', 'B36', 'C35' }". This is translated into constraints in the .xcf (or .ncf) file in the following way:

```
# Loc constraints
NET "din(2)" LOC = "D35";
NET "din(1)" LOC = "B36";
NET "din(0)" LOC = "C35";
```

Clock Handling in HDL

In this section we describe how System Generator handles hardware clocks in the HDL it generates. Assume the design is named *<design>*, and *<design>* is an acceptable HDL identifier. When System Generator compiles the design, it writes a collection of HDL entities or modules, the topmost of which is named *<design>*, and is stored in a file named *<design>.vhd/.v*.

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

Of course, there must be a way to generate these clocks and clock enables. System Generator produces a separate *clock wrapper* (written to a file named *<design>_cw.vhd/.v*) to do this. This wrapper is external to the files described above. The idea is to make the HDL flexible. In some applications,

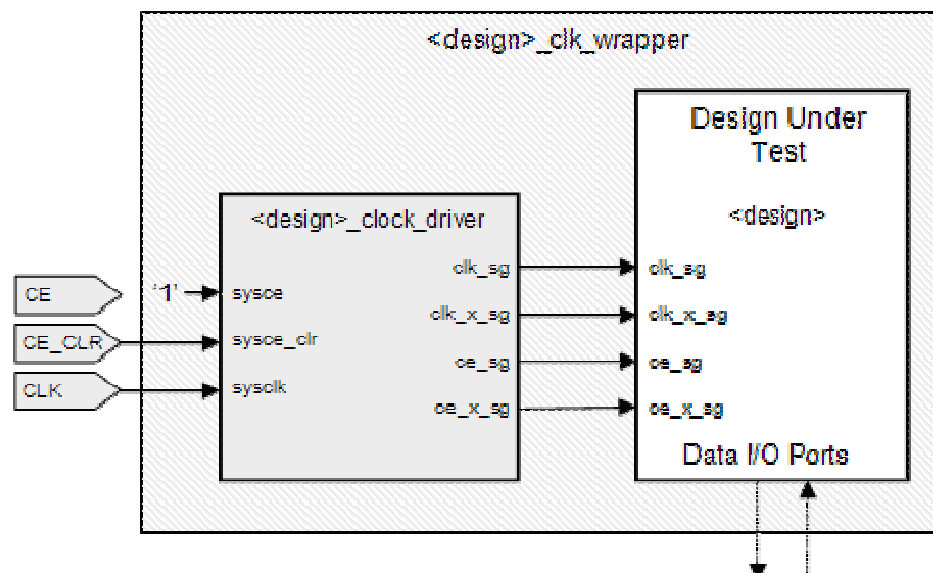
the files described above are added to a larger design, but the clock wrapper is omitted. In this case, the user is responsible for generating clocks and clock enables, but a finer degree of control is obtained. If, on the other hand, the clock wrapper is suitable for the application, then the designer includes it. As an additional convenience, System Generator generates a *DCM wrapper* (written to a file named `<design>_dw_vhd/_v`) that encloses the clock wrapper. The DCM wrapper deskews the hardware FPGA clock. When incorporating System Generator HDL into a larger design, each of the following is possible:

- Use the HDL that System Generator writes, but exclude the clock and DCM wrappers;
- Use the HDL that System Generator writes, and use the clock wrapper, but exclude the DCM wrapper.
- Use the HDL that System Generator writes, and use both the clock and DCM wrappers.

If you want to use the DCM wrapper in your work, rename the file by replacing the final `_vhd` or `_v` with `.vhd` or `.v`.

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

The clock wrapper consists of two components: one for the design itself, and one *clock driver component* that generate clocks and clock enables. The clock driver is contained in a file named `<design>_cw.vhd/.v`. The logic within the `<design>_cw` generates the `ce_x` signals. The optional `ce_clr` port would be generated if the design was generated by selecting “Provide clock enable clear pin” on the System Generator block. The ports that are not clocks or clock enables are passed through to the exterior of the clock wrapper. Schematically, the clock wrapper looks like the diagram below.



Note: The clock wrapper exposes a port named *ce*. The port does nothing except to serve as a companion to the *clk* port on the wrapper. The reason for having the port is to allow the clock wrapper to be used as a black box in System Generator designs.

Core Caching

System Generator uses *cores* produced by Xilinx CORE Generator (coregen) to implement parts of designs. Generating cores can be expensive, so System Generator caches previously generated ones. Before coregen is called, System Generator looks in the cache, and if the core has already been generated, System Generator reuses it.

By default, the cache is the directory <sysgen>/xilinx/sysgen/core_cache, where <sysgen> denotes the directory in which System Generator is installed. By default System Generator caches no more than 2,000 cores. When the limit is reached, System Generator deletes cached cores to make room for new ones.

Note: Environment variables can be used to change the location of the cache and the cache size limit. The variables are described below.

Environment Variable	Description
SGCORECACHE	Location to store cached files. Setting this variable to a string of blanks instructs System Generator not to cache cores.
SGCORECACHELIMIT	Maximum number of cores to cache.

HDL Testbench

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

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

1. A file named <design>_tb.vhd/.v that contains a testbench HDL entity;
2. Various .dat files that contain test vectors for use in an HDL testbench simulation.
3. Scripts vcom.do and vsim.do that can be used in ModelSim to compile and simulate the testbench, comparing Simulink test vectors against those produced in HDL.

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

Importing Compilation Results into Designs

It can be useful to instruct System Generator to do the following:

1. Compile a subsystem into HDL and EDIF;
2. Associate the files with a black box;
3. Insert the black box and the subsystem from which it was derived into a Simulink configurable subsystem;
4. Substitute the configurable subsystem for the original subsystem.
5. Tell System Generator that for subsequent compilations, the black box files should be used.

This procedure is known as *incremental netlisting*. The advantage to incremental netlisting is that subsequent compilations are faster. (Compiling a black box requires nothing more than simply copying its files.) This is worthwhile when the design is large, and changes are confined to a small portion. There are, however, a few drawbacks:

- Incremental netlisting can make the design somewhat harder to work on.
- The black box produced by incremental netlisting can become “stale”, i.e., it no longer faithfully reflects the subsystem from which it was derived. In this case, it must be regenerated by the user.
- The black box produced by using VHDL as a hardware description language would not work in a Verilog incremental netlisting flow and vice versa.

How to Netlist Incrementally

To instruct System Generator to netlist a subsystem incrementally, do the following:

1. Drag a System Generator block into the subsystem, and open the GUI for the System Generator block.
2. Select **HDL Netlist** under **Compilation**, turn on the checkbox labeled **Import as Configurable Subsystem**, and press the **Generate** button.

System Generator responds by translating the subsystem into HDL and EDIF, and attaching the files to a black box. It then replaces the subsystem with a configurable subsystem consisting of the original subsystem and the black box. Inside the original subsystem is a *configurable subsystem manager* block.

1. Double click on the configurable subsystem to open the original subsystem.
2. Open the configurable subsystem manager block, and select the black box as the generation choice.



Configurable Subsystem
Manager

The effect is that when simulating, the original subsystem is used, but when translating, the black box HDL is reused.

Fine Points

- Configurable subsystems are an ordinary part of Simulink, and manipulating one built by System Generator is no different than manipulating any other. System Generator documentation concerning configurable subsystems is at Configurable Subsystems and System Generator. The Simulink documentation for configurable subsystems can be found at <http://www.mathworks.com/access/helpdesk/help/toolbox/simulink/slref/configurablesubsystem.shtml>.
- When a configurable subsystem is used in a System Generator design, there are two choices to be made: which underlying block is to be used for simulation, and which is to be used for translation. For simulation, System Generator uses the configurable subsystem's "block choice", and for translation it uses the block specified in the configurable subsystem manager block.
- The black box for an incrementally netlisted subsystem becomes "stale" if the subsystem changes after the black box has been generated. In this case, you must replace the stale black box with a freshly generated one. The steps are the following:
 1. Open the library for the configurable subsystem. To do this, right click on the configurable subsystem block, and select **Library link options->Go to link source**.
 2. Unlock the library by selecting **Edit->Unlock library**.
 3. Open the GUI for the template, and turn off the checkbox corresponding to the stale black box.
 4. Delete the stale black box from the library, and close the library.
 5. Double click on the configurable subsystem to open the original subsystem, and open the original subsystem's System Generator block dialog box.
 6. Select **HDL Netlist** under **Compilation**, turn on the checkbox labeled **Import as Configurable Subsystem**, and press the **Generate** button.
 7. Open the configurable subsystem manager block inside the original subsystem, and select the black box as the generation choice.

Note: A subsystem can change in an obvious way (e.g., by having a block added), or a subtle one (e.g., having the type of an incoming signal change).

- If your design contains several incrementally netlisted subsystems, you must specify a different target directory for each. If several share the same directory, files will be overwritten.
- Occasionally, you may want to undo the effects of incremental netlisting. The steps are the following:
 1. Open the library for the configurable subsystem. To do this, right click on the configurable subsystem block, and select **Library link options->Go to link source**.
 2. Copy the block labeled **Original Subsystem**.
 3. Delete the configurable subsystem, and paste the original subsystem in its place.
 4. Right click on the original subsystem, and select **Library link options->Disable link**.
 5. Double click to open the original subsystem, and then delete the configurable subsystem manager block.

- The black box that results when a subsystem is netlisted incrementally is an ordinary black box. For details concerning black boxes, see Importing HDL Modules into System Generator.

Compiling MATLAB into an FPGA

Xilinx provides two methods for compiling MATLAB into FPGAs. For MATLAB that represents complete DSP algorithms users should consider the use of AccelDSP™ Synthesis. AccelDSP Synthesis is a high level DSP design environment that takes floating-point MATLAB as input and generates RTL or a custom IP block for use within System Generator. AccelDSP provides design automated design exploration for quantization, DSP building block macro architectures, loop unrolling and pipeline insertion. More information about AccelDSP is available at:
http://www.xilinx.com/ise/dsp_design_prod/acceldsp

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

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

- Example 1 shows how to implement a function that returns the maximum value of its inputs;
- Example 2 shows how to implement simple arithmetic operations;
- Example 3 shows how to build a complex multiplier with latency;
- Example 4 shows how to implement shift operations;
- Example 5 shows how to pass parameters into a MCode block;
- Example 6 shows how to implement optional input ports on an MCode block;
- Example 7 shows how to implement a finite state machine;
- Example 8 shows how to build a parameterizable accumulator;
- Example 9 shows how to model FIR blocks and how to do system verification;
- Example 10 shows how to model a RPN calculator – a stack machine;
- Example 11 shows how to use disp function to print variable values.

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

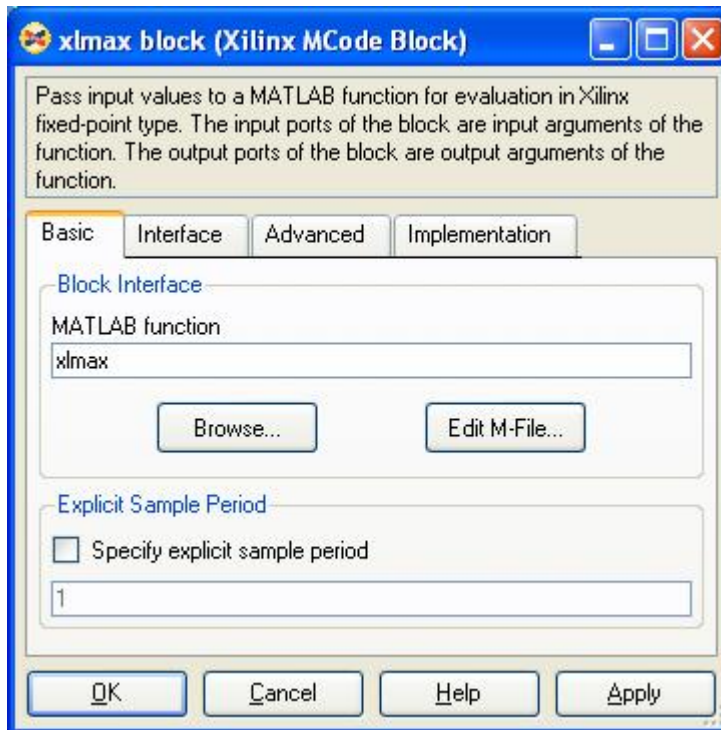
Simple Selector

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

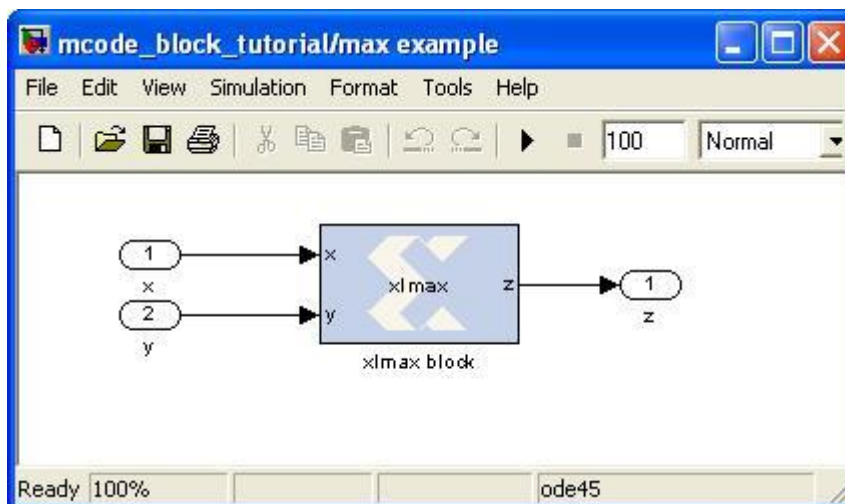
```
function z = xlmax(x, y)
    if x > y
        z = x;
    else
        z = y;
    end
```

The `xlmax.m` file should be either saved in the same directory of the model file or should be in the

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



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



Simple Arithmetic Operations

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


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

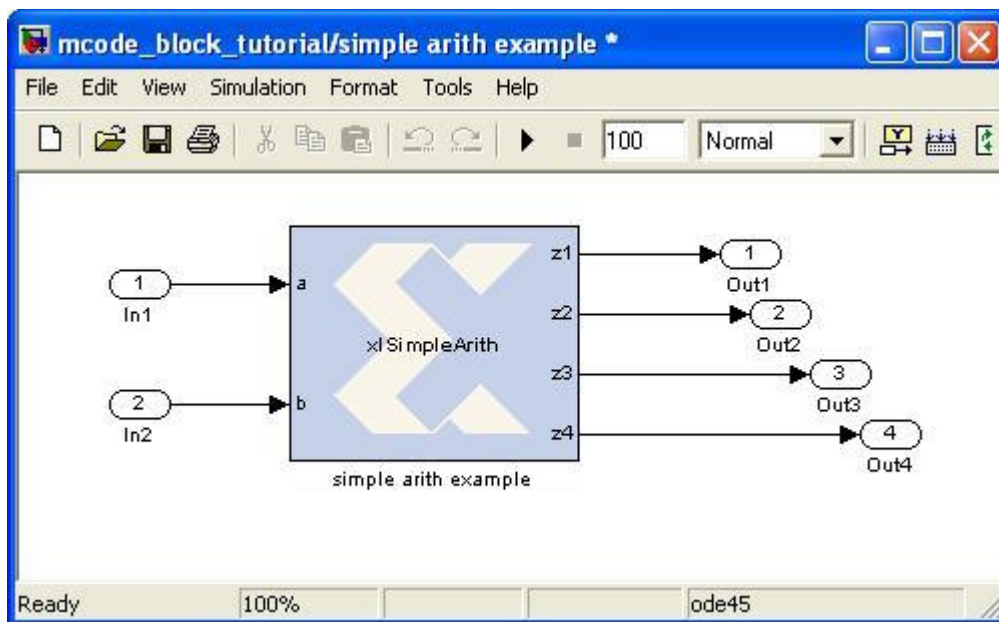
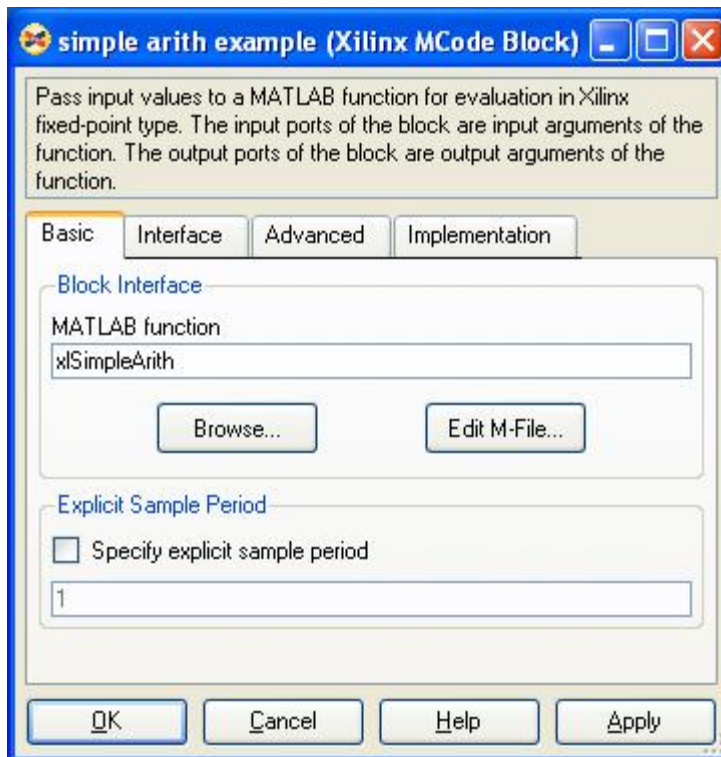
z3 = z1 - z2;
% convert z3 to Fix_12_8 with saturation for overflow
z3 = xfix({xlSigned, 12, 8, xlTruncate, xlSaturate}, z3);

% z4 is true if both inputs are positive
z4 = a>const1 & b>-1;
```

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

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

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



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

```

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

```

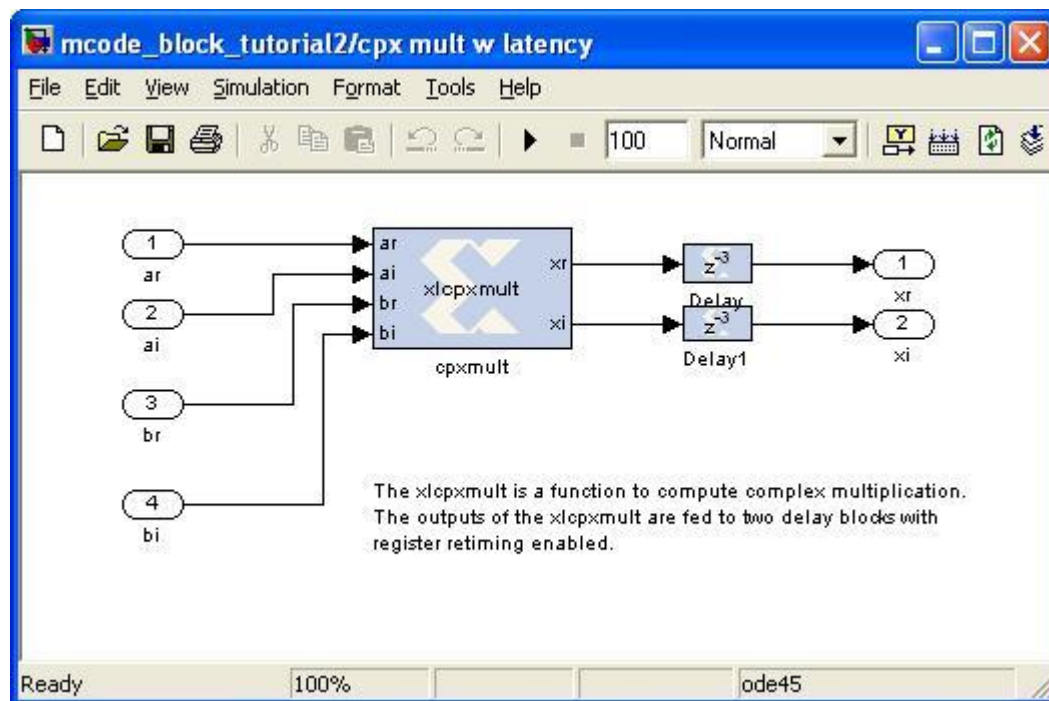
Notice that the two integer arguments (2 and 3) are converted to fixed-point numbers automatically. If you have a floating-point number as an argument, an `xfix` call is required.

Complex Multiplier with Latency

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

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

The following diagram shows the sub-system:



Two delay blocks are added after the MCode block. By selecting the Enable Register Retiming parameter of the Delay blocks, System Generator uses registers to implement the Delay blocks so the synthesis tools are able to perform appropriate register retiming to achieve high performance.

Shift Operations

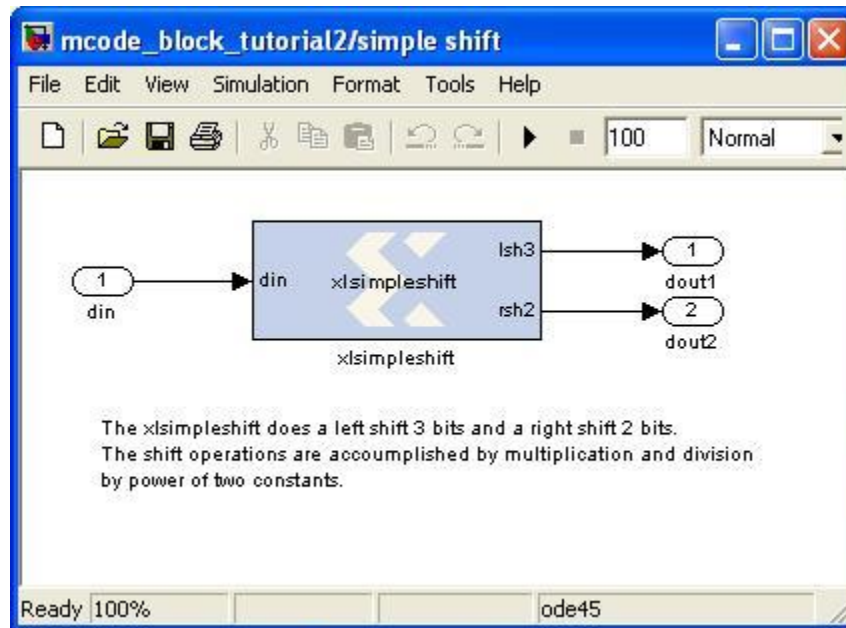
This example shows how to implement bit-shift operations using the MCode block. Shift operations are accomplished with multiplication and division by powers of two. For example, multiplying by 4 is equivalent to a 2-bit left-shift, and dividing by 8 is equivalent to a 3-bit right-shift. Shift operations 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 `xlshiftright.m` file which specifies one left-shift and one right-shift:

```
function [lsh3, rsh2] = xlshiftright(din)
```

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

The following diagram shows the sub-system after compilation:



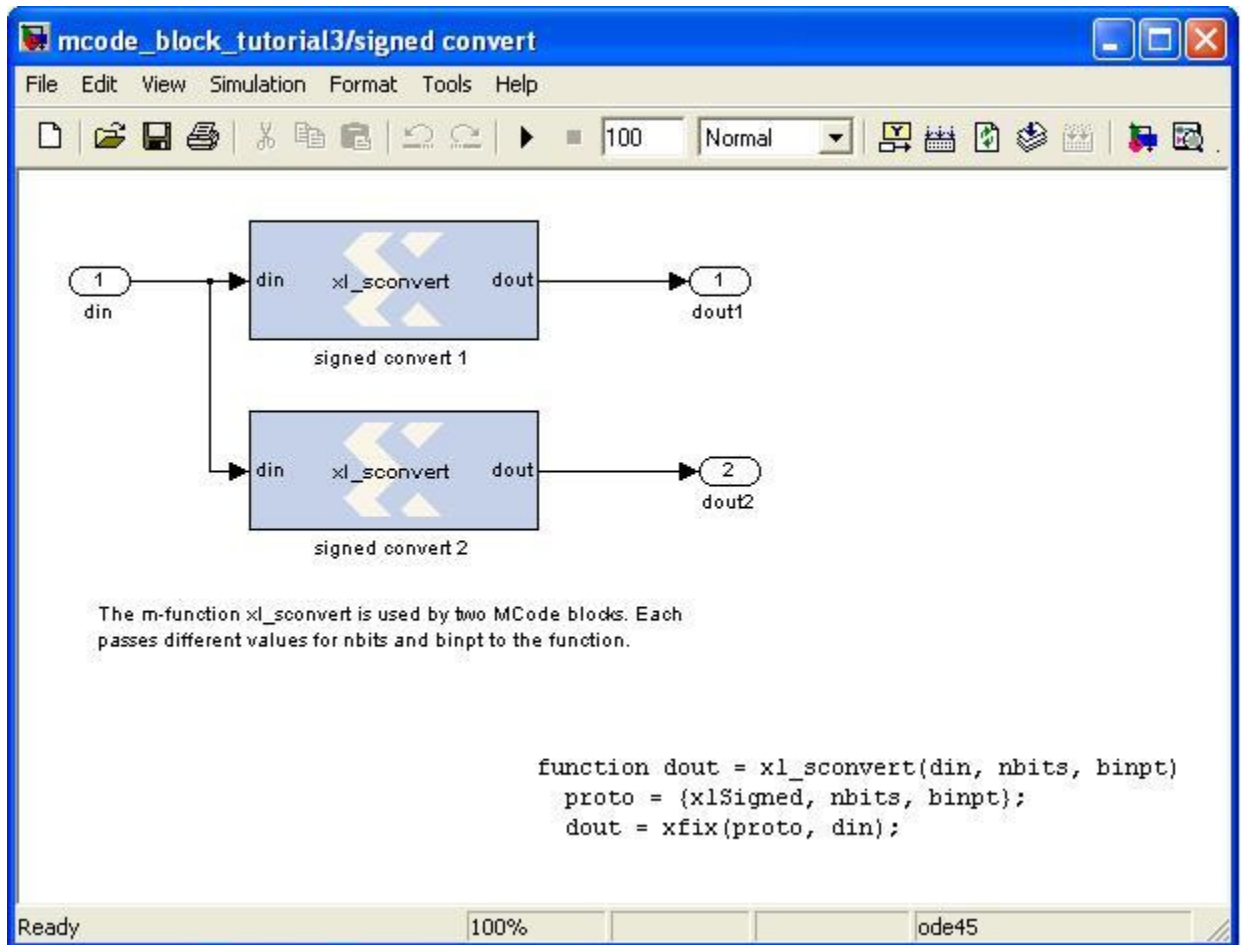
Passing Parameters into the MCode Block

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

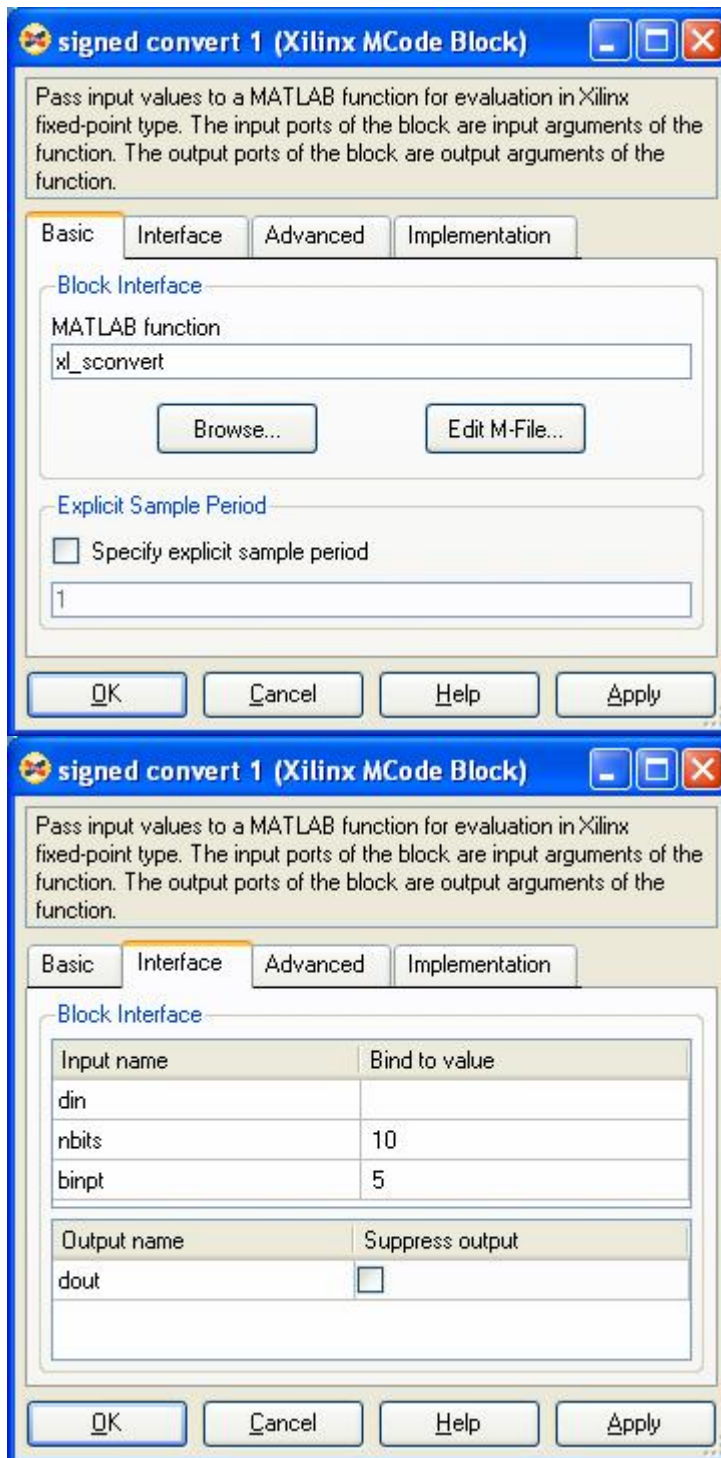
The following m-code defines an m-function `xl_sconvert` is contained in file `xl_sconvert.m`:

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

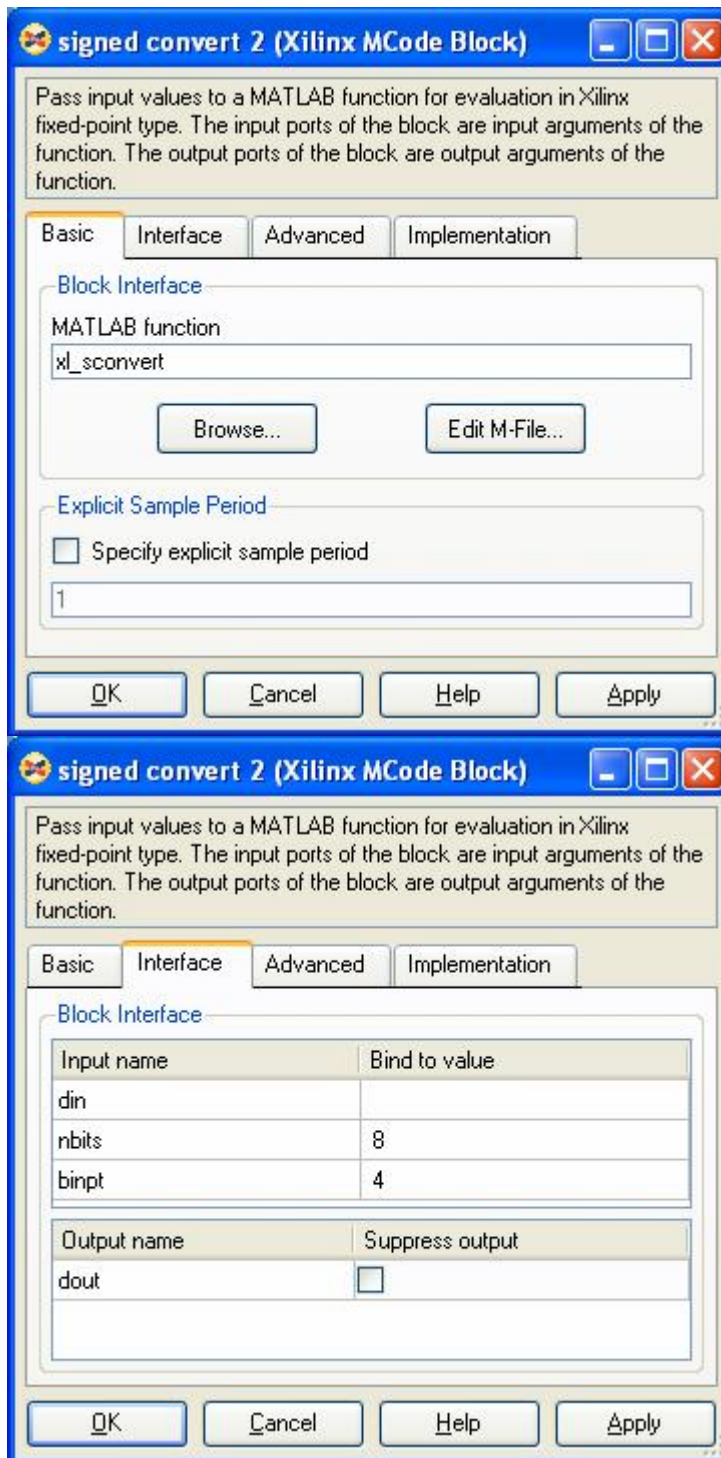
The following diagram shows a subsystem containing two MCode blocks that use m-function `xl_sconvert`. The arguments `nbits` and `binpt` of the m-function are specified differently for each block by passing different parameters to the MCode blocks. The parameters passed to the MCode block labeled `signed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



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



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



The above interface window sets the m-function argument `nbits` to be 7 and `binpt` to be 4.

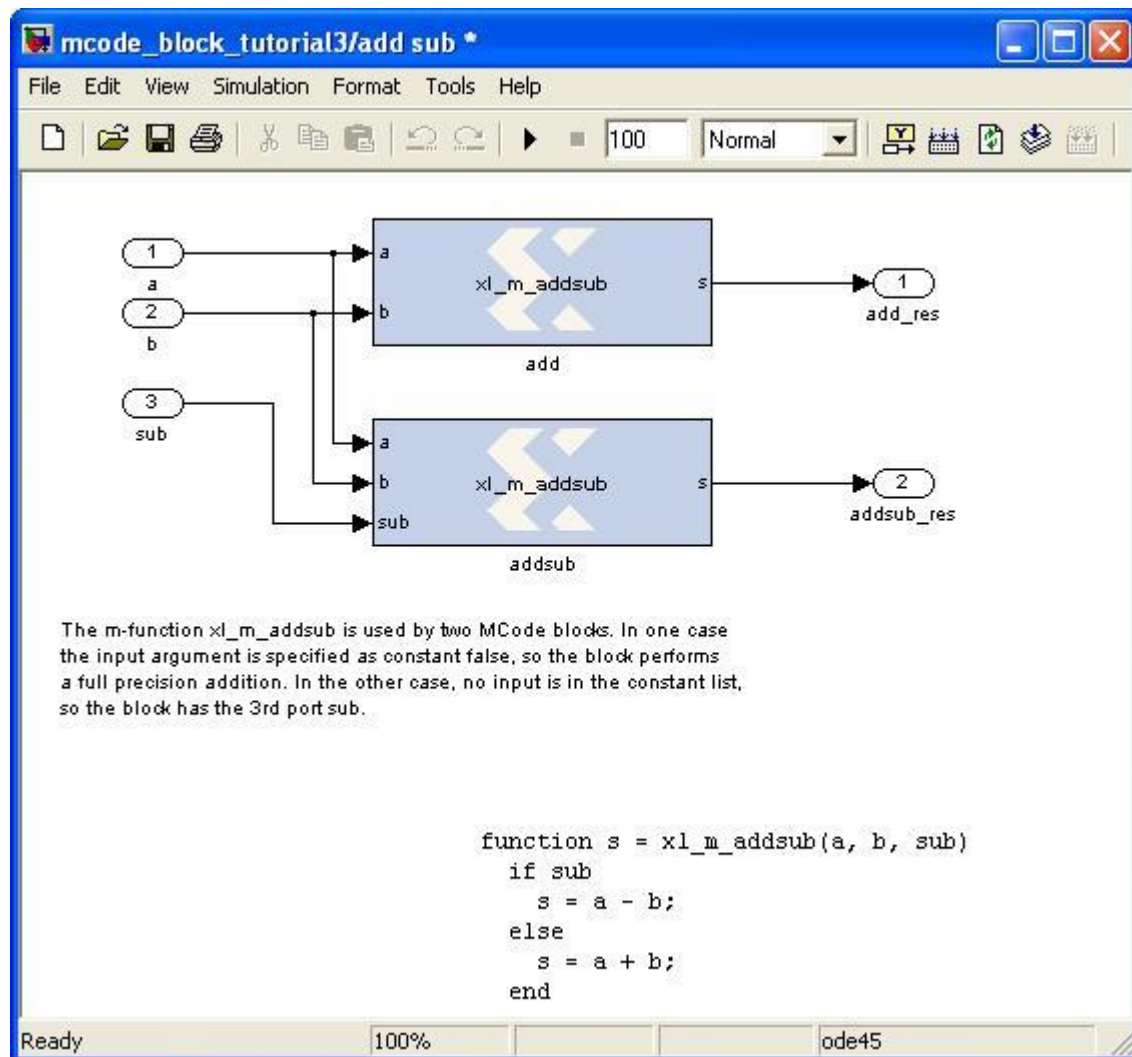
Optional Input Ports

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

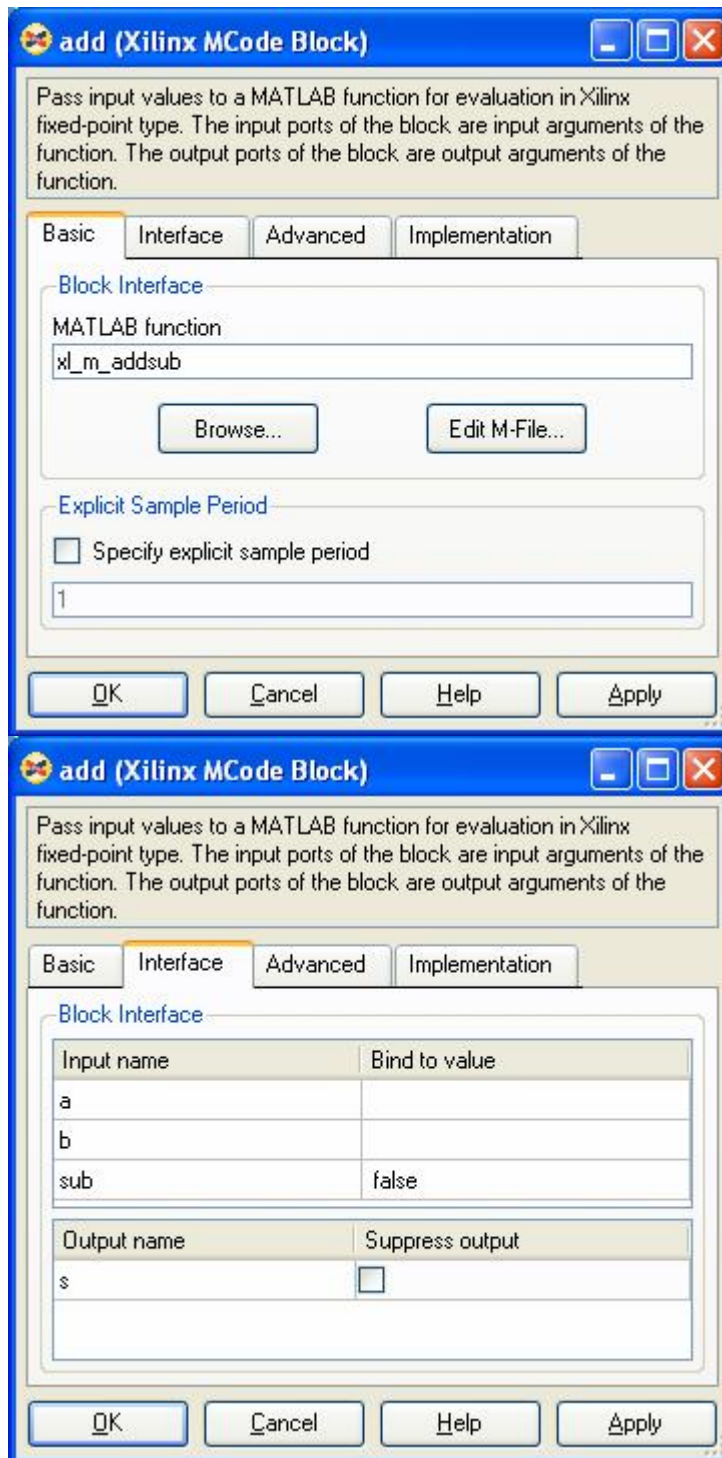
The following m-code, which defines m-function `xl_m_addsub` is contained in file `xl_m_addsub.m`:

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

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



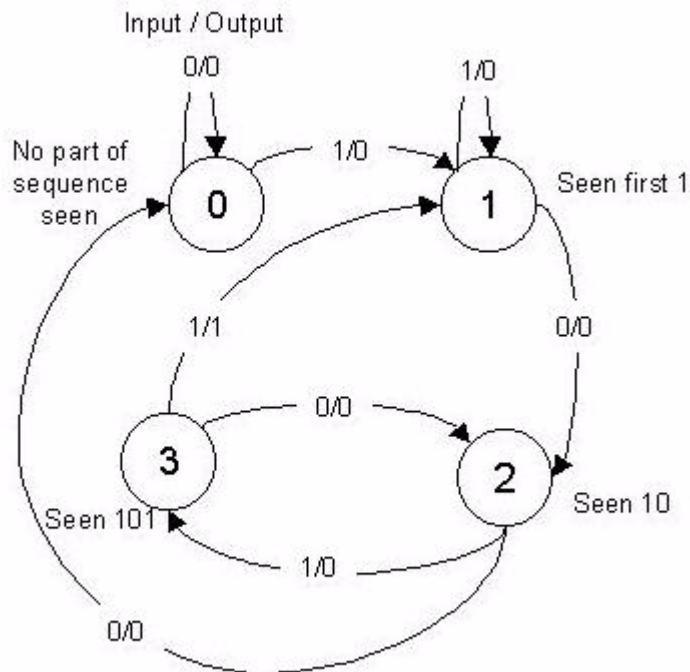
The Block Interface Editor of the MCode block labeled `add` is shown in below.



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

Finite State Machines

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



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

```

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

seen_none = 0; % initial state, if input is 1, switch to seen_1
seen_1 = 1;    % first 1 has been seen, if input is 0, switch
               % to seen_0
seen_0 = 0;    % 0 has been detected, if input is 1, switch to
               % seen_1
seen_10 = 2;   % 10 has been detected, if input is 1, switch to
               % seen_101
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;

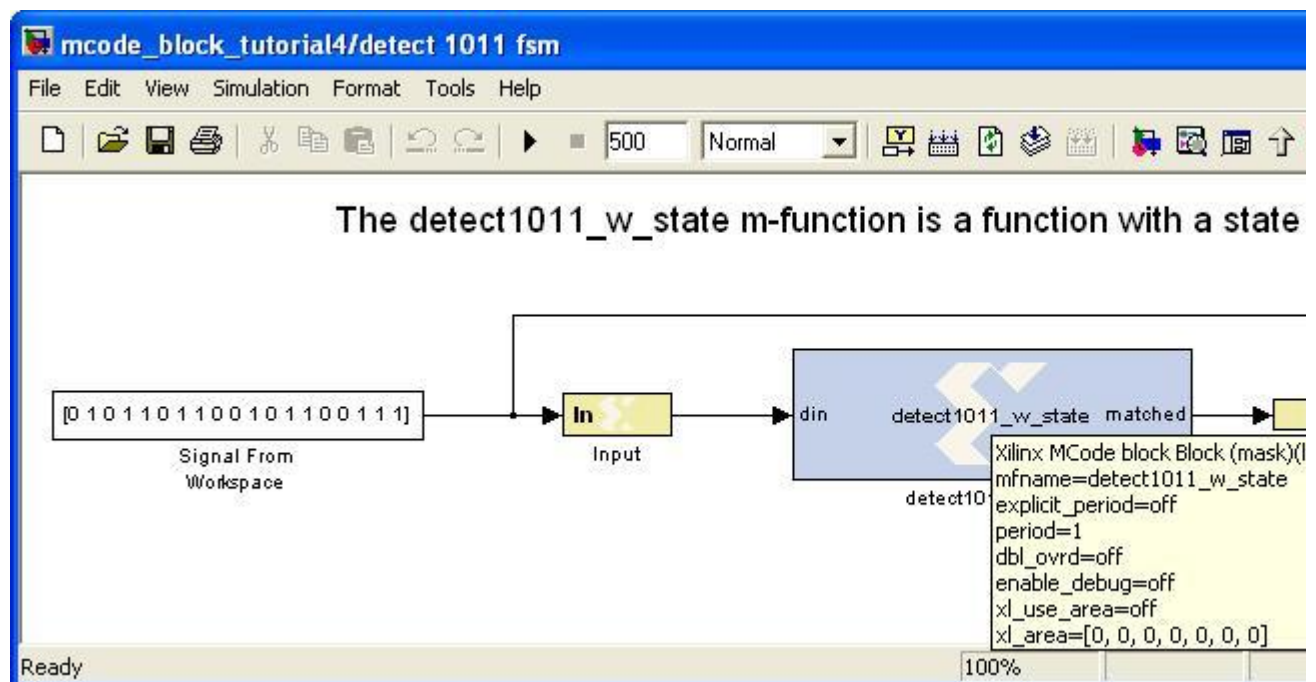
```

```

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
end

```

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



Parameterizable Accumulator

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

```

function q = xl_accum(b, rst, load, en, nbits, ov, op,
    feed_back_down_scale)
% q = xl_accum(b, rst, nbits, ov, op, feed_back_down_scale) is
% equivalent to our Accumulator block.

```

```

binpt = xl_binpt(b);
init = 0;

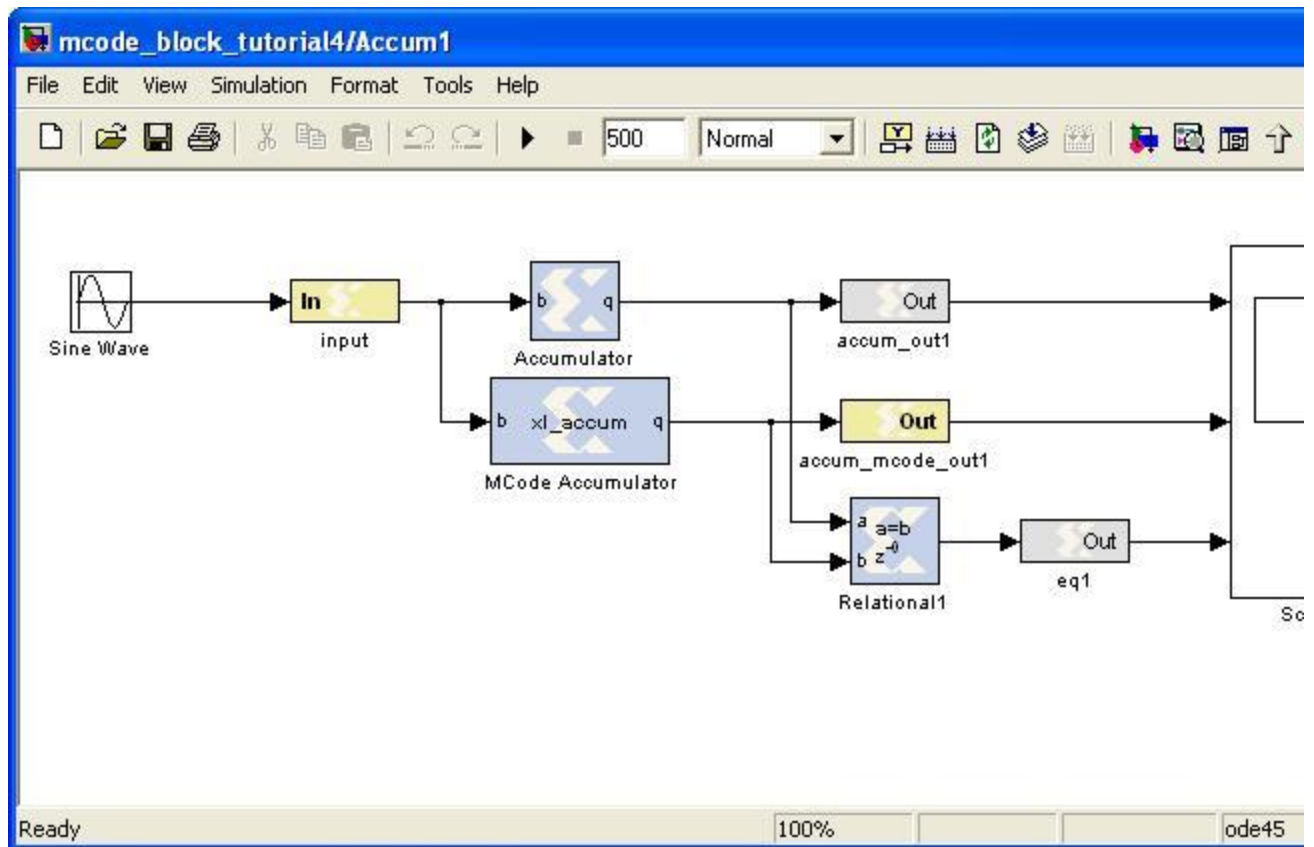
precision = {xlSigned, nbits, binpt, xlTruncate, ov};
persistent s, s = xl_state(init, precision);

q = s;

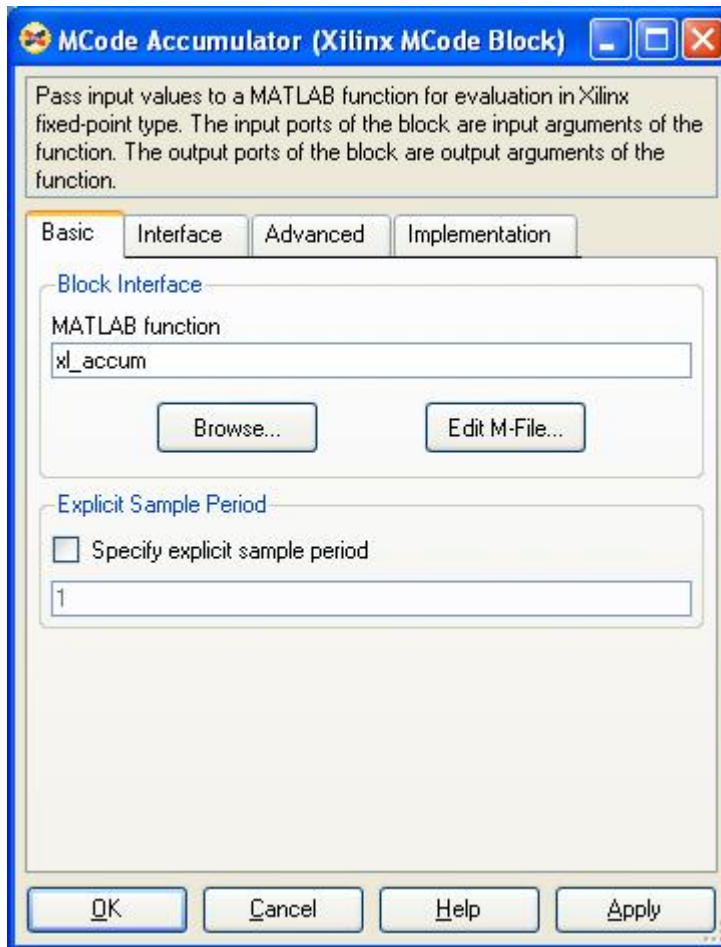
if rst
    if load
        % reset from the input port
        s = b;
    else
        % reset from zero
        s = init;
    end
else
    if ~en
    else
        % if enabled, update the state
        if op==0
            s = s/feed_back_down_scale + b;
        else
            s = s/feed_back_down_scale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

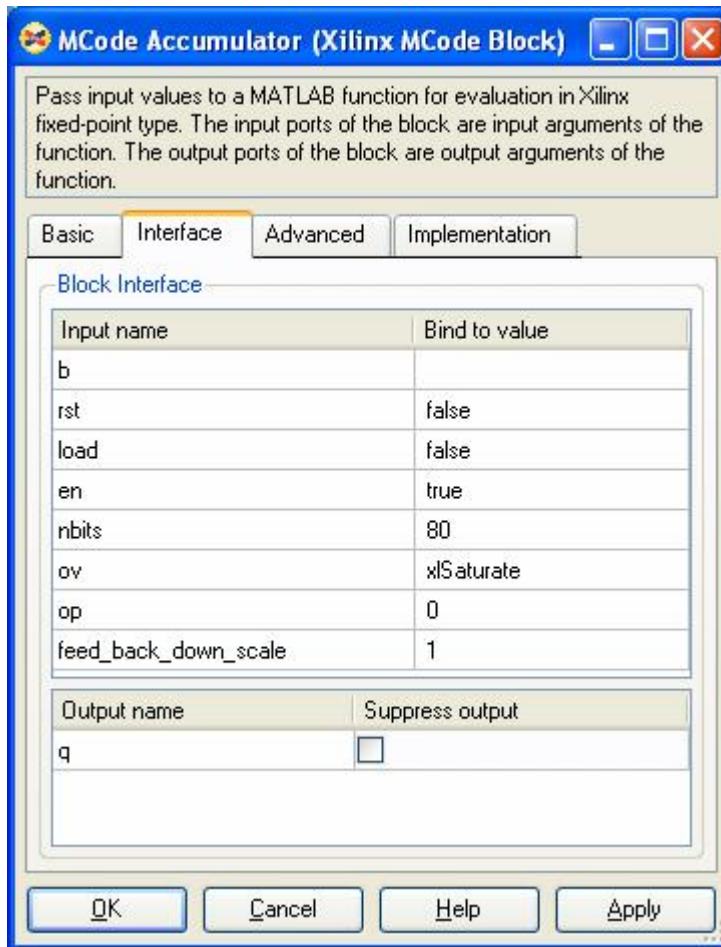
```

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



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

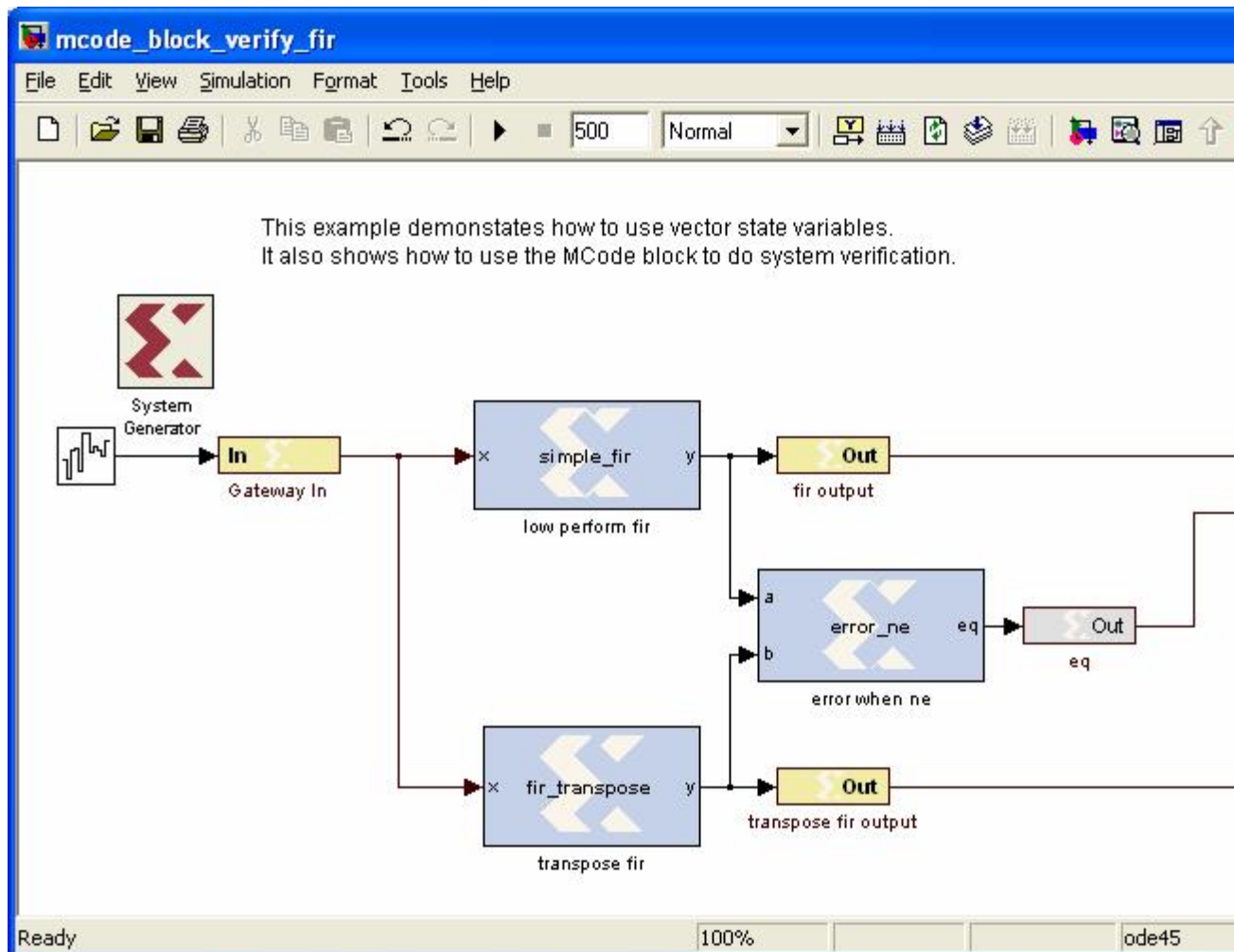




The example contains two additional accumulator subsystems with MCode blocks using the same m-function, but different parameter settings to accomplish different accumulator implementations.

FIR Example and System Verification

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



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

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

    coefs_xfix = xfix(coef_prec, coefs);

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

    sum = x * coef_vec(0);

    for idx = 1:len-1
        sum = sum + x_line(idx-1) * coef_vec(idx);
        sum = xfix(out_prec, sum);
    end

    y = p.back;
    p.push_front_pop_back(sum);
```



```

x_line.push_front_pop_back(x);

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

    coefs_xfix = xfix(coef_prec, coefs);

    persistent coef_vec, coef_vec = xl_state(coefs_xfix, coef_prec);
    persistent reg_line, reg_line = xl_state(zeros(1, len), out_prec);

    if lat <= 0
        error('latency must be at least 1');
    end

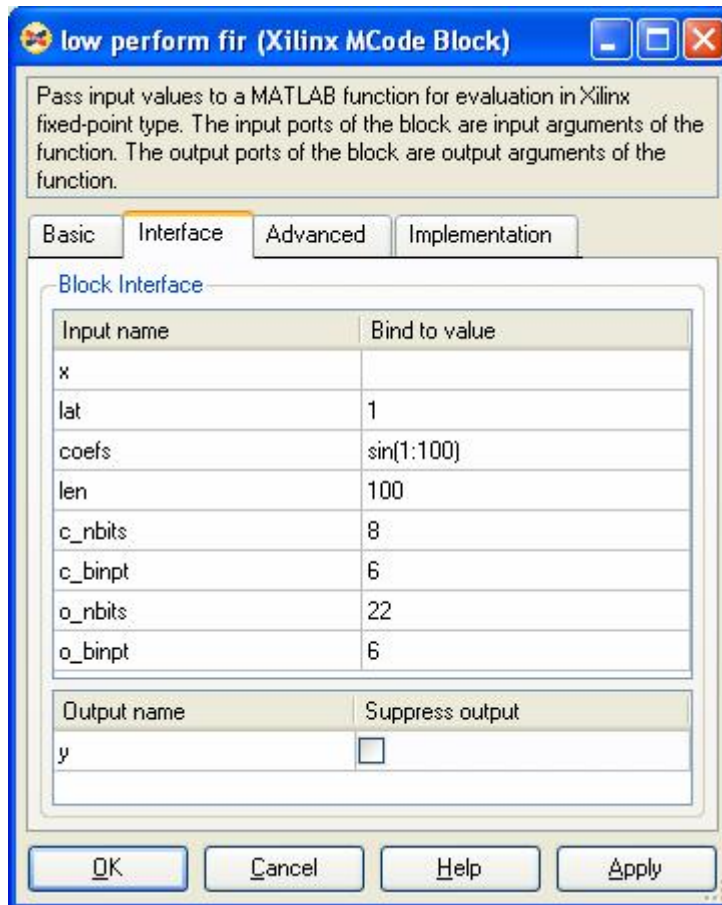
    lat = lat - 1;
    persistent dly,

    if lat <= 0
        y = reg_line.back;
    else
        dly = xl_state(zeros(1, lat), out_prec, lat);
        y = dly.back;
        dly.push_front_pop_back(reg_line.back);
    end

    for idx = len-1:-1:1
        reg_line(idx) = reg_line(idx - 1) + coef_vec(len - idx - 1) * x;
    end
    reg_line(0) = coef_vec(len - 1) * x;

```

The parameters are configured as following:



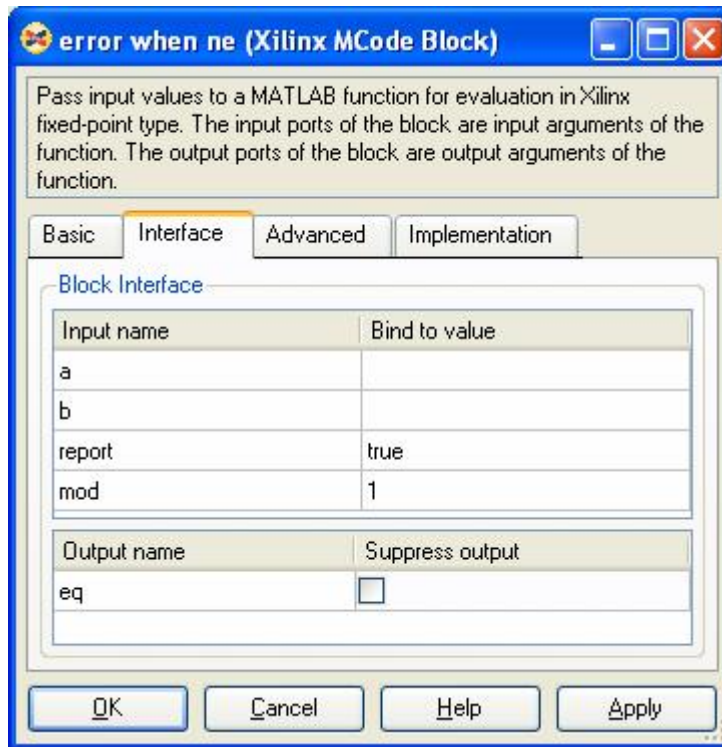
In order to verify that the functionalities of two blocks are equal, we also use another MCode block to compare the outputs of two blocks. If the two outputs are not equal at any given time, the error checking block will report the error. The following function does the error checking:

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

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

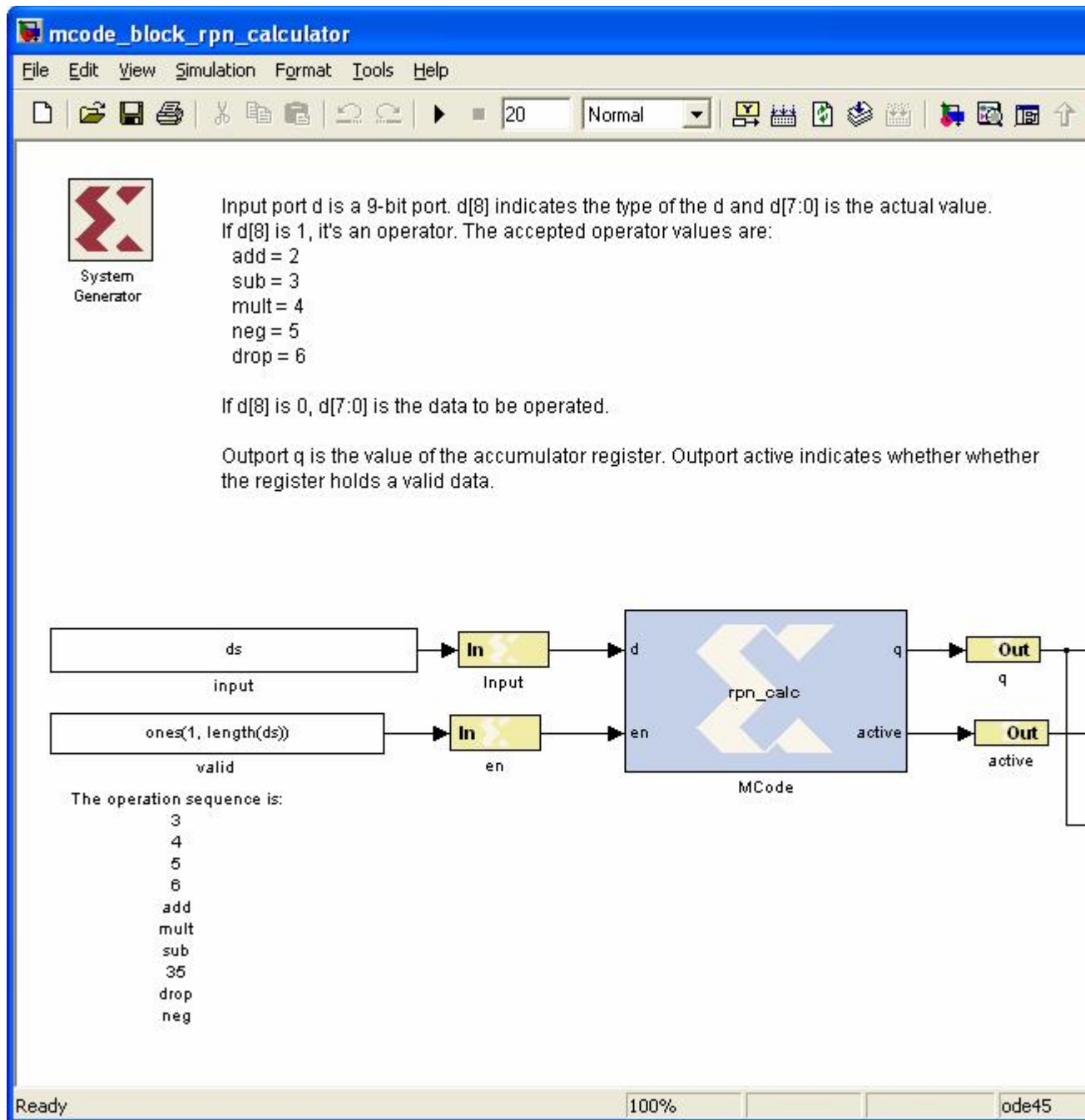
cnt = cnt + 1;
```

The block is configured as following:



RPN Calculator

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



The following function models the RPN calculator.

```
function [q, active] = rpn_calc(d, rst, en)
```

```
    d_nbits = xl_nbits(d);
```

```
    % the first bit indicates whether it's a data or operator
```

```
    is_oper = xl_slice(d, d_nbits-1, d_nbits-1)==1;
```

```
    din = xl_force(xl_slice(d, d_nbits-2, 0), 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 for disp Function

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

```

function x = testdisp(a, b)
    persistent dly, dly = xl_state(zeros(1, 8), a);
    persistent rom, rom = xl_state([3, 2, 1, 0], a);

    disp('Hello World!');
    disp(['num2str(dly) is ', num2str(dly)]);
    disp('disp(dly) is ');
    disp(dly);
    disp('disp(rom) is ');
    disp(rom);
    a2 = dly.back;
    dly.push_front_pop_back(a);

    x = a + b;

    disp(['a = ', num2str(a), ', ', ' ', ...
        'b = ', num2str(b), ', ', ' ', ...
        'x = ', num2str(x)]);

    disp(num2str(true));

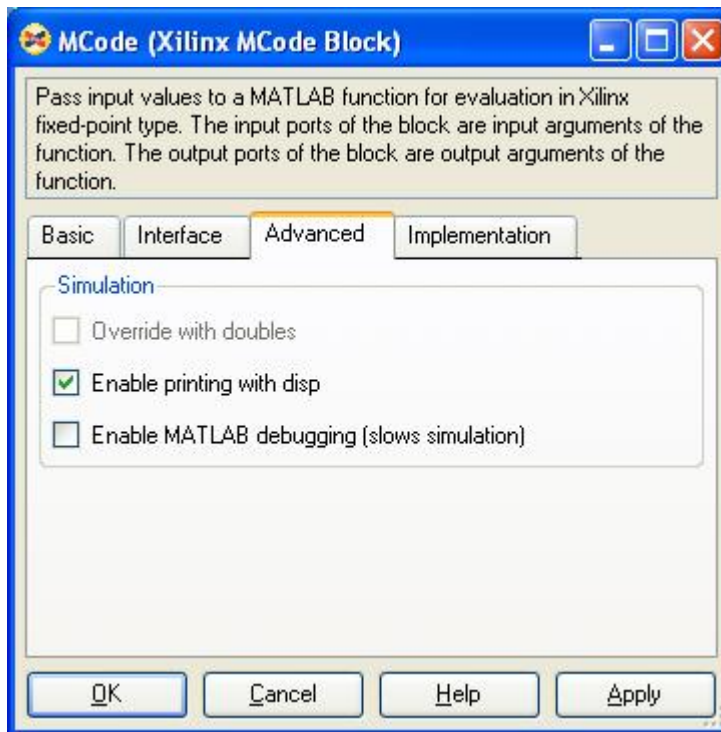
    disp('disp(10) is');
    disp(10);
    disp('disp(-10) is');
    disp(-10);

    disp('disp(a) is ');
    disp(a);

    disp('disp(a == b)');
    disp(a==b);

```

The Enable print with disp option must be checked.



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

```
mcode_block_disp/MCode (Simulink time: 0.000000, FPGA clock: 0)
Hello World!
num2str(dly) is [0.000000, 0.000000, 0.000000, 0.000000, 0.000000,
0.000000, 0.000000, 0.000000]
disp(dly) is
    type: Fix_11_7,
    maxlen: 8,
    length: 8,
    0: binary 0000.00000000, double 0.000000,
    1: binary 0000.00000000, double 0.000000,
    2: binary 0000.00000000, double 0.000000,
    3: binary 0000.00000000, double 0.000000,
    4: binary 0000.00000000, double 0.000000,
    5: binary 0000.00000000, double 0.000000,
    6: binary 0000.00000000, double 0.000000,
    7: binary 0000.00000000, double 0.000000,

disp(rom) is
    type: Fix_11_7,
    maxlen: 4,
    length: 4,
    0: binary 0011.00000000, double 3.0,
    1: binary 0010.00000000, double 2.0,
    2: binary 0001.00000000, double 1.0,
    3: binary 0000.00000000, 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: 0000.00000000, double: 0.000000
disp(a == b)
  type: Bool, binary: 1, double: 1
```


Importing a System Generator Design into a Bigger System

A System Generator design is often incorporated as a part of a larger HDL design. This section shows how to embed two System Generator designs into a larger design, and how VHDL created by System Generator can be incorporated into a simulation model of the overall system.

The most convenient way to incorporate a System Generator design into an HDL design is to encapsulate the entire design into a single binary module in the NGC binary netlist format used by the Xilinx ISE tool suite. In this case, the System Generator design is viewed as a black box by the logic synthesis tool.

The design flow to import a System Generator design into a larger system is shown in Figure 1.

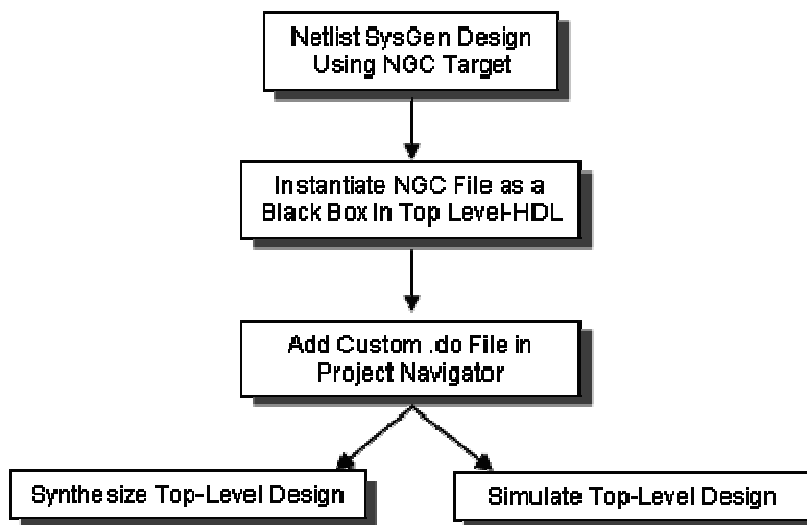


Figure 1: Compilation Flow

NGC Netlist Compilation

Selecting the NGC Netlist compilation target from the System Generator block (Figure 2) instructs System Generator to compile the design into a standalone Xilinx NGC binary netlist file. See System Generator Compilation Types for more details. When System Generator is configured to include clock wrapper logic, the design is compiled into `<design>_cw.ngc` in the target directory, where `<design>` is derived from the name of the portion of the System Generator model being compiled. Alternatively, the design is saved as `<design>.ngc` when the clock wrapper is not included.

The NGC netlist contains both the logic and constraint information for your design. This means that all HDL, cores, and constraint files normally created by System Generator are encapsulated within a single file. We will show how multiple NGC files can be used as modules in a larger design.

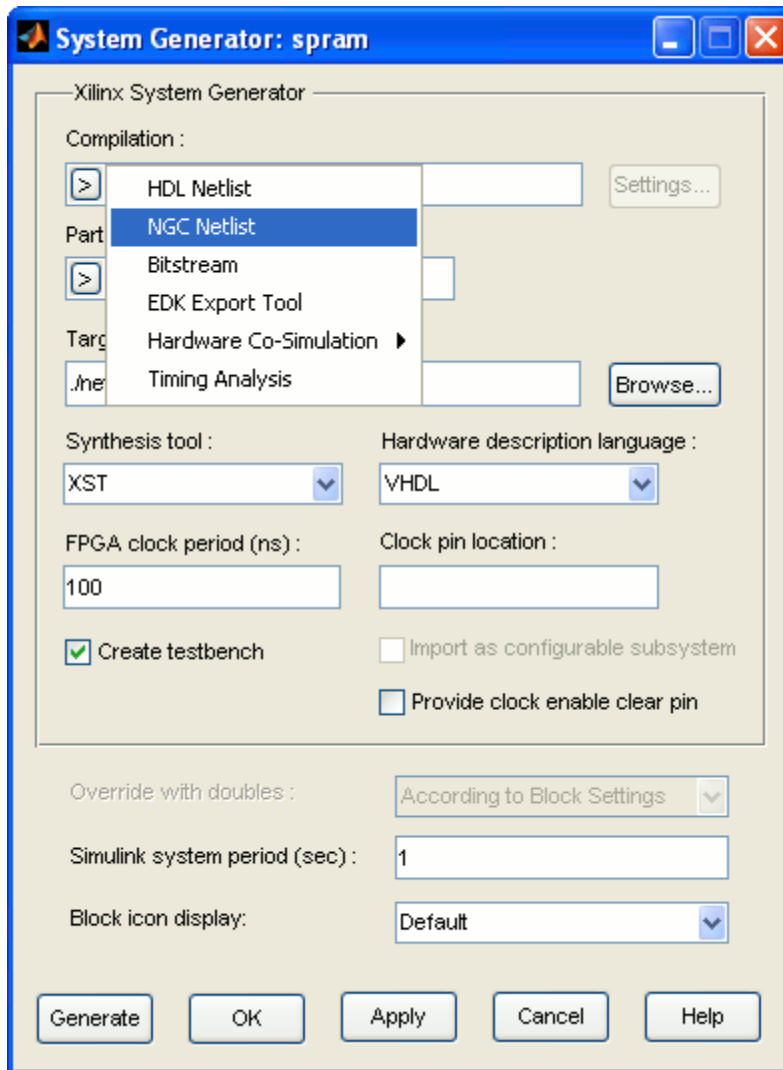


Figure 2: NGC Compilation Target

Design Rules

When a System Generator model is to be included into a larger design, there are two design rules that must be followed. First, no Gateway or System Generator block should specify an IOB/CLK location constraint. Otherwise, the NGDBuild tool will issue the following warning:

```
WARNING:NgdBuild:483 - Attribute "LOC" on "clk" is on the wrong type of
object. Please see the Constraints Guide for more information on this
attribute.
```

Second, I/O buffers must not be inserted in the NGC netlist during synthesis. Instead, I/O buffers should be instantiated in the top level HDL code.

Synthesis

Figure 3 shows the synthesis flow for an entire design when the NGC Netlist compilation target is used.

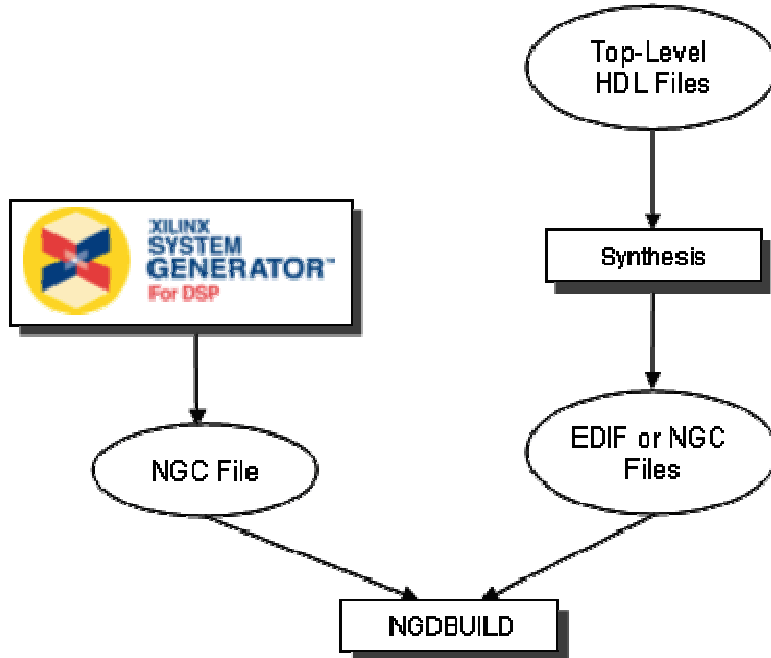


Figure 3: Synthesis Flow

The System Generator NGC module can be directly instantiated inside the top level VHDL entity as follows:

```

component foo_cw is
  port (
    . . .
  );
end component;
attribute box_type of foo_cw : component is "black_box";
attribute syn_black_box of foo_cw : component is true;
  
```

To make this process easier, System Generator creates an HDL component instantiation template when the design is compiled using the NGC target. When VHDL is selected as the hardware description language, the template is saved in the target directory as <design>_cw.vho (or <design>.vho when the clock wrapper is excluded). The file is saved using a .veo extension when Verilog is selected as the hardware description language. You may use the component template to instantiate the component in your top level entity.

The `box_type` attribute should be used when synthesizing your VHDL with XST and the `syn_black_box` attribute should be used with Synplify.

Note: When instantiating the System Generator design in your VHDL, you may notice a clock enable pin named `ce`. The `ce` port is not connected to registers within the design. It is there to allow the VHDL file to be imported as a Black Box within System Generator.

Simulation

When you compile a System Generator model into an NGC target, the VHDL files created by System Generator are necessary only for HDL simulation. They should not be included into the Project Navigator project for synthesis. This increases the performance of Project Navigator and synthesis of the top level.

If you wish to run the entire design through an HDL simulator, you need to specify a custom ModelSim .do file in Project Navigator, since the VHDL files are not included in the project. In addition to the VHDL files, you will need to place the memory initialization (.mif) and coefficient (.coe) files in the same directory as the VHDL files.

Step-by-Step Example

In this example, two NGC netlists from System Generator are imported into a larger VHDL design. Design #1 is named SPRAM and design #2 is MAC_FIR. The top level VHDL entity combines the two data ports and a control signal from the SPRAM design to create a bidirectional bus. The top level VHDL also instantiates the MAC_FIR design and supplies it a separate clock signal named clk2. A block diagram of this design is shown in Figure 4.

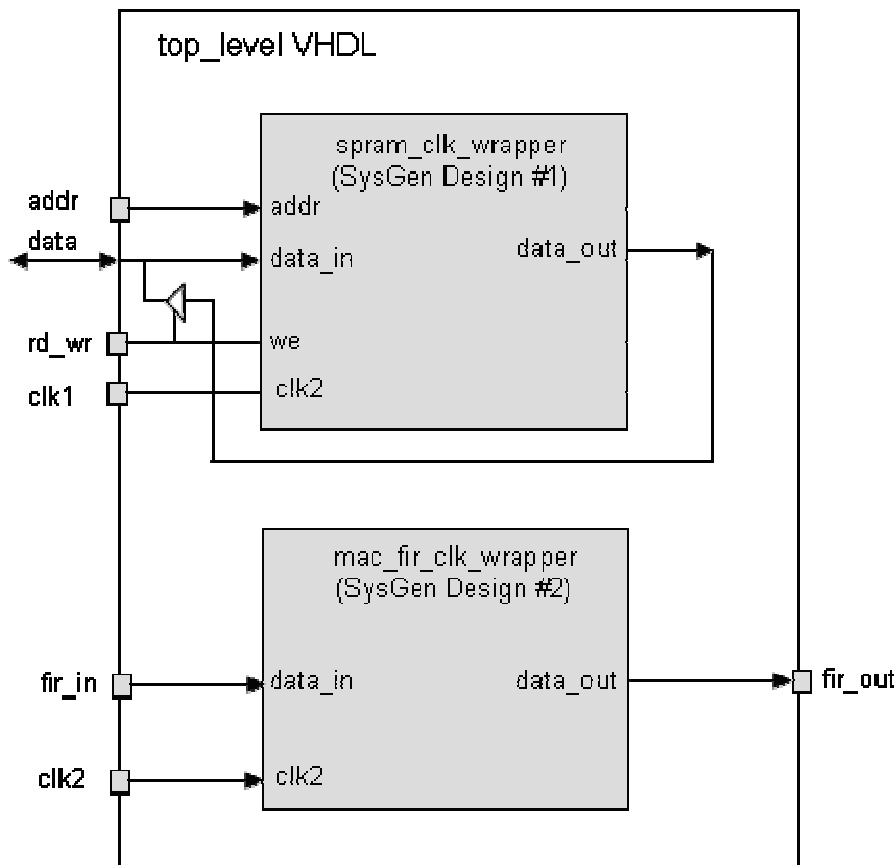


Figure 4: Design Block Diagram

The files used in this example are provided in <path_to_sysgen>/examples/import. The default path to sysgen would be \$MATLAB/toolbox/xilinx/sysgen. The following files are provided:

- spram.mdl - System Generator design #1

- `mac_fir.mdl` - System Generator design #2

Files within the sub-directory named `top_level`:

- `top_level.vhd` – Top level VHDL file
- `top_level_testbench.vhd` – Top level VHDL testbench file
- `top_level.isc` – Project Navigator project for compiling `top_level` design
- `top_level_testbench.do` – Custom ModelSim .do file
- `wave.do` – ModelSim .do file called by `top_level_testbench.do` to display waveforms

Generating the NGC files for the System Generator Designs

The steps used to create the NGC files are as follows:

1. Open the first design, `spram.mdl`, in MATLAB. This is a multirate design due to the down sampling block placed after the output of the Single Port RAM. You should verify the constraints for this design have been applied properly by looking at the PAR report.
2. Double click on the System Generator block, select the NGC Netlist target and press the Generate button. By pressing the Generate button, the NGC file for this design is created in the
`<path_to_sysgen>/examples/import/ngc_netlist1` directory.
3. Repeat steps 1 and 2 for the `mac_fir.mdl` model. The NGC file for this design is created in the
`<path_to_sysgen>/examples/import/ngc_netlist2` directory.

Synthesizing the Top Level Design

The next two steps are used to synthesize the `top_level` design:

1. Copy all the .ngc files from the `ngc_netlist1` and `ngc_netlist2` directories to the `top_level` project directory so that NGDBUILD can fill the Black Box in the `top_level.vhd` file. The .ngc files created are named `spram_cw.ngc` and `mac_fir_cw.ngc`
2. Now we synthesize the top-level design. In Windows Explorer, go to the
`<path_to_sysgen>/examples/import/top_level` directory and double click on the Project Navigator project file named `top_level.isc`. Select `top_level-behavior` in the Sources in Project window and then double left click on the Place & Route Report process (the process is under Implement Design -> Place & Route) process in the Processes for Source window.

Note that in the PAR report the multirate constraints were met:

Constraint	Requested	Actual	Logic Levels	Absolute Slack	Number of errors

TS_clk_c4b7e244 = PERIOD TIMEGRP "clk_c4b7e244" 100 ns HIGH 50%	100.000ns	6.846ns	1	93.154ns	0
TS_clk_f488215c = PERIOD TIMEGRP "clk_f488215c" 100 ns HIGH 50%	100.000ns	5.194ns	1	94.806ns	0
TS_ce_2_f488215c_group_to_ce_2_f488215c_group = MAXDELAY FROM TIMEGRP "ce_2_f488215c_group" TO TIMEGRP "ce_2_f488215c_group" 200 ns	200.000ns	4.866ns	0	195.134ns	0
TS_ce_16_c4b7e244_group_to_ce_16_c4b7e244_group = MAXDELAY FROM TIMEGRP "ce_16_c4b7e244_group" TO TIMEGRP "ce_16_c4b7e244_group" 1600 ns	1600.000ns	2.526ns	1	1597.474ns	0

These constraints were created by System Generator and applied to each netlist when the System Generator portions of the design were synthesized by XST. A system sample period of 100 ns was set in the System Generator block for both designs. The TS_clk_c4b7e244 constraints in green are from the FIR design and TS_clk_f488215c constraints in yellow are from the SRAM design. The down sampling block in the SRAM design performs a down sample by 2. The ce2_f488215c_group_to_ce2_f488215c_group constraint is for all the synchronous elements after the down sampler and is set to twice the system sample period.

Although a top-level user constraint file (.ucf) was not specified in the top_level project, there is no reason why one cannot be specified.

Simulating the Entire Design

To perform a behavioral simulation of the top_level design:

1. System Generator creates VHDL files and invokes the synthesis tool to produce the NGC netlist. These VHDL files are used when simulating the top_level design. The VHDL files generated for a design are named <design>_cw.vhd, and <design>.vhd. Open the custom ModelSim .do file named top_level_testbench.do to see how the VHDL files for both designs are referenced.
2. Memory initialization (.mif) and coefficient (.coe) files that are used during simulation must be placed in the same directory as the top level VHDL file. Copy over .mif and .coe files for both designs into the top_level directory.
3. In Project Navigator, select **Behavioral Simulation** option from the pull-down menu in the Sources window. Then select top_level_testbench-structural(top_level.vhd) source file. This file is imported into the project as a testbench file it thus allowing you to simulate the design using the ModelSim Simulator.
4. In the **Processes** window, right click on the **Simulate Behavioral Model** option under **ModelSim Simulator**, and select **Properties...** Note that a Custom Do File has been specified as shown in Figure 5.

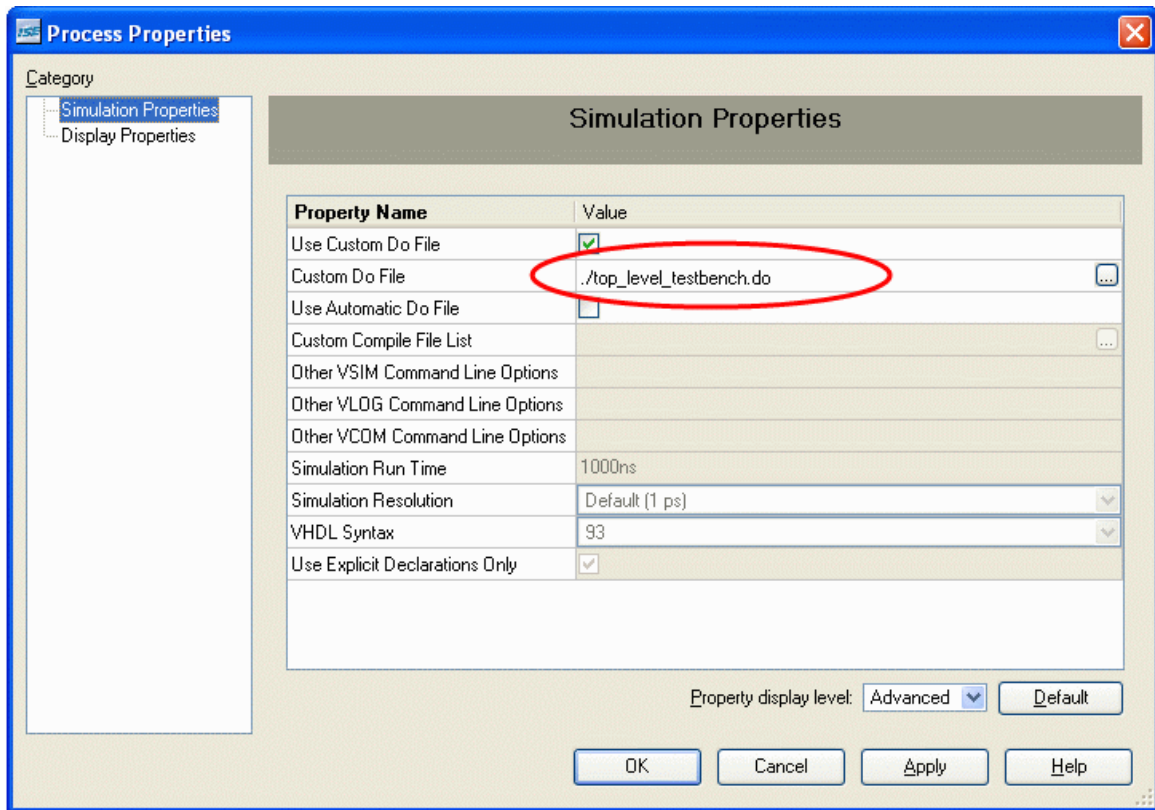


Figure 5: Specification of Custom .do File

The custom .do file is as follows:

```
vlib work
## Files for Design #1
vcom -93 -nowarn 1 ../ngc_netlist1/spram.vhd
vcom -93 -nowarn 1 ../ngc_netlist1/spram_cw.vhd

## Files for Design #2
vcom -93 -nowarn 1 ../ngc_netlist2/mac_fir.vhd
vcom -93 -nowarn 1 ../ngc_netlist2/mac_fir_cw.vhd

## Top level files
vcom -93 -explicit top_level.vhd
vcom -93 -explicit top_level_testbench.vhd

vsim -t 1ps -lib work top_level_testbench
do wave.do
run 10000ns
```

The highlighted lines show the ModelSim commands used to compile the VHDL code generated by System Generator. If you allow Project Navigator to automatically create a ModelSim .do file, these VHDL files are not included since they are not part of the project. These files are intentionally not included into the project because they have already been compiled and imported as a NGC black box and are only used for simulation purposes. This approach increases the performance of Project Navigator, and decreases the time needed to synthesize your top-level design.

5. To simulate the top_level design, double left click on the Simulate Behavioral Model process. The ModelSim .do file compiles the VHDL code and runs the simulation for 10000 ns. The resulting waveform is shown in Figure 6.

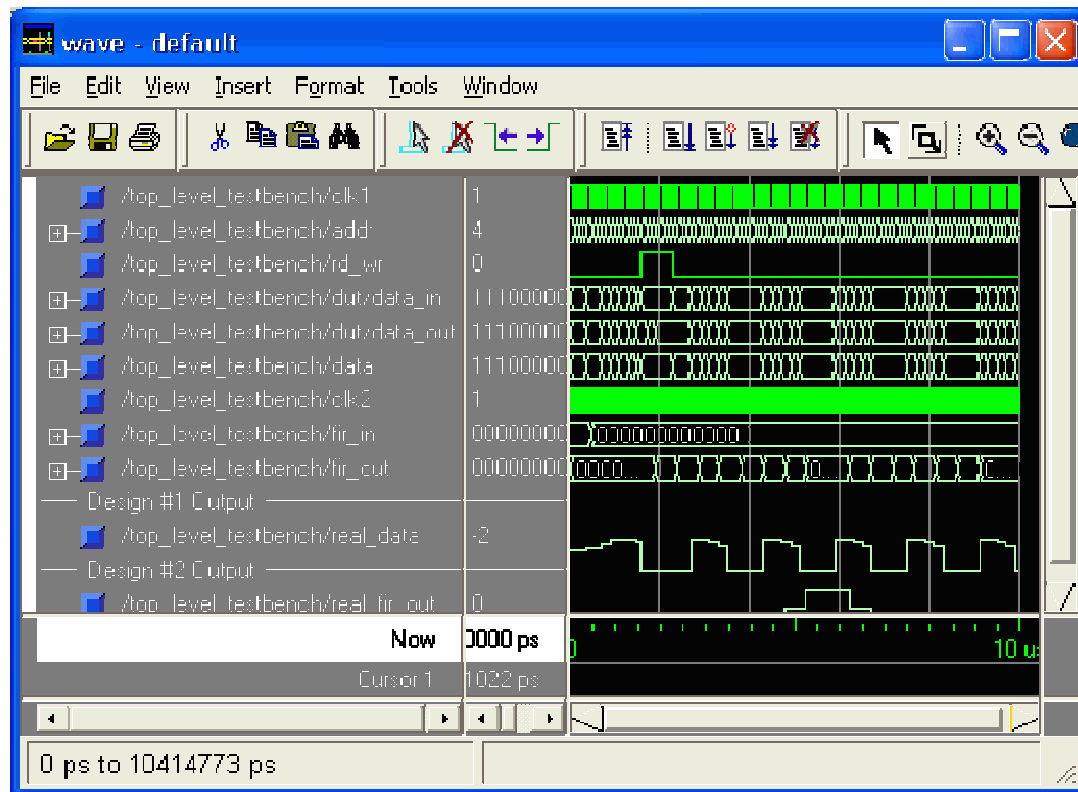


Figure 6: ModelSim Waveform

Summary

This section has shown how to import a System Generator Design into a larger system. There are a few important things to keep in mind during each phase of the process.

While creating a System Generator design:

- IOB constraints should not be specified on Gateways in the System Generator model; neither should the System Generator block specify a clock pin location.
- Use the NGC Netlist compilation target in the System Generator block. The NGC netlist file that System Generator produces contains both the logic and constraint information for your design.

To instantiate System Generator in the top_level HDL:

- Use a black box and assign the appropriate black box attribute
- The `ce` port is not connected to registers within your design. It is provided so that the VHDL file can be imported as a Black Box within System Generator.

For top level simulation:

- Create a custom ModelSim .do file in order to compile the VHDL files created by System Generator. Modify the Project Navigator settings to use this custom .do file.

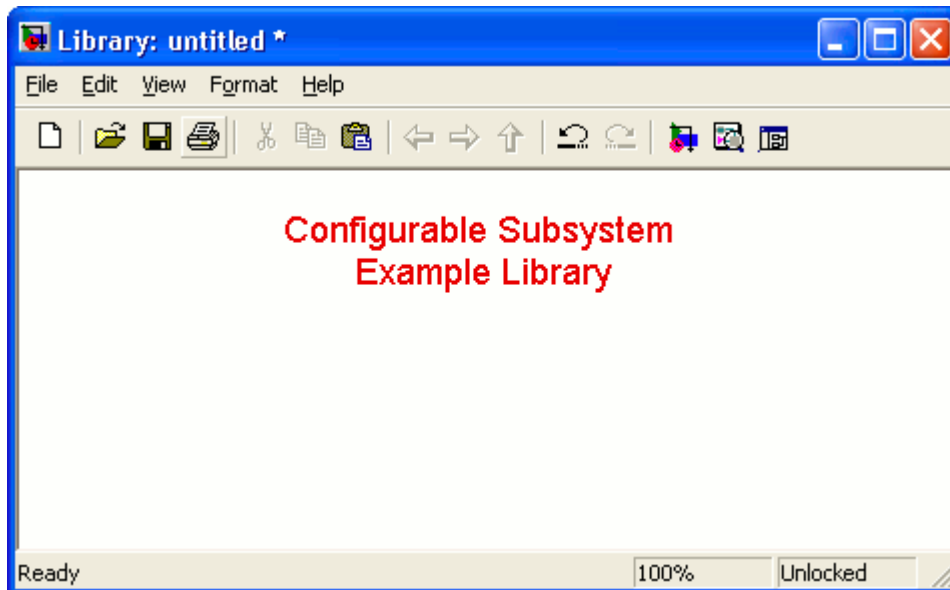
Configurable Subsystems and System Generator

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

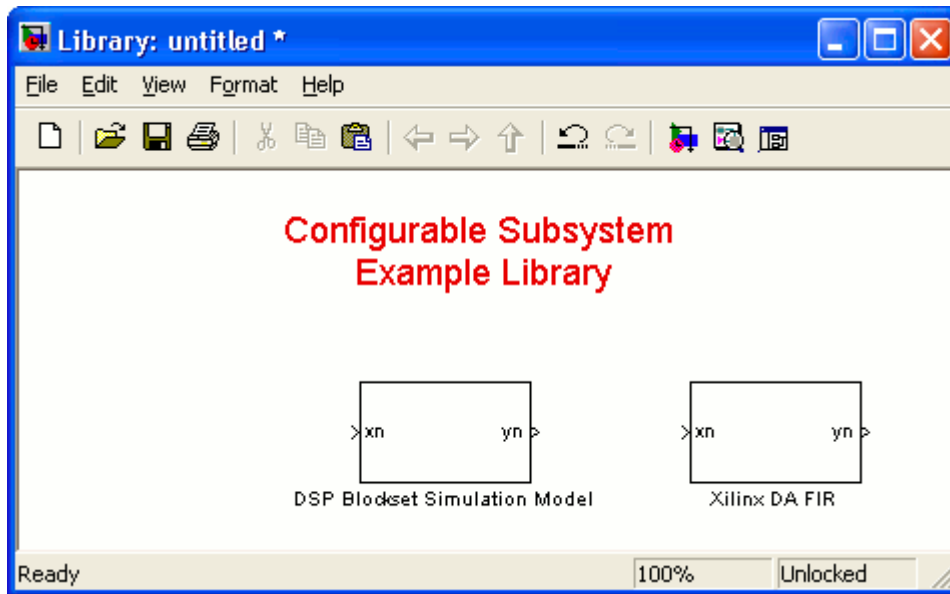
Defining a Configurable Subsystem

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

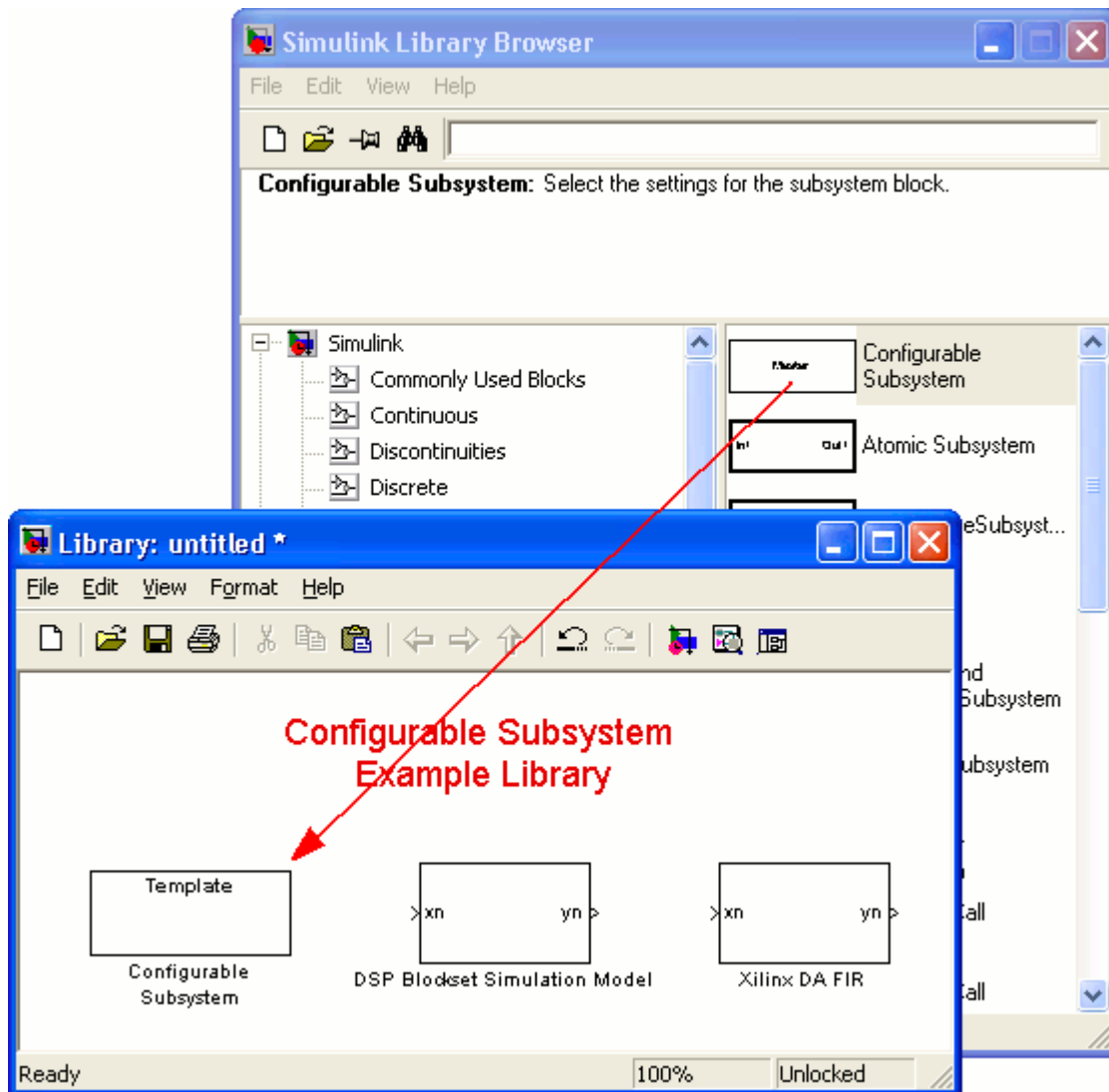
- Make a new empty library.



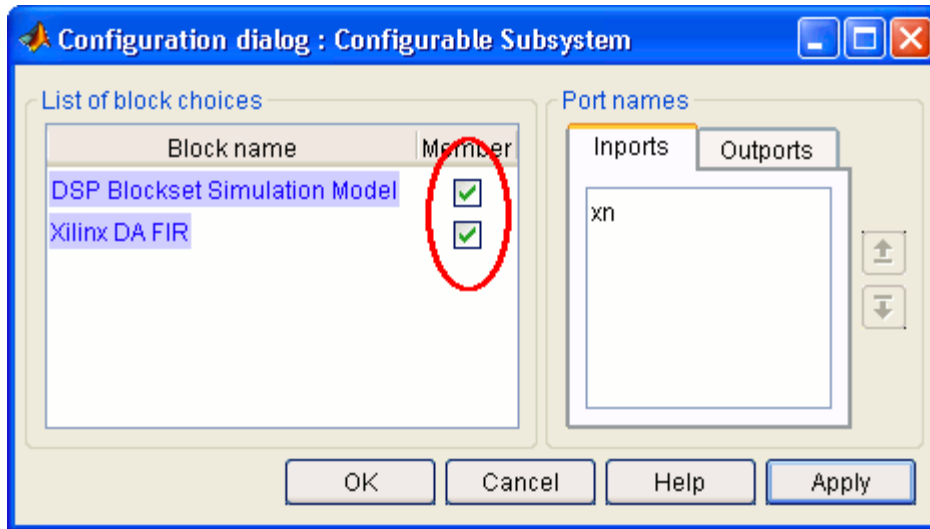
- Add the underlying blocks to the library.



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



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

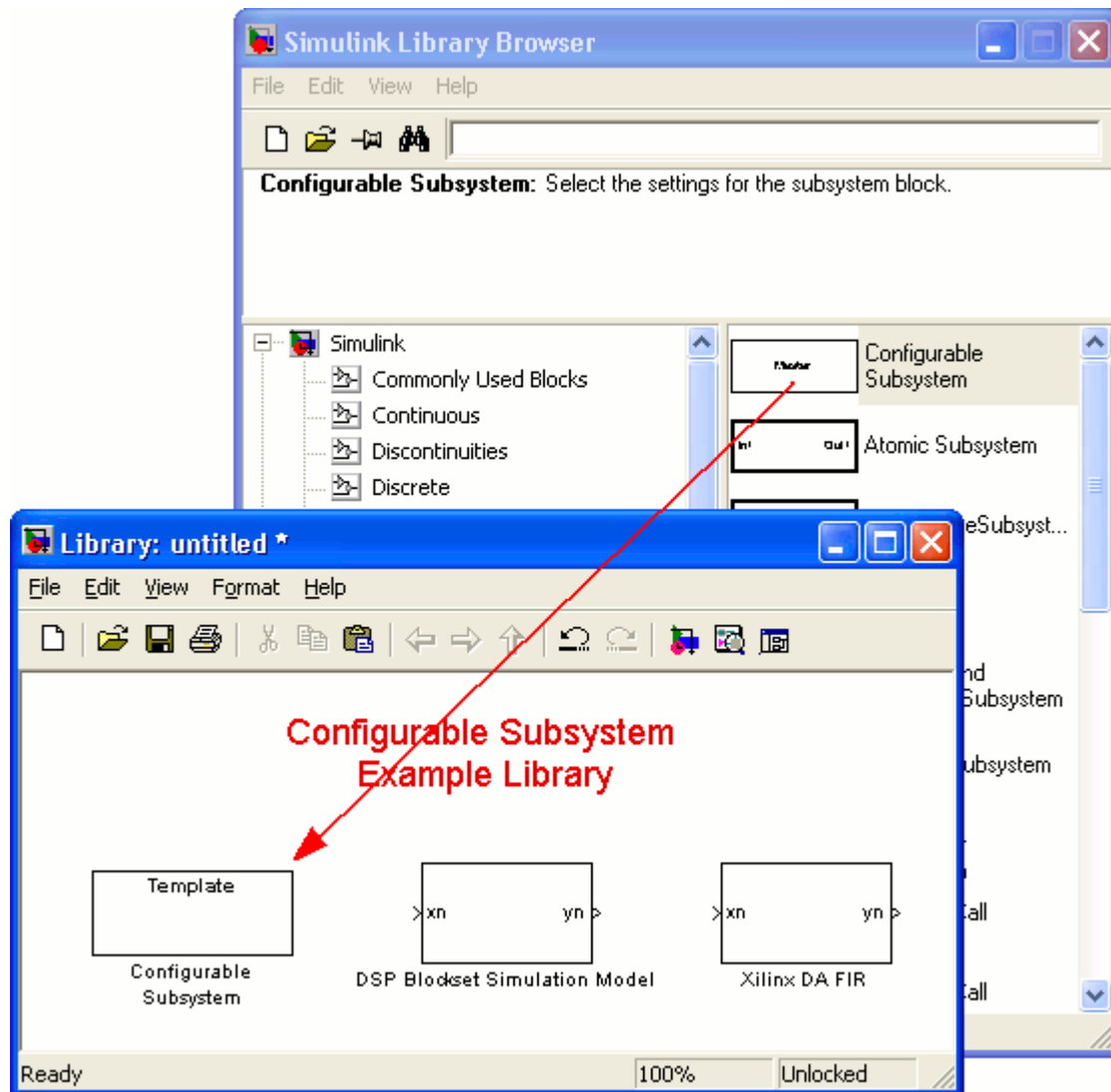


- Press OK, and then save the library again.

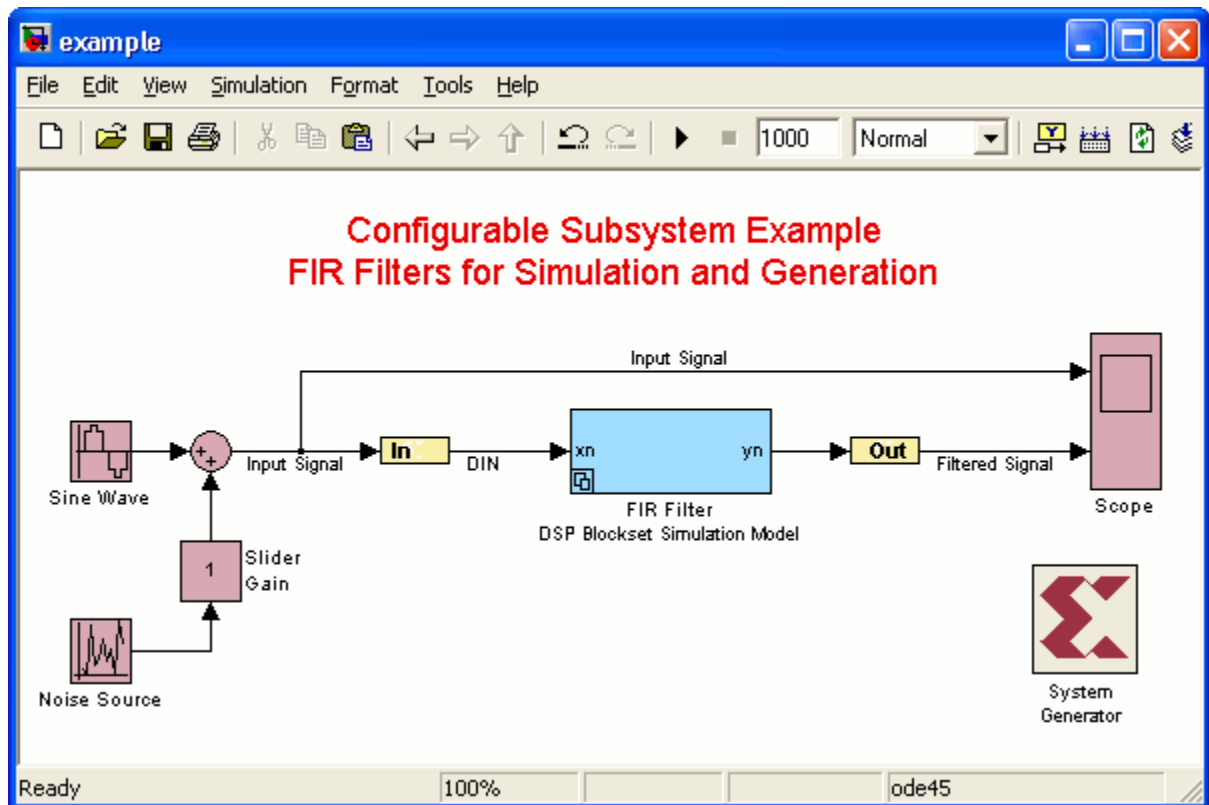
Using a Configurable Subsystem

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

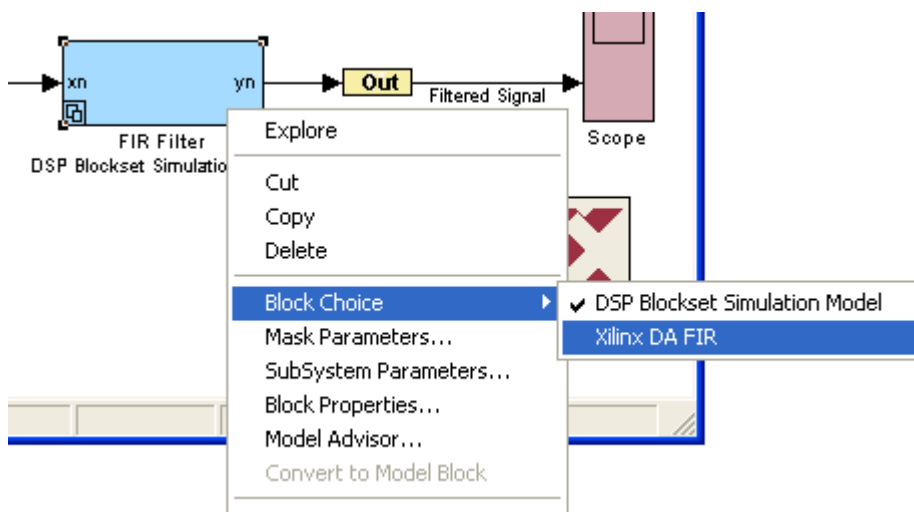
- As described above, create the library that defines the configurable subsystem.
- Open the library.
- Drag a copy of the template block from the library to the appropriate part of the design.



The copy becomes an instance of the configurable subsystem.



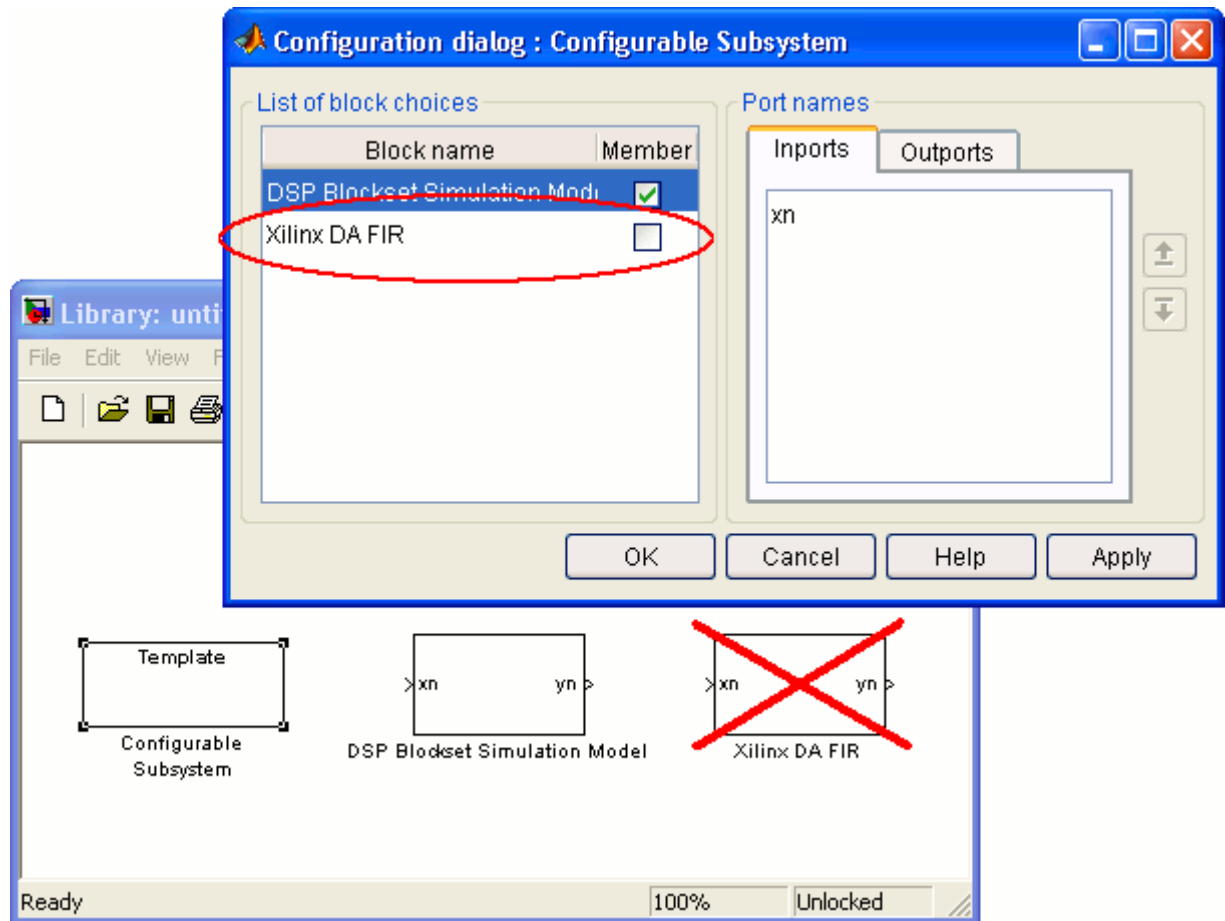
- Right-click on the instance, and under **Block choice** select the block that should be used as the underlying implementation for the instance.



Deleting a Block from a Configurable Subsystem

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

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

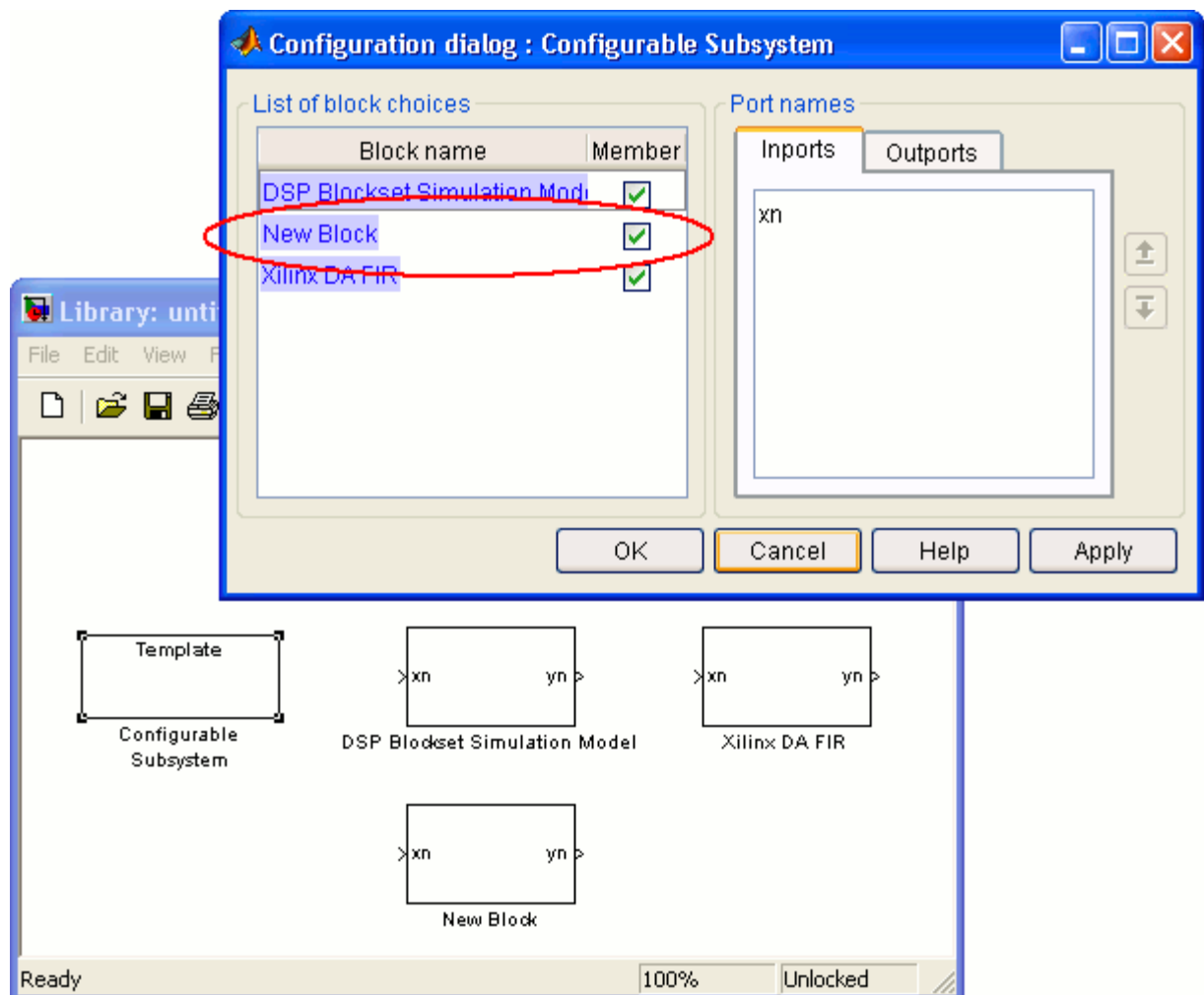


- Save the library.
- Compile the design by typing `Ctrl-d`.
- If necessary, update the choice for each instance of the configurable subsystem.

Adding a Block to a Configurable Subsystem

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

- Open and unlock the library for the subsystem.
- Drag a block into the library.
- Double click on the template, and turn on the checkbox next to the added block.

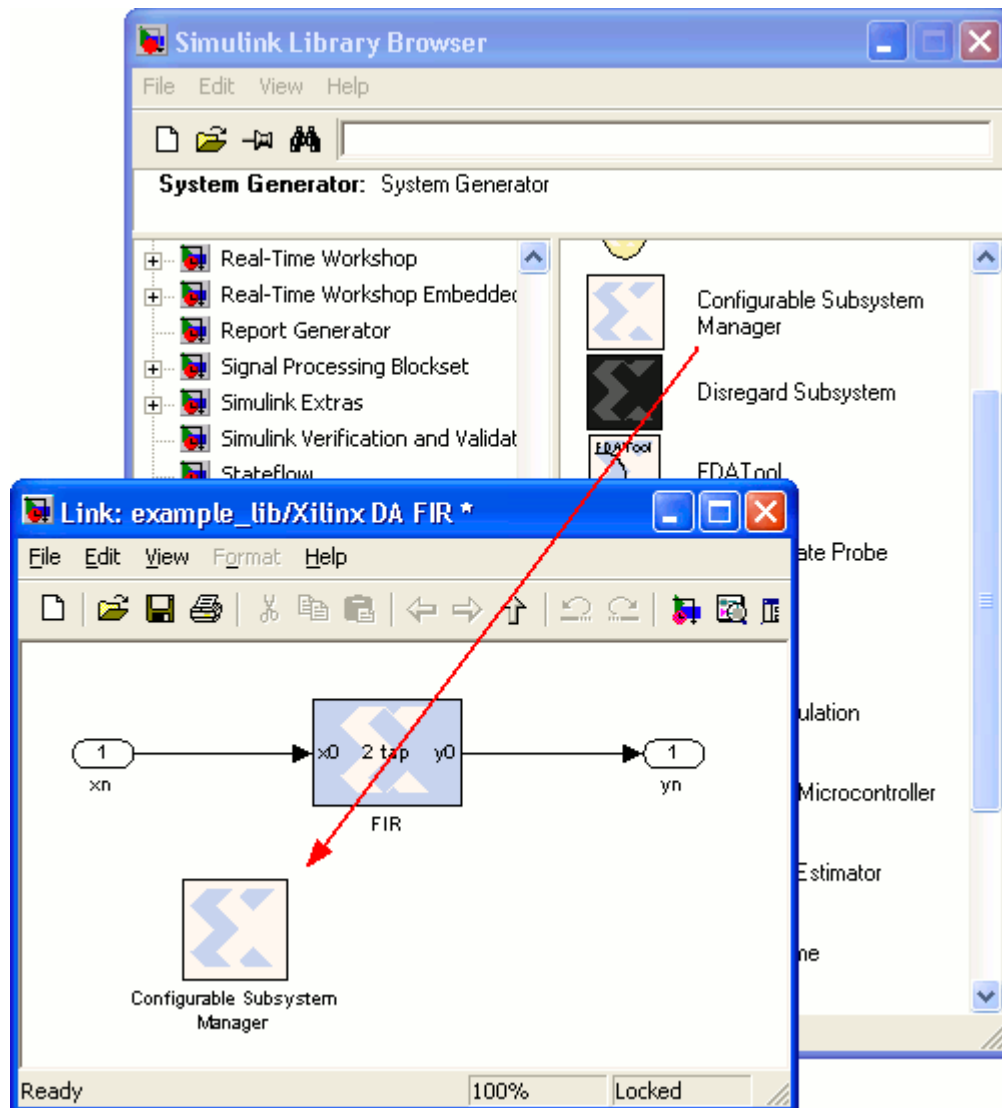


- Press OK, and then save the library.
- Compile the design by typing **Ctrl-d**.
- If necessary, update the choice for each instance of the configurable subsystem.

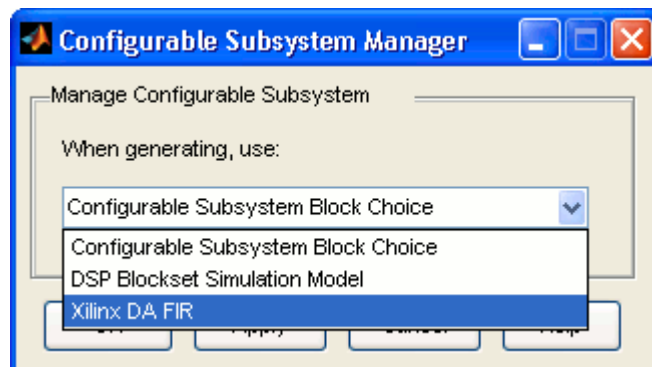
Generating Hardware from Configurable Subsystems

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

- Open and unlock the library for the configurable subsystem.
- Select one of the blocks in the library, and double click to open it. (Aside from the template any block will do, provided the block is itself a subsystem. If there is no such subsystem in the library, it is not possible to use a configurable subsystem manager.)
- Drag a manager block into the subsystem opened above. (The manager block can be found in *Xilinx Blockset/Tools/Configurable Subsystem Manager*.)



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



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

The Mathworks description of configurable subsystems can be found in
<http://www.mathworks.com/access/helpdesk/help/toolbox/simulink/slref/configurablesubsystem.shtml>.

Notes for High Performance FPGA Design

Using the following design practices helps System Generator produce efficient and high performance hardware realizations.

- **Pay close attention to the Hardware Notes included in the block dialog boxes.** Many blocks in the Xilinx Blockset library have notes that explain how to achieve the most hardware efficient implementation. For example, the notes point out that the Scale block costs nothing in hardware. By contrast, the Shift block (which is sometimes used for the same purpose) can use hardware.
- **Register inputs and outputs of your design.** This can be done by placing a Delay block having latency 1 or a Register block after the Gateway In and before Gateway Out blocks. Selecting any of the Register block features adds hardware.
- **Double registering the I/Os may 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 since this block is implemented using a SRL16 and cannot be packed into an IOB.
- **Use the IOB Timing Constraint option: Data Rate, Set 'FAST' Attribute on all Gateway In and Gateway Out blocks.** When this attribute is selected on gateways, IOB delay is reduced (but I/O noise and power consumption increases).
- **Insert pipeline registers wherever possible.** Deep pipelines are efficiently implemented with the Delay blocks since the SRL16 primitive is used. The Delay block has an option to turn on Register retiming under the implementation tab. This allows the synthesis tool to move registers in order to reduce the levels of logic in a combinatorial path. If an initial value is needed on a register, the Register block should be used.
- **Saturation arithmetic and rounding have area and performance costs.** Use only if necessary.
- **Use System Generator Timing Analysis Tool to Meet Timing Requirements.** System Generator provides a Timing Analysis tool that can help resolve timing related issues. The timing analysis tool shows the user the slowest paths and those paths which are failing to meet the timing requirements. For more information refer to Timing Analysis Tool section.

Processing a System Generator Design with FPGA Physical Design Tools

HDL Simulation

System Generator creates custom .do files for use with your generated project and a ModelSim simulator. To use these files, you must have ModelSim (PE or EE/SE) or the Xilinx Edition of ModelSim (MXE). You may run your simulations from the standalone ModelSim tool, or you may associate it with the Xilinx ISE Project Navigator, and run your simulations from within Project Navigator as part of the full software implementation flow.

Compiling your IP

Before you can simulate your design, you must compile your IP (cores) libraries with ModelSim.

ModelSim (PE or EE/SE)

There are multiple ways to compile your IP libraries. Complete instructions for running `compplib` can be found on section *Simulating Your Design*, subsection *Compiling Xilinx Simulation Libraries* on the ISE Software Manual, which is available online at:

<http://toolbox.xilinx.com/docsan/xilinx82/books/docs/sim/sim.pdf>

From the Windows command line you can compile the necessary HDL libraries using the `compplib` program. For example, the following command can be used to compile all the HDL libraries with ModelSim SE:

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

MXE libraries

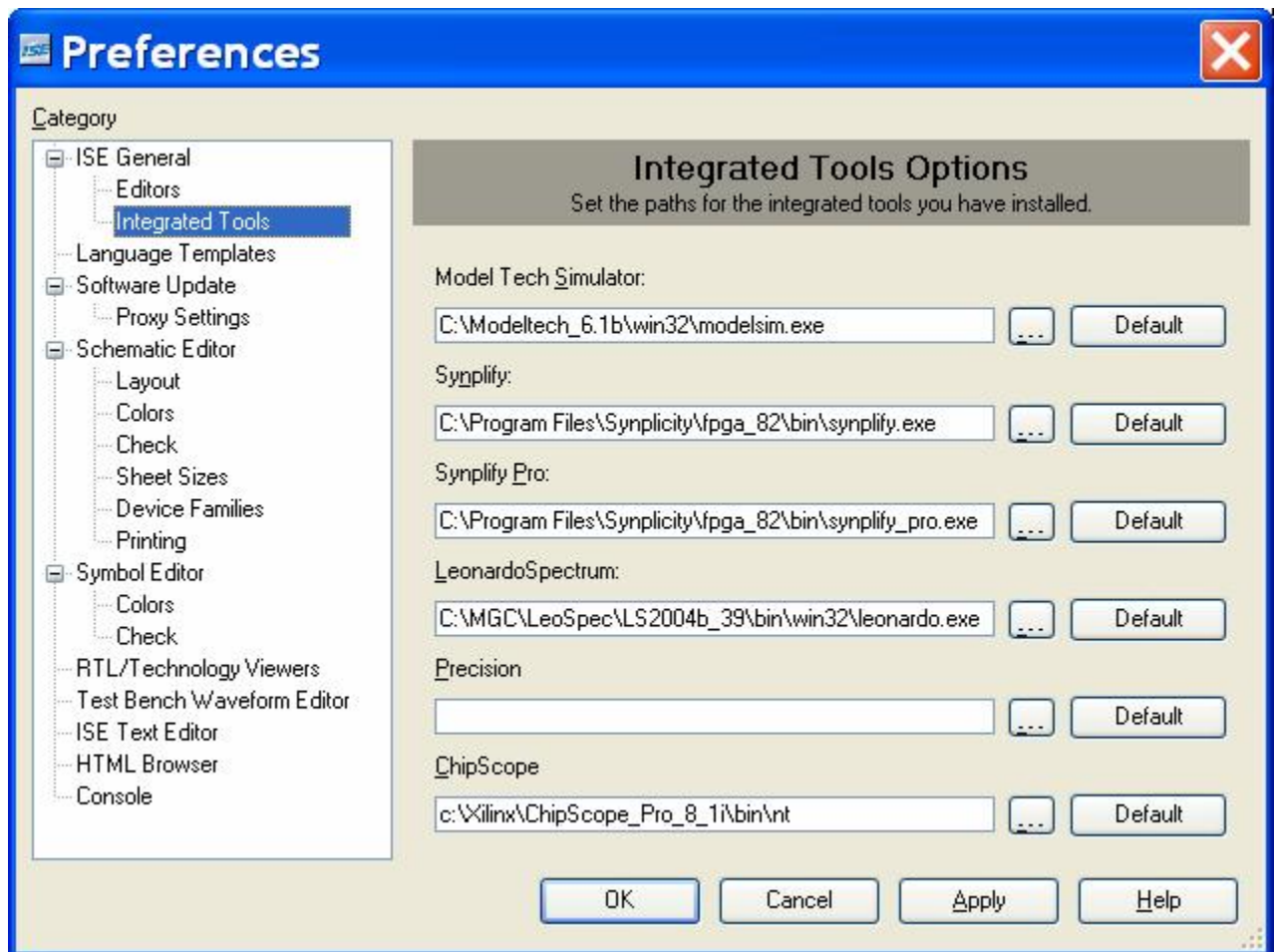
If you plan to use ModelSim XE (Xilinx Edition), download the MXE pre-compiled libraries from the Xilinx web site. The latest libraries are located at:

http://www.xilinx.com/ise/optional_prod/mxe.htm

Unzip these MXE libraries into your MXE installed directory, e.g.: C:/Modeltech_XE/. This is the location where MXE expects to find your Xilinx compiled libraries, so you do not need to make any changes to your modelsim.ini file. This file should point to the correct installed location.

Simulation using ModelSim within Project Navigator

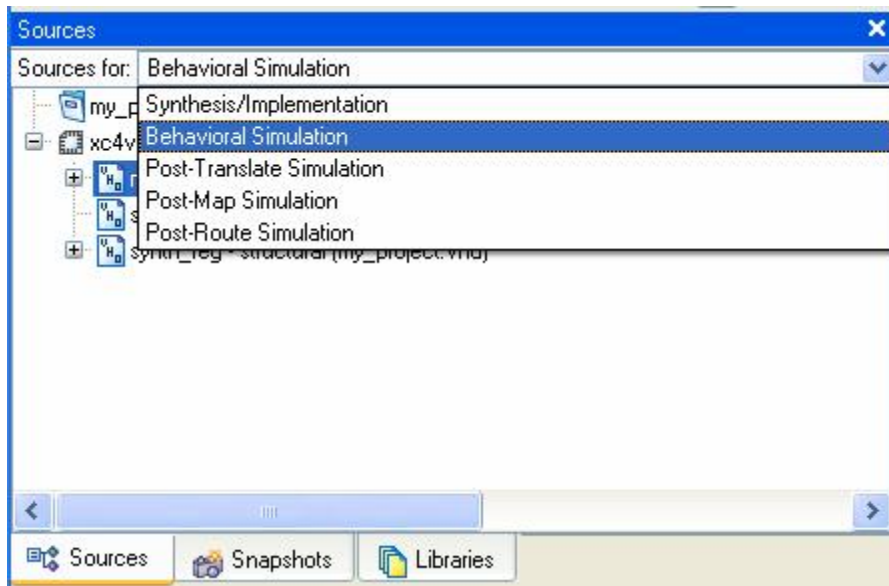
Before you can launch ModelSim from Project Navigator you must specify the location of your installed version of ModelSim (either MXE or ModelSim EE/SE/PE). To do so, open Project Navigator and choose the main menu **Edit->Preferences**. This brings up a dialog box. Choose the **ISE General->Integrated Tools** category in the dialog box. Enter the full path to the version of ModelSim on your PC in the Model Tech Simulator edit box. You must include the name of the executable file in this field.



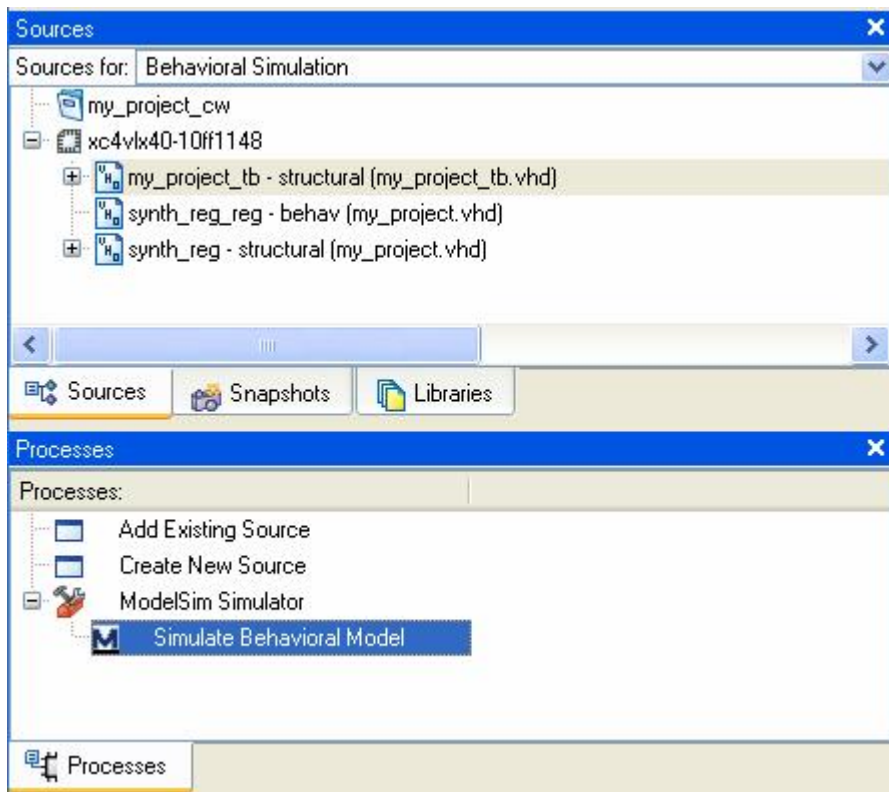
The Project Navigator project is already set up to run simulations at four different stages of implementation. System Generator creates four different ModelSim .do files when the Create Testbench option is selected on the System Generator block. The ModelSim do files created by System Generator are:

- **pn_behavioral.do** - for a behavioral (HDL) simulation on the HDL files in the project, before any synthesis or implementation.
- **pn_posttranslate.do** - this file runs a simulation on the output of the Xilinx translation (ngdbuild) step, the first step of implementation.
- **pn_postmap.do** - to run a simulation after your design has been mapped. This file also includes a back-annotated simulation on the post-mapped design.
- **pn_postpar.do** - to run a simulation after your design has been placed and routed. This file also includes a back-annotated simulation step.

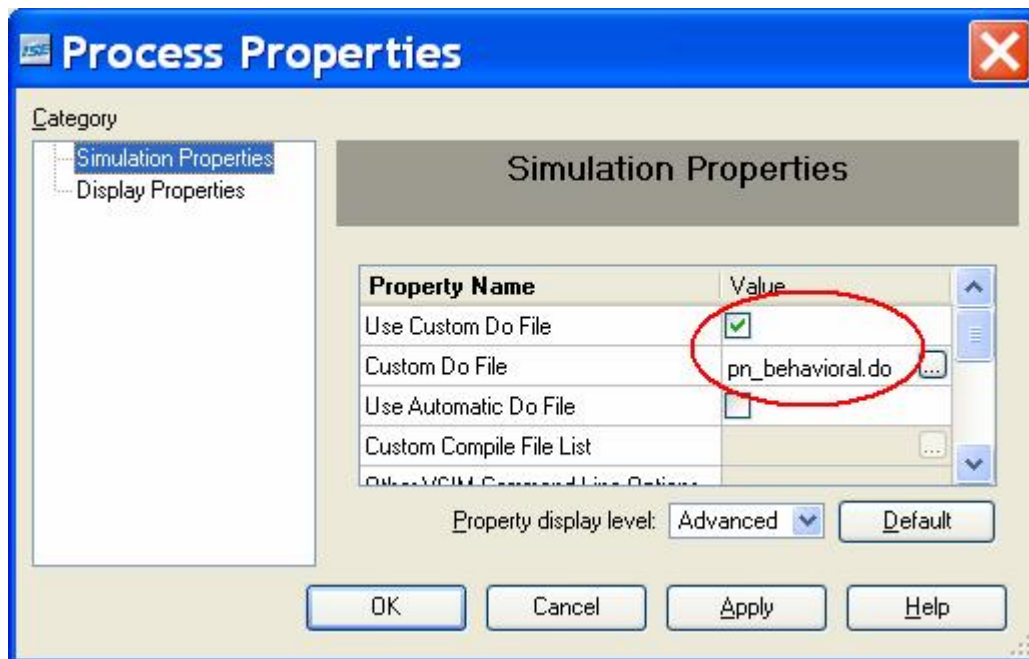
In the Project Navigator **Sources** window, use the pull-down menu labeled **Sources for** to select Behavioral Simulation, Post-Translate Simulation, Post-Map Simulation, or Post-Route Simulation (corresponding to pn_behavioral.do, pn_posttranslate.do, pn_postmap.do, and pn_postpar.do respectively).



If you select the <your design>_tb.vhd/.v file in the Project Navigator **Sources** window, the ModelSim Simulator will become available in the **Process** window. Expand the ModelSim Simulator process by clicking on the plus button to the left of it. A simulation process associated with the ModelSim Simulator will appear (in the image below the process is labeled Simulate Behavioral Model).



The **Process Properties** dialog box shows that the System Generator .do file is already associated as a custom file for this process.



Now if you double-click on the simulation process, the ModelSim console opens, and the associated custom do file is used to compile and run your System Generator testbench. The testbench uses the same input stimuli that was generated in Simulink, and compares the HDL simulation results with the Simulink results. Provided that your design was error free, ModelSim reports that the simulation finished without errors.

Generating an FPGA Bitstream

Xilinx ISE Project Navigator

During code generation, the System Generator creates several project files for use in Xilinx and partner software tools. One of these project files is for the Xilinx ISE Project Navigator tool. By opening this project file, you can import your System Generator design into the Project Navigator, and from there, you can synthesize, simulate, and implement the design. This file is called `<design_name>_cw.ise` and it is created in the target directory specified in the System Generator block.

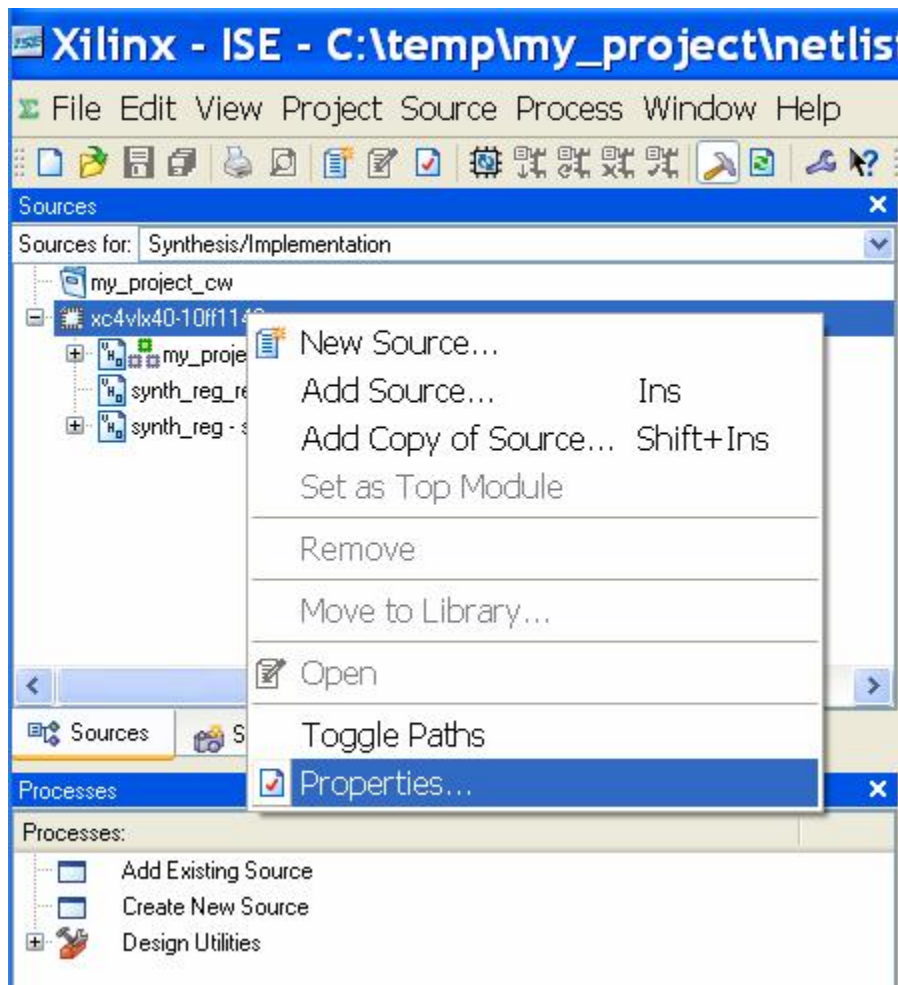
Note: `my_project_cw.ise` is used in the following discussion.

Opening a System Generator project

You may double-click on your `.ise` file in Windows Explorer. The Project Navigator file association with `.ise` causes Project Navigator to launch, opening your `my_project_cw.ise` System Generator design project. You may also open the Project Navigator tool directly, then choose **File -> Open Project** from the top level pull down menu. Browse to the location of your System Generator `my_project_cw.ise` and open it.

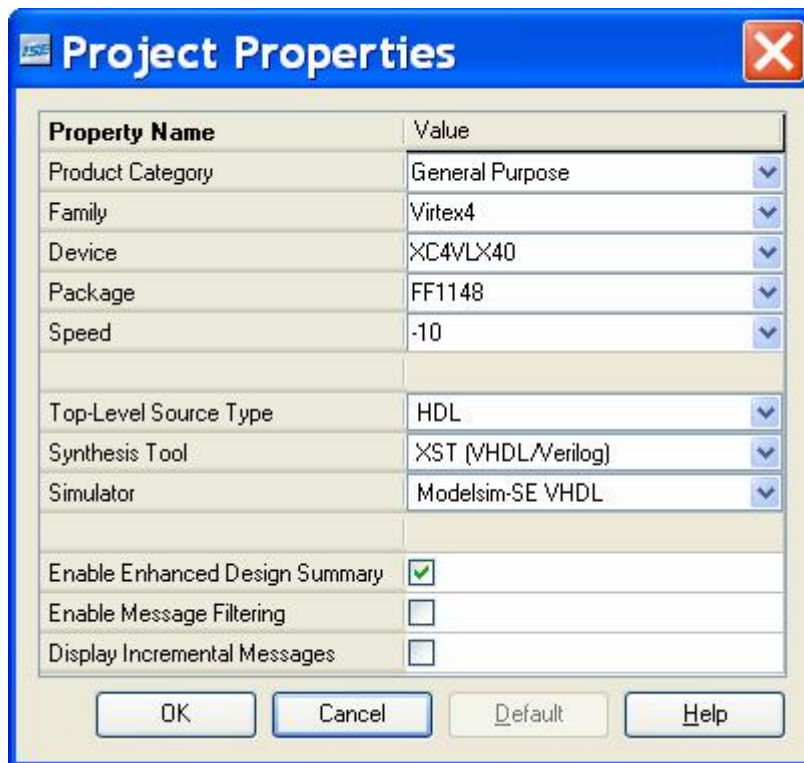
Customizing your System Generator project

When first opening your System Generator project, you will see that it has been set up with the synthesis tool, device, package, and speed grade that you specified in the System Generator block. To change these settings, open the Project Navigator properties dialog, right-click on the device and default package at the top of the sources window, and select **Properties**.



Launching Project Navigator properties dialog

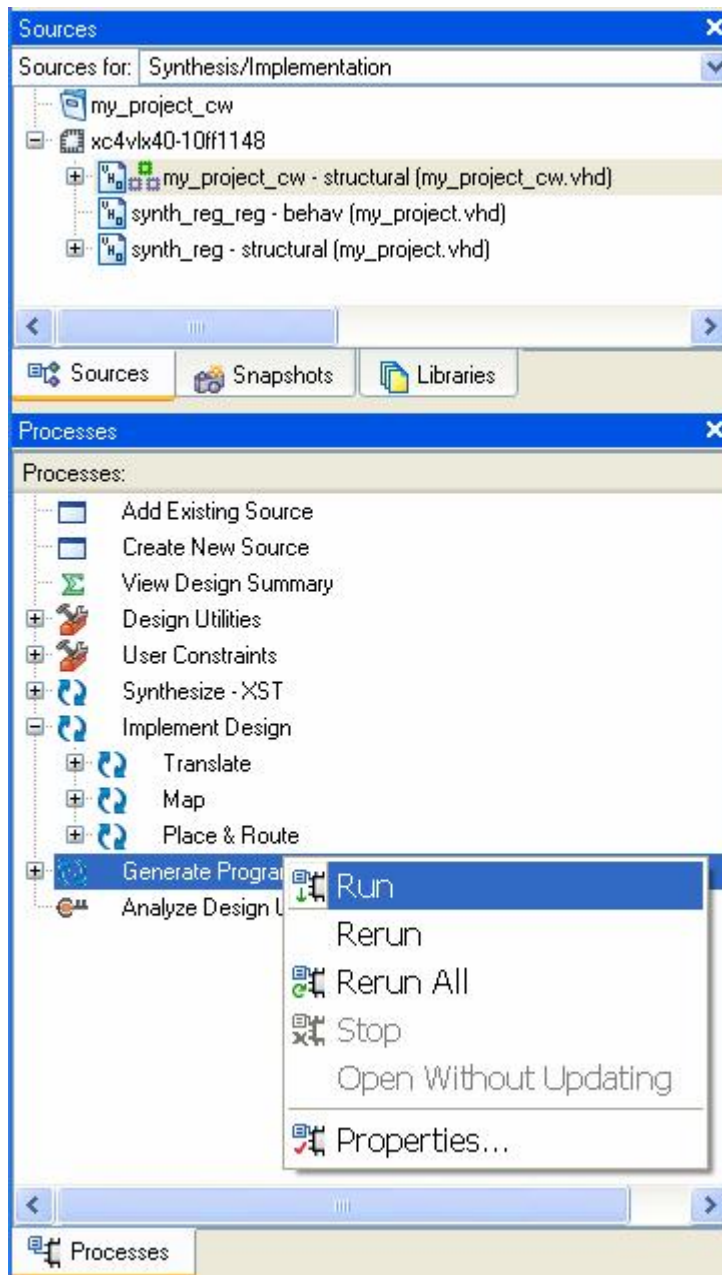
This brings up the Project Properties window. From this window, you can change your part, package, speed, and synthesis compiler. Note that if you change the device family, the Xilinx IP cores that were produced by System Generator must be regenerated. In such a case, it is better if you return to the System Generator and re-generate your project.



Customizing Project Navigator properties

Implementing your design

You have many options within Project Navigator for working on your project. You can open any of the Xilinx software tools such as the Floorplanner, Constraints Editor, report viewers, etc. To implement your design, you can simply instruct Project Navigator to run your design all the way from synthesis to bitstream. In the **Sources** window, select the top level HDL module in your design. In our example the top level HDL module is named `my_project_cw - structural`. The **Processes** window shows the processes that can be run on the top level HDL module.

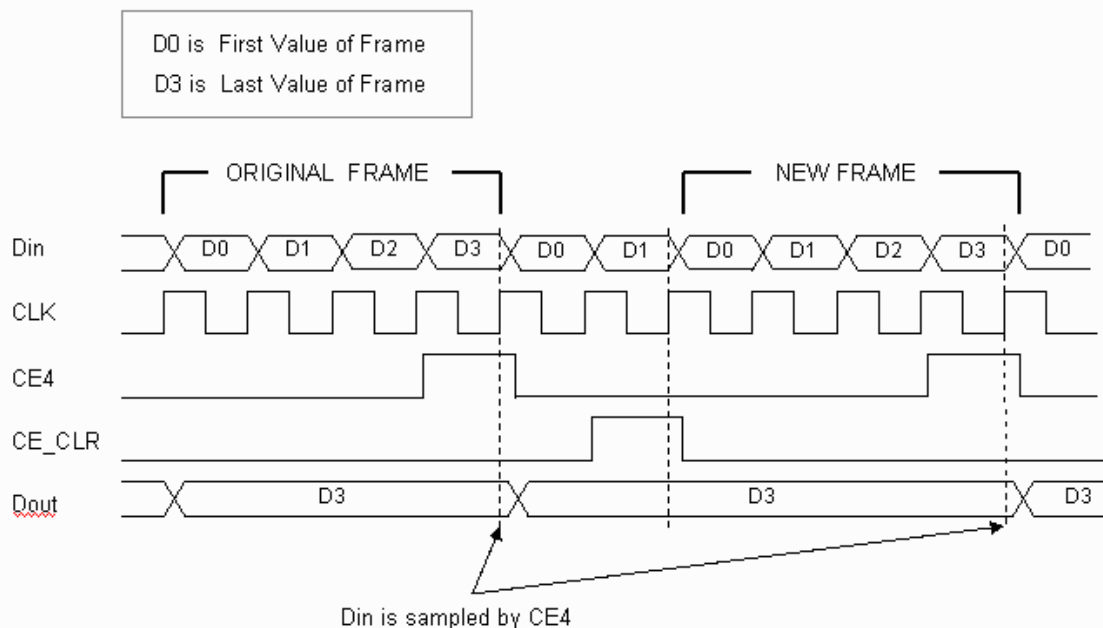


In the **Processes** window, if you right-click on Generate Programming File and select Run, you are instructing Project Navigator to run through whatever processes are necessary to produce a programming file (FPGA bitstream) from the selected HDL source. In the messages console window, you see that Project Navigator is synthesizing, translating, mapping, routing, and generating a bitstream for your design.

Now that you have generated a bitstream for your design, you have access to all the files that were produced on the way to bitstream creation.

Resetting Auto Generated Clock Enable Logic

System Generator provides a bit and cycle accurate modeling of FPGA hardware in the Simulink environment. To provide cycle accurate behavior with multirate models, System Generator uses a single clock accompanied by clock enables (ce) to keep various sample domains in sync. Multirate clocking is described in detail in compilation results. System Generator models are often included as part of a bigger system design which need dynamic control for specifying the beginning of data path sampling. To allow this control within a bigger framework System Generator block provides an optional `ce_clr` port in the top level HDL clock wrapper for resetting the clock enable generation logic. The figure below shows the reset of the CE4 signal generation logic after `ce_clr` signal is de-asserted.



The effect of `ce_clr` signal cannot be simulated using the original System Generator design. To model this behavior within Simulink follow the steps below:

- Select "Provide clock enable clear pin" and "NGC Netlist" compilation option on the System Generator block.
- Press the generate button on the System Generator block.
- Run the following command from the Matlab console to produce the post translate VHDL netlist. Use "-ofmt verilog" with netgen for generating Verilog netlist.

```
>> !netgen -ofmt vhd1 ./<target_directory>/<design_name>_cw.ngc
```
- Bring in the post translate VHDL/Verilog file as a black box within Simulink and use HDL co-simulation to model the effect of asserting `ce_clr` signal on your design.

ce_clr and Rate Changing Blocks

The `ce_clr` signal changes the sampling phase of all the multi-sample data signals. This behavior has the potential of changing the functionality of all rate changing blocks which rely heavily on the `ce` signal to have a periodic occurrence. The various rate changing blocks and their behavior with regards to the de-assertion of the `ce_clr` signal is explained in the table below. These blocks were characterized by importing and simulating the post translate HDL model as a black box.

Block Name	Synchronized to ce_clr	Synchronized to ce after ce_clr deasserted (1 sample cycle delay)	Behavior after ce_clr is deasserted and the next ce pulse
Down Sampler with Last Value of frame	Yes	N/A	The last sampled value is held till the new ce signal arrives.
Down Sampler with First Value of frame	No	No	Re-synchronization does not occur after de-assertion of the ce_clr signal.
Up Sampler with copy samples	Yes	N/A	In hardware, this block is implemented as a wire.
Up Sampler with zeros inserted	No	Yes	The last value (zero or sample) is held till the next destination ce signal arrives.
Time Division Multiplexer	No	Yes	The TDM block samples through all the remaining input channels and then sets the output to 0 till the next ce arrives. The new ce signal resynchronizes the output to the new frame definition.
Time Division Demultiplexer	No	Yes	The TDD block holds the output channels to the same value till the next ce signal arrives. The new ce signal resynchronizes the output to the new frame definition.
Parallel to Serial	No	Yes	The p2s block samples through all the remaining data words and then holds the output to the last sampled word until the next ce arrives. The new ce signal starts the conversion of the parallel data stream to a serial one.
Serial to Parallel	No	Yes	The s2p block holds the output when the ce_clr is asserted. When de-asserted,

			the input is sampled on the last value of the input sample frame, and the output occurs on the first ce pulse corresponding to the output rate.
Addressable Shift Register (ASR)	No	Yes	The ASR block will hold the values in the shift register when ce_clr is asserted. When de-asserted, the stored values will be shifted out, and new data will be put into the shift register.
Polyphase FIR	No	No	Interpolating or Decimating FIR does not work with the ce_clr signal unless the optional reset port is used to reset the FIR after the ce_clr is de-asserted.

ce_clr usage recommendations

Based on the above analysis, the ce_clr signal can be used if the following recommendations are adhered to:

- Replace down sampler blocks with first value of frame behavior with an equivalent circuit using down sampler block with last value of frame selected.
- Design for N clock cycles of invalid data after ce_clr is de-asserted, where N is the slowest ce associated with the block.
- Design the model to always use down sampler with last value of frame and up sampler with copy samples.
- If N cycle invalid data is not desired replace parallel to serial, serial to parallel, time division multiplexer and time division demultiplexer block with an equivalent circuit built out of a counter, mux and up/down sampler blocks. The equivalent design circuit should also have a reset port pulled to the top level and connected to the same signal driving the ce_clr port.
- Counters used in performing operations like “multiply-accumulate” should always be reset using a combination of user reset which is tied to the ce_clr signal and ce signal extracted from the clock enable probe block.
- Always verify the effect of ce_clr signal on the design by by importing and simulating the post translate HDL model as a black box.

Hardware Software Co-Design in System Generator

Hardware Software Co-Design Technologies in System Generator

System Generator provides three ways for processors to be brought into a model; processors can be imported through a Black Box block, a PicoBlaze block and an EDK Processor block.

Black Box

The Black Box approach provides the largest degree of flexibility for the users, at the cost of design complexity. A user can interface any processor HDL into a System Generator design in this manner. All ports and buses on the processor can be exposed to the System Generator diagram, and the user is free to engineer the required connectivity between the processor and other System Generator blocks. The user also has complete control over software compilation issues. Please refer to the documentation on the Black Box interface for more information.

PicoBlaze

The PicoBlaze block provides the smallest degree of flexibility but is the least complex to use. The Xilinx PicoBlaze Microcontroller block implements an embedded 8-bit microcontroller using the PicoBlaze macro, and exposes a fix interface to System Generator. Ordinarily, a single block ROM containing 1024 or fewer 8 bit words serves as the program store. Users may program the PicoBlaze using the PicoBlaze Assembler language. This flow is documented in the example entitled Designing PicoBlaze Microcontroller Applications.

EDK Processor

The EDK Processor block provides an interface to MicroBlaze processors created using the Xilinx Embedded Development Kit. The EDK Processor block allows for Shared-Memory blocks to be associated with a processor. Once associated, that memory can be read or written in software running on the MicroBlaze processor. This flow is documented in “Integrating a processor with custom logic”.

The EDK Processor block can import a MicroBlaze specified through an EDK project created using Xilinx Platform Studio and Base System Builder. Alternatively, a System Generator design with an EDK Processor block can also be exported into an EDK project. The export process creates an FSL-based pcore that can be added to any EDK system with FSL interfaces; MicroBlaze and PowerPC with APU-FSL bridges.

Integrating a Processor with Custom Logic

Integrating a processor with a piece of user defined logic is typically a fairly involved process. The communications between a processor and a custom piece of hardware often occurs over a shared bus. Additionally, the information conveyed frequently consists of different types of data; for instance data for processing, data denoting the status of the hardware or data affecting the mode of operation. Organizing how this data is transferred between the processor and custom logic is a tedious and error prone process that would benefit from automation. Furthermore connectivity is only half of the problem, writing software to communicate with custom logic can also be challenging.

The EDK Processor block provides a solution to both these problems through automation. The EDK Processor block encourages the interface between the processor and the custom logic to be specified via shared-memories. Shared-memories are used to provide storage locations that can be referenced by name. This allows a memory-map and the associated software drivers to be generated.

Please refer to the EDK Processor block documentation for information on the use of the block. The sections that follow describe the memory-map and software features of the drivers that are generated.

* Memory-map creation	Explains the memory map generated when shared memories are added to a processor.
* Hardware generation	Documents the different hardware generation options.
* Hardware co-simulation	Explains how to create a hardware co-simulation model for the EDK Processor block.
* Generating software drivers	Documents how software drivers are created.
* Writing software for EDK Processors	Documents the process of writing software to control hardware created in System Generator.

Memory-map creation

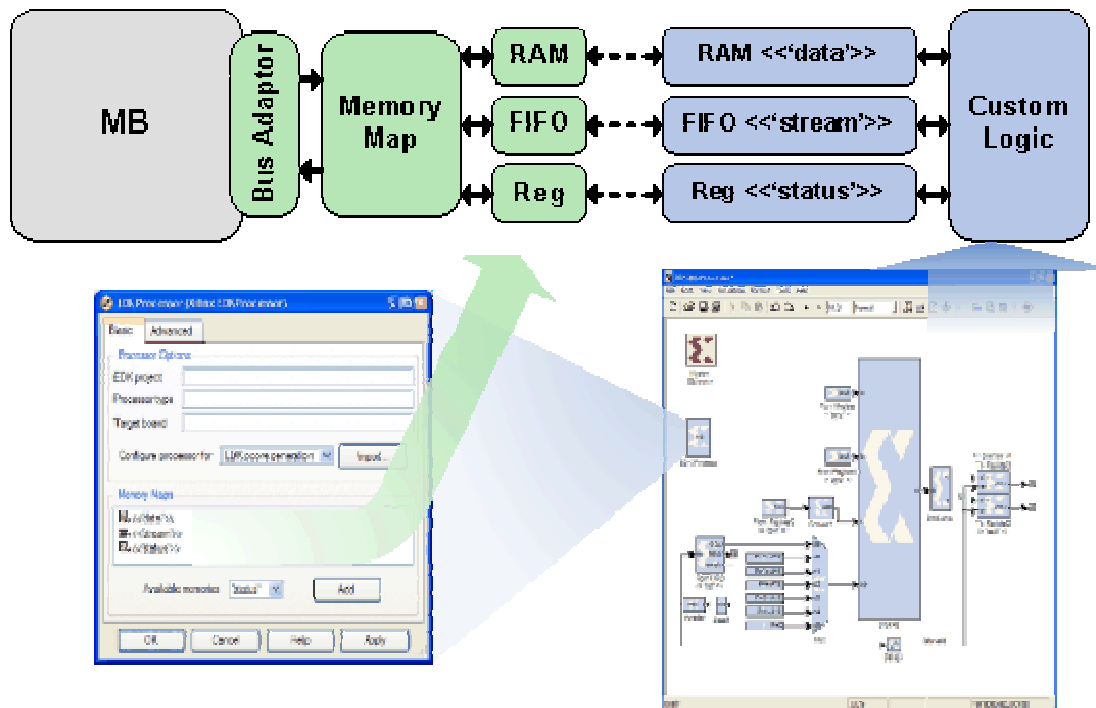


Figure 1. EDK Processor block memory map.

A System Generator model is shown on the bottom right of Figure 1. The System Generator model corresponds to custom logic that will be integrated with the MicroBlaze processor. In the construction of the model, shared-memories were used in locations where software access was required. For instance, the status of the hardware might be kept in a register. To make that status information visible in the processor, the register is replaced by a named shared-register. Naming the shared-register “status” gives the name of the memory context that will be useful later on during software development.

The block GUI of the EDK Processor block allows these shared-memories to be added to the memory map of the processor (bottom left of Figure 1). The block diagram at the top of Figure 1 shows the flow of data. When a shared-memory is added to the memory-map of the processor the EDK Processor block creates the corresponding matching shared-memory. This shared-memory is attached to the memory-map that is generated for that EDK Processor block. Next a bus adaptor is used to connect that memory-map to the MicroBlaze processor.

When hardware is generated, each shared-memory pair is implemented with a single physical memory. The implementation for each class of shared memory is documented in the section entitled Shared Memory Support, found in the Chapter called Using FPGA Hardware in the Loop.

Hardware Generation

The EDK Processor block supports two modes of operation: EDK pcore generation or HDL netlisting. The different mode of operation can be chosen from a list-box in the EDK Processor block’s GUI.

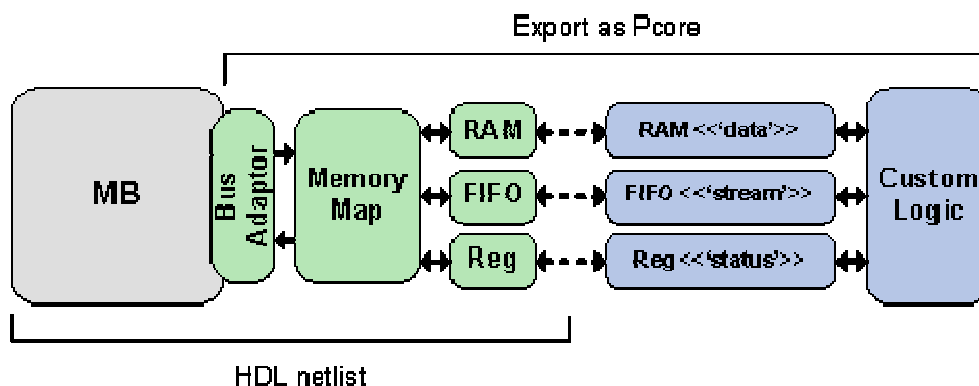


Figure 2. EDK Processor block modes of operation

EDK PCore generation mode

The Xilinx Embedded Development Kit (EDK) allows peripherals to be attached to processors created within the EDK. These peripherals can be packaged as a pcore. Each pcore contains a collection of files describing the peripheral's hardware description, software drivers, bus connectivity and documentation.

When set in EDK pcore generation mode and used with the EDK Export Tool (selected via the System Generator block), System Generator is able to create a pcore from the given System Generator model. Figure 2 shows the part of the model that will be created as a pcore. When set in this mode, the assumption is that the MicroBlaze added to the model is just a place-holder. Its actual implementation will be filled in by the EDK when the peripheral is finally added into an EDK project. As such the pcore that is created consists of the custom logic, the generated memory-map and their virtual connections to the custom logic, and the bus adaptor.

HDL netlist mode

An EDK processor can also be brought into a System Generator model when HDL netlisting mode is selected. The EDK Processor block can be set to HDL netlisting mode only when an EDK project is supplied to the block. When in HDL netlisting mode, the processor described in the EDK project will be imported into System Generator as a black box. The supplied EDK project is also augmented with the bus interfaces necessary to connect the System Generator memory-map to the processor. During netlisting, the MicroBlaze and the generated memory-map hardware are both netlisted into hardware. This mode can be used for software-based simulation of the processor, or hardware in the loop simulation. Please refer to "Various hardware-software simulation techniques" for more information on these two modes of simulation.

Hardware co-simulation

Currently the EDK Processor block provides both software-based simulation and hardware based simulation through hardware co-simulation. Creation of a hardware co-simulation block follows the standard co-simulation flow described in "Using FPGA Hardware in the Loop".

When hardware co-simulation detects a processor, the software tab in the co-simulation block's GUI becomes active.

Users may use the EDK's XPS tool to write and compile their software. However before simulation can begin, the "Compile and update bitstream" button in the co-simulation block's software tab must be used to put the compiled c-code into the bitstream.

When used in conjunction with a hardware-board supported by network-based hardware co-simulation, it is possible to free up the JTAG port on the FPGA and use that for software debug with XMD.

Generating software drivers

In both modes of operation, the software driver templates are automatically created when a memory-map is generated. The driver templates are only elaborated during the compilation of software libraries by the EDK. This can be accomplished from the EDK. Once the software libraries have been compiled, the drivers can be referenced and the software documentation can also be accessed.

When the EDK Processor block is placed in EDK pc core generation mode, the elaboration of the software drivers depend on the instance name of the pc core. For example, a pc core created from System Generator may be called `sysgen_fft_sm`. This is the name of the peripheral. Since more than one of these peripherals can be added to an EDK processor, each instance of the peripheral needs a unique name. The EDK automatically assigns a numeric postfix to a peripheral's name. So when the peripheral is first added, it might be called `sysgen_fft_sm_0`, this is the instance name of the peripheral. A user can change the instance name of a peripheral. Please refer to the EDK XPS documentation for further information.

When the EDK Processor block is placed in HDL netlisting mode and an EDK Project is imported into the System Generator model, System Generator automatically places a special peripheral into the EDK project. The peripheral provides the connectivity between the MicroBlaze and the System Generator model. This peripheral is given the instance name `xlsg_iface`. Only one instance of this peripheral is allowed; an EDK Project can only be associated with one EDK Processor block.

Writing software for EDK Processors

Software drivers and documentation for a peripheral is elaborated after all software libraries have been compiled in the EDK. After compilation, the software documentation can be accessed from the EDK.

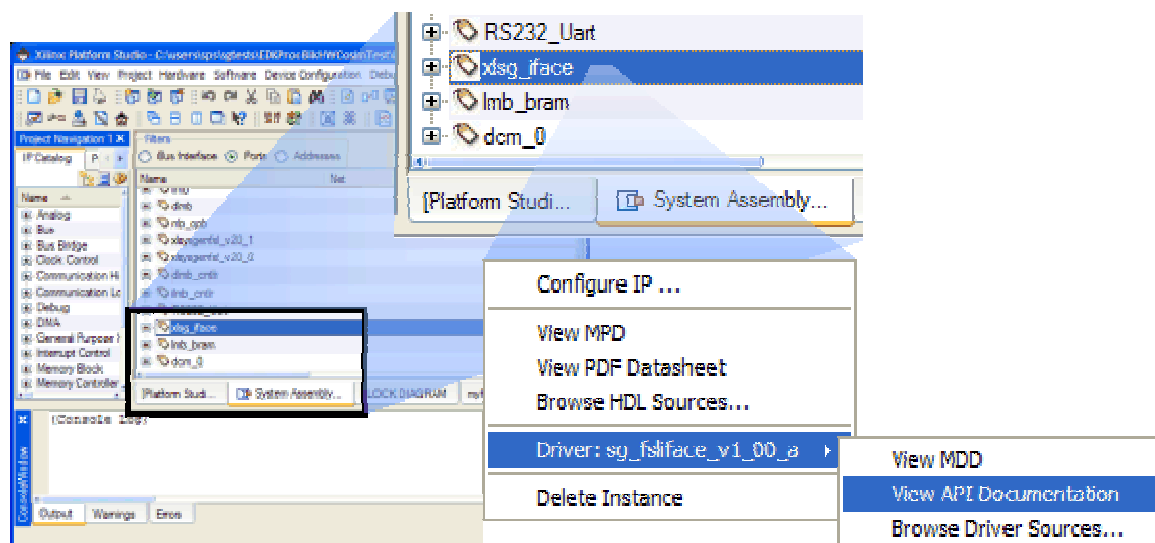


Figure 3. Accessing driver API documentation from the EDK

Figure 3 shows a screen capture of the EDK XPS tool. Locate the System Generator peripheral from the “System Assembly” view and right click for a pop-up menu. Select “View API Documentation” to bring up the documentation. If this option is not available, the drivers need to be compiled. This can be

done in XPS by selecting the menu option “Software->Generate Libraries and BSPs”.

Please refer to the generated documentation for header file information, driver calls, memory-maps and also example code.

The generated software drivers contain six basic functions for accessing shared-memories. In the table below, <inst> refers to the instance name given to the peripheral.

```
int <inst>_Read      (unsigned int memName,
                    unsigned int addr,
                    unsigned int* val);
int <inst>_ArrayRead (unsigned int memName,
                    unsigned int startAddr,
                    unsigned int transferLength,
                    unsigned int** valBuf);
int <inst>_Write     (unsigned int memName,
                    unsigned int addr,
                    unsigned int val);
int <inst>_ArrayWrite (unsigned int memName,
                    unsigned int startAddr,
                    unsigned int transferLength,
                    const unsigned int* valBuf);
```

EDK Support

Importing an EDK Processor

A processor created using the Xilinx Platform Studio (XPS) tool, found in the Xilinx EDK suite of tools, can be imported into a System Generator model using the EDK Import Wizard.

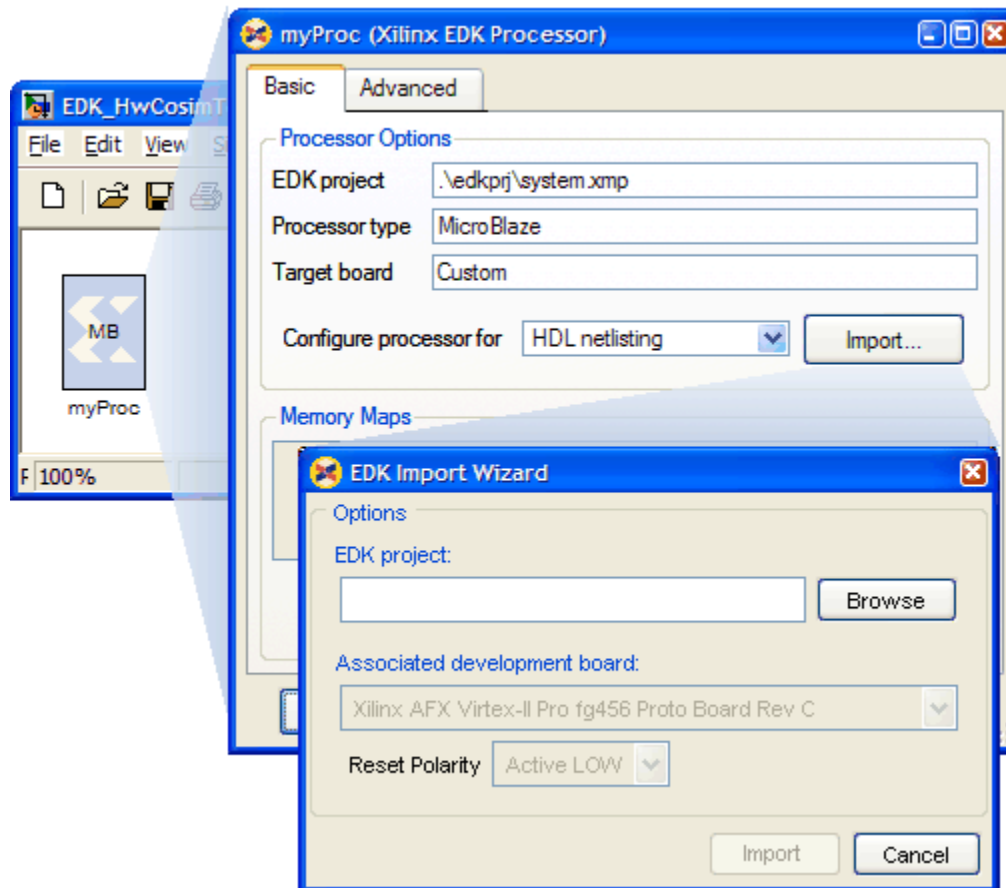


Figure 1. Launching the EDK Import Wizard

There are two ways for launching the EDK Import wizard in the EDK Processor block in two ways: pressing the “Import...” button, or by selecting “HDL netlisting” when the “EDK project” field is empty.

Note that when importing the EDK Project into System Generator, there are modifications made to the EDK project, which are described in the following section.

EDK Board Descriptions

The first time the wizard is run all the available EDK board description (XBD) files are located and cached in System Generator. If new EDK board description files are added at a later date, simply delete this cache file and the EDK import wizard will rescan the EDK for new board descriptions. This cache file can be found in:

`%SYSGEN\data\xbdlist.xml`

EDK Import Wizard

When the wizard starts up, it prompts the user for an EDK project file (xmp file).

Next, the user has to identify the development board that the EDK project board is associated with. If the project was created using Base System Builder, this will be automatically detected when you browse to an EDK project. If a “Custom Board” is selected, the user must also supply the polarity of the reset signal.

Clicking the “Import” button will begin the import process. Note the import process will alter your EDK project to work inside of System Generator. If you wish to retain an unadulterated version please make a copy before importing.

When an EDK project is imported into System Generator, the EDK project is augmented with a pair of FSLs and a pcore called `xlsg_iface`. The FSLs are added to allow data transfer between the System Generator design and the processor. The pcore is added to expose the FSL signals to System Generator, and also to provide software drivers. The MHS and MSS files in the EDK project will be altered. Following that, the HDL files that describe the processor will be generated and linked to your System Generator project.

Making changes to processor hardware after an import

After importing an EDK project, further changes to the hardware inside of the EDK will not be reflected inside of System Generator. In other words, the hardware makeup of the processor is now fixed. If changes to the processor hardware are to be made, the EDK project must be re-imported using the EDK Import Wizard.

It is recommended to re-import an EDK project when it is changed. The EDK Import Wizard can detect FSLs and pcors added by the previous import and will not include additional hardware.

Limitations

Currently the wizard can only import single processor projects. Only the MicroBlaze processor is supported. Peripherals added to the processor cannot conflict with the resources used by other System Generator services. For instance, if network-based hardware co-simulation is used, the EDK project cannot make use of the peripherals using the Ethernet MAC.

Exposing processor ports to System Generator

The preferred mechanism for getting data to and from the processor and System Generator is via shared-memories. It is however possible to expose ports on the top-level of the processor to System Generator.

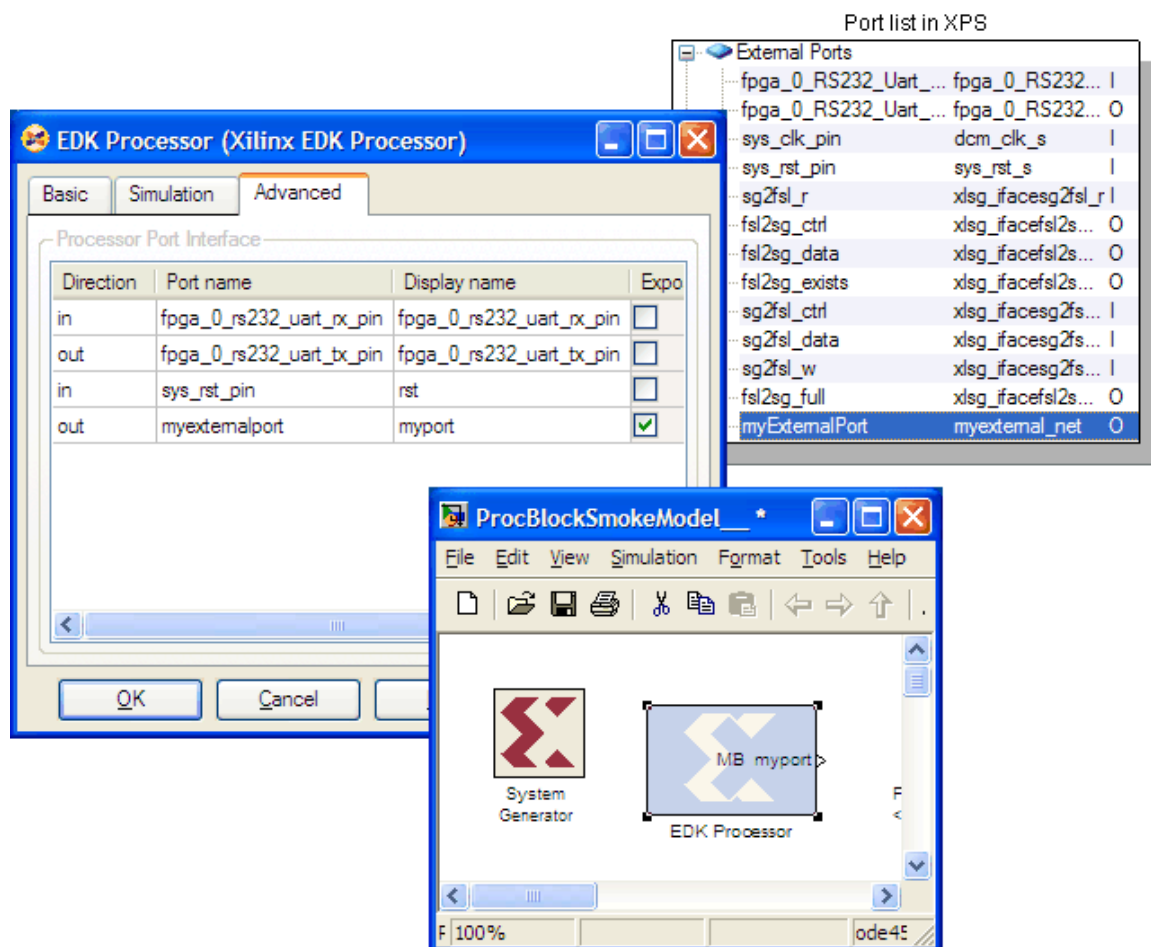


Figure 1. Exposing top level ports

The top right box of Figure 1 shows a snippet from an EDK project in XPS. The external port list has amongst other ports, a user defined port called “myExternalPort”.

After importing the EDK project, open up the processor’s block GUI in System Generator. Select the “Advanced” tab to reveal the processor port interface table.

The port list shows all the top-level ports available on the processor. This port list has been filtered to remove clock ports and also signals used by System Generator to implement the memory-map interface. In this example, the RS232 ports, sys_rst_pin and myexternalport are shown to be ports that can be exposed to the toplevel of the System Generator block. Selecting the “expose” check box will cause the port to be exposed on the EDK Processor block. The display name of the port can be changed, should the original name be too long, as shown in Figure 1.

This mechanism allows ports from the processor to be directly exposed to the System Generator design without going through the memory-map generated by System Generator. Users may choose to do this

to expose rest ports on the processor, or to expose interrupt ports directly to the System Generator diagram.

Exporting a pcore

System Generator designs containing an EDK Processor block can be exported as an EDK pcore using the “EDK Export Tool” compilation target on the System Generator block.

Before exporting to the EDK as a pcore, the EDK Processor block must be configured for “EDK pcore generation”. This can be done by opening the EDK Processor block GUI and selecting the relevant drop down option in the “Configure processor for” parameter.

Please refer to the EDK Export Tool documentation for more information.

Designing with Embedded Processors and Microcontrollers

Designing PicoBlaze Microcontroller Applications

The PicoBlaze block in System Generator implements an 8-bit microcontroller. Applications requiring a complex, but non-time critical state machine as well as data processing applications are candidates to employ this block. The microcontroller is fully embedded into the device and requires no external support. Any additional logic can be connected to the microcontroller inside the device providing ultimate flexibility.

PicoBlaze Overview

This tutorial uses PicoBlaze 3 (hereto referred to simply as PicoBlaze), which is optimized for low resource requirements. A memory block is used as a program store for up to 1024 instructions.

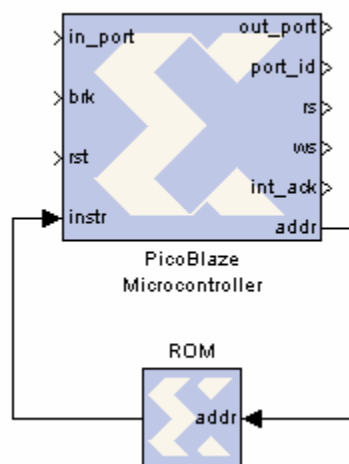


Figure 1: Block Diagram

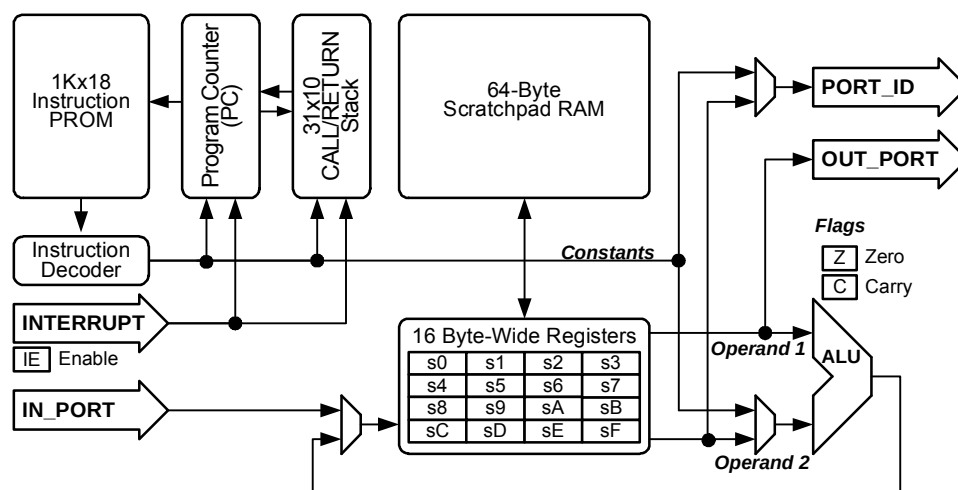
Signal	Direction	Description
in_port[7:0]	Input	Input Data Port. During an INPUT operation, data is transferred from the port to a register.
brk	Input	Interrupt. Must be at least two clock cycles in duration.
rst	Input	Reset.
instr[17:0]	Input	Instruction Input.
out_port[7:0]	Output	Output Data Port.
port_id[7:0]	Output	Port Address.
rs	Output	Read Strobe.
ws	Output	Write Strobe.
addr[9:0]	Output	Address of the next instruction.

ack	Output	Interrupt Acknowledge.

Figure 2: PicoBlaze Port Description

Architecture Highlights

- Predictable performance, two clock cycles per instruction.
- 43 - 66 MIPS (dependent upon device type and speed grade).
- Fast interrupt response.
- 96 slices, 0.5 to 1 block RAM.
- 16 8-bit general-purpose registers.
- 64-byte internal RAM.
- Internal 31-location CALL/RETURN stack.
- 256 input and 256 output ports.


Figure 3: PicoBlaze Architecture

PicoBlaze Instruction Set Architecture

PicoBlaze is a hardware centric microcontroller, which can be programmed using assembly code. It supports a program length up to 1024 instructions. Requirements for larger program space are typically addressed by using multiple microcontrollers.

16 General Purpose Registers

There are 16 8-bit general-purpose registers specified 's0' to 'sF'.

ALU

The Arithmetic Logic Unit (ALU) provides operations such as **add**, **sub**, **load**, **and**, **or**, **xor**, **shift**, **rotate**, **compare**, and **test**. The first operand to each instruction is a register to which the result is stored. Operations requiring a second operand can specify either a second register or an 8-bit constant value.

Flags and Program Control

The result of an ALU operation determines the status of the zero and carry flags. The zero flag is set

whenever the result is zero. The carry flag is set when there is an overflow from an arithmetic operation. The status of the flags can be used to determine the execution sequence of the program using conditional program flow control instructions such as **jump** and **call**.

Input/Output

There are 256 input ports and 256 output ports. The port being accessed is indicated by an 8-bit address value provided on `port_id`. The port address can be specified in the program as an absolute value or indirectly specified as the contents of a register. During an **input** operation, the value provided to `in_port` is transferred into any of the 16 registers. During an output operation, a value is transferred from a register to `out_port`.

Interrupt

The processor provides a single interrupt input port, `brk`. When interrupts are enabled, setting `brk` to 1 causes the program counter to be set to memory location 0x3FF, where a jump vector to the interrupt service routine is stored. At this time, a pulse is generated on the `int_ack` port, the control flags are preserved and further interrupts are disabled. The **return** instruction ensures that the end of an interrupt routine restores the status of the control flags and specifies if future interrupts should be enabled.

For extensive details regarding the feature and instruction set, please refer to application note.

Using PicoBlaze in System Generator

In this example, you modify a PicoBlaze program that alters the output frequency of a Direct Digital Synthesizer (DDS) during an interrupt.

A Simulink model and PicoBlaze assembler code are provided but need modification.

1. From the MATLAB console, change directory to `<path_to_sysgen>/examples/picoblaze`. The default path to System Generator is `$MATLAB/toolbox/xilinx/sysgen`. The following files are located in this directory:
 - `Pico_dds.mdl` – An unfinished Simulink model
 - `Pico_code.psm` – Unfinished PicoBlaze code
2. Open `Pico_dds.mdl`.
3. Modify the design.
 - Find the PicoBlaze block in the Xilinx Blockset Library under Index or Control Logic and add it to the model where indicated. The default settings of the block do not give the same number of ports as is expected by the model. This will be corrected in the following step. You may need to resize the block to fit into the space allocated in the design.
 - Double-click the block and set Version to PicoBlaze 3. Turn off the option to display the internal state. Connect the ports to the existing lines in the model.

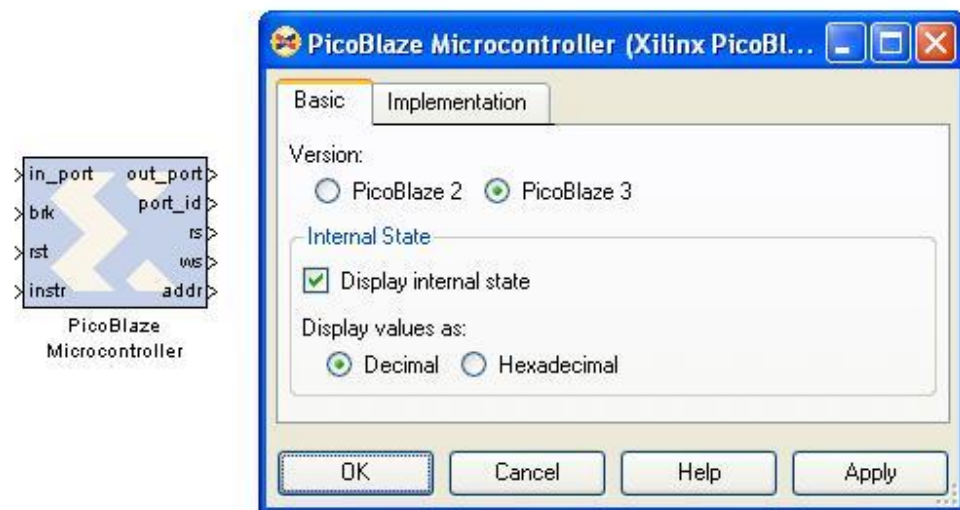


Figure 4: PicoBlaze MicroController

- Find the “PicoBlaze Instruction Display” block in the Index or Tools Library and add it to the model where indicated. Make sure it is connected properly (see Figure 6).
- Double-click the block and set the version to PicoBlaze 3. Check the Disable Display option. Disabling the display option allows simulation to run without the overhead of updating the block display.

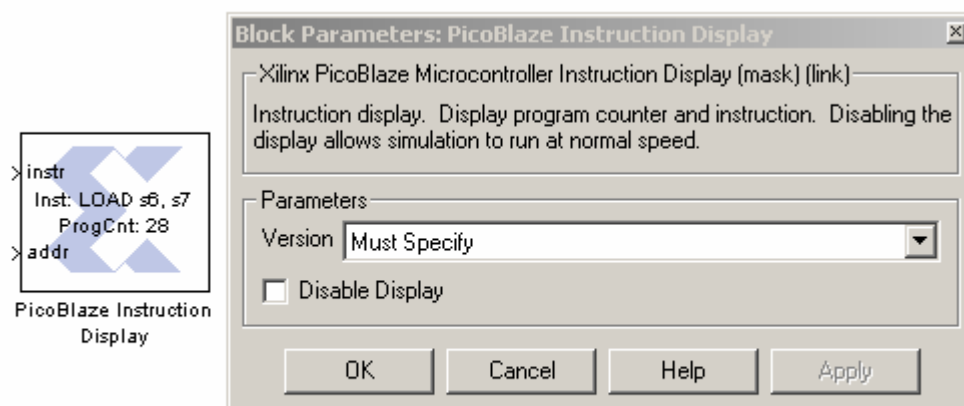


Figure 5: PicoBlaze Instruction Display Block

- Find the ROM block in the Memory Library and add it to the model where indicated. Flip the block by Right clicking on the block and selecting Format->Flip Block. Attach the ports to the existing lines.
- Change the Single-Step Simulation block to be in continuous mode by double clicking on the block.

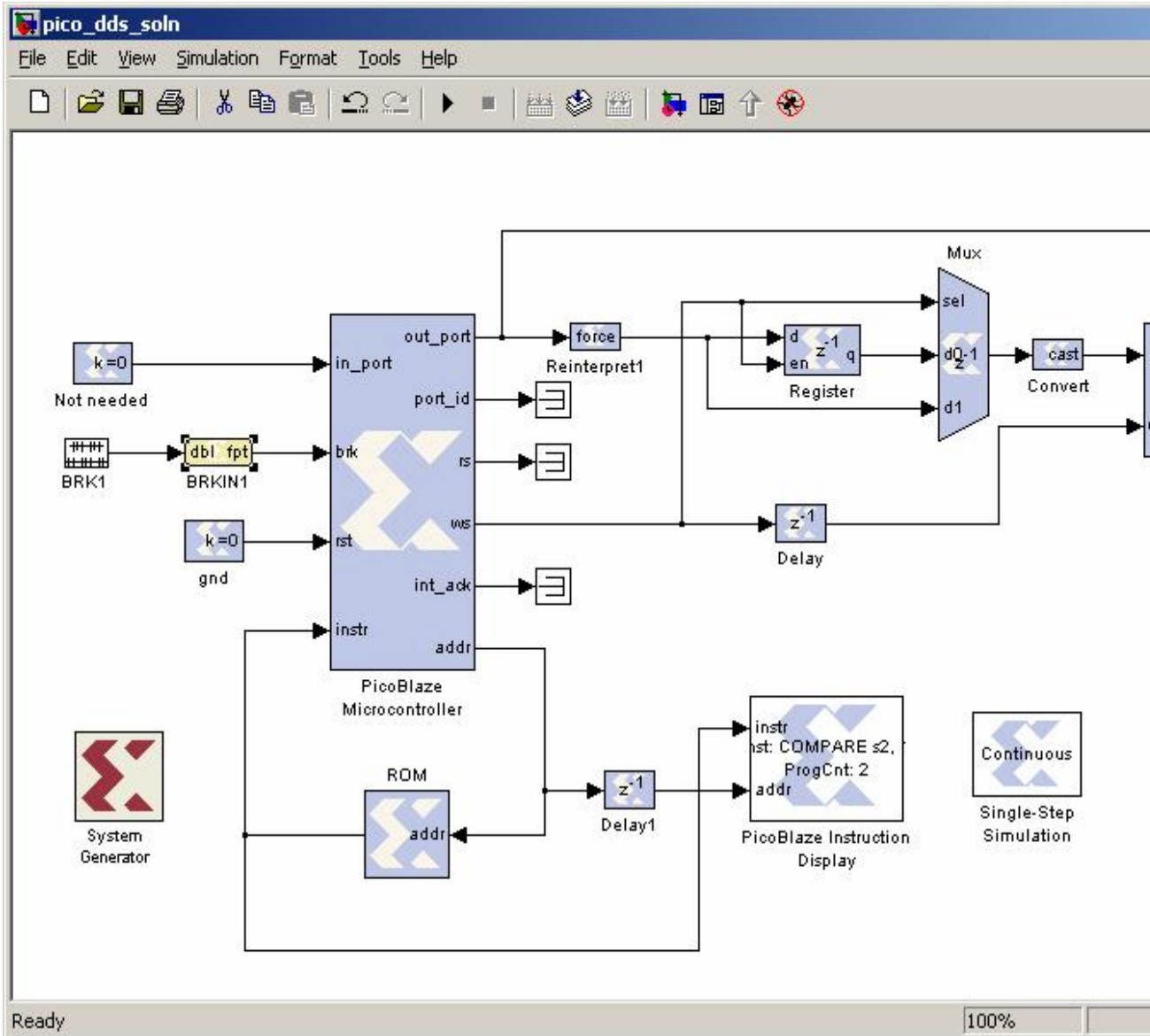
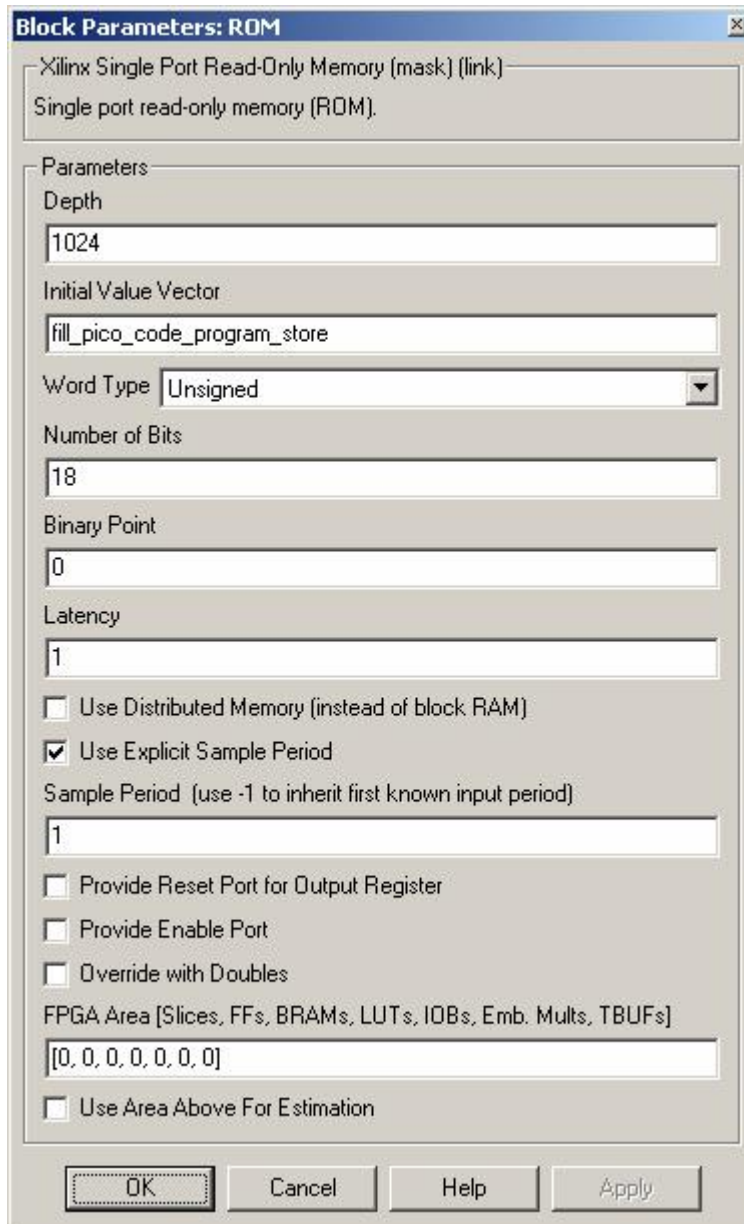


Figure 6: PicoBlaze model solution example

4. Configure the program store. Double click the ROM to do the following.

- The ROM block is used to store the PicoBlaze instructions. The depth of the ROM must be set to 1024. This is because the program uses interrupts and setting brk to 1 causes the program counter to be set to 0x3FF.
- As detailed in step 5, the code is assembled and produces an initialization file for the memory named `fill_pico_code_program_store.m`. Hence the ROM Initial Value Vector should be set to `fill_pico_code_program_store`.
- The **Word Type** should be unsigned and **Number of Bits** should be set to 18 with the **Binary Point** at 0.
- To increase the performance for synchronous designs, the **Latency** should be set to 1.

The ROM should be configured like this:



Block Parameters: ROM

Xilinx Single Port Read-Only Memory (mask) (link)
Single port read-only memory (ROM).

Parameters

Depth
1024

Initial Value Vector
fill_pico_code_program_store

Word Type Unsigned

Number of Bits
18

Binary Point
0

Latency
1

☐ Use Distributed Memory (instead of block RAM)

☒ Use Explicit Sample Period

Sample Period (use -1 to inherit first known input period)
1

☐ Provide Reset Port for Output Register

☐ Provide Enable Port

☐ Override with Doubles

FPGA Area [Slices, FFs, BRAMs, LUTs, IOBs, Emb. Mults, TBUFs]
[0, 0, 0, 0, 0, 0, 0]

☐ Use Area Above For Estimation

OK Cancel Help Apply

Figure 7: ROM Parameters

5. Edit the PicoBlaze assembly program.

- Open `pico_code.psm`.
- Add instructions as described in the `pico_code.psm` file. For detailed information about the PicoBlaze instruction set see XAPP627.

- Save the file.

6. Run the assembler to generate the memory initialization file.

- In the MATLAB command window, type

```
>> xlpb_as -p pico_code.psm
```

xlpb_as stands for Xilinx PicoBlaze Assembler.
A file named fill_pico_code_program_store.m was created.

7. Simulate the Simulink model.

- Run simulation by clicking on the Start Simulation Icon



Output should look like this:

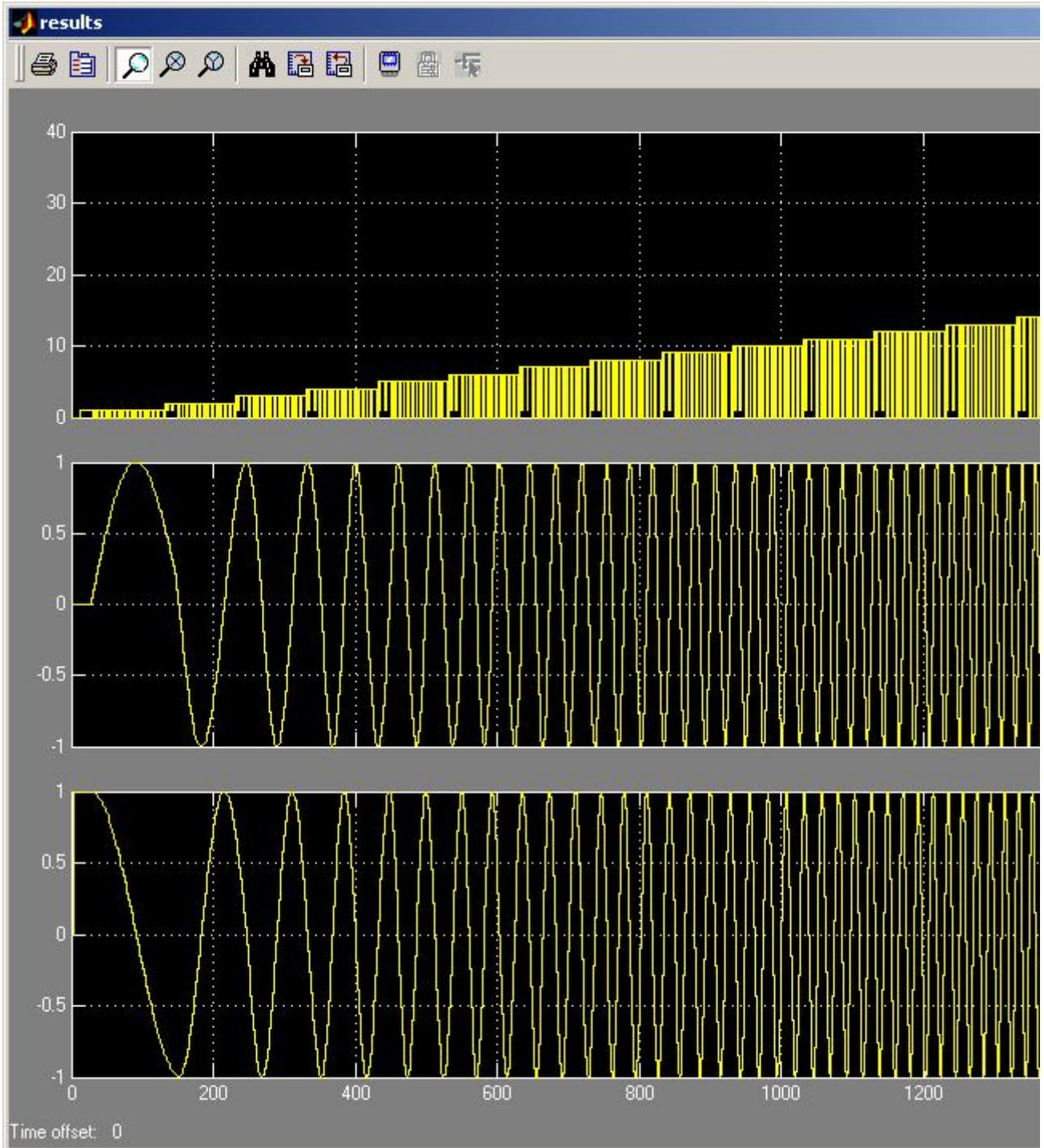


Figure 8: Output from pico_dds.mdl

Notice the sine wave frequency increasing proportionally to the phase increment.

8. Utilize Debug Tools

If the program is not working properly, there are several tools that can be utilized to ease debugging. Deselecting the **Disable Display** checkbox in the PicoBlaze

Instruction Display block causes the block to be activated, displaying the updated program counter and instruction each clock cycle. In conjunction with enabling the display, the registers and control flag values can be viewed by selecting the **Display Internal State** in the PicoBlaze Microcontroller block. Change the Single-Step Simulation block to single-step mode by double clicking on the block. Step through the simulation to debug.

Designing and Exporting MicroBlaze Processor Peripherals

The Xilinx Platform Studio (XPS) tool suite delivered with the Xilinx Embedded Development Kit (EDK) allows the development of user-designed hardware peripherals attached to the MicroBlaze processor. A user-designed hardware peripheral must be packaged up into a bundle of files with a specific structure as required by the XPS tool. These files describe the hardware implementation, the connection interface, and software attributes. The bundle of files constitutes what the EDK calls a pcore.

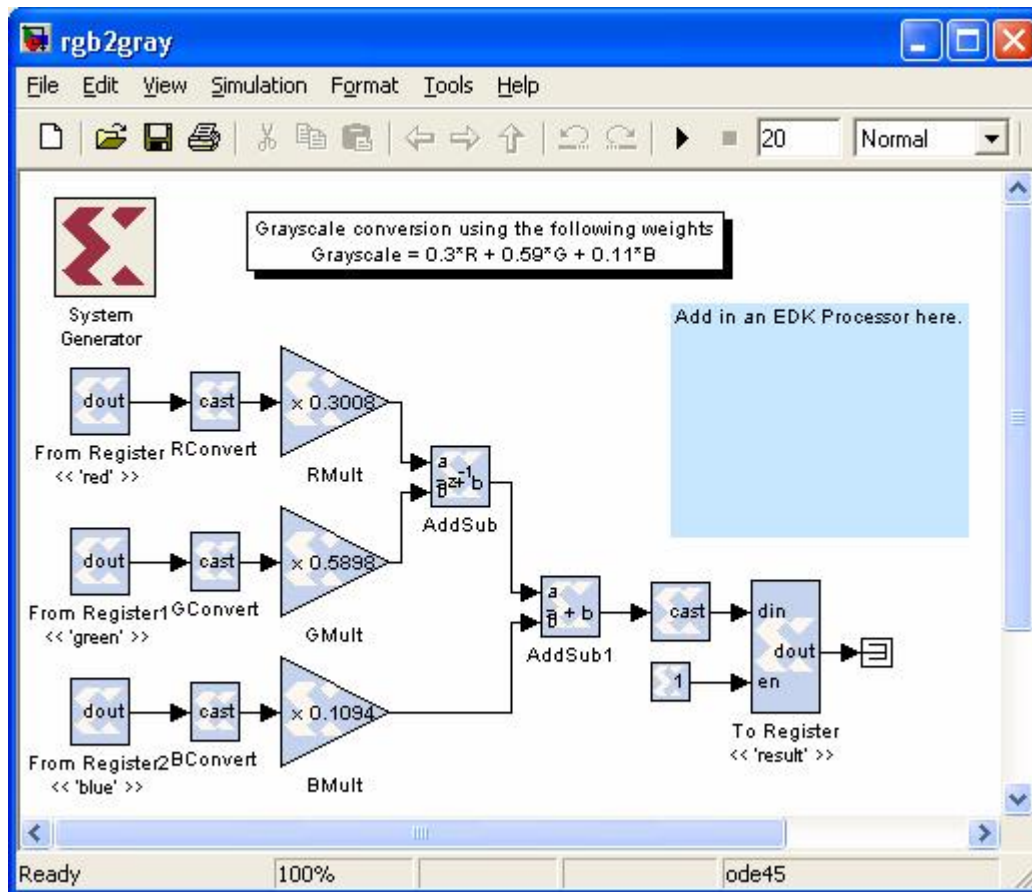
The EDK Processor block in conjunction with the EDK export tool allow MicroBlaze hardware peripherals to be designed and exported in System Generator. The exported design can be imported and used in an XPS project.

The following tutorial illustrates the creation of a pcore from System Generator.

The files used in this tutorial can be found in:

`<path_to_sysgen>\examples\EDK\rgb2gray`, where `<path_to_sysgen>` denotes the System Generator installation directory.

Create peripherals in System Generator

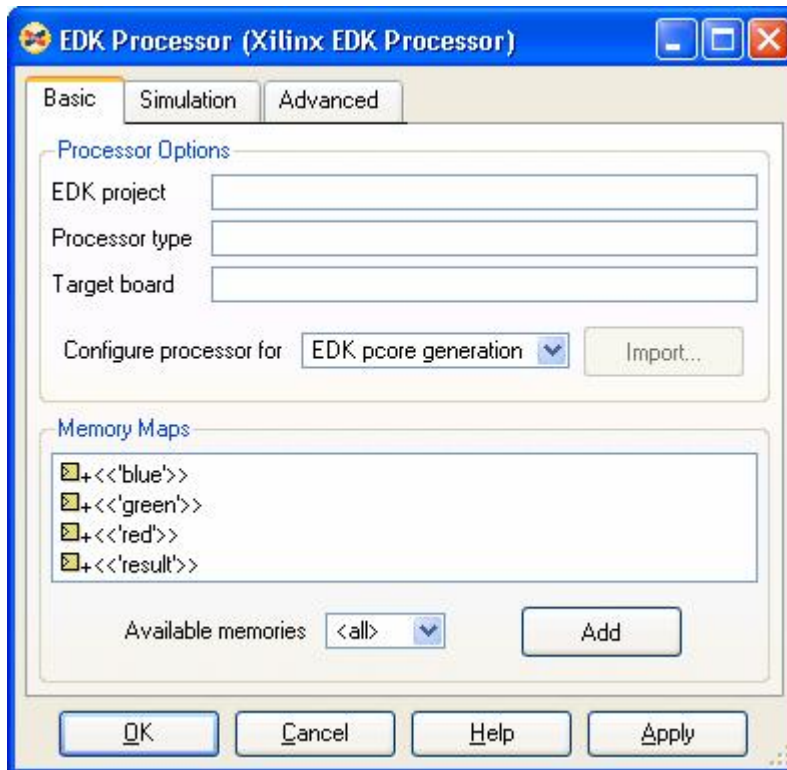


Open the rgb2gray model from <path_to_sysgen>\examples\EDK\rgb2gray. The peripheral contains three inputs, which are 32-bit red, green and blue pixel values. These values are scaled and summed to produce a result that represents the 32-bit grayscale value.

The red, green and blue values are sourced from three shared registers named 'red', 'green' and 'blue'. The result is written back to a shared register called 'result'.

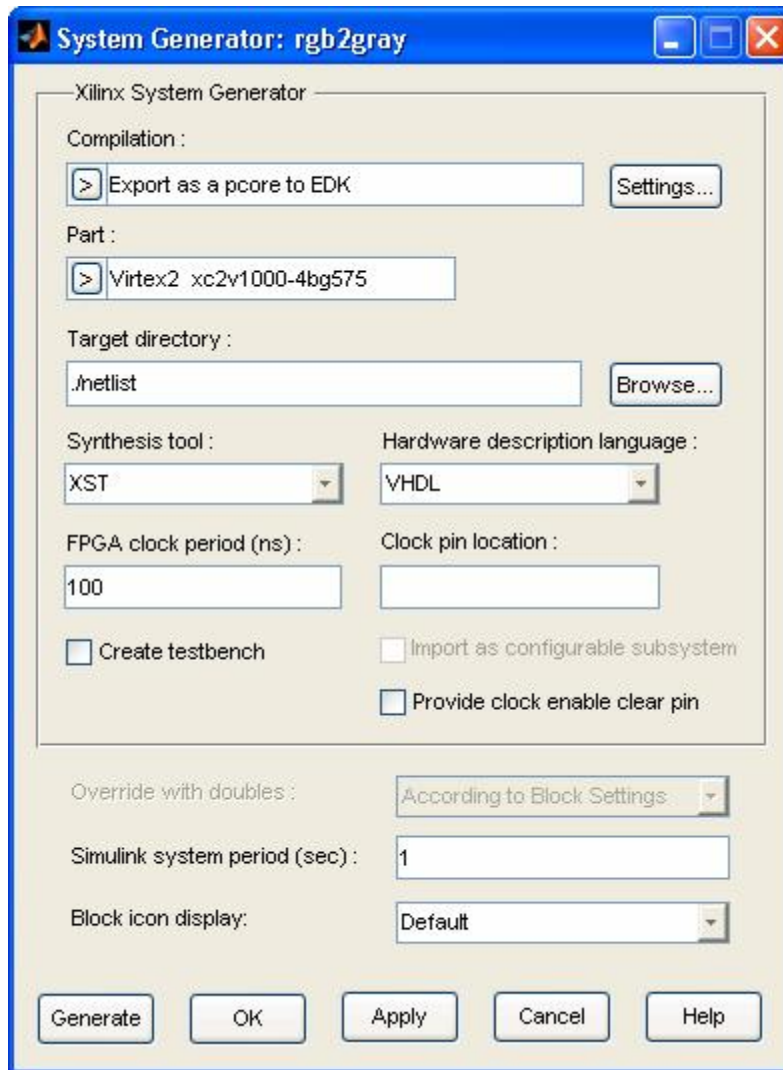
Export the pcore

The user needs to drag an EDK processor block into the model. To configure the processor block, double click on the block to bring up the block's dialog.



Add all available shared memories in the model to the EDK Processor by selecting *<all>* and clicking the Add button. Ensure that the EDK Processor block has been configured for *EDK pcore generation* in the “Configure processor for” drop-down menu. Dismiss the GUI by clicking the OK button. The EDK Processor will then create a memory map for the shared memories.

Double click on the System Generator block to open up the System Generator GUI. We will use the EDK Export Tool to create the pcore. Options in the EDK Export tool are more fully detailed in the Chapter on System Generator Compilation Types.



Set the Compilation type to be “Export as a pcore to EDK”. Clicking on the settings button will open up options for the compilation target. Accept the default settings so that the pcore is generated and exported into the model’s target directory.

Click on the Generate button to initiate the pcore export process.

Integrate the exported pcore in the EDK

We will now create an EDK project, and integrate the pcore into the EDK project. Information on how to create an EDK project can be found in the section entitled Using the EDK. Follow the directions there to create an EDK project.

Once the EDK project is created, copy the pcore that is created by System Generator into the EDK project. Since System Generator is instructed to place the pcore inside the target directory in the previous step, you should find a directory named pcore inside the target directory. Copy the contents of the directory into the corresponding pcore directory inside your EDK project. If your EDK project does not contain a pcore directory, create one before copying.

In the XPS menu select Project -> Rescan User Repositories. The pc core exported by System Generator will appear on the list of EDK Peripherals after the rescan.

Follow the directions in the section entitled Using the EDK for information on how to connect up a pc core to the MicroBlaze processor in the EDK tool.

After connecting up the pc core, compile the netlist by selecting Hardware -> Generate Netlist.

Write Software

Create a new software application in your EDK project. Again, information on how to do this can be found in the section entitled Using the EDK. Add the following code to your application and compile the software.

```
#include "xparameters.h"
#include "rgb2gray_sm_0.h"

int main (void) {
    int i;
    int gray, red, green, blue;

    print("-- Entering main() --\n\r");
    for (i=15;i<30;i++){
        red = i;
        green = i+10;
        blue = i+20;

        // Write RGB value to peripheral
        rgb2gray_sm_0_Write(RGB2GRAY_SM_0_RED,
                           RGB2GRAY_SM_0_RED_DIN, red);
        rgb2gray_sm_0_Write(RGB2GRAY_SM_0_GREEN,
                           RGB2GRAY_SM_0_GREEN_DIN, green);
        rgb2gray_sm_0_Write(RGB2GRAY_SM_0_BLUE,
                           RGB2GRAY_SM_0_BLUE_DIN, blue);

        xil_printf("R = 0x%x, G = 0x%x, B = 0x%x -- ",
                   red, green, blue);

        rgb2gray_sm_0_Read(RGB2GRAY_SM_0_RESULT,
                           RGB2GRAY_SM_0_RESULT_DOUT, &gray);

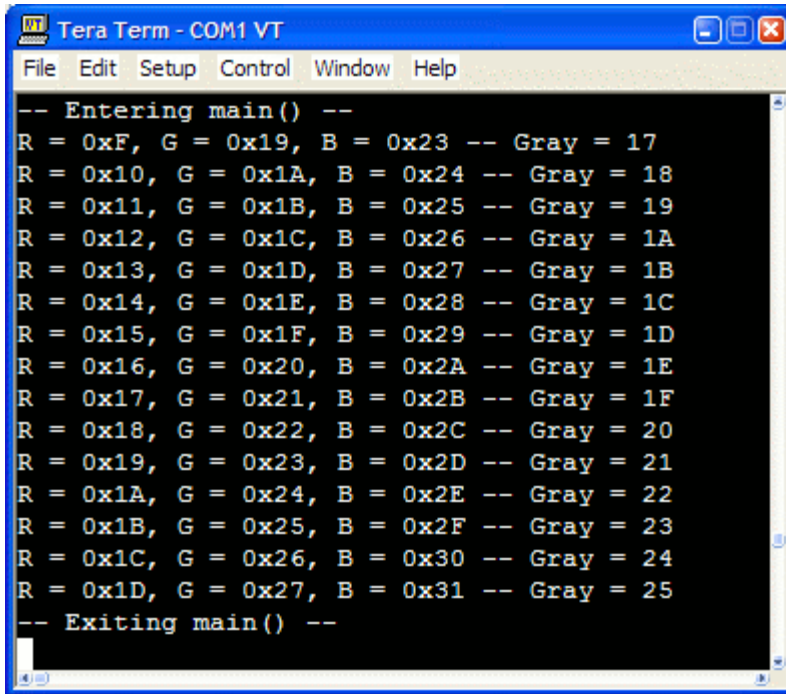
        xil_printf("Gray = %x \n\r", gray);
    }

    print("-- Exiting main() --\n\r");
    return 0;
}
```

When added to the EDK project, the instance name of the pc core is user customizable. The above code example assumes that the pc core is named rgb2gray_sm_0.

The section Integrating a processor with custom logic contains more information on how the hardware is wired up and other software issues.

Running the code will produce the following print out on a RS232 terminal.



```
-- Entering main() --
R = 0xF, G = 0x19, B = 0x23 -- Gray = 17
R = 0x10, G = 0x1A, B = 0x24 -- Gray = 18
R = 0x11, G = 0x1B, B = 0x25 -- Gray = 19
R = 0x12, G = 0x1C, B = 0x26 -- Gray = 1A
R = 0x13, G = 0x1D, B = 0x27 -- Gray = 1B
R = 0x14, G = 0x1E, B = 0x28 -- Gray = 1C
R = 0x15, G = 0x1F, B = 0x29 -- Gray = 1D
R = 0x16, G = 0x20, B = 0x2A -- Gray = 1E
R = 0x17, G = 0x21, B = 0x2B -- Gray = 1F
R = 0x18, G = 0x22, B = 0x2C -- Gray = 20
R = 0x19, G = 0x23, B = 0x2D -- Gray = 21
R = 0x1A, G = 0x24, B = 0x2E -- Gray = 22
R = 0x1B, G = 0x25, B = 0x2F -- Gray = 23
R = 0x1C, G = 0x26, B = 0x30 -- Gray = 24
R = 0x1D, G = 0x27, B = 0x31 -- Gray = 25
-- Exiting main() --
```

Designing and Simulating MicroBlaze Processor Systems

This section shows an example on how to design and simulate a System Generator model containing a MicroBlaze processor. A DSP48 co-processor is developed using System Generator. Using the EDK Processor block, a user imports a MicroBlaze processor, customized in Xilinx Platform Studio (XPS), into the System Generator model. The DSP48 co-processor is then attached to the imported MicroBlaze processor through the automatic memory mapping mechanisms provided by the EDK Processor block.

Two Simulation Techniques

There are two techniques for simulating the complete system with both hardware platform and software programs running on MicroBlaze: *software based simulation* and *hardware-in-the-loop co-simulation*.

Software based simulation

For software based simulation, the MicroBlaze processor is simulated using the software simulation models provided by XPS, while the rest of the design is simulated using System Generator simulation models. The software based simulation does not require any hardware device. The EDK Processor block offers two simulation modes for software based simulation: *Single Step* and *Single Step Only On Shared Memory Access*.

Single Step simulation mode provides cycle-accurate information of the complete system and is very useful in many development scenarios. However a cycle accurate simulation model may not be necessary in some scenarios, for instance in situations when the processor is working on data internal to the processor and rarely communicates to the System Generator design. In these cases, simulation speeds may be improved by using *Single Step Only On Shared Memory Access* simulation mode.

Compared with the *Single Step* mode, the *Single Step Only On Shared Memory Access* simulation mode synchronizes the XPS software simulation models and the System Generator simulation models on each Simulink simulation cycle when there is data being transmitted over the shared memory interface between the MicroBlaze processor and the System Generator model. The processor and the System Generator design effectively run as asynchronous processes until data is read or written to the memory-map. When shared-memory access is detected, simulation toggles into single-step mode, and returns to asynchronous mode after the data transaction has completed.

In addition, the software based simulation provides the following two features to facilitate application development.

- Virtual UART and GPIO console: when the MicroBlaze processor contains OPB (On-chip Peripheral Bus) based UART and GPIO (General-Purpose I/O) peripherals, virtual consoles representing the UART and GPIO peripherals contained by the XPS project appear during simulation. The user can check the status of these peripherals and interact with the MicroBlaze processor through these virtual platform consoles.
- GDB based debugging: the user can launch the GNU GDB debugger to debug the software program running on MicroBlaze. The user can stop/pause the simulation when the MicroBlaze processor reaches a specific line of the software program, or use the various stepping techniques provided by GDB. Once the simulation is stopped/paused, the user can examine the status of the MicroBlaze processor (e.g., register files, a specific memory location) through the GDB GUI, and examine the output of System Generator blocks using

Simulink Scope blocks.

Hardware-in-the-loop simulation

For the hardware-in-the-loop simulation technique, the MicroBlaze processor is compiled into hardware, while the DSP48 co-processor model is left in the System Generator diagram for software simulation. In the example shown in this section, the hardware and software simulation communicate with each other using the point-to-point Ethernet co-simulation technology.

This tutorial example contains the following section:

- Create an XPS Project
- Create a DSP48 Co-Processor Model
- Import an XPS Project
- Configure Memory Map Interface
- Write Software Programs
- Launch Software Based Simulation
- Debug using GDB
- Create a Hardware-in-the-loop Co-Simulation Block
- Create a Testbench Model
- Update the Co-Simulation Block with Compiled Software
- Run the Simulation

This example uses the *Xilinx Virtex 4 ML402 Evaluation Platform*.

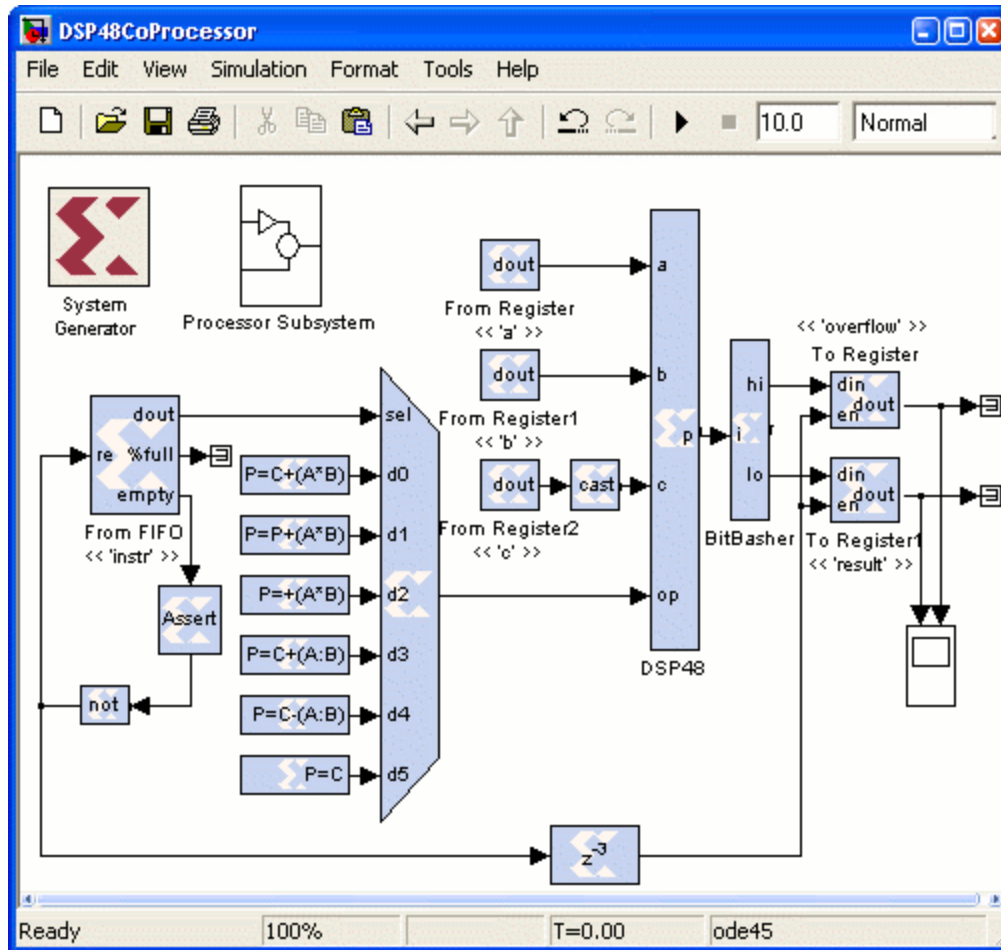
The files used in this tutorial can be found in:

`<path_to_sysgen>\examples\EDK\DSP48CoProcessor`, where
`<path_to_sysgen>` denotes the System Generator installation directory.

Create an XPS Project

First of all, you will need to create a new XPS project, which contains an OPB based UART peripheral. A tutorial on how to create a new EDK project can be found in the section entitled “Using the EDK”.

Create a DSP48 Co-Processor Model



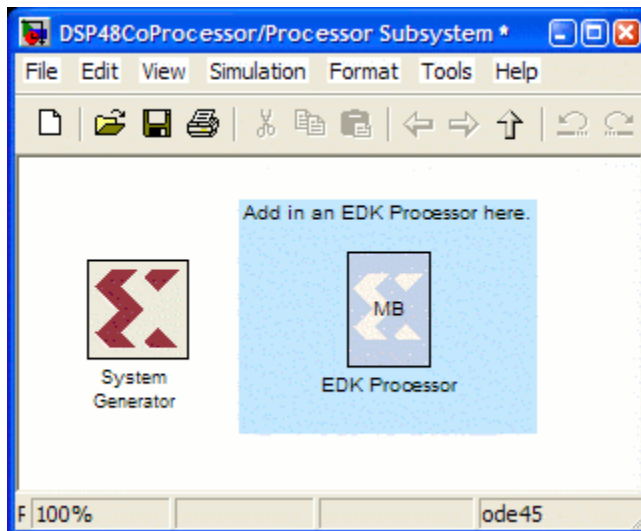
Copy the DSP48CoProcessorModel found in
`<path_to_sysgen>\examples\EDK\DSP48CoProcessor` into a temporary working directory and open up the model.

The model contains a DSP48 block with the *a*, *b* and *c* ports fed from three shared *From Registers* with corresponding names. The *op* port receives signals from a multiplexer whose select line is sourced from a shared *From FIFO* named *instr*.

The output port *p* of the DSP48 block is sliced and fed into another two shared *To Registers*: the top 16 bits into the *overflow* register and the bottom 32 bits into the *result* register.

Import XPS Project

In this section, we import an XPS project that contains a MicroBlaze processor into the DSP48 Co-Processor model. Double click the subsystem called *Processor Subsystem* and look into it. In that subsystem, you will find a System Generator block, and a blue text box as the place holder for an EDK Processor block. Open up the System Generator block set and drag an EDK Processor block from the Index library into the Processor Subsystem. Your augmented subsystem should look like the one shown below:



We will now configure the EDK Processor block to import the XPS project. The import process will make changes to the XPS project. Thus, ensure that the XPS project is *not* currently opened by Xilinx Platform Studio before importing.

Double click on the processor block to bring up the block dialog. In the *Configure processor* for drop-down menu, select *HDL netlisting*. The *Import...* button is enabled as a result of the selection.

Note that the *Import...* button is disabled when the processor is configured for *EDK pcore generation*. In *EDK pcore generation* mode, it is expected that the user creates a pcore in System Generator and export it to be used in another XPS project. In this case, the processor is not instantiated inside the EDK Processor block. In *HDL netlisting* mode, it is expected that the user imports an XPS project into System Generator model and netlists it with other System Generator blocks.

If no XPS project is ever imported, configuring the processor for *HDL netlisting* will automatically trigger the launching of XPS Import Wizard. The XPS Import Wizard can be launched manually by pressing the *Import...* button.

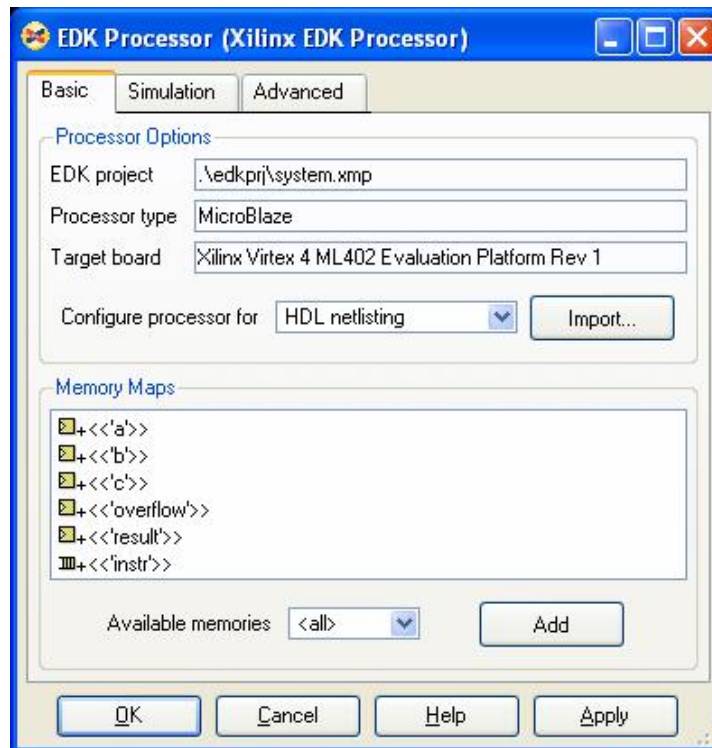
In XPS Import Wizard, browse to the XPS project created in earlier steps and start the import process. The import process copies necessary files into the XPS project and changes the project accordingly to allow the MicroBlaze processor to communicate with the System Generator model. It also invokes XPS to compile software simulation models for the imported XPS project. Note that if the compilation of software simulation models fails, the import process may still be successful. In this case, you cannot use software based simulation to simulate the complete system (all the items under the *Simulation* tab are disabled). However, you can still follow the hardware-in-the-loop co-simulation flow and netlist the complete system.

If there are any software applications contained by the imported XPS project, they are also

automatically compiled during import. Note that this automatic compilation will *not* report any errors. Failing to compile the software applications will *not* cause the XPS import to fail. Once the XPS import succeeds, the user can manually compile the software applications by clicking the *Compile Software Program* button under the *Simulation* pane on the EDK Processor block dialog.

Configure Memory Map Interface

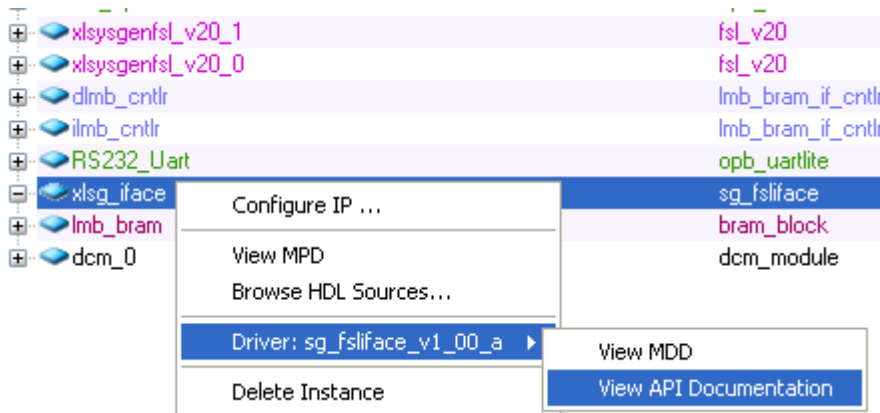
Re-open the dialog of the EDK Processor block. Add all the shared memories in the model to the processor's memory map by selecting *<all>* in the *Available memories* pull-down menu and then pressing the *Add* button. The EDK Processor block dialog should look like the following screenshot. Dismiss the dialog by clicking the OK button at the bottom.



Write Software Programs

We will write software programs running on MicroBlaze to read from and write to the shared memories. Re-open the Xps project in Xilinx Platform Studio. Create a new software application called *MyProject*. Make sure that the *MyProject* software application is marked for download into BRAM while the other software applications are unmarked. Refer to the Using the EDK section on how to add a new software application to an EDK project.

Create a new source code file *MyProject.c* for *MyProject* and open it in the XPS code editor.

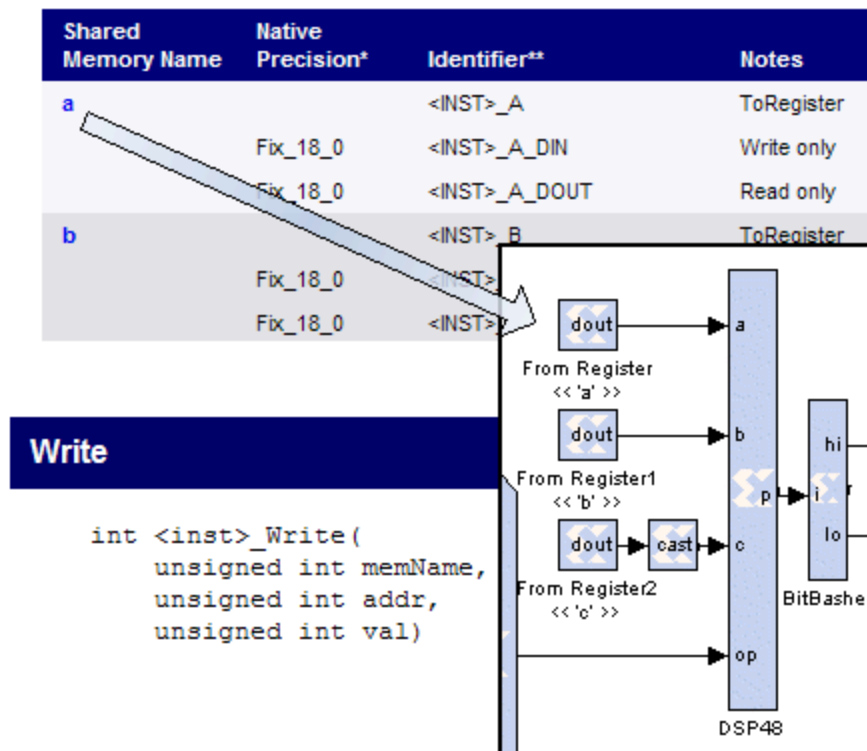


The figure above shows the Assembly View of the XPS project in Xilinx Platform Studio. After the XPS project is imported, a pair of FSL (Fast Simplex Link) buses and an *xlsg_iface* peripheral are added to it. The pair of FSL buses is simply FIFOs used for accessing the MicroBlaze processor. The *xlsg_iface* peripheral is used to connect the FSLs to the System Generator design, and to capture information on how to generate the device drivers for the System Generator design. Right click on the *xlsg_iface* in the Assembly View to see its API documentation.

Include the following header file in *MyProject.c*.

```
#include "xlsg_iface.h"
```

Before reviewing the code that we will run on the processor, let us first consider how to write data to the 'a' register on the model. Look at the DSP48 Co-Processor model. Recall that the 'a' port of the DSP48 block is driven by the output of a shared register by the same name. We wish to write a value to that shared register from within MicroBlaze code. By referring to the driver API, we can see that the shared memory called 'a' is a register. It has a write-only port called <INST>_A_DIN. If we read the driver API we will see that <INST>_A_DIN expands to XLSG_IFACE_A_DIN because the instance name of the peripheral is *xlsg_iface*.



So in order to write a value to the 'a' shared register, we would use the single-word write access function thus:

```
xlsg_iface_Write(XLSG_IFACE_A, XLSG_IFACE_A_DIN, 2);
```

The first parameter contains the memory name, in this case XLSG_IFACE_A. The next parameter contains the addr we wish to write to. Because we are writing to a register (similarly for a FIFO), we don't have a numeric address to write to. Instead, we have a symbolic address that is shown in the **Identifier** column of the documentation. Since we wish to write values into the register, we select the XLSG_IFACE_A_DIN address to write to.

The full code of *MyProject.c* is shown as follows.

```

#include "xparameters.h"
#include "xlsg_iface.h"

int main (void) {
    int result;
    int overflow;
    char c;

    print("Performing p=2*4\r\n");
    // -- Set the a port register to 2
    xlsg_iface_Write(XLSG_IFACE_A, XLSG_IFACE_A_DIN, 2);
    // -- Set the b port register to 4
    xlsg_iface_Write(XLSG_IFACE_B, XLSG_IFACE_B_DIN, 4);
    // -- Set the instr register to 2: p=a*b
    xlsg_iface_Write(XLSG_IFACE_INSTR, XLSG_IFACE_INSTR_DIN, 2);

    print("Press a key to read back result\r\n\r");
  
```

```

c = inbyte();

// -- Read back the result register
xlsg_iface_Read(XLSG_IFACE_RESULT, XLSG_IFACE_RESULT_DOUT, &result);
// -- Read back the overflow register
xlsg_iface_Read(XLSG_IFACE_OVERFLOW, XLSG_IFACE_OVERFLOW_DOUT,
                &overflow);
xil_printf("Read back: result=%d, overflow=%d\n\r",
           result, overflow);

print("Performing p=c+(a:b)\n\r");
// -- Set the aport register to 262140
xlsg_iface_Write(XLSG_IFACE_A, XLSG_IFACE_A_DIN, 262140);
// -- Set the bport register to 262140
xlsg_iface_Write(XLSG_IFACE_B, XLSG_IFACE_B_DIN, 262140);
// -- Set the cport register to 1
xlsg_iface_Write(XLSG_IFACE_C, XLSG_IFACE_C_DIN, 1);
// -- Set the instr register to 2: p=c+(a:b)
xlsg_iface_Write(XLSG_IFACE_INSTR, XLSG_IFACE_INSTR_DIN, 3);

print("Press a key to read back result\n\r");
c = inbyte();

// -- Read back the result register
xlsg_iface_Read(XLSG_IFACE_RESULT, XLSG_IFACE_RESULT_DOUT, &result);
// -- Read back the overflow register
xlsg_iface_Read(XLSG_IFACE_OVERFLOW, XLSG_IFACE_OVERFLOW_DOUT,
                &overflow);
xil_printf("Read back: result=%d, overflow=%d\n\r",
           result, overflow);

return 0;
}

```

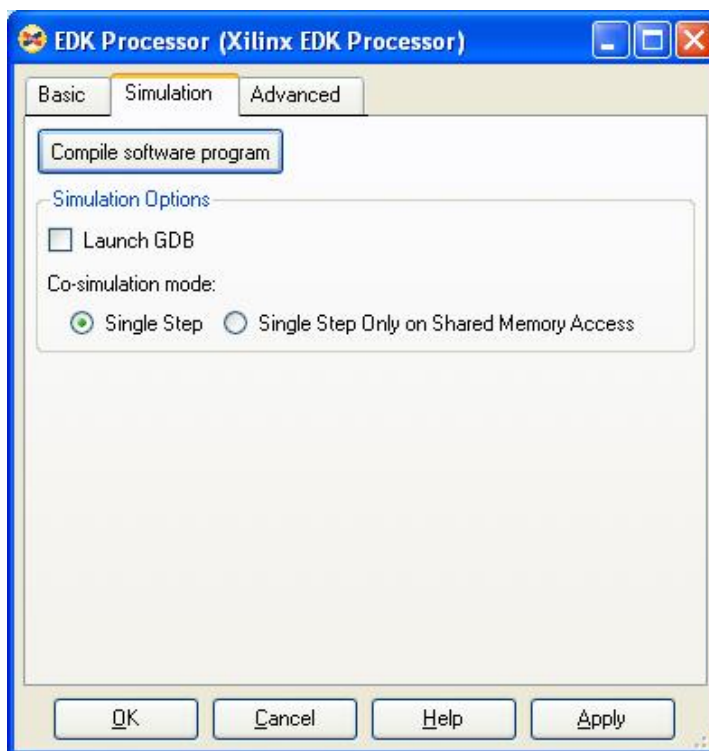
Copy and paste the above code into *MyProject.c*.

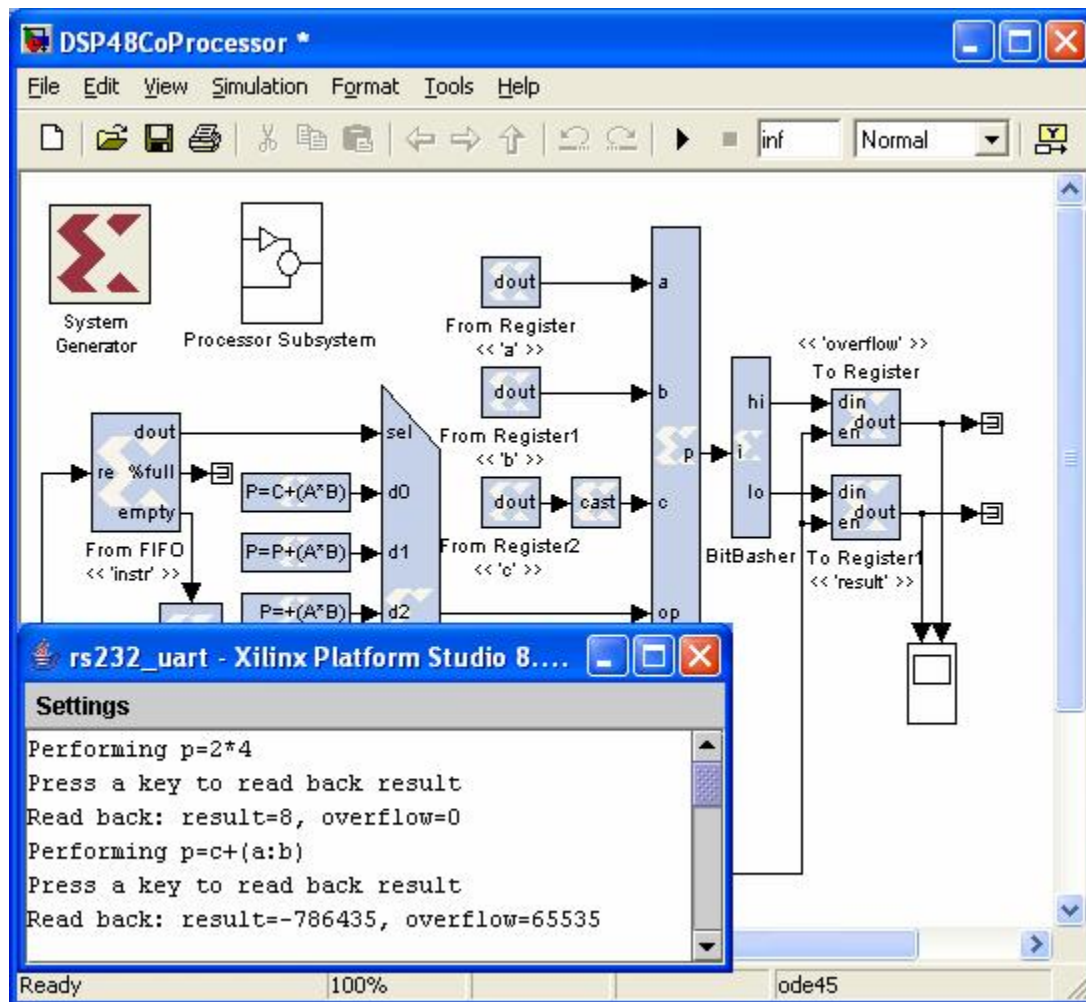
To compile the software program, click the *Compile software program* button under the *Simulation* tab on the EDK Processor block diagram. If no errors are reported during the compilation, we are ready to simulate the complete system.

Launch Software Based Simulation

Click the *Simulation* tab on the EDK Processor block dialog to activate it. Make sure that the software programs are compiled by clicking the *Compile software program* button. The compilation through the EDK Processor block diagram allows System Generator to pick up the right executable file to be run on MicroBlaze.

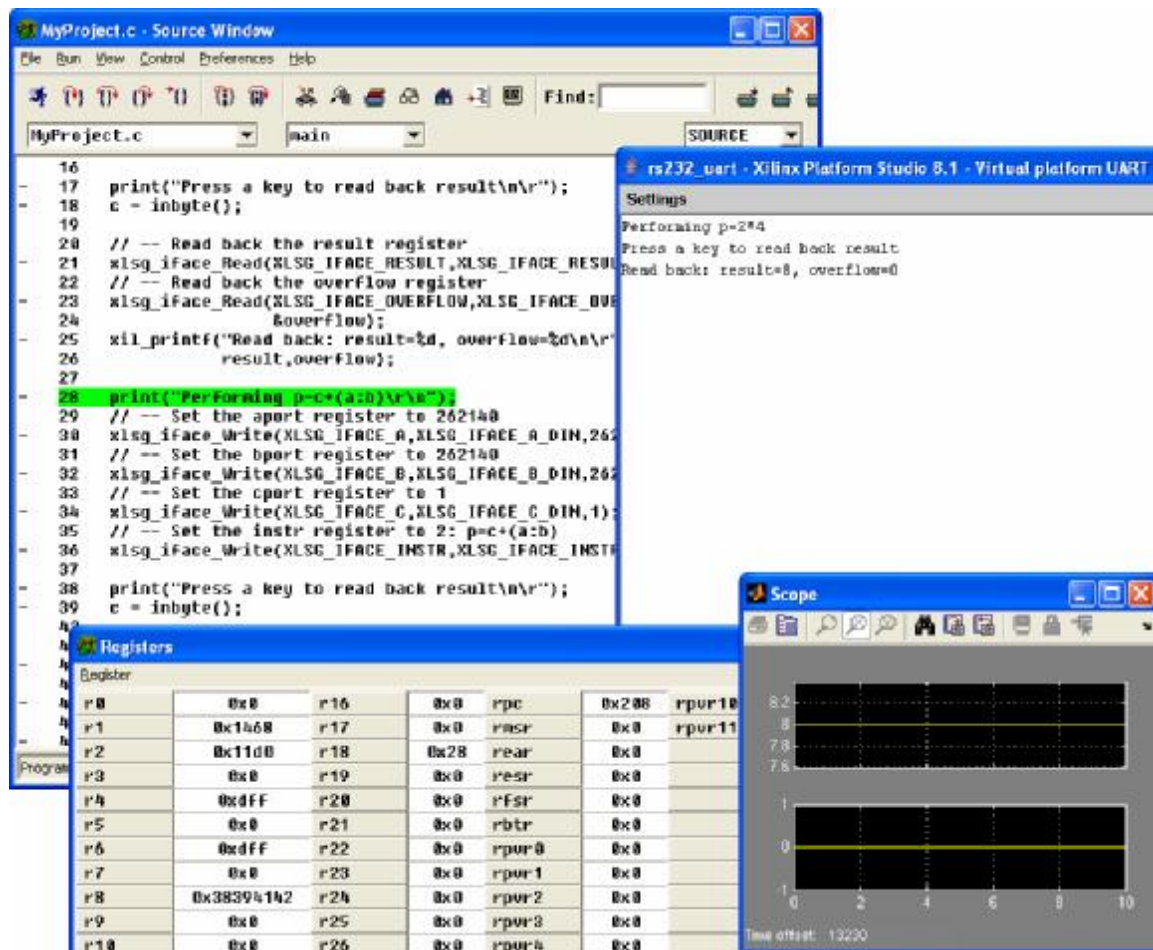
Click on the Simulink tab. Select either *Single Step* or *Single Step on Shared Memory Access* co-simulation mode. Simply start the Simulink simulation. A Virtual UART console appears automatically at the simulation start, and shows that the MicroBlaze processor is going to perform the 2*4 operation. It then waits for a key press before displaying the results read back from the System Generator simulation. Press Enter on the Virtual UART console to get the results from the DSP48 co-processor model be shown on it.





Debug Using GDB

In the *Simulation* tab, select *Launch GDB*. Then, when starting Simulink simulation, a GDB GUI window automatically pops up, displays the software programs running on MicroBlaze, and highlights the line under execution. You can use the various debugging capabilities provided by GDB to set breakpoints and/or step through the software programs. You can examine the status of the MicroBlaze processor through the GDB GUI (e.g., the content of the MicroBlaze register file shown below). Also, you can open the Simulink Scope in the model to examine the status of the System Generator blocks.

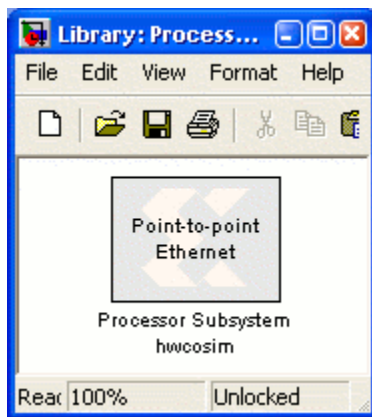


Create a Hardware-in-the-loop Co-Simulation Block

In addition to software-based simulation, the complete system can be simulated using the hardware-in-the-loop co-simulation technique. Make sure that the shared memories are added into the *Memory Maps* window and the EDK Processor block is configured for *HDL netlisting*. These are required for the hardware-in-the-loop co-simulation.

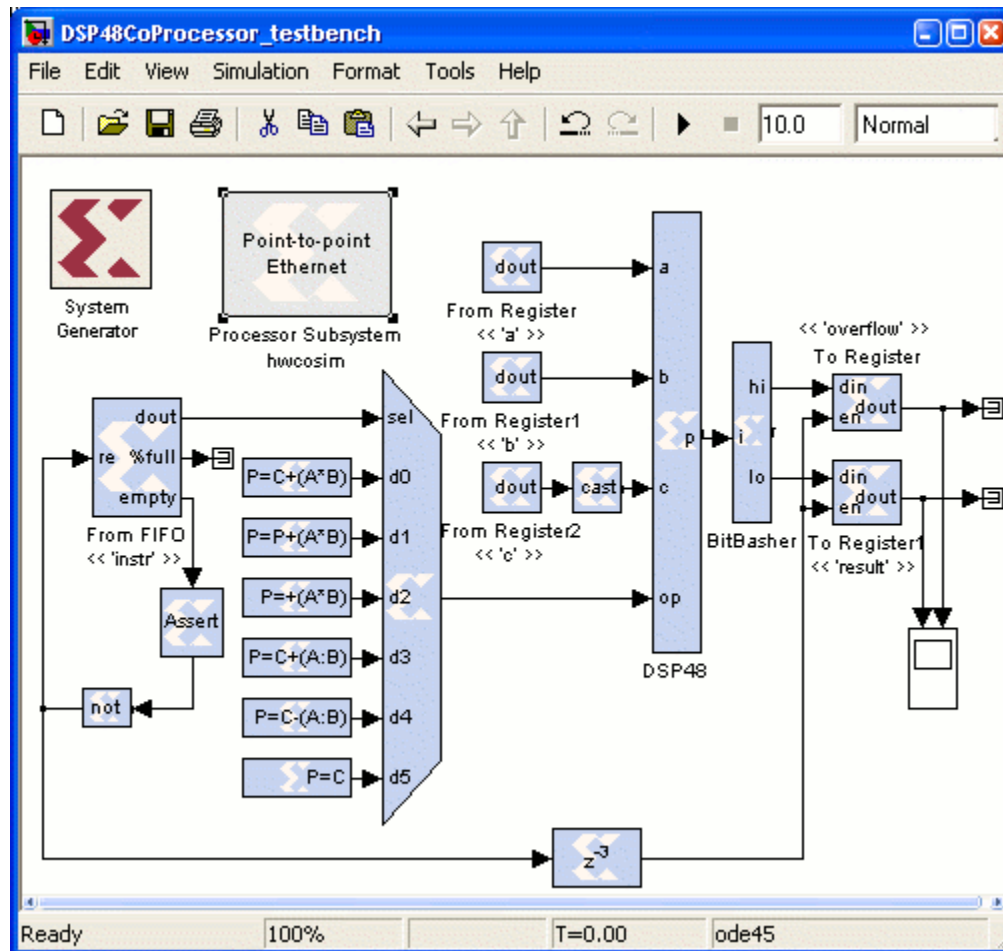
Open the block dialog of the System Generator block in the same subsystem as the EDK Processor block. We generate the hardware-in-the-loop co-simulation block from this level of the model so that only the imported MicroBlaze processor run in hardware while the rest of the design is kept in Simulink for software simulation.

Under the *Compilation* menu select Hardware Co-simulation -> ML402 -> Ethernet -> Point-to-point. Next press the *Generate* button to begin the compilation process. This may take some time. Upon completion, a hardware co-simulation block is created that contains a MicroBlaze processor.

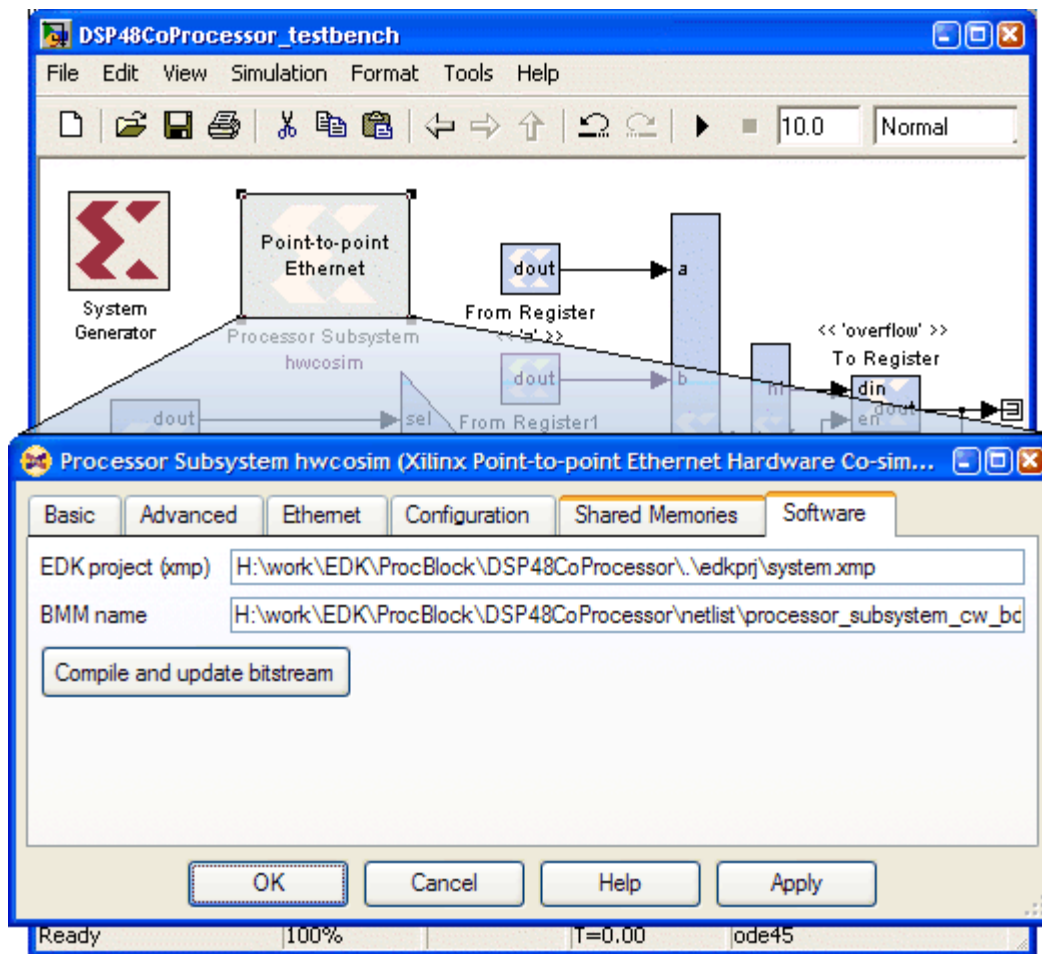


Create a Testbench Model

A testbench model will be created to use the co-simulation block created in the previous step. Open the DSP48CoProcessor model, and delete the Processor Subsystem. Copy the Processor Subsystem hwcosim block just generated into the DSP48CoProcessor model. Save the model as *DSP48CoProcessor_testbench.mdl*.



Update the Co-Simulation Block with Compiled Software



Return to the testbench model. Double click on the Processor Subsystem hwcosim block to bring up the dialog shown above. To compile the software contained in the XPS project listed in the *Software* tab and load it into the hardware co-simulation bitstream, click the button labeled *Compile and update bitstream*.

Since Point-to-point Ethernet co-simulation is chosen, we need to configure the Ethernet interface and also the Configuration interface of the Processor Subsystem hwcosim block. Select a valid Host interface for your Ethernet communications, and set the configure interface to Point-to-point Ethernet. Refer to the Using FPGA Hardware in the Loop section for more usage information of the hardware-in-the-loop co-simulation block

Run the Simulation

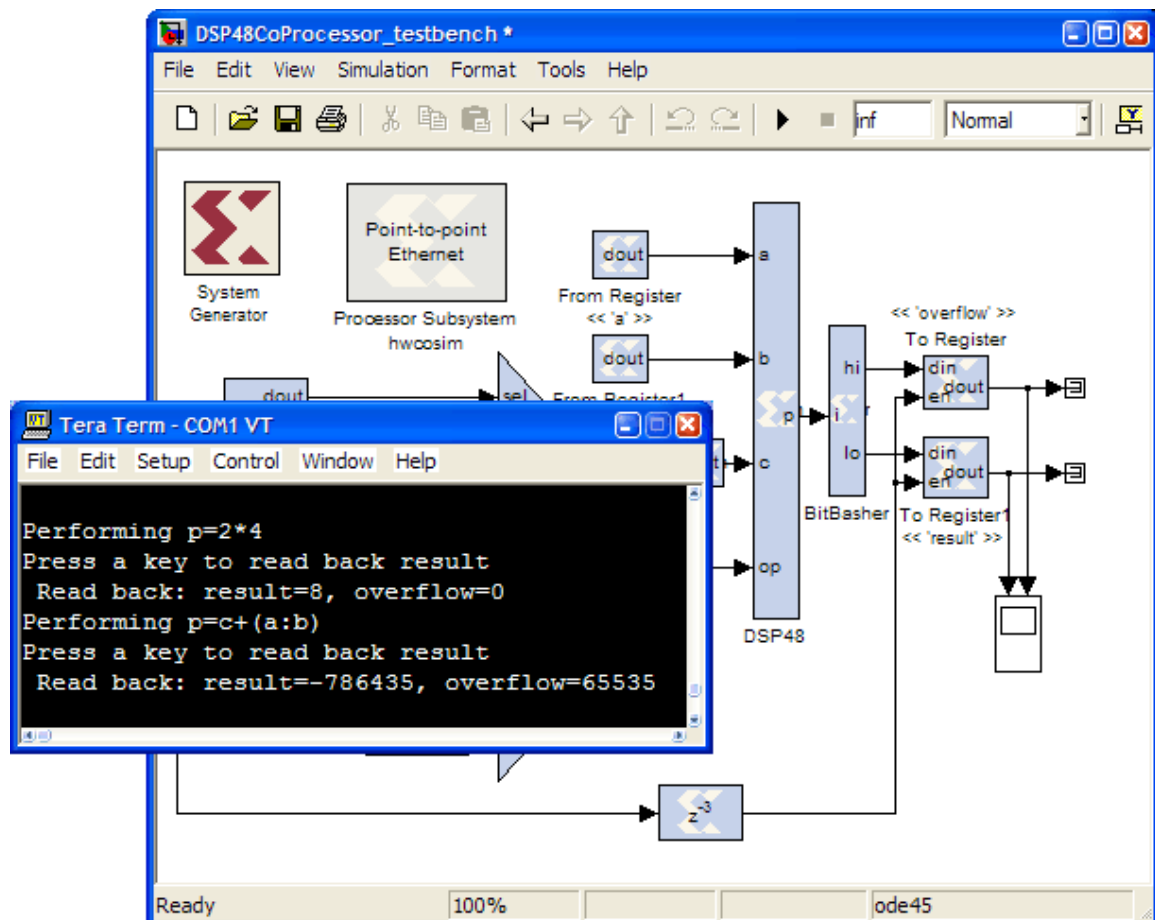
Before starting the simulation, we need to set up a terminal connected to the COM port of your computer. This allows for text inputs and outputs to be read and fed through the RS232 port in the MicroBlaze.

Open up your favorite terminal program. Windows comes with a Hyperterminal application, which can be found in Start -> All Programs -> Accessories -> Communications -> Hyperterminal. Set up the terminal program to listen to the COM port that you have wired your RS232 to.

Configure your terminal as follows:

- Baud rate = 115200
- Data = 8 bits
- Parity = none
- Stop = 1 bit
- Flow control = none

Set the simulation time of the testbench model to *inf*, allowing enough simulation time for the MicroBlaze processor to wake up and respond.



Start the Simulink simulation. Similar to the software based simulation, the MicroBlaze will wake up and print to the Hyperterminal that it is going to perform the $2*4$ operation. It then waits for a key press before displaying the results read back from the System Generator simulation. The results you

see from the Hyperterminal should match with what you see on the UART virtual console during software based simulation.

Software Based Simulation Modes

This tutorial uses an LMS (Least Mean Square) filtering application to illustrate the two software based simulation modes offered by the EDK Processor block, and how to use them to speed up the simulation in System Generator.

The files used in this tutorial can be found in:

`<path_to_sysgen>\examples\EDK\LMSFilter`, where `<path_to_sysgen>` denotes the System Generator installation directory.

Two Software Based Simulation Modes

For software based simulation, the imported MicroBlaze processor can be simulated using the simulation models provided by EDK (Embedded Development Kit). The EDK simulation models and other System Generator simulation models need to synchronize and exchange simulation data with each other. Due to the different simulation speeds of the EDK simulation models and the different requirements for hardware and software simulation, the EDK Processor block offers two simulation modes, *Single Step* mode and *Single Step Only On Shared Memory Access* mode, to allow for efficient simulation of the complete system.

In *Single Step* mode, the synchronization and data exchange between the EDK simulation models and the System Generator simulation models take place on each Simulink clock cycle. This presents a cycle-accurate simulation image of the complete system.

In *Single Step Only On Shared Memory Access* mode, the synchronization and data exchange between the EDK simulation models and the System Generator simulation models occur depending on the communication activity over the shared memory based memory map interface generated by the EDK Processor block. More specifically, the EDK simulation models are automatically switched between two co-simulation modes based on the communication occurrence conditions on the memory map interface during simulation. The communication occurrence conditions and the automatic switch of co-simulation modes based on these conditions are further explained in the following paragraphs.

The *Single Step* mode is very useful when there is light processing on the MicroBlaze processor and the user needs to know the interactions between the MicroBlaze processor and the System Generator designs on each clock cycle (e.g., when developing the hardware and software interface). In contrast, the *Single Step Only On Shared Memory Access* mode is very useful when there is heavy processing on the MicroBlaze processor.

Communication Occurrence Conditions

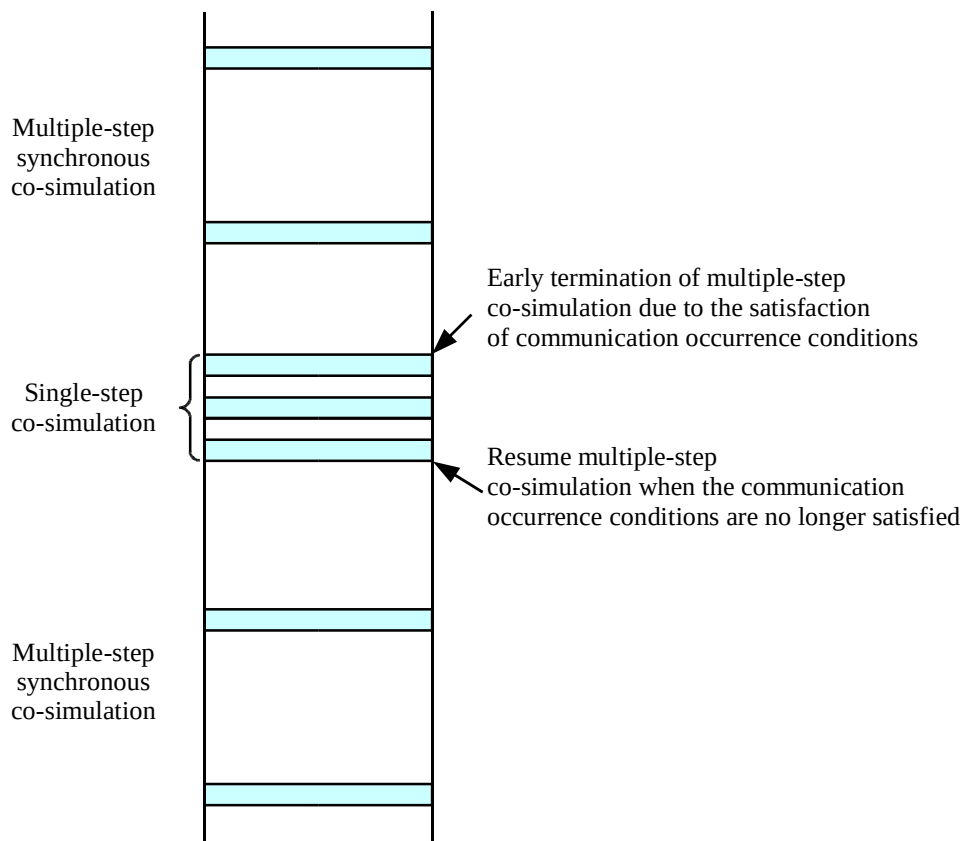
The communication occurrence conditions indicate whether there is communication over the shared memory based memory map interface. Each condition consists of three fields: the port that involves in the communication, the value on the port that indicates the start of a communication operation, and the duration of the communication operation. The memory map interface generated by the EDK Processor block consists of a pair of FSLs. The communication occurrence conditions for this memory map interface are shown in the table below.

Port	Value	Duration
<code>sg2fsl_r</code>	1	1
<code>fsl2sg_exists</code>	1	3
<code>sg2fsl_w</code>	1	1

Port *fsl2sg_exists* becoming 1 indicates that the MicroBlaze processor has data to be sent to the System Generator designs. Once port *fsl2sg_exists* becomes 0, indicating that the MicroBlaze processor does not have any more data to be sent to the System Generator designs, the communication operation for sending data from the MicroBlaze processor to the is finished.

Port *sg2fsl_r* becoming 1 indicates that the System Generator designs have successfully accepted the data from the MicroBlaze processor. The accepting operation lasts from three clock cycles once initiated, allowing time for the MicroBlaze processor to check the status of the slave FSL channel and properly absorb the acknowledgement from the System Generator designs.

Port *sg2fsl_w* becoming 1 indicates that the System Generator designs have data to be written to the MicroBlaze processor. The data writing operation is finished when port *sg2fsl_w* become 0.



Automatic Switch of Co-Simulation Modes

In *Single Step Only On Shared Memory Access* mode, the EDK simulation models may operate in the following co-simulation modes in relative to Simulink.

- In *multiple-step synchronous* mode, the EDK simulation model advances N clock cycles during one Simulink clock cycle. N is set to 500, considering the simulation speeds of the simulation models and the simulation requirements.
- In *single-step* mode, the EDK simulation model advances one clock cycle during one Simulink clock cycle. The *single-step* model is a special case of the multiple-step synchronous mode with N = 1.

During simulation, the EDK simulation models are automatically switched between multi-step synchronous mode and single-step mode depending on the communication occurrence conditions described in the previous paragraphs. More specifically,

- When the imported EDK processor is considered to be not communicating with other Simulink designs, the EDK simulation models are put into multiple-step synchronous mode and advance N clock cycles in one Simulink clock cycle. Note that if during any of the N clock cycles, the EDK simulation models may trigger the communication occurrence conditions by changing the status of the ports on the communication interface. In this case, the multiple-step synchronous mode is terminated immediately. In the next Simulink clock cycle, the EDK simulation models will be turned into *single-step* mode.
- When the MicroBlaze processor and the System Generator designs are considered to be communicating with each other, the EDK simulation models are trapped into single-step mode and advance one clock cycle in one Simulink clock cycle.

At the end of a Simulink clock cycle, the EDK simulation models and the System Generator simulation models exchange simulation data with each other.

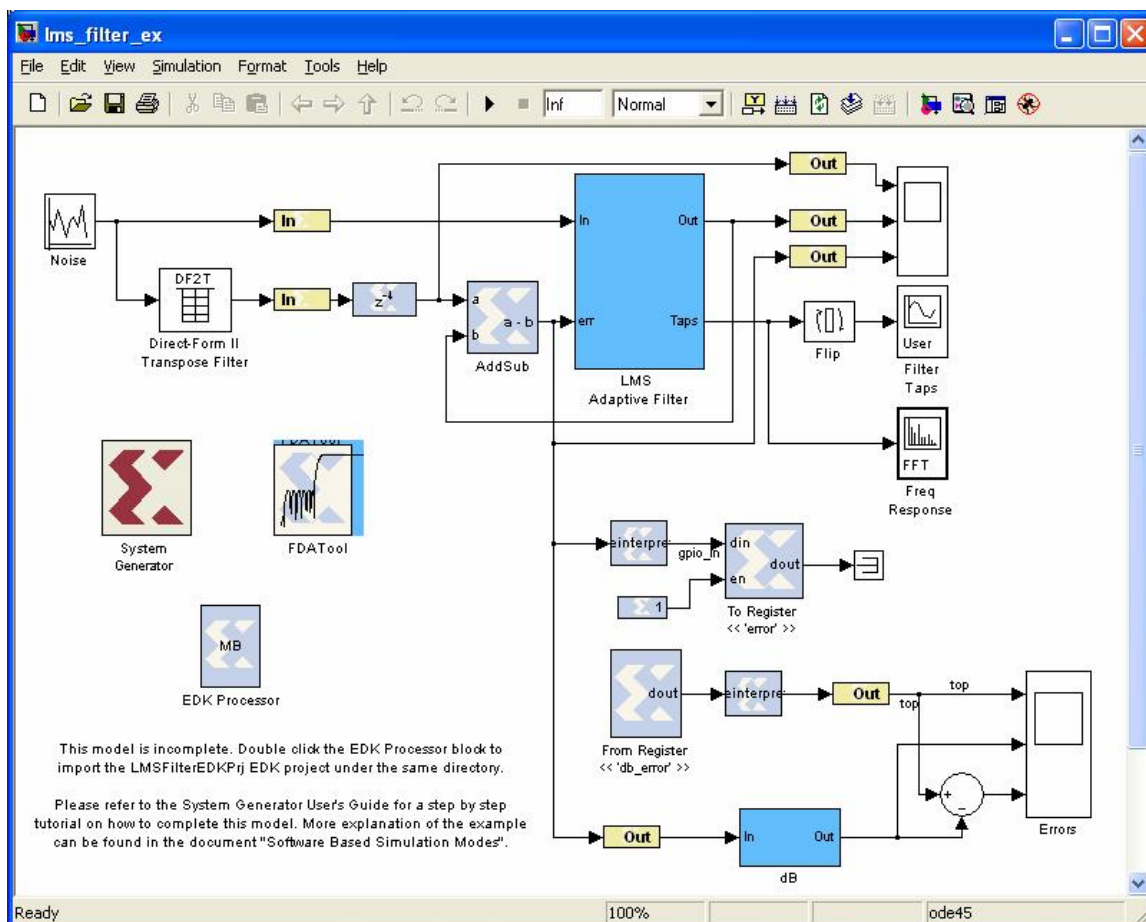
One example scenario of automatic co-simulation mode switching is illustrated in Figure 2.

Open the LMS Filter Example

Open the `lms_filter_ex` model from `<path_to_sysgen>\examples\EDK\LMSFilter`. The System Generator model contains an *LMS Adaptive Filter* subsystem for filtering the input data. The errors of the LMS adaptive filter are then converted to dBs using a MicroBlaze processor and using Simulink.

The 'error' To Register accepts the errors from the LMS adaptive filter while the dB calculation results from the MicroBlaze processor are sent to the 'error_db' From Register.

Double click the EDK Processor block. On the pop-up dialog box, click the *Import...* button to invoke the EDK Import Wizard. Follow the instruction on the EDK Import Wizard to import the *LMSFilterEDKPrj* EDK project. Refer to *Designing and Simulating MicroBlaze Processor Systems* and *Using the EDK for information on how to create an EDK project, import it into the System Generator model, add the shared memories in the model to the EDK Processor block, and invoke the automatic memory map generation.*

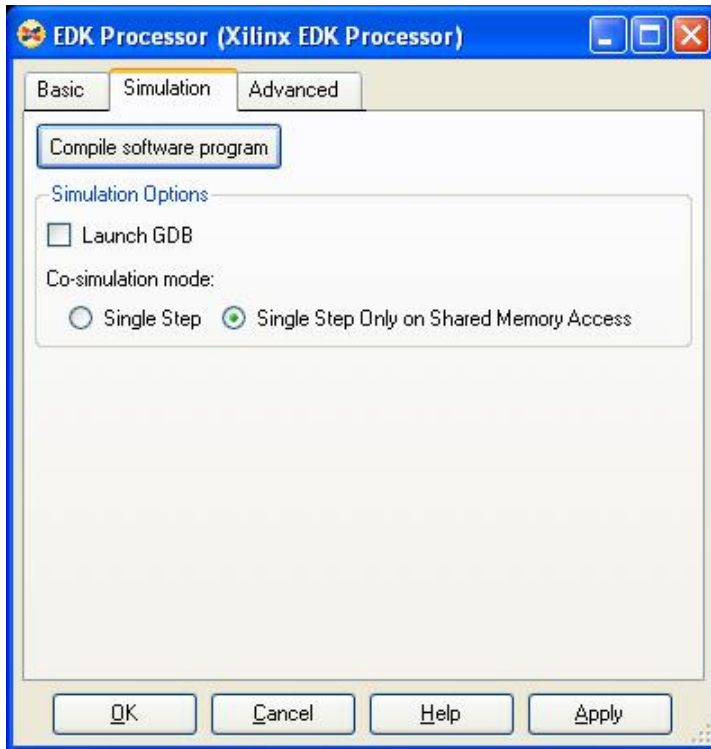


Compile Software Programs

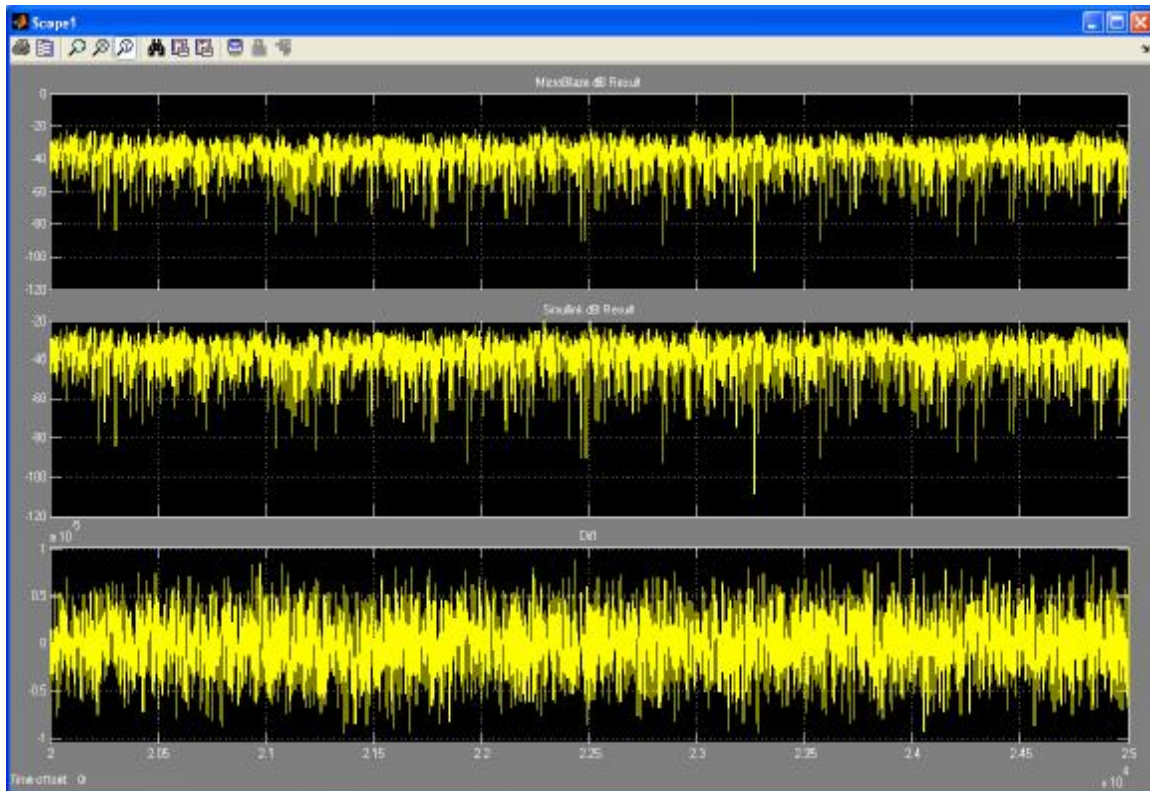
Create a new software application in the EDK project and add the *microblaze_db_calc.c* file to it. Compile the software program by clicking the “Compile software program” button under the Simulation tab on the EDK Processor block dialog box.

Run Simulation in “Single Step Only on Shared Memory Access” Mode

Double click the EDK Processor block to activate the dialog box. Under the Simulation tab, select “Single Step Only on Shared Memory Access” co-simulation mode.



Start Simulink simulation. Double click the Simulink Scope at the bottom left corner of the model and you will see the following simulation results. The MicroBlaze dB results and the Simulink dB results are almost identical to each other. They differ slightly since the Simulink computation uses double-precision floating-point operations while the MicroBlaze computation uses fixed-point operations.

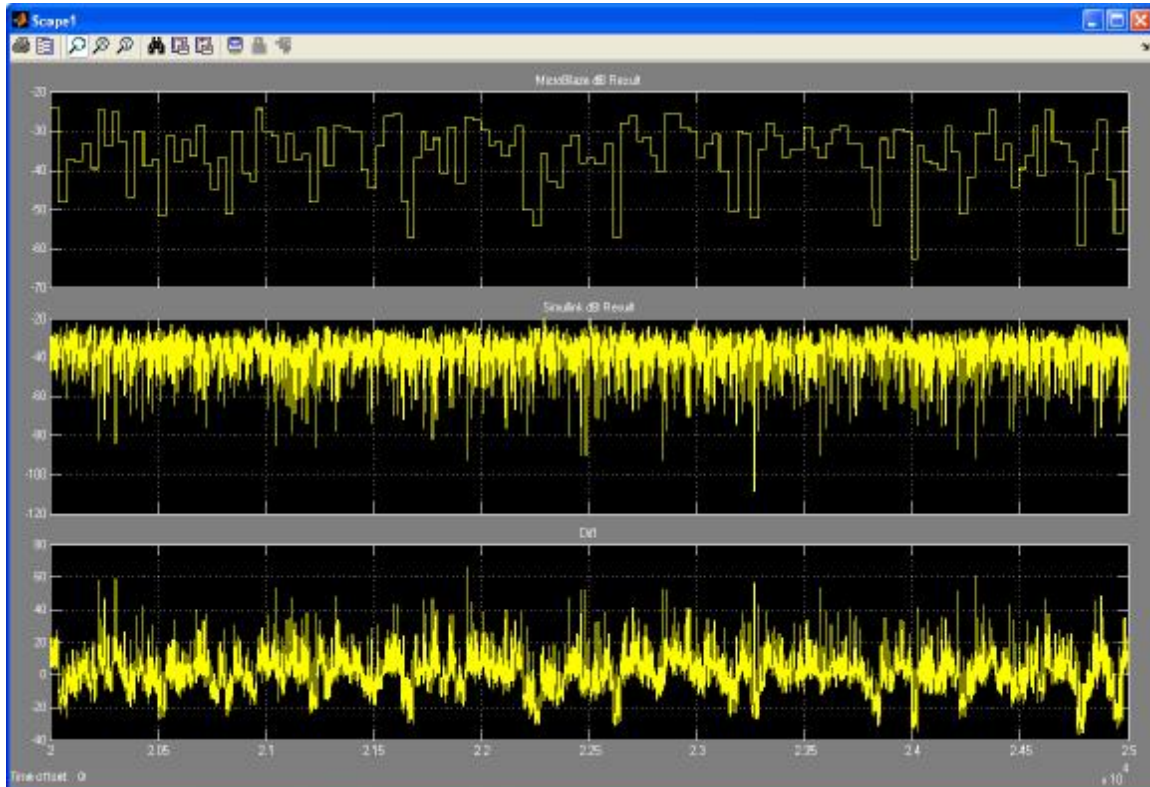


The dB calculation in the MicroBlaze processor requires a few hundreds of simulation clock cycles while the dB calculation in Simulink can generate one output on each clock cycle. However, since the dB calculation does not require the communication between the MicroBlaze processor and the System Generator models, the EDK simulation models advance a few hundreds of clock cycles in a single Simulink clock cycle, and is able to finish the computation of one output sample on every Simulink clock cycle. Thus, the dB calculation in MicroBlaze and that in Simulink match with each other.

Run Simulation in “Single Step” Mode

Double click the EDK Processor block to activate the dialog box. Under the Simulation tab, select “Single Step” co-simulation mode.

Start Simulink simulation. Double click the Simulink Scope at the bottom left corner of the model and you will see the following simulation results. The MicroBlaze dB results and the Simulink dB results are very different from each other in this case.



As mentioned above, the dB calculation in the MicroBlaze processor requires a few hundreds of clock cycles while the dB calculation in Simulink generates one output sample on each simulation clock cycle. In *Single Step* mode, it takes a few hundreds of Simulink simulation clock cycle for the MicroBlaze processor to finish computing one output sample, which is apparently unable to keep up with the dB calculation in Simulink. Thus, the dB calculation in MicroBlaze and that in Simulink are very different from each other as seen on the Simulink Scope.

Using the EDK

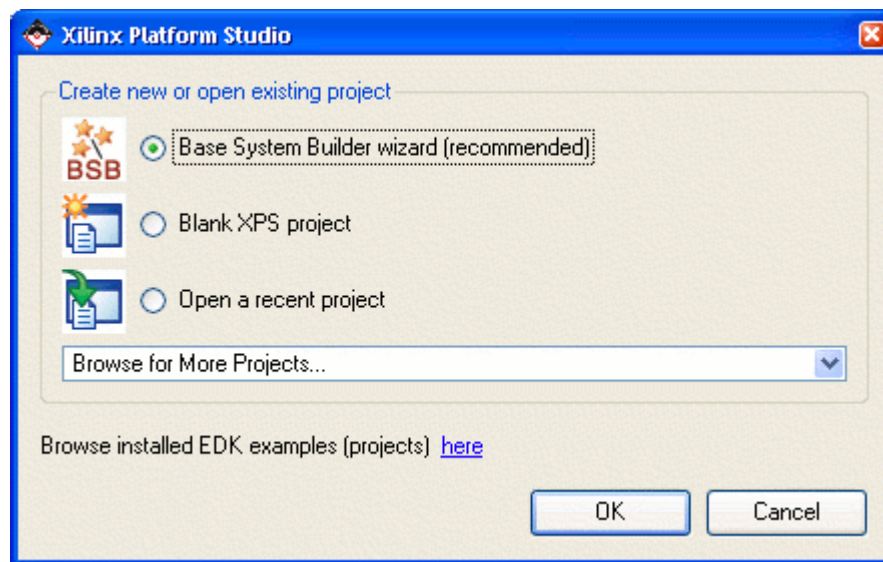
This section provides a quick tutorial on several aspects of the Xilinx Embedded Development Kit (EDK). Please refer to the EDK documentation for more in depth documentation and tutorials.

- Creating a new XPS Project
- Adding a new software application
- Adding a pcore to an EDK project

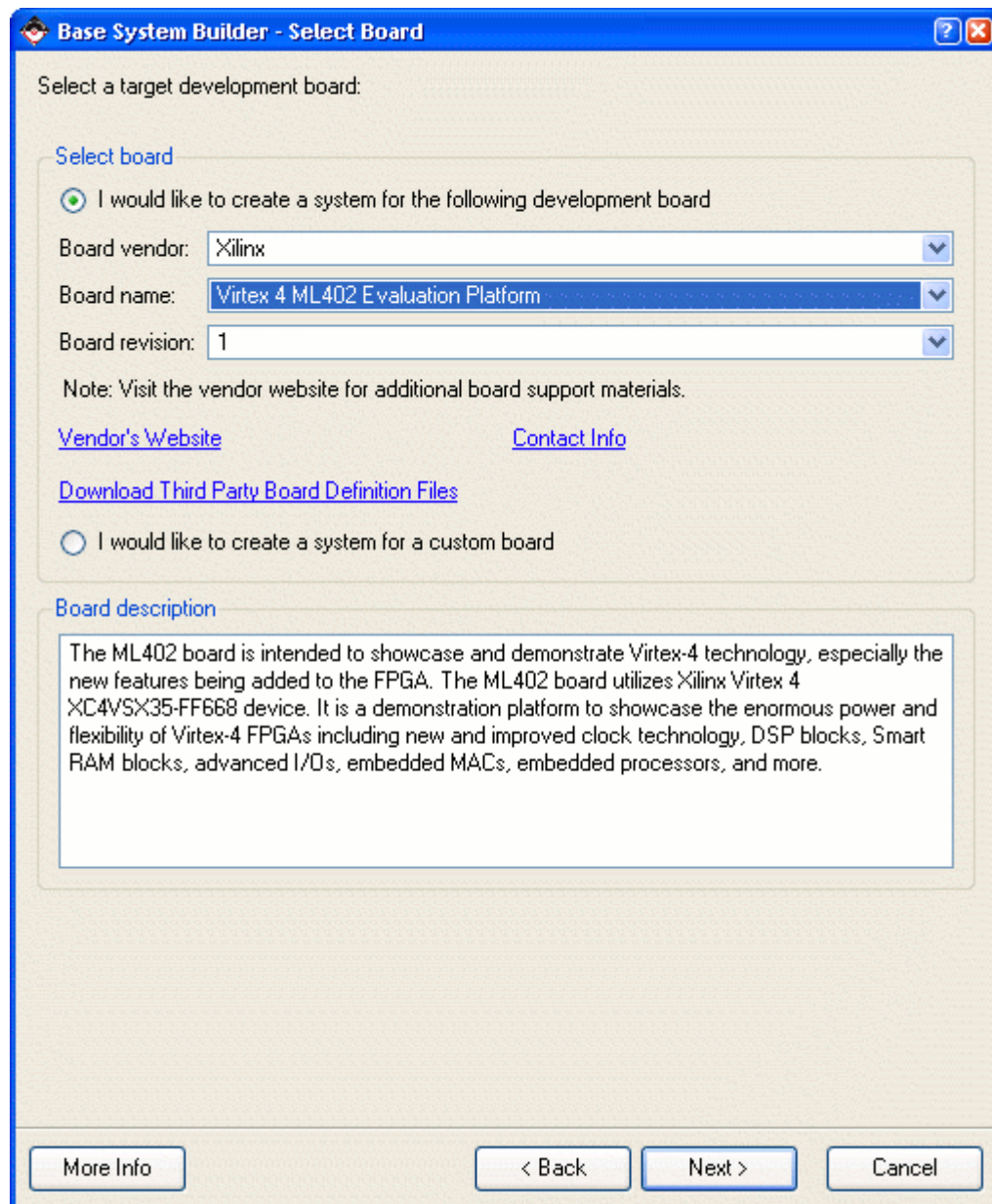
Creating a new XPS Project

The Base System Builder is an EDK wizard to help you construct a fully configured EDK project. This section walks you through creating an EDK project configured with a MicroBlaze Processor, running on a Xilinx ML402 hardware development board.

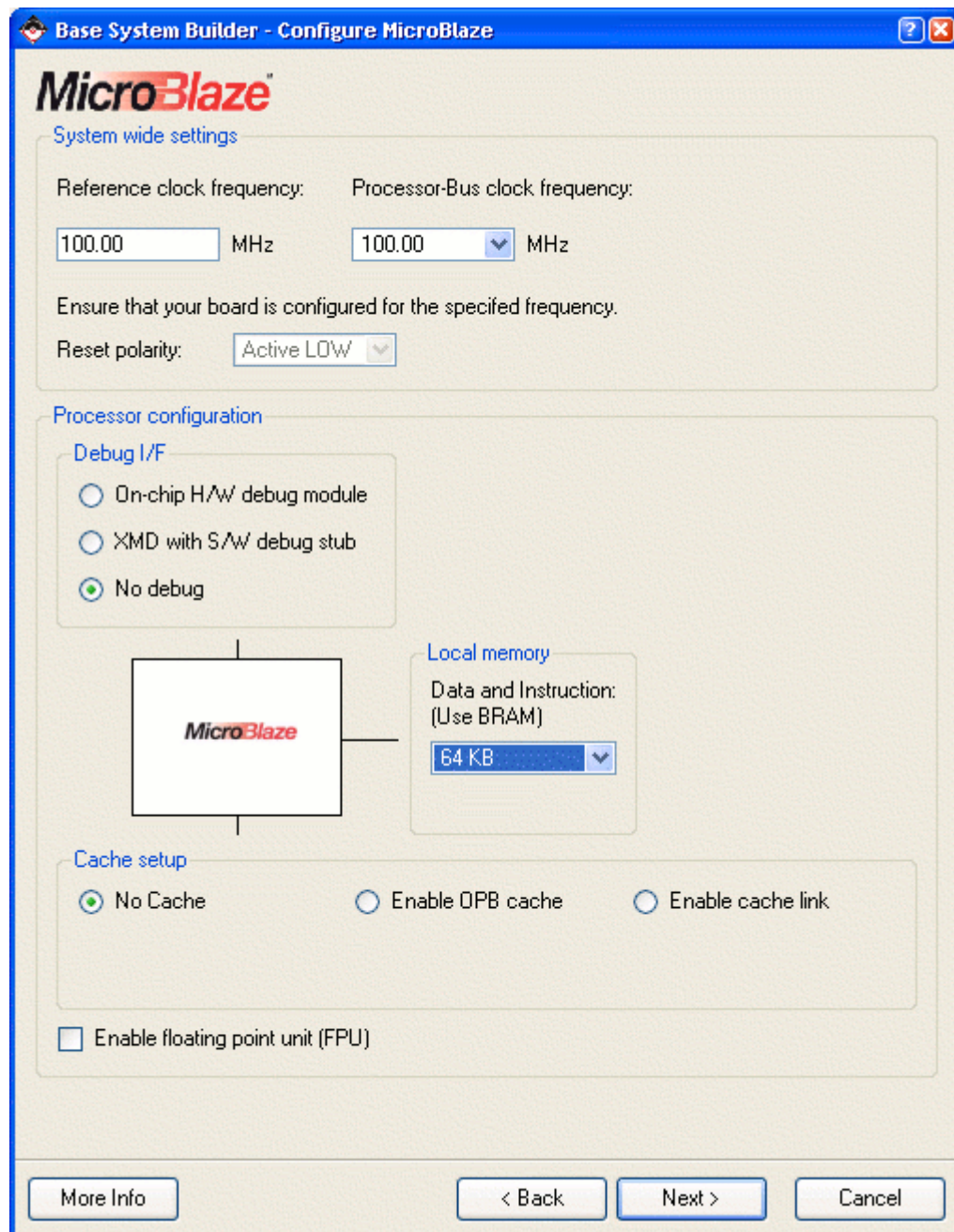
1. Launch Xilinx Platform Studio from the Windows start menu.
2. When XPS launches, the following dialog should appear. Select “Base System Builder wizard” and click OK.



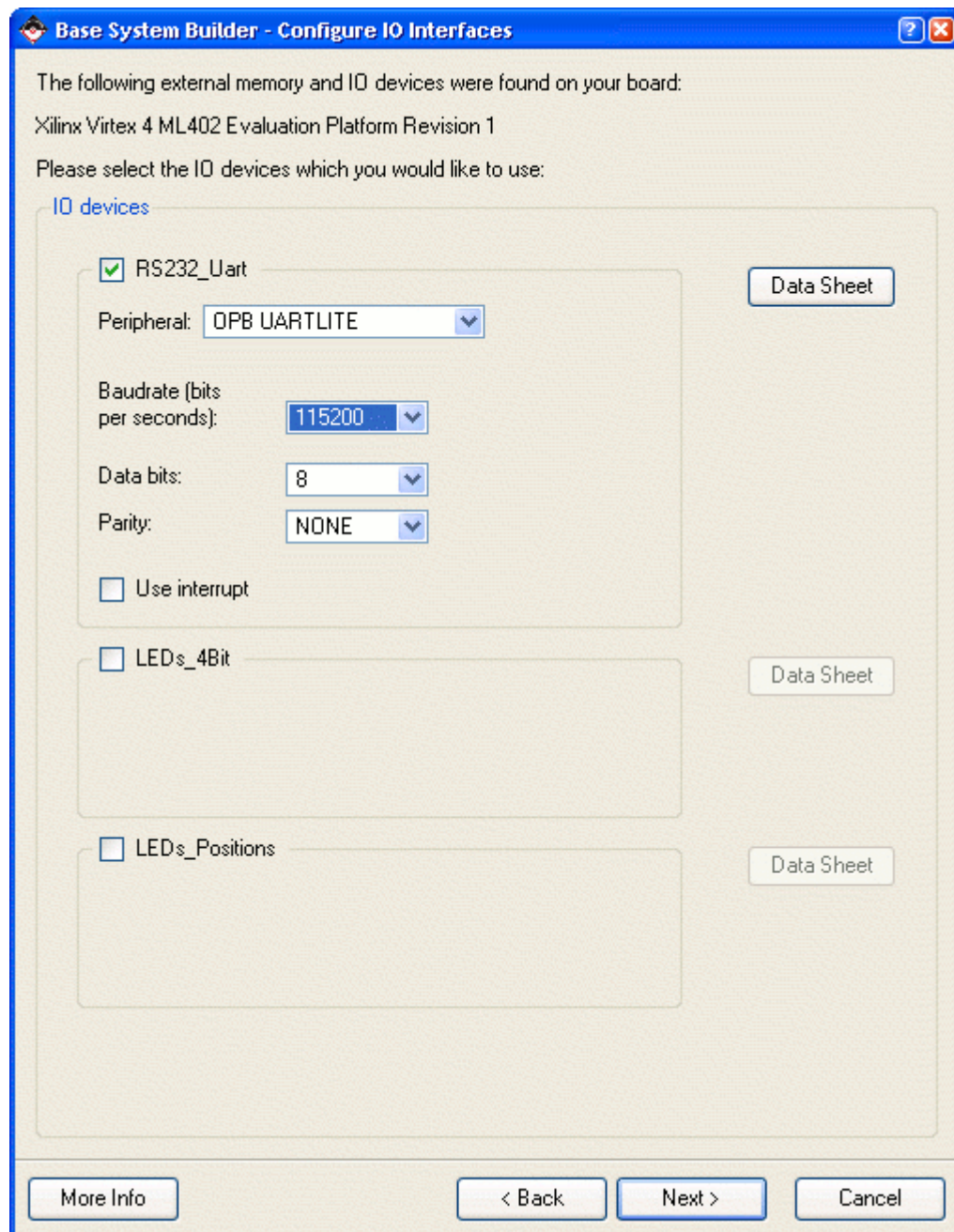
3. Next tell BSB that you wish to create a new design by selecting the “I would like to create a new design” radio button. Click Next.
4. Base System Builder – Select Board. Select the kind of board you want to make use of and click next.



5. Base System Builder – Select a processor. Next you will be asked to select a processor. Select MicroBlaze and click next.
6. Base System Builder – Configure MicroBlaze. You will next have to configure the MicroBlaze. Configure the dialog as shown below.



7. Base System Builder – Configure IO interfaces. We only want the RS232 interface. Set the parameters of the RS232_Uart as shown below, and unselect the other IO interfaces.



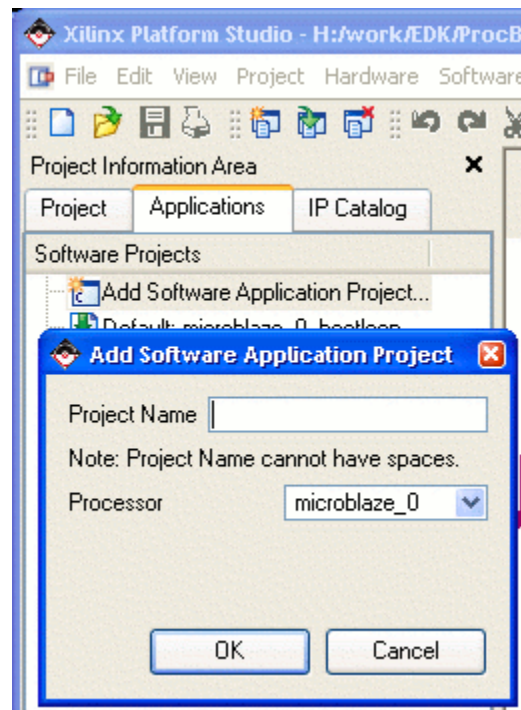
8. Base System Builder – Configure Additional IO Interfaces. The next dialog contains additional IO interfaces supported by this board. Depending on the board, there may be multiple dialog pages of “Additional IO Interfaces”. At this time, we have no need for any additional IO interfaces, so unselect them all, and click next.
9. Base System Builder – Add Internal Peripherals. We have no internal peripherals to add. Click next.
10. Base System Builder – Software Setup. In this dialog, you can select the input and output of STDIN and STDOUT. By default they should be routed to the RS232_Uart. You may also

select to have example code produced in the project for you. Click next.

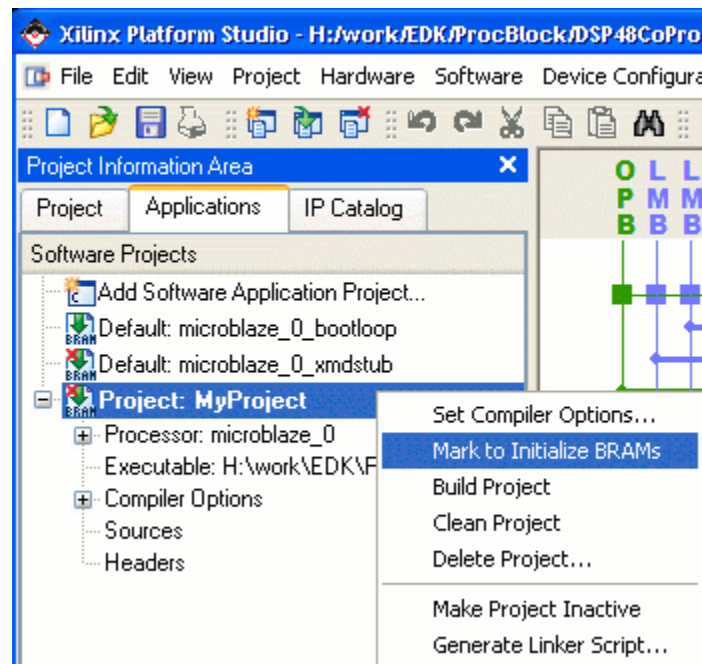
11. Base System Builder – System Created. This dialog shows a summary of the options selected and click on Generate will cause the EDK project to be created.
12. Base System Builder – Finished. The last dialog page shows the files generated. Click finish will end the Base System Builder and allow you to continue the EDK flow using XPS. At this point you will have created a basic EDK system.

Adding a new software application

1. To add a new software application to an EDK Project, first open up the EDK project in the EDK.
2. In the *Project Information Area*, click on the *Applications* tab to reveal the *Software Projects* page.



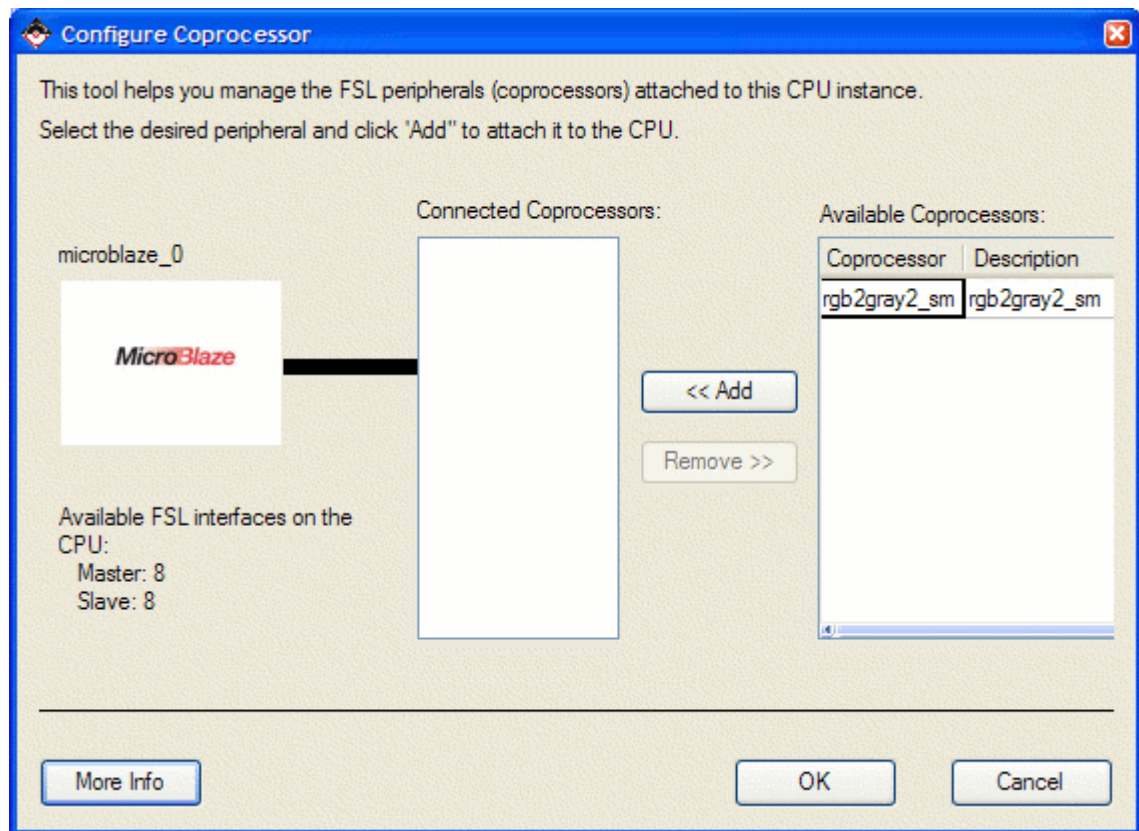
3. The first item on that page is *Add Software Application Project* Double click on that to bring up the *Add Software Application Project* dialog box. Type in a project name and select OK.
4. By default the project is created and not set to be initialized into BRAMS. Make sure to make the project to be initialized into BRAMS, otherwise the software code will not be compiled and added to the bitstream. Also if you have more than one application, ensure that all other applications have *Marked to Initialize BRAM* unchecked.



- Next create Source or header files. Double clicking on the *Sources* branch of a project tree will cause a File Open Dialog to pop-up. The dialog will be rooted at the base location of your EDK project. It is good to create a directory named after your project and keep your source and header files in there; in this case MyProject. Create the directory in the same directory as your EDK .xmp file.

Adding a pcore to an EDK project

- Pcores in an EDK project must be in the user repository, or in a directory named pcores, in the same directory level as the EDK project file.
- To ensure that the pcore has been loaded, from XPS, select Project -> Rescan User Repositories.
- Pcores in System Generator are currently FSL based, so we may use the Configure Coprocessor tool. The tool can be launched from XPS by selecting Hardware -> Configure Coprocessor...



Available FSL based pcores are listed on the right hand window. Select the relevant pcore and click on the Add button. The Configure Coprocessor tool will take care of connecting of clock and reset signals for the FSL bus. However any user signals must be wired up by the user.

Using FPGA Hardware in the Loop

Compiling a Model for Hardware Co-Simulation

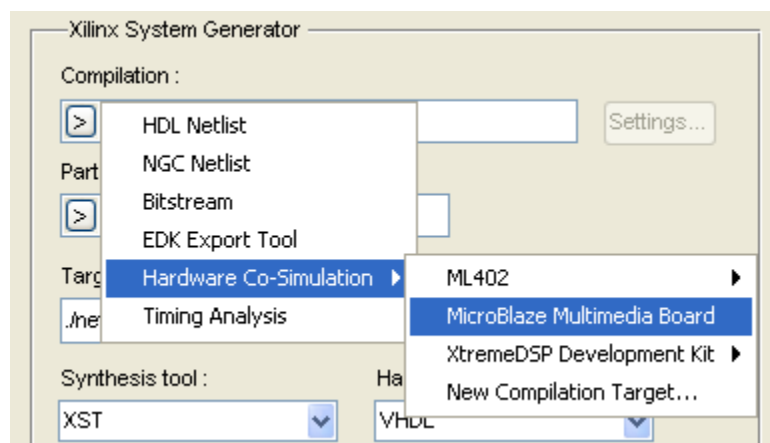
The starting point for hardware co-simulation is the System Generator model or subsystem you would like to run in hardware. A model can be co-simulated, provided it meets the requirements of the underlying hardware platform. This model must include a System Generator block; this block defines how the model should be compiled into hardware. The first step, once you have a model that you are ready to run in hardware, is to open the System Generator block dialog box and select a compilation type under **Compilation**.

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

Choosing a Compilation Target

You may choose the hardware co-simulation platform you would like System Generator to compile code for by selecting an appropriate compilation type in the System Generator block dialog box. Hardware co-simulation targets are organized under the **Hardware Co-Simulation** submenu in the **Compilation** dialog box field. When you install System Generator, several hardware co-simulation compilation targets are automatically installed. You may also install additional plugin compilation targets that add hardware co-simulation support for your FPGA platform.

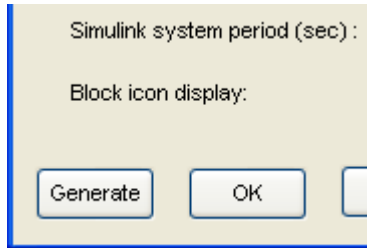
For more information on installing System Generator hardware co-simulation plugins, see [Hardware Co-Simulation Installation](#).



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

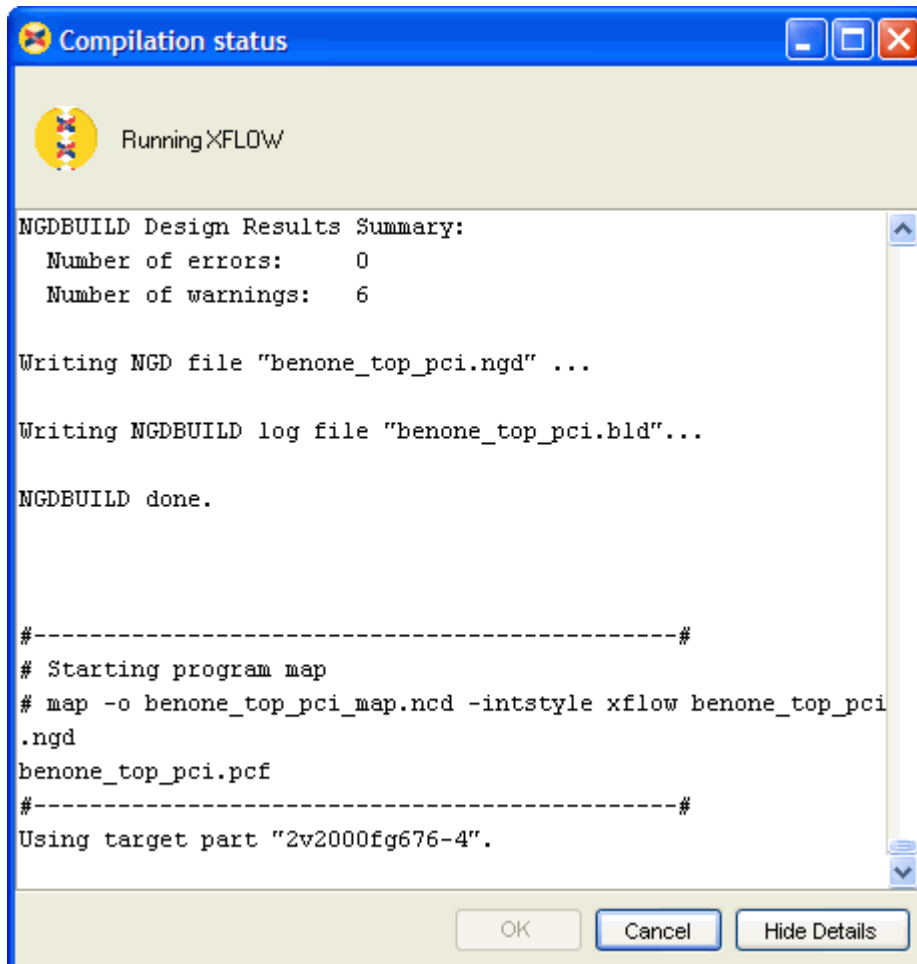
Invoking the Code Generator

Once you have selected a compilation target you can invoke the System Generator code generator to compile the model for hardware co-simulation. The code generator is invoked by pressing the **Generate** button in the System Generator block dialog box.



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

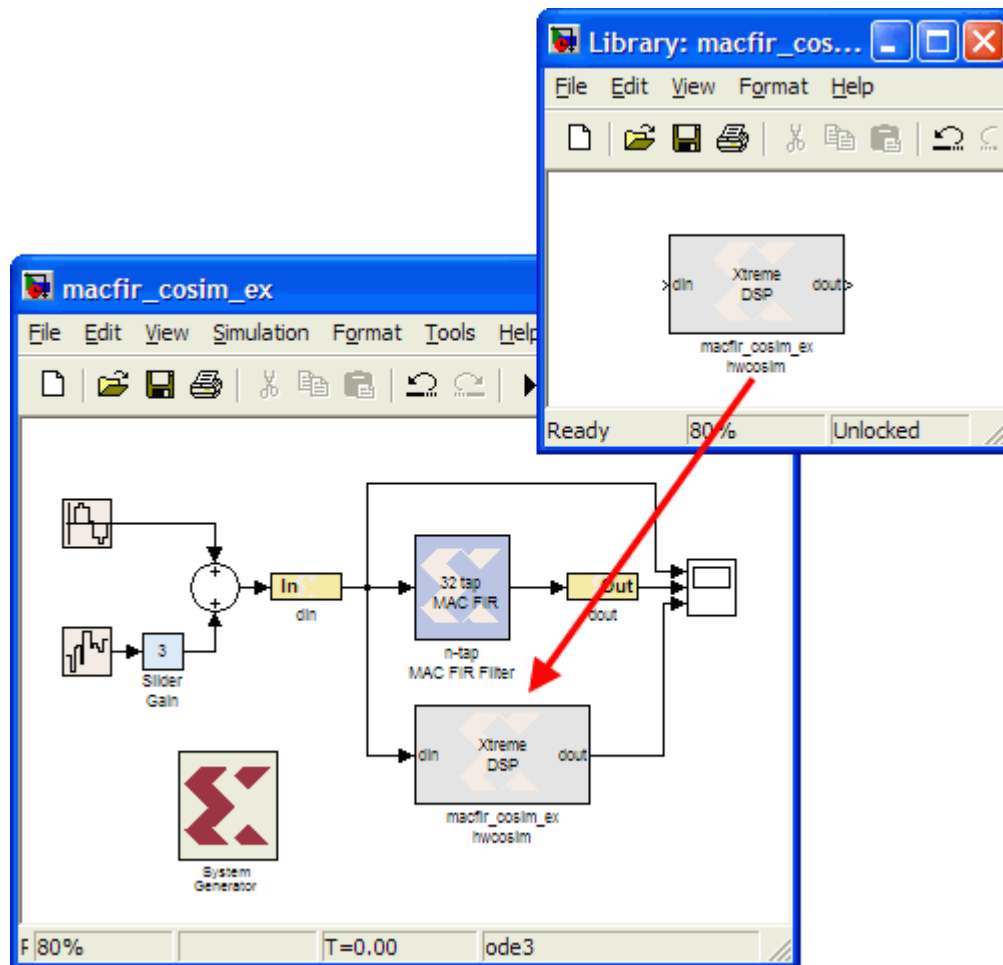
Note: A status dialog box will appear after you press the **Generate** button. During compilation, the status box provides a **Cancel** and **Show Details** button. Pressing the **Cancel** button will stop compilation. Pressing the **Show Details** button exposes details about each phase of compilation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



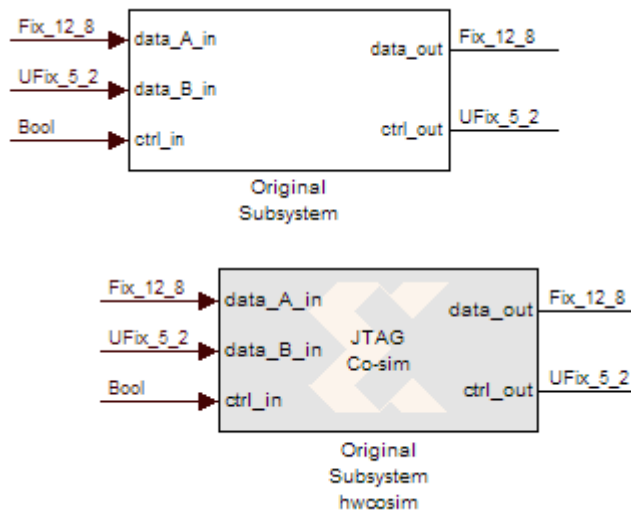
The configuration bitstream contains the hardware associated with your model, and also contains additional interfacing logic that allows System Generator to communicate with your design using a physical interface between the platform and the PC. This logic includes a memory map interface over which System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

Hardware Co-Simulation Blocks

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



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



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

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

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

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

Hardware Co-Simulation Clocking

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

Single-Step Clock

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

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

Free-Running Clock

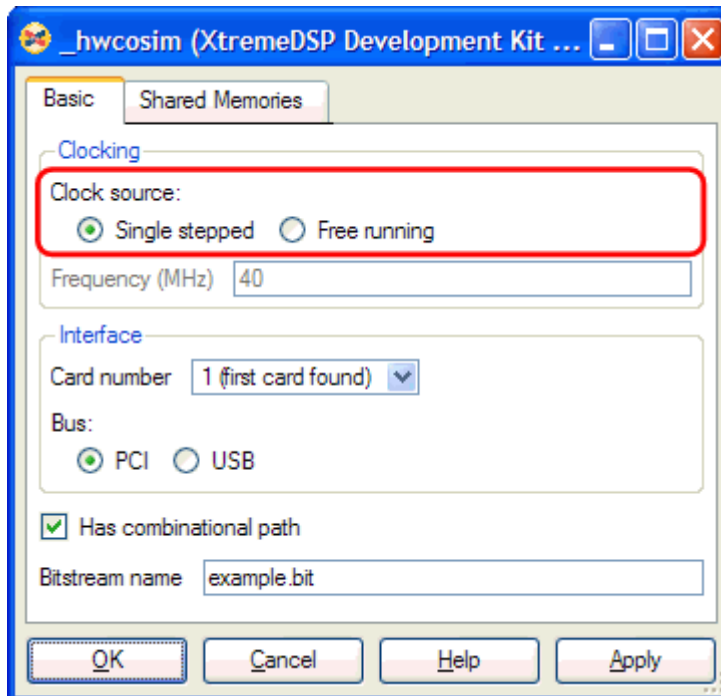
In free-running clock mode, the hardware runs asynchronously relative to the software simulation. Unlike the single-step clock mode, where Simulink effectively generates the FPGA clock, in free-running mode, the hardware clock runs continuously inside the FPGA itself.

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

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

Selecting the Clock Mode

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



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

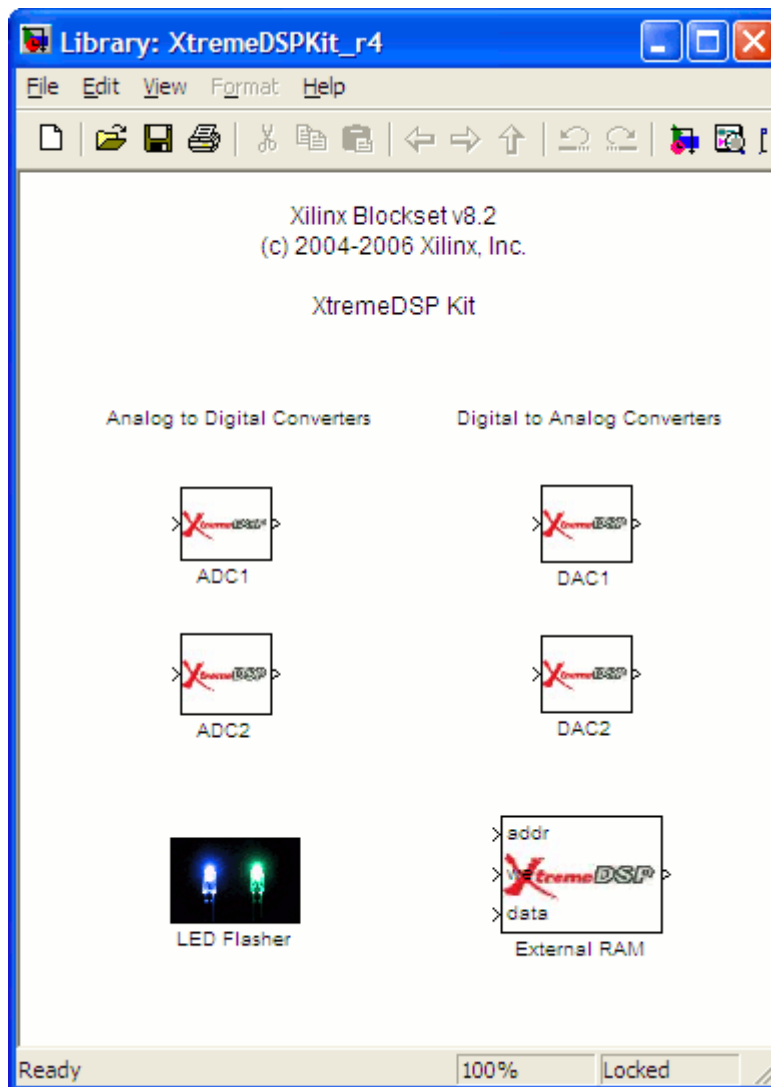
Board-specific I/O Ports

FPGA platforms often include a variety of on-board devices (e.g., external memory, analog to digital converters, etc.) that the FPGA can communicate with. For a variety of reasons, it may be useful to form connections to these components in your System Generator models, and to use these components during hardware co-simulation. For example, if your board includes external memory, you may want to define the control and interface logic to this memory in your System Generator design, and use the physical memory during hardware co-simulation.

You can interface to these types of components by including board-specific I/O ports in your System Generator models. A board-specific port is a port that is wired to an FPGA pad when the model is compiled for hardware co-simulation. Note that this type of port differs from standard co-simulation ports that are controlled by a corresponding port on a hardware co-simulation block.

A board-specific I/O port is implemented using special *non-memory mapped gateway blocks* that tell System Generator to wire the signals to the appropriate FPGA pins when the model is compiled into hardware. To connect a System Generator signal to a board-specific port, connect the appropriate wire to the special gateway (in the same way as is done for a traditional gateway).

Non-memory mapped gateways that are common to a specific device are often packaged together in a Simulink subsystem or library. The XtremeDSP Development Kit, for example, provides a library of external device interface subsystems, including analog to digital converters, digital to analog converters, LEDs, and external memory. The interface subsystems are constructed using Gateways that specify board-specific port connections. These subsystems are treated like other System Generator subsystems during simulation (i.e., they perform double precision to Xilinx fixed-type conversions). When System Generator compiles the design into hardware it connects the signals that are associated with the Gateways to the appropriate external devices they signify in hardware.



I/O Ports in Hardware Co-simulation

A hardware co-simulation block does not include board-specific ports on its external interface. This means that if a model includes a gateway that corresponds to a board-specific port, the corresponding port is connected to the simulation model instead of the actual hardware when the design is compiled for hardware co-sim. To leave the port connected to a real port, use a non-memory mapped gateway instead. See the section on non memory mapped ports in supporting new platforms.

JTAG Hardware Co-Simulation

Hardware Requirements

An FPGA platform can support the JTAG hardware co-simulation interface, provided it includes the following hardware components:

- A Xilinx FPGA part that is available in System Generator as a supported device (i.e., a device that can be chosen in the **Part** field of the System Generator block dialog box);
- An on-board oscillator that supplies the FPGA with a free-running clock source;
- A JTAG header that provides access to the FPGA.

Supporting New Platforms

Although the JTAG hardware co-simulation interface is generic, an FPGA platform must provide its own board support package before it can be supported in System Generator. A board support package is comprised of four files that provide information about the board, or platform. A number of FPGA platforms already have board support packages available. See Hardware Co-Simulation Installation for more information on how to download these files.

You may have an FPGA platform that does not have a hardware co-simulation board support package. In this case, you can create your own, assuming your platform meets the specified hardware requirements. Creating a new board support package for a platform is a straightforward process. System Generator provides a utility, the System Generator Board Description Builder (SBDBuilder), that allows you to create new board support packages in a graphical environment. It is also possible to define board support packages manually by editing a series of template files that are included in the System Generator software tree.

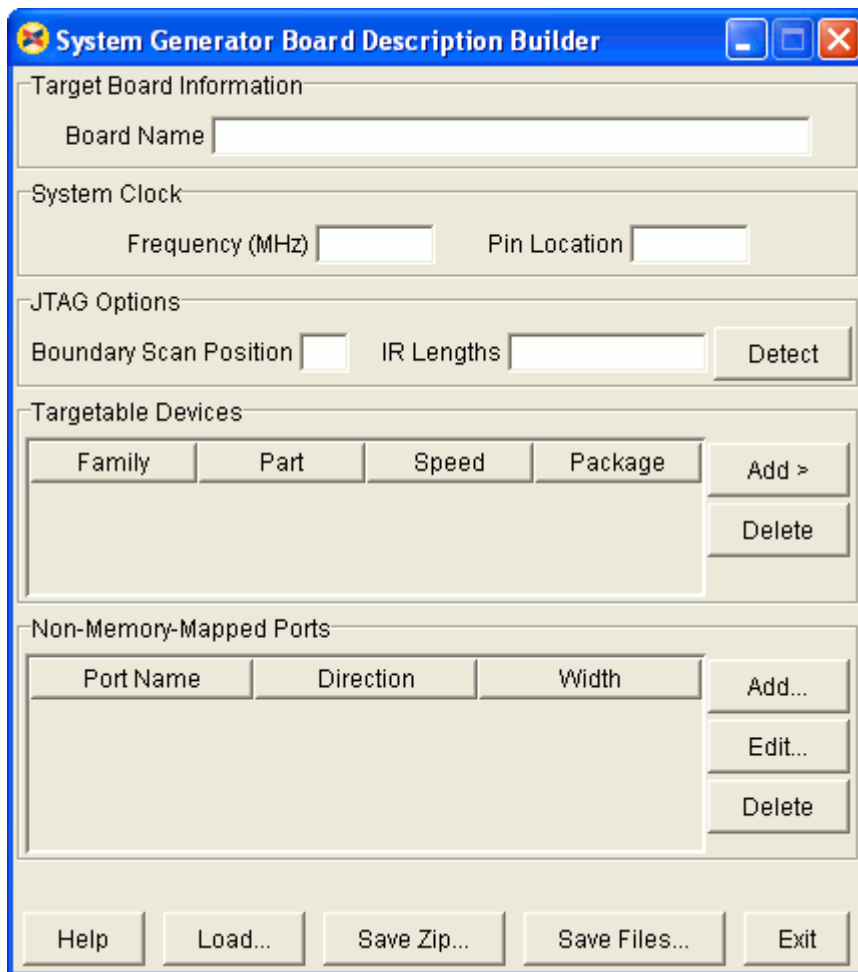
Using SBDBuilder

SBDBuilder is a graphical utility that makes it easy to specify and generate new board support packages. There are three ways to invoke the SBDBuilder application:

- From the System Generator Toolbar menu, select *Tools->Create Plugins*;
- From the *Compilation* menu on the System Generator block dialog box, choose *Hardware Co-Simulation -> New Compilation Target*;
- From the MATLAB command window, type: *xlSBDBuilder*.

SBDBuilder Dialog Box

After invoking SBDBuilder, the main dialog box will appear as shown in Figure 1.



System Generator Board Description Builder

Target Board Information

Board Name

System Clock

Frequency (MHz) Pin Location

JTAG Options

Boundary Scan Position IR Lengths Detect

Targetable Devices

Family	Part	Speed	Package

Add > Delete

Non-Memory-Mapped Ports

Port Name	Direction	Width

Add... Edit... Delete

Help Load... Save Zip... Save Files... Exit

Figure 1: SBDBuilder main dialog box

Once the main dialog box is open, you may create a board support package by filling in the required fields described below :

- **Board Name**: Tells a descriptive name of the board. This is the name that will be listed in System Generator when selecting your JTAG hardware co-simulation platform for compilation.
- **System Clock**: JTAG hardware co-simulation requires an on-board clock to drive the System Generator design. The fields described below specify information about the board's system clock:
 - **Frequency (MHz)**: Specifies the frequency of the on-board system clock in MHz.
 - **Pin Location**: Specifies the FPGA input pin to which the system clock is connected.
- **JTAG Options**: System Generator needs to know several things about the FPGA board's JTAG chain to be able to program the FPGA for hardware co-simulation. Section Obtaining Platform Information describes how and where to find the information required for these fields. If you are unsure of the specifications of your board, please reference the manufacturer's documentation. The fields specific to JTAG Options are described below:
 - **Boundary Scan Position**: Specifies the position of the target FPGA on the JTAG chain. This value should be indexed from 1. (e.g. the first device in the chain has an index of 1, the second device has an index of 2, etc.)
 - **IR Lengths**: Specifies the lengths of the instruction registers for all of the devices on the JTAG chain. This list may be delimited by spaces, commas, or semicolons.
 - **Detect**: This action will attempt to identify the IR Lengths automatically by querying the FPGA board. The board must be powered and connected to a Parallel Cable IV for this to

function properly. Any unknown devices on the JTAG chain will be represented with a "?" in the list, and must be specified manually.

- **Targetable Devices:** This table displays a list of available FPGAs on the board for programming. This is not a description of all of the devices on the JTAG chain, but rather a description of the possible devices that may exist at the aforementioned boundary scan position. For most boards, only one device needs to be specified, but some boards may have alternate, e.g., a choice between an xcv1000 or an xcv2000 in the same socket. Use the **Add** and **Delete** buttons described below to build the device list:
 - **Add:** Brings up a menu to select a new device for the board. Devices are organized by family, then part name, then speed, and finally the package type (Figure 2).
 - **Delete:** Remove the selected device from the list.

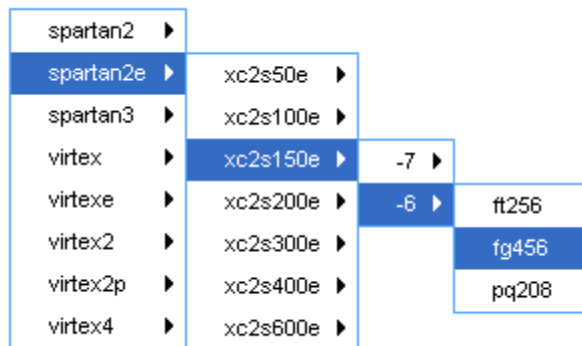


Figure 2: Target device selection

- **Non-Memory-Mapped Ports:** You can add support for your own board-specific ports when creating a board support package. Board-specific ports are useful when you have on-board components (e.g., external memories, DACs, or ADCs) that you would like the FPGA to interface to during hardware co-simulation. Board specific ports are also referred to as non-memory-mapped because when the design is compiled for hardware co-simulation, these ports will be mapped to their physical locations, rather than creating Simulink ports. See *Specifying Non-Memory Mapped Ports* for more information. The **Add**, **Edit**, and **Delete** buttons provide the controls needed for configuring non-memory mapped ports.
 - **Add:** Brings up the dialog to enter information about the new port.
 - **Edit:** Make changes to the selected port.
 - **Delete:** Remove the selected port from the list.
- **Help:** Displays this documentation.
- **Load:** Fill in the form with values stored in an SBDBuilder Saved Description XML file. This file is automatically saved with every plugin that you create, so it is useful for reloading old plugin files for easy modification.
- **Save Zip:** Prompts you for a filename and path to store the plugin to. This will create a zip file with all of the plugin files for System Generator. The zip will be in a suitable format for passing to the System Generator xInstallPlugin function.
- **Save Files:** Prompts you for a directory in which to save all of the raw plugin files. See *Board Support Package Files* for a description of each of the files that are created.
- **Exit:** Quit the application.

Specifying Non-Memory Mapped Ports

You may use SBDBuilder to specify the non-memory mapped ports for your FPGA platform. When you choose to **Add** or **Edit** a non-memory mapped port from the main dialog, the port editor dialog will come up (Figure 3).

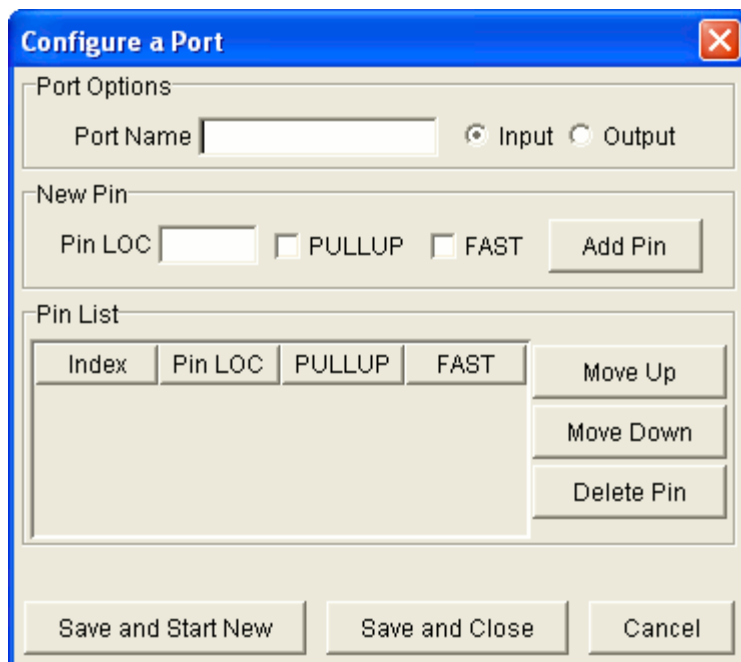
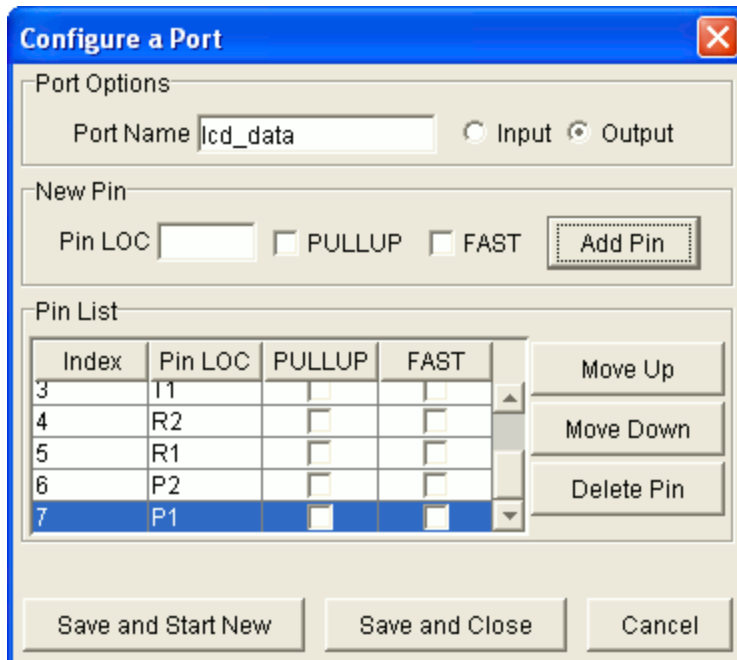


Figure 3: Port editor dialog box

The port editor dialog presents the following controls for port configuration:

- **Port Options:** Specifies the options that will affect the entire port.
 - **Port Name:** This is the name that will describe the port in System Generator. It should be a MATLAB-compatible name (begins with a letter, followed by only letters, numbers, and underscores).
 - **Input/Output:** Specifies the direction of the port.
- **New Pin:** This is the entry point to add pins to a port. Ports may consist of a single pin for a Boolean value, or multiple pins for a vector or bus.
 - **Pin LOC:** Defines the absolute placement of the pin within the FPGA by specifying a location constraint. It is necessary to define this for every pin to make sure that the FPGA programming corresponds to the actual hardware connections.
 - **PULLUP:** A constraint that can be applied to each pin. It guarantees a logic High level to allow 3-stated nets to avoid floating when not being driven.
 - **FAST:** A constraint that can be applied to each pin. It increases the speed of an IOB output. FAST produces a faster output but may increase noise and power consumption.
 - **Add Pin:** Add a pin to the port. Note that the pin is not part of the port until this button is selected.
Tip: Pressing 'enter' while the cursor is in the Pin LOC field is equivalent to pressing this button.
- **Pin List:**
 - **Index:** (Cannot edit directly) Since a port can be more than one bit, it is represented as a vector of pins. The index indicates which bit position a particular pin represents in the port. Zero is the least-significant bit.
 - **Move Up/Down:** Move the selected pin up or down in the pin list. This is useful to correct the vector bit-ordering of the port.
 - **Delete Pin:** Removes the selected pin from the list.
- **Save and Start New:** Save the port to the board support package. The form will then be cleared so that you may enter a new port.
- **Save and Close:** Save the port to the board support package and return to the main screen.
- **Cancel:** Discard changes to the current port and return to the main screen.

When you are finished entering a port, it will look similar to the dialog box shown in Figure 4.



The dialog box titled "Configure a Port" contains the following sections:

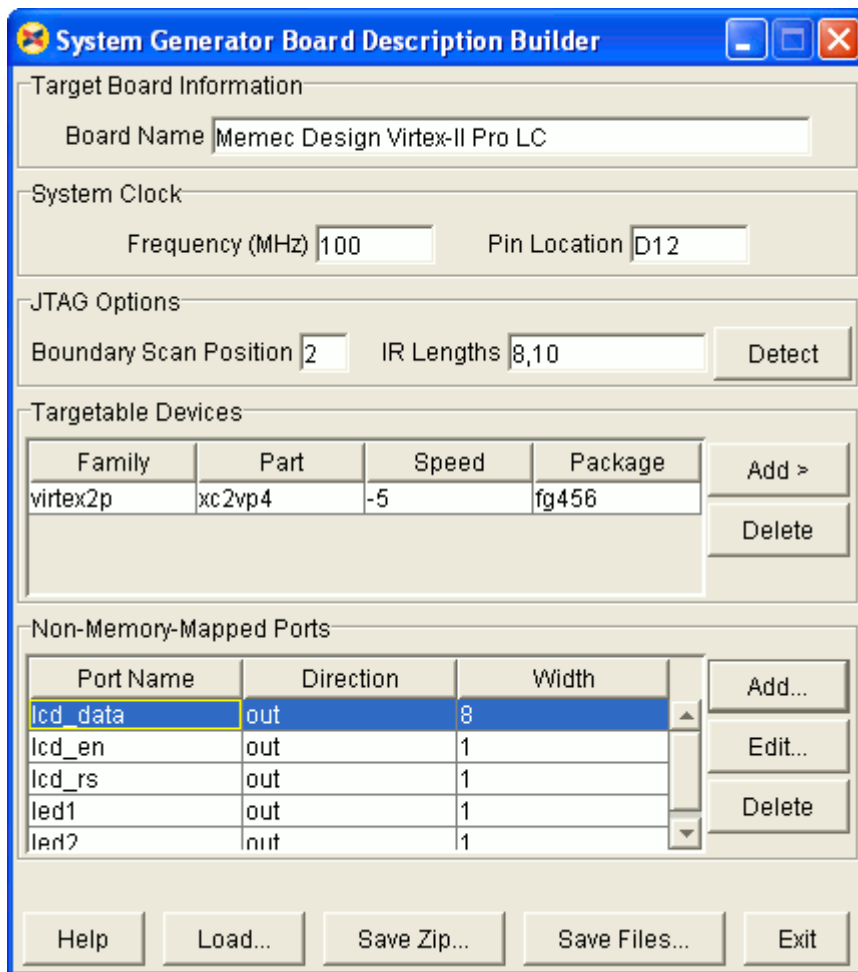
- Port Options:** A text field for "Port Name" containing "lcd_data", and radio buttons for "Input" and "Output", with "Output" selected.
- New Pin:** A text field for "Pin LOC", checkboxes for "PULLUP" and "FAST", and an "Add Pin" button.
- Pin List:** A table with columns "Index", "Pin LOC", "PULLUP", and "FAST". It contains five rows of data. To the right of the table are buttons for "Move Up", "Move Down", and "Delete Pin".
- Buttons:** "Save and Start New", "Save and Close", and "Cancel" at the bottom.

Index	Pin LOC	PULLUP	FAST
3	T1	☐	☐
4	R2	☐	☐
5	R1	☐	☐
6	P2	☐	☐
7	P1	☐	☐

Figure 4: Example port configuration

Saving Plugin Files

Once you have filled out the dialog box with information about your platform, it should resemble the dialog box shown in Figure 5.



System Generator Board Description Builder

Target Board Information
Board Name: Memec Design Virtex-II Pro LC

System Clock
Frequency (MHz): 100 Pin Location: D12

JTAG Options
Boundary Scan Position: 2 IR Lengths: 8,10 Detect

Targetable Devices

Family	Part	Speed	Package
virtex2p	xc2vp4	-5	fg456

Add > Delete

Non-Memory-Mapped Ports

Port Name	Direction	Width
lcd_data	out	8
lcd_en	out	1
lcd_rs	out	1
led1	out	1
led2	out	1

Add... Edit... Delete

Help Load... Save Zip... Save Files... Exit

Figure 5: Example board configuration information

At this point, you can save the board support package into a System Generator plugin zip file or as the raw board support package files described in the Board Support Package Files section, plus the additional SBDBuilder files described below:

- **yourboard.xml**: This is the SBDBuilder Saved Description, which allows SBDBuilder to reload plugins you have previously created. The name you select for this file ('yourboard') will propagate into the names of the other files as well.
- **yourboard_libgen.m**: Automates the process of creating the gateways for the non-memory-mapped ports on this device. Running this script will result in the creation a library like that of Figure 6.

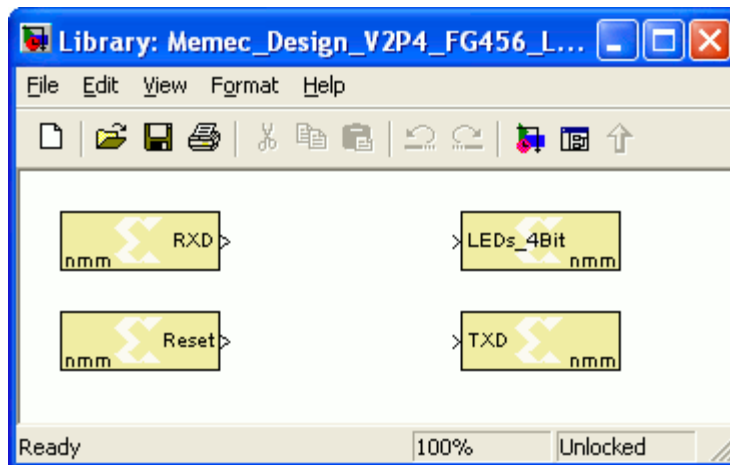


Figure 6: Example generated port library

Board Support Package Files

An FPGA platform that supports JTAG hardware co-simulation is defined in System Generator by its board support package. This package tells System Generator useful information about the platform, such as the appropriate part settings and additional information related to the JTAG and Boundary Scan interface provided by the platform. A board support package is comprised of the files listed below:

Note: In this document, three of the filenames are prefixed by the name `yourboard`. This prefix should be replaced with a moniker suitable for your board (e.g., `xtremedsokit`, `mbazedemo`).

1. `xltarget.m` – Tells System Generator that your FPGA platform is a compilation target. There is a unique `xltarget.m` file for each compilation target. This function tells the tool the name of the compilation target (this name is shown in the compilation target field of the System Generator block dialog box) and also the name of the function where it can look for information about the particular board.
2. `yourboard_target.m` – Configures the System Generator block dialog box with information about the FPGA platform, including device and part information, clock frequency, and the location of the clock pin.
3. `yourboard_postgeneration.m` – Tells System Generator the scripts to run after HDL netlist generation in order to produce an FPGA configuration file that is suitable for your platform. It also specifies non-System Generator block dialog box related information, including the position of the device in the platform's Boundary Scan chain and the instruction register lengths of each device. This function is referred to as a post-generation function.
4. `yourboard.ucf` – User Constraints File (UCF) for the FPGA platform. Specifies clock pin location and frequency, and optionally constrains any board-specific ports.

Included in the System Generator software tree are templates for the files listed above. If you would

like to manually support a new board, you may customize each of the four template files with information that is specific to your platform. You must also rename the files by substituting a suitable name in place of the yourboard prefix.

Each template is fully annotated with step-by-step instructions that indicate which fields should be modified, and the types of values that should be given to these fields. The fields that must be modified are underlined using “~~~” notation. The template files can be found in the `sysgen/hwcosim/jtag/templates` directory of your System Generator install tree.

Obtaining Platform Information

SBDBuilder (or alternatively, the board support package template files) require certain information about your FPGA platform. The table below lists the information you need.

Information	Description
Clock pin location	Pin location constraint for the FPGA system clock source.
Clock period	Period constraint for the FPGA system clock source.
Device position in the Boundary Scan Chain	Tells the position of the target FPGA in the platform's Boundary Scan Chain. Indexing begins at 1, with device 1 being the first device in the chain.
Instruction register lengths	Instruction register length of every device in the Boundary Scan Chain.

You may obtain the clock pin location and period from any number of possible sources, including the vendor documentation, existing constraints files, or vendor online documentation/support.

If you do not know which devices are in your platform's boundary scan chain, you may use *iMPACT* to assist you in finding this information. *iMPACT* is a tool that is included with the Xilinx ISE software that allows you to perform device configuration and file generation functions. When the tool is invoked, it automatically detects the contents of your platform's boundary scan chain, and display these contents graphically.

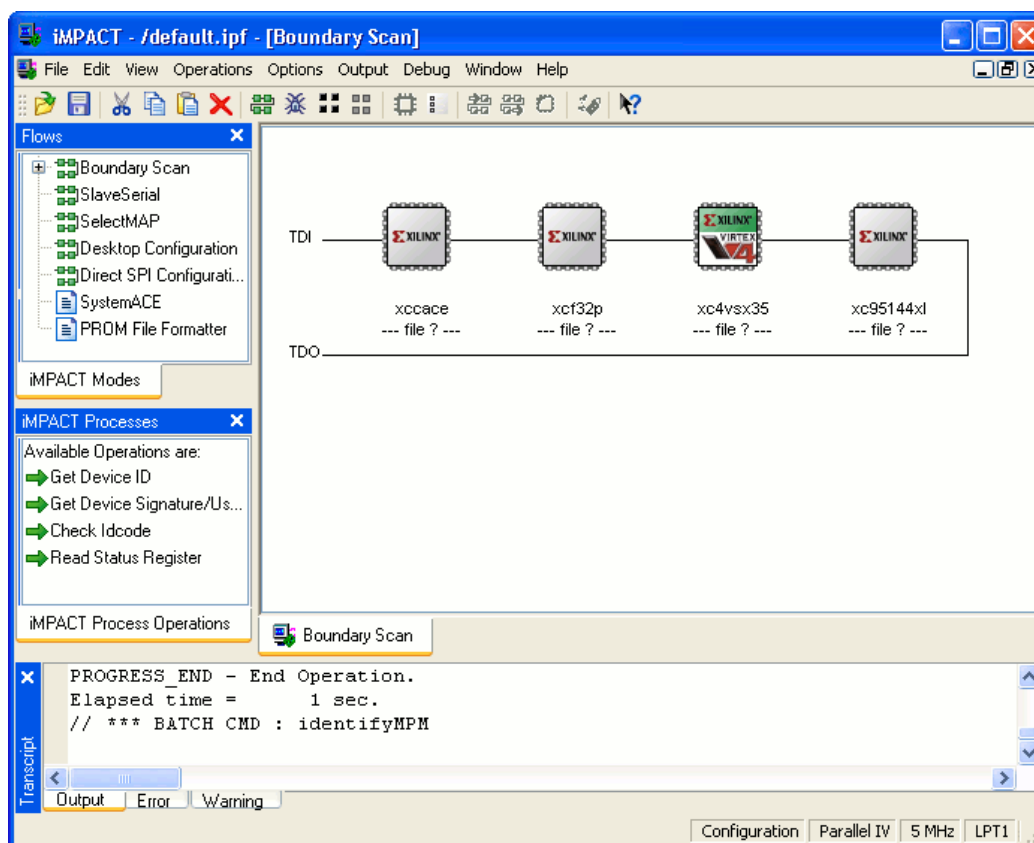


Figure 7: iMPACT screenshot showing Boundary Scan Chain devices.

Once you have determined which devices are in the Boundary Scan Chain, you must determine the instruction register lengths for each device. The table below specifies the instruction register lengths for various Xilinx families. You may use the auto detection capability of SBDBuilder to determine the instruction register lengths. If this utility does not work, you may use the following table to find the instruction register lengths for a particular part family.

Family	IR Length
XC9500 / XC9500XL / XC9500XV	8
XC1800 / XC18V00	8
XC4000XL/XLA	3
Spartan-XL	3
System_ACE-CF	8
Virtex / Virtex-E(EM)	5
Spartan-II / Spartan-IIE	5
Virtex-II	6
Spartan-3	6
Spartan-3E	6
Virtex-II Pro 2, 4, 7	10
Virtex-II Pro 20, 30, 40, 50, 70, 100	14

Virtex-II Pro 125	16
Virtex-4 LX	10
Virtex-4 SX	10
Virtex-4 FX 12, 20	10
Virtex-4 FX 40, 60, 100, 140	14
Virtex-5 LX	10
XCR3000XL	5
XCR3000A / XCR3128	4
XCR3320 / XCR3960	5
XCR5128 / XCR5032C / XCR5064C / XCR5128C	4
CoolRunner-II	8
Platform Flash XCFxxS	8
Platform Flash XCFxxP	16

Manual Specification of Board-Specific Ports

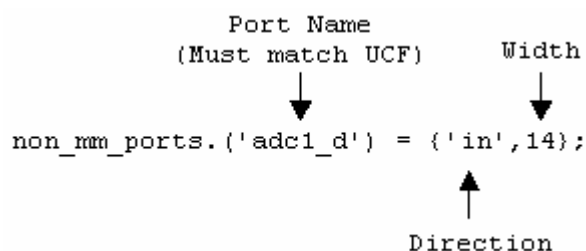
You can manually specify your own board-specific ports when creating a board support package. To define board-specific ports for your FPGA platform, you must do the following:

- Add all board-specific ports to the `yourboard.ucf` template file. Each constraint should be accompanied by a special comment, `<port> contingent`, where `<port>` is the name of the board specific port. When System Generator compiles a model for hardware, it creates a custom UCF file. Constraints associated with signals that aren't used in the model are removed from the custom UCF file.

Example constraints for ports `adc1_d(0)` and `adc1_d(1)`:

```
net adc1_d(0) loc = af20; # adc1_d contingent
net adc1_d(1) loc = ad18; # adc1_d contingent
```

- Declare all board-specific ports in the `yourboard_postgeneration.m` function.



```
non_mm_ports.{'adc1_d'} = {'in',14};
```

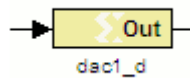
Note: Bi-directional ports are currently not supported.

Include this line in `yourboard_postgeneration.m` function:

```
params.('non_memory_mapped_ports') = non_mm_ports;
```

- Customize a gateway with the board-specific port information:

- Create a library and add a gateway;
- Name the Gateway with the name of your board specific port (this name must match the port name used in the post-generation function and UCF file);



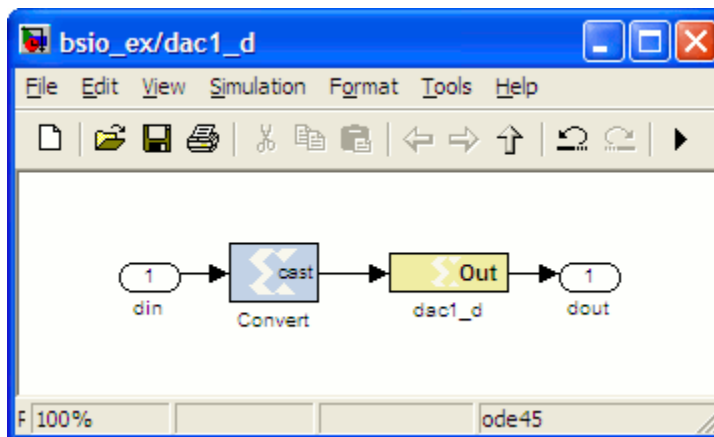
- Select the Gateway by clicking on it;
- In the MATLAB command window, type the following:

```
>> xlSetNonMemMap(gcb, 'Xilinx', 'jtaghwcosim');
```

- Save the library.

You are now ready to use your board-specific gateway in System Generator. When you include the gateway in your model, you must make sure the signals that drive (or are driven by) the gateway have widths that match the widths of the ports in hardware. You can force the width of a signal driving a gateway out by preceding it with a convert block.

Note: A subsystem is a convenient place to store the gateway out and convert block pairs.



Providing Your Own Top level

When a model is compiled for JTAG hardware co-simulation, System Generator produces a generic top level HDL entity for the design. This entity instantiates the logic required by the model and the interfacing logic required for JTAG hardware co-simulation.

Sometimes your platform may have a special requirement that precludes you from using this generic top level. For example, your platform may have components that rely on clocks that are generated by a DCM that resides in the platform's FPGA. In these situations, System Generator allows you to use your own top level netlist when it compiles the model into hardware.

Note: If you choose to use your own top level component, you must provide a previously synthesized version (.ngc, .edf, .edn) to System Generator.

Note: Your top level component must instantiate the generic JTAG hardware co-simulation top level component. The component instantiation must include the required clocking signals, plus any board-specific I/O ports your board may support. An example component instantiation is provided below:

```
component jtagcosim_top port (
```

```

-- required clocking ports
sys_clk      : in std_logic;
cosim_clk    : out std_logic;
sys_clk_buf  : out std_logic;
-- board specific ports
adc1_d       : in std_logic_vector(13 downto 0);
dac1_d       : out std_logic_vector(13 downto 0);
dac1_div0    : out std_logic;
dac1_div1    : out std_logic;
dac1_mod0    : out std_logic;
dac1_mod1    : out std_logic;
dac1_reset   : out std_logic
);
end component;

```

You may specify your own top level netlist in `yourboard_postgeneration.m` as follows:

```
params.vendor_toplevel = 'yourboard_toplevel';
```

Here `yourboard_toplevel` is the name of the precompiled, top level netlist component you would like System Generator to use for the top level. You must also tell System Generator the netlist file names that are associated with the top level component. These files are specified as shown below in `yourboard_postgeneration.m`.

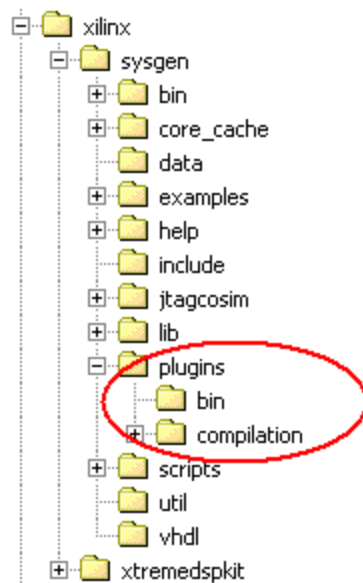
```
params.vendor_netlists = {'yourboard_toplevel.ngc', 'foo.edf'};
```

Installing Board Support Packages

SBDBuilder can generate a plugin zip file for your board support package that may be installed automatically using the `xlInstallPlugin` utility provided with System Generator. You may manually install the board support package files if an appropriate plugin zip file is not provided. This section describes how to install the files manually in your System Generator software tree.

Plugins Directory

The System Generator software provides a special directory in which the board support package files for new compilation targets can be added. This directory, `plugins/compilation`, provides a repository for System Generator compilation target plugins, and has unique properties that are discussed later in this section. Your System Generator software tree should resemble the tree hierarchy shown below.

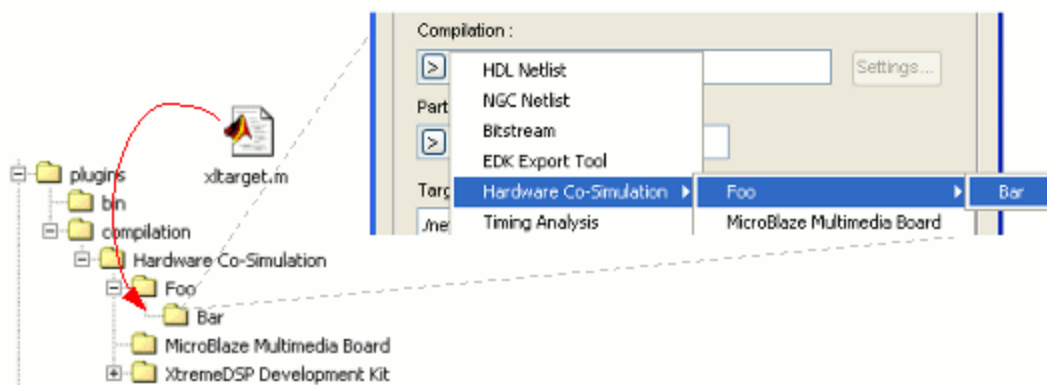


The board support package files for your platform should be saved in a subdirectory, or series of subdirectories, under the `plugins/compilation` directory.

Note: All configuration files associated with a board support package must be saved in the same directory.

System Generator searches this directory (and subdirectories) for compilation targets. Recall that the `xltarget.m` file tells System Generator the platform should be used as a compilation target. When the tool searches the `plugins/compilation` directory, it adds a compilation target to the System Generator block dialog box for every `xltarget.m` file that it encounters.

The System Generator block dialog box Compilation submenus mirror the directory structure under the `plugins/compilation` directory. When you create a new directory, or directory hierarchy, for a board support package, the names of the directories define the taxonomy of the compilation target submenus.



Detecting New Packages

In order for System Generator to recognize the new target, you must tell it to search for new compilation targets by entering the following command in the MATLAB command window:

`xlrehash_xltarget_cache`

You can now select the FPGA platform from the list of compilation targets in the System Generator block dialog box.

Note: If you have a System Generator block dialog box open when you enter this command, it will not show up until you close and re-open the dialog box.

Ethernet Hardware Co-Simulation

Network-based Ethernet Hardware Co-Simulation

The network-based Ethernet hardware co-simulation interface provides co-simulation access to an FPGA platform over an IPv4 network infrastructure. Because IPv4 networks are widespread, the interface provides a straightforward way to communicate with remote hardware connected to either a wired or wireless network. This interface is ideal in situations where the FPGA platform is remote (e.g., across the office or across the country) or when multiple designers must share a single development board.

Interface Features

The interface supports operations in 10/100/1000 Mbps half/full duplex modes. For FPGA device configuration, the interface supports Ethernet-based configuration over the same network connection for co-simulation. This means that a separate programming cable (e.g., Parallel Cable IV) is *not* required.

Note: This co-simulation interface utilizes an evaluation version of the Ethernet MAC core. Because this is an evaluation version of the core, it will become dysfunctional after continuous, prolonged operation (e.g., around 7 hours) in the target FPGA. Operation of the core will restart with a new simulation. For more information about obtaining the full version of the core, please visit the product page at http://www.xilinx.com/xlnx/xebiz/designResources/ip_product_details.jsp?key=TEMAC.

Supported FPGA Development Boards

The Xilinx ML402 and ML506 development platform is currently supported for the network-based Ethernet co-simulation.

Setup Procedures

1. Network-based Ethernet co-simulation performs device configuration over the network configuration. Before using network configuration, you must ensure the IP address, MAC address, and configuration server are properly setup on the System ACE CompactFlash. Refer to the Configuration using System ACE section for information on how to do this.
2. Install the CompactFlash on the target FPGA platform, power up the board, and verify that the configuration server is running properly.
3. The target FPGA listens on the UDP port 9999. Please ensure the underlying network does not block the associated traffic.

Note: If you run into problems using network-based Ethernet hardware co-simulation, please refer to the Troubleshooting section in Configuration using System ACE.

Known Issues

IP fragmentation is not supported by the network-based Ethernet co-simulation interface. Please ensure the connection established between the host and the target FPGA platform can handle a maximum transmission unit (MTU) size of at least 1300 bytes without fragmentation.

Point-to-point Ethernet Hardware Co-Simulation

Point-to-point Ethernet Hardware Co-simulation provides a co-simulation interface using a raw Ethernet connection. The raw Ethernet connection refers to a Layer 2 (a.k.a. Data-Link Layer) Ethernet connection, between a supported FPGA development board and a host PC, with no network routing equipment along the path. By taking the advantage of the ubiquity and advancement of Ethernet technologies, the interface facilitates a convenient and high-bandwidth co-simulation to an external FPGA device.

Interface Features

The interface supports 10/100/1000 Mbps half/full duplex modes. Jumbo Frame is also supported on a Gigabit Ethernet, provided it is enabled by the underlying connection. For FPGA device configuration, the interface supports either JTAG-based configuration over a Parallel IV or a Platform USB cable, or Ethernet-based configuration over the same Point-to-point Ethernet connection for co-simulation.

Note: This co-simulation interface utilizes an evaluation version of the Ethernet MAC core. Because this is an evaluation version of the core, it will become dysfunctional after continuous, prolonged operation (e.g., around 7 hours) in the target FPGA. Operation of the core will restart with a new simulation. For more information about obtaining the full version of the core, please visit the product page at http://www.xilinx.com/xlnx/xebiz/designResources/ip_product_details.jsp?key=TEMAC.

Supported FPGA Development Boards

The Xilinx ML402 and ML506 development platforms are currently supported for the Point-to-point Ethernet co-simulation.

Prerequisites

Co-simulation through the interface requires WinPcap version 4.0, which may be obtained from <http://www.winpcap.org>.

Setup Procedures

1. When using the Point-to-point Ethernet configuration method 1) configure the MAC address and server on the System ACE (refer to Configuration using System ACE for detailed documentation), 2) insert the CompactFlash into the target FPGA platform, power up the board, 3) and verify that the configuration server is running properly (as described later in this document).
2. Install an Ethernet network interface card (NIC) on the host PC. The NIC may operate in 10, 100, or 1000 Mbps.
3. Configure the NIC properly under [Control Panel] > [Network Connections]. Jumbo Frames support can be optionally enabled to maximize the co-simulation performance.
4. Connect the host NIC and the Ethernet connector on the FPGA board through an unshielded twisted pair (UTP) crossover cable.
5. Verify the connectivity by 1) examining the Ethernet status LEDs (if available) on the target FPGA board and; 2) the status of the NIC for use in co-simulation under [Control Panel] > [Network Connections] in the host's Windows environment.

The Ethernet status LEDs should reflect the connection speed and duplex mode under operation, and the NIC status should be shown as “Enabled”.

Co-simulation using Point-to-point Ethernet interface

Hardware co-simulation using the point-to-point Ethernet interface is similar to other interfaces supported by System Generator. It involves the following procedures in general before actually performing the co-simulation in hardware.

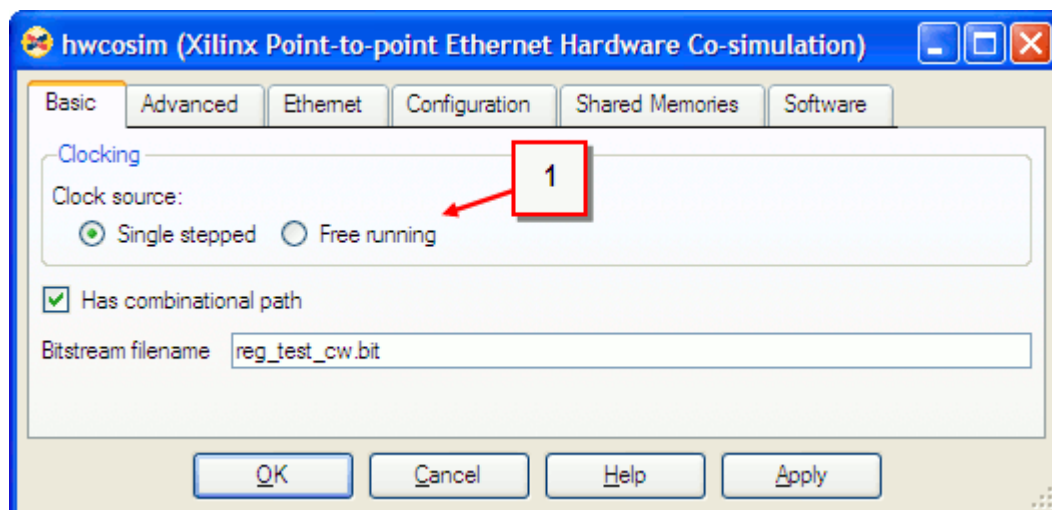
- Compile the design for hardware co-simulation to obtain a hardware co-simulation block.
- Insert the generated co-simulation block into the Simulink diagram, and connect appropriate signals to the corresponding ports on the block.
- Open the parameters dialog box by double clicking on the co-simulation block. Configure the block parameters according to the co-simulation requirements and interface properties.

Configuring co-simulation block parameters

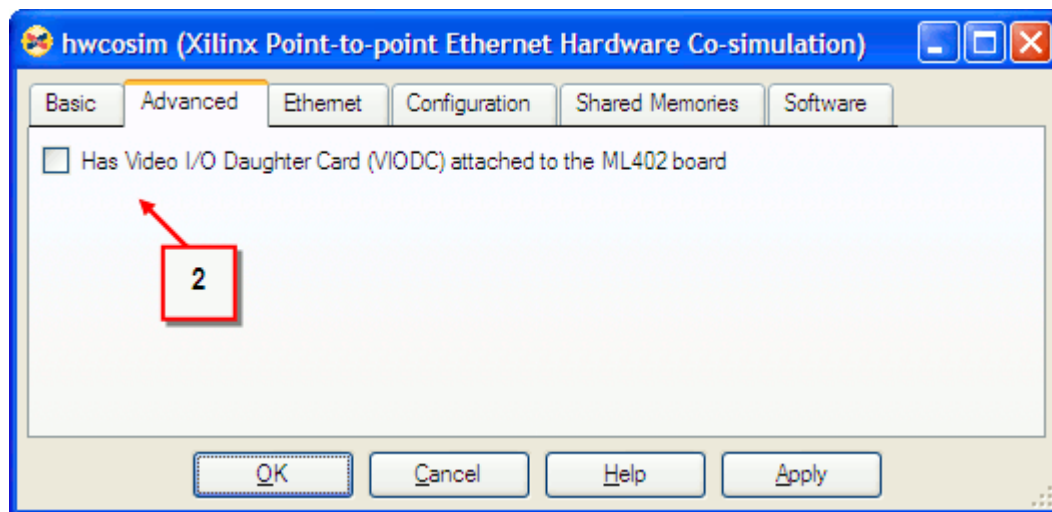
There are several block parameters specific to the point-to-point Ethernet co-simulation interface. The rest of this guide describes step-by-step how to configure the parameters of the point-to-point Ethernet co-simulation block. Refer to Point-to-point Ethernet Co-Simulation in the Block Reference Pages for details of all the block parameters.

General settings

1. Select the appropriate clock source for the co-simulation in the Basic tab.



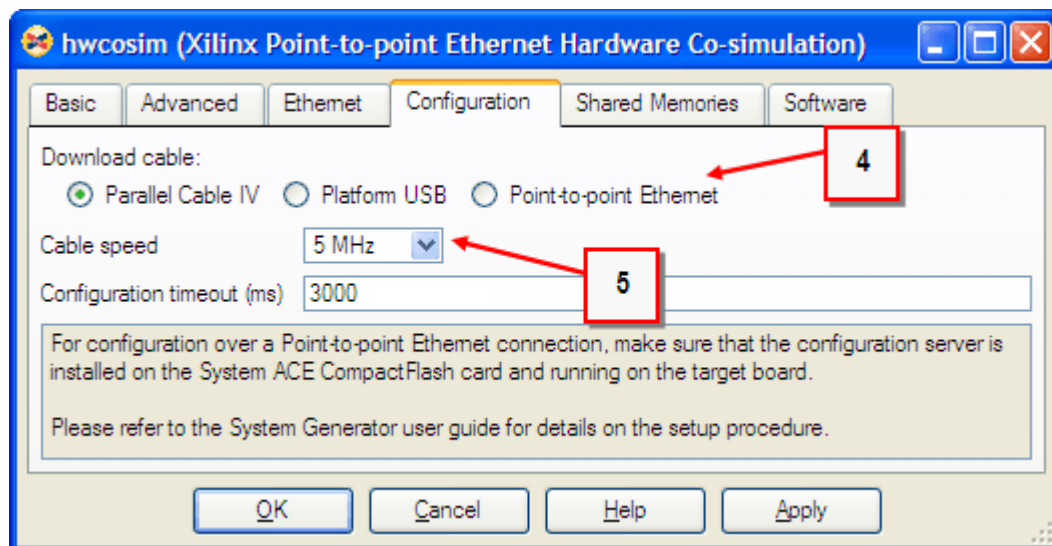
2. If there is a Video I/O daughter card attached to the ML402 board, check the "Has Video I/O Daughter Card (VIODC) on the ML402 board" in the Advanced tab.



Choose the configuration method

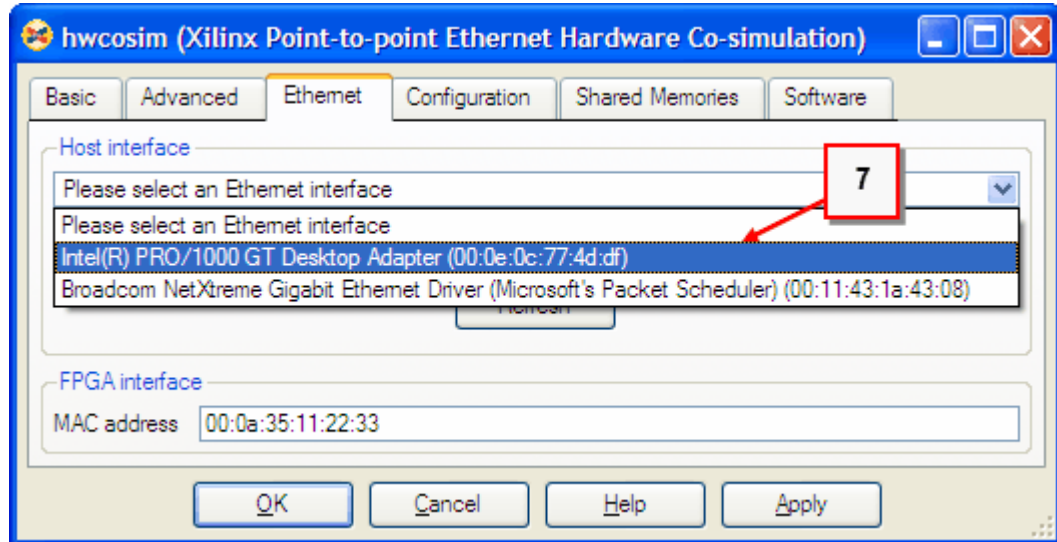
3. Select the Configuration tab.
4. Choose the download cable for configuring the target board.
5. For JTAG-based download cables (Parallel IV or Platform USB), change the cable speed if the default value is not suitable for the cable in use.

Note: Change the configuration timeout value only when necessary. The default value should suffice in most cases. A larger value is needed when it takes a considerable amount of time to re-establish a network connection with the FPGA platform after device configuration completes.



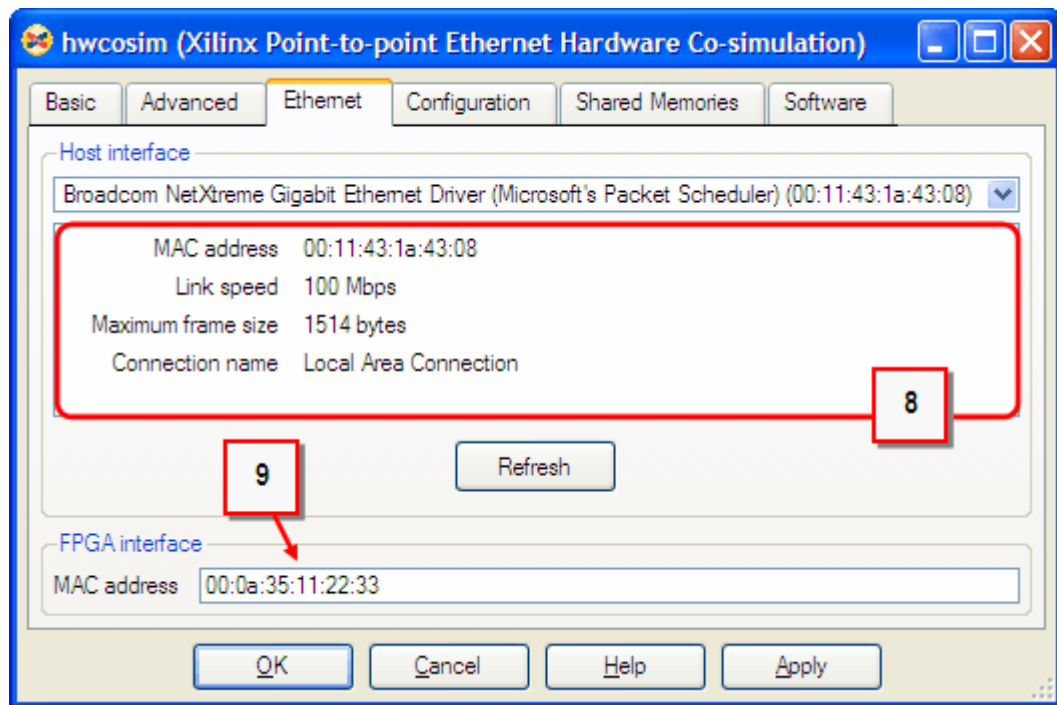
Configure the Ethernet interface settings

6. Select the Ethernet tab.
7. From the host interface panel, navigate the pull down list and select the appropriate network interface used for co-simulation.



Note: The pull down list only shows those Ethernet-compatible network interfaces installed on the host, which support 10/100/1000 Mbps, and are currently enabled and attached to an active Ethernet segment. If the target interface is not listed as expected, examine the connection and click the Refresh button to update the list.

8. The information box beneath the pull down list provides the details about the selected interface. Examine the information to ensure the appropriate interface is chosen, and adjust the network settings in the operating system when necessary.



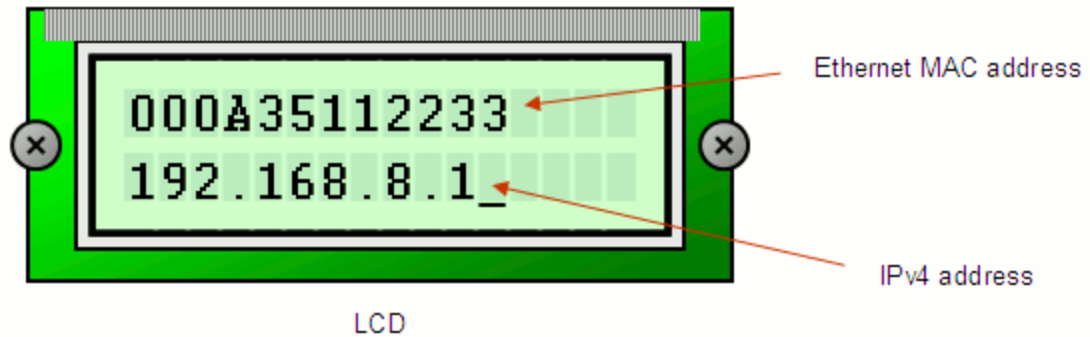
9. Depending on which configuration method is chosen, the MAC address in the FPGA interface panel may need to be changed.

(a) For JTAG-based configuration over a Parallel IV or a Platform USB cable:

The MAC address need not be changed unless default value conflicts with other network equipments on the same Ethernet segment, or when the co-simulation is running over multiple boards. In either case, an arbitrary but non-conflicting MAC address can be assigned to each point-to-point Ethernet co-simulation block.

(b) For point-to-point Ethernet-based configuration:

Observe the MAC address displayed on the LCD screen of the target board when the configuration boot-loader is running. Change the FPGA MAC address in the co-simulation block if the default value does not match the target board. Refer to Configuration using System ACE for details about assigning the MAC address on the target board.

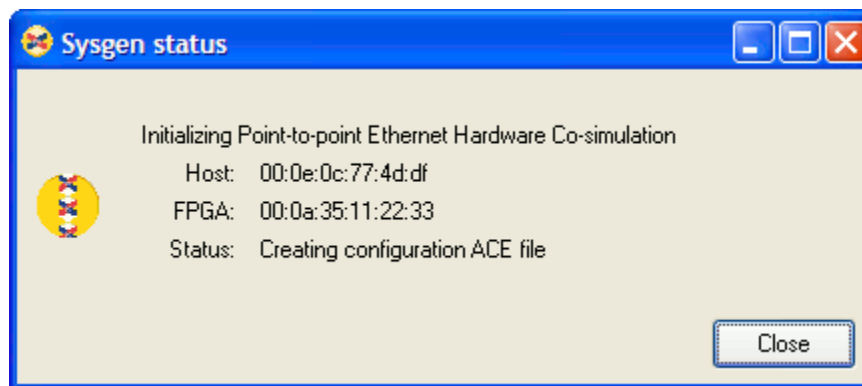


Note: The MAC address must be specified using six pairs of two-digit hexadecimal number separated by colons (e.g. 00:0a:35:11:22:33).

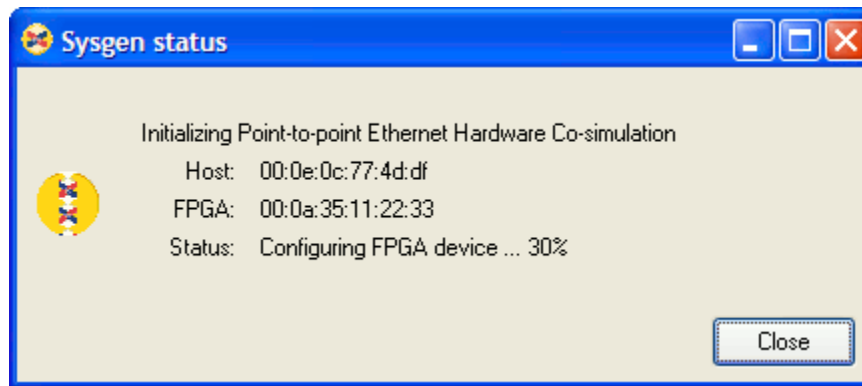
Co-simulating the design

After setting the block parameters appropriately, you may begin co-simulation by pressing the Simulink Play button. System Generator automates the device configuration process and transfers the design under test (DUT) into the FPGA device for co-simulation. A dialog box will be shown to describe the status of the process.

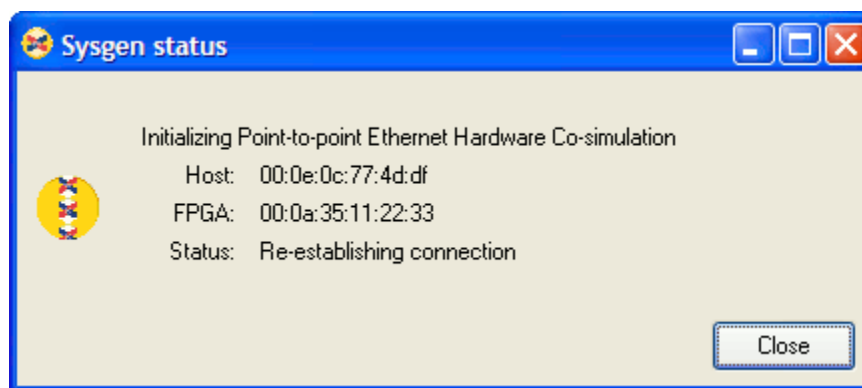
1. The final configuration file is first generated based on the input bitstream specified in the block parameters.



2. The final configuration file is then transferred to the target board using the selected download cable, and used to configure the FPGA device. The progress of configuration is shown in the dialog box when the configuration is performed over a Point-to-point Ethernet connection.



3. Upon the completion of device configuration, the co-simulation engine re-establishes the connection to the target board, and starts co-simulating the design.



Known Issues

If you encounter problems transmitting data through the Point-to-point Ethernet co-simulation interface or experience instability issues, please disable the Hyper-Threading option in the BIOS on an Intel platform.

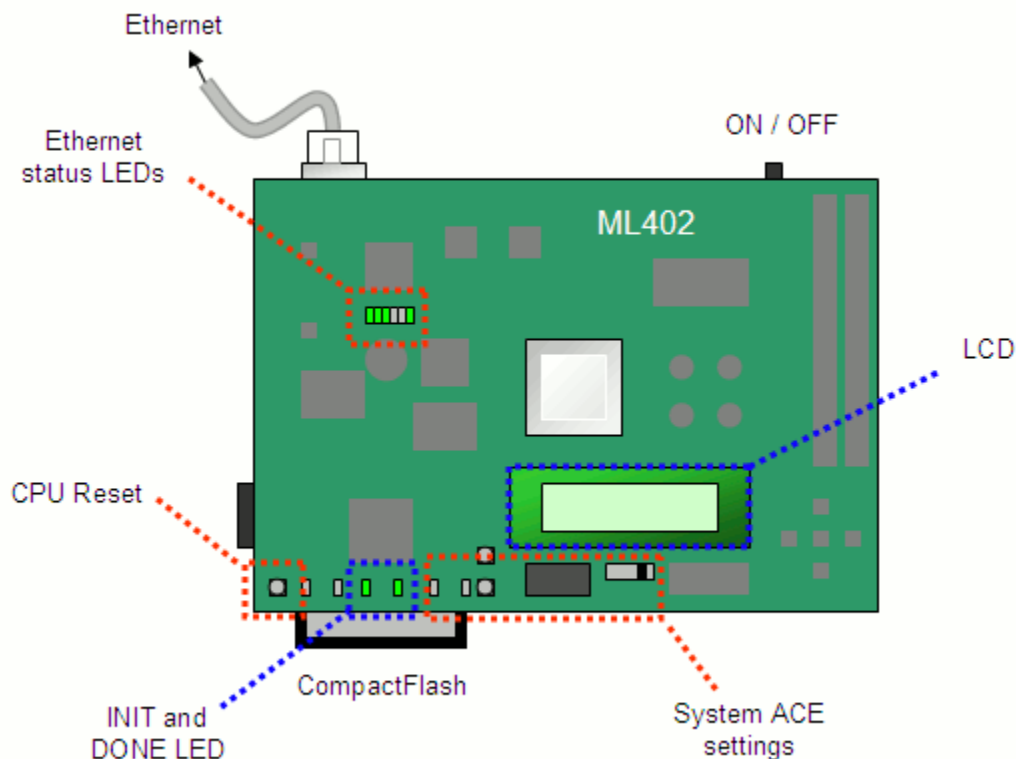
Configuration using System ACE

Xilinx System Advanced Configuration Environment (System ACE) provides a convenient and flexible solution for FPGA configuration. System Generator utilizes the System ACE solution to support device configuration over an Ethernet-connection for both network-based and point-to-point Ethernet hardware co-simulation interfaces. Using this interface eliminates the need for a second programming cable (e.g., a Xilinx Parallel Cable IV) for device configuration.

For point-to-point Ethernet co-simulation, the System ACE configuration is available as the Point-to-point Ethernet option under the Configuration tab on the block parameters dialog box. For network-based Ethernet co-simulation, the System ACE configuration solution is the only configuration option.

Setup Procedure

The procedure for setting up the Xilinx ML402/ML506 development board to support System ACE-based configuration with hardware co-simulation is described below.



Prepare the System ACE CompactFlash card

1. Insert the CompactFlash (CF) card into a CompactFlash reader. For the following steps, we assume 'e:' is the drive name associated with the CompactFlash reader.
2. Extract the default CF image onto the card by typing the following command under the MATLAB console.

For ML402,

```
unzip(fullfile(xlFindSysgenRoot, 'hwcosim/ml402/sysace_cf.zip'),  
'e:')
```

For ML506,

```
unzip(fullfile(xlFindSysgenRoot,  
'plugins/bin/ML506_sysace_cf.zip'), 'e:')
```

Note: The card may need to be formatted to a FAT16 file system before the CF image can be correctly extracted onto card. Please use the *mkdosfs* utility to format the card. The *mkdosfs* program can be obtained from <http://www.xilinx.com/products/boards/ml410/designs/mkdosfs.zip>.

The following is an example command to format the card as FAT16 file system. Type the command under the Windows command prompt:

```
mkdosfs -v -F 16 e:
```

Warning: ensure the drive name (e.g., 'e:') is specified correctly prior to running the program.

Assign an Ethernet MAC address and IPv4 address

3. After writing the CF image to the card, you will find two files, *mac.dat* and *ip.dat*, in the CF card's root directory. The *mac.dat* and *ip.dat* files specify the Ethernet MAC address and IPv4 address associated with the board, respectively. These addresses are used to uniquely identify a target board during Ethernet hardware co-simulation.

Note: Steps 4 and 5 are optional and are necessary only when:

- The default MAC and IP addresses conflict with your default network settings.
 - You wish to co-simulate two or more development boards concurrently.
4. Open *mac.dat* in a text editor to change the Ethernet MAC address. The MAC address must be specified as a six pair of two-digit hexadecimal separated by colons (e.g. 00:0a:35:11:22:33). All-zeros, broadcast, or multicast MAC addresses are not supported.
 5. Open *ip.dat* in a text editor to change the IP address. The IP address must be specified in IPv4 dotted decimal notation (e.g. 192.168.8.1). All-zeros, broadcast, multicast, or loop-back IP address are not supported.

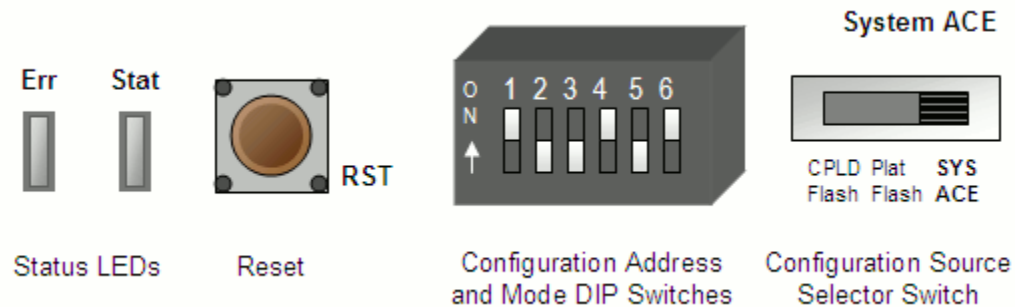
Adjust on-board settings for System ACE

6. Turn off the development board. Install the CompactFlash card (as prepared in the previous steps) securely into the corresponding slot on the development board.

Adjust the on-board settings as follows:

Configuration Address jumpers: [1:on, 2:off, 3:off, 4:on, 5:off, 6:on]
(Address 4 with JTAG mode)

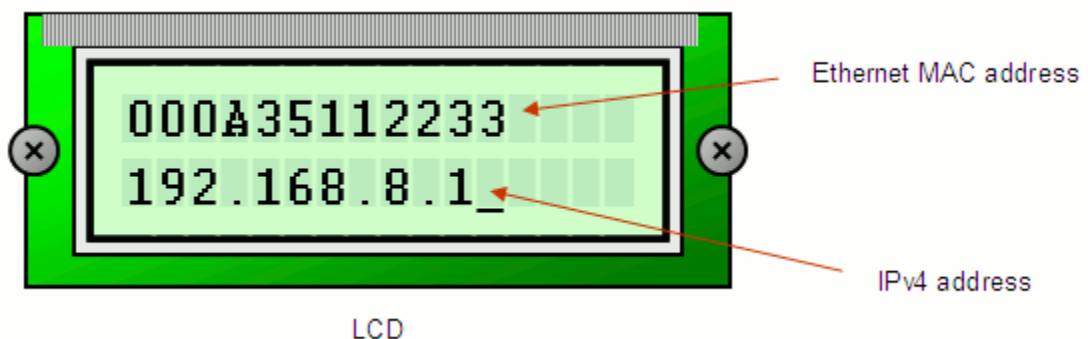
Configuration Source Selector Switch: sys ACE (System ACE method)



Troubleshooting

Verify System ACE settings

1. Power on the development board. Check the on-board status LEDs to ensure the FPGA is configured with the initial bootloader image from the CompactFlash card. If the configuration succeeded, the DONE LED should on and all error LEDs should be off. Otherwise, please reexamine the steps in the System ACE setup procedure.
2. Check the information displayed on the 16-character x 2-line LCD screen of the board. If no error occurred, the Ethernet MAC address (without colons) should appear on the first line of the LCD and the IPv4 address should appear on the second line.



3. If the LCD does not display the information correctly, press the System ACE Reset button to reset the System ACE controller and reconfigure the FPGA. Check the status LEDs again to ensure device configuration completed successfully.

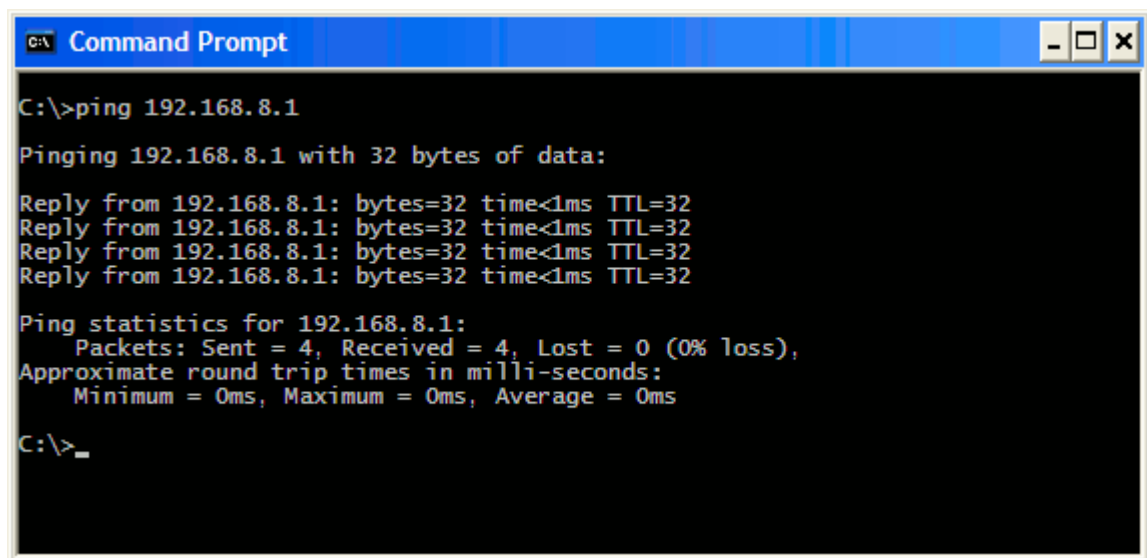
Verify Ethernet interface and connection status

1. Connect the Ethernet interface of the board to a network connection, or directly to a host.
2. Check the on-board Ethernet status LEDs to make sure the Ethernet interface is attached to an active Ethernet segment. The LEDs should reflect the link speed and the duplex mode at which the interface is operating. If no LED is on, press the CPU Reset button to reset the FPGA, and also examine whether the Ethernet segment is active.



Ethernet status LEDs

3. To ensure the board is reachable by the host, issue ICMP ping from the host to check the connectivity. For example, type "ping 192.168.8.1" on a console to test the connectivity to a board with IP address 192.168.8.1.



```
C:\> Command Prompt

C:\>ping 192.168.8.1

Pinging 192.168.8.1 with 32 bytes of data:

Reply from 192.168.8.1: bytes=32 time<1ms TTL=32
Reply from 192.168.8.1: bytes=32 time<1ms TTL=32
Reply from 192.168.8.1: bytes=32 time<1ms TTL=32
Reply from 192.168.8.1: bytes=32 time<1ms TTL=32

Ping statistics for 192.168.8.1:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 0ms, Maximum = 0ms, Average = 0ms

C:\>_
```

4. The target FPGA listens on the UDP port 9999. Please ensure the underlying network does not block the associated traffic when network-based Ethernet configuration is used. This does not affect point-to-point Ethernet configuration.

Known Issues

- If you encounter problems transmitting data over a point-to-point Ethernet connection or experience instability issues, please disable the Hyper-Threading option in the BIOS on an Intel platform.
- IP fragmentation is not supported by the network-based Ethernet configuration. Please ensure the connection established between the host and the target FPGA platform can handle a maximum

transmission unit (MTU) size of at least 1300 bytes without fragmentation.

Shared Memory Support

Compiling Shared Memories for Hardware Co-Simulation

System Generator allows shared memory and shared memory derivative (e.g., shared FIFO and shared register) blocks to be compiled for hardware co-simulation. Designs that include shared memories are compiled for hardware co-simulation the same way traditional System Generator designs are compiled – by selecting a compilation target and pressing the **Generate** button on the System Generator dialog box. A design containing shared memory blocks may be compiled for hardware co-simulation provided the requirements described in the Restrictions on Shared Memories section are satisfied.

When a shared memory is compiled for hardware co-simulation, it is implemented in hardware either by a core or HDL component. The table below shows how shared memory blocks are mapped to hardware implementations.

To Block	From Block	Hardware Implementation
Shared Memory	Shared Memory	Dual Port Block Memory 6.1
To FIFO	From FIFO	Fifo Generator 2.1
To Register	From Register	synth_reg_w_init.(vhd,v)

There are two ways in which shared memories are compiled for hardware co-simulation. The type of compilation depends on whether the shared memory name is unique in the design, or if the shared memory has a partner who shares the same name. The following sections describe the two types of compilation behavior.

Single Shared Memory Compilation

A shared memory block is considered “single” if it has a unique name within a design. When a single shared memory is compiled for hardware co-simulation, System Generator builds the other half of the memory and automatically wires it into the resulting netlist. Additional interfacing logic is attached to the memory that allows it to communicate with the PC. When you co-simulate the shared memory, one half of the memory is used by the logic in your System Generator design. The other half communicates with the PC interfacing logic (Figure 1). In this manner, it is possible to communicate with the shared memory embedded inside the FPGA while a simulation is running.

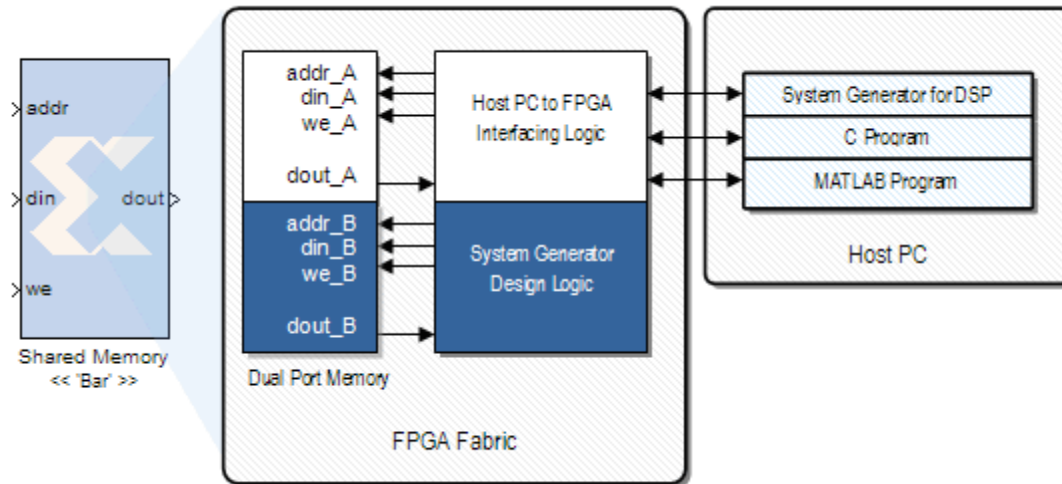


Figure 1: Shared memory FPGA hardware implementation

The shared memory hardware and interface logic are completely encapsulated by the hardware co-simulation block that is generated for the design. By co-simulating a hardware co-simulation block that contains a shared memory, it is possible for the user design logic and host PC software to share a common address space on the FPGA.

Note: The name of the hardware shared memory is the same as the shared memory name used by the original shared memory block. For example, if a shared memory block used “my_memory,” the hardware implementation of the block can be accessed using “my_memory” name.

All shared memories embedded inside the FPGA are automatically created and initialized before the start of a simulation by their respective co-simulation blocks. This means that any other shared memory objects that wish to access the hardware shared memory must specify **Ownership** and **initialization** parameter as **Owned** and **initialized elsewhere**. Doing so causes the software-based shared memories to attach automatically to the shared memories that reside inside the FPGA.

Compiling Shared Memory Pairs

It is also possible to compile a shared memory pair (i.e., two shared memories that specify the same name) for hardware co-simulation. In this case, the two shared memory halves are merged into a single hardware implementation during compilation. Unlike single shared memories, both sides of a shared memory pair connect to System Generator user design logic. For example, Figure 2 shows the hardware implementation for a To / From FIFO shared memory pair.

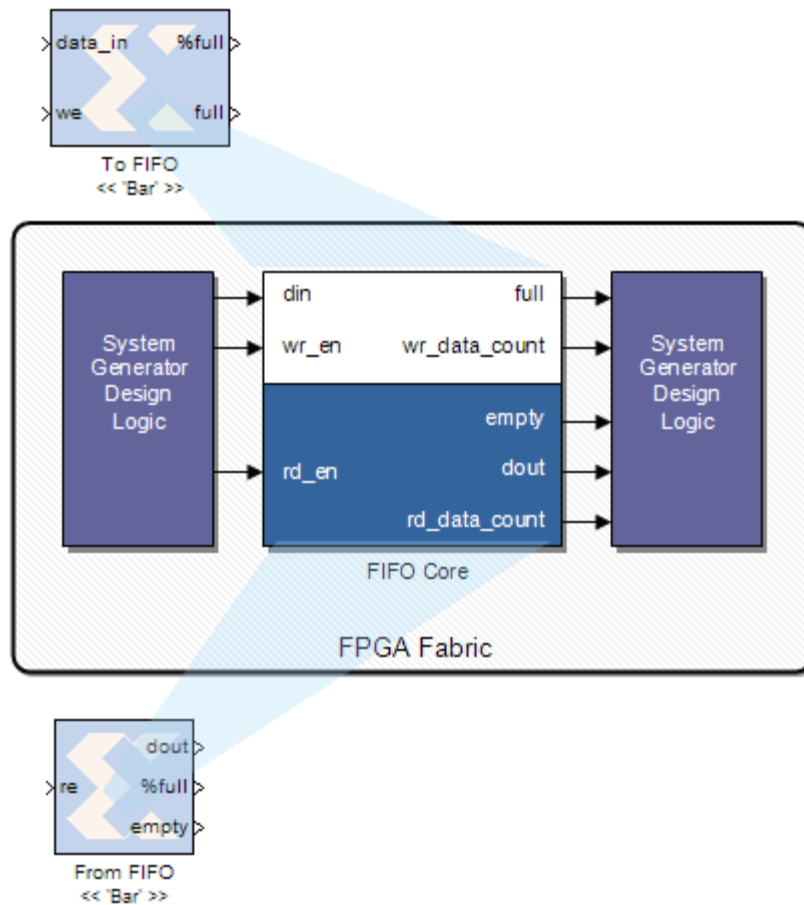


Figure 2: Shared FIFO pair FPGA hardware implementation

Note that because both sides of the shared memory connect to user design logic, it is not possible to communicate with these shared memories directly from the host PC.

Viewing Shared Memory Information

Hardware co-simulation blocks allow you to view information about the shared memories that were compiled as part of the design. A hardware co-simulation block that contains shared memories will have an enabled **Shared Memories** tab in the block configuration dialog box (Figure 2). Clicking on this tab exposes a table of information about each shared memory in the design.

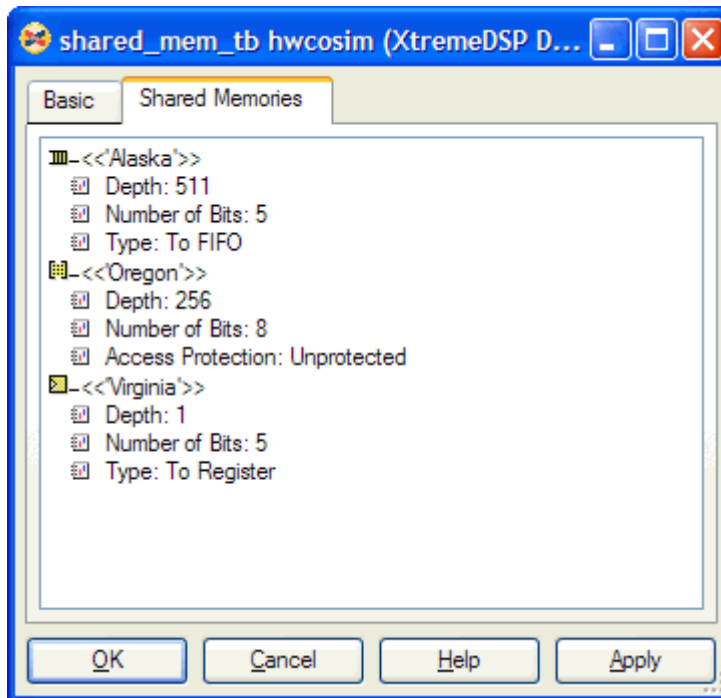


Figure 3: Example Shared Memories tab

The shared memory information table describes the type, bit width, and depth of each shared memory in the design. For Shared Memory blocks, it also specifies the Access Protection mode. Clicking on the [+] or [-] symbol next to the shared memory icon expands or collapses the shared memory table, respectively.

The icons associated with each shared memory type are shown in the table below.

Memory Type	Icon
Shared Memory	
Shared FIFO	
Shared Register	

Co-simulating Unprotected Shared Memories

Unprotected shared memory blocks may be written to or read from at any time – this type of memory has no notion of mutually exclusive access. Data transfers to and from an unprotected hardware shared memory occur a single-word at a time, unlike the high-speed data transfer mode used by lockable shared memories. To ensure data coherency between software and hardware, a single image of the shared memory data is shared between hardware and software. This image is stored in the FPGA using dual port memory. System Generator allows both hardware design logic and other software-based shared memory objects on the host PC to access the shared memory data concurrently. When software shared memory objects read or write data to the shared memory, a proxy seamlessly handles communication with the hardware memory resource.

Figure 1 shows an example of unprotected shared memory implemented in the FPGA that is communicating with three shared memory objects running on the host PC. In this example, the software shared memory objects access the hardware shared memory by specifying the same shared memory name, `my_mem`. From the perspective of the software shared memories, the implementation of the shared memory resource is irrelevant; the hardware shared memory is treated as any another shared memory object. Read and writes to the shared memory are handled by the shared memory API.

Note: Not all shared memory objects need to be created or executed in the Simulink environment. A C++ application (Figure 1) is just one example of an external application that may communicate with the hardware shared memory data using the shared memory API.

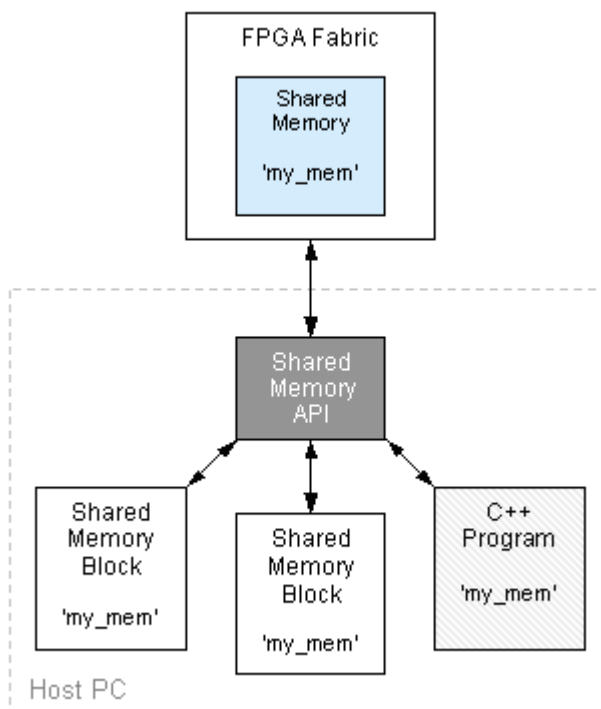


Figure 1: Shared memory in unprotected mode used by three software processes and FPGA hardware

Co-simulating Lockable Shared Memories

In lockable access mode, the System Generator co-simulation hardware must acquire lock over the shared memory object before it may access its contents. When the hardware acquires (releases) lock of the shared memory, the memory contents are transferred to (from) the FPGA using a high-speed data transfer. Using this methodology, it is possible to implement System Generator hardware co-simulation designs with high memory bandwidth requirements. For more information on how to do this, refer to the tutorial entitled Real-time Signal Processing using Hardware Co-simulation.

Unlike unprotected shared memories, two images of the shared memory data are used when a lockable shared memory is co-simulated. One memory image is stored using dual port memory in the FPGA. This image is accessed by the System Generator hardware co-simulation design and co-simulation interfacing logic. The other image is implemented as a shared memory object on the host PC. This software shared memory image is accessed by any software shared memory objects used in a design.

In lockable mode, a software process or hardware circuit that wishes to access the shared memory must first obtain the lock. If the hardware has lock of the memory, no software objects may access the memory contents. Likewise, if a software object controls the memory, the hardware cannot read or write to the memory. Note that lockable hardware shared memories include additional logic to handle the mutual exclusion. The interaction between hardware and software lockable shared memories is shown in Figure 1.

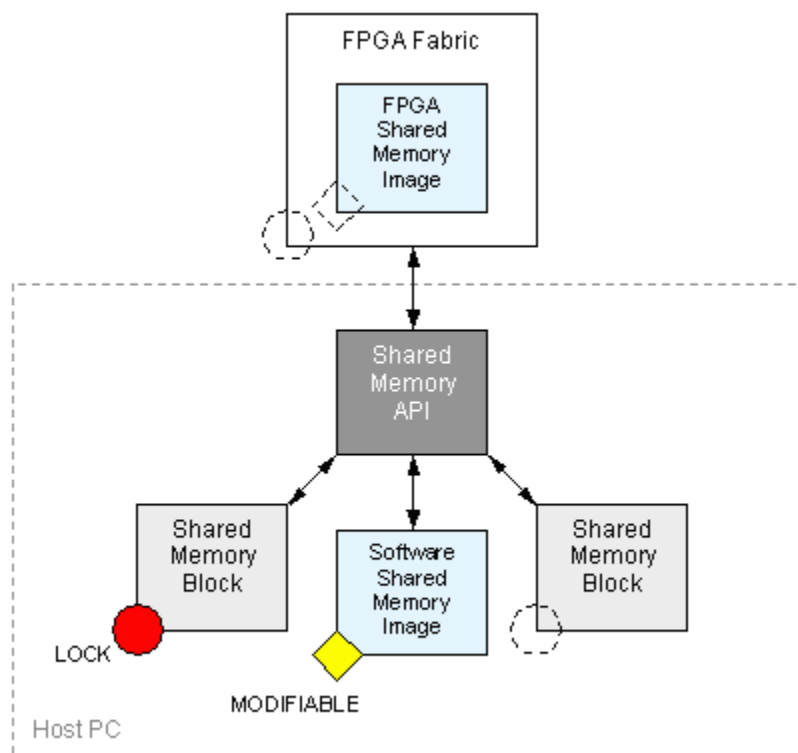


Figure 1: Lockable shared memory used by three software processes and FPGA hardware

The red circle in Figure 1 represents a lock token. This token may be passed to any shared memory object, regardless of whether it is implemented in hardware or software. The dashed circle represents lock placeholders and signifies that lock can be passed to the block it is associated with. The diamond in Figure 1 represents a modifiable token. This token illustrates that when hardware has lock of the

memory, the hardware shared memory image may be modified. Likewise, when a software shared memory object has lock, the software shared memory image may be modified.

Having two shared memory images requires synchronization between software and hardware to ensure the images are coherent. This synchronization is accomplished by transferring the memory image between software and hardware upon lock transfer. System Generator performs high speed data transfers between the host PC and FPGA. The semantics associated with these transactions are shown in Figure 2.

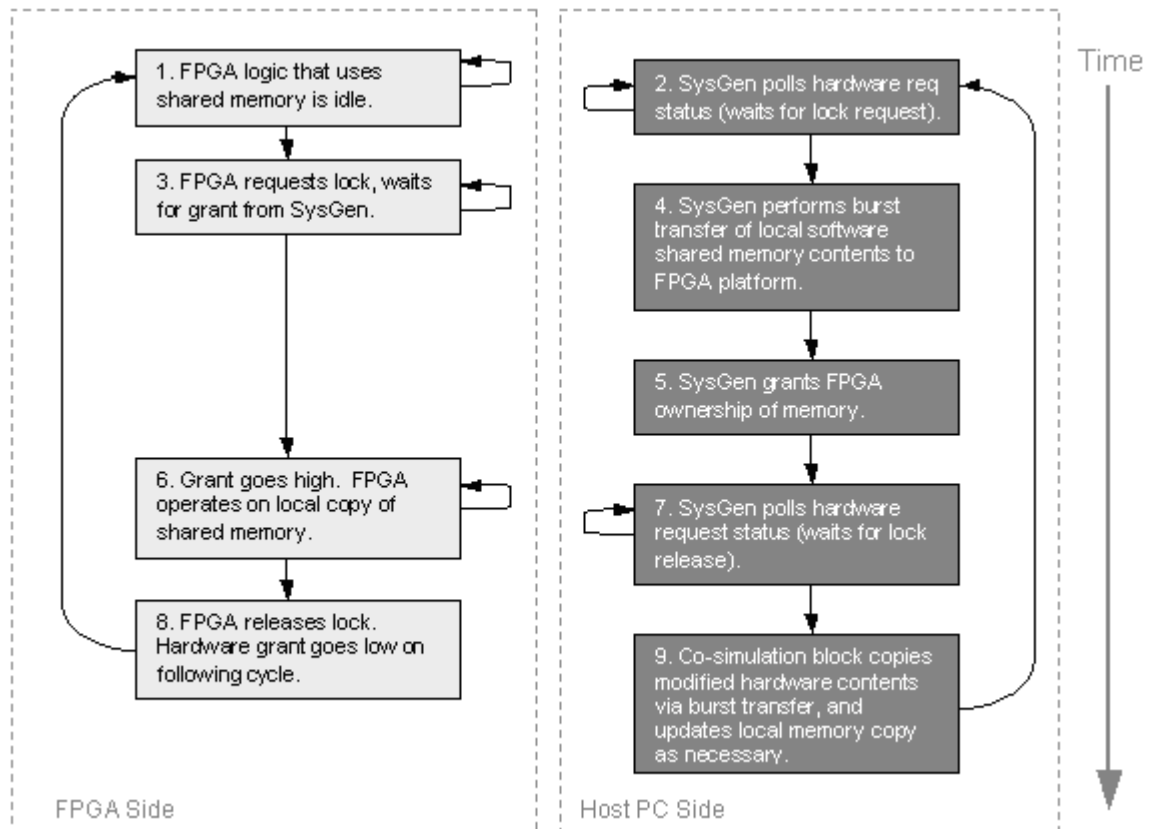


Figure 2: FPGA / System Generator operations on a lockable shared memory

Co-simulating Shared Registers

A To Register, From Register, or *shared register pair* may be generated and co-simulated in FPGA hardware. Here and throughout this section, a shared register pair is defined as a To Register block and From Register block that specify the same name (e.g., 'Bar'). In hardware, a shared register is implemented using a synthesizable register component (for VHDL) or a module (for Verilog). This section explains how single shared registers and shared register pairs behave during hardware co-simulation.

When a design that includes a shared register pair is compiled for hardware co-simulation, the pair is replaced by a single register instance. Both sides of the register attach to *user design logic*; that is, logic that originated from the original System Generator model. Unlike designs compiled using the Multiple Subsystem Generator block, all ports on the hardware register attach to signals in the same clock domain. In this case, control of the register is not shared between the PC and FPGA hardware since all register ports are attached to user design logic. Compiling a shared register pair into hardware is equivalent to compiling a System Generator Register or Delay block.

Compiling a single To Register or From Register block for hardware co-simulation results in a different type of implementation. A single register is still created to replace the To or From Register block. Only in this case, the register connects to both the PC interface and FPGA logic. The side of the register in the original model remains connected to user design logic. The other side of the register attaches to data and control ports that interface with the PC.

For example, when a From Register block is compiled for hardware co-simulation (Figure 1), the `dout` register port remains attached to the user design. The `din`, `ce`, and `clk` register ports attach to control and data ports that interface with the PC. In this way, it is possible for the PC to write to the register using System Generator's hardware co-simulation interfaces. When a To Register block is compiled for hardware co-simulation (Figure 2), the input ports are wired to user logic while the output port is wired to PC interface logic. You may access a shared register during hardware co-simulation using the other half of the shared register (i.e., using a To or From Register block), a C program or executable, or a MATLAB program.

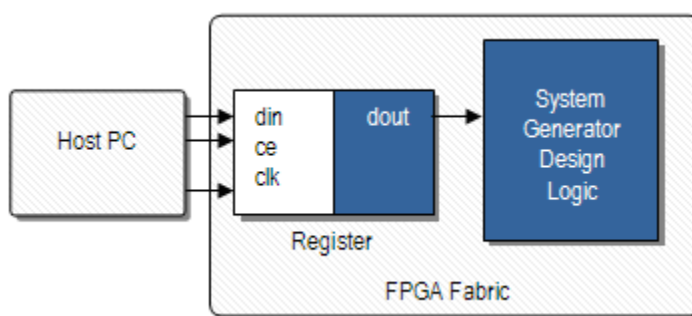


Figure 1: From Register block compiled for hardware co-simulation.

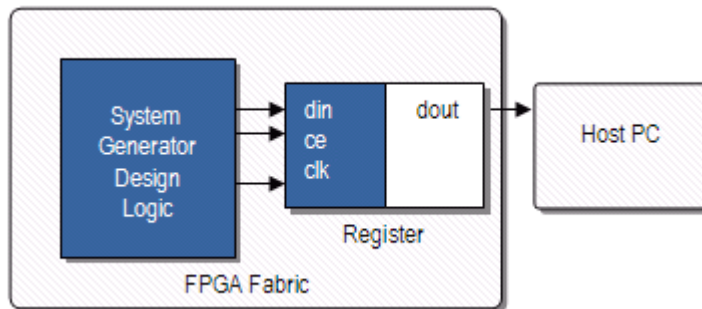


Figure 2: To Register block compiled for hardware co-simulation.

For designs that use hardware co-simulation, shared register pairs are typically distributed between software and FPGA hardware. In other words, one half of the pair is implemented in the FPGA while the other half is simulated in software using a To or From Register block. When data is written to a software To Register block, the hardware register is updated to with the same data. Similarly, when data is written into the hardware register, the same data is read by the From Register software block. A software shared register may connect to a hardware shared register simply by specifying the name of the shared register as it was compiled for hardware co-simulation.

*Note: You may find the names of all shared memories embedded inside an FPGA co-simulation design by viewing the **Shared Memories** tab on a hardware co-simulation block.*

When a software / hardware shared memory pair is co-simulated, System Generator transparently manages the interaction between the PC and FPGA hardware. This means that a shared register pair simulated in software should behave the same as a shared register pair distributed between the PC and FPGA hardware.

Co-simulating Shared FIFOs

A To FIFO, From FIFO, or *shared FIFO pair* may be generated and co-simulated in hardware. Here and throughout this section, a shared FIFO pair is defined as a To FIFO block and From FIFO block that specify the same name (e.g., 'Bar'). In hardware, a shared FIFO is implemented using the FIFO Generator core. The core is configured to use independent (asynchronous) clocks, and block memory for data storage. This section explains why co-simulating shared FIFOs is useful, and also how these blocks behave in hardware.

Asynchronous FIFOs are typically used in multi-clock applications to safely cross clock domain boundaries. When a free-running clock mode is used for hardware co-simulation, the FPGA operates asynchronously relative to the Simulink simulation. That is, the FPGA is not kept in lockstep with the simulation. Using the free-running clock mode effectively establishes two clock domains: the Simulink simulation clock domain and the FPGA free-running clock domain. In these designs, Shared FIFOs provide a reliable and safe way to transfer data between the host PC and FPGA platform.

Shared FIFOs may also be used to support burst transfers during co-simulation. It is possible to create vectors or frames of data, and transfer the data to the FPGA in a single transaction with the hardware. These interfaces can be used to further accelerate simulation speeds beyond what is typically possible with hardware co-simulation. For more information on how this is accomplished, refer to the User Guide section entitled Frame-based Acceleration using Hardware Co-simulation.

When a shared FIFO pair is generated for co-simulation, a single asynchronous FIFO core replaces the two software shared FIFO blocks. The read / write FIFO sides are attached to *user design logic* (i.e., logic derived from the original System Generator model) that attached to the From FIFO and To FIFO blocks, respectively (Figure 1). Because both FIFO sides attach to user logic in hardware, the PC does not share control of the FIFO with the design. Instead, the FIFO behavior is similar to a System Generator design that includes a traditional FIFO block.

Note that even though the FIFO exposes independent clock ports, the same co-simulation clock drives both ports when a FIFO pair is compiled. This is different from compiling a shared FIFO pair using the Multiple Subsystem Generator block, where the clocks are from distinct clock domains.

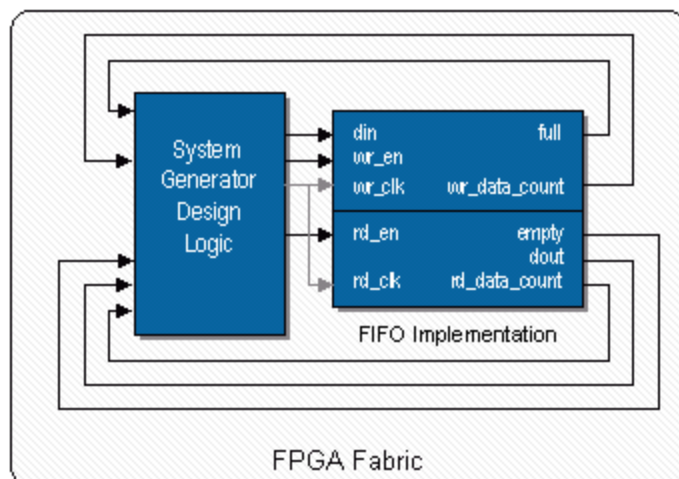


Figure 1: Shared FIFO pair compiled for hardware co-simulation.

Single shared FIFO blocks are treated differently than shared FIFO pairs. A single To FIFO or From FIFO block is replaced by an asynchronous FIFO core when it is compiled for hardware co-simulation. One side of the FIFO (i.e., the unused shared FIFO half in System Generator) is connected to PC

interface logic. The other side is connected to user design logic that attached to the original To or From FIFO block. In this manner, control over the FIFO is distributed between the PC and FPGA design.

When a To FIFO block is compiled for hardware co-simulation, the write side of the FIFO is connected to the same logic that attached to To FIFO block in user design (Figure 2). The read side of the FIFO is connected to PC interface logic that allows the PC to read data from the FIFO during simulation.

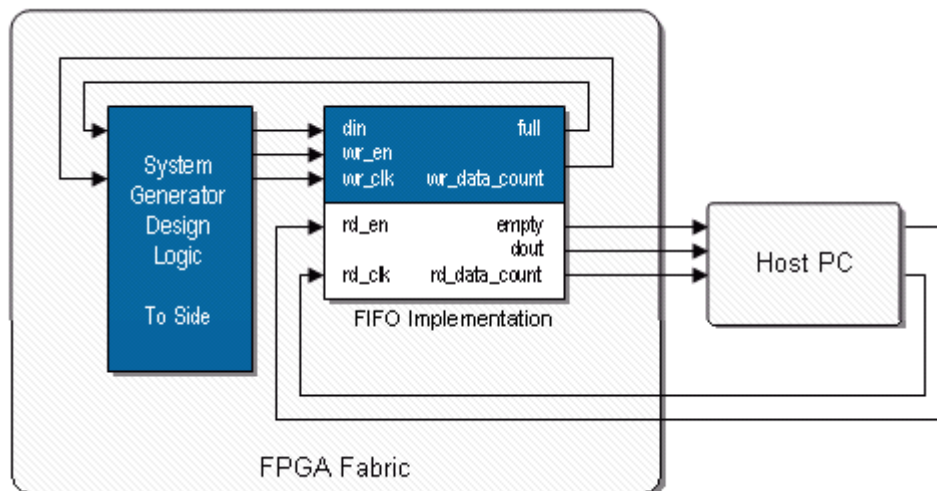


Figure 2: To FIFO block compiled for hardware co-simulation.

The opposite wiring approach is used when a From FIFO block is compiled for hardware co-simulation (Figure 3). In this case, the write side of the FIFO is connected to PC interface logic, while the read side is connected to the user design logic. The host PC writes data into the FIFO and the design logic may read data from the FIFO.

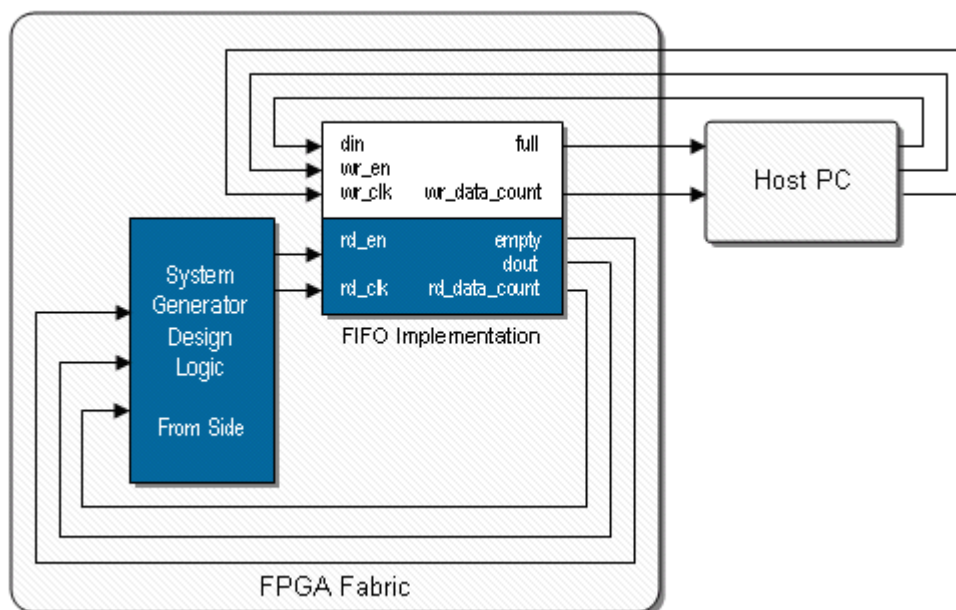


Figure 3: From FIFO block compiled for hardware co-simulation.

For designs that use hardware co-simulation, shared FIFO pairs are typically distributed between software and FPGA hardware. In other words, one half of the pair is implemented in the FPGA while the other half is simulated in software using a To or From FIFO block. Together, the software and hardware portions form a fully functional asynchronous FIFO. When a software / hardware shared FIFO pair is co-simulated, System Generator transparently manages the necessary transactions between the PC and FPGA hardware.

When data is written to a software To FIFO block during simulation, the same data is written to the FIFO in hardware. The design in hardware may then retrieve this data by reading from the FIFO. Similarly, when data is written into the hardware FIFO by design logic, the data may be read by the From FIFO software block. Note that the empty, full, read and write count ports on the shared FIFO blocks pessimistically reflect the state of the hardware FIFO counterpart. A software shared FIFO may connect to a hardware shared FIFO simply by specifying the name of the shared FIFO as it was compiled for hardware co-simulation.

*Note: You may find the names of all shared memories embedded inside an FPGA co-simulation design by viewing the **Shared Memories** tab on a hardware co-simulation block.*

Restrictions on Shared Memories

The following restrictions apply to System Generator designs that use shared memory, register, or FIFO blocks in conjunction with hardware co-simulation:

- The access protection mode of a shared memory may not be modified once it has been compiled for hardware co-simulation.
- Shared memory address port widths are limited to 24-bits (or less), allowing an address space of 16,777,216 words;
- Shared memory, register, and FIFO data port widths are currently limited to 32-bits or less.
- Shared memories and FIFOs are implemented in hardware using block memories; neither distributed nor external memory implementations are currently supported.
- No more than two shared memories with the same shared memory name may be compiled for hardware co-simulation.
- Two or more hardware co-simulation blocks that have shared memory names in common may not concurrently be used in the same design.

Specifying Xilinx Tool Flow Settings

When a design is compiled for System Generator hardware co-simulation, the command line tool, XFLOW, is used to implement and configure your design for the selected FPGA platform. XFLOW defines various flows that determine the sequence of programs that should be run on your design during compilation. There are typically multiple flows that must be run in order to achieve the desired output results, which in the case of hardware co-simulation targets, is a configuration bitstream.

System Generator uses two flows, *implementation* and *configuration*, in order to produce a configuration bitstream. The implementation flow is responsible for compiling the synthesis tool netlist output (e.g., EDIF or NGC) into a placed and routed NCD file. To accomplish this, it runs the Xilinx tools NGDBuild, MAP, and PAR. The implementation flow can also execute TRACE (for timing analysis purposes), although this program is typically omitted in order to expedite the compilation process. The configuration flow runs the tools necessary to create an FPGA bitstream, using the fully elaborated NCD file as input.

The implementation and configuration flow types have separate XFLOW options files associated with them. An XFLOW options file declares the programs that should be run for a particular flow, and defines the command line options that are used by these tools. Each hardware co-simulation compilation target provides options files that define the default configuration options for these tools. Sometimes you may want to use options files that use settings that differ (e.g., to specify a higher placer effort level in PAR) from the default options provided by the target. In this case, you may create your own options files, or edit the default options files to include your desired settings.

The Hardware Co-Simulation Settings GUI shown in Figure 1 allows you to specify options files other than the default options files provided by the compilation target.

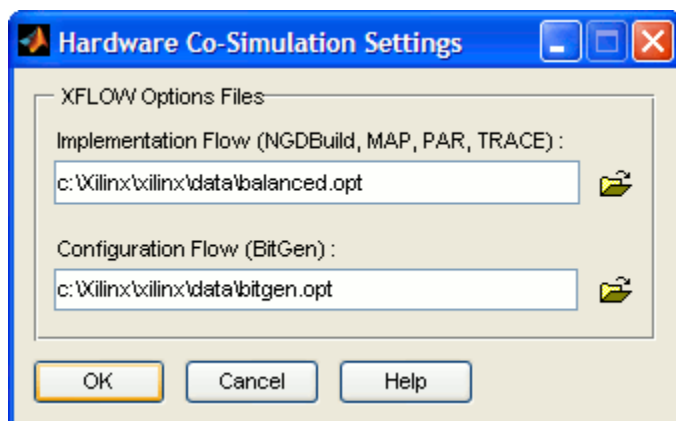


Figure 1: Hardware Co-Simulation Settings GUI

Parameters available on the Hardware Co-Simulation Settings GUI are:

- Implementation: Specifies the options file that is used by the implement flow type. By default, System Generator will use the implement options file that is specified by the compilation target.
- Configuration: Specifies the options file that is used by the config flow type. By default, System Generator will use the config options file that is specified by the compilation target.

The Xilinx ISE software includes several example XFLOW options files. From the base directory of your Xilinx ISE software tree (e.g., c:\Xilinx\), these files can be located under xilinx\data directory (e.g., c:\Xilinx\xilinx\data). Three commonly used implementation options files include:

- `balanced.opt`;
- `fast_runtime.opt`;
- `high_effort.opt`.

Note: It is possible to define options files that may cause errors in the System Generator hardware co-simulation flow. As a result, it is typically a good idea to make backup copies of the default options files before modifying them. In addition, the configuration options file should be edited with caution, as most FPGA hardware platforms have specific configuration parameter requirements.

Importing HDL Modules into System Generator

The Black Box Interface

The Configuration M-Function and Configuration Wizard

Note: System Generator provides several step-by-step examples that illustrate many of the topics described in the sections that follow. These examples can be found in the Black Box Examples section of the User Guide.

The Configuration M-Function

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

- Name of the top level entity for the module;
- VHDL or Verilog language selection;
- Port descriptions;
- Generics required by the module;
- Clocking and sample rates;
- Files associated with the module;
- Whether the module has any combinational paths.

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

The screenshot shows a dialog box with two tabs: 'Basic' and 'Implementation'. The 'Implementation' tab is active. It contains the following fields:

- Block configuration m-function:** A text field containing the text 'parity_block_config'.
- Simulation mode:** A dropdown menu with 'External co-simulator' selected.
- HDL co-simulator to use (specify helper block by name):** A text field containing the text 'ModelSim'.

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

```
function sample_block_config(this_block)
```

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

Language Selection

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

VHDL Module:

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

Verilog Module:

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

Note: The configuration wizard automatically selects the appropriate language when it generates a configuration m-function.

Specifying the Top level Entity

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

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

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

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

Note: The configuration wizard automatically sets the name of the top level entity when it generates a configuration m-function.

Defining Block Ports

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

Adding New Ports

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

Adding an input port:

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

Adding an output port:

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

The string parameter passed to methods `addSimulinkInputport` and `addSimulinkOutputport` 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.

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

Note: The configuration wizard automatically sets the port names when it generates a configuration m-function.

Obtaining a Port Object

Once a port has been added to a block descriptor, it is often necessary to configure individual attributes on the port. Before configuring the port, you must obtain a descriptor for the port you would like to configure. `SysgenBlockDescriptor` provides methods for accessing the port objects that are associated with it. For example, the following method retrieves the port named `din` on the `this_block` descriptor:

Accessing a `SysgenPortDescriptor` object:

```
din = this_block.port('din');
```

In the above code, an object `din` is created and assigned to the descriptor returned by the `port` function call.

`SysgenBlockDescriptor` also provides methods, `inport` and `outport`, that return a port object given a port index. A port index is the index of the port (in the order shown on the block interface) and is some value between 1 and the number of input/output ports on the block. These methods are useful when you need to iterate through the block's ports (e.g., for error checking).

Configuring Port Types

`SysgenPortDescriptor` provides methods for configuring individual ports. For example, assume port `dout` is unsigned, 12 bits, with binary point at position 8. The code below shows one way in which this type can be defined.

```
dout = this_block.port('dout');  
dout.setWidth(12);  
dout.setBinPt(8);  
dout.makeUnsigned();
```

The following also works:

```
dout = this_block.port('dout');  
dout.setType('Ufix_12_8');
```

The first code segment sets the port attributes using individual method calls. The second code segment defines the signal type by specifying the signal type as a string. Both code segments are functionally equivalent.

The black box supports HDL modules with 1-bit ports that are declared using either single bit port

(e.g., `std_logic`) or vectors (e.g., `std_logic_vector(0 downto 0)`) notation. By default, System Generator assumes ports to be declared as vectors. You may change the default behavior using the `useHDLVector` method of the descriptor. Setting this method to `true` tells System Generator to interpret the port as a vector. A `false` value tells System Generator to interpret the port as single bit.

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

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

Note: The configuration wizard automatically sets the port types when it generates a configuration m-function.

Configuring Port Sample Rates

The black box block supports ports that have different sample rates. By default, the sample rate of an output port is the sample rate inherited from the input port (or ports, if the inputs run at the same sample rate). Sometimes it is necessary to explicitly specify the sample rate of a port (e.g., if the output port rate is different than the block's input sample rate).

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

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

Note: The rate parameter passed to the `setRate` method is not necessarily the Simulink sample rate of that the port runs at. Instead, it is a positive Integer value that defines the ratio between the desired port sample period and the Simulink system clock period defined by the System Generator 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, the example `dout` port is assigned a rate of 3 by invoking the `setRate` method as follows:

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

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

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

Dynamic Output Ports

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

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

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

Note: A black box's configuration m-function is invoked at several different times when a model is compiled. The configuration function may be invoked before the data types and rates have been propagated to the black box.

The SysgenBlockDescriptor object provides Boolean member variables `inputTypesKnown` and `inputRatesKnown` that tell whether the port types and rates have been propagated to the block. If you are setting dynamic output port types or rates based on input port configurations, the configuration calls should be nested inside conditional statements that check that values of `inputTypesKnown` and `inputRatesKnown`.

The following code shows how to set the width of a dynamic output port `dout` to have the same width as input port `din`:

```
if (this_block.inputTypesKnown)
    dout.setWidth(this_block.port('din').width);
end
```

Setting dynamic rates works in a similar manner. The code below sets the sample rate of output port `dout` to be twice as slow as the sample rate of input port `din`:

```
if (this_block.inputRatesKnown)
    dout.setRate(this_block.port('din').rate*2);
end
```

Black Box Clocking

In order to import a multirate module, you must tell System Generator information about the module's clocking in the configuration m-function. System Generator treats clock and clock enables differently than other types of ports. A clock port on an imported module must always be accompanied by a clock enable port (and vice versa). In other words, clock and clock enables must be defined as a pair, and exist as a pair in the imported module. This is true for both single rate and multirate designs.

Note: Although clock and clock enables must exist as pairs, System Generator drives all clock ports on your imported module with the FPGA system clock. The clock enable ports are driven by clock enable signals derived from the FPGA system clock.

SysgenBlockDescriptor provides a method, `addClkCEPair`, which allows you to define clock and clock enable information for a black box. This method accepts three parameters. The first parameter defines the name of the clock port (as it appears in the module). The second parameter defines the name of the clock enable port (also as it appears in the module).

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

- The clock port must contain the substring "clk"
- The clock enable must contain the substring "ce"
- The strings containing the substrings "clk" and "ce" must be the same (e.g., "my_clk_1" and "my_ce_1").

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

For example, assume you have a clock enable port named "ce_3" that would like to have a period three

times larger than the system clock period. The following function call establishes this clock enable port:

```
addClkCEPair('clk_3','ce_3',3);
```

When System Generator compiles a black box into hardware, it produces the appropriate clock enable signals for your module, and automatically wires them up to the appropriate clock enable ports.

Combinational Paths

If the module you are importing has at least one combinational path (i.e., a change on any input can effect an output port without a clock event), you must indicate this in the configuration m-function. SysgenBlockDescriptor object provides a `tagAsCombinational` method that indicates your module has a combinational path. It should be invoked as follows in the configuration m-function:

```
this_block.tagAsCombinational;
```

Specifying VHDL Generics and Verilog Parameters

You may specify a list of generics that get passed to the module when System Generator compiles the model into HDL. Values assigned to these generics can be extracted from mask parameters and from propagated port information (e.g., port width, type, and rate). This flexible means of generic assignment allows you to support highly parametric modules that are customized based on the Simulink environment surrounding the black box.

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

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

It is also possible to set generic values based on port or propagated input port information (e.g., a generic specifying the width of a dynamic output port).

Because a black box's configuration m-function is invoked at several different times when a model is compiled, the configuration function may be invoked before the data types (or rates) have been propagated to the black box. If you are setting generic values based on input port types or rates, the `addGeneric` calls should be nested inside a conditional statement that checks the value of the `inputTypesKnown` or `inputRatesKnown` variables. For example, the width of the `dout` port can be set based on the value of `din` as follows:

```
if (this_block.inputTypesKnown)
    % set generics that depend on input port types
    this_block.addGeneric('dout_width', ...
        this_block.port('din').width);
end
```

Generic values can be configured based on mask parameters associated with a block box. SysgenBlockDescriptor provides a member variable, `blockName`, which is a string representation of the black box's name in Simulink. You may use this variable to gain access the black box associated with the particular configuration m-function. For example, assume a black box defines a parameter named `init_value`. A generic with name `init_value` can be set as follows:

```
simulink_block = this_block.blockName;
init_value = get_param(simulink_block,'init_value');
this_block.addGeneric('init_value','String',init_value);
```

Note: You can add your own parameters (e.g., values that specify generic values) to the black box by doing the following:

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

Error Checking

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

Black Box API

SysgenBlockDescriptor Member Variables

Type	Member	Description
String	entityName	Name of the entity or module.
String	blockName	Name of the black box block.
Integer	numSimulinkInports	Number of input ports on black box.
Integer	numSimulinkOutports	Number of output ports on the black box.
Boolean	inputTypesKnown	true if all input types are defined, and false otherwise.
Boolean	inputRatesKnown	true if all input rates are defined, and false otherwise.
Array of Doubles	inputRates	Array of sample periods for the input ports (indexed as in <code>inport(indx)</code>). Sample period values are expressed as integer multiples of the Simulink System Period value specified by the master System Generator 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

<code>setTopLevelLanguage(language)</code>	Declares language for the top level entity (or module) of the black box. language should be 'VHDL' or 'Verilog'.
<code>setEntityName(name)</code>	Sets name of the entity or module.
<code>addSimulinkInport(pname)</code>	Adds an input port to the black box. pname tells the name the port should have.
<code>addSimulinkOutport(pname)</code>	Adds an output port to the black box. pname tells the name the port should have.
<code>setSimulinkPorts(in,out)</code>	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.
<code>tagAsCombinational()</code>	Indicate that the block has a combinational path (i.e., direct feedthrough) from an input port to an output port.
<code>addClkCEPair(clkPname, cePname, rate)</code>	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.
<code>port(name)</code>	Returns the SysgenPortDescriptor that describes a given input port. indx tells the index of the port to look for, and should be between 1 and numInputPorts.
<code>inport(indx)</code>	Returns the SysgenPortDescriptor that describes a given input port. indx tells the index of the port to look for, and should be between 1 and numInputPorts.
<code>outport(indx)</code>	Returns the SysgenPortDescriptor that describes a given output port. indx tells the index of the port to look for, and should be between 1 and numOutputPorts.
<code>addGeneric(identifier, value)</code>	Defines a generic (or parameter if using Verilog) for the block. identifier is a string that tells the name of the generic. value can be a double or a string. The type of the generic is inferred from value's type. If value is an integral double, e.g., 4.0, the type of the generic is set to integer. For a non-integral double, the type is set to real. When value is a string containing only zeros and ones, e.g., '0101', the type is set to bit_vector. For any other string value the type is set to string.
<code>addGeneric(identifier, type, value)</code>	Explicitly specifies the name, type, and value for a generic (or parameter if using Verilog) for the block. All three arguments are strings. identifier tells the name, type tells the type, and value tells the value.

<code>addFile(fn)</code>	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 (respectively, Verilog) file names should end in <code>.vhd</code> (resp., <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.
<code>getDeviceFamilyName()</code>	Gets the name of the FPGA device corresponding to the Blackbox.
<code>setSimulatorCompilationScript(script)</code>	Overrides the default HDL co-simulation compilation script that the black box generates. <code>script</code> tells the name of the script to use. This method can, for example, be used to short-circuit the compilation phase for repeated simulations where the HDL for the black box remains unchanged.
<code>setError(message)</code>	Indicates that an error has occurred, and records the error message. <code>message</code> gives the error message.

SysgenPortDescriptor Member Variables

Type	Member	Description
String	<code>name</code>	Tells the name of the port.
Integer	<code>simulinkPortNumber</code>	Tells the index of this port in Simulink. Indexing starts with 1 (as in Simulink).
Boolean	<code>typeKnown</code>	True if this port's type is known, and false otherwise.
String	<code>type</code>	Type of the port, e.g., <code>UFix_<n>_</code> , <code>XFix_<n>_</code> , or <code>Bool</code>
Boolean	<code>isBool</code>	True if port type is <code>Bool</code> , and false otherwise.
Boolean	<code>isSigned</code>	True if type is signed, and false otherwise.
Boolean	<code>isConstant</code>	True if port is constant, and false otherwise.
Integer	<code>width</code>	Tells the port width.
Integer	<code>binpt</code>	Tells the binary point position, which must be an integer in the range <code>0..width</code> .
Boolean	<code>rateKnown</code>	True if the rate is known, and false otherwise.
Double	<code>rate</code>	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>_, Xfix_<n>_, signed or unsigned. The last two choices leave the width and binary point position unchanged.
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.
setRate(rate)	Assigns the rate for this port. rate must be a positive integer expressed as a MATLAB double or Inf for constants.
useHDLVector(s)	Tells whether a 1-bit port is represented as single-bit (ex: std_logic) or vector (ex: std_logic_vector(0 downto 0)).
HDLTypesVector()	Sets representation of the 1-bit port to std_logic_vector(0 downto 0).

The Configuration Wizard

The black box provides a configuration wizard that makes it easy to associate a VHDL or Verilog module to a black box. The configuration wizard parses the VHDL or Verilog module that you are trying to import, and automatically constructs a configuration m-function based on its findings. It then associates the configuration m-function it produces to the black box in your model. Whether or not you can use the configuration m-function as is depends on the complexity of the black box you are importing. Sometimes the configuration m-function must be customized by hand to specify details the configuration wizard misses.

Using the Configuration Wizard

The black box configuration wizard opens automatically when a new black box block is added to a model.

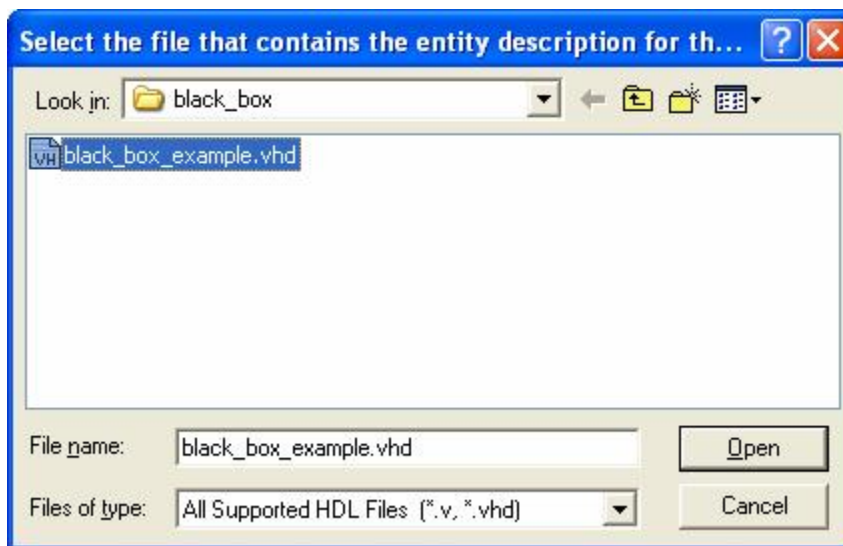
Note: Before running the configuration wizard, ensure the VHDL or Verilog you are importing meets the specified HDL requirements and restrictions.

For the configuration wizard to find your module, the model must be saved in the same directory as the module you are trying to import. This means, in particular, that the model must be saved to some directory.

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



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



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

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

Configuration Wizard Fine Points

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

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

- If your model has a combinational path, you must call the `tagAsCombinational` method of the block's `SysgenBlockDescriptor` object.
- The configuration wizard only knows about the top level entity that is being imported. There are typically other files that go along with this entity. These files must be added manually in the configuration m-function by invoking the `addFile` method for each additional file.
- The configuration wizard creates a single-rate black box. This means that every port on the black box runs at the same rate. In most cases, this is acceptable. You may want to explicitly set port rates, which can result in a faster simulation times.

HDL Requirements and Restrictions

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

- The entity name must not collide with any other entity name in the design.
- Bi-directional ports are not allowed on the top level black box entity.
- For Verilog black boxes, the module and port names must be lower case and must follow standard VHDL naming conventions.
- Any port that is a clock or clock enable must be of type `std_logic`. (For Verilog black boxes, ports must be of non-vector inputs, e.g., input `clk`.)
- Clock and clock enable ports in black box HDL should be expressed as follows: Clock and clock enables must appear as pairs (i.e., for every clock, there is a corresponding clock enable, and vice-versa). Although a black box may have more than one clock port, a single clock source is used to drive each clock port. Only the clock enable rates differ.
- Each clock name (respectively, clock enable name) must contain the substring `clk` (resp., `ce`).
- 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`.

HDL Co-Simulation

Configuring the HDL Simulator

Black box HDL can be co-simulated with Simulink using the System Generator interface to either ISE Simulator or the ModelSim simulation software from Model Technology, Inc.

ISE Simulator

To use ISE Simulator for co-simulating the HDL associated with the black box, select *ISE Simulator* as the option for *Simulation mode* parameter on the black box as shown in the figure below. The model is then ready to be simulated, and the HDL co-simulation takes place automatically.

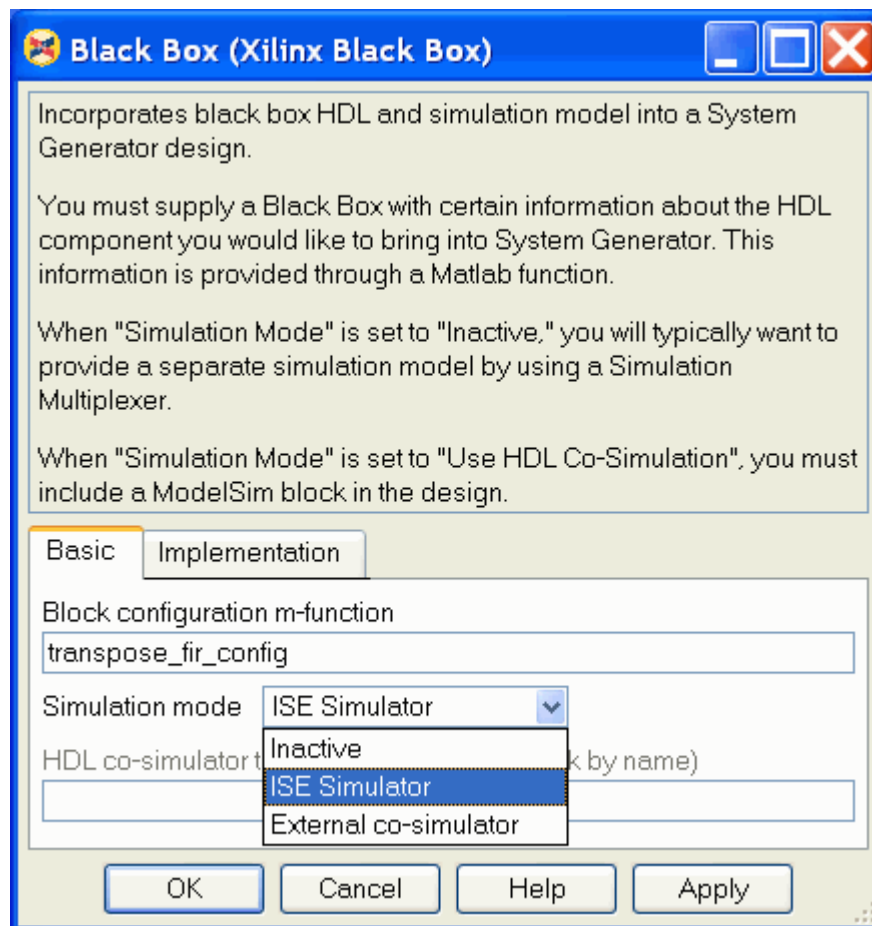


Figure 1 Configuring a black box to use ISE simulator for simulation

ModelSim Simulator

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



For each black box that you wish to have co-simulated using ModelSim simulator, you need to open its block parameterization dialog and set it to use the ModelSim session represented by the black box that was just added. You do this by making the following two settings:

1. Change the Simulation Mode field from *Inactive* to *External co-simulator*.
2. Enter the name of the ModelSim block (e.g., ModelSim) in the *HDL Co-Simulator to use* field.

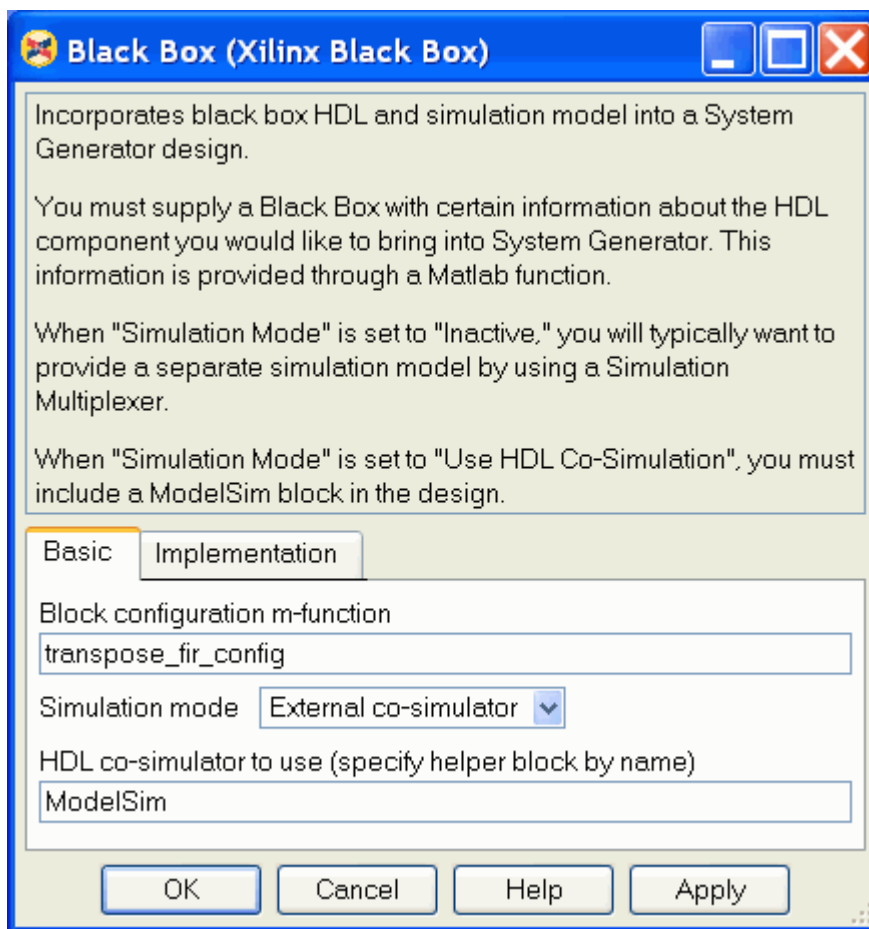


Figure 2 Configuring a black box to use ModelSim for simulation

The block parameter dialog for the ModelSim block includes some parameters that you can use to control various options for the ModelSim session. See the ModelSim block help pages for details. The model is then ready to be simulated with these options, and the HDL co-simulation takes place automatically.

Co-Simulating Multiple Black Boxes

System Generator allows many black boxes to share a common ModelSim co-simulation session. I.e., many black boxes can be set to "use" the same ModelSim block. In this case, System Generator automatically combines all black box HDL components into a single shared top level co-simulation component. This is transparent to the user. It does mean, however, that only one ModelSim simulation license is needed to co-simulate several black boxes in the Simulink simulation.

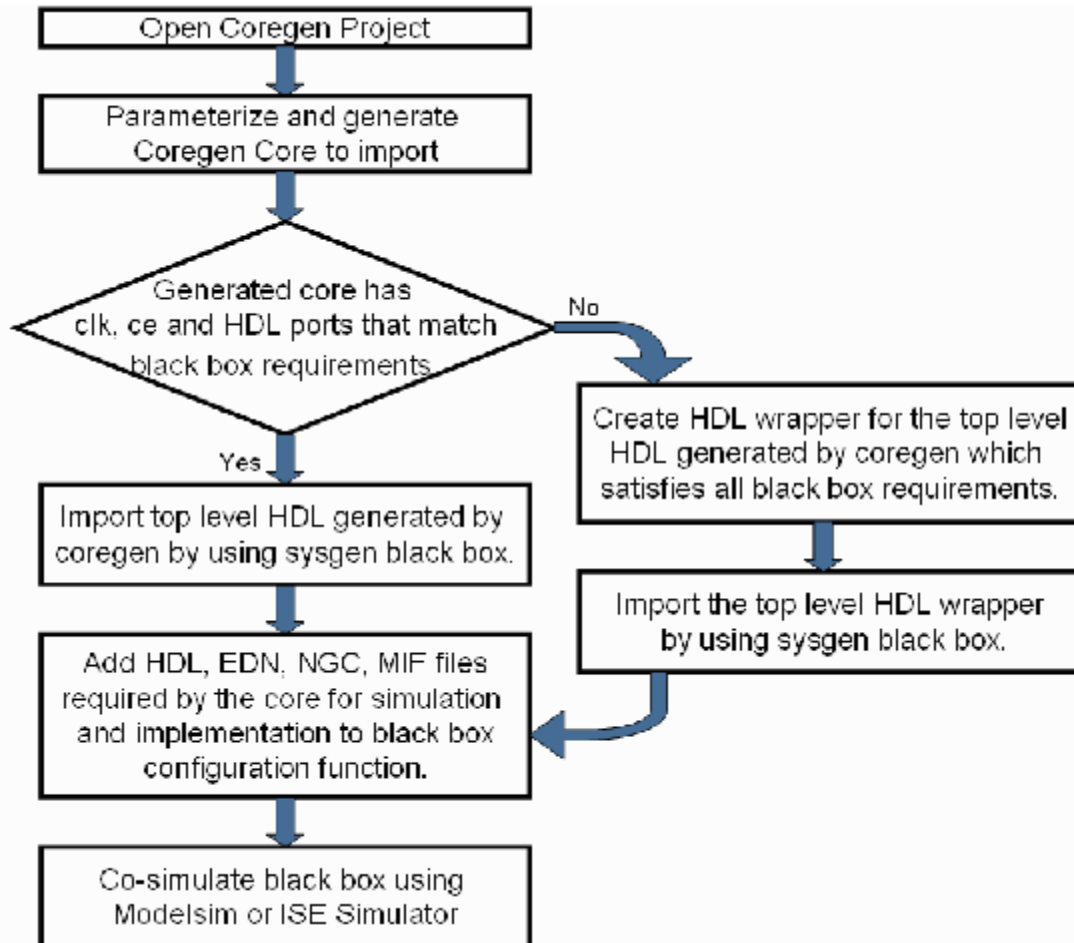
For an example of how to do this, see [Simulating Several Black Boxes Simultaneously](#).

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

Black Box Examples

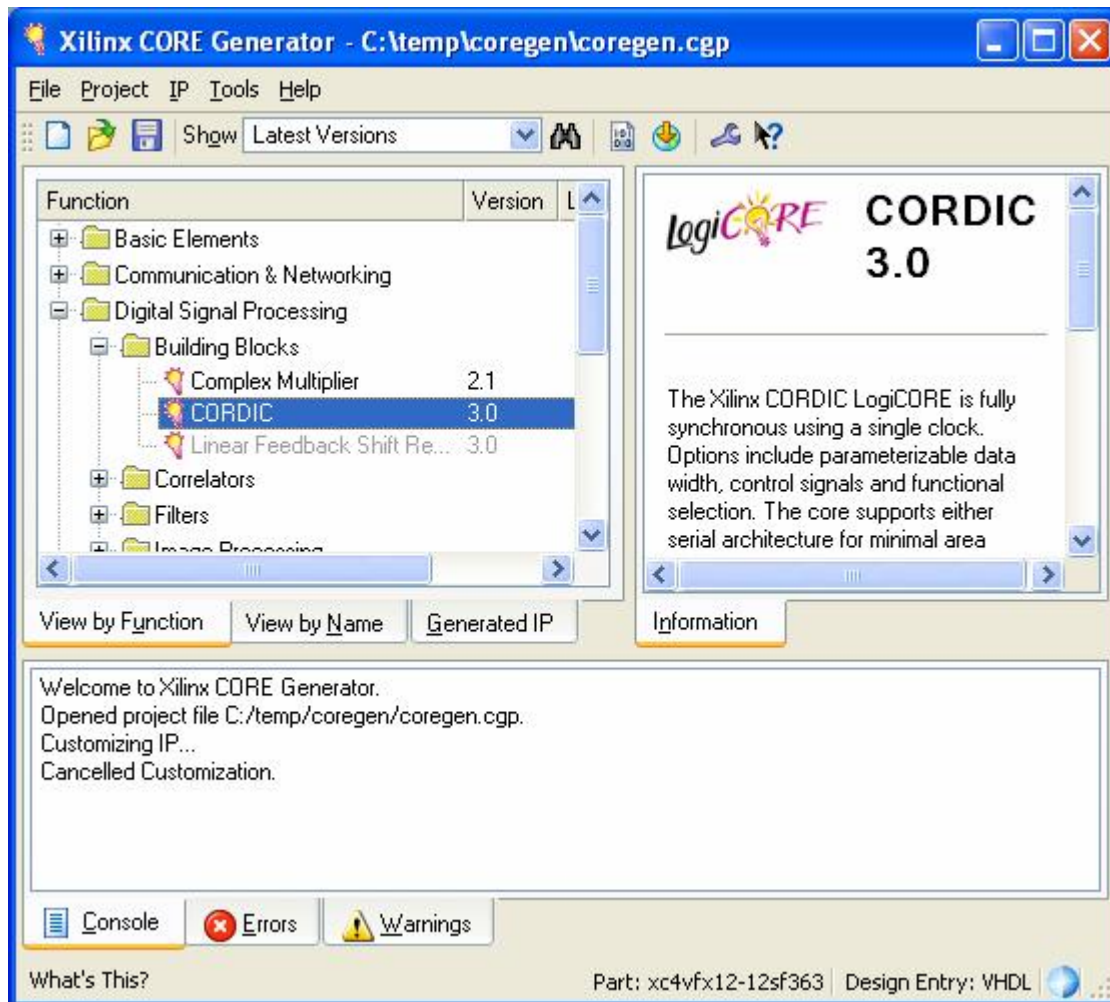
Importing a Xilinx COREGEN Module

This section describes two different ways of importing Xilinx CORE Generator modules, as black boxes, into System Generator. The first example shows how to import blocks which satisfy black box HDL requirements and restrictions. The second example shows to write a VHDL wrapper to import CORE generator modules as black boxes. The flow graph below illustrates the process of importing CORE generator modules.



Example 1. Importing a Coregen Core that satisfies black box HDL requirements

1. Start coregen and open the coregen project file
(./Xilinx/sysgen/examples/coregen_import/example1/coregen_import_example1.cgp)
2. Double click the CORDIC 3.0 icon to launch the customization GUI



3. Parameterize and generate the CORDIC 3.0 core with component name "cordic_sincos", functional Selection of "Sin and Cos" and the remaining options set to be the default values as shown below:

CORDIC

Parameters Core Overview Contact Web Links



CORDIC

Component Name

Functional Selection

☐ Rotate
☒ Sin and Cos
☐ Arc Tan
☐ Square Root

☐ Translate
☐ Sinh and Cosh
☐ Arc Tanh

Architectural Configuration

☐ Word Serial
☒ Parallel

Pipelining Mode

☐ No Pipelining
☐ Optimal
☒ Maximum

< Back

Next >

Page 1 of 4

Generate

Dismiss

Data Sheet...

Version Info...

☐ Display Core Footprint

CORDIC

Parameters Core Overview Contact Web Links



CORDIC

Data Format

☒ Signed Fraction
☐ Unsigned Fraction
☐ Unsigned Integer

Phase Format

☒ Radians
☐ Scaled Radians

Synchronization Enable

☐ Override
☒ No Override

Optional Pin Selection

☐ ACLR
☒ CE

☒ RDY
☐ SCLR
☐ ND

☒ X Out
☒ Y Out
☐ Phase Output

< Back

Next >

Page 2 of 4

Generate

Dismiss

Data Sheet...

Version Info...

☐ Display Core Footprint

CORDIC

Parameters

Core Overview

Contact

Web Links

LogiCORE

CORDIC

X_IN

Y_IN

PHASE_IN

ND

CE

CLK

X_OUT

Y_OUT

PHASE_OUT

RFD

RDY

ACLR

SCLR

Round Mode

Truncate

Round Pos Neg Inf

Round Pos Inf

Nearest Even

Input/Output Options

Register Inputs

Input Width

16

Valid Range 8..48

Register Outputs

Output Width

16

Valid Range 8..48

< Back

Next >

Page 3 of 4

Generate

Dismiss

Data Sheet...

Version Info...

Display Core Footprint

CORDIC

Parameters

Core Overview

Contact

Web Links

LogiCORE

CORDIC

X_IN

Y_IN

PHASE_IN

ND

CE

CLK

X_OUT

Y_OUT

PHASE_OUT

RFD

RDY

ACLR

SCLR

Advanced Configuration Parameter

Iterations

0

Valid Range 0..48

Precision

0

Valid Range 0 or 16..48

Coarse Rotation

Compensation Scaling

No Scale Compensation

CCM Scale Compensation

Block Multiplier

Layout

Create RPM

Design Feedback

Latency

20

< Back

Next >

Page 4 of 4

Generate

Dismiss

Data Sheet...

Version Info...

4. Coregen produces the following files after generation:

Xilinx System Generator User's Guide
www.xilinx.com

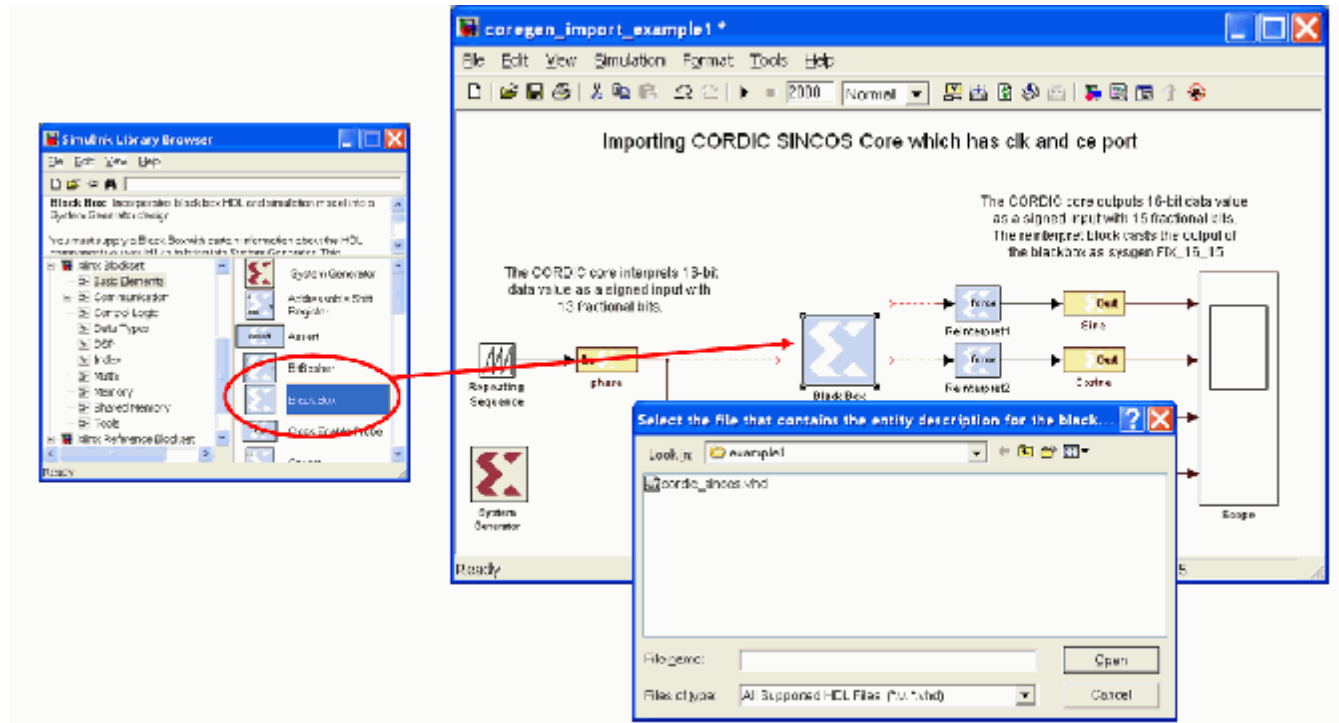
262

- cordic_sincos.edn: Implementation netlist
- cordic_sincos.vhd: VHDL wrapper for behavioral simulation
- cordic_sincos.vho: Core instantiation template
- cordic_sincos.xco: Parameters selected for core generation

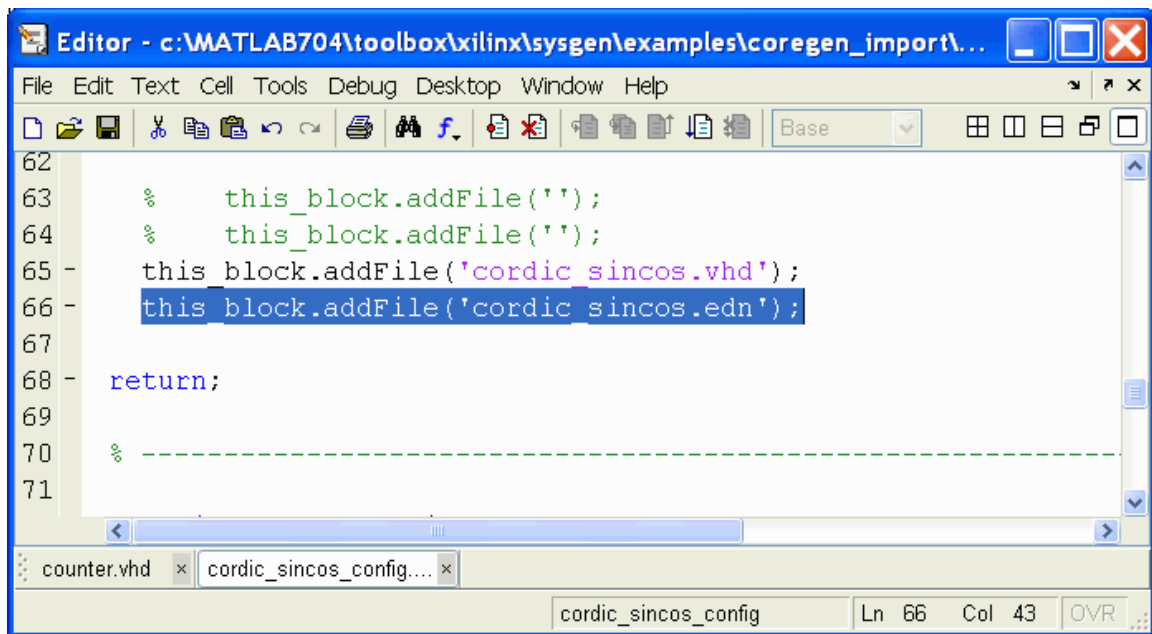
5. Start Simulink and open the design file

(./Xilinx/sysgen/examples/coregen_import/example1/coregen_import_example1.mdl)

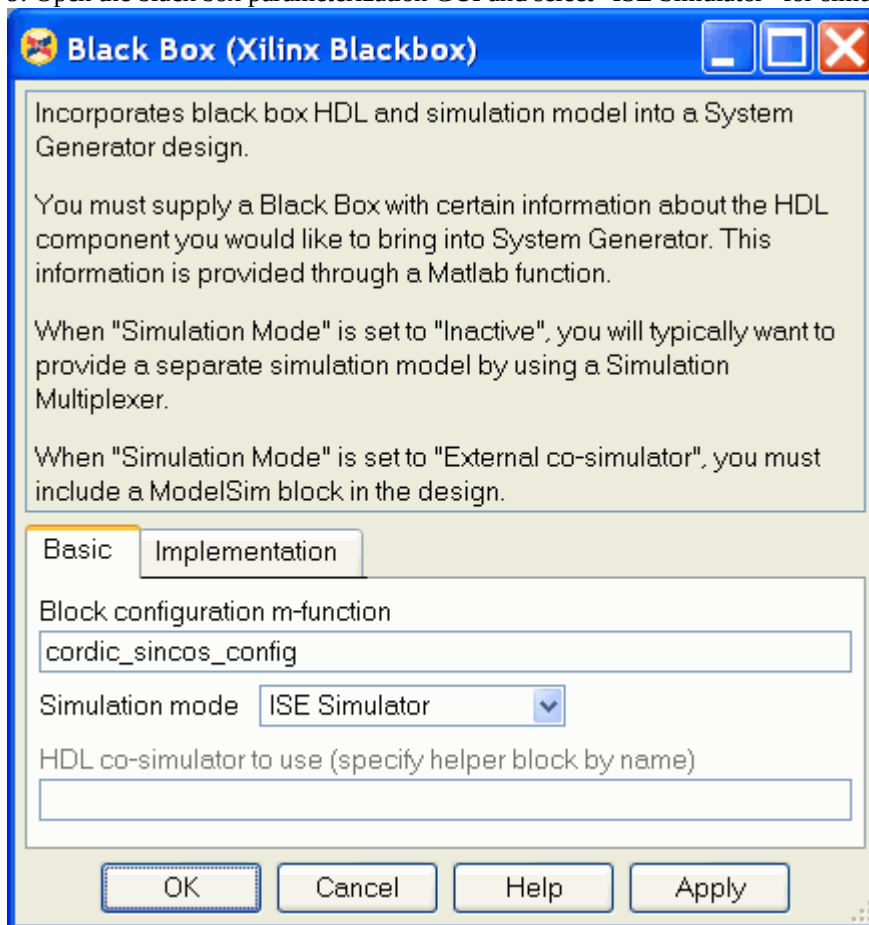
6. Drag and drop the black box from the “Basic Elements” library in the coregen_import_example1.mdl. Select cordic_sincos.vhd for the top level HDL file.



7. Connect the input and output ports of the black box to the open wires.
8. Open the cordic_sincos_config.m file, and add the EDIF netlist to the black box file list as shown below. This file would get included as part of the sysgen netlist for the design when it is netlisted.

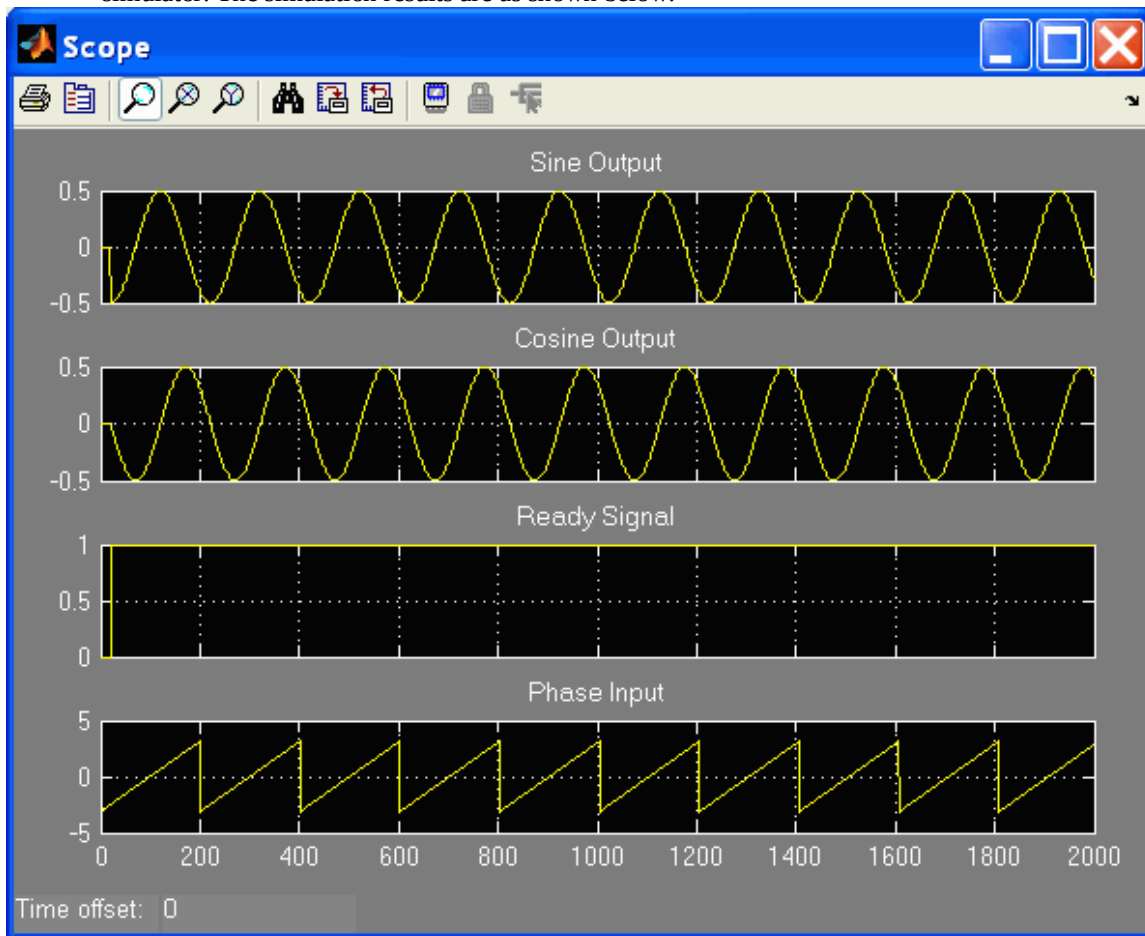


9. Open the black box parameterization GUI and select “ISE Simulator” for simulation mode.



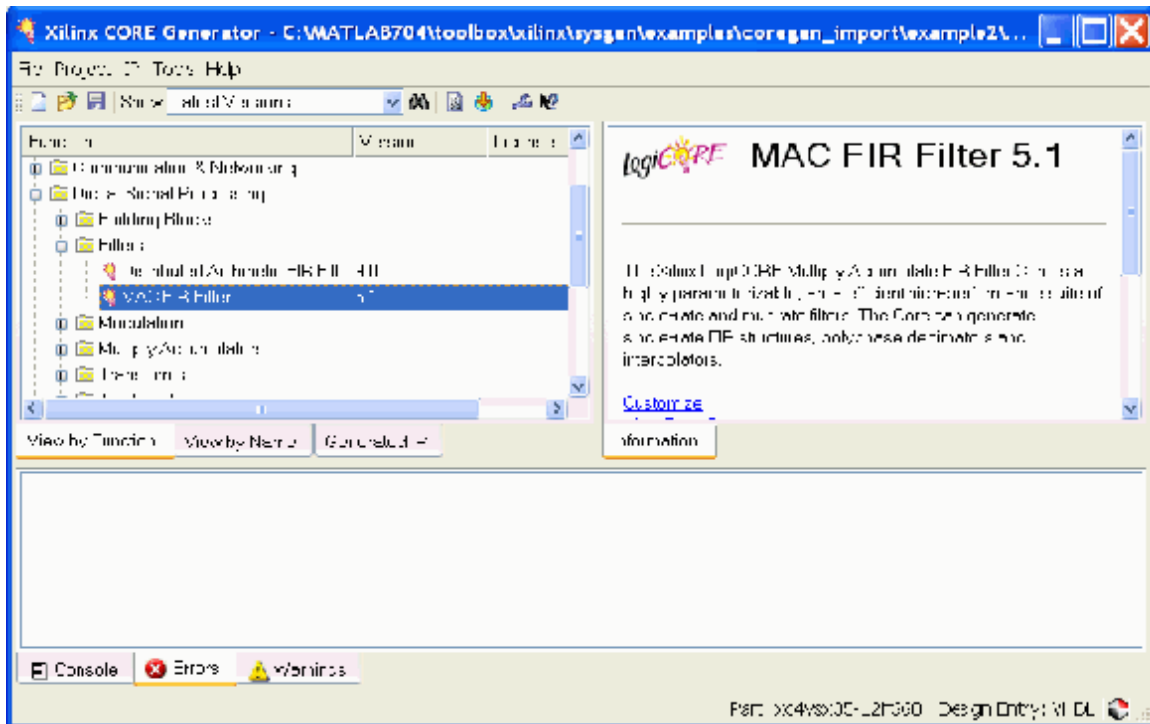
10. Pressing the simulate button would compile and co-simulate the CORDIC core using the ISE

simulator. The simulation results are as shown below.



Example 2. Importing a Coregen Core that needs a VHDL wrapper to satisfy black box HDL requirements

1. Start coregen and open the coregen project file
(./Xilinx/sysgen/examples/coregen_import/example1/coregen_import_example2.cgp).
2. Double click the MAC FIR Filter 5.1 icon to launch the customization GUI



3. Customize and generate the MAC FIR Filter 5.1 core with the following parameters,

- Component Name : "mac_fir_8tap"
- Number of Taps: 8
- Impulse Response: "Symmetric"
- Load Coefficients: mac_fir_8tap.coe file located in sysgen directory
- System Clock Rate: 100 Mhz, Input rate: 25 Mhz
- Other parameters are set to default value.

MAC FIR Filter

ParametersCore OverviewContactWeb Links



MAC FIR Filter

DIN
ND
RESET
CLK
COEF_SEL

DOUT
RDY
RFD
SEL_I
SEL_O

Component Name

mac_fir_8tap

Filter Type

☒ Single Rate FIR

☐ Polyphase Interpolator

☐ Polyphase Decimator

Filter Options

Interpolation Factor

1

Valid Range 2..256

Decimation Factor

1

Valid Range 2..256

Channels

1

< Back

Next >

Page 1 of 5

Generate

Dismiss

Data Sheet...

Version Info...

☐ Display Core Footprint

Xilinx System Generator User's Guide
www.xilinx.com

267

MAC FIR Filter

Parameters

Core Overview

Contact

Web Links



MAC FIR Filter

DIN

ND

RESET

CLK

COEF_SEL

DOUT

RDY

RFD

SEL_1

SEL_0

Structure

Taps Valid Range 1..1024

Impulse Response

☒ Symmetric
☐ Non Symmetric
☐ Negative Symmetric

Coefficients

Coefficient Width Valid Range 2..18

Number of Coefficient Sets Valid Range 1..256

Coefficient Type

☒ Signed
☐ Unsigned

Coefficient Buffer Type

☒ Block Memory
☐ Distributed Memory

< Back

Next >

Page 2 of 5

Generate

Dismiss

Data Sheet...

Version Info...

☐ Display Core Footprint

Xilinx System Generator User's Guide
www.xilinx.com

268

MAC FIR Filter

ParametersCore OverviewContactWeb Links



MAC FIR Filter

DIN
ND
RESET
CLK
COEF_SEL

DOUT
RDY
RFD
SEL_I
SEL_O

COE File

.COE File Name

gen_importexample2\mac_fir_8tap.coe

Load CoefficientsShow Coefficients

Input Data

Data Width

16

Valid Range 2..17

Data Type

☒ Signed☐ Unsigned

Data Buffer Type

☒ Block RAM☐ Distributed RAM

< BackNext >

Page 3 of 5

GenerateDismissData Sheet...Version Info...

☐ Display Core Footprint

Xilinx System Generator User's Guide
www.xilinx.com

269

MAC FIR Filter

Parameters

Core Overview

Contact

Web Links



MAC FIR Filter

DIN

ND

RESET

CLK

COEF_SEL

DOUT

RDY

RFD

SEL_I

SEL_O

Performance Optimization

☒ Auto

☐ Minimal Area/High Speed

☐ Minimum Area

☐ Maximum Speed

System Clock Rate

100.0

Mhz

Valid Range: 25.0..300.0

Input Sample Rate

25

Mhz

Valid Range: 0.01..100.0

Layout

☒ Registered Output

☐ Create RPM

< Back

Next >

Page 4 of 5

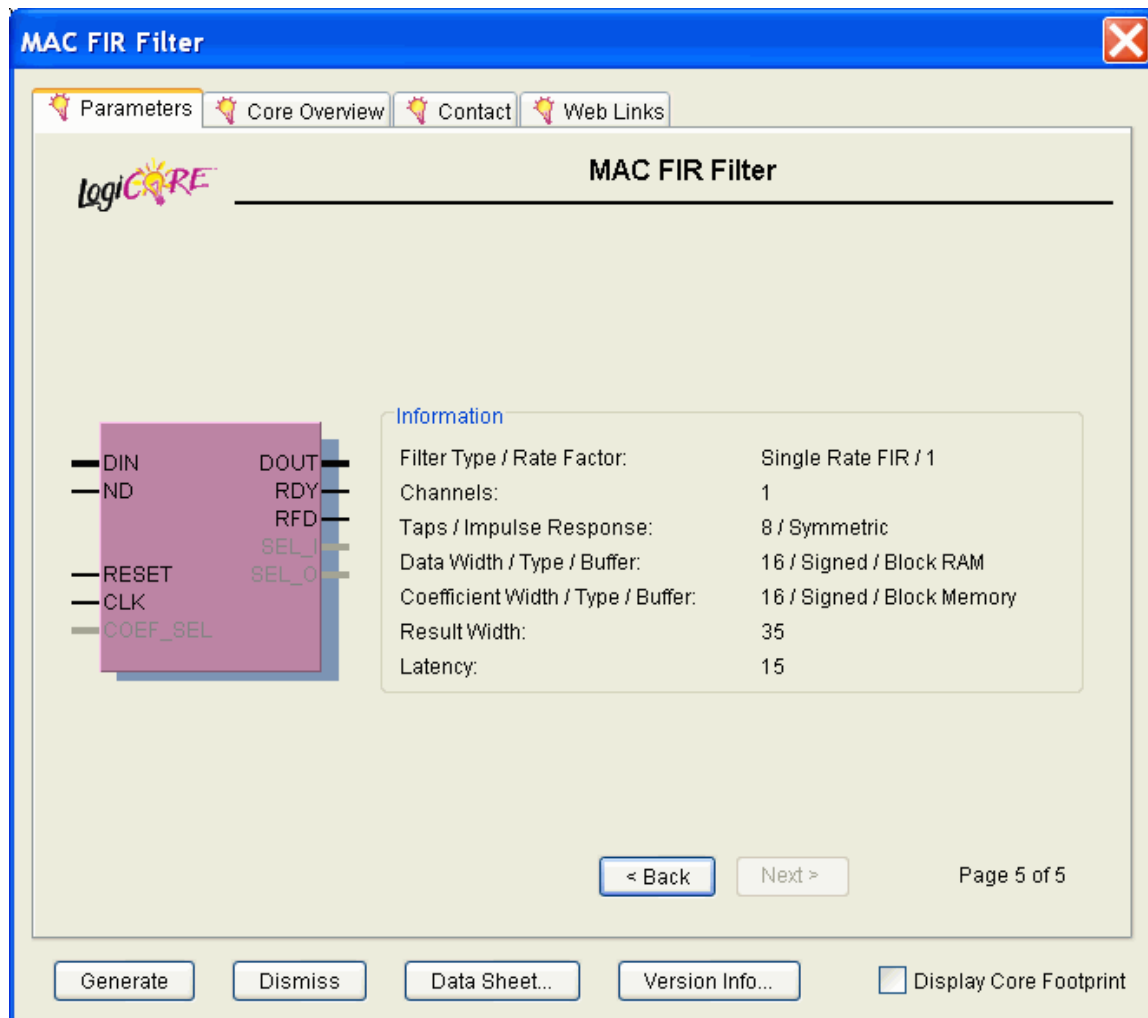
Generate

Dismiss

Data Sheet...

Version Info...

☐ Display Core Footprint



4. Coregen produces the following files after generation:

- mac_fir_8tap.edn: Implementation netlist
- mac_fir_8tap.vhd: VHDL wrapper for behavioral simulation
- mac_fir_8tap.vho: Core instantiation template
- mac_fir_8tap.xco: Parameters selected for core generation
- mac_fir_8tap.mif, COEF_BUFFER.mif : Memory initialization file for functional simulation

5. Since the MACFIR core does not have a “ce” port and sysgen blackbox requires clk, ce pair, we would need to specify a core wrapper to add a ce port to the top level.

6. Open the empty template wrapper file

(./Xilinx/sysgen/examples/coregen_import/example2/mac_fir_8tap_wrapper.vhd). This file contains an empty entity declaration.

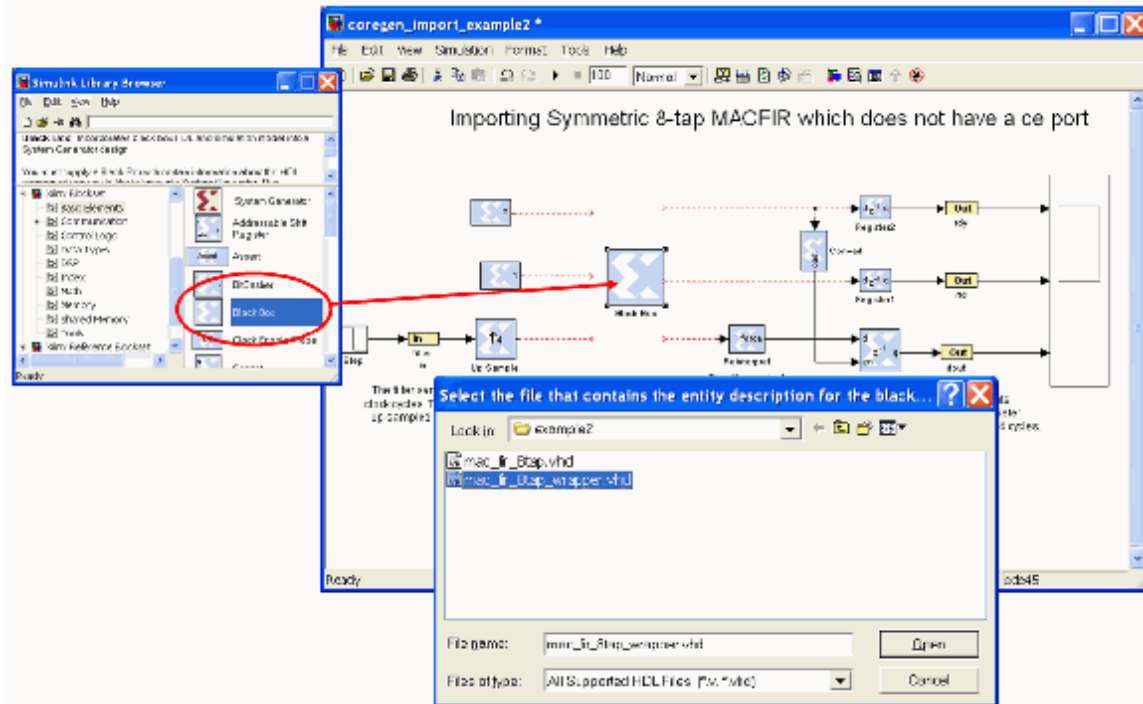
7. Modify the template wrapper according to the instructions below:

- Open the mac_fir_8tap.vho file.
- Copy the component declaration from mac_fir_8tap.vho and paste it in mac_fir_8tap_wrapper.vhd in the component declaration area. (after -- Add Component Declaration from VHO file -----)
- Copy the core instantiation template from mac_fir_8tap.vho and paste it in mac_fir_8tap_wrapper.vhd in the architecture body. (after ----- ADD INSTANTIATION Template -----)
- Copy port declaration for the component mac_fir_8tap and paste it for the mac_fir_8tap entity

- declaration (after ---- Add Port declaration for entity ----)
- Add “ce” port to the top level entity declaration, and change case of “CLK” port to “clk”.

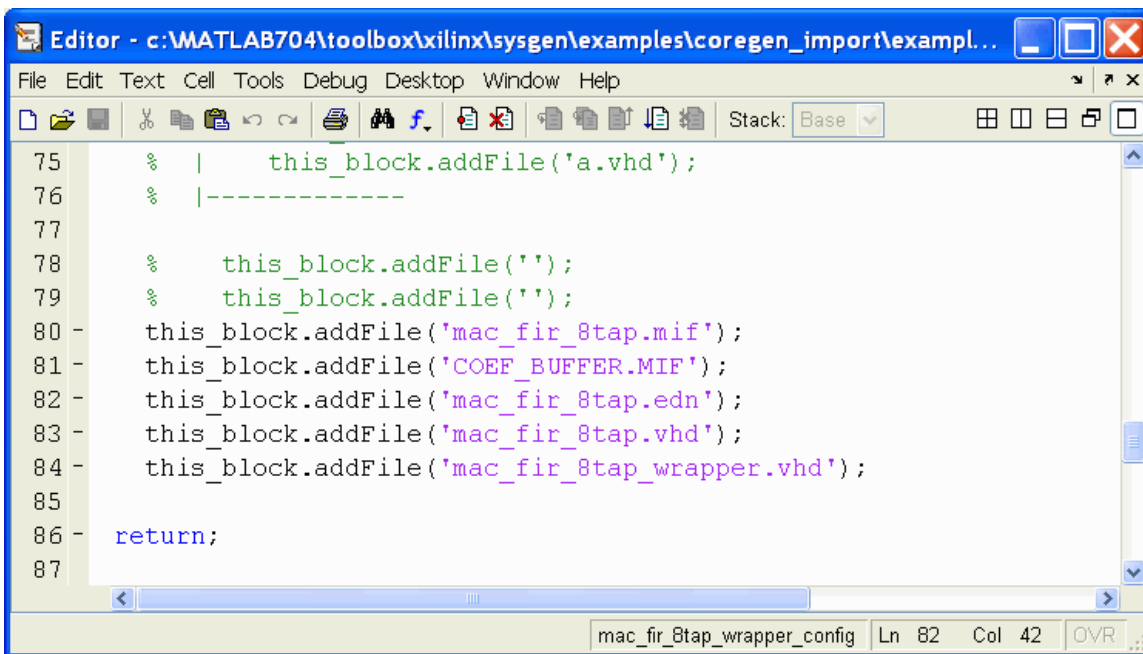


- Start Simulink and open the design file
(./Xilinx/sysgen/examples/coregen_import/example1/coregen_import_example2.mdl)
- Drag and drop the black box from the “Basic Elements” library in the coregen_import_example2.mdl.
Select mac_fir_8tap_wrapper.vhd for the top level HDL file.



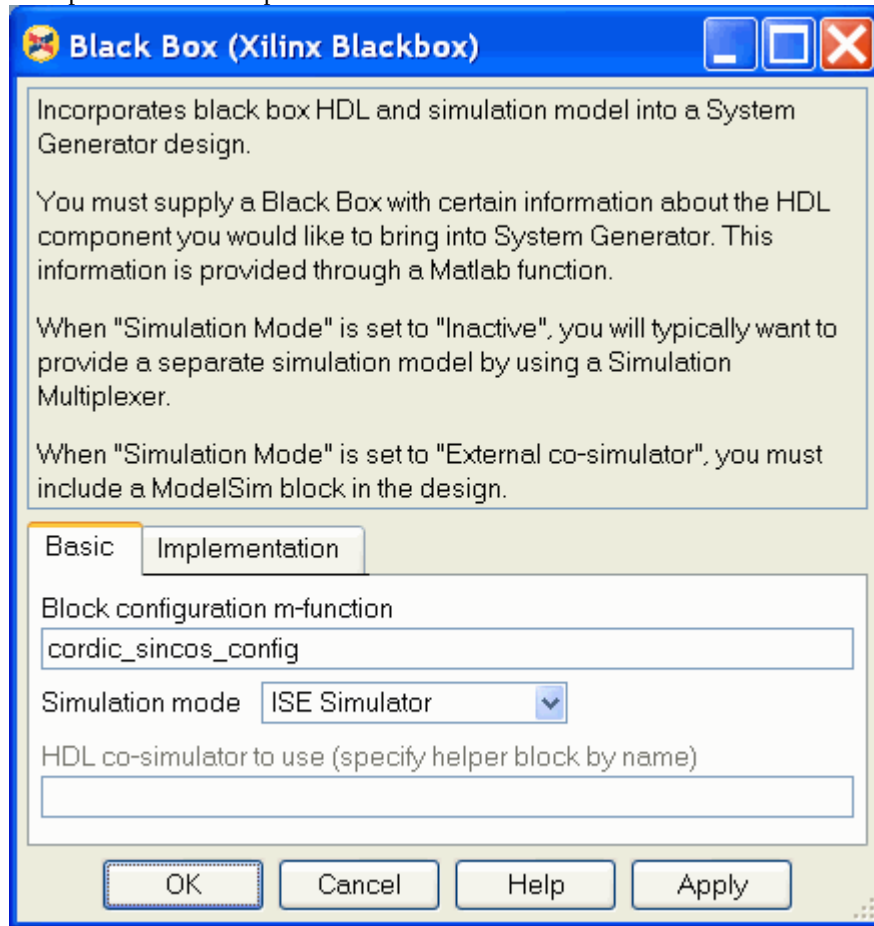
10. Connect up the black box to the open wires.

11. Open the `mac_fir_8tap_wrapper_config.m` file, and add the VHDL file, EDIF netlist and MIF files to the black box file list as shown below. These files would get included as part of the sysgen netlist for the design when it is generated.

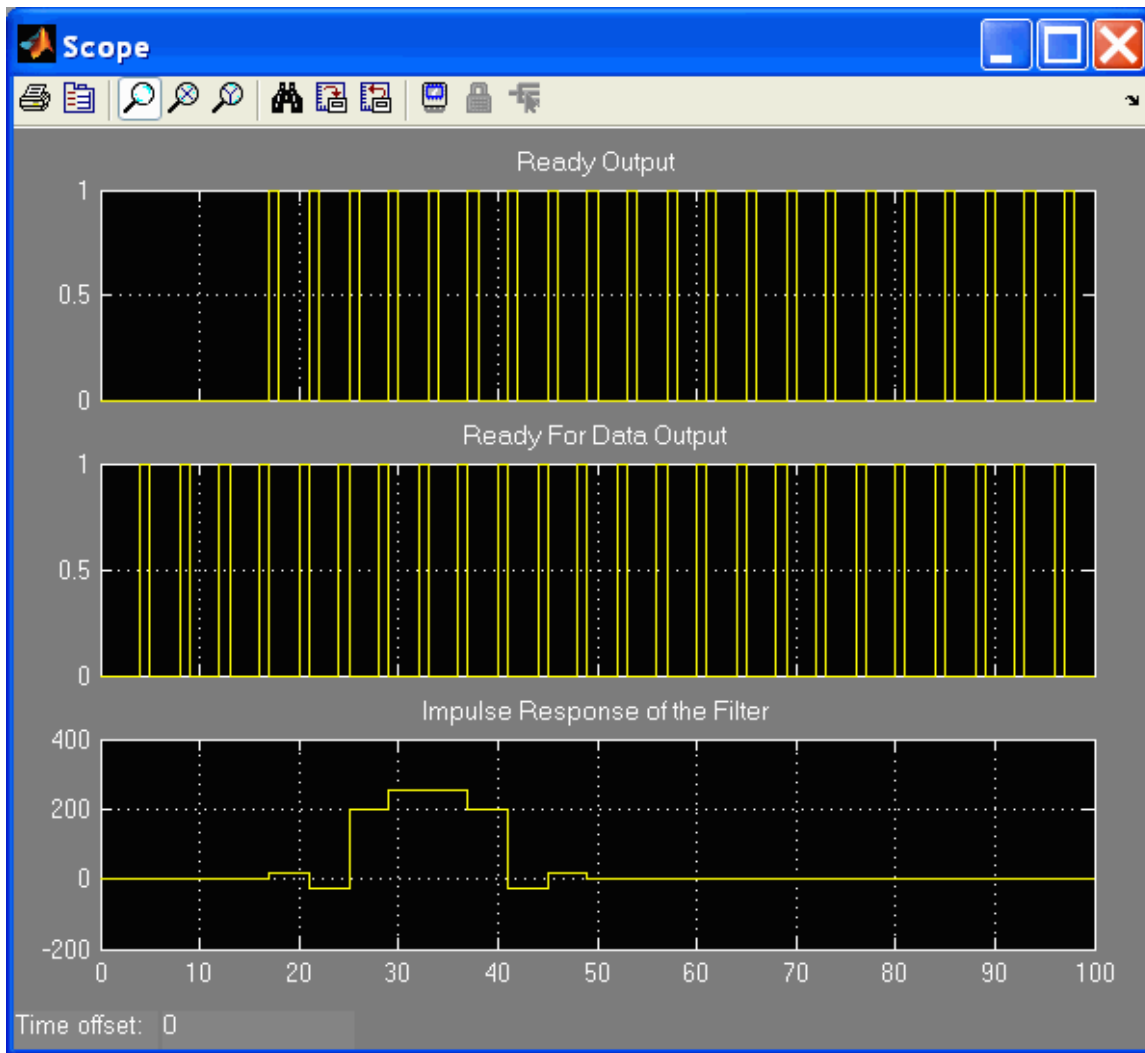


NOTE: The order in which the files are added in the configuration function is the order in which they get compiled during synthesis and simulation.

12. Open the black box parameterization GUI and select “ISE Simulator” for simulation mode.



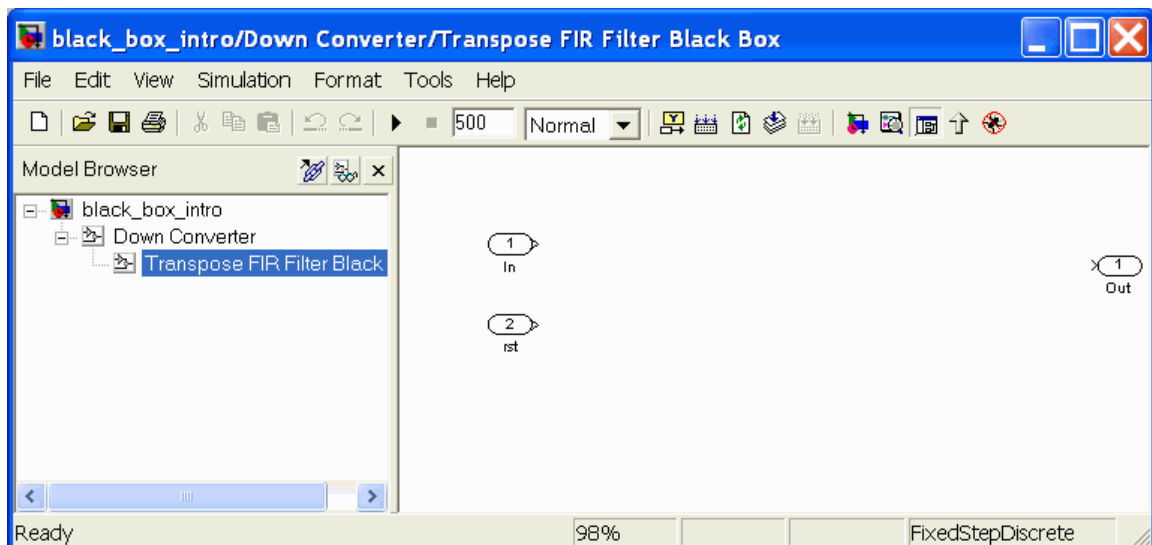
13. Pressing the simulate button would compile and co-simulate the FIR core using the ISE simulator. The simulation results are as shown below.



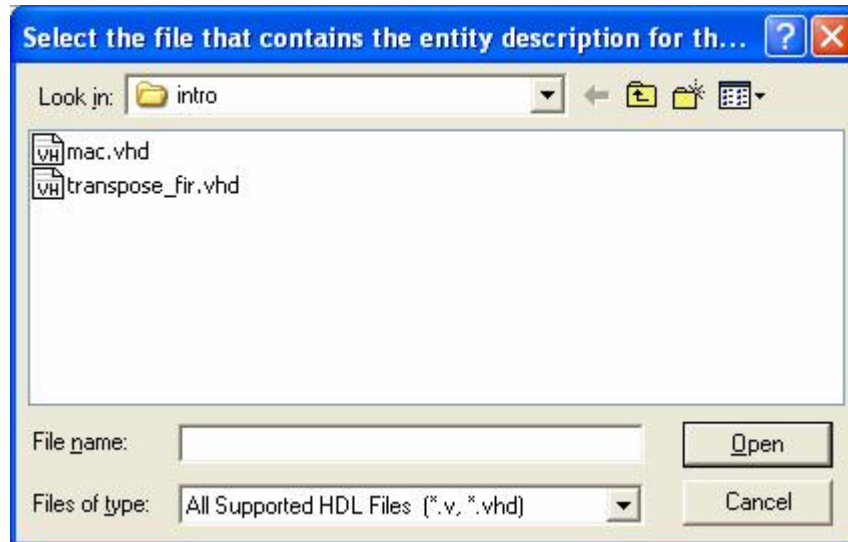
Importing a VHDL Module

This section explains how to use the black box to import VHDL into a System Generator design and how to use Modelsim to co-simulate the VHDL module.

1. From the MATLAB console, change directory to `<path_to_sysgen>/examples/black_box/intro`. The following files are located in this directory:
 - `black_box_intro.mdl` - A Simulink model containing an example black box.
 - `transpose_fir.vhd` - Top level VHDL for a transpose form FIR filter. This file is the VHDL that is associated with the black box.
 - `mac.vhd` - Multiply and add component used to build the transpose FIR filter.
2. Open the `black_box_intro` model from the MATLAB command window by typing
`>> black_box_intro`
3. Open the subsystem named “Transpose FIR Filter Black Box”. At this point, the subsystem contains two inports and one output. The black box subsystem is shown below.



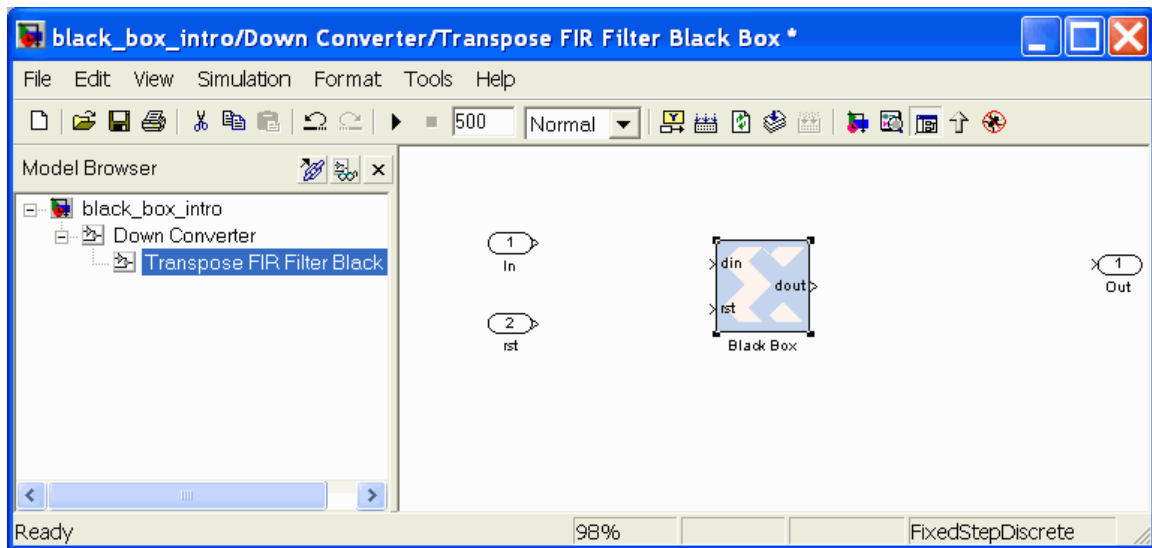
4. Go to the Simulink Library Browser and add a black box block to this subsystem. The black box is located in the Xilinx Blockset's Basic Elements library. The black box configuration wizard is automatically invoked when a new black box is added to the subsystem. A browser window appears that lists the VHDL source files that can be associated with the black box. From this window, select the top level VHDL file `transpose_fir.vhd`. This is illustrated in the figure below.



Note: The wizard will only run if the black box is added to a model that has been saved to a file. If the model has not been saved, the wizard would not know where to search for files and System Generator will instead display a warning that looks like the following.



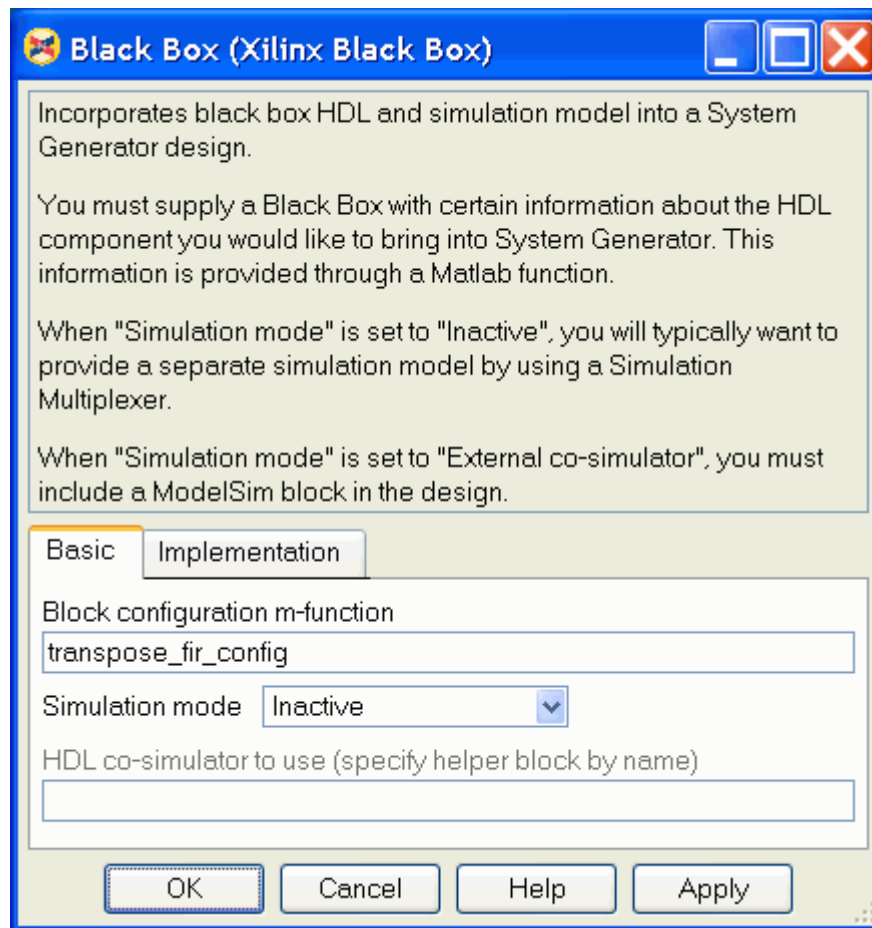
5. The wizard parses the VHDL to generate a “configuration m-function” for the black box. This is a MATLAB script that, among other things, associates the black box to the VHDL and creates black box ports. Once the function has run, the ports on the black box match those in the top level VHDL entity (not including clock and clock enable ports). This is illustrated below.



Please be aware of the following notes when working this example:

- A synchronous HDL design that is associated with a black box must have one or more clock and clock enable ports. These ports must occur in pairs, one clock for each clock enable, and vice-versa. Each of these ports must be of type `std_logic`. The name of the clock port must contain the substring `clk`. The name of the clock enable port must be the same as the name of the clock port, but with `ce` substituted for `clk`.
- The clock enable port has a specific meaning to System Generator and is not a general purpose user enable for the block. Refer to the **black box section** for details.

6. Double click on the black box block. The dialog box shown below appears.



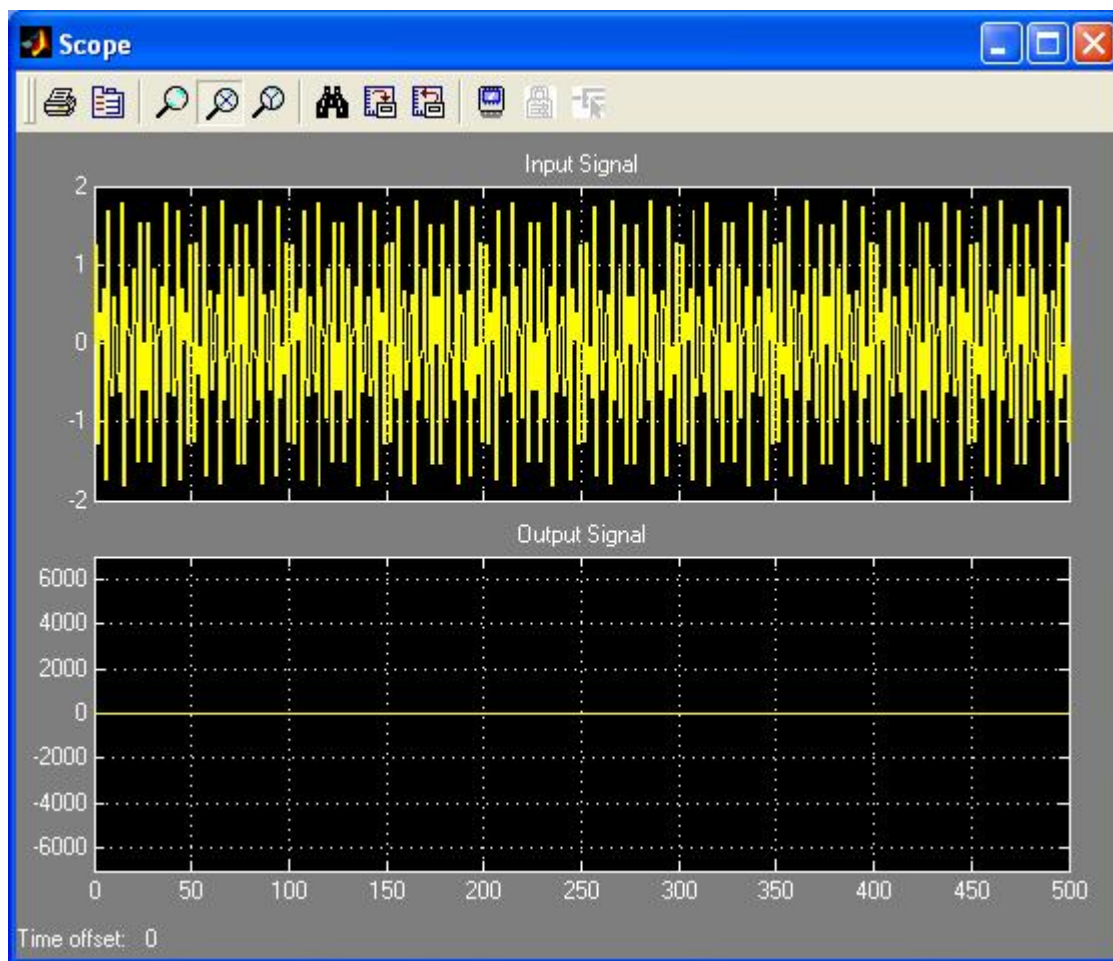
The following are the fields in the dialog box:

- **Block configuration m-function** - This specifies the name of the configuration m-function for the black box. In this example the field contains the name of the function that was generated by the configuration wizard. By default the black box uses the function the wizard produces. You can, however, substitute one you produce yourself. For more information on the configuration m-function, see [The Configuration M-Function and Configuration Wizard](#).
- **Simulation mode** - There are three simulation modes:
 1. **Inactive** - When the mode is *Inactive*, the black box participates in the simulation by ignoring its inputs and producing zeros. This setting is typically used when a separate simulation model is available for the black box, and the model is wired in parallel with the black box using a simulation multiplexer. Example 1 in this tutorial shows how this is accomplished.
 2. **External Co-Simulator** - When the mode is *External co-simulator*, it is necessary to add a ModelSim HDL co-simulation block to the design, and to specify the name of the ModelSim block in the field labeled *HDL Co-Simulator To Use*. In this mode the black box is simulated using HDL co-simulation.

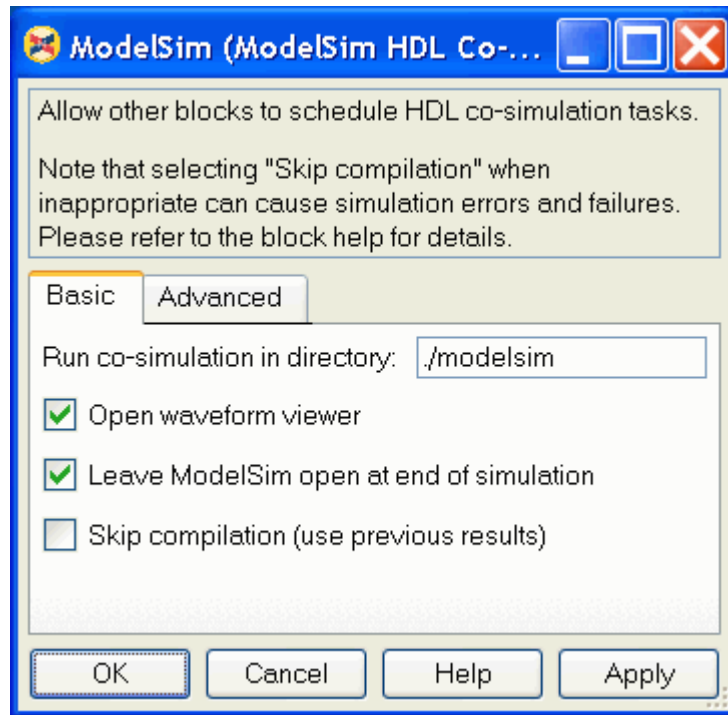
3. **ISE Simulator** - When the mode is *ISE Simulator*, simulation results for the black box are produced using co-simulation on the HDL associated to the black box.
- * **FPGA Area** - The numbers entered in this field are estimates of how much of the FPGA is used by the HDL for the black box. These numbers must be entered by hand. The numbers are only needed if you would like to use the resource estimating utilities supplied with System Generator. For more information, see **Resource Estimation**.

To continue the tutorial, leave the parameters set as they currently are.

7. Wire the black box's ports to the corresponding subsystem ports.
8. Run the simulation by clicking the simulation play button and then double click on the scope block. Notice the black box output shown in the **Output Signal** scope is zero. This is expected as the black box is configured to be inactive during simulation.



9. Go to the Simulink Library Browser and add a ModelSim block to this subsystem. The ModelSim block is located in the **Xilinx Blockset / Tools** library. This block enables the black box to communicate with a Modelsim simulator. Double click on the ModelSim block to open the dialog box shown below:



10. Make sure the parameters match those shown in the preceding figure. Close the dialog box.
11. From the Simulink menu, select **Port Data Types** from the **Format** menu to display the port types for the black box. Compile the model (Ctrl-d) to ensure the port data types are up to date. Notice that the black box port output type is `UFix_26_0`. This means it is unsigned, 26 bits wide and has a binary point 0 positions to the left of the least significant bit.
12. Open the configuration m-function `transpose_fir_config.m` and change the output type from `UFix_26_0` to `Fix_26_12`. The modified line should read:

```
dout_port.setType('Fix_26_12');
```

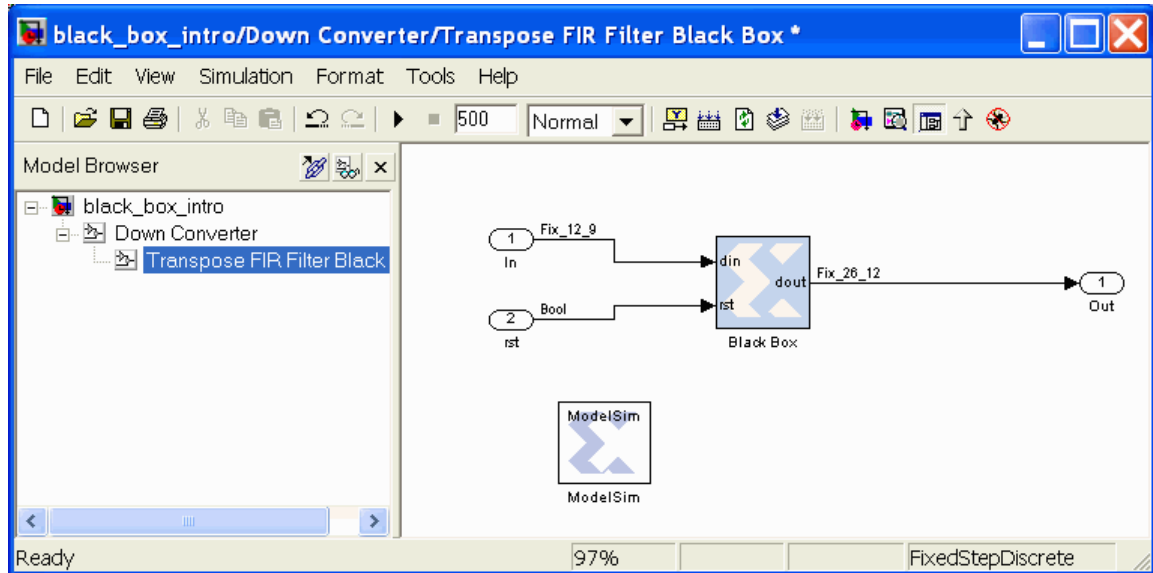
13. Edit the configuration m-function to associate an additional HDL file with the black box. Locate the line:

```
this_block.addFile('transpose_fir.vhd');
```

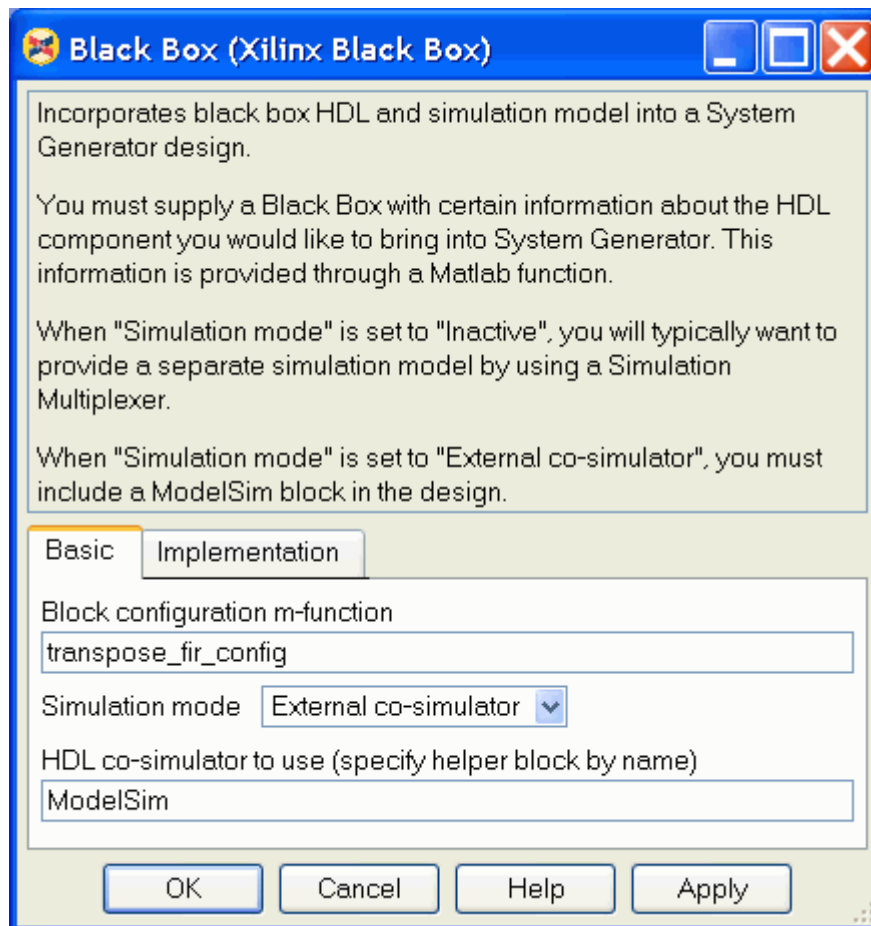
Immediately above this line, add the following:

```
this_block.addFile('mac.vhd');
```

14. Save the changes to the configuration m-function and recompile the model (Ctrl-d). Your subsystem should appear as follows:

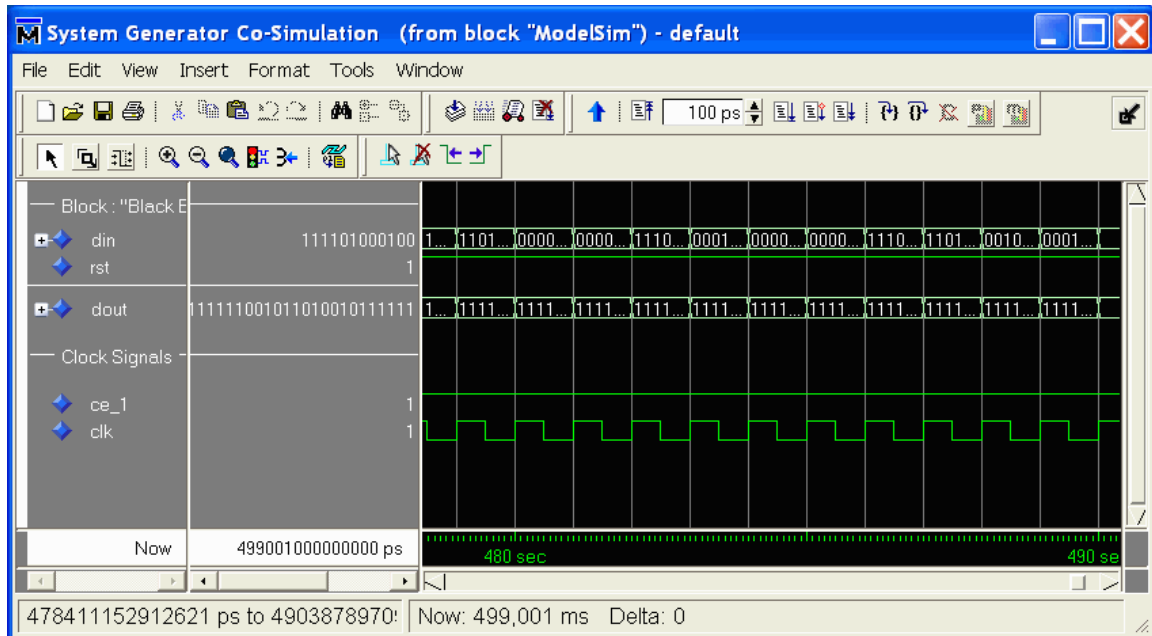


15. From the black box block parameter dialog box, change the Simulation mode field from **Inactive** to **Use HDL Co-Simulation**. Enter **ModelSim** in the **HDL Co-Simulator To Use** field. The name in this field corresponds to the name of the ModelSim block that you added to the model. The black box dialog box should appear as follows.



Note: Only ModelSim XE and SE simulators are supported by System Generator.

16. Run the simulation. A Modelsim command window and waveform viewer opens. Modelsim simulates the VHDL while Simulink controls the overall simulation. The resulting waveform looks something like the following:

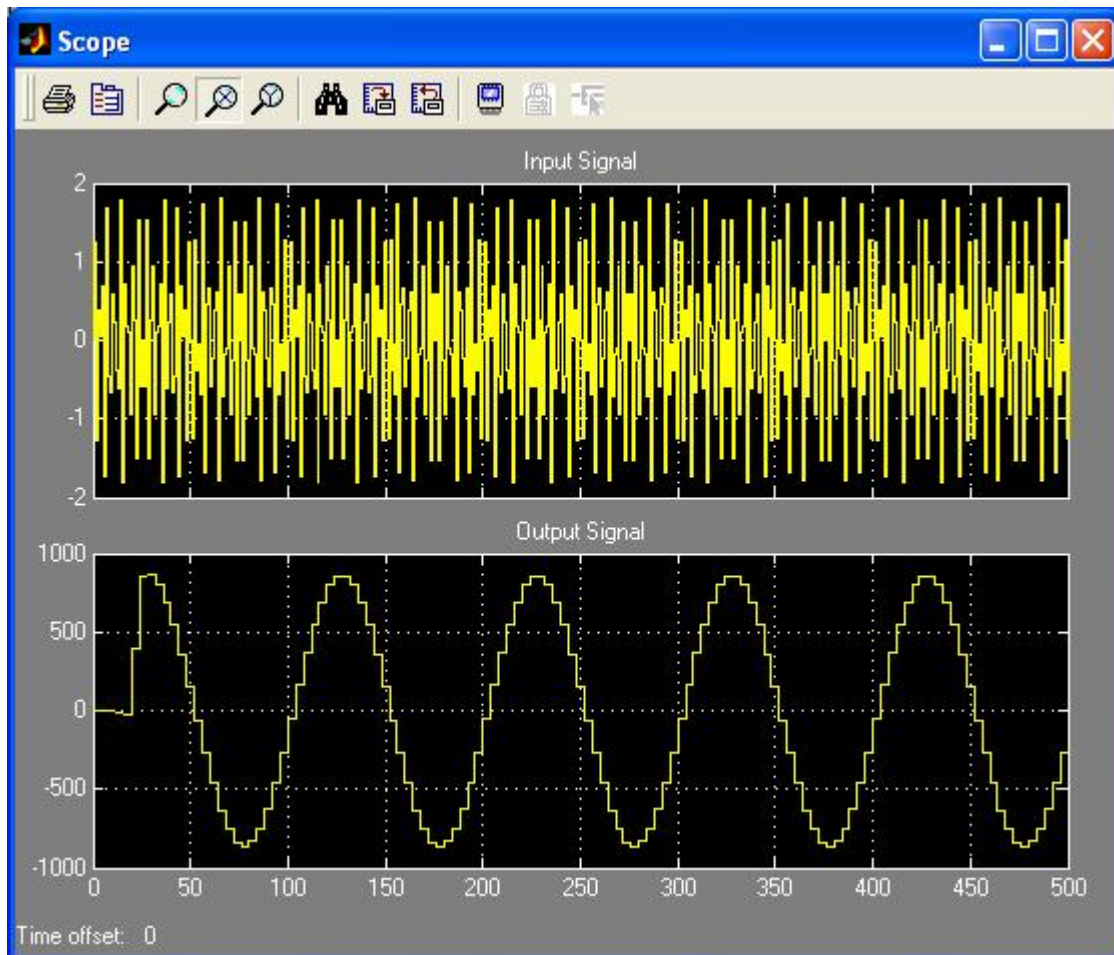


The following warnings received in ModelSim can safely be ignored.

```
# ** Warning: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic
operand, the result will be 'X'(es).
# Time: 0 ps Iteration: 0 Instance: /
xlcossim_black_box_ex1_down_converter_transpose_fir_filter_black_box_modelsim/
black_box_ex1_down_converter_transpose_fir_filter_black_box_black_box/g0_22/g_last/m2
```

They are caused by the black box VHDL not specifying initial values at the start of simulation.

17. Examine the scope output after the simulation has completed. When Simulation Mode was set to **Inactive**, the Output Signal scope displayed constant zero. Notice the waveform is no longer zero. Instead, Output Signal shows the results from the Modelsim simulation.



Importing a Verilog Module

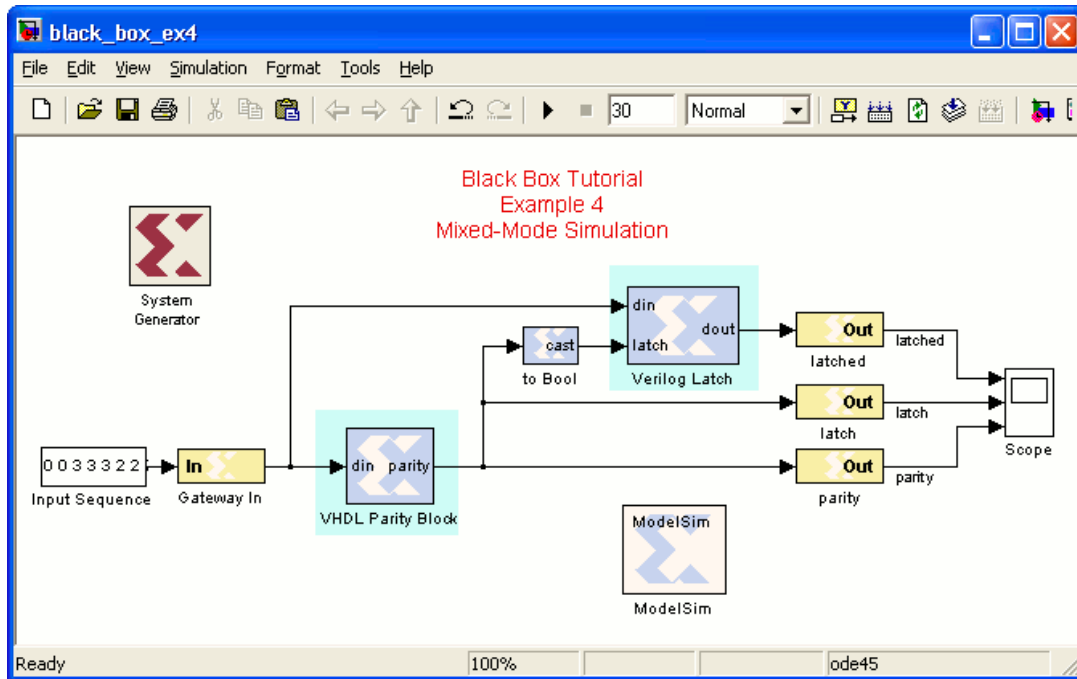
This example demonstrates how Verilog black boxes can be used in System Generator and co-simulated using ModelSim. Verilog modules are imported the same way VHDL modules are imported. For more information on how this is done, see The Configuration M-Function and Configuration Wizard. System Generator provides all of the code that is needed to incorporate Verilog black boxes, both to generate hardware and to co-simulate HDL. System Generator also allows Verilog black boxes to be parameterized. This example demonstrates all of these capabilities. The files for this example are contained in the directory `<path_to_sysgen>/examples/black_box/example4`.

The files contained in this directory are:

- `black_box_ex4.mdl` – A Simulink model with two black boxes, one using VHDL and the other using Verilog.
- `word_parity_block.vhd` – The VHDL for the combinational portion of the state machine seen in word parity example presented above. This is a purely combinational (stateless) block that computes the parity of each input word and outputs the parity bit. It has been parameterized with a generic so that it can accept any input type (see the description of dynamic black boxes for a discussion of generics).
- `word_parity_block_config.m` – The configuration m-function for the VHDL black box, including the generic setting. The m-function tags this block as combinational so that it simulates correctly in Simulink.
- `shutter.v` – The Verilog for a simple synchronous latch. The code has been parameterized so that the input port `din` can have arbitrary width.
- `shutter_config.m` – The configuration m-function for the Verilog black box, including the parameter setting. The configuration m-function uses methods referring to VHDL syntax even for configuring Verilog black boxes. Thus for this black box, we have the lines:

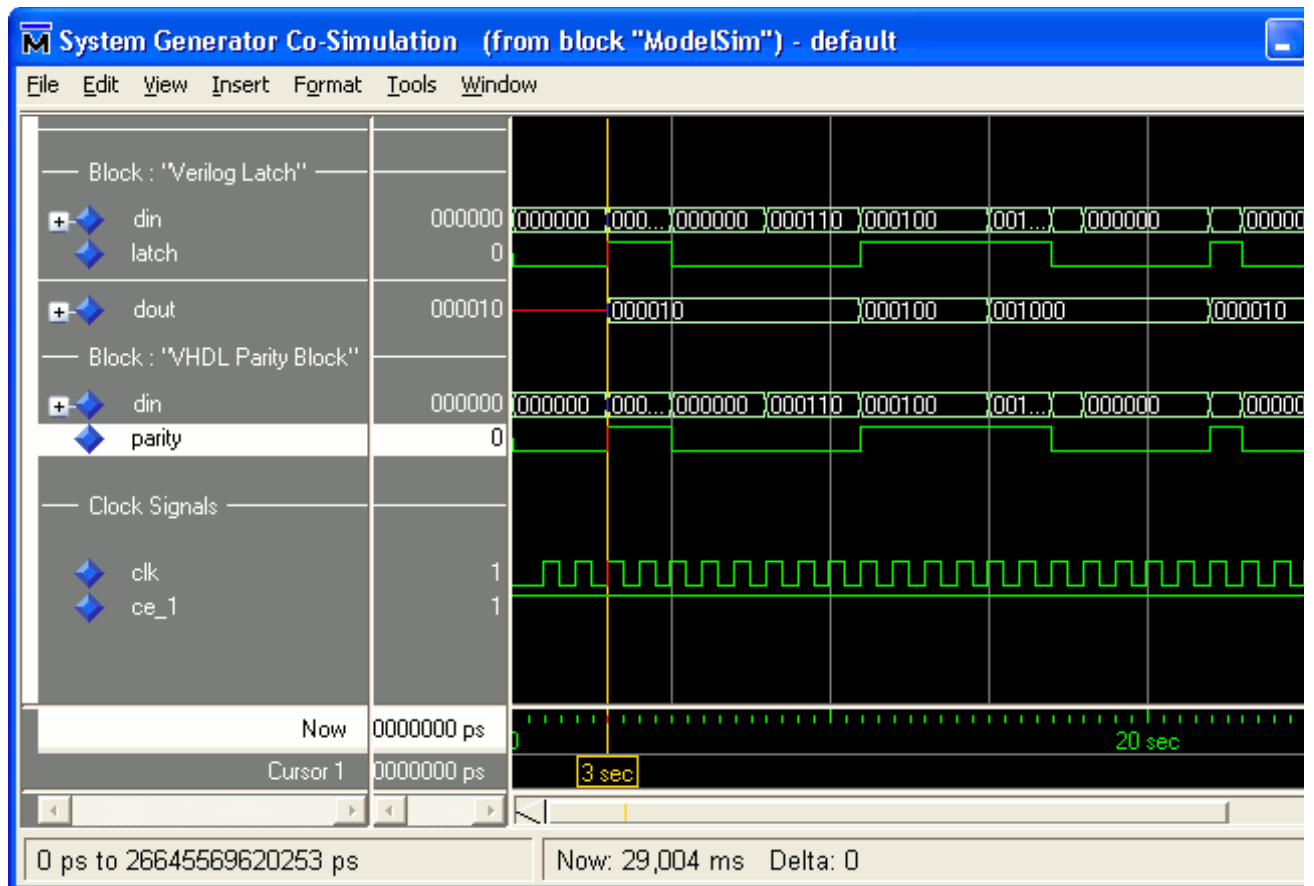
```
this_block.setEntityName('shutter');  
this_block.addGeneric('din_width', dwidth);
```

1. Navigate into the `example4` directory and open the example model. This is a simple design with two black boxes, one VHDL and the other Verilog. The VHDL black box computes the parity of each input word, and the Verilog black box latches the words that have odd parity. No Simulink model is used to compute the behavior of the black boxes; instead, HDL co-simulation is used. The example model is shown in the figure below.



You must have a license for mixed-mode ModelSim simulation to run this example. If you do and you run the simulation, you will see a ModelSim waveform window that looks like the one captured below. The behavior of both black boxes is shown. You can browse the design structure in ModelSim to see how System Generator has combined the two black boxes.

2. Change the input type to an arbitrary type and rerun the simulation. Both black boxes adjust in the appropriate way to the change.

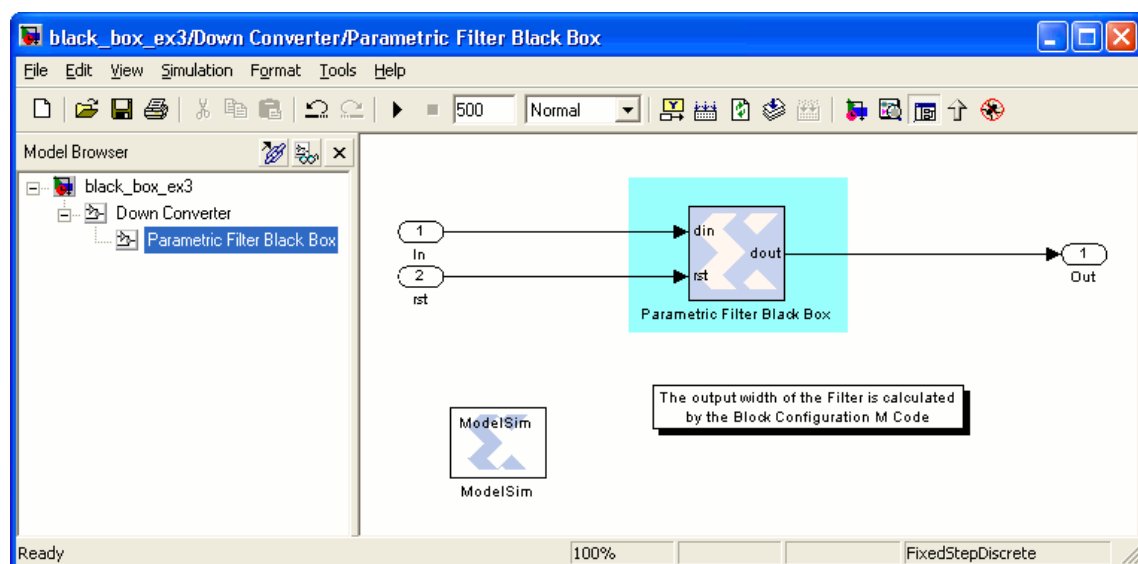


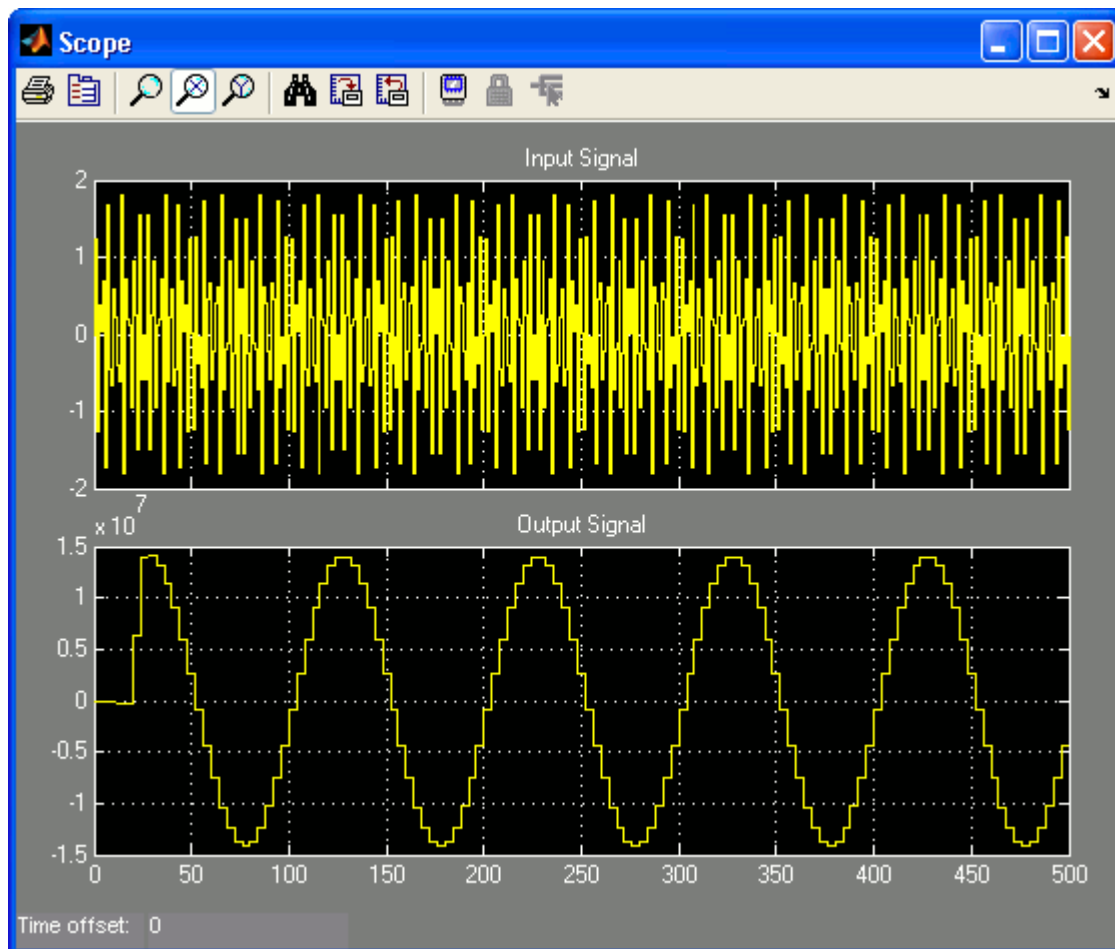
Dynamic Black Boxes

This example extends the transpose FIR filter black box so that it is dynamic, i.e., able to adjust to changes in the widths of its inputs. The example is contained in the directory `$MATLAB/toolbox/xilinx/sysgen/examples/black_box/example3`. For this example to run correctly, you must change your directory (cd within the MATLAB command window) to this directory before launching the example model.

The files contained in this directory are:

- `black_box_ex3.mdl` - A Simulink model containing a dynamic black box.
 - `transpose_fir_parametric.vhd` - The VHDL for the transpose FIR filter.
 - `mac.vhd` - Multiply and add component used to build the transpose FIR filter.
 - `transpose_fir_parametric_config.m` - The configuration m-function for the black box.
1. Open the model by typing `black_box_ex3` at the MATLAB command prompt.
 2. Run the simulation from the top level model, and view the results displayed in the scopes.
 3. Reduce the number of bits on the gateway **Din Gateway In** from 16 bits down to 12 and the binary point from 14 to 10, then run the simulation again. Note that both the input and output widths on the black box adjust automatically. The black box subsystem and simulation results should look like those shown below.





4. The black box is able to adjust to changes in input width because of its configuration m-function. To make this work, the m-function must be augmented by hand. Open the m-function file `transpose_fir_parametric.m`. The important points are described below.

- Obtaining data input width:

```
input_bitwidth = this_block.port('din').width;
```

- Calculating output width:

```
output_bitwidth = ceil(log2(2^(input_bitwidth-1)*2^(coef_bitwidth-1) *  
number_of_coef));
```

- Setting output data type:

```
dout_port.makeSigned;  
dout_port.width = output_bitwidth;  
dout_port.binspt = 12;
```

- Passing input and output bit widths to VHDL as generics:

```
this_block.addGeneric('input_bitwidth',this_block.port('din').width);  
this_block.addGeneric('output_bitwidth',output_bitwidth);
```

For details concerning the black box configuration m-function, see The Configuration M-Function and

Configuration Wizard.

If you examine the black box VHDL file `transpose_fir_parametric.vhd` you see generics `input_bitwidth` and `output_bitwidth` that specify input and output width. These are passed to lower level VHDL components.

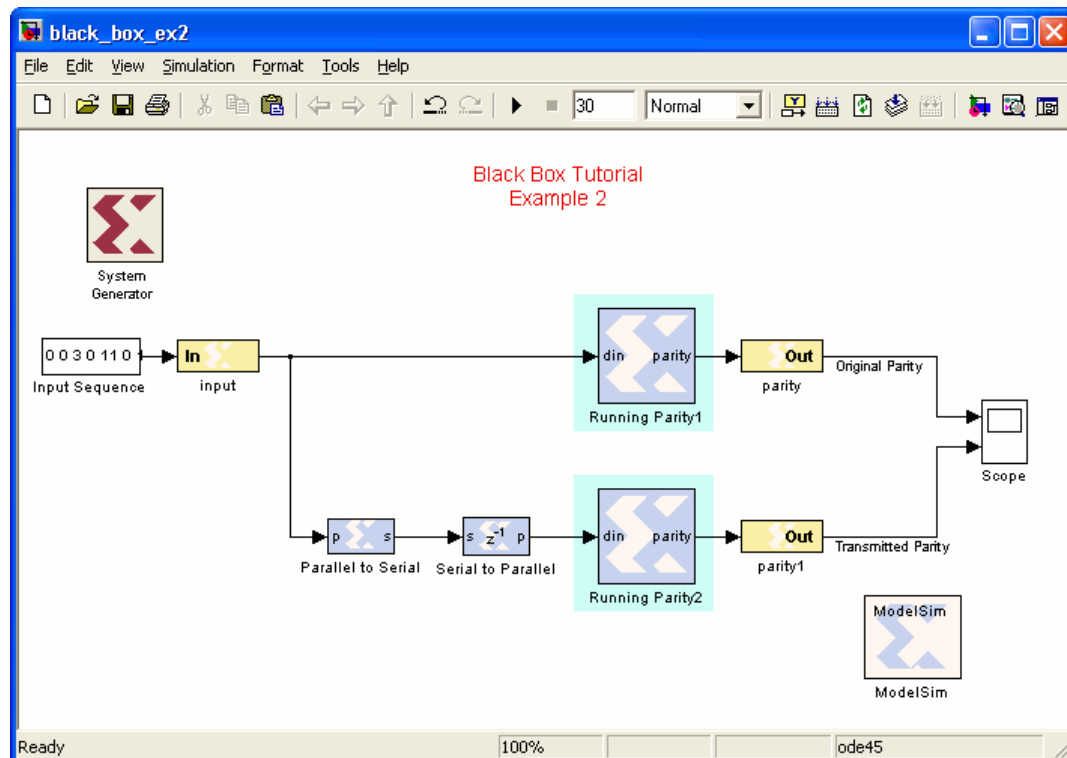
Simulating Several Black Boxes Simultaneously

Several System Generator black boxes can co-simulate simultaneously, using only one ModelSim license while doing so. The example shown below illustrates this. The files for the example are contained in the directory `<path_to_sysgen>/examples/black_box/example2`.

The files contained in this directory are:

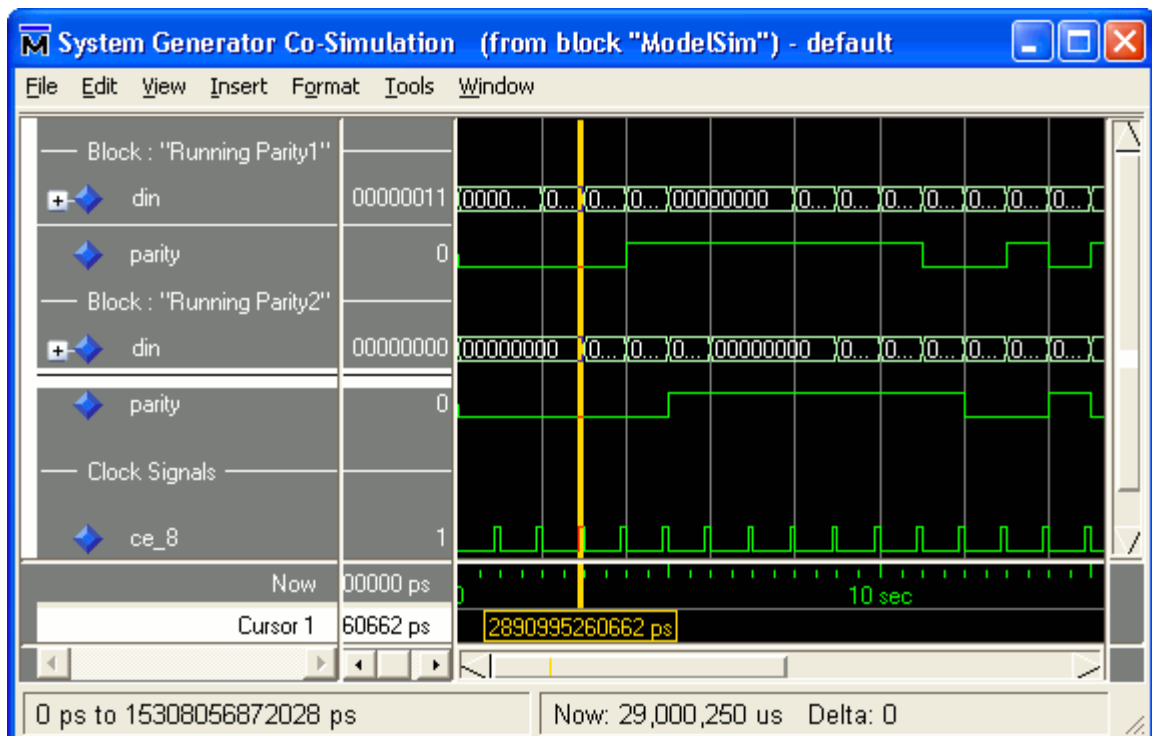
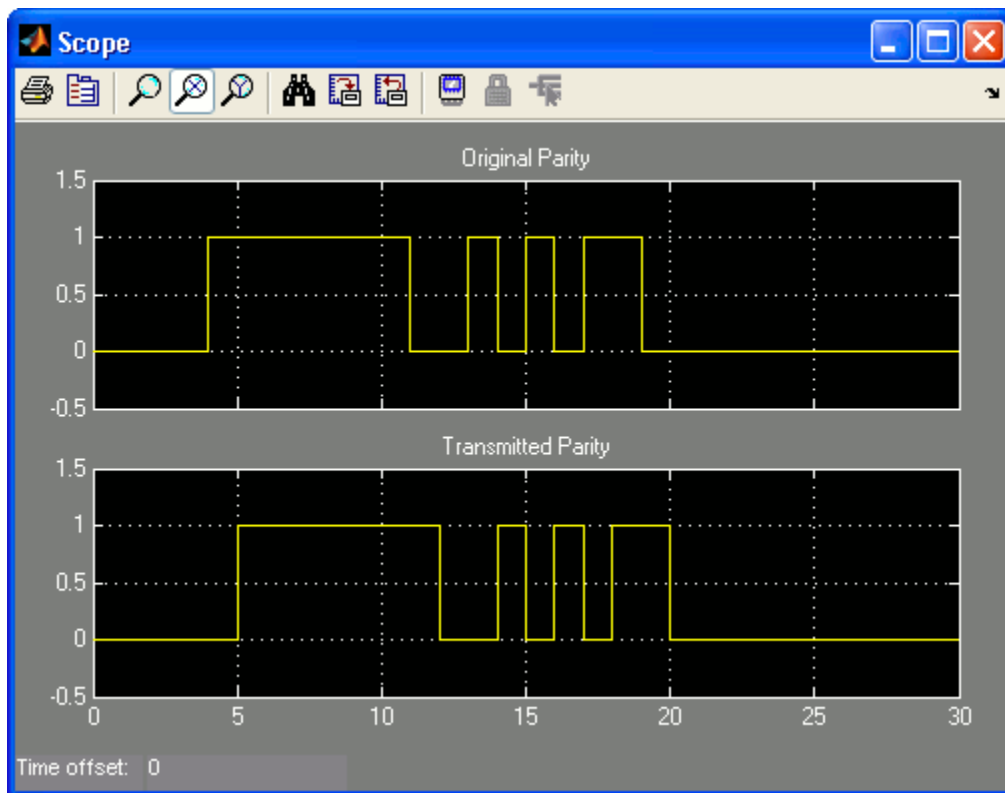
- `black_box_ex2.mdl`: A Simulink model containing two black boxes.
- `parity_block.vhd`: VHDL for a simple state machine that tracks the running parity of an 8-bit input word.
- `parity_block_config.m`: The configuration m-function for the black boxes. The code has barely been changed from what was produced by the configuration wizard: the line that tagged the block as having a combinational feed-through path (`this_block.tagAsCombinational`) has been removed.

Navigate into the `example2` directory and open the `example.mdl`. This is a simple model with two identical black boxes, each implementing a state machine. The state machines compute the running parity of their inputs. One black box is fed the input stream of the model and the other is fed the input stream after it has been serialized and de-serialized. Notice that no simulation model is provided for either state machine. Instead, HDL co-simulation is used to produce simulation results. The ModelSim block provides the connection between the black boxes and ModelSim. The example model is shown in the figure below.



If you run the simulation, you will see a Simulink scope and ModelSim waveform window that look like the figures below. The scope shows that the black boxes produce matching parity results (as expected), but with one delayed from the other by one clock cycle. The waveform window shows the

same results, but viewed in ModelSim and expressed in binary. System Generator automatically configures the waveform viewer to display the input and output signals of each black box. You can also browse the design structure in ModelSim to see how System Generator has elaborated the design to combine the two black boxes.



System Generator Compilation Types

HDL Netlist Compilation

System Generator uses the HDL Netlist compilation type as the default generation target. More details regarding the HDL Netlist compilation flow can be found in the “Compilation Results” section of the Users Guide.

You may select HDL netlist compilation by left-clicking the **Compilation** submenu control on the System Generator block dialog box, and selecting the **HDL Netlist** target (Figure 1).

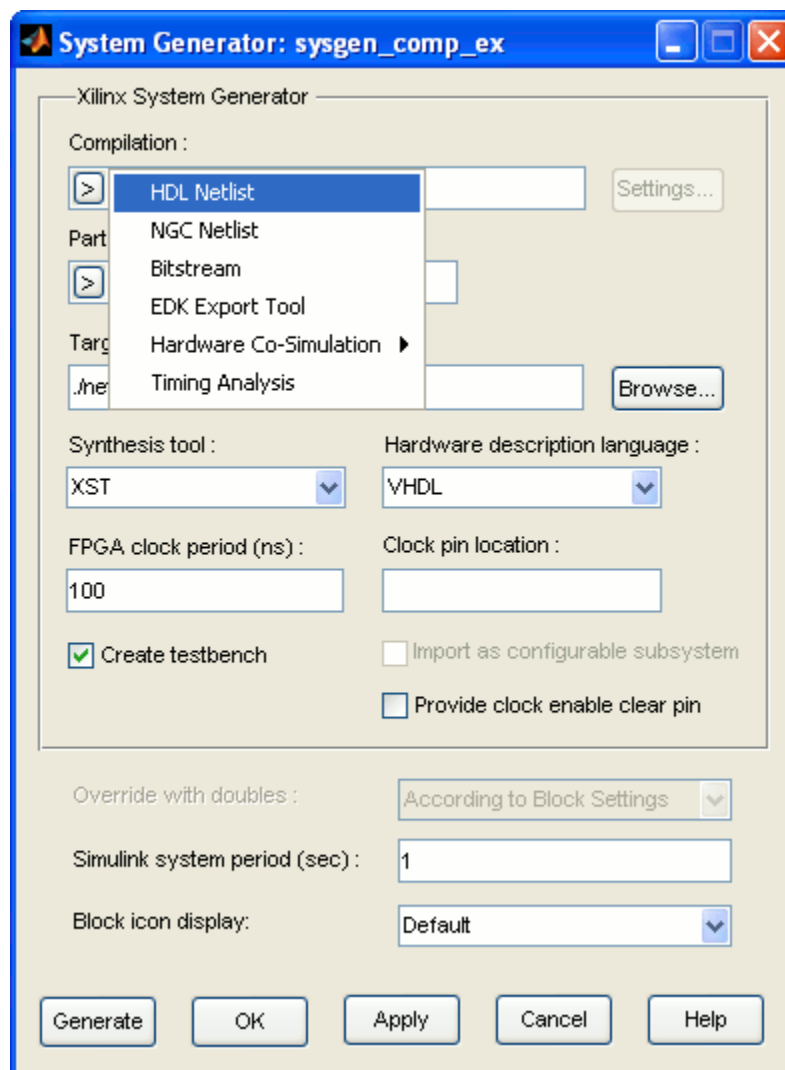


Figure 1: Selecting HDL Netlist compilation for a design.

NGC Netlist Compilation

The NGC Netlist compilation target allows you to compile your design into a standalone Xilinx NGC binary netlist file. The NGC netlist file that System Generator produces contains the logical and optional constraint information for your design. This means that the HDL, cores, and constraints file information that correspond to a System Generator design are self-contained within a single file.

Depending on whether you have chosen to include clock wrapper logic in your design, the netlist file is saved as `<design>_cw.ngc` – when the clock wrapper is included, or `<design>.ngc` otherwise. Here `<design>` is derived from the portion of the design being compiled. This file can be used as a module in a larger design, or as input into NGDBuild when the netlist constitutes the complete design. For an example showing how a System Generator design can be used as a component in a larger design, refer to the Importing a System Generator Design into a Bigger System section in the User Guide.

The NGC compilation target generates an HDL component instantiation template that makes it easy to include your System Generator design as a component in a larger design. For VHDL compilation, the template is saved as `<design>_cw.vho` when the clock wrapper is included, `<design>.vho` otherwise. Alternatively, a `.veo` extension is used for Verilog compilation. The instantiation template is saved in the design's target directory.

System Generator produces the NGC netlist file by performing the following steps during compilation:

- Generating an HDL netlist for the design;
- Running the selected synthesis tool to produce a lower-level netlist. The type of netlist (e.g., EDIF for Synplify, NGC for XST) depends on which synthesis tool is chosen for compilation.
Note: IO buffers are not inserted in the design during synthesis.
- Combines synthesis results, core netlists, black box netlists, and optionally the constraints files into a single NGC file.

You may select the NGC compilation target by left-clicking the **Compilation** submenu control on the System Generator block dialog box, and selecting the **NGC NetList** target (Figure 1).

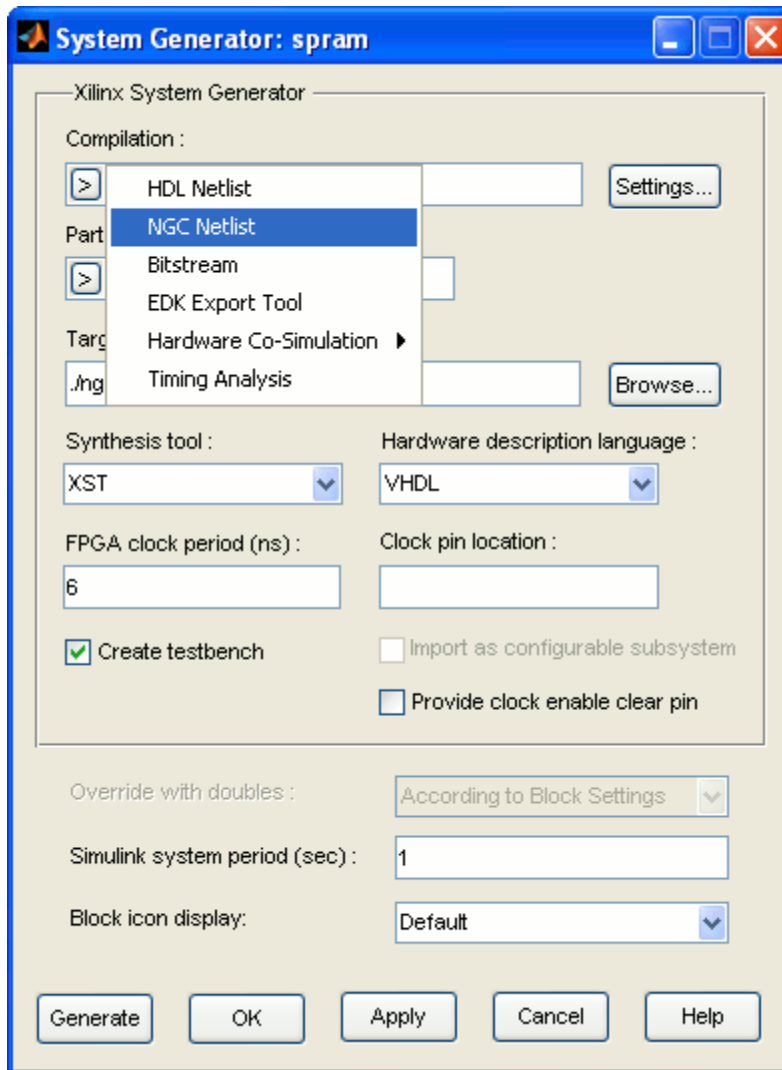


Figure 1: Selecting NGC Netlist compilation for a design.

You may access additional compilation settings specific to NGC Netlist compilation by clicking on the **Settings...** button when NGC Netlist is selected as the compilation type in the System Generator block dialog box. The dialog box containing these settings is shown in Figure 2.

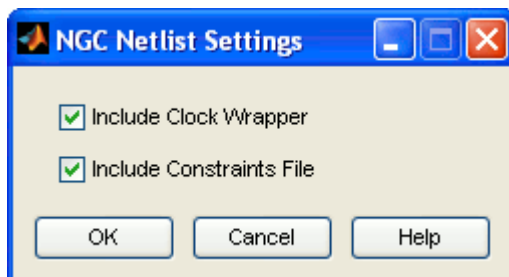


Figure 2: NGC Netlist Settings dialog box.

Parameters specific to the NGC Netlist Settings dialog box include:

- **Include Clock Wrapper:** Selecting this checkbox tells System Generator whether the clock wrapper portion of your design should be included in the NGC netlist file. Refer to the

Compilation Results section in the User Guide for more information on the clock wrapper. Note that by excluding the clock wrapper from multi-rate designs, you will need to drive the clock enable ports with appropriate signals from your own design top-level.

- **Include Constraints File:** Selecting this checkbox tells System Generator whether the constraints file associated with the design should be included in the NGC netlist file. Note that when the constraints file is excluded, you should supply your own constraints to ensure the multi-cycle paths in the System Generator design are appropriately constrained.

Bitstream Compilation

The Bitstream compilation type allows you to compile your design into a Xilinx configuration bitstream file that is suitable for the FPGA part that is selected in the System Generator dialog box. The bitstream file is named `<design>_cw.bit` and is placed in the design's target directory, where `<design>` is derived from the portion of the design being compiled.

System Generator produces the bitstream file by performing the following steps during compilation:

- Generating an HDL netlist for the design;
- Running the selected synthesis tool to produce a lower-level netlist. The type of netlist (e.g., EDIF for Synplify, NGC for XST) depends on which synthesis tool is chosen for compilation.
- Running XFLOW to produce a configuration bitstream.

You may select the Bitstream compilation by left-clicking the **Compilation** submenu control on the System Generator block dialog box, and selecting the **Bitstream** target (Figure 1).

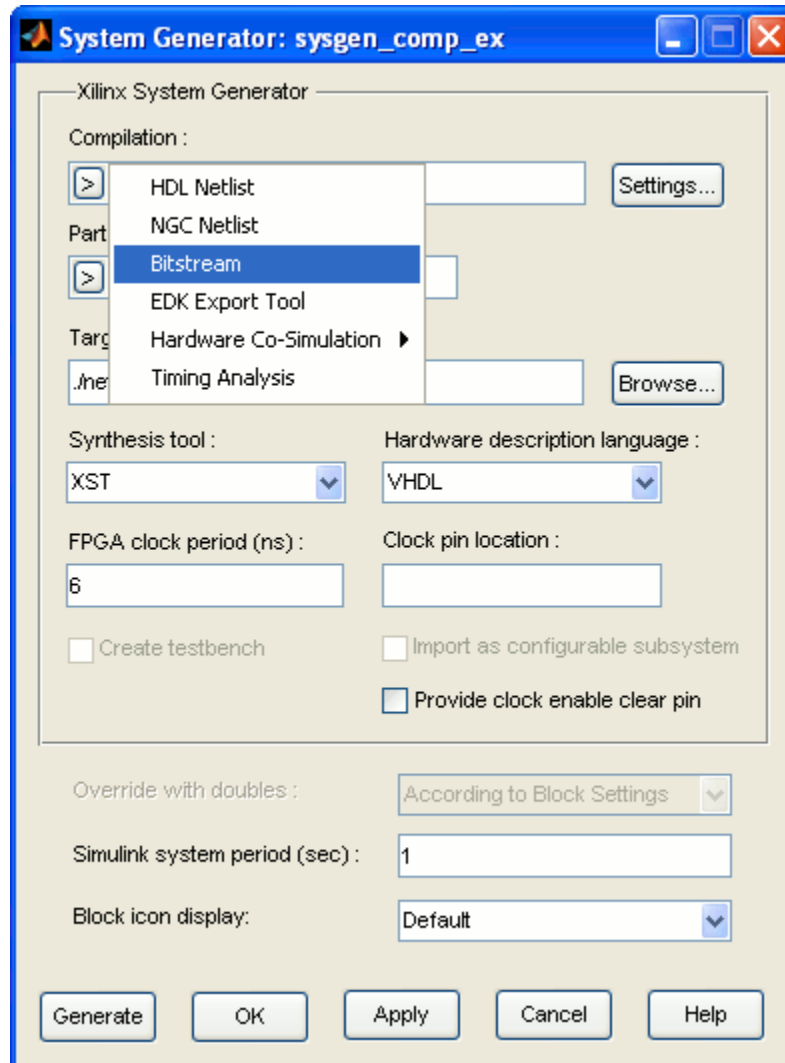


Figure 1: Selecting Bitstream compilation for a design.

System Generator uses XFLOW to run the tools necessary to produce the configuration bitstream.

Execution of XFLOW is broken into two flows, *implementation* and *configuration*.

The implementation flow is responsible for compiling the synthesis tool netlist output (e.g., EDIF or NGC) into a placed and routed NCD file. In summary, the implementation flow performs the following tasks:

- Combines synthesis results, core netlists, black box netlists, and constraints files using NGDDBuild.
- Runs MAP, PAR, and Trace on the design (in that particular order).

The configuration flow type runs the tools (e.g., BitGen) necessary to create an FPGA bitfile, using the fully elaborated NCD file as input.

The implementation and configuration flow types have separate XFLOW options files associated with them. An XFLOW options file declares the programs that should be run for a particular flow, and defines the command line options that are used by these tools. The Xilinx ISE software includes several example XFLOW options files. From the base directory of your Xilinx ISE software tree (e.g., c:\Xilinx\), these files can be located under xilinx\data directory (e.g., c:\Xilinx\xilinx\data). Three commonly used implementation options files include:

1. balanced.opt;
2. fast_runtime.opt;
3. high_effort.opt.

Note: By default, System Generator uses the **balanced.opt** file for the implementation flow, and **bitgen.opt** file for the configuration flow.

Sometimes you may want to use options files that use settings that differ (e.g., to specify a higher placer effort level in PAR) from the default options provided by the target. In this case, you may create your own options files, or edit the default options files to include your desired settings. The Bitstream settings dialog box allows you to specify options files other than the default files.

You may access additional compilation settings specific to Bitstream compilation by clicking on the **Settings** button when **Bitstream** is selected as the compilation type in the System Generator block dialog box. The dialog box containing these settings is shown in Figure 2.

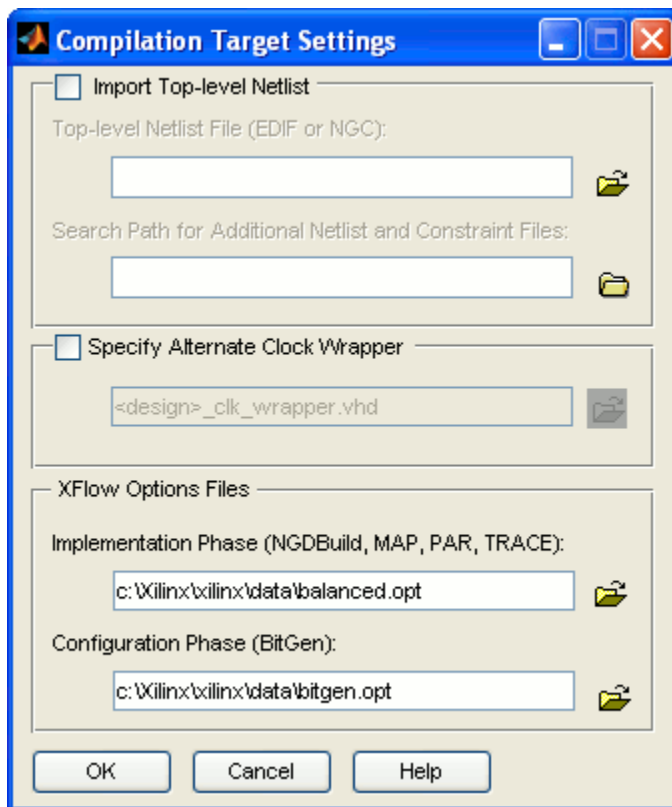


Figure 2: Bitstream Settings dialog box.

Parameters specific to the Bitstream Settings dialog box include:

- **Import Top-level Netlist:** Selecting this checkbox allows you to specify your own top-level netlist into which the System Generator portion of the design is included as module. You may choose to import your own top-level netlist if you have a larger design that instantiates the System Generator clock wrapper level as a component. Refer to the Compilation Results section in the User Guide for more information on the clock wrapper level. This top-level netlist is included in the bitstream file that is generated during compilation. Selecting this checkbox enables the edit fields Top-level Netlist File (EDIF or NGC) and Search Path for Additional Netlist and Constraint Files.
 - **Top-level Netlist File (EDIF or NGC) :** Specifies the name and location of the top-level netlist file to include during compilation. Note that any HDL components that are used by your top-level (including the top-level itself) must have been previously synthesized into netlist files.
 - **Search Path for Additional Netlist and Constraint Files :** Specifies the directory where System Generator should look for additional netlist and constraint files that go along with the top-level netlist file. System Generator copies all netlist (e.g., .edn, .edf, .ngc) and constraints files (e.g., .ucf, .xcf, .ncf) into the implementation directory when this directory is specified. If you do not specify a directory, System Generator will only copy the netlist file specified in the Top-level Netlist File field.
- **Specify Alternate Clock Wrapper :** Allows you to substitute your own clock wrapper logic in place of the clock wrapper HDL System Generator produces. The clock wrapper level is the top-level HDL file that is created for a System Generator design, and is responsible for driving the clock and clock enable signals in that design. Sometimes you may want to supply your own clock wrapper, for example, if your design uses multiple clock signals, or if you have a board-specific hardware you would like your design to interface to. **Note that the name of the**

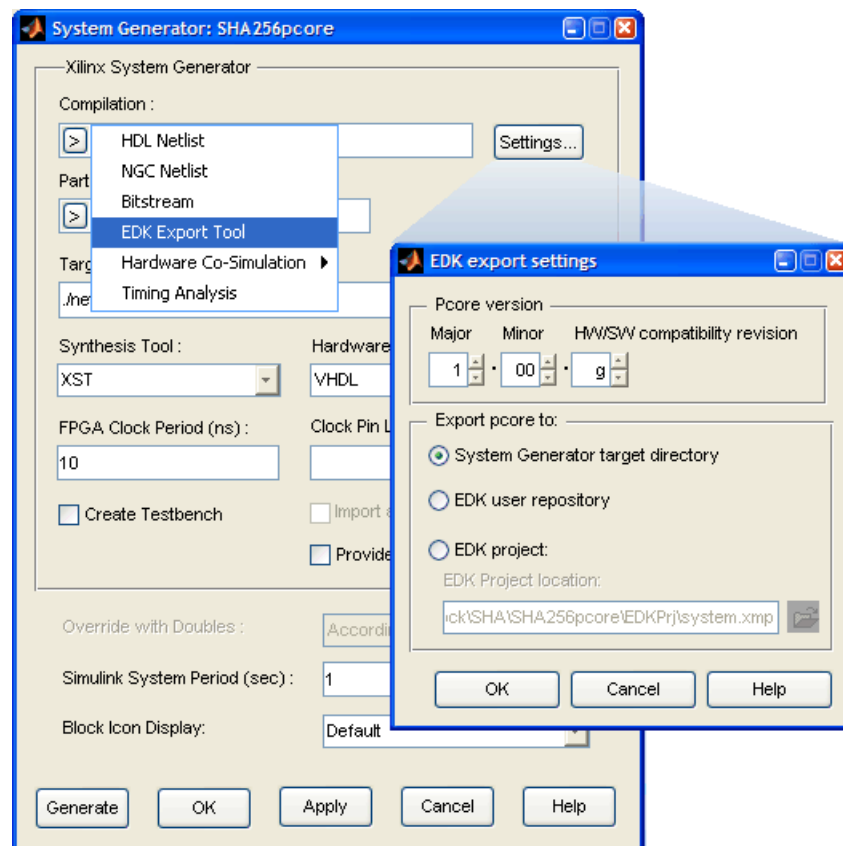
alternate clock wrapper file must be named <design>_cw.vhd or <design>_cw.v or it will not be used during bitstream generation.

- **XFLOW Option Files** : When a design is compiled for System Generator hardware co-simulation, the command line tool, XFLOW, is used to implement and configure your design for the selected FPGA platform. XFLOW defines various “flows” that determine the sequence of programs that should be run on your design during compilation. There are typically multiple flows that must be run in order to achieve the desired output results, which in the case of hardware co-simulation targets, is a configuration bitstream.
 - **Implementation Phase (NBDBuild, MAP, PAR, TRACE)**: Specifies the options file that is used by the implement flow type. By default, System Generator will use the implement options file that is specified by the compilation target.
 - **Configuration Phase (BitGen)** : Specifies the options file that is used by the configuration flow type. By default, System Generator will use the configuration options file that is specified by the compilation target.

EDK Export Tool

The EDK Export Tool allows a System Generator design to be exported to a Xilinx Embedded Development Kit (EDK) project. The EDK Export Tool simplifies the process of creating a peripheral by automatically generating the files required by the EDK.

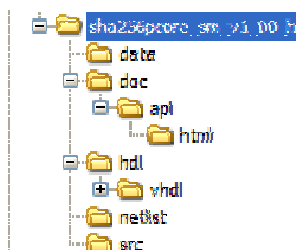
The EDK Export Tool can be accessed from the System Generator block GUI under the Compilation pull-down menu – the figure below shows this being done. After the EDK Export Tool is selected, the ‘Settings...’ button will be enabled.



Clicking on the ‘Settings...’ button will bring up the EDK export settings dialog. This allows you to assign a version number to your pcore, and also instruct System Generator where to export the pcore to.

Export as FSL peripheral to EDK

When a System Generator design is exported to the EDK, the name of the peripheral has the postfix “_sm” appended to the model name. For instance, when a model called mul_accumulate is exported to the EDK, it will be called mul_accumulate_sm on the EDK side.



The figure above shows an example of the directory structure generated by System Generator with pcpre export. Its content is detailed in the following table.

Directory	Description
data	The data directory contains four files: BBD, PAO, MPD and TCL. The BBD (black-box definition) file tells the EDK what EDN or NGC files are used in the design. The PAO (peripheral analyze order) file tells the EDK the analyze order of the HDL files. The MPD (Microprocessor Peripheral Description) file tells the EDK how the peripheral will connect to the processor. The TCL file is used by LibGen when elaborating software drivers for this peripheral.
doc	Documentation files in HTML format.
hdl	The hdl directory contains the hdl files produced by System Generator.
netlist	The netlist directory contains the EDN and NGC files listed by the BBD file.
src	Source files for the software drivers.

System Generator ports as top level ports in EDK

Input and output ports created in System Generator will be made available to the EDK tool as ports on the peripheral. The user may pull these ports to the top-level of the EDK design. This is useful for instance when the System Generator design has ports that go to the input/output pads on the FPGA device.

Supported processors and current limitations

Currently only FSL links to the MicroBlaze processor can be exported with the EDK Export Tool. There can only be one instance of an EDK Processor block.

See Also:

EDK Processor block

Hardware Co-Simulation Compilation

System Generator can compile designs into FPGA hardware that can be used in the loop with Simulink simulations. This capability is discussed in the Using FPGA Hardware in the Loop chapter of the User Guide.

You may select a hardware co-simulation target by left-clicking the **Compilation** submenu control on the System Generator dialog box, and selecting the desired hardware co-simulation platform (Figure 1). Note that the list of available co-simulation platforms depends on which hardware co-simulation plugins are installed on your system.

Note: If you have an FPGA platform that is not listed as a compilation target, you may create a new System Generator compilation target that uses JTAG to communicate with the FPGA hardware. Refer to the Supporting New Platforms section of the User Guide for more information on how to do this.

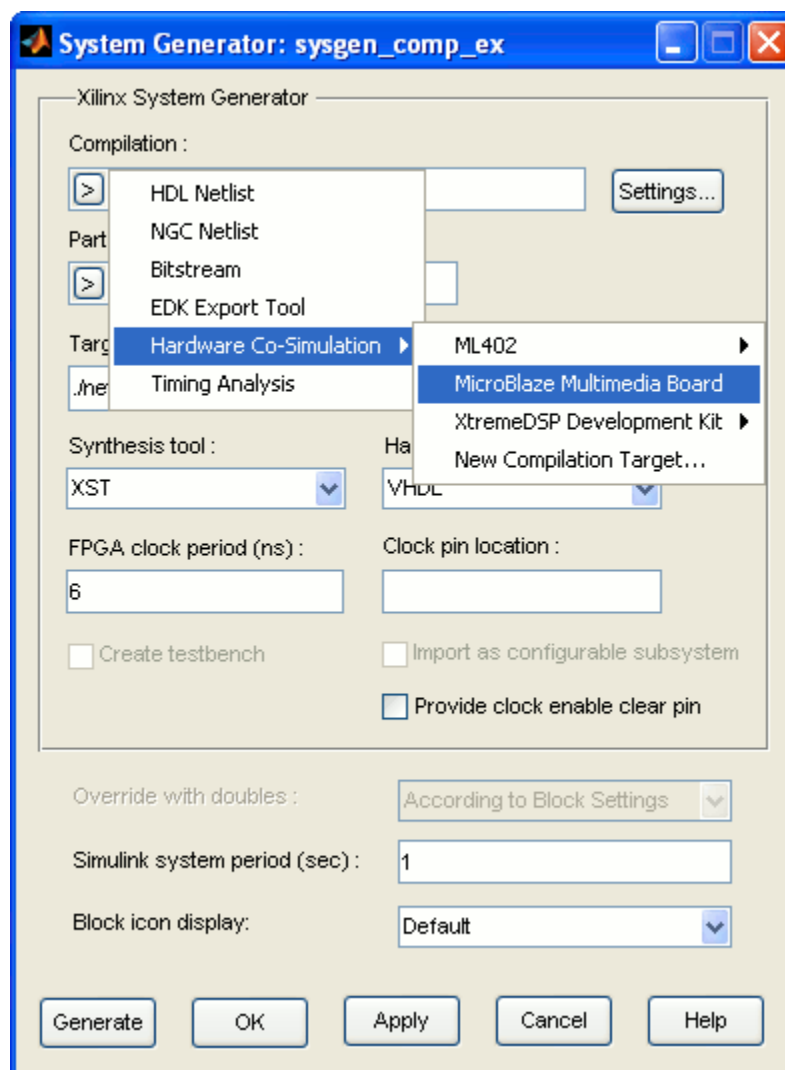


Figure 1: Selecting a hardware co-simulation compilation target.

Creating Compilation Targets

The HDL and netlist files that System Generator produces when it compiles a design into hardware must be run through additional tools in order to produce a configuration bitstream file that is suitable for your FPGA. A typical flow that allows you to generate an FPGA configuration file is ProjectNavigator. There are other ways in which a bitstream can be generated for your model. For example, it is possible to configure System Generator to automatically run the tools necessary to produce a configuration file when it compiles a design. This is advantageous since the complete bitstream generation process is accomplished inside the tool. Moreover, you can have System Generator run different tools (e.g., ChipScope Pro Analyzer and iMPACT) once the configuration file is generated for a model.

The way in which System Generator compiles a model into hardware depends on the compilation target that is chosen for the design. The HDL Netlist compilation target is most common, and generates an HDL netlist of your design plus any cores that go along with it. New compilation targets can be created that extend the HDL Netlist target so that additional tools can be applied to the resulting HDL netlist files.

This section explains how you can create new compilation targets that extend the HDL Netlist target in order to produce and configure FPGA hardware. More specifically, it describes how to configure System Generator to produce a bitstream for a model, and how to invoke various tools once the bitstream is created.

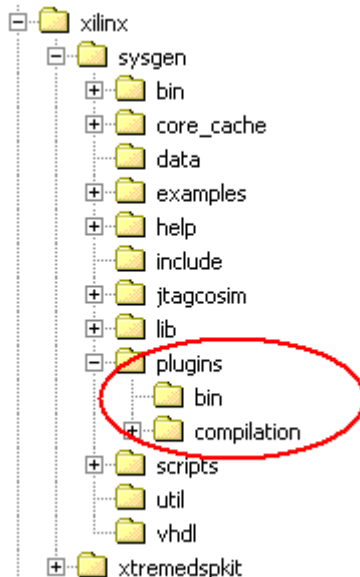
Defining New Compilation Targets

You can create new compilation targets to run tools that process the output files associated with HDL Netlist compilation. A compilation target is defined by a minimum of two MATLAB functions. The first function, `xltarget.m`, tells System Generator to support the target (i.e., make it selectable from the System Generator block dialog box), and specifies the MATLAB function where more information about the target can be found. This function is called a “target info” function. A target info function defines information about the target, and can take any name, provided it is specified correctly in the target’s `xltarget.m` function. In some cases, a target info function defines a post-generation function. A post-generation function is responsible for invoking tools or scripts after normal HDL netlist compilation is complete. These functions are discussed in more detail below.

The `xltarget` Function

An `xltarget` function specifies one or more compilation targets that should be supported by System Generator. It also provides entry points through which System Generator can find out more information about these targets.

Note: System Generator determines which compilation targets to support by searching the plugins/compilation (and its subdirectories) of your System Generator software install tree for `xltarget.m` files.



Although an `xltarget` function can specify multiple targets, it is not uncommon for each compilation target to have its own `xltarget` function. The directories these functions are saved in distinguish the targets. This means that each `xltarget.m` file must be saved in its own subdirectory under the `plugins/compilation` directory.

An `xltarget` function returns a cell array of target information. Different elements in this cell array define different compilation targets. The elements in this cell array are MATLAB structs that define two parameters:

1. The name of the compilation target as it should appear in the Compilation field of the System Generator parameters dialog box;
2. The name of the MATLAB function it should invoke to find out more information (e.g., System Generator dialog box parameters, which post-generation function to use, if any) about the target.

The following code shows how to define three compilation targets named Standalone Bitstream, iMPACT, and ChipScope Pro Analyzer:

```
function s = xltarget

s = {};

target_1('name') = 'Standalone Bitstream';
target_1('target_info') = 'xltools_target';

target_2('name') = 'iMPACT';
target_2('target_info') = 'xltools_target';

target_3('name') = 'ChipScope Pro Analyzer';
target_3('target_info') = 'xltools_target';

s = {target_1, target_2, target_3};
```


The name field in the code shown above specifies the name of the compilation target, as it should appear in the Compilation field of the System Generator dialog box:

```
target_1('name') = 'Standalone Bitstream';
```

The target_info field tells System Generator the target info function it should call to find out more information about the target. This function can have any name provided it is saved in the same directory as the corresponding xltarget.m file, or it is saved somewhere in the MATLAB path.

```
target_1('target_info') = 'xltools_target';
```

Note: An example xltarget function is included in the examples/comp_targets directory of your System Generator install tree. You can modify this function to define your own bitstream-related compilation targets.

Target Info Functions

A target info function (specified by the target_info field in the code above) is responsible for two things:

- It defines the available and default settings for the target in the System Generator block dialog box;
- It specifies the functions System Generator should call before and after the standard code generation process.

Note: An example target info function, xltools_target.m, is included in the examples/comp_targets directory of your System Generator install tree.

One such function that is particularly useful to compilation targets is the post-generation function. A post-generation function is run after standard code generation. The code below shows how a post-generation function is specified in a target info function:

```
settings('postgeneration_fcn') =  
'xltools_postgeneration';
```

Post-generation Functions

One way to extend System Generator compilation is by defining a new variety of compilation that specifies a *post-generation function*. A post-generation function is a MATLAB function that tells System Generator how to process the HDL and netlist files once they are generated. This function is run after System Generator finishes the normal code generation steps involved with HDL Netlist compilation (i.e., producing an HDL description of the design, running CORE Generator, etc). For example, a hardware co-simulation target defines a post-generation function that in turn runs the tools necessary to produce hardware that can be used in the Simulink simulation loop.

Note: Two post-generation functions xlBitstreamPostGeneration.m and xltools_postgeneration.m, are included in the examples/comp_targets directory of your System Generator install tree.

xlBitstreamPostGeneration.m

This example post-generation function compiles your model into a configuration bitstream that is appropriate for the settings (e.g., FPGA part, clock frequency, clock pin location) given in the System Generator dialog box of your design. This function invokes the Synthesis tool specified in the System

Generator dialog box to synthesize your design.

Note: Results for model synthesis are stored in a `synth_model` directory under the model's target directory.

It then uses an XFLOW-based flow to invoke the Xilinx tools necessary to produce an FPGA configuration bitstream.

Note: Results from XFLOW are stored in an `xflow` directory under the model's target directory.

It is possible to configure the tools and configurations for each tool invoked by XFLOW. For more information on how to do this, refer to the section in this example entitled "Using XFLOW."

`xltools_postgeneration.m`

Sometimes you may want to run tools that configure and run the FPGA after a configuration bitstream has been generated (e.g., iMPACT, ChipScope Pro Analyzer). The `xltools_postgeneration` function first calls the `xlBitstreamGeneration` function to generate the bitstream. It then invokes the appropriate tool (or tools) depending on the compilation target that is selected.

For example, you may want a compilation target that invokes iMPACT after the bitstream is generated. This can be done as follows (assuming iMPACT is in your system path):

```
if (strcmp(params.compilation, 'iMPACT'))
    dos('impact');
end;
```

The first line checks the name of the compilation target. The second line sets up a DOS command that invokes iMPACT. ChipScope Pro Analyzer can be invoked similarly to the code above:

```
if (strcmp(params.compilation, 'ChipScope Pro
Analyzer'))
    xlCallChipScopeAnalyzer;
end;
```

Note: `xlCallChipScopeAnalyzer` is a MATLAB function provided by System Generator to invoke ChipScope.

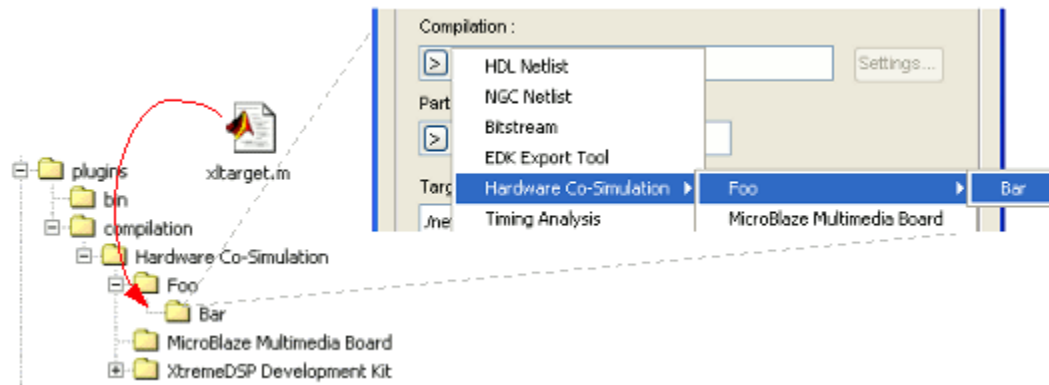
Configuring and Installing the Compilation Target

Listed below are the steps necessary to configure and install new bitstream compilation targets.

1. Copy the `xltarget.m`, `xltools_postgeneration.m`, and `xltools_target.m` files from `examples/comp_targets` into a temporary directory.
2. Change the permissions of the above files so they can be modified.
3. Add the desired compilation targets (e.g., iMPACT, ChipScope Analyzer Pro) to the `xltarget.m` file.
4. Add the desired tool invocations to the `xltools_postgeneration.m` file.
5. Create a new directory (e.g., Bitstream) under the `plugins/compilation` directory of your System Generator software install tree. Copy the `xltarget.m`,

`xltools_postgeneration.m`, and `xltools_target.m` files into this directory.

Note: The System Generator *Compilation* submenus mirror the directory structure under the `plugins/compilation` directory. When you create a new directory, or directory hierarchy, for the compilation target files, the names of the directories define the taxonomy of the compilation target submenus.



6. Copy the `xlBitstreamPostGeneration.m`, `xlToolsMakebit.pl`, `balanced_xltools.opt` and `bitgen_xltools.opt` files from the `examples/comp_targets` directory into a directory that is in your MATLAB path. These files must be in a common directory.
7. In the MATLAB command window, type the following:


```
>> rehash toolboxcache
>> xlrehash_xltarget_cache
```
8. You can now access the newly installed compilation target from the System Generator graphical interface.

Using XFLOW

The post-generation scripting included with this example uses XFLOW to produce a configuration file for your FPGA. XFLOW allows you to automate the process of design synthesis, implementation, and simulation using a command line interface. XFLOW uses command files to tell it which tools to run, and how they should be run.

More information on XFLOW can be found here:

<http://toolbox.xilinx.com/docsan/xilinx4/data/docs/dev/xflow2.html>

This example contains two XFLOW options files, `balanced_xltools.opt` and `bitgen_xltools.opt`. These files are associated with the implementation and configuration flows of XFLOW, respectively. The `balanced_xltools.opt` options files runs the Xilinx NGDBUILD, MAP, and PAR tools. The settings for each of tool are specified in the options files. The `bitgen_xltools.opt` file runs BITGEN to produce a configuration file for your FPGA. You may modify these files as desired (e.g., to run the timing analyzer after PAR).

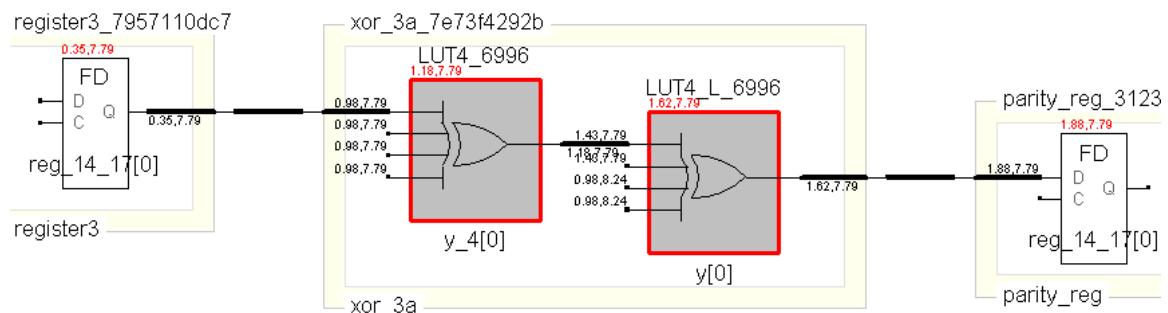
Timing Analysis Tool

Sometimes the hardware created by System Generator may not meet the requested timing requirements. System Generator provides a Timing Analysis tool that can help you resolve timing related issues. The timing analysis tool shows the user; both in graphical and textual formats, the slowest system paths and those paths which are failing to meet the timing requirements. This allows the user to concentrate on methods of speeding up those paths. Methods for doing so will be discussed.

Underlying the System Generator Timing Analysis tool is Trace, a software application delivered as part of the ISE tool suite used to analyze timing paths.

Period and Slack

This brief section is intended for those with little or no knowledge of logic path analysis. A timing failure usually means there is a setup time violation in the design. A setup time violation means that a particular signal cannot get from the output of one synchronous element to the input of another synchronous element within the requested clock period and subject to the second synchronous element's setup time requirement. A typical path is shown in this Synplify schematic:



The path shown is from the Q output of the register on the left (*register3*) to the D input of the register on the right (*parity_reg*). The path goes through two LUTs (lookup tables) that are configured as 4-input XOR gates. This path has two levels of logic. That means that it goes through two separate combinational elements (the two LUTs).

The requested period for this path is 10ns. This path easily meets timing. The second of the two red comma-separated numbers above each logic elements shows the *slack* for the path. The slack is the amount of time by which the path 'meets timing'. In this case the slack is 7.79ns. That means that the path could be 7.79ns slower and still meet the 10ns period requirement. A negative slack value indicates that the path does not meet timing and has a setup (or hold) time violation.

Example Path Analysis

Let us examine this path in more detail. The first value on the top of *register3* is 0.35ns. This means that the clk→out time of the register is 0.35ns, so the data will appear on the Q output 0.35ns after the rising edge of the clock signal. (The clock signal, not shown, drives the C inputs of both registers.)

The input of the LUT *y_4[0]* shows two numbers on each input. The first is the arrival time of the signal. This value is 0.98ns. This means that the signal arrives at the input 0.98ns after the rising edge of the clock. Therefore the *net delay* is (0.98ns-0.35ns)=0.63ns. Any path delay is divided into *net delays* and *logic delays*. In an FPGA, the net delays are normally the predominant type of delay. This is because the configurable routing fabric of the FPGA requires that a net traverse many delay-inducing switchboxes in order to reach its destination.

The path leaves `y_4[0]` and travels along another net to `y[0]`. The first of the two values at the output of `y[0]` shows the arrival time of the signal at the output of that LUT. This value is 1.62ns. The signal travels along the final net, incurring a net delay of 0.26ns to arrive at the D input of `parity_reg` at 1.88ns after the clock edge. This register has a required *setup time*. The setup time for this register is 0.33ns. This means that the signal must arrive at the D input 0.33ns before the rising edge of the next clock. Therefore the total path requires (1.88ns+0.33ns)=2.21ns. Subtracted from 10ns, this yields the 7.79ns slack value.

Clock Skew and Jitter

The net delay values shown here are estimates provided by Synplify. The synthesizer doesn't know the actual net delay values because these are not determined until after the place & route process. An actual path contains other variables which must be accounted for, including clock skew and clock jitter. *Clock skew* is the amount of time between clock arrival at the source and destination synchronous elements. *Clock jitter* is a variation of the clock period from cycle to cycle. Jitter is created by the DCMs (digital clock managers) and by other means. The timing analysis is carried out with worst-case values for the given part's delay values, jitter, skew, and temperature derating.

Using the Timing Analyzer

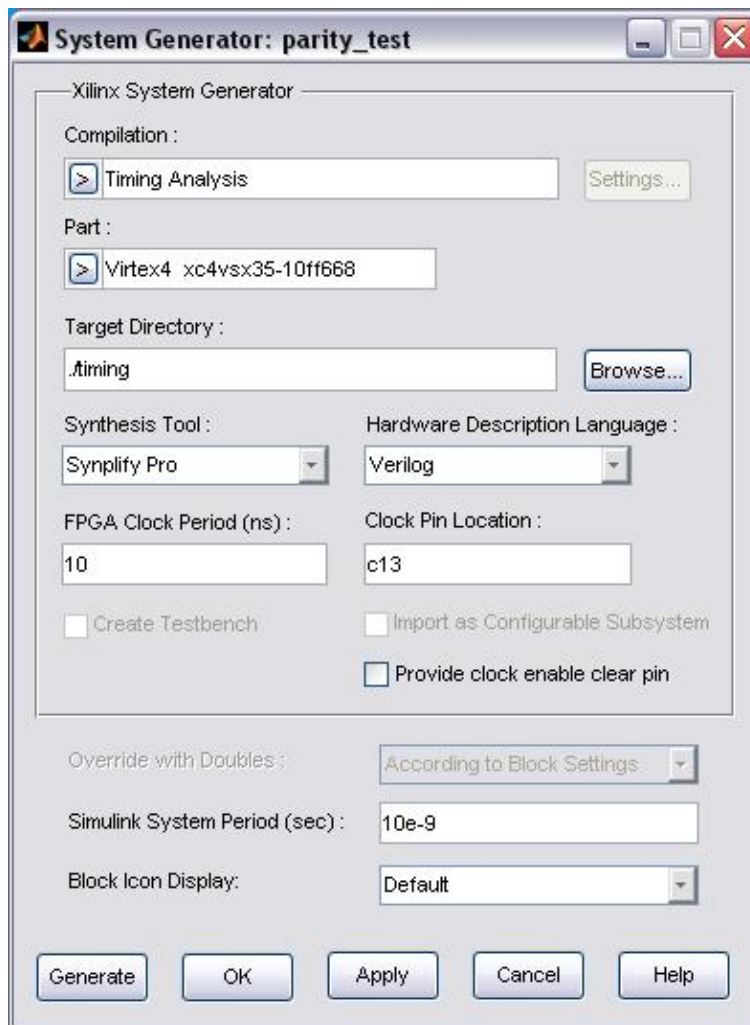
Starting the Timing Analyzer GUI

The timing analyzer GUI may be opened by one of two ways.

Selecting Timing Analysis as the Compilation Target

The most typical fashion will be to choose the timing analyzer as the target when generating the

design. To do this with your design, double-click on the System Generator block  to open the System Generator dialog box:



To use the tool, select the *Timing Analysis* option from the *Compilation* submenu. The part is selected as desired. It is important to choose the exact part you wish to target as the size and speed of the part will affect the path delays. Result files shall be put in the target directory. The value in the *FPGA Clock Period* box is the value that will be used during place & route.

After filling out the box and clicking on the *Generate* button, System Generator will perform the following steps:

1. The design will be compiled using Simulink.
2. The design will be netlisted by System Generator into HDL source.
3. The synthesizer will be called to turn the HDL into an EDIF (Synplify) or NGC (XST) netlist.
4. NGD Build will be called to turn the netlist into an NGD netlist file.
5. The ISE Mapper software will be called to map elements of logic together into slices; this creates an NCD file.
6. The ISE Place & Route software will be called to place the slices and other elements on the Xilinx die and to route the connections between the slices. This creates another NCD file.
7. The ISE Trace software will be called to analyze the NCD and to find the paths with the worst slack. This creates a trace report.
8. The System Generator Timing Analyzer tool appears, displaying the data from the trace report.

Opening the Timing Analyzer from the Command Line

The other method of opening the timing analyzer is used when you wish to view previously-generated data. If timing data was generated using the previously-described method, then it can be viewed at a later time using the timing analyzer. To do this, enter the following command at the Matlab command line:

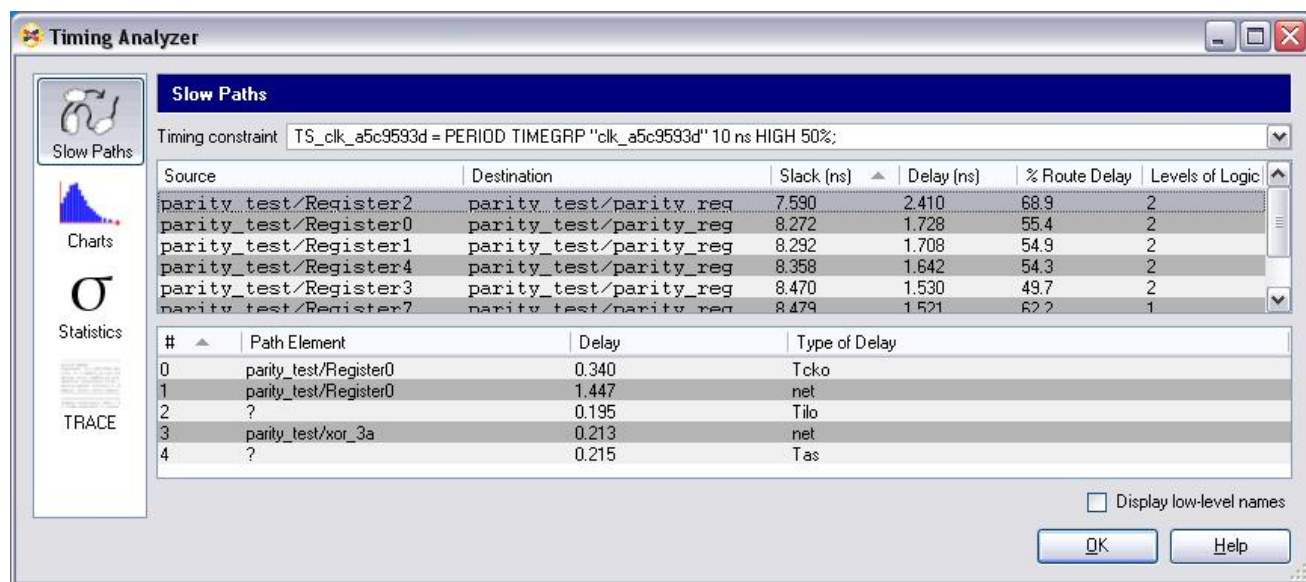
```
>>xlTimingAnalysis('timing')
```

where 'timing' is the name of the target directory in which a prior analysis was carried out.

Observing the Slow Paths



Clicking on the  icon displays the paths with the least slack for each timing constraint. An example is shown below:



The top section of the display shows a list of slow paths, while the bottom section of the display shows details of the path that is selected. The elements of this display are explained here:

- **Timing Constraint:** The user may opt to view the paths from all timing constraints or just a single constraint. A typical System Generator design has but a single timing constraint which defines the period of the system clock. This is the constraint shown in this example. `TS_clk_a5c9593d` is the name of the constraint; the (sometimes confusing) suffix is a hash meant to make the identifier unique when multiple System Generator designs are used as components inside a larger design. The timing group `clk_a5c9593` is a group of synchronous logic, again with a hash suffix. The group in this case contains all the synchronous elements in the design. The period of the clock here is 10ns with a 50% duty cycle.
- **Source:** The System Generator block that drives the path.
- **Destination:** This is the System Generator block that is the terminus of the path.
- **Slack:** The slack for this particular path. See the section entitled *Period and Slack* for more details.
- **Delay (Path):** The delay of the entire path, including the setup time requirement.
- **% Route Delay:** This is the percentage of the path that is consumed by routing (net) delay. The remainder portion of the path is consumed by logic delay.
- **Levels of Logic:** The number of levels of combinatorial logic in the path. The combinatorial logic typically comprises LUTs, F5 muxes, and carry chain muxes.
- **Path Element:** This shows the logic and net elements in the highlighted path.

- **Delay (Element):** This shows the delay through the logic and net elements in the highlighted path.
- **Type of Delay:** This is the kind of delay incurred by the given path element. These values are defined in the Xilinx part's datasheet. In the example shown above, *Tcko* is the clk→out time of a flip-flop; *net* is a net delay; *Tilo* is the delay through a LUT, and *Tas* is the setup time of a flip-flop.

You may click on the column headings to reorder the paths or elements according to delay, slack, path name, or other column headings. Failing paths are highlighted in red/pink.

Name Unmunging and Displaying Low-Level Names

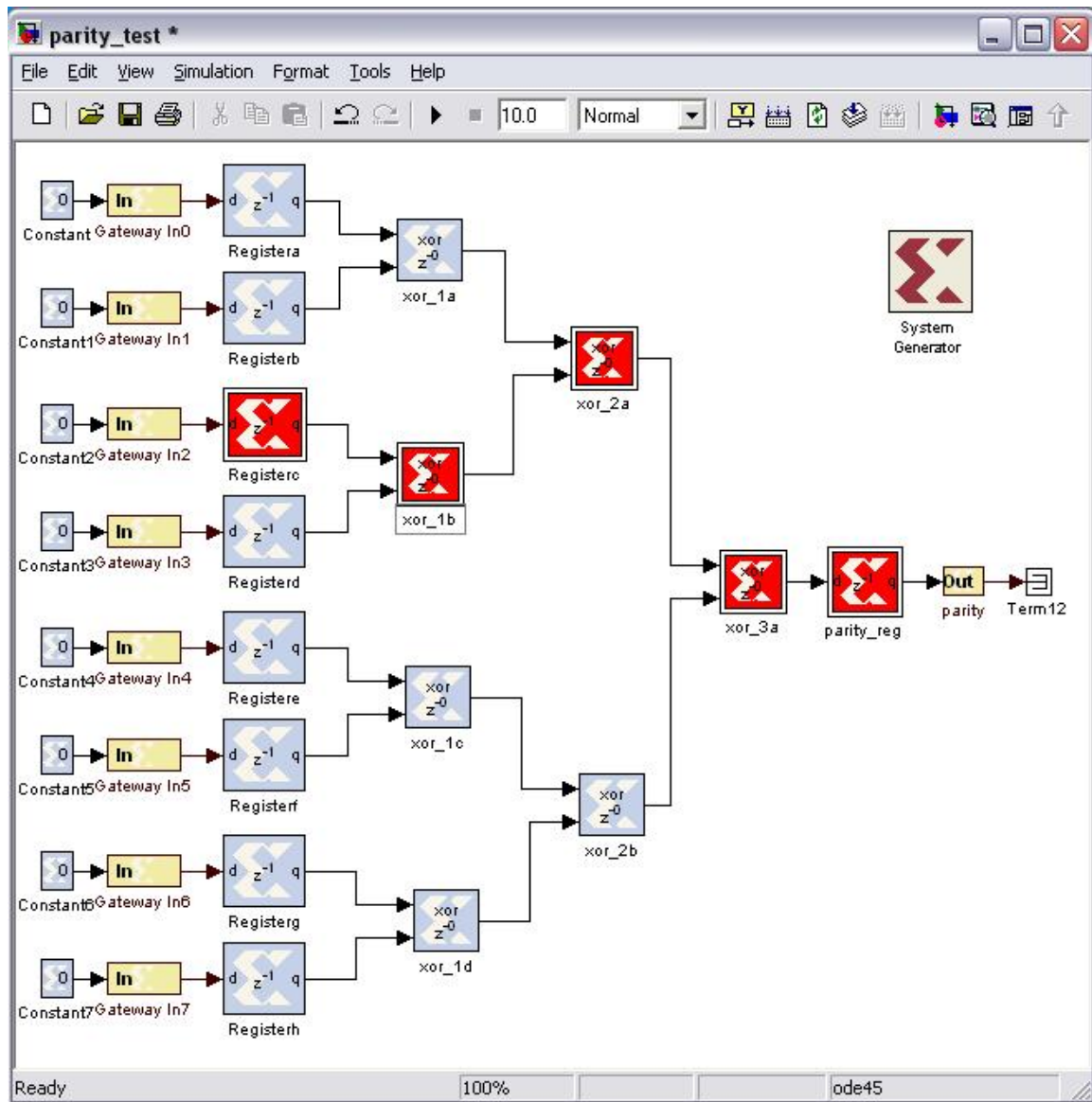
Part of the magic of the timing analyzer lies in its ability to perform the unglorious task of *name unmunging*, the task of automatically correlating System Generator components with the low-level component names produced by the Xilinx implementation tools. The names of these components often differ considerably. In fact, the logic blocks and wires that appear in a System Generator diagram may have only a loose relation to the actual logic that gets generated during the synthesis process. The System Generator timing analyzer must correlate the names of logic elements and nets in the trace report to blocks and wires in the System Generator diagram.

The timing analyzer cannot always perform this unmunging process. In the path shown in the screen capture above, path elements #2 and #5 have a question mark displayed in the name field. This means that the timing analyzer could not unmunge the name from the trace report and correlate it to a System Generator block.

To see the actual names from the trace report, check the *Display low-level names* box. This will show the trace report names. You may be able to correlate them to System Generator elements by observation.

Cross-Probing

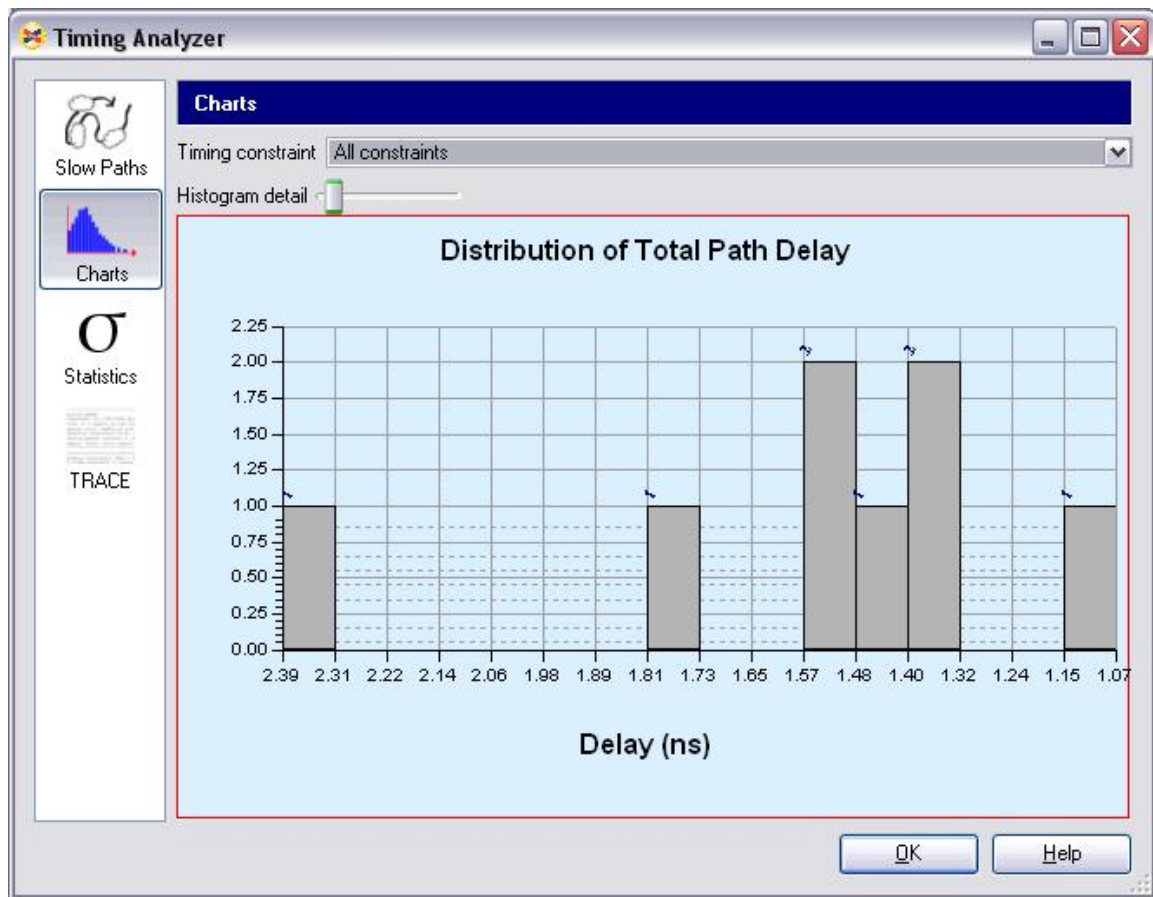
Highlighting a path in the *Slow Paths* view will highlight the blocks in the path in the System Generator diagram. The path's source and destination blocks, as well as combinational blocks through which the path passes, will be highlighted in red. The diagram below shows how the model appears when the path that has *Registerc* as its source and *parity_reg* as its destination is highlighted. The blocks *xor_1b*, *xor_2a*, and *xor_3a* are also highlighted because they are part of the path.



Histogram Charts



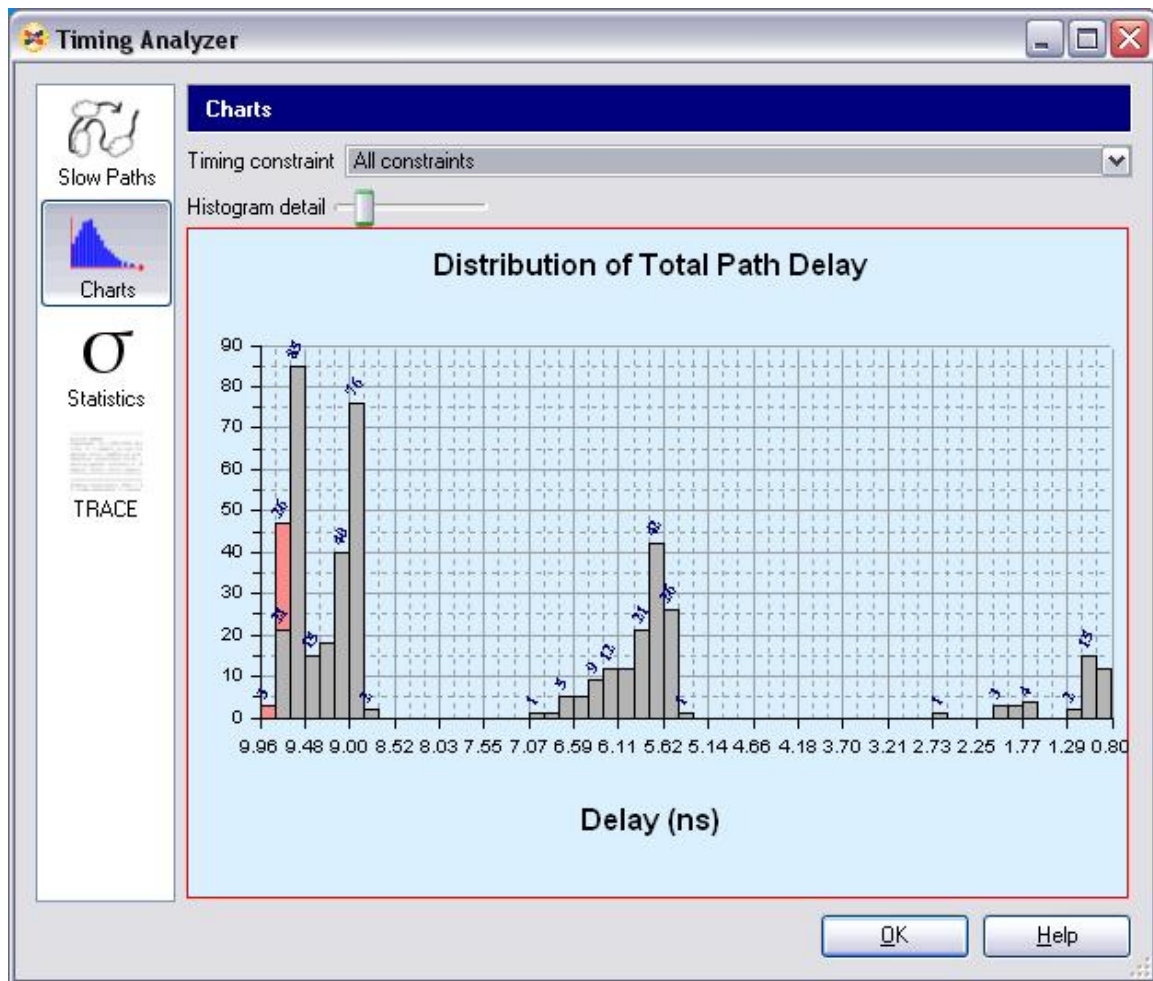
Clicking on the  icon displays a histogram of the slow paths. This histogram is a useful metric in analyzing the design. You may know that the design will only run at, for example, 99MHz in your part when you wish it to run at 100MHz. But how close is the design to meeting timing and how much work is involved in meeting this requirement? The histogram will quickly give you an estimate of the work involved. For example, look at the histogram of the results of a simple design below:



This shows that most of the slow paths are concentrated about 1.5ns. The slowest path is about 2.35ns. The numbers at the tops of the bins show the number of paths in each bin. There is only one path in the bin which encompasses the time range 2.31ns-2.39ns. The bins to the right of it are empty. This shows that the slowest path is an outlier and that if your timing requirement were for a period of, for example, 2ns, you would need only to speed up this single path to meet your timing requirements.

Histogram Detail

The slider bar allows you to adjust the width of the bins in the histogram. This allows you to get more detail about the paths if desired. The display below shows the results of a different design with a larger number of bins than the diagram above:



This diagram shows the paths grouped into three regions, with each forming a rough bell curve distribution. These groups are probably from different portions of the circuit or from different timing constraints that are from different clock regions. If you wish to analyze the paths from a single timing constraint, you may select a single constraint for viewing from the *Timing constraint* pulldown menu at the top of the display.

Note the bins and portions thereof shown in red. These are the paths that have negative slack; i.e., they do not meet the timing constraint. In this example you can see that some paths have failed but not by a large margin so it seems reasonable that with some work this design could be reworked to meet timing.

Statistics

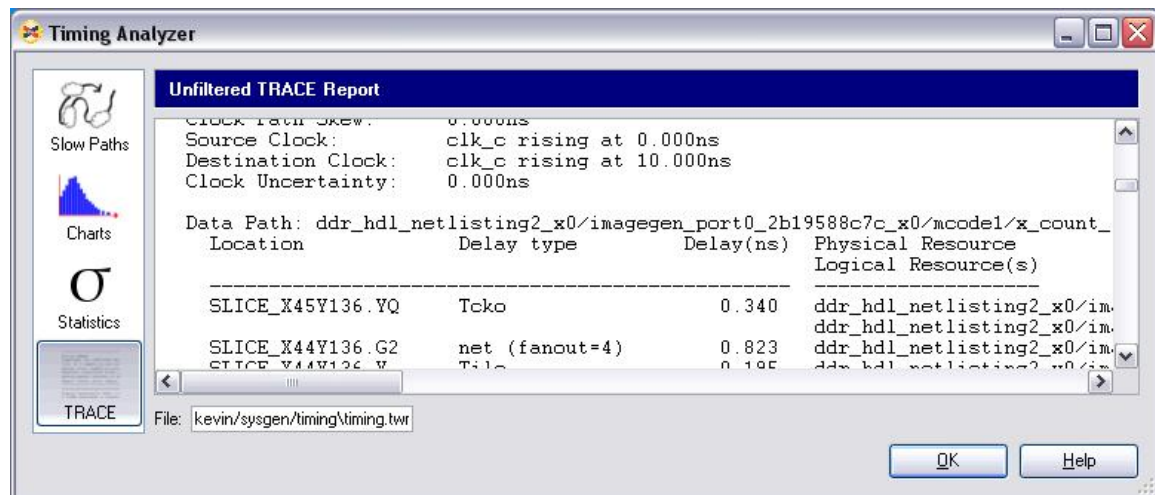


Clicking on the  icon displays several design statistics, including the number of constraints, paths analyzed, and maximum frequency of the design.

Trace Report



Clicking on the **TRACE** icon shows the raw text report from the Trace program. This file gives considerable detail about the paths analyzed. Each path analyzed contains information about every net and logic delay, clock skew, and clock uncertainty. The box at the bottom left of this display shows the pathname of the timing report.



Improving Failing Paths

“Now I have information about my failing paths; but what do I do now?” you may ask yourself. This is the trick for which there is no simple answer, and this is where you may need to delve into the lower-level aspects of FPGA design.

In general, steps that may be taken to meet timing are, in this order:

1. **Change the source design.** Just about any timing problem can be solved by changing the source design and this is the easiest way to speed up the circuit. Unfortunately, this is often the last step taken by designers, who often look for a quick solution such as using a faster part. The source design may be changed in several ways:
 - a. **Pipelining.** This is the surest way to improve speed, but may also be tricky. Adding pipelining registers increases latency. For designs with feedback, this may require great care since portions of the design may require pipeline rebalancing. See the later example for more details on pipelining.
 - b. **Parallelization.** This is probably the second most-important improvement you can make. Do you have a FIR filter that won't operate at the correct speed? You can use two FIR filters in parallel, each operating at half-rate, and interleave the outputs. This is the classic *speed/area tradeoff*.
 - c. **Retiming.** This involves taking existing registers and moving them to different points within the combinational logic to rob from Peter to pay Paul, so to speak. This works if, to stretch the maxim, Paul is bereft of slack, while Peter has a surfeit. Some synthesis tools can perform a degree of retiming automatically.
 - d. **Replication.** Replication of registers or buffers increases the amount of logic but reduces the fanout on the replicated objects. This decreases the capacitance of the net and reduces net delay. The replicated registers may also be *floorplanned* to place them closer to the logic groups they drive. Replication is often performed automatically by the tools

- and manual replication is not a common practice in a high-level design environment like System Generator.
- e. **Shannon Expansion.** This method involves replicating the faster logic in a critical path in order to remove dependencies on slower logic. This is sometimes done automatically by the synthesizer.
 - f. **Using Hard Cores.** Are you using a ROM that is implemented in distributed RAM when it would operate much faster in a block memory hard core? Do you have a wide adder that would benefit from being put in a DSP48 block, which can operate at 500MHz? Take advantage of the embedded hard cores.
 - g. **New Paradigms.** Do you need to create a large delay? Instead of using a counter with a long carry chain, why not build a delay out of cascaded Johnson rings using SRL16s? Or how about using an LFSR? Neither requires a carry chain and can operate much faster. Sometimes you have to rethink certain design elements completely.
2. **Eliminate overconstraints.** Ensure that elements of your design that only need to be operated at a subsampled rate are designed that way by using the downsample and upsample blocks in System Generator. If these blocks are not used, then the timing analyzer is not aware that these sections of the circuit are subsampled, and the design is overconstrained.
 3. **Change the constraints.** Is it possible to run the design at a lower clock speed? If so, this is an easy way to meet your requirements. Unfortunately, this is rarely possible due to design requirements.
 4. **Increase PAR effort levels.** The mapper and place & route tools (PAR) in ISE take effort levels as arguments. When using ISE (from the Project Navigator GUI), try the *-timing* option in MAP. You may also increase the PAR effort levels which will increase the PAR execution time but may also result in a faster design.
 5. **Multipass PAR.** PAR is an iterative process and is somewhat chaotic in that the initial conditions can vastly influence the final result. PAR uses a seed value to determine the initial conditions. This is referred to as the *cost table* value. You may change this value in the Project Navigator by hand. Even better, you may perform a multipass PAR process which runs PAR multiple times with different cost table values. This is time-consuming but often effective.
 6. **Floorplanning.** This step should be avoided if possible, but can yield huge improvements. The automatic placer in PAR can be improved upon by human intervention. Floorplanning places critical elements close to each other on the Xilinx die, reducing net delays. The PACE and Floorplanner tools in ISE may be used for this. A more advanced tool, PlanAhead, is also available separately from Xilinx to aid in this task.
 7. **Use a faster part.** This is often the first solution seized upon, but is also expensive. If you are using an old Xilinx part, porting your design to a newer, faster Xilinx part may often save money because the new parts may be cheaper on account of Moore's Law. However, moving to a faster part in the same family incurs significant extra costs, and often isn't necessary if the previous steps are followed.

Example Design

See the Example: Using the System Generator Timing Analysis Tool to Meet Timing Requirements for a tutorial on using the timing analyzer.

Additional Topics and Tutorials

Designing for the DSP48 block

Xilinx Virtex-4 devices offer an efficient building block for DSP applications called the DSP48 (also known as the Xtreme DSP Slice). The DSP48 is available as a System Generator block which is a wrapper for the DSP48 UNISIM primitive. DSP48. Architectural and usage information for this primitive can be found in the DSP48 users guide in the Virtex-4 databook. The DSP48 is available in all Virtex-4 devices and is featured in the SX series devices which contain up to 512 DSP48 blocks.

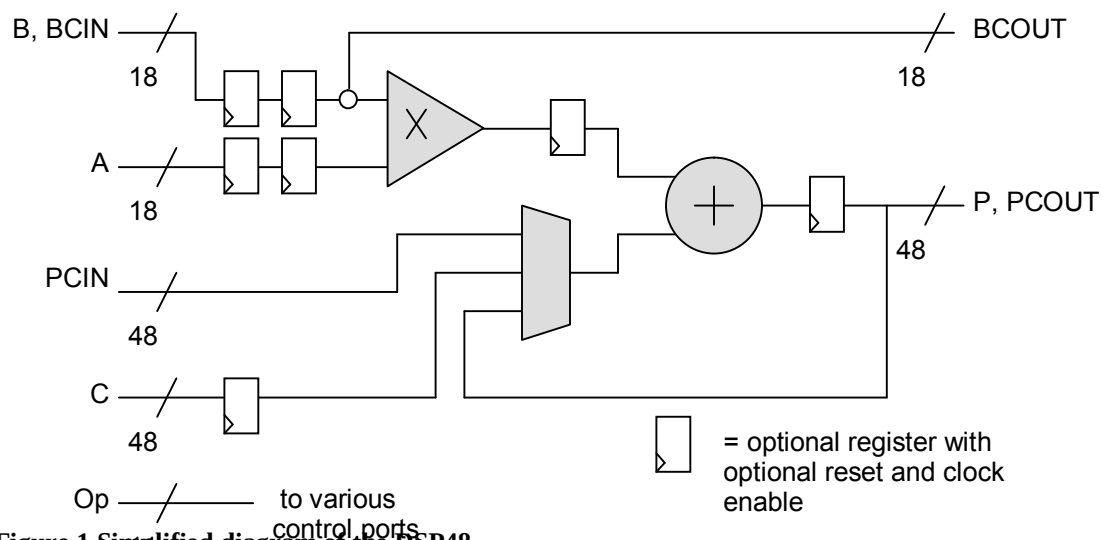


Figure 1 Simplified diagram of the DSP48

The DSP48 combines an 18-bit by 18-bit signed multiplier with a 48-bit adder and a programmable mux to select the adder's inputs. It implements the basic operation: $p = a * b + (c + c_{in})$; however other operations can be selected dynamically. Optional input and multiplier pipeline registers are also included and must be used to achieve maximum speed. Also included with the DSP48 are high performance local interconnects between adjacent DSP48 blocks (BCIN-BCOUT and PCIN-PCOUT). The DSP48 also includes support for symmetric rounding. This combination of features enables DSP systems which use the DSP48 to be clocked at over 500 MHz using Virtex-4-11 devices.

DSP design styles for the DSP48 block

There are three ways to program DSP48s in System Generator:

- **Use Standard Components** - Map designs to mult and adder blocks or use higher level IP such as the MACFIR filter generator blocks. This approach is useful if the design needs to be compatible with V2P or S3 devices or uses a lower speed clock and the mapping to DSP48s is not required.
- **Synthesizable Designs** - Structure the design to map onto the DSP48's internal architecture and

- compose the design from synthesizable multiplier, adder, mux and delay blocks. This approach relies on synthesis to infer DSP48 blocks where appropriate. This approach gives the compiler the most freedom and can often achieve full rate performance.
- **Use DSP48 blocks** - Use System Generator's DSP48 and DSP48 Macro blocks to directly implement DSP48 based designs. This is the highest performance design technique. Be aware however that obtaining maximum performance and minimum area for designs using DSP48s may require careful mapping of the target algorithm to the DSP48's internal architecture, as well as physical planning of the design.

Designs using standard components

Designs for Xilinx FPGAs such as Virtex2P and Spartan3 will compile to the Virtex-4 devices. Multipliers will be mapped into the DSP48 block, however synthesis tools cannot pack adders and muxes into the DSP48 block since these blocks are delivered as cores which prevents synthesis from optimizing the logic. Place and route tools do place the MULT18x18S and MULT18x18 into the DSP48 block but do not pack the adder, or mux into the DSP48 block. (PAR will however pack the mux into the LUT based adder).

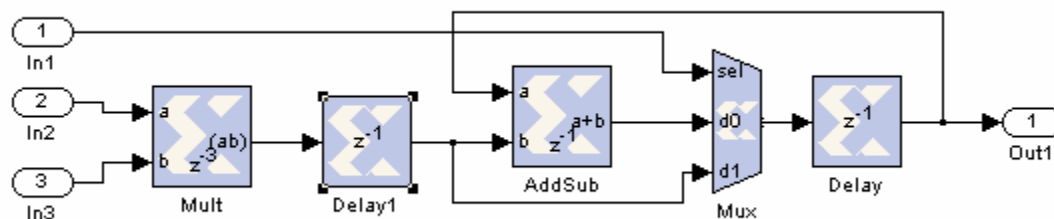


Figure 2 DSP design using Mult, AddSub and Mux blocks

To obtain the best possible performance, set the multiplier latency to 3 and include an input register to cover the delay from the DSP48's output to the adder. The performance of this circuit is in the 200-300 MHz range with V-44-11 devices, and is limited by the adder speed. In Virtex-4, unlike V2P and S3 devices, the multiply speed is nearly independent of bit width. For medium speed designs, this approach works fine.

An additional way to use the DSP48 is to use IP blocks optimized for the DSP48 such as the macfir block available from coregen, or to use the architecture wizard to generate a custom configured DSP48. Both of these approaches require importing the logic containing the DSP48 as a black box into system generator. Simulation will require modelsim HDL cosim.

Designs using synthesizable mult, mux and addsub blocks

Synthesis tools now have the ability to infer DSP48 logic. This enables the tools to pack adders, multipliers and muxes into the DSP48 block, as well as to enable application of retiming and other synthesis techniques such as register duplication.

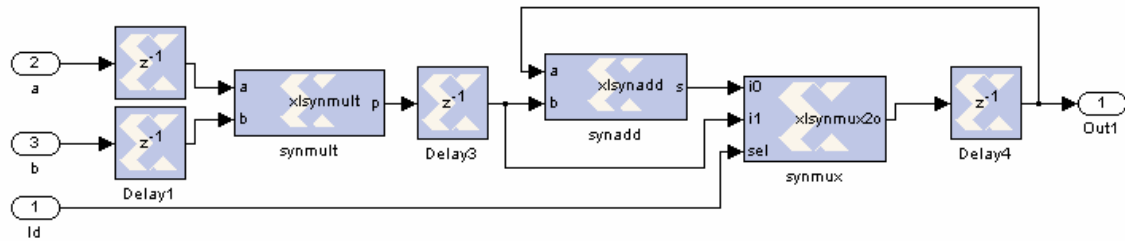


Figure 3 synthesizable DSP48 design using m-code blocks.

If the design is composed of synthesizable blocks, both SynplifyPro and XST have demonstrated the ability to infer DSP48s and to make use of the DSP48's local interconnect busses (PCOUT-PCIN and BCOUT-BCIN). In the above example, three blocks have been built using the m-code blocks which are defined by the following m-code functions.

```
function o = xlsynmux2(i0,i1,sel)
```

```
if (sel==0) o=i0; else o=i1; end
```

```
function p = xlsynmult(a,b)
```

```
p=a*b;
```

```
function s = xlsynadd(a,b)
```

```
s=a+b;
```

For synthesis to work, the circuit must be mappable to the DSP48 and signal bitwidths must be less than the equivalent busses in the DSP48.

It should be kept in mind that the synthesis tools are rapidly evolving and that inferring DSP48 configurations is more of an art than a science. This means that some mappable designs may not be mapped efficiently, or that the mapping results may not be consistent. It will be necessary to inspect the post synthesis netlist using a tool similar to Synplify Pro's gate level technology viewer to determine if the design is being correctly mapped. If not, it may be possible to recast it to be correctly inferred.

Click on this link ([dsp48/synth_fir_tb](#)) to open a simulink model containing a fully synthesizable FIR filter.

DSP48 and DSP48 Macro based designs

DSP48 Block

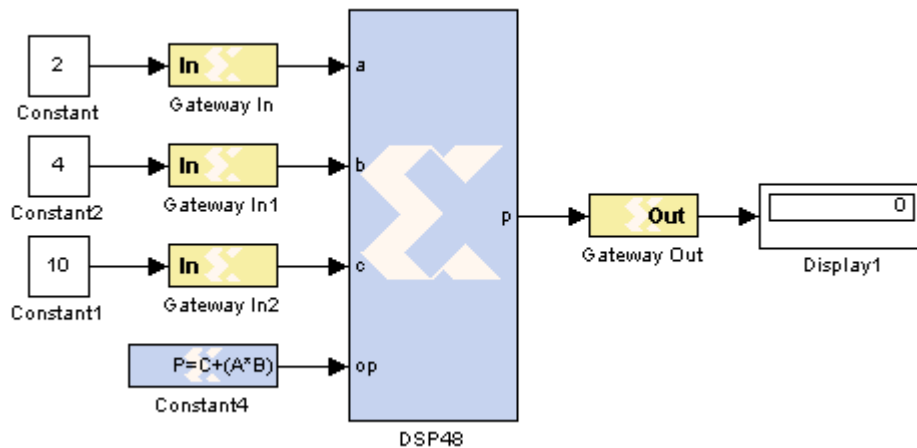


Figure. 4 DSP48 design using ‘DSP48’ and ‘Constant’ blocks.

The DSP48 block is effectively a wrapper for the DSP48 UNISIM primitive. Because of this, any possible DSP48 design can be implemented. This low level implementation however requires an 11-bit binary opmode to be routed to the DSP48’s control ports in order to configure its function. The Constant block has a special mode enabling it to generate a DSP48 control field. The DSP48’s parameters dialog box is used to configure the pipelining mode of the DSP48 as well as the use of the DSP48’s local interconnect busses named PCOUT-PCIN and BCOUT-BCIN.

Click on this link ([dsp48/dsp48](#)) to open a simulink model and try out the DSP48 block.

Dynamic Control of the DSP48

The DSP48 has the unique capability of being able to change its operation on a per cycle basis. This is useful in applications where the DSP48 is used in a ‘resource shared’ mode such as a FIR filter where multiple taps are implemented by the same multiplier. A simple method of generating this type of control pattern is to use a mux to select the DSP48 instruction on a clock by clock basis.

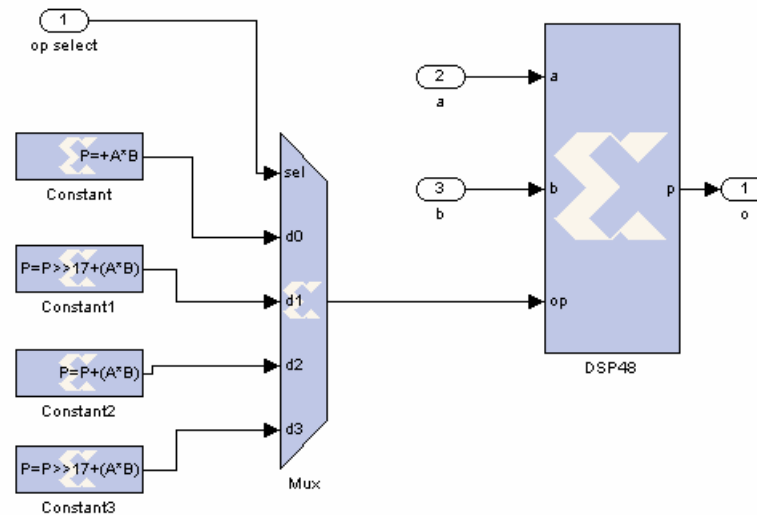


Figure 5 - using a mux to select the DSP48s operation dynamically.

The above example illustrates the use of a DSP48 and Constant blocks to implement a 35-bit by 35-bit multiplier over 4 clock cycles. During synthesis, the mux and constant logic is reduced by logic optimization. In the example above, the 4 'DSP48 Op' blocks and the 4:1 mux are reduced to just two 4-LUTs

Click on this link ([dsp48/mult35x35_tb](#)) to open a Simulink model which illustrates how to implement both parallel and sequential 35*35-bit multipliers using dynamic operation for the sequential mode of operation

DSP48 Macro Block

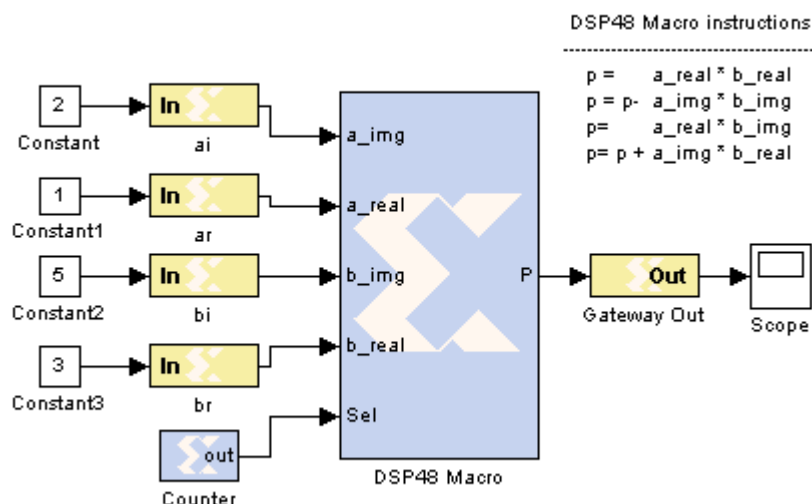


Figure 6 – The DSP48 Macro block configured to implement a complex multiplier.

The DSP48 Macro block is a wrapper for the DSP48 block which makes it simple to implement a sequence of DSP48 instructions (known as dynamic instructions). In addition it provides support for specifying input and output types. For example in the model above, a DSP48 Macro block is configured to implement a complex multiplier using a sequence of four different instructions. The instructions are entered in a text window in the DSP48 Macro's dialog menu. Click on this link ([dsp48/dsp48_mac](#)) to open the model and try out the DSP48 Macro.

DSP48 Design Techniques

Designing Filters with the DSP48

The DSP48 is ideal an ideal block to implement FIR filters. Click on this link ([dsp48/dsp48_firs_tb](#)) to open a model which demonstrates the use of the DSP48 for Type 1 and Type 2 FIR filters, as well as sequential and semi/parallel FIR filters.

Design techniques for very high performance designs

DSP48 based designs will usually require also I/O, BRAMS and SLICE logic. Typically this associated SLICE logic is used to implement delay registers, SRL16s, muxes, counters and control logic. Since the DSP48 block is expected to operate at speeds greater than 500 MHz, these other components will also be required to operate at the same speed. This generally requires special design techniques for the non-DSP48 logic.

At 500 MHz only 2 ns is available in each clock. For V4-11 devices, roughly 300 ps are required for register clock to out and 300 ps for setup. For comparison a LUT delay is 166 ps. Special inputs and outputs such as clock enables and DSP48 and BRAM signals generally have setup and clock to out times closer to 500 ps. With clock skew and jitter roughly 1 ns is available for net delays. This restriction will generally allow only 1 net in each path and it must be fairly short.

There are a number of guidelines that can be used to insure operation at DSP48 speeds. Some of these are outlined below.

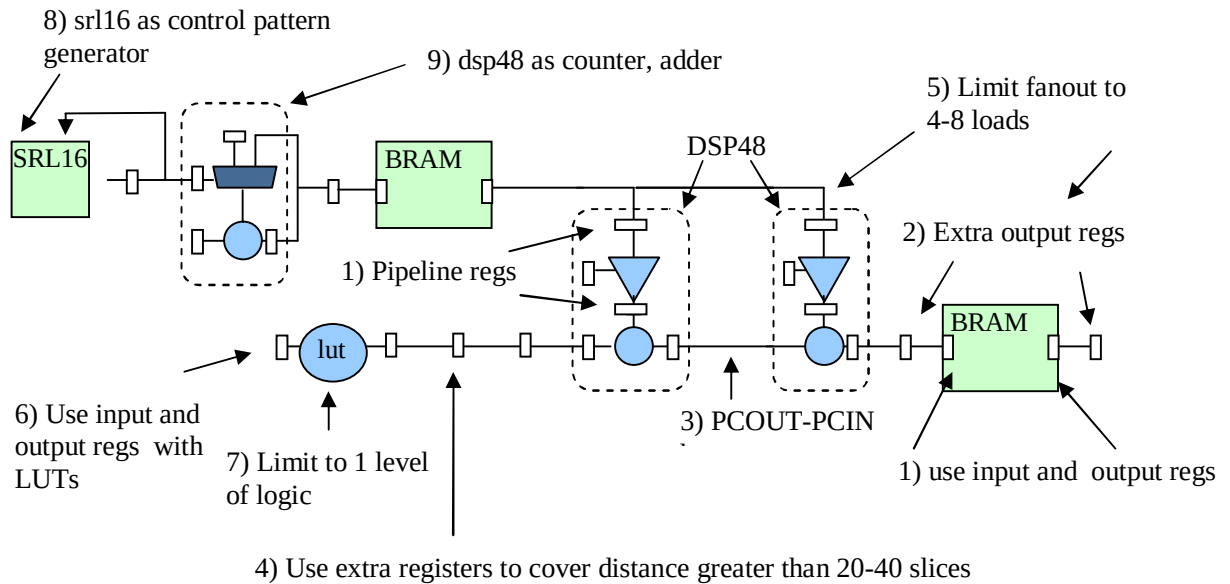


Figure 7 Guidelines for very high performance designs

1. Always use DSP48, BRAM16, FIFO16 with input, mult and output registers
2. Use additional FF to buffer DSP48 and BRAM outputs if necessary
3. Plan out usage of PCOUT-PCIN bus to allow DSP48 chaining
4. Add registers to any path which is greater than 20 - 40 slices long
5. Limit fanout to 32 loads located within 20 slice distance
6. Add output registers to any LUT based logic
7. Limit LUTs to 1 levels or a 4:1 MUX and insure a local register for input or output
8. Use RAMs, SRL16 to clock out control patterns instead of state machines
9. Use DSP48 to implement counters and adders greater than 8-16 bits
10. Use area constraints – “INST ff1* LOC = SLICE_X0Y8:SLICE_X1Y23;”

Physical planning for DSP48 based designs

The DSP48 requires correct placement to achieve dense, high performance designs. While the automatic place and route tools do a good job, the best results may require placement of DSP48 and RAM blocks. There are several additional issues with the DSP48s

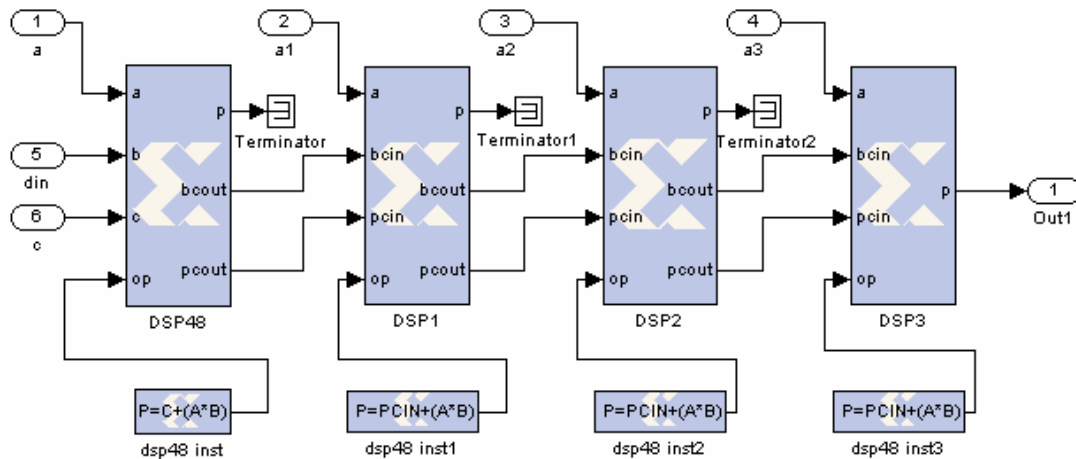


Figure 8 Guidelines for very high performance designs

Cascade routing busses

Adjacent DSP48 blocks are connected with two local busses called PCOUT, and BCOUT. The PCOUT bus is used to pass accumulation data from one DSP48 to the next. The BCOUT bus is used to pass delayed B input data to the next DSP48. The DSP48 and DSP48 Macro block both support PCOUT and BCOUT busses. The use of the busses is shown in the figure above, which illustrates a pipelined 4-Tap Type 1 FIR filter.

C-input sharing

Each pair of DSP48s share a single C input. The user should be aware of this when resource planning. Since the placer will not always find the most optimal placement to share C inputs, DSP48s should avoid using C inputs if possible.

Adder Trees planning

Tree based filter topologies are problematic for efficient DSP48 implementation. An adder tree requires isolated 2-input adders. Two input 36-bit adders can be implemented using a single DSP48, however this requires a C input and precludes the use of the multiplier. In addition the long signals between DSP48s may require additional pipeline stages. A better approach is to convert the tree into a pipelined cascade.

Placement

Most designs will benefit from some placement of DSP48 and BRAMs. Use of area constraints to constrain LUT fabric logic placement may also be beneficial.

Signal length planning

At 500 MHz limit signals lengths should be limited to around 20 slices in length. This means that long signals should have multiple pipeline stages.

Clock Enable planning

Clock enables will often be the limiting path at high frequencies. This is partially due to System Generator's use of LUTs to gate clocks at the destination. To avoid clock enables in the critical path, avoid using the system generator upsampled and downsampled clock domains. This requires manual use of clock enables for logic which runs at less than the system clock rate.

Place and route flow

- Use map -timing with effort level high for both map and place
- Use trce -v 100 to get a good sense of the failing nets and inspect the xflow/design.twr file to understand the nature of the design's timing.
- The file bitstream_v4.opt is available in the examples/dsp48 directory. This file can be used with the 'bitstream' compile target to set the par options mentioned above.

Synthesis Flow

- Use Synplify Pro with retiming and pipelining enabled to avoid having to manually pipeline every LUT and signal.
- Use Synplify Pro with the fanout limit set around 32 to avoid long net delays.
- Open compiled projects in Synplify and inspect the generated logic using the RTL and Gate level views to get a good idea of what logic is being generated.
- The file syn.pl is available in the examples/dsp48 directory. Place this file in \$MATLAB\toolbox\xilinx\sysgen\scripts directory to modify the synthesis options in System Generator

Logic Depth planning

The following rules seem to allow the LUT fabric to run at 450 MHz using a -11 V4 device

- Only one net can be allowed in a critical path at 450 MHz. This allows a 4:1 mux to a reg a 4_input LUT to a reg. or a net through a LUT directly to a DSP48
- Counters up to 16-bits can be used, but do not use count limited counters without additional pipelining
- If accumulators or counters are used invert the enable line to an active low condition to prevent a extra LUT being inserted in the critical path
- Any adders must have local input registers. It may be necessary to place control counters in the DSP48 to insure speed.

Fanout planning

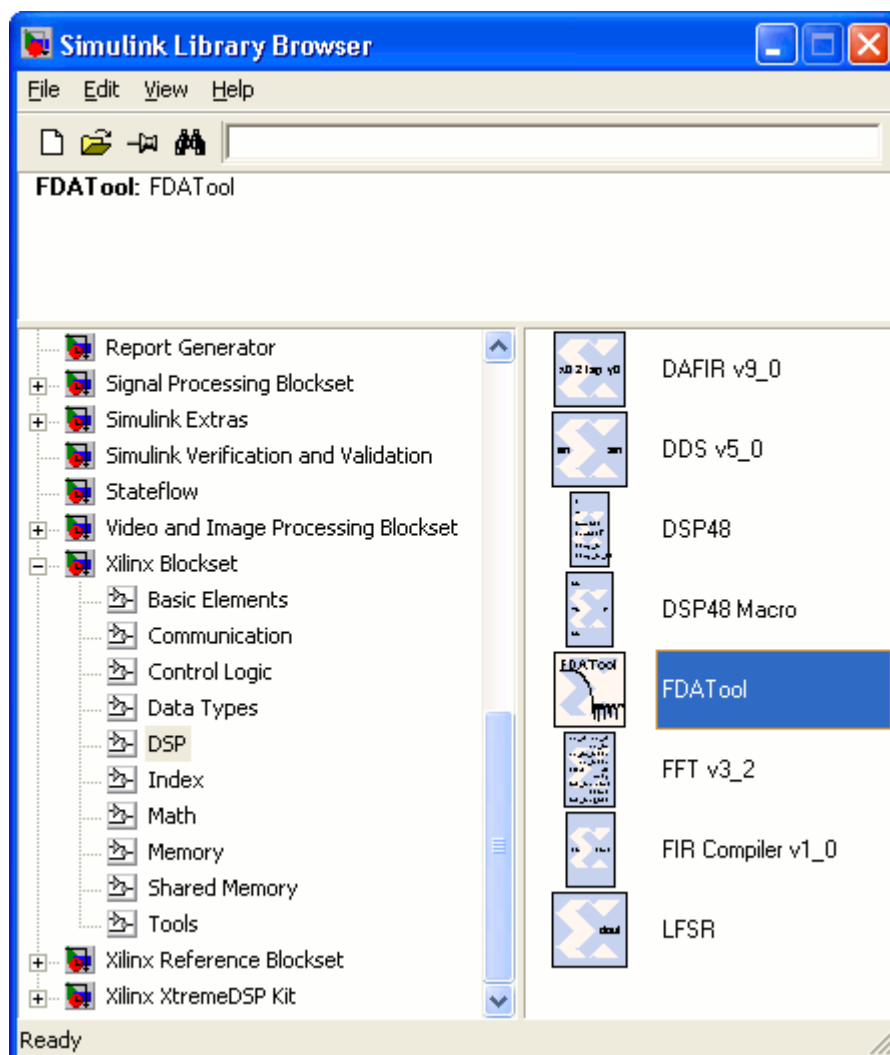
Avoid fanouts of more than 32 LUTs or 8 DSP48s or BRAMs. This can be avoided by inserting additional pipeline registers on these signals.

Register Retiming

Check retiming on delay blocks to allow them to be used as registers for pipelining. Then use Synplify or XST with retiming enabled to allow the synthesis tool to move registers into optimal position.

Using the FDATool in Digital Filtering Applications

This example demonstrates one way of specifying, implementing, and simulating a FIR filter using the FDATool provided in the “DSP” and “Tools” Xilinx Blockset libraries. The FDATool block is used to define the filter order and coefficients and the Xilinx Blocksets are used to implement a MAC-based FIR filter using a single MAC (Multiply-ACcumulate) engine. The quality of frequency response is then validated by comparing it to a double-precision Simulink filter model.



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

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

```
>> demo blockset xilinx
```

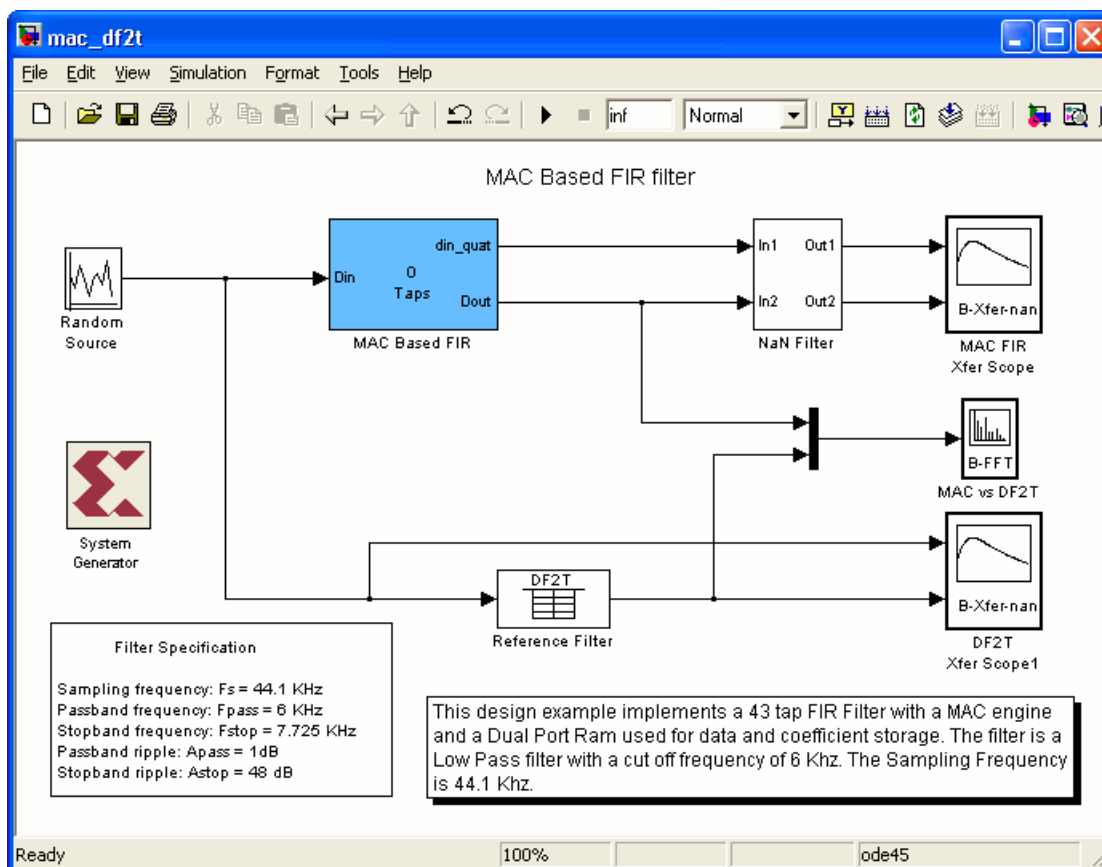
Then select **FIR filtering: Polyphase 1:8 filter using SRL16Es** from the list of demo designs.

Design Overview

In this design, we use the random number source block from the DSP Blockset library to drive two different implementations of a FIR filter:

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

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



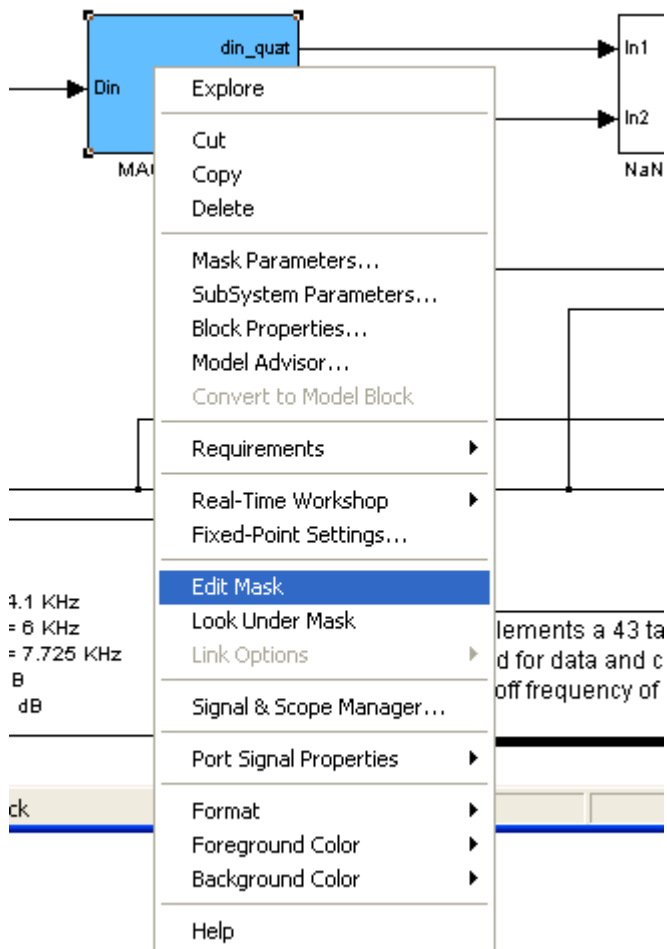
Open and Generate the Coefficients for this FIR Filter

1. To open the MAC filter model from the MATLAB console window, first, **cd** into the directory `$MATLAB\toolbox\xilinx\sysgen\examples\mac_fir`.
2. Open the design model by typing `mac_df2t` from your MATLAB command window.

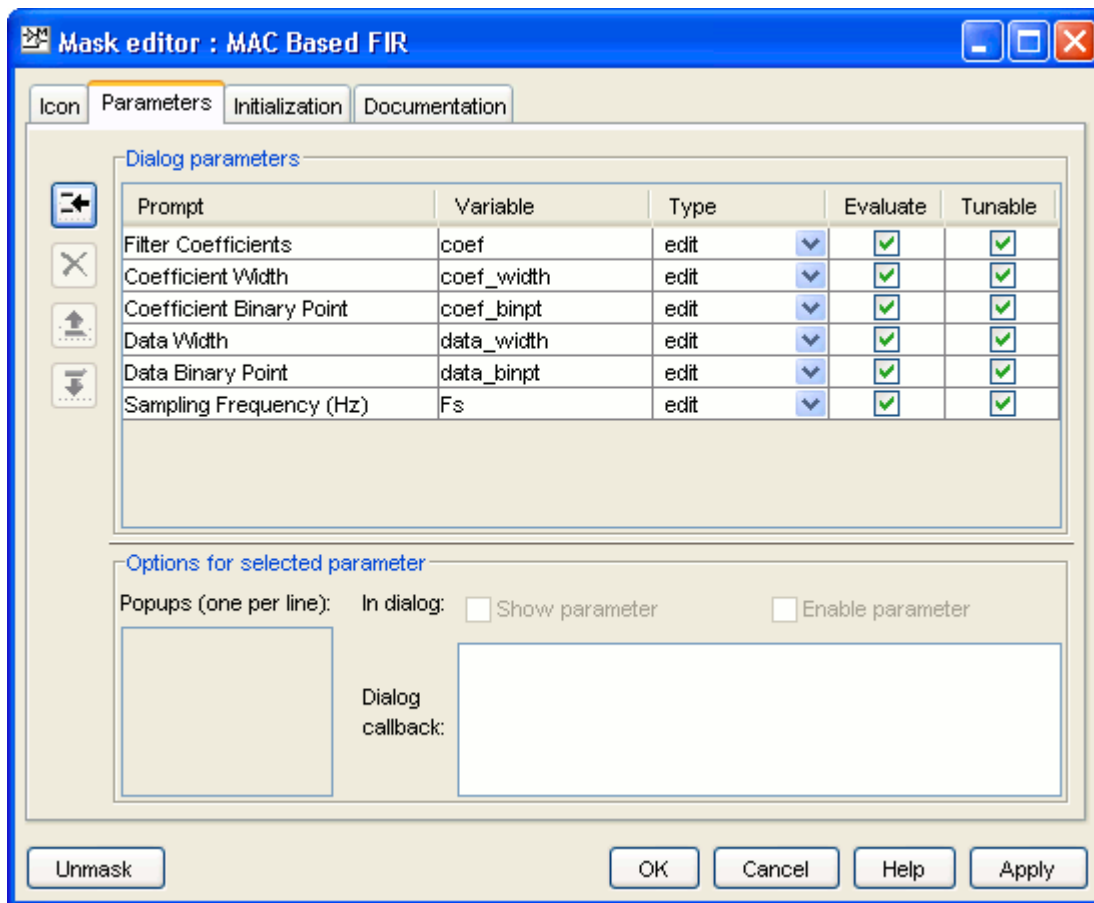
For the purpose of this tutorial, the variables `coef`, `coef_width`, `coef_binpt`, `data_width`, `data_binpt` and `Fs` are not defined. We would first use these variables as mask parameters to the MAC block and then design and assign the filter coefficients using the FDATool. The fully functional model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

Parameterize the MAC Based FIR Block

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



2. Add the parameters `coef`, `data_width` and `data_binpt` as shown below.



Generate and Assign Coefficients for the FIR Filter

3. Drag and drop the FDATool block in your model from the DSP Xilinx Blockset Library.
4. Double-click on the FDATool block and enter the following specifications in the Filter Design & Analysis Tool for a low-pass filter designed to eliminate high-frequency noise in audio systems:

- Sampling frequency: $F_s = 44.1$ KHz
- Passband frequency: $F_{pass} = 6$ KHz
- Stopband frequency: $F_{stop} = 7.725$ KHz
- Passband ripple: $A_{pass} = 1$ dB
- Stopband ripple: $A_{stop} = 48$ dB

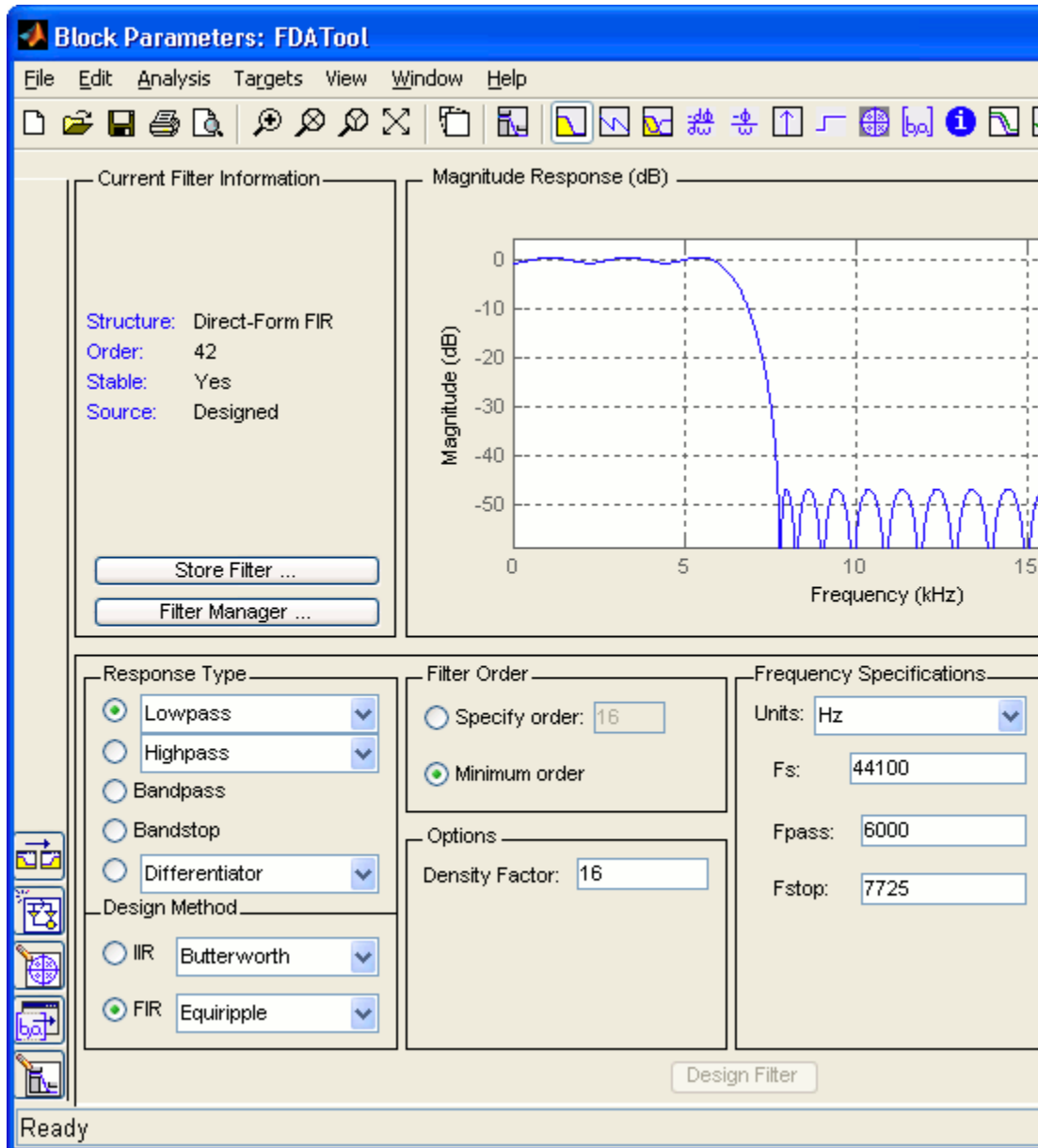


Figure 1: FDATool Session using MATLAB Signal Processing Toolbox

5. Click on **Design Filter** at the bottom of the tool window to find out the filter order and observe the magnitude response.

You can also view the phase response, impulse response, coefficients and more by selecting the appropriate icon at the top right of the GUI. Based on the FDATool, a 43-tap FIR filter is required in order to meet the design specifications listed above.

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

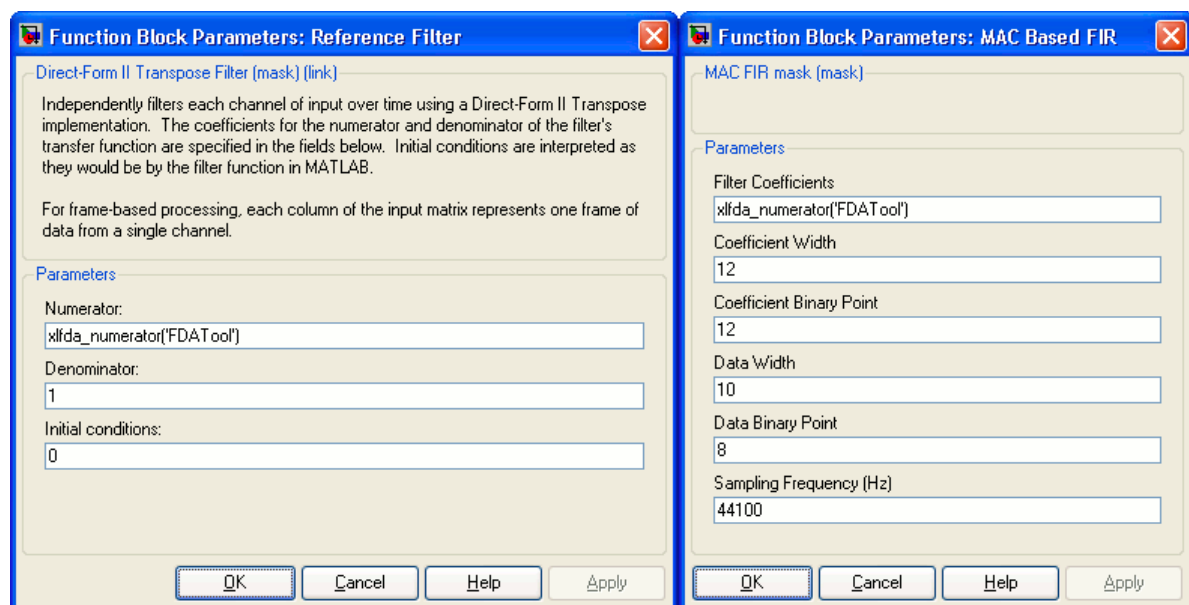
```
>> xlfda_numerator('FDATool')
```

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

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

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

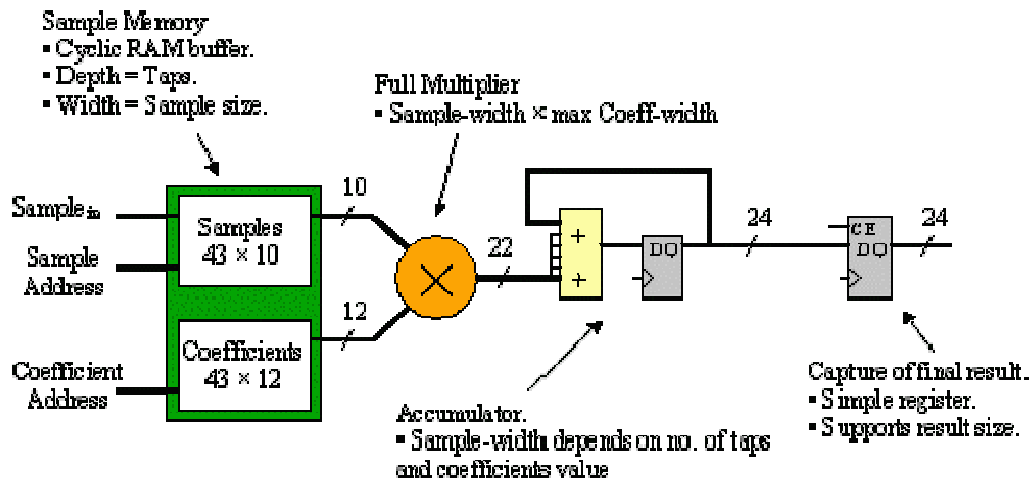
6. Enter the parameter values for `coef`, `coef_width`, `coef_binpt`, `data_width`, `data_binpt` and `Fs` as shown below.



In order to be able to preload the coefficients without having to regenerate them with the FDA Tool each time the `mac_df2t.mdl` simulink model is opened, we are now going to save it into a file.

Browse through and understand the Xilinx Filter block

Here is the block diagram showing how the MAC-based FIR filter has been implemented for this tutorial.



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

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

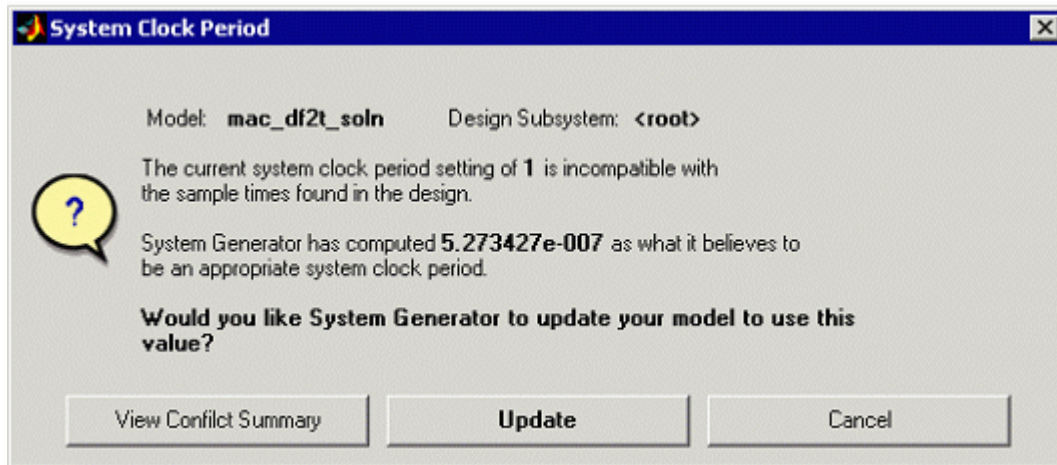
$$\text{acc\_nbits} = \text{ceil}(\log_2(\text{sum}(\text{abs}(\text{coef}^2 \wedge \text{coef\_width\_bp})))) + \text{data\_width} + 1;$$

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

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

Run the Simulation

When you run the simulation you will get the following message as shown in the figure below. System Generator gets its input sample period from the din “Gateway In” block which has 1/Fs specified as the data input sample period. As the MAC-based FIR filter is oversampled according to the number of taps, the System Clock Period would always be equal to $1/(\text{Filter Taps} * F_s)$. Click on Update to specify the System Clock Period as $5.273427\text{e-}007 = 1/(43 * 44100)$.



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

As you can see, the filter passband response measurement as well as zeros can clearly be seen. Your FIR filter is perfect!

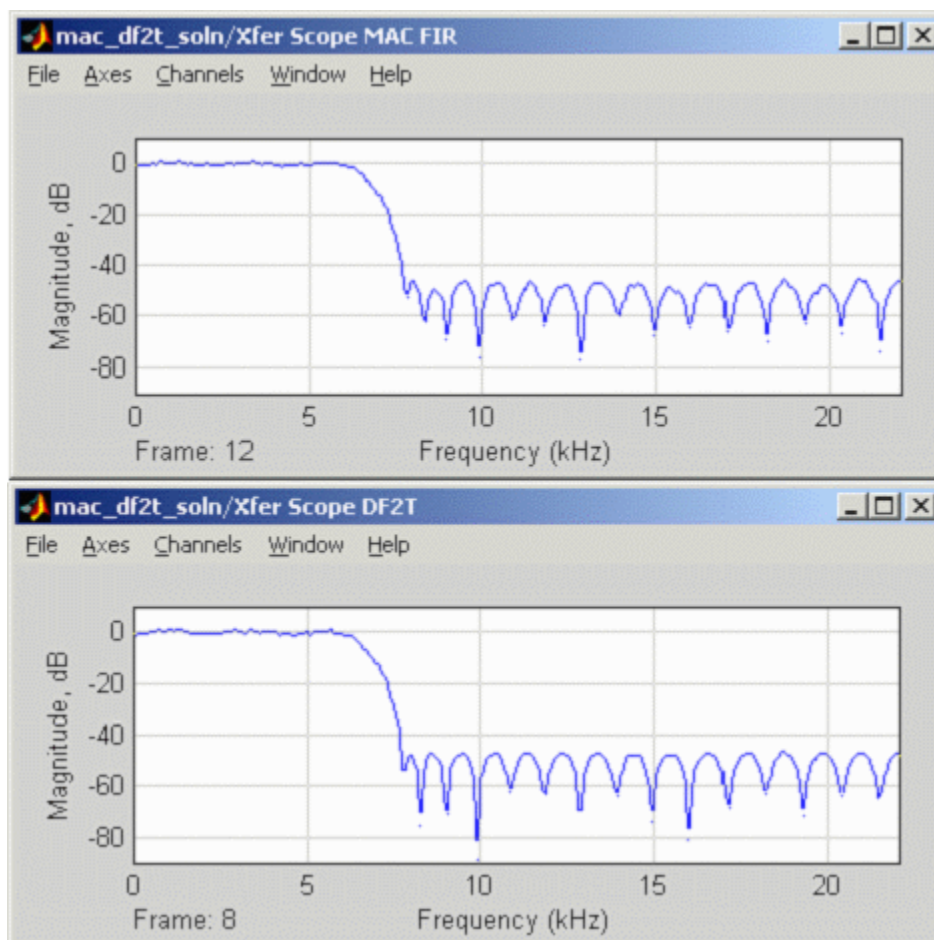


Figure 2: Frequency Response of MAC FIR vs Reference Filter

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

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

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

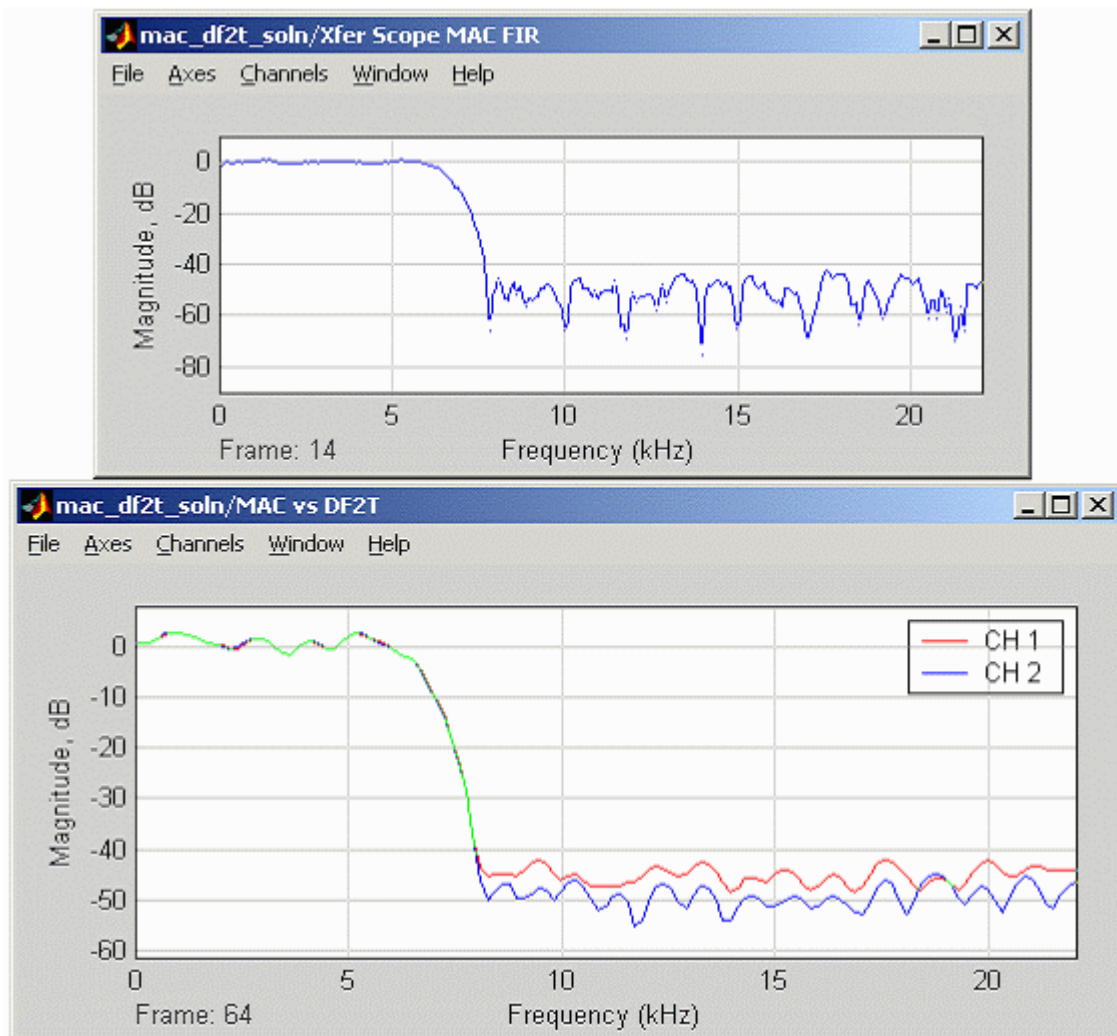


Figure 3: Degraded Frequency Response of MAC FIR vs Reference Filter

Finally, open the System Generator block and select **Override with Double -> Everywhere in the Subsystem**. At this point, all the Xilinx Blocks are grayed out since a netlist cannot be generated in a double precision mode.

Re-run the simulation and look at the MAC vs DF2T scope. You should notice that the two outputs completely overlap, showing the accuracy of the Xilinx model.

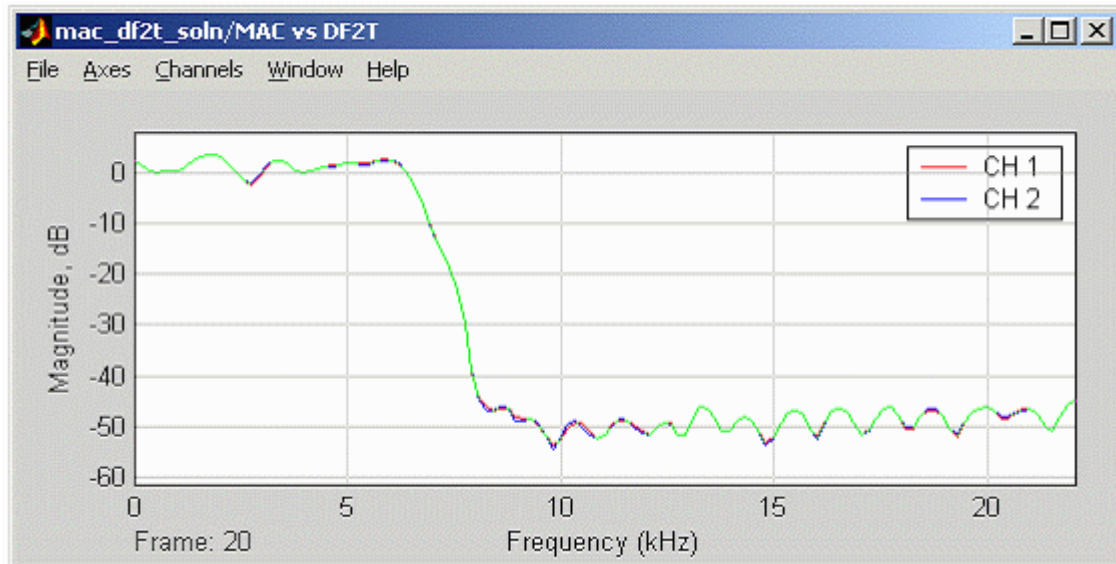


Figure 4: Frequency response with double precision datapath selection

Advanced Black Box Example Using ModelSim

The following topics are discussed in this example:

- How to design a black box with a dynamic port interface;
- How to configure a black box using mask parameters;
- How to assign generic values based on input port data types;
- Saving black box blocks in Simulink libraries for later reuse;
- How to specify custom scripts for ModelSim HDL co-simulation.

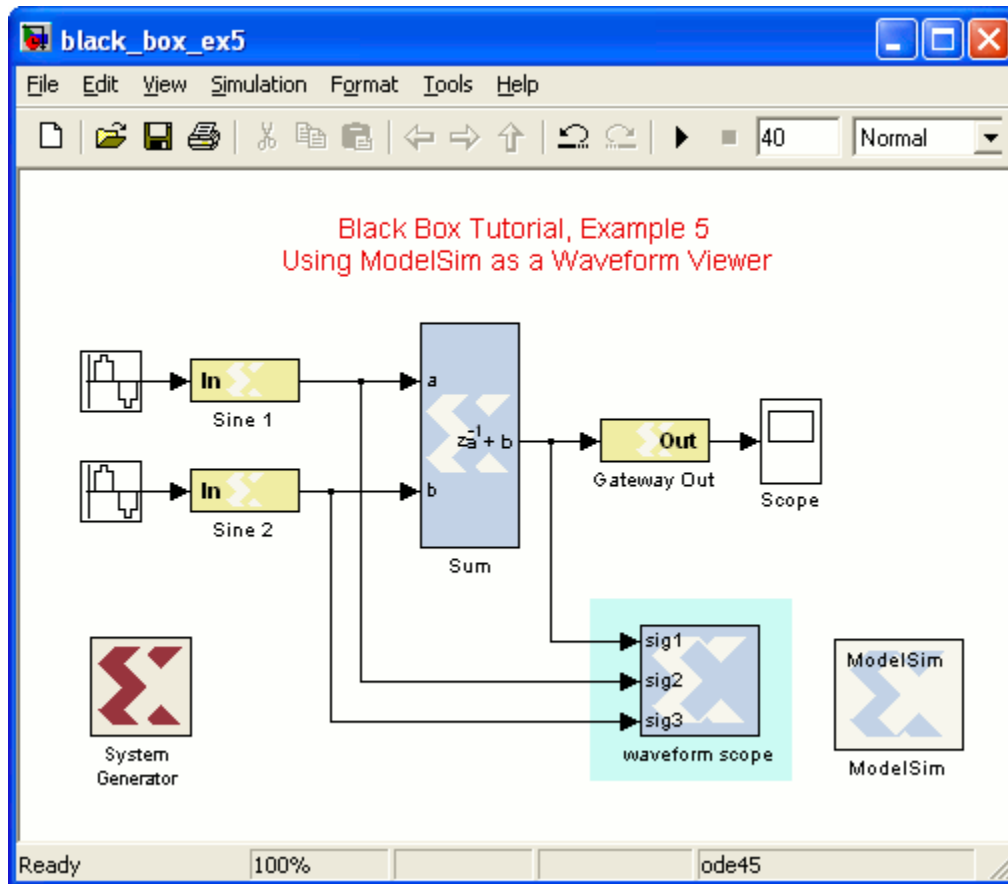
This example also shows a way to view signals coming from a black box. In Simulink, waveforms are typically viewed with a scope. The Simulink scope block serves this purpose and the System Generator WaveScope block is available in versions 8.1 and later. The waveform viewer in Modeltech's ModelSim simulator may also be used to view waveforms. In this example, a black box is configured as a specialized ModelSim waveform scope for Xilinx fixed-point signals. When a model that uses the black box scope is simulated, the signals that drive the black box are displayed in ModelSim.

The files for this example are contained in the directory
<path_to_sysgen>/examples/black_box/example5.

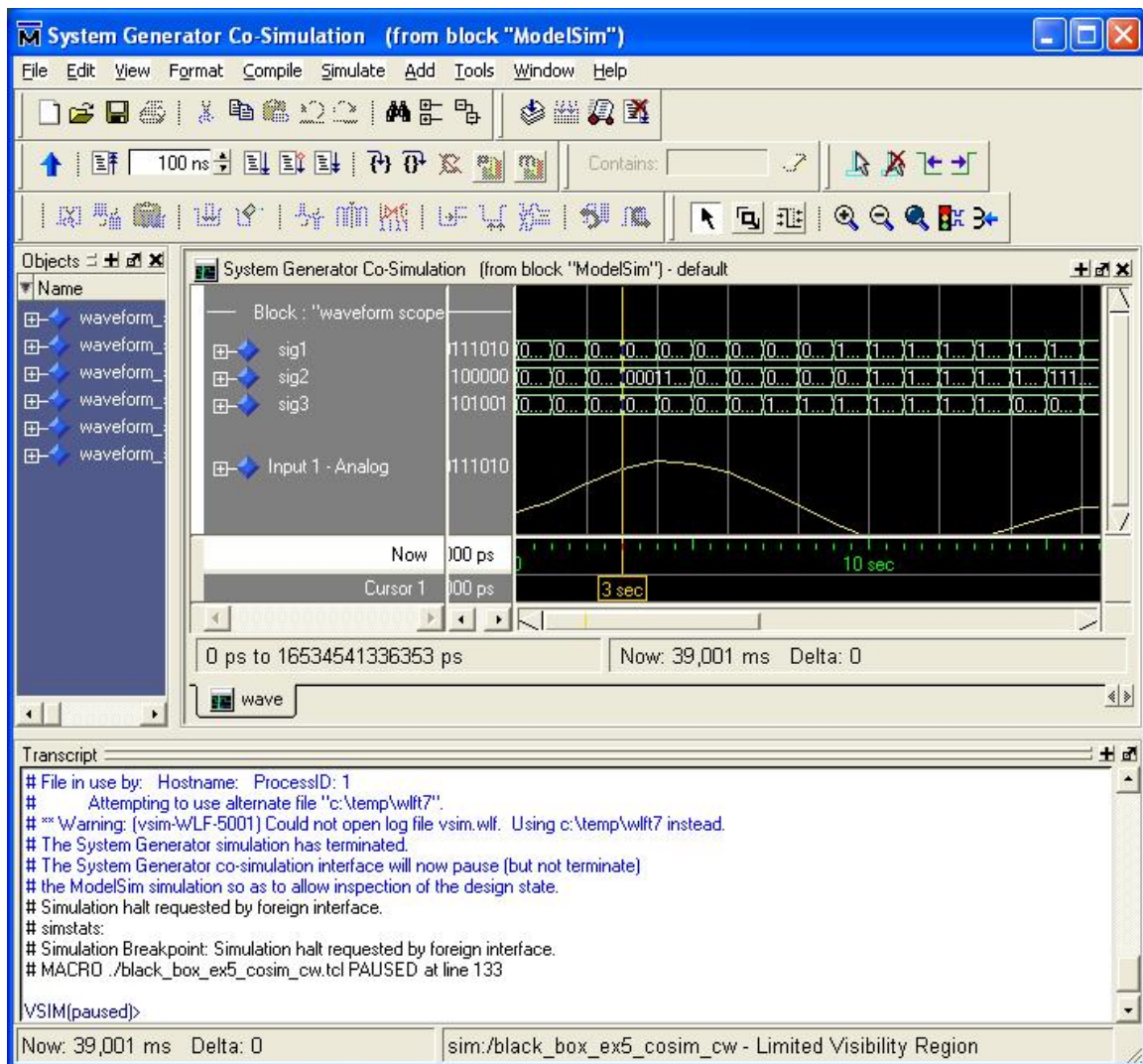
The files contained in this directory are:

- `black_box_ex5.mdl`: A Simulink model containing a black box scope.
- `scope_lib.mdl`: A Simulink library containing the black box waveform viewer.
- `scope_config.m`: The configuration m-function for the black box waveform viewer.
- `scope1.vhd`, `scope2.vhd`, `scope3.vhd`, `scope4.vhd`: Black box VHDL for the signal scope that accept one, two three, and four input signals, respectively.
- `waveform.do` – A script that instructs ModelSim how to display signals during simulation.

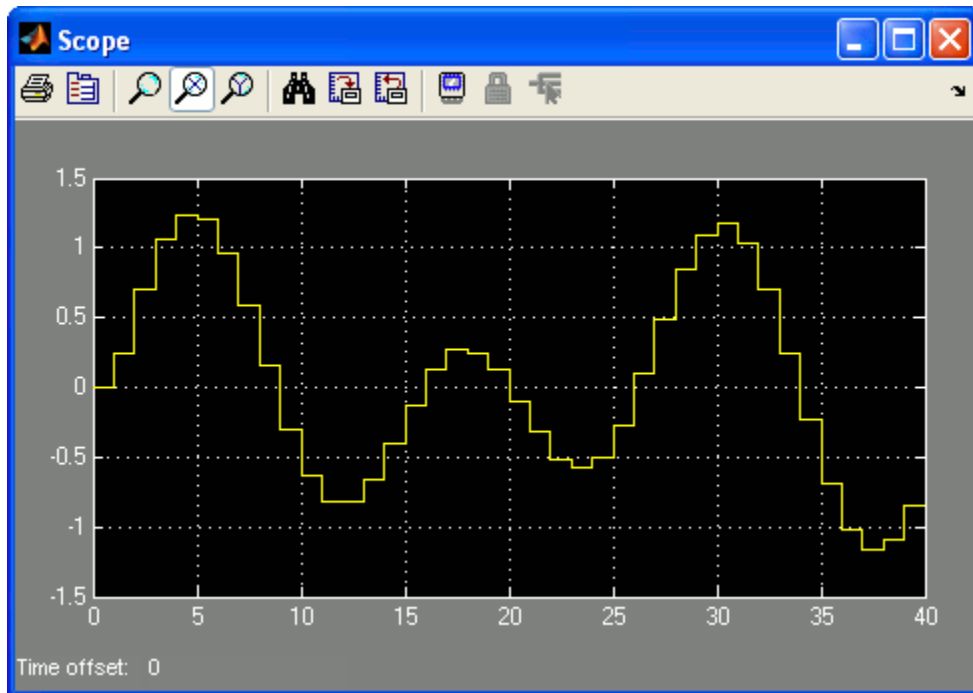
1. Navigate into the `example5` directory and open the example `black_box_ex5.mdl` file. The model includes an adder that is driven by two input gateways. The gateways are configured to produce signed 8-bit values, each with six bits to the right of the binary point. Sine wave generators drive the gateways. The model also includes a black box named *waveform scope*. This is driven by three signals. The first input is driven by the adder. The other two are driven by the inputs to the adder. The ModelSim block enables HDL co-simulation. The example model is shown below.



2. Simulate the `black_box_ex5` model. A ModelSim window opens and ModelSim compiles the files necessary for simulation. After the compilation is complete, both MATLAB and ModelSim simulations begin. A ModelSim waveform viewer opens and displays four signals. The first input to the block, `sig1`, is driven by the adder. This signal is represented in two ways in the ModelSim viewer – binary and analog. The ModelSim waveforms for the `black_box_ex5` simulation are shown below.

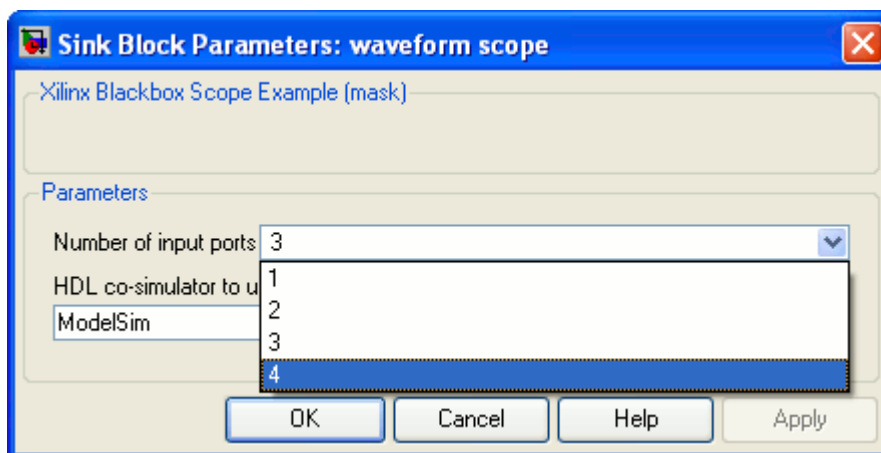


3. Double click on the Simulink scope in the model. The output is shown below and resembles the analog signal in the ModelSim waveform viewer.

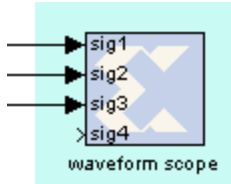


The black box in this example is configured using mask parameters. There are many situations in which this is useful. In this case, the number of black box input ports, i.e., the number of scope inputs, is determined by a mask parameter.

4. Double click on the waveform scope black box. Notice a **Number of Input Ports** field is included in the block dialog box and is unique to this black box instance. The dialog box is shown below:



5. Change the number of input ports from 3 to 4 and apply the changes. The black box now has an additional input port labeled "sig4" and should look like the following:



Every black box has a standard list of mask parameters. The black box in this example has an additional mask parameter `nports` that stores the number of input ports selected by the user. To change a black box mask it is necessary to disable the link to the library. When a black box is changed in this way, it is best to save the black box in a library. (See the Simulink documentation on libraries for details.) The tutorial library `scope_lib.mdl` contains the modified signal scope black box used in this example. When a black box configuration m-function adds an HDL file, the path to the file can be relative to the directory in which the library is saved. This eliminates the need to copy the HDL into the same directory as the model.

The black box's configuration m-function is invoked whenever the block parameter dialog box is modified. This allows the m-function to check the mask parameters and configure the black box accordingly. In this example, the m-function adjusts the number of block input ports based on the `nports` parameter specified in the mask.

6. Open the file `scope_config.m` that defines the configuration m-function for the example black box. Locate the line:

```
simulink_block = this_block.blockName;
```

This obtains the Simulink name of the black box and assigns it to the variable `simulink_block`. The name is useful because it is the handle that MATLAB functions need to manipulate the block.

7. Locate the line:

```
nports = eval(get_param(simulink_block, 'nports'));
```

The value of the `nports` mask parameter is obtained by the `get_param` command. The `get_param` returns a string containing the number of ports. An `eval` encloses the `get_param` and converts the string into an integer that is assigned to the `nports` variable.

8. Once the number of input ports is determined, the m-function adds the input ports to the black box. The code that does this is shown below.

```
for i=1:nports
    this_block.addSimulinkInport(sprintf('sig%d', i));
end
```

There are four VHDL files, named `scope1.vhd`, `scope2.vhd`, `scope3.vhd`, and `scope4.vhd`, which the black box in this example can use. The black box associates itself to the one that declares an appropriate number of ports.

9. The configuration m-function selects the appropriate VHDL file for the black box. Locate the following line in `scope_config.m`:

```
entityName = sprintf('scope%d', nports);
```

The HDL entity name for the black box is constructed by appending the value of `nports` to `scope`.

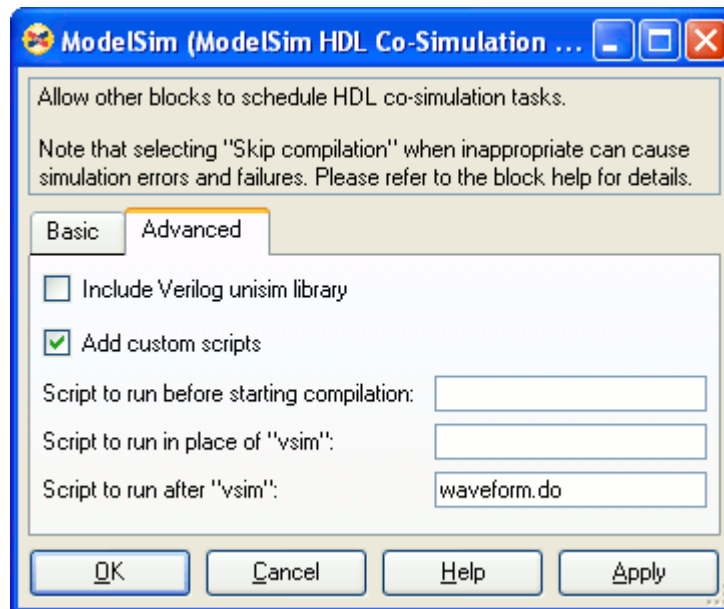
The VHDL is associated with the black box in the following line:

```
this_block.addFile(['vhdl/' entityName '.vhd']);
```

10. The input port widths for each VHDL entity are assigned using generics. The generic name identifies the input port to which the width is assigned. For example, the width3 generic specifies the width of the third input. In `scope_config.m`, the generic names and values are set as follows:

```
% -----
if (this_block.inputTypesKnown)
    for i=1:nports
        width = this_block.inport(i).width;
        this_block.addGeneric(sprintf('width%d', i), width);
    end
end % if(inputTypesKnown)
% -----
```

11. You can change the way ModelSim displays the signal waveforms during simulation by using custom tcl scripts in the ModelSim block. Double click on the ModelSim block in the `black_box_ex5` model. The following dialog box appears:



Custom scripts are defined by selecting the `Add Custom Scripts` checkbox. In this case, a script named `waveform.do` is specified in the `Script to Run after vsim` field. This script contains the ModelSim commands necessary to display the adder output as an analog waveform.

Generating Block Diagrams Programmatically

While the Simulink graphical block editor is commonly used for schematic capture of a system, it appears less widely appreciated that Simulink models can be constructed *programmatically* through a MATLAB API that supports block and signal instantiation, customization, deletion, and other construction methods. Using this API, it is possible to write MATLAB scripts that generate entirely new Simulink models (e.g., System Generator models). It can also be useful to use the API to customize a Simulink model in situ from its mask initialization block. This allows a subsystem to modify its *structure* in response to parameter changes.

Consider for example a rectangular-to-polar conversion block implemented using a CORDIC algorithm that is well suited to an FPGA. The computation is defined by the following set of iteration equations:

$$\begin{aligned}x_{i+1} &= x_i + y_i \delta_i 2^{-i} \\ y_{i+1} &= y_i - x_i \delta_i 2^{-i} \\ z_{i+1} &= z_i + \delta_i \tan^{-1}(2^{-i})\end{aligned}$$

where the variables δ_i are chosen to be either 1 or -1 so that y_i tends monotonically to zero.

The CORDIC algorithm is deceptively simple; it computes its output using nothing but shift and add operations. In a conventional processor, the iteration equations would be computed within an iterative loop (e.g., “for”, “while”). Although this can be done in an FPGA as well, it is also possible to achieve far higher throughput by parallelizing the computation, using a separate processing element (Figure 1) for each iteration step.

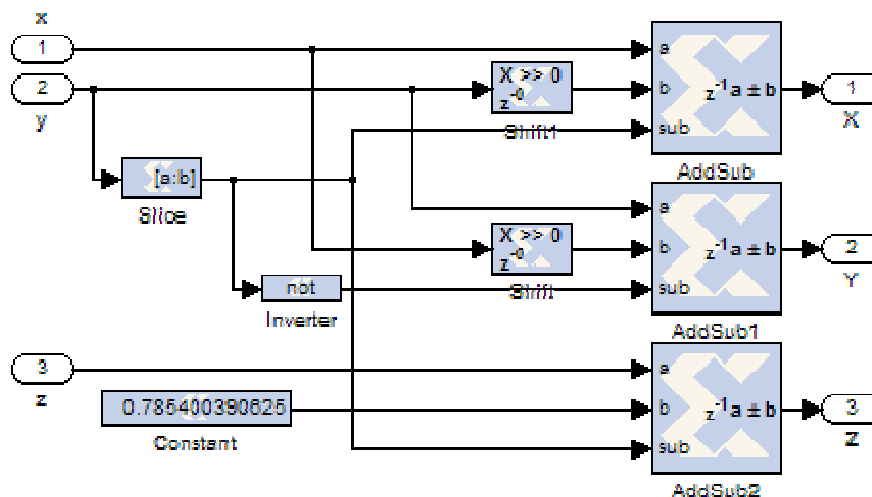


Figure. 1 CORDIC Processing Element

The CORDIC processor is built with a cascaded chain of such processing elements, each implemented as a Simulink subsystem, as shown in Figure 2. At the output of the final element in the chain, z_n is an approximation for $\tan^{-1}(x_0)$, and x_n is an approximation of $K(x^2 + y^2)^{1/2}$, where constant $K = 1.646760...$

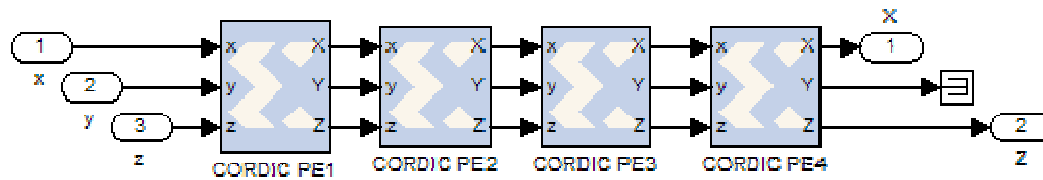


Figure. 2 CORDIC Processing Element Chain

The bit precision at the output of the CORDIC processor is a function of the number of iteration steps, or equivalently, the number of processing elements in the chain. This number can be stored as a mask parameter, and the diagram can be dynamically adjusted according to parameter changes.

In VHDL, this kind of iterated block instantiation would be accomplished using a FOR GENERATE statement. In System Generator, it can be done by invoking MATLAB API functions from the mask initialization code for the subsystem shown in Figure 1. The function calls

```
cordic_pe = find_system(gcb, 'lookUnderMasks', 'all',
    'FollowLinks','on','masktype', 'CORDIC parallel PE');

a=find_system(cordic_pe{1}, 'lookUnderMasks', 'all',
    'FollowLinks','on','masktype', 'CORDIC iteration PE');
```

produce a vector (cell array) 'a' of all contained CORDIC PE subsystems, each of which can be deleted as follows:

```
delete_block(a{i});
```

The function call

```
delete_line(cordic_pe{1}, ['CORDIC PE' int2str(i-1) '/'
    int2str(j)], ['CORDIC PE' int2str(i) '/' int2str(j)]);
```

deletes the lines between elements; similar calls delete the lines to the subsystem input and output ports. The desired number of iteration subsystems can be reinserted into the subsystem thusly:

```
add_block([cordic_pe{1} '/CORDIC PE1'], [cordic_pe{1} '/CORDIC
    PE' int2str(i)], 'ii', int2str(i-1), 'pe_nbits',
    'pe_nbits', 'pe_binpt', 'pe_binpt', 'pipeline', ['pipeline(1, '
    int2str(i) ' )'], 'position', [150+(i-1)*125, 70, 205+(i-1)*125,
    130]);
```

where mask parameters `pe_nbits` and `pe_binpt` define the bit precision of each processing element. The subsystem blocks are connected using the following function:

```
add_line(cordic_pe{1}, ['CORDIC PE' int2str(i-1) '/' int2str(j)],
    ['CORDIC PE' int2str(i) '/' int2str(j)], 'autorouting', 'on');
```

The Xilinx Reference Blockset contains several examples of subsystems that make use of this API, including several variants on the CORDIC algorithm. For more information on the syntax of the API, refer to the Simulink documentation.

Note: if you are embedding commands that modify the diagram into the mask initialization code of a subsystem (as in the CORDIC demos), it is useful to enclose the code into a MATLAB “try” “catch” clause, and in the catch clause to call `disp(lasterr)`.

Note #2: To make it more convenient to set the parameters on blocks, Xilinx provides a MATLAB function that will describe all of the parameters that can be set on a block and what values they take. To use this function, select a copy of the block you are interested in parameterizing, and then run the following function (shown here with the output it will produce for the Xilinx Accumulator Block):

```
>> xlShowParams('xbsIndex_r4/Accumulator')

-----
Xilinx Accumulator Block

Parameters    Valid Settings
-----
operation:    'Add' or 'Subtract'
n_bits:       <edit field>
overflow:     'Wrap' or 'Saturate'
              or 'Flag as error'
scale:        '1' or '1/2' or '1/4' or '1/8'
              or '1/16' or '1/32' or '1/64'
              or '1/128' or '1/256'
rst:          'on' or 'off'
hasbypass:    'on' or 'off'
en:           'on' or 'off'
-----

Examples of set_param usage:
set_param(blk, 'operation', 'Add')
set_param(blk, 'operation', 'Subtract')
set_param(blk, 'n_bits', 'f(x)')
set_param(blk, 'overflow', 'Wrap')
set_param(blk, 'overflow', 'Saturate')
set_param(blk, 'overflow', 'Flag as error')
set_param(blk, 'scale', '1')
set_param(blk, 'scale', '1/2')
set_param(blk, 'scale', '1/4')
set_param(blk, 'scale', '1/8')
set_param(blk, 'scale', '1/16')
set_param(blk, 'scale', '1/32')
set_param(blk, 'scale', '1/64')
set_param(blk, 'scale', '1/128')
set_param(blk, 'scale', '1/256')
set_param(blk, 'rst', 'on')
set_param(blk, 'rst', 'off')
set_param(blk, 'hasbypass', 'on')
set_param(blk, 'hasbypass', 'off')
set_param(blk, 'en', 'on')
set_param(blk, 'en', 'off')

>>
```

Using ChipScope Pro Analyzer for Real-time Hardware Debugging

This tutorial demonstrates how to connect and use the Xilinx Debug Tool called ChipScope Pro within Xilinx System Generator. The integration of ChipScope Pro in the System Generator flow allows real-time debugging at system speed. By inserting a ChipScope block into your System generator design, you can debug and verify all the internal signals and nodes within your FPGA. After reviewing some characteristics of the ChipScope Pro Debug tool, this tutorial walks the user through the steps of adding the ChipScope block to a simple Simulink model, running it on a hardware platform and probing internal signals.

The following topics are discussed:

- ChipScope Pro Overview
- ChipScope within System Generator
- Real-Time Debug with the ChipScope Pro Analyzer
- Importing data back into MATLAB workspace from ChipScope

Software Requirements: Refer to the Software Prerequisites page.

Hardware Requirements: Any hardware demo board that has a JTAG port (the XtremeDSP Development kit is used in this tutorial).

ChipScope Pro Overview

The increasing density of FPGA devices has rendered attaching test probes to these devices impractical. The ChipScope™ Pro tools integrate key logic analyzer hardware components with the target design inside Xilinx Virtex™, Virtex-E, Virtex-II, Virtex-II Pro™, Spartan™-II, Spartan-IIE and Spartan-3 devices. The ChipScope Pro tools communicate with these components during system operation and in effect provide the designer with a logic analyzer for nodes inside the Xilinx FPGA. ChipScope gives you a deep trace memory, fast clock speeds and multiple trigger options, which can vary in complexity. You can easily capture and view signal activity inside your FPGA without having to dedicate critical logic space, come up with complex capture schemes, or allocate additional I/O pins. Data samples are captured based on user-defined trigger conditions and stored in internal block memory. All control and data transfer is done via the JTAG port eliminating the need to drive data off-chip using I/O pins.

Please refer to the following Web page for further details on ChipScope Pro:

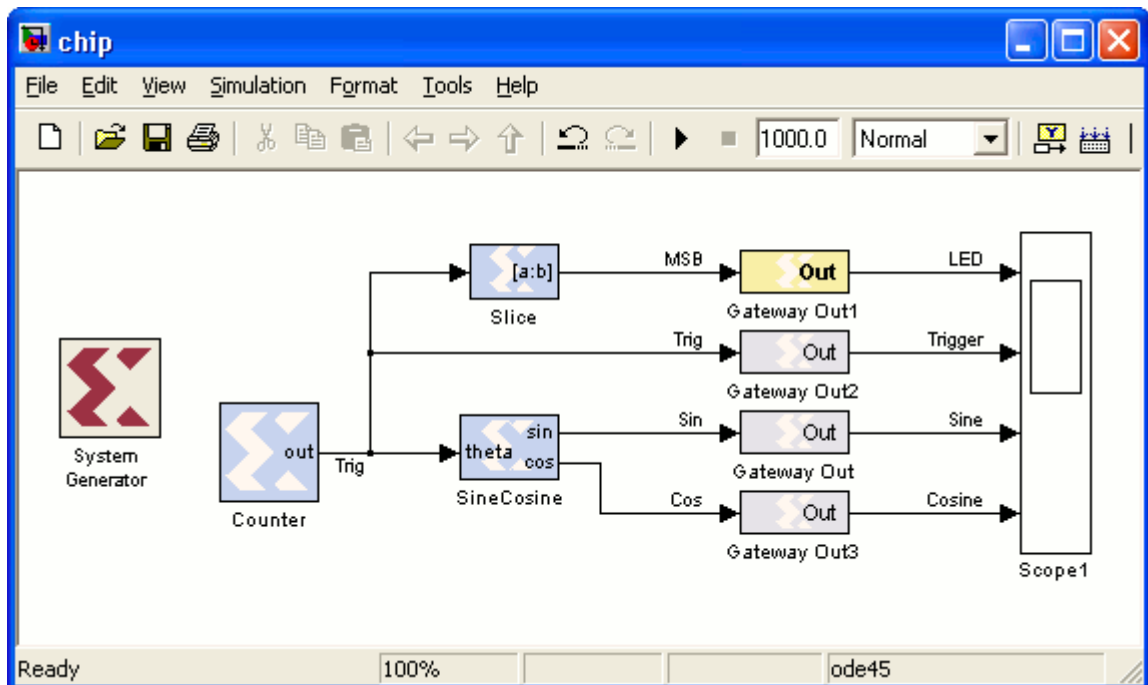
http://www.xilinx.com/ise/optional_prod/cspro.htm

ChipScope in System Generator

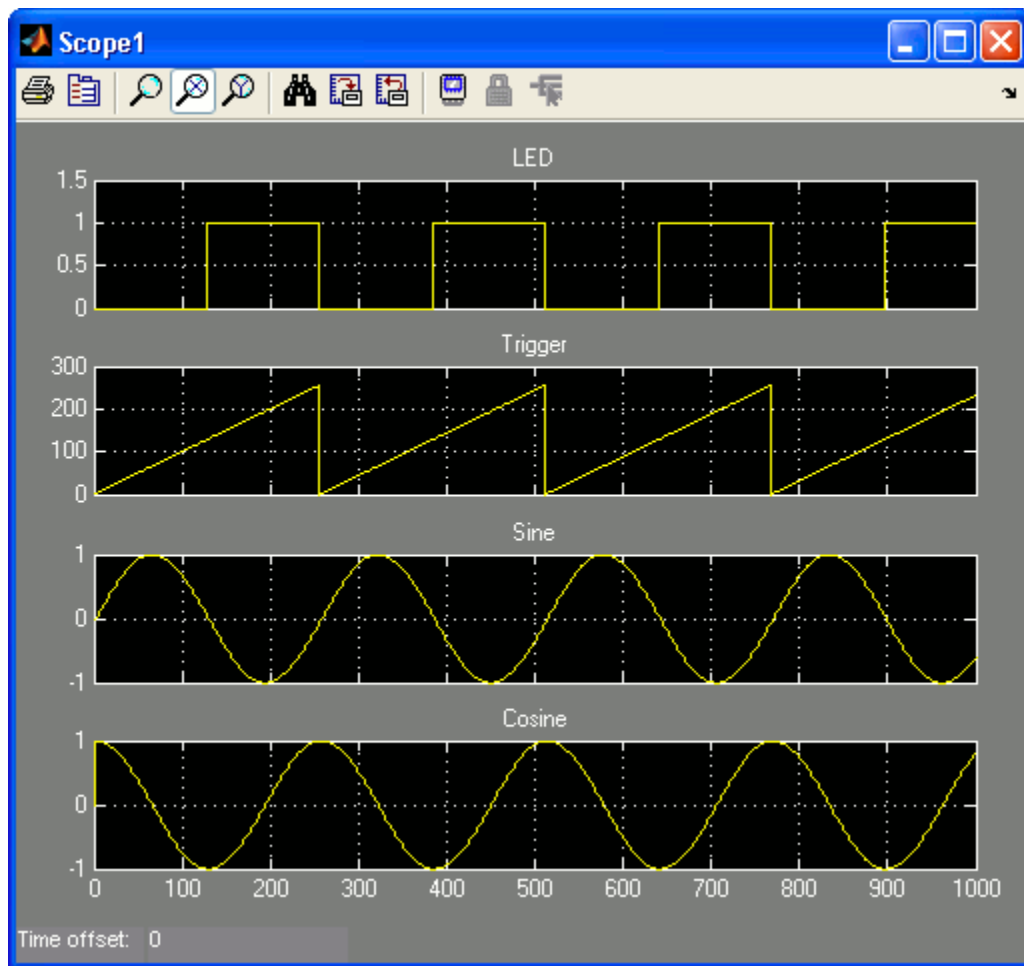
This example shows how to modify a Simulink model to integrate the ChipScope block and to select the data to be captured and viewed for debugging. The steps are the following:

1. From the MATLAB console, change the directory to `<path_to_sysgen>/examples/chipscope`. The default path to System Generator is `$MATLAB/toolbox/xilinx/sysgen`. The following files are located in this directory:
 - `chip.mdl` – Your working model.
 - `chip_soln.mdl` – Solution model, including the ChipScope block.

- `osc_clock_2v80.bit` – Bitstream to program the XC2V80 device on the XtremeDSP development kit. This device is used for clock management and is required for proper operation.
2. Open `chip.mdl` model from the MATLAB console.
This model represents a simple sine/cosine table driven by an 8-bit counter. Both sine and cosine functions have been selected, enabling you to probe, and later plot both waveforms.
 3. The 8-bit counter counts modulo 255. It is to be used to trigger ChipScope. The most significant bit is extracted with a slice block and drives an LED on the XtremeDSP board.



4. Simulate the model by clicking on the “Start simulation” Icon  At this point, without modifying the model, you should be able to see the following plot.



- The first plot represents the most significant bit of the 8-bit counter. The MSB becomes 1 when the counter output is within the range of 128 through 255.
- The second plot represents the full output of the counter.
- The third plot shows the sine. If you zoom in at the beginning of time, you will notice a 2 clock-cycle delay due to the pipelining implemented in the SineCosine table.
- The fourth output represents the cosine. This output has the same 2 clock-cycle delay as the Sine Wave.

5. Integrate ChipScope in the Simulink model.

The ChipScope block can be found in the Simulink Library Browser in the Xilinx Blockset, under the Tools library. While holding down the left mouse button, select the ChipScope block and drag it into the open chip Simulink model.

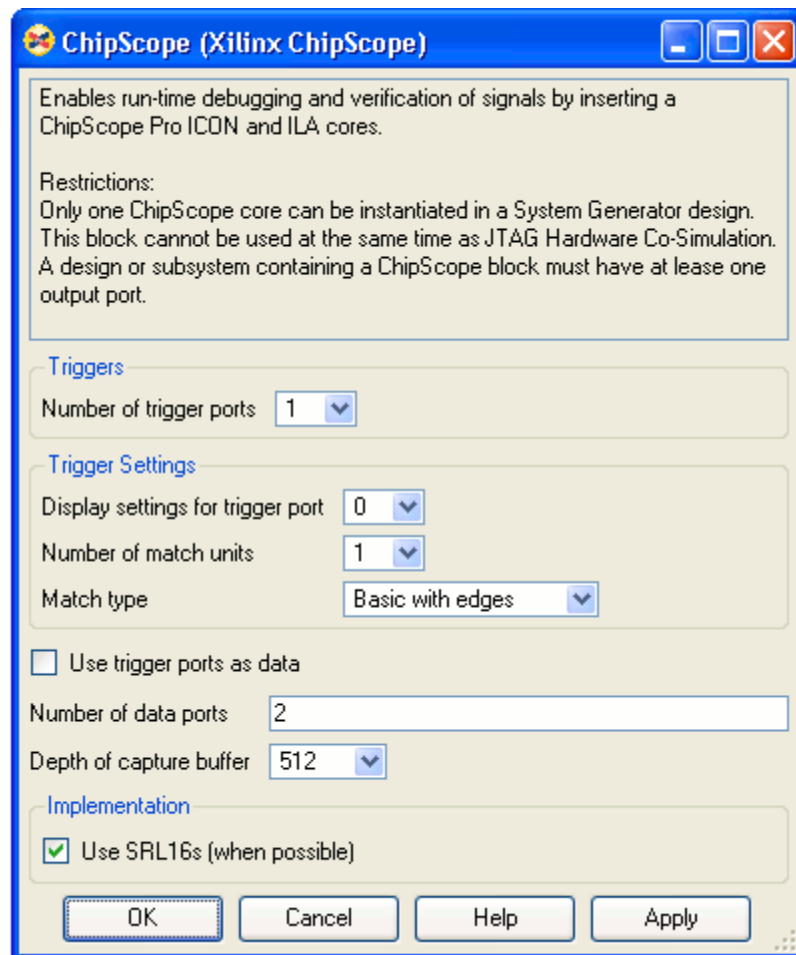
6. Double click on the ChipScope block in order to set the following parameters:

- **Number of trigger ports:** Multiple trigger ports allow a larger range of events to be detected and can reduce the number of values that must be stored. Up to 16 trigger ports can be selected. In this example, we only use one.
- **Display settings for trigger port:** For each trigger port, the number of match units and the match type need to be set. The pulldown menu displays options for a particular trigger port. For N ports, the display options for trigger port 0 to N-1 can be shown. In this example, we have one Trigger port, named Trig0. This option should therefore be set to 0.

- **Number of match units:** Using multiple match units per trigger port increases the flexibility of event detection. One to four match units can be used in conjunction to test for a trigger event. In this example, this option should be set to 1 since we are only checking for one condition (i.e., the 8-bit counter value). You will set the trigger value at run-time in the ChipScope Pro Analyzer.
- **Match type:** This option can be set to one of the following six types:
 1. Basic: performs = or <> comparisons
 2. Basic With Edges: in addition to the basic operations high/low, low/high transitions can also be detected
 3. Extended: performs =, <>, >, <, <=, >= comparisons
 4. Extended With Edges: in addition to the extended operations, high/low, low/high transitions can also be detected.
 5. Range: performs =, <>, >, >=, <, <=, in range, not in range comparisons
 6. Range With Edges: in addition to the range operations, high/low, low/high transitions can also be detected.

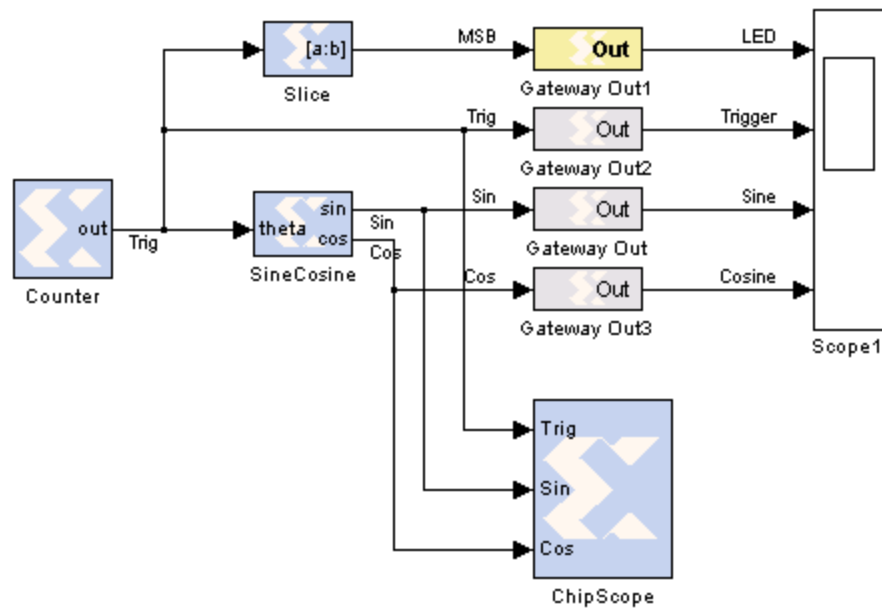
In this example, set the Match Type to **Basic with edges**
- **Number of data ports:** Up to 256 bits can be captured per sample. This means that the sum over all ports of the bits used per port must be less than or equal to 256. System Generator propagates the data width automatically; therefore, only the number of data ports needs to be specified. In this example, we want to view the sine and cosine, hence we enter 2.
- **Depth of capture buffer:** The depth of the capture buffer is a power of 2, up to 16384 samples for Virtex-II, Virtex-II Pro, and Spartan-3 device families, and 4096 for Virtex, Virtex-E, Spartan-II and Spartan-IIE device families. In this example, set the depth to 512.

After parameterization the ChipScope GUI should look like the following:



7. Connecting the ChipScope Block

The signal used to trigger ChipScope is the counter output. The two buses that we want to probe are the sine and cosine from the Sine/Cosine table. Connect the signals appropriately as shown on the following screen shot:



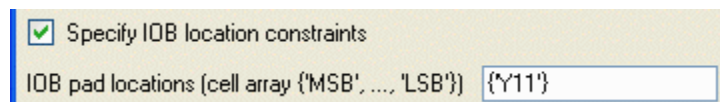
Note that the names of the ports on the ChipScope block are specified by names given to the signals connected to the block, e.g. Sin and Cos.

8. Location Constraints

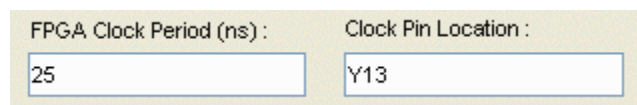
Now that the design is fully implemented and simulates correctly, the next step is to prepare it for connection to our hardware target. Although it can work on any hardware platform, the process is described for the XtremeDSP Development kit.

Two pins need to be locked down in this design: The LED and the clock pin.

- **LED Pin:** Double click on the Gateway Out1 block, select Specify IOB Location constraint and type in {'Y11'} (note the need for single quotes).



- **Clock Pin:** Double click on the System Generator block, set the clock period to 25ns and the clock pin location to Y13.



If you are using a different board, pin location should be modified appropriately.

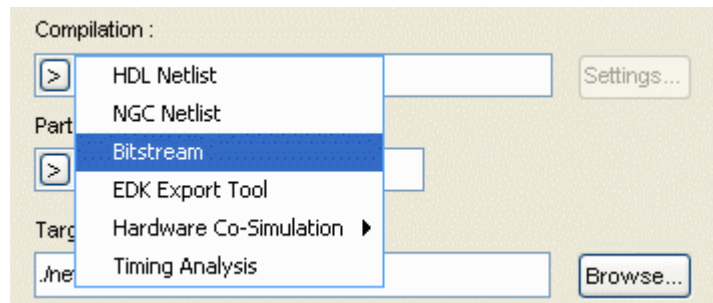
9. System Generator GUI settings

The last two parameters that should be updated before generating a bitstream are the target device and the compilation target.

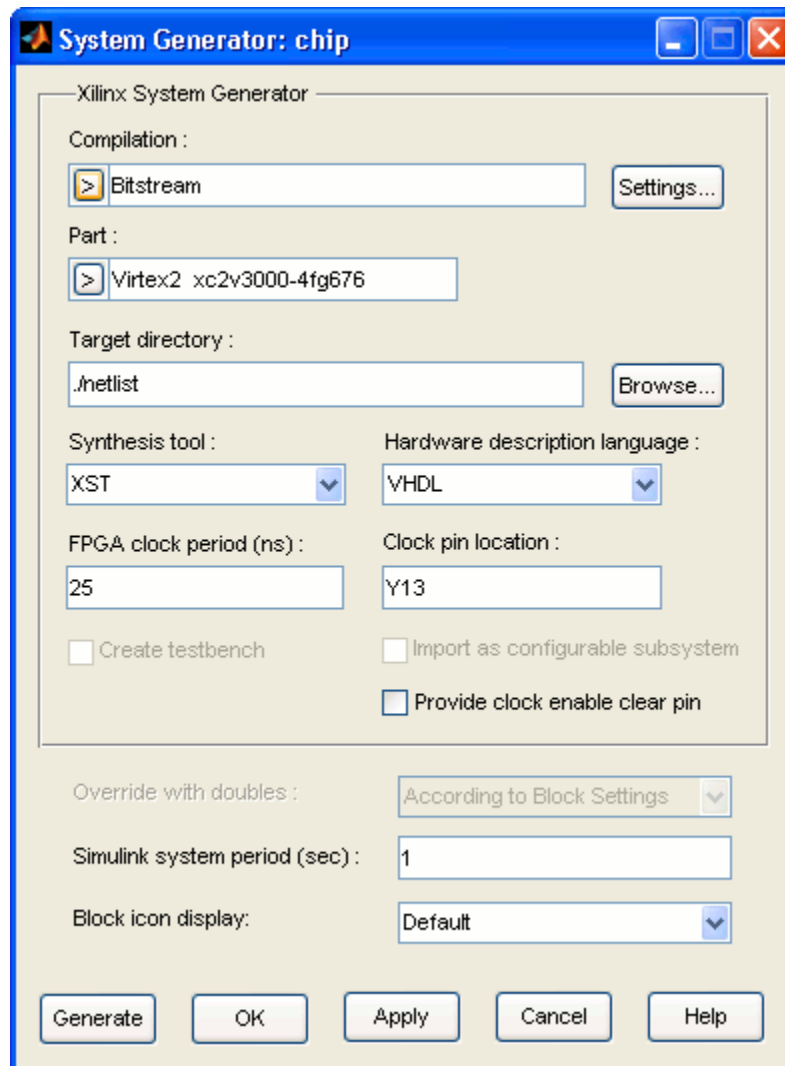
- Double click on the System Generator token and select the following part:

Virtex2 xc2v2000-4fg676 or xc2v3000-4fg676 depending on which part you have on your board

- Set the Compilation target to Bitstream.



- Double check that your System Generator parameters match the ones shown on the following screen shot:



10. Bitstream Generation

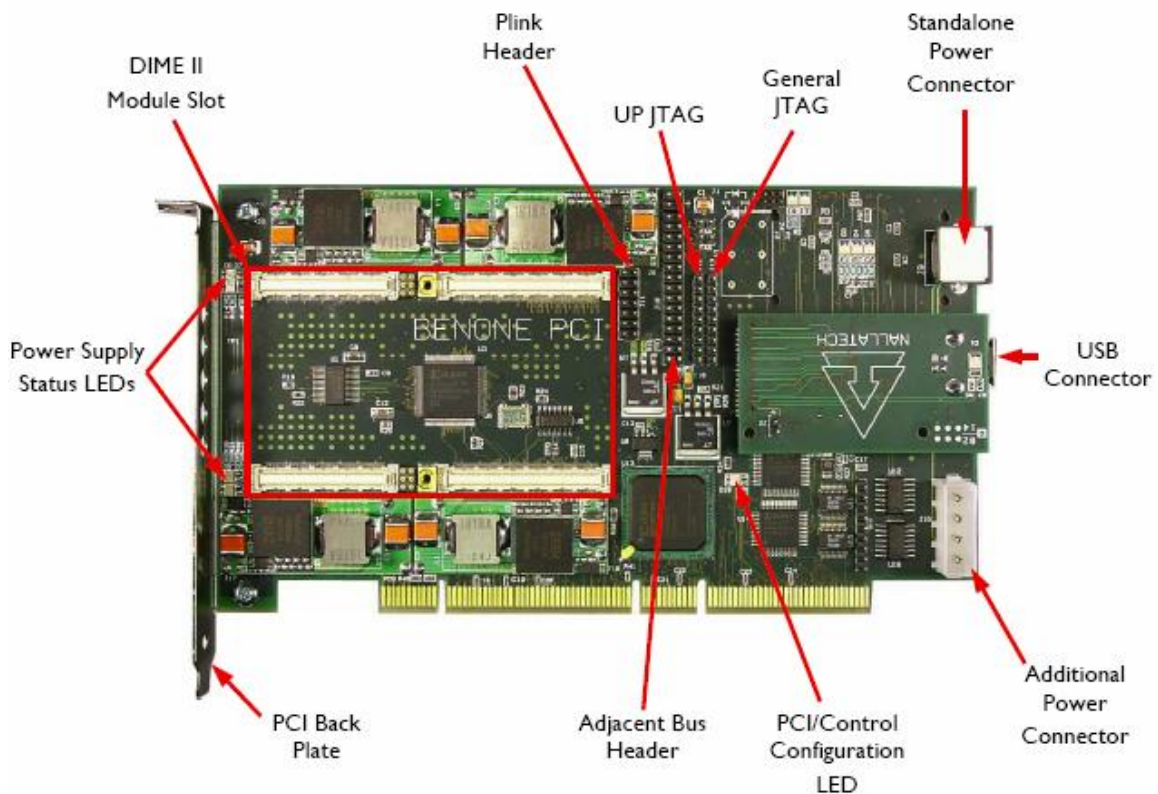
Xilinx System Generator software automatically calls both the Core Generator and ChipScope generator to create the netlist and cores. In addition, when the Bitstream target is selected, a configuration bitstream is created.

- Create a bitstream by pressing the **Generate** button on the System Generator GUI.
- The Core Generator is automatically called to generate the Sine/Cosine table and Counter netlists. ChipScope Generator is called to create an Internal Logic Analyzer (ILA) core and an ICON core to communicate with the ChipScope Pro software via the JTAG port.

Real-time Debug

The next step is to run the design on the XtremeDSP development kit and view the probed outputs with the ChipScope Pro Analyzer.

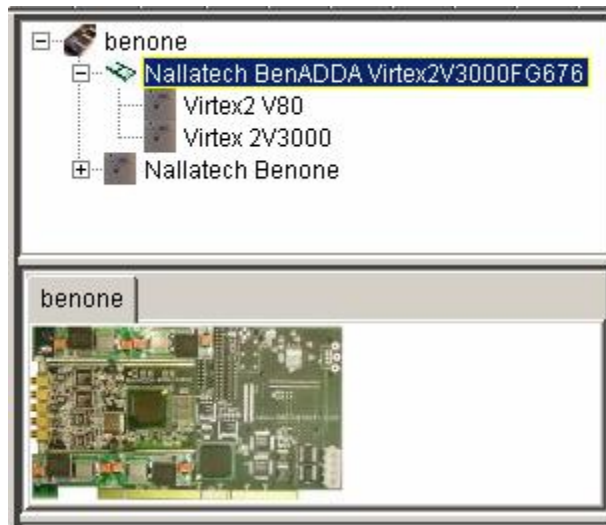
1. Connect Parallel Cable IV to the General JTAG (J13) header of the XtremeDSP development kit. The VCC pin is the pin closest to the top edge of the board, furthest away from the PCI connector. It is also necessary to connect it to the USB port for power up.



2. Open Fuse to power up the board

With the XtremeDSP kit, it is first necessary to power up the board with the FUSE software.

- Select **Start->Programs-> FUSE -> Software -> FUSE Probe**
- From FUSE, select **Card Control -> Open Card**. Then select USB for the interface and click on Locate Card. At this point it should find the BenOne card and you should see the following on the top left corner of your FUSE window.



If you have any issues with this procedure, please refer to the XtremeDSP development kit User's Guide.

3. Open ChipScope Pro Analyzer(Refer to Software Prerequisites page for details on the version to use)

- Open JTAG Chain by clicking on the following Icon , or by clicking on **JTAG Chain -> Xilinx Parallel Cable** and selecting **Xilinx Parallel IV Cable**.
Note the index for the Virtex-II devices available on the XtremeDSP kit: xc2v2000 or xc2v3000 is index 3, and xc2v80 is index 4.

4. Configure the FPGAs

- Under the New Project Window, right click on **Device 3** and select **Configure-> Device 3**. At this point, you need to look for the bitstream which was generated in step 10 of the previous section. Select **New File** and scroll to your project directory:
<path_to_sysgen>/examples/chipscope/netlist/chip_cw.bit. After configuration, the status window at the bottom of the ChipScope Analyzer should reflect that one ChipScope core was found in the JTAG chain.
- Similarly, select Device 4 and Configure it with the osc_clock_2v80.bit file provided in the ChipScope project directory. This second bitstream is used to program the clock driver FPGA on the board.

5. Import ChipScope Project File

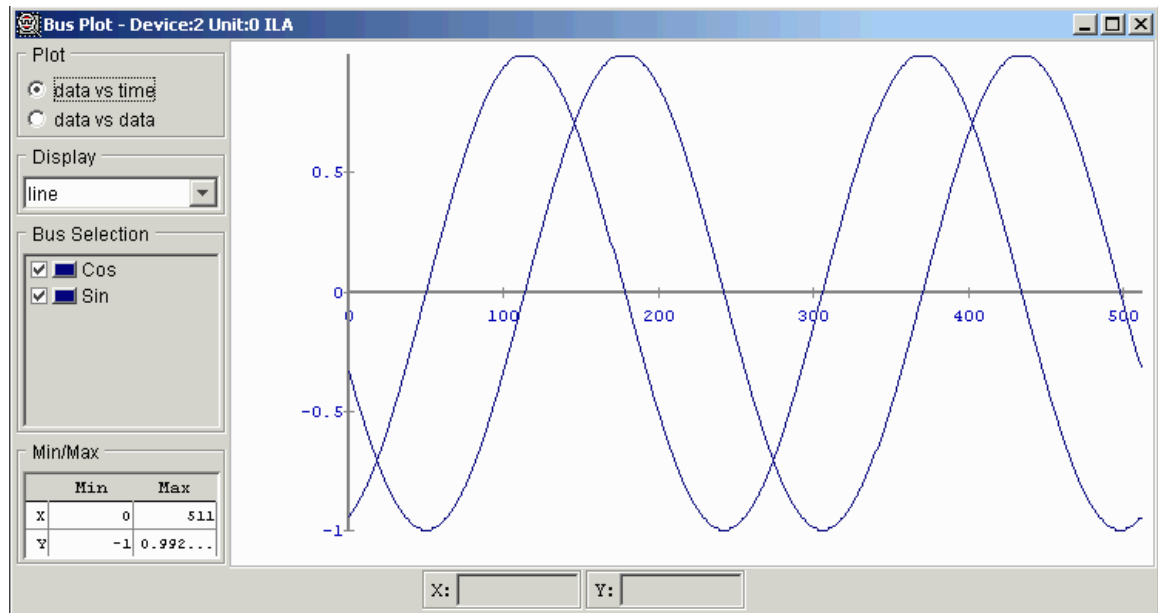
System generator creates a project file for ChipScope in order to group data signals into buses. A bus is created for each data port so that it can be viewed in the same manner (sign and precision) in which it was viewed in the Simulink environment.

- Load this project file by going under **File-> Import -> Select New File**, and select <path_to_sysgen>/examples/chipscope/netlist/temp/chip_chipscope.cdc.

6. Plot the Sine Waves

In the New Project window, under **Device 3 -> Unit 0 ILA**, double click on **Bus Plot**.

A Bus Plot window appears. Select Cos and Sin in the Bus Selection section, and then arm the trigger by hitting the  button. Since we have not yet set any trigger conditions, values are captured immediately. Both the sine and cosine appear as shown below. You can change the display option to represent the waveforms with points, lines, or both.



7. Setup Trigger

In the Trigger Setup window, change the current XXXX-XXXX value with all 0000-0000. Once the counter hits 0, ChipScope starts capturing values. Earlier, we setup the buffer to 512, so 512 data points can be visualized in ChipScope.

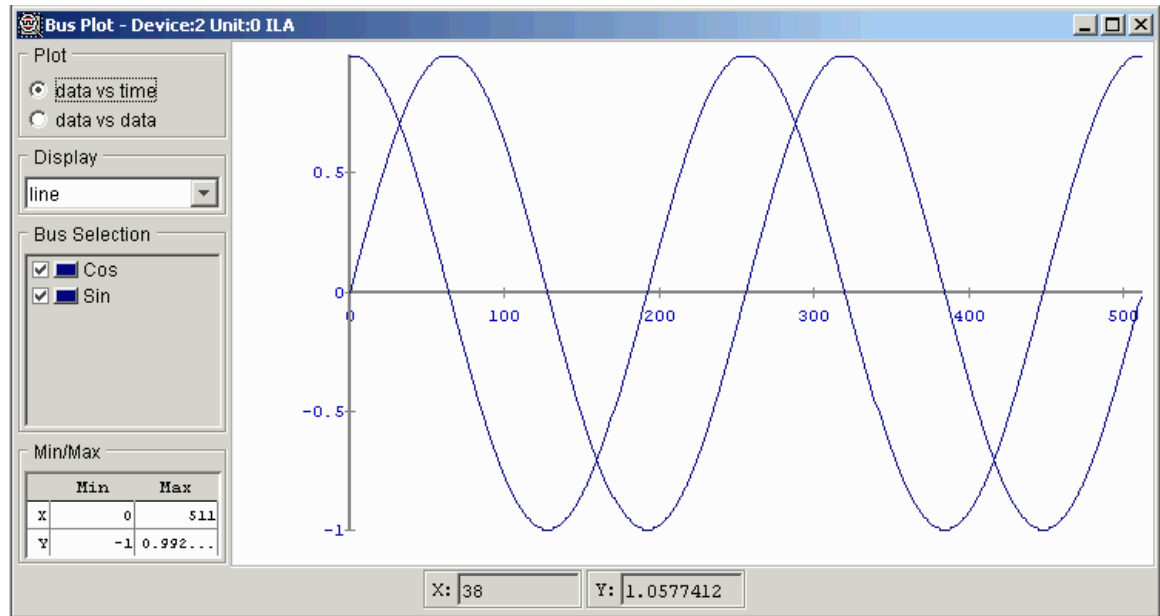
Re-capture the data by hitting the  button.



Trigger Setup - Device:3 Unit:0 ILA

Match Unit	Function	Value	Radix	Counter
+M0:TRIG0	==	0000_0000	Bin	exactly one clock cycle

Again, you see a 2 clock delay at the beginning of time on the sine wave due to the 2 clock latency through the SineCos Look Up Table. Modify the trigger value to 0000-0010 (decimal 2) and re-capture the data. Now the sine and cosine start respectively at 0 and 1.

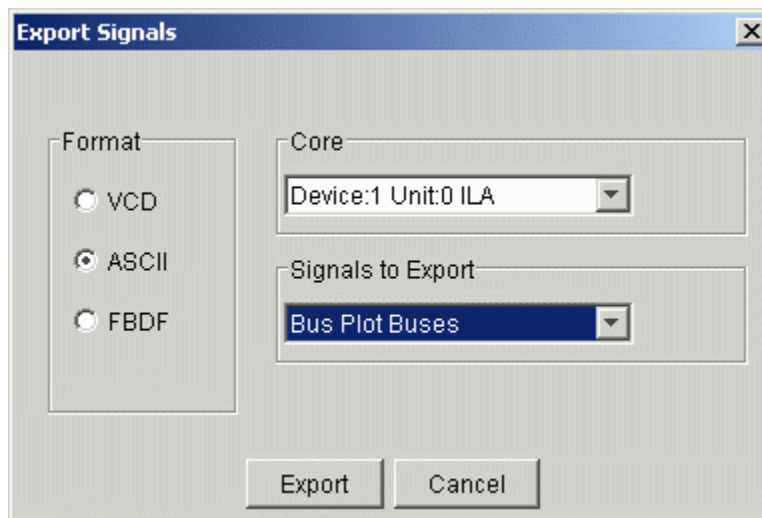


Importing Data Into MATLAB Workspace From ChipScope

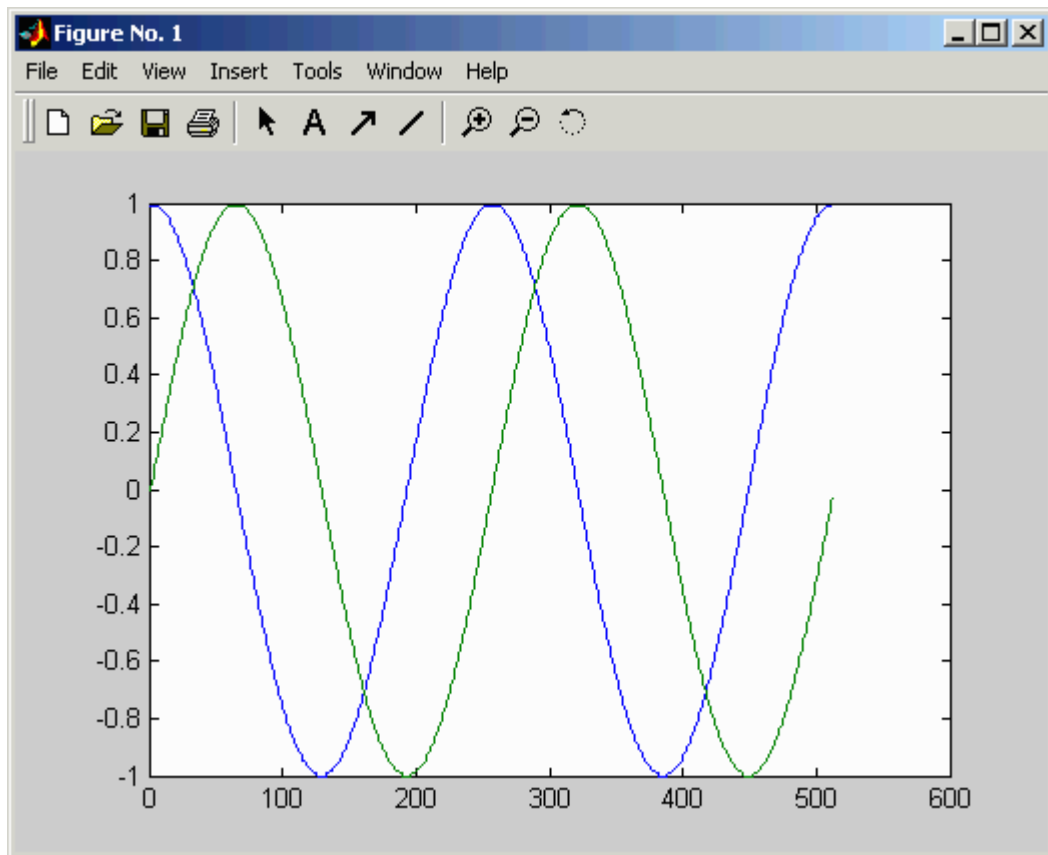
Now we can export the data captured by ChipScope back into the MATLAB workspace.

1. Export data from ChipScope Pro Analyzer

- **Select File -> Export** option from within ChipScope Pro Analyzer. Select ASCII format and choose Bus Plot Buses to export. Press the Export button and save the file as `sinecos.prn`.



2. Start MATLAB and change the current working directory to the location where you saved sinecos.prn.
 - Type `xlLoadChipScopeData('sinecos.prn');` This loads the data from the .prn file into the MATLAB workspace. In the workspace there are two new arrays named Sin and Cos.
3. You can plot the values using the MATLAB plot function.
 - Type: `plot(1:512, Sin, 1:512, Cos)` and the following plot is generated:



Generating Multiple Cycle-True Islands for Distinct Clocks

System Generator's shared memory interfaces allow you to implement designs that are driven by multiple clock sources. These multi-clock designs may employ a combination of distinct clocks and derived clock enables to implement advanced clocking strategies completely within a single design environment. This section describes how to implement multi-clock designs in System Generator through discussions of the following topics:

- Applications that benefit from multiple clocks;
- Using hierarchy to partition a System Generator model into two or more clock domains;
- Using shared memories to cross clock domains;
- Simulating and netlisting multiple clock designs;
- Wiring multiple clock domains together using the Xilinx Multiple Subsystem Generator block.

A step-by-step example is provided to help clarify the topics listed above. Although the example uses two clocks, the concepts presented here can be extended so that System Generator designs requiring any number of clock sources can be constructed using similar techniques.

Before continuing with the example, you may want to familiarize yourself with standard System Generator clocking terminology and implementation methodologies. This information is covered in-depth in the Timing and Clocking section of the User Guide. In general, System Generator designs are driven by a single, system clock source. Multi-rate design portions are handled using clock enables derived from the system clock source. It is possible, however, to use System Generator to implement designs that are driven by distinct clock sources.

Broadly speaking, the approach is the following:

Divide the design into several subsystems, each of which is to be driven by a different clock. In the example, we call these subsystems *asynchronous clock islands*. Xilinx shared memory blocks should be used as bridges that communicate between these clock islands. Once the design is partitioned, the Xilinx Multiple Subsystem Generator block may be used to translate the design into hardware that uses multiple distinct clock sources.

Multiple Clock Applications

A common application for multiple clock domains is for interfacing different pieces of external hardware that operate at different clock rates. For example, you may need to provide a set of I/O registers to a microprocessor, and the processor must be able to read and write these registers synchronous to its own clock. You may get data from a clock/data recovery unit and need to resynchronize the data to your local clock domain. You may need to feed data to a digital-to-analog converter that must be running at a precise sample rate which is different from your system clock.

Another important application for multiple clock domains is in employing a high-speed processing unit. Let us take an example of an interpolating FIR filter. The filter gets symbol data from an external unit, and the filter needs to take the symbols and perform a 4X interpolation that creates four output samples for each input symbol. The output samples are fed to a digital-to-analog converter (DAC) that is clocked at the sample rate.

The FIR filter may be clocked at any of several rates. It may be clocked at the symbol rate, and on each cycle it must create four samples which will then be fed to the DAC at the sample rate. This highly parallel implementation has large hardware resource requirements and would only be employed if the sample rate were very fast. An alternative approach is to clock the FIR filter at the sample rate,

creating one sample per cycle. This scenario takes an intermediate amount of hardware and would be used for intermediate sample rates. If the sample rate is slow, the FIR filter may be clocked at a rate several times faster than the sample rate, perhaps by means of a DCM that multiplies the sample-rate clock. In this way the multiplier-accumulator units of the FIR filter may be reused several times during the calculation of each sample output, requiring the least amount of hardware. This last method would use a symbol-rate clock domain, a high-speed processing clock domain, and a sample-rate clock domain.

A good FPGA design practice is to have each resource in the FPGA device operating at the highest possible rate to optimize hardware usage. In general, it is best to use a single clock domain when possible and to use clock enables to gate slower circuitry, creating multicycle paths. The drawback to this technique is that it increases power consumption and may make it difficult to route the high-speed clock enable. As a result, separate domains for high-speed processing are preferable in some instances. Also, it may not be possible to avoid dealing with different clock domains when dealing with asynchronous data inputs and outputs.

Clock Domain Partitioning

Partitioning a multiple-clock design into multiple domains is an important aspect of FPGA design. System Generator uses design hierarchy to support clock domain partitioning. More specifically, when a design uses multiple clock domains, the logic associated with each distinct clock domain should be grouped together in a Simulink subsystem.

The subsystems, or in this case, synchronous islands, are cycle-true in the sense that the hardware that is generated for an island is faithful to the Simulink behavior of the island model. The notion of bit and cycle accuracy is preserved only within the individual synchronous islands. The end model containing the synchronous islands is not necessarily cycle-true, because it drives the islands with asynchronous clocks. Although System Generator and Simulink are able to simulate the design using ideal clock sources, the complexities involved with asynchronous clocking systems can result in discrepancies between the software simulations and hardware realizations.

The advantages to partitioning a design using subsystems are manifold:

- The physical clock lines are abstracted away from the block diagram;
- Cross-domain transfers are well-defined and can be handled with metastable-safe blocks from the Xilinx Blockset;
- Because the domains are well-defined, System Generator can accurately produce timing constraints for the synchronous islands.

The abstraction level of System Generator reduces the risk that users will perpetrate some of the more common design errors. These include:

- *Gated Clocks*: because the clocks in System Generator are inferred during hardware generation, it is not possible to connect non-clock lines to clock inputs (i.e., gated clocks).
- *Asynchronous Clears*: because the asynchronous resets in System Generator are inferred during hardware generation, it is not possible to explicitly clear synchronous logic using the asynchronous reset, which often results in timing problems.
- *Inferred Latches*: latches will not be generated from System Generator designs.

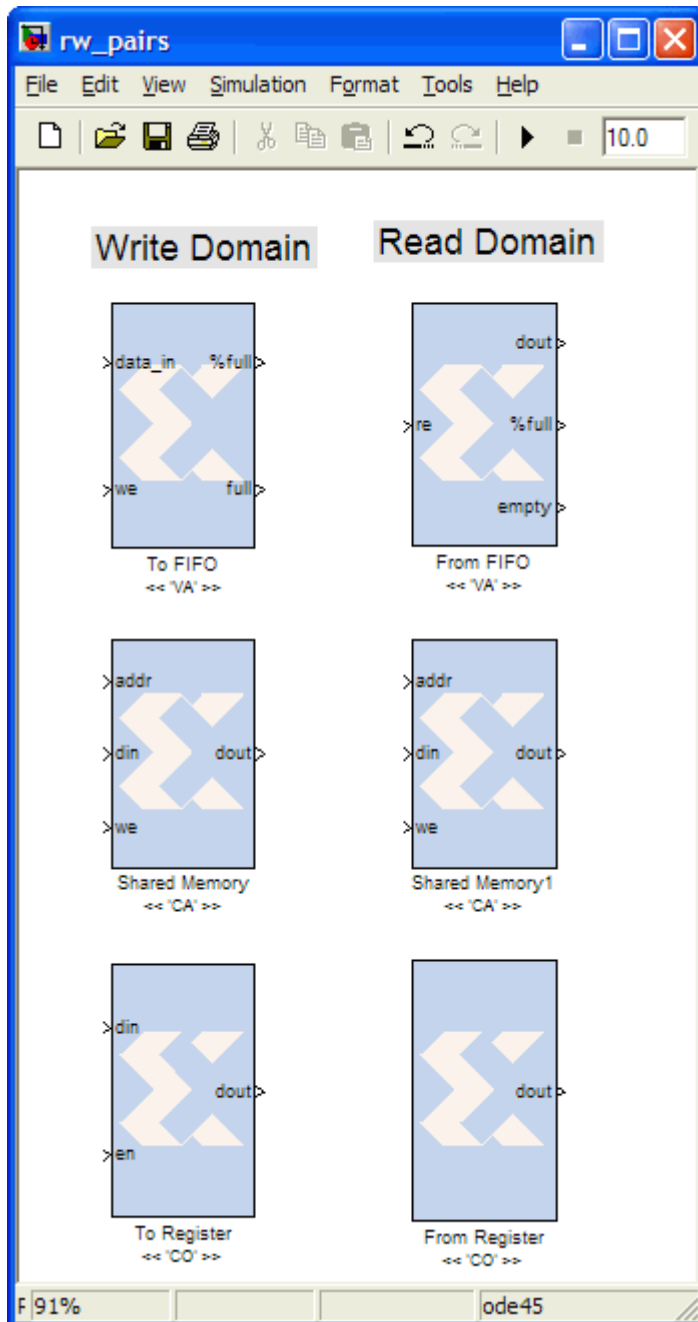
Crossing Clock Domains

System Generator shared memory blocks should be used whenever it is necessary to cross clock domains. The tool provides several blocks for transferring data across clock domains, each of which is

available in the Xilinx Shared Memory library:

- Shared Memory
- To FIFO / From FIFO
- To Register / From Register

When these shared memory blocks are used to cross clock domains, each set should be split into a matched pair.



The To FIFO block is put in the domain in which it is to be written. The From FIFO is put in the domain in which it is to be read. The two blocks are linked by the name of the Shared memory name parameter. The FIFO is implemented in hardware using the Xilinx FIFO Generator core. Using

FIFO blocks is the safest and easiest-to-use of the three blocks which cross domains and is the best for high-bandwidth, sequential data transfers.

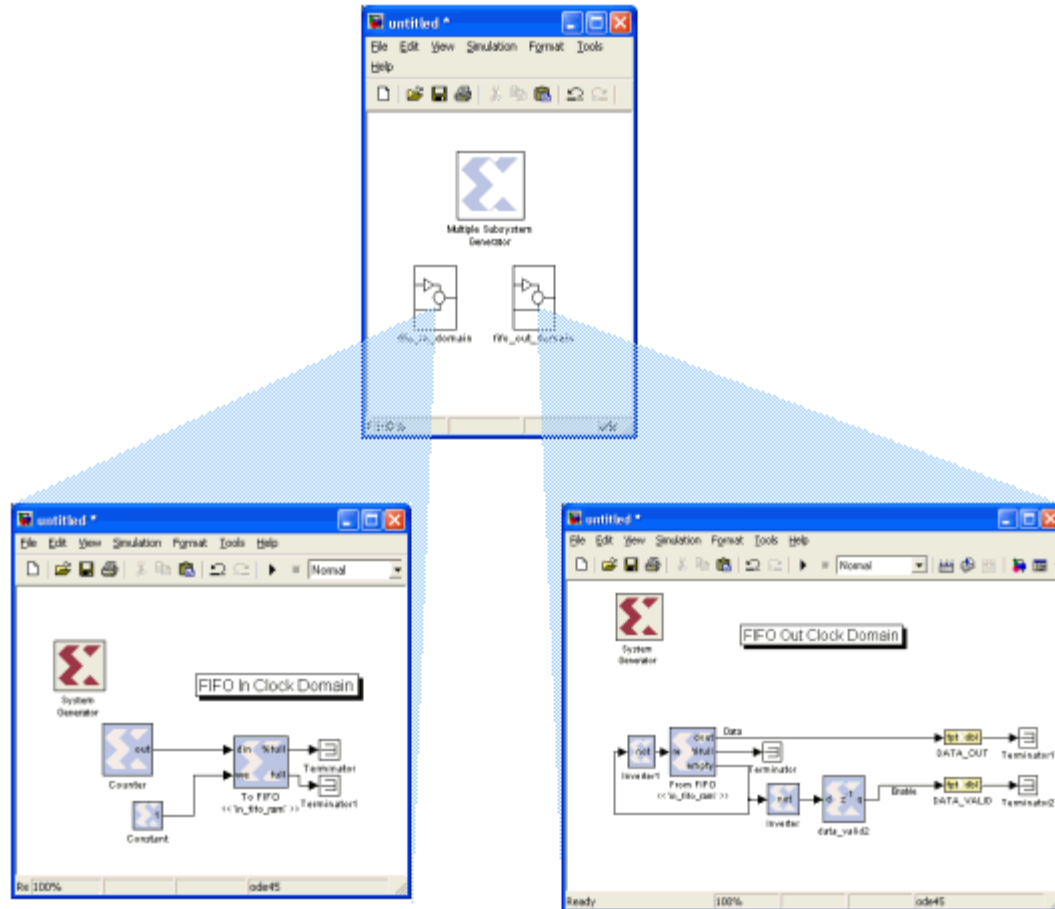
A pair of **Shared Memory** blocks is implemented as embedded Xilinx dual-port block RAM core. The two blocks are linked by the name of the shared memory object. Each member of the pair resides in a different domain. Because the RAM is a true dual-port, each domain may write to the RAM. Care must be taken, by means of semaphores or other logic, to ensure that two writes or a read and a write to the same address do not happen simultaneously. For example, if domain A writes to a memory location at the same time that domain B is reading from it, the data read may not be valid. The shared memory is implemented as a using Xilinx Dual Port Block Memory core to ensure that large memories are efficiently mapped across multiple BRAMs.

The **To Register** is put in the domain in which it is to be written, and the **From Register** in the domain from which it is to be read. The two blocks are linked by the name of the shared memory. The **To Register** may also be read synchronously in its own domain. The register may be of variable width and will synthesize as flip-flops. A 1-bit **To/From Register** pair will synthesize as a single flop.

Note: crossing domains in this manner can be unsafe, and requires the use of metastability-reducing synchronization flops and semaphores for multiple-bit transfers. This technique should only be used when the hardware pitfalls are well-understood.

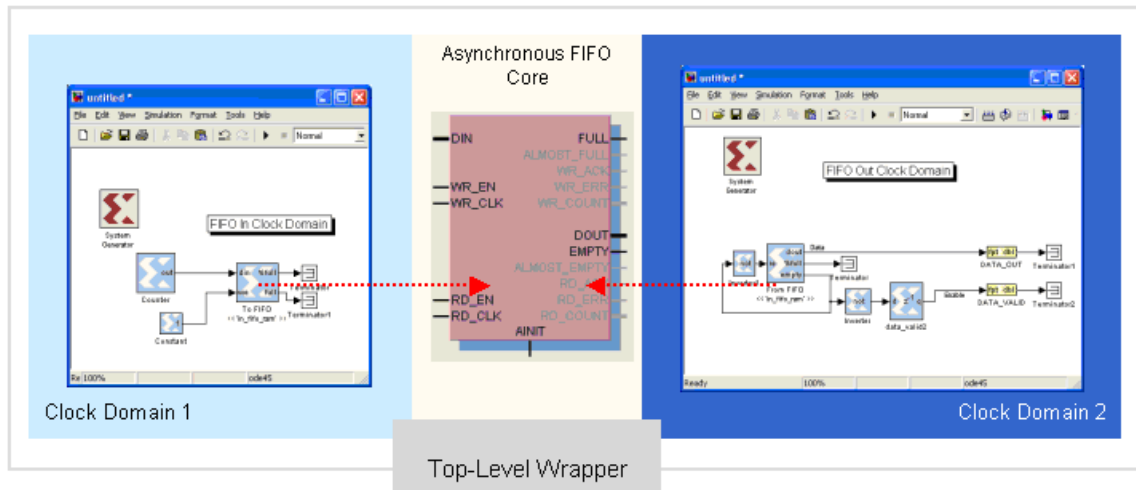
Netlisting Multiple Clock Designs

Each clock domain should have its own subsystem in a System Generator design. The diagram below shows a two-domain design. The top-level block contains the Multiple Subsystem Generator block and two subsystems which each comprise a clock domain. Each subsystem has a System Generator block that sets the system clock period for that clock domain.



The diagram below illustrates the concept of putting domain-crossing blocks into their own subsystem. When a multiple-domain design is netlisted, System Generator does the following:

- Creates an HDL file for Domain 0 (on the left), excluding the To FIFO block, and calls the netlister to create a black-box netlist delivered as an NGC file;
- Creates an HDL file for Domain 1 (on the right), excluding the From FIFO block, and calls the netlister to create a black-box netlist delivered as an NGC file;
- Invokes the Xilinx CORE Generator to produce a core for the FIFO block (middle);
- Creates a top-level HDL wrapper that instantiates three black components.



Step-by-Step Example

This example shows how design hierarchy can be used to partition a System Generator design into multiple asynchronous clock islands. The example also demonstrates how Xilinx Shared Memory blocks may be used to communicate between these islands. Lastly, the example describes how the Multiple Subsystem Generator block can be used to netlist the complete multi-clock design.

1. From the MATLAB window, change directory to `<path_to_sysgen>/examples/multiple_clocks/`.
2. Open the `two_async_clks` model from the MATLAB command window, and save it into a temporary directory of your choosing.

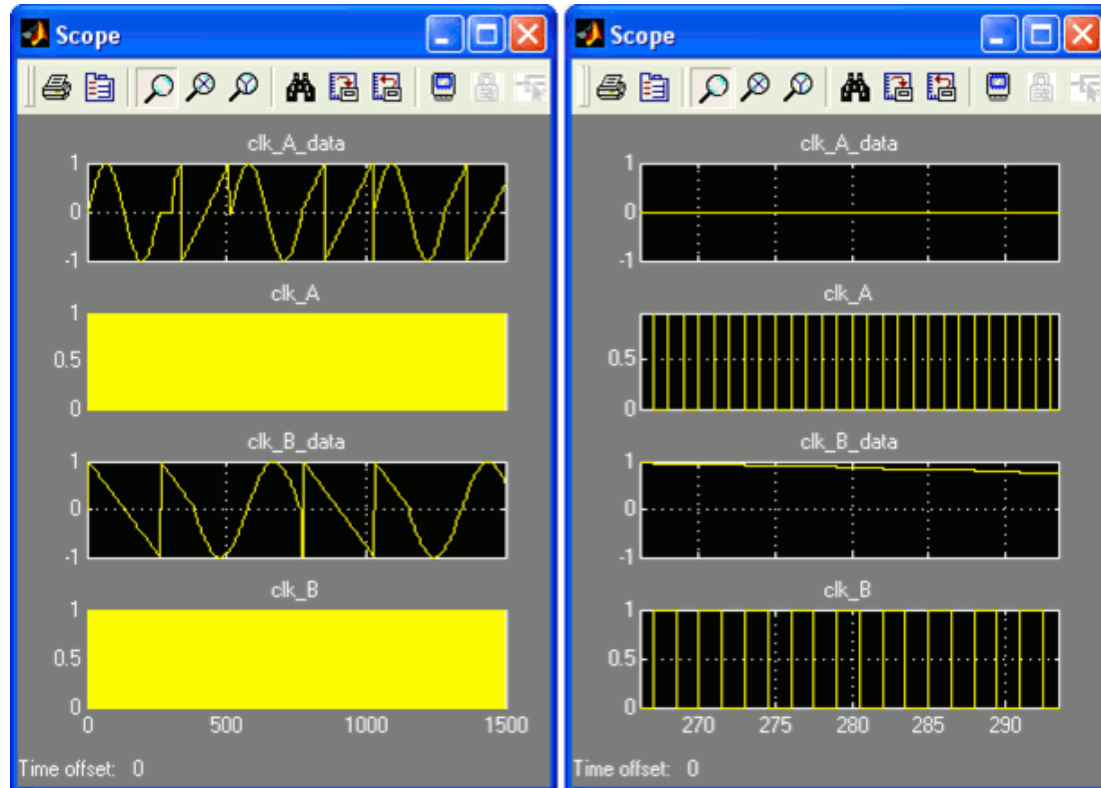
Subsystem hierarchy is used in the example to partition the design into two synchronous clock domains, to which we refer as domains A and B, that are internally synchronous to a single clock, but asynchronous relative to each other. The design includes two subsystems named `ss_clk_domainA` and `ss_clk_domainB`, which include logic associated with clock domains, A and B, respectively. The blocks inside the `ss_clk_domainA` subsystem operate in clock domain A while all blocks inside the `ss_clk_domainB` subsystem operate in a second clock domain, B.

The asynchronous islands in the example communicate with one another via a shared memory interface implemented using a pair of Xilinx Shared Memory blocks. The two Shared Memory blocks are distributed so that one block resides in domain `ss_clk_domainA` and the other resides in domain `ss_clk_domainB`. Both blocks specify the same shared memory object name, `bram_iface`. This allows the Shared Memory blocks to access a common address space during simulation. Note that in the diagram there is no physical connection shown between the two shared memory halves. This is because the connection is implicitly defined by the fact that the two Shared Memory blocks specify the same shared memory object name and therefore, share an address space. When the two subsystems are wired together and translated into hardware, the shared memory blocks are moved from their respective subsystems and merged into a block RAM core. For more information on how this works, refer to the Multiple Subsystem Generator section in the User Guide.

The synchronous islands sample different input sources. Island `ss_clk_domainA` samples a sinusoid input, while `ss_clk_domainB` samples a saw-tooth wave input. Each subsystem writes its samples into opposite halves of the shared memory. Once an island has filled its half of memory, it reads samples from the other island's half. You can simulate the design to visualize of the model's behavior.

3. Press the **Simulink Play** button to simulate the design.
4. Open the scope to visualize the output signals.

Also shown in the output scope are the two clocks, `clk_A` and `clk_B`. At the default time scale, it is difficult to distinguish the two. Zoom in to get a more detailed view.

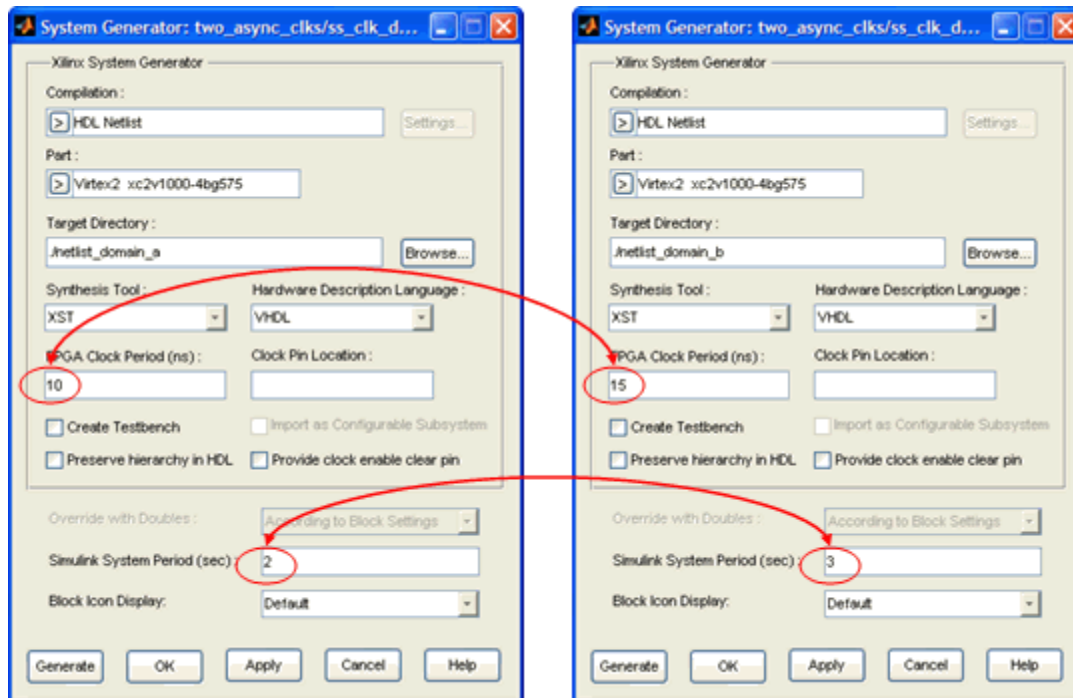


Notice that `clk_A` and `clk_B` have different periods and are out of phase with one another. Earlier, we claimed that System Generator uses a single clock source per design. In the scope, we clearly see two different clocks. How is this possible?

The answer is in the hierarchical construction of the design. All blocks are buried in at least one level of hierarchy using subsystems. Because there is no System Generator block at the top level, we can consider each subsystem as a completely separate System Generator design (at least for the time being). In this model, we have effectively defined two clock domains by giving the `ss_clk_domainA` and `ss_clk_domainB` subsystems different Simulink system periods. This is allowed since we are treating these subsystems as separate System Generator designs. The clock probes in the `ss_clk_domainA` and `ss_clk_domainB` subsystems use the Simulink system periods in their respective subsystems to determine their output, hence different system periods yield different system clocks.

Let's now consider clocks defined by the System Generator block in the `ss_clk_domainA` and `ss_clk_domainB` subsystems.

5. Open the System Generator block parameter dialog boxes inside the `ss_clk_domainA` and `ss_clk_domainB` subsystems.

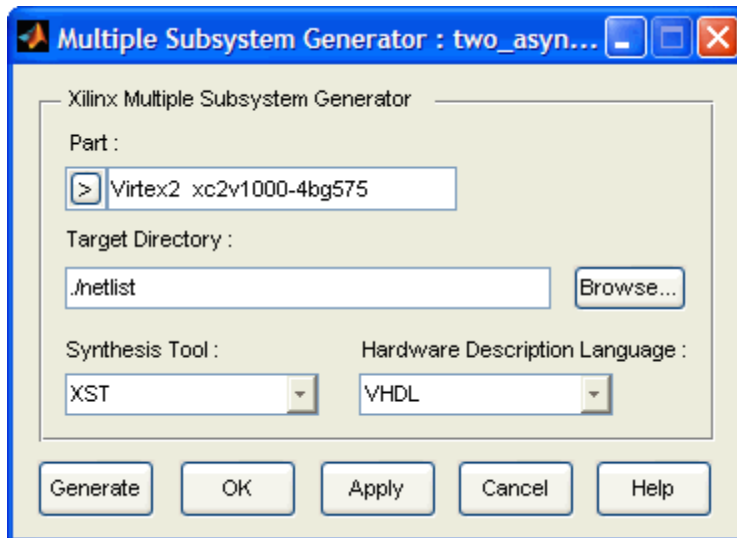


The System Generator block dialog box in the `ss_clk_domainA` subsystem defines an FPGA clock period of 10ns (i.e., a frequency of 100MHz). To simplify the sample period values in the model, the 10 ns clock is normalized to a Simulink system period value of 2 sec. In the `ss_clk_domainB` subsystem, an FPGA clock period of 15ns (i.e., a frequency 66.7 MHz) is defined. Normalizing this clock period gives us a Simulink system period value of 3 sec.

Because the two subsystems in this example implement multiple, synchronous, System Generator domains, we will use the Multiple Subsystem Generator block to wire the subsystems together into a single HDL top-level component that exposes two clock ports. When the Multiple Subsystem Generator translates a design into hardware, it generates each subsystem individually as an NGC netlist file. It also creates a top-level VHDL component or Verilog module that instantiates the subsystem netlist files as black boxes, and wires them together using shared memory cores as clock domain bridges.

We'll begin by using the Multiple Subsystem Generator block to netlist subsystems `ss_clk_domainA` and `ss_clk_domainB`.

6. Open the Multiple Subsystem Generator dialog box by double clicking on the Multiple Subsystem Generator block included in the top-level of the `two_async_clks` model.
7. Pick a suitable target directory inside the Multiple Subsystem Generator dialog box. The default directory is `netlist`.
8. Press the **Generate** button. You may leave the **Part**, **Synthesis Tool**, and **Hardware Description Language** fields as they are.



Once the Multiple Subsystem Generator block is finished running, it will display a message box indicating that generation is complete. It is worthwhile to take a look at the generated results.

9. CD into the design's target directory, `netlist`.

There are two NGC files in this directory: `ss_clk_domaina_cw.ngc` and `ss_clk_domainb_cw.ngc`. These files store the netlist and constraints information corresponding to the subsystems `ss_clk_domaina` and `ss_clk_domainb`. Note that these NGC files include the clock wrapper layer logic associated with each subsystem. This is necessary to ensure that any clock enable logic required by a multi-rate design is included in netlist file. By using the clock wrapper layer of a design, the corresponding clock driver components are automatically included in the netlist.

Also in this directory is a dual port memory core netlist file named `dual_port_block_memory_virtex2_6_1_ef64ec122427b7be.edn`. This core provides the hardware implementation for the Shared Memory blocks used in the original design. The width and depth of the memory are based on values used in the Shared Memory block configurations.

We will now take a look at the top-level HDL component that the Multiple Subsystem Generator block produced for the design.

10. Open the `two_async_clks.vhd` file in a text editor.

This component defines the HDL top-level for the `two_async_clks` model.

```
entity two_async_clks is
  port (
    din_a: in std_logic_vector(7 downto 0);
    din_b: in std_logic_vector(7 downto 0);
    ss_clk_domaina_cw_ce: in std_logic := '1';
    ss_clk_domaina_cw_clk: in std_logic;
    ss_clk_domainb_cw_ce: in std_logic := '1';
    ss_clk_domainb_cw_clk: in std_logic;
    dout_a: out std_logic_vector(7 downto 0);
    dout_b: out std_logic_vector(7 downto 0)
  );
end two_async_clks;
```

There are several interesting things to notice about the port interface. First, the component exposes two clock ports (shown in yellow). The two clock ports are named after the subsystems from which they are derived (e.g., `ss_clk_domaina`), and are wired to their respective subsystem NGC netlist files. Also note that the top-level ports of each subsystem (e.g., `din_a` and `dout_a`) appear as top-level ports in the port interface.

The Multiple Subsystem Generator block does not generate circuitry (e.g., a DCM) to generate multiple clock sources. You may modify the top-level HDL component to include the circuitry, or instantiate the top-level HDL as a component in a separate wrapper that includes the clocking circuitry.

Creating a Top-Level Wrapper

If you decide to create a top-level HDL wrapper for your multi-clock System Generator design, it should perform the following tasks at a minimum:

- Instantiate the System Generator top-level component along with other wrapper logic (e.g., a DCM);
- Wire the System generator component to the other logic;
- Create a new top-level port map which supercedes that from the System Generator component.

The following is an example of making a top-level HDL component to instantiate clocking circuitry. In this example, we take the output created when the example from the previous section is generated using the Multiple Subsystem Generator block. The resulting System Generator design is called `two_async_clks` and the top-level HDL component is called `top_wrapper` (for the case of VHDL synthesis).

Because the clock lines and main clock enables are inferred, the names of the clocks and clock enables (with the `_ce` and `_clk` suffixes above) are generated automatically by putting suffixes on the subsystem names from which the clocks are inferred. The other port names, such as `dout_a`, are taken directly from the names given to the gateway blocks in the System Generator design.

An example VHDL top-level wrapper to instantiate the entity `two_async_clks`, with deletions made for clarity, is provided below. Note that the wrapper uses a DCM component to generate the two clocks required by the System Generator design.

```
-----
--
-- top_wrapper.vhd
-- Example Top Level Wrapper
--
-- This is an example top-level wrapper for instantiating a System
Generator
-- design along with a DCM. In this example, the DCM connects the two
clock
-- inputs of the System Generator block ('two_async_clks') to two
buffered
-- outputs of the DCM, namely, CLK0 and CLKFX. CLK0 is the same
frequency
-- and phase as the input clock, and CLKFX is configured to be twice the
-- frequency of the input clock.
-----
library IEEE;
library unisim;
use IEEE.std_logic_1164.all;
use unisim.vcomponents.all;

entity top_wrapper is
  port (
```

```

    clk          : in  std_logic;
    din_a        : in  std_logic_vector(7 downto 0);
    din_b        : in  std_logic_vector(7 downto 0);
    dout_a       : out std_logic_vector(7 downto 0);
    dout_b       : out std_logic_vector(7 downto 0)
  );
end top_wrapper;

architecture structural of top_wrapper is
  -----
  -- SysGen Model Component Declaration
  -----
  component two_async_clks
    port (
      din_a: in std_logic_vector(7 downto 0);
      din_b: in std_logic_vector(7 downto 0);
      ss_clk_domaina_cw_ce: in std_logic := '1';
      ss_clk_domaina_cw_clk: in std_logic;
      ss_clk_domainb_cw_ce: in std_logic := '1';
      ss_clk_domainb_cw_clk: in std_logic;
      dout_a: out std_logic_vector(7 downto 0);
      dout_b: out std_logic_vector(7 downto 0)
    );
  end component;
  component bufg
    port(i: in std_logic;
         o: out std_logic);
  end component;

  -----
  -- DCM Component Declaration
  -----
  component dcm
    -- synopsys translate_off
    generic (clkout_phase_shift : string := "fixed";
             dll_frequency_mode : string := "low";
             duty_cycle_correction : boolean := true;
             clkdv_divide : real := 3;
             clkfx_multiply : integer := 2;
             clkfx_divide : integer := 1);
    -- synopsys translate_on
    port (clkkin : in std_logic;
          clkfb : in std_logic;
          dssn : in std_logic;
          psincdec : in std_logic;
          psen : in std_logic;
          psclock : in std_logic;
          rst : in std_logic;
          clk0 : out std_logic;
          clk90 : out std_logic;
          clk180 : out std_logic;
          clk270 : out std_logic;
          clk2x : out std_logic;
          clk2x180 : out std_logic;
          clkdv : out std_logic;
          clkfx : out std_logic;
          clkfx180 : out std_logic;
          locked : out std_logic;
          psdone : out std_logic;
          status : out std_ulogic_vector(7 downto 0));
  end component;

  -----
  -- DCM Attributes
  -----
  attribute dll_frequency_mode : string;
  attribute duty_cycle_correction : string;
  attribute startup_wait : string;

```



```

attribute clkdv_divide          : string;
attribute clkfx_multiply       : string;
attribute clkfx_divide         : string;
attribute clkln_period         : string;
attribute duty_cycle_correction of dcm0 : label is "true";
attribute startup_wait         of dcm0  : label is "false";
attribute dll_frequency_mode   of dcm0  : label is "low";
attribute clkdv_divide         of dcm0  : label is "3";
attribute clkfx_multiply       of dcm0  : label is "2";
attribute clkfx_divide         of dcm0  : label is "1";
attribute clkln_period         of dcm0  : label is "10";

signal clk0unbuf                : std_logic;
signal clk0buf                  : std_logic;
signal clkfxbuf                 : std_logic;
signal clk2xunbuf               : std_logic;
signal clkfxunbuf               : std_logic;
signal clkdvunbuf               : std_logic;
signal clkdvbuf                 : std_logic;
signal ff1,ff2,ff3,ff4         : std_logic;
signal dcm_rst                  : std_logic;
signal intlock                  : std_logic;

```

```

-----
-- The top level instantiates the SysGen design, a DCM, and two BUFs.
-- The DCM generates two clocks of different frequencies.
-- These two clocks are used to drive the two different clock domains
-- in the SysGen block.
-----

```

```

begin
  dcm0: dcm
    -- synopsys translate_off
    generic map (dll_frequency_mode => frequency_mode,
                 clkdv_divide       => clkdv_divide_generic,
                 clkfx_multiply     => clkfx_multiply_generic,
                 clkfx_divide       => clkfx_divide_generic)
    -- synopsys translate_on
    port map (clkln    => clk,
              clkfb    => clk0buf,
              dssen    => '0',
              psincdec => '0',
              psen     => '0',
              psclk    => '0',
              rst      => dcm_rst,
              clk0     => clk0unbuf,
              clk2x    => clk2xunbuf,
              clkfx    => clkfxunbuf,
              clkdv    => clkdvunbuf,
              locked   => intlock);
  bufg_clk0: bufg
    port map (i => clk0unbuf,
              o => clk0buf);
  bufg_clkfx: bufg
    port map (i => clkfxunbuf,
              o => clkfxbuf);

```

```

-----
-- This is the DCM reset. It is a four-cycle shift register used to
-- hold the DCM in reset for a few cycles after programming.
-----

```

```

flop1: FDS port map (D => '0', C => clk, Q => ff1, S => '0');
flop2: FD port map  (D => ff1, C => clk, Q => ff2);
flop3: FD port map  (D => ff2, C => clk, Q => ff3);
flop4: FD port map  (D => ff3, C => clk, Q => ff4);
dcm_rst <= ff2 or ff3 or ff4;

```

```
-----  
-- SysGen Component Port Mapping  
-- One clock input is being connected to clk0 of the DCM,  
-- and the other clock is being connected to clkfx.  
-----  
two_async_clks: two_async_clks  
  port map (  
    din_a                                => din_a,  
    din_b                                => din_b,  
    ss_clk_domaina_cw_ce                 => '1',  
    ss_clk_domaina_cw_clk                => clk0buf,  
    ss_clk_domainb_cw_ce                 => '1',  
    ss_clk_domainb_cw_clk                => clkfxbuf,  
    dout_a                               => dout_a,  
    dout_b                               => dout_b);  
  
end structural;
```

Frame-based Acceleration using Hardware Co-simulation

With the tremendous growth in programmable device size and computational power, lengthy simulation times have become an expected, yet undesirable part of life for most engineers. Depending on the design size and complexity, the required simulation time can be quite large, sometimes on the order of days to run to completion. This problem is exacerbated by the fact that most systems must be simulated many times before the design is considered functional and ready for deployment. Fortunately, System Generator for DSP provides hardware co-simulation interfaces that allow you to dramatically accelerate simulation speeds of your FPGA designs.

There are several factors that influence exactly how much acceleration can be gained by using hardware co-simulation. These considerations include the size of the design, the number of ports on the model, and the hardware over-sampling rate. Under normal operation, the PC communicates with the FPGA during each Simulink simulation cycle. These software / hardware transactions often involve significant overhead and can end up being the limiting factor in simulation performance. Also of importance is the co-simulation interface being used. Some interfaces (e.g., PCI) are faster than others (e.g., JTAG). For a reasonably large design, the typical simulation is accelerated by an order of magnitude when co-simulated in hardware.

Keeping the above points in mind, there are ways to further bolster simulation performance. Remember that every time the PC interacts with hardware, there is an overhead cost that impacts simulation performance. One of the ways the number of FPGA transactions can be mitigated is by utilizing Simulink vector and frame signal types. Here and throughout the rest of this tutorial, we refer to FPGA transactions involving Simulink vector and frame signals as simply **vector transfers**. This idea is straightforward – bundle as many input data samples together as possible and have the FPGA process the data in a single transaction. Fewer transactions with the FPGA results in better simulation performance.

In this tutorial, we describe how Simulink vector and frame signals may be used to increase simulation performance beyond what is traditionally possible with hardware co-simulation. A step-by-step example filter design is presented to help illustrate these concepts. By using the techniques described in this tutorial, it is possible to further increase simulation performance, sometimes by as much as two orders of magnitude over software-based simulations.

Before diving into the details, it is worth exploring exactly what we are trying to accomplish from a high-level perspective. In summary, we will do the following during a Simulink simulation cycle:

- Buffer a series of scalar input data values into a Simulink vector;
- Transfer the vector data to a buffer residing on the FPGA using a burst transfer;
- Use the FPGA, in free-running clock mode, to sequentially process the entire input buffer;
- Use the FPGA to write the data into an output buffer;
- Transfer the contents of the output buffer back into Simulink and reconstruct the data as a Simulink vector;
- Unbuffer the vector into a series of output scalar values.

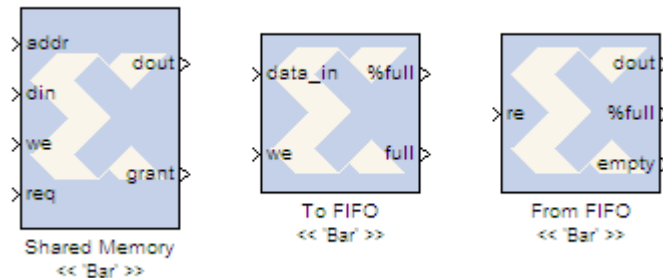
Shared Memories

Before a System Generator design can support vector transfers, it must be augmented with appropriate input and output buffers. In hardware, these buffers are implemented using internal memory (e.g.,

BRAMs) and are used to store vectors of simulation data that are written to and read from the FPGA by the PC. This means that the maximum size of the buffers is limited by the amount of internal memory available on the target device. In System Generator, shared memory blocks provide interfaces that implement such buffers.

A question that quickly comes to mind is why not use standard FIFO or memory blocks? The buffers required for hardware co-simulation differ from traditional FIFOs and memories in that they must be controllable by both the PC and FPGA user design logic. The standard FIFO and memory blocks provided by System Generator can only interface with user design logic.

There are two types of shared memories that provide this control: lockable shared memories and shared FIFOs. These blocks provide different buffering styles; each with their own handshaking protocols that determine when and how burst transactions with the FPGA occur. In this tutorial, primary attention is focused on shared FIFO buffers. For an example on how to use lockable shared memories, please refer to the System Generator User Guide tutorial entitled *Real-time Signal Processing using Hardware Co-simulation*. You may find the lockable shared memory and FIFO blocks in the Shared Memory library of the Xilinx Blockset.

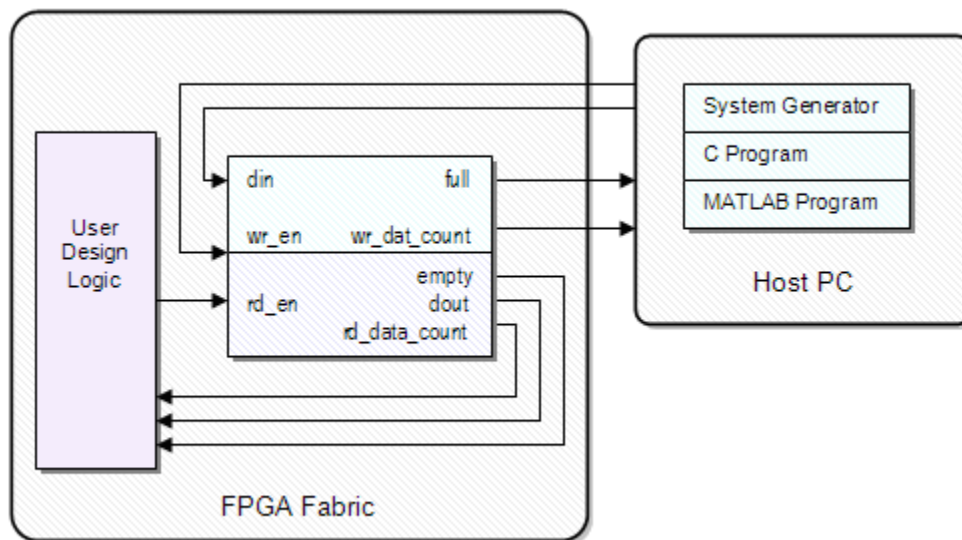


Because shared FIFOs play a central role in enabling vector transfers, it is worth a brief aside to discuss their behavior. A shared FIFO pair is comprised of a To FIFO block and a From FIFO block that specify the same name (e.g., *Bar* in the figure above). The To FIFO block provides the “write side” control signals, while the From FIFO block provides the “read side” control signals. When used together, a shared FIFO pair is conceptually the same thing as a single FIFO – only the control signals for the two sides are graphically disjoint. This means that a shared FIFO pair shares the same FIFO memory space. For example, if you write data into a To FIFO block, you may retrieve the same data by reading from the From FIFO block. The connection between these two blocks is implicit; shared FIFOs are associated with one another by name and not by explicit Simulink wires.

Shared FIFOs and shared memories in general may be compiled for hardware co-simulation. Note that although this tutorial touches briefly on how shared FIFOs are co-simulated, it is useful to refer to the User Guide section entitled *Co-simulating Shared FIFOs* for more in-depth information. When one-half of a shared FIFO block is compiled for hardware co-simulation, a full FIFO block is embedded in the FPGA using the FIFO Generator core. One side of the FIFO connects to user design logic (i.e., the System Generator logic that connected to the shared FIFO block). The other half connects to interface logic that allows it to be controlled by the PC. This side of the FIFO may be controlled by other System Generator software model logic (e.g., the half of the shared FIFO), by a C program or software executable, or by a MATLAB program. By compiling shared FIFOs for hardware co-simulation, we create embedded FIFO-style buffers in the FPGA that can be controlled directly by a PC.

For example, the figure below shows how a From FIFO block is implemented for hardware co-simulation. The “To” side of the FIFO operates under PC control. The “From” side connects to user design logic. The opposite wiring approach is used when a To FIFO is compiled for hardware co-

simulation (i.e., the “To” side connects to user design logic while the “From” side is controlled by the PC).

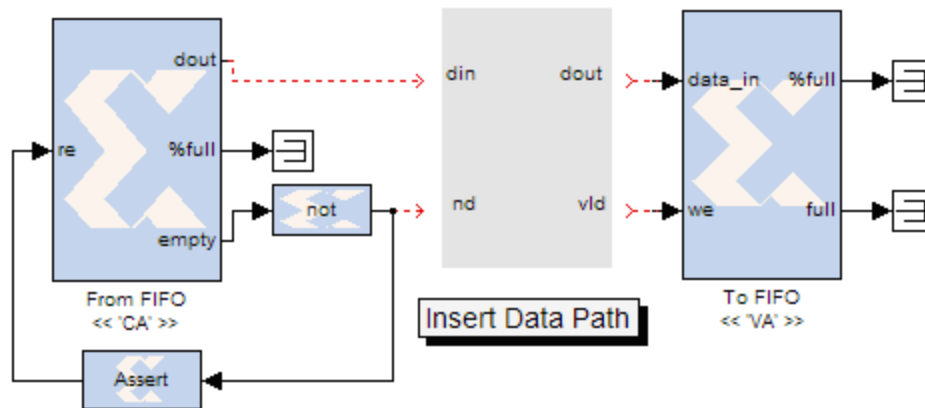


There are several ways to communicate with a shared FIFO that is embedded inside the FPGA. The most common approach is to include the other half of the shared FIFO in the System Generator design. It is also possible to communicate with the shared FIFO using a C program or MATLAB program. System Generator provides additional blocks that support vector transfers to and from the FIFO. We will touch on these blocks later in the tutorial as they play a key role in supporting burst transfers to and from the FPGA.

Adding Buffers to a Design

Having gained an understanding of how shared FIFOs work in hardware, we turn our attention towards building designs that can utilize these buffers for high-speed vector processing in the FPGA.

Consider the scenario in which we have an FPGA data path that we would like to accelerate using vector transfers. We need to include input buffer storage in the FPGA that can store data input samples that are written by the PC. An output buffer is also required so that the processed data values can be stored while the FPGA waits for the PC to retrieve them. With these requirements in mind, a From FIFO block is used to implement the input data buffer and a To FIFO block is used to implement the output data buffer. In the model shown below, data is written into the data path as soon as it shows up in the input FIFO. Note that the data path block contains new data (**nd**) and data valid (**vld**) flow control ports. These ports support a simple flow control scheme that determines when new data enters and valid data leaves the data path. The **nd** signal is asserted whenever there is data available in the input FIFO. Conversely, data is written into the output FIFO whenever valid data is present on the data path.

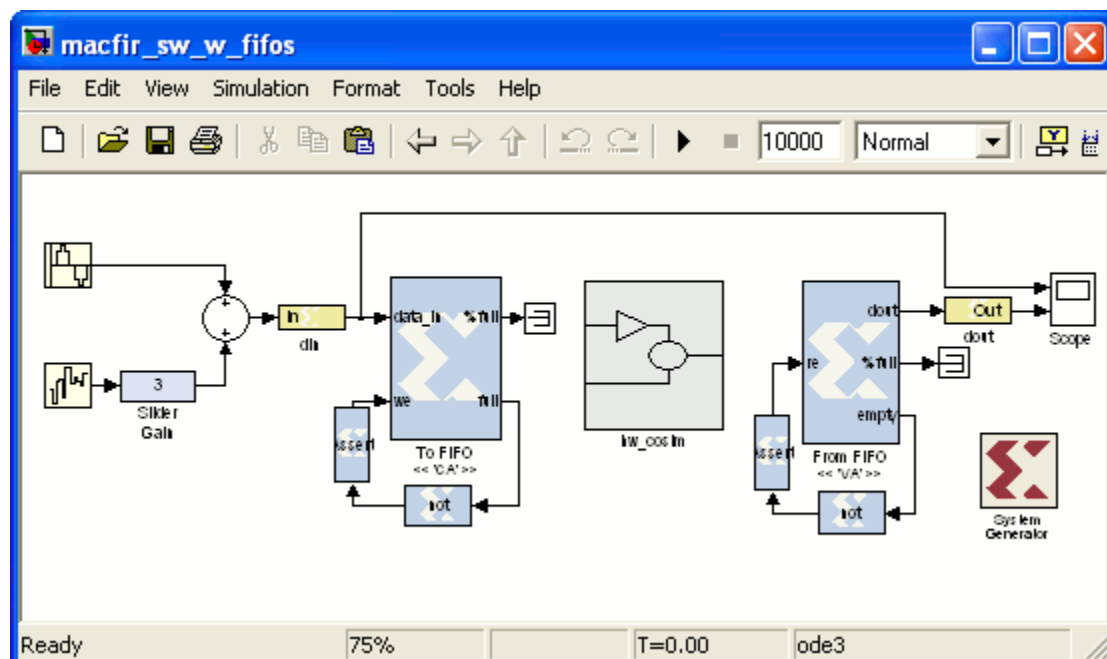


To gain a better understanding of how the Shared FIFOs are used, we will now take a look at an example design that uses vector transfers to accelerate a MAC filter design.

1. From the MATLAB console, change directory to
`<path_to_sysgen>/examples/shared_memory/hardware_cosim/frame_acc.`

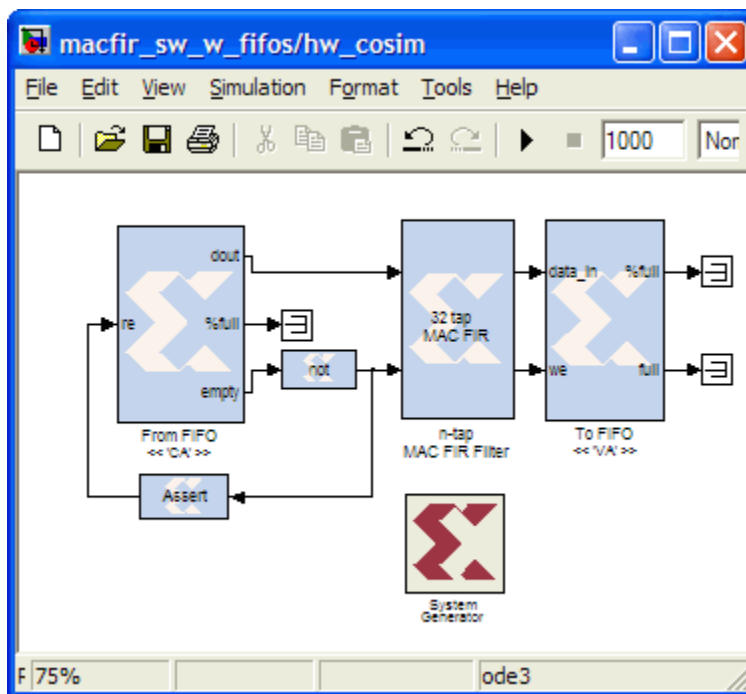
The default path to System Generator is \$MATLAB/toolbox/Xilinx/sysgen.

2. Open `macfir_sw_w_fifos.mdl` from the MATLAB console.



The example design implements a 32-tap MAC FIR filter that removes additive white noise from a sinusoid input source. The amount of white noise can be adjusted interactively by moving the **Slider Gain** control bar before or during simulation. An output scope compares the filtered output data against the unfiltered input data. The MAC filter itself is contained inside a subsystem named

`hw_cosim`. This subsystem contains all of the logic that will be compiled into the FPGA for hardware co-simulation. We consider everything else in the design (i.e., all blocks in the top-level) as our design testbench.

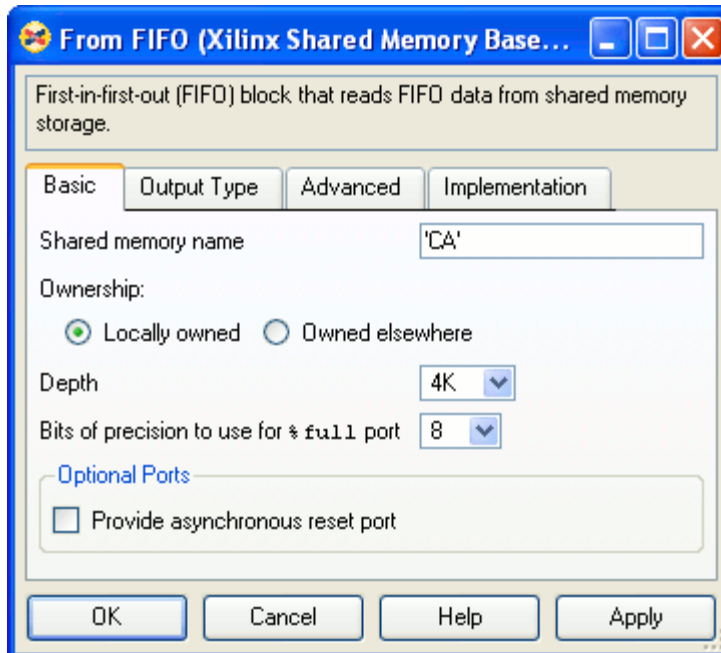


Pushing into the `hw_cosim` subsystem, we have an `n-tap MAC FIR Filter` block that implements the design data path. Wrapping the filter are `From FIFO` and `To FIFO` blocks that provide the input and output buffers, respectively. The MAC filter in the example design is a modified version of the `n-tap MAC filter` available in the System Generator DSP Reference Blockset library. In the example, the filter is modified to include valid in and valid out ports in order to support the FIFO flow control scheme.

In total, there are four shared memory blocks in the design that define the CA and VA shared FIFO pairs. In truth, we only need the shared FIFO blocks contained inside the `hw_cosim` subsystem to successfully compile the design for hardware co-simulation. Because we would like to simulate the complete design, including FPGA hardware, we include a CA To FIFO block and VA From FIFO block in the testbench logic. These shared FIFO blocks are responsible for writing and reading test data from the shared FIFOs in the `hw_cosim` subsystem.

Unfiltered data from the `din` Gateway In block is written into the CA To FIFO block. At this point, the CA From FIFO block in the `hw_cosim` subsystem reads data from the FIFO and writes it into the MAC filter. The MAC filter in turn processes the data and writes it into the output buffer, represented by the VA To FIFO block. Lastly, the VA From FIFO block in the top-level reads the data and sends it to the Scope block for visualization.

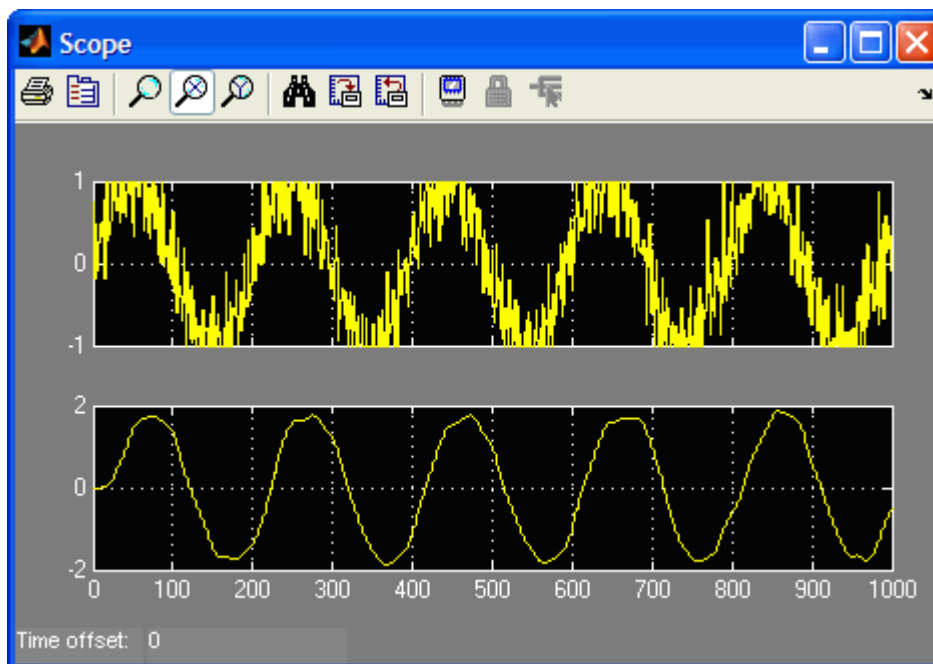
For this example, we have chosen a maximum buffer size of 4K. This parameter is set by specifying 4K for the `Depth` parameter on the CA From FIFO and VA To FIFO block dialog boxes. Note that because shared FIFOs are implemented using asynchronous FIFO Generator cores, the actual depth of the hardware FIFO is $n-1$ words, where n is the depth specified on the dialog box.



You will now have a chance to simulate the design to see how fast it runs in software.

3. Press the Simulink Play button to simulate the design in software.
4. Record the time required to simulate the design for 10000 cycles. To get an accurate measurement, it is preferable to leave the scope block closed since the graphic updates may affect simulation performance.

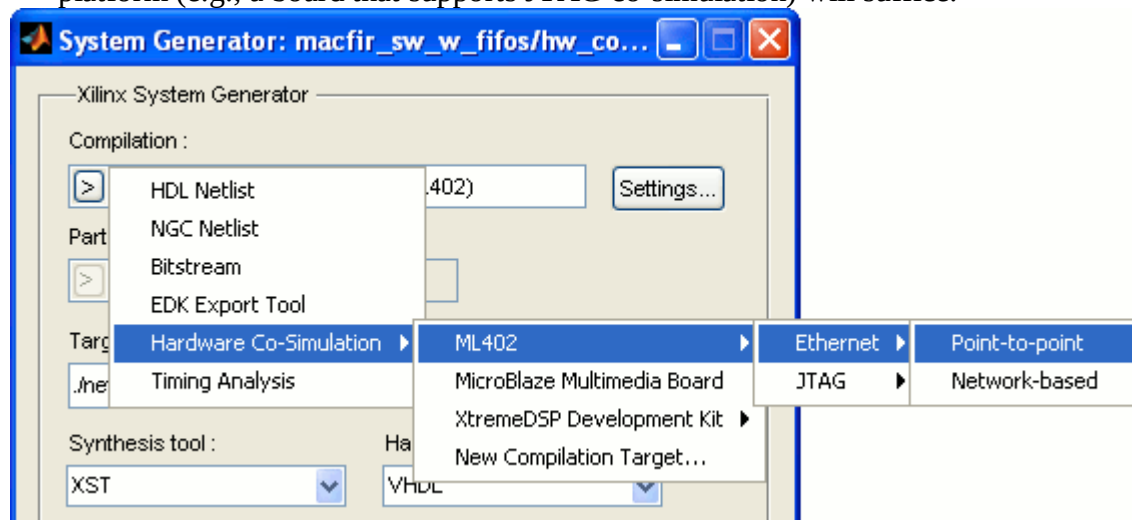
You may adjust the Slider Gain bar during simulation to see how the presence of additional noise affects the filter performance. You may view the filtered and unfiltered data in the output scope block. The top axis shows the unfiltered input data. The bottom axis shows the filtered data results.



Compiling for Hardware Co-simulation

We will now compile the design for hardware co-simulation. Before performing the following steps, ensure that you have an appropriate hardware co-simulation platform installed in System Generator and attached to your PC. In this example, we only want to compile the portion of the design that resides inside the `hw_cosim` subsystem. This is because we want the CA To FIFO and VA From FIFO blocks to remain in software as part of the design testbench (while their partner shared FIFOs are compiled into FPGA logic).

5. Double-click on the System Generator block in the `hw_cosim` subsystem to open the System Generator dialog box.
6. From the **Compilation** submenu, choose an appropriate hardware co-simulation target. Note that although we use the Point-to-point Ethernet hardware co-simulation interface in this example, any installed hardware co-simulation platform (e.g., a board that supports JTAG co-simulation) will suffice.



7. Press the **Generate** button on the System Generator dialog box to generate the design.

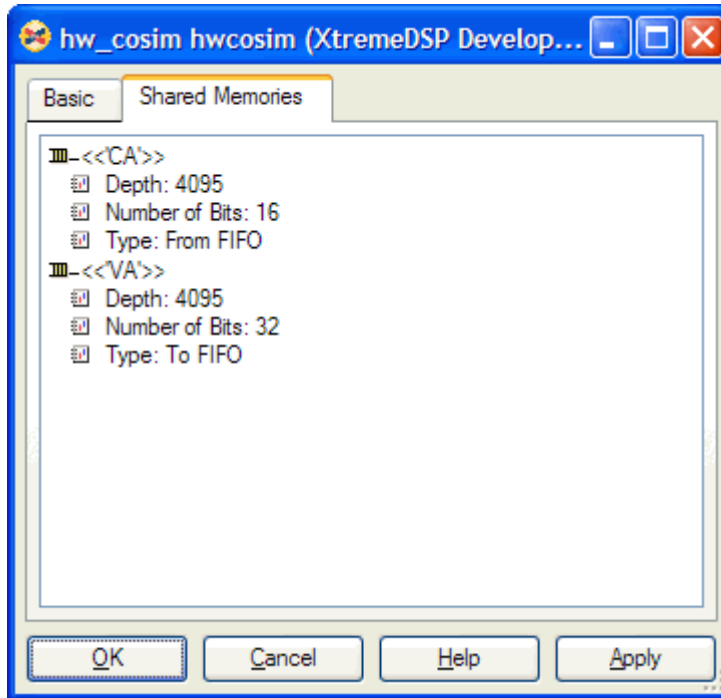
A new hardware co-simulation library and block are created once System Generator finishes compiling the design. Note that the new hardware co-simulation block does not have any input or output ports. This is because the subsystem that was compiled did not contain gateway blocks or Simulink ports. Instead, all connections to other Simulink blocks are handled implicitly through shared memories that were compiled into the FPGA. Because we left the To FIFO and From FIFO blocks as part of the software testbench, the software FIFOs will automatically attach to the FIFOs in hardware at the beginning of simulation.

It is often necessary to examine the type and configuration of a shared memory that was compiled for hardware co-simulation. The information about each shared memory is available in a **Shared Memories** tab on the hardware co-simulation block dialog box. This tab contains a tree view of information about each shared memory embedded in the design.

8. Double-click on the hardware co-simulation block to open the parameters dialog box.

9. Select the **Shared Memories** tab in the hardware co-simulation block dialog box.

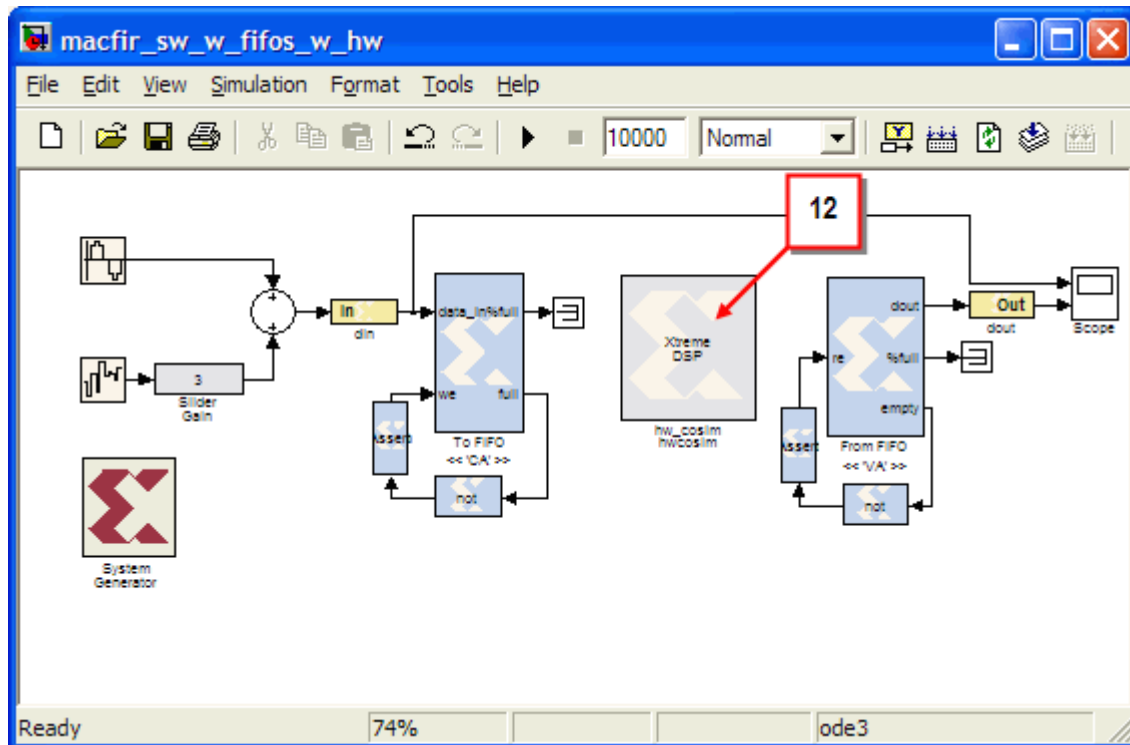
The tree-view contains information about the CA and VA shared FIFO blocks that were compiled. If your co-simulation design contains other shared memory blocks, information about these blocks will also be displayed here. You may expand or collapse shared memory information by clicking on the (+) or (-) icons located adjacent to the shared memory icons.



10. Close the parameters dialog box.

We are now ready to insert the hardware co-simulation block in the original design. Before continuing on with the next steps, it is worthwhile to either rename the design or create a backup of the original since you will be making modifications.

11. Remove the `hw_cosim` subsystem from the design.
12. Insert the hardware co-simulation block in place of the `hw_cosim` subsystem.

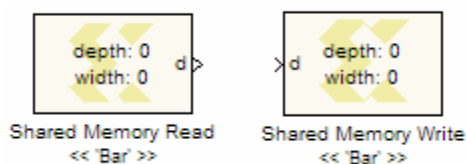


13. Configure the hardware co-simulation block with any settings necessary to co-simulate using single-step clock mode.
14. Press the Simulink Play button to start the design.
15. Record the amount of time required to simulate the design for 10000 cycles.
16. Close the design, but leave the hardware co-simulation library open since you will need it in the next section.

In the simulation above, hardware co-simulation uses single word transfers. That is, whenever there is a new simulation value to be read or written to the hardware co-simulation, the PC initiates a transaction with the FPGA. The next section describes how vector transfers may be used to further increase simulation speed by making more efficient use of the available hardware co-simulation bandwidth.

Using Vector Transfers

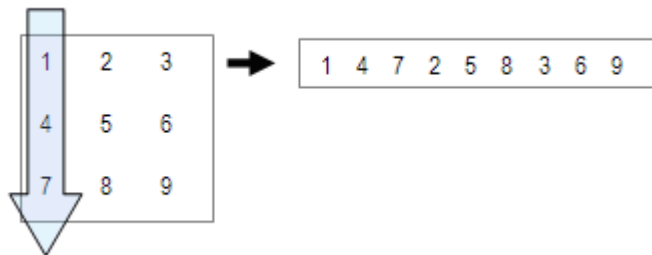
The System Generate Shared Memory Read and Write blocks allow you to use vector transfers with hardware co-simulation. These blocks may be found in the Shared Memory library in the Xilinx Blockset.



The Shared Memory Write block accepts a Simulink scalar, vector, matrix or frame data type and

writes the data sequentially into a shared memory. The complete contents of the Simulink signal are written into the shared memory in a single simulation cycle. As is the case with all shared memory blocks, an association is made between a Shared Memory Read or Write block and another shared memory by specifying the same shared memory name.

Matrix types are treated as having a **column-major** order. That is, when data is written sequentially into a shared memory, the elements in a column are written first before advancing to the next column. For example, assume we have the matrix of data shown below. During simulation, this matrix data is written into the FIFO (or shared memory) in the following order:



Using these blocks, it is possible to read or write full vector, frame, or matrix signals into shared memories, provided the following conditions are met:

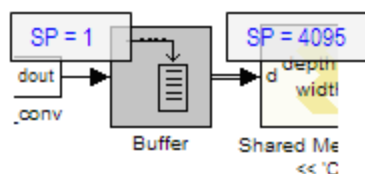
- The input signal driven to a shared memory write block is an 8-bit, 16-bit, or 32-bit signed or unsigned integer;
- The number of elements in the vector or matrix does not exceed the depth of the shared memory or FIFO.
- The data width of the Shared Memory Read or Write block (i.e., the bitwidth of the scalar, or vector or matrix element) equals the shared memory or FIFO data width.

We can use these blocks in the example design to read and write vectors of data samples to the MAC filter in a single software / hardware transaction.

17. Open `macfir_hw_w_frames_tb.mdl` from the MATLAB console.

This design is a very similar to the previous design, with a few modifications made to support the Shared Memory Read and Write blocks. Before simulating the design, we consider each of these modifications. Most importantly, Shared Memory Read and Write blocks have been substituted in place of the To and From FIFO testbench blocks in the previous design. By specifying CA and VA as the Write and Read shared memory names, respectively, an association is automatically made to the input and output FIFO buffers in the FPGA hardware during simulation.

A Simulink Buffer block builds a frame of scalar input samples by sequentially buffering the unfiltered input data. A simple analogy is that the Buffer block is performing a serial to parallel conversion. Recalling that we compiled the FIFO buffers with a depth of 4K, we choose a frame size of 4095.

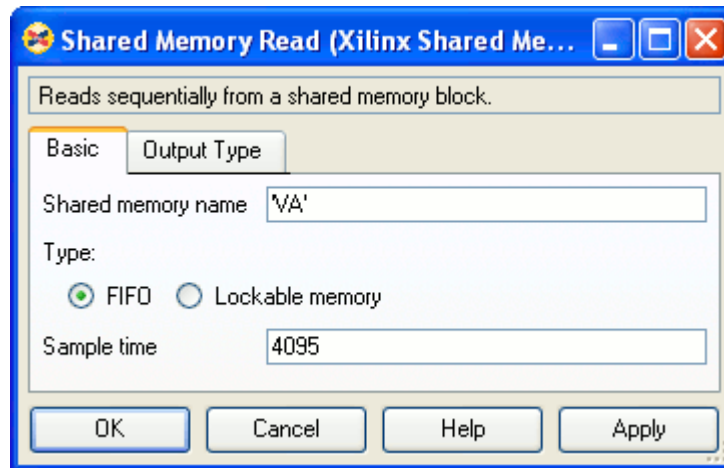


(SP: Sample Period)

Note that the buffer block introduces a sample rate change in the design. For every 4095 inputs, there is only one output. Thus if the data input sample period is 1, the buffer data output sample period is 4095. This means that the Shared Memory Write block need only send a new frame of data to the FPGA on every 4095th simulation cycle (which is considerably more efficient than initiating a hardware transaction during every simulation cycle).

Because the Buffer block introduces a rate change, we must adjust the downstream blocks to accommodate the slower sample period. We begin by telling the Shared Memory Read block to read a frame of data every 4095th simulation cycle.

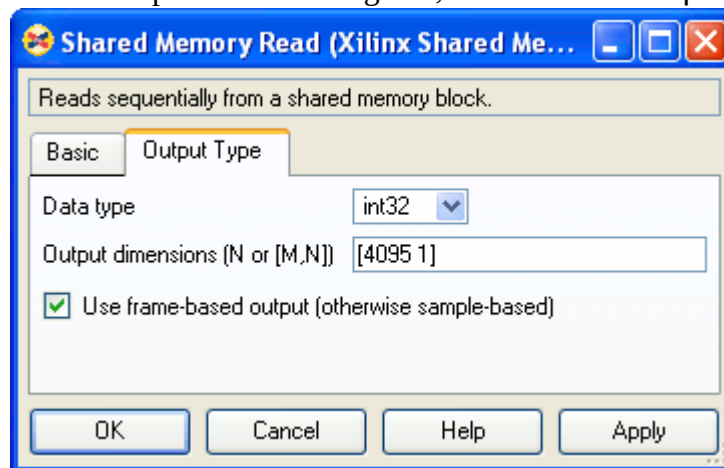
18. Double-click on the Shared Memory Read block to open its parameters dialog box.



On the Type field under the Basic tab, we have configured the block to use shared FIFOs. To ensure a new frame is read at the appropriate time, we configure the Shared Memory Read block with a Sample time value of 4095.

The Shared Memory Read block allows you to specify the output data type and dimensions.

19. On the parameters dialog box, switch to the Output Type tab.

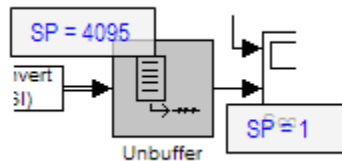


There are several things of interest on this tab. First, we set the output data type as an int32 to match the filter data path output width of 32-bits. Note the design will not simulate unless these widths

match. Secondly, we choose an output dimension that is 4095 words deep in the **Output dimensions** field. Finally, we tell the block to generate **frame-based output** since frame data types are required by the downstream Unbuffer block.

20. Close the parameters dialog box.

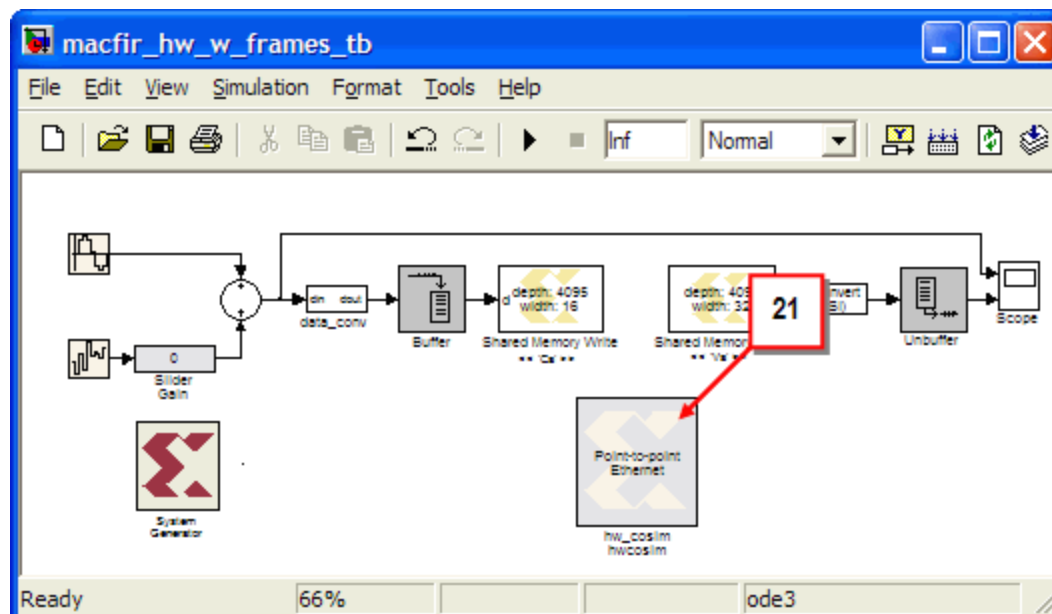
The Simulink Unbuffer block takes the frame data from the Shared Memory Read block and deserializes it into sequential scalar values. The Simulink Unbuffer block also introduces a sample rate change in the diagram. Because the input sample period to the block is 4095, and the frame size is 4095 words, the Unbuffer block output sample period is 1. This works out nicely since we have data moving through the overall system at an effective sample period of 1.



Because the Shared Memory Write and Read block operate on integer values, we must insert Simulink type conversion blocks into the diagram so that the data is interpreted correctly in various portions of the model. The `in_data_conv` subsystem converts the Simulink doubles into 16-bit integer values that can be interpreted appropriately by the FPGA hardware. On the output side, the `out_data_conv` subsystem converts the 32-bit integers into 32-bit Simulink fixed-precision values.

Before simulating the design, you must add the hardware co-simulation block you created from the previous design.

21. Add the hardware co-simulation block to the design as shown below.



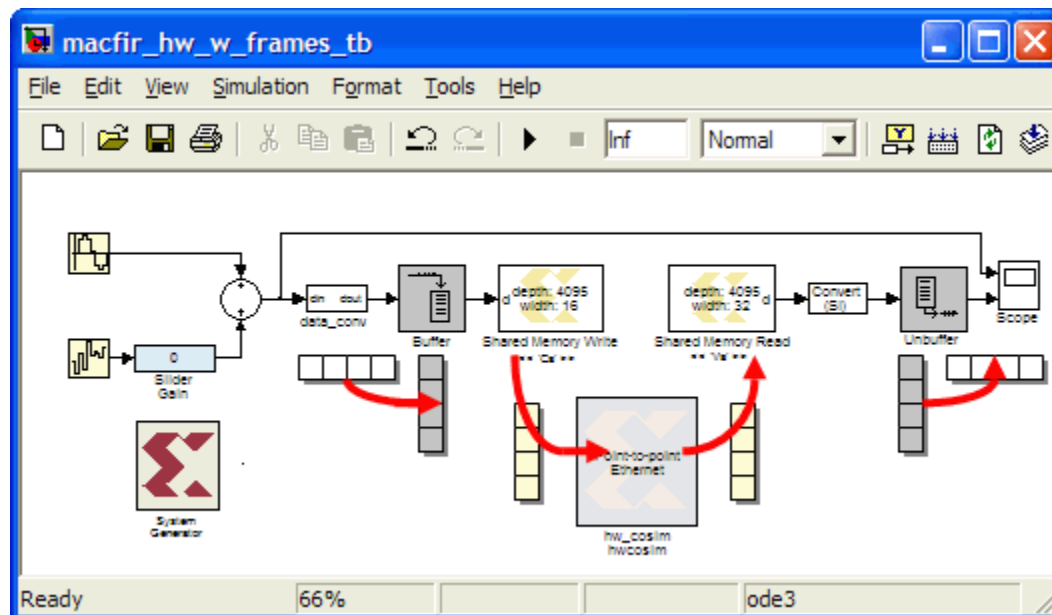
As mentioned before, the Shared Memory Write block writes a new input frame of 4095 words to the FPGA on every 4095th clock cycle. Likewise, the Shared Memory Read block reads an output frame of 4095 words from the FPGA on every 4095th clock cycle. This means that the FPGA must process

the entire frame in a single-cycle. How exactly is this accomplished?

The first step is to configure the FPGA in free-running clock mode. In doing so, we allow the FPGA to process data considerably faster than if it were otherwise kept in lockstep with the Simulink simulation. Whereas in single-step mode the FPGA can only process one data per Simulink cycle, the FPGA processing speed is limited only by the system clock frequency when operating in free-running clock mode. Even so, if the buffer is large enough, the FPGA may not have time to process the complete buffer before the next block in the design is woken up. We still need a way to stall the rest of the simulation while the FPGA processes the entire buffer.

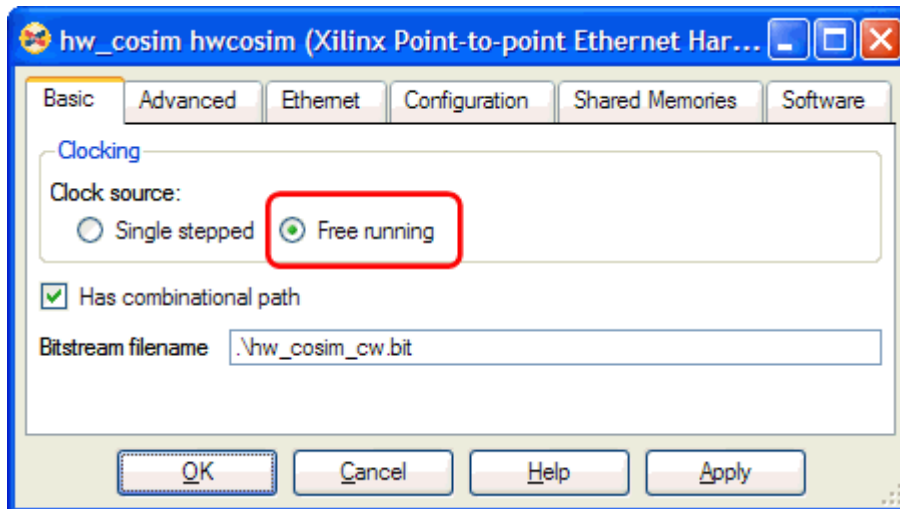
The Shared Memory Read block checks the number of FIFO words available in the output buffer before trying to read a frame. If the number of words in the buffer is insufficient, the Read block waits for a small amount of time, and then checks again to determine if the words have become available. It only reads the frame once all of the words are available in the output buffer, in this case 4095. In this manner, the Shared Memory Read block can stall the simulation until the complete frame has been processed by the FPGA.

The simulation flow of data through the diagram is shown below.



Two steps necessary to run the simulation using Simulink frames signals are provided below:

22. Double-click on the hardware co-simulation block to bring up the parameters dialog box.
23. Select **Free running clock mode** under the **Clocking** tab.



24. Configure the hardware co-simulation block with any additional settings necessary for simulation according to the requirements of your co-simulation platform.
25. Press the Simulink Play button to start the design.
26. Record the amount of time required to simulate the design for 10000 cycles.

Displaying and Changing Versions of System Generator

It is possible to have several versions of System Generator installed. The MATLAB command `xlVersion` displays which versions are installed, and makes it possible to switch from one to another. `xlVersion` is useful when updating a model to run in the latest version of System Generator.

Entering “`xlVersion`” in the MATLAB console displays the versions of System Generator that are installed, and entering “`xlVersion <version>`” switches to the specified version. For example, if the versions that are installed are 8.2 and 8.1, and the currently selected version is 8.2, then entering “`xlVersion`” displays

```
Available System Generator installations:
  Version 8.2 in C:/MATLAB71/toolbox/xilinx/sysgen
  Version 8.1 in C:/MATLAB71/toolbox/xilinx.8.1/sysgen
Current version of System Generator is 8.2.
```

Entering “`xlVersion 8.1`” switches the System Generator version to 8.1.

Occasionally, it is necessary to restart MATLAB to make it possible to switch. In this case, the response to entering “`xlVersion 8.1`” looks like the following:

```
Please restart MATLAB and run xlVersion 8.1 again to switch.
```

When the switch succeeds, `xlVersion` prints the following:

```
Your System Generator has been switched. Please restart MATLAB.
```

If you install System Generator 8.1 after you install 8.2, you need to install 8.2 again in order to make `xlVersion` work.

Once you switch System Generator version, you need to switch to the right version of ISE in order to make System Generator work correctly.

Real-time Signal Processing using Hardware Co-simulation

The shared memory interfaces available in System Generator allow signal processing designs with high bandwidth and memory requirements to be co-simulated using FPGA hardware. When used in conjunction with the Xilinx Shared Memory Read and Write blocks, it is possible for hardware co-simulation designs to process complete Simulink vector and matrix signals in a single simulation cycle. These large data transactions between Simulink and the FPGA are realized using burst transfers, and depending on the co-simulation interface, often provide sufficient throughput for real-time signal processing applications.

There are two types of System Generator interfaces that support burst transfers when compiled into FPGA hardware. These interfaces include lockable shared memories and shared FIFO blocks. Both blocks provide different handshaking protocols that determine how and when transactions between the FPGA and host PC occur. Before using these blocks, it is useful to understand how they work in relation to hardware co-simulation. For more information, please refer to the following sections in the User Guide:

Co-simulating Lockable Shared Memories
Co-simulating Shared FIFOs

In this document we present a high-speed co-simulation buffering interface implemented as a System Generator model. The example interface uses lockable shared memories to implement the required buffer storage. Note that it is relatively straightforward to modify the flow control logic so that shared FIFOs may be used in place of the shared memories.

The high-speed buffering interface is discussed first, followed by an example in which the interface is used to support real-time processing of a video stream using a 5x5 filter kernel. Lastly, we describe how an additional unprotected shared memory is applied to the system to support dynamic reloading of the image kernel during co-simulation.

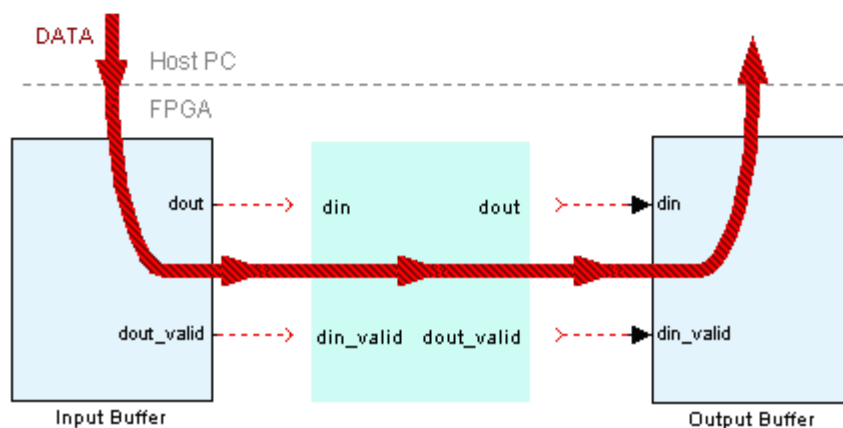
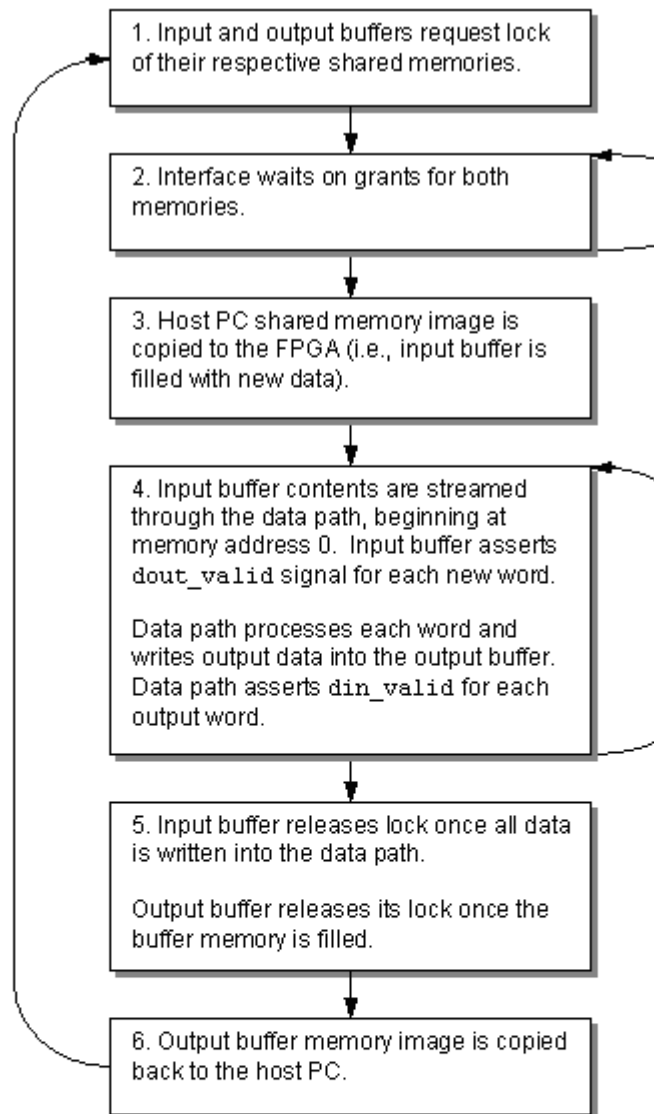
Shared Memory I/O Buffering Example

When a lockable shared memory is compiled for hardware co-simulation, additional circuitry is included in the FPGA to handle the mutual exclusion. Part of this circuitry includes logic to enable high-speed transfers of the memory image when the FPGA acquires or releases lock of the memory. We take advantage of the lockable shared memory mutual exclusion semantics to implement a high-speed I/O buffering interface for hardware co-simulation. This section describes this interface, which is included as an example model in your System Generator software installation.

1. From the MATLAB console, change directory to
`<path_to_sysgen>/examples/shared_memory/hardware_cosim/io_buffering`. The default path to System Generator is
`$MATLAB/toolbox/Xilinx/sysgen`.
2. Open `highspeed_iobuf_ex.mdl` from the MATLAB console.

The I/O buffering interface allows you to easily buffer and stream data through a System Generator signal processing data path during hardware co-simulation. The example design is comprised of two subsystems that implement input and output buffer storage, named **Input Buffer** and **Output Buffer**, respectively. The turquoise block in the center of the diagram is a placeholder for the signal processing data path which you will substitute into the design.

At the heart of each buffering subsystem is a lockable shared memory block that provides the buffer storage. Each shared memory is wrapped by logic that controls the flow of data from the host PC, through the interface, and back to the host PC. Operation of the I/O buffering interface is shown in the flow chart below:



Notice that the buffering interface design includes several *data valid* ports. These ports are used for

data flow control. A “true” output from the Input Buffer `dout_valid` port indicates new data is ready to be processed by the data path. Likewise, when the data path is finished processing the data, it should drive the Output Buffer subsystem’s `din_valid` port to “true” to indicate valid output data (the `din_valid` port is analogous to a write enable control signal).

The example includes a placeholder that should be replaced by a System Generator data path. You may insert any data path in the buffer interface provided that it works within the valid signal semantics described above.

Note: the output buffer shared memory does not release lock until the output buffer is full. To avoid deadlock, the number of valid assertions by the data path should equal the output memory buffer size for a given processing cycle.

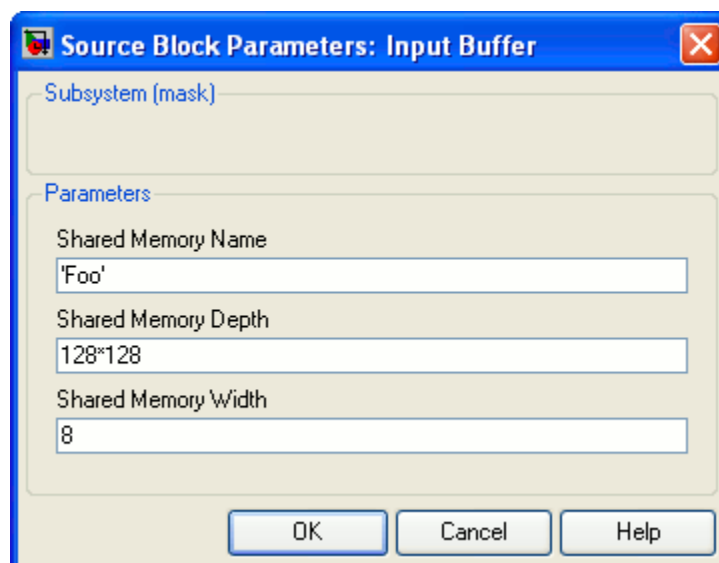
Applying a 5x5 Filter Kernel Data Path

We will now apply a data path to the I/O buffering interface to demonstrate a complete system capable of processing a 128x128 8-bit grayscale video stream in real-time. We have chosen to use a 5x5 image processing kernel to implement the data path portion of the high-speed buffering interface. For more information about the filter kernel, refer to the System Generator demo entitled `sysgenConv5x5`. We begin by considering various aspects of the design implementation.

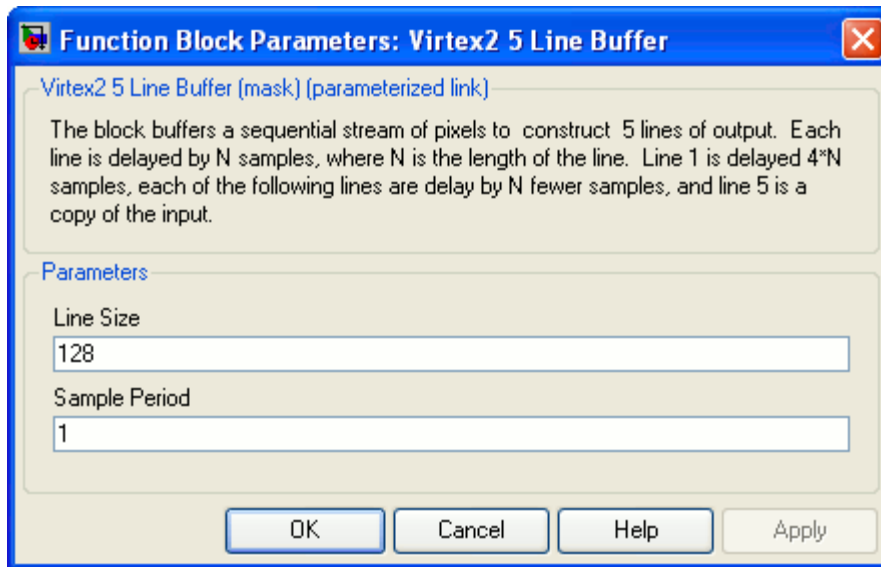
3. From the MATLAB console, change directory to `$SYGEN/examples/shared_memory/hardware_cosim/conv5x5_video`.
4. Open `conv5x5_video_ex.mdl` from the MATLAB console.

Buffer and Data Path Configuration

With the frame and pixel constraints in mind, the input and output buffer parameter dialog boxes are configured with a depth of 128x128 (16K) words and a word width of 8-bits. This depth allows the interface to process a complete frame in a single simulation cycle. Note that these configuration parameters are propagated automatically to the lockable shared memories that implement the buffer storage.

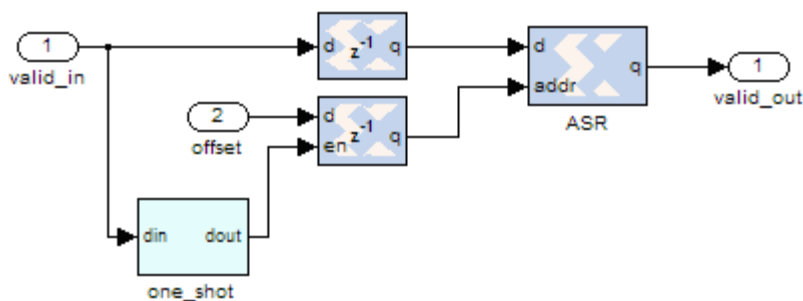


The data path uses line buffers to properly align data samples in the filter kernel. The size of these line buffers can be parameterized to accommodate different frame sizes. In this example, the line buffers are implemented in the Virtex2 5 Line Buffer block in the `conv5x5_video_ex/5x5_filter` subsystem, and are pre-configured with a line size of 128. If you decide to process a different size frame, the Line Size parameter should be updated accordingly.



Valid Bit Generation

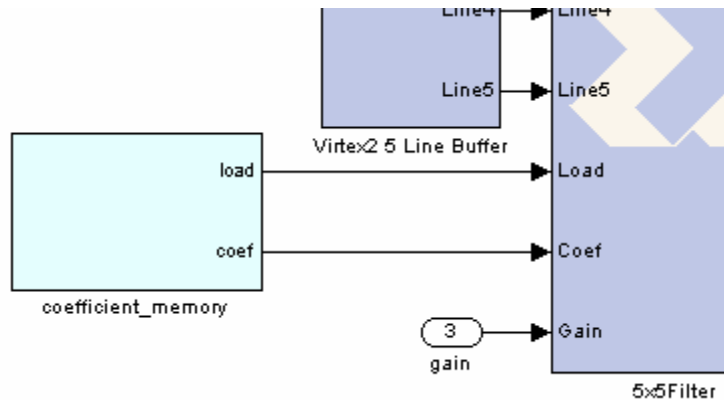
The data path includes a subsystem named `valid_generator` that is responsible for driving the `din_valid` port of the output buffer block. The subsystem has two inputs, `valid_in` and `offset`. The `valid_in` port is driven by the `dout_valid` signal from the input buffer block, which is delayed by a variable number of cycles before it is driven to the `valid_out` port. The logic associated with the `valid_generator` subsystem is shown below.



An addressable shift register block (ASR) is used to delay the valid bit. The `offset` port is used to control the address of the ASR block, which in turn controls the amount of latency the valid bit incurs. By simply delaying the valid bit generated by the input buffer block, we ensure the number of words written to the output buffer is always equal to the buffer size. Note that when the design is run in hardware, a change in the offset value will cause the vertical alignment of the filtered images to change.

Support for Coefficient Reloading

An interesting characteristic of the kernel data path is that its coefficients can be dynamically reloaded at run-time. The 5x5 filter block includes Load and Coef control ports, which are driven by the coefficient_memory subsystem.



The `coefficient_memory` contains a copy of the most recently loaded filter coefficients, which are stored in an *unprotected* shared memory named `coef_buffer`. During run-time, the subsystem monitors the shared memory contents, and initiates a reload sequence if detects a change. By co-simulating the unprotected shared memory, any process on the host PC may write new kernel coefficients simply by writing to a shared memory object named `coef_buffer`. This interface is convenient, as communication with the FPGA hardware is completely abstracted through the Shared Memory API.

Compiling for Hardware Co-simulation

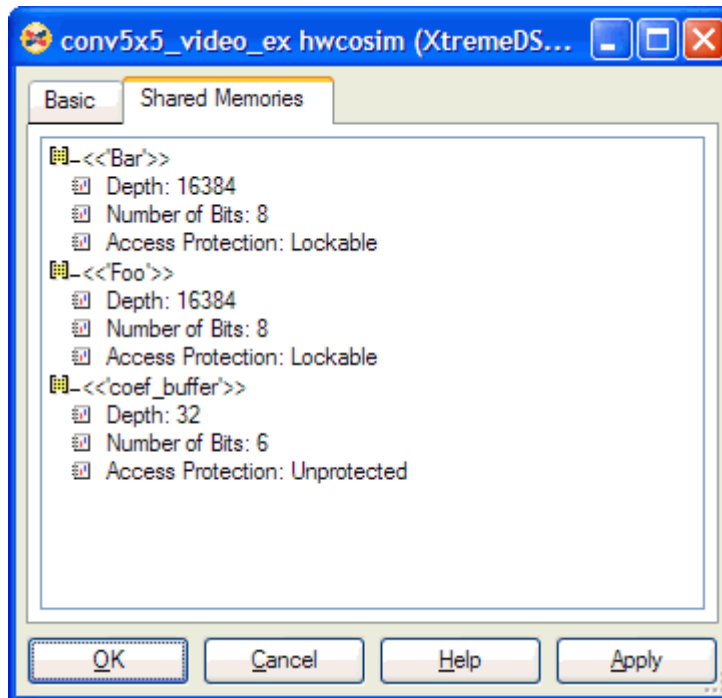
The full filter kernel design must be compiled for hardware co-simulation before it can be simulated.

5. Double click on the System Generator block located at the top of the `conv5x5_video_ex` model.
6. Select an appropriate hardware co-simulation target.
7. Press the **Generate** button to compile the design for hardware co-simulation.

A hardware co-simulation block is created once the design finishes compiling.



Hardware co-simulation blocks include information about any shared memories, registers, or FIFOs that were compiled as part of the design. You may view this information by double-clicking on the hardware co-simulation block to open the parameters dialog box. Once the dialog box is open, selecting the **Shared Memories** tab reveals information about each shared memory in the compiled design.



Go ahead and leave the hardware co-simulation library open. In the next section you will include the hardware co-simulation block in a video processing testbench design.

5x5 Filter Kernel Test Bench

Included with the example files is a Simulink test bench model that uses the hardware co-simulation block to filter a looped video sequence.

8. From the `$SYGEN/examples/shared_memory/hardware_cosim/conv5x5_video` directory, open `conv5x5_video_testbench.mdl`.

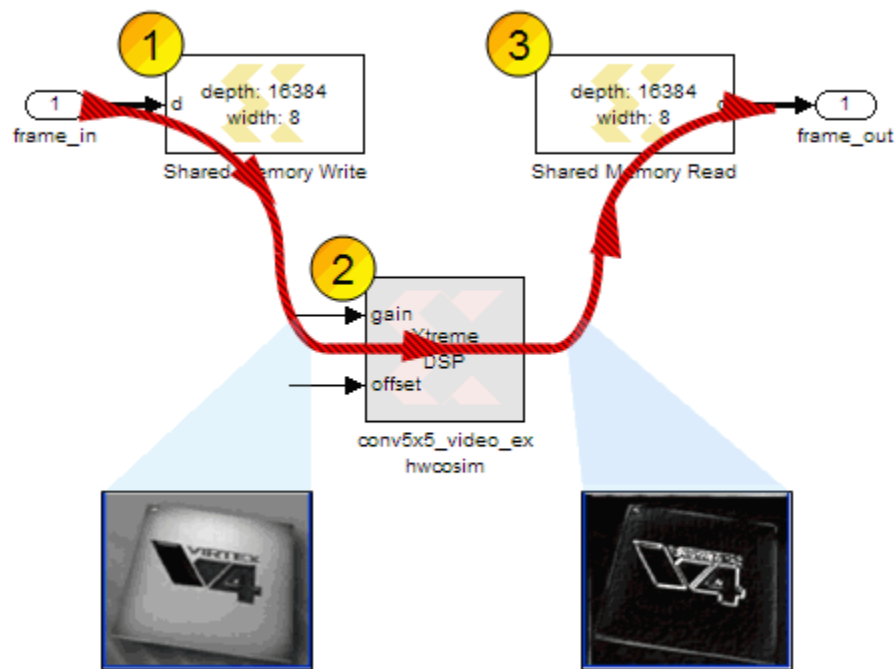
The testbench model uses a From Workspace block to produce the looped video sequence. Each frame of the video sequence is represented as a 128x128 `uint8` Simulink matrix (a pre-load function loads and initializes the video sequence automatically when the model is opened). Video frames are written into the **FPGA Processing** subsystem where they are filtered at the rate of one frame per simulation cycle. The filtered output is then written to a Matrix Viewer block for analysis.

The **FPGA Processing** subsystem contains a stub for the hardware co-simulation block, as well as Shared Memory Read and Write blocks. In this example, the Shared Memory Read and Write blocks are responsible for managing video frame I/O to and from the shared memories operating inside the FPGA. The operation of these blocks is described below:

1. The Shared Memory Write block wakes up and requests lock of the input buffer lockable shared memory `FOO`. Once lock is granted, the block writes the video frame data input into the lockable shared memory and releases lock.
2. The hardware co-simulation block wakes up and requests lock of the input and output buffer shared memories `FOO` and `Bar`. The host PC shared memory

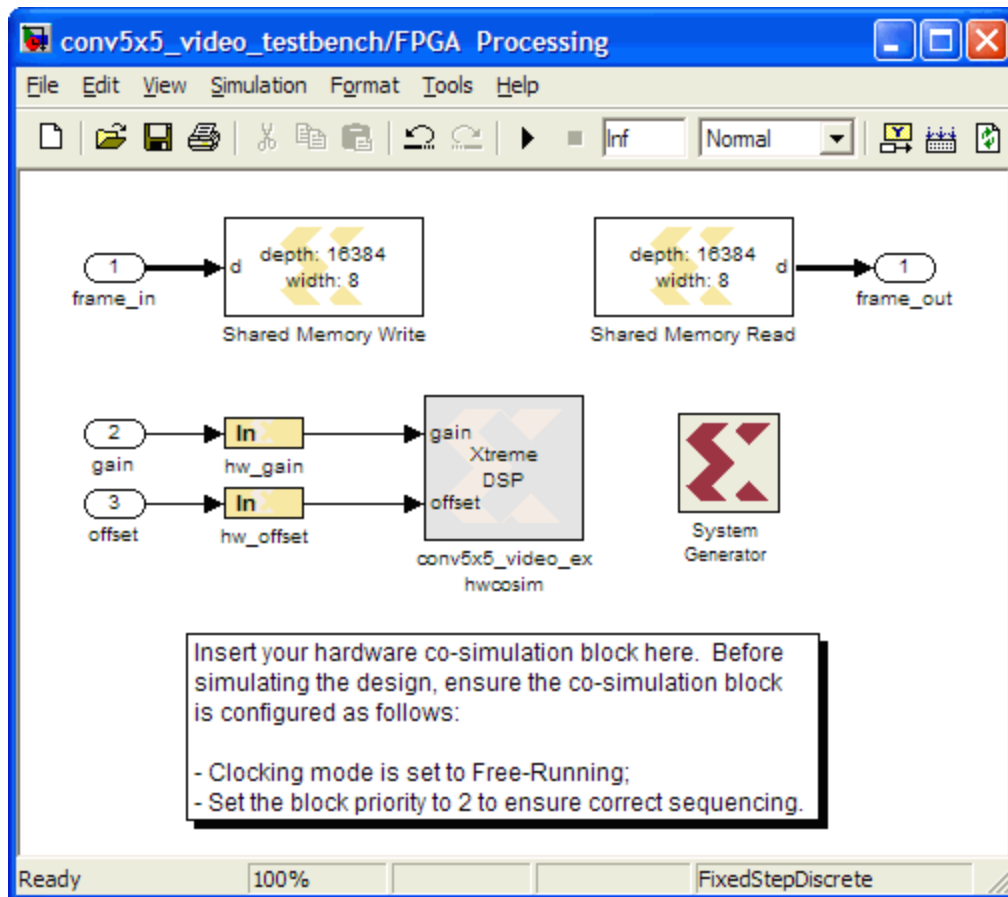
images are transferred to the FPGA and lock is granted. The FPGA processes the input buffer data and writes the output into the output buffer. Lastly the FPGA releases lock of Foo and Bar, causing the FPGA shared memory images to be transferred back to the host PC.

3. The Shared Memory Read block wakes up and requests lock of the output buffer lockable shared memory Bar. The block reads a video from the output buffer and drives its output port with the processed video frame data.



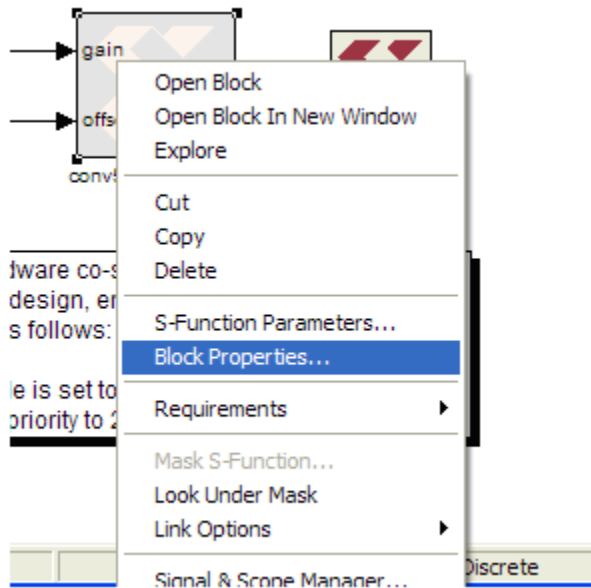
Note that the three steps listed above assume a specific sequencing of the hardware co-simulation and Shared Memory Read and Write blocks. To ensure these blocks are properly sequenced, you can set block priorities, where a lower priority block is woken up first during simulation.

9. Add the hardware co-simulation block to the testbench model in place of the turquoise placeholder residing in the FPGA Processing subsystem.



The Shared Memory Write block in the testbench is pre-configured with a priority of 1, and the Shared Memory Read block is pre-configured with a priority of 3. Since we want the hardware co-simulation block to wake up second in the simulation sequence, we must set the hardware co-simulation block priority to 2.

10. Right-click on the hardware co-simulation block, and select **Block Properties**.

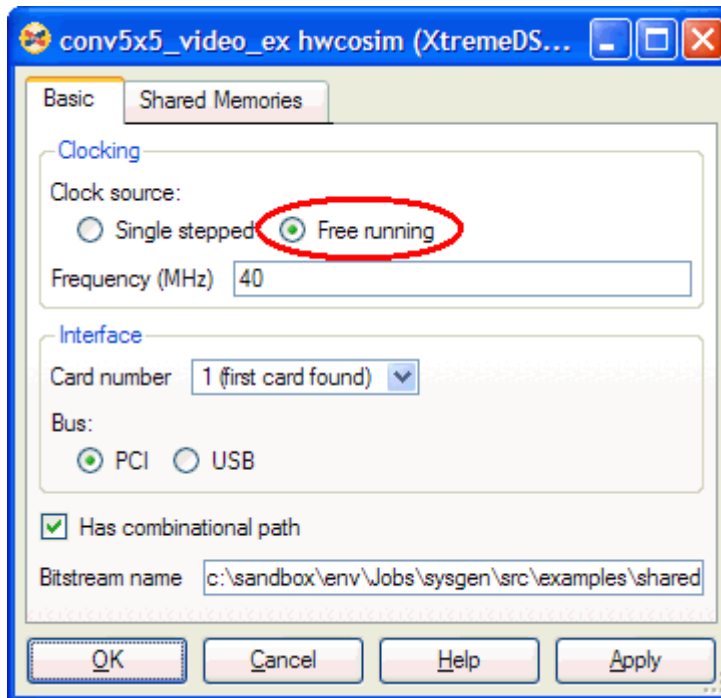


11. Specify a Priority of 2 in the Block Properties dialog box.



For high-speed processing applications, the hardware co-simulation block should be configured to operate in **Free Running** clock mode. When this mode is used, the synchronization between the FPGA and Simulink are handled entirely by the lockable shared memories. By running the FPGA in free-running mode, we allow it to run fast enough to process a complete video frame in a single Simulink cycle. Keep in mind that the hardware co-simulation block circuitry waits to acquire lock before processing data. Since the lock cannot be granted until the hardware co-simulation block is woken up, the FPGA sits idle until new data is presented in the input buffer.

12. Double-click on the hardware co-simulation block and choose a **Free Running Clock** under the **Basic** Tab.



You are now ready to simulate the design.

13. Press the Simulink **Play** button to start simulation.

Two windows will appear showing the original and filtered video streams.



The left image is the original video frame. The image on the right is the same frame that has been processed using the “smooth” filter kernel. Note that the smoothing filter is just one of several filters that can be applied to the video source.

Reloading the Kernel

The filter data path is designed so that the filter kernel can be reloaded dynamically while hardware co-simulation is running. Once the simulation is running, you may use the `xlReloadFilterCoef` function to load a new kernel. The function accepts a string kernel identifier (e.g., `sobelxy`) as an input parameter. A list of available filter kernels can be viewed by typing `help xlReloadFilterCoef` in the MATLAB console. The function is supplied as a MATLAB source file and can be found in the `$$SYGEN/examples/shared_memory/hardware_cosim/conv5x5_video` directory.

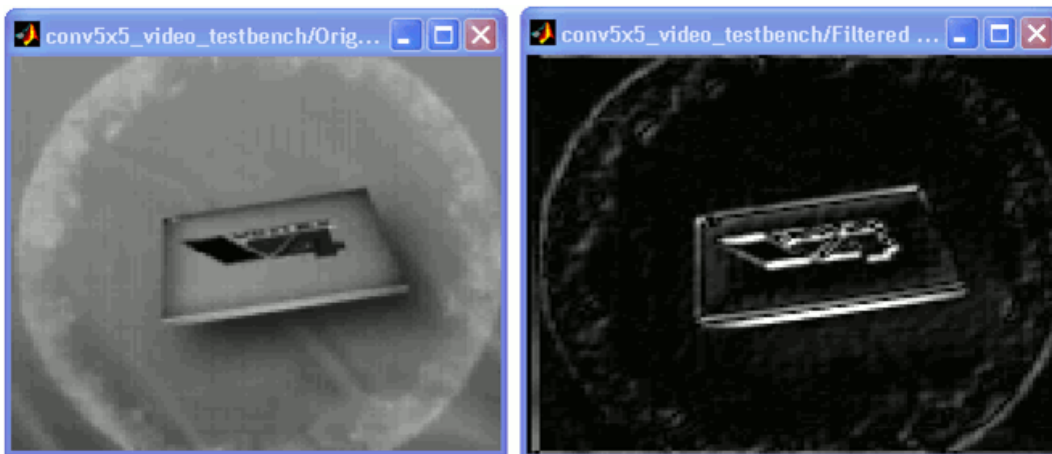
Note: once you have reloaded the filter, you may choose to adjust the coefficient gain. The gain can be adjusted using the **Coefficient Adjust** slider control at the top-level of the testbench model. This also demonstrates how System Generator's traditional, port-based, hardware co-simulation interfacing can be used in conjunction with the shared memory hardware co-simulation interfaces.

It is worthwhile to note that System Generator provides a MATLAB object interface to shared memory objects. The `xlReloadFilterCoef` function uses this object interface to write new coefficients into the unprotected shared memory named `coef_buffer` running in the FPGA. The function is fully annotated with comments that explain how the shared memory object is created, written to, and released when the operation is complete.

Note: The source code for the MATLAB object interface is supplied with the System Generator software installation, and can be found in the `$$SYGEN/examples/shared_memory/mex_function` directory. Also included in this directory is MATLAB m-code that demonstrates how the mex-function source code was built.

14. After ensuring the testbench design is running, load the SobelXY filter kernel into the FPGA by typing `xlReloadFilterCoef('sobelxy')` from the MATLAB command window.

You will now see the video output generated using the SobelX-Y kernel.

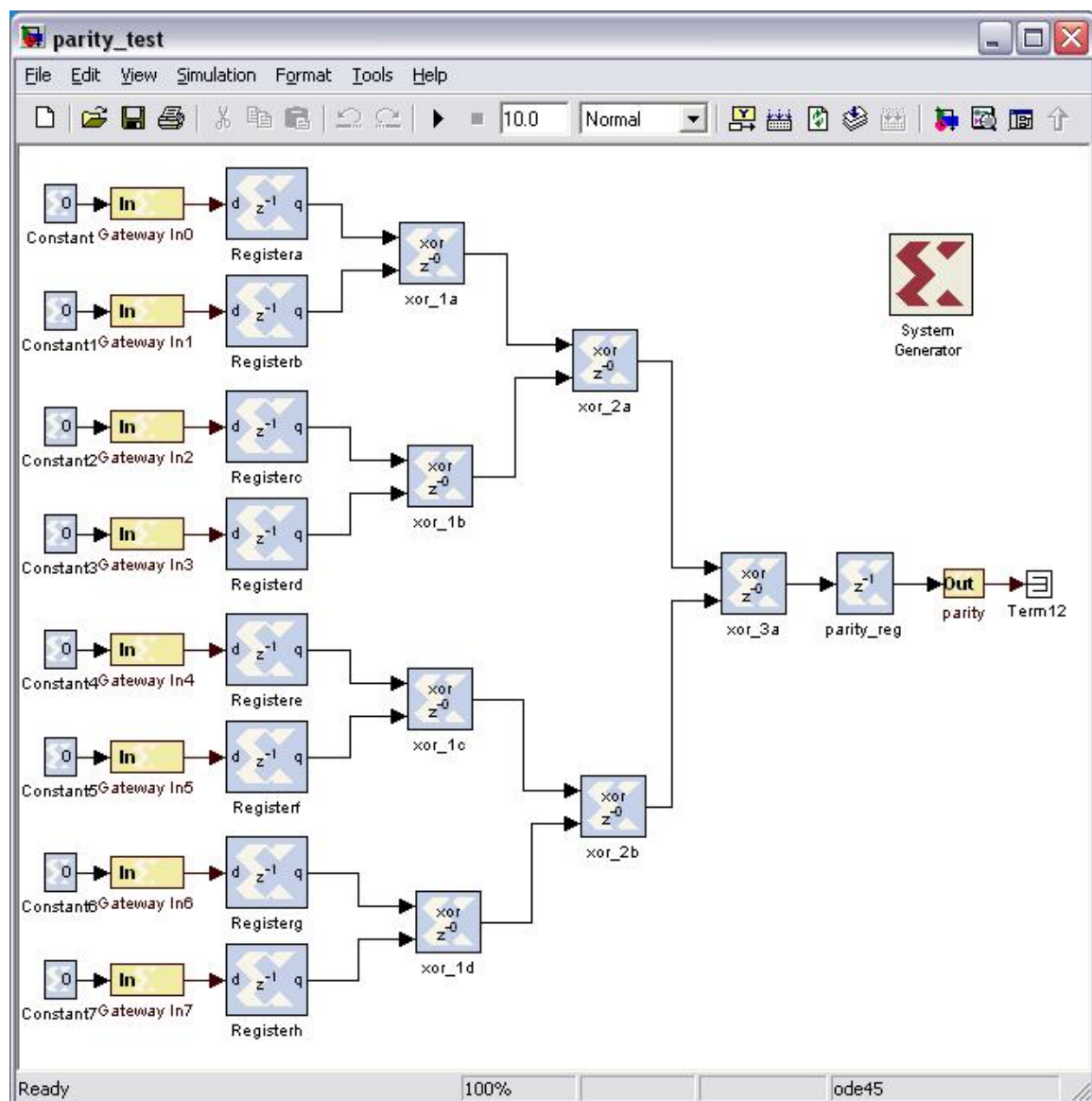


Using the System Generator Timing Analysis Tool to Meet Timing Requirements

Sometimes the hardware created by System Generator may not meet the requested timing requirements. This is typically because of a setup time violation in the design. A setup time violation means that a particular signal cannot get from the output of one synchronous element to the input of another synchronous element within the requested clock period and subject to the second synchronous element's setup time requirement.

Let us use an example to show how we would use the timing analyzer to improve circuit performance. Our example will be a parity calculator that will find the parity of a byte by using an 8-input XOR. The design can be found at:

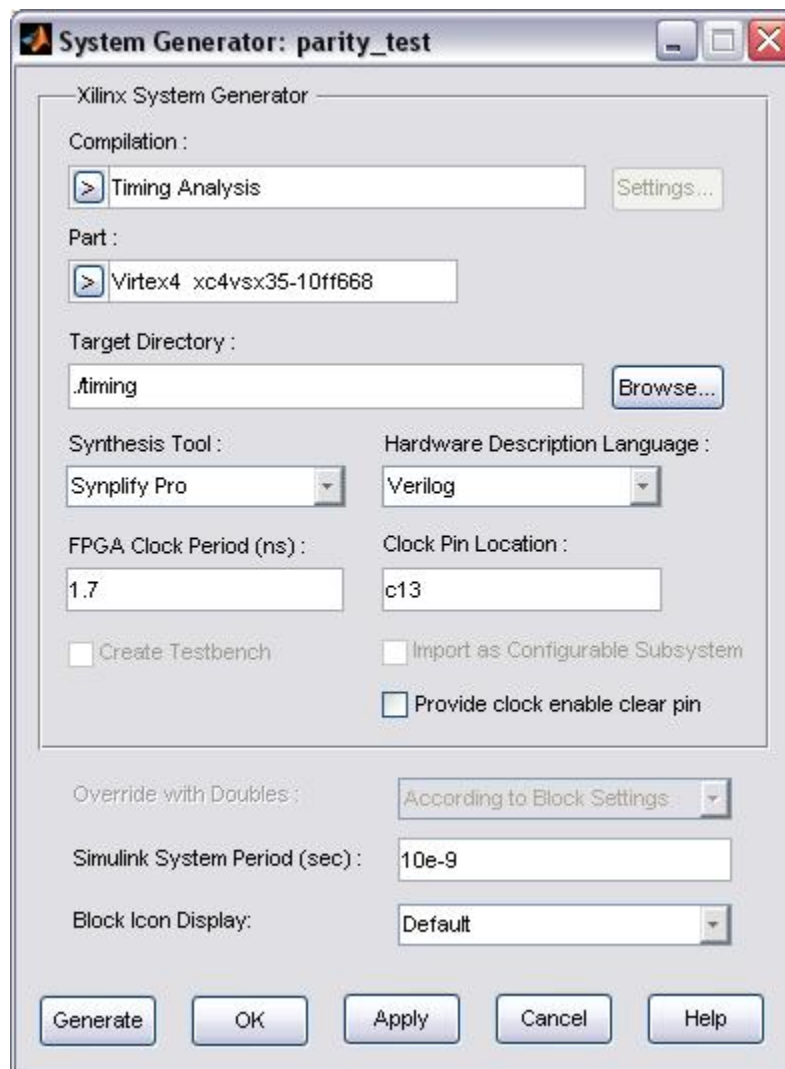
`<path_to_sysgen>/examples/timing_analysis/parity_test.mdl`



The design has eight one-bit gateway inputs that are registered by one-bit registers. These are processed by seven 2-input XOR blocks. These have a latency of zero and thus are purely combinational. The final register, *parity_reg*, registers the final result (the parity) which is connected to an output gateway. The design appears to have three levels of logic, because each path fanning in to *parity_reg* goes through three XOR blocks.

Generating the Example Design

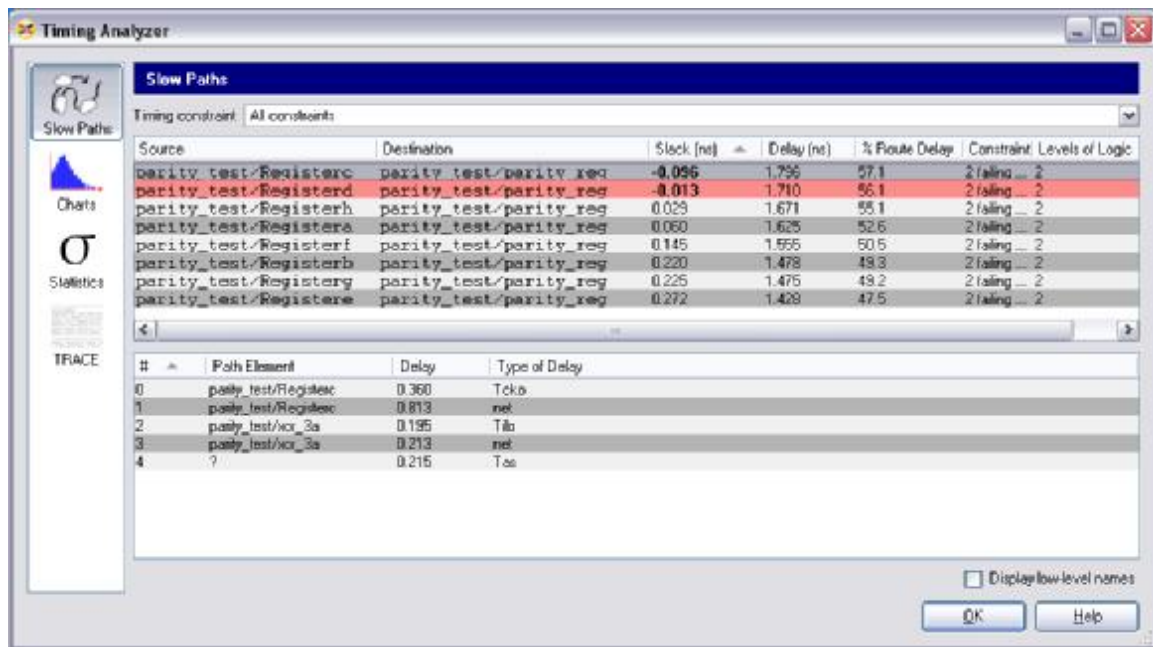
We'll generate the design using the *Timing Analysis* target and a requested period of 1.7ns (588MHz). This is admittedly a very high clock frequency, but we wish some paths to fail for demonstration purposes. We set these parameters in the System Generator token:



Note that you may also use XST as the synthesis tool. The results will be similar.

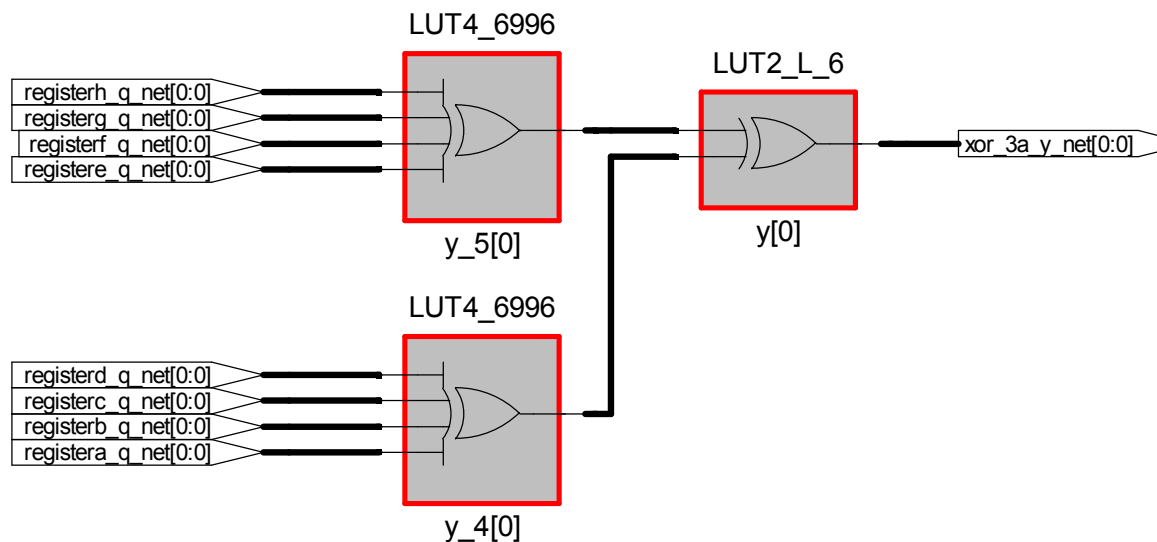
Examining the Slow Paths in the Example Design

After clicking on *Generate*, we wait, and eventually the timing analyzer window appears. Clicking on the *Slow Paths* icon yields this display:



There are two failing paths, normally highlighted in red/pink. (The top path is gray because it is selected.) The negative slack values are shown in boldface. The worst of the two fails by 96ps.

Note that there are two levels of logic in the path shown. How can this be? The System Generator diagram shows three levels of logic in all paths. The reason is that the implemented design does not correlate exactly to the System Generator diagram. In this case, the synthesizer has compressed some of the 2-bit XOR blocks into 4-input LUTs and created the 8-input XOR using only two levels of logic as shown in this Synplify schematic:



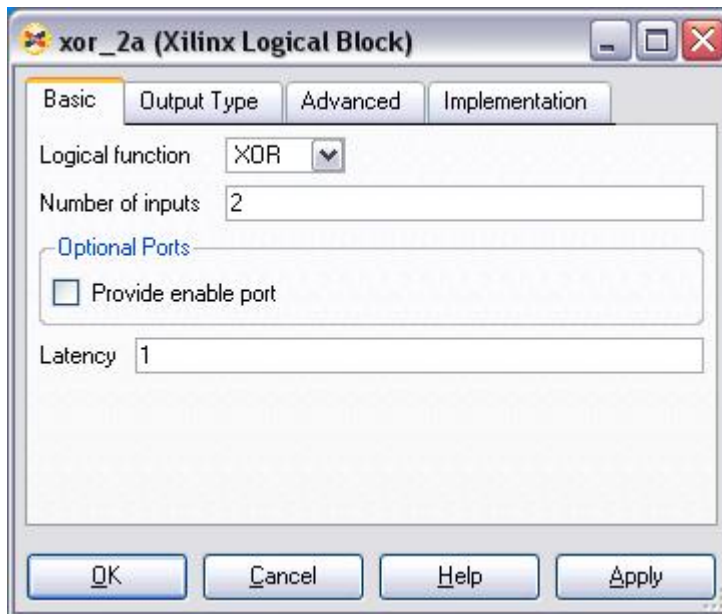
Note how the net and block names have all been munged, requiring the magic unmunging capabilities of the timing analyzer.

Also note the details of the selected path. The logic delays cannot be reduced. One of the net delays is 813ps. This could possibly be reduced by means of floorplanning, multipass PAR, or simply by increasing the PAR effort level.

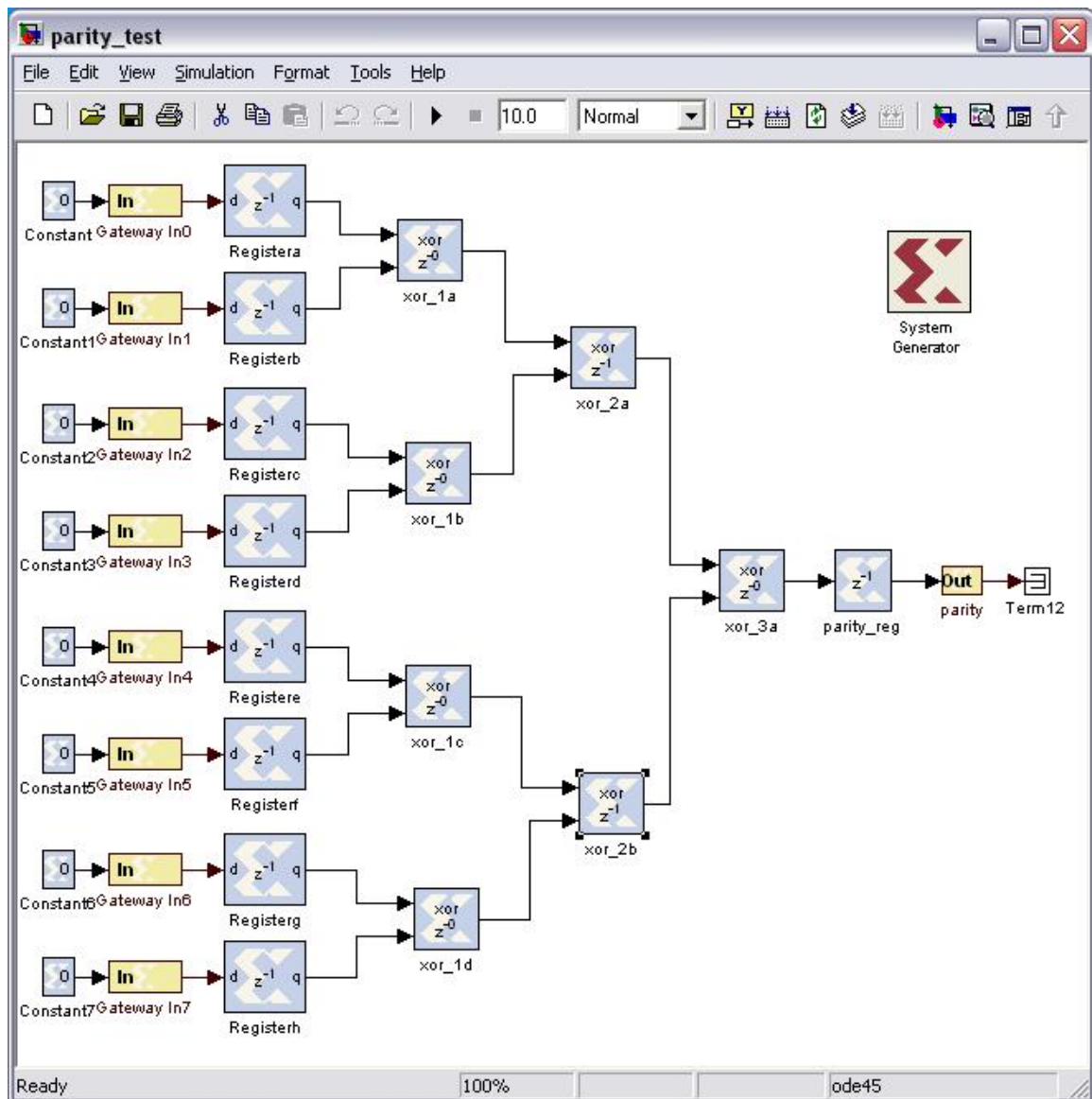
Rescuing the Design

Instead, let us attempt a more robust solution to fix the path by changing the source design. There are no feedback paths in this design, so let us assume we can add a cycle of latency and pipeline the design. There are two levels of logic in the failing paths. Any design can theoretically be reimplemented with only a single level of logic. We will do this now.

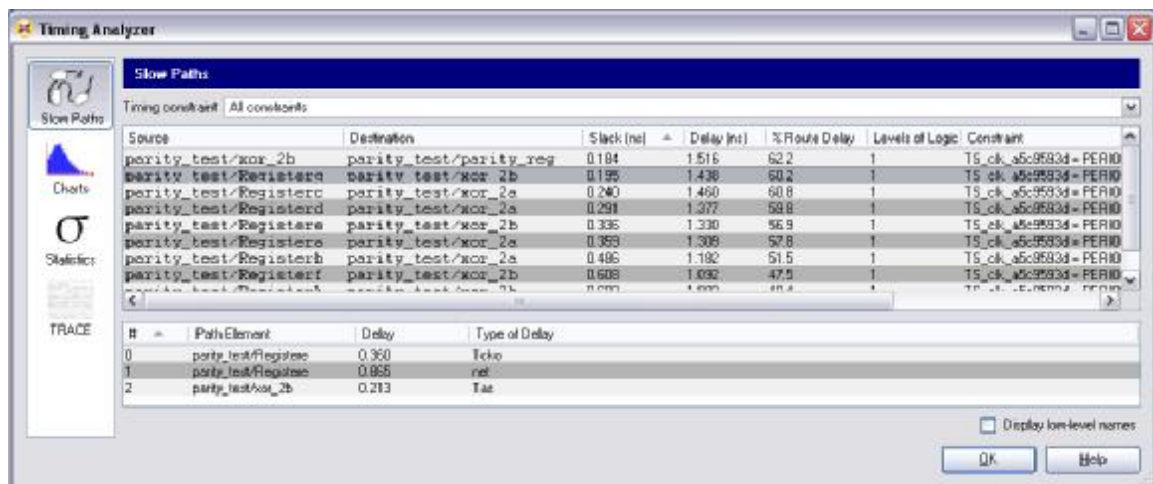
To add a pipeline stage, we will merely add latency to selected XOR blocks. By clicking on an XOR block, you may change its latency from zero to one like so:



This will add a register to the end of the XOR gate. We will change the latency on blocks *xor_2a* and *xor_2b*. We know from examining the Synplify schematic that the outputs of these blocks form the output of the first level of logic in the synthesized design. The modified System Generator looks very similar with the exception of the z^{-1} on the labels of the two modified XOR blocks, indicating their new latency.

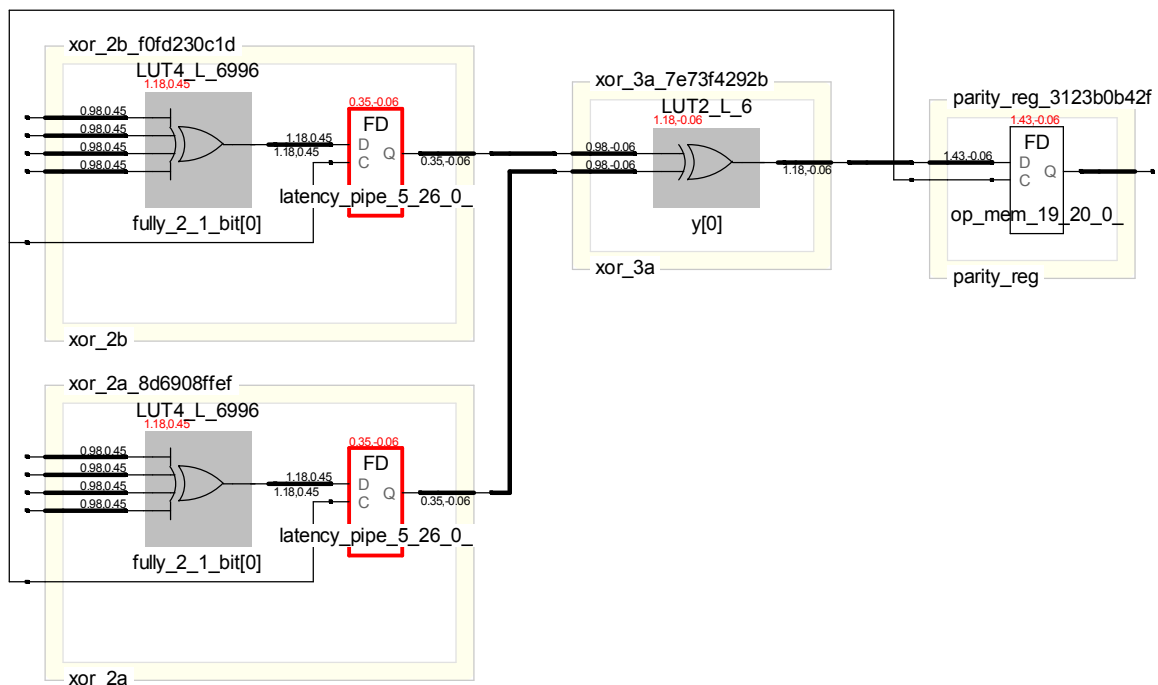


We generate this design as before and examine the slow paths:



Voilà! No more failing paths! The design has been rescued, all in record time and without using a more expensive part. Surely raises and promotions shall follow you for all your days.

Note that all paths now have but a single level of logic. What exactly has happened here? Let us examine the Synplify schematic to see how the modified circuit was synthesized:



See that there is an extra set of registers (highlighted in red) in between the two levels of logic. The circuit functions the same as before but with an additional cycle of latency.

Rescuing the Example Design with Retiming

If a cycle of latency had to be eliminated to match the latency of the original design, it might be possible to remove the final output register or the input registers. This would increase the constraints upon the paths outside the Xilinx chip (i.e., the copper paths on the PCB), but it may be feasible depending upon board-level path delays. This would be an example of *retiming*, because the latency is the same but the registers have been moved into the logic “cloud”.

System Generator Cache

Both the System Generator simulator and the design generator incorporate 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.

Setting the size

By default, the cache will use up to 500 MB of disk space to store files. To specify the amount of disk space the cache should use, set the `SYSGEN_CACHE_SIZE` environment variable to the size of the cache in megabytes. Set this number to a higher value when working on several large designs.

Setting the number of entries

The cache entry database stores a fixed number of entries. The default is 20,000 entries. To set size of the cache entry database, set the `SYSGEN_CACHE_ENTRIES` environment variable to the desired number of entries. Setting this number too small will adversely affect cache performance. Set this number to a higher value when working on several large designs.

Blockset Reference Pages

Organization of Blocks in Libraries

The Xilinx Blockset contains building blocks for constructing DSP and other digital systems in FPGAs using Simulink. The blocks are grouped into libraries according to their function, and some blocks with broad applicability (e.g., the Gateway I/O blocks) are linked into multiple libraries. The following libraries are provided:

Library	Description
Index	Includes every block in the Xilinx Blockset.
Basic Elements	Includes standard building blocks for digital logic.
Communication	Includes library of forward error correction and modulator blocks, commonly used in digital communications systems.
Control Logic	Includes blocks used for control circuitry and state machines.
Data Types	Includes blocks that convert data types (includes gateways).
DSP	Includes digital signal processing (DSP) blocks.
Math	Includes blocks that implement mathematical functions.
Memory	Includes memories.
Shared Memory	Includes blocks that allow access to the Xilinx shared memory object.
Tools	Includes “utility” blocks, e.g., code generation (System Generator block), resource estimation, HDL co-simulation, etc.

Common Options in Block Parameters Dialog Boxes

Precision

The fundamental computational mode in the Xilinx blockset is arbitrary precision fixed point arithmetic. Most blocks give you the option of choosing the precision, i.e. the number of bits and binary point position.

By default, the output of Xilinx blocks is *full* precision; that is, sufficient precision to represent the result without error. Most blocks have a *User-Defined* precision option that fixes the number of total and fractional bits

Arithmetic Type

In the **Type** field of the block parameters dialog box, you can choose unsigned or signed (two's complement) as the data type of the output signal.

Number of Bits

Fixed-point numbers are stored in data types characterized by their word size as specified by number of bits, binary point, and arithmetic type parameters. The maximum number of bits supported is 4096.

Binary Point

The binary point is the means by which fixed-point numbers are scaled. The binary point parameter indicates the number of bits to the right of the binary point (i.e., the size of the fraction) for the output port. The binary point position must be between zero and the specified number of bits.

Overflow and Quantization

When user-defined precision is selected, errors may result from overflow or quantization. Overflow errors occur when a value lies outside the representable range. Quantization errors occur when the number of fractional bits is insufficient to represent the fractional portion of a value.

The Xilinx fixed point data type supports several options for user-defined precision. For overflow the options are to saturate to the largest positive/smallest negative value, to wrap (i.e., to discard bits to the left of the most significant representable bit), or to flag an overflow as a Simulink error during simulation.

For quantization the options are to round to the nearest representable value (or to the value furthest from zero if there are two equidistant nearest representable values), or to truncate (i.e., to discard bits to the right of the least significant representable bit).

The following is an image showing the Quantization and Overflow options.

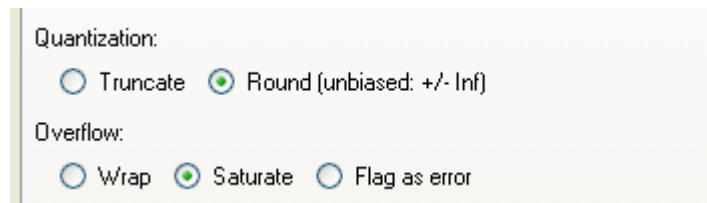


Figure 1: Specify Explicit Sample Period option (available if selected)

It is important to realize that whatever option is selected, the generated HDL model and Simulink model will behave identically.

Latency

Many elements in the Xilinx blockset have a latency option. This defines the number of sample periods by which the block's output is delayed. One sample period may correspond to multiple clock cycles in the corresponding FPGA implementation (for example, when the hardware is overclocked with respect to the Simulink model). System Generator does not perform extensive pipelining; additional latency is usually implemented as a shift register on the output of the block.

Override with Doubles

The “Override with Doubles” feature is disabled for System Generator v8.1 and 8.2.

An *Override with Doubles* option appears on many Xilinx blocks. Most Simulink blocks use double precision floating point signals and arithmetic. However, when such a signal passes through Xilinx Gateway In block, it is converted to a fixed point signal. Later, when passing through a Xilinx Gateway Out block, the signals are converted back into double precision floating point.

In the Simulink environment, the Override with Doubles option allows you to simulate the entire design in double precision floating point.

This option is useful in selecting fixed point widths or when debugging. If you detect unacceptable quantization errors with fixed point signals, you can choose to simulate your entire design, or only specific blocks, using double precision floating point signals and arithmetic operations. This option will help you discover which part of your design is responsible for the unacceptable quantization error.

You may choose Override with Doubles on a particular block. You may also choose this option for an entire sheet or an entire subsystem (the sheet plus underlying hierarchy) by instantiating a System Generator icon on the sheet, and choosing Override with Doubles as one of the System Generator block's configurable parameters.

When the output of one block with Override with Doubles set is connected to the input of another block where the option is also set, data samples are transmitted in double precision.

You can easily identify which blocks are currently set to Override with Doubles. When this option is set, affected Xilinx blocks are displayed in gray rather than the normal blue or yellow.

Provide Synchronous Reset Port

Selecting the **Provide Synchronous Reset Port** option activates an optional reset (**rst**) pin on the block.

When the reset signal is asserted the block goes back to its initial state. Reset signal has precedence over the optional enable signal available on the block. The reset signal has to run at a multiple of the block's sample rate. The signal driving the reset port must be **BOO L**.

Provide Enable Port

Selecting the `Provide Enable Port` option activates an optional enable (`en`) pin on the block.

When the enable signal is not asserted the block holds its current state until the enable signal is asserted again or the reset signal is asserted. Reset signal has precedence over the enable signal. The enable signal has to run at a multiple of the block 's sample rate. The signal driving the enable port must be Boolean.

Sample Period

Data streams are processed at a specific sample rate as they flow through Simulink. Typically, each block detects the input sample rate and produces the correct sample rate on its output. Xilinx blocks Up Sample and Down Sample provide a means to increase or decrease sample rates.

Specify Explicit Sample Period

If you select Specify Explicit Sample Period rather than the default, you may set the sample period required for all the block outputs. This is useful when implementing features such as feedback loops in your design. In a feedback loop, it is not possible for System Generator to determine a default sample rate, because the loop makes an input sample rate depend on a yet-to-be-determined output sample rate. System Generator under these circumstances requires you to supply a hint to establish sample periods throughout a loop.

The following image shows the option Specify Explicit Sample Period selected.

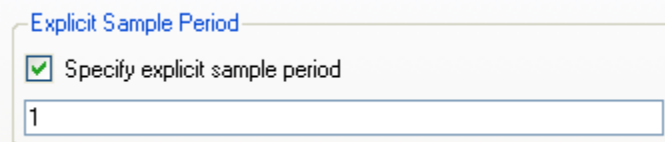


Figure 1: Specify Explicit Sample Period option (available if selected)

Use Behavioral HDL (otherwise use core)

When this checkbox is checked, the behavioral HDL generated by the MCode simulation is used instead of the structural HDL from the cores.

The MCode simulation creates the C simulation and this C simulation creates behavioral HDL. When this option is selected, it is this behavioral HDL that is used for further synthesis. When this option is not selected, the structural HDL generated from the cores and HDL templates (corresponding to each of the blocks in the model) is used instead for synthesis. Cores are generated for each block in a design once and cached for future netlisting. This capability ensures the fastest possible netlist generation while guaranteeing that the cores will be available for downstream synthesis and place and route tools.

Use Core Placement Information

If **Use Core Placement Information** is selected, the generated core includes relative placement information. This generally results in a faster implementation. Because the placement is constrained by this information, it can sometimes hinder the place and route software.

Placement Style

For the multiplier core, this option is presented if **Use Core Placement Information** is selected. This option allows specification of the layout shape in which the multiplier core will be placed in hardware. The **Rectangular** option will generate a rectangular placed core with loosely placed LUTs. **Triangular** packing will create a more compact shape with denser placement of LUTs.

FPGA Area (Slices, FFs, LUTs, IOBs, Embedded Mults, TBUFs) / Use Area Above For Estimation

These fields are used by the Resource Estimator block. The Resource Estimator gives you the ability to calculate the hardware resources needed for your System Generator design.

If you have placed a Resource Estimator in your design, you can use the **FPGA Area** field to manually enter the FPGA area utilization of a specific block. If you do not fill in these values, the Resource Estimator will calculate and fill in these values automatically.

If you wish to manually enter your own values for a specific block, then you must check the **Define FPGA area for resource estimation** box in order to force the Resource Estimator to use your entered values. Otherwise, the Resource Estimator will recalculate the FPGA Area and overwrite any values that you have entered into this field.

There are seven values available to enter into the **FPGA Area** field. You must enter or read each value in its correct position. If 'value=[1,2,3,4,5,6,7];' then:

- value(1) = Slices utilized by the block. An FPGA slice usually consists of two flip-flops, two LUTs and some associated mux, carry, and control logic.
- value(2) = Flip Flops utilized by the block.
- value(3) = Block RAM (BRAMs) utilized by the block.
- value(4) = LUTs utilized by the block.
- value(5) = IOBs consumed by the block.
- value(6) = Embedded (Emb.) multipliers utilized by the block. For example, the Virtex-II device contains embedded 18x18 multipliers.
- value(7) = Tristate Buffers (TBUFs) utilized by the block.

Only the Xilinx blocks that have a hardware cost (i.e., blocks that require physical hardware resources) will be considered by the Resource Estimator. The **FPGA Area** field is omitted from blocks with no associated hardware.

Although Slices are related to LUTs and Flops (Each Slice contains 1 LUT and 1 Flip Flop), they are entered separately since the number of packed slices will vary depending on the particular design.

Some Xilinx blocks do not support automatic resource estimation, as indicated in the Resource Estimator block documentation in this guide. The **FPGA Area** field for these blocks will not be updated automatically, and attempting to do so will cause a warning message to be displayed in the MATLAB console.

Block Reference Pages

System Generator

This block is listed in the following Xilinx Blockset libraries: Basic Elements, Tools, and Index.



The System Generator block provides control of system and simulation parameters, and is used to invoke the code generator. Every Simulink model containing any element from the Xilinx Blockset must contain at least one System Generator block. Once a System Generator block is added to a model, it is possible to specify how code generation and simulation should be handled.

For a detailed discussion on how to use the block, see *Compiling and Simulating using the System Generator Block*.

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink mode.

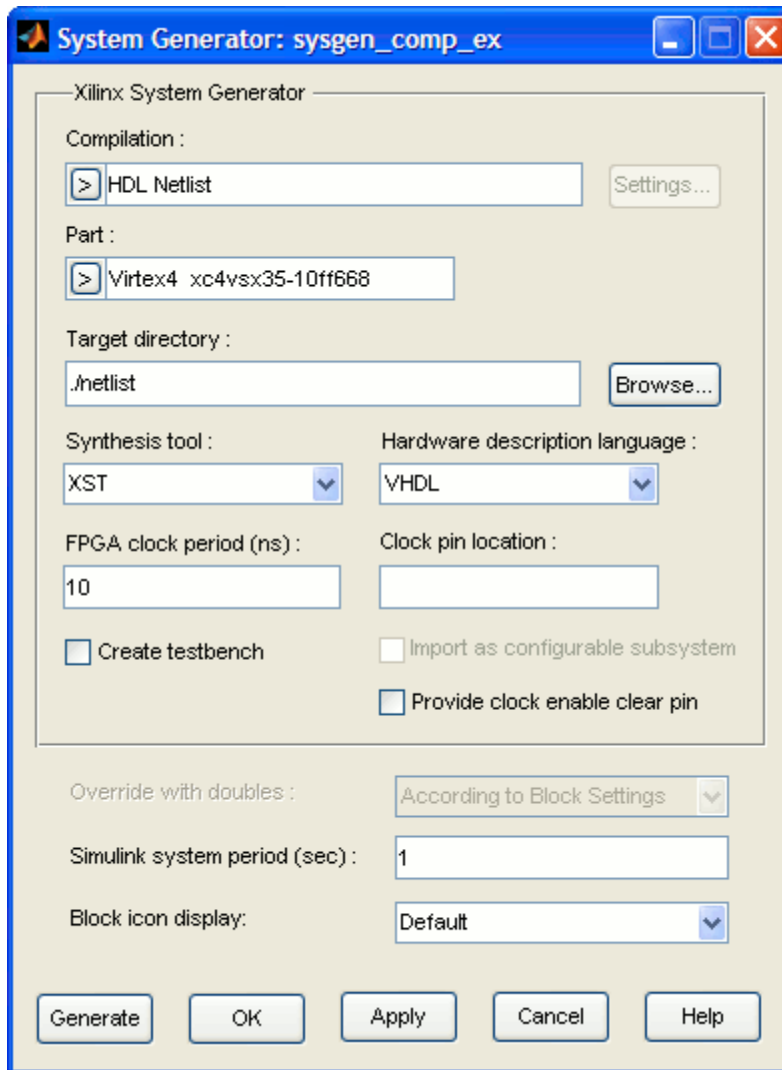
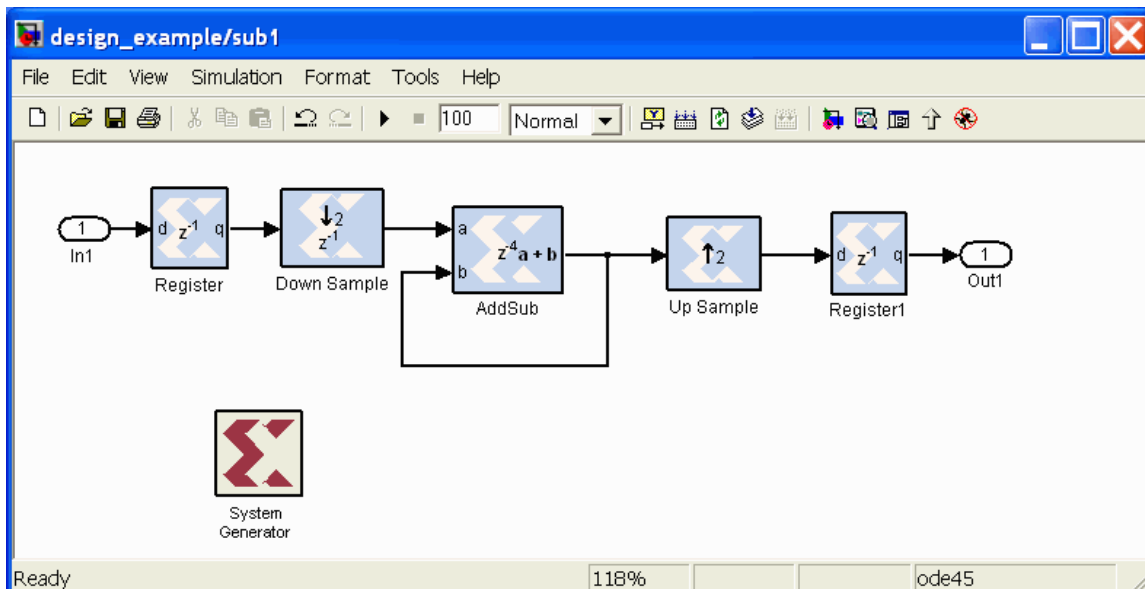


Figure 1: System Generator block parameters dialog box

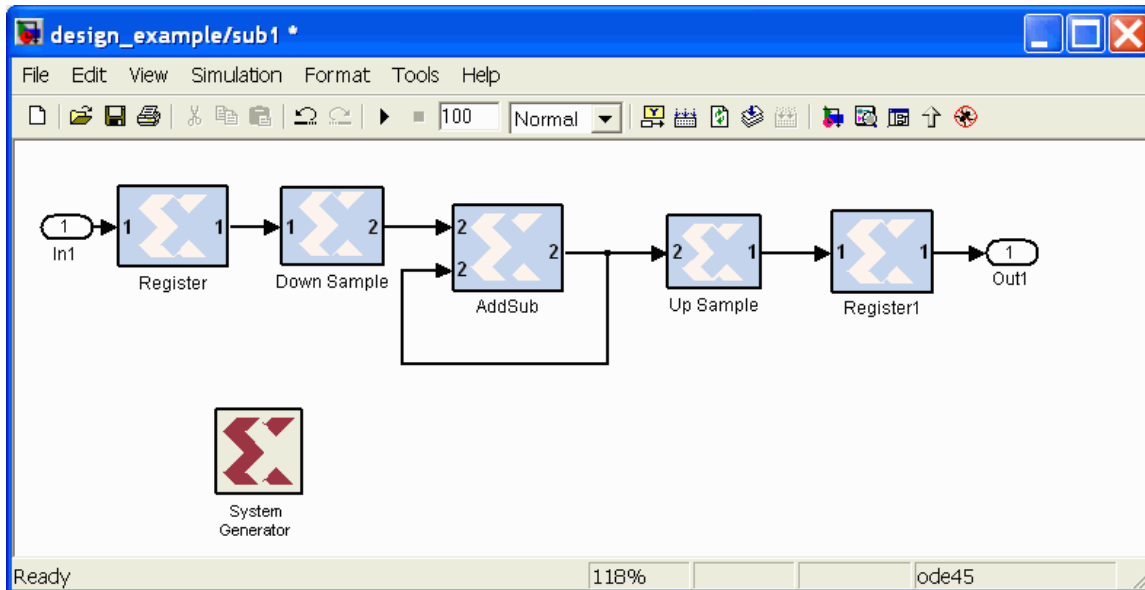
Parameters specific to the System Generator block are:

- **Compilation:** Specifies the type of compilation result that should be produced when the code generator is invoked. See System Generator Compilation Types for more details.
- **Part:** Defines the FPGA part to be used.
- **Target Directory:** Defines where System Generator should write compilation results. Because System Generator and the FPGA physical design tools typically create many files, it is best to create a separate target directory, i.e., a directory other than the directory containing your Simulink model files.
- **Synthesis Tool:** Specifies the tool to be used to synthesize the design. The possibilities are Synplicity's Synplify Pro, Synplify, and Xilinx's XST.
- **Hardware Description Language:** Specifies the HDL language to be used for compilation of the design. The possibilities are VHDL and Verilog.
- **FPGA Clock Period:** Defines the period in nanoseconds of the hardware clock. The value need not be an integer. The period is passed to the Xilinx implementation tools through a constraints file, where it is used as the global PERIOD constraint. Multicycle paths are constrained to integer multiples of this value.
- **Clock Pin Location:** Defines the pin location for the hardware clock. This information is passed to the Xilinx implementation tools through a constraints file.

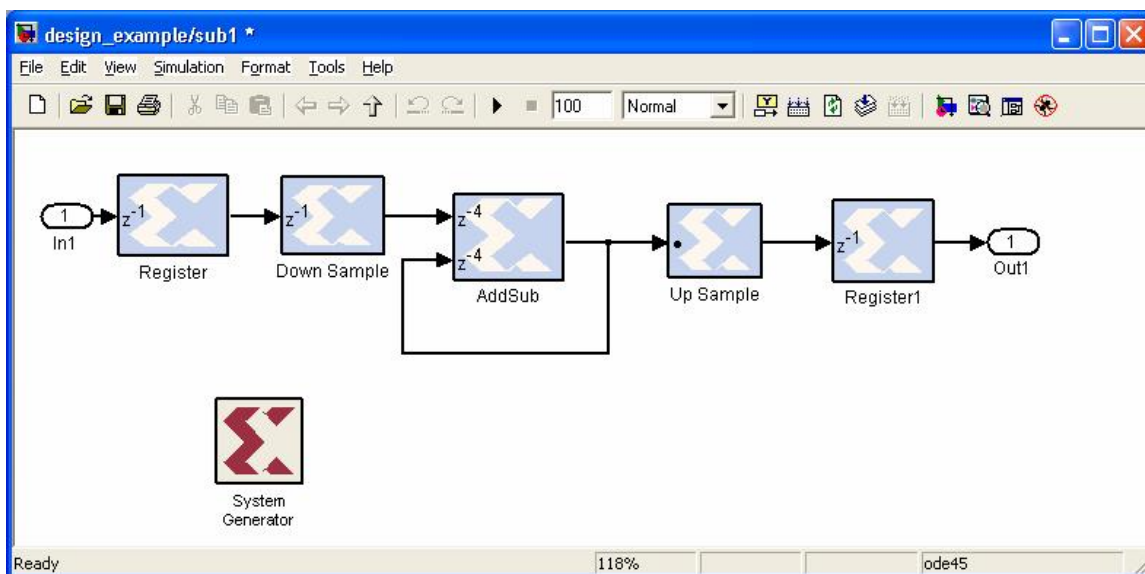
- **Create Testbench:** This instructs System Generator to create a HDL testbench. Simulating the testbench in an HDL simulator compares Simulink simulation results with ones obtained from the compiled version of the design. To construct test vectors, System Generator simulates the design in Simulink, and saves the values seen at gateways. The top HDL file for the testbench is named `<name>_testbench.vhd/.v`, where `<name>` is a name derived from the portion of the design being tested.
- **Import as Configurable Subsystem:** Tells System Generator to do two things: 1) Construct a block to which the results of compilation are associated, and 2) Construct a configurable subsystem consisting of block and the original subsystem from which the block was derived. See Configurable Subsystems and System Generator for details.
- **Provide clock enable clear pin:** This instructs System Generator 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. Capability to reset clock enable generations logic allows designs to have dynamic control for specifying the beginning of data path sampling. See Resetting Auto Generated Clock Enable Logic for details.
- **Override with Doubles:** Specifies that all calculations within the scope of the block should be done using double precision arithmetic. Calculating using double precision is described in detail in Override with Doubles.
- **Simulink System Period:** 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 propagation rules, or implied by a *hardware oversampling rate* in blocks with this option. In the latter 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 may be modified only in a master block.
- **Block Icon Display:** Specifies the type of information to be displayed on the block icon. The block icon is updated with the selected display option after the design has been compiled. The various display options are described in details below:
 - **Default:** Displays the default block icon. A block's default icon is derived from the `xbsIndex` library.



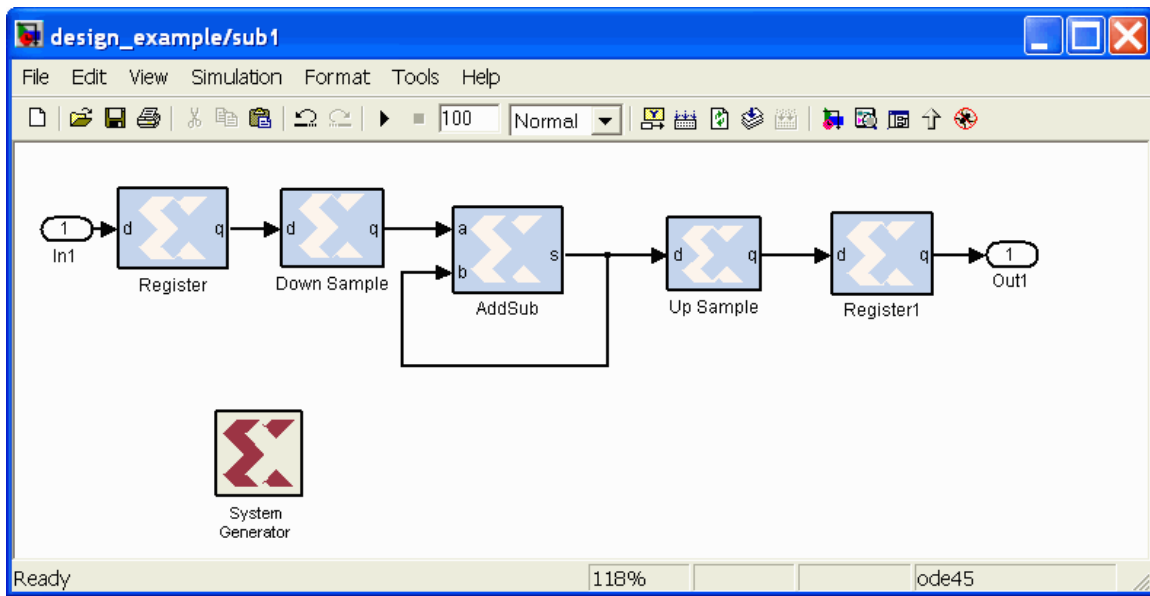
- Sample rates:** Displays the normalized sample periods for all the input and output ports on the block. For example, if the **Simulink System Period** is set to 4 and the sample period propagated to a block port is 4 then the normalized period that is displayed for the block port will be 1 and if the period propagated to the block port is 8 then the sample period displayed would be 2 i.e. a larger number indicates a slower rate.



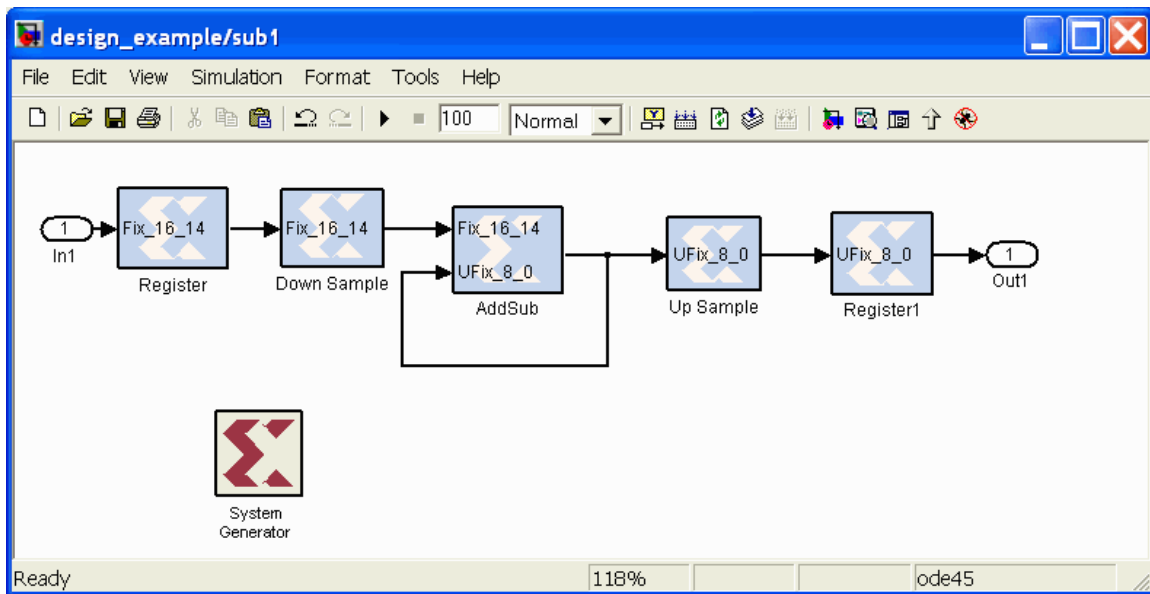
- Pipeline stages:** Displays the latency information from the input ports of the block. The displayed pipeline stage might not be accurate for certain high level blocks such as the FFT, RS Encoder/ Decoder, Viterbi Decoder, etc. In this case the displayed pipeline information can be used to determine whether a block has a combinational path from the input to the output. For example, the **Up Sample** block in the figure below shows that it has a combinational path from the input to the output port.



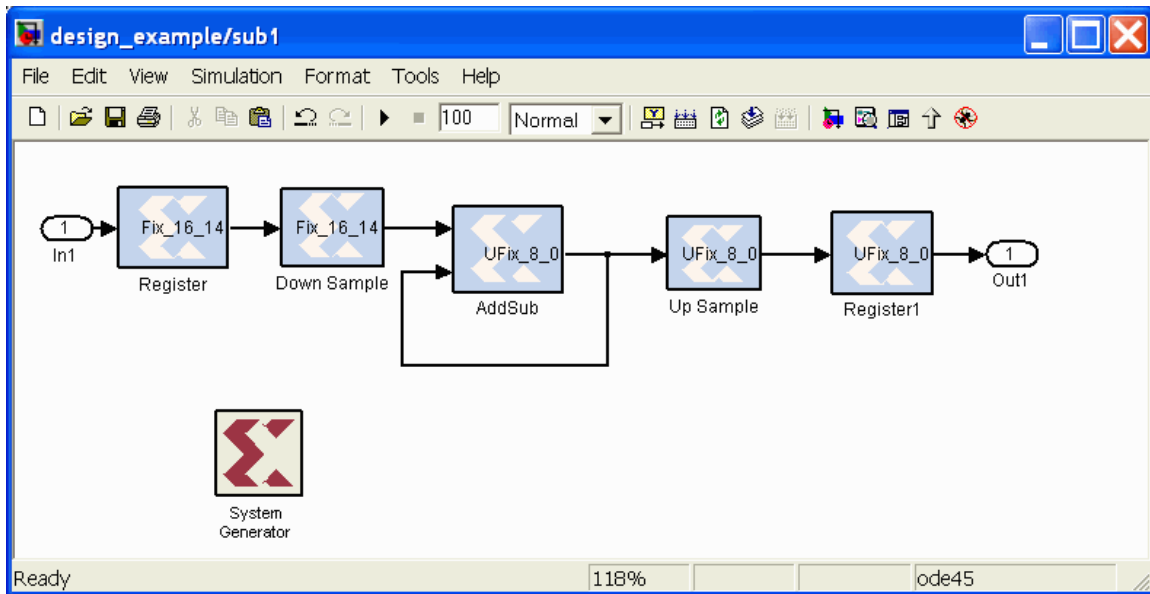
- HDL port names:** Displays the corresponding HDL input and output port names on the netlisted entity for the block.



- **Input data types:** Displays the data types of the signals driving the input port of the block.

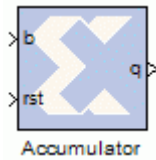


- **Output data types:** Displays the data types for the output ports on the block.



Accumulator

This block is listed in the following Xilinx Blockset libraries: *Math and Index*.



The Xilinx Accumulator block implements an adder or subtractor based scaling accumulator. The block's current input is accumulated with a scaled current stored value. The scale factor is a block parameter.

Block Interface

The block has an input **b** and an output **q**. The output must have the same width as the input data. The output will have the same arithmetic type and binary point position as the input. The output **q** is calculated as follows:

$$q(n) = \begin{cases} 0 & \text{if } rst=1 \\ q(n-1) \times FeedbackScaling \pm b(n-1) & \text{otherwise} \end{cases}$$

A subtractor based accumulator replaces addition of the current input $b(n)$ with subtraction.

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

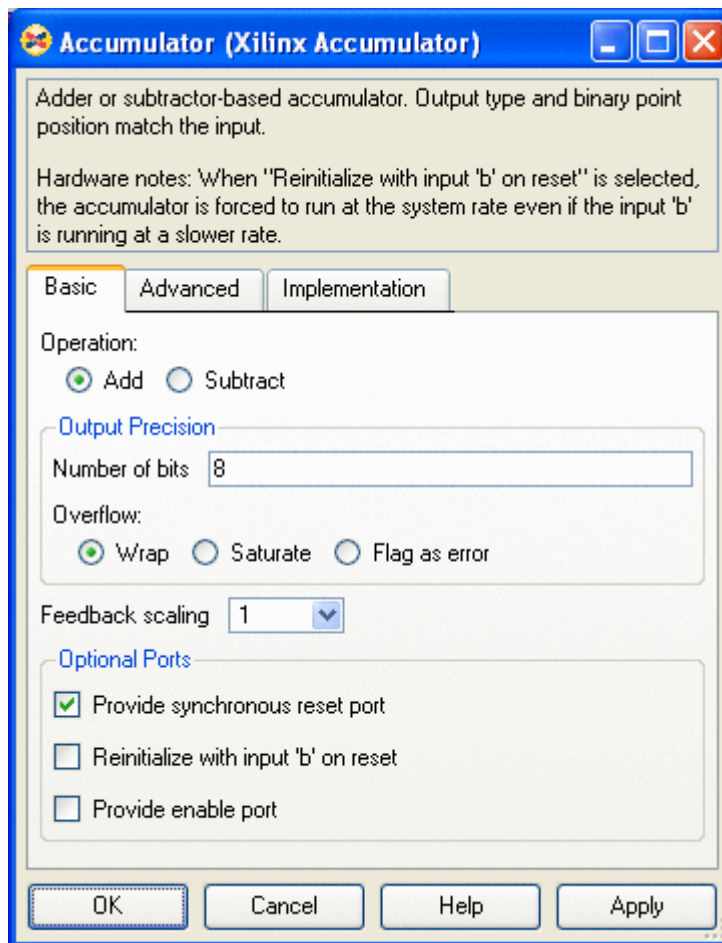


Figure 1: Accumulator block parameters dialog box

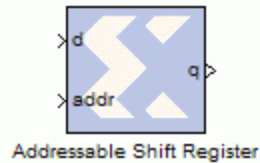
Parameters specific to the block are:

- **Operation:** This determines whether the block is adder or subtractor based.
- **Feedback scaling:** specifies the feedback scale factor to be one of the following: 1, 1/2, 1/4, 1/8, 1/16, 1/32, 1/64, 1/128, or 1/256.
- **Reinitialize with input 'b' on reset:** when selected, the output of the accumulator is reset to the data on input port b. When not selected, the output of the accumulator is reset to zero. This option is available only when the block has a reset port. Using this option has clock speed implications if the accumulator is in a multi-rate system. In this case the accumulator is forced to run at the system rate since the clock enable (CE) signal driving the accumulator runs at the system rate and the reset to input operation is a function of the CE signal.

Other parameters used by this block are explained in the Common Parameters section at the beginning of this chapter. The accumulator block always has a latency of 1.

Addressable Shift Register

This block is listed in the following Xilinx Blockset libraries: *Basic Elements, Memory, and Index.*



The Xilinx Addressable Shift Register block is a variable-length shift register in which any register in the delay chain can be addressed and driven onto the output data port.

The block operation is most easily thought of as a chain of registers, where each register output drives an input to a multiplexer, as shown below. The multiplexer select line is driven by the address port (*addr*). The output data port is shown below as *q*.

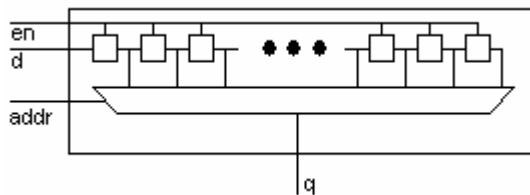


Figure 1: Addressable Shift Register

The Addressable Shift Register has a maximum depth of 1024 and a minimum depth of 2. The address input port, therefore, can be between 1 and 10 bits (inclusive). The data input port width must be between 1 and 255 bits (inclusive) when this block is implemented with the Xilinx LogiCORE (i.e. when “use behavioral HDL (otherwise use core)” is unchecked).

The Addressable Shift Register provides an optional enable port (*en*). As shown in the figure above, the enable signal is connected only to the registers. A change on the address port will cause the data output port to change, even if the enable is not asserted.

In hardware, the address port is asynchronous relative to the output port. In the block S-function, the address port is therefore given priority over the input data port, i.e. on each successive cycle, the addressed data value is read from the register and driven to the output before the shift operation occurs. This order is needed in the Simulink software model to guarantee one clock cycle of latency between the data port and the first register of the delay chain. (If the shift operation were to come first, followed by the read, then there would be no delay, and the hardware would be incorrect.)

Block Interface

The block interface (inputs and outputs as seen on the Addressable Shift Register icon) are as follows:

Input signals:

d	data input
addr	address
en	enable signal

Output signals:

q	data output

Block Parameters Dialog Box

The Addressable Shift Register Block Parameters Dialog Box can be invoked by double-clicking the icon in your Simulink model.

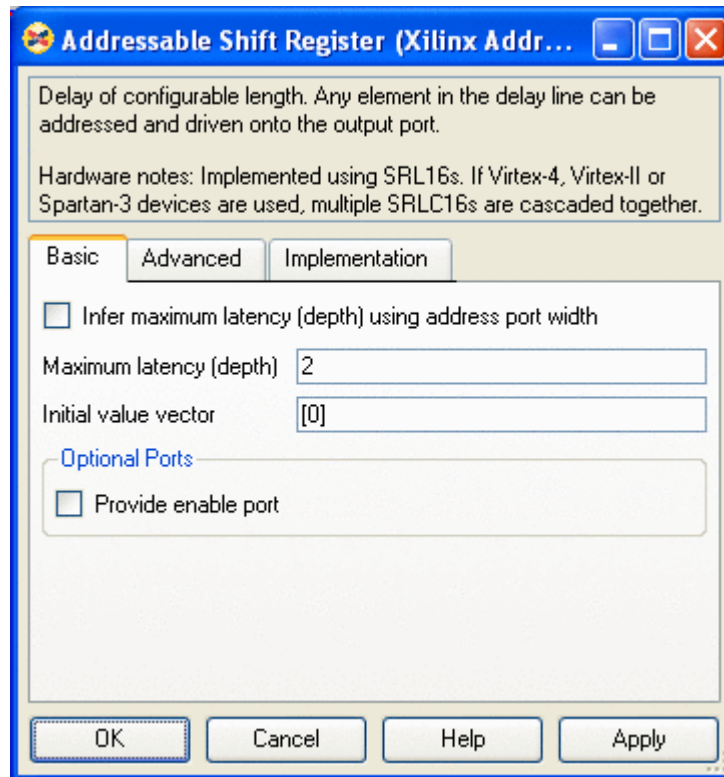


Figure 2: Addressable Shift Register block parameters dialog box

Parameters specific to the Addressable Shift Register block are:

- **Infer maximum latency (depth) using address port width:** you can choose to allow the block to automatically determine the depth or maximum latency of the shift-register based on the bit-width of the address port.
- **Maximum latency (depth):** in the case that the maximum latency is not inferred (previous option), the maximum latency can be set explicitly.
- **Initial value vector:** specifies the initial register values. When the vector is longer than the shift register depth, the vector's trailing elements are discarded. When the shift register is deeper than the vector length, the shift register's trailing registers are initialized to zero.

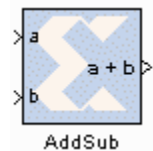
Other parameters used by this block are explained in the Common Parameters section at the beginning of this chapter.

Xilinx LogiCORE

When not using behavioral HDL this block uses the Xilinx LogiCORE Ram-based Shift Register. The data input port width must be between 1 and 255 bits (inclusive) when using the LogiCORE.

AddSub

This block is listed in the following Xilinx Blockset libraries: *Math and Index*.



The Xilinx AddSub block implements an adder/subtractor. The operation can be fixed (Add or Subtract) or changed dynamically under control of the SUB mode signal.

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

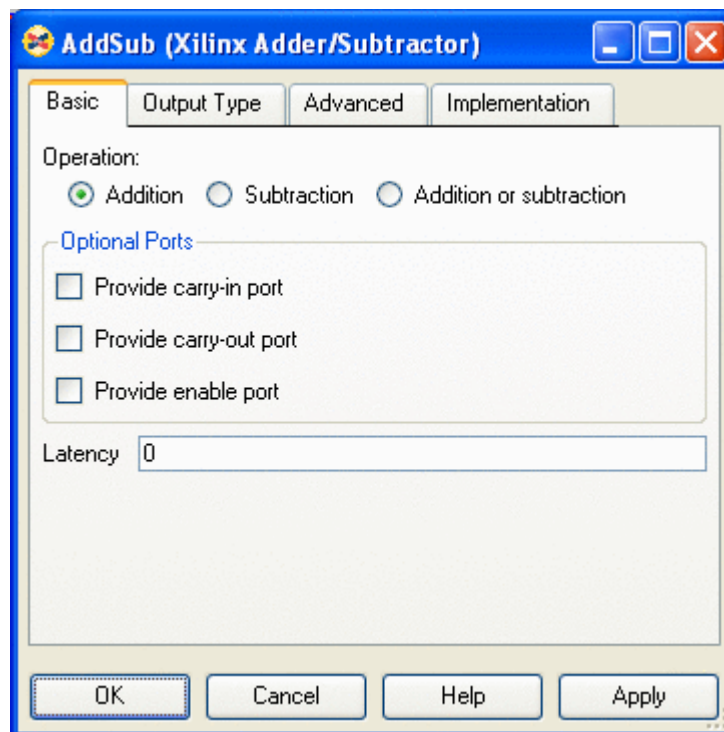


Figure 1: Addsub block Basic Tab

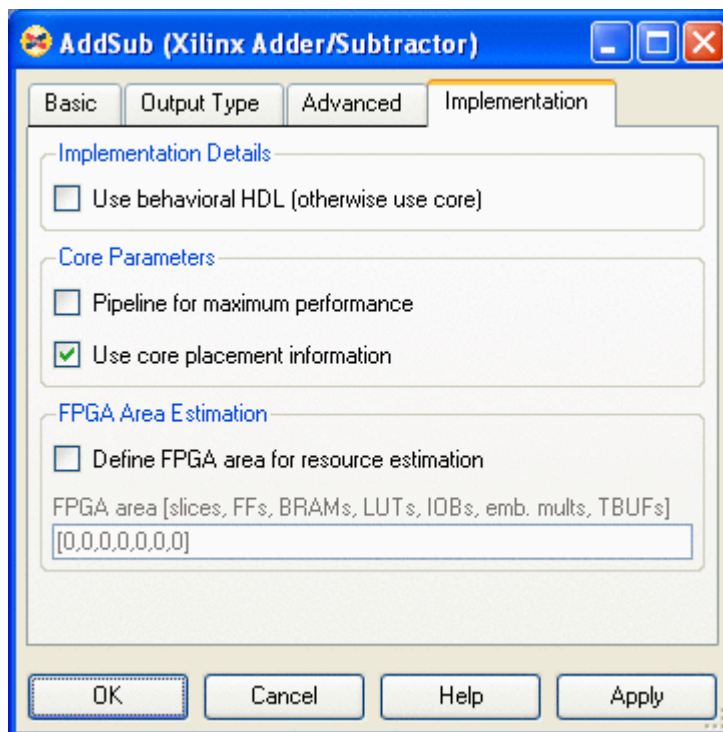


Figure 2: Addsub block Implementation Tab

Parameters specific to the AddSub block are:

- **Operation (Basic):** Specifies the block operation to be Addition, Subtraction, or Addition/Subtraction. When Addition/Subtraction is selected, the block operation is determined by the sub input port, which must be driven by a Boolean signal. When the sub input is 1, the block performs subtraction. Otherwise, it performs addition.
- **Provide carry-in Port (Basic):** When selected, allows access to the carry-in port, cin. The carry-in port is available only when user defined precision is selected and the binary point of the inputs is set to zero.
- **Provide carry-out Port (Basic):** When selected, allows access to the carry-out port, cout. The carry-out port is available only when user defined precision is selected, the inputs and output are unsigned, and the number of output integer bits equals x , where $x = \max(\text{integer bits } a, \text{integer bits } b)$.
- **Pipeline for maximum performance (Implementation):** The Xilinx LogiCORE can be internally pipelined to improve speed. Selecting this option will ensure the maximum usable latency will be used as internal core pipeline stages.

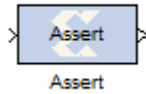
Other parameters used by this block are explained in the Common Parameters section at the beginning of this chapter.

Xilinx LogiCORE

The when not using behavioral HDL this block uses the Xilinx LogiCORE AddSub.

Assert

This block is listed in the following Xilinx Blockset libraries: *Index*.



The Xilinx Assert block is used to assert a rate and/or a type on a signal. This block has no cost in hardware and can be used to resolve rates and/or types in situations where designer intervention is required.

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

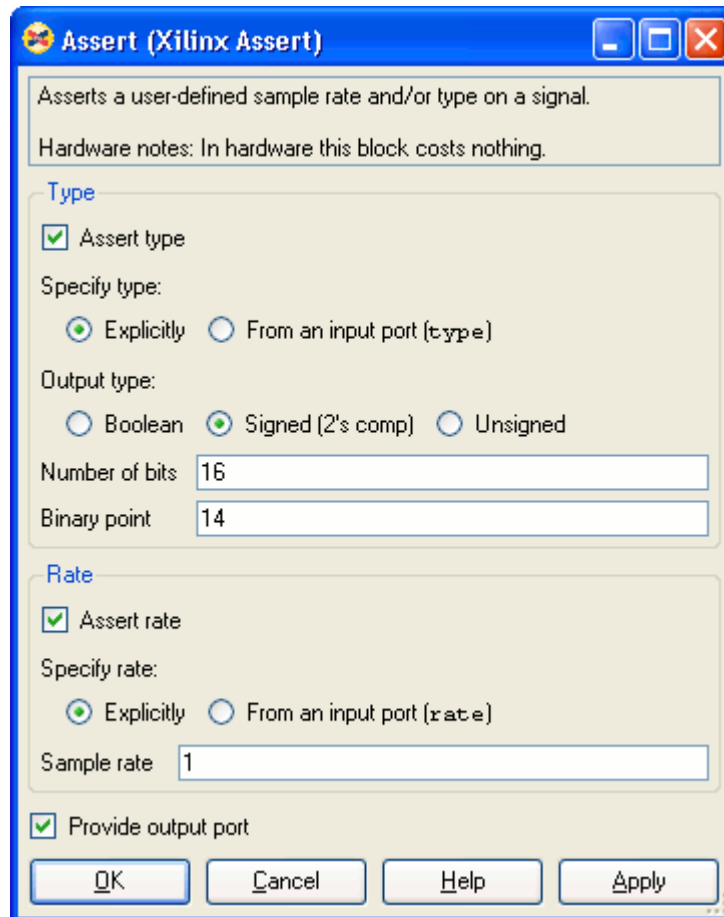


Figure 1: Assert block parameters dialog box

Parameters specific to the Assert block are:

- **Assert type:** specifies whether or not the block will assert that the type at its input is the same as the type specified. If the types are not the same, an error message is reported.
- **Specify type:** specifies whether or not the type to assert will be provided from a signal connected to an input port named `type` or whether it will be specified explicitly from parameters in the Assert block dialog box.
- **Assert rate:** specifies whether or not the block will assert that the rate at its input is the same as the rate specified. If the rates are not the same, an error message is reported.
- **Specify rate:** specifies whether or not the initial rate to assert will be provided from a signal connected to an input port named `rate` or whether it will be specified explicitly from the Sample rate parameter in the Assert block dialog box.

- **Provide output port:** specifies whether or not the block will feature an output port. The type and/or rate of the signal presented on the output port is the type and/or rate specified for assertion.

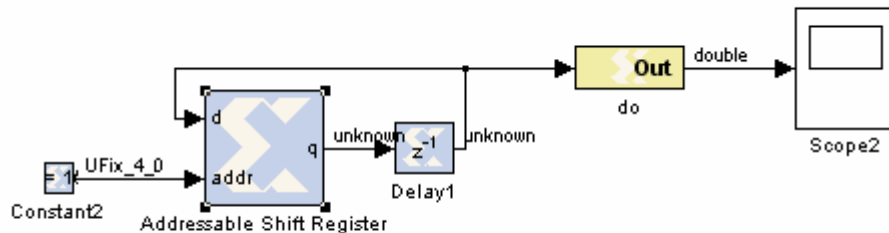
Other parameters used by this block are explained in the Common Parameters section at the beginning of this chapter. The **Output Type** parameter in this block uses the same description as the **Arithmetic Type** description.

The Assert block does not use a Xilinx LogiCORE and does not use resources when implemented in hardware.

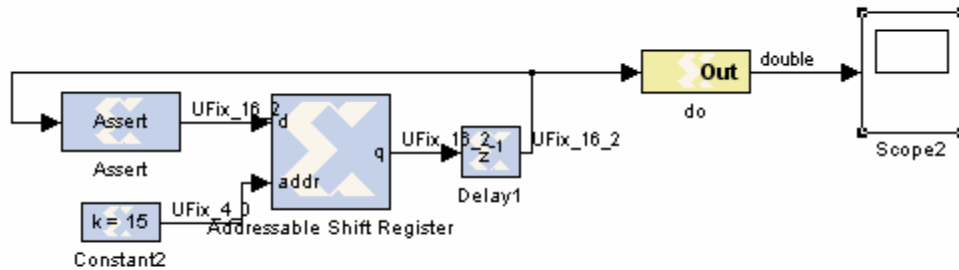
Using the Assert block to resolve rates and types

In cases where the simulation engine cannot resolve rates or types, the Assert block can be used to force a particular type or rate. In general this may be necessary when using components which use feedback and act as a signal source. For instance, the circuit below requires an Assert block to force the rate and type of an SRL16. In this case we use an Assert block to 'seed' the rate which is then propagated back to the SRL16 input through the SRL16 and back to the Assert block. The design below fails with the following message when the Assert block is not used.

"The data types could not be established for the feedback paths through this block. You may need to add "Assert" blocks to instruct the system how to resolve types."



To resolve this error an Assert block is introduced in the feedback path as shown:

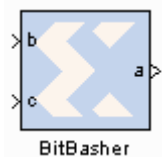


In the example, the Assert block is required to resolve the type, but the rate could have been determined by assigning a rate to the constant clock. The decision whether to use Constant blocks or Assert blocks to force rates is arbitrary and can be determined on a case by case basis.

As of System Generator v8.1, System Generator resolves rates and types deterministically, however in some cases, the use of Assert blocks may be necessary for some System generator components, even if they are resolvable. These blocks may include black box components and certain IP blocks.

BitBasher

This block is listed in the following Xilinx Blockset libraries: Basic Elements, Data Types and Index.



The Xilinx BitBasher block performs slicing, concatenation and augmentation of inputs attached to the block. The operation to be performed is described using Verilog syntax which will be detailed in this document. The block may have up to four output ports and the number of output ports is equal to the number of expressions. The block does not cost anything in hardware.

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

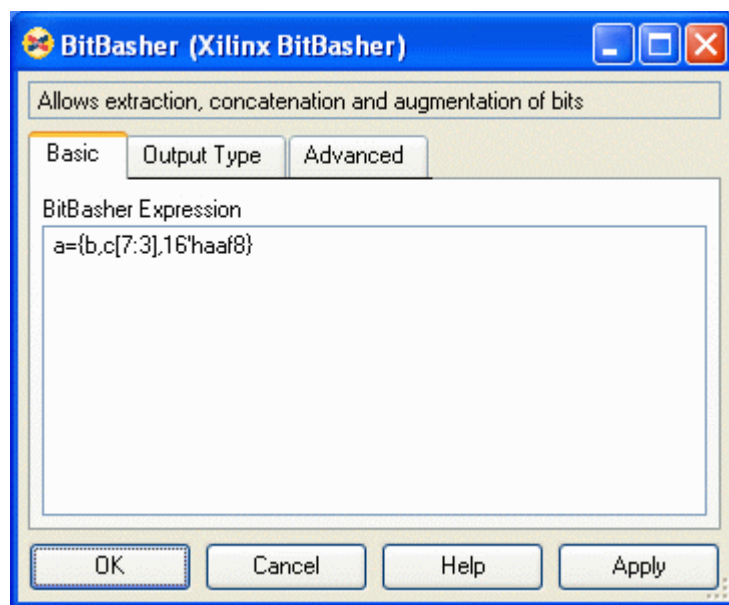


Figure 1:BitBasher block Basic Tab

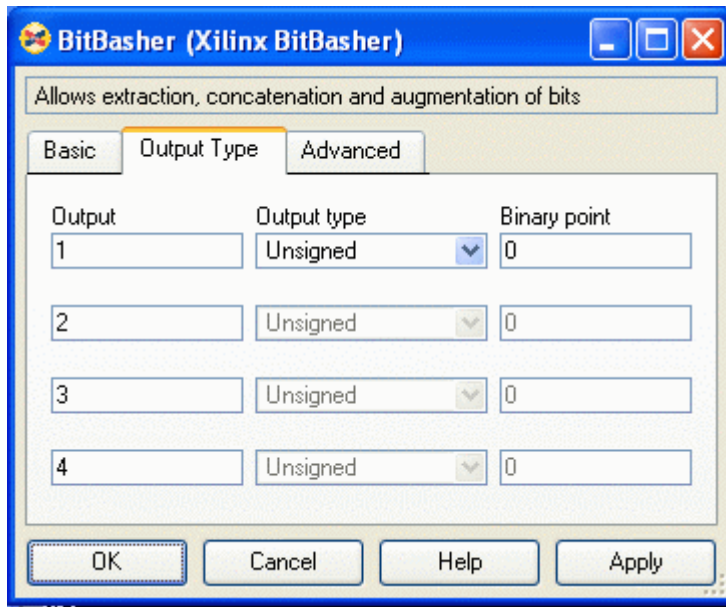


Figure 2: BitBasher block Output Type Tab

Parameters specific to the block are:

- **BitBasher Expression (Basic):** Bitwise manipulation expression based on Verilog Syntax. Multiple expressions (limited to a maximum of 4) can be specified using new line as a separator between expressions.
- **Output (Output Type):** This refers to the port on which the data type is specified
- **Output type (Output Type):** Arithmetic type to be forced onto the corresponding output
- **Binary Point (Output Type):** Binary point location to be forced onto the corresponding output

Supported Verilog Constructs

The BitBasher block only supports a subset of Verilog expression constructs that perform bitwise manipulations including slice, concatenation and repeat operators. All specified expressions must adhere to the following template expression:

$$output\_var = \{bitbasher\_expr\}$$

bitbasher_expr: A slice, concat or repeat expression based on Verilog syntax or simply an input port identifier.

output_var: The output port identifier. An output port with the name *output_var* will appear on the block and will hold the result of the wire expression *bitbasher_expr*

Concat

$$output\_var = \{bitbasher\_expr1, bitbasher\_expr2, bitbasher\_expr3\}$$

The concat syntax is supported as shown above. Each of *bitbasher_exprN* could either be an expression or simply an input port identifier.

The following are some examples of this construct:

```
a1 = {b, c, d, e, f, g}
a2 = {e}
a3 = {b, {f, c, d}, e}
```

Slice

```
output_var = {port_identifier[bound1:bound2]}...(1)
output_var = {port_identifier[bitN]}...(2)
```

port_identifier : The input port from which the bits are extracted.

bound1, bound2 : Non-negative integers that lie between 0 and (bit-width of *port_identifier* – 1)

bitN : Non-negative integers that lie between 0 and (bit-width of *port_identifier* – 1)

As shown above there are two schemes to extract bits from the input ports. If a range of consecutive bits need to be extracted then the first (1) expression should be used. If only one bit is to be extracted then the alternative scheme (2) is also provided.

The following are some examples of this construct:

```
a1 = {b[7:3]}
```

a1 holds bits 7 through 3 of input *b* in the same order in which they appear in bit *b* ie if *b* were 110110110 then *a1* would be 1011.

```
a2 = {b[3:7]}
```

a2 holds bits 7 through 3 of input *b* in the reverse order in which they appear in bit *b* ie if *b* were 110110110 then *a1* would be 1101.

```
a3 = {b[5]}
```

a3 holds bit 5 of input *b*.

```
a4 = {b[7:5], c[3:9], {d, e}}
```

The above expression makes use of a combination of slice and concat constructs. Bits 7 through 5 of input *b*, bits 3 through 9 of input *c* and all the bits of *d* and *e* are concatenated.

Repeat

```
output_var = {N{bitbasher_expr}}
```

N: A positive integer that represents the repeat factor in the expression

The following are some examples of this construct:

```
a1 = {4{b[7:3]}}
```

The above expression is equivalent to *a1* = {*b*[7:3], *b*[7:3], *b*[7:3], *b*[7:3]}

```
a2 = {b[7:3], 2{c, d}}
```

The above expression is equivalent to *a2* = {*b*[7:3], *c*, *d*, *c*, *d*}

Constants

Binary Constant : *N*'**b***bin_const*

Octal Constant : *N*'**o***octal_const*

Decimal Constant : *N*'**d***doctal_const*

Hexadecimal Constant : *N*'**h***octal_const*

N: A positive integer that represents the **number of bits** that will be used to represent the constant

bin_const: A legal binary number string made up of **0** and **1**

octal_const: A legal octal number string made up of **0, 1, 2, 3, 4, 5, 6 and 7**

decimal_const: A legal decimal number string made up of **0, 1, 2, 3, 4, 5, 6, 7, 8 and 9**

hexadecimal_const: A legal binary number string made up of **0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e and f**

A constant can only be used to augment expressions already derived from input ports. In other words, a BitBasher block cannot be used to simply source constant like the constant block.

The following examples make use of this construct:

a1 = {4'b1100, e}

if e were 110110110 then *a1* would be 1100110110110.

a1 = {4'hb, e}

if e were 110110110 then *a1* would be 1101110110110.

a1 = {4'o10, e}

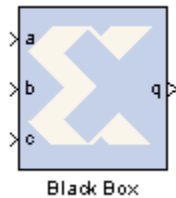
if e were 110110110 then *a1* would be 1000110110110.

Limitations

- * Does not support masked parameterization on the bitbasher expressions.
- * An expression cannot contain only constants, that is, each expression must include at least one input port.

Black Box

This block is listed in the following Xilinx Blockset libraries: Basic Elements, Control Logic, and Index.



The System Generator Black Box block provides a way to incorporate hardware description language (HDL) models into System Generator. The block is used to specify both the simulation behavior in Simulink and the implementation files to be used during code generation with System Generator. A black box's ports produce and consume the same sorts of signals as other System Generator blocks. When a black box is translated into hardware, the associated HDL entity is automatically incorporated and wired to other blocks in the resulting design.

The black box can be used to incorporate either VHDL or Verilog into a Simulink model. Black box HDL can be co-simulated with Simulink using the System Generator interface to either ISE simulator or the ModelSim simulation software from Model Technology, Inc. You can find more information on this topic in the documentation for the ModelSim block and in the User's Guide section on HDL co-simulation.

In addition to incorporating HDL into a System Generator model, the black box can be used to define the implementation associated with an external simulation model (e.g., a hardware co-simulation block).

System Generator also includes several black box and HDL co-simulation examples that demonstrate the capabilities and use of the black box.

Requirements on HDL for Black Boxes

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

- The entity name must not collide with any entity name that is reserved by System Generator (e.g., `xfir`, `xlregister`).
- Bi-directional ports are not allowed on the top-level black box entity.
- Top level ports should be ordered most significant bit down to least significant bit, as in `std_logic_vector(7 downto 0)`, and not `std_logic_vector(0 to 7)`.
- For Verilog black boxes, the module and port names must be lower case and follow standard VHDL naming conventions.
- Clock and clock enable ports must be named according to the conventions described below.
- Any port that is a clock or clock enable must be of type `std_logic`. (For Verilog black boxes, such ports must be non-vector inputs, e.g., `input clk`.)
- Clock and clock enable ports on a black box are not treated like other ports. When a black box is translated into hardware, System Generator drives the clock and clock enable ports with signals whose rates can be specified according to the block's configuration and the sample rates that drive it in Simulink.

To understand how clocks work for black boxes, it helps to understand how System Generator handles clocks in general. To produce multiple rates in hardware, System Generator uses a single clock along with multiple clock enables, one enable for each rate. The enables activate different portions of hardware at the appropriate times. Each clock enable rate is related to a corresponding sample period in Simulink. Every System Generator block that requires a clock has at least one clock and clock enable port in its HDL counterpart. Blocks having multiple rates have additional clock and clock enable ports.

Clocks for black boxes work like those for other System Generator blocks. The black box HDL must have a separate clock and clock enable port for each associated sample rate in Simulink. Clock and clock enable ports in black box HDL should be expressed as follows:

- Clock and clock enables must appear as pairs (i.e., for every clock, there is a corresponding clock enable, and vice-versa). Although a black box may have more than one clock port, a single clock source is used to drive each clock port. Only the clock enable rates differ.
- Each clock name (respectively, clock enable name) must contain the substring `clk` (resp., `ce`).
- 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`.

Clock and clock enable ports are not visible on the black box block icon. A workaround is required to make the top-level HDL clock enable port visible in System Generator; the workaround is to add a separate enable port to the top-level HDL and AND this signal with the actual clock enable signal.

The Black Box Configuration Wizard

The Configuration Wizard is a tool that makes it easy to associate a Verilog or VHDL component to a black box. The wizard is invoked whenever a black box is added to a model. To use the wizard, copy the file that defines the HDL component for a black box into the directory that contains the model. When a new black box is added to a model, the Configuration Wizard opens automatically. An example screenshot is shown in the figure below.

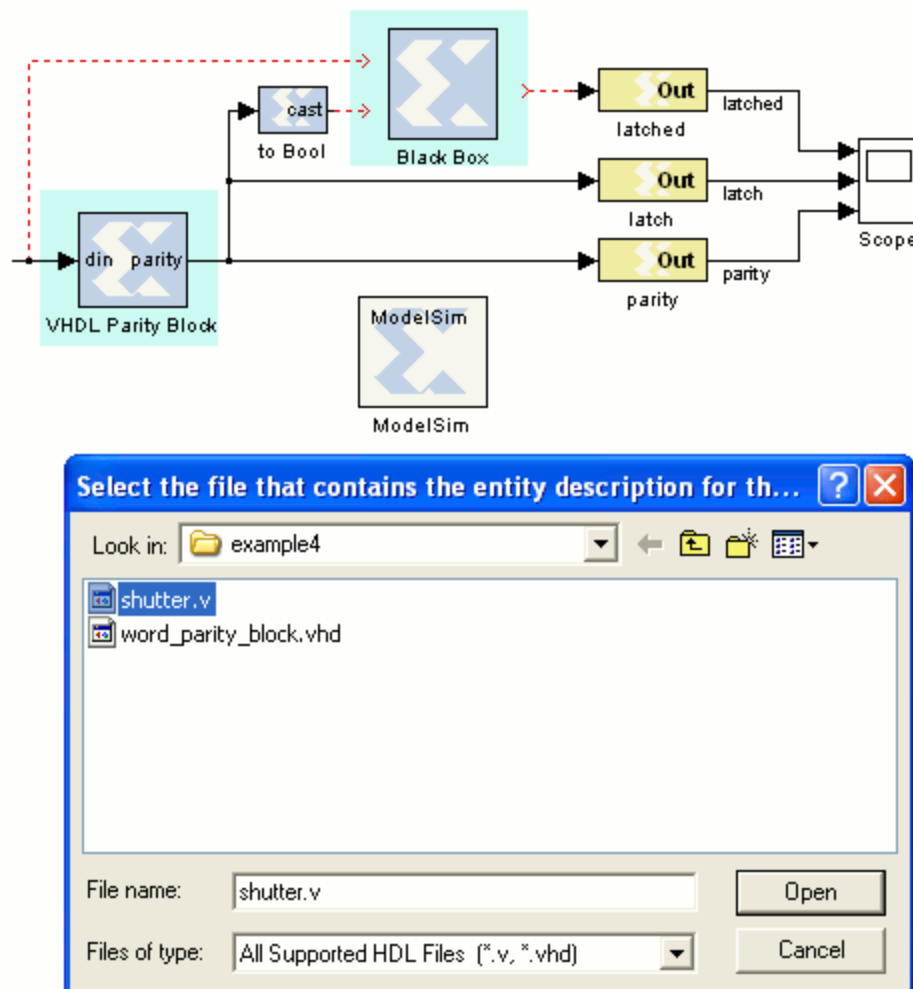


Figure 1: Black Box Configuration Wizard

From this wizard choose the HDL file that should be associated to the black box, then press the *Open* button. The wizard generates a configuration m-function (described below) for the black box, and associates the function with the block. The configuration m-function produced by the wizard can usually be used without change, but occasionally the function must be tailored by hand. Whether the configuration m-function needs to be modified depends on how complex the HDL is.

The Black Box Configuration M-Function

A black box must describe its interface (e.g., ports and generics) and its implementation to System Generator. It does this through the definition of a MATLAB m-function (or p-function) called the the block's *configuration m-function*. The name of this function must be specified in the block parameter dialog box under the Block Configuration M-Function parameter.

Details on writing and modifying configuration m-functions can be found both below and in more detailed discussions in earlier chapters of the *System Generator User Guide*.

The configuration m-function does the following:

- It specifies the top-level entity name of the HDL component that should be associated with the black box;

- It selects the language (i.e., VHDL or Verilog);
- It describes ports, including type, direction, bit width, binary point position, name, and sample rate. Ports can be static or dynamic. Static ports do not change; dynamic ports change in response to changes in the design. For example, a dynamic port might vary its width and type to suit the signal that drives it.
- It defines any necessary port type and data rate checking;
- It defines any generics required by the black box HDL;
- It specifies the black box HDL and other files (e.g., EDIF) that are associated with the block;
- It defines the clocks and clock enables for the block (see the following section on clock conventions).
- It declares whether the HDL has any combinational feed-through paths.

System Generator provides an object-based interface for configuring black boxes consisting of two types of objects: `SysgenBlockDescriptors`, used to define entity characteristics, and `SysgenPortDescriptors`, used to define port characteristics. This interface is used to provide System Generator information in the configuration m-function for black box about the block's interface, simulation model, and implementation.

If the HDL for a black box has at least one combinational path (i.e., a direct feed-through from an input to an output port), the block must be tagged as combinational in its configuration m-function using the `tagAsCombinational` method. A black box can be a mixture (i.e., some paths can be combinational while others are not). *It is essential that a block containing a combinational path be tagged as such. Doing so allows System Generator to identify such blocks to the Simulink simulator. If this is not done, simulation results will be incorrect.*

The configuration m-function for a black box is invoked several times when a model is compiled. The function typically includes code that depends on the block's input ports. For example, sometimes it is necessary to set the data type and/or rate of an output port based on the attributes on an input port. It is sometimes also necessary to check the type and rate on an input port. At certain times when the function is invoked, Simulink may not yet know enough for such code to be executed.

To avoid the problems that arise when information is not yet known (in particular, exceptions), `BlockDescriptor` members `inputTypesKnown` and `inputRatesKnown` can be used. These are used to determine if Simulink is able, at the moment, to provide information about the input port types and rates respectively. The following code illustrates this point.

```
if (this_block.inputTypesKnown)
% set dynamic output port types
    % set generics that depend on input port types
    % check types of input ports
end
```

If all input rates are known, this code sets types for dynamic output ports, sets generics that depend on input port types, and verifies input port types are appropriate. Avoid the mistake of including code in these conditional blocks (e.g., a variable definition) that is needed by code outside of the conditional block.

Note that the code shown above uses an object named `this_block`. Every black box configuration m-function automatically makes `this_block` available through an input argument. In MATLAB, `this_block` is the object that represents the black box, and is used inside the configuration m-function to test and configure the black box. Every `this_block` object is an instance of the `SysgenBlockDescriptor` MATLAB class. The methods that can be applied to `this_block` are specified in Appendix A. A good way to generate example configuration m-function is to run the Configuration Wizard (described below) on simple VHDL entities.

The black box examples are an excellent way to become familiar with black box configuration options.

Sample Periods

The output ports, clocks, and clock enables on a black box must be assigned sample periods in the configuration m-function. If these periods are dynamic, or the black box needs to check rates, then the function must obtain the input port sample periods. Sample periods in the black box are expressed as integer multiples of the system rate as specified by the *Simulink System Period* field on the master System Generator block. For example, if the *Simulink System Period* is 1/8, and a black box input port runs at the system rate (i.e., at 1/8), then the configuration m-function sees 1 reported as the port's rate. Likewise, if the *Simulink System Period* is specified as pi, and an output port should run four times as fast as the system rate (i.e., at 4*pi), then the configuration m-function should set the rate on the output port to 4. The appropriate rate for constant ports is Inf.

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

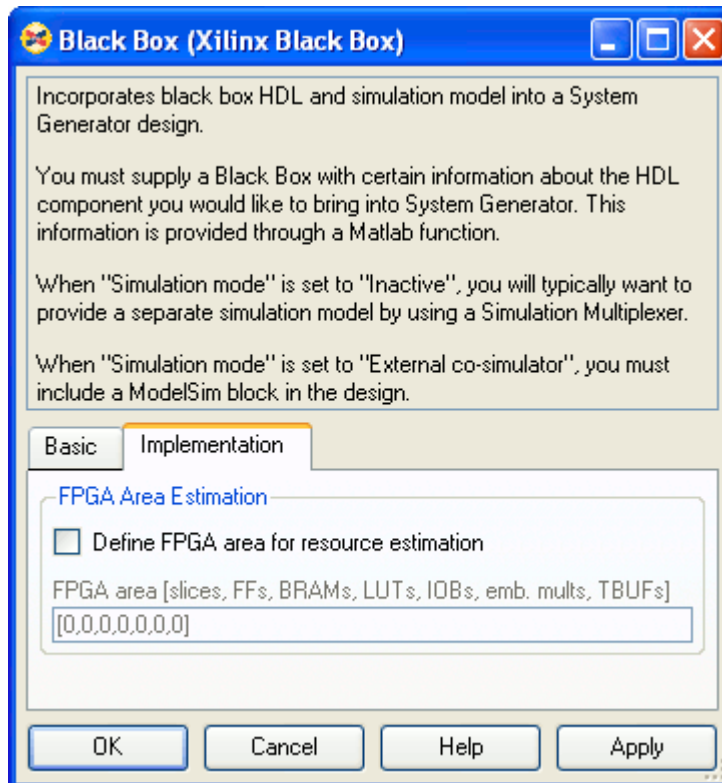
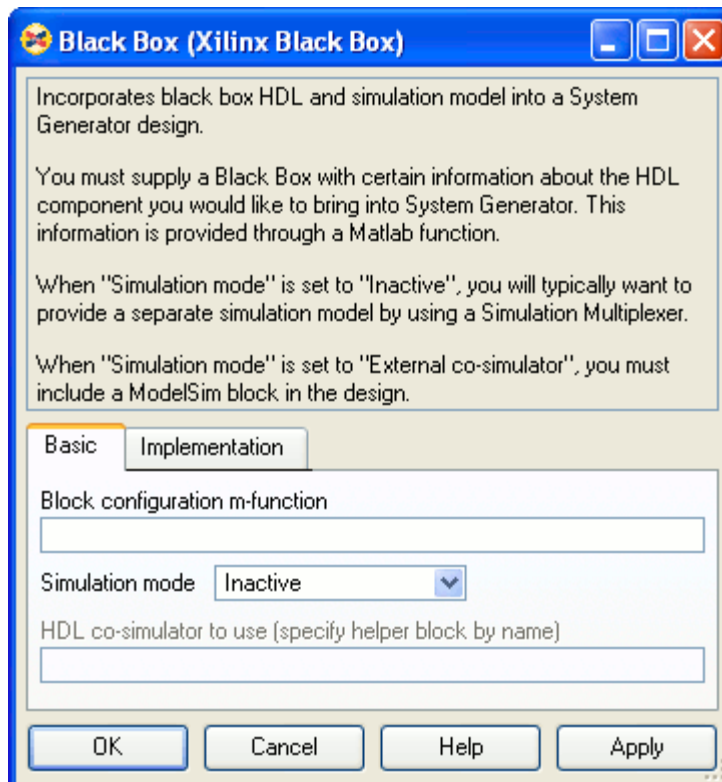


Figure 2: Black Box parameters dialog box

Parameters specific to the black box are:

- **Block Configuration M-Function:** Specifies the name of the configuration m-function that is associated to the black box. Ordinarily the file containing the function is stored in the directory containing the model, but it can be stored anywhere on the MATLAB path. Note that MATLAB limits all function names (including those for configuration m-functions) to 63 characters. Do not include the file extension (".m" or ".p") in the edit box.
- **Simulation Mode:** Tells the mode (Inactive, ISE Simulator or External co-simulator) to use for simulation. When the mode is Inactive, the black box ignores all input data and writes zeroes to its output ports. Usually for this mode the black box should be coupled, using a Configurable Subsystem.

System Generator uses Configurable Subsystems to allow two paths to be identified – one for producing simulation results, and the other for producing hardware. This approach gives the best simulation speed, but requires that a simulation model be constructed. When the mode is *ISE Simulator or External co-simulator*, simulation results for the black box are produced using co-simulation on the HDL associated with the black box. When the mode is *External co-simulator*, it is necessary to add a ModelSim HDL co-simulation block to the design, and to specify the name of the ModelSim block in the field labeled *HDL Co-Simulator To Use*. An example is shown below.

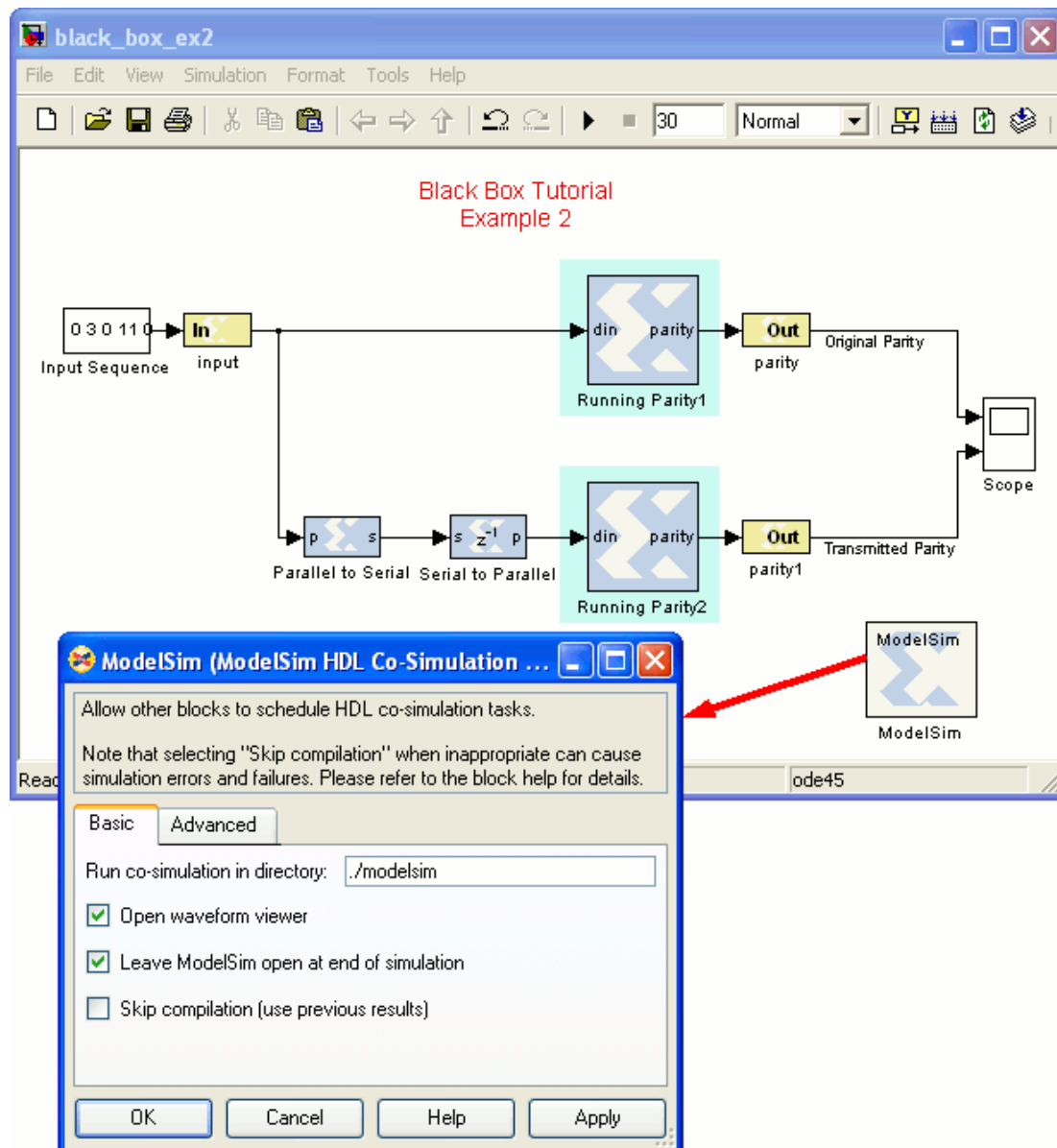


Figure 3: Black Box Specifying Block Named Modelsim as Its Co-simulator

System Generator supports the ModelSim simulator from Mentor Graphics®, Inc. for HDL co-simulation. For co-simulation of Verilog black boxes, a mixed mode license is required. This is necessary because the portion of the design that System Generator writes is VHDL.

Usually the co-simulator block for a black box is stored in the same subsystem that contains the black box, but it is possible to store the block elsewhere. The path to a co-simulation block can be absolute, or can be relative to the subsystem containing the black box (e.g., `../ModelSim`). When simulating, each co-simulator block uses one license. To avoid running out of licenses, several black boxes can share the same co-simulation block. System Generator automatically generates and uses the additional VHDL needed to allow multiple blocks to be combined into a single ModelSim simulation.

Incremental Netlisting

The System Generator black box is well suited to netlisting incrementally, i.e., to translating the design to HDL in pieces, then reusing the HDL to translate the design as a whole. This can be useful when

the design is large. See the section of the User's Guide devoted to incremental netlisting for details.

Data Type Translation for HDL Co-Simulation

During co-simulation, ports in System Generator drive ports in the HDL simulator, and vice-versa. Types of signals in the tools are not identical, and must be translated. The rules used for translation are the following.

- A signal in System Generator can be Boolean, unsigned or signed fixed point. Fixed-point signals can have indeterminate values, but Boolean signals cannot. If the signal's value is indeterminate in System Generator, then all bits of the HDL signal become 'X', otherwise the bits become 0's and 1's that represent the signal's value.
- To bring HDL signals back into System Generator, standard logic types are translated into Booleans and fixed-point values as instructed by the black box configuration m-function. When there is a width mismatch, an error is reported. Indeterminate signals of all varieties (weak high, weak low, etc.) are translated to System Generator indeterminates. Any signal that is partially indeterminate in HDL simulation (e.g., a bit vector in which only the topmost bit is indeterminate) becomes entirely indeterminate in System Generator.
- HDL to System Generator translations can be tailored by adding a custom simulation-only top-level wrapper to the VHDL component. Such a wrapper might, for example, translate every weak low signal to 0 or every indeterminate signal to 0 or 1 before it is returned to System Generator.

An Example

The following is an example VHDL entity that can be associated to a System Generator black box. (This entity is taken from black box example 2).

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity word_parity_block is
    generic (width : integer := 8);
    port (din : in std_logic_vector(width-1 downto 0);
          parity : out std_logic);
end word_parity_block;

architecture behavior of word_parity_block is
begin
    WORD_PARITY_Process : process (din)
        variable partial_parity : std_logic := '0';
    begin
        partial_parity := '0';
        XOR_BIT_LOOP: for N in din'range loop
            partial_parity := partial_parity xor din(N);
        end loop; -- N

        parity <= partial_parity after 1 ns ;
    end process WORD_PARITY_Process;
end behavior;
```

The following is an example configuration m-function. It makes the VHDL shown above available inside a System Generator black box.

```
function word_parity_block_config(this_block)

    this_block.setTopLevelLanguage('VHDL');
    this_block.setEntityName('word_parity_block');
    this_block.tagAsCombinational;
    this_block.addSimulinkInport('din');
```

```
this_block.addSimulinkOutport('parity');
parity = this_block.port('parity');
parity.setWidth(1);
parity.useHDLVector(false);

% -----
if (this_block.inputTypesKnown)
    this_block.addGeneric('width',
        this_block.port('din').width);
end % if(inputTypesKnown)
% -----

% -----
if (this_block.inputRatesKnown)
    din = this_block.port('din');
    parity.setRate(din.rate);
end % if(inputRatesKnown)
% -----

this_block.addFile('word_parity_block.vhd');

return;
```

See Also

SysgenBlockDescriptor, SysgenPortDescriptor – descriptions of the objects and methods used in configuration m-functions.

ChipScope

This block is listed in the following Xilinx Blockset libraries: Tools and Index.



The Xilinx ChipScope block enables run-time debugging and verification of signals within an FPGA. Deep capture memory and multiple trigger options are provided. Data is captured based on user defined trigger conditions and stored in internal block memory.

The Xilinx ChipScope block can be accessed at run-time using the ChipScope Pro Analyzer software. The Analyzer is used to configure the FPGA, setup trigger conditions and view the captured data at run-time. All control and data transfer is done via the JTAG port, eliminating the need to drive data off-chip using I/O pins. Data can be exported from the Analyzer and read back into the Matlab workspace.

Hardware and Software Requirements

The ChipScope Pro software (refer to Software Prerequisites section to obtain information on software to be installed to use this block), a download cable and a FPGA board with a JTAG connector are required. More information about purchasing ChipScope Pro can be found at http://www.xilinx.com/ise/optional_prod/cspro.htm.

The ChipScope Pro Analyzer supports the following download cables for communication between the PC and devices in the JTAG Boundary Scan chain:

- Parallel Cable III
- Parallel Cable IV
- MultiLINX (JTAG mode only)
- Agilent E5904B Option 500, FPGA Trace Port Analyzer (Agilent E5904B TPA).

Supported device families include Virtex-4, Virtex-II Pro, Virtex-II, Virtex-E, Virtex, Spartan-3E, Spartan-3, Spartan-IIE and Spartan-II.

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

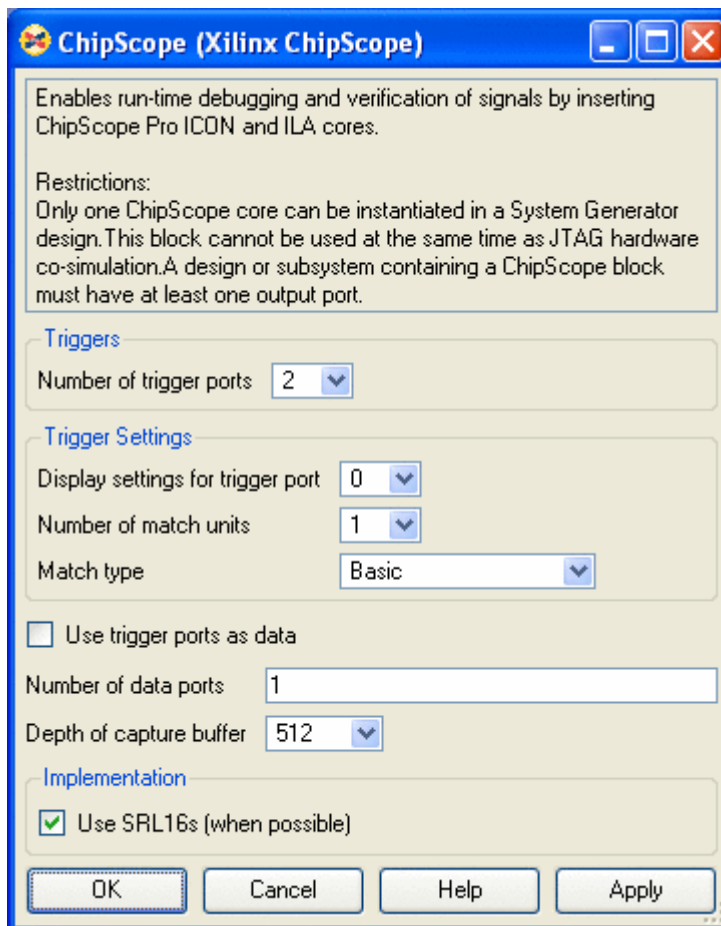


Figure 1: ChipScope block parameters dialog box

Parameters specific to the block are:

- **Number of Trigger Ports:** Multiple trigger ports allow a larger range of events to be detected and can reduce the amount of data that is stored. Up to 16 Trigger Ports can be selected. Trigger Port-numbering starts from 0 and they are named Trig0, Trig1, ... TrigN-1 by default. The trigger port can be renamed by specifying a name on the signal that is connected to the port.
- **Display settings for trigger Port:** For each trigger port, the number of match units and the match type need to be set. The pulldown menu displays settings for a particular trigger port. For N ports, the display options for trigger port 0 to N-1 can be shown.
- **Number of match units:** Using multiple match units per trigger port increases the flexibility of event detection. One to four match units can be used in conjunction to test for a trigger event. The trigger value is set at run-time in the ChipScope Pro Analyzer.
- **Match Type for Trigger Port:** This option can be set to one of the following six types:
 1. **Basic:** performs = or <> comparisons
 2. **Basic With Edges:** in addition to the basic operations high/low, low/high transitions can also be detected
 3. **Extended:** performs =, <>, >, <, <=, >= comparisons
 4. **Extended With Edges:** in addition to the extended operations, high/low, low/high transitions can also be detected.
 5. **Range:** performs =, <>, >, >=, <, <=, in range, not in range comparisons

6. **Range With Edges:** in addition to the range operations, high/low, low/high transitions can also be detected.

Hardware Note: The Basic match type is the most area efficient and can compare 8-bits per FPGA slice. The Basic With Edges match unit comparers 4-bits per slice, Extended and Extended With Edges operates on 2-bits per slice and, Range and Range With Edges can compare 1-bit per slice.

- **Number of data ports:** Up to 256 bits of data can be captured per sample. This implies that the number of Data Ports multiplied by the number of bits-per-port should be less than or equal to 256. System Generator propagates the data width automatically; therefore only the number of data ports need to be specified.
- **Use trigger ports as data:** When this option is selected, the data and trigger ports are identical and are named trig0/data0, trig1/data1, ... trigN-1/dataN-1, where N is the number of trigger ports. This mode is very common in most logic analyzers, since it enables the data that is used to trigger the ChipScope block to be captured and collected. This mode conserves hardware resources by limiting the amount of data that is captured.

When this option is not selected the data ports are completely independent of the trigger ports. The trigger ports are named trig0, trig1, ... trigN-1, and the data ports are named data0, data1, ... dataN-1. The ports can be renamed by specifying a name on the signal that is connected to the port.

- **Depth of capture buffer:** The depth of the capture buffer is a power of 2, up to 16384 samples for Virtex-II, Virtex-II Pro, and Spartan-3 device families, and 4096 for Virtex, Virtex-E, Spartan-II and Spartan-IIE device families.

ChipScope Project File

System Generator creates a project file for ChipScope Pro in order to group data signals connected to the block into buses. A bus is created for each data port so that it can be viewed as an analog waveform by using the Bus Plot feature in the ChipScope Pro Analyzer. Each data bus is scaled based on the binary point used in Simulink model. If the signals connected to the ChipScope block are named, these names will be used in the ChipScope project file to name the busses.

A project can be loaded into the ChipScope Analyzer by selecting the File -> Import -> Select New File menu option and by choosing the ChipScope project file associated with the design. The project is saved as <block name>.cdc (<block name> is derived from the name of the chipscope block in the design being compiled) in the model's target directory .

Importing Data Into Matlab Workspace From ChipScope

To export data from the ChipScope Pro Analyzer, first select the buses in the Bus Plot window that are to be exported. Then select the File -> Export option, select the ASCII format and choose 'Bus Plot Buses' to export. Press the Export button and save the file with a .prn extension. Within Matlab, change the current working directory to the location where the .prn file has been saved and type:

```
xlLoadChipScopeData('<your file name>.prn');
```

This loads the data from the .prn file into the MATLAB workspace. The names of the new workspace variables are the ports names of the ChipScope block. If the signals connected to the ChipScope block are named, these names are used to create the MATLAB workspace variables. If signal names are not

specified the port names will depend on the **Use Trigger Ports as Data option**. If this option is selected, the default the workspace variables will be named trig0_data0, trig1_data1, ... trigN-1_dataN-1. If the option is not selected, by default the names of the variables are data0 and data1, ... dataN.

Known Issues

- Refer to Software Prerequisites section to obtain information on software to be installed to use this block
- Only one ChipScope core can be instantiated in a System Generator design. Simulink Goto and From blocks can be used to easily route signals to the ChipScope block.
- The ChipScope block cannot be used at the same time as JTAG Hardware Co-Simulation since both use the JTAG port.
- A design or subsystem containing a ChipScope block must have at least one output port. If an output port does not exist, the ChipScope block will be optimized away during VHDL synthesis.

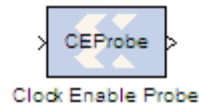
More Information

Please refer to the following web page for further details on the ChipScope Pro software:
http://www.xilinx.com/ise/optional_prod/cspro.htm.

For a step-by-step tutorial on how to use this block, please refer to the ChipScope tutorial in the System Generator Users Guide.

Clock Enable Probe

This block is listed in the following Xilinx Blockset libraries: *Basic Elements and Index*.



The Xilinx Clock Enable (CE) Probe provides a mechanism for extracting derived clock enable signals from Xilinx signals in System Generator models.

The probe accepts any Xilinx signal type as input, and produces a **BOOL** output signal. The **BOOL** output can be used at any point in the design where **BOOL**s are acceptable. The probe output is a cyclical pulse that mimics the behavior of an ideal clock enable signal used in the hardware implementation of a multi-rate circuit. The frequency of the pulse is derived from the input signal's sample period. The enable pulse is asserted at the end of the input signal's sample period for the duration of one Simulink system period. For signals with a sample period equal to the Simulink system period, the block's output is always one.

Shown below is an example model with an attached analysis scope that demonstrates the usage and behavior of the Clock Enable Probe. The Simulink system sample period for the model is specified in the System Generator block as 1.0 seconds. In addition to the Simulink system period, the model has three other sample periods defined by the Down Sample blocks. Clock Enable Probes are placed after each Down Sample block and extract the derived clock enable signal. The probe outputs are run to output gateways and then to the scope for analysis. Also included in the model is CLK probe that produces a Double representation of the hardware system clock. The scope output shows the output from the four Clock Enable probes in addition to the CLK probe output.

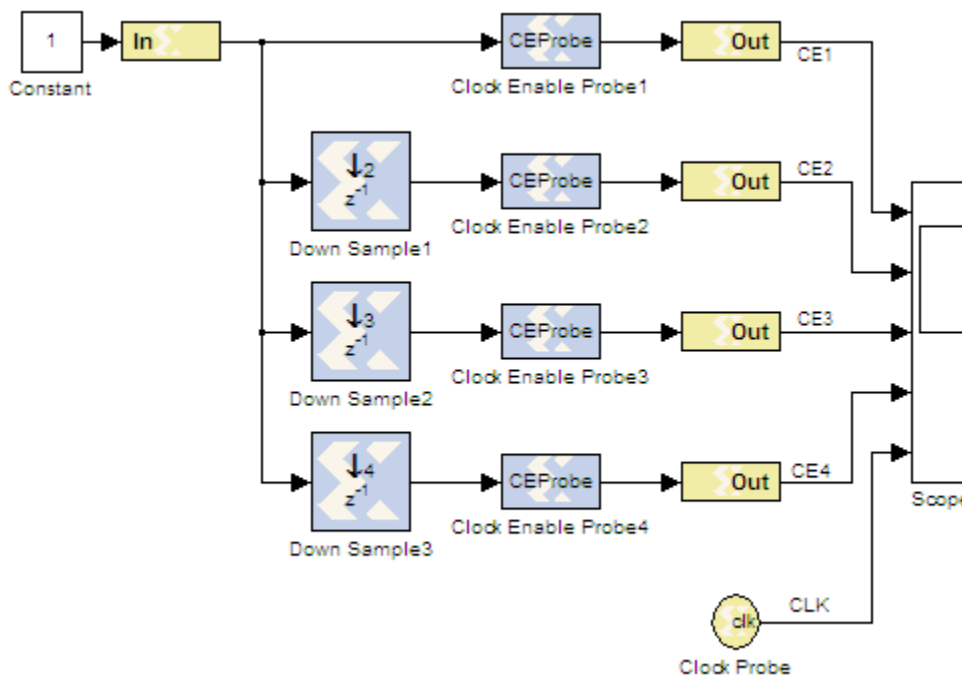


Figure 1: Example model demonstrating CE probe use

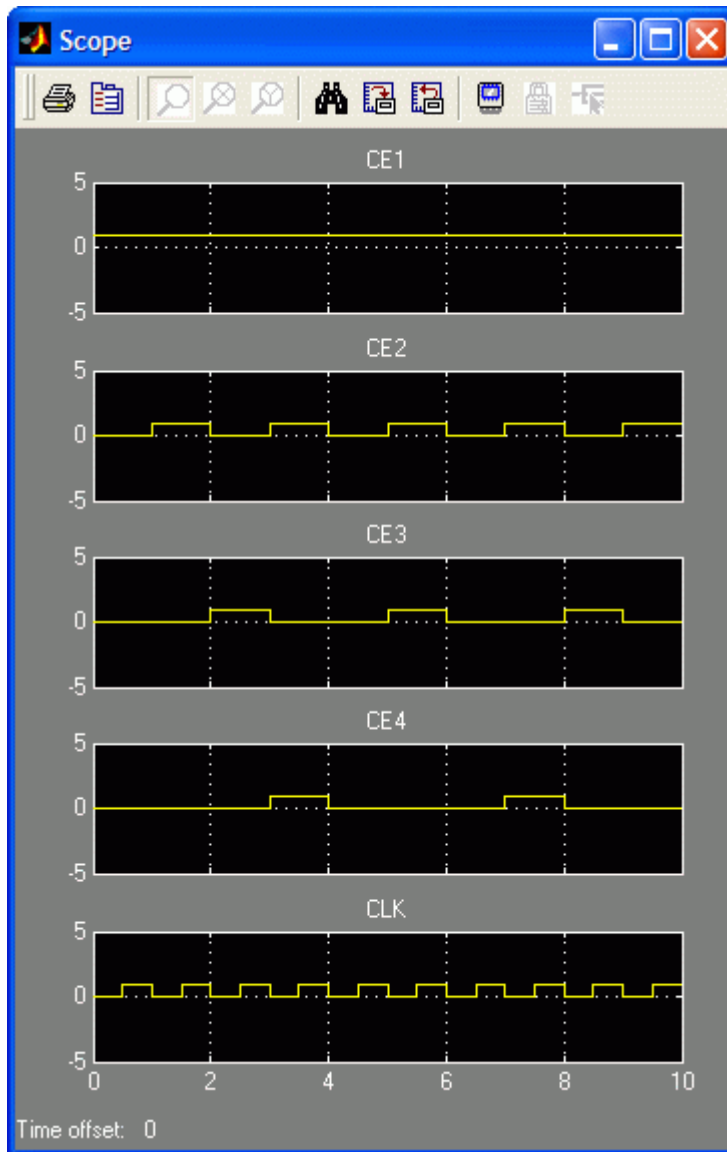


Figure 2: Scope output demonstrating CE probe output

The Clock Enable block has no parameters.

Clock Probe

This block is listed in the following Xilinx Blockset libraries: Tools and Index.



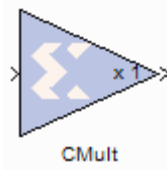
The Xilinx Clock Probe generates a double-precision representation of a clock signal with a period equal to the Simulink system period.

The output clock signal has a 50/50 duty cycle with the clock asserted at the start of the Simulink sample period. The Clock Probe's double output is useful only for analysis, and cannot be translated into hardware.

There are no parameters for this block.

CMult

This block is listed in the following Xilinx Blockset libraries: *Math and Index*.



The Xilinx CMult block implements a *gain* operator, with output equal to the product of its input by a constant value. This value can be a MATLAB expression that evaluates to a constant.

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

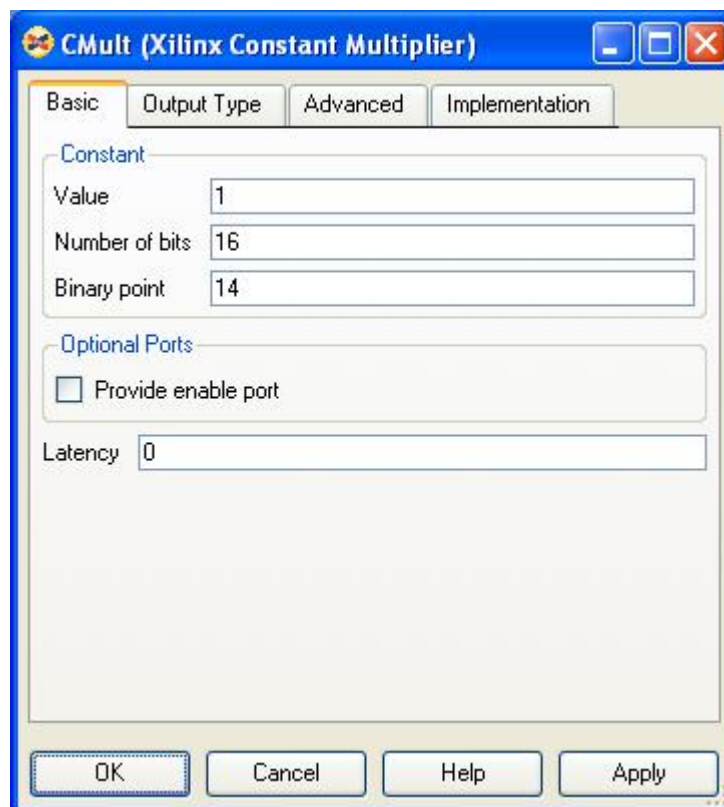


Figure 1: CMult block basic parameters tab

Parameters specific to the CMult block basic tab are:

- **Constant value:** may be a constant or an expression. If the constant cannot be expressed exactly in the specified fixed point type, its value is rounded and saturated as needed. A positive value is implemented as an unsigned number, a negative value as signed.
- **Constant Number of bits:** specifies the bit location of the binary point of the constant, where bit zero is the least significant bit.
- **Constant Binary point:** position of the binary point.

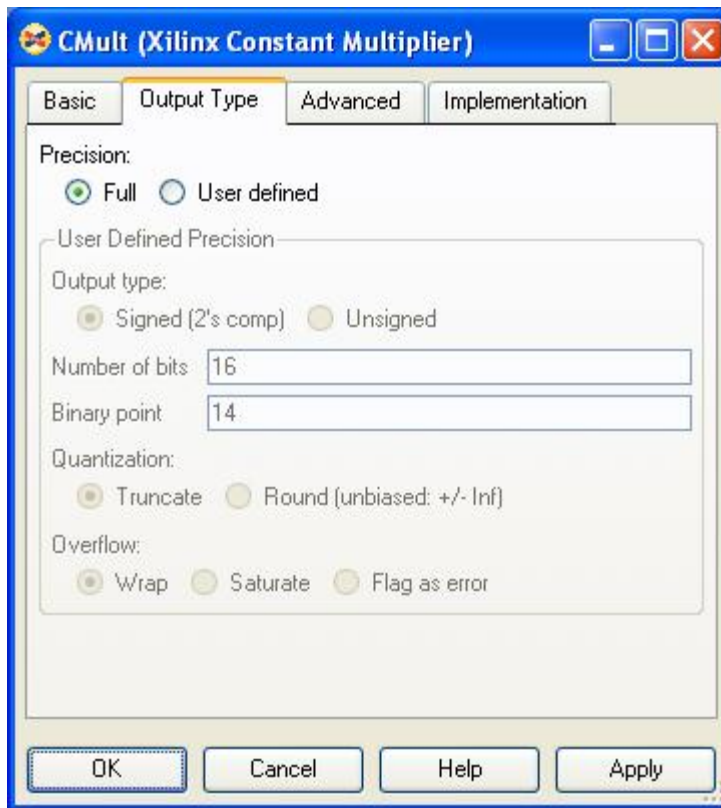


Figure 2: CMult block output type parameters tab

The parameters on the output type tab define the precision of the output of the CMult block. These parameters are described in the Common Parameters section at the beginning of this chapter.

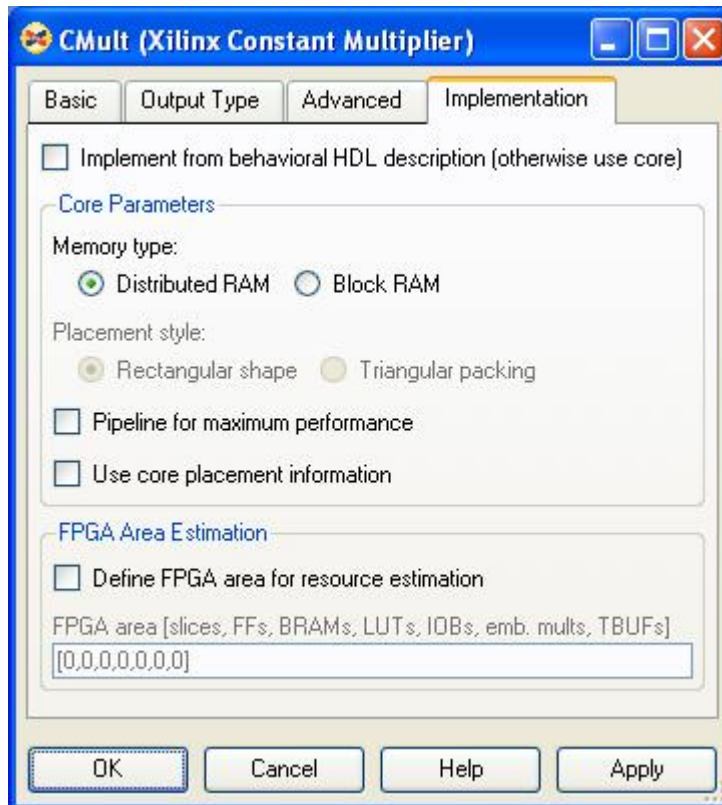


Figure 2: CMult block implementation parameters tab

Parameters specific to the CMult block implementation tab are:

- **Implement from behavioral HDL description (otherwise use core):** when selected use behavioral HDL, otherwise use the Xilinx LogiCORE Multiplier Generator.
- **Memory Type:** specifies whether to use distributed RAM or block RAM.
- **Placement style:** when using core placement information, two shape options are available. Rectangular creates a roughly rectangular design. Triangular creates a more densely packed design.
- **Pipeline for maximum performance:** when checked, directs System Generator to pipeline the LogiCORE implementation to the fullest extent possible.

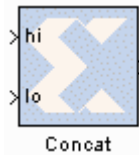
Other parameters used by this block are explained in the Common Parameters section at the beginning of this chapter.

Xilinx LogiCORE

When requested this block uses the Xilinx LogiCORE Multiplier Generator.

Concat

This block is listed in the following Xilinx Blockset libraries: *Basic Elements, Data Types, and Index.*



The Xilinx Concat block performs a concatenation of n bit vectors represented by unsigned integer numbers, i.e. n unsigned numbers with binary points at position zero.

The Xilinx Reinterpret block provides capabilities that can extend the functionality of the Concat block.

Block Interface

The block has n input ports, where n is some value between 2 and 1024, inclusively, and one output port. The first and last input ports are labeled `hi` and `lo`, respectively. Input ports between these two ports are not labeled. The input to the `hi` port will occupy the most significant bits of the output and the input to the `lo` port will occupy the least significant bits of the output.

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

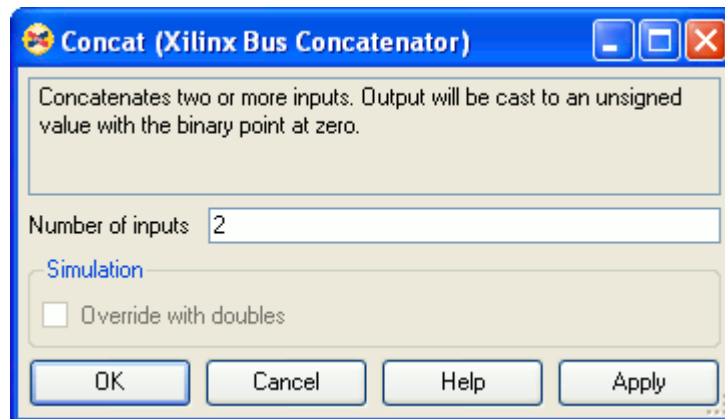


Figure 1: Concat block parameters dialog box

Parameters specific to the Concat block are:

- **Number of Inputs:** specifies number of inputs, between 2 and 1024, inclusively, to concatenate together.

Other parameters used by this block are explained in the Common Parameters section at the beginning of this chapter of the Reference Guide.

The Concat block does not use a Xilinx LogiCORE.

Configurable Subsystem Manager

This block is listed in the following Xilinx Blockset libraries: Tools and Index.



The Xilinx Configurable Subsystem Manager extends Simulink's configurable subsystem capabilities to allow a subsystem configurations to be selected for hardware generation as well as for simulation. It can be used to create Simulink library blocks (subsystems) that have special capabilities when used with the System Generator software. Another section in the User Guide discusses in detail how configurable subsystems are used by System

Generator.

System Generator will automatically insert Configurable Subsystem Manager blocks into library subsystems that it generates through its "Import as Configurable Subsystem" capability. It is also possible to hand-build library subsystems that take advantage of the Simulink and System Generator configurable subsystem capabilities.

Recall that a configurable subsystem consists of a collection of subblocks, exactly one of which "represents" the subsystem at any given time. (The so-called "block choice" for the subsystem specifies which subblock should be the representative.) The representative is the subblock used to produce results for the subsystem when simulating.

System Generator designs can be simulated, but can also be translated into hardware, and it is often useful to identify a second block to be used as a configurable subsystem's "hardware representative". The hardware representative is the subblock used to translating the configurable subsystem into hardware. For example, suppose a configurable subsystem consists of two subblocks, namely a black box whose HDL implements a filter, and a subsystem that implements the same filter using ordinary System Generator blocks. Then it is natural to use the subsystem as the representative and the black box as the hardware representative, i.e., to use the subsystem in simulations, and the black box HDL to generate hardware.

The configurable subsystem manager specifies which subblock in a System Generator configurable subsystem should be the hardware representative. To specify the hardware representative, do the following: 1) Place a manager inside one of the subblocks, and 2) Use the manager's **When generating, use** parameter to select the hardware representative.

Notes:

1. It is only possible to use a configurable subsystem manager by placing it inside a subblock of a configurable subsystem. This means that at least one subblock must be a subsystem.
2. When several subblocks contain managers, the managers automatically synchronize so they agree on the choice of hardware representative.

Block Parameters Dialog Box

The dialog box for a configurable subsystem manager is shown below.

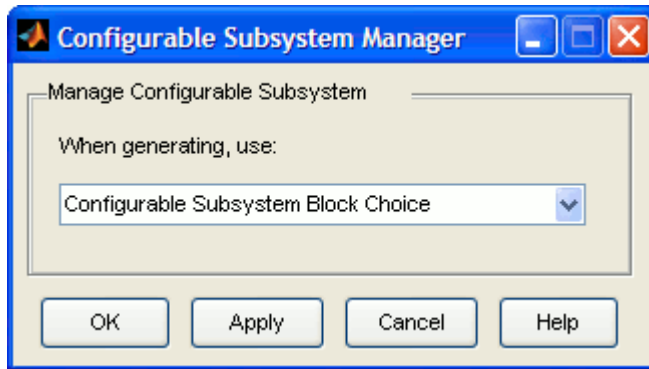
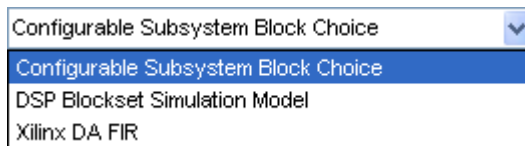


Figure 1: Configurable Subsystem Manager Block Parameters Dialog box

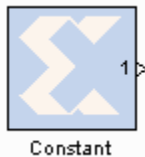
This block has one parameter, labeled **When generating, use**. The parameter specifies which subblock to use as the hardware representative. An example list of choices is shown below.



When **Configurable Subsystem Block Choice** is selected, the subblock specified as the representative for the configurable subsystem is also used for generating hardware. Otherwise, the subblock selected from the list is used as the hardware representative.

Constant

This block is listed in the following Xilinx Blockset libraries: *Basic Elements, Control Logic, Math, and Index.*



The Xilinx Constant block generates a constant that can be a fixed-point value, a Boolean value, or a DSP48 instruction. This block is similar to the Simulink constant block, but can be used to directly drive the inputs on Xilinx blocks.

DSP48 Instruction Mode

The constant block, when set to create a DSP48 instruction, is useful for generating DSP48 control sequences. The Figure 1 below shows an example. The example implements a 35x35-bit multiplier using a sequence of four instructions in a DSP48 block. The constant blocks supply the desired instructions to a multiplexer that selects each instruction in the desired sequence.

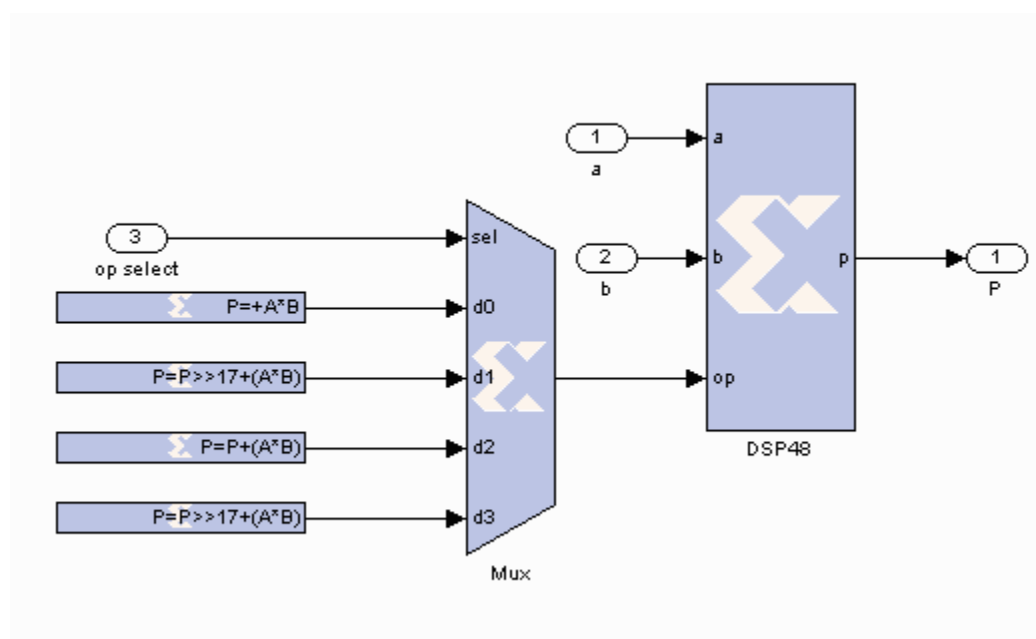


Figure 1: Using Constant blocks as control sequences for DSP48

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

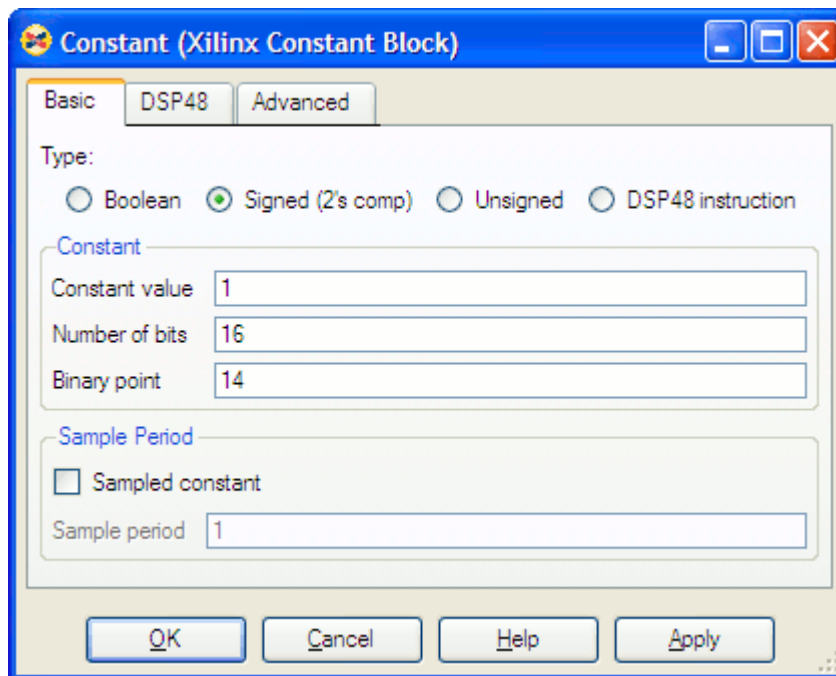


Figure 2: Constant block parameters dialog box

Parameters specific to the block are:

- **Constant Value:** specifies the value of the constant. When changed, the new value appears on the block icon.
- **Type:** specifies the type of constant. Can be one of Boolean, signed fixed-point, unsigned fixed-point, or DSP48 instruction.
- **Sampled Constant:** allows a sample period to be associated with the constant output and inherited by blocks that the constant block drives. (This is useful mainly because the blocks eventually target hardware and the Simulink sample periods are used to establish hardware clock periods.)

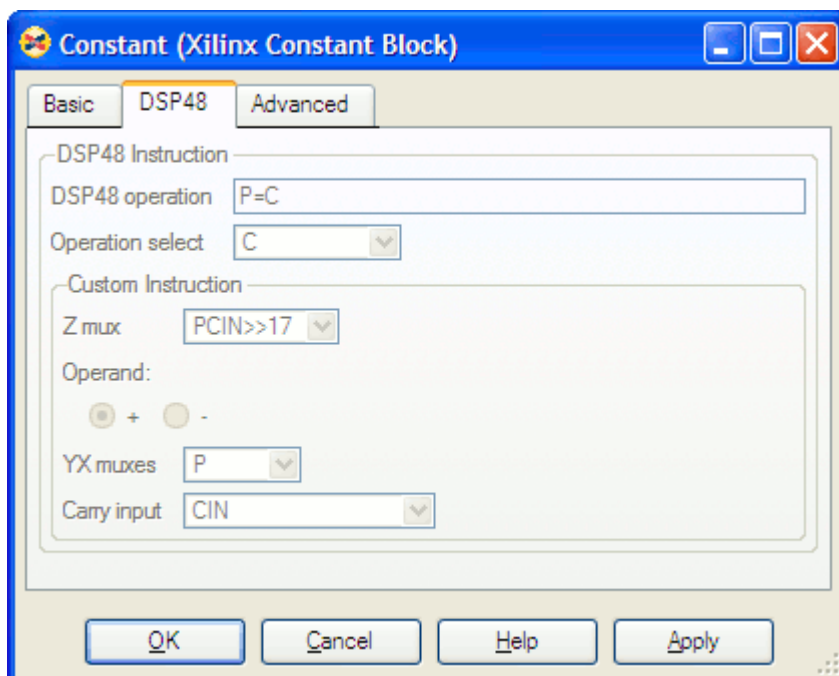


Figure 3: Constant block parameters dialog box specifying a DSP48 instruction

When DSP48 Instruction is selected for type, the DSP48 tab is activated. A detailed description of the DSP48 can be found in the DSP48 block description.

- **DSP48 operation:** displays the selected DSP48 instruction.
- **Operation select:** allows the selection of a DSP48 instruction. Selecting custom reveals mask parameters that allow the formation of an instruction in the form $z\_mux \pm (yx\_mux + carry)$.
- **Z Mux:** specifies the 'Z' source to the DSP48's adder to be one of {'0', 'C', 'PCIN', 'P', 'C', 'PCIN>>17', 'P>>17'}.
- **Operand:** specifies whether the DSP48's adder is to perform addition or subtraction.
- **YX Muxes:** specifies the 'YX' source to the DSP48's adder to be one of {'0', 'P', 'A:B', 'A*B', 'C', 'P+C', 'A:B+C'}. 'A:B' implies that A[17:0] is concatenated with B[17:0] to produce a 36-bit value to be used as an input to the DSP48 adder.
- **Carry Input:** specifies the 'carry' source to the DSP48's adder to be one of {'0', '1', 'CIN', '~SIGN(P or PCIN)', '~SIGN(A*B or A:B)', '~SIGND(A*B or A:B)'}. '~SIGN (P or PCIN)' implies that the carry source is either P or PCIN depending on the Z Mux setting. '~SIGN(A*B or A:B)' implies that the carry source is either A*B or A:B depending on the YX Mux setting. The option '~SIGND (A*B or A:B)' selects a delayed version of '~SIGN(A*B or A:B)'.

Other parameters used by this block are explained in the Common Parameters section at the beginning of this chapter.

The constant block does not use a Xilinx LogiCORE.

Appendix: DSP48 Control Instruction Format

Instruction Field Name	Location	Mnemonic	Description
YX Mux	op[3:0]	0	0
		P	DSP48 output register

		A:B	Concat inputs A and B (A is MSB)
		A*B	Multiplication of inputs A and B
		C	DSP48 input C
		P+C	DSP48 input C plus P
		A:B+C	Concat inputs A and B plus C register
Z Mux	op[6:4]	0	0
		PCIN	DSP48 cascaded input from PCOUT
		P	DSP48 output register
		C	DSP48 C input
		PCIN>>17	Cascaded input downshifted by 17
		P>>17	DSP48 output register downshifted by 17
Operand	op[7]	+	Add
		-	Subtract
Carry In	op[8]	0 or 1	Set carry in to 0 or 1
		CIN	Select cin as source
		'~SIGN(P or PCIN)	Symmetric round P or PCIN
		'~SIGN(A:B or A*B)	Symmetric round A:B or A*B
		'~SIGND(A:B or A*B)	Delayed symmetric round of A:B or A*B

Convert

This block is listed in the following Xilinx Blockset libraries: Basic Elements, Data Types, Math, and Index.



The Xilinx Convert block converts each input sample to a number of a desired arithmetic type. For example, a number can be converted to a signed (two's complement) or unsigned value.

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

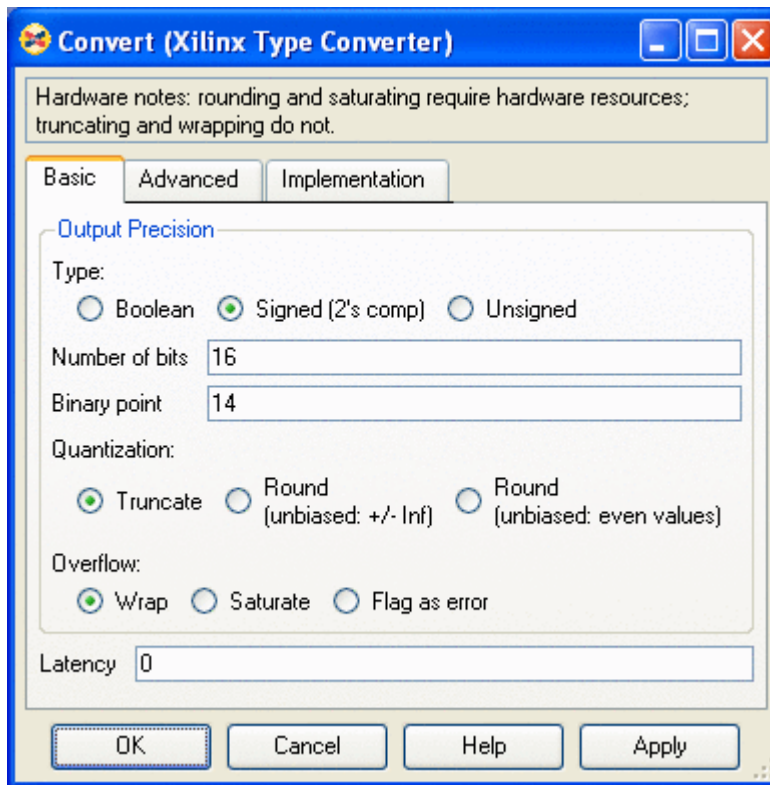


Figure 1: Convert block Basic Tab

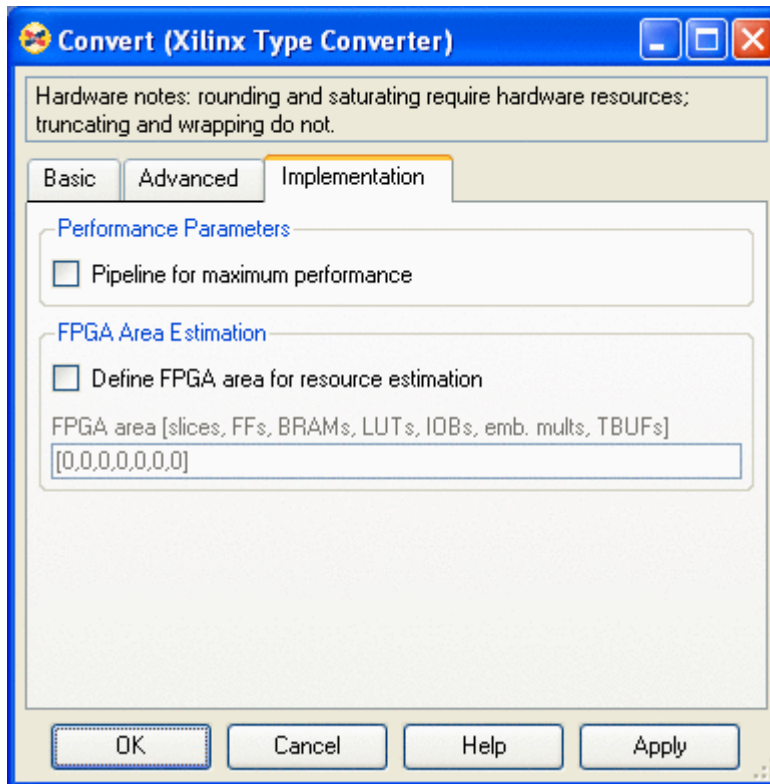


Figure 2: Convert block Implementation Tab

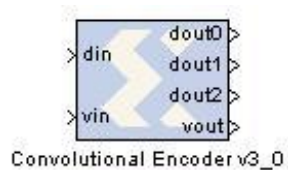
Parameters specific to the Convert block are:

- Pipeline for maximum performance:** directs the block to use the latency value to pipeline to the fullest extent possible. Latency is distributed in the following priority based on the selected pipeline option.
 - Latency (pipeline=0): Delay pipeline at the output.
 - Latency (pipeline=1): Output register, register before saturation, register before quantization and extra latency as a delay pipeline at the output.

Other parameters used by this block are explained in the Common Parameters section at the beginning of this chapter. The **Type** parameter in this block uses the same description as the **Arithmetic Type** description.

Convolutional Encoder v3_0

This block is listed in the following Xilinx Blockset libraries: *Communication and Index*.



The Xilinx Convolutional Encoder block implements an encoder for convolutional codes. Ordinarily used in tandem with a Viterbi decoder, this block performs forward error correction (FEC) in digital communication systems.

Values are encoded using a linear feed forward shift register which computes modulo-two sums over a sliding window of input data, as shown in the figure below. The length of the shift register is specified by the constraint length. The convolution codes specify which bits in the data window contribute to the modulo-two sum. Resetting the block will set the shift register to zero. The encoder rate is the ratio of input to output bit length; thus, for example a rate 1/2 encoder outputs two bits for each input bit. Similarly, a rate 1/3 encoder outputs three bits for each input bit.

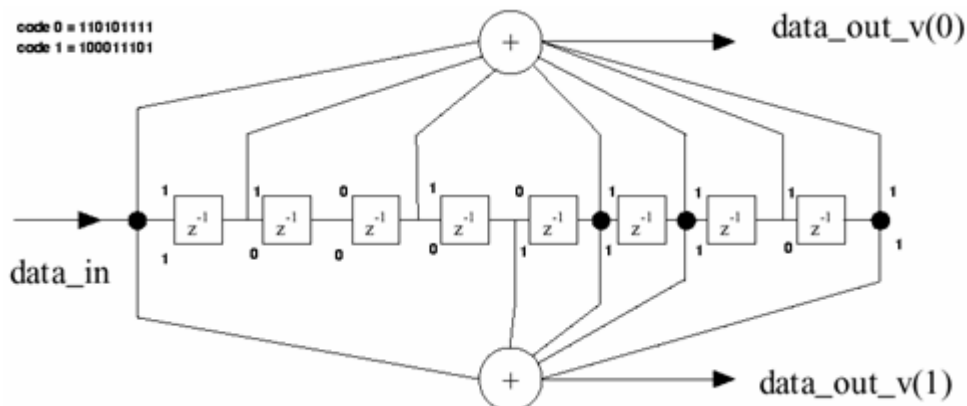


Figure 1: Constraint length 9 convolutional encoder

Block Interface

The block has between two to four input ports and three to eight output ports. The `din` port must have type `UFix1_0`. It accepts the values to be encoded. The `vin` port indicates that the values presented on `din` are valid. Only valid values are encoded. The ports `dout1` through `dout7` output the encoded data. The port `dout1` corresponds to the first code in the array, `dout2` to the second, and so on. The number of codes in the array sets the output rate of the encoder and consequently the number of data output ports. The output port `vout` indicates the validity of output values.

Block Parameters Dialog Box

The following figure shows the block parameters dialog box.

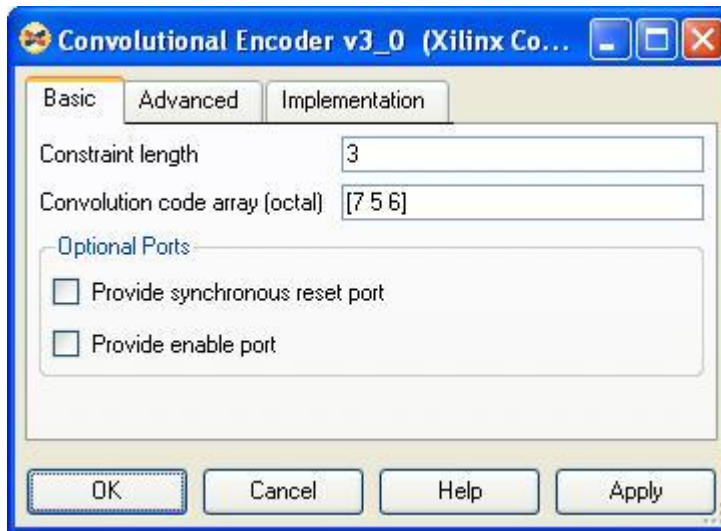


Figure 2: Convolutional encoder block parameters dialog box

Parameters specific to the block are:

- **Constraint length:** Constraint Length: Equals $n+1$, where n is the length of the constraint register in the encoder.
- **Convolution code array (octal) :** Array of octal convolution codes. Output rate is derived from the array length. Between 2 and 7 (inclusive) codes may be entered.

Other parameters used by this block are described in the Common Parameters section at the beginning of this chapter.

Xilinx LogiCORE

The block uses the Xilinx LogiCORE Convolution Encoder.

Convolutional Encoder v6_0



Convolutional Encoder v6_0

This block is listed in the following Xilinx Blockset libraries:

Communication and Index.

The Xilinx Convolutional Encoder block implements an encoder for convolutional codes. Ordinarily used in tandem with a Viterbi decoder, this block performs forward error correction (FEC) in digital communication systems.

Values are encoded using a linear feed forward shift register which computes modulo-two sums over a sliding window of input data, as shown in the figure below. The length of the shift register is specified by the constraint length. The convolution codes specify which bits in the data window contribute to the modulo-two sum. Resetting the block will set the shift register to zero. The encoder rate is the ratio of input to output bit length; thus, for example a rate 1/2 encoder outputs two bits for each input bit. Similarly, a rate 1/3 encoder outputs three bits for each input bit.

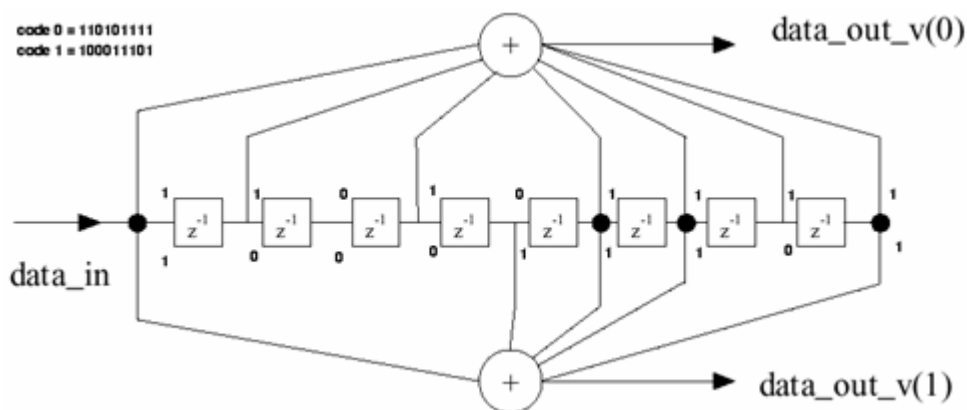


Figure 1: Constraint length 9 convolutional encoder

Block Interface

The block has between two to four input ports and three to eight output ports. The `din` port must have type `UFix1_0`. It accepts the values to be encoded. The `vin` port indicates that the values presented on `din` are valid. Only valid values are encoded. The ports `dout1` through `dout7` output the encoded data. The port `dout1` corresponds to the first code in the array, `dout2` to the second, and so on. The number of codes in the array sets the output rate of the encoder and consequently the number of data output ports. The output port `vout` indicates the validity of output values.

Block Parameters Dialog Box

The following figure shows the block parameters dialog box.

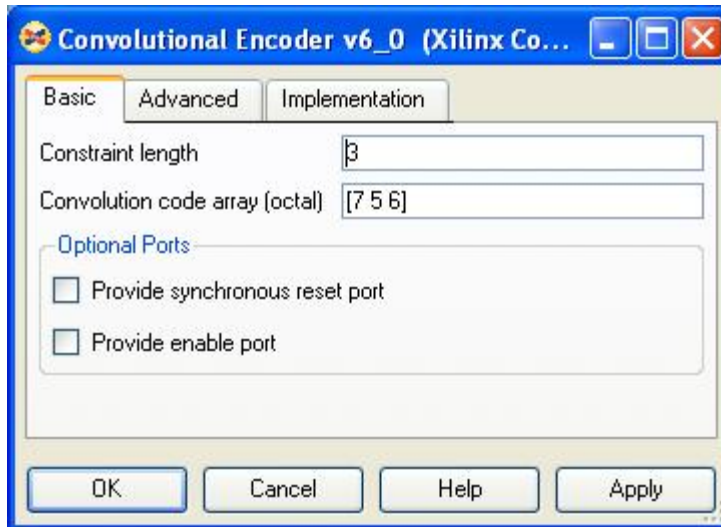


Figure 2: Convolutional encoder block parameters dialog box

Parameters specific to the block are:

- **Constraint length:** Constraint Length: Equals $n+1$, where n is the length of the constraint register in the encoder.
- **Convolution code array (octal) :** Array of octal convolution codes. Output rate is derived from the array length. Between 2 and 7 (inclusive) codes may be entered.

Other parameters used by this block are described in the Common Parameters section at the beginning of this chapter.

Xilinx LogiCORE

The block uses the Xilinx LogiCORE Convolution Encoder.

Counter

This block is listed in the following Xilinx Blockset libraries: *Basic Elements, Control Logic, Math, and Index.*



The Xilinx Counter block implements a free running or count-limited type of an up, down, or up/down counter. The counter output can be specified as a signed or unsigned fixed point number.

Free running counters are the least expensive in FPGA hardware. The free running up, down, or up/down counter can also be configured to load the output of the counter with a value on the input *din* port by selecting the **Provide Load Pin** option in the block's parameters.

The output for a free running up counter is calculated as follows:

$$out(n) = \begin{cases} InitialValue & \text{if } n = 0 \\ (out(n-1) + Step) \bmod 2^N & \text{otherwise} \end{cases}$$

The output for a free running up counter with load capability is calculated as follows:

$$out(n) = \begin{cases} InitialValue & \text{if } n = 0 \\ din(n-1) & \text{if } load(n-1) = 1 \\ (out(n-1) + Step) \bmod 2^N & \text{otherwise} \end{cases}$$

Here *N* denotes the number of bits in the counter. The free running down counter calculations replace addition with subtraction. For the free running up/down counter, the counter performs addition when input *up* port is 1 or subtraction when the input *up* port is 0.

A count-limited counter is implemented by combining a free running counter with a comparator. Count limited counters are limited to only 64 bits of output precision. Count limited types of a counter can be configured to step between the initial and ending values, provided the step value evenly divides the difference between the initial and ending values.

The output for a count limited up counter is calculated as follows:

$$out(n) = \begin{cases} InitialValue & \text{if } n = 0 \text{ or } out(n-1) = CountLimit \\ (out(n-1) + Step) \bmod 2^N & \text{otherwise} \end{cases}$$

The count-limited down counter calculation replaces addition with subtraction. For the count limited up/down counter, the counter performs addition when input *up* port is 1 or subtraction when input *up* port is 0.

The output for a free running up counter with load capability is calculated as follows:

$$out(n) = \begin{cases} StartCount & \text{if } n = 0 \text{ or } rst(n) = 1 \\ din & \text{if } rst(n) = 0 \text{ and } load(n) = 1 \\ (out(n-1) + CountByValue) \bmod 2^N & \text{otherwise} \end{cases}$$

Here N denotes the number of bits in the counter. The down counter calculations replace addition by subtraction.

Block Parameters Dialog Box

The Counter block parameters dialog box is invoked by double-clicking the block icon.

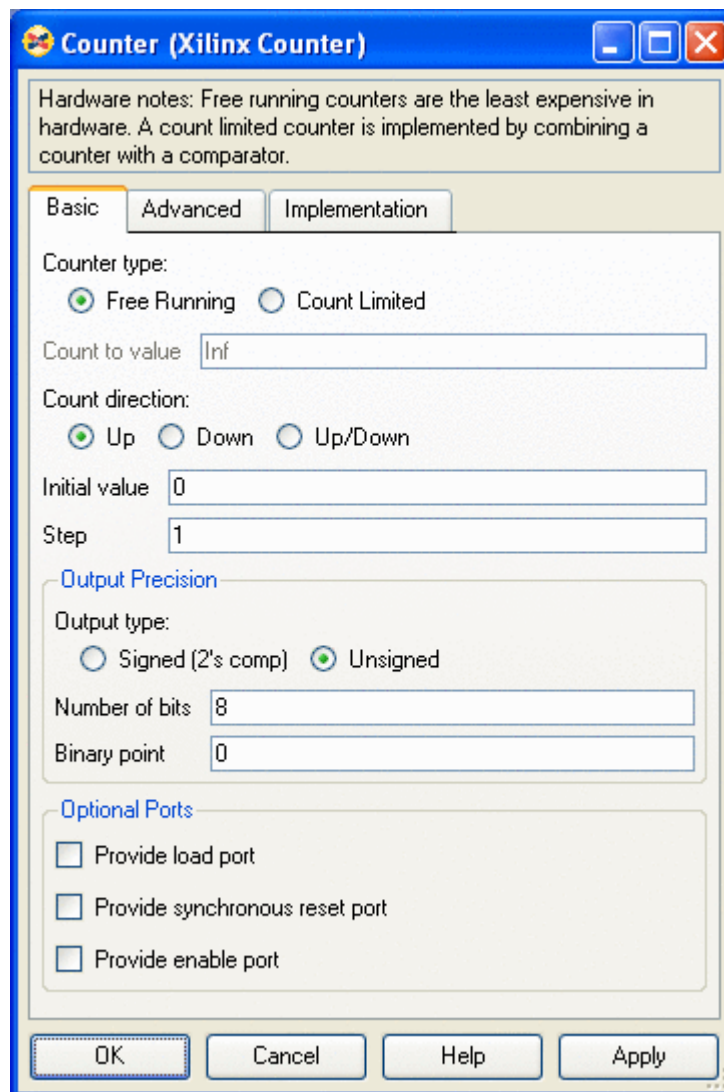


Figure 1: Counter block parameters dialog box

Parameters specific to the block are:

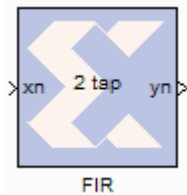
- **Counter type:** specifies the counter to be a count-limited or free running counter.
- **Number of bits:** specifies the number of bits in the block output.

- **Binary point:** specifies the location of the binary point in the block output.
- **Output type:** specifies the block output to be either Signed or Unsigned.
- **Initial value:** specifies the initial value to be the output of the counter.
- **Count to value:** specifies the ending value, the number at which the count limited counter resets. A value of `Inf` denotes the largest representable output in the specified precision. This cannot be the same as the initial value.
- **Step:** specifies the increment or decrement value.
- **Count direction:** specifies the direction of the count (up or down) or provides an optional input port `up` (when up/down is selected) for specifying the direction of the counter.
- **Provide load Port:** when checked, the block operates as a free running load counter with explicit `load` and `din` port. The load capability is available only for the free running counter.

Other parameters used by this block are described in the Common Parameters section at the beginning of this chapter.

DAFIR v9_0

This block is listed in the following Xilinx Blockset libraries: *DSP and Index*.



The Xilinx DAFIR filter block implements a distributed arithmetic finite-impulse response (FIR) digital filter, or a bank of identical FIR filters (multichannel mode). An N-tap filter is defined by N filter coefficients (or taps) $h(0)$, $h(1)$, ..., $h(n-1)$. Here each $h(i)$ is a Xilinx fixed point number.

The filter block accepts a stream of Xilinx fixed point data samples $x(0)$, $x(1)$, ..., and at time n computes the output.

Block Interface

The FIR block can be configured to have one to eight data channels as well as several optional ports.

- **vin**: marks each xn symbol as valid or invalid. For a decimating FIR filter, the state of the vin port must match for every group of samples to be decimated, i.e. the groupings of N vin samples, where N is the decimation factor, must all be either 1 or 0. The sample groupings are aligned from the start of the simulation ($t=0$).
- **vout**: marks each symbol produced on yn as valid or invalid.
- **rfd**: indicates whether the block is ready to accept new data. This port drives a Boolean signal at the same data rate as the input port, xn. This signal is asserted at the start of simulation and remains asserted (true) until a reload cycle is initiated. The rfd signal will go low on the cycle immediately following an assertion of the load signal. The rfd signal is reasserted once the block has completed the reload sequence. Available when reloading coefficients or when serial input is selected.
- **sel_in**: indicates current filter input channel number when serial input is selected.
- **sel_out**: indicates current filter output channel number when serial input is selected.

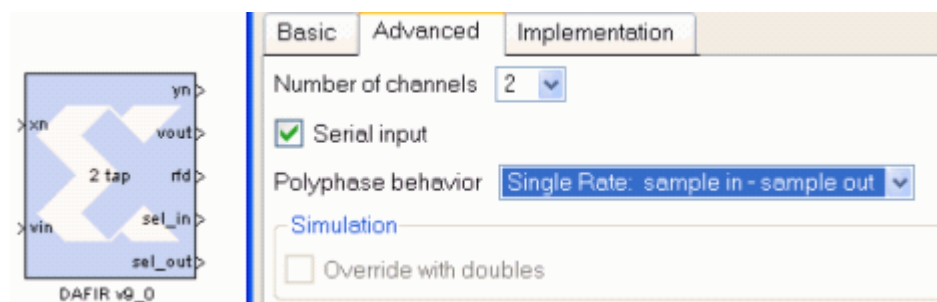


Figure 1: DA FIR icon with serial input ports exposed

Reloading Coefficients

The DA FIR filter provides optional ports for coefficient reloading. When a reload sequence is initiated, the filter stops accepting new data input samples and begins accepting new filter coefficients. Once all of the new coefficients have been written, the filter processes the coefficients and initializes the necessary internal data structures. The amount of time required for the filter to reload is a function of the filter length and type. After the reloading sequence has completed, the filter comes back online and continues to accept new input data samples. For more information about the reload sequence and filter reload time, please refer to the FIR core datasheet. An example reload sequence timing diagram is shown below:

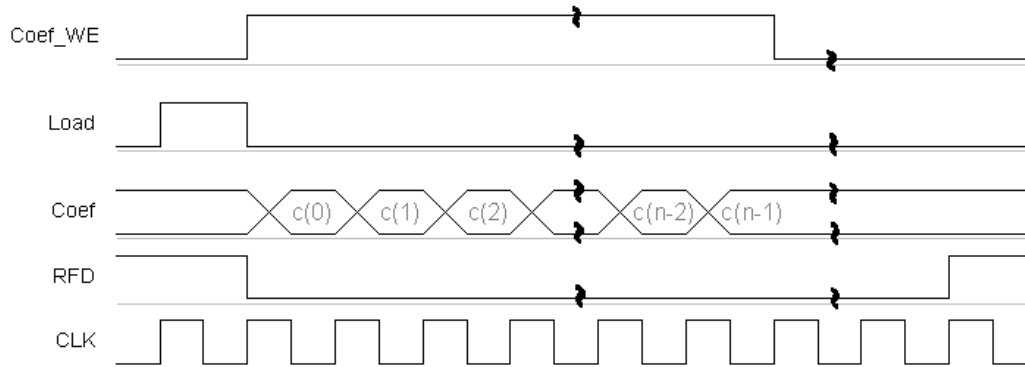


Figure 2: Example reload sequence timing diagram

Optional Ports for Reloading Coefficients

- **coef** : new filter coefficients are written to the block through this port. The number of bits and binary point position of the coef port must match the number of coefficient bits and coefficient binary point position specified in the block mask. This port must run at the same data rate as the input port, xn.
- **coef_we** : a write enable signal that controls when the coef port data is written to the block. This port can be used to stagger the time at which new coefficients are written into the filter once a reload cycle has started. The first new coefficient can be written to the filter on the cycle following the assertion of the load signal. This port should be driven by a Boolean signal with a data rate equal to the data rate of the input port, xn.
- **Load** : an assertion (true) of the load port initiates a coefficient reload sequence. The load signal should be pulsed for one cycle; subsequent assertions during a reload sequence will restart the reloading process. This port should be driven by a Boolean signal with a data rate equal to the data rate of the input port, xn.

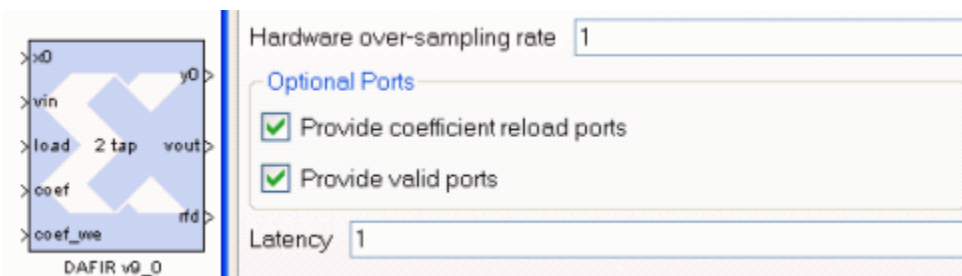


Figure 3: DA FIR icon shown with optional ports exposed

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

DAFIR v9_0 (Xilinx Finite Impulse Resp...

Hardware notes: Implemented using distributed arithmetic (DA). The hardware over sampling rate determines the degree of parallelism. A rate of one produces a fully parallel DA filter. A rate of n (resp., n+1) for an n-bit input signal produces a fully serial implementation for a non-symmetric (resp., symmetric) impulse response. Intermediate values produce implementations with intermediate levels of parallelism.

Basic Advanced Implementation

Coefficient Settings

Coefficients

Structure

Number of bits (always signed)

Binary point

Hardware over-sampling rate

Optional Ports

☐ Provide coefficient reload ports

☐ Provide valid ports

Latency

OK Cancel Help Apply

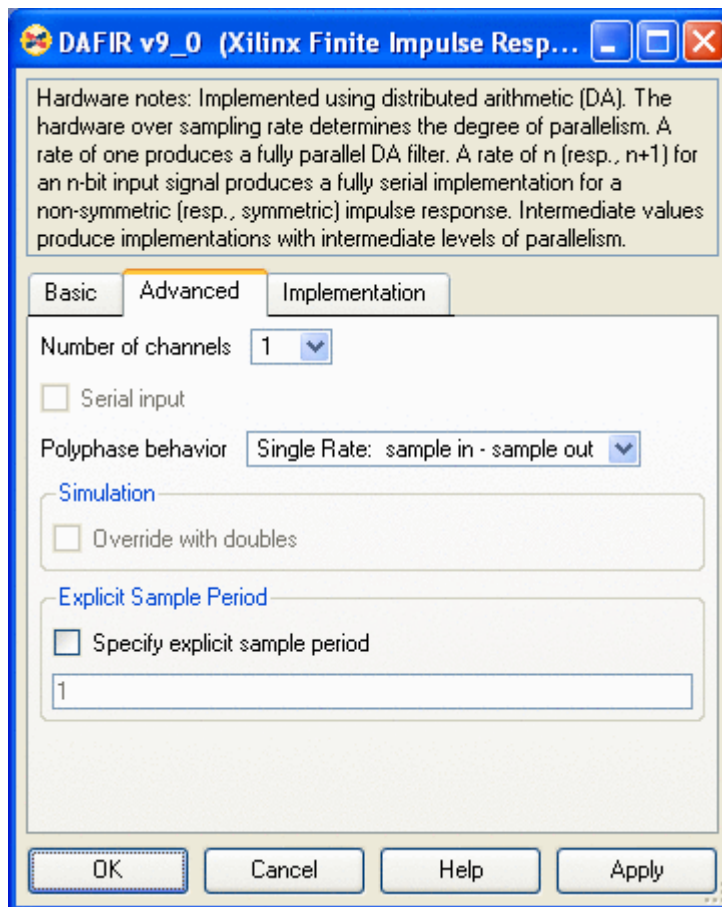


Figure 4: FIR block parameters dialog box

Parameters specific to the block are:

- **Coefficients:** vector of filter coefficients; note that these can be evaluated from a MATLAB workspace variable and may in turn be computed by MATLAB. You can also refer to examples in the System Generator Tutorial.
- **Structure:** the Xilinx Smart-IP™ FIR core's preferred implementation depends on the structure of the sequence of filter taps. You can choose one of these: inferred from coefficients, none, symmetric, negative symmetric, half band, and interpolate fir.
- **Number of bits (always signed):** Number of bits to use for representing the filter coefficients.
- **Binary Point:** Number of fractional bits to use for representing the filter coefficients.
- **Number of channels:** one to eight, inclusive. For multi-channel filters, polyphase behavior is not supported, i.e. the filter must be single rate. The core, which processes the channels serially, will be overclocked by the System Generator by a factor equaling the number of channels so as to provide the necessary throughput. To reduce control logic overhead, the block requires that the valid bits match on all inputs.
- **Serial input:** when the number of channels is greater than one, the input to the filter can be either serial (time division multiplexed) or parallel.
- **Polyphase behavior:** Decimation, Interpolation, Single rate.
- **Latency:** specify input sample period latency.
- **Hardware over-sampling Rate:** hardware clocks per sample. This affects hardware implementation only, and has no effect on simulation. In multi-channel mode, this factor will multiply the implicit oversampling factor.

- **Provide coefficient reload ports:** selecting this parameter exposes the block ports that are used in coefficient reloading: coef, load, coef_we, and rfd.
- **Provide valid ports:** enables or disables the optional valid ports vin and vout.

Other parameters used by this block are explained in the Common Parameters section at the beginning of this chapter.

Xilinx LogiCORE

The block always uses the Xilinx LogiCORE Distributed Arithmetic FIR Filter.

The Simulink model operates on a sample in/sample out basis, but the core has the capability of using serial arithmetic by overclocking. Although this adds latency, it has the benefit of reducing the hardware required for the filter. Refer to the core datasheet for more details of the filter modes and parameters.

DDS v4_0

This block is listed in the following Xilinx Blockset libraries: *DSP and Index*.



The Xilinx DDS Block is a direct digital synthesizer, also commonly called a numerically controlled oscillator (NCO). The block uses a lookup table scheme to generate sinusoids. A digital integrator (accumulator) generates a phase that is mapped by the lookup table into the output waveform.

To understand the DDS, it is necessary to know how the block is implemented in FPGA hardware. The following figure shows a high-level view of the Xilinx LogiCORE. The phase increment and phase offset can be defined as constants or can be set dynamically through optional input ports. These values are defined in terms of cycles per sample. For example, a phase increment of one tenth ($1/10$) implies that in 10 time samples, one sinusoid is completed. After the phase increment is accumulated, the phase offset is added to the result. If dithering is used, the dithering sequence (which prevents phase error from being introduced by the quantizer) is added prior to quantization. The quantized value is then used to index into the sine/cosine lookup table, mapping phase-space into time.

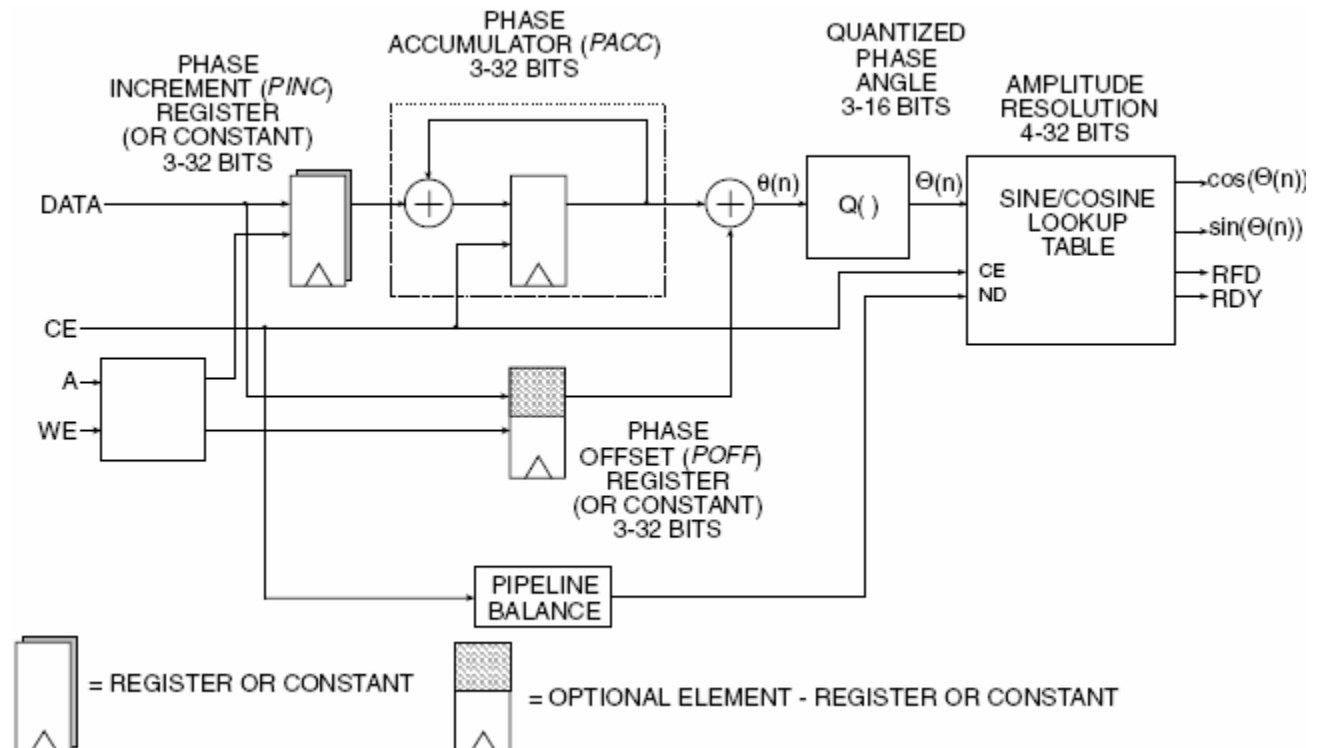


Figure 1: High-Level View of LogiCORE DDS Implementation

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

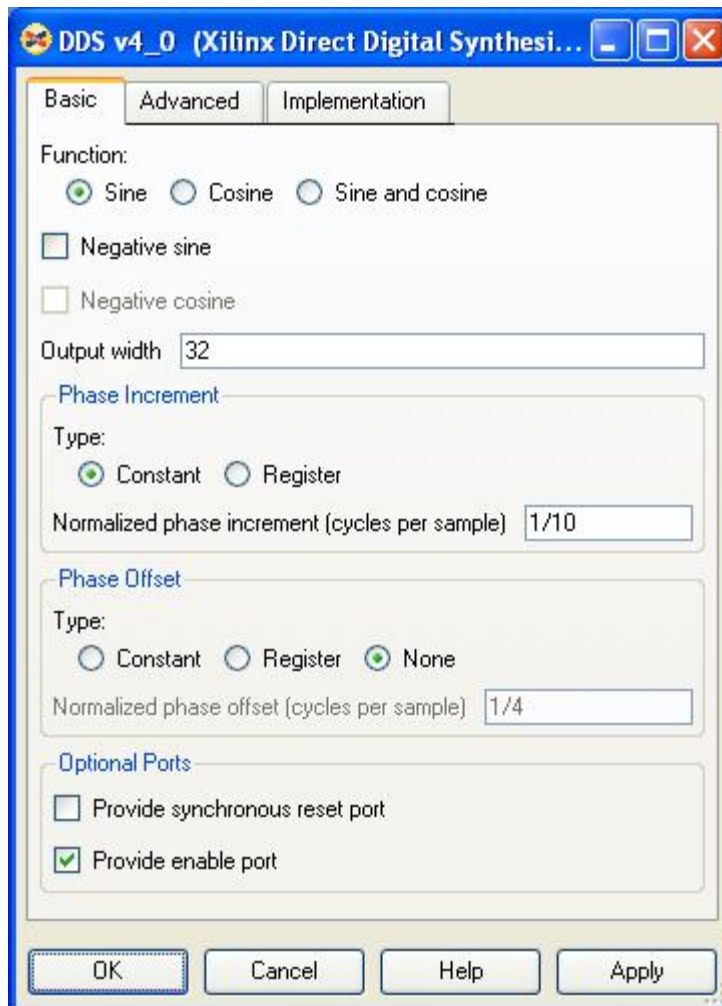


Figure 2: DDS block basic tab parameters

Parameters specific to the DDS block on the basic tab are:

- **Function:** specifies the function that the block will calculate.
- **Negative sine:** output from the sine port will be negated.
- **Negative cosine:** output from the cosine port will be negated.
- **Output width:** number of bits in the output signal; value must be between 4 and 32 inclusive.
- **Phase increment type:** specifies $\Delta\theta$ to be either constant or register. Choosing register activates optional ports on the block.
- **Normalized phase increment:** specifies value of phase increment constant, a multiple of 2π . The number of bits is determined in one of two ways. If the increment type is Register, the number of bits is set to the width of the data port. If the increment type is Constant, the number of bits is inferred from the phase increment value.
- **Phase offset type:** specifies phase offset to be Constant, Register, or None. Choosing register activates optional ports on the block.
- **Normalized phase offset:** specifies value of phase offset constant, as a multiple of 2π . The number of bits is determined in one of two ways. If the offset type is Register, the number of bits is set to the width of the data port. If the offset type is Constant, the number of bits is inferred from the phase offset value.

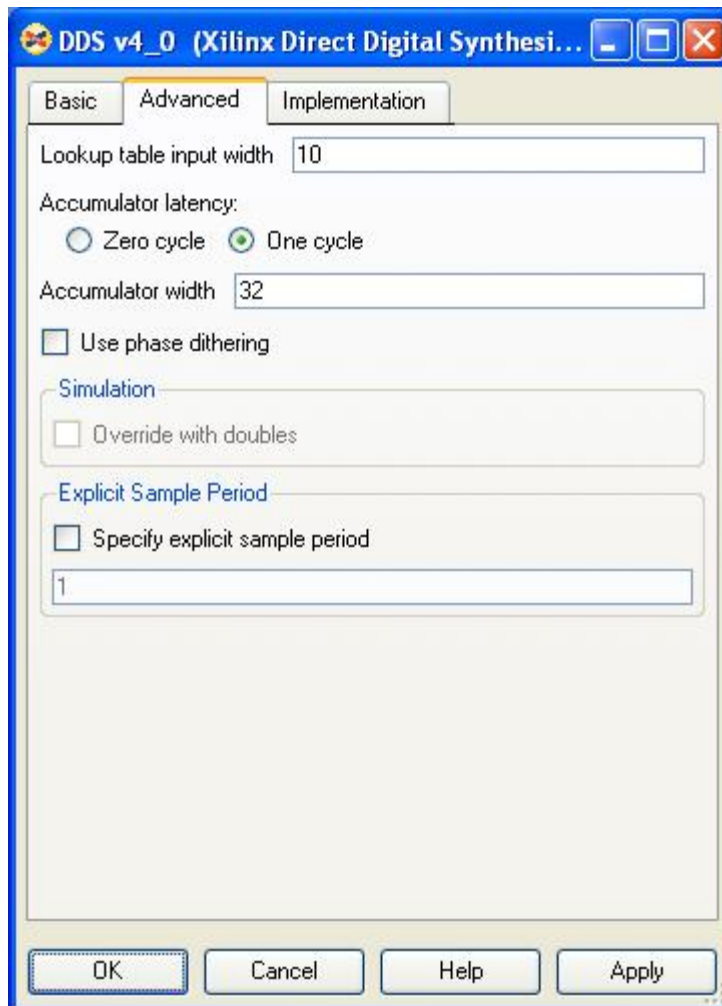


Figure 3: DDS block advanced tab parameters

Parameters specific to the DDS block on the advanced tab are:

- **Lookup table input width:** specifies the number of address bits into the sine/cosine lookup table; value must be at least 3. The width must be less than or equal to $\min(a, b)$ where a is the accumulator width, and b is 16 (if block RAM is used) or 10 (if distributed RAM is used).
- **Accumulator latency:** specifies the latency in the phase accumulator to be zero or one.
- **Accumulator width:** specifies the phase accumulator width; value must be between 3 and 32 inclusive.
- **Use phase dithering:** when checked, a dither sequence is added to the result of the phase accumulator.

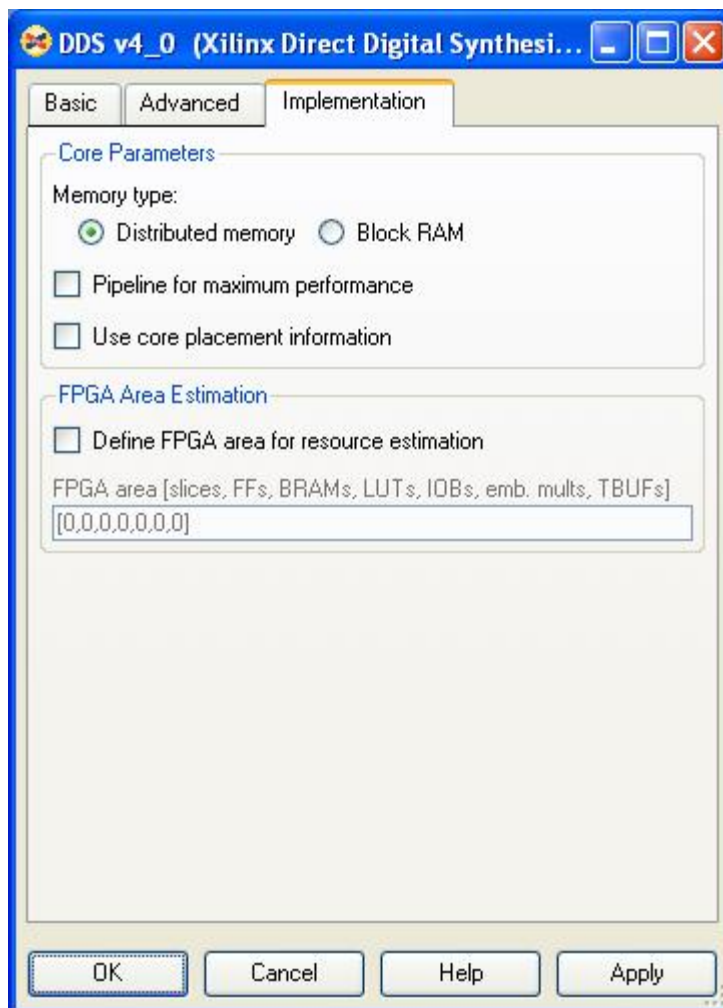


Figure 4: DDS block implementation tab parameters

Parameters specific to the DDS block on the implementation tab are:

- **Memory type:** directs the block to be implemented either with distributed or block RAM.
- **Pipeline for maximum performance:** when checked, the implementation is fully pipelined.

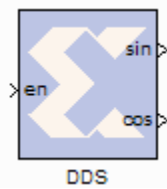
Other parameters used by this block are described in the Common Parameters section at the beginning of this chapter.

Xilinx LogiCORE

This block uses the Xilinx LogiCORE Direct Digital Synthesizer.

DDS v5_0

This block is listed in the following Xilinx Blockset libraries: *DSP and Index*.



The Xilinx DDS Block is a direct digital synthesizer, also commonly called a numerically controlled oscillator (NCO). The block uses a lookup table scheme to generate sinusoids. A digital integrator (accumulator) generates a phase that is mapped by the lookup table into the output waveform.

To understand the DDS, it is necessary to know how the block is implemented in FPGA hardware. The following figure shows a high-level view of the Xilinx LogiCore. The phase increment and phase offset can be defined as constants or can be set dynamically through optional input ports. These values are defined in terms of cycles per sample. For example, a phase increment of one tenth (1/10) implies that in 10 time samples, one sinusoid is completed. After the phase increment is accumulated, the phase offset is added to the result. If dithering is used, the dithering sequence (which prevents phase error from being introduced by the quantizer) is added prior to quantization. The quantized value is then used to index into the sine/cosine lookup table, mapping phase-space into time.

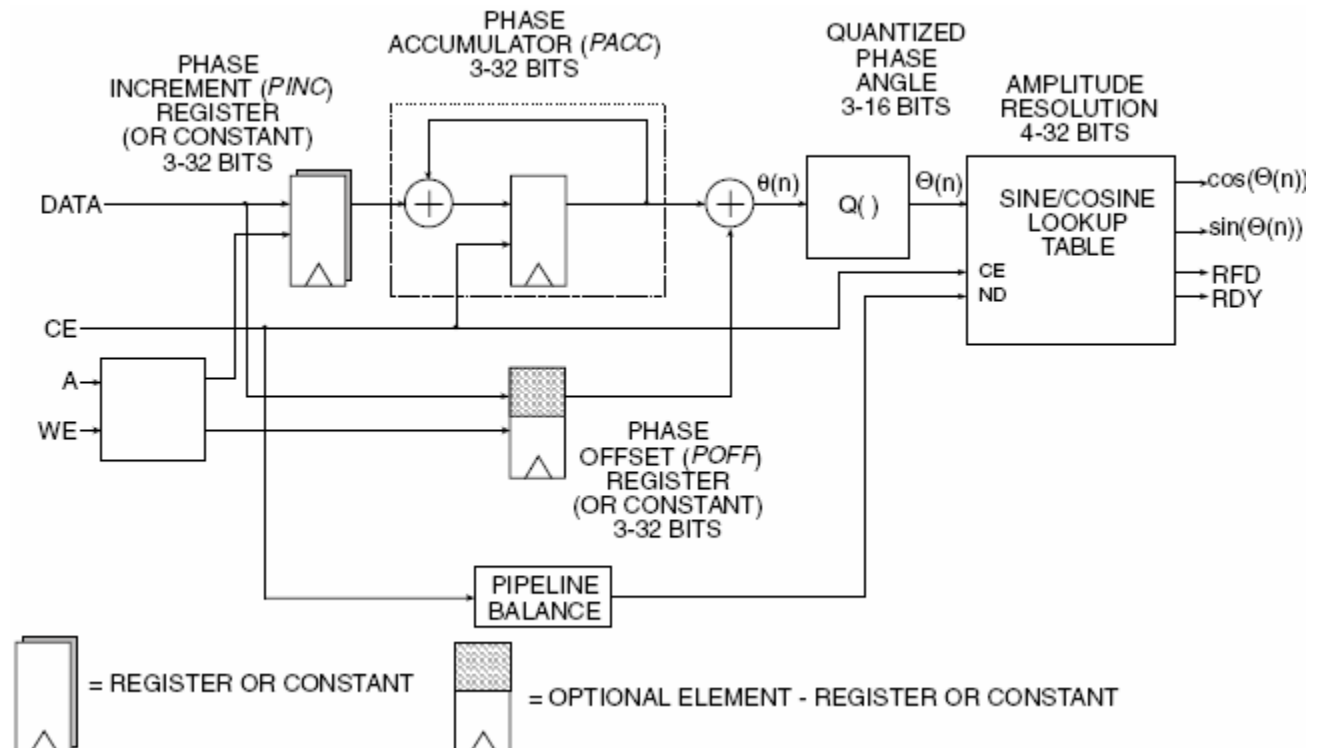
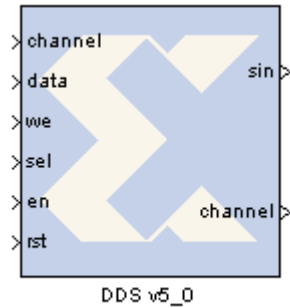


Figure 1: High-Level View of LogiCORE DDS Implementation

Block Ports



Port functions on the DDS block:

- **channel**: specifies with which channel the current input (or output) is associated. The channel input port maps to the least significant 4 bits on the DDS v5 core's A input port.
- **data**: time shared data bus. The data port is used for supplying values to the programmable offset frequency memory and/or programmable phase offset memory.
- **we**: write enable. Enables a write operation to the offset frequency memory and/or the programmable frequency memory. Which memory is written to is determined by the **sel** port value.
- **sel**: select port. Selects whether to write the data to the offset frequency memory or the phase offset memory. When **sel** is 0 and **we** is 1, data is written to the offset frequency memory. When **sel** is 1 and **we** is 1, data is written to the phase offset memory.
- **en**: user enable. When '1', the block is active.
- **rst**: reset. When '1', the internal memories of the block are reset.
- **sin**: sine output value.
- **COS**: cosine output value.

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

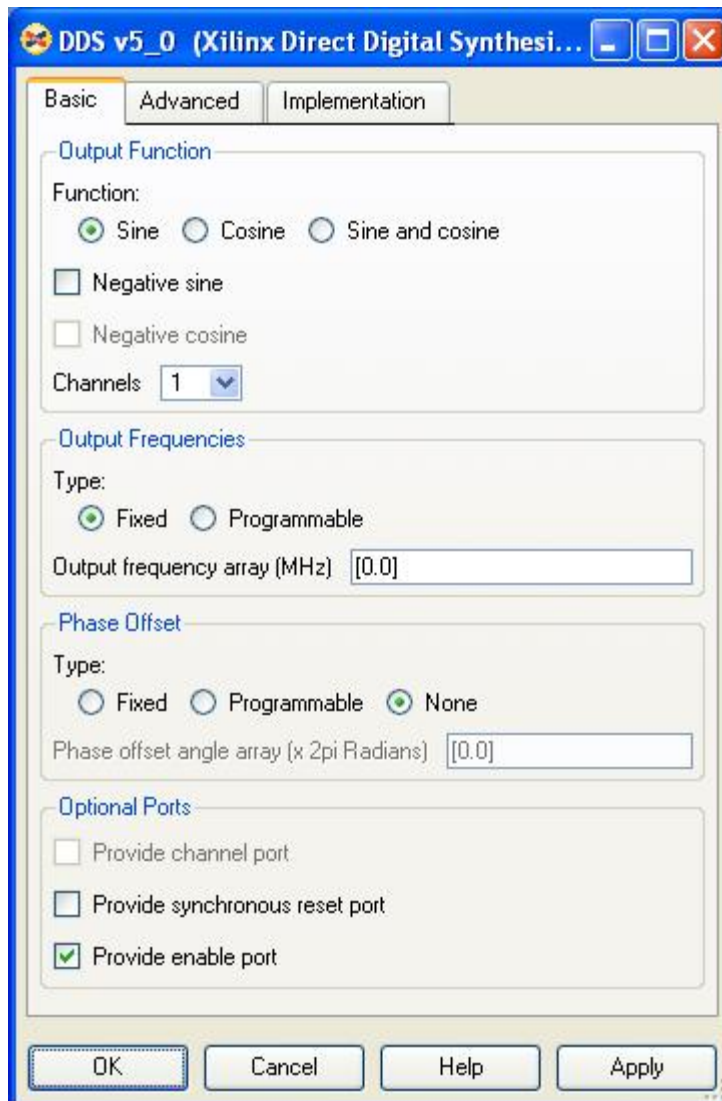


Figure 2: DDS block basic tab parameters

Parameters specific to the DDS block on the basic tab are:

- **Function:** specifies the function(s) that the block will calculate.
- **Negative Sine:** negates the sine output.
- **Negative Cosine:** negates the cosine output.
- **Output Frequency Type:** specifies the output frequency to be either constant or register. Choice of register activates optional ports on the block.
- **Output frequency array (MHz):** for each channel, an independent frequency can be entered into an array.
- **Phase Offset Type:** specifies phase offset to be Constant, Register, or None. Choice of register activates optional ports on the block.
- **Phase offset angle array:** for each channel, an independent offset can be entered into an array. The entered values will be multiplied by 2π radians.
- **Provide channel port:** provide an output channel port to indicate which channel the current output sample corresponds to.

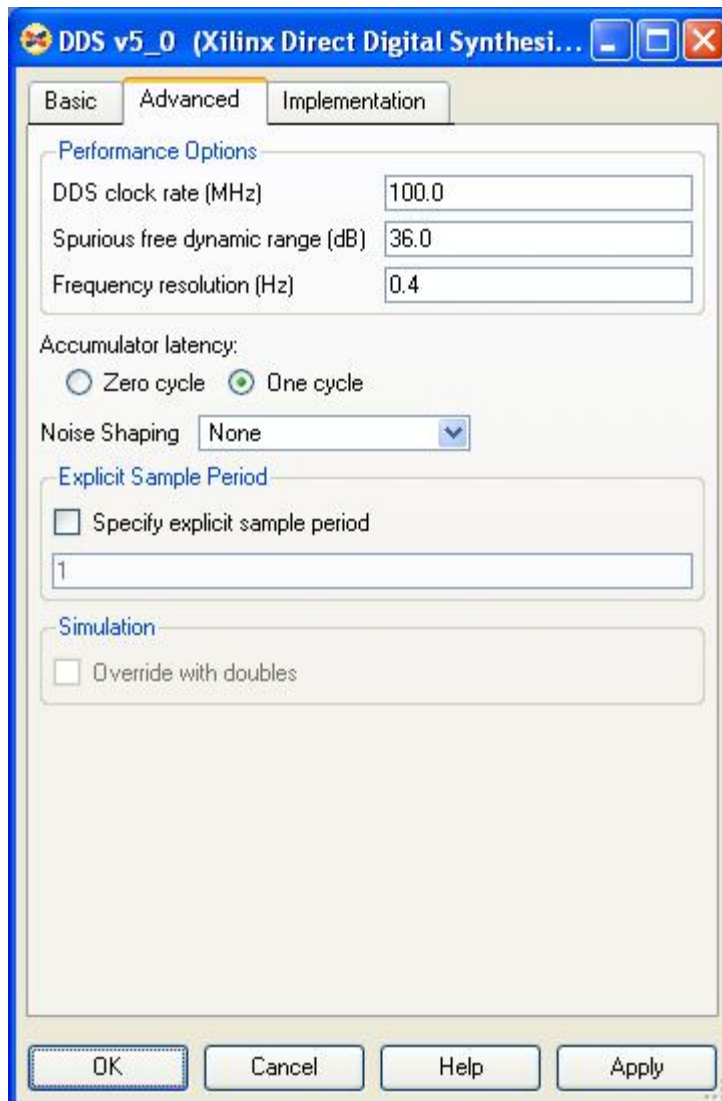


Figure 3: DDS block advanced tab parameters

Parameters specific to the DDS block on the advanced tab are:

- **DDS clock rate (MHz):** frequency at which the DDS core will be clocked.
- **Spurious free dynamic range (dB):** defines the frequency domain requirements of the out-of-band noise generated by the DDS outputs. The range is from 16 to 115 dB of spur suppression. Note that an SFDR value of 102 dB or greater will force an implementation to employ a Taylor Series Correction which requires the use of embedded multipliers.
- **Frequency resolution (Hz):** determines the granularity of the tuning frequency.
- **Accumulator latency:** specifies the latency in the phase accumulator to be zero or one.
- **Noise Shaping:** determines whether a phase truncation, dithered DDS, or Taylor series corrected DDS implementation is generated.

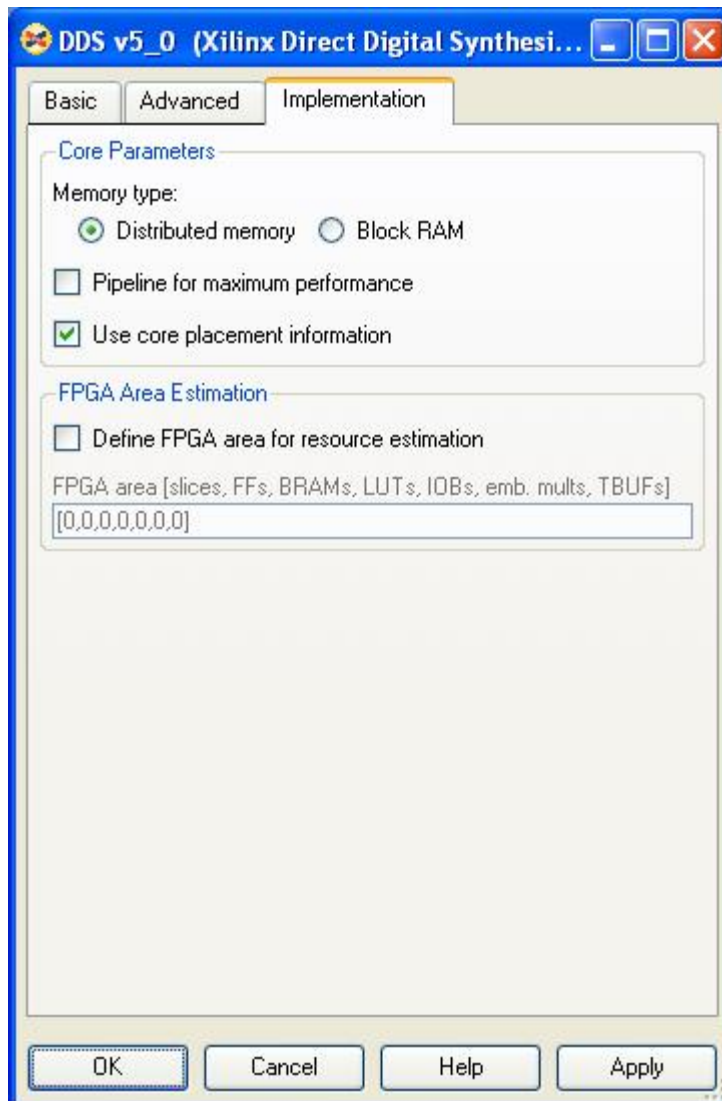


Figure 4: DDS block parameters dialog box with “Show Configuration Parameters” selected

Parameters specific to the DDS block are:

- **Memory type:** directs the block to be implemented either with distributed or block RAM.
- **Pipeline for maximum performance:** when checked, the implementation is fully pipelined.

When noise shaping or memory type is set to auto, the following table shows how the setting of the spurious free dynamic range affects the setting of these parameters.

SFDR	Noise Shaping	Memory Type
≤ 36	None	Distributed memory
≤ 48	Phase Dithering	Distributed memory
≤ 60	None	Block RAM
≤ 102	Phase Dithering	Block RAM
≤ 115	Taylor Series Corrected	Block RAM

Other parameters used by this block are described in the Common Parameters section at the beginning

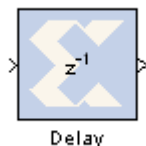
of this chapter.

Xilinx LogiCORE

This block uses the Xilinx LogiCORE Direct Digital Synthesizer.

Delay

This block is listed in the following Xilinx Blockset libraries: *Basic Elements, Memory, and Index.*



The Xilinx Delay block is a delay line (also called a shift register) of configurable length, allowing you to add latency to your design. Data presented at the input will appear at the output after a user specified number of sample periods.

The Delay block differs from the Register in that the Register allows only latency of 1, and contains an Initial Value parameter. The Delay block supports a specified latency, but no initial value, other than zeroes.

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

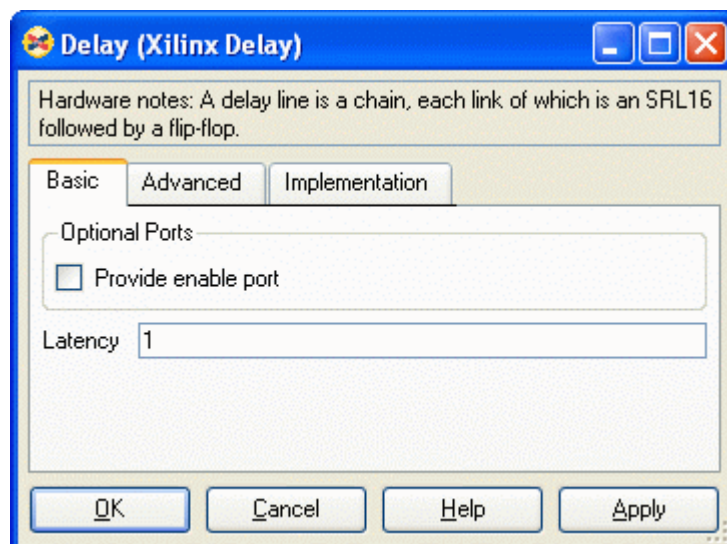


Figure 1: Delay block basic tab parameters

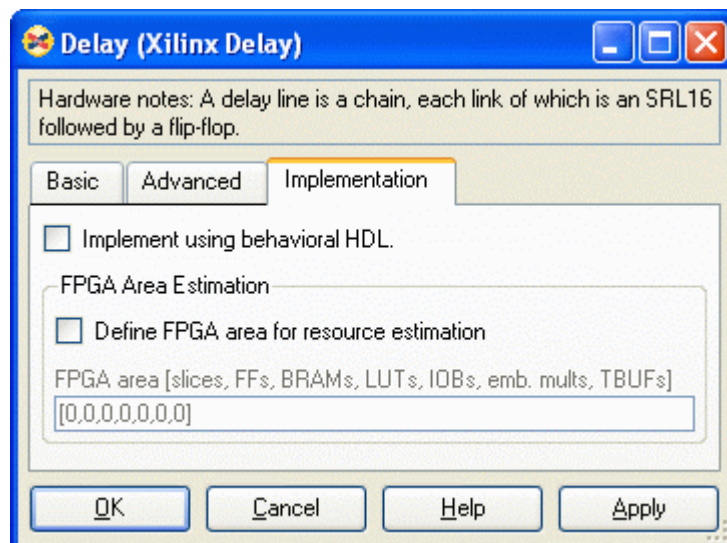


Figure 2: Delay block implementation tab parameters

Parameters specific to the Delay block are:

Implement using behavioral HDL: uses behavioral HDL as the implementation.

Other parameters used by this block are explained in the Common Parameters section at the beginning of this chapter.

Xilinx LogiCORE

The Delay block does not use a Xilinx LogiCORE, but is efficiently mapped to utilize the SRL16 feature of Xilinx devices.

Depuncture

This block is listed in the following Xilinx Blockset libraries: *Communication and Index*.

The Xilinx Depuncture block allows you to insert an arbitrary symbol into your input data at the location specified by the depuncture code. The Xilinx depuncture block accepts data of type $U\text{Fix}_N_0$ where N equals the length of insert string x (the number of ones in the depuncture code) and produces output data of type $U\text{Fix}_K_0$ where K equals the length of insert string multiplied by the length of the depuncture code.

The Xilinx Depuncture block can be used to decode a range of punctured convolution codes. The following diagram illustrates an application of this block to implement soft decision Viterbi decoding of punctured convolution codes.

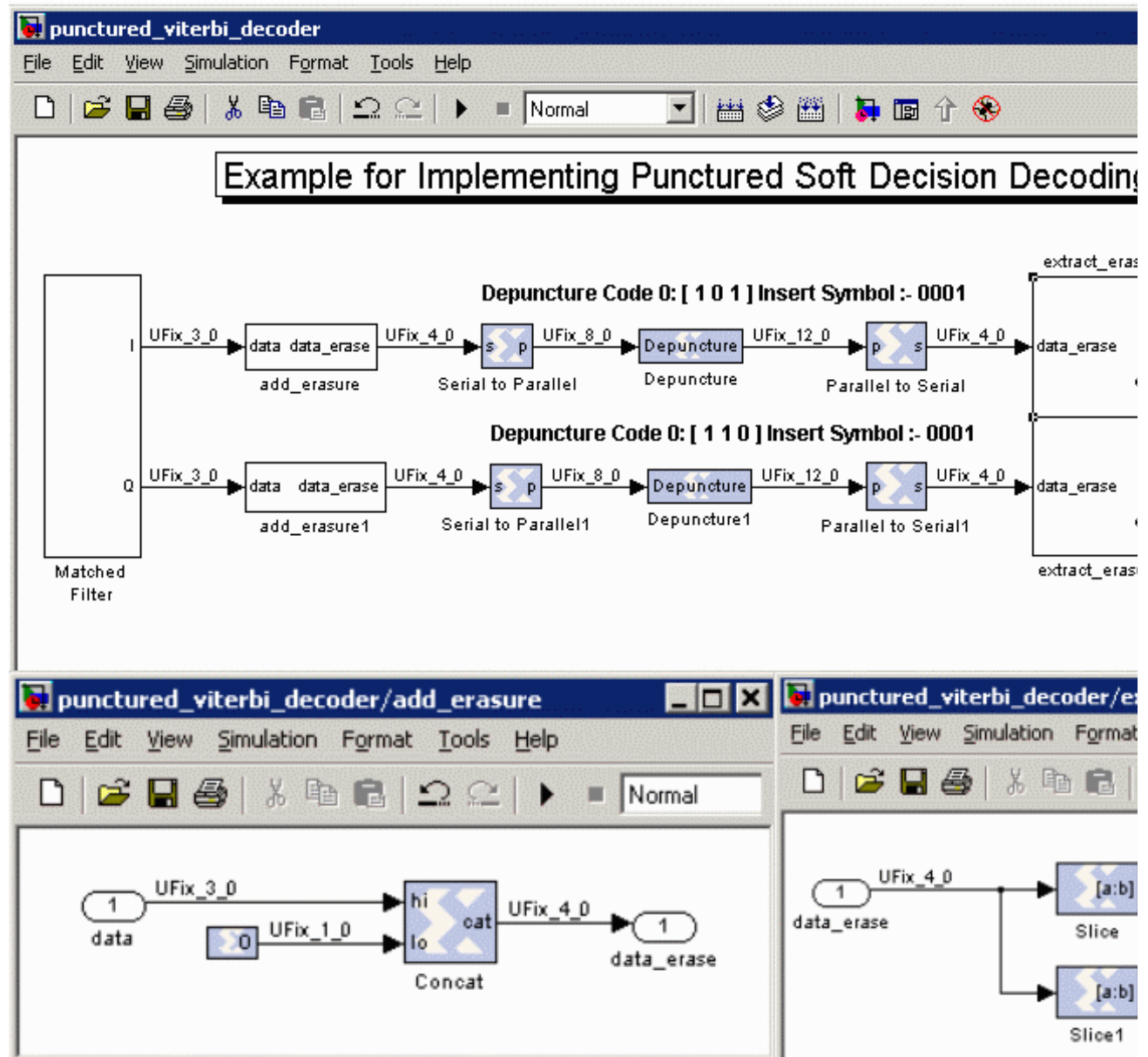


Figure 1: Example of Depuncture block use

The previous diagram shows a matched filter block connected to a `add_erasure` subsystem which attaches a 0 to the input data to mark it as a non-erasure signal. The output from the `add_erasure` subsystem is then passed to a serial to parallel block. The serial to parallel block concatenates two continuous soft inputs and presents it as a 8-bit word to the depuncture block. The depuncture block inserts the symbol '0001' after the 4-bits from the MSB for code 0 ([1 0 1]) and 8-bits from the MSB for code 1 ([1 1 0]) to form a 12-bit word. The output of the depuncture block is serialized as 4-bit words using the parallel to serial block. The `extract_erasure` subsystem takes the input 4-bit word and extracts 3-bits from the MSB to form a soft decision input data word and 1-bit from the LSB to form the erasure signal for the Viterbi decoder.

Block Parameters Dialog Box

The Xilinx depuncture block can be configured using its Block Parameters dialog box.

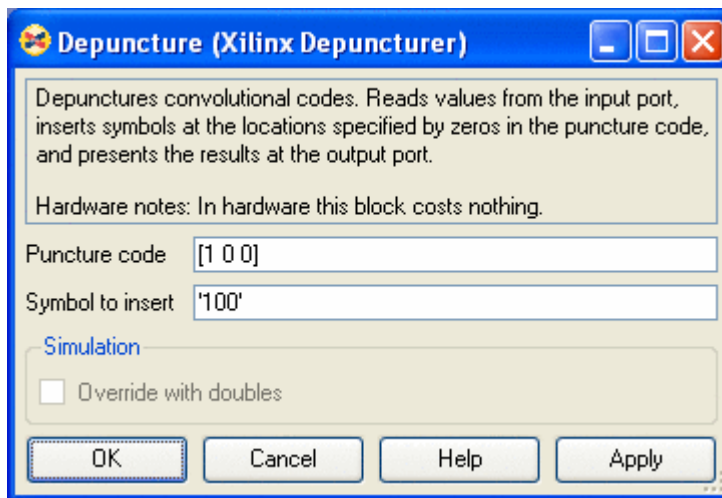


Figure 2: Depuncture block parameters dialog box

Parameters specific to the Xilinx Puncture block are:

- **Puncture Code:** specifies the puncture pattern for inserting the string to the input.
- **Symbol to Insert:** specifies the binary word to be inserted in the puncture code.

Other parameters used by this block are described in the Common Parameters section at the beginning of this chapter.

Disregard Subsystem

This block is listed in the following Xilinx Blockset libraries: Tools and Index.



Disregard Subsystem
For Generation

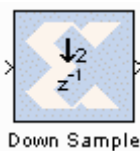
Note: This block has been deprecated since System Generator version 6.2. The block may be eliminated in a future version of System Generator. The functionality supplied by this block is now available through System Generator's support for Simulink's configurable subsystem. Configurable subsystems offers several advantages over the Disregard Subsystem block.

The Disregard Subsystem for Generation block can be placed into any subsystem of your model to indicate that you do not wish System Generator to generate hardware for that subsystem. This block can be used in combination with the simulation multiplexer block to build alternative simulation models for a portion of a design, for example, to provide a simulation model for a black box.

This block has no parameters.

Down Sample

This block is listed in the following Xilinx Blockset libraries: *Basic Elements and Index*



The Xilinx Down Sample block reduces the sample rate at the point where the block is placed in your design. The input signal is sampled at even intervals, at either the beginning (first value) or end (last value) of a frame. The sampled value is presented on the output port and held until the next sample is taken.

A Down Sample frame consists of l input samples, where l is sampling rate. An example frame for a Down Sample block configured with a sampling rate of 4 is shown below.

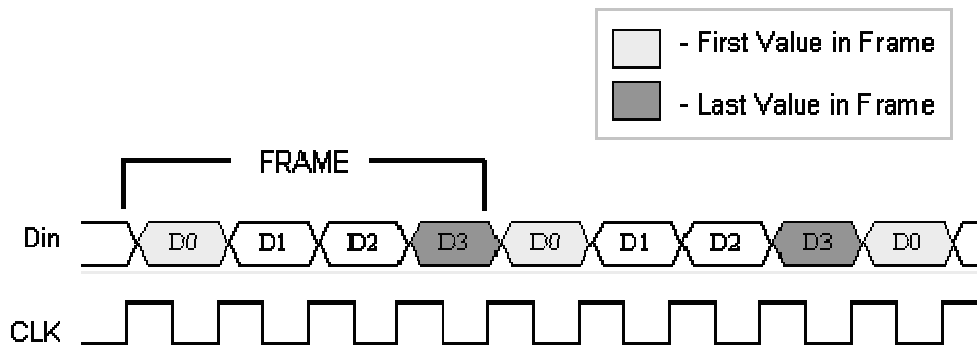


Figure 1: Example Down Sample frame

The Down Sample block is realized in hardware using one of three possible implementations that vary in terms of implementation efficiency. The block receives two clock enable signals in hardware, `Src_CE` and `Dest_CE`. `Src_CE` is the faster clock enable signal and corresponds to the input data stream rate. `Dest_CE` is the slower clock enable, corresponding to the output stream rate, i.e., down sampled data. These enable signals control the register sampling in hardware.

Zero Latency Down Sample

The zero latency Down Sample block must be configured to sample the first value of the frame. The first sample in the input frame passes through the mux to the output port. A register samples this value during the first sample duration and the mux switches to the register output at the start of the second sample of the frame. The result is that the first sample in a frame is present on the output port for the entire frame duration. This is the least efficient hardware implementation as the mux introduces a combinational path from `Din` to `Dout`. A single bit register adjusts the timing of the destination clock enable, so that it is asserted at the start of the sample period, instead of the end. The hardware implementation is shown below:

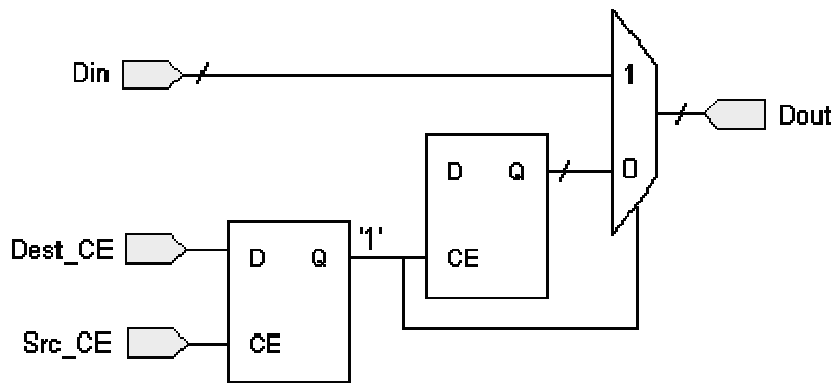


Figure 2: Zero latency Down Sample block

Down Sample with Latency

If the Down Sample block is configured with latency greater than zero, a more efficient implementation is used. One of two implementations is selected depending on whether the Down Sample block is set to sample the first or last value in a frame.

Sample First Value in Frame

In this case, two registers are required to correctly sample the input stream. The first register is enabled by the adjusted clock enable signal so that it samples the input at the start of the input frame. The second register samples the contents of the first register at the end of the sample period to ensure output data is aligned correctly.

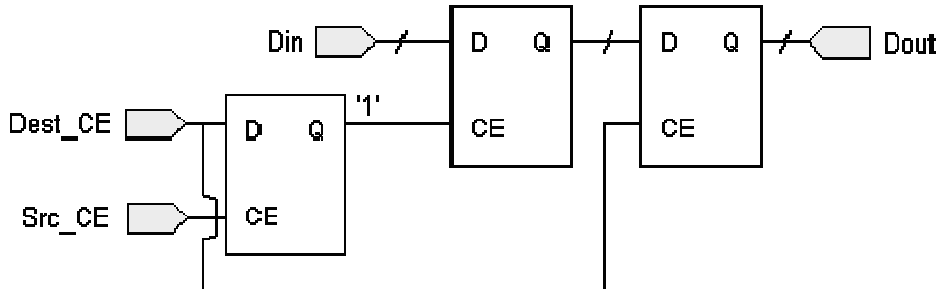


Figure 3: Down Sample with latency equals 1 configured to sample first value of frame

Sample Last Value of Frame

The most efficient implementation is used when the Down Sample block is configured to sample the last value of the frame. In this case, a register samples the data input data at the end of the frame. The sampled value is presented for the duration of the next frame.

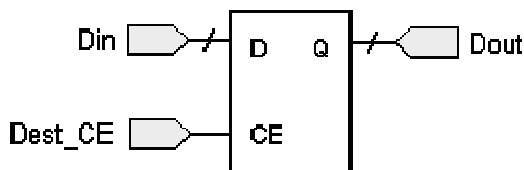


Figure 4: Down Sample with latency equals 1 block configured to sample last value of frame

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

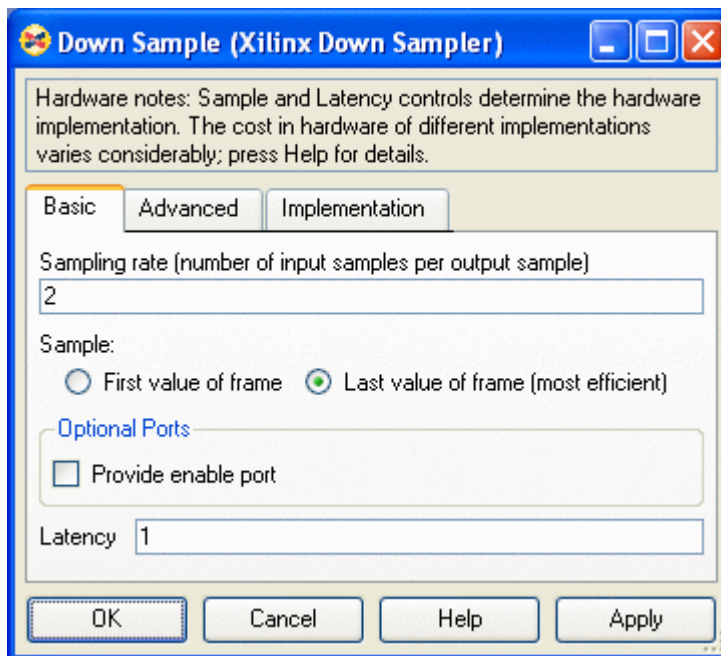


Figure 5:Downsample block Basic Tab

Parameters specific to the block are:

- **Sampling Rate (number of input samples per output sample)**
(Basic): must be an integer greater or equal to 2. This is the ratio of the output sample period to the input, and is essentially a sample rate divider. For example, a ratio of 2 indicates a 2:1 division of the input sample rate. If a non-integer ratio is desired, the Up Sample block can be used in combination with the Down Sample block.
- **Sample (Basic):** The Down Sample block can sample either the first or last value of a frame. This parameter will determine which of these two values is sampled.

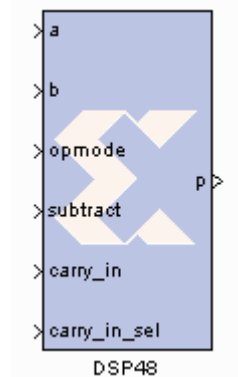
Other parameters used by this block are explained in the Common Parameters section at the beginning of this chapter.

Xilinx LogiCORE

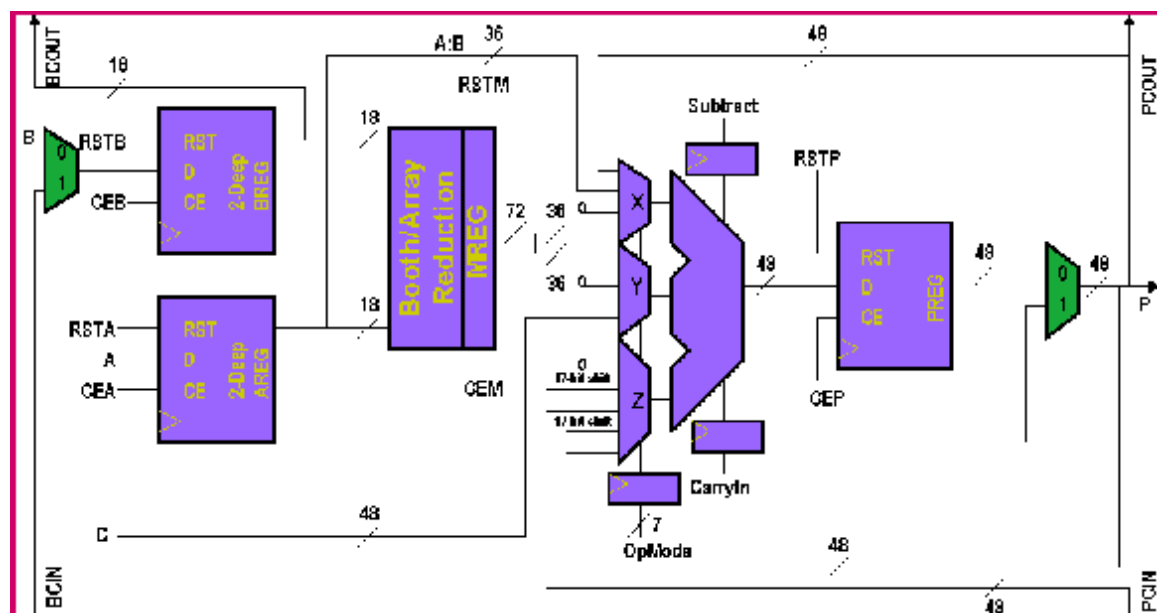
The Down Sample block does not use a Xilinx LogiCORE.

DSP48

This block is listed in the following Xilinx Blockset libraries: Index, DSP.



The Xilinx DSP48 block is an efficient building block for DSP applications that use Xilinx Virtex-4 devices. The DSP48 combines an 18-bit by 18-bit signed multiplier with a 48-bit adder and programmable mux to select the adder's input. Operations can be selected dynamically. Optional input and multiplier pipeline registers can be selected as well as registers for the subtract, carryin and opmode ports. The DSP48 block can also target devices that do not contain the DSP48 hardware primitive if the `Use synthesizable model` option is selected.



Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

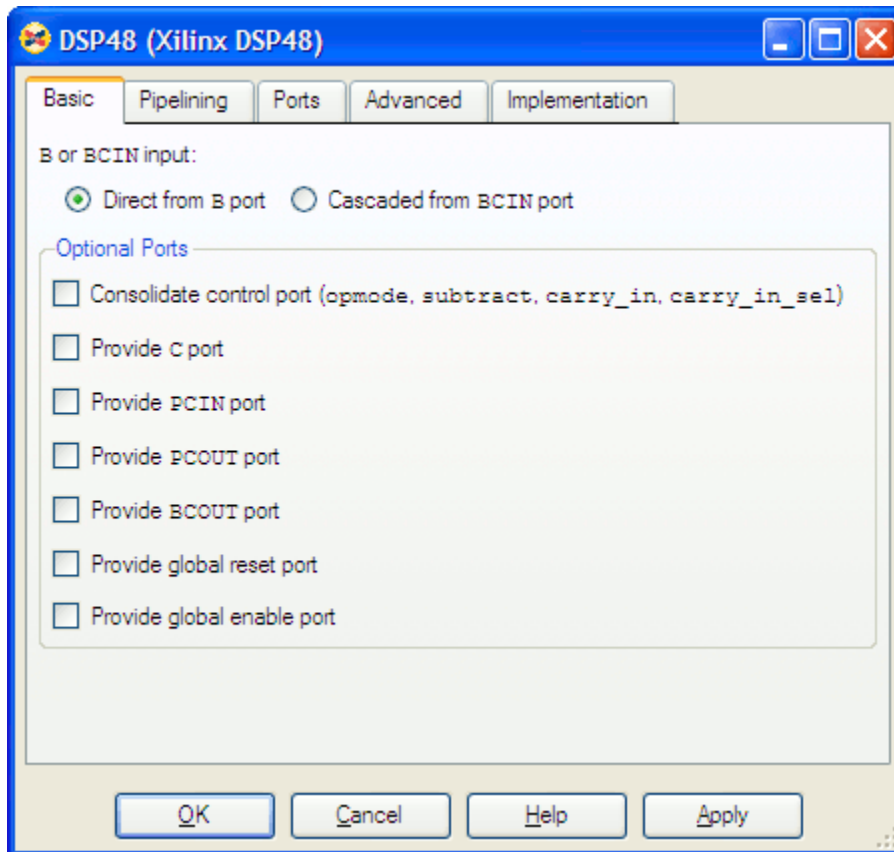


Figure 1: DSP48 block Basic tab

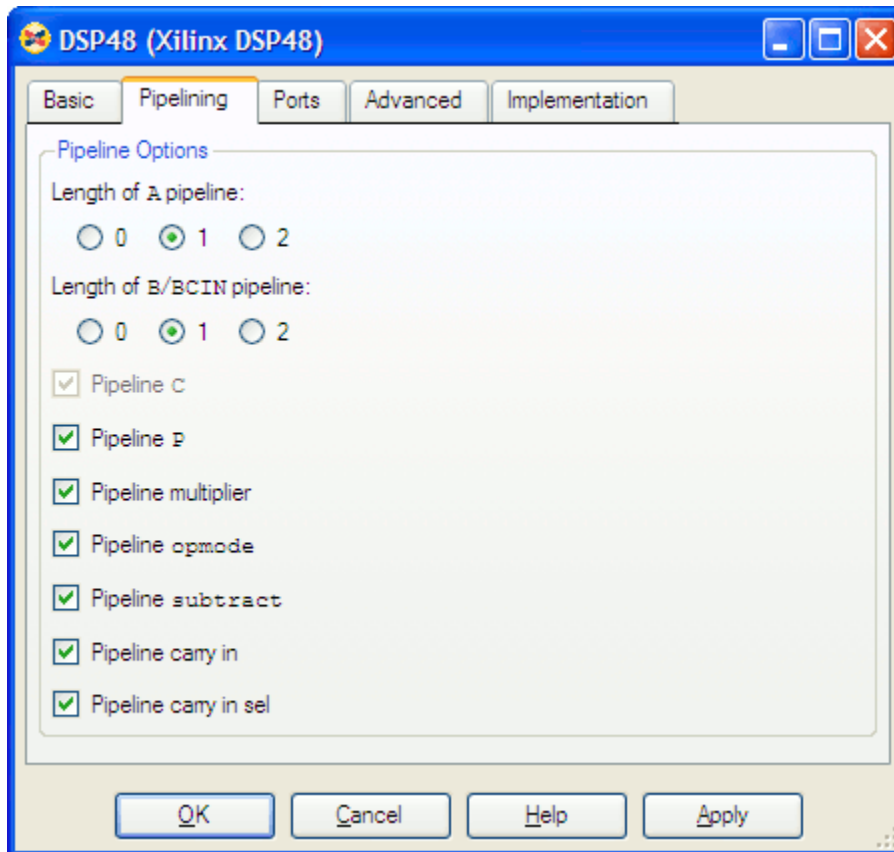


Figure 2: DSP48 block Pipelining tab

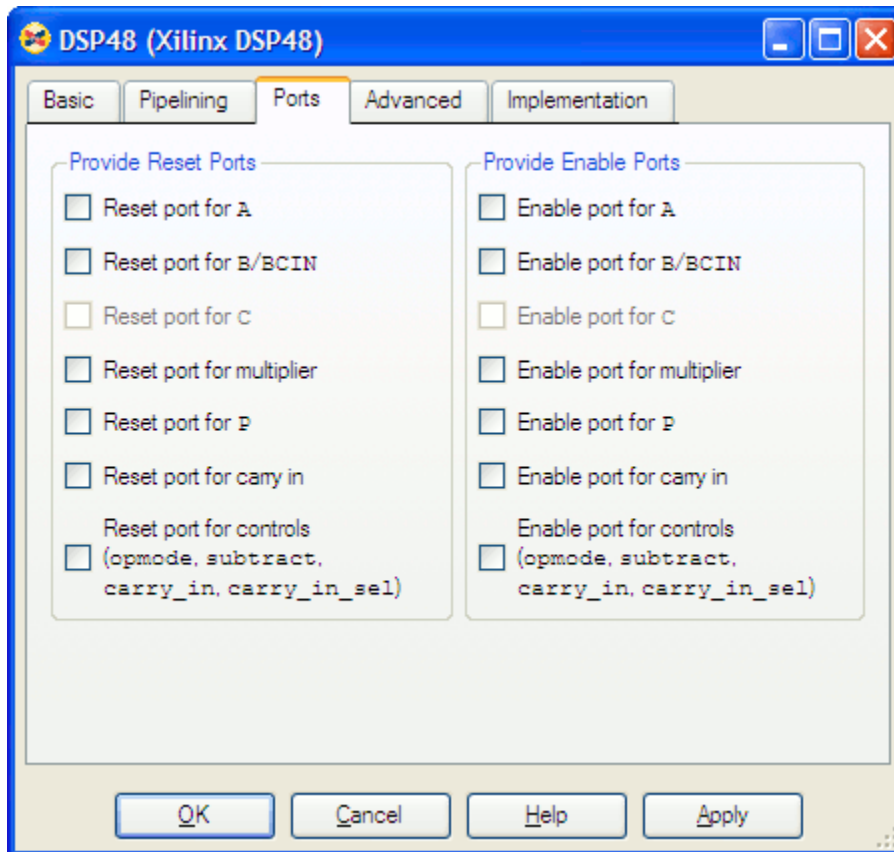


Figure 3: DSP48 block Ports tab

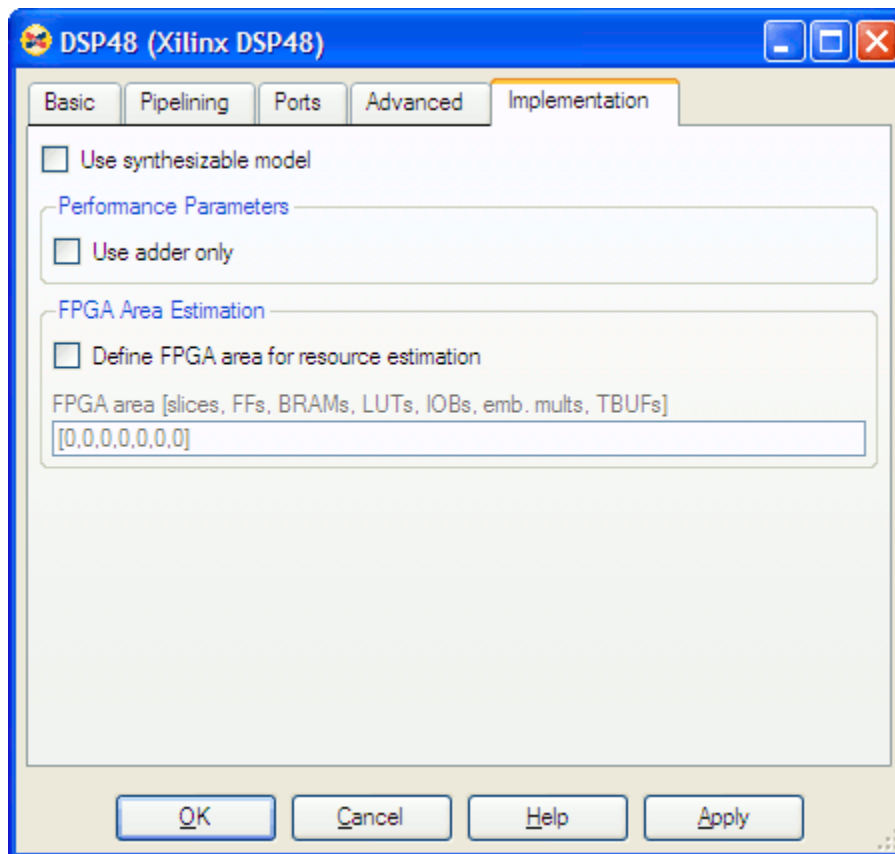


Figure 4: DSP48 block Implementation tab

Parameters specific to the block are:

- **B or BCIN input:** specifies if the B input should be taken directly from the b port or from the cascaded bcin port. The bcin port can only be connected to another DSP48 block.
- **Consolidate control port:** when selected, combines the opmode, subtract, carry_in and carry_in_sel ports into one 11-bit port. Bits 0 to 6 are the opmode, bit 7 is the subtract port, bit 8 is the carry_in port, and bits 9 and 10 are the carry_in_sel port. This option should be used when a Opmode block is used to generate a DSP48 instruction.
- **Provide C port:** when selected, the c port is made available. Otherwise, the c port is tied to '0'.
- **Provide PCIN port:** when selected, the pcin port is exposed. The pcin port must be connected to the pcout port of another DSP48 block.
- **Provide PCOUT port:** when selected, the pcout output port is made available. The pcout port must be connected to the pcin port of another DSP48 block.
- **Provide BCOUT port:** when selected, the bcout output port is made available. The bcout port must be connected to the bcin port of another DSP48 block.
- **Provide global reset port:** when selected, the port rst is made available. This port is connected to all available reset ports based on the pipeline selections.
- **Provide global enable port:** when selected, the optional en port is made available. This port is connected to all available enable ports based on the pipeline selections.
- **Length of A pipeline:** specifies the length of the pipeline on input register A. A pipeline of length 0 removes the register on the input.
- **Length of B/BCIN pipeline:** specifies the length of the pipeline for the b input whether it is read from b or bcin.
- **Pipeline C:** indicates whether the input from the c port should be registered.
- **Pipeline P:** indicates whether the outputs p and pcout should be registered.

- **Pipeline multiplier:** indicates whether the internal multiplier should register its output.
- **Pipeline opmode:** indicates whether the `opmode` port should be registered.
- **Pipeline subtract:** indicates whether the `subtract` port should be registered.
- **Pipeline carry in:** indicates whether the `carry_in` port should be registered.
- **Pipeline carry in sel:** indicates whether the `carry_in_sel` port should be registered.
- **Reset port for A:** when selected, a port `rst_a` is made available. This resets the pipeline register for port `a` when set to '1'.
- **Reset port for B:** when selected, a port `rst_b` is made available. This resets the pipeline register for port `b` when set to '1'.
- **Reset port for C:** when selected, a port `rst_c` is made available. This resets the pipeline register for port `c` when set to '1'.
- **Reset port for multiplier:** when selected, a port `rst_m` is made available. This resets the pipeline register for the internal multiplier when set to '1'.
- **Reset port for P:** when selected, a port `rst_p` is made available. This resets the output register when set to '1'.
- **Reset port for carry in:** when selected, a port `rst_carryin` is made available. This resets the pipeline register for carry in when set to '1'.
- **Reset port for controls (opmode, subtract, carry_in, carry_in_sel):** when selected, a port `rst_ctrl` is made available. This resets the pipeline register for the subtract register (if available), `opmode` register (if available) and `carry_in_sel` register (if available) when set to '1'.
- **Enable port for A:** when selected, an enable port `ce_a` for the port `A` pipeline register is made available.
- **Enable port for B:** when selected, an enable port `ce_b` for the port `B` pipeline register is made available.
- **Enable port for C:** when selected, an enable port `ce_c` for the port `C` register is made available.
- **Enable port for multiplier:** when selected, an enable port `ce_m` for the multiplier register is made available.
- **Enable port for P:** when selected, an enable port `ce_p` for the port `P` output register is made available.
- **Enable port for carry in:** when selected, an enable port `ce_carry_in` for the carry in register is made available.
- **Enable port for controls (opmode, subtract, carry_in, carry_in_sel):** when selected, the enable ports `ce_ctrl` and `ce_cinsub` are made available. The port `ce_ctrl` controls the `opmode` and carry in select registers while `ce_cinsub` controls the subtract register.
- **Use synthesizable model:** when selected, the DSP48 is implemented from an RTL description which may not map directly to the DSP48 hardware. This is useful if a design using the DSP48 block is targeted at device families that do not contain DSP48 hardware primitives.
- **Use adder only:** when selected, the block is optimized in hardware for maximum performance without using the multiplier. If an instruction using the multiplier is encountered in simulation, an error is reported.

Other parameters used by this block are explained in the Common Parameters section at the beginning of this chapter.

See Also

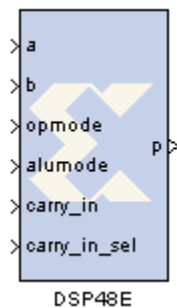
The following documents are provided as part of the System Generator User's guide and give valuable insight into using and understanding the DSP48 block:

DSP48 Macro
Generating Block Diagrams Programmatically

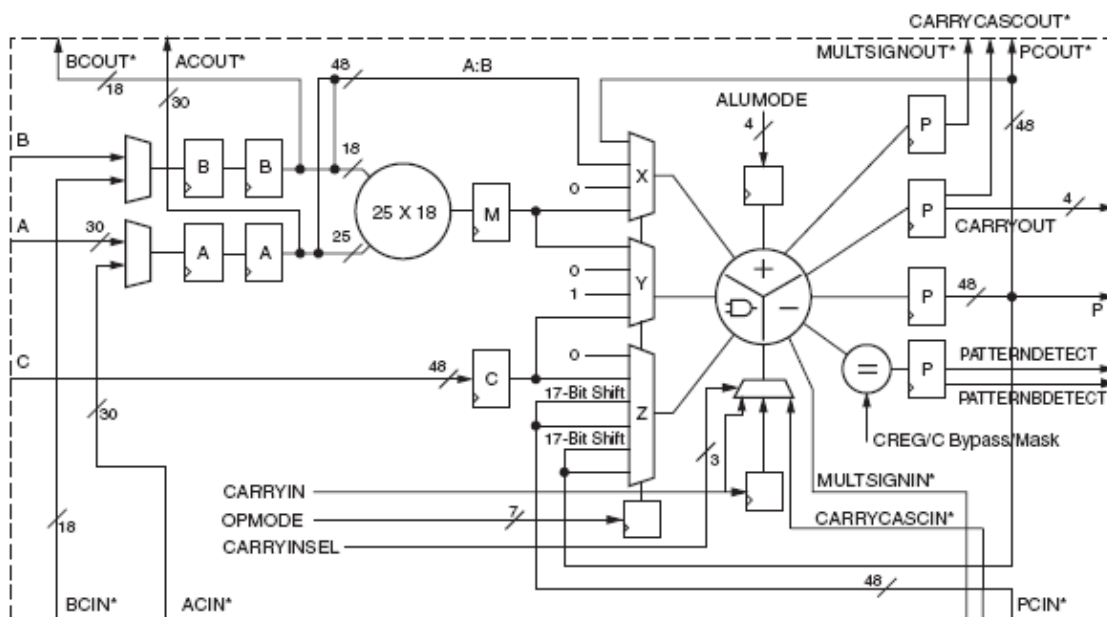
XtremeDSP™ Slice

DSP48E

This block is listed in the following Xilinx Blockset libraries: Index, DSP.



The Xilinx DSP48E block is an efficient building block for DSP applications that use Xilinx Virtex-5 devices. The DSP48E combines an 18-bit by 25-bit signed multiplier with a 48-bit adder and programmable mux to select the adder's input. Operations can be selected dynamically. Optional input and multiplier pipeline registers can be selected as well as registers for the alumode, carryin and opmode ports. The DSP48E block can also target devices that do not contain the DSP48E hardware primitive if the `Use synthesizable model` option is selected on the implementation tab.



*These signals are dedicated routing paths internal to the DSP48E column. They are not accessible via fabric routing resources.

Figure 1: DSP48E Slice

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

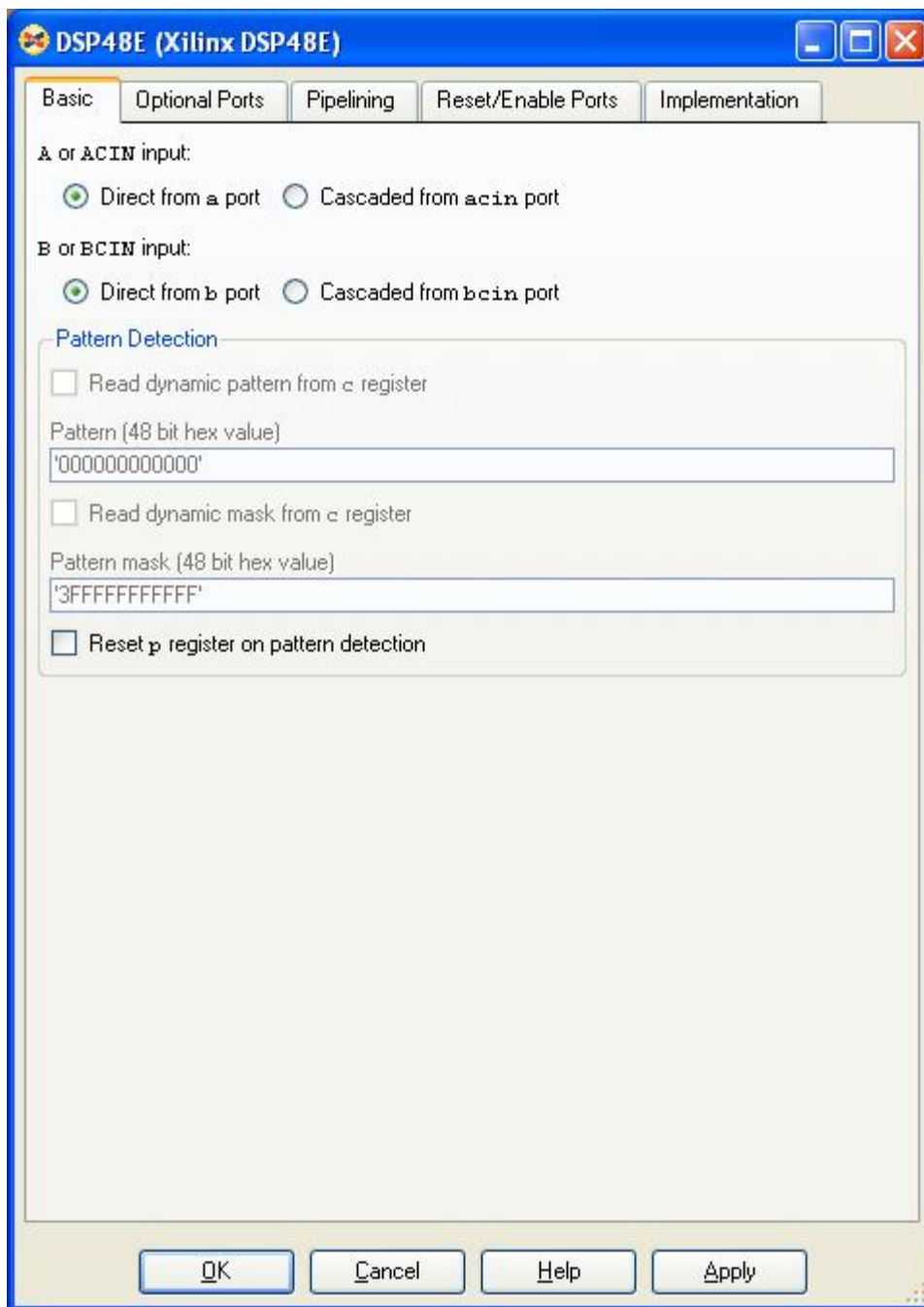


Figure 1: DSP48E block basic tab

Parameters specific to the basic tab are:

- **A or ACIN input:** specifies if the A input should be taken directly from the a port or from the cascaded acin port. The acin port can only be connected to another DSP48 block.
- **B or BCIN input:** specifies if the B input should be taken directly from the b port or from the cascaded bcin port. The bcin port can only be connected to another DSP48 block.
- **Read dynamic pattern from c register:** when selected, the pattern used in pattern detection is read from the c port.
- **Pattern (48 bit hex value):** value is used in pattern detection logic which is best described as an equality check on the output of the adder/subtractor/logic unit

- Read dynamic mask from c register: when selected, the mask used in pattern detection is read from the c port.
- Pattern mask (48 bit hex value): 48-bit value used to mask out certain bits during pattern detection.
- Reset p register on pattern detection: if selected and the pattern is detected, reset the p register on the next cycle.

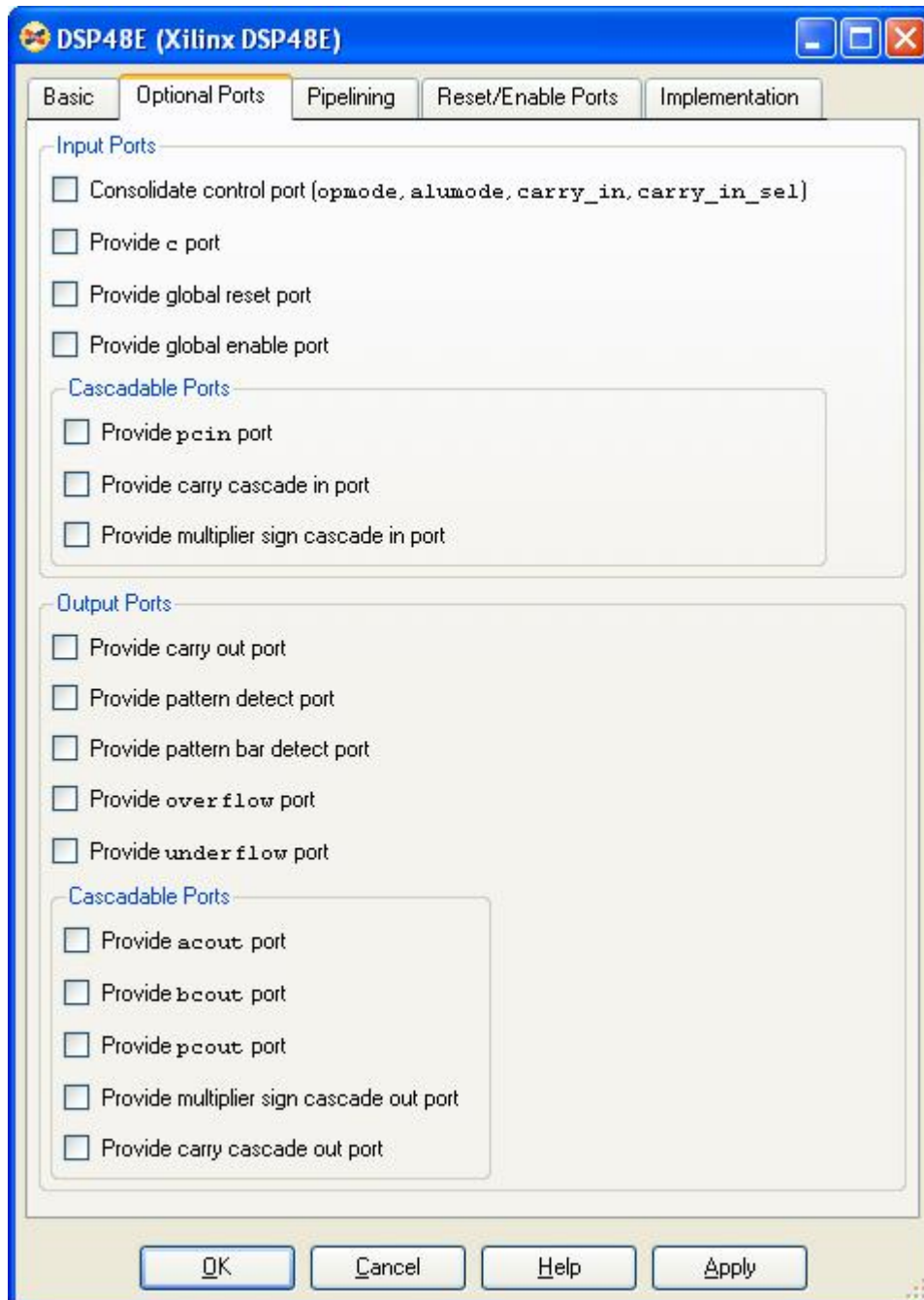


Figure 2: DSP48E block option ports tab

Parameters specific to the optional ports tab are:

- Consolidate control port: when selected, combines the opmode, alumode, carry_in and carry_in_sel ports into one 15-bit port. Bits 0 to 6 are the opmode, bits 7

to 10 are the `alumode` port, bit 11 is the `carry_in` port, and bits 12 to 14 are the `carry_in_sel` port. This option should be used when the `Opmode` block is used to generate a DSP48E instruction.

- **Provide `c` port:** when selected, the `c` port is made available. Otherwise, the `c` port is tied to '0'.
- **Provide `global reset` port:** when selected, the port `rst` is made available. This port is connected to all available reset ports based on the pipeline selections.
- **Provide `global enable` port:** when selected, the optional `en` port is made available. This port is connected to all available enable ports based on the pipeline selections.
- **Provide `pcin` port:** when selected, the `pcin` port is exposed. The `pcin` port must be connected to the `pcout` port of another DSP48 block.
- **Provide `carry cascade in` port:** when selected, the carry cascade in port is exposed. This port can only be connected to a carry cascade out port on another DSP48E block.
- **Provide `multiplier sign cascade in` port:** when selected, the multiplier sign cascade in port (`multsigncascin`) is exposed. This port can only be connected to a multiplier sign cascade out port of another DSP48E block.
- **Provide `carryout` port:** when selected, the `carryout` output port is made available. When the mode of operation for the adder/subtractor is set to one 48-bit adder, the carryout port is 1-bit wide. When the mode of operation is set to two 24 bit adders, the carryout port is 2 bits wide. The MSB corresponds to the second adder's carryout and the LSB corresponds to the first adder's carryout. When the mode of operation is set to four 12 bit adders, the carryout port is 4 bits wide with the bits corresponding to the addition of the 48 bit input split into 4 12-bit sections.
- **Provide `pattern detect` port:** when selected, the pattern detection output port is provided. When the pattern, either from the mask or the `c` register, is matched the pattern detection port is set to '1'.
- **Provide `pattern bar detect` port:** when selected, the pattern bar detection (`patternbndetect`) output port is provided. When the inverse of the pattern, either from the mask or the `c` register, is matched the pattern bar detection port is set to '1'.
- **Provide `overflow` port:** when selected, the overflow output port is provided. This port indicates when the operation in the DSP48E has overflowed beyond the bit `P[N]` where `N` is between 1 and 46. `N` is determined by the number of 1s in the mask whether set by the GUI mask field or the `c` port input.
- **Provide `underflow` port:** when selected, the underflow output port is provided. This port indicates when the operation in the DSP48E has underflowed. Underflow occurs when the number goes below `-P[N]` where `N` is determined by the number of 1s in the mask whether set by the GUI mask field or the `c` port input.
- **Provide `ACOUT` port:** when selected, the `acout` output port is made available. The `acout` port must be connected to the `acin` port of another DSP48E block.
- **Provide `BCOUT` port:** when selected, the `bcout` output port is made available. The `bcout` port must be connected to the `bcin` port of another DSP48E block.
- **Provide `PCOUT` port:** when selected, the `pcout` output port is made available. The `pcout` port must be connected to the `pcin` port of another DSP48 block.
- **Provide `multiplier sign cascade out` port:** when selected, the multiplier sign cascade out port (`multsigncascout`) is made available. This port can only be connected to the multiplier sign cascade in port of another DSP48E block and is used to support 96-bit accumulators/adders and subtractors which are built from two DSP48Es.
- **Provide `carry cascade out` port:** when selected, the carry cascade out port (`carrycascout`) is made available. This port can only be connected to the carry cascade in port of another DSP48E block.

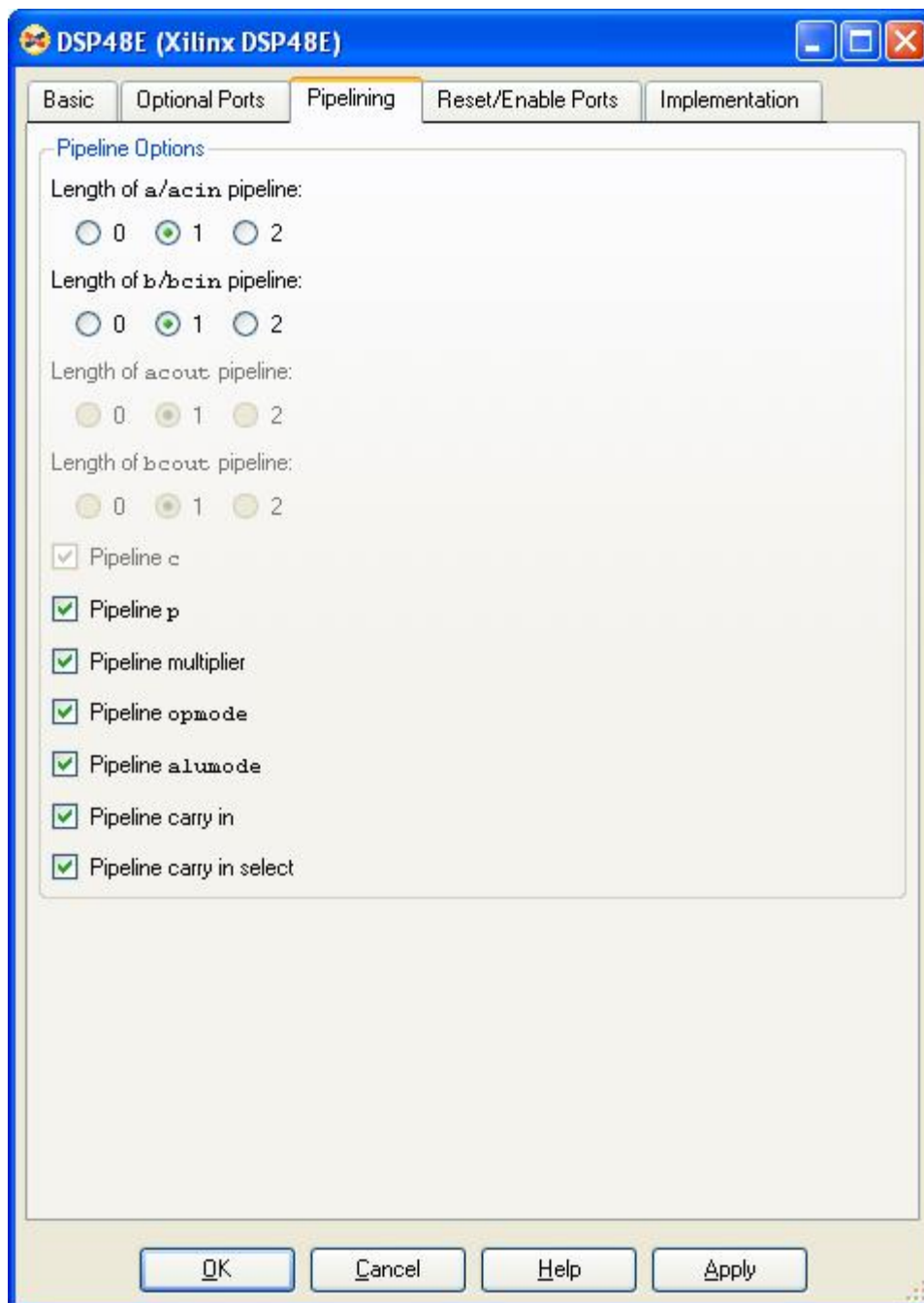


Figure 3: DSP48E block pipelining tab

Parameters specific to the pipelining tab are:

- **Length of a/acin pipeline:** specifies the length of the pipeline on input register A. A pipeline of length 0 removes the register on the input.
- **Length of b/bcin pipeline:** specifies the length of the pipeline for the b input whether it is read from b or bcin.
- **Length of acout pipeline:** specifies the length of the pipeline between the a/acin input and the acout output port. A pipeline of length 0 removes the register from the acout pipeline length. Must be less than or equal to the length of the a/acin pipeline.

- **Length of bcout pipeline:** specifies the length of the pipeline between the b/bcin input and the bcout output port. A pipeline of length 0 removes the register from the bcout pipeline length. Must be less than or equal to the length of the b/bcin pipeline.
- **Pipeline c:** indicates whether the input from the c port should be registered.
- **Pipeline p:** indicates whether the outputs p and pcout should be registered.
- **Pipeline multiplier:** indicates whether the internal multiplier should register its output.
- **Pipeline opmode:** indicates whether the opmode port should be registered.
- **Pipeline alumode:** indicates whether the alumode port should be registered.
- **Pipeline carry in:** indicates whether the carry in port should be registered.
- **Pipeline carry in sel:** indicates whether the carry in select port should be registered.

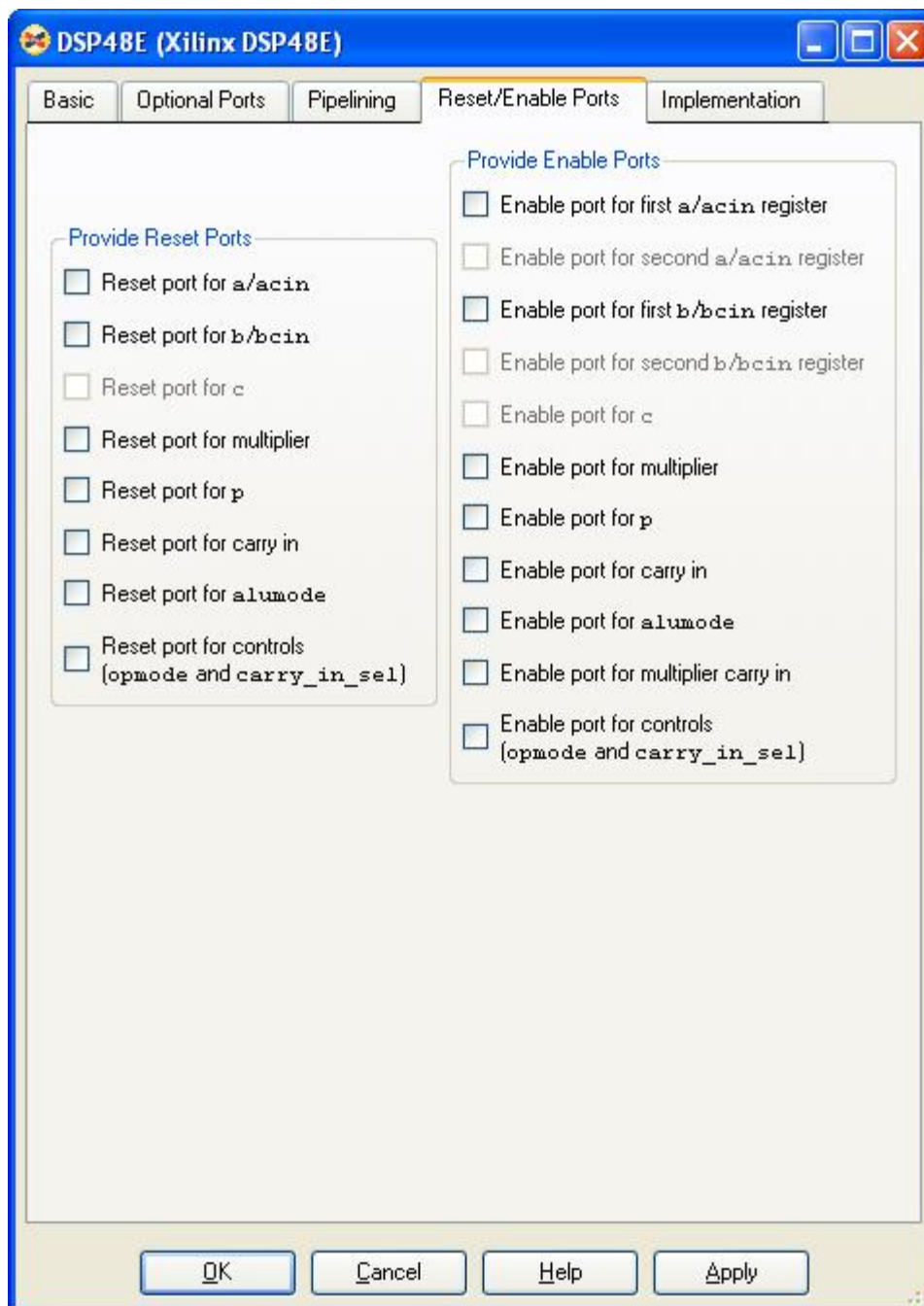


Figure 4: DSP48E block reset and enable ports tab

Parameters specific to the reset and enable tab are:

- **Reset port for a/acin:** when selected, a port `rst_a` is made available. This resets the pipeline register for port a when set to '1'.
- **Reset port for b/bcin:** when selected, a port `rst_b` is made available. This resets the pipeline register for port b when set to '1'.
- **Reset port for c:** when selected, a port `rst_c` is made available. This resets the pipeline register for port c when set to '1'.
- **Reset port for multiplier:** when selected, a port `rst_m` is made available. This resets the pipeline register for the internal multiplier when set to '1'.

- Reset port for P: when selected, a port `rst_p` is made available. This resets the output register when set to '1'.
- Reset port for carry in: when selected, a port `rst_carryin` is made available. This resets the pipeline register for carry in when set to '1'.
- Reset port for alumode: when selected, a port `rst_alumode` is made available. This resets the pipeline register for the alumode port when set to '1'.
- Reset port for controls (opmode and carry_in_sel): when selected, a port `rst_ctrl` is made available. This resets the pipeline register for the opmode register (if available) and the carry_in_sel register (if available) when set to '1'.
- Enable port for first a/acin register : when selected, an enable port `ce_a1` for the first a pipeline register is made available.
- Enable port for second a/acin register : when selected, an enable port `ce_a2` for the second a pipeline register is made available.
- Enable port for first b/bcin register : when selected, an enable port `ce_b1` for the first b pipeline register is made available.
- Enable port for second b/bcin register : when selected, an enable port `ce_b2` for the second b pipeline register is made available.
- Enable port for c: when selected, an enable port `ce_c` for the port C register is made available.
- Enable port for multiplier: when selected, an enable port `ce_m` for the multiplier register is made available.
- Enable port for p: when selected, an enable port `ce_p` for the port P output register is made available.
- Enable port for carry in: when selected, an enable port `ce_carry_in` for the carry in register is made available.
- Enable port for alumode: when selected, an enable port `ce_alumode` for the alumode register is made available.
- Enable port for multiplier carry in: when selected, an enable port `mult_carry_in` for the multiplier register is made available.
- Enable port for controls (opmode and carry_in_sel): when selected, the enable port `ce_ctrl` is made available. The port `ce_ctrl` controls the opmode and carry in select registers.

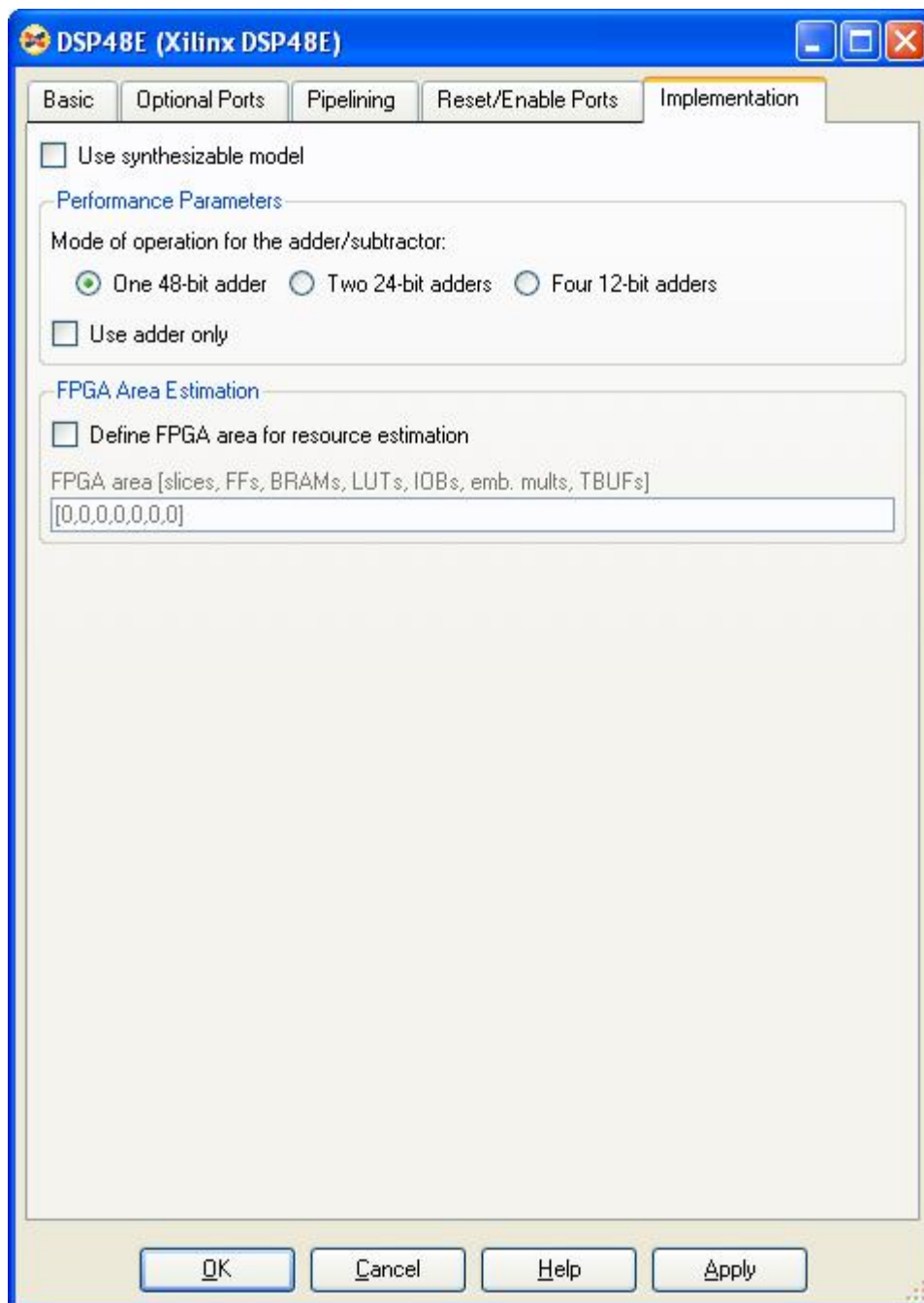


Figure 5: DSP48E block implementation tab

Parameters specific to the implementation tab are:

- **Use synthesizable model:** when selected, the DSP48E is implemented from an RTL description which may not map directly to the DSP48E hardware. This is useful if a design using the DSP48E block is targeted at device families that do not contain DSP48E hardware primitives.
- **Mode of operation for the adder/subtractor:** this mode can be used to implement small add-subtract functions at high speed and lower power with less logic utilization. The adder and subtractor in the adder/subtracted/logic unit can also be split into two 24-bit fields or four 12-bit fields. This is achieved by setting the mode of operation to “Two 24-bit

- adders” or “Four 12-bit adders”. See the Virtex-5 XtremeDSP Design Considerations for more details.
- **Use adder only:** when selected, the block is optimized in hardware for maximum performance without using the multiplier. If an instruction using the multiplier is encountered in simulation, an error is reported.

Other parameters used by this block are explained in the Common Parameters section at the beginning of this chapter.

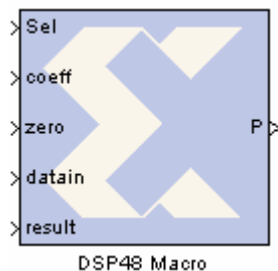
See Also

The following documents are provided as part of the System Generator User’s guide and give valuable insight into using and understanding the DSP48E block:

DSP48 Macro
Generating Block Diagrams Programmatically
Virtex-5 XtremeDSP™ Design Considerations
DSP48E Slice

DSP48 Macro

This block is listed in the following Xilinx Blockset libraries: Index, Control.



The System Generator DSP48 Macro block provides a wrapper to the DSP48/DSP48E block. It differs from most of the other system generator blocks, in that, it is programatically generated (see Generating Block Diagrams Programatically) and consists of other System Generator blocks under the mask. It wraps one DSP48 block along with reinterpret and convert blocks for data type alignment, multiplexers to handle multiple opmodes and inputs, and registers.

Block Interface

The DSP48 Macro block has a variable number of inputs and outputs determined from user-specified parameter values. The input data ports are determined by the opmodes entered in the Instructions field of the DSP48 Macro. Input port Sel appears if more than one opmode is present in the Instructions field. The Instructions field is discussed in greater detail in the section on Entering Opmodes in the DSP48 Macro block.

Port P, an output data port, is the only port appearing in all configurations of the DSP48 Macro. Output ports PCOUT, BCOUT, ACOUT, CARRYOUT and CARRYCASCOUT appear depending on the user-selections.

DSP48 Macro Parameters Dialog Box

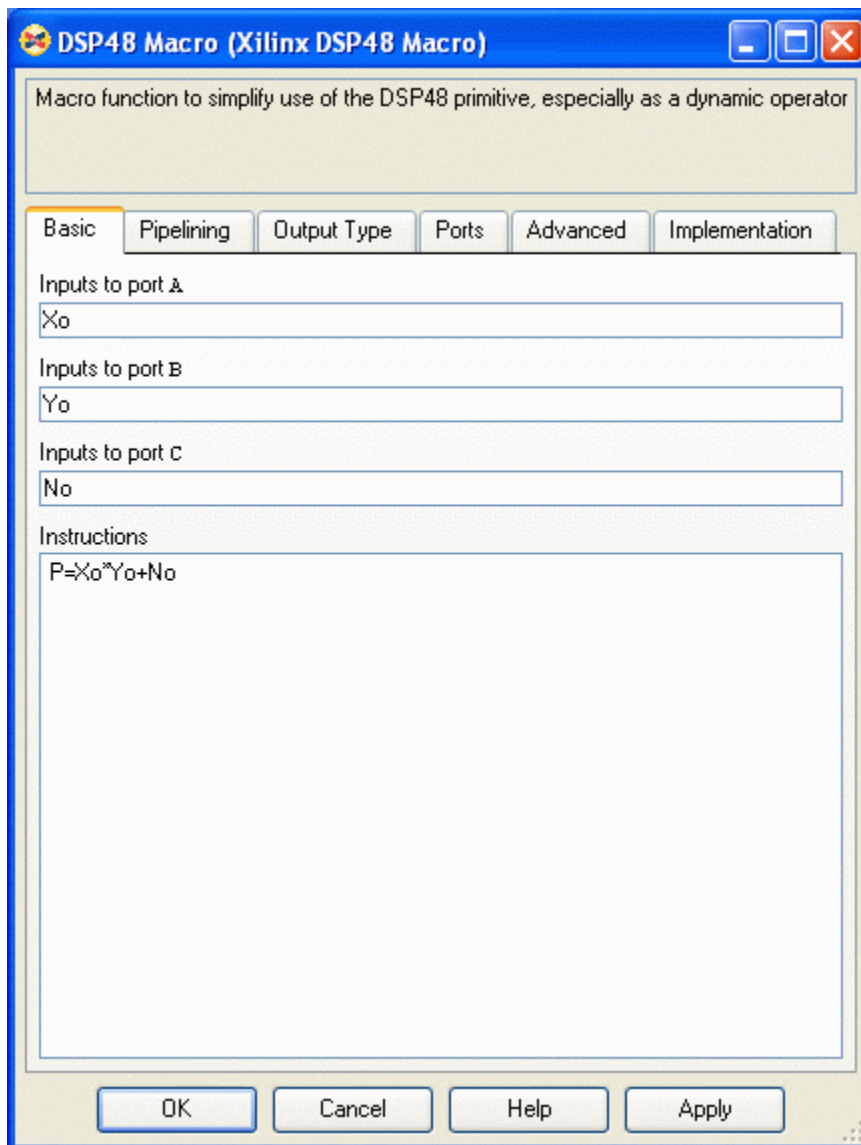


Figure 1 DSP48 Macro block Basic Tab

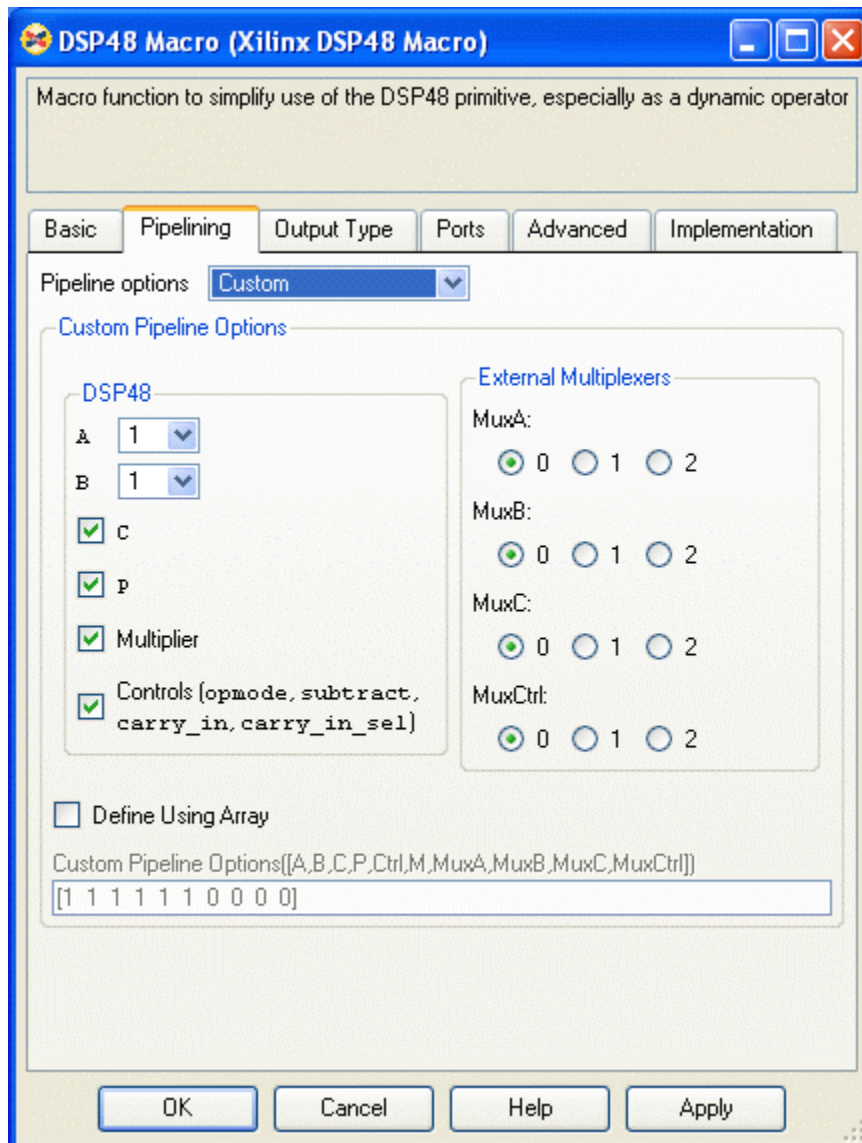


Figure 2: DSP48 Macro Pipelining Tab

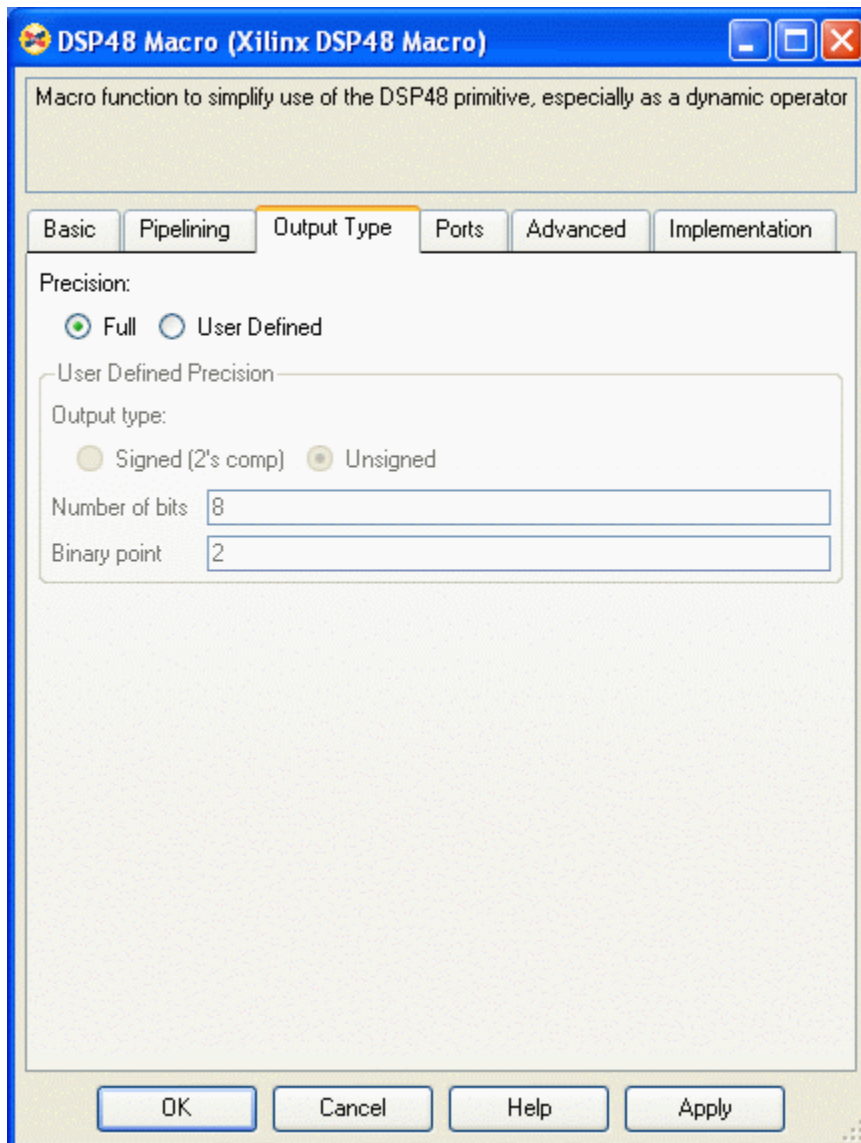


Figure 3: DSP48 Macro Output Type Tab

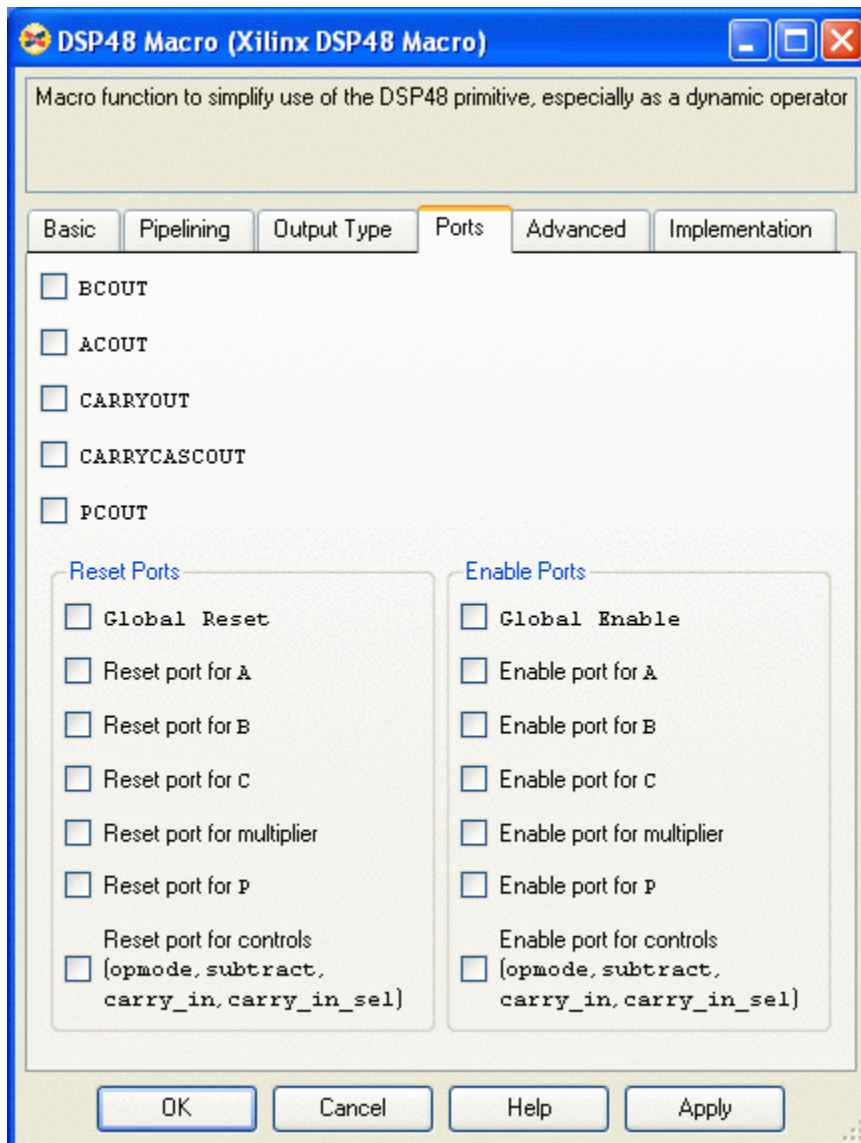


Figure 4: DSP48 Macro Ports Tab

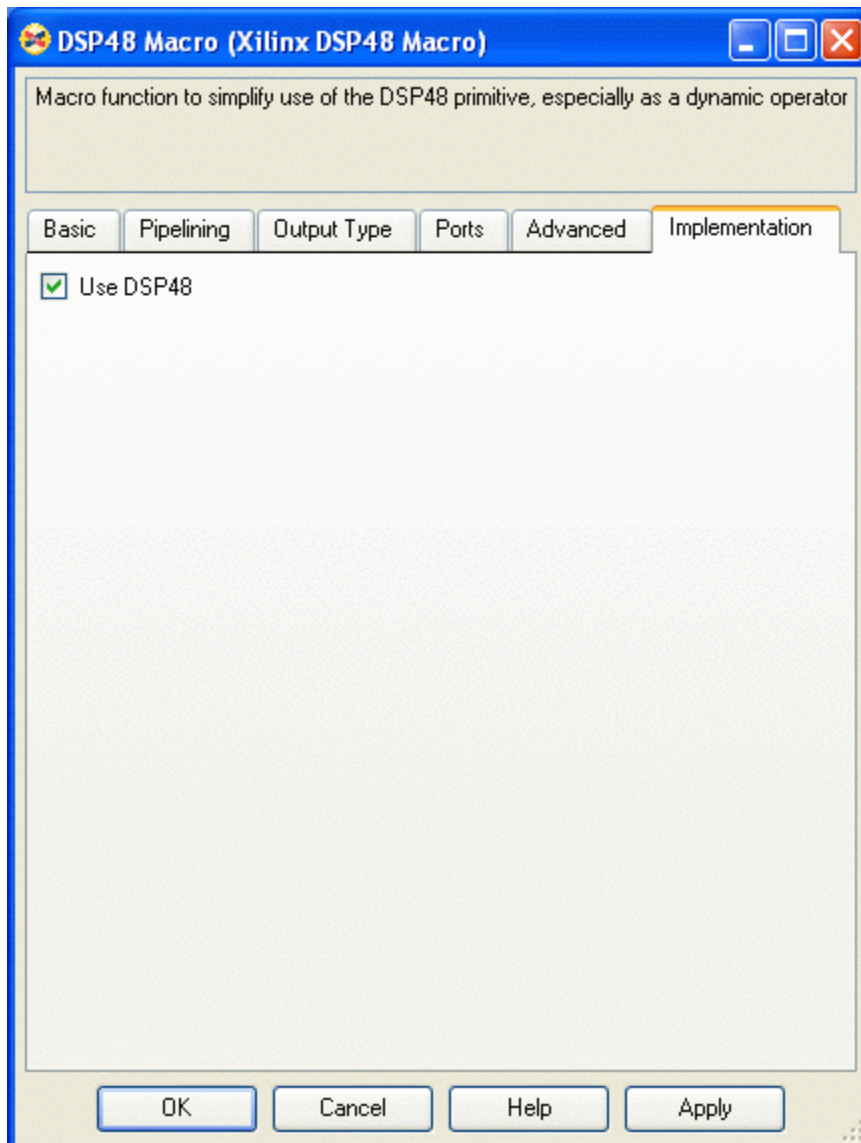


Figure 5: DSP48 Macro Implementation Tab

Parameters specific to the DSP48 Macro are:

- **Inputs to Port A (Basic):** This field specifies symbolic port identifiers or operands appearing in the Instructions field as connected to port A or port A:B on the DSP48.
- **Inputs to port B (Basic):** This field specifies symbolic port identifiers or operands appearing in the Instructions field as connected to port B.
- **Inputs to port C (Basic):** This field specifies symbolic port identifiers or operands appearing in the Instructions field as connected to port C.
- **Instructions (Basic):** This field specifies instructions for the Macro. Refer to the section on Entering Opmodes in the DSP48 Macro.
- **Pipeline Options (Pipelining):** This field specifies the pipelining options on the DSP48 and latency on the data presented to each port of the DSP48. Available options include 'External Registers', 'No External Registers' and 'Custom'. When 'External Registers' is selected multiplexer outputs (underneath DSP48 Macro) are registered (this allows high speed operation). If the DSP48 Macro configures the DSP48 as an adder only (inferred from the operations entered in the instructions field), then the latency is 3 else the latency is 4. When 'No External Registers' is selected, multiplexer outputs are not registered and the latency of the DSP48 Macro becomes

- two. When 'Custom' is selected all register instances inside and outside of the DSP48 are inferred from the Custom Pipeline Options. If the Instructions require the use of DSP48 as adder and multiplier then it must be configured to use Custom Pipeline Option.
- **Custom Pipeline Options (Pipelining):** This group of controls is active only when Pipeline Options is set to Custom. Provides individual control for instancing DSP48 and multiplexer registers.
 - **Custom Pipeline Options (Pipelining):** This field enables specifying Custom Pipeline Options as an array of integers
 - **Use DSP48 (Implementation):** This field specifies if the XtremeDSP Slice on Virtex 4 is targeted. If unchecked, a synthesizable model of DSP48 is used that can be used in devices other than Virtex4.

The Ports tab on the block GUI consists of controls to expose the BCOUT, PCOUT, CARRYCASCOUT, CARRYOUT, ACOUT and the various DSP48 rst and enable ports.

Entering Opcodes in DSP48 Macro Block

The DSP48 is capable of performing different arithmetic operations on input data depending on the input to its opcode port; this capability enables the DSP48 to operate like a dynamic operator unit. The DSP48 Macro simplifies using the DSP48 as a dynamic operator unit. It orders multiple operands and opcodes with multiplexers and appropriately aligns signals on the data ports. The ordering of operands and opcode execution is determined by the order of opcodes entered in the Instructions field. The Instructions field must contain at least one opcode and a maximum of eight opcodes. This section details all the issues involved with entering opcodes in the Instructions field of the DSP48 Macro.

Opcode Format

A newline character is used to separate two different opcodes. Each opcode must strictly adhere to the rules listed below:

- Each opcode is an assignment to P and must begin with 'P='
- The expression following the assignment operator('=') must be entirely made up of +/~/* operators and symbolic port identifiers (see Operand Format) for operands.
- Only opcodes that can be implemented on the DSP48 are legal. A list of opcodes supported on the DSP48 Macro is provided in Table 2.

In Figure 1, an example DSP48 Macro block mask is shown with four legal opcodes entered in the Instructions field.

Operand Format

Each operand (symbolic port identifier) used in an opcode must follow the rules listed below:

- Each symbolic port identifier must begin with an alphabet[a-z,A-Z] and can be followed by any number of alphanumeric characters or underscore('_').
- The symbolic port identifiers must not match any of the reserved port identifiers listed in Table 1 irrespective of case
- Each of the symbolic port identifiers must be listed once and only once in the Inputs to Port A, Port B, or Port C fields. Multiple symbolic port identifiers in the same list must be separated using a space or ';'.

In Figure 1, a0, a1, b0, c0 and c1 are the symbolic port identifiers. Examples of legal symbolic port identifiers/operands are a1, signal_1, delayed_signal etc. Examples of illegal symbolic port identifiers include Cin, _port1, delay\$, 12signal etc.

Reserved Port Identifier	Port Type	Usage
PCIN	Input. Connected to port PCIN on the DSP48	This port appears depending on the opcode used. Refer to Table 2, Opcodes 0x10-0x1f use the PCIN Input. The PCIN port must be connected to the

		PCOUT port of another DSP48 block/DSP48 Macro block.
BCIN	Input. Connected to port BCIN on the DSP48	This port appears if in any of the opmodes listed in Table 2, B(not A:B) is replaced with BCIN. Must be connected to the BCOUT port of another DSP48 block/DSP48 Macro block.
PCIN>>17	Input. Connected to port PCIN on the DSP48	Refer to Table 2. Opmodes 0x50-0x5f use this port identifier. PCIN, is right shifted by 17 and input to the DSP48 adder through DSP48's z multiplexer.
CIN	Input. Connected to port carry_in on the DSP48	This port appears if the opmode contains Cin. Refer to Table 2. Optional on all opmodes except 0x00.
PCOUT	Output. Connected to port PCOUT on the DSP48	This port appears if 'Provide PCOUT' on the block mask is selected.
BCOUT	Output. Connected to port BCOUT on the DSP48	This port appears if 'Provide BCOUT' on the block mask is selected.
RST	Input. Connected to rst on the DSP48 as well as all registers' reset	This port appears if 'Provide Reset' on the block mask is selected.
EN	Input. Connected to en on the DSP48 as well as all registers' enable	This port appears if 'Provide Enable' on the block mask is selected.
Sel	Input	Appears only when more than one instruction is specified in the Instructions field. Used to select an opmode from the list of opmodes in the Instructions field.
P	Output	Always present.
P>>17	-	Refer to Table 2. Opmodes 0x60-0x6f. P, right shifted by 17 is input to the DSP48 adder through the DSP48's z multiplexer.

Table 1 Reserved Port Identifiers

Opmode Selection

More than one opmode specified in the Instructions field necessitates opmode selection. This is achieved through the use of the 'Sel' port that appears when there are multiple opmodes in the Instructions field. The 'Sel' port is connected to multiplexers instantiated underneath the mask as shown in **Error! Reference source not found.**; any signal connected to the Sel port must be of the appropriate data type. The value of the Sel signal for each opmode listed in the Instructions field corresponds to the number adjacent to each opmode in the grayed-out area to the left of the Instructions field.

Using Reserved Identifiers

There are two categories of reserved identifiers. Reserved identifiers that manifest as ports on the DSP48 Macro block and reserved identifiers that do not. Descriptions and usage of each of the reserved word identifiers is listed in Table 1. Example of using PCIN and BCIN reserved words is depicted in Figure 6.

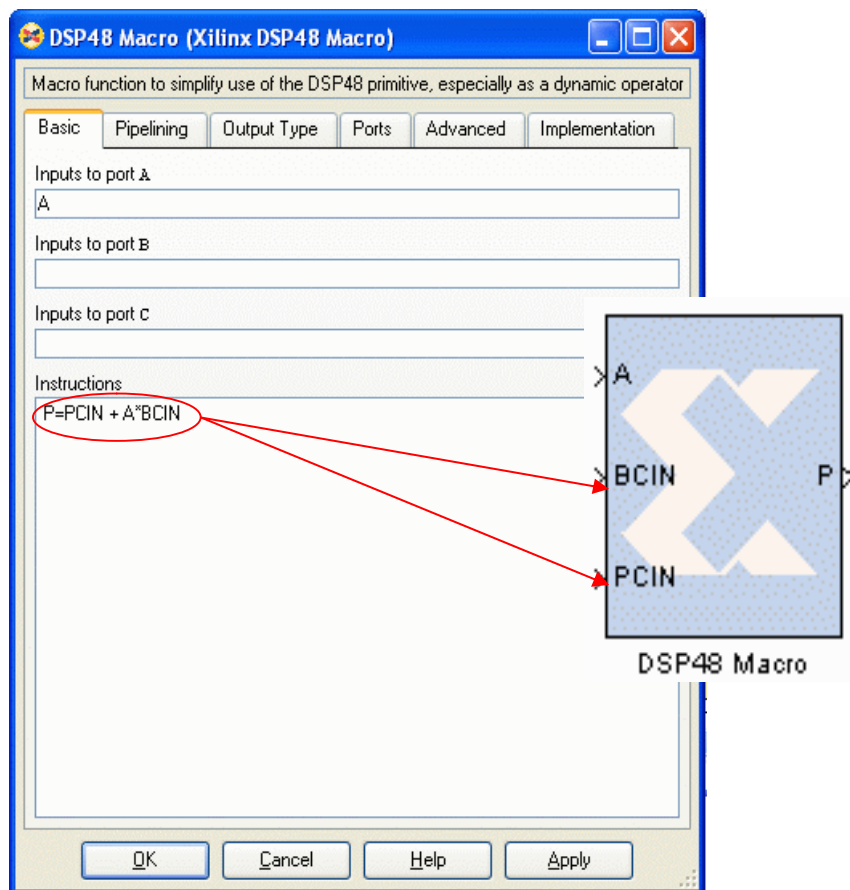


Figure 6 Configuring DSP48 Macro to use BCIN and PCIN ports

Mode Selection

The DSP48 Macro can be operated in two modes: Adder Mode and Multiplier Mode. Mode selection depends on the DSP48 Macro opmodes used; the opmodes supported by each of the modes is listed in Table 2. When A and B ports are routed as inputs to the DSP48's adder, they are concatenated as one signed 36-bit input (refer to the DSP48 documentation). The DSP48's multiplier interprets the ports as two disjoint signed 2's complement 18-bit inputs.

Hex Opmode	Binary Opmode	DSP48 Macro Pseudo Opmode	DSP48 Macro Mode
0x00	0000000	P=Cin P=+Cin P=-Cin	----
0x02	0000010	P=P+Cin P=-P-Cin	----
0x03	0000011	P = A:B + Cin	Adder
0x05	0000101	P = A*B + Cin P = -A*B - Cin	Multiplier
0x0c	0001100	P=C+Cin P=-C-Cin	----

0x0e	0001110	$P = +C + P + Cin$ $P = -C - P - Cin$	----
0x0f	0001111	$P = \mathbf{A:B} + C + Cin$ $P = -\mathbf{A:B} - C - Cin$	Adder
0x10	0010000	$P = PCIN + Cin$ $P = PCIN - Cin$	----
0x12	0010010	$P = PCIN + P + Cin$ $P = PCIN - P - Cin$	----
0x13	0010011	$P = PCIN + \mathbf{A:B} + Cin$ $P = PCIN - \mathbf{A:B} - Cin$	Adder
0x15	0010101	$P = PCIN + A*B + Cin$ $P = PCIN - A*B - Cin$	Multiplier
0x1c	0011100	$P = PCIN + C + Cin$ $P = PCIN - C - Cin$	----
0x1e	0011110	$P = PCIN + C + P + Cin$ $P = PCIN - P - C - Cin$	----
0x1f	0011111	$P = PCIN + \mathbf{A:B} + C + Cin$ $P = PCIN - \mathbf{A:B} - C - Cin$	Adder
0x20	0100000	$P = P - Cin$	----
0x22	0100010	$P = P + P + Cin$ $P = P - P - Cin$	----
0x23	0100011	$P = P - \mathbf{A:B} - Cin$ $P = P + \mathbf{A:B} + Cin$	Adder
0x25	0100101	$P = P + A*B + Cin$	Multiplier
0x2c	0101100	$P = P + C + Cin$ $P = P - C - Cin$	----
0x2e	0101110	$P = P + C + P + Cin$ $P = P - C - P - Cin$	----
0x2f	0101111	$P = P + \mathbf{A:B} + C + Cin$ $P = P - \mathbf{A:B} - C - Cin$	Adder
0x30	0110000	$P = C - Cin$	----
0x32	0110010	$P = C - P - Cin$	----
0x33	0110010	$P = C - \mathbf{A:B} - Cin$	Adder
0x35	0110101	$P = C - A*B - Cin$	Multiplier
0x3c	0111100	$P = C + C + Cin$ $P = C - C - Cin$	----
0x3e	0111110	$P = C + C + P + Cin$ $P = C - C - P - Cin$	----
0x50	1010000	$P = PCIN \gg 17 + Cin$, $P = PCIN \gg 17 Cin$	----
0x52	1010010	$P = PCIN \gg 17 + P + Cin$ $P = PCIN \gg 17 - P - Cin$	----
0x53	1010011	$P = PCIN \gg 17 + \mathbf{A:B} + Cin$ $P = PCIN \gg 17 - \mathbf{A:B} - Cin$	Adder
0x55	1010101	$P = PCIN \gg 17 + A*B + Cin$ $P = PCIN \gg 17 - A*B - Cin$	Multiplier
0x5c	1011100	$P = PCIN \gg 17 + C + Cin$ $P = PCIN \gg 17 - C - Cin$	----
0x5e	1011110	$P = PCIN \gg 17 + P + C + Cin$ $P = PCIN \gg 17 - P - C - Cin$	----
0x5f	1011111	$P = PCIN \gg 17 + C + \mathbf{A:B} + Cin$ $P = PCIN \gg 17 - C - \mathbf{A:B} - Cin$	Adder

0x60	1010000	P=P>>17+Cin P=P>>17-Cin	----
0x62	1010010	P=P>>17+P+Cin P=P>>17-P-Cin	----
0x63	1010011	P=P>>17+A:B+Cin P=P>>17-A:B-Cin	Adder
0x65	1010101	P=P>>17+A*B+Cin P=P>>17-A*B-Cin	Multiplier
0x6c	1011100	P=P>>17+C+Cin P=P>>17-C-Cin	----
0x6e	1011110	P=P>>17+P+C+Cin P=P>>17-P-C-Cin	----
0x6f	1011111	P=P>>17+C+A:B+Cin P=P>>17-C-A:B-Cin	Adder

Table 2 Opmodes DSP48 block. Cin is optional in all the Opmodes. **A:B** refers to all the symbolic port identifiers in 'Inputs to Port A' field of DSP48 Macro block mask supplying inputs to the Adder of DSP48 block. Symbols A, B, and C refer to symbolic identifiers in Inputs to Port A, Port B and Port C fields respectively. All other symbols are reserved (refer to Table 1 for more details).

Entering Pipeline Options and Editing Custom Pipeline Options

Since the data paths for the A, B and C ports are different and can have a different number of registers, time-alignment issues arise. Control signals also suffer from the same issue. This makes the pipeline model extremely important. There are three pipeline options available in the DSP48 Macro block mask. These include 'External Registers', 'No External Registers' and 'Custom'.

External Registers

This option aligns all the control and data signals by also using additional registers external to the DSP48 block. These external registers are required to register the output of the multiplexers to ensure high-speed operation. If all the opmodes entered into the DSP48 Macro instructions field are such that, they require the use of multiplier, then the latency on the DSP48 Macro is 4. If none of the instructions on the DSP48 Macro require the use of the multiplier, the latency on the DSP48 Macro is 3.

No External Registers

This option aligns all the control and data signals without using registers external to the DSP48 block. The MREG is not selected in this mode. The latency of the DSP48 Macro is 2.

Custom

This option gives the user control over instantiating each register of the DSP48 Macro block. When this option is selected the 'Custom Pipeline Options' group of controls becomes active and each of the individual registers can be selected. When the DSP48 Macro contains instructions that require using the multiplier in the DSP48 and the Adder with A:B as one of the inputs, Custom pipeline is the only legal option.

DSP48 Macro Limitations

Though the DSP48 Macro eases the use of the DSP 48 block it is not without limitations:

- It does not support the DSP48's rounding features
- It supports carry-in only from fabric
- It does not support all input data types. Input data types that exceed the data type restrictions of a single DSP48 are not supported currently. For example if, after alignment of inputs, the input to Port A of DSP48 exceeds 18bits then it will result in an error

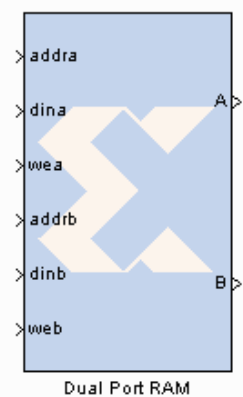
See Also

The following documents are provided as part of the System Generator User's guide and give valuable insights into using and understanding the DSP48 Macro block:

DSP48
Generating Block Diagrams Programmatically

Dual Port RAM

This block is listed in the following Xilinx Blockset libraries: Control Logic, Memory, and Index.



The Xilinx Dual Port RAM block implements a random access memory (RAM). Dual ports enable simultaneous access to the memory space at different sample rates using multiple data widths.

Block Interface

The block has two independent sets of ports for simultaneous reading and writing. Independent address, data, and write enable ports allow shared access to a single memory space. Each port set has one output port and three input ports for address, input data, and write enable. The result of simultaneous access to both ports is as described below:

- If both ports read simultaneously from the same memory cell, the read operation is successful.
- If both ports try to write simultaneously to the same memory cell, both outputs are marked as indeterminate.
- If one port writes and the other reads from the same memory cell, the write operation succeeds and the read operation results in an indeterminate state except in the case when the write port is in the Read before Write mode.

The Dual Port RAM block also supports various Form Factors (FF). Form factor is defined as:

$$FF = W_B / W_A \text{ where } W_B \text{ is data width of Port B and } W_A \text{ is Data Width of Port A.}$$

The Dual port RAM block allows FF of 1, 2, 4, 8, 16 for Virtex and 1, 2, 4, 8, 16 or 32 for Virtex-II device families, provided that:

$$\text{Mod} [(D_A \times W_A) , W_B] = 0 \text{ for a given FF where } D_A : \text{Depth specified for Port A}$$

The Depth of port B (D_B) is inferred from the specified form factor as follows:

$$D_B = D_A / FF.$$

The data input ports on Port A and B can have different arithmetic type and binary point position for a form factor of 1. For form factors greater than 1, the data input ports on Port A and Port B should have an unsigned arithmetic type with binary point at 0. The output ports, labeled A and B, have the same types as the corresponding input data ports.

The location in the memory block can be accessed for reading or writing by providing the valid address on each individual address port. A valid address is an unsigned integer from 0 to $d-1$, where d denotes the RAM depth (number of words in the RAM) for the particular port. An attempt to read past the end of the memory is caught as an error in simulation. The initial RAM contents can be specified through a block parameter. Each write enable port must be a boolean value. When the WE port is 1, the value on the data input is written to the location indicated by the address line.

The output during a write operation depends on the write mode. When the WE is 0, the output port has

the value at the location specified by the address line. During a write operation (WE asserted), the data presented on the input data port is stored in memory at the location selected by the port's address input. During a write cycle, the user can configure the behavior of the data out ports A/B to one of the following choices:

- Read After Write
- Read Before Write
- No Read On Write

The write modes can be described with the help of the figure below. In the figure, the memory has been set to an initial value of 5 and the address bit is specified as 4. When using No Read On Write mode, the output is unaffected by the address line and the output is the same as the last output when the WE was 0. For the other two modes, the output is obtained from the location specified by the address line, and hence is the value of the location being written to. This means that the output can be the old value which corresponds to Read After Write.

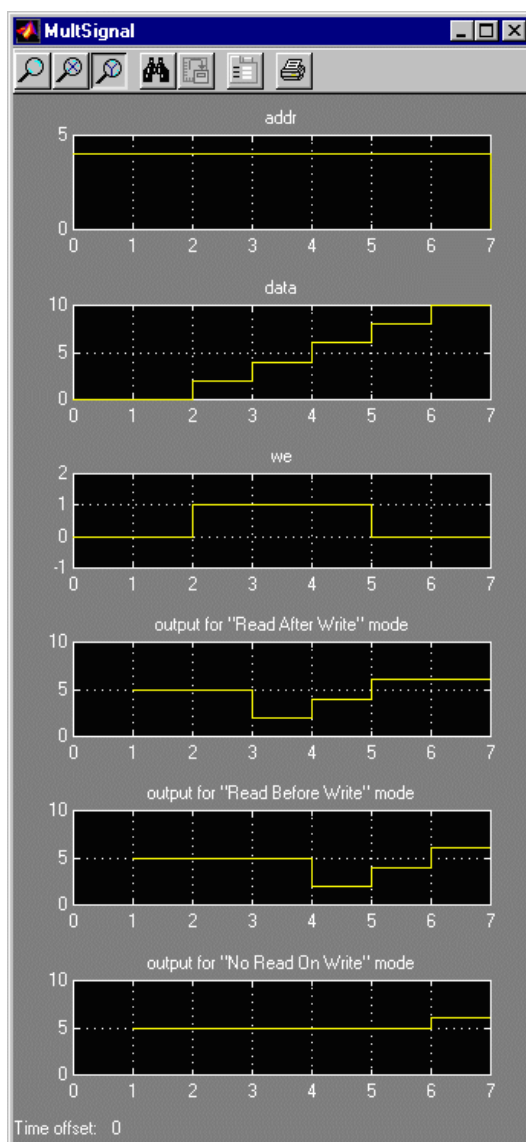


Figure 1: Illustration of write modes

Virtex, Virtex-E and Spartan-II families support only Read After Write. Virtex-II, Virtex-II Pro,

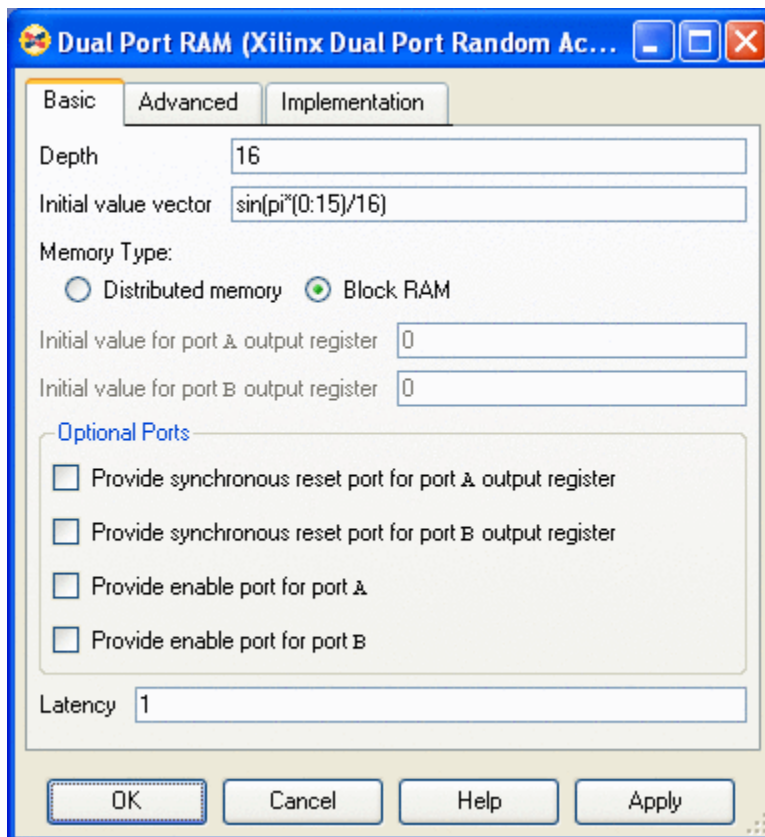
Spartan-3 and Virtex-4 support all modes.

When implementing dual port RAM blocks on Virtex-4 devices, maximum timing performance is possible if the following conditions are satisfied:

- Provide synchronous reset port for port A output register is unchecked.
- Provide synchronous reset port for port B output register is unchecked.
- Depth is less than 16,384.
- Latency is set to 2 or higher.

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.



Dual Port RAM (Xilinx Dual Port Random Ac...)

Basic | Advanced | Implementation

Depth: 16

Initial value vector: $\sin(\pi \cdot [0:15]/16)$

Memory Type:

☐ Distributed memory ☒ Block RAM

Initial value for port A output register: 0

Initial value for port B output register: 0

Optional Ports:

☐ Provide synchronous reset port for port A output register

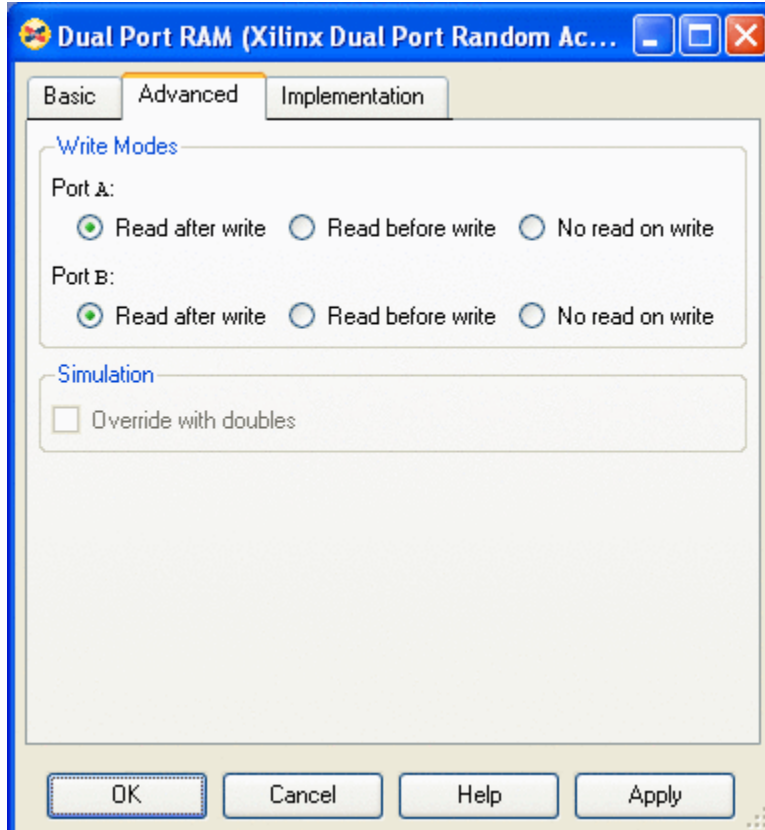
☐ Provide synchronous reset port for port B output register

☐ Provide enable port for port A

☐ Provide enable port for port B

Latency: 1

OK Cancel Help Apply



Dual Port RAM (Xilinx Dual Port Random Ac...)

Basic | Advanced | Implementation

Write Modes

Port A:

☒ Read after write ☐ Read before write ☐ No read on write

Port B:

☒ Read after write ☐ Read before write ☐ No read on write

Simulation

☐ Override with doubles

OK Cancel Help Apply

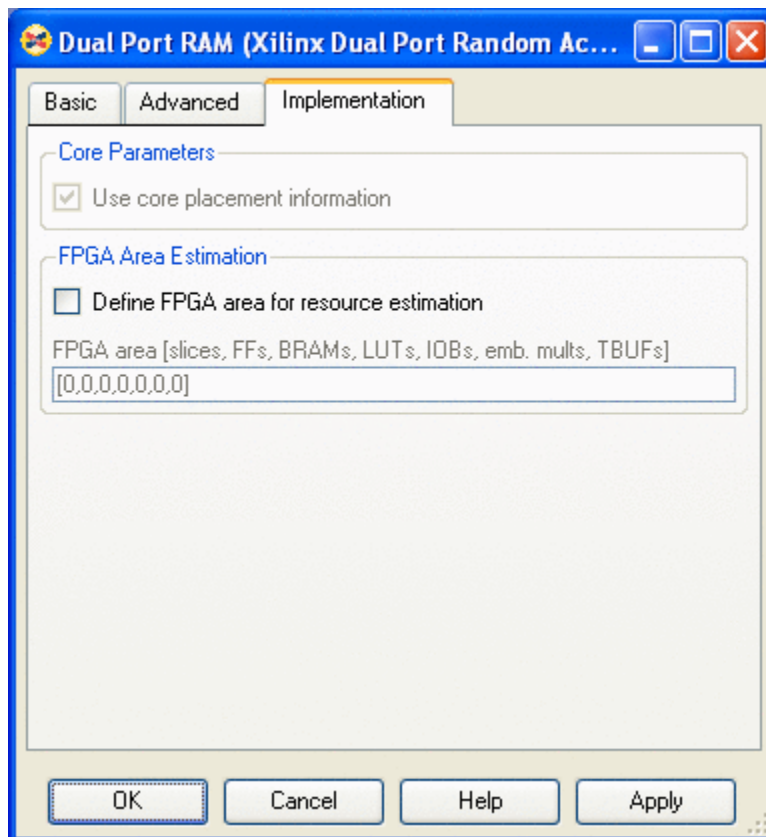


Figure 2: Dual Port RAM block parameters dialog box

Parameters specific to the block are:

- **Depth:** specifies the number of words in the memory for Port A, which must be a positive integer. The Port B depth is inferred from the form factor specified by the input data widths.
- **Initial value vector:** specifies the initial memory contents. The size and precision of the elements of the initial value vector are based on the data format specified for Port A. When the vector is longer than the RAM, the vector's trailing elements are discarded. When the RAM is longer than the vector, the RAM's trailing words are set to zero. The initial value vector is saturated and rounded according to the precision specified on the data port A of RAM.
- **Memory Type:** option to select between block and distributed RAM. The distributed dual port RAM is always set to use port A in Read Before Write mode and port B in read-only mode.
- **Write Modes (A/B Ports):** specifies the memory behavior to be Read Before Write, Read After Write, or No Read On Write. There are device specific restrictions on the applicability of these modes.
- **Provide synchronous reset port for port A output register:** when selected, allows access to the reset port available on the port A output register of the Block RAM. The reset port is available only when the latency of the Block RAM is set to 1.
- **Initial value for port A output Register:** specifies the initial value for port A output register. The initial value is saturated and rounded according to the precision specified on the data port A of RAM. The option to set initial value is available only for Virtex-II, Virtex-II Pro, Spartan-3, and Virtex-4 devices.
- **Provide synchronous reset port for port B output register:** when selected, allows access to the reset port available on the port B output register of the Block RAM. The reset port is available only when the latency of the Block RAM is set to 1.
- **Initial value for port B output register:** specifies the initial value for port B output register. The initial value is saturated and rounded according to the precision specified on

- the data port B of RAM. The option to set initial value is available only for Virtex-II, Virtex-II Pro, Spartan-3, and Virtex-4 devices.
- **Provide enable port for port A:** when selected, allows access to the enable port for port A. The enable port is available only when the latency of the block is greater than or equal to 1.
 - **Provide enable port for port B:** when selected, allows access to the enable port for port B. The enable port is available only when the latency of the block is greater than or equal to 1.

Other parameters used by this block are explained in the Common Parameters section at the beginning of this chapter.

Xilinx LogiCORE

The block always uses a Xilinx LogiCORE: Dual Port Block Memory or Distributed Memory.

For the dual port block memory, the address width must be equal to $\text{ceil}(\log_2(d))$ where d denotes the memory depth. The maximum width of data words in the block memory depends on the depth specified; the maximum depth depends on the device family targeted. Figure 3 and Figure 4 provide the maximum data word width for a given block memory depth.

Depth	Width
2 to 2048	256
2049 to 4096	192
4097 to 8192	96
8193 to 16K	48
16K+1 to 32K	24
32K+1 to 64K	12
64K+1 to 128K	6
128K+1 to 256K	3

Figure 3: Maximum Width for Various Depth Ranges (Virtex/Virtex-E/Spartan-3)

Depth	Width
2 to 8192	256
8193 to 16K	192
16K+1 to 32K	96
32K+1 to 64K	48
64K+1 to 128K	24
128K+1 to 256K	12
256K+1 to 512K	6
512K+1 to 1024K	3

Figure 4: Width for Various Depth Ranges (Virtex-II/Virtex-II Pro/Virtex-4)

When the distributed memory parameter is selected, LogiCORE Distributed Memory is used. The depth must be between 16 and 65536, inclusive for Virtex-II, Virtex-II Pro, Spartan-3, and Virtex-4; depth must be between 16 to 4096, inclusive for the other FPGA families. The word width must be between 1 and 1024, inclusive.

EDK Processor

This block is listed in the following Xilinx Blockset libraries: *Index, Control*.



The EDK Processor block allows user logic developed in System Generator to be attached to embedded processor systems created using the Xilinx Embedded Development Kit (EDK) and the Xilinx Platform Studio tool suite. The EDK Processor block supports two design flows: *EDK Pcore generation* and *HDL netlisting*. In *HDL netlisting* flow, the embedded processor systems created using the EDK tool are imported into System Generator models. In *EDK Pcore generation* flow, the System Generator models are exported as a pcore, which can be later imported into EDK projects and attached to embedded processors.

Memory Map Interface

The EDK Processor block automatically generates a Shared-Memory based memory map interface for the embedded processor and the user logic developed using System Generator to communicate with each other. C device drivers are also automatically generated by EDK Processor block in order for the embedded processors to access the attached shared memories.

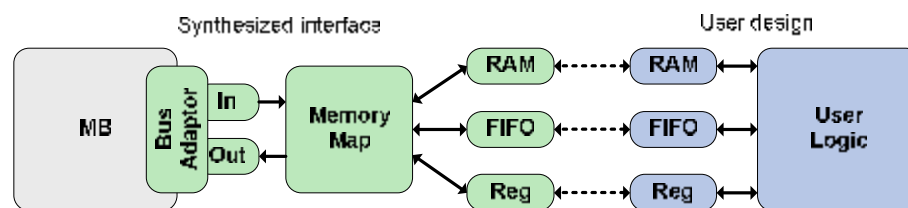


Figure 1: Memory map interface

Figure 1 shows the memory map interface generated by the EDK Processor block. The user logic developed in System Generator is connected to a set of shared memories. These shared memories can be added to the EDK Processor block through the block dialog box described below. The EDK Processor block automatically generates the other half of the shared memories and a memory map interface that connects the shared memories to the MicroBlaze processor through a pair of FSL (Fast Simplex Link) buses. C device drivers are also automatically generated so that the MicroBlaze processor can get access to these shared-memories, by their names or their locations on the memory map interface.

The memory map interface is generated by the EDK Processor block in either *EDK Pcore generation* flow or *HDL netlisting* flow. In *EDK Pcore generation* flow, only the hardware to the right of the Bus Adaptor is netlisted into the exported pcore. In *HDL netlisting* flow, all the hardware shown in Figure 1 (including the MicroBlaze processor, the memory map interface, the shared memories, and the user logic) is netlisted together, just like other System Generator designs.

Refer to Hardware and Software Co-Design in System Generator for more details about the design and simulation techniques offered by the EDK Processor block.

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the EDK Processor block in your Simulink model.

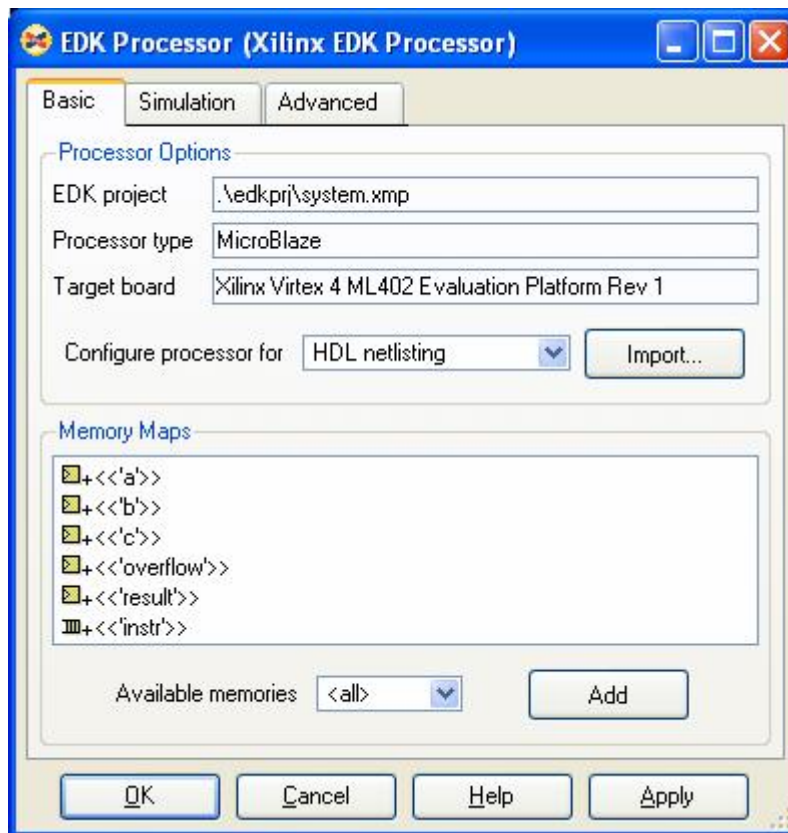


Figure 2: Basic Tab

Parameters on the Basic tab as shown in Figure 2 are:

- **EDK project:** Name of the imported EDK project file (.xmp file). Right-click on the text box reveals a pop-up menu, which can invoke the EDK Import Wizard. The EDK Import Wizard will read in an EDK project file, and then modify and import the EDK project.
- **Processor type:** The type of processor that is detected when importing the EDK project.
- **Target board:** The hardware board targetted by the EDK project.
- **Configure processor for:** The EDK Processor block can be configured for *EDK Pcore generation* or *HDL netlisting*. The EDK Import Wizard runs automatically when HDL netlisting is chosen and the EDK project field is empty.
- **Import :** Launch the EDK Import Wizard.
- **Memory Maps:** contains a reeview that shows the shared memories associated with the processor. Right-click on the tree view items reveals a menu of possible operations on the shared memories: configure, delete, or resynchronize the shared memories, refresh the tree view. Resynchronizing shared memories helps to keep the shared memories used by the user logic in consistent with the shared memories automatically generated by the EDK Processor block.

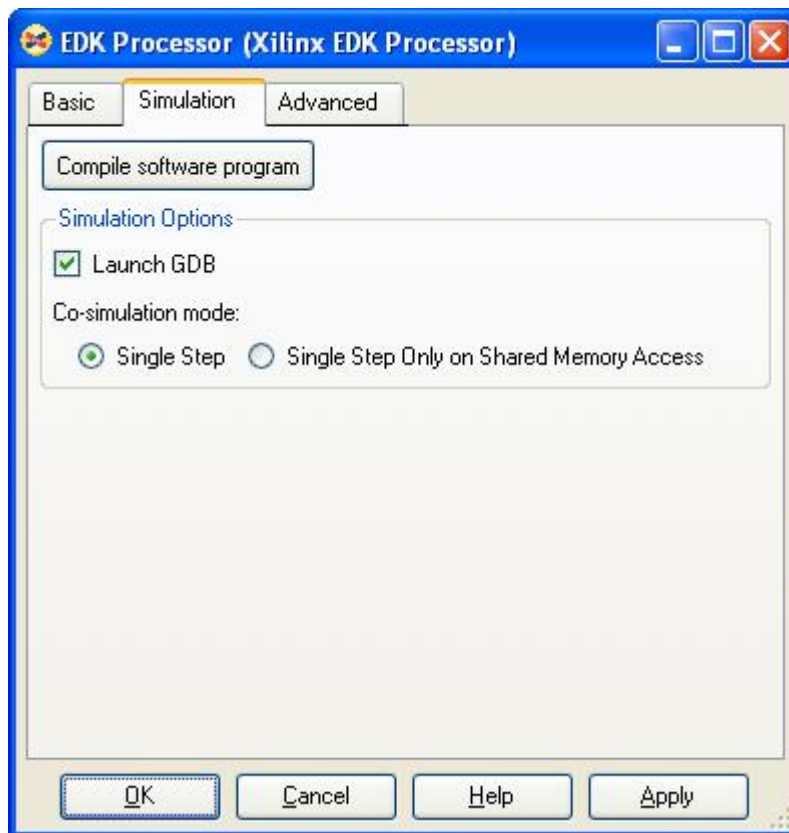


Figure 3: Simulation Tab

Parameters on the Simulation tab as shown in Figure 3 are:

- **Compile software program:** Launch the XPS tool suite to compile the active software programs in the imported EDK project which are selected to initialize the memory of the embedded processors.
- **Launch GDB:** Launch GDB for debugging upon simulation starts. The user can examine the status of the embedded processors and control the progress of Simulation simulation through the GDB GUI.
- **Co-simulation mode:** Run software based simulation in *Single Step* mode or in *Single Step Only on Shared Memory Access* mode. In *Single Step* mode, simulation of the imported EDK project is synchronized with the simulation of other Simulink blocks on a cycle-accurate basis. In *Single Step Only on Shared Memory Access* mode, simulation of the imported EDK project is synchronized with the simulation of other Simulink blocks only where there is data to be transmitted over the shared memories on the shared memory map. Refer to Software Based Simulation Modes for more information about the two co-simulation modes in software based simulation.

Note that the parameters on the Simulation tab are disabled in *EDK Pcore generation* flow or on failing to compile software based simulation models in *HDL netlisting* flow.

Known Issues

- Only one EDK Processor block per design is supported.
- The software based simulation only supports a subset of the MicroBlaze hardware peripherals. Please refer to the section EDK tool documents regarding to Virtual Platform for a list of supported hardware peripherals.

- Only one MicroBlaze processor per design is supported. Use of multiple MicroBlaze processors per design and the embedded PowerPC processor are not supported.

Online Documentation for the MicroBlaze Processor

More information for the MicroBlaze can be found at

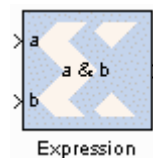
http://www.xilinx.com/xlnx/xebiz/designResources/ip_product_details.jsp?key=micro_blaze

See Also

Hardware Software Co-Design Technologies in System Generator, Various Hardware-Software Simulation Techniques, Shared-Memories

Expression

This block is listed in the following Xilinx Blockset libraries: *Basic Elements, Control Logic, Math, and Index.*



The Xilinx Expression block performs a bitwise logical expression. The expression is specified with operators described in the table below. The number of input ports is inferred from the expression.

The input port labels are identified from the expression, and the block is subsequently labeled accordingly. For example, the expression: $\sim((a1 \mid a2) \& (b1 \wedge b2))$ would result in a block with 4 input ports labeled 'a1', 'a2', 'b1', and 'b2'.

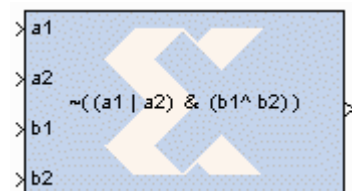


Figure 1: Expression block with ports inferred from custom expression

The expression will be parsed and an equivalent statement will be written in VHDL(or Verilog). Shown below, in decreasing order of precedence, are the operators that can be used in the Expression block.

Operator	Symbol
Precedence	()
NOT	~
AND	&
OR	
XOR	^

Figure 2: Operators that can be used in Expression block

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

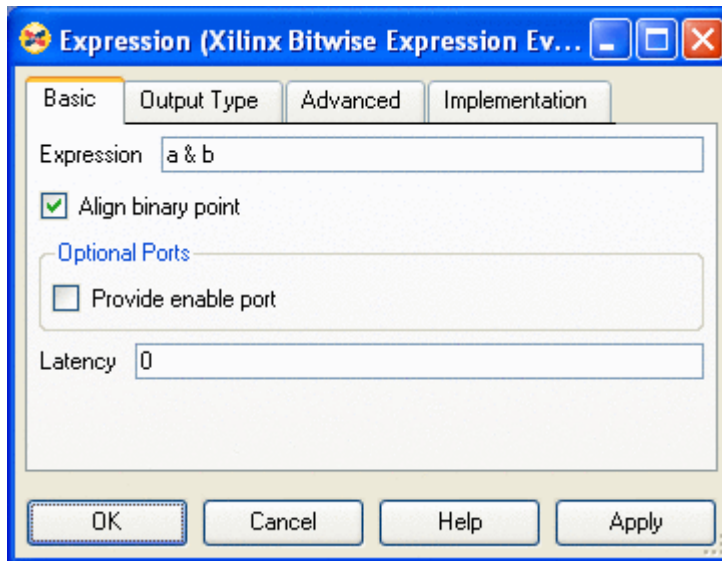


Figure 3: Expression block Basic Tab

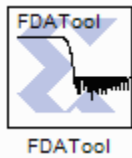
Parameters specific to the block are:

- **Expression (Basic):** Bitwise logical expression.
- **Align Binary Point (Basic):** specifies that the block must align binary points automatically. If not selected, all inputs must have the same binary point position.

Other parameters used by this block are explained in the Common Parameters section at the beginning of this chapter.

FDATool

This block is listed in the following Xilinx Blockset libraries: *DSP, Tools, and Index*.



The Xilinx FDATool block provides an interface to the FDATool software available as part of the MATLAB Signal Processing Toolbox. The block does not function properly and should not be used if the Signal Processing Toolbox is not installed.

This block provides a means of defining an FDATool object and storing it as part of a System Generator model. FDATool provides a powerful means for defining digital filters with a graphical user interface.

Example of Use

Copy an FDATool block into a subsystem where you would like to define a filter. Double-clicking the block icon opens up an FDATool session and graphical user interface. The filter is stored in a data structure internal to the FDATool interface block, and the coefficients can be extracted using MATLAB helper functions provided as part of System Generator. The function call `xlfd_a_numerator('fdablk')` returns the numerator of the transfer function (e.g., the impulse response of a finite impulse response filter) of the FDATool block named 'fdablk'. Similarly, the helper function `xlfd_a_denominator('fdablk')` retrieves the denominator for a non-FIR filter.

A typical use of the FDATool block is as a companion to an FIR filter block, where the `Coefficients` field of the filter block is set to `xlfd_a_numerator('fdablk')`. An example is shown in the following diagram:

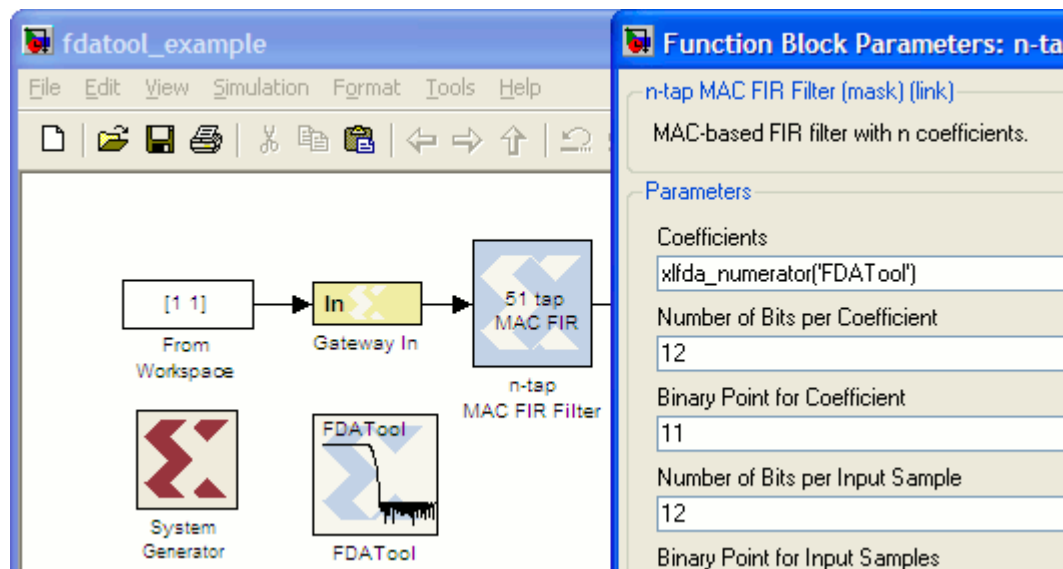


Figure 1: Example of FDATool used to produce coefficients for FIR filter

Note that `xlfd_a_numerator()` can equally well be used to initialize a memory block or a 'coefficient' variable for a masked subsystem containing an FIR filter.

This block does not use any hardware resources.

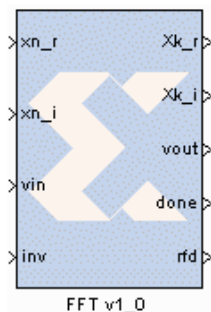
FDATool Interface

Double-clicking the icon in your Simulink model opens up an FDATool session and its graphical user interface. Upon closing the FDATool session, the underlying FDATool object is stored in the `UserData` parameter of the Xilinx FDATool block. Use the `xlfd_a_numerator()` helper function and

`get_param()` to extract information from the object as desired.

FFT v1_0

This block is listed in the following Xilinx Blockset libraries: *DSP and Index*.



The Xilinx FFT v1_0 block is supported only for the Virtex device. For other device family support refer to the FFT v3_1 block. The Xilinx FFT Block computes the Discrete Fourier Transform (DFT) using the radix-4 Cooley-Tukey algorithm, explained below.

Cooley-Tukey algorithm

The N-point DFT of a complex vector $x(n) = [x(0), x(1), \dots, x(N-1)]$, is the vector $X(k) = [X(0), X(1), \dots, X(N-1)]$, where the k-th element:

$$X(k) = \sum_{m=0}^{N-1} x(m) W_N^{mk}$$

for $k=0, 1, \dots, N-1$, where:

$$W_N = e^{(-j)\frac{2\pi}{N}}$$

is a principal N-th root of unity. The FFT block accepts as input a stream of complex data represented as a pair of Xilinx fixed point data and computes successive DFTs of non-overlapping frames of N data samples.

Block Interface

The block interface (inputs and outputs as seen on the FFT icon) is as follows:

Input signals:

- xn_r** real component of input data stream
- xn_i** imaginary component of input data stream
- vin** marks each data input as valid or invalid. Signal can be used to align the start of the FFT to your data frames. See timing diagram below.
- inv** 0 for forward transform, 1 for inverse

Output signals:

- Xk_r** real component of output data stream
- Xk_i** imaginary component of output data stream
- vout** marks the output data as valid or invalid. If any of the N inputs of a frame is marked as invalid, the corresponding output frame will be marked as invalid.
- done** active high on first output sample in a frame
- rfd** active high when block can accept input data

Block Parameters Dialog Box

The FFT block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

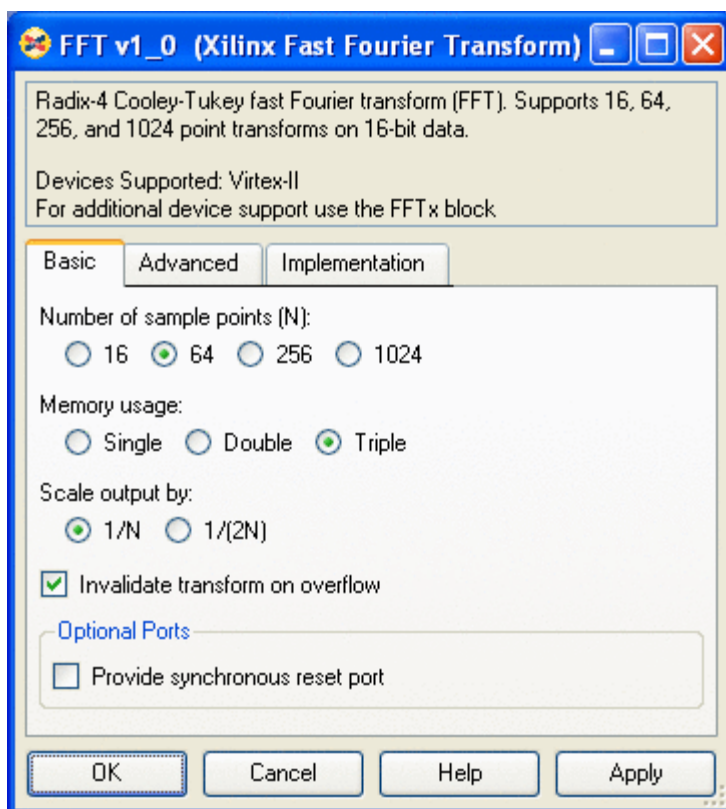


Figure 1: FFT v1_0 block parameters dialog box

Parameters specific to the FFT block are:

- **Number of Sample Points:** transform length, one of 16, 64, 256, or 1024.
- **Memory Usage:** number of memory banks used to compute the transform, one of Single, Double, Triple (not used for 16 point FFTs).
- **Scale Output By:** one of 1/N or 1/(2N).
- **Invalidate Transform on Overflow:** block behavior when internal overflow occurs; you may choose to invalidate the output (if checkbox is selected) or to stop the simulation in the event of an overflow (if checkbox is not selected).

Other parameters used by this block are explained in the Common Parameters section at the beginning of this chapter.

Block Timing

The timing diagram below illustrates the behavior of the FFT block. The diagram indicates the number of sample periods between the taking of input samples and the production of the output samples for a particular frame. (Note that the timing characteristics depend on the number of points in the FFT and the memory usage mode selected. For triple memory configurations, the timing numbers are specified in terms of the output data sample period.)

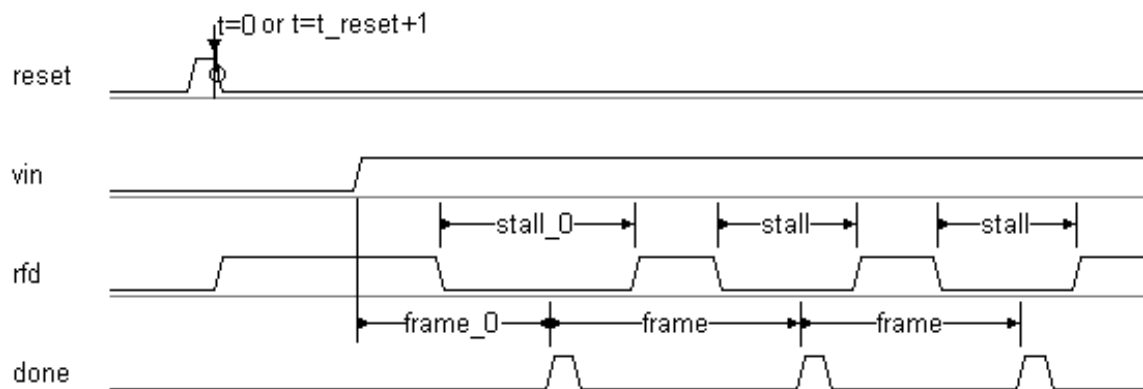


Figure 2: FFT Timing Diagram

	Single Memory	Double Memory	Triple Memory
64-point	stall_0 = 275 stall = 275 frame_0 = 277 frame = 339	stall_0 = 146 stall = 128 frame_0 = 276 frame = 192	stall_0 = 0 stall = 0 frame_0 = 406 frame = 192
256-point	stall_0 = 1074 stall = 1074 frame_0 = 1076 frame = 1330	stall_0 = 789 stall = 768 frame_0 = 1075 frame = 1024	stall_0 = 0 stall = 0 frame_0 = 1589 frame = 768
1024-point	stall_0 = 5170 stall = 5170 frame_0 = 5172 frame = 6194	stall_0 = 4117 stall = 4096 frame_0 = 5171 frame = 5120	stall_0 = 0 stall = 0 frame_0 = 8246 frame = 4096

Figure 3: FFT Timing Characteristics

For 16-point FFTs, the block is always in the "ready for data" state and output frames are delivered continuously. Thus, there are no stall periods ($\text{stall} = \text{stall}_0 = 0$), and the frame variable of the timing diagram defaults to 16 sample periods. There is, however, a pipeline delay (i.e., it takes some time for the first output frame to appear) with $\text{frame}_0 = 84$ sample periods.

Xilinx LogiCORE

The block always uses the Xilinx LogiCORE FFT V1.0. The number of points supported are $N=16, 64, 256$, or 1024 . The $64, 256$, and 1024 point FFTs contain external memories implemented with the LogiCORE Dual Port Block Memory. The number of memory blocks (either 1, 2, or 3) determines the timing characteristics and size of the implementation. The FFT LogiCORE support only 16-bit data, although in simulation, the System Generator FFT block supports other data sizes.

For more information on implementing FFTs for Virtex device, please see the LogiCORE information section for the FFT.

Note: Virtex-II device support has been deprecated for the FFT block. Switch your design to use the FFT v3_1 block instead.

FFT v3_1

This block is listed in the following Xilinx Blockset libraries: *DSP and Index*.



FFT v3_1

Cooley-Tukey algorithm

The N-point DFT of a complex vector $x(n) = [x(0), x(1), \dots, x(N-1)]$, is the vector $X(k) = [X(0), X(1), \dots, X(N-1)]$, where the k-th element is defined as:

$$X(k) = \sum_{m=0}^{N-1} x(m) e^{-jmk 2\pi / N} \text{ for } k=0, 1, \dots, N-1$$

The inverse DFT (IDFT) is computed as,

$$x(m) = \sum_{k=0}^{N-1} X(k) e^{jmk 2\pi / N} \text{ for } m=0, 1, \dots, N-1$$

Block Interface

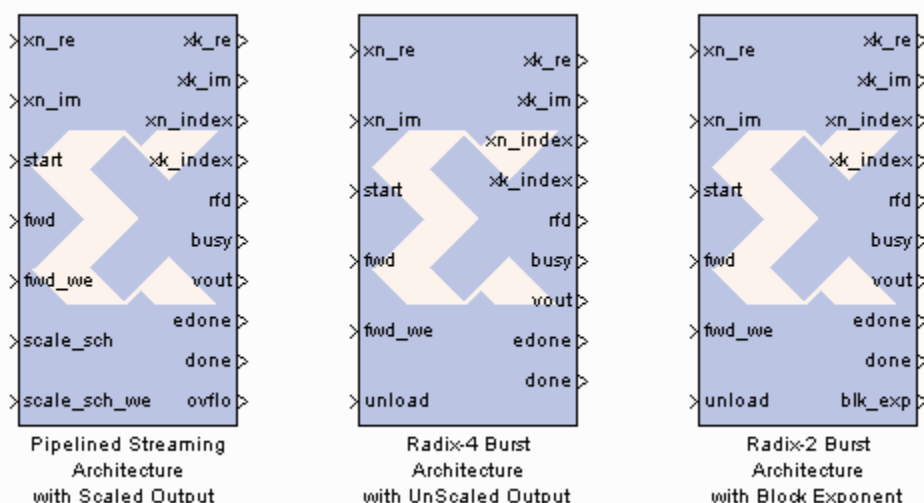


Figure 1: FFT v3_1 Block Interface for different Architectures and Scaling Options

The block interface (inputs and outputs as seen on the FFT icon) for various implementation and scaling modes is as follows:

Input signals:

<code>xn_re</code>	real component of input data stream. The signal driving <code>xn_re</code> can be a signed data type of width S with binary point at $S-1$, where $S = 8, 12, 16, 20$ or 24 .
<code>xn_im</code>	imaginary component of input data stream. The signal driving <code>xn_im</code> can be a signed data type of width S with binary point at $S-1$, where $S = 8, 12, 16, 20$ or 24 .
<code>start</code>	marks the beginning of each data frame. The <code>start</code> signal can be asserted as a pulse to start processing an input data frame or it can be tied to high. The signal driving <code>start</code> must be <code>Bool</code> .
<code>unload</code>	is used to read the output in either natural or bit reversed order. The <code>unload</code> port is available only for implementing the Radix-4 Burst I/O or Radix-2 Minimum Resources architecture. The <code>unload</code> signal is sampled after the block is done processing the input frame. The block outputs data in natural order after the <code>unload</code> signal is asserted high. If the <code>start</code> signal is asserted before the <code>unload</code> signal, the block outputs data in bit reversed order. The signal driving <code>unload</code> must be <code>Bool</code> .
<code>fwd</code>	0 for inverse transform, 1 for forward transform. The signal driving <code>fwd</code> must be <code>Bool</code> .
<code>fwd_we</code>	when asserted, loads the transform type from the input port <code>fwd</code> for the next input data frame. The signal driving <code>fwd_we</code> must be <code>Bool</code> .
<code>scale_sch</code>	provides the scaling schedule to be used for the input data frame. The <code>scale_sch</code> port is available only for Fixed Point Scaled mode. The signal driving <code>scale_sch</code> must be unsigned signal of width S with binary point at 0, where, $S = 2 * \log_2 N$, for Radix-2 Minimum Resources, $2 * \text{ceil}(\log_2 N / 2)$, otherwise.
<code>scale_sch_we</code>	when asserted, loads the scaling schedule from the input port <code>scale_sch</code> for the next input data frame. The <code>scale_sch_we</code> port is available only for Fixed Point Scaled mode. The signal driving <code>scale_sch_we</code> must be <code>Bool</code> .
<code>nfft</code>	provides the point size for the next input data frame. The <code>nfft</code> port is available only when the checkbox for Enable Dynamic Transform Size is selected. The signal driving <code>nfft</code> must be unsigned signal of width S with binary point at 0, where $S = \log_2 N$.

nfft_we when asserted, resets the current operation of the block and loads the point size from the input port **nfft** for the next input data frame. The **nfft_we** port is available only when the checkbox for **Enable Dynamic Transform Size** is selected. The signal driving **nfft_we** must be **Bool**.

Note: Both **xn_re** and **xn_im** signals should have the same data type.

Output signals:

xk_re real component of output data stream. **xk_re** is same as the input **xn_re** for **Scaled** and **Block Floating Point** mode. The width of **xk_re** signal grows left from the **xn_re** binary point in the **Unscaled** mode by $(1+\log_2 N)$.

xk_im imaginary component of output data stream. **xk_im** is same as the input **xn_im** for **Scaled** and **Block Floating Point** mode. The width of **xk_im** signal grows left from the **xn_im** binary point in the **Unscaled** mode by $(1+\log_2 N)$.

xn_index marks the index of the input data. **xn_index** signal is marked as an unsigned signal of width $\log_2 N$ with binary point at 0.

xk_index marks the index of the output data. **xk_index** signal is marked as an unsigned signal of width $\log_2 N$ with binary point at 0.

rfd active high after the start signal is asserted till the **xn_index** count reaches $N-1$. The **rfd** signal is marked as **Bool**.

busy active high when the block is processing the current input data frame. The **busy** signal is marked as **Bool**.

vout marks the output data as valid or invalid. **vout** signal is marked as **Bool**.

edone active high one sample period before the block is ready to output the processed data frame. **edone** is marked as **Bool**.

done active high when the block is ready to output the processed data frame. **done** is marked as **Bool**.

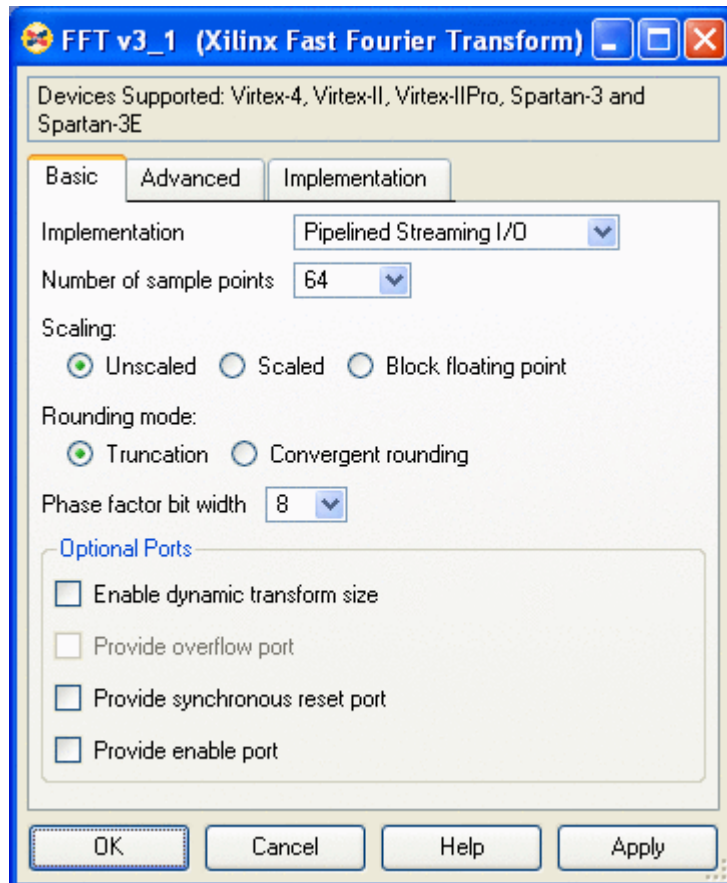
ovflo marks the output data frame with active high signal if an overflow condition was detected while processing the input data frame in the **Scaled** mode. **ovflo** signal is marked as **bool**.

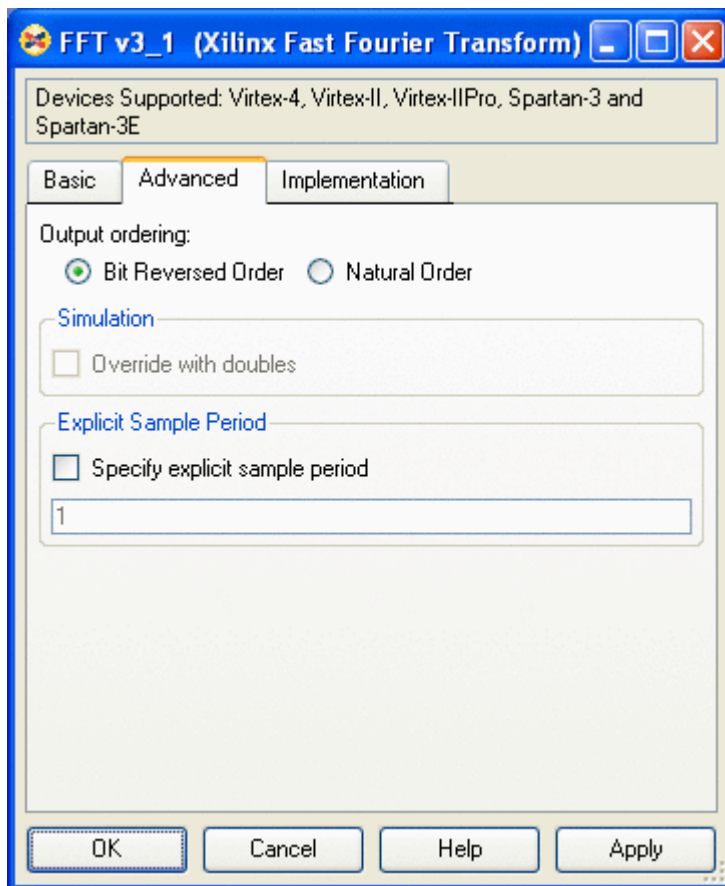
blk_exp specifies the exponent value for the output data frame in **Block Floating Point** mode. **blk_exp** is marked as

an unsigned signal of width 4 with binary point at 0.

Block Parameters Dialog Box

The FFT v3_1 block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.





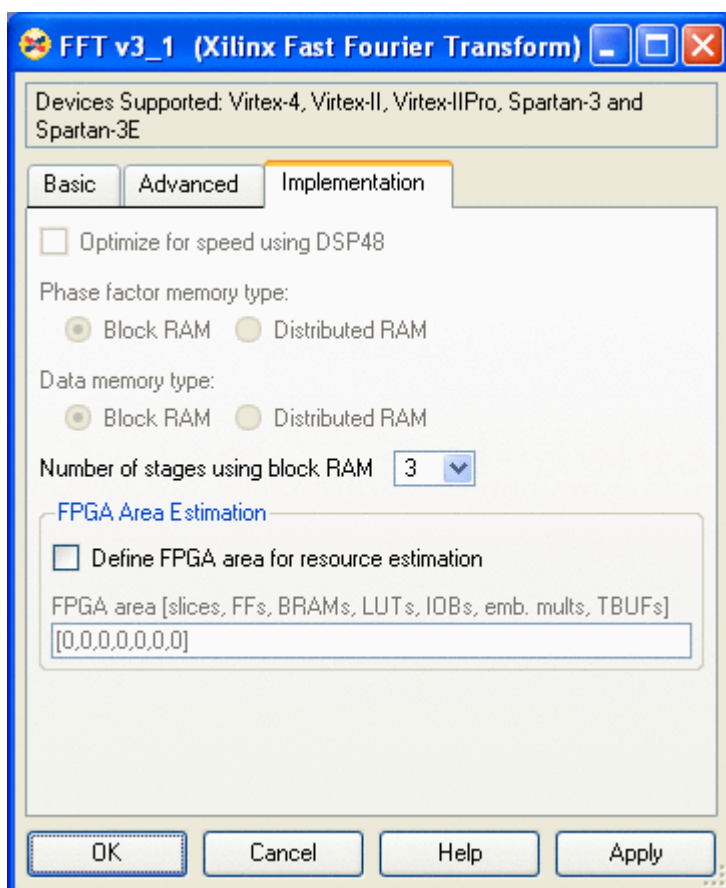


Figure 2: FFT v3_1 block parameters dialog box

Parameters specific to the FFTx block are:

- **Implementation:** option to choose between Radix-4 Streaming I/O, Radix-4 Burst I/O, Radix-2 Minimum Resources or Pipelined Streaming I/O.
- **Number of Sample Points:** transform length, one of $N = 2^{(3..16)} = 8 - 65536$. The FFTx block supports sample points, 64 - 8192 for Radix-4 Streaming I/O, 64 - 65536 for Radix-4 Burst I/O, and 8 - 65536 for Radix-2 Minimum Resources and Pipelined Streaming I/O, implementation.
- **Scaling:** option to select between Unscaled, Scaled and Block Floating Point output data types. The Block Floating Point option is not available for Radix-4 Streaming I/O and Pipelined Streaming I/O implementation.
- **Rounding Mode:** option to choose between Truncation and Convergent Rounding to be applied at the output of each rank.
- **Phase Factor Bit Width:** option to choose between 8, 12, 16, 20 or 24 to be used as bit widths for phase factors.
- **Output Ordering:** option to choose between bit reversed or natural order output. This option is available only for Pipelined Streaming I/O implementation.
- **Number of Stages Using Block RAM:** option to store data and phase factor in block ram and partially in distributed ram. This option is available only for Pipelined Streaming I/O implementation.
- **Phase Factor Memory Type:** option to choose between Block RAM and Distributed RAM. This option is available only for sample points 16 till 1024. This option is not available for Pipelined Streaming I/O implementation.
- **Data Memory Type:** option to choose between Block RAM and Distributed RAM. This option is available only for Radix-2 Minimum Resources implementation.

- **Enable Dynamic Transform Size:** option to have optional input ports `N` and `N_we` for dynamically varying the point size of input data frames.
- **Provide Overflow Port:** option to have optional output ports `ovflo` when Scaled scaling option is selected.
- **Optimize for Speed Using DSP48:** option to implement the entire dragonfly using DSP48's. Selecting this option would allow a faster maximum clock speed at the cost of using more DSP48's. This option is available only for Radix-4 Burst I/O implementation.

Other parameters used by this block are explained in the Common Parameters section at the beginning of this chapter.

Block Timing

To better understand the FFTx blocks control behavior and timing, please consult the core datasheet.

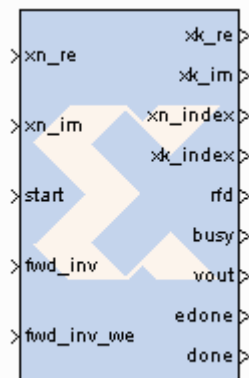
Xilinx LogiCORE

This block uses the Xilinx LogiCORE Fast Fourier Transform.

Note: Radix-4 Streaming I/O architecture would be obsoleted in subsequent release of the FFT block. Switch your design to use the Pipelined Streaming I/O architecture.

FFT v3_2

This block is listed in the following Xilinx Blockset libraries: *DSP and Index*.



The Xilinx FFT v3_2 block is supported only for Virtex4, Virtex-II, Virtex-II Pro and Spartan-3 devices. The FFT implements an efficient algorithm for computing the Discrete Fourier Transform (DFT). The N-point (where, $N = 2^m$, $m = 4 - 14$) forward or inverse DFT (IDFT) is computed on a vector of N complex values represented using data widths from 8 to 24, inclusive. The transform computation uses the Cooley-Tukey decimate-in-time algorithm as explained below.

Cooley-Tukey algorithm

The N-point DFT of a complex vector $x(n) = [x(0), x(1), \dots, x(N-1)]$, is the vector $X(k) = [X(0), X(1), \dots, X(N-1)]$, where the k-th element is defined as:

$$X(k) = \sum_{m=0}^{N-1} x(m) e^{-jmk \cdot 2\pi / N} \text{ for } k=0, 1, \dots, N-1$$

The inverse DFT (IDFT) is computed as,

$$x(m) = \sum_{k=0}^{N-1} X(k) e^{jmk \cdot 2\pi / N} \text{ for } m=0, 1, \dots, N-1$$

Block Interface

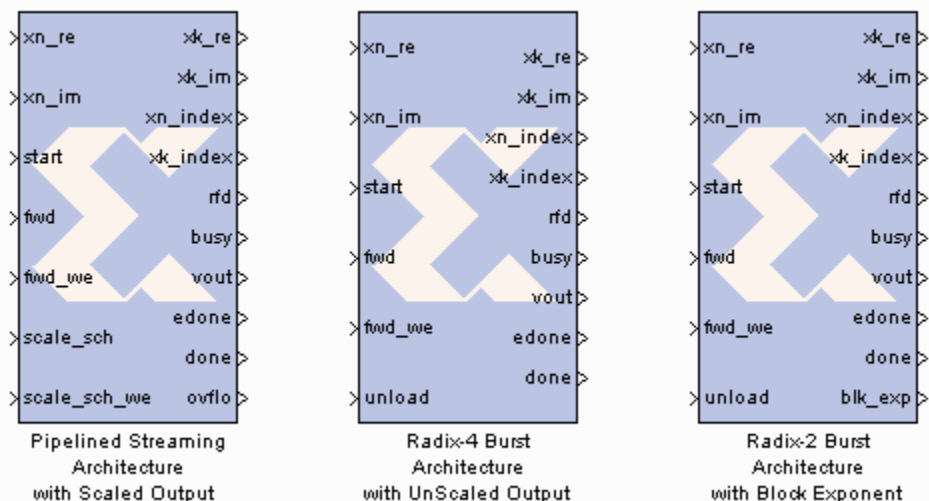


Figure 1: FFT v3_2 Block Interface for different Architectures and Scaling Options

The block interface (inputs and outputs as seen on the FFT icon) for various implementation and scaling modes is as follows:

Input signals:

<code>xn_re</code>	real component of input data stream. The signal driving <code>xn_re</code> can be a signed data type of width <code>S</code> with binary point at <code>S-1</code> , where <code>S</code> is a value between 8 and 24, inclusive. Eg: <code>Fix_8_7</code> , <code>Fix_24_23</code>
<code>xn_im</code>	imaginary component of input data stream. The signal driving <code>xn_im</code> can be a signed data type of width <code>S</code> with binary point at <code>S-1</code> , where <code>S</code> is a value between 8 and 24, inclusive. Eg: <code>Fix_8_7</code> , <code>Fix_24_23</code>
<code>start</code>	marks the beginning of each data frame. The <code>start</code> signal can be asserted as a pulse to start processing an input data frame or it can be tied to high. The signal driving <code>start</code> must be <code>Bool</code> .
<code>unload</code>	is used to read the output in natural order. The <code>unload</code> port is available only for implementing the <code>Radix-4 Burst I/O</code> or <code>Radix-2 Minimum Resources</code> architecture. The <code>unload</code> signal is sampled after the block is done processing the input frame. The block outputs data in natural order after the <code>unload</code> signal is asserted high. If the <code>start</code> signal is asserted before the <code>unload</code> signal, the block outputs data in bit reversed order. The signal driving <code>unload</code> must be <code>Bool</code> .
<code>fwd</code>	0 for inverse transform, 1 for forward transform. The signal driving <code>fwd</code> must be <code>Bool</code> .
<code>fwd_we</code>	when asserted, loads the transform type from the input port <code>fwd</code> for the next input data frame. The signal driving <code>fwd_we</code> must be <code>Bool</code> .
<code>scale_sch</code>	provides the scaling schedule to be used for the input data frame. The <code>scale_sch</code> port is available only for <code>Fixed Point Scaled</code> mode. The signal driving <code>scale_sch</code> must be unsigned signal of width <code>S</code> with binary point at 0, where, $S = 2 * \log_2 N$, for <code>Radix-2 Minimum Resources</code> , $2 * \text{ceil}(\log_2 N / 2)$, otherwise.
<code>scale_sch_we</code>	when asserted, loads the scaling schedule from the input port <code>scale_sch</code> for the next input data frame. The <code>scale_sch_we</code> port is available only for <code>Fixed Point Scaled</code> mode. The signal driving <code>scale_sch_we</code> must be <code>Bool</code> .
<code>nfft</code>	provides the point size for the next input data frame. The <code>nfft</code> port is available only when the checkbox for <code>Enable</code>

Dynamic Transform Size is selected. The signal driving `nfft` must be unsigned signal of width `S` with binary point at 0, where $S = \log_2 N$.

`nfft_we` when asserted, resets the current operation of the block and loads the point size from the input port `nfft` for the next input data frame. The `nfft_we` port is available only when the checkbox for **Enable Dynamic Transform Size** is selected. The signal driving `nfft_we` must be `Bool`.

Note: Both `xn_re` and `xn_im` signals should have the same data type.

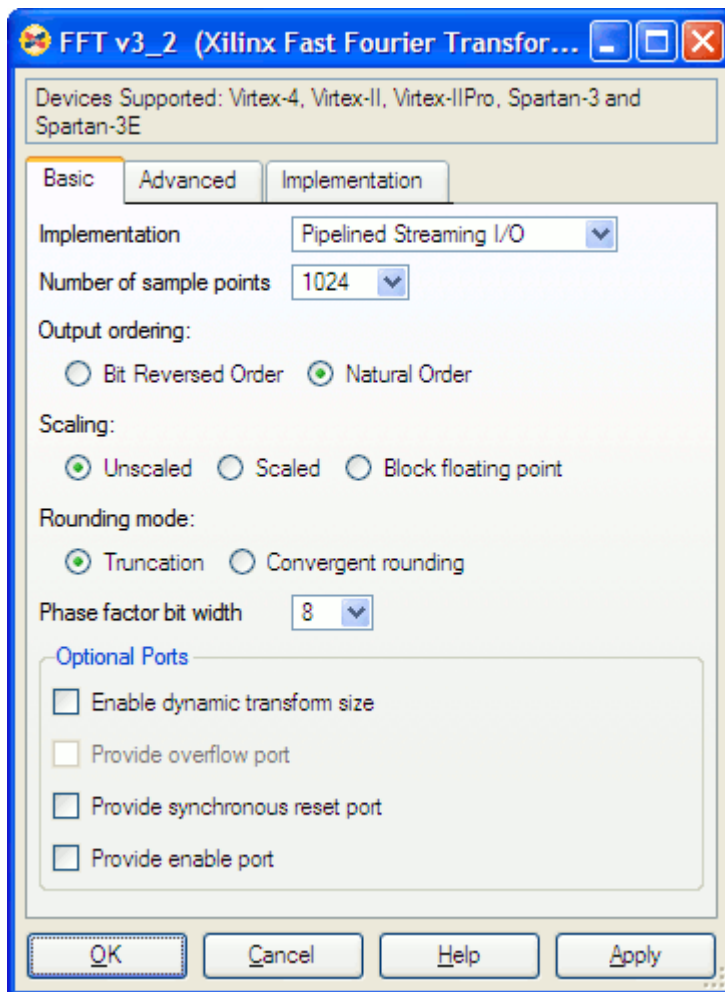
Output signals:

<code>xk_re</code>	real component of output data stream. <code>xk_re</code> is same as the input <code>xn_re</code> for Scaled and Block Floating Point mode. The width of <code>xk_re</code> signal grows left from the <code>xn_re</code> binary point in the Unscaled mode by $(1 + \log_2 N)$.
<code>xk_im</code>	imaginary component of output data stream. <code>xk_im</code> is same as the input <code>xn_im</code> for Scaled and Block Floating Point mode. The width of <code>xk_im</code> signal grows left from the <code>xn_im</code> binary point in the Unscaled mode by $(1 + \log_2 N)$.
<code>xn_index</code>	marks the index of the input data. <code>xn_index</code> signal is marked as an unsigned signal of width $\log_2 N$ with binary point at 0.
<code>xk_index</code>	marks the index of the output data. <code>xk_index</code> signal is marked as an unsigned signal of width $\log_2 N$ with binary point at 0.
<code>rfd</code>	active high after the start signal is asserted till the <code>xn_index</code> count reaches $N-1$. The <code>rfd</code> signal is marked as <code>Bool</code> .
<code>busy</code>	active high when the block is processing the current input data frame. The <code>busy</code> signal is marked as <code>Bool</code> .
<code>vout</code>	marks the output data as valid or invalid. <code>vout</code> signal is marked as <code>Bool</code> .
<code>edone</code>	active high one sample period before the block is ready to output the processed data frame. <code>edone</code> is marked as <code>Bool</code> .
<code>done</code>	active high when the block is ready to output the processed data frame. <code>done</code> is marked as <code>Bool</code> .
<code>ovflo</code>	marks the output data frame with active high signal if an overflow condition was detected while processing the input data frame in the Scaled mode. <code>ovflo</code> signal is marked as

`blk_exp` `bool`.
specifies the exponent value for the output data frame in Block Floating Point mode. `blk_exp` is marked as an unsigned signal of width 4 with binary point at 0.

Block Parameters Dialog Box

The FFT v3_2 block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.



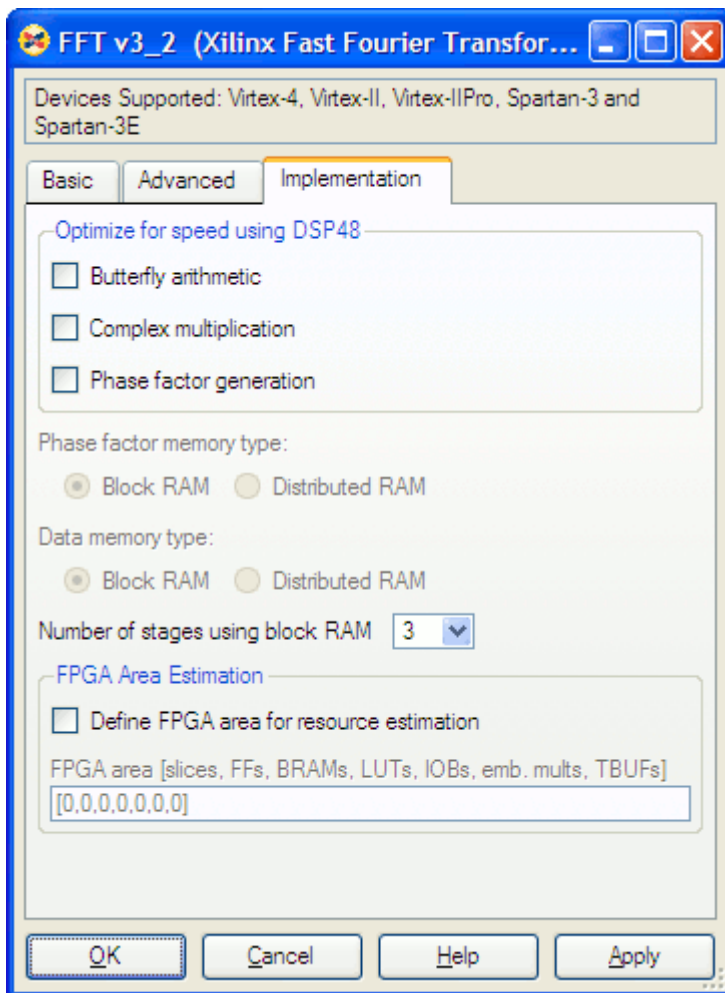


Figure 2: FFT v3_2 block parameters dialog box

Parameters specific to the FFTx block are:

- **Implementation:** option to choose between Radix-4 Streaming I/O, Radix-4 Burst I/O, Radix-2 Minimum Resources or Pipelined Streaming I/O.
- **Number of Sample Points:** transform length, one of $N = 2^{(3..16)} = 8 - 65536$. The FFTx block supports sample points, 64 - 8192 for Radix-4 Streaming I/O, 64 - 65536 for Radix-4 Burst I/O, and 8 - 65536 for Radix-2 Minimum Resources and Pipelined Streaming I/O, implementation.
- **Output Ordering:** option to choose between bit reversed or natural order output.
- **Scaling:** option to select between Unscaled, Scaled and Block Floating Point output data types. The Block Floating Point option is not available for Radix-4 Streaming I/O and Pipelined Streaming I/O implementation.
- **Rounding Mode:** option to choose between Truncation and Convergent Rounding to be applied at the output of each rank.
- **Phase Factor Bit Width:** option to choose between 8, 12, 16, 20 or 24 to be used as bit widths for phase factors.
- **Enable Dynamic Transform Size:** option to have optional input ports N and N_we for dynamically varying the point size of input data frames.
- **Provide Overflow Port:** option to have optional output ports ovflo when Scaled scaling option is selected
- **Optimize for Speed Using DSP48:** In Virtex-4, the complex multiplications, the butterfly additions/subtractions, and part of the phase factor generator can be computed in Xtreme DSP™ slices.

- o **Butterfly arithmetic:** option to implement the additions and subtractions of the butterflies using DSP48's. This option is only available if the output width is less than or equal to 30.
- o **Complex multiplication:** option to implement the complex multipliers built out of 4 real multipliers instead of 3. This allows the entire real complex multiplication to be calculated within the Xtreme DSP slices, resulting in faster clock speeds. Select this option for the largest increase in clock speed with a minimal increase in the number of extra DSP48's used.
- o **Phase factor generation:** option to implement the phase factor generator using two Xtreme DSP slices for every phase factor table, increasing clock speeds for the larger point sizes with wide phase factor bit widths.
- * **Number of Stages Using Block RAM:** option to store data and phase factor in block ram and partially in distributed ram. This option is available only for **Pipelined Streaming I/O** implementation.
- * **Phase Factor Memory Type:** option to choose between **Block RAM** and **Distributed RAM**. This option is available only for sample points 16 till 1024. This option is not available for **Pipelined Streaming I/O** implementation.
- * **Data Memory Type:** option to choose between **Block RAM** and **Distributed RAM**. This option is available only for sample points 16 till 1024. This option is not available for **Pipelined Streaming I/O** implementation.

Other parameters used by this block are explained in the Common Parameters section at the beginning of this chapter.

Block Timing

To better understand the FFT blocks control behavior and timing, please consult the core datasheet.

Xilinx LogiCORE

This block uses the Xilinx LogiCORE Fast Fourier Transform.

FFT v4_1

This block is listed in the following Xilinx Blockset libraries: *DSP and Index*.



The Xilinx FFT v4_1 block is supported for Virtex-5, Virtex-4, Virtex-II Pro, Spartan-3, Spartan-3E and Spartan-3A devices. The FFT implements an efficient algorithm for computing the Discrete Fourier Transform (DFT). The N-point (where, $N = 2^m$, $m = 4 - 14$) forward or inverse DFT (IDFT) is computed on a vector of N complex values represented using data widths from 8 to 24, inclusive. The transform computation uses the Cooley-Tukey decimate-in-time algorithm as explained below.

Cooley-Tukey algorithm

The N-point DFT of a complex vector $x(n) = [x(0), x(1), \dots, x(N-1)]$, is the vector $X(k) = [X(0), X(1), \dots, X(N-1)]$, where the k-th element is defined as:

$$X(k) = \sum_{m=0}^{N-1} x(m) e^{-jmk \cdot 2\pi / N} \text{ for } k=0, 1, \dots, N-1$$

The inverse DFT (IDFT) is computed as,

$$x(m) = \sum_{k=0}^{N-1} X(k) e^{jmk \cdot 2\pi / N} \text{ for } m=0, 1, \dots, N-1$$

Block Interface

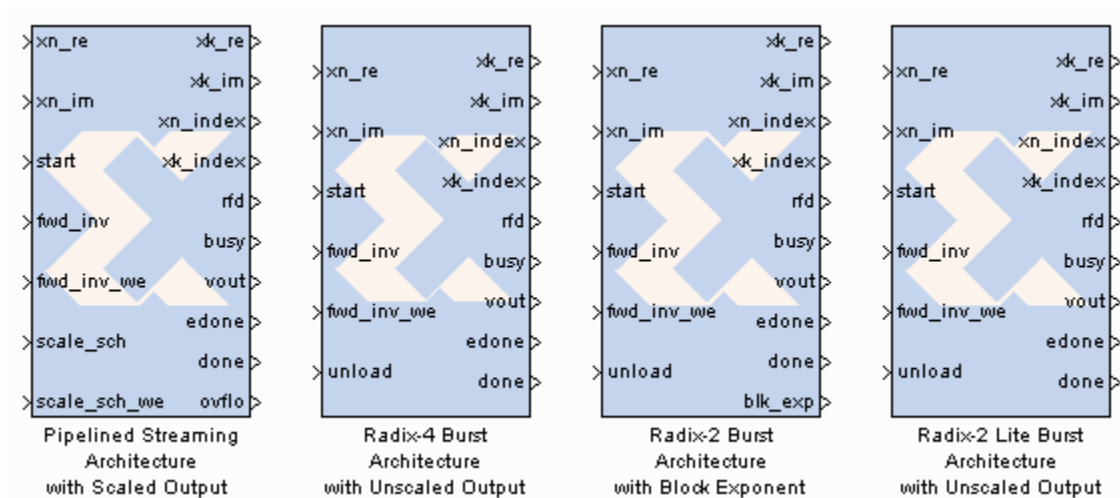


Figure 1: FFT v4_1 Block Interface for different Architectures and Scaling Options

The block interface (inputs and outputs as seen on the FFT icon) for various implementation and scaling modes is as follows:

Input signals:

<code>xn_re</code>	real component of input data stream. The signal driving <code>xn_re</code> can be a signed data type of width <code>S</code> with binary point at <code>S-1</code> , where <code>S</code> is a value between 8 and 24, inclusive. Eg: <code>Fix_8_7</code> , <code>Fix_24_23</code>
<code>xn_im</code>	imaginary component of input data stream. The signal driving <code>xn_im</code> can be a signed data type of width <code>S</code> with binary point at <code>S-1</code> , where <code>S</code> is a value between 8 and 24, inclusive. Eg: <code>Fix_8_7</code> , <code>Fix_24_23</code>
<code>start</code>	marks the beginning of each data frame. The <code>start</code> signal can be asserted as a pulse to start processing an input data frame or it can be tied to high. The signal driving <code>start</code> must be <code>Bool</code> .
<code>unload</code>	is used to read the output in natural order. The <code>unload</code> port is available only for implementing the Radix-4 Burst I/O, Radix-2 Burst I/O or Radix-2 Lite Burst I/O architecture. The <code>unload</code> signal is sampled after the block is done processing the input frame. The block outputs data in natural order after the <code>unload</code> signal is asserted high. If the <code>start</code> signal is asserted before the <code>unload</code> signal, the block outputs data in bit reversed order. The signal driving <code>unload</code> must be <code>Bool</code> .
<code>fwd</code>	0 for inverse transform, 1 for forward transform. The signal driving <code>fwd</code> must be <code>Bool</code> . By default, the FFT is configured for forward transform.
<code>fwd_we</code>	when asserted, loads the transform type from the input port <code>fwd</code> for the next input data frame. The signal driving <code>fwd_we</code> must be <code>Bool</code> .
<code>scale_sch</code>	provides the scaling schedule to be used for the input data frame. The <code>scale_sch</code> port is available only for Fixed Point Scaled mode. The signal driving <code>scale_sch</code> must be unsigned signal of width <code>S</code> with binary point at 0, where, $S = 2 * \log_2 N$, for Radix-2 architectures, $2 * \text{ceil}(\log_2 N / 2)$, otherwise.
<code>scale_sch_we</code>	when asserted, loads the scaling schedule from the input port <code>scale_sch</code> for the next input data frame. The <code>scale_sch_we</code> port is available only for Fixed Point Scaled mode. The signal driving <code>scale_sch_we</code> must be <code>Bool</code> .
<code>nfft</code>	provides the point size for the next input data frame. The <code>nfft</code> port is available only when the checkbox for Enable

Dynamic Transform Size is selected. The signal driving `nfft` must be unsigned signal of width 5 with binary point at 0, `UFix_5_0`.

`nfft_we` when asserted, resets the current operation of the block and loads the point size from the input port `nfft` for the next input data frame. The `nfft_we` port is available only when the checkbox for **Enable Dynamic Transform Size** is selected. The signal driving `nfft_we` must be `Bool`.

Note: Both `xn_re` and `xn_im` signals should have the same data type.

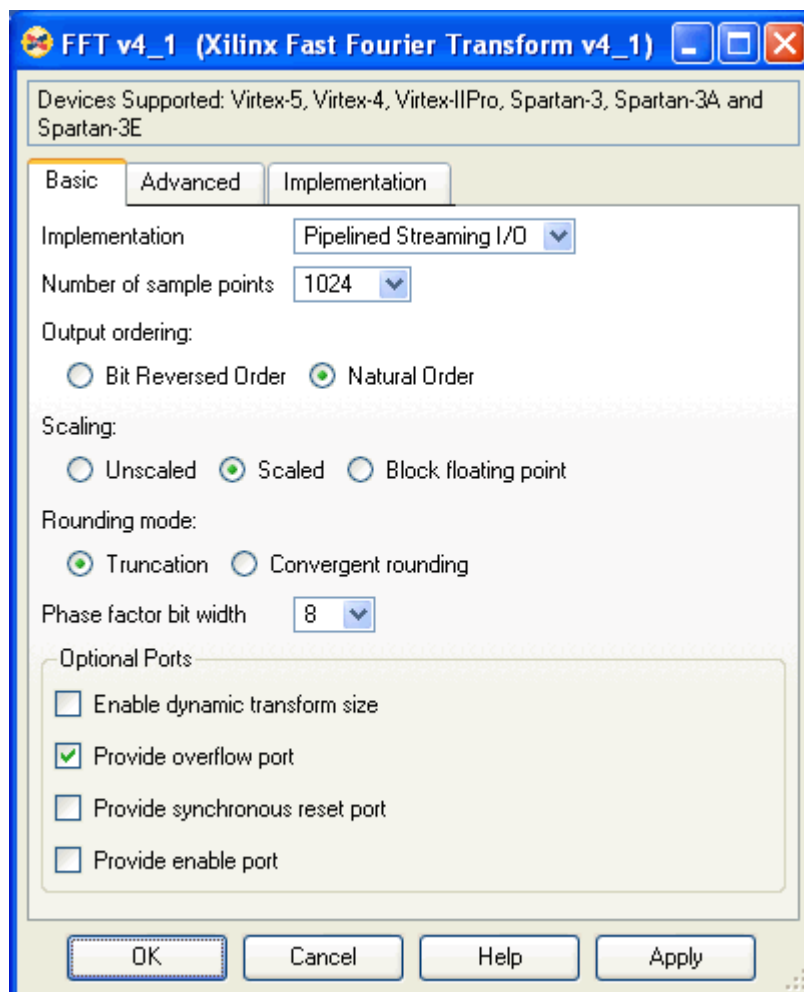
Output signals:

<code>xk_re</code>	real component of output data stream. <code>xk_re</code> is same as the input <code>xn_re</code> for Scaled and Block Floating Point mode. The width of <code>xk_re</code> signal grows left from the <code>xn_re</code> binary point in the Unscaled mode by $(1+\log_2 N)$.
<code>xk_im</code>	imaginary component of output data stream. <code>xk_im</code> is same as the input <code>xn_im</code> for Scaled and Block Floating Point mode. The width of <code>xk_im</code> signal grows left from the <code>xn_im</code> binary point in the Unscaled mode by $(1+\log_2 N)$.
<code>xn_index</code>	marks the index of the input data. <code>xn_index</code> signal is marked as an unsigned signal of width $\log_2 N$ with binary point at 0.
<code>xk_index</code>	marks the index of the output data. <code>xk_index</code> signal is marked as an unsigned signal of width $\log_2 N$ with binary point at 0.
<code>rfd</code>	active high after the start signal is asserted till the <code>xn_index</code> count reaches $N-1$. The <code>rfd</code> signal is marked as <code>Bool</code> .
<code>busy</code>	active high when the block is processing the current input data frame. The <code>busy</code> signal is marked as <code>Bool</code> .
<code>vout</code>	marks the output data as valid or invalid. <code>vout</code> signal is marked as <code>Bool</code> .
<code>edone</code>	active high one sample period before the block is ready to output the processed data frame. <code>edone</code> is marked as <code>Bool</code> .
<code>done</code>	active high when the block is ready to output the processed data frame. <code>done</code> is marked as <code>Bool</code> .
<code>ovflo</code>	marks the output data frame with active high signal if an overflow condition was detected while processing the input data frame in the Scaled mode. <code>ovflo</code> signal is marked as

`blk_exp` `bool`.
 specifies the exponent value for the output data frame in Block Floating Point mode. `blk_exp` is marked as an unsigned signal of width 5 with binary point at 0.

Block Parameters Dialog Box

The FFT v4_1 block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.



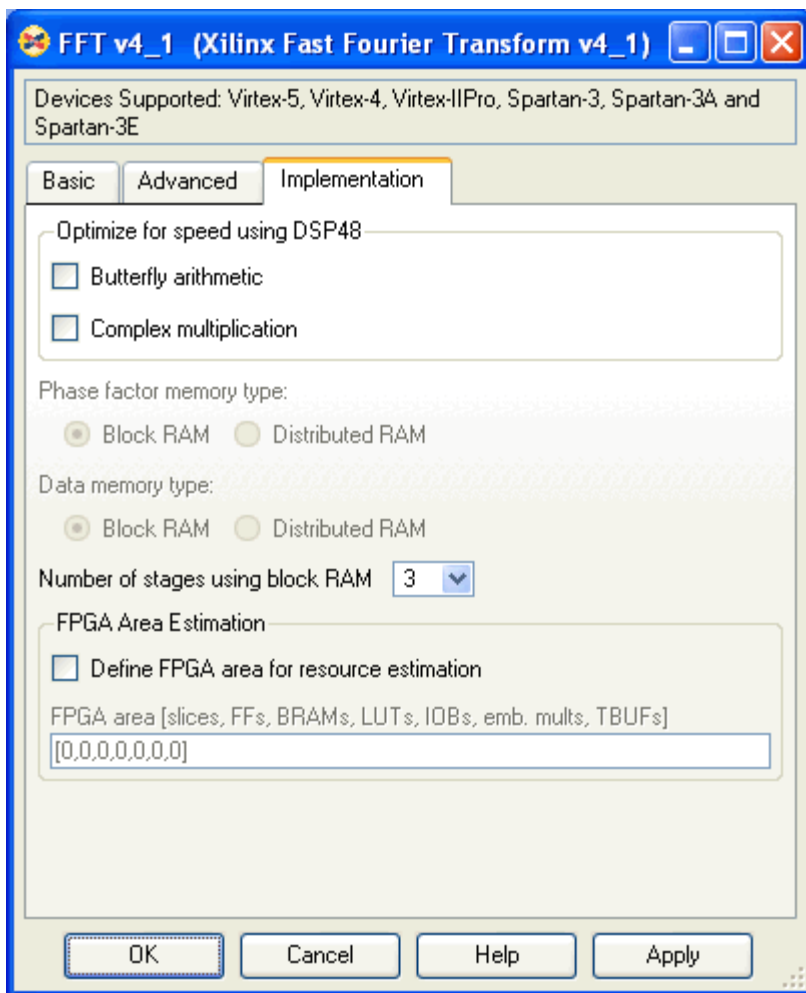


Figure 2: FFT v4_1 block parameters dialog box

Parameters specific to the FFT v4_1 block are:

- **Implementation:** option to choose between Radix-4 Burst I/O, Radix-2 Burst I/O, Pipelined Streaming I/O or Radix-2 Lite Burst I/O.
- **Number of Sample Points:** transform length, one of $N = 2^{(3..16)} = 8 - 65536$. The FFT block supports sample points, 64 - 65536 for Radix-4 Burst I/O, and 8 - 65536 otherwise.
- **Output Ordering:** option to choose between bit reversed or natural order output.
- **Scaling:** option to select between Unscaled, Scaled and Block Floating Point output data types. The Block Floating Point option is not available for Pipelined Streaming I/O implementation.
- **Rounding Mode:** option to choose between Truncation and Convergent Rounding to be applied at the output of each rank.
- **Phase Factor Bit Width:** option to choose a value between 8 and 24, inclusive to be used as bit widths for phase factors.
- **Enable Dynamic Transform Size:** option to have optional input ports `nfft` and `nfft_we` for dynamically varying the point size of input data frames.
- **Provide Overflow Port:** option to have optional output port `ovflo` when Scaled scaling option is selected.
- **Optimize for speed Using DSP48:** In Virtex-4 and Virtex-5, the complex multiplications and the butterfly additions/subtractions can be computed in Xtreme DSP™ slices.

- o **Butterfly arithmetic:** option to implement the additions and subtractions of the butterflies using DSP48's. This option is only available in Virtex-4, if the output width is less than or equal to 30. In Virtex-5, this feature is available for all output widths.
- o **Complex multiplication:** option to implement the complex multipliers built out of 4 real multipliers instead of 3. This allows the entire real complex multiplication to be calculated within the Xtreme DSP slices, resulting in faster clock speeds. Select this option for the largest increase in clock speed with a minimal increase in the number of extra DSP48's used. This option is only available for Virtex-4. In Virtex-5 it is always selected.
- * **Number of Stages Using Block RAM:** option to store data and phase factor in block ram and partially in distributed ram. This option is available only for **Pipelined Streaming I/O** implementation.
- * **Phase Factor Memory Type:** option to choose between **Block RAM** and **Distributed RAM**. This option is available only for all sample points 1024 and under. This option is not available for **Pipelined Streaming I/O** implementation.
- * **Data Memory Type:** option to choose between **Block RAM** and **Distributed RAM**. This option is available only for all sample points 1024 and under. This option is not available for **Pipelined Streaming I/O** implementation.

Other parameters used by this block are explained in the Common Parameters section at the beginning of this chapter.

Block Timing

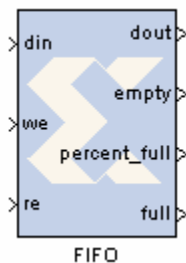
To better understand the FFT blocks control behavior and timing, please consult the core datasheet.

Xilinx LogiCORE

This block uses the Xilinx LogiCORE Fast Fourier Transform.

FIFO

This block is listed in the following Xilinx Blockset libraries: *Control Logic, Memory, and Index.*



The Xilinx FIFO block implements a FIFO memory queue.

Values presented at the module's data-input port are written to the next available empty memory location when the write-enable input is one. By asserting the read-enable input port, data can be read out of the FIFO via the data output port (dout) in the order in which they were written. The FIFO can be implemented using block or distributed RAM.

The `full` output port is asserted to one when no unused locations remain in the module's internal memory. The `percent_full` output port indicates the percentage of the FIFO that is full, represented with user-specified precision. When the `empty` output port is asserted the FIFO is empty.

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

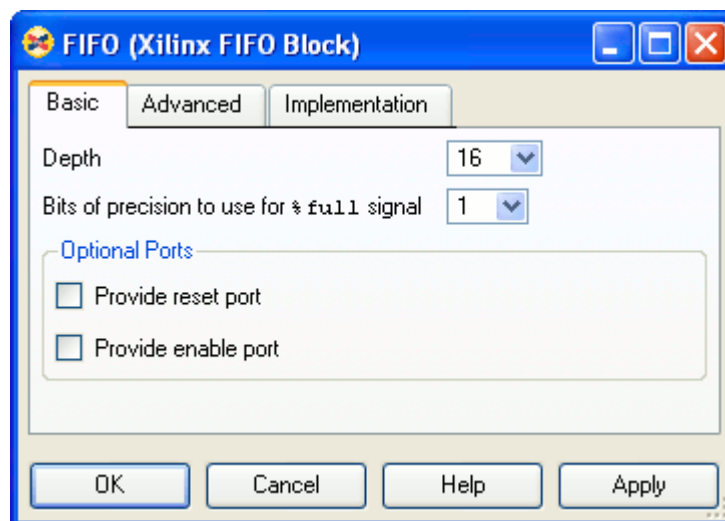


Figure 1: FIFO block Basic tab

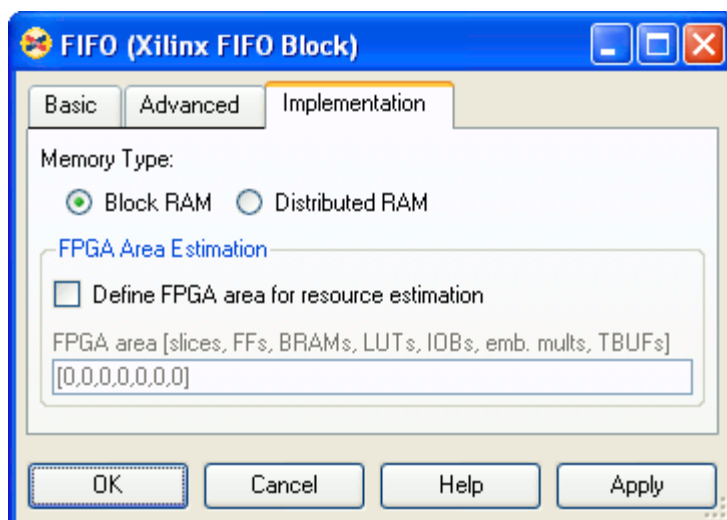


Figure 2: FIFO block Implementation tab

Parameters specific to the FIFO block are:

- **Depth:** specifies the number of words that can be stored.
- **Bits of precision to use for %full signal:** specifies the bit width of the %full port. The binary point for this unsigned output is always at the top of the word. Thus, if for example precision is set to one, the output can take two values: 0.0 and 0.5, the latter indicating the FIFO is at least 50% full.
- **Memory Type:** specifies how the FIFO is implemented in the FPGA; possible choices include block or distributed RAM.

Parameter	Block RAM	Distributed RAM
Depth (minimum)	16	16
Depth (maximum)	64K	256
Data Width (minimum)	1	1
Data Width (maximum)	256	256
Percent Full Bits (minimum)	1	1
Percent Full Bits (maximum)	minimum of $\log_2(\text{Depth})$ or 5	minimum of $\log_2(\text{Depth})$ or 5

Figure 3: Parameter ranges for different FIFO memory types.

Other parameters used by this block are described in the Common Parameters section at the beginning of this chapter.

Xilinx LogiCORE

This block is implemented with the Xilinx LogiCORE FIFO Generator.

FIR Compiler v1_0

This block is listed in the following Xilinx Blockset libraries: *DSP and Index*.



The Xilinx Fir Compiler v1_0 block implements a high speed MAC based FIR filter. It accepts a stream of input data and computes filtered output with a fixed delay, based on the filter configuration. The filter is implemented using cascaded DSP48 slices as shown in Figure 1.

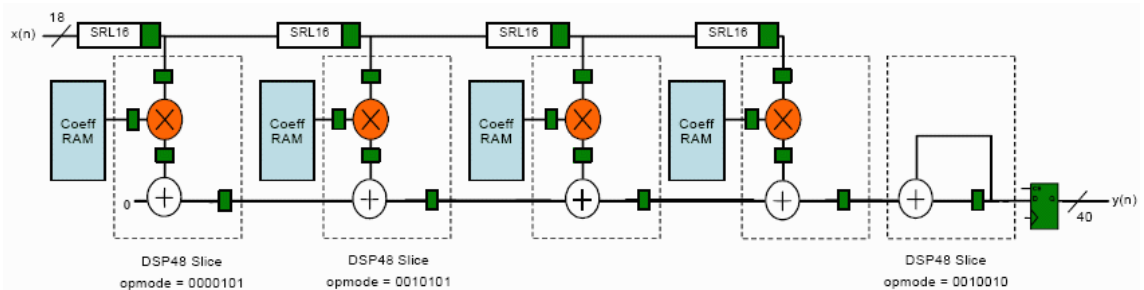


Figure 1 Fir Compiler architecture

Block Interface

The FIR Compiler v1_0 block can be configured to have a number of optional ports in addition to din and dout ports which appear in all filter configurations.

- * **din** : data in port on the FIR Compiler. The data for all channels is provided to the FIR Compiler in a time multiplexed manner(Figure 2) through this port.

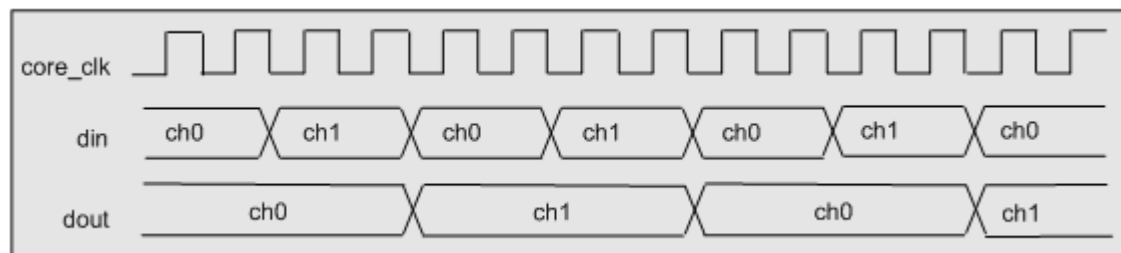


Figure 2 Multi channel data input and output for a decimating filter with sample rate change of 2 and hardware over-sampling of 2

- * **vin** : valid in port. Marks each input on the din port as valid(high) or invalid(low). For multi-rate, multi-channel filter cases, if the signal driving vin port is high for channel 0's data sample, then it must remain high until a corresponding sample for all the other channels has been clocked in(Figure 3). In other words, a data sample for each channel from the first channel through to the final channel must be provided on consecutive clock cycles and the vin port must remain high over all these cycles. A simulation error will be reported if this rule is not adhered to.

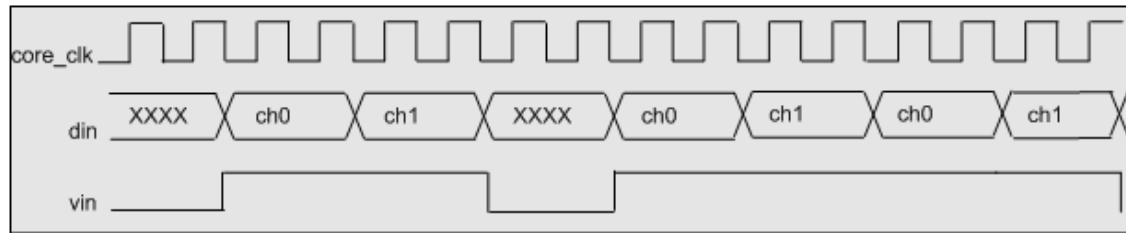


Figure 3 Valid port usage with multi-channel filters

- **en** : synchronous enable port
- **rst** : synchronous reset port
- **rfd** : ready for data port. The output on this port goes high if the FIR block is ready to accept new input data.
- **dout** : data out port on the FIR compiler. Filtered output data for all the channels is provided through this port in a time multiplexed fashion.
- **vout** : indicates if the current output data on the dout port is valid(high) or invalid(low).
- **core_rdy** : control port to indicate when the block has initialised and is ready to accept data on the input ports. All inputs presented to the block will be ignored until **core_rdy** has been asserted by the core. This is a one shot control signal. It remains low at the start of the simulation and goes high after a time corresponding to the startup latency of the core. Once this signal goes high it will stay high for the remaining runtime. This port cannot be instanced if **rst** and **rfd** ports are used.
- **chan_out** : indicates which channel the current output data on the dout port belongs to.
- **chan_in** : indicates which channel's input data sample should be driven onto the **din** port next. As soon as the current input sample on **din** is clocked into the core the value on this output port will change indicating the next channel for which a data input is required.

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

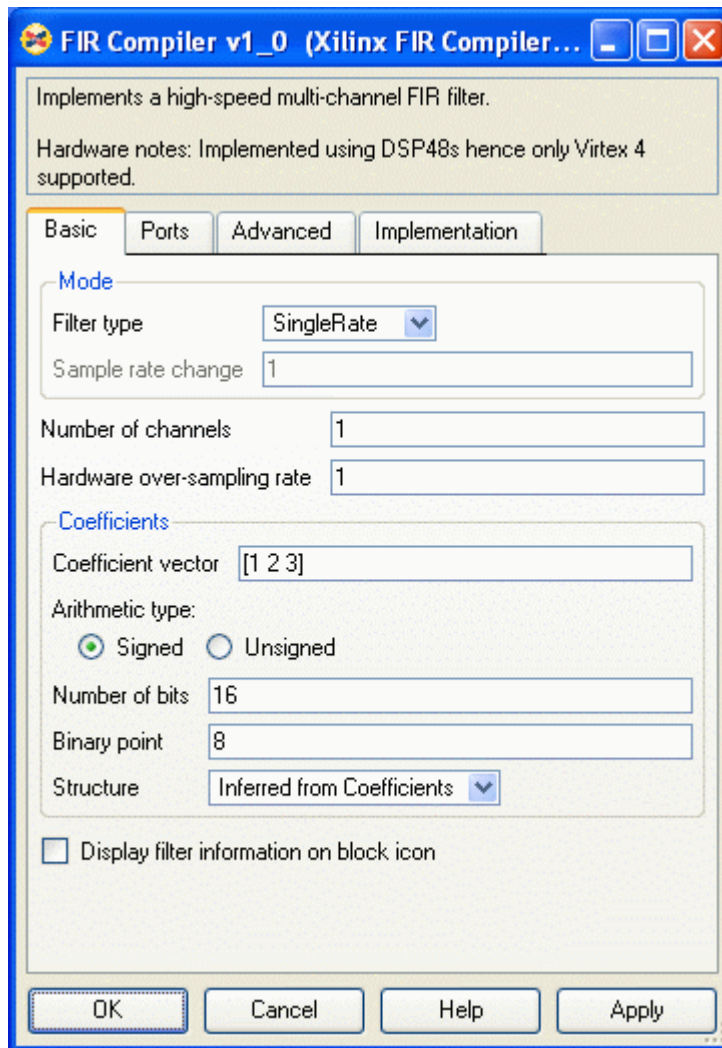


Figure 4 FIR Compiler v1_0 Basic tab

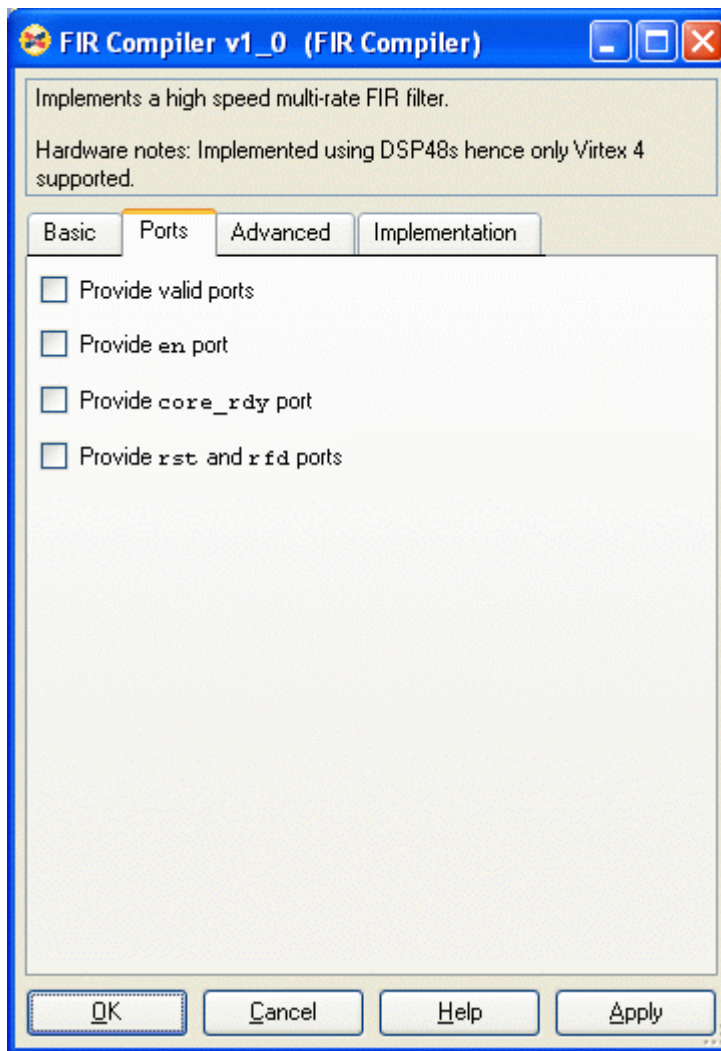


Figure 5 FIR Compiler v1_0 Ports tab

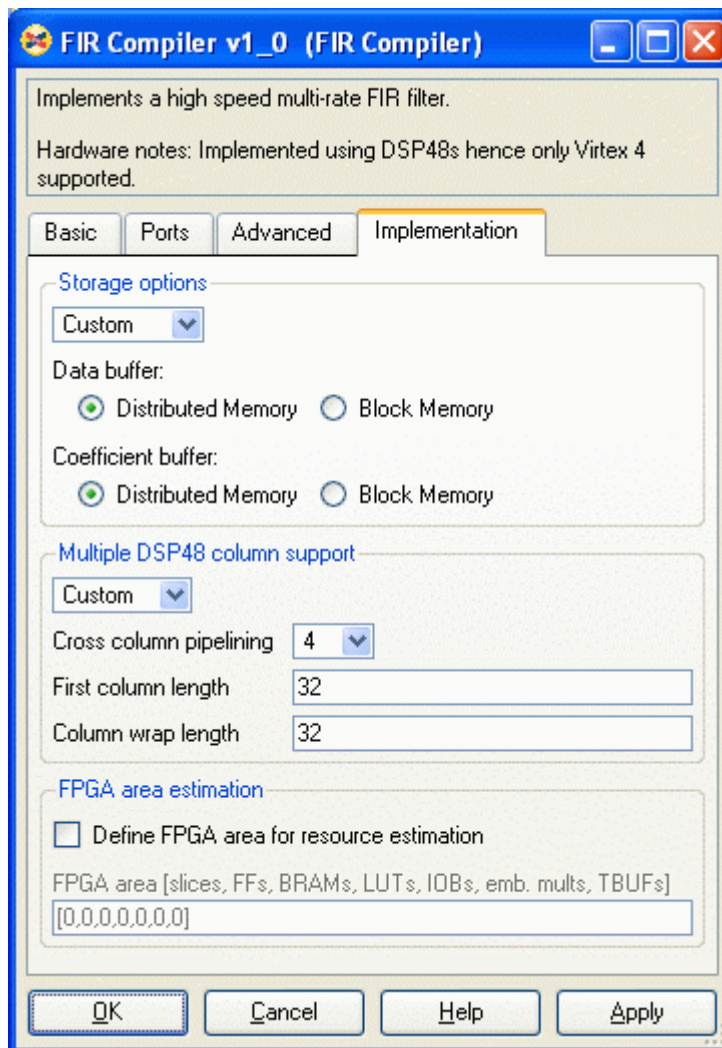


Figure 6 FIR Compiler v1_0 Implementation tab

Parameters specific to the block are:

- **Filter type(Basic):** Allows user to specify the filter as one of the following:
 - **SingleRate:** The data rate of the input and the output are the same.
 - **Decimation:** The data rate of the output is slower than the input by a factor specified in **Sample rate change** parameter
 - **Interpolation :** The data rate of the output is faster than the input by a factor specified in **Sample rate change** parameter
- **Sample rate change(Basic):** Allows user to specify the factor by which the sample rate of the output changes compared to input. This parameter is only active when the **Filter Type** is set to either **Decimation** or **Interpolation**. When **Decimation** is selected the sample rate of the output decreases with respect to the input by a factor equal to **Sample rate change**. When **interpolation** is selected the sample rate of the output increases with respect to the input by a factor equal to **Sample rate change**.
- **Number of channels(Basic):** The number of data channels to be processed by the Fir Compiler block. The multiple channel data is passed to the core in a time multiplexed manner. A maximum of 16 channels is supported.
- **Hardware over-sampling rate(Basic) :** Specifies the ratio between the faster of input data rate and the output data rate versus the internal core rate. The overclocking rate determines the number of filter taps folded onto each DSP48 in the filter.

- **Coefficient vector(Basic):** Specifies the coefficient vector as a single matlab row vector. The number of taps is inferred from the length of the matlab row vector. The number of coefficients in a filter configuration, the specified coefficient structure and the hardware over-sampling factor determine the number of DSP48s instanced by the filter. It is possible to enter these coefficients using the FDATool block as well.
- **Coefficients Structure(Basic):** Specifies the coefficient structure. Depending on the coefficient structure optimizations are made in the core to reduce the amount of hardware required to implement a particular filter configuration. The selected structure can be any of the following:
 - Inferred from coefficients
 - Non-Symmetric
 - Symmetric
 - Half Band

The vector of coefficients specified must match the structure specified unless **Inferred from Coefficients** is selected in which case the structure is determined automatically from these coefficients.

- **Provide valid ports(Ports):** Provides vin and vout ports on the block.
- **Provide core_rdy port(Ports):** Provides an output port on the block which is low initially for a number of cycles equal to the start up latency of the core. After this signal is driven high the core has completed its initialisation sequence and is able to accept input data. Refer to the data sheet for more information on the startup latency.
- **Provide rst and rfd ports(Ports):** Provides rst and rfd ports on the block.
- **Storage options(Implementation):** Allows custom memory options to be selected for the storage of both coefficients and data.
- **Data buffer(Implementation):** Specifies the type of memory used to store data samples.
- **Coefficient buffer(Implementation):** Specifies the type of memory used to store the coefficients.
- **Multiple DSP48 column support(Implementation):** Specifies the method for implementing filters over multiple columns of DSP48 slices in a Virtex-4 device. Custom mode allows the user to specify an exact multi-column implementation and setting this control to disabled turns off all support for multi-column implementations.
- **First column length(Implementation):** Specifies the length of the first column in a multi-column DSP48 filter implementation. This control is only active when Multiple DSP48 column support is set to custom.
- **Cross column pipelining(Implementation):** Specifies the length of the pipelines between columns in a multi-column DSP48 filter implementation. This control is only active when Multiple DSP48 column support is set to custom.
- **Column wrap length(Implementation):** Specifies the length of subsequent columns in a multi-column DSP48 filter implementation. This value must be greater than or equal to the first column length specified. This control is only active when Multiple DSP48 column support is set to custom.

Other parameters used by this block are explained in the Common Parameters section at the beginning of this chapter.

Xilinx LogiCORE

The block always uses the Xilinx LogiCORE FIR Compiler.

Known Issues

The following known issues are associated with certain features of Fir Compiler in SysGen:

- Currently the FIR Compiler block only supports Mac Fir for Virtex 4 devices.

- There is no support for re-loadable coefficients or multiple coefficient sets
- There is no automatic multi-column support
- There is no automatic multi-column support
- There is support only for a maximum of 512 taps
- If a design containing the block simulates but produces the error reproduced below during netlist phase then refer to the coregen.log file located in \$target_directory/sysgen/coregen_XXXX/coregen_tmp:

standard exception: XNetlistEngine: An exception
was raised: com.xilinx.sysgen.netlist.f: ERROR:
coreutil - Failure to generate output products
at C:/MATLAB701/toolbox/xilinx/sysgen/scripts/
SgGenerateCores.pm line 590

\$target_directory refers to the netlist target directory as specified on the SysGen token.

FIR Compiler v2_0

This block is listed in the following Xilinx Blockset libraries: DSP and Index.

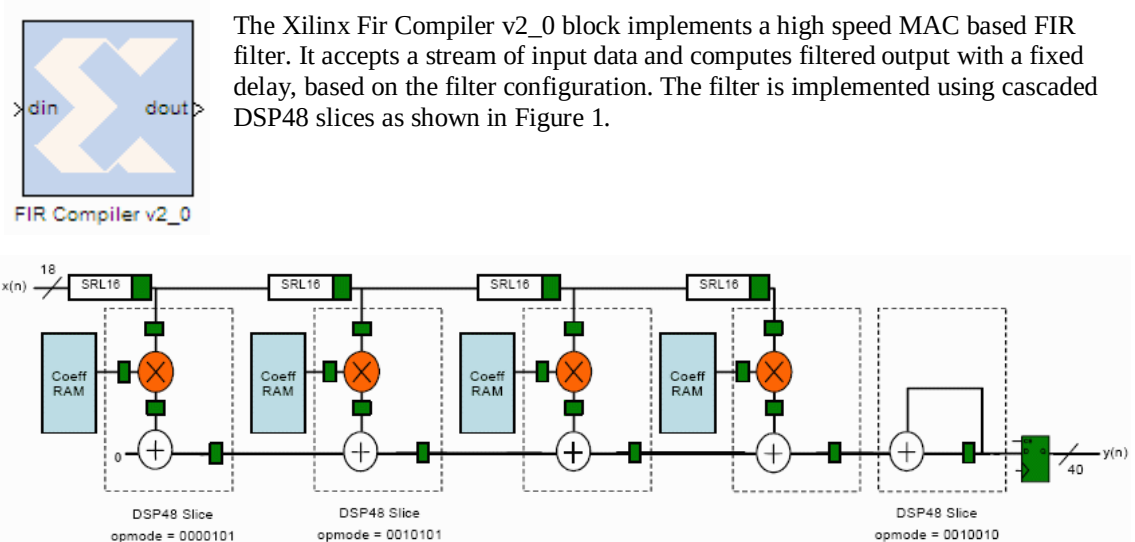


Figure 1 FIR Compiler architecture

Block Interface

The FIR Compiler v2_0 block can be configured to have a number of optional ports in addition to din and dout ports which appear in all filter configurations. The first data is consumed by the FIR compiler when the core is ready. This can be inferred by monitoring the core_rdy signal which has been described in this section.

- **din** : data in port on the FIR Compiler. The data for all channels is provided to the FIR Compiler in a time multiplexed manner (Figure 2) through this port.

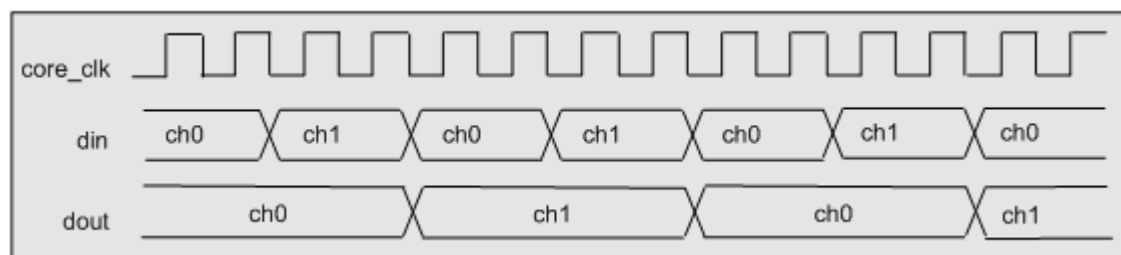


Figure 2 Multi channel data input and output for a decimating filter with sample rate change of 2 and hardware over-sampling of 2

- **vin** : valid in port. Marks each input on the din port as valid (high) or invalid (low).

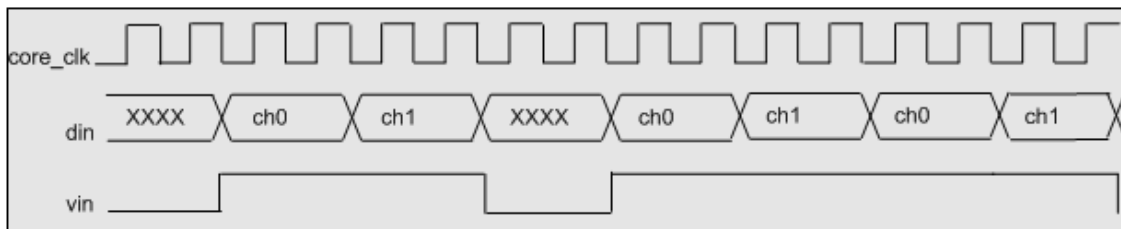


Figure 3 Valid port usage with multi-channel filters

- **en** : synchronous enable port
- **rst** : synchronous reset port
- **rfd** : ready for data port. The output on this port goes high if the FIR block is ready to accept new input data.
- **dout** : data out port on the FIR compiler. Filtered output data for all the channels is provided through this port in a time multiplexed fashion.
- **vout** : indicates if the current output data on the dout port is valid(high) or invalid(low).
- **core_rdy** : control port to indicate when the block has initialised and is ready to accept data on the input ports. All inputs presented to the block will be ignored until **core_rdy** has been asserted by the core. This is a one shot control signal. It remains low at the start of the simulation and goes high after a time corresponding to the startup latency of the core. Once this signal goes high it will stay high for the remaining runtime. This port cannot be instantiated if **rst** and **rfd** ports are used. The startup latency is a maximum of one cycle.
- **chan_out** : indicates which channel the current output data on the dout port belongs to.
- **chan_in** : indicates which channel's input data sample should be driven onto the din port next. As soon as the current input sample on din is clocked into the core, the value on this output port will change indicating the next channel for which a data input is required.
- **coef_ld** : coefficient load port. The signal driving **coef_ld** must be boolean. For more information on the **coef_ld** port refer to the section on re-loadable coefficients.
- **coef_we** : coefficient write enable port. The signal driving **coef_we** must be boolean. For more information on the **coef_we** port refer to the section on re-loadable coefficients.
- **coef_din** : coefficient data input port. The signal driving **coef_din** must have a type identical to the one specified for coefficients in the block mask. For more information on the **coef_din** port refer to the section on re-loadable coefficients.

Reloading Coefficients

The FIR Compiler allows for reloading new coefficient sets dynamically. Three ports **coef_ld**, **coef_we** and **coef_din**, are used to accomplish this. The timing relationship between the signals on various ports is shown in figure 4.

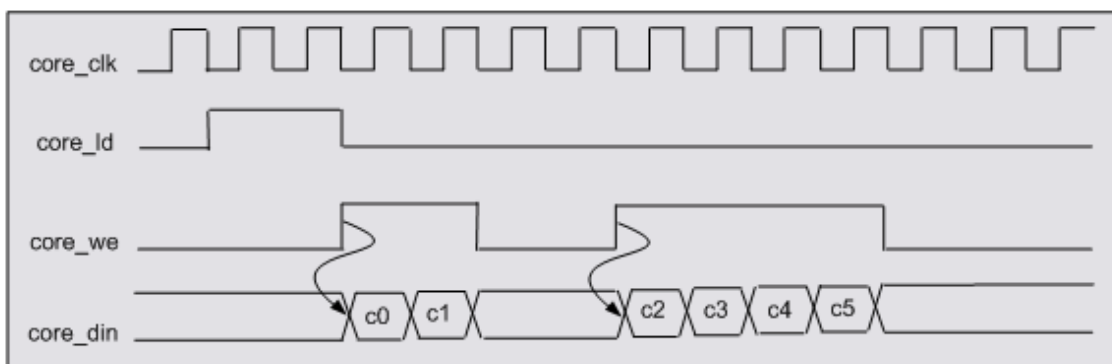


Figure 4 : Timing Relationship for Option Reloadable Coefficient Support

The **core_we** can go high at any time after the falling edge of **coef_ld** signal. This will initiate the loading of the coefficients. Also observe that all coefficients need not be loaded continuously. When the last coefficient is loaded, after a delay of two **core_clk** cycles (a delay of one in the case of halfband filter coefficients), the new set of coefficients becomes active.

The order that the FIR filter is loaded is determined by the configuration of the FIR Compiler. A utility function called **xlGetReloadOrder** is also provided to determine this order.

xlGetReloadOrder

xlGetReloadOrder is a Matlab utility function that returns the mapping between the time order of the reloaded coefficient and their tap position in the FIR filter. The function xlGetReloadOrder accepts the following arguments :

- * **n_taps** : number of filter taps in the FIR Compiler
- * **coef_struct** : Specifies the coefficient structure. It can take one of the following String values
 - * 'Symmetric'
 - * 'Negative-Symmetric'
 - * 'Halfband'
 - * 'Non-Symmetric'
- * **filter_type** : Specifies the type of filter being used. It can take one of the following numeric values
 - * 'SingleRate'
 - * 'Decimation'
 - * 'Interpolation'
- * **rate_change** : Interpolation or decimation rate change (1-16)
- * **overclock** : overclocking factor (1 – n_taps)
- * **multiple_column_support** : Disable or custom specify multiple column support
 - * 'Disabled'
 - * 'Custom'
- * **first_column_length** : Integer specifying the first column length when multiple column support is enabled
- * **column_wrap_length** : Integer specifying the column wrap length when multiple column support is specified

For example, the coefficient reload order of an FIR Compiler block configured as a 16-tap symmetric coefficients decimate by 2 filter, which is overclocked by 2 can be obtained as shown below:

```
>> xlGetReloadOrder( 16, 'Symmetric', 'Decimation', 2, 2, 'Disabled',
4, 32)
ans =
    5
    7
    4
    6
    1
    3
    0
    2
```

The utility returns a vector that captures the reload order. The first element of the vector here is 5. This implies that the first coefficient passed to the Fir Compiler forms the 5th tap of the FIR filter (and hence also the 10th because the filter has symmetric coefficient structure). Similarly the second element value is 7 which denotes that the 2nd coefficient passed to the FIR Compiler forms the 7th (and also 8th) tap of the FIR filter.

Alternatively, the block handle or the block name can also be passed to the function to obtain the reload order :

```
>> xlGetReloadOrder('TestModel/FirCompiler')
ans =
    5
```

7
4
6
1
3
0
2

An example model with reloadable coefficients is located here

Symmetric Interpolation

Symmetric interpolation cases have a reload scheme that is different from all the others in that the coefficients are loaded in a combined manner. Under this configuration the `xlGetReloadOrder` returns a string describing the combination :

```
>> xlGetReloadOrder(16, 'Symmetric', 'Interpolation', 2, 2, 'Disabled',
4, 32)
ans =
{4 - 5} {6 - 7} {4 + 5} {6 + 7} {0 - 1} {2 - 3} {0 + 1} {2 + 3}
```

The above string implies that the first sample clocked into the `coef_din` port is interpreted as the difference between coefficient 4 and coefficient 5. The second sample is interpreted as the difference between coefficient 6 and coefficient 7.

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

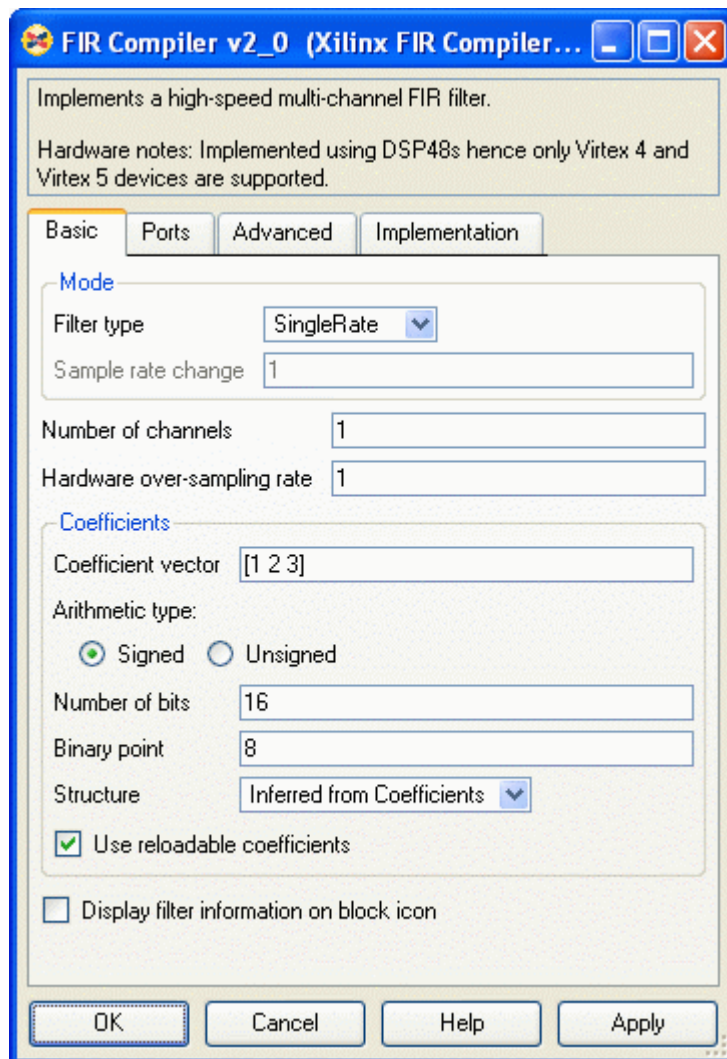


Figure 5 FIR Compiler v2_0 Basic tab

Parameters specific to the FIR Compiler block on the Basic tab are:

- **Filter type:** Allows user to specify the filter as one of the following:
 - **SingleRate:** The data rate of the input and the output are the same.
 - **Decimation:** The data rate of the output is slower than the input by a factor specified in **Sample rate change** parameter
 - **Interpolation :** The data rate of the output is faster than the input by a factor specified in **Sample rate change** parameter
- **Sample rate change:** Allows user to specify the factor by which the sample rate of the output changes compared to input. This parameter is only active when the **Filter Type** is set to either **Decimation** or **Interpolation**. When **Decimation** is selected the sample rate of the output decreases with respect to the input by a factor equal to **Sample rate change**. When interpolation is selected the sample rate of the output increases with respect to the input by a factor equal to **Sample rate change**.
- **Number of channels:** The number of data channels to be processed by the Fir Compiler block. The multiple channel data is passed to the core in a time multiplexed manner. A maximum of 16 channels is supported.

- **Hardware over-sampling rate:** Specifies the ratio between the faster of input data rate and the output data rate versus the internal core rate. The overclocking rate determines the number of filter taps folded onto each DSP48/DSP48E in the filter.
- **Coefficient vector:** Specifies the coefficient vector as a single Matlab row vector. The number of taps is inferred from the length of the Matlab row vector. The number of coefficients in a filter configuration, the specified coefficient structure and the hardware over-sampling factor determine the number of DSP48s/DSP48Es instanced by the filter. It is possible to enter these coefficients using the FDATool block as well.
- **Coefficients Structure:** Specifies the coefficient structure. Depending on the coefficient structure, optimizations are made in the core to reduce the amount of hardware required to implement a particular filter configuration. The selected structure can be any of the following:
 - Inferred from coefficients
 - Non-Symmetric
 - Symmetric
 - Half Band

The vector of coefficients specified must match the structure specified unless **Inferred from Coefficients** is selected in which case the structure is determined automatically from these coefficients.

- **Use reloadable coefficients:** Enables the ability to reload filter coefficients at run-time.
- **Display filter information on block icon:** (Number of taps, type of filter, etc. is shown on block icon).

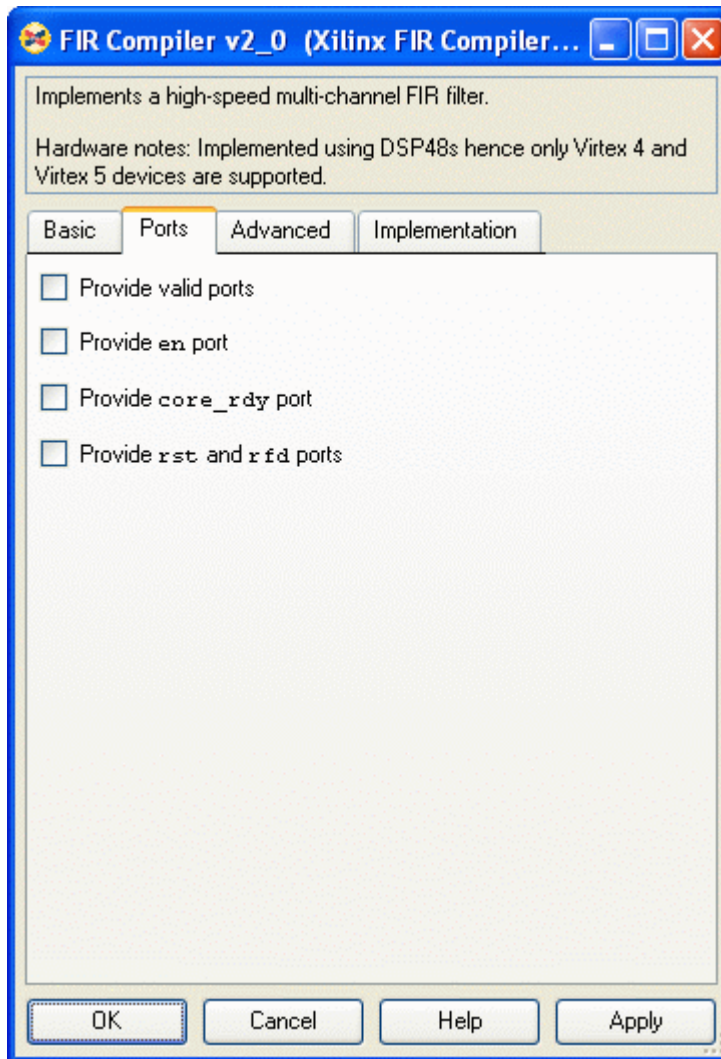


Figure 6 FIR Compiler v2_0 Ports tab

Parameters specific to the FIR Compiler block on the Ports tab are:

- **Provide valid ports(Ports Tab):** Provides vin and vout ports on the block.
- **Provide core_rdy port(Ports Tab):** Provides an output port on the block which is low initially for a number of cycles equal to the start up latency of the core. After this signal is driven high the core has completed its initialisation sequence and is able to accept input data. Refer to the data sheet for more information on the startup latency.
- **Provide rst and rfd ports(Ports Tab):** Provides rst and rfd ports on the block.

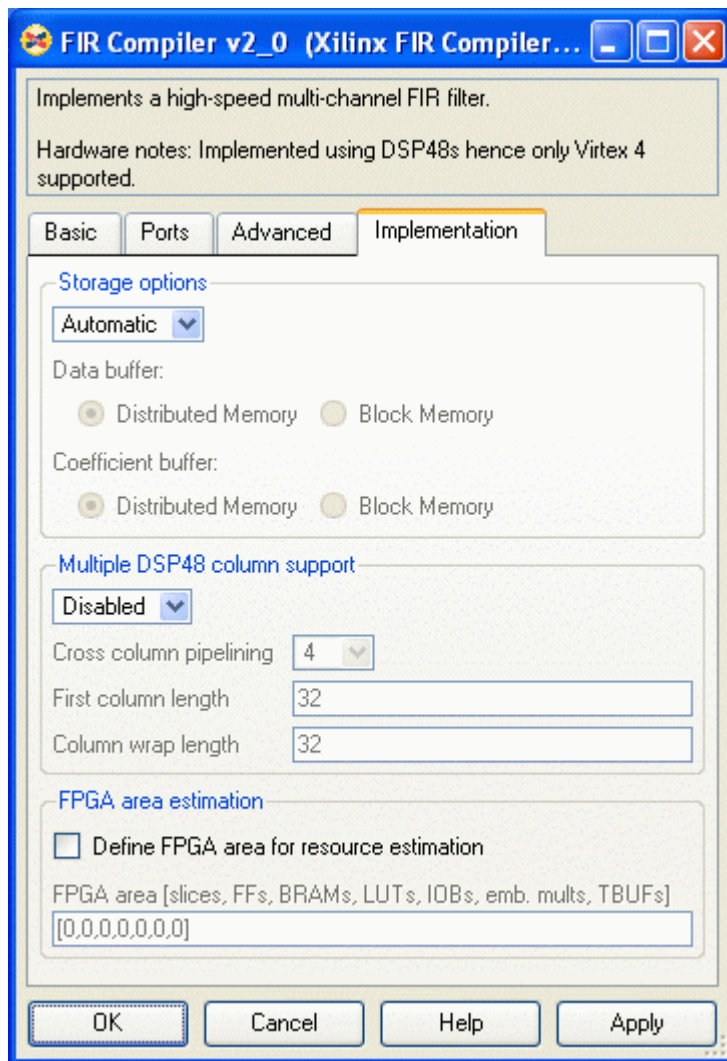


Figure 7 FIR Compiler v2_0 Implementation tab

Parameters specific to the FIR Compiler block on the Implementation tab are:

- **Storage options:** Allows custom memory options to be selected for the storage of both coefficients and data.
- **Data buffer:** Specifies the type of memory used to store data samples.
- **Coefficient buffer:** Specifies the type of memory used to store the coefficients.
- **Multiple DSP48 column support:** Specifies the method for implementing filters over multiple columns of DSP48/DSP48E slices in a Virtex-4/Virtex-5 device. **Custom** mode allows the user to specify an exact multi-column implementation and setting this control to **disabled** turns off all support for multi-column implementations.
- **First column length:** Specifies the length of the first column in a multi-column DSP48/DSP48E filter implementation. This control is only active when **Multiple DSP48 column support** is set to **custom**.
- **Cross column pipelining:** Specifies the length of the pipelines between columns in a multi-column DSP48/DSP48E filter implementation. This control is only active when **Multiple DSP48 column support** is set to **custom**.
- **Column wrap length:** Specifies the length of subsequent columns in a multi-column DSP48/DSP48E filter implementation. This value must be greater than or equal to the first column

length specified. This control is only active when Multiple DSP48 column support is set to custom.

Other parameters used by this block are explained in the Common Parameters section at the beginning of this chapter.

Xilinx LogiCORE

The block always uses the Xilinx LogiCORE FIR Compiler.

Known Issues

The following known issues are associated with certain features of Fir Compiler in SysGen:

- Currently the FIR Compiler block only supports MAC FIR for Virtex-4 and Virtex-5 devices.
- There is no automatic multi-column support
- Simulations speed may vary depending on configuration
- There is support only for a maximum of 512 taps
- If a design containing the block simulates but produces the error pattern reproduced below during netlist phase then refer to the coregen.log file located in \$target_directory/sysgen/coregen_XXXX/coregen_tmp:

```
standard exception: XNetlistEngine: An exception
was raised: com.xilinx.sysgen.netlist.f: ERROR:
coreutil - Failure to generate output products
at C:/MATLAB701/toolbox/xilinx/sysgen/scripts/
SgGenerateCores.pm
```

\$target_directory refers to the netlist target directory as specified on the SysGen token.

From FIFO

This block is listed in the following Xilinx Blockset libraries: *Index, Shared Memory*.



The Xilinx From FIFO block implements the trailing half of a first-in-first-out memory queue.

By asserting the read-enable input port `re`, data can be read from the FIFO via the data output port `dout`. The `empty` output port is asserted when the FIFO is empty. The percent full output port indicates the percentage of the FIFO that is full, represented with user-specified precision.

The From FIFO is implemented in hardware using the FIFO Generator v2.1 core. System Generator's hardware co-simulation interfaces allow the From FIFO block to be compiled and co-simulated in FPGA hardware. When used in System Generator co-simulation hardware, shared FIFOs facilitate high-speed transfers between the host PC and FPGA, and bolster the tool's real-time hardware co-simulation capabilities.

Block Parameters Dialog Box

The following figure shows the block parameters dialog box.

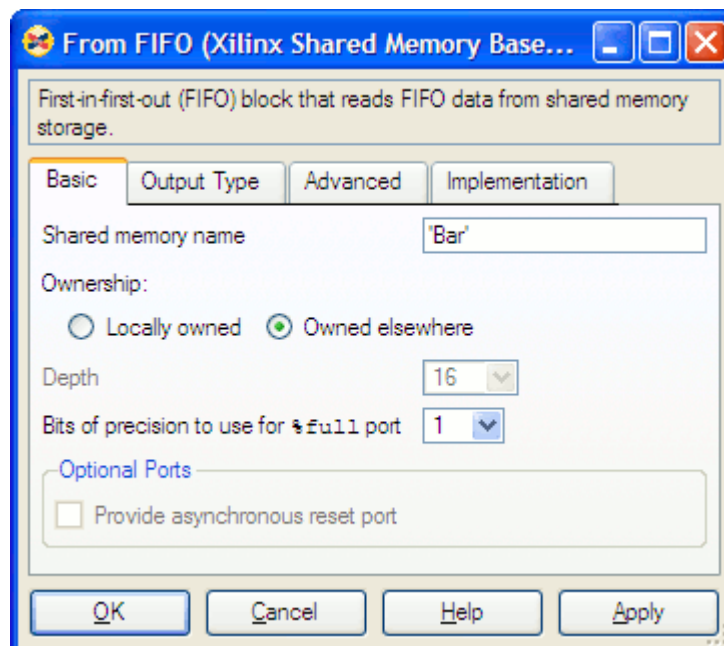


Figure 1: From FIFO block parameters dialog box

Parameters specific to the block are:

- **Shared memory name:** name of the shared FIFO. All FIFOs with the same name share the same physical FIFO.
- **Ownership:** indicates whether the memory is **Locally owned** or **Owned elsewhere**. A block that is **Locally owned** is responsible for creating an instance of the FIFO. A block that is **Owned elsewhere** attaches itself to a FIFO instance that has already been created.
- **Depth:** specifies the number of words in the memory. The word size is inferred from the bit width of the port `din`.

- **Bits of precision to use for %full port:** specifies the bit width of the %full port. The binary point for this unsigned output is always at the top of the word. Thus, for example, if precision is set to one, the output can take two values: 0.0 and 0.5, the latter indicating the FIFO is at least 50% full.

Other parameters used by this block are described in the Common Parameters section at the beginning of this chapter.

Xilinx LogiCORE

The block uses the Xilinx LogiCORE FIFO Generator.

See Also

The following documents are provided as part of the System Generator User's guide and give valuable insight into using and understanding the From FIFO block:

To FIFO
Multiple Subsystem Generator
Hardware Co-Simulation

From Register

This block is listed in the following Xilinx Blockset libraries: *Index*.



The Xilinx From Register block implements the trailing half of a D flip-flop based register. The physical register can be shared among two designs or two portions of the same design.

The block reads data from a register that is written to by the corresponding To Register block. The dout port presents the output of the register. The bit width specified on the mask must match the width of the corresponding To Register

block.

Block Parameters Dialog Box

The following figure shows the block parameters dialog box.

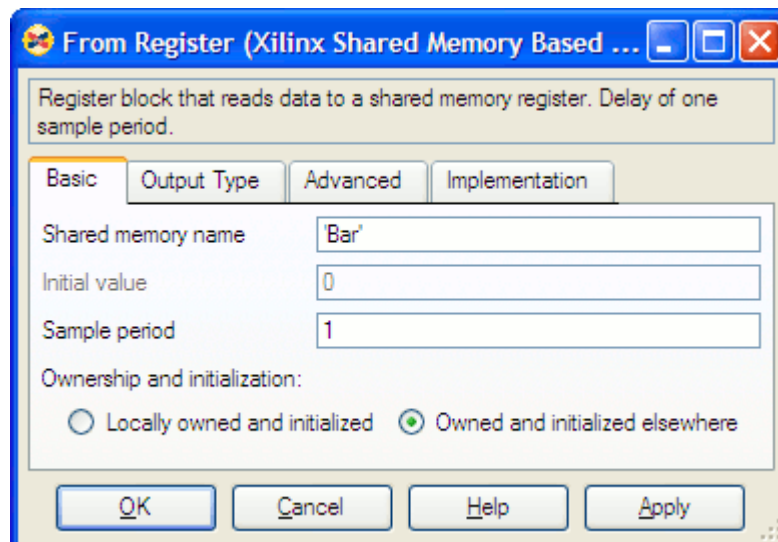


Figure 1: From Register block parameters dialog box

Parameters specific to the block are:

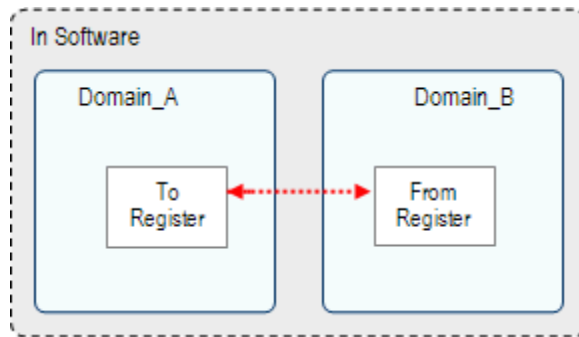
- **Shared Memory Name:** name of the shared register. There must be exactly one To Register and exactly one From Register block for a particular register name. In addition, the name must be distinct from all other shared memory names in the design.
- **Initial value:** specifies the initial value in the register.
- **Ownership and initialization:** indicates whether the register is **Locally owned and initialized** or **Owned and initialized elsewhere**. A block that is locally owned is responsible for creating an instance of the register. A block that is owned elsewhere attaches itself to a register instance that has already been created. As a result, if two shared register blocks are used in two different models during simulation, the model containing the locally owned block has to be started first.

Other parameters used by this block are described in the Common Parameters section at the beginning of this chapter.

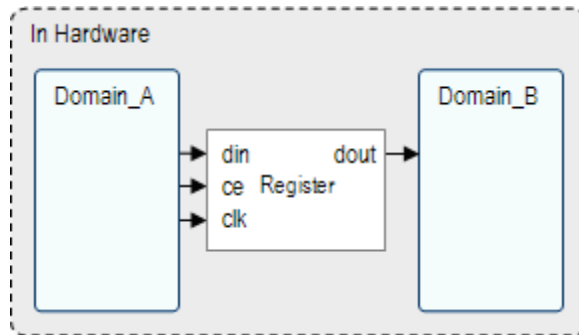
Crossing Clock Domains

When a To Register and From Register block pair are used to cross clock domain boundaries, a single

register is implemented in hardware. This register is clocked by the To Register block clock domain. For example, assume a design has two clock domains, `Domain_A` and `Domain_B`. Also assume that a shared register pair are used to cross the two clock domains shown below.



When the design is generated using the Multiple Subsystem Generator block, only one register is included in the design. The clock and clock enable register signals are driven from the `Domain_A` domain.



Crossing domains in this manner may be unsafe. To reduce the chance of metastability, include two Register blocks immediately following the From Register block to resynchronize the data to the From Register's clock domain.

See Also

The following documents are provided as part of the System Generator User's guide and give valuable insight into using and understanding the From Register block:

To Register
Multiple Subsystem Generator
Hardware Co-Simulation

Gateway In

This block is listed in the following Xilinx Blockset libraries: Basic Elements, Data Types, and Index.



Gateway In Xilinx Gateway In blocks are the inputs into the Xilinx portion of the user's Simulink design. These blocks convert Simulink integer, double and fixed point data types into the System Generator fixed point type. Each block defines a top-level input port in the HDL design generated by System Generator.

While converting a double type to a System Generator fixed point type, the Gateway In uses the selected overflow and quantization options. For overflow, the options are to saturate to the largest positive/smallest negative value, to wrap (i.e., to discard bits to the left of the most significant representable bit), or to flag an overflow as a Simulink error during simulation. For quantization, the options are to round to the nearest representable value (or to the value furthest from zero if there are two equidistant nearest representable values), or to truncate (i.e., to discard bits to the right of the least significant representable bit).

It is important to realize that overflow and quantization do not take place in hardware – they take place in the block software itself, before entering the hardware phase.

Gateway Blocks

The Xilinx Gateway In block has several functions:

- Converting data from Simulink integer, double and fixed point types to the System Generator fixed point type during simulation in Simulink.
- Defining top-level input ports in the HDL design generated by System Generator.
- Defining testbench stimuli when the **Create Testbench** box is checked in the System Generator block. In this case, during HDL code generation, the inputs to the block that occur during Simulink simulation are logged as a logic vector in a data file. During HDL simulation, an entity that is inserted in the top level testbench checks this vector and the corresponding vectors produced by Gateway Out blocks against expected results.
- Naming the corresponding port in the top level HDL entity.

The block parameters dialog box can be invoked by double-clicking the icon in the Simulink model.

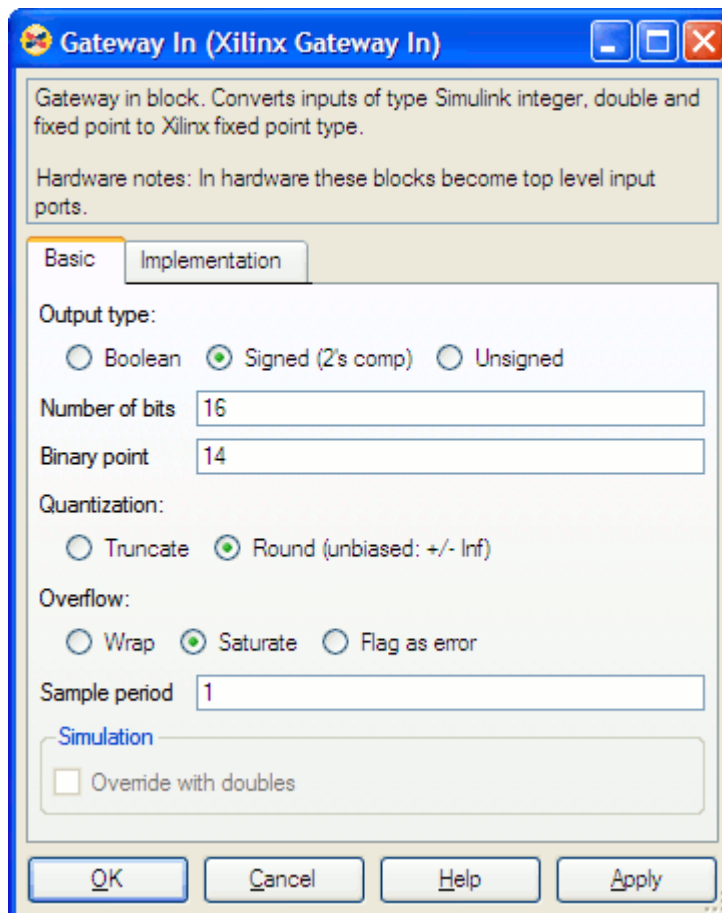


Figure 1(a): Gateway In block parameters dialog box (showing Basic parameters)

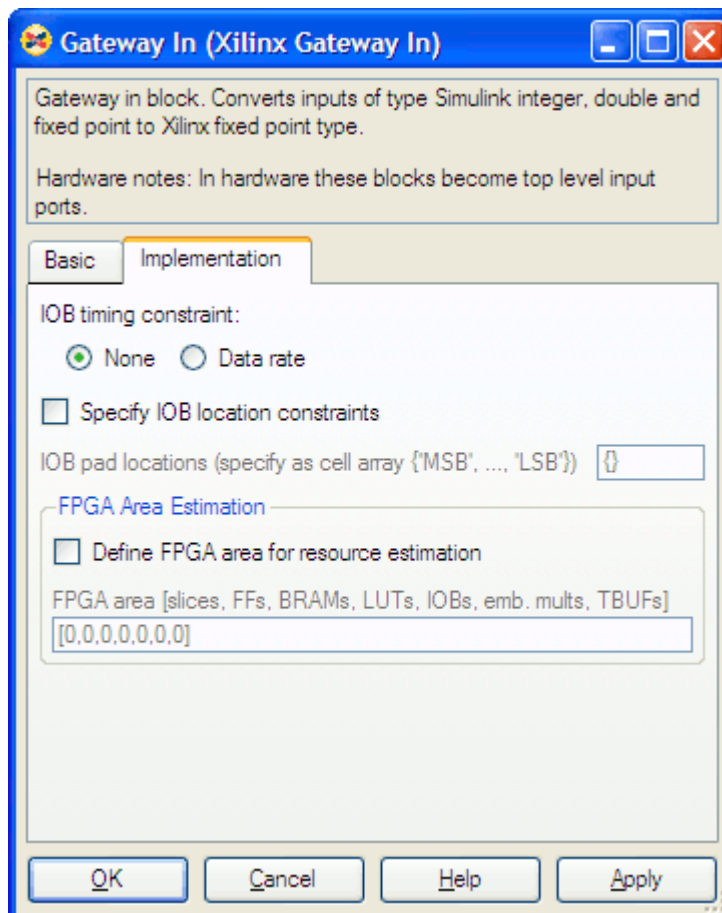


Figure 2(b): Gateway In block parameters dialog box (showing Implementation Parameters)

Parameters specific to the Gateway In block are:

- **IOB Timing Constraint:** In hardware, a Gateway In is realized as a set of input/output buffers (IOBs). There are two ways to constrain the timing on IOBs. They are **None** and **Data Rate**.

If **None** is selected, no timing constraints for the IOBs are put in the constraint file (.xcf if using the XST synthesis tool, .ncf otherwise) produced by System Generator. This means the paths from the IOBs to synchronous elements are not constrained.

If **Data Rate** is selected, the IOBs are constrained at the data rate at which the IOBs operate. The rate is determined by the **System Clock Period** field in the System Generator block and the sample rate of the Gateway relative to the other sample periods in the design. For example, the following OFFSET = IN constraints are generated for a Gateway In named 'Din' that is running at the system period of 10 ns:

```
# Offset in constraints
NET "Din(0)" OFFSET = IN : 10.0 : BEFORE "clk";
NET "Din(1)" OFFSET = IN : 10.0 : BEFORE "clk";
NET "Din(2)" OFFSET = IN : 10.0 : BEFORE "clk";
```

- **Specify IOB Location Constraints:** When this box is checked, a new edit box appears that allows the user to specify IOB location constraints, discussed below.
- **IOB Pad Locations, e.g. {'MSB', ..., 'LSB'}:** IOB pin locations can be specified as a cell array of strings in this edit box. The locations are package-specific. For the above example, if a Virtex-E 2000 in a FG680 package is used, the location constraints for the Din bus can be specified in the dialog box as {'C36', 'B36', 'D35'}. This is translated into constraints in the .xcf (or .ncf) file in the following way:

```
# Loc constraints
NET "Din(0)" LOC = "D35";
NET "Din(1)" LOC = "B36";
NET "Din(2)" LOC = "C35";
```

Other parameters used by this block are described in the Common Parameters section at the beginning of this chapter. However, the Gateway In block, as opposed to other blocks, will not use extra hardware resources when selecting **Round** for the **Quantization** field or **Saturate** for the **Overflow** field

Gateway Out

This block is listed in the following Xilinx Blockset libraries: Basic Elements, Data Types, and Index.



Xilinx Gateway Out blocks are the outputs from the Xilinx portion of the user's Simulink design. This block converts the System Generator fixed point data type into Simulink Double.

According to its configuration, the Gateway Out block can either define an output port for the top level of the HDL design generated by System Generator, or be used simply as a test point that will be trimmed from the hardware representation.

Gateway Blocks

The Xilinx Gateway Out blocks have several functions:

- Converting data from System Generator fixed point type to Simulink double.
- Defining I/O ports for the top level of the HDL design generated by System Generator. A Gateway Out block defines a top level output port.
- Defining testbench result vectors when the System Generator Create Testbench box is checked. In this case, during HDL code generation, the outputs from the block that occur during Simulink simulation are logged as logic vectors in a data file. For each top level port, an HDL component is inserted in the top level testbench that checks this vector against expected results during HDL simulation.
- Naming the corresponding output port on the top level HDL entity.

The block parameters dialog box can be invoked by double-clicking the icon in the Simulink model.

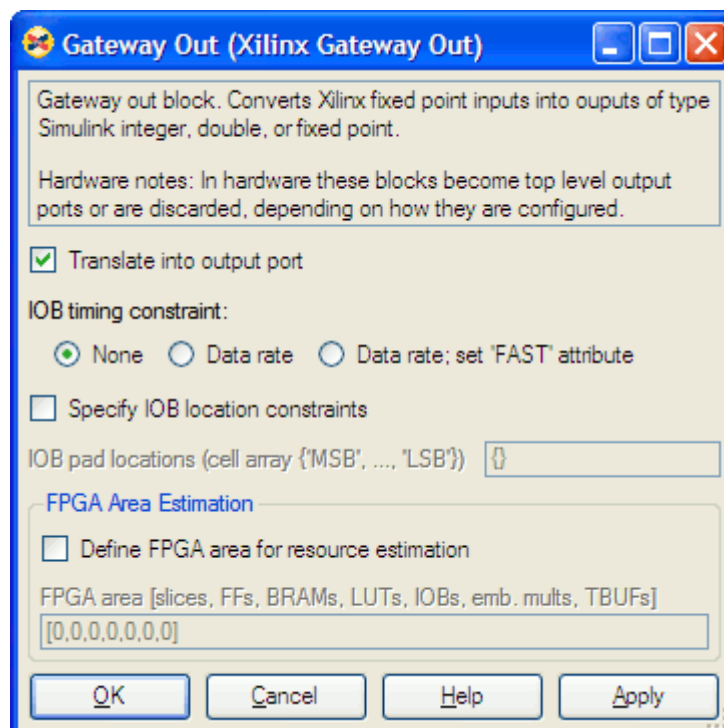


Figure 1: Gateway Out block parameters dialog box

Parameters specific to the Gateway Out block are:

- **Translate into Output Port:** Having this box unchecked prevents the gateway from becoming an actual output port when translated into hardware. This checkbox is on by default, enabling the output port. When this option is not selected, the Gateway Out block is used only during debugging, where its purpose is to communicate with Simulink Sink blocks for probing portions of the design. In this case, the Gateway Out block will turn gray in color, indicating that the gateway will not be translated into an output port.
- **IOB Timing Constraint:** In hardware, a Gateway Out is realized as a set of input/output buffers (IOBs). There are three ways to constrain the timing on IOBs. They are None, Data Rate, and Data Rate, Set 'FAST' Attribute.

If **None** is selected, no timing constraints for the IOBs are put in the user constraint file (.xcf if using the XST synthesis tool, .ncf otherwise) produced by System Generator. This means the paths from the IOBs to synchronous elements are not constrained.

If **Data Rate** is selected, the IOBs are constrained at the data rate at which the IOBs operate. The rate is determined by System Clock Period provided on the System Generator block and the sample rate of the Gateway relative to the other sample periods in the design. For example, the following OFFSET = OUT constraints are generated for a Gateway Out named 'Dout' that is running at the system period of 10 ns:

```
# Offset out constraints
NET "Dout(0)" OFFSET = OUT : 10.0 : AFTER "clk";
NET "Dout(1)" OFFSET = OUT : 10.0 : AFTER "clk";
NET "Dout(2)" OFFSET = OUT : 10.0 : AFTER "clk";
```

If **Data Rate, Set 'FAST' Attribute** is selected, the OFFSET = OUT constraints described above are produced. In addition, a FAST slew rate attribute is generated for each IOB. This reduces delay but increases noise and power consumption. For the previous example, the following additional attributes are added to the .xcf (or .ncf) file

```
NET "Dout(0)" FAST;
NET "Dout(1)" FAST;
NET "Dout(2)" FAST;
```

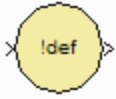
- **Specify IOB Location Constraints:** Checking this option allows IOB location constraints to be specified.
- **IOB Pad Locations, e.g. {'MSB', ..., 'LSB'}:** IOB pin locations can be specified as a cell array of strings in this edit box. The locations are package-specific. For the above example, if a Virtex-E 2000 in a FG680 package is used, the location constraints for the Dout bus can be specified in the dialog box as {'B34', 'D33', 'B35'}. This is translated into constraints in the .xcf (or .ncf) file in the following way:

```
# Loc constraints
NET "Dout(0)" LOC = "B35";
NET "Dout(1)" LOC = "D33";
NET "Dout(2)" LOC = "B34";
```

Other parameters used by this block are described in the Common Parameters section at the beginning of this chapter.

Indeterminate Probe

This block is listed in the following Xilinx Blockset libraries: Tools and Index.



The output of the Xilinx Indeterminate Probe indicates whether the input data is indeterminate (MATLAB value NaN). An indeterminate data value corresponds to a VHDL indeterminate logic data value of 'X'. The probe accepts any Xilinx signal as input and produces a double signal as output. Indeterminate data on the probe input will result in an assertion of the output signal indicated by a value one. Otherwise, the probe output is zero.

Interleaver Deinterleaver v4_0

This block is listed in the following Xilinx Blockset libraries: *Communication and Index*.



The Xilinx Interleaver/Deinterleaver block implements an interleaver or a deinterleaver. An interleaver is a device that rearranges the ordering of a sequence of symbols in a one-to-one deterministic manner. Associated with any interleaver is a deinterleaver, a device that restores the reordered sequence.

When the block is in interleaver mode, the input data sampled on the `din` port shall be multiplexed into and out of B shift registers onto the `dout` port using two (synchronized) commutator arms, as illustrated in the figure below. B is the number of branches as entered in the block's parameters dialog. Branch 0 shall have a shift register of zero length. Branch 1 shall have a shift register of length L . Branch 2 shall have a shift register of length $2L$. Branch $(B-1)$ shall have a shift register of length $(B-1)L$. L is the branch length constant entered as an array with a length of one.

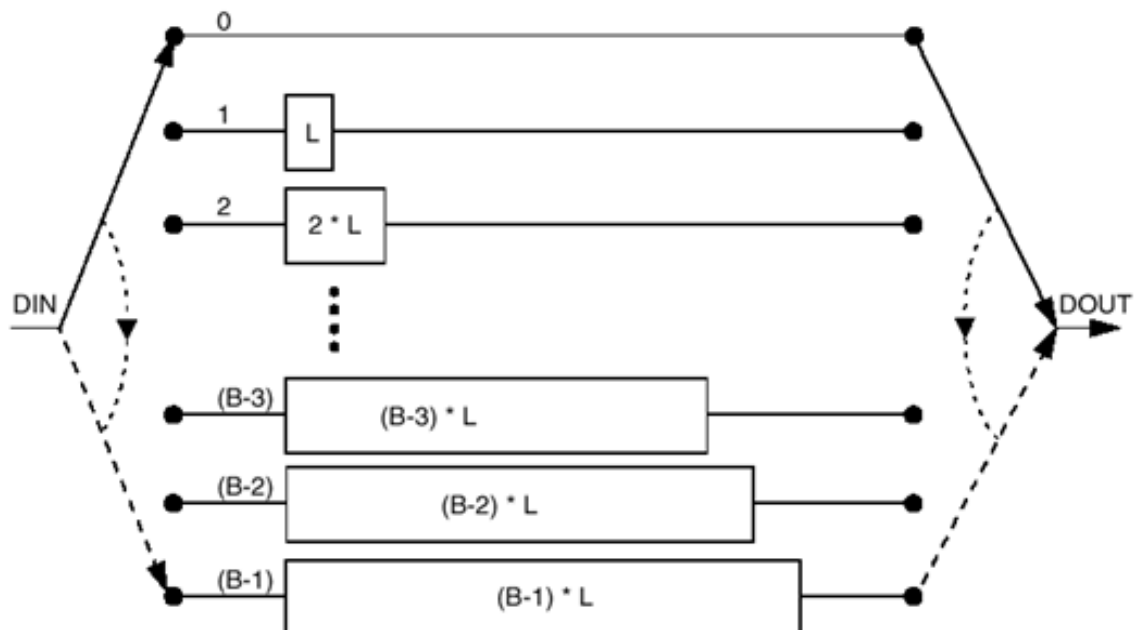


Figure 1: Forney convolutional interleaver with a constant difference between consecutive branches

When the block is in deinterleaver mode, the input data sampled on the `din` port is multiplexed into and out of B shift registers onto the `dout` port using two (synchronized) commutator arms. Branch 0 will have a shift register of length $(B-1)L$. Branch $(B-1)$ shall have a shift register length of zero.

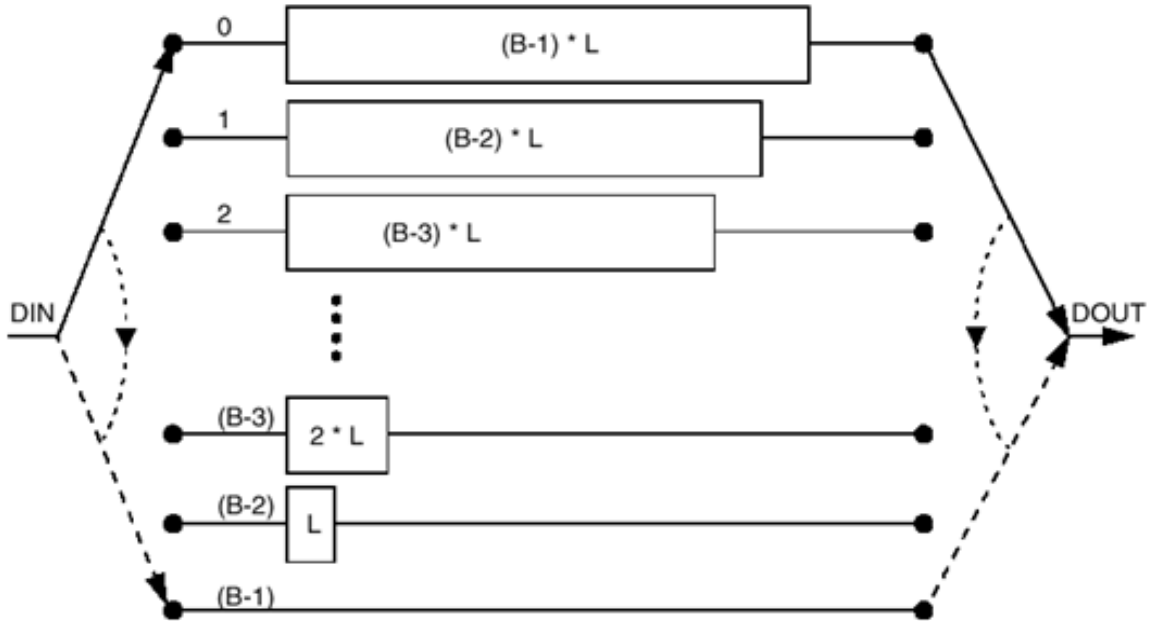


Figure 2: Forney convolutional deinterleaver with a constant difference between consecutive branches

When the branch lengths are specified as an array, the block operates the same in either interleaver or deinterleaver mode because the array fully defines the length of all the branches. The array must have length B , matching the number of branches.

The reset pin (`rst`) will set the commutator arms to branch 0, but will not clear the branches of data.

Block Interface

The Interleaver/Deinterleaver block has two to four input and two output ports. The input port, `din`, must be between 1 and 256 (inclusive) bits. The `vin` port indicates that the values presented on the `din` port are valid. Only valid data is multiplexed into and out of the shift registers. The `vout` port indicates that the values presented on the `dout` port are valid. The size of the output port, `dout`, is the same as the input port, `din`.

Block Parameters Dialog Box

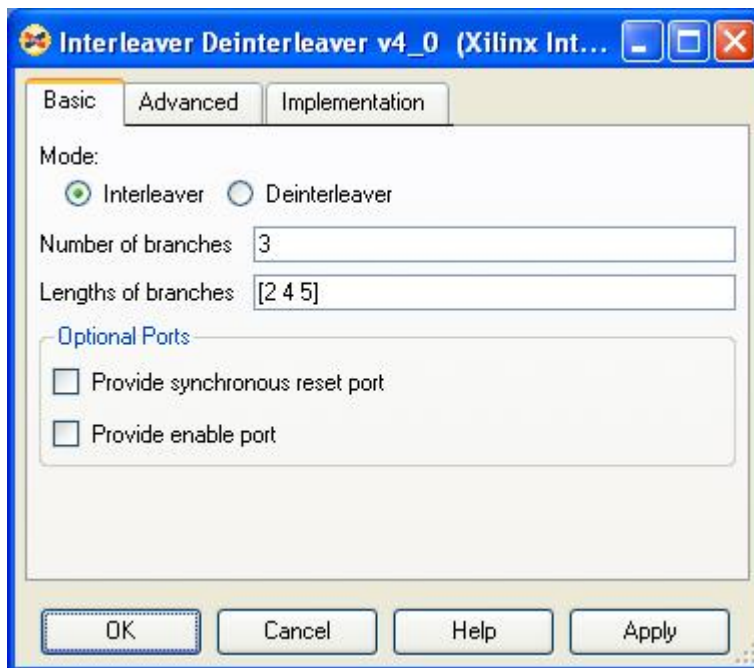


Figure 3: Interleaver/Deinterleaver basic tab parameters

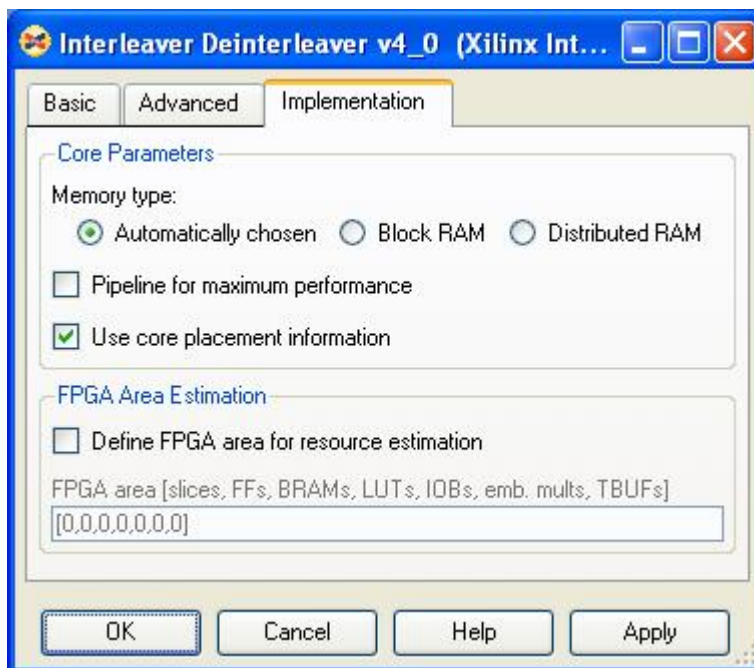


Figure 4: Interleaver/Deinterleaver implementation tab parameters

Parameters specific to the block are:

- **Mode:** Interleaver or Deinterleaver
- **Number of branches:** 1 to 256 (inclusive)
- **Lengths of branches:** 1 to MAX (inclusive). MAX depends on the number of branches and size of core input. Branch length must be an array of either length one or number of branches. If the array size is one, the value is used as a constant difference between consecutive branches. Otherwise, each branch has a unique length.

- **Memory type:** Automatically chosen, block RAM or distributed RAM.
- **Pipeline for maximum performance:** pipeline the core.

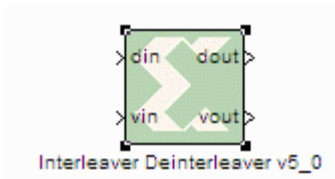
Other parameters used by this block are described in the Common Parameters section at the beginning of this chapter.

Xilinx LogiCORE

The block uses the Xilinx LogiCORE Interleaver/De-interleaver.

Interleaver Deinterleaver v5_0

This block is listed in the following Xilinx Blockset libraries: *Communication and Index*.



The Xilinx Interleaver/Deinterleaver block implements an interleaver or a deinterleaver. An interleaver is a device that rearranges the ordering of a sequence of symbols in a one-to-one deterministic manner. Associated with any interleaver is a deinterleaver, a device that restores the reordered sequence.

When the block is in interleaver mode, the input data sampled on the `din` port shall be multiplexed into and out of B shift registers onto the `dout` port using two (synchronized) commutator arms, as illustrated in the figure below. B is the number of branches as entered in the block's parameters dialog. Branch 0 shall have a shift register of zero length. Branch 1 shall have a shift register of length L . Branch 2 shall have a shift register of length $2L$. Branch $(B-1)$ shall have a shift register of length $(B-1)L$. L is the branch length constant entered as an array with a length of one.

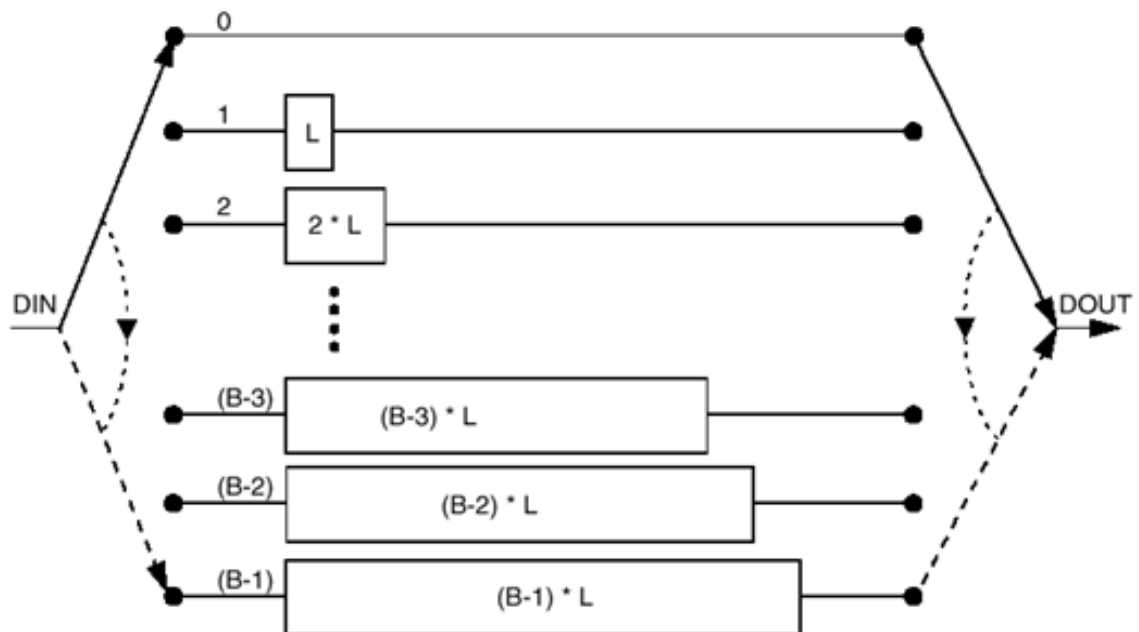


Figure 1: Forney convolutional interleaver with a constant difference between consecutive branches

When the block is in deinterleaver mode, the input data sampled on the `din` port is multiplexed into and out of B shift registers onto the `dout` port using two (synchronized) commutator arms. Branch 0 will have a shift register of length $(B-1)*L$. Branch $(B-1)$ shall have a shift register length of zero.

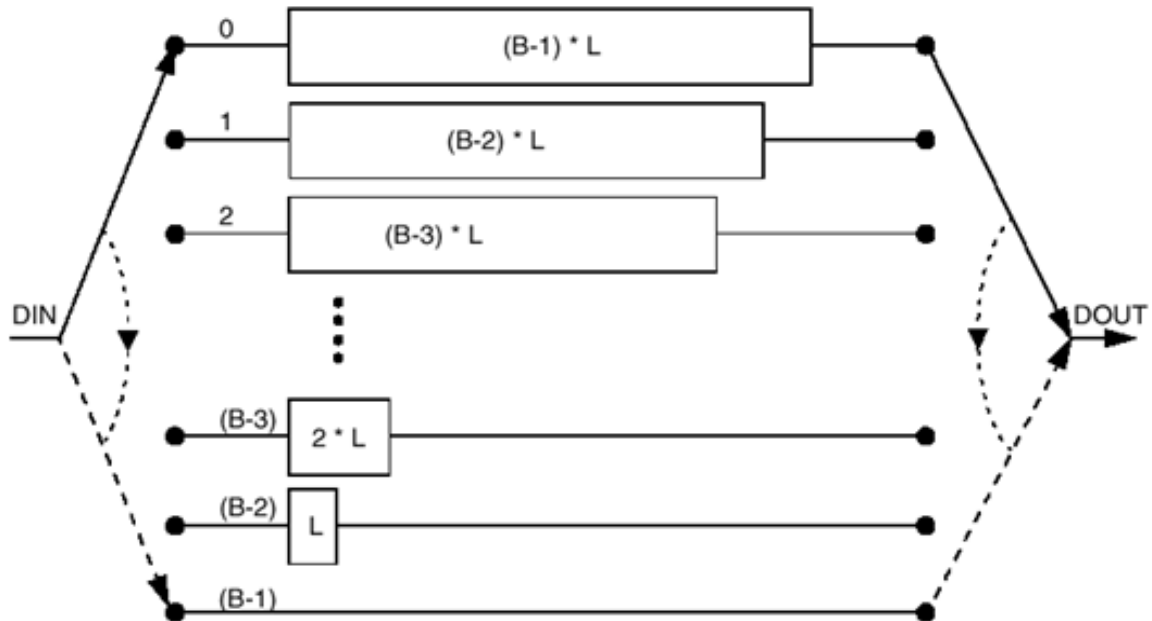


Figure 2: Forney convolutional deinterleaver with a constant difference between consecutive branches

When the branch lengths are specified as an array, the block operates the same in either interleaver or deinterleaver mode because the array fully defines the length of all the branches. The array must have length B , matching the number of branches.

The reset pin (`rst`) will set the commutator arms to branch 0, but will not clear the branches of data.

Block Interface

The Interleaver/Deinterleaver block has two to four input and two output ports. The input port, `din`, must be between 1 and 256 (inclusive) bits. The `vin` port indicates that the values presented on the `din` port are valid. Only valid data is multiplexed into and out of the shift registers. The `vout` port indicates that the values presented on the `dout` port are valid. The size of the output port, `dout`, is the same as the input port, `din`.

Block Parameters Dialog Box

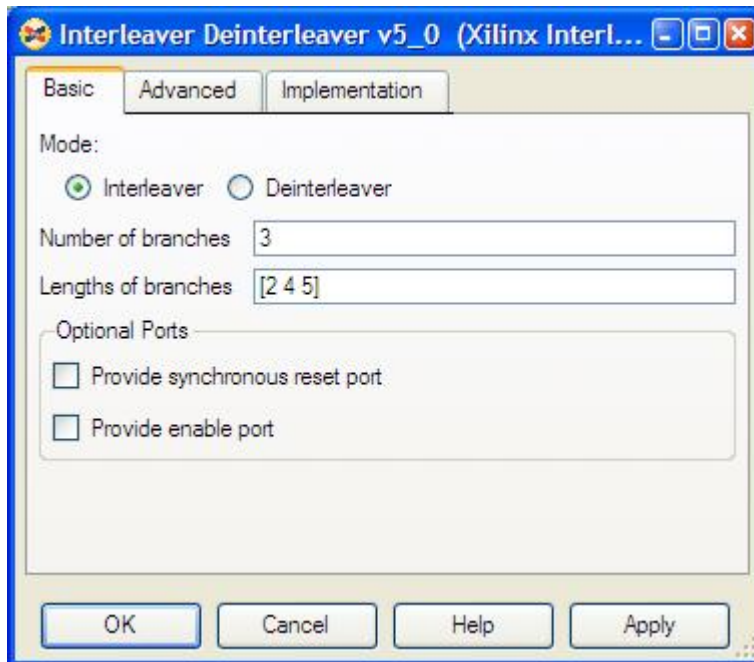


Figure 3: Interleaver/Deinterleaver basic tab parameters

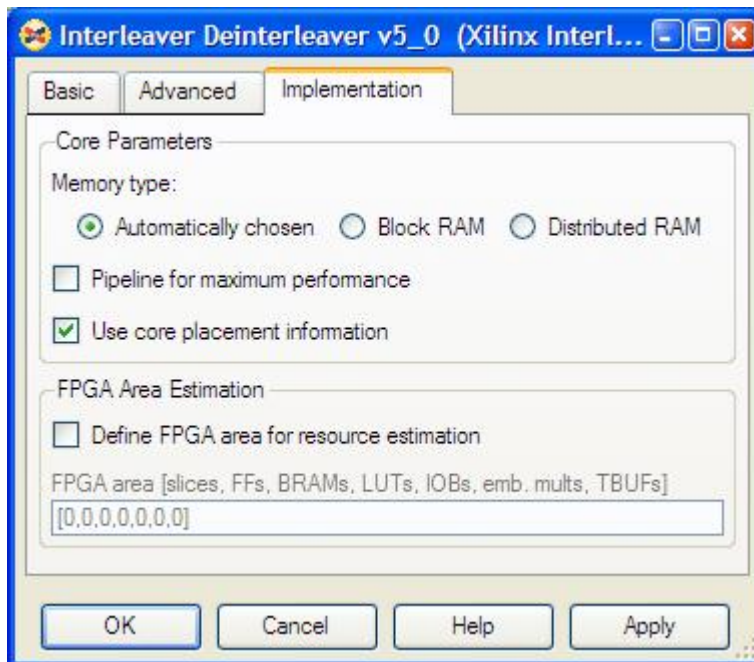


Figure 4: Interleaver/Deinterleaver implementation tab parameters

Parameters specific to the block are:

- **Mode:** Interleaver or Deinterleaver
- **Number of branches:** 1 to 256 (inclusive)
- **Lengths of branches:** 1 to MAX (inclusive). MAX depends on the number of branches and size of core input. Branch length must be an array of either length one or number of branches. If the array size is one, the value is used as a constant difference between consecutive branches. Otherwise, each branch has a unique length.

- **Memory type:** Automatically chosen, block RAM or distributed RAM.
- **Pipeline for maximum performance:** pipeline the core.

Other parameters used by this block are described in the Common Parameters section at the beginning of this chapter.

Xilinx LogiCORE

The block uses the Xilinx LogiCORE Interleaver/De-interleaver.

Inverter

This block is listed in the following Xilinx Blockset libraries: Basic Elements, Control Logic, Math, and Index.



The Xilinx Inverter block calculates the bitwise logical complement of a fixed point number. The block is implemented as a synthesizable VHDL module.

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

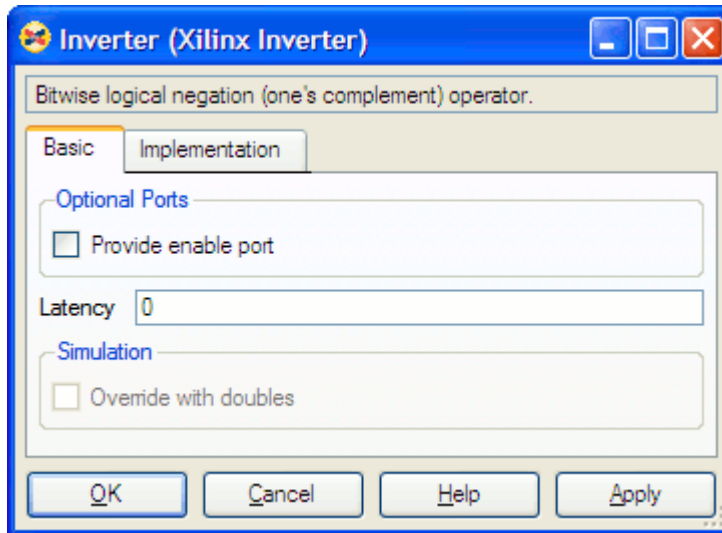
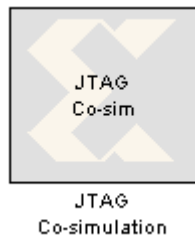


Figure 1: Inverter block basic tab parameters

Parameters used by this block are explained in the Common Parameters section at the beginning of this chapter.

JTAG Co-Simulation



The Xilinx JTAG Co-Simulation block allows you to perform hardware co-simulation using JTAG and a Parallel Cable IV or Platform USB. The JTAG hardware co-simulation interface takes advantage of the ubiquity of JTAG to extend System Generator's hardware in the simulation loop capability to numerous other FPGA platforms.

The port interface of the co-simulation block varies. When a model is implemented for JTAG hardware co-simulation, a new library is created that contains a custom JTAG co-simulation block with ports that match the gateway names (or port names if the subsystem is not the top level) from the original model. The co-simulation block interacts with the FPGA hardware platform during a Simulink simulation. Simulation data that is written to the input ports of the block are passed to the hardware by the block. Conversely, when data is read from the co-simulation block's output ports, the block reads the appropriate values from the hardware and drives them on the output ports so they can be interpreted in Simulink. In addition, the block automatically opens, configures, steps, and closes the platform.

Refer to JTAG Hardware Co-Simulation for JTAG hardware requirements, and information on how to support new platforms.

Block Parameters Dialog Box

The dialog box tabs specific to the JTAG hardware co-simulation block are shown below.

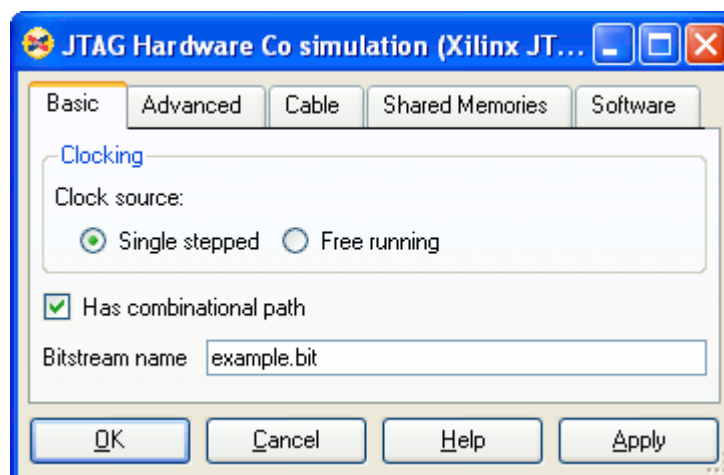


Figure Error! No text of specified style in document.-1: JTAG Co-Simulation Block Basic Tab

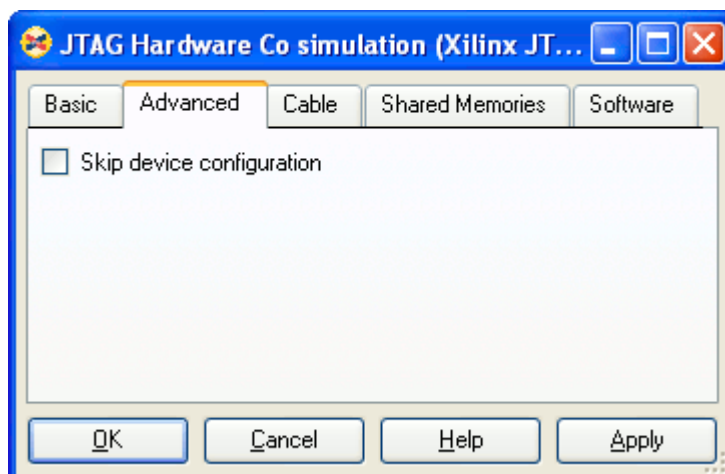


Figure Error! No text of specified style in document.-2: JTAG Co-Simulation Block Advanced Tab

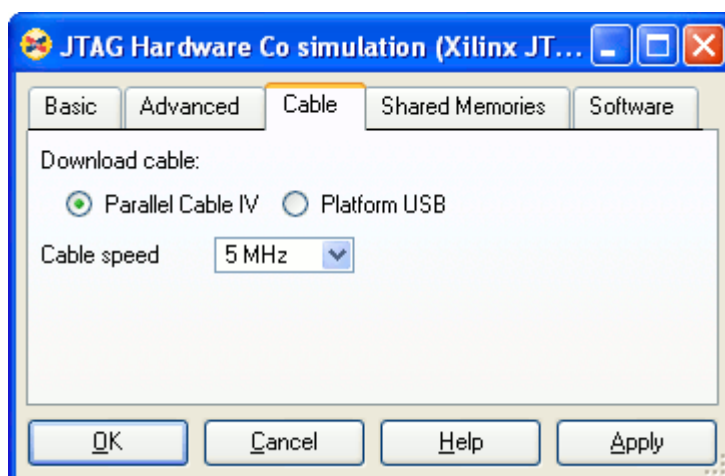


Figure Error! No text of specified style in document.-3: JTAG Co-Simulation Block Cable Tab

Parameters specific to the block are:

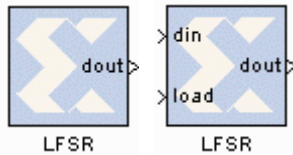
- **Clock source (Basic):** You may select between **Single stepped** and **Free running** clock sources. Selecting a **Single stepped** clock allows the block to step the board one clock cycle at a time. Each clock cycle step corresponds to some duration of time in Simulink. Using this clock source ensures the behavior of the co-simulation hardware during simulation will be bit and cycle accurate when compared to the simulation behavior of the subsystem from which it originated. Sometimes single stepping is not necessary and the board can be run with a **Free Running** clock. In this case, the board will operate asynchronously to the Simulink simulation.
- **Has combinational path (Basic):** Sometimes it is necessary to have a direct combinational feedback path from an output port on a hardware co-simulation block to an input port on the same block (e.g., a wire connecting an output port to an input port on a given block). If you require a direct feedback path from an output to input port, and your design does not include a combinational path from any input port to any output port, unchecking this box will allow the feedback path in the design.
- **Bitstream name (Basic):** Specifies the co-simulation FPGA configuration file for the JTAG hardware co-simulation platform. When a new co-simulation block is created during compilation, this parameter is automatically set so that it points to the appropriate configuration file. You need only adjust this parameter if the location of the configuration file changes.
- **Skip device configuration (Advanced):** Selecting this option causes the co-simulation block to skip the device configuration phase at the beginning of a simulation. Doing so

is useful for co-simulation designs that do not need to be reset (or reprogrammed) at the end of a simulation. This checkbox should be used with caution since the co-simulation platform is not programmed when this checkbox is selected. This means that it is possible to perform hardware co-simulation without a co-simulation bitstream loaded on the hardware platform.

- **Download cable (Cable):** You may select between **Parallel Cable IV** and **Platform USB** programming cables for performing JTAG hardware co-simulation.
- **Cable speed (Cable):** Sometimes you may need to run the programming cable at a frequency less than the default (maximum) speed setting for hardware co-simulation. This menu allows you to choose a cable speed that is suitable for your hardware setup. Normally the default speed will suffice, however, it is recommended to try a slower cable speed if System Generator fails to configure the device for co-simulation.

LFSR

This block is listed in the following Xilinx Blockset libraries: *Basic Elements, DSP, Memory, and Index.*



The Xilinx LFSR block implements a Linear Feedback Shift Register (LFSR). This block supports both the Galois and Fibonacci structures using either the XOR or XNOR gate and allows a reloadable input to change the current value of the register at any time. The LFSR output and reloadable input can be configured as either serial or parallel ports.

Block Interface

Port Name	Port Description	Port Type
din	Data input for reloadable seed	Optional serial or parallel input
load	Load signal for din	Optional boolean input
rst	Reset signal	Optional boolean input
en	Enable signal	Optional boolean input
dout	Data output of LFSR	Required serial or parallel output

Figure 1: Block interface for LFSR

As shown in the table above, there can be between 0 and 4 block input ports and exactly one output port. If the configuration selected requires 0 inputs, the LFSR will be set up to start at a specified initial seed value and will step through a repeatable sequence of states determined by the LFSR structure type, gate type and initial seed.

The optional `din` and `load` ports provide the ability to change the current value of the LFSR at runtime. After the load completes, the LFSR will behave as with the 0 input case and start up a new sequence based upon the newly loaded seed and the statically configured LFSR options for structure and gate type.

The optional `rst` port will reload the statically specified initial seed of the LFSR and continue on as before after the `rst` signal goes low. And when the optional `en` port goes low, the LFSR will remain at its current value with no change until the `en` port goes high again.

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

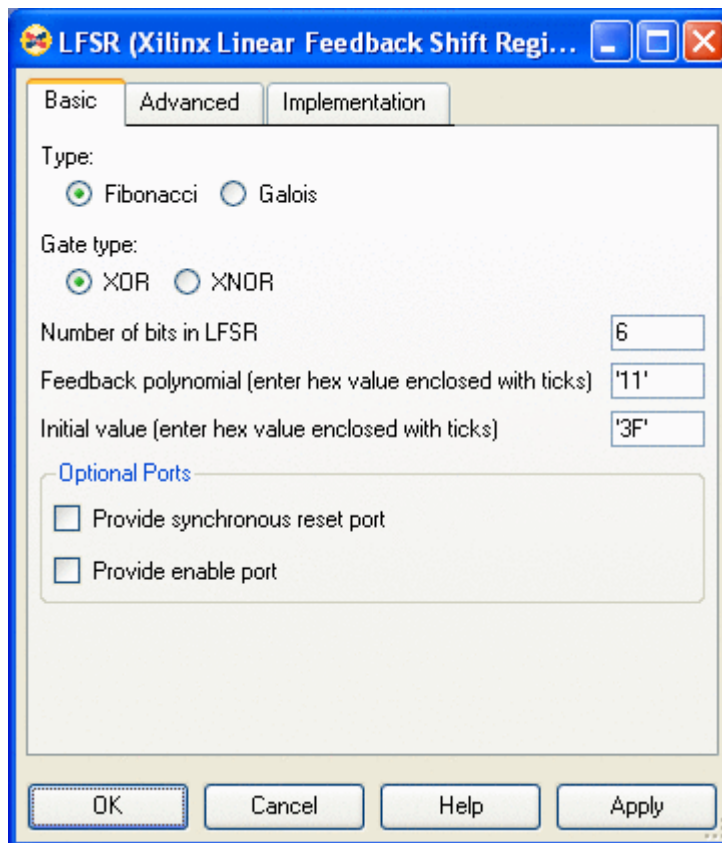


Figure 2: LFSR block Basic Tab

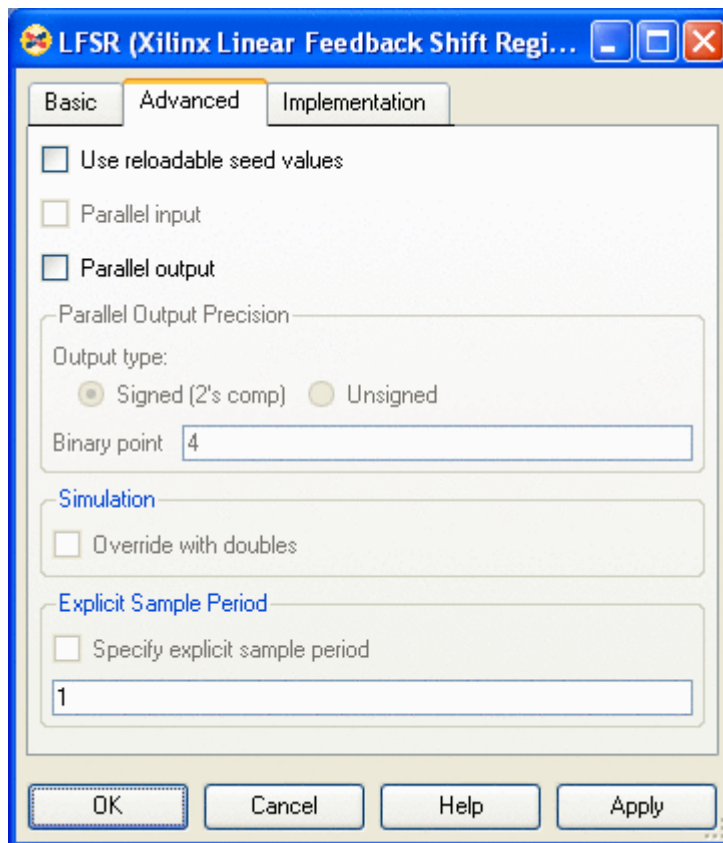


Figure 3: LFSR block Advanced Tab

Parameters specific to this block are:

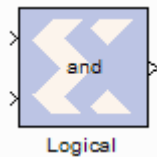
- **Type (Basic):** Fibonacci or Galois. This field specifies the structure of the feedback. Fibonacci has one XOR (or XNOR) gate at the beginning of the register chain that XORs (or XNORs) the taps together with the result going into the first register. Galois has one XOR (or XNOR) gate for each tap and gates the last register in the chains output with the input to the register at that tap.
- **Gate type (Basic):** XOR or XNOR. This field specifies the gate used by the feedback signals.
- **Number of bits in LFSR (Basic):** This field specifies the number of registers in the LFSR chain. As a result, this number specifies the size of the input and output when selected to be parallel.
- **Feedback polynomial (Basic):** This field specifies the tap points of the feedback chain and the value must be entered in hex with single quotes. The *lsb* of this polynomial always must be set to 1 and the *msb* is an implied 1 and is not specified in the hex input. Please see the LFSR core datasheet link shown below for more information on how to specify this equation and for optimal settings for the maximum repeating sequence.
- **Initial value (Basic):** This field specifies the initial seed value where the LFSR begins its repeating sequence. The initial value may not be all zeroes when choosing the XOR gate type and may not be all ones when choosing XNOR, as those values will stall the LFSR.
- **Parallel output (Advanced):** This field specifies whether all of the bits in the LFSR chain are connected to the output or just the last register in the chain (serial or parallel).
- **Use reloadable seed values (Advanced):** This field specifies whether or not an input is needed to reload a dynamic LFSR seed value at runtime.
- **Parallel input (Advanced):** This field specifies whether the reloadable input seed is shifted in one bit at a time or if it happens in parallel.

Other parameters used by this block are explained in the Common Parameters section at the beginning

of this chapter of the Reference Guide.

Logical

This block is listed in the following Xilinx Blockset libraries: *Basic Elements*, *Control Logic*, *Math*, and *Index*.



The Xilinx Logical block performs bitwise logical operations on 2, 3, or 4 fixed point numbers. Operands are zero padded and sign extended as necessary to make binary point positions coincide; then the logical operation is performed and the result is delivered at the output port.

In hardware this block is implemented as synthesizable VHDL. If you build a tree of logical gates, this synthesizable implementation is best as it facilitates logic collapsing in synthesis and mapping.

Block Parameters Dialog Box

The dialog box for the block can be invoked by double-clicking the icon in your Simulink model.

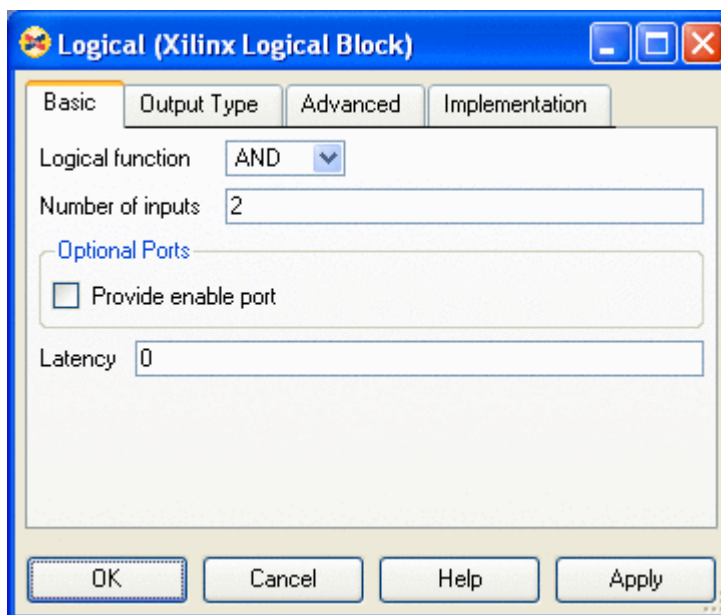


Figure 1: Logical block Basic tab

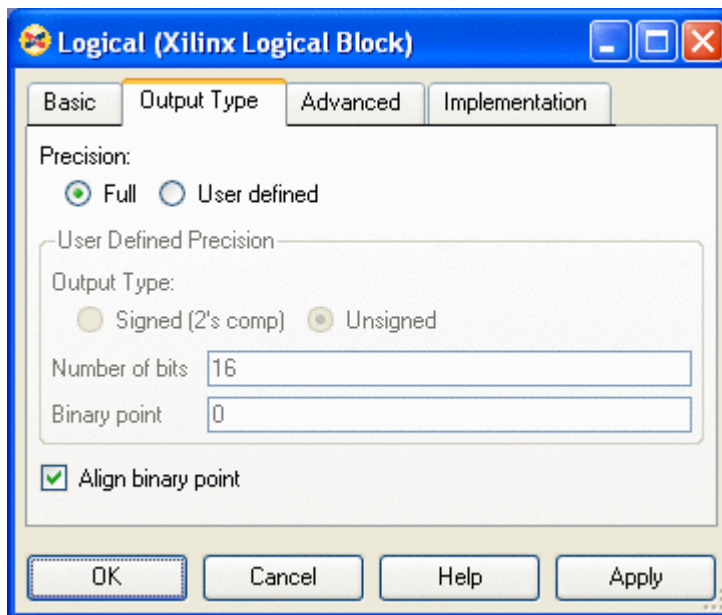


Figure 2: Logical block Output Type tab

Parameters specific to the block are:

- **Logical function:** specifies one of the following bitwise logical operators: AND, NAND, OR, NOR, XOR, XNOR.
- **Number of inputs:** specifies the number of inputs (1 - 1024).
- **Align binary point:** specifies that the block must align binary points automatically. If not selected, all inputs must have the same binary point position.

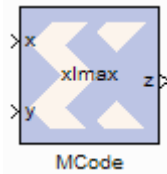
Other parameters used by this block are described in the Common Parameters section at the beginning of this chapter.

Xilinx LogiCORE

This block does not use a Xilinx LogiCORE.

MCode

This block is listed in the following Xilinx Blockset libraries: Control Logic, Math, and Index.



The Xilinx MCode block is a container for executing a user-supplied MATLAB function within Simulink. A parameter on the block specifies the m-code function name. The block executes the m-code to calculate block outputs during a Simulink simulation. The same code is translated in a straightforward way into equivalent behavioral VHDL/Verilog when hardware is generated. The block's Simulink interface is derived from the MATLAB function signature, and from block mask parameters. There is one input port for each parameter to the function, and one output port for each value the function returns. Port names and ordering correspond to the names and ordering of parameters and return values.

The MCode block supports a limited subset of the MATLAB language that is useful for implementing arithmetic functions, finite state machines and control logic. Users who wish to implement complete MATLAB algorithms on fixed-point FPGA hardware should consider using the Xilinx AccelDSP Algorithmic Synthesis tool. AccelDSP can be used to create custom IP blocks, from high-level, floating-point MATLAB, for use in combination with the Xilinx DSP blockset.

The MCode block has the following three primary coding guidelines that must be followed

- All block inputs and outputs must be of Xilinx fixed point type.
- The block must have at least one output port.
- The code for the block must exist on the MATLAB path or in the same directory as the model that uses the block.

The MCode block tutorial shows three examples of functions for the MCode block. The first example (also described below) consists of a function `xlmax` which returns the maximum of its inputs. The second illustrates how to do simple arithmetic. The third shows how to build finite state machines. The example models are linked from the Examples section of this guide.

Configuring an MCode block

The MATLAB Function parameter of an MCode block specifies the name of the block's .m code function. This function must exist in one of the three locations at the time this parameter is set. The three possible locations are:

- The directory where the model file is located.
- A subdirectory of the model directory named **private**.
- A directory in the MATLAB path.

The block icon displays the name of the m-code function. To illustrate these ideas, consider the file `xlmax.m` containing function `xlmax`:

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

An MCode block based on function `xlmax` will have input ports `x` and `y` and output port `z`.

The following figure shows how to set up an MCode block to use function `xlmax`.

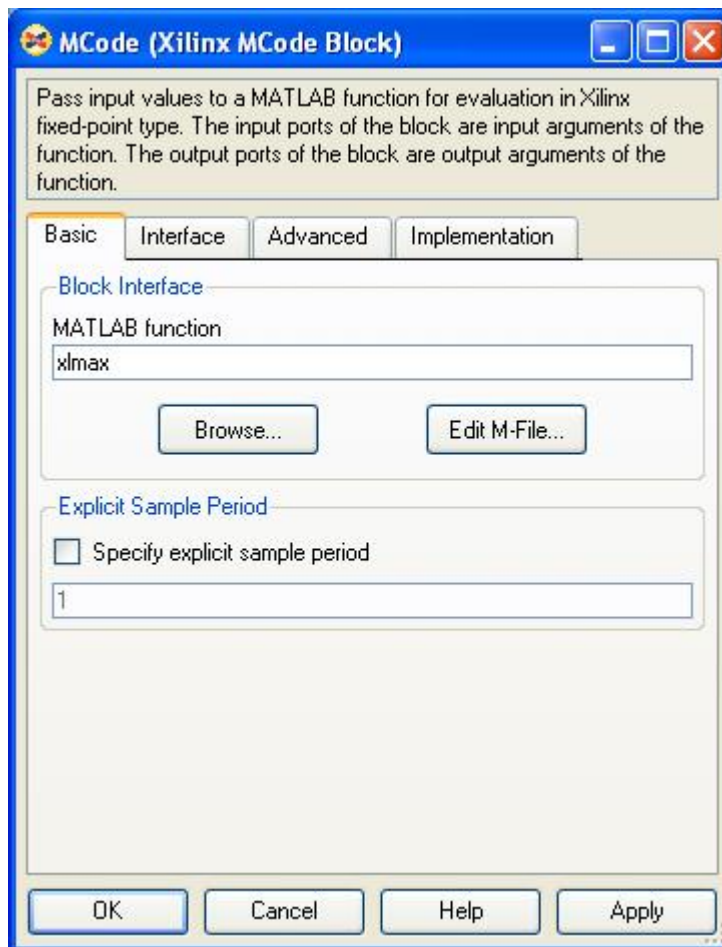


Figure 1: Setting up the MCode block parameters dialog to use xlmaz

Once the model is compiled, the xlmaz MCode block will appear like the block illustrated in Figure 2.

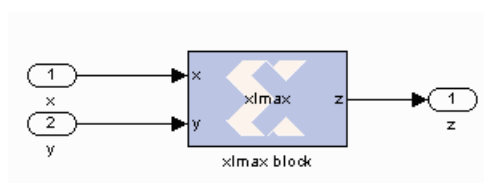


Figure 2: MCode block icon, after being compiled to use function xlmaz

MATLAB Language Support

The MCode block supports the following MATLAB language constructs:

- Assignment statements
- Simple and compound `if/else/elseif` end statements
- `switch` statements
- Arithmetic expressions involving only addition and subtraction
- Addition
- Subtraction
- Multiplication

- Division by a power of two
- Relational operators:
 - < Less than
 - <= Less than or equal to
 - > Greater than
 - >= Greater than or equal to
 - == Equal to
 - ~= Not equal to
- Logical operators:
 - & And
 - | Or
 - ~ Not

The MCode block supports the following MATLAB functions.

- Type conversion. The only supported data type is `xfix`, the Xilinx fixed-point type. The `xfix()` type conversion function is used to convert to this type. The conversion is done implicitly for integers but must be done explicitly for floating point constants. All values must be scalar - arrays are not supported.
- Functions that return `xfix` properties:
 - `xl_nbits()` Returns number of bits
 - `xl_binpt()` Returns binary point position
 - `xl_arith()` Returns arithmetic type
- Bit-wise logical functions:
 - `xl_and()` Bit-wise and
 - `xl_or()` Bit-wise or
 - `xl_xor()` Bit-wise xor
 - `xl_not()` Bit-wise not
- Shift functions: `xl_lsh()` and `xl_rsh()`
- Slice function: `xl_slice()`
- Concatenate function: `xl_concat()`
- Reinterpret function: `xl_force()`
- Internal state variables: `xl_state()`
- MATLAB Functions:
 - `disp()` Displays variable values
 - `error()` Displays message and abort function
 - `isnan()` Tests whether a number is NaN
 - `NaN` Returns Not-a-Number
 - `num2str()` Converts a number to string
 - `ones(1, N)` Returns 1-by-N vector of ones
 - `pi` Returns pi
 - `zeros(1, N)` Returns 1-by-N vector of zeros

Data Types

There are three kinds of `xfix` data types: unsigned fixed-point (`xlUnsigned`), signed fixed-point (`xlSigned`), and boolean (`xlBoolean`). Arithmetic operations on these data types produce signed and unsigned fixed-point values. Relational operators produce a boolean result. Relational operands can be any `xfix` type, provided the mixture of types makes sense. Boolean variables can be compared to boolean variables, but not to fixed-point numbers; boolean variables are incompatible

with arithmetic operators. Logical operators can only be applied to boolean variables. Every operation is performed in full precision, i.e., with the minimum precision needed to guarantee no information is lost.

Literal Constants

Integer, floating-point, and boolean literals are supported. Integer literals are automatically converted to `xfix` values of appropriate width having a binary point position at zero. Floating-point literals must be converted to the `xfix` type explicitly with the `xfix()` conversion function. The predefined MATLAB values `true` and `false` are automatically converted to boolean literals.

Assignment

The left-hand side of an assignment can only contain one variable. A variable can be assigned more than once.

Control Flow

The conditional expression of an `if` statement must evaluate to a boolean. Switch statements can contain cases and an otherwise clause. The types of a switch selector and its cases must be compatible; thus, the selector can be boolean provided its cases are. All cases in a switch must be constant; equivalently, no case can depend on an input value.

When the same variable is assigned in several branches of a control statement, the types being assigned must be compatible. For example,

```
if (u > v)
    x = a;
else
    x = b;
end
```

would be acceptable only if `a` and `b` were both boolean or both arithmetic.

Constant Expressions

An expression is constant provided its value does not depend on the value of any input argument. Thus, for example, the variable `c` defined by

```
a = 1;
b = a + 2;
c = xfix({xlSigned, 10, 2}, b + 3.345);
```

can be used in any context that demands a constant.

`xfix()` Conversion

The `xfix()` conversion function converts a `double` to an `xfix`, or changes one `xfix` into another having different characteristics. A call on the conversion function looks like the following

```
x = xfix(type_spec, value)
```

Here `x` is the variable that receives the `xfix`. `type_spec` is a cell array that specifies the type of `xfix` to create, and `value` is the value being operated on. Value can be floating point or `xfix` type. The `type_spec` cell array is defined using curly braces in the usual MATLAB method. For example,

```
xfix({xlSigned, 20, 16, xlRound, xlWrap}, 3.1415926)
```

returns an `xfix` approximation to `pi`. The approximation is signed, occupies 20 bits (16 fractional), quantizes by rounding, and wraps on overflow.

The `type_spec` consists of 1, 3, or 5 elements. Some elements can be omitted. When elements are

omitted, default element settings are used. The elements specify the following properties (in the order presented): **data type**, **width**, **binary point position**, **quantization mode**, and **overflow mode**. The **data type** can be `xlBoolean`, `xlUnsigned`, or `xlSigned`. When the type is `xlBoolean`, additional elements are not needed (and must not be supplied). For other types, width and binary point position must be supplied. The quantization and overflow modes are optional, but when one is specified, the other must be as well. Three values are possible for quantization: `xlTruncate`, `xlRound`, and `xlRoundBanker`. The default is `xlTruncate`. Similarly, three values are possible for overflow: `xlWrap`, `xlSaturate`, and `xlThrowOverflow`. For `xlThrowOverflow`, if an overflow occurs during simulation, an exception occurs.

All values in a *type_spec* must be known at compilation time; equivalently, no *type_spec* value can depend on an input to the function.

The following is a more elaborate example of an `xfix()` conversion:

```
width = 10, binpt = 4;
z = xfix({xlUnsigned, width, binpt}, x + y);
```

This assignment to `z` is the result of converting `x + y` to an unsigned fix-point number that is 10 bits wide with 4 fractional bits using `xlTruncate` for quantization and `xlWrap` for overflow.

If several `xfix()` calls need the same *type_spec* value, you can assign the *type_spec* to a variable, then use the variable for `xfix()` calls. For example, the following is allowed:

```
proto = {xlSigned, 10, 4};
x = xfix(proto, a);
y = xfix(proto, b);
```

xfix Properties: `xl_arith`, `xl_nbits`, and `xl_binpt`

Each `xfix` number has three properties: the arithmetic type, the bit width, and the binary point position. The MCode blocks provide three functions to get these properties of a fixed point number. The results of these functions are constants and will be evaluated when Simulink compiles the model.

Function `a = xl_arith(x)` returns the arithmetic type of the input number `x`. The return value is either 1, 2, or 3 for `xlUnsigned`, `xlSigned`, or `xlBoolean` respectively.

Function `n = xl_nbits(x)` returns the width of the input number `x`.

Function `b = xl_binpt(x)` returns the binary point position of the input number `x`.

Bit-wise Operators: `xl_or`, `xl_and`, `xl_xor`, and `xl_not`

The MCode block provides four built-in functions for bit-wise logical operations: `xl_or`, `xl_and`, `xl_xor`, and `xl_not`.

Function `xl_or`, `xl_and`, and `xl_xor` perform bit-wise logical or, and, and xor operations respectively. Each function is in the form of

```
x = xl_op(a, b, ...).
```

Each function takes at least two fixed point numbers and returns a fixed point number. All the input arguments are aligned at binary point position.

Function `xl_not` performs a bit-wise logical not operation. It is in the form of `x = xl_not(a)`. It only takes one `xfix` number as its input argument and returns a fixed point number.

The following are some examples of these function calls:

```

X = xl_and(a, b);
Y = xl_or(a, b, c);
Z = xl_xor(a, b, c, d);
N = xl_not(x);

```

Shift Operators: `xl_rsh`, and `xl_lsh`

Functions `xl_lsh` and `xl_rsh` allow you to shift a sequence of bits of a fixed-point number. The function is in the form:

`x = xl_lsh(a, n)` and `x = xl_rsh(a, n)` where *a* is a xfix value and *n* is the number of bits to shift.

Left or right shift the fixed-point number by *n* number of bits. The right shift (`xl_rsh`) moves the fixed-point number toward the least significant bit. The left shift (`xl_lsh`) function moves the fixed-point number toward the most significant bit. Both shift functions are a full precision shift. No bits are discarded and the precision of the output is adjusted as needed to accommodate the shifted position of the binary point.

Here are some examples:

```

% left shift a 5 bits
a = xfix({xlSigned, 20, 16, xlRound, xlWrap},
3.1415926)
b = xl_rsh(a, 5);

```

The output *b* is of type `xlSigned` with 21 bits and the binary point located at bit 21.

Slice Function: `xl_slice`

Function `xl_slice` allows you to access a sequence of bits of a fixed-point number. The function is in the form:

```

x = xl_slice(a, from_bit, to_bit).

```

Each bit of a fixed-point number is consecutively indexed from zero for the LSB up to the MSB. For example, given an 8-bit wide number with binary point position at zero, the LSB is indexed as 0 and the MSB is indexed as 7. The block will throw an error if the `from_bit` or `to_bit` arguments are out of the bit index range of the input number. The result of the function call is an unsigned fixed-point number with zero binary point position.

Here are some examples:

```

% slice 7 bits from bit 10 to bit 4
b = xl_slice(a, 10, 4);
% to get MSB
c = xl_slice(a, xl_nbits(a)-1, xl_nbits(a)-1);

```

Concatenate Function: `xl_concat`

Function `x = xl_concat(hi, mid, ..., low)` concatenates two or more fixed point numbers to form a single fixed point number. The first input argument occupies the most significant bits, and the last input argument occupies the least significant bits. The output is an unsigned fixed point number with binary point position at zero.

Reinterpret Function: `xl_force`

Function `x = xl_force(a, arith, binpt)` forces the output to a new type with `arith` as its new arithmetic type and `binpt` as its new binary point position. The `arith` argument can be one of `xlUnsigned`, `xlSigned`, or `xlBoolean`. The `binpt` argument must be from 0 to the bit width inclusively. Otherwise, the block will throw an error.

State Variables: `xl_state`

A MCode block can have internal state variables that hold their values from one simulation step to the next. A state variable is declared with the MATLAB keyword `persistent` and must be initially assigned with an `xl_state` function call.

The following code models a 4-bit accumulator:

```
function q = accum(din, rst)
    init = 0;
    persistent s, s = xl_state(init, {xlSigned, 4, 0});
    q = s;
    if rst
        s = init;
    else
        s = s + din;
    end
```

The state variable `s` is declared as `persistent`, and the first assignment to `s` is the result of the `xl_state` invocation. The `xl_state` function takes two arguments. The first is the initial value and must be a constant. The second is the precision of the state variable. It can be a type cell array as described in the `xfix` function call. It can also be an `xfix` number. In the above code, if `s = xl_state(init, din)`, then state variable `s` will use `din` as the precision. The `xl_state` function must be assigned to a `persistent` variable.

The `xl_state` function behaves in the following way:

1. In the first cycle of simulation, the `xl_state` function initializes the state variable with the specified precision.
2. In the following cycles of simulation, the `xl_state` function retrieves the state value left from last cycle and assigns the value to the corresponding variable with the specified precision.

`v = xl_state(init, precision)` returns the value of for a state variable. Input argument `init` is the initial value, `precision` is the precision for this state variable. `precision` can be a cell array in form of `{type, nbits, binpt}` or `{type, nbits, binpt, q, ov}`. `precision` can also be an `xfix` number.

`v = xl_state(init, precision, maxlen)` returns a vector object. The vector will be initialized with `init` and will have `maxlen` for maximum length it can be. The vector will be initialized with `init`. For example, `v = xl_state(zeros(1, 8), prec, 8)` creates a vector of 8 zeros, `v = xl_state([], prec, 8)` creates an empty vector with 8 as maximum length, `v = xl_state(0, prec, 8)` creates a vector of one zero as content and with 8 as the maximum length.

Conceptually, a vector state variable is a double ended queue. It has two ends, the front which is the element at address 0 and the back which is the element at `length - 1`.

Methods available for vector are:

<code>val = v(idx);</code>	Returns the value of element at address idx.
<code>v(idx) = val;</code>	Assigns the element at address idx with val.
<code>f = v.front;</code>	Returns the value of the front end. An error will be thrown if the vector is empty.
<code>v.push_front(val);</code>	Pushes val to the front and the increases the vector length by 1. An error will be thrown if the vector is full.
<code>v.pop_front;</code>	Pops one element from front and decreases the vector length by 1. An error will be thrown if the vector is empty.
<code>b = v.back;</code>	Returns the value of the back end. An error will be thrown if the vector is empty.
<code>v.push_back(val);</code>	Pushes val to the back and the increases the vector length by 1. An error will be thrown if the vector is full.
<code>v.pop_back;</code>	Pops one element from back and decreases the vector length by 1. An error will be thrown if the vector is empty.
<code>v.push_front_pop_back(val);</code>	Pushes val to the front and pops one element out from back. It's the shift operation. The length of the vector is unchanged. The vector cannot be empty to perform this operation.
<code>full = v.full;</code>	Returns true if the vector is full, otherwise, false.
<code>empty = v.empty;</code>	Returns true if the vector is empty, otherwise, false.
<code>len = v.length;</code>	Returns the number of elements in the vector.

A method of a vector that queries a state variable is called a *query method*. It has a return value. The following methods are query method: `v(idx)`, `v.front`, `v.back`, `v.full`, `v.empty`, `v.length`, `v.maxlen`. A method of a vector that changes a state variable is called an *update method*. An update method does not return any value. The following methods are update methods: `v(idx) = val`, `v.push_front(val)`, `v.pop_front`, `v.push_back(val)`, `v.pop_back`, and `v.push_front_pop_back(val)`. All query methods of a vector must be invoked before any update method invocation during any simulation cycle. An error will be thrown during model compilation if such rule is broken.

The MCode block may map a vector state variable into a vector of registers, a delay line, an addressable shift register, a single port ROM, or a single port RAM based on the usage of the state variable. The `xl_state` function can also be used to convert a MATLAB 1-D array into a zero indexed constant array. If the MCode block cannot map a vector state variables into FPGA components, an error will be thrown during model netlist time.

Here are some examples of using vector state variables.

Delay Line

The state variable in the following function will be mapped into a delay line.

```
function q = delay(d, lat)
    Persistent r, r = xl_state(zeros(1, lat), d, lat);
    q = r.back;
    r.push_front_pop_back(d);
```

Line of Registers

The state variable in the following function will be mapped into a line of registers.

```
function s = sum4(d)
    persistent r, r = xl_state(zeros(1, 4), d);
    s = r(0) + r(1) + r(2) + r(3);
    r.push_front_pop_back(d);
```


Vector of Constants

The state variable in the following function will be mapped into a vector of constants.

```
function s = myadd(a, b, c, d, nbits, binpt)
    p = {x1Signed, nbits, binpt, x1Round, x1Saturate};
    persistent coef, coef = xl_state([3, 7, 3.5, 6.7], p);
    s = a*coef(0) + b*coef(1) + c*coef(2) + c*coef(3);
```

Addressable Shift Register

The state variable in the following function will be mapped into an addressable shift register.

```
function q = addrsr(d, addr, en, depth)
    persistent r, r = xl_state(zeros(1, depth), d);
    q = r(addr);
    if en
        r.push_front_pop_back(d);
    end
```

Single Port ROM

The state variable in the following function will be mapped into a single port ROM.

```
function q = addrsr(contents, addr, arith, nbits, binpt)
    proto = {arith, nbits, binpt};
    persistent mem, mem = xl_state(contents, proto);
    q = mem(addr);
```

Single Port RAM

The state variable in the following function will be mapped into a single port RAM.

```
function dout = ram(addr, we, din, depth, nbits, binpt)
    proto = {x1Signed, nbits, binpt};
    persistent mem, mem = xl_state(zeros(1, depth), proto);
    dout = mem(addr);
    if we
        mem(addr) = din;
    end
```

MATLAB Functions

disp()

Displays expression value. In order to see the printing on the MATLAB console, the Enable printing with disp option must be checked. It only takes one input argument. The argument can be a string, an xfix number, or an MCode state variable. If it's an xfix number, it will print the type, binary value, and double precision value. For example, if variable x is assigned with xfix({x1Signed, 10, 7}, 2.75), disp(x) will print the following line:

```
type: Fix_10_7, binary: 010.11000000, double: 2.75
```

If the argument is a vector state variable, disp will print out the type, maximum length, current length, and the binary and double values of all the elements. For each simulation step, when the Enable printing with disp is on and when a disp function is invoked, a title line will be printed for the

corresponding block. The title line includes the block name, Simulink simulation time, and FPGA clock number.

The following MCode function shows several examples of using the disp function.

```
function x = testdisp(a, b)
    persistent dly, dly = xl_state(zeros(1, 8), a);
    persistent rom, rom = xl_state([3, 2, 1, 0], a);

    disp('Hello World!');

    disp(['num2str(dly) is ', num2str(dly)]);
    disp('disp(dly) is ');
    disp(dly);
    disp('disp(rom) is ');
    disp(rom);
    a2 = dly.back;
    dly.push_front_pop_back(a);

    x = a + b;

    disp(['a = ', num2str(a), ', ', ' ', ...
        'b = ', num2str(b), ', ', ' ', ...
        'x = ', num2str(x)]);

    disp(num2str(true));

    disp('disp(10) is');
    disp(10);
    disp('disp(-10) is');
    disp(-10);

    disp('disp(a) is ');
    disp(a);

    disp('disp(a == b)');
    disp(a==b);
```

The following lines are the result for the first simulation step.

```
xlmcode_testdisp/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 0000.00000000, double 0.000000,
    1: binary 0000.00000000, double 0.000000,
    2: binary 0000.00000000, double 0.000000,
    3: binary 0000.00000000, double 0.000000,
    4: binary 0000.00000000, double 0.000000,
    5: binary 0000.00000000, double 0.000000,
    6: binary 0000.00000000, double 0.000000,
    7: binary 0000.00000000, double 0.000000,

disp(rom) is
    type: Fix_11_7,
    maxlen: 4,
    length: 4,
    0: binary 0011.00000000, double 3.0,
```

```

1: binary 0010.00000000, double 2.0,
2: binary 0001.00000000, double 1.0,
3: binary 0000.00000000, 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: 0000.00000000, double: 0.000000

disp(a == b)
    type: Bool, binary: 1, double: 1

```

You can find the above example in the MCode block tutorial.

error()

Displays message and abort function. See Matlab help on this function for more detailed information. Message formatting is not supported by the MCode block. For example:

```

if latency <=0
    error('latency must be a positive');
end

```

isnan()

Returns true for Not-a-Number. `isnan(X)` returns true when X is Not-a-Number. X must be a scalar value of double or Xilinx fixed point number. This function is not supported for vectors or matrices. For example:

```

if isnan(incr) & incr == 1
    cnt = cnt + 1;
end

```

NaN()

The `NaN()` function generates an IEEE arithmetic representation for Not-a-Number. A NaN is obtained as a result of mathematically undefined operations like 0.0/0.0 and inf-inf. `NaN(1,N)` generates a 1-by-N vector of NaN values. Here are some examples of using NaN.

```

if x < 0
    z = NaN;
else
    z = x + y;
end

```

num2Str()

Converts a number to a string. `Num2str(X)` converts the X into a string. X can be a scalar value of double, a Xilinx fixed point number, or a vector state variable. The default number of digits is based on the magnitude of the elements of X. Here's an example of `num2str`:

```

if opcode <=0 | opcode >= 10

```

```
error(['opcode is out of range: ', num2str(opcode)]);
end
```

ones()

The ones() function generates a specified number of one values. Ones(1,N) generates a 1-by-N vector of ones. Ones(M,N) where M must be 1. It's usually used with xl_state() function call. For example, the following line creates a 1-by-4 vector state variable initialized to [1, 1, 1, 1].

```
persistant m, m = xl_state(ones(1, 4), proto)
```

zeros()

The zeros() function generates a specified number of zero values. Zeros(1,N) generates a 1-by-N vector of zeros. zeros(M,N) where M must be 1. It's usually used with xl_state() function call. For example, the following line creates a 1-by-4 vector state variable initialized to [0, 0, 0, 0].

```
persistant m, m = xl_state(zeros(1, 4), proto)
```

FOR Loop

FOR statement is fully unrolled. The following function sums n samples.

```
function q = sum(din, n)
    persistent regs, regs = xl_state(zeros(1, 4), din);
    q = reg(0);
    for i = 1:n-1
        q = q + reg(i);
    end
    regs.push_front_pop_back(din);
```

The following function does a bit reverse.

```
function q = bitreverse(d)
    q = xl_slice(d, 0, 0);
    for i = 1:xl_nbits(d)-1
        q = xl_concat(q, xl_slice(d, i, i));
    end
```

Variable Availability

MATLAB code is sequential (i.e., statements are executed in order). The MCode block requires that every possible execution path assigns a value to a variable before it is used (except as a left-hand side of an assignment). When this is the case, we say the variable is *available* for use. The MCode block will throw an error if its m-code function accesses unavailable variables.

Consider the following m-code:

```
function [x, y, z] = test1(a, b)
    x = a;
    if a>b
        x = a + b; y = a;
    end
    switch a
        case 0
```

```
z = a + b;  
case 1  
z = a - b;  
end
```

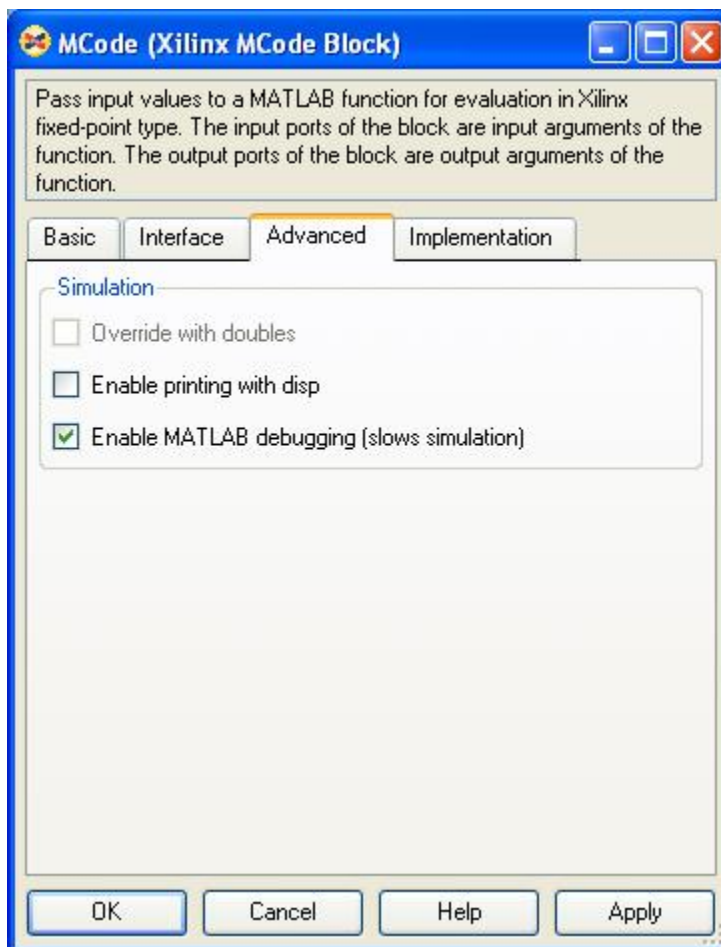
Here **a**, **b**, and **x** are available, but **y** and **z** are not. Variable **y** is not available because the **if** statement has no **else**, and variable **z** is not available because the **switch** statement has no **otherwise** part.

DEBUG MCode

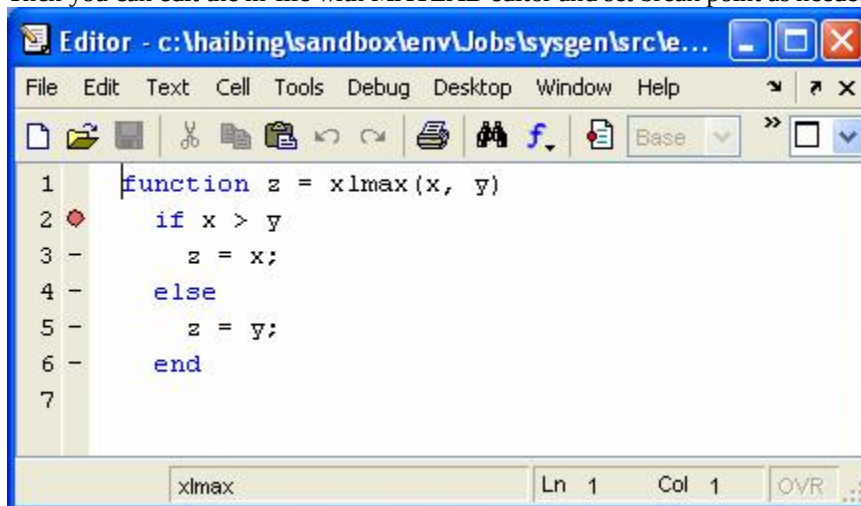
There are two ways to debug your MCode. One is to insert `disp()` function in your code and enable printing, the other is to use the MATLAB debugger. For usage of the `disp` function, please reference the section for the `disp` function.

If you want to use the MATLAB debugger, you need to check the Enable MATLAB debugging option. Then you can open your MATLAB function with MATLAB editor, set break points, and debug your M- function. Just be aware, every time you modify your script, you need to execute `clear functions` in MATLAB console.

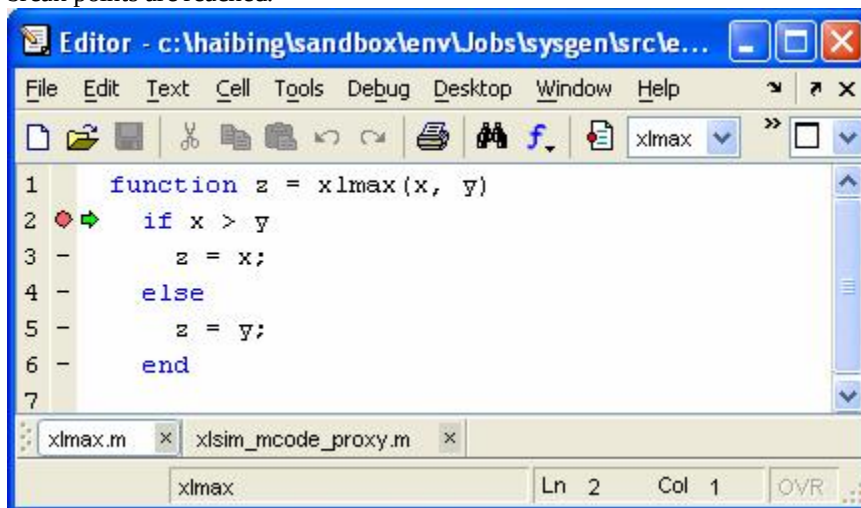
To start debugging your m-function, you need to first check the Enable MATLAB debugging checkbox in the Advanced tab of the MCode block parameters dialog and click the OK or Apply button.



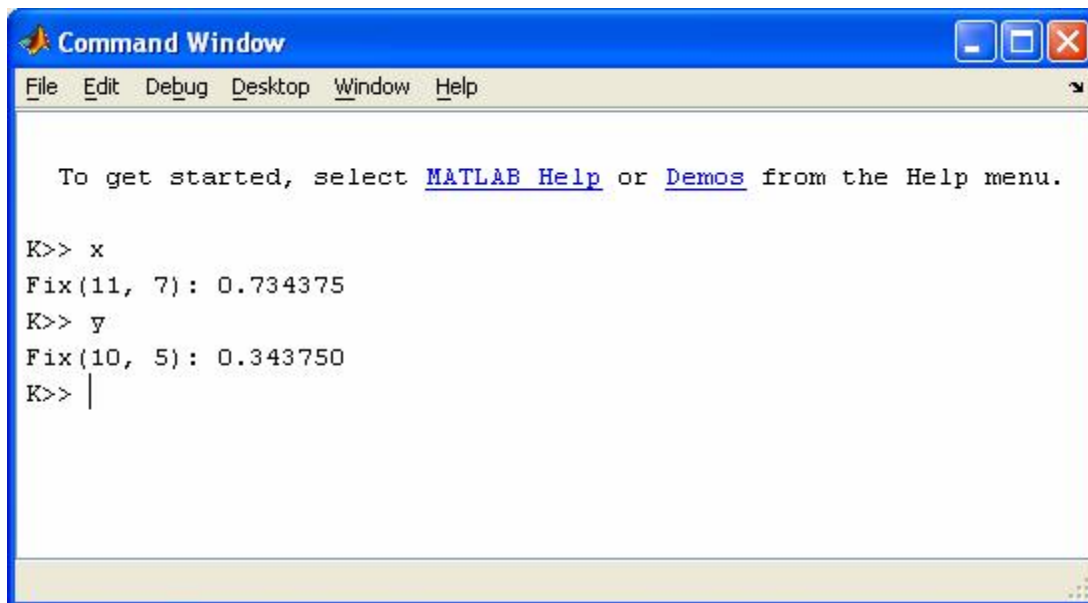
Then you can edit the m-file with MATLAB editor and set break point as needed.



During the Simulink simulation, the MATLAB debugger will stop at the break points you set when the break points are reached.



When debugging, you can also examine the values of the variables by typing the variable names in the MATLAB console.



There is one special case to consider when the function for an MCode block is executed from the MATLAB debugger. A `switch/case` expression inside an MCode block must be an `xfix`, however, executing a `switch/case` expression from the MATLAB console requires that the expression must be a `double` or `char`. To facilitate execution in the MATLAB console, a call to `double()` must be added. For example, if the function contains

```
switch i
    case 0
        x = 1
    case 1
        x = 2
end
```

where `i` is an `xfix`, then to run from the console this must be changed to

```
switch double(i)
    case 0
        x = 1
    case 1
        x = 2
end
```

The `double()` function call only has an effect when the `.m` code is run from the console. The MCode block ignores the `double()` call.

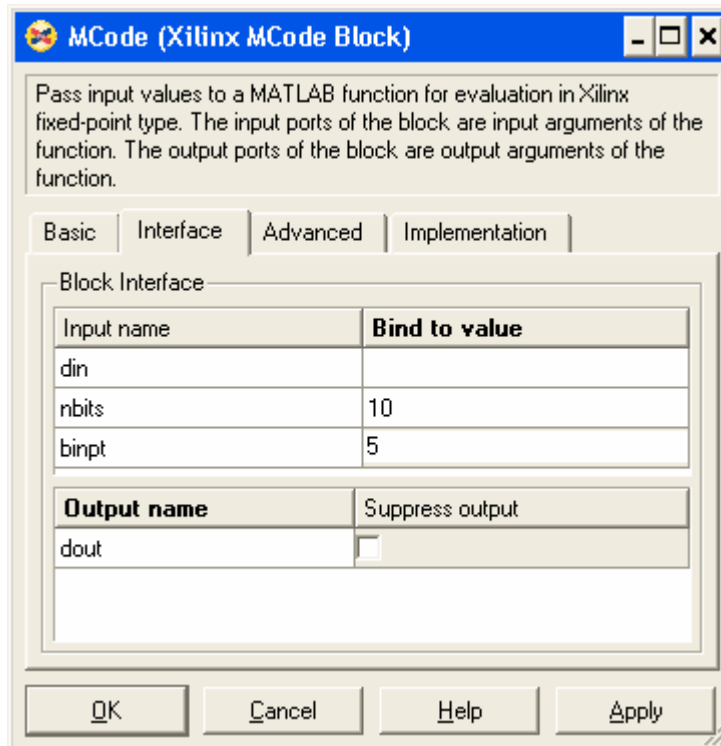
Passing Parameters

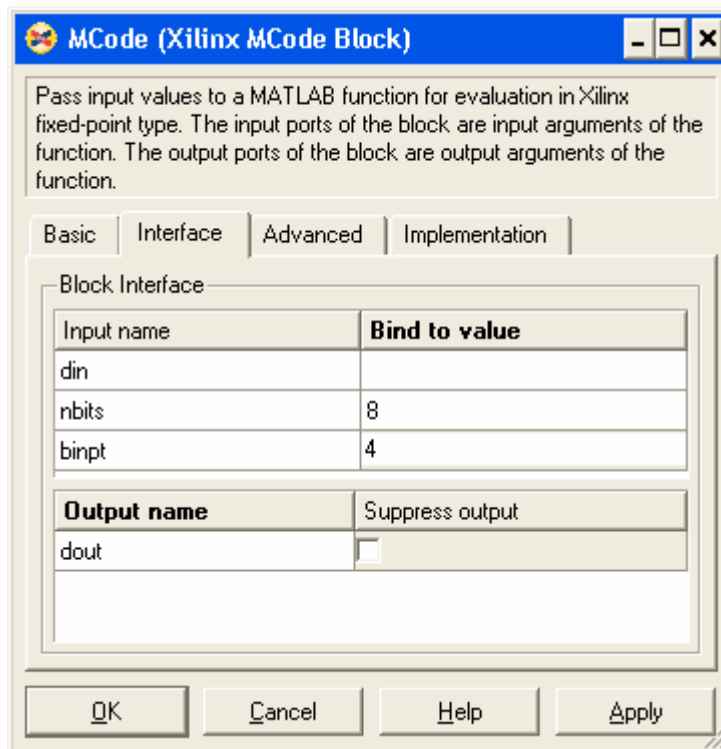
It is possible to use the same m-function in different MCode blocks, passing different parameters to the m-function so that each block may behave differently. This is achieved by binding input arguments to some values. To bind the input arguments, select the interface tab on the block GUI. After you bind those arguments to some values, these m-function arguments will not be shown as input ports of the MCode block.

Consider for example, the following m-function:

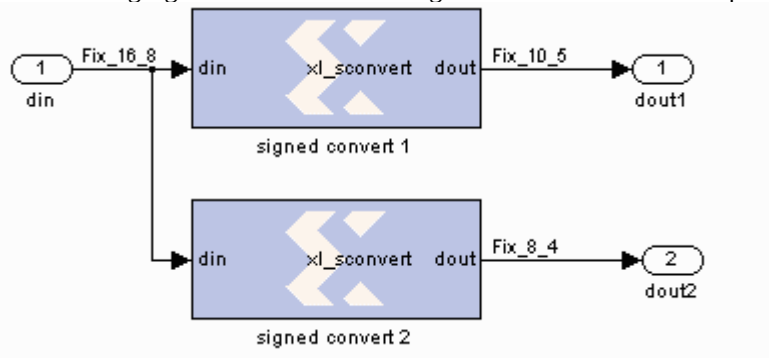
```
function dout = xl_sconvert(din, nbits, binpt)
    proto = {xlSigned, nbits, binpt};
    dout = xfix(proto, din);
```

The following figure shows how the Function Parameter Bindings parameter is set for blocks using `xl_sconvert`.





The following figure shows the block diagram after the model is compiled.



Parameters can only be double or logical numbers.

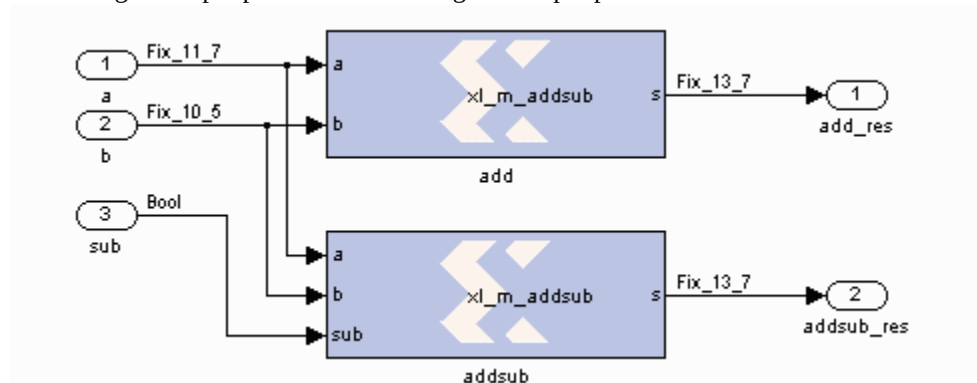
Optional Input Ports

The parameter passing mechanism allows the MCode block to have optional input ports. Consider for example, the following m-function:

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

If `sub` is set to be `false`, the MCode block that uses this m-function will have two input ports `a` and `b` and will perform full precision addition. If it is set to an empty cell array `{}`, the block will have three input ports `a`, `b`, and `sub` and will perform full precision addition or subtraction based on the value of input port `sub`.

The following figure shows the block diagram of two blocks using the same xl_m_addsub function, one having two input ports and one having three input ports.



Constructing a State Machine

There are two ways to build a state machine using an MCode block. One way is to specify a stateless transition function using a MATLAB function and pair an MCode block with one or more state register blocks. Usually the MCode block drives a register with the value representing the next state, and the register feeds back the current state into the MCode block. For this to work, the precision of the state output from the MCode block must be static, i.e., independent of any inputs to the block. Occasionally you may find you need to use `xfix()` conversions to force static precision. The following code illustrates this:

```
function nextstate = fsm1(currentstate, din)
% some other code
nextstate = currentstate;
switch currentstate
    case 0, if din==1, nextstate = 1; end
end
% a xfix call should be used at the end
nextstate = xfix({xlUnsigned, 2, 0}, nextstate);
```

Another way is to use state variables. The above function can be re-written as the following:

```
function currentstate = fsm1(din)
persistent state,
state=xl_state(0,{xlUnsigned,2,0});
currentstate = state;
switch double(state)
    Case 0, if din==1; state = 1; end
end
```

Reset and Enable Signals for State Variables

The MCode block can automatically infer register reset and enable signals for state variables when conditional assignments to the variables contain two or fewer branches.

For example, the following m-code infers an enable signal for conditional assignment of persistent state variable `r1`:

```
function myFn = aFn(en, a)
    persistent r1, r1 = xl_state(0, {xlUnsigned, 2, 0});
    myFn = r1;
    if en
        r1 = r1 + a
    else
        r1 = r1
    end
end
```

There are two branches in the conditional assignment to persistent state variable `r1`. A register is used to perform the conditional assignment. The input of the register is connected to `r1 + a`, the output of the register is `r1`. The register's enable signal is inferred; the enable signal is connected to `en`, when `en` is asserted persistent state variable `r1` is assigned to `r1 + a`. When `en` evaluates to false, the enable signal on the register is deasserted resulting in the assignment of `r1` to `r1`. The following m-code will also infer an enable signal on the register used to perform the conditional assignment:

```
function myFn = aFn(en, a)
    persistent r1, r1 = xl_state(0, {xlUnsigned, 2, 0});
    myFn = r1;
    if en
        r1 = r1 + a
    end
end
```

An enable is inferred instead of a reset because the conditional assignment of persistent state variable `r1` is to a non-constant value, `r1 + a`.

If there were three branches in the conditional assignment of persistent state variable `r1`, the enable signal would not be inferred. The following m-code illustrates the case where there are three branches in the conditional assignment of persistent state variable `r1` and the enable signal is not inferred:

```
function myFn = aFn(en, en2, a, b)
    persistent r1, r1 = xl_state(0, {xlUnsigned, 2, 0});
    if en
        r1 = r1 + a
    elseif en2
        r1 = r1 + b
    else
        r1 = r1
    end
end
```

The reset signal can be inferred if a persistent state variable is conditionally assigned to a constant; the reset is synchronous. Consider the following m-code example which infers a reset signal for the assignment of persistent state variable `r1` to `init`, a constant, when `rst` evaluates to true and `r1 + 1` otherwise:

```
function myFn = aFn(rst)
    persistent r1, r1 = xl_state(0, {xlUnsigned, 4, 0});
    myFn = r1;
    init = 7;
    if (rst)
```

```

    r1 = init
else
    r1 = r1 + 1
end

```

The m-code example above which infers reset can also be written as:

```

function myFn = aFn(rst)
    persistent r1, r1 = xl_state(0, {xlUnsigned,4,0});
    init = 1;
    myFn = r1;
    r1 = r1 +1
    if (rst)
        r1 = init
    end
end

```

In both code examples above, the reset signal of the register containing persistent state variable `r1` is assigned to `rst`. When `rst` evaluates to true, the register's reset input is asserted and the persistent state variable is assigned to constant `init`. When `rst` evaluates to false, the register's reset input is deasserted and persistent state variable `r1` is assigned to `r1 + 1`. Again, if the conditional assignment of a persistent state variable contains three or more branches, a reset signal is not inferred on the persistent state variable's register.

It is possible to infer reset and enable signals on the register of a single persistent state variable. The following m-code example illustrates simultaneous inference of reset and enable signals for the persistent state variable `r1`:

```

function myFn = aFn(rst,en)
    persistent r1, r1 = xl_state(0, {xlUnsigned, 4, 0});
    myFn = r1;
    init = 0;
    if rst
        r1 = init
    else
        if en
            r1 = r1 + 1
        end
    end
end

```

The reset input for the register of persistent state variable `r1` is connected to `rst`; when `rst` evaluates to true, the register's reset input is asserted and `r1` is assigned to `init`. The enable input of the register is connected to `en`; when `en` evaluates to true, the register's enable input is asserted and `r1` is assigned to `r1 + 1`. It is important to note that an inferred reset signal takes precedence over an inferred enable signal regardless of the order of the conditional assignment statements. Consider the second code example above; if both `rst` and `en` evaluate to true, persistent state variable `r1` would be assigned to `init`.

Inference of reset and enable signals also works for conditional assignment of persistent state variables using switch statements, provided the switch statements contain two or less branches.

The MCode block performs dead code elimination and constant propagation compiler optimizations when generating code for the FPGA. This can result in the inference of reset and/or enable signals in conditional assignment of persistent state variables, when one of the branches is never executed. For this to occur, the conditional must contain two branches that are executed after dead code is eliminated

and constant propagation is performed.

Pipelining Combinational Logic via Retiming

The generated FPGA code for a MCode block may contain many levels of combinational logic and hence a large critical path delay. To allow a downstream logic synthesis tool to automatically pipeline the combinational logic, you can add delay blocks before the MCode block inputs or after the MCode block outputs. These delay blocks should have the parameter 'Enable Register Retiming' set, which instructs the code generator to implement delay with registers. Or, you can use the vector state variables to model delays.

Shift Operations with Multiplication and Division

The MCode block can detect when a number is multiplied or divided by constants that are powers of two. If detected, the MCode block will perform a shift operation. For example, multiplying by 4 is equivalent to left shifting 2 bits and dividing by 8 is equivalent to right shifting 3 bits. A shift is implemented by adjusting the binary point, expanding the `xfix` container as needed. For example, a `Fix_8_4` number multiplied by 4 will result in a `Fix_8_2` number, and a `Fix_8_4` number multiplied by 64 will result in a `Fix_10_0` number.

Using the `xl_state` Function with Rounding Mode

The `xl_state` function call creates an `xfix` container for the state variable. The container's precision is specified by the second argument passed to the `xl_state` function call. If precision uses `xlRound` for its rounding mode, hardware resources will be added to accomplish the rounding. If rounding the initial value is all that is required, an `xfix` call to round a constant does not require additional hardware resources. The rounded value can then be passed to the `xl_state` function. For example:

```
init = xfix({xlSigned,8,5,xlRound,xlWrap}, 3.14159);  
persistent s, s = xl_state(init, {xlSigned, 8, 5});
```

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the block icon in a Simulink model.

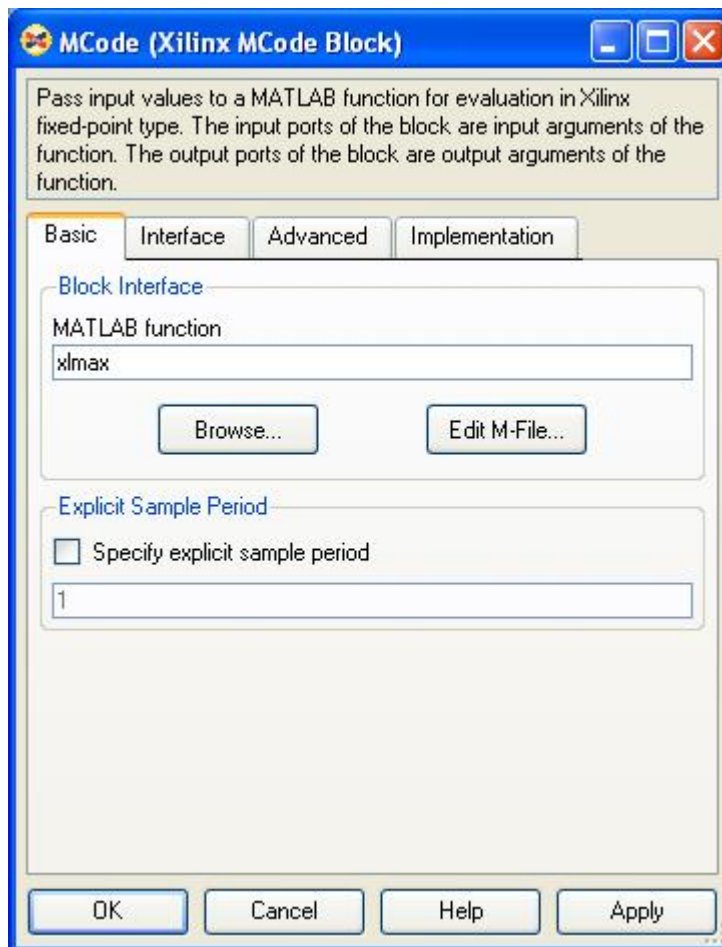


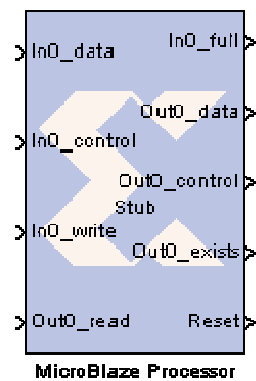
Figure 3: MCode block parameters dialog box basic tab

As described earlier in this chapter, the MATLAB Function parameter on an MCode block tells the name of the block's function, and the Function Parameter Bindings parameter specifies a list of constant inputs and their values. Other parameters used by this block are explained in the Common Parameters section at the beginning of this chapter of the Reference Guide.

MicroBlaze Processor

This block is listed in the following Xilinx Blockset libraries: Index, Control.

This block is now obsolete. Please use the EDK Processor block instead.



The Xilinx MicroBlaze Processor block provides a way to design and simulate peripherals created to target the EDK MicroBlaze Processor. Users can connect customized IP to the MicroBlaze via Fast Simplex Links (FSLs) that are available on the processor (Figure 1). FSLs can be thought of as unidirectional FIFOs. The MicroBlaze can include a maximum of eight input and eight output FSLs (a total of 16). Both synchronous and asynchronous FIFOs can be used; users can create System Generator peripherals that run synchronously or asynchronously to the MicroBlaze processor. As depicted in the Figure below, instancing of the FSL FIFOs is left to the EDK tool (Embedded Development Kit). Simulation of such systems may be done via hardware co-simulation. Creation, management and configuration of the simulation model is accessed via the block parameter mask and will be explained in further detail in the following text.

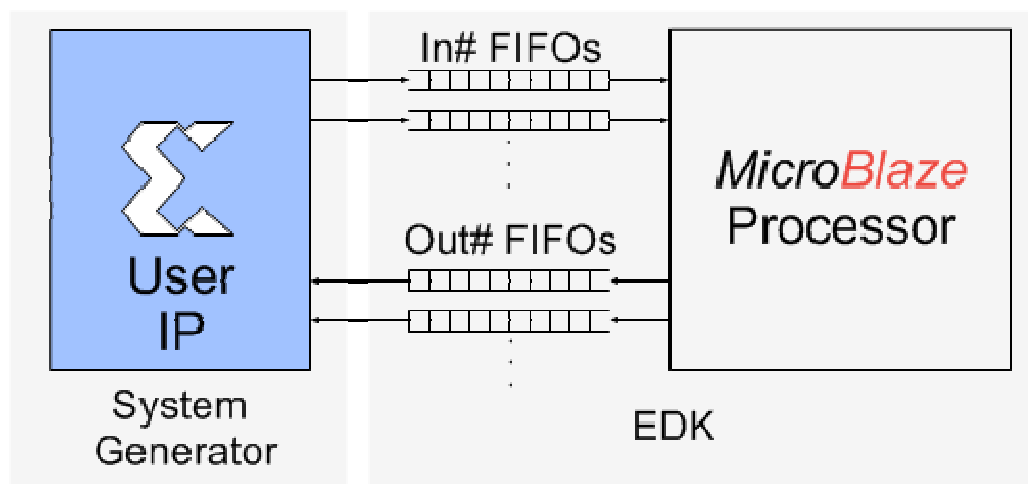


Figure 1: Connecting User IP to the MicroBlaze

Block Interface

In the following discussion, the symbol # represents a number from 0 to 7. The block has a user-configurable number of input and output interfaces (Figure 4). For each input interface, 4 ports are created: 3 input ports (In#_data, In#_control and In#_write) and 1 output port (In#_full). Similarly, 4 ports are also created for each output interface: 1 input port (Out#_read) and 3 output ports (Out#_data, Out#_control and Out#_exists). A maximum of 8 input and 8 output interfaces are supported. An optional Rst port is made available on selecting the 'Provide Reset Port' option (Figure 4). Note that the Rst port will appear as an output port on the MicroBlaze Processor block. This port can be connected to any of the asynchronous reset

ports/pins on the MicroBlaze (from the EDK environment), and provides a way for the MicroBlaze to reset any connected System Generator designs.

Descriptions of the ports are available in Table 1.

Port Name	Port Type	Port Width (Bits)	Port Description	FSL Connection
In#_data	Input	32	Data to be written to the FSL FIFO	Masters the FSL connection, i.e. writing to the FSL FIFO.
In#_control	Input	1	Flag bit. High indicates that data written onto the FIFO is a control word	
In#_write	Input	1	High enables writing to an FSL FIFO on which the EDK MicroBlaze is connected as a Slave Peripheral	
In#_full	Output	1	High indicates that the FSL FIFO is full	
Out#_data	Output	32	Data read from the FSL FIFO	Slave to the FSL connection, i.e. reading from the FSL FIFO.
Out#_control	Output	1	Flag bit. High indicates that data read from the FIFO is a control word	
Out#_read	Input	1	High enables reading from an FSL FIFO on which the EDK MicroBlaze is connected as a Master Peripheral	
Out#_exists	Output	1	High indicates that the FIFO is non-empty.	
Rst	Output	1	Indicates the state of an asynchronous Reset port/pin	N/A

Table 1: Ports on the MicroBlaze Processor Block

The timing relationship between the various signals on the ports for successful read and write operation are depicted in Figure 2 and Figure 3 respectively.

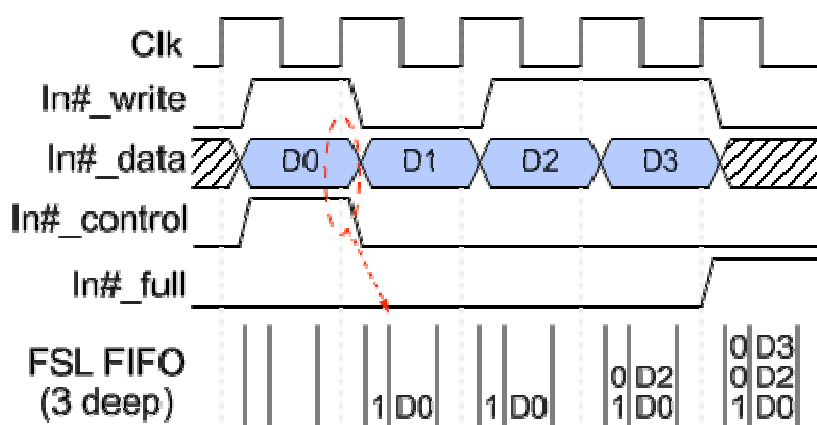


Figure 2: Timing Diagram for Write Operation

As shown in Figure 2, if In#_full is low and In#_write is driven high, the value of In#_data is written into the FIFO. The In#_full signal is high when the FIFO is full, writes to the FIFO when In#_full is high will be ignored. In# ports Masters the FSL that it connects to, and must be configured as a Master

peripheral in the EDK project.

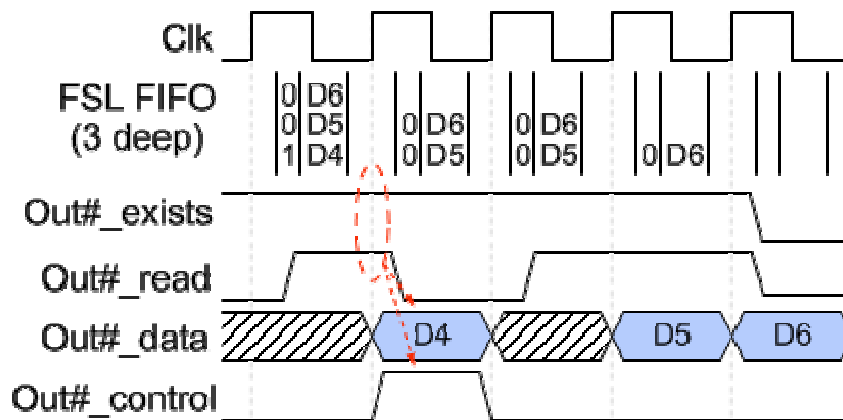


Figure 3: Timing Diagram for Read Operation

As shown in Figure 3, setting **Out#_read** to high when **Out#_exists** is also high, causes the first data item in the FIFO to be read. The data read is available on the **Out#_data** port. The **Out#_exists** signal goes low when the FIFO is empty. Reading from an empty FIFO returns an undefined value. **Out#** ports are Slaves to the FSL that it connects to, and must be configured as a Slave peripheral in the EDK project.

After designing the FSL peripherals in System Generator, the model must be exported to the EDK environment using the EDK Export Tool. System Generator, in addition to VHDL source code, generates a Microprocessor Peripheral Definition (MPD) file, a Peripheral Analyze Order (PAO) file and a Black Box Definition (BBD) file for this block. Refer to the EDK Export Tool for more information.

Note: The mapping of **In#** and **Out#** to FSLs is completely controlled by the user from within the EDK environment. That is, **In0** will not necessarily be connected to **SFSL0**, **Out0** will not necessarily be connected to **MSFL0** and so on.

Block Parameters Dialog Box

The Block Parameters Dialog Box can be opened by double clicking the block in the Simulink model. The block parameters dialog box contains three tabbed sections: General, Hardware and Software.

General Tab

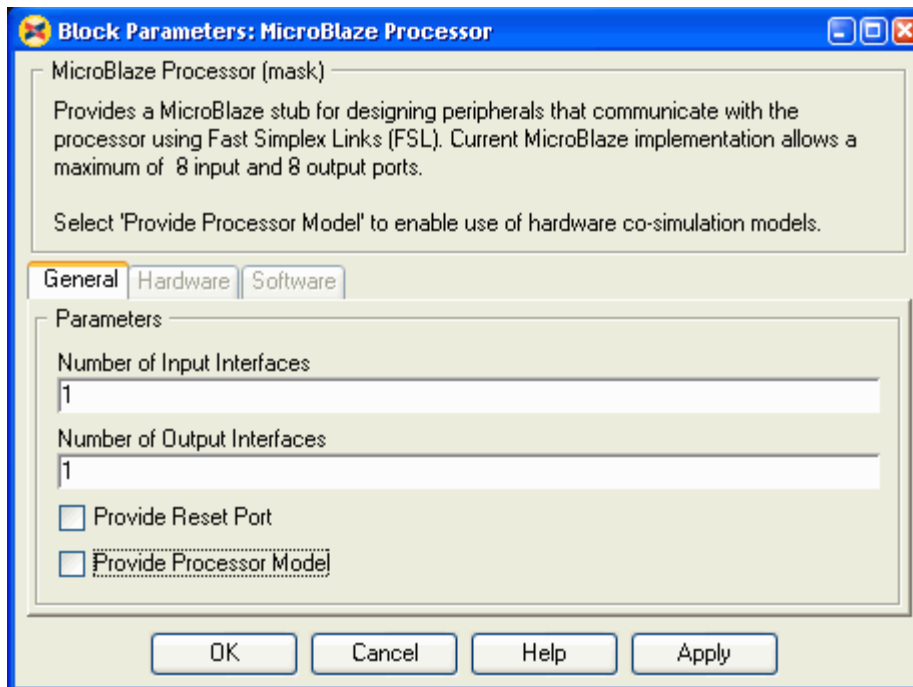


Figure 4: Block Parameters Dialog Box: General Tab

The General tab contains parameters that customize the MicroBlaze Processor, and enables the hardware co-simulation feature of the block.

The Parameters specific to the block are

- **Number of Input Interfaces:** Determines the number of System Generator to MicroBlaze FSL interfaces. The block interface is appropriately configured with a number of input and output ports. The number of input interfaces must be less than or equal to 8.
- **Number of Output Interfaces:** Determines the number of MicroBlaze to System Generator FSL interfaces. The block interface is appropriately configured with a number of input and output ports. The number of output interfaces must be less than or equal to 8.
- **Provide Reset Port:** Adds an output port “Rst” on the block interface. This provides a way for the user to allow resetting of the System Generator design by the MicroBlaze.
- **Provide Processor Model:** Enables the hardware co-simulation feature of the MicroBlaze. Selecting the Provide Processor Model checkbox will enable the Hardware and Software tabs.

Hardware Tab

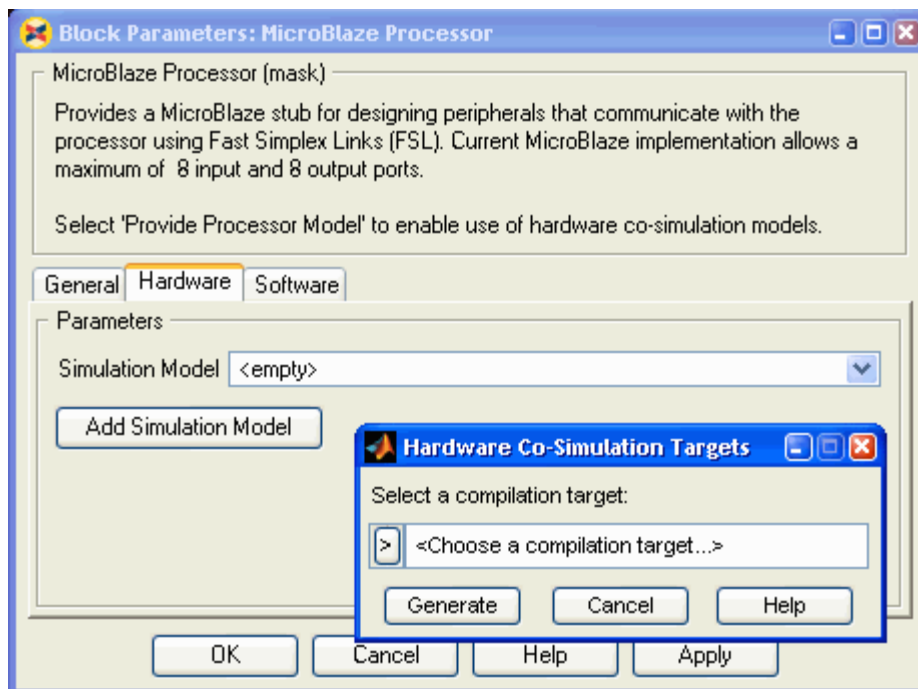


Figure 5: Block Parameters Dialog Box: Hardware Tab

The Hardware tab contains parameters for configuring the simulation model.

- **Simulation Model:** The System Generator processor core cache will be empty when the block is first used, so no simulation models will be available in the 'Simulation Model' pull down menu.
A simulation model must be tied to a hardware platform (board) so that the MicroBlaze processor is aware of board specific information, such as whether a RS232 port is available. If a simulation model is present in the core cache, it will be listed in the 'Simulation Model' pull down menu under its compilation target (board) name, e.g. `companyxyz_boardnm_partnm_packagenum_rev_1`.
- **Add Simulation Model:** Clicking on the 'Add Simulation Model' button will cause the 'Hardware Co-Simulation Targets' dialog box to appear. This allows a compilation target to be specified. Clicking the 'Generate' button will set in motion a series of compilations that may take sometime to complete:
 1. An EDK project is created in the Target Directory specified in the System Generator token block. (XPS)
 2. The EDK project is netlisted. (EDK+XFlow)
 3. A hardware co-simulation token is created out of the EDK netlists, and saved in the System Generator core cache. (System Generator, XFlow)
 4. EDK software libraries are compiled. (EDK)

Note: When creating a new board support package using the SBDBuilder tool, it is important to specify the board's system Reset and RS232 ports as non-memory mapped ports, otherwise they will not be correctly routed. Further, the port name for the reset port should be labeled as 'Reset', the port name for the receive port of the RS232 port should be labeled as 'RXD' and the transmit port should be labeled as 'TXD'. Labeling these ports in this manner allows System Generator to correctly detect and route the signals to the relevant pins.

If a RS232 port is available, a UART with the following parameters is created in the MicroBlaze:

Baud rate (bits per second): 115200, Data bits: 8, Parity: None, Stop bits: 1, Flow control: none. The `stdin` and `stdout` channels will be mapped to the UART, allowing for input and output via the RS232.

Software Tab

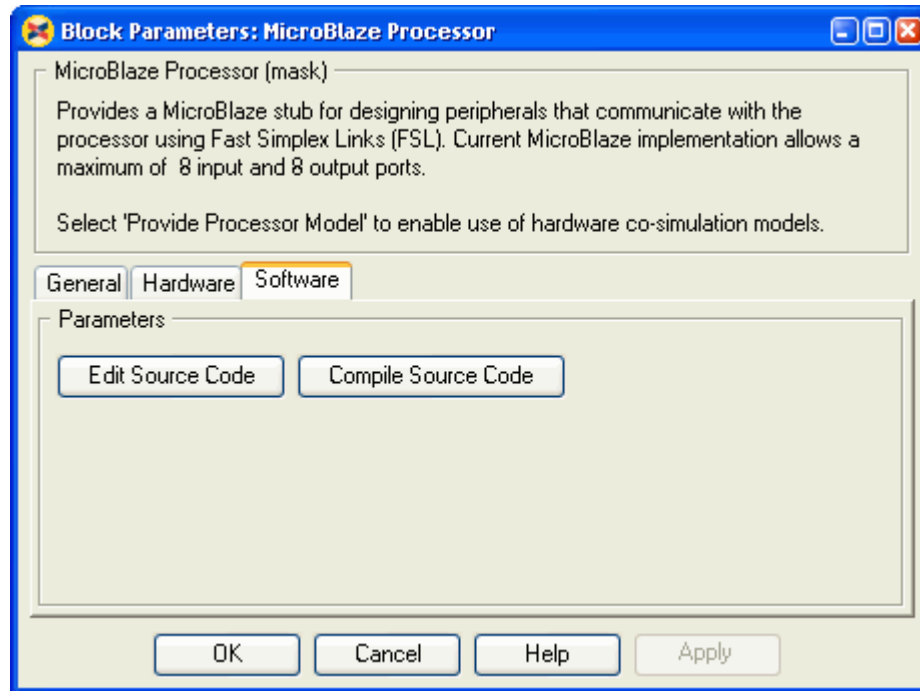


Figure 6: Block Parameters Dialog Box: Software Tab

The Software Tab provides buttons to edit and compile the source code that will execute in the simulation model.

- **Edit Source Code:** Clicking on the 'Edit Source Code' button will bring up the source code associated with this MicroBlaze block. This code is used only by the simulation model.

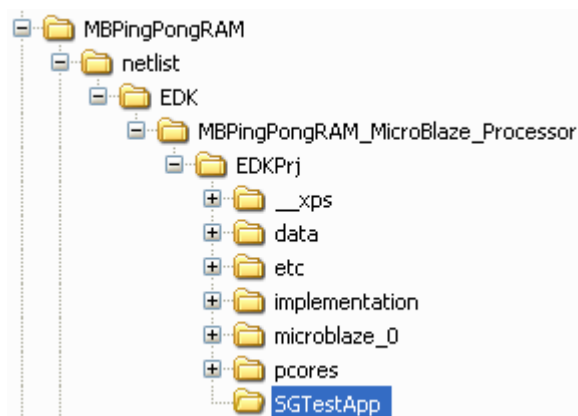


Figure 7: Directory structure

Figure 7 shows the directory structure created by System Generator. In this case, "MBPingPongRAM" is the name of the Simulink model and "netlist" is the user specified Target

Directory. The EDK project is generated under a directory created by appending the model name to the MicroBlaze's full path. The source code associated with the block is kept in the highlighted directory; SGTestApp, and is called MainProg.c.

The default Matlab editor is used to edit the source file. This is a user configurable option in Matlab and can be edited from the Matlab menu bar: File->Preferences, under the Editor/Debugger section.

- **Compile Source Code:** Clicking on 'Compile Source Code' compiled the source code and updates the hardware co-simulation bit file with the binary code created.

MicroBlaze Software Issues

Accessing FSL Peripherals from Software

System Generator peripherals can be accessed by the MicroBlaze through assembly instructions that access the relevant FSLs. The EDK provides eight C macros that simplifies read and writes to FSLs. Please refer to the EDK documentation for more details.

Non-blocking data read and write
microblaze_nbread_datafsl(val,id); microblaze_nbwrite_datafsl(val,id);
Non-blocking control read and write
microblaze_nbread_cntlfsl(val,id); microblaze_nbwrite_cntlfsl(val,id);
Blocking data read and write
microblaze_bread_datafsl(val,id); microblaze_bwrite_datafsl(val,id);
Blocking control read and write
microblaze_bread_cntlfsl(val,id); microblaze_bwrite_cntlfsl(val,id);

In the macro calls shown above, val refers to the 32-bit data value that will be read or written to the FSL. The id parameter refers to the FSL being accessed. Blocking read or write will stall the MicroBlaze until a read or write can occur. Non-blocking read or writes will not stall the MicroBlaze even if a read or write was unable to complete. A data write will write val to the FSL's data port and a 0 (zero) to the FSL's control port. A control write will write val to the FSL's data port and a 1 (one) to the FSL's control port. Please refer to the EDK MicroBlaze documentation and the following System Generator tutorials for more information:

- Designing and Exporting MicroBlaze Processor Peripherals
- Designing and Simulating MicroBlaze Processor Systems

Correspondence between EDK FSL buses and System Generator ports

It is important to understand the difference between FSL instances and FSL bus connections in the EDK. The MicroBlaze contains sixteen bus connections that can connect to FSLs (or any peripheral that mimics the FSL interface). Eight of these bus connections are inputs and are called SFSL (for Slave FSL) in the EDK. The other eight bus connections are outputs and are called MFSL (for Master FSL) in the EDK. FSL instances refer to the physical implementation of the FIFO hardware which

makes up an FSL.

In the EDK, the SFSL and MFSL bus connection numbers are independent to FSL instances. In other words, SFSL0 need not be connect to instance 0 of an FSL and similarly MFSL1 need not be connected to instance 1 of an FSL. This means that in a System Generator block, port In0_* need not be connected to SFSL0. The consequence of this is that a user has to be aware of the FSL connectivity to write correct software code. The assumption that a `microblaze_nbwrite_datafsl` instruction to FSL0 will output data to System Generator in Out0_*, is not necessarily correct.

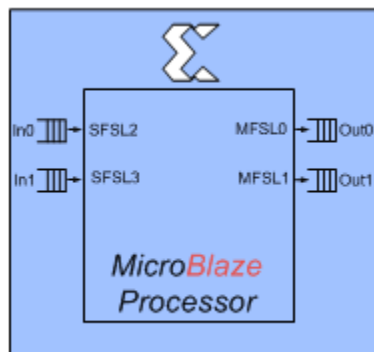


Figure 8: A System Generator MicroBlaze block

The access functions provided by the EDK for accessing FSLs, refer to FSL bus connections and not FSL instances. Figure 8 shows a System Generator MicroBlaze block, with a MicroBlaze that has been manually configured by the user (the inner box). The MicroBlaze processor that the System Generator block represents is also shown (the outer box). Here In0 is connected to bus connection SFSL2 and In1 to SFSL3.

In the case shown in Figure 8, software code written to access data from In0 should access SFSL2. A non-blocking data read from In0 would thus be:

```
int val;
microblaze_nbread_datafsl(val,2);
```

Similarly a non-block write to Out0 would be:

```
microblaze_nbwrite_datafsl(val,0);
```

Reading and Writing to FSLs during Simulation

Normally, when exporting to the EDK, it is the user's responsibility to wire up bus connections. During simulation, System Generator provides a pre-configured MicroBlaze; In 0-7 is connected to SFSL 0-7, Out 0-7 is connected to MFSL 0-7.

During simulation, reading from FSL id 0, corresponds to reading from In0. Writing to FSL id 0 corresponds to writing to Out0.

FSL read and write errors

The MicroBlaze EDK documentation uses the big-endian naming convention to label buses. To be

consistent with that document, the following discussions will also use the big-endian naming convention; bit 0 corresponds to the most significant bit.

The MicroBlaze Status Register (MSR) stores status conditions in the MicroBlaze and can be used to determine if errors have occurred.

Reading the MSR Register for Error Condition

Errors that occur during FSL read and write operations are returned to the MicroBlaze MSR register. The MSR register can be read using the `mfs` MicroBlaze assembly instruction. It is recommended that the following macro be used to read the MSR register.

```
#define readmsr(val, dep) asm("mfs %0,rmsr" : "=d" (##val##) : "d" (dep))
```

Define the macro at the top of your C program and use it in the following manner:

```
int val, mymsr;  
microblaze_nbread_cntlfs1(val, 0);  
readmsr(mymsr, val);
```

The macro establishes dependence between the writing of the `val` register and the reading of the MSR register. This is especially important when used within a loop; if a dependency is not created, the compiler may move the `mfs` instruction out of the loop, when it performs software code optimizations.

FSL Read Errors

Reads from FSLs can fail in two possible ways: *data invalid* and *FSL error*. A *data invalid* occurs when a blocking or non-blocking read fails because there is no data in the FSL. An *FSL error* occurs when a blocking or non-blocking control read is used to read a data value that does not have the control flag set to one. Similarly, an *FSL error* also occurs when a blocking or non-blocking data read is used to read a data value that has the control flag set to one.

When a *data invalid* error occurs, the MSR's carry flag is set high. The mask for the carry flag is 0x4 and corresponds to bit 29. The carry flag is also replicated on the most significant bit (bit 0) of the MSR register.

When an *FSL error* occurs, the FSL flag is set high. The mask for the FSL flag is 0x10 and corresponds to bit 27.

FSL Write Errors

Writes to FSLs fail if the FSL being written is full. Both blocking and non-blocking writes return a *data invalid* error when attempts to write to a full FSL are detected.

When a *data invalid* error occurs, the MSR's carry flag is set high. The mask for the carry flag is 0x4 and corresponds to bit 29. The carry flag is also replicated on the most significant bit (bit 0) of the MSR register.

Known Issues

- Only one MicroBlaze block per design is supported. However, it is possible to connect multiple MicroBlaze blocks to the created FSL interfaces from the EDK side.
- The MicroBlaze must be instantiated on the top level when exporting to EDK. When using hardware co-simulation model, it is important to guard FSL reads and writes and check for error conditions. When simulation starts, the MicroBlaze program will execute in hardware and may read or write from FSLs before the System Generator model has a chance to execute. Refer to Designing and Simulating MicroBlaze Processor Systems, EDK Export Tool for full explanation.
- Verilog netlisting is not supported for this block.

Online Documentation for the MicroBlaze Processor

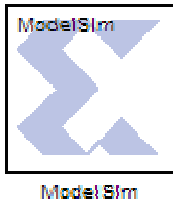
More information for the MicroBlaze can be found at
http://www.xilinx.com/xlnx/xebiz/designResources/ip_product_details.jsp?key=micro_blaze

See Also

Designing and Exporting MicroBlaze Processor Peripherals, Designing and Simulating MicroBlaze Processor Systems, EDK Export Tool

ModelSim

This block is listed in the following Xilinx Blockset libraries: Tools and Index.



The System Generator block box provides a way to incorporate existing HDL files into a model. When the model is simulated, co-simulation can be used to allow black boxes to participate. The ModelSim HDL co-simulation block configures and controls co-simulation for one or several black boxes.

During a simulation, each ModelSim block spawns one copy of ModelSim, and therefore uses one ModelSim license. If licenses are scarce, several black boxes can share the same block.

In detail, the ModelSim block does the following:

- Constructs the additional VHDL and Verilog needed to allow black box HDL to be simulated inside ModelSim;
- Spawns a ModelSim session when a Simulink simulation starts;
- Mediates the communication between Simulink and ModelSim;
- Reports if errors are detected when black box HDL is compiled;
- Terminates ModelSim, if appropriate, when the simulation is complete.

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

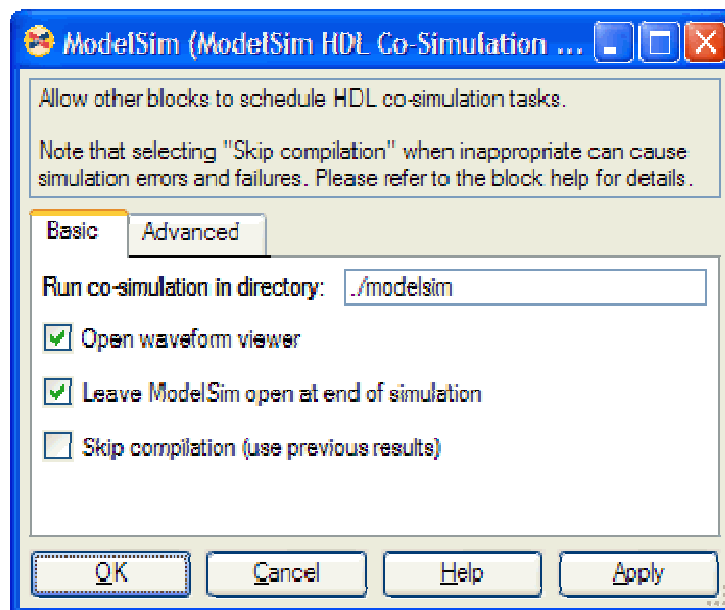


Figure 1(a): ModelSim block parameters dialog box (showing basic parameters)

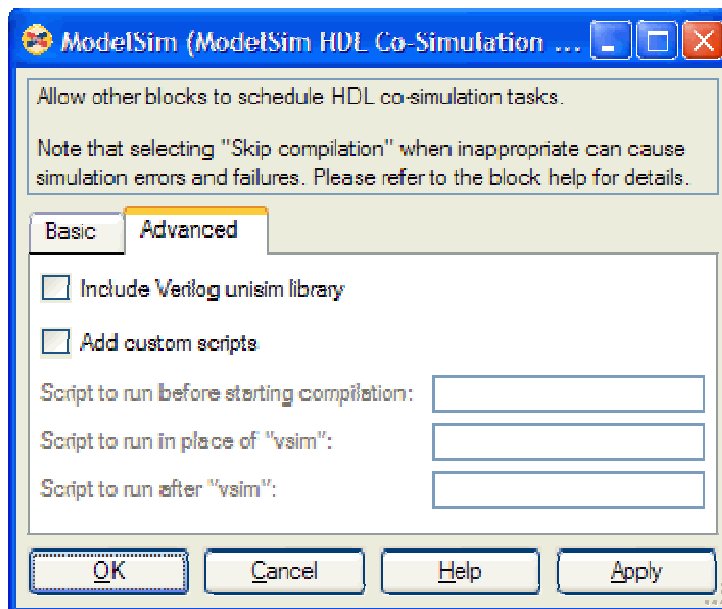


Figure 2(b): ModelSim block parameters dialog box (showing advanced parameters)

Parameters specific to the ModelSim block are:

- **Run co-simulation in directory (Basic):** ModelSim is started in the directory named by this field. The directory is created if necessary. All black box files are copied into this directory, as are the auxiliary files System Generator produces for co-simulation. Existing files are overwritten silently. The directory can be specified as an absolute or relative path. Relative paths are interpreted with respect to the directory in which the Simulink .mdl file resides.
- **Open waveform viewer (Basic):** When this checkbox is selected, the ModelSim waveform window opens automatically, displaying a standard set of signals. The signals include all inputs and outputs of all black boxes and all clock and clock enable signals supplied by System Generator. The signal display can be customized with an auxiliary tcl script. To specify the script, select Add Custom Scripts and enter the script name (e.g., myscript.do) in the Script to Run After vsim field. An example showing a customized waveform viewer is included in <path_to_sysgen>/examples/black_box/example5. That example is also discussed in another section of the System Generator User Guide.
- **Leave ModelSim open at end of simulation (Basic):** When this checkbox is selected, the ModelSim session is left open after the Simulink simulation has finished.
- **Skip compilation (use previous results) (Basic):** When this checkbox is selected, the ModelSim compilation phase is skipped in its entirety for all black boxes that are using the ModelSim block for HDL co-simulation. To select this option is to assert that: (1) underneath the directory in which ModelSim will run, there exists a ModelSim work directory, and (2) that the work directory contains up-to-date ModelSim compilation results for all black box HDL. Selecting this option can greatly reduce the time required to start-up the simulation, however, if it is selected when inappropriate, the simulation can fail to run or run but produce false results.
- **Include Verilog unisim library (Advanced):** Selecting this checkbox ensures that ModelSim includes the Verilog UniSim library during simulation. Note: the Verilog unisim library must be mapped to UNISIMS_VER in ModelSim. In addition, selecting this checkbox ensures the "gbl.v" module is compiled and invoked during simulation.
- **Add custom scripts (Advanced):** Selecting this checkbox exposes fields Script to Run Before Starting Compilation, Script to Run in Place of "vsim", and Script to Run after "vsim". Scripts entered in these fields are not run unless the checkbox is selected.
- **Script to run before starting compilation (Advanced):** The script listed here is executed by ModelSim before compiling black box HDL files.

- **Script to run in place of "vsim" (Advanced):** The script listed here is executed by ModelSim at the point when System Generator would ordinarily instruct ModelSim to begin a simulation.

If this parameter is left blank, or Add custom scripts is not selected, then System Generator instructs ModelSim to execute the default command `vsim $stoplevel -title {System Generator Co-Simulation (from block $blockname)}`. Here `$stoplevel` is the name of the top level entity for simulation (e.g., `work.my_model_mti_block`) and `$blockname` is the name of the ModelSim block in the Simulink model associated with the current co-simulation. To avoid problems, certain characters in the block name (e.g., newlines) are sanitized.

If this parameter is not blank and Add custom scripts is selected, then System Generator instead instructs ModelSim to execute `do $* $stoplevel $blockname`. Here `$stoplevel` and `$blockname` are as above and `$*` represents the literal text entered in the field. If, for example the literal text is `foo.do`, then ModelSim executes `foo.do`. `foo.do` can reference `$stoplevel` and `$blockname` as `$1` and `$2`, respectively. Thus, `foo.do` can run `vsim` on `topLevel` by executing `vsim $1`.

- **Script to run after "vsim" (Advanced):** The script listed here is executed by ModelSim after all HDL for black boxes has been successfully compiled, and after the ModelSim simulation has completed successfully. If the Open Waveform Viewer checkbox has been selected, System Generator issues all commands it ordinarily uses to open and customize the waveform viewer before running this script. This allows the user to customize the waveform viewer as desired (either by adding signals to the default viewer or by creating a fully custom viewer). The black box tutorial includes an example that customizes the waveform viewer.

It is often convenient to use relative paths in a custom script. Relative paths are interpreted with respect to the directory that contains the model's `.mdl`. A relative path in the Run co-simulation in directory field is also interpreted with respect to the directory that contains the model's `.mdl` file. Thus, for example, if Run co-Simulation in directory specifies `./modelsim` as the directory in which ModelSim should run, the relative path `../foo.do` in a script definition field refers to a file named `foo.do` in the directory that contains the `.mdl`.

Fine Points

The time scale in ModelSim matches that in Simulink, i.e., one second of Simulink simulation time corresponds to one second of ModelSim simulation time. This makes it easy to compare times at which events occur in the two settings. The typically large Simulink time scale is also useful because it allows System Generator to schedule events without running into problems related to the timing characteristics of the HDL model. Users needn't worry too much about the details System Generator event scheduling in co-simulation models. The following example is offered to illustrate the broader points.

This example model shown here can be found in the System Generator directory `<path_to_sysgen>/example/black_box/example4`. The example is also discussed in another section of the *System Generator User Guide*.

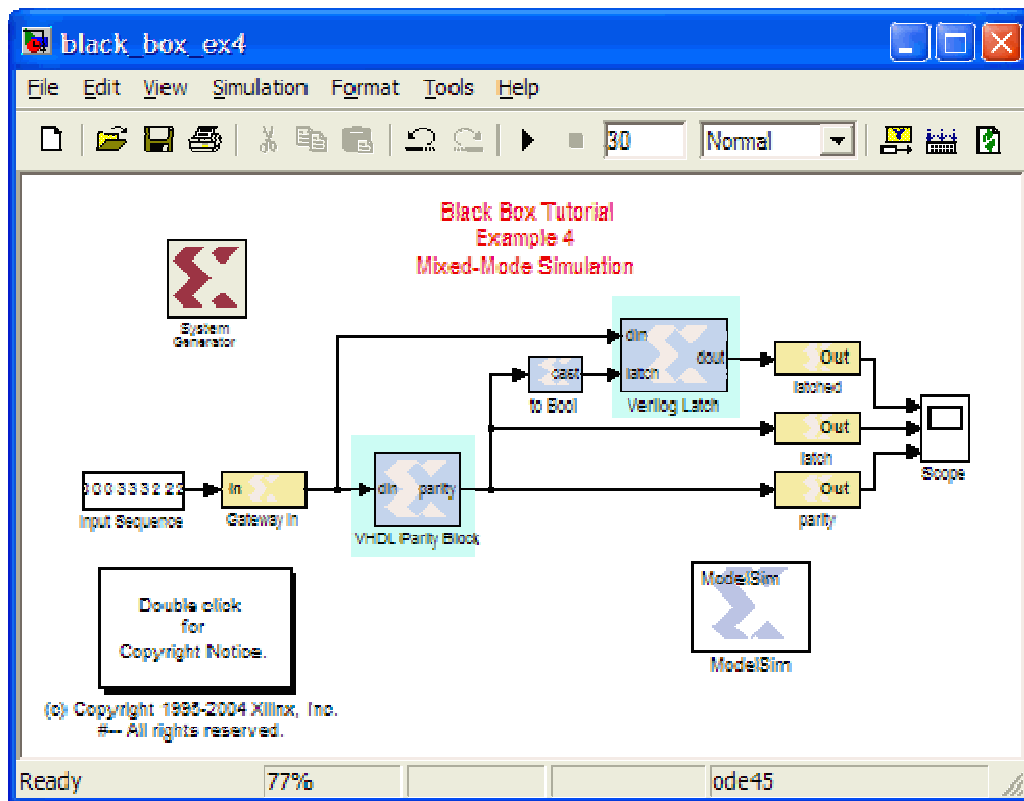
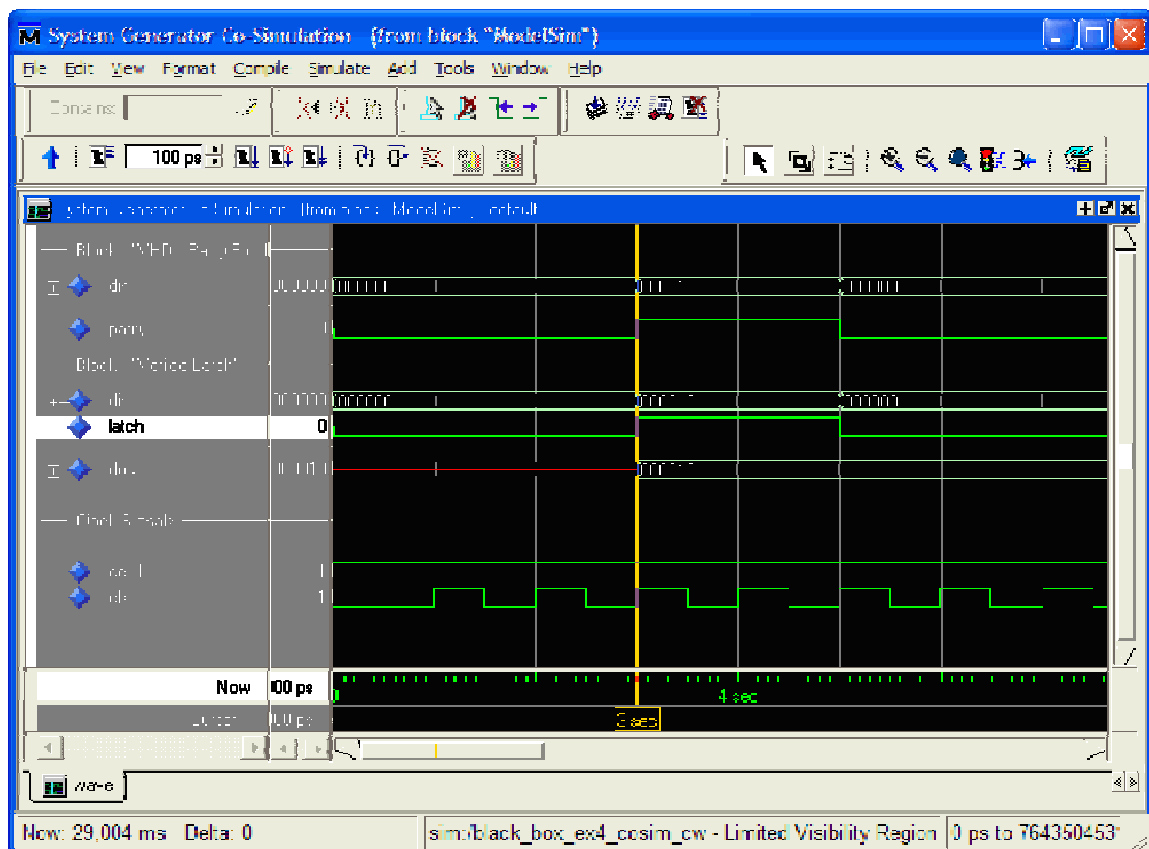
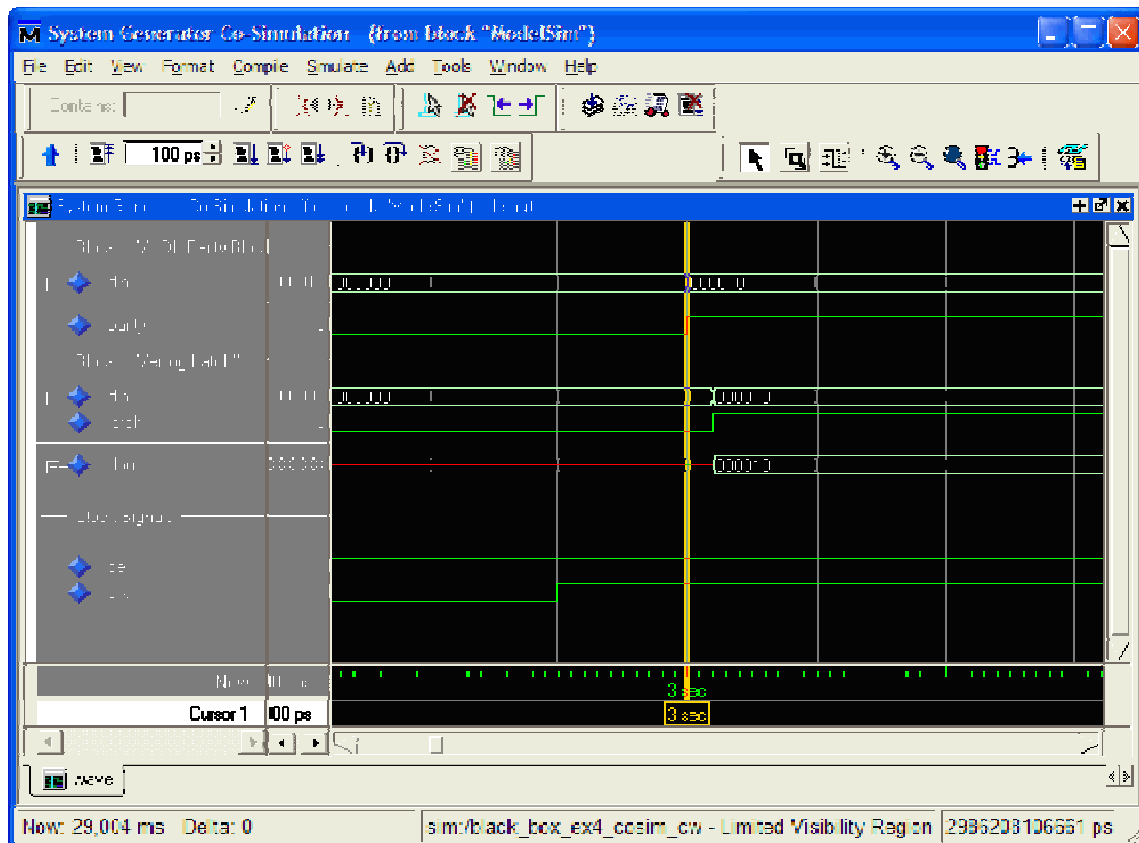


Figure 3: ModelSim Black Box Example

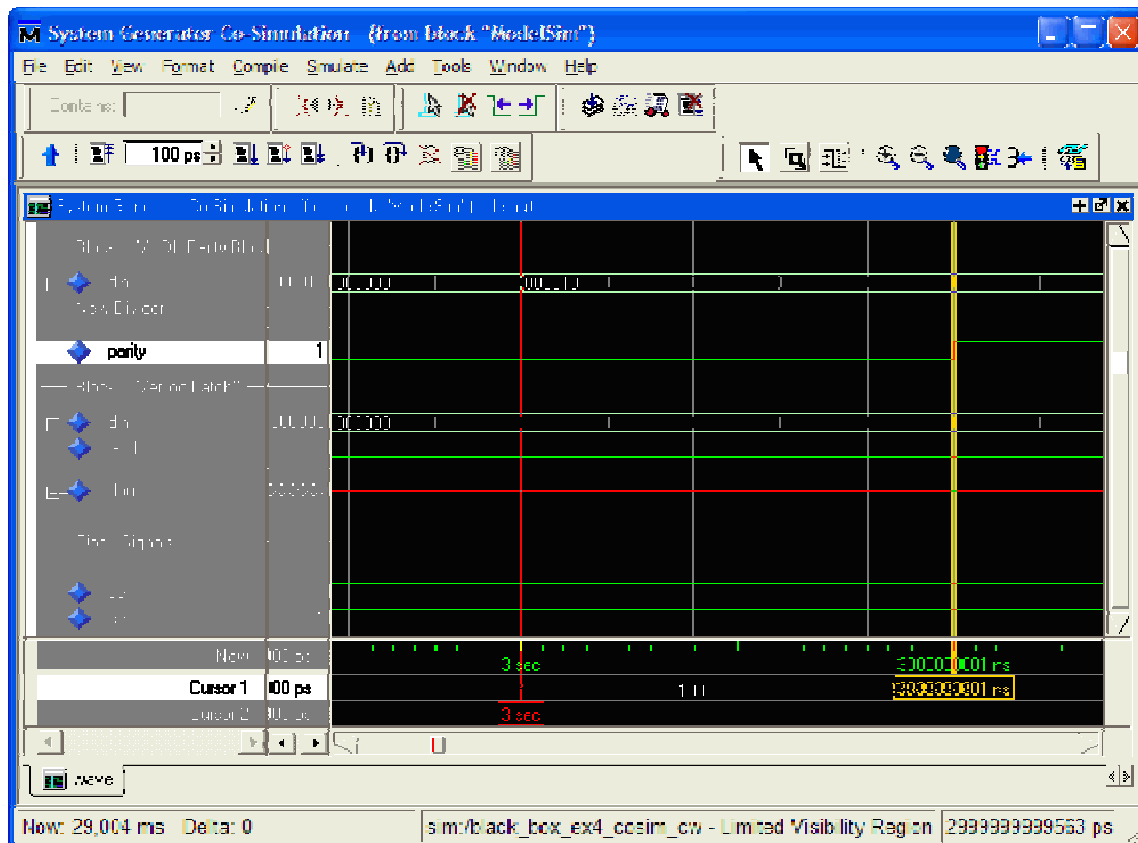
When the above model is run, the following waveforms are displayed by ModelSim:



At the time scale presented here (the above shows a time interval of six seconds), the rising clock edge at three seconds and the corresponding transmission of data through each of the two black boxes appear simultaneous, much as they do in the Simulink simulation. Looking at the model, however, it is clear that the output of the first black box feeds the second black box. Both of the black boxes in this model have combinational feed-throughs, i.e., changes on inputs translate into immediate changes on outputs. Zooming in toward the three second event reveals how System Generator has resolved the dependencies. Note the displayed time interval has shrunk to ~20 ms.



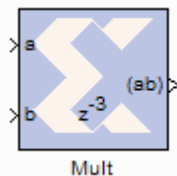
The above figure reveals that System Generator has shifted the rising clock edge so it occurs before the input value is collected from Simulink and presented to the first of the black boxes. It then allows the value to propagate through the first black box and presents the result to the second at a slightly later time. Zooming in still further shows that the HDL model for the first black box includes a propagation delay which System Generator has effectively abstracted away through the use of large time scales. The actual delay through the first black box (exactly 1 ns) can be seen in the figure below.



In propagating data through black box components, System Generator allocates 1/ 1000 of the system clock period down to 1us, then shrinks the allocation to 1/100 of the system clock period down to 5ns, and below that threshold resorts to delta-delay stepping, i.e. issuing "run 0 ns" commands to ModelSim. If the HDL includes timing information (e.g., transport delays) and the Simulink System Period is set too low, then the simulation results will be incorrect. The above model begins to fail when the Simulink system period setting is reduced below 5e-7, which is the point at which System Generator resorts to delta-delay stepping of the black boxes for data propagation.

Mult

This block is listed in the following Xilinx Blockset libraries: *Math and Index*.



The Xilinx Mult block implements a multiplier. It computes the product of the data on its two input ports, producing the result on its output port.

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

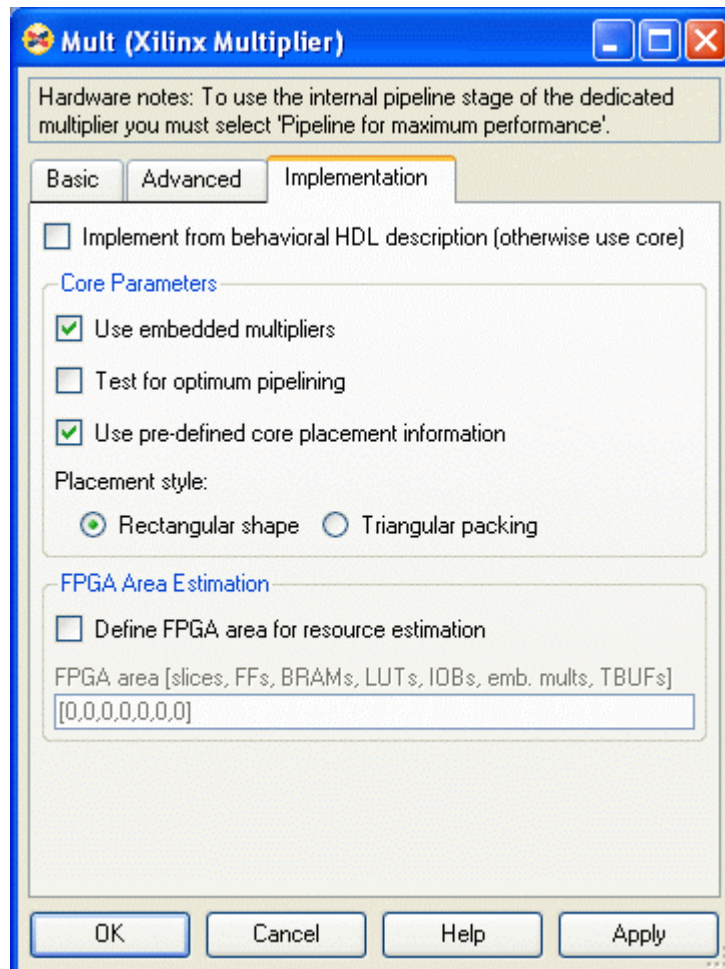


Figure 1: Mult block Implementation tab

Parameters specific to the Mult block are:

- **Use embedded multipliers:** when checked, directs the core to use embedded multipliers (available for parallel multipliers in Virtex-II, Virtex-4 and Spartan-3 only).
- **Test for optimum pipelining:** Checks if the Latency provided is at least equal to the optimum pipeline length. Latency values that pass this test imply that the core produced would be optimized for speed of operation.

Other parameters used by this block are explained in the Common Parameters section at the beginning of this chapter.

Xilinx LogiCORE

The Multiplier block uses the Xilinx LogiCORE Multiplier Generator except when Implement from behavioral HDL description is checked.

Multiple Subsystem Generator

This block is listed in the following Xilinx Blockset libraries: Shared Memory and Index.



Multiple Subsystem Generator

The Xilinx Multiple Subsystem Generator block wires two or more System Generator designs into a single top-level HDL component that incorporates multiple clock domains. This top-level component includes the logic associated with each System Generator design and additional logic to allow the designs to communicate with one another. In software, this communication is handled using shared memory and shared memory derivative blocks (e.g., Shared Memory, To/From FIFO, and To/From Register blocks). In hardware, the designs are interfaced to hardware implementations (e.g., dual-port memory, asynchronous FIFOs, and registers) of their shared memory counterparts, making it possible to partition and implement systems with multiple clock domains. The block parameters dialog box for the Multiple Subsystem Generator block is shown in Figure 1.

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the Multiple Subsystem Generator icon in your Simulink model.

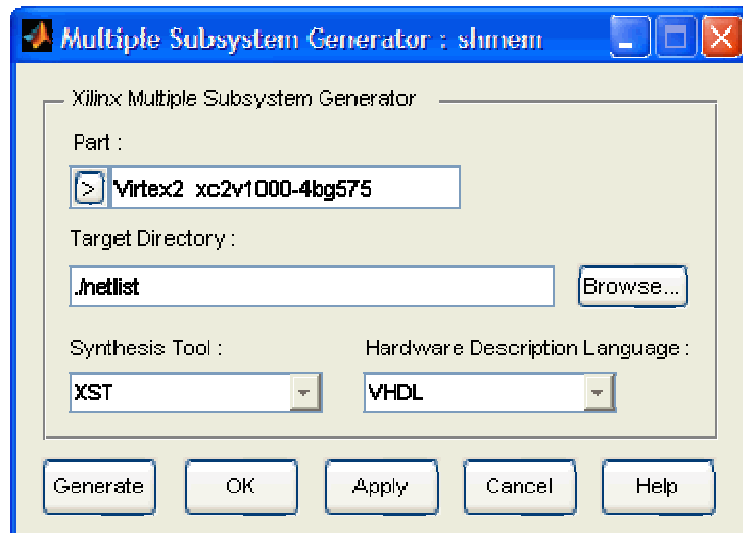


Figure 1: Multiple Subsystem Generator block parameters dialog box

Parameters specific to the Multiple Subsystem Generator block are:

- **Part:** Defines the FPGA part to be used.
- **Target Directory:** Defines where System Generator should write compilation results. Because System Generator and the FPGA physical design tools typically create many files, it is best to specify a separate target directory, i.e., a directory other than the directory containing your Simulink model files.
- **Synthesis Tool:** Specifies the tool to be used to synthesize the design. Tool choices are Synplify's Synplify Pro or Synplify, and Xilinx's XST.
- **Hardware Description Language:** Tells the type of HDL language (Verilog or VHDL) that should be generated for each design.

Design Generation

The Multiple Subsystem Generator block performs the following steps when a user presses the **Generate** button in the block's parameters dialog box :

1. It determines the System Generator designs that should be generated and wired together.
2. It configures each System Generator design with appropriate settings and generates the designs individually.
3. It produces hardware implementations (e.g., core netlists) for the shared memory blocks.
4. It generates a top-level HDL file that includes the System Generator designs wired together with the corresponding shared memory hardware implementations.

The Multiple Subsystem Generator block determines which subsystems to implement and wire together by searching for subsystems that contain System Generator blocks that reside at the same level of hierarchy as the Multiple Subsystem Generator block. Inclusion of the Multiple Subsystem Generator block in a Simulink design is restricted in the following ways:

- System Generator blocks may not be included in the same level of hierarchy as the Multiple Subsystem Generator block.
- There must be at least two master System Generator Blocks in subsystems located in the same level of hierarchy as the Multiple Subsystem Generator block.
- Only one Multiple Subsystem Generator block may be included in a given level of hierarchy.

For example, consider the example block diagram shown in Figure 2. This diagram comprises two subsystems, and it is assumed that each subsystem contains a System Generator block along with some amount of System Generator logic. Note that although only two subsystems are shown in the diagram, the Multiple Subsystem Generator block can accommodate any number of subsystems. A Multiple Subsystem Generator block is included in the same level of hierarchy as the two subsystems. When a user chooses to generate the overall design using the Multiple Subsystem Generator block, the subsystems are generated and then wired together.

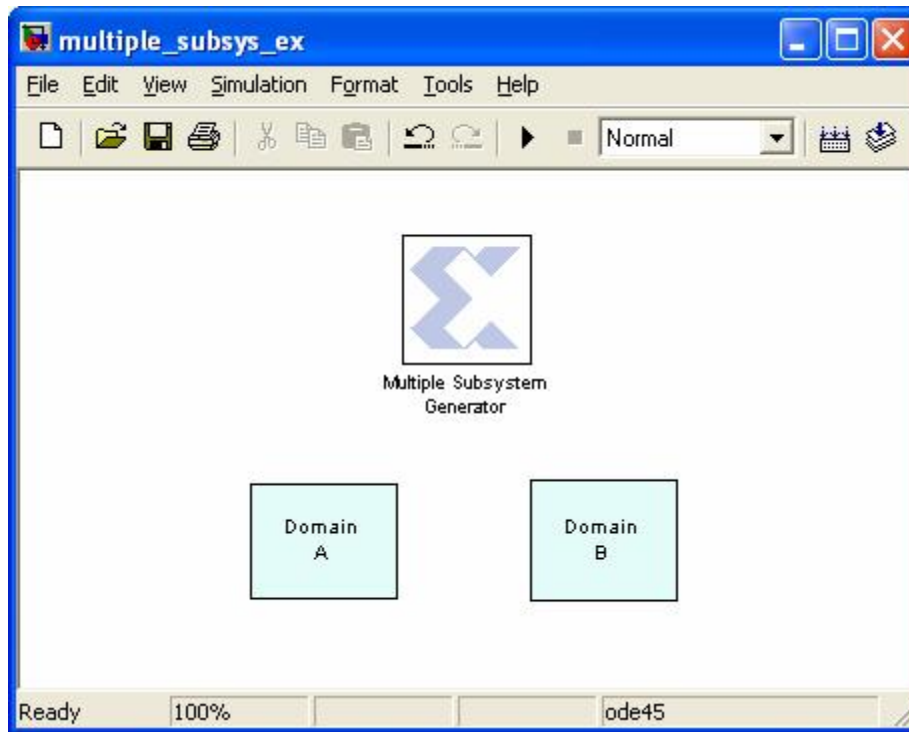


Figure 2: Diagram showing a Multiple Subsystem Generator block with two System Generator subsystems.

A subsystem that includes a master System Generator block is implemented using the NGC compilation target when the **Generate** button is pressed on the Multiple Subsystem Generator block. Using the NGC compilation target has the advantage of allowing the resulting HDL netlist, cores, and constraints to be delivered as a single netlist file. The HDL component that stitches the designs together instantiates the System Generator designs as black boxes; the NGC files provide the black box implementations. For the example shown in Figure 2, three separate NGC files would be generated – one corresponding to each subsystem.

Before a design is generated, it is configured with the **Part**, **Synthesis Tool**, and **Hardware Description Language** parameters specified in the Multiple Subsystem Generator dialog box. These settings override the settings of the master System Generator blocks. Note that the original System Generator block settings are restored once generation is complete.

Subsystems that are wired together using the Multiple Subsystem Generator block can communicate with one another using a pair of Shared Memory blocks, To/From FIFO blocks, or To/From Register blocks. The block pairs must be partitioned so that one block resides in one subsystem (e.g., To FIFO block) while the other partner half resides in a different subsystem (e.g., From FIFO block).

When the complete design is translated into hardware, the two FIFO halves are pulled out of their respective subsystems. The System Generator logic that was previously attached to shared memory ports (e.g., data in, data out) are then wired to new top-level ports for that design. This means that one subsystem HDL component includes ports for one half of the shared memory, while the other half has ports for the other shared memory side. A hardware implementation of the shared memory is then created and wired to the top-level shared memory ports.

Note: The Multiple Subsystem Generator block does not currently support multiple shared memory blocks referencing the same shared memory object in the same subsystem. For example, a To FIFO block cannot be used to communicate to two From FIFO blocks placed in other subsystems.

Consider an example with two subsystems, A and B, where subsystem A contains a To FIFO block and subsystem B contains a From FIFO block. The opposing halves of the FIFO specify the same shared memory name, `my_fifo`. When the design is netlisted using the Multiple Subsystem Generator block, the To FIFO and From FIFO blocks are removed from their respective subsystems, and merged into a single core implementation (e.g., Asynchronous FIFO Core). This process is shown in Figure 3.

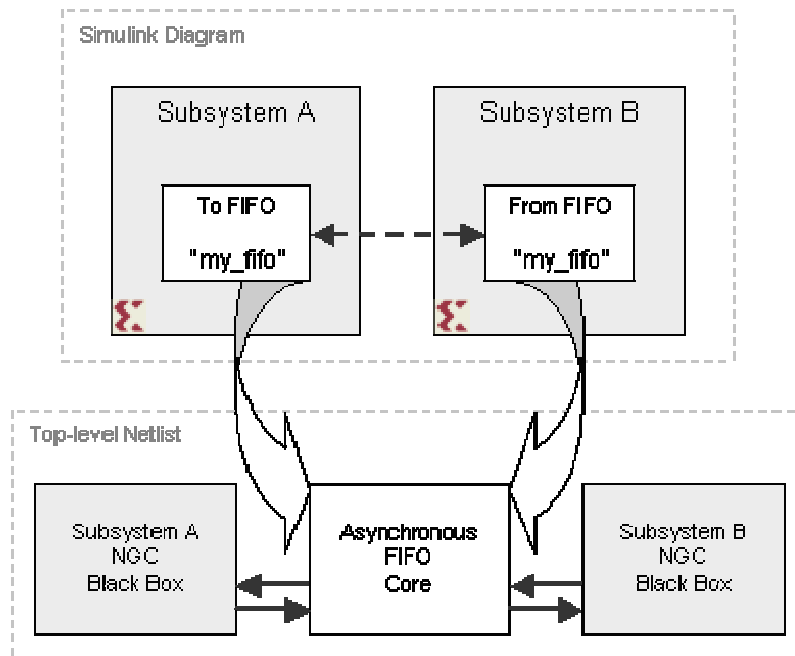


Figure 3: Hardware translation of a Simulink diagram containing two subsystems.

The table below provides the core or HDL component implementation that is used to implement shared memory and shared memory derivative blocks.

To Block	From Block	Hardware Implementation
Shared Memory	Shared Memory	Dual Port Block Memory 6.1
To FIFO	From FIFO	Fifo Generator 2.1
To Register	From Register	<code>synth_reg_w_init.(vhd,v)</code>

Note: Shared memory blocks should be used as the only means of communication between the subsystems. Do not use explicit System Generator signals to communicate between subsystems, as these are ultimately translated into top-level ports on the top-level HDL component that is created by the Multiple Subsystem Generator block.

All gateway ports included in the System Generator designs considered by the Multiple Subsystem Generator block are included in the top-level HDL component port interface. In addition, individual clock and clock enable ports are included in the port interface for each System Generator subsystem. The clock and clock enable port names are differentiated by the design name, which prefixes the port names. For example, consider the design shown in Figure 2. Assume the subsystem named `Domain A` has one input port named `inport_a` and one output port named `outport_a`. Also assume the subsystem named `Domain B` has one input port named `inport_b` and one output port named `outport_b`. The VHDL port interface for the resulting top-level entity is provided below :

```
entity multiple_subsys_ex is
port (
```

```

domain_a_ce: in std_logic := '1';
domain_a_clk: in std_logic;
domain_b_ce: in std_logic := '1';
domain_b_clk: in std_logic;
inport_a: in std_logic_vector(17 downto 0);
inport_b: in std_logic_vector(17 downto 0);
outport_a: out std_logic_vector(17 downto 0);
outport_b: out std_logic_vector(17 downto 0)
);
end multiple_subsys_ex;

```

Multiple Clock Support

Because each subsystem considered by the Multiple Subsystem Generator block has a master System Generator block, it is possible to specify different clocking information (e.g., Simulink system period, FPGA clock period) in each block. By specifying different Simulink system periods, each System Generator design can run at a different rate during simulation, allowing you to effectively model systems that utilize asynchronous clock domains.

The Multiple Subsystem Generator creates a separate clock port for each subsystem that was generated. The clock ports are then routed to the corresponding clock port on the System Generator design. When a design that uses multiple clocks is netlisted (i.e., translated from a high-level model into a lower level HDL description) the two shared memory halves are moved from their respective subsystems into the upper level of hierarchy. The two halves of the shared memory pair are then replaced with a single HDL component that implements the clock domain bridge (e.g., a dual-port memory). Clocks from the two domains are then connected to the opposing sides of the bridge component, along with the necessary data and control signals.

Files Generated

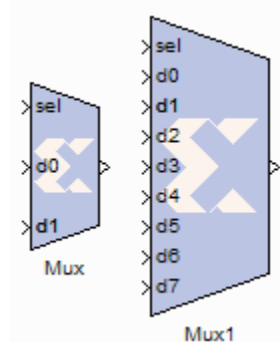
The Multiple Subsystem Generator produces several low level files when the **Generate** button is pushed. These files are written to the target directory specified on the Multiple Subsystem Generator block dialog box. The key files produced by this block are defined in the following table:

File Name or Type	Description
<design>.vhd (or .v)	Top-level HDL component that contains the System Generator designs stitched together.
.edn files	Besides writing HDL, the Multiple Subsystem Generator runs CORE Generator to implement shared memory hardware implementations. Coregen writes EDIF files whose names typically look something like multiplier_virtex2_6_0_83438798287b830b.edn.
globals	This file consists of key/value pairs that describe the design. The file is organized as a Perl hash table so that the keys and values can be made available to Perl scripts using Perl evals.
<design>.xcf (or .ncf)	This contains timing and port location constraints. These are used by the Xilinx synthesis tool XST and the Xilinx implementation tools. If the synthesis tool is set to something other than XST, then the suffix is changed to .ncf.
hdlFiles	This tells full list of HDL files written by the Multiple Subsystem Generator block. The files are listed in the usual HDL dependency order.

<design>.npl	This allows the HDL and EDIF to be brought into the Xilinx project management tool Project Navigator.

Mux

This block is listed in the following Xilinx Blockset libraries: Basic Elements, Control Logic, and Index.



The Xilinx Mux block implements a multiplexer.

The block has one select input (type unsigned) and a user-configurable number of data bus inputs, ranging from 2 to 1024.

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

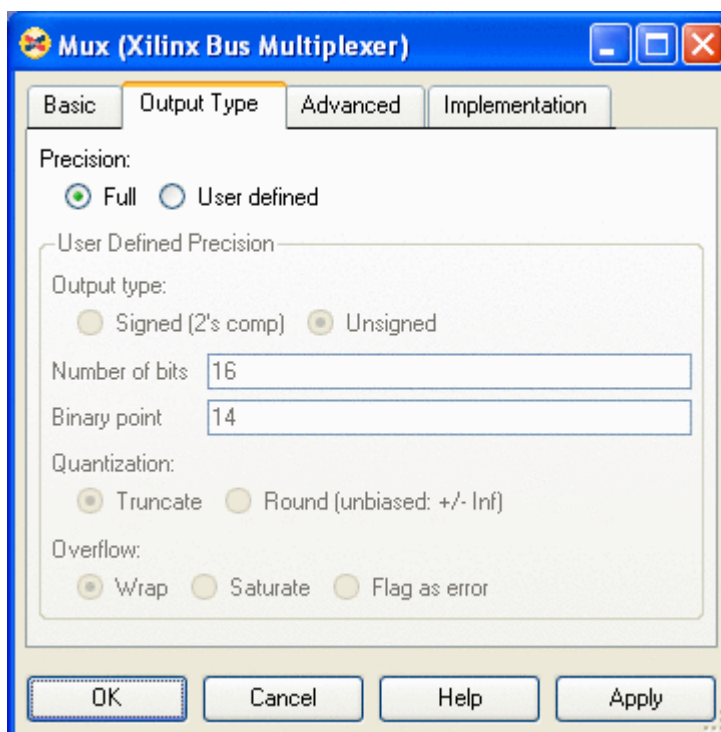
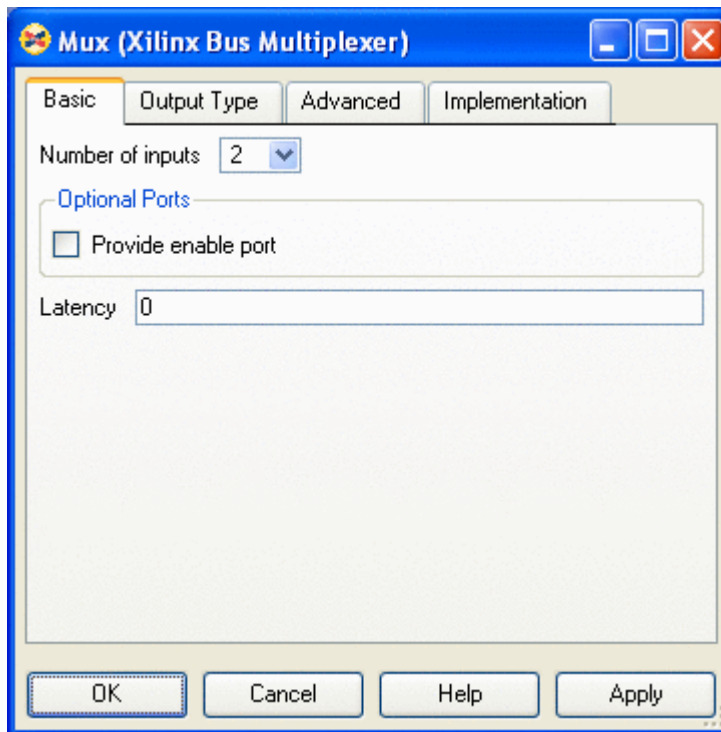


Figure 1: Mux block parameters dialog box

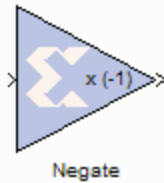
Parameters specific to the block are:

- **Number of Inputs:** specifies the number of data bus inputs, from 2 to 32.

Other parameters used by this block are described in the Common Parameters section at the beginning of this chapter.

Negate

This block is listed in the following Xilinx Blockset libraries: *Math and Index*.



The Xilinx Negate block computes the arithmetic negation (two's complement) of its input.

The block can be implemented either as a Xilinx LogiCORE or as a synthesizable VHDL module.

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

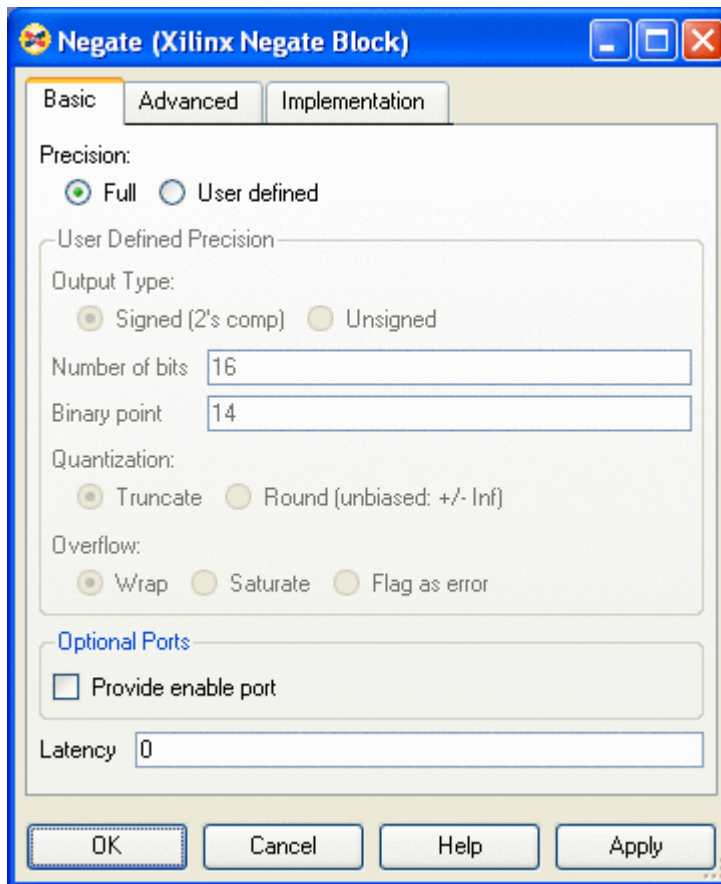
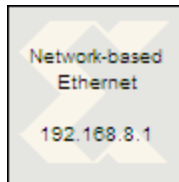


Figure 1: Negate block parameters dialog box

Parameters used by this block are explained in the Common Parameters section at the beginning of this chapter.

Network-based Ethernet Co-Simulation



The Xilinx Network-based Ethernet Co-Simulation block provides an interface to perform hardware co-simulation through an Ethernet connection over the IPv4 network infrastructure. Refer to Network-based Ethernet Hardware Co-Simulation for further details about the interface, its prerequisites and setup procedures.

The port interface of the co-simulation block varies. When a model is implemented for network-based Ethernet hardware co-simulation, a new library is created that contains a custom network-based Ethernet co-simulation block with ports that match the gateway names (or port names if the subsystem is not the top level) from the original model. The co-simulation block interacts with the FPGA hardware platform during a Simulink simulation. Simulation data that is written to the input ports of the block are passed to the hardware by the block. Conversely, when data is read from the co-simulation block's output ports, the block reads the appropriate values from the hardware and drives them on the output ports so they can be interpreted in Simulink. In addition, the block automatically opens, configures, steps, and closes the platform.

Block Parameters Dialog Box

The dialog box tabs specific to the network-based Ethernet hardware co-simulation block are shown below.

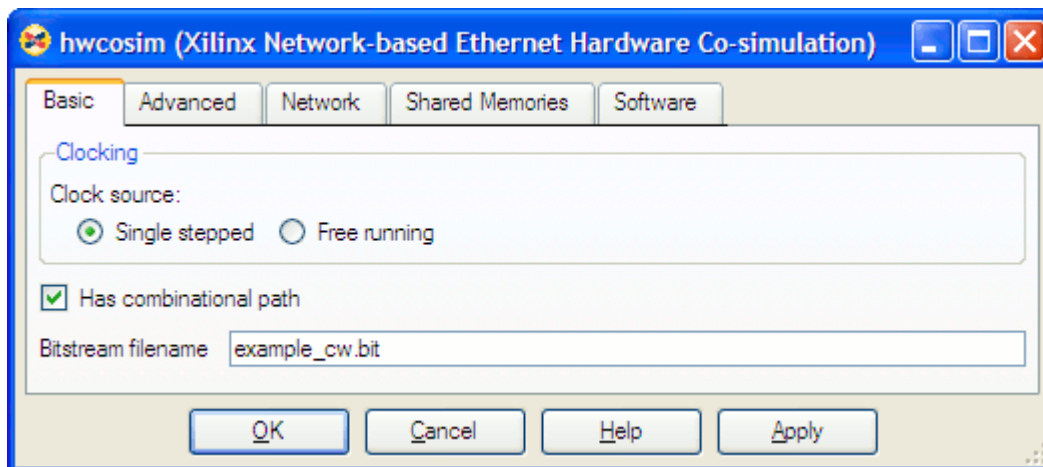


Figure 1: Network-based Ethernet Co-simulation Block Basic Tab

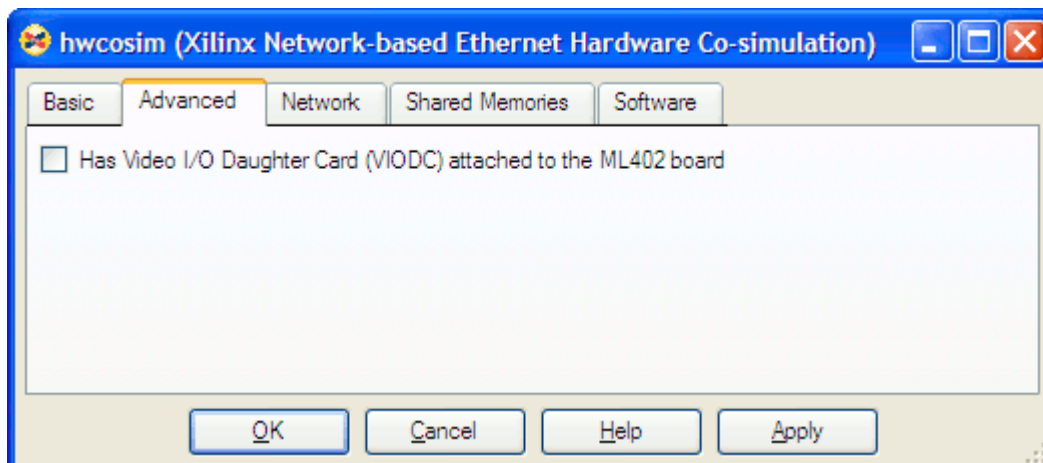


Figure 2: Network-based Ethernet Co-simulation Block Advanced Tab

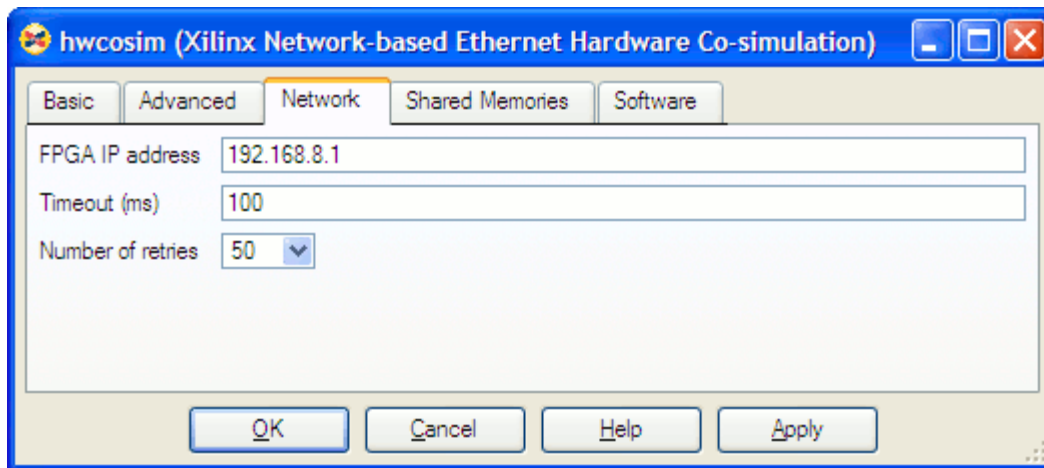


Figure 3: Network-based Ethernet Co-simulation Block Network Tab

Parameters specific to the block are:

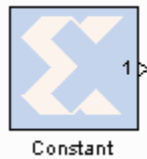
- **Clock source (Basic):** You may select between **Single stepped** and **Free running** clock sources. Selecting a **Single stepped** clock allows the block to step the board one clock cycle at a time. Each clock cycle step corresponds to some duration of time in Simulink. Using this clock source ensures the behavior of the co-simulation hardware during simulation will be bit and cycle accurate when compared to the simulation behavior of the subsystem from which it originated. Sometimes single stepping is not necessary and the board can be run with a **Free Running** clock. In this case, the board will operate asynchronously to the Simulink simulation.
- **Has Combination Path (Basic):** Sometimes it is necessary to have a direct combinational feedback path from an output port on a hardware co-simulation block to an input port on the same block (e.g., a wire connecting an output port to an input port on a given block). If you require a direct feedback path from an output to input port, and your design does not include a combinational path from any input port to any output port, unchecking this box will allow the feedback path in the design.
- **Bitstream filename (Basic):** Specifies the co-simulation FPGA configuration file for the network-based Ethernet hardware co-simulation platform. When a new co-simulation block is created during compilation, this parameter is automatically set so that it points to the appropriate configuration file. You need only adjust this parameter if the location of the configuration file changes.
- **Has Video I/O Daughter Card (VIODC) attached to the ML402 board (Advanced):** This option must be checked, if a Video I/O daughter card is attached on the target ML402 board, to allow proper device configuration. It should be otherwise unchecked.
- **FPGA IP address (Network):** Specify the IPv4 address associated with the target FPGA platform. The IP address must be specified using IPv4 dotted decimal notation (e.g. 192.168.8.1). Refer to Configuration using System ACE for details on configuring the IP address of the FPGA platform.
- **Timeout (Network):** Specifies the timeout value, in milliseconds, for packet retransmission in case of packet loss during the configuration and co-simulation process. The default value should suffice in the general case, but be advised that a larger value may be needed if the network connection is slow, with high latency, or congested.
- **Number of retries (Network):** Specifies the number of retries for packet retransmission in case of packet loss during the configuration and co-simulation process. The default value should suffice in the general case, but be advised that a larger value may be needed if the network connection experiences a considerably amount of packet loss.

See Also

Ethernet Hardware Co-Simulation, Network-based Ethernet Hardware Co-Simulation, Configuration using System ACE

Opmode

This block is listed in the following Xilinx Blockset libraries: *DSP and Index*.



The Xilinx Opmode block generates a constant that is a DSP48 or DSP48E instruction. The instruction is an 11-bit value for the DSP48 or a 15-bit value for the DSP48E. The instruction consists of the opmode, carry-in, carry-in select and either the subtract or alumode bits (depending upon the selection of DSP48 or DSP48E).

The Opmode block is useful for generating DSP48 or DSP48E control sequences. Figure 1 shows an example. The example implements a 35x35-bit multiplier using a sequence of four instructions in a DSP48 block. The opmode blocks supply the desired instructions to a multiplexer that selects each instruction in the desired sequence.

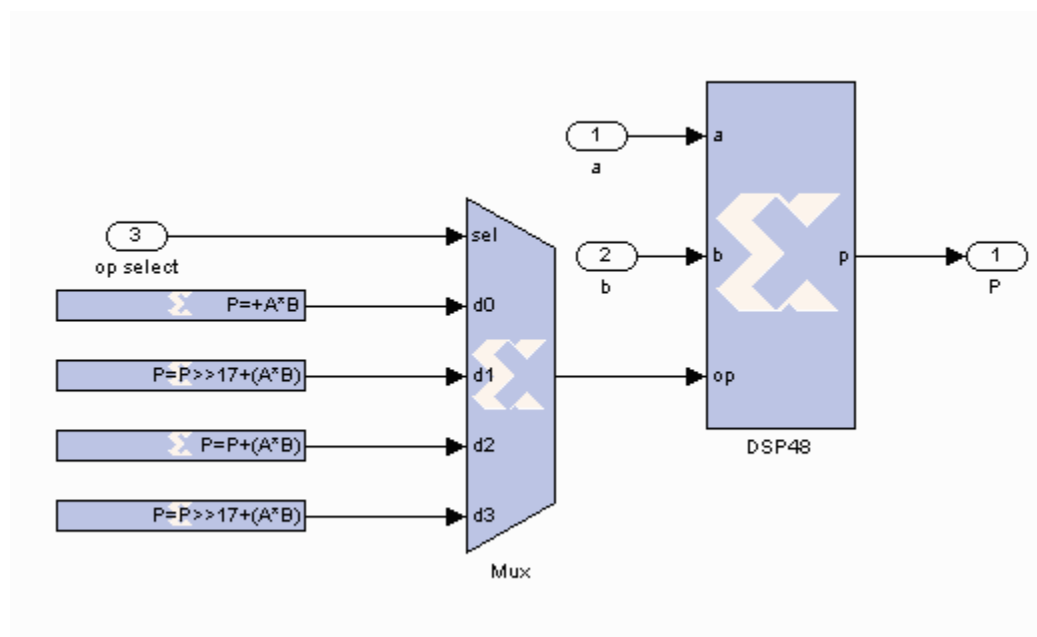


Figure 1: Using opmode blocks as control sequences for DSP48

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

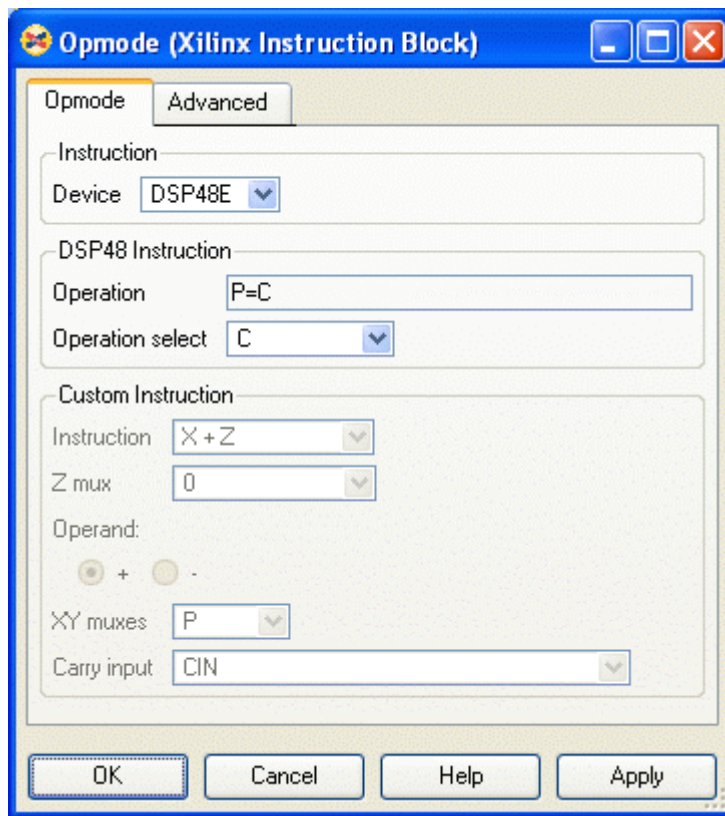


Figure 2: Opmode block parameters dialog box

Parameters specific to the block are:

- **Device:** specifies whether to generate an instruction for the DSP48 or DSP48E device.
- **Operation:** displays the instruction that is generated by the block (instruction is also displayed on the block).
- **Operation select:** allows the selection of a DSP48 or DSP48E instruction. Selecting custom reveals mask parameters that allow the formation of an instruction in the form $z\_mux \pm (y\_mux + carry)$.
- **Instruction:** available for the DSP48E. Determines what operation the adder is to perform or if a logic instruction should be performed instead of an arithmetic operation.
- **Z Mux:** specifies the 'Z' source to the DSP48(E)'s adder to be one of {'0', 'C', 'PCIN', 'P', 'C', 'PCIN>>17', 'P>>17'}.
- **Operand:** specifies whether the DSP48's adder is to perform addition or subtraction. In the DSP48E, the operand selection is made in the instruction pulldown.
- **YX Muxes:** specifies the 'YX' source to the DSP48's or DSP48E's adder to be one of {'0', 'P', 'A:B', 'A*B', 'C', 'P+C', 'A:B+C'}. 'A:B' implies that A is concatenated with B to produce a value to be used as an input to the adder.
- **Carry Input:** specifies the 'carry' source to the DSP48's adder to be one of {'0', '1', 'CIN', '~SIGN(P or PCIN)', '~SIGN(A:B or A*B)', '~SIGND(A:B or A*B)'}. '~SIGN (P or PCIN)' implies that the carry source is either P or PCIN depending on the Z Mux setting. '~SIGN(A*B or A:B)' implies that the carry source is either A*B or A:B depending on the YX Mux setting. The option '~SIGND (A*B or A:B)' selects a delayed version of '~SIGN(A*B or A:B)'.

Other parameters used by this block are explained in the Common Parameters section at the beginning of this chapter.

The Opmode block does not use a Xilinx LogiCORE.

See Also

The following documents are provided as part of System Generator User's guide and give more information on blocks that can be used in conjunction with opmode block :

DSP48

DSP48E

Appendix: DSP48 Control Instruction Format

Instruction Field Name	Location	Mnemonic	Description
YX Mux	op[3:0]	0	0
		P	DSP48 output register
		A:B	Concat inputs A and B (A is MSB)
		A*B	Multiplication of inputs A and B
		C	DSP48 input C
		P+C	DSP48 input C plus P
		A:B+C	Concat inputs A and B plus C register
Z Mux	op[6:4]	0	0
		PCIN	DSP48 cascaded input from PCOUT
		P	DSP48 output register
		C	DSP48 C input
		PCIN>>17	Cascaded input downshifted by 17
		P>>17	DSP48 output register downshifted by 17
Operand	op[7]	+	Add
		-	Subtract
Carry In	op[10:8]	0 or 1	Set carry in to 0 or 1
		CIN	Select cin as source. This option adds a CIN port on the Opmode block.

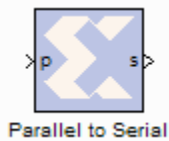
Appendix: DSP48E Control Instruction Format

Instruction Field Name	Location	Mnemonic	Description
YX Mux	op[3:0]	0	0
		P	DSP48 output register
		A:B	Concat inputs A and B (A is MSB)
		A*B	Multiplication of inputs A and B
		C	DSP48 input C

Z Mux	op[6:4]	P+C	DSP48 input C plus P
		A:B+C	Concat inputs A and B plus C register
		0	0
		PCIN	DSP48 cascaded input from PCOUT
		P	DSP48 output register
		C	DSP48 C input
		PCIN>>17	Cascaded input downshifted by 17
		P>>17	DSP48 output register downshifted by 17
Alumode	op[10:7]	X + Z	Addition
		Z – X	Subtraction
		X logic Z	Logic instruction
Carry In Select	op[14:12]	0 or 1	Set carry in to 0 or 1
		CIN	Select cin as source. This adds a CIN port to the Opmode block whose value is inserted into the mnemonic at bit location 11.

Parallel to Serial

This block is listed in the following Xilinx Blockset libraries: *Basic Elements, Data Types, and Index.*



The Parallel to Serial block takes an input word and splits it into N time-multiplexed output words where N is the ratio of number of input bits to output bits. The order of the output can be either least significant bit first or most significant bit first.

The following waveform illustrates the block's behavior:

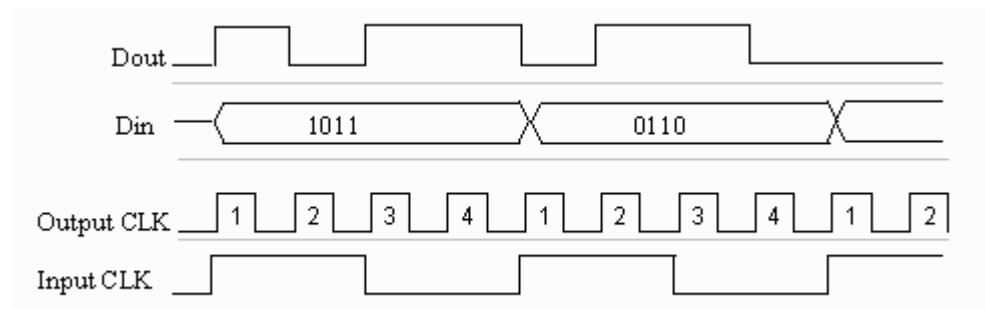


Figure 1: Example of Parallel to Serial behavior

This example illustrates the case where the input width is 4, output word size is 1, and the block is configured to output the most significant word first.

Block Interface

The Parallel to Serial block has one input and one output port. The input port can be any size. The output port size is indicated on the block parameters dialog box.

Block Parameters Dialog Box

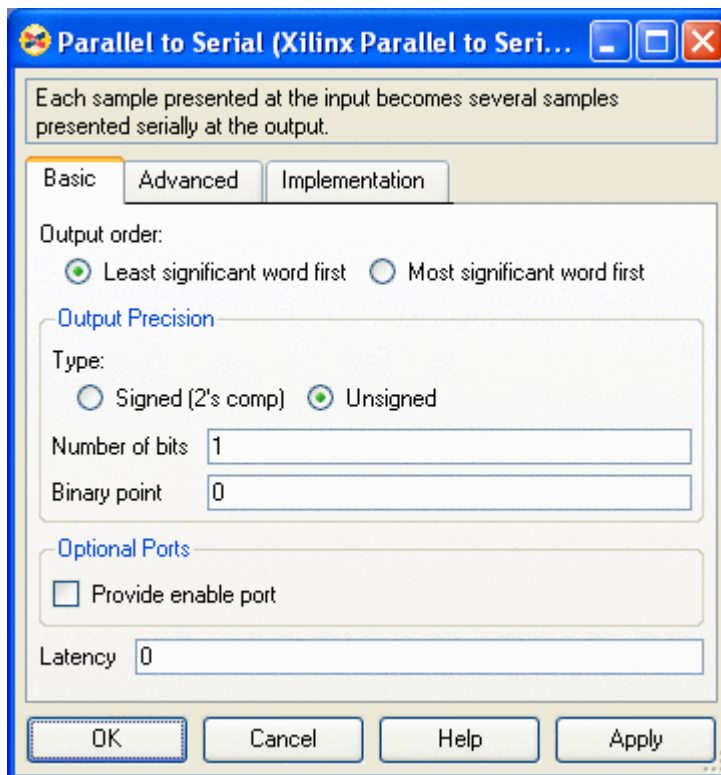


Figure 2: Parallel to Serial block parameters dialog box

Parameters specific to the block are:

- **Output order:** Most significant word first or least significant word first.
- **Type:** signed or unsigned.
- **Number of bits:** Output width. Must divide Number of Input Bits evenly.
- **Binary Point:** Binary point location.

The minimum latency of this block is 1.

Other parameters used by this block are explained in the Common Parameters section at the beginning of this chapter.

Pause Simulation

This block is listed in the following Xilinx Blockset libraries: Tools and Index.



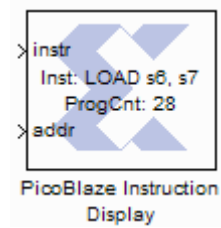
The Xilinx Pause Simulation block pauses the simulation when the input is non-zero. The block accepts any Xilinx signal type as input.

When the simulation is paused, it can be restarted by selecting the play button on the model toolbar. 

There are no parameters for this block.

PicoBlaze Instruction Display

This block is listed in the following Xilinx Blockset libraries: *Tools and Index*.



The PicoBlaze Instruction Display block takes an encoded 18 bit PicoBlaze instruction and a 10 bit address and displays the decoded instruction and the program counter on the block icon. This feature is useful when debugging PicoBlaze designs and can be used in conjunction with the Single-Step Simulation block to step through each instruction.

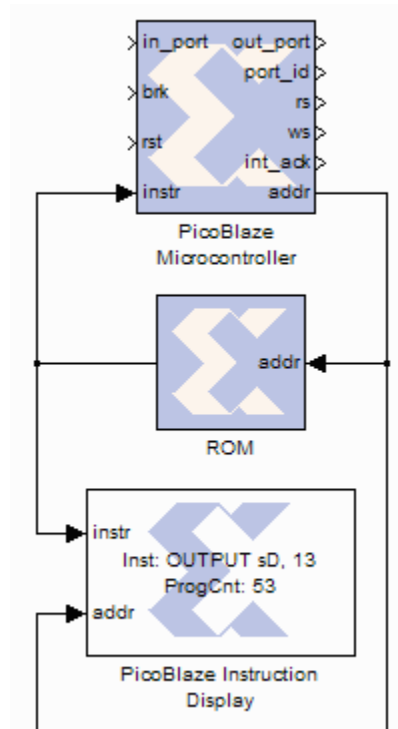


Figure 1: Using the Instruction Display

Block Interface

The PicoBlaze Instruction Display block has two input ports. The `instr` port accepts an 18 bit encoded instruction. The `addr` port accepts a 10 bit address value which is the program counter.

Block Parameters Dialog Box

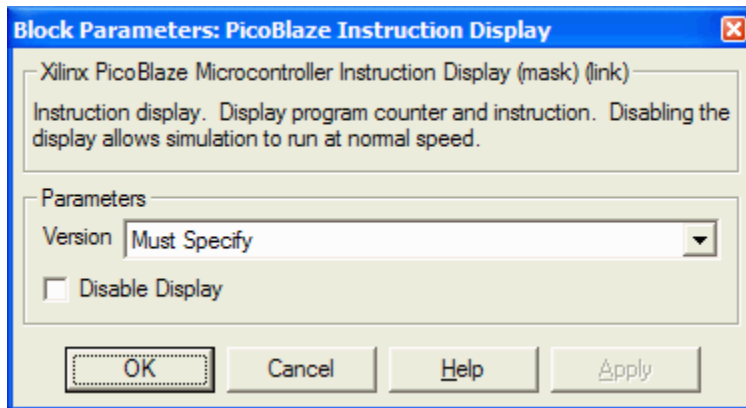


Figure 2: PicoBlaze Instruction Display block parameters dialog box

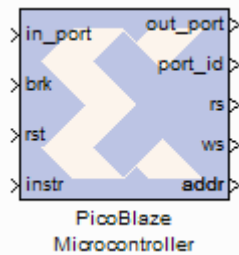
Parameters specific to the block are:

- **Version:** PicoBlaze 2 or PicoBlaze 3.
- **Disable Display:** When selected, the display is no longer updated which will speed up your simulation when not in debug mode.

The PicoBlaze Instruction Display block does not use a Xilinx LogiCORE.

PicoBlaze Microcontroller

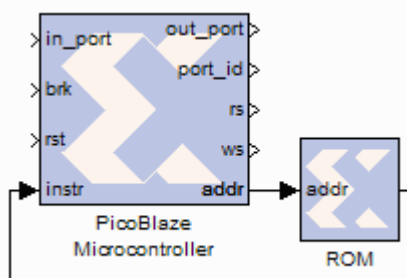
This block is listed in the following Xilinx Blockset libraries: Control Logic and Index.



The Xilinx PicoBlaze Microcontroller block implements an embedded 8-bit microcontroller using the PicoBlaze macro. The block provides access to two versions of PicoBlaze. PicoBlaze 2 supports Virtex-II and PicoBlaze 3 supports Virtex-II, Virtex-IIPro, Spartan-3, and Virtex-4. The PicoBlaze 2 macro provides 49 instructions, 32 8-bit general-purpose registers, 256 directly and indirectly addressable ports, and a maskable interrupt. By comparison, the PicoBlaze 3 provides 53 instructions, 16 8-bit general-purpose registers, 256 directly and indirectly addressable ports, and a maskable interrupt, as well as 64

bytes of internal scratchpad memory accessible using the STORE and FETCH instructions. The PicoBlaze 2 embedded controller and its instruction set are described in detail in the Xilinx Application Note XAPP627, which can be found at <http://www.xilinx.com/bvdocs/appnotes/xapp627.pdf>.

Ordinarily, a single block ROM containing 1024 or fewer 8 bit words serves as the program store. The microcontroller and ROM are connected as shown below.



Block Interface

Both versions of the block have four input ports. The 8-bit data port, `in_port`, is read during an INPUT operation. The value can be transferred into any of the 32 registers. The program can be interrupted by setting the port `brk` to 1. The processor can be reset by setting `rst` to 1. This clears registers and forces the processor to begin executing instructions at address 0. The 8-bit input port `instr` accepts PicoBlaze instructions.

The PicoBlaze 2 block has five output ports. The PicoBlaze 3 block has six output ports. The 8-bit output port `out_port` is written during an OUTPUT instruction. During a read/write, the `port_id` output identifies the location from which a value is read/written. The output ports `rs` (read strobe) and `ws` (write strobe) indicate whether a read (INPUT) or write (OUTPUT) operation is occurring. `Addr` is the address of the next instruction to be executed by the processor. The processor has no internal program store. The output port `addr` specifies the next location from which an instruction should be executed. The `int_ack` port (PicoBlaze 3 only) indicates when the interrupt service routine is started (i.e. the program counter is set to 0x3FF).

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.



Figure 1: PicoBlaze Microcontroller block parameters dialog box

Parameters specific to the PicoBlaze Microcontroller block are:

- **Version:** PicoBlaze 2 or PicoBlaze 3.
- **Display Internal State:** When checked, the registers and control flags are made available in the MATLAB workspace. The information is present as a structure with the following naming convention:
`< design name >_< subsystem name >_< PicoBlaze block name >.`

The structure contains a field for each register (i.e. `s00`, `s01`, etc.) and the control flags `CARRY` and `ZERO`.

- **Display Values As:** Tells the radix to use for displaying values.

Other parameters used by this block are explained in the Common Parameters section at the beginning of this chapter of the Reference Guide.

How to use the PicoBlaze Assembler

1. Write a PicoBlaze program. Save the program with a `.psm` file extension.
2. Run the assembler from the Matlab command prompt. The command is `xlpb_as -p <your_psm_file>`. The default is to assemble a program for PicoBlaze 3. To assemble a program for PicoBlaze 2 use the `-v 2` option. This script runs the PicoBlaze assembler and generates a m-code program which should be used to populate the ROM or RAM used as the program store.

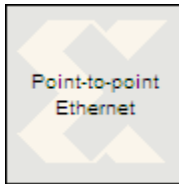
Known Issues

- It is not possible to synthesize a design containing a PicoBlaze block using Synplify or Synplify Pro.
- The PicoBlaze assembler `xlpb_as` fails when the assembly code file is found in a directory whose full path name contains more than 58 characters.
- Verilog netlisting is not supported for this block. Refer to the section on Generating Verilog When the Design Contains VHDL-Only Blocks.

PicoBlaze Microprocessor online documentation

More information can be found at
http://www.xilinx.com/products/design_resources/proc_central/index.htm.

Point-to-point Ethernet Co-Simulation



The Xilinx Point-to-point Ethernet Co-Simulation block provides an interface to perform hardware co-simulation through a raw Ethernet connection. Refer to Point-to-point Ethernet Hardware Co-Simulation for further details about the interface, its prerequisites and setup procedures.

A new Point-to-point Ethernet co-simulation block is created by selecting “Point-to-point Ethernet Cosim” as the compilation target in a System Generator block. The resulting block will have ports corresponding to the original gateways (or subsystem ports). The generated block can then be used just like any other Sysgen block. The co-simulation block interacts with the FPGA hardware platform during a Simulink simulation. Simulation data written to the input ports of the block passes to the hardware via the block. Conversely, when data is read from the co-simulation block's output ports, the block reads the appropriate values from the hardware and drives them on the output ports so they can be interpreted in Simulink. In addition, the block automatically opens, configures, steps, and closes the platform.

Block Parameters Dialog Box

The dialog box tabs specific to the point-to-point Ethernet hardware co-simulation block are shown below.

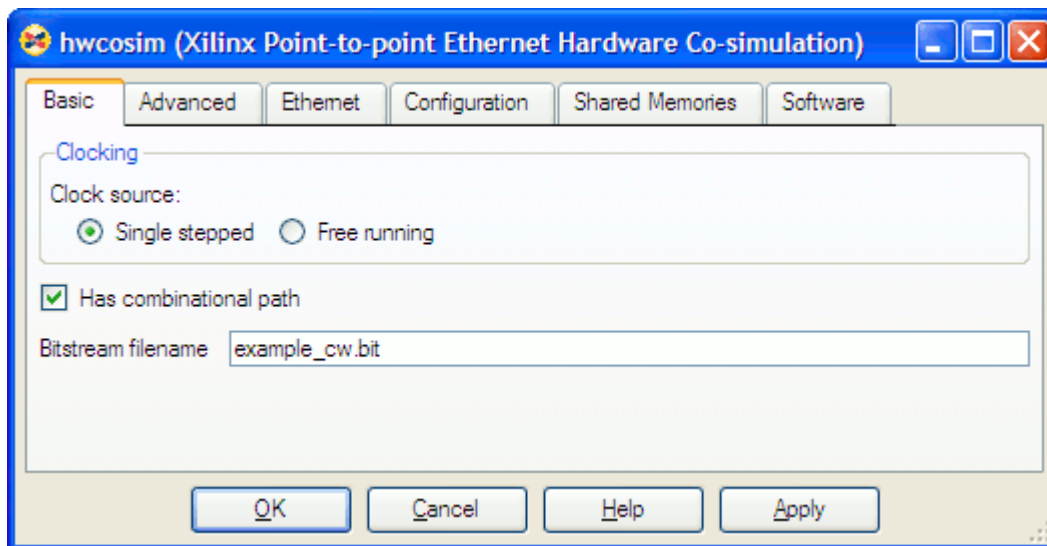


Figure 1: Point-to-point Ethernet Co-simulation Block Basic Tab

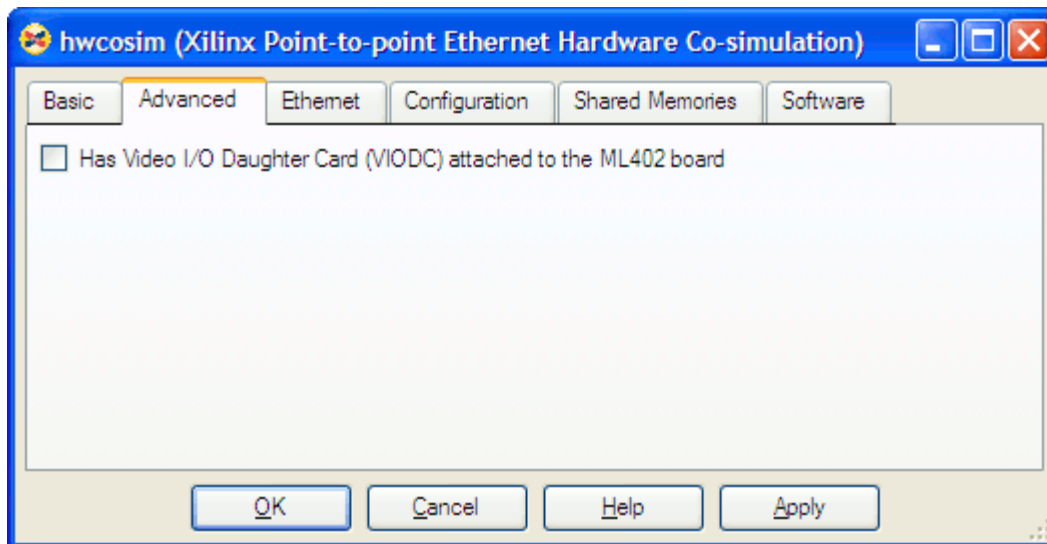


Figure 2: Point-to-point Ethernet Co-simulation Block Advanced Tab

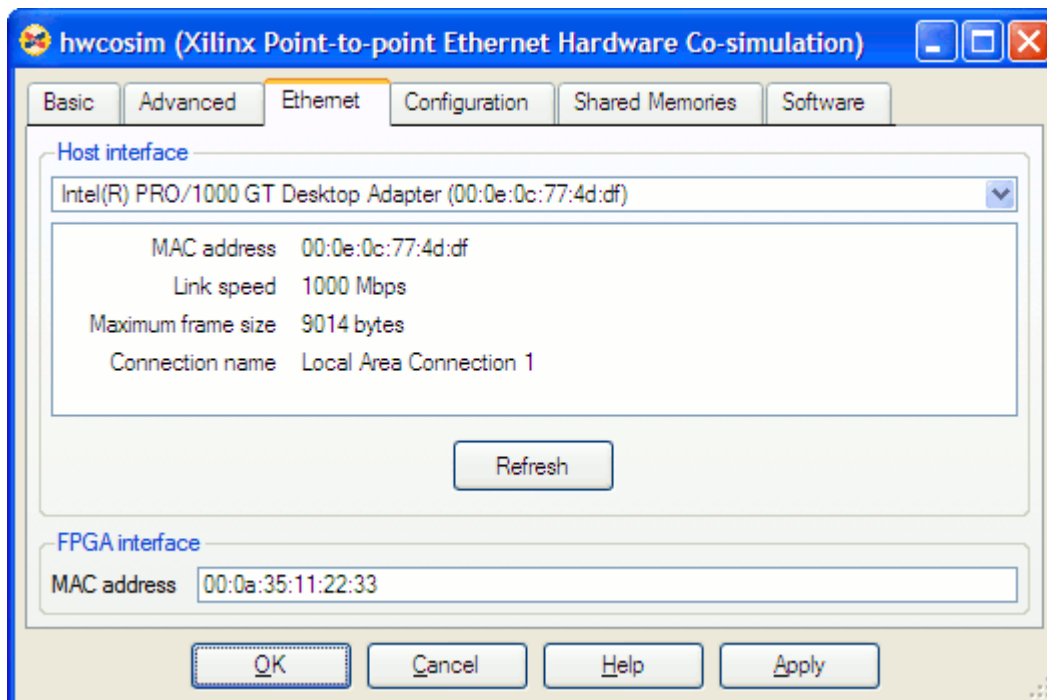


Figure 3: Point-to-point Ethernet Co-simulation Block Ethernet Tab

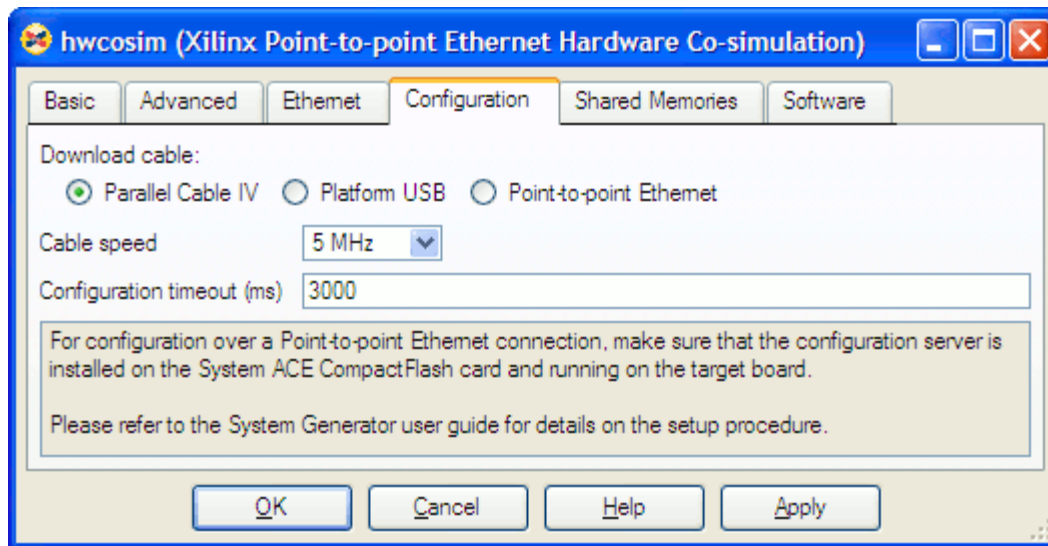


Figure 4: Point-to-point Ethernet Co-simulation Block Configuration Tab

Parameters specific to the block are:

- **Clock source (Basic):** You may select between **Single stepped** and **Free running** clock sources. Selecting a **Single stepped** clock allows the block to step the board one clock cycle at a time. Each clock cycle step corresponds to some duration of time in Simulink. Using this clock source ensures the behavior of the co-simulation hardware during simulation will be bit and cycle accurate when compared to the simulation behavior of the subsystem from which it originated. Sometimes single stepping is not necessary and the board can be run with a **Free Running** clock. In this case, the board will operate asynchronously to the Simulink simulation.
- **Has Combination Path (Basic):** Sometimes it is necessary to have a direct combinational feedback path from an output port on a hardware co-simulation block to an input port on the same block (e.g., a wire connecting an output port to an input port on a given block). If you require a direct feedback path from an output to input port, and your design does not include a combinational path from any input port to any output port, unchecking this box will allow the feedback path in the design.
- **Bitstream filename (Basic):** Specifies the co-simulation FPGA configuration file for the Point-to-point Ethernet hardware co-simulation platform. When a new co-simulation block is created during compilation, this parameter is automatically set so that it points to the appropriate configuration file. You need only adjust this parameter if the location of the configuration file changes.
- **Has Video I/O Daughter Card (VIODC) attached to the ML402 board (Advanced):** This option must be checked, if a Video I/O daughter card is attached on the target ML402 board, to allow proper device configuration. It should be otherwise unchecked.
- **Host interface (Ethernet):** Specifies the host network interface card that is used to establish a connection to the target FPGA platform for co-simulation. The pop-down list shows all active network interface cards that can be used for point-to-point Ethernet co-simulation. The information panel displays the MAC address, link speed, maximum frame size of the chosen interface, and its corresponding connection name in the Windows environment. The list of available interfaces and the information panel may not reflect the up-to-date status, and in such case, they can be updated by clicking Refresh button.
- **FPGA interface MAC address (Ethernet):** Specify the Ethernet MAC address associated with the target FPGA platform. The MAC address must be specified using six pairs of two-digit hexadecimal number separated by colons (e.g. 00:0a:35:11:22:33). For JTAG-based configuration, the MAC address can be arbitrarily assigned to each FPGA platform provided there is no conflicting address in the Ethernet LAN. For configuration over the point-to-point Ethernet connection, refer to Configuration using System ACE for details on configuring the MAC address of the FPGA platform.

- **Download cable (Configuration):** You may select between Parallel Cable IV and Platform USB programming cables for JTAG-based device configuration, or alternatively, select Point-to-point Ethernet to perform device configuration over the Ethernet connection. Refer to Configuration using System ACE for further details on the setup procedure.
- **Cable speed (Configuration):** For JTAG-based configuration only. Sometimes you may need to run the programming cable at a frequency less than the default (maximum) speed setting for hardware co-simulation. This menu allows you to choose a cable speed that is suitable for your hardware setup. Normally the default speed will suffice, however, it is recommended to try a slower cable speed if System Generator fails to configure the device for co-simulation.
- **Configuration timeout (Configuration):** Specifies the timeout value, in milliseconds, for the configuration process. The default value should suffice in the general case, but be advised that a larger value may be needed if it takes a considerable amount of time to re-establish the connection after device configuration.

See Also

Ethernet Hardware Co-Simulation, Point-to-point Ethernet Hardware Co-Simulation, Configuration using System ACE

Puncture

This block is listed in the following Xilinx Blockset libraries: *Communication and Index*.

The Xilinx Puncture block removes a set of user-specified bits from the input words of its data stream. Based on the puncture code parameter, a binary vector that specifies which bits to remove, it converts input data of type $UFixN_0$ (where N is equal to the length of the puncture code) into output data of type $UFixK_0$ (where K is equal to the number of ones in the puncture code). The output rate is identical to the input rate.

This block is commonly used in conjunction with a convolutional encoder to implement punctured convolutional codes as shown in the figure below.

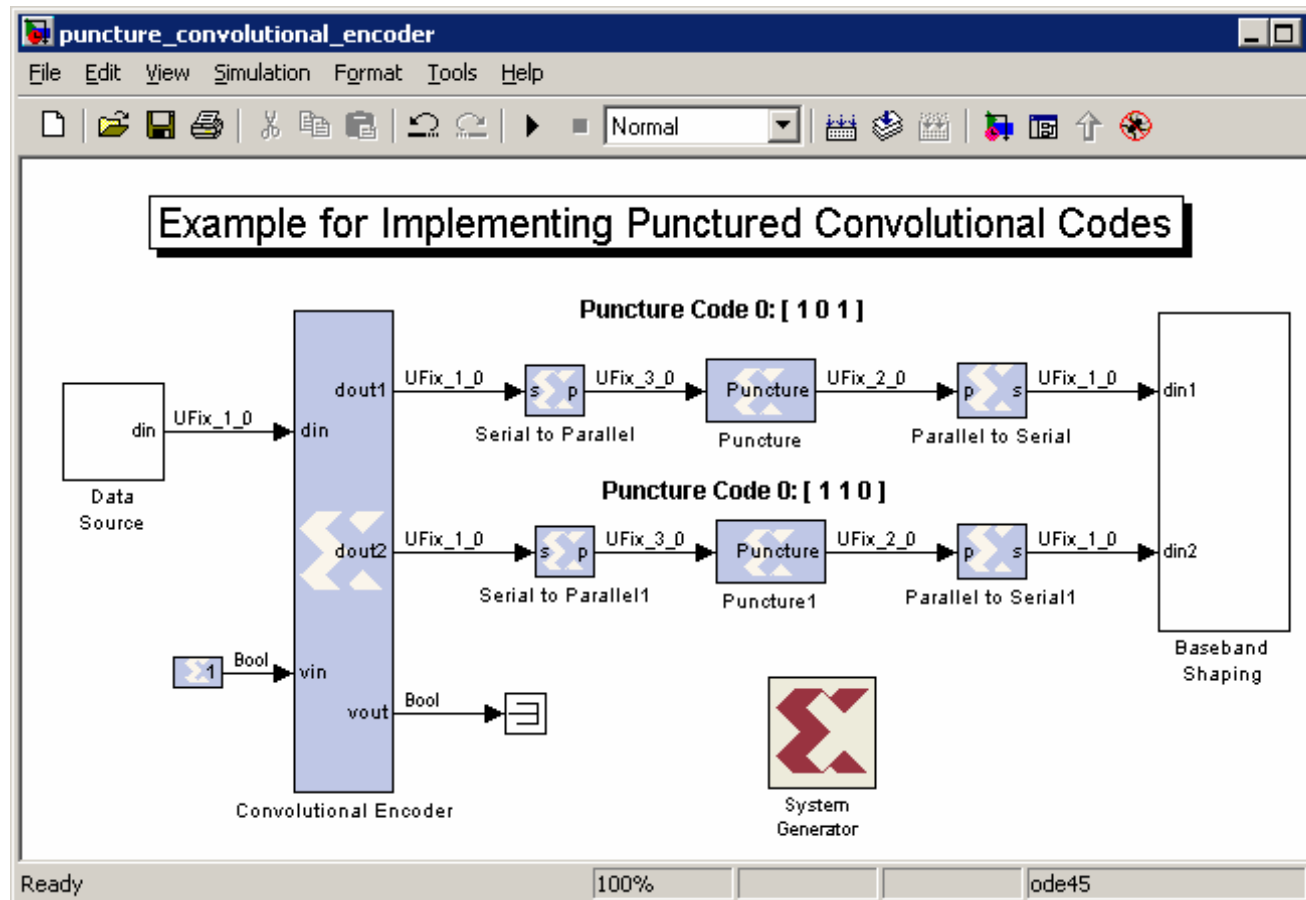


Figure 1: Example of a Puncture block application

The system shown implements a rate $\frac{1}{2}$ convolutional encoder whose outputs are punctured to produce four output bits for each three input bits. The top puncture block removes the center bit for code 0 ([1 0 1]) and bottom puncture block removes the least significant bit for code 1 ([1 1 0]), producing a 2-bit punctured output. These data streams are serialized into 1-bit in-phase and quadrature data streams for baseband shaping.

Block Parameters Dialog Box

The Xilinx puncture block can be configured using its Block Parameters dialog box.

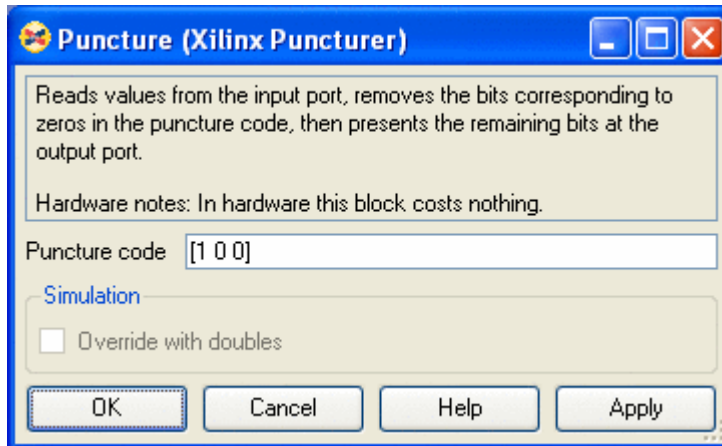


Figure 2: Puncture block parameters dialog box

Parameters specific to the Xilinx Puncture block are:

- **Puncture Code:** the puncture pattern represented as a bit vector, where a zero in position i indicates bit i is to be removed.

Other parameters used by this block are described in the Common Parameters section at the beginning of this chapter.

Register

This block is listed in the following Xilinx Blockset libraries: *Basic Elements, Control Logic, Memory, and Index.*



The Xilinx Register block models a D flip flop-based register, having latency of one sample period.

Block Interface

The block has one input port for the data and an optional input reset port. The initial output value is specified by the user in the block parameters dialog box (below). Data presented at the input will appear at the output after one sample period. Upon reset, the register assumes the initial value specified in the parameters dialog box.

The Register block differs from the Xilinx Delay block by providing an optional reset port and a user specifiable initial value.

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

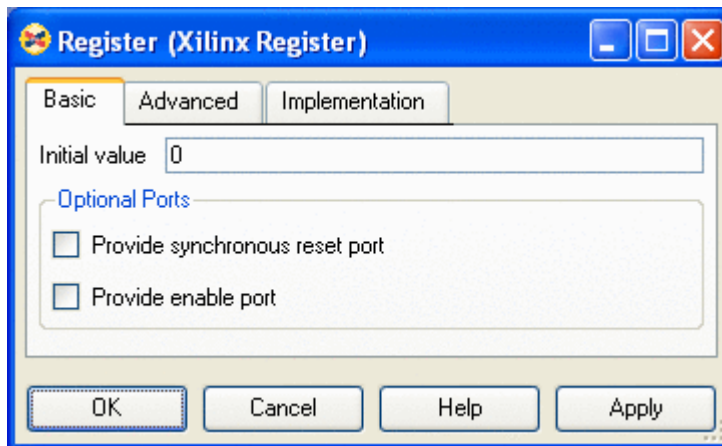


Figure 1: Register block Basic Tab

Parameters specific to the block are:

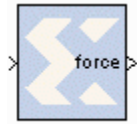
- **Initial value Basic):** specifies the initial value in the register.

Other parameters used by this block are explained in the Common Parameters section at the beginning of this chapter.

The Register block is implemented as a synthesizable VHDL module. It does not use a Xilinx LogiCORE.

Reinterpret

This block is listed in the following Xilinx Blockset libraries: Basic Elements, Math, and Index.



Reinterpret

The Reinterpret block forces its output to a new type without any regard for retaining the numerical value represented by the input. The binary representation is passed through unchanged, so in hardware this block consumes no resources. The number of bits in the output will always be the same as the number of bits in the input.

The block allows for unsigned data to be reinterpreted as signed data, or, conversely, for signed data to be reinterpreted as unsigned. It also allows for the reinterpretation of the data's scaling, through the repositioning of the binary point within the data. The Xilinx Scale block provides an analogous capability.

An example of this block's use is as follows: if the input type is 6 bits wide and signed, with 2 fractional bits and the output type is forced to be unsigned with 0 fractional bits, then an input of -2.0 (1110.00 in binary, two's complement) would be translated into an output of 56 (111000 in binary).

This block can be particularly useful in applications that combine it with the Xilinx Slice block or the Xilinx Concat block. To illustrate the block's use, consider the following scenario:

Given two signals, one carrying signed data and the other carrying two unsigned bits (a `UFix_2_0`), we want to design a system that concatenates the two bits from the second signal onto the tail (least significant bits) of the signed signal.

We can do so using two Reinterpret blocks and one Concat block. The first Reinterpret block is used to force the signed input signal to be treated as an unsigned value with its binary point at zero. The result is then fed through the Concat block along with the other signal's `UFix_2_0`. The Concat operation is then followed by a second Reinterpret that forces the output of the Concat block back into a signed interpretation with the binary point appropriately repositioned.

Though three blocks are required in this construction, the hardware implementation will be realized as simply a bus concatenation, which has no cost in hardware.

Block Parameters Dialog box

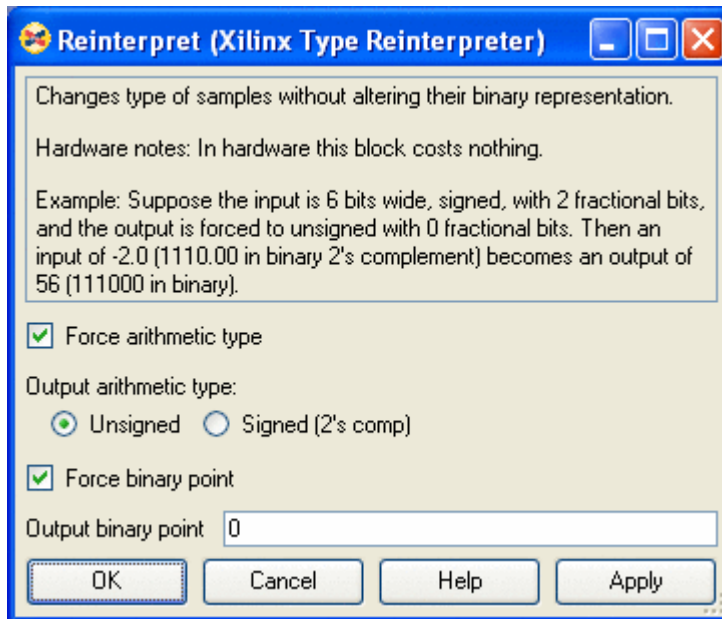


Figure 1: Reinterpret block parameters dialog box

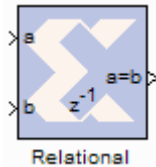
Parameters specific to the block are:

- **Force arithmetic type:** When checked, the Output Arithmetic Type parameter can be set and the output type will be forced to the arithmetic type chosen according to the setting of the Output Arithmetic Type parameter. When unchecked, the arithmetic type of the output will be unchanged from the arithmetic type of the input.
- **Output arithmetic type:** The arithmetic type (unsigned or signed, 2's complement) to which the output is to be forced.
- **Force binary point:** When checked, the Output Binary Point parameter can be set and the binary point position of the output will be forced to the position supplied in the Output Binary Point parameter. When unchecked, the arithmetic type of the output will be unchanged from the arithmetic type of the input.
- **Output binary point:** The position to which the output's binary point is to be forced. The supplied value must be an integer between zero and the number of bits in the input (inclusive).

This block does not use any hardware resources.

Relational

This block is listed in the following Xilinx Blockset libraries: *Basic Elements*, *Control Logic*, *Math*, and *Index*.



The Xilinx Relational block implements a comparator. The supported comparisons are the following:

- equal-to ($a = b$)
- not-equal-to ($a \neq b$)
- less-than ($a < b$)
- greater-than ($a > b$)
- less-than-or-equal-to ($a \leq b$)
- greater-than-or-equal-to ($a \geq b$)

The output of the block is a Bool.

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

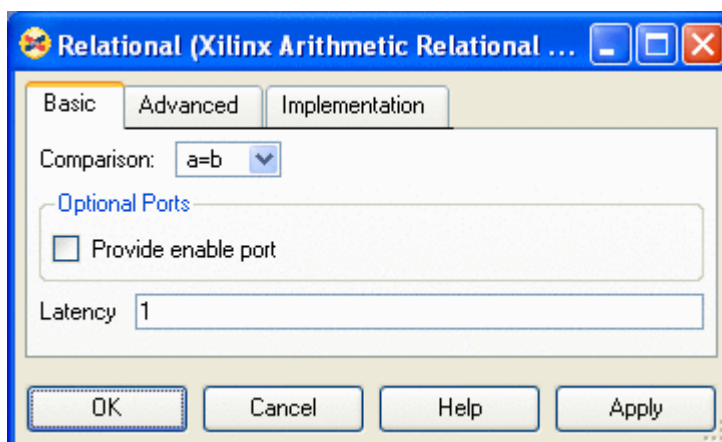


Figure 1: Relational block parameters dialog box

The only parameter specific to the Relational block is:

- **Comparison:** specifies the comparison operation computed by the block.

Other parameters used by this block are explained in the Common Parameters section at the beginning of this chapter.

Xilinx LogiCORE

The Relational block does not use a Xilinx LogiCORE.

Resource Estimator

This block is listed in the following Xilinx Blockset libraries: *Tools and Index*.



The Xilinx Resource Estimator block provides fast estimates of FPGA resources required to implement a System Generator subsystem or model. These estimates are computed by invoking block-specific estimators for Xilinx blocks, and summing these values to obtain aggregated estimates of lookup tables (LUTs), flip-flops (FFs), block memories (BRAM), 18x18 multipliers, tristate buffers, and I/Os.

Every Xilinx block that requires FPGA resources has a mask parameter that stores a vector containing its resource requirements. The Resource Estimator block can invoke underlying functions to populate these vectors (e.g. after parameters or data types have been changed), or aggregate previously computed values that have been stored in the vectors. Each block has a checkbox control **Define FPGA area for resource estimation** that short-circuits invocation of the estimator function and uses the estimates stored in the vector instead.

An estimator block can be placed in any subsystem of a model. When another estimator block is situated in the sub-hierarchy below an estimator, the blocks interact as described below.

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

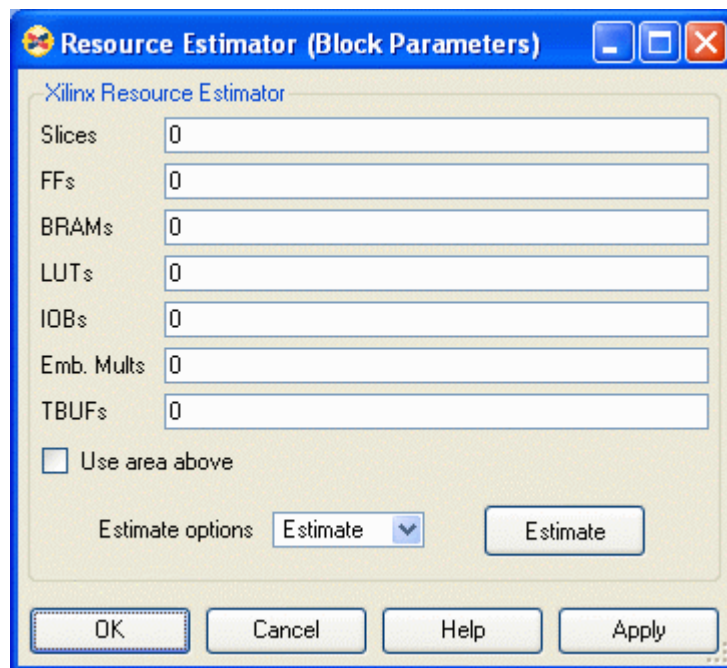


Figure 1: Resource Estimator block parameters dialog box

Parameters in the Resource Estimator dialog box are:

- **Slices:** Slices utilized by the block. (A slice normally consists of two flip-flops, two LUTs and associated mux, carry and control logic.)
- **FFs:** Flip Flops utilized by the block.
- **BRAMs:** Block RAMs utilized by the block.
- **LUTs:** Look-up Tables utilized by the block.
- **IOBs:** Input/Output blocks consumed by the block.

- **Embedded Mults:** Embedded multipliers utilized by the block. (For example, the Virtex-II device contains embedded 18X18 multipliers.)
- **TBUFs:** Tristate Buffers utilized by the block.
- **Use Area Above:** When this box is checked, any resource estimation performed on this subsystem will return the numbers entered in the edit boxes of the dialog box (The data represented by these fields is equivalent to the FPGA Area field in the individual System Generator blocks). Any blocks at the level of the subsystem where this block resides, or below, will have no automatic resource estimation performed when this box is checked.
- **Estimate Options:** Allows selection of estimation method as one of the following: Estimate, Quick, Post Map and Read Mrp. These options are explained in greater detail in the next section.
- **Estimate:** Launches resource estimation

Perform Resource Estimation buttons

The FPGA Area fields described above can either be manually entered or filled in by launching resource estimation with **Estimate Options** set to one of the following:

- **Estimate:** Invokes block estimation functions top-down for each block and subsystem recursively. Blocks that do not have an estimation function but can be implemented in hardware (except shared memory blocks) are automatically estimated using post-map area. If any block has the **Define FPGA area for resource estimation** option selected, its estimation function is short-circuited and its current estimate is used. If **Use area above** option is selected for a Resource Estimator block, this block's estimate will be used for the entire subsystem containing it, and no other block estimation functions will be invoked for that portion of the model hierarchy.
- **Quick:** Causes the Estimate button to sum all of the FPGA Area fields on the blocks and subsystems at or below the current subsystem. No underlying estimation functions are invoked.

- **Post-Map Area:** Causes the Estimate button to automatically invoke Xilinx map tool on the entire subsystem and read back the results from the created Map Report File (MRP). In order to use this option a System Generator block along with the resource estimator block must be instantiated in the subsystem being estimated.
- **Read MRP:** Causes the Estimate button to open a file browser as shown in **Figure 2** and results from a selected MRP file are read into the Resource Estimator. This method of obtaining resource information is available for subsystems that have been previously synthesized, translated and mapped. This can be useful for complex Xilinx blocks that have no estimation function and will no longer change in a design.

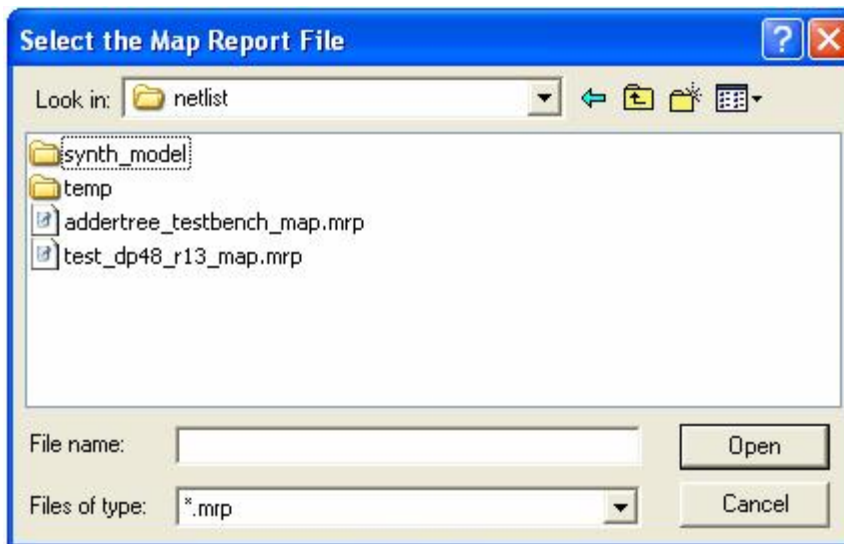


Figure 2 Read MRP file browser

The numbers from the map report file and those inserted into the Resource Estimator dialog box area fields may be slightly different (this applies to Post Map Area option also). Any IOB FF resources found in the MRP file will be added into the estimators FFs field. Along the same lines, half of the MRP's IOB FF resources will be added into the estimators Slices field and the estimators IOBs field will always be set to 0 after performing a Post-Map Area MRP or Read MRP. Since the usefulness of this feature generally occurs in estimating subsystems, IOB resources must be included in the CLB utilization numbers to prevent incorrectly reporting IOB resources not used in the final design.

Blocks Supported by Resource Estimation

Blocks that have fast resource estimation functions:

Accumulator, Addressable Shift Register, AddSub, CMult (sequential version not supported), Convert, Counter, Delay, Down Sample, Dual Port RAM, FIFO, FFT, FFTx, Gateway In, Gateway Out, Inverter, LFSR, Logical, Mult (sequential version not supported), Mux (tristate version not supported), Negate, Parallel to Serial, PicoBlaze Processor, Register, Relational, ROM, Serial To Parallel, Shift, Sine Cosine, Single Port RAM, Threshold, Up Sample.

Blocks that use Post Map Area Estimates:

System generator blocks that do not have fast resource estimation functions and use hardware are estimated using post-map area. In order to avoid using this method enter in a constant or a user-created estimation function into the FPGA Area field of the block and click on the Define FPGA area for resource estimation checkbox.

Blocks that do not use any hardware:

System Generator, Clear Quantization Error, Clock Enable Probe, Clock Probe, Concat, Constant, Discard Subsystem, FDATool, Indeterminate Probe, ModelSim, Pause Simulation, PicoBlaze Instruction Display, Quantization Error, Reinterpret, Sample Time, Scale, Simulation Multiplexer, Single-Step Simulation, Slice, BitBasher.

Blocks with special handling:

Discard Subsystem (Resource Estimator will ignore any resources in a subsystem containing this block).

Shared memory blocks are not estimated. In designs containing Shared Memory blocks, use the Multiple System Generator block to generate the HDL netlist files. Use ISE tools to create the Map Report File for the design and use the Read MRP option to obtain the results contained in the MRP file produced.

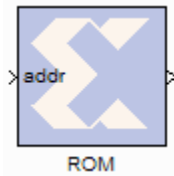
Known Issues for Resource Estimation

Resource estimation in System Generator has the following known issues:

- Estimations are based upon the data types for the inputs and outputs of each block that Simulink calculates during the compilation phase. If significant trimming takes place in a design that is not seen at the block level, the resource estimation tool will overestimate those trimmed resources.
- Any logic that the synthesis tools can combine across blocks will be overestimated. For example, when using blocks that have no latency, there is a good chance combinational logic will be optimized across block boundaries.
- Multirate designs contain clock enable generation logic that is underestimated. System Generator handles multirate designs by using one clock and generating a different clock enable for each rate. In order to accurately predict the amount of logic in the clock enable drivers, the estimator would need to look at the system as a whole instead of at the block level. Note, this underestimation will also include resources associated with additional clock enable connections that will be made to each of the blocks that were not visible to the block estimation functions.
- Shared Memory Blocks are not estimated. In designs containing Shared Memory blocks, the estimates reported do not include the resources used by the Shared Memory blocks.

ROM

This block is listed in the following Xilinx Blockset libraries: *Control Logic, Memory, and Index.*



The Xilinx ROM block is a single port read-only memory (ROM).

Values are stored by word and all words have the same arithmetic type, width, and binary point position. Each word is associated with exactly one address. An address can be any unsigned fixed point integer from 0 to $d-1$, where d denotes the ROM depth (number of words). The memory contents are specified through a block parameter. The block has one input port for the memory address and one output port for data out. The address port must be an unsigned fixed point integer. The block has two possible Xilinx LogiCORE implementations, using either distributed or block memory.

When implementing ROM blocks on Virtex-4 devices, maximum timing performance is possible if the following conditions are satisfied:

- Provide Reset Port for Output Register is unchecked.
- Depth is less than 16,384.
- Latency is set to 2 or higher.

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

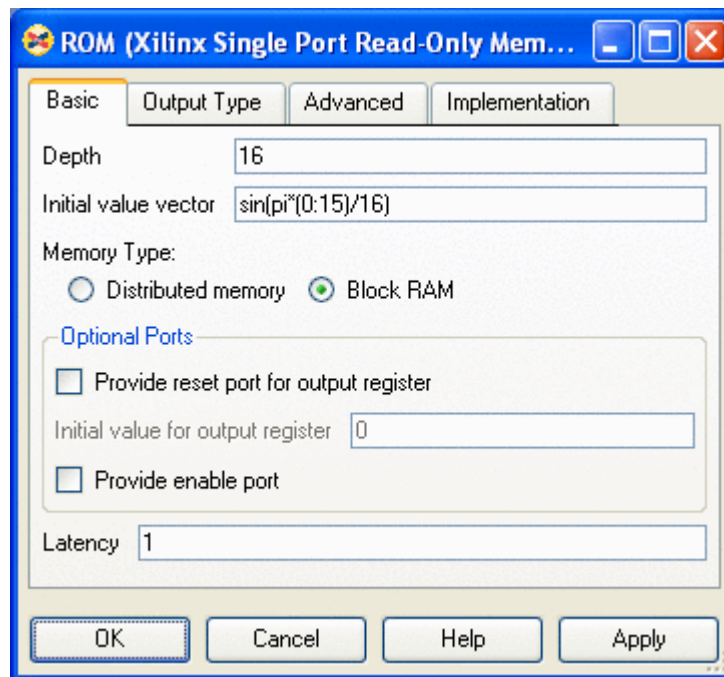


Figure 1: ROM block Basic tab

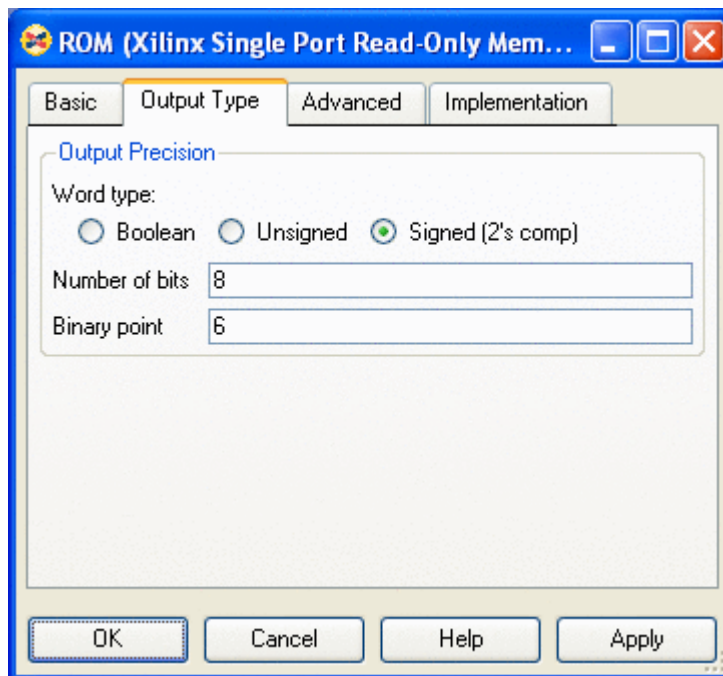


Figure 2: ROM block Output Type tab

Parameters specific to this block are:

- **Depth:** specifies the number of words stored; must be a positive integer.
- **Initial value vector:** specifies the initial value. When the vector is longer than the ROM depth, the vector's trailing elements are discarded. When the ROM is deeper than the vector length, the ROM's trailing words are set to zero. The initial value vector is saturated or rounded according to the data precision specified for the ROM.
- **Memory Type:** specifies block implementation to be distributed RAM or Block RAM.
- **Provide reset port for output register:** when selected, allows access to the reset port available on the output register of the Block ROM. The reset port is available only when the latency of the Block ROM is set to 1.
- **Initial value for output register:** specifies the initial value for output register. The initial value is saturated and rounded according to the data precision specified for the ROM. The option to set initial value is available only for Virtex-II, Virtex-II Pro, Spartan-3, and Virtex-4 devices.
- **Word type:** specifies the data to be Signed or Unsigned.
- **Number of bits:** specifies the number of bits in a memory word.
- **Binary point:** specifies the location of the binary point in the memory word.

Other parameters used by this block are explained in the Common Parameters section at the beginning of this chapter.

Xilinx LogiCORE

The block always uses a Xilinx LogiCORE: Single Port Block Memory or Distributed Memory.

For the block memory, the address width must be equal to $\text{ceil}(\log_2(d))$ where d denotes the memory depth. The maximum width of data words in the block memory depends on the depth specified; the maximum depth is depends on the device family targeted. Figure 3 and Figure 4 provide the maximum data word width for a given block memory depth.

Depth	Width
2 to 2048	256

2049 to 4096	192
4097 to 8192	96
8193 to 16K	48
16K+1 to 32K	24
32K+1 to 64K	12
64K+1 to 128K	6
128K+1 to 256K	3

Figure 2: Maximum Width for Various Depth Ranges (Virtex/Virtex-E/Spartan-3)

Depth	Width
2 to 8192	256
8193 to 16K	192
16K+1 to 32K	96
32K+1 to 64K	48
64K+1 to 128K	24
128K+1 to 256K	12
256K+1 to 512K	6
512K+1 to 1024K	3

Figure 3: Width for Various Depth Ranges (Virtex-II/Virtex-II Pro/Virtex-4)

When the distributed memory parameter is selected, LogiCORE Distributed Memory is used. The depth must be between 16 and 65536, inclusive for Virtex-II, Virtex-II Pro, Spartan-3, and Virtex-4; depth must be between 16 to 4096, inclusive for the other FPGA families. The word width must be between 1 and 1024, inclusive.

RS Decoder v5_1

This block is listed in the following Xilinx Blockset libraries: *Communication and Index*.



Reed-Solomon (RS) codes are block-based error correcting codes with a wide range of applications in digital communications and storage. They are used to correct errors in many systems such as digital storage devices, wireless/ mobile communications, and digital video broadcasting.

The Reed-Solomon decoder processes blocks generated by a Reed-Solomon encoder, attempting to correct errors and recover information symbols. The number and type of errors that can be corrected depend on the characteristics of the code.

Reed-Solomon codes are Bose-Chaudhuri-Hocquenghem (BCH) codes, which in turn are linear block codes. An (n,k) linear block code is a k -dimensional subspace of an n -dimensional vector space over a finite field. Elements of the field are called *symbols*. For a Reed-Solomon code, n ordinarily is $[equation]$, where s is the width in bits of each symbol. When the code is *shortened*, n is smaller. The decoder handles both full length and shortened codes. It is also able to handle *erasures*, that is, symbols that are known with high probability to contain errors.

When the decoder processes a block, there are three possibilities:

1. The information symbols are recovered. This is the case provided $2p+r < n-k$, where p is the number of errors and r is the number of erasures.
2. The decoder reports it is unable to recover the information symbols.
3. The decoder fails to recover the information symbols but does not report an error.

The probability of each possibility depends on the code and the nature of the communications channel. Simulink provides excellent tools for modeling channels and estimating these probabilities.

Block Interface

The Xilinx RS Decoder block has inputs `data_in`, `sync` and `reset` and outputs `data_out`, `blk_strt`, `blk_end`, `err_found`, `err_cnt`, `fail`, `ready` and `rfd`. It also has optional inputs `n_in`, `erase`, `rst`, and `en`, and optional output ports `erase_cnt` and `data_del`.

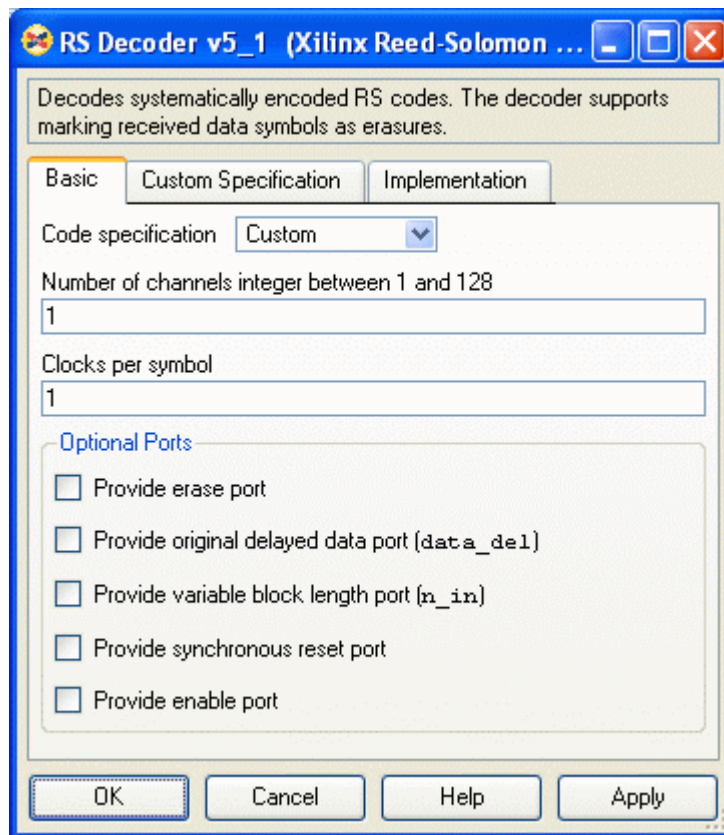
The following describes these ports in detail:

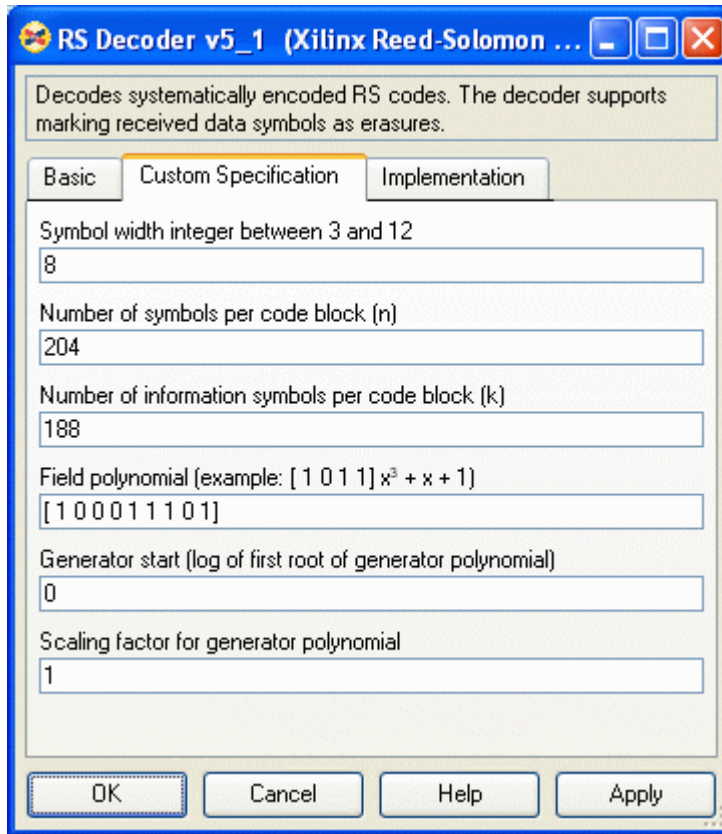
- **data_in**: presents blocks of n symbols to be decoded. The `din` signal must have type `UFIX_s_0`, where s is the width in bits of each symbol.
- **sync**: tells the decoder when to begin processing symbols from `data_in`. The decoder discards input symbols until the first time `sync` is asserted. The symbol on which `sync` is asserted marks the beginning of the first n symbol block to be processed by the decoder. The `sync` signal is ignored till the decoder is ready to accept another code block. The signal driving `sync` must be Bool.
- **reset**: asynchronously resets the decoder. The signal driving `reset` must be Bool. Note: `reset` must be asserted high for atleast 1 sample period before the decoder can start decoding code symbols.
- **erase**: indicates the symbol currently presented on `din` should be treated as an erasure. The signal driving `erase` must be Bool.
- **n_in**: `n_in` is sampled at the start of each block. The new block's length, `n_block`, is set to `n_in` sampled. The `n_in` signal must have type `UFIX_s_0`, where s is the width in bits of each symbol.
- **rst**: carries the reset signal for the decoder. The signal driving `rst` must be Bool.

- **en**: carries the enable signal for the decoder. The signal driving **en** must be Bool.
- **data_out**: produces the information and parity symbols resulting from decoding. The type of **data_out** is the same as that for **data_in**.
- **blk_strt**: presents a 1 at the time **data_out** presents the first symbol of the block. **blk_strt** produces a signal of UFIX_1_0 type.
- **blk_end**: presents a 0 at the time **data_out** presents the last symbol of the block. **blk_end** produces a signal of UFIX_1_0 type.
- **err_found**: presents a value at the time **data_out** presents the last symbol of the block. The value is 1 if the decoder detected any errors or erasures during decoding. **err_found** must have type UFIX_1_0.
- **err_cnt**: presents a value at the time **data_out** presents the last symbol of the block. The value is the number of errors that were corrected. **err_cnt** must have type UFIX_b_0 where b is the number of bits needed to represent n-k.
- **fail**: presents a value at the time **dout** presents the last symbol of the block. The value is 1 if the decoder was unable to recover the information symbols, and 0 otherwise. **fail** must have type UFIX_1_0.
- **ready**: value is 1 when the decoder is ready to sample **data_in** input, and 0 otherwise. **ready** must have type UFIX_1_0.
- **rffd**: value is 1 when the decoder is ready to sample the first symbol of a code block on **data_in** input, and 0 otherwise. **rffd** must have type UFIX_1_0.
- **data_del**: produces the undecoded symbols alongside the decoded symbols on **data_out**. The type of **data_del** is the same as that for **data_in**.
- **erase_cnt**: only available when erasure decoding is enabled. Presents a value at the time **dout** presents the last symbol of the block. The value is the number of erasures that were corrected. **erase_cnt** must have type UFIX_b_0 where b is the number of bits needed to represent n.

Block Parameters Dialog Box

The RS Decoder block can be configured using its Block Parameters dialog box.





Decodes systematically encoded RS codes. The decoder supports marking received data symbols as erasures.

Basic Custom Specification Implementation

Symbol width integer between 3 and 12
8

Number of symbols per code block (n)
204

Number of information symbols per code block (k)
188

Field polynomial (example: [1 0 1 1] $x^3 + x + 1$)
[1 0 0 0 1 1 1 0 1]

Generator start (log of first root of generator polynomial)
0

Scaling factor for generator polynomial
1

OK Cancel Help Apply

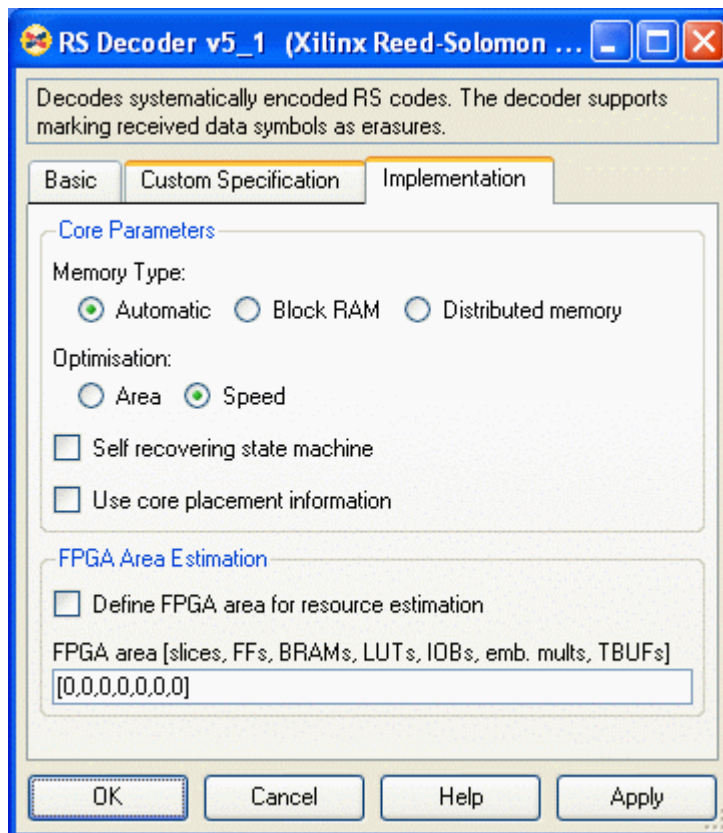


Figure 1: Reed-Solomon Decoder block parameters dialog box

Parameters specific to the RS Decoder block are:

- **Code specification:** specifies the type of RS Decoder desired. The choices are:
 - * **Custom:** allows you to set all the block parameters.
 - * **ATSC:** implements ATSC (Advanced Television Systems Committee) standard (207, 187) shortened RS code.
 - * **CCSDS:** implements CCSDS (Consultative Committee for Space Data Systems) standard (255, 223) full length RS code.
 - * **DVB:** implements DVB (Digital Video Broadcasting) standard (204, 188) shortened RS code.
 - * **IESS-308 (126):** implements IESS-308 (INTELSAT Earth Station Standard) specification (126, 112) shortened RS code.
 - * **IESS-308 (194):** implements IESS-308 specification (194, 178) shortened RS code.
 - * **IESS-308 (208):** implements IESS-308 specification (208, 192) shortened RS code.
 - * **IESS-308 (219):** implements IESS-308 specification (219, 201) shortened RS code.
 - * **IESS-308 (225):** implements IESS-308 specification (225, 205) shortened RS code.
 - * **IEEE-802.16d:** implements IEEE-802.16d specification (255, 239) full length RS code.
- **Number of channels:** tells the number of separate time division multiplexed channels to be processed by the encoder. The encoder supports upto 128 channels.
- **Clocks per symbol:** tells the number of sample periods to use per input data symbol. This may be increased to reduce the processing delay and support continuous decoding of codewords. The input data should be held for the number of clock symbols specified.
- **Provide erase port:** when checked, the block is given a `erase` input.
- **Provide variable block length port (`n_in`):** when checked, the block is given a `n_in` input.

- **Provide original delayed data port (data_del):** when checked, the block is given a data_del output.
- **Symbol width:** tells the width in bits for symbols in the code. The encoder support widths from 3 to 12.
- **Number of symbols per code block(n):** tells the number of symbols in the blocks the encoder produces. Acceptable numbers range from 3 to $2^s - 1$, where s denotes the symbol width.
- **Number of information symbols per code block(k):** tells the number of information symbols each block contains. Acceptable values range from $\max(n - 256, 1)$ to $n - 2$.
- **Field polynomial:** specifies the polynomial from which the symbol field is derived. It must be specified as a binary array whose first (respectively, last) element tells the coefficient of the highest degree (respectively, constant) term. This polynomial must be primitive. A value of zero indicates the default polynomial should be used. Default polynomials are listed in the table below.

Symbol Width	Default Polynomials	Array Representation
3	$x^3 + x + 1$	[1 0 1 1]
4	$x^4 + x + 1$	[1 0 0 1 1]
5	$x^5 + x^2 + 1$	[1 0 0 0 1 1]
6	$x^6 + x + 1$	[1 0 0 0 0 1 1]
7	$x^7 + x^3 + 1$	[1 0 0 0 1 0 0 1]
8	$x^8 + x^4 + x^3 + x^2 + 1$	[1 0 0 0 1 1 1 0 1]
9	$x^9 + x^4 + 1$	[1 0 0 0 0 1 0 0 0 1]
10	$x^{10} + x^3 + 1$	[1 0 0 0 0 0 0 1 0 0 1]
11	$x^{11} + x^2 + 1$	[1 0 0 0 0 0 0 0 0 1 0 1]
12	$x^{12} + x^6 + x^4 + x + 1$	[1 0 0 0 0 0 1 0 1 0 0 1 1]

- **Generator start:** specifies the first root r of the generator polynomial. The generator polynomial $g(x)$, is given by:

$$g(x) = \prod_{j=r}^{r-1} (x - \alpha^{h(j-r)})$$

where α is a primitive element of the symbol field, and the scaling factor is described below.

- **Scaling factor for generator polynomial:** (represented in the previous formula as h) specifies the scaling factor for the code. Ordinarily, h is 1, but can be as large as $2S - 1$ where s is the symbol width. The value must be chosen so that α^h is primitive. That is, h must be relatively prime to $2S - 1$.
- **Memory type:** allows to select between distributed, block and automatic memory choices.
- **Optimisation:** allows to select between area and speed optimisation.
- **Self recovering state machine:** when checked, the block synchronously resets itself if it enters an illegal state.

Other parameters used by this block are described in the Common Parameters section at the beginning of this chapter.

Xilinx LogiCore

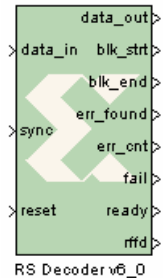
The RS Decoder block uses Xilinx LogiCORE RS Decoder.

This is a licensed core, available for purchase on the Xilinx web site at:

http://www.xilinx.com/xlnx/xebiz/designResources/ip_product_details.jsp?key=DO-DI-RSD.

RS Decoder v6_0

This block is listed in the following Xilinx Blockset libraries: *Communication and Index*.



Reed-Solomon (RS) codes are block-based error correcting codes with a wide range of applications in digital communications and storage. They are used to correct errors in many systems such as digital storage devices, wireless/ mobile communications, and digital video broadcasting.

The Reed-Solomon decoder processes blocks generated by a Reed-Solomon encoder, attempting to correct errors and recover information symbols. The number and type of errors that can be corrected depend on the characteristics of the code.

Reed-Solomon codes are Bose-Chaudhuri-Hocquenghem (BCH) codes, which in turn are linear block codes. An (n,k) linear block code is a k -dimensional subspace of an n -dimensional vector space over a finite field. Elements of the field are called *symbols*. For a Reed-Solomon code, n ordinarily is $[equation]$, where s is the width in bits of each symbol. When the code is *shortened*, n is smaller. The decoder handles both full length and shortened codes. It is also able to handle *erasures*, that is, symbols that are known with high probability to contain errors.

When the decoder processes a block, there are three possibilities:

1. The information symbols are recovered. This is the case provided $2p+r < n-k$, where p is the number of errors and r is the number of erasures.
2. The decoder reports it is unable to recover the information symbols.
3. The decoder fails to recover the information symbols but does not report an error.

The probability of each possibility depends on the code and the nature of the communications channel. Simulink provides excellent tools for modeling channels and estimating these probabilities.

Block Interface

The Xilinx RS Decoder block has inputs `data_in`, `sync` and `reset` and outputs `data_out`, `blk_strt`, `blk_end`, `err_found`, `err_cnt`, `fail`, `ready` and `rfd`. It also has optional inputs `n_in`, `erase`, `rst`, and `en`, and optional output ports `erase_cnt` and `data_del`.

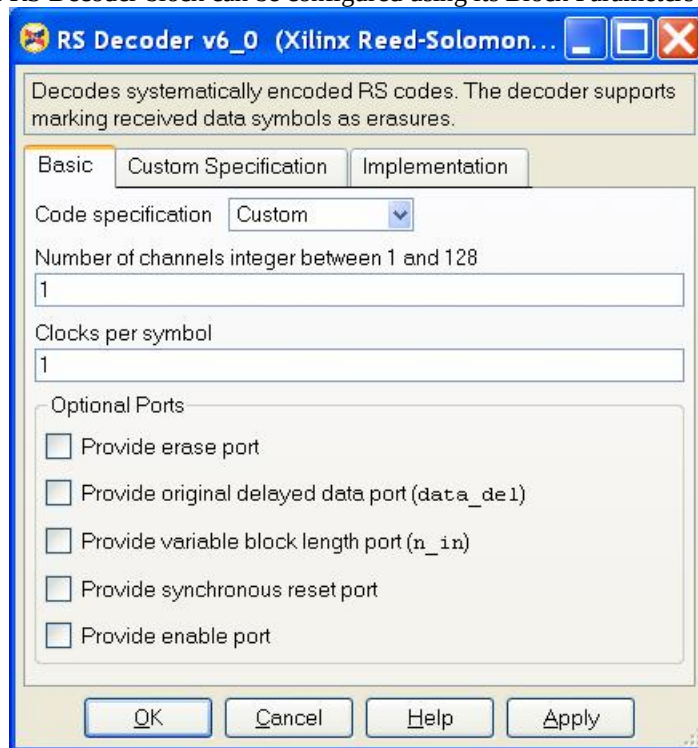
The following describes these ports in detail:

- **data_in**: presents blocks of n symbols to be decoded. The `din` signal must have type `UFIX_S_0`, where s is the width in bits of each symbol.
- **sync**: tells the decoder when to begin processing symbols from `data_in`. The decoder discards input symbols until the first time `sync` is asserted. The symbol on which `sync` is asserted marks the beginning of the first n symbol block to be processed by the decoder. The `sync` signal is ignored till the decoder is ready to accept another code block. The signal driving `sync` must be Bool.
- **reset**: asynchronously resets the decoder. The signal driving `reset` must be Bool. Note: `reset` must be asserted high for atleast 1 sample period before the decoder can start decoding code symbols.
- **erase**: indicates the symbol currently presented on `din` should be treated as an erasure. The signal driving `erase` must be Bool.
- **n_in**: `n_in` is sampled at the start of each block. The new block's length, `n_block`, is set to `n_in` sampled. The `n_in` signal must have type `UFIX_S_0`, where s is the width in bits of each symbol.
- **rst**: carries the reset signal for the decoder. The signal driving `rst` must be Bool.
- **en**: carries the enable signal for the decoder. The signal driving `en` must be Bool.

- **data_out**: produces the information and parity symbols resulting from decoding. The type of **data_out** is the same as that for **data_in**.
- **blk_strt**: presents a 1 at the time **data_out** presents the first symbol of the block. **blk_strt** produces a signal of **UFIX_1_0** type.
- **blk_end**: presents a 0 at the time **data_out** presents the last symbol of the block. **blk_end** produces a signal of **UFIX_1_0** type.
- **err_found**: presents a value at the time **data_out** presents the last symbol of the block. The value is 1 if the decoder detected any errors or erasures during decoding. **err_found** must have type **UFIX_1_0**.
- **err_cnt**: presents a value at the time **data_out** presents the last symbol of the block. The value is the number of errors that were corrected. **err_cnt** must have type **UFIX_b_0** where **b** is the number of bits needed to represent **n-k**.
- **fail**: presents a value at the time **dout** presents the last symbol of the block. The value is 1 if the decoder was unable to recover the information symbols, and 0 otherwise. **fail** must have type **UFIX_1_0**.
- **ready**: value is 1 when the decoder is ready to sample **data_in** input, and 0 otherwise. **ready** must have type **UFIX_1_0**.
- **rffd**: value is 1 when the decoder is ready to sample the first symbol of a code block on **data_in** input, and 0 otherwise. **rffd** must have type **UFIX_1_0**.
- **data_del**: produces the undecoded symbols alongside the decoded symbols on **data_out**. The type of **data_del** is the same as that for **data_in**.
- **erase_cnt**: only available when erasure decoding is enabled. Presents a value at the time **dout** presents the last symbol of the block. The value is the number of erasures that were corrected. **erase_cnt** must have type **UFIX_b_0** where **b** is the number of bits needed to represent **n**.

Block Parameters Dialog Box

The RS Decoder block can be configured using its Block Parameters dialog box.



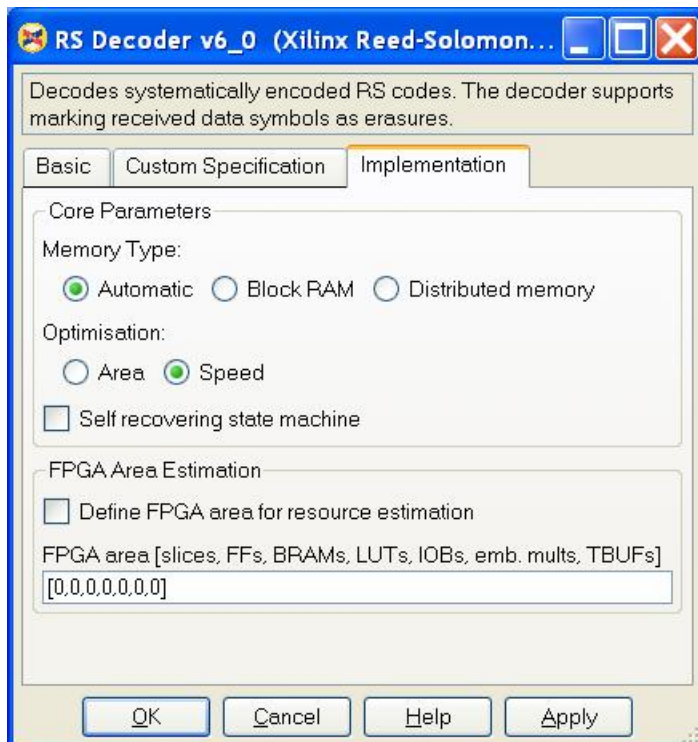
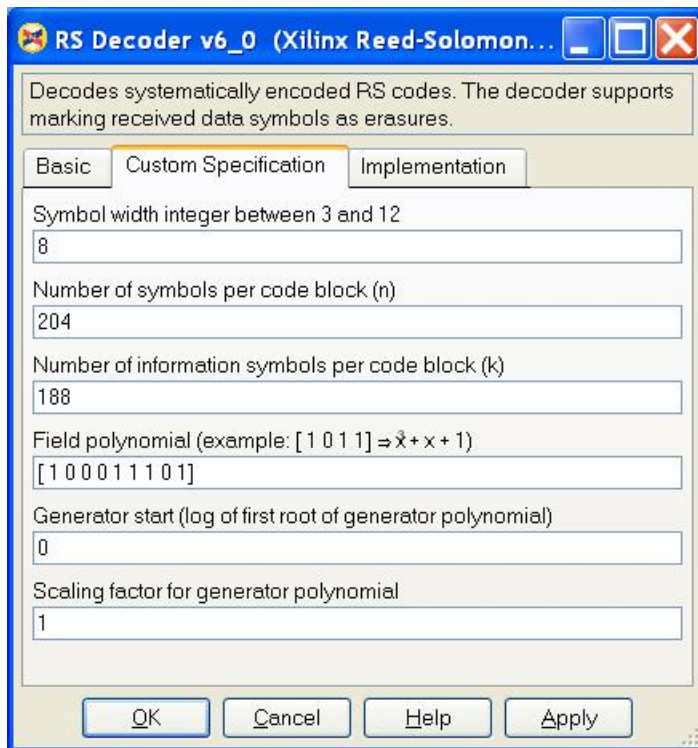


Figure 1: Reed-Solomon Decoder block parameters dialog box

Parameters specific to the RS Decoder block are:

- **Code specification:** specifies the type of RS Decoder desired. The choices are:
 - **Custom:** allows you to set all the block parameters.

- * **ATSC**: implements ATSC (Advanced Television Systems Committee) standard (207, 187) shortened RS code.
- * **CCSDS**: implements CCSDS (Consultative Committee for Space Data Systems) standard (255, 223) full length RS code.
- * **DVB**: implements DVB (Digital Video Broadcasting) standard (204, 188) shortened RS code.
- * **IESS-308 (126)**: implements IESS-308 (INTELSAT Earth Station Standard) specification (126, 112) shortened RS code.
- * **IESS-308 (194)**: implements IESS-308 specification (194, 178) shortened RS code.
- * **IESS-308 (208)**: implements IESS-308 specification (208, 192) shortened RS code.
- * **IESS-308 (219)**: implements IESS-308 specification (219, 201) shortened RS code.
- * **IESS-308 (225)**: implements IESS-308 specification (225, 205) shortened RS code.
- * **IEEE-802.16d**: implements IEEE-802.16d specification (255, 239) full length RS code.
- * **Number of channels**: tells the number of separate time division multiplexed channels to be processed by the encoder. The encoder supports upto 128 channels.
- * **Clocks per symbol**: tells the number of sample periods to use per input data symbol. This may be increased to reduce the processing delay and support continuous decoding of codewords. The input data should be held for the number of clock symbols specified.
- * **Provide erase port**: when checked, the block is given a `erase` input.
- * **Provide variable block length port (n_in)**: when checked, the block is given a `n_in` input.
- * **Provide original delayed data port (data_del)**: when checked, the block is given a `data_del` output.
- * **Symbol width**: tells the width in bits for symbols in the code. The encoder support widths from 3 to 12.
- * **Number of symbols per code block(n)**: tells the number of symbols in the blocks the encoder produces. Acceptable numbers range from 3 to $2^s - 1$, where s denotes the symbol width.
- * **Number of information symbols per code block(k)**: tells the number of information symbols each block contains. Acceptable values range from $\max(n - 256, 1)$ to $n - 2$.
- * **Field polynomial**: specifies the polynomial from which the symbol field is derived. It must be specified as a binary array whose first (respectively, last) element tells the coefficient of the highest degree (respectively, constant) term. This polynomial must be primitive. A value of zero indicates the default polynomial should be used. Default polynomials are listed in the table below.

Symbol Width	Default Polynomials	Array Representation
3	$x^3 + x + 1$	[1 0 1 1]
4	$x^4 + x + 1$	[1 0 0 1 1]
5	$x^5 + x^2 + 1$	[1 0 0 0 1 1]
6	$x^6 + x + 1$	[1 0 0 0 0 1 1]
7	$x^7 + x^3 + 1$	[1 0 0 0 1 0 0 1]
8	$x^8 + x^4 + x^3 + x^2 + 1$	[1 0 0 0 1 1 1 0 1]
9	$x^9 + x^4 + 1$	[1 0 0 0 0 1 0 0 0 1]
10	$x^{10} + x^3 + 1$	[1 0 0 0 0 0 0 1 0 0 1]
11	$x^{11} + x^2 + 1$	[1 0 0 0 0 0 0 0 0 1 0 1]
12	$x^{12} + x^6 + x^4 + x + 1$	[1 0 0 0 0 0 1 0 1 0 0 1 1]

- * **Generator start**: specifies the first root r of the generator polynomial. The generator polynomial $g(x)$, is given by:

$$g(x) = \prod_{i=r}^{n-r-1} (x - \alpha^{2^i - 1})$$

- where α is a primitive element of the symbol field, and the scaling factor is described below.
- **Scaling factor for generator polynomial:** (represented in the previous formula as h) specifies the scaling factor for the code. Ordinarily, h is 1, but can be as large as $2S - 1$ where s is the symbol width. The value must be chosen so that αh is primitive. That is, h must be relatively prime to $2S - 1$.
 - **Memory type:** allows to select between distributed, block and automatic memory choices.
 - **Optimisation:** allows to select between area and speed optimisation.
 - **Self recovering state machine:** when checked, the block synchronously resets itself if it enters an illegal state.

Other parameters used by this block are described in the Common Parameters section at the beginning of this chapter.

Xilinx LogiCore

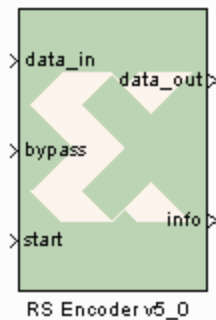
The RS Decoder block uses Xilinx LogiCORE RS Decoder.

This is a licensed core, available for purchase on the Xilinx web site at:

http://www.xilinx.com/xlnx/xebiz/designResources/ip_product_details.jsp?key=DO-DI-RSD.

RS Encoder v5_0

This block is listed in the following Xilinx Blockset libraries: Communication and Index.



RS Encoder v5_0

Reed-Solomon (RS) codes are block-based error correcting codes with a wide range of applications in digital communications and storage. They are used to correct errors in many systems such as digital storage devices, wireless or mobile communications, and digital video broadcasting.

The Reed-Solomon encoder augments data blocks with redundant symbols so that errors introduced during transmission can be corrected. Errors may occur for a number of reasons (noise or interference, scratches on a CD, etc.). The Reed-Solomon decoder attempts to correct errors and recover the original data. The number and type of errors that can be corrected depends on the characteristics of the code.

A typical system is shown below:

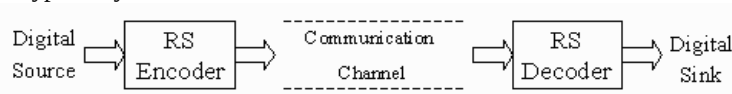


Figure 1: Example of a system using Reed-Solomon codes

Reed-Solomon codes are Bose-Chaudhuri-Hocquenghem (BCH) codes, which in turn are linear block codes. An (n, k) linear block code is a k -dimensional subspace of an n -dimensional vector space over a finite field. Elements of the field are called symbols. For a Reed-Solomon code, n ordinarily is $2^s - 1$, where s is the width in bits of each symbol. When the code is shortened, n is smaller. The encoder handles both full length and shortened codes.

The encoder is systematic. This means it constructs code blocks of length n from information blocks of length k by adjoining $n-k$ parity symbols.

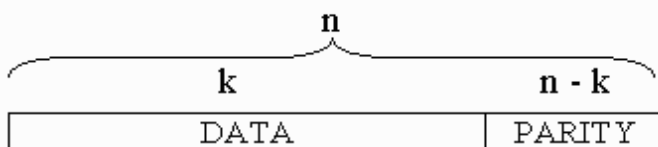


Figure 2: Example of a Reed Solomon codeword

A Reed-Solomon code is characterized by its field and generator polynomials. The field polynomial is used to construct the symbol field, and the generator polynomial is used to calculate parity symbols. The encoder allows both polynomials to be configured. The generator polynomial has the form:

$$g(x) = (x - \alpha^j)(x - \alpha^{j+1}) \dots (x - \alpha^{j+n-k-1})$$

where α is a primitive element of the finite field having $n + 1$ elements.

Block Interface

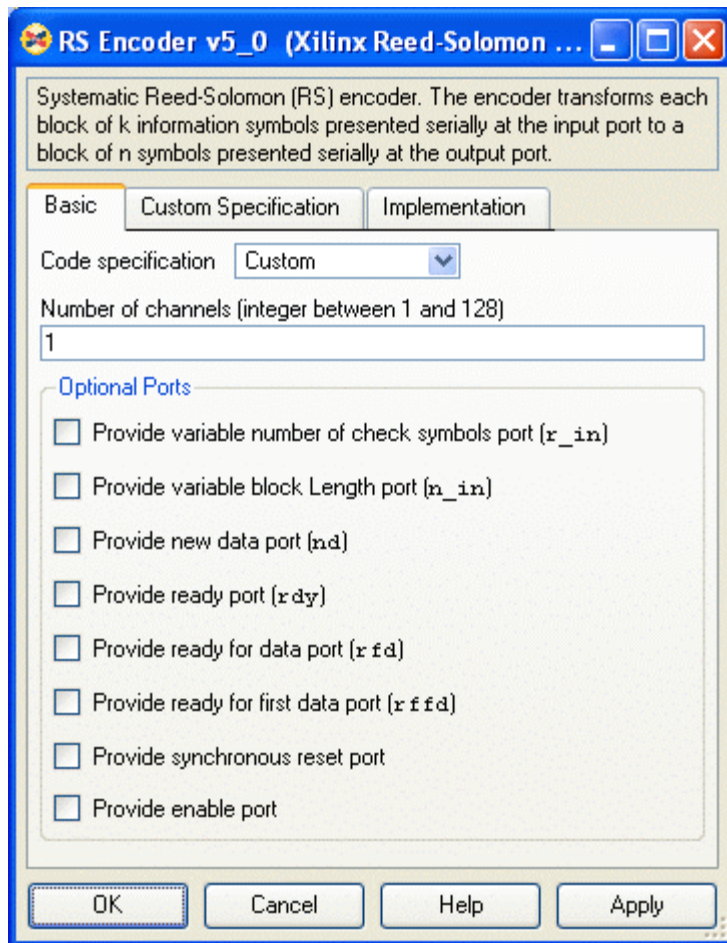
The Xilinx Reed-Solomon Encoder block has inputs `data_in`, `bypass` and `start`, and outputs `data_out` and `info`. It also has optional inputs `n_in`, `r_in`, `nd`, `rst` and `en`. It also has optional outputs `rdy`, `rfd`, and `rffd`.

The following describes the ports in detail:

- **data_in**: presents blocks of symbols to be encoded. Each block consists of k information symbols followed by $n - k$ uninterpreted filler symbols. The **din** signal must have type **UFIX_s_0**, where s is the width in bits of each symbol.
- **start**: tells the encoder when to begin processing symbols from **din**. The encoder discards input symbols until the first time **start** is asserted. The symbol on which **start** is asserted marks the beginning of the first n symbol blocks to be processed by the encoder. If **start** is asserted for more than one sample period, the value at the last period is taken as the beginning of the block. The **start** signal is ignored if **bypass** is asserted simultaneously. The signal driving **start** must be **Bool**.
- **bypass**: when **bypass** is asserted, the value on **din** is passed unchanged to **dout** with a delay of 4 (6 in the case of CCSDS) sample periods. The **bypass** signal has no effect on the state of the encoder. The signal driving **bypass** must be **Bool**.
- **n_in**: **n_in** is sampled at the start of each block. The new block's length, **n_block**, is set to **n_in** sampled. The **n_in** signal must have type **UFIX_s_0**, where s is the width in bits of each symbol.
- **r_in**: **n_in** is sampled at the start of each block. The new block's length, **r_block**, is set to **r_in** sampled. The **r_in** signal must have type **UFIX_p_0**, where p is the number of bits required to represent the parity bits ($n-k$) in the default codeword.
- **nd**: marks each **data_in** symbol as part of the information symbols for processing parity symbols. The signal driving **nd** must be **Bool**.
- **rst**: carries the reset signal. The signal driving **rst** must be **Bool**.
- **en**: carries the enable signal. The signal driving **en** must be **Bool**.
- **data_out**: produces blocks of n symbols that represent the results of encoding blocks of k information symbols read from **data_in**. The type of **data_out** is the same as that for **data_in**.
- **info**: equals 1 (respectively, 0) when the value presented on **data_out** is an information (respectively, parity) symbol. **info** must have type **UFIX_1_0**.
- **rdy**: marks each symbol produced on **data_out** as valid or invalid. **rdy** must have type **UFIX_1_0**.
- **rfd**: equals 1 when the encoder is accepting and producing information symbols, and is 0 when producing parity symbols. **rfd** must have type **UFIX_1_0**.
- **rffd**: equals 1 when the encoder is ready to accept a new start pulse. **rffd** must have type **UFIX_1_0**.

Block Parameters Dialog Box

The RS Encoder block can be configured using its Block Parameters dialog box.



 **RS Encoder v5_0 (Xilinx Reed-Solomon ...)** [Minimize] [Maximize] [Close]

Systematic Reed-Solomon (RS) encoder. The encoder transforms each block of k information symbols presented serially at the input port to a block of n symbols presented serially at the output port.

Basic Custom Specification Implementation

Symbol width (integer between 3 and 12)

n (number of symbols per code block)

k (number of information symbols per code block)

Field polynomial (example: [1 0 1 1] => $x^3 + x + 1$)

Generator start (log of first root of generator polynomial)

Scaling factor for generator polynomial

Simulation

☐ Override with doubles

OK Cancel Help Apply

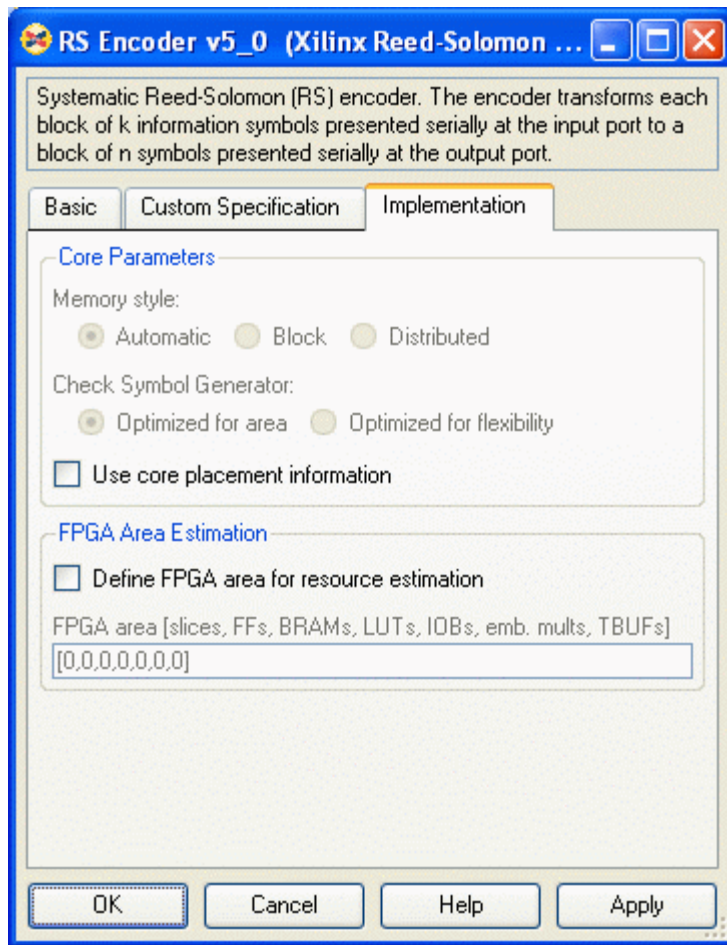


Figure 3: Reed-Solomon Encoder block parameters dialog box

Parameters specific to the RS Encoder block are:

- **Code specification:** specifies the encoder type. The choices are:
 - **Custom:** allows you to set all the block parameters.
 - **ATSC:** implements ATSC (Advanced Television Systems Committee) standard (207, 187) shortened RS code.
 - **CCSDS:** implements CCSDS (Consultative Committee for Space Data Systems) standard (255, 223) full length RS code.
 - **DVB:** implements DVB (Digital Video Broadcasting) standard (204, 188) shortened RS code.
 - **IESS-308 (126):** implements IESS-308 (INTELSAT Earth Station Standard) specification (126, 112) shortened RS code.
 - **IESS-308 (194):** implements IESS-308 specification (194, 178) shortened RS code.
 - **IESS-308 (208):** implements IESS-308 specification (208, 192) shortened RS code.
 - **IESS-308 (219):** implements IESS-308 specification (219, 201) shortened RS code.
 - **IESS-308 (225):** implements IESS-308 specification (225, 205) shortened RS code.
 - **ITU-J.83 Annex B:** implements ITU-J.83 Annex B specification (128, 122) extended RS code.
- **Number of channels:** tells the number of separate time division multiplexed channels to be processed by the encoder. The encoder supports upto 128 channels.
- **Provide variable number of check symbols (r_in):** when checked, the block is given a r_in and n_in input.

- **Provide variable block length port (n_in):** when checked, the block is given a n_in input.
- **Provide new data port (nd):** when checked, the block is given a nd input.
- **Provide ready port (rdy):** when checked, the block is given a rdy output.
- **Provide ready for data port (rfd):** when checked, the block is given a rfd output.
- **Provide ready for first data port (rffd):** when checked, the block is given a rffd output.
- **Symbol width:** tells the width in bits for symbols in the code. The encoder support widths from 3 to 12.
- **n (number of symbols per code block):** tells the number of symbols in the blocks the encoder produces. Acceptable numbers range from 3 to $2^s - 1$, where s denotes the symbol width.
- **k (number of information symbols per code block):** tells the number of information symbols each block contains. Acceptable values range from $\max(n - 256, 1)$ to $n - 2$.
- **Field polynomial:** specifies the polynomial from which the symbol field is derived. It must be specified as a binary array whose first (resp., last) element tells the coefficient of the highest degree (resp., constant) term. This polynomial must be primitive. A value of zero indicates the default polynomial should be used. Default polynomials are listed in the table below.

Symbol Width	Default Polynomials	Array Representation
3	$x^3 + x + 1$	[1 0 1 1]
4	$x^4 + x + 1$	[1 0 0 1 1]
5	$x^5 + x^2 + 1$	[1 0 0 0 1 1]
6	$x^6 + x + 1$	[1 0 0 0 0 1 1]
7	$x^7 + x^3 + 1$	[1 0 0 0 1 0 0 1]
8	$x^8 + x^4 + x^3 + x^2 + 1$	[1 0 0 0 1 1 1 0 1]
9	$x^9 + x^4 + 1$	[1 0 0 0 0 1 0 0 0 1]
10	$x^{10} + x^3 + 1$	[1 0 0 0 0 0 0 1 0 0 1]
11	$x^{11} + x^2 + 1$	[1 0 0 0 0 0 0 0 0 1 0 1]
12	$x^{12} + x^6 + x^4 + x + 1$	[1 0 0 0 0 0 1 0 1 0 0 1 1]

- **Generator start:** specifies the first root r of the generator polynomial. The generator polynomial $g(x)$ is given by:

$$g(x) = \prod_{i=0}^{n-k-1} (x - \alpha^{k(r+i)})$$

where α is a primitive element of the symbol field, and the scaling factor h is described below.

- **Scaling factor h:** specifies the scaling factor for the code. Ordinarily h is 1, but can be as large as $2^s - 1$ where s is the symbol width. The value must be chosen so that α^h is primitive, i.e., h must be relatively prime to $2^s - 1$.
- **Memory style:** allows selection of the distributed, block or automatic memory choices. This option is available only for CCSDS codes.
- **Check symbol generator:** allows selection of optimization for area or flexibility. This option is available when variable number of check symbols are presented at the encoder input.

Other parameters used by this block are described in the Common Parameters section at the beginning of this chapter.

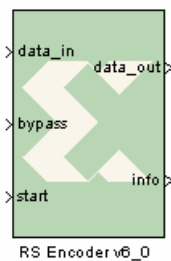
Xilinx LogiCore

The RS Encoder block uses Xilinx LogiCORE RS Encoder.

This is a licensed core, available for purchase on the Xilinx web site at:
http://www.xilinx.com/xlnx/xebiz/designResources/ip_product_details.jsp?key=DO-DI-RSE.

RS Encoder v6_0

This block is listed in the following Xilinx Blockset libraries: Communication and Index.



Reed-Solomon (RS) codes are block-based error correcting codes with a wide range of applications in digital communications and storage. They are used to correct errors in many systems such as digital storage devices, wireless or mobile communications, and digital video broadcasting.

The Reed-Solomon encoder augments data blocks with redundant symbols so that errors introduced during transmission can be corrected. Errors may occur for a number of reasons (noise or interference, scratches on a CD, etc.). The Reed-Solomon decoder attempts to correct errors and recover the original data. The number and type of errors that can be corrected depends on the characteristics of the code.

A typical system is shown below:

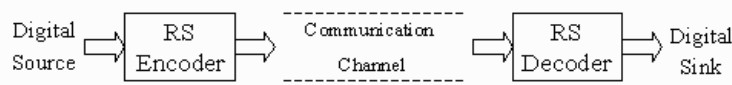


Figure 1: Example of a system using Reed-Solomon codes

Reed-Solomon codes are Bose-Chaudhuri-Hocquenghem (BCH) codes, which in turn are linear block codes. An (n, k) linear block code is a k -dimensional subspace of an n -dimensional vector space over a finite field. Elements of the field are called symbols. For a Reed-Solomon code, n ordinarily is $2^s - 1$, where s is the width in bits of each symbol. When the code is shortened, n is smaller. The encoder handles both full length and shortened codes.

The encoder is systematic. This means it constructs code blocks of length n from information blocks of length k by adjoining $n-k$ parity symbols.

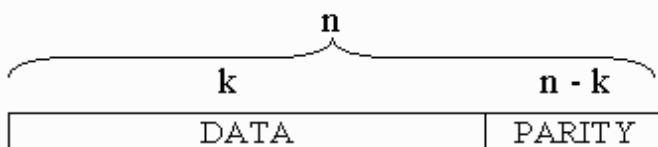


Figure 2: Example of a Reed Solomon codeword

A Reed-Solomon code is characterized by its field and generator polynomials. The field polynomial is used to construct the symbol field, and the generator polynomial is used to calculate parity symbols. The encoder allows both polynomials to be configured. The generator polynomial has the form:

$$g(x) = (x - \alpha^j)(x - \alpha^{j+1}) \dots (x - \alpha^{j+n-k-1})$$

where α is a primitive element of the finite field having $n + 1$ elements.

Block Interface

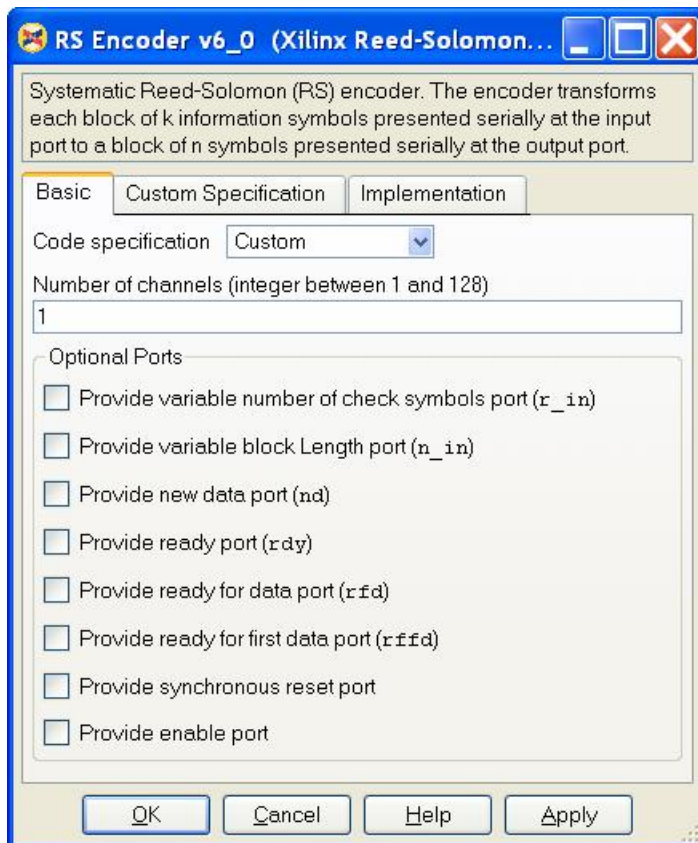
The Xilinx Reed-Solomon Encoder block has inputs `data_in`, `bypass` and `start`, and outputs `data_out` and `info`. It also has optional inputs `n_in`, `r_in`, `nd`, `rst` and `en`. It also has optional outputs `rdy`, `rfd`, and `rffd`.

The following describes the ports in detail:

- **data_in**: presents blocks of symbols to be encoded. Each block consists of k information symbols followed by $n - k$ uninterpreted filler symbols. The **din** signal must have type **UFIX_s_0**, where s is the width in bits of each symbol.
- **start**: tells the encoder when to begin processing symbols from **din**. The encoder discards input symbols until the first time **start** is asserted. The symbol on which **start** is asserted marks the beginning of the first n symbol blocks to be processed by the encoder. If **start** is asserted for more than one sample period, the value at the last period is taken as the beginning of the block. The **start** signal is ignored if **bypass** is asserted simultaneously. The signal driving **start** must be **Bool**.
- **bypass**: when **bypass** is asserted, the value on **din** is passed unchanged to **dout** with a delay of 4 (6 in the case of CCSDS) sample periods. The **bypass** signal has no effect on the state of the encoder. The signal driving **bypass** must be **Bool**.
- **n_in**: **n_in** is sampled at the start of each block. The new block's length, **n_block**, is set to **n_in** sampled. The **n_in** signal must have type **UFIX_s_0**, where s is the width in bits of each symbol.
- **r_in**: **n_in** is sampled at the start of each block. The new block's length, **r_block**, is set to **r_in** sampled. The **r_in** signal must have type **UFIX_p_0**, where p is the number of bits required to represent the parity bits ($n-k$) in the default codeword.
- **nd**: marks each **data_in** symbol as part of the information symbols for processing parity symbols. The signal driving **nd** must be **Bool**.
- **rst**: carries the reset signal. The signal driving **rst** must be **Bool**.
- **en**: carries the enable signal. The signal driving **en** must be **Bool**.
- **data_out**: produces blocks of n symbols that represent the results of encoding blocks of k information symbols read from **data_in**. The type of **data_out** is the same as that for **data_in**.
- **info**: equals 1 (respectively, 0) when the value presented on **data_out** is an information (respectively, parity) symbol. **info** must have type **UFIX_1_0**.
- **rdy**: marks each symbol produced on **data_out** as valid or invalid. **rdy** must have type **UFIX_1_0**.
- **rfd**: equals 1 when the encoder is accepting and producing information symbols, and is 0 when producing parity symbols. **rfd** must have type **UFIX_1_0**.
- **rffd**: equals 1 when the encoder is ready to accept a new start pulse. **rffd** must have type **UFIX_1_0**.

Block Parameters Dialog Box

The RS Encoder block can be configured using its Block Parameters dialog box.



RS Encoder v6_0 (Xilinx Reed-Solomon...

Systematic Reed-Solomon (RS) encoder. The encoder transforms each block of k information symbols presented serially at the input port to a block of n symbols presented serially at the output port.

Basic Custom Specification Implementation

Symbol width (integer between 3 and 12)
8

n (number of symbols per code block)
204

k (number of information symbols per code block)
188

Field polynomial (example: [1 0 1 1] => $x^3 + x + 1$)
[1 0 0 0 1 1 1 0 1]

Generator start (log of first root of generator polynomial)
0

Scaling factor for generator polynomial
1

Simulation
☐ Override with doubles

OK Cancel Help Apply

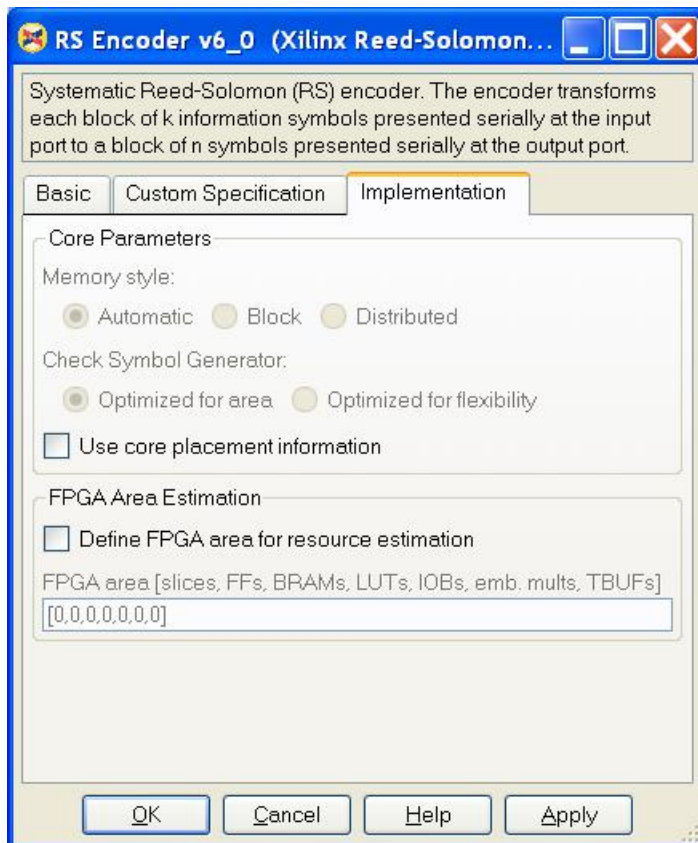


Figure 3: Reed-Solomon Encoder block parameters dialog box

Parameters specific to the RS Encoder block are:

- **Code specification:** specifies the encoder type. The choices are:
 - **Custom:** allows you to set all the block parameters.
 - **ATSC:** implements ATSC (Advanced Television Systems Committee) standard (207, 187) shortened RS code.
 - **CCSDS:** implements CCSDS (Consultative Committee for Space Data Systems) standard (255, 223) full length RS code.
 - **DVB:** implements DVB (Digital Video Broadcasting) standard (204, 188) shortened RS code.
 - **IESS-308 (126):** implements IESS-308 (INTELSAT Earth Station Standard) specification (126, 112) shortened RS code.
 - **IESS-308 (194):** implements IESS-308 specification (194, 178) shortened RS code.
 - **IESS-308 (208):** implements IESS-308 specification (208, 192) shortened RS code.
 - **IESS-308 (219):** implements IESS-308 specification (219, 201) shortened RS code.
 - **IESS-308 (225):** implements IESS-308 specification (225, 205) shortened RS code.
 - **ITU-J.83 Annex B:** implements ITU-J.83 Annex B specification (128, 122) extended RS code.
- **Number of channels:** tells the number of separate time division multiplexed channels to be processed by the encoder. The encoder supports upto 128 channels.
- **Provide variable number of check symbols (r_in):** when checked, the block is given a r_in and n_in input.
- **Provide variable block length port (n_in):** when checked, the block is given a n_in input.
- **Provide new data port (nd):** when checked, the block is given a nd input.
- **Provide ready port (rdy):** when checked, the block is given a rdy output.

- **Provide ready for data port (rfd):** when checked, the block is given a rfd output.
- **Provide ready for first data port (rffd):** when checked, the block is given a rffd output.
- **Symbol width:** tells the width in bits for symbols in the code. The encoder support widths from 3 to 12.
- **n (number of symbols per code block):** tells the number of symbols in the blocks the encoder produces. Acceptable numbers range from 3 to $2^s - 1$, where s denotes the symbol width.
- **k (number of information symbols per code block):** tells the number of information symbols each block contains. Acceptable values range from $\max(n - 256, 1)$ to $n - 2$.
- **Field polynomial:** specifies the polynomial from which the symbol field is derived. It must be specified as a binary array whose first (resp., last) element tells the coefficient of the highest degree (resp., constant) term. This polynomial must be primitive. A value of zero indicates the default polynomial should be used. Default polynomials are listed in the table below.

Symbol Width	Default Polynomials	Array Representation
3	$x^3 + x + 1$	[1 0 1 1]
4	$x^4 + x + 1$	[1 0 0 1 1]
5	$x^5 + x^2 + 1$	[1 0 0 0 1 1]
6	$x^6 + x + 1$	[1 0 0 0 0 1 1]
7	$x^7 + x^3 + 1$	[1 0 0 0 1 0 0 1]
8	$x^8 + x^4 + x^3 + x^2 + 1$	[1 0 0 0 1 1 1 0 1]
9	$x^9 + x^4 + 1$	[1 0 0 0 0 1 0 0 0 1]
10	$x^{10} + x^3 + 1$	[1 0 0 0 0 0 0 1 0 0 1]
11	$x^{11} + x^2 + 1$	[1 0 0 0 0 0 0 0 0 1 0 1]
12	$x^{12} + x^6 + x^4 + x + 1$	[1 0 0 0 0 0 1 0 1 0 0 1 1]

- **Generator start:** specifies the first root r of the generator polynomial. The generator polynomial $g(x)$ is given by:

$$g(x) = \prod_{i=0}^{n-k-1} (x - \alpha^{k(r+i)})$$

where α is a primitive element of the symbol field, and the scaling factor h is described below.

- **Scaling factor h:** specifies the scaling factor for the code. Ordinarily h is 1, but can be as large as $2^s - 1$ where s is the symbol width. The value must be chosen so that α^h is primitive, i.e., h must be relatively prime to $2^s - 1$.
- **Memory style:** allows selection of the distributed, block or automatic memory choices. This option is available only for CCSDS codes.
- **Check symbol generator:** allows selection of optimization for area or flexibility. This option is available when variable number of check symbols are presented at the encoder input.

Other parameters used by this block are described in the Common Parameters section at the beginning of this chapter.

Xilinx LogiCore

The RS Encoder block uses Xilinx LogiCORE RS Encoder.

This is a licensed core, available for purchase on the Xilinx web site at:

http://www.xilinx.com/xlnx/xebiz/designResources/ip_product_details.jsp?key=DO-DI-RSE.

Sample Time

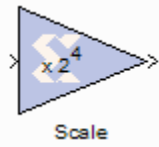
This block is listed in the following Xilinx Blockset libraries: Tools and Index.



The Sample Time block reports the normalized sample period of its input. A signal's normalized sample period is not equivalent to its Simulink absolute sample period. In hardware, this block is implemented as a constant.

Scale

This block is listed in the following Xilinx Blockset libraries: *Data Types, Math, and Index.*



The Xilinx Scale block scales its input by a power of two. The power can be either positive or negative. The block has one input and one output. The scale operation has the effect of moving the binary point without changing the bits in the container.

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

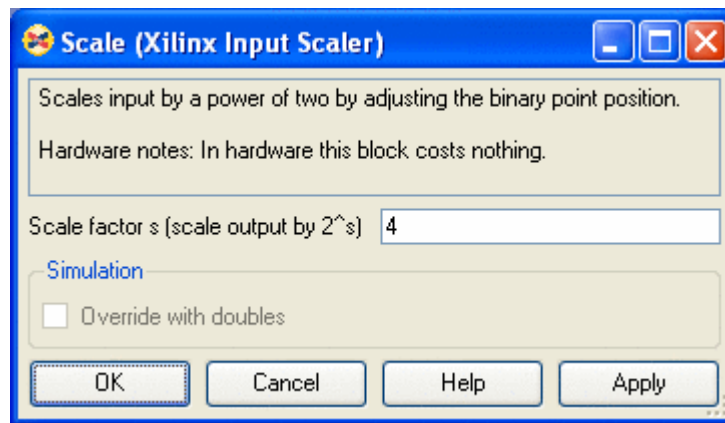


Figure 1: Scale block parameters dialog box

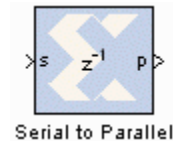
The only parameter that is specific to the Scale block is **Scale factor s**. It can be a positive or negative integer. The output of the block is $i \cdot 2^k$, where i is the input value and k is the scale factor. The effect of scaling is to move the binary point, which in hardware has no cost (a *shift*, on the other hand, may add logic).

The other parameters used by this block are explained in the Common Parameters section at the beginning of this chapter.

The Scale block does not use a Xilinx LogiCORE.

Serial to Parallel

This block is listed in the following Xilinx Blockset libraries: *Basic Elements, Data Types, and Index.*



The Serial to Parallel block takes a series of inputs of any size and creates a single output of a specified multiple of that size. The input series can be ordered either with the most significant word first or the least significant word first.

The following waveform illustrates the block's behavior:

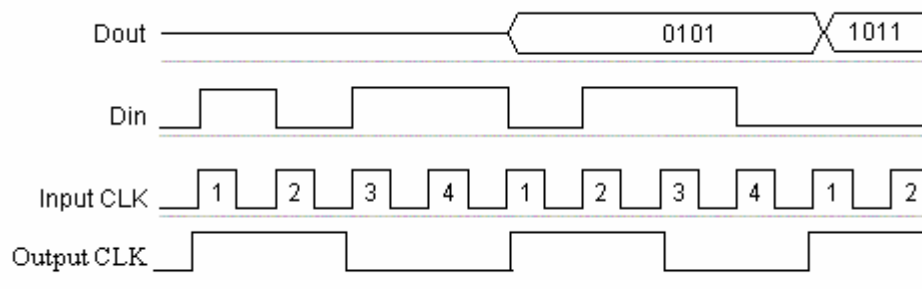


Figure 1: Example of Serial to Parallel behavior

This example illustrates the case where the input width is 1, output width is 4, word size is 1 bit, and the block is configured for most significant word first.

Block Interface

The Serial to Parallel block has one input and one output port. The input port can be any size. The output port size is indicated on the block parameters dialog box.

Block Parameters Dialog Box

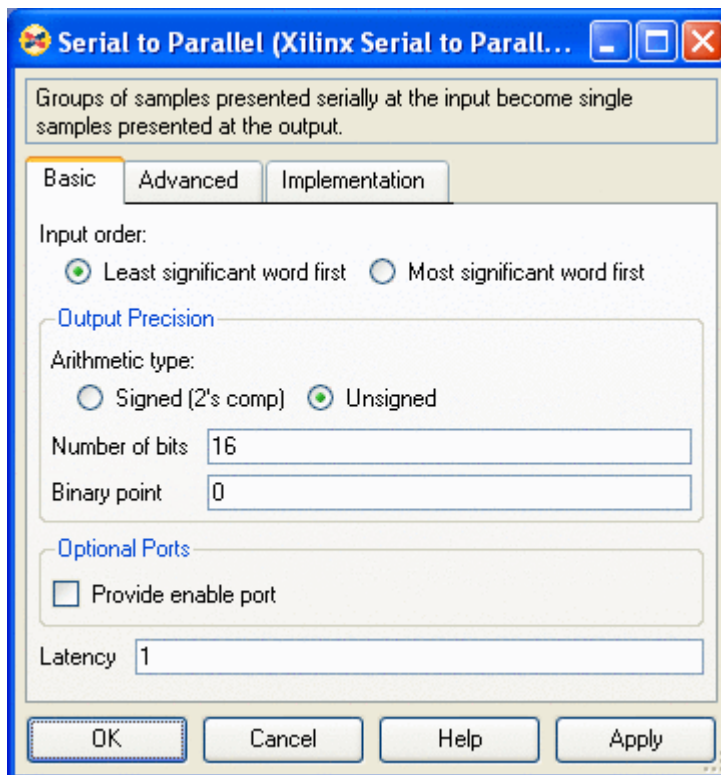


Figure 2: Serial to Parallel block Basic Tab

Parameters specific to the block are:

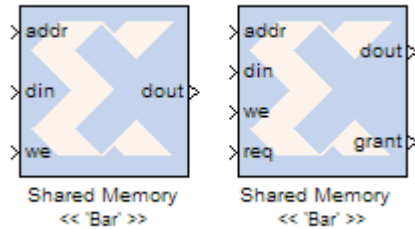
- **Input order:** Least or most significant word first.
- **Arithmetic type:** Signed or unsigned output.
- **Number of bits:** Output width which must be a multiple of the number of input bits.
- **Binary point:** Output binary point location.

Other parameters used by this block are explained in the Common Parameters section at the beginning of this chapter.

An error is reported when the number of output bits cannot be divided evenly by the number of input bits. The minimum latency for this block is zero.

Shared Memory

This block is listed in the following Xilinx Blockset libraries: Index, Shared Memory.



The Xilinx Shared block implements a random access memory (RAM) that can be shared among multiple designs or sections of a design. A Shared Memory Block is uniquely identified by its name. In the blocks above, the shared memory has been named "Bar". Instances of Shared Memory "Bar", whether within the same model or in different models or even different instances of Matlab, will share the same memory space. System Generator's hardware co-simulation interfaces allow shared memory blocks to be compiled and co-

simulated in FPGA hardware. These interfaces make it possible for hardware-based shared memory resources to map transparently to common address spaces on a host PC. When used in System Generator co-simulation hardware, shared memories facilitate high-speed data transfers between the host PC and FPGA, and bolster the tool's real-time hardware co-simulation capabilities.

Block Interface

By default, the shared memory block has 3 inputs (`addr`, `din` and `we`) and 1 output (`dout`). Access to the shared memory can be protected by setting the `Access protection` parameter to `Lockable`. Setting access protection to `Lockable` causes two additional ports to appear; an input port `req` and an output port `grant`.

The `addr` port should be driven by a signal of type `UFIX_N_0`, where N equals $\text{ceil}(\log_2(\text{depth}))$. The memory word size is determined, at compile-time, by the bit width of the signal driving `din`. Driving the write enable port (`we`) with 1 indicates that the value on the `din` port should be written to the memory address pointed to by port `addr`.

When access protection is set to `Lockable`, the `req` and `grant` ports are used to control access to the memory. Before a read or write can occur, a request must first be made by setting `req` to 1. When `grant` becomes 1, the request for access has been allowed and read or write operations can proceed. Figure 1 shows the relationship between the `req`, `grant` and `we` ports. The figure also shows that the block output is suppressed until access to the memory is granted.

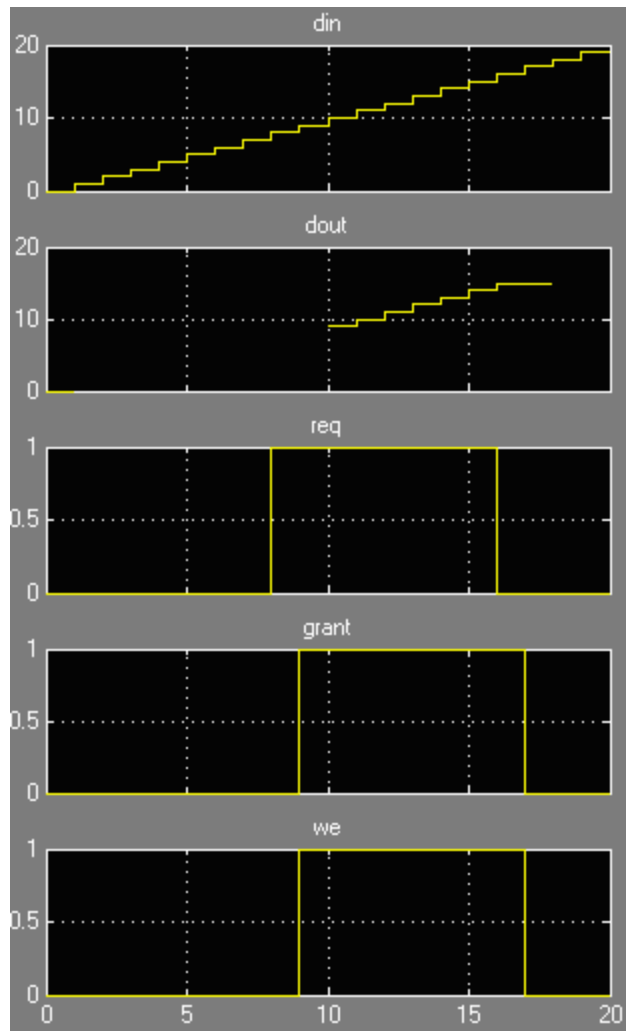


Figure 1: Illustration of grant and request operations.

The output during a write operation depends on the write mode. When the **we** is 0, the output port has the value at the location specified by the address line. During a write operation (**we** asserted), the data presented on the input data port is stored in memory at the location selected by the port's address input. During a write cycle, the user can configure the behavior of the data out port to one of the following choices:

- Read After Write
- Read Before Write
- No Read On Write

The write modes can be described with the help of the figure below. In Figure 2, the memory has been set to an initial value of 5 and the address bit is specified as 2. When using No Read On Write mode, the output is unaffected by the address line and the output is the same as the last output when **we** was 0. When **we** is 1, **dout** holds its previous value until **we** is 1. In Figure 2, we see **dout** reflecting the value of **addr** position 2, one cycle after **we** is set to 1.

For the other two modes, the output is obtained from the location specified by the address line, and hence is the value of the location being written to. This means that the output can be the old value which corresponds to Read After Write.

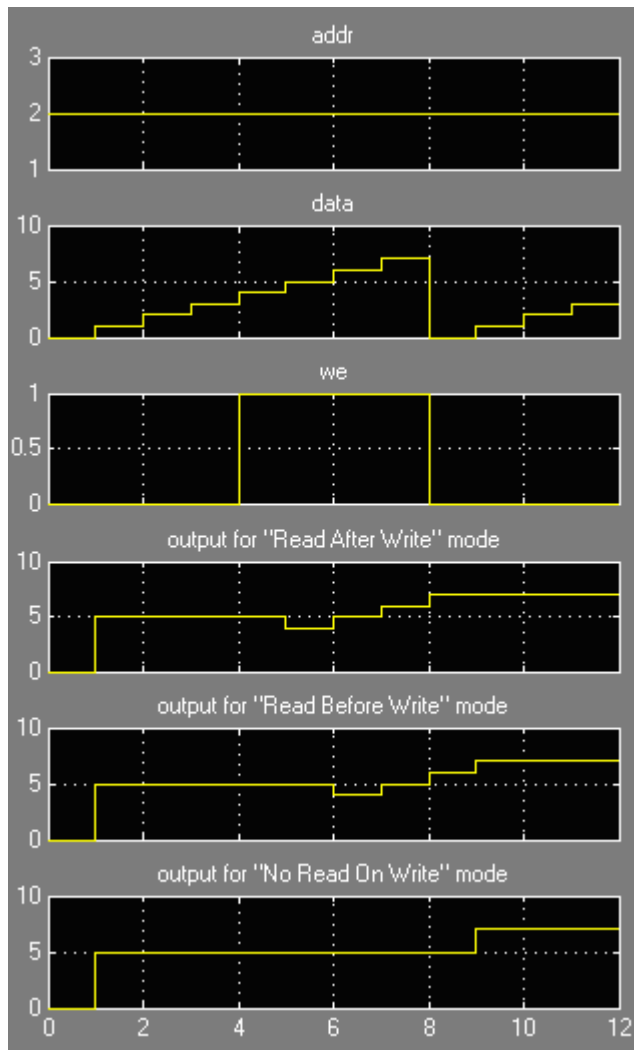


Figure 2: Illustration of write modes

Virtex, Virtex-E and Spartan-II families support only Read After Write. Virtex-II, Virtex-II Pro, Virtex-4 and Spartan-3 support all modes.

Block Parameters Dialog Box

The following figure shows the block parameters dialog box.

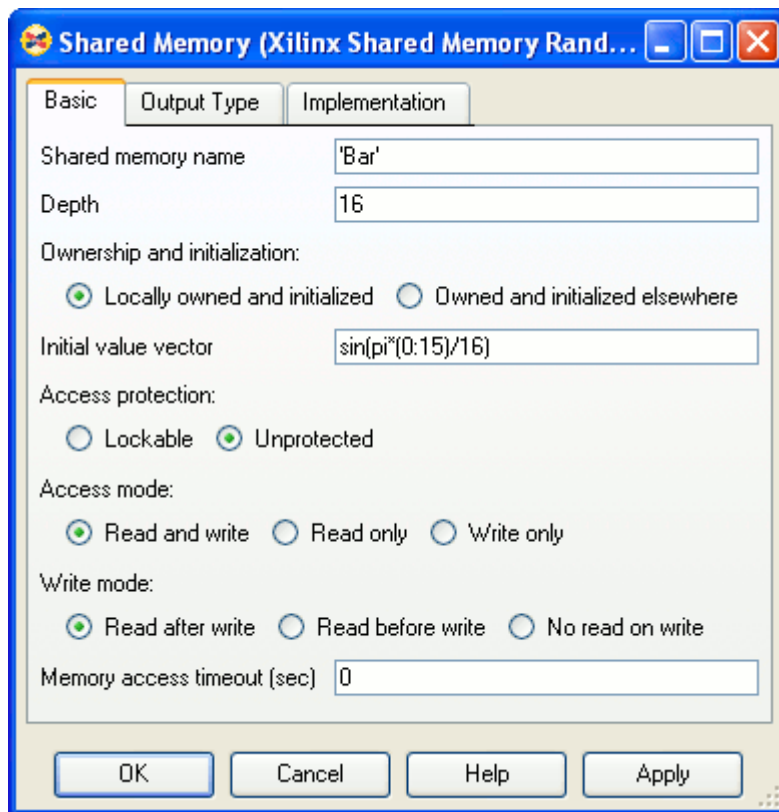


Figure 2: Shared Memory block parameters dialog box

Parameters specific to the block are:

- **Shared memory name:** name of the shared memory. All memories with the same name share the same physical memory.
- **Depth:** specifies the number of words in the memory. The word size is inferred from the bit width of the data port `din`.
- **Ownership and initialization:** indicates whether the memory is **Locally owned and initialized** or **Owned and initialized elsewhere**. If the memory is locally owned and initialized, the **Initial value vector** parameter is made available. A block that is **Locally owned and initialized** is responsible for creating an instance of the memory. A block that is **Owned and initialized elsewhere** attaches itself to a memory instance that has already been created. As a result, if two shared memory blocks are used in two different models during simulation, the model containing the **Locally owned and initialized** block has to be started first.
- **Initial value vector:** specifies initial memory contents. The size and precision of the elements of the initial value vector are inferred from the type of the data samples that drive `din`. When the vector is longer than the RAM, the vector's trailing elements are discarded. When the RAM is longer than the vector, the RAM's trailing words are set to zero. The initial value vector is saturated and rounded according to the precision specified on the data port `din`.
- **Access protection:** either **Lockable** or **Unprotected**. An unprotected memory has no restrictions concerning when a read or write can occur. In a locked shared memory, the block can only be written to when granted access to the memory. When the `grant` port outputs a 1, access is granted to the memory and the write request can proceed.
- **Access mode:** specifies the way in which the memory is used by the design. When **Read and write** mode is used, the block is configured with `din` and `dout` ports. When **Read only** mode is used, the block is configured with a `dout` port for memory read access. When **Write only** mode is used, the block is configured with a `din` port for memory write access.

- **Write mode:** specifies the memory behavior to be Read after write, Read before write, or No read on write. There are device specific restrictions on the applicability of these modes.
- **Memory access timeout (sec):** when the memory is running in hardware, this specifies the maximum time to wait for the memory to respond to a request.

Other parameters used by this block are described in the Common Parameters section at the beginning of this chapter.

Xilinx LogiCORE

The block uses the Xilinx LogiCORE Dual Port Block Memory.

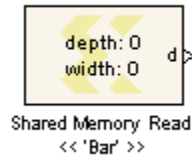
See Also

The following documents are provided as part of the System Generator User's guide and give valuable insight into using and understanding the Shared Memory block:

Multiple Subsystem Generator
Hardware Co-Simulation

Shared Memory Read

This block is listed in the following Xilinx Blockset libraries: Shared Memory and Index.



The Xilinx Shared Memory Read block provides a high-speed interface for reading data from a Xilinx shared memory object. Both FIFO and lockable shared memory objects are supported by the block. The requested data is read out of the shared memory and into a Simulink scalar, vector, or matrix signal which is written to the block's output port. The bracketed text beneath the block indicates shared memory with which this block interfaces.

The depth and width displays on the block indicate the size of the shared memory. These values are updated at runtime when the block makes the connection to the shared memory object.

The Shared Memory Read block performs several transactions with its associated shared memory object when it is woken up during a simulation. The frequency at which the block is woken up is determined by its **Sample Time** parameter. The type of transactions performed depends on whether the block is associated with a FIFO or lockable shared memory object.

FIFO Transactions

The transactions with a shared FIFO object are listed below in their order of occurrence during a simulation cycle:

- **Wait for Data:** The Shared Memory Read block waits for the number of words specified in the **Output dimensions** field to become available in the shared FIFO object. If the number of words fails to become available in the FIFO after 15 seconds, it will time out and the simulation will terminate.
- **Read Data:** Once the block ensures a sufficient number of words are available, the Shared Memory Read block will read data from the shared FIFO object.

Lockable Memory Transactions

The transactions with a lockable shared memory are listed below in their order of occurrence during a simulation cycle:

- **Acquire Lock:** Before the Shared Memory Read block may read the shared memory contents, it must acquire lock over the shared memory object. If the block fails to gain lock after 15 seconds, it will time out and the simulation will terminate.
- **Read Data:** Once lock is acquired, the Shared Memory Read block will read data from the shared memory object.
- **Release Lock:** The Shared Memory Read block releases the lock after reading data from the shared memory object.

The Shared Memory Read block is useful for simulation only and is ignored during netlisting. In particular, the Shared Memory Read block can be applied to hardware co-simulation designs with high throughput requirements. For more information on how this done, see the Real-time Signal Processing using Hardware Co-simulation section in the User Guide.

Block Parameters Dialog Box

The dialog box tabs specific to the Shared Memory Read block are shown below.

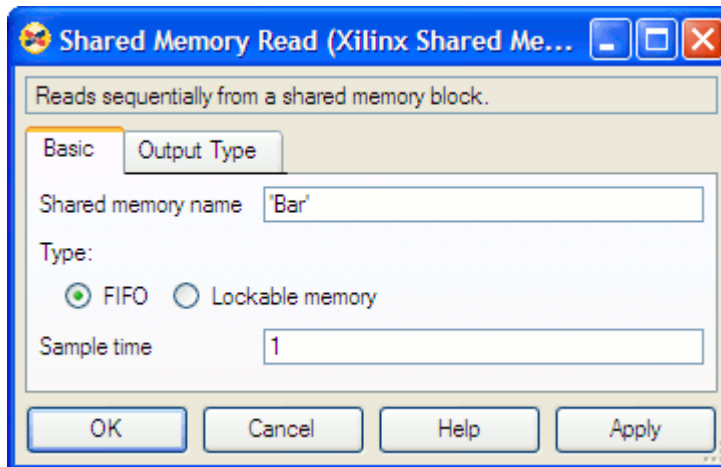


Figure 1: Shared Memory Read block Basic tab

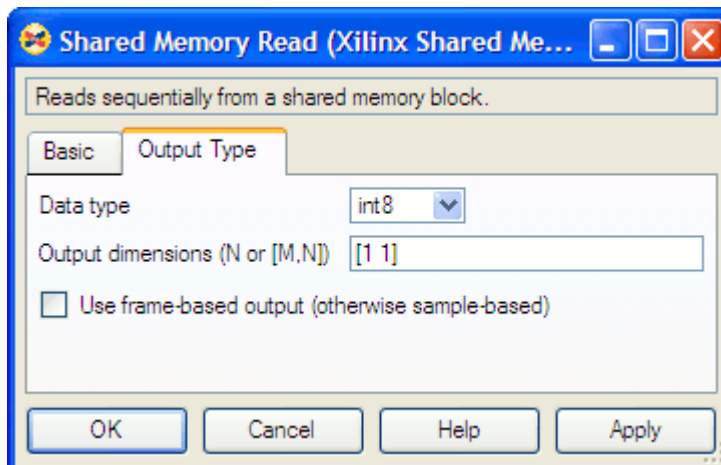


Figure 2: Shared Memory Read block Output Type tab

Parameters specific to the Shared Memory Read block are:

- **Shared memory name (Basic):** This parameter tells the unique string identifier for the Xilinx shared memory object from which data should be read. The shared memory must be a shared FIFO or lockable memory that is created and initialized elsewhere (i.e., the Shared Memory Read block does not create the specified shared memory object).
- **Type (Basic):** Tells whether the block should read from a Xilinx shared FIFO or Lockable memory object.
- **Sample time (Basic):** Specifies how often this block should read from the shared memory.
- **Data type (Output Type):** Specifies how shared memory data words should be interpreted by the Shared Memory Read block. The Simulink scalar, vector, or matrix signal that is generated will be of the chosen data type. The supported data types are `int8`, `uint8`, `int16`, `uint16`, `int32`, and `uint32`. The width of the chosen data type must match the width of the data stored in the shared memory object. For example, if the width of the shared memory data is 16 bits, then you may choose `int16` or `uint16`.
- **Output dimensions (Output Type):** Specifies how the shared memory data image should be interpreted, by giving the size of each available dimension. For a vectored output, only a single dimension (N) must be specified. For a matrix output, specify the dimensions in a two-element array `[M, N]`, where M gives the number of rows, and N gives the number of columns.

The total number of elements in the output (N , or $M*N$) must not be greater than the depth of the shared memory.

- **Use frame-based output (Output Type):** Specifies whether the output signal from the Shared Memory Read block should be represented as a frame-based signal or a sample-based signal. Frame-based signals represent consecutive sample-based signals that have been buffered together. For example, a frame-based output would be suitable for driving a Simulink Unbuffer block. Note that enabling frame-based output requires a two-dimensional output specified in the **Output Dimensions** parameter.

See Also

Shared Memory Write, Real-time Signal Processing using Hardware Co-simulation

Shared Memory Write

This block is listed in the following Xilinx Blockset libraries: Shared Memory and Index.



The Xilinx Shared Memory Write block provides a high-speed interface for writing data into a Xilinx shared memory object. Both FIFO and lockable shared memory objects are supported by the block. The Shared Memory Write block input port should be driven by the Simulink scalar, vector, or matrix signal containing the data you would like written into the shared memory object. The bracketed text beneath the block indicates the shared memory with which this block interfaces. The depth and width displays on the block indicate the size of the shared memory - these values are updated at runtime when the block makes the connection to shared memory. The width of the input data must match the width of the shared memory, and the total number of elements in the input must not be bigger than the depth of the shared memory object.

The Shared Memory Write block performs several transactions with its associated shared memory object when it is woken up during a simulation. The frequency at which the block is woken up is determined by its sample period, which is inherited from the signal driving its input port. The type of transactions performed depends on whether the block is associated with a FIFO or lockable shared memory object.

FIFO Transactions

The transactions with a shared FIFO object are listed below in their order of occurrence during a simulation cycle:

- **Wait for Available Storage:** The Shared Memory Write block waits for storage to become available in the shared FIFO object. The amount of storage depends on the size (i.e., the number of words) of the signal driving the data input port. For example, if the input signal is 256 words wide, the Shared Memory Write block waits for 256 words to become available in the shared FIFO. If the storage fails to become available after 15 seconds, it will time out and the simulation will terminate.
- **Write Data:** Once the block ensures a sufficient amount of available, the Shared Memory Write block will write data into the shared FIFO object.

Lockable Memory Transactions

- **Acquire Lock:** Before the Shared Memory Write block may write to the shared memory contents, it must acquire lock over the shared memory object. If the block fails to gain lock after 15 seconds, it will time out and the simulation will terminate.
- **Write Data:** Once lock is acquired, the Shared Memory Write block will write data to the shared memory object.
- **Release Lock:** The Shared Memory Write block releases the lock after writing data to the shared memory object.

The Shared Memory Write block is useful for simulation only and is ignored during netlisting. In particular, the Shared Memory Write block can be applied to hardware co-simulation designs with high throughput requirements. For more information on how this done, see the Real-time Signal Processing using Hardware Co-simulation section in the User Guide.

Block Parameters Dialog Box

The Shared Memory Write block can be configured using its Block Parameters dialog box.

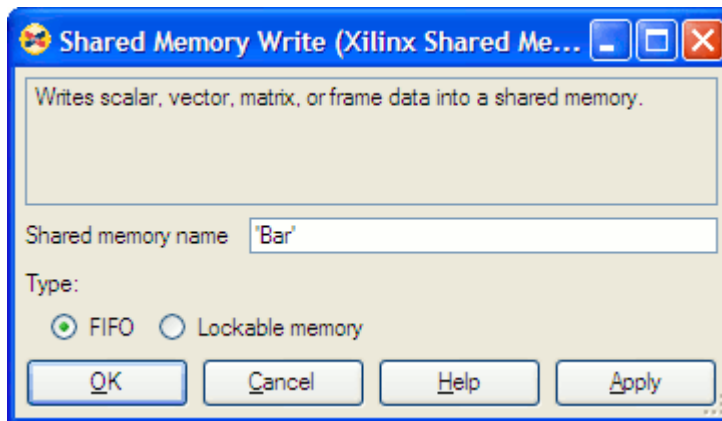


Figure 1: Shared Memory Write block parameters dialog box

Parameters specific to the Shared Memory Write block are:

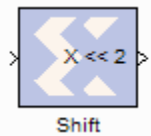
- **Shared Memory Name**: This parameter gives the unique string identifier for the shared memory to which the block should write the incoming data. The memory must be a lockable memory that is created and initialized elsewhere (i.e., the Shared Memory Write block does not create the specified shared memory object).
- **Type (Basic)**: Tells whether the block should write to a Xilinx shared FIFO or Lockable memory object.

See Also

Shared Memory Read, Real-time Signal Processing using Hardware Co-simulation

Shift

This block is listed in the following Xilinx Blockset libraries: *Control Logic, Data Types, Math, and Index.*



The Xilinx Shift block performs a left or right shift on the input signal. The result will have the same fixed-point container as that of the input.

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

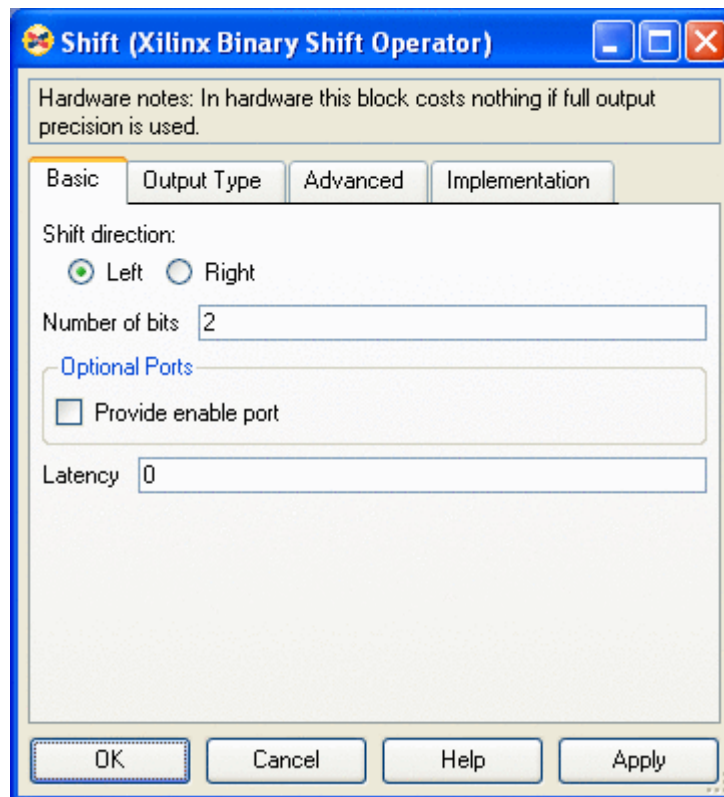


Figure 1: Shift block Basic tab

Parameters specific to the Shift block are:

- **Shift direction:** specifies a direction, Left or Right. The Right shift moves the input toward the least significant bit within its container, with appropriate sign extension. Bits shifted out of the container are discarded. The Left shift moves the input toward the most significant bit within its container with zero padding of the least significant bits. Bits shifted out of the container are discarded.
- **Number of bits:** specifies how many bits are shifted. If the number is negative, direction selected with Shift direction is reversed.

Other parameters used by this block are explained in the Common Parameters section at the beginning of this chapter.

The Shift block does not use a Xilinx LogiCORE.

Single-Step Simulation



This block is listed in the following Xilinx Blockset libraries: Tools and Index.

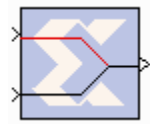
The Xilinx Single-Step Simulation block pauses the simulation each clock cycle when in single-step mode. Double-clicking on the icon switches the block from single-step to continuous mode. When the simulation is paused, it can be restarted by selecting the play button

on the model toolbar .

There are no parameters for this block.

Simulation Multiplexer

This block appears only in the Index library of the Xilinx Blockset.



Note: The Simulation Multiplexer has been deprecated in System Generator. It is expected that the block will be eliminated in a future version of the Xilinx Blockset. The functionality supplied by this block is now available through System Generator's support for Simulink's configurable subsystem capabilities. The use of configurable subsystems offers several advantages over the use of Simulation Multiplexer blocks.

The Simulation Multiplexer is a System Generator block that allows two portions of a design to work in parallel, with simulation results provided by the first portion and hardware provided by the second.

This is useful, for example, when a subsystem is defined in the usual way with Simulink blocks, but *black box* HDL is used to implement the subsystem in hardware. An example is shown below.

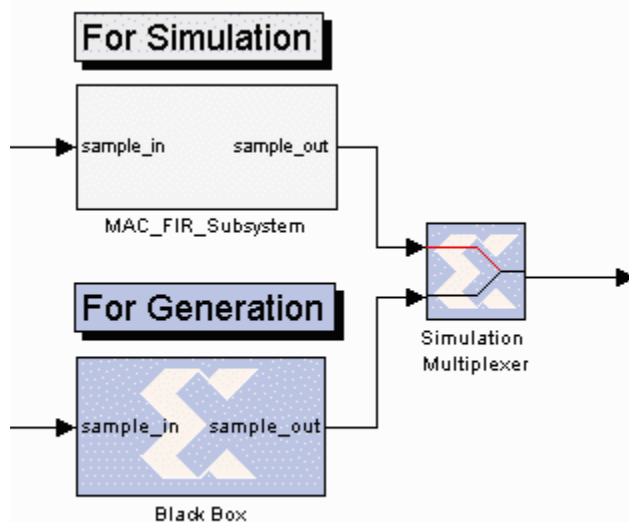


Figure 1: Example using a subsystem for simulation and black box for hardware

Using Subsystem for Simulation and Black Box for Hardware

The Simulation Multiplexer has two inputs ports. In the block parameters dialog box, one port can be identified as *For Simulation* and a second as *For Generation*. The portion of the design that drives the *For Simulation* port is used as the simulation model, and the portion that drives *For Generation* is used to produce hardware. The same port can be used for both. In this case the portion of the design that drives the combined *For Simulation/For Generation* port is used both for simulation and to produce hardware, while the other portion is ignored. It should be noted that simulation results from a design that contains a Simulation Multiplexer need not be bit and cycle accurate.

The Simulation Multiplexer is useful whenever there is a difference between what should be used for simulation and what should be used in hardware. For example, a hardware cosimulation token with an accompanying FPGA bitstream can be simulated but cannot be translated into hardware. If the HDL used to produce the bitstream is available, a black box can incorporate the HDL. Driving a Simulation Multiplexer's *For Simulation* port with the token and its *For Generation* port with the black box makes it possible both to simulate the design and to produce hardware. Another use for the multiplexer is to switch between black boxes that incorporate different types of HDL. One might provide behavioral HDL to be used in simulation, and the other might provide RTL to be used for implementation.

The Simulation Multiplexer can also be used for incremental netlisting. To do this, the design is initially built in the usual way, i.e., entirely from System Generator blocks. Once a subsystem has been netlisted, a Simulation Multiplexer is used to pair the subsystem with a black box that incorporates the files (e.g., EDIF, NGC, UCF) that implement the subsystem in hardware.

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

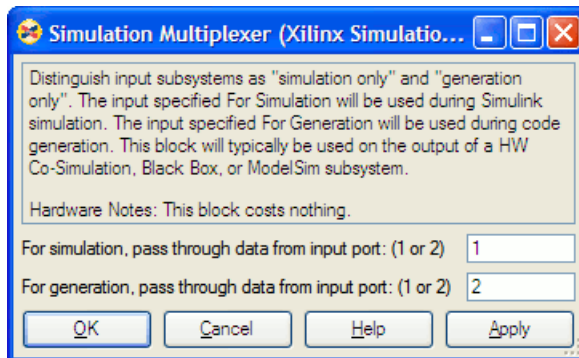


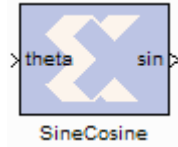
Figure 2: Simulation Multiplexer block parameters dialog box

Parameters specific to the block are:

- **For Simulation, Pass Through Data from Input Port:** Determines which input port (either 1 or 2) is used for simulation.
- **For Generation, Pass Through Data from Input Port:** Determines which input port (either 1 or 2) is used for generation.

SineCosine

This block is listed in the following Xilinx Blockset libraries: *Math and Index*.



The Xilinx Sine Cosine block computes $\sin(x)$ and/or $\cos(x)$. It stores a reference sinusoid in a read-only memory (ROM) whose depth is defined by the width of the block's single input port. An N-bit input address results in a logical ROM containing 2^N equally spaced samples of one period. (In practice, the implementation may reduce memory size by storing only a fraction of one full period.) The input signal must be an unsigned integer.

The block can produce a sine or cosine (or its negative) at one output port, or both sine and cosine (or their negatives) at two output ports, depending on parameter settings. Stepping through the memory produces sampled sinusoids on the block's output port(s), with output frequency determined by the address increment.

When non-symmetrical output is selected, output samples are in the interval

$$\left[-1, \frac{1}{2^w - 1} \right].$$

(Here w denotes the output width.) When symmetrical output is selected, output samples are in $[-1, 1]$.

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

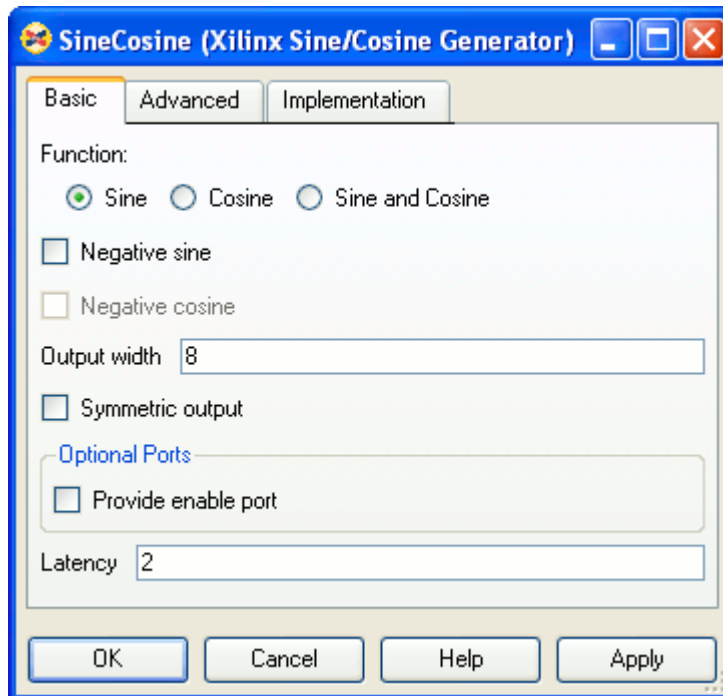


Figure 1: SineCosine block parameters dialog box

Parameters specific to this block are:

- **Function:** specifies output to be sine, cosine, or both.

- **Negative sine:** when selected, the sine output is negated.
- **Negative cosine:** when selected, the cosine output is negated.
- **Output width:** specifies the number of bits in the output. The valid range is from 4 to 32, inclusive. The output is stored as a two's complement value with one integer sign bit. As a result, the range of values stored in the table lies in the half-open interval $[-1, 1)$.
- **Symmetric output:** When selected, the output is symmetric.

Other parameters used by this block are described in the Common Parameters section at the beginning of this chapter.

Xilinx LogiCORE

This block always uses the Xilinx LogiCORE Sine/Cosine Look-Up Table. The input and output widths determine whether the ROM stores a full or quarter wave. When distributed memory is used, the ROM stores a full wave for table depths less than or equal to 64. This corresponds to one CLB per output bit. If the table depth is greater than 64, a quarter wave is stored, and additional logic is used to generate the remaining portions of the wave. Storing only the quarter wave for the large tables reduces the area needed. Block memory stores a full wave for all table depths and widths that can be implemented in a single block memory. Otherwise, values are stored as a quarter wave. Latency for the distributed ROM implementation is determined by the input width, pipelining, and the selected latency.

Input Width	Latency Range using Distributed ROM
3-6	1-2
7-8	1-4
9-10	1-5

The minimum pipeline for block ROM implementations is 1, thus the minimum latency is 1. The maximum latency for block ROM is also 1 except for the cases outlined in the table below.

Input Width	Output Width	Latency Using Block ROM
Greater than 10	Greater than 16	2
Equal to 10	Greater than 4	2
Greater than 9	Greater than 8	2

Single Port RAM

This block is listed in the following Xilinx Blockset libraries: Control Logic, Memory, and Index.



The Xilinx Single Port RAM block implements a random access memory (RAM) with one data input and one data output port.

Block Interface

The block has one output port and three input ports for address, input data, and write enable (WE). Values in a Single Port RAM are stored by word, and all words have the same arithmetic type, width, and binary point position.

A single-port RAM can be implemented using either block memory or distributed memory resources in the FPGA. Each data word is associated with exactly one address that must be an unsigned integer in the range 0 to $d-1$, where d denotes the RAM depth (number of words in the RAM). An attempt to read past the end of the memory is caught as an error in the simulation, though if a block memory implementation is chosen, it may be possible to read beyond the specified address range in hardware (with unpredictable results). The initial RAM contents can be specified through the block parameters.

The write enable signal must be **BOOL**, and when its value is 1, the data input is written to the memory location indicated by the address input. The output during a write operation depends on the choice of memory implementation.

The behavior of the output port depends on the write mode selected (see below). When the WE is 0, the output port has the value at the location specified by the address line.

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

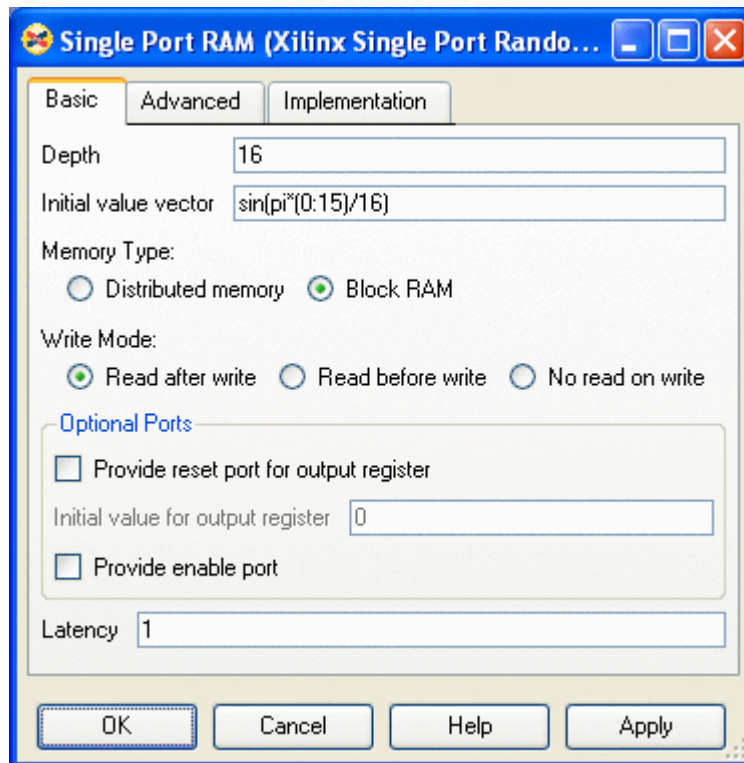


Figure 1: Single Port RAM block parameters dialog box

Parameters specific to this block are:

- **Depth:** the number of words in the memory; must be a positive integer.
- **Initial value vector:** the initial contents of the memory. When the vector length exceeds the memory depth, values with index higher than depth are ignored. When the depth exceeds the vector length, memory locations with addresses higher than the vector length are initialized to zero. Initialization values are saturated and rounded (if necessary) according to the precision specified on the data port.
- **Write Mode:** specifies memory behavior when WE is asserted. Supported modes are: Read before write, Read after write, and No read on write. Read before write indicates the output value reflects the state of the memory before the write operation. Read after write indicates the output value reflects the state of the memory after the write operation. No read on write indicates that the output value remains unchanged irrespective of change of address or state of the memory. There are device specific restrictions on the applicability of these modes. Also refer to the write modes and hardware notes section below for more information.
- **Memory Type:** option to select between block and distributed RAM.
- **Provide reset port for output register:** exposes a reset port controlling the output register of the Block RAM. Note: this port does *not* reset the memory contents to the initialization value. The reset port is available only when the latency of the Block RAM is set to 1.
- **Initial value for output register:** the initial value for output register. The initial value is saturated and rounded as necessary according to the precision specified on the data port of the Block RAM. The option to set initial value is available only for Virtex-II, Virtex-II Pro, Spartan-3, and Virtex-4 devices.

Other parameters used by this block are explained in the Common Parameters section at the beginning of this chapter.

Write Modes

During a write operation (WE asserted), the data presented to the data input is stored in memory at the location selected by the address input. The user can configure the behavior of the data out port A upon a write operation to one of the following modes:

- Read after write
- Read before write
- No read On write

These modes can be described with the help of the figure shown below. In the figure the memory has been set to an initial value of 5 and the address bit is specified as 4. When using **No read on write** mode, the output is unaffected by the address line and the output is the same as the last output when the WE was 0. For the other two modes, the output is obtained from the location specified by the address line, and hence is the value of the location being written to. This means that the output can be either the old value (**Read before write** mode), or the new value (**Read after write** mode).

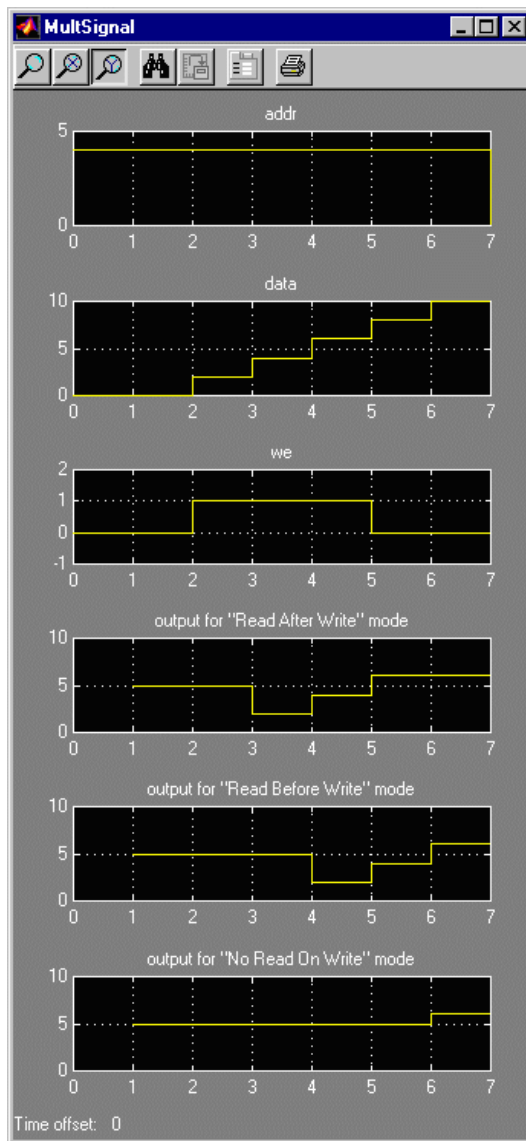


Figure 2: Illustration of write modes

Virtex, Virtex-E, and Spartan-II FPGA families support only Read After Write mode. Virtex-II, Virtex-II Pro, Spartan-3, and Virtex-4 support all modes.

Hardware Notes

The distributed memory LogiCORE supports only the Read before write mode. The Xilinx Single Port RAM block also allows distributed memory with write mode option set to Read after write when specified latency is greater than 0. The Read after write mode for the distributed memory is achieved by using extra hardware resources (a MUX at the distributed memory output to latch data during a write operation).

When implementing single port RAM blocks on Virtex-4 devices, maximum timing performance is possible if the following conditions are satisfied:

- Provide Reset Port for Output Register is unchecked
- Depth is less than 16,384

- Latency is set to 2 or higher

Xilinx LogiCORE

The block always uses a Xilinx LogiCORE: Single Port Block Memory or Distributed Memory.

For the block memory, the address width must be equal to $\text{ceil}(\log_2(d))$ where d denotes the memory depth. The maximum width of data words in the block memory depends on the depth specified; the maximum depth depends on the device family targeted. Figure 3 and Figure 4 provide the maximum data word width for a given block memory depth.

Depth	Width
2 to 2048	256
2049 to 4096	192
4097 to 8192	96
8193 to 16K	48
16K+1 to 32K	24
32K+1 to 64K	12
64K+1 to 128K	6
128K+1 to 256K	3

Figure 3 Maximum Width for Various Depth Ranges (Virtex/Virtex-E/Spartan-3)

Depth	Width
2 to 8192	256
8193 to 16K	192
16K+1 to 32K	96
32K+1 to 64K	48
64K+1 to 128K	24
128K+1 to 256K	12
256K+1 to 512K	6
512K+1 to 1024K	3

Figure 4 Width for Various Depth Ranges (Virtex-II/Virtex-II Pro/Virtex-4)

When the distributed memory parameter is selected, LogiCORE Distributed Memory is used. The depth must be between 16 and 65536, inclusive for Virtex-II, Virtex-II Pro, Spartan-3, and Virtex-4; depth must be between 16 to 4096, inclusive for the other FPGA families. The word width must be between 1 and 1024, inclusive.

Slice

This block is listed in the following Xilinx Blockset libraries: *Basic Elements, Control Logic, Data Types, and Index.*



The Xilinx Slice block allows you to *slice off* a sequence of bits from your input data and create a new data value. This value is presented as the output from the block. The output data type is unsigned with its binary point at zero.

The block provides several mechanisms by which the sequence of bits can be specified. If the input type is known at the time of parameterization, the various mechanisms do not offer any gain in functionality. If, however, a Slice block is used in a design where the input data width or binary point position are subject to change, the variety of mechanisms becomes useful. The block can be configured, for example, always to extract only the top bit of the input, or only the integral bits, or only the first three fractional bits. The following diagram illustrates how to extract all but the top 16 and bottom 8 bits of the input.

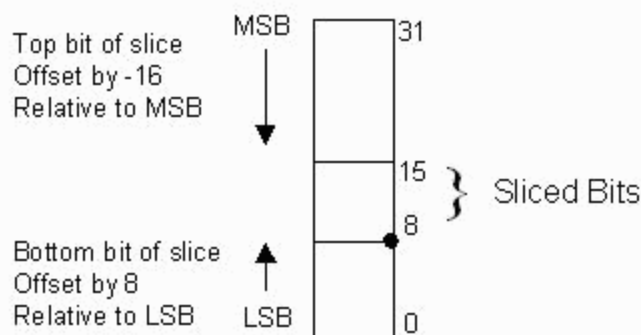


Figure 1: Slice block operation

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

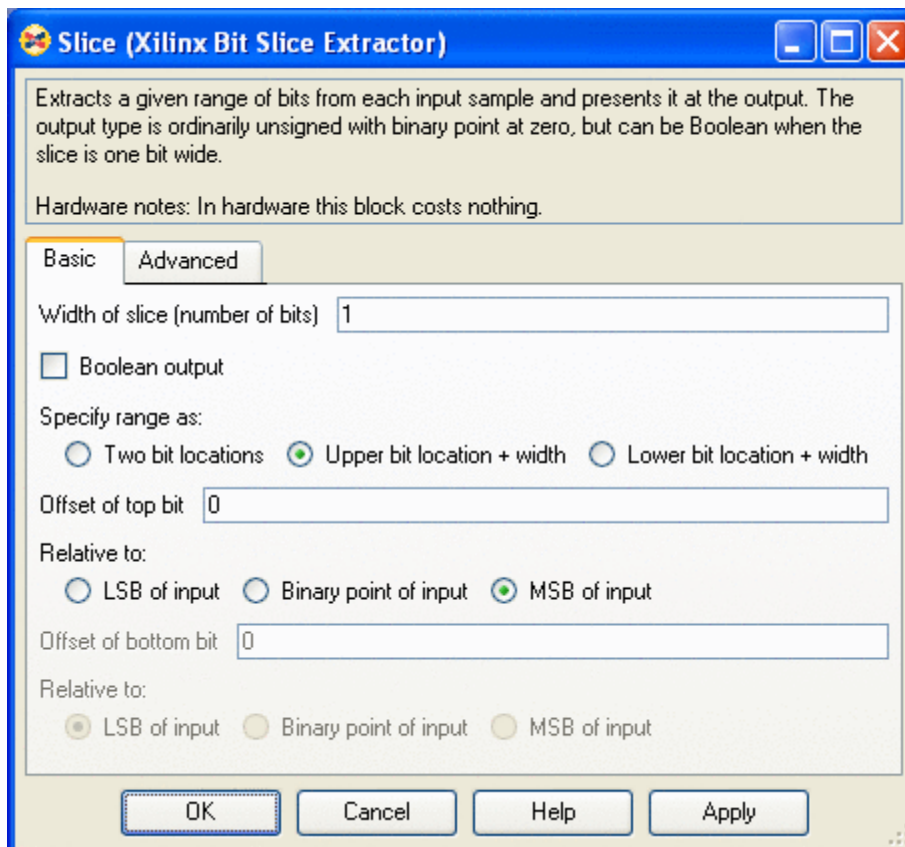


Figure 2: Slice block Basic Tab

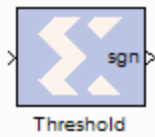
Figure 3: Slice block parameters dialog box (w/ offsets only specified)

Parameters specific to the block are:

- **Width of slice (Number of bits) (Basic):** specifies the number of bits to extract.
- **Boolean output (Basic):** Tells whether single bit slices should be type Boolean.
- **Specify range as (Basic):** (Two bit locations|Upper bit location + width|Lower bit location + width). Allows the user to specify either the bit locations of both end-points of the slice or one end-point along with number of bits to be taken in the slice.
- **Offset of top bit (Basic):** specifies the offset for the ending bit position from the LSB, MSB or binary point.
- **Offset of bottom bit:** specifies the offset for the ending bit position from the LSB, MSB or binary point.
- **Relative to (Basic):** specifies the bit slice position relative to the Most Significant Bit (MSB), Least Significant Bit (LSB), or Binary point of the top or the bottom of the slice. Other parameters used by this block are explained in the Common Parameters section at the beginning of this chapter.

Threshold

This block is listed in the following Xilinx Blockset libraries: Math and Index.



The Xilinx Threshold block tests the sign of the input number. If the input number is negative, the output of the block is -1; otherwise, the output is 1. The output is a signed fixed point integer that is 2 bits long. The block has one input and one output.

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

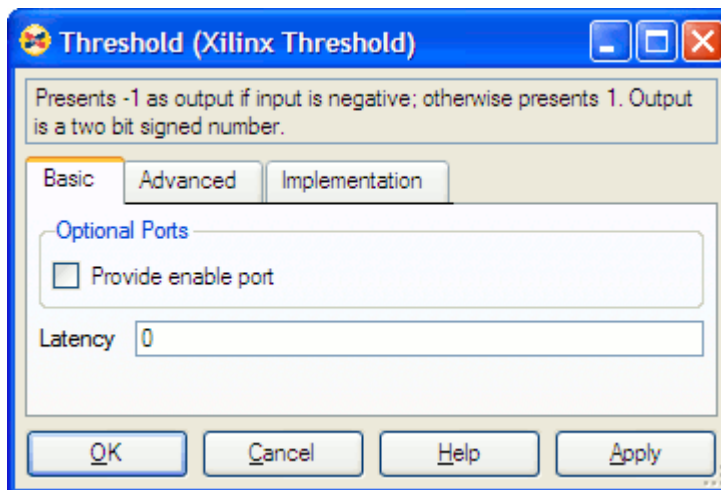


Figure 1: Threshold block parameters dialog box

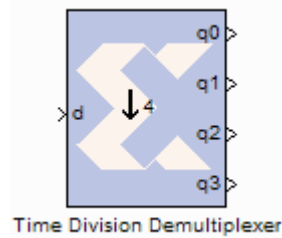
The block parameters do not control the output data type because the output is always a signed fixed point integer that is 2 bits long.

All the parameters used by this block are explained in the Common Parameters section at the beginning of this chapter.

The Threshold block does not use a Xilinx LogiCORE.

Time Division Demultiplexer

This block is listed in the following Xilinx Blockset libraries: *Basic Elements and Index*.



The Xilinx Time Division Demultiplexer block accepts input serially and presents it to multiple outputs at a slower rate.

Block Interface

The block has one data input port and a user-configurable number of data outputs, ranging from 1 to 32. The data output ports have the same arithmetic type and precision as the input data port. The time division demultiplexer block also has optional input-valid port (*vin*) and output-valid port (*vout*). Both the valid ports are of type Bool. The block has two possible implementations, single or multiple channel.

Single Channel Implementation

For single channel implementation, the time division demultiplexer block has one data input and output port. Optional data valid input and output ports are also allowed. The length of the frame sampling pattern establishes the length of the input data frame. The position of 1 indicates the input value to be downsampled and the number of 1's correspond to the downsampling factor. The behavior of the demultiplexer block in single channel mode can best be illustrated with the help of the figure below. Based on the frame sampling pattern entered, the first and second input values of every input data frame are sampled and presented to the output at the rate of 2.

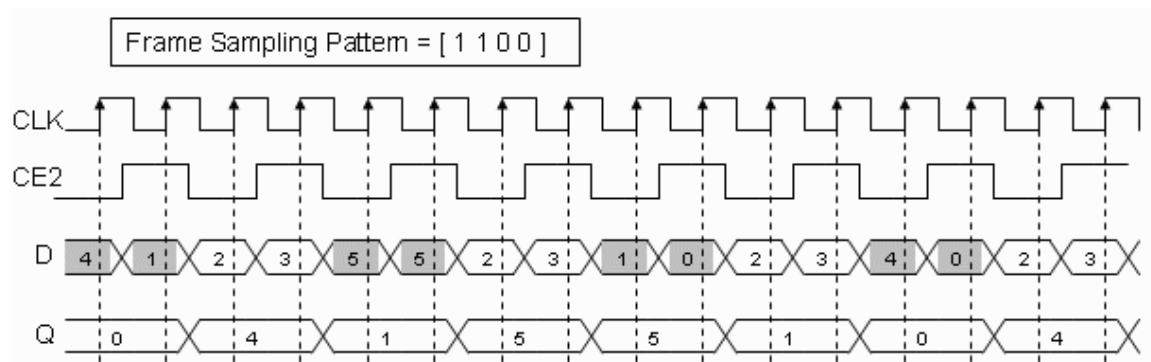


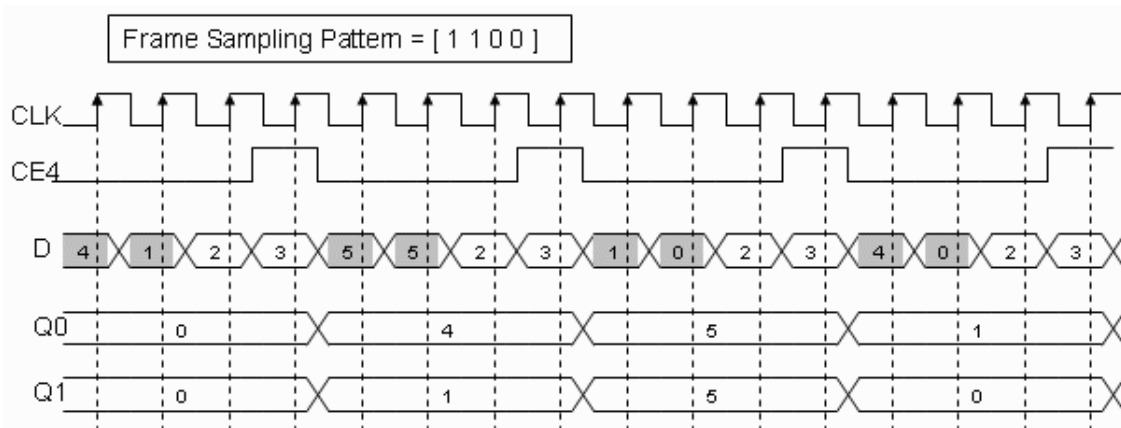
Figure 1: Timing diagram for a single channel time division demultiplexer

For single channel implementation, the number of values to be sampled from a data frame should evenly divide the size of the input frame. Every input data frame value can also be qualified by using the optional valid port.

Multiple Channel Implementation

For multiple channel implementation, the time division demultiplexer block has one data input port and multiple output ports equal to the number of 1's in the frame sampling pattern. Optional data valid input and output ports are also allowed. The length of the frame sampling pattern establishes the length of the input data frame. The position of 1 indicates the input value to be downsampled and presented to the corresponding output data channel. The behavior of the demultiplexer block in multiple channel

mode can best be illustrated with the help of the figure below. Based on the frame sampling pattern entered, the first and second input values of every input data frame are sampled and presented to the corresponding output channel at the rate of 4.



For multiple channel implementation, the down sampling factor is always equal to the size of the input frame. Every input data frame value can also be qualified by using the optional valid port.

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

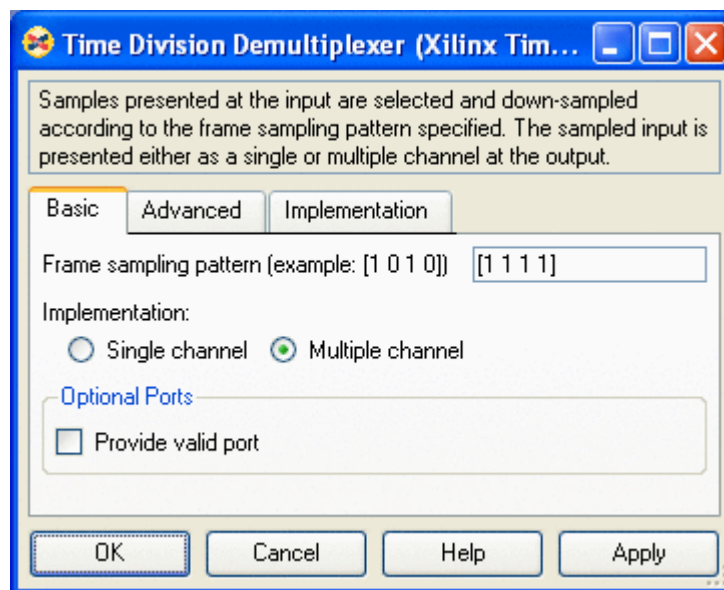


Figure 3: Time Division Demultiplexer block parameters dialog box

Parameters specific to this block are:

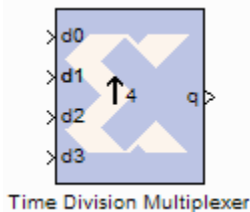
- **Frame sampling pattern:** specifies the size of the serial input data frame. The frame sampling pattern must be a MATLAB vector containing only 1's and 0's.
- **Implementation:** specifies the demultiplexer behavior to be either in single or multiple channel mode. The behaviors of these modes are explained above.

- **Provide valid Port:** when selected, the demultiplexer has optional input and output valid ports (`vin / vout`). The `vin` port allows to qualify every input data value as part of the serial input data frame. The `vout` port marks the state of the output ports as valid or not.

Other parameters used by this block are explained in the Common Parameters section at the beginning of this chapter.

Time Division Multiplexer

This block is listed in the following Xilinx Blockset libraries: *Basic Elements* and *Index*.



The Xilinx Time Divison Multiplexer block multiplexes values presented at input ports into a single faster rate output stream.

Block Interface

The block has two to 32 input ports and one output port. All input ports must have the same arithmetic type, precision, and rate. The output port has the same arithmetic type and precision as the inputs. The output rate is nr , where n is the number of input ports and r is their common rate. The block also has optional ports v_{in} and v_{out} that specify when input and output respectively are valid. Both valid ports are of type Boolean.

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

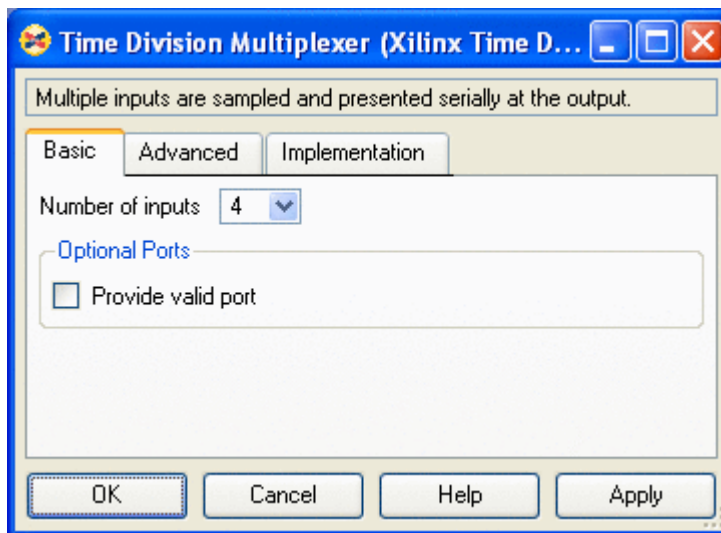


Figure 1: Block parameters dialog box

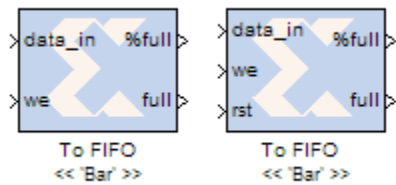
Parameters specific to this block are:

- **Number of Inputs:** specifies the number of inputs (2 to 32).
- **Provide valid Port:** when selected, the multiplexer is augmented with input and output valid ports named v_{in} and v_{out} respectively. When the v_{in} port indicates that input values are invalid, the v_{out} port indicates the corresponding output frame is invalid

Other parameters used by this block are explained in the Common Parameters section at the beginning of this chapter.

To FIFO

This block is listed in the following Xilinx Blockset libraries: *Index*.



The Xilinx To FIFO block implements the leading half of a first-in-first-out memory queue.

Values presented at the module's data port are written to the next available empty memory location when `we` input is one. The `full` output port is asserted when the FIFO is full. The percent full output port indicates the percentage of the FIFO that is full, represented with user-specified

precision.

The To FIFO is implemented in hardware using the FIFO Generator v2.1 core. System Generator's hardware co-simulation interfaces allow the To FIFO block to be compiled and co-simulated in FPGA hardware. When used in System Generator co-simulation hardware, shared FIFOs facilitate high-speed transfers between the host PC and FPGA, and bolster the tool's real-time hardware co-simulation capabilities.

Block Parameters Dialog Box

The following figure shows the block parameters dialog box.

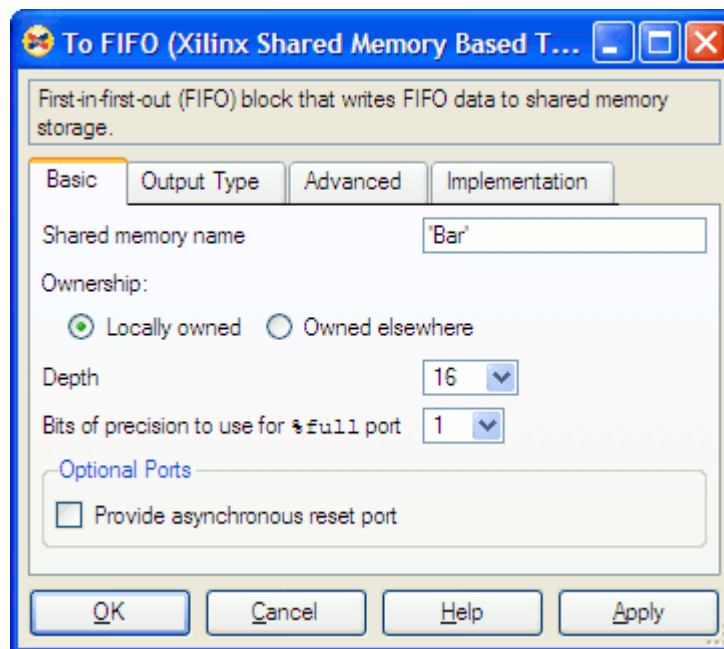


Figure 1: To FIFO block parameters dialog box

Parameters specific to the block are:

- **Shared memory name** : name of the shared FIFO. All FIFOs with the same name will share the same physical FIFO.
- **Ownership**: indicates whether the memory is **Locally owned** or **Owned elsewhere**. A block that is **Locally owned** is responsible for creating an instance of the FIFO. A block that is **Owned elsewhere** attaches itself to a FIFO instance that has already been created.

- **Depth** : specifies the number of words in the memory. The word size is inferred from the bit width of the port **din**.
- **Bits of precision to use for %full port**: specifies the bit width of the %full port. The binary point for this unsigned output is always at the top of the word. Thus, for example, if precision is set to one, the output can take two values: 0.0 and 0.5, the latter indicating the FIFO is at least 50% full.
- **Provide asynchronous reset port**: Activates an optional asynchronous reset (**rst**) port on the block.

Other parameters used by this block are described in the Common Parameters section at the beginning of this chapter.

Xilinx LogiCORE

The block uses the Xilinx LogiCORE FIFO Generator.

See Also

The following documents are provided as part of the System Generator User's guide and give valuable insight into using and understanding the To FIFO block:

From FIFO
Multiple Subsystem Generator
Hardware Co-Simulation

To Register

This block is listed in the following Xilinx Blockset libraries: *Index*.



The Xilinx To Register block implements the leading half of a D flip-flop based register, having latency of one sample period. The register can be shared among multiple designs or sections of a design.

The block has two input ports. The **din** port accepts input data and sets the bit width of the register. The initial output value is specified by the user in the block parameters dialog box (below). When the enable port **en** is asserted, data presented at the input appears at the output **dout** after one sample period. When **en** is not asserted, the last value written to the register is presented to the output port **dout**.

Block Parameters Dialog Box

The following figure shows the block parameters dialog box.

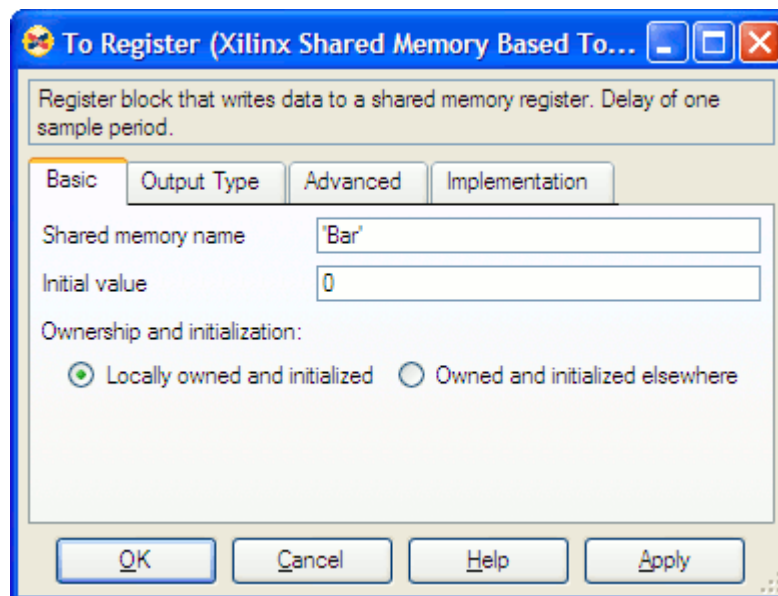


Figure 1: To Register block dialog box

Parameters specific to the block are:

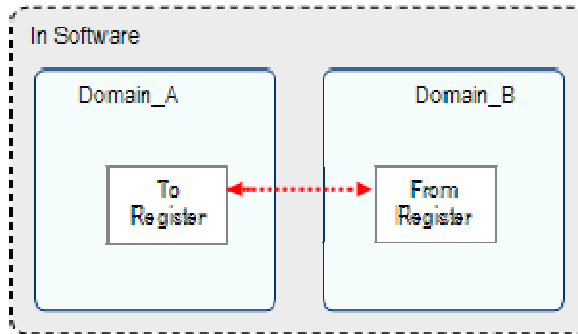
- **Shared memory name:** name of the shared register. There must be exactly one To Register block for a particular physical register. In addition, the shared memory name must be distinct from all other shared memory names in the design.
- **Initial value:** specifies the initial value in the register.
- **Ownership and initialization:** indicates whether the register is **Locally owned and initialized** or **Owned and initialized elsewhere**. A block that is locally owned is responsible for creating an instance of the register. A block that is owned elsewhere attaches itself to a register instance that has already been created. As a result, if two shared register blocks are used in two different models during simulation, the model containing the locally owned block has to be started first.

Other parameters used by this block are described in the Common Parameters section at the beginning of this chapter.

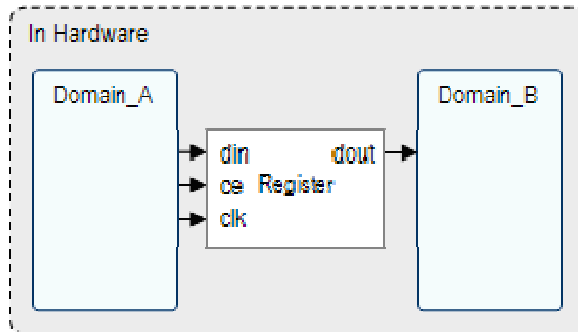
The To Register block is implemented as a synthesizable VHDL module. It does not use a Xilinx LogiCORE.

Crossing Clock Domains

When a To Register and From Register block pair are used to cross clock domain boundaries, a single register is implemented in hardware. This register is clocked by the To Register block clock domain. For example, assume a design has two clock domains, `Domain_A` and `Domain_B`. Also assume that a shared register pair are used to cross the two clock domains shown below.



When the design is generated using the Multiple Subsystem Generator block, only one register is included in the design. The clock and clock enable register signals are driven from the `Domain_A` domain.



Crossing domains in this manner may be unsafe. To reduce the chance of metastability, include two Register blocks immediately following the From Register block to resynchronize the data to the From Register's clock domain.

See Also

The following documents are provided as part of the System Generator User's guide and give valuable insight into using and understanding the To Register block:

From Register
Multiple Subsystem Generator
Hardware Co-Simulation

Toolbar

This block is listed in the following Xilinx Blockset libraries: *Index, Tools*.



The Xilinx Toolbar block provides quick access to several useful utilities in System Generator. The Toolbar simplifies the use of the zoom feature in Simulink and adds new auto layout and route capabilities to Simulink models. The Toolbar also houses several productivity improvement tools described below.

Block Interface

Double clicking on the Xilinx Toolbar block launches the GUI shown in Figure 1. The Toolbar can also be launched from the command line via `xLTBUtils`, a collection of functions used by the Toolbar.

```
xLTBUtils('Toolbar');
```

Only one Toolbar GUI can be opened at a time, that is, the Toolbar GUI is a singleton. Regardless of where a Toolbar block is placed, the Toolbar will always perform actions on the current Simulink model in focus. In other words, if the Toolbar is invoked from model A, it can still be used on model B so long as model B is in focus.

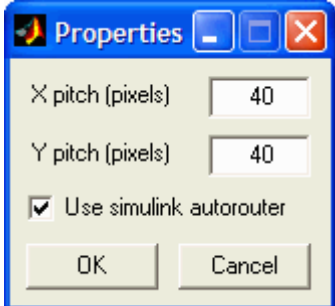


Figure 1: Toolbar GUI

Toolbar Buttons	Descriptions
	Undo: Cancels the most recent change applied to the model layout by the Toolbar and reverts the layout state to the one prior to this change. Can undo up to three changes.
	Reroute: Reroutes lines to enhance model readability. If lines are selected, only those lines will be rerouted. Otherwise all lines in the model will be rerouted.
	Auto Layout: Relocates blocks and reroutes lines to enhance model readability.
	Add Terms: Calls on the <code>xLAddTerms</code> function to add sources and sinks to the current model in focus. System Generator blocks are sourced with a System Generator constant block, while Simulink blocks are sourced with a Simulink constant block. Terminators are used as sinks.

	Help: Opens this document.
	Zoom: Allows the user to get either a closer view of a portion of the Simulink model or a wider view of the model depending on the position of the slider or the value of the zoom factor. The user can either position the slider or edit the Zoom Factor. The Zoom Factor is limited to be between 5 and 1000.

Toolbar Menus

Toolbar Menus	Description
Tools	
Create Plugins	Launches the System Generator Board Description Builder tool.
Inspect Selected	Opens up the Simulink Inspector with the properties of the blocks that are currently selected. This is useful when trying to set the size of several blocks, or the horizontal position of blocks drawn on a model.
Toolbar Properties	Launches the Properties Dialog Box shown in Figure 2. Allows the user to set parameters for the Auto Layout and Reroute tool. X and Y pitch indicate distance (pixels) between blocks placed next to each other in the X and Y directions respectively. The toolbar uses the Simulink autorouter when “Use simulink autorouter” is checked. Otherwise, a direct line is drawn from source to destination.  Figure 2: Properties Dialog Box
Help	Opens this document.

References

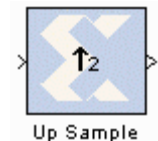
- 1) E.R.Gansner, E.Koutsofios, S.C.North, KVo, “A Technique for Drawing Directed Graphs”, <http://www.graphviz.org/Documentation/TSE93.pdf>
- 2) The Reroute and Auto Layout buttons invoke an open source package called Graphviz. More information on this package is also available at <http://www.graphviz.org/>

See Also

xlAddTerms, xlSBDBuilder, xlTBUtils

Up Sample

This block is listed in the following Xilinx Blockset libraries: *Basic Elements and Index*.



The Xilinx Up Sample block increases the sample rate at the point where the block is placed in your design. The output sample period is l/n , where l is the input sample period and n is the sampling rate. The input signal is up sampled so that within an input sample frame, an input sample is either presented at the output n times if samples are copied, or presented once with $(n-1)$ zeroes interspersed if zero padding is used.

In hardware, the Up Sample block has two possible implementations. If the **Copy Samples** option is selected on the block parameters dialog box, the **Din** port is connected directly to **Dout** and no hardware is expended. Alternatively, if zero padding is selected, a mux is used to switch between the input sample and inserted zeros. The corresponding circuit for the zero padding Up Sample block is shown in Figure 1.

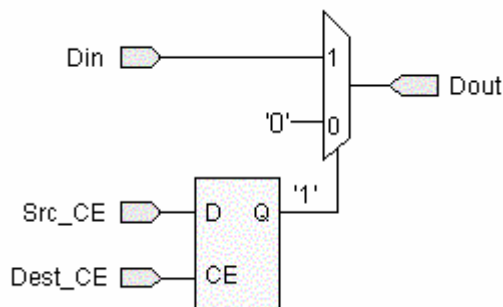


Figure 1: Up Sample with zero padding hardware implementation

Block Interface

The Up Sample block receives two clock enable signals, **Src_CE** and **Dest_CE**. **Src_CE** is the clock enable signal corresponding to the input data stream rate. **Dest_CE** is the faster clock enable, corresponding to the output data stream rate. Notice that the circuit uses a single flip-flop in addition to the mux. The flip-flop is used to adjust the timing of **Src_CE**, so that the mux switches to the data input sample at the start of the input sample period, and switches to the constant zero after the first input sample. It is important to notice that the circuit has a combinational path from **Din** to **Dout**. As a result, an Up Sample block configured to zero pad should be followed by a register whenever possible.

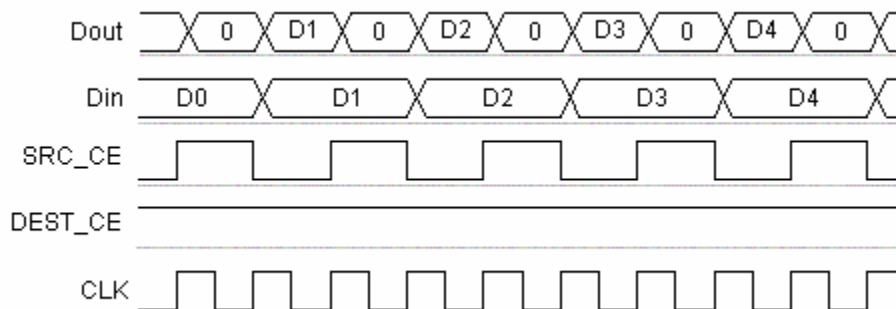


Figure 2: Example of up sample block behavior with zero padding

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

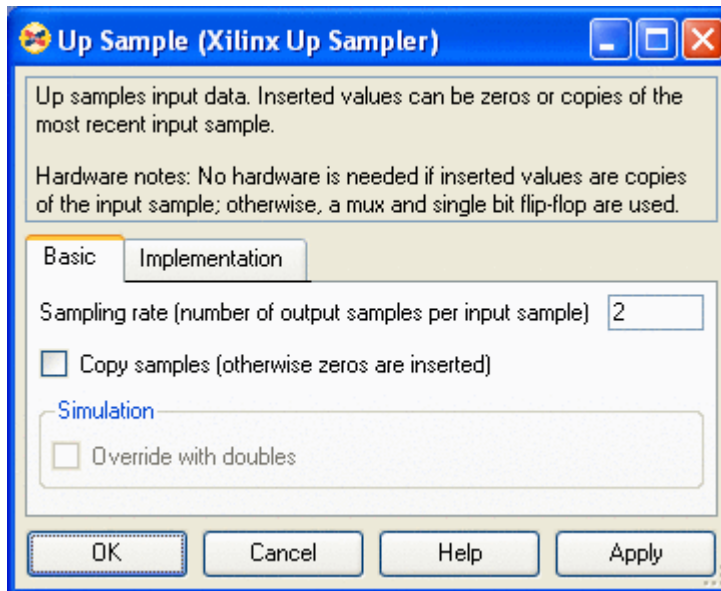


Figure 3: Up Sample block Basic Tab

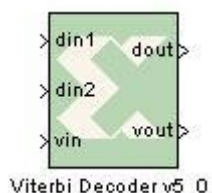
Parameters specific to the block are:

- **Sampling rate (number of output samples per input sample)** (Basic): must be an integer with a value of 2 or greater. This is the ratio of the output sample period to the input, and is essentially a sample rate multiplier. For example, a ratio of 2 indicates a doubling of the input sample rate. If a non-integer ratio is desired, the Up Sample block can be used in combination with the Down Sample block.
- **Copy samples (otherwise zeros are inserted)** (Basic): allows you to choose what to do with the additional samples produced by the increased clock rate. By selecting Copy Samples, the same sample will be duplicated (copied) during the extra sample times. If this checkbox is not selected, the additional samples are zero.

Other parameters used by this block are explained in the Common Parameters section at the beginning of this chapter.

Viterbi Decoder v5_0

This block is listed in the following Xilinx Blockset libraries: *Communication and Index*.



Data encoded with a convolution encoder may be decoded using the Xilinx Viterbi decoder block. There are two steps to the decode process. The first weighs the cost of incoming data against all possible data input combinations; either a Hamming or Euclidean metric may be used to determine the cost. The second step traces back through the trellis and determines the optimal path. The length of the trace through the trellis can be controlled by the traceback length parameter

The decoder achieves minimal error rates when using optimal convolution codes; the table below shows various optimal codes. For correct operation, convolution codes used for encoding must match with that for decoding.

Constraint length	Optimal convolution codes for 1/2 rate (octal)	Optimal convolution codes for 1/3 rate (octal)
3	[7 5]	[7 7 5]
4	[17 13]	[17 13 15]
5	[37 33]	[37 33 25]
6	[57 65]	[57 65 71]
7	[117 127]	[117 127 155]
8	[357 233]	[357 233 251]
9	[755 633]	[755 633 447]

Figure 1: Optimal Convolution Codes

Block Interface

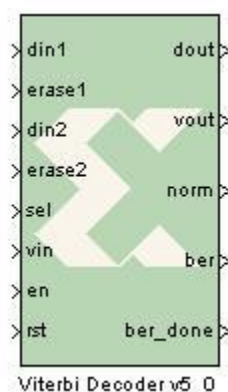


Figure 2: Rate 1/2 Viterbi Decoder block showing optional ports

The Viterbi decoder supports rates from 1/2 to 1/7 and consequently displays two to seven input ports labeled `din1` through `din7`. Hard coding requires each data input to be 1 bit wide. Soft coding allows widths to be between 3 to 8 bits (inclusive). The `vin` port indicates that the values presented on the `din` ports are valid. When using external puncturing, depending on the decoder rate, up to seven erase ports become available. If an erase pin is high, the corresponding data pins are treated as a null-symbol. For a given constraint length and traceback-length the block can function as a dual decoder, i.e. two convolution codes and two output rates. An input port labeled `sel` indicates the convolution

code to which the input data corresponds, when `sel` is 0 (respectively, 1) the data is decoded using convolution code array 1 (respectively, 2).

The Viterbi Decoder can have two to five output ports. The `dout` port outputs the 1 bit decoded result and `vout` indicates that the value is valid. The `ber` port gives a measurement of the bit error rate of the channel by counting the differences between the re-encoded `dout` and the delayed `din` values. The number of errors detected is divided by 8 and output on the `ber` port. The `ber_done` port indicates when the number of input samples for error count (as indicated on the mask) have been processed. The `norm` signal indicates when normalization has occurred within the block. The `norm` port gives immediate monitoring of errors on the channel. The more frequent the normalization (i.e. the `norm` port going high), the higher the rate of errors present.

Block Parameters Dialog Box

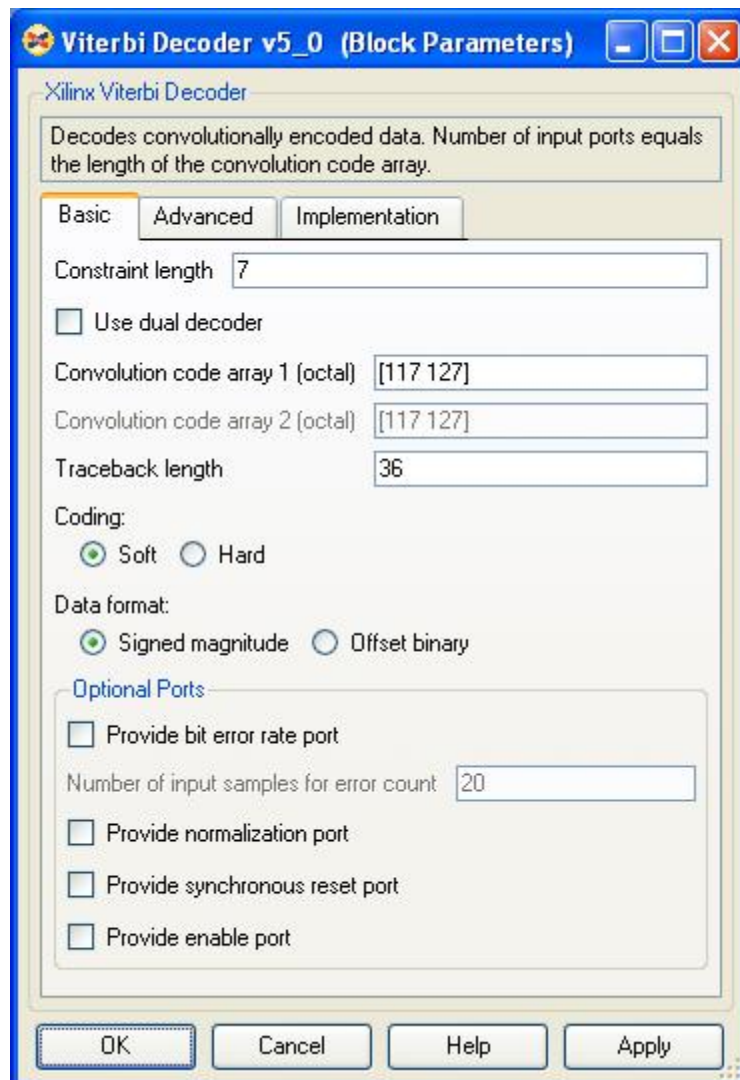


Figure 3: Viterbi Decoder basic tab parameters dialog box

Parameters specific to the Viterbi Decoder basic tab are:

- **Constraint length:** Equals $n+1$, where n is the length of the constraint register in the encoder.

- **Use dual decoder:** When selected, the block behaves as a dual decoder. This makes the `sel` input port available.
- **Convolution code array 1 (octal):** First array of octal convolution codes. Output rate is derived from the array length. Between 2 and 7 (inclusive) codes may be entered. When dual decoding is used, a value of 0 on the `sel` port corresponds to this array.
- **Convolution code array 2 (octal):** Second array of octal convolution codes. Output rate is derived from the array length. Between 2 and 7 (inclusive) codes may be entered. When dual decoding is used, a value of 1 on the `sel` port corresponds to this array. Output rate implied by convolution code array 2 does not need to be the same as that implied from convolution code array 1.
- **Traceback length:** Length of the traceback through the Viterbi trellis. Optimal length is 5 to 7 times the constraint length.
- **Coding: Hard or Soft:** **Hard** (respectively, **soft**) encoding uses the Hamming (resp., Euclidean) metric to calculate costs. **Hard** coding requires the input data be 1 bit wide, for **soft** coding, widths are between 3 and 8 bits. Soft coding is required for dual decoding, external puncturing, and serial architecture.
- **Data format:** **Signed magnitude** or **Offset binary** (available for soft coding only).
- **Provide bit error rate port:** When selected `ber` and `ber_done` ports are added to the block.
- **Number of input samples for error count:** Indicates the number of input samples over which the bit error rate is calculated.
- **Provide normalization port:** When selected the `norm` port is added to the block.

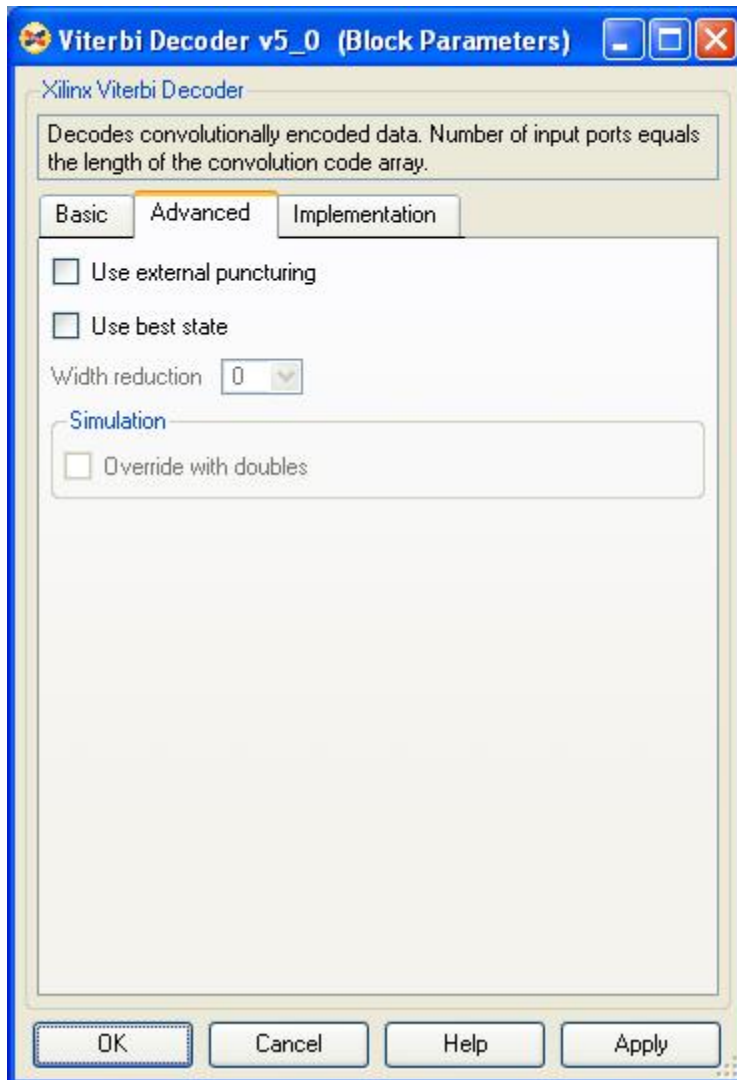


Figure 4: Viterbi Decoder advanced tab parameters dialog box

Parameters specific to the Viterbi Decoder advanced tab are:

- **Use external puncturing:** When selected erase ports are added to the block.
- **Use best state:** When selected the traceback starts from the optimal state.
- **Width reduction:** Indicates how many of the least significant bits to ignore when saving the cost used to determine the best state.

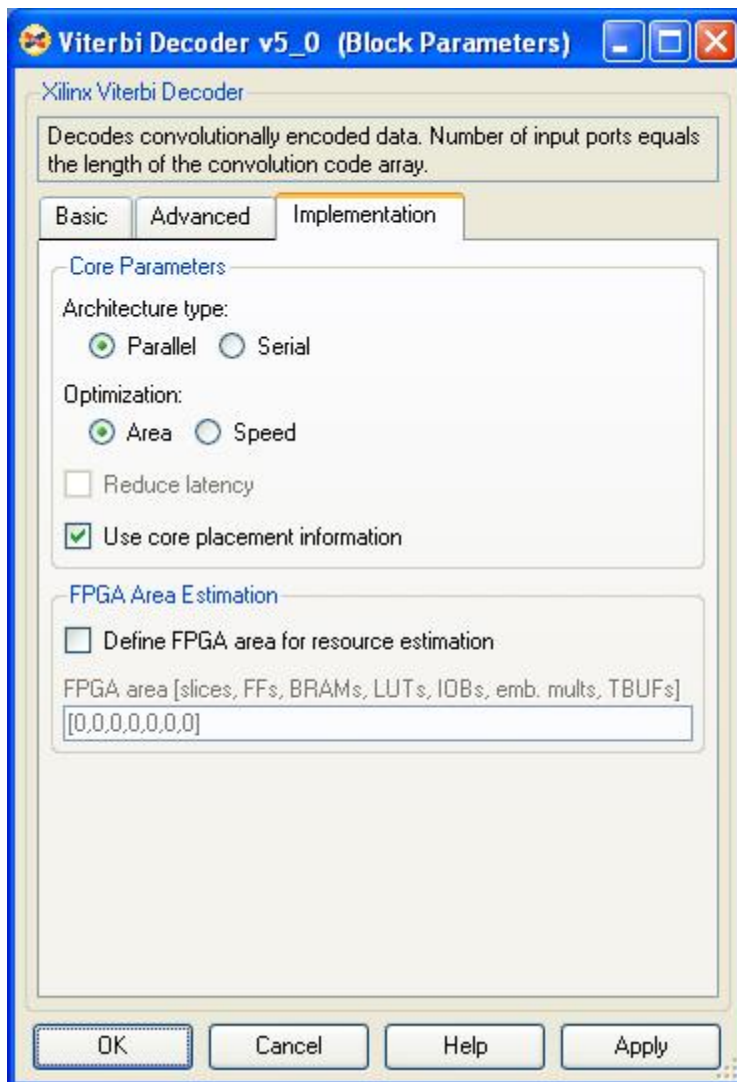


Figure 5: Viterbi Decoder advanced tab parameters dialog box

Parameters specific to the Viterbi Decoder implementation tab are:

- **Architecture type:** Parallel or Serial.
- **Optimization:** Area or Speed (available for parallel architecture only).
- **Reduce latency:** When selected the block latency is reduced by approximately 50%.

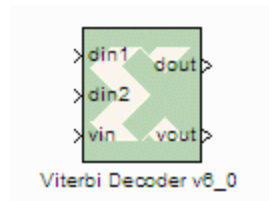
Other parameters used by this block are described in the Common Parameters section at the beginning of this chapter.

Xilinx LogiCore

This block uses the Xilinx LogiCORE Viterbi Decoder.

Viterbi Decoder v6_0

This block is listed in the following Xilinx Blockset libraries: *Communication and Index*.



Data encoded with a convolution encoder may be decoded using the Xilinx Viterbi decoder block. There are two steps to the decode process. The first weighs the cost of incoming data against all possible data input combinations; either a Hamming or Euclidean metric may be used to determine the cost. The second step traces back through the trellis and determines the optimal path. The length of the trace through the trellis can be controlled by the traceback length parameter

The decoder achieves minimal error rates when using optimal convolution codes; the table below shows various optimal codes. For correct operation, convolution codes used for encoding must match with that for decoding.

Constraint length	Optimal convolution codes for 1/2 rate (octal)	Optimal convolution codes for 1/3 rate (octal)
3	[7 5]	[7 7 5]
4	[17 13]	[17 13 15]
5	[37 33]	[37 33 25]
6	[57 65]	[57 65 71]
7	[117 127]	[117 127 155]
8	[357 233]	[357 233 251]
9	[755 633]	[755 633 447]

Figure 1: Optimal Convolution Codes

Block Interface



Figure 2: Rate 1/2 Viterbi Decoder block showing optional ports

The Viterbi decoder supports rates from 1/2 to 1/7 and consequently displays two to seven input ports labeled `din1` through `din7`. Hard coding requires each data input to be 1 bit wide. Soft coding allows widths to be between 3 to 8 bits (inclusive). The `vin` port indicates that the values presented on the `din` ports are valid. When using external puncturing, depending on the decoder rate, up to seven erase ports become available. If an erase pin is high, the corresponding data pins are treated as a null-symbol. For a given constraint length and traceback-length the block can function as a dual decoder, i.e. two convolution codes and two output rates. An input port labeled `sel` indicates the convolution code to which the input data corresponds, when `sel` is 0 (respectively, 1) the data is decoded using

convolution code array 1 (respectively, 2).

The Viterbi Decoder can have two to five output ports. The `dout` port outputs the 1 bit decoded result and `vout` indicates that the value is valid. The `ber` port gives a measurement of the bit error rate of the channel by counting the differences between the re-encoded `dout` and the delayed `din` values. The number of errors detected is divided by 8 and output on the `ber` port. The `ber_done` port indicates when the number of input samples for error count (as indicated on the mask) have been processed. The `norm` signal indicates when normalization has occurred within the block. The `norm` port gives immediate monitoring of errors on the channel. The more frequent the normalization (i.e. the `norm` port going high), the higher the rate of errors present.

Block Parameters Dialog Box

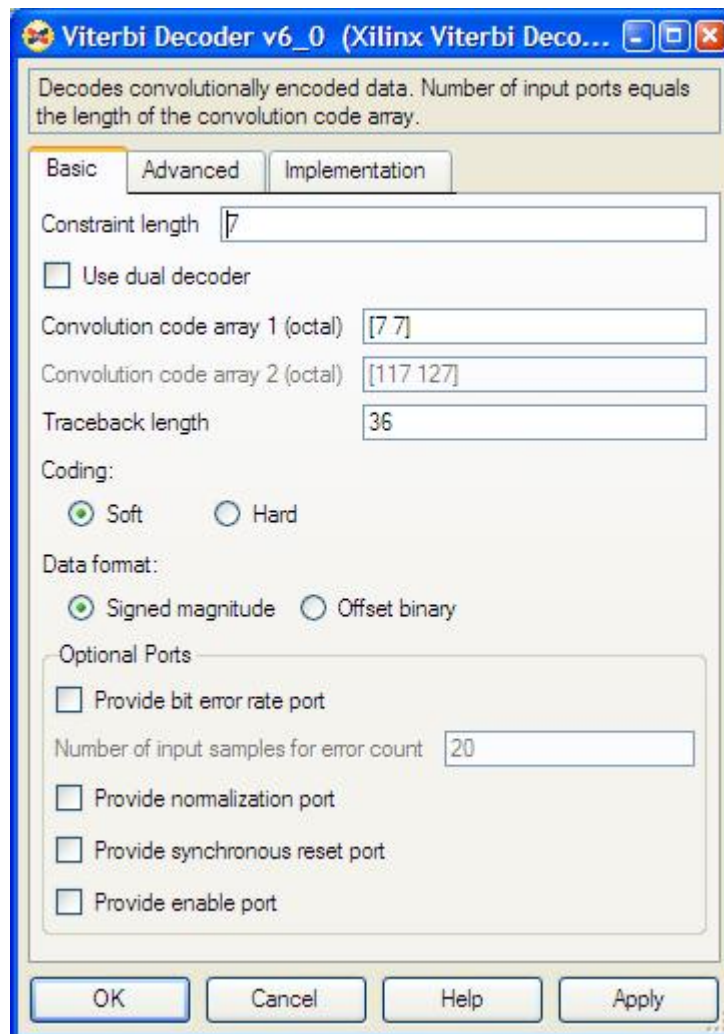


Figure 3: Viterbi Decoder basic tab parameters dialog box

Parameters specific to the Viterbi Decoder basic tab are:

- **Constraint length:** Equals $n+1$, where n is the length of the constraint register in the encoder.
- **Use dual decoder:** When selected, the block behaves as a dual decoder. This makes the `sel` input port available.

- **Convolution code array 1 (octal):** First array of octal convolution codes. Output rate is derived from the array length. Between 2 and 7 (inclusive) codes may be entered. When dual decoding is used, a value of 0 on the `sel` port corresponds to this array.
- **Convolution code array 2 (octal):** Second array of octal convolution codes. Output rate is derived from the array length. Between 2 and 7 (inclusive) codes may be entered. When dual decoding is used, a value of 1 on the `sel` port corresponds to this array. Output rate implied by convolution code array 2 does not need to be the same as that implied from convolution code array 1.
- **Traceback length:** Length of the traceback through the Viterbi trellis. Optimal length is 5 to 7 times the constraint length.
- **Coding: Hard or Soft:** **Hard** (respectively, **soft**) encoding uses the Hamming (resp., Euclidean) metric to calculate costs. **Hard** coding requires the input data be 1 bit wide, for **soft** coding, widths are between 3 and 8 bits. **Soft** coding is required for dual decoding, external puncturing, and serial architecture.
- **Data format: Signed magnitude or Offset binary** (available for soft coding only).
- **Provide bit error rate port:** When selected `ber` and `ber_done` ports are added to the block.
- **Number of input samples for error count:** Indicates the number of input samples over which the bit error rate is calculated.
- **Provide normalization port:** When selected the `norm` port is added to the block.

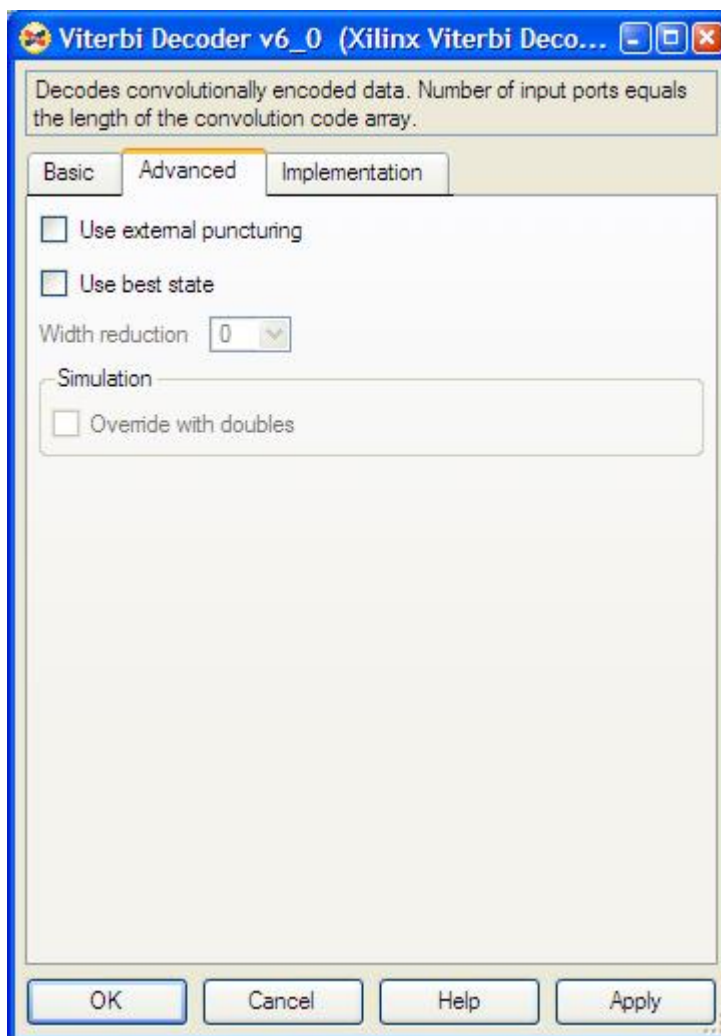


Figure 4: Viterbi Decoder advanced tab parameters dialog box

Parameters specific to the Viterbi Decoder advanced tab are:

- **Use external puncturing:** When selected erase ports are added to the block.
- **Use best state:** When selected the traceback starts from the optimal state.
- **Width reduction:** Indicates how many of the least significant bits to ignore when saving the cost used to determine the best state.

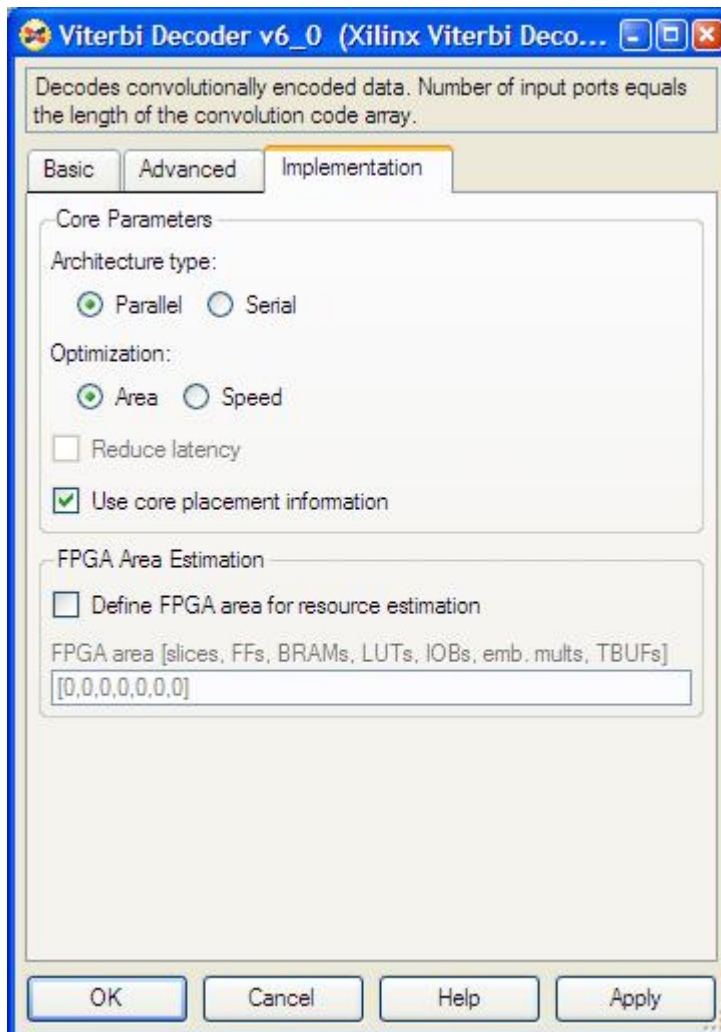


Figure 5: Viterbi Decoder advanced tab parameters dialog box

Parameters specific to the Viterbi Decoder implementation tab are:

- **Architecture type:** Parallel or Serial.
- **Optimization:** Area or Speed (available for parallel architecture only).
- **Reduce latency:** When selected the block latency is reduced by approximately 50%.

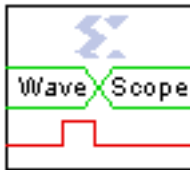
Other parameters used by this block are described in the Common Parameters section at the beginning of this chapter.

Xilinx LogiCore

This block uses the Xilinx LogiCORE Viterbi Decoder.

WaveScope

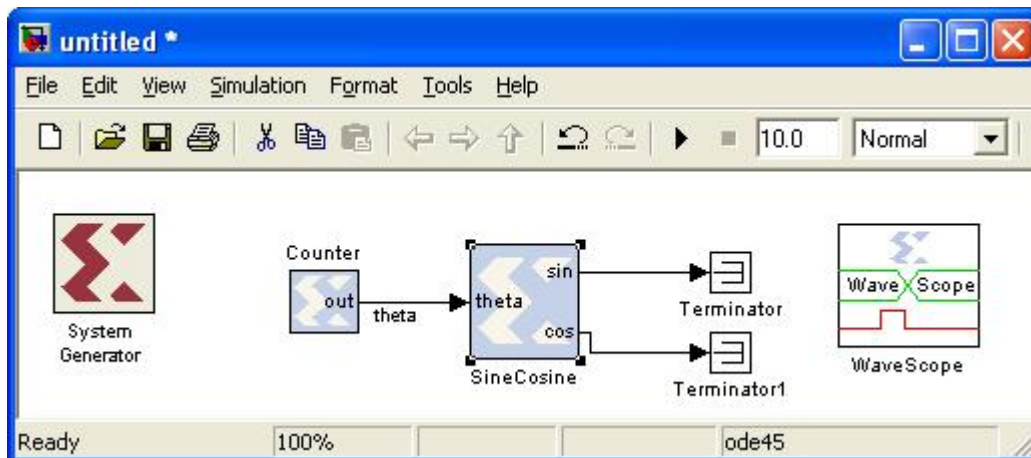
This block is listed in the following Xilinx Blockset libraries: *Index, Tools*.



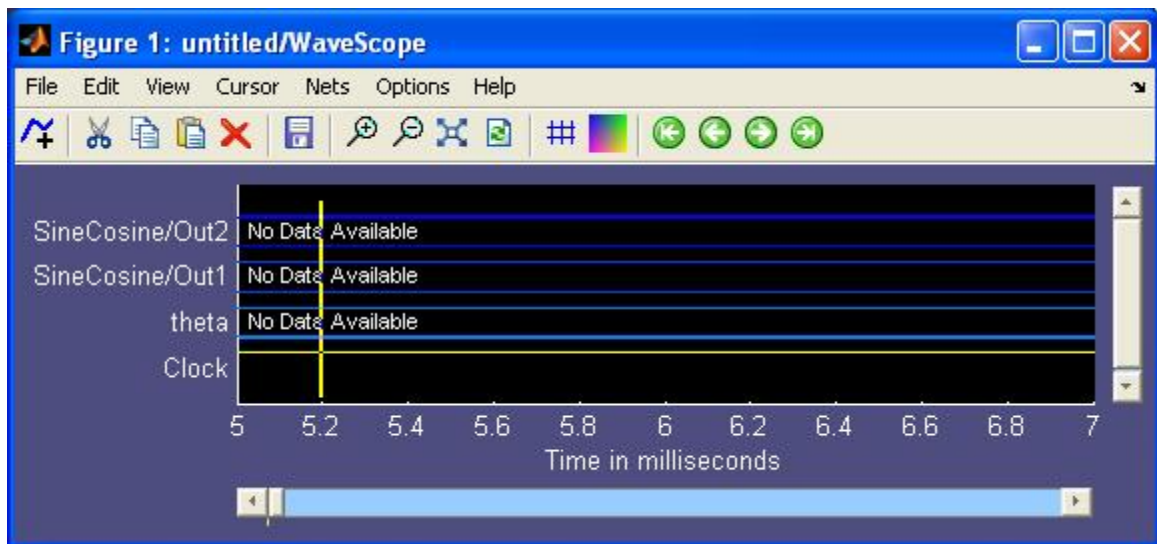
The System Generator WaveScope block provides a powerful and easy-to-use waveform viewer for analyzing and debugging System Generator designs. The viewer allows the user to observe the time-changing values of any wires in the SysGen design after the conclusion of the simulation. The signals may be formatted in a logic or analog format and may be viewed in binary, hex, or decimal radices.

Quick Tutorial

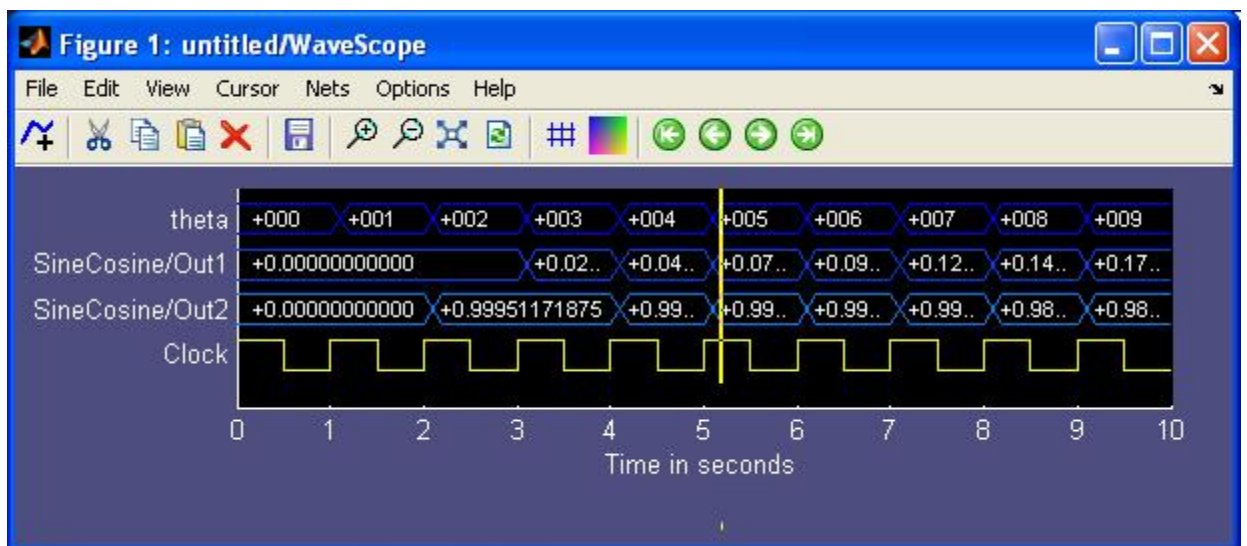
Let us use a simple example to show how to use the WaveScope with this simple model:



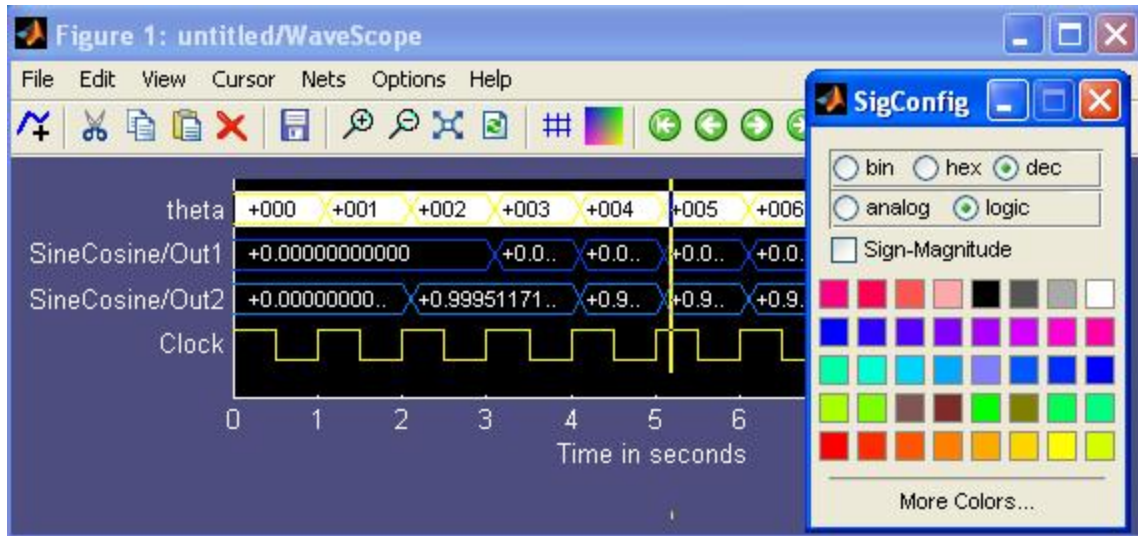
Note that we have dropped the WaveScope block into the model. We double-click on the WaveScope block to open it, which brings up the blank waveform viewer. Now we highlight the three wires in the model by clicking on all three wires while holding the 'shift' key. Then we push the 'Add Selected Nets'  button in the waveform viewer to add those wires to the viewer. The WaveScope window now appears as shown:



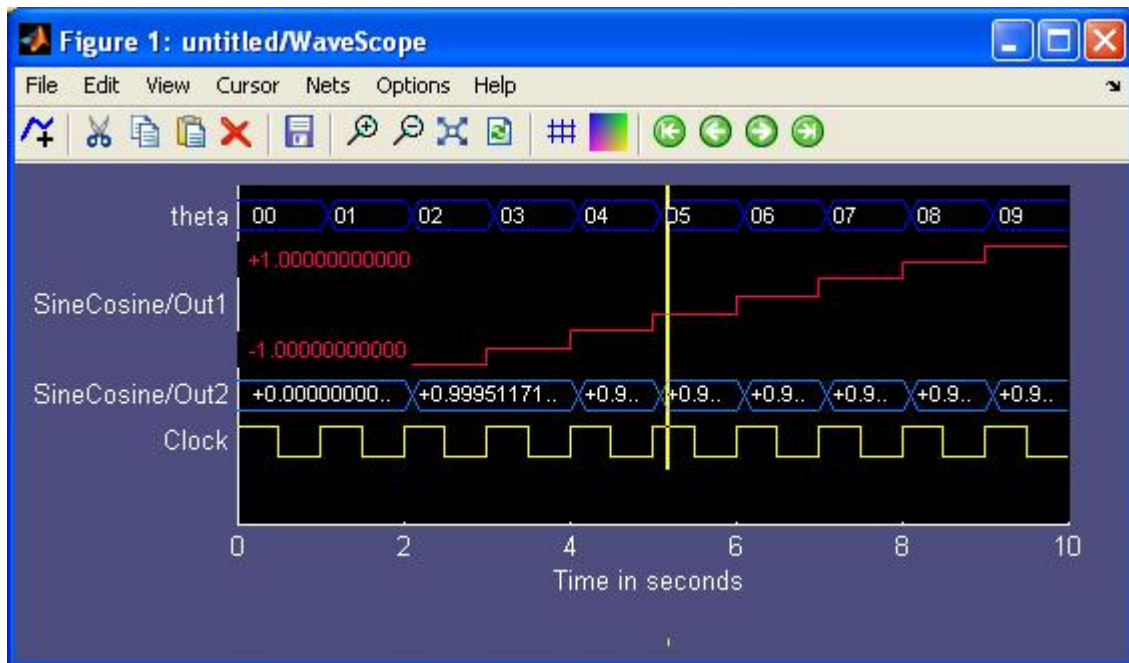
The three signals appear in the viewer. Two of the signals have been automatically named because they were not explicitly named in the model. Now we run the simulation using the 'Play' button on the model's window. This simulation has a period of 1s and runs for 10s. The waveform viewer automatically updates. We zoom out to the full view using the button and the viewer appears as shown:



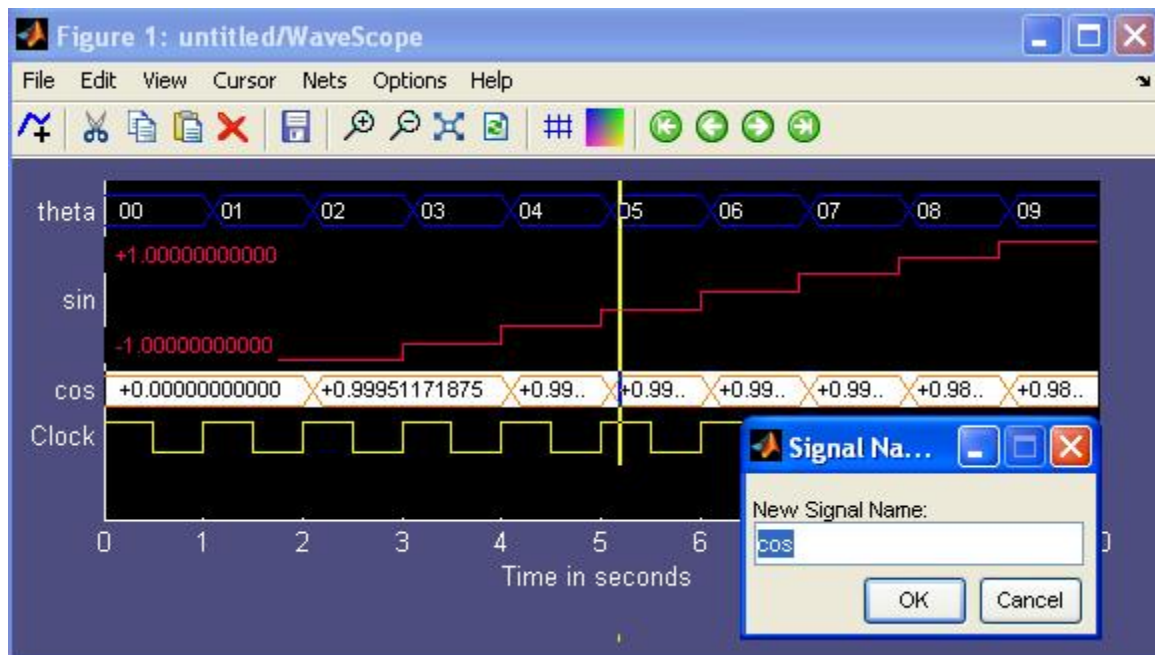
We will change the radix of 'theta' to hex. We click on the name 'theta' or the associated signal waveform to highlight it and then double-click on the highlighted signal (not on the name) to bring up the formatting menu:



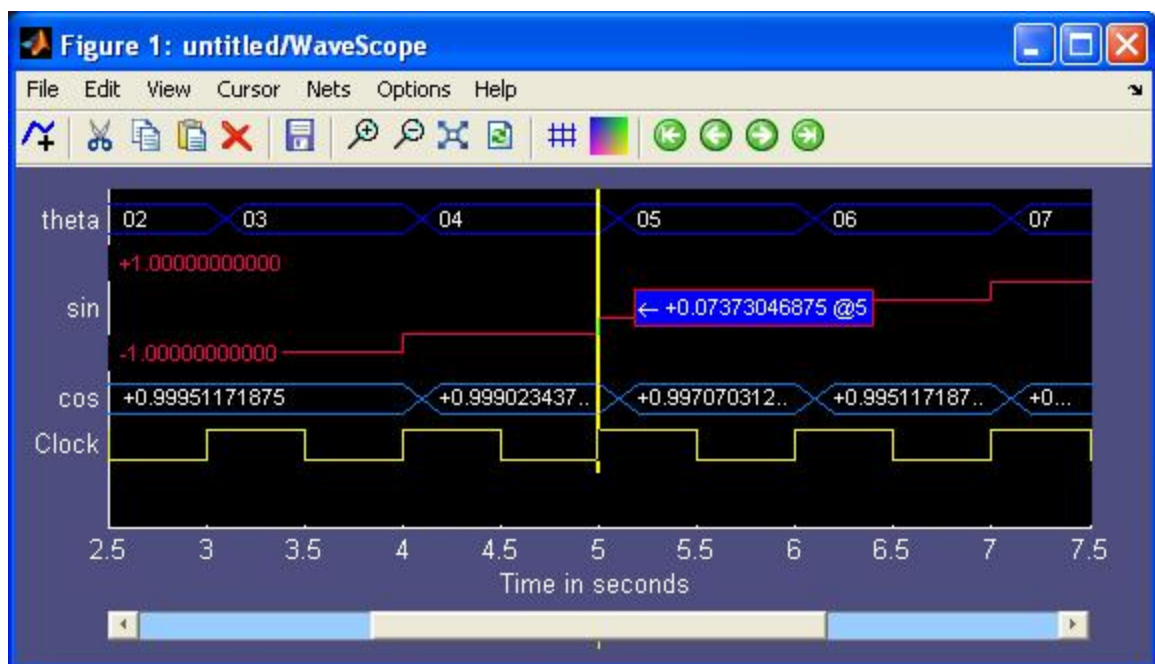
We select the 'hex' radio button to format 'theta' as binary. In a similar fashion, we format the signal 'SineCosine/Out1' as 'analog' and change the color to red:



We can change the names of the signals by double-clicking on the signal names and entering new names in the text box:



The new signal names will be displayed in the model. By using the  button we can zoom in on a portion of the simulation. We can bring the yellow cursor to the center of the screen using the **Cursor->Center Cursor** menu option and observe the value for any signal under the cursor by placing the mouse pointer on the cursor:



Block Interface

Double-clicking the WaveScope icon opens up the WaveScope window. If the WaveScope window is

closed, it will open automatically at the end of a simulation. The WaveScope window is a powerful “scope” in which the simulation results may be displayed in several ways.

WaveScope displays the signal on a given net or nets. The signal can be viewed in more than one way simultaneously, for example, viewing it both in logical and analog formats. Each signal can be displayed either as logic or analog, and the values can be displayed in hexadecimal, binary, or decimal radices. At the bottom of the display is the clock signal for reference.

The WaveScope window can be used to

- Choose which nets’ signals to view
- Configure the signals’ presentation
- View the signals

Selecting Nets

There are two ways to select nets to view in the WaveScope window. Select any output net(s) of a Xilinx block (or the blocks themselves) in the Simulink window, then press the  icon in the WaveScope toolbar (“Add selected nets”). Multiple blocks/nets may be selected by holding the ‘shift’ key while selecting. The signal for the selected net(s) will appear in the WaveScope window. For any blocks that were selected, all of the inputs and outputs to the selected blocks will be added to the WaveScope window. There will be no data for the WaveScope to display until the model is simulated. After simulation, the data will appear in the WaveScope window.

Pressing the “Add selected nets” button in the tool bar multiple times will display the signal in the WaveScope viewer multiple times.

The second method of choosing nets is to use the “Nets” menu. This contains a hierarchical list of blocks and nets in your model. In a complex diagram it may be easier to use this menu to navigate to a particular net.

At the bottom of the display you will see a clock signal representing the highest rate clock in the design. This signal is always displayed whenever any signal is displayed.

Selecting and Moving Signals

Click on a signal or the corresponding net name with the left mouse button to select the signal. Once a signal has been selected it can be moved in the display by dragging it to a different location. If you wish to select several signals at once, use Shift-click or Control-click on the *net names* only; it will not work with the signals themselves.

If you select multiple signals, which need not be contiguous signals in the display, and move them in the display, they will all be moved to a contiguous block of signals. This is handy for displaying several related signals together so they can all be seen at once.

You cannot select or move the clock signal in the WaveScope display. The clock signal will always be the last signal displayed.

Deleting Signals from the WaveScope Window

If you decide not to view a signal after adding it to the WaveScope window, just select the signal and press the “Delete Signals” button on the toolbar. The del key is the keyboard shortcut to this function, and the Edit menu provides a “Delete” item as well.

The standard “Cut”, “Copy” and “Paste” functions are available for signals as well. Using the Copy and Paste icons on the toolbar, keyboard shortcuts Control-X for cut, Control-C for copy and Control-V for paste, or the “Copy” and “Paste” entries in the Edit menu, allows you to display a net multiple times in the WaveScope.

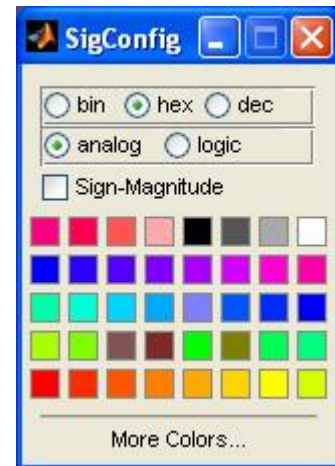
You cannot copy, paste, or delete the clock signal from the WaveScope display.

Configuring the Signals' Presentation

Some signals are naturally viewed as numerical values in which the value is of primary concern, and some as logical states in which the transition is the key datum. With WaveScope you can choose which way to view the signal.

Select the signal(s) in question, then double-click on the signal. (Double-click on the signal itself, not on the signal's name.) A menu will appear with four choices:

- **Format** – Select “logic” to show the signal as a logical signal with transitions emphasized. The value is written after each transition. Select “analog” to display the signal as a graph of the value. The high and low values for the signal are display in the left of the graph as well. The size of the analog signal may be changed by dragging the bottom of the selected analog signal.
- **Radix** – Select “hex,” “binary” or “dec” to choose the radix of the displayed numbers. Numbers will always be displayed with the proper radix point. For example, the decimal number 10.5 would be displayed as A.8 in hex.
- **Sign-Magnitude** - Select “Sign-Magnitude” to have WaveScope interpret the values as a sign-magnitude rather than a two's complement number. Decimal values are always displayed in sign-magnitude format.
- **Color** – WaveScope chooses a default value for a color. Use a colored button to select a new color for all the selected signals.

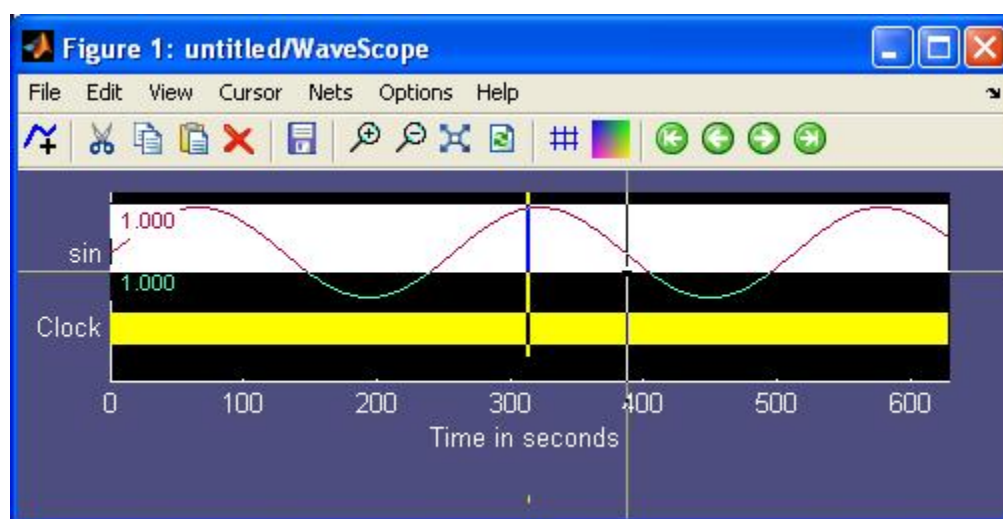


A logic signal will, by default, have the values displayed in the graph. To turn this off, unselect the “Show values” item in the “Options” menu.

You cannot change the clock signal's presentation.

Changing the Height of an Analog Signal

To change the size of the analog signal, grab the bottom edge of a selected analog signal (as shown) and move the bottom up or down to make the analog signal smaller or bigger, respectively:



Changing the Signal Name

Double-click on a signal's name to change it. You may also change the name on the wire in the model.

In this case when the simulation is re-run or the WaveScope window is refreshed using the  button, the signal name will be updated in the WaveScope window.

Rainbowing the Signals

It is easier to observe signals when they are separated in the visual spectrum. As signals are added, a new color is selected from a rainbow palette. A group of signals may be re-rainbowed by selecting a group of signals and pushing the rainbow button. To re-rainbow all of the signals, select them all using Control-A and push the rainbow button:



Viewing the Signals

Once you have selected the signals and simulated the model, WaveScope starts displaying the signals.

Zooming and Scrolling

You can zoom in and out with either the magnifying glass icons, the view menu, or the 'i' and 'o' keys on your keyboard. You may also zoom to a box by dragging a rubberband box in the WaveScope window. Arrow keys will scroll the display, or you can use your mouse in the sliders at the right and below the signal display. Note that when there is sufficient room to display the signals in one dimension or the other, the sliders will not display.

The control key allows for finer-resolution zooming and panning. Holding down the control key while pushing the left and right arrow keys will pan by one clock cycle. Holding down the control keys in conjunction with the 'i' and 'o' keys will zoom in and out by a smaller factor.

Changing the Recording Limits

There are times when you may want to display only a subset of your data. For instance, your simulation may run for a long time, but you are only interested in looking at the last 1000 steps of the simulation.

The more data that is displayed in the WaveScope, the slower the WaveScope will be. One possibility is to zoom in on the desired data, but if there is a lot of data the WaveScope will still be slow. In this case a better solution is to reduce the Recording Limits of the WaveScope.

By default, WaveScope will record all the values on a signal from start to finish of a simulation. You can change these limits by using the “Options” menu and selecting the “Recording Limits” submenu. A dialog will open in which you can set the start and ending time for recording. The dialog is prepopulated with the current lowest and highest value. You can enter any number here. The end time can be set to “Inf”, as well, indicating no preset upper limit. See Figure below.



Figure 1 The set recording limits dialog

Once the recording limits are set, the WaveScope will only display values in that time range. You cannot zoom back out of that range. When you rerun the simulation, only the values at times in that range are recorded.

The Grid

Displaying the Grid  – Clicking the “Toggle Grid” icon on the toolbar will display vertical lines at each labeled x-axis value.

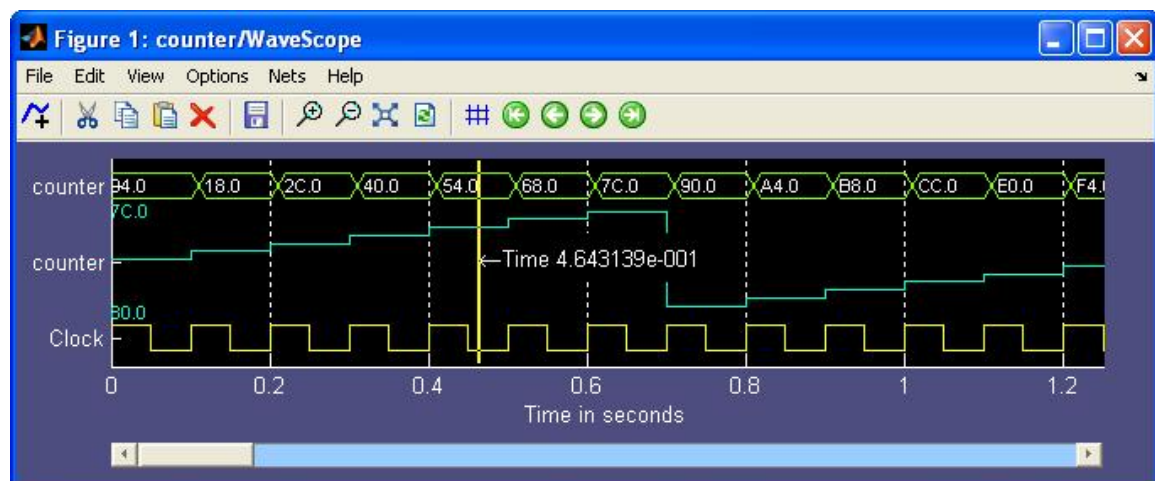


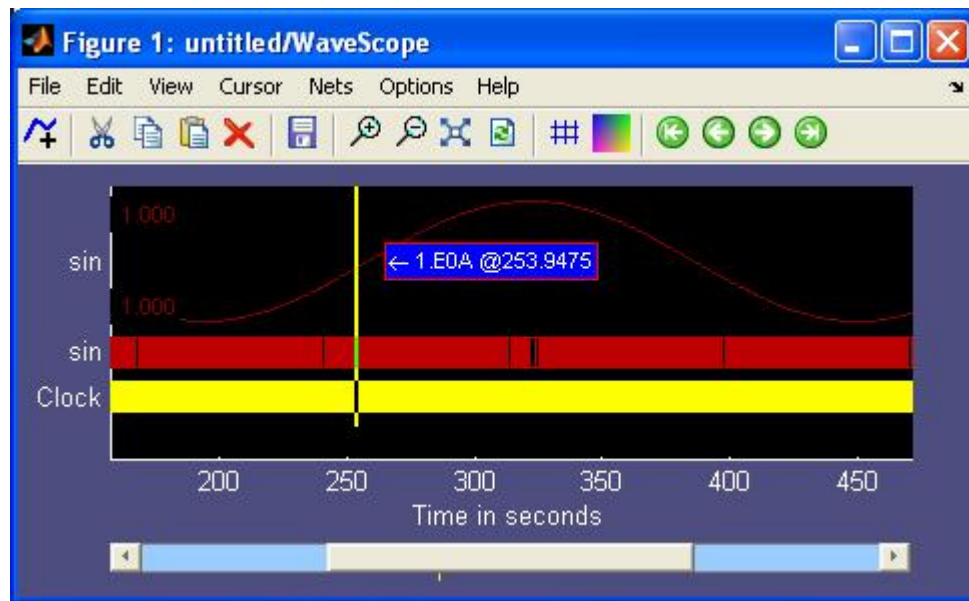
Figure 2 The WaveScope window with the Grid turned on

The Cursor

The cursor is helpful for visually aligning signals or marking a point of interest. The cursor may be brought to the currently-viewed timespan by clicking underneath the time axis. When moving the pointer underneath the time axis, the mouse pointer changes to a cross, indicating that the cursor may be moved to that location and moved around within the current view.

The cursor may also be brought to the center of the screen using the 'c' key or the **Cursor->Center Cursor** menu option. Once onscreen, the cursor may be moved around by dragging it. When the mouse pointer is placed over the cursor, the pointer will change to a cross to show it may be dragged.

When the mouse pointer is over the cursor, a tooltip shows the value of the signal underneath the mouse pointer. This is valuable for displaying the value of an analog signal or the full value of a logical signal when the zoom factor is such that the full value cannot be displayed on the signal:

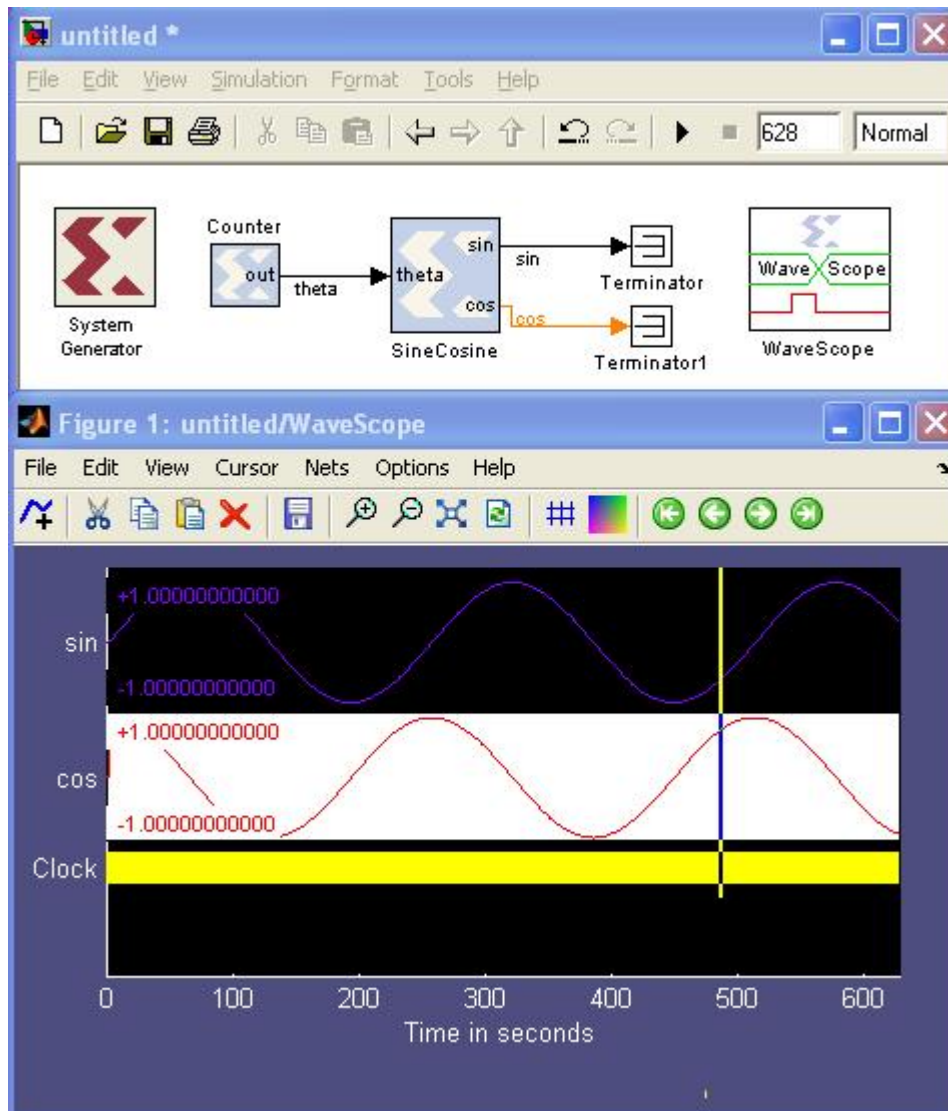


As the cursor is dragged, the tooltip will be updated. Note the minicursor underneath the scrollbar, which appears as a yellow tick mark. When the cursor is not in the selected view, the minicursor shows where the cursor resides on the time axis. To jump to the current cursor location, use the 'j' key or the **Cursor->Jump to Cursor** menu option.

It is often helpful to be able to jump to the next signal transition without having to pan and search for the transition. To jump to the next transition, place the cursor on the screen and select the signal of interest. Press 'enter' or use the **Cursor->Move Cursor Next** menu option to move the cursor to the next signal transition. If the cursor moves offscreen the view will be panned to keep the cursor onscreen.

Crossprobing

When a signal(s) is selected in the WaveScope window, it is cross-probed by highlighting the corresponding wire in the model in orange, as shown here:



If the highlighted signal is underneath some layers of hierarchy, the appropriate mother blocks will be highlighted in orange.

The Cursor Menu

There are four options in the cursor menu.

Center Cursor

This option will bring the cursor to the center of the currently-viewed timespan. This action may also be performed with the 'c' key.

Jump to Cursor

This option moves the current view to the cursor's location. This action may also be performed with the 'j' key.

Move Cursor Next

This option moves the cursor to the next transition of the most recent of the currently-selected signals. This action may also be performed with the 'enter' key.

Move Cursor Last

This option moves the cursor to the previous transition of the most recent of the currently-selected signals. This action may also be performed with shift-enter.

The Options Menu

There are four options in the options menu.

Grid Lines

This option toggles display of the time grid. See **Error! Reference source not found.** for a WaveScope without grid lines. See Figure 2 for a WaveScope with grid lines

Show Values

Show Values toggles the display of numerical values on the WaveScope. By default, WaveScope will display values. Turn off the display with this option.

Run at End of Sim

This option toggles whether the WaveScope should run at the end of a simulation. By default, the WaveScope will display. If you don't want the WaveScope to appear at the end of simulation, use this option

Recording Limits

As explained above in Changing the Recording Limits, this option is used to restrict the simulation time displayed in the WaveScope.

XtremeDSP Analog to Digital Converter



fractional bits.

The Xilinx XtremeDSP ADC block allows System Generator components to connect to the two analog input channels on the Nallatech BenAdda board when a model is prepared for hardware co-simulation. Separate ADC blocks, ADC1 and ADC2 are provided for analog input channels one and two, respectively.

In Simulink, the ADC block is modeled using an input gateway that drives a register. The ADC block accepts a double signal as input and produces a signed 14-bit Xilinx fixed point signal as output. The output signal uses 13

In hardware, a component that is driven by the ADC block output will be driven by one of the two 14-bit AD6644 analog to digital converter devices on the BenAdda board. When a System Generator model that uses an ADC block is translated into hardware, the ADC block is translated into a top-level input port on the model HDL. The appropriate pin location constraints are added in the BenAdda constraints file, thereby ensuring the port is driven appropriately by the ADC component.

A free running clock should be used when a hardware co-simulation model contains an ADC block. In addition, the programmable clock speed should not be set higher than 64 MHz.

Block Parameters Dialog Box

The Block Parameters Dialog Box can be invoked by double clicking the icon in the Simulink™ model.

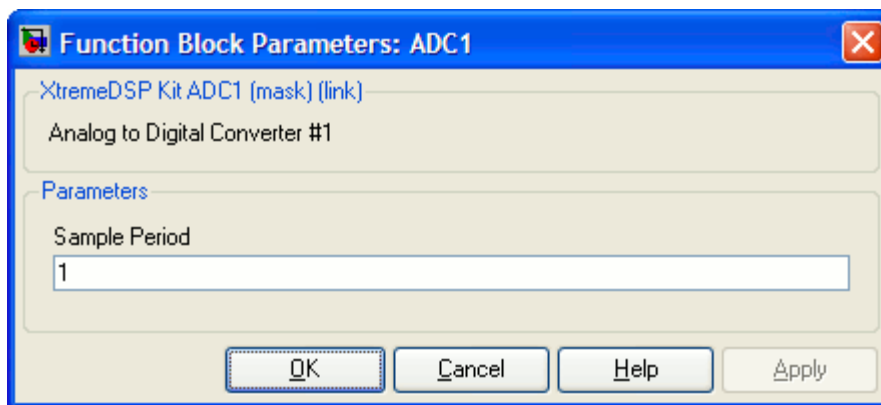


Figure 1: XtremeDSP ADC Block Mask

Data Sheet

A data sheet for the AD6644 device is provided in the XtremeDSP development kit install directory. If FUSE denotes the directory containing the Nallatech FUSE software, the data sheet can be found in the following location:

FUSE\XtremeDSP Development Kit\Docs\Datasheets\ADC ad6644.pdf

XtremeDSP Co-Simulation



The Xilinx XtremeDSP Co-simulation block can be used in place of a Simulink subsystem that was compiled for XtremeDSP co-simulation. During simulation, the block behaves exactly as the subsystem from which it originated, except that the simulation data is processed in hardware instead of software.

The port interface of the co-simulation block will vary. When a model is compiled for co-simulation, a new library is created that contains a custom XtremeDSP hardware co-simulation block. This block has input and output ports that match the gateway names (or port names if the subsystem is not the top level) from the original model.

The hardware co-simulation block interacts with the XtremeDSP development kit board during a Simulink simulation. Simulation data that is written to the input ports of the block are passed to the hardware by the block. Conversely, when data is read from the co-simulation block's output ports, the block reads the appropriate values from the hardware and drives them on the output ports so they can be interpreted in Simulink. In addition, the block automatically opens, configures, steps, and closes the development kit board.

Block Parameters Dialog Box

The Block Parameters Dialog Box can be invoked by double clicking the icon in the Simulink model.

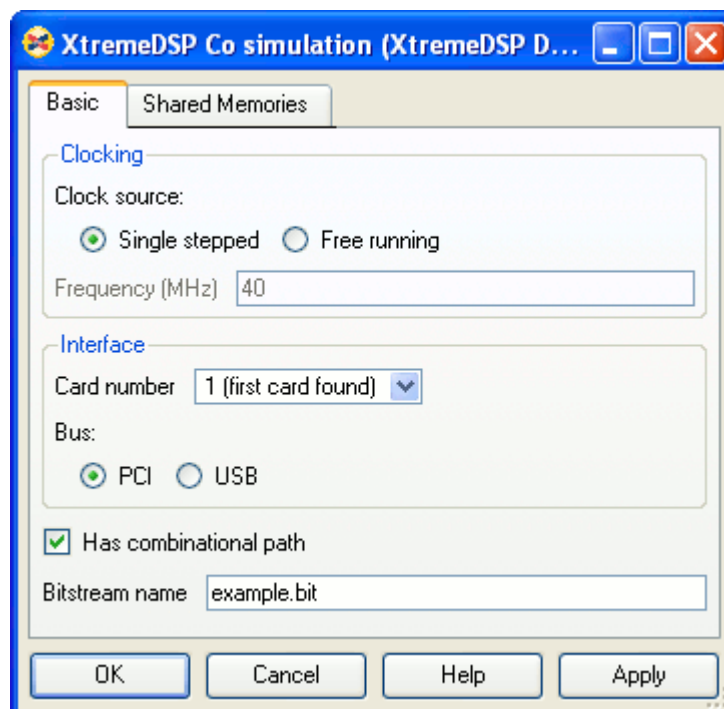


Figure 1: XtremeDSP Co-Simulation Block Mask

The Parameters specific to the block are

- **Clock source:** You may select between **Single stepped** and **Free running** clock sources. Selecting a **Single Stepped** clock allows the block to step the board one clock cycle at a time. Each clock cycle step corresponds to some duration of time in Simulink. Using

this clock source ensures the behavior of the co-simulation hardware during simulation will be bit and cycle accurate when compared to the simulation behavior of the subsystem from which it originated. Sometimes single stepping is not necessary and the board can be run with a **Free Running** clock. In this case, the board will operate asynchronously to the Simulink simulation.

- **Frequency (MHz):** When **Free Running** clock mode is selected, you may specify the operating frequency that the free running clock should be programmed to run at during simulation. The selected clock frequency will be rounded to the nearest valid frequency available from the programmable oscillator. *Note: You must take care to specify a frequency that does not exceed the maximum operating frequency of the model's FPGA implementation.* The valid operating frequencies of the programmable oscillator are listed below:

20 MHz; 25 MHz; 30 MHz; 33.33 MHz; 40 MHz; 45 MHz; 50 MHz; 60 MHz; 66.66 MHz; 70 MHz; 75 MHz; 80 MHz; 90 MHz; 100 MHz; 120 MHz.

- **Card number:** Specifies the index of the XtremeDSP development kit card to use for hardware co-simulation. A default value of 1 should be used unless you have multiple XtremeDSP kit boards installed.
- **Bus:** Allows you to choose the interface in which the co-simulation block communicates with the XtremeDSP development kit board during a Simulink simulation. You may select between **PCI** and **USB** interfaces.
- **Has combinational path:** Sometimes it is necessary to have a direct combinational feedback path from an output port on a hardware co-simulation block to an input port on the same block (e.g., a wire connecting an output port to an input port on a given block). If you require a direct feedback path from an output to input port, and your design does not include a combinational path from any input port to any output port, unchecking this box will allow the feedback path in the design.
- **Bitstream name:** Specifies the co-simulation FPGA configuration file for the XtremeDSP development kit board. When a new co-simulation block is instantiated during compilation, this parameter is automatically set so that it points to the appropriate configuration file. You need only adjust this parameter if the location of the configuration file changes.

XtremeDSP Digital to Analog Converter



The Xilinx XtremeDSP DAC block allows System Generator components to connect to the two analog output channels on the Nallatech BenAdda board when a model is prepared for hardware co-simulation. Separate DAC blocks DAC1 and DAC2 are provided for analog output channels one and two respectively.

In Simulink, the DAC block is modeled by a register block that drives an output gateway. All DAC control signals are appropriately wired to constants.

The DAC block must be driven by a 14-bit Xilinx fixed point signal, with the binary point at position 13. The output port of the DAC block produces a signal of type double.

In hardware, a component that drives a DAC block input will drive one of the two 14-bit AD9772A digital to analog converter devices on the BenAdda board. When a System Generator model that uses DAC block is translated into hardware, the DAC block is translated into a top-level output port on the model HDL. The appropriate pin location constraints are added in the BenAdda constraints file, thereby ensuring the output port drives the appropriate DAC pins.

A free running clock should be used when a hardware co-simulation model contains a DAC block. In addition, the programmable clock speed should not be set higher than 64 MHz.

Block Parameters Dialog Box

The Block Parameters Dialog Box can be invoked by double clicking the icon in the Simulink™ model.

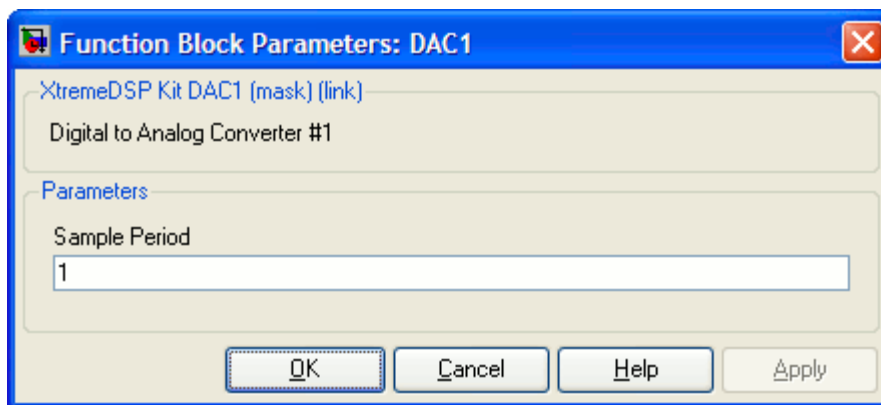


Figure 1: XtremeDSP DAC Block Mask

Data Sheet

A data sheet for the AD9772A device is provided in the directory to which the XtremeDSP development kit has been installed. If FUSE denotes the directory containing the FUSE software, the data sheet can be found in the following location:

FUSE\XtremeDSP Development Kit\Docs\Datasheets\DAC AD9772A.pdf

XtremeDSP External RAM



The Xilinx XtremeDSP External RAM block allows System Generator components to connect to the external 256K x 16 ZBT SRAM on the Nallatech BenAdda board when a model is prepared for hardware co-simulation.

The block provides a Simulink simulation model for the memory device. The ports on the block look and behave like ports on a traditional synchronous RAM device. The **address** port should be driven by an unsigned 18-bit Xilinx fixed-point signal having binary point at position 0.

The **we** port should be driven by a Xilinx Boolean signal. The **data** port should be driven by a 16-bit Xilinx fixed-point signal. The block drives 16-bit Xilinx fixed-point data values on its output port.

In hardware, components that read from and write to the block in Simulink read from and write to the Micron ZBT SRAM device on the BenAdda board. When a System Generator model that uses an external RAM block is translated into hardware, the ports on the RAM block are translated into top-level input and output ports on the model HDL. The appropriate pin location constraints for these ports are included in the BenAdda constraints file. The ZBT SRAM device uses the same clock as the System Generator portion of the hardware co-simulation implementation.

Block Parameters Dialog Box

The Block Parameters Dialog Box can be invoked by double clicking the icon in the Simulink™ model.

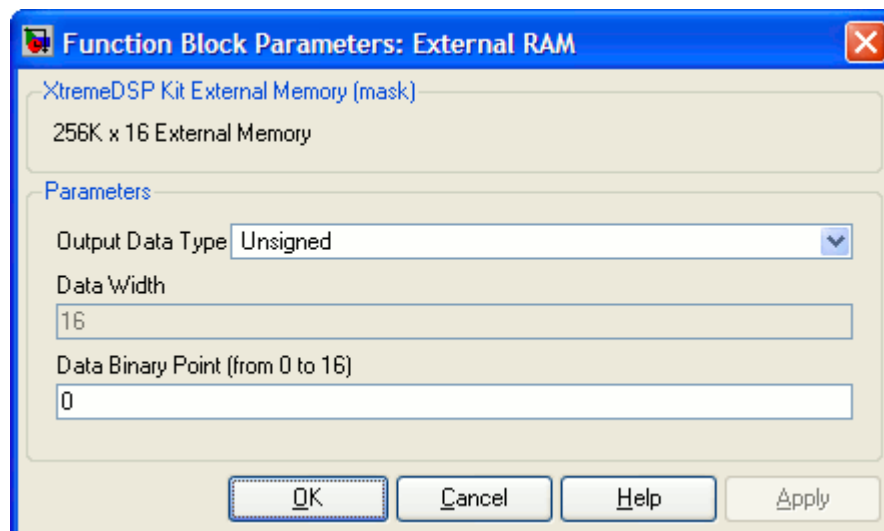


Figure 1: XtremeDSP External RAM Block Mask

The parameters specific to the block are:

- **Output Data Type** : selects the output data type of the RAM. You may choose between unsigned and signed (two's complement) data types.
- **Data Binary Point** : selects the binary point position of the data values stored as the memory contents. The binary point position must be between 0 and 16 (the data width).

XtremeDSP LED Flasher



LED Flasher

The Xilinx XtremeDSP LED Flasher block allows System Generator models to use the tri-color LEDs on the BenADDA board when a model is prepared for co-simulation. When the model is co-simulated, the LEDs will cycle through red, green and yellow colors. The LEDs are driven by the two most significant bits of a 27-bit free running counter. To see the LEDs cycle through the three colors, you should select a free running clock during model simulation.

Block Parameters Dialog Box

The Block Parameters Dialog Box can be invoked by double clicking the icon in the Simulink™ model.

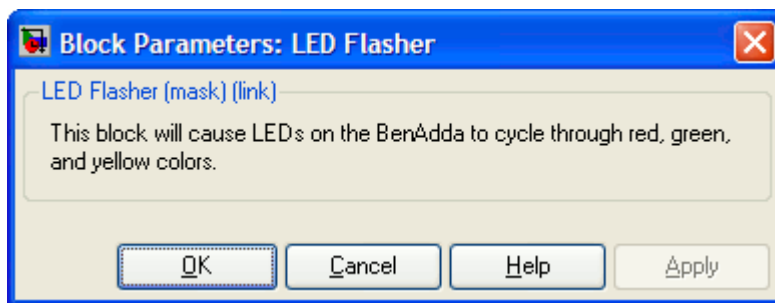
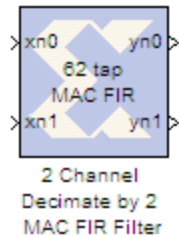


Figure 1: XtremeDSP LED Flasher Block Mask

Reference Design Library

2 Channel Decimate by 2 MAC FIR Filter (DSP)



The Xilinx n-tap 2 Channel Decimate by 2 MAC FIR Filter reference block implements a multiply-accumulate based FIR filter. One dedicated multiplier and one Dual Port Block RAM are used in the n-tap filter. The same MAC engine is used to process both channels that are time division multiplexed (TDM) together. Completely different coefficient sets can be specified for each channel as long as they have the same number of coefficients. The filter also provides a fixed decimation by 2 using a polyphase filter technique. The filter configuration helps illustrate techniques for storing multiple coefficient sets and data samples in filter

design. The Virtex FPGA family (and Virtex family derivatives) provide dedicated circuitry for building fast, compact adders, multipliers, and flexible memory architectures. The filter design takes advantage of these silicon features by implementing a design that is compact and resource efficient.

Annotation is provided in the filter design subsystems. Please click on the reference block and refer to this documentation seen in the block model itself for more specific implementation details.

Block Parameters Dialog Box

The block parameters dialog may be opened by double clicking on the icon in your Simulink model:

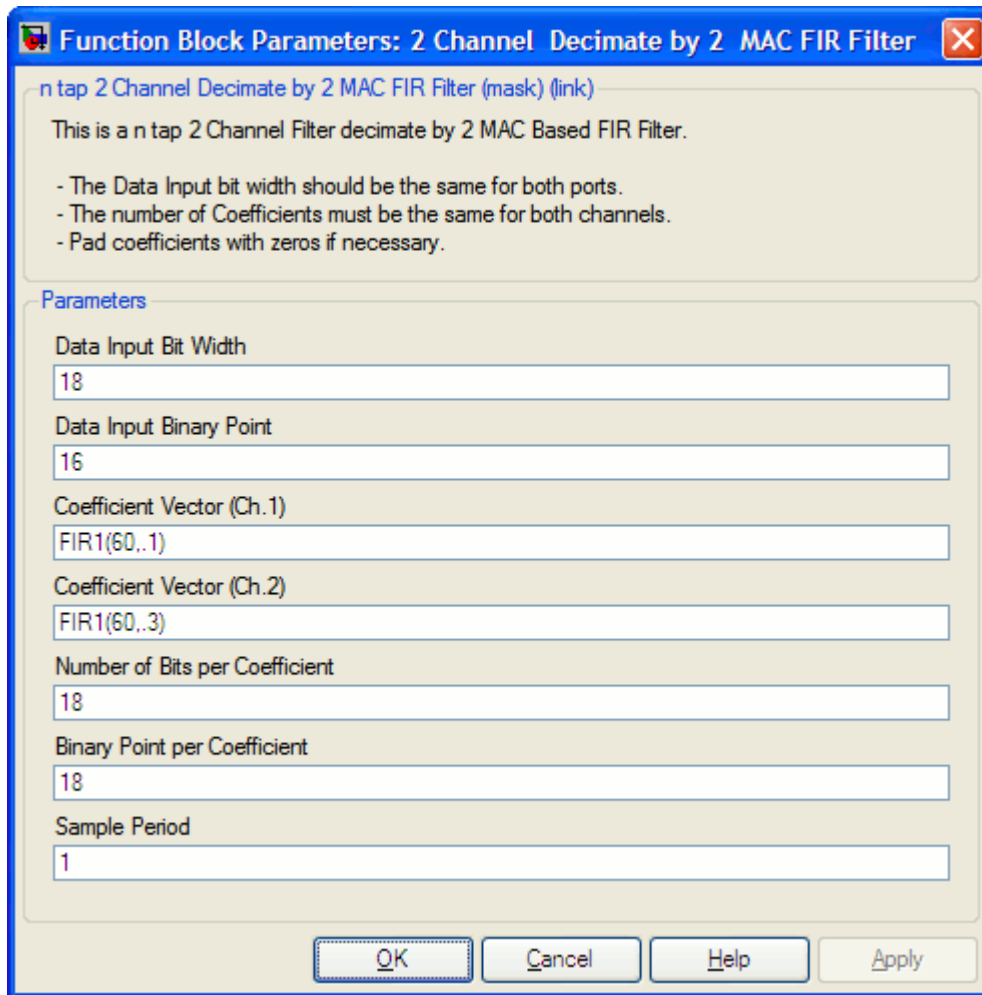


Figure 1: n-tap 2 Channel Decimate by 2 MAC FIR Filter parameters dialog box

Parameters specific to this reference block are:

- **Data Input Bit Width:** Width of input sample.
- **Data Input Binary Point:** Binary point location of input.
- **Coefficient Vector (Ch.1):** Specify coefficients for Channel 1 of the filter. Number of taps is inferred from size of coefficient vector.
- **Coefficient Vector (Ch.2):** Specify coefficients for Channel 2 of the filter. Number of taps is inferred from size of coefficient vector.

Note: Coefficient Vectors must be the same size. Pad coefficients if necessary to make them the same size.

- **Number of Bits per Coefficient:** Bit width of each coefficient.
- **Binary Point per Coefficient:** Binary point location for each coefficient.

Note: Coefficient Vectors must be the same size. Pad coefficients if necessary to make them the same size.

- **Sample Period:** Sample period of input.

Reference

J. Hwang and J. Ballagh. *Building Custom FIR Filters Using System Generator*. 12th International Field-Programmable Logic and Applications Conference (FPL). Montpellier, France, September 2002. Lecture Notes in Computer Science 2438.

2n+1-tap Linear Phase MAC FIR Filter (DSP)



The Xilinx 2n+1-tap Linear Phase MAC FIR Filter reference block implements a multiply-accumulate based FIR filter. The 2n+1-tap Linear Phase MAC FIR filter exploits coefficient symmetry for an odd number of coefficients to increase filter throughput. These filter designs exploit silicon features found in Virtex family FPGAs such as dedicated circuitry for building fast, compact adders, multipliers, and flexible memory architectures.

Annotation is provided in the filter design subsystems. Please click on the reference block and refer to this documentation seen in the block model itself for more specific implementation details.

Block Parameters Dialog Box

The block parameters dialog may be opened by double clicking on the icon in your Simulink model:

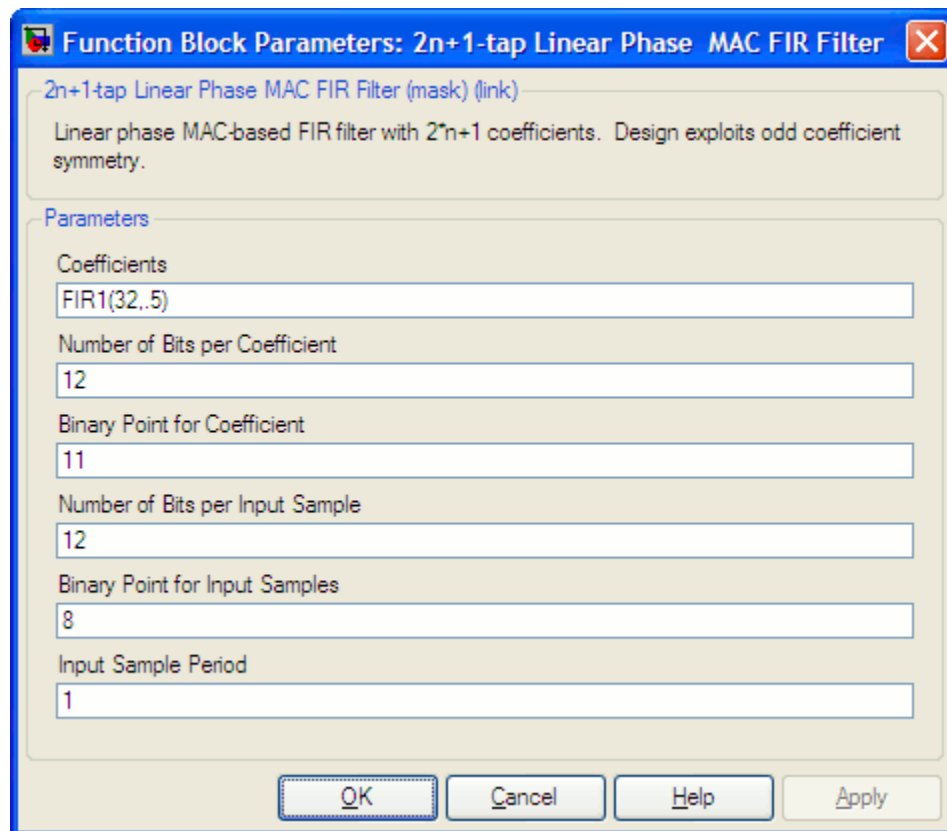


Figure 1: 2n+1-tap Linear Phase MAC FIR Filter block parameters dialog box

Parameters specific to this reference block are:

- Coefficients: Specify coefficients for the filter. Number of taps is inferred from size of coefficient vector.
- Number of Bits per Coefficient: Bit width of each coefficient.
- Binary Point for Coefficient: Binary point location for each coefficient.
- Number of Bits per Input Sample: Width of input sample.
- Binary Point for Input Samples: Binary point location of input.

- Input Sample Period: Sample period of input.

Reference

J. Hwang and J. Ballagh. *Building Custom FIR Filters Using System Generator*. 12th International Field-Programmable Logic and Applications Conference (FPL). Montpellier, France, September 2002. Lecture Notes in Computer Science 2438.

2n-tap Linear Phase MAC FIR Filter (DSP)



The Xilinx 2n-tap linear phase MAC FIR filter reference block implements a multiply-accumulate based FIR filter. The block exploits coefficient symmetry for an even number of coefficients to increase filter throughput. These filter designs exploit silicon features found in Virtex family FPGAs such as dedicated circuitry for building fast, compact adders, multipliers, and flexible memory architectures.

Annotation is provided in the filter design subsystems. Please click on the reference block and refer to this documentation seen in the block model itself for more specific implementation details.

Block Parameters Dialog Box

The block parameters dialog may be opened by double clicking on the icon in your Simulink model:

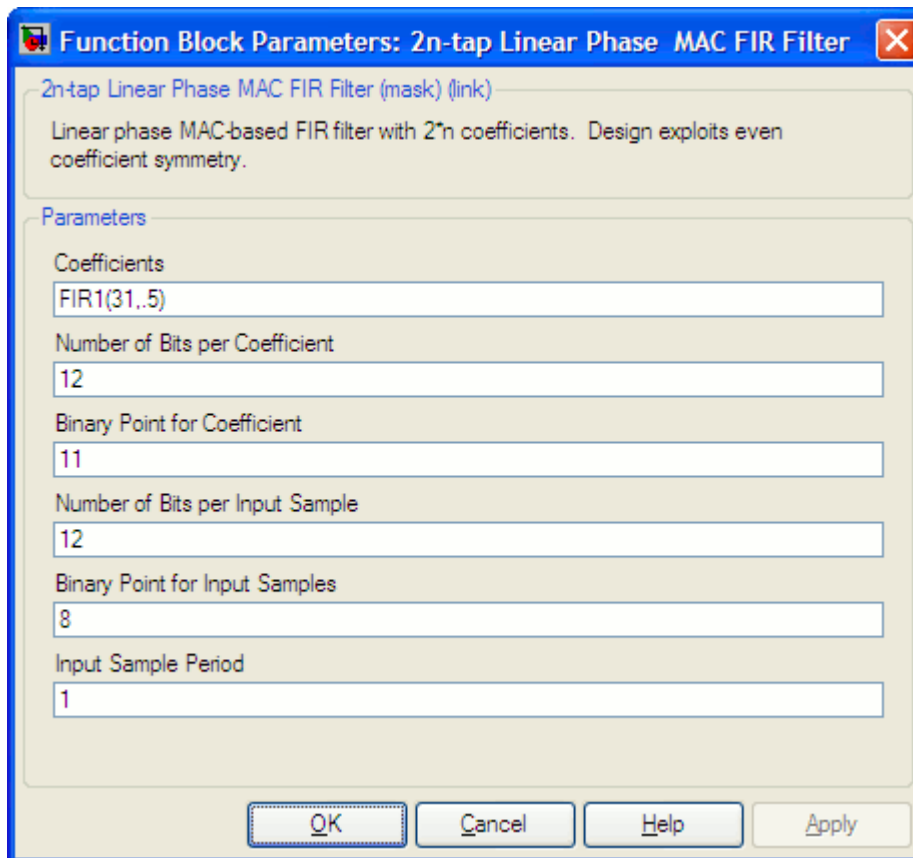


Figure 1: 2n-tap Linear Phase MAC FIR Filter block parameters dialog box

Parameters specific to this reference block are:

- Coefficients: Specify coefficients for the filter. Number of taps is inferred from size of coefficient vector.
- Number of Bits per Coefficient: Bit width of each coefficient.
- Binary Point for Coefficient: Binary point location for each coefficient.
- Number of Bits per Input Sample: Width of input sample.
- Binary Point for Input Samples: Binary point location of input.

- Input Sample Period: Sample period of input.

Reference

J. Hwang and J. Ballagh. *Building Custom FIR Filters Using System Generator*. 12th International Field-Programmable Logic and Applications Conference (FPL). Montpellier, France, September 2002. Lecture Notes in Computer Science 2438.

2n-tap MAC FIR Filter (DSP)



The Xilinx 2n-tap MAC FIR Filter reference block implements a multiply-accumulate based FIR filter. One, two, and four Virtex-II dedicated multipliers are used in the n-tap, the 2n-tap, and the 4n-tap filters, respectively. The three filter configurations help illustrate the tradeoffs between filter throughput and device resource consumption. The Virtex FPGA family (and Virtex family derivatives) provide dedicated circuitry for building fast, compact adders, multipliers, and flexible memory architectures. Each filter design takes advantage of these silicon features by implementing a design that is compact and resource efficient.

Annotation is provided in the filter design subsystems. Please click on the reference block and refer to this documentation seen in the block model itself for more specific implementation details.

Block Parameters Dialog Box

The block parameters dialog may be opened by double clicking on the icon in your Simulink model:

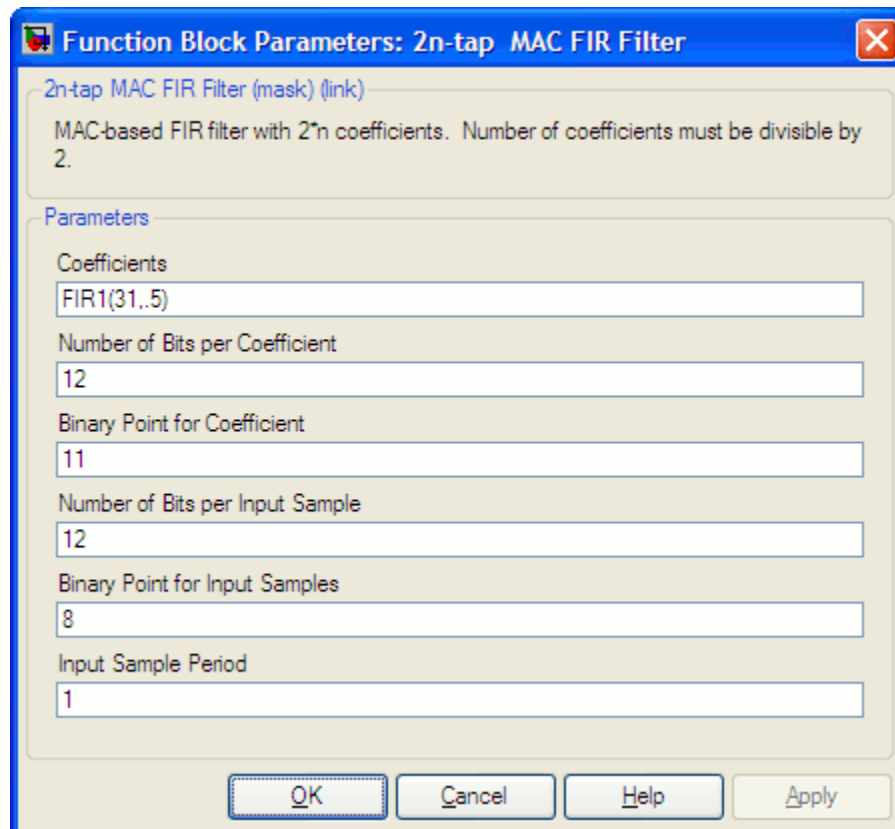


Figure 1: 2n-tap MAC FIR Filter block parameters dialog box

Parameters specific to this reference block are:

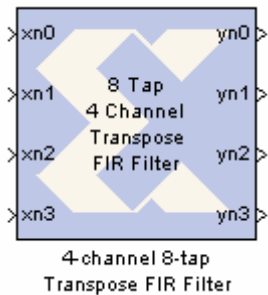
- **Coefficients:** Specify coefficients for the filter. Number of taps is inferred from size of coefficient vector.
- **Number of Bits per Coefficient:** Bit width of each coefficient.
- **Binary Point for Coefficient:** Binary point location for each coefficient.
- **Number of Bits per Input Sample:** Width of input sample.

- Binary Point for Input Samples: Binary point location of input.
- Input Sample Period: Sample period of input.

Reference

J. Hwang and J. Ballagh. *Building Custom FIR Filters Using System Generator*. 12th International Field-Programmable Logic and Applications Conference (FPL). Montpellier, France, September 2002. Lecture Notes in Computer Science 2438.

4-channel 8-tap Transpose FIR Filter (DSP)



The Xilinx 4-channel 8-tap Transpose FIR Filter reference block implements a 4-channel 8-tap transpose FIR filter. The transpose structure is well suited for data path processing in Xilinx FPGAs, and is easily extended to produce larger filters (space accommodating). The filter takes advantage of silicon features found in the Virtex family FPGAs such as dedicated circuitry for building fast, compact adders, multipliers, and flexible memory architectures.

Annotation is provided in the filter design subsystems. Please click on the reference block and refer to the documentation seen in the block model itself for more specific implementation details.

Block Parameters Dialog Box

The block parameters dialog may be opened by double clicking on the icon in your Simulink model:

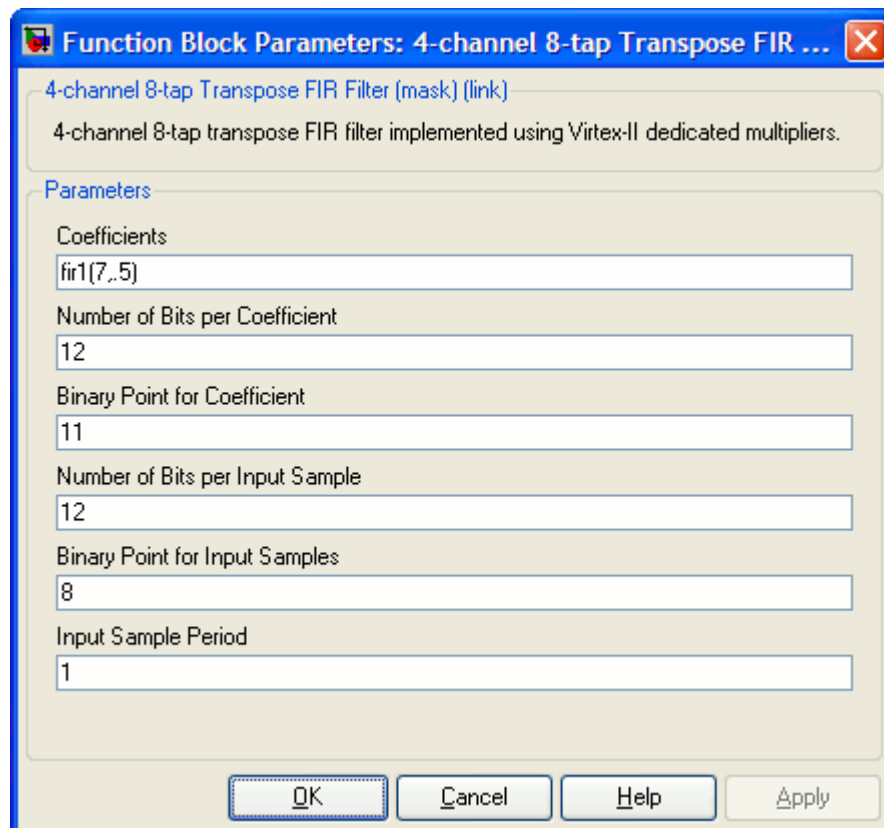
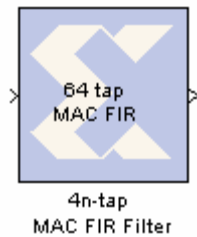


Figure 1: 4-channel 8-tap Transpose FIR Filter block parameters dialog box

Parameters specific to this reference block are:

- **Coefficients:** Specify coefficients for the filter. Number of taps is inferred from size of coefficient vector.
- **Number of Bits per Coefficient:** Bit width of each coefficient.
- **Binary Point for Coefficient:** Binary point location for each coefficient.
- **Number of Bits per Input Sample:** Width of input sample.
- **Binary Point for Input Samples:** Binary point location of input.
- **Input Sample Period:** Sample period of input.

4n-tap MAC FIR Filter (DSP)



The Xilinx 4n-tap MAC FIR Filter reference block implements a multiply-accumulate based FIR filter. One, two, and four Virtex-II dedicated multipliers are used in the n-tap, the 2n-tap, and the 4n-tap filters, respectively. The three filter configurations help illustrate the tradeoffs between filter throughput and device resource consumption. The Virtex FPGA family (and Virtex family derivatives) provide dedicated circuitry for building fast, compact adders, multipliers, and flexible memory architectures. Each filter design takes advantage of these silicon features by implementing a design that is compact and resource efficient.

Annotation is provided in the filter design subsystems. Please click on the reference block and refer to this documentation seen in the block model itself for more specific implementation details.

Block Parameters Dialog Box

The block parameters dialog may be opened by double clicking on the icon in your Simulink model:

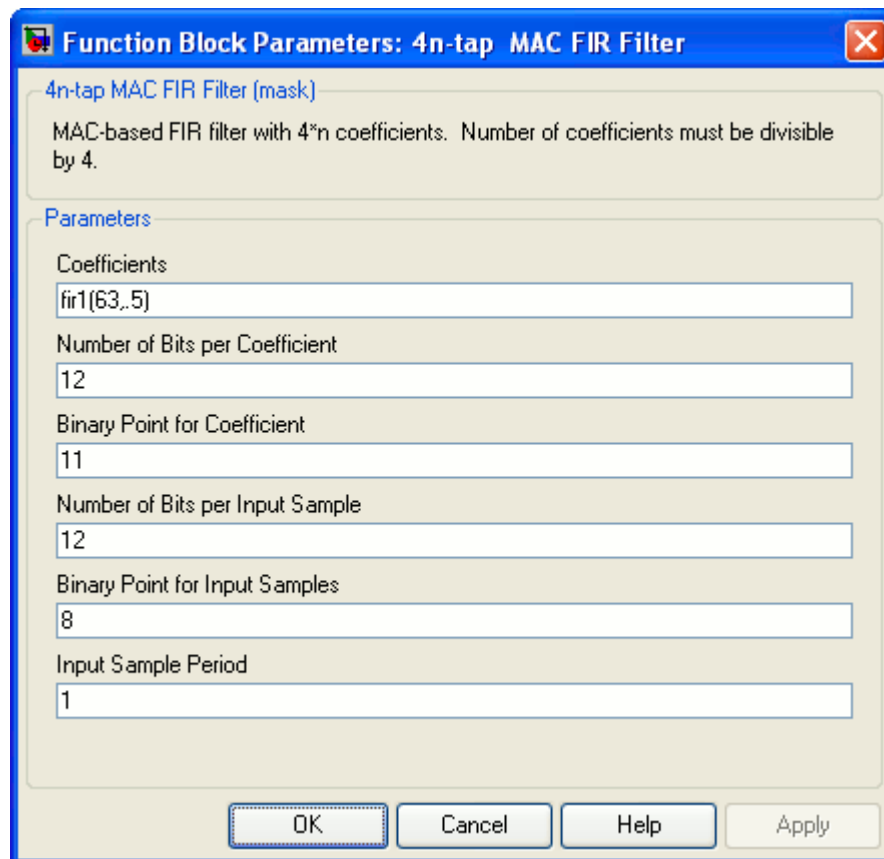


Figure 1: 4n-tap MAC FIR Filter block parameters dialog box

Parameters specific to this reference block are:

- **Coefficients:** Specify coefficients for the filter. Number of taps is inferred from size of coefficient vector.
- **Number of Bits per Coefficient:** Bit width of each coefficient.
- **Binary Point for Coefficient:** Binary point location for each coefficient.
- **Number of Bits per Input Sample:** Width of input sample.

- Binary Point for Input Samples: Binary point location of input.
- Input Sample Period: Sample period of input.

Reference

J. Hwang and J. Ballagh. *Building Custom FIR Filters Using System Generator*. 12th International Field-Programmable Logic and Applications Conference (FPL). Montpellier, France, September 2002. Lecture Notes in Computer Science 2438.

5x5 Filter (Imaging)



The Xilinx 5x5 Filter reference block is implemented using 5 n-tap MAC FIR Filters. The filters can be found in the DSP library of the Xilinx Reference Blockset.

Nine different 2-D filters have been provided to filter grayscale images. The filter can be selected by changing the mask parameter on the 5x5 Filter block. The 2-D filter coefficients are stored in a block RAM, and the model makes no specific optimizations for these coefficients. You can substitute your own coefficients and scale factor by modifying the mask of the 5x5 filter block, under the Initialization tab.

The coefficients used are shown below for the 9 filters. The output of the filter is multiplied by the scale factor named <filter name>Div.

```
edge = [ 0 0 0 0 0; ...
0 -1 -1 -1 0; ...
0 -1 -1 -1 0; ...
0 0 0 0 0];
edgeDiv = 1;
```

```
sobelX = [ 0 0 0 0 0; ...
0 -1 0 1 0; ...
0 -2 0 2 0; ...
0 -1 0 1 0; ...
0 0 0 0 0];
sobelXDiv = 1;
```

```
sobelY = [ 0 0 0 0 0; ...
0 1 2 1 0; ...
0 0 0 0 0; ...
0 -1 -2 -1 0; ...
0 0 0 0 0];
sobelYDiv = 1;
```

```
sobelXY = [ 0 0 0 0 0; ...
0 0 -1 -1 0; ...
0 1 0 -1 0; ...
0 1 1 0 0; ...
0 0 0 0 0];
sobelXYDiv = 1;
```

```
blur = [ 1 1 1 1 1; ...
1 0 0 0 1; ...
1 0 0 0 1; ...
1 0 0 0 1; ...
1 1 1 1 1];
blurDiv = 1/16;
```

```
smooth = [ 1 1 1 1 1; ...
1 5 5 5 1; ...
1 5 44 5 1; ...
1 5 5 5 1; ...
1 1 1 1 1];
```

```
smoothDiv = 1/100;

sharpen = [ 0  0  0  0 0; ...
0 -2 -2 -2 0; ...
0 -2 32 -2 0; ...
0 -2 -2 -2 0; ...
0  0  0  0 0];
sharpenDiv = 1/16;

gaussian = [1 1 2 1 1; ...
1 2 4 2 1; ...
2 4 8 4 2; ...
1 2 4 2 1; ...
1 1 2 1 1];
gaussianDiv = 1/52;

identity = [ 0  0  0  0 0; ...
0  0  0  0 0; ...
0  0  1  0 0; ...
0  0  0  0 0; ...
0  0  0  0 0];
identityDiv = 1;
```

This filter occupies 309 slices, 5 dedicated multipliers, and 5 block rams of a Xilinx xc2v250-6 part and operates at 213 MHz (advanced speeds files 1.96, ISE 4.2.01i software).

The underlying 5-tap MAC FIR filters are clocked 5 times faster than the input rate. Therefore the throughput of the design is 213 MHz / 5 = 42.6 million pixels/ second. For a 64x64 image, this is $42.6 \times 10^6 / (64 \times 64) = 10,400$ frames/sec. For a 256x256 image the throughput would be 650 frames/sec, and for a 512x512 image it would be 162 frames/sec.

Block Parameters Dialog Box

The block parameters dialog may be opened by double clicking on the icon in your Simulink model:

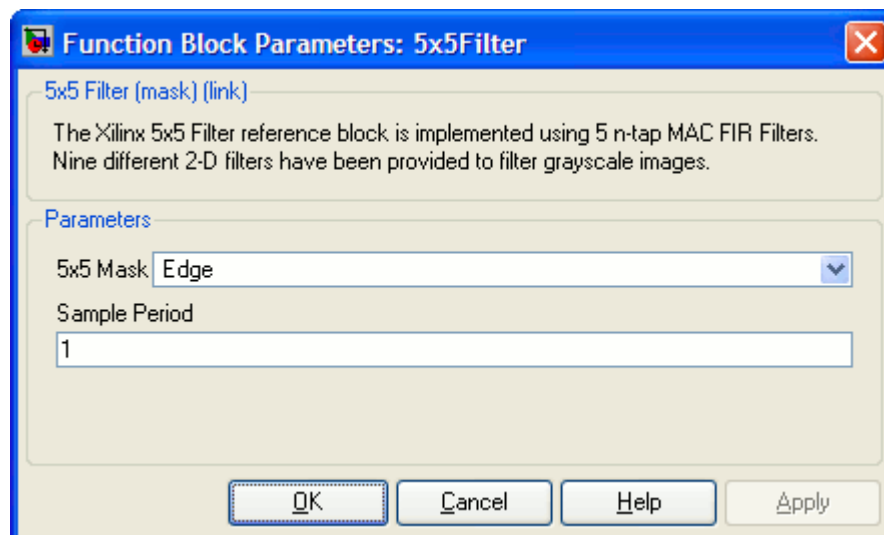
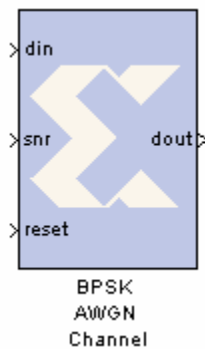


Figure 1: 5x5 Filter block parameters dialog box

Parameters specific to the block are:

- **5x5 Mask:** The coefficients for an Edge, Sobel X, Sobel Y, Sobel X-Y, Blur, Smooth, Sharpen, Gaussian, or Identity filter can be selected.
- **Sample Period:** The sample period at which the input signal runs at is required.

BPSK AWGN Channel (Communication)



The Xilinx BPSK AWGN Channel reference block adds scaled white Gaussian noise to an input signal. The noise is created by the white Gaussian noise generator reference block, described elsewhere in this chapter.

The noise is scaled based on the SNR to achieve the desired noise variance, as shown below. The SNR is defined as (E_b/N_0) in dB for uncoded BPSK with unit symbol energy ($E_s = 1$). The SNR input is UFix8_4 and the valid range is from 0.0 to 15.9375 in steps of 0.0625dB.

To use the AWGN in a system with coding and/or to use the core with different modulation formats, it is necessary to adjust the SNR value to accommodate the difference in spectral efficiency. If we have BPSK modulation with rate 1/2 coding and keep $E_s = 1$ and N_0 constant, then $E_b = 2$ and $E_b/N_0 = \text{SNR} + 3$ dB. If we have uncoded QPSK modulation with $I = \pm 1$ and $Q = \pm 1$ and add independent noise sequences, then each channel looks like an independent BPSK channel and the $E_b/N_0 = \text{SNR}$. If we then add rate 1/2 coding to the QPSK case, we have $E_b/N_0 = \text{SNR} + 3$ dB.

The overall latency of the AWGN Channel is 15 clock cycles. Channel output is a 17 bit signed number with 11 bits after the binary point. The input port snr can be any type. The reset port must be Boolean and the input port din must be of unsigned 1-bit type with binary point position at zero.

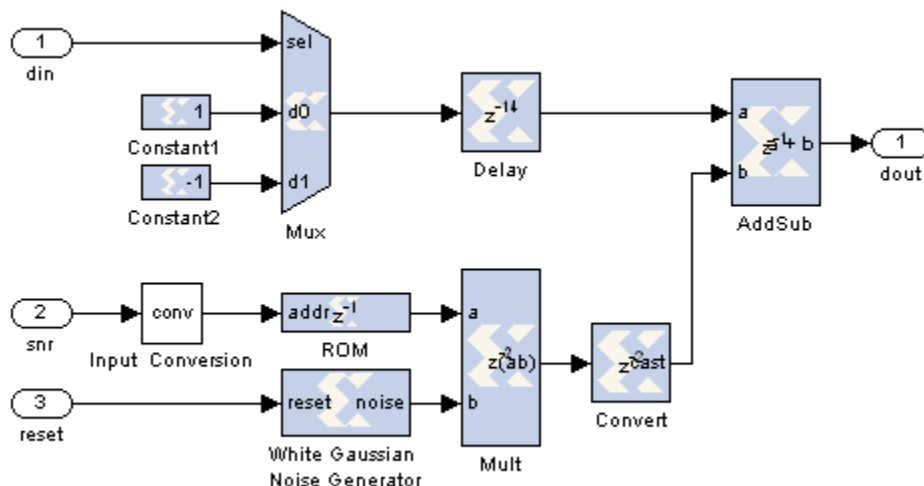


Figure 1: SNR Mapping

Block Parameters Dialog Box

The block parameters dialog may be opened by double clicking on the icon in your Simulink model:

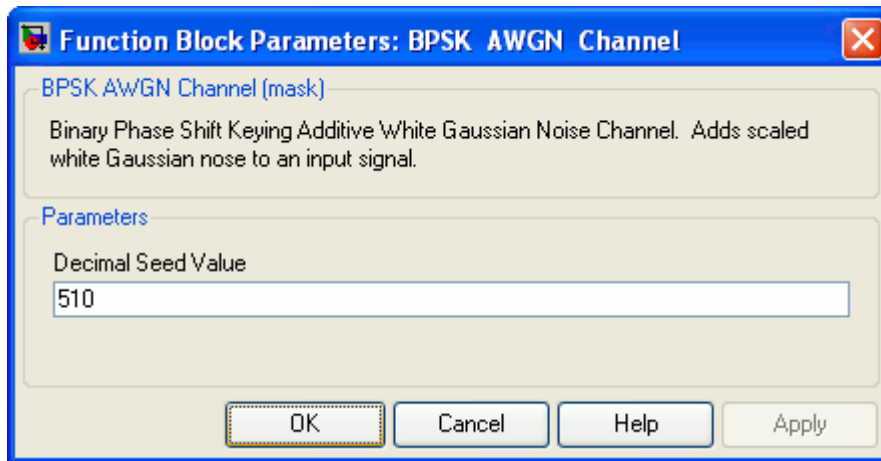


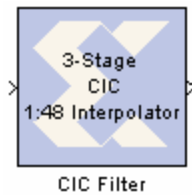
Figure 2: BPSK AWGN Channel block parameters dialog box

The block parameter is the decimal starting seed value.

Reference

- [1] A. Ghazel, E. Boutillon, J. L. Danger, G. Gulak and H. Laamari, Design and Performance Analysis of a High Speed AWGN Communication Channel Emulator, IEEE PACRIM Conference, Victoria, B. C., Aug. 2001.
- [2] Xilinx Data Sheet: Additive White Gaussian Noise (AWGN) Core v1.0, Xilinx, Inc. October 2002

CIC Filter (DSP)



Cascaded integrator-comb (CIC) filters are multirate filters used for realizing large sample rate changes in digital systems. Both decimation and interpolation structures are supported. CIC filters contain no multipliers; they consist only of adders, subtractors and registers. They are typically employed in applications that have a large excess sample rate; that is, the system sample rate is much larger than the bandwidth occupied by the signal. CIC filters are frequently used in digital down-converters and digital up-converters.

Annotation is provided in the filter design subsystems. Please click on the reference block and refer to the documentation seen in the block model itself for more specific implementation details.

Block Interface

The CIC Block has a single data input port and a data output port:

- x_n : data input port, can be between 1 and 128 bits (inclusive).
- y_n : data output port

The two basic building blocks of a CIC filter are the integrator and the comb. A single integrator is a single-pole IIR filter with a transfer function of:

$$H(z) = (1 - z^{-1})^{-1}$$

The integrator's unity feedback coefficient is $y[n] = y[n-1] + x[n]$.

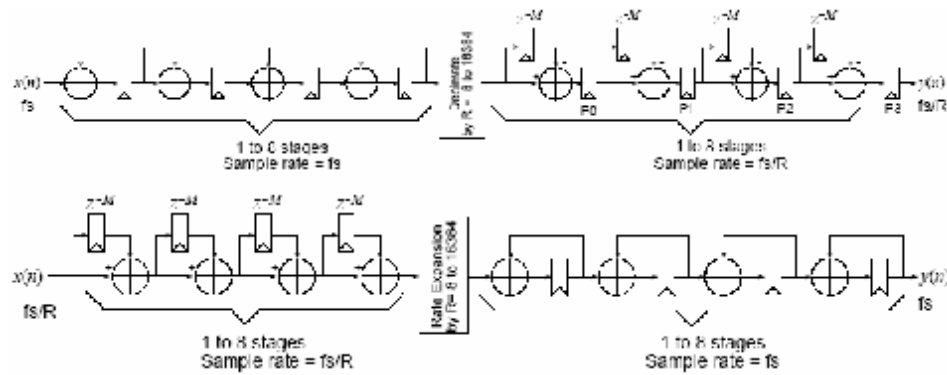
A single comb filter is an odd-symmetric FIR filter described by:

$$y[n] = x[n] - x[n - RM]$$

M is the differential delay selected in the block dialog box, and R is the selected integer rate change factor. The transfer function for a single comb stage is

$$H(z) = 1 - z^{-RM}$$

As seen in the two figures below, the CIC filter cascades N integrator sections together with N comb sections. To keep the integrator and comb structures independent of rate change, a rate change block (i.e., an up-sampler or down-sampler) is inserted between the sections. In the interpolator, the up-sampler causes a rate increase by a factor of R by inserting $R-1$ zero-valued samples between consecutive samples of the comb section output. In the decimator, the down-sampler reduces the sample rate by a factor of R by taking subsamples of the output from the last integrator stage.



Block Parameters Dialog Box

The block parameters dialog may be opened by double clicking on the icon in your Simulink model:

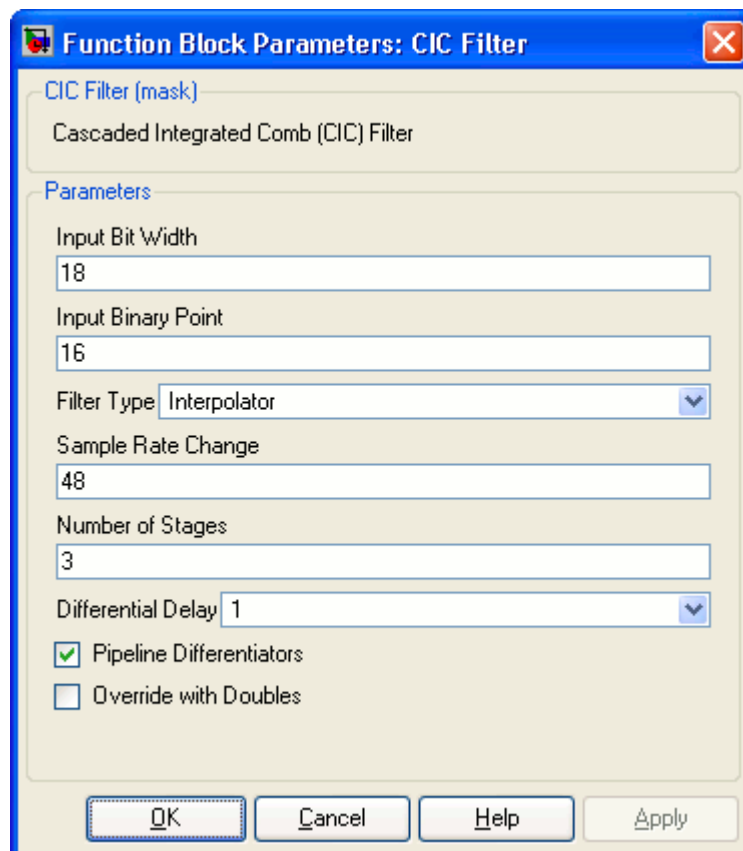


Figure 1: CIC Filter block parameters dialog box

Parameters specific to this reference block are:

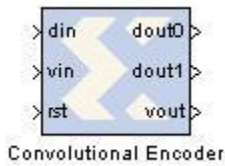
- Input Bit Width: Width of input sample.
- Input Binary Point: Binary point location of input.
- Filter Type: Interpolator or Decimator
- Sample Rate Change: 8 to 16384 (inclusive)
- Number of Stages: 1 to 32 (inclusive)
- Differential Delay: 1 to 4 (inclusive)
- Pipelined Differentiators: On or Off

Other parameters used by this block are described in the Common Parameters section at the beginning of this chapter.

Reference

E. B. Hogenauer. *An economical class of digital filters for decimation and interpolation*. IEEE Transactions on Acoustics, Speech and Signal Processing, ASSP- 29(2):155{162, 1981.

Convolutional Encoder



The Xilinx Convolutional Encoder Model block implements an encoder for convolutional codes. Ordinarily used in tandem with a Viterbi decoder, this block performs forward error correction (FEC) in digital communication systems.

Values are encoded using a linear feed forward shift register which computes modulo-two sums over a sliding window of input data, as shown in the figure below. The length of the shift register is specified by the constraint length. The convolution codes specify which bits in the data window contribute to the modulo-two sum. Resetting the block will set the shift register to zero. The encoder rate is the ratio of input to output bit length; thus, for example a rate 1/2 encoder outputs two bits for each input bit. Similarly, a rate 1/3 encoder outputs three bits for each input bit.

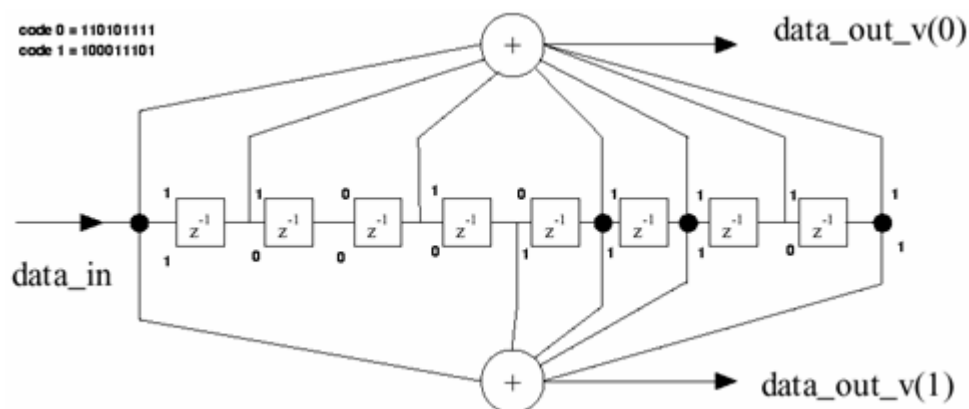


Figure 1: Constraint length 9 convolutional encoder

Implementation

The block is implemented using a form of parameterizable mux-based collapsing. In this method constants drive logic blocks. Here the constant is the convolution code which is used to determine which register in the linear feed forward shift register is to be used in computing the output. All logic driven by a constant will be optimized away by the synthesizer.

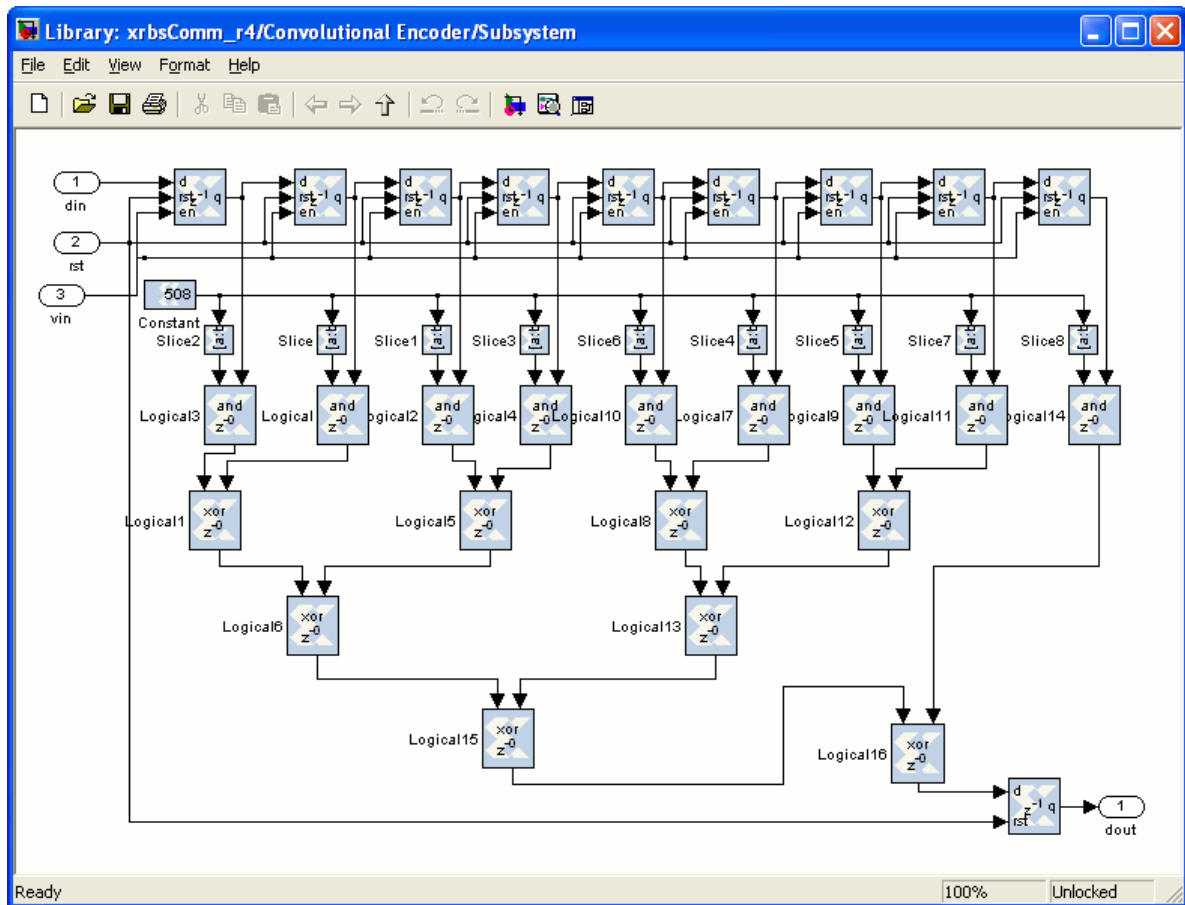


Figure 2: Implementation of the linear feed forward shift register and logic to compute dout.

Block Interface

The block currently has three input ports and three output ports. The `din` port must have type `UFix1_0`. It accepts the values to be encoded. The `vin` port indicates that the values presented on `din` are valid. Only valid values are encoded. The reset port will reset the convolution encoder when high. To add an enable port, open the subsystem and change the constant "Enable" to an input port. The output ports `dout0` and `dout1` output the encoded data. The port `dout0` corresponds to the first code in the array, `dout1` to the second, and so on. To add additional output ports, open the subsystem and follow the directions in the model. The output port `vout` indicates the validity of output values.

Block Parameters Dialog Box

The following figure shows the block parameters dialog box.

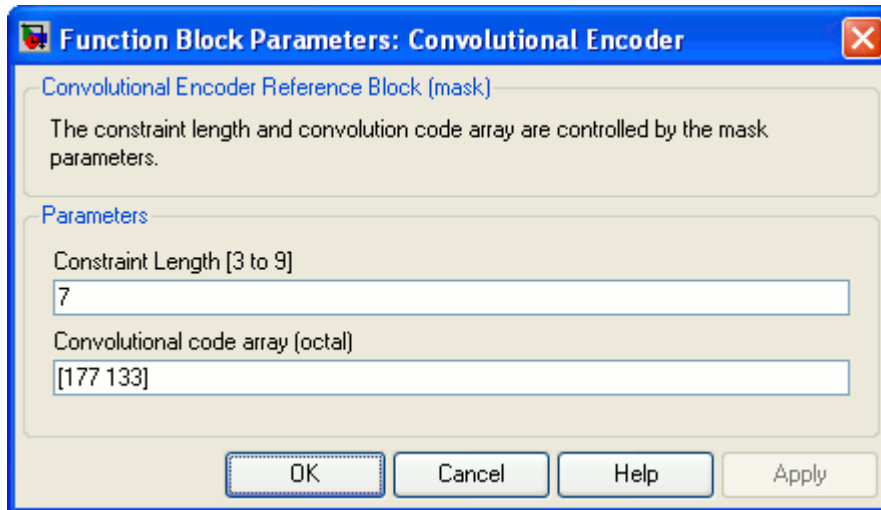
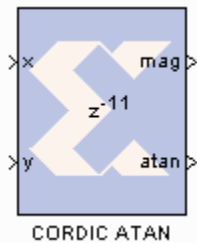


Figure 2: Convolutional encoder block parameters dialog box

Parameters specific to the block are:

- **Constraint length:** Constraint Length: Equals $n+1$, where n is the length of the constraint register in the encoder.
- **Convolution code array (octal):** Array of octal convolution codes. Output rate is derived from the array length. Between 2 and 7 (inclusive) codes may be entered.

CORDIC ATAN (Math)



The Xilinx CORDIC ATAN reference block implements a rectangular-to-polar coordinate conversion using a fully parallel CORDIC (COordinate Rotation Digital Computer) algorithm in Circular Vectoring mode.

That is, given a complex-input $\langle x, y \rangle$, it computes a new vector $\langle m, a \rangle$, where magnitude $m = K \times \sqrt{x^2 + y^2}$, and the angle $a = \arctan(y/x)$. As is common, the magnitude scale factor $K = 1.646760\dots$ is not compensated in the processor, i.e. the magnitude output should be scaled by this factor. The CORDIC processor is implemented using building blocks from the Xilinx

blockset.

The CORDIC ATAN algorithm is implemented in the following 3 steps:

1. Coarse Angle Rotation. The algorithm converges only for angles between $-\pi/2$ and $\pi/2$, so if $x < \text{zero}$, the input vector is reflected to the 1st or 3rd quadrant by making the x-coordinate non-negative.
2. Fine Angle Rotation. For rectangular-to-polar conversion, the resulting vector is rotated through progressively smaller angles, such that y goes to zero. In the i -th stage, the angular rotation is by either $\pm \arctan(1/2^i)$, depending on whether or not its input y is less than or greater than zero.
3. Angle Correction. If there was a reflection applied in Step 1, this step applies the appropriate angle correction by subtracting it from $\pm \pi$.

Block Parameters Dialog Box

The block parameters dialog may be opened by double clicking on the icon in your Simulink model:

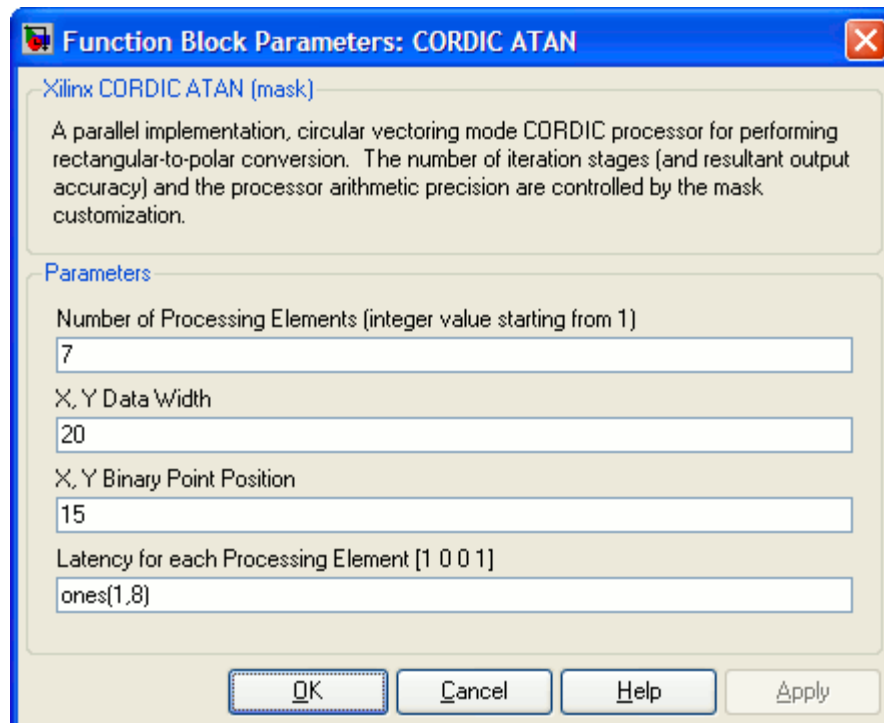


Figure 1: CORDIC ATAN block parameters dialog box

Parameters specific to the block are:

- **Number of Processing Elements**: specifies the number of iterative stages used for fine angle rotation.
- **X,Y Data Width** : specifies the width of the inputs x and y. The inputs x, and y should be signed data type having the same data width.
- **X,Y Binary Point Position**: specifies the binary point position for inputs x and y. The inputs x and y should be signed data type with the same binary point position.
- **Latency for each Processing element**: This parameter sets the pipeline latency after each circular rotation stage.

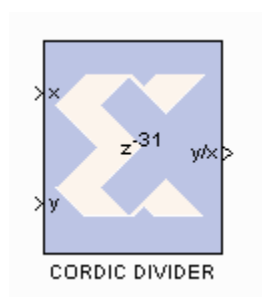
The latency of the CORDIC arc tangent block is calculated based on the formula specified below:

$$\text{Latency} = 3 + \text{sum (latency of Processing Elements)}$$

Reference

- 1) J. E. Volder, "The CORDIC Trigonometric Computing Technique", IRE Trans. On Electronic Computers, Vol. EC-8, 1959, pp. 330-334.
- 2) J. S. Walther, "A Unified Algorithm for Elementary Functions", Spring Joint Computer Conference (1971) pp. 379-385.
- 3) Yu Hen Hu, "CORDIC-Based VLSI Architectures for Digital Signal Processing", IEEE Signal Processing Magazine, pp. 17-34, July 1992.

CORDIC DIVIDER (Math)



The Xilinx CORDIC DIVIDER reference block implements a divider circuit using a fully parallel CORDIC (COordinate Rotation DIgital Computer) algorithm in Linear Vectoring mode.

That is, given an input $\langle x, y \rangle$, it computes the output y/x . The CORDIC processor is implemented using building blocks from the Xilinx blockset.

The CORDIC divider algorithm is implemented in the following 4 steps:

1. **Co-ordinate Rotation:** The CORDIC algorithm converges only for positive values of x . The input vector is always mapped to the 1st quadrant by making the x and y coordinate non-negative. The divider circuit has been designed to converge for all values of X and Y , except for the most negative value.
2. **Normalization:** The CORDIC algorithm converges only for y less than or equal to $2x$. The inputs x and y are shifted to the left until they have a 1 in the most significant bit (MSB). The relative shift of y over x is recorded and passed on to the co-ordinate correction stage.
3. **Linear Rotations:** For ratio calculation, the resulting vector is rotated through progressively smaller angles, such that y goes to zero. In the final stage, the rotation yields y/x .
4. **Co-ordinate Correction:** Based on the co-ordinate axis and a relative shift applied to y over x , this step assigns the appropriate sign to the resulting ratio and multiplies it with $2^{(\text{relative shift of } y \text{ over } x)}$.

Block Parameters Dialog Box

The block parameters dialog may be opened by double clicking on the icon in your Simulink model:

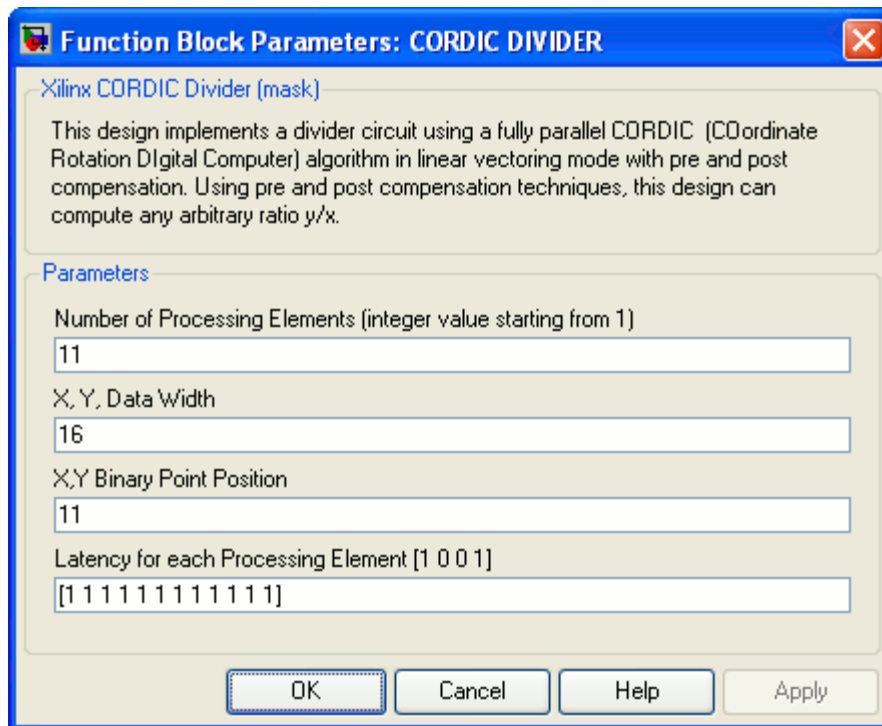


Figure 1: CORDIC DIVIDER block parameters dialog box

Parameters specific to the block are:

- **Number of Processing Elements:** specifies the number of iterative stages used for linear rotation.
- **X, Y Data Width:** specifies the width of the inputs x and y. The inputs x and y should be signed data type with the same data width.
- **X, Y Binary Point Position:** specifies the binary point position for inputs x and y. The inputs x and y should be signed data type with the same binary point position.
- **Latency for each Processing element:** This parameter sets the pipeline latency after each iterative linear rotation stage.

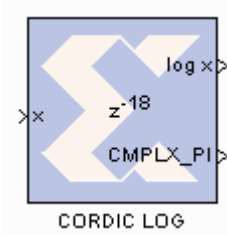
The latency of the CORDIC divider block is calculated based on the formula specified below:

$$\text{Latency} = 4 + \text{data width} + \text{sum (latency of Processing Elements)}$$

Reference

- 1) J. E. Volder, "The CORDIC Trigonometric Computing Technique", IRE Trans. On Electronic Computers, Vol. EC-8, 1959, pp. 330-334.
- 2) J. S. Walther, "A Unified Algorithm for Elementary Functions", Spring Joint Computer Conference (1971) pp. 379-385.
- 3) Yu Hen Hu, "CORDIC-Based VLSI Architectures for Digital Signal Processing", IEEE Signal Processing Magazine, pp. 17-34, July 1992.

CORDIC LOG (Math)



The Xilinx CORDIC LOG reference block implements a natural logarithm circuit using a fully parallel CORDIC (COordinate Rotation Digital Computer) algorithm in Hyperbolic Vectoring mode.

That is, given an input x , it computes the output $\log(x)$ and also provides a flag for adding complex pi value to the output if a complex output is desired. The CORDIC processor is implemented using building blocks from the Xilinx blockset.

The natural logarithm is calculated indirectly by the CORDIC algorithm by applying the identities listed below.

$$\log(w) = 2 \times \tanh^{-1}[(w-1)/(w+1)]$$

$$\log(w \times 2^E) = \log(w) + E \times \log(2)$$

The CORDIC log algorithm is implemented in the following 4 steps:

1. Co-ordinate Rotation: The CORDIC algorithm converges only for positive values of x . If $x < 0$, the input data is converted to a non-negative number. If $x = 0$, a zero detect flag is passed along to the last stage which can be exposed at the output stage. The log circuit has been designed to converge for all values of x , except for the most negative value.
2. Normalization: The CORDIC algorithm converges only for x , between the values 0.5 (inclusive) and 1. During normalization, the input X is shifted to the left till it has a 1 in the most significant bit. The log output is derived using the identity $\log(w) = 2 \times \tanh^{-1}\{(w-1)/(w+1)\}$. Based on this identity, the input w gets mapped to, $x = w + 1$ and $y = w - 1$.
3. Linear Rotations: For $\tanh^{-1}\{(w-1)/(w+1)\}$ calculation, the resulting vector is rotated through progressively smaller angles, such that y goes to zero.
4. Co-ordinate Correction: If the input was negative a CMPLX_PI flag is provided at the output for adding PI if a complex output is desired. If a left shift was applied to X , this step adjusts the output by using the equation $\log(w \times 2^E) = \log(w) + E \times \log(2)$.

Block Parameters Dialog Box

The block parameters dialog may be opened by double clicking on the icon in your Simulink model:

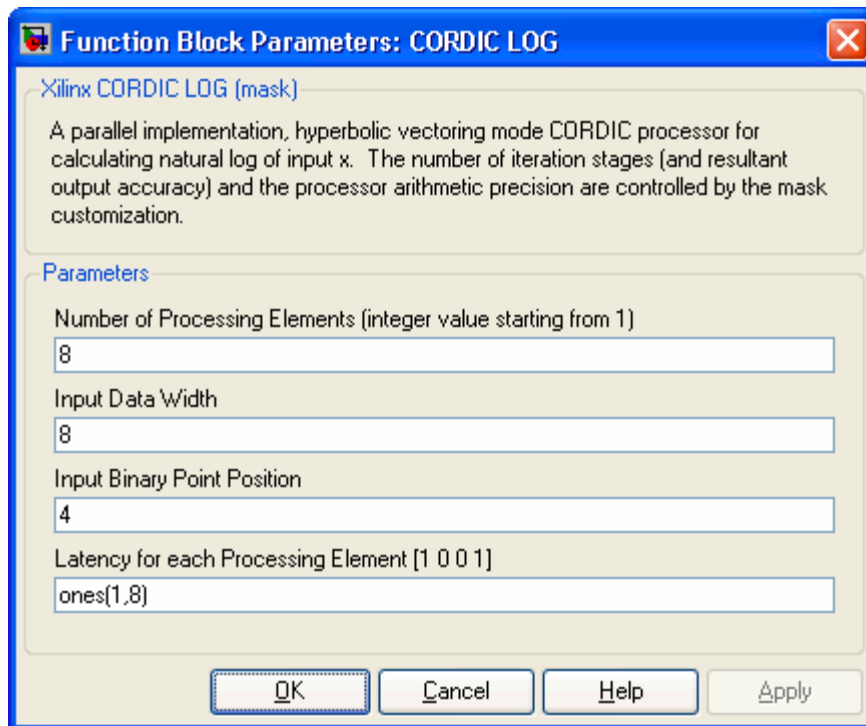


Figure 1: CORDIC LOG block parameters dialog box

Parameters specific to the block are:

- **Number of Processing Elements:** specifies the number of iterative stages used for hyperbolic rotation.
- **Input Data Width:** specifies the width of the input x. The input x should be a signed data type having the specified data width.
- **Input Binary Point Position:** specifies the binary point position for input x. The input x should be a signed data type having the specified binary point position.
- **Latency for each Processing element:** This parameter sets the pipeline latency after each iterative hyperbolic rotation stage.

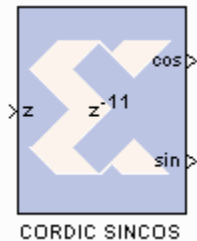
The latency of the CORDIC log block is calculated based on the formula specified below:

$$\text{Latency} = 2 + \text{Data Width} + \text{sum (latency of Processing Elements)}$$

Reference

- 1) J. E. Volder, "The CORDIC Trigonometric Computing Technique", IRE Trans. On Electronic Computers, Vol. EC-8, 1959, pp. 330-334.
- 2) J. S. Walther, "A Unified Algorithm for Elementary Functions", Spring Joint Computer Conference (1971) pp. 379-385.
- 3) Yu Hen Hu, "CORDIC-Based VLSI Architectures for Digital Signal Processing", IEEE Signal Processing Magazine, pp. 17-34, July 1992.

CORDIC SINCOS (Math)



The Xilinx CORDIC SINCOS reference block implements Sine and Cosine generator circuit using a fully parallel CORDIC (COordinate Rotation Digital Computer) algorithm in Circular Rotation mode.

That is, given input angle z , it computes the output cosine (z) and sine (z). The CORDIC processor is implemented using building blocks from the Xilinx blockset.

The CORDIC sine cosine algorithm is implemented in the following 3 steps:

1. **Coarse Angle Rotation:** The algorithm converges only for angles between $-\pi/2$ and $\pi/2$. If $z > \pi/2$, the input angle is reflected to the 1st quadrant by subtracting $\pi/2$ from the input angle. When $z < -\pi/2$, the input angle is reflected back to the 3rd quadrant by adding $\pi/2$ to the input angle. The sine cosine circuit has been designed to converge for all values of z , except for the most negative value.
2. **Fine Angle Rotation:** By setting x equal to $1/1.646760$ and y equal to 0, the rotational mode CORDIC processor yields cosine and sine of the input angle z .
3. **Co-ordinate Correction:** If there was a reflection applied in Step 1, this step applies the appropriate correction.

For $z > \pi/2$: using $z = t + \pi/2$, then

$$\sin(z) = \sin(t) \cdot \cos(\pi/2) + \cos(t) \cdot \sin(\pi/2) = \cos(t)$$

$$\cos(z) = \cos(t) \cdot \cos(\pi/2) - \sin(t) \cdot \sin(\pi/2) = -\sin(t)$$

For $z < -\pi/2$: using $z = t - \pi/2$, then

$$\sin(z) = \sin(t) \cdot \cos(-\pi/2) + \cos(t) \cdot \sin(-\pi/2) = -\cos(t)$$

$$\cos(z) = \cos(t) \cdot \cos(-\pi/2) - \sin(t) \cdot \sin(-\pi/2) = \sin(t)$$

Block Parameters Dialog Box

The block parameters dialog may be opened by double clicking on the icon in your Simulink model:

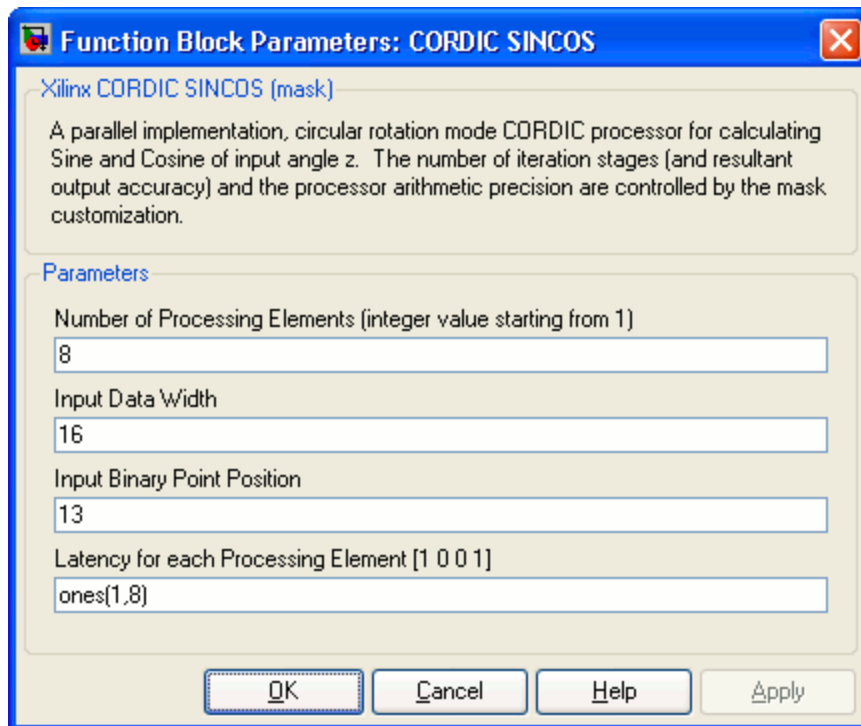


Figure 1: CORDIC SINCOS block parameters dialog box

Parameters specific to the block are:

- **Number of Processing Elements:** specifies the number of iterative stages used for linear rotation.
- **Input Data Width:** specifies the width of the input z . The input z should be signed data type with the same data width.
- **Input Binary Point Position:** specifies the binary point position for input z . The input z should be signed data type with the same binary point position. The binary point should be chosen to provide enough bits for representing $\pi/2$.
- **Latency for each Processing element:** This parameter sets the pipeline latency after each iterative circular rotation stage.

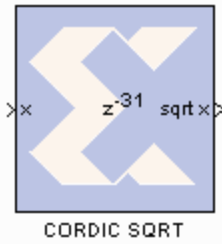
The latency of the CORDIC SINCOS block is calculated based on the formula specified below:

$$\text{Latency} = 3 + \text{sum}(\text{latency of Processing Elements})$$

Reference

- 1) J. E. Volder, "The CORDIC Trigonometric Computing Technique", IRE Trans. On Electronic Computers, Vol. EC-8, 1959, pp. 330-334.
- 2) J. S. Walther, "A Unified Algorithm for Elementary Functions", Spring Joint Computer Conference (1971) pp. 379-385.
- 3) Yu Hen Hu, "CORDIC-Based VLSI Architectures for Digital Signal Processing", IEEE Signal Processing Magazine, pp. 17-34, July 1992.

CORDIC SQRT (Math)



The Xilinx CORDIC SQRT reference block implements a square root circuit using a fully parallel CORDIC (COordinate Rotation DIgital Computer) algorithm in Hyperbolic Vectoring mode.

That is, given input x , it computes the output $\sqrt{x}$. The CORDIC processor is implemented using building blocks from the Xilinx blockset.

The square root is calculated indirectly by the CORDIC algorithm by applying the identity listed below.

$$\sqrt{w} = \sqrt{\{(w + 0.25)^2 - (w - 0.25)^2\}}$$

The CORDIC square root algorithm is implemented in the following 4 steps:

1. **Co-ordinate Rotation:** The CORDIC algorithm converges only for positive values of x . If $x < 0$, the input data is converted to a non-negative number. If $x = 0$, a zero detect flag is passed to the co-ordinate correction stage. The square root circuit has been designed to converge for all values of x , except for the most negative value.
2. **Normalization:** The CORDIC algorithm converges only for x between 0.25 (inclusive) and 1. During normalization, the input x is shifted to the left till it has a 1 in the most significant non-signed bit. If the left shift results in an odd number of shift values, a right shift is performed resulting in an even number of left shifts. The shift value is divided by 2 and passed on to the co-ordinate correction stage. The square root is derived using the identity $\sqrt{w} = \sqrt{\{(w + 0.25)^2 - (w - 0.25)^2\}}$. Based on this identity the input x gets mapped to, $X = x + 0.25$ and $Y = x - 0.25$.
3. **Hyperbolic Rotations:** For $\sqrt{x^2 - y^2}$ calculation, the resulting vector is rotated through progressively smaller angles, such that y goes to zero.
4. **Co-ordinate Correction:** If the input was negative and a left shift was applied to x , this step assigns the appropriate sign to the output and multiplies it with $2^{-\text{shift}}$. If the input was zero, the zero detect flag is used to set the output to 0.

Block Parameters Dialog Box

The block parameters dialog may be opened by double clicking on the icon in your Simulink model:

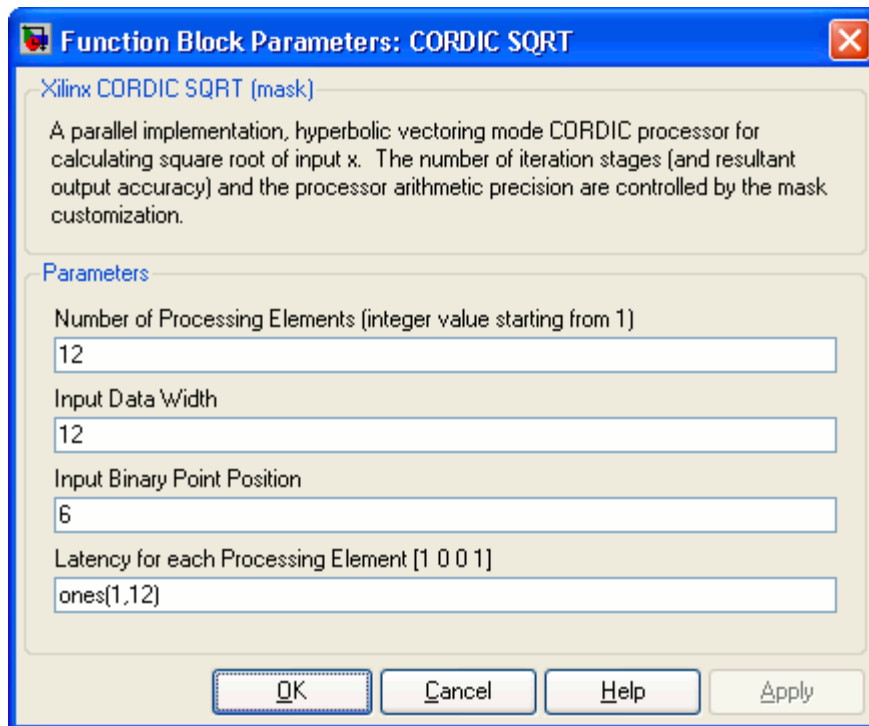


Figure 1: CORDIC SQRT block parameters dialog box

Parameters specific to the block are:

- **Number of Processing Elements:** specifies the number of iterative stages used for linear rotation.
- **Input Data Width:** specifies the width of the inputs x. The input x should be signed data type with the same data width.
- **Input Binary Point Position:** specifies the binary point position for input x. The input x should be signed data type with the specified binary point position.
- **Latency for each Processing element:** This parameter sets the pipeline latency after each iterative hyperbolic rotation stage.

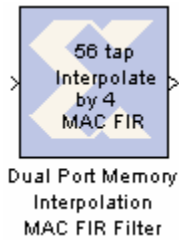
The latency of the CORDIC square root block is calculated based on the formula specified below:

$$\begin{aligned} \text{Latency} = & 7 + (\text{data width} - \text{binary point}) \\ & + \text{mod} \{ (\text{data width} - \text{binary point}), 2 \} \\ & + \text{sum} (\text{latency of Processing Elements}) \end{aligned}$$

Reference

- 1) J. E. Volder, "The CORDIC Trigonometric Computing Technique", IRE Trans. On Electronic Computers, Vol. EC-8, 1959, pp. 330-334.
- 2) J. S. Walther, "A Unified Algorithm for Elementary Functions", Spring Joint Computer Conference (1971) pp. 379-385.
- 3) Yu Hen Hu, "CORDIC-Based VLSI Architectures for Digital Signal Processing", IEEE Signal Processing Magazine, pp. 17-34, July 1992.

Dual Port Memory Interpolation MAC FIR Filter (DSP)



Dual Port Memory
Interpolation
MAC FIR Filter

The Xilinx Dual Port Memory Interpolation MAC FIR filter reference block implements a multiply-accumulate based FIR filter to perform a user-selectable interpolation. One dedicated multiplier and one Dual Port Block RAM are used in the n-tap filter. The filter configuration helps illustrate a cyclic RAM buffer technique for storing coefficients and data samples in a single block ram. The filter allows users to select the interpolation factor they require. The Virtex FPGA family (and Virtex family derivatives) provide dedicated circuitry for building fast, compact adders, multipliers, and flexible memory architectures. The filter design takes advantage of these silicon features by implementing a design that is compact and resource-efficient.

Annotation is provided in the filter design subsystems. Please click on the reference block and refer to the documentation seen in the block model itself for more specific implementation details.

Block Parameters Dialog Box

The block parameters dialog may be opened by double clicking on the icon in your Simulink model:

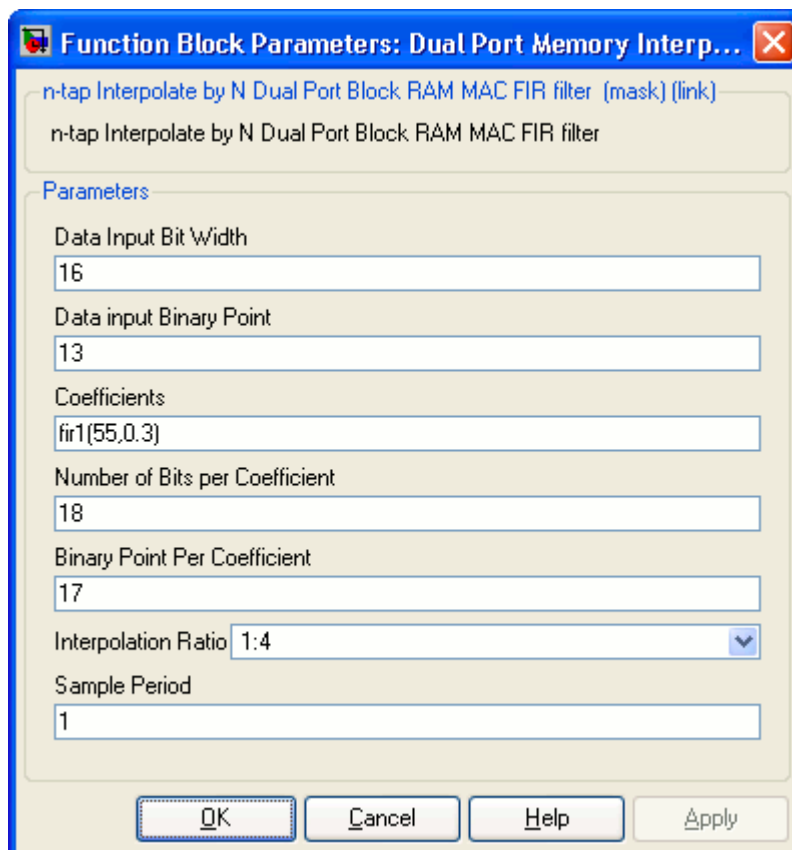


Figure 1: Dual Port Memory Interpolation MAC FIR Filter block parameters dialog box

Parameters specific to this reference block are:

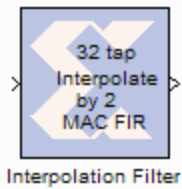
- Data Input Bit Width: Width of input sample.
- Data Input Binary Point: Binary point location of input.

- **Coefficients:** Specify coefficients for the filter. Number of taps is inferred from size of coefficient vector.
- **Number of Bits per Coefficient:** Bit width of each coefficient.
- **Binary Point Per Coefficient:** Binary point location for each coefficient.
- **Interpolation Ratio:** Select the Interpolation Ratio of the filter (2 to 10, inclusive).
- **Sample Period:** Sample period of input.

Reference

J. Hwang and J. Ballagh. *Building Custom FIR Filters Using System Generator*. 12th International Field-Programmable Logic and Applications Conference (FPL). Montpellier, France, September 2002. Lecture Notes in Computer Science 2438.

Interpolation Filter (DSP)

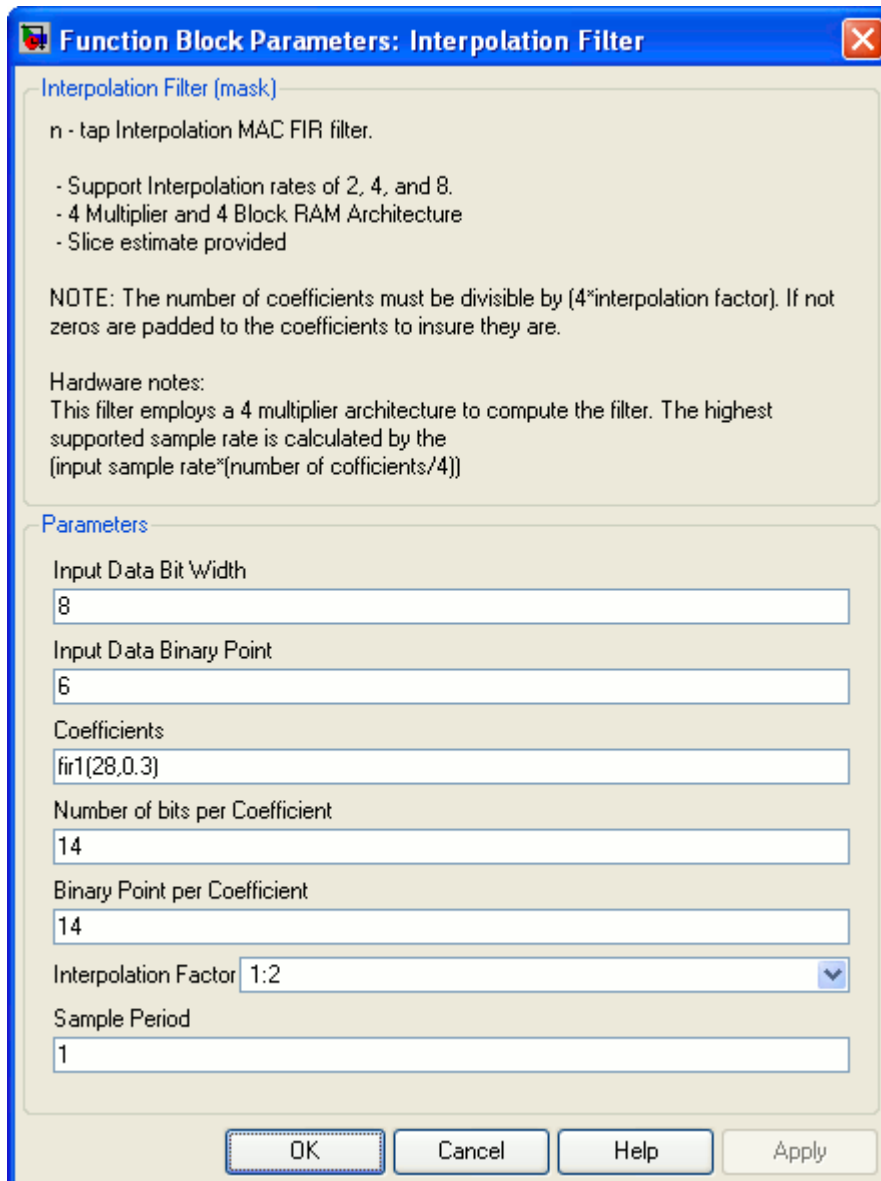


The Xilinx n-tap Interpolation Filter reference block implements a multiply-accumulate based FIR filter to perform a user selected interpolation. One dedicated multiplier and one Dual Port Block RAM are used in the n-tap filter. The filter configuration helps illustrate a cyclic RAM buffer technique for storing coefficients and data samples in a single block ram. The filter allows users to select the interpolation factor they require. The Virtex FPGA family (and Virtex family derivatives) provide dedicated circuitry for building fast, compact adders, multipliers, and flexible memory architectures. The filter design takes advantage of these silicon features by implementing a design that is compact and resource efficient.

Annotation is provided in the filter design subsystems. Please click on the reference block and refer to this documentation seen in the block model itself for more specific implementation details.

Block Parameters Dialog Box

The block parameters dialog may be opened by double clicking on the icon in your Simulink model:



Function Block Parameters: Interpolation Filter

Interpolation Filter (mask)

n - tap Interpolation MAC FIR filter.

- Support Interpolation rates of 2, 4, and 8.
- 4 Multiplier and 4 Block RAM Architecture
- Slice estimate provided

NOTE: The number of coefficients must be divisible by (4*interpolation factor). If not zeros are padded to the coefficients to insure they are.

Hardware notes:
This filter employs a 4 multiplier architecture to compute the filter. The highest supported sample rate is calculated by the
(input sample rate*(number of coefficients/4))

Parameters

Input Data Bit Width
8

Input Data Binary Point
6

Coefficients
fir1(28,0.3)

Number of bits per Coefficient
14

Binary Point per Coefficient
14

Interpolation Factor 1:2

Sample Period
1

OK Cancel Help Apply

Figure 1: Interpolation Filter block parameters dialog box

Parameters specific to this reference block are:

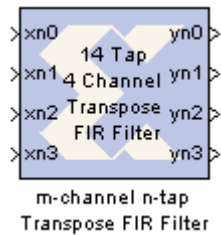
- **Input Data Bit Width:** Width of input sample.
- **Input Data Binary Point:** Binary point location of input.
- **Coefficients:** Specify coefficients for the filter. Number of taps is inferred from size of coefficient vector.
- **Number of Bits per Coefficient:** Bit width of each coefficient.
- **Binary Point per Coefficient:** Binary point location for each coefficient.
- **Interpolation Factor:** Select the Interpolation Ratio of the filter. Range from 2 to 10.
- **Sample Period:** Sample period of input.

Reference

J. Hwang and J. Ballagh. *Building Custom FIR Filters Using System Generator*. 12th International Field-Programmable Logic and Applications Conference (FPL). Montpellier, France, September 2002.

Lecture Notes in Computer Science 2438.

m-channel n-tap Transpose FIR Filter (DSP)



The Xilinx m-channel n-tap Transpose FIR Filter uses a fully parallel architecture with Time Division Multiplexing. The Virtex FPGA family (and Virtex family derivatives) provide dedicated shift register circuitry called the SRL16E, which are exploited in the architecture to achieve optimal implementation of the multi-channel architecture. The Time Division Multiplexer and Time Division Demux can be selected to be implemented or not. Embedded Multipliers are used for the multipliers.

As the number of coefficients changes so to does the structure underneath as it is a dynamically built model.

Annotation is provided in the filter design subsystems. Please click on the reference block and refer to this documentation seen in the block model itself for more specific implementation details.

Block Parameters Dialog Box

The block parameters dialog may be opened by double clicking on the icon in your Simulink model:

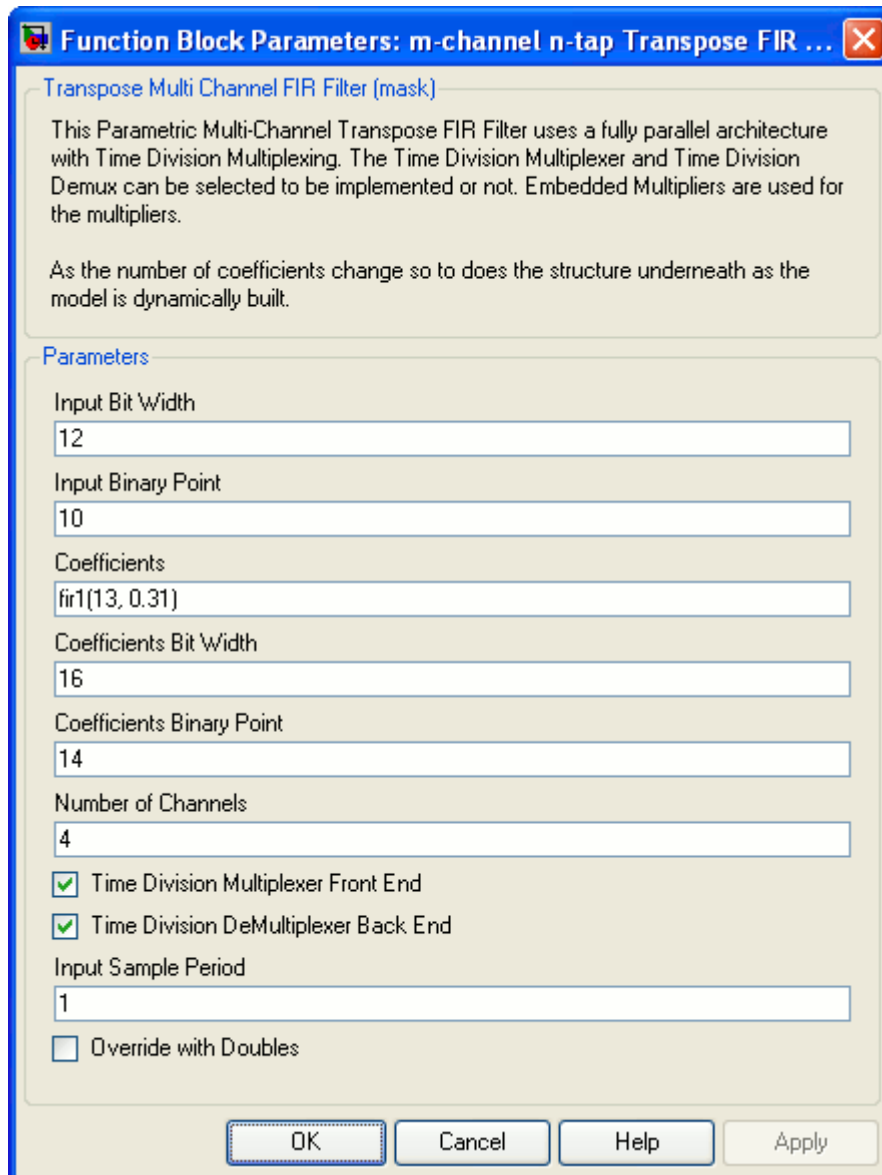


Figure 1: m-channel n-tap Transpose FIR Filter block parameters dialog box

Parameters specific to this reference block are:

- **Data Input Bit Width:** Width of input sample.
- **Data Input Binary Point:** Binary point location of input.
- **Coefficients:** Specify coefficients for the filter. Number of taps is inferred from size of coefficient vector.
- **Coefficients Bit Width:** Bit width of each coefficient.
- **Coefficients Binary Point:** Binary point location for each coefficient.
- **Number of Channels:** Specify the number of channels desired. There is no limit to the number of channels supported.
- **Time Division Multiplexer Front End:** The TDM front-end circuit can be implemented or not (if the incoming data is already TDM)
- **Time Division DeMultiplexer Back End:** The TDD back-end circuit can be implemented or not (if you desire a TDM output). This is useful if the filter feeds another multi-channel structure.
- **Input Sample Period:** Sample period of input.

Other parameters used by this block are described in the Common Parameters section at the beginning of this chapter.

Mealy State Machine (Control)

A “Mealy machine” is a finite state machine whose output is a function of state transition, i.e., a function of the machine’s current state and current input. A Mealy machine can be described with the following block diagram:

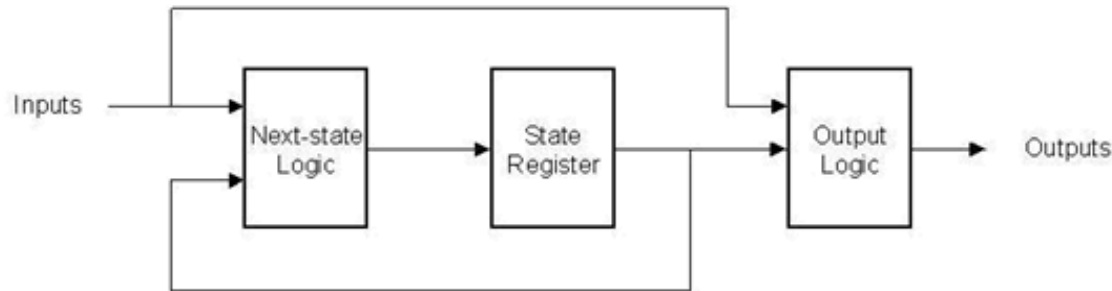
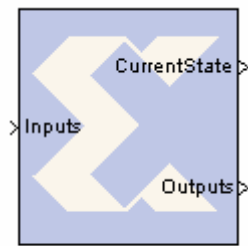


Figure 1: Mealy machine block diagram



Registered Mealy State Machine

There are many ways to implement such state machines in System Generator (e.g., using the MCode block to implement the transition function, and registers to implement state variables). This reference block provides a method for implementing a Mealy machine using block and distributed memory. The implementation is very fast and efficient. For example, a state machine with 8 states, 1 input, and 2 outputs that are registered can be realized with a single block RAM that runs at more than 150 MHz in a Xilinx Virtex device.

The transition function and output mapping are each represented as an $N \times M$ matrix, where N is the number of states, and M is the size of the input alphabet (e.g., $M = 2$ for a binary input). It is convenient to number rows and columns from 0 to $N - 1$ and 0 to $M - 1$ respectively. Each state is represented as an unsigned integer from 0 to $N - 1$, and each alphabet character is represented as an unsigned integer from 0 to $M - 1$. The row index of each matrix represents the current state, and the column index represents the input character

For the purpose of discussion, let F be the $N \times M$ transition function matrix, and O be the $N \times M$ output function matrix. Then $F(i,j)$ is the next state when the current state is i and the current input character is j , and $O(i,j)$ is the corresponding output of the Mealy machine.

Example

Consider the problem of designing a Mealy machine to recognize the pattern '1011' in a serial stream of bits. The state transition diagram and equivalent transition table are shown below.

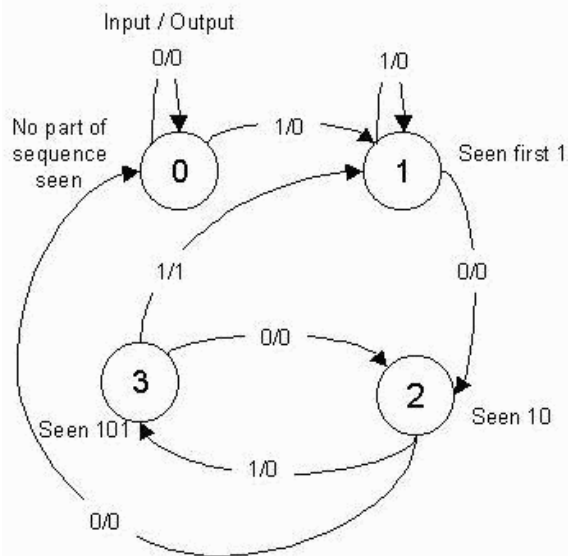


Figure 2: Example transition diagram for Mealy machine

Next State/Output Table

Current State	If Input = 0	If Input = 1
0	0, 0	1, 0
1	2, 0	1, 0
2	0, 0	3, 0
3	2, 0	1, 1

Cell Format: Next State, Output

Figure 3: Example transition table for Mealy machine

The table lists the next state and output that result from the current state and input. For instance, if the current state is 3 and the input is 1, the next state is 1 and the output is 1, indicating the detection of the desired sequence.

The Mealy State Machine block is configured with next state and output matrices obtained from the next state/output table discussed above. These matrices are constructed as follows:

Next State/Output Table

Current State	If Input = 0	If Input = 1
0	0	1
1	2	1
2	0	3
3	2	1

Cell Format: Next State, Output



Next State Matrix

Output Matrix

Figure 4: Construction of next state and output matrices

Rows of the matrices correspond to states, and columns correspond to input values.

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

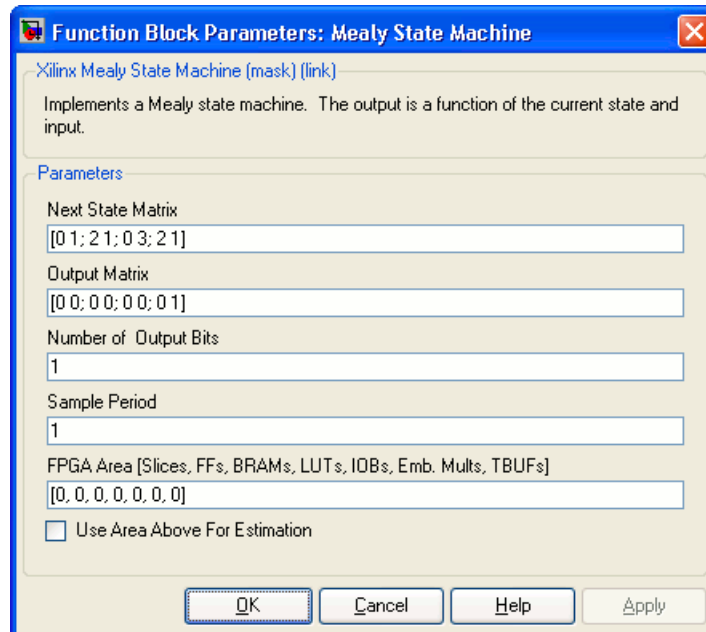


Figure 5: Mealy state machine block parameters dialog box

The next state logic, state register, and output logic are implemented using high speed dedicated block RAM. The output logic is implemented using a distributed RAM configured as a lookup table, and therefore has zero latency.

The number of bits used to implement a Mealy state machine is given by the equations:

$$\begin{aligned} \text{depth} &= (2^k)(2^i) = 2^{k+i} \\ \text{width} &= k + o \\ N &= \text{depth} * \text{width} = (k + o)(2^{k+i}) \end{aligned}$$

where

N = total number of block RAM bits
 k =
 s = number of states
 i = number of input bits
 o = number of output bits

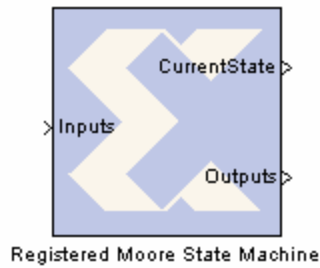
The following table gives examples of block RAM sizes necessary for various state machines:

Number of States	Number of Input Bits	Number of Output Bits	Block RAM Bits Needed
2	5	10	704

4	1	2	32
8	6	7	5120
16	5	4	4096
32	4	3	4096
52	1	11	2176
100	4	5	24576

The block RAM width and depth limitations are described in the online help for the Single Port RAM block.

Moore State Machine (Control)



A “Moore machine” is a finite state machine whose output is only a function of the machine’s current state. A Moore state machine can be described with the following block diagram:

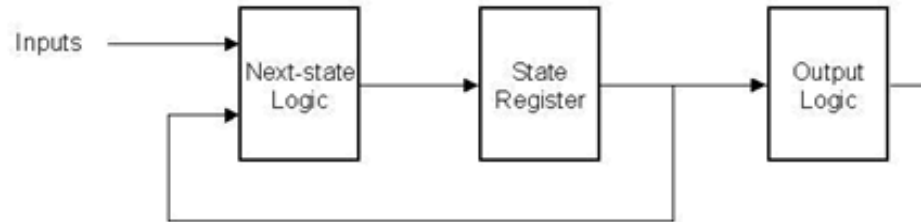


Figure 1: Moore State Machine block diagram

There are many ways to implement such state machines in System Generator (e.g., using the MCode block to implement the transition function, and registers to implement state variables). This reference block provides a method for implementing a Moore machine using block and distributed memory. The implementation is very fast and efficient. For example, a state machine with 8 states, 1 input, and 2 outputs that are registered can be realized with a single block RAM that runs at more than 150 MHz in a Xilinx Virtex device.

The transition function and output mapping are each represented as an $N \times M$ matrix, where N is the number of states, and M represents the number of possible input values (e.g., $M = 2$ for a one bit input). It is convenient to number rows and columns from 0 to $N - 1$ and 0 to $M - 1$ respectively. Each state is represented as an unsigned integer from 0 to $N - 1$, and each alphabet character is represented as an unsigned integer from 0 to $M - 1$. The row index of each matrix represents the current state, and the column index represents the input character.

For the purpose of discussion, let F be the $N \times M$ transition function matrix, and O be the $N \times M$ output function matrix. Then $F(i,j)$ is the next state when the current state is i and the current input character is j , and $O(i,j)$ is the corresponding output of the Mealy machine.

Example

Consider the problem of designing a Moore machine to recognize the pattern '1011' in a serial stream of bits. The state transition diagram and equivalent transition table are shown below.

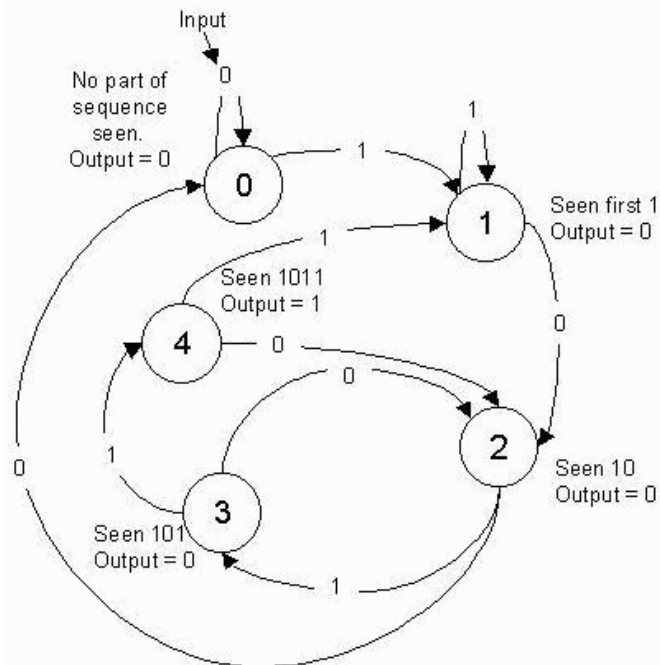


Figure 2: Moore State Machine example transition diagram
Next State/Output Table

Current State	If Input = 0	If Input = 1	Output
0	0	1	0
1	2	1	0
2	0	3	0
3	2	4	0
4	2	1	1

Figure 3: Moore State Machine example transition table

The table lists the next state and output that result from the current state and input. For example, if the current state is 4, the output is 1 indicating the detection of the desired sequence, and if the input is 1 the next state is state 1.

The Registered Moore State Machine block is configured with next state matrix and output array obtained from the next state/output table discussed above. They are constructed as follows:

Next State/Output Table

Current State	If Input = 0	If Input = 1	Output
0	0	1	0
1	2	1	0
2	0	3	0
3	2	4	0
4	2	1	1

$$\begin{bmatrix} 0 & 1 \\ 2 & 1 \\ 0 & 3 \\ 2 & 4 \\ 2 & 1 \end{bmatrix}$$

Next State Matrix

$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

Output Array

Figure 4: Construction of Next State and Output matrices

The rows of the matrices correspond to the current state. The next state matrix has one column for each input value. The output array has only one column since the input value does not affect the output of the state machine. .

Block Parameters Dialog Box

The block parameters dialog can be invoked by double-clicking the icon in your Simulink model.

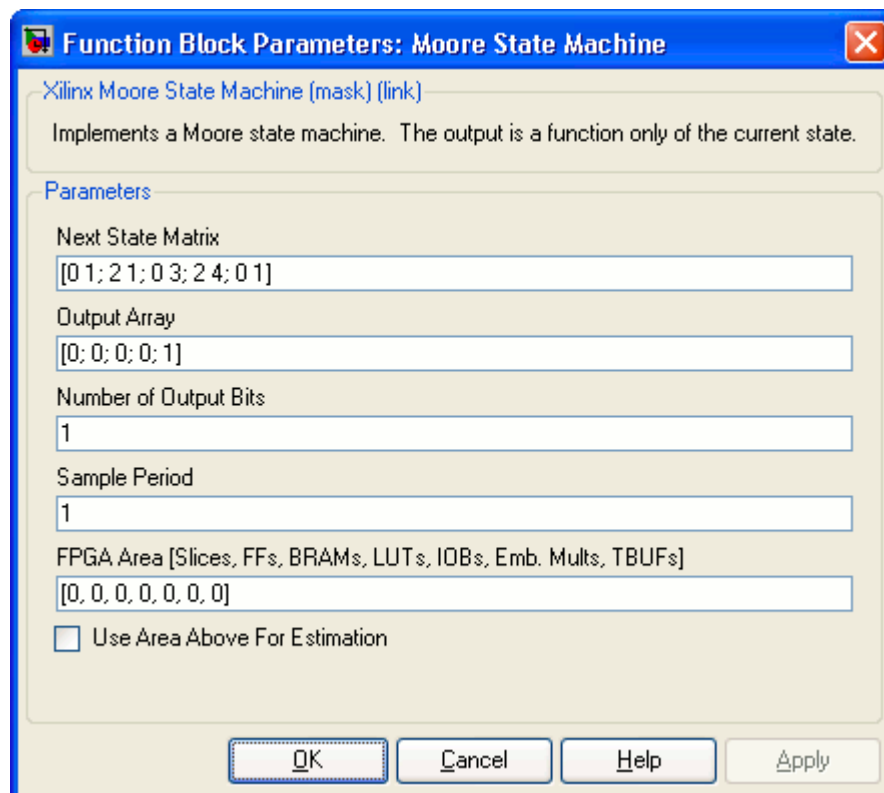


Figure 5: Moore State Machine block parameters dialog box

The next state logic and state register in this block are implemented with high speed dedicated block RAM. The output logic is implemented using a distributed RAM configured as a lookup table, and therefore has zero latency.

The number of bits used to implement a Moore state machine is given by the equations:

$$d_s = (2^k)(2^i) = 2^{k+i}$$

$$w_s = k$$

$$N_s = d_s * w_s = (k)(2^{k+i})$$

where

N_s = total number of next state logic block RAM bits

k =

d_s = depth of state logic block RAM

w_s = width of state logic block RAM

s = number of states

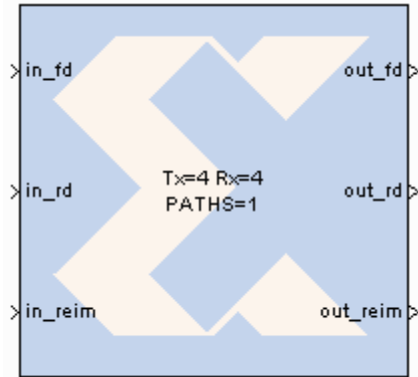
i = number of input bits

The following table gives examples of block RAM sizes necessary for various state machines:

Number of States	Number of Input Bits	Block RAM Bits Needed
2	5	64
4	1	8
8	6	1536
16	5	2048
32	4	2560
52	1	768
100	4	14336

The block RAM width and depth limitations are described in the core datasheet for the Single Port Block Memory.

Multipath Fading Channel Model



Multipath Fading Channel Model

The Multipath Fading Channel Model block implements a model of a fading communication channel. The model supports both Single Input/Single Output (SISO) and Multiple Input/Multiple Output (MIMO) channels. The model provides functionality similar to the Simulink 'Multipath Rayleigh Fading Channel' block in a hardware realizable form. This enables high speed hardware co-simulation of entire communication links.

Theory

The block implements the *Kronecker model*. This model is suitable for systems with antenna arrays not exceeding four elements. The primary model parameters are:

- M_T : The number of antennas in the transmit array. For SISO systems this is 1.
- M_R : The number of antennas in the receive array. For SISO systems this is 1.
- N : The number of discrete paths between the arrays. For frequency flat channels this is 1.

The model can be represented by the discrete time equation:

$$\mathbf{y}(nT) = \sum_{k=1}^N g_k \mathbf{H}_k(nT) \mathbf{x}(nT - d_k)$$

Where:

- $\mathbf{x}(\cdot)$: Transmit symbol column vector (M_T complex elements, time varying).
- T : Sample interval.
- n : Sample index.
- d_k : Delay for path k .
- $\mathbf{H}_k(\cdot)$: Channel coefficient matrix ($M_R \times M_T$ complex elements, time varying).
- g_k : Gain for path k .
- $\mathbf{y}(\cdot)$: Receive symbol column vector (M_R complex elements, time varying).

The channel coefficient matrix can be further defined in terms of the spatial covariance matrices of the antenna arrays:

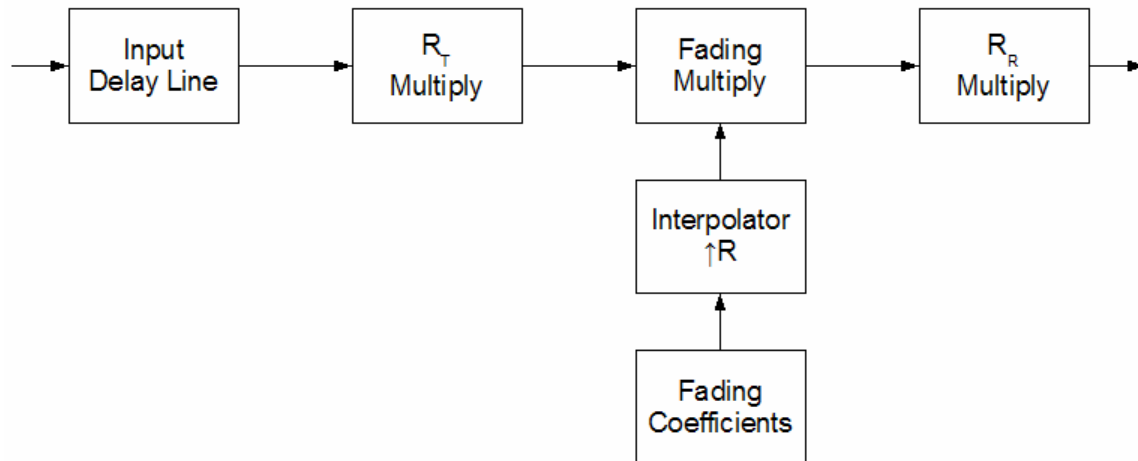
$$\mathbf{H}_k(nT) = \mathbf{R}_{R,k}^{1/2} \mathbf{H}_{U,k}(nT) \mathbf{R}_{T,k}^{1/2}$$

Where:

- $\mathbf{R}_{T,k}$: Transmit array spatial covariance matrix for path k .
- $\mathbf{H}_{U,k}(\cdot)$: Uncorrelated channel coefficient matrix for path k ($M_R \times M_T$ elements, time varying).
- $\mathbf{R}_{R,k}$: Receive array spatial covariance matrix for path k .

Implementation

The above equations can be rephrased as sparse matrix operations. This allows the elimination of the path summation. The model can then be implemented as follows:



Block Parameters Dialog Box

The block parameters dialog may be opened by double clicking on the icon in your Simulink model. The dialog is split into four panes:

Paths Pane

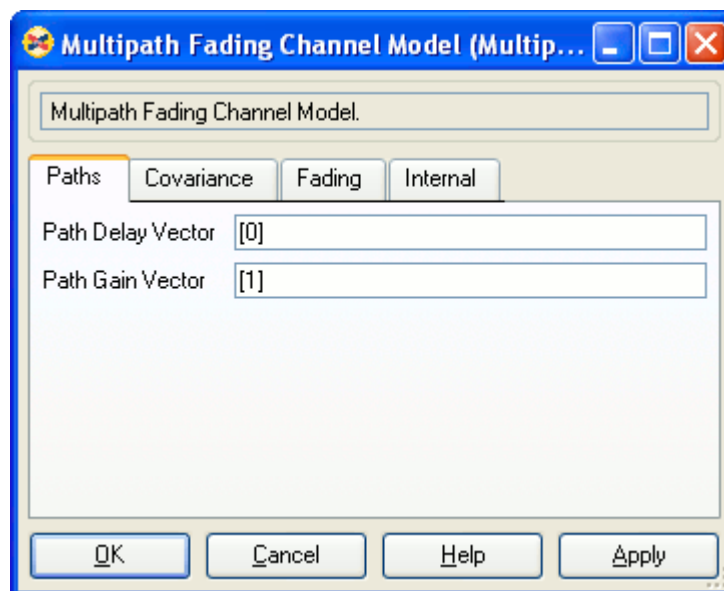


Figure 1: Paths Pane

- Path Delay Vector: Specify the delay spread for each path in the model. Each element represents the number of samples to delay the path by. The value must be an N element vector.
- Path Gain Delay Vector: Specify the gain for each path in the model. Each element represents the linear gain of the path. The value must be an N element vector.

Covariance Pane

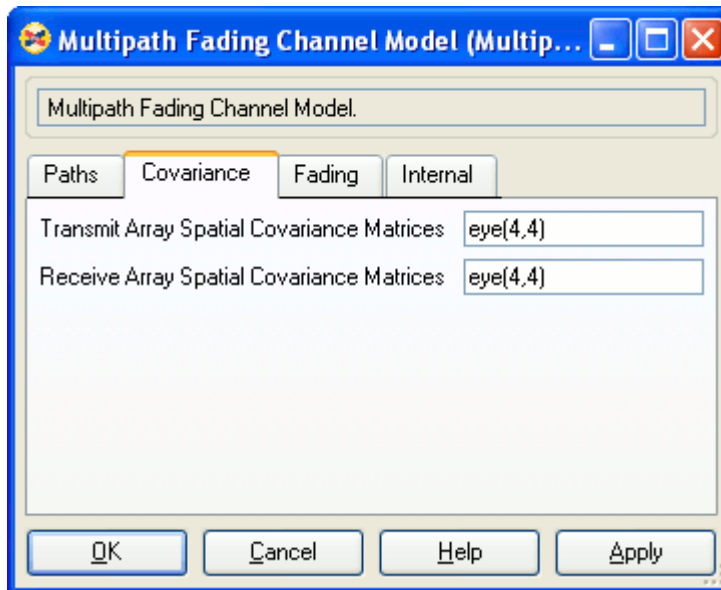


Figure 2: Covariance Pane

To support frequency selective channels ($N > 1$), these parameters can be specified as three dimensional arrays. The first two dimensions specify the square covariance matrix, the third specifies the path. If a two dimensional array is specified for a frequency selective channel, it is automatically replicated to produce a three dimensional array. The third dimension is optional for frequency flat ($N = 1$) channels.

- Transmit Array Spatial Covariance Matrices: Specify the transmit antenna array covariance matrix for each path. The value can be a $M_T \times M_T$ matrix, or a $M_T \times M_T \times N$ array.
- Receive Array Spatial Covariance Matrices: Specify the receive antenna array covariance matrix for each path. The value can be a $M_R \times M_R$ matrix, or a $M_R \times M_R \times N$ array.

Fading Pane

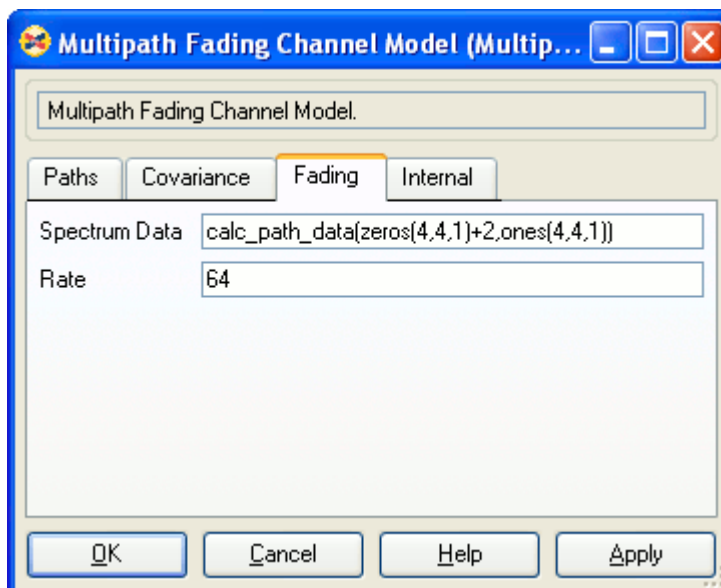


Figure 3: Fading Pane

- Spectrum Data: Specify the fading phase and frequency response of each physical path. The number of physical paths is the product of the number of discrete paths (N), and the number of paths between each element of the transmit and receive antenna arrays ($M_T \times M_R$). Spectrum data must be a multidimensional structure with dimensions $M_R \times M_T \times N$.

- Rate: Specify the interpolation rate from maximum Doppler frequency (F_{DMAX}) to channel sample frequency (F_S). It can be determined as follows:

$$R = \left\lceil \frac{F_S}{256 \cdot F_{DMAX}} \right\rceil$$

Internal Pane

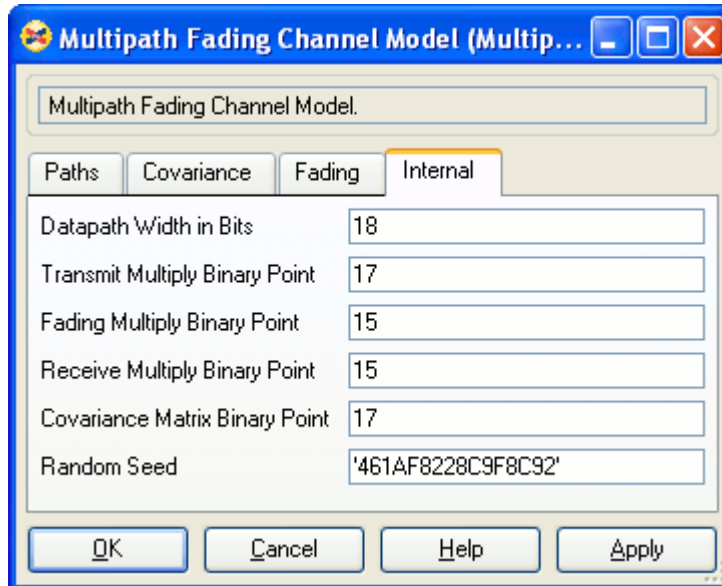


Figure 4: Internal Pane

- Datapath Width: Specify the width in bits of all internal datapaths.
- Transmit Multiply Binary Point: Specify the binary point position at the output of the R_T multiply block.
- Fading Multiply Binary Point: Specify the binary point position at the output of the fading multiply block.
- Receive Multiply Binary Point: Specify the binary point position at the output of the R_R multiply block.
- Covariance Matrix Multiply Binary Point: Specify the binary point position of the covariance matrix coefficients.
- Random Seed: Specify the 61-bit (16 hexadecimal digits) seed of the phase noise random number generator.

Functions

The model includes two Matlab functions to simply parameter generation.

create_r_la

The 'create_r_la(M, P, phi0, d, lambda, AS)' function generates a covariance matrix from steering vectors as described in [1].

- M: Specify the number of antennas in the array (transmit or receive).
- P: Specify the number of random paths to integrate over to generate the matrix (a value of 50000 gives good results).
- phi0: Specify the mean angle of departure (for transmit arrays) or arrival (for receive arrays). Value is in radians.
- d: Specify antenna spacing as a vector of antenna positions along a baseline. If this value is specified as a scalar value, the function assumes a uniform linear array (ULA) with the elements evenly distributed about the baseline origin.

- `lambda`: Specify the wavelength, in meters.
- `AS`: Specify the angular spread around the mean angle in radians.

For example, to create a matrix for a 3 element ULA with element spacing of $\lambda/2$ at 2GHz with an angular spread of 15°:

```
lambda=2.0e9/2.99e8;
create_r_la(3,50000,0,lambda/2,lambda,15*(2*pi/360))
```

calc_path_data

The `'calc_path_data(spec_type, spec_fd)'` function generates spectrum data for a model.

- `spec_type`: Specify the spectrum type for each physical path in the model. This value must be a multidimensional array with dimensions $M_R \times M_T \times N$. Each element specifies the spectrum type for the physical path.
- `spec_fd`: Specify the spectrum Doppler frequency for each physical path, normalized to the maximum Doppler frequency (F_{DMAX}). This value can be a multidimensional array with dimensions $M_R \times M_T \times N$ or scalar, in which case the value is applied to all physical paths. If omitted a value of unity is assumed.

The value of each spectrum type element specifies the spectrum shape to use for that physical path.

Four spectrum types are supported.

- Type 0: Specify a null physical path. The path coefficients are zero, and the path exhibits no transmission.
- Type 1: Specify an impulse physical path. An impulse path has a single impulse in its spectrum. They can be used to represent the line-of-sight (LOS) paths in a channel model (such as required by Rician channels).
- Type 2: Specify a classic spectrum physical path. The classic spectrum is also known as the Jakes or Clarke spectrum. It is used to model wireless links with mobile stations [2] [3] [4] and is defined as:

$$S(f) \propto \frac{1}{\sqrt{1 - \left(\frac{f}{f_d}\right)^2}} \quad |f| \leq f_d$$

$$S(f) = 0 \quad elsewhere$$

- Type 3: Specify a rounded spectrum physical path. The rounded spectrum is used to model wireless links with fixed stations [5] and is defined as:

$$S(f) \propto 1 - 1.72 \left(\frac{f}{f_d}\right)^2 + 0.785 \left(\frac{f}{f_d}\right)^4 \quad |f| \leq f_d$$

$$S(f) = 0 \quad elsewhere$$

Once generated, each spectrum is normalized to unity power.

For example, to create and plot spectrum data for a $M_T=4$, $M_R=3$ and $N=2$ channel, where the two paths combine to give Rician fading (i.e. impulse and classic). We assume that the mobile station (MS) is receding from the base station (BS) at $0.707 \times v_{MS}$ (giving $f_d=0.707$ for the LOS physical paths):

```
Mt=4; Mr=3; N=2;
spec_type=cat(3,ones(Mr,Mt)*1,ones(Mr,Mt)*2);
spec_fd =cat(3,ones(Mr,Mt)*0.707,ones(Mr,Mt)*1);
spec_data=calc_path_data(spec_type,spec_fd);
plot([spec_data.spectrum]);
```

Data Format

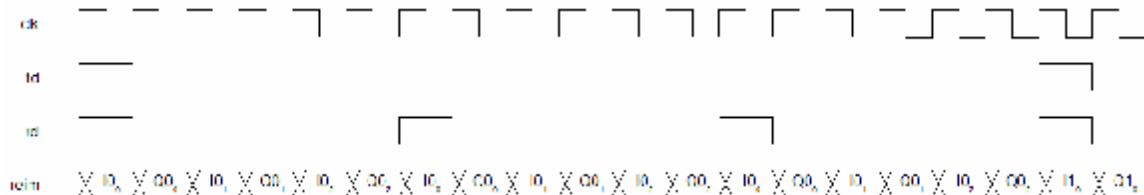
Internally the model uses a three signal interface for transferring complex vector quantities between blocks. This interface allows matrix/vector operations to be chained together. Vectors are transferred as streams of interleaved real and imaginary samples tagged with frame and repetition handshaking

signals. This interface allows vectors to be repeated multiple times per frame. This feature can be used to simplify matrix-vector multiplies, where the vector values are required repeatedly, once per matrix row.

The three signal interface is as follows:

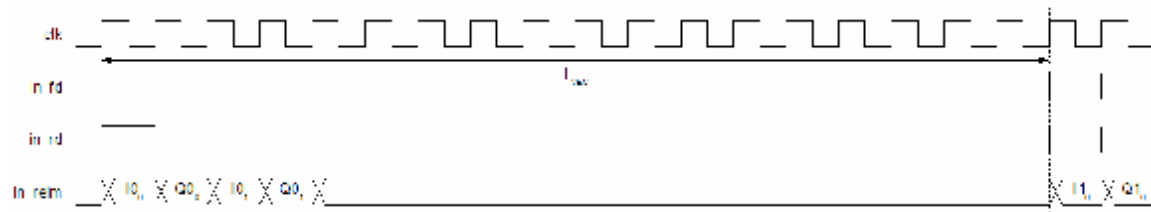
- reim: Stream of interleaved real and imaginary (I and Q) samples for each vector. Potentially each vector is transferred multiple times, as indicated by the rd signal.
- fd: Indicates the start of each vector frame.
- rd: Indicates the start of each vector repetition.

The diagram below shows how a 3-element vector would be represented before multiplication by a 3x3 matrix. The vector is repeated 3 times (once for each matrix row) greatly simplifying the multiplication logic.



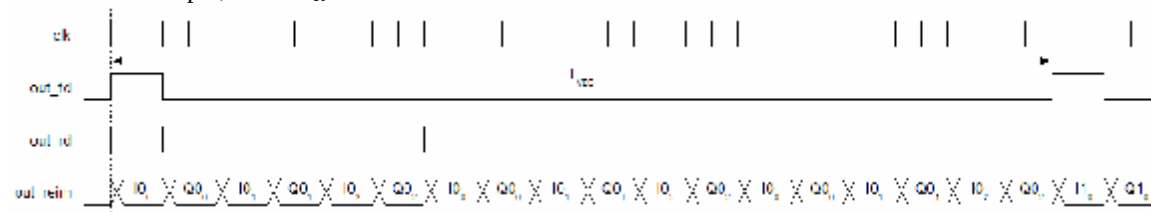
Input

Input data is presented on the `in_fd`, `in_rd`, and `in_reim` ports. Vector repetition is not required at the input, hence the `in_rd` signal is ignored, and only the first $2 \times M_T$ samples are used. For example, for a $M_T=2$ channel:



Output

Output data is presented on the out_fd, out_rd, and out_reim ports. The data is repeated throughout the frame. For example, for a $M_R=3$ channel:



Timing

The number of samples between successive fd pulses (T_{VEC}), must be sufficient for the internal blocks to process the data. The number of cycles required by each block is a function of the M_T , M_R , N , and $RATE$ parameters as follows:

- RT Multiply: Requires $2 \times M_T \times M_T \times N$ cycles
- Fading Multiply: Requires $2 \times M_T \times M_R \times N \times \text{ceil}(64/\text{RATE})$ cycles
- RR Multiply: Requires $2 \times M_R \times M_R \times N$ cycles

Hence, the minimum value of T_{VEC} is:

$$T_{VEC} = 2N \max \left(\max(M_T, M_R)^2, M_T M_R \left\lceil \frac{64}{RATE} \right\rceil \right)$$

The model will produce an error during simulation if this constraint is not met.

Initialization

The model requires approximately $3 \times R$ input frames for the fading coefficient generator to initialize. During this period the channel coefficients, and consequentially the output data, will be zero.

Demonstrations

Two demonstrations are included that show how the model can be used. Each includes notes on how parameters can be calculated.

- SISO Channel Model (Open this model): A demo showing a SISO channel based on 3GPP TS 25.104, Annex B.2, Case 4.
- MIMO Channel Model (Open this model): A demo showing a frequency flat MIMO channel.

Hardware Co-simulation Example

An example of how to use the model for hardware co-simulation is included in the

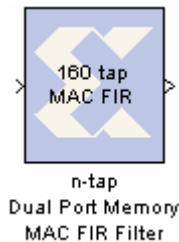
`<path_to_sysgen>/examples/mfcm_hwcossim` directory. The directory contains three files:

- `mfcm_hw.mdl` (Open this model): Model specifying the hardware component of the co-simulation design. Design consists of a shared memory for data input, a channel model, and a shared memory for data output.
- `mfcm_hw_cw.bit`: The 'mfcm_hw.mdl' design compiled for the XtremeDSP kit (XC2V3000-4).
- `mfcm_cossim.mdl` (Open this model): Model specifying the software component of the co-simulation. The shared memory blocks are used to pass packets of data to the hardware for processing, and to receive packets of processed data. By default this design will use the pre-generated 'mfcm_hw_cw.bit' – this will have to be regenerated for different hardware targets.

Reference

- [1] A. Forenza and R.W. Heath Jr. *Impact of Antenna Geometry on MIMO Communication in Indoor Clustered Channels*, Wireless Networking and Communications Group, ECE Department, The University of Texas at Austin.
- [2] 3GPP TS 25.101 V6.7.0 (2005-03) *Annex B, User Equipment (UE) radio transmission and reception (FDD)*, Technical Specification Group Radio Access Network, 3rd Generation Partnership Project.
- [3] 3GPP TS 25.104 V6.8.0 (2004-12) *Annex B, Base Station (BS) radio transmission and reception (FDD)*, Technical Specification Group Radio Access Network, 3rd Generation Partnership Project.
- [4] 3GPP TR 25.943 V6.0.0 (2004-12), *Deployment aspects*, Technical Specification Group Radio Access Network, 3rd Generation Partnership Project.
- [5] IEEE 802.16.3c-01/29r4 (2001-07-16) *Channel Models for Fixed Wireless Applications*, IEEE 802.16 Broadband Wireless Access Working Group.

n-tap Dual Port Memory MAC FIR Filter (DSP)



The Xilinx n-tap Dual Port Block RAM MAC FIR Filter reference block implements a multiply-accumulate based FIR filter. One dedicated multiplier and one dual port block RAM are used in the filter. The filter configuration illustrates a technique for storing coefficients and data samples in filter design. The Virtex FPGA family (and Virtex family derivatives) provide dedicated circuitry for building fast, compact adders, multipliers, and flexible memory architectures. The filter design takes advantage of these silicon features by implementing a design that is compact and resource efficient.

Annotation is provided in the filter design subsystems. Please click on the reference block and refer to this documentation seen in the block model itself for more specific implementation details.

Block Parameters Dialog Box

The block parameters dialog may be opened by double clicking on the icon in your Simulink model:

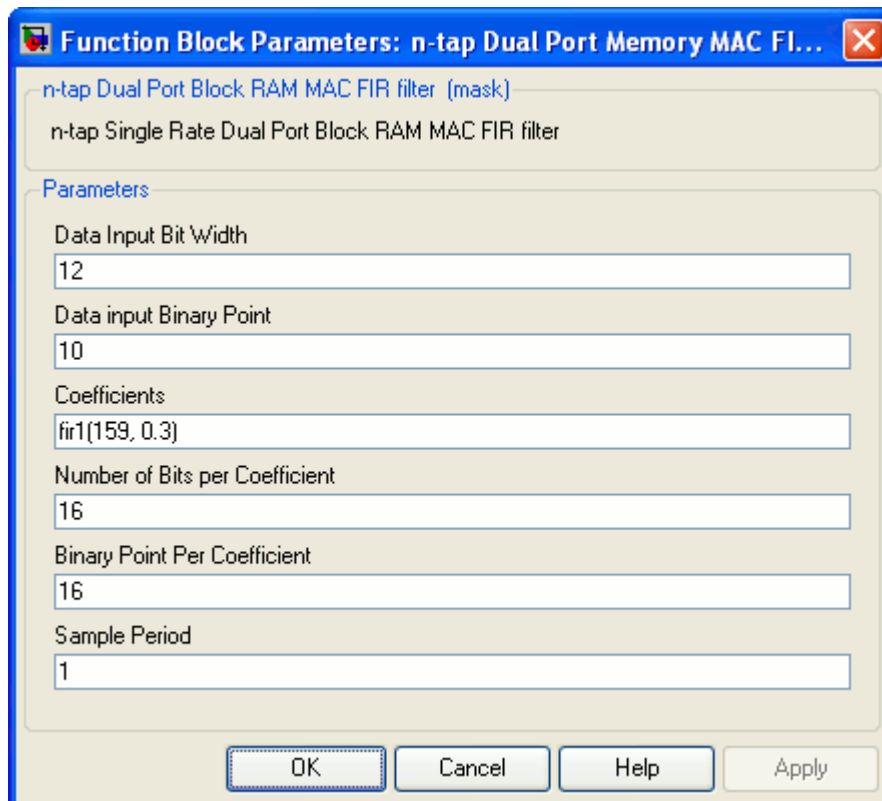


Figure 1: n-tap Dual Port Block RAM MAC FIR Filter block parameters dialog box

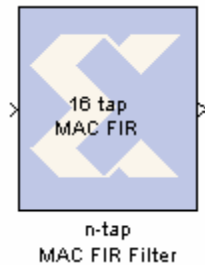
Parameters specific to this reference block are:

- **Data Input Bit Width:** Width of input sample.
- **Data Input Binary Point:** Binary point location of input.
- **Coefficients:** Specify coefficients for the filter. Number of taps is inferred from size of coefficient vector.
- **Number of Bits per Coefficient:** Bit width of each coefficient.
- **Binary Point per Coefficient:** Binary point location for each coefficient.
- **Sample Period:** Sample period of input.

Reference

J. Hwang and J. Ballagh. *Building Custom FIR Filters Using System Generator*. 12th International Field-Programmable Logic and Applications Conference (FPL). Montpellier, France, September 2002. Lecture Notes in Computer Science 2438.

n-tap MAC FIR Filter (DSP)



The Xilinx n-tap MAC FIR Filter reference block implements a multiply-accumulate based FIR filter. One, two, and four Virtex-II dedicated multipliers are used in the n-tap, the 2n-tap, and the 4n-tap filters, respectively. The three filter configurations help illustrate the tradeoffs between filter throughput and device resource consumption. The Virtex FPGA family (and Virtex family derivatives) provide dedicated circuitry for building fast, compact adders, multipliers, and flexible memory architectures. Each filter design takes advantage of these silicon features by implementing a design that is compact and resource efficient.

Annotation is provided in the filter design subsystem. Please click on the reference block and refer to this documentation seen in the block model itself for more specific implementation details.

Block Parameters Dialog Box

The block parameters dialog may be opened by double clicking on the icon in your Simulink model:

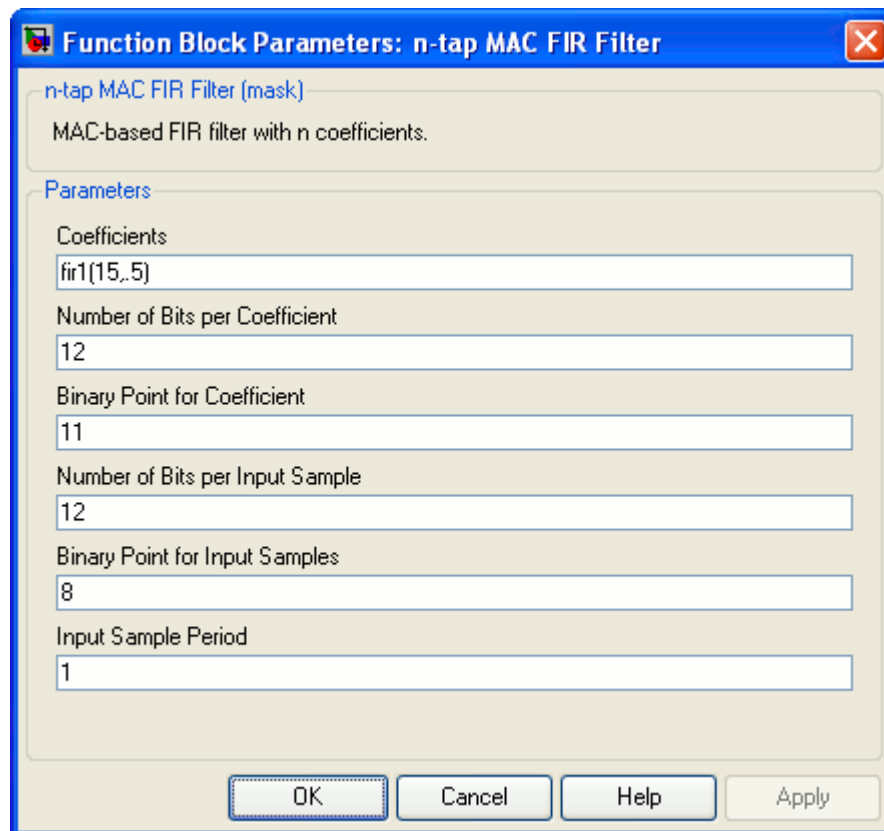


Figure 1: n-tap MAC FIR Filter block parameters dialog box

Parameters specific to this reference block are:

- **Coefficients:** Specify coefficients for the filter. Number of taps is inferred from size of coefficient vector.
- **Number of Bits per Coefficient:** Bit width of each coefficient.
- **Binary Point for Coefficient:** Binary point location for each coefficient.
- **Number of Bits per Input Sample:** Width of input sample.
- **Binary Point for Input Samples:** Binary point location of input.

- Input Sample Period: Sample period of input.

Reference

[1] J. Hwang and J. Ballagh. *Building Custom FIR Filters Using System Generator*. 12th International Field-Programmable Logic and Applications Conference (FPL). Montpellier, France, September 2002. Lecture Notes in Computer Science 2438.

Registered Mealy State Machine (Control)

A “Mealy machine” is a finite state machine whose output is a function of state transition, i.e., a function of the machine’s current state and current input. A “registered Mealy machine” is one having registered output, and can be described with the following block diagram:

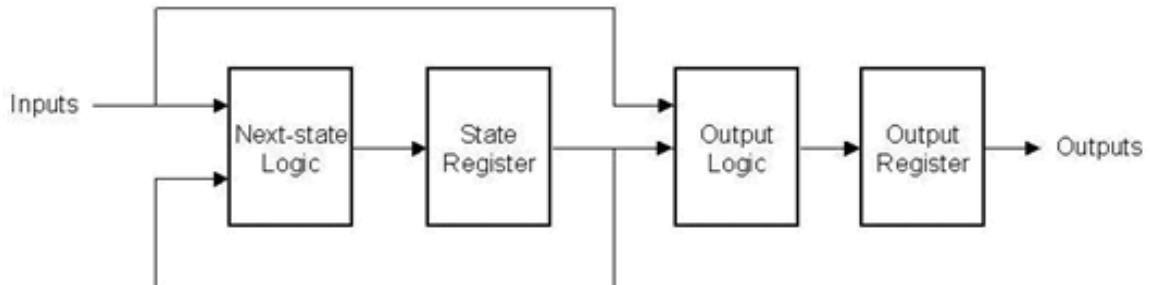
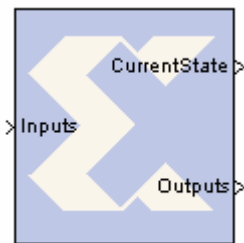


Figure 1: Registered Mealy State Machine block diagram



Registered Mealy State Machine

There are many ways to implement such state machines in System Generator (e.g., using the MCode block to implement the transition function, and registers to implement state variables). This reference block provides a method for implementing a Mealy machine using block and distributed memory. The implementation is very fast and efficient. For example, a state machine with 8 states, 1 input, and 2 outputs that are registered can be realized with a single block RAM that runs at more than 150 MHz in a Xilinx Virtex device.

The transition function and output mapping are each represented as an $N \times M$ matrix, where N is the number of states, and M is the size of the input alphabet (e.g., $M = 2$ for a binary input). It is convenient to number rows and columns from 0 to $N - 1$ and 0 to $M - 1$ respectively. Each state is represented as an unsigned integer from 0 to $N - 1$, and each alphabet character is represented as an unsigned integer from 0 to $M - 1$. The row index of each matrix represents the current state, and the column index represents the input character

For the purpose of discussion, let F be the $N \times M$ transition function matrix, and O be the $N \times M$ output function matrix. Then $F(i,j)$ is the next state when the current state is i and the current input character is j , and $O(i,j)$ is the corresponding output of the Mealy machine.

Example

Consider the problem of designing a Mealy machine to recognize the pattern '1011' in a serial stream of bits. The state transition diagram and equivalent transition table are shown below.

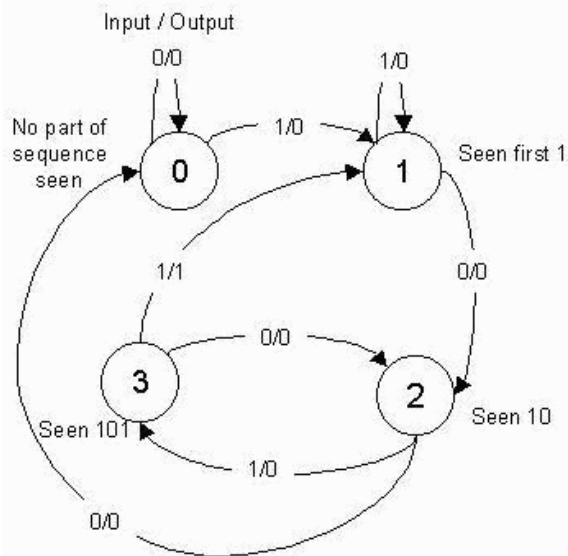


Figure 2: Example transition diagram for registered Mealy state machine diagram

Next State/Output Table

Current State	If Input = 0	If Input = 1
0	0, 0	1, 0
1	2, 0	1, 0
2	0, 0	3, 0
3	2, 0	1, 1

Cell Format: Next State, Output

Figure 3: Example transition table for registered Mealy State machine

The table lists the next state and output that result from the current state and input. For instance, if the current state is 3 and the input is 1, the next state is 1 and the output is 1, indicating the detection of the desired sequence.

The Registered Mealy State Machine block is configured with next state and output matrices obtained from the next state/output table discussed above. These matrices are constructed as follows:

Next State/Output Table

Current State	If Input = 0	If Input = 1
0	0, 0	1, 0
1	2, 0	1, 0
2	0, 0	3, 0
3	2, 0	1, 1

Cell Format: Next State, Output

$\begin{bmatrix} 0 & 1 \\ 2 & 1 \\ 0 & 3 \\ 2 & 1 \end{bmatrix}$	$\begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix}$

Next State Matrix

Output Matrix

Figure 4: Construction of next state and output matrices

Rows of the matrices correspond to states, and columns correspond to input values.

Block Parameters Dialog Box

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

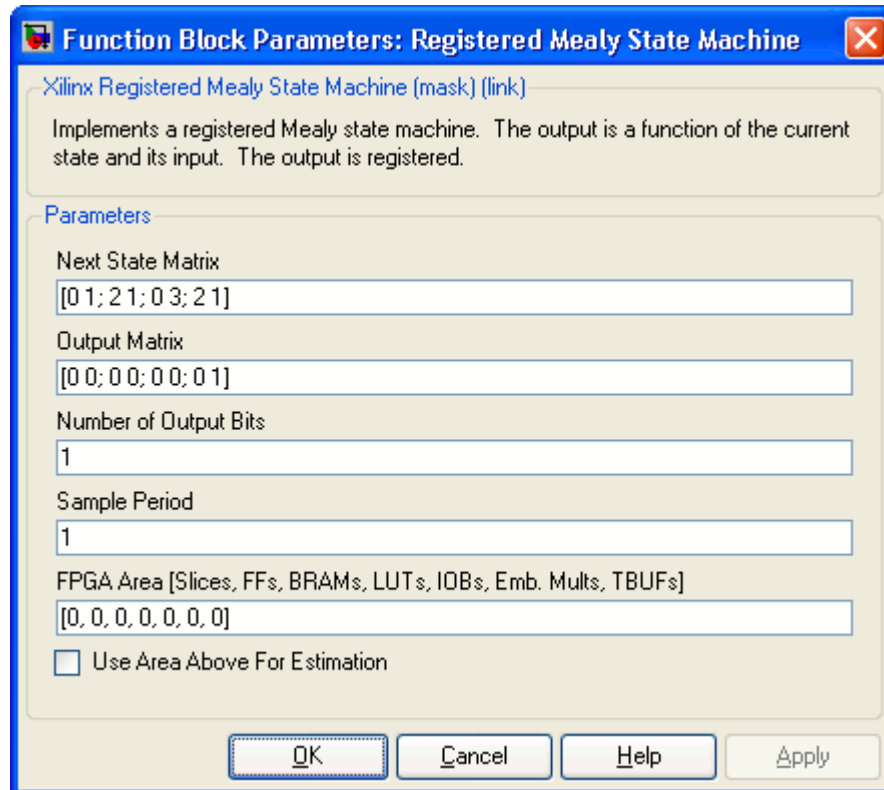


Figure 5: Registered Mealy state machine block parameters dialog box

The next state logic, state register, output logic, and output register are implemented using high speed dedicated block RAM. Of the four blocks in the state machine library, this is the fastest and most area efficient. However, the output is registered and thus the input does not affect the output instantaneously.

The number of bits used to implement a registered mealy state machine is given by the equations:

$$\begin{aligned} \text{depth} &= (2^k)(2^i) = 2^{k+i} \\ \text{width} &= k + o \\ N &= \text{depth} * \text{width} = (k + o)(2^{k+i}) \end{aligned}$$

where

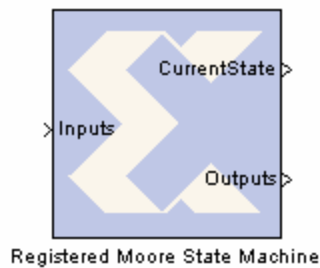
N = total number of block RAM bits
k =
s = number of states
i = number of input bits
o = number of output bits

The following table gives examples of block RAM sizes necessary for various state machines:

Number of States	Number of Input Bits	Number of Output Bits	Block RAM Bits Needed
2	5	10	704
4	1	2	32
8	6	7	5120
16	5	4	4096
32	4	3	4096
52	1	11	2176
100	4	5	24576

The block RAM width and depth limitations are described in the online help for the Single Port RAM block.

Registered Moore State Machine (Control)



A “Moore machine” is a finite state machine whose output is only a function of the machine’s current state. A “registered Moore machine” is one having registered output, and can be described with the following block diagram:

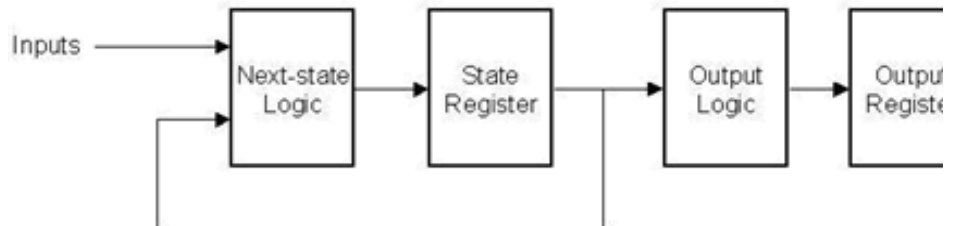


Figure 1: Registered Moore State Machine block diagram

There are many ways to implement such state machines in System Generator, e.g., using the Mcode block. This reference block provides a method for implementing a Moore machine using block and distributed memory. The implementation is very fast and efficient. For example, a state machine with 8 states, 1 input, and 2 outputs that are registered can be realized with a single block RAM that runs at more than 150 MHz in a Xilinx Virtex device.

The transition function and output mapping are each represented as an $N \times M$ matrix, where N is the number of states, and M represents the number of possible input values (e.g., $M = 2$ for a one bit input). It is convenient to number rows and columns from 0 to $N - 1$ and 0 to $M - 1$ respectively. Each state is represented as an unsigned integer from 0 to $N - 1$, and each alphabet character is represented as an unsigned integer from 0 to $M - 1$. The row index of each matrix represents the current state, and the column index represents the input character.

For the purpose of discussion, let F be the $N \times M$ transition function matrix, and O be the $N \times M$ output function matrix. Then $F(i,j)$ is the next state when the current state is i and the current input character is j , and $O(i,j)$ is the corresponding output of the Mealy machine.

Example

Consider the problem of designing a Moore machine to recognize the pattern '1011' in a serial stream of bits. The state transition diagram and equivalent transition table are shown below.

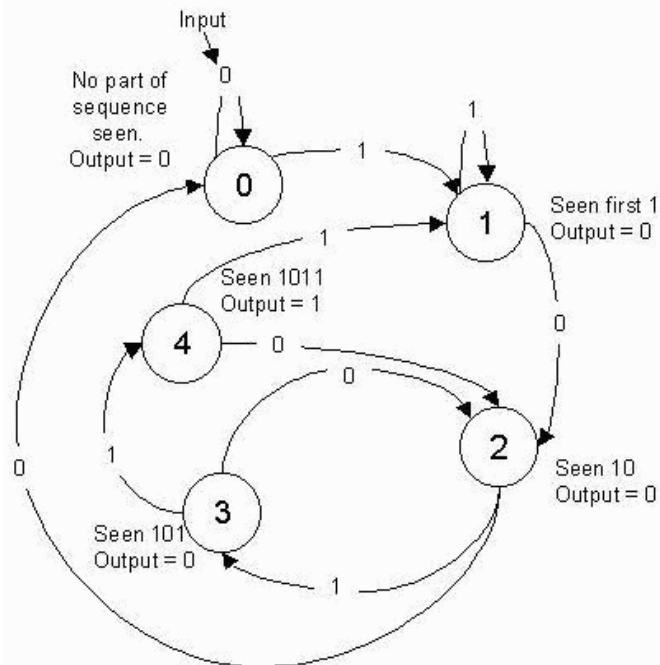


Figure 2: Registered Moore State Machine example transition diagram
Next State/Output Table

Current State	If Input = 0	If Input = 1	Output
0	0	1	0
1	2	1	0
2	0	3	0
3	2	4	0
4	2	1	1

Figure 3: Registered Moore State Machine example transition table

The table lists the next state and output that result from the current state and input. For example, if the current state is 4, the output is 1 indicating the detection of the desired sequence, and if the input is 1 the next state is state 1.

The Registered Moore State Machine block is configured with next state matrix and output array obtained from the next state/output table discussed above. They are constructed as follows:

Next State/Output Table

Current State	If Input = 0	If Input = 1	Output
0	0	1	0
1	2	1	0
2	0	3	0
3	2	4	0
4	2	1	1

$$\begin{bmatrix} 0 & 1 \\ 2 & 1 \\ 0 & 3 \\ 2 & 4 \\ 2 & 1 \end{bmatrix}$$

Next State Matrix

$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

Output Array

Figure 4: Construction of Next State and Output matrices

The rows of the matrices correspond to the current state. The next state matrix has one column for each input value. The output array has only one column since the input value does not affect the output of the state machine.

Block Parameters Dialog Box

The block parameters dialog can be invoked by double-clicking the icon in your Simulink model:

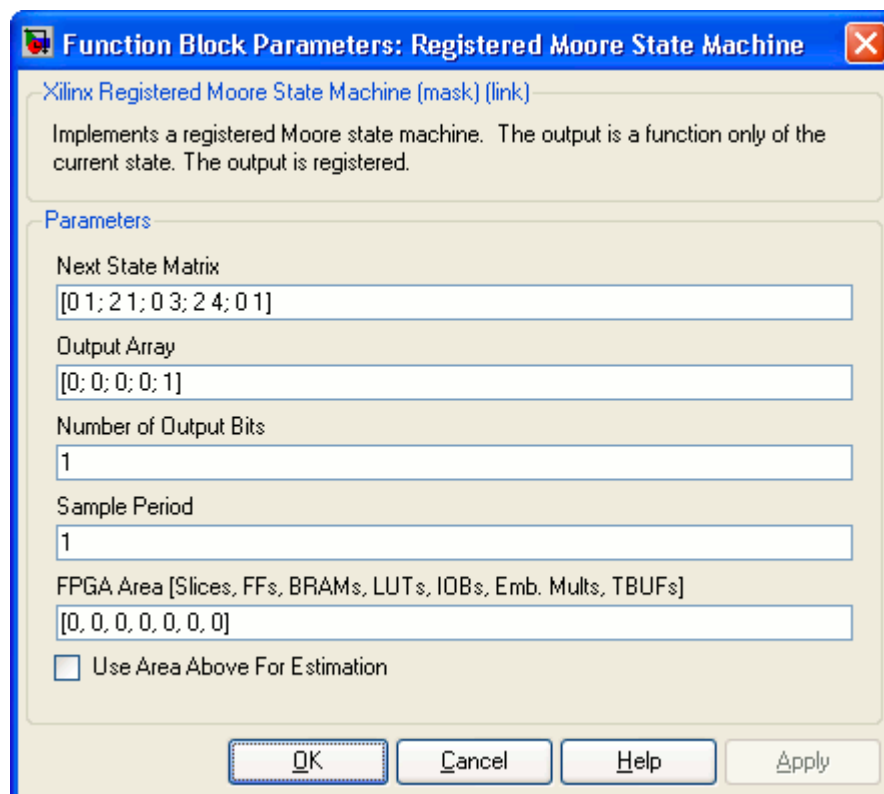


Figure 5: Registered Moore State Machine block parameters dialog box

The next state logic, state register, output logic, and output register are implemented using high speed dedicated block RAM.

The number of bits used to implement a Moore state machine is given by the equations:

$$d_s = (2^k)(2^i) = 2^{k+i}$$

$$w_s = k$$

$$N_s = d_s * w_s = (k)(2^{k+i})$$

where

N_s = total number of next state logic block RAM bits

k =

d_s = depth of state logic block RAM

w_s = width of state logic block RAM

s = number of states

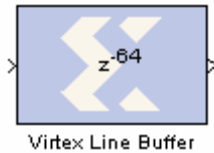
i = number of input bits

The following table gives examples of block RAM sizes necessary for various state machines:

Number of States	Number of Input Bits	Block RAM Bits Needed
2	5	64
4	1	8
8	6	1536
16	5	2048
32	4	2560
52	1	768
100	4	14336

The block RAM width and depth limitations are described in the core datasheet for the Single Port Block Memory.

Virtex Line Buffer (Imaging)



The Xilinx Virtex Line Buffer reference block delays a sequential stream of pixels by the specified buffer depth.

Block Parameters Dialog Box

The block parameters dialog may be opened by double clicking on the icon in your Simulink model:

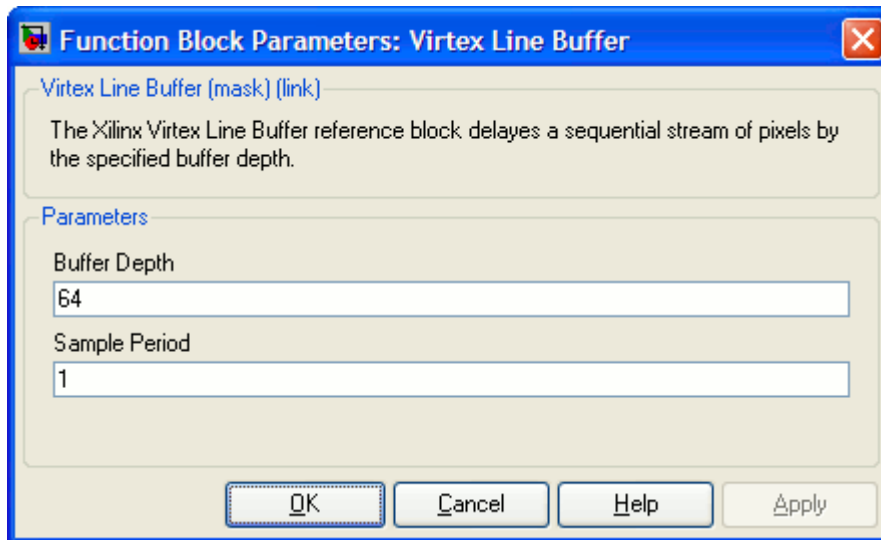


Figure 1: Virtex Line Buffer block parameters dialog box

Parameters specific to this reference block are:

- **Buffer Depth:** Number of samples the stream of pixels will be delayed.
- **Sample Period:** Sample rate at which the block will run.

Virtex2 Line Buffer (Imaging)



The Xilinx Virtex2 Line Buffer reference block delays a sequential stream of pixels by the specified buffer depth. It is optimized for the Virtex2 family since it uses the Read Before Write option on the underlying Single Port RAM block.

Block Parameters Dialog Box

The block parameters dialog may be opened by double clicking on the icon in your Simulink model:

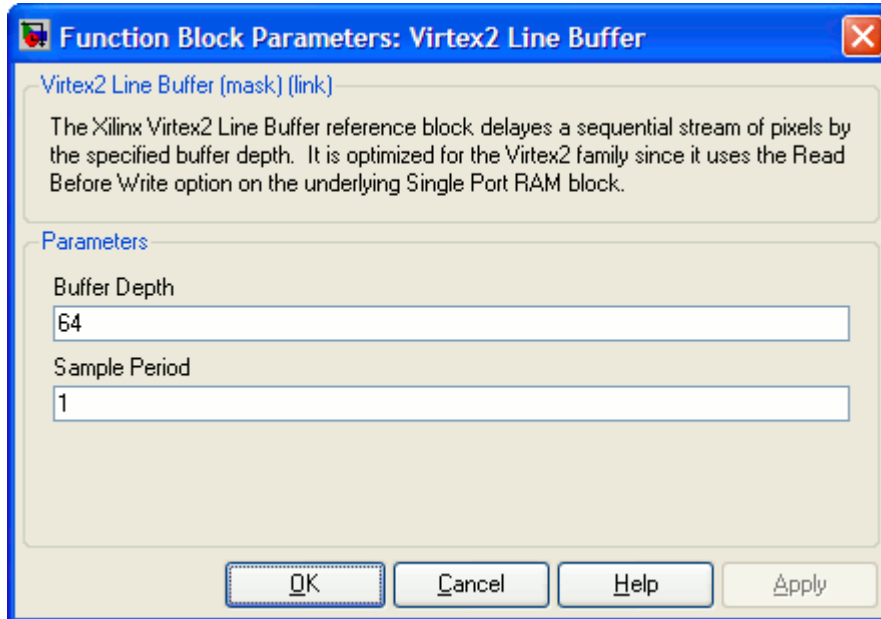
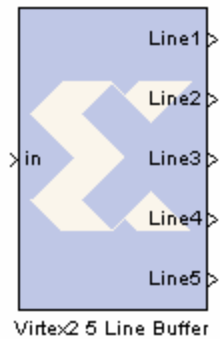


Figure 1: Virtex2 Line Buffer block parameters dialog box

Parameters specific to this reference block are:

- **Buffer Depth:** Number of samples the stream of pixels will be delayed.
- **Sample Period:** Sample rate at which the block will run.

Virtex2 5 Line Buffer (Imaging)



The Xilinx Virtex2 5 Line Buffer reference block buffers a sequential stream of pixels to construct 5 lines of output. Each line is delayed by N samples, where N is the length of the line. Line 1 is delayed $4*N$ samples, each of the following lines are delay by N fewer samples, and line 5 is a copy of the input.

This block uses Virtex2 Line Buffer block which is located in the Imaging library of the Xilinx Reference Blockset.

Block Parameters Dialog Box

The block parameters dialog may be opened by double clicking on the icon in your Simulink model:

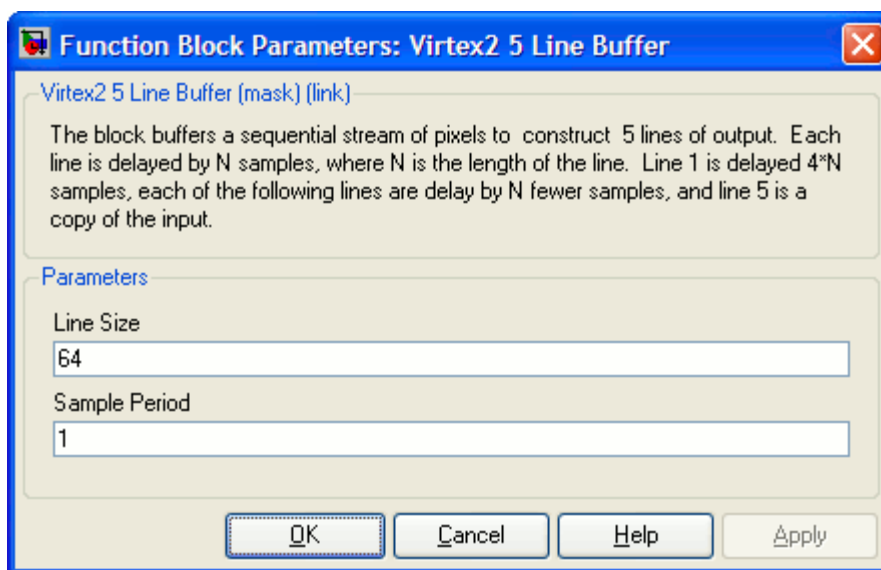


Figure 3-140: Virtex2 5 Line Buffer block parameters dialog box

Parameters specific to this reference block are:

- **Line Size** : Number of samples each line will be delayed.
- **Sample Period**: Sample rate at which the block will run.

White Gaussian Noise Generator (Communication)



The Xilinx White Gaussian Noise Generator (WGNG) generates white Gaussian noise using a combination of the Box-Muller algorithm and the Central Limit Theorem following the general approach described in [1] (reference listed below).

The Box-Muller algorithm generates a unit normal random variable via a transformation of two independent random variables that are uniformly distributed over [0,1]. This is accomplished by storing Box-Muller function values in ROMs and addressing them with uniform random variables.

The uniform random variables are produced by multiple-bit leap-forward LFSRs. A standard LFSR generates one output per clock cycle. K-bit leap-forward LFSRs are able to generate k outputs in a single cycle. For example, a 4-bit leap-forward LFSR outputs a discrete uniform random variable between 0 and 15. A portion of the 48-bit block parameter seed initializes each LFSR allowing customization. The outputs of four parallel Box-Muller subsystems are averaged to obtain a probability density function (PDF) that is Gaussian to within 0.2% out to 4.8sigma. The overall latency of the WGNG is 10 clock cycles. The output port noise is a 12 bit signed number with 7 bits after the binary point.

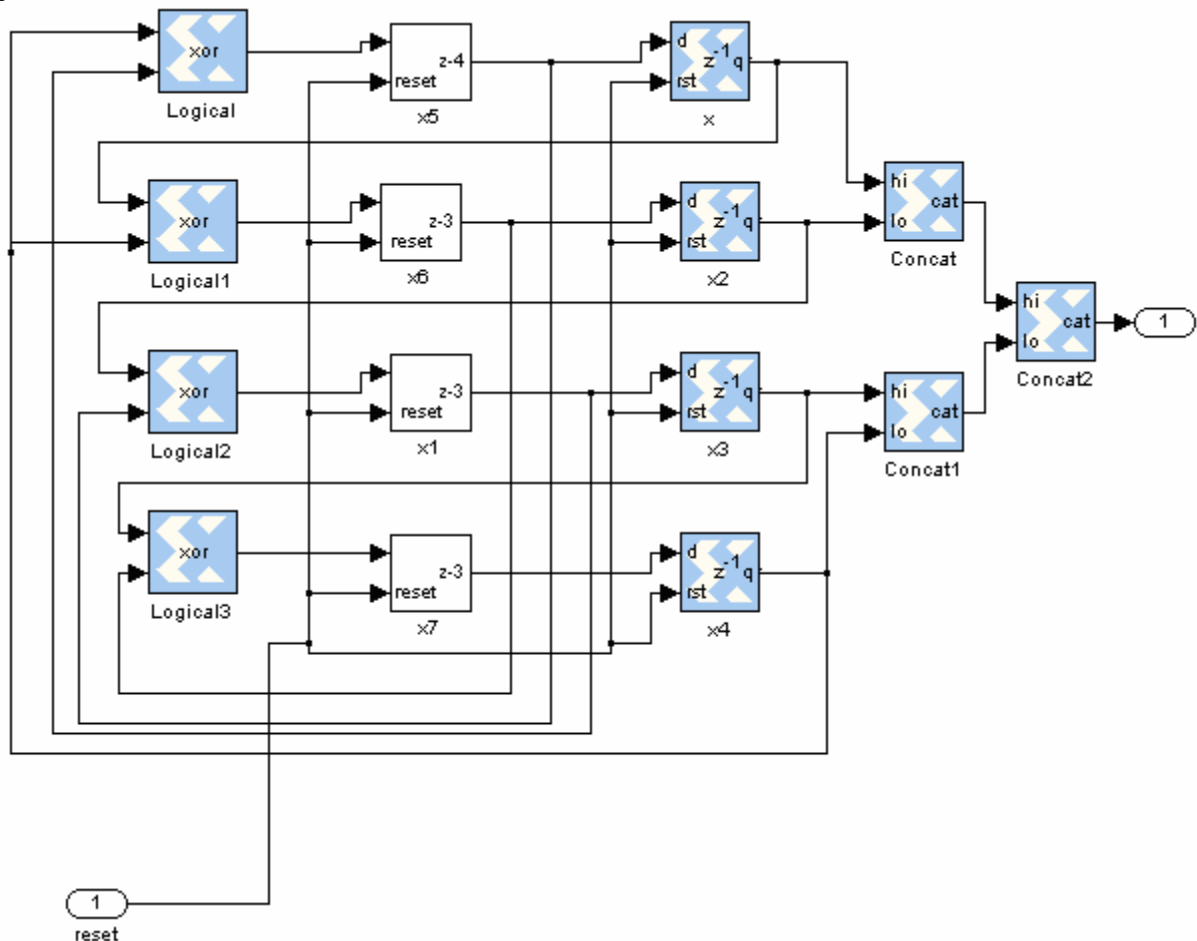


Figure 1: 4-bit Leap-Forward LFSR

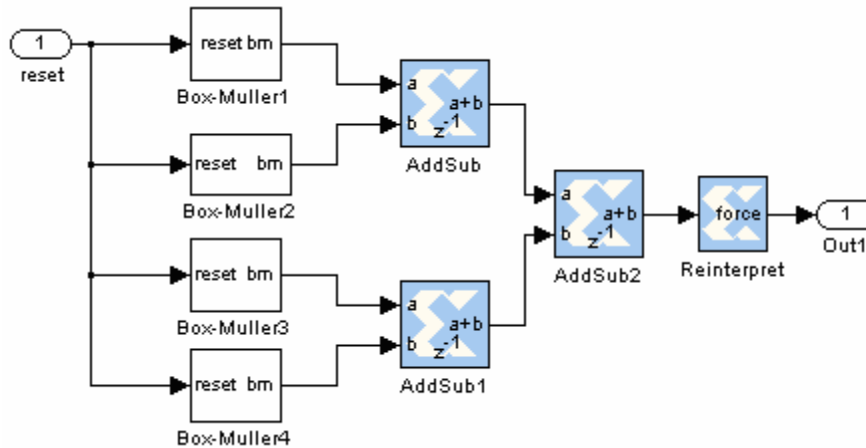


Figure 2: Box-Muller Method

Block Parameters Dialog Box

The block parameters dialog may be opened by double clicking on the icon in your Simulink model:

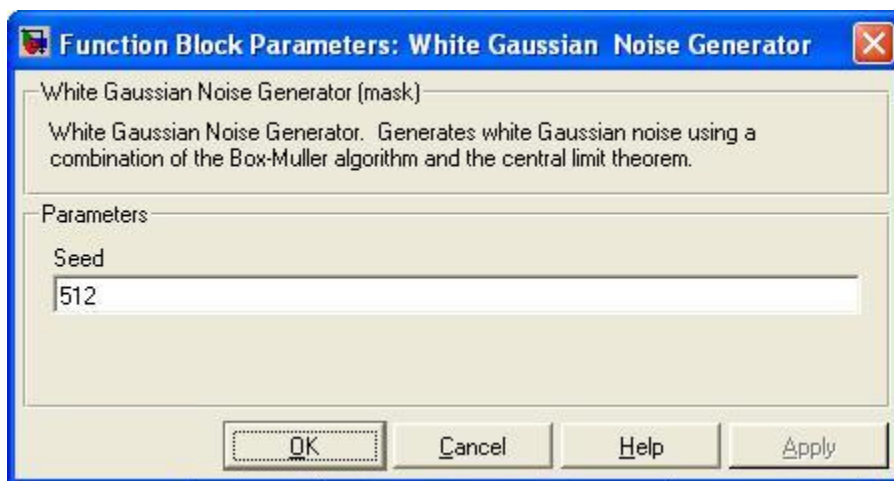


Figure 3: White Gaussian Noise Generator block parameters dialog box

The block parameter is a decimal starting seed value.

Reference

- [1] A. Ghazel, E. Boutillon, J. L. Danger, G. Gulak and H. Laamari, Design and Performance Analysis of a High Speed AWGN Communication Channel Emulator, IEEE PACRIM Conference, Victoria, B. C., Aug. 2001.

Xilinx LogiCORE Versions

Many System Generator blocks use Xilinx LogiCOREs. The following table indicates which LogiCORE version a particular block is using.

System Generator Block	Xilinx LogiCORE	LogiCORE Version / Data Sheet	Spartan 2, 2E	Spartan 3, 3E	Spartan 3A	Virtex, Virtex E	Virtex 2, Virtex 2P	Virtex 4	Virtex 5
Accumulator	Accumulator	V7.0	■	■		■	■	■	
	Accumulator	V9.1			■				■
Addressable Shift Register	RAM-based Shift Register	V7.0	■	■		■	■	■	
	RAM-based Shift Register	V9.1			■				■
AddSub	Adder Subtractor	V7.0	■	■		■	■	■	
	Adder Subtractor	V9.1			■				■
CMult	Multiplier	V7.0	■	■		■	■	■	
	Multiplier	V9.0							■
Counter	Binary Counter	V7.0	■	■		■	■	■	
	Binary Counter	V9.1			■				■
Convolutional Encoder v3_0	Convolution Encoder	V3.0	■	■		■	■		
Convolutional Encoder v6_0	Convolution Encoder	V6.0		■			■	■	■

DDS v4_1	DDS	V4.1	■	—	■	■	—	—
DDS v5_0	DDS	V5.0	■	■	■	■	■	—
Dual Port RAM	Dual Port Block Memory	V6.1	■	■	■	■	■	—
	Block Memory Generator	V2.4	—	—	■	—	—	■
	Distributed Memory	V7.1	■	■	■	■	■	—
	Distributed Memory Generator	V3.3	—	—	■	—	—	■
FIFO	Synchronous FIFO	V5.0	■	■	■	■	■	—
	FIFO Generator	V3.3	—	—	■	—	—	■
FIR Compiler v1_0	FIR Compiler	V1.0	—	—	—	—	■	—
FIR Compiler v2_0	FIR Compiler	V2.0	—	—	—	—	■	■
FFT v1_0	16 Pt Complex Fast Fourier Transform	V1.0	—	—	■	—	—	—
	64 Pt Complex Fast Fourier Transform	V1.0	—	—	■	—	—	—

	256 Pt Complex Fast Fourier Transform	V1.0		—		■		—	
	1024 Pt Complex Fast Fourier Transform	V1.0		—		■		—	
FFT v3_1	Fast Fourier Transform	V3.1		■			■	■	
FFT v3_2	Fast Fourier Transform	V3.2		■			■	■	
FFT v4_1	Fast Fourier Transform	V4.1		■	■		■	■	■
DAFIR v9_0	Distributed Arithmetic FIR Filter	V9.0	■	■		■	■	■	
Interleaver/De-Interleaver v4_0	Interleaver/Deinterleaver	V4.0	■	■		■	■	■	
Interleaver/De-Interleaver v5_0	Interleaver/Deinterleaver	V5.0	■	■		■	■	■	
Mult	Multiplier	V9.0	■	■	■	■	■	■	■
ROM	Single Port Block Memory	V6.1	■	■		■	■	■	
	Block Memory Generator	V2.4			■				■
	Distributed Memory	V7.1	■	■		■	■	■	
	Distributed	V3.3			■				■

	Memory Generator								
RS Decoder v5_1	RS Decoder	V5.1	■	■		■	■	■	
RS Decoder v6_0	RS Decoder	V6.0		■	■		■	■	■
RS Encoder v5_0	RS Encoder	V5.0	■	■		■	■	■	
RS Encoder v6_0	RS Encoder	V6.0		■			■	■	■
SineCosine	Sine Cosine	V5.0	■	■		■	■	■	
Single Port RAM	Single Port Block Memory	V6.1	■	■		■	■	■	
	Block Memory Generator	V2.4			■				■
	Distributed Memory	V7.1	■	■		■	■	■	
	Distributed Memory Generator	V3.3			■				■
Viterbi Decoder v5_0	Viterbi Decoder	V5.0	■	■		■	■	■	
Viterbi Decoder v6_0	Viterbi Decoder	V6.0		■	■		■	■	■

System Generator Utilities

xlAddTerms

xlAddTerms is similar to the addterms command in Simulink, in that it adds blocks to terminate or drive unconnected ports in a model. With xlAddTerms, output ports are terminated with a Simulink terminator block, and input ports are correctly driven with either a Simulink or System Generator constant block. Additionally System Generator gateway blocks can also be conditionally added.

The optionStruct argument can be configured to instruct xlAddTerms to set a block's property (e.g. set a constant block's value to 5) or to use different source or terminator blocks.

Syntax

```
xlAddTerms(arg1,optionStruct)
```

e.g.

```
xlAddTerms(gcs)
```

```
xlAddTerms(find_system('tt/mySubsystem','SearchDepth',1))
```

```
xlAddTerms(blklist,optionStruct)
```

Description

In the following description, 'source block' refers to the block that is used to drive an unconnected port. And 'term block' refers to the block that is used to terminate an unconnected port.

xlAddTerms(arg1,optionStruct)

xlAddTerms takes either 1 or 2 arguments. The second argument, optionStruct argument is optional. The first argument can be the name of a system, or a block list.

arg1	Description
gcs	A string-handle of the current system
'top/test1'	A string-handle of a system called test1. In this case, xlAddTerms is passed a handle to a system. This will run xlAddTerms on all the blocks under test1, including all children blocks of subsystems.
{'top/test1'}	A block list of string handles. In this case, xlAddTerms is passed a handle to a block. This will run xlAddTerms only on the block called test1, and will not process child blocks.
{'t/b1';'t/b2';'t/b3'}	A block list of string handles.
[1;2;3]	A block list of numeric handles.

The optionStruct argument is optional, but when included, should be a Matlab structure. The following table describes the possible values in the structure. The structure field names (as is true with all Matlab

structure field names) are case sensitive.

optionStruct	Description						
Source	xlAddTerms can terminate in-ports using any source block (refer to SourceWith field). The parameters of the source block can be specified using the Source field of the optionStruct by passing the parameters as sub-fields of the Source field. The Source field prompts xlAddTerms to do a series of set_params on the source block. Since it is possible to change the type of the source block, it is left to the user to ensure that the parameters here are relevant to the source block in use. E.g. when a Simulink constant block is used as a Source Block, setting the block's value to 10 can be done with: Source.value = '10' And when a System Generator Constant block is used as a Source Block, setting the constant block to have a value of 10 and of type UFIX_32_0 can be done with: <pre>Source.const = '10'; Source.arith_type='Unsigned'; Source.bin_pt=0; Source.n_bits=32;</pre>						
SourceWith	The SourceWith field allows the source block to be specified. Default is to use a constant block. SourceWith has two sub-fields which must be specified. <table border="1"> <thead> <tr> <th>Sub-fields</th><th>Description</th></tr> </thead> <tbody> <tr> <td>SourceWith.Block</td><td>A string specifying the fullpath and name of the block to be used. e.g. 'built-in/Constant' or 'xbsIndex_r3/AddSub'.</td></tr> <tr> <td>SourceWith.Port</td><td>A string specifying the port number used to connect. E.g. '1' or '3' Specifying '1' instructs xlAddTerms to connect using port 1, etc.</td></tr> </tbody> </table>	Sub-fields	Description	SourceWith.Block	A string specifying the fullpath and name of the block to be used. e.g. 'built-in/Constant' or 'xbsIndex_r3/AddSub'.	SourceWith.Port	A string specifying the port number used to connect. E.g. '1' or '3' Specifying '1' instructs xlAddTerms to connect using port 1, etc.
Sub-fields	Description						
SourceWith.Block	A string specifying the fullpath and name of the block to be used. e.g. 'built-in/Constant' or 'xbsIndex_r3/AddSub'.						
SourceWith.Port	A string specifying the port number used to connect. E.g. '1' or '3' Specifying '1' instructs xlAddTerms to connect using port 1, etc.						
TermWith	The TermWith Field allows the term block to be specified. Default is to use a Simulink terminator block. TermWith has two sub-fields which must be specified. <table border="1"> <thead> <tr> <th>Sub-fields</th><th>Description</th></tr> </thead> <tbody> <tr> <td>TermWith.Block</td><td>A string specifying the fullpath and name of the block to be used. e.g. 'built-in/Terminator' or 'xbsIndex_r3/AddSub'.</td></tr> <tr> <td>TermWith.Port</td><td>A string specifying the port number used to connect. E.g. '1' or '3' Specifying '1' instructs xlAddTerms to connect using port 1, etc.</td></tr> </tbody> </table>	Sub-fields	Description	TermWith.Block	A string specifying the fullpath and name of the block to be used. e.g. 'built-in/Terminator' or 'xbsIndex_r3/AddSub'.	TermWith.Port	A string specifying the port number used to connect. E.g. '1' or '3' Specifying '1' instructs xlAddTerms to connect using port 1, etc.
Sub-fields	Description						
TermWith.Block	A string specifying the fullpath and name of the block to be used. e.g. 'built-in/Terminator' or 'xbsIndex_r3/AddSub'.						
TermWith.Port	A string specifying the port number used to connect. E.g. '1' or '3' Specifying '1' instructs xlAddTerms to connect using port 1, etc.						
UseGatewayIns	Instructs xlAddTerms to insert System Generator gateway ins when required. The existence of the field is used to denote insertion of gateway						

	ins. This field must not be present if gateway ins are not to be used.
GatewayIn	If gateway ins are inserted, their parameters can be set using this field, in a similar way as for Source and Term. E.g. GatewayIn.arith_type='Unsigned'; GatewayIn.n_bits='32' GatewayIn.bin_pt='0' will set the gateway in to output an ufix_32_0.
UseGatewayOuts	Instructs xlAddTerms to insert System Generator gateway outs when required. The existence of the field is used to denote insertion of gateway outs. This field must not be present if gateway outs are not to be used.
GatewayOut	If gateway outs are inserted, their parameters can be set using this field, in a similar way as for Source and Term. E.g. GatewayOut.arith_type='Unsigned'; GatewayOut.n_bits='32' Gatewayout.bin_pt='0' will set the gateway out to take an input of ufix_32_0.
RecurseSubSystems	Instructs xlAddTerm to recursively run xlAddTerm under all child subsystems. Expects a scalar number, 1 or 0.

Examples

Example 1: Runs xlAddTerms on the current system, with the default parameters: constant source blocks are used, and gateways are not added. Subsystems will be recursively terminated.

```
xlAddTerms(gcs);
```

Example 2: runs xlAddTerms on all the blocks in the subsystem tt./mySubsystem.

```
xlAddTerms(find_system('tt/mySubsystem', 'SearchDepth', 1));
```

Example 3: runs xlAddTerms on the current system, setting the source block's constant value to 1, using gateway outs and changing the term block to use a Simulink display block.

```
s.Source.const = '10';
s.UseGatewayOuts = 1;
s.TermWith.Block = 'built-in/Display';
s.TermWith.Port = '1';
s.RecurseSubSystem = 1;
xlAddTerms(gcs, s);
```

Remarks

Note that field names are case sensitive. When using the fields 'Source', 'GatewayIn' and 'GatewayOut', users have to ensure that the parameter names to be set are valid.

See Also

Toolbar, xlTBUtils

xlGenerateButton

The xlGenerateButton function provides a programmatic way to invoke the System Generator code generator.

Syntax

```
status = xlGenerateButton(sysgenblock)
```

Description

xlGenerateButton invokes the System Generator code generator and returns a status code. Invoking xlGenerateButton with a System Generator block as an argument is functionally equivalent to opening the System Generator GUI for that token, and clicking on the Generate button. The list of possible status codes returned by xlGenerateButton is enumerated in the table that follows.

Status	Description
1	Canceled
2	Simulation running
3	Check param error
4	Compile/generate netlist error
5	Netlister error
6	Post netlister script error
7	Post netlist error
8	Post generation error
9	External view mismatch when importing as a configurable subsystem

See Also

xlsetparam, xlgetparam, xlgetparams, System Generator block

xlInstallPlugin

This function installs the specified System Generator hardware co-simulation plugin. Once the installer has completed, the new compilation target may be selected from the System Generator block dialog box.

Syntax

```
xlInstallPlugin(plugin)
```

```
e.g. "xlInstallPlugin('plugin.zip')"  
     "xlInstallPlugin('plugin')"
```

Description

This function accepts one parameter, `plugin`, which contains the name of the plugin file to install. The `plugin` parameter can include path information if desired, and the `.zip` extension is optional.

See Also

Hardware Co-Simulation Installation, Supporting New Platforms, `xlSBDBuilder`

xlLoadChipScopeData

The xlLoadChipScopeData function loads a ChipScope Pro .prn files, creates workspace variables and conditionally plots the results.

Syntax

```
status = xlLoadChipScopeData(filename, plotResults);
```

Description

Load the .prn file specified in filename, and plots the results if plotResults == 1. Returns a status of -1 if the file specified in filename cannot be found. Returns a status of 0 on success. Note: Only signed and unsigned decimal numbers are supported.

Examples

Example 1:

```
xlLoadChipScopeData('SineWave.prn',0);
```

See Also

ChipScope Block

xlSetNonMemMap

Sets a Gateway In or Gateway Out block to be used as a non memory mapped port when doing hardware co-simulation. This option is often used when a Gateway is intended to be routed to hardware external to the FPGA, instead of being routed to the hardware co-simulation memory map.

Syntax

```
xlSetNonMemMap(block, company, project)
```

Description

A call to xlSetNonMemMap must be made with at least three parameters. The first is the name or handle of the gateway that is to be marked as non memory mapped. The marking of a gateway as non memory mapped is predicated upon a company and project name. The second and third parameters are strings that identify the company and project names.

Examples

Example 1:

```
xlSetNonMemMap(gcbh, 'Xilinx', 'jtaghwcosim');
```

The first parameter in the example returns the handle of the block that is currently selected. That gateway is marked as non memory mapped when generating for Xilinx's JTAG hardware co-simulation.

Example 2:

```
xlSetNonMemMap(gcbh, 'Nallatech', 'xdspkit');
```

The first parameter in the example returns the handle of the block that is currently selected. That gateway is marked as non memory mapped when generating for Nallatech's xTreme DSP kit.

See Also

Using FPGA hardware in the loop, Supporting new platforms

xlSetUseHDL

This function sets the 'Use behavioral HDL' option of blocks in a model or subsystem.

Syntax

```
xlSetUseHDL(system, mode)
```

Description

The model or system specified in the parameter system will be set to either use cores or behavioral HDL, depending on the mode. Mode is a number, where 0 refers to using cores, and 1 refers to using behavioral HDL.

Examples

Example 1:

```
xlSetUseHDL(gcs, 0)
```

This call sets the currently selected system to use cores.

See Also

xlSBDBuilder

The System Generator Board Description (SBD) Builder application aids the designing of new JTAG hardware co-simulation plugins by providing a graphical user interface that prompts for relevant information about the co-simulation platform.

Syntax

```
xlSBDBuilder;
```

Description

Invoking this function opens the SBDBuilder.

See Also

Hardware Co-Simulation Installation, Supporting New Platforms, xlInstallPlugin

xLTBUtils

The xLTBUtils command provides access to several features of the Xilinx Toolbar block. This includes access to the layout, rerouting functions and to functions that return selected blocks and lines.

Syntax

```
xLTBUtils(function, args)

e.g.
xLTBUtils('ToolBar')

xLTBUtils('Layout', struct('verbose',1, 'autoroute',0))
xLTBUtils('Layout',optionStruct)

xLTBUtils('Redrawlines', struct('autoroute',0))
xLTBUtils('RedrawLines',optionStruct)

[lines,blks]=xLTBUtils('GetSelected','All')
```

Description

`xLTBUtils(function [,args])`

xLTBUtils is a collection of functions that are used by the Xilinx Toolbar block. The `function` argument specifies the name of the function to execute. Further arguments (if required) can be tagged on as supplementary arguments to the function call. Note that the `function` argument string is not case sensitive. Possible values are enumerated below and explained further in the relevant subsections.

Function	Description
'ToolBar'	Launches the Xilinx Toolbar GUI. If the GUI is already open, it will be brought to the front.
'Layout'	Runs the layout algorithm on a model to place and reroute lines on the model. Layout can be customized using the option structure that is detailed below.
'RedrawLines'	Runs the routing algorithm on a model to reroute lines on the model. RedrawLines can be customized using the option structure detailed below.
'GetSelected'	Returns Matlab Simulink handles to blocks and lines that are selected on the system in focus.

`xLTBUtils('Layout',optionStruct)`

Automatically places and routes a Simulink model. `optionStruct` is a Matlab struct data-type, that contains the parameters for Layout. The `optionStruct` argument is optional.

Layout expects circuits to be placed left to right. After placement, Layout uses Simulink to autoroute the wire connections. Simulink will route avoiding anything visible on screen, including block labels. Setting "ignore_labels" will 'allow' Simulink to route over labels – after which it is possible to

manually move the labels to a more reasonable location. Note that field names are case sensitive.

Field Names	Description [Default values]
x_pitch, y_pitch	The gaps (pitch) between block (pixels). x_pitch specifies the amount of spacing to leave between blocks horizontally, and y_pitch specifies vertical spacing. [30].
x_start, y_start	Left (x_start) and top(y_start) margin spacing (pixels). The amount of spacing to leave on the left and top of a model. [10].
autoroute	Turns on Simulink auto-routing of lines. (1 0) [1]
ignore_labels	When auto-routing lines, Simulink will attempt to auto-route around text labels. Setting ignore_labels to 1 will minimize text label size during the routing process.
sys	Name of the system to layout. [gcs]
verbose	When set to 1, a wait bar will be shown during the layout process.

```
xLTBUtils('RedrawLines',optionStruct)
```

The RedrawLines command will redraw all lines in a Simulink model. If there are lines selected, only selected lines are redrawn otherwise all lines are redrawn. If a branch is selected, the entire line will be redrawn; main trunk and all other sub-branches.

Field Names	Description [Default values]
autoroute	Turns on Simulink auto-routing of lines. (1 0) [1]
sys	Name of the system to act on. [gcs]

```
[lines,blks]=xLTBUtils('GetSelected',arg)
```

The GetSelected command returns handles to selected blocks and lines of the system in focus. The argument `arg` is optional. It should be a one of the string values described in the table below.

Arg	Description
'all'	Gets both selected lines and blocks (default).
'lines'	Gets only selected lines.
'blocks'	Gets only selected blocks.

The GetSelected command will return an array with two items, an array of a structure containing line information (lines) and an array of block handles (blks). If the 'lines' argument is used, `blks` will be an empty array; similarly when the 'blocks' argument is used, `lines` will be an empty array.

Examples

Example 1a: Performing Layouts

```
a.verbose = 1;
a.autoroute= 0;
xLTBUtils('Layout',a);
```

This will invoke the layout tool with verbose on and autoroute off.

Example 1b: Performing Layouts

```
xlTBUtils('Layout',struct('verbose',1,'autoroute',0));
```

This will also invoke the layout tool with verbose on and autoroute off.

Example 2: Redrawing lines

```
xlTBUtils('Redrawlines',struct('autoroute',0));
```

This will redraw the lines of the current system, with auto-routing off.

Example 3: Getting selected lines and blocks

```
[lines,blks]=xlTBUtils('GetSelected')
lines =
```

1x3 struct array with fields:

```
Handle
Name
Parent
SrcBlock
SrcPort
DstBlock
DstPort
Points
Branch
```

```
blks =
```

```
1.0e+003 *
3.0320
3.0480
```

This will return all selected lines and blocks in the current system. In this case, 3 lines and 2 blocks were selected. The first line handle can be accessed via the command

```
lines(1).Handle
```

```
ans =
```

```
3.0740e+003
```

The handle to the first block can be accessed via the command

```
blks(1)
```

```
ans =
```

```
3.0320e+003
```

Remarks

Note that actions performed by Layout and RedrawLines will not be in the undo stack. Save a copy of the model before performing the actions, in order to revert to the original model.

See Also

Toolbar, xlAddTerms

xlTimingAnalysis

The System Generator timing analyzer GUI is typically launched by using the Timing Analysis compilation target from the System Generator GUI in MATLAB. The xlTimingAnalysis MATLAB command is another way of launching the timing analyzer GUI. The Timing Analysis compilation target causes the tool to compile the design, run place and route, and perform other operations prior to displaying the timing analyzer GUI. By using the xlTimingAnalysis command, it is possible to launch the GUI on previously generated timing data without performing the additional operations of the compilation target.

Syntax

```
xlTimingAnalysis(target_directory);
```

Description

Calling xlTimingAnalysis with the name of a directory that contains timing data will launch the System Generate Timing Analyzer GUI.

The timing analyzer GUI will display the data that is contained in the timing.twx and name_translations data files in the specified target directory.

The target directory name may be either a relative or an absolute path name.

Example

```
>> xlTimingAnalysis('timing')
```

Where 'timing' is the name of the target directory in which a prior timing analysis was carried out.

xlVersion

It is possible to have multiple versions of System Generator installed. The MATLAB command `xlVersion` displays which versions are installed, and makes it possible to switch from one to another. Occasionally, it is necessary to restart MATLAB to make it possible to switch versions; the `xlVersion` command will instruct you to do so in these cases.

If you install System Generator 8.1 after you install 8.2, you need to install 8.2 again in order to make `xlVersion` work.

Syntax

```
xlVersion;  
xlVersion ver;  
xlVersion -add directory;
```

Description

A call to `xlVersion` with no parameters will display the current version of System Generator installed, and also all available versions.

The `ver` option specifies the version of System Generator to switch to.

The `-add` option allows a directory to be specified. The directory is expected to hold a System Generator installation. The specified instance of System Generator will be loaded as the current working System Generator installation.

Examples

Example 1:

```
>> xlVersion  
Available System Generator installations:  
  Version 8.2 in C:/MATLAB71/toolbox/xilinx/sysgen  
  Version 8.1 in C:/MATLAB71/toolbox/xilinx.8.1/sysgen  
Current version of System Generator is 8.2.
```

Example 2:

```
>> xlVersion 8.1  
Your System Generator has been switched. Please restart MATLAB.
```

Example 3:

```
>> xlVersion -add c:/matlab/toolbox/xilinx81/sysgen
```

See Also

Displaying and Changing Versions of System Generator

xlfda_denominator

The xlfda_denominator function returns the denominator of the filter object stored in the Xilinx FDATool block.

Syntax

```
[den] = xlfda_denominator(fdablk_name);
```

Description

Returns the denominator of the filter object stored in the Xilinx FDATool block named fdablk_name, or throws an error if the named block does not exist. The block name can be local (e.g. 'FDATool'), relative (e.g. '../../FDATool'), or absolute (e.g. 'untitled/foo/bar/FDATool').

See Also

xlfda_numerator, FDATool

xlfda_numerator

The `xlfda_numerator` function returns the numerator of the filter object stored in the Xilinx FDATool block.

Syntax

```
[num] = xlfda_numerator(fdablk_name);
```

Description

Returns the numerator of the filter object stored in the Xilinx FDATool block named `fdablk_name`, or throws an error if the named block does not exist. The block name can be local (e.g. 'FDATool'), relative (e.g. '.././FDATool'), or absolute (e.g. 'untitled/foo/bar/FDATool').

See Also

`xlfda_denominator`, `FDATool`

xlgetparam and xlsetparam

Used to get and set parameter values in a System Generator block. Both functions are similar to the Simulink `get_param` and `set_param` commands and should be used instead of the Simulink functions.

Syntax

```
[value1, value2, ...] = xlgetparam(sysgenblock, param1, param2, ...)  
xlsetparam(sysgenblock, param1, value1, param2, value2, ...)
```

Description

The System Generator block differs from other blocks in one significant manner; multiple sets of parameters are stored for an instance of a System Generator block. The different sets of parameters stored correspond to different compilation targets available to the System Generator block. The 'compilation' parameter is the switch used to toggle between different compilation targets stored in the System Generator block. In order to get or set parameters associated with a particular compilation type, it is necessary to first use `xlsetparam` to change the 'compilation' parameter to the correct compilation target, before getting or setting further values.

```
[value1, value2, ...] = xlgetparam(sysgenblock, param1, param2, ...)
```

The first input argument of `xlgetparam` should be a handle to the System Generator block. Subsequent arguments are taken as names of parameters. The output returned will be an array that matched the number of input parameters. If a requested parameter does not exist, the returned value of `xlgetparam` will be empty. The `xlgetparams` function can be used to get all the parameters for the current compilation target.

```
xlsetparam(sysgenblock, param1, value1, param2, value2, ...)
```

The `xlsetparam` function also takes a handle to a System Generator block as the first argument. Subsequent arguments must be provided in pairs, the first should be the parameter name and the second the parameter value.

Examples

Example 1: Changing the synthesis tool used for HDL netlist.

```
xlsetparam(sysgenblock, 'compilation', 'HDL Netlist');  
xlsetparam(sysgenblock, 'synthesis_tool', 'XST')
```

The first `xlsetparam` is used to set the compilation target to 'HDL Netlist'. The second `xlsetparam` is used to change the synthesis tool used to 'XST'.

Example 2: Getting family and part information.

```
[fam, part]=xlgetparam(sysgenblock, 'xilinxfamily', 'part')
```

```
fam =  
Virtex2  
  
part =  
xc2v1000
```

See Also

xlGenerateButton, xlgetparams

xlgetparams

The `xlgetparams` command is used to get all parameter values in a System Generator block associated with the current compilation type. The `xlgetparams` command can be used in conjunction with the `xlgetparam` and `xlsetparam` commands to change or retrieve a System Generator block's parameters.

Syntax

```
paramstruct = xlgetparams(sysgenblk);
```

Description

All the parameters available to a System Generator block can be retrieved using the `xlgetparams` command. For more information regarding the parameters, please refer to the System Generator block documentation.

```
paramstruct = xlgetparams(sysgenblock);
```

The first input argument of `xlgetparams` should be a handle to the System Generator block. The function returns a Matlab structure that lists the parameter value pairs.

Examples

Example 1:

```
params=xlsetparams(sysgenblock)

params =

    compilation: 'HDL Netlist'
 compilation_lut: [1x1 struct]
 simulink_period: '1'
   incr_netlist: 'off'
    trim_vbits: 'Everywhere in SubSystem'
      dbl_ovrd: 'According to Block Masks'
 deprecated_control: 'off'
   xilinxfamily: 'Virtex2'
          part: 'xc2v1000'
         speed: '-4'
       package: 'bg575'
 synthesis_tool: 'XST'
    directory: './netlist'
     testbench: 'off'
   sysclk_period: '100'
 core_generation: 'According to Block Masks'
   run_coregen: 'off'
    eval_field: '0'
     clock_loc: ''
 synthesis_language: 'VHDL'
```

The `compilation_lut` parameter is another structure that lists the other compilation types that are stored in this System Generator block. Using `xlsetparam` to set the compilation type will allow the parameters

associated with that compilation type to be visible to either xlgetparams or xlgetparam.

See Also

xlGenerateButton, xlgetparam and xlsetparam

Programmatic Access in System Generator

Shared Memory API

To use this API, you will need to include `SharedMemory.h` (found in `sysgen/include`) and link against `sysgen.dll` (found in `sysgen/lib`).

The following objects are defined in the Sysgen Namespace:

- * class [SharedMemory](#)
- * class [LockableSharedMemory](#)
- * class [SharedMemoryProxy](#)
- * struct [Request](#)
- * class [NamedPipeWriter](#)
- * class [NamedPipeReader](#)

Class Documentation

LockableSharedMemory Class Reference

Inherits [SharedMemory](#).

Public Types

- * typedef void(* [callback](#))(LockableSharedMemory &, void *)

Public Methods

- * [LockableSharedMemory](#) (const std::string &name, int nwords, int word_size, [creation_tag_dispatch](#))
- * [LockableSharedMemory](#) (const std::string &name, unsigned start_address=0, int nwords=[INHERIT](#), int word_size=[INHERIT](#), double timeout_sec=15.0)
- * virtual [~LockableSharedMemory](#) ()
- * virtual bool [couldBlockOnReadOrWrite](#) () const
- * virtual bool [acquireLock](#) (double timeout_sec=[NEVER](#))
- * virtual bool [acquireLock](#) ([callback](#) function, void *arg, double timeout_sec=[NEVER](#))
- * virtual bool [lockedByMe](#) () const
- * virtual void [releaseLock](#) ()
- * virtual const StdLogicVectorVector & [viewAsStdLogicVectorVector](#) () const
- * virtual StdLogicVectorVector & [viewAsStdLogicVectorVector](#) ()
- * const uint32 * [getRawDataPtr](#) () const
- * uint32 * [getRawDataPtr](#) ()

Static Public Attributes

- * const int [NEVER](#) = -1
- * const int [INHERIT](#) = -1

Member Typedef Documentation

```
typedef void(* callback)(LockableSharedMemory&, void*)
```

Constructor & Destructor Documentation

LockableSharedMemory (const std::string & *name*, int *nwords*, int *word_size*, [creation_tag_dispatch](#) *T*)

Similar to the matching base class ([SharedMemory](#)) constructor, but a shared memory with locking (mutex) semantics is created. The [LockableSharedMemory](#) class extends the [SharedMemory](#) class with [acquireLock\(\)](#) and [releaseLock\(\)](#) methods.

LockableSharedMemory (const std::string & *name*, unsigned *start_address* = 0, int *nwords* = [INHERIT](#), int *word_size* = [INHERIT](#), double *timeout_sec* = 15.0)

Similar to the matching base class ([SharedMemory](#)) constructor, but a shared memory with locking (mutex) semantics is created. The [LockableSharedMemory](#) class extends the [SharedMemory](#) class with [acquireLock\(\)](#) and [releaseLock\(\)](#) methods.

~LockableSharedMemory () [virtual]

Usage is identical to the base class ([SharedMemory](#)) destructor, except that the [LockableSharedMemory](#) destructor will release the lock if it is currently holding it.

Member Function Documentation

virtual bool couldBlockOnReadOrWrite () const [inline, virtual]

Returns:

True if a call to [read\(\)](#) or [write\(\)](#) could either block (if the *timeout_sec* parameter to the read/write call is set to NEVER), or timeout. A [SharedMemory](#) object that did not create the memory could be referencing a memory that was created, on the other side, as either as [LockableSharedMemory](#) or a [SharedMemoryProxy](#). In either of these cases, it is possible for read/write calls to block. In the case of a [SharedMemory](#) object interfacing to a [LockableSharedMemory](#), read and write operations force implicit [acquireLock](#) and [releaseLock](#) semantics.

Reimplemented from [SharedMemory](#).

bool acquireLock (double *timeout_sec* = [NEVER](#)) [virtual]

Attempt to acquire the lock.

Parameters:

timeout_sec The period, in seconds, over which the `acquireLock` operation will be attempted.

Returns:

True if the lock can be obtained within `timeout_sec` seconds, and false otherwise. If `timeout_sec` is `NEVER`, the method invocation will either return true or else never return.

`bool acquireLock (callback function, void * arg, double timeout_sec = NEVER)`
`[virtual]`

Attempt to acquire the lock, and if successful, set a callback function that other users (in-process) can use to have the lock released. User applications typically should not use this method; it is used in certain internal System Generator applications where there are multiple memory clients in a single thread that could otherwise become deadlocked.

Parameters:

function Callback function that may be invoked by another shared memory client that needs the lock.

arg void* argument that will be passed to the callback function

timeout_sec The period, in seconds, over which the `acquireLock` operation will be attempted.

Returns:

True if the lock can be obtained within `timeout_sec` seconds, and false otherwise. If `timeout_sec` is `NEVER`, the method invocation will either return true or else never return.

`bool lockedByMe () const [virtual]`

Returns:

True if the calling instance is holding the lock.

`void releaseLock () [virtual]`

Release the lock if the calling instance has it. If it does not have the lock, the call becomes a no-op.

`const Sysgen::StdLogicVectorVector & viewAsStdLogicVectorVector () const`
`[virtual]`

Returns:

A const `StdLogicVectorVector` reference whose internal data store is mapped onto the physical shared memory. This method should only be used in high-performance applications. It allows fast, but unchecked and therefore dangerous, access.

`Sysgen::StdLogicVectorVector & viewAsStdLogicVectorVector () [virtual]`

Returns:

A `StdLogicVectorVector` reference whose internal data store is mapped onto the physical shared memory. This method should only be used in high-performance applications. It allows fast, but unchecked and therefore dangerous, access.

const Sysgen::uint32 * getRawDataPtr () const

Returns:

A const raw data pointer to the internal data store of the physical shared memory. This method should only be used in high-performance applications. It allows fast, but unchecked and therefore dangerous, access.

Sysgen::uint32 * getRawDataPtr ()

Returns:

A raw data pointer to the internal data store of the physical shared memory. This method should only be used in high-performance applications. It allows fast, but unchecked and therefore dangerous, access.

Member Data Documentation

const int NEVER = -1 [static]

Used to parameterize methods with timeout settings such that they never timeout.

Reimplemented from [SharedMemory](#).

const int INHERIT = -1 [static]

Used inherit characteristics from an already created shared memory.

Reimplemented from [SharedMemory](#).

NamedPipeReader Class Reference

Public Methods

- * [NamedPipeReader](#) (const std::string &name, int nwords=[INHERIT](#), int word_size=[INHERIT](#), double timeout_sec=15.0)
- * [~NamedPipeReader](#) ()
- * void [peek](#) (StdLogicVector &value) const
- * bool [read](#) (StdLogicVector &value, double timeout_sec=[NEVER](#))
- * bool [readArray](#) (unsigned nwords, StdLogicVectorVector &buffer, double timeout_sec=[NEVER](#))
- * unsigned [getNWords](#) () const
- * unsigned [getWordSize](#) () const

- * bool [isEmpty](#) () const
- * unsigned [numAvailable](#) () const

Static Public Attributes

- * const int [NEVER](#) = -1
- * const int [INHERIT](#) = -1

Constructor & Destructor Documentation

NamedPipeReader (const std::string & name, int nwords = [INHERIT](#), int word_size = [INHERIT](#), double timeout_sec = 15.0)

This constructor creates a [NamedPipeReader](#) instance that will read data from a named pipe that was previously created by a [NamedPipeWriter](#). The name pipe is found, through the OS, via the supplied name. If the named pipe does not already exist, and does not come to exist before the timeout expires, a Sysgen::Error is thrown.

Parameters:

name The name used by the [NamedPipeWriter](#) to create the pipe.

nwords Can be smaller than the depth of the pipe created by the [NamedPipeWriter](#), but if it is larger, a Sysgen::Error will be thrown. Defaults to INHERIT.

word_size Number of bits per word. Must match the word size specified by the [NamedPipeWriter](#), or a Sysgen::Error will be thrown. Defaults to INHERIT.

timeout_sec The period, in seconds, for which the constructor will wait for the named pipe to be made available through the OS. Defaults to 15 seconds. Can be set to NEVER.

~NamedPipeReader ()

Member Function Documentation

void peek (StdLogicVector & value) const

See also [read\(\)](#).

Retrieves the value that is sitting at the end of the pipe, ie. the same value that would be retrieved through a [read\(\)](#) invocation, but without changing the state of the pipe. The word seen by [peek\(\)](#) is not removed from the pipe. Because [peek\(\)](#), unlike [read\(\)](#), does not change the state of the pipe, there is no implied mutex requirement, and the operation will always succeed if the pipe is not empty.

If the pipe is empty a Sysgen::Error exception will be thrown.

Parameters:

value reference to a StdLogicVector whose contents will be overwritten by the value from the pipe. The StdLogicVector must have been constructed by the caller to have the appropriate type and size.

bool read (StdLogicVector & value, double timeout_sec = [NEVER](#))

See also [peek\(\)](#), [readArray\(\)](#).

If the pipe is empty a Sysgen::Error exception will be thrown.

Parameters:

value reference to a StdLogicVector whose contents will be overwritten by the value read from the pipe. The StdLogicVector must have been constructed by the caller to have the appropriate type and size.

timeout_sec The period, in seconds, over which the read operation will be attempted. There is an implicit mutex between [NamedPipeWriter](#) and [NamedPipeReader](#) access to a particular pipe.

Returns:

True if the read is successful. If timeout_sec is set to NEVER, then the read method will either return true or never return. If the read method returns false, the operation timed out.

bool readArray (unsigned nwords, StdLogicVectorVector & buffer, double timeout_sec = NEVER)

See also [read\(\)](#).

If the pipe does not contain sufficient (nwords) words available for, reading, a Sysgen::Error exception will be thrown. The caller should check that nwords < [numAvailable\(\)](#).

Parameters:

nwords The number of words to be written.

buffer Reference to a StdLogicVectorVector whose contents will be copied into the pipe. The StdLogicVectorVector must have been constructed by the caller to have the appropriate type, number of words (equaling or exceeding nwords), and number of bits per word.

timeout_sec The period, in seconds, over which the read operation will be attempted. There is an implicit mutex between [NamedPipeWriter](#) and [NamedPipeReader](#) access to a particular pipe.

Returns:

True if the read is successful. If timeout_sec is set to NEVER, then the readArray method will either return true or never return. If the readArray method returns false, the operation timed out.

unsigned getNWords () const

Returns:

The number of words that the Pipe can hold (not the number currently held). See also [numAvailable\(\)](#).

unsigned getWordSize () const

Returns:

The number of bits per word for the data conveyed by the pipe. For a particular NamedPipe instance, this value will be constant, i.e., fixed at the time of construction.

bool isEmpty () const [inline]

unsigned numAvailable () const

Returns:

The number of words that are in the pipe and are available for reading. See also [getNWords\(\)](#).

Member Data Documentation

`const int NEVER = -1 [static]`

`const int INHERIT = -1 [static]`

NamedPipeWriter Class Reference

Public Methods

- * [NamedPipeWriter](#) (const std::string &name, int nwords, int word_size)
- * [~NamedPipeWriter](#) ()
- * bool [write](#) (const StdLogicVector &value, double timeout_sec=[NEVER](#))
- * bool [writeArray](#) (unsigned nwords, const StdLogicVectorVector &buffer, double timeout_sec=[NEVER](#))
- * unsigned [getNWords](#) () const
- * unsigned [getWordSize](#) () const
- * bool [isFull](#) () const
- * unsigned [numAvailable](#) () const

Static Public Attributes

- * const int [NEVER](#) = -1

Constructor & Destructor Documentation

NamedPipeWriter (const std::string & name, int nwords, int word_size)

This constructor creates the physical memory (shared through the OS) that underlies the named pipe object. The caller must specify the number of words that the pipe can hold as well as the number of bits per word.

A named pipe can have only one writer. Nothing prevents it from having more than one reader, or from having readers that come and go.

Parameters:

name The name by which the shared named pipe published to the operating system, and with which other threads can discover it.

nwords number of words that the pipe will hold

word_size number of bits per word

~NamedPipeWriter ()

Releases the instance's handle to the shared OS resource that represents the named pipe. This is a reference counted resource; it may continue to persist after the [NamedPipeWriter](#) that established it is destroyed. [NamedPipeReader](#) instances that are using the same resource can continue to access it and read out data (until it is empty; there will be no way for new data to be added to the pipe).

Member Function Documentation

bool write (const StdLogicVector & *value*, double *timeout_sec* = [NEVER](#))

See also [writeArray\(\)](#).

If the pipe is full a Sysgen::Error exception will be thrown.

Parameters:

value reference to a StdLogicVector whose contents will be copied into the named pipe storage. The StdLogicVector must have been constructed by the caller to have the appropriate number of bits to match the pipe.

timeout_sec The period, in seconds, over which the write operation will be attempted. There is an implicit mutex between [NamedPipeWriter](#) and [NamedPipeReader](#) access to a particular pipe.

Returns:

True if the write is successful. If *timeout_sec* is set to NEVER, then the write method will either return true or never return. If the write method returns false, the operation timed out.

bool writeArray (unsigned *nwords*, const StdLogicVectorVector & *buffer*, double *timeout_sec* = [NEVER](#))

See also [write\(\)](#).

If the pipe does not contain sufficient (*nwords*) space, a Sysgen::Error exception will be thrown.

Parameters:

nwords The number of words to be written.

buffer Reference to a StdLogicVectorVector whose contents will be moved into the named pipe. The StdLogicVectorVector must have been constructed by the caller to have the appropriate type, number of words (equaling or exceeding *nwords*), and number of bits per word.

timeout_sec The period, in seconds, over which the write operation will be attempted. There is an implicit mutex between [NamedPipeWriter](#) and [NamedPipeReader](#) access to a particular pipe.

Returns:

True if the write is successful. If *timeout_sec* is set to NEVER, then the write method will either return true or never return. If the write method returns false, the operation timed out.

unsigned getNWords () const

Returns:

The number of words that the Pipe can hold (not the number currently held). See also [numAvailable\(\)](#).

unsigned getWordSize () const

Returns:

The number of bits per word for the data conveyed by the pipe. For a particular NamedPipe instance, this value will be constant, i.e., fixed at the time of construction.

bool isFull () const [inline]

unsigned numAvailable () const

Returns:**Member Data Documentation**

const int NEVER = -1 [static]

SharedMemory Class Reference

Inherited by [LockableSharedMemory](#), and [SharedMemoryProxy](#).

Public Types

- * enum [creation_tag_dispatch](#) { [creation_tag](#) }
- * enum [owner_type](#) { [base](#), [lockable](#), [proxy](#) }

Public Methods

- * [SharedMemory](#) (const std::string &name, int nwords, int word_size, [creation_tag_dispatch](#))
- * [SharedMemory](#) (const std::string &name, unsigned start_address=0, int nwords=[INHERIT](#), int word_size=[INHERIT](#), double timeout_sec=[NEVER](#))
- * virtual [~SharedMemory](#) ()
- * std::string [getName](#) () const
- * unsigned [getNWords](#) () const
- * unsigned [getWordSize](#) () const
- * [owner_type](#) [getOwnerType](#) () const
- * virtual bool [couldBlockOnReadOrWrite](#) () const
- * virtual bool [read](#) (unsigned addr, StdLogicVector &value, double timeout_sec=[NEVER](#)) const
- * virtual bool [write](#) (unsigned addr, const StdLogicVector &value, double timeout_sec=[NEVER](#))
- * virtual bool [readArray](#) (unsigned addr, unsigned nwords, StdLogicVectorVector &buffer, double timeout_sec=[NEVER](#)) const
- * virtual bool [writeArray](#) (unsigned addr, unsigned nwords, const StdLogicVectorVector &buffer, double timeout_sec=[NEVER](#))

Static Public Attributes

- * const int [NEVER](#) = -1
- * const int [INHERIT](#) = -1

Protected Types

- * enum [protected_constructor_tag_dispatch](#) { [protected_constructor_tag](#) }

Protected Methods

- * [SharedMemory](#) (const std::string &name, int nwords, int word_size, [protected_constructor_tag_dispatch](#))
- * [SharedMemory](#) ()

Protected Attributes

- * SharedMemoryImpl * [_impl](#)

Member Enumeration Documentation

enum creation_tag_dispatch

Enumeration values:

creation_tag Used exclusively to distinguish the constructor that creates the physical memory from the constructor that accesses an already extant physical memory

enum owner_type

Enumeration values:

base Physical memory created as [SharedMemory](#)

lockable Physical memory created as [LockableSharedMemory](#)

proxy Physical memory created as [SharedMemoryProxy](#)

enum protected_constructor_tag_dispatch [protected]

Enumeration values:

protected_constructor_tag

Constructor & Destructor Documentation

SharedMemory (const std::string & name, int nwords, int word_size, [creation_tag dispatch T](#))

This tag-dispatched constructor creates the physical memory (shared by the OS) that underlies the object. The caller must specify the number of words that the memory will store as well as the number of bits per word. The final argument to the constructor should be the enumerated constant [SharedMemory::creation_tag](#).

Parameters:

name The name by which the shared memory is published to the operating system, and with which other threads can discover the created memory.
nwords number of words that the memory will store
word_size number of bits per word
T can only hold the value [SharedMemory::creation_tag](#); this parameter is used to make it clear to the compiler that this constructor is desired and not the constructor which finds and existing Shared Memory with the specified name

SharedMemory (const std::string & name, unsigned start_address = 0, int nwords = [INHERIT](#), int word_size = [INHERIT](#), double timeout_sec = [NEVER](#))

This constructor creates a [SharedMemory](#) instance that utilizes an already created physical memory store. The existing memory is found, through the OS, via the supplied name. If the named memory does not already exist, and does not come to exist before the timeout expires, a Sysgen::Error is thrown.

Parameters:

name The name by which the shared memory was published to the operating system.
start_address An offset into the address space of the physical memory (defaults to zero).
nwords The size of the imaged memory. Could be smaller than the physical memory's size, but if it is larger (or extends beyond the end of the physical memory if start_address is set), a Sysgen::Error will be thrown. Defaults to INHERIT, in which case the imaged memory will extend to the end of the physical memory.
word_size Number of bits per word. Must match the physical memory, or a Sysgen::Error will be thrown. Defaults to INHERIT.
timeout_sec The period, in seconds, for which the constructor will wait for the physical shared memory to be made available through the OS. Defaults to 15 seconds. Can be set to NEVER.

~SharedMemory () [[virtual](#)]

The destruction of a [SharedMemory](#) object releases its handle to the physical memory that is being shared across the OS. The physical memory is a reference counted resource; it is freed when all handles to the resource are released. Thus if a memory is created by one thread and then accessed by a second thread, the second thread can continue to access the memory store even after the creating thread destroys the object that created and initialized the physical memory.

SharedMemory (const std::string & name, int nwords, int word_size, [protected_constructor_tag dispatch T](#)) [[protected](#)]

A protected constructor used by the derived classes -- not part of the public class API.

Parameters:

nwords number of words that the memory will store

word_size number of bits per word

T can only hold the value [SharedMemory::protected_constructor_tag](#). This parameter is used to make it clear to the compiler that this constructor is desired and not a public constructor.

SharedMemory () [protected]

The default constructor creates a [SharedMemory](#) with no underlying implementation (the `_impl` pointer is NULL). As such, this constructor is declared as private and used only by derived classes which need to establish the implementation of their parent.

Member Function Documentation

std::string getName () const

Returns:

The name that was used to create the memory, which is the name that other [SharedMemory](#) instances can use to attach to the same memory.

unsigned getNWords () const

Returns:

The number of words that can be stored in the memory. The memory can be therefore be indexed with addresses 0 through [getNWords\(\)](#)-1. For a particular [SharedMemory](#) instance, this value will be constant, i.e., fixed at the time of construction.

unsigned getWordSize () const

Returns:

The number of bits per word for the memory. For a particular [SharedMemory](#) instance, this value will be constant, i.e., fixed at the time of construction.

[Sysgen::SharedMemory::owner_type](#) getOwnerType () const

Returns:

One of the enumerated constant values:

- * [SharedMemory::base](#) if the physical memory was created through the base [SharedMemory](#) constructor.
- * [SharedMemory::lockable](#) if the physical memory was created through the derived [LockableSharedMemory](#) constructor.
- * [SharedMemory::proxy](#) if the physical memory was created through the derived [SharedMemoryProxy](#) constructor.

bool couldBlockOnReadOrWrite () const [virtual]

Returns:

True if a call to [read\(\)](#) or [write\(\)](#) could either block (if the `timeout_sec` parameter to the read/write call is set to NEVER), or timeout. A [SharedMemory](#) object that did not create the memory could be referencing a memory that was created, on the other side, as either as [LockableSharedMemory](#) or a [SharedMemoryProxy](#). In either of these cases, it is possible for read/write calls to block. In the case of a [SharedMemory](#) object interfacing to a [LockableSharedMemory](#), read and write operations force implicit `acquireLock` and `releaseLock` semantics.

Reimplemented in [LockableSharedMemory](#), and [SharedMemoryProxy](#).

bool read (unsigned *addr*, StdLogicVector & *value*, double *timeout_sec* = NEVER) const [virtual]

See also [couldBlockOnReadOrWrite\(\)](#), [readArray\(\)](#).

Parameters:

addr The address to be read. Must be in the range `[0, getNWords\(\)-1]`, or a `Sysgen::Error` exception will be thrown.
value reference to a `StdLogicVector` whose contents will be overwritten by the value read from memory. The `StdLogicVector` must have been constructed by the caller to have the appropriate type and size.
timeout_sec The period, in seconds, over which the read operation will be attempted.

Returns:

True if the read is successful. If `timeout_sec` is set to NEVER, then the read method will either return true or never return. If the read method returns false, the operation timed out.

bool write (unsigned *addr*, const StdLogicVector & *value*, double *timeout_sec* = NEVER) [virtual]

See also [couldBlockOnReadOrWrite\(\)](#), [writeArray\(\)](#).

Parameters:

addr The address to be written. Must be in the range `[0, getNWords\(\)-1]`, or a `Sysgen::Error` exception will be thrown.
value reference to a `StdLogicVector` whose contents will be copied into the physical, shared memory. The `StdLogicVector` must have been constructed by the caller to have the appropriate number of bits to match the memory.
timeout_sec The period, in seconds, over which the write operation will be attempted.

Returns:

True if the write is successful. If `timeout_sec` is set to NEVER, then the write method will either return true or never return. If the write method returns false, the operation timed out.

bool readArray (unsigned *addr*, unsigned *nwords*, StdLogicVectorVector & *buffer*, double *timeout_sec* = NEVER) const [virtual]

See also [couldBlockOnReadOrWrite\(\)](#), [read\(\)](#).

Parameters:

addr The first address to be read. Must be in the range `[0, getNWords\(\)-1]`, or a `Sysgen::Error` exception will be thrown.

nwords The number of words to be read.

buffer Reference to a StdLogicVectorVector whose contents will be overwritten by the values read from memory. The StdLogicVectorVector must have been constructed by the caller to have the appropriate type, number of words (equaling or exceeding nwords), and number of bits per word.

If $\text{addr} + \text{nwords} > \text{getNWords}()$, a Sysgen::Error exception will be thrown.

timeout_sec The period, in seconds, over which the readArray operation will be attempted.

Returns:

True if the read is successful. If timeout_sec is set to NEVER, then the readArray method will either return true or never return. If the readArray method returns false, the operation timed out.

bool writeArray (unsigned addr, unsigned nwords, const StdLogicVectorVector & buffer, double timeout_sec = NEVER) [virtual]

See also [couldBlockOnReadOrWrite\(\)](#), [write\(\)](#).

Parameters:

addr The first address to be written. Must be in the range $[0, \text{getNWords}() - 1]$, or a Sysgen::Error exception will be thrown.

nwords The number of words to be written.

If $\text{addr} + \text{nwords} > \text{getNWords}()$, a Sysgen::Error exception will be thrown.

buffer Reference to a StdLogicVectorVector whose contents will be moved into the physical, shared memory. The StdLogicVectorVector must have been constructed by the caller to have the appropriate type, number of words (equaling or exceeding nwords), and number of bits per word.

timeout_sec The period, in seconds, over which the writeArray operation will be attempted.

Returns:

True if the write is successful. If timeout_sec is set to NEVER, then the writeArray method will either return true or never return. If the writeArray method returns false, the operation timed out.

Member Data Documentation

const int NEVER = -1 [static]

Used to parameterize methods with timeout settings such that they never timeout.

Reimplemented in [LockableSharedMemory](#), and [SharedMemoryProxy](#).

const int INHERIT = -1 [static]

Used inherit characteristics from an already created shared memory.

Reimplemented in [LockableSharedMemory](#).

SharedMemoryImpl* _impl [protected]

SharedMemoryProxy Class Reference

Inherits [SharedMemory](#).

Public Types

- * typedef void(* [requestServicer](#))(const [Request](#) &, SharedMemoryProxy &, void *arg)

Public Methods

- * [SharedMemoryProxy](#) (const std::string &name, int nwords, int word_size, [requestServicer](#) rs, void *rs_arg=NULL)
- * [~SharedMemoryProxy](#) ()
- * virtual bool [couldBlockOnReadOrWrite](#) () const
- * void [service](#) ()
- * virtual const StdLogicVectorVector & [viewAsStdLogicVectorVector](#) () const
- * virtual StdLogicVectorVector & [viewAsStdLogicVectorVector](#) ()
- * const uint32 * [getRawDataPtr](#) () const
- * uint32 * [getRawDataPtr](#) ()

Static Public Attributes

- * const int [NEVER](#) = -1

Member Typedef Documentation

typedef void(* requestServicer)(const [Request](#)&, SharedMemoryProxy&, void *arg)

A function pointer, of the type declared by this typedef, is passed to the constructor of the [SharedMemoryProxy](#) constructor. See the constructor documentation for details.

Constructor & Destructor Documentation

[SharedMemoryProxy](#) (const std::string & name, int nwords, int word_size, [requestServicer](#) rs, void * rs_arg = NULL)

This constructor creates the physical memory (shared by the OS) that underlies the object. The caller must specify the number of words that the memory will store as well as the number of bits per word. A [SharedMemoryProxy](#) creates a physical memory that allows other clients to request service (read / write), but not to directly access the stored data. The clients can access the memory through the base class [SharedMemory](#) object. The service requests are passed off to the [SharedMemoryProxy](#) object that created the physical memory. This construction allows the [SharedMemoryProxy](#) to marshall data off to remote storage (e.g., the actual stored data may be marshalled off to a hardware platform or to a remote machine).

The function pointer must point to a function that will service the requests to the [SharedMemoryProxy](#) (made by other memory clients). These requests can take on the form of read_request's or write_request's as encoded by the passed [SharedMemoryProxy::Request](#) object. This object will also contain the number of words to be read / written and the start address. A reference to the [SharedMemoryProxy](#) is also passed to the servicer function, along with the void pointer used in the constructor.

Parameters:

name The name by which the shared memory is published to the operating system, and with which other threads can discover the created memory.
nwords number of words that the memory will store
word_size number of bits per word
rs The callback function that will service memory requests
rs_arg A void* argument that will be passed along the the requestServicer callback

Returns:

~SharedMemoryProxy ()

Releases the instance's handle to the shared OS resource (through which service requests are made). This is a reference counted resource; it may continue to persist after the [SharedMemoryProxy](#) that established it is destroyed. This means that when the [SharedMemoryProxy](#) destructor is called any remaining clients of the memory will run into trouble -- in particular any read / write calls that they make will go unserved (and either hang or timeout).

Member Function Documentation

virtual bool couldBlockOnReadOrWrite () const [inline, virtual]

Returns:

True if a call to [read\(\)](#) or [write\(\)](#) could either block (if the timeout_sec parameter to the read/write call is set to NEVER), or timeout. A [SharedMemory](#) object that did not create the memory could be referencing a memory that was created, on the other side, as either as [LockableSharedMemory](#) or a [SharedMemoryProxy](#). In either of these cases, it is possible for read/write calls to block. In the case of a [SharedMemory](#) object interfacing to a [LockableSharedMemory](#), read and write operations force implicit acquireLock and releaseLock semantics.

Reimplemented from [SharedMemory](#).

void service ()

Will check to see if a client has made a service request, and if it has, the requestServicer callback (established by the [SharedMemoryProxy](#) constructor) will be called.

const Sysgen::StdLogicVectorVector & viewAsStdLogicVectorVector () const [virtual]

Returns:

A const StdLogicVectorVector reference whose internal data store is mapped onto the physical shared memory. This method should only be used in high-performance applications. It allows fast, but unchecked and therefore dangerous, access.

Sysgen::StdLogicVectorVector & viewAsStdLogicVectorVector () [virtual]

Returns:

A StdLogicVectorVector reference whose internal data store is mapped onto the physical shared memory. This method should only be used in high-performance applications. It allows fast, but unchecked and therefore dangerous, access.

const Sysgen::uint32 * getRawDataPtr () const

Returns:

A const raw data pointer to the internal data store of the physical shared memory. This method should only be used in high-performance applications. It allows fast, but unchecked and therefore dangerous, access.

Sysgen::uint32 * getRawDataPtr ()

Returns:

A raw data pointer to the internal data store of the physical shared memory. This method should only be used in high-performance applications. It allows fast, but unchecked and therefore dangerous, access.

Member Data Documentation

const int NEVER = -1 [static]

Used to parameterize methods with timeout settings such that they never timeout.

Reimplemented from [SharedMemory](#).

Request Struct Reference

Public Types

- * enum [Type](#) { [read_request](#), [write_request](#) }

Public Attributes

- * enum [Sysgen::SharedMemoryProxy::Request::Type type](#)
- * unsigned [start_address](#)
- * unsigned [nwords](#)

Used to encode information passed to [SharedMemoryProxy::requestServicer](#) callback functions. For details, see the [SharedMemoryProxy](#) constructor documentation.

Member Enumeration Documentation

enum Type

Enumeration values:

read_request client wants to read the memory

write_request client wants to write to the memory

Member Data Documentation

enum [Sysgen::SharedMemoryProxy::Request::Type](#) type

unsigned start_address

first address to read / write

unsigned nwords

number of words to read / write